Foreword

Welcome to Tonc, a guide to Game Boy Advance programming originally published in 2004.

You are reading a new, revamped version of it, powered by mdbook and now maintained by the community.

> **WORK IN PROGRESS NOTICE**
>
> We recently finished the porting of this book. Improvements ongoing, if you spot any problems please feel free to ask for help or get involved!

Contributing

This book is open source, released under the CC-BY-NC-SA license. Everyone is welcome to help, provide feedback and propose additions or improvements. The git repository is hosted at github.com/gbadev-org/tonc, where you can learn more about how you can help, find detailed contribution guidelines and procedures, file Issues and send Pull Requests.

More resources about GBA development can be found at gbadev.net. There is also a Discord chat dedicated to the gbadev community.

Authors

Jasper "cearn" Vijn is the original author, creating and maintaining Tonc till 2013.

exelotl, avivace, PinoBatch, copyrat90, LunarLambda, gwilymk, mtthgn, and djedditt ported the contents to markdown and migrated the underlying rendering technology to mdbook.

Using this document

In the top navigation bar, you will find a series of icons.

By clicking on the icon you will toggle an interactive table of contents to navigate the document. You can also use → and ← keys on your keyboard to go to the following and previous page.

The lets you choose among 6 different themes and color schemes to please your reading experience, including the classic Tonc theme.

You can search anywhere by pressing s on your keyboard or clicking the icon.

The icon allows you to suggest an edit on the current page by directly opening the source file in the git repository.

This document version was produced from git commit `a44c1e` (2024-04-23 13:38:04 +0200).

Introduction

- [Organisation](#)
- [Terminology and Notation](#)
- [On errors, suggestions](#)

Organisation

TONC consists of three components: a **text** section (the actual tutorial), an **examples** repo with all the source code & makefiles of the various demos, and a library called **libtonc** which holds all the useful/reusable code introduced throughout the tutorial.

Previously these were all distributed as zip files, but times have changed and now they exist in separate git repositories. They will now be explained in more detail:

Tonc text

The text section, which you're reading now, covers the principles of GBA programming in detail. The focus here is not so much on how to get something done, but how things actually *work*, and why it's done the way it's done. After that the *how* often comes naturally. Every chapter has one of more demonstrations of the covered theory, and a brief discussion of the demo itself.

Please, do not make the mistake of only reading the demo discussion: to properly understand how things work you need to read the text in full. While there are optional parts, and whole pages of boring text that seem to have little to do with actual GBA coding, they are there for a reason, usually there's extra conceptual information or gotchas.

At first, the text part had only very little code in it, because I figured the demo code would be at hand and flicking between them would not be annoying. Well, I've realized that I figured wrong and am in the process of including more of the code into these pages; maybe not quite enough to copy-paste and get a clean compile, but enough to go with the explanations of the demos.

The main language will be C, and a smidgeon of assembly. These are the two main languages used in GBA programming even though there are others around. Since the basics of programming are independent of language, it should be possible to adapt them for your chosen language easily.

GBA programming is done close to the hardware, so I hope you know your pointers, hexadecimal numbers and boolean algebra/bit-operations. There's also a fair amount of math here, mostly vector and matrix stuff so I hope your linear algebra is up to speed. Lastly, I am assuming your intellectual capacities exceed those of a random lab monkey, so I won't elaborate on what I consider trivial matters too much.

Aside from the introduction and appendices, the text is divided into 3 parts. First there's 'basics', which explains the absolute essentials for getting anything done. This includes setting up the development environment, basic use of graphics and buttons. It also contains text on what it means to do low level programming and programming efficiently; items that in my view you'd better learn sooner rather than later. The second part covers most of the other items of the GBA like special graphic effects, timers and interrupts. The final section covers more advanced items that uses elements from all chapters. This includes writing text (yes, that's an advanced topic on the GBA), mode 7 graphics and a chapter on ARM assembly.

The Markdown source for the text is all available on GitHub, so please feel free to contribute if you find a typo or something that can be improved.

Tonc code (libtonc & examples)

The source code to all the demos mentioned in the text can be found in the libtonc-examples repository. Like the text, the examples themselves are divided into 3 parts:

basic, *extended* and *advanced*. There is also a `lab` directory with a few interesting projects, but which might not be quite ready. Still interesting to look at, though.

The language we'll be using is **C** with a dash of assembly (but *not* C++). I am working under the assumption that you are familiar with this language. If not, go learn it first because I'm not going to show you; this is not a C course. I do have some links to C tutorials in the references.

Unlike some older GBA tutorials, tonc uses **makefiles** instead of batch scripts to build the example projects, because they're just Plain Better™. How to use these will be explained in the next chapter, but if you just wanted to check out the pre-compiled example ROMs, those are still available here: tonc-bin.zip.

The examples depend on libtonc, a library containing all the important #defines and functions introduced throughout the tutorial. This also includes text writers for all video modes, BIOS routines, a pretty advanced interrupt dispatcher, safe and fast memory copy and fill routines and much more. Historically `libtonc` was included alongside the examples, but nowadays it comes with devkitARM, so you don't have to download it yourself.

Statement of Purpose

I wrote Tonc for two reasons. Firstly, as a way to organize my own thoughts. You often see things in a different light when you write things down and learn from that experience. Secondly, there is a lot of *very bad* information in other tutorials out there (the only exceptions I know of are the new PERN and Deku's sound tutorial. Yes, I am aware of how that sounds, but unfortunately it happens to be true. A number of examples:

- Only very basic information given, sometimes even incorrect info.
- Strong focus on bitmap modes, which are hardly ever used for serious GBA programming.
- Bad programming habits. Adding code/data to projects by #including the files, Using ancient toolchains, non-optimal compiler settings and data-types, and

inefficient (sometimes *very* inefficient) code.

If you are new and have followed the other tutorials, everything will seem to work fine, so what's the problem? Well, that's part of the problem actually. Everything will *seem* fine, until you start bigger projects, at which time you'll find hidden errors and that slow code really bogs things down and you'll have unlearn all the bad habits you picked up and redo everything from the start. The GBA is one of the few platforms where efficient coding still means something, and sometimes all it takes is a change of datatype or compiler switch. These things are better done right from the start.

I've tried to go for completeness first, simplicity second. As a certain wild-haired scientist once said: "Make things as simple as possible, but no simpler." This means things can seem a little technical at times, but that's only because things *are* pretty technical at times, and there's no sense in pretending they're not.

In short, Tonc is *not* "GBA Programming for Dummies", never was, never will be. There's far too much of stuff for Dummies already anyway. If you consider yourself a dummy (and I do mean dummy, not newbie), maybe Tonc isn't the right place. If you're serious about learning GBA programming, however, accept no substitute.

Terminology and Notation

I'm a physicist by training which means that I know my math and its notational conventions. I use both quite often in Tonc, as well as a number of html-tag conventions. To make sure we're all on the same page here's a list:

Type	notation	example
bit n in a `foo`	`foo {n}`	`REG_DISPCNT{4}` (active page bit)
code	<code> tag	`sx`
command/file	<tt> tag	`vid.h`
matrix	bold, uppercase	**P**

memory	hex + code	`0400:002eh`
new term	bold, italic	***charblock***
variable	italics	*x*
vector	bold, lowercase	**v**

I also use some non-ASCII symbols that may not show up properly depending on how old your browser is. These are:

symbol	description
α, β, γ	Greek letters
≈	approximately
½	one half
¼	one quarter
¾	three quarters
≥	greater or equal
↔	double-sided arrow
∈	is in (an interval)
⟨ ⟩	'bra' & 'ket'
→	right arrow
2	superscript 2
×	times

I also make liberal use of shorthand for primitive C types like `char` and `int` and such. These are typedefs that better indicate the size of the variable that's used. Since this is very important in console programming, they're quite common. Anyway, here's a list.

base type	alt name	unsigned	signed	volatile
char	byte	u8	s8	vu8 / vs8
short	halfword	u16	s16	vu16 / vs16

int	word	u32	s32	vu32 / vs32

Finally, there are a number of different notations for hex that I will switch between, depending on the situation. The C notation ('0x' prefix, 0x0400) is common for normal numbers, but I'll also use the assembly affix at times ('h', 0400:0000h). The colon here is merely for ease of reading. It's hard to tell the number of zeros without it.

Register names and descriptions

Getting the GBA to do things often involves the use of the so-called *IO registers*. Certain bits at certain addresses of memory can be used as switches for the various effects that the GBA is capable of. Each register is aliased as a normal variable, and you need to set/clear bits using bit operations. We'll get to where these registers are and what bit does what later; right now I want to show you how I will *present* these, and refer to them in the text.

Each register (or register-like address) is mapped to a dereferenced pointer, usually 16bits long. For example, the display status register is

```
#define  REG_DISPSTAT *(u16*)0x04000004
```

Every time I introduce a register I will give an overview of the bits like this:

```
REG_DISPSTAT @ 0400:0004h
```

F E D C B A 9 8	7 6	5	4	3	2	1	0
VcT	–	VcI	HbI	VbI	VcS	HbS	VbS

The table lists the register's name (`REG_DISPSTAT` , its address (0400:0000h) and the individual bits or bitfields. Sometimes, bits or entire registers are read- or write-only. **Read-only** is indicated with a red overbar (as used here). **Write-only** uses a blue underbar. After it will be a list that describes the various bits, and also gives the #define or #defines I use for that bit:

bits	name	define	description

0	VbS	DSTAT_IN_VBL	VBlank status, read only. Will be set inside VBlank, clear in VDraw.
		other fields	
8-F	VcT	*DSTAT_VCT#*	VCount trigger value. If the current scanline is at this value, bit 2 is set and an interrupt is fired if requested.

The full list of REG_DISPSTAT can be found here. The #defines are usually specific to tonc, by the way. Each site and API has its own terminology here. This is possible because it's not the names that are important, but the numbers they stand for. That goes for the names of the registers themselves too, of course. One last point on the #defines: some of the ones listed have a hash ('#') affix. This is a shorthand notation to indicate that that field has *foo* `_SHIFT` and *foo* `_MASK` #defines, and a *foo* `()` macro. For example, the display register has an 8-bit trigger VCount field, which has '*DSTAT_VCT#*' listed in the define column. This means that the following three things exist in the tonc headers:

```
#define DSTAT_VCT_MASK      0xFF00
#define DSTAT_VCT_SHIFT         8
#define DSTAT_VCT(_n)       ((_n)<<DSTAT_VCT_SHIFT)
```

Lastly, as shorthand for a specific bit in a register, I will use accolades. The number will be a hexadecimal number. For example, REG_DISPCNT{0} is the VBlank status bit (VbS above), and REG_DISPCNT{8-F} would be the whole byte for the VCount trigger.

On errors, suggestions

As much as we (cearn and the gbadev.net community) have tried to weed out things like spelling/grammar errors and broken links, it's surely possible some have slipped by. If you find some, please raise an issue (or even better, make a pull request) on the GitHub repo. Of course, if things are unclear or **gasp** incorrect, or if you have

suggestions, we'd like to know that as well! You can also reach us on Discord / IRC or the Forums.

And, of course:

> This distribution is provided as is, without warranty of any kind. I cannot be held liable for any damage arising out of the use or inability to use this distribution. Code has been tested on emulator and real hardware as well as I could, but I can't guarantee 100% correctness. Both text and code may be modified at any time. Check in once in a while to see if anything's changed. There is also a log in the appendices.

OK that's it. Have fun.

> – *Jasper Vijn* (jakvijn at gmail dot com) and the *gbadev.net* community

1. GBA Hardware

Meet the GBA

The Nintendo Game Boy Advance (GBA) is a portable games console. As if you didn't know already. The CPU is a 32-bit ARM7tdmi chip running at 16.78 MHz. It has a number of distinct memory areas (like work RAM, IO and video memory) which we will look into shortly. The games are stored on *Game Paks*, consisting of ROM for code and data, and fairly often some RAM for saving game info. The GBA has a 240x160 LCD screen capable of displaying 32768 colors (15 bits).

Unfortunately, the screen is not back-lit, which made a lot of people very angry and has generally been regarded as a bad move. So, in 2003 Nintendo launched the GBA SP, a sort of GBA 2.0, which features a fold-up screen reminiscent of the old Game & Watch games (remember those? You do? Boy, you are *old*! (For the record, I still have mine too `: )`)). Then came the final GBA version, the Game Boy Micro, a very, very small GBA which easily fits in everyone's pockets. The differences the GBA, GBA-SP and Micro are mainly cosmetic, though, they're the same thing from a programming point of view.

The original Game Boy took the world by storm in 1989. Not bad for a monochrome handheld console, eh? Later the Game Boy Color was released which finally put some color on the aging machine, but it was still very much a simple Game Boy. The true successor was the GBA, released in 2001. The GBA is backward compatible with the Game Boy, so you can play all the old GB games as well.

In terms of capabilities the GBA is a lot like the Super NES (SNES): 15-bit color, multiple background layers and hardware rotation and scaling. And shoulder buttons, of course. A cynic might look at the enormous amount of SNES ports and say that the GBA *is* a SNES, only portable. This is true, but you can hardly call that a bad thing.

Fig 1.1: original GBA.

Fig 1.2: GBA-SP.

GBA specs and capabilities

Below is a list of the specifications and capabilities of the GBA. This is not a full list, but these are the most important things you need to know.

- Video
    - 240x160 pixel, 15-bit color LCD screen. The original GBA screen was not backlit, but the SP's and Micro's are.

- ○ 3 bitmap modes and 3 tilemap modes and sprites.
- ○ 4 individual tilemap layers (backgrounds) and 128 sprites (objects).
- ○ Affine transformations (rotate/scale/shear) on 2 backgrounds and 32 objects.
- ○ Special graphic effects: mosaic, additive blend, fade to white/black.
- Sound
  - ○ 6 channels total
  - ○ 4 tone generators from the original Game Boy: 2 square wave, 1 general wave and one noise generator.
  - ○ 2 'DirectSound' channels for playing samples and music.
- Miscellaneous
  - ○ 10 buttons (or keys): 4-way directional pad, Select/Start, fire buttons A/B, shoulder buttons L/R.
  - ○ 14 hardware interrupts.
  - ○ 4-player multiplayer mode via a multiboot cable.
  - ○ Optional infrared, solar and gyroscopic interfaces. Other interfaces have also been made by some.
  - ○ Main programming platforms: C/C++ and assembly, though there are tools for Pascal, Forth, Lua and others as well. Easy to start with, yet hard to truly master.

From a programming point of view, the GBA (or any other console for that matter) is totally different from a PC. There is no operating system, no messing with drivers and hardware incompatibilities; it's bits as far as the eye can see. Well, PCs are also just bits, but that's several layers down; on consoles it's just you, the CPU and memory. Basically, it's the Real Programmer's dream.

To get anything done, you use *memory-mapped IO*. Specific areas of memory are mapped directly to hardware functions. In the first demo, for example, we will write the number `0x0403` to memory address `0400:0000h` . This tells the GBA to enable background 2 and set the graphics mode to 3. What this actually *means* is, of course, what this tutorial is for `:)` .

CPU

As said, the GBA runs on a ARM7tdmi RISC chip at 16.78 MHz (2^{24} cycles/second). It is a 32-bit chip that can run on two different instruction sets. First, there's is **ARM code**, which is a set of 32-bit instructions. Then there's **Thumb**, which uses 16-bit instructions. Thumb instructions are a subset of the ARM instruction set; since the instructions are shorter, the code can be smaller, but their power is also reduced. It is recommended that normal code be Thumb code in ROM, and for time-critical code to be ARM code and put in IWRAM. Since all tonc-demos are still rather simple, most (but not all) code is Thumb code.

For more information on the CPU, go to www.arm.com or to the assembly chapter

Memory Sections

This section lists the various memory areas. It's basically a summary of the GBATEK section on memory.

area	start	end	length	port-size	description
System ROM	`0000:0000`	`0000:03FF`	16 KB	32 bit	Bios memory. You can execute it, but not read it (i.o.w, touch, don't look)
EWRAM	`0200:0000h`	`0203:FFFFh`	256 KB	16 bit	External work RAM. Is available for your code and data. If you're using a multiboot cable, this is where the

area	start	end	length	port-size	description
					downloaded code goes and execution starts (normally execution starts at ROM). Due to the 16-bit port, you want this section's code to be Thumb code.
IWRAM	`0300:0000h`	`0300:7FFFh`	32 KB	32 bit	This is also available for code and data. The 32-bit bus and the fact it's embedded in the CPU make this the fastest memory section. The 32-bit bus means that ARM instructions can be loded at once, so put your ARM code here.
IO RAM	`0400:0000h`	`0400:03FFh`	1 KB	32 bit	Memory-mapped IO registers. These have nothing to do with the CPU registers you use in assembly so the name can be

area	start	end	length	port-size	description
					a bit confusing. Don't blame me for that. This section is where you control graphics, sound, buttons and other features.
PAL RAM	`0500:0000h`	`0500:03FFh`	1 KB	16 bit	Memory for two palettes contaning 256 entries of 15-bit colors each. The first is for backgrounds, the second for sprites.
VRAM	`0600:0000h`	`0601:7FFFh`	96 KB	16 bit	Video RAM. This is where the data used for backgrounds and sprites are stored. The interpretation of this data depends on a number of things, including video mode and background and sprite settings.

area	start	end	length	port-size	description
OAM	0700:0000h	0700:03FFh	1 KB	32 bit	Object Attribute Memory. This is where you control the sprites.
PAK ROM	0800:0000h	var	var	16 bit	Game Pak ROM. This is where the game is located and execution starts, except when you're running from a multiboot cable. This size is variable, but the limit is 32 MB. It's a 16-bit bus, so Thumb code is preferable over ARM code here.
Cart RAM	0E00:0000h	var	var	8 bit	This is where saved data is stored. Cart RAM can be in the form of SRAM, Flash ROM or EEPROM. Programatically they all do the same thing: store data. The total size is variable,

area	start	end	length	port-size	description
					but 64 KB is a good indication.

The various RAM sections (apart from Cart RAM) are zeroed at start-up by BIOS. The areas you will deal with them most are IO, PAL, VRAM and OAM. For simple games and demos it will usually suffice to load your graphics data into PAL and VRAM at the start use IO and OAM to take care of the actual interaction. The layout of these two sections is quite complex and almost impossible to figure out on your own (almost, because emulator builders obviously have done just that). With this in mind, reference sheets like GBATEK and the CowBite Spec are unmissable documents. In theory this is all you need to get you started, but in practice using one or more tutorials (such as this one) with example code will save a lot of headaches.

2. Setting up a development environment

Introduction

Unless you want to punch in the instructions in binary in a hex editor, you'll need a development environment to turn human readable code into machine code. This chapter will show you how to set up the necessary components and use them to compile Tonc's examples.

By the end you should have:

- A text editor
- A GBA emulator *(mGBA)*
- A cross-compiler toolchain *(devkitARM)*
- Libraries used for GBA programming *(libtonc in particular)*
- The examples which accompany this tutorial

> **SOME COMMAND-LINE SKILLS REQUIRED**
>
> To compile a GBA game, you'll need a basic undestanding of the command-line. If this is unfamiliar to you, the following Unix command-line tutorial may be

helpful.

If you're on Windows, you should use the **MSYS2** terminal which comes with devkitARM. On other OS's, the built-in terminal should be perfectly adequate.

Choosing a text editor

A decent text editor is essential for programming. At the bare minimum you'll want something that supports syntax highlighting and gives you control over indentation and line endings. That means *notepad.exe* sadly won't cut it.

There are many options, and you may already have a favourite. But in case you don't, here are some suggestions:

- Visual Studio Code - a popular and featureful editor that works on Linux, Windows & Mac

- Kate - another powerful editor, a bit lighter and fully open-source

- Geany - runs well on low-end machines and is still very extensible via plugins

- Notepad++ - a lightweight and widely-loved choice on Windows

Once you've chosen an editor and gotten comfortable with it, you can move onto the next section.

Fig 2.1: Editing a file in VS Code.

In many editors it's possible to set a hotkey (usually `F5` or `Ctrl+Enter`) to compile and run your code. This can be an effective workflow, but for the purposes of this tutorial we'll use the command-line, because it's essential to know what's going on under the hood.

Likewise, code-completion and error highlighting are also valuable features which you may want to spend time setting up, but are outside the scope of this chapter.

Installing a GBA emulator

Needless to say, you'll need a way to actually run your GBA programs. Testing on real hardware from time to time is highly recommended (and part of the fun), but for everyday development you'll want something more convenient. That's where emulators come in.

At the time of writing, the most suitable emulator for GBA development is *mGBA*. It's highly accurate and has features for developers such as memory viewers, debug logging, and a GDB server for step debugging, all of which will make your life a lot easier when things go wrong (and they will)!

Fig 2.2: A GBA ROM running inside mGBA

Other emulators which you might want to use are: NanoBoyAdvance and SkyEmu, which are both *cycle accurate* and effectively the closest you can get to playing on real hardware without actually doing so.

Finally no$gba (debug version) is a somewhat older and less accurate Windows-only GBA emulator, but has some unique debugging features you won't find elsewhere. Namely a visual debugger, performance profiler, CPU usage meters, and memory access checking which can catch buffer overflows and such. If you can get it working, it's an invaluable tool!

Installing devkitARM

devkitARM has been the standard toolchain for GBA homebrew for many years. It is provided by a team called *devkitPro* (dkP), though informally the tools are often referred to as devkitPro too (much to the maintainers' lament).

To install devkitARM, visit the devkitPro Getting Started page and follow the instructions for your OS.

> **DO NOT USE SPACES IN PATHS**
>
> devkitARM uses `make` for building projects, which doesn't cope well with spaces in paths (such as `My Documents`). The reason for this is that `make` uses spaces as a separator between command-line options, but unlike e.g. shell scripts, it doesn't provide an adequate form of quoting/escaping, especially not when working with lists of filenames.

Windows tips

If you are on Windows, there is a GUI installer which downloads and installs the components automatically. Be sure to select "GBA Development" during installation, as shown in fig 2.3.
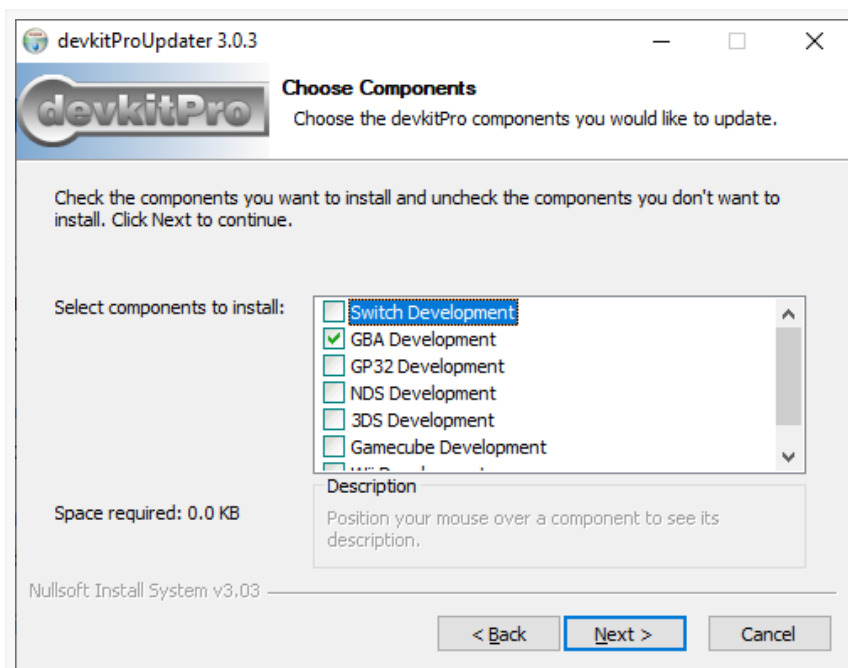

Fig 2.3: Installing devkitARM with the GBA packages on Windows.

Linux & Mac tips

If you are using Linux or Mac, after following the instructions on dkP's Getting Started page, you should install the `gba-dev` package group via `dkp-pacman` in your terminal

(or just `pacman` *if you use Arch Linux).* To do this, run the following command:

```
sudo dkp-pacman -S gba-dev
```

When asked which packages to install *("Enter a selection (default=all):")* you should simply hit `Enter` to install everything in the entire `gba-dev` group.

Obtaining Tonc's example code

This tutorial comes with a full set of examples to demonstrate the concepts taught in each chapter.

Additionally, *libtonc* is the GBA programming library that accompanies Tonc, and is necessary to compile the examples. In the past, libtonc had to be downloaded separately and placed where your projects could find it. But nowadays it comes included as part of devkitARM. As long as you selected the `gba-dev` packages during installation, *you already have libtonc*.

The bad news is devkitARM doesn't include the Tonc examples, so you still have to download those yourself. You can get them via *"Code -> Download Zip"* on the repository page, or by using git in your terminal:

```
git clone https://github.com/gbadev-org/libtonc-examples
```

> **TOOLBOX.H VS LIBTONC**
>
> In the early chapters, we'll be building our own library called `toolbox.h` which replicates parts of libtonc for educational purposes. But for real-world usage,

> sticking to a more featureful, tried-and-tested library (such as libtonc itself) should be preferred.

Compiling the examples

To test your installation, let's try building one of the examples.

In the terminal, navigate to the directory where one of the examples is located (let's say, the *hello* example) and run `make`:

```
cd libtonc-examples/basic/hello
make
```

When invoked, `make` will build the project by following the rules in the file called *'Makefile'* in the current working directory. Assuming this was successful, a `.gba` file will be produced, which you can run in your emulator of choice:
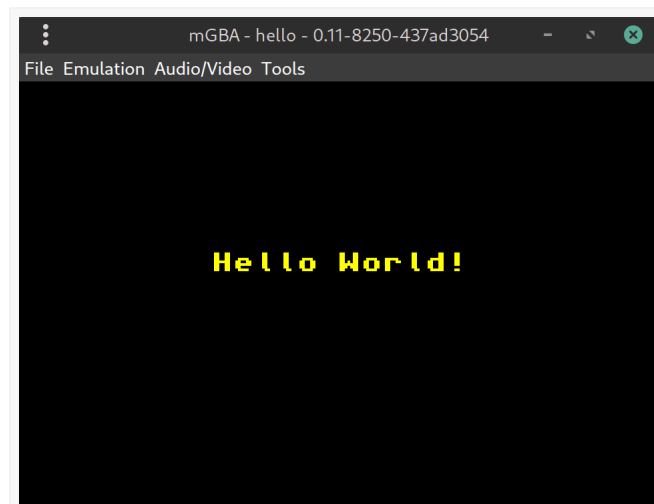
Fig 2.4: One of the Tonc examples running in mGBA.

If you've gotten this far, congratulations! You are now ready to start writing your own GBA programs.

You can move onto the next chapter, or keep reading for more details.

SETTING ENVIRONMENT VARIABLES

If you get an error such as `Please set DEVKITPRO in your environment`, it means your environment variables aren't set properly. The solution to this differs between machines, but usually you want to edit a file called `.bashrc` in your home directory, and add the following lines to it:

```
export DEVKITPRO=/opt/devkitpro
export DEVKITARM=/opt/devkitpro/devkitARM
export DEVKITPPC=/opt/devkitpro/devkitPPC

export PATH=$DEVKITARM/bin:$DEVKITPRO/tools/bin:$PATH  # optional
```

The last line adds the compiler and related tools to your `PATH` environment variable, allowing you to use them directly in your terminal.

This is optional, because the example makefiles also set `PATH` during the build process. But having the tools on hand is useful, and *required* if you want to follow along in the next section.

After editing `.bashrc`, you will have to close and reopen your terminal to apply the changes. Or you can run `source ~/.bashrc` to persist these changes in the current shell.

Manual steps to build a GBA ROM

We've just seen how to compile a GBA program via `make`. Copying the makefile and using it for your own projects is absolutely encouraged! That said, it's valuable to know what's happening under the hood.

Converting your C/C++/asm sources into a valid GBA ROM involves 4 steps, which can be seen in the output from running `make` :

```
$ make
hello.c                 # <--- invoke the compiler
linking cartridge       # <--- invoke the linker
built ... hello.gba     # <--- elf stripped
ROM fixed!              # <--- header fixed
```

The steps are as follows:

1. **Compile/assemble the sources**. We turn the human readable C or C++ files (`.c` / `.cpp`) or assembly files (`.s` / `.asm`) to a binary format known as object files (`.o`). There is one object file for each source file.

   The tool for this is called `arm-none-eabi-gcc` . Actually, this is just a front-end for the real compiler, but that's just details. The `arm-none-eabi-` here is a prefix which means this version of GCC produces machine code for bare-metal ARM platforms; other target platforms have different prefixes. Note that C++ uses `g++` instead of `gcc` .

2. **Link the object files**. After that, the separate object files are linked into a single executable ELF file. Any precompiled code libraries (`.a`) you may have specified are linked at this stage too.

   You can actually compile and link at the same time, but it is good practice to keep them separate: serious projects usually contain multiple source files and you don't want to have to wait for the whole world to recompile when you only changed one. This becomes even more important when you start adding data (graphics, music, etc).

   Again, `arm-none-eabi-gcc` is used for invoking the linker, although the actual linker is called `arm-none-eabi-ld` .

3. **Strip to raw binary**. The ELF file still contains debug data and can't actually be read by the GBA (though many emulators will accept it). `arm-none-eabi-objcopy` removes the debug data and makes sure the GBA will accept it. Well, almost.

4. **Fix the header**. Each GBA game has a header with a checksum to make sure it's a valid GBA ROM. The linking step makes room for one, but leaves it blank, so we have to use a tool like DarkFader's `gbafix` to fix the header. This tool comes with devkitARM, so you don't have to download it separately.

You can of course run all these commands in the terminal yourself without a makefile, provided the dkP tools are in your `PATH`.

Let's try it with the example named *first* - this is the easiest one to compile because it doesn't depend on any libraries.

```
cd libtonc-examples/basic/first/source

# Compile first.c to first.o
arm-none-eabi-gcc -mthumb -c first.c

# Link first.o (and standard libs) to first.elf
arm-none-eabi-gcc -specs=gba.specs -mthumb first.o -o first.elf

# Strip to binary-only
arm-none-eabi-objcopy -O binary first.elf first.gba

# Fix header
gbafix first.gba
```

There you have it - a GBA program compiled from scratch! Well… we can always go deeper but this is probably a good place to stop for now. ⟨ x ⟩

There are various options passed to the tools here that may not be immediately obvious. These are explained in the makefile appendix if you're interested.

AVOID BATCH FILES FOR COMPILING

> You may be tempted to stick all these commands into a batch file or shell script, and use that to compile your project. This is simple, but not recommended.
>
> The reason becomes apparent as soon as your project has more than one source file: if you make an edit to a single file, you shouldn't have to recompile *all* of the sources, only the one that changed. A build system such as `make` is smart enough to realise this, whereas simple shell scripts are not.
>
> When you get to the point where your project has dozens of source files, this makes a big difference!

Alternative toolchains

The advantage of devkitARM is that it provides a consistent environment for compiling GBA homebrew on Windows, Mac and Linux. However, if you're feeling adventurous there are other good options available nowadays:

- gba-toolchain - uses the CMake build system instead of Makefiles
- meson-gba - uses the Meson build system instead of Makefiles
- gba-bootstrap - the bare minimum needed to compile a GBA program. In other words, *roll your own toolchain*, with the hard bits done for you.

Why would you want to use these? They might be easier to install (many Linux distros offer their own builds of `arm-none-eabi-gcc` and related packages, which is essentially the same thing devkitARM provides), or you could be using a machine for which devkitARM is not available (such as a Raspberry Pi). Or perhaps you just want a better build system than makefiles.

Tonc assumes you're using devkitARM, but most of the information is relevant no matter which toolchain you're using.

AVOID 'DEVKITADVANCE'

You may encounter a toolchain called *devkitAdvance*. This is an ancient toolchain which hasn't been updated since 2003. By using it, you will be missing out on *two decades worth* of compiler improvements and optimisations. If somebody recommends this to you, run away!

3. My first GBA demo

Finally, your first GBA program

Now that you have your development environment ready, it's time to take a look at a simple GBA program. For this we will use the code from the C-file *first.c*. The plan at this point is not full understanding; the plan is to get something to compile and get something running. The code will be discussed in this chapter, but what it all means will be covered in later chapters.

```c
// First demo. You are not expected to understand it
// (don't spend too much time trying and read on).
// But if you do understand (as a newbie): wow!

int main()
{
    *(unsigned int*)0x04000000 = 0x0403;

    ((unsigned short*)0x06000000)[120+80*240] = 0x001F;
    ((unsigned short*)0x06000000)[136+80*240] = 0x03E0;
    ((unsigned short*)0x06000000)[120+96*240] = 0x7C00;

    while(1);

    return 0;
}
```

Don't worry about the code just yet, there's time for that later. And don't leave yet, I'll give a nicer version later on. All that matters for now is that you're able to compile and run it.

As explained in the previous chapter, you can run `make` in your terminal to build the project. Under the hood this does the following:

Fig 3.1: picture of the first demo

- **compile** *first.c* to *first.o*,
- **link** the list of object files (currently only *first.o*) to *first.elf*,
- **translate** *first.elf* to *first.gba* by stripping all excess ELF information,
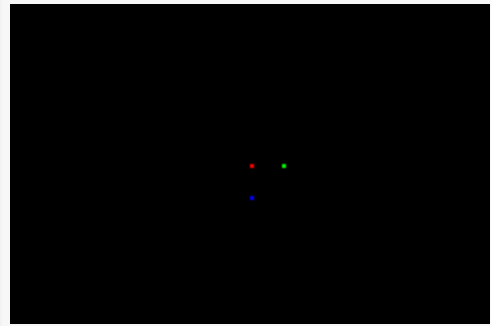- **fix the header** so that the GBA will accept it.

After the makefile has run, you should have a file called *first.gba*. If you don't, there's a problem with your setup because the code sure isn't wrong! Go back to make sure you didn't miss anything - but if you're still having trouble, feel free to stop by on Discord / IRC or the Forums, there's a good chance someone will be able to help!

If you do find yourself with a GBA executable, run it on hardware or your emulator of choice and you should get a red, a green, and a blue pixel at positions (120, 80), (136, 80) and (120, 96), respectively.

Now, for the code itself…

Huh?

If you're somewhat confused by it, you wouldn't be alone. I expect that unless you already know a thing or two about GBA programming or have experience with low-level programming from other platforms, the code will be a total mystery. If you're proficient enough in C you may have some idea what's making the three pixels appear, but I admit that it is *very* hard to see.

And that was kind of my point actually. If one were to hand this in for a test at a programming class, you would fail *so* hard. And if not, the professors should be fired. While the code show above does work, the fact that it's almost unreadable makes it bad code. Writing good code is not only about getting results, it's also about making sure *other* people can understand what's happening without too much trouble.

The code of *first.c* also serves another purpose, namely as a reminder that GBA programming is *very* low-level. You interact directly with the memory, and not through multiple layers of abstraction brought by APIs. To be able to do that means you have to really understand how computers work, which all programmers should know at least to some degree. There are APIs (for lack of a better word) like HAM that take care of the lowest levels, which definitely has its merits as it allows you to deal with more important stuff like actual *game* programming, but on the other hand it hides a lot of details – details that sometimes are better left in the open.

Those who want a better, more intelligible, version of the previous code can skip the next section and move on to the second first demo. The warped minds who can't just let it go and want to have an explanation right now (for the record, I count myself among them), here's what's going on.

Explanation of the code

This is a quick and dirty explanation of the earlier code. Those previously mentioned warped minds for whom this section is intended will probably prefer it that way. A more detailed discussion will be given later.

As I said, GBA programming is low-level programming and sometimes goes right down to the bit. The `0x04000000` and `0x06000000` are parts of the accessible memory sections. These numbers themselves don't mean much, by the way; they just refer to different sections. There aren't really `0x02000000` between these two sections. As you can see in the memory map, these two sections are for the IO registers and VRAM, respectively.

To work with these sections in C, we have to make pointers out of them, which is what the '`unsigned int*`' and '`unsigned short*`' do. The types used here are almost arbitrary; almost, because some of them are more convenient than others. For example, the GBA has a number of different video modes, and in modes 3 and 5 VRAM is used to store 16-bit colors, so in that case casting it to halfword pointers is a good idea. Again, it is not *required* to do so, and in some cases different people will use

different types of pointers. If you're using someone else's code, be sure to note the datatypes of the pointers used, not just the names.

The word at 0400:0000 contains the main bits for the display control. By writing `0x0403` into it, we tell the GBA to use video mode 3 and activate background 2. What this actually means will be explained in the video and bitmap mode chapters.

In mode 3, VRAM is a 16-bit bitmap; when we make a halfword pointer for it, each entry is a pixel. This bitmap itself is the same size as the screen (240x160) and because of the way bitmaps and C matrices work, by using something of the form 'array[*y**width + *x*]' are the contents of coordinates (*x*, *y*) on screen. That gives us our 3 pixel locations. We fill these with three 16-bit numbers that happen to be full red, green and blue in 5.5.5 BGR format. Or is that RGB, I always forget. In any case, that's what makes the pixels appear. After that there is one more important line, which is the infinite loop. Normally, infinite loops are things to be avoided, but in this case what happens after `main()` returns is rather undefined because there's little to return *to*, so it's best to avoid that possibility.

And that's about it. While the Spartan purity of the code does appeal to a part of me, I will stress again that this is *not* the right way to program in C. Save the raw numbers for assembly please.

Your second first GBA program

So, let's start over again and do it *right* this time. Or at least more right than before. There are a number of simple ways to improve the legibility of the code. Here is the list of things we'll do.

- First and foremost is the use of **named literals**, that is to say #defined names for the constants. The numbers that went into the display control will get proper names, as will the colors that we plotted.

- We'll also use #define for the **memory mapping**: the display control and VRAM will then work more like normal variables.
- We'll also create some **typedefs**, both for ease of use and to indicate conceptual types. For instance, a 16-bit color is essentially a halfword like any other, but if you typedef it as, say, `COLOR` , everyone will know that it's not a normal halfword, but has something to do with colors.
- Finally, instead of plotting pixels with an array access, which could still mean anything, well use a **subroutine** for it instead.

Naturally, this will expand the total lines of code a bit. Quite a bit, in fact. But it is well worth it. The code is actually a two-parter. The actual code, the thing that has all the functionality of the first demo, can be found in *second.c*. All the items discussed above, the typedefs, #defines and inlines, are put in *toolbox.h*.

```
// toolbox.h:
//
// === NOTES ===
// * This is a _small_ set of typedefs, #defines and inlines that can
//   be found in libtonc, and might not represent the
//   final forms.


#ifndef TOOLBOX_H
#define TOOLBOX_H


// === (from tonc_types.h) =========================================


typedef unsigned char   u8;
typedef unsigned short  u16;
typedef unsigned int    u32;


typedef u16 COLOR;


#define INLINE static inline


// === (from tonc_memmap.h) =========================================


#define MEM_IO      0x04000000
#define MEM_VRAM    0x06000000


#define REG_DISPCNT     *((volatile u32*)(MEM_IO+0x0000))


// === (from tonc_memdef.h) =========================================


// --- REG_DISPCNT defines ---
#define DCNT_MODE0      0x0000
#define DCNT_MODE1      0x0001
#define DCNT_MODE2      0x0002
#define DCNT_MODE3      0x0003
#define DCNT_MODE4      0x0004
#define DCNT_MODE5      0x0005
// layers
#define DCNT_BG0        0x0100
#define DCNT_BG1        0x0200
#define DCNT_BG2        0x0400
#define DCNT_BG3        0x0800
#define DCNT_OBJ        0x1000



// === (from tonc_video.h) =========================================


#define SCREEN_WIDTH   240
#define SCREEN_HEIGHT  160
```

```
#define vid_mem      ((u16*)MEM_VRAM)

INLINE void m3_plot(int x, int y, COLOR clr)
{    vid_mem[y*SCREEN_WIDTH+x]= clr;      }

#define CLR_BLACK    0x0000
#define CLR_RED      0x001F
#define CLR_LIME     0x03E0
#define CLR_YELLOW   0x03FF
#define CLR_BLUE     0x7C00
#define CLR_MAG      0x7C1F
#define CLR_CYAN     0x7FE0
#define CLR_WHITE    0x7FFF


INLINE COLOR RGB15(u32 red, u32 green, u32 blue)
{    return red | (green<<5) | (blue<<10);    }

#endif // TOOLBOX_H
```

```
#include "toolbox.h"

int main()
{
    REG_DISPCNT= DCNT_MODE3 | DCNT_BG2;

    m3_plot( 120, 80, RGB15(31, 0, 0) );    // or CLR_RED
    m3_plot( 136, 80, RGB15( 0,31, 0) );    // or CLR_LIME
    m3_plot( 120, 96, RGB15( 0, 0,31) );    // or CLR_BLUE

    while(1);

    return 0;
}
```

As you can see, the number of lines in *toolbox.h* is actually much larger than that of the real code. This may seem like a bit of a waste now, but this is only because it's such a small demo. None of the contents of *toolbox.h* is actually compiled, so there is no cost in terms of memory use. In fact, if it did it wouldn't belong in a header file, but that's a discussion I'll go into another time. Right now, let's see what we actually have in *toolbox.h*

The toolbox

Types and typedefs

First of all, we create some shorthand notations of commonly used types. No matter what anyone says, **brevity is a virtue**. For example, unsigned types are very common and writing out the full names (e.g, ' `unsigned short` ') serves little purpose. The shorthand ' `u16` ' is just much more convenient. Besides convenience, it also gives better information on the size of the variable, which is also of great importance here.

I've also added a *conceptual typedef*. While it's true that, in principle, an int is an int no matter what it's used for, it is helpful if you can tell what its supposed use is from its type. In this case, I have a `COLOR` alias for `u16` when I want to indicate a particular halfword contains color information.

Memory map defines

To be able to work directly specific addresses in memory, you'll have to cast them to pointers or arrays and work with those. In this demo's case, the addresses we're interested in are `0600:0000` (VRAM) and `0400:0000` (the display control register). In the first demo I did the casts manually, but it's better to use names for them so that you don't have to remember all the numbers and also because nobody else would have any clue to what's going on.

For the IO registers I'm using the official names, which are recognized by all parties. The display control is known as REG_DISPCNT, and is defined as the word at `0400:0000` . Note that neither the name nor the type are set in stone: you could as easily have called it "BOO" and even used a halfword pointer. The full list of register #defines can be found in libtonc's *regs.h*.

For those who aren't as familiar with pointers as you should (boy, are you gonna be in trouble `:P`), here is the structure of the REG_DISPCNT #define. I'm using `vu32` as a typedef for 'volatile u32' here.

```
#define REG_DISPCNT *((volatile u32*)(MEM_IO+0x0000))
```

code	type	description
MEM_IO+0x0000	Address	MEM_IO= 0x04000000 , so this is address 0400:0000
(vu32*)0x04000000	pointer	A pointer to an unsigned int of the volatile persuasion (ignore this last part for now)
* (vu32*)0x04000000	'variable'	By *dereferencing* the pointer (the '*' unary operator), we access the contents of the pointer. Id est, the whole thing becomes usable as a variable.

So for all intents and purposes, REG_DISPCNT is a variable just like any other. You can assign values to it, read its contents, perform bit operations on it and so forth. Which is good, because that's just the way we want to use that register.

A similar procedure is carried out for VRAM, only this is still in its pointer form in the end. Because of that, `vid_mem` works as an *array*, not a variable. Again, this is exactly how we want things. Please be careful with the definition, though: all the parentheses there are **required**! Because of the operator precedence between casts and arrays, leaving out the outer parentheses pair gives compilation errors.

IO register and their bits

The IO registers (not to be confused with the CPU registers) are a collection of switches in the form of bitfields that control various operations of the GBA. The IO registers can be found in the `0400:0000` range of memory, and are usually clumped into words or halfwords according to personal preference. To get anything done, you have to set specific bits of the IO registers. While you can try to remember all the numerical values of these bits, it's more convenient to use #defines instead.

The toolbox header lists a number of the #defines I use for REG_DISPCNT. The full list can be found in *vid.h* of libtonc, and the register itself is described in the video chapter. For now, we only need DCNT_MODE3 and DCNT_BG2. The former sets the video mode to mode 3, which is simplest of the 3 available bitmap modes, and the latter activates background 2. Out of a total of four, bg 2 is the only one available in the bitmap modes and you have to switch it on if you want anything to show up. You have to admit that these names are a lot more descriptive than `0x0003` and `0x0400`, right?

I've also added a list of useful color defines, even though I'm not using them in *second.c*. They may or may not be useful in the future, though, so it's good to have them around.

Creating the register #defines is probably the biggest part of header files. As a rough estimate, there are 100 registers with 16 bits each, so that would be 1600 #defines. That's a lot. The exact number may be smaller, but it is still large. Because the names of the #defines in and of themselves aren't important, you can expect different naming schemes for different people. I am partial to my own set of names, other older GBA coders may use PERN's names and more recent ones may use libgba's, which comes with devkitARM. Take your pick.

Macros and inline functions

You can also create #defines that work a little like functions. These are called *macros*. I'm not using them here, but there are plenty to be found in libtonc's headers. Like all #defines, macros are part of the preprocessor, not the compiler, which means that the debugger never sees them and they can have many hidden errors in them. For that reason, they have been depreciated in favor of *inline functions*. They have all the benefits of macros (i.e., integrated into the functions that call them so that they're fast), but are still function-like in syntax and resolved at compile time. At least that's the theory, in practice they're not *quite* as speedy as macros, but often preferable anyway.

One inline function I'm using is `m3_plot()`, which, as you may have guessed, is used to plot pixels in mode 3. In mode 3, VRAM is just a matrix of 16bit colors, so all we have to do to plot a pixel is enter a halfword in the right array element. `m3_plot()` looks exactly like a normal function, but because the '`static inline`' in front of it makes it an inline function. Note that inlining is only a recommendation to the compiler, not a commandment, and it only works if optimizations are switched on.

```
// Mode 3 plotter as macro ...
#define M3_PLOT(x, y, clr)   vid_mem[(y)*SCREEN_WIDTH+(x)]=(clr)

// and as an inline function
static inline void m3_plot(int x, int y, COLOR clr)
{   vid_mem[y*SCREEN_WIDTH+x]= clr;   }
```

The second inline function is `RGB15()`, which creates a 16bit color from any given red, green and blue values. The GBA uses 16bit colors for its graphics – or actually 15bit colors in a 5.5.5 BGR format. That's 32 shades of red in the first (lowest) 5 bits, 32 greens in bits 5 to 9, and 32 blues in 10-14. The `RGB15()` inline takes care of all the shifting and masking to make that happen.

The working code

Making use of the contents of *toolbox.h* makes the code of the demo much more understandable.

The first line in `main()` sets a few bits in the display control, commonly known as REG_DISPCNT. I use `DCNT_MODE3` to set the video mode to mode 3, and activate background 2 with `DCNT_BG2`. This translates to `0x0403` as before, but this method gives a better indication of what's happening than entering the raw number. Using a variable-like #define instead of the raw dereferenced pointer is also preferable; especially as the latter is sure to wig out people new to C.

So how do I know what bit does what to create the #defines in the first place? Simple, I looked them up in GBATEK, the essential reference to GBA programming. For every IO register I use in these pages I'll give a description of the bits and a list of #defines as

they're defined in libtonc. The formats for these descriptions were given in the preface, and the table for REG_DISPCNT can be found in the video chapter.

Actually plotting the pixels is now done with the inline function `m3_plot()`, which is formatted much the same way as every kind of pixel plotter in existence: 2 coordinates and the color. Much better than raw memory access, even though it works exactly the same way. The colors themselves are now created with an inline too: `RGB15` takes 3 numbers for the red, green and blue components and ties them together to form a valid 16-bit color.

Finally, there is an endless loop to prevent the program from ever ending. But aren't endless loops bad? Well usually yes, but not here. Remember what happens when PC programs end: control is kicked back to the operating system. Well, we don't *have* an operating system. So what happens after `main()` returns is undefined. It is possible to see what happens by looking at a file called *ctrs0.S*, which comes with your dev-kit, but that's not a thing for beginners so at the moment my advice is to simply not let it happen. Ergo, endless loop. For the record, there are better ways of stopping GBA programs, but this one's the easiest. And now we've reached the end of the demo.

Better, no?

And that is what proper code looks like. As a basic rule, try to avoid raw numbers: nobody except you will know what they mean, and after a while you may forget them yourself as well. Typedefs (and enums and structs) can work wonders in structuring code, so can subroutines (functions, inline functions and macros). Basically, every time you notice you are repeating yourself (copy&paste coding), it might be time to think about creating some subs to replace that code.

These are just a few basic guidelines. If you need more, you can find some more here, for example. Google is your friend. Now, if you've followed any classes on C, you should already know these things. If not, you have been cheated. Books and tutorials may sometimes skip these topics, so it may be necessary to browse around for more guidelines on programming styles. That's all they are, by the way: *guidelines*. While

the rules are usually sensible, there's no need to get fascist about them. Sometimes the rules won't quite work for your situation; in that case feel free to break them. But please keep in mind that these guidelines have been written for a reason: more often than not you will benefit from following them.

First demo v3?

There are many ways that lead to Rome. You've already seen two ways of coding that essentially do the same thing, though one was easily superior. But sometimes things aren't so clear cut. In many cases, there are a number of equally valid ways of programming. The obvious example is the names you give your identifiers. No one's forcing you to a particular set of names because it's not the names that are important, it's what they stand for. Another point of contention is whether you use macros, functions, arrays or what not for dealing with the memory map. In most cases, there's no difference in the compiled code.

The code below shows yet another way of plotting the 3 pixels. In this case, I am using the color #defines rather than the RGB inline, but more importantly I'm using an array typedef `M3LINE` with which I can map VRAM as a matrix so that each pixel is represented by a matrix element. Yes, you can do that, and in some way it's even better than using an inline or macro because you're not limited to just setting pixels; getting, masking and what not are all perfectly possible with a matrix, but if you were to go the subroutine way, you'd have to create more for each type of action.

As you can see, there's all kinds of ways of getting something done, and some are more practical than others. Which one is appropriate for your situation is pretty much up to you; it's just part of software design.

```
#include "toolbox.h"

// extra stuff, also in tonc_video.h
#define M3_WIDTH    SCREEN_WIDTH
// typedef for a whole mode3 line
typedef COLOR       M3LINE[M3_WIDTH];
// m3_mem is a matrix; m3_mem[y][x] is pixel (x,y)
#define m3_mem      ((M3LINE*)MEM_VRAM)

int main()
{
    REG_DISPCNT= DCNT_MODE3 | DCNT_BG2;

    m3_mem[80][120]= CLR_RED;
    m3_mem[80][136]= CLR_LIME;
    m3_mem[96][120]= CLR_BLUE;

    while(1);
    return 0;
}
```

General notes on GBA programming

Console programming is substantially different from PC programming, especially for something like the GBA. There is no operating system, no complex API to learn, it's just you against the memory. You need to have intimate knowledge of the GBA memory sections to make things work, and good pointer and bit-operation skills are a must. Also remember that you don't have a 2GHz CPU, half a gig of RAM and a GPU that can do a gazillion polygons per second. It's just a 16 MHz CPU with 96kB video memory. And *no* floating point support or even hardware division. These are all things you need to be aware of when designing your GBA software.

Another thing that you need to remember is that the GBA has a tendency to do things just a tiny bit different than you may expect. The primary example of this is the matrix used for affine transformations like rotation and scaling. All of the popular tutorials give the wrong matrix for a rotation-scale transformation, even though the reference documents give the correct description of each element. Other good examples are the

end result of trying to write a single byte to VRAM, the fact that bits for key-states are actually *set* when the button's unpressed instead of the other way around, or what the timer register REG_TMxD *really* does.

I've tried to be complete in my explanations of all these things, but I'm positive I've missed a thing or two. If you encounter problems, you're probably not the first one. There are plenty of FAQs and forums where you can find the solution. If that fails, it never hurts to ask. If any of my information is incorrect or incomplete, please don't hesitate to tell me.

GBA Good/bad practices

For some reason, there are a lot of bad programming practices among the GBA development community. The main reason for this is probably that people just copy-paste from tutorial code, most of which use these practices. Here's a short list of things to avoid, and things to adopt.

- **Don't believe everything you read**. Bottom line: people make mistakes. Sometimes, the information that is given is incorrect or incomplete. Sometimes the code doesn't work; sometimes it does, but it's inefficient or inconsistent or just contains practices that will come back to bite you later on. This is true for most (if not all) older tutorials. Don't automatically assume you're doing it wrong: there is a chance it's the source material.

  Unfortunately, if you're new to programming you might not recognize the bad and adopt the standards exhibited by some sources. *Do not learn C programming from GBA tutorials!* I'd almost extent that suggestion to on-line tutorials in general, especially short ones. Books are usually more accurate and provide a better insight into the material. (But again, not always.)

- **RTFAQ / RTFR**. Read the gbadev forum FAQ. Should go without saying. It covers a lot of common problems. Additionally, read the fuckin reference, by which I mean GBATEK, which covers just about everything.

- **Makefiles are good**. Many tutorials use batchfiles for building projects. This is a very easy method, I agree, but in the long run, it's very inefficient, Windows only and is prone to maintainability problems. Makefiles are better for Real World projects, even though there may be a hurdle setting them up initially. Fortunately, you don't have to worry about it that much, because DevkitPro comes with a **template makefile/project** (see `${DEVKITPRO}/examples/gba/template`) where all you need to do is say in which directories the source/header/data files are kept. The makefiles I use for the advanced and lab projects are an adaptation of these.

- **Thumb-code is good**. The standard sections for code (ROM and EWRAM) have 16bit buses. ARM instructions will clog the bus and can seriously slow down performance. Thumb instructions fit better here. Thumb code is often smaller too. Note that because of the 32bit bus of IWRAM, there is no penalty for ARM code there.

- **Enabling interworking, optimizations and warnings are good.** Interworking (`-mthumb-interwork`) allows you to use switch between ARM and Thumb code; you may want this if you have a few high-performance routines in ARM/IWRAM that you want to call from ROM code. Optimizations (`-O #`) make GCC not be an idiot when compiling C into machine code (I'm serious: without them the output is attrociously bad in every way). It produces faster code, and usually smaller as well. Warnings `-Wall` should be enabled because you *will* do stupid things that will produce compilable output, but won't do what you expected. Warnings are reminders that something funky may be going on.

- **32bit variables are good**. Every CPU has a 'native' datatype, also known as the **word**, or in C-speak, the `int` . Generally speaking, the CPU is better equipped to deal with that datatype than any other. The GBA is called a 32bit machine because the CPU's native datatype is 32-bit. The instruction sets are optimised for word-sized chunks. It *likes* words, so you'd better feed it words unless you have no other choice.

In a very real way, the 32bit integer is the *only* datatype the GBA has. The rest are essentially emulated, which carries a small performance penatly (2 extra shift instructions for bytes and halfwords). Do **not** use `u8` or `u16` for loop-indices for example, doing so can cut the speed of a loop *in half*! (The belief that using smaller types means lower memory-use only holds for aggregates and maybe globals; for local variables it actually *costs* memory). Likewise, if you have memory to copy or fill, using words can be about twice as fast as halfwords. Just be careful when casting, because an ARM CPU is very picky when it comes to alignment.

- **Data in header files is *bad*, very bad**. I'll go in a little detail about it when talking about data. And see also here and here.

Those are points where other GBA tutorials often err. It's not an exclusive list, but the main points are there I think. There are also a few things on (C) programming in general that I'd like to mention here.

- **Know the language; know the system**. It should go without saying that if you're programming in a certain language or on a certain system, you should know a little (and preferably a lot) about both. However, I have seen a good deal of code that was problematic simply because the author apparently didn't know much about either. As I said in the beginning of this section, the GBA has a few interesting quirks that you need to know about when programming for it. That, of course, is what Tonc is all about. Some things stem from lack of C skills – the 'int'-thing is an example of this. Another *very* common problem is correct memory and pointer use, something that I will cover a little later and also in the section on data. With C, you have different kinds of datatypes, pointers, the preprocessor and bit-operators at your disposal. Learn what they do and how to use them effectively.

- **Think first, code later**. *Don't* open up an editor, type some code and hope it works correctly. It won't. How can it, if you haven't even defined what 'correctly' means? Think of what you want to do first, then what you need to get it done and *then* try to implement it.

A lot of programming (for me anyway) is not done in a text editor at all. For example, for anything involving math (which can include graphics as well), it's better to make a diagram of the situation. I have pages of diagrams and equations for the affine transformation and mode 7, just to see what what going on. Pen and paper are your friends here.

- **Learn to generalize and simplify**. This is actually not about programming, but problem-solving in general. Specific problems are often special cases of more general problems. For example, 2D math is a subset of multi-dimensional math; vector analysis and transformations such as rotations and scaling are parts of linear algebra. If you know the general solution, you can always (well, *often*, at any rate) work down to the specific case. However, what is often taught (in school, but in universities as well) are the specific solutions, not the general ones. While using the special case solutions are often faster in use, they won't work at all if the case is *just* a little different than the example in the book. If you'd learned the general solution – better yet, how to arrive at the general solution – you'd stand a much better change of solving the task at hand.

  A related issue is simplification. For example, if you have long expressions in a set of equations (or algorithms), you can group them to together under a new name. This means less writing, less writing and a lower risk of making a mistake somewhere.

- **Learn basic optimization strategies**. By this I don't mean that you should know every trick in the book, but there are a few things that you can use in writing code that can speed things up (sometimes even significantly) without cost to readbility and maintainability. In fact, sometimes the code actually becomes easier to read because of it. A few examples:

  - **Use a better algorithm**. Okay, so this one may not always be simple, but it's still very true.
  - **Use ints**. The `int` is loosely defined as the native datatype. Processors tend to perform better when they deal with their native datatype.

- ○ **Use temporary variables for common expressions**. If you use a long expression more than a few times, consider dumping it in a temp. This means less writing for you, and less reading for everyone. It can also make your code faster because you don't need to evaluate the entire expression all the time. This is especially true for global variables, which have to be reloaded after each function-call because the values may have changed.

- ○ **Use pointers**. Pointers have the reputation of being dangerous, but they're a *very* powerful tool if used correctly. Pointer arithmetic is usually faster than indexing because it's closer to hardware, and by assigning temp pointers to deeply nested structure expressions (see above), you can gratly reduce the length of the expressions, which makes things easier on the compiler and the reader alike.

- ○ **Precalculate.** This is related to the previous two points. If you have a loop in which things don't depend on the loop-variable, precalculate that part before the loop. Also, to avoid (complex) calculations, you could dump the data in a Look-up Table and simply grab a value from there.

- ○ **Avoid branches**. Things that redirect program flow (ifs, loops, switches) generally cost more than other operations such as arithmetic. Sometimes it's possible to effectively do the branch with arithmetic (for example, `(int)x>>1` gives −1 or 0, depending on the sign of *x*) There are many more optimization techniques, of course. Wikipedia has a nice overview, and you can find pages discussing particular techniques here[b0rked] and there. Some of these techniques will be done by the compiler anyway, but not always.

- **Learn to optimize *later***. Also known as "premature optimization is the root of all evil". Optimization should be done in the final stages, when most code is in pace and you can actually tell where optimization is necessary (if it's necessary at all). However, this does *not* mean you should actually strive for the slowest solution in the early phases. Often there is a cleaner and/or faster (sometimes even *much* faster) algorithm then the trivial one, which will come to you with just a small amount of thought. This isn't optimization, it's simply a matter of not being stupid. A few of the points mentioned above fall in this category.

- **There are always exceptions**. There is no programming guideline that doesn't have its exception. Except maybe this one.

I'll leave it at that for now. Entire books have been written on how to code efficiently. Resources are available on the well as well: search for things like "optimization", "coding standards" or "coding guidelines" should give you more than enough. Also look up Design Pattern and Anti-pattern. Also fun are books and sites that show how *not* to code. Sometimes these are even more useful. Worse than Failure is one of these (in particular the codeSOD category); The programming section of Computer stupidities is also nice. If you want to see why the use of global variables is generally discouraged, do a search for 'alpha' in the latter page.

A few examples of good/bad practices

Here are a few examples of code that, while functional, could be improved in terms of speed, amount of code and/or maintainability.

Ints versus non-ints

Above, I noted that use of non-ints can be problematic. Because this bad habit is particularly common under GBA and NDS code (both homebrew *and* commercial), I'd like to show you an example of this.

```
// Force a number into range [min, max>
#define CLAMP(x, min, max)    \
    ( (x)>=(max) ? ((max)-1) : ( ((x)<(min)) ? (min) : (x) ) )

// Change brightness of a palette (kinda) (70)
void pal_brightness(u16 *pal, u16 size, s8 bright)
{
    u16 ii;
    s8 r, g, b;

    for(ii=0; ii<size; ii++)
    {
        r= (pal[ii]    )&31;
        g= (pal[ii] >>5)&31;
        b= (pal[ii]>>10)&31;

        r += bright;    r= CLAMP(r, 0, 32);
        g += bright;    g= CLAMP(g, 0, 32);
        b += bright;    b= CLAMP(b, 0, 32);

        pal[ii]= r |(g<<5) | (b<<10);
    }
}
```

This routine brightens or darkens a palette by adding a brightness-factor to the color components, each of which is then clamped to the range [0,31⟩ to avoid funky errors. The basic algorithm is sound, even the implementation is, IMHO, pretty good. What isn't good, however is the datatypes used. Using `s8` and `u16` here adds an extra shift-pair practically every time any variable is used! The loop itself compiles to about 90 Thumb instructions. In contrast, when using `int` s for everything except `pal` the loop is only 45 instructions long. Of course the increase in size means an increase in time as well: the int-only version is 78% faster than the one given above. To repeat that: **the code has doubled in size and slowed down by 78%** *just* **by using the wrong datatype**!

I'll admit that this example is particularly nasty because there is a lot of arithmetic in it. Most functions would incur a smaller penalty. However, there is no reason for losing that performance in the first place. There is no benefit of using `s8` and `u16`; it does not increase redability – all it does is cause bloat and slow-down. **Use 32-bit variables when you can, the others only when you have to**.

Now, before this becomes another goto issue, non-ints do have their place. Variables can be divided into two groups: worker variables (things in registers) and memory variables. Local variables and function parameters are worker variables. These should be 32-bit. Items that are in memory (arrays, globals, structs, and what not) could benefit from being as small as possible. Of course, memory variables still have to be loaded into registers before you can do anything with them. An explicit local variable may be useful here, but it depends on the case at hand.

Pointer problems

One of the most common mistakes GBA neophytes make is mixing up array/pointer sizes when copying data. Data is data, but you can access it in different ways. For example, here's code that copies bitmap-data from an array into VRAM.

```c
// An array representing a 240x160@16 bitmap, converted
// to an array by some graphics conversion tool.
const u8 fooBitmap[240*160*2]=
{
    // Maaaaany, many lines of data.
}

int main()
{
    REG_DISPCNT= DCNT_MODE3 | DCNT_BG2;

    // Copy 240x160 pixels to VRAM (ORLY?)
    int ii;
    for(ii=0; ii<240*160; ii++)
        vid_mem[ii]= fooBitmap[ii];

    return 0;
}
```

The `fooBitmap` array represents some bitmap. In order to display that bitmap on the screen, you need to copy its data into VRAM. That's simple enough: we have `vid_mem` from before, and we can copy from `fooBitmap` to VRAM by copying elements using a simple for-loop.

However, it's not quite as simple as that. `vid_mem` is an `u16` array; so defined because in mode 3 each pixel is an 16-bit color. But `fooBitmap` is a byte-array: *two* elements of this array represent *one* color, and copying bytes-to-halfwords leaves the top-byte of each pixel empty, giving a very skewed result. Such a source-destination is incredibly common, partly because people don't know how pointers and arrays represent memory, but also because they don't pay attention to the datatype.

Here's a version that would work:

```
// An array representing a 240x160@16 bitmap, converted
// to an array by some graphics conversion tool.
const u8 fooBitmap[240*160*2]=
{
    // Maaaaany, many lines of data.
}

int main()
{
    REG_DISPCNT= DCNT_MODE3 | DCNT_BG2;

    u16 *src= (u16*)fooBitmap;   // Cast source to u16-array

    // Copy 240x160 pixels to VRAM (YARLY!)
    int ii;
    for(ii=0; ii<240*160; ii++)
        vid_mem[ii]= src[ii];

    return 0;
}
```

By ensuring the source and destinations are of the same type, the copying leaves no gaps. Note that the underlying data hasn't changed – only how it's used. There are actually a lot more things you need to know about how to use data and memory, which will be covered in a later chapter.

Simplification

Consider the following function (basically taken from the Rinkworks site mentioned earlier):

```
int foo(int x)
{
    switch(x)
    {
    case 1: return 1;
    case 2: return 2;
    case 3: return 3;
    case 4: return 4;
    case 5: return 5;
    case 6: return 6;
    case 7: return 7;
    }
    return 0;
}
```

What this function does is this: if *x* is between 1 and 7, return that number, otherwise return 0. The thing to note is that the case-value and the return code are the same, so instead of the switch-block you could have just returned *x*.

```
int foo(int x)
{
    if(x >= 1 && x <= 7)
        return x;
    else
        return 0;
}
```

Simplifications like this often present themselves if you just think about what you're doing for a little while, rather than just entering code. Now, this would should be rather obvious, but more difficult switch-blocks can often be replaces by something like this as well. For example, if there is a simple mathematical relation between the input and the return value (some addition or multiplication, for example), you can just use that relation. Even if there is not such a simple relation, there can be possibilities. If you're returning constants, you could put those constants in a table and use *x* as an index.

The above is a simplification in terms of the algorithm used. Another kind of simplification is in readability. Of course, everybody has their own ideas about what's readable. Personally, I prefer to keep my statements short, especially in the place where the action happens. The next function is an example of bounding circle

collision detection. Basically, you have two circles at points $\mathbf{p}_1 = (x_1, y_1)$ and $\mathbf{p}_2 = (x_2, y_2)$ and radii r_1 and r_2. The distance between these two points can be calculated with the Pythagorean theorem. If this distance is smaller than the sum of the two radii, the circles overlap. A function that checks whether the player sprite hits any of the enemy sprites could look something like this:

```
// Some basic structs and a sprite array.
// #defines for sprite-indices and amounts omitted.
typedef struct { int x, y; } POINT;

typedef struct
{
    POINT position;
    int radius;
} TSprite;

TSprite gSprites[SPRITE_MAX];

// Collision function.

int player_collision()
{
    int ii;

    for(ii=0; ii<ENEMY_MAX; ii++)
    {
        // Test for hit between player and enemy ii
        if( (gSprites[ENEMY_ID+ii].position.x -
gSprites[PLAYER_ID].position.x) *
            (gSprites[ENEMY_ID+ii].position.x -
gSprites[PLAYER_ID].position.x) +
            (gSprites[ENEMY_ID+ii].position.y -
gSprites[PLAYER_ID].position.y) *
            (gSprites[ENEMY_ID+ii].position.y -
gSprites[PLAYER_ID].position.y) <
            (gSprites[ENEMY_ID+ii].radius + gSprites[PLAYER_ID].radius)
*
            (gSprites[ENEMY_ID+ii].radius + gSprites[PLAYER_ID].radius)
)
        {
            return 1;
        }
    }

    // Not hit
    return 0;
}
```

Personally, I have a hard time reading what actually goes on inside the if-statement there. Because the expression is 6 lines long, I actually have to sit down and parse what it actually does, and hope that the parentheses are all correct, etc. Now, note that a number of things are used multiple times here: the `gSprites` accesses (6× for

the player, 6× for the enemy) and then the positions as well. These can all be accessed with less code by using pointers and other local variables. Also, the player's attributes are loop invariant (they don't change during the loop), so they can be loaded outside the loop.

```c
int player_collision()
{
    int ii;
    int r1= gSprites[PLAYER_ID].radius, r2, dx, dy;
    POINT *pt1= &gSprites[PLAYER_ID].position, *pt2;
    TSprite *enemy= &gSprites[ENEMY_ID];

    for(ii=0; ii<ENEMY_MAX; ii++)
    {
        r2= enemy[ii].radius;
        pt2= &enemy[ii].position;
        dx= pt2->x - pt1->x;
        dy= pt2->y - pt1->y;

        // Test for hit between player and enemy ii
        if( dx*dx + dy*dy < (r1+r2)*(r1+r2) )
            return 1;
    }

    // Not hit
    return 0;
}
```

There may not have been a real change in the number of lines, but the lines themselves are shorter and easier to read. Also, instead of a 6-line `if` -expression, it now fits on a single line and you can actually see what it does. Personally, I'd call that a win.

Testing your code on a real GBA

If you're just starting GBA programming, chances are you're using the emulators that are out there, and will be content with those. However, if you look through the forums

you'll see many people urging you to test on hardware regularly. They are absolutely right.

Now, it isn't that the emulators are bad. On the contrary, in fact; the most popular emulators have things like tile, map and memory viewers that are essential to debugging. An emulator like VBA is very, very good, but not quite perfect. Take the Tonc demos, for example: they run the same on VBA as on a real GBA in all cases … mostly. For one thing, timing is a real issue on most of them (the exception here is no$gba, which I've never seen off the mark by more than 2%, usually a lot less). Also, in a few rare occasions (like in *cbb_demo* and *win_demo*) there were small but important differences between GBA and emulator outputs, and if you've never tested on the real thing, you'd never know.

One other thing that is very different is the colors. Since it's not back-lit the GBA screen is much darker than a PC monitor. Or maybe that's just my room `;)` . Also, on an emulator you have the luxury of scaling your view; the real GBA is always 3" screen. There's world of difference, trust me on this. Take that *first.gba* example I showed above: the pixels are so tiny it's almost impossible to see on a real GBA! Even an 8x8 tile is pretty small. Also, the use of a keyboard in an emu is *nothing* like holding a real GBA in your hands.

And, of course, the whole idea of creating something that works on a console has an air of coolness that defies description. Well, almost anyway. The word is progasm. Says it all really, doesn't it?

Multiboot & linkers

OK, so now you know you should test on hardware, how do you do it? After all, you can't exactly plug a GBA into your PC like a USB memory stick or a printer? Well, yes you can … with the right equipment. The two most common ways are a **multiboot cable** or a **flash linker**.

Flash Card & Linker

A flash card is a GBA cart with a difference: it is completely rewritable. There are a number of different sets available: different sized carts (64Mbit to 1024Mbit), USB or Parallel port versions; sets that use a separate linker (where you have to take the cart out of the GBA, write to it, and reinsert) or ones that write directly to the cart or transfer through the multiboot port. Ideally you'd use one of these. However, they can be rather pricy ($60 - $200 (and up?)) and generally only available through online stores, which also means shipping and taxes and such.

Multimedia cards

A solution that's becoming more and more popular is using standard multimedia cards (eg. SD, CompactFlash) and an adapter like GBAMP and SuperCard. Memory cards can be very cheap (like $10) and bought in most electronics stores; the adapters are genereally $25 and up.

> ### SUPERCARD VS WAITSTATES
>
> There is one small technical problem with SuperCards: they use slow memory that doesn't support 3/1 ROM waitstates. This is a faster setting than the default 4/2 and anything that uses the former simply won't run. This shouldn't be a problem with most homebrew things, but a handful of binaries will fail and you wouldn't be able to make use of the speed-up yourself either.

Multiboot cable

The other way is a multiboot cable. This is a cable that plugs into the GBA multiboot port (like multiplayer games do) and one of the PC ports, usually the parallel port. These are a lot cheaper than a flash kit. You can even build one yourself `:)`! You can find the instructions and necessary software to build an Xboo communication cable

on www.devkitpro.org, which works like a charm. Basically all you need to do is connect one end of the link cable to a male parallel port cable. If you shop around you should be able to get all you need for as little as $5.

But, like always, there's no such thing as a free lunch. What happens in a multiboot game is that the code is written to EWRAM. That's how you can use one cart in a multiplayer game. The multiboot cable is the same thing, only with the PC as the host. The trouble is that EWRAM is only 256kb in size; you won't be able to fit an entire game on it. And, of course, it runs always through your computer, so unless you have a laptop, forget about taking it outside to show off to your friends.

Fig 3.2: efa flash card.

Fig 3.3: SuperCard, compact flash version.

Fig 3.4: xboo multiboot c

Compiling for real hardware

This is almost the same as for emulators. The only real things you have to worry about are a) that you can only use the binary after the `objcopy` treatment, and b) that you need to have a valid GBA header, which it usually doesn't. If the intro screen shows "Game Boy" as normal, but the "Nintendo" at the bottom is garbled, you have a bad header. To get a valid header, use a program called `gbafix.exe`. This is originally by darkfader, but you can also find it at www.devkitpro.org. I already mentioned the extra steps for a multiboot game earlier.

Flash kits usually come with software that can take care of all this stuff for you (or so I'm told, I don't have one). The Xboo zip-file also has a little app that sends your binary to the GBA.

4. Introduction to GBA Graphics

- General introduction
- Draw and blank periods
- Colors and palettes
- Bitmaps, backgrounds and sprites
- Display registers: REG_DISPCNT, REG_DISPSTAT and REG_VCOUNT
- Vsyncing part I, the busy-wait loop

General introduction

The GBA has an LCD screen that is 240 pixels wide, 160 pixels high and is capable of displaying 32768 (15 bit) colors. The refresh rate is just shy of 60 frames per second (59.73 Hz). The GBA has 5 independent layers that can contain graphics: 4 *backgrounds* and one *sprite* layer and is capable of some special effects that include blending two layers and mosaic and, of course, rotation and scaling.

Whereas sound and joypad functionality have to make do with only a few measly registers, the video system has a great deal of memory at its disposal (relatively speaking). Apart from a multitude of registers in I/O memory, there's the 96kb of video memory (starting at `0600:0000h`), palette memory (`0500:0000h`) and OAM memory (`0700:0000h`).

Draw and blank periods

As said, the entire GBA screen is refreshed every 60th of a second, but there's more to it than that. After a scanline has been drawn (the HDraw period, 240 pixels), there is a

pause (HBlank, 68 pixels) before it starts drawing the next scanline. Likewise, after the 160 scanlines (VDraw) is a 68 scanline blank (VBlank) before it starts over again. To avoid tearing, positional data is usually updated at the VBlank. This is why most games run at 60 or 30 fps. (FYI, syncing at the VBlank is also why we in PAL countries often had slower games: PAL TVs run (ran) at 50Hz, hence only 50 fps instead of 60, hence a 17% slower game if nobody bothered to account for it. Few companies ever did :( ).

Both the CowBite Spec and GBATEK give you some interesting details about the timings of the display. A full screen refresh takes exactly 280896 cycles, divided by the clock speed gives a framerate of 59.73. From the Draw/Blank periods given above you can see that there are 4 cycles per pixel, and 1232 cycles per scanline. You can find a summary of timing details in table 4.1.
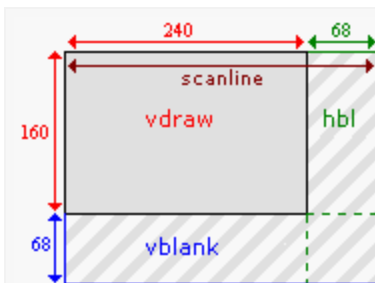

Fig 4.1: vdraw, vblank and hblank periods.

subject	length	cycles
pixel	1	4
HDraw	240px	960
HBlank	68px	272
scanline	Hdraw+Hbl	1232
VDraw	160*scanline	197120
VBlank	68*scanline	83776
refresh	VDraw+Vbl	280896

Table 4.1: Display timing details

Colors and palettes

The GBA is capable of displaying 16bit colors in a 5.5.5 format. That means 5 bits for red, 5 for green and 5 for blue; the leftover bit is unused. Basically, the bit-pattern

looks like this: " `xbbbbbgggggrrrrr` ". There are a number of defines and macros in `color.h` that will make dealing with color easier.

Now, as for palettes…

`<rant>`

Guys, the word here is ***"palette"****! One 'l', two 't's and an 'e' at the end. It is not a* ***"pallet"****, which is "a low, portable platform, usually double-faced, on which materials are stacked for storage or transportation, as in a warehouse", nor is it a* ***"pallette"****, meaning "a plate protecting the armpit, in a suit of armor". The word* ***"pallete"****, its most common variant, isn't even in the dictionary, thus not even worth considering. It's "palette", people, "palette".*

`</rant>`

Anyhoo, the GBA has two palettes, one for sprites (objects) and one for backgrounds. Both palettes contain 256 entries of 16bit colors (512 bytes, each). The background palette starts at `0500:0000h` , immediately followed by the sprite palette at `0500:0200h` . Sprites and backgrounds can use these palettes in two ways: as a single palette with 256 colors (8 bits per pixel); or as 16 sub-palettes or *palette banks* of 16 colors (4 bits per pixel).

One final thing about palettes: index 0 is the ***transparency index***. In paletted modes, pixels with a value of 0 will be transparent.

Bitmaps, backgrounds and sprites

All things considered, the GBA knows 3 types of graphics representations: ***bitmaps***, ***tiled backgrounds*** and ***sprites***. The bitmap and tiled background (also simply known as background) types affect how the whole screen is built up and as such cannot both be activated at the same time.

In bitmap mode, video memory works just like a *w×h* bitmap. To plot a pixel at location (*x,y*), go to location *y**w+*x* and fill in the color. Note that you cannot build up a

screen-full of individual pixels each frame on the GBA, there are simply too many of them.

Tiled backgrounds work completely different. First, you store 8x8 pixel *tile*s in one part of video memory. Then, in another part, you build up a tile-map, which contains indices that tells the GBA which tiles go into the image you see on the screen. To build a screen you'd only need a 30x20 map of numbers and the hardware takes care of drawing the tiles that these numbers point to. This way, you *can* update an entire screen each frame. There are very few games that do not rely on this graphics type.

Finally, we have sprites. Sprites are small (8x8 to 64x64 pixels) graphical objects that can be transformed independently from each other and can be used in conjunction with either bitmap or background types.

> **PREFER TILE MODES OVER BITMAP MODES**
>
> In almost all types of games, the tile modes will be more suitable. Most other tutorials focus on bitmap modes, but that's only because they are easier on beginners, not because of their practical value for games. The vast majority of commercial games use tile modes; that should tell you something.

Those are the three basic graphical types, though other classifications also spring to mind. For example, the bitmap and tiled backgrounds types, since they're mutually exclusive and use the entire screen, constitute the *background*-types. Also, it so happens that the tiles of tiled backgrounds and the sprites have the same memory layout (namely, in groups of 8x8 pixel tiles). This makes tiled backgrounds and sprites the tiled-types.

Display registers: REG_DISPCNT, REG_DISPSTAT and REG_VCOUNT

There are three I/O registers that you will encounter when doing anything graphical: the display control `REG_DISPCNT (0400:0000h)`, the display status `REG_DISPSTAT (0400:0004h)` and the scanline counter `REG_VCOUNT (0400:0006h)`. Those names are simply defines to the memory locations and can, in principle, be chosen at will. However, we will use the names as they appear in the Pern Project, which are the most common.

The REG_DISPCNT register is the primary control of the screen. The bit-layout of this register and their meanings can be found in the following table. This is the general format I will use for registers or register-like sections. The details of the format have already been explained in the preface.

REG_DISPCNT @ 0400:0000h

F	E	D	C	B	A	9	8	7	6	5	4	$\overline{3}$
OW	W1	W0	Obj	BG3	BG2	BG1	BG0	FB	OM	HB	PS	GB

bits	name	define	description
0-2	Mode	DCNT_MODEx. *DCNT_MODE#*	Sets video mode. 0, 1, 2 are tiled modes; 3, 4, 5 are bitmap modes.
$\overline{3}$	GB	DCNT_GB	Is set if cartridge is a GBC game. Read-only.
4	PS	DCNT_PAGE	Page select. Modes 4 and 5 can use page flipping for smoother animation. This bit selects the displayed page (and allowing the other one to be drawn on without artifacts).
5	HB	DCNT_OAM_HBL	Allows access to OAM in an HBlank. OAM is normally locked in VDraw. Will reduce the amount of sprite pixels rendered per line.

6	OM	DCNT_OBJ_1D	Object mapping mode. Tile memory can be seen as a 32x32 matrix of tiles. When sprites are composed of multiple tiles high, this bit tells whether the next row of tiles lies beneath the previous, in correspondence with the matrix structure (2D mapping, `OM`=0), or right next to it, so that memory is arranged as an array of sprites (1D mapping `OM`=1). More on this in the sprite chapter.
7	FB	DCNT_BLANK	Force a screen blank.
8-B	BG0-BG3, Obj	DCNT_BGx, DCNT_OBJ. *DCNT_LAYER#*	Enables rendering of the corresponding background and sprites.
D-F	W0-OW	DCNT_WINx, DCNT_WINOBJ	Enables the use of windows 0, 1 and Object window, respectively. Windows can be used to mask out certain areas (like the lamp did in Zelda:LTTP).

Setting the display control is probably the first thing you'll be doing. For simple demos, you can just set it once and leave it at that, though switching between the video-modes can have some interesting results.

Now the other two registers I mentioned, `REG_DISPSTAT` and `REG_VCOUNT`. The latter tells you the scanline that is currently being worked on. Note that this counter keeps going into the VBlank as well, so it counts to 227 before starting at 0 again. The former gives you information about the Draw/Blank status and is used to set display interrupts. You can also do some really cool stuff with the interrupts that you can enable here. For one thing, the HBlank interrupt is used in creating Mode 7 graphics, and you want to know how that works, don't you?

`REG_DISPSTAT @ 0400:0004h`

F E D C B A 9 8	7 6	5	4	3	$\overline{2}$	$\overline{1}$	$\overline{0}$
VcT	–	VcI	HbI	VbI	VcS	HbS	VbS

bits	name	define	description
0	VbS	DSTAT_IN_VBL	VBlank status, read only. Will be set inside VBlank, clear in VDraw.
1	HbS	DSTAT_IN_HBL	HBlank status, read only. Will be set inside HBlank.
2	VcS	DSTAT_IN_VCT	VCount trigger status. Set if the current scanline matches the scanline trigger (`REG_VCOUNT` == `REG_DISPSTAT` {8-F})
3	VbI	DSTAT_VBL_IRQ	VBlank interrupt request. If set, an interrupt will be fired at VBlank.
4	HbI	DSTAT_HBL_IRQ	HBlank interrupt request.
5	VcI	DSTAT_VCT_IRQ	VCount interrupt request. Fires interrupt if current scanline matches trigger value.
8-F	VcT	*DSTAT_VCT#*	VCount trigger value. If the current scanline is at this value, bit 2 is set and an interrupt is fired if requested.

`REG_VCOUNT @ 0400:0006h (read-only)`

F E D C B A 9 8	7 6 5 4 3 2 1 0
–	Vc

bits	name	description
0-7	Vc	Vertical count. Range is [0,227]

Vsyncing part I, the busy-wait loop

As said, use the VBlank as a timing mechanism and to update the game data. This is called *vsync* (vertical **sync**ronisation). There are a number of ways to vsync. The two most common methods use a while loop and check `REG_VCOUNT` or `REG_DISPSTAT`. For example, since the VBlank starts at scanline 160, you could see when `REG_VCOUNT` goes beyond this value.

```c
#define REG_VCOUNT *(u16*)0x04000006

void vid_vsync()
{    while(REG_VCOUNT < 160);    }
```

Unfortunately, there are a few problems with this code.

First of all, if you're simply doing an empty `while` loop to wait for 160, the compiler may try to get smart, notice that the loop doesn't change `REG_VCOUNT` and put its value in a register for easy reference. Since there is a good chance that that value will be below 160 at some point, you have a nice little infinite loop on your hand. To prevent this, use the keyword *volatile* (see `regs.h`).

Second, in small demos simply waiting for the VBlank isn't enough; you may still be in that VBlank when you call `vid_sync()` again, which will be blazed through immediately. That does not sync to 60 fps. To do this, you first have to wait until the *next* VDraw. This makes our `vid_sync` look a little like this:

```c
#define REG_VCOUNT *(vu16*)0x04000006

void vid_vsync()
{
    while(REG_VCOUNT >= 160);   // wait till VDraw
    while(REG_VCOUNT < 160);    // wait till VBlank
}
```

This will always wait until the start of the next VBlank occurs. And `REG_VCOUNT` is now *volatile* (the " `vu16` " is `typedef` ed as a volatile unsigned (16bit) short. I'll be using

a lot of this kind of shorthand, so get used to it). That's one way to do it. Another is checking the last bit in the display status register, `REG_DISPSTAT` {0}.

So we're done here, right? Errm … no, not exactly. While it's true that you now have an easy way to vsync, it's also a very poor one. While you're in the while loop, you're still burning CPU cycles. Which, of course, costs battery power. And since you're doing absolutely nothing inside that while-loop, you're not just using it, you're actually wasting battery power. Moreover, since you will probably make only small games at first, you'll be wasting a *LOT* of battery power. The recommended way to vsync is putting the CPU in low-power mode when you're done and then use interrupts to bring it back to life again. You can read about the procedure here, but since you have to know how to use interrupts and BIOS calls, you might want to wait a while.

5. The Bitmap modes (mode 3, 4, 5)

- [Introduction](#)
- [The GBA bitmap modes](#)
- [Page flipping](#)
- [On data and how to use it](#)
- [Conclusions](#)

Introduction

In this chapter, we'll look at the bitmap modes. Bitmap modes are a good place to start because there is a one to one relation between the contents of memory and the pixels on the screen. The essentials of all the bitmap modes will be discussed briefly, with a closer look at what you can do in mode 3 as an example. We'll also see a bit of page flipping (mode 4), which allows for smoother animation.

The chapter will close with a section on how to deal with data and computer memory in general. Because GBA programming is very close to the hardware, you *need* to know these things. If you've been programming (in C or assembly) for a long time and have already gained a good understanding on data, datatypes and memory you can probably skip it; for the rest of you, I would strongly urge to read it, because it is very important for all the chapters to come.

Bitmap 101

In fig 5.1 you can find a bitmap of one of the game characters that made Nintendo great. This is probably how most people think of bitmaps: a grid of colored pixels. In order to use bitmaps in a program we need to know how they're arranged in memory. For that we use

Fig 5.1: Link (24x24 bitmap).

fig 5.2 (below); this is a zoomed out version of fig 5.1, with a pixel grid imposed over it and some numbers.

A bitmap is little more than a *w*×*h* matrix of colors (or color-indices), where *w* is the number of columns (the width) and *h* the number of rows (the height). A particular pixel can be referred to with a coordinate pair: (*x, y*). By the way, the y-axis of the GBA points *down*, not up. So pixel (0, 0) is in the top-left corner. In memory, the lines of the bitmap are laid out sequentially, so that the following rule holds: in a *w*×*h* bitmap, the pixel (*x, y*) is the (*w*×*y* + *x*)-th pixel. This is true for all C matrices, by the way.

Fig 5.2 shows how this works. This is a *w*=24 by *h*=24 bitmap, at 8bpp (8 <u>B</u>its <u>P</u>er <u>P</u>ixel (=1 byte)). The numbers in yellow indicate the memory locations; you can count them for yourself if you don't believe me. The first pixel, (0, 0), can be found at location 0. The *last* pixel of the *first* row (23, 0) is at *w*−1 (=23 in this case). The first pixel of the second row (0, 1) is at *w* (=24) etc, etc, till the last pixel at *w*×*h*−1.
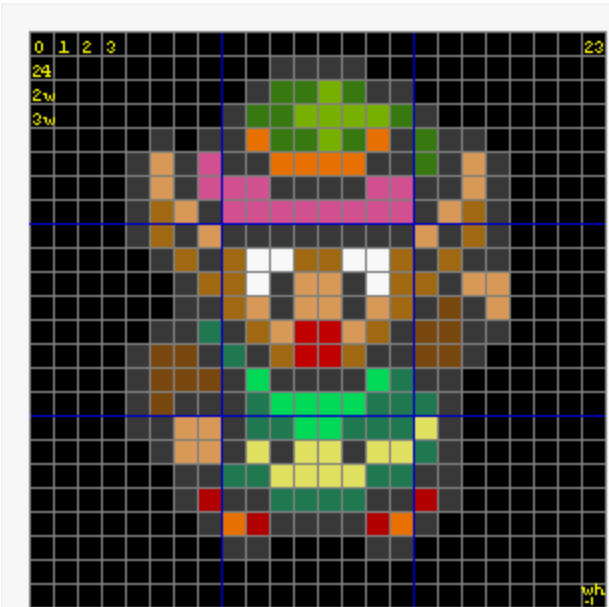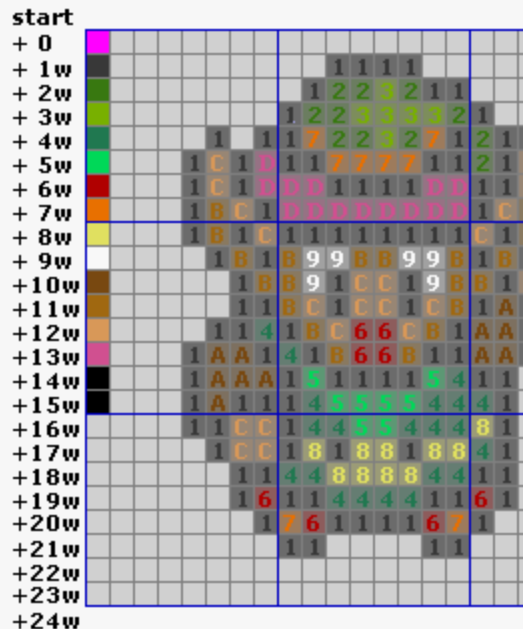

Fig 5.2a: zoom out of fig 5.1, with pixel offsets.

Fig 5.2b: zoom out of fig 5.1, with pixel values omitted for clarity. Palette on the lefthand side

Note, however, that when you use another bitdepth, the addresses change too. For example, at 16bpp (2 bytes per pixel), you'd need to multiply the pixel-number by 2.

Or use another datatype for your array. The general formula is left as an exercise for the reader.

Usually it's not actually the width (i.e., the number of pixels in a row) that's important, but the *pitch*. The pitch is defined as the number of bytes in a scanline. For 8bpp images the pitch and width will usually be the same, but for, say, 16bpp images (2 bytes per pixel) the pitch is the width times two. There's another catch: memory alignment. Alignment will be covered in a later section, but the upshot is that systems generally have a 'preferred' type size and can better deal with data if the addresses are a multiple of that type size. This is why scanlines in some bitmap file formats are always aligned to 32-bit boundaries.

The GBA bitmap modes

Video modes 3, 4 and 5 are the bitmap modes. To use them, put 3, 4 or 5 in the lowest bits of REG_DISPCNT and enable `BG2`. You may wonder why we start with mode 3, rather than mode 0. The reason for this is that bitmaps are a lot easier to come to terms with than tilemaps. And this is the *only* reason. The truth of the matter is that the bitmap modes are just too slow to be used for most conventional GBA games. I can't give an exact figure, but if someone told me 90% or more of GBA games used tile modes and not bitmap modes, I wouldn't be surprised. The only time when bitmap modes would be beneficial would be either for very static screens (introductory demos) or very dynamic screens (3D games like *Star Fox* or *Doom*).

The bitmap modes have the following characteristics:

mode	width	height	bpp	size	page-flip
3	240	160	16	1× 12C00h	No
4	240	160	8	2× 9600h	Yes
5	160	128	16	2× A000h	Yes

Table 5.1: Bitmap mode characteristics

What width, height and bpp mean should be clear by now; the size that the bitmap requires is simply *width × height × bpp/8*. Page flipping may need some more explanation, but first we'll look at some examples of mode 3 graphics.

Drawing primitives in mode 3

We've already seen how to plot pixels, now it's time for some lines and rectangles. Horizontal lines are nearly trivial: because the pixels are in adjacent memory, all you need is a simple loop from the starting *x* to the final *x*. Vertical lines are nearly as easy: while the pixels aren't right next to each other, they do have a fixed offset between them, namely the pitch. So again a simple loop is all you need. Rectangles are essentially multiple horizontal lines, so those are easy as well.

Diagonal lines are a little trickier, for a number of reasons. Diagonal lines have a slope that indicates how many horizontal steps you need to take before moving to the next scanline. That would only work if the absolute value were lower than one, otherwise you'd get gaps between pixels. For higher slopes, you need to increment vertically, and plot horizontally.

Another point is how to make the routine fast enough to be of real use. Fortunately, these things have all been figured out in the past already, so we'll just use the results here. In this case, we'll use a Bresenham Midpoint algorithm for the line drawing, modified to deal with horizontal and vertical lines separately. While I could explain what the routine does exactly, it is out of the scope of the chapter, really.

Two points I have ignored here are normalization and clipping. ***Normalization*** means making sure the routine runs in the right direction. For example, when implementing a line drawing routine that runs from `x1` to `x2` via an incrementing `for` loop, you'd best be sure that `x2` is actually higher than `x1` in the first place. ***Clipping*** means cutting the primitive down to fit inside the viewport. While this is a good thing to do, we will omit it because it can get really hairy to do it well.

The code below is an excerpt from *toolbox.c* from the *m3_demo* and contains functions for drawing lines, rectangles and frames on a 16bpp canvas, like in mode 3

and mode 5. `dstBase` is the base-pointer to the canvas and `dstPitch` is the pitch. The rest of the parameters should be obvious.

```c
#include "toolbox.h"

//! Draw a line on a 16bpp canvas
void bmp16_line(int x1, int y1, int x2, int y2, u32 clr,
    void *dstBase, uint dstPitch)
{
    int ii, dx, dy, xstep, ystep, dd;
    u16 *dst= (u16*)(dstBase + y1*dstPitch + x1*2);
    dstPitch /= 2;

    // --- Normalization ---
    if(x1>x2)
    {    xstep= -1;   dx= x1-x2;   }
    else
    {    xstep= +1;   dx= x2-x1;   }

    if(y1>y2)
    {    ystep= -dstPitch;   dy= y1-y2;   }
    else
    {    ystep= +dstPitch;   dy= y2-y1;   }


    // --- Drawing ---

    if(dy == 0)          // Horizontal
    {
        for(ii=0; ii<=dx; ii++)
            dst[ii*xstep]= clr;
    }
    else if(dx == 0)     // Vertical
    {
        for(ii=0; ii<=dy; ii++)
            dst[ii*ystep]= clr;
    }
    else if(dx>=dy)      // Diagonal, slope <= 1
    {
        dd= 2*dy - dx;

        for(ii=0; ii<=dx; ii++)
        {
            *dst= clr;
            if(dd >= 0)
            {    dd -= 2*dx; dst += ystep;   }

            dd += 2*dy;
            dst += xstep;
        }
    }
    else                     // Diagonal, slope > 1
    {
```

```
        dd= 2*dx - dy;

        for(ii=0; ii<=dy; ii++)
        {
            *dst= clr;
            if(dd >= 0)
            {   dd -= 2*dy; dst += xstep;  }

            dd += 2*dx;
            dst += ystep;
        }
    }
}

//! Draw a rectangle on a 16bpp canvas
void bmp16_rect(int left, int top, int right, int bottom, u32 clr,
    void *dstBase, uint dstPitch)
{
    int ix, iy;

    uint width= right-left, height= bottom-top;
    u16 *dst= (u16*)(dstBase+top*dstPitch + left*2);
    dstPitch /= 2;

    // --- Draw ---
    for(iy=0; iy<height; iy++)
        for(ix=0; ix<width; ix++)
            dst[iy*dstPitch + ix]= clr;
}

//! Draw a frame on a 16bpp canvas
void bmp16_frame(int left, int top, int right, int bottom, u32 clr,
    void *dstBase, uint dstPitch)
{
    // Frame is RB exclusive
    right--;
    bottom--;

    bmp16_line(left,  top,    right, top,    clr, dstBase, dstPitch);
    bmp16_line(left,  bottom, right, bottom, clr, dstBase, dstPitch);

    bmp16_line(left,  top,    left,  bottom, clr, dstBase, dstPitch);
    bmp16_line(right, top,    right, bottom, clr, dstBase, dstPitch);
}
```

These functions are very general: they will work for anything that has 16-bit colors.
That said, it may be annoying to have to add the canvas pointer and pitch all the time,

so you could create an *interface layer* specifically for mode 3 and mode 5. The ones for mode 3 would look something like this:

```
typedef u16 COLOR;

#define vid_mem          ((COLOR*)MEM_VRAM)

#define M3_WIDTH         240


// === PROTOTYPES ========================================================

INLINE void m3_plot(int x, int y, COLOR clr);
INLINE void m3_line(int x1, int y1, int x2, int y2, COLOR clr);
INLINE void m3_rect(int left, int top, int right, int bottom, COLOR
clr);
INLINE void m3_frame(int left, int top, int right, int bottom, COLOR
clr);


// === INLINES ===========================================================

//! Plot a single \a clr colored pixel in mode 3 at (\a x, \a y).
INLINE void m3_plot(int x, int y, COLOR clr)
{
    vid_mem[y*M3_WIDTH+x]= clr;
}

//! Draw a \a clr colored line in mode 3.
INLINE void m3_line(int x1, int y1, int x2, int y2, COLOR clr)
{
    bmp16_line(x1, y1, x2, y2, clr, vid_mem, M3_WIDTH*2);
}

//! Draw a \a clr colored rectangle in mode 3.
INLINE void m3_rect(int left, int top, int right, int bottom, COLOR clr)
{
    bmp16_rect(left, top, right, bottom, clr, vid_mem, M3_WIDTH*2);
}

//! Draw a \a clr colored frame in mode 3.
INLINE void m3_frame(int left, int top, int right, int bottom, COLOR
clr)
{
    bmp16_frame(left, top, right, bottom, clr, vid_mem, M3_WIDTH*2);
}
```

Finally, there is a `m3_fill()` function, that fills the entire mode 3 canvas with a single color.

```
//! Fill the mode 3 background with color \a clr.
void m3_fill(COLOR clr)
{
    int ii;
    u32 *dst= (u32*)vid_mem;
    u32 wd= (clr<<16) | clr;

    for(ii=0; ii<M3_SIZE/4; ii++)
        *dst++= wd;
}
```

Now, note what I'm doing here: instead of treating VRAM as an array of 16-bit values which are appropriate for 16bpp colors, I'm using a 32-bit pointer and filling VRAM with a 32-bit variable containing two colors. When filling large chunks of memory, it makes no difference if I fill it in *N* 16-bit chunks, or ½*N* 32-bit chunks. However, because you only use half the number of iterations in the

Fig 5.3: drawing in mode 3.

latter case, it's roughly twice as fast. In C, it's perfectly legal to do something like this (provided that strict aliasing is satisfied) and often actually useful. This is why it's important to know the principles of data and memory. Also note that I'm using pointer arithmetic here instead of array indices. While the compiler generally make the conversion itself, doing it manually is still often a little faster. (When in doubt, read the assembly language that GCC generates.)
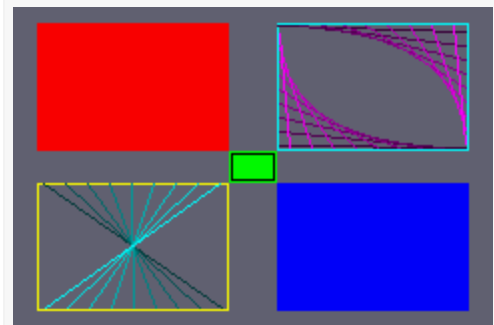
While this method is already twice as fast as the 'normal' method, there are actually much faster methods as well. We will meet these later, when we stop using separate toolkit files and start using libtonc, the code library for tonc. Tonclib contains the functions described above (only faster), as well as 8bpp variations of the `bmp16_` routines and interfaces for mode 4 and mode 5.

Below you can find the main code for *m3_demo*, which uses the `m3_` functions to draw some items on the screen. Technically, it's bad form to use this many magic numbers, but for demonstration purposes it should be okay. The result can be seen in fig 5.3.

```c
#include "toolbox.h"

int main()
{
    int ii, jj;

    REG_DISPCNT= DCNT_MODE3 | DCNT_BG2;

    // Fill screen with grey color
    m3_fill(RGB15(12, 12, 14));

    // Rectangles:
    m3_rect( 12,  8, 108,  72, CLR_RED);
    m3_rect(108, 72, 132,  88, CLR_LIME);
    m3_rect(132, 88, 228, 152, CLR_BLUE);

    // Rectangle frames
    m3_frame(132,  8, 228,  72, CLR_CYAN);
    m3_frame(109, 73, 131,  87, CLR_BLACK);
    m3_frame( 12, 88, 108, 152, CLR_YELLOW);

    // Lines in top right frame
    for(ii=0; ii<=8; ii++)
    {
        jj= 3*ii+7;
        m3_line(132+11*ii, 9, 226, 12+7*ii, RGB15(jj, 0, jj));
        m3_line(226-11*ii,70, 133, 69-7*ii, RGB15(jj, 0, jj));
    }

    // Lines in bottom left frame
    for(ii=0; ii<=8; ii++)
    {
        jj= 3*ii+7;
        m3_line(15+11*ii, 88, 104-11*ii, 150, RGB15(0, jj, jj));
    }

    while(1);

    return 0;
}
```

A dash of mode 4

Mode 4 is another bitmap mode. It also has a 240×160 frame-buffer, but instead of 16bpp pixels it uses 8bpp pixels. These 8 bits are a ***palette index*** to the background palette located at `0500:0000`. The color you'll see on screen is the color found in the palette at that location.

Pixels of a bitdepth of 8 mean you can only have 256 colors at a time (instead of 32768 in the case of 15bpp), but there are benefits as well. For one, you can manipulate the colors of many pixels by simply changing the color in the palette. An 8bpp frame-buffer also takes up half as much memory as a 16bpp buffer. Not only is it faster to fill (well, in principle anyway), but there is now also room for a second buffer to allow page flipping. Why that's useful will be covered in a minute.

There is, however, one major downsize to using mode 4, which stems from a hardware limitation. With 8-bit pixels, it'd make sense to map VRAM as an array of bytes. This would be fine if it weren't for the rather annoying fact that VRAM does not allow byte-writes! Now, because this is a very important point, let me repeat that: **You cannot write to VRAM in byte-sized chunks!!!**. Byte reads are ok, but writes have to be done in 16-bit or 32-bit chunks. If you *do* write in bytes to VRAM, the halfword you're accessing will end up with that byte duplicated into both the lower and upper bytes: you're setting two pixels at once. Note that this no-byte-write rule also extends to palette memory and OAM, but there it doesn't cause trouble because you won't be using that as bytes anyway.

So how to plot single-pixels then? Well, you have to read the whole halfword you're trying to access, mask off the bits you don't want to overwrite, insert your pixels and then write it back. In code:

```
#define M4_WIDTH    240      // Width in mode 4
u16 *vid_page= vid_mem;      // Point to current frame buffer

INLINE void m4_plot(int x, int y, u8 clrid)
{
    u16 *dst= &vid_page[(y*M4_WIDTH+x)/2];  // Division by 2 due to
u8/u16 pointer mismatch!
    if(x&1)
        *dst= (*dst& 0xFF) | (clrid<<8);     // odd pixel
    else
        *dst= (*dst&~0xFF) |  clrid;         // even pixel
}
```

As you can see, it's a little more complicated than `m3_plot()`. It takes a lot longer to run as well. Still, once you have a pixel plotter, you can create other rendering routines with ease. The basic code for drawing lines, rectangles, circles and the like are pretty much independent of how pixels are formatted. For example, drawing a rectangle is basically plotting pixels in a double loop.

```
void generic_rect(int left, int top, int right, int bottom, COLOR clr)
{
    int ix, iy;
    for(iy=top; iy<bottom; iy++)
        for(ix=left; ix<right; ix++)
            generic_plot(ix, iy, clr);
}
```

This is the generic template for a rectangle drawing routine. As long as you have a functional pixel plotter, you're in business. However, business will be *very* slow in mode 4, because of the complicated form of the plotter. In all likelihood, it'll be so slow to make it useless for games. There is a way out, though. The reason `m4_plot()` is slow is because you have to take care not to overwrite the other pixel. However, when you're drawing a horizontal line (basically the `ix` loop here), chances are that you'll have to give that other pixel the same color anyway, so you needn't bother with read-mask-write stuff except at the edges. The implementation of this faster (*much* faster) line algorithm and subsequently rectangle drawer is left as an exercise for the reader. Or you can seek out *tonc_bmp8.c* in libtonc.

VRAM VS. BYTE WRITES

You cannot write individual bytes into VRAM (or the palette or OAM for that matter). Halfwords or words only, please. If you want to write single bytes, you have to read the full (half)word, insert the byte, and put it back.

Please don't skip this note, and make yourself aware of the full ramifications of this. Errors due to pointer-type mismatches are very easy to make, and you may be writing to VRAM as bytes more often than you think.

GENERIC VS. SPECIFIC RENDERING ROUTINES

Every kind of graphics surface needs its own pixel plotter. In principle, more complicated (multi-pixel) shapes are surface independent. For example, a line routine follows the same algorithm, but simply uses a different plotter for drawing pixels. These generic forms are great in terms of re-usability and maintainability, but can be *disastrous* when it comes to speed. Creating surface-specific renderers may be extra work, but can on occasion save you up to a factor of 100 in speed.

Complications of bitmap modes

While I could go on to discuss more complicated matters, such as drawing rectangles and blits and text, there's very little reason to do so at this junction. As I said before, the bitmap modes are useful to learn some basic functionality, but for most practical purposes, you're better off with tiled modes.

The primary issue is speed. Even simple primitives such as the ones shown here can take a lot of time, especially if you're not careful in your implementation. For example, a full mode 3 screen-wipe would take about 60% of a VBlank **at best**! In bad implementations of a screen-wipe, like doing it with a rectangle drawer that calls a non-inline pixel-plotting function, could take as much as 10 frames. And *then* you still

have to draw all your backgrounds and sprites and do the game logic. The phrase 'crawling horror' somehow springs to mind at the thought of this.

Aside from that, bitmap modes can use only one background and have no hardware scrolling to speak of. Also, though this is jumping the gun a bit, it overlaps the memory that contains the sprite tiles, which starts at `0601:0000h`. For that reason, you will only be able to use sprite-tiles 512 to 1023 when in modes 3-5.

Page flipping can alleviate some of these items, but that's not available in mode 3. It is in mode 5, but that uses only a small portion of the screen, so gaming with only that looks awkward. As for mode 4, well, that's one of those places where you will *really* see what programming close to the hardware means: it doesn't allow you to write to VRAM in byte-sized chunks! The only way to have a single-pixel resolution is to combine 2 adjacent pixels and write those, which costs a lot of extra time.

So basically, use the bitmap modes for testing and/or static images, but not much else unless you know the tilemodes can't do what you want.

> **BITMAP MODES ARE NOT FOR GAMING**
>
> Do not get too comfortable with bitmap modes. Though they're nice for gbadev introductory sections because they are easier to work with than tile modes, and they have advantages for 3D games, they are *not* suitable for most types of games because the GBA simply can't push pixels fast enough. Tinker with them to get a feel for IO registers and the like, then move on.

Page flipping

Page flipping is a technique that eliminates nasty artifacts like tearing in animation. There are two things going on at the same time in an animation: placing the pixels on bitmap (writing), and drawing the bitmap on screen (displaying). Software takes care

of writing, updating the positions of characters etc; hardware does the displaying: it simply takes the bitmap and copies it to the screen. The problem is that both these processes take time. What's worse, they happen at the same time. And when the game state changes in mid draw, the bottom section will be of the current state, while the top section will represent the previous state. Needless to say, this is bad.

Fig 5.4: Page flipping procedure. No data is copied, only the 'display' and 'write' pointers are swapped.
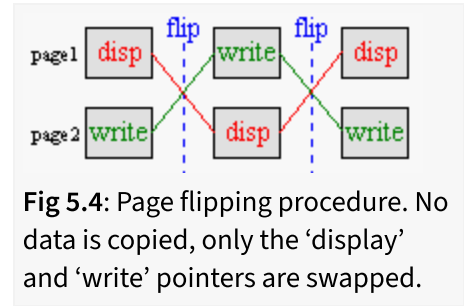
Enter page flipping. Instead of using one single bitmap to write and display, you use two. While one bitmap is displayed, you write all you need onto a second bitmap (the back-buffer). Then, when you're finished, you tell the hardware to display that second bitmap and you can prepare the next frame on the first. No artifacts whatsoever.

While the procedure works great, there are some snares. For the first, consider this. Given are the pointers to the two pages `page1` and `page2` . Right now, `page1` is displayed and `page2` is being prepared; so far so good. But when you switch to the second page, this only makes `page2` the display-page; you have to make `page1` the write-page yourself! The solution to this problem is simple: use a write-buffer pointer, but it can catch you off-guard if you're new to this stuff.

The second problem concerns a little nasty in the age-old method of animation. The canonical animation does this. Frame1: draw object. Frame2: erase old object, draw object in new state. This doesn't work for page flipping since Frame2 is written on an entirely different bitmap than Frame1, so trying to erase Frame1's old object doesn't. What you need to erase is the object from 2 frames ago. Again, easy solution, but you have be aware of the problem. (Of course, erasing the entire frame each time would work too, but who's got the time?)

PAGEFLIPPING, NOT DOUBLE BUFFERING

Another method of smoother animation is double buffering: draw on a secondary buffer (the backbuffer) and copy it to the screen when finished. This

> is a fundamentally different technique than page flipping! Even though both use two buffers, in page flipping you don't copy the backbuffer to the display buffer, you *make* backbuffer the display buffer.
>
> What the GBA does is page flipping, so refer to it as such.

GBA page flipping

The second page of the GBA is located at location `0600:A000h`. If you look at the size required for mode 3, you'll see why it doesn't have page-flipping capabilities: there's no room for a second page. To set the GBA to display the second page, set `REG_DISPCNT` {4}. My page flipping function looks a little like this:

```c
u16 *vid_flip()
{
    // toggle the write_buffer's page
    vid_page= (u16*)((u32)vid_page ^ VID_FLIP);
    REG_DISPCNT ^= DCNT_PAGE;            // update control register
    return vid_page;
}
```

The code is relatively straightforward. `vid_page` is the pointer that always points to the write-page. I had to pull a little casting trickery to get the `XOR` to work (C doesn't like it when you try it on pointers). On the GBA, the steps for page flipping are perfectly xorrable operations. Sure, you *could* just put the equivalent in an `if-else` block, but where's the fun in that :P?

Page flipping demo

What follows is the code (sans data) for the *pageflip* demo. The actual part concerned with page flipping is very small. In fact, the actual flip is merely a call to `vid_flip()` once every 60 frames = 1 second (point 3). We'll also have to set the video mode to something that actually has pages to flip, which in this case is mode 4.

What we'll have to do as well is load the data that will be displayed on these two pages. I'm using the standard C routine `memcpy()` for the copy, because that's the standard way of copying things in C. While it's faster than manual loops, it does come with a few snares that you need to be aware of before using it everywhere. Tonclib comes with faster and safer routines, but we'll get to those when it's time.

Loading a bitmap is very simple in theory, but the bitmap(s) I'm using are only 144×16 in size, while the VRAM page's pitch is 240 pixels wide. This means that we'll have to copy each scanline separately, which is done at point (1). Note that I'm copying `frontBitmap` to `vid_mem_front` and `backBitmap` to `vid_mem_back`, because those are the starting locations of the two pages.

Since these are mode 4 bitmaps, they'll also need a palette. Both palettes use `frontPal`, but instead of using `memcpy()` to copy it to the background palette memory, I'm using a `u32` array because … well, just because I guess.

Lastly, you can pause and unpause the demo by holding the Start Button.

```c
#include <string.h>

#include <toolbox.h>
#include "page_pic.h"

void load_gfx()
{
    int ii;
    // (1) Because my bitmaps here don't fit the  screen size,
    // I'll have to load them one scanlline at a time
    for(ii=0; ii<16; ii++)
    {
        memcpy(&vid_mem_front[ii*120], &frontBitmap[ii*144/4], 144);
        memcpy(&vid_mem_back[ii*120], &backBitmap[ii*144/4], 144);
    }

    // (2) You don't have to do everything with memcpy.
    // In fact, for small blocks it might be better if you didn't.
    // Just mind your types, though. No sense in copying from a 32-bit
    // array to a 16-bit one.
    u32 *dst= (u32*)pal_bg_mem;
    for(ii=0; ii<8; ii++)
        dst[ii]= frontPal[ii];
}

int main()
{
    int ii=0;

    load_gfx();
    // Set video mode to 4 (8bpp, 2 pages)
    REG_DISPCNT= DCNT_MODE4 | DCNT_BG2;

    while(1)
    {
        while(KEY_DOWN_NOW(KEY_START)); // pause with start
        vid_vsync();

        // (3) Count 60 frames, then flip pages
        if(++ii == 60)
        {
            ii=0;
            vid_flip();
        }
    }
    return 0;
}
```
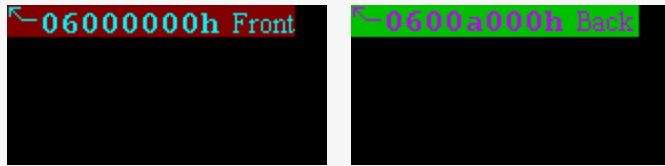
Fig 5.5: the page flipping demo switches between these two blocks.

On data and how to use it

This section is a little boring (ok, very boring) but it needs to be said. While books and tutorials on C may use data for whatever purpose, they often gloss over what data actually *is* at the lowest level, and how to deal with it correctly. As you'll be dealing directly with hardware and memory here, it is important that you are aware of these items, preferably even understand them, so that they don't bite you in the ass at some later point.

The first two subsections are about how to get graphics into your game, something that you'll really need to know. After that I'll discuss a few nasty and highly technical things that may or may not cause problems later on. These are optional and you can skip to the data-loading/interpreting demo at any time. That said, I urge you to read them anyway because they may save you a lot of debugging time.

> **RELAX, IT'S ONLY 1S AND 0S**
>
> When you get right down to it, everything on computers is merely a big mess of bits without any purpose by itself. It is the interaction between hardware and software that makes sequences of bits appear as valid executable code, a bitmap, music or whatever.

Yes, we have no files

This may be a good point to say a few words on data. Strictly speaking, *everything* is data, but in this case I'm referring to data that on PC games would be separate from the executable: graphics, music, maybe scripts and text-files and what not. This all works fine on a PC, but not so fine on the GBA because there *is no file system*. This means that you cannot use the standard file I/O routines (`fscanf()` , `fread()` , etc) to read the data, because there are no files to read them from.

All the game's data has to be added directly to the binary. There are a number of ways to do this. The most common way is to convert the raw binary files to C-arrays, then compile those and link them to the project. Well, the most common among homebrewers is probably converting to C arrays and using `#include` on them, but that's something that you should *never* do. Also popular are assembly arrays. These are a useful alternative to C arrays because a) they *can't* be `#include` d and b) because they bypass the compilation step and compilation of arrays is very intensive. Of course, you would have to know how to work with the assembler. Another nice thing about the assembler is that you can include binary files directly into them, eliminating the need for a converter. Lastly, while the GBA doesn't have a native file system, you can always write your own. A common one is GBFS by the gbadev forum FAQ maintainer, tepples. Using a file system is actually the recommended method, but for now, I'll stick to C arrays because they are the easiest to use.

> **AHEM. ACTUALLY, WE DO HAVE FILES**
>
> There *were* no files in the past, but in July of 2006, Chishm gave us libfat, which is a FAT-like file system for GBA and Nintendo DS. It is distributed via devkitPro Updater as well, so chances are you have it already.

Where do my arrays go?

By default, arrays go into IWRAM. You know, the one that's only 32 KiB long. Now, a mode 3 bitmap is 240×160×2 = 77 kB. Obviously, trying to put a 77 kB object into a 32 KiB section would fit nicely into the bad things category. To avoid this, put it in the read-only section (ROM), which is much larger. All you have to do for this is add the `const` keyword to the definition if you're using C, or the `.rodata` directive in assembly. Note that for multiboot programs, ROM actually means EWRAM, which is only 256 KiB long. The latter would fit three mode 3 bitmaps; more would again be bad unless you use compression.

Note that what I said about arrays is true for *all* arrays, not just data arrays: if you want any kind of large array (like a backbuffer for mode 3), it would also default to and *overfill* IWRAM. But you can't make it `const` because then you'd not be able to write on it. GCC has attributes that lets you choose where things are put – in EWRAM for instance. Here are the commonly seen `#define` macros for the attributes that can be used for specific section placement.

```
#define EWRAM_DATA __attribute__((section(".ewram")))
#define IWRAM_DATA __attribute__((section(".iwram")))
#define  EWRAM_BSS __attribute__((section(".sbss")))

#define EWRAM_CODE __attribute__((section(".ewram"), long_call))
#define IWRAM_CODE __attribute__((section(".iwram"), long_call))
```

CONST IS GOOD

Data that you don't expect to change in your game should be defined as constant data using the `const` keyword, lest it trashes your IWRAM.

Converted and const arrays in C++

There are two little snags that you can trip on if you're using (converted) data arrays in C++. The first is that tools that generate the arrays will output C files, not C++ files. This is not a problem in itself because those files will be compiled just the same. What *is* a problem is that C++ uses something known as Name mangling to allow overloading and stuff like that. C doesn't and as a result, the name that the C++ file looks for isn't the same one as in the C file and you get undefined references. To fix this, use `extern "C"` in front or around the declarations of the stuff in the C files.

```
// This:

extern "C" const unsigned char C_array[];

// Or this:
extern "C"
{
const unsigned char C_array1[];
const unsigned char C_array2[];
}
```

Another problem with C++ is that const-arrays are considered static (local to the file that contains it) unless you add an external declaration to it. So if you just have `const u8 foo[]= { etc }` in a file, the array will be invisible to other files. The solution here would be to add the declaration inside the file itself as well.

```
// foo.cpp. Always have an external declaration
// inside the file as well.

extern const unsigned char foo[];

const unsigned char foo[]=
{
    // data
};
```

Data conversion

It's rather easy to write a tool that converts a binary file to a C or asm array. In fact, devkitARM comes with two that do just that: raw2c.exe and bin2s.exe. It also comes with the basic tools for gbfs by the way. But being able to attach binary files to your game is only part of the story. Consider a bitmap, for example. In principle, a bitmap is a binary file just like any other. There's nothing inherently graphical about it, and it doesn't magically appear as a bitmap whenever you use it by itself. Yes, when you double-click on it, an image viewer may pop up and display it, but that's only because there's some serious work by the OS going on underneath. Which we don't have here.

Most files will follow a certain format to tell it what it is, and how to use it. For bitmaps, that usually means width, height, bitdepths and a few other fields as well. The point is that they're not directly usable. You can't just attach, say, a BMP file to your project and copy it to VRAM and think that everything will work out. No, you have to *convert* it to a GBA-usable format. Now, you can do this internally (on the GBA itself), or externally (on the PC and attach the converted data to the project). Because the latter is a much more efficient use of GBA resources, that is the usual procedure.

There are many conversion tools, one might almost say too many. Some are one-trick ponies: a single file-type to a single graphics mode for example. Some are very powerful and can handle multiple file-types, multiple files, different conversion modes with lots of options on the side, and compression. It should be obvious which are of the most value.

A good one is gfx2gba. This is a command-line tool so that it can be used in a makefile, but there is a GUI front-end for it as well. This tool has the Good Things I mentioned earlier, plus some map-exporting options and palette merging, but the input file must be 8-bit and I hear that while it does compress data, the array-size is still given as its uncompressed size for some unfortunate reason. This tool comes with the HAM installation, and is quite common, so definitely recommended. Unfortunately, there seems to be another tool with the same name. You'll want the v0.13 version by Markus, not the other one.

Personally, I use Usenti, which is my own tool. This is a bitmap editor (paint program) with exporting options thrown in. It allows different file-types, different bitdepths, different output files, all modes, some map-exporting stuff, meta-tiling, compression and a few others. It may not be as powerful as big photo-editing tools as Photoshop, GIMP, Aseprite, and the like, but it gets the job done. If you're still drawing your graphics with Microsoft Paint, please stop that and use this one instead. The exporter is also available separately in the form of the open source project called (win)grit, which comes in a command-line interface (grit) and a GUI (wingrit). As of January 2007, it is also part of the devkitPro distribution.

BITMAP CONVERSION VIA CLI

There are many command-line interfaces available for graphics conversion, but to make them function you need the correct flags. Here are examples for gfx2gba and grit, converting a bitmap *foo.bmp* to a C array for modes 3, 4 and 5. This is just an example, because this is not the place for a full discussion on them. Look in their respective readme's for more details.

```
# gfx2gba
# mode 3, 5 (C array; u16 foo_Bitmap[]; foo.raw.c)
    gfx2gba -fsrc -c32k foo.bmp
# mode 4 (C array u8 foo_Bitmap[], u16 master_Palette[];
foo.raw.c, mastel.pal.c)
    gfx2gba -fsrc -c256 foo.bmp
```

```
# grit
# mode 3, 5 (C array; u32 fooBitmap[]; foo.c foo.h)
    grit foo.bmp -gb -gB16
# mode 4 (C array; u32 fooBitmap[], u16 fooPal[]; foo.c foo.h)
    grit foo.bmp -gb -gB8
```

Below, you can see a partial listing of modes.c, which contains the bitmap and the palette used in the *bm_modes* demo discussed at the end of this section, as exported by Usenti. It is only a very small part of the file because at over 2700 lines it is way too long to display here, which wouldn't serve much of a purpose anyway. Note that both

are `u32` arrays, rather than the `u8` or `u16` arrays you might encounter elsewhere. What you need to remember is that **it doesn't matter** in what kind of an array you put the data: in memory it'll come out the same anyway.

Well, that's not *quite* true. Only with `u32` arrays is proper data alignment guaranteed, which is a good thing. More importantly, you have to be careful with the byte-order of multi-byte types. This is called the endianness of types. In a *little endian* scheme, least significant bytes will go first and in a *big endian*, most significant bytes will go first. See table 2 for an example using `0x01`, `0x02`, `0x03` and `0x04`. The GBA is a little endian machine, so the first word of the `modesBitmap` array, `0x7FE003E0` is the halfwords `0x03E0` (green) followed by `0x7FE0` (cyan). If you want more examples of this, open up VBA's memory viewer and play around with the 8-bit, 16-bit and 32-bit settings.

big u32		0x01020304		
big u16		0x0102		0x0304
u8	0x01	0x02	0x03	0x04
little u16		0x0201		0x0403
little u32		0x04030201		

Table 5.2: Big endian vs little endian interpretation of byte-sequence 01h, 02h, 03h, 04h

The key point here: the data itself doesn't change when you use different data-types for the arrays, only the way you *represent* it does. That was also the point of the *bm_modes* demo: it's the same data in VRAM all the time; it's just used in a different way.

```
//=========================================================================
//
//  modes, 240x160@16,
//  + bitmap not compressed
//  Total size: 76800 = 76800
//
//  Time-stamp: 2005-12-24, 18:13:22
//  Exported by Cearn's Usenti v1.7.1
//  (comments, kudos, flames to "daytshen@hotmail.com")
//
//=========================================================================

const unsigned int modesBitmap[19200]=
{

0x7FE003E0,0x7FE07FE0,0x7FE07FE0,0x7FE07FE0,0x7FE07FE0,0x7FE07FE0,0x7FE0
7FE0,0x7FE07FE0,

0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F
080F,0x080F080F,

0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F
080F,0x080F080F,

0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F
080F,0x080F080F,
// ...
// over 2500 more lines like this
// ...

0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F
080F,0x080F080F,

0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F
080F,0x080F080F,

0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F080F,0x080F
080F,0x080F080F,

0x7FE07FE0,0x7FE07FE0,0x7FE07FE0,0x7FE07FE0,0x7FE07FE0,0x7FE07FE0,0x7FE0
7FE0,0x7FE07FE0,
};

const unsigned int modesPal[8]=
{

0x7FE07C1F,0x03FF0505,0x03E00505,0x7C000000,0x0000080F,0x00000000,0x0000
```

```
0000,0x080F0000,
};
```

Those 2700 lines represent a 77 kB bitmap. One *single* bitmap. In all likelihood, you'll need at least a couple of them to make anything worthwhile. Most games have lots of data in them, not only graphics but maps and sound and music as well. All this adds up to a huge amount of data, certainly too much for just EWRAM and maybe even for a full cart. That is why **compression** is also important. The GBA BIOS has decompression routines for bit-packing, run-length encoding, LZ77 and Huffman. Converters sometimes have the appropriate compressors for these routines, which can drastically shrink the amount of memory used. Usenti and (win)grit support these compressors. So does gfx2gba, which even has some more. A tool that just does compression on binary files (but does it very well) is GBACrusher. I won't go into compression that much (or at all), but you can read up on the subject here.

> **UNDERSTANDING DATA**
>
> It is vital that you understand what data is, how the different datatypes work. Preferably endianness and alignment too. Emulators and hex editors can help you with this. Once you have compilation working, just make a few random arrays and see what they look like in the VBA memory viewer for a while.

`#include` code or data considered harmful

Most non-trivial projects will have multiple files with code and data. The standard way of dealing with these is to compile these separately and then link the results to the final binary. This is the recommended strategy. However, most other tutorials and many of the example code you can find on the web do something else: a unity build.

Fig 5.6: even Bart knows …

They `#include` everything into the main source file and compile that. This is *not* a recommended practice and should be avoided.

"But why not? It seems to work fine, and it's so easy!"

Yes, it is easy; and it does seem to work. The main problem is that a unity build isn't scalable. For small projects (a handful of files) you probably won't notice, but as projects grow to hundreds and perhaps thousands of files, you will run into some very annoying problems. The main issue is what #include actually does. It copies the whole included file into the includer to form a single larger file. This leads to the following issues.

- **Massive files to compile**. So, `#include` creates one big file. If you have a lot of stuff, you'll have one *very* big file. This will cost large amounts of memory and slows down compilation. As the project grows, what starts as a compile time of a second can grow to several, then minutes and perhaps even hours.

  At some point, there was also the problem that the compiler couldn't handle files exceeding 4 MB, putting a limit on how much you could `#include` in a C file. I'm not sure if this is still an issue.

- **Recompiling the world**. The main problem is that when you `#include` everything, you need to recompile everything as well. If you make one change *anywhere*, no matter how small, causes *everything* to be compiled. For small projects (say, a handful of files), a full rebuild would take a few seconds so it's not a problem. But larger projects can have hundreds or thousands of files, and the time is not measured in seconds, but in minutes or perhaps hours. Sure it's a good excuse to go sword fighting, but terribly annoying if you want to do something productive.

- **Bloat**. Even if your own code and data are relatively small in number, you're probably using some code library for API functions. Normally, these are pre-compiled and only the functions used are linked into your binary. But if those worked by #include as well (in other words, if their creators had followed the

practice I'm warning against), every function in that library would be included as well, including the ones you're not using. This increases the filesize, *and* increases the problems mentioned above.

- **Undeclared identifiers, multiple definitions and circular dependencies**. In a nutshell, C requires that you declare an identifier before it's referenced, and it can only be defined once. The first point means that the order of inclusions starts to matter: if, say, *fileB.c* needs something from *fileA.c*, the latter needs to be included before the former to get a compile. The second means that you could only `#include` a file once in the whole project: if *fileB.c* and *fileC.c* both need stuff from *fileA.c*, you can't `#include` it in them both because when they're `#include` d in *main.c*, *fileA.c* is effectively `#include` d twice and the compiler will balk.

  These points can technically be overcome by being careful, such as using an include guard. But, again, when projects grow, things can get increasingly more difficult to keep track of which comes before what and why. There is, however, one point at which it *will* go wrong, namely when there are circular dependencies: *fileB.c* needs *fileA.c* and vice versa. Each file would require the other to go first, which simply isn't possible because it'd cause multiple definitions.

- **Data alignment**. I'll get to what this means in a minute, but right now know that copy routines work better if the data is aligned to 32-bit boundaries (even for byte and halfword arrays). Some of them won't even work properly if this isn't the case. This is usually guaranteed if you compile separately, but if the arrays are #included and no steps have been taken to force alignment, you simply never know.

  It's not much of a problem nowadays because most graphics converters force data alignment, but you still need to know about it. Because data alignment is a fairly esoteric concept, it's next to impossible to track down unless you're aware of the problems it can bring.

So please, do yourself a favor and do not `#include` every file you have into *main.c* or its counterpart in your project. Put function and variable definitions in separate source files to be compiled separately and linked later. The `#include` directive is only to be used for files with preprocessor directives, declarations, type definitions, and `inline` functions.

Proper build procedure

Separate compilation

So what do you do instead? Well, for starters keep all the code and data in separate source files. Compile these separately by invoking gcc on each file. This gives you a list of object files. These you then link together. In batch files, you'd need to add extra commands for each file, but a properly setup makefile uses a list of object files, and the makefile's rules will then take care of the rest automatically. Using the makefile of the second demo as a reference, you'll get something like this:

```
# partial makefile for using multiple source files
# some steps omitted for clarity

# 3 targets for compilation
OBJS     := foo.o bar.o boo.o

# link step: .o -> .elf
$(PROJ).elf : $(OBJS)
    $(LD) $^ $(LDFLAGS) -o $@

# compile step .c -> .o
$(OBJS) : %.o : %.c
    $(CC) -c $< $(CFLAGS) -o $@
```

The `OBJS` variable contains the names of three object files, which would be the targets of compiling *foo.c*, *bar.c* and *boo.c*. Remember, makefiles list rules by target, not by prerequisite. The compilation step uses a static pattern rule, which for each object file (`.o`) in `OBJS` compiles the source file (`.c`) file with the same title. This is what runs the compiler for our three source files. In the linking step, the automatic

variable `$^` expands to the prerequisites of the rule, which is the list of all object files, and this is how the files are all linked together. If you need more files, add them to the `OBJS` list.

Note that the devkitARM and tonc template files take care of these things automatically. Just put the source files into the right directory and you're good to go.

Symbols, declarations and definitions

If you have been doing everything via `#include`, you should consider refactoring all of your stuff to separate source files. No, let me rephrase that, you *need* to do this because you'll benefit from it in the end. If you're already well in your project, this is going to suck because it's boring and time consuming and most likely it's not even going to *work* properly when you try the first build afterwards. I expect you'll get a whole slew of errors, particularly these three:

- `foo' undeclared
- redefinition of `foo'
- multiple definition of `foo'

To understand what these mean, you need to know a little bit more about how C (and indeed programs) actually works.

As I said before, there aren't really things like programs, bitmaps, sound on computers; it's all just bits. Bits, bits and more bits. What makes a sequence of bits work as a program is the way it is fed to the CPU, VRAM, and other sections. Somewhere in the build process, there has to be a translation of all the C code to data and machine instructions. This, of course, is the compiler's job.

But wait, there's more. C allows you to compile each file separately, and then link them later into the actual program. This is a good idea, because it allows you to save time by only compiling the files that you have recently modified, as well as the use of code *libraries*, which are little more than a bunch of precompiled source files. If you're not convinced that this is a good idea, consider what it would take without it. You'd

have to have *all* the source code that you wanted to use (including for things like `printf()` and all the API code), and compile all those megabytes of source files each time. Sounds like fun? No, I didn't think so either.

However, you need a little more bookkeeping to make this all work. Because everything is just bits, you'd need a way to find out where the function or data you want to use actually is. The contents of the compiled files (the object files) isn't just raw binary, it contains **symbols**. This is just a word for the group of things that have actual binary information attached to them. Among other things, the object file keeps track of the symbol's name, section, size, and where its content is in the object file. A function is a symbol, because it contains instructions. A variable is also a symbol, as is data for bitmaps, sound, maps et cetera. Preprocessor `#define`s, `typedef`s and `struct`/`class` declarations are *not* symbols, because they only don't have actual content in them, but allow you to structure your code better.

The other bookkeeping note is that each source/object file is a separate entity. In principle, it knows nothing about the outside world. This makes sense because it limits the dependency on other files, but it does create a little problem when you want to make files work together. This is where **declarations** come in.

You may have noticed that C is pretty strict when it comes to names of stuff. Before you can use anything, it requires you to mention what it is beforehand. For example, if you use a function `foo()` in your code and you never defined its code, or even if you put it after the call to `foo()`, the compiler will complain that it doesn't know what you're talking about. That is, it will say that '`foo' is undeclared'. You have to admit it has a right to stop there: how would it know how to use the thing if you never told it what it was?

The code snippet below gives an example of when a reference is and is not declared, and why it's important to have a declaration. Function `a()` calls `foo()`, which is not known at the time, so an error is produced. Function `b()` also calls `foo()`, which *is* known at that time, but still gives an error because `foo()` just happens to require an integer as an argument. If the declaration wasn't mandatory and the call in `a()` was

allowed, `foo()` would have been processing the wrong kind of information at runtime. There are ways around such problems, of course, languages like PHP, VB and others work fine without mandatory declarations, but the cost for that is speed and possibly a lot more runtime errors.

```c
//# C requires identifiers to be declared or defined before first use.

// ERROR: `foo' is undefined.
void a()
{
    foo();
}

// Definition of foo(). Now the system 'knows' what foo is.
void foo(int x)
{
    // code
}

// foo is known and used correctly: no errors.
void b()
{
    foo(42);
}

// foo is known but used incorrectly. Compiler issues error.
void c()
{
    foo();
}
```

Now back to our separate files, and the difference between declarations and definitions of symbols. A *definition* is something of actual content: it is what actually forms the symbol. Examples are the value(s) in variables, and the code in functions. A *declaration* is just an empty reference. It just says that there is *something* in the project with a particular name, and indicates how that something is supposed to be used: whether it's a function or variable, what datatype, which arguments, that sort of things. This is how you can use symbols from other object files.

You should be familiar with what a definition looks like. A declaration looks very similar. The basic variable declaration is the variable name and attributes (type, const,

section) preceded by `extern` . For functions, replace the code block by a semi-colon. You can also add `extern` there, but it's not required.

```c
// -----------------------------------------------------------------
// DECLARATIONS. Put these in source (.c) or header (.h) files.
// -----------------------------------------------------------------
extern int var;
extern const unsigned int data[256];
void foo(int x);


// -----------------------------------------------------------------
// DEFINITIONS. Put these in source (.c) only.
// -----------------------------------------------------------------

// uninitialized definition
int var;

// initialized definition
const unsigned int data[256]=
{
    // data
};

void foo(int x)
{
    // code
}
```

Now, a definition is also a declaration, but this does *not* work the other way. How can it, the declaration is supposed to be empty. The distinction is subtle, but it's the reason you might get multiple definition errors when linking the files together. Think of what would happen if you have the definition of function `foo()` in multiple files. Each file itself would know what `foo()` is because definitions are also declarations, so it would pass the compilation stage. So now you have multiple object files, each containing a symbol called `foo` . But then you try to link them into one file. The linker sees different versions of `foo` , and stops because it doesn't know which one you are actually trying to use. The moral here is that you can have as many declarations as you want, but there can be only *one* definition.

Another point I should raise is that the declaration defines how a symbol is to be dealt with, as it is the only point of reference if the definition is in another file. This means that, in theory, you could have a variable `var` defined as an `int`, but declared as a `short`, or even a function! While not exactly recommended, but it is an interesting item.

Lastly: the distinction of what should go in source files, and what in headers. Source files can actually contain anything, so that's an easy one. Remember that they will contain everything after the preprocessor step anyway, because that's what `#include` really does. So what matters is what you put in headers. The purpose of header files is to have a place for all the **non-symbol** stuff that you want to use in different source files. That means declarations, `#define`s, macros, `typedef`s, `struct`/`class` descriptions. It also means `static inline` functions, because these don't form symbols either, but are integrated in the functions that call them.

Summary

All this stuff about separate compilation, declarations, and definitions is rather important for C programming, but the preceding text may be a little much to take in at once. So here is a summary of the most important points.

- **Symbols**. Symbols are those parts of the code that form actual binary content in the final program. This includes functions, variables, data, but not preprocessor or type description stuff.
- **Declarations/definitions**. A definition of a symbol is where the actual content is. A declaration just says that something of a certain name exists, but will be added to the project later. Multiple (identical) declarations may exist, but there can be only one definition in the project. Definitions are also declarations.
- **Source/object files are selfcontained entities**. They contain the definitions of the symbols that are in the code, and a list of references to outside symbols, as indicated by the declarations.
- **Header files contain meta-data, not symbols**. Header files cannot be compiled, but are intended contain the 'glue' that allow difference sources to work

together (i.e., declarations) and stuff that makes writing the sources easier (like `#define` s and macros). They are meant to be included in multiple files, so they cannot create symbols because that would lead to multiple definitions.

Potential problems during compilation or linking:

- **`foo' undeclared**. Compiler error. The identifier `foo` is not known at this point. Check the spelling, or add the appropriate declaration or header file containing the declaration.
- **redefinition of `foo'**. Compiler error. The identifier as a previous declaration or definition conflicting with the current one in the same file or included headers. Usually accompanied by a message of the previous definition.
- **multiple definition of 'foo'**. Linker error. The symbol name `foo` is shared by multiple object files. Replace all but one definitions of `foo` in the source files with the appropriate declarations. Usually accompanied with a message indicating the object file with the other definition(s).

Data alignment

Data alignment is about the 'natural' memory addresses of variables. It is often beneficial to have a variable of a certain length to start at an address divisible by that length. For example, a 32-bit variable likes to be put at addresses that are a multiple of 4. Processors themselves also have certain preferred alignments. Addressing will work faster if you stick to their native types and alignment (say, 32-bit everything for 32-bit CPUs). For PCs it is not required to do any of this, it'll just run slower. For RISC systems, however, things *must* be aligned properly or data gets mangled.

In most cases, the compiler will align things for you. It will put all halfwords on even boundaries and words on quad-byte boundaries. As long as you stick to the normal programming rules, you can remain completely oblivious to this alignment stuff. Except that you *won't* always stick to the rules. In fact, C is a language that allows you to break the rules whenever you feel like it. It trusts you to know what you're doing. Whether that trust is always justified is another matter `:P`

The best example of breaking the rules is pointer casting. For example, most graphics converters will output the data as `u16` arrays, so you can copy it to VRAM with a simple `for` loop. You can speed up copying by roughly 160% if you copy by words (32-bit) rather than halfwords (16-bit). Run the *txt_se2* demo and see for yourself. All you have to do for this is one or two pointer casts, as shown here.

```
#define fooSize ...
const u16 fooData[]= { ... };

// copy via u16 array (the de facto standard)
u16 *dst= (u16*)vid_mem, *src= (u16*)fooData;
for(ii=0; ii<fooSize/2; ii++)
    dst[ii]= src[ii];

// copy via u32 array (mooch faster)
u32 *dst= (u32*)vid_mem, *src= (u32*)fooData;
for(ii=0; ii<fooSize/4; ii++)
    dst[ii]= src[ii];
```

Both these routines copy `fooSize` bytes from `fooData` to VRAM. Only the second version is much faster because there are half as many loop iterations and also because the ARM CPU is just better at dealing with 32-bit chunks. The only danger here is that while `fooData` will be halfword aligned, it need *not* be word aligned, which is a requirement for the second version. For those readers that think casts like this and mis-alignment only happen to other people, think again: the faster copy routines (`memcpy()`, `CpuFastSet()`, and DMA too) cast to word pointers implicitly. Use them (and you should) and you run the risk of misalignment.

There are many ways of ensuring proper alignment. The easiest way is to not mix converted data with the rest of your stuff. That is, don't #include data-files. This should suffice. Another method is to convert to `u32` arrays in the first place. In assembly files, you can control alignment by using the `.p2align *n*` directive, where *n* aligns to 2^n bytes. C itself doesn't allow manual alignment, but there is an extension for this in GCC: `__attribute__(( aligned(4) ))`. Add that to the definition and it'll be word aligned. This is often `#define`d as `ALIGN4` in some headers. Files in GBFS are also always correctly aligned.

Struct alignment

One other area where alignment can cause problems is in `struct` definitions. Look at the following code. Here we have a `struct` named `FOO` consisting of one byte, *b*, one word *w* and one halfword *h*. So that's 1+4+2=7 bytes for the `struct` right? Wrong. Because of the alignment requirement, *w* doesn't immediately follow *b* but leaves 3 bytes of padding. When defining arrays of this type, you'll also see that there are also two padding bytes after *h*, because otherwise later array entries would run into trouble.

```
// one byte, one word, one halfword. 7 byte struct?
// Well let's see ...
struct FOO
{
     u8 b;
    u32 w;
    u16 h;
};

// Define a FOO array
struct FOO foos[4]=
{
    { 0x10, 0x14131211, 0x1615 },
    { 0x20, 0x24232221, 0x2625 },
    { 0x30, 0x34333231, 0x3635 },
    { 0x40, 0x44434241, 0x4645 },
};

// In memory. 4x12 bytes.
// 10 00 00 00 | 11 12 13 14 | 15 16 00 00
// 20 00 00 00 | 21 22 23 24 | 25 26 00 00
// 30 00 00 00 | 31 32 33 34 | 35 36 00 00
// 40 00 00 00 | 41 42 43 44 | 45 46 00 00
```

The *real* size is actually 12 bytes. Not only is this almost twice the size, if you ever try to copy the array using a hard-coded 7 rather than `sizeof(struct FOO)`, you completely mess it up. Take this lesson to heart. It's a very easy mistake to make and difficult to detect after the fact. If you were unaware of this fact and you've already done some GBA coding, check your `struct` (or `class`) declarations now; chances are there are gaps that shouldn't be there. Simply rearranging some of the members

should suffice to make them fit better. Note that this is not specific to the GBA: `struct` s on PCs may behave the same way, as I noticed when I was writing my TGA functions.

There are ways of forcing packing, using the ' `__attribute__((packed))` ' attribute. If `struct FOO` had that, it really would be 7 bytes long. The downside of this is that the non-byte members could be mis-aligned, and the compiler emits code to put the value together byte for byte. This is very much slower than the non-packed version, so only use this attribute if you have no other choice. What happens with mis-aligned (half)words then I can't tell you though, but I'm sure it's not pretty.

> **FORCING ALIGNMENT AND PACKING**
>
> GCC has two attributes that allow you to force alignment of arrays, and remove member-alignment in `struct` s.
>
> ```
> // Useful macros
> #define ALIGN(n)    __attribute__((aligned(n)))
> #define PACKED      __attribute__((packed))
>
> // force word alignment
> const  u8 array[256] ALIGN(4) = {...};
> typedef struct FOO {...} ALIGN(4) FOO;
>
>
> // force struct packing
> struct FOO {...} PACKED;
> ```

Devkits and struct alignment

As far as I've been able to tell, `struct` s have always had word alignment. This was useful because it made copying `struct` s faster. C allows you to copy structs with a single assignment, just like the standard data types. Because of the word-alignment, these copies are fast because GCC will make use of ARM's block-copy instructions, which are much faster than copying member by member.

However, this does not seem to be true under devkitARM r19 (and presumably higher) anymore. The new rule seems to be that `struct`s are aligned to their largest member. This does make more sense as a struct of two bytes would actually be two bytes long. However, it does mean that GCC will now call `memcpy()` for non-aligned `struct`s. Apart from it being a function with quite a bit of overhead (i.e., it's *very* slow if you want to copy a single small `struct`), it will actually **fail** to produce correct results in some cases. The problem is that low-length copies it will copy by the byte, which is something you cannot do for VRAM, OAM, or the palette. For example, objects that we'll see later use a `struct` of four halfwords; using a `struct` copy there, something I am very fond of doing, screws up everything. The only way to make it work properly is to force word alignment on the `struct`.

```
// This doesn't work on devkitARM r19 anymore
typedef struct OBJ_ATTR
{
    u16 attr0, attr1, attr2;
    s16 fill;
} OBJ_ATTR;

OBJ_ATTR a, b;
b= a;    // Fails because of memcpy

// Forcing alignment: this works properly again
typedef struct OBJ_ATTR
{
    u16 attr0, attr1, attr2;
    s16 fill;
} ALIGN(4) OBJ_ATTR;

OBJ_ATTR a, b;
b= a;    // No memcpy == no fail and over 10 times faster
```

FORCING STRUCT-ALIGNMENT IS A GOOD THING

The rules for `struct` alignment have changed since devkitARM r19. Instead of being always word-aligned, they are now aligned as well as their members will allow. If this means they're not necessarily word-aligned, then they will use `memcpy()` for `struct` copies, which is slow for small structs, and may even be

> wrong (see next section). If you want to be able to do `struct` copies fast and safe, either force alignment or cast to other datatypes.

Copying, memcpy() and sizeof

There are many different ways of copying data on this platform. Arrays, `struct` copies, standard copiers like `memcpy()`, and GBA specific routines like `CpuFastSet()` and DMA. All of these have their own strengths and weaknesses. All of them can be affected by misalignment and the no-byte-write rule. I discuss some of them in the txt_se2 demo.

I've chosen to use `memcpy()` in the early demos for a number of reasons. The main one is that it is part of the standard C library, meaning that C programmers should already be familiar with it. Secondly, it is somewhat optimized (see the txt_se2 demo for details). However, there are two potential pitfalls with the routine. The first is data alignment (yes, *that* again). If *either* the source *or* the destination is not word-aligned, you're in trouble. Secondly, if the number of bytes is too small, you're in trouble too.

Both of these have to do with the basic function of `memcpy()`, namely to be a fast *byte* copier. But as you know, you can't copy single bytes to VRAM directly. Fortunately, it has an optimised mode that uses an unrolled word-copy loop if two conditions are satisfied:

1. When both source and destinations are word aligned.
2. When you are copying more than 16 bytes.

This is usually the case so I figured it'd be safe enough for the demos. There are also look-alikes in libtonc that do the same thing only better, namely `memcpy16()` and `memcpy32()`, but these are in assembly so I thought I wouldn't lay them on you so soon. Highly recommended for later though.

On a related subject, there is also `memset()` for memory fills. Be careful with that one, because that will *only* work with bytes. Tonclib also includes 16- and 32-bit versions of

this routine, but also in assembly.

The last thing I want to discuss is the `sizeof()` operator. In other tutorials you will see this being used to find the size in bytes of arrays, which is then used in `memcpy()`. It's a good procedure but will not always work. First, `sizeof()` actually gives the size of the *variable*, which need not always be the array itself. For example, if you use it on a pointer to the array, it'll give the size of the pointer and *not* of the array. The compiler never complains, but you might when hardly anything is copied. Secondly, `sizeof()` is an *operator*, not a function. It is resolved at compile-time, so it needs to be able to find the size at that time as well. To do this, either the declaration (in the header) should indicate the size, or the array definition (in the source file) should be visible.

Bottom line: you can use `sizeof()`, just pay attention to what you use it on.

Okay, that was the long and boring –yet necessary– section on data. Congratulations if you've managed to stay awake till this point, especially if you've actually understood all of it. It's okay if you didn't though, in most cases you won't run into the problems discussed here. But just remember this section for if you do run into trouble when copying and you can't find it in the code; it might save you a few hours of debugging.

Data interpretation demo

The *bm_modes* is an example of how the same data can result in different results depending on interpretation (in this case, modes 3, 4 and 5). In the code below, I make *one* copy into VRAM, and switch between the modes using Left and Right. The results can be seen in Fig 5.7a-c.

I've arranged the data of the bitmap in such a way that the name of the current mode can be read clearly, as well as indicated the mode's boundaries in memory. Because the data intended for the other modes is still present, but not interpreted as intended, that part of the bitmap will look a little distorted. And that's partly the point of the demo: when filling VRAM, you need to know how the GBA will use the data in it, and

make sure it'll be used. If the bitmap ends up being all garbled, this is the likely suspect; check the bitdepth, dimensions and format (linear, tiled, compressed, etc) and if something conflicts, fix it.

Now, sometimes this is not as easy as it sounds. The general procedure for graphics is to create it on the PC, then use an exporter tool to convert it to a raw binary format, then copy it to VRAM. If the exporter has been given the wrong options, or if it can't handle the image in the first place, you'll get garbage. This can happen with some of the older tools. In some cases, it's the bitmap editor that is the culprit. For paletted images, a lot depends on the exact layout of the palette, and therefore it is **vital** that you have a bitmap editor that allows total control over the palette, and leaves it intact when saving. Microsoft Paint and Pyxel Edit for example do neither. Even very expensive photo editing tools don't, so be careful.

For this image, I used <plug>my own bitmap editor Usenti</plug>, which not only has some nice palette control options, and tiling functions, but a built-in GBA graphics exporter as well. To make the background be the same color in all modes, the two bytes of the 16-bit background color of modes 3 and 5 had to serve as palette entries for mode 4, both using that 16-bit color again. In this case, the color is `0x080F`, sort of a brownish color. The bytes are 8 and 15, so that's the palette entries where the color goes too. Normally you don't have to worry about switching bitdepths mid-game, but knowing how to read data like this is a useful debugging skill.

```c
#include <string.h>
#include "toolbox.h"
#include "modes.h"

int main()
{
    int mode= 3;
    REG_DISPCNT= mode | DCNT_BG2;

    // Copy the data and palette to the right
    // addresses
    memcpy(vid_mem, modesBitmap, modesBitmapLen);
    memcpy(pal_bg_mem, modesPal, modesPalLen);

    while(1)
    {
        // Wait till VBlank before doing anything
        vid_vsync();

        // Check keys for mode change
        key_poll();
        if(key_hit(KEY_LEFT) && mode>3)
            mode--;
        else if(key_hit(KEY_RIGHT) && mode<5)
            mode++;

        // Change the mode
        REG_DISPCNT= mode | DCNT_BG2;
    }

    return 0;
}
```


Fig 5.7a: bm_modes in mode 3.

Fig 5.7b: bm_modes in mode 4.
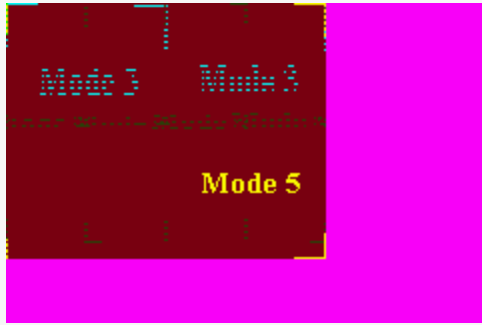
Fig 5.7c: `bm_modes` in mode 5.

Conclusions

Now we've seen some of the basics of the GBA bitmap modes: the properties of modes 3, 4 and 5, page flipping, rudimentary drawing for mode 3 and one of the most important rules of VRAM interactions: you cannot write to VRAM in bytes. There is much more that can be said, of course. Bitmap graphics is a rich subject, but going into more detail right now may not be the best idea. For one, the bitmap modes are very rarely used in games anyway, but also because there are other things to talk about as well. Things like button input, which is what the next chapter is about.

This chapter also discussed a few things about handling data, a very important topic when you're this close to the hardware. Datatypes matter, especially when accessing memory through pointers, and you need to be aware of the differences between them, and the opportunities and dangers of each. Even if you don't remember every little detail in the data section, at least remember where to look when things go screwy.

Before continuing with further chapters, this may be a good time to do some experimenting with data: try changing the data arrays and see what happens. Look at the different data interpretations, different casts, and maybe some intentional errors as well, just to see what kinds of problems you might face at some point. It's better to make mistakes early, while programs are still short and simple and you have less potential problems.

Or not, of course `:P` . Maybe it's worth waiting a little longer with that; or at least until we've covered basic input, which allows for much more interesting things than just passive images.

6. The GBA buttons (a.k.a. keys)

- Introduction
- Keypad registers
- Beyond basic button states
- A simple key demo

Introduction

As you no doubt already know, the GBA has one 4-way directional pad (D-pad); two control buttons (Select and Start); two regular fire buttons (A and B) and two shoulder buttons (L and R), making a total of 10 *keys*. This is all you have in terms of user-GBA interaction, and for most purposes it is plenty. The principles of key-handling are pretty simple: you have one register with the keystates and you see which buttons are pressed based on whether its bits are set or cleared. I will cover this, but I'll also give some more advanced functions that you will probably want to have at some point.

Keypad registers

The keypad register, REG_KEYINPUT

As said, the GBA has ten buttons, often referred to as keys. Their states can be found in the first 10 bits of the `REG_KEYINPUT` register at location `0400:0130h` (a.k.a. `REG_P1`). The exact layout is shown below. I will refrain from giving a bit-by-bit description because it should be quite obvious. The names of the defined constants I use are "`KEY_` *x*", where *x* is the name of the button, in caps.

```
REG_KEYINPUT @ 0400:0130h
```

F E D C B A	9	8	7	6	5	4	3	2	1
–	L	R	down	up	left	right	start	select	B

Checking whether a key is pressed (down) or not would be obvious, if it weren't for one little detail: the bits are *clear*ed when a key is down. So the default state of `REG_KEYINPUT` is `0x03FF`, and not `0`. As such, checking if `key` is down goes like this:

```
#define KEY_DOWN_NOW(key)  (~(REG_KEYINPUT) & key)
```

In case your bit-operation knowledge is a bit hazy (get it cleared up. Fast!), this first inverts `REG_KEYINPUT` to a more intuitive (and useful) 'bit is set when down' setting and then masks it with the key(s) you want to check. Note that `key` can in fact be a combination of multiple keys and the result will be the combination of keys that are actually down.

Key states are inverted

The key bits are low-active, meaning that they are **cleared** when a button is pressed and **set** when they're not. This may be a little counter-intuitive, but that's the way it is.

The key control register, REG_KEYCNT

Just about everything you will ever need in terms of key-handling can be done with `REG_KEYINPUT`. That said, you might like to know there is another key-register for some extra control. The register in question is `REG_KEYCNT`, the key control register. This register is used for keypad interrupts, much like `REG_DISPSTAT` was used for video interrupts. The layout is the same as for `REG_KEYINPUT`, except for the top two bits, see the table below. With `REG_KEYCNT`{14} you can enable the keypad interrupt. The conditions for raising this interrupt are determined by `REG_KEYCNT`{0-9}, which say what keys to watch out for and `REG_KEYCNT`{15}, which state the exact conditions. If this bit is clear, then any of the aforementioned keys will raise the interrupt; if set, then they must all be down for the interrupt to be raised. I wouldn't be surprised if

this is how you can reset most games by pressing Start+Select+B+A. Of course, to make use of this register you need to know how to work with interrupts first.

```
REG_KEYCNT @ 0400:0132h
```

F	E	D C B A	9	8	7	6	5	4	3	2
Op	I	–	L	R	down	up	left	right	start	select

bits	name	define	description
0-9	keys	KEY_x	keys to check for raising a key interrupt.
E	I	KCNT_IRQ	Enables keypad interrupt
F	Op	KCNT_OR, KCNT_AND	Boolean operator used for determining whether to raise a key- interrupt or not. If clear, it uses an OR (raise if any of the keys of bits 0-9 are down); if set, it uses an AND (raise if all of those keys are down).

Beyond basic button states

While checking for the keystate with `KEY_DOWN_NOW()` is nice and simple, there are better and/or more preferable methods of key-state handling. I will discuss two (or three) of them here. First, *synchronous keystates*. This is just a fancy way of reading the key-state at a given point and using that variable, instead of repeated reads of REG_KEYINPUT when you process input. An outshoot of this is *transitional states*, where you track not only the current state, but also the previous one. This lets you test for *changes* in keystates, rather than just the keystates themselves.. Lastly, *tribools*: three-state variables (in this cases −1, 0 and +1) that can be used to simplify direction processing.

Synchronous and transitional key states

The use of `KEY_DOWN_NOW()` is a form of *asynchronous* key handling: you check the state at the time the code needs it. While it works, it's not always the best approach. Firstly, it is less efficient in terms of code because the register is loaded and read every time it is necessary (it's volatile, remember?). A secondary concern is that a simultaneous multi-button tap may not be registered as such because the code reading the button states are a little apart.

But those are just minor concerns; the main issue is that there's just little you can really do with them. You can get the current state, but that's it. As a simple example of why this is insufficient for games, consider (un)pausing a game. This is usually done by pressing Start, and then Start again for unpausing. That's fine until you consider that the game runs faster than you can react (this is a basic fact of life; the only reason you can win games is because the game lets you. Deal), so the `Start` button will be down for multiple frames. With `KEY_DOWN_NOW()`, the game will pause *and* unpause during this time; the state of the game when you finally release the button is essentially random. Needless to say, this is a Bad Thing™.

Enter synchronous states. Simply read the state once, at the beginning of the frame for example, and use that as 'the' state for the whole frame. That takes care of the excess readings of REG_KEYINPUT, and potentially missed simultaneity. For tracking state changes, we also save the state of the previous frame. So at the very least, we need two variables and a function that updates them, and for good measure, some functions that check the states. Because these will be quite small, it makes sense to inline them as well.

```
// === (tonc_core.c) =================================================
// Globals to hold the key state
u16 __key_curr=0, __key_prev=0;
```

```
// === (tonc_input.h) =====================================================
extern u16 __key_curr, __key_prev;

#define KEY_A          0x0001
#define KEY_B          0x0002
#define KEY_SELECT     0x0004
#define KEY_START      0x0008
#define KEY_RIGHT      0x0010
#define KEY_LEFT       0x0020
#define KEY_UP         0x0040
#define KEY_DOWN       0x0080
#define KEY_R          0x0100
#define KEY_L          0x0200

#define KEY_MASK       0x03FF

// Polling function
INLINE void key_poll()
{
    __key_prev= __key_curr;
    __key_curr= ~REG_KEYINPUT & KEY_MASK;
}

// Basic state checks
INLINE u32 key_curr_state()        {    return __key_curr;          }
INLINE u32 key_prev_state()        {    return __key_prev;          }
INLINE u32 key_is_down(u32 key)    {    return  __key_curr & key;   }
INLINE u32 key_is_up(u32 key)      {    return ~__key_curr & key;   }
INLINE u32 key_was_down(u32 key)   {    return  __key_prev & key;   }
INLINE u32 key_was_up(u32 key)     {    return ~__key_prev & key;   }
```

The key states are stored in `__key_curr` and `__key_prev`. The function that updates them is `key_poll()`. Note that this function already inverts REG_KEYINPUT, so that the variables are active high, which makes later operations more intuitive. For example, to test whether A is currently down (pressed), just mask `__key_curr` with `KEY_A`, the bit for A. This is what `key_is_down()` does. While `KEY_DOWN_NOW()` gives (almost) the same answer, I would still recommend using `key_is_down()` instead.

INVERT REG_KEYINPUT READS AS SOON AS POSSIBLE

The things that you might check the keystates for are simply easier in active-high settings. Therefore, it is a good idea to make the keystate variables work

> that way.

Transitional states

Back to the pause/unpause issue. The nasty behaviour `KEY_DOWN_NOW()` causes is known as *key bounce*. This is because the macro only checks the current state. What you need for proper (un)pausing is something that checks whether a key is *going* down, rather than just down: you need to check the transition. That's where the previous state comes in. When a key is hit, i.e., the moment of it going down, it will be pressed in the current state, but not the one before. In other words, the keys that are 'hit' are down currently, and not before: `__key_curr&~__key_prev`. After that, checking for a particular key can be achieved with a simple mask as usual. This is done by `key_hit()`.

That's really all there is to it, and you can create similar functions to check for releases (before AND NOT now), if it is held (before AND now), et cetera. Again, it all seems so simple because the states were already inverted; when I first made these functions, I had a terrible time figuring out what the right bit-ops were because the active-low logic was throwing me off. Well okay, not *really* but it would have been easier if I had them inverted from the start.

```c
// Transitional state checks.

// Key is changing state.
INLINE u32 key_transit(u32 key)
{   return ( __key_curr ^  __key_prev) & key;   }

// Key is held (down now and before).
INLINE u32 key_held(u32 key)
{   return ( __key_curr &  __key_prev) & key;  }

// Key is being hit (down now, but not before).
INLINE u32 key_hit(u32 key)
{   return ( __key_curr &~ __key_prev) & key;  }

Key is being released (up now but down before)
INLINE u32 key_released(u32 key)
{   return (~__key_curr &  __key_prev) & key;  }
```

Key tribool states

This is a little technique taken from the PA_Lib wiki. It isn't so much about keys per se, but a shorthand in how you can use the functions, and you will have to make up for yourself whether what's discussed in this subsection is right for you.

Imagine you have a game/demo/whatever in which you can move stuff around. To make a character move left and right, for example, you might do use something like this.

```
// variable x, speed dx
if(key_is_down(KEY_RIGHT))
    x += dx;
else if(key_is_down(KEY_LEFT))
    x -= dx;
```

Thing moves right, *x* increases; thing moves left, *x* decreases, simple enough. Works fine too. However, and this may just be my ifphobia acting up, it's not very pretty code. So let's see if we can find something smoother.

Take a look at what the code is actually doing. Depending on two choices, the variable is either increased (+), decreased (−), or unchanged (0). That's a pretty good definition of a *tribool*, a variable with three possible states, in this case +1, 0 and −1. What I'm after is something that lets you use these states to do the following.

```
 x += DX*key_tri_horz();
```

I suppose I could just wrap the `if`s in this function, but I prefer to do it via bit operations. All I need to do for this is shift the bits for specific keys down, mask that with one, and subtract the results.

```
// === (tonc_core.h) ===================================================
// tribool: 1 if {plus} on, -1 if {minus} on, 0 if {plus}=={minus}
INLINE int bit_tribool(u32 x, int plus, int minus)
{   return ((x>>plus)&1) - ((x>>minus)&1);   }
```

```
// === (tonc_input.h) =================================================
enum eKeyIndex
{
    KI_A=0, KI_B, KI_SELECT, KI_START,
    KI_RIGHT, KI_LEFT, KI_UP, KI_DOWN,
    KI_R, KI_L, KI_MAX
};

// --- TRISTATES ---
INLINE int key_tri_horz()        // right/left : +/-
{    return bit_tribool(__key_curr, KI_RIGHT, KI_LEFT);  }

INLINE int key_tri_vert()        // down/up : +/-
{    return bit_tribool(__key_curr, KI_DOWN, KI_UP);      }

INLINE int key_tri_shoulder()    // R/L : +/-
{    return bit_tribool(__key_curr, KI_R, KI_L);          }

INLINE int key_tri_fire()        // B/A : -/+
{    return bit_tribool(__key_curr, KI_A, KI_B);          }
```

The inline function `bit_tribool()` creates a tribool value from any two bits in a number (register or otherwise). The rest of the functions listed here use the current keystate and the key-bits to create tribools for horizontal, vertical, shoulder and fire buttons; others can be creates with relative ease. These functions make the code look cleaner and are faster to boot. You will be seeing them quite often.

While the functions mentioned above only use `__key_curr`, it is easy to write code that uses other key-state types. For example, a right-left `key_hit` variant might look something like this:

```
// increase/decrease x on a right/left hit
x += DX*bit_tribool(key_hit(-1), KI_RIGHT, KI_LEFT);
```

It's just a call to `bit_tribool()` with using `key_hit()` instead of `__key_curr`. In case you're wondering what the "−1" is doing there, I just need it to get the full hit state. Remember that −1 is `0xFFFFFFFF` in hex, in other words a full mask, which will be optimized out of the final code. You will see this use of tribools a couple of times as well.

A simple key demo

The *key_demo* demo illustrates how these key functions can be used. It shows a mode 4 picture of a GBA (a 240x160 8bit bitmap); the colors change according to the button presses. The normal state is grey; when you press the key, it turns red; when you release it, it goes yellow; and as long as it's held it's green. Fig 6.1 shows this for the L and B buttons. Here's the code that does the real work:

Fig 6.1: key_demo screenshot, with L and B held.

```c
#include <string.h>

#include "toolbox.h"
#include "input.h"

#include "gba_pic.h"

#define BTN_PAL_ID  5
#define CLR_UP    RGB15(27,27,29)

int main()
{
    int ii;
    u32 btn;
    COLOR clr;
    int frame=0;

    memcpy(vid_mem, gba_picBitmap, gba_picBitmapLen);
    memcpy(pal_bg_mem, gba_picPal, gba_picPalLen);

    REG_DISPCNT= DCNT_MODE4 | DCNT_BG2;

    while(1)
    {
        vid_vsync();
        // slowing down polling to make the changes visible
        if((frame & 7) == 0)
            key_poll();
        // check state of each button
        for(ii=0; ii<KI_MAX; ii++)
        {
            clr=0;
            btn= 1<<ii;
            if(key_hit(btn))
                clr= CLR_RED;
            else if(key_released(btn))
                clr= CLR_YELLOW;
            else if(key_held(btn))
                clr= CLR_LIME;
            else
                clr= CLR_UP;
            pal_bg_mem[BTN_PAL_ID+ii]= clr;
        }
        frame++;
    }

    return 0;
}
```

`BTN_PAL_ID` is the starting index of the palette-part used for the buttons and `CLR_UP` is a shade of grey; the rest of the colors should be obvious. To make sure that you can actually see the changes in button colors I'm only polling the keys once every 8 frames. If I didn't do that, you'll hardly ever see a red or yellow button. (By the way, I don't actually change the buttons' colors, but only the palette color that that button's pixels use; palette animation is a Good Thing™).

7. Sprite and background overview

Sprites and backgrounds introduction

Although you can make games based purely on the bitmap modes, you'll find very few that do. The simple reason for this is that all graphics would be rendered by software. No matter how good your code is, that's always going to be a slow process. Now, I'm not saying it can't be done: there are several FPSs on the GBA (Wolfenstein and Doom for example). I *am* saying that unless you're willing to optimize the crap out of your code, you'll have a hard time doing it.

The vast majority uses the GBA's **hardware graphics**, which come in the forms of **sprites** and **tiled backgrounds** (simply "background" or "bg" for short). As I said in the video introduction, a tiled background is composed of a matrix of tiles (hence the name) and each tile contains an index to an 8x8 pixel bitmap known as a **tile**. So what ends up on screen is a matrix of tiles. There are four of these backgrounds with sizes between 128x128 pixels (32x32 tiles) to 1024x1024 pixels (128x128 tiles). Sprites are smaller objects between 8x8 to 64x64 pixels in size. There are 128 of them and you can move them independently of each other. Like backgrounds, sprites are built out of tiles.

The hardware takes care of several other aspects of rendering besides mere raster blasting. For one thing, it uses **color keying** to exclude some pixels from showing up

(i.e., these are transparent). Basically, if the tile's pixel has a value of zero it is transparent. Furthermore, the hardware takes care of a number of other effects like flipping, alpha-blending and affine transformations like rotation and scaling.

The trick is setting up the process. There are three basic steps to be aware of: *control*, *mapping* and *image data*. The boundaries of these steps are a bit vague, but it helps to see it in this manner since it allows you to see sprites and backgrounds as two sides of the same coin. And unification is a Good Thing®. There are still differences, of course, but only in the details.

This page gives a broad overview of what I will talk about in the next couple of pages. Don't worry if you don't understand right away, that's not really the point right now. Just let it seep into your brain, read the other pages and then you'll see what I'm on about here.

Sprite and background control

The first step of rendering is control. Control covers things that act on the sprites or backgrounds as a whole, like activation of the things themselves, whether to use 16 or 256 color tiles, and effects like alpha-blending and transformations. First up is whether or not you want to use the things in the first place. This is done by setting the right bits in the display control register `REG_DISPCNT` . Once you've done that there are further control registers for backgrounds: the `REG_BGxCNT` registers (`0400:0008h` - `0400:000Fh`). For sprites there's the *Object Attribute Memory*, or OAM, which can be found at `0700:0000h` . Each of the 128 sprites has three so-called *attributes* (hence OAM) which covers both the control and mapping aspects of the sprites.

Sprite and background mapping

There's a lot of grey area between control and mapping, but here goes. Mapping concerns everything about which tiles to use and where they go. As said, the screen appearance of both sprites and backgrounds are constructed of tiles, laid out side by side. You have to tell the GBA which tiles to blit to what position. Fig 7.1a-c (below) illustrates this. In fig 7.1b you see the tiles. Note that both sprites and backgrounds have their own set of tiles.

In fig 7.1c you see how these tiles are used. The background uses a *tile-map*, which works just like an ordinary paletted bitmap except that it's a matrix of *screenblock entries* (with tile-indices) instead of pixels (containing color-indices). Excuse me, a what?!? Screenblock entry. Yes, I know the name is a bit silly. The thing is that you need keep a clear distinction between the entries in the map (the screenblock entries, *SE* for short) and the image-data (the actual tiles). Unfortunately, the term "tile" is often used for both. I'll stick to tiles for the actual graphical information, and since the tile-map is stored in things called screenblocks, screenblock entries or SE for the map data. Anyway, each SE has its own tile-index. It also contains bits for horizontal and vertical flipping and, if it's a 16-color background, an index for the palbank as well. In fig 7.1c, you only see the tile-index, though.

For sprites, it's s a bit different, but the basic steps remain. You give *one* tile-index for the whole sprite; the GBA then figures out the other tiles to use by looking at the shape and size of the sprite and the **sprite mapping-mode**. I'll explain what this means later; suffice to say that the mapping mode is either 1D or 2D, depending on `REG_DISPCNT{6}` . In this case, I've used 1D mapping, which states that the tiles that a sprite should use are consecutive. Like backgrounds, there's additional flipping flags and palette-info for 16-color sprites. Unlike backgrounds, these work on the *whole* sprite, not just on one tile. Also, the component tiles of sprites are always adjoining, so you can see a sprite as a miniature tiled-background with some imagination.

What belongs to the mapping step as well is the affine transformation matrix, if any. With this 2x2 matrix you can rotate, scale or shear sprites or backgrounds. There

seems to be a lot of confusion about how this works so I've written a detailed, mathematical description on how this thing works. Bottom line: the matrix maps from screen space to texture-space, and *not* the other way round. Though all the reference documents do state this in a roundabout way, almost every rotation-scale matrix I've seen so far is incorrect. If your code is based on PERN's, chances are yours is too.

Fig 7.1a: 2 sprites on a background.
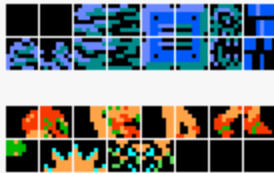
Fig 7.1b: background (above) and sprite (below) tiles.
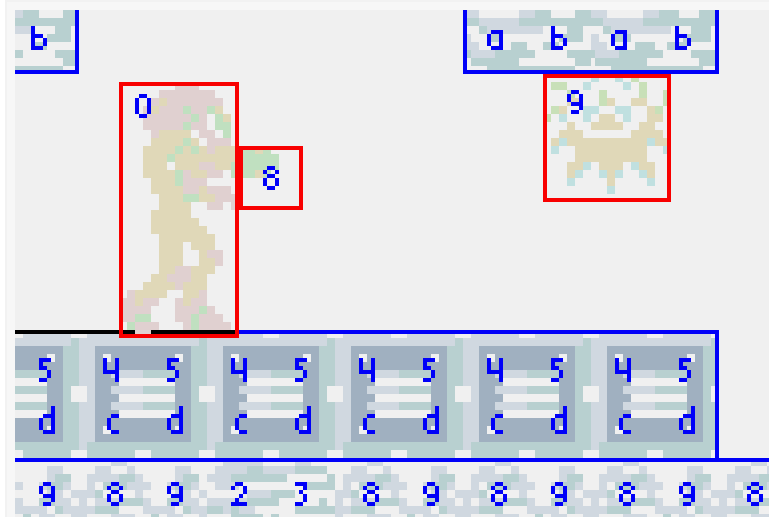
Fig 7.1c: tile usage by bgs and sprites. One tile per SE for bgs, and the top-left tile for sprites. Default tiles (with index 0 are omitted for clarity's sake.

Sprite and background image data

Image data is what the GBA actually uses to produce an image. This means two things: tiles and palettes.

Tiles

Sprites and backgrounds are composed of a matrix of smaller bitmaps called *tile*s. Your basic tile is an 8x8 bitmap. Tiles come in 4bpp (16 colors / 16 palettes) and 8bpp (256 colors / 1 palette) variants. In analogy to floating point numbers, I refer to these as *s-tiles* (single-size tile) and *d-tiles* (double-size tiles). An s-tile is 32 (20h) bytes long, a d-tile 64 (40h) bytes. The default type of tile is the 4bpp variant (the s-tile). If I talk about tiles without mentioning which type, it either doesn't matter or it's an s-tile. Just pay attention to the context.

There is sometimes a misunderstanding about what working in tiles really means. In tiled modes, VRAM is *not* a big bitmap out of which tiles are selected, but a collection of 8x8 pixel bitmaps (i.e., the tiles). It is important that you understand the differences between these two methods! Consider an 8x8 rectangle in a big bitmap, and an 8x8 tile. In the big bigmap, the data after the first 8 pixels contain the next 8 pixels of the same scanline; the next line of the 'tile' can be found further on. In tiled modes, the next scanline of the tile immediately follows the current line.

Basically, VRAM works as an 8×N·8 bitmap in the tiled modes. Because such a small width is impractical to work with, they're usually presented as a wider bitmap anyway. An example is the VBA tile viewer, which displays char blocks as a 256x256 bitmap; I do something similar in fig 7.2a. It is important to remember that these do not accurately mimic the contents of VRAM; to reproduce the actual content of VRAM you'd need something like fig 7.2b, but, of course, no-one is insane enough to edit bitmaps in that manner. In all likelihood, you need a tool that can break up a bitmap into 8x8 chunks. Or restructure it to a bitmap with a width of 8 pixels, which in essence is the same thing.

As with all bitmaps, it is the programmer's responsibility (that means you!) that the bit-depth of the tiles that sprites and backgrounds correspond to the bit-depth of the data in VRAM. If this is out of sync, something like fig 7.2a may appear as fig 7.2c. Something like this is likely to happen sooner or later, because all graphics need to be converted outside of the system before use; one misplaced conversion option is all it takes.
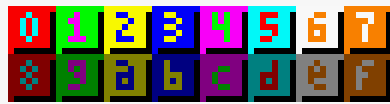
Fig 7.2a: 8bpp tiles.
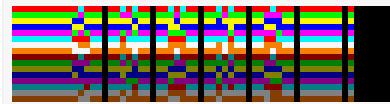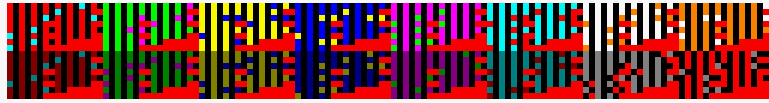
Fig 7.2b: 8bpp tiles as bitmap.

Fig 7.2c: the data of fig 7.2a, interpreted as 4bpp data. If you see something like this (and you will), you now know why.

TILED GRAPHICS CONSIDERATIONS

Remember and understand the following points:

1. The data of each tile are stored sequentially, with the next row of 8 pixels immediately following the previous row. VRAM is basically a big bitmap 8 pixels wide. Graphics converters should be able to convert bigger bitmaps into this format.
2. As always, watch your bitdepth.

TIP FOR GRAPHICS CONVERTERS

If you want to make your own conversion tools, here's a little tip that'll help you with tiles. Work in stages; do *not* go directly from a normal, linear bitmap to writing the data-file. Create a tiling function that takes a bitmap and arranges the tiles into a bitmap 1 tile wide and *H* tiles high. This can then be exported normally. If you allow for a variable tile-width (not hard-coding the 8-pixel width), you can use it for other purposes as well. For example, to create 16x16 sprites, first arrange with width=16, then with width=8.

Tile blocks (aka charblocks)

All the tiles are stored in *charblocks*. As much as I'd like them to be called tile-blocks because that's what they're blocks *of*, tradition has it that tiles are characters (not to be confused with the programming type of characters: an 8bit integer) and so the critters are called charblock. Each charblock is 16kb (4000h bytes) long, so there's room for 512 (4000h/20h) s-tiles or 256 (4000h/40h) d-tiles. You can also consider charblocks to be matrices of tiles; 32x16 for s-tiles, 16x16 (or 32x8) for d-tiles. The whole 96kb of VRAM can be seen as 6 charblocks.

As said, there are 6 tile-blocks, that is 4 for backgrounds (0-3) and 2 for sprites (4-5). For tiled backgrounds, tile-counting starts at a given *character base block* (block for the character base, CBB for short), which are indicated by `REG_BGxCNT` {2-3}. Sprite tile-indexing always starts at the lower sprite block (block 4, starting at `0601:0000h`).

It'd be nice if tile-indexing followed the same scheme for backgrounds and sprites, but it doesn't. For sprites, numbering always follows s-tiles (20h offsets) even for d-tiles, but backgrounds stick to their indicated tile-size: 20h offsets in 4bpp mode, 40h offsets for 8bpp mode.

> **BG VS SPRITE TILE INDEXING**
>
> Sprites always have 32 bytes between tile indices, bg tile-indexing uses 32 or 64 byte offsets, depending on their set bitdepth.

Now, both regular backgrounds and sprites have 10 bits for tile indices. That means 1024 allowed indices. Since each charblock contains 512 s-tiles, you can access not only the base block, but also the one after that. And if your background is using d-tiles, you can actually access a total of four blocks! Now, since tiled backgrounds can start counting at any of the four background charblocks, you might be tempted to try to use the sprite charblocks (blocks 4 and 5) as well. On the emulators I've tested, this does indeed work. On a real GBA, however, it does not. This is one of the reasons why

you *need* to test on real hardware. For more on this subject see the background tile subtleties and the `cbb_demo` .

Another thing you need to know about available charblocks is that in one of the bitmap modes, the bitmaps extend into the lower sprite block. For that reason, you can only use the higher sprite block (containing tiles 512 to 1023) in this case.

Thanks to the wonderful concept of `typedef` s, you can define types for tiles and charblocks so that you can quickly come up with the addresses of tiles by simple array-accesses. An alternative to this is using macros or inline functions to calculate the right addresses. In the end it hardly matters which method you choose, though. Of course, the typedef method allows the use of the `sizeof` operator, which can be quite handy when you need to copy a certain amount of tile. Also, struct-copies are faster than simple loops, and require less C-code too.

```
// tile 8x8@4bpp: 32bytes; 8 ints
typedef struct { u32 data[8];  } TILE, TILE4;
// d-tile: double-sized tile (8bpp)
typedef struct { u32 data[16]; } TILE8;
// tile block: 32x16 tiles, 16x16 d-tiles
typedef TILE  CHARBLOCK[512];
typedef TILE8 CHARBLOCK8[256];

#define tile_mem  ( (CHARBLOCK*)0x06000000)
#define tile8_mem ((CHARBLOCK8*)0x06000000)

//In code somewhere
TILE *ptr= &tile_mem[4][12];   // block 4 (== lower object block), tile
12

// Copy a tile from data to sprite-mem, tile 12
tile_mem[4][12] = *(TILE*)spriteData;
```

Palettes and tile colors

Sprites and backgrounds have separate palettes. The background palette goes first at `0500:0000h` , immediately followed by the sprite palette (`0500:0200h`). Both palettes contain 256 entries of 15bit colors.

In 8-bit color mode, the pixel value in the tiles is palette-index for that pixel. In 4-bit color mode, the pixel value contains the lower nybble of the palette index; the high nybble is the *palbank* index, which can be found in either the sprite's attributes, or the upper nybble of the tiles. If the pixel-value is 0, then that pixel won't be rendered (i.e., will be transparent).

Because of 16-color mode and the transparency issue, it is *essential* that your bitmap editor leaves the palette intact. I know from personal experience that MS-Paint and the Visual C bitmap editor don't, so you might want to use something else. Favorites among other GBA developers are Graphics Gale and GIMP. Of course, since I have my my own bitmap editor, I prefer to use that.

Summary

This is a short list of various attributes of sprites and backgrounds. It's alright if you don't understand it right away; I'll explain in more detail in the following pages.

Subject	Backgrounds	Sprites
Number	4 (2 affine)	128 (32 affine)
Max size	**reg**: 512x512 **aff**: 1024x1024	64x64
Control	`REG_BGxCNT`	OAM
Base tile block	0-3	4
Available tiles *id*s	**reg**: 0-1023 **aff**: 0-255	**modes 0-2**: 0-1023 **modes 3-5**: 512-1023
Tile memory offsets	Per tile size: **4bpp**: start= base + *id**32	Always per 4bpp tile size:

	8bpp: start= base + *id**64	start= base + *id**32
Mapping	**reg**: the full map is divided into map-blocks of 32×32 tegels. (banked map) **aff**: one matrix of tegels, just like a normal bitmap (flat map)	If a sprite is *m* × *n* tiles in size: **1D mapping**: the *m*n* successive tiles are used, starting at *id* **2D mapping**: tile-blocks are 32×32 matrices; the used tiles are the *n* columns of the *m* rows of the matrix, starting at *id*.
Flipping	Each tile can be flipped individually	Flips the whole sprite
Palette	`0500:0000h`	`0500:0200h`

What's in a name?

Well, since you are a programmer you should know the answer: plenty. If you disagree, visit the How To Write Unmaintanable Code website and look at a number of their entries. My naming scheme is a bit different from that of the GBA community. I don't do this just because I feel like being contrary. I find some of the conventional names are incomplete, misleading and ambiguous. I feel little need, at least at present, to follow tradition simply because everyone else does. But you still need to know the traditional names, simply because everyone else does. So here's a list of differences in names.

Subject	Traditional	Tonc
Sprite and bg image data	tiles	tiles
Tile-map entries	tiles (can you feel the confusion?)	screenblock entries / SE
Matrix for transformations	Rot/Scale matrix	affine matrix / **P**
Sprite types	?? vs Rot/Scale	regular vs affine
Background types	text vs rot	regular vs affine
Depository for sprite tiles (`0601:0000`)	`OAMData` (i.e., not the *real* OAM, which is at `0700:0000`)	`tile_mem_obj`
OAM (`0700:0000`)	`OAMData` or `OAMMem`	`oam_mem`

8. Regular sprites

Sprite introduction

According to Webster's, a sprite is "an imaginary being or spirit, as a fairy, elf, or goblin". Right, glad that's cleared up. For games, though, when referring to a sprite one is usually talking about "a [small] animated object that can move freely from the background" (PERN). Primary examples are game characters, but status objects like scores and life bars are often sprites as well. Fig 8.1 on the right shows a sprite of everybody's favorite vampire jellyfish, the metroid. I will use this sprite in the demo at the end of this chapter.

Fig 8.1. Metroid. Rawr.

Sprites are a little trickier to use than a bitmap background, but not by much. You just have to pay a little more attention to what you're doing. For starters, the graphics have to be grouped into 8×8 tiles; make sure your graphics converter can do that. Aside from the obvious actions such as enabling sprites in the display control and loading up the graphics and palette, you also have to set-up the attributes of the sprites correctly in OAM. Miss any of these steps and you'll see nothing. These things and more will be covered in this chapter.

ESSENTIAL SPRITE STEPS

There are 3 things that you have to do right to get sprites to show up:

- Load the graphics and palette into object VRAM and palette.
- Set attributes in OAM to use the appropriate tiles and set the right size.
- Switch on objects in `REG_DISPCNT` , and set the mapping mode there too.

SPRITES AREN'T OBJECTS

Or something like that. I know it sounds weird, but the more I think about it, the more I realize that sprites and objects shouldn't be considered interchangeable. The term 'object', is a hardware feature, controlled in OAM. Right now, I think that 'sprite' is more of a conceptual term, and should be reserved for actors, like playing characters, monsters, bullets, etc. These can in fact be built up of multiple hardware objects, or even use a background.

You could also thing of it in this way: objects are *system* entities linked to the console itself, and sprites are *game* entities, living in the game world. The difference may be subtle, but an important one.

This is merely my opinion, and I can't say how right I am in this. Tonc still switches back and forth between the two words because it's too late to do anything about it now. Mea culpa. I'd love to hear the opinion of others on the subject, so feel free to speak your mind if you want.

Sprite image data and mapping mode

Like I said in the sprite and background overview, sprites are composed of a number of 8×8-pixel mini-bitmaps called tiles, which come in two types: 4bpp (s-tiles, 32 bytes long) and 8bpp (d-tiles, 64 bytes long). The tiles available for sprites are stored in *object VRAM*, or *OVRAM* for short. OVRAM is 32 KiB long and is mapped out by the last

two charblocks of `tile_mem`, which are also known as the lower (block 4, starting at `0601:0000h`) and higher (block 5, `0601:4000h`) sprite blocks. Counting always starts at the lower sprite-block and is *always* done in 32 byte offsets, meaning that sprite-tile #1 is at `0601:0020h`, no matter what the bit depth is (see table 8.1). With 4000h bytes per charblock, a quick calculation will show you that there are 512 tiles in each charblock, giving a total range of 1024. However, since the bitmap modes extend into the lower sprite block, you can only use the higher sprite block (containing tiles 512 to 1023) in modes 3-5.

It may seem that calculating those tile addresses can be annoying, and it would be if you had to do it manually. That's why I have mapped the whole of VRAM with a charblock/tile matrix called `tile_mem`, as discussed in the overview. Need tile #123 of OVRAM? That'd be `tile_mem[4][123]`. Need its address? Use the address operator: `&tile_mem[4][123]`. Quick, easy, safe.

Also, don't forget that the sprites have their own palette which starts at `0500:0200h` (right after the background palette). If you are certain you've loaded your tiles correctly but nothing shows up, it's possible you filled the wrong palette.

memory 0601:	0000	0020	0040	0060	0080	0100	...
4bpp tile	0	1	2	3	4	5	
8bpp tile	0		2		4		

Table 8.1: tile counting for sprites, always per 32 bytes. (You can use odd numbers for 8bpp tiles, but be sure you fill the VRAM accordingly.)

BITMAP MODES AND OBJECT VRAM

Only the higher sprite block is available for sprites in modes 3-5. Indexing still starts at the lower block, though, so the tile range is 512-1023.

The sprite mapping mode

Sprites aren't limited to a single tile. In fact, most sprites are larger (see Table 8.3 for a list of the available sizes for GBA sprites). Larger sprites simply use multiple tiles, but this may present a problem. For backgrounds, you choose each tile explicitly with the tile-map. In the case of sprites, you have two options: 1D and 2D mapping. The default is 2D mapping, and you can switch to 1D mapping by setting `REG_DISPCNT{6}` .

How do these work? Consider the example sprite of fig 8.2a, showing the metroid of fig 8.1 divided into tiles. In 2D mapping, you're interpreting the sprite charblocks as one big bitmap of 256×256 pixels and the sprite a rectangle out of that bitmap (still divided into tiles, of course). In this case, each tile-row of a sprite is at a 32-tile offset. This is shown in fig 8.2b. On the other hand, you can also consider the charblocks as one big array of tiles, and the tiles of every sprite are consecutive. This is shown in fig 8.2c. The numbers in fig 8.2a show the difference between 1D and 2D mapping. Assuming we start at tile 0, the red and cyan numbers follow 2D and 1D mapping, respectively.

From a GBA-programming viewpoint, it is easier to use 1D mapping, as you don't have to worry about the offset of each tile-row when storing sprites. However, actually *creating* sprites is easier in 2D mode. I mean, do you *really* want to edit a bitmap tile by tile? That's what I thought. Of course, it should be the exporting tool's job to convert your sprites from 2D to 1D mapping for you. You can do this with Usenti too.
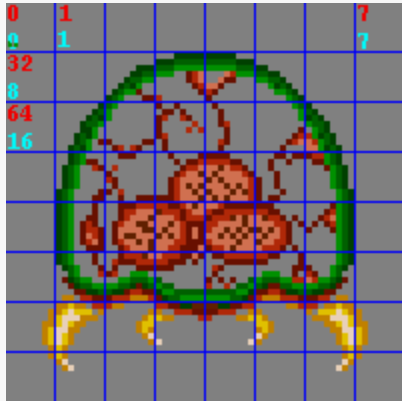
Fig 8.2a: zoomed out version of Fig 8.1, divided into tiles; colored numbers indicate mapping mode: red for 2D, cyan for 1D.
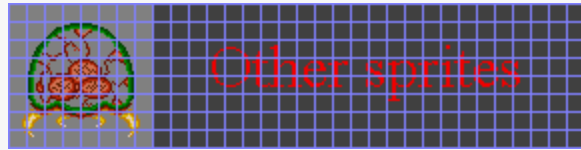
Fig 8.2b: how fig 8.2a should be stored in memory when using 2D mapping.

Fig 8.2c: how fig 8.2a should be stored in memory when using 1D mapping.

OBJECT DATA CONVERSION VIA CLI

Some command-line interfaces can tile bitmaps for use with objects (and tilemaps). In some cases, they can also convert images with multiple sprite-frames to a set of object tiles in 1D object mapping mode. If *foo.bmp* is a 64×16 bitmap with 4 16×16 objects, here's how you can convert it to 8×8 4bpp tiles using gfx2gba and grit (flags for 1D mapping are given in brackets)

```
# gfx2gba
# 4x 16x16@4 objects (C array; u8 foo_Bitmap[], u16
master_Palette[]; foo.raw.c, master.pal.c)
    gfx2gba -fsrc -c16 -t8 [-T32] foo.bmp


# grit
# 4x 16x16@4 objects (C array; u32 fooTiles[], u16 fooPal[];
foo.c, foo.h)
    grit foo.bmp -gB4 [-Mw 2 -Mh 2]
```

Two notes on the 1D mapping flags here. First, gfx2gba can only meta-tile (-T) square objects; for something like 16×8 objects you'd need to do the 1D mapping yourself. Second, grit's meta-tiling flags (-Mw and -Mh) can be anything, and use tile units, not pixels.

SIZE UNITS: TILES VS PIXELS

The default unit for bitmap dimensions is of course the pixel, but in tiled graphics it is sometimes more useful to use tiles as the basic unit, that is, the pixel size divided by 8. This is especially true for backgrounds. In most cases the context will suffice to indicate which one is meant, but at times I will denote the units with a 'p' for pixels or 't' for tiles. For example, a 64x64p sprite is the same as a 8×8t sprite.

Sprite control: Object Attribute Memory

Much unlike in the bitmap modes, you don't have to draw the sprites yourself: the GBA has special hardware that does it for you. This can get the sprites on screen faster than you could ever achieve programmatically. There are still limits, though: there is a limit to the amount of sprite pixels you can cram in one scanline. About 960, if the fora are anything to go by.

So you don't have to draw the sprites yourself; however, you *do* need to tell the GBA how you want them. This is what the *Object Attribute Memory* –OAM for short– is for. This starts at address `0700:0000h` and is 1024 bytes long. You can find two types of structures in OAM: the *OBJ_ATTR* struct for regular sprite attributes, and the *OBJ_AFFINE* struct containing the transformation data. The definitions of these structures can be found below. Note that names may vary from site to site.

```
typedef struct tagOBJ_ATTR
{
    u16 attr0;
    u16 attr1;
    u16 attr2;
    s16 fill;
} ALIGN4 OBJ_ATTR;

typedef struct OBJ_AFFINE
{
    u16 fill0[3];
    s16 pa;
    u16 fill1[3];
    s16 pb;
    u16 fill2[3];
    s16 pc;
    u16 fill3[3];
    s16 pd;
} ALIGN4 OBJ_AFFINE;
```

There are a few interesting things about these structures. First, you see a lot of `fill` er fields. Second, if you would take 4 `OBJ_ATTR` structures and lay them over one `OBJ_AFFINE` structure, as done in table 8.2, you'd see that the fillers of one would exactly cover the data of the other, and vice versa. This is no coincidence: OAM is in fact a weave of `OBJ_ATTR` s and `OBJ_AFFINE` s. Why would Nintendo use a weave instead of simply having one section of attributes and one for transform data? That's a good question and deserves a good answer. When I have one, I'll tell you (I'm guessing it's a data-alignment thing). Also, note that the elements of the `OBJ_AFFINE` are *signed* shorts. I've gone through a world of hurt with the `obj_aff` code because I used `u16` instead of `s16` . With 1024 bytes at our disposal, we have room for 128 `OBJ_ATTR` structures and 32 `OBJ_AFFINE` s. The rest of this file will explain regular sprites that only use `OBJ_ATTR` . I want to give the affine transformation matrix the full mathematical treatment it deserves and will save affine sprites for later.

mem (u16)	0	3	4	7	8	b	c	f
OBJ_ATTR	0 1 2		0 1 2		0 1 2		0 1 2	
OBJ_AFFINE		pa		pb		pc		pd

Table 8.2: memory interlace of OBJ_ATTR and OBJ_AFFINE structures.

> **FORCE ALIGNMENT ON OBJ_ATTRS**
>
> As of devkitARM r19, there are new rules on struct alignments, which means that structs may not always be word aligned, and in the case of `OBJ_ATTR` structs (and others), means that `struct` copies like the one in `oam_update()` later on, will not only be slow, they may actually break. For that reason, I will force word-alignment on many of my structs with `ALIGN4`, which is a macro for `__attribute__((aligned(4)))`. For more on this, see the section on data alignment.

Object attributes: OBJ_ATTR

The basic control for every sprite is the `OBJ_ATTR` structure. It consists of three 16-bit attributes for such qualities as size, shape, position, base tile and more. Each of the three attributes is covered below.

Attribute 0

The first attribute controls a great deal, but the most important parts are for the *y* coordinate, and the shape of the sprite. Also important are whether or not the sprite is transformable (an affine sprite), and whether the tiles are considered to have a bit depth of 4 (16 colors, 16 sub-palettes) or 8 (256 colors / 1 palette).

```
OBJ_ATTR.attr0
```

F E	D	C	B A	9 8	7 6 5 4 3 2 1 0
Sh	CM	Mos	GM	OM	Y

bits	name	define	description

0-7	Y	*ATTR0_Y#*	**Y coordinate**. Marks the top of the sprite.
8-9	OM	ATTR0_REG, ATTR0_AFF, ATTR0_HIDE, ATTR0_AFF_DBL. *ATTR0_MODE#*	**(Affine) object mode**. Use to hide the sprite or govern affine mode. • **00**. Normal rendering. • **01**. Sprite is an affine sprite, using affine matrix specified by `attr1{9-D}` • **10**. Disables rendering (hides the sprite) • **11**. Affine sprite using double rendering area. See affine sprites for more.
A-B	GM	ATTR0_BLEND, ATTR0_WIN. *ATTR0_GFX#*	**Gfx mode**. Flags for special effects. • **00**. Normal rendering. • **01**. Enables alpha blending. Covered here. • **10**. Object is part of the object window. The sprite itself isn't rendered, but serves as a mask for bgs and other sprites. (I think, haven't used it yet) • **11**. Forbidden.
C	Mos	ATTR0_MOSAIC	Enables mosaic effect. Covered here.
D	CM	ATTR0_4BPP, ATTR0_8BPP	**Color mode**. 16 colors (4bpp) if cleared; 256 colors (8bpp) if set.
E-F	Sh	ATTR0_SQUARE, ATTR0_WIDE,	**Sprite shape**. This and the sprite's size (`attr1{E-F}`) determines the sprite's real size, see table 8.4.

ATTR0_TALL. *ATTR0_SHAPE#*	

Two extra notes on attribute 0. First, `attr0` contains the *y* coordinate; `attr1` contains the *x* coordinate. For some reason I keep messing these two up; if you find your sprite is moving left when it should be moving up, this may be why. Second, the affine and gfx modes aren't always named as such. In particular, `attr0{9}` is simply referred to as *the* double-size flag, even though it only works in that capacity if bit 8 is set too. If it isn't, then it hides the sprite. I think that it's actually taken out of the object rendering stage entirely leaving more time for the others, but I'm not 100% sure of that.

shape\size	00	01	10	11
00	8×8	16×16	32×32	64×64
01	16×8	32×8	32×16	64×32
10	8×16	8×32	16×32	32×64

Table 8.3: GBA sprite sizes

Attribute 1

The primary parts of this attribute are the *x* coordinate and the size of the sprite. The role of bits 9 to 13 depend on whether or not this is a affine sprite (determined by `attr0{8}`). If it is, these bits specify which of the 32 `OBJ_AFFINE` s should be used. If not, they hold flipping flags.

OBJ_ATTR.attr1

F E	D	C	B A 9	8 7 6 5 4 3 2 1 0
Sz	**VF**	**HF**	–	**X**
–		**AID**		–

bits	name	define	description
0-8	X	*ATTR1_X#*	**X coordinate**. Marks left of the sprite.

9-D	AID	*ATTR1_AFF#*	**Affine index**. Specifies the `OAM_AFF_ENTY` this sprite uses. Valid *only* if the affine flag (`attr0` {8}) is set.
C-D	HF, VF	ATTR1_HFLIP, ATTR1_VFLIP. *ATTR1_FLIP#*	**Horizontal/vertical flipping** flags. Used *only* if the affine flag (`attr0`) is clear; otherwise they're part of the affine index.
E-F	Sz	*ATTR1_SIZE#*	**Sprite size**. Kinda. Together with the shape bits (`attr0` {E-F}) these determine the sprite's real size, see table 8.3.

I'll say it here too: `attr0` contains *y*, `attr1` contains *x*. Note that bits 12 and 13 have a double role as either flipping flags or affine index. And if you are wondering if you can still flip affine sprites, the answer is yes: simply use negative scales in the matrix.

Attribute 2

This attribute tells the GBA which tiles to display and its background priority. If it's a 4bpp sprite, this is also the place to say what sub-palette should be used.

```
OBJ_ATTR.attr2
```

F E D C	B A	9 8 7 6 5 4 3 2 1 0
PB	Pr	TID

bits	name		description
0-9	TID	ATTR2_ID#	Base **tile index** of sprite. Note that in bitmap modes, this must be 512 or higher.
A-B	Pr	ATTR2_PRIO#	**Priority**. Higher priorities are drawn first (and therefore can be covered by later sprites and backgrounds). Sprites cover backgrounds of the same priority, and for sprites of the same priority, the higher `OBJ_ATTR` s are drawn first.

C-F	PB	ATTR2_PALBANK#	**Palette bank** to use when in 16-color mode. Has no effect if the color mode flag (`attr0` {C}) is set.

Attribute 3

There is *no* attribute 3. Although the `OBJ_ATTR` struct does *have* a fourth halfword, this is only a filler. The memory in that filler actually belongs to the `OBJ_AFFINE` s. Nobody is to mistreat `attr3` in any way … if there's any affine sprite active.

OAM double buffering

You *could* write all your sprite data directly to the OAM at `0700:0000h` , but that might not always be the best move. If it's done during VDraw there's the possibility of tearing. Even worse, you might change the sprite's tile-index in mid-render so that the top is in one animation frame and the bottom is in another. Not a pretty sight. Actually, this isn't something to worry about because you *can't* update OAM during VDraw; it's locked then. What's often done is creating a separate buffer of OAM entries (also known as the **object shadow**) that can be modified at any time, and then copy that to the real OAM during VBlank. Here's my take on this.

```
OBJ_ATTR obj_buffer[128];
OBJ_AFFINE *const obj_aff_buffer= (OBJ_AFFINE*)obj_buffer;
```

I'm using 128 now, but I suppose you could use a lower number if you don't use all the sprites. Anyway, now you have a double buffer for both `OBJ_ATTR` and `OBJ_AFFINE` data, which is available at any given time. Just make sure you copy it to the *real* OAM when the time is right.

Bitfield macros (OAM or otherwise)

Setting and clearing individual bits is easy, but sometimes it's not too convenient to do it all yourself. This is especially true for field of bits like positions or palette banks, which would involve long statements with masks and shifts if you want to do it nicely. To improve on this a little bit, I have a number of macros that may shorted the amount of actual code. There are essentially three classes of macros here, but before I go into that, I have to explain a little bit more about the hashed (*foo*'#") defines in the attribute lists above.

The hash means that for each of these, there will be three `#define`s with *foo* as their roots: *foo* `_MASK` , *foo* `_SHIFT` , and *foo* `(_n)` . These give the bitmask, bitshift and a bitfield set macro for the corresponding type.

For example, the one attached to the tile index, `ATTR2_ID#` . The tile index field has 10 bits and starts at bit 0. The corresponding defines therefore are:

```
// The 'ATTR2_ID#' from the attr2 list means these 3 #defines exist
#define ATTR2_ID_MASK        0x03FF
#define ATTR2_ID_SHIFT           0
#define ATTR2_ID(n)         ((n)<<ATTR2_ID_SHIFT)
```

Most GBA libraries out there have `#define`s like these, albeit with different names. The actual macro isn't 100% safe because it does no range checking, but it's short and sweet. Now, as far as Tonc's text is concerned, every time you see the hash in the define list for a register, it'll have these three `#define`s to go with that name.

I also have a second batch of macros you can use for setting and getting specific fields, which use the mask and shift names explained above. I'll admit the macros look horrible, but I assure you they make sense and can come in handy.

```
// bit field set and get routines
#define BF_PREP(x, name)        ( ((x)<<name##_SHIFT)& name##_MASK  )
#define BF_GET(x, name)         ( ((x) & name##_MASK)>> name##_SHIFT )
#define BF_SET(y, x, name)      (y = ((y)&~name##_MASK) |
BF_PREP(x,name) )

#define BF_PREP2(x, name)       ( (x) & name##_MASK )
#define BF_GET2(y, name)        ( (y) & name##_MASK )
#define BF_SET2(y, x, name)     (y = ((y)&~name##_MASK) | BF_PREP2(x,
name) )
```

Well, I did warn you. The `name` argument here is the *foo* from before. The preprocessor concatenation operator is use to create the full mask and shift names. Again using the tile-index as an example, these macros expand to the following:

```
// Create bitfield:
attr2 |= BF_PREP(id, ATTR0_SHAPE);
// becomes:
attr2 |= (id<<ATTR2_ID_SHIFT) & ATTR2_ID_MASK;

// Retrieve bitfield:
id= BF_GET(attr2, ATTR2_ID);
// becomes:
id= (attr2 & ATTR2_ID_MASK)>>ATTR2_ID_SHIFT;

// Insert bitfield:
BF_SET(attr2, id, ATTR2_ID);
// becomes:
attr2= (attr&~ATTR2_ID_MASK) | ((id<<ATTR2_ID_SHIFT) & ATTR2_ID_MASK);
```

`BF_PREP()` can be used to prepare a bitfield for later insertion or comparison. `BF_GET()` gets a bitfield from a value, and `BF_SET()` sets a bitfield in a variable, without disturbing the rest of the bits. This is basically how bitfields normally work, except that true bitfields cannot be combined with OR and such.

The macros with a '2' in their names work in a similar way, but do not apply shifts. These can be useful when you have already shifted `#define` s like `ATTR0_WIDE` , which can't use the other ones.

```
// Insert pre-shifted bitfield:
// BF_SET2(attr0, ATTR0_WIDE, ATTR0_SHAPE);
attr0= (attr0&~ATTR0_SHAPE_MASK) | (id & ATTR0_SHAPE_MASK);
```

Note that none of these three have anything GBA specific in them; they can be used on any platform.

Finally, what I call my build macros. These piece together the various bit-flags into a single number in an orderly fashion, similar to HAM's tool macros. I haven't used them that often yet, and I'm not forcing you to, but on occasion they are useful to have around especially near initialization time.

```
// Attribute 0
#define ATTR0_BUILD(y, shape, bpp, mode, mos, bld, win)           \
(                                                                 \
    ((y)&255) | (((mode)&3)<<8) | (((bld)&1)<<10) | (((win)&1)<<11) \
    | (((mos)&1)<<12) | (((bpp)&8)<<10) | (((shape)&3)<<14)        \
)

// Attribute 1, regular sprites
#define ATTR1_BUILD_R(x, size, hflip, vflip)           \
( ((x)&511) | (((hflip)&1)<<12) | (((vflip)&1)<<13) | (((size)&3)<<14) )

// Attribute 1, affine sprites
#define ATTR1_BUILD_A(x, size, aff_id)                 \
( ((x)&511) | (((aff_id)&31)<<9) | (((size)&3)<<14) )

// Attribute 2
#define ATTR2_BUILD(id, pbank, prio)                   \
( ((id)&0x3FF) | (((pbank)&15)<<12) | (((prio)&3)<<10) )
```

Instead of doing ORring the bitflags together yourself, you can use these and perhaps save some typing. The order of arguments maybe annoying to remember for some, and the amount of safety checking may be a bit excessive (gee, ya think?!?), but if the numbers you give them are constants the operations are done at compile time so that's okay, and sometimes they really can be helpful. Or not `:P` . Like I said, I'm not forcing you to use them; if you think they're wretched pieces of code (and I admit they are) and don't want to taint your program with them, that's fine.

Note that with the exception of `bpp`, the arguments are all shifted by the macros, meaning that you should *not* use the `#define` flags from the lists, just small values like you'd use if they were separate variables rather than bits in a variable.

Demo time

Now, to actually use the bloody things. The code below is part of the *obj_demo*. It is the most complex I've shown yet, but if you take it one step at a time you'll be fine. Essentially, this demo places the tiles of a boxed metroid in the VRAM allotted for objects and then lets you screw around with various `OBJ_ATTR` bits like position and flipping flags. The controls are as follows:

- Control Pad
  Moves the sprite. Note that if you move far enough off-screen, it'll come up on the other side.
- A and B Buttons
  Flip the sprite horizontally or vertically, respectively.
- Select Button
  Makes it glow. Well, makes it palette-swap, actually. Handy for damage-flashing.
- Start Button
  Toggles between 1D and 2D mapping modes. Fig 8.2b and fig 8.2c should explain what happens. Since the sprite is in 1D mode, there's really not much to see when you switch to 2D mapping, but I had a few buttons to spare, so I thought why not.
- L and R Buttons
  Decreases or increase the starting tile, respectively. Again, I had a few keys to spare.

```
// Excerpt from toolbox.h

void oam_init(OBJ_ATTR *obj, uint count);
void oam_copy(OBJ_ATTR *dst, const OBJ_ATTR *src, uint count);

INLINE OBJ_ATTR *obj_set_attr(OBJ_ATTR *obj, u16 a0, u16 a1, u16 a2);
INLINE void obj_set_pos(OBJ_ATTR *obj, int x, int y);
INLINE void obj_hide(OBJ_ATTR *oatr);
INLINE void obj_unhide(OBJ_ATTR *obj, u16 mode);

// === INLINES ===========================================================

//! Set the attributes of an object.
INLINE OBJ_ATTR *obj_set_attr(OBJ_ATTR *obj, u16 a0, u16 a1, u16 a2)
{
    obj->attr0= a0; obj->attr1= a1; obj->attr2= a2;
    return obj;
}

//! Set the position of \a obj
INLINE void obj_set_pos(OBJ_ATTR *obj, int x, int y)
{
    BF_SET(obj->attr0, y, ATTR0_Y);
    BF_SET(obj->attr1, x, ATTR1_X);
}

//! Hide an object.
INLINE void obj_hide(OBJ_ATTR *obj)
{   BF_SET2(obj->attr0, ATTR0_HIDE, ATTR0_MODE);        }

//! Unhide an object.
INLINE void obj_unhide(OBJ_ATTR *obj, u16 mode)
{   BF_SET2(obj->attr0, mode, ATTR0_MODE);          }
```

```c
// toolbox.c

void oam_init(OBJ_ATTR *obj, uint count)
{
    u32 nn= count;
    u32 *dst= (u32*)obj;

    // Hide each object
    while(nn--)
    {
        *dst++= ATTR0_HIDE;
        *dst++= 0;
    }
    // init oam
    oam_copy(oam_mem, obj, count);
}

void oam_copy(OBJ_ATTR *dst, const OBJ_ATTR *src, uint count)
{

// NOTE: while struct-copying is the Right Thing to do here,
//    there's a strange bug in DKP that sometimes makes it not work
//    If you see problems, just use the word-copy version.
#if 1
    while(count--)
        *dst++ = *src++;
#else
    u32 *dstw= (u32*)dst, *srcw= (u32*)src;
    while(count--)
    {
        *dstw++ = *srcw++;
        *dstw++ = *srcw++;
    }
#endif

}
```

This is the basic utility code for the demo, and contains most of the things you'd actually like to have functions for. Note that the inline functions make good use of the bitfield macros shown earlier; if I hadn't done that, the code would be a good deal longer.

Another point that I need to make is that if I'd put everything into *toolbox.h*, the file would be pretty big, around 700 lines or so. And with future demos, it'd be a lot longer. With that in mind, I've started redistributing the contents a little: all the types

go in *types.h*, everything to do with the memory map goes into *memmap.h*, all the register defines go into *memdef.h* and the input inlines and macros can be found in *input.h*. The rest is still in *toolbox.h*, but will find themselves redistributed in the end as well.

The two functions in *toolbox.c* need some more clarification as well I guess. In `oam_init()` I cast the objects to a word pointer and use that for setting things; again, this is simply because it's a lot faster. Because it may be used to initialixe something other than the real OAM, I copy the initialized buffer to OAM just in case.

The other point concerns something of a very specific bug in the optimizer of the current compiler (devkitARM r19b). I expect this to be fixed in a later addition and the basic version here *should* work, but just in case it isn't, set the `#if` expression to 0 if you see OAM get corrupted. If you must know, the problem seems to be `struct` copying of OBJ_ATTRs in a `for` loop. Yes, it's that specific. Even though `struct` copying is legal and fast if they're word aligned, it seems GCC gets confused with 8-byte blocks in loops and uses `memcpy()` for each struct anyway, something that wouldn't work on OAM. Oh well.

```c
#include <string.h>
#include "toolbox.h"
#include "metr.h"


OBJ_ATTR obj_buffer[128];
OBJ_AFFINE *obj_aff_buffer= (OBJ_AFFINE*)obj_buffer;

void obj_test()
{
    int x= 96, y= 32;
    u32 tid= 0, pb= 0;        // (3) tile id, pal-bank
    OBJ_ATTR *metr= &obj_buffer[0];

    obj_set_attr(metr,
        ATTR0_SQUARE,                  // Square, regular sprite
        ATTR1_SIZE_64,                  // 64x64p,
        ATTR2_PALBANK(pb) | tid);    // palbank 0, tile 0

    // (4) position sprite (redundant here; the _real_ position
    // is set further down
    obj_set_pos(metr, x, y);

    while(1)
    {
        vid_vsync();
        key_poll();

        // (5) Do various interesting things
        // move left/right
        x += 2*key_tri_horz();
        // move up/down
        y += 2*key_tri_vert();

        // increment/decrement starting tile with R/L
        tid += bit_tribool(key_hit(-1), KI_R, KI_L);

        // flip
        if(key_hit(KEY_A))  // horizontally
            metr->attr1 ^= ATTR1_HFLIP;
        if(key_hit(KEY_B))  // vertically
            metr->attr1 ^= ATTR1_VFLIP;

        // make it glow (via palette swapping)
        pb= key_is_down(KEY_SELECT) ? 1 : 0;

        // toggle mapping mode
        if(key_hit(KEY_START))
            REG_DISPCNT ^= DCNT_OBJ_1D;
```

```
        // Hey look, it's one of them build macros!
        metr->attr2= ATTR2_BUILD(tid, pb, 0);
        obj_set_pos(metr, x, y);

        oam_copy(oam_mem, obj_buffer, 1);    // (6) Update OAM (only one
now)
    }
}

int main()
{
    // (1) Places the tiles of a 4bpp boxed metroid sprite
    //    into LOW obj memory (cbb == 4)
    memcpy(&tile_mem[4][0], metr_boxTiles, metr_boxTilesLen);
    memcpy(pal_obj_mem, metrPal, metrPalLen);

    // (2) Initialize all sprites
    oam_init(obj_buffer, 128);
    REG_DISPCNT= DCNT_OBJ | DCNT_OBJ_1D;

    obj_test();

    while(1);

    return 0;
}
```

Setting up sprites

Before any sprites show up, there are three things you have to do, although not necessarily in this order. They are: copying sprite graphics to VRAM, setting up OAM to use these graphics, and enabling sprites in the display control, `REG_DISPCNT` .

Display control

Starting with the last one, you enable sprites by setting bit 12 of `REG_DISPCNT` . Usually you'll also want to use 1D mapping, so set bit 6 as well. This is done at **point (2)** of the code.

Hiding all sprites

The other step performed here is a call to `oam_init()`. This isn't strictly necessary, but a good idea nonetheless. What `oam_init()` does is hide all the sprites. Why is this a good idea? Well, because a fully zeroed out OAM does *not* mean the sprites are invisible. If you check the attributes you'll see that this will mean that they're all 8×8-pixel sprites, using tile 0 for their graphics, located at (0,0). If the first tile isn't empty, you'll start with 128 versions of that tile in the top-left corner, which looks rather strange. So, make sure they're all invisible first. The demo also comes with `obj_hide()` and `obj_unhide()` functions, although they aren't used here.

Loading sprite graphics

The first thing to do (**point (1)**) is to store the sprite graphics in object VRAM. As I've already said a few times now, these graphics should be stored as 8×8-pixel tiles, not as a flat bitmap. For example, my sprite here is 64×64p in size, so to store it I've had to convert this to 8×8 separate tiles first. If you do *not* do this, your sprites will look very strange indeed.

Exactly where you put these tiles is actually not all that relevant (apart from the obvious, like mapping mode, and tile alignment, of course). Object VRAM works as a texture pool and has nothing to do with the screen directly. You store the tiles that you want to be available there, and it is by manipulating the OAM attributes that the system knows which tiles you want to use and where you want them. There is no reason why sprite 0 couldn't start at tile 42, or why multiple sprites couldn't use the same tiles. This is also why `OAMData`, which is sometimes used for object VRAM, is such a misnomer: object VRAM has nothing to do with OAM. *Nothing*! If your headers use this name for `0601:0000`, or even `0601:4000`, change it. Please. And be careful where you put things in the bitmap modes, as you can't use tiles 0-511 there.

As I said, loading the sprites happens at **point (1)** in the code. If you paid attention to the overview, you'll remember that `tile_mem[][]` is a two dimensional array, mapping charblocks and 4-bit tiles. You'll also remember that object VRAM is

charblocks 4 and 5, so `&tile_mem[4][0]` points to the first tile in object VRAM. So I'm loading my boxed metroid into the first 64 tiles of object VRAM.

I am also loading its palette into the sprite palette. That's *sprite* palette (`0500:0200`), not background palette. Load it to the wrong place and you won't see anything.

> ### FINDING TILE ADDRESSES
>
> Use `tile_mem` or a macro to find the addresses to copy your tiles too, it's much more readable and maintainable than calculating them manually. You should not have any hard-coded VRAM addresses in your code, ever.

> ### OAMDATA
>
> Headers from other sites sometimes `#define OAMData` as part of VRAM. It is not. Rename it.

Setting attributes

Lastly, I'll set up one `OBJ_ATTR` so that it actually uses the metroid tiles. This is done at **point (3)**, using the `obj_set_attr()` inline function. All it does is three assignments to the attributes of the first argument, by the way, nothing spectacular. This just saves typing doing it this way rather than three separate statements. With this particular call, I tell this sprite that it's a 64×64 pixel (8×8 tile) sprite, and its starting tile is `tid` , which is 0. This means that it'll use the 64 tiles, starting at tile 0.

Note that the sprite I'm setting is actually part of the OAM buffer, not the real OAM. This means that even after I set the attributes there, nothing happens yet. To finalize the sprite I need to update the *real* OAM, which is done by a call to `oam_copy()` (**point (6)**). This carries two arguments: an index and a count denoting how many sprites to

update, and which sprite to start at. I also have `obj_copy()`, which only copies attributes 0, 1 and 2, but *not* 3! This is necessary when you start using affine sprites, which may be copied incorrectly otherwise.

The previous steps are enough to get the metroid sprite on-screen. The story doesn't end there, of course. Here are a few things that you can do with sprites.

Sprite positioning

The first order of business is usually to place it at some position on screen, or even off screen. To do this you have to update the bits for the *y* and *x* positions in attributes 0 and 1, respectively. One mistake I often seem to make is fill *x* into attr0 and *y* into attr1, when it should be the other way around. If your sprite moves strangely, this might be why.

Note that these coordinates mark the **top-left** of the sprite. Also, the number of bits for the coordinates means we have 512 possible *x*-values and 256 *y*-values. The coordinate ranges wrap around, so you could also say that these are signed integers, with the ranges $x \in$ [-256, 255] and $y \in$ [-128, 127]. Yes, that would make the highest *y*-value smaller than the height of the screen, but thanks to the wrapping it all works out. Well, almost. Anyway, thanks to the 2s-complement nature of integers, simply masking the *x* and *y* values by `0x01FF` and `0x00FF`, respectively, will give proper 9- and 8-bit signed values. You can do this manually, or use the `obj_set_pos()` function used at **point (4)**.

You might see code that clears the lower bits of the attributes and then directly ORRs in *x* and *y*. This is not a good idea, because negative values are actually represented by upper half of a datatype's range. −1 for example is all bits set (0xFFFFFFFF). Without masking off the higher bits, negative values would overwrite the rest of the attribute bits, which would be bad.

MASK YOUR COORDINATES

If you're making a sprite positioning function or use someone else's **make sure** you mask off the bits in *x* and *y* before you insert them into the attributes. If not, negative values will overwrite the whole attribute.

This is bad

```
obj->attr0= (obj->attr0 &~ 0x00FF) | (y);
```

This is good:

```
obj->attr0= (obj->attr0 &~ 0x00FF) | (y & 0x00FF);
```

Position variables and using tribools

Instead of using an `OBJ_ATTR` to store the sprite's position, it is better to keep them in separate variables, in this case `x` and `y`. This avoids having to mask coordinate fields all the time, but more importantly, the positions can extend beyond the size of the screen. As most game worlds aren't restricted to a single screen, this is an important point. Then, when the time is right, these are fed to `oam_set_pos()` to update the sprite.

Also, note the use of my tribool key functions to update the positions. Input processing often follows a pattern of "key X pressed: increment, key opposite of Y pressed, decrement" The tribool functions bring that kind of code down from four lines to one, which makes the code easier to read (once you get over the initial hurdle). For example, `key_tri_horz()` returns +1 if 'right' is pressed, −1 if 'left' is pressed, and 0 if neither or both are pressed. `key_tri_vert()` does something similar for vertical movement and the line with `bit_tribool()` function makes a variant using `key_hit()` and R and L to increment or decrement the tile index.

Other attrs

Sprite coordinates are only two of the many sprite attributes that can be controlled with via specific OAM bits, even while the sprite is already active. Some of the obvious ones are flipping or mirroring it, which can be done using A and B here. Or, if you're using a 4bpp sprite, you can swap palettes so that all the colors change. Pressing Select in the demo switches from palette bank 0 to 1, which happens to have a grey to white gradient. Toggling between these palette banks quickly can make the sprite flash. You could also change the priorities in which the sprites are rendered, or toggle alpha blending, although I haven't done those things here.

Now, these things don't really change the overall image of the sprite. What you should realize though is that it *is* possible to do that. As I've already noted before, it's not true that the contents of VRAM *are* the sprite, rather that a sprite *uses* parts of VRAM to show something, anything, on screen. You could, for example, change the starting tile `tid` that the sprite uses, which in this case can be done using L and R. Not only is this legal, it's the standard practice for animation (although you can also overwrite VRAM for that – resetting the tile index is just faster). Understanding this is one of the points of moving from a user to a developer perspective: the user only sees the surface; the coder looks below it and sees what's really going on.

And that's it for regular sprites. Using multiple sprites isn't much different – seen one, seen them all. Basic animation shouldn't be problematic either, until you run out of VRAM to put them in. There are still a few regions left untouched like blending and mosaic, but I'll deal with those later.

9. Regular tiled backgrounds

Tilemap introduction

Tilemaps are the bread and butter for the GBA. Almost every commercial GBA game makes use of tile modes, with the bitmap modes seen only in 3D-like games that use ray-tracing. Everything else uses tiled graphics.

The reason why tilemaps are so popular is that they're implemented in hardware and require less space than bitmap graphics. Consider fig 9.1a. This is a 512 by 256 image, which even at 8bpp would take up 128 KiB of VRAM, and we simply don't have that. If you were to make one big bitmap of a normal level in a game, you can easily get up to 1000×1000 pixels, which is just not practical. And *then* there's the matter of scrolling through the level, which means updating all pixels each frame. Even when your scrolling code is fully optimized that'd take quite a bit of time.

Now, notice that there are many repeated elements in this image. The bitmap seems to be divided into groups of 16×16 pixels. These are the *tiles*. The list of unique tiles is the *tileset*, which is given in fig 9.1b. As you can see, there are only 16 unique tiles making up the image. To create the image from these tiles, we need a *tilemap*. The image is divided into a matrix of tiles. Each element in the matrix has a *tile index* which indicates which tile should be rendered there; the tilemap can be seen in fig 9.1c.

Suppose both the tileset and map used 8-bit entries, the sizes are 16×(16×16) = 4096 bytes for the tileset and 32×16 = 512 bytes for the tilemap. So that's 4.5 KiB for the whole scene rather than the 128 KiB we had before; a size reduction of a factor of 28.
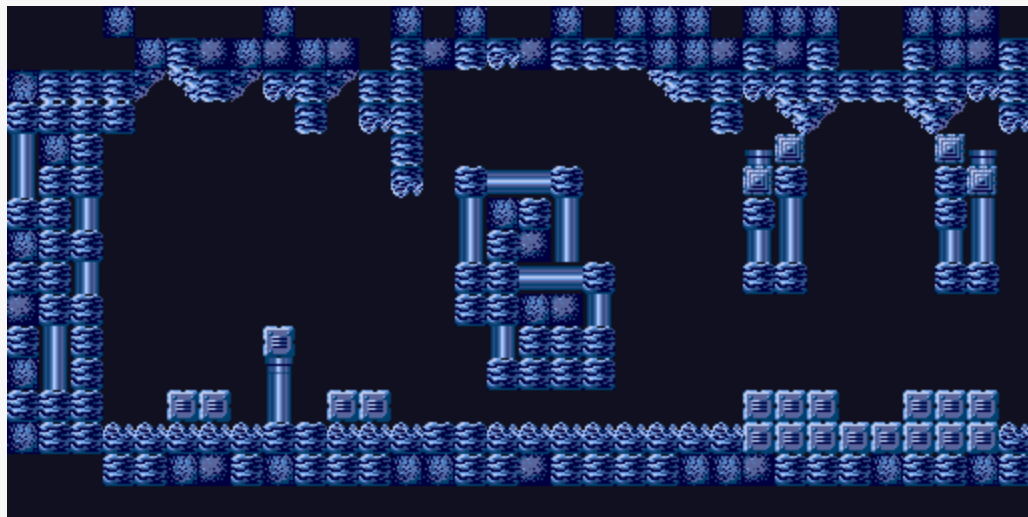
Fig 9.1a: image on screen.

The tile mapping process. Using the tileset of fig 9.1b, and the tile map of fig 9.1c, the end-result is fig 9.1a.

Fig 9.1b: the tile set.

Fig 9.1c: the tile map (with the proper tiles as a backdrop).

That's basically how tilemaps work. You don't define the whole image, but group pixels together into tiles and describe the image in terms of those groups. In the fig 9.1, the tiles were 16×16 pixels, so the tilemap is 256 times smaller than the bitmap. The unique tiles are in the tileset, which can (and usually will) be larger than the tilemap. The size of the tileset can vary: if the bitmap is highly variable, you'll probably have many unique tiles; if the graphics are nicely aligned to tile boundaries already (as it is here), the tileset will be small. This is why tile-engines often have a distinct look to them.

Tilemaps for the GBA

In the tiled video-modes (0, 1 and 2) you can have up to four backgrounds that display tilemaps. The size of the maps is set by the control registers and can be between 128×128 and 1024×1024 pixels. The size of each tile is always 8×8 pixels, so fig 9.1 isn't *quite* the way it'd work on the GBA. Because accessing the tilemaps is done in units of tiles, the map sizes correspond to 16×16 to 128×128 tiles.

Both the tiles and tilemaps are stored in VRAM, which is divided into ***charblocks*** and ***screenblocks***. The tileset is stored in the charblocks and the tilemap goes into the screenblocks. In the common vernacular, the word "tile" is used for both the graphical tiles and the entries of the tilemaps. Because this is somewhat confusing, I'll use the term *screen entry* (*SE* for short) as the items in the screenblocks (i.e., the map entries) and restrict tiles to the tileset.

64 KiB of VRAM is set aside for tilemaps (`0600:0000h` - `0600:FFFFh`). This is used for both screenblocks *and* charblocks. You can choose which ones to use freely through the control registers, but be careful that they can overlap (see table 9.1). Each screenblock is 2048 (`800h`) bytes long, giving 32 screenblocks in total. All but the smallest backgrounds use multiple screenblocks for the full tilemap. Each charblock is 16 KiB (`4000h` bytes) long, giving four blocks overall.

Memory	0600:0000	0600:4000	0600:8000	0600:C000
charblock	0	1	2	3

screenblock	0	...	7	8	...	15	16	...	23	24	...

Table 9.1: charblock and screenblock overlap.

TILES VS 'TILES'

Both the entries of the tilemap and the data in the tileset are often referred to as 'tiles', which can make conversation confusing. I reserve the term 'tile' for the graphics, and 'screen(block) entry' or 'map entry' for the map's contents.

CHARBLOCKS VS SCREENBLOCKS

Charblocks and screenblocks use the same addresses in memory. Each charblock overlaps eight screenblocks. When loading data, make sure the tiles themselves don't overwrite the map, or vice versa.

Size was one of the benefits of using tilemaps, speed was another. The rendering of tilemaps in done in hardware and if you've ever played PC games in hardware and software modes, you'll know that hardware is good. Another nice point is that scrolling is done in hardware too. Instead of redrawing the whole scene, you just have to enter some coordinates in the right registers.

As I said in the overview, there are three stages to setting up a tiled background: control, mapping and image-data. I've already covered most of the image-data in the overview, as well as some of the control and mapping parts that are shared by sprites and backgrounds alike; this chapter covers only things specific to backgrounds in general and regular backgrounds in particular. I'm assuming you've read the overview.

ESSENTIAL TILEMAP STEPS

- Load the graphics: tiles into charblocks and colors in the background palette.
- Load a map into one or more screenblocks.
- Switch to the right mode in `REG_DISPCNT` and activate a background.
- Initialize that background's control register to use the right CBB, SBB and bitdepth.

Background control

Background types

Just like sprites, there are two types of tiled backgrounds: regular and affine; these are also known as text and rotation backgrounds, respectively. The type of the background depends of the video mode (see table 9.2). At their cores, both regular and affine backgrounds work the same way: you have tiles, a tile-map and a few control registers. But that's where the similarity ends. Affine backgrounds use more and different registers than regular ones, and even the maps are formatted differently. This page only covers the regular backgrounds. I'll leave the affine ones till after the page on the affine matrix.

mode	BG0	BG1	BG2	BG3
0	reg	reg	reg	reg
1	reg	reg	aff	-
2	-	-	aff	aff

Table 9.2: video modes and background type

Control registers

All backgrounds have 3 primary control registers. The primary control register is `REG_BGxCNT`, where *x* indicates the backgrounds 0 through 3. This register is where you say what the size of the tilemap is, and which charblock and screenblock it uses. The other two are the scrolling registers, `REG_BGxHOFS` and `REG_BGxVOFS`.

Each of these is a 16-bit register. `REG_BG0CNT` can be found at `0400:0008`, with the other controls right behind it. The offsets are paired by background, forming coordinate pairs. These start at `0400:0010`

Register	length	address
REG_BGxCNT	2	0400:0008h + 2·*x*
REG_BGxHOFS	2	0400:0010h + 4·*x*
REG_BGxVOFS	2	0400:0012h + 4·*x*

Table 9.3: Background register addresses

The description of `REG_BGxCNT` can be found below. Most of it is pretty standard, except for the size: there are actually *two* lists of possible sizes; one for regular maps and one for affine maps. The both use the same bits you may have to be careful that you're using the right `#define`s.

```
REG_BGxCNT @ 0400:0008 + 2x
```

F E	D	C B A 9 8	7	6	5 4	3 2	1 0
Sz	Wr	SBB	CM	Mos	–	CBB	Pr

bits	name	define	description
0-1	Pr	*BG_PRIO#*	**Priority**. Determines drawing order of backgrounds.
2-3	CBB	*BG_CBB#*	**Character Base Block**. Sets the charblock that serves as the base for character/tile indexing. Values: 0-3.
6	Mos	BG_MOSAIC	**Mosaic** flag. Enables mosaic effect.

7	CM	BG_4BPP, BG_8BPP	**Color Mode**. 16 colors (4bpp) if cleared; 256 colors (8bpp) if set.
8-C	SBB	*BG_SBB#*	**Screen Base Block**. Sets the screenblock that serves as the base for screen-entry/map indexing. Values: 0-31.
D	Wr	BG_WRAP	**Affine Wrapping** flag. If set, affine background wrap around at their edges. Has no effect on regular backgrounds as they wrap around by default.
E-F	Sz	*BG_SIZE#, see below*	**Background Size**. Regular and affine backgrounds have different sizes available to them. The sizes, in tiles and in pixels, can be found in table 9.4.

Sz-flag	define	(tiles)	(pixels)
00	`BG_REG_32x32`	32×32	256×256
01	`BG_REG_64x32`	64×32	512×256
10	`BG_REG_32x64`	32×64	256×512
11	`BG_REG_64x64`	64×64	512×512

Table 9.4a: regular bg sizes

Sz-flag	define	(tiles)	(pixels)
00	`BG_AFF_16x16`	16×16	128×128
01	`BG_AFF_32x32`	32×32	256×256
10	`BG_AFF_64x64`	64×64	512×512
11	`BG_AFF_128x128`	128×128	1024×1024

Table 9.4b: affine bg sizes

Each background has two 16-bit scrolling registers to offset the rendering (`REG_BGxHOFS` and `REG_BGxVOFS`). There are a number of interesting points about these. First, because regular backgrounds wrap around, the values are essentially

modulo *mapsize*. This is not really relevant at the moment, but you can use this to your benefit once you get to more advanced tilemaps. Second, these registers are **write-only**! This is a little annoying, as it means that you can't update the position by simply doing `REG_BG0HOFS++` and the like.

And now the third part, which may be the most important, namely what the values actually *do*. The simplest way of looking at them is that they give the coordinates of the screen on the map. Read that again, carefully: it's the position of the screen on the map. It is *not* the position of the map on the screen, which is how sprites work. The difference is only a minus sign, but even something as small as a sign change can wreak havoc on your calculations.
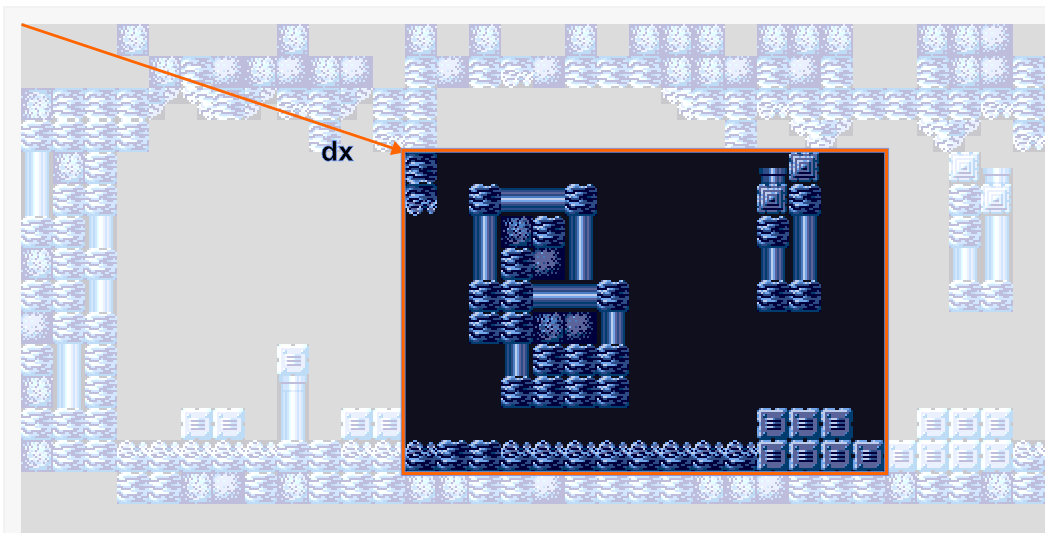
Fig 9.2: Scrolling offset **dx** sets is the position of the screen on the map. In this case, **dx** = (192, 64).

So, if you increase the scrolling values, you move the screen to the right, which corresponds to the map moving *left* on the screen. In mathematical terms, if you have map position **p** and screen position **q**, then the following is true:

$$(9.1) \quad \begin{aligned} q + dx &= p \\ q &= p - dx \end{aligned}$$

DIRECTION OF OFFSET REGISTERS

The offset registers REG_BGxHOFS and REG_BGxVOFS indicate which map location is mapped to the top-left of the screen, meaning positive offsets scroll the map left and up. Watch your minus signs.

OFFSET REGISTERS ARE WRITE ONLY

The offset registers are **write-only**! That means that direct arithmetic like `+=` will not work.

Useful types and #defines

Tonc's code has several useful extra types and macros that can make life a little easier.

```
// === Additional types (tonc_types.h) ================================

//! Screen entry conceptual typedef
typedef u16 SCR_ENTRY;

//! Affine parameter struct for backgrounds, covered later
typedef struct BG_AFFINE
{
    s16 pa, pb;
    s16 pc, pd;
    s32 dx, dy;
} ALIGN4 BG_AFFINE;

//! Regular map offsets
typedef struct BG_POINT
{
    s16 x, y;
} ALIGN4 BG_POINT;

//! Screenblock struct
typedef SCR_ENTRY    SCREENBLOCK[1024];


// === Memory map #defines (tonc_memmap.h) ============================

//! Screen-entry mapping: se_mem[y][x] is SBB y, entry x
#define se_mem          ((SCREENBLOCK*)MEM_VRAM)

//! BG control register array: REG_BGCNT[x] is REG_BGxCNT
#define REG_BGCNT       ((vu16*)(REG_BASE+0x0008))

//! BG offset array: REG_BG_OFS[n].x/.y is REG_BGnHOFS / REG_BGnVOFS
#define REG_BG_OFS      ((BG_POINT*)(REG_BASE+0x0010))

//! BG affine params array
#define REG_BG_AFFINE   ((BG_AFFINE*)(REG_BASE+0x0000))
```

Strictly speaking, making a `SCREEN_ENTRY typedef` is not necessary, but makes its use clearer. `se_mem` works much like `tile_mem`: it maps out VRAM into screenblocks screen-entries, making finding a specific entry easier. The other typedefs are used to map out arrays for the background registers. For example, `REG_BGCNT` is an array that maps out all `REG_BGxCNT` registers. `REG_BGCNT[0]` is `REG_BG0CNT`, etc. The `BG_POINT` and `BG_AFFINE` types are used in similar fashions. Note that `REG_BG_OFS` still covers the same registers as `REG_BGxHOFS` and `REG_BGxVOFS` do, and the write-only-ness of

them has not magically disappeared. The same goes for `REG_BG_AFFINE`, but that discussion will be saved for another time.

In theory, it is also useful create a sort of background API, with a struct with the temporaries for map positioning and functions for initializing and updating the registers and maps. However, most of tonc's demos are not complex enough to warrant these things. With the types above, manipulating the necessary items is already simplified enough for now.

Regular background tile-maps

The screenblocks form a matrix of screen entries that describe the full image on the screen. In the example of fig 9.1, the tilemap entries just contained the tile index. The GBA screen entries bahave a little differently.

For regular tilemaps, each screen entry is 16-bits long. Besides the tile index, it contains flipping flags and a palette bank index for 4bpp / 16-color tiles. The exact layout can be found in "Screen entry format" below. The affine screen entries are only 8 bits wide and just contain an 8-bit tile index.

```
Screen entry format for regular backgrounds
 F E D C    B     A    9 8 7 6 5 4 3 2 1 0
    PB      VF    HF             TID
```

bits	name	define	description
0-9	TID	SE_ID#	**Tile-index** of the SE.
A-B	HF, VF	SE_HFLIP, SE_VFLIP. SE_FLIP#	**Horizontal/vertical flipping** flags.
C-F	PB	SE_PALBANK#	**Palette bank** to use when in 16-color mode. Has no effect for 256-color bgs

(`REG_BGxCNT{6}` is set).

Map layout

VRAM contains 32 screenblocks to store the tilemaps in. Each screenblock is 800h bytes long, so you can fit 32×32 screen entries into it, which equals one 256×256 pixel map. The bigger maps simply use more than one screenblock. The screenblock index set in `REG_BGxCNT` is the **screen base block** which indicates the start of the tilemap.

Now, suppose you have a tilemap that's *tw*×*th* tiles/SEs in size. You might expect that the screen entry at tile-coordinates (*tx*, *ty*) could be found at SE-number $n = tx + ty \cdot tw$, because that's how matrices always work, right? Well, you'd be wrong. At least, you'd be *partially* wrong.

Within each screenblock the equation works, but the bigger backgrounds don't simply *use* multiple screenblocks, they're actually accessed as four separate maps. How this works can be seen in table 9.5: each numbered block is a contingent block in memory. This means that to get the SE-index you have to find out which screenblock you are in and then find the SE-number inside that screenblock.

Table 9.5: screenblock layout of
regular backgrounds.

This kind of nesting problem isn't as hard as it looks. We know how many tiles fit in a screenblock, so to get the SBB-coordinates, all we have to do divide the tile-coords by the SBB width and height: *sbx*=*tx*/32 and *sby*=*ty*/32. The SBB-number can then be found with the standard matrix→array formula. To find the in-SBB SE-number, we have to use *tx*%32 and *ty*%32 to find the in-SBB coordinates, and then again the conversion from 2D coords to a single element. This is to be offset by the SBB-number tiles the size of an SBB to find the final number. The final form would be:

```
//! Get the screen entry index for a tile-coord pair
//  And yes, the div and mods will be converted by the compiler
uint se_index(uint tx, uint ty, uint pitch)
{
    uint sbb= (ty/32)*(pitch/32) + (tx/32);
    return sbb*1024 + (ty%32)*32 + tx%32;
}
```

The general formula is left as an exercise for the reader – one that is well worth the effort, in my view. This kind of process crops up in a number of places, like getting the offset for bitmap coordinates in tiles, and tile coords in 1D object mapping.

If all those operations make you queasy, there's also a faster version specifically for a 2×2 arrangement. It starts with calculating the number as if it's a 32×32t map. This will be incorrect for a 64t wide map, which we can correct for by adding 0x0400−0x20 (i.e., tiles/block − tiles per row). We need another full block correction is the size is 64×64t.

```
//! Get the screen entry index for a tile-coord pair.
/*! This is the fast (and possibly unsafe) way.
*    \param bgcnt    Control flags for this background (to find its size)
*/

uint se_index_fast(uint tx, uint ty, u16 bgcnt)
{
    uint n= tx + ty*32;
    if(tx >= 32)
        n += 0x03E0;
    if(ty >= 32 && (bgcnt&BG_REG_64x64)==BG_REG_64x64)
        n += 0x0400;
    return n;
}
```

I would like to remind you that n here is the SE-number, not the address. Since the size of a regular SE is 2 bytes, you need to multiply n by 2 for the address. (Unless, of course, you have a pointer/array of `u16`s, in which case n will work fine.) Also, this works for regular backgrounds only; affine backgrounds use a linear map structure, which makes this extra work unnecessary there. By the way, both the screen-entry and map layouts are different for affine backgrounds. For their formats, see the map format section of the affine background page.

Background tile subtleties

There are two additional things you need to be aware of when using tiles for tile-maps. The first concerns tile-numbering. For sprites, numbering went according to 4-bit tiles (s-tiles); for 8-bit tiles (d-tiles) you'd have use multiples of 2 (a bit like u16 addresses are always multiples of 2 in memory). In tile-maps, however, d-tiles are numbered by the d-tile. To put it in other words, for sprites, using index *id* indicates the same tile for both 4 and 8-bit tiles, namely the one that starts at *id*·20h. For tile-maps, however, it starts at *id*·20h for 4-bit tiles, but at *id*·40h for 8-bit tiles.

memory offset	000h	020h	040h	060h	080h	100h	...
4bpp tile	0	1	2	3	4	5	...
8bpp tile	0		1		2		...

Table 9.6: tile counting for backgrounds, sticks to its bit-depth.

The second concerns, well, also tile-numbering, but more how many tiles you can use. Each map entry for regular backgrounds has 10 bits for a tile index, so you can use up to 1024 tiles. However, a quick calculation shows that a charblock contains 4000h/20h= 512 s-tiles, or 4000h/40h= 256 d-tiles. So what's the deal here? Well, the charblock index you set in `REG_BGxCNT` is actually only the block where tile-counting starts: its ***character base block***. You can use the ones after it as well. Cool, huh? But wait, if you can access subsequent charblocks as well; does this mean that, if you set the base charblock to 3, you can use the sprite blocks (which are basically blocks 4 and 5) as well?

The answer is: yes. And NO!

Emulators from the early 2000s allow you to do this. However, a real GBA doesn't. It does output *something*, though: the screen-entry will be used as tile-data itself, but in a manner that simply defies explanation. Trust me on this one, okay? Of the current tonc demos, this is one of the times that VBA gets it wrong.

AVAILABLE TILES

For both 4bpp and 8bpp regular bgs, you can access 1024 tiles. The only caveat here is that you cannot access the tiles in the object charblocks even if the index would call for it.

Another thing you may be wondering is if you can use a particular screenblock that is within a currently used charblock. For example, is it allowed to have a background use charblock 0 and screenblock 1. Again, yes you can do this. This can be useful since you're not likely to fill an entire charblock, so using its later screenblocks for your map data is a good idea. (A sign of True Hackerdom would be if you manage to use the same data for both tiles and SEs and still get a meaningful image (this last part is important). If you have done this, please let me know.)

TILEMAP DATA CONVERSION VIA CLI

A converter that can tile images (for objects), can also create a tileset for tilemaps, although there will likely be many redundant tiles. A few converters can also reduce the tileset to only the unique tiles, and provide the tilemap that goes with it. The Brinstar bitmap from fig 9.1 is a 512×256 image, which could be tiled to a 64×32 map with a 4bpp tileset reduced for uniqueness in tiles, including palette info and mirroring.

```
# gfx2gba
# (C array; u8 foo_Tiles[], u16 foo_Map[],
# u16 master_Palette[]; foo.raw.c, foo.map.c, master.pal.c)
    gfx2gba -fsrc -c16 -t8 -m foo.bmp


# grit
# (C array; u32 fooTiles[], u16 fooMap[], u16 fooPal[]; foo.c,
foo.h)
    grit foo.bmp -gB4 -mRtpf
```

Two notes on gfx2gba: First, it merges the palette to a single 16-color array, rearranging it in the process. Second, while it lists metamapping options in the

> readme, it actually doesn't give a metamap and meta-tileset, it just formats the map into different blocks.

Tilemap demos

There are four demos in this chapter. The first one is *brin_demo*, which is very, very short and shows the basic steps of tile loading and scrolling. The next ones are called *sbb_reg* and *cbb_demo*, which are tech demos, illustrating the layout of multiple screenblocks and how tile indexing is done on 4bpp and 8bpp backgrounds. In both these cases, the map data is created manually because it's more convenient to do so here, but using map-data created by map editors really isn't that different.

Essential tilemap steps: brin_demo

As I've been using a 512×256 part of Brinstar throughout this chapter, I thought I might as well use it for a demo.

There are a few map editors out there that you can use. Two good ones are Nessie's MapEd or Mappy, both of which have a number of interesting features. I have my own map editor, mirach, but it's just a very basic thing. Some tutorials may point you to GBAMapEditor. Do *not* use this editor as it's pretty buggy, leaving out half of the tilemaps sometimes. Tilemaps can be troublesome enough for beginners without having to worry about whether the map data is faulty.

In this cause, however, I haven't used any editor at all. Some of the graphics converters can convert to a tileset+tilemap – it's not the standard method, but for small maps it may well be easier. In this case I've used Usenti to do it, but grit and gfx2gba work just as well. Note that because the map here is 64×32 tiles, which requires splitting into screenblocks. In Usenti this is called the 'sbb' layout, in grit it's '-mLs' and for gfx2gba you'd use '-mm 32' … I think. In any case, after a conversion you'd have a palette, a tileset and a tilemap.
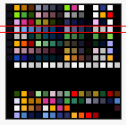
Fig 9.3a: *brin_demo* palette.

Fig 9.3b: *brin_demo* tileset.

```
const unsigned short brinMap[2048]=
{
    // Map row 0

0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x3001,0x3002,

0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,

0x3001,0x3002,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,

0x3001,0x3002,0x0000,0x0000,0x3001,0x3002,0x0000,0x0000,

    // Map row 1

0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x3003,0x3004,

0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,

0x3003,0x3004,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,

0x3003,0x3004,0x0000,0x0000,0x3003,0x3004,0x0000,0x0000,

    // Map row 2

0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,0x0000,
    0x3001,0x3002,0x3005,0x3006,0x3007,0x3008,
    // ... etc
```

In fig 9.3 you can see the full palette, the tileset and part of the map. Note that the tileset of fig 9.3b is not the same as that of fig 9.1b because the former uses 8×8 tiles while the latter used 16×16 tiles. Note also that the screen entries you see here are either 0 (i.e., the empty tile) or of the form `0x3xxx`. The high nybble indicates the palette bank, in this case three. If you'd look to the palette (fig 9.3a) you'd see that this gives bluish colors.

Now on to using these data. Remember the essential steps here:

- Load the graphics: tiles into charblocks and colors in the background palette.
- Load a map into one or more screenblocks.
- Switch to the right mode in `REG_DISPCNT` and activate a background.

- Initialize that background's control register to use the right CBB, SBB and bitdepth.

If you do it correctly, you should have something showing on screen. If not, go to the tile/map/memory viewers of your emulator; they'll usually give you a good idea where the problem is. A common one is having a mismatch between the CBB and SBB in `REG_BGxCNT` and where you put the data, which most likely would leave you with an empty map or empty tileset.

The full code of *brin_demo* is given below. The three calls to `memcpy()` load up the palette, tileset and tilemap. For some reason, probably related to where the NES and 8-bit Game Boy put screenblocks in video memory, it's become conventional to place the maps in the last screenblocks on GBA as well. In this case, that's 30 rather than 31 because we need two blocks for a 64×32t map. For the scrolling part, I'm using two variables to store and update the positions because the scrolling registers are write-only. I'm starting at (192, 64) here because that's what I used for the scrolling picture of fig 9.2 earlier.

```c
#include <string.h>

#include "toolbox.h"
#include "input.h"
#include "brin.h"

int main()
{
    // Load palette
    memcpy(pal_bg_mem, brinPal, brinPalLen);
    // Load tiles into CBB 0
    memcpy(&tile_mem[0][0], brinTiles, brinTilesLen);
    // Load map into SBB 30
    memcpy(&se_mem[30][0], brinMap, brinMapLen);

    // set up BG0 for a 4bpp 64x32t map, using
    //   using charblock 0 and screenblock 31
    REG_BG0CNT= BG_CBB(0) | BG_SBB(30) | BG_4BPP | BG_REG_64x32;
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    // Scroll around some
    int x= 192, y= 64;
    while(1)
    {
        vid_vsync();
        key_poll();

        x += key_tri_horz();
        y += key_tri_vert();

        REG_BG0HOFS= x;
        REG_BG0VOFS= y;
    }

    return 0;
}
```
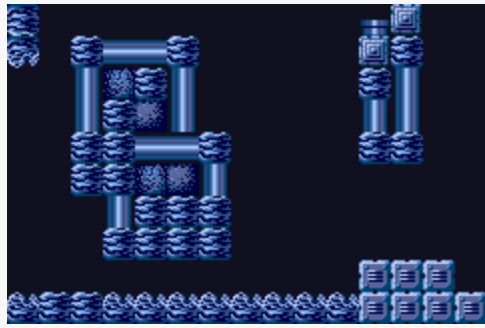
Fig 9.4a: *brin_demo* at **dx**=(192, 64).
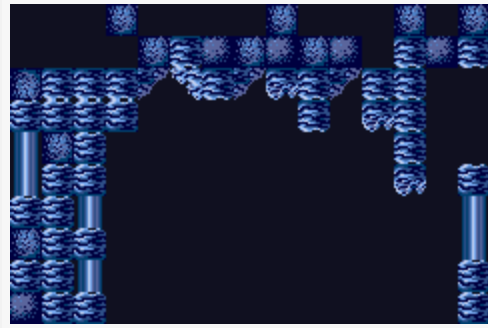
Fig 9.4b: *brin_demo* at **dx**=(0, 0).

Interlude: Fast-copying of non sbb-prepared maps

This is not exactly required knowledge, but should make for an interesting read. In this demo I use a multi-sbb map that was already prepared for that. The converter made sure that the left block of the map came before the right block. If this weren't the case then you couldn't load the whole map in one go because the second row of the left block would use the first row of the right block and so on (see fig 9.5).
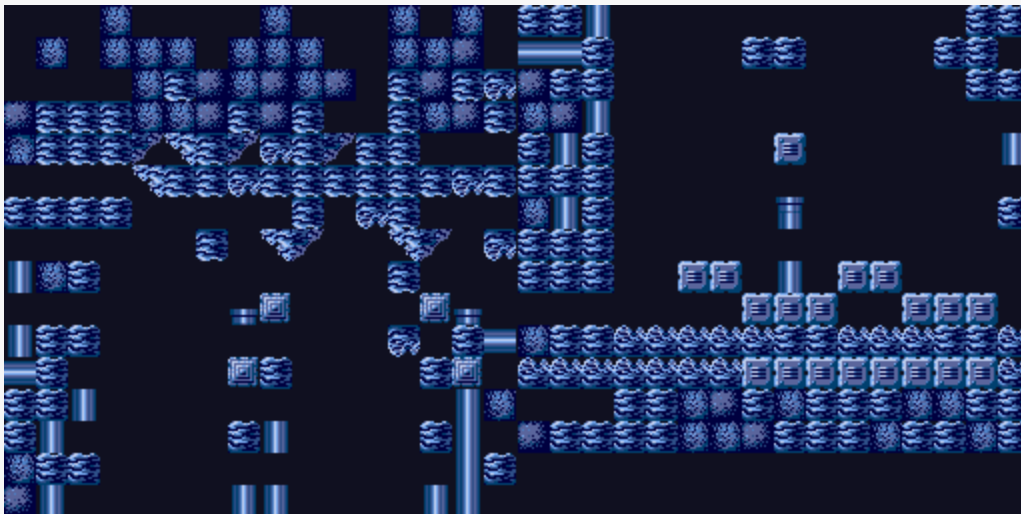
Fig 9.5 *brin_demo* without blocking out into SBB's first.

There are few simple and slow ways and one simple and fast way of copying a non sbb-prepared map to a multiple screenblocks. The slow way would be to perform a double loop to go row by row of each screenblock. The fast way is through struct-copies and pointer arithmetic, like this:

```
typedef struct { u32 data[8]; } BLOCK;

int iy;
BLOCK *src= (BLOCK*)brinMap;
BLOCK *dst0= (BLOCK*)se_mem[30];
BLOCK *dst1= (BLOCK*)se_mem[31];

for(iy=0; iy<32; iy++)
{
    // Copy row iy of the left half
    *dst0++= *src++;      *dst0++= *src++;

     // Copy row iy of the right half
    *dst1++= *src++;      *dst1++= *src++;
}
```

A `BLOCK` struct-copy takes care of half a row, so two takes care of a whole screenblock row (yes, you could define `BLOCK` as a 16-word struct, but that wouldn't work out anymore. Trust me). At that point, the `src` pointer has arrived at the right half of the map, so we copy the next row into the right-hand side destination, `dst1`. When done with that, `src` points to the second row of the left side. Now do this for all 32 lines. Huzzah for struct-copies, and pointers!

A screenblock demo

The second demo, *sbb_reg*, uses a 64×64t background to indicate how multiple screenblocks are used for bigger maps in more detail. While the *brin_demo* used a multi-sbb map as well, it wasn't easy to see what's what because the map was irregular; this demo uses a very simple tileset so you can clearly see the screenblock boundaries. It'll also show how you can use the `REG_BG_OFS` registers for scrolling rather than `REG_BGxHOFS` and `REG_BGxVOFS`.

```c
#include "toolbox.h"
#include "input.h"

#define CBB_0  0
#define SBB_0 28

#define CROSS_TX 15
#define CROSS_TY 10

BG_POINT bg0_pt= { 0, 0 };
SCR_ENTRY *bg0_map= se_mem[SBB_0];


uint se_index(uint tx, uint ty, uint pitch)
{
    uint sbb= ((tx>>5)+(ty>>5)*(pitch>>5));
    return sbb*1024 + ((tx&31)+(ty&31)*32);
}

void init_map()
{
    int ii, jj;

    // initialize a background
    REG_BG0CNT= BG_CBB(CBB_0) | BG_SBB(SBB_0) | BG_REG_64x64;
    REG_BG0HOFS= 0;
    REG_BG0VOFS= 0;

    // (1) create the tiles: basic tile and a cross
    const TILE tiles[2]=
    {
        {{0x11111111, 0x01111111, 0x01111111, 0x01111111,
          0x01111111, 0x01111111, 0x01111111, 0x00000001}},
        {{0x00000000, 0x00100100, 0x01100110, 0x00011000,
          0x00011000, 0x01100110, 0x00100100, 0x00000000}},
    };
    tile_mem[CBB_0][0]= tiles[0];
    tile_mem[CBB_0][1]= tiles[1];

    // (2) create a palette
    pal_bg_bank[0][1]= RGB15(31,  0,  0);
    pal_bg_bank[1][1]= RGB15( 0, 31,  0);
    pal_bg_bank[2][1]= RGB15( 0,  0, 31);
    pal_bg_bank[3][1]= RGB15(16, 16, 16);

    // (3) Create a map: four contingent blocks of
    //   0x0000, 0x1000, 0x2000, 0x3000.
    SCR_ENTRY *pse= bg0_map;
    for(ii=0; ii<4; ii++)
        for(jj=0; jj<32*32; jj++)
```

```
            *pse++= SE_PALBANK(ii) | 0;
}

int main()
{
    init_map();
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0 | DCNT_OBJ;

    u32 tx, ty, se_curr, se_prev= CROSS_TY*32+CROSS_TX;

    bg0_map[se_prev]++; // initial position of cross
    while(1)
    {
        vid_vsync();

        key_poll();

        // (4) Moving around
        bg0_pt.x += key_tri_horz();
        bg0_pt.y += key_tri_vert();

        // (5) Testing se_index
        // If all goes well the cross should be around the center of
        // the screen at all times.
        tx= ((bg0_pt.x>>3)+CROSS_TX) & 0x3F;
        ty= ((bg0_pt.y>>3)+CROSS_TY) & 0x3F;

        se_curr= se_index(tx, ty, 64);
        if(se_curr != se_prev)
        {
            bg0_map[se_prev]--;
            bg0_map[se_curr]++;
            se_prev= se_curr;
        }

        REG_BG_OFS[0]= bg0_pt;  // write new position
    }
    return 0;
}
```

The `init_map()` contains all of the initialization steps: setting up the registers, tiles, palettes and maps. Unlike the previous demo, the tiles, palette and the map are all created manually because it's just easier in this case. At point (1), I define two tiles. The first one looks a little like a pane and the second one is a rudimentary cross. You can see them clearly in the screenshot (fig 9.4). The pane-like tile is loaded into tile 0, and is therefore the 'default' tile for the map.

The palette is set at point (2). The colors are the same as in table 9.5: red, green, blue and grey. Take note of which palette entries I'm using: the colors are in different palette banks so that I can use palette swapping when I fill the map. Speaking of which …
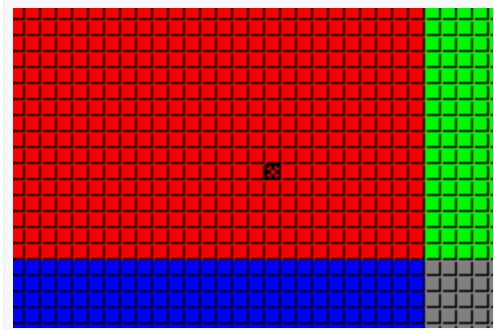

Fig 9.6: *sbb_reg*. Compare table 9.5, 64×64t background. Note the little cross in the top left corner.

Loading the map itself (point (3)) happens through a double loop. The outer loop sets the palette-bank for the screen entries. The inner loop fills 1024 SEs with palette-swapped tile-0's. Now, if big maps used a flat layout, the result would be a big map in four colored bands. However, what actually happens is that you see *blocks*, not bands, proving that indeed regular maps are split into screenblocks just like table 9.5 said. Yes, it's annoying, but that's just the way it is.

That was creating the map, now we turn to the main loop in `main()`. The keys (point (4)) let you scroll around the map. The RIGHT button is tied to a positive change in *x*, but the map itself actually scrolls to the *left*! When I say it like that it may seem counter-intuitive, but if you look at the demo you see that it actually makes sense. Think of it from a hypothetical player sprite point of view. As the sprite moves through the world, you need to update the background to keep the sprite from going off-screen. To do that, the background's movement should be the opposite of the sprite's movement. For example, if the sprite moves to the *right*, you have to move the background to the *left* to compensate.

Finally, there's one more thing to discuss: the cross that appears centered on the map. To do this as you scroll along, I keep track of the screen-entry at the center of the screen via a number of variables and the `se_index()` function. Variables `tx` and `ty` are the tile coordinates of the center of the screen, found by shifting and masking the background pixel coordinates. Feeding these to `se_index()` gives me the screen-entry offset from the screen base block. If this is different than the previous offset, I repaint the former offset as a pane, and update the new offset to the cross. That way, the cross seems to move over the map; much like a sprite would. This was actually

designed as a test for `se_index()` ; if the function was flawed, the cross would just disappear at some point. But it doesn't. Yay me `^_^`

The charblock demo

The third demo, *cbb_demo*, covers some of the details of charblocks and the differences in 4bpp and 8bpp tiles. The backgrounds in question are BG 0 and BG 1. Both will be 32×32t backgrounds, but BG 0 will use 4bpp tiles and CBB 0 and BG 2 uses 8bpp tiles and CBB 2. The exact locations and contents of the screenblocks are not important; what is important is to load the tiles to the starts of all 6 charblocks and see what happens.

```c
#include <toolbox.h>
#include "cbb_ids.h"

#define CBB_4 0
#define SBB_4 2

#define CBB_8 2
#define SBB_8 4

void load_tiles()
{
    int ii;
    TILE *tl= (TILE*)ids4Tiles;
    TILE8 *tl8= (TILE8*)ids8Tiles;

    // Loading tiles. don't get freaked out on how it looks
    // 4-bit tiles to blocks 0 and 1
    tile_mem[0][1]= tl[1];       tile_mem[0][2]= tl[2];
    tile_mem[1][0]= tl[3];       tile_mem[1][1]= tl[4];
    // and the 8-bit tiles to blocks 2 though 5
    tile8_mem[2][1]= tl8[1];    tile8_mem[2][2]= tl8[2];
    tile8_mem[3][0]= tl8[3];    tile8_mem[3][1]= tl8[4];
    tile8_mem[4][0]= tl8[5];    tile8_mem[4][1]= tl8[6];
    tile8_mem[5][0]= tl8[7];    tile8_mem[5][1]= tl8[8];

    // And let's not forget the palette (yes, obj pal too)
    u16 *src= (u16*)ids4Pal;
    for(ii=0; ii<16; ii++)
        pal_bg_mem[ii]= pal_obj_mem[ii]= *src++;
}

void init_maps()
{
    // se4 and se8 map coords: (0,2) and (0,8)
    SB_ENTRY *se4= &se_mem[SBB_4][2*32], *se8= &se_mem[SBB_8][8*32];
    // show first tiles of char-blocks available to bg0
    // tiles 1, 2 of char-block CBB_4
    se4[0x01]= 0x0001;       se4[0x02]= 0x0002;

    // tiles 0, 1 of char-block CBB_4+1
    se4[0x20]= 0x0200;       se4[0x21]= 0x0201;

    // show first tiles of char-blocks available to bg1
    // tiles 1, 2 of char-block CBB_8 (== 2)
    se8[0x01]= 0x0001;       se8[0x02]= 0x0002;

    // tiles 1, 2 of char-block CBB_8+1
    se8[0x20]= 0x0100;       se8[0x21]= 0x0101;

    // tiles 1, 2 of char-block CBB_8+2 (== CBB_OBJ_LO)
```

```
    se8[0x40]= 0x0200;        se8[0x41]= 0x0201;

    // tiles 1, 2 of char-block CBB_8+3 (== CBB_OBJ_HI)
    se8[0x60]= 0x0300;        se8[0x61]= 0x0301;
}

int main()
{
    load_tiles();
    init_maps();

    // init backgrounds
    REG_BG0CNT= BG_CBB(CBB_4) | BG_SBB(SBB_4) | BG_4BPP;
    REG_BG1CNT= BG_CBB(CBB_8) | BG_SBB(SBB_8) |  BG_8BPP;
    // enable backgrounds
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0 | DCNT_BG1 | DCNT_OBJ;

    while(1);

    return 0;
}
```

The tilesets can be found in *cbb_ids.c*. Each tile contains two numbers: one for the charblock I'm putting it and one for the tile-index in that block. For example, the tile that I want in charblock 0 at tile 1 shows '01', CBB 1 tile 0 shows '10', CBB 1, tile 1 has '11', etc. I have twelve tiles in total, 4 s-tiles to be used for BG 0 and 8 d-tiles for BG 1.

Now, I have six pairs of tiles and I intend to place them in the first tiles of each of the 6 charblock (except for CBBs 0 and 2, where tile 0 would be used as default tiles for the background, which I want to keep empty). Yes six, I'm loading into the sprite charblocks as well. I could do this by hand, calculating all the addresses manually (`0600:0020` for CBB 0, tile 1, etc) and hope I don't make a mistake and can remember what I'm doing when revisiting the demo later, or I can just use my `tile_mem` and `tile8_mem` memory map matrices and get the addresses quickly and without any hassle. Even better, C allows struct assignments so I can load the individual tiles with a simple assignment! That is exactly what I'm doing in `load_tiles()`. The source tiles are cast to `TILE` and `TILE8` arrays for 4bpp and 8bpp tiles respectively. After that, loading the tiles is very simple indeed.

The maps themselves are created in `init_maps()`. The only thing I'm interested in for this demo is to show how and which charblocks are used, so the particulars of the map aren't that important. The only thing I want them to do is to be able to show the tiles that I loaded in `load_tiles()`. The two pointers I create here, `se4` and `se8`, point to screen-entries in the screenblocks used for BG 0 and BG 1, respectively. BG 0's map, containing s-tiles, uses 1 and 512 offsets; BG 1's entries, 8bpp tiles, carries 1 and 256 offsets. If what I said before about tile-index for different bitdepths is true, then you should see the contents of all the loaded tiles. And looking at the result of the demo (fig 9.7), it looks as if I did my math correctly: background tile-indices follow the bg's assigned bitdepth, in contrast to sprites which always counts in 32 byte offsets.

There is, however, one point of concern: on hardware, you won't see the tiles that are actually in object VRAM (blocks 4 and 5). While you might expect to be able to use the sprite blocks for backgrounds due to the addresses, the actual wiring in the GBA seems to forbid it. This is why you should test on hardware is important: emulators aren't always perfect. But if hardware testing is not available to you, test on multiple emulators; if you see different behaviour, be wary of the code that produced it.
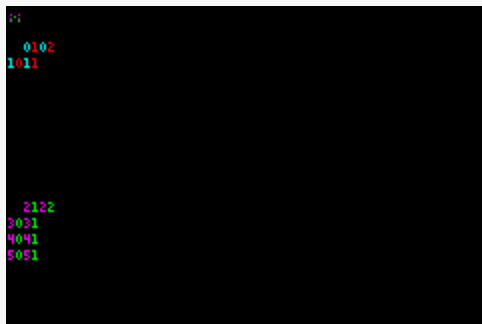

Fig 9.7a: *cbb_demo* on obsolete emulators (such as VBA and Boycott Adv).
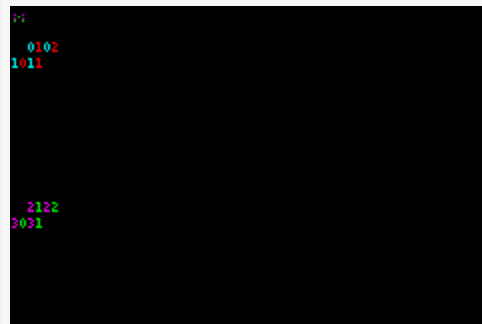

Fig 9.7b: *cbb_demo* on hardware. Spot the differences!

Bonus demo: the 'text' in text bg and introducing libtonc

Woo, bonus demo! This example will serve a number of purposes. The first is to introduce libtonc, a library of code to make life on the GBA a bit easier. In past demos,

I've been using *toolbox.h/c* to store useful macros and functions. This is alright for very small projects, but as code gets added, it becomes very hard to maintain everything. It's better to store common functionality in libraries that can be shared among projects.

The second reason is to show how you can output text, which is obviously an important ability to have. Tonclib has an extensive list of options for text rendering – too much to explain here – but its interface is pretty easy. For details, visit the Tonc Text Engine chapter.

Anyway, here's the example.

```c
#include <stdio.h>
#include <tonc.h>

int main()
{
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    // Init BG 0 for text on screen entries.
    tte_init_se_default(0, BG_CBB(0)|BG_SBB(31));

    tte_write("#{P:72,64}");        // Goto (72, 64).
    tte_write("Hello World!");      // Print "Hello world!"

    while(1);

    return 0;
}
```


Fig 9.8a: *hello* demo.

Fig 9.8b: tileset of the *hello* demo.

Yes, it is indeed a "hello world" demo, the starting point of nearly every introductory C/C++ tutorial. However, those are usually for meant for PC platforms, which have native console functionality like `printf()` or `cout` . These do not exist for the GBA.

(Or "didn't", I should say; there are ways to make use of them nowadays. See tte:conio for details.)

Tonc's support for text goes through `tte_` functions. In this case, `tte_init_se_default()` sets up background 0 for tile-mapped text. It also loads the default 8×8 font into charblock 0 (see fig 9.8b). After that, you can write to text with `tte_write`. The sequence `#{P:x,y}` is the formatting command that TTE uses to position the cursor. There are a number of these, some of which you'll also see in later chapters.

From this point on, I'll make liberal use of libtonc's text capabilities in examples for displaying values and the like. This will mostly happen without explanation, because that won't be part of the demo. Again, to see the internals, go to the TTE chapter.

Creating and using code libraries

Using the functions themselves is pretty simple, but they are spread out over multiple files and reference even more. This makes it a hassle to find which files you need to add to the list of sources to compile a project. You could add everything, of course, but that's not a pleasant prospect either. The best solution is to pre-compile the utility code into a library.

Libraries are essentially clusters of object files. Instead of linking the objects into an executable directly, you *archive* them with arm-none-eabi-ar. The command is similar to the link step as well. Here is how you can create the library libfoo.a from objects foo.o, bar.o and baz.o.

```
# archive rule
libfoo : foo.o bar.o baz.o
    arm-none-eabi-ar -crs libfoo.a foo.o bar.o baz.o
# shorthand rule: $(AR) rcs $@ $^
```

The three flags stand for **c**reate archive, **r**eplace member and create **s**ymbol table, respectively. For more on these and other archiving flags, I will refer you to the

manual, which is part of the binutils toolset. The flags are followed by the library name, which is followed by all the objects (the 'members' you want to archive).

To use the library, you have to link it to the executable. There are two linker flags of interest here: `-L` and `-l`. Upper- and lowercase 'L'. The former, `-L` adds a library path. The lowercase version, `-l`, adds the actual library, but there is a twist here: only need the root-name of the library. For example, to link the library *libfoo.a*, use `-lfoo`. The prefix *lib* and extension *.a* are assumed by the linker.

```
 # using libfoo (assume it's in ../lib)
$(PROJ).elf : $(OBJS)
     $(LD) $^ $(LDFLAGS) -L../lib -lfoo -o $@
```

Of course, these archives can get pretty big if you dump a lot of stuff in there. You might wonder if all of it is linked when you add a library to your project. The answer is no, it is not. The linker is smart enough to use only the files which functions you're actually referencing. In the case of this demo, for example, I'm using various text functions, but none of the affine functions or tables, so those are excluded. Note that the exclusion goes by *file*, not by *function*. If you only have one file in the library (or `#include` d everything, which amounts to the same thing), everything will be linked.

I intend to use libtonc in a number of later demos. In particular, the memory map, text and copy routines will be present often. Don't worry about what they do for the demo; just focus on the core content itself. Documentation of libtonc can be found in the *libtonc* folder (`tonc/code/libtonc`) and at Tonclib's website.

BETTER COPY AND FILL ROUTINES: MEMCPY16/32 AND MEMSET16/32

Now that I am using libtonc as a library for its text routines, I might as well use it for its copy and fill routines as well. Their names are `memcpy16()` and `memcpy32()` for copies and `memset16()` and `memset32()` for fill routines. The 16 and 32 denote their preferred datatypes: halfwords and words, respectively. Their arguments are similar to the conventional `memcpy()` and `memset()`, with

the exception that the size is the number of items to be copied, rather than in bytes.

```
void memset16(void *dest, u16 hw, uint hwcount);
void memcpy16(void *dest, const void *src, uint hwcount);

void memset32(void *dest, u32 wd, uint wcount) IWRAM_CODE;
void memcpy32(void *dest, const void *src, uint wcount)
IWRAM_CODE;
```

These routines are optimized assembly so they are fast. They are also safer than the dma routines, and the BIOS routine `CpuFastSet()`. Basically, I highly recommend them, and I will use them wherever I can.

LINKER OPTIONS: OBJECT FILES BEFORE LIBRARIES

In most cases, you can change the order of the options and files freely, but in the linker's case it is important the object files of the projects are mentioned *before* the linked libraries. If not, the link will fail. Whether this is standard behaviour or if it is an oversight in the linker's workings I cannot say, but be aware of potential problems here.

In conclusion

Tilemaps are essential for most types of GBA games. They are trickier to get to grips with than the bitmap modes or sprites because there are more steps to get exactly right. And, of course, you need to be sure the editor that gave you the map actually supplied the data you were expecting. Fool around with the demos a little: run them, change the code and see what happens. For example, you could try to add scrolling code to the brin_demo so you can see the whole map. Change screen blocks, change charblock, change the bitdepth, mess up *intentionally* so you can see what can go

wrong, so you'll be prepared for it when you try your own maps. Once you're confident enough, only then start making your own. I know it's the boring way, but you will benefit from it in the long run.

10. The Affine Transformation Matrix (a.k.a. P)

- About this page
- Texture mapping and affine transformations.
- "Many of the truths we cling to depend greatly upon our own point of view."
- Finishing up
- Tonc's affine functions

About this page

As you probably know, the GBA is capable of applying geometric transformations like rotating and/or scaling to sprites and backgrounds. To set them apart from the regular items, the transformable ones are generally referred to as Rot/Scale sprites and backgrounds. The transformations are described by four parameters, `pa`, `pb`, `pc` and `pd`. The locations and exact names differ for sprites and backgrounds but that doesn't matter for now.

There are two ways of interpreting these numbers. The first is to think of each of them as individual offsets to the sprite and background data. This is how the reference documents like GBATEK and CowBite Spec describe them. The other way is to see them as the elements of a 2x2 matrix which I will refer to as **P**. This is how pretty much all tutorials describe them. These tutorials also give the following matrix for rotation and scaling:

$$(10.1) \qquad \mathbf{P} = \begin{bmatrix} p_a & p_b \\ p_c & p_d \end{bmatrix} = \begin{bmatrix} s_x \cos(\alpha) & s_y \sin(\alpha) \\ -s_x \sin(\alpha) & s_y \cos(\alpha) \end{bmatrix}$$

Now, this is indeed a rotation and scale matrix. Unfortunately, it's also the wrong one! Or at least, it probably does not do what you'd expect. For example, consider the case with a scaling of s_x = 1.5, s_y = 1.0 and a rotation of α= 45. You'd probably expect something like fig 10.2a, but what you'd actually get is fig 10.2b. The sprite has rotated, but in the wrong direction, it has shrunk rather than expanded and there's an extra shear as well. Of course, you can always say that you meant for this to happen, but that's probably not quite true.
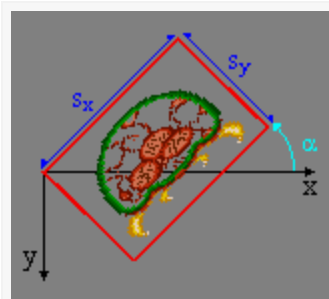
Fig 10.2a: when you say 'rotate and scale', you probably expect this…
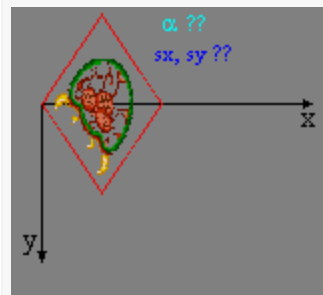
Fig 10.2b: but with **P** from eq 10.1, this is what you get.

Unfortunately, there is a lot of incorrect or misleading information on the transformation matrix around; the matrix of eq 10.1 is just one aspect of it. This actually starts with the moniker "Rot/Scale", which does not fit with what actually occurs, continues with the fact that the terms used are never properly defined and that most people often just copy-paste from others without even considering checking whether the information is correct or not. The irony is that the principle reference document, GBATEK, gives the correct descriptions of each of the elements, but somehow it got lost in the translation to matrix form in the tutorials.

In this chapter, I'll provide the **correct** interpretation of the **P**-matrix; how the GBA uses it and how to construct one yourself. To do this, though, I'm going into full math-mode. If you don't know your way around vector and matrix calculations you may have some difficulties understanding the finer points of the text. There is an appendix on linear algebra for some pointers on this subject.

This is going to be a purely theoretical page: you will find nothing that relates directly to sprites or backgrounds here; that's what the next two sections are for. Once again, we will be assisted by the lovely metroid (keep in cold storage for safe use). Please mind the direction of the y-axis and the angles, and do *not* leave without reading the finishing up paragraph. This contains several key implementation details that will be ignored in the text preceding it, because they will only get in the way at that point.

> ### BE WARY OF DOCUMENTS COVERING AFFINE PARAMETERS
>
> It's true. Pretty much every document I've seen that deals with this subject is problematic in some way. A lot of them give the wrong rotate-scale matrix (namely, the one in eq 10.1), or misname and/or misrepresent the matrix and its elements.

Texture mapping and affine transformations.

General 2D texture mapping

What the GBA does to get sprites and tiled backgrounds on screen is very much like texture mapping. So forget about the GBA right now and look at how texture mapping is done. In fig 10.4a, we see a metroid texture. For convenience I am using the standard Cartesian 2D coordinate system (y-axis points up) and have normalised the texture, which means that the right and top side of the texture correspond precisely with the unit-vectors e_x and e_y (which are of length 1). The texture mapping brings **p** (in texture space) to a point **q** (in screen space). The actual mapping is done by a 2×2 matrix **A**:

$$\mathbf{q} = \mathbf{A} \cdot \mathbf{p}$$

So how do you find **A**? Well, that's actually not that hard. The matrix is formed by lining up the transformed base vectors, which are **u** and **v** (this works in any number

of dimensions, btw), so that gives us:

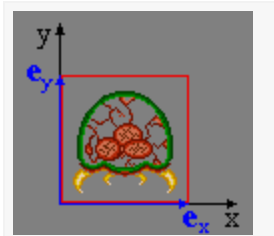$$\mathbf{A} = [\begin{array}{cc} u_x & v_x \\ u_y & v_y \end{array}]$$
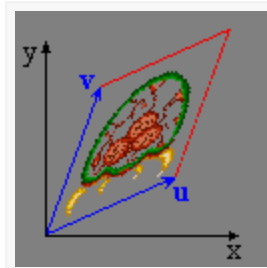

Fig 10.4a: a texture.

→

Fig 10.4b: a texture mapped

A forward texture mapping via affine matrix **A**.

Affine transformations

The transformations you can do with a 2D matrix are called *affine* transformations. The technical definition of an affine transformation is one that preserves parallel lines, which basically means that you can write them as matrix transformations, or that a rectangle will become a parallelogram under an affine transformation (see fig 10.4b).

Affine transformations include rotation and scaling, but *also* shearing. This is why I object to the name "Rot/Scale": that term only refers to a special case, not the general transformation. It is akin to calling colors shades of red: yes, reds are colors too, but not all colors are reds, and to call them that would give a distorted view of the subject.
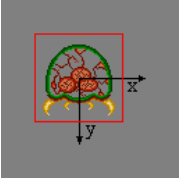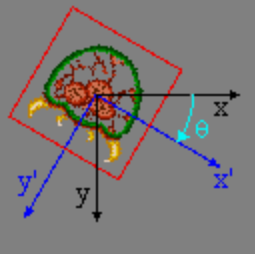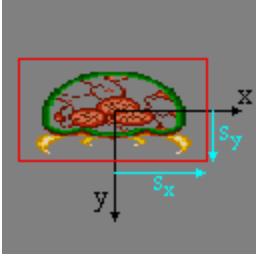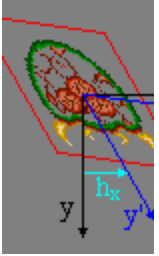
As I said, there are three basic 2d transformations, though you can always describe one of these in terms of the other two. The transformations are: rotation (**R**), scaling (**S**) and shear (**H**). Table 10.1 shows what each of the transformations does to the regular metroid sprite. The black axes are the normal base vectors (note that *y* points down!), the blue axes are the transformed base vectors and the cyan variables are the

arguments of the transformation. Also given are the matrix and inverse matrix of each transformation. Why? You'll see.

(10.2) $\mathbf{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}$ $\mathbf{A}^{-1} \equiv \frac{1}{ad-bc}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$

Table 10.1: transformation matrices and their inverses.

Identity	Rotation	Scaling	S
$\mathbf{I} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	$\mathbf{R}(\theta) = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$	$\mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}$	$\mathbf{H}(h_x, h_y)$
$\mathbf{I}^{-1} = \mathbf{I}$	$\mathbf{R}^{-1}(\theta) = \mathbf{R}(-\theta)$	$\mathbf{S}^{-1}(s_x, s_y) = \mathbf{S}(\frac{1}{s_x}, \frac{1}{s_y})$	$\mathbf{H}^{-1}(h_x, h_y$

We can now use these definitions to find the correct matrix for enlargements by s_x and s_y, followed by a **counter-clockwise** rotation by α ($=-\theta$), by matrix multiplication.

(10.3) $\mathbf{A} = \mathbf{R}(-\alpha) \cdot \mathbf{S}(s_x, s_y) = \begin{bmatrix} s_x \cos(\alpha) & s_y \sin(\alpha) \\ -s_x \sin(\alpha) & s_y \cos(\alpha) \end{bmatrix}$

… ermm, wait a sec … I'm having this strange sense of déja-vu here …

CLOCKWISE VS COUNTERCLOCKWISE

It's a minor issue, but I have to mention it. If the definition of **R** uses a clockwise rotation, why am I suddenly using a counter-clockwise one? Well, traditionally **R** is given as that particular matrix, in which the angle runs from the x-axis towards the y-axis. Because y is downward, this comes down to clockwise.

> However, the affine routines in BIOS use a counter-clockwise rotation, and I thought it'd be a good idea to use that as a guideline for my functions.

> **NOMENCLATURE: AFFINE VS ROT/SCALE**
>
> The matrix **P** is not a rotation matrix, not a scaling matrix, but a general affine transformation matrix. Rotation and scaling may be what the matrix is mostly used for, but that does not mean they're the only things possible, as the term 'Rot/Scale' would imply.
>
> To set them apart from regular backgrounds and sprites, I suppose 'Rotation' or 'Rot/Scale' are suitable enough, just not entirely accurate. However, calling the **P**-matrix by those names is simply wrong.

"Many of the truths we cling to depend greatly upon our own point of view."

As you must have noticed, eq 10.3 is identical to eq 10.1, which I said was incorrect. So what gives? Well, if you enter this matrix into the `pa-pd` elements you do indeed get something different than what you'd expect. Only now I've proven what you were supposed to expect in the first place (namely a scaling by s_x and s_y, followed by a counter-clockwise rotation by α). The *real* question is of course, why doesn't this work? To answer this I will present two different approaches to the 2D mapping process.
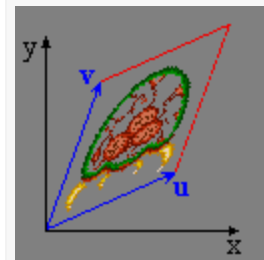
Fig 10.5: Mapping process as seen by humans. **u** and **v** are the columns of **A** (in screen space).

Human point of view

"Hello, I am Cearn's brain. I grok geometry and can do matrix- transformations in my head. Well, his head actually. When it comes to texture mapping I see the original map

(in texture space) and then visualize the transformation. I look at the original map and look at where the map's pixels end up on screen. The transformation matrix for this is **A**, which ties texel **p** to screen pixel **q** via **q** = **A** · **p**. The columns of **A** are simply the transformed unit matrices. Easy as π."

Computer point of view

"Hello, I am Cearn's GBA. I'm a lean, mean gaming machine that fits in your pocket, and I can push pixels like no one else. Except perhaps my owner's GeForce 4 Ti4200, the bloody show-off. Anyway, one of the things I do is texture mapping. And not just ordinary texture-mapping, I can do cool stuff like rotation and scaling as well. What I do is fill pixels, all I need to know is for you to tell me where I should get the pixel's color from. In other words, to fill screen pixel **q**, I need a matrix **B** that gives me the proper texel **p** via **p** = **B** · **q**. I'll happily use any matrix you give me; I have complete confidence in your ability to supply me with the matrix for the transformation you require."
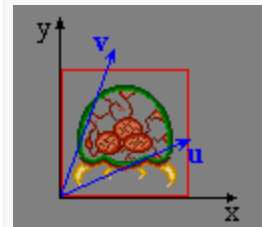

Fig 10.6: Mapping process as seen by computers. **u** and **v** (in texture space) are the columns of **B** and are mapped to the principle axes in screen space.

Resolution

I hope you spotted the crucial difference between the two points of view. **A** maps *from* texture space *to* screen space, while **B** does the exact opposite (i.e., **B** = **A**$^{-1}$). I think you know which one you should give the GBA by now. That's right: **P** = **B**, not **A**. This one bit of information is the crucial piece of the affine matrix puzzle.

So now you can figure out **P**'s elements in two ways. You can stick to the human POV and invert the matrix at the end. That's why I gave you the inverses of the affine transformations as well. You could also try to see things in the GBA's way and get the right matrix directly. Tonc's main affine functions (*tonc_video.h*, *tonc_obj_affine.c* and *tonc_bg_affine.c*) do things the GBA way, setting **P** directly; but inverted functions are also available using an " `_inv` " affix. Mind you, these are a little slower. Except for when scaling is involved; then it's a *lot* slower.

In case you're curious, the proper matrix for scale by (s_x, s_y) and counter-clockwise rotation by α is:

$$\mathbf{P} = \mathbf{A}^{-1}$$

$$\mathbf{A} = \mathbf{R}(\text{-}\alpha) \cdot \mathbf{S}(s_x, s_y) \qquad = (\mathbf{R}(\text{-}\alpha) \cdot \mathbf{S}(s_x, s_y))^{-1}$$

$$= \mathbf{S}^{-1}(s_x, s_y) \cdot \mathbf{R}^{-1}(\text{-}\alpha)$$

Using the inverse matrices given earlier, we find

$$(10.4) \qquad \mathbf{P} = \begin{bmatrix} p_a & p_b \\ p_c & p_d \end{bmatrix} = \begin{bmatrix} \frac{\cos(\alpha)}{s_x} & \frac{\text{-}\sin(\alpha)}{s_x} \\ \frac{\sin(\alpha)}{s_y} & \frac{\cos(\alpha)}{s_y} \end{bmatrix}$$

Just to make it perfectly clear:

The affine matrix **P** maps from screen space *to* texture space, not the other way around!

In other words:

p_a : texture *x*-increment / pixel

p_b : texture *x*-increment / scanline

p_c : texture *y*-increment / pixel

p_d : texture *y*-increment / scanline

Finishing up

Knowing what the **P**-matrix is used for is one thing, knowing how to use them properly is another. There are three additional points you need to remember when you're going to deal with affine objects/backgrounds and the affine matrices.

  1. Datatypes
  2. Luts

3. Initialisation

Data types of affine elements

Affine transformations are part of mathematics and, generally speaking, math numbers will be real numbers. That is to say, floating point numbers. However, if you were to use floating points for the **P** elements, you'd be in for two rude surprises.

The first one is that the matrix elements are not floats, but integers. The reason behind this is that the GBA has no floating point unit! All floating-point operations have to be done in software and without an FPU, that's going to be pretty slow. Much slower than integer math, at any rate. Now, when you think about this, it does create some problems with precision and all that. For example, the (co)sine and functions have a range between −1 and 1, a range which isn't exactly large when it comes to integers. However, the range would be much greater if one didn't count in units of 1, but in fractions, say in units of 1/256. The [−1, +1] range then becomes [−256, +256],

This strategy of representing real numbers with scaled integers is known as *fixed point arithmetic*, which you can read more about in this appendix and on wikipedia. The GBA makes use of fixed point for its affine parameters, but you can use it for other things as well. The **P**-matrix elements are 8.8 fixed point numbers, meaning a halfword with 8 integer bits and 8 fractional bits. To set a matrix to identity (1s on the diagonals, 0s elsewhere), you wouldn't use this:

```
// Floating point == Bad!!
pa= pd= 1.0;
pb= pc= 0.0;
```

but this:

```
// .8 Fixed-point == Good
pa= pd= 1<<8;
pb= pc= 0;
```

In a fixed point system with Q fractional bits, '1' ('one') is represented by 2^Q or $1<<Q$, because simply that's how fractions work.

Now, fixed point numbers are still just integers, but there are different types of integers, and it is important to use the right ones. 8.8f are 16bit variables, so the logical choice there is `short`. However, this should be a *signed* short: `s16`, not `u16`. Sometimes is doesn't matter, but if you want to do any arithmetic with them they'd better be signed. Remember that internally the CPU works in words, which are 32bit, and the 16bit variable will be converted to that. You really want, say, a 16bit "−1" (`0xFFFF`) to turn into a 32bit "−1" (`0xFFFFFFFF`), and not "65535" (`0x0000FFFF`), which is what happens if you use unsigned shorts. Also, when doing fixed point math, it is recommended to use signed ints (the 32bit kind) for them, anything else will slow you down and you might get overflow problems as well.

> **USE 32-BIT SIGNED INTS FOR AFFINE TEMPORARIES**
>
> Of course you should use 32bit variables for everything anyway (unless you actually *want* your code to bloat and slow down). If you use 16bit variables (`short` or `s16`), not only will your code be slower because of all the extra instructions that are added to keep the variables 16bit, but overflow problems can occur much sooner.
>
> Only in the final step to hardware should you go to 8.8 format. Before that, use the larger types for both speed and accuracy.

LUTs

So fixed point math is used because floating point math is just to slow for efficient use. That's all fine and good for your own math, but what about mathematical functions like sin() and cos()? Those are still floating point internally (even worse, `double`s!), so those are going to be ridiculously slow.

Rather than using the functions directly, we'll use a time-honored tradition to weasel our way out of using costly math functions: we're going to build a *look-up table* (LUT) containing the sine and cosine values. There are a number of ways to do this. If you want an easy strategy, you can just declare two arrays of 360 8.8f numbers and fill them at initialization of your program. However, this is a poor way of doing things, for reasons explained in the section on LUTs in the appendix.

Tonclib has a single sine lut which can be used for both sine and cosine values. The lut is called `sin_lut`, a `const short` array of 512 4.12f entries (12 fractional bits), created by my excellut lut creator. In *tonc_math.h* you can find two inline functions that retrieve sine and cosine values:

```
//! Look-up a sine and cosine values
/*! \param theta Angle in [0,FFFFh] range
*    \return .12f sine value
*/

INLINE s32 lu_sin(uint theta)
{    return sin_lut[(theta>>7)&0x1FF];          }

INLINE s32 lu_cos(uint theta)
{    return sin_lut[((theta>>7)+128)&0x1FF];    }
```

Now, note the angle range: 0-10000h. Remember you don't *have* to use 360 degrees for a circle; in fact, on computers it's better to divide the circle in a power of two instead. In this case, the angle is in 2^{16} parts for compatibility with BIOS functions, which is brought down to a 512 range inside the look-up functions.

Initialization

When flagging a background or object as affine, you *must* enter at least some values into `pa-pd`. Remember that these are zeroed out by default. A zero-offset means it'll use the first pixel for the whole thing. If you get a single-colored background or sprite, this is probably why. To avoid this, set **P** to the identity matrix or any other non-zero matrix.

Tonc's affine functions

Tonclib contains a number of functions for manipulating the affine parameters of objects and backgrounds, as used by the `OBJ_AFFINE` and `BG_AFFINE` structs. Because the affine matrix is stored differently in both structs you can't set them with the same function, but the functionality is the same. In table 10.2 you can find the basic formats and descriptions; just replace *foo* with `obj_aff` or `bg_aff` and *FOO* with `OBJ` or `BG` for objects and backgrounds, respectively. The functions themselves can be found in *tonc_obj_affine.c* for objects, *tonc_bg_affine.c* for backgrounds, and inlines for both in `tonc_video.h` … somewhere.

Function	Description
void *foo*_copy(FOO_AFFINE *dst, const FOO_AFFINE *src, uint count);	Copy affine parameters
void *foo*_identity(FOO_AFFINE *oaff);	$P = I$
void *foo*_postmul(FOO_AFFINE *dst, const FOO_AFFINE *src);	Post-multiply: $D = D \cdot S$
void *foo*_premul(FOO_AFFINE *dst, const FOO_AFFINE *src);	Pre-multiply: $D = S \cdot D$
void *foo*_rotate(FOO_AFFINE *aff, u16 alpha);	Rotate counter-clockwise by α·π/8000h.
void *foo*_rotscale(FOO_AFFINE *aff, FIXED sx, FIXED sy, u16 alpha);	Scale by $\frac{1}{s_x}$ and $\frac{1}{s_y}$, then rotate counter-clockwise by α·π/8000h.
void *foo*_rotscale2(FOO_AFFINE *aff, const AFF_SRC *as);	As `foo_rotscale()`, but input stored in an `AFF_SRC` struct.
void *foo*_scale(FOO_AFFINE *aff, FIXED sx, FIXED sy);	Scale by $\frac{1}{s_x}$ and $\frac{1}{s_y}$
void *foo*_set(FOO_AFFINE *aff, FIXED pa, FIXED pb, FIXED pc, FIXED pd);	Set P's elements
void *foo*_shearx(FOO_AFFINE *aff, FIXED hx);	Shear top-side right by h_x

Function	Description
void *foo*_sheary(FOO_AFFINE *aff, FIXED hy);	Shear left-side down by h_y

Table 10.2: affine functions

Sample rot/scale function

My code for a object version of the scale-then-rotate function (à la eq 10.4) is given below. Note that it is from the computer's point of view, so that `sx` and `sy` scale down. Also, the alpha `alpha` uses 10000h/circle (i.e., the unit of α is π/8000h = 0.096 mrad, or 180/8000h = 0.0055°) and the sine lut is in .12f format, which is why the shifts by 12 are required. The background version is identical, except in name and type. If this were C++, templates would have been mighty useful here.

```c
void obj_aff_rotscale(OBJ_AFFINE *oaff, int sx, int sy, u16 alpha)
{
    int ss= lu_sin(alpha), cc= lu_cos(alpha);

    oaff->pa= cc*sx>>12;    oaff->pb=-ss*sx>>12;
    oaff->pc= ss*sy>>12;    oaff->pd= cc*sy>>12;
}
```

With the information in this chapter, you know most of what you need to know about affine matrices, starting with why they should be referred to *affine* matrices, rather than merely rotation or rot/scale or the other names you might see elsewhere. You should now know what the thing actually does, and how you can set up a matrix for the effects you want. You should also know a little bit about fixed point numbers and luts (for more, look in the appendices) and why they're Good Things; if it hadn't been clear before, you should be aware that the choice of the data types you use actually *matters*, and you should not just use the first thing that comes along.

What has not been discussed here is how you actually set-up objects and backgrounds to use affine transformation, which is what the next two chapters are for. For more on affine transformations, try searching for 'linear algebra'

11. Affine sprites

Affine sprite introduction

Essentially, **affine sprites** are still sprites. The difference with regular sprites is that you can perform an affine transformation (hence the name) on them before the rendering stage by setting the right bits in the object attributes and filling in the **P** matrix. You can read about affine transformations and the **P** matrix here. It is required reading for this section, as are the sprite and background overview and the regular sprite page.

You may wonder whether this is really worth a separate section. The short answer is yes. A longer answer is yes, because using affine sprites involves a lot more math than regular sprites and I didn't want to freak out the, erm, 'mathematically challenged'. The section on regular sprites can stand on its own and you can use it in blissful ignorance of the nasty math that it required for affine sprites.

In this chapter we'll see how to set-up object to use affine transformations. This in itself is rather easy. Also discussed are a number of potential graphical problems you might run into sooner or later –one of them almost immediately, actually– and how to correct the sprite's position to make it seem like the transformation's origin is at an arbitrary point. And, as usual, there will be demo-code illustrating the various topics raised in this chapter.

Affine sprite initialization

To turn a regular sprite into an affine sprite you need to do two things. First, set `OBJ_ATTR.attr0{8}` to indicate this is a affine sprite. Second, put a number between 0 and 31 into `OBJ_ATTR.attr1{8-C}` . This number indicates which of the 32 Object Affine Matrices (`OBJ_AFFINE` structures) should be used. In case you've forgotten, the `OBJ_AFFINE` looks like this:

```
typedef struct OBJ_AFFINE
{
    u16 fill0[3];
    s16 pa;
    u16 fill1[3];
    s16 pb;
    u16 fill2[3];
    s16 pc;
    u16 fill3[3];
    s16 pd;
} ALIGN4 OBJ_AFFINE;
```

The *signed* 16-bit members `pa` , `pb` , `pc` and `pd` are 8.8 fixed point numbers that form the actual matrix, which I will refer to as **P**, in correspondence with the elements' names. For more information about this matrix, go to the affine matrix section. Do so now if you haven't already, because I'm not going to repeat it here. If all you are after is a simple scale-then-rotate matrix, try this: for a zoom by s_x and s_y followed by a counter-clockwise rotation by α, the correct matrix is this:

$$P = \begin{bmatrix} p_a & p_b \\ p_c & p_d \end{bmatrix} = \begin{bmatrix} \cos(\alpha)/s_x & -\sin(\alpha)/s_x \\ -\sin(\alpha)/s_y & \cos(\alpha)/s_y \end{bmatrix}$$

Note that the origin of the transformation is *center* of the sprite, not the top-left corner. This is worth remembering if you want to align your sprite with other objects, which we'll do later.

ESSENTIAL AFFINE SPRITE STEPS

- Set-up an object as usual: load graphics and palette, set `REG_DISPCNT`, set-up an OAM entry.
- Set bit 8 of attribute 0 to enable affinity for that object, and choose an object affine matrix to use (attribute 1, bits 8-12).
- Set that obj affine matrix to something other than all zeroes, for example the identity matrix.

Graphical artifacts

The clipping and discretization artifacts

The procedure that the GBA uses for drawing sprites is as follows: the sprite forms a rectangle on the screen defined by its size. To paint the screen pixels in that area (**q**) uses texture-pixel **p**, which is calculated via:

$$(11.1) \qquad p - p_0 = P \cdot (q - q_0)$$

where p_0 and q_0 are the centers of the sprite in texture and screen space, respectively.

The code below is essentially what the hardware does; it scans the screen-rectangle between plus and minus the half-width and half-height (half-sizes because the center is the reference point), calculates the texture-pixel and plots that color.

```
// pseudocode for affine objects
hwidth= width/2;    // half-width of object screen canvas
hheight= hheight/2;    // half-height of object screen canvas
for(iy=-hheight; iy<hheight; iy++)
{
    for(ix=-hwidth; ix<hwidth; ix++)
    {
        px= (pa*ix + pb*iy)>>8;    // get x texture coordinate
        py= (pc*ix + pd*iy)>>8;    // get y texture coordinate
        color= GetPixel(px0+px, py0+py);   // get color from (px,py)
        SetPixel(qx0+ix, qy0+iy, color);     // set color to (qx, qy)
    }
}
```

This has two main consequences, the clipping artifact and a discretization artifact.

The *clipping artifact* is caused by scanning only over the rectangle **on-screen**. But almost all transformations will cause the texture pixels to exceed that rectangle, and the pixels outside the rectangle will not be rendered. Fig 11.1 shows the screen rect (grey, blue border) and a rotated object (inside the red border). The parts that extend the blue borderlines will not be cut off.
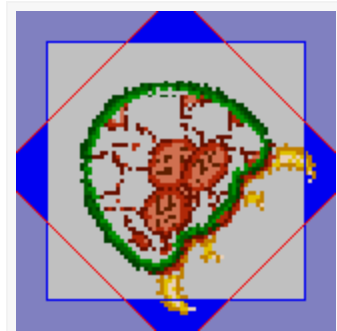
Fig 11.1: a partially defanged metroid, since the parts outside the blue square are clipped off.

As this is an obvious flaw, there is of course a way around it: set the sprite's affine mode to **double-sized affine** (`ATTR0_AFF_DBL`, `OBJ_ATTR.attr0{8,9}`). This will double the screen range of valid **q** coordinates, so you'd have + and – the width and height to play with instead of the half-sizes. This double (well quadruple, really) area means that you can safely rotate a sprite as the maximum distance from the center is ½√2 ≈ 0.707. Of course, you can still get the clipping artifact if you scale up beyond the doubled ranges. Also, note that the sprites' origin is shifted to the center of this rectangle, so that q_0 is now one full sprite-size away from the top-left corner.

The double-size flag also has a second use. Or perhaps I should say misuse. If you set it for a regular sprite, it will be hidden. This is an alternative way to hide unused sprites.

The second artifact, if you can call it that, is a *discretization* artifact. This is a more subtle point than the clipping artifact and you might not even ever notice it. The problem here is that the transformation doesn't actually take place at the center of the object, but at the **center pixel**, rounded up. As an example, look at fig 11.2. Here we have a number-line from 0 to 8; and in between them 8 pixels from 0 to 7. The number at the center is 4, of course. The central pixel is 4 as well, however its location is actually halfway between numbers 4 and 5. This creates an unbalance between the number of pixels on the left and on the right.

The center pixel is the reference point of the transformation algorithm, which has indices (ix, iy) = (0, 0). Fill that into the equations and you'll see that this is invariant under the transformation, even though mathematically it should not be. This has consequences for the offsets, which are calculated from the pixel, not the position. In fig 11.2, there are 4 pixels on the left, but only 3 on the right. A mirroring operation that would center on pixel 4 would effectively move the sprite one pixel to the right.

Fig 11.3 shows how this affects rotations. It displays lines every grey gridlines every 8 pixels and a 16×16 sprite of a box. Note that at the start the right and left sides do not lie on the gridlines, because the sprite's width and height is 16, not 17. The other figures are rotations in increments of 90°, which gives nice round numbers in the matrix. When rotating, the center pixel (the red dot in the middle) stays in the same position, and the rest rotate around it, and this process will carry the edges out of the designated 16×16 box of the sprite (the dashed lines).
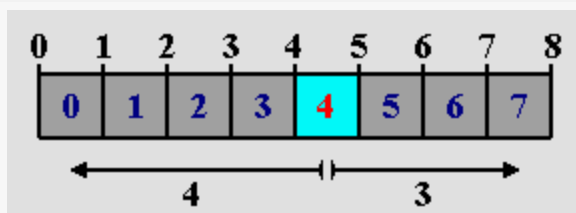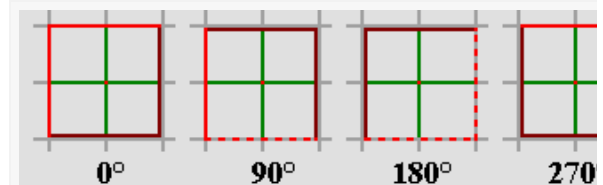

Fig 11.2: pixels are between, not on, coordinates.

Fig 11.3: Rotations in 90° increments.

THE OFFSETS MEASURE DISTANCE FROM THE CENTER PIXEL, NOT CENTER POSITION.

> The offsets that are calculated from the affine matrix use the distances from the center pixel (*w*/2, *h*/2), not the center point. As such, there is a half a pixel deviation from the mathematical transformation, which may result in a ±pixel offset for the sprite as a whole and lost texture edges.

The wrapping artifact

Apart from the clipping artifact, there seems to be another; one that I have actually never seen mentioned anywhere. It's what I call the wrapping artifact. As you know, the position for sprites is given in a 9-bit *x*-value and an 8-bit *y*-value, which values wrap around the screen. For *x*, you can just interpret this as having the [-256, 255] range. For *y* values, you can't really do that because the top value for a signed 8-bit integer is 127, which would mean that you'd never be able to put a sprite at the bottom 32 lines. But since the values wrap around, it all works out in the end anyway. With one exception.

There's never any trouble with regular sprites, and hardly any for affine sprites; the one exception is when you have a 64×64 or 32×64 affine sprite with the double size flag switched on. Such a sprite has a bounding box of 128×128. Now there are three different ways of interpreting the meaning of *y* > 128:

1. Full-wrap: the top of the sprite would show at the bottom of the screen, and vice versa.
2. Positive precedence: consider the [128, 159] range as indicative of the bottom of the screen, and forget the wrap.
3. Negative precedence: if *y* value would make the sprite appear partially at the top, consider it to be negative, again neglecting the wrap.

As it happens, the GBA uses the third interpretation. In other words, it uses

```
// pseudo code
if(oam.y + bbox_height > 256)
    oam.y -= 256;
```

Note, by the way, that some older emulators (VBA and BoycottAdvance) both use interpretation #2, which may seem more logical, but is incorrect. As you can tell, it can only happen with a 32×64 or 64×64, double-sized sprite, and even then you'll only notice it under very specific conditions, namely if the transformed sprite has visible pixels inside the top 32 lines of the bounding box. In the case that you have this problem, as far as I can tell the only way to get the sprite showing at the bottom of the screen is if you reduce the height to 32 for the time being.

A very (af)fine demo

I have a really interesting demo for you this time called *obj_aff*. It features a normal (boxed) metroid, which can be scaled, rotated and scaled. Because these transformations are applied to the *current* state of the matrix, you can end up with every kind of affine matrix possible by concatenating the different matrices. The controls are as follows:

L,R	Rotates the sprite CCW and CW, respectively.
D-pad	Shears the sprite.
D-pad+Sel	Moves sprite around.
A,B	Expands horizontally or vertically, respectively.
A,B+Sel	Shrinks horizontally or vertically, respectively. (I ran out of buttons, so had to do it like this).
Start	Toggles double-size flag. Note that a) the corners of a rotated sprite are no longer clipped and b) the position shifts by 1/2 sprite size.
Start+Sel	Resets **P** to normal.
Select	Control button (see A, B and Start).

The interesting point of seeing the transformations back to back is that you can actually see the difference between, for example, a scaling followed by a rotation

(**A**=**S·R**), and a rotate-then-scale (**A**=**R·S**). Fig 11.4 and fig 11.5 show this difference for a 45° rotation and a 2× vertical scale. Also, note that the corners are cut off here: the clipping artifact at work – even though I've already set the double-size flag here.
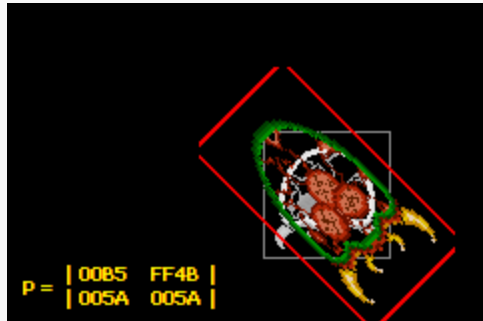
Fig 11.4: obj_aff, via **S**(1,2), then **R**(45°)
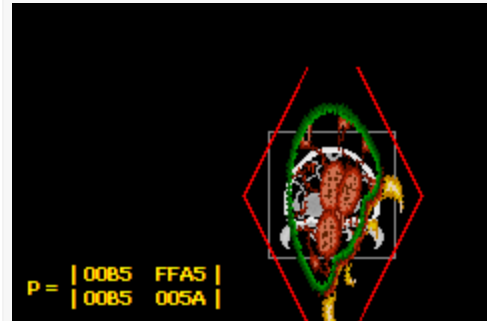
Fig 11.5: obj_aff, via **R**(45°), then **S**(1,2)

The full source code for the *obj_aff* demo is given below. It's quite long, mostly because of the amount of code necessary for managing the different affine states that can be applied. The functions that actually deal with affine sprites are `init_metr()`, `get_aff_new()` and part of the game loop in `objaff_test()`; the rest is essentially fluff required to making the whole thing work.

```c
// obj_aff.c

#include <tonc.h>
#include <stdio.h>

#include "metr.h"

OBJ_ATTR obj_buffer[128];
OBJ_AFFINE *obj_aff_buffer= (OBJ_AFFINE*)obj_buffer;


// affine transformation constants and variables
enum eAffState
{
    AFF_NULL=0, AFF_ROTATE, AFF_SCALE_X, AFF_SCALE_Y,
    AFF_SHEAR_X, AFF_SHEAR_Y, AFF_COUNT
};

// 'speeds' of transformations
const int aff_diffs[AFF_COUNT]= { 0, 128, 4, 4, 4, 4 };
// keys for transformation direction
const int aff_keys[AFF_COUNT]=
{ 0, KEY_L, KEY_SELECT, KEY_SELECT, KEY_RIGHT, KEY_UP };
int aff_state= AFF_NULL, aff_value= 0;


void init_metr()
{
    // Places the tiles of a 4bpp metroid sprite into LOW obj VRAM
    memcpy32(tile_mem[4], metr_boxTiles, metr_boxTilesLen/4);
    memcpy32(pal_obj_mem, metrPal, metrPalLen/4);

    // Set up main metroid
    obj_set_attr(obj_buffer,
        ATTR0_SQUARE | ATTR0_AFF,              // Square affine sprite
        ATTR1_SIZE_64 | ATTR1_AFF_ID(0),   // 64x64, using obj_aff[0]
        0 | 0);                            // palbank 0, tile 0
    obj_set_pos(obj_buffer, 96, 32);
    obj_aff_identity(&obj_aff_buffer[0]);

    // Set up shadow metroid
    obj_set_attr(&obj_buffer[1],
        ATTR0_SQUARE | ATTR0_AFF,              // Square affine sprite
        ATTR1_SIZE_64 | ATTR1_AFF_ID(31),  // 64x64, using obj_aff[0]
        ATTR2_PALBANK(1) | 0);             // palbank 1, tile 0
    obj_set_pos(&obj_buffer[1], 96, 32);
    obj_aff_identity(&obj_aff_buffer[31]);

    oam_update_all();
}
```

```
int get_aff_state()
{
    if(key_is_down(KEY_L | KEY_R))
        return AFF_ROTATE;
    if(key_is_down(KEY_A))
        return AFF_SCALE_X;
    if(key_is_down(KEY_B))
        return AFF_SCALE_Y;
    if(key_is_down(KEY_LEFT | KEY_RIGHT))
        return AFF_SHEAR_X;
    if(key_is_down(KEY_UP | KEY_DOWN))
        return AFF_SHEAR_Y;
    return AFF_NULL;
}

void get_aff_new(OBJ_AFFINE *oa)
{
    int diff= aff_diffs[aff_state];
    aff_value += (key_is_down(aff_keys[aff_state]) ? diff : -diff);

    switch(aff_state)
    {
    case AFF_ROTATE:     // L rotates left, R rotates right
        aff_value &= SIN_MASK;
        obj_aff_rotate(oa, aff_value);
        break;
    case AFF_SCALE_X:   // A scales x, +SELECT scales down
        obj_aff_scale_inv(oa, (1<<8)-aff_value, 1<<8);
        break;
    case AFF_SCALE_Y:   // B scales y, +SELECT scales down
        obj_aff_scale_inv(oa, 1<<8, (1<<8)-aff_value);
        break;
    case AFF_SHEAR_X:   // shear left and right
        obj_aff_shearx(oa, aff_value);
        break;
    case AFF_SHEAR_Y:   // shear up and down
        obj_aff_sheary(oa, aff_value);
        break;
    default:            // shouldn't happen
        obj_aff_identity(oa);
    }
}

void objaff_test()
{
    OBJ_ATTR *metr= &obj_buffer[0], *shadow= &obj_buffer[1];
    OBJ_AFFINE *oaff_curr= &obj_aff_buffer[0];
    OBJ_AFFINE *oaff_base= &obj_aff_buffer[1];
    OBJ_AFFINE *oaff_new=  &obj_aff_buffer[2];
```

```
    int x=96, y=32;
    int new_state;

    // oaff_curr = oaff_base * oaff_new
    // oaff_base changes when the aff-state changes
    // oaff_new is updated when it doesn't
    obj_aff_identity(oaff_curr);
    obj_aff_identity(oaff_base);
    obj_aff_identity(oaff_new);

    while(1)
    {
        key_poll();

        // move sprite around
        if( key_is_down(KEY_SELECT) && key_is_down(KEY_DIR) )
        {
            // move
            x += 2*key_tri_horz();
            y += 2*key_tri_vert();

            obj_set_pos(metr, x, y);
            obj_set_pos(shadow, x, y);
            new_state= AFF_NULL;
        }
        else    // or do an affine transformation
            new_state= get_aff_state();

        if(new_state != AFF_NULL)   // no change
        {
            if(new_state == aff_state)  // increase current
transformation
            {
                get_aff_new(oaff_new);
                obj_aff_copy(oaff_curr, obj_aff_base, 1);
                obj_aff_postmul(oaff_curr, oaff_new);
            }
            else        // switch to different transformation type
            {
                obj_aff_copy(oaff_base, oaff_curr, 1);
                obj_aff_identity(oaff_new);
                aff_value= 0;
            }
            aff_state= new_state;
        }

        // START: toggles double-size flag
        // START+SELECT: resets obj_aff to identity
        if(key_hit(KEY_START))
```

```
            {
                if(key_is_down(KEY_SELECT))
                {
                    obj_aff_identity(oaff_curr);
                    obj_aff_identity(oaff_base);
                    obj_aff_identity(oaff_new);
                    aff_value= 0;
                }
                else
                {
                    metr->attr0 ^= ATTR0_DBL_BIT;
                    shadow->attr0 ^= ATTR0_DBL_BIT;
                }
            }

            vid_vsync();

            // we only have one OBJ_ATTR, so update that
            obj_copy(obj_mem, obj_buffer, 2);

            // we have 3 OBJ_AFFINEs, update these separately
            obj_aff_copy(obj_aff_mem, obj_aff_buffer, 3);

            // Display the current matrix
            tte_printf("#{es;P:8,136}P =   "
                "#{y:-7;Ps}| %04X\t%04X#{Pr;x:72}|"
                "#{Pr;y:12}| %04X\t%04X#{Pr;p:72,12}|",
                (u16)oaff_curr->pa, (u16)oaff_curr->pb,
                (u16)oaff_curr->pc, (u16)oaff_curr->pd);
    }
}

int main()
{
    REG_DISPCNT= DCNT_BG0 | DCNT_OBJ | DCNT_OBJ_1D;
    oam_init(obj_buffer, 128);
    init_metr();

    tte_init_chr4_b4_default(0, BG_CBB(2)|BG_SBB(28));
    tte_init_con();
    tte_set_margins(8, 128, 232, 160);

    objaff_test();

    return 0;
}
```

Making the metroid an affine sprite is all done inside `init_metr()`. As you've seen how bits are set a number of times by now, it should be understandable. That said, do note that I am filling the first `OBJ_AFFINE` (the one that the sprite uses) to the identity matrix **I**. If you keep this fully zeroed-out, you'll just end up with a 64×64-pixel rectangle of uniform color. Remember that **P** contains pixel offsets; if they're all zero, there is no offset and the origin's color is used for the whole thing. In essence, the sprite is scaled up to infinity.

To be frank though, calling `obj_aff_identity()` isn't necessary after a call to `oam_init()`, as that initializes the matrices as well. Still, you need to be aware of potential problems.

That's the set-up, now for how the demo does what it does. At any given time, you will have some transformation matrix, **P**. By pressing a button (or not), a small transformation of the current state will be performed, via matrix multiplication.

$$P_{\text{new}} = P_{\text{old}} \cdot D^{-1}$$

where **D** is either a small rotation (**R**), scaling (**S**) or shear (**H**). Or a no-op (**I**). However, there is a little hitch here. This would work nice in theory, but in *practice*, it won't work well because the fixed point matrix multiplications will result in unacceptable round-off errors very very quickly. Fortunately, all these transformations have the convenient property that

$$D(a) \cdot D(b) = D(c)$$

That is to say, multiple small transformations work as one big one. All you have to do is keep track of the current chosen transformation (the variable `aff_state`, in `get_aff_state()`), modify the state variable (`aff_value`), then calculate full transformation matrix (`get_aff_new()`) and apply that (with `obj_aff_postmul()`). When a different transformation type is chosen, the current matrix is saved, the state value is reset and the whole thing continues with that state until yet another is picked.

The majority of the code is for keeping track of these changes; it's not pretty, but it gets the job done.

Off-center reference points and object combos

As mentioned earlier, affine sprites always use their centers as affine origins, but there are times when one might want to use something else to rotate around – to use another point as the reference point. Now, you can't actually do this, but you can make it *look* as if you can. To do this, I need to explain a few things about what I like to call anchoring. The *anchor* is the position that is supposed to remain 'fixed'; the spot where the texture (in this case the object) is anchored to the screen.
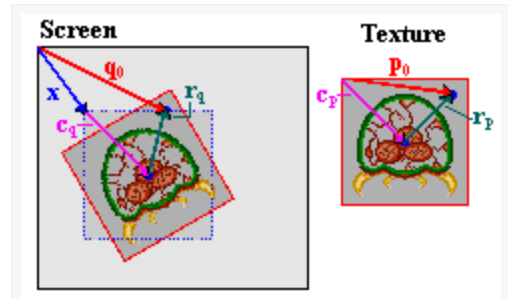
Fig 11.6: rotation of object around an off-center point.

For anchoring, you actually need one set of coordinates for each coordinate-space you're using. In this case, that's two: the texture space and the screen space. Let's call these points $\mathbf{p}_0$ and $\mathbf{q}_0$, respectively. Where these actually point *from* is largely immaterial, but for convenience' sake let's use the screen and texture origins for this. These points are only the start. In total, there are *seven* vectors that we need to take into account for the full procedure, and they are all depicted in fig 11.6. Their meanings are explained in the table below.

point	description
$\mathbf{p}_0, \mathbf{q}_0$	Anchors in texture and screen space.
$\mathbf{c}_p, \mathbf{c}_q$	Object centers in texture and screen space. With the object sizes, $\mathbf{s}$=(w,h), we have $\mathbf{c}_p$=½$\mathbf{s}$ and $\mathbf{c}_q$=$m\mathbf{s}$, where m is ½ or 1, depending on the double-size flag.

r_p, r_q	Distances between object centers and anchors. By definition, $r_p = P \cdot r_q$
x	Desired object coordinates.

Yes, it is a whole lot of vectors, but funnily enough, most are already known. The center points (c_p and c_q) can be derived from the objects size and double-size status, the anchors are known in advance because those are the input values, and r_p and r_q fit the general equation for the affine transformation, eq 11.2, so this links the two spaces. All that's left now is to write down and solve the set of equations.

$$(11.2) \quad \begin{aligned} x + c_q + r_q &= q_0 \\ c_p + r_p &= p_0 \\ r_p &= P \cdot r_q \end{aligned}$$

Three equations with three unknowns, means it is solvable. I won't post the entire derivation because that's not all that difficult; what you see in eq 11.3 is the end result in the most usable form.

$$(11.3) \quad x = q_0 - ms - P^{-1} \cdot (p_0 - \tfrac{1}{2}s)$$

The right-hand side here has three separate vectors, two of which are part of the input, a scaling flag for the double-size mode, and the inverted affine matrix. Yes, I did say inverted. This is here because the translations to position the object correctly mostly take place in screen-space. The whole term using it is merely r_q, the transformed difference between anchor and center in texture space, which you need for the final correction.

Now, this matrix inversion means two things. First, that you will likely have to set up *two* matrices: the affine matrix itself, and its inverse. For general matrices, this might take a while, especially when considering that if you want scaling, you will have to do a division somewhere. Secondly, because you only have 16 bits for the matrix elements, the inverse won't be the *exact* inverse, meaning that aligning the objects

exactly will be difficult, if not actually impossible. This is pretty much guaranteed by the hardware itself and I'll return to this point later on. For now, let's look at a function implementing eq 11.3 in the case of a 2-way scaling followed by a rotation.

```
// === in tonc_types.h ===

// This is the same struct that's used in BgAffineSet,
// where it is called BGAffineSource, even though its uses go
// beyond just backgrounds.
typedef struct tagAFF_SRC_EX
{
    s32 tex_x, tex_y;    // vector p0: anchor in texture space (.8f)
    s16 scr_x, src_y;    // vector q0: anchor in screen space (.0f)
    s16 sx, sy;          // scales (Q.8)
    u16 alpha;           // CCW angle ( integer in [0,0xFFFF] )
} AFF_SRC_EX;

// === in tonc_core.c ===
// Usage: oam_sizes[shape][size] is (w,h)
const u8 oam_sizes[3][4][2]=
{
    { { 8, 8}, {16,16}, {32,32}, {64,64} },
    { {16, 8}, {32, 8}, {32,16}, {64,32} },
    { { 8,16}, { 8,32}, {16,32}, {32,64} },
};

// === in tonc_obj_affine.c ===
void obj_rotscale_ex(OBJ_ATTR *obj, OBJ_AFFINE *oa, AFF_SRC_EX *asx)
{
    int sx= asx->sx, sy= asx->sy;
    int sina= lu_sin(asx->alpha)>>4, cosa= lu_cos(asx->alpha)>>4;

    // (1) calculate P
    oa->pa= sx*cosa>>8;     oa->pb= -sx*sina>>8;
    oa->pc= sy*sina>>8;     oa->pd=  sy*cosa>>8;

    // (2) set-up and calculate A= P^-1
    // sx = 1/sx, sy = 1/sy (.12f)
    sx= Div(1<<20, sx);
    if(sx != sy)
        sy= Div(1<<20, sy);
    else
        sy= sx;
    FIXED aa, ab, ac, ad;    // .8f
    aa=  sx*cosa>>12;    ab= sy*sina>>12;
    ac= -sx*sina>>12;    ad= sy*cosa>>12;

    // (3) get object size
    sx= oam_sizes[obj->attr0>>14][obj->attr1>>14][0];
    sy= oam_sizes[obj->attr0>>14][obj->attr1>>14][1];

    // (4) calculate dx =  q0 - ms - A*(p0-s/2)
    int dx= asx->src_x, dy= asx->src_y;    // .0f
    if(obj->attr0&ATTR0_DBL_BIT)
```

```
    {   dx -= sx;        dy -=sy;          }
    else
    {   dx -= sx>>1;     dy -= sy>>1;      }

    sx= asx->tex_x - (sx<<7);       // .8f
    sy= asx->tex_y - (sy<<7);       // .8f
    dx -= (aa*sx + ab*sy)>>16;       // .0 - (.8f*.8f/.16f)
    dy -= (ac*sx + ad*sy)>>16;       // .0 - (.8f*.8f/.16f)

    // (5) update OBJ_ATTR
    obj_set_pos(obj, dx, dy);
}
```

The `AFF_SRC_EX` struct and `oam_sizes` arrays are supporting entities of the function that does the positioning, which is `obj_rotscale_ex()`. This creates the affine matrix (`pa-pd`), and carries out all the necessary steps for eq 11.3, namely create the inverse matrix **A** (`aa-ad`), calculate all the offsets and correcting for the sizes, and finally updating the `OBJ_ATTR`. Note that the fixed point accuracy varies a lot, so it is important to comment often on this

As I said, this is not a particularly fast function; it takes roughly a scanline worth of cycles. If you need more speed, I also have a Thumb assembly version which is about 40% faster.

Affine object combo demo

The demo for this section, *oacombo,* will display three versions of essentially the same object, namely the circle of fig 11.7. The difference between them is in how they are constructed

Fig 11.7: object for oacombo.

0. 1 32×32p object, full circle.
1. 2 32×16p objects, two semi-circles.
2. 4 16×16p objects, four quarter-circles.
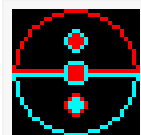
The point of this demo will be to rotate them and position the components of the combined sprites (**object combos**) as if they were a single sprite. This requires off-center anchors and therefore ties in nicely with the subject of this section. To manage the combos, I make use of the following struct.

```
typedef struct OACOMBO
{
    OBJ_ATTR *sub_obj;  // obj pointer for sub-objects
    POINT *sub_pos;     // Local sub-object coords (.8f)
    int sub_count;      // Number of sub-objects
    POINT pos;          // Global position (.8f)
    POINT anchor;       // Local anchor (.8f)
    s16 sx, sy;         // scales (.8f)
    u16 alpha;          // CCW angle
} OACOMBO;
```

Each combo is composed of `sub_count` objects; `sub_oe` is a pointer to the array storing these objects, and `sub_pos` is a pointer to the list of (top-left) coordinates of these objects, relative to the top-left of the full sprite. This global position is in `pos`. The anchor (in `anchor`) is also relative to this position. The global screen-anchor would be at `pos+anchor`, and the texture-anchor of sub-object *ii* at `anchor-sub_pos[ii]`.

The rotation will take place around the center of the circle, so that's an anchor of (16,16). Or, rather (16,16)*256 because they're .8 fixed point numbers, but that's not important right now. For the full circle, this will be the center of the object, but it'll still need to be corrected for the double-size flag. For the other combos, the anchor will *not* be at the center of their sub-objects.

Because the sub-objects share the same **P** matrix, it'd be a waste to recalculate it the whole time, so I'm using a modified version of it especially tailored to `OACOMBO` structs called `oac_rotscale()`. The code is basically the same though. The `oacs[]` array forms the three combos, which are initialized at definition because that makes things so much easier. The full circle is at (16,20), the semis at (80,20) and the one composed of quarter circles is at (48,60). The `obj_data[]` array contains the data for our seven objects, and is copied to `obj_buffer` in the initialization function. While it is generally true that magic numbers (such as using hex for OAM attributes) are evil, it is also true that they really aren't central to this story and to spend a lot of space on initializing all of them in the 'proper' fashion may actually do more harm than good …

this time. I am still using `#define` s for the anchor and a reference point though, because they appear multiple times in the rest of the code.

```c
// oacombo.c

#include <stdio.h>
#include <tonc.h>


#include "oac_gfx.h"

#define AX   (16<<8)      // X-anchor
#define AY   (16<<8)      // Y-anchor
#define X0    120         // base X
#define Y0    36          // base Y


// === GLOBALS ========================================================

OBJ_ATTR obj_buffer[128];
OBJ_AFFINE *obj_aff_buffer= (OBJ_AFFINE*)obj_buffer;

// Obj templates
const OBJ_ATTR obj_data[7]=
{
    // obj[0]  , oaff[0]: 1 full 32x32p double-affine circle
    { 0x0300, 0x8200, 0x0000, 0x0000 },
    // obj[1-2], oaff[1]: 2 32x16p double-affine semi-circles
    { 0x4300, 0x8200, 0x0000, 0x0000 },
    { 0x4300, 0x8200, 0x0008, 0x0000 },
    // obj[3-7], oaff[1]: 4 16x16p double-affine quarter-circles
    { 0x0300, 0x4400, 0x0010, 0x0000 },
    { 0x0300, 0x4400, 0x0014, 0x0000 },
    { 0x0300, 0x4400, 0x0018, 0x0000 },
    { 0x0300, 0x4400, 0x001C, 0x0000 },
};

POINT sub_pos[7]=
{
    {0,0},
    {0,0},{0,AY},
    {0,0},{AX,0}, {0,AY},{AX,AY},
};

OACOMBO oacs[3]=
{
    // full 32x32p double-affine circle
    { &obj_buffer[0], &sub_pos[0], 1,
        {(X0-48)<<8, Y0<<8}, {AX, AY}, 256, 256, 0 },
    // 2 32x16p double-affine semi-circles
    { &obj_buffer[1], &sub_pos[1], 2,
        {(X0+16)<<8, Y0<<8}, {AX, AY}, 256, 256, 0 },
    // 4 16x16p double-affine quarter-circles
    { &obj_buffer[3], &sub_pos[3], 4,
```

```
            {(X0-16)<<8, (Y0+40)<<8}, {AX, AY}, 256, 256, 0 },
};

void oac_rotscale(OACOMBO *oac)
{
    int alpha= oac->alpha;
    int sx= oac->sx, sy= oac->sy;
    int sina= lu_sin(alpha)>>4, cosa= lu_cos(alpha)>>4;

    // --- create P ---
    OBJ_AFFINE *oaff=
        &obj_aff_buffer[BF_GET(oac->sub_obj->attr1, ATTR1_AFF_ID)];
    oaff->pa=  cosa*sx>>8;     oaff->pb= -sina*sx>>8;
    oaff->pc=  sina*sy>>8;     oaff->pd=  cosa*sy>>8;

    // --- create A ---
    // sx = 1/sx, sy = 1/sy (.12f)
    sx= Div(1<<20, sx);
    if(sx != sy)
        sy= Div(1<<20, sy);
    else
        sy= sx;
    FIXED aa, ab, ac, ad;
    aa=  sx*cosa>>12;    ab= sy*sina>>12;    // .8f
    ac= -sx*sina>>12;    ad= sy*cosa>>12;    // .8f

    int ii;
    OBJ_ATTR *obj= oac->sub_obj;
    POINT *pt= oac->sub_pos;
    // --- place each sub-object ---
    for(ii=0; ii<oac->sub_count; ii++)
    {
        int dx, dy;      // all .8f
        sx= oam_sizes[obj->attr0>>14][obj->attr1>>14][0]<<7;
        sy= oam_sizes[obj->attr0>>14][obj->attr1>>14][1]<<7;

        dx= oac->pos.x+oac->anchor.x - sx;  // .8f
        dy= oac->pos.y+oac->anchor.y - sy;  // .8f

        if(obj->attr0&ATTR0_DBL_BIT)
        {   dx -= sx;    dy -= sy;    }

        sx= oac->anchor.x - pt->x - sx;
        sy= oac->anchor.y - pt->y - sy;

        dx -= (aa*sx + ab*sy)>>8;        // .8f
        dy -= (ac*sx + ad*sy)>>8;        // .8f
        BF_SET(obj->attr0, dy>>8, ATTR0_Y);
        BF_SET(obj->attr1, dx>>8, ATTR1_X);
```

```
            obj++;    pt++;
        }
    }

    void init_main()
    {
        memcpy32(pal_obj_mem, oac_gfxPal, oac_gfxPalLen/4);
        memcpy32(tile_mem[4], oac_gfxTiles, oac_gfxTilesLen/4);

        // init objs and obj combos
        oam_init();
        memcpy32(obj_buffer, obj_data, sizeof(obj_data)/4);

        REG_DISPCNT= DCNT_BG0 | DCNT_OBJ | DCNT_OBJ_1D;

        tte_init_chr4_b4_default(0, BG_CBB(2)|BG_SBB(28));
        tte_init_con();

        // Some labels
        tte_printf("#{P:%d,%d}1 full #{P:%d,%d}2 semi #{P:%d,%d}4 quarts",
            X0-48, Y0-16, X0+20, Y0-16, X0-20, Y0+74);
    }

    int main()
    {
        init_main();

        int ii, alpha=0;
        while(1)
        {
            vid_vsync();
            key_poll();
            alpha -= 128*key_tri_shoulder();

            for(ii=0; ii<3; ii++)
            {
                oacs[ii].alpha= alpha;
                oac_rotscale(&oacs[ii]);
            }
            oam_copy(oam_mem, obj_buffer, 128);
        }
        return 0;
    }
```

Fig 11.8 on the right shows a screenshot of the demo. There are three main things to point out here. First, all three objects are indeed roughly the same shape, meaning that the function(s) work. But this was never really much in doubt anyway, since it just follows the math. The second point is that there appear to be gaps in the semi- and

quarter-circle combos. If you play with the demo yourself for a while, you'll see these gaps appear and disappear seemingly at random. Meanwhile, the full-circle object looks fine throughout. Well mostly anyway.

Fig 11.8: `oacombo` in action. Note the gaps.

The cause of this is related to the third point. Compare the pixel clusters of all three circles, in particular the smaller circles within each of them. Note that even though they use the exact same **P** matrix, their formations are different! The reason for this is that while we may have positioned the sub-objects to make them form a bigger object, the pixel-mapping for each of them *still* starts at their centers. This means that the cumulative offsets that determine which source pixel is used for a given screen pixel will be different and hence you'll get a different picture, which is especially visible at the seams.

If this is a little hard to visualize, try this: open a bitmap editor and draw a single-width diagonal line. Now duplicate this with a (1, 1) pixel offset. Instead of a single thick line, you'll have two thin ones with a slit in between. The same thing happens here.

The point is that getting affine objects to align perfectly at the seams will be pretty much impossible. Alright, I suppose in some simple cases you might get away with it, and you could spend time writing code that corrects the textures to align properly, but generally speaking you should expect a hardware-caused uncertainty of about a pixel. This will be a noticeable effect at the off-center reference point, which will tend to wobble a bit, or at the seams of affine object combos, where you'll see gaps. A simple solution to the former would be to rearrange the object's tiles so that the ref-point is not off-center (sounds cheap I know, but works beautifully), or to have transparent pixels there – you can't notice something wobbling if it's invisible, after all. This would also work for the combo, which might also benefit from having the objects overlap slightly, although I haven't tried that yet. It *may* be possible to gain
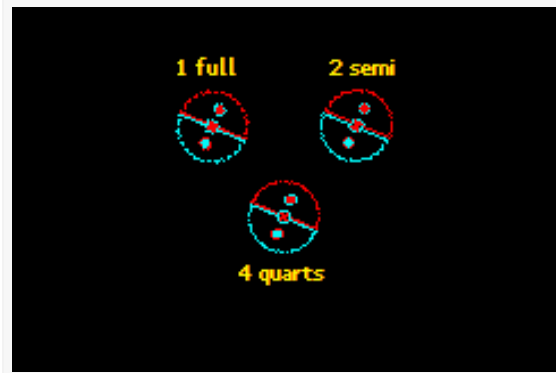
some accuracy by adding rounding terms to the calculations, but I have a hunch that it won't do that much. Feel free to try though.

Don't let all this talk of the pitfalls of affine objects get to you too much, I'm just pointing out that it might not be quite as simple as you might have hoped. So they come with a few strings, they're still pretty cool effects. When designing a game that uses them, take the issues raised in this chapter to heart and make sure your math is in order, it might save you a lot of work later on.

12. Affine backgrounds

Introduction

This section covers *affine backgrounds*: the ones on which you can perform an affine transformation via the **P** matrix. And that's all it does. If you haven't read – and understood! – the sprite/bg overview and the sections on regular backgrounds and the affine transformation matrix, do so before continuing.

If you know how to build a regular background and have understood the concepts behind the affine matrix, you should have little problems here. The theory behind an affine backgrounds are the same as for regular ones, the practice can be different at a number of very crucial points. For example, you use different registers for positioning and both the map-layout and their format are different.

Of the four backgrounds the GBA has, only the last two can be used as affine backgrounds, and only in specific video modes (see table 12.1). The sizes are also different for affine backgrounds. You can find a list of sizes in table 12.2.

mode	0	1	2
bg0	reg	reg	-
bg1	reg	reg	-
bg2	reg	aff	aff

Sz	define	(tiles)	(
00	BG_AFF_16x16	16x16	1
01	BG_AFF_32x32	32x32	2
10	BG_AFF_64x64	64x64	5

bg3	reg	-	aff

Table 12.1: video modes and background type

11	BG_AFF_128x128	128x128	10

Table 12.2: affine bg sizes

Affine background registers

Like their regular counterparts, the primary control for affine backgrounds is `REG_BGxCNT` . If you've forgotten what it does, you can read a description here. The differences with regular backgrounds are the sizes, and that `BG_WRAP` actually does something now. The other important registers are the *displacement vector* **dx** (`REG_BGxX` and `REG_BGxY`), and the *affine matrix* **P** (`REG_BGxPA` - `REG_BGxPD`). You can find their addresses in table 12.3.

Register	length	address
REG_BGxCNT	2	0400:0008h + 2·x
REG_BGxPA-PD	2	0400:0020h + 10h·(x-2)
REG_BGxX	4	0400:0028h + 10h·(x-2)
REG_BGxY	4	0400:002ch + 10h·(x-2)

Table 12.3: Affine background register addresses.
Note that *x* is 2 or 3 only!

There are a couple of things to take note of when it comes to displacement and transformation of affine backgrounds. First, the displacement *dx* uses different registers than regular backgrounds: `REG_BGxX` and `REG_BGxY` instead of `REG_BGxHOFS` and `REG_BGxVOFS` . A second point here is that they are 24.8 fixed numbers rather than pixel offsets. (Actually, they are 20.8 fixed numbers but that's not important right now.)

I usually use the affine parameters via BG_AFFINE struct instead of `REG_BGxPA` , etc. The memory map in *tonc_memmap.h* contains a `REG_BG_AFFINE` for this purpose.

Setting the registers this way is advantageous at times because you'll usually have a BG_AFFINE struct set up already, which you can then copy to the registers with a single assignment. An example of this is given below.

The elements of the affine transformation matrix **P** works exactly like they do for affine sprites: 8.8 fixed point numbers that describe the transformation from screen to texture space. However for affine backgrounds they are stored consecutively (2 byte offset), whereas those of sprites are at an 8 byte offset. You can use the `bg_aff_foo` functions from *tonc_bg_affine.c* to set them to the transformation you want.

```c
typedef struct tagBG_AFFINE
{
    s16 pa, pb;
    s16 pc, pd;
    s32 dx, dy;
} ALIGN4 BG_AFFINE;

//! BG affine params array
#define REG_BG_AFFINE    ((BG_AFFINE*)(REG_BASE+0x0000))
```

```c
// Default BG_AFFINE data (tonc_core.c)
const BG_AFFINE bg_aff_default= { 256, 0, 0, 256, 0, 0 };

// Initialize affine registers for bg 2
REG_BG_AFFINE[2] = bg_aff_default;
```

REGULAR VS AFFINE TILEMAP SCROLLING

Affine tilemaps use **different** scrolling registers! Instead of REG_BG*x*HOFS and REG_BG*x*VOFS, they use REG_BG*x*X and REG_BG*x*Y. Also, these are 32bit fixed point numbers, not halfwords.

Positioning and transforming affine backgrounds

Now that we know what the displacement and transformation registers are, now let's look at what they do. This is actually a lot trickier subject that you might think, so pay attention. Warning: this is gonna get mathematical again.

The displacement vector **dx** works the same as for regular backgrounds: **dx** contains the background-coordinates that are mapped to the screen origin. (And *not* the other way around!) However, this time **dx** is in fixed number notation. Likewise, the affine transformation matrix **P** works the same as for affine sprites: **P** describes the transformation from screen space to texture space. To put it mathematically, if we define

(12.1a)	$T(\mathbf{dx})\mathbf{p} := \mathbf{p} + \mathbf{dx}$ $T^{-1}(\mathbf{dx}) = T(-\mathbf{dx})$
(12.1b)	$\mathbf{P} = \mathbf{A}^{-1}$

then

(12.2a)	$T(\mathbf{dx})\mathbf{q} = \mathbf{p}$
(12.2b)	$\mathbf{P} \cdot \mathbf{q} = \mathbf{p}$

where

p	is a point in texture space,
q	is a point in screen space,
dx	is the displacement vector (`REG_BGxX` and `REG_BGxY`).
A	is the transformation from texture to screen space,
P	is the transformation screen from to texture space, (`REG_BGxPA` - `REG_BGxPD`).

The problem with eq 12.2 is that these only describe what happens if you use either a displacement or a transformation. So what happens if you want to use both? This is

an important question because the order of transformation matters (like we have seen in the affine sprite demo), and this is true for the order of transformation and displacement as well. As it happens, translation goes first:

(12.3)	$\mathbf{q}$	=	$\mathbf{A} \cdot \mathrm{T}(-\mathbf{dx})\, \mathbf{p}$
	$\mathrm{T}(\mathbf{dx})\, \mathbf{P} \cdot \mathbf{q}$	=	$\mathbf{p}$
	$\mathbf{dx} + \mathbf{P} \cdot \mathbf{q}$	=	$\mathbf{p}$

Another way to say this is that the transformation always uses the top left of the screen as its origin and the displacement tells which background pixels is put there. Of course, this arrangement doesn't help very much if you want to, say, rotate around some other point on the screen. To do that you'll have to pull a few tricks. To cover them all in one swoop, we'll combine eq 12.3 and the general coordinate transformation equation:

(12.4)	$\mathbf{dx} + \mathbf{P} \cdot \mathbf{q}$	=	$\mathbf{p}$	
	$\mathbf{P} \cdot (\mathbf{q} - \mathbf{q}_0)$	=	$\mathbf{p} - \mathbf{p}_0$	–
	$\mathbf{dx} + \mathbf{P} \cdot \mathbf{q}_0$	=	$\mathbf{p}_0$	
	$\mathbf{dx}$	=	$\mathbf{p}_0 - \mathbf{P} \cdot \mathbf{q}_0$	

So what the hell does *that* mean? It means that if you use this **dx** for your displacement vector, you perform your transformation around texture point $\mathbf{p}_0$, which then ends up at screen point $\mathbf{q}_0$; the $\mathbf{P} \cdot \mathbf{q}_0$ term is the correction in texture-space you have to perform to have the rotation point at $\mathbf{q}_0$ instead of (0,0). So what the hell does *that* mean? It means that before you try to use this stuff you should think about which effect you are actually trying to pull off and that you have *two* coordinate systems to work with, not one. When you do, the meaning of eq 12.4 will become apparent. In any case, the function I use for this is `bg_rotscale_ex()`, which basically looks like this:

```
typedef struct tagAFF_SRC_EX
{
    s32 tex_x, tex_y;    // vector p0: origin in texture space (24.8f)
    s16 scr_x, scr_y;    // vector q0: origin in screen space (16.0f)
    s16 sx, sy;          // scales (8.8f)
    u16 alpha;           // CCW angle ( integer in [0,0xFFFF] )
} ALIGN4 AFF_SRC_EX;

void bg_rotscale_ex(BG_AFFINE *bgaff, const AFF_SRC_EX *asx)
{
    int sx= asx->sx, sy= asx->sy;
    int sina= lu_sin(asx->alpha), cosa= lu_cos(asx->alpha);

    FIXED pa, pb, pc, pd;
    pa=  sx*cosa>>12;   pb=-sx*sina>>12;    // .8f
    pc=  sy*sina>>12;   pd= sy*cosa>>12;    // .8f

    bgaff->pa= pa;  bgaff->pb= pb;
    bgaff->pc= pc;  bgaff->pd= pd;

    bgaff->dx= asx->tex_x - (pa*asx->scr_x + pb*asx->scr_y);
    bgaff->dy= asx->tex_y - (pc*asx->scr_x + pd*asx->scr_y);
}
```

This is very similar to the `obj_rotscale_ex()` function covered in the off-center object transformation section. The math is identical, but the terms have been reshuffled a bit. The background version is actually simpler because the affine offset correction can be done in texture space instead of screen space, which means no messing about with **P**'s inverse matrix. Or with sprite-size corrections, thank IPU. For the record, yes you can apply the function directly to `REG_BG_AFFINE` .

Internal reference point registers

There's one more important thing left to mention about the displacement and transformation registers. Quoting directly from GBATEK (except the bracketed parts):

The above reference points [the displacement registers] are automatically copied to internal registers during each vblank, specifying the origin for the first scanline. The internal registers are then incremented by dmx [`REG_BGxPB`] and dmy [`REG_BGxPD`] after each scanline. Caution: Writing to a reference point

> register by software outside of the Vblank period does immediately copy the new value to the corresponding internal register, that means: in the current frame, the new value specifies the origin of the *current* scanline (instead of the topmost scanline).

Normally this won't matter to you, but if you try to write to `REG_BGxY` during an HBlank things, might not go as expected. As I learned the hard way when I tried to get my Mode 7 stuff working. This only affects affine backgrounds, though; regular ones use other registers.

Mapping format

Both the map layout and screen entries for affine backgrounds are very different from those of regular backgrounds. Ironically, they are also a lot simpler. While regular backgrounds divide the full map into quadrants (each using one full screenblock), the affine backgrounds use a flat map, meaning that the normal equation for getting a screenentry-number *n* works, making things a whole lot easier.

$$ (12.5) \qquad n = tx + ty \cdot tw $$

The screen entries themselves are also different from those of regular backgrounds as well. In affine maps, they are *1 byte long* and only contain the index of the tile to use. Additionally, you can *only* use 256 color tiles. This gives you access to all the tiles in the base charblock, but not the one(s) after it.

And that's about it, really. No, wait there's one more issue: you have to be careful when filling or changing the map because *VRAM can only be accessed 16 or 32 bits at a time*. So if you have your map stored in an array of bytes, you'll have to cast it to `u16` or `u32` first. Or use DMA. OK, now I'm done.

> **REGULAR VS AFFINE TILEMAP MAPPING DIFFERENCES**
>
> There are two important differences between regular and affine map formats. First, affine screen entries are merely one-byte tile indices. Secondly, the maps use a linear layout, rather than the division into 32x32t maps that bigger regular maps use.

sbb_aff demo

sbb_aff is to affine backgrounds what *sbb_reg* was to regular ones, with a number of extras. The demo uses a 64x64 tile affine background, shown in fig 12.1. This is divided into 16 parts of 256 bytes, each of which is filled with tiles of one color and the number of that part on it. Now, if the map-layout for affine backgrounds was the same as regular ones, each part would form a 16x16t

Fig 12.1: *sbb_aff* demo.

square. If it is a flat memory layout, each part would be a 64x16t strip. As you can see in fig 12.1, it is the latter. You can also see that, unlike regular backgrounds, this map doesn't wrap around automatically at the edges.
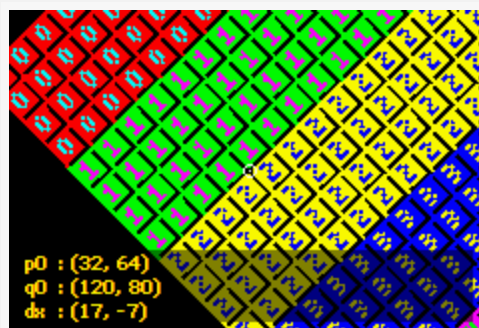
The most interesting thing about the demo are the little black and white crosshairs. The white crosshairs indicates the rotation point (the anchor). As I said earlier, you cannot simply pick a map-point $\mathbf{p}_0$ and say that that is 'the' rotation point. Well you could, but it wouldn't give the desired effect. Simply using a map-point will give you a rotating map around that point, but on screen it'll always be in the top-left corner. To move the map anchor to a specific location on the screen, you need an anchor there as well. This is $\mathbf{q}_0$. Fill both into eq 12.4 to find the displacement vector you need:

$dx = p_0 - P \cdot q_0$. This **dx** is going to be quite different from both p_0 and q_0. Its path is indicated by the black crosshairs.

The demo lets you control both p_0 and q_0. And rotation and scaling, of course. The full list of controls is.

D-pad	move map rotation point, p_0
D-pad + A	move screen rotation point, q_0
L,R	rotate the background.
B(+Se)	scale up and down.
St	Toggle wrapping flag.
St+Se	Reset anchors and **P**

```c
#include <stdio.h>
#include <tonc.h>
#include "nums.h"

#define MAP_AFF_SIZE 0x0100


// ---------------------------------------------------------------------
// GLOBALS
// ---------------------------------------------------------------------

OBJ_ATTR *obj_cross= &oam_mem[0];
OBJ_ATTR *obj_disp= &oam_mem[1];

BG_AFFINE bgaff;

// ---------------------------------------------------------------------
// FUNCTIONS
// ---------------------------------------------------------------------

void win_textbox(int bgnr, int left, int top, int right, int bottom, int bldy)
{
    REG_WIN0H= left<<8 | right;
    REG_WIN0V=  top<<8 | bottom;
    REG_WIN0CNT= WIN_ALL | WIN_BLD;
    REG_WINOUTCNT= WIN_ALL;

    REG_BLDCNT= (BLD_ALL&~BIT(bgnr)) | BLD_BLACK;
    REG_BLDY= bldy;

    REG_DISPCNT |= DCNT_WIN0;

    tte_set_margins(left, top, right, bottom);
}

void init_cross()
{
    TILE cross=
    {{
        0x00011100, 0x00100010, 0x01022201, 0x01021201,
        0x01022201, 0x00100010, 0x00011100, 0x00000000,
    }};
    tile_mem[4][1]= cross;

    pal_obj_mem[0x01]= pal_obj_mem[0x12]= CLR_WHITE;
    pal_obj_mem[0x02]= pal_obj_mem[0x11]= CLR_BLACK;

    obj_cross->attr2= 0x0001;
    obj_disp->attr2= 0x1001;
}
```

```c
void init_map()
{
    int ii;

    memcpy32(&tile8_mem[0][1], nums8Tiles, nums8TilesLen/4);
    memcpy32(pal_bg_mem, numsPal, numsPalLen/4);

    REG_BG2CNT= BG_CBB(0) | BG_SBB(8) | BG_AFF_64x64;
    bgaff= bg_aff_default;

    // fill per 256 screen entries (=32x4 bands)
    u32 *pse= (u32*)se_mem[8];
    u32 ses= 0x01010101;
    for(ii=0; ii<16; ii++)
    {
        memset32(pse, ses, MAP_AFF_SIZE/4);
        pse += MAP_AFF_SIZE/4;
        ses += 0x01010101;
    }
}

void sbb_aff()
{
    AFF_SRC_EX asx=
    {
        32<<8, 64<<8,          // Map coords.
        120, 80,               // Screen coords.
        0x0100, 0x0100, 0      // Scales and angle.
    };

    const int DX=256;
    FIXED ss= 0x0100;

    while(1)
    {
        vid_vsync();
        key_poll();

        // dir + A : move map in screen coords
        if(key_is_down(KEY_A))
        {
            asx.scr_x += key_tri_horz();
            asx.scr_y += key_tri_vert();
        }
        else    // dir : move map in map coords
        {
            asx.tex_x -= DX*key_tri_horz();
            asx.tex_y -= DX*key_tri_vert();
        }
```

```c
        // rotate
        asx.alpha -= 128*key_tri_shoulder();


        // B: scale up ; B+Se : scale down
        if(key_is_down(KEY_B))
            ss += (key_is_down(KEY_SELECT) ? -1 : 1);

        // St+Se : reset
        // St : toggle wrapping flag.
        if(key_hit(KEY_START))
        {
            if(key_is_down(KEY_SELECT))
            {
                asx.tex_x= asx.tex_y= 0;
                asx.scr_x= asx.scr_y= 0;
                asx.alpha= 0;
                ss= 1<<8;
            }
            else
                REG_BG2CNT ^= BG_WRAP;
        }

        asx.sx= asx.sy= (1<<16)/ss;

        bg_rotscale_ex(&bgaff, &asx);
        REG_BG_AFFINE[2]= bgaff;

        // the cross indicates the rotation point
        // (== p in map-space; q in screen-space)
        obj_set_pos(obj_cross, asx.scr_x-3, (asx.scr_y-3));
        obj_set_pos(obj_disp, (bgaff.dx>>8)-3, (bgaff.dy>>8)-3);

        tte_printf("#{es;P}p0\t: (%d, %d)\nq0\t: (%d, %d)\ndx\t: (%d,
%d)",
            asx.tex_x>>8, asx.tex_y>>8, asx.scr_x, asx.scr_y,
            bgaff.dx>>8, bgaff.dy>>8);
    }
}

int main()
{
    init_map();
    init_cross();

    REG_DISPCNT= DCNT_MODE1 | DCNT_BG0 | DCNT_BG2 | DCNT_OBJ;

    tte_init_chr4_b4_default(0, BG_CBB(2)|BG_SBB(28));
    tte_init_con();
    win_textbox(0, 8, 120, 232, 156, 8);
```

```
    sbb_aff();

    return 0;
}
```

13. Graphic Effects

So you know how to put sprites and backgrounds on screen, do ya? Now, how about some extra effects to liven up the place? When discussing sprites and backgrounds, we left some flags untouched, namely the mosaic and blending flags. There will be covered here. We'll also be looking into windowing, with which you can create regions to mask out backgrounds or sprites.

Mosaic

The best description of mosaic is that it makes sprites or tiles look blocky. A mosaic works in two dimensions with parameters w_m and h_m. These numbers divide your sprite or background into blocks of $w_m \times h_m$ pixels. The top-left pixel of each block is used to fill the rest of that block, which makes it blocky. Fig 13.1 shows a 1x4 mosaic for a metroid sprite. The blue lines indicate the vertical block-boundaries. The first line of each block is copied to the rest of the block, just like I said. Other examples of the mosaic effect are Zelda:LTTP when you hit an electric baddie, or Metroid Fusion when an X changes shape.
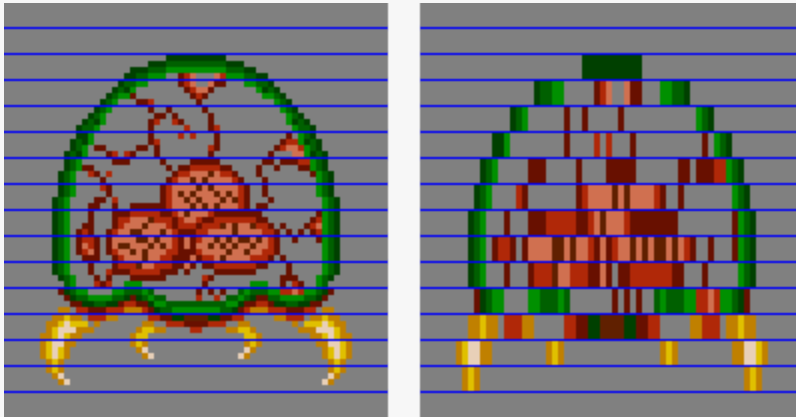
Fig 13.1: a 1×4 mosaiced metroid.

Using mosaic: sprite/bg flags and REG_MOSAIC

To use mosaic you must do two things. First, you need to enable mosaic. For individual sprites, set `OBJ_ATTR.attr0` {C}. For backgrounds, set `REG_BGxCNT` {7}. The set the mosaic levels through `REG_MOSAIC` , which looks like this:

```
REG_MOSAIC @ 0400:004Ch
```

F E D C	B A 9 8	7 6 5 4	3 2 1 0
Ov	Oh	Bv	Bh

bits	name	define	description
0-3	Bh	MOS_BH#	Horizontal BG stretch.
4-7	Bv	MOS_BV#	Vertical BG stretch.
8-B	Oh	MOS_OH#	Horizontal object stretch.
C-F	Ov	MOS_OV#	Vertical object stretch.

The *stretch* is across how many pixels the base-pixel is stretched. This corresponds to w_m*−1 or h_m*−1. With a nybble for each effect, you have stretches between 0 and 15, giving mosaic widths and heights between 1 and 16.

> **ENABLING MOSAIC**
>
> For backgrounds, set bit 7 of REG_BGxCNT. For sprites, set bit 12 in attribute 0. Then set the mosaic levels in REG_MOSAIC.

A small mosaic demo

There is a demo called *mos_demo* that illustrates the use of mosaic for both objects and backgrounds.

```
// mos_demo.c
//   bg 0, cbb  0, sbb 31, pb 0: text
//   bg 1, cbb  1, sbb 30, pb 1: bg metroid
//   oam 0: tile 0-63: obj metroid

#include <stdio.h>
#include <tonc.h>
#include "metr.h"

void test_mosaic()
{
    tte_printf("#{P:48,8}obj#{P:168,8}bg");
    tte_set_margins(4, 130, 128, 156);

    POINT pt_obj={0,0}, pt_bg={0,0};
    POINT *ppt= &pt_obj;
    while(1)
    {
        vid_vsync();

        // control the mosaic
        key_poll();

        // switch between bg or obj mosaic
        ppt= key_is_down(KEY_A) ? &pt_bg : &pt_obj;
        ppt->x += key_tri_horz();       // inc/dec h-mosaic
        ppt->y -= key_tri_vert();       // inc/dec v-mosaic

        ppt->x= clamp(ppt->x, 0, 0x80);
        ppt->y= clamp(ppt->y, 0, 0x80);

        REG_MOSAIC= MOS_BUILD(pt_bg.x>>3, pt_bg.y>>3, pt_obj.x>>3,
pt_obj.y>>3);

        tte_printf("#{es;P}obj h,v: %2d,%2d\n bg h,v: %2d,%2d",
            pt_obj.x>>3, pt_obj.y>>3, pt_bg.x>>3, pt_bg.y>>3);
    }
}

void load_metr()
{
    int ix, iy;

    memcpy32(&tile_mem[1][0], metrTiles, metrTilesLen/4);
    memcpy32(&tile_mem[4][0], metrTiles, metrTilesLen/4);
    memcpy32(pal_obj_mem, metrPal, metrPalLen/4);

    // create object: oe0
    OBJ_ATTR *metr= &oam_mem[0];
    obj_set_attr(metr, ATTR0_SQUARE | ATTR0_MOSAIC, ATTR1_SIZE_64, 0);
```

```
    obj_set_pos(metr, 32, 24);                // left-center

    // create bg map: bg1, cbb1, sbb 31

    for(ix=1; ix<16; ix++)
        pal_bg_mem[ix+16]= pal_obj_mem[ix] ^ CLR_WHITE;

    SCR_ENTRY *pse= &se_mem[30][3*32+18];    // right-center
    for(iy=0; iy<8; iy++)
        for(ix=0; ix<8; ix++)
            pse[iy*32+ix]= (iy*8+ix) | SE_PALBANK(1);

    REG_BG1CNT= BG_CBB(1) | BG_SBB(30) | BG_MOSAIC;
}

int main()
{
    // setup sprite
    oam_init(oam_mem, 128);
    load_metr();
    REG_DISPCNT= DCNT_BG0 | DCNT_BG1 | DCNT_OBJ | DCNT_OBJ_1D;

    // set-up text: bg0, cbb0, sbb31
    tte_init_chr4_b4_default(0, BG_CBB(2)|BG_SBB(31));
    tte_init_con();

    test_mosaic();
    return 0;
}
```

I use two metroids in this demo. The sprite metroid is on the left, and the background metroid with inverted colors is on the right. I've shown how to set-up sprites and backgrounds before, so you should be able to follow the steps here because it's nothing new. Well, except setting the mosaic flags in `OBJ_ATTR.attr0` and REG_BG0CNT, which I've put in bold here.
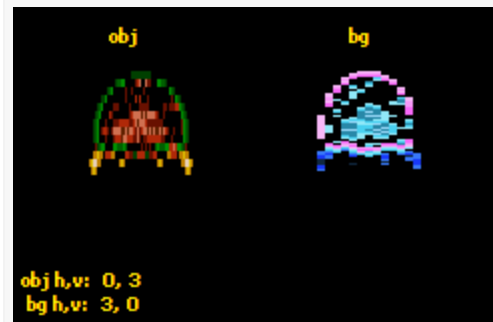

Fig 13.2: *mos_demo*.

The mosaic effect is regulated inside the `test_mosaic()`. I use two 2d points to keep track of the current level of mosaic. The D-pad is used to increase or decrease the mosaic levels; just the D-pad sets the object's mosaic and holding down A sets that of the background.

On a code design note, I could have used two if-blocks here, one for objects and one for the background, but I can also switch the mosaic context via a pointer, which saves me some code. Hurray for pointers. Also, the coordinates are in .3 fixed point format, which is how I slow down the changes in the mosaic levels. Again, I could have used timer variables and more checks to see if they had reached their thresholds, but fixed-point timers are much easier and in my view cleaner too.

You should really see the demo on hardware, by the way. Somehow both VBA and no$gba are both flawed when it comes to mosaic. After VBA 1.7.2, it has a problem with horizontal sprite mosaic. I do believe I've seen inconsistencies between hardware and scrolling mosaiced backgrounds, but can't remember where I saw it. As for no$gba, vertical mosaic appears to be disabled for both sprites and backgrounds.

EMULATORS AND MOSAIC

VBA and no$gba, the most popular GBA emulators both have problems with mosaic. Watch your step.

Blending

If you're not completely new to gaming or graphics, you may have heard of **_alpha blending_**. It allows you to combine the color values two overlapping layers, thus creating transparency (also known as semi-transparency, because something that's _completely_ transparent is invisible). Some bitmap types also come with an alpha channel, which indicates either the transparency or opacity of the pixel in question.

The basic idea behind blending is this. You have two layers, A and B, that overlap each other. Consider A to be on top of B. The color-value of the a pixel in this region is defined as

$$(13.1) \qquad C = w_A \cdot A + w_B \cdot B,$$

where w_A and w_B are the **weights** of the layers. The weights are generally normalised (between 0 and 1), with 0 being fully transparent and 1 being fully visible. It is also convenient to think of color-components in this way. Here's a few things you can do with them:

w_A	w_B	effect
1	0	layer A fully visible (hides B; standard)
0	1	layer B fully visible (or A is invisible)
α	1−α	Alpha blending. α is opacity in this case.

Note that in these examples the sum of the weights is 1, so that the final color C is between 0 (black) and 1 (white) as well. As we'll see, there are instances where you can drop out of these ranges; if this happens the values will be clipped to the standard range.

GBA Blending

Backgrounds are always enabled for blending. To enable sprite-blending, set `OBJ_ATTR.attr0` {a}. There are three registers that control blending, which unfortunately go by many different names. The ones I use are `REG_BLDCNT`, `REG_BLDALPHA` and `REG_BLDY`. Other names are `REG_BLDMOD`, `REG_COLEV` and `REG_COLEY`, and sometimes the 'E' in the last two is removed. Be warned. Anyway, the first says how and on which layers the blend should be performed, the last two contain the weights. Oh, since the GBA doesn't do floating point, the weights are fixed-point numbers in 1.4 format. Still limited by 0 and 1, of course, so there are 17 blend levels.

```
REG_BLDCNT (REG_BLDMOD) @ 0400:0050h
```

F E	D	C	B	A	9	8	7 6	5	4	3	2	1
–	bBD	bOBJ	bBG3	bBG2	bBG1	bBG0	BM	aBD	aObj	aBG3	aBG2	aBG1	a

bits	name	define	description
0-5	aBG0-aBD	*BLD_TOP#*	The A (top) layers. **BD**, by the way, is the *back drop*, a solid plane of color 0. Set the bits to make that layer use the A-weights. Note that these layers must actually be in front of the B-layers, or the blend will fail.
6-7	BM	BLD_OFF, BLD_STD, BLD_WHITE, BLD_BLACK, *BLD_MODE#*	**Blending mode**. • **00**: blending is off. • **01**: normal blend using the weights from `REG_ALPHA`. • **10**: blend A with white (fade to white) using the weight from `REG_BLDY` • **11**: blend A with black (fade to black) using the weight from `REG_BLDY`
8-D	bBG0-bBD	*BLD_BOT#*	The B (bottom) layers. Use the B-weights. Note that these layers must actually lie behind the A-layers, or the blend will not work.

The `REG_BLDALPHA` and `REG_BLDY` registers hold the blending weights in the form of eva, evb and ey, all in 1.4 fixed-point format. And no, I do not know why they are called that; they just are.

REG_BLDALPHA (REG_COLEV) @ 0400:0052h

F E D	C B A 9 8	7 6 5	4 3 2 1 0
–	evb	–	eva

bits	name	define	description
0-4	eva	*BLD_EVA#*	**Top blend weight**. Only used for normal blending

8-C	evb	BLD_EVB#	**Bottom blend weight**. Only used for normal blending

REG_BLDY (REG_COLEY) @ 0400:0054h

F E D C B A 9 8 7 6 5	4 3 2 1 0
–	ey

bits	name	define	description
0-4	ey	BLDY#	**Top blend fade**. Used for white and black fades.

Blending caveats

Blending is a nice feature to have, but keep these points in mind.

- The A layers *must* be in front of the B layers. Only then will the blend actually occur. So watch your priorities.
- In the alpha-blend mode (mode 1) the blend will only take place on the **overlapping, non-transparent** pixels of layer A and layer B. Non-overlapping pixels will still have their normal colors.
- Sprites are affected differently than backgrounds. In particular, the blend mode specified by `REG_BLDCNT` {6,7} is applied only to the *non*-overlapping sections (so that effectively only fading works). For the overlapping pixels, the standard blend is *always* in effect, regardless of the current blend-mode.
- If you are using windows, you need to set the bits 5 and/or 13 in REG_WININ or REG_WINOUT for the blending to work.

The obligatory demo

```c
// bld_demo.c

//    bg 0, cbb  0, sbb 31, pb 15: text
//    bg 1, cbb  2, sbb 30, pb 1: metroid
//    bg 2, cbb  2, sbb 29, pb 0: fence
//    oam 0: tile 0-63: obj metroid

#include <stdio.h>
#include <tonc.h>
#include "../gfx/metr.h"

void test_blend()
{
    tte_printf("#{P:48,8}obj#{P:168,8}bg");
    tte_set_margins(16, SCR_H-4-4*12, SCR_W-4, SCR_H-4);

    u32 mode=0;
    // eva, evb and ey are .4 fixeds
    // eva is full, evb and ey are empty
    u32 eva=0x80, evb= 0, ey=0;

    REG_BLDCNT= BLD_BUILD(
        BLD_OBJ | BLD_BG0,  // Top layers
        BLD_BG1,            // Bottom layers
        mode);              // Mode

    while(1)
    {
        vid_vsync();
        key_poll();

        // Interactive blend weights
        eva += key_tri_horz();
        evb -= key_tri_vert();
        ey  += key_tri_fire();

        mode += bit_tribool(key_hit(-1), KI_R, KI_L);

        // Clamp to allowable ranges
        eva = clamp(eva, 0, 0x81);
        evb = clamp(evb, 0, 0x81);
        ey  = clamp(ey, 0, 0x81);
        mode= clamp(mode, 0, 4);

        tte_printf("#{es;P}mode :\t%2d\neva :\t%2d\nevb :\t%2d\ney :\t%2d",
            mode, eva/8, evb/8, ey/8);
```

```
        // Update blend mode
        BFN_SET(REG_BLDCNT, mode, BLD_MODE);

        // Update blend weights
        REG_BLDALPHA= BLDA_BUILD(eva/8, evb/8);
        REG_BLDY= BLDY_BUILD(ey/8);
    }
}

void load_metr()
{
    // copy sprite and bg tiles, and the sprite palette
    memcpy32(&tile_mem[2][0], metrTiles, metrTilesLen/4);
    memcpy32(&tile_mem[4][0], metrTiles, metrTilesLen/4);
    memcpy32(pal_obj_mem, metrPal, metrPalLen/4);

    // set the metroid sprite
    OBJ_ATTR *metr= &oam_mem[0]; // use the first sprite
    obj_set_attr(metr, ATTR0_SQUARE | ATTR0_BLEND, ATTR1_SIZE_64, 0);
    obj_set_pos(metr, 32, 24);          // mid-center

    // create the metroid bg
    // using inverted palette for bg-metroid
    int ix, iy;
    for(ix=0; ix<16; ix++)
        pal_bg_mem[ix+16]= pal_obj_mem[ix] ^ CLR_WHITE;

    SCR_ENTRY *pse= &se_mem[30][3*32+18]; // right-center
    for(iy=0; iy<8; iy++)
        for(ix=0; ix<8; ix++)
            pse[iy*32+ix]= iy*8+ix + SE_PALBANK(1);

    REG_BG0CNT= BG_CBB(0) | BG_SBB(30);
}

// set-up the fence background
void load_fence()
{

    // tile 0 / ' ' will be a fence tile
    const TILE fence=
    {{
        0x00012000, 0x00012000, 0x00022200, 0x22220222,
        0x11122211, 0x00112000, 0x00012000, 0x00012000,
    }};
    tile_mem[2][64]= fence;
    se_fill(se_mem[29], 64);

    pal_bg_mem[0]= RGB15(16, 10, 20);
```

```
    pal_bg_mem[1]= RGB15( 0,   0, 31);
    pal_bg_mem[2]= RGB15(16, 16, 16);

    REG_BG2CNT= BG_CBB(2) | BG_SBB(29);
}

int main()
{
    oam_init(oam_mem, 128);
    load_metr();
    load_fence();

    tte_init_chr4_b4_default(0, BG_CBB(0)|BG_SBB(31));
    tte_init_con();

    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0 | DCNT_BG1 | DCNT_BG2 |
        DCNT_OBJ | DCNT_OBJ_1D;

    test_blend();

    return 0;
}
```

As always, there's a demo that goes with all this stuff. *bld_demo* features 2 metroids (the left one is a sprite, the right one (palette inverted) is on background 0) on a fence-like background (bg 1 to be precise) and lets you modify the mode, and the 3 weights independently. The mode, by the way, is given in the top left corner. The controls are:
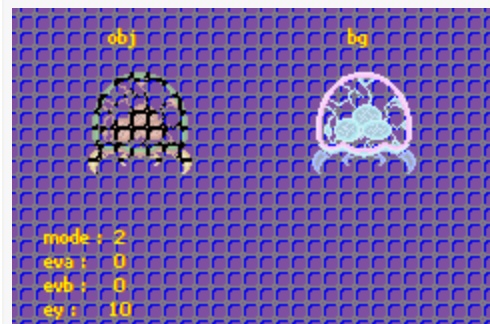
Fig 13.3: blend demo; mode=2, eva=0, evb=0, ey=10.

left, right	changes `eva` . Note that `eva` is at maximum initially.
down,up	changes `evb` .
B,A	Changes `ey`
L,R	Changes mode.

The function of interest is `test_blend()` . This is where the key handling takes place and where the blend settings are altered. Similar to *mos_demo*, .3 fixeds are used for the blend weight variables to slow the rate of change to more comfortable levels. To

set the blend registers themselves I'm using `BUILD()` macros and `BF_SET()` , which work well enough for these purposes. It would be trivially easy to write wrapper functions here of course. Most of the code is pretty standard; just play around with the blend modes and weights and see what happens.

Do take note of how, like I said earlier, the sprite metroid is affected differently than the bg-metroid. The background-background blend behaves exactly as the mode says it should; the sprite, on the other hand, always has a blend if they overlap with the fence's pixels, and the rest obeys the mode, which is what I told you in the caveats.

Windowing

Windowing allows you to divide the screen into rectangular areas known as, well, windows. There are two basic windows: *win0* and *win1*. There's also a third type of window, the *object* window. This creates a window out of the visible pixels of the sprites. You can enable the windows by setting `REG_DISPCNT` {d,e,f}, respectively.

A rectangular window is defined by its *left*, *right*, *top* and *bottom* sides. Unless you're one of *those* people, who think it's funny to say that a rectangle has only two sides: an inside and an outside. In fact, this is truer than you think. The union of win0 and win1 is the *inside* window. There's also the *outside* window, which is everything else. In other words:

winIn = win0 | win1
winOut = ~(winIn)
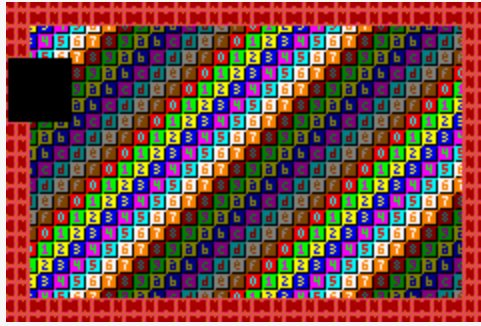
Fig 13.4a: showing win0, win1, and win_out windows.
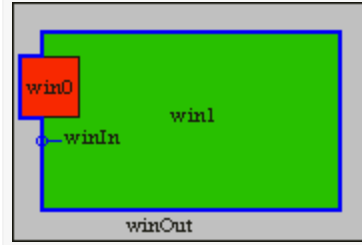
Fig 13.4b: win0 in red, win1 in green, winIn is win0 | win1 (blue edge), winOut in grey.

Window boundaries

Both win0 and win1 have 2 registers that define their boundaries. In order these are `REG_WIN0H` (`0400:0040h`), `REG_WIN1H` (`0400:0042h`), `REG_WIN0V` (`0400:0044h`) and `REG_WIN1V` (`0400:0046h`), which have the following layout:

REG_WINxH and REG_WINxV @ 0400:0040−0400:0047h

reg	F E D C B A 9 8	7 6 5 4 3 2 1 0
REG_WINxH	left	right
REG_WINxV	top	bottom

bits	name		description
0-7	right		Right side of window (exclusive)
8-F	left		Left side of window (inclusive)
-			
0-7	bottom		Bottom side of window (exclusive)
8-F	top		Top side of window (inclusive)

So you have one byte for each value. That's bytes as in *unsigned* chars. The contents of a window are drawn from starting at the top-left up to, but not including, the bottom-right. What you have to realize is that this is also true when, say, the right value is lower than the left value. In such a case, there's a wrap-around and everything on that line is inside the window, except the pixels between R and L. If both $R < L$ and $B < T$ then you get a window in the shape of a cross.

Window content

The possible content for the windows are backgrounds 0-3 and objects. No suprise there, right? In total, we have regions: win0, win1, winOut and winObj. `REG_WININ` (`0400:0048h`) controls win0 and win1, `REG_WINOUT` (`0400:004ah`) takes care of winOut and winObj. There's one bit for each content-type, plus one for blending, which you will need if you intend to use blending on the contents of that particular window.

register	F E	D	C	B	A	9	8	7 6	5	4	3	2	1
bits	–	Bld	Obj	BG3	BG2	BG1	BG0	–	Bld	Obj	BG3	BG2	BG
REG_WININ	–			win1				–			win0		
REG_WINOUT	–			winObj				–			winOut		

bits	name	define	description
0-5	BGx, Obj, Bld	WIN_BGx, WIN_OBJ, WIN_BLD, *WIN_LAYER#*	**Windowing flags**. To be used with all bytes in REG_WININ and REG_WINOUT.

There is little in the way of macros or bit-defines here because they're not really necessary. Do have these in *tonc_memdef.h* though:

```
#define WIN_BUILD(low, high)     \
    ( ((high)<<8) | (low) )

#define WININ_BUILD(win0, win1)     WIN_BUILD(win0, win1)

#define WINOUT_BUILD(out, obj)      WIN_BUILD(out, obj)
```

There are still a few things you should know about windows. First of all, when you turn on windowing in `REG_DISPCNT` , nothing will show up. There are two reasons for this. Firstly, the boundary registers are all 0, so the whole screen is basically winOut. Secondly, and this is really important: a background or object will only show up in the windows in which it is enabled! This means that unless at least *some* bits have been set in `REG_WININ` or `REG_WINOUT` nothing will show. This presents you with an effective way of hiding stuff, as we'll see in the demo. There is a third thing that you must remember, namely that win0 takes precedence over win1, which in turn takes precedence over winOut. I'm not sure how winObj fits into this yet.

WINDOWING NECESSITIES

To make windowing work for you, you need to do the following things:

- Enable windows in `REG_DISPCNT`
- Indicate in which window you want things to show up by setting the appropriate bits in `REG_WININ` and `REG_WINOUT` . You **must** set at least some bits here if you have windowing enabled, or nothing will show up at all!
- Set the desired window sizes in `REG_WINxH/V` . If you don't, everything will be considered in the Out-window.

Caveats

There's something really weird going on when either the top or bottom is outside of the screen. Multiple somethings in fact, see the demo *on hardware!* for details.

- If the top is in the [-29, 0) range (i.e., [227, 255]), the window will *not* be rendered at all. Likewise, if the bottom is inside this range, the window covers the height of the screen. I cannot say exactly what the cause is, but since the VCount also stops at 227, that might have something to do with it.
- Also, if you move the bottom from, 161 to 160, the window will also cover the whole length, but only for a frame or so.
- The points mentioned above assume *T<B*. If the top is bigger, then the effect is reversed.

WINDOWING WEIRDNESS NOT ON EMULATORS

This behaviour does *not* appear on the emulators I've tested on.

VBA clips the windows, like common sense would lead you to believe. (Of course, common sense also tells you that the Sun orbits the Earth or that the stars are pinpricks on a large black canvas. Common sense is hardly common).

MappyVM and BoycottAdvance simply remove the window if any of the boundaries goes off the screen.

Demo: there's a rocket in my pocket

In case you hadn't noticed yet, I like the Metroid series. I really like the Metroid series. If you have ever played Super Metroid chances are that you've used the X-ray scope, which let's you see through the layers and find items and secret passages with much more ease. Guess how that was done? Yup, windowing. The windowing demo *win_demo* essentially does the same thing. There's a rocket-item hidden behind the background layers and you have an X-ray rectangle which you can move around the screen so you can find it.

The controls are simple: use the D-pad to move the window around; START repositions the rocket. I've also added finer movement (A + D-pad) so you can see the strange behaviour the windows seem to exhibit at certain positions.

dir	Moves the rectangle.
A + dir	Move rectangle by tapping for finer control.
start	Randomly change the position of the rocket.

What follows below is the majority of the demo's code. I have removed the functions that set up the backgrounds and sprite because there's nothing in them that you haven't seen before already. The earlier fig 13.4a is a screenshot of the demo in action.

```
// win_demo.c

//   bg 0, cbb  0, sbb  2, pb 0: numbered forground
//   bg 1, cbb  0, sbb  3, pb 0: fenced background
//   oam 0: tile 0-3: rocket

//   win 0: objects
//   win 1: bg 0
//   win out : bg 1

#include <tonc.h>
#include "nums.h"
#include "rocket.h"

typedef struct tagRECT_U8 { u8 ll, tt, rr, bb; } ALIGN4 RECT_U8;

// window rectangle regs are write only, so buffers are necessary
// Objects in win0, BG 0 in win1
RECT_U8 win[2]=
{
    { 36, 20,  76,  60 },   // win0: 40x40 rect
    { 12, 12 ,228, 148 }    // win1: screen minus 12 margin.
};

// gfx loaders omitted for clarity
void init_front_map();  // numbers tiles
void init_back_map();   // fence
void init_rocket();     // rocket

void win_copy()
{
    REG_WIN0H= win[0].ll<<8 | win[0].rr;
    REG_WIN1H= win[1].ll<<8 | win[1].rr;
    REG_WIN0V= win[0].tt<<8 | win[0].bb;
    REG_WIN1V= win[1].tt<<8 | win[1].bb;
}

void test_win()
{
    win_copy();
    while(1)
    {
        key_poll();
        vid_vsync();

        // key_hit() or key_is_down() 'switch'
        // A depressed: move on direction press (std movement)
        // A pressed  : moves on direction hit (fine movement)
        int keys= key_curr_state();
        if(key_is_down(KEY_A))
```

```
            keys &= ~key_prev_state();

        if(keys & KEY_RIGHT)
        {    win[0].ll++;          win[0].rr++;      }
        else if(keys & KEY_LEFT )
        {    win[0].ll--;          win[0].rr--;      }
        if(keys & KEY_DOWN)
        {    win[0].tt++;          win[0].bb++;      }
        else if(keys & KEY_UP )
        {    win[0].tt--;          win[0].bb--;      }

        // (1) randomize rocket position
        if(key_hit(KEY_START))
            obj_set_pos(&oam_mem[0],
                qran_range(0, 232), qran_range(0, 152));

        win_copy();
    }
}

int main()
{
    // obvious inits
    oam_init();
    init_front_map();
    init_back_map();
    init_rocket();

    // (2) windowing inits
    REG_DISPCNT= DCNT_BG0 | DCNT_BG1 | DCNT_OBJ | DCNT_OBJ_1D |
        DCNT_WIN0 |      // Enable win 0
        DCNT_WIN1;       // Enable win 1

    REG_WININ= WININ_BUILD(WIN_OBJ, (WIN_BG0));
    REG_WINOUT= WINOUT_BUILD(WIN_BG1, 0);

    win_copy();      // Initialize window rects

    test_win();

    return 0;
}
```

Initializing the windows is done at point 2: both win0 and win1 in `REG_DISPCNT` ,
objects in win 0, bg 0 in win 1 and bg1 in winOut. The windows' sizes are set using
`win_copy()` in each frame. I am using two rectangle variables to keep track of where

the windows are, because the window-rectangle registers themselves are write only. See fig 13.4 again for the result.

Usually, objects are shown in front of backgrounds. However, because objects are now only set to appear inside win 0, they are effectively hidden everywhere else: you will only see the rocket or parts of it if the rocket and win 0's rectangle overlap. Furthermore, you will notice that because *only* objects are set for win 0, the window itself is completely black.

The rest of the demo is rather uneventful. I could explain that the way mask the variable `keys` with the previous keystate when **A** is held down lets me switch between the `key_hit()` and `key_is_down()` functions, giving me the functionality I require to switch between direct and fine motion for the X-ray window, but it's not all that interesting and quite besides the point of the demo. What's also beside the point of the demo, but *is* interesting to mention, is the randomization of the rocket's position.

Random numbers

Random numbers on a computer is a somewhat quaint notion. The whole point of a computer is to have a reliable calculator, and random numbers are pretty much the antithesis of that. Computer generated random numbers are also called *pseudo-random*, because they aren't intrinsically random, just deterministically generated to *seem* that way. There are statistical tests to see if a given routine is sufficiently random. However, this isn't nuclear physics we're talking about, this is game programming. We mostly need something that can, say, make an enemy zig or zag without any discernable pattern; that it can kill a Monte Carlo simulation is totally irrelevant.

One class of generators are *linear congruential generators*, which follow the pattern $N_{i+1} = (a{\cdot}N_i + c)\%m$, with $N_i \in [0, m\rangle$. With properly picked parameters a, c and m, the routine can be quite adequate. If you ever encounter a `rand()` function in any kind of

standard library, chances are it's one of these. Not only are these easy to implement, they are likely to be fast as well.

The following routine `qran()` is taken from my numerical methods book, Numerical Recipes, pp 275, where it is labelled a quick and dirty generator, but an adequate one. Consisting of one addition and one multiply (m=2^{32}, so done automatically), it is *very* fast. The actual number returned are the top 15 bits from N, because the upper bits are apparently more random than the lower, and also because 15 gives a [0,32767] range, which is something of an unofficial standard, AFAIK. Note that there is a second function, `sqran()` used to **seed** the generator. Since the process itself is still deterministic, you need a seed to ensure that you don't get the same sequence every time. Unless, that is, you actually *want* that to happen. This isn't such a strange idea if you think about it: you could use it to generate maps, for example. Instead of storing the whole map so that it looks the same every time you load it, you just store the seed and you're done. This is how the planetary terrains in Star Control 2 are made; I very much doubt it would have been possible to store bitmaps of all the 1000+ planets it had. This is why `sqran()` also returns the current N, so you can reset it later if necessary.

```
// from tonc_core.h/.c
// A Quick (and dirty) random number generator and its seeder

int __qran_seed= 42;      // Seed / rnd holder

// Seed routine
int sqran(int seed)
{
    int old= __qran_seed;
    __qran_seed= seed;
    return old;
}

//! Quick (and very dirty) pseudo-random number generator
/*! \return random in range [0,8000h>
*/
INLINE int qran()
{
    __qran_seed= 1664525*__qran_seed+1013904223;
    return (__qran_seed>>16) & 0x7FFF;
}
```

I'll say again, this is not a very advanced random generator, but it'll be enough for what I need. If you want a better (but slower) one, try the Mersenne Twister. You can find a nice implementation on it on PERN's new sprite page.

Ranged random numbers

Getting a random number is one thing; getting a random number in a particular range is another. It seems simple enough, of course: for a number between, say, 0 and 240 you'd use modulo 240. However, as the GBA doesn't have a hardware divide, it'll cost quite a number of cycles. Fortunately, there is a simple way out of it.

I said that `qran()`, like stdlib's `rand()` has a range between 0 and 0x8000. You can also see this as a range between 0 and 1, if you interpret them as .15 fixed point numbers. By multiplying with 240, you'll have the desired ranged random number, and it only costs a multiplication and a shift. This technique works for every random number generator, as long as you pay attention to its maximum range and integer

overflow (which you should pay attention to anyway). Tonclib's version of this is called `qran_range()`.

```
//! Ranged random number
/*! \return random in range [\a min, \a max>
*   \note (max-min) must be lower than 8000h
*/
INLINE int qran_range(int min, int max)
{    return (qran()*(max-min)>>15)+min;    }
```

In the demo, I'm using `qran_range()` twice to keep the sprite position inside the screen at all times. While the position itself could be predicted beforehand with some investigation, I don't think it'll be that easy. And if you really put that kind of effort in it, I'd say you would deserve something for your troubles. If you reload the demo a few times, you will notice that the sequence of positions is always the same. This is why they're called *pseudo*-random. To get a different sequence, the seed value should be different. I haven't even seeded it once here because it's not really important for this, but the usual trick to seed it with something involving time: for example, the number of frames or cycles before one actually starts a game, counted from the various intro screens that may precede it. Even a small difference in the seed can produce wildly varying sequences.

Conclusions

Technically speaking you probably won't really need mosaic, blending or windowing in games, but they're great for subtle effects, like a 'shock-hit' or spotlights. They're also of great use for various types of scene transitions; a fade to black can be easily implemented using the blend registers. Also fun are various HBlank effects using windows, changing the rectangles every HBlank to give beams, side-way wipes or circlular windows. However, to be able to do that you need to know how to work with interrupts. Or a special case of DMA known as HDMA, which just happens to be up next.

14. Direct Memory Access

- DMA … que?
- DMA registers
- Some DMA routines
- DMA demo : circular windows

DMA … que?

Direct Memory Access (DMA) is fast way of copying data from one place to another. Or, rather, a way of *transferring* data fast; as it can be used for copying data, but also filling memory. When you activate DMA the so-called *DMA controller* takes over the hardware (the CPU is actually halted), does the desired transfer and hands control back to the CPU before you even knew it was missing.

There are four DMA channels. Channel 0 has the highest priority; it is used for time-critical operations and can only be used with internal RAM. Channels 1 and 2 are used to transfer sound data to the right sound buffers for playback. The lowest priority channel, channel 3, is for general-purpose copies. One of the primary uses for this channel is loading in new bitmap or tile data.

DMA registers

Every kind of transfer routine needs 3 things: a source, a destination and the amount of data to copy. The *whence*, *whither* and *how much*. For DMA, the source address is put into `REG_DMAxSAD` and destination address into `REG_DMAxDAD`. A third register, `REG_DMAxCNT`, not only indicates the amount to transfer, but also controls other

features possible for DMA, like when it should start the transfer, chunk-size, and how the source and destination addresses should be updated after each individual chunk of data. All the DMA registers are 32bits in length, though they can be divided into two 16bit registers if so desired. Those of channel 0 start at `0400:00B0h` ; subsequent channels start at an offset of 12 (see table 14.1).

reg	function	address
`REG_DMAxSAD`	source	`0400:00B0h  + 0Ch·x`
`REG_DMAxDAD`	destination	`0400:00B4h + 0Ch·x`
`REG_DMAxCNT`	control	`0400:00B8h + 0Ch·x`

Table 14.1: DMA register addresses

DMA controls

The use of the source and destination registers should be obvious. The control register needs some explaining. Although the `REG_DMAxCNT` registers themselves are 32bits, they are often split into two separate registers: one for the count, and one for the actual control bits.

`REG_DMAxCNT @ 0400:00B8+12x`

1F	1E	1D 1C	1B	1A	19	18 17	16 15	14 13 12 11 10	F E D C B A 9 8 7
En	I	TM	–	CS	R	SA	DA	–	N

bits	name	define	description
00-0F	N		**Number** of transfers.
15-16	DA	DMA_DST_INC, DMA_DST_DEC, DMA_DST_FIXED, DMA_DST_RELOAD	**Destination adjustment.** • **00**: increment after each transfer (default) • **01**: decrement after each transfer • **10**: none; address is fixed

			• **11**: increment the destination during the transfer, and reset it so that repeat DMA will always start at the same destination.
17-18	SA	DMA_SRC_INC, DMA_SRC_DEC, DMA_SRC_FIXED,	**Source Adjustment**. Works just like the two bits for the destination. Note that there is no `DMA_SRC_RESET`; code 3 for source is forbidden.
19	R	DMA_REPEAT	Repeats the copy at each VBlank or HBlank if the DMA timing has been set to those modes.
1A	CS	DMA_16, DMA_32	**Chunk Size**. Sets DMA to copy by halfword (if clear) or word (if set).
1C-1D	TM	DMA_NOW, DMA_AT_VBLANK, DMA_AT_HBLANK, DMA_AT_REFRESH	**Timing Mode**. Specifies when the transfer should start. • **00**: start immediately. • **01**: start at VBlank. • **10**: start at HBlank. • **11**: Never used it so far, but here's how I gather it works. For DMA1 and DMA2 it'll refill the FIFO when it has been emptied. Count and size are forced to 1 and 32bit, respectively. For DMA3 it will start the copy at the start of each rendering line, but with a 2 scanline delay.
1E	I	DMA_IRQ	**Interrupt request**. Raise an interrupt when finished.

| 1F | En | DMA_ENABLE | **Enable** the DMA transfer for this channel. |

Source and destination addresses

The registers for source and destination addresses work just as you'd expect: just put in the proper addresses. Oh, I should tell you that the sizes for the source and destination addresses are 28 and 27 bits wide, respectively, and not the full 32. This is nothing to worry about though, you can't access addresses above `1000:0000h` anyway. For destination addresses, you can't use the section above `0800:0000h` . But then, being able to copy to ROM would be kind of strange, wouldn't it?

DMA flags

The `REG_DMAxCNT` registers can be split in two parts: one with actual flags, and one for the number of copies to do. Either way will work but you must be careful how the flags are defined: using 32-bit #defines for 16-bit registers or vice versa is not a good idea.

There are options to control what will be the next source and destination addresses when one chunk has been transferred. By default, both will increment so that it works as a copier. But you could also keep the source constant so that it'd work more as a memory fill.

What goes into the lower half of `REG_DMAxCNT` is the number of transfers. This is the number of *chunks*, not bytes! Best be careful when using `sizeof()` or something similar here, missing a factor 2 or 4 is very easy. A chunk can be either 16 or 32 bit, depending on bit 26.

More on DMA timing

What the immediate DMA does is easy to imagine, it works as soon as you enable the DMA. Well *actually* it takes 2 cycles before it'll set in, but it's close enough. The other

timing settings aren't that more difficult conceptually, but there is one point of confusion.

Consider the following situation: you want to do something cool to your otherwise standard backgrounds; specifically, you want to do something that requires the background registers to be updated every scanline. I just said that you can copy data at every HBlank (via the `DMA_AT_HBLANK` timing flag), which seems perfect for the job. If you think about it for a minute, however, you may ask yourself the following question:

When you set the timing to, say, `DMA_AT_HBLANK`, does it do *all* the *N* copies at the next HBlank, or one copy at each HBlank until the list is done?

There is a crucial difference between the two. The first option seems pointless because all copied would be done at once; if your destination is fixed (like they are for background registers), all copies except the last would be lost. In the case of the second one, how would you do more than one copy per HBlank? Clearly, something's amiss here. There is, on two counts.

For the record, I'm not 100% sure about what I'm going to say here, but I think it's pretty close to what's actually going on. The main thing to realize is that as long as the channel is not enabled (`REG_DMAxCNT` {1f} is cleared), that channel won't do squat; only after `REG_DMAxCNT` {1f} has been set will the DMA process be initiated. At the appropriate time (determined by the timing bits), DMA will do all *N* copies and then shut itself off again.

Unless, that is, the repeat-bit (`REG_DMAxCNT` {19}) is set. In that case, it will keep doing the copies at the right time until you disable the channel yourself.

Some DMA routines

While it's not that much trouble to set the three registers manually, it is preferable to hide the direct interaction in subroutines. Now, in older code, you might come across something like this:

```
// Don't do this. Please.
void dma_copy(int ch, void* src, void* dest, uint count, u32 mode)
{
    switch(ch)
    {
    case 0:
        // set DMA 0
    case 1:
        // set DMA 1
... // etc
    }
}
```

This will work, but it's not a nice way of doing things. If your switch-cases differ by a single number, you can usually replace it by a simpe lookup. There are a number of ways of fixing this, but the easiest is by mapping a struct array over the DMA registers, similar to what I did for tile memory. After that, you can just select the channel with the channel variable and simply fill in the addresses and flags.

```
typedef struct DMA_REC
{
    const void *src;
    void *dst;
    u32 cnt;
} DMA_REC;

#define REG_DMA ((volatile DMA_REC*)0x040000B0)
```

The following are my three of my DMA routines. First the `DMA_TRANSER()` macro, which is the overall macro that can be used for anything. Then two routines for general memory copies and fills using 32bit transfers with DMA 3.

```
// in tonc_core.h

//! General DMA transfer macro
#define DMA_TRANSFER(_dst, _src, count, ch, mode)    \
do {                                                 \
    REG_DMA[ch].cnt= 0;                              \
    REG_DMA[ch].src= (const void*)(_src);        \
    REG_DMA[ch].dst= (void*)(_dst);              \
    REG_DMA[ch].cnt= (count) | (mode);           \
} while(0)

//! General DMA copier
INLINE void dma_cpy(void *dst, const void *src, uint count, int ch, u32
mode)
{
    REG_DMA[3].cnt = 0; // shut off any previous transfer
    REG_DMA[3].src = src;
    REG_DMA[3].dst = dst;
    REG_DMA[3].cnt = count;
}

//! General DMA full routine
INLINE void dma_fill(void *dst, volatile u32 src, uint count, int ch,
u32 mode)
{
    REG_DMA[3].cnt = 0; // shut off any previous transfer
    REG_DMA[3].src = (const void*)&src;
    REG_DMA[3].dst = dst;
    REG_DMA[3].cnt = count | DMA_SRC_FIXED;
}

//! Word copy using DMA 3
INLINE void dma3_cpy(void *dst, const void *src, u32 size)
{   dma_cpy(dst, src, size/4, 3, DMA_CPY32);   }

//! Word fill using DMA 3
INLINE void dma3_fill(void *dst, const void *src, u32 size)
{   dma_fill(dst, src, size/4, 3, DMA_CPY32);   }
```

In all cases, I disable any previously operating transfers first. This may seem
redundant if DMA stops the CPU, but remember that DMA transfers can be timed as
well – you wouldn't want it to start in the middle of setting the registers. After that, it's
simply a matter of filling the registers. Now, it so happens that there is a 2-cycle delay
before any transfer really begins. This means that you could lose a transfer if you ask

for transfers in immediate succession. I'm not sure if this is very likely though: memory wait-states themselves already take that much time so you *should* be safe.

Other notes on these routines: the `DMA_TRANSFER()` macro's code sits between a `do {} while(0);` loop. The problem with macros is that when expanded multiple statements might break nesting-blocks. For example, calling it in the body of an `if` without braces around it would have the first line as the body, but the rest fall outside it. This kind of loop is one of the ways of preventing that. Another problem with macros is that if the arguments' names may hide other parts of the macro's code. Like the `src` and `dst` members of the `DMA_REC` struct; which is why they're underscored. The fill routines also have something remarkable going on, which you can read about in the next subsection. Lastly, the `dma3` inlines use word-transfers and take the byte-size as their last arguments, making them very similar to the standard `memcpy()` and `memset()`.

I used to have the following macro for my transfers. It uses one of the more exotic capabilities of the preprocessor: the merging-operator '##', which allows you to create symbol names at compile-time. It's scary, totally unsafe and generally unruly, but it does work. The other macro I gave is better, but I still like this thing too.

```
#define DMA_TRANSFER(_dst, _src, _count, _ch, _mode)  \
    REG_DMA##_ch##SAD = (u32)(_src),                  \
    REG_DMA##_ch##DAD = (u32)(_dst),                  \
    REG_DMA##_ch##CNT = (_count) | (_mode)            \
```

As long as you are using a literal number for `_ch` it'll form the correct register names. And yes, those comma operators between the statements actually work. They keep the statements separate, and also guard against wrongful nesting just like the `do{} while(0)` construct does.

On DMA fills

DMA can be used to fill memory, but there are two problems that you need to be aware of before you try it. The first can be caught by simply paying attention. DMA fills

don't work *quite* in the same way as `memset()` does. What you put into `REG_DMAxSAD` isn't the value you want to fill with, but its *address*!

"Very well, I'll put the value in a variable and use its address." Yes, and that brings us to our second problem, a bug which is almost impossible to find. If you try this, you'll find that it doesn't work. Well it fills with *something*, but usually not what you wanted to fill with. The full explanation is somewhat technical, but basically because you're probably only using the variable's address and not its *value*, the optimizer doesn't ever initialize it. There is a simple solution, one that we've seen before, make it volatile. Or you can use a (inline) function like `dma_fill()`, which has its source argument set as volatile so you can just insert a number just as you'd expect. Note that if you remove the volatile keyword there, it'll fail again.

In short: DMA fills need addresses, not direct values. Globals will always work, but if you use local variables or arguments you'll need to make them volatile. Note that the same thing holds true for the BIOS call CpuFastSet().

DMA; don't wear it out

DMA is fast, there's no question about that. It can be up to ten times as fast as array copies. However, think twice about using it for every copy. While it is fast, it doesn't quite blow every other transfer routine out of the water. CpuFastSet() comes within 10% of it for copies and is actually 10% faster for fills. The speed gain isn't that big a deal. Another problem is that it stops the CPU, which can screw up interrupts, causing seemingly random bugs. It does have its specific uses, usually in conjunction with timers or interrupts, but for general copies, you might consider other things as well. CpuFastSet() is a good routine, but libtonc also comes with `memcpy16()/32()` and `memset16()/32()` routines that are safer than that, and less restrictions. They are assembly routines, though, so you'll need to know how to assemble or use libraries.

DMA demo : circular windows

The demo for this chapter may seem a little complicated, but the effect is worth it. The basic use of DMA transfers is so easy that it's hardly worth having a demo of. Use of *triggered* DMA is another matter. In this case, we'll look at HBlank-triggered DMA, or HDMA for short. We'll use it to update the window sized inside the HBlank to give a circular window effect.

Fig 14.1: palette for `dma_demo`.

This is easier said than done, of course. The first step in the design is how to use HDMA for this in the first place. Because we need to copy to `REG_WIN0H` each HBlank, we need to keep the destination fixed. Technically, it needs to be *reset* to the original destination, but with only one halfword to copy this means the same thing. For the source, we'll keep track of the data that needs to be copied there in an array with one entry for each scanline, and we'll progress through the array one scanline at a time (i.e, incrementing source). And of course, the transfer has to occur at *each* scanline, so we set it to repeat. so basically we need this:

```
#define DMA_HDMA     (DMA_ENABLE | DMA_REPEAT | DMA_AT_HBLANK |
DMA_DST_RELOAD)
```

As for the circle, we need a routine that can calculate the left and right edges of a circle. There are a couple of algorithms around that can draw circles, for example Bresenham's version. We'll use a modified version of it because we only need to store the left and right points instead of drawing a pixel there. Why left-right and not top-bottom? Because the array is scanline-based, so that indicates the *y*-values already.

It doesn't really matter what you use actually, as long as you can find the edges. Once you have, all you need to do is setup the DMA in the VBlank and you're done.

The end result will show something like fig 14.1. It's the Brinstar background (again) inside the window, and a striped bg outside. The text indicates the position and radius

of the window, which can be moved with the D-pad and scaled by A and B.

```
#include
#include

#include "brin.h"

// From tonc_math.h
//#define IN_RANGE(x, min, max) ( (x) >= (min) && (x) < (max) )


// The source array
u16 g_winh[SCREEN_HEIGHT+1];

//! Create an array of horizontal offsets for a circular window.
/*! The offsets are to be copied to REG_WINxH each HBlank, either
*      by HDMA or HBlank isr. Offsets provided by modified
*      Bresenham's circle routine (of course); the clipping code is not
*      optional.
*   \param winh Pointer to array to receive the offsets.
*   \param x0   X-coord of circle origin.
*   \param y0   Y-coord of circle origin.
*   \param rr   Circle radius.
*/
void win_circle(u16 winh[], int x0, int y0, int rr)
{
    int x=0, y= rr, d= 1-rr;
    u32 tmp;

    // clear the whole array first.
    memset16(winh, 0, SCREEN_HEIGHT+1);

    while(y >= x)
    {
        // Side octs
        tmp  = clamp(x0+y, 0, SCREEN_WIDTH);
        tmp += clamp(x0-y, 0, SCREEN_WIDTH)<<8;

        if(IN_RANGE(y0-x, 0, SCREEN_HEIGHT))        // o4, o7
            winh[y0-x]= tmp;
        if(IN_RANGE(y0+x, 0, SCREEN_HEIGHT))        // o0, o3
            winh[y0+x]= tmp;

        // Change in y: top/bottom octs
        if(d >= 0)
        {
            tmp  = clamp(x0+x, 0, SCREEN_WIDTH);
            tmp += clamp(x0-x, 0, SCREEN_WIDTH)<<8;

            if(IN_RANGE(y0-y, 0, SCREEN_HEIGHT))   // o5, o6
                winh[y0-y]= tmp;
```

```
                if(IN_RANGE(y0+y, 0, SCREEN_HEIGHT))    // o1, o2
                    winh[y0+y]= tmp;

                d -= 2*(--y);
            }
            d += 2*(x++)+3;
        }
        winh[SCREEN_HEIGHT]= winh[0];
}

void init_main()
{
    // Init BG 2 (basic bg)
    dma3_cpy(pal_bg_mem, brinPal, brinPalLen);
    dma3_cpy(tile_mem[0], brinTiles, brinTilesLen);
    dma3_cpy(se_mem[30], brinMap, brinMapLen);

    REG_BG2CNT= BG_CBB(0)|BG_SBB(30);

    // Init BG 1 (mask)
    const TILE tile=
    {{
        0xF2F3F2F3, 0x3F2F3F2F, 0xF3F2F3F2, 0x2F3F2F3F,
        0xF2F3F2F3, 0x3F2F3F2F, 0xF3F2F3F2, 0x2F3F2F3F
    }};
    tile_mem[0][32]= tile;
    pal_bg_bank[4][ 2]= RGB15(12,12,12);
    pal_bg_bank[4][ 3]= RGB15( 8, 8, 8);
    pal_bg_bank[4][15]= RGB15( 0, 0, 0);
    se_fill(se_mem[29], 0x4020);

    REG_BG1CNT= BG_CBB(0)|BG_SBB(29);

    tte_init_chr4_b4_default(0, BG_CBB(2)|BG_SBB(28));
    tte_init_con();
    tte_set_margins(8, 8, 232, 40);

    // Init window
    REG_WIN0H= SCREEN_WIDTH;
    REG_WIN0V= SCREEN_HEIGHT;

    // Enable stuff
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0 | DCNT_BG1 | DCNT_BG2 |
DCNT_WIN0;
    REG_WININ= WIN_BUILD(WIN_BG0|WIN_BG2, 0);
    REG_WINOUT= WIN_BUILD(WIN_BG0|WIN_BG1, 0);
}

int main()
{
```

```
    int rr=40, x0=128, y0=120;

    init_main();

    while(1)
    {
        vid_vsync();
        key_poll();

        rr += key_tri_shoulder();   // size with B/A
        x0 += key_tri_horz();       // move left/right
        y0 += key_tri_vert();       // move up/down

        if(rr<0)
            rr= 0;

        // Fill circle array
        win_circle(g_winh, x0, y0, rr);

        // Init win-circle HDMA
        DMA_TRANSFER(&_WIN0H, &g_winh[1], 1, 3, DMA_HDMA);

        tte_printf("#{es;P}(%d,%d) | %d", x0, y0, rr);
    }

    return 0;
}
```

The initialization function is mostly just fluff. Mostly, because there is one thing of interest: the calls to `dma_cpy` to copy the Brinstar palette, tiles and map. Aside from that, nothing to see here.

The main function itself is also pretty standard. Of interest here are the call to `win_circle()`, which sets up the source-array, and to `DMA_TRANSFER()` to initialize the HDMA. Note that I'm actually making it start at `g_winh[1]` instead of just `g_winh[0]`. The reason for this is that the HBlank occurs *after* a given scanline, not before it, so we'll lag one otherwise. The `g_winh` array is actually 160+1 long, and both entry 0 and 160 describe the data for scanline 0. What's also important, but not exactly visible here, is that HDMA only occurs on the *visible* HBlanks, not the ones in the VBlank. This saves up a whole lot of trouble determining how many scanlines to correct for when setting it up.

And then there's `win_circle()`. If you're aware of how the Bresenham circle algorithm work, you know it calculates an offset for one octant and then uses it for the 7 others via symmetry rules. This happens here as well. What doesn't happen in the original probably is all the clipping (the `clamp()`s and `IN_RANGE()s`). However, these steps are absolutely vital here. Going out of bounds horizontally would mean wrong windowing offsets which would make the window turn in on itself. Going out of bounds vertically means going OOB on `g_winh` for all kind of horrible. Trust me, they are necessary.

Also, notice that I wipe the whole array clean first; this can be done inside the loop, but sometimes it's just faster to fill the whole thing first and then only update the parts you need. Lastly, as mentioned before, the first scanline's data is copied to the final entry of the array to account for the way HBlanks happen.

And here ends the chapter on DMA. The use of HDMA in this manner is great for all kinds of effects, not just circular windows. All you need is an array containing scanline-data and a function that sets it up beforehand. Be careful you don't get your channels mixed up, though.

DMA is the fastest method of copying, but as you block interrupts using `memcpy32()` is probably safer. The speed difference is only 10% anyway. DMA is also used for sound FIFO, in conjunction with timers. I can't really show you how to use it for sound, but I can tell you how timers work, and will do so in the next chapter.

15. Timers

Timing is everything

Think of every time you've heard a joke ruined because the punch line came too late or too early; think of all the failed jumps in Super Mario Bros (or any other platform game); all the occasions that you skidded at the start of a Mario Kart race for revving too soon; that your invincibility wore off just *before* you got a red shell up your a[censored]s; that you didn't quite dodge that hail of bullets in old-skool shooters because of a sudden slow-down. Think of all this and situations like them and you'll agree that in games, as in life, Timing Is Everything.

Ironically, tim*ers* are of less importance. Throughout video-game history programmers have built their games around one timing mechanism: the vertical refresh rate of the screen. In other words, the VBlank. This is a machine-oriented timer (you count frames) rather than a human-oriented one (where you'd count seconds). For consoles, this works very well as the hardware is always the same. (Except, of course, that some countries use NTSC televisions (@ 60 Hz) and others use PAL TVs (@ 50 Hz). Everyone living in the latter category and has access to both kinds knows the difference and curses the fact that it's the NTSC countries that most games stem from.) While the VBlank timer is pervasive, it is not the only one. The GBA has four clock timers at your disposal. This section covers these timers.

GBA Timers

All conceivable timers work in pretty much the same way. You have something that oscillates with a certain fixed frequency (like a CPU clock or the swing of a pendulum). After every full period, a counter is incremented and you have yourself a timer. Easy, innit?

The basic frequency of the GBA timers is the CPU frequency, which is $2^{24} \approx 16.78$ Mhz. In other words, one *clock cycle* of the CPU takes $2^{-24} \approx 59.6$ ns. Since this is a very lousy timescale for us humans, the GBA allows for 4 different frequencies (or, rather periods): 1, 64, 256 and 1024 cycles. Some details of these frequencies are shown in table 15.1. By clever use of the timer registers, you can actually create timers of any frequency, but more on that later. It should be noted that the screen refreshes every 280,896 cycles, exactly.

#cycles	frequency	period
1	16.78 MHz	59.59 ns
64	262.21 kHz	3.815 µs
256	65.536 kHz	15.26 µs
1024	16.384 kHz	61.04 µs

Table 15.1: Timer frequencies

Timer registers

The GBA has four timers, timers 0 to 3. Each of these has two registers: a data register (`REG_TMxD`) and a control register (`REG_TMxCNT`). The addresses can be found in table 15.2.

reg	function	address
`REG_TMxD`	data	`0400:0100h + 04h ·x`
`REG_TMxCNT`	control	`0400:0102h + 04h ·x`

Table 15.2: Timer register addresses

REG_TMxCNT

REG_TMxCNT @ 0400:0102 + 4*x*

F E D C B A 9 8	7	6	5 4 3	2	1 0
–	En	I	–	CM	Fr

bits	name	define	description
0-1	Fr	TM_FREQ_y	Timer **frequency**. 0-3 for 1, 64, 256, or 1024 cycles, respectively. `y` in the define is the number of cycles.
2	CM	TM_CASCADE	**Cascade mode**. When the counter of the *preceding* (`x` −1) timer overflows (`REG_TM(x-1)D=` `0xffff`), this one will be incremented too. A timer that has this bit set does *not* count on its own, though you still have to enable it. Obviously, this won't work for timer 0. If you plan on using it make sure you understand exactly what I just said; this place is a death-trap for the unwary.
6	I	TM_IRQ	Raise an interrupt on overflow.
7	En	TM_ENABLE	Enable the timer.

REG_TMxD

The data register `REG_TMxD` is a 16-bit number that works a little bit differently than you might expect at first, but in the end it makes sense. The number that you **read** from the register is the **current** timer-count. So far, so good. However, the number that you **write** to `REG_TMxD` is the **initial value** that the counter begins at when the timer is either enabled (via `TM_ENABLE`) or overflows. This has number of 'interesting' consequences. To make things a little easier, define variables *n* of the initial value (the write-number) and *c* for the current count (the read number).

First of all, when you set an *n* (of, say, `c000h`) like this:

```
    REG_TM2D= 0xc000;
```

you will *not* have set the current timer-count *c* to *n* (= `c000h`). In fact, if the timer is disabled, then *c*= 0. However, as soon as you do enable the counter, then *c* = *n* and proceeds from there. And when the timer overflows, it will reset to this value as well. By the way, because *n* is only the starting value it is important to set *n* first, and enable the timer afterwards.

Secondly, ask yourself this: what happens when you disable the timer again? Well, the counter retains its current value. However, when you *enable* it afterwards, *c* will reset to *n* again. This is a bit of a drag if you want to disable the timer for a while (during a game-pause for instance) and then pick up where it left of. Well, yeah, but there is a way to make it happen. How? By turning it into a cascade timer via `TM_CASCADE` ! Having that bit set in the `REG_TMxCNT` will cause the timer to be increased only when the preceding one overflows. If you prevent that from ever happening (if it's disabled for instance) then you will have effectively disabled your timer.

Lastly, given a certain *n*, then the timer will overflow after *T*= `10000h` −*n* increments. Or, thanks to the wonders of two's complement, just *T*= −*n*. Combined with a cascade timer (or interrupts) you can build timers of any frequency, which is what you want from a timer.

> **WRITING TO REG_TMXD IS WEIRD**
>
> Writing into REG_TMxD may not do what you think it does. It does *not* set the timer value. Rather, it sets the *initial* value for the next timer run.

Timer demo : like clockwork

In today's demo, I'm going to show how to make a simple digital clock with the timers. To do this, we'll need a 1 Hz timer. As that's not available directly, I'm going to set up a cascading timer system with timers 2 and 3. Timer 3 will be set to cascade mode, which is updated when timer 2 overflows. It is possible to set the overflow to happen at a frequency of exactly one Hertz. The clock frequency is 2^{24}, or 1024*0x4000. By setting timer 2 to `TM_FREQ_1024` and to start at −0x4000, the cascading timer 3 will effectively be a 1 Hz counter.

Whenever timer 3 is updated, the demo turns the number of seconds into hours, minutes and seconds and prints that on screen (see fig 15.1). Yes, I am using divisions and moduli here because it is the simplest procedure and I can spare the cycles in this particular demo.

The demo can be (un)paused with Select and Start. Start disables timer 2, and thus timer 3 too.

Fig 15.1: `tmr_demo` .

Select turns timer 2 into a cascade timer as well, and since timer 1 is disabled, doing this also stops timer 2 (and 3). The difference is what happens when you unpause. By disabling a timer, it will start again at the initial value; but stopping it with a cascade actually keeps the timer active and it will simply resume counting once the cascade is removed. The difference is a subtle one, but the latter is more appropriate.

```c
// Using a the "Berk" font from headspins font collection.

#include <stdio.h>
#include <tonc.h>
#include "berk.h"

void tmr_test()
{
    // Overflow every ~1 second:
    // 0x4000 ticks @ FREQ_1024

    REG_TM2D= -0x4000;          // 0x4000 ticks till overflow
    REG_TM2CNT= TM_FREQ_1024;   // we're using the 1024 cycle timer

    // cascade into tm3
    REG_TM3CNT= TM_ENABLE | TM_CASCADE;

    u32 sec= -1;

    while(1)
    {
        vid_vsync();
        key_poll();

        if(REG_TM3D != sec)
        {
            sec= REG_TM3D;
            tte_printf("#{es;P:24,60}%02d:%02d:%02d",
                sec/3600, (sec%3600)/60, sec%60);
        }

        if(key_hit(KEY_START))  // pause by disabling timer
            REG_TM2CNT ^= TM_ENABLE;

        if(key_hit(KEY_SELECT)) // pause by enabling cascade
            REG_TM2CNT ^= TM_CASCADE;
    }
}

int main()
{
    // set-up berk font
    tte_init_se(0, BG_CBB(0)|BG_SBB(31), 1, 0, 0, &berkFont, se_drawg);
    tte_init_con();
    memcpy16(pal_bg_mem, berkPal, berkPalLen/4);

    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    tmr_test();
```

```
    return 0;
}
```

This was a rather simple use of timers. Of course, I could have just as easily used the VBlank to keep track of the seconds, which is how it's usually done anyway. The hardware timers are usually reserved for timed DMA's, which are used in sound mixers, not for game timers. There is one other use that comes to mind, though, namely profiling: examining how fast your functions are. One of the text system demos uses that to check the speeds of a few copying routines.

16. Interrupts

Introduction

Under certain conditions, you can make the CPU drop whatever it's doing, go run another function instead, and continue with the original process afterwards. This process is known as an *interrupt* (two 'r's, please). The function that handles the interrupt is an *interrupt service routine*, or just interrupt; triggering one is called *raising* an interrupt.

Interrupts are often attached to certain hardware events: pressing a key on a PC keyboard, for example, raises one. Another PC example is the VBlank (yes, PCs have them too). The GBA has similar interrupts and others for the HBlank, DMA and more. This last one in particular can be used for a great deal of nifty effects. I'll give a full list of interrupts shortly.

Interrupts halt the current process, quickly do 'something', and pass control back again. Stress the word "quickly": interrupts are supposed to be short routines.

Interrupts registers

There are three registers specifically for interrupts: `REG_IE` (`0400:0200h`), `REG_IF` (`0400:0202h`) and `REG_IME` (`0400:0208h`). `REG_IME` is the master interrupt control; unless this is set to '1', interrupts will be ignored completely. To enable a specific interrupt you need to set the appropriate bit in `REG_IE`. When an interrupt occurs, the corresponding bit in `REG_IF` will be set. To acknowledge that you've handled an interrupt, the bit needs to be cleared again, but the way to do that is a little counter-intuitive to say the least. To acknowledge the interrupt, you actually have to *set* the bit again. That's right, you have to write 1 to that bit (which is already 1) in order to clear it.

Apart from setting the bits in `REG_IE`, you also need to set a bit in other registers that deal with the subject. For example, the HBlank interrupt also requires a bit in `REG_DISPSTAT`. I think (but please correct me if I'm wrong) that you need both a sender and receiver of interrupts; `REG_IE` controls the receiver and registers like `REG_DISPSTAT` control the sender. With that in mind, let's check out the bit layout for `REG_IE` and `REG_IF`.

```
REG_IE @ 0400:0200 and REG_IF @ 0400:0202
```

F E	D	C	B A 9 8	7	6 5 4 3	2	1	0
–	C	K	Dma	Com	Tm	Vct	Hbl	Vbl

bits	name	define	description
0	Vbl	IRQ_VBLANK	**VBlank** interrupt. Also requires `REG_DISPSTAT` {3}
1	Hbl	IRQ_HBLANK	**HBlank** interrupt. Also requires `REG_DISPSTAT` {4} Occurs *after* the HDraw, so that things done here take effect in the next line.
2	Vct	IRQ_VCOUNT	**VCount** interrupt. Also requires `REG_DISPSTAT` {5}. The high byte of

			`REG_DISPSTAT` gives the VCount at which to raise the interrupt. Occurs at the *beginning* of a scanline.
3-6	Tm	IRQ_TIMER*x*	**Timer** interrupt, 1 bit per timer. Also requires `REG_TMxCNT` {6}. The interrupt will be raised when the timer overflows.
7	Com	IRQ_COM	**Serial communication** interrupt. Apparently, also requires `REG_SCCNT` {E}. To be raised when the transfer is complete. Or so I'm told, I really don't know squat about serial communication.
8-B	Dma	IRQ_DMA*x*	**DMA** interrupt, 1 bit per channel. Also requires `REG_DMAxCNT` {1E}. Interrupt will be raised when the full transfer is complete.
C	K	IRQ_KEYPAD	**Keypad** interrupt. Also requires `REG_KEYCNT` {E}. Raised when any or all or the keys specified in `REG_KEYCNT` are down.
D	C	IRQ_GAMEPAK	**Cartridge** interrupt. Raised when the cart is removed from the GBA.

Interrupt Service Routines

You use the interrupt registers described above to indicate which interrupts you want to use. The next step is writing an interrupt service routine. This is just a typeless function (`void func(void)`); a C-function like many others. Here's an example of an HBlank interrupt.

```
void hbl_pal_invert()
{
    pal_bg_mem[0] ^= 0x7FFF;
    REG_IF = IRQ_HBLANK;
}
```

The first line inverts the color of the first entry of the palette memory. The second line resets the HBlank bit of `REG_IF` indicating the interrupt has been dealt with. Since this is an HBlank interrupt, the end-result is that that the color changes every scanline. This shouldn't be too hard to imagine.

If you simply add this function to an existing program, nothing would change. How come? Well, though you have an isr now, you still need to tell the GBA where to find it. For that, we will need to take a closer look at the interrupt process as a whole.

> **ON ACKNOWLEDGING INTERRUPTS CORRECTLY**
>
> To acknowledge that an interrupt has been dealt with, you have to **set** the bit of that interrupt in `REG_IF`, and *only* that bit. That means that '`REG_IF = IRQ_x`' is usually the correct course of action, and not '`REG_IF |= IRQ_x`'. The |= version acknowledges all interrupts that have been raised, even if you haven't dealt with them yet.
>
> Usually, these two result in the same thing, but if multiple interrupts come in at the same time things will go bad. Just pay attention to what you're doing.

The interrupt process

The complete interrupt process is kind of tricky and part of it is completely beyond your control. What follows now is a list of things that you, the programmer, need to know. For the full story, see GBATEK : irq control.

1. Interrupt occurs. Some black magic deep within the deepest dungeons of BIOS happens and the CPU is switched to IRQ mode and ARM state. A number of

registers (`r0-r3, r12, lr`) are pushed onto the stack.

2. BIOS loads the address located at `0300:7FFC` and branches to that address.

3. The code pointed to by `0300:7FFC` is run. Since we're in ARM-state now, this *must* to be ARM code!

4. After the isr is done, acknowledge that the interrupt has been dealt with by writing to `REG_IF` , then return from the isr by issuing a `bx lr` instruction.

5. The previously saved registers are popped from stack and program state is restored to normal.

Steps 1, 2 and 5 are done by BIOS; 3 and 4 are yours. Now, in principle all you need to do is place the address of your isr into address `0300:7FFC` . To make our job a little easier, we will first create ourselves a function pointer type.

```
typedef void (*fnptr)(void);
#define REG_ISR_MAIN *(fnptr*)(0x03007FFC)

// Be careful when using it like this, see notes below
void foo()
{
    REG_ISR_MAIN= hbl_pal_invert;  // tell the GBA where my isr is
    REG_DISPSTAT |= VID_HBL_IRQ;   // Tell the display to fire HBlank
interrupts
    REG_IE |= IRQ_HBLANK;          // Tell the GBA to catch HBlank
interrupts
    REG_IME= 1;                    // Tell the GBA to enable interrupts;
}
```

Now, this will probably work, but as usual there's more to the story.

- First, the code that `REG_ISR_MAIN` jumps to *must* be ARM code! If you compile with the `-mthumb` flag, the whole thing comes to a screeching halt.

- What happens when you're interrupted inside an interrupt? Well, that's not quite possible actually; not unless you do some fancy stuff we'll get to later. You see, `REG_IME` is not the only thing that allows interrupts, there's a bit for irqs in the *program status register* (PSR) as well. When an interrupt is raised, the CPU disables interrupts there until the whole thing is over and done with.

- `hbl_pal_invert()` doesn't check whether it has been activated by an HBlank interrupt. Now, in this case it doesn't really matter because it's the only one enabled, but when you use different types of interrupts, sorting them out is essential. That's why we'll create an interrupt switchboard in the next section.
- Lastly, when you use BIOS calls that require interrupts, you also need to acknowledge them in `REG_IFBIOS` (== `0300:7FF8`). The use is the same as `REG_IF`.

> **ON SECTION MIRRORING**
>
> GBA's memory sections are mirrored ever so many bytes. For example IWRAM (`0300:0000`) is mirrored every 8000h bytes, so that `0300:7FFC` is also `03FF:FFFC`, or `0400:0000` −4. While this is faster, I'm not quite sure if this should be taken advantage of. no$gba v2.2b marks it as an error, even though this was apparently a small oversight and fixed in v2.2c. Nevertheless, consider yourself warned.

Creating an interrupt switchboard

The `hbl_pal_invert()` function is an example of a single interrupt, but you may have to deal with multiple interrupts. You may also want to be able to use different isr's depending on circumstances, in which case stuffing it all into one function may not be the best way to go. Instead, we'll create an interrupt switchboard.

An *interrupt switchboard* works a little like a telephone switchboard: you have a call (i.e., an interrupt, in `REG_IF`) coming in, the operator checks if it is an active number (compares it with `REG_IE`) and if so, connects the call to the right receiver (your isr).

This particular switchboard will come with a number of additional features as well. It will acknowledge the call in both `REG_IF` and `REG_IFBIOS`), even when there's no

actual ISR attached to that interrupt. It will also allow nested interrupts, although this requires a little extra work in the ISR itself.

Design and interface considerations

The actual switchboard is only one part of the whole; I also need a couple of structs, variables and functions. The basic items I require are these.

- `__isr_table[]`. An interrupt table. This is a table of function pointers to the different isr's. Because the interrupts should be prioritized, the table should also indicate which interrupt the pointers belong to. For this, we'll use an `IRQ_REC` struct.
- `irq_init()` / `irq_set_master()`. Set master isr. `irq_init()` initializes the interrupt table and interrupts themselves as well.
- `irq_enable()` / `irq_disable()`. Functions to enable and disable interrupts. These will take care of both `REG_IE` and whatever register the sender bit is on. I'm keeping these bits in an internal table called `__irq_senders[]` and to be able to use these, the input parameter of these functions need to be the *index* of the interrupt, not the interrupt flag itself. Which is why I have `II_` *foo* counterparts for the `IRQ_` *foo* flags.
- `irq_set()` / `irq_add()` / `irq_delete()`. Function to add/delete interrupt service routines. The first allows full prioritization of isr's; `irq_add()` will replace the current irs for a given interrupt, or add one at the end of the list; `irq_delete()` will delete one and correct the list for the empty space.

All of these functions do something like this: disable interrupts (`REG_IME` =0), do their stuff and then re-enable interrupts. It's a good idea to do this because being interrupted while mucking about with interrupts is not pretty. The functions concerned with service routines will also take a function pointer (the `fnptr` type), and also return a function pointer indicating the previous isr. This may be useful if you want to try to chain them.

Below you can see the structs, tables, and the implementation of `irq_enable()` and `irq_add()`. In both functions, the `__irq_senders[]` array is used to determine which bit to set in which register to make sure things send interrupt requests. The `irq_add()` function goes on to finding either the requested interrupt in the current table to replace, or an empty slot to fill. The other routines are similar. If you need to see more, look in *tonc_irq.h/.c* in libtonc.

```c
//! Interrups Indices
typedef enum eIrqIndex
{
    II_VBLANK=0, II_HBLANK, II_VCOUNT, II_TIMER0,
    II_TIMER1,   II_TIMER2, II_TIMER3, II_SERIAL,
    II_DMA0,     II_DMA1,   II_DMA2,   II_DMA3,
    II_KEYPAD,   II_GAMEPAK,II_MAX
} eIrqIndex;

//! Struct for prioritized irq table
typedef struct IRQ_REC
{
    u32 flag;   //!< Flag for interrupt in REG_IF, etc
    fnptr isr;  //!< Pointer to interrupt routine
} IRQ_REC;

// === PROTOTYPES =========================================================

IWRAM_CODE void isr_master_nest();

void irq_init(fnptr isr);
fnptr irq_set_master(fnptr isr);

fnptr irq_add(enum eIrqIndex irq_id, fnptr isr);
fnptr irq_delete(enum eIrqIndex irq_id);

fnptr irq_set(enum eIrqIndex irq_id, fnptr isr, int prio);
void irq_enable(enum eIrqIndex irq_id);
void irq_disable(enum eIrqIndex irq_id);
```

```c
// IRQ Sender information
typedef struct IRQ_SENDER
{
    u16 reg_ofs;      //!< sender reg - REG_BASE
    u16 flag;         //!< irq-bit in sender reg
} ALIGN4 IRQ_SENDER;

// === GLOBALS ==========================================================

// One extra entry for guaranteed zero
IRQ_REC __isr_table[II_MAX+1];

static const IRQ_SENDER __irq_senders[] =
{
    { 0x0004, 0x0008 },      // REG_DISPSTAT,    DSTAT_VBL_IRQ
    { 0x0004, 0x0010 },      // REG_DISPSTAT,    DSTAT_VHB_IRQ
    { 0x0004, 0x0020 },      // REG_DISPSTAT,    DSTAT_VCT_IRQ
    { 0x0102, 0x0040 },      // REG_TM0CNT,      TM_IRQ
    { 0x0106, 0x0040 },      // REG_TM1CNT,      TM_IRQ
    { 0x010A, 0x0040 },      // REG_TM2CNT,      TM_IRQ
    { 0x010E, 0x0040 },      // REG_TM3CNT,      TM_IRQ
    { 0x0128, 0x4000 },      // REG_SCCNT_L      BIT(14) // not sure
    { 0x00BA, 0x4000 },      // REG_DMA0CNT_H,   DMA_IRQ>>16
    { 0x00C6, 0x4000 },      // REG_DMA1CNT_H,   DMA_IRQ>>16
    { 0x00D2, 0x4000 },      // REG_DMA2CNT_H,   DMA_IRQ>>16
    { 0x00DE, 0x4000 },      // REG_DMA3CNT_H,   DMA_IRQ>>16
    { 0x0132, 0x4000 },      // REG_KEYCNT,      KCNT_IRQ
    { 0x0000, 0x0000 },      // cart: none
};


// === FUNCTIONS ========================================================

//! Enable irq bits in REG_IE and sender bits elsewhere
void irq_enable(enum eIrqIndex irq_id)
{
    u16 ime= REG_IME;
    REG_IME= 0;

    const IRQ_SENDER *sender= &__irq_senders[irq_id];
    *(u16*)(REG_BASE+sender->reg_ofs) |= sender->flag;

    REG_IE |= BIT(irq_id);
    REG_IME= ime;
}

//! Add a specific isr
fnptr irq_add(enum eIrqIndex irq_id, fnptr isr)
{
    u16 ime= REG_IME;
```

```
    REG_IME= 0;

    int ii;
    u16 irq_flag= BIT(irq_id);
    fnptr old_isr;
    IRQ_REC *pir= __isr_table;

    // Enable irq
    const IRQ_SENDER *sender= &__irq_senders[irq_id];
    *(u16*)(REG_BASE+sender->reg_ofs) |= sender->flag;
    REG_IE |= irq_flag;

    // Search for previous occurance, or empty slot
    for(ii=0; pir[ii].flag; ii++)
        if(pir[ii].flag == irq_flag)
            break;

    old_isr= pir[ii].isr;
    pir[ii].isr= isr;
    pir[ii].flag= irq_flag;

    REG_IME= ime;
    return old_isr;
}
```

The master interrupt service routine

The main task of the master ISR is to seek out the raised interrupt in `___isr_table`, and acknowledge it in both `REG_IF` and `REG_IFBIOS`. If there is an irq-specific service routine, it should call it; otherwise, it should just exit to BIOS again. In C, it would look something like this.

```
// This is mostly what libtonc's isr_master does, but
// you really need asm for the full functionality
IWRAM_CODE void isr_master_c()
{
    u32 ie= REG_IE;
    u32 ieif= ie & REG_IF;
    IRQ_REC *pir;

    // (1) Acknowledge IRQ for hardware and BIOS.
    REG_IF      = ieif;
    REG_IFBIOS |= ieif;

    // (2) Find raised irq
    for(pir= __isr_table; pir->flag!=0; pir++)
        if(pir->flag & ieif)
            break;

    // (3) Just return if irq not found in list or has no isr.
    if(pir->flag == 0 || pir->isr == NULL)
        return;

    // --- If we're here have an interrupt routine ---
    // (4a) Disable IME and clear the current IRQ in IE
    u32 ime= REG_IME;
    REG_IME= 0;
    REG_IE &= ~ieif;

    // (5a) CPU back to system mode
    //> *(--sp_irq)= lr_irq;
    //> *(--sp_irq)= spsr
    //> cpsr &= ~(CPU_MODE_MASK | CPU_IRQ_OFF);
    //> cpsr |= CPU_MODE_SYS;
    //> *(--sp_sys) = lr_sys;

    pir->isr();             // (6) Run the ISR

    REG_IME= 0;             // Clear IME again (safety)

    // (5b) Back to irq mode
    //> lr_sys = *sp_sys++;
    //> cpsr &= ~(CPU_MODE_MASK | CPU_IRQ_OFF);
    //> cpsr |= CPU_MODE_IRQ | CPU_IRQ_OFF;
    //> spsr = *sp_irq++
    //> lr_irq = *sp_irq++;

    // (4b) Restore original ie and ime
    REG_IE= ie;
    REG_IME= ime;
}
```

Most of these points have been discussed already, so I won't repeat them again. Do note the difference is acknowledging `REG_IF` and `REG_IFBIOS` : the former uses a simple assignment and the latter an |=. Steps 4, 5 and 6 only execute if the current IRQ has its own service routine. Steps 4a and 5a work as initialization steps to ensure that the ISR (step 6) can work in CPU mode and that it can't be interrupted unless it asks for it. Steps 4b and 5b unwind 4a and 5a.

This routine would work fine in C, were it not for items 5a and 5b. These are the code to set/restore the CPU mode to system/irq mode, but the instructions necesasry for that aren't available in C. Another problem is that the link registers (these are used to hold the return addresses of functions) have to be saved somehow, and these *definitely* aren't available in C.

Note: I said register**s**, plural! Each CPU mode has its own stack and link register, and even though the names are the same (`lr` and `sp`), they really aren't identical. Usually a C routine will save `lr` on its own, but since you need it twice now it's very unsafe to leave this up to the compiler. Aside from that, you need to save the saved program status register `spsr` , which indicates the program status when the interrupt occurred. This is another thing that C can't really do. As such, assembly is required for the master ISR.

So, assembly it is then. The function below is the assembly equivalent of `irs_master_c()` . It is almost a line by line translation, although I am making use of a few features of the instruction set the compiler wont't or can't. I don't expect you to really understand everything written here, but with some imagination you should be able to follow most of it. Teaching assembly is *way* beyond the scope of this chapter, but worth the effort in my view. Tonc's assembly chapter should give you the necessary information to understand most of it and shows where to go to learn more.

```
        .file   "tonc_isr_master.s"
        .extern __isr_table;

/*! \fn IWRAM_CODE void isr_master()
        \brief  Default irq dispatcher (no automatic nesting)
*/
        .section .iwram, "ax", %progbits
        .arm
        .align
        .global isr_master

        @ --- Register list ---
        @ r0 : &REG_IE
        @ r1 : __isr_table / isr
        @ r2 : IF & IE
        @ r3 : tmp
        @ ip : (IF<<16 | IE)

isr_master:
        @ Read IF/IE
        mov     r0, #0x04000000
        ldr     ip, [r0, #0x200]!
        and     r2, ip, ip, lsr #16     @ irq= IE & IF

        @ (1) Acknowledge irq in IF and for BIOS
        strh    r2, [r0, #2]
        ldr     r3, [r0, #-0x208]
        orr     r3, r3, r2
        str     r3, [r0, #-0x208]

        @ (2) Search for irq.
        ldr     r1, =__isr_table
.Lirq_search:
        ldr     r3, [r1], #8
        tst     r3, r2
        bne     .Lpost_search       @ Found one, break off search
        cmp     r3, #0
        bne     .Lirq_search        @ Not here; try next irq

        @ (3) Search over : return if no isr, otherwise continue.
.Lpost_search:
        ldrne   r1, [r1, #-4]           @ isr= __isr_table[ii-1].isr
        cmpne   r1, #0
        bxeq    lr                      @ If no isr: quit

        @ --- If we're here, we have an isr ---

        @ (4a) Disable IME and clear the current IRQ in IE
        ldr     r3, [r0, #8]            @ Read IME
        strb    r0, [r0, #8]           @ Clear IME
```

```
    bic     r2, ip, r2
    strh    r2, [r0]                @ Clear current irq in IE

    mrs     r2, spsr
    stmfd   sp!, {r2-r3, ip, lr}    @ sprs, IME, (IE,IF), lr_irq

    @ (5a) Set mode to sys
    mrs     r3, cpsr
    bic     r3, r3, #0xDF
    orr     r3, r3, #0x1F
    msr     cpsr, r3

    @ (6) Call isr
    stmfd   sp!, {r0,lr}            @ &REG_IE, lr_sys
    mov     lr, pc
    bx      r1
    ldmfd   sp!, {r0,lr}            @ &REG_IE, lr_sys

    @ --- Unwind ---
    strb    r0, [r0, #8]            @ Clear IME again (safety)
    @ (5b) Reset mode to irq
    mrs     r3, cpsr
    bic     r3, r3, #0xDF
    orr     r3, r3, #0x92
    msr     cpsr, r3

    @ (4b) Restore original spsr, IME, IE, lr_irq
    ldmfd   sp!, {r2-r3, ip, lr}    @ sprs, IME, (IE,IF), lr_irq
    msr     spsr, r2
    strh    ip, [r0]
    str     r3, [r0, #8]

    bx      lr
```

NESTED IRQS ARE NASTY

Making a nested interrupt routine work is not a pleasant exercise when you only partially know what you're doing. For example, that different CPU modes used different stacks took me a while to figure out, and it took me quite a while to realize that the reason my nested isrs didn't work was because there are different link registers too.

The `isr_master_nest` is largely based on libgba's interrupt dispatcher, but also borrows information from GBATEK and A. Bilyk and DekuTree's analysis of the

whole thing as described in forum:4063. Also invaluable was the home-use debugger version of no$gba, hurray for breakpoints.

If you want to develop your own interrupt routine, these sources will help you immensely and will keep the loss of sanity down to somewhat acceptable levels.

DEPRECATION NOTICE

I used to have a different master service routine that took care of nesting and prioritizing interrupts automatically. Because it was deemed too complicated, it has been replaced with this one.

Nested interrupts are still possible, but you have to indicate interruptability inside the isr yourself now.

Nested interrupt demo

Today's demo shows a little bit of everything described above:

- It'll display a color gradient on the screen through the use of an HBlank interrupt.
- It will allow you to toggle between two different master isrs: The switchboard `isr_master` which routes the program flow to an HBlank isr, and an isr in C that handles the HBlank interrupt directly. For the latter to work, we'll need to use ARM-compiled code, of course, and I'll also show you how in a minute.
- Finally, having a nested isr switchboard doesn't mean much unless you can actually see nested interrupts in action. In this case, we'll use two interrupts: VCount and HBlank. The HBlank isr creates a vertical color gradient. The VCount isr will reset the color and tie up the CPU for several scanlines. If interrupts don't

nest, you'll see the gradient stop for a while; if they do nest, it'll continue as normal.

- And just for the hell of it, you can toggle the HBlank and VCount irqs on and off.

The controls are as follows:

A	Toggles between asm switchboard and C direct isr.
B	Toggles HBlank and VCount priorities.
L,R	Toggles VCount and HBlank irqs on and off.

```c
#include <stdio.h>
#include <tonc.h>

IWRAM_CODE void isr_master();
IWRAM_CODE void hbl_grad_direct();

void vct_wait();
void vct_wait_nest();


CSTR strings[]=
{
    "asm/nested",    "c/direct",
    "HBlank",        "VCount"
};

// Function pointers to master isrs.
const fnptr master_isrs[2]=
{
    (fnptr)isr_master,
    (fnptr)hbl_grad_direct
};

// VCount interrupt routines.
const fnptr vct_isrs[2]=
{
    vct_wait,
    vct_wait_nest
};


// (1) Uses tonc_isr_master.s' isr_master() as a switchboard
void hbl_grad_routed()
{
    u32 clr= REG_VCOUNT/8;
    pal_bg_mem[0]= RGB15(clr, 0, 31-clr);
}

// (2a) VCT is triggered at line 80; this waits 40 scanlines
void vct_wait()
{
    pal_bg_mem[0]= CLR_RED;
    while(REG_VCOUNT<120);
}

// (2b) As vct_wait(), but interruptable by HBlank
void vct_wait_nest()
{
    pal_bg_mem[0]= CLR_RED;
    REG_IE= IRQ_HBLANK;      // Allow nested hblanks
```

```c
    REG_IME= 1;
    while(REG_VCOUNT<120);
}


int main()
{
    u32 bDirect=0, bVctPrio= 0;

    tte_init_chr4_b4_default(0, BG_CBB(2)|BG_SBB(28));
    tte_set_drawg((fnDrawg)chr4_drawg_b4cts_fast);
    tte_init_con();
    tte_set_margins(8, 8, 128, 64);

    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    // (3) Initialize irqs; add HBL and VCT isrs
    // and set VCT to trigger at 80
    irq_init(master_isrs[0]);
    irq_add(II_HBLANK, hbl_grad_routed);
    BFN_SET(REG_DISPSTAT, 80, DSTAT_VCT);
    irq_add(II_VCOUNT, vct_wait);
    irq_add(II_VBLANK, NULL);

    while(1)
    {
        //vid_vsync();
        VBlankIntrWait();
        key_poll();

        // Toggle HBlank irq
        if(key_hit(KEY_R))
            REG_IE ^= IRQ_HBLANK;

        // Toggle Vcount irq
        if(key_hit(KEY_L))
            REG_IE ^= IRQ_VCOUNT;

        // (4) Toggle between
        // asm switchblock + hbl_gradient (red, descending)
        // or purely hbl_isr_in_c (green, ascending)
        if(key_hit(KEY_A))
        {
            bDirect ^= 1;
            irq_set_master(master_isrs[bDirect]);
        }

        // (5) Switch priorities of HBlank and VCount
        if(key_hit(KEY_B))
        {
```

```
        //irq_set(II_VCOUNT, vct_wait, bVctPrio);
        bVctPrio ^= 1;
        irq_add(II_VCOUNT, vct_isrs[bVctPrio]);
    }

    tte_printf("#{es;P}IRS#{X:32}: %s\nPrio#{X:32}: %s\nIE#{X:32}:
%04X",
        strings[bDirect], strings[2+bVctPrio], REG_IE);
    }

    return 0;
}
```

The code listing above contains the main demo code, the HBlank, and VCount isrs that will be routed and some sundry items for convenience. The C master isr called `hbl_grad_direct()` is in another file, which will be discussed later.

First, the contents of the interrupt service routines (points 1 and 2). Both routines are pretty simple: the HBlank routine (`hbl_grad_routed()`) uses the value of the scanline counter to set a color for the backdrop. At the top, `REG_VCOUNT` is 0, so the color will be blue; at the bottom, it'll be 160/8=20, so it's somewhere between blue and red: purple. Now, you may notice that the first scanline is actually red and not blue: this is because a) the HBlank interrupt occurs *after* the scanline (which has caused trouble before in the DMA demo) and b) because HBlanks happen during the VBlank as well, so that the color for line 0 is set at `REG_VCOUNT` =227, which will give a bright red color.
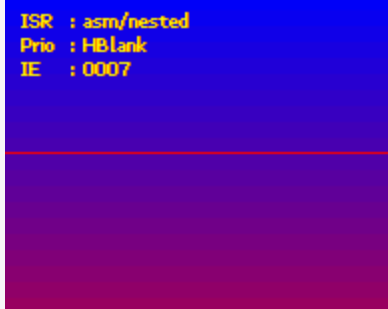
The VCount routines activate at scanline 80. They set the color to red and then waits until scanline 120. The difference between the two is that `vct_wait()` just waits, but `vct_wait_nest()` enables the HBlank interrupt. Remember that `isr_master` disables interrupts before calling an service routine, so the latter Vcount routine should be interrupted by `hbl_grad_routed()` , but the former would not. As you can see from fig 16.1a and fig 16.1b, this is exactly what happens.

Point 3 is where the interrupts are set up in the first place. The call to `irq_init()` clears the isr table and sets up the master isr. Its argument can be NULL, in which case the tonc's default master isr is used. The calls to `irq_add()` initialize the HBlank and VCount interrupts and their service routines. If you don't supply a service routine, the
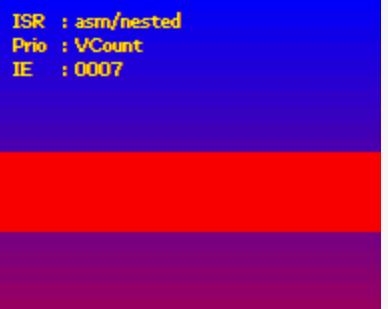
switchboard will just acknowledge the interrupt and return. There are times when this is useful, as we'll see in the next chapter. `irq_add()` already takes care of both `REG_IE` and the IRQ bits in `REG_DISPSTAT`; what it doesn't do yet is set the VCount at which the interrupt should be triggered, so this is done separately. The order of `irq_add()` doesn't really matter, but lower orders are searched first so it makes sense to put more frequent interrupts first.

You can switch between master service routines with `irq_set_master()`, as is done at point 4. Point 5 chooses between the nested and non-nested VCount routine.

Fig 16.1a: Gradient; nested `vct_wait_nested`.

Fig 16.1b: Gradient; non-nested `vct_wait`.

Fig 16.1c: Gradient; HBla. master ISR in C.

This explains most of what the demo can show. For Real Life use, `irq_init()` and `irq_add()` are pretty much all you need, but the demo shows some other interesting things as well. Also interesting is that the result is actually a little different for VBA, no$gba and hardware, which brings up another point: interrupts are time-critical routines, and emulating timing is rather tricky. If something works on an emulator but not hardware, interrupts are a good place to start looking.

This almost concludes demo section, except for one thing: the direct HBlank isr in C. But to do that, we need it in ARM code and to make it efficient, it should be in IWRAM as well. And here's how we do that.

Using ARM + IWRAM code

The master interrupt routines have to be ARM code. As we've always compiled to Thumb code, this would be something new. The reason that we've always compiled to Thumb code is that the 16bit buses of the normal code sections make ARM-code slow there. However, what we could do is put the ARM code in IWRAM, which has a 32bit bus (and no waitstates) so that it's actually beneficial to use ARM code there.

Compiling as ARM code is actually quite simple: use `-marm` instead of `-mthumb`. The IWRAM part is what causes the most problems. There are GCC extensions that let you specify which section a function should be in. Tonclib has the following macros for them:

```c
#define EWRAM_DATA __attribute__((section(".ewram")))
#define IWRAM_DATA __attribute__((section(".iwram")))
#define  EWRAM_BSS __attribute__((section(".sbss")))

#define EWRAM_CODE __attribute__((section(".ewram"), long_call))
#define IWRAM_CODE __attribute__((section(".iwram"), long_call))


// --- Examples of use: ---
// Declarations
extern EWRAM_DATA u8 data[];
IWRAM_CODE void foo();


// Definitions
EWRAM_DATA u8 data[8]= { ... };

IWRAM_CODE void foo()
{
    ....
}
```

The EWRAM/IWRAM things should be self-explanatory. The `DATA_IN_x` things allow global data to be put in those sections. Note that the default section for data is IWRAM anyway, so that may be a little redundant. `EWRAM_BSS` concerns uninitialized globals. The difference with initialized globals is that they don't have to take up space in ROM: all you need to know is how much space you need to reserve in RAM for the array.

The function variants also need the `long_call` attribute. Code branches have a limited range and section branches are usually too far to happen by normal means and this is what makes it work. You can compare them with 'far' and 'near' that used to be present in PC programming.

It should be noted that these extensions can be somewhat fickle. For one thing, the placement of the attributes in the declarations and definitions seems to matter. I think the examples given work, but if they don't try to move them around a bit and see if that helps. A bigger problem is that the long_call attribute doesn't always want to work. Previous experience has led me to believe that the `long_call` is ignored *unless* the definition of the function is in another file. If it's in the same file as the calling function, you'll get a 'relocation error', which basically means that the jump is too far. The upshot of this is that you have to separate your code depending on section as far as functions are concerned. Which works out nicely, as you'll want to separate ARM code anyway.

So, for ARM/IWRAM code, you need to have a separate file with the routines, use the `IWRAM_CODE` macro to indicate the section, and use `-marm` in compilation. It is also a good idea to add `-mlong-calls` too, in case you ever want to call ROM functions from IWRAM. This option makes every call a long call. Some toolchains (including DKP) have set up their linkscripts so that files with the extension *.iwram.c* automatically go into IWRAM, so that `IWRAM_CODE` is only needed for the declaration.

In this case, that'd be the file called *isr.iwram.c*. This contains a simple master isr in C, and only takes care of the HBlank and acknowledging the interrupts.

```c
#include <tonc.h>

IWRAM_CODE void hbl_grad_direct();

// an interrupt routine purely in C
// (make SURE you compile in ARM mode!!)
void hbl_grad_direct()
{
    u32 irqs= REG_IF & REG_IE;

    REG_IFBIOS |= irqs;
    if(irqs & IRQ_HBLANK)
    {
        u32 clr= REG_VCOUNT/8;
        pal_bg_mem[0]= RGB15(0, clr, 0);
    }

    REG_IF= irqs;
}
```

FLAGS FOR ARM+IWRAM COMPILATION

Replace the '-mthumb' in your compilation flags by '-marm -mlong-calls'. For example:

```make
CBASE   := $(INCDIR) -O2 -Wall

# ROM flags
RCFLAGS := $(CBASE) -mthumb-interwork -mthumb
# IWRAM flags
ICFLAGS := $(CBASE) -mthumb-interwork -marm -mlong-calls
```

For more details, look at the makefile for this project.

17. BIOS Calls

Introduction

Apart from hardware interrupts, like HBlank and cartridge interrupts, there are also things called *software interrupts*, also known as *BIOS calls*. The software interrupts work very much like ordinary functions: you set-up input, call the routine, and get some output back. The difference lies in how you reach the code; for normal functions you just, well, jump to the routine you want. Software interrupts use the `swi` instruction, which diverts the program flow to somewhere in BIOS, carries out the requested algorithm and then restores the normal flow of your program. This is similar to what hardware interrupts do, only now you raise the interrupt programmatically. Hence: software interrupt.

The GBA BIOS has 42 software interrupts, with basic routines for copying, math (division, square root), affine transformations for sprites and backgrounds, decompression among others. There are also some very special functions like the `IntrWait` routines, which can stop the CPU until a hardware interrupt occurs. The VBlank variant is highly recommended, which is what makes this chapter important.

Using software interrupts isn't too hard if it weren't for one thing: the `swi` instruction itself. This again requires some assembly. However, not *much* assembly, and it's easy to write C wrappers for them, which we'll also cover here.

The BIOS functions

Calling the BIOS functions can be done via the '`swi n`' instruction, where *n* is the BIOS call you want to use. Mind you, the exact numbers you need to use depends on whether your code is in ARM or Thumb state. In Thumb the argument is simply the *n* itself, but in ARM you need to use *n*<<16. Just like normal functions, the BIOS calls can have input and output. The first four registers (r0-r3) are used for this purpose; though the exact purpose and the number of registers differ for each call.

Here's a list containing the names of each BIOS call. I am not going to say what each of them does since other sites have done that already and it seems pointless to copy their stuff verbatim. For full descriptions go to GBATEK, for example. I will give a description of a few of them so you can get a taste of how they work.

Full list

id	Name		id	Name
0x00	SoftReset		0x08	Sqrt
0x01	RegisterRamReset		0x09	ArcTan
0x02	Halt		0x0A	ArcTan2
0x03	Stop		0x0B	CPUSet
0x04	IntrWait		0x0C	CPUFastSet
0x05	VBlankIntrWait		0x0D	BiosChecksum
0x06	Div		0x0E	BgAffineSet
0x07	DivArm		0x0F	ObjAffineSet

0x10	BitUnPack		0x18	Diff16bitUnFilter
0x11	LZ77UnCompWRAM		0x19	SoundBiasChange
0x12	LZ77UnCompVRAM		0x1A	SoundDriverInit
0x13	HuffUnComp		0x1B	SoundDriverMode
0x14	RLUnCompWRAM		0x1C	SoundDriverMain
0x15	RLUnCompVRAM		0x1D	SoundDriverVSync
0x16	Diff8bitUnFilterWRAM		0x1E	SoundChannelClear
0x17	Diff8bitUnFilterVRAM		0x1F	MIDIKey2Freq
0x20	MusicPlayerOpen		0x28	SoundDriverVSyncOff
0x21	MusicPlayerStart		0x29	SoundDriverVSyncOn
0x22	MusicPlayerStop		0x2A	GetJumpList
0x23	MusicPlayerContinue			
0x24	MusicPlayerFadeOut			
0x25	MultiBoot			
0x26	HardReset			
0x27	CustomHalt			

Div, Sqrt, Arctan2 and ObjAffineSet descriptions

0x06: Div

Input:

r0: numerator

r1: denominator

Output:

r0: numerator / denominator

r1: numerator % denominator

r3: abs(numerator / denominator)

Note: do NOT divide by zero!

0x08: Sqrt

Input:

r0: num, a unsigned 32-bit integer

Output:

r1: sqrt(num)

0x0a: ArcTan2

Input:

r0: x, a **signed 16bit** number (`s16`)

r1: y, a **signed 16bit** number (`s16`)

Output:

r0: $x{\geq}0 : \vartheta= \arctan(y/x) \lor x{<}0 : \vartheta= \mathrm{sign}(y)^{*}(\pi - \arctan(|y/x|)$.

This does the full inverse of $y = x^{*}\tan(\theta)$. The problem with the tangent is that the domain is a semi-circle, as is the range of arc tangent. To get the full circle range, both x and y values are required for not only their quotient, but their signs as well. The mathematical range of ϑ is $[-\pi, \pi\rangle$, which corresponds to $[-0x8000, 0x8000\rangle$ (or $[0, 2\pi\rangle$ and $[0, 0xFFFF]$ if you like)

0x0f: ObjAffineSet

Input:

r0: source address

r1: destination address

r2: number of calculations

r3: Offset of **P** matrix elements (2 for bgs, 8 for objects)

The source address points to an array of `AFF_SRC` structs (also known as `ObjAffineSource`, which is a bit misleading since you can use them for backgrounds as well). The `AFF_SRC` struct consist of two scales s_x, s_y and an angle α, which again uses the range [0, 0xFFFF] for 2π. The resulting **P**:

$$(17.1) \quad \mathbf{P} = \begin{bmatrix} s_x \cdot \cos\alpha & -s_x \cdot \sin\alpha \\ s_y \cdot \sin\alpha & s_y \cdot \cos\alpha \end{bmatrix}$$

By now you should know what this does: it scales horizontally by $1/s_x$, vertically by $1/s_y$ followed by a counter-clockwise rotation by α. `ObjAffineSet()` does almost exactly what `obj_aff_rotscale()` and `bg_aff_rotscale()` do, except that `ObjAffineSet()` can also set multiple matrices at once.

The source data is kept in `ObjAffineSource` (i.e., `AFF_SRC`) structs. Now, as the routine sets affine matrices, you might think that the destinations are either `OBJ_AFFINE` or `ObjAffineDest` structs. However, you'd be wrong. Well, partially anyway. The problem is that the destination always points to a p_a-element, which is not necessarily the first element in struct. You *will* make the mistake of simply supplying an `OBJ_AFFINE` pointer when you try to use it to fill those. Don't say I didn't warn you.

Two other things need to be said here as well. First, once again we have a bit of a misnomer: ObjAffineSet doesn't really have much to do with objects per se, but can be used in that capacity by setting `r3` to 8 instead of 2. The second is that the routine can also be used to set up multiple arrays via `r2`. However, *be careful* when you do this with devkitPro 19. `ObjAffineSet()` expects its source structs to be word-aligned, which they won't be unless you add the alignment attributes yourself.

```
// Source struct. Note the alignment!
typedef struct AFF_SRC
{
    s16 sx, sy;
    u16 alpha;
} ALIGN4 AFF_SRC, ObjAffineSource;

// Dst struct for background matrices
typedef struct Aff_DST
{
    s16 pa, pb;
    s16 pc, pd;
} ALIGN4 AFF_DST, ObjAffineDest;

// Dst struct for objects. Note that r1 should be
// the address of pa, not the start of the struct
typedef struct OBJ_AFFINE
{
    u16 fill0[3];    s16 pa;
    u16 fill1[3];    s16 pb;
    u16 fill2[3];    s16 pc;
    u16 fill3[3];    s16 pd;
} ALIGN4 OBJ_AFFINE;
```

Using BIOS calls

Assembly for BIOS calls

You might think this whole discussion was rather pointless since you can't access the registers and the `swi` instruction unless you use assembly, which will be pretty tough, right? Well, no, yes and no. The necessary assembly steps for BIOS calls are actually rather simple, and are given below.

```
@ In tonc_bios.s

@ at top of your file
    .text           @ aka .section .text
    .code 16        @ aka .thumb

@ for each swi (like division, for example)
    .align 2        @ aka .balign 4
    .global Div
    .thumb_func
Div:
    swi     0x06
    bx      lr
```

This is assembly code for the GNU assembler (GAS); for Goldroad or ARM STD the syntax is likely to be slightly different. The first thing you need to do is give some *directives*, which tells some details about the following code. In this case, we use the ' `.text` ' to put the code in the `text` section (ROM or EWRAM for multiboot). We also say that the code is Thumb code by using ' `.code 16` ' or ' `.thumb` '. If you place these at the top of the file, they'll hold for the rest of the thing. For each BIOS call, you'll need the following 6 items.

- **Word-alignment**. Or at least halfword alignment, but words are probably preferable. There are two directives for this, `.align` *n* and `.balign` *m* . The former aligns to 2^n so requires ' `.align 2` '; the latter aligns to *m* so you can just use ' `balign` *m* '. Note that both will only work on the *next* piece of code or data and no further, which is why it's best to add it for each function.
- **Scope**. The `.global` *name* directive makes a symbol out of *name*, which will then be visible for other files in the project as well. A bit like `extern` or, rather, an anti- `static` .
- **Thumb indicator** It would seem that `.code 16` alone isn't enough, you also need `.thumb_func` . In fact, if I read the manual correctly this one also implies `.code 16` , which would make that directive redundant.
- **Label**. '*name:*' marks where the symbol *name* starts. Obviously, to use a function it must actually exist.

- **BIOS call** To actually activate the BIOS call, use 'swi *n*', with *n* the BIOS call you want.
- **Return** And we're practically done already, all we have to do now is return to the caller with 'bx lr'.

See? It's really not that complicated. Sometimes you might want a little more functionality than this, but for the most part you only need two measly instructions.

The Arm Architecture Procedure Call Standard

That's all fine and good, but that still leaves the questions of a) how do I combine this with C code and b) where'd all the input and output go? The answer to the first is simple: just add a function declaration like usual:

```
// In tonc_bios.h

int Div(int num, int denom);
```

Mkay, but that *still* doesn't explain where my input and output went. Well actually … it *does*.

> "I am not sure how clouds get formed. But the clouds know how to do it, and that is the important thing"

Found that quote long ago in one of those Kids on Science lists, and I'm always reminded of it when programming. The thing about computers is that they don't think in terms of input, output, text, pictures etc. Actually, they don't think at all, but that's another story. All a computer sees is data; not even code and data, just data since code is data too. Of course, *you* may not see it that way because you're used to C or VB or whatever, but when all is said and done, it's all just ones and zeros. If the ones and zeros come to the CPU via the program counter (**PC** register, `r15`) it's code, otherwise it's data.

So how does that explain the input/output? Well, it doesn't do it directly, but it points to how you should be looking at the situation. Consider you're the compiler and you have to convert some bloke's C code into machine code (or assembly, which is almost the same thing) that a CPU can actually use. You come across the line " `q= Div(x,y);` ". What does `Div()` do? Well, if there's no symbol in the C-file for that name (and there isn't, as it's in *tonc_bios.s*), you wouldn't know. Technically, you don't even know *what* it is. But `Div` knows, and that's the important thing. At least, that's *almost* how it works. The compiler should still need to know what sort of critter `Div` is to avoid confusion: A variable? A macro? A function? That's what the declarations are for. And the declaration above says that `Div` is a function that expects two signed integers and returns one too. As far as the compiler's concerned, it ends there.

Of course, that still doesn't explain how the compiler knows what do to. Well, it simply follows the *ARM Architecture Procedure Call Standard*, **AAPCS** for short. This states how functions should pass arguments to each other. This PDF document can be found here and if you're contemplating assembly is a very worthwhile download.

For now, here's what you need to know. The first four arguments are placed in the first four registers `r0-r3`, every one after that is placed on the stack. The output value is placed in `r0`. As long as you take the argument list of the BIOS call as the list in the declaration, it should work fine. Note that the declaration also takes care of any casting that needs to be done. It is important that you realize just what the declaration means here: *it* determines how the function is called, not the actual *definition* assembly function. Or even C function. Things can go very wrong if you mess up the declaration.

Another thing the AAPCS tells you is that register r0-r3 (and r12) are so-called *scratch* registers. This means that the caller *expects* the called function to mess these up. After the function returns their contents are to be considered undefined – unless you're the one writing both asm functions, in which case there may be certain … allowances. Having these as scratch registers means that a function can use them without needing to push and pop the originals on and off the stack, thus saving time. This does not hold for the other registers, though: r4-r11, r13, r14 *must* be returned in the way the

calling function got them. The last one, r15, is exempt from this, as you shouldn't screw around with the program counter.

Inline assembly

Actually, you don't even need a full assembly file for BIOS calls: you could use *inline assembly*. With inline assembly, you can mix C code and assembly code. Since the functions are usually rather simple, you could use something like

```c
// In a C file
int Div(int num, int denom)
{    asm("swi 0x06");    }
```

This does exactly the same thing as the assembly version of `Div`. However, you need to be careful with inline assembly because you can't see the code around it and might accidentally *clobber* some registers that you shouldn't be messing with, thus ruining the rest of the code. For the full rules on inline assembly, see the GCC manual. You can also find a short faq on inline assembly use at devrs.com. The 'proper' syntax of inline assembly isn't the friendliest in the world, mind you, and there are other problems as well. Consider the C function given above. Since it doesn't really do anything itself, the optimiser may be tempted to throw it away. This will happen with `-O3` unless you take appropriate precautions. Also, the compiler will complain that the function doesn't return anything, even though it should. It has a point, of course, considering that part is taken care of inside the assembly block. There are probably a few other problems that I'm not aware of at present; in the end it's easier to use the full-assembly versions so you know what's happening.

The `swi_call` macro

On the other hand, there are also BIOS calls that use no arguments, which can be run via a mere macro. The `swi_call(x)` macro will run the BIOS call *x*, and can be found in *swi.h*, and in Wintermute's libgba, which is where I got it from. It's a little more refined than the `Div` function given above. First, it uses the `volatile` keyword, which should keep your optimizer from deleting the function (just like we did for all

the registers). Secondly, it uses a *clobber list* (after the triple colons). This will tell the compiler which registers are used by the inline assembly. Thirdly, it will take care of the Thumb/ARM switch automatically. If you use the `-mthumb` compiler option, the compiler will define `__thumb__` for us, which we will now use to get the right swi-number. Clever, eh?

```
#ifndef(__thumb__)
#define swi_call(x)   asm volatile("swi\t"#x ::: "r0", "r1", "r2", "r3")
#else
#define swi_call(x)   asm volatile("swi\t"#x"<<16" ::: "r0", "r1", "r2",
"r3")
#endif
```

By the way, if you want more information about assembly, you can find a number of tutorials on ARM assembly at gbadev.org. Another nice way to learn is by using the `-S` compiler flag, which will give you a compiler-generated assembly file of your code. This will show you exactly what the compiler does to your code, including optimisation steps and use of the AAPCS. Really, you should see this at least once.

It may also help to use `-fverbose-asm`, which will write out the original variable names and operations in comments. Usually in the right place too. Also handy is the `ASM_CMT()` macro shown below. This will give you some hints as to where specific blocks of code are. But again, not all the time.

```
#define ASM_CMT(str) asm volatile("@ " str)

//In code. Outputs "@ Hi, I'm here!" in the generated asm
ASM_CMT("Hi, I'm here!");
```

Demo graphs

To illustrate the use of BIOS calls I am using Div, Sqrt, ArcTan and ObjAffineSet to create graphs if a hyperbole, square root, sine and cosine. I've scaled them in such a way so that they fit nicely on the 240x160 screen. The definitions are

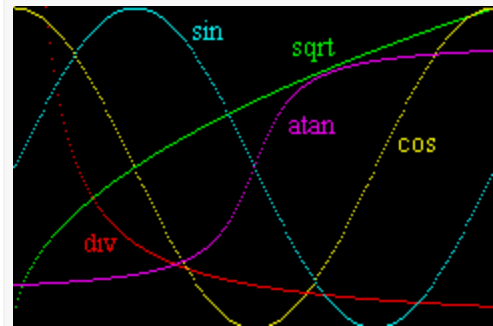| division	$y= 2560/x$	
square root	$y= 160*sqrt(x/240)$	
arctan	$y= 80 + 64*(2/\pi)*(arctan(x-120)/16))$	
sine	$y= 1*sy*sin(2\pi \cdot x/240)$	; sy= 80
cosine	$y= 80*sx*cos(2\pi \cdot x/240)$	; sx= 1

Fig 17.1: div, sqrt, arctan2, sin and cos graphs, courtesy of BIOS.

and these functions have been plotted in fig 1. If you're wondering how I got the sine and cosine values, as there are no calls for those, take a look at eq 1 again. The **P** matrix has them. I've used p_a for the cosine and p_c for the sine. Note that the graphs appear instantly; there is no sense of loading time while the graphs are plotted. An earlier version of the mode 7 demo (or PERNs mode 7 demo) used calls to the actual division, sine and cosine functions to build up the LUTs. Even with the symmetry rules of trigonometry, `sin()` and `cos()` are still noticeably slower than the BIOS equivalent.

```
#include <stdio.h>
#include <tonc.h>

// === swi calls ===========================================

// Their assembly equivalents can be found in tonc_bios.s

void VBlankIntrWait()
{   swi_call(0x05); }

int Div(int num, int denom)
{   swi_call(0x06); }

u32 Sqrt(u32 num)
{   swi_call(0x08); }

s16 ArcTan2(s16 x, s16 y)
{   swi_call(0x0a); }

void ObjAffineSet(const AFF_SRC *src, void *dst, int num, int offset)
{   swi_call(0x0f); }


// === swi demos ===========================================

// NOTE!
// To be consistent with general mathematical graphs, the
// y-axis has to be reversed and the origin moved to the
// either the bottom or mid of the screen via
// "iy = H - y"
// or
// "iy = H/2 - y"
//
// functions have been scaled to fit the graphs on the 240x160 screen

// y= 2560/x
void div_demo()
{
    int ix, y;

    for(ix=1; ix<SCREEN_WIDTH; ix++)
    {
        y= Div(0x0a000000, ix)>>16;
        if(y <= SCREEN_HEIGHT)
            m3_plot(ix, SCREEN_HEIGHT - y, CLR_RED);
    }
    tte_printf("#{P:168,132;ci:%d}div", CLR_RED);
}

// y= 160*sqrt(x/240)
```

```
void sqrt_demo()
{
    int ix, y;
    for(ix=0; ix<SCREEN_WIDTH; ix++)
    {
        y= Sqrt(Div(320*ix, 3));
        m3_plot(ix, SCREEN_HEIGHT - y, CLR_LIME);
    }
    tte_printf("#{P:160,8;ci:%d}sqrt", CLR_LIME);
}

// y = 80 + tan((x-120)/16) * (64)*2/pi
void arctan2_demo()
{
    int ix, y;
    int ww= SCREEN_WIDTH/2, hh= SCREEN_HEIGHT/2;
    for(ix=0; ix < SCREEN_WIDTH; ix++)
    {
        y= ArcTan2(0x10, ix-ww);
        m3_plot(ix, hh - y/256, CLR_MAG);
    }
    tte_printf("#{P:144,40;ci:%d}atan", CLR_MAG);
}

// wX= 1, wY= 80
// cc= 80*sx*cos(2*pi*alpha/240)
// ss=  1*sy*sin(2*pi*alpha/240)
void aff_demo()
{
    int ix, ss, cc;
    ObjAffineSource af_src= {0x0100, 0x5000, 0};     // sx=1, sy=80,
alpha=0
    ObjAffineDest af_dest= {0x0100, 0, 0, 0x0100};  // =I (redundant)

    for(ix=0; ix<SCREEN_WIDTH; ix++)
    {
        ObjAffineSet(&af_src, &af_dest, 1, BG_AFF_OFS);
        cc= 80*af_dest.pa>>8;
        ss= af_dest.pc>>8;
        m3_plot(ix, 80 - cc, CLR_YELLOW);
        m3_plot(ix, 80 - ss, CLR_CYAN);
        // 0x010000/0xf0 = 0x0111.111...
        af_src.alpha += 0x0111;
    }

    tte_printf("#{P:48,38;ci:%d}cos", CLR_YELLOW);
    tte_printf("#{P:72,20;ci:%d}sin", CLR_CYAN);
}

// === main ============================================================
```

```
int main()
{
    REG_DISPCNT= DCNT_MODE3 | DCNT_BG2;

    tte_init_bmp_default(3);
    tte_init_con();

    div_demo();
    sqrt_demo();
    aff_demo();

    arctan2_demo();

    while(1);

    return 0;
}
```

Vsyncing part II, VBlankIntrWait

Until now, all demos used the function `vid_vsync` to synchronize the action to the VBlank (see the graphics introduction). What this did was to check `REG_VCOUNT` and stay in a while loop until the next VBlank came along. While it works, it's really a pretty poor way of doing things for two reasons. First, because of the potential problem when you are in a VBlank already, but that one had been covered. The second reason is more important: while you're in the while loop, you're wasting an awful lot of CPU cycles, all of which slurp battery power.

There are a number of BIOS calls that can put the CPU into a low power mode, thus sparing the batteries. The main BIOS call for this is Halt (#2), but what we're currently interested in is VBlankIntrWait (#5). This will set things up to wait until the next VBlank interrupt. To use it, you have to have interrupts switched on, of course, in particular the VBlank interrupt. As usual, the VBlank isr will have to acknowledge the interrupt by writing to `REG_IF`. But it *also* has to write to its BIOS equivalent, `REG_IFBIOS`. This little bit of information is a little hard to find elsewhere (in part because few tutorials

cover BIOS calls); for more info, see GBATEK, BIOS Halt Functions. Fortunately for us, the switchboard presented in the interrupts section has this built in.

To show you how to set it up, see the `swi_vsync` demo. The most important code is given below; a screen shot can be found in fig 2. What it does is give a rotating metroid sprite with an angular
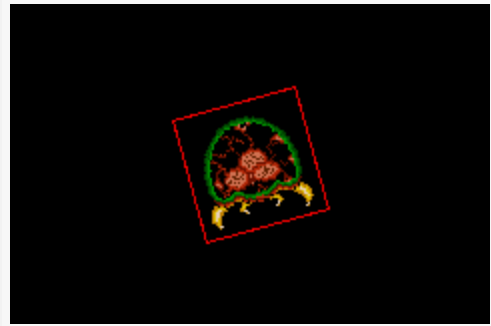

Fig 17.2: `swi_vsync` demo.

velocity of π rad/s (this corresponds to Δθ = 0x10000/4/60 = 0x0111). The basic steps for interrupt handling should be familiar, except the fact that there's no real VBlank isr because the switchboard already takes care of acknowledging the interrupt. After that it's pretty simple: we use `ObjAffineSet()` to calculate the required affine matrix and `VBlankIntrWait` puts the CPU on Halt until the next VBlank interrupt.

```
// inside main, after basic initialisations

AFF_SRC as= { 0x0100, 0x0100, 0 };
OBJ_AFFINE oaff;

// enable isr switchboard and VBlank interrupt
irq_init(NULL);
irq_add(II_VBLANK, NULL);

while(1)
{
    VBlankIntrWait();

    // Full circle = 10000h
    // 10000h/4/60= 111h -> 1/4 rev/s = 1 passing corner/s
    as.alpha += 0x0111;
    ObjAffineSet(&as, &oaff.pa, 1, 8);

    obj_aff_copy(obj_aff_mem, &oaff, 1);
}
```

PREFER VBLANKINTRWAIT() OVER VID_VSYNC()

Waiting for the VBlank via `vid_vsync()` (or its functional equivalent) is not a good idea: it wastes too much battery power. The recommended procedure is using `VBlankIntrWait()` to halt the processor, to be woken again on the VBlank interrupt.

ACKNOWLEDGING INTRWAIT ROUTINES

`VBlankIntrWait()` is only one of the BIOS's `IntrWait()` routines that can stop the CPU until an interrupt has been raised. However, it doesn't look at `REG_IF` but at `REG_IFBIOS` (0300:7FF8) for the acknowledgement of the interrupt. If your game locks up after trying `VBlankIntrWait()`, this may be why. Note that you may find the address under other names, as there isn't really an official one for it.

Final thoughts

Now that you know how to use them, I should warn you that you shouldn't go overboard with them. It appears that the BIOS routines have been designed for space, not speed, so they aren't the fastest in the world. Not only that, there's an overhead of at least 60 cycles for each one (mind you, normal functions seem to have a 30 cycle overhead). If speed is what you're after then the BIOS calls may not be the best thing; you can probably find faster routines on the web … somewhere. This doesn't mean that the BIOS routines can't be useful, of course, but if you have alternative methods, use those instead. Just remember that that's an optimisation step, which you shouldn't do prematurely.

18. Beep! GBA sound introduction

Introduction to GBA sound

Apart from graphics and interaction, there is one other sense important to games: audio. While graphics may set the scene, sound sets the mood, which can be even more important that the graphics. Try playing *Resident Evil* with, say, "Weird Al" Yankovic playing: it simply doesn't work, the atmosphere is lost.

The GBA has six sound channels. The first four are roughly the same as the original Game Boy had: two square wave generators (channels 1 and 2), a sample player (channel 3) and a noise generator (channel 4). Those are also referred to as the DMG channels after the Game Boy's code name "Dot Matrix Game." New are two Direct Sound channels A and B (not to be confused with Microsoft's DirectSound, the DirectX component). These are 8-bit digital pulse code modulation (PCM) channels.

I should point out that I really know very little about sound programming, mostly because I'm not able to actually put together a piece of music (it's kinda hard to do that when you already have music playing). If you want to really learn about sound programming, you should look at Belogic.com, where almost everybody got their information from, and deku.gbadev.org, which shows you how to build a sound mixer. Both of these sites are excellent.

I may not know much about sound creation/programming, but at its core sound is a wave in matter; waves are mathematical critters, and I *do* know a thing or two about

math, and that's kind of what I'll do here for the square wave generators.

Sound and Waves

Consider if you will, a massive sea of particles, all connected to their neighbours with little springs. Now give one of them a little push. In the direction of the push, the spring compresses and relaxes, pushing the original particle back to its normal position and passing on the push to the neighbour; this compresses the next spring and relays the push to *its* neighbour, and so on and so on.

This is a prime example of wave behaviour. Giving a precise definition of a wave that covers all cases is tricky, but in essence, a **wave** is a **transferred disturbance**. There are many kinds of waves; two major classes are **longitudinal** waves, which oscillate in the direction of travel, and **transverse** waves, which are perpendicular to it. Some waves are periodic (repeating patterns over time or space), some aren't. Some travel, some don't.

Waves

The canonical wave is the **harmonic wave**. This is any function $\psi(x)$ that's a solution to eq 18.1. The name of the variable doesn't really matter, but usually it's either spatial (x, y, z) or temporal (t), or all of these at the same time. The general solution can be found in eq 18.2. Or perhaps I should say solution**s**, as there are many ways of writing them down. They're all equivalent though, and you can go from one to the other with some trickery that does not concern us at this moment.

$$(18.1) \qquad \frac{\mathrm{d}^2}{\mathrm{d}x^2}\psi(x) + k^2\,\psi(x) = 0$$

General solution(s):

$$
\begin{aligned}
\psi(x) \;&=\; A \cdot cos(kx) + B \cdot sin(kx) \\
&=\; C \cdot e^{ikx} + D \cdot e^{-ikx} \\
&=\; E \cdot sin(kx + \varphi_0)
\end{aligned}
$$

(18.2)

A full wave can be described by three things. First, there's the **amplitude** A, which gives half-distance between the minimum and maximum. Second, the **wavelength** λ, which is the length after which the wave repeats itself (this is tied to wave-number $k= 2\pi/\lambda$). Then there's **phase constant** ϕ_0, which defines the

Fig 18.1: a harmonic wave

stating point. If the wave is in time, instead of a wavelength you have **period** T, **frequency** f=1/T (and angular frequency ω= 2πf= 2π/T). You can see what each of these parameters is in fig 18.1.
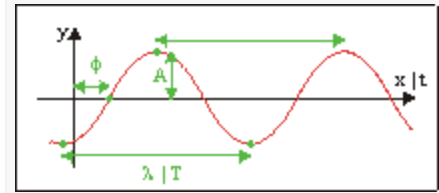
One of the interesting things about the wave equation is that it is a linear operation on ψ. What that means is that any combination of solutions is also a solution; this is the **superposition principle**. For example, if you have two waves ψ_1 and ψ_2, then Ψ = $a\psi_1$ + $b\psi_2$ is also a wave. This may sound like a trivial thing but I assure you it's not. The fact that non-linear equations (and they exist too) tend to make scientists cringe a little should tell you something about the value of linear equations.

Sound waves

Sound is also a wave. In fact, it is a longitudinal pressure wave in matter and pretty much works as the system of particles on springs mentioned earlier with whole sets of molecules moving back and forth. In principle, it has both spatial and temporal structure, and things can get hideously complex if you want to deal with everything. But I'll keep it easy and only consider two parts: amplitude A and period and frequency T and f. As you probably know, the tone of a sound is related to the frequency. Human hearing has a range between 20 Hz and 20 kHz, and the higher the frequency (that is, the more compressed the wave), the higher the tone. Most sounds are actually a conglomeration of different waves, with different amplitudes and

frequencies – the superposition principle at work. The funny thing about this is that if you added all those components up to one single function and plot it, it wouldn't look like a sine wave at all anymore. What's even funnier is that you can also reverse the process and take a function –*any* function– and break it up into a superposition of sine and cosine waves, and so see what kind of frequencies your sound has. This is called Fourier Transformation, and we'll get to that in a minute.

Musical scale

While the full range between 20 Hz and 20 kHz is audible, only a discrete set of frequencies are used for music, which brings us to the notion of the *musical scale*. Central to these are *octaves*, representing a frequency doubling. Each octave is divided into a number of different notes; 12 in Western systems, ranging from A to G, although octave numbering starts at C for some reason. Octave 0 starts at the *central C*, which has a frequency of about 262 Hz (see also table 18.1. And yes, I know there are only 7 letters between A and G, the other notes are flats and sharps that lie between these notes. The '12' refers to the number of half-notes in an octave. The musical scale is **logarithmic**; each half-note being $2^{1/12}$ apart. Well, almost anyway: for some reason, some notes don't *quite* fit in exactly.

half-note	0	1	2	3	4	5	6	7
name	C	C#	D	D#	E	F	F#	G
freq (Hz)	261.7	277.2	293.7	311.2	329.7	349.3	370.0	392.0

Table 18.1: notes & frequencies of octave 0

Fourier transforms and the square wave

Fourier transformations are a way of going describing a function in the time domain as a distribution of frequencies called a *spectrum*. They're also one of the many ways that professors can scare the bejebus out of young, natural-science students. Don't worry, I'm sure you'll get through this section unscathed >:) . For well- to

reasonably-behaved functions, you can rewrite them as series of *very* well-behaved functions such as polynomials, exponentials and also waves. For example, as a Fourier series, a function may look like eq 18.3.

$$(18.3) \qquad f(x) = \frac{1}{2}A_0 + \sum_{n>0} A_m \cos(m\omega t) + \sum_{n>0} B_m \sin(m\omega t)$$

Of course, the whole thing relies on being able to find the coefficients A_m and B_m.

While it is fairly straightforward to derive the equations for them, I'll leave that as an exercise for the reader and just present the results in the form of eq 18.4. I should mention that there are actually a few ways of defining Fourier transforms. For example, there are versions that don't integrate over $[0,T]$, but over $[-\frac{1}{2}T, \frac{1}{2}T]$; or use the complex exponential instead of sines and cosines, but in the end they're all doing the same thing.

$$(18.4) \qquad \begin{aligned} A_m &= \tfrac{2}{T}\int_0^T f(t)\cos(m\omega t)dt \\ B_m &= \tfrac{2}{T}\int_0^T f(t)\sin(m\omega t)dt \end{aligned}$$

As an example, let's take a look at the square wave shown in fig 18.2. A square wave is on (1) for a certain time (parameter h), then off (0) for the rest of the cycle. It's still a periodic wave, so it doesn't really matter where we place the thing along the t-axis. I centered it on the peak for convenience: doing so

Fig 18.2: a square wave

makes it a symmetrical wave which has the nice properly of removing *all* the anti-symmetrical sine waves. $A_0=h/T$ because it's the average of the function and the rest of the A_m's follow from eq 18.4.
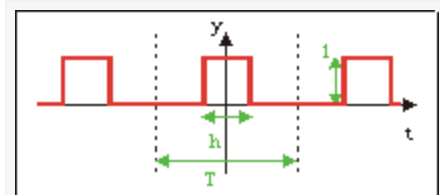
$$(18.5) \qquad A_m = \frac{2}{\pi} \cdot \frac{\sin(\pi m h / T)}{m} = \frac{2T}{h} \cdot \frac{\sin(\pi h / T \cdot m)}{\pi h / T \cdot m}$$

A_m is a **sinc** function: $\sin(x)/x$. For high m it approaches zero (as it should, since higher terms should be relatively less important), but also interesting is that of the higher terms some will also *vanish* because of the sine. This will happen whenever m is a multiple of T/h.

GBA sound

Sound registers

For graphics, you only had to deal with two registers (`REG_DISPCNT` and `REG_BGxCNT`) to get a result; for sound, you have to cover a lot of registers before you get *anything*. The DMG channels each have 2 or 3 registers – some with similar functionality, some not. Apart from that, there are four overall control registers.

The register nomenclature seems particularly vexed when it comes to sound. There are basically two sets of names that you can find: one consisting of `REG_SOUNDxCNT` followed by `_L` , `_H` and `_X` in a rather haphazard manner; the other one uses a `REG_SGxy` and `REG_SGCNTy` structure (x=1, 2, 3 or 4 and y=0 or 1). I think the former is the newer version, which is funny because the older is more consistent. Oh well. In any case, I find neither of them very descriptive and keep forgetting which of the L/H/X or 0/1 versions does what, so I use a *third* set of names based on the ones found in tepples' pin8gba.h, which IMHO makes more sense than the other two.

offset	function	old	new	tonc
60h	channel 1 (sqr) sweep		SOUND1CNT_L	REG_SND1SWEEP
62h	channel 1 (sqr) len, duty, env	REG_SG10	SOUND1CNT_H	REG_SND1CNT
64h	channel 1 (sqr) freq, on	REG_SG11	SOUND1CNT_X	REG_SND1FREQ

68h	channel 2 (sqr) len, duty, env	REG_SG20	SOUND2CNT_L	REG_SND2CNT
6Ch	channel 2 (sqr) freq, on	REG_SG21	SOUND2CNT_H	REG_SND2FREQ
70h	channel 3 (wave) mode	REG_SG30	SOUND3CNT_L	REG_SND3SEL
72h	channel 3 (wave) len, vol		SOUND3CNT_H	REG_SND3CNT
74h	channel 3 (wave) freq, on	REG_SG31	SOUND3CNT_X	REG_SND3FREQ
78h	channel 4 (noise) len, vol, env	REG_SG40	SOUND4CNT_L	REG_SND4CNT
7Ch	channel 4 (noise) freq, on	REG_SG41	SOUND4CNT_H	REG_SND4FREQ
80h	DMG master control	REG_SGCNT0	SOUNDCNT_L	REG_SNDDMGCNT
82h	DSound master control		SOUNDCNT_H	REG_SNDDSCNT
84h	sound status	REG_SGCNT1	SOUNDCNT_X	REG_SNDSTAT
88h	bias control	REG_SGBIAS	SOUNDBIAS	REG_SNDBIAS

Table 18.2: Sound register nomenclature.

"Oh great. This is going to be one of 'tegel' things isn't it? Where *you* think you've got something nice but different going, then later you revert to the standard terminology to conform with the rest of the world. Right?"

No, I'll stick to these names. Probably. Hopefully. … To be honest, I really don't know
`:P` . This is not really a big deal, though: you can easily switch between names with a
few defines or search & replaces. Anyway, `REG_SNDxFREQ` contains frequency
information and `REG_SNDxCNT` things like volume and envelope settings; in some
cases, the bit layouts are even exactly the same. Apart from the sweep function of
channel 1, it is exactly the same as channel 2.

Master sound registers

`REG_SNDDMGCNT` , `REG_SNDDSCNT` and `REG_SNDSTAT` are the master sound controls;
you have to set at least some bits on each of these to get anything to work.

REG_SNDDMGCNT (SOUNDCNT_L / SGCNT0_L) @ 0400:0080h

F	E	D	C	B	A	9	8	7	6 5 4	3	2 1 0
R4	R3	R2	R1	L4	L3	L2	L1	–	RV	–	LV

bits	name	define	description
0-2	LV		Left volume
4-6	RV		Right volume
8-B	L1-L4	SDMG_LSQR1, SDMG_LSQR2, SDMG_LWAVE, SDMG_LNOISE	Channels 1-4 on left
C-F	R1-R4	SDMG_RSQR1, SDMG_RSQR2, SDMG_RWAVE, SDMG_RNOISE	Channels 1-4 on right

`REG_SNDDMGCNT` controls the main volume of the DMG channels and which ones are
enabled. These controls are separate for the left and right speakers. Below are two
macros that make manipulating the register easier. Note that they *don't* actually set
the register, just combine the flags.

```
#define SDMG_SQR1    0x01
#define SDMG_SQR2    0x02
#define SDMG_WAVE    0x04
#define SDMG_NOISE   0x08

#define SDMG_BUILD(_lmode, _rmode, _lvol, _rvol)    \
    ( ((_lvol)&7) | (((_rvol)&7)<<4) | ((_lmode)<<8) | ((_rmode)<<12) )

#define SDMG_BUILD_LR(_mode, _vol) SDMG_BUILD(_mode, _mode, _vol, _vol)
```

REG_SNDDSCNT (SOUNDCNT_H / SGCNT0_H) @ 0400:0082h

F	E	D	C	B	A	9	8	7 6 5 4	3	2	1 0
BF	BT	BL	BR	AF	AT	AL	AR	-	BV	AV	DMGV

bits	name	define	description
0-1	DMGV	SDS_DMG25, SDS_DMG50, SDS_DMG100	DMG Volume ratio. • **00**: 25% • **01**: 50% • **10**: 100% • **11**: forbidden
2	AV	SDS_A50, SDS_A100	DSound A volume ratio. 50% if clear; 100% of set
3	BV	SDS_B50, SDS_B100	DSound B volume ratio. 50% if clear; 100% of set
8-9	AR, AL	SDS_AR, SDS_AL	**DSound A enable** Enable DS A on right and left speakers
A	AT	SDS_ATMR0, SDS_ATMR1	**Dsound A timer**. Use timer 0 (if clear) or 1 (if set) for DS A
B	AF	SDS_ARESET	**FIFO reset for Dsound A**. When using DMA for Direct sound, this will cause DMA to reset the FIFO buffer after it's used.
C-F	BR, BL,	SDS_BR, SDS_BL, SDS_BTMR0,	As bits 8-B, but for DSound B

	BT,	SDS_BTMR1,	
	BF	SDS_BRESET	

Don't know too much about `REG_SNDDSCNT`, apart from that it governs PCM sound, but also has some DMG sound bits for some reason. `REG_SNDSTAT` shows the status of the DMG channels *and* enables all sound. If you want to have any sound at all, you need to set bit 7 there.

```
REG_SNDSTAT (SOUNDCNT_X / SGCNT1) @ 0400:0084h
```

F E D C B A 9 8	7	6 5 4	3̄	2̄	1̄	0̄
–	MSE	–	4A	3A	2A	1A

bits	name	define	description
0̲-3	1A-4A	SSTAT_SQR1, SSTAT_SQR2, SSTAT_WAVE, SSTAT_NOISE	**Active channels**. Indicates which DMG channels are currently playing. They do *not* enable the channels; that's what `REG_SNDDMGCNT` is for.
7	MSE	SSTAT_DISABLE, SSTAT_ENABLE	**Master Sound Enable**. Must be set if any sound is to be heard at all. Set this **before** you do anything else: the other registers can't be accessed otherwise, see GBATEK for details.

SOUND REGISTER ACCESS

Emulators may allow access to sound registers even if sound is disabled (`REG_SNDSTAT` {7} is clear), but hardware doesn't. Always enable sound before use.

GBA Square wave generators

The GBA has two square sound generators, channels 1 and 2. The only difference between them is channel 1's *frequency sweep*, which can make the frequency rise or

drop exponentially as it's played. That's all done with `REG_SND1SWEEP`. `REG_SNDxCNT` controls the wave's length, envelope and duty cycle. Length should be obvious. The *envelope* is basically the amplitude as function of time: you can make it fade in (*attack*), remain at the same level (*sustain*) and fade out again (*decay*). The envelope has 16 volume levels and you can control the starting volume, direction of the envelope and the time till the next change. Volumes are linear: 12 produces twice the amplitude of 6. The *duty* refers to the ratio of the 'on' time and the period, in other words $D = h/T$.

Of course, you can control the frequency as well, namely with `REG_SNDxFREQ`. However, it isn't the frequency that you enter in this field. It's not exactly the period either; it's something I'll refer to as the *rate* R. The three quantities are related, but different in subtle ways and chaos ensues when they're confused – and they often *are* in documentation, so be careful. The relation between frequency f and rate R is described by eq 18.6; if the rate goes up, so does the frequency. Since $R \in [0, 2047]$, the range of frequencies is [64 Hz, 131 kHz]. While this spans ten octaves, the highest ones aren't of much use because the frequency steps become too large (the denominator in eq eq 18.6 approaches 0).

$$ (18.6a) \qquad f(R) = \frac{2^{17}}{2048 - R} $$

$$ (18.6b) \qquad R(f) = 2048 - \frac{2^{17}}{f} $$

Square sound registers

Both square-wave generators have registers `REG_SNDxCNT` for envelope/length/duty control and `REG_SNDxFREQ` for frequency control. Sound 1 also has sweep control in the form of `REG_SND1SWEEP`. Look in table 18.2 for the traditional names; note that in traditional nomenclature the suffixes for control and frequency are *different* for channels 1 and 2, even though they have exactly the same function.

```
REG_SND1CNT (SOUND1CNT_H / SG10_H) @ 0400:0062h
and
REG_SND2CNT (SOUND2CNT_L / SG20_L) @ 0400:0068h
```

F E D C	B	A 9 8	7 6	5 4 3 2 1 0
EIV	ED	EST	D	L

bits	name	define	description
0-5	L	SSQR_LEN#	Sound **Length**. This is a *write-only* field and only works if the channel is timed (`REG_SNDxFREQ{E}`). The length itself is actually (64−*L*)/256 seconds for a [3.9, 250] ms range.
6-7	D	SSQR_DUTY1_8, SSQR_DUTY1_4, SSQR_DUTY1_2, SSQR_DUTY3_4, SSQR_DUTY#	Wave **duty cycle**. Ratio between on and of times of the square wave. Looking back at eq 18.2, this comes down to *D*=*h*/*T*. The available cycles are 12.5%, 25%, 50%, and 75% (one eighth, quarter, half and three quarters).
8-A	EST	SSQR_TIME#	Envelope **step-time**. Time between envelope changes: Δt = *EST*/64 s.
B	ED	SSQR_DEC, SSQR_INC	Envelope **direction**. Indicates if the envelope decreases (default) or increases with each step.
C-F	EIV	SSQR_IVOL#	Envelope **initial value**. Can be considered a **volume** setting of sorts: 0 is silent and 15 is full volume. Combined with the direction, you can have fade-in and fade-outs; to have a sustaining sound, set initial volume to 15 and an increasing direction. To vary the *real* volume, remember `REG_SNDDMGCNT` .

$$A_m = \frac{2}{\pi} \cdot \frac{sin(\pi Dm)}{m}$$

Some more on the duty cycle. Remember we've done a Fourier analysis of the square wave so we could determine the frequencies in it. Apart from the **base frequency**, there are also **overtones** of frequencies $m \cdot f$. The spectrum (see fig 18.3) gives the amplitudes of all these frequencies. Note that even though the figure has lines, only integral values of m are allowed. The base frequency at $m=1$ has

Fig 18.3: Square wave spectrum. (integer m only)

the highest significance and the rest falls off with $1/m$. The interesting part is when the sine comes into play: whenever $m \cdot D$ is an integer, that component vanishes! With a fractional duty number –like the ones we have– this happens every time m is equal to the denominator. For the 50% duty, every second overtone disappears, leaving a fairly smooth tone; for 12.5%, only every eighth vanishes and the result is indeed a noisier sound. Note that for *both* ¼ and ¾ duties every fourth vanishes so that they should be indistinguishable. I was a little surprised about this result, but sure enough, when I checked they really did sound the same to me.
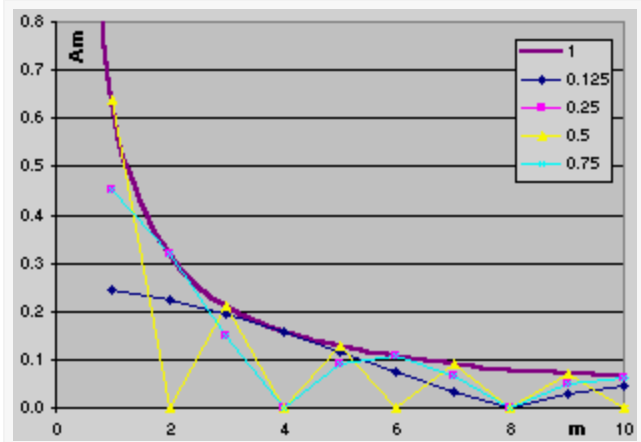
```
REG_SND1FREQ (SOUND1CNT_X / SG11) @ 0400:0062h
and
REG_SND2FREQ (SOUND2CNT_H / SG21) @ 0400:006Ch
```

F	E	D C B	A 9 8 7 6 5 4 3 2 1 0
Re	T	–	R

bits	name	define	description
0-A	R	SFREQ_RATE#	Sound **rate**. Well, initial rate. That's *rate*, not frequency. Nor period. The relation between rate and frequency is $f = 2^{17}/(2048 - R)$. Write-only field.
E	T	SFREQ_HOLD, SFREQ_TIMED	**Timed** flag. If set, the sound plays for as long as the length field (`REG_SNDxCNT` {0-5}) indicates. If clear, the sound plays forever. Note that even if a decaying envelope has

			reached 0, the sound itself would still be considered on, even if it's silent.
F	Re	SFREQ_RESET	Sound **reset**. Resets the sound to the initial volume (and sweep) settings. Remember that the rate field is in this register as well and due to its write-only nature a simple ' `|= SFREQ_RESET` ' will *not* suffice (even though it might on emulators).

```
REG_SND1SWEEP (SOUND1CNT_L / SG10_L) @ 0400:0060h
```

F E D C B A 9 8 7	6 5 4	3	2 1 0
–	T	M	N

bits	name	define	description
0-2	N	SSW_SHIFT#	Sweep **number**. *Not* the number of sweeps; see the discussion below.
3	M	SSW_INC, SSW_DEC	Sweep **mode**. The sweep can take the rate either up (default) or down (if set).
4-6	T	SSW_TIME#	Sweep **step-time**. The time between sweeps is measured in 128 Hz (not kHz!): $\Delta t = T/128$ ms $\approx 7.8T$ ms; if $T=0$, the sweep is disabled.

I'm reasonably confident that the *exact* workings of shifts are explained without due care in most documents, so here are a few more things about it. Sure enough, the sweep *does* make the pitch go up or down which is controlled by bit 3, and the step-time *does* change the pitch after that time, but exactly what the sweep-shift does is ambiguous at best. The information is in there, but only if you know what to look for. The usual formula given is something like:

$$T = T \pm T \cdot 2^{-n}$$

That's what belogic gives and if you know what the terms are you'll be fine. Contrary to what you may read, the sweep does *not* apply to the frequency (*f*). It does *not* apply

to the period (*T*, see above). It applies to the **rate** (*R*). If you look in emulators, you can actually *see* the rate-value change.

Second, the *n* in the exponent is *not* the current sweep index that runs up to the number of sweep shifts. It is in fact simply the **sweep shift number**, and the sweeps continue until the rate reaches 0 or the maximum of 2047.

The formulas you may see do say that, but it's easy to misread them. I did. Eq 18.7 holds a number of correct relations. *R* is the rate, *n* is the sweep shift (18.7c explains why it's called a *shift* (singular, not plural)), and *j* is the current sweep index. You can view them in a number of ways, but they all boil down to exponential functions, that's what '$dy^*(x) = a \cdot y(x)dx^*$' means, after all. For example, if *n*=1, then you get $1\frac{1}{2}^j$ and $\frac{1}{2}^j$ behaviour for increasing and decreasing sweeps, respectively; with *n*=2 it's $1\frac{1}{4}^j$ and $\frac{3}{4}^j$, etc. The higher the shift, the slower the sweep.

(18.7a)	$\Delta R = 2^{-n} \cdot R$

$$
\begin{aligned}
R_j &= R_{j-1} \pm R_{j-1} \cdot 2^{-n} \\
\text{(18.7b)} \quad &= R_{j-1} \cdot (1 \pm 2^{-n}) \\
&= R_0 \cdot (1 \pm 2^{-n})^j
\end{aligned}
$$

(18.7c)	`R += R >> n;`

Playing notes

Even though the rates are equal, some may be considered more equal than others. I've already given a table with the frequencies for the standard notes (table 18.1) of octave 0. You can of course convert those to rates via eq 18.6b and use them as such. However, it might pay to figure out how to play the notes of *all* octaves.

To do this, we'll use some facts I mentioned in section 18.2.3. about the make-up of the musical scale. While I *could* make use of the logarithmic relation between successive notes ($\Delta f = 2^{1/12} \cdot f$), I'll restrict myself to the fact that notes between octaves differ by a factor of two. We'll also need the rate-frequency relation (obviously). That's

the basic information you need, I'll explain more once we get through all the math. Yes, it's more math, but it'll be the last of this page, I promise.

The equations we'll start with are the general frequency equation and the rate-frequency relation. In these we have rate R, frequency f and octave c. We also have a base octave C and frequency F in that base octave.

$$
\begin{aligned}
f(F,c) &= F \cdot 2^{c-C} \\
R(F,c) &= 2^{11} - \frac{2^{17}}{f(F,c)}
\end{aligned}
$$

And now for the magic. And you *are* expected to understand this.

$$
\begin{aligned}
R(F,c) &= 2^{11} - \frac{2^{17}}{f(F,c)} \\
&= 2^{11} - \frac{2^{17}}{F \cdot 2^{c-C}} \\
(18.8) \qquad &= 2^{11} - \frac{2^{17+C-c}}{F} \\
&= 2^{11} - \frac{1}{F} \cdot 2^{17+C+m-(c+m)} \\
&= 2^{11} - \frac{2^{17+C+m}}{F} \cdot 2^{-(c+m)}
\end{aligned}
$$

Right, and now for *why* this thing's useful. Remember that the GBA has no hardware division or floating-point support, so we're left with integers and (if possible) shifts. That's why the last term in the last step of eq 18.8 was separated. The term with F gives a rate offset for the base octave, which we need to divide (read: shift) by the octave offset term for the different octaves. Remember that integer division truncates, so we need a big numerator for the most accuracy. This can be done with a large C and by adding an extra term m. Basically, this makes it an mf fixed point division. The workable octave range is −2 to 5, so we take C=5. The value for m is *almost* arbitrary, but needs to be higher than two because of the minimum octave is −2, and a shift can never be negative. m=4 will suffice.

Note that there is *still* a division in there. Fortunately, there are only twelve values available for F, so might just as well store the whole term in a look-up table. The final

result is listing 18.1 below.

```
// Listing 18.1: a sound-rate macro and friends

typedef enum
{
    NOTE_C=0, NOTE_CIS, NOTE_D,   NOTE_DIS,
    NOTE_E,   NOTE_F,   NOTE_FIS, NOTE_G,
    NOTE_GIS, NOTE_A,   NOTE_BES, NOTE_B
} eSndNoteId;

// Rates for equal temperament notes in octave +5
const u32 __snd_rates[12]=
{
    8013, 7566, 7144, 6742, // C , C#, D , D#
    6362, 6005, 5666, 5346, // E , F , F#, G
    5048, 4766, 4499, 4246  // G#, A , A#, B
};

#define SND_RATE(note, oct) ( 2048-(__snd_rates[note]>>(4+(oct))) )

// sample use: note A, octave 0
    REG_SND1FREQ= SFREQ_RESET | SND_RATE(NOTE_A, 0);
```

Here you have a couple of constants for the note-indices, the LUT with rate-offsets `__snd_rates` and a simple macro that gives you what you want. While `__snd_rates` is constant here, you may consider a non-const version to allow tuning. Not that a square wave is anything worth tuning, but I'm just saying … y'know.

One possible annoyance is that you have to splice the note into a note and octave part and to do that dynamically you'd need division and modulo by 12. Or do you? If you knew a few things about division by a constant is multiplication by its reciprocal, you'd know what to do. (Hint: $c=(N*43>>9)-2$, with N the total note index between 0 and 95 (octave −2 to +5).)

Demo time

I think I've done about enough theory for today, don't you dear reader?

" @_@ "

I'll take that as a yes. The demo in question demonstrates the use of the various macros of this chapter, most notably `SND_RATE` . It also shows how you can play a little song – and I use the term lightly – with the square wave generator. I hope you can recognize which one.

```c
#include <stdio.h>
#include <tonc.h>

u8 txt_scrolly= 8;

const char *names[]=
{    "C ", "C#", "D ", "D#", "E ", "F ", "F#", "G ", "G#", "A ", "A#", "B
"   };

// === FUNCTIONS ========================================================

// Show the octave the next note will be in
void note_prep(int octave)
{
    char str[32];
    siprintf(str, "[  %+2d]", octave);
    se_puts(8, txt_scrolly, str, 0x1000);
}


// Play a note and show which one was played
void note_play(int note, int octave)
{
    char str[32];

    // Clear next top and current rows
    SBB_CLEAR_ROW(31, (txt_scrolly/8-2)&31);
    SBB_CLEAR_ROW(31, txt_scrolly/8);

    // Display note and scroll
    siprintf(str, "%02s%+2d", names[note], octave);
    se_puts(16, txt_scrolly, str, 0);

    txt_scrolly -= 8;
    REG_BG0VOFS= txt_scrolly-8;

    // Play the actual note
    REG_SND1FREQ = SFREQ_RESET | SND_RATE(note, octave);
}


// Play a little ditty
void sos()
{
    const u8 lens[6]= { 1,1,4, 1,1,4 };
    const u8 notes[6]= { 0x02, 0x05, 0x12,  0x02, 0x05, 0x12 };
    int ii;
    for(ii=0; ii<6; ii++)
    {
        note_play(notes[ii]&15, notes[ii]>>4);
```

```
        VBlankIntrDelay(8*lens[ii]);
    }
}

int main()
{
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    irq_init(NULL);
    irq_add(II_VBLANK, NULL);

    txt_init_std();
    txt_init_se(0, BG_CBB(0) | BG_SBB(31), 0, CLR_ORANGE, 0);
    pal_bg_mem[0x11]= CLR_GREEN;

    int octave= 0;

    // turn sound on
    REG_SNDSTAT= SSTAT_ENABLE;
    // snd1 on left/right ; both full volume
    REG_SNDDMGCNT = SDMG_BUILD_LR(SDMG_SQR1, 7);
    // DMG ratio to 100%
    REG_SNDDSCNT= SDS_DMG100;

    // no sweep
    REG_SND1SWEEP= SSW_OFF;
    // envelope: vol=12, decay, max step time (7) ; 50% duty
    REG_SND1CNT= SSQR_ENV_BUILD(12, 0, 7) | SSQR_DUTY1_2;
    REG_SND1FREQ= 0;

    sos();

    while(1)
    {
        VBlankIntrWait();
        key_poll();

        // Change octave:
        octave += bit_tribool(key_hit(-1), KI_R, KI_L);
        octave= wrap(octave, -2, 6);
        note_prep(octave);

        // Play note
        if(key_hit(KEY_DIR|KEY_A))
        {
            if(key_hit(KEY_UP))
                note_play(NOTE_D, octave+1);
            if(key_hit(KEY_LEFT))
                note_play(NOTE_B, octave);
            if(key_hit(KEY_RIGHT))
```

```
                note_play(NOTE_A, octave);
            if(key_hit(KEY_DOWN))
                note_play(NOTE_F, octave);
            if(key_hit(KEY_A))
                note_play(NOTE_D, octave);
        }

        // Play ditty
        if(key_hit(KEY_B))
            sos();
    }
    return 0;
}
```

The bolded code in `main()` initializes the sound register; nothing fancy, but it has to be done before you hear anything at all. It is important to start with `REG_SNDSTAT` bit 7 (`SSTAT_ENABLE`), i.e., the master sound enable. Without it, you cannot even access the other registers. Setting volume to something non-zero is a good idea too, of course. Then we turn off the sweep function and set sound 1 to use a fading envelope with a 50% duty. And that's where the fun starts.

I'll explain what `sos()` in a little while, first something about the controls of the demo. You can play notes with the D-pad and A (hmm, there's something familiar about that arrangement). The octave c you're working in can be changed with L and R; the background color changes with it. B plays `sos()` again.

A / D-pad	Play a note
	↑ : D (next octave)
	← : B
	→ : A
	↓ : F
	A : D
L / R	Decrease / Increase current octave ([-2, 5], wraps around)
B	Play a little tune.

The D-pad and A select a note to play, which is handled by `note_play()`. The bolded line there plays the actual note, the rest is extra stuff that writes the note just played to the screen and scrolls along so you can see the history of what's been played. The code for this is kinda ugly, but is not exactly central to the story so that's fine.

Playing a little ditty

So what is `sos()` all about then? Let's take another look.

```
void sos()
{
    const u8 lens[6]= { 1,1,4, 1,1,4 };
    const u8 notes[6]= { 0x02, 0x05, 0x12,  0x02, 0x05, 0x12 };
    int ii;
    for(ii=0; ii<6; ii++)
    {
        note_play(notes[ii]&15, notes[ii]>>4);
        VBlankIntrDelay(8*lens[ii]);
    }
}
```

There are two arrays here, `notes` and `lens`, and a loop over all elements. We take a byte from `notes` and use the nybbles for octave and note information, play the note, then wait a while –the length is indicated by the `lens` array– before the next note is played. Basically, we're playing music. Hey, if the likes of *Schnappi* and *Crazy Frog* can make it into the top 10, I think I'm allowed to call *this* music too, alright? Alright.

The point I'm trying to make is that it's very well possible to play a tune with just the tone generators. Technically you don't need digitized music and all that stuff to play something. Of course, it'll sound better if you do, but if you just need a little jingle the tone generators may be all you need. Twelve years of Game Boy games using only tone generators prove this. Just define some notes (the nybble format for octaves and notes will do) and some lengths and you have the basics already. You could even use more than one channel for different effects.

If you understood that, then get this: the note+length+channel idea is pretty much what tracked music (mod, it, xm, etc) does, only they use a more sophisticated wave

than a square wave. But the principle is the same. Getting it to work takes a little more effort, but that's what Deku's sound mix tutorial is for.

19. Text systems

> **DEPRECATION NOTICE**
>
> This chapter has been superceded by TTE. Information from this chapter can still be useful, but for serious work, TTE should be preferred.

Introduction

```c
#include <stdio.h>

int main()
{
    printf("Hello World");
    return 0;
}
```

Aaah, yes, "Hello world": the canonical first example for every C course and system. Except for consoles. While printing text on a PC is the easiest thing in the world, it is actually a little tricky on a console. It's not that there's no `printf()` function, but rather that there is nowhere for it to write to or even a font to write with (and that's hardly the full list of things to consider). Nope, if you want to be able to display text,

you'll have to build the whole thing from scratch yourself. And you do want to be able to write text to the screen,

So, what do we need for a text system? Well, that's actually not a simple question. Obviously, you need a font. Just a bitmap with the various characters here, no need to depress ourselves with vector-based fonts on a GBA. Second, you need a way of taking specific characters and show them on the screen.

But wait a minute, which video mode are we using? There's tilemaps, bitmap modes and sprites to choose from, all of which need to be dealt with in entirely different ways. Are we settling for one of them, or create something usable for all? Also, what is the font we're using, and what are the character sizes? Fixed width or variable width? Variable width and sizes aren't much of a problem for the bitmap modes, but it's a bitch to splice them for tiles. Also, just for tiles, do we keep the full font in VRAM? If so, that's a lot of tiles, especially considering you'll hardly be using all of them at the same time. It would be more VRAM efficient to only copy in the glyphs that you're using at that time. This will take some management, though.

Just with these items, you'd have enough options for over 20 different text system implementations, all incompatible in very subtle ways. At the very least you'll need `putc()` and `puts()` for each. And then perhaps a `printf()` -like function too; for each text-type, mind you, because glyph placement goes on the inside. Maybe a screen clear too; or how about scrolling functionality. Well, you get the idea.

I suppose it's possible to create a big, complicated system, tailoring to every need anyone could possibly have. But I'm not going to. Firstly, because it's a bit waste of time: the chances you'll need the ability to run, say, bitmap and tilemap modes concurrently are virtually –if not actually– nil. Most of the time, you'll use a single video mode and stick to that. Spending time (and space) for allow every variation imaginable, when hardly any will ever be used is probably not worth the trouble. Besides, writing tons of code that is almost identical except for some small detail in the heart of the routine is just plain bleh.

The point of this chapter is to show how to build and use a set of simple, lightweight text writers. Don't expect the mother of all text systems, I'm mainly interested in getting the essential thing done, namely getting the characters of a string on the screen. This is a core text system, with the following features:

- Bitmap (mode 3, 4, 5), regular tilemap (mode 0, 1) and sprite support.
- There will be a `xxx_puts()` for showing the string, and a `xxx_clrs()` to wipe it. Their arguments will a string, the position to plot to, and some color information. If you want scrolling and/or format specifiers, I'll leave that up to you.
- The font is a fixed width, monochrome font with one 8x8 tile per character. The glyphs can be smaller than 8x8, and I'll even leave in hooks that allow variable widths, but things just get horrible if I'd allowed for multi-tile fonts.
- A variable character map. This is a great feature if you plan on using only a small set of characters, or non-ascii glyph orders.

This arrangement allows for the most basic cases and allows for some variations in set-up, but very little on the side. However, those extras would probably be very game specific anyway, and might be ill suited for a general text system. If you want extras, it shouldn't be too hard to write them yourself.

NO PRINTF(). O RLY?

I said that there is no `printf()` on the GBA, but this isn't quite true; not anymore, anyway. It is possible to hook your own IO-system to the standard IO-routines, which is done in `libgba` .

SEMI-OBSOLETE

I have another text system here that is much more powerful (as in, really working on every video mode and has a printf too) than what's described in this page. However, it's rather large, not completely finished and it would take some time to write the description page and alter the text to fit the demos again. A

libtonc version that has the relevant changes can be found at
http://www.coranac.com/files/misc/tonclib-1.3b.rar.

Text system internals

Variables

For keeping track of the text-system's state, we'll need a couple of variables. The obvious variables are a font and a character map. Because I like to keep things flexible, I'll also use two pointers for these so that you can use your own font and char-map if you want. You also need to know where it is you want to write to, which is done via a base-destination pointer. As extras, I'll also have character size variables for variable glyph spacing, and even a pointer to a char-width array, for a possible variable-width font.

I'll use a struct to store these, partially because it's easier for me to maintain, but also because the CPU and compiler can deal with them more efficiently. I'll also leave a few bytes empty for any eventual expansion. Finally, an instance of this struct, and a pointer to it so you can switch between different systems if you ever need to (which is unlikely, but still). Yes, I am wasting a few bytes, but if you max out IWRAM over this, I dare say you have bigger problems to worry about.

```
// In text.h
typedef struct tagTXT_BASE
{
    u16 *dst0;      // writing buffer starting point
    u32 *font;      // pointer to font used
     u8 *chars;     // character map (chars as in letters, not tiles)
     u8 *cws;       // char widths (for VWF)
     u8  dx,dy;     // letter distances
    u16  flags;     // for later
     u8  extra[12]; // ditto
} TXT_BASE;

extern TXT_BASE __txt_base, *gptxt;

// In text.c
TXT_BASE __txt_base;                    Main TXT_BASE instance
TXT_BASE *gptxt= &__txt_base;            and a pointer to it
```

The font
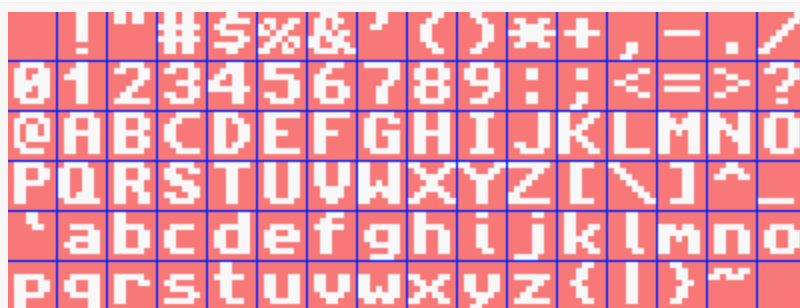
Fig 19.1: Default tonc font: mini-ascii, monochrome, 8x8 pixels per glyph.

fig 19.1 shows the font I'll be using. This particular font is monochrome and each of the glyphs fits into an 8x8 box. The 96 glyphs themselves a subset of the full ASCII that I'll refer to as *mini-ascii*. It's the lower ascii half that contains the majority of the standard ASCII table, but leaves out ASCII 0-31 because they're escape codes and not really part of the printable characters anyway.

It is possible to use a different font with another glyph order, but the functions I'll present below rely on one tile per glyph, *and* in tile layout. I need this arrangement because I intend to use it for all modes, and non single-tile formats would be hell in tile modes.

Another restriction is that the font must be bitpacked to 1bpp. I have a couple of reasons for this. First, there is the size consideration. A 96 glyph, 16bit font (for modes 3/5) would take up 12kB. Pack that to 1bpp and it's less that one kB! Yes, you're restricted to monochrome, but for a font, that's really not much of a problem. Often fonts are monochrome anyway and using 16 bits where you only need one seems a bit of a waste. Secondly, how would you get a 16bpp font to work for 4bpp or 8bpp tiles? Going from a low bpp to a higher one is just a lot easier. Of course, if you don't like this arrangement, feel free to write your own functions.

As for the font data itself, here is the whole thing.

```c
const unsigned int toncfontTiles[192]=
{
    0x00000000, 0x00000000, 0x18181818, 0x00180018, 0x00003636,
0x00000000, 0x367F3636, 0x0036367F,
    0x3C067C18, 0x00183E60, 0x1B356600, 0x0033566C, 0x6E16361C,
0x00DE733B, 0x000C1818, 0x00000000,
    0x0C0C1830, 0x0030180C, 0x3030180C, 0x000C1830, 0xFF3C6600,
0x0000663C, 0x7E181800, 0x00001818,
    0x00000000, 0x0C181800, 0x7E000000, 0x00000000, 0x00000000,
0x00181800, 0x183060C0, 0x0003060C,
    0x7E76663C, 0x003C666E, 0x181E1C18, 0x00181818, 0x3060663C,
0x007E0C18, 0x3860663C, 0x003C6660,
    0x33363C38, 0x0030307F, 0x603E067E, 0x003C6660, 0x3E060C38,
0x003C6666, 0x3060607E, 0x00181818,
    0x3C66663C, 0x003C6666, 0x7C66663C, 0x001C3060, 0x00181800,
0x00181800, 0x00181800, 0x0C181800,
    0x06186000, 0x00006018, 0x007E0000, 0x0000007E, 0x60180600,
0x00000618, 0x3060663C, 0x00180018,

    0x5A5A663C, 0x003C067A, 0x7E66663C, 0x00666666, 0x3E66663E,
0x003E6666, 0x06060C78, 0x00780C06,
    0x6666361E, 0x001E3666, 0x1E06067E, 0x007E0606, 0x1E06067E,
0x00060606, 0x7606663C, 0x007C6666,
    0x7E666666, 0x00666666, 0x1818183C, 0x003C1818, 0x60606060,
0x003C6660, 0x0F1B3363, 0x0063331B,
    0x06060606, 0x007E0606, 0x6B7F7763, 0x00636363, 0x7B6F6763,
0x00636373, 0x6666663C, 0x003C6666,
    0x3E66663E, 0x00060606, 0x3333331E, 0x007E3B33, 0x3E66663E,
0x00666636, 0x3C0E663C, 0x003C6670,
    0x1818187E, 0x00181818, 0x66666666, 0x003C6666, 0x66666666,
0x00183C3C, 0x6B636363, 0x0063777F,
    0x183C66C3, 0x00C3663C, 0x183C66C3, 0x00181818, 0x0C18307F,
0x007F0306, 0x0C0C0C3C, 0x003C0C0C,
    0x180C0603, 0x00C06030, 0x3030303C, 0x003C3030, 0x00663C18,
0x00000000, 0x00000000, 0x003F0000,

    0x00301818, 0x00000000, 0x603C0000, 0x007C667C, 0x663E0606,
0x003E6666, 0x063C0000, 0x003C0606,
    0x667C6060, 0x007C6666, 0x663C0000, 0x003C067E, 0x0C3E0C38,
0x000C0C0C, 0x667C0000, 0x3C607C66,
    0x663E0606, 0x00666666, 0x18180018, 0x00301818, 0x30300030,
0x1E303030, 0x36660606, 0x0066361E,
    0x18181818, 0x00301818, 0x7F370000, 0x0063636B, 0x663E0000,
0x00666666, 0x663C0000, 0x003C6666,
    0x663E0000, 0x06063E66, 0x667C0000, 0x60607C66, 0x663E0000,
0x00060606, 0x063C0000, 0x003E603C,
    0x0C3E0C0C, 0x00380C0C, 0x66660000, 0x007C6666, 0x66660000,
0x00183C66, 0x63630000, 0x00367F6B,
    0x36630000, 0x0063361C, 0x66660000, 0x0C183C66, 0x307E0000,
0x007E0C18, 0x0C181830, 0x00301818,
```

```
    0x18181818, 0x00181818, 0x3018180C, 0x000C1818, 0x003B6E00,
 0x00000000, 0x00000000, 0x00000000,
 };
```

Yes, this is the *entire* font, fitting nicely on one single page. This is what bitpacking can do for you but, like any compression method, it may be a little tricky seeing that it is indeed the font given earlier, so here's a little explanation of what you got in front of you.

Bitpacking

Bitpacking isn't hard to understand. Data is little more a big field of bits. In bitpacking, you simply drop bits at regular intervals and tie the rest back together. Our font is monochrome, meaning we only have one bit of information. Now, even in the smallest C datatype, bytes, this would leave 7 bits unused if you were to use one byte per pixel. However, you could also cram eight pixels into one byte, and thus save a factor 8 in space. For the record, that's a compression level of 88%, pretty good I'd say. Of course, if you read all the other pages already, you'd have already recognized instances of bitpacking: 4bpp tiles are bitpacked with 2 pixels/byte. So this stuff shouldn't be completely new.

big u32		0x01020304		
big u16	0x0102		0x0304	
u8	0x01	0x02	0x03	0x04
little u16	0x0201		0x0403	
little u32		0x04030201		

table 19.1: Big endian vs little endian interpretation of byte-sequence 01h, 02h, 03h, 04h

Bitpacking can save a lot of room, and in principle, it's easy to do, as it's just a matter of masking and shifting. There is one major catch, however: *endianness*. You already seen one incarnation of this in other data-arrays: the word `0x01234567` would actually be stored as the byte-sequence `0x67`, `0x45`, `0x23`, `0x01` on ARM (and intel) systems. This is called *little-endian*, because the little end (the lower bytes of a multi-byte type) of the word are stored in the lower addresses. There is also *big-endian*, which stores the most significant bytes first. You can see the differences in table 19.1. Some hex editors or memory viewers (in VBA for example) allow you to switch viewing data as bytes, halfwords or words, so you can see the differences interactively there.

Please remember that the data itself does *not* change because of this, you just *look* at it in a different way.

For bitpacking, you also have to deal with endianness at the bit level. The font data is packed in a consistent bit-little and byte-little format for three reasons. First, this is how GBA bitpacked stuff works anyway, so you can use the BIOS BitUnpack routine for it. Second, it is a more natural form in terms of counting: lower bits come first. Third, because you can shift down all the time and discard covered bits that way, masking is easier and faster. Now, big-endian would be more natural visually due to the fact we write numbers that way too, so bitmaps are often bit-little as well. Windows BMP files, for example, these have their leftmost pixels in the most significant bits, making them bit-big. However, Windows runs on Intel architecture, which is actually *byte* little-endian, for maximum confusion. Sigh. Oh well.

In case it's still a bit hazy, fig 19.2 shows how the 'F' is packed from 8x8 pixels into 2 words. All 64 pixels are numbered 0 to 63. These correspond to the bit-numbers. Each eight successive bits form a byte: 0-7 make up byte 0, 8-15 form byte 1, etc. Note how the bits seem to mirror horizontally, because we generally write numbers big-endian. So try to forget about that and think of bits in memory to walk through from 0 to 63. You can also view the bits as words, bits 0-31 for word 0 and 32-63 for word 1.
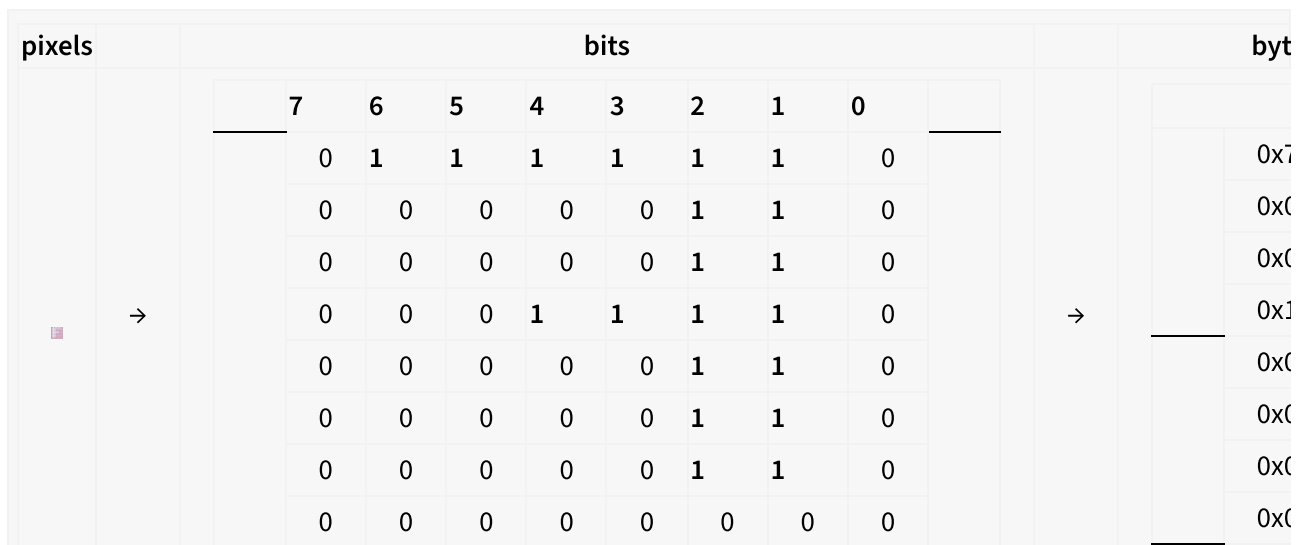
pixels		bits									byt
		7	6	5	4	3	2	1	0		
		0	1	1	1	1	1	1	0		0x7
		0	0	0	0	0	1	1	0		0x0
		0	0	0	0	0	1	1	0		0x0
→		0	0	0	1	1	1	1	0	→	0x1
		0	0	0	0	0	1	1	0		0x0
		0	0	0	0	0	1	1	0		0x0
		0	0	0	0	0	1	1	0		0x0
		0	0	0	0	0	0	0	0		0x0

fig 19.2: 'F', from 8x8 tile to 1bpp bit-little, byte-little words.

Character map

Having the mini-ascii font is nice and all but as strings are full-ascii, this may present a problem. Well, not really, but there are several ways of going about the conversion.

First, you can create a giant switch-block that converts, say, 'A' (ascii 65) into glyph-index 33. And do that for all 96 glyphs. It should be obvious that this is a dreadful way of going about things. Well it *should*, but apparently it's not because code like that is out there; I only mention it here so you can recognize it for what it is and stay to far, far away from it. Simply put, if you have a switch-block where the only difference between the cases is returning a different offset –and a *fixed* offset at that– you're doing something very, very wrong.

A second method which is an enormous improvement in every way is to simply subtract 32. That's how mini-ascii was defined after all. Quick, short, and to the point.

However, I kinda like the third option: look-up tables. We've already seen how useful LUTs can be for mathematics, but you can use them for a lot more than that. In this case, the lut is a **charcter map**, containing the glyph-index for each ascii character. This has almost all the benefits of the simple subtract (a look-up may be a few cycles slower), but is much more flexible. For example, you can have non-ascii charmaps or alias the cases, things like that. Another 'interesting' thing is that you don't really need the font to be text as such, it can be any kind of mapped image data; with a lut you could easily use the text system for drawing borders, as long as you have a border 'font' for it. The lut I'm using is 256 bytes long. This may not be enough for Unicode (sorry Eastern dudes), but it's enough to suit my purposes.

General design

The first thing to do code-wise is to initialize the members of the text-base. That means attach the font, set the glyph sizes, and initialize the lut. This can be done with `txt_init_std()`.

```
u8 txt_lut[256];

// Basc initializer for text state
void txt_init_std()
{
    gptxt->dx= gptxt->dy= 8;

    gptxt->dst0= vid_mem;
    gptxt->font= (u32*)toncfontTiles;
    gptxt->chars= txt_lut;
    gptxt->cws= NULL;

    int ii;
    for(ii=0; ii<96; ii++)
        gptxt->chars[ii+32]= ii;
}
```

Depending on the type of text, you may need more specialized initializers, which we'll get to when the time comes. As for writing a string, the basic structure can bee seen below. It's actually quite simple and very general, but unfortunately the fact that `xxx_putc()` is in the inner loop means that you have to have virtually identical wrappers around each char-plotter for each text method. I also have functions called `xxx_clrs()` that clear the string from the screen (they don't wipe the whole screen). They are almost identical to their `puts()` siblings in form and also rather simple, so I won't elaborate on them here.

```
// Pseudo code for xxx_puts
void xxx_puts(int x, int y, const char *str, [[more]])
{
    [[find real writing start]]
    while(c=*str++)      // iterate through string
    {
        switch(c)
        {
        case [[special chars ('\n' etc)]]:
            [[handle special]]
        case [[normal chars]]:
            [[xxx_putc(destination pointer, lut[c])]]
            [[advance destination]]
        }
    }
}
```

Bitmap text

Bitmap text concerns modes 3, 4 and 5. If you can do mode 3, you pretty much have mode 5 as well, as the two differ only by the pitch and, perhaps, the starting point. Mode 4 is different, not only because it's 8bpp, but also because this means we have to do 2 pixels at once.

Internal routines

I tend to do bitmap related functions in two parts: there are internal 16bit and 8bit functions that take an address and pitch as their arguments, and then inline interface functions with coordinates that call these. The internal 16bit writer is given in below, with an explanation of the main parts below that.

```c
void bm16_puts(u16 *dst, const char *str, COLOR clr, int pitch)
{
    int c, x=0;

    while((c=*str++) != 0)        // (1) for each char in string
    {
        // (2) real char/control char switch
        if(c == '\n')          // line break
        {
            dst += pitch*gptxt->dy;
            x=0;
        }
        else                       // normal character
        {
            int ix, iy;
            u32 row;
            // (3) point to glyph; each row is one byte
            u8 *pch= (u8*)&gptxt->font[2*gptxt->chars[c]];
            for(iy=0; iy<8; iy++)
            {
                row= pch[iy];
                // (4) plot pixels until row-byte is empty
                for(ix=x; row>0; row >>= 1, ix++)
                    if(row&1)
                        dst[iy*pitch+ix]= clr;
            }
            x += gptxt->dx;
        }
    }
}
```

1. Traditional way to loop through all characters in a string. `c` will be the character we have to deal with, unless it's the delimiter (`'\0'`), then we'll stop.
2. Normal char/control char switch. Control characters like `'\n'` and `'\t'` should be taken care of separately. I'm only checking for the newline right now, but others could easily be added.
3. This is where it gets interesting. What this line does is first use the lut to look up the glyph index in the font, look up the actual glyph in the font (multiply by 2 because there are 2 words/glyph), and then set-up a byte-pointer `pch` to point to the glyph.
   A couple of things come together here. First, because all glyphs are exactly 8 bytes apart, finding the glyph data is very easy. If you create your own text

system with your own fonts, I'd advise using constant offsets, even if it wastes pixels like you would for small characters like 'I'. Second, because of the 1bpp tiled format, each row is exactly one byte long, and all the glyphs bits are in consecutive bytes, so you don't have to jump around for each new row. This is a good thing.

4. The `ix` loop is even more interesting. First, we read the actual row of pixels into the (word) variable `row`. To test whether we need to write a pixel, we simply check for a given bit. However, because the packing is *little* endian, this allows for two shortcuts.

   The first one is that looping through the bits goes from low to high bits, meaning that we can simply shift-right on each iteration and test bit 0. The corollary to this is that the bits we've already done are thrown away, and *this* means that when `row` is 0, there will be no more pixels, and we're done for that row. As this short-circuit happens inside the inner of a *triple* loop, the speed-up can be substantial.

This function only does the bare essentials to get a string on screen. It plots the non-zero pixels only (transparent characters), there is no wrapping at the side and no scrolling. The only non-trivial feature is that it can do line-breaks. When those happen, the cursor returns to the original x-position on screen.

The 8bit function is almost identical to this one, 'almost' because of the no-byte-write rule for VRAM. The obvious ones are that the pitch and character spacing need to be halved. I'm also making it **requirement** that the start of each character needs to be on an even pixel boundary. By doing so, you can have an almost identical inner loop as before; it just does two pixels in it instead of one. Yes, it's a hack; no, I don't care.

```
void bm8_puts(u16 *dst, const char *str, u8 clrid)
{
    int c, x=0, dx= gptxt->dx >> 1;

    while((c=*str++) != 0)
    {
        // <snip char-switch and iy loop>
                for(ix=x; row>0; row >>= 2, ix++)
                {
                    pxs= dst[iy*120+ix];
                    if(row&1)
                        pxs= (pxs&0xFF00) | clrid;
                    if(row&2)
                        pxs= (pxs&0x00FF) | (clrid<<8);

                    dst[iy*120+ix]= pxs;
                }
        // <snip>
    }
}
```

Interface functions

The interface functions are straightforward. All they have to do is set-up the destination start for the internal routines, and for the 16bit versions, provide a pitch. Mode 3 uses `vid_mem` as its base, and mode 4 and 5 use `vid_page` to make sure it works with page flipping. `m4_puts()` also ensures that the characters start at even pixels, and please remember that this routine uses a color-index, rather than a true color.

```
// Bitmap text interface. Goes in text.h
INLINE void m3_puts(int x, int y, const char *str, COLOR clr)
{   bm16_puts(&vid_mem[y*240+x], str, clr, 240);      }

INLINE void m4_puts(int x, int y, const char *str, u8 clrid)
{   bm8_puts(&vid_page[(y*240+x)>>1], str, clrid);    }

INLINE void m5_puts(int x, int y, const char *str, COLOR clr)
{   bm16_puts(&vid_page[y*160+x], str, clr, 160);     }
```

Clearing text

Doing a text clear is almost the same as writing out a string. The only functional difference is that you're always putting a space (or rather, a solid filled rectangle) instead of the original characters. You still need the full string you tell you how long the line goes on, and how many lines there are.

With that in mind, the `bm16_clrs()` function below shouldn't be that hard to understand. The whole point of it is to read the string to find out the length in pixels of each line in the string (`nx*gptxt->dx`), then fill the rectangle spanned by that length and the height of the characters (`gptxt->dy`). There's some bookkeeping to make sure it all goes according to plan, but in the end that's all it does. The same goes for the clear routines of the other text-types, so I'm not going to show those.

```c
void bm16_clrs(u16 *dst, const char *str, COLOR clr, int pitch)
{
    int c, nx=0, ny;

    while(1)
    {
        c= *str++;
        if(c=='\n' || c=='\0')
        {
            if(nx>0)
            {
                nx *= gptxt->dx;
                ny= gptxt->dy;
                while(ny--)
                {
                    memset16(dst, clr, nx);
                    dst += pitch;
                }
                nx=0;
            }
            else
                dst += gptxt->dy*pitch;
            if(c=='\0')
                return;
        }
        else
            nx++;
    }
}
```

Tilemap text

In some ways, text for tile-modes is actually easier than for bitmaps, as you can just stuff the font into a charblock and then you don't need any reference to the font itself anymore. That is, unless you want to have a variable width font, in that case you'll be in bit-shifting hell. But I'm sticking to a fixed width, single tile font, which keeps things very simple indeed.

Tile initialisation

The first order of business is to be able to unpack the font to either 4 or 8 bit. The easiest way of doing this is to just setup a call to `BitUnpack()` and be done with it. However, VBA's implementation of it isn't (or wasn't, they may have fixed it by now) quite correct for what I had planned for it, so I'm going to roll my own. Arguments `dstv` and `srcv` are the source and destination addresses, respectively; `len` is the number of source bytes and `bpp` is the destination bitdepth. `base` serves two purposes. Primarily, it is a number to be added to all the pixels if bit 31 is set, or to all except zero values if it is clear. This allows a greater range of outcomes than just the 0 and 1 that a source bitdepth of one would supply; and an other cute trick that I'll get to later.

```
// Note, the BIOS BitUnpack does exactly the same thing!
void txt_bup_1toX(void *dstv, const void *srcv, u32 len, int bpp, u32
base)
{
    u32 *src= (u32*)srcv;
    u32 *dst= (u32*)dstv;

    len= (len*bpp+3)>>2;      // # dst words
    u32 bBase0= base&(1<<31);     // add to 0 too?
    base &= ~(1<<31);

    u32 swd, ssh=32;      // src data and shift
    u32 dwd, dsh;         // dst data and shift
    while(len--)
    {
        if(ssh >= 32)
        {
            swd= *src++;
            ssh= 0;
        }
        dwd=0;
        for(dsh=0; dsh<32; dsh += bpp)
        {
            u32 wd= swd&1;
            if(wd || bBase0)
                wd += base;
            dwd |= wd<<dsh;
            swd >>= 1;
            ssh++;
        }
        *dst++= dwd;
    }
}
```

The actual map-text initialization is done by `txt_init_se()`. Its first two arguments are exactly what you'd expect: the background that the system should use for text and the control-flags that should go there (charblock, screenblock, bitdepth, all that jazz). The third argument, `se0`, indicates the 'base' for palette and tile indexing, similar to the base for unpacking. The format is just like normal screen entries: `se0` {0-9} indicate the tile offset, and `se0` {C-F} are for the 16 color palette bank. `clrs` contains the color for the text, which will go into the palette indicated by the sub-palette and the fifth argument, `base`, the base for bit-unpacking.

For now, ignore the *second* color in `clrs` , and the extra palette write for 4 bpp. In all likelihood, you don't want to know. I'm going to tell you about them later anyway, though.

```c
void txt_init_se(int bgnr, u16 bgcnt, SB_ENTRY se0, u32 clrs, u32 base)
{
    bg_cnt_mem[bgnr]= bgcnt;
    gptxt->dst0= se_mem[BF_GET(bgcnt, BG_SBB)];

    // prep palette
    int bpp= (bgcnt&BG_8BPP) ? 8 : 4;
    if(bpp == 4)
    {
        COLOR *palbank= &pal_bg_mem[BF_GET(se0, SE_PALBANK)<<4];
        palbank[(base+1)&15]= clrs&0xFFFF;
        palbank[(base>>4)&15]= clrs>>16;
    }
    else
        pal_bg_mem[(base+1)&255]= clrs&0xFFFF;

    // account for tile-size difference
    se0 &= SE_ID_MASK;
    if(bpp == 8)
        se0 *= 2;

    // Bitunpack the tiles
    txt_bup_1toX(&tile_mem[BF_GET(bgcnt, BG_CBB)][se0],
        toncfontTiles, toncfontTilesLen, bpp, base);
}
```

If you don't want to deal with all kinds of offsets, you can just leave the third and fifth arguments zero. It's probably not a good idea to leave the others zero, but for those two it's not a problem.

Screen entry writer

This is arguably the most simple of the text writers. As there is one glyph per screen entry, all you have to do is write a single halfword to the screenblock in the right position and you have a letter. Repeat this for a whole string.

There are a few things to note about this implementation, though. First, like before, no kind of wrapping or scrolling. If you want that, you'll have to do all that yourself.

Also, the *x* and *y* coordinates are still in *pixels*, not tiles. I've done this mainly for consistency with the other writers, nothing more. Oh, in case you hadn't noticed before, `gptxt->dst0` is initialized to point to the start of the background's screenblock in `txt_init_se()`. Lastly, `se0` is added to make up the actual screen entry; if you had a non-zero `se0` in initialization, chances are you'd want to use it here too.

```c
void se_puts(int x, int y, const char *str, SB_ENTRY se0)
{
    int c;
    SB_ENTRY *dst= &gptxt->dst0[(y>>3)*32+(x>>3)];

    x=0;
    while((c=*str++) != 0)
    {
        if(c == '\n')     // line break
        {    dst += (x&~31) + 32;     x=0;     }
        else
            dst[x++] = (gptxt->chars[c]) + se0;
    }
}
```

Sprite text

Sprite text is similar to tilemap text, only you use OBJ_ATTRs now instead of screen entries. You have to set the position manually (attributes 0 and 1), and attribute 2 is almost the same as the screen entry for regular tilemaps. The initializer `txt_init_obj()` is similar to `txt_init_se()`, except that the tilemap details have been replaced by their OAM counterparts. Instead of a screenblock, we point to a base OBJ_ATTR `oe0`, and `attr2` works in much the same way as `se0` did. The code is actually simpler because we can always use 4bpp tiles for the objects that we use, without upsetting the others.

```
// OAM text initializer
void txt_init_obj(OBJ_ATTR *oe0, u16 attr2, u32 clrs, u32 base)
{
    gptxt->dst0= (u16*)oe0;

    COLOR *pbank= &pal_obj_mem[BF_GET(attr2, ATTR2_PALBANK)<<4];
    pbank[(base+1)&15]= clrs&0xFFFF;
    pbank[(base>>4)&15]= clrs>>16;

    txt_bup_1toX(&tile_mem[4][attr2&ATTR2_ID_MASK], toncfontTiles,
        toncfontTilesLen, 4, base);
}
```

```
// OAM text writer
void obj_puts(int x, int y, const char *str, u16 attr2)
{
    int c, x0= x;
    OBJ_ATTR *oe= (OBJ_ATTR*)gptxt->dst0;

    while((c=*str++) != 0)
    {
        if(c == '\n')     // line break
        {    y += gptxt->dy; x= x0; }
        else
        {
            if(c != ' ') // Only act on a non-space
            {
                oe->attr0= y & ATTR0_Y_MASK;
                oe->attr1= x & ATTR1_X_MASK;
                oe->attr2= gptxt->chars[c] + attr2;
                oe++;
            }
            x += gptxt->dx;
        }
    }
}
```

The structure of the writer itself should feel familiar now. The `attr2` again acts as a base offset to allow palette swapping and an offset tile start. Note that I'm only entering the position in attributes 0 and 1, and nothing else. I can do this because the rest of the things are already set to what I want, namely, 8x8p sprites with 4bpp tiles and no frills. Yes, this may screw things up for some, but if I *did* mask out everything

properly, it'd screw up other stuff. This is a judgement call, feel free to disagree and change it.

That writer always starts at a fixed OBJ_ATTR, overwriting any previous ones. Because that might be undesirable, I also have a secondary sprite writer, `obj_puts2`, which takes an OBJ_ATTR as an argument to serve as the new base.

```
INLINE void obj_puts2(int x, int y, const char *str, u16 attr2, OBJ_ATTR
*oe0)
{
    gptxt->dst0= (u16*)oe0;
    obj_puts(x, y, str, attr2);
}
```

There are some side notes on memory use that I should mention. Remember, there are only 128 OBJ*ATTRs, and at one entry/glyph it may become prohibitively expensive if used extensively. In the same vein, 1024 tiles may seem like a lot, but you can run out quickly if you have a couple of complete animations in there as well. Also, remember that you only have 512 tiles in the bitmap modes: a full ASCII character set in bitmap modes would take up _half* the sprite tiles!

If you're just using it to for a couple of characters you're not likely to run into trouble, but if you want screens full of text, you might be better of with something else. There are ways to get around these things, of course; quite simple ways, even. But because they're really game-specific, it's difficult to give a general solution for it.

Some demos

Bitmap text demo

I suppose I could start with "Hello world", but as that's pretty boring I thought I'd start with something more interesting. The `txt_bm` demo does something similar to `bm_modes`: namely show something on screen and allow switching between modes 3,

4 and 5 to see what the differences are. Only now, we're going to use the bitmap `puts()` versions to write the actual strings indicating the current mode. Because that's still pretty boring, I'm also going to put a movable cursor on screen and write out its coordinates. Here's the full code:

```c
#include <stdio.h>
#include <tonc.h>

#define CLR_BD      0x080F

const TILE cursorTile=
{{  0x0, 0x21, 0x211, 0x2111, 0x21111, 0x2100, 0x1100, 0x21000  }};

void base_init()
{
    vid_page= vid_mem;

    // init interrupts
    irq_init(NULL);
    irq_add(II_VBLANK, NULL);

    // init backdrop
    pal_bg_mem[0]= CLR_MAG;
    pal_bg_mem[CLR_BD>>8]= CLR_BD;
    pal_bg_mem[CLR_BD&255]= CLR_BD;
    m3_fill(CLR_BD);

    // init mode 4 pal
    pal_bg_mem[1]= CLR_LIME;
    pal_bg_mem[255]= CLR_WHITE;

    // init cursor
    tile_mem[5][0]= cursorTile;
    pal_obj_mem[1]= CLR_WHITE;
    pal_obj_mem[2]= CLR_GRAY;
}

int main()
{
    base_init();

    txt_init_std();

    // (1) print some string so we know what mode we're at
    m3_puts( 8,  8, "mode 3", CLR_CYAN);
    m4_puts(12, 32, "mode 4", 1);
    m5_puts(16, 40, "mode 5", CLR_YELLOW);

    // init variables
    u32 mode=3, bClear=0;
    OBJ_ATTR cursor= { 80, 120, 512, 0 };

    // init video mode
    REG_DISPCNT= DCNT_BG2 | DCNT_OBJ | 3;
```

```
    // init cursor string
    char str[32];
    siprintf(str, "o %3d,%3d", cursor.attr1, cursor.attr0);

    while(1)
    {
        VBlankIntrWait();
        oam_mem[0]= cursor;
        key_poll();

        if(key_hit(KEY_START))
            bClear ^= 1;

        // move cursor
        cursor.attr1 += key_tri_horz();
        cursor.attr0 += key_tri_vert();

        // adjust cursor(-string) only if necessary
        if(key_is_down(KEY_ANY))
        {
            // (2) clear previous coords
            if(bClear)
                bm_clrs(80, 112, str, CLR_BD);

            cursor.attr0 &= ATTR0_Y_MASK;
            cursor.attr1 &= ATTR1_X_MASK;
            // (3) update cursor string
            siprintf(str, "%c %3d,%3d", (bClear ? 'c' : 'o'),
                cursor.attr1, cursor.attr0);
        }

        // switch modes
        if(key_hit(KEY_L) && mode>3)
            mode--;
        else if(key_hit(KEY_R) && mode<5)
            mode++;
        REG_DISPCNT &= ~DCNT_MODE_MASK;
        REG_DISPCNT |= mode;

        // (4) write coords
        bm_puts(80, 112, str, CLR_WHITE);
    }

    return 0;
}
```

Controls:

D-pad	Moves cursor.
Start	Toggles string clearing.
L, R	Decrease or increase mode.

Many things here should be either self explanatory or fairly irrelevant. The interesting things are indicated by numbers, so let's go through them, shall we?

fig 19.3: `txt_bm` demo.

1. Mode indicators. This is where we write three strings to VRAM, indicating the modes. Note that the interfaces are nearly identical; the only real difference is that the fourth argument for `m4_puts()` is a palette index, rather than a real color.

2. Clear previous cursor-string. The cursor string keeps track of the cursor as you move across the screen. The first thing you'll notice is that the string turns into a horrible mess because the bitmap writers only write the *non-zero* pixels of the font. In other words, it does *not* clear out the rest of the space allotted for that glyph. Essentially `mx_puts()` are transparent string writers.

Sure, I could have added a switch that would erase the whole glyph field to the writers. Quite easily, actually, it only takes an extra `else` clause. However, the current way is actually more practical. For one thing, what if you actually *want* transparency? You'd have to write another routine just for that. The method I've chosen is to have an extra clearing routine (which you'd probably need anyway). To overwrite the whole glyphs, simply call `mx_clrs()` first; which is what I'm doing here. Well, as long as the `bClear` variable is set (toggle with Start).

A second reason is that this method is just so much faster. Not only because I wouldn't be able to use my premature breaking from the `ix` -loop if I had to erase the whole field and the mere presence of an extra branch adds more cycles (inside a triple loop), but plotting individual characters will always be slower than to do it by whole blocks at a time. `mx_clrs()` uses `memset16()`, which is basically `CpuFastSet()` plus safeties, and will be faster after just a mere half a dozen pixels.

Oh, in case you're wondering why I'm talking about `mx_clrs()` when the code mentions `bm_clrs()`, the latter function is merely a function that uses a switch-block with the current bitmap mode to call the correct mode-specific string clearer.

3. Updating the cursor string. As the writers don't have format specification fields, how can we write numbers? Simple, use `sprintf()` to prepare a string first, and then use that one instead. Or rather, use `siprintf()`. This is an integer-only version of `sprintf()`, which is better suited to GBA programming since you're not supposed to use floating point numbers anyway. It should be relatively simple to create functions to wrap around `siprintf()` and `mx_puts()`, but I'm not sure it's worth the effort.

I should perhaps point out that using `siprintf` and other routines that can turn numbers into strings use division by 10 to do so, and you know what that means. And even if you do not ask it to convert numbers, it calls a dozen or so routines from the standard library, which adds around 25kb to your binary. This isn't much for ROM, but for multiboot things (256kb max) it may become problematic. With that in mind, I'd like you to take a look at **posprintf** by Dan Posluns. This is hand-coded assembly using a special algorithm for the decimal conversion. It may not be as rich in options as `siprintf()`, but it's both faster and smaller by a very large margin, so definitely worth checking out.

4. Write cursor string. This writes the current cursor string to position (80, 120). Like in the cases of wiping the string, I'm using a `bm_puts()` function that switches between the current mode writers.

Sprite text; Hello world!

Yes! Hello world! Now, in principle, all you have to do is call `txt_init()`, `txt_init_obj()` and then `obj_puts()` with the right parameters, but again that's just boring, so I'll add some interesting things as well. The `txt_obj` demo shows one of the things best performed with sprites: individual letter animation. The letters of the phrase "hello world!" will fall from the top of the screen, bouncing to a halt on the floor (a green line halfway across the screen).

```c
#include <tonc.h>

// === CONSTANTS & STRUCTS =========================================

#define POS0 (80<<8)
#define GRAV 0x40
#define DAMP 0xD0
#define HWLEN 12

const char hwstr[]= "Hello world!";

typedef struct
{
    u32 state;
    int tt;
    FIXED fy;
    FIXED fvy;
} PATTERN;

// === FUNCTIONS ===================================================

void pat_bounce(PATTERN *pat)
{
    if(pat->tt <= 0)     // timer's run out: play pattern
    {
        pat->fvy += GRAV;
        pat->fy += pat->fvy;

        // touched floor: bounce
        if(pat->fy > POS0)
        {
            // damp if we still have enough speed
            // otherwise kill movement
            if(pat->fvy > DAMP)
            {
                pat->fy= 2*POS0-pat->fy;
                pat->fvy= DAMP-pat->fvy;
            }
            else
            {
                pat->fy= POS0;
                pat->fvy= 0;
            }
        }
    }
    else    // still in waiting period
        pat->tt--;
}

int main()
```

```
{
    REG_DISPCNT= DCNT_MODE3 | DCNT_BG2 | DCNT_OBJ;

    irq_init(NULL);
    irq_add(II_VBLANK, NULL);
    memset16(&vid_mem[88*240], CLR_GREEN, 240);

    // (1) init sprite text
    txt_init_std();
    txt_init_obj(&oam_mem[0], 0xF200, CLR_YELLOW, 0xEE);
    // (2) 12 px between letters
    gptxt->dx= 12;

    // (3) init sprite letters
    OBJ_ATTR *oe= oam_mem;
    obj_puts2(120-12*HWLEN/2, 8, hwstr, 0xF200, oe);

    int ii;
    PATTERN pats[HWLEN];

    for(ii=0; ii<HWLEN; ii++)
    {
        // init patterns
        pats[ii].state=0;
        pats[ii].tt= 3*ii+1;
        pats[ii].fy= -12<<8;
        pats[ii].fvy= 0;

        // init sprite position
        oe[ii].attr0 &= ~ATTR0_Y_MASK;
        oe[ii].attr0 |= 160;
    }

    while(1)
    {
        VBlankIntrWait();

        for(ii=0; ii<HWLEN; ii++)
        {
            pat_bounce(&pats[ii]);

            oe[ii].attr0 &= ~ATTR0_Y_MASK;
            oe[ii].attr0 |= (pats[ii].fy>>8)& ATTR0_Y_MASK;
        }
    }

    return 0;
}
```

Very little of this code is actually concerned with the string itself, namely the items 1, 2 and 3. There's a call to `txt_init_std()` for the basic initialization and a call to the sprite text initializer, `txt_init_obj()`. The second argument is the base for attribute 2 (if you don't remember what attribute 2 is, see the chapter on sprites again); `0xF200` means I'm using the sub-palette 15 and start the character tiles at tile-index 512 (because

fig 19.4: `txt_obj` demo.

of the bitmap mode). The font color will be yellow, and out at index 255. That's 240 from the pal-bank, `0x0E` =14 from the unpacking and 1 for the actual 1bpp pixels 240+14+1=255. After this call, I'm also setting the horizontal pixel offset to 12 to spread out the letters a little bit. After that, I just call `obj_puts2()` to set up the first few sprites of OAM so that they show "hello world!" centered at the top of the screen.
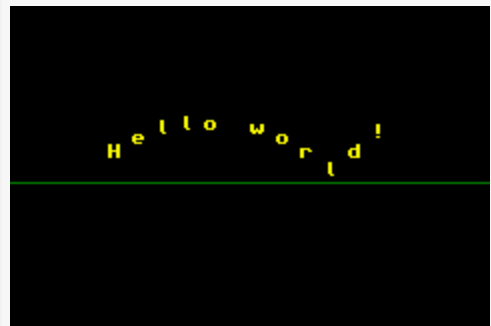
I could have stopped there, but the demo is actually just beginning. The thing about using sprites as glyphs is that they can still *act* as normal sprites; `obj_puts()` just sets them up to use letters instead of graphics that are more sprite-like.

Bouncy, bouncy, bouncy

The goal here is to let the letters drop from the top of the screen, the bounce up again when it hits a floor, but with a little less speed than before due to friction and what not. Physically, the falling part is done using a constant acceleration, g. Acceleration is the change in velocity, so the velocity is linear; velocity is the change in position, so the height is parabolic. At the bounce, we do an ***inelastic collision***; in other words, one where energy is lost. In principle, this would mean that the difference between the squares of the velocities before and after the collision differ by a constant ($|v_{out}|^2$ - $|v_{in}|^2$ = Q). However, this would require a square root to find the new velocity, and I don't care for that right now so I'm just going to scrap the squares here. I'm sure there are situations where this is actually quite valid $\boxed{:P}$. As a further simplification, I'm

doing a first-order integration for the position. With this, the basic code for movement becomes very simple

```
// 1D inelastic reflections
// y, vy, ay: position, velocity, acceleration.
// Q: inelastic collision coefficient.
vy += ay;
 y += vy;
if(y>ymay)  // collision
{
    if((ABS(vy)>Q)
    {
        vy= -(vy-SGN(vy)*Q);  // lower speed, switch direction
        y= 2*ymay-y;          // Mirror y at r: y= r-(y-r)= 2r-y
    }
    else  // too slow: stop at ymay
    {   vy= 0; y= ymay; }
}
```

This could be replaced by the following, more accurate code, using second-order integration and 'proper' recoil, but you hardly notice anything from the improved integration. I actually prefer the look of the simple, linear recoil over the square root though.

```
// accelerate
 k= vx+GRAV;
// Trapezium integration rule:
//    x[i+1]= x[i] + (v[i]+v[i+1])/2;
 x += (vx+k)/2;
vx= k;
if(x>xmax)  // collision
{
    if(vx*vx > Q2)
    {   vx= -Sqrt(vx*vx-Q2); x= 2*xmax-x; }
    else
    {   vx= 0;               x= xmax;     }
}
```

Map text : colors and borders

Next up is the first of two map text demos. The official name for what I call a regular background is "text background", and they're called that for a reason: in most cases

when there is text, it's done using regular backgrounds. Of course, in most cases everything else is *also* done with those, so strictly speaking associating them with "text" is a misnomer, but we'll let that one slide for today. The first demo is about how you can use the text functions for a variety of effects. Apart from simply showing text (boring), you'll see palette swapping and framing text, and how you can easily use different fonts and borders concurrently. Because of the way I've designed my functions, all this takes is a change in a parameter. Cool huh.

The demo will also feature adding shading to a monochrome font, and adding an opaque background for it. Now, the way I'm going about this will probably reserve me a place in the Computer Science Hell, but, well, the coolness of the tricks will probably keep me from burning up there.

```c
#include <tonc.h>
#include "border.h"

// === CONSTANTS & STRUCTS =========================================

#define TID_FRAME0        96
#define TID_FRAME1       105
#define TID_FONT           0
#define TID_FONT2        128
#define TID_FONT3        256
#define TXT_PID_SHADE   0xEE
#define TXT_PID_BG      0x88


// === FUNCTIONS ===================================================

void init()
{
    int ii;
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    irq_init(NULL);
    irq_add(II_VBLANK, NULL);

    txt_init_std();

    // (1a) Basic se text initialization
    txt_init_se(0, BG_CBB(0) | BG_SBB(31), 0x1000, CLR_RED, 0x0E);

    // (1b) again, with a twist
    txt_init_se(0, BG_CBB(0) | BG_SBB(31), 0xF000|TID_FONT2,
        CLR_YELLOW | (CLR_MAG<<16), TXT_PID_SHADE);

    // (1c) and once more, with feeling!
    txt_init_se(0, BG_CBB(0) | BG_SBB(31), 0xE000|TID_FONT3,
        0, TXT_PID_SHADE);
    u32 *pwd= (u32*)&tile_mem[0][TID_FONT3];
    for(ii=0; ii<96*8; ii++)
        *pwd++ |= quad8(TXT_PID_BG);

    // extra border initialisation
    memcpy32(pal_bg_mem, borderPal, borderPalLen/4);
    memcpy32(&tile_mem[0][TID_FRAME0], borderTiles, borderTilesLen/4);

    // (2) overwrite /\ [] `% ^_ to use border tiles
    //   / ^ \
    //   [ # ]
    //   `   _ '
    const u8 bdr_lut[9]= "/^\\[#]`_\'";
    for(ii=0; ii<9; ii++)
        gptxt->chars[bdr_lut[ii]]= TID_FRAME0+ii;
```

```c
    // (3) set some extra colors
    pal_bg_mem[0x1F]= CLR_RED;
    pal_bg_mem[0x2F]= CLR_GREEN;
    pal_bg_mem[0x3F]= CLR_BLUE;

    pal_bg_mem[0xE8]= pal_bg_mem[0x08]; // bg
    pal_bg_mem[0xEE]= CLR_ORANGE;    // shadow
    pal_bg_mem[0xEF]= pal_bg_mem[0x0F]; // text
}

void txt_se_frame(int l, int t, int r, int b, u16 se0)
{
    int ix, iy;
    u8 *lut= gptxt->chars;
    u16 *pse= (u16*)gptxt->dst0;
    pse += t*32 + l;
    r -= (l+1);
    b -= (t+1);

    // corners
    pse[32*0  + 0] = se0+lut['/'];
    pse[32*0  + r] = se0+lut['\\'];
    pse[32*b  + 0] = se0+lut['`'];
    pse[32*b  + r] = se0+lut['\''];

    // horizontal
    for(ix=1; ix<r; ix++)
    {
        pse[32*0+ix]= se0+lut['^'];
        pse[32*b+ix]= se0+lut['_'];
    }
    // vertical + inside
    pse += 32;
    for(iy=1; iy<b; iy++)
    {
        pse[0]= se0+lut['['];
        pse[r]= se0+lut[']'];
        for(ix=1; ix<r; ix++)
            pse[ix]= se0+lut['#'];
        pse += 32;
    }
}

int main()
{
    init();

    // (4a) red, green, blue text
    se_puts(8, 16, "bank 1:\n  red",    0x1000);
```

```
    se_puts(8, 40, "bank 2:\n  green", 0x2000);
    se_puts(8, 72, "bank 3:\n  blue",  0x3000);
    // (4b) yellow text with magenta shadow
    se_puts(8, 96, "bank 15:\n yellow, \nwith mag \nshadow",
0xF000|TID_FONT2);

    // (5a) framed text, v1
    txt_se_frame(10, 2, 29, 9, 0);
    se_puts( 88, 24, "frame 0:", 0);
    se_puts(104, 32, "/^\\[#]`_'", 0);
    se_puts( 88, 40, "bank  0:\n  basic text,\n  transparent bg", 0);

    // (5b) framed text, v2
    txt_se_frame(10, 11, 29, 18, TID_FRAME1-TID_FRAME0);
    se_puts( 88,  96, "frame 1:", 0xE000|TID_FONT3);
    se_puts(104, 104, "/^\\[#]`_'", 9);
    se_puts( 88, 112, "bank 14:\n  shaded text\n  opaque bg",
0xE000|TID_FONT3);

    while(1)
        VBlankIntrWait();
    return 0;
}
```
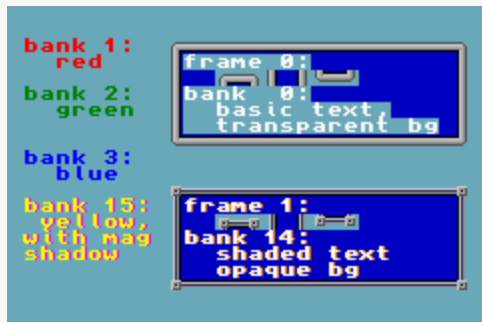

fig 19.5a: First map text demo.

fig 19.5b: accompanying tileset.

Code rundown

fig 19.5 shows what this code produces. All the actual text drawing is done in the main function, and I'll go by them one by one. The first three things are red, green and blue text (point 4a), done using palette swapping. I've loaded up red, green and blue to palette indices `0x1F`, `0x2F` and `0x3F` (point 3), and can switch between them with the last parameter of `se_puts()`, which you will recall is added to each of the screen

entries. The values `0x1000`, `0x2000` and `0x3000` indicate that we'll use palette banks 1, 2 and 3, respectively.

If you look closely, you'll see that fourth text (point 4b) is yellow with a magenta (no it's not pink, it's *magenta*) shading on the right edge of each letter. At least part of this is done with the `se0` parameter, which is now `0xF080`. The reason it's shaded is because of the last part: I'm actually using a slightly different font, one that starts at tile 128. I'll repeat, the reason I can do all this with the same function is because of that offset parameter of `se_puts()`.

Points (5a) and (5b) are for framing, and the text inside it. The function `txt_se_frame()` draws my border. It takes a rectangle as its input, and draws a frame on it. Note that the frame includes the top-left, but excludes the bottom-right. Again, I have one extra `se0` parameter as an offset. This is how the second border is actually done; I just offset the thing by the difference between border tiles.

The borders themselves are actually drawn pretty much as if they were text. In `init()` I've reassigned nine characters in the character lut to use the tile indices for the primary border tileset (point 2). There is no particular reason I'm doing this, other than the mere fact that I can. Just illustrating the things you can do with a text writer and some clever lut manipulation.

The texts inside the frames are an interesting story as well. As you can see from the text in the first frame, the standard text doesn't quite work. The problem is that the main tileset I'm using is transparent, but the frame's background isn't. Mix the two and they'll clash. So how to solve that? Well, you create *another* font, one that does not have 0 as its background color. There are a number of ways to do that, one of them being adding 1<<31 to the bit-unpacking flag. But I'm opting for another method, which I'll get into later. Note that whatever I'm doing, it does work: the text in the second frame is opaque after all. Note that I'm writing that text using pal-bank 14, and am now using a *third* tileset for the fonts.

Now, up to this point it's all been pretty easy. The usage of `se_puts()` and `txt_se_frame()` I mean. I hope you understood all of the above, because the rest is

going to be pretty interesting. Not quite "oh god, oh god, we're all gonna die"-interesting, but still a mite hairy for some.

Bit fiddling fun

I've indicated that I'm using three different fonts. But if you study the code, you will find no trace of font definitions or copies. That's because there are none: it's all based on the same bit-packed font I showed earlier. Also, the mathematically inclined will have noticed that bitpacking a 1bpp font will result in two colors. That's what 1bpp *means*, after all. But I have a background color, a foreground color, and shading; that's three. Furthermore, there doesn't seem to be any code that does the shading. This all leads to one simple question, namely: what the hell am I doing?

Well … this:

```
#define TID_FONT           0
#define TID_FONT2        128
#define TID_FONT3        256
#define TXT_PID_SHADE    0xEE
#define TXT_PID_BG       0x88

// (1a) Basic se text initialization
txt_init_se(0, BG_CBB(0) | BG_SBB(31), 0x1000, CLR_RED, 0x0E);

// (1b) again, with a twist
txt_init_se(0, BG_CBB(0) | BG_SBB(31), 0xF000|TID_FONT2,
    CLR_YELLOW | (CLR_MAG<<16), TXT_PID_SHADE);

// (1c) and once more, with feeling!
txt_init_se(0, BG_CBB(0) | BG_SBB(31), 0xE000|TID_FONT3,
    0, TXT_PID_SHADE);
u32 *pwd= (u32*)&tile_mem[0][TID_FONT3];
for(ii=0; ii<96*8; ii++)
    *pwd++ |= quad8(TXT_PID_BG);
```

These six statements set up the three fonts, complete with shading and opacity. The first one sets up the standard font, in charblock 0, screenblock 31, pal-bank 1 and using `0x0E` for the bit-unpacking offset, so that the text color is at `0x1F`. We've seen the same thing with the object text.

The second call to `txt_se_init()` sets up the second font set, the one with shading. `se0` indicates the use of pal-bank 15 and

bit	val		7	6	5	4	3	2	1	0	
0	0		.	.	.	.	.	.	.	0	
1	1		.	.	.	.	.	E	F	.	
2	1		.	.	.	.	E	F	.	.	
3	1		.	.	.	E	F	.	.	.	
4	0		.	.	.	0	.	.	.	.	
5	0		.	.	0	.	.	.	.	.	
6	1		E	F	.	.	.	.	.	.	
7	0		0	.	.	.	.	.	.	.	
OR:			E	F	0	E	F	F	F	0	

Table 19.2: bit-unpacking with with base `0xEE` .

to start at 128, but the important stuff happens in the `clrs` and `base` parameters. There are now two colors in `clrs` , yellow and magenta. The lower halfword will be the text color, and the upper halfword the shading color.

The actual shading happens because of the value of `base` , which is `0xEE` , and the way the whole bit-unpacking routine works. The offset is added to each 'on'-bit in the packed font, giving `0xEF` , which is then ORred to the current word with the appropriate shift. Because we're dealing with a 4bpp font, the result will actually overflow into the next nybble. Now, if the next bit is also on, it'll OR `0xEF` with the overflow value of `0x0E` . As `0xF | 0xE` is just `0xF` , it's as if the overflow never happened. But, if the next bit was *off*, the value for that pixel would be `0xE` . Lastly, if there was no overflow for a zero source bit, the result is a 0. And now we have the three possible values: 0 (background), 14 (shade) and 15 (text). table 19.2 shows the procedure more graphically. The bits for the source byte are on the left, and the bit-unpacked result for each bit on the grid on the right, in the correct position. These are then ORed together for the end result. For `0x46` that'd be the word `0xEF0EFFF0` . One word is one row of 8 pixels in a 4bpp tile, and because lower nybbles are the left-most pixels, the shade will be on the right of the character even though it uses the more significant bits.

The base `0xEE` is one of many values for which this trick works. The key thing is that the high nybble must be completely overwritten by the lower nybble+1. Any number with equal and even nybbles will work.

Now, I'll be the first to admit that this is something of a hack. A lot of things have to come together for it to work. The word-size must be able to fit a whole tile row, both packed and unpacked data must be little-endian in both bit and byte order, and the unpacking routine must actually allow overflow, and probably a few other things that escape me right now. All of these conditions are satisfied on the GBA, but I doubt very much if you can use the trick on other systems. There are other ways of applying shading, of course, better ones at that. It's just so deliciously nasty that I can't resist using it.

The final `txt_se_init()` work pretty much in the same way as the second one: shading through overflow. What it doesn't do is make the tiles opaque. While it's possible to do that with BitUnpack, you can't have that *and* shading with one call, that simply doesn't work. But there are other ways. All we really need for the tiles to be opaque is some value other than zero for it for the background pixels. Well, that's easily done: just offset (add or OR) everything by a number. In this case I can't add a value because the text value is already at maximum, so I'll use OR here. The value I'll OR with is `0x88888888`, which doesn't change the text or shading, but sets the background pixels to use `8`, so we've got what we wanted.

And that, as they say, is how we do that. Or at least how *I* do that. If the above seems like mumbo-jumbo to you, no one's forcing you to do it in the same way. You can always take the easy way out and include multiple fonts into the program rather than construct them from what you have. I'm just showing what can be done with a little creating coding.

Map text : profiling

The last thing I'll show you is an easy one, but something that might come in handy when it's time to optimize a few things. In case you haven't noticed, debugging GBA programs isn't quite as easy as debugging PC programs. There is the possibility of

debugging with Insight and the GDB (the GCC debugger), but even then things are iffy, or so I hear. Well, now that you can print your own text, you can at least do something of that sort. Write out diagnostic messages and the like.

But that's not what I'm going to show you now. The last demo will show you how to do something that usually comes *after* debugging: profiling. Profiling tells you how much time is spent doing what, so you can tell what would be the best places to try to optimize. What I'll show you is a simple way of getting the time spent inside a function. Stuff like that is good to know, especially on a platform like this where you still have to worry about things like speed and efficiency and other silly stuff like that.

The next demo will clock five different ways of copying data, in this case a mode 4 bitmap from EWRAM (my code is set up for multiboot by default, which means everything goes in EWRAM rather than ROM) to VRAM. The methods are:

- **u16 array**. Copy in 16-bit (halfword) chunks. Probably the one you'll see most in other tutorials, but not here. With reason, as we'll see in a minute.
- **u32 array**. Copy in 32-bit (word) chunks.
- `memcpy()`. The standard C copy routine, the one I'm using in the earlier demos. Well, nowadays I am.
- `memcpy32()`. Home grown assembly, explained in detail here. Basically does what `CpuFastSet()` does, only without the restriction that the number of words must be a multiple of 8.
- `dma_memcpy()`. Copy via 32-bit DMA.

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>

#include <tonc.h>

#include "gba_pic.h"

// === CONSTANTS & STRUCTS ============================================

int gtimes[5];

const char *strs[5]=
{    "u16 array", "u32 array", "memcpy", "memcpy32", "DMA32" };

// === FUNCTIONS ======================================================

// copy via u16 array
void test_0(u16 *dst, const u16 *src, u32 len)
{
    u32 ii;
    profile_start();
    for(ii=0; ii<len/2; ii++)
        dst[ii]= src[ii];
    gtimes[0]= profile_stop();
}

// copy via u32 array
void test_1(u32 *dst, const u32 *src, u32 len)
{
    u32 ii;
    profile_start();
    for(ii=0; ii<len/4; ii++)
        dst[ii]= src[ii];
    gtimes[1]= profile_stop();
}

// copy via memcpy
void test_2(void *dst, const void *src, u32 len)
{
    profile_start();
    memcpy(dst, src, len);
    gtimes[2]= profile_stop();
}

// copy via my own memcpy32
void test_3(void *dst, const void *src, u32 len)
{
    profile_start();
    memcpy32(dst, src, len/4);
```

```
        gtimes[3]= profile_stop();
}

// copy using DMA
void test_4(void *dst, const void *src, u32 len)
{
    profile_start();
    dma3_cpy(dst, src, len);
    gtimes[4]= profile_stop();
}

int main()
{
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    irq_init(NULL);
    irq_add(II_VBLANK, NULL);

    test_0((u16*)vid_mem, (const u16*)gba_picBitmap, gba_picBitmapLen);
    test_1((u32*)vid_mem, (const u32*)gba_picBitmap, gba_picBitmapLen);
    test_2(vid_mem, gba_picBitmap, gba_picBitmapLen);
    test_3(vid_mem, gba_picBitmap, gba_picBitmapLen);
    test_4(vid_mem, gba_picBitmap, gba_picBitmapLen);

    // clear the screenblock I'm about to use
    memset32(&se_mem[7], 0, SBB_SIZE/4);

    // init map text
    txt_init_std();
    txt_init_se(0, BG_SBB(7), 0, CLR_YELLOW, 0);

    // print results
    int ii;
    char str[32];
    for(ii=0; ii<5; ii++)
    {
        siprintf(str, "%12s %6d", strs[ii], gtimes[ii]);
        se_puts(8, 8+8*ii, str, 0);
    }

    while(1)
        VBlankIntrWait();

    return 0;
}
```

The code should be self-explanatory. I have five functions for the things I want to profile. I chose separate functions because then I know optimisation will not interfere

(it sometimes moves code around). After running these functions, I set-up my text functions and print out the results.

The profiling itself uses two macros, `profile_start()` and `profile_stop()`. These can be found in `core.h` of libtonc. What the macros do is start and stop timers 2 and 3, and then return the time in between the calls. This does mean that the code you're profiling cannot use those timers.

```
INLINE void profile_start()
{
    REG_TM2D= 0;     REG_TM3D= 0;
    REG_TM2CNT= 0;   REG_TM3CNT= 0;
    REG_TM3CNT= TM_ENABLE | TM_CASCADE;
    REG_TM2CNT= TM_ENABLE;
}

INLINE u32 profile_stop()
{
    REG_TM2CNT= 0;
    return (REG_TM3D<<16)|REG_TM2D;
}
```


fig 19.6a: `txt_se2` on VBA.

fig 19.6b: `txt_se2` on no$gba.

	hardware	vba	no$gba	vba err	no$ err
u16 array	614571	499440	614571	-18.73	0.00
u32 array	289825	230383	288098	-20.51	-0.60
memcpy	195156	161119	194519	-17.44	-0.33
memcpy32	86816	79336	85329	-8.62	-1.71
DMA32	76889	250	76888	-99.67	0.00

table 19.3: timing results for hardware, vba and no$gba.

Fig 19.6 shows the timing results, as run in VisualBoy Advance and no$gba. Note that they are not quite the same. So you do what you should always do when two opinions differ: get a third one. In this case, I'll use the only one that really matters, namely hardware. You can see a comparison of the three in table 19.3, which will tell you that no$gba is very accurate in its timing, but VBA not so much. I guess you can still use it to get an estimate or relative timings, but true accuracy will not be found there. For that you need hardware or no$gba.

About the numbers themselves. The spread is about a factor 9, which is quite a lot. None of the techniques shown here are particularly hard to understand, and data copying is something that you could spend a lot of time doing, so might as well take advantage of the faster ones from the get go.

Most of the tutorial code and probably a lot of demo code you can find out there uses the u16-array method of copying; presumably because byte-copies are unavailable for certain sections. But as you can see, **u16 copies are more than twice as slow as u32 copies**! Granted, it is not the slowest method of copying data, but not by much (using u16 loop variables –also a common occurence– would be slower by about 20%; try it and you'll see). The GBA is a 32-bit machine. It *likes* 32-bit data, and its instruction sets are better at dealing with 32-bit chunks. Let go of the u16 fetish you may have picked up elsewhere. Use word-sized data if you can, the others only if you have to. That said, do watch your data alignment! u8 or u16 arrays aren't always word-aligned, which will cause trouble with casting.

GCC AND WAITSTATES VS TIMING RESULTS

Giving exact timing results is tricky due to a number of factors. First, on the hardware side there are different memory sections with different wait states that complicate things unless you sit down, read the assembly and add up the cycle-counts of the instructions. This is a horrible job, trust me. The second problem is that GCC hasn't reached the theoretical optimum for this code yet,

> so the results tend to vary with new releases. What you see above is a good indication, but your mileage may vary.

There are a number of fast ways of copying large chunks of data. Faster than writing your own simple loop that is. Common ones are the standard `memcpy()`, which is available for any platform, and two methods that are GBA specific: the `CpuFastSet()` BIOS call (or my own version `memcpy32()` and DMA. The first two *require* word-alignment; DMA merely works better with it. The performance of `memcpy()` is actually not too shabby, and the fact that it's available everywhere means that it's a good place to start. The others are faster, but come at a cost: `memcpy32()` is hand written assembly; `CpuFastSet()` requires a word-count divisible by 8, and DMA locks up the CPU, which can interfere with interrupts. You would do well to remember these things when you find you need a little more speed.

Other considerations

These couple of functions barely scratch the surface as far as text systems are concerned. You can have larger fonts, colored fonts, proper shading, variable character widths, and more. Each of these can apply to each of the modes, with extra formatting for text justification and alignment, updating tile-memory in conjunction with map/OAM changes to cut down on VRAM use, etc, etc. To take an in-depth look at all the variations would take an entire site by itself, so I'll leave it at this. I just hope you've picked up on some of the basics that go into text systems. What you do with that knowledge I leave up to you.

20. Mode 7 Part 1

Right, and now for something cool: mode 7. Not just how to implement it on the GBA, but also the math behind it. You need to know your way around tiled backgrounds (especially the transformable ones) interrupts. Read up on those subjects if you don't. The stuff you'll find here explains the basics of Mode 7. I also have an advanced page, but I urge you to read this one first, since the math is still rather easy compared to what I'll use there.

Introduction

Way, way back in 1990, there was the Super NES, the 16bit successor to the Nintendo Entertainment System. Apart from the usual improvements that are inherent to new technology, the SNES was the first console to have special hardware for graphic tricks that allowed linear transformations (like rotation and scaling) on backgrounds and sprites. *Mode7* took this one step further: it not only rotated and scaled a background, but added a step for perspective to create a 3D look.

One could hardly call Mode 7 yet another pretty gimmick. For example, it managed to radically change the racing game genre. Older racing games (like Pole Position and Outrun) were limited to simple left and right bends. Mode 7 allowed more interesting tracks, as your vision wasn't limited to the part of the track right in front of you. F-Zero was the first game to use it and blew everything before it out of the water (the original

Fire Field is still one of the most vicious tracks around with its hairpins, mag-beams and mines). Other illustrious games were soon to follow, like Super Mario Kart (mmmm, Rainbow Road. 150cc, full throttle all the way through *gargle*) and Pilotwings.

Since the GBA is essentially a miniature SNES, it stands to reason that you could do Mode7 graphics on it as well. And, you'd be right, although I heard the GBA Mode7 is a little different as the SNES'. On the SNES the video modes really did run up to #7 (see the SNESdev Wiki) The GBA only has modes 0-5. So technically "GBA Mode 7" is a yet another misnomer. However, for everyone who's not a SNES programmer (which *is* nearly everyone, me included) the term is synonymous with the graphical effect it was famous for: a perspective view. And you can create a perspective view on the GBA, so in that sense the term's still true.

I'm not sure about the SNES, but GBA Mode 7 is a very much unlike true 3D APIs like OpenGL and Direct3D. On those systems, you can just give the proper perspective matrix and place it into the pipeline. On the GBA, however, you only have the general 2D transformation matrix **P** and displacement **dx** at your disposal and you have to do all the perspective calculations yourself. This basically means that you have to alter the scaling and translation on every scanline using either the HBlank DMA or the HBlank interrupt.

In this tutorial, I will use the 64x64t affine background from the sbb_aff demo (which looks a bit like fig 20.1), do the Mode7 mojo and turn it into something like depicted in fig 20.2. The focus will be on showing, in detail, how the magic works. While the end result is given as a HBlank interrupt function; converting to a HBlank DMA case shouldn't be to hard.

Fig 20.1: this is your map (well, kinda)
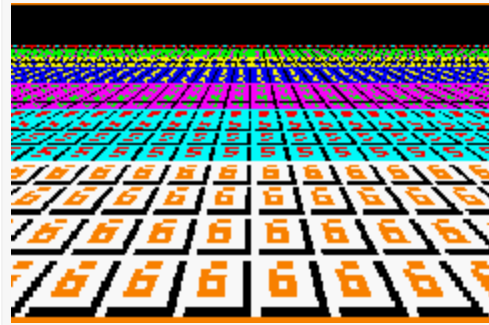
Fig 20.2: this is your map in mode7.

Getting a sense of perspective

(If you are familiar with the basics of perspective, you can just skim this section.) If you've ever looked at a world map or a 3D game, you know that when mapping from 3D to 2D, something' has to give. The technical term for this is *projection*. There are many types of projection, but the one we're concerned with is *perspective*, which makes objects look smaller the further off they are.

We start with a 3D space like the one in fig 20.3. In computer graphics, it is customary to have the *x*-axis pointing to the right and the *y*-axis pointing up. The *z*-axis is the determined by the handedness of the space: a *right-handed* coordinate system has it pointing to the back (out of the screen), which in a left-handed system it's pointing to the front. I'm using a right-handed system because my mind gets hopelessly confused in a left-handed system when it comes to rotation and calculating normals. Another reason is that this way the screen coordinates correspond to (*x, z*) values. It is also customary to have the viewer at the origin (for a different viewer position, simply translate the world in the other direction). For a right-handed system, this means that you're looking down the negative *z*-axis.

Of course, you can't see everything: only the objects inside the **viewing volume** are visible. For a perspective projection this is defined by the viewer position (the origin in our case) and the **projection plane**, located in front of the viewer at a distance D. Think of it as the screen. The projection plane has a width W and height H. So the viewing volume is actually a **viewing pyramid**, though in practice it is usually a viewing *frustum* (a beheaded pyramid), since there is a minimum and maximum to the distance you can perceive as well.

Fig 20.4 shows what the perspective projection actually does. Given is a point (y, z) which is projected to point $(y_p, -D)$ on the projection plane. The projected z-coordinate, by definition, is $-D$. The projected y-coordinate is the intersection of the projection plane and the line passing through the viewer and the original point:

$$(20.1) \qquad y_p = y \cdot D/z$$

Basically, you divide by z/D. Since it is so important a factor it has is own variable: the **zoom factor** λ:

$$(20.2) \qquad \lambda = z / D = y / y_p$$

As a rule, everything in front the projection plane ($\lambda < 1$) will be enlarged, and everything behind it ($\lambda > 1$) is shrunk.
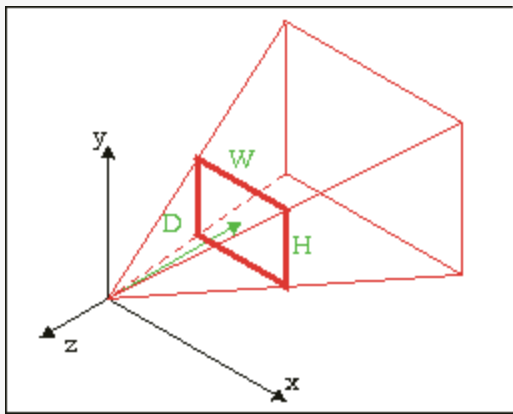
Fig 20.3: 3D coordinate system showing the viewing pyramid defined by the origin, and the screen rectangle ($W \times H$) at $z = -D$
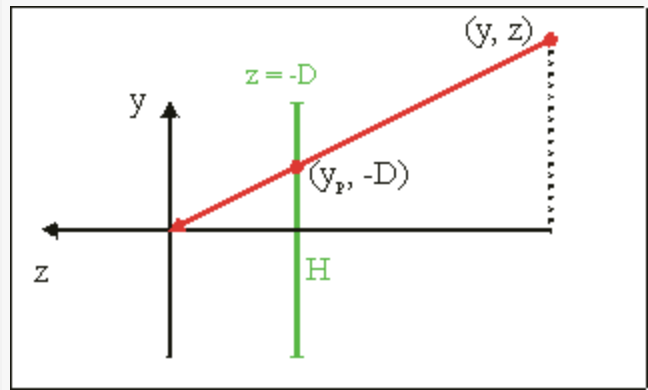
Fig 20.4: Side view; point (y, z) is projected onto the ($z = -D$ plane. The projected point is $y_p = y \cdot D/z$

Enter Mode 7

Fig 20.3 and fig 20.4 describe the general case for perspective projection in a 3D world with tons of objects and viewer orientations. The case for Mode 7 is considerably less complicated than that:

- **Objects**. We only work with two objects: the viewer (at point **a** = (a_x, a_y, a_z)) and the floor (at $y=0$, by definition).
- **Viewer orientation**. In a full 3D world, the viewer orientation is given by 3 angles: yaw (y-axis), pitch (x-axis) and roll (z-axis). We will limit ourselves to yaw to keep things simple.
- **The horizon issue**. Because the view direction is kept parallel to the floor, the horizon should go in the center of the screen. This would leave the top half of the screen empty, which is a bit of a waste. To remedy this we only use the bottom half of the viewing volume, so that the horizon is at the top of the screen. Note that even though the top and bottom view-lines are now the same as when you would look down a bit, the cases are *NOT* equal as the projection plane is still vertical. It is important that you realize the difference.
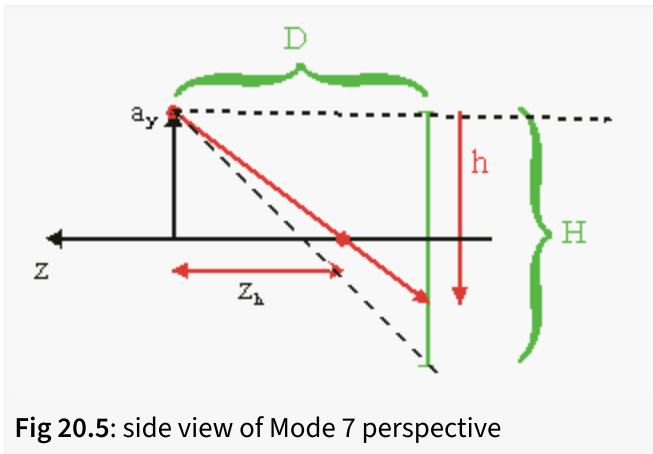
Fig 20.5: side view of Mode 7 perspective

Fig 20.5 shows the whole situation. A viewer at $y = a_y$ is looking in the negative z-direction. At a distance D in front of the viewer is the projection plane, the bottom half of which is displayed on the GBA screen of height H (=160). And now for the fun part. The GBA doesn't have any real 3D hardware capabilities, but you can fake it by cleverly manipulating the scaling and translation `REG_BGxX–REG_BGxPD` for every scanline. You just have to figure out which line of part of the floor goes into which scanline, and at which zoom level. Effectively, you're building a very simple ray-caster.

The math

Conceptually, there are four steps to Mode 7, depicted in fig 20.6a-d. Green figures indicate the original map; red is the map after the operation. Given a scanline h, here's what we do:

1. **Pre-translation** by a= (a_x, a_z). This places the viewer at the origin, which is where we need it for steps b and c.
2. **Rotation** by α. This takes care of the yaw angle. These steps have been the same as for normal transformable backgrounds so you shouldn't have any difficulty understanding them.
3. **Perspective division**. Next, we scale the whole thing by $1/\lambda$. From eq 20.2 we have $\lambda = a_y/h$. The line $z = z_h$ is the line that belongs on scanline h. The new position of this line after scaling is $z = -D$, since that was the whole point of perspective division.

4. **Post-translation** by (−x$_S$). Note the minus sign. After the perspective division, all that remains is moving the fully transformed map back to its proper screen position (the beige area). For obvious reasons the horizontal component should be half the screen width. The vertical move should move the floor-line to the scanline, so the vector is:

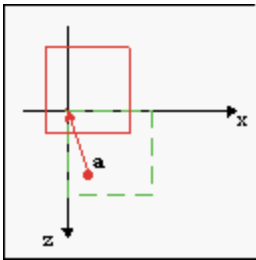$$(20.3) \qquad \begin{aligned} x_s &= W/2 = 120 \\ y_s &= (D+h) \end{aligned}$$

Fig 20.6a: pre-translate by (a_x, a_z)	Fig 20.6b: rotate by α	Fig 20.6c: scale by 1/λ	Fig 20.6d: post-translate by (x_S, y_S)
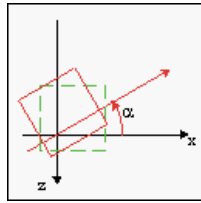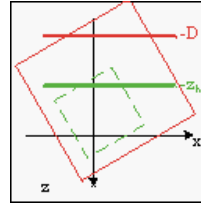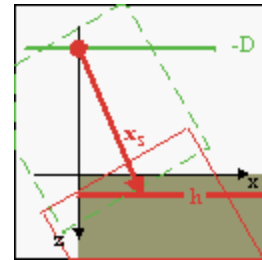
Fig 20.6a-d: The 4 steps of mode 7

Putting it all together

While the steps described above are indeed the full procedure, there are still a number of loose ends to tie up. First of all, remember that the GBA's transformation matrix **P** maps from screen space to background space, which is actually the inverse of what you're trying to do. So what you should use is:

$$(20.4) \qquad P = S(\lambda) \cdot R(a) = \begin{bmatrix} \lambda \cdot \cos(a) & -\lambda \cdot \sin(a) \\ \lambda \cdot \sin(a) & \lambda \cdot \cos(a) \end{bmatrix}$$

And yes, the minus sign is correct for a counter-clockwise rotation (**R** is defined as a clockwise rotation). Also remember that the GBA uses the following relation between screen point **q** and background point **p**:

$$(20.5) \qquad dx + P \cdot q = p$$

that is, <u>one</u> translation and <u>one</u> transformation. We have to combine the pre- and post-translations to make it work. We've seen this before in eq 4 in the affine background page, only with different names. Anyway, what you need is:

(20.6)	$$dx + P \cdot q = p$$ $$P \cdot (q - x_s) = p - a$$ $$\overline{dx + P \cdot x_s = a}$$ $$dx = a - P \cdot x_s$$

So for each scanline you do the calculations for the zoom, put the **P** matrix of eq 20.4 into `REG_BGxPA–REG_BGxPD` , and **a−P·x**$_s$ into `REG_BGxX` and `REG_BGxY` and presto! Instant Mode 7.

Well, almost. Remember what happens when writing to `REG_BGxY` inside an HBlank interrupt: the *current* scanline is perceived as the screen's origin null-line. In other words, it does the +*h* part of y_s automatically. Renaming the true y_s to y_{s0}, what you *should* use is

(20.7)	$y_s = y_{s0} - h = D$

Now, in theory you have everything you need. In practice, though, there are a number of things that can go wrong. Before I go into that, here's a nice, (not so) little demo.

Threefold demo

As usual, there is a demo. Actually, I have several Mode 7 demos, but that's not important right now. The demo is called `m7_demo` and the controls are:

D-pad	Strafe.
L, R	turn left and right (i.e., rotate map right and left, respectively)

A, B	Move up and down, though I forget which is which.
Select	Switch between 3 different Mode7 types (A, B, C)
Start	Resets all values (**a** = (256, 32, 256), α = 0)

"Switch between 3 different Mode7 types"? That's what I said, yes. Make sure you move around in all three types. Please. There's a label in the top-left corner indicating the current type.
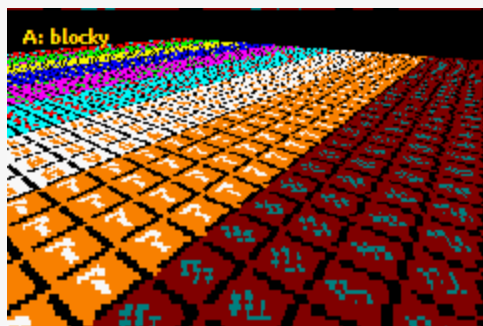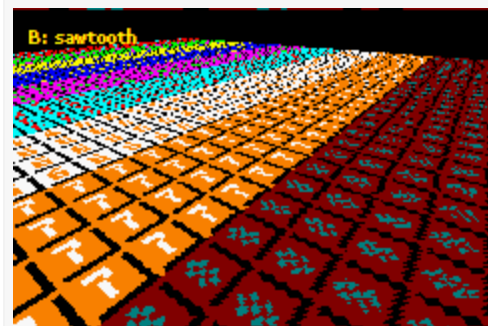

Fig 20.7a: Type A: blocked.

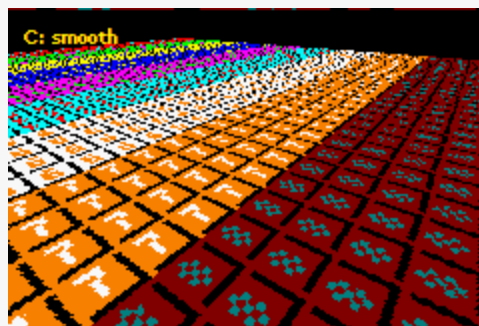**Fig 20.7b**: Type B: sawtooth.

Fig 20.7c: Type C: smooth.

Order, order!

Fiddled with my demo a bit? Good. Noticed the differences between the three types? Even better! For reference, take a look at Figs 20.7a-c, which correspond to the types. They adequately show what's different.

- Type A is horribly blocky. Those numbers in the red tiles are supposed to be '8's. Heh, numbers? What numbers!
- Type B is better. The left-hand side is smooth, but there's still some trouble on the right-hand side. But at least you can see eights with some imagination.
- Type C. Now we're talking! The centerline is clear, which is important since that's what you're looking at most of the time. But even on the sides, things are looking pretty decent.

So we have three very different Mode7 results, but I guarantee you it's all based on the same math. So how come one method looks so crummy, and the other looks great?

The code

Here are the two HBlank ISRs that create the types. Types A and B are nearly identical, except for one thing. Type C is very different from the others. If you have a thing for self-torture, try explaining the differences from the code alone. I spent most of yesterday night figuring out what made Type C work, so I have half a mind of leaving you hanging. Fortunately for you, that half's asleep right now.

```
#define M7_D    128

extern VECTOR cam_pos;          // Camera position
extern FIXED g_cosf, g_sinf;    // cos(phi) and sin(phi), .8f
```

```
// --- Type A ---
// (offset * zoom) * rotation
// All .8 fixed
void m7_hbl_a()
{
    FIXED lam, xs, ys;

    lam= cam_pos.y*lu_div(REG_VCOUNT)>>16;  // .8*.16/.16 = .8

    // Calculate offsets (.8)
    xs= 120*lam;
    ys= M7_D*lam;

    REG_BG2PA= (g_cosf*lam)>>8;
    REG_BG2PC= (g_sinf*lam)>>8;

    REG_BG2X = cam_pos.x - ( (xs*g_cosf-ys*g_sinf)>>8 );
    REG_BG2Y = cam_pos.z - ( (xs*g_sinf+ys*g_cosf)>>8 );
}
```

```
// --- Type B ---
// (offset * zoom) * rotation
// Mixed fixed point: lam, xs, ys use .12
void m7_hbl_b()
{
    FIXED lam, xs, ys;

    lam= cam_pos.y*lu_div(REG_VCOUNT)>>12;  // .8*.16/.12 = .12

    // Calculate offsets (.12f)
    xs= 120*lam;
    ys= M7_D*lam;

    REG_BG2PA= (g_cosf*lam)>>12;
    REG_BG2PC= (g_sinf*lam)>>12;

    REG_BG2X = cam_pos.x - ( (xs*g_cosf-ys*g_sinf)>>12 );
    REG_BG2Y = cam_pos.z - ( (xs*g_sinf+ys*g_cosf)>>12 );
}
```

```
// --- Type C ---
// offset * (zoom * rotation)
// Mixed fixed point: lam, lcf, lsf use .12
// lxr and lyr have different calculation methods
void m7_hbl_c()
{
    FIXED lam, lcf, lsf, lxr, lyr;

    lam= cam_pos.y*lu_div(REG_VCOUNT)>>12;  // .8*.16 /.12 = 20.12
    lcf= lam*g_cosf>>8;                     // .12*.8 /.8 = .12
    lsf= lam*g_sinf>>8;                     // .12*.8 /.8 = .12

    REG_BG2PA= lcf>>4;
    REG_BG2PC= lsf>>4;

    // Offsets
    // Note that the lxr shifts down first!

    // horizontal offset
    lxr= 120*(lcf>>4);      lyr= (M7_D*lsf)>>4;
    REG_BG2X= cam_pos.x - lxr + lyr;

    // vertical offset
    lxr= 120*(lsf>>4);      lyr= (M7_D*lcf)>>4;
    REG_BG2Y= cam_pos.z - lxr - lyr;
}
```

The discussion (technical)

All three versions do the following things: calculate the zoom-factor λ, using eq 2 and a division LUT, calculate the affine matrix using λ and stored versions of $\cos(\phi)$ and $\sin(\phi)$, and calculate the affine offsets. Note that only p_a and p_c are actually calculated; because the scanline offset is effectively zero all the time, p_b and p_d have no effect and can be ignored. Those are the similarities, but what's more interesting are the differences:

1. **Fixed point**. Type A uses .8 fixed point math throughout, but B and C use a combination of .12 and .8 fixeds.
2. **Calculation order of the affine offset** The affine displacement **dx** is a combination of 3 parts: scale, rotation and offsets. Type A and B use **dx** =

(offset*scale)*rotation, while C uses **dx** = offset*(scale*rotation). Because type C does the offsets last, it can also use different fixed-points for the offsets.

These two (well, 2 and a half, really) differences are enough to explain the differences in the results. Please remember that the differences in the code are quite subtle: fixed point numbers are rarely used outside consoles, and results changing due to

h	1/h	λ (true)	λ(.8)
157	0.01a16d..h	0.342da7h	0.34h
158	0.019ec8..h	0.33d91dh	0.33h
159	0.019c2d..h	0.3385a2h	0.33h
160	0.019999..h	0.333333h	0.33h

Table 20.1: division tables and zoom factors. a_y=32

the order of calculation is probably even rarer. Yet is these two items that make all the difference here.

Let's start with types A and B, which differ only by the fixed-point of `lam` . λ is the ration of the camera height and the scanline, which will often be quite small – smaller than 1 at any rate. table 20.1 shows a few of the numbers. Note that using a λ with only 8 fractional bits means that you'll often have the same number for multiple scanlines, which carries through in the later calculations. This is why type A, which plays by the rules and uses a constant fixed-point like a good little boy, is so blocky at low altitudes. The four extra bits of type B gives much better results. Rules are nice and all, but sometimes they needs to be broken to get results.

Now, you will notice that type B still has a bit of distortion, so why only go to .12 fixeds in type B, why not 16? Well, with 16 you can get into trouble with integer overflow. It'd be alright for calculating `xs` and `ys` , but we still have to rotate these values later on as well. OK, so we'll use 64bit math, then the 32bit overflow wouldn't matter and we could use *even more* fixed point bits! After all, more == better, right?

Well, no. Bigger/stronger/more does not always mean better (see the DS vs PSP). The remaining distortion is not a matter of the number of fixed-point bits; not exactly. You could use a 128bit math and .32f division and trig tables for all I care; it wouldn't matter here, because that's not were the problem is.

The problem, or at least part of it, is the basic algorithm used in types A and B. If you look back to the theory, you'll see that the affine matrix is calculated first, then the offsets. In other words, first combine the scale and rotation, then calculate the offset-correction, $\mathbf{P} \cdot \mathbf{x}_s$. This is how the affine parameters in the GBA work anyway. However, this is actually only the first step. If you follow that procedure, you'd still get the jagged result. The *real* reason for these jaggies is the order of calculation of `lxr`.

```
// Multiply, then shift to .8 (A and B)
    lxr= (120*lcf)>>4;

// Shift to .8 first, then multiply (C)
    lxr= 120*(lcf>>4);
```

Getting `lxr` $= p_{a/c} \cdot x_s$ requires two parts: multiplication with $\mathbf{P}$ elements and the shift down to .8 fixeds. You might expect doing the shift last would be better because it has a higher precision. The funny thing is that it **doesn't**! Shifting p_a or p_c down to 8 fractional bits before the multiplication is what gets rid of the remaining distortions, reversing the order of operations doesn't.

As for why, I'm not 100% sure, but I can hazard a guess. The affine transformation takes place around the origin of the screen, and to place the origin somewhere else we need to apply a post-translation by $\mathbf{x}_s$. The crucial point I think is that $\mathbf{x}_s$ is a point in screen-space which uses normal integers, not fixed points. However, it only applies to x_s because that *really* represents an on-screen offset; y_s is actually not a point on the screen but the focal distance of the camera. On the other hand, it might have something to do with the internal registers for the displacement.

The verdict

Obviously, type C is the one you want. It really bugs the hell out of me that I didn't think of it myself. And the fact that I *did* use the scale-rotation multiplication but abandoned it because I screwed up with the multiplication by the projection distance *D* doesn't help either (yes, this sentence makes sense). The code of `m7_hbl_c` shown above works, even though it only uses 32-bit math. As long as you do the scale-

rotation multiplication first and shift down to .8 fixeds before you multiply by 120 in the calculation of `wxr` everything should be fine.

Final Thoughts

This has been one of those occasions that show that programming (especially low-level programming) is as much of a science as an art. Even though the theory for the three mode 7 versions was the same, the slight differences in the order and precision of the calculations in the implementations made for very noticeable differences in the end result. When it comes to mode 7, calculate the affine matrix before the correction offset. But most importantly, the *x*-offset for the screen should not be done in fixed point.

Secondly, this was only the basic theory behind mode 7 graphics. No sprites, no pitch-angle and no horizon, and tailored to the GBA hardware from the start. In the next chapter, we'll derive the theory more extensively following standard 3D theory with linear algebra. This chapter will also show how to position sprites in 3D and how to do other things with them like animating for rotation and sorting, and also present variable-pitch and a horizon. If this sounds complicated, well, I supposed that it is. It's definitely worth a look, though.

21. Mode 7 Part 2

Introduction

Mode 7: part 1 covered the basics of how to get an affine background to look like a 3D plane, and discussed some of the trickier parts of the fixed point arithmetic involved. Getting the basic look of a 3D plane is only the first step.

In this chapter, we'll look into the general math involved of creating a 3D world and translate it back to a form we can use for mode 7. This includes translations in all directions and looking around (yaw) like before, but also a pitch angle for looking up and down. We'll also see how to deal with the horizon and use a background for the backdrop above the horizon. I'll even throw in a bit of fogging to occlude the distant parts of the ground.

I'll discuss also working with sprites in 3D space. Not just the transformation from 3D space to 2D screen, but also culling, scaling for distance (which is not as simple as one might think), animation and sorting. Note that this part of the chapter is basic 3D sprite theory, and can be applied to 3D games that use sprites in some way.

The theory part of the chapter is going to be very math-heavy, as 3D theory always is. Knowing a little bit about linear algebra certainly wouldn't hurt. The full story about

geometry is beyond the scope of Tonc, but this stuff is quite general; most books on 3D programming will have a chapter on geometric transformations, so you can look at those if you get a little lost here.

This chapter touches on almost all of the topics covered so far. It uses affine objects, backgrounds (both regular and affine), interrupts, color effects and a few more. If your understanding of any of these is lacking, you could be in for a rough time here.
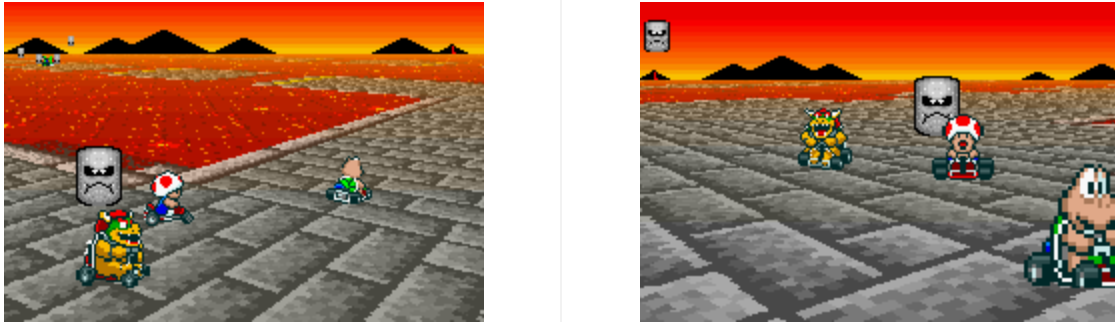
**Fig 21.1**: `m7_ex`; with horizon, sprites, variable pitch angle and distance fogging.

What we're going to try to do is re-create a scene from the SNES Mario Kart (see fig 21.1; apologies to Nintendo for using the graphics, but I don't have a lot of options here `:\`). This is just a freeze-frame of the game, not actual game play is involved, but this should present a nice target to aim for. The code is distributed over a number of files: `mode7.c` for the simple mode 7 functions and `mode7.iwram.c` for the less simple mode 7 functions and interrupt routines. The code of demo-specific code can be found in `m7_ex.c` , which does the set-up, interaction and main loop. The basic controls are as follows:

D-pad	Looking
A/B	Back/forward
L/R	Strafing
Select+A/B	Float up/down
Start	Menu

Movement and looking follows FPS/aircraft motion, or at least as well as could be expected with the number of buttons available. There are several extra options which have been put in a menu. First is *motion control* which sets difference methods of movement. Option 'local' follows the camera axis for flight-controls, 'level' gives movement parallel to the ground, like FPSs usually do, and 'global' uses world axis for movement. Other options include toggling fog on or off and resetting the demo.

Basic mode 7 theory

Fig 21.2 shows what we're up against: we have a camera located at $\mathbf{a}_{cw}$, which has some orientation with respect to the world coordinate system. What we have to do is find the transformation that links screen point $\mathbf{x}_S$ to world point $\mathbf{x}_W$. There are a number of ways to do this. You already saw one in the first mode 7 chapter, where we I had the GBA hardware in mind from the start. You could extend this to the general mode 7 case (with a non-zero pitch) with some effort. You could also use pure trigonometry, which is a minefield of minus signs and potential sine-cosine mix-ups. Still, it is possible. What I'll use here, though, is linear algebra. There are several reasons for this choice. Firstly, linear algebra has a very concise notation, so you can write down the final solution in just a few lines (in fact, once you get through the definitions, the solution that covers all cases can be written down in 2 lines). Furthermore, the equations are well structured and uniform in appearance, making debugging easier. Then there's the fact that inverting the whole thing is very easy. And lastly, it's what true 3D systems use too, so the theory can be applied outside the mode 7 arena as well. Conversely, if you know basic 3D theory, you'll feel right at home here.
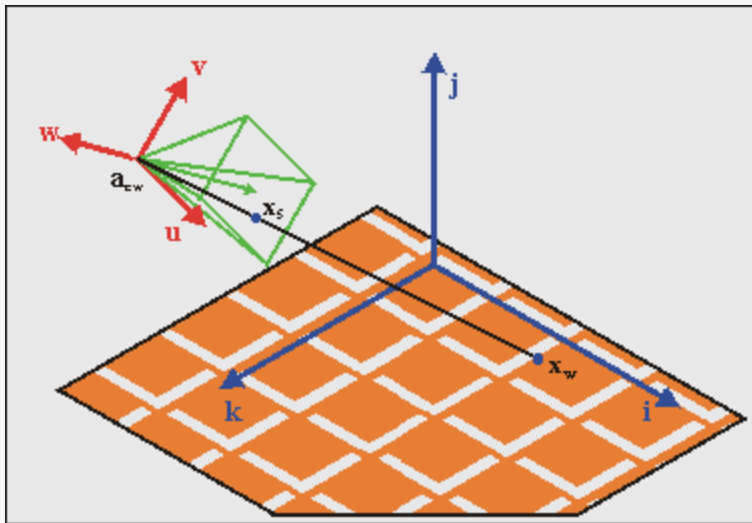
Fig 21.2: The basic 3D situation. The trick is to relate screen point x_s to world point x_w, taking the camera position a_{cw} and its orientation into account.

Definitions

Before you can do anything, though, you need to know *exactly* what we're going to use. The first thing to note is that we have two main coordinate systems: the ***world*** system S_w and the ***camera*** system S_c. Inside the camera system we have two minor coordinate systems, namely the ***projection*** space S_p and ***screen*** space S_s. Now, for every transformation between systems S_i and S_j the following relation holds:

$$(21.1) \qquad M_{ij} \cdot x_j = x_i - a_{ji}$$

where

x_i	the coordinate vector in system S_i;
x_j	the coordinate vector in system S_j;
a_{ji}	the origin of system S_j, expressed in coordinates of system S_i;
M_{ij}	the transformation matrix, which is basically the matrix formed by the principle vectors of S_j, in terms of S_i.

Once you get over the initial shock of the many indices (meh, in general relativity you have something called the Riemann tensor, which has *four* indices), you'll see that this equation makes sense. If you don't get it right away, think of them as arrays and matrices. An observant reader will also recognise the structure in the screen↔map transformation we had for affine maps: **P·q** = **p** − **dx**. Eq 21.1 is a very general equation, by the way, it holds for every kind of linear coordinate transformation. In fact, systems S_i and S_j don't even have to have the same number of dimensions!

As said, we have 4 systems in total, so we have 4 subscripts for **w**(orld), **c**(amera), **p**(rojection), **s**(creen). Remember these, for they will appear on a very regular basis. The final forms of the matrices and origins depend very much on the exact definitions of these systems, so make sure you know exactly what each means.

World system

The first of these, the world system S_w, is easy to deal with. This is simply a right-handed Cartesian system with principle axes **i**, **j**, and **k**, which are its x-, y- and z-axes, respectively. In the right-handed system that is used in computer graphics, the x-axis (**i**) points to the right, the y-axis (**j**) points up and the z-axis (**k**) points *backward*! This means that you're looking in the negative *z* direction, which may seem weird at first. If you absolutely must have a forward pointing **k**, you could use a left-handed system. While this utterly destroys my 3d intuition, if you want it be my guest. Before you do that, though, remember that the map marks the floor of world space and in a right-handed system, the texture coordinates will match up neatly to world coordinates.

The camera frame

The transformation to the camera system is probably the major hurdle in the whole thing. At least it would be if it wasn't for matrices. Rewriting eq 21.1, the transformation between camera and world space is given by

$$(21.2) \qquad C \cdot x_c = x_w - a_{cw}$$

As you can expect, the origin of camera space is the camera position, a_{cw}. The camera matrix **C** is formed by the principle axes of camera space, which are **u, v** and **w** for the local x-, y- and z-axes, respectively. This means that the camera matrix is **C** = [**u v w**].

The orientation of the camera with respect to world space is defined by 3 angles: *pitch* (rotation around the x-axis), *yaw* (rotation around the y-axis) and *roll* (around z-axis). The combination of these give **C**. Traditionally, the rotation direction of these is such that if you

Fig 21.3: Camera orientation {**u, v, w**} in world space {**i, j, k**}, given by angles θ and φ

look down one of these axes, a positive angle turns the system counter-clockwise. However, I'll do the exact opposite, because it makes a number of things easier. Additionally, I will only be using two angles: pitch and yaw. For mode 7 it is impossible to incorporate roll into the picture. Why? Look at it this way: if you're rolled on your side, the ground would be on the right or left of the screen, which would require a vertical perspective division, which is impossible to achieve since we can only change the affine parameters at HBlank. Therefore, only pitch (θ) and yaw (φ) are allowed. I want my positive θ and φ to the view down and right, respectively, meaning I need the following rotation matrices:
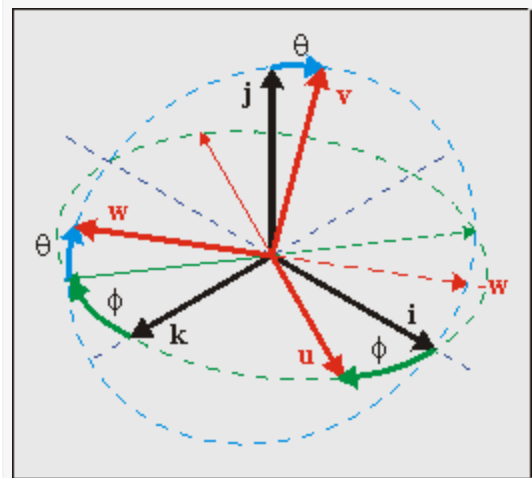
(21.3a)	$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{bmatrix}$
(21.3b)	$R_y(\varphi) = \begin{bmatrix} \cos(\varphi) & 0 & -\sin(\varphi) \\ 0 & 1 & 0 \\ \sin(\varphi) & 0 & \cos(\varphi) \end{bmatrix}$

But now the next problem arises: do we do pitch first, or yaw? That really depends on what kind of effect you want to have *and* in relation to what system you do your

rotation. There is actually only one order that is possible for the same reason that roll wasn't allowed: you cannot have a vertical perspective. What this boils down to is that **u** (the x-axis of the camera frame) *must* be parallel to the ground plane, i.e., u_y must be zero. In order to do that, you must do pitch first, then yaw. This is depicted in fig 21.3. To get a feel for this: stand up, tilt your head down (pitch θ>0), then turn to your right (yaw φ>0). The full camera matrix then becomes:

(21.4)	$C(\theta, \varphi) = R_y(\varphi) \cdot R_x(\theta) = \begin{bmatrix} \cos(\varphi) & \sin(\varphi) \cdot sin(\theta) & -\sin(\varphi) \cdot \cos(\theta) \\ 0 & \cos(\theta) & \sin(\theta) \\ \sin(\theta) & -\cos(\theta) \cdot \sin(\theta) & \cos(\varphi) \cdot \cos(\theta) \end{bmatrix}$

Aside from being correct, this matrix has two nice properties. Firstly, the column vectors are of unit length. Secondly, the component vectors are perpendicular. This means that **C** is an ***orthogonal matrix***, which has the very nice feature that $C^{-1} = C^T$. This makes the world→camera transformation a relatively simple operation.

One last thing here: if you were to rotate the camera system by 180° around **i**, this would give you a forward pointing **w** and a downward pointing **v**, both of which decrease the number of awkward minus signs in later calculations, at the expense of an awkward camera frame. Whether you want to do this is up to you.

> **MATRIX TRANSFORMS AND THE SYSTEM THEY OCCUR IN.**
>
> I said that to mimic the rotations of **C** you to tilt your head first (θ), then rotate your body (φ). You might think that you can get the same effect by doing it the other way: turn first, then look down. However, this is incorrect.
>
> It may *feel* the same, but in the second case you'd not actually be using the $R_x(\theta)$ to invoke the tilt. A matrix isn't a thing in itself, it 'lives' in a space. In this case, both $R_x(\theta)$ and $R_y(\phi)$ are defined in terms of the *world* coordinate system, and when applying them the directions follow the world's axes. The turn-then-tilt order would use $R_x(\theta)$ in a local frame, which is a legal operation, but not the one that the math requires.

> I know it's a subtle point, but there really is an important difference. Try visualizing it with a 90° rotation in both orders, maybe that'd help.

The projection plane

To create the illusion of depth we need a *perspective view*. For this, you need a ***center of projection*** (COP) and a ***projection plane***. Naturally, both need to be in camera space. While you're free to choose these any way you want, you can simplify matters by placing the center of projection at the origin of camera space and the projection plane at a distance D in

Fig 21.4: perspective projection.

front of the camera, so that the plane is given by $\mathbf{x_p} = (x_p, y_p, -D)$. Yes, that's a negative z_p, because we're looking in the negative z-direction. The projected coordinates are the intersections of the line between COP and $\mathbf{x_c}$, and the projection plane. Since the COP is at the origin, the relation between $\mathbf{x_c}$ and $\mathbf{x_p}$ is

$$(21.5) \qquad \lambda\, x_p = x_c$$

Here λ is a simple scaling factor, the value of which can be determined in a variety of ways, depending of the information available at the point in your derivations. For example, since $z_p = -D$, by definition, we have $\lambda = -z_c / D$. Later we'll see another expression. The interesting thing about this expression is that λ is proportional to the distance in camera space, which in turn tells you how much the camera position is to be scaled *down*, or zoomed. This is useful, since the scaling parameters of the affine matrix scales down as well. Also, the distance D attenuates the scaling, which means that it acts as a ***focus length***. Note that when $z_c = -D$, the scale is one, meaning that the objects at this distance appear in their normal size.
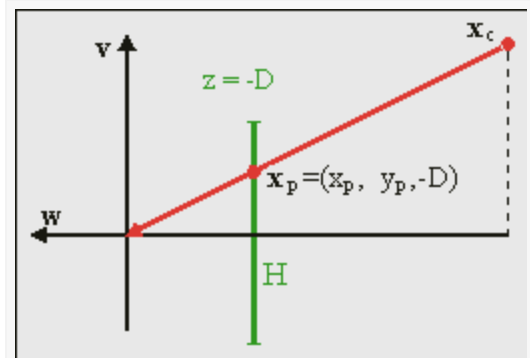
Viewport and viewing volume

Before I give the last step of the transformation to the screen, I have to say a few words about the viewport and the viewing volume. As you can imagine, you can only see a certain portion of the world. You see the world through a region called the *viewport*. This is an area on the projection plane, usually rectangular, that defines the horizontal and vertical boundaries of what you can see. In particular, you have a left side (*L*), right side (*R*), top (*T*) and bottom (*B*).

Fig 21.5: Viewing frustum in camera space. The green rectangle is the visible part of the projection plane (i.e., the screen).

With the axes defined as they are and the origin is usually centered (see fig 21.5, inset), we have *R>0>L* and *T>0>B*. Yup, in this particular case *L* is negative, and *T* is positive!

The width and height of the viewport are $W = |R-L|$ and $H = |B-T|$, respectively. Together with the center of projection, the viewport defines the *viewing volume* (see fig 21.5). For a rectangular viewport this will be a pyramid.

Most of the time you will want boundaries in depth as well, because things too near will obstruct everything else from view (besides, dividing by 0 is never good), and very distant objects will become so small that they are barely noticeable, and why waste so many calculations on a handful of pixels? These boundaries in depth are called the *near* (*N*) and *far* (*F*) planes, and will turn the viewing volume in a frustum. The numbers for these distances are a matter of taste. Whatever you use, be aware that the z-values are actually negative. I would prefer to have the values of *N* and *F* positive, so that the order or distance is 0>−*N*>−*F*.

Another point is the notion of the *field of view* (FOV). This is the horizontal angle α that you can see, meaning that
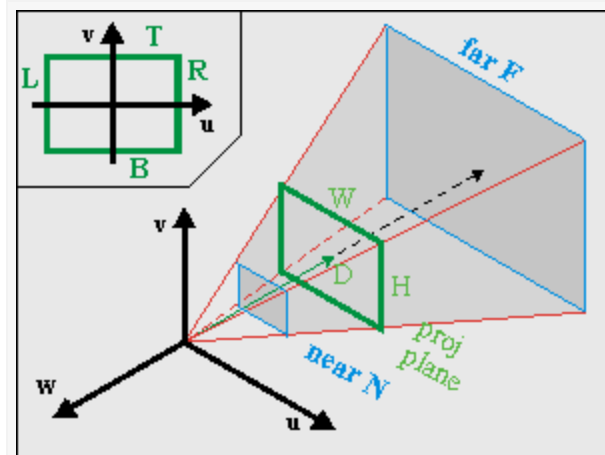
$$(21.6) \qquad \tan(\tfrac{1}{2}\,\alpha) = \frac{\tfrac{1}{2}\,W}{D}$$

I am told that a commonly used FOV is about 90°, which would require $D = \frac{1}{2}W$. With $D = 128$ you get close enough to this requirement, with the added benefit that it's a power of 2, but that, of course, is an implementation detail. *However*, it seems that $D = 256$ is more common, so we'll use that instead.

The screen

The last step is the one from the projection plane onto the screen. This step is almost trivial, but the almost can cause you a lot of trouble if you're not careful. The situation is shown in fig 21.6, where you are looking through the camera. The axes **u** and **v** are the up and right axes of the camera system, while the green arrows denote the x- and y-axes of screen space. And if you have paid attention to any of the tutorials, you should know

Fig 21.6: screen space vs camera space

that the screen's y-axis points *down*. This is bugfest number 1. Also, the origins of camera and screen space differ. Since the screen corresponds to the viewport, the origin of the screen in camera/projection space is $\mathbf{a}_{sp} = (L,\ T,\ -D)$. Be careful not to reverse the signs here; that would be bugfest number 2. Also remember that since this is in camera space, L is negative and T is positive. Taking both the inverted vertical axis and the origin of screen-space in mind, we have

$$(21.7) \qquad S(1,\,-1,\,1)\cdot x_s = x_p - a_{sp}$$

The scaling matrix reverses the sign of the y-axis. We could have avoided the extra matrix if we had rotated the camera frame by another 180°, in which case **v** would have pointed down and **w** would have pointed forward. But I didn't, so we'll have to live with it here. Also, since the origin of the screen in camera space, is $\mathbf{a}_{sp} = (L,\ T,\ -D)$, the screen position is $\mathbf{x}_s = (x_s, y_s, 0)$, in other words z_s is always zero. If you want to

check whether everything is OK, see if the corners of the viewport give the right screen coordinates.

Theory summary

And that's basically it, phew. Since it took three pages to get here, I'll repeat the most important things. First, the main equations we need are:

(21.8a)	$\lambda C \cdot x_p = x_w - a_{cw}$
(21.8b)	$S(1, -1, 1) \cdot x_s = (x_p - a_{sp})$

where

x_w	coordinates in world space;
x_p	coordinates on the projection plane, $x_p = (x_p, y_p, -D)$;
x_s	coordinates on the screen, $x_s = (x_s, y_s, 0)$;
a_{cw}	the location of the camera in world space;
a_{sp}	the location of the screen origin in camera space space, $a_{sp} = (L, T, -D)$;
C	the camera matrix, as function of pitch θ and yaw ϕ: $C = R_y(\phi) \cdot R_x(\theta)$;
λ	the scaling factor. Its value can be determined by the boundary conditions.

Remember these equations and terms, for I will refer to them often. The break between eq 21.8a and eq 21.8b is by design: all the real information is in eq 21.8a; eq 21.8b is just a final step that needs to be taken to complete the transformation. In the remainder of the text, I will make frequent use of eq 21.8a and leave out eq 21.8b unless necessary. Other interesting things to know:

- World system $S_w = \{i, j, k\}$ and camera system $S_c = \{u, v, w\}$ are right-handed Cartesian coordinate systems. As expected, the columns of camera matrix C are the principle axes of S_c: $C = [u\ v\ w]$;

- The viewport and viewing frustum are in camera space, meaning that their boundaries are too. This means that

$R > 0 > L$	(horizontal)
$T > 0 > B$	(vertical)
$0 > -N > -F$	(depth)

- If we use the GBA screen size as a basis ($W = 240$, $H = 160$), and $D = 256$, reasonable values for the viewing frustum boundaries are the following, but you can pick others if you want.

$L = -120$	$R = -120$
$T = 80$	$B = -80$
$N = 24$	$F = 1024$

Horizon and backdrop

Take the essential mode 7 case: a floor in perspective. Due to the perspective division, the distant parts of the floor will approach a single line: the *horizon*. Since the map really is just a floor, the horizon really will be just that: one horizontal line. The space above that is usually empty, but to make it a little less bland, we will use a *backdrop*: a panorama view of the distant environment that moves along with the camera's rotation.

Finding the horizon

Roughly put, the horizon is where $z = -\infty$. If you have lines on the floor, the horizon is where all parallel lines seem to meet: the vanishing line. Naturally, if you only have a floor, then you should only draw it below the horizon and the graphics above it should be part of a skybox. I'm sure you've seen this in the original Mario Kart and other

mode 7 racers. Since we're limited to a roll-less camera, the horizon will always be a horizontal line: one scanline $y_{s,h}$. To find it, all we have to do is take the y-component of eq 21.8a and rearrange the terms to get

$$
\begin{aligned}
\text{(21.9a)} \quad \lambda(v_y\, y_{p,h} - w_y D) &= -a_{cw,y} \\
y_{p,h} &= (w_y D - a_{cw,y}\, /\, \lambda)\, /\, v_y
\end{aligned}
$$

And if we were to take our horizon at infinity, them $\lambda = -\infty$, which would reduce eq 21.9 to

$$
\text{(21.9b)} \qquad y_{p,h} = D\, w_y\, /\, v_y = D\tan(\theta)
$$

However, you need to think about whether you want to use this simplified equation. At very large λ, the gaps in displayed map points are so large that you're effectively showing noise, which can be very ugly indeed. A better way would be making use of the far clipping plane at $z_c = -F$. In that case, $\lambda = F/D$ and we can use eq 21.9 to calculate the horizon, which will be something like

$$
\text{(21.9c)} \qquad y_{p,h} = D\, /\, F \cdot (Fw_y - a_{cw,y})\, /\, v_y
$$

As expected, if $F = -\infty$ then eq 21.9c reduces to eq 21.9b. Regardless of whether you chose a finite of infinite z_c, the horizon will be at scanline $y_{s,h} = T - y_{p,h}$.

Using the horizon

The horizon marks the line between the map and 'far far away': between the floor and the backdrop. The floor should be an affine background, obviously; for the backdrop, we will use a regular background, although that's not required. What we need to a way to switch between the two at the horizon scanline. The simplest way is by HBlank interrupt: once the horizon scanline is reached, make the switch between floor and backdrop settings in the BG control registers and perhaps initiate HDMA for the affine parameter transfers if you chose to use DMA for that.

Switching between the backdrop and floor backgrounds is actually trickier than it sounds. You could, for example, have a separate background for each and enable/disable them depending on your needs. The problem is that it seems to take about 3 scanlines before a background is fully set up in hardware (see forum:1303), so you'll see crap during that time. In other words, this solution is no good.

An other way would be to have one background for both and switch the video-mode from 0 to 1 or 2. This won't give you 3 lines of garbage, but now another problem arises: chances are very high that the backdrop and floor have very different tiles and map attributes. This is easy to solve though: simply change the screen (and char) base blocks in `REG_BGxCNT`.

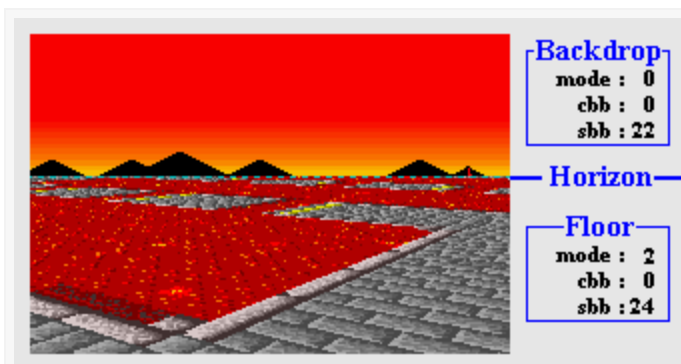**Fig 21.7**: Switch video-mode and background parameters at the horizon.
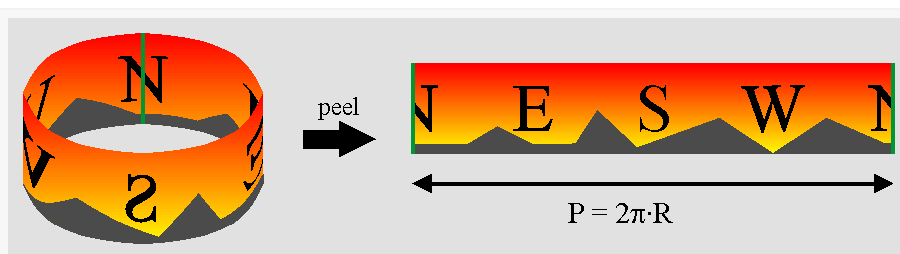

Fig 21.8: peeling a panoramic view from a cylinder.

Making and placing the backdrop

The space directly above the horizon is for the backdrop. You probably want a nice image of a distant town or tree line over there, not just a boring empty sky. The backdrop offers a panoramic view, which can be considered a map painted on the

inside of a cylinder and then peeled off to a normal 2D surface (see fig 21.8). The idea is to put that surface on a background and the scroll around.

Vertically, the bottom of the background should connect to the horizon. Because regular backgrounds use wrap-around coordinates this is actually quite easy: place the ground-level of the backdrop at the bottom of a screen-block and set the vertical offset to $-y_{s,h}$.

Horizontally, there are several issues to be aware of. The first is the width of the map, which is simply the perimeter P of the cylinder. As we should have a scrolled a full map's width for a 360° rotation, the correct scroll ratio per unit angle is simply $P/2\pi = R$, the radius. In principle, R is arbitrary, but the best result can be had when the field of view formed by the angle of the panorama ($\alpha_p = W/R$), is equal to the camera field-of-view angle α_c from eq 21.6. If all is right we should have $\alpha_p = \alpha_c = \alpha$.

$$
\begin{aligned}
\alpha &= 2 \cdot \arctan(\frac{\frac{1}{2}\,W}{D}) \\[2mm]
\alpha &= \frac{W}{R} \\[2mm]
R &= \frac{\frac{1}{2}\,W}{\arctan(\frac{\frac{1}{2}\,W}{D})} \\[2mm]
&\approx \frac{D}{1 - (\frac{\frac{1}{2}\,W}{D})^2 \;/\; 3}
\end{aligned}
\tag{21.10}
$$

That last approximation stems from the first couple of terms of the Taylor series of the arctangent. Interestingly enough, even $R \approx D$ seems somewhat adequate. Anyway, filling in $W = 240$ and $D = 256$ gives $P = 1720$, which isn't a very convenient map size, is it? Now, it is possible to create a map of any size and update VRAM if we go outside the screenblock's boundaries (commercial games do it all the time), but doing so would distract for the subject at hand, so you know what? We're going to bend the rules a bit and just force $P = 1024$.

"Wait a sec … you can't do that!" Well, yes I can actually. I'm not *supposed* to do it, but that's another issue. The fact of the matter is that, I don't think there is a *single* mode

7 game that scrolls the backdrop properly! For example, the Mario Kart's often use multiple backgrounds with different scrolling speeds in their backdrops, which is absolutely ridiculous, mathematically speaking, because looking around doesn't change relative lines of sight. But I guess nobody noticed or at least nobody cares. What I'm trying to say is: we're in good company $\boxed{:P}$

So, we just define a perimeter value and with it backdrop map-width ourselves. In this case I'm going to use P = 1024, which is a nice round number and for which we can use a 512 px wide tile-map will effectively end up as a panorama with 180° rotational symmetry. Taking into account the circle partitioning of $2\pi \Leftrightarrow 10000$h, the scrolling value is simply $\phi * P / 10000$h $= \phi / 64$. We'll have to offset this by L as well because I want to map $\phi = 0$ to due north. The final position of the backdrop is given in 21.11.

$$
(21.11) \qquad
\begin{aligned}
dx &= \varphi / 64 + L \\
dy &= -y_{s,h}
\end{aligned}
$$

The floor

Affine parameters for the floor

Eq 21.8 describes the world↔screen transformation but that information uses 3D vectors, while the GBA only has a 2×2 affine matrix **P** and a 2D displacement vector **dx** at its disposal. So we have some rewriting to do. Now, I could give you the full derivation, 2d↔3d conversions and all, but something tells me you really don't want to see that. So instead, I'll give you the set of equations you need to solve, and hints on how to do that.

$$
(21.12) \qquad
\begin{aligned}
\lambda C \cdot x_p &= x_w - a_{cw} \\
S(1, -1, 1) \cdot x_s &= (x_p - a_{sp}) \\
P \cdot q &= p - dx
\end{aligned}
$$

The first two equations are just eq 21.8 again, I just them list for completeness. The last equation is the relation between screen point **q** and map point **p** for an affine map, an equation that should be familiar by now. Now, remember that our map lies on the floor, in other words **p** = (x_w, z_w). The 2D screen point **q** is, of course, similar to the 3D screen vector of $\mathbf{x}_s$. The only thing that you have to remember is that when writing to `REG_BGxY`, the left of the current scanline is taken as the origin, so that effectively **q** = $(x_s, 0)$, which in turn means that p_b and p_d are of no consequence. The values of the other elements of **P** are simply the x- and z-components of the scaled camera x-axis, $\lambda\mathbf{u}$. If you use these values, you will see that eventually you will end up with an expression that can best be summed up by:

$$(21.13) \qquad dx' = a_{cw} + \lambda C \cdot b$$

where

$$dx' = (dx,\ 0,\ dy)$$
$$b = (L,\ T - y_{s'},\ -D)$$

Everything you need for the displacement is neatly packed into this one equation, now we need to disassemble it to construct the algorithm. First, we can use the y-component of **dx'** to calculate λ. Once we have that we can use it to calculate the other two elements, i.e., the actual affine offsets. The affine matrix was already given earlier.

Eq 21.14 gives all the relations explicitly, though I hope you'll forgive me when I prefer the conciseness of eq 21.13 myself.

$$(21.14) \quad \begin{aligned} \lambda &= a_{cw,y} \;/\; ((y_s - T)v_y + Dw_y) \\ p_a &= \lambda\, u_x \\ p_c &= \lambda\, u_z \\ dx &= a_{cw,x} + \lambda(Lu_x + (T - y_s)v_x - Dw_x) \\ dy &= a_{cw,z} + \lambda(Lu_z + (T - y_s)v_z - Dw_z) \end{aligned}$$

Note that if we take the top at 0 and no pitch (T=0 and θ=0) we have exactly the same result as in the first mode 7 chapter, and if we look straight down (θ=90°), the whole thing reduces to a simple scaling/rotation around point ($-L$, T), which is exactly it should be. Eq 21.14 is the general equation for mode 7; for the implementation, you can often make a number of shortcuts that speed up calculation, but well get to that later.

Distance fogging

In the real world, light coming from far away objects has to travel through the atmosphere, which scatters the photons, attenuating the beam. What you'll end up seeing is partly the object itself and partly the ambient color, and the further the original object, the smaller its contribution is. Because such effect is most easily visible in fog conditions, I'll call this effect *fogging*.

Fogging offers a hint of distance and including it can increase the sense of depth. Also, it can hide objects popping into view as they're loaded. GBA-wise, it can be implemented by using different alpha-blends at every scanline.

The fundamental equation for this is the following differential equation:

$$dI = -I\, k(\nu)\, \rho\, dz$$

where I is the intensity; $k(\nu)$ is the absorption coefficient of the medium, which depends on the frequency of the light, ν and possibly position; ρ is the density and z is the distance. Solving this would lead to an exponential decay over distance. And I do mean real distance, with squares and roots and everything.

Fortunately, we don't have to use something that complicated; all we really need is some functional relation that gives 0 at infinity and 1 close up. Interestingly enough, we already have something like that, namely λ as function of the scanline (see 21.14). This is basically a hyperbola, all you have to do then is fiddle with scalers and offsets a bit to get something that looks nice. In my case, λ*6/16 seems to work well enough.
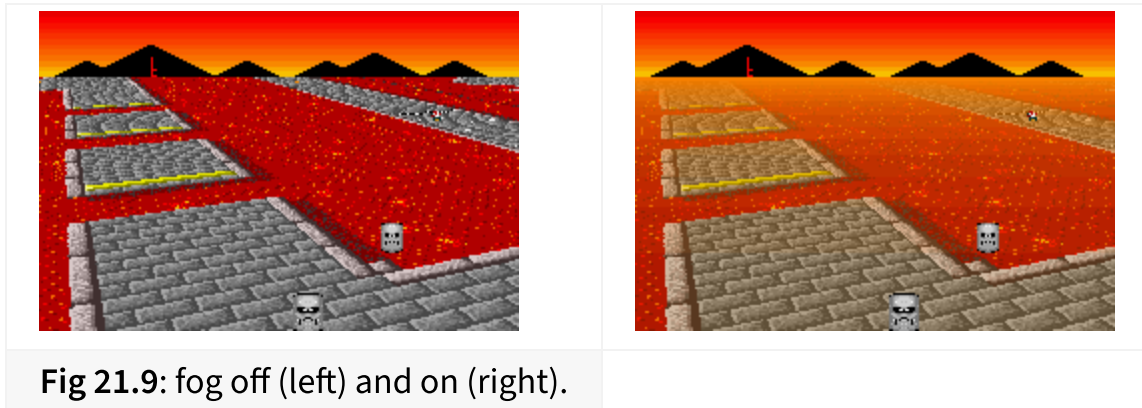
Fig 21.9: fog off (left) and on (right).

Fig 21.9 shows screenshots with and without the fogging effect as seen from a fairly high altitude. The distance to the floor is relatively small at the bottom of the screen, so those are still very visible. At the horizon, the floor is completely obscured by the orange fog; which is actually a good thing, as the lines near the horizon are usually not much to look at anyway.

By the way, note that I said *orange* fog. If you'd paid attention in the graphics effects chapter will know that the GBA only has fading modes for white and black. Nevertheless, fades to an arbitrary color are very much possible, but I'll explain once we get to the implementation. While you ponder over how it can be done, I'll move on to 3D sprites.

Sprites

Sprites and 3D are a strange combination. By their very nature, sprites are 2D objects – like stickers stuck against the viewport (i.e., the screen). To make them appear part of the 3D world, you have to make them move over the screen in such a way that they

appear to move with the world and scale them according to their distance. Once again, the basic of this is eq 21.8, but there is considerably more to it.

Four topics must be covered here. The first is sprite **positioning**. Eq 21.8 will work at point/pixel level, and a sprite is a simple rectangle. While it's possible to rewrite the sprite's pixels to work around that, it kind of defeats the purpose of using sprites in the first place. Instead, we'll link one point on the object to the world coordinate of the sprite and set the OAM position and matrix to accommodate this. This is basically the theory of anchoring discussed in the affine object chapter.

Next up: sprite **culling**. Once you have the correct OAM positions you can't use them as is, you have to make sure the sprite is only active if it is actually visible inside the viewport. If not, it should be disabled.

Then there's the matter of sprite **animation**. Consider Toad's kart in fig 21.10, which has the correct anchored position, but no matter which angle you look at it, it'll always show the same side. To make it look as if you can actually move around the object, we'll use different frames of animation to show different sides.

Lastly, sprite **sorting**. By default, objects will be ordered according to the objects' numbers: obj 0 over obj 1, over obj 2, etc. Always linking a sprite to the same object means that the order would be wrong if you look at them from the other side, so we need to sort them by distance.

Those are the main issues to deal with. There are a few others, like placing a shadow, and using pre-scaled objects to get around the hardware limitation of 32 affine matrices, but these are fairly easy if the other points are already taken care of. One thing I will discuss as well is what I call object **normalization**: applying an extra scaling for objects so that they don't grow too big for their clipping rectangle.

Fig 21.10: anchored sprite. The position is good, but no matter how you turn, Toad always turns away. Maybe it's the hat.

Positioning and anchoring

Positioning sprites consists of two facets. The first is to transform the sprites world position x_w to a position on the screen x_s. After that, you need to use that point to determine the most appropriate OAM coordinates.

The first part is just another application of eq 21.8 again, only in reverse. Normally, inverting 3D matrix is a particularly un-fun process, but the camera matrix happens to be an orthonormal matrix. An *orthonormal matrix* is a matrix of which the component vectors are orthogonal (perpendicular to each other) and have a length of 1. The neat thing about an orthonormal matrix is that its inverse is simply its transpose: $C^{-1} = C^T$. That leads us to the following equations:

$$(21.15) \quad \begin{aligned} x_p &= C^T \cdot (x_w - a_{cw}) / \lambda \\ x_s &= S(1, -1, 1) \cdot (x_p - a_{sp}) \end{aligned}$$

The only real unknown here is λ, which we can calculate by using the fact that $z_p = -D$. Now let the distance between camera and sprite be $\mathbf{r} = \mathbf{x_w} - \mathbf{a_{cw}}$; using $\mathbf{C} = [\mathbf{u}\ \mathbf{v}\ \mathbf{w}]$, we find

$$\begin{aligned} \lambda &= -w \cdot r / D \\ x_p &= u \cdot r / \lambda \\ y_p &= v \cdot r / \lambda \end{aligned}$$

Finding the screen position of x_W is trivial after that. And now the anchoring part. Instead of stickers, think of objects as pieces of pater to be tacked onto a board (i.e., the screen). The tack goes through one spot of the object, and that spot is fixed to the board. That spot is the *anchor*. For affine objects it's not quite as simple as that, because we have to specify OAM coordinates rather than anchor coords, so there is some math involved in how to express the OAM coordinates **x** in terms of the texture anchor p_0 and the screen anchor q_0. This theory was covered in the affine object chapter, which led to eq 21.16. The other quantities there are size of the objects, **s** = (w, h), and m which is ½ for normal affine objects and 1 for double-size affine objects.
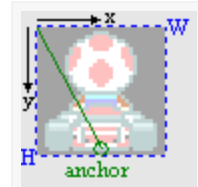
Fig 21.11: a 32×32 sprite, with the anchor p_0 relative to the top-left.

$$(21.16) \qquad x = q_0 - ms - P^{-1} \cdot (p_0 - \tfrac{1}{2}\,s)$$

Now the task is to link the data we have to this equation. The screen anchor q_0 is just x_S. The texture anchor p_0 is the pixel in texture space you want to keep fixed and is yours to choose. For the kart-sprite, it makes sense to put it near the bottom of the kart, as is depicted in fig 21.11. 'Vector' **s** is given by the size of the object, which in this case is (32, 32) and because I'm choosing to always use double-size objects here, m=1. The **P**-matrix is just a scaling by λ, unless you want to add other things as well. All that remains then is just to fill it in the numbers.

Sprite culling

Culling is the process removing any part of the world that cannot be seen. In this case, it means removing those sprites that do not fall within the viewing volume. This is a very smart thing to do, and it makes even more sense for sprites, because not doing so would seriously screw things up because OAM couldn't cope with the possible range of x_S.

The first thing to do would be a distance check: if the object is too far away, it should not be seen. It's also a good idea to have a near-plane distance check. Then you have

to test it for intersections with the viewport. Each sprite is bounded by a certain rectangle on the projection plane and if this is completely outside the viewport, the object should not be rendered.

Fig 21.12 shows a few examples of this. Objects *a* and *b* have already been projected onto the projection plane. Object *a* is outside the viewport, and should be disabled. Object *b* is partially

Fig 21.12: View-frustum with sprites *a*, *b* and *c*. *b* and *c* are visible, *a* is not.

visible and should be rendered. Object *c* is not projected yet, but falls between the near and far plane and should at least be tested (and then found fully visible).

It's actually easier to do the view volume checks in 3D camera space instead of 2D projection space. The object rectangle can easily be calculated from $\mathbf{x}_c = \mathbf{C}^T \cdot \mathbf{r}$, the anchor $\mathbf{p}_0$ and the size $\mathbf{s}$. The viewport will have to be scaled by λ, and this gives us the following rests to perform:

	Object position	Visible if
Depth	$d = -z_c = w \cdot r$	$N \le d \ \&\& \ d < F$
Horizontal	$l = x_c - p_{0,x}$	$\lambda L \le l + w \ \&\& \ l < \lambda R$
Vertical	$t = -y_c - p_{0,y}$	$\lambda T \le t + h \ \&\& \ t < -\lambda B$

Table 21.1: Object rect and culling tests in camera space. Note the signs!

If all these conditions are true, then the object should be visible. Now, please note the *signs* of the tests, particularly in the vertical checks.

Animation

Rotation animation, to be precise. As we saw in fig 21.10, the sprite will show the same side regardless of where you are looking from. This is only logical, as the sprite is not actually a 3D entity. To make it *look* a little more 3D, we need to have images of the sprite taken from different camera angles, and then pick the one we need depending on which angle we're looking from.
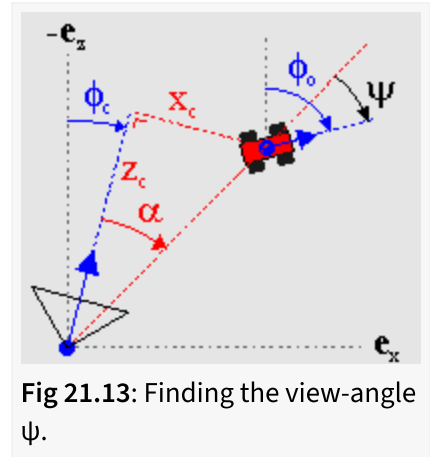
Fig 21.13: Finding the view-angle ψ.

First, finding the correct view angle, ψ. Fig 21.13 shows the general situation. The angle you need is the angle between the vector between the camera and the object (red, dashed) and the global looking direction of the object. In the figure, you can see the global direction angles for the camera and object: ϕ_c and ϕ_o, respectively. Also indicated is the angle between the camera direction and the sprite, α. If you look at these angles closely, you'll see that $\phi_c + \alpha + \psi = \phi_o$. In other words:

$$\begin{aligned}(21.17) \qquad \psi \quad &= \varphi_0 - \varphi_c - \alpha \\ &= \varphi_0 - \varphi_c - \arctan(x_c\ /\ -z_c)\end{aligned}$$

Whether the minus-sign inside the arctan() is necessary depends on how you define the terms all the terms. Eq 21.17 is the fully correct version, but if the arctan doesn't appeal to you, you'll be glad to know that in most cases the α-term can be safely ignored without anyone noticing.

Now that we have our viewing angle, we need to use it somehow. Suppose you have *N* frames of rotation, which divides the circle into equal parts each $2\pi/N$ radians wide. To get the slice that ψ is in, we merely have to divide by the angle of each slice: $i = \psi/(2\pi/N) = N\cdot\psi/(2\pi)$. If you have defined your circle in power-of-two divisions (which we have) then this part is ridiculously easy: just use a right-shift. Once you have the frame-index, the rest should be easy. Mostly. There are some intricacies that

that can fog things up, but those are implementation-dependent and will be saved for later.

Sprite sorting

Disregarding priority bits for the moment, the order of objects on-screen is determined by the object number: a lower number will be in front of higher numbers. In 2D games, you can often ignore this because sprites will be on the same layer; in 3D games, you really, really can't. Take fig 21.14, for example. The four thwomps here have a specific object order. In the left picture, the closest thwomp happens to have the lowest object and the visual ordering is correct. When viewed from the other side, however, (middle picture) things are a little different. There are two visual cues for depth: scaling (more distance is smaller) and occlusion (distance objects are obscured by closer objects). In the middle picture, these two conflict because the closest object has the *highest* number, making the overall picture a little disconcerting. In the picture on the right, everything looks okay again, because steps were taken to ensure the correct object order.
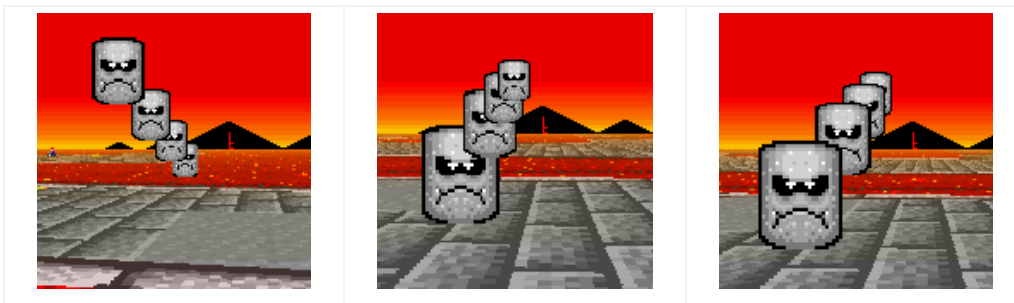
Fig 21.14. Non-sorted objects look alright (left) from one angle, but not from the other way (middle). You need to sort them to get the correct order (right).

What needs to be done is sort the objects in OAM according to depth; a kind of Z-buffer for objects. The depth of a sprite is simply z_c, and we need to fill OAM with the sprite's object attributes in order of ascending z_c. For good measure, it's probably a good idea to give hidden objects the maximum depth-value possible or to leave them out of the sorting process entirely.

There are many possible strategies for sorting the objects. My own choice aright now would be to not sort the sprites or objects directly but to create an **index table**, which indicates the order the sprites' attributes should go into OAM. The pseudo-code for this is given below. Which algorithm you use to sort the keys doesn't really matter at this time, as long as it does the job. I'm sure that faster methods can be found, but probably at the expense of more code and I want to keep things relatively simple.

```
// Pseudo code for sorting sprites for OAM
void spr_sort()
{
    int ids[N];     // Index table
    int keys[N];    // Sort keys

    // Create initial index and sort-key table
    for ii=0; ii<N; ii++)
    {
        ids[ii]= ii;
        keys[ii]= is_visible(sprite[ii]) ? sprite[ii].depth : DEPTH_MAX;
    }

    // Sort keys (i.e., fill ids)
    id_sort(ids, keys);

    // Fill OAM according to
    for(ii=0; ii<N; ii++)
        oam_mem[ii]= sprite[ids[ii]].obj;
}
```

Renormalization

I wouldn't be surprised if you've never heard of this term before. *Normalization* means that you scale a quantity to a user-friendly value – usually 1. You have already scaled the sprite by a factor λ, but that's not enough. In most cases, you have to scale it further, i.e *renormalize* it. Here's why.

By definition, the scaling factor λ will be one when $z_c = -D$. Now consider what happens if you look at a closer object, say at $z_c = -\frac{1}{2}D$. In this case, λ will be ½ and the object will be scaled by a factor of two. In other words, it'll already fill the double-size

canvas. And higher scales are possible too: with the suggested values of $D = 256$ and $N = 24$, you could end up with scaling of 10! This will not do.

It is possible to get around this by moving the near-plane further away. However, then you'll see object disappearing if they're still quite far off, which will look just as strange as seeing them clipped. A better solution is to give the objects an extra scaling factor. In `m7_ex` I have scaled the objects by an additional factor of ¼, so that a 32x32 sprite is actually only 8x8 'world'-pixels in size. This seems to work out quite nicely.

This renormalization means that you're actually working with *two separate* scaling factors: one for the coordinate transformation, and one for visual effects. It is the *visual* scaling you need to use in positioning and culling the sprites, not the transformation scaling; the latter's influence stops once you've found the screen-position of the anchor.

There's probably an official term for this process, but I wouldn't know what it is. I'm familiar with the process of renormalization from physics (a few Dutch professors got the Nobel Prize for this subject a few years back) and it seemed to fit. If you know the official term, I'd like to hear it.

Fig 21.15. Object renormalization. Left: normal (ew, no!).
Middle: ×½ (hmmm, no). Right: ×¼ (yeah, that's it).
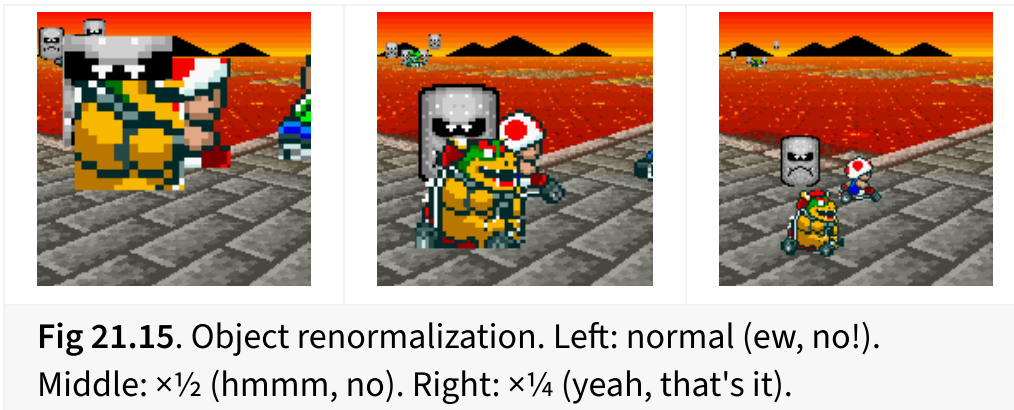
And with that, we've reached the end of the theory. Now to actually implement all of this.

Implementation

Design considerations.

My aim here is not to merely dish out a couple of functions that can make mode 7 happen, but also provide something that can be easily modified if necessary. The code of the `m7_ex` demo is spread over 4 files: one for the things specific to the demo itself `m7_ex.c`; and three for mode 7 specific stuff, `mode7.h`, `mode7.c` and `mode7.iwram.c`. Yes, iwram functions too; some of these things are going to be computation extensive and I want them as fast as possible right off the bat. I'm also borrowing the object sorter from the priority demo.

There are three main areas of interest here: the **camera**, **background stuff** and **sprites**. For each of these we will use a struct and/or array to keep their data so it's nice and OOPy. There will also be a sort of manager struct for the mode 7 stuff as a whole. And, of course, we need constants for the view volume, focus length and a few other items. A handful of functions will then operate on these items to give up what we need.

Constants

There aren't too many constants. Most have to do with the viewport, the others with the focus and renormalization.

```c
#define M7_D          256     //!< Focal length
#define M7_D_SHIFT      8     //!< Focal shift
#define M7O_NORM        2     //!< Object renormalization shift (by /4)

// View frustum limits
#define M7_LEFT      (-120)      //!< Viewport left
#define M7_RIGHT      120        //!< Viewport right
#define M7_TOP         80        //!< Viewport top (y-axis up)
#define M7_BOTTOM    (-80)       //!< Viewport bottom (y-axis up!)
#define M7_NEAR        24        //!< Near plane (objects)
#define M7_FAR        512        //!< Far plane (objects)

#define M7_FAR_BG     768        //!< Far plane (floor)
```

Structs and variables

Mode 7 would be a wonderful place to use classes, but since I'm working in C, not C++, I'm sticking to structs. Apart from the `BG_AFFINE` struct I presented in the affine background page, you need one struct for the camera and one struct for the mode 7 objects. I'm also using a mode 7 container struct to keep track of all the parts that go into the mode 7 functionality, so that you won't have loose global variables lying around the place.

You're free to create your own structs for these, but the ones I will use are given below. If you've been paying attention, most of the members should be familiar. Oh, the `POINT` and `VECTOR` structs are 2D and 3D vectors, of course.

```
//! 3D sprite struct
typedef struct M7_SPRITE
{
    VECTOR pos;       //!< World position.
    POINT anchor;     //!< Sprite anchor.
    OBJ_ATTR obj;     //!< Object attributes.
    s16 phi;          //!< Azimuth angle.
    u8 obj_id;        //!< Object index.
    u8 aff_id;        //!< OBJ_AFFINE index.
    TILE *tiles;      //!< Gfx pointer.
    VECTOR pos2;      //!< Position in cam space (subject to change)
} M7_SPRITE;

//! 3D camera struct
typedef struct M7_CAM
{
    VECTOR pos;       //!< World position.
    int theta;        //!< Polar angle.
    int phi;          //!< Azimuth angle.
    VECTOR u;         //!< local x-axis (right)
    VECTOR v;         //!< local y-axis (up)
    VECTOR w;         //!< local z-axis (back)
} M7_CAM;


//! One struct to bind them all
typedef struct M7_LEVEL
{
    M7_CAM *camera;         //!< Camera variables
    BG_AFFINE *bgaff;       //!< Affine parameter array
    M7_SPRITE *sprites;     //!< 3D sprites
    int horizon;            //!< Horizon scanline (sorta)
    u16 bgcnt_sky;          //!< BGxCNT for backdrop
    u16 bgcnt_floor;        //!< BGxCNT for floor
} M7_LEVEL;
```

There's not much more I have to say about these structs. The `M7_SPRITE` has the attributes of its object as a member itself, rather than an index or pointer to any sort of buffer. The reason behind this is essentially "why the hell not". Because I have to sort the objects anyway, using an extra buffer might not be worthwhile, so I chose this. I'm also keeping track of the position in camera space because I need it on more than one occasion, and a TILE pointer for graphics. The reason for this will become apparent when we implement animation.

The `M7_LEVEL` holds pointers to the main variables for mode 7 (the camera, affine array and sprites) as well as the horizon scanline needed to switch from backdrop to floor, and two variables containing the data of the bg control register, as this will be different for the backdrop and floor.

Now we need these four variables using these structs. Because these are technically part of the demo itself, I've put them in `m7_ex.c` instead of the main mode 7 code, although that code does require an actual `m7_level` variable to exist for the HBlank interrupt. `SPR_COUNT` is the number of sprites, which is *definitely* demo specific. There are 161 entries in `m7_bgaffs` rather than just 160 for the same reason as in the DMA demo: HBlank sets up the next line, rather than the current one, and having this is better (and faster) than the alternative with if/else blocks.

```
M7_CAM m7_cam;
BG_AFFINE m7_bgaffs[SCREEN_HEIGHT+1];
M7_SPRITE m7_sprites[SPR_COUNT];

M7_LEVEL m7_level;
```

> ### TYPE AND ORDER OF STRUCT MEMBERS
>
> My usual advice is to use ints for your data types, but for structs this may not always be the best thing to do. Local variables may not use up memory, but structs do. And when you have arrays of structs, the extra space that word-sized members cost adds up quickly. So in that case feel free to use non-ints.
>
> Having said that, when it's time to use those members it can pay to copy its data to a local 32bit variable, rather an using a byte or halfword member for all the calculations.
>
> Also, and this is *very* important, you won't be saving any space if you don't pay attention to the order of the members. An int will still require word-alignment, even when it comes right after a byte member. The compiler may add padding after bytes and halfwords to ensure the next member is correctly aligned. It'd

> be best if you ordered the members in such a way that there's as little padding as possible.

Background functions

These are my four main background functions:

- `void m7_prep_horizon(M7_LEVEL *level)`. Calculates the horizon scanline.
- `IWRAM_CODE void m7_prep_affines(M7_LEVEL *level)`. Calculates the affine parameters for the floor, based on camera position and orientation..
- `void m7_update_sky(const M7_LEVEL *level)`. Positions the backdrop.
- `IWRAM_CODE void m7_hbl_floor()`. HBlank interrupt routine. Switches to mode 2 when necessary and copies affine parameters and creates fog effect.

`m7_prep_horizon()` and `m7_update_sky()` are simple implementations of eq 21.9 and eq 21.17, respectively, so I can be brief with these.

```
//! Calculate the horizon scanline
void m7_prep_horizon(M7_LEVEL *level)
{
    int horz;
    M7_CAM *cam= level->camera;

    if(cam->v.y != 0)
    {
        horz= M7_FAR_BG*cam->w.y - cam->pos.y;
        horz= M7_TOP - Div(horz*M7_D, M7_FAR_BG*cam->v.y);
    }
    else    // looking straight down (w.y > 0) means horizon at -inf
scanline
        horz= cam->w.y > 0 ? INT_MIN : INT_MAX;

    level->horizon= horz;
}

//! Update sky-bg position
void m7_update_sky(const M7_LEVEL *level)
{
    REG_BG2HOFS= (level->camera->phi>>6)+M7_LEFT;
    REG_BG2VOFS= -clamp(level->horizon, 0, 228)-1;
}
```

The horizon calculation makes use of a clipping far-plane, though this is not strictly necessary. If you want the horizon at infinity, remove the subtraction by the camera's height and use `M7_FAR_BG` = 1. Note the check for $v_y = 0$. As $v_y = \cos(\theta)$, this will be true when looking straight up or straight down. The distinction is important because sees the sky (no affine bg) and one sees only floor (no backdrop). Technically these should be ±infinity, but as this is fixed-point, `INT_MIN/MAX` will have to do.

As for the backdrop placement: I'm taking a *lot* of shortcuts here. A mathematically correct backdrop would use a background map 1720 pixels wide. It can be done, but mostly it's just annoying. Instead, I'm using a 512x256p regular background and use $P = 1024$ in the angle→scroll-offset conversion. This means the map shows up twice in one 360° rotation and that the *dx* is just φ/64. Yes, the floor and backdrop field-of-view will be slightly out of sync, but you'll only notice if you know what to look for, so that's alright.

Strictly speaking, the vertical offset should be bgHeight − horizon, but the bg-height can be ignored due to wrapping. The reason I'm also clamping the horizon to the size of the viewport is because the horizon scanline can become very large – the tan(θ) in it will approach infinity when looking up, remember? If you don't clamp it you'll scroll through the whole backdrop map a couple of times when panning up, which just looks awful.

Preparing the affine parameter table

Calculating the affine parameters happens in `m7_prep_affines()`. You could try to do this in the HBlank isr, but because it requires a division, it would simply take too long. Also, doing it in one spot is more efficient, as you only have to set-up the variables once. This routine carries out the calculations of eq 21.14. It has to do quite a number of calculations for each scanline, including a division, so you can expect it to be rather costly; which is why I'm putting it in IWRAM right from the start.

Now, you don't have to calculate things for every scanline: just the ones below the horizon. As for implementing eq 21.14 itself: it turns out that it works much better if you take the camera matrix apart again and work with sines and cosines of θ and φ, rather than the nine matrix entries. This next paragraph will explain how, but feel free to skip it and go onto the code.

Remember that the camera matrix is $\mathbf{C} = \mathbf{R}_y(\phi) \cdot \mathbf{R}_x(\theta)$; and that λ and $\mathbf{dx}$ are calculated via eq 21.13: $\mathbf{dx'} = \mathbf{a}_{cw} + \lambda \cdot \mathbf{C} \cdot \mathbf{b}$. You can break up $\mathbf{C}$ can combine it with $\mathbf{b}$ to form $\mathbf{b'} = \mathbf{R}_x(\theta) \cdot \mathbf{b}$. This new vector takes care of the pitch entirely – it's as if we only had a rotation around the vertical axis, i.e., the case discussed in the previous chapter. With this pre-rotation, the code becomes simpler and faster.

```c
IWRAM_CODE void m7_prep_affines(M7_LEVEL *level)
{
    if(level->horizon >= SCREEN_HEIGHT)
        return;

    int ii, ii0= (level->horizon>=0 ? level->horizon : 0);

    M7_CAM *cam= level->camera;
    FIXED xc= cam->pos.x, yc= cam->pos.y, zc=cam->pos.z;

    BG_AFFINE *bga= &level->bgaff[ii0];

    FIXED yb, zb;             // b' = Rx(theta) *  (L, ys, -D)
    FIXED cf, sf, ct, st;    // sines and cosines
    FIXED lam, lcf, lsf;     // scale and scaled (co)sine(phi)
    cf= cam->u.x;        sf= cam->u.z;
    ct= cam->v.y;        st= cam->w.y;
    for(ii= ii0; ii<SCREEN_HEIGHT; ii++)
    {
        yb= (ii-M7_TOP)*ct + M7_D*st;
        lam= DivSafe( yc<<12,  yb);       // .12f

        lcf= lam*cf>>8;                    // .12f
        lsf= lam*sf>>8;                    // .12f

        bga->pa= lcf>>4;                   // .8f
        bga->pc= lsf>>4;                   // .8f

        // lambda·Rx·b
        zb= (ii-M7_TOP)*st - M7_D*ct;    // .8f
        bga->dx= xc + (lcf>>4)*M7_LEFT - (lsf*zb>>12);  // .8f
        bga->dy= zc + (lsf>>4)*M7_LEFT + (lcf*zb>>12);  // .8f

        // hack that I need for fog. pb and pd are unused anyway
        bga->pb= lam;
        bga++;
    }
    level->bgaff[SCREEN_HEIGHT]= level->bgaff[0];
}
```

We begin by getting the scanline to begin calculating at (which may be nothing), and defining *lots* of temporaries. Not all of the temporaries are necessary, but they make the code more readable. Names aside, the code within the loop is very similar to that of `hbl_mode7_c` in the first mode 7 demo, except that in calculating λ we use a rotated y_s-value, and in calculating the offsets a rotated z_s ($= -D$) value. Annnd, that's it.

The comments behind the calculations indicate the fixed-point count of the results, which in this case can be either .8f or .12f. Now hear this: it is **very** important that the scaled (co)sine of φ, `lcf` and `lsf` , use 12 bits of precision or more. I've tried 8, it's not pretty – the displacements are all off at close range. Secondly, note the order of multiplications and shifts in the displacements; it is also very important that these stay the way they are. Particularly the one with *L*: the multiplication by `M7_LEFT` **must** happen after the shift, trust me on this.

The last interesting point is the line after the loop, which copies the parameters for scanline 0 to the back of the array to compensate for the HBlank-interrupt obiwan error.

This function is probably as fast as you can make it in C, and it the compiler does its job pretty well so there is little to be gained by going to manual assembly. This does not mean it doesn't still take quite some time. The division alone costs something like 100 to 400 cycles (the cycle-count for BIOS division is roughly 90 + 13/significant bit). At one division per scanline, this can really add up. The best strategy to deal with this is to *not do it* if you don't have to. If you use a fixed pitch angle, you can precalculate all the divisions and just look them up. If you must have a variable pitch, you can also go the trig way. Look back at fig 21.14. If β is the angle between $(0, y_p, -D)$ and $(0, 0, -D)$, then $\tan(\beta) = y_p/D$. With a good deal of trigonometry, you could rewrite the formula for λ to

$$(21.18) \qquad \lambda \; = a_{cw,\, y} \; / \, D \; \cdot \cos(\beta) \,/ \sin(\theta + \beta)$$

You can get β via an arctan LUT of 160 entries, one for each scanline (hey, you could even put that into p_d!), and then use a 1/sine LUT. You have to be careful to use large enough LUTs, though. Since the arguments of LUTs are integers, β will be truncated, and you will lose a *lot* of accuracy though this, especially near the horizon. Now, I haven't actually tried the trig-way yet, but I have done some basic tests in Excel which would suggest that with a 1/sine LUT of 512/circle, you'd get λ-errors well over 10% near the horizon, and errors around 1% everywhere else. With that in mind, I'd

suggest 1024/circle at least. Or interpolating between LUT entries, which you can do with libtonc's `lu_lerp16()` and `lu_lerp32()` functions.

Aside from going triggy with it, you can probably speed up the division as well in a number of ways. But before you go and optimize this, ask yourself if you really need it first. Premature optimization is the root of all evil, after all.

> ### SPEED-UPS FOR AFFINE CALCULATIONS
>
> Tried three optimizations recently. First, ARM/IWRAM, which brings the thing down to 23k-58k cycles. Then, a little refactoring that presented itself in a discussion with sgeos: the camera vectors can resolve to a smaller set of variables, saving 10-20%. Then, the trig thing, which can bring the whole thing down to 10-20k or even 7k-14k max, depending on whether you get $\cos(\beta)$ and $1/\sin(\theta+\beta)$ via large luts, or smaller luts and linear interpolation. Once you get the math, shifts, and signs in order, it works like a charm.

The mode 7 HBlank interrupt routine

To keep things simple, nearly everything that has to happen during VDraw happens inside one HBlank isr called `m7_hbl_floor()`. Earlier versions of this demo used a system of VCount/HBlank interrupts, but that turned out to be more trouble than it's worth. This is also an IWRAM routine because it really needs to be as fast as possible. The interrupt service routine does the following things:

1. **Check vcount for floor-range**. If this scanline is not part of the floor, return.
2. **Check vcount for horizon**. At reaching the horizon scanline the video mode should change and `REG_BG2CNT` should be set to the floor's settings.
3. **Copy affine parameters to `REG_BG_AFFINE[2]`**. Copy the *next* scanline's parameters to `REG_BG_AFFINE[2]`, as we've already past the current scanline.
4. **Fogging**. Fade to orange in this case.

```c
// from tonc_core.h
//! Range check; true if xmin<=x<xmax
#define IN_RANGE(x, min, max)  ( (x) >= (min) && (x) < (max) )
```

```c
IWRAM_CODE void m7_hbl_floor()
{
    int vc= REG_VCOUNT;
    int horz= m7_level.horizon;

    // (1) Not in floor range: quit
    if(!IN_RANGE(vc, horz, SCREEN_HEIGHT) )
        return;

    // (2) Horizon: switch to mode 1; set-up bg control for floor
    if(vc == horz)
    {
        BF_SET(REG_DISPCNT, DCNT_MODE1, DCNT_MODE);
        REG_BG2CNT= m7_level.bgcnt_floor;
    }

    // (3) Looking at floor: copy affine params
    BG_AFFINE *bga= &m7_level.bgaff[vc+1];
    REG_BG_AFFINE[2] = *bga;

    // (4) A distance fogging with high marks for hack-value
    u32 ey= bga->pb*6>>12;
    if(ey>16)
        ey= 16;

    REG_BLDALPHA= BLDA_BUILD(16-ey, ey);
}
```

Points (3) and (4) could benefit from a bit more explanation. As mentioned several times now, the isr of any scanline *vc* should set-up the parameters of *next* scanline, which is why we're copying from `level.bgaff[vc+1]` rather than just `[vc]`. Scanline zero's uses the set from *vc* = 160, which is alright because we've copied zero's data to the last element in the array. As usual, struct copies ftw.

For the fogging I use p_b which filled with λ in `m7_prep_affines()` for this very reason.

A scaled λ is not the most accurate model for fogging, but the effect looks well enough. Because the blending registers cap at 16, I need to make sure it doesn't wrap around at higher values.

This *still* leaves the question of what I'm actually blending with, as orange isn't part of the GBA's fade repertoire. At least, not *directly*. It is, however, quite possible to blend with the backdrop, which just shows bg-color 0. This color can be anything, including orange.

Sprites and objects

Sprite and object handling has been distributed over the following three functions:

- `void update_sprites()`. This is the main sprite handler, which calls other functions to do positioning, sorting and animation.
- `IWRAM_CODE void m7_prep_sprite(M7_LEVEL *level, M7_SPRITE *spr)`. This calculates the correct position and scale of the sprite.
- `void kart_animate(M7_SPRITE *spr, const M7_CAM *cam)`. This selects the correct frame for rotating around the karts.

Only `m7_prep_sprite()` is actually part of the mode 7 functions; the others could very well differ for every mode 7 game you have in mind. The main sprite handler, `update_sprites()`, is pretty simple: it needs to call `m7_prep_sprite()` for each sprite and create the sprite's sorting key, sort all the sprites and copy the sorted attributes to OAM. It also calls `kart_animate()` for each kart-sprite for their animations; if I had animations for the thwomps or other sprites they'd probably go here as well.

```
void update_sprites()
{
    int ii;

    M7_SPRITE *spr= m7_level.sprites;
    for(ii=0; ii<SPR_COUNT; ii++)
    {
        m7_prep_sprite(&m7_level, &spr[ii]);

        // Create sort key
        if(BF_GET2(spr[ii].obj.attr0, ATTR0_MODE) != ATTR0_HIDE)
            sort_keys[ii]= spr[ii].pos2.z;
        else
            sort_keys[ii]= INT_MAX;
    }

    // Sort the sprites
    id_sort_shell(sort_keys, sort_ids, SPR_COUNT);

    // Animate karts
    for(ii=0; ii<8; ii++)
        kart_animate(&spr[ii], m7_level.camera);

    // Update real OAM
    for(ii=0; ii<SPR_COUNT; ii++)
        obj_copy(&oam_mem[ii], &spr[sort_ids[ii]].obj, 1);
}
```

Most of the code has to do with sorting the sprites, which was already described in the theory. The `pos2` member of the sprites is set by `m7_prep_sprite()` to contain the sprite's position in camera space. The sorting routine `id_sort_shell()` is the index-table sorter described in the priority section.

If I had wanted to have more advanced animation or sprite things, they'd be put here as well. But I didn't, so I haven't.

Sprite positioning and scaling

The function `m7_prep_sprite()` calculates the correct on-screen position for a sprite, sets up the affine matrix with the proper (renormalized) scales and hides the sprite if it falls outside the view volume.

The first step is convert to the convert the world-position of the sprite to a vector in the camera space, using the first part of eq 21.15: $\mathbf{x}_c = \mathbf{C}^T \cdot \mathbf{r}$, with $\mathbf{r}$ being the position of the sprite relative to the camera: $\mathbf{r} = \mathbf{x}_w - \mathbf{a}_{cw}$. This is put into variable `vc`, but with the signs of y and z switched! This makes subsequent calculations a little easier. This vector is also stored in `spr->pos2`, which is used in sorting elsewhere.

The second step is checking whether the sprite would actually be visible, using the conditions from table 21.1, with one exception: the checks now use the *renormalized* rectangle of the sprite. Leaving that part out could create artifacts for some orientations. To calculate the sprite rectangle I'm using the sizes of the object rectangle. It is possible to get a tighter fit if you'd also define a sprite rectangle indicating the visible pixels within the object frame, but that might be going a little too far here.

Note that most of the code from the bounds checks on is done in a `do-while(0)` loop. This pattern is sort of a poor-man's `try/catch` block – I *could* have used `goto`s here, but as they're considered harmful I decided against it. Anyway, an out-of-bounds 'exception' here would indicate that the sprite should be hidden, which is done in step (5).

If we've passed the bounds-checks, we need to set-up the affine matrix and calculate the object's position via the anchoring equation of eq 21.16.

```c
//! Setup an object's attr/affine with the right attributes
/*! \param level    Mode 7 level data
*   \param spr      3D sprite to calculate for
*/
IWRAM_CODE void m7_prep_sprite(M7_LEVEL *level, M7_SPRITE *spr)
{
    M7_CAM *cam= level->camera;
    VECTOR vr, vc;        // Difference and inverted-cam vector
    int sx, sy;           // Object size
    RECT rect;            // Object rectangle

    // (1) Convert to camera frame
    vec_sub(&vr, &spr->pos, &cam->pos);
    vc.x=  vec_dot(&vr, &cam->u);
    vc.y= -vec_dot(&vr, &cam->v);
    vc.z= -vec_dot(&vr, &cam->w);
    spr->pos2= vc;

    OBJ_ATTR *obj= &spr->obj;
    sx= obj_get_width(obj);
    sy= obj_get_height(obj);

    // --- Check with viewbox ---
    do
    {
        // (2a) check distance
        if(M7_NEAR*256 > vc.z || vc.z > M7_FAR*256)
            break;

        // (2b) check horizontal
        rect.l= vc.x - spr->anchor.x*(256>>M7O_NORM);
        rect.r= rect.l + sx*(256>>M7O_NORM);
        if(M7_LEFT*vc.z > rect.r*M7_D || rect.l*M7_D > M7_RIGHT*vc.z)
            break;

        // (2c) check vertical
        rect.t= vc.y - spr->anchor.y*(256>>M7O_NORM);
        rect.b= rect.t + sy*(256>>M7O_NORM);
        if(-M7_TOP*vc.z > rect.b*M7_D || rect.t*M7_D > -M7_BOTTOM*vc.z)
            break;

        // (3) Set-up affine matrix
        OBJ_AFFINE *oa= &obj_aff_mem[spr->aff_id];
        oa->pa= oa->pd= vc.z>>(M7_D_SHIFT-M7O_NORM);    // normalized
lambda

        oa->pb= oa->pb= 0;

        FIXED scale= DivSafe(M7_D<<16, vc.z);    // (.16 / .8) = .8

        // (4) anchoring
```

```
            // Base anchoring equation:
            // x = q0 - s - A(p0 - s/2)
            // In this case A = 1/lam; and q0 = xc/lam
            // -> x = (xc - p0 + s/2)/lam - s + screen/2
            int xscr, yscr;
            xscr  = spr->anchor.x*256 - sx*128;            // .8
            xscr  = (vc.x - (xscr>>M70_NORM))*scale>>16;    // .0
            xscr += -sx - M7_LEFT;

            yscr  = spr->anchor.y*256 - sy*128;            // .8
            yscr  = (vc.y - (yscr>>M70_NORM))*scale>>16;    // .0
            yscr += -sy + M7_TOP;
            obj_unhide(obj, ATTR0_AFF_DBL);
            obj_set_pos(obj, xscr, yscr);

            return;
        }
    while(0);

        // (5) If we're here, we have an invisible sprite
        obj_hide(obj);
    }
```

Kart animation

The basic theory for animation around a sprite is simple, namely eq 21.17: the viewing angle ψ is the difference between the global sprite angle ϕ_o, and the camera angle ϕ_c and the angle to the sprite in camera-space α: $\psi = \phi_o - \phi_c - \alpha$. The angle translates to an animation frame to use and you're done.

In theory.

The practice has a number of snares, especially the way SMK does it. First, look at fig 21.16. These 12 frames are the ones that Super Mario Kart uses for Toad. The first complication is that this is only the right side of the rotation; the left side is done via mirroring. That's easy enough: just change the sign of p_a of the view-angle is negative.

The second problem is the number of tiles. 12 frames for half a circle means 24 for the full rotation (well 22 actually, as we don't need to duplicate the front and back frames). At 4x4=16 tiles each, this gives 384 tiles for Toad alone (and only the rotation

animation at that!) Multiply by 8 for the full set of characters and you're *way* out of VRAM. This means that you can't load all the frames into VRAM in one go and use an object's tile-index for animation: you have to dynamically load frames you need. This is why the sprite struct had a `tiles` member, pointing to the full sprite sheet in ROM.

The third complication is that the frames aren't uniformly divided over the circle. If you look closely, the first 8 frames are for angles 0° through 90°, the remaining four for 90°-180°. The reason behind this is that most of the time you'll see the karts from the back, so it pays to have more frames for those. Now, in the theory we could calculate the animation frame quite nicely, namely $N \cdot \psi / 2^{16}$. However, that relied on having N equal slices, which we don't have anymore. Or do we?

Well no, we don't have equal slices anymore. But we can *make* equal slices again, using a sort of mapping. Fig 21.17 shows could the principle works. In the figure there are 12 main partitions (inside circle), with 0, 1, 10 and 11 covering more angular space than 2-9. However, we can also divide the circle into 16 parts (outer circle), and use the same frame for multiple entries. For example, slice-0 of the main sequence is now covered by slice-0 and slice-1 of the new sequence. While it's possible to use if/else blocks to the mapping, it's easier on everyone to just use a LUT for it. This actually takes care of two other problems I hadn't mentioned before, namely that mirroring would require some sort of reversal of the normal sequence, and the fact that the slices actually have to be offset by half a slice so that you don't have a slice-switch when looking exactly at the front or back, for example. A LUT solves all those problems in one go.

Fig 21.16: Toad's frames from different angles.
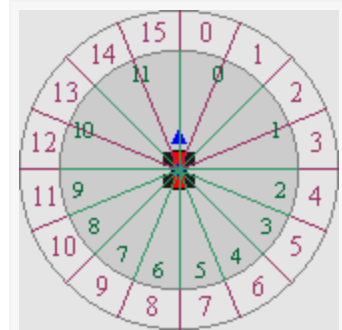
Fig 21.17: Using ψ for 16-element LUT entry, instead of 12 non-equal partitions.

```c
const u8 cKartFrames[32]=
{
     0,  1,  2, 3, 4, 5, 6, 7, 7, 8, 8, 9, 9, 10, 10, 11,
    11, 10, 10, 9, 9, 8, 8, 7, 7, 6, 5, 4, 3,  2,  1,  0,
};

//! Animate kart sprite
void kart_animate(M7_SPRITE *spr, const M7_CAM *cam)
{
    OBJ_ATTR *obj= &spr->obj;

    if(BF_GET2(obj->attr0,ATTR0_MODE) == ATTR0_HIDE)
        return;

    TILE *dst= &tile_mem[4][BF_GET(obj->attr2, ATTR2_ID)];
    s16 psi= spr->phi - cam->phi;

    // Extra arctan angle for correctness
    if(g_state & STATE_OBJ_VIEW_ATAN)
        psi -= ArcTan2(spr->pos2.z>>8, spr->pos2.x>>8);

    memcpy32(dst, &spr->tiles[cKartFrames[(psi>>11)&31]*16], 16*8);

    OBJ_AFFINE *oa= &obj_aff_mem[spr->aff_id];
    if(psi < 0)
        oa->pa= -oa->pa;
}
```

The snippet above shows the kart's angle-LUT and animation routine. The LUT has 32 entries, with the first and last 7 using single chunks and the rest being doubled. Also

note that the LUT is symmetric, which is required for the mirroring.

The routine itself isn't exactly pretty, but it gets the job done. It checks whether the sprite is visible first and bugs out if it's not: no point in doing work if we can't see its results. The sprite's in-camera angle, α, requires an arctan. I've added a switch in the menu so you can see the results with and without the α-correction, and I think you'll find that the difference is pretty small. Since I always use the same VRAM for each sprite, finding the destination of the tile-copy is easy; finding the source frame looks a little ugly, but it's just the ψ→slice conversion and the look-up, really.

Rounding up: the main loop and other fluff

The hard parts of mode 7 have more or less been covered now, with the possible exception of the main loop, which we'll get to in a moment. There is, of course, initialization of the registers, the sprites and mode 7 variables, loading of VRAM and input, but that's rather easy and tends to vary from game to game anyway. For those things, please see the actual code.

The main program flow

In the snippet below you can see the `main()` function and its major branches. `init_main()` sets up the main mode 7 variables, `m7_level` through `m7_init()`, initializes the VBlank and HBlank interrupts and various other things. The main loop is quite short. The function `input()` does both the movement of the camera and menu.

After that come the actual mode 7 functions. `m7_prep_horizon()` has to come first, but the order of the rest is pretty arbitrary. I would suggest calling `m7_prep_affines()` last, though: it's the most costly function here, but it'd be alright to let it run into VDraw time. Not that that happens here (I've clocked the main loop to end around scanline 170-210), but it'd be okay if it did.

```c
int main()
{
    init_main();

    while(1)
    {
        VBlankIntrWait();
        input();

        m7_prep_horizon(&m7_level);
        // Switch to backdrop display.
        if(m7_level.horizon > 0)
        {
            BF_SET(REG_DISPCNT, DCNT_MODE0, DCNT_MODE);
            REG_BG2CNT= m7_level.bgcnt_sky;
            REG_BLDALPHA= 16;
        }
        m7_update_sky(&m7_level);

        update_sprites();
        m7_prep_affines(&m7_level);
    }

    return 0;
}
```

Movement in 3D

This is the last thing I want to cover: how to move things in 3D. To be precise: how to do different methods of motion in 3D; which I'm sure people might want to know.

3D movement is actually much the same as 2D movement, except with an extra dimension. The reason why people sometimes find it difficult is that they think in terms of angles, when what they *should* be thinking in is vectors. Vector-based movement (or vector-based anything) usually makes things much easier than with angles and trigonometry. This is also why the theory of this chapter has been using vectors and matrices.

Here I'll look into three different modes of camera movements: one using the world coordinate system, one using the camera system, and one somewhere in between so

that it stays parallel to the ground. But first, let's take a look at what moving in a certain direction actually *means*.

Every object in 3D space has its own little coordinate space, the ***local frame***. This is defined as a set of 3 vectors, denoting the directions of the local *x*, *y* and *z* directions. In the case of the camera, I named these **u**, **v** and **w**, respectively. The local matrix is just another way of writing down this set of vectors. Movement is usually defined as steps along these vectors.

As an example of this, consider your head to be the camera and use arrows to indicate the local axes: **u** would stick out of your right ear, **v** out of the top of your head and **w** out the back. A step right would be along the **u** direction, and one forward along −**w**. A general movement could be written as *x* steps right, *y* steps up, and *z* steps back. *x*, *y* and *z* are used as *multipliers* for the direction vectors, and the final displacement in global space is **dx** = *x*·**u** + *y*·**v** + *z*·**w**.

And where matrices come in. Those three multipliers can be written a vector **r** = (*x*, *y*, *z*), which is the distance vector in *local* space. The three directions formed a matrix **C**, The definition of **dx** given above is nothing else than the long way of writing down **dx** = **C**·**r**. The matrix multiplication is just shorthand for "scale the vectors of **C** by the elements of **r** and add them all up". Note that this procedure would work for any object, in any orientation. All you need to do is find the correct local matrix.

In my case, I construct vector **r** in `input()`, based on various buttons. At this point it doesn't really mean anything yet. Each of the movement methods has its own set of directions and hence its own matrix that has to be applied to **r**; I have functions that can perform them and add the results to the camera position. All of these can be found in table 21.2 and the code below it.

The 'level' (that is, level to the ground) is probably the most common for camera systems for ground-based objects, though using the local system might make sense for flying objects. Experiment with them and see what you like.

Method	Function	Transformation

Global frame	`m7_translate_global()`	$dx = I \cdot r = r$
Local (camera) frame	`m7_translate_local()`	$dx = C(\theta, \phi) \cdot r$
Level: local but parallel to ground	`m7_translate_level()`	$dx = R_y(\phi) \cdot r$

Table 21.2: Movement methods and their associated transformations to world-space. New position of an object is given by $x_w \mathrel{+}= v_w$.

```c
//! Translate by \a dir in global frame
void m7_translate_global(M7_CAM *cam, const VECTOR *dir)
{
    vec_add_eq(&cam->pos, dir);
}

//! Translate by \a dir in local frame
void m7_translate_local(M7_CAM *cam, const VECTOR *dir)
{
    cam->pos.x += (cam->u.x * dir->x + cam->v.x * dir->y + cam->w.x *
dir->z) >> 8;
    cam->pos.y += ( 0               + cam->v.y * dir->y + cam->w.y *
dir->z) >> 8;
    cam->pos.z += (cam->u.z * dir->x + cam->v.z * dir->y + cam->w.z *
dir->z) >> 8;
}

//! Translate by \a dir using local frame for x/y, but global z
void m7_translate_level(M7_CAM *cam, const VECTOR *dir)
{
    cam->pos.x += (cam->u.x * dir->x - cam->u.z * dir->z)>>8;
    cam->pos.y += dir->y;
    cam->pos.z += (cam->u.z * dir->x + cam->u.x * dir->z)>>8;
}
```

If you're not really familiar with matrices they may seem bright and scary, but they can be a lifesaver once you get used to them a bit. There is a *reason* why large 3D systems use them non-stop; doing everything by raw trig is hard, very hard. Matrices allow you to work within whatever coordinate system is most natural to the task at hand, and then transform to whatever system you need in the end. If you have any work related to geometry, learning more about the basics of linear algebra (the rules for vector and matrix use) is well worth the effort.

> ### SIDE NOTE : CENTERING ON A SPRITE
>
> As an example of how easy matrices can make life, consider the issue of centering the camera on a given sprite and then rotating around it. You have the camera matrix **C**, the distance you want to view from, Z and presumably the sprite position, $\mathbf{x_w}$. What you need to do is: move the camera to the sprite's position, then take Z steps back. In other words $\mathbf{a_{cw}} = \mathbf{x_w} + \mathbf{C} \cdot (0, 0, Z)$, which boils down to $\mathbf{a_{cw}} = \mathbf{x_w} + Z\mathbf{w}$,
>
> Once you know the camera matrix, positioning it practically writes itself.

Concluding remarks

It's done, finally! This chapter's text explained the most important elements of a mode 7 game: calculation of the affine parameters, adding a horizon and backdrop, positioning, sorting *and* animating 3D sprites and as a bonus how to use create a distance fogging effect. In the preceding text, I've used stuff from just about every subject described in the rest of Tonc, and not just the easy parts. If you're here and understood all or most of the above, congratulations.

But still I've omitted a few things that would make it a little better. Getting rid of all those divisions in the λ calculations, for instance. Or getting around the 32 affine object limitation or placing shadows for the sprites on the floor. Nor have I shown how to correctly allow for loopings, instead of clamping the pitch at straight up or down. These things are relatively easy to grasp, conceptually, but implementing them would require a lot more code. If you understood this text, I'm sure you can figure it out on your own.

22. Tonc's Text Engine

Introduction

The other page on text described how you could get text on backgrounds and objects. It worked, but there were several limitations. For instance, it was limited to 8×8 fonts, didn't support all video modes and had no formatted text capabilities.

Tonc's Text Engine (TTE) remedies many of these shortcomings. In this chapter I'll describe the goals and basic design of the system and some of the implementation details. In particular, I'll describe how to build writers for use of the different kinds of surfaces. In some cases, I'll optimize the living hell out of them because it is possible for a glyph renderer to take multiple scanlines for a single character if you don't pay attention. And yes, this will be done in assembly.

I'll also show how you can add some basic scripting to change cursor positions, colors and even fonts dynamically. A few years ago, Wintermute changed the standard C library in devkitARM to make the stdio routines accessible for GBA and NDS. I'll also show how you can make use of this.

And, of course, there will be demos. Oh, will there be demos. There are about 10 of them in fact, so I'm going to do things a little bit differently than before: there will be one project containing a menu with all the examples. Not all examples will be shown here because that'd just be too much.

Lastly, it is expected that by now you have a decent knowledge of GBA programming, so I'm going to keep the amount of GBA-specific exposition to a minimum. When you see functions used that haven't been covered already, turn to GBATEK, the project's code or libtonc's code for details.

Basic design

TTE Goals

The following list has the things I most wanted in TTE:

- **A comprehensive and extensible set of glyph writers, usable for all occasions**. Well *almost* all occasions. The old system worked for regular backgrounds, bitmap modes and objects, I'm now extending that set to affine backgrounds and tile-rendering. If what you need isn't present in the standard set, you can easily create your own writer and use that one instead. The writer will accept UTF-8 strings, meaning you're not limited to 256 characters.
- **Fonts: arbitrary widths and heights and variable width characters**. Instead of being limited to 8x8@1 glyphs; the standard writers in TTE are able to use fonts of any width and height (within reason: no screen-filling glyphs please) and variable width fonts (again, within reason: VWF for tilemaps makes little sense). In principle, there are possibilities to use arbitrary bitdepths as well, but the standard renderers are limited to 1bpp.
- **A simple writer-interface independent of surface details**. For the old system I had `m3_puts()`, `se_puts()`, `obj_puts()` and such. This worked, but it meant you had to use something different for the different modes. In TTE, there are

different initializers for the different modes to set up the system, and a single string writer `tte_write()` that just works.

- **Scripting for text parameters**. By that I mean that you can control parameters like position and output color by the strings themselves. The functionality for this is pretty basic, but it works well enough. Note: this is *not* a full dialog system! That said, it should be possible to build one around it.
- `printf()` **support**. For rather obvious reasons.

Structures and main components

All the relevant information for TTE is kept gathered in three structs: a ***text context***, `TTC` ; a ***font description***, `TFont` ; and a ***graphic surface description***, `TSurface` .

The `TTC` struct contains the main parameters for the engine: information about the surface were rendering to, cursor positions, font information, color attributes and a few other things. It also contains two callbacks for drawing and erasing glyphs.

The `TFont` struct has a pointer to the glyph data, glyph/cell dimensions and a few other things. There are also pointers to width and height tables to allow variable width and height fonts. I've hacked a `TFont` creator into usenti a while back so that I could easily create these things from standard fonts, but you can also make your own from scratch.

The `TSurface` struct actually has nothing to do with text. Instead, it's a struct describing the kind of surface we're rendering on. This can be bitmaps, tiles, tilemaps or whatever. Tonclib has basic pixel, line and rectangle routines for dealing with these surfaces, so I might as well use them.

```
//# From tonc_tte.h : main TTE types.

typedef struct TFont
{
    const void  *data;      //!< Character data.
    const u8    *widths;    //!< Width table for variable width font.
    const u8    *heights;   //!< Height table for variable height font
(mostly unused).
    u16 charOffset;         //!< Character offset
    u16 charCount;          //!< Number of characters in font.
    u8  charW;              //!< Character width (fwf).
    u8  charH;              //!< Character height.(fhf).
    u8  cellW;              //!< Glyph cell width.
    u8  cellH;              //!< Glyph cell height.
    u16 cellSize;           //!< Cell-size (bytes).
    u8  bpp;                //!< Font bitdepth;
    u8  extra;              //!< Padding. Free to use.
} TFont;

//! TTE context struct.
typedef struct TTC
{
    // Members for renderers
    TSurface dst;           //!< Destination surface.
    s16 cursorX;            //!< Cursor X-coord.
    s16 cursorY;            //!< Cursor Y-coord.
    TFont *font;            //!< Current font.
    u8  *charLut;           //!< Character mapping lut, if any.
    u16 cattr[4];           //!< ink, shadow, paper and special color
attributes.
    // Higher-up members
    u16 reserved;
    u16 ctrl;               //!< BG control flags.
    u16 marginLeft;
    u16 marginTop;
    u16 marginRight;
    u16 marginBottom;
    s16 savedX;
    s16 savedY;
    // Callbacks and table pointers
    fnDrawg  drawgProc;         //!< Glyph render procedure.
    fnErase  eraseProc;         //!< Text eraser procedure.
    const TFont **fontTable;    //!< Pointer to font table for
    const char  **stringTable;  //!< Pointer to string table for
} TTC;
```

```
//# Supporting types

//! Glyph render function format.
typedef void (*fnDrawg)(int);

//! Erase rectangle function format.
typedef void (*fnErase)(int left, int top, int right, int bottom);

typedef struct TSurface
{
    u8  *data;          //!< Surface data pointer.
    u32 pitch;          //!< Scanline pitch in bytes.
    u16 width;          //!< Image width in pixels.
    u16 height;         //!< Image width in pixels.
    u8  bpp;            //!< Bits per pixel.
    u8  type;           //!< Surface type.
    u16 palSize;        //!< Number of colors.
    u16 *palData;       //!< Pointer to palette.
} TSurface;
```

TFont details

Fig 22.1 shows a character sheet that `TFont` can use. The sheet is a matrix of *cell*s and each cell contains a character. The `cellW/H` members are the dimensions of these cells; `cellSize` is the number of bytes per cell.

Each cell has one glyph, but the actual glyphs can be smaller than the cells (white vs magenta parts). This does waste a bit of memory, but it also has several benefits. One of the benefits is that you can use `cellSize` to quickly find the address of any given glyph. Second, because I want by fonts to be usable for both bitmaps *and* tiles, my glyph boxes would have to be multiples of 8 anyway. Additionally, this particular font will be 1bpp, meaning that even with the wasted parts I'll still have a very low memory footprint (3.5kB).

Fig 22.1: Verdana 9 character sheet

For fixed-width or fixed-height fonts, members `charW` and `charH` denote the actual character width and height. For fonts of variable widths, the `widths` member points to the a byte-array containing the widths of the glyphs and something similar is true for the `heights`. `charOffset` is the (ASCII) character the data starts at. Font sheets often start at a space (' '), so this tends to be 32. `charCount` is the number of characters and can be used if you need to copy the whole sheet to VRAM (like in the case of tile-mapping).

Please note that how the data in a `TFont` is used depends almost entirely on the glyph renderer. Most renderers that come with libtonc expect this format:

- Bitpacked to **1 bpp**, for size reasons. And for rendering speed too, actually, since memory loads are expensive.
- Tiled-by-glyph. The data for each glyph is contingent with `cellSize` bytes between each glyph. This is similar to how 1D object work with one important difference:
- the tiles in each glyph are **column-major** (tile 1 is under tile 0). This in contrast to objects, which tend to be row-major (tile 1 is to the right of tile 0). I will refer to this format as **tile-strips**. The reason behind this choice will be given later.

There are exceptions to this, but most renderers presented here will use this format. If you want to make your own renderers, you're free to use any format for the data you think is appropriate.

TTC details

The text context, `TTC`, contains the most important data of the system. Starting at the top: the surface, `dst`. This defines the surface we're rendering to. The most relevant items there are its memory address, **pitch**: the number of bytes per scanline. The pitch is a *very* important parameter for rendering, more important than the width and height in fact. The surface also has palette members, which can be used to access its colors. Much like the `TFont` members, how this data is used largely depends on the renderer.

The members `cursorX/Y` are for the current cursor position. The `margin` rectangle indicates which part of the screen should be used for text. If the cursor exceeds the right margin, it will be moved to the left margin and one line down. The margins are also used for screen-clears and returning to the top of the page.

The `cattr` table is something special. Its entries are **color attribute**s. Parameters for ink (foreground color), shadow, paper (background color) are put here, along with a 'special' field which is very much context-specific. Note that these color attributes do not necessarily represent colors. In modes 3 and 5 they're colors, but for mode 4 and tile writers they're color indices. There's probably a nicer name for this than 'color attribute', but sodomy non sapiens.

Rendering glyphs and erasing (parts of) the screen is done through the callbacks `drawgProc` and `eraseProc`. The idea is that you initialize the system with the routines appropriate for your text format and TTE uses them to do the actual writing. I should point out that using callbacks for rendering a single glyph can have a significant overhead, especially for the simpler kinds of text like tilemaps.

Main TTE variables and functions.

The state of the TTE system is kept in a `TTC` variable accessible through `tte_get_context()`. All changes to the system go through there. In *some* cases, it is useful to have two sets of state and switch between them when appropriate (like when you have two screens. Y hello thar, NDS). For that you can use `tte_set_context)` to redirect the pointer.

```
TTC __tte_main_context;
TTC *gp_tte_context= &__tte_main_context;

//! Get the master text-system.
INLINE TTC *tte_get_context()
{    return gp_tte_context;                                    }


//! Set the master context pointer.
void tte_set_context(TTC *tc)
{
    gp_tte_context= tc ? tc : &__tte_main_context;
}
```

To print characters, you can use `tte_putc()` and `tte_write()`.

```c
//! Get the glyph index of character \a ch.
INLINE uint tte_get_glyph_id(int ch)
{
    TTC *tc= tte_get_context();
    ch -= tc->font->charOffset;
    return tc->charLut ? tc->charLut[ch] : ch;
}

//! Get the width of glyph \a id.
INLINE uint tte_get_glyph_width(uint gid)
{
    TFont *font= tte_get_context()->font;
    return font->widths ? font->widths[gid] : font->charW;
}

//! Render a character.
int tte_putc(int ch)
{
    TTC *tc= tte_get_context();
    TFont *font= tc->font;

    // (4) translate from character to glyph index
    uint gid= tte_get_glyph_id(ch);

    // (5) get width for cursor update
    int charW= tte_get_glyph_width(gid);

    if(tc->cursorX+charW > tc->marginRight)
        [[ simulate newline ]]

    // (6) Draw and update position
    tc->drawgProc(gid);
    tc->cursorX += charW;

    return charW;
}
```

```c
//! Render a string.
/*! \param text String to parse and write.
    \return     Number of parsed characters.
*/
int tte_write(const char *text)
{
    int ch;
    uint gid, charW;
    const char *str= text;
    TTC *tc= tte_get_context();

    while( (ch= *str++) != '\0' )
    {
        // (1) Act according to character type
        switch(ch)
        {
        case '\n':  [[ update cursorX/Y for newline ]];     break;
        case '\t':  [[ update cursorX for tab ]];           break;
        default:
            // (2) more special thingies
            if(ch=='#' && str[0]=='{')           // (2a) Command sequence
            {
                str= tte_cmd_default(str+1);
                break;
            }
            else if(ch=='\\' && str[0]=='#')     // (2b) Escaped command
                ch= *str++;
            else if(ch>=0x80)                    // (2c) UTF8 character
                ch= utf8_decode_char(str-1, &str);

            // (3) draw character
            tte_putc(ch);
        }
    }

    return str - text;
}
```

I've omitted the code for a few things here, the idea should be clear. First, read a character. Then, check whether it's a special character (new line, tab, formatting command) and if so, act accordingly. Because `tte_write()` supports UTF-8, we also check for that and decode the string for a full UFT-8 character. After that's all done, we pass the character on to `tte_putc()`, which translates it to a glyph index, draws the glyph and advances the cursor.

Note: the way described here is *a* method of doing things; it's not *the* method, because that doesn't actually exist. Several steps done here may be overkill for the kind of text you had in mind. For example, getting from the character to the glyph index is done by the font's character offset and a potential character look-up table, neither of which is strictly necessary. Likewise, wrapping at the edges may already be done in the string itself with newline characters. On the other hand, you might like more complex wrapping, text alignment, scrolling, and who knows what else. If you want these things, creating your own routine shouldn't be too difficult.

On nomenclature

Some terms I use in TTE have a very specific meaning. Because the differences between terms can be subtle, it is important to define the term explicitly. Additionally, TTE uses several acronyms and abbreviations that need to be clarified.

- **char(acter) vs glyph index**. 'Character' refers to the ASCII character; the 'glyph index' is the corresponding index in the font. For example, 'A' is character 65, but if the font starts at a space (' ', ASCII 32) the 'A' is glyph index 65−32=33. As a rule, variables named `ch` are characters and `gid` means glyph index. The input of the renderers is the glyph index, and not the character.

- **Surface**. Surface is the term I'm using to describe whatever I'm manipulating to show text. This is usually VRAM, but can be other things as well, like OBJ_ATTRs for object text.

- **Pitch**. The pitch is actually a common term in graphics, but since graphics terms may not be so common, it's worth repeating. Technically, the *pitch* is the number of scanlines between rows. I'm extending it a little to mean the *characteristic major distance* for matrices. Matrices are 2D entities, and they'll have adjacent elements in one direction and a larger distance for the other. These usually are *x* and *y*, respectively, but not always. The *minor* distance will be referred to as *stride*.

- **Color vs color attribute**. The 'color' is the real 5.5.5 BGR color; the 'color attribute' is whatever the renderer will use on the surface. This can be a color, but it can also be a palette index or something else entirely. The interpretation is up to the renderer.

- **Render/Text family**. This is a conceptual group-name for a specific kind of text. Table 22.1 gives an overview of the families available. This largely corresponds to the `TSurface` types.

- **Renderer types**. Within each family there can be different renderers for different kinds of fonts and effects. For example, when rendering to an 8bpp bitmap (the bmp8 family), you can have different renderers for different font bitdepths (1bpp or 8bpp, for example) or glyph layouts (bitmapped or tiled). They can render some pixels transparently, or apply anti-aliasing. Or any combination of those. The point is there are a *lot* of options here.

Because I really don't like names that span a whole line, I will use abbreviations in the renderer's name to indicate what it does; see table 22.2 for what they mean. For the most part, the renderers will be for fonts with arbitrary width and height, with a 1bpp tile-stripped glyphs, with they will draw them transparently and re-coloring of pixels. This is indicated by `*_b1cts`.

Family	prefix	Initializer
Regular tilemap (mode 0/1)	se	void tte_init_se(int bgnr, u16 bgcnt, SCR_ENTRY se0, u32 colors, u32 bupofs, const TFont *font, fnDrawg proc);
Affine tilemap (mode 1/2)	ase	void tte_init_ase(int bgnr, u16 bgcnt, u8 ase0, u32 colors, u32 bupofs, const TFont *font, fnDrawg proc);
4bpp Tiles (modes 0/1/obj)	chr4*(c/r)*	void tte_init_chr4*(c/r)*(int bgnr, u16 bgcnt, u32 cattrs, u32 colors, const TFont *font, fnDrawg proc);
8bpp bitmap	bmp8	void tte_init_bmp(int vmode, const TFont

(mode 4)		*font, fnDrawg proc);
16bpp bitmap (mode3/5)	bmp16	void tte_init_bmp(int vmode, const TFont *font, fnDrawg proc);
objects	obj	void tte_init_obj(OBJ_ATTR *dst, u32 attr0, u32 attr1, u32 attr2, u32 colors, u32 bupofs, const TFont *font, fnDrawg proc);

Table 22.1: TTE render family indicators and initializers. 4bpp Tiles can be row or column major (`crh4r` or `chr4c`).

Code	Description
b*x*	**B**itdepth of source font. (`b1` = 1 bpp)
w*x*	Specific **w**idth (`w8` = width 8)
h*x*	Specific **h**eight (`h8` = height 8)
c	Re-**c**oloring. Color attributes are applied to the pixels in some way.
t/o	**T**ransparent or **o**paque paper pixels.
s	Glyphs are in tile-**s**trip format.

Table 22.2: Render type summary.

Lastly, a note on some of the abbreviations I use in the rendering code. A number of terms come up again and again, and I've taken to use a shorthand notation for these items. The basic format is `fooX` where `foo` is the relevant bitmap/surface and `X` is a one-letter code for things like width, height, data and others. Yes, the use of single-letter names is frowned upon and I don't advocate their use in general, but I've found that in this particular case, if used judiciously, they have helped me read my own code.

Term	Meaning
`fooW`	Width of `foo`
`fooH`	Height of `foo`
`fooB`	Bitdepth of `foo`
`fooP`	Pitch of `foo`

`fooD`	Primary data-pointer for `foo`
`fooL`	Secondary data-pointer for `foo`
`fooS`	Size of `foo`
`fooN`	Number/count of `foo`

Table 22.3: Abbreviations used in rendering code.

Tilemapped text

Regular tilemap text

Tilemapped text is the easiest to implement, because you don't really have to render anything at all. You simply load up all the font's tiles into a charblock and place screen-entries for the actual text.

The initializer for regular tilemaps is `tte_init_se()`. It's identical to `txt_init_se()` except for the two extra parameters at the end: `font` and `proc`. These represent the font to use and the renderer that does the surface manipulation. Every initializer in TTE has those two parameters. It's safe to pass NULL to them if you're not sure what to use; in that case, the default option for that family will be used.

If `font` is NULL, you'll get the default font. This is either `system8Font` for fixed-width occasions or `verdana9Font` (fig 22.1) when variable width is suitable. These can also be referenced via `fwf_default` and `vwf_default`, respectively.

Each family also has a default renderer, #defined as `foo` `_drawg_default`, where *foo* is the family prefix. The default renderers are the general routines, suitable for all character widths and heights (fixed or variable fonts). Of course, this does mean that they will be slower than routines written to work with a specific glyph size. This is particularly true for tilemapped text, and for that reasons specific `_w8h8` and `_w8h16` versions are available there as well.

The initializers tend to be long and boring, so I won't waste too much space on them here. Basically, they clear out the text context, assign some sensible values to the margins and surface variables, set up the font, the renderer and the eraser. They also fill some of the palette and color attributes.

The code I'll show in this chapter will mostly be about the renderers themselves. Below you can see the code for the default screen-entry writer, `se_drawg_s()`, and the one specific for 8×8 fonts, `se_drawg_w8h8`

```
//! Character-plot for reg BGs, any sized, vertically tiled font.
void se_drawg_s(uint gid)
{
    int ix, iy;

    // (1) Get main variables.
    TTC *tc= tte_get_context();
    TFont *font= tc->font;
    uint charW= (font->cellW+7)/8, charH= (font->cellH+7)/8;

    uint x0= tc->cursorX, y0= tc->cursorY;
    uint dstP= tc->dst.pitch/2;
    u16 *dstD= (u16*)(tc->data + (y0*dstP+x0)*2);

    // (2) Get the base tile index.
    u32 se= tc->cattr[TTE_SPECIAL] + gid*charW*charH;

    // (3) Loop over all tiles to draw glyph.
    for(ix=0; ix<charW; ix++)
    {
        for(iy=0; iy<charH; iy++)
            dstD[iy*dstP]= se++;
        dstD++;
    }
}

//! Character-plot for reg BGs using an 8x8 font.
void se_drawg_w8h8(uint gid)
{
    TTC *tc= tte_get_context();

    uint x0= tc->cursorX, y0= tc->cursorY;
    uint dstP= tc->dst.pitch/2;
    u16 *dstD= (u16*)(tc->data + (y0*dstP+x0)*2);

    dstD[0]= tc->cattr[TTE_SPECIAL] + gid;
}
```

Let's start with the simpler one: `se_drawg_w8h8()`. An 8×8 glyph on a GBA tilemap simply means write a single screen-entry to the right place. The right place here is derived from the cursor position and the surface data (`tc->dst`). The 'special' color attribute is used as a modifier to the glyph index for things like palette swapping.

Note that the routine just handles plotting the glyph. Transforming from ASCII to glyph index and repositioning the cursor is all done elsewhere.

The more generalized routine, `se_drawg_s()` is a little more complex. It still starts by getting a pointer to the glyph's destination, `dstD` , and pitch (the distance to the next line), `dstP` . **All** renderers start with something like this. All renderers also retrieve the character's width and height – unless the sizes are specified in advance. The names I use for rendering are always the same, so you should be able to tell what means what even when the formulas for initializing them can be a tad icky.

Anyway, after getting the pointer and pitch, the tile-index for the top-left of the glyph is calculated and put this into `se` . After that, we loop over the different tiles of the glyph in both directions. Note that the order of the loop is column-major, not row-major, because that's the way the default fonts were ordered.

As it happens, column-major rendering tends to be more efficient for text, because glyphs are usually higher than they are wide. Also, for tilemap text `charW` and `charH` tend to be small – often 1 or 2. This means that it is extremely inefficient to use loops; we'll see how inefficient in the "Profiling the renderers" subsection.. Unrolling them, like `se_drawg_w8h8()` and `se_drawg_w8h16()` do, gives a much better performance.

Regular tilemap example

```c
void test_tte_se4()
{
    irq_init(NULL);
    irq_add(II_VBLANK, NULL);
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    // --- (1) Base TTE init for tilemaps ---
    tte_init_se(
        0,                      // Background number (BG 0)
        BG_CBB(0)|BG_SBB(31),   // BG control (for REG_BGxCNT)
        0,                      // Tile offset (special cattr)
        CLR_YELLOW,             // Ink color
        14,                     // BitUnpack offset (on-pixel = 15)
        NULL,                   // Default font (sys8)
        NULL);                  // Default renderer (se_drawg_s)

    // --- (2) Init some colors ---
    pal_bg_bank[1][15]= CLR_RED;
    pal_bg_bank[2][15]= CLR_GREEN;
    pal_bg_bank[3][15]= CLR_BLUE;
    pal_bg_bank[4][15]= CLR_WHITE;
    pal_bg_bank[5][15]= CLR_MAG;

    pal_bg_bank[4][14]= CLR_GRAY;

    // --- (3) Print some text ---

    // "Hello world in different colors"
    tte_write("\n Hello world! in yellow\n");
    tte_write(" #{cx:0x1000}Hello world! in red\n");
    tte_write(" #{cx:0x2000}Hello world! in green\n");

    // Color use explained
    tte_set_pos(8, 64);
    tte_write("#{cx:0x0000}C#{cx:0x1000}o#{cx:0x2000}l");
    tte_write("#{cx:0x3000}o#{cx:0x4000}r#{cx:0x5000}s");
    tte_write("#{cx:0} provided by \\#{cx:#}.");


    // --- (4) Init for 8x16 font and print something ---
    GRIT_CPY(&tile_mem[0][256], cyber16Glyphs); // Load tiles
    tte_set_font(&cyber16Font);                 // Attach font
    tte_set_special(0x4100);                    // Set special to tile
256, pal 4
    tte_set_drawg(se_drawg_w8h16);              // Attach renderer

    tte_write("#{P:8,80}Also available in 8x16");
```

```
    key_wait_till_hit(KEY_ANY);
}
```

The code above demonstrates a few of the things you can do with TTE for tilemaps. The call to `tte_init_se()` initializes the system to display text on BG 0, using charblock 0 and screenblock 31 and to use the default font and renderer. Parameter five is the bit-unpack offset; by setting it to 14, all the 1-valued pixels in the font move to 14+1=15, the last index in a palette bank. I'm also setting a few other colors so that the palette will look like fig 22.3b.

In step 3, I print some text with `tte_write()`. The different colors are done by using `#{cx: num }` in the string, which sets the special color-attribute to *num*. More on these kinds of commands in the "Scripting, console IO and other niceties" section.. Since the `se` -renderers add this value to the glyph index for the final output, it can be used for palette swapping.

Step 4 demonstrates how to load up and use a second font. The `cyber16Font` is a rendition of the 8×16 font used in ye olde SNES game, Cybernator (see fig 22.2). This font was exported as 4bpp data so I can just copy it into VRAM directly, but I do need to use an offset because I want to keep the old font as well. The charblock now has two sets of glyphs (see fig 22.3c).
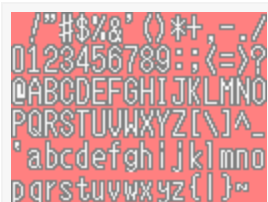

Fig 22.2: Cybernator font: 8×16.

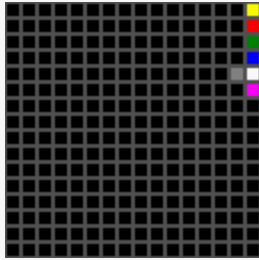**Fig 22.3a**: `test_tte_se4` output.

Fig 22.3b: Palette.   **Fig 22.3c**: Tileset.

In principle, all I need to do to use a different font is to select it with `tte_set_font()`, but since the tiles are at an offset, I also need to adjust the special color attribute. The value of 0x4100 is used here to account for the offset (0x0100) as well as the palette-bank (0x4000). I'm also selecting a different renderer for the occasion, although that's mostly for show here because the default renderer can handle 8×16 fonts just as well. After that, I just call `tte_write()` again to print a new string in using the new font.

Affine tilemap text

Text for affine tilemaps works almost the same as for regular tilemaps; you just have to remember the differences between the two kinds of backgrounds, like map size and available bitdepth. The functions' prototypes are the same, except that `se` is replaced by `ase`.

Internally, the only real difference is in what the renderers are to output, namely bytes instead of halfwords. And here we run into that quaint little fact of VRAM again: you can't write single bytes to VRAM. This means that the renderers will be a little more complicated. But only a little: simply call a byte-plotting routine for the screen-entry placement. Because affine maps are essentially 8bpp bitmap surfaces, I can use the standard plotter for 8bpp bitmap surfaces: `_sbmp8_plot()`. Aside from this one difference, the `ase_` renderers are identical to the `se_` counterparts.

```
//! Character-plot for affine BGs using an 8x8 font.
void ase_drawg_w8h8(uint gid)
{
    TTC *tc= tte_get_context();
    u8 se= tc->cattr[TTE_SPECIAL] + gid;

    _sbmp8_plot(&tc->dst, tc->cursorX/8, tc->cursorY/8, se);
}

//! Character-plot for affine BGs, any sized, vertically oriented font.
void ase_drawg_s(int gid)
{
    TTC *tc= tte_get_context();
    TFont *font= tc->font;
    uint charW= (font->cellW+7)/8, charH= (font->cellH+7)/8;
    uint x0= tc->cursorX/8, y0= tc->cursorY/8;

    u8 se= tc->cattr[TTE_SPECIAL] + gid*charW*charH;

    int ix, iy;
    for(ix=0; ix<charW; ix++)
        for(iy=0; iy<charH; iy++, se++)
            _sbmp8_plot(&tc->dst, ix+x0, iy+y0, se);
}
```

The demo for affine map text is `text_tte_ase()`. The idea is simple here: set up the text for a 256×256 pixel map, write some text onto it and rotate the background to illustrate that it is indeed an affine background. The center of rotation is the "o" at the center of the screen. To place it there, I've used the `#{P:x,y}` code; this sets the cursor to the absolute position given by (x, y). The other string is also positioned on the map in this manner.

```c
void test_tte_ase()
{
    irq_init(NULL);
    irq_add(II_VBLANK, NULL);
    REG_DISPCNT= DCNT_MODE1 | DCNT_BG2;

    // Init affine text for 32x32t bg
    tte_init_ase(
        2,                          // BG number
        BG_CBB(0) | BG_SBB(28) | BG_AFF_32x32,  // BG control
        0,                          // Tile offset (special cattr)
        CLR_YELLOW,                 // Ink color
        0xFE,                       // BUP offset (on-pixel = 255)
        NULL,                       // Default font (sys8)
        NULL);                      // Default renderer (ase_drawg_s)

    // Write something
    tte_write("#{P:120,80}o");
    tte_write("#{P:72,104}Round, round, #{P:80,112}round we go");

    // Rotate it
    AFF_SRC_EX asx= { 124<<8, 84<<8, 120, 80, 0x100, 0x100, 0 };
    bg_rotscale_ex(&REG_BG_AFFINE[2], &asx);

    while(1)
    {
        VBlankIntrWait();
        key_poll();

        asx.alpha += 0x111;
        bg_rotscale_ex(&REG_BG_AFFINE[2], &asx);

        if(key_hit(KEY_START))
            break;
    }
}
```


Fig 22.4: `test_tte_ase` .

Bitmapped text

Bitmap text rendering is a little different from map text and can range in difficulty from easy to insane, depending on your wishes. At its core, though, it's always the same process: loop over all pixels and draw them on the destination surface. For example, a generic glyph renderer that draws pixels transparently could look something like this.

```
// Pseudo code for a general glyph printer.
void foo_drawg(uint gid)
{
    TTC *tc= tte_get_context();
    TFont *font= tc->font;

    // Drawing with color keying.
    // Loop over all pixels. If the glyph's pixel is not zero, draw it.
    // Other wise, nevermind.
    for(iy=0; iy<tte_get_glyph_height(gid); iy++)
    {
        for(ix=0; ix<tte_get_glyph_width(gid); ix++)
        {
            u16 color= font_get_pixel(font, gid, ix, iy);
            if(color != 0)
                foo_plot(&tc->dst, tc->cursorX+ix, tc->cursorY+iy,
color);
        }
    }
}
```

Here, `foo` can mean any rendering family. `foo_plot()` is a general pixel plotter and `font_get_pixel()` a pixel retriever. The implementations of those functions depend on the specifics of the font and surface, but the glyph renderer doesn't need to know about that.

Basic bmp16 to bmp16 glyph printer

This next function is an example of a 16bpp font to 16bpp bitmap printer.

```
//! Glyph renderer from bmp16 glyph to bmp16 destination.
void bmp16_drawg(uint gid)
{
    // (1a) Basic variables
    TTC *tc= tte_get_context();
    TFont *font= tc->font;

    u16 *srcD= (u16*)(font->data+gid*font->cellSize), *srcL= srcD;
    uint charW= tte_get_glyph_width(gid)
    uint charH= tte_get_glyph_height(gid);

    uint x0= tc->cursorX, y0= tc->cursorY;
    uint srcP= font->cellW, dstP= tc->dst.pitch/2;
    u16 *dstD= (u16*)(tc->dst.data + (y0*dstP + x0)*2);

    // (2) The actual rendering
    uint ix, iy;
    for(iy=0; iy<charH; iy++)
    {
        for(ix=0; ix<charW; ix++)
            if(srcD[ix] != 0)
                dstD[ix]= srcD[ix];

        srcD += srcP;
        dstD += dstP;
    }
}
```

Blocks 1a and 1b set up the main variables to use in the loop. The most important are the source and destination pointers, `srcD` and `dstD`, and their pitches, `srcP` and `dstP`. Notice that the source pitch, `srcP`, is the not the character width, but the cell width, because the fonts are organized on a cell-grid. The code at point 2 selectively copies pixels from the font to the surface.

Intermezzo : considerations for performance

You may wonder why `bmp16_drawg()` doesn't follow the pattern in the earlier `foo_drawg()` more closely. The answer is, of course, performance. And before anyone quotes Knuth on me: not every effort to make your code fast is premature optimization. When you can improve the speed of the code without spending too much effort of a loss of readability, there's not much reason not to.

In this case, the optimizations I've applied here fall into two categories: local variables and pointer arithmetic. These techniques – that every C programmer *should* know – managed to boost the speed by a factor of 5.

Let's start with pointers. I'm using pointer to my advantage in two places here. First, instead of using the font-data pointer and destination pointer directly, I create pointers `srcD` and `dstD` and direct them the top-lefts of the glyph and where it will be rendered to. Short-circuiting the accesses like this means that I don't have to apply extra offsets to get to where I want to go in the loop. This will be both faster and in fact more readable as well, because the loops won't contain any non-essential expressions.

```
//# Example of a more standard bitmap copier.
for(iy=0; iy < charH; iy++)
    for(ix=0; ix < charW; ix++)
        dstD[iy*dstP + ix]= srcD[iy*srcP+ix];
```

A second point here is using incremental offsets instead of the `y*pitch+x` form (see above). I suppose this is mostly a matter of preference, but avoiding the wholly unnecessary multiplications does matter.

The second optimization is local variables. By this I mean loading variables that reside in memory (globals and struct members) or oft-used function results in local temporaries. This may seem like a silly thing to point out, but the amount of time you can save with this is actually quite high.

Consider the use of `tte_get_glyph_width()` here. I *know* the width of a glyph won't change during the loop, so calling a function to get the width in the loop-condition itself is just stupid. Another example of this would be calling `strlen()` when looping over all characters in a string. For those who do this: NO! Bad programmer, bad! Save the value in a local variable and use that instead.

The other point is to pre-load things like globals and struct/class members if you use them more than once. Consider the following code. It's the same as the one given

before, only now I have not loaded character height and the pitches into local temporaries.

```
//# Another bitmap-copy example. DO NOT USE THIS !!!
for(iy=0; iy < font->charH; iy++)
    for(ix=0; ix < charW; ix++)
        dstD[iy*tc->dst.pitch/2 + ix]= srcD[iy*font->cellW + ix];
```

The result: the speed of the function was **halved**! I expected it to be slower, but that this innocuous-looking modification would actually cost me a factor two was quite a surprise to me.

So yes, spam your code width locals for loop-invariant, memory-based quantities. This avoids them being loaded from memory every time. As a bonus, the loops themselves win contain less text and be more generalized, making it more reusable.

Both the pointer work and pre-loading variables are actually the job of the compiler's optimizer, but the current version of GCC doesn't do these very well or at all. Also, sometimes it *can't* do this optimization. When functions are called between memory dereferences, the compiler has to reload the data because those functions may have changed their contents. Obviously, this wouldn't happen for locals.

> **USE LOCAL VARIABLES FOR STRUCT MEMBERS AND GLOBALS**
>
> Struct members and global variables live in memory, not CPU registers. Before the CPU can use their data, they have to be loaded from memory into registers first, and this often happen more times then necessary. Since memory access (especially ROM access) is slower than register access, this can really bog down an algorithm if the thing is used more than once. You can avoid this needless work by creating local variables for them.
>
> Aside from the speed-boost, local variables can use shorter names, resulting in shorter and more readable code in the parts that actually do the work. It's win-freakin'-win, baby.

Glyph and surface formats

The renderer described above assumes that the glyphs are in formatted as 16-bpp bitmaps. However, TTE's default fonts are in a 1-bpp tilestrip format, so I'll have to use something else. Before I go into the details of that function, I'd like to discuss the different glyph formats and why I'm using tile-strips instead of just plain bitmaps.

When I say glyph formats, what I really mean is the order in which pixels are accessed. Three key variations exist.

- **Linear** or **bitmap** layout. This is a simple, row-major matrix. This gives you two loops; one for y and one for x.
- **Tiled**. In particular: 8×8 tiled. This is the standard GBA tile format where each group of 8×8 pixels form a row-major matrix, and then the tiles themselves are part of a larger row-major matrix again. Going through this required **four** loops: two for each matrix.
- **Tile-strips**. This also uses 8×8 tiles, but this time the tiles ordered in a column-major order. In other words, tile 1 comes below tile 0 instead of to the right. This has the rather nice property that the rows in successive tiles are consecutive. It eliminates the break in the y direction, resulting in only 3 loops and has simpler code to boot.

Fig 22.5 shows these three layouts, including the loop structure and the order in which the pixels are traversed for 1bpp fonts. The case is a little bit different because of the bit-packing: 1 bpp means 8 pixels per byte. As a result, the bitmap-x loops have to be broken up into groups of 8, so that the bitmap format now uses three nests of loops instead of just two. There is no difference for the tiled formats, as those are grouped by 8 pixels anyway. Not only that, if you were to calculate the total loop-overhead for commonly used glyph sizes it turns out that this arrangement actually works particularly well for tile-strips. This is left as an exercise for the reader (hint: count the number of comparisons).
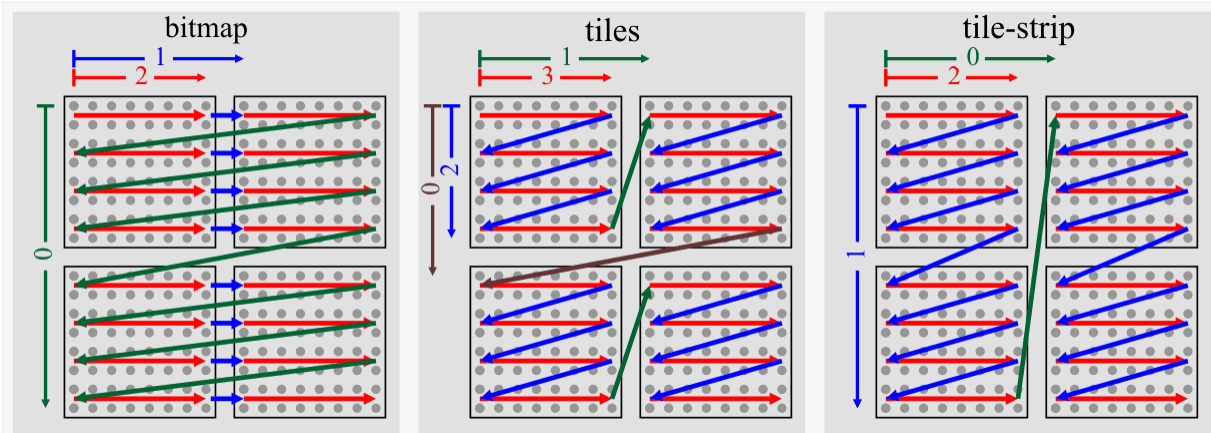
Fig 22.5: Pixel traversal in glyphs for 1-bpp bitmap, tile and tile-strip formats. The numbers indicate the loops and their nestings.

bmp16_drawg_b1cts : 1- to 16-bpp with transparency and coloring

The next routine takes 1-bpp tile-strip glyphs and turns them into output suitable for 16-bpp bitmap backgrounds. The output will use the ink attribute for color and it will only draw a pixel if the bit was 1 in the source data, giving us transparency. The three macros at the top declare and define the basic variables, comparable to step 1a in `bmp16_drawg()`.

```c
void bmp16_drawg_b1cts(uint gid)
{
    // (1) Basic variables
    TTE_BASE_VARS(tc, font);
    TTE_CHAR_VARS(font, gid, u8, srcD, srcL, charW, charH);
    TTE_DST_VARS(tc, u16, dstD, dstL, dstP, x0, y0);
    uint srcP= font->cellH;

    dstD += x0;

    u32 ink= tc->cattr[TTE_INK], raw;

    // (2) Rendering loops.
    uint ix, iy, iw;
    for(iw=0; iw<charW; iw += 8)           // loop over tile-strips
    {
        dstL= &dstD[iw];
        for(iy=0; iy<charH; iy++)          // loop over tile-lines
        {
            raw= srcL[iy];
            for(ix=0; raw>0; raw>>=1, ix++) // loop over tile-scanline
 (8 pixels)
                if(raw&1)
                    dstL[ix]= ink;

            dstL += dstP/2;
        }
        srcL += srcP;
    }
}
```

The routine starts by calls to three macros: `TTE_CHAR_VARS()`, `TTE_CHAR_VARS` and `TTE_DST_VARS()`. These declare and define most of the relevant local variables, similar to step 1a in `bmp16_drawg()`. Note that two of the arguments here are datatype identifiers for the source and destination pointers, respectively. `srcD` and `srcL` will initially point to the start of the source data. The other pointers, `dstD` and `dstL`, point to the start *of the scanline* in the destination area. They haven't been corrected for the *x*-position just yet; that's done right after it.

The reason I'm using two pairs of pointers here (a main *data* pointer, `fooD` and a *line* pointer `fooL`) is because of the pointer arithmetic. The data-pointer stays fixed and the line-pointer moves around in the inner loop.

The tile-strip portion of fig 22.5 illustrates how the routine moves over all the pixels. Because it's for a 1-bpp bitpacked font and because there are 8 pixels per tile-line, we can get an entire line's worth of pixels in one byte-read. Rendering transparently gives us a nice chance for optimization as well: if the tile-line is empty (i.e., `raw` ==0), we have no more visible pixels in that line and we can move on to the next. A glance at the verdana 9 font in fig 22.1, will tell you that you may be able to skip 50% of the pixels because of this.

bmp8_drawg_b1cts : 1 to 8 bpp with transparency and coloring

The 8 bpp counterpart of the previous function is called `bmp8_drawg_b1cts()` , and is given below. The code is very similar to the 16 bpp function, but because the pixels are now bytes there are a few differences in the details.

```
void bmp8_drawg_b1cts(uint gid)
{
    // (1) Basic variables
    TTE_BASE_VARS(tc, font);
    TTE_CHAR_VARS(font, gid, u8, srcD, srcL, charW, charH);
    TTE_DST_VARS(tc, u16, dstD, dstL, dstP, x0, y0);
    uint srcP= font->cellH;

    dstD += x0/2;

    u32 ink= tc->cattr[TTE_INK], raw, px;
    uint odd= x0&1;                             // (2) Source offset.

    uint ix, iy, iw;
    for(iw=0; iw<charW; iw += 8)                // Loop over strips.
    {
        dstL= &dstD[iw/2];
        for(iy=0; iy<charH; iy++)               // Loop over lines.
        {
            raw= srcL[iy]<<odd;                 // (3) Apply source offset.
            for(ix=0; raw>0; raw>>=2, ix++)     // Loop over pixels.
            {
                // (4) 2-bit -> 2-byte unpack, then used as masks.
                px= ( (raw&3)<<7 | (raw&3) ) &~ 0xFE;
                dstL[ix]= (dstL[ix]&~(px*255)) + ink*px;
            }
            dstL += dstP/2;
        }
        srcL += srcP;
    }
}
```

The only real difference with `bmp16_drawg_b1cts` is in the inner-most loop. The no-byte-write issue for VRAM means that we need to write two pixels in one pass. To do this, I retrieve and unpack two bits into two bytes and use them to create the new pixels and the pixel masks. The first line in the inner loop does the unpacking. It transforms the bit-pattern `ab` into `0000 000 a  0000 000 b`. Both bytes in this halfword are now 0 or 1, depending on whether a and b were on or off. By multiplying with `ink` and 255, you can get the colored pixels and the appropriate mask for insertion.

```
# 2-bit to 2-byte unpacking.
0000 0000  hgfe dcba     p  = raw (start)
0000 0000  0000 00ba     p &= 3
0000 000b  a000 00ba     p |= p<<7;
0000 000b  0000 000a     p &= ~0xFE;
```

Preparing the right halfword is only part of the work. If `cursorX` (i.e., `x0`) is odd, then the glyph should be plotted to an odd starting location as well. However, the destination pointer `dstL` is halfword pointer and these must always be halfword aligned. To take care of this, note that unpacking the pattern '`abcd efgh`' to an odd boundary is equivalent to unpacking '`a bcde fgh 0`' to an even boundary. This is exactly what the extra shift by `odd` is for.

Example : sub-pixel rendering

For the demo of this section, I'd like to use a technique called *sub-pixel rendering*. This is a method for effectively tripling the horizontal resolution for rendering by 'borrowing' colors from other pixels.

Consider the letter 'A' as shown in fig 22.6a. As you know, each pixel is composed of three colors: red, green and blue. These are the sub-pixels. The letter on the sub-pixel grid looks like fig 22.6b. Notice how the colors are still grouped by pixels, which on the sub-pixel grid gives very jagged edges. The trick to sub-pixel rendering is to shift groups of sub-pixels left or right, resulting in smoother edges (fig 22.6c). Now combine the pixels to RGB colors again to get fig 22.6d. Zoomed in as it is in fig 22.6, sub-pixel rendering may not look like much, but when used in the proper size the effects can be quite stunning.
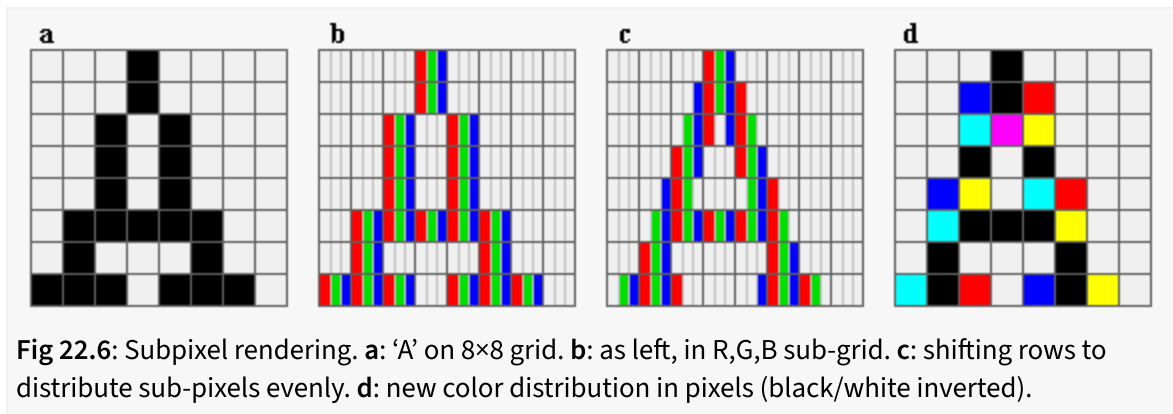
Fig 22.6: Subpixel rendering. **a**: 'A' on 8×8 grid. **b**: as left, in R,G,B sub-grid. **c**: shifting rows to distribute sub-pixels evenly. **d**: new color distribution in pixels (black/white inverted).

Sub-pixel rendering isn't useful for everything. Because it muddles the concept of pixel and color a little, it's only useful for gray-scale images. This does make it great for text, of course. Secondly, the order in which the sub-pixels are ordered also matters. The process shown in **fig 22.6** will work for RGB-ordered screens, but would fail quite spectacularly when the pixels are BGR-ordered. Going into all the gritty details it too much to do here, so I'll refer you to http://www.grc.com/ctwhat.htm, which explains the concept in more detail and gives a few examples too.

JanoS (http://www.haluz.org/yesh/) has created nice 4×8 sub-pixel font for use on GBA and NDS (see fig 22.7). A width of 4 is really tiny; it's impossible to have glyphs of that size with normal rendering and still have readable text. With sub-pixel rendering, however, it still looks good and now you can have many more characters on the screen than usual.
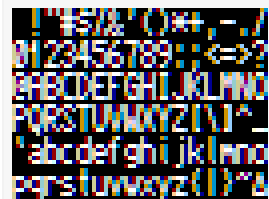
Fig 22.7: 4×8 subpixel font

The output of the demo can be seen in fig 22.8. Because sub-pixel rendering is so closely tied to the hardware you're viewing with, it will probably look crummy on most screens or paper. You really have to see it on a GBA screen for the full effect.

In this particular case, I've converted the font to work with `bmp16_drawg()` : a 16bpp font in bitmap layout. Creating an 8-bit version would not be very hard either. A 1-bpp bitpacked font would of course be impossible because the font has more than two colors. To make sub-pixel fonts look right, you'll actually need a lot of colors: one for each combination of R,G,B, and with difference shades of each. That said, JanoS has

managed to reduce the amount of colors to 20 here without too much loss in quality. If anyone wants it, I also have a 15-color version to use with 4bpp fonts.

```c
//! Testing a bitmap renderer with JanoS' sub-pixel font.
void test_tte_bmp16()
{
    irq_init(NULL);
    irq_add(II_VBLANK, NULL);
    REG_DISPCNT= DCNT_MODE3 | DCNT_BG2;

    tte_init_bmp(3, &yesh1Font, bmp16_drawg);
    tte_init_con();

    const char *str=
    "https://en.wikipedia.org/wiki/Subpixel_rendering :\n"
    "Subpixel rendering is a way to increase the "
    "apparent \nresolution of a computer's liquid crystal "
    "display (LCD).\nIt takes advantage of the fact that "
    "each pixel on a color\nLCD is actually composed of "
    "individual red, green, and\nblue subpixel stripes to "
    "anti-alias text with greater\ndetail.\n\n"
    "   4x8 sub-pixel font by JanoS.\n"
    "   http://www.haluz.org/yesh/\n";

    tte_write(str);
    key_wait_till_hit(KEY_ANY);
}
```
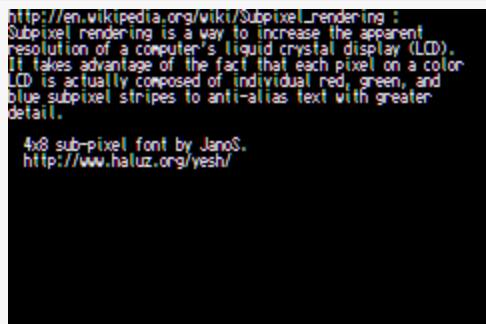

Fig 22.8: Sub-pixel rendering demo

Object text

Object text is useful if you want the characters to move around a bit, or if you simply don't have any room on a background. There are a few possibilities for object text. The most obvious one is to load all the characters into object VRAM and set the tile-indices of the objects to use the right tiles. This is what the TTE object system uses.

In many ways, this kind of object text is similar to tilemap text. The tiles are loaded up front and you change the relevant mapping entries (in this case `attr2` of the objects) to the right number. Of course, there are some notable differences as well.

For one thing, the positions of the characters must be written to the objects. But not only that, the objects also need to know how big they're supposed to be, and whether they have any other interesting qualities like rotation and palettes. For that reason, I've chosen to use the color attributes 0, 1 and 2 to store the object attributes 0, 1 and 2.

Another problem is which objects to use and how many. This last one could present a big problem, actually, because you may also want to use objects for normal sprites and it would be a really bad idea if they were suddenly overridden by the text system.

For the latter issue, I use (or perhaps abuse) the `dst` member of the context. Each glyph is represented by an object, so I'll need an object array, but I'm doing it with a little twist. I'm going to start at the *end* of the array, so that the lower objects can still be used for sprites as normal. Essentially, I'm using OAM as an empty-descending stack. In this arrangement, `dst.data` points to the top of the stack (i.e., the last element in the array), `dst.pitch` is the index to the current object, and `dst.width` is the length of the stack.

The default plotter of objects is `obj_drawg`. Remember, `dst.pitch` is used as an index here and `dst.data` is the top of the stack, so a *negative* index is used to get the current object. After that, the coordinates and the correct glyph index are merged with the color attributes to create the final object.

```
//! Glyph-plotter using objects.
void obj_drawg(uint gid)
{
    TTC *tc= tte_get_context();
    TFont *font= tc->font;
    uint x0= tc->cursorX, y0= tc->cursorY;

    // (1) find the right object, and increment index.
    uint id= tc->dst.pitch;
    OBJ_ATTR *obj= &((OBJ_ATTR*)tc->dst.data)[-id];
    tc->dst.pitch= (id+1 < tc->dst.width ? id+1 : 0);

    // (2) Set object attributes.
    obj->attr0= tc->cattr[0] + (y0 & ATTR0_Y_MASK);
    obj->attr1= tc->cattr[1] + (x0 & ATTR1_X_MASK);
    obj->attr2= tc->cattr[2] + gid*font->cellW*font->cellH/64;
}
```

And, yes, I know that this use of the `dst` member is somewhat … unorthodox; but it wasn't used here anyway so why not. I am considering using something more proper, but not just yet. Also, remember that this system assumes that the font is already loaded into VRAM and that this can take up a *lot* of the available tiles. Using the verdana 9 font, that'd be 2*240 = 480 tiles. That's nearly half of object VRAM. A safer alternative would be to load the necessary tiles dynamically, but that would require more resource management.

> **TTE OBJECT TEXT IS UGLY**
>
> The way object text is handled in TTE works, but it the implementation is not exactly pretty. The way I'm using `TTC.dst` here is, well, bad. There is a good chance I'll clean it up a bit later, or at the very least hide the implementation better.

Example: letters. Onna path

The defining characteristic of objects is that they're separate from backgrounds; they can move around the screen independently. Object text is most likely used for text

that is dynamic or has to travel along some sort of path. In this case, I'll make them fly on a parameterized path called a Lissajous curve (see fig 22.9).

The code is given below. After initializing the usual suspects, `tte_init_obj()` is called. The object stack starts at the back of OAM, which is also what the system defaults to if `NULL` passed as the first parameter. The next three are the object attributes. Because I want to use the default variable width font, verdana 9, the attributes should be set to 8×16 objects. The bitdepth of the

Fig 22.9: Object text on a path.

tiles will always be 4 to keep the number of used tiles within limits. The rest of the initialization should be easy to understand.

Making the string itself is done at step 2. Note that the string also set the paper color attribute (which corresponds to obj.attr2) to 0x1000 to make the "omg" red. After these few lines, the text handling itself is complete.

In step 3, the coordinates on the path are calculated. The `t` parameter indicates the how far along the path we are. It is used to calculate the coordinates of each letter – the first one using `t` itself, and the rest are essentially time-delayed. Don't be distracted by the magic numbers: the only reason for their values is to make the effect look alright. Try tweaking them a little to see what they do exactly.

```
// Object text demo
void test_tte_obj()
{
    // Base inits
    irq_init(NULL);
    irq_add(II_VBLANK, NULL);
    REG_DISPCNT= DCNT_MODE0 | DCNT_OBJ | DCNT_OBJ_1D;
    oam_init(oam_mem, 128);
    VBlankIntrWait();

    // (1) Init object text, using verdana 9 (8x16 objects)
    OBJ_ATTR *objs= &oam_mem[127];
    tte_init_obj(
        objs,                   // Start at back of OAM
        ATTR0_TALL,             // attr0: 8x16 objects
        ATTR1_SIZE_8,           // attr1: 8x16 objects
        0,                      // attr2: nothing special
        CLR_YELLOW,             // Yellow ink
        0x0E,                   // ink pixel 14+1 = 15
        &vwf_default,             // Verdana 9 font
        NULL);                  // Default renderer (obj_drawg)

    pal_obj_bank[1][15]= CLR_RED;

    // (2) Write something (and prep for path)
    const char *str= "Parametrized object text, omg!!!";
    const int len= strlen(str);
    tte_write("Parametrized object text, #{cp:0x1000}omg#{cp:0}!!!");

    // Play with the objects
    int ii, t= 0x9000;
    while(1)
    {
        VBlankIntrWait();
        key_poll();

        // (3) Make lissajous figure
        for(ii=0; ii<len; ii++)
        {
            int ti= t-0x380*ii;             // Get the path param for
letter ii
            obj_set_pos(&objs[-ii],
                (96*lu_cos(  ti)>>12)+120,  // y= Ay*cos(  t) + y0
                (64*lu_sin(2*ti)>>12)+80);  // x= Ax*sin(2*t) + x0
        }
        t += 0x00A0;

        if(key_hit(KEY_START))
            break;
```

```
        }
    }
```

Rendering to tiles

Using tilemaps for text is nice, but will only work if the dimensions of the glyphs are multiples of 8. There are a few drawbacks in terms of readability: narrow characters such as 'i' will seem either overly wide, or be surrounded by many empty pixels. Also, you can't put many characters on a line because there are only so many tiles.

Variable-width fonts (*vwf*; also known as proportional fonts) solve this problem. Using variable-width fonts on bitmaps is quite easy, as shown in the "Bitmapped text" section.. However, using it in tilemap modes is a little trickier: how do you draw on a tilemap where the tiles are 8×8 in size?

Well, you don't. Not exactly. The key is not to draw to the map, but to the tiles that the map shows.

Basic tile rendering

The usual way to work with tilemaps is that you load up a tileset, and then select the ones you want to show up on the screen by filling the tilemap. In those circumstances, the tileset is often static, with the map being updated for things like scrolling. Rendering to tiles reverses that procedure.

First, you need to set up a map where each entry points to a unique tile. This essentially forms a graphical surface out of the tiles, which you can then draw to like any other. The most obvious way to do this is to simply fill up the screenblock with consecutive numbers (see fig 22.10a). However, a better way to map the tiles is by mapping tiles in column-major order (see fig 22.10b) , for the same reason I chose it for the glyph format: the words in a column of tiles are consecutive.
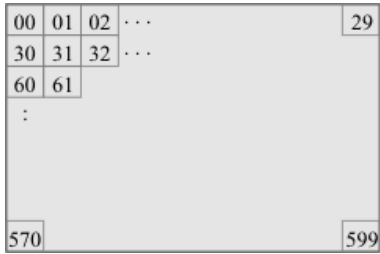
Fig 22.10a. Row-major tile indexing.
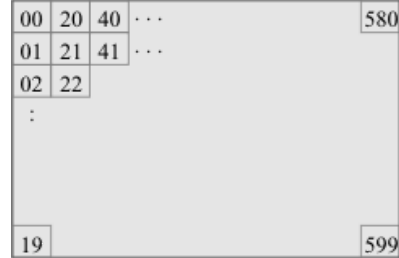
Fig 22.10b. Column-major tile indexing.

Preparing the map is the easy part; the problem is knowing which part of which tile to edit to plot a pixel. First, you need to split the coordinates into tile coordinates and pixel-in-tile coordinates. This comes down to division and modulo by 8, respectively. Note that in column-major mode you only need to do this for the x coordinate. With this information, you can find the right word. The horizontal in-tile coordinate tells you which nybble in the word to update and at that point it's the usual bitfield insertion.

Tonclib has routines for drawing onto 4bpp, column-major tiles (referred to as *chr4c* mode). The plotter and the map preparation functions are given below, along with a demonstration routine to explain their use.

```
//# From tonc_schr4c.c

//! Plot a pixel on a 4bpp tiled, column-major surface.
void schr4c_plot(const TSurface *dst, int x, int y, u32 clr)
{
    uint xx= x;       // fluff to make x unsigned.
    u32 *dstD= (u32*)(dst->data + xx/8*dst->pitch);
    uint shift= xx%8*4;

    dstD[y] = (dstD[y] &~ (15<<shift)) | (clr&15)<<shift;
}

//! Prepare a screen-entry map for use with chr4c mode.
void schr4c_prep_map(const TSurface *srf, u16 *map, u16 se0)
{
    uint ix, iy;
    uint mapW= srf->width/8, mapH= srf->height/8, mapP= srf->pitch/32;

    for(iy=0; iy<mapH; iy++)
        for(ix=0; ix<mapW; ix++)
            map[iy*32+ix]= ix*mapP + iy + se0;
}

//# --- Simple test --------------------------------------------------

void test_chr4()
{
    // (1) The usual
    irq_init(NULL);
    irq_add(II_VBLANK, NULL);
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;
    REG_BG0CNT= BG_CBB(0) | BG_SBB(31);

    pal_bg_mem[1]= CLR_RED;
    pal_bg_mem[2]= CLR_GREEN;
    pal_bg_mem[3]= CLR_BLUE;
    pal_bg_mem[4]= CLR_WHITE;

    // (2) Define a surface
    TSurface srf;
    srf_init(&srf,
        SRF_CHR4C,            // Surface type.
        tile_mem[0],          // Destination tiles.
        SCREEN_WIDTH,         // Surface width.
        SCREEN_HEIGHT,        // Surface height.
        4,                    // Bitdepth (ignored due to SRF_CHR4C).
        pal_bg_mem);          // Palette.

    // (3) Prepare the map
    schr4c_prep_map(&srf, se_mem[31], 0);
```

```
    // (4) Plot some things
    int ii, ix, iy;
    for(iy=0; iy<20; iy++)
        for(ix=0; ix<20; ix++)
            schr4c_plot(&srf, ix+3, iy+11, 4);

    for(ii=0; ii<20; ii++)
    {
        schr4c_plot(&srf, ii+4,    12, 1);  // Red line
        schr4c_plot(&srf, ii+4, ii+12, 2);  // Green line
        schr4c_plot(&srf,    4, ii+12, 3);  // Blue line
    }
}
```

The pixel plotter starts by finding the tile-column that the desired pixel is in. The column-index is simply $x/8$; this is multiplied by the pitch to get a pointer to the top of the column. Note that pitch is used a little different than usual. Normally, it denotes the number of bytes to the next scanline, but in this case it's used as the byte-offset to next tile-column. For a column-major mode, this comes down to the *height×bpp**8/8, but all that is done in `srf_init()`. Once you have the right tile, the pixel you want is in the $x\%8^{th}$ nybble, meaning the required shift for the insertion is $x\%8*4$. After that, it's just a matter of inserting the color. (For the curious: I'm casting x to unsigned int first because division and modulo will then be optimized to shifts/masks properly.)

The `schr4c_prep_map()` function just initializes the map in the order given in fig 22.10b. Well, almost.I'm also adding a value to each screen-entry like I usually do for palettes and tile-offsets.

The output of `test_chr4()` can be seen in fig 22.11a. It's a white rectangle with red, green and blue lines, as expected. Fig 22.11b is a picture taken from VBA's tile viewer, showing how the contents of the surface. Doesn't quite look what's on the screen, does it? Still, if you look closely, you can figure out how it works. Each set of 20 tiles forms one tile-column on the screen (indicated by yellow blocks). When you place these tiles on top of each other, you'll see the picture of fig 22.11a emerge.

Fig 22.11a: chr4_test() output
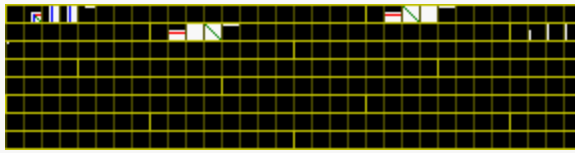
Fig 22.11b: chr4_test() tiles. The yellow blocks indicate tiles of a single column.

Text rendering on tiles

Version 1 : pixel by pixel

The easiest way to render glyphs to tiles is to follow the template from the "Bitmapped text" section.. This is done in the function below.

```c
//! Simple version of chr4 renderer.
void chr4_drawg_b1cts_base(uint gid)
{
    TTE_BASE_VARS(tc, font);
    TTE_CHAR_VARS(font, gid, u8, srcD, srcL, charW, charH);
    uint x0= tc->cursorX, y0= tc->cursorY;
    uint srcP= font->cellH;

    u32 ink= tc->cattr[TTE_INK], raw;

    uint ix, iy, iw;
    for(iw=0; iw<charW; iw += 8)
    {
        for(iy=0; iy<charH; iy++)
        {
            raw= srcD[iy];
            for(ix=0; raw>0; raw>>=1, ix++)
                if(raw&1)
                    schr4c_plot(&tc->dst, x0+ix, y0+iy, ink);
        }
        srcD += srcP;
        x0 += 8;
    }
}
```

Now, you may think that this runs pretty slowly thanks to all the recalculations in `schr4c_plot()`. And you'd be right, but in truth, it's not as bad as I originally thought.

It is possible to speed it up by simply inlining things, but the real gain comes from drawing pixels in parallel.

Version 2 : 8 pixels at once.

Instead of plotting pixel individually, you can also plot multiple pixels simultaneously. The `bmp8_drawg_b1cts()` renderer we saw earlier did this: it unpacked 2 pixels and drew together. In the case of 4bpp tiles, you can unpack the source byte into one (32bit) word and plot *eight* pixels at once. The only downside is that you'll probably have to split it over two tiles.

The next function is TTE's main glyph renderer for tiles, and it is a doozy. There are two stages for the rendering in the inner loop: bit unpacking the source byte, `raw` and splitting the prepared pixel `px` into two adjacent tiles. These correspond to steps 3 and 4, respectively.

Normally, the bitunpack is done in a loop, but sometimes it's faster to do it in other ways. For details, see my document on bit tricks. The first five lines of step 3 do the unpacking. For example, it turns a binary `0001 1011` into a hexadecimal `0x00011011`. This is then multiplied by 15 and `ink` to give the pixel mask `pxmask` and the colored pixels `px`, respectively.

Step 4 distributes the word with the pixels over two tiles if necessary. In step 1, left and right shifts were prepared to supply the bit offsets for this procedure. Now, for larger glyphs this will mean that certain destination words are used twice, but this can't be helped (actually it can, but the procedure is ugly and possibly not worth it). An alternative to this is using the destination once and read (and unpack/color) the source twice; however, as VRAM is considerably faster than ROM I doubt this would be beneficial.

```c
//! Render 1bpp fonts to 4bpp tiles; col-major order.
void chr4c_drawg_b1cts(uint gid)
{
    // Base variables.
    TTE_BASE_VARS(tc, font);
    TTE_CHAR_VARS(font, gid, u8, srcD, srcL, charW, charH);
    uint x= tc->cursorX, y= tc->cursorY, dstP= tc->dst.pitch/4;
    uint srcP= font->cellH;

    // (1) Prepare dst pointers and shifts.
    u32 *dstD= (u32*)(tc->dst.data + (y + x/8*dstP)*4), *dstL;
    x %= 8;
    uint lsl= 4*x, lsr= 32-4*x, right= x+charW;

    // Inner loop vars.
    u32 px, pxmask, raw;
    u32 ink= tc->cattr[TTE_INK];
    const u32 mask= 0x01010101;

    uint iy, iw;
    for(iw=0; iw<charW; iw += 8)    // Loop over strips
    {
        // (2) Update and increment main data pointers.
        srcL= srcD;     srcD += srcP;
        dstL= dstD;     dstD += dstP;

        for(iy=0; iy<charH; iy++)   // Loop over scanlines
        {
            raw= *srcL++;
            if(raw)
            {
                // (3) Unpack 8 bits into 8 nybbles and create the mask
                raw |= raw<<12;
                raw |= raw<<6;
                px    = raw & mask<<1;
                raw &= mask;
                px    = raw | px<<3;

                pxmask= px*15;
                px    *= ink;

                // (4a) Write left tile:
                dstL[0] = (dstL[0] &~ (pxmask<<lsl) ) | (px<<lsl);

                // (4b) Write right tile (if any)
                if(right > 8)
                    dstL[dstP]= (dstL[dstP] &~ (pxmask>>lsr) ) |
(px>>lsr);
            }
            dstL++;
```

```
            }
        }
    }
```

`chr4c_drawg_b1cts()` is pretty fast. It certainly is faster than the earlier version by about 33%. It's actually even faster than the bmp8 renderer, but only by a slim margin.

Of course, you can always go one better. The various shifts and conditionals make it perfect for ARM code, rather than Thumb. And to make sure it goes exactly according to plan, I'm doing this in assembly.

Version 3: ARM asm

The next function is `chr4_drawg_b1cts_fast()`, the ARM assembly equivalent of version 2. There's an almost one-to-one correspondence between the C and asm loops, so just loop to the C version for the explanation.

Speed-wise, the asm version is *much* better than the C version. Even in ROM, which is *very* bad for ARM code, it is still faster than the Thumb version. There are one or two tiny details by which you can speed this thing up, but by and large this should be it for fonts of arbitrary dimensions. Of course, if you have fixed sizes for your font and do not require recoloring or transparency, things will be a little different.

```
// Include TTC/TFont member offsets plz.
#include "tte_types.s"

/*
IWRAM_CODE void chr4c_drawg_b1cts_fast(int gid);
*/
    .section .iwram, "ax", %progbits
    .arm
    .align
    .global chr4c_drawg_b1cts_fast
chr4c_drawg_b1cts_fast:
    stmfd   sp!, {r4-r11, lr}

    ldr     r5,=gp_tte_context
    ldr     r5, [r5]

    @ Preload dstBase (r4), dstPitch (ip), yx (r6), font (r7)
    ldmia   r5, {r4, ip}
    add     r3, r5, #TTC_cursorX
    ldmia   r3, {r6, r7}

    @ Get srcD (r1), width (r11), charH (r2)
    ldmia   r7, {r1, r3}            @ Load data, widths
    cmp     r3, #0
    ldrneb  r11, [r3, r0]          @ Var charW
    ldreqb  r11, [r7, #TF_charW]   @ Fixed charW
    ldrh    r3, [r7, #TF_cellS]
    mla     r1, r3, r0, r1         @ srcL
    ldrb    r2, [r7, #TF_charH]    @ charH
    ldrb    r10, [r7, #TF_cellH]   @ cellH

    @ Positional issues: dstD(r0), lsl(r8), lsr(r9), right(lr), cursorX
    mov     r3, r6, lsr #16        @ y
    bic     r6, r6, r3, lsl #16    @ x

    add     r0, r4, r3, lsl #2     @ dstD= dstBase + y*4
    mov     r3, r6, lsr #3
    mla     r0, ip, r3, r0

    and     r6, r6, #7             @ x%7
    add     lr, r11, r6            @ right= width + x%8
    mov     r8, r6, lsl #2         @ lsl = x%8*4
    rsb     r9, r8, #32            @ lsr = 32-x%8*4

    ldr     r6,=0x01010101
    ldrh    r7, [r5, #TTC_ink]

    @ --- Reg-list for strip/render loop ---
    @ r0    dstL
    @ r1    srcL
```

```
@ r2     scanline looper
@ r3     raw
@ r4     px / tmp
@ r5     pxmask
@ r6     bitmask
@ r7     ink
@ r8     left shift
@ r9     right shift
@ r10    dstD
@ r11    charW
@ ip     dstP
@ lr     split indicator (right edge)
@ sp00   charH
@ sp04   deltaS = cellH-charH     (delta srcL)

    cmp     r11, #8
    @ Prep for single-strip render
    suble   sp, sp, #8
    ble     .Lyloop
    @ Prep for multi-strip render
    sub     r3, r10, r2
    mov     r10, r0
    stmfd   sp!, {r2, r3}          @ Store charH, deltaS
    b       .Lyloop

    @ --- Strip loop ---
.Lsloop:
        @ (2) Update and increment main data pointers.
        ldmia   sp, {r2, r3}       @ Reload charH and deltaS
        add     r10, r10, ip       @ (Re)set dstD/dstL
        mov     r0, r10
        add     r1, r1, r3
        sub     lr, lr, #8


        @ --- Render loop ---
.Lyloop:
            @ (3) Prep px and pxmask
            ldrb    r3, [r1], #1
            orrs    r3, r3, r3, lsl #12
            beq     .Lnopx              @ Skip if no pixels
            orr     r3, r3, r3, lsl #6
            and     r4, r3, r6, lsl #1
            and     r3, r3, r6
            orr     r3, r3, r4, lsl #3

            rsb     r5, r3, r3, lsl #4
            mul     r4, r3, r7

            @ (4a) Render to left tile
            ldr     r3, [r0]
```

```
        bic     r3, r3, r5, lsl r8
        orr     r3, r3, r4, lsl r8
        str     r3, [r0]

        @ (4b) Render to right tile
        cmp     lr, #8
        ldrgt   r3, [r0, ip]
        bicgt   r3, r3, r5, lsr r9
        orrgt   r3, r3, r4, lsr r9
        strgt   r3, [r0, ip]
.Lnopx:
        add     r0, r0, #4
        subs    r2, r2, #1
        bne     .Lyloop

    @ Test for strip loop
    subs    r11, r11, #8
    bgt     .Lsloop

    add     sp, sp, #8
    ldmfd   sp!, {r4-r11, lr}
    bx      lr

@ EOF
```

Multi-color and shaded fonts.

Bitpacked fonts will give you monochrome glyphs. If you want more colors – for shading or anti-aliasing – you'll need to use more bits. The code for this is nearly identical to the 1bpp bitpacked version; the most important differences being a different source datatype and an alternative method for finding the right mask. Oh, and you won't have to unpack the bits anymore, of course.

The following snippet shows how you can make a transparency mask out of a word of 4bit pixels. Essentially, you mask all the bits of a nybble together and mask out the other bits of that nybble. This gives 0 if the whole nybble was empty, or 1 is it wasn't. This can then again be multiplied by 15 to give the proper mask.

```
// Create pixel mask from 8x 4 bits
u32 *srcL= ...;          // Source is now 32bit.

raw      = *srcL++;      // Source word: 8x 4 bits
pxmask   = raw;
pxmask |= pxmask>>2;     // bit0 = bit0 | bit2
pxmask |= pxmask>>1;     // bit0 = bit0 | bit1 | bit2 | bit3;
pxmask &= 0x11111111;    // bit0 is 0 only if bits 0-3 were all 0
pxmask *= 15;
```

Shaded characters

No, not shad*y* characters; shad*ed* characters. What you'll often
see in games is that the text has either an outline or a bit of
shading on one side. While it is possible to create shading with a
1bpp font, it's easier to simply build it into the font itself (see fig
22.12). Because this means more colors than 1bpp can handle,
you may be tempted to use a 2bpp font here. However, unless
you are *really* stressed for memory, it's more convenient to use
4bpp here as well.
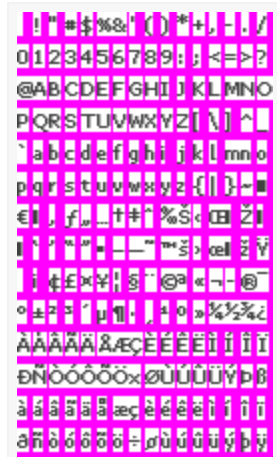
Fig 22.12: Verdana 9,
with shade.

At that point, you can follow the procedure described earlier. But by cleverly using the
bits that make up the shading, you can allow the shadow color to be variable as well.
For example, you can designate bit 0 as the 'ink' bit, bit 1 as the 'shadow' bit and if
necessary bit 2 as the 'paper' bit. Then `raw&0x11111111` gives the 'ink' mask, and
`(raw>>1)&0x11111111` gives the 'shadow' mask; these can then be used to apply
colors and to create the full mask. The following is a demonstration of how this can be
done. Note that each line here corresponds exactly to one ARM instruction, so this
should be an efficient method. Well, in ARM code anyway.

```
// Use bits 0 and 1 from each nybble to create masks and apply colors.
u32 *srcL= ...;

raw       = *srcL++;                // Source word: 8x 4 bits
px        = raw    & 0x11111111;    // Bit 0 for ink pixels
raw       = raw>>1 & 0x11111111;    // Bit 1 for shadow pixels
pxmask    = px | raw;               // Mask of ink and shadow bits
pxmask *= 15;

px        = px * ink;               // Color with ink
px       += raw* shadow;            // Add shadow pixels
```

The `chr4c_drawg_b4cts()` renderer uses this method to color both the ink and shadow pixels. It's essentially `chr4c_drawg_b4cts()` except for the things in bold and the removal of the bit unpacking. Also note the 'no-pixel' condition here. If `pxmask` is zero, there's nothing to do; and so we won't.

```
//! 4bpp font, tilestrips with ink/shadow coloring.
void chr4c_drawg_b4cts(uint gid)
{
    TTE_BASE_VARS(tc, font);
    TTE_CHAR_VARS(font, gid, u32, srcD, srcL, charW, charH);
    uint x= tc->cursorX, y= tc->cursorY;
    uint srcP= font->cellH, dstP= tc->dst.pitch/4;

    // (1) Prepare dst pointers and shifts.
    u32 *dstD= (u32*)(tc->dst.data + (y+x/8*dstP)*4), *dstL;
    x %= 8;
    uint lsl= 4*x, lsr= 32-4*x, right= x+charW;

    // Inner loop vars
    u32 amask= 0x11111111;
    u32 px, pxmask, raw;
    u32 ink=   tc->cattr[TTE_INK];
    u32 shade= tc->cattr[TTE_SHADOW];

    uint iy, iw;
    for(iw=0; iw<charW; iw += 8)    // Loop over strips
    {
        srcL= srcD;      srcD += srcP;
        dstL= dstD;      dstD += dstP;

        for(iy=0; iy<charH; iy++)   // Loop over scanlines
        {
            raw= *srcL++;

            // (3a) Prepare pixel mask
            px    = (raw     & amask);
            raw    = (raw>>1 & amask);
            pxmask= px | raw;
            if(pxmask)
            {
                px *= ink;          // (3b) Color ink pixels
                px += raw*shade;    // (3c) Color shadow pixels
                pxmask *= 15;       // (3d) Create mask

                // (4a) Write left tile:
                dstL[0] = (dstL[0] &~ (pxmask<<lsl) ) | (px<<lsl);

                // (4b) Write right tile (if any)
                if(right > 8)
                    dstL[dstP]= (dstL[dstP] &~ (pxmask>>lsr) ) |
(px>>lsr);
            }
            dstL++;
        }
    }
```

```
        }
  }
```

Tips for fast tile rendering

I've done a fair bit of profiling for these tile renderers and think I have a decent knowledge of which techniques will be efficient and which won't. These are some of my observations.

- **Profile**. Before conjuring up tricky routines, make sure the original simple version warrants optimizing *and* that the clever routine is actually faster.
- **Render transparently**. Now, you'd think that this would be slower, but it may not be. The thing about transparent text is that there are much less foreground pixels then there are background pixels, so the number of pixels to render is lower as well.
- **Don't buffer**. My first trials had separate stages for unpacking/coloring and inserting into VRAM. It put the prepared pixels into an IWRAM buffer, then copied that to VRAM. If I recall correctly, combining the loops and tossing the buffer saved me 30%.
- **Parallelize**. The road to getting the right data is long. It helps if you don't have to travel it that much. That said, if you have many empty pixels, drawing 8 of them at once may be a waste of effort. This will depend on the font.
- **ARM code is teh r0xx0rz**. There are lots of shifts, masks and quantities in these routines. This makes them particularly apt for ARM code instead of Thumb. In fact, even in ROM with its 16-bit bus, the ARM versions beat out the Thumb-compiled ones. Having said that …
- **Do not let GCC use constant masks in ARM**. There is an unfortunate bug in the ARM optimizer concerning ANDing literals (like 0x11111111). Instead of emitting a simple `ldr` + `and` pair, it will get clever and avoid the load by splitting the mask out over multiple byte-size masks. So instead of one instruction in the inner-loop, you now have four. Perhaps even more, depending on how many extra registers this takes. Note, this *only* happens for constants and only for ARM-compiled code. A work-around is to have the mask in a global variable to be

loaded before the loops. This is in part why I've hand-assembled some of the routines.

- **Code for special case if you can**. If you only have one font and don't require things like coloring, you can code for that case only and potentially save much time. Using constants for source and destination dimensions instead of using the ones in memory will also help a little.
- **Use column-major accessing**. The routines presented above require extra code to move from one tile-row to another. If you use the tiles in a column-major layout, you won't have to do this.

Please apply the standard disclaimer to this list. I've found these techniques to work for my cases, but they won't apply to every case. For example, other systems (*cough* NDS) will have different CPU architectures and memory characteristics, and that would affect the speed.

Colored text on a dialog window.

The situation depicted in fig 22.13 should be familiar. The key point here is that there is a background map, and a dialog box with text in it. This text is static, but the position in the top-left corner is continually updated as you scroll along the map.

Fig 22.13: Text on tiles.

The core function for this demo is `test_tte_chr4()`. The first thing it does is call `tte_init_chr4c()` to initialize the text system for chr4c-mode. The third argument is the offset for map-entries: `0xF000` meaning it uses sub-palette 15. The fourth is a word for the color attributes: 13 for the ink, 15 for the shadow and 0 for the others. For this demonstration, I'm using the 4bpp version of verdana9 (see fig 22.12) and the fast assembly version to render the glyphs.

Step 2 loads the background map and the dialog box. Note that the dialog box is copied to the tiles that the text is rendered to. When the text is printed, this will show

that the glyphs indeed are rendered transparently. This does more or less mean that I can't use the standard eraser, because that'd wipe the box as well.

The dialog text is drawn in step 3. The `ci` and `cs` tags set the ink and shadow color attributes, respectively. This makes the string "arrows" use colors 1 and 2 (well, 0xF1 and 0xF2), and so forth.

```
//! Set up a rectangle for text, with the non-text layers darkened for
contrast.
void win_textbox(uint bgnr, int left, int top, int right, int bottom,
uint bldy)
{
    REG_WIN0H= left<<8 | right;
    REG_WIN0V=  top<<8 | bottom;
    REG_WIN0CNT= WIN_ALL | WIN_BLD;
    REG_WINOUTCNT= WIN_ALL;

    REG_BLDCNT= (BLD_ALL&~BIT(bgnr)) | BLD_BLACK;
    REG_BLDY= bldy;

    REG_DISPCNT |= DCNT_WIN0;

    tte_set_margins(left, top, right, bottom);
}

//! Test chr4 shaded text renderer
void test_tte_chr4()
{
    irq_init(NULL);
    irq_add(II_VBLANK, NULL);
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0 | DCNT_BG2;

    // (1) Init for text
    tte_init_chr4c(
        0,                              // BG number.
        BG_CBB(0)|BG_SBB(10),           // BG control.
        0xF000,                         // Screen-entry base
        bytes2word(13,15,0,0),          // Color attributes.
        CLR_BLACK,                      // Ink color
        &verdana9_b4Font,               // Verdana 9, with shade.
        (fnDrawg)chr4c_drawg_b4cts_fast);    // b4cts renderer, asm
version
    tte_init_con();                     // Initialize console I/O

    // (2) Load graphics
    LZ77UnCompVram(dungeon01Map, se_mem[12]);
    LZ77UnCompVram(dungeon01Tiles, tile_mem[2]);
    LZ77UnCompVram(dungeon01Pal, pal_bg_mem);

    GRIT_CPY(&tile_mem[0][16*30], dlgboxTiles);
    GRIT_CPY(pal_bg_bank[15], dlgboxPal);

    // (3) Create and print to a text box.
    win_textbox(0, 8, 160-32+4, 232, 160-4, 8);
    CSTR text=
        "#{P}Scroll with #{ci:1;cs:2}arrows#{ci:13;cs:15}, "
        "quit with #{ci:1;cs:2}start#{ci:13;cs:15}\n"
```

```
        "Box opacity with #{ci:3;cs:4}L/R#{ci:7;cs:9}";
    tte_write(text);

    // Reset margins for coord-printing
    tte_set_margins(8, 8, 232, 20);

    int x=128, y= 32, ey=8<<3;

    REG_BG2HOFS= x;
    REG_BG2VOFS= y;

    // Invisible map buildup!
    REG_BG2CNT= BG_CBB(2) | BG_SBB(12) | BG_REG_64x64;
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0 | DCNT_BG2 | DCNT_WIN0;

    while(1)
    {
        VBlankIntrWait();
        key_poll();

        // (4) Scroll and blend
        x = clamp(x + key_tri_horz(), 0, 512+1-SCREEN_WIDTH);
        y = clamp(y + key_tri_vert(), 0, 512+1-SCREEN_HEIGHT);
        ey= clamp(ey+ key_tri_shoulder(), 0, 0x81);

        REG_BG2HOFS= x;
        REG_BG2VOFS= y;
        REG_BLDY= ey>>3;

        // (5) Erase and print new position.
        tte_printf("#{es;P}%d, %d", x, y);

        if(key_hit(KEY_START))
            break;
    }
}
```

I'll close off this section with a word on the text box. If you look carefully, you'll see that it's semi-transparent. Or, to be precise, the text and the box itself are at normal intensity, but the background that it covers is darker than usual. Two things are necessary for this nice, little effect.

- An inside and outside window must be defined. Both windows should contain all layers, but the inner window must be set to use blending (WIN_BLD). This enables blending for the inside only.

- The blending mode should be set to fade-to-black (BLD_BLACK) for all layers except the background with the text box.

This is what `win_textbox()` is for. The function also sets the margins so that the text would wrap nicely inside the box.

Scripting, console IO and other niceties

TTE formatting commands

The TTE context contains members that control positioning, colors, fonts as well as a few other things. There are two approaches to changing these parameters. The first is to hard code changes in the state through direct member access or functions like `tte_set_ink()`. This works nice and fast, but isn't very flexible. The second is to use *formatting tags* in the strings themselves – the system parses the string for these tags and interprets them accordingly. This is basically a form of scripting.

The tags that TTE uses look like this:

```
#{`_`tag0`_`:`_`args`_`; `_`tag1`_`:`_`args`_`}
```

The code itself is starts with ` `#{` ' and ends with ` `}` '. Each command consists of a tag, followed by a colon and comma-separated arguments when appropriate. Multiple commands can be separated by a semi-colon. For example, ` `#{es; P:10,16}` ' would clear the screen and set the cursor to (10, 16).

Now, I could show you how to parse this, but the parser currently in use for this is, well, let's just say it's long and very ugly. Essentially, it's a massive switch-block (sometimes a *double* switch-block) with stuff like this:

```c
char *tte_cmd_default(const char *str)
{
    int ch, val;
    char *curr= (char*)str, *next;

    TTC *tc= tte_get_context();

    while(1)
    {
        ch= *curr;
        next= curr+1;

        // (1) Check first character
        switch(ch)
        {
        // (2) --- Absolute Positions ---
        case 'X':
            tc->cursorX= curr[1]==':'          // If there's an
argument ...
                ? strtol(curr+2, &next, 0)     // set cursor X to arg
                : tc->marginLeft;              // else move to start of
line.
            break;

        // ... more cases ...

        // (3) Find EOS/EOC/token and act on it
        curr= tte_cmd_next(next);

        if(curr[0] == '\0')
            return curr;
        else if(curr[0] == '}')
            return curr+1;
    }
}
```

Like I said, ugly; but it'll have to do for now. The incoming pointer points to the first character past the ' `#{` '. The command tags are all single or double-lettered; the switch looks for a recognized letter and acts accordingly.

One of the tags is ' `X` ', which sets the absolute X-coordinate of the cursor. The `tc->cursorX` will be set to the argument if it is present, or to the left margin if it is not. Note the use of `strtol()` here. This is a very interesting function. Not only does it work for both decimal and hex strings, but through the second argument you can

retrieve a pointer to right after the number in the string. Alternatives would be `sscanf()` or `atoi()`, but `strtol()` is nicer.

After handling a tag, it'll look for more tags, or exit if the end delimiter or end of string is found.

Table 22.4 shows the available tags. Note that they are case-sensitive and some items can do more than one thing, depending on the number of parameters.

Code	Description
P	Reset position to top-left margin.
Pr	Restore cursor position (see also `Ps`).
Ps	Save cursor position.
P: *x,y*	Set cursor to coordinates (*x*, *y*).
X	Reset `cursorX` to left margin.
X: *x*	Set `cursorX` to *x*.
Y	Reset `cursorY` to top margin.
Y: *y*	Set `cursorY` to *y*.
c[ispx]: *cattr*	Set ink (`ci`), shadow (`cs`), paper (`cp`) or special (`cx`) color attribute to *cattr*.
e[slbf]	Erase the screen between margins (`es`), the current line (`el`), the current line up to the cursor (`eb`; backwards), the current line from the cursor (`ef`; forwards).
er: *l,t,r,b*	Erase a rectangle given by (*l,t*) to (*r,b*).
f: *idx*	Set `font` to `TTC.fontTable[`*idx*`]`.
m[ltrb]: *value*	Set left (`ml`), top (`mt`),right (`mr`) or bottom (`mb`) margin to *value*.

m: *l,t,r,b*	Set margins to rectangle (*l,t*) - (*r,b*)
p: *dx, dy*	Move the cursor by (*dx, dy*).
s: *idx*	Print the *idx*'th string in `TTC.stringTable`.
w: *count*	Wait for *count* frames.
x: *dx*	Move the cursor to the right by *dx*.
y: *dy*	Move the cursor down by *dy*.

Table 22.4: Available TTE formatting tags.

I should point out that at present the commands are still fragile, so be careful with this stuff. For example, the positioning commands will simply move the cursor, but not clip to the margins. Also take care with the font and string commands (`f` and `s`, respectively). `tte_cmd_default()` doesn't test whether the index is out of the bounds of the arrays, so you could end up with … odd things. At some point, I hope to fix these things, but it's not a priority right now. If anyone has something more robust that I can use, please speak up.

TTE FORMATTING COMMANDS : CAVEAT EMPTOR.

The current commands in TTE aren't exactly idiot-proof yet. If you stick to sensible things, it should work quite nicely. But it is still easy to shoot yourself in the foot if you're not careful.

Using console I/O

Something like `tte_write()` is nice for pure strings, but what would really help is if you had something like `printf()`. In the old days (pre-2006), `printf()`, `putc` and other console output functions were unavailable, but Wintermute added a mechanism to devkitArm's standard C library that allows it on consoles as well.

The key to this is the `devoptab_t` struct, defined in `sys/iosupport.h`. This contains a table of function pointers to device operations. The pointer we're interested in here is `write_r`; this is the function that `printf()` et al. call for the final output.

```c
// Partial devoptab_t definition
typedef struct {
    const char *name;
    int structSize;
    int (*open_r)(struct _reent *r, void *fileStruct, const char *path,
        int flags,int mode);
    int (*close_r)(struct _reent *r,int fd);
    int (*write_r)(struct _reent *r,int fd,const char *ptr,int len);
    ...
} devoptab_t;
```

The key to making the standard console routines work on a GBA is to redirect the default `write_r` for console output to one of our own making. Before explaining how this works, I want you to understand that this comes very close to black magic. It involves descending to the roots of the library and there is next to no documentation about how this stuff works. This story is the closest thing I could find to a full description: Embedding GNU: Newlib, Part 2, but this isn't high on explanations either.

To put it in other way: you're in a cave; it's pitch black and there are grues about.

Now that that's done, let's continue. The first step is creating our replacement writer. In TTE's case, this is `tte_con_write()`. It is almost identical to `tte_write()`, but has to fit in the format given by `devoptab_t.write_r`. It comes down to this:

```c
//! internal output routine used by printf.
/*! \param r     Reentrancy parameter.
    \param fd    File handle (?).
    \param text Text buffer containing the string prepared by printf.
    \param len  Length of string.
    \return      Number of output bytes (?)
    \note        \a text is NOT zero-terminated!!!!one!
*/
int tte_con_write(struct _reent *r, int fd, const char *text, int len)
{
    // (1) Safety checks
    if(!sConInitialized || !text || len<=0)
        return -1;

    int ch, gid, charW;
    const char *str= text, *end= text+len;

    // (2) check for end of text
    while( (ch= *str) != 0 && str < end)
    {
        str++;
        switch(ch)
        {

        // (3) --- VT100 sequence ( ESC[foo; ) ---
        case 0x1B:
            if(str[0] == '[')
                str += tte_cmd_vt100(str);
            break;

        //# (4) Other character cases. See tte_write()
        }
    }

    return str - text;
}
```

While I've added documentation for the arguments here, it's mostly based on guesswork. The `r` parameter contains re-entrancy information, useful if you have multiple threads. Since the GBA is a single-thread system, this should not concern us. I believe `fd` is a file handle of some sort, but since we're not writing to files this again does not concern us.

The real arguments of interest are `text` and `len`. The `text` argument points to the buffer with the string to render. In the case of `printf()`, it's the string *after* formatting: all codes like `%d` are already done. And now for the most important part: `text` is **not** null-terminated. This is why there's a length variable as well.

As far as I can tell, `printf` uses a large buffer (approximately 1300 bytes) on the stack to which it writes the formatted numbers. This buffer isn't cleared you call it again, or terminated by '\0' when sent to the writer. This has the following consequences:

- 1300 bytes is a fair bit of IWRAM. Make sure you have enough room for it. Do *not* call `printf()` from interrupts, as the routine is slow and the things can start to nest and clobber everything.
- Don't forget the `len` parameter. As the buffer isn't zeroed, remnants of old data may still be there, and you get crap.
- There's an additional potential danger with respect to parsing of formatting commands here. When strings exceed the buffer length, I imagine that it's broken up into smaller chunks. I don't know what will happen if the break occurs in the middle of a command, but I doubt it's good. Of course, you shouldn't have strings that long anyway, as the screen isn't big enough to fit them.

Aside from that, `tte_con_write()` is straightforward. As said, the contents of the loop are nearly identical to the one in `tte_write()`. The only real difference is point 3. This is a test for VT100 formatting strings, which will be covered in the next subsection.

To make use of the new writer, you have to hook it into the device list somehow. First, create a `devoptab_t` instance which the writer in the right place. There is a list of device operations called `devoptab_list`. The devices of interest are the streams `stdout` and `stderr`, which are entries `STD_OUT` and `STD_ERR` in the list. Simply point these entries to your own struct.

A second item is to set the buffers for these streams. I'm not sure this is really necessary, but that's how it's done in libgba and its author knows this system best so

I'm not going to argue here. The function for this is `setvbuf()`. You find the required initialization steps below.

```c
static int sConInitialized= 0;

const devoptab_t tte_dotab_stdout=
{
    "ttecon",
    0,
    NULL,
    NULL,
    tte_con_write,
    NULL,
    NULL,
    NULL
};

//! Init stdio capabilities for TTE.
void tte_init_con()
{
    // attach our operations to stdout and stderr.
    devoptab_list[STD_OUT] = &tte_dotab_stdout;
    devoptab_list[STD_ERR] = &tte_dotab_stdout;

    // Set buffers.
    setvbuf(stderr, NULL , _IONBF, 0);
    setvbuf(stdout, NULL , _IONBF, 0);

    sConInitialized = 1;
}
```

Calling `tte_init_con()` activates stdio's functionality so you can use `printf()` and such. Note that the raw `printf()` is rather heavy and it also has floating point options, which are rarely used in a GBA environment, if ever. For that reason, you'll usually use its integer-only cousin, `iprintf()`. Also note that TTE's implementation is **different** from libgba's, and the two should not be confused. For that reason, I've hidden the `iprintf()` name behind a `tte_printf` macro.

The following is a short example of its use. I'm using `tte_printf()` here, but `printf()` or `iprintf()` would have worked just as well.

```c
#include <stdio.h>
#include <tonc.h>

int main()
{
    REG_DISPCNT= DCNT_MODE0 | DCNT_BG0;

    // Init BG 0 for text on screen entries.
    tte_init_se_default(0, BG_CBB(0)|BG_SBB(31));

    // Enable TTE's console functionality
    tte_init_con();

    tte_printf("#{P:72,64}");       // Goto (72, 64).
    tte_printf("Hello World!");     // Print "Hello world!"

    while(1);

    return 0;
}
```

PRINTF BAGAGE

As wonderful as `printf()` is, there are some downsides to it too. First, it's a very heavy function that calls quite a large amount of functions which all have to be linked in. Second, it is pretty damn slow. Because it can do so much, it has to check for all these different cases. Also, for the string to decimal conversion it uses divisions, which is really bad for the GBA.

Be aware of how much `printf()` costs. If it turns out to be a bottle-neck, try making your own slimmed down version. A decent `sprintf()` alternative is `posprintf()`, created by Dan Posluns.

VT100 escape sequences

Every book on C will tell you that you can place text on a console screen. What they usually don't tell you is that, in some environments, you can control formatting as well. One such environment is the VT100, which used *escape sequences* to indicate

formatting. The libraries that devkitPro distributes for various systems use these sequences, so it's a good idea to support them as well.

The general format for the codes is this:

```
CSI n1;n2 ... letter
```

CSI here is the ASCII code for the ***command sequence indicator***, which in this case is the escape character (27, 0x1B or 033) followed by '['. The letter at the end denotes the kind of formatting code, and *n1, n2* … are the formatting parameters. Wikipedia has a nice overview of the standard set here and there's another one here: VT100 commands and control sequences. Note that not all of the codes are supported in the devkitPro libraries. The ones you'll encounter most are the following:

ESC[*dy*A	Move cursor up *dy* rows.
ESC[*dy*B	Move cursor down *dy* rows.
ESC[*dx*C	Move cursor right *dx* columns.
ESC[*dx*D	Move cursor left *dx* columns.
ESC[*y;x*H	Set cursor to column *x*, row *y*.
ESC[2J	Erase screen.
ESC[*n*K	0. Erase to end of line. 1. Erase to start of line. 2. Erase whole line.
ESC[*y;x*f	As ESC[*y;x*H
ESC[s	Save cursor position.
ESC[u	Restore cursor position.

Table 22.5: Common VT100 sequences

If you compare this list to table 22.4, you'll see that most of these codes have corresponding TTE commands. You can use either, but if you plan to make something

that's supposed to be cross-platform, use the VT100 codes.

> **DEVIATIONS FROM THE STANDARD**
>
> I'm trying to keep my implementation as close to the standard as possible. This is mainly because TTE uses other things just 8x8 characters on a regular background. In particular, scrolling is absent here and there are no color codes. Yet.

UTF-8

You may have heard of a little thing called ASCII. This is (or was; I'm not sure) the standard encoding for character strings. Each character is 1 byte long, giving 256 numbers for letters, numbers et cetera. Fig 22.1 and fig 22.12 contain character 32 to 255, as they usually appear on Windows. ASCII works fine for Western languages but are completely inadequate for languages like Japanese, which have thousands of characters. To remedy this, they came up with Unicode, which has 16 bits per character.

An intermediate between this is UTF-8. This still uses 8-bit characters for the lower 128 ASCII codes, but bytes over 0x80 denote the start of a multi-byte code, where it and a few of the following characters form a single, larger character of up to 21 bits.

UTF-8 is a nice way of having your cake and eating it too: you can still use normal characters for Latin characters, meaning it'll still work with ASCII programs, but you also have a method of representing bigger numbers.

String (binary)	Number (binary)	Range (hex)
0zzzzzzz	0zzzzzzz	0x000000 - 0x00007F (7 bit)
110yyyyy 10zzzzzz	00000yyy yyzzzzzz	0x000080 - 0x0007FF (11 bit)

1110xxxx 10yyyyyy 10zzzzzz	xxxxyyyy yyzzzzzz	0x000800 - 0x00FFFF (16 bit)
11110www 10xxxxxx 10yyyyyy 10zzzzzz	000wwwxx xxxxyyyy yyzzzzzz	0x010000 - 0x10FFFF (21 bit)

Table 22.6. UTF-8 to u32 conversion table.

Table 22.6 shows the conversion works. If a byte is lower than 128, it's a simple ASCII character. If it's higher, it can fall into three classes of multi-byte numbers. The range of the byte determines the number of bytes for the whole thing; once you know that, you need to grab the appropriate bit-patterns from these bytes and join them into a single number as the table indicates. For more details, I will refer you to the wikipedia page.

Below you can find a routine that reads and decodes a single utf-8 character from a string. Yes, it's a cluster-f**k of conditions, but that's necessary to check whether all the characters really follow the format; and if it doesn't, it'll interpret the first byte of the range as an extended ASCII character. If you want, you can omit all the `if((*src>>6)!=2) break;` statements.

```c
//! Retrieve a single multibyte utf8 character.
uint utf8_decode_char(const char *ptr, char **endptr)
{
    uchar *src= (uchar*)ptr;
    uint ch8, ch32;

    // Poor man's try-catch.
    do
    {
        ch8= *src;
        if(ch8 < 0x80)                      // 7bit
        {
            ch32= ch8;
        }
        else if(0xC0<=ch8 && ch8<0xE0)      // 11bit
        {
            ch32  = (*src++&0x1F)<< 6;  if((*src>>6)!=2)    break;
            ch32 |= (*src++&0x3F)<< 0;
        }
        else if(0xE0<=ch8 && ch8<0xF0)      // 16bit
        {
            ch32  = (*src++&0x0F)<<12;  if((*src>>6)!=2)    break;
            ch32 |= (*src++&0x3F)<< 6;  if((*src>>6)!=2)    break;
            ch32 |= (*src++&0x3F)<< 0;
        }
        else if(0xF0<=ch8 && ch8<0xF8)      // 21bit
        {
            ch32  = (*src++&0x0F)<<18;  if((*src>>6)!=2)    break;
            ch32 |= (*src++&0x3F)<<12;  if((*src>>6)!=2)    break;
            ch32 |= (*src++&0x3F)<< 6;  if((*src>>6)!=2)    break;
            ch32 |= (*src++&0x3F)<< 0;
        }
        else
            break;

        // Proper UTF8 char: set endptr and return
        if(endptr)
            *endptr= (char*)src;

        return ch32;
    } while(0);

    // Not really UTF: interpret as single byte.
    src= (uchar*)ptr;
    ch32= *src++;
    if(endptr)
        *endptr= (char*)src;
```

```
        return ch32;
}
```

Both `tte_write()` and `tte_write_con()` use `utf_decode_char()` when the string requires it. The larger characters can be used to access larger font sheets. You could use the larger sheets for better language support, or perhaps to extend the standard set of characters with arrows, and other types of symbols.

There is, however, one catch to using UTF-8 with stdio. Internally, stdio is really picky about what's acceptable. For example, the copywrite symbol, © is extended number 0xA9. In non-UTF-8, you could use just 0xA9 in a string and it'd use the right symbol. However, 0xA9 alone wouldn't fit any of the formats from Table 22.6, so it's an invalid code in UTF-8. While `utf8_decode_char()` is forgiving in this case, stdio isn't, an will interpret it as a terminator. In other words, be careful with extended ASCII character; you *have* to use the proper UTF-8 formats if you want to use the stdio functions.

> **PRINTF, UTF-8, AND EXTENDED ASCII**
>
> As of devkitArm r22, `printf()` and the other stdio functions use the UTF-8 locale. This effectively means that you cannot use characters like '©' and 'è' directly like you used to in older versions. You need to use the full multi-byte UTF-8 notations.

Profiling the renderers

It's always a good idea to see how fast the things you make are. This is particularly true when the functions are complex, like most of the bitmap and tile renderers are.

Table 22.7 lists the cycles per glyph for the majority of the available renderers. These have been measured with the string (and library code) in ROM with the default waitstates, under -O2 optimization. The font used was verdana 9, with has a cell size of 8x16, meaning it can be used for both fixed and variable widths with ease. The test string was a 194 character line from Portal:

"Please note that we have added a consequence for failure. Any contact with the chamber floor will result in an 'unsatisfactory' mark on your official testing record followed by death. Good luck!"

Renderer	Cycles/char
null	221
se_drawg	595
se_drawg_w8h16	370
ase_drawg_w8h16	458
chr4_drawg_b1cts_base	3049
chr4_drawg_b1cts	2044
chr4_drawg_b1cts_fast	631
bmp8_drawg_b1cts_base	2875
bmp8_drawg_b1cts	2078
bmp8_drawg_b1cts_fast	619
bmp16_drawg_b1cts_base	2456
bmp16_drawg_b1cts	1503
obj_drawg	423

Table 22.7: Renderer times. Conditions: 194 chars, verdana 9, ROM code, default waits, -O2.

First, note the great differences in values: from hundreds for the tilemaps and objects to *thousands* in the case of bitmaps and tile renderers. And this is per character, so writing large swats of text can lead to significant slowdown.

The `null()` renderer is a dummy renderer, used to find the overhead of the TTE system. 200 isn't actually that bad, all things considered (remember: ROM code). That said, now compare this number to the regular tilemap time: the overhead is takes up

a significant fraction of the time here. Also note the difference between the standard and 8×16 versions of `se_drawg` : this is purely due the loops

Half of the TTE overhead actually comes from the wrapping code; cursor setting and checking can be relatively slow. And I'm not even considering clipping here.

For the bitmap and tile renderers, I've timed three versions. A 'base' version, using the template from `chr4_drawg_b1cts_base()` in the "Text rendering on tiles" section.; C-optimized versions, which are the default renderers; and a fast asm version.

The `bmp16` variants are faster than the others because you don't have to mask items into the surface. What's interesting, though, is that the difference between `bmp8` and `chr4` is practically zero. This probably has something to do with the layout of the font itself.

Also note how the base, the normal and the fast versions compare. `chr4_drawg_b1cts()` is 33% faster than the base version, and `chr4_drawg_b1cts_fast` is three times faster still. And remember, 200 of that 631 is TTE overhead, so it's actually 4.5 times faster. This is not just from the IWRAM benefit: it also has to do with ARM vs Thumb, and hand-crafted assembly vs compiled code.

Conclusions

As far as I'm concerned, this chapter is basically the earlier text chapter done right. It's covered all types of graphics:regular/affine tilemaps, 8bpp/16bpp bitmaps, 4bpp tiles and objects. Okay, so I left 8bpp tiles out, but that's an awful mode for tile-rendering anyway. The functions for glyph rendering given here are work for arbitrary sizes, fixed and variable width fonts and should be doing so efficiently as well.

Furthermore, it has presented Tonc's Text Engine, a system for handling all these different text families with relative ease. After the initial set-up, the surface-specific aspects are basically dealt with, making its functionality much more re-usable. I've

also covered the most basic aspects of processing strings for printing: how to translate from a UTF-8 encoded character to a glyph-index in a font-sheet, and how you can implement formatting tags to change positions, colors and fonts dynamically. Lastly, I illustrated how you can build a callback that the stdio routines can call for output, making `printf()` and its friends available for general use.

This whole chapter has been a showcase for TTE and what it can do. Even though it's not in a fully finished state, I think that it can be a valuable asset for dealing with text. If nothing else, the concepts put forth here should help you design your own glyph renderers or text systems.

23. Whirlwind Tour of ARM Assembly

Introduction

Very broadly speaking, you can divide programming languages into 4 classes. At the lowest level is machine code: raw numbers that the CPU decodes into instructions to execute. One step up is assembly. This is essentially machine code in words: each assembly instruction corresponds to one machine code instruction. Above this are compiled languages like C, which use structured language element to read more like English, but need to be compiled to machine code to be able to run. Finally, there are scripted languages like PHP (and usually VB and Java) which are run through interpreters configured to run the right kinds of machine code for the desired effects.

Every step up the ladder increases the human readability factor and portability, at the cost of runtime speed and program size. In the old days, programmers were Real Programmers and did their work in machine code or assembly because of clock speed and/or memory constraints. For PCs, these days are long gone and most work is done in the higher level languages. This, admittedly, is a good thing: code can be written faster and maintained more easily. However, there are still a few instances where the higher languages are insufficient. The GBA, with its 16.7Mhz CPU and less than 1 MB of work RAM is one of them. Here the inefficiency of the highest languages will cost you dearly, if it'd run at all. This is why most GBA work is done in C/C++, sometimes

affectionately nicknamed 'portable assembly', because it still has the capability of working with memory directly. But sometimes even that isn't enough. Sometimes you *really* have to make every cycle count. And for this, you need **assembly**.

Now, in some circles the word "assembly" can be used to frighten small programmers. Because it is so closely tied to the CPU, you can make it do everything; but that also means you *have* to do everything. Being close to hardware also means you're bypassing all the safety features that higher languages may have, so that it's *much* easier to break things. So yeah, it is harder and more dangerous. Although some may prefer the term 'adventurous'.

To program in assembly, you need to know how a processor actually works and write in a way it can understand, rather than rely on a compiler or interpreter to do it for you. There are no structured for- or while- loops or even if/else branches, just `goto`; no structs or classes with inheritance, and even datatypes are mostly absent. It's anarchy, but the lack of bureaucracy is exactly what makes fast code possible.

Speed/size issues aside, there are other reasons why learning assembly might be a good idea. Like I said, it forces you to actually *understand* how the CPU functions, and you can use that knowledge in your C code as well. A good example of this is the 'best' datatype for variables. Because the ARM processor is 32bit, it will prefer ints for most things, and other types will be slower, sometimes much slower. And while this is obvious from the description of the processor itself, knowledge of assembly will show you *why* they are slower.

A third reason, and not an inconsiderable one, is just for general coolness `=B)`. The very fact that it is harder than higher languages should appeal to your inner geek, who relishes such challenges. The simplicity of the statements themselves have an aesthetic quality as well: no messing about with classes, different loop styles, operator precedence, etc – it's one line, one opcode and never more than a handful of parameters.

Anyway, about this chapter. A complete document on assembly is nothing less than a full user's manual for a CPU. This would require an entire book in itself, which is not

something I'm aiming at. My intention here is to give you an introduction (but a thorough one) to ARM assembly. I'll explain the most important instructions of the ARM and Thumb instruction sets, what you can and cannot do with them (and a little bit about why). I'll also cover how to use GCC's assembler to actually assemble the code and how to make your assembly and C files work together. Lastly, I'll give an example of a fast memory copier as an illustration of both ARM and Thumb code.

With that information, you should be able to do a lot of stuff, or at least know how to make use of the various reference documents out there. This chapter is not an island, I am assuming you have some or all of the following documents:

- The rather large official ARM7TDMI Technical manual: DDI0210B.pdf.
- GBATEK's ARM CPU reference: ARM + Thumb.
- Official ARM quick-references (PDF): ARM + Thumb
- Re-eject's quick-references (PDF): GAS / ARM / Thumb. (note: minor syntax discrepancies at times)
- GNU Assembler manual: GAS.

If you want more ARM/Thumb guides, you'll have to find them yourself.

General assembly

Assembly is little more than a glorified macro language for machine code. There is a one-to-one relationship between the assembly instructions and the actual machine code and assembly uses *mnemonics* for the operations the processor is capable of, which are much easier to remember than the raw binary. The tool that converts the asm code into machine code is the *assembler*.

Basic operations

Every processor must be able to do basic data processing: arithmetic and bit manipulation. They should also have instructions to access memory, and be able to

jump from one place in the code to another of conditionals and loops and such. However, different processors will have different ways of doing these things, and some operations of one set might not be present in another. For example, ARM lacks a division instruction, and can't perform data processing on memory directly. However, the ARM instruction set has some benefits too, like a fair amount of general-purpose registers and a simple instruction set, for the proper definition of "simple". And it has a *very* nifty way of dealing with bit-shifts.

In the snippet below you can find a few examples of additions and memory reads in three different assembly languages: x86 (Intel), 68000, and ARM. The basic format is usually something like '`operation operand1, operand2, ...`', though there are always exceptions. Note that where x86 and ARM put the destination in *Op1*, 68000 asm puts it in the last. The terminology of the registers is also different. Some semantics are pretty universal, the addition '`x += y` is found in all three, for example, but x86 also has a special instruction for increments by one, and in ARM the result register can be different from the two operands. These differences correspond to how the processors actually work! Higher languages allow you to use operations that do not seem present in the instruction set, but in fact they only *appear* to do so: the compiler/interpreter will convert it to a form the processor can actually handle.

Another point I must make here is that even for a given processor, there can be differences in how you write assembly. Assemblers aren't difficult to write, and there's nothing to stop you from using a different kind of syntax. Apart from the wrath of other programmers, of course.

```
// Some examples
// Addition and memory loads in different assemblies

// === x86 asm ===========================================================
add     eax, #2             // Add immediate:    eax += 2;
add     eax, ebx            // Add register:     eax += ebx;
add     eax, [ebx]          // Add from memory: eax += ebx[0];
inc     eax                 // Increment:        eax++;

mov     eax, DWORD PTR [ebx]        // Load int from memory:    eax=
ebx[0];
mov     eax, DWORD PTR [ebx+4]      // Load next int:           eax=
ebx[1];

// === 68000 asm =========================================================
ADD     #2, D0              // Add immediate:   D0 += 2;
ADD     D1, D0              // Add register:    D0 += D1;
ADD     (A0), D0            // Add from memory: D0 += A0[0];

MOVE.L  (A0), D0            // Load int from memory:    D0= A0[0];
MOVE.L  4(A0), D0           // Load next int:           D0= A0[1];

// === ARM asm ===========================================================
add     r0, r0, #2          // Add immediate:   r0 += 2;
add     r0, r0, r1          // Add register:    r0 += r1;
add     r0, r1, r2          // Add registers:   r0= r1 + r2;

ldr     r0, [r2]            // Load int from memory:    r0= r2[0];
ldr     r0, [r2, #4]        // Load int from memory:    r0= r2[1];
ldmia   r2, {r0, r1}        // Load multiple:           r0= r2[0]; r1=
r2[1];
```

Variables: registers, memory and the stack

In HLLs you have variables to work on, in assembly you can have registers, variables (that is, specific ranges in memory), and the stack. A *register* is essentially a variable inside the chip itself, and can be accessed quickly. The downside is that there are usually only a few of them, from just one to perhaps a few dozen. Most programs will require a lot more, which is why you can put variables in addressable memory as well. There's a lot more bytes in memory than in registers, but it'll also be slower to use. Note that both registers and memory are essentially **global** variables, change them in one function and you'll have changed them for the rest of the program. For local variables, you can use the stack.

The *stack* is a special region of memory used as, well, a stack: a Last-In, First-Out mechanism. There will be a special register called the **stack pointer** (SP for short) which contains the address of the top of the stack. You can *push* variables onto the top of the stack for safe keeping, and then *pop* them off once you're done with them, restoring the registers to their original values. The address of the stack (that is, the top of the stack, the contents of SP) is not fixed: it grows as you move deeper in the code's hierarchy, and shrinks as you move out again. The point is that each block of code should clean up after itself so that the stack pointer is the same before and after it. If not, prepare for a spectacular failure of the rest of the program.

For example, suppose you have functions `foo()` and which uses registers A, B, C and D. Function `foo()` calls function `bar()`, which also uses A, B and C, but in a different context than `foo()`. To make sure `foo()` would still work, `bar()` pushes A, B and C onto the stack at its start, then uses them the way it wants, and then pops them off the stack into A, B and C again when it ends. In pseudo code:

```
// Use of stack in pseudo-asm

// Function foo
foo:
    // Push A, B, C, D onto the stack, saving their original values
    push    {A, B, C, D}

    // Use A-D
    mov     A, #1          // A= 1
    mov     B, #2          // B= 2
    mov     C, #3          // well, you get the idea
    call    bar
    mov     D, global_var0

    // global_var1 = A+B+C+D
    add     A, B
    add     A, C
    add     A, D
    mov     global_var1, A

    // Pop A-D, restoring then to their original values
    pop     {A-D}
    return

// Function bar
bar:
    // push A-C: stack now holds 1, 2, 3 at the top
    push    {A-C}

    // A=2; B=5; C= A+B;
    mov     A, #2
    mov     B, #5
    mov     C, A
    add     C, B

    // global_var0= A+B+C (is 2*C)
    add     C, C
    mov     global_var, C

    // A=2, B=5, C=14 here, which would be bad when we
    // return to foo. So we restore A-C to original values.
    // In this case to: A=1, B=2, C=3
    pop     {A-C}
    return
```

While the syntax above is asm-like, it's not actually part of any assembly – at least not as far as I know. It is also particularly *bad* assembly, because it's inefficient in its use of

registers, for one. If you were to write the corresponding C code and compile it (with optimizations, mind you), you get better code. But the point was here stack-use, not efficiency.

What you see here is that `foo()` sets A, B and C to 1, 2 and 3, respectively (`mov` stands for 'move', which usually comes down to assignment), and then calls `bar()` , which sets them to something else and sets a global variable called `global_var0` to A+B+C. Because A, B and C are now different from what they were `foo()` , that function would use the wrong values in later calculations. To counter that, `bar()` uses the stack to save and restore A, B and C so that functions that call `bar()` still work. Note that `foo()` also uses the stack for A, B, C and D, because the function calling `foo()` may want to use them as well.

Stacking registers inside the called function is only a *guideline*, not a law. You could make the caller save/restore the variables that it uses. You could even not use the stack at all, as if you meant A, B and C to change and consider them return values of the function. By not setting the registers manually in `bar()` , A and B would effectively be function arguments. Or you could use the stack for function arguments. And return values. Or use both registers and the stack. The point is, you are free to do deal with them in any way you want. At least, in principle. In practice, there are guidelines written down by the original manufacturers, and while not written in stone, it can be considered bad form not to adhere to them. And you can see just *how* bad a form if you intend to make the code interface with compiled code, which *does* adhere to them.

Branching and condition codes

The normal operation for a computer is to take instructions one by one and execute them. A special register known as the ***program counter*** (PC) indicates the address of the next instruction. When it's time, the processor reads that instruction, does its magic and increments the program counter for the next instruction. This is a relatively straightforward process; things start to get interesting when you can set the program

counter to a completely different address, redirecting the flow of the program. Some might say that you can really only speak of a *computer* if such a thing is possible.

The technical term for this redirection is ***branching***, though the term 'jump' is used as well. With branching you can create things like loops (infinite loops, mind you) and implement subroutines. The usual mnemonic for branching is something like `b` or `j(mp)`

```
// Asm version of the while(1) { ... } endless loop

// Label for (possible) branching destination
endless:

    ...          // stuff

    b endless // Branch to endless, for an endless loop.
```

The full power of branching comes from branching only when certain ***conditions*** are met. With that, you can perform if-else blocks and loops that can actually end. The conditions allowed depend on the processor, but the most common ones are:

- **Zero** (Z). If the result of operation was 0.
- **Negative** (N). Result was negative (i.e. most significant bit set).
- **Carry bit set** (C). If the 'mostest' significant bit is set (like bit 32 for 32bit operations).
- **Arithmetic overflow** (V). Like adding two positive numbers and getting a negative number because the result got too big for the registers.

These condition flags are stores in the *Program Status Register* (PSR), and each data processing instruction will set these one of more of these flags, depending on the outcome of the operation. Special versions of the branch instruction can use these flags to determine whether to make the jump.

Below you can see a simple example of a basic for-loop. The `cmp` instruction compares `A` to 16 and sets the PSR flags accordingly. The instruction `bne` stands for 'branch if Not Equal', which corresponds to a clear Z-flag. the reason for the Zero-flag's

involvement is that the equality of two numbers is indicated by whether the difference between them is zero or not. So if there's a difference between `A` and 16, we jump back to `for_start` ; if not, then we continue with the rest of the code.

```
// Asm version of for(A=0; A != 16; A++)

    mov     A, #0
// Start of for-loop.
for_start:

    ...             // stuff

    add     A, #1
    cmp     A, #16   // Compare A to 16
    bne for_start    // Branch to beginning of loop if A isn't 16
```

The number and names of the conditional codes depends on the platform. The ARM has 16 of these, but I'll cover these later.

An example: GCC generated ARM assembly

Before getting into ARM assembly itself, I'd like to show you a real-life example it. Assembly is an intermediary step of the build process, and you can capture GCC's assembly output by using the ' `-S` ' or ' `-save-temps` ' flags. This gives you the opportunity to see what the compiler is actually doing, to compare the C and assembly versions of a given algorithm, and provides quick pointers on how to code non-trivial things in assembly, like function calling, structures, loops etc. This section is optional, and you may not understand all the things here, but it is very educational nonetheless.

```
# Makefile settings for producing asm output
    $(CC) $(RCFLAGS) -S $<
```

```
// gen_asm.c :
//   plotting two horizontal lines using normal and inline functions.
#include <tonc.h>

void PlotPixel3(int x, int y, u16 clr)
{
    vid_mem[y*240+x]= clr;
}

int main()
{
    int ii;

    // --- using function ---
    ASM_CMT("using function");
    for(ii=0; ii<240; ii++)
        PlotPixel3(ii, 16, CLR_LIME);

    // --- using inline ---
    ASM_CMT("using inline");
    for(ii=0; ii<240; ii++)
        m3_plot(ii, 12, CLR_RED);

    while(1);

    return 0;
}
```

```
@@ gen_asm.s :
@@ Generated ASM (-O2 -mthumb -mthumb-interwork -S)
@@ Applied a little extra formatting and comments for easier reading.
@@ Standard comments use by @; my comments use @@

@@ Oh, and DON'T PANIC! :)

    .code   16
    .file   "gen_asm.c"      @@ - Source filename (not required)
    .text                    @@ - Code section (text -> ROM)

@@ <function block>
    .align  2                @@ - 2^n alignment (n=2)
    .global PlotPixel3        @@ - Symbol name for function
    .code   16               @@ - 16bit Thumb code (BOTH are required!)
    .thumb_func              @@ /
    .type   PlotPixel3, %function   @@ - symbol type (not req)
@@ Declaration : void PlotPixel3(int x, int y, u16 clr)
@@ Uses r0-r3 for params 0-3, and stack for param 4 and over
@@   r0: x
@@   r1: y
@@   r2: clr
PlotPixel3:
    lsl     r3, r1, #4       @@ \
    sub     r3, r3, r1       @@ - (y*16-1)*16 = y*240
    lsl     r3, r3, #4       @@ /
    add     r3, r3, r0       @@ - (y*240+x)
    mov     r1, #192         @@ - 192<<19 = 0600:0000
    lsl     r1, r1, #19      @@ /
    lsl     r3, r3, #1       @@ - *2 for halfword, not byte, offset
    add     r3, r3, r1
    @ lr needed for prologue
    strh    r2, [r3]         @@ store halfword at vid_mem[y*240+x]
    @ sp needed for prologue
    bx  lr
    .size   PlotPixel3, .-PlotPixel3    @@ - symbol size (not req)
@@ </ function block>

    .align  2
    .global main
    .code   16
    .thumb_func
    .type   main, %function
main:
    push    {r4, lr}                @@ Save regs r4, lr
    @ --- using function ---       @@ Comment from ASM_CMT, indicating
    .code   16                      @@   the PlotPixel3() loop
    mov     r4, #0                  @@ r4: ii=0
.L4:
    mov     r2, #248                @@ - r2: clr= 248*4= 0x03E0= CLR_LIME
```

```
        lsl     r2, r2, #2      @@ /
        mov     r0, r4          @@ r0: x= ii
        mov     r1, #16         @@ r1: y= 16
        add     r4, r4, #1      @@ ii++
        bl      PlotPixel3      @@ Call PlotPixel3 (params in r0,r1,r2)
        cmp     r4, #240        @@ - loop while(ii<240)
        bne .L4                 @@ /
    @ --- using inline ---      @@ Comment from ASM_CMT, indicating
    .code   16                  @@   the m3_plot() loop
    ldr     r3, .L14            @@ r3: starting/current address
(vid_mem[12*240])
    ldr     r2, .L14+4          @@ r2: terminating address
(vid_mem[13*240])
    mov     r1, #31             @@ r1: clr (CLR_RED)
.L6:
        strh    r1, [r3]        @@ - *r3++ = clr
        add     r3, r3, #2      @@ /
        cmp     r3, r2          @@ - loop while(r3<r2)
        bne .L6                 @@ /
.L12:
    b   .L12
.L15:
    .align  2
.L14:
    .word   100669056           @@ 0600:1680 =&vid_mem[12*240]
    .word   100669536           @@ 0600:1886 =&vid_mem[13*240]
    .size   main, .-main

    .ident   "GCC: (GNU) 4.1.0 (devkitARM release 18)"
```

After the initial shock of seeing a non-trivial assembly file for the first time, you may be able to notice a few things, even without any prior knowledge of assembly.

- First, the assembly is much longer than the C file. This is not surprising as you can only have one instruction per line. While it makes the file longer, it also makes parsing each line easier.

- There are four basic types of line formats: labels (lines ending in a colon ':'), and instructions, and then in lines starting with a period or not. The instructions that start with a period are not really instructions, but *directives*; they are hints to the assembler, not part of the CPU's instruction set. As such, you can expect them to differ between assemblers.

The real instructions are usually composed of a mnemonic (`add`, `ldr`, `b`, `mov`) followed by register identifiers, numbers or labels. With a little thought, you should be able to piece together what each of these might do. For example, `add` performs an addition, `ldr` read something from memory, `b` branches, i.e. jumps to another memory address and `mov` does an assignment.

- Function structure and calling. In GAS, a function is preceded by a number of directives for alignment, code section and instruction set, and a '`.global`' directive to make it globally visible. And a label to mark the start of the function of course. Note that for thumb functions, require a '`.thumb_func`' directive as well as either '`.code 16`' or '`.thumb`'. GCC also inserts a size info, but this are not required.

  Calling and returning from functions uses the `bl` and `bx` instructions. What isn't very clear from this code, except in my added comments, is that the arguments of the functions are put in registers r0-r3. What you definitely don't see is that if there are more than 4 parameters, these are put on the stack, and that the return value is put in r0.

  You *also* don't see that r0-r3 (and r12) are expected to be trashed in each function, so that the functions calling them should save their values if they want to use them after the call. The other registers (r4-r15) should be pushed into the stack by the called function. The standard procedure for function calling can be found in the AAPCS. Failure to adhere to this standard will break your code if you try to combine it with C or other asm.

- Loading the CLR_LIME color (0x03E0) doesn't happen in one go, but is spread over two instructions: a move and a shift. Why not move it in one go? Well, because it can't. The ARM architecture only allows byte-sized immediate values; bigger things have to be constructed in other ways. I'll get back to this later.

- The last thing I'd like to mention is the performance of the `PlotPixel3()` loop versus the `m3_plot()` loop, which you can find in the assembly because I've

used a macro that can write asm comments in C. The `m3_plot()` loop contains 4 instructions. The `PlotPixel3()` loop takes 8, plus an additional 10 from the function itself. So that's 4 instructions against 18 instructions. The C code *seems* pretty much the same, so what gives?

Welcome to the wonderful world of *function call overhead*. In principle, you only need the instructions of the shorter loop: a store, an add for the next destination, a compare and a loop-branch. Because `m3_plot()` is inlined, the compiler can see that this is all that's required and optimize the loop accordingly.

In contrast, because `PlotPixel()` is a full function, the caller does not know what its internal code is, hence no optimizations are possible. The loop should reset the registers on *every iteration* because `PlotPixel()` will clobber them, making the loop in `main()` unnecessarily long. Furthermore, `PlotPixel3()` doesn't know under what conditions it will be called, so there are no optimizations there either. That means piecing together the destination in every iteration, rather than just incrementing it like the inline version does. All in all, you get a line plotter that's nearly 4 times as slow *purely* because you've used a function for a single line of code instead of inlining it via a macro or inline function. While anyone could have told you something like that would happen, actually looking at the differences leaves a stronger impression.

There is a lot more that could be learned from this code, but I'll leave it at this for now. The main aim was to show you what assembly (in this case Thumb asm) looks like. In this small piece of code, you can already see many of the elements that go into a full program. Even though the lack of variable identifiers is a bit of a pain, it should be possible to follow along with the code, just as you would with a C program. See, it's not all that bad now, is it?

ON WORKING BY EXAMPLE

> Looking at other people's code (in this case GCC's assembly) is a nice way of learning how to make things work, it is *not* a substitute for the manual. It may show you how to get something done, there is always the danger of getting them done wrongly or inefficiently. Programming is hardly ever trivial and you are likely to miss important details: the compiler may not be not optimising correctly, you could misinterpret the data, etc. This kind of learning often leads to cargo-cult programming, which often does more harm than good. If you want examples of these problems, look at nearly all other GBA tutorials and a lot of the available GBA demo code out there.

Assembling assembly

The assembler of the GNU toolchains is known as the GNU assembler or GAS, and the tool's name is `arm-none-eabi-as`. You can call this directly, or you can use the familiar `arm-none-eabi-gcc` to act as a gateway. The latter is probably a better choice, as it'll allow the use of the C preprocessor with '`-x assembler-with-cpp`'. That's right, you can then use macros, C-style comments *and* #include if you wish. A rule for assembling things might look something like this.

```
AS      := arm-none-eabi-gcc
ASFLAGS := -x assembler-with-cpp

# Rule for assembling .s -> .o files
$(SOBJ) : %.o : %.s
    $(AS) $(ASFLAGS) -c $< -o $@
```

This rule should work on the generated output of `gcc -S`. Note that it will probably not assemble under other assemblers (ARM SDT, Goldroad) because they have different standards for directives and comments and the like. I'll cover some important directives of GAS later, after we've seen what ARM assembly itself is like.

The ARM instruction set

The ARM core is a *RISC* (Reduced Instruction Set Computer) processor. Whereas CISC (Complex Instruction Set Computer) chips have a rich instruction set capable of doing complex things with a single instruction, RISC architectures try to go for more generalized instructions and efficiency. They have a comparatively large number of general-purpose registers and data instructions usually use three registers: one destination and two operands. The length of each instruction is the same, easing the decoding process, and RISC processors strive for 1-cycle instructions.

There are actually two instruction sets that the ARM core can use: ARM code with 32bit instructions, and a subset of this called Thumb, which has 16bit long instructions. Naturally, the ARM set is more powerful, but because the most used instructions can be found in both, an algorithm coded in Thumb uses less memory and may actually be faster if the memory buses are 16bit; which is true for GBA ROM and EWRAM and the reason why most of the code is compiled to Thumb. The focus in this section will be the ARM set, to learn Thumb is basically a matter of knowing which things you cannot do anymore.

The GBA processor's full name is ARM7TDMI, meaning it's an ARM 7 code (aka ARM v4), which can read **T**HUMB code, has a **D**ebug mode and a fast **M**ultiplier. This chapter has this processor in mind, but most of it should be applicable to other chips in the ARM family as well.

Basic features

ARM registers

ARM processors have 16 32bit registers named r0-r15, of which the last three are usually reserved for special purposes: **r13** is used as the stack pointer (SP); **r14** is the *link register* (LR), indicating where to return to from a function, and **r15** is the program counter (PC).. The rest are free, but there are a few conventions. The first four, **r0-r3**, are *argument* and/or *scratch registers*; function parameters go here (or

onto the stack), and these registers are expected to be clobbered by the called function. **r12** falls into this category too. The rest, **r4-r11**, are also known as *variable registers*.

std	gcc	arm	description
r0-r3	r0-r3	a1-a4	argument / scratch
r4-r7	r4-r7	v1-v4	variable
r8	r8	v5	variable
r9	r9	v6/SB	platform specific
r10	sl	v7	variable
r11	fp	v8	variable / frame pointer
r12	ip	IP	Intra-Procedure-call scratch
r13	sp	SP	Stack Pointer
r14	lr	LR	Link Register
r15	pc	PC	Program Counter

Table 23.1. Standard and alternative register names.

ARM instructions

Nearly all of the possible instructions fall into the following three classes: *data operations*, such as arithmetic and bit ops; *memory operations*, load and store in many guises and *branches* for jumping around code for loops, ifs and function calls. The speed of instructions almost follows this scheme as well. Data instructions usually happen in a cycle; memory ops uses two or three and branches uses 3 or 4. The whole timing thing is actually a *lot* more complicated than this, but it's a useful rule of thumb.

All instructions are conditional

On most processors, you can only use branches conditionally, but on ARM systems you can attach the conditionals to *all* instructions. This can be very handy for small if/else blocks, or compound conditions, which would otherwise require the use of the more time-consuming branches. The code below contains asm versions of the familiar `max(a, b)` macro. The first one is the traditional version, which requires two labels, two jumps (although only one of those is executed) and the two instructions that actually do the work. The second version just uses two `mov` 's, but only one of them will actually be executed thanks to the conditionals. As a result, it is shorter, faster and more readable.

These kinds of conditionals shouldn't be used blindly, though. Even though you won't execute the instruction if the conditional fails, you still need to read it from memory, which costs one cycle. As a rough guideline, after about 3 skipped instructions, the branch would actually be faster.

```
@ // r2= max(r0, r1):
@ r2= r0>=r1 ? r0 : r1;

@ Traditional code
    cmp     r0, r1
    blt .Lbmax      @ r1>r0: jump to r1=higher code
    mov     r2, r0  @ r0 is higher
    b   .Lrest      @ skip r1=higher code
.Lbmax:
    mov     r2, r1  @ r1 is higher
.Lrest:
    ...             @ rest of code

@ With conditionals; much cleaner
    cmp     r0, r1
    movge   r2, r0  @ r0 is higher
    movlt   r2, r1  @ r1 is higher
    ...             @ rest of code
```

Another optional item is whether or not the status flags are set. Test instructions like `cmp` always set them, but most of the other require an ' `-s` ' affix. For example, `sub` would not set the flags, but `sub s` would. Because this kinda clashes with the plural

's', I'm using adding an apostrophe for the plural form, so `subs` means `sub` with status flags, but `sub` 's means multiple `sub` instructions.

> **ALL INSTRUCTIONS ARE CONDITIONAL**
>
> Each instruction of the ARM set can be run conditionally, allowing shorter, cleaner and faster code.

The barrel shifter

A barrel shifter is a circuit dedicated to performing bit-shifts. Well, shifts and rotations, but I'll use the word 'shift' for both here. The barrel shifter is part of the ARM core itself and comes before any arithmetic so that you it can handle shifted numbers very fast. The real value of the barrel shifter comes from the fact that almost all instructions can apply a shift to one of their operands at no extra cost.

There are four barrel-shift operations: left shift (`lsl`), logical right-shift (`lsr`), arithmetic right-shift (`asr`) and rotate-right (`ror`). The difference between arithmetic and logical shift right is one of signed/unsigned numbers; see the bit ops section for details. These operations are attached to the last register in an operation, followed by an immediate value or a register. For example, instead of simply `Rm` you can have '`Rm, lsl #2`' means `Rm<<2` and '`Rm, lsr Rs`' for `Rm>>Rs` . Because shifted registers can apply to almost all instructions and I don't want to write it in full all the time, I will designate the shifted register as *Op2*.

Now this may seem like esoteric functionality, but it's actually very useful and more common than you think. One application is multiplications by $2^n \pm 1$, without resorting to relatively slow multiplication instructions. For example, $x*9$ is the same as $x*(1+8) = x + x*8 = x+(x<<3)$. This can be done in a single `add` . Another use is in loading values from arrays, for which indices would have to be multiplied by the size of the elements to get the right addresses.

```
@ Multiplication by shifted add/sub

add r0, r1, r1, lsl #3      @ r0= r1+(r1<<3) = r1*9
rsb r0, r1, r1, lsl #4      @ r0= (r1<<2)-r1 = r1*15

@ word-array lookup: r1= address (see next section)
ldr r0, [r1, r2, lsl #2]    @ u32 *r1; r0= r1[r2]
```

Other uses are certainly possible as well. Like the conditional, you might not really need use of shifted `add` 's and such, but they allow for some *wonderfully* optimized code in the hands of the clever programmer. These are things that make assembly fun.

SHIFTS ARE FREE!

Or at least very nearly so. Shifts-by-value can be performed at no extra cost, and shifts-by-register cost only one cycle. Actually, bit-rotates behave like this as well, but they're rather rare and since I don't know of the correct term that encompasses both, I'll use the word "shift" for both.

Restricted use of immediate values

And now for one of the points that makes ARM assembly less fun. As I said, each instruction is 32bits long. Now, there are 16 condition codes, which take up 4 bits of the instruction. Then there's 2x four for the destination and first operand registers, one for the set-status flag, and then an assorted number of bits for other matters like the actual opcodes. The bottom line is that you only have 12 bits left for any immediate value you might like to use.

That's right 12. **A whole 12 bits**. You may have already figured out that, since this will only allow for 4096 distinct values, this presents a bit of a problem. And you'd be right. This is one of the major points of bad news for RISC processors: after assigning bits to instruction-type, registers and other fields, there's very little room for actual numbers

left. So how is one to load a number like `0601:0000` (object VRAM) then? Well … you *can't*! At least, not in one go.

So, there is only a limited amount of numbers that can be used directly; the rest must be pieced together from multiple smaller numbers. Instead of just taking the 12 bits for a single integer, what the designers have done is split it into an 8bit number (*n*) and a 4bit rotation field (*r*). The barrel shifter will take care of the rest. The full immediate value *v* is given by:

$$(23.1) \qquad v = n \ ror \ 2 * r$$

This means that you can create values like 255 (*n*=255, *r*=0) and 0x06000000 (*n*=6, *r*=4 (remember, rotate-*right*)). However, 511 and 0x06010000 are still invalid because the bit-patterns can't fit into one byte. For these invalid numbers you have two options: construct them in multiple instructions, or load them from memory. Both of these can become quite expensive so if it is possible to avoid them, do so.

The faster method of forming bigger numbers is a matter of debate. There are many factors involved: the number in mind, memory section, instruction set and amount of space left, all interacting in nasty ways. It's probably best not to worry about it too much, but as a guideline, I'd say if you can do it in two data instructions do so; if not, use a load. The easiest way of creating big numbers is with a special form of the `ldr` instruction: '`ldr Rd,=num`' (note: no '#'!). The assembler will turn this into a mov if the number allows it, or an `ldr` if it doesn't. The space that the number needs will be created automatically as well.

```
@ form 511(0x101) with mov's
mov     r0, #256    @ 256= 1 ror 24, so still valid
add     r0, #255    @ 256+255 = 511

@ Load 511 from memory with special ldr
@ NOTE: no '#' !
ldr     r0,=511
```

That there is only room for an 8bit number + 4bit rotate for immediate operands is something you'll just have to learn to live with. If the assembler occasionally

complains about invalid constants, you now know what it means and how you can correct for it. Oh, and if you thought this was bad, think of how it would work for Thumb code, which only has 16 bits to work with.

THE ONLY VALID IMMEDIATE VALUES ARE ROTATED BYTES

When instructions allow immediate values, the only permissible values are those that can be reduced to a byte rotated by an even number. 0xFF and 0x100 are allowed, but 0x101 is not. This has consequences for data operations, but also for memory addressing since it will not be possible to load a full 32bit address in one go. You can either construct the larger value out of smaller parts, or use a load-assignment construct: '`ldr Rd ,= num`' which the assembler will convert into a `mov` if possible, or a PC-relative load if not.

REMEMBER THE PREVIOUS NOTE

Is this worth a separate note? Maybe not, but the previous note is important enough to remember. It is not exactly intuitive that code should behave that way and if you found yourself staring at the enigmatic invalid constant error message, you'd probably be lost without this bit of info.

Data instructions

The data operations carry out the calculations of a program, which includes both arithmetic and logical operations. You can find a summary of the data instructions in table 23.2. While this lists them in four groups, the only real division is between the multiplies and the rest. As you can see, there is **no** division instruction. While this can be considered highly annoying, as it turns out the need for division is actually quite small – small enough to cut it out of the instruction set, anyway.

Unlike some processors, ARM can only perform data processing on registers, not on memory variables directly. Most data instructions use one destination register and two operands. The first operand is always a register, the second can be four things: an immediate value or register (#*n* / `Rm`) or a register shifted by an immediate value or register ('`Rm, lsl # n`', '`Rm, lsl Rs`', and similar for `lsr`, `asr` and `ror`). Because this arrangement is quite common, it is often referred to as simply *Op2*, even if it's not actually a second operand.

Like all instructions, data instructions can be executed conditionally by adding the appropriate affix. They can also alter the status flags by appending the - `s` prefix. When using both, the conditional affix always comes first.

opcode	operands	function
Arithmetic		
adc	Rd, Rn, Op2	Rd = Rn + Op2 + C
add	Rd, Rn, Op2	Rd = Rn + Op2
rsb	Rd, Rn, Op2	Rd = Op2 - Rn
rsc	Rd, Rn, Op2	Rd = Op2 - Rn - !C
sbc	Rd, Rn, Op2	Rd = Rn - Op2 -!C
sub	Rd, Rn,	Rd = Rn -

opcode	operands	function
Status ops		
cmp	Rn, Op2	Rn - Op2
cmn	Rn, Op2	Rn + Op2
teq	Rn, Op2	Rn & Op2
tst	Rn, Op2	Rn ^ Op2
Multiplies		
mla	Rd, Rm, Rs, Rn	Rd = Rm * Rs + Rn
mul	Rd, Rm, Rs	Rd = Rm * Rs
smlal	RdLo, RdHi, Rm, Rs	RdHiLo += Rm * Rs
smull	RdLo, RdHi, Rm, Rs	RdHiLo = Rm * Rs
umlal	RdLo, RdHi,	RdHiLo += Rm *

	Op2	Op2
Logical ops		
and	Rd, Rn, Op2	Rd = Rn & Op2
bic	Rd, Rn, Op2	Rd = Rn &~ Op2
eor	Rd, Rn, Op2	Rd = Rn ^ Op2
mov	Rd, Op2	Rd = Op2
mvn	Rd, Op2	Rd = ~Op2
orr	Rd, Rn, Op2	Rd = Rn | Op2

	Rm, Rs	Rs
umull	RdLo, RdHi, Rm, Rs	RdHiLo = Rm * Rs

23.2: Data processing instructions. Basic format `op{cond}{s} Rd, Rn, Op2`, `cond` and `s` are the optional condition and status codes, and *Op2* a shifted register.

The first group, arithmetic, only contains variants of addition and subtraction. `add` and `sub` are their base forms. `rsb` is a special thing that reverses the operand order; the difference with the regular `sub` is that *Op2* is now the *minuend* (the thing subtracted from). Only *Op2* is allowed to have immediate values and shifted registers, which allows you to negate values (0–*x*) and fast-multiply by 2^n-1.

The variants ending in ' `c` ' are additions and subtractions with carry, which allows for arithmetic for values larger than the register size. For example, consider you have 8bit registers and want to add 0x00FF and 0x0104. Because the latter doesn't fit into one register, you have to split it and then add twice, starting with with the least significant byte. This gives 0xFF+0x04=0x103, represented by 0x02 in the destination register and

a set carry flag. For the second part you have to add 0x00 and 0x01 from the operands, *and* the carry from the lower byte, giving 0x00+0x01+1 = 0x02. Now string the separate parts together to give 0x0203.

Because ARM registers are 32bit wide you probably won't be using the those instructions much, but you never know.

The second group are the bit operations, most of which you should be familiar with already. The all have exact matches in C operators, with the exception of bit-clear. However, the value of such an instruction should be obvious. You will notice a distinct absence of shift instructions here, for the simple reason that they're not really necessary: thanks to the barrel shifter, the `mov` instruction can be used for shifts and rotates. 'r1 = r0<<4' could be written as '`mov r1, r0, lsl #4`'.

I have mentioned this a couple of times now, but as we're dealing with another language now it bares repeating: there is a difference between right-shifting a signed and unsigned numbers. Right-shifts remove bits from the top; unsigned numbers should be zero-extended (filled with 0), but signed numbers should be sign-extended (filled with the original MSB). This is the difference between a *logical shift right* and an *arithmetic shift right*. This doesn't apply to left-shifts, because that fills zeroes either way.

The third group isn't much of a group, really. The status flag operations set the status bits according to the results of their functionality. Now, you can do this with the regular instructions as well; for example, a compare (`cmp`) is basically a subtraction that also sets the status flags, i.e., a `subs` . The only real difference is that this time there's no register to hold the result of the operation.

Lastly, the multiplication formats. At the table indicates, you cannot use immediate values; if you want to multiply with a constant you *must* load it into a register first. Second, no *Op2* means no shifted registers. There is also a third rule that Rd and Rm can't use the same register, because of how the multiplication algorithm is implemented. That said, there don't *seem* to be any adverse effects using Rd=Rm.

The instruction `mla` stands for 'multiply with accumulate', which can be handy for dot-products and the like. The `mull` and `mlal` instructions are for 64bit arithmetic, useful for when you expect the result not to fit into 32bit registers.

```
@ Possible variations of data instructions
add     r0, r1, #1          @ r0 = r2 + 1
add     r0, r1, r2          @ r0 = r1 + r2
add     r0, r1, r2, lsl #4  @ r0 = r1 + r2<<4
add     r0, r1, r2, lsl r3  @ r0 = r1 + r2<<r3

@ op= variants
add     r0, r0, #2          @ r0 += 2;
add     r0, #2              @ r0 += 2; alternative  (but not on all
assemblers)

@ Multiplication via shifted add/sub
add     r0, r1, r1, lsl #4  @ r0 = r1 + 16*r1 = 17*r1
rsb     r0, r1, r1, lsl #4  @ r0 = 16*r1 - r1 = 15*r1
rsb     r0, r1, #0          @ r0 =     0 - r1 = -r1

@ Difference between asr and lsr
mvn     r1, #0              @ r1 = ~0 = 0xFFFFFFFF = -1
mov     r0, r1, asr #16     @ r0 = -1>>16 = -1
mov     r0, r1, lsr #16     @ r0 = 0xFFFFFFFF>>16 = 0xFFFF = 65535


@ Signed division using shifts. r1= r0/16
@ if(r0<0)
@      r0 += 0x0F;
@   r1= r0>>4;
mov     r1, r0, asr #31     @ r0= (r0>=0 ? 0 : -1);
add     r0, r0, r1, lsr #28 @ += 0 or += (0xFFFFFFFF>>28 = 0xF)
mov     r1, r0, asr #4      @ r1 = r0>>4;
```

Memory instructions: load and store

Because ARM processors can only perform data processing on registers, interactions with memory only come in two flavors: loading values from memory into registers and storing values into memory from registers.

The basic instructions for that are `ldr` (LoaD Register) and `str` (STore Register), which load and store words. Again, the most general form uses two registers and an *Op2*:

> *op*{cond}{type} Rd, [Rn, *Op2*]

Here *op* is either `ldr` or `str`. Because they're so similar in appearance, I will just use `ldr` for the remainder of the discussion on syntax, except when things are different. The condition flag again goes directly behind the base opcode. The *type* refers to the datatype to load (or store), which can be words, halfwords or bytes. The word forms do not use any extension, halfwords use `-h` or `-sh`, and bytes use `-b` and `-sb`. The extra `s` is to indicate a signed byte or halfword. Because the registers are 32bit, the top bits need to be sign-extended or zero-extended, depending on the desired datatype.

The first register here, `Rd` can be either the destination or source register. The thing between brackets always denotes the memory address; `ldr` means load *from* memory, in which case `Rd` is the destination, and `str` means store *to* memory, so `Rd` would be the source there. `Rn` is known as the **base register**, for reasons that we will go into later, and *Op2* often serves as an offset. The combination works very much like array indexing and pointer arithmetic.

> **MEMORY OPS VS C POINTERS/ARRAYS**
>
> To make the comparison to C a little easier, I will sometimes indicate what happens using pointers, but in order to do that I will have to indicate the type of the pointer somehow. I could use some horrid casting notation, but it would be easiest to use a form of arrays for this, and use the register-name + an affix to show the data type. I'll use '_w' for words, '_h' for halfwords, and '_b' for bytes, and '_sw', etc. for their signed versions. For example, `r0_sh` would indicate that `r0` is a signed halfword pointer. This is just a useful bit of shorthand, not actually part of assembly itself.
>
> ```
> @ Basic load/store examples. Assume r1 contains a word-aligned
> address
> ldr     r0, [r1]    @ r0= *(u32*)r1; //or r0= r1_w[0];
> str     r0, [r1]    @ *(u32*)r1= r0; //or r1_w[1]= r0;
> ```

Addressing modes

There are several ways of interacting, known as **addressing modes**. The simplest form is *direct addressing*, where you indicate the address directly via an immediate value. However, that mode is unavailable to ARM systems because full addresses don't fit in the instruction. What we do have is several indirect addressing forms.

The first available form is **register indirect addressing**, which gets the address from a register, like '`ldr Rd, [Rn]`'. An extension of this is **pre-indexed addressing**, which adds an offset to the base register before the load. The base form of this is '`ldr Rd, [Rn, Op2]`'. This is very much like array accesses. For example '`ldr r1, [r0, r2, lsl #2]`' corresponds to `r0_w[r2]` : an word-array load using `r2` as the index.

Another special form of this is **PC-relative addressing**, which makes up for not having direct addressing. Suppose you have a variable in memory somewhere. While you may not be able to use that variable's address directly, what you can do is store the address close to where you are in the code. *That* address is at a certain allowed offset from the program counter register (PC), so you could load the variable's address from there and then read the variable's contents. You can also use this to load constants that are too large to fit into a shifted byte.

While it is possible to calculate the required offset manually, you'll be glad to know you can let the assembler do this for you. There are two ways of doing this. The first is to create a **data-pool** where you intend to put the addresses and constants, and label it. You can then get its address via '`ldr Rd, LabelName`' (note the absence of brackets here). The assembler will turn this into pc-relative loads. The second method is to let the assembler do all the work by using '`ldr Rd,= foo`', where *foo* is the variable name or an immediate value. The assembler will then allocate space for *foo* itself. Please remember that using *=varname* does **not** load the variable itself, only its address.

And then there are the so-called **write-back** modes. In the pre-index mode, the final address was made up of `Rn` *+Op2*, but that had no effect on `Rn` . With write-back, the

final address is put in `Rn` . This can be useful for walking through arrays because you won't need an actual index.

There are two forms of write-back, pre-indexing and post-indexing. Pre-indexing write-back works much like the normal write-back and is indicated by an exclamation mark after the brackets: ' `ldr Rd, [Rn, Op2 ]!` '. Post-indexing doesn't add *Op2* to the address (and `Rn`) until *after* the memory access; its format is ' `ldr Rd, [Rn], Op2` '.

```
@ Examples of addressing modes
@ NOTE: *(u32*)(address+ofs) is the same as ((u32*)address)[ofs/4]
@   That's just how array/pointer offsets work
    mov     r1, #4
    mov     r2, #1
    adr     r0, fooData      @ u32 *src= fooData;
@ PC-relative and indirect addressing
    ldr     r3, fooData              @ r3= fooData[0];    // PC-relative
    ldr     r3, [r0]                 @ r3= src[0];        // Indirect
addressing
    ldr     r3, fooData+4            @ r3= fooData[1];    // PC-relative
    ldr     r3, [r0, r1]             @ r3= src[1];        // Pre-indexing
    ldr     r3, [r0, r2, lsl #2]     @ r3= src[1]         // Pre-index,
via r2
@ Pre- and post-indexing write-back
    ldr     r3, [r0, #4]!            @ src++;    r3= *src;
    ldr     r3, [r0], #4             @ r3= *src; src++;
@ u32 fooData[3]= { 0xF000, 0xF001, 0xF002 };
fooData:
    .word   0x0000F000
    .word   0x0000F001
    .word   0x0000F002
```

PC-RELATIVE SPECIALS

PC-relative instructions are common, and have a special shorthand that is easier and shorter to use than creating a pool of data to load from. The format for this is ' `ldr Rd,= foo` ', where *foo* is a label or an immediate value. In both cases, a pool is created to hold these numbers automatically. Note that the value for the label is its *address*, not the address' contents.

If the label is near enough you can also use `adr` , which is assembled to a PC-add instruction. This will not create a pool-entry.

```
@ Normal pc-relative method:
@   create a nearby pool and load from it
    ldr     r0, .Lpool      @ Load a value
    ldr     r0, .Lpool+4    @ Load far_var's address
    ldr     r0, [r0]        @ Load far_var's contents
.Lpool:
    .word   0x06010000
    .word   far_var


@ Shorthand: use ldr= and GCC will manage the pool for you
    ldr     r0,=0x06010000  @ Load a value
    ldr     r0,=far_var     @ Load far_var's address
    ldr     r0, [r0]        @ Load far_var's contents
```

Note that I'm not actually creating `far_var` here; just storage room for its address. Creation of variables is covered later.

Data types

It is also possible to load/store bytes and halfwords. The opcodes for loads are `ldrb` and `ldrh` for unsigned, and `ldrsb` and `ldrsh` for signed bytes and halfwords, respectively. The 'r' in the signed versions is actually optional, so you'll also see `ldsb` and `ldsh` now and then. As stores can cast away the more significant bytes anyway, `strb` and `strh` will work for both signed and unsigned stores..

All the things you can do with `ldr/str` , you can do with the byte and halfword versions as well: PC-relative, indirect, pre/post-indexing it's all there … with one exception. The signed-byte load (`ldsb`) and *all* of the halfword loads and stores cannot do shifted register-loads. Only `ldrb` has the complete functionality of the word instructions. The consequence is that signed-byte or halfword arrays may require extra instructions to keep the offset and index in check.

Oh, one more thing: alignment. In C, you could rely on the compiler to align variables to their preferred boundaries. Now that you're taking over from the compiler, it stands to reason that you're also in charge of alignment. This can be done with the '.align *n*' directive, with aligns the next piece of code or data to a 2^n boundary. Actually, you're supposed to properly align code as well, something I'm taking for granted in these snippets because it makes things easier.

```
    mov     r2, #1
@ Byte loads
    adr     r0, bytes
    ldrb    r3, bytes        @ r3= bytes[0];     // r3= 0x000000FF= 255
    ldrsb   r3, bytes        @ r3= (s8)bytes[0]; // r3= 0xFFFFFFFF= -1
    ldrb    r3, [r0], r2     @ r3= *r0_b++;      // r3= 255, r0++;
@ Halfword loads
    adr     r0, hwords
    ldrh    r3, hwords+2     @ r3= words[1];     // r3= 0x0000FFFF= 65535
    ldrsh   r3, [r0, #2]     @ r3= (s16)r0_h[1]; // r3= 0xFFFFFFFF= -1
    ldrh    r3, [r0, r2, lsl #1]    @ r3= r0_h[1]? No! Illegal
instruction :(

@ Byte array: u8 bytes[3]= { 0xFF, 1, 2 };
bytes:
    .byte   0xFF, 1, 2
@ Halfword array u16 hwords[3]= { 0xF001, 0xFFFF, 0xF112 };
    .align  1    @ align to even bytes REQUIRED!!!
hwords:
    .hword  0xF110, 0xFFFF, 0xF112
```

Block transfers

Block transfers allow you to load or store multiple successive words into registers in one instruction. This is useful because it saves on instructions, but more importantly, it saves time because individual memory instructions are quite costly and with block transfers you only have to pay the overhead once. The basic instructions for block transfers are `ldm` (LoaD Multiple) and `stm` (STore Multiple), and the operands are a base register (with an optional exclamation mark for `Rd` write-back) and a list of registers between braces.

op{cond}{mode} Rd{!}, {*Rlist*}

This register list can be comma separated, or hyphenated to indicate a range. For example, `{r4-r7, lr}` means registers r4, r5, r6, r7 and r14. The order in which the registers are actually loaded or stored are **not** based on the order in which they are specified in this list! Rather, the list indicates the number of words used (in this case 5), and the order of addresses follows the index of the registers: the lowest register uses the lowest address in the block, etc.

The block-transfer opcodes can take a number of affixes that determine how the block extends from the base register `Rd`. The four possibilities are: `-IA` / `-IB` (Increment After/Before) and `-DA` / `-DB` (Decrement After/Before). The differences are essentially those between pre/post-indexing and incrementing or decrementing from the base address. It should be noted that these increments/decrements happen regardless of whether the base register carries an exclamation mark or not: that thing only indicates that the base register *itself* is updated afterwards.

```
    adr      r0, words+16     @ u32 *src= &words[4];
                              @             r4, r5, r6, r7
    ldmia    r0, {r4-r7}      @ *src++    :  0,  1,  2,  3
    ldmib    r0, {r4-r7}      @ *++src    :  1,  2,  3,  4
    ldmda    r0, {r4-r7}      @ *src--    : -3, -2, -1,  0
    ldmdb    r0, {r4-r7}      @ *--src    : -4, -3, -2, -1
    .align   2
 words:
    .word    -4, -3, -2, -1
    .word     0,  1,  2,  3, 4
```

The block transfers are also used for stack-work. There are four types of stacks, depending on whether the address that `sp` points to already has a stacked value or not (Full or Empty), and whether the stack grows down or up in memory (Descending/Ascending). These have special affixes (`-FD`, `-FA`, `-ED` and `-EA`) because using the standard affixes would be awkward. For example, the GBA uses an FD-type stack, which means that pushing is done with `stmdb` because decrementing after the store would overwrite an already stacked value (full stack), but popping requires `ldmia` for similar reasons. A `stmfd/ldmfd` pair is much easier to deal with. Or you could just use `push` and `pop`, which expand to '`stmfd sp!,`' and '`ldmfd sp!,`', respectively.

Block op	Standard	Stack alt
Increment After	ldmia / stmia	ldmfd / stmea
Increment Before	ldmib / stmib	ldmed / stmfa
Decrement After	ldmda / stmda	ldmfa / stmed
Decrement Before	ldmdb / stmdb	ldmea / stmfd

Table 23.3: Block transfer instructions.

PUSH AND POP ARE NOT UNIVERSAL ARM INSTRUCTIONS

They seem to work for devkitARM r15 and up (haven't checked older versions), but DevKitAdv for example doesn't accept them. Just try and see what happens.

Conditionals and branches

Higher languages typically have numerous methods for implementing choices, loops and function calls. The all come down to the same thing though: the ability to move the program counter and thereby diverting the flow of the program. This procedure is known as branching.

There are three branching instructions in ARM: the simple branch `b` for ifs and loops, the branch with link `bl` used for function calls, and branch with exchange `bx` used for switching between ARM and Thumb code, returning from functions and out-of-section jumps. `b` and `bl` use a label as their argument, but `bx` uses a register with the address to branch to. Technically there are more ways of branching (PC is just another register, after all) but these three are the main ones.

Status flags and condition codes

I've already mentioned part of this in the introduction, so I'll make this brief. The ARM processor has 4 status flags, (**Z**)ero, (**N**)egative, (**C**)arry and signed o(**V**)erflow, which

can be found in the program status register. There are actually two of these: one for the *current* status (CPSR) and a *saved* status register (SPSR), which is used in interrupt handlers. You won't have to deal with either of these, though, as reacting to status registers usually goes through the conditional codes (table 23.4). But first, a few words about the flags themselves:

- **Zero** (Z). If the result of operation was 0.
- **Negative** (N). Result was negative (i.e. most significant bit set).
- **Carry bit set** (C). If the 'mostest' significant bit is set (like bit 32 for 32bit operations).
- **Arithmetic overflow** (V). Like adding two positive numbers and getting a negative number because the result got too big for the registers.

Each of the data instructions can set the status flags by appending `-s` to the instruction, except for `cmp`, `cmn`, `tst` and `teq`, which always set the flags.

Table 23.4 lists 16 affixes that can be added to the basic branch instruction. For example, `bne Label` would jump to `Label` if the status is non-zero, and continue with the next instruction if it isn't.

Affix	Flags	Description
eq	Z=1	Zero (EQual to 0)
ne	Z=0	Not zero (Not Equal to 0)
cs / hs	C=1	Carry Set / unsigned Higher or Same
cc / lo	C=0	Carry Clear / unsigned LOwer
mi	N=1	Negative (MInus)
pl	N=0	Positive or zero (PLus)
vs	V=1	Sign overflow (oVerflow Set)
vc	V=0	No sign overflow (oVerflow Clear)
hi	C=1 & Z=0	Unsigned HIgher
ls	C=0	Z=1	Unsigned Lower or Same
ge	N=V	Signed Greater or Equal

lt	N != V	Signed Less Than
gt	Z=0 & N=V	Signed Greater Than
le	Z=1 | N != V	Signed Less or Equal
al	-	ALways (default)
nv	-	NeVer

Table 23.4: conditional affixes.

To use these condition codes properly, you need to know what each stands for, but also how the data operations set the flags. The effect on the status flags depends on the instruction itself, and not all flags are affected by all instructions. For example, overflow only has meaning for arithmetic, not bit operations.

In the case of Z and N, the case is pretty easy. The operation gives a certain 32bit value as its result; if it's 0, then the Zero flag is set. Because of two's complement, the Negative flag is the same as bit 31. The reason `-eq` and `ne` are linked to the zero flags is because a comparison (`cmp`) is basically a subtraction: it looks at the difference between the two numbers and when that's zero, then the numbers are equal.

For the carry bit it can get a little harder. The best way to see it is as an extra most significant bit. You can see how this work in the example of table 23.5. Here we add two unsigned numbers, 2^{31} = 0x80000000. When adding them, the result would overflow 32bits, giving 0 and not 2^{32}. However, that overflowed bit will go into the carry. With the `adc` instruction you could then go on to build adders for numbers larger thatn the registers.

2^{31}	8000 0000	
2^{31}	8000 0000	+
2^{32}	**1 0000** 0000	

Table 23.5: carry bit in
(unsigned) addition.

Bit-operations like `orr` or `and` don't affect it because they operate purely on the lower 32bits. Shifts, however do.

You may find it odd that `-cc` is the code for unsigned higher than. As mentioned, a comparison is essentially a subtraction, but when you subtract, say 7−1, there doesn't really seem to be a carry here. The key here is that subtractions are infact forms of additions: 7−1 is actually 7+0xFFFFFFFF, which would cause an overflow into the carry bit. You can also thing of subtractions as starting out with the carry bit set.

The overflow flag indicates *signed* overflow (the carry bit would be unsigned overflow). Note, this is *not* merely a sign change, but a sign change the wrong way. For example, an addition of two positive numbers *should* always be positive, but if the numbers are big enough (say, 2^{30}, see table 23.6) then the results of the lower 30 bits may overflow into bit 31, therefore changing the sign and you'll have an incorrect addition. For subtraction, there can be a similar problem. Short of doing the full operation and checking whether the signs are correct, there isn't a simple way of figuring out what counts as overflow, but fortunately you don't have to. Usually overflow is only important for signed comparisons, and the condition mnemonics themselves should provide you with enough information to pick the right one.

$+2^{30}$	`4000 0000`	
$+2^{30}$	`4000 0000`	+
-2^{31}	`8000 0000`	

Table 23.6: sign overflow.

With these points in mind, the conditional codes shouldn't be too hard to understand. The descriptions tell you what code you should use when. Also, don't forget that any

instruction can be conditionally executed, not just a branch.

The basic branch

Let's start with the most basic of branches, `b` . This is the most used branch, used to implement normal conditional code and loops of all kinds. It is most often used in conjunction with one of the 16 conditional codes of table 23.4. Most of the times a branch will look something like this:

```
@ Branch example, pseudo code
    data-ops, Rd, Rn, Op2   @ Data operation to set the flags
    bcnd-code .Llabel       @ Branch upon certain conditions

    @ more code A

.Llabel:                    @ Branch goes here
    @ more code B
```

First, you have a data processing instruction that sets the status flags, usually a `subs` or `cmp` , but it can be any one of them. Then a `b` *cond* diverts the flow to `.Llabel` if the conditions are met. A simple example of this would be a division routine which checks if the denominator is zero first. For example, the `Div()` routine that uses BIOS Call #6 could be safeguarded against division by 0 like this:

```
@ int DivSafe(int num, int den);
@ \param num    Numerator (in r0)
@ \param den    Denominator (in r1)
@ \return       r0= r0/r1, or INT_MAX/INT_MIN if r1 == 0
DivSafe:
    cmp     r1, #0
    beq     .Ldiv_bad   @ Branch on r1 == 0
    swi     0x060000
    bx      lr
.Ldiv_bad:
    mvn     r1, #0x80000000    @ \
    sub     r0, r1, r0, asr #31 @ - r0= r0>=0 ? INT_MAX : INT_MIN;
    bx      lr
```

The numerator and denominator will be in registers r0 and r1, respectively. The `cmp` checks whether the denominator is zero. If it's not, no branch is taken, the swi 6 is executed and the function returns afterwards. If it is zero, the `beq` will take the code to `.Ldiv_bad`. The two instructions there set r0 to either INT_MAX ($2^{31}-1 =$ 0x7FFFFFFF) or INT_MIN ($-2^{31} = $ 0x80000000), depending on whether r0 is positive or negative. If it's a little hard to see that, `mvn` inverts bits, so the first line after `.Ldiv_bad` sets r0 to INT_MAX. The second line we've seen before: '`r0, asr #31`' does a sign-extension in to all other bits, giving 0 or −1 for positive and negative numbers, respectively, giving INT_MAX− −1 = INT_MIN for negative values of r0. Little optimizing tricks like these decide if you're fit to be an assembly programmer; if not you could just as well let the compiler do them, because it does know. (It's where I got the '`asr #31`' thing from in the first place.)

Now in this case I used a branch, but in truth, it wasn't even necessary. The non-branch part consists of one instruction, and the branched part of two, so using conditional instructions throughout would have been both shorter and faster:

```
@ Second version using conditionally executed code
DivSafe:
    cmp     r1, #0
    mvneq   r1, #0x80000000
    subeq   r0, r1, r0, asr #31
    swine   0x060000
    bx      lr
```

If the denominator is not zero, the `mvneq` and `subeq` are essentially skipped. Actually, not so much skipped, but turned into `nop` : non-operations. So is `swine` (i.e., `swi` + `ne` , no piggies here) if it is zero. True, the division line has increased by a cycle, not taking the branch makes the exception line a little faster and the function itself has shrunk from 7 to 5 instructions.

SYMBOL VS INTERNAL LABELS

> In the first `DivSafe` snippet, the internal branch destination used a `.L` prefix, while the function label did not. The `.L` prefix is used by GCC to indicate labels for the sake of labels, as opposed to symbol labels like `DivSafe`. While not required, it's a useful convention.

Major and minor branches

Any sort of branch will create a fork in the road and, depending on the conditions, one road will be taken more often. That would be the *major* branch. The other one would be the *minor* branch, probably some sort of exception. The branch instruction, `b`, represents a deviation from the normal road and is relatively costly, therefore it pays to have to branch to the exceptions. Consider these possibilities:

```c
// Basic if statement in C
if(r0 == 0)
{   /* IF clause */   }
...
```

```
@ === asm-if v1 : 'bus stop' branch ===
    cmp     r0, #0
    beq     .Lif
.Lrest:
    ...
    bx      lr      @ function ends
.Lif
    @ IF clause
    b   .Lrest


@ === asm-if v2 : 'skip' branch ===
    cmp     r0, #0
    bne     .Lrest
    @ IF clause
.Lrest:
    ...
    bx      lr      @ function ends
```

The first version is more like the C version: it splits off for the IF-clause and the returns to the rest of the code. The flow branches twice if the conditions are met, but if they aren't the rest of the code doesn't branch at all. The second version branches if the conditions *aren't* met: it skips the IF-clause. Overall, the assembly code is simpler and shorter, but the fact that the branch-conditions are inverted with respect to the C version could take some getting used to.

So which to use? Well, that depends actually. All things being equal, the second one is better because it's one instruction and label shorter. As far as I know, this is what GCC uses. The problem is that some things may be more equal than others. If the IF-clause is exceptional (i.e., the minor branch), it'd mean that the second version almost always takes the branch, while the first version would hardly ever branch, so on average the latter would be faster.

Which one you chose is entirely up to you as you know your intentions best. I hope. For the remainder of this chapter I'll use the skip-branch because in demonstrations things usually are equal. Of course, if the clause is small enough you can just use conditional instructions and be done with it `:)` .

Common branching constructs

Even though all you have now is `b` , it doesn't mean you can't implement branching construct found in HLLs in assembly. After all, the compiler seems to manage. Here's a couple of them.

if-elseif

The `if-elseif` is an extension of the normal `if-else` , and from it you can extend to longer `if-elseif-else` chains. In this case I want to look at a wrapping algorithm for keeping numbers within certain boundaries: the number *x* should stay within range [*mn*, *mx*), if it exceeds either boundary it should come out the other end. In C it looks like this:

```
// wrap(int x, int mn, int mx), C version:
int res;
if(x >= mx)
    res= mn + x-mx;
else if(x < mn)
    res= mx + x-mn;
else
    res= x;
```

The straightforward compilation would be:

```
@ r0= x ; r1= mn ; r2= mx
    cmp     r0, r2
    blt     .Lx_lt_mx       @ if( x >= mx )
    add     r3, r0, r1      @   r0= mn + x-mx
    sub     r0, r3, r2
    b       .Lend
.Lx_lt_mx:
    cmp     r0, r1          @
    bge     .Lend           @ if( x < mn )
    add     r3, r0, r2      @   r0= mx + x-mn;
    sub     r0, r3, r1
.Lend:
    ...
```

This is what GCC gives, and it's pretty good. The ordering of the clauses remained, which means that the condition for the branches have to be inverted, so '`if(x >= mx) {}`' becomes 'skip if NOT $x >= mx$'. At the end of each clause, you'd need to skip all the others: branch to `.Lend`. The conditional branches mean 'go to the next branch'.

And now an optimized version. First, a `cmp` is equivalent to `sub` except that it doesn't put the result in a register. However, as we need the result later on anyway, we might as well combine the '`cmp`' and '`sub`'. Secondly, the clauses are pretty small, so we can use conditional ops as well. The new version would be:

```
@ Optimized wrapper
    subs    r3, r0, r2      @ r3= x-mx
    addge   r0, r3, r1      @   x= x-mx + mn
    bge     .Lend
    subs    r3, r0, r1      @ r3= x-mn
    addlt   r0, r3, r2      @   r0= x-mn + mx;
 .Lend:
    ...
```

Cleans up nicely, wouldn't you say? Less branches, less code and it matches the C code more closely. We can even get rid of the last branch too because we can execute the `subs` conditionally as well. Because `ge` and `lt` are each others complements there won't be any interference. So the final version is:

```
@ Optimized wrapper, version 2
    subs    r3, r0, r2      @ r3= x-mx
    addge   r0, r3, r1      @   x= x-mx + mn
    sublts  r3, r0, r1      @ r3= x-mn
    addlt   r0, r3, r2      @   r0= x-mn + mx;
    ...
```

Of course, it isn't always possible to optimize to such an extent. However, if the clauses have small bodies conditional instructions may become attractive. Also, converting a compare to some form of data operation that you'd need later anyway is common and recommended.

Compound logical expressions

Higher languages often allow you to string multiple conditions together using logical AND (`&&`) and logical OR (`||`). What the books often won't say is that these are merely shorthand notations of a chain of `if` s. Here's what actually happens.

```
// === if(x && y) { /* clause */ } ===
if(x)
{
    if(y)
    { /* clause */ }
}

// === if(x || y) { /* clause */ } ===
if(x)
{ /* clause */ }
else if(y)
{ /* clause */ }
```

The later terms in the AND are only evaluated if earlier expressions were true. If not, they're simply skipped. Much fun can be had if the second term happens to be a function with side effects. A logical OR is basically an if-else chain with identical clauses; this is just for show of course, in the final version there's one clause which is branched to. In assembly, these would look something like this.

```
@ if(r0 != 0 && r1 != 0) { /* clause */ }
    cmp     r0, #0
    beq     .Lrest
    cmp     r1, #0
    beq     .Lrest
    @ clause
.Lrest:
    ...

@ Alternative
    cmp     r0, #0
    cmpne   r1, #0
    beq     .Lrest
    @ clause
.Lrest:
    ...
```

```
@ if( r0 != 0 || r1 != 0 ){ /* clause */ }
    cmp     r0, #0
    bne     .Ltrue
    cmp     r1, #0
    beq     .Lrest
.Ltrue:
    @ clause
.Lrest:
    ...
```

As always, alternative solutions will present themselves for your specific situation. Also note that you can transform ANDs into ORs using De Morgan's Laws.

Loops

One of the most important reasons to use assembly is speeding up oft-used code, which will probably involve loops because that's where most of the time will be spent. If you can remove one instruction in a non-loop situation, you'll have won one cycle. If you remove one from a loop, you'll have gained one for every iteration of the loop. For example, saving a cycle in a clear-screen function would save 240*160 = 19200 cycles – more, actually, because of memory wait-states. That one cycle can mean the difference between smooth and choppy animation.

In short, optimization is pretty much all about loops, especially inner loops. Interestingly, this is where GCC often misses the mark, because it adds more stuff than necessary. For example, in older versions (DKA and DKP r12, something like that) it often kept constructed memory addresses (VRAM, etc) inside the loop. Unfortunately, DKP r19 also has major issues with struct copies and `ldm/stm` pairs, which are now only give a small benefit over other methods. Now, before you blame GCC for slow loops, it's also often the C programmer that forces GCC to produce slow code. In the introduction, you could see the major difference that inlining makes, and in the profiling demo I showed how much difference using the wrong datatype makes.

Anyway, loops in assembly. Making a loop is the easiest thing in the world: just branch to a previous label. The differences between `for`, `do-while` and `while` loops are a matter of where you increment and test. In C, you usually use a for-loop with an

incrementing index. In assembly, it's customary to use a while-loop with a decrementing index. Here are two examples of a word-copy loop that should show you why.

```
@ Asm equivalents of copying 16 words.
@ u32 *dst=..., *src= ..., ii     // r0, r1, r2

@ --- Incrementing for-loop ---
@ for(ii=0; ii<16; ii++)
@     dst[ii]= src[ii];
    mov     r2, #0
.LabelF:
    ldr     r3, [r1, r2, lsl #2]
    str     r3, [r0, r2, lsl #2]
    add     r2, r2, #1
    cmp     r2, r2, #16
    blt .LabelF

@ --- Decrementing while-loop ---
@ ii= 16;
@ while(ii--)
@     *dst++ = *src++;
    mov     r2, #16
.LabelW:
    ldr     r3, [r1], #4
    str     r3, [r0], #4
    subs    r2, r2, #1
    bne .LabelW
```

In an incrementing for-loop you need to increment and then compare against the limit. In the decrementing while loop you subtract and test for zero. Because the zero-test is already part of every instruction, you don't need to compare separately. True, it's not much faster, maybe 10% or so, but many 10 percents here and there do add up. There are actually many versions of this kind of loop, here's another one using block-transfers. The benefit of those is that they also work in Thumb:

```
@ Yet another version, using ldm/stm

    add     r2, r0, #16
.LabelW:
    ldmia   r1!, {r3}
    stmia   r0!, {r3}
    cmp     r2, r0
    bne  .LabelW
```

This is one of those occasions where knowing assembly can help you write efficient C. Using a decrementing counter and pointer arithmetic will usually be a little faster, but usually GCC willl do this for you anyway. Another point is using the right datatype. And with 'right' I mean `int`, of course. Non-ints require implicit casts (`lsl` / `lsr` pairs) after every arithmetic operation. That's two instructions after **every** plus, minus or whatever. While GCC has become quite proficient in converting non-ints into ints where possible, this has not always been the case, and it may not always be possible. I've seen the loops above cost between 600% more because the index and pointer were `u16`, I shit you not.

When dealing with loops, be extremely careful with how you start and stop the loop. It's very easy to come up with a loop that runs once too often or too little. I'm pretty sure these two versions are correct. The way I usually check it is to see how it runs when the count should be 1 or 2. If that works out, larger numbers will too.

MERGE COMPARISONS WITH DATA INSTRUCTIONS

The **Z** status flag revolves around the number zero, so if you use 0 to compare to you can often combine the comparison with the data instruction that sets the flags.

This is also true for testing individual bits. The **N** and **C** flags are effectively bits 31 and 32, respectively. If you can manipulate the algorithm to use those, you don't need a `cmp` or `tst`.

> **"NO, THERE IS ANOTHER"**
>
> You probably know this already, but this is a good time to repeat the warning: watch out for off-by-one errors (also known as obi-wan errors). It is just way too easy to do one iteration too few or too many, so always check whether the code you have does the right thing. Goes for other programming languages too, of course.

Function calls

Function calls use a special kind of branching instruction, namely `bl`. It works exactly like the normal branch, except that it saves the address after the `bl` in the link register (`r14` or `lr`) so that you know where to return to after the called function is finished. In principle, you can return with to the function using '`mov pc, lr`', which points the program counter back to the calling function, but in practice you might be better off with `bx` (Branch and eXchange). The difference is that `bx` can also switch between ARM and Thumb states, which isn't possible with the `mov` return. Unlike `b` and `bl`, `bx` takes a register as its argument, instead of a label. This register will usually be `lr`, but the others are allowed as well.

There's also the matter of passing parameters to the function and returning values from it. In principle you're free to use any system you like, it is recommended to ARM's own ARM Architecture Procedure Call Standard (AAPCS) for this. For the majority of the work this can be summarized like this.

- The first 4 arguments go into r0-r3. Later ones go on the stack, in order of appearance.
- The return value goes into r0.
- The scratch registers r0-r3 (and r12) are free to use without restriction in a function. As such, after *calling* a function they should be considered 'dirty'.

- The other registers must leave a function with the same values as they came in. Push them on the stack before use, and pop them when leaving the function. Note that another `bl` sets `lr`, so stack that one too in that case.

Below is a real-world example of function calling, complete with parameter passing, stackwork and returning from the call. The function `oamcpy()` copies OBJ_ATTRs. The function uses the same argument order as `memcpy()`, and these need to be set by the calling function; before and after the call, `lr` is pushed and popped. These two things are part of what's called the function overhead, which can be disastrous for small functions, as we've already seen. Inside `oamcpy()` we either jump back immediately if the count was 0, or proceed with the copies and then return. Note that `r4` is stacked here, because that's what the caller expects; if I hadn't and the caller used `r4` as well, I'd be screwed and rightly so. I should probably point out that `r12` is usually considered a scratch register as well, which I could have used here instead of `r4`, removing the need for stacking.

```
@ Function calling example: oamcpy
@ void oamcpy(OBJ_ATTR *dst, const OBJ_ATTR *src, u32 nn);
@ Parameters: r0= dst; r1= src; r2= nn;
    .align  2
oamcpy:
    cmp     r2, #0
    bxeq    lr          @ Nothing to do: return early
    push    {r4}        @ Put r4 on stack
.Lcpyloop:
        ldmia   r1!, {r3, r4}
        stmia   r0!, {r3, r4}
        subs    r2, #1
        bne     .Lcpyloop
    pop     {r4}        @ Restore r4 to its original value
    bx      lr          @ Return to calling function

@ Using oamcpy.
    @ Set arguments
    mov     r0, #0x07000000
    ldr     r1,=obj_buffer
    mov     r2, #128
    push    {lr}        @ Save lr
    bl      oamcpy      @ Call oamcpy (clobbers lr; assumes clobbering
of r0-r3,r12)
    pop     {lr}        @ Restore lr
```

> **USE BX INSTEAD OF MOV PC,LR**
>
> The `bx` instruction is what makes interworking between ARM and Thumb function possible. Interworking is good. Therefore, `bx` is good.

This concludes the primary section on ARM assembly. There are more things like different processor states, and data-swap (`swp`) and co-processor instructions, but those are rare. If you need more information, look them up in the proper reference guides. The next two subsections cover instruction speeds and what an instruction actually looks like in binary, i.e., what the processor actually processes. Neither section is necessary in the strictest sense, but still informative. If you do not want to be informed, move on to the next section: the Thumb instruction set.

Cycle counting

Since the whole reason for coding in asm is speed (well, that and space efficiency), it is important to know how fast each instruction is so that you can decide on which one to use where. The term 'cycle' actually has two different meanings: there is the ***clock cycle***, which measures the amount of clock ticks, and there's ***functional cycle*** (for lack of a better word), which indicates the number of stages in an instruction. In an ideal world these two would be equal. However, this is the real world, where we have to deal with waitstates and buswidths, which make functional cycles cost multiple clock cycles. A ***wait(state)*** is the added cost for accessing memory; memory might just not be as fast as the CPU itself. Memory also as a fixed ***buswidths***, indicating the maximum number of bits that can be sent in one cycle: if the data you want to transfer is larger than the memory bus can handle, it has to be cut up into smaller sections and put through, costing additional cycles. For example, ROM has a 16bit bus which is fine for transferring bytes or halfwords, but words are transferred as two halfwords, costing two functional cycles instead of just one. If you hadn't guessed already, this is why Thumb code is recommended for ROM/EWRAM code.

There are three types of functional cycles: the *non-sequential* (N), the *sequential* (S) and the *internal* (I) cycle. There is a fourth, the coprocessor cycle (C), but as the GBA doesn't have a coprocessor I'm leaving that one out.

Anyway, the N- and S-cycles have to do with memory fetches: if the transfer of the current (functional) cycle is not related to the previous cycle, it is non-sequential; otherwise, it's sequential. Most instructions will be sequential (from the instruction fetch), but branches and loads/stores can have non-sequentials as they have to look up another address. Both sequential and non-sequential cycles are affected by section waitstates. The internal cycles is one where the CPU is already doing something else so that even though it's clear what the next action should be, it'll just have to wait. I-cycles do not suffer from waitstates.

Instruction	Cycles
Data	1S
ldr(type)	$1N + 1N_d + 1I$
str(type)	$1N + 1N_d$
ldm {n}	$1N + 1N_d + (n-1)S_d + 1I$
stm {n}	$1N + 1N_d + (n-1)S_d$
b/bl/bx/swi	$2S + 1N$

Section	Bus	Wait (N/S)	Access 8/16/32
BIOS	32	0/0	1/1/1
EWRAM	16	2/2	3/3/6
IWRAM	32	0/0	1/1/1
IO	32	0/0	1/1/1
PAL	16	0/0	1/1/2
VRAM	16	0/0	1/1/2
OAM	32	0/0	1/1/1
ROM	16	4/2	5/5/8

Table 23.8: Section default timing details. See also GBATEK memory map.

Thumb bl	3S + 1N
mul	1S + mI
mla/mull	1S + (m+1)I
mlal	1S + (m+2)I

Table 23.7: Cycle times for the most important instructions.

Table 23.7 shows how much the instructions cost in terms of N/S/I cycles. How one can arrive to these cycle times is explained below. Table 23.8 lists the buswidths, the waitstates and the access times in clock cycles for each section. Note that these are the default wait states, which can be altered in REG_WAITCNT.

The data presented here is just an overview of the most important items, for all the gory details you should look them up in GBATEK or the official documents.

- The cost of an instruction begins with fetching it from memory, which is a 1S operation. For most instructions, it ends there as well.

- Memory instructions also have to get data from memory, which will cost $1N_d$;

  I've added a subscript d here because this is an access to the section where the *data* is kept, whereas other waitstates are taken from the section where the code resides. This is an important distinction. Also, because the address of the next instruction won't be related to the current address, its timing will begin as a 1N instead of a 1S. This difference is encompassed in the transfer timing. Note however that most documentation list `ldr` as 1S+1N+1I, but this is false! If you actually test it, you'll see that it is really $1N+1N_d+1I$.

- Block transfer behave like normal transfers, except that all accesses after the first are S_d-cycles.

- Branches need an extra 1N+1S for jumping to the new address and fetching the resetting the pipeline (I think). Anything that changes `pc` can be considered a branch. The Thumb `bl` is actually two instructions (or, rather, one instruction and an address), which is why that has an additional 1S.

- Register-shifted operations add 1I to the base cost because the value has to be read from the register before it can be applied.

- Multiplies are 1S operations, plus 1I for every significant byte of the *second* operand. Yes, this does mean that the cost is asymmetric in the operands. If you can estimate the ranges of values for the operands, try to put the lower one in the second operand. Another 1I is required for the add of `mla`, and one more for long multiplications.

> **THERE IS NO 1S IN LOADS!**
>
> Official documentation gives $1S+1N_d+1I$ as the timing of `ldr`, but this is not entirely accurate. It is actually $1\mathbf{N}+1N_d+1I$. The difference is small and only visible for ROM instructions, but could be annoying if you're wondering why what you predicted and what you measured for your routine doesn't match exactly. This applies to `ldm` and perhaps `swp` too.
>
> See forum:9602 for a little more on the subject.

Anatomy of an addition

As an example of how instructions are actually formatted, I invite you to an in-depth look at the `add` instruction. This will be the absolute rock bottom in terms of GBA

programming, the lowest level of them all. Understanding this will go a long way in understanding the hows and whys (and why-nots!) of ARM assembly.

Before I show the bits, I should point out that `add` (and in fact all data instructions) come in three forms, depending on the second operand. This can be an immediate field (numeric value), an immediate-shifted register or a register-shifted register. Bits 4 and 19h indicate the type of `add`, the lower 12 bits describe the second operand; the rest are the same for all `add` forms.

Table 23.9: The add instruction(s)

	1F – 1C	1B 1A	19	18 – 15	14	13 – 10	F – C	B – 8	
add Rd, Rn, #								IR	
add Rd, Rn, Rm Op #	cnd	TA	IF	TB	S	Rn	Rd	IS	
add Rd, Rn, Rm Op Rs								Rs	

Top 20 bits for `add`; denote instruction type, status/conditional flags and destination and first operand registers.

bits	name	description
C - F	Rd	**Destination register**.
10 - 13	Rn	**First operand register**.
14	S	Set **Status** bits (the `-s` affix).
15-18	TB	**Instruction-type** field. Must be 4 for `add`.
19	IF	**Immediate flag**. The second operand is an immediate value if set, and a (shifted) register if clear.
1A - 1C	TA	Another **instruction-type** field. Is zero for all data instructions.

1D - 1F	cnd	**Condition** field.

Lower 12 bits for `add` ; these form the second operand.

bits	name	description
0 - 7	IN	**Immediate Number** field. The second operand is `IN ror 2*IR` .
8 - B	IR	**Immediate Rotate** field. This denotes the rotate-right amount applied to *IN*.
0 - 3	Rm	**Second operand** register.
4	SF	**Shift-operand flag**. If set, the shift is the immediate value in *IS*; if clear, the shift comes from register *Rs*.
5 - 6	ST	**Shift type**. 0: `lsl` , 1: `lsr` 2: `asr` , 3: `ror`
7 - B	IS	**Immediate Shift** value. Second operand is `Rm Op IS` .
8 - B	Rs	**Shift Register**. Second operand is `Rm Op Rs` .

These kinds of tables should feel familiar: yes, I've also used them for IO-registers throughout Tonc. The fact of the matter is that the instructions are coded in a very similar way. In this case, you have a 32bit value, with different bitfields describing the type of instruction (`TA` =0 and `TB` =4 indicate is an `add` instruction), the registers to use (`Rd` , `Rd` and maybe `Rm` and `Rs` too) and a few others. We have seen this thing a number of times by now, so there should be no real difficulty in understanding here. The assembly instructions are little more than the BUILD macros I've used a couple of times, only this time it's the assembler that turn them into raw numbers instead of the preprocessor. Having said that, it *is* possible to construct the instructions manually, even at run-time, but whether you really want to do such a thing is another matter.

Now, the top 20 bits indicate the kind of instruction it is and which registers it uses. The bottom 12 are for *Op2*. If this involves a shifted register the bottom 4 bits indicate `Rm` . Bits 5 and 6 describe the type of shift-operation (shift-left, shift-right or rotate-right) and depending on bit 4, bits 7 to 11 form either the register to shift by (`Rs`) or a shift-value (5 bits for 0 to 31). And then there's the immediate operand …

Sigh. Yes, here are the mere twelve bits you can use for an immediate operand, divided into a 4bit rotate part and 8bit immediate part. The allowable immediate values are given by `IN ror 2*IR`. This seems like a small range, but interestingly enough you can get quite far with just these. It does mean that you can never load variable addresses into a register in one go; you have to get the address first with a PC-relative load and then load its value.

```
@ Forming 511(0x101)
    mov      r0, #511    @ Illegal instruction! D:

    mov      r0, #256    @ 256= 1 ror 24, so still valid
    add      r0, #255    @ 256+255 = 511

    @ Load 511 from memory with ldr
    ldr      r0, .L0

    @ Load 511 from memory with special ldr
    @ NOTE: no '#' !
    ldr      r0,=511
.L0:
    .word    511
```

Anyway, the bit patterns of table 23.9 is what the processor actually sees when you use an `add` instruction. You can see what the other instructions look like in the references I gave earlier, especially the quick references. The orthogonality of the whole instruction set shows itself in very similar formatting of a given class of instructions. For example, the data instructions only differ by the `TB` field: 4 for `add`, 2 for `sub`, et cetera.

Thumb assembly

The Thumb instruction set is a subset of the full list of ARM instructions. The defining feature of Thumb instructions is that they're only 16 bits long. As a result a function in Thumb can be much shorter than in ARM, which can be beneficial if you don't have a lot of room to work with. Another point is that 16bit instructions will pass through a

16bit databus in one go and be executed immediately, whereas execution of 32bit instructions would have to wait for the second chunk to be fetched, effectively halving the instruction speed. Remember that ROM and EWRAM, the two main areas for code have 16bit buses, which is why Thumb instructions are advised for GBA programming.

There are downsides, of course; you can't just cut the size of an instruction in half and expect to get away with it. Even though Thumb code uses many of the same mnemonics as ARM, functionality has been severely reduced. For example, the only instruction that can be conditional is the branch, `b`; instructions can no longer make use of shifts and rotates (these are separate instructions now), and most instructions can only use the lower 8 registers (`r0-r7`); the higher ones are still available, but you have to move things to the lower ones because you can use them.

In short, writing efficient Thumb code is much more challenging. It's not exactly bondage-and-disciple programming, but if you're used to the full ARM set you might be in for a surprise now and then. Thumb uses most of ARM's mnemonics, but a lot of them are restricted in some way so learning how to code in Thumb basically comes down to what you *can't* do anymore. With that in mind, this section will cover the differences between ARM and Thumb, rather than the Thumb set itself.

- **Removed instructions**. A few instructions have been cut altogether. Of the various multiplication instructions only `mul` remains, reverse subtractions (`rsb`, `rsc`) are gone, as are the swapping and coprocessor instructions, but those are rare anyway.

- **'New' instructions**. The mnemonics are new, but really, they're just special cases of conventional ARM instructions. Thumb has separate shift/rotate opcodes: `lsl`, `lsr`, `asr` and `ror` codes, which are functionally equivalent to '`mov Rd, Rm, Op2`'. There is also a '`neg Rd,Rm`' for *Rd*= 0−*Rm*, essentially an `rsb`. And I suppose you could call `push` and `pop` new, as they don't appear as ARM opcodes in some devkits.

- **No conditionals**. Except on branch. Hello, gratuitous labelling `:\`.

- The **Set Status** flag is always on. So in Thumb `sub` will always work as a `subs`, etc.

- **No barrel shifter**. Well, it still exist, of course; you just can't use it in conjunction with the instructions anymore. This is why there are separate bitshift/-rotate opcodes.

- **Restricted register availability**. Unless explicitly stated otherwise, the instructions can only use `r0-r7`. The exceptions here are `add`, `mov` and `cmp`, which can at times use high-regs as operands. This restriction also applies to memory operations, with small exceptions: `ldr/str` can still use PC-or SP-relative stuff; `push` allows `lr` in its register list and `pop` allows `pc`. With these, you could return from functions quickly, but you should use `bx` for that anyway. Fortunately, `bx` also allows use of every register so you can still do '`bx lr`'.

- **Little to no immediate or second operand support**. In ARM-code, most instructions allowed for a second operand *Op2*, which was either an immediate value or a (shifted) register. Most Thumb data instructions are of the form '`ins Rd, Rm`' and correspond to the C assignment operators like `+=` and `|=`. Note that `Rm` is a register, not an immediate. The only instructions that break this pattern are the shift-codes, `add`, `sub`, `mov` and `cmp`, which can have both immediate values and second operands. See the reference docs for more details.

- **No write-back in memory instructions**. That means you will have to use at least one extra register and extra instructions when traversing arrays. There is one exception to this, namely block-transfers. The only surviving versions are `ldmia` and `stmia`, and in both versions the write-back is actually required.

- **Memory operations are tricky**. Well, they are! ARM memory opcodes were identical in what they could do, but here you have to be on your toes. Some features are completely gone (write-back and shifted register offsets), and the others aren't always available to all types. Register-offset addressing is always available, but immediate offsets do not work for the signed loads (`ldrsh`,

`ldrsb`). Remember that the registers can only be `r0-r7` , except for `ldr/str` : there you can also use PC and SP-relative stuff (with immediate offsets). Table 23.10 gives an overview.

	[Rn,Rm]	[Rn,#]	[pc/sp,#]
ldr/str	+	+	+
ldrh/strh	+	+	-
ldrb/strb	+	+	-
ldrsh/ldrsb	+	-	-

Table 23.10. Thumb addressing mode availability.

Actually, '`ldrh Rd,=X`' also seem to work, but these are actually converted into '`ldr Rd,=X`' internally.

Is that it? Well no, but it's enough. Remember, Thumb is essentially ARM Lite: it looks similar, but it has lost a lot of substance. I'd suggest learning Thumb code in that way too: start with ARM then learn what you can't do anymore. This list gives most of the things you need to know; for the rest, just read at the assembler messages you will get from time to time and learn from the experience.

GAS: the GNU assembler

The instructions are only part of functional assembly, you also need **directives** to tie code together, control sections and alignment, create data, etc. Somewhat fittingly, directives seem to be as unportable as assembly itself: directives for one assembler might not work under others.

This section covers the main directives of the GNU assembler, GAS. Since we're already working with the GNU toolchain, the choice for this assembler is rather obvious. GAS is already part of the normal build sequence, so there is no real loss of

functionality, and you can work together with C files just as easily as with other assembly; it's all the same to GCC. Another nice feature is that you can use the preprocessor so if you have header files with just preprocessor stuff (#include and #define only), you can use those here as well. Of course, you could do that anyway because `cpp` is a standalone tool, but you don't have to resort to trickery here.

But back to directives. In this section you'll see some of the most basic directives. This includes creating symbols for functions (both ARM and Thumb) and variables. With out these you wouldn't be able to do anything. I'll also cover basic datatypes and placing things in specific sections. There are many other directives as well, but these should be the most useful. For the full list, go to the GAS manual at www.gnu.org.

Symbols

Data (variable and constant) and functions are collectively known as **symbols** and, just like in C, these have declarations and definitions. Any label (a valid identifier on a separate line ending with a colon) is potentially a symbol definition, but it's better to distinguish between global and local labels. Simply put, a label is global if there is a '`.global` *lname*' directive attached to it that makes it visible to the outside world. Local labels are everything else, and conventionally start with '`.L`', though this is not required. If you want to use symbols from outside, you have to use '`.extern` *lname*'.

Now, unless you're using some notational device, a label tells you nothing about what it actually stands for: it gives you no information on whether it's data or a function, nor the number of arguments for the latter. There is '`.type,` *str*' directive that lets you indicate if it's a function (*str* = `%function`) or some form of data (*str* = `%object`), but that's about it. Since you can tell that difference by looking at what's after the label anyway, I tend to leave this out. For other information, please comment on what the symbols mean.

The directives you'd use for data will generally tell you what the datatypes are, but that's something for a later subsection. Right now, I'll discuss a few directives for functions. The most important one is '`.code` *n*', where *n* is 32 or 16 for ARM or

Thumb code respectively. You can also use the more descriptive `.arm` and `thumb` directives, which do the same thing. These are global settings, remaining active until you change them. Another important directive is `.thumb_func`, which is **required** for interworking Thumb functions. This directive applies to the next symbol label. Actually, `.thumb_func` already implies `.thumb`, so adding the latter explicitly isn't necessary.

A very important and sneaky issue is alignment. **You** are responsible for aligning code and data, not the assembler. In C, the compiler did this for you and the only times you might have had problems was with alignment mismatches when casting, but here both code *and* data can be misaligned; in assembly, the assembler just strings your code and data together as it finds it, so as soon as you start using anything other than words you have the possibility of mis-alignments.

Fortunately, alignment is very easy to do: '`.align` *n*' aligns to the next 2^n byte boundary and if you don't like the fact that *n* is a power here, you can also use '`.balign` *m*', which aligns to *m* bytes. These will update the current location so that the next item of business is properly aligned. Yes, it applies to the *next* item of code/data; it is not a global setting, so if you intend to have mixed data-sizes, be prepared to align things often.

Here are a few examples of how these things would work in practice. Consider it standard boilerplate material for the creation and use of symbols.

```
@ ARM and Thumb versions of m5_plot
@ extern u16 *vid_page;
@ void m5_plot(int x, int y, u16 clr)
@ {   vid_page[y*160+x]= clr;    }

@ External declaration
@ NOTE: no info on what it's a declaration of!
    .extern vid_page            @ extern u16 *vid_page;

@ ARM function definition
@ void m5_plot_arm(int x, int y, u16 clr)
    .align 2                    @ Align to word boundary
    .arm                        @ This is ARM code
    .global m5_plot_arm         @ This makes it a real symbol
    .type m5_plot_arm STT_FUNC  @ Declare m5_plot_arm to be a function.
m5_plot_arm:                    @ Start of function definition
    add     r1, r1, lsl #2
    add     r0, r1, lsl #5
    ldr     r1,=vid_page
    ldr     r1, [r1]
    mov     r0, r0, lsl #1
    strh    r2, [r1, r0]
    bx      lr

@ Thumb function definition
@ void m5_plot_thumb(int x, int y, u16 clr)
    .align 2                    @ Align to word boundary
    .thumb_func                 @ This is a thumb function
    .global m5_plot_thumb       @ This makes it a real symbol
    .type m5_plot_thumb STT_FUNC    @ Declare m5_plot_thumb to be a
function.
m5_plot_thumb:                  @ Start of function definition
    lsl     r3, r1, #2
    add     r1, r3
    lsl     r1, #5
    add     r0, r1
    ldr     r1,=vid_page
    ldr     r1, [r1]
    lsl     r0, #1
    strh    r2, [r1, r0]
    bx      lr
```

The functions above show the basic template for functions: three lines of directives, and a label for the function. Note that there is no required order for the four directives, so you may see others as well. In fact, the `.global` directive can be separated completely from the rest of the function's code if you want. Also note the

use of `.extern` to allow access to `vid_page`, which in libtonc always points to the current back buffer. To be honest, it isn't even necessary because GAS assumes that all unknown identifiers come from other files; nevertheless, I'd suggest you use it anyway, just for maintenance sake.

And yes, these two functions do actually form functional mode 5 pixel plotters. As an exercise, try to figure out how they work and why they're coded the way they are. Also, notice that the Thumb function is only two instructions longer than the ARM version; if this were ROM-code, the Thumb version would be a whole lot faster due to the buswidth, which is exactly why Thumb code is recommended there.

GCC 4.7 NOTE: SYMBOL-TYPE FOR FUNCTIONS NOW REQUIRED

As of GCC 4.7, the `.type` directive is pretty much required for functions. Or, rather, it is required if you want ARM and Thumb interworking to work. Just add the following line to each function definition:

```
.type [function-name] STT_FUNC
```

`STT_FUNC` is an internal macro that expands to the correct attribute (presumably `%function`). Replace `[function-name]` with the real function name.

IMPLICIT EXTERN CONSIDERED HARMFUL

The `.extern` directive for external symbols is actually not required: GAS assumes that unknown identifiers are external. While I can see the benefits of implicit declarations/definitions, I still think that it is a *bad* idea. If you've ever misspelled an identifier in languages that have implicit definitions, you'll know why.

And yes, I know this is actually a non-issue because it'll get caught by the linker anyway, but explicitly mentioning externals is probably still a good idea. `:P`

Definition of variables

Of course, you can also have data in assembly, but before we go there a word about acceptable number formats and number uses. GAS uses the same number representations as C: plain numbers for decimals, '`0`' for octal and '`0x`' for hexadecimal. There is also a binary representation, using the '`0b`' prefix: for example, `0b1100` is 12. I've already used numbers a couple of times now, and you should have noticed that they're sometimes prepended by '`#`'. The symbol is not actually part of the number, but is the indicator for an immediate value.

It is also possible to perform arithmetic on numbers in assembly. That is to say, you can have something like '`mov r0, #1+2+3+4`' to load 10 into `r0`. Not only arithmetic, but bit operations will work too, which can be handy if you want to construct masks or colors. Note, this only works for constants.

And now for adding data to your code. The main data directives are `.byte`, `.hword`, and `.word`, which create bytes, halfwords and words, respectively. If you want you can count `.float` among them as well, but you don't want to because floats are evil on the GBA. Their use is simple: put the directive on a line and add a number after it, or even a comma-separated list for an array. If you add a label in front of the data, you have yourself a variable. There are also a few directives for strings, namely `.ascii`, `.asciz` and `.string`. `.asciz` and `.string` differ from `.ascii` in that they add the terminating '`\0`' to the string, which is how strings usually work. Just like the other data directives, you can make an array of strings by separating them with commas.

You can see some examples below; note that what should have been the `hword_var` will definitely be misaligned and hence useless.

```
    .align 2
word_var:                   @ int word_var= 0xCAFEBABE
    .word   0xCAFEBABE
word_array:                 @ int word_array[4]= { 1,2,3,4 }
    .word   1, 2, 3, 4       @ NO comma at the end!!
byte_var:                   @ char byte_var= 0;
    .byte   0
hword_var:                  @ NOT short hword_var= 0xDEAD;
    .hword  0xDEAD          @   due to bad alignment!
str_array:                    @ Array of NULL-terminated strings:
    .string "Hello", "Nurse!"
```

Data sections

So now you know how to make code and variables, you have to put them into the proper sections. A *section* is a contained area where code and data are stored; the linker uses a linkscript to see where the different sections are and then adds all your symbols and links accordingly. The format for sections is ' `.section` *secname* ', with optional ' `, "` *flags* `", %` *type* ' information I'll get to in a minute.

Traditionally, the section for code is called `.text` and that for data is called `.data` , but there are a few more to consider: the general sections `.bss` and `.rodata` , and the GBA-specific `.ewram` and `.iwram` . In principle, these four are data sections, but they can be used for code by setting the correct section flags. As you might have guessed, `.ewram` stands for the EWRAM section (0200:0000h), `.iwram` is IWRAM (0300:0000h) and `.rodata` is ROM (0800:0000). The `.bss` section is a section intended for variables that are either uninitialized or filled with zero. The nice thing about this section is that it requires no ROM space, as it doesn't have data to store there. The section will be placed in IWRAM, just like the `.data` . You may also sometimes see `.sbss` which stands for 'small bss' and has a similar function as the standard `.bss` , but happens to be placed in EWRAM.

These data sections can be used to indicate different kinds of data symbols. For example, constants (C keyword `const`) should go into `.rodata` . Non-zero (and non-const, obviously) initialised data goes into `.data` , and zero or uninitialized data is to be placed into `.bss` . Now, you still have to indicate the amount of storage you need

for each bss-variable. This can be done with '`.space` *n*', which indicates *n* zero bytes (see also `.fill` and `.skip`), or '`.comm` *name*, *n*, *m*', which creates a bss symbol called *name*, allocates *n* bytes for it and aligns it to *m* bytes too. GCC likes to use this for uninitialized variables.

```c
// C symbols and their asm equivalents

// === C versions ===
int var_data= 12345678;
int var_zeroinit= 0;
int var_uninit;
const u32 cst_array[4]= { 1, 2, 3, 4 };
u8 charlut[256] EWRAM_BSS;
```

```asm
@ === Assembly versions ===
@ Removed alignment and global directives for clarity

@ --- Non-zero Initialized data ---
    .data
var_data:
    .word   12345678

@ -- Zero initialized data ---
    .bss
var_zeroinit:
    .space     4

@ --- Uninitialized data ---
@ NOTE: .comm takes care of section, label and alignment for you
@    so those things need not be explicitly mentioned
    .comm var_uninit,4,4

@ --- Constant (initialized) data ---
    .section .rodata
cst_array:
    .word 1, 2, 3, 4

@ --- Non-zero initialized data in ewram ---
    .section .sbss
charlut:
    .space 256
```

ASSEMBLY FOR DATA EXPORTERS

Assembly is a good format for exporting data to. Assembling arrays is faster than compilation, the files can be bigger and you can control alignment more easily. Oh, any you can't be tempted to #include the data, because that simply will not work.

```
    .section .rodata    @ in ROM, please
    .align  2           @ Word alignment
    .global foo         @ Symbol name
foo:
    @ Array goes in here. Type can be .byte, .hword or .word
    @ NOTE! No comma at the end of a line! This is important
    .hword
0x0000,0x0001,0x0002,0x0003,0x0004,0x0005,0x0006,0x0007
    .hword
0x0008,0x0009,0x000A,0x000B,0x000C,0x000D,0x000E,0x000F
    .hword
0x0010,0x0011,0x0012,0x0013,0x0014,0x0015,0x0016,0x0017

    ...
```

You need a const section, word alignment, a symbol declaration and definition and the data in soe form of array. To make use of it, make a suitable declaration of the array in C and you're all set.

Code sections

That was data in different sections, now for code. The normal section for code is `.text`, which will equate to ROM or EWRAM depending on the linker specs. At times, you will want to have code in IWRAM because it's a lot faster than ROM and EWRAM. You might think that '`.section .iwram` does the trick, but unfortunately this does not seem generally true. Because IWRAM is actually a data section, you have to add section-type information here as well. The full section declaration needs to be '`.section .iwram, "ax", %progbits`', which marks the section as allocatable and executable (`"ax"`), and that the section contains data as well (`%progbits`), although this last bit doesn't seem to be required.

Another interesting point is how to call the function once you have it in IWRAM. The problem is that IWRAM is too far away from ROM to jump to in one go, so to make it work you have to load the address of your function in a register and then jump to it using `bx` . And set `lr` to the correct return address, of course. The usual practice is to load the function's address into a register and branch to a dummy function that just consists of a `bx` using that register. GCC has these support functions under the name `_call_via_` *rx* , where *rx* is the register you want to use. These names follow the GCC naming scheme as given in table 23.1.

```
@ --- ARM function in IWRAM: ---
    .section .iwram, "ax", %progbits
    .align 2
    .arm
    .global iw_fun
    .type iw_fun STT_FUNC
iw_fun:
    @ <code goes in here>


@ --- Calling iw_fun somewhere ---
    ldr r3,=iw_fun      @ Load address
    bl  _call_via_r3   @ Set lr, jump to long-call function


@ --- Provided by GCC: ---
_call_via_r3:
    bx  r3        @ Branch to r3's address (i.e., iw_fun)
                  @ No bl means the original lr is still valid
```

The `_call_by_` *rx* indirect branching is how function calling works when you use the `-mlong-calls` compiler flag. It's a little slower than straight branching, but generally safer. Incidentally, this is also how interworking is implemented.

> **STANDARD AND SPECIAL SECTIONS**
>
> Sections `.text` , `.data` and `.bss` are standard GAS sections and do not need explicit mention of the `.section` directive. Sections `.ewram` , `.iwram` , `.rodata` and `.sbss` and more GBA specific, and do need `.section` in front of them.

With this information, you should be able to create functions, variables and even place them into the right sections. This is probably 90% of whay you might want to do with directives already. For the remaining few, read the manual or browse through GCC generated asm. Both should point you in the right direction.

A real world example: fast 16/32-bit copiers

In the last section, I will present two assembly functions – one ARM and one Thumb – intended for copying data quickly and with safety checks for alignment. They are called `memcpy16()` and `memcpy32()` and I have already used these a number of times throughout Tonc. `memcpy32()` does what `CpuFastSet()` does, but without requiring that the word-count is a multiple of 8. Because this is a primary function, it's put in IWRAM as ARM code. `memcpy16()` is intended for use with 16bit data and calls `memcpy32()` if alignments of the source and destination allow it and the number of copies warrant it. Because its main job is deciding whether to use `memcpy32`, this function can stay in ROM as Thumb code.

This is not merely an exercise; these functions are there to be used. They are optimized and take advantage of most of the features of ARM/Thumb assembly. Nearly everything covered in this chapter can be found here, so I hope you've managed to keep up. To make things a little easier, I've added the C equivalent code here too, so you can compare the two.

Also, these functions are not just for pure assembly projects, but can also be used in conjunction with C code, and I'll show you how to do this too. As you've already seen demos using these functions without any hint that they were assembly functions (apart from me saying so), this part isn't actually to hard. So that's the program for this section. Ready? Here we go.

memcpy32()

This function will copy words. *Fast*. The idea is to use 8-fold block-transfers where possible (just like `CpuFastSet()`), and then copy the remaining 0 to 7 words of the word count with simple transfers. Yes, one could make an elaborate structure of tests and block-transfers to do these residuals in one go as well, but I really don't think that's worth it.

One could write a function for this in C, which is done below. However, even though GCC does use block-transfers for the BLOCK struct-copies, I've only seen it go up to 4-fold `ldm/stm` s. Furthermore, it tends to use more registers than strictly necessary. You could say that GCC doesn't do its job properly, but it's hard to understand what humans mean, alright? If you want it done as efficient as possible, do it your damn self. Which is exactly what we're here to do, of course.

```c
// C equivalent of memcpy32
typedef struct BLOCK { u32 data[8]; } BLOCK;

void memcpy32(void *dst, const void *src, uint wdcount) IWRAM_CODE
{
    u32 blkN= wdcount/8, wdN= wdcount&7;
    u32 *dstw= (u32*)dst, *srcw= (u32*)src;
    if(blkN)
    {
        // 8-word copies
        BLOCK *dst2= (BLOCK*)dst, *src2= (BLOCK*)src;
        while(blkN--)
            *dst2++ = *src2++;
        dstw= (u32*)dst2;  srcw= (u32*)src2;
    }
    // Residual words
    while(wdN--)
        *dstw++ = *srcw++;
}
```

The C version should be easy enough to follow. The number of words, `wdcount` is split into a block count and residual word count. If there are full block to copy, then we do so and adjust the pointers so that the residuals copy correctly. Note that `wdcount` is

the number of words to copy, not the number of bytes, and that `src` and `dst` are assumed to be word-aligned.

The assembly version –surprise, surprise– does exactly the same thing, only much more efficient than GCC will make it. There is little I can add about what it does because all things have been covered already, but there are a few things about *how* it does them that deserve a little more attention.

First, note the general program flow. The `movs` gives the number of blocks to copy, and if that's zero then we jump immediately to the residuals, `.Lres_cpy32` . What happens there also interesting: the three instructions after decrementing the word-count (in `r12`) all carry the `cs` flag. This means that if `r12` is (unsigned) lower than one, (i.e., `r12` ==0) these instructions are ignored. This is exactly what we want, but usually there is a separate check for zero-ness before the body of the loop and these extra instructions cost space and time. With clever use of conditionals, we can spare those.

The main loop doesn't use these conditionals, nor, it would seem, a zero-check. The check here is actually done at that `movs` lines as well: if it doesn't jump, we can be sure there are blocks to copy, so another check is unnecessary. Also note that the non-scratch registers `r4-r10` are only stacked when we're sure they'll actually be used. GCC normally stacks at the beginning and end of functions, but there is no reason not to delay it until it's actually necessary.

Lastly, a few words on non-assembly matters. First, the general layout: I use one indent for everything except labels, and sometimes more for what in higher languages would be loops or if-blocks. Neither is required, but I find that it makes reading easier. I also make sure that the instruction parameters are all in line, which works best if you reserve 8 spaces for the mnemonic itself. How you set the indents is a matter of personal preference and a subject of many holy wars, so I'm not touching that one here `:P`

Another point is comments. Comments are even more important in assembly than in C, but don't overdo it! Overcommenting just drowns out the comments that are

actually useful and maybe even the code as well. Comment on blocks and what each register is and maybe on important tests/branches, but do you really have to say that '`subs r2, r2, #1`' decrements a loop variable? No, I didn't think so either. It might also help to give the intended declaration of the function if you want to use it in C.

Also, it's a good idea to always add section, alignment and code-set before a function-label. Yes, these things aren't strictly necessary, but what if some yutz decides to add a function in the middle of the file which screws up these things for functions that follow it? Lastly, try to distinguish between symbol-labels and branch-labels. GCC's take on this is starting the latter with '`.L`', which is as good of a convention as any.

```
@ === void memcpy32(void *dst, const void *src, uint wdcount)
IWRAM_CODE; =============
@ r0, r1: dst, src
@ r2: wdcount, then wdcount>>3
@ r3-r10: data buffer
@ r12: wdn&7
    .section .iwram,"ax", %progbits
    .align  2
    .code   32
    .global memcpy32
    .type   memcpy32 STT_FUNC
memcpy32:
    and     r12, r2, #7     @ r12= residual word count
    movs    r2, r2, lsr #3  @ r2=block count
    beq     .Lres_cpy32
    push    {r4-r10}
    @ Copy 32byte chunks with 8fold xxmia
    @ r2 in [1,inf>
.Lmain_cpy32:
        ldmia   r1!, {r3-r10}
        stmia   r0!, {r3-r10}
        subs    r2, #1
        bne     .Lmain_cpy32
    pop     {r4-r10}
    @ And the residual 0-7 words. r12 in [0,7]
.Lres_cpy32:
        subs    r12, #1
        ldrcs   r3, [r1], #4
        strcs   r3, [r0], #4
        bcs     .Lres_cpy32
    bx  lr
```

memcpy16()

The job of the halfword copier `memcpy16()` isn't really copying halfwords. If possible, it'll use `memcpy32()` for addresses that can use it, and do the remaining halfword parts (if any) itself. Because it doesn't do much copying on its own we don't have to waste IWRAM with it; the routine can stay as a normal Thumb function in ROM.

Two factors decide whether or not jumping to `memcpy32()` is beneficial. First is the number of halfwords (`hwcount`) to copy. I've ran a number of checks and it seems that the break-even point is about 6 halfwords. At that point, the power of word copies in IWRAM already beats out the cost of function-call overhead and Thumb/ROM code.

The second is whether the incoming source and destination addresses can be resolved to word addresses. This is true if bit 1 of the source and destinations are equal (bit 0 is zero because these are valid halfword addresses), in other words: `(src^dst)&2` should not be zero. If it resolvable, do one halfword copy to word-align the addresses if necessary, then call `memcpy32()` for all the word copies. After than, adjust the original halfword stuff and if there is anything left (or if `memcpy32()` couldn't be used) copy by halfword.

```
// C equivalent of memcpy16
void memcpy16(void *dst, const void *src, uint hwcount)
{
    u16 *dsth= (u16*)dst, *srch= (u16*)src;
    // Fast-copy if and only if:
    //    (1) enough halfwords and
    //    (2) equal src/dst alignment
    if( (hwcount>5) && !(((u32)dst^(u32)src)&2) )
    {
        if( ((u32)src)&1 )  // (3) align to words
        {
            *dsth++= *srch++;
            hwcount--;
        }
        // (4) Use memcpy32 for main stint
        memcpy32(dsth, srch, hwcount/2);
        // (5) and adjust parameters to match
        srch += hwcount&~1;
        dsth += hwcount&~1;
        hwcount &= 1;
    }
    // (6) Residual halfwords
    while(hwcount--)
        *dsth++ = *srch++;
}
```

The C version isn't exactly pretty due to all the casting and masking, but it works well enough. If you were to compile it and compare it to the assembly below you should see many similarities, but it won't be exactly equal because the assembly programmer is allowed greater freedoms than GCC, and doesn't have to contend with the syntax of C.

Anyway, before the function actually starts I state the declaration, the use of the registers, and the standard boilerplate for functions. As I need more than 4 registers and I'm calling a function, I need to stack `r4` and `lr`. This time I am doing this at the start and end of the function because it's just too much of a hassle not to. One thing that may seem strange is why I pop `r4` and then `r3` separately, especially as it's `lr` that I need and not `r3`. Remember the register restrictions: `lr` is actually `r14`, which can be reached by `push`, but not `pop`. So I'm using `r3` here instead. I'm also doing this separately from the `r4`-pop because '`pop {r4,r3}`' pops the registers in the wrong order (lower regs are loaded first).

The rest of the code follows the structure of the C code; I've added numbered points to indicate where we are. Point 1 checks the size, and point 2 checks the relative alignment of source and destination. Note that what I actually do here is not AND with 2, but shift by 31, which pushes bit 1 into the carry bit; Thumb code can only AND between registers and rather than putting 2 in a register and ANDing, I just check the carry bit. You can also use the sign bit to shift to, of course, which is what GCC would do. I do something similar to check whether the pointers are already word-aligned or if I have to do that myself.

At point 4 I set up and call `memcpy32()`. Or, rather, I call `_call_via_r3`, which calls `memcpy32()`. I can't use '`bl memcpy32`' directly because its in IWRAM and `memcpy16()` is in ROM and the distance is simply too big. The `_call_via_r3` is an intermediary (in ROM) consisting only of '`bx r3`' and since `memcpy32()`'s address was in `r3` we got where we wanted to go. Returning from `memcpy32()` will work fine, as that was set by the call to `_call_via_r3`.

The C code's point 5 consisted of adjusting the source and destination pointers to account for the work done by `memcpy32()`; in the assembly code, I'm being a very sneaky bastard by not doing any of that. The thing is, after `memcpy32()` is done `r0` and `r1` would *already* be where I want them; while the rules say that `r0-r3` are clobbered by calling functions and should therefore be stacked, if I *know* that they'll only end up the way I want them, do I really have to do the extra work? I think not. Fair enough, it's not recommended procedure, but where's the fun in asm programming if you can't cheat a little once in a while? Anyway, the right-shift from `r4` counters the left-shift into `r4` that I had before, corresponding to a `r2&1`; the test after it checks whether the result is zero, signifying that I'm done and I can get out now.

Lastly, point 6 covers the halfword copying loop. I wouldn't have mentioned it here except for one little detail: the array is copied back to front! If this were ARM code I'd have used post-indexing, but this is Thumb code where no such critter exists and I'm restricted to using offsets. I could have used another register for an ascending offset (one extra instruction/loop), or incrementing the `r0` and `r1` (two extra per loop), or I could copy backwards which works just as well. Also note that I use `bcs` at the end of

the loop and not `bne` ; `bcs` is essential here because `r2` could already be 0 on the first count, which `bne` would miss.

```
@ === void memcpy16(void *dst, const void *src, uint hwcount);
=============
@ Reglist:
@  r0, r1: dst, src
@  r2, r4: hwcount
@  r3: tmp and data buffer
    .text
    .align  2
    .code   16
    .thumb_func
    .global memcpy16
    .type   memcpy16 STT_FUNC
memcpy16:
    push    {r4, lr}
    @ (1) under 5 hwords -> std cpy
    cmp     r2, #5
    bls     .Ltail_cpy16
    @ (2) Unreconcilable alignment -> std cpy
    @ if (dst^src)&2 -> alignment impossible
    mov     r3, r0
    eor     r3, r1
    lsl     r3, #31         @ (dst^src), bit 1 into carry
    bcs     .Ltail_cpy16    @ (dst^src)&2 : must copy by halfword
    @ (3) src and dst have same alignment -> word align
    lsl     r3, r0, #31
    bcc     .Lmain_cpy16    @ ~src&2 : already word aligned
    @ Aligning is necessary: copy 1 hword and align
        ldrh    r3, [r1]
        strh    r3, [r0]
        add     r0, #2
        add     r1, #2
        sub     r2, #1
    @ (4) Right, and for the REAL work, we're gonna use memcpy32
.Lmain_cpy16:
    lsl     r4, r2, #31
    lsr     r2, r2, #1
    ldr     r3,=memcpy32
    bl      _call_via_r3
    @ (5) NOTE: r0,r1 are altered by memcpy32, but in exactly the right
    @ way, so we can use them as is.
    lsr     r2, r4, #31
    beq     .Lend_cpy16
    @ (6) Copy residuals by halfword
.Ltail_cpy16:
    sub     r2, #1
    bcc     .Lend_cpy16     @ r2 was 0, bug out
    lsl     r2, r2, #1      @ r2 is offset (Yes, we're copying backward)
.Lres_cpy16:
        ldrh    r3, [r1, r2]
        strh    r3, [r0, r2]
```

```
        sub     r2, r2, #2
        bcs     .Lres_cpy16
.Lend_cpy16:
    pop     {r4}
    pop     {r3}
    bx  r3
```

Using memcpy32() and memcpy16() in C

While you're working on uncrossing your eyes, a little story on how you can call these functions from C. It's ridiculously simple actually: all you need is a declaration. Yup, that's it. GCC does really care about the language the functions are in, all it asks is that they have a consistent memory interface, as covered in the AAPCS. As I've kept myself to this standard (well, mostly), there is no problem here.

```c
// Declarations of memcpy32() and memcpy16()
void memcpy16(void *dst, const void *src, uint hwcount);
void memcpy32(void *dst, const void *src, uint wdcount) IWRAM_CODE;

// Example use
{
    extern const u16 fooPal[256];
    extern const u32 fooTiles[512];

    memcpy16(pal_bg_mem, fooPal, 256);      // Copy by halfword. Fine
    memcpy32(pal_bg_mem, fooPal, 256/2);    // Copy by word; Might be
unsafe
    memcpy32(tile_mem, fooTiles, 512);      // Src is words too, no
prob.
}
```

See? They can be called just as any other C function. Of course, you have to assemble and link the assembly files instead of #include them, but that's how you're supposed to build projects anyway.

You do need to take care to provide the **correct** declaration, though. The declaration tells the compiler how a function expects to be called, in this case destination and source pointers (in that order), and the number of (half)words to transfer. It is legal to change the order of the parameters or to add or remove some – the function won't *work* anymore, but the compiler does allow it. This is true of C functions too and

should be an obvious point, but assembly functions don't provide an easy check to determine if you've used the correct declaration, so please be careful.

MAKE THE DECLARATION FIT THE FUNCTION

This is a very important point. Every function has expectations on how it's called. The section, return-type and arguments of the declaration must match the function's, lest chaos ensues. The best way would be not explicitly mention the full declaration near the function definition, so that the user just has to copy-paste it.

USE 'EXTERN "C"' FOR C++

Declarations for C++ work a little different, due to the name mangling it expects. To indicate that the function name is *not* mangled, add ' `extern "C"` ' to the declaration.

```
// C++ declarations of memcpy32() and memcpy16()
extern "C" void memcpy16(void *dst, const void *src, uint
hwcount);
extern "C" void memcpy32(void *dst, const void *src, uint wdcount)
IWRAM_CODE;
```

And that all folks. As far as this chapter goes anyway. Like all languages, it takes time to fully learn its ins and outs, but with this information and a couple of (quick)reference documents, you should be able to produce some nice ARM assembly, or at least be able to read it well enough. Just keep your wits about you when writing assembly, please. Not just in trying to avoid bugs, but also in keeping the assembly maintainable. Not paying attention in C is bad enough, but here it can be absolutely disastrous. Think of what you want to do first, *then* start writing the instructions.

Also remember: yes, assembly can be fun. Think of it as one of those shuffle-puzzles: you have a handful of pieces (the registers) and ways of manipulating them. The goal is to get to the final picture in then least amount of moves. For example, take a look at what an optimized palette blend routine would look like. Now it's your turn `:P` .

24. The Lab

Lab introduction

The Lab stands for "laboratory", a place for me to toy with new stuff. If I have something new that may be useful, but isn't quite ready to be up anywhere else yet, I'll put it here for a while. It does mean it may get messy at times, but that's alright because that's what laboratories are for anyway. As a by-product, "lab" may double for "labyrinth", but that's just a little bonus `:)`.

Priority and drawing order

This section covers the last of the bits in the background and object control that I haven't discussed yet: *priority*. There are four priority levels, 0-3, and you can set the priority for backgrounds in REG_BGxCNT bits 0 and 1, and for objects in attribute 2, bits 10 and 11. The concept of priority is simple: higher priorities are rendered first, so they appear behind things with a lower priority. This will allow you to have objects behind backgrounds, for example.

This all sounds very simple, and it is, but there is a little more to the order of the rendering process than this. On the one hand you have the priority settings, on the other you have the obj and bg *numbers*. Objects are numbered 0 to 127, backgrounds from 0 to 3. Again, higher numbers appear behind lower numbers: in a stack of objects, obj 0 is on top; bg 0 covers the others, and objects are drawn in front of

backgrounds. This is *not* due to the priority settings; in fact the whole point of priorities is so that the default order can be altered.

The above is true for objects and backgrounds of *the same* priority. You could argue that the final obj/bg order is composed of both the priority and the obj/bg number, where priority is the most significant part. So if, for example you have obj 1 and bg 2 at priority 0, and obj 0 and bg 1 at priority 1, the order would be obj1 (prio0), bg2 (prio0), obj0 (prio1), bg1 (prio1).

Well, *mostly* …

The object-priority bug and object sorting

For the most part, the order is as mentioned above, except for the parts where obj 0 and obj 1 overlap. Due to a, well I guess you can call it a design flaw, the notion of order = priority.number isn't quite true: if you have two objects in which the priorities and numbers are asymmetrical and there is a background in between, the object that's supposed to be below the background will shine through the background in the region where the two objects overlap. This sounds very complicated, but that's just because words can't really capture what happens. Basically, something like obj1+prio0, bg0+prio0, obj0+prio1 will cause a pretty nasty graphical artifact if the rectangles of these objects overlap. However, obj0+prio0, bg0+prio0, obj1+prio1 will work fine, because the object numbers are now in line with the priorities.

Which brings us to **object sorting**: the process of making sure the object that should be first will actually *be* first, i.e., have a lower obj number. This is actually a separate issue from priorities, but it's nice to do them both in one go. In principle, an object sort is a sort like any other: you have an array or list of things, in this case OBJ_ATTR's, and you have to put them into order via some sort of **key**, a value that indicates what the sorted order should be.

Now, the key can basically be anything. A lot of top-down or isometric games use Y-sorting, as higher Y-values means the object is more in the foreground. 3D and mode 7 games can use Z-sorting, of which Y-sorting is technically a special case.

You'll also need a sorting algorithm. There are plenty of them to choose from the wiki on sorting algorithms. When picking an algorithm, remember that the number of items to sort here is maybe about a hundred tops, and that it's likely that the items don't change order all that often. Finally, you need a way to put the whole thing together; make it work with the sprite and OBJ_ATTR structs that you have.

For the moment, I've chosen primarily for *simplicity*, not speed. The sorter, `id_sort_shell()`, uses a slightly modified version of the Shell sort algorithm found in Numerical Recipes (ch 8, pp 321). Its parameters are an array of key values and the number of elements. However, it does not sort these directly (which would be fairly pointless as they're not tied to objects here), but keeps track of the sorting results in an index table, `ids[]`.

An *index table* is, well, a table of indices, obviously; it will provide the sorted order of the keys after the routine has finished. This strategy allows me to keep the object double buffer intact, which I like because it makes sprite management simpler. Also, I don't have to swap whole structs (although that's usually done by pointers anyway), and it makes the routine usable as a general sorter, not just for objects. The choice of bytes as the type for the index table does limit this, but that's just one of those space trade-offs one has to make sometimes. Changing it to use full integers isn't exactly hard, of course.

```
// sort routine (in IWRAM!)

//! Sort indices via shell sort
/*! \param keys Array of sort keys
    \param ids Array of indices. After completion keys[ids[ii]]
      will be sorted in ascending order.
    \param number of entries.
*/
IWRAM_CODE void id_sort_shell(int keys[], u8 ids[], int count)
{
    u32 ii, inc;
    // find initial 'inc' in sequence x[i+1]= 3*x[i]+1 ; x[1]=1
    for(inc=1; inc<=count; inc++)
        inc *= 3;
    // actual sort
    do
    {
        // division is done by reciprocal multiplication. So no worries.
        inc /= 3;                       // for ARM compile
        // inc = (inc*0x5556)>>16); // for Thumb compile
        for(ii=inc; ii<count; ii++)
        {
            u32 jj, id0= ids[ii];
            int key0= keys[id0]
            for(jj=ii; jj>=inc && keys[ids[jj-inc]]>key0; jj -= inc)
                ids[jj]= ids[jj-inc];
            ids[jj]= id0;
        }
    } while(inc > 1);
}
```

```
// example of use
IWRAM_CODE void id_sort_shell(int keys[], u8 ids[], int count);

int sort_keys[SPR_COUNT];       // sort keys
u8 sort_ids[SPR_COUNT];         // sorted OAM indices

void foo()
{
    int ii;
    for(ii=0; ii<SPR_COUNT; ii++)
    {
        // setup sort keys  ... somehow
        sort_keys[ii]= ... ;
    }

    // sort the indices
    id_sort_shell(sort_keys, sort_ids, SPR_COUNT);

    // custom object update
    for(ii=0; ii<SPR_COUNT; ii++)
        oam_mem[ii]= obj_buffer[sort_ids[ii]];
}
```

Note that I intend the routine to be in IWRAM (and compiled as ARM code) because it's so **very f%#$@*g slow**! Or perhaps I shouldn't say slow, just costly.

Think of how a basic sort works. You have N elements to sort. In principle, each of these has to be checked with every other element, so that the routine's speed is proportional to N^2, usually expressed as $O(N^2)$, where the O stands for order of magnitude. For sorting, $O(N^2)$ is bad. For example, when N=128, you would be looking at 16k checks. Times the number of cycles that the actual checks and updates would take. Not pleasant.

Fortunately, there are faster methods, you'd want at least an $O(N{\cdot}\log_2(N))$ for sorting algorithms, and as you can see from the aforementioned wiki, there are plenty of those and shellsort is one of them. Unfortunately, even this can be quite expensive. Again, with N=128 this is still about 900, and you can be sure the multiplier can be high, as in 80+. With ARM+IWRAM, I can manage to bring that down to 20-30, and a simple exercise in assembly gives me an acceptable 13 to 22 × $N{\cdot}\log_2(N)$.

> **THE BIG O NOTATION**
>
> The 'Big O' or order notation is a useful expression for comparing algorithms. The notation is $O(f(N))$, where N is the number of elements to work on and $f(N)$ a function, usually a combination of powers and logarithms. It shows how the runtime of an algorithm rises with increasing N. As lower order functions will eventually be overtaken by higher order ones, the former is generally preferable.
>
> The keyword here, though, is 'eventually'. It does not mention the scale of the algorithm, which varies from case to case. In some cases if N is low enough and the scales are different enough, a higher-order routine may actually outperform a lower-order one.

Now, I'll be the first to admit that the current design isn't exactly optimal anyway. Using linked lists instead of an index table may work faster, and there are other things too (the division isn't a problem, as it can be faked). However, then it wouldn't be quite as simple anymore, which is what I was going for here.

Once `id_sort_shell()` is finished, we have an table of indices arranged in such a way that `obj_buffer[sort_ids[ii]]` gives the sorted OAM entries, which is used to update to the real OAM.

Caged DNA

The demo for this section is probably the coolest and most complicated one yet. It features a double helix of objects, revolving around the center of a toroidal cage (see 24.1). All four backgrounds are used here, one for text (little as there is of that), and three parts of the cage: one front end, which obscures the objects, one back end that lies behind everything else and the middle of the toroid around which the the object rotate, i.e., they pass both in front and behind it at different times. And then there are the objects that form the helix. The two strands each consist of 48 spherical 16x16 objects. The strands are distinguishable by their colors: one is red and the other cyan. These will turn to dark red and cyan as they pass behind the central plane. Priority

settings are used to allow the objects to pass behind nearer backgrounds, and priorities *and* sorting make the object order smooth and avoid the previously mentioned obj-bg-obj bug. To summarize:

- 4 backgrounds with varying priority settings
- 96 objects revolving (in 3D) around a central pillar.
- Object priorities and number sorting to ensure proper order.
- Palette swapping to distinguish near from far objects.

You can see a schematic representation of the whole thing in fig 24.1b; table 24.1 explains the colors.
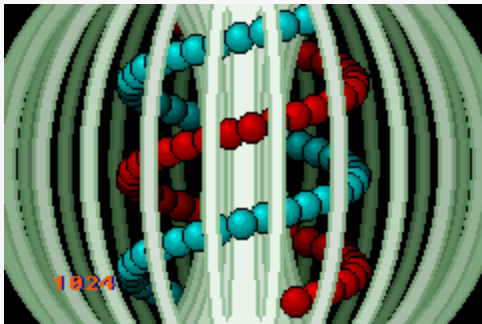
Fig 24.1a (left): Priority and sprite order demo.
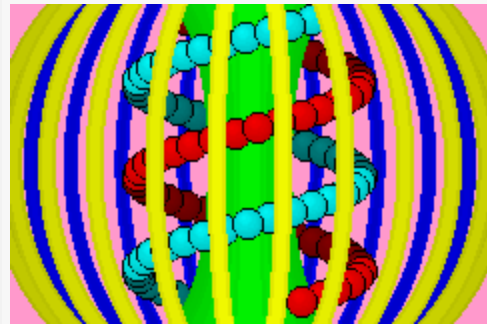
Fig 24.1b: schematic of 24.1a.

color	description	obj/bg	prio
yellow	cage near	bg1	prio0
green	cage center	bg2	prio1
blue	cage far	bg3	prio2
red	strand 1	obj_buffer[00..47]	var
cyan	strand 2	obj_buffer[48..95]	var
lt red/cyan	near orbs	OAM[0..47]	prio 1
dark red/cyan	far orbs	OAM[48..95]	prio 2

Table 24.1: legend for 24.1b.

Sprites and the helix pattern

As you can imagine, the sprite part is the trickiest thing about this demo. The helix is inherently a three dimensional path, so we need a 3D vector for each orb's position, with the coordinates being fixed-point numbers, of course. It also needs an index to the OAM shadow, linking a sprite (the orb) to the right OBJ_ATTR (the object on screen).

```c
typedef struct tagSPR_BASE
{
    VECTOR pos; // position (x, y, z)
    int id;     // oe-id in OAM buffer
} SPR_BASE;

#define SPR_COUNT 96

SPR_BASE sprites[SPR_COUNT];    // Sprite list
```

A helix is simply a circle parameterization extruded in the direction of its normal axis (see fig 24.2). Note the directions of the three principal axes: it is a right-handed system, with *x* and *y* following the directions of the screens axes, and *z* pointing into the screen. A helix rotating around the *y*-axis can be described by the following relation:

Fig 24.2: 3 periods of a helix.

$$(24.1) \qquad x(y, t) = \begin{bmatrix} A \cdot \cos(k \cdot y + \omega t) \\ y \\ A \cdot \sin(k \cdot y + \omega t) \end{bmatrix}$$

A is the radius of the helix, k is the wave number ($k=2\pi/\lambda$) and ω the angular velocity ($\omega=2\pi/T$). The wave number defines the spacing between the layers of the helix (i.e., the pitch), the angular velocity gives the speed of rotation. Note that to create the helix in fig 24.2 I actually need a negative wave number, but that's not really important right now.

In the actual code I'm going to make a slight change to the formula above to make ω variable without upsetting the whole helix. Instead of simply ωt, I'll use the integration of it for the initial phase angle: $\phi_0 = \int \omega(t)dt$. ϕ_0 will be a parameter to the function that creates the helix, and managed elsewhere. Another parameter for the pattern is $\mathbf{p}_0$, the reference point of the helix. You gotta have one of those.

```
// some constants
const VECTOR P_ORG= { 112<<8, 0<<8, 0<<8 };
#define AMP      0x3800  // amplitude (.8)
#define WAVEN   -0x002C  // wave number (.12)
#define OMEGA0   0x0200  // angular velocity (.8)

//   phi0(t) = INT(w(t'), t', 0, t)
//   (x,y,z) = ( x0+A*cos(k*y+ft), y0+y, z0+A*sin(k*y+phi0) )
void spr_helix(const VECTOR *p0, int phi0)
{
    int ii, phi;
    VECTOR dp= {0, 0, 0};
    SPR_BASE *sprL= sprites, *sprR= &sprites[SPR_COUNT/2];

    for(ii=0; ii<SPR_COUNT/2; ii++)
    {
        // phi: 0.9f ; dp: 0.8f ; WAVEN:0.12f ; phi0: 0.8f
        phi= (WAVEN*dp.y>>11) + (phi0>>7);

        // red helix
        dp.x=  AMP*lut_cos(phi)>>8;
        dp.z=  AMP*lut_sin(phi)>>8;
        vec_add(&sprL[ii].pos, p0, &dp);

        // cyan helix
        dp.x= -dp.x;
        dp.z= -dp.z;
        vec_add(&sprR[ii].pos, p0, &dp);

        dp.y += 144*256/(SPR_COUNT/2);
    }
}
```

The routine is fairly straightforward. A running counter for the y is kept in the form of `dp.y`, which is used to calculate the full phase, from which we get our sines and cosines. Since the red and cyan helices are in counter-phase, I can simply get the x and z offsets for one by switching the signs of the other. The only really tricky part is

managing the different fixed point scales for the phase; when dealing with fixed point math, always indicate the number of fractional bits, it's so very easy to get lost there.

Now that we have the double helix pattern, we need a way to link it to the objects, complete with sorting and all.

```c
void spr_update()
{
    int ii, prio, zz, *key;
    u32 attr2;
    int *key= sort_keys;
    SPR_BASE *spr= sprites;
    OBJ_ATTR *oe;

    for(ii=0; ii<SPR_COUNT; ii++)
    {
        oe= &obj_buffer[spr->id];
        // set x/y pos
        obj_set_pos(oe, spr->pos.x>>8, spr->pos.y>>8);

        // set priority based on depth.
        // HAX 1: palette swapping
        attr2= oe->attr2 & ~(ATTR2_PRIO_MASK |
(1<<ATTR2_PALBANK_SHIFT));
        zz= spr->pos.z;
        if(zz>0)
        {
            prio= 2;
            attr2 |= 1<<ATTR2_PALBANK_SHIFT;
        }
        else
            prio= 1;
        oe->attr2= attr2 | (prio<<ATTR2_PRIO_SHIFT);

        // HAX 2: sort-key contruction
        *key++= (prio<<30) + (zz>>2) - 6<<28;
        spr++;
    }

    if(g_state & S_SORT) // sort and update
    {
        id_sort_shell(sort_keys, sort_ids, SPR_COUNT);
        for(ii=0; ii<SPR_COUNT; ii++)
            oam_mem[ii]= obj_buffer[sort_ids[ii]];
    }
    else                            // regular update
        oam_update(0, SPR_COUNT);
}
```

The big loop here updates the OAM shadow, *not the real OAM*! It updates the object's position using the sprites *x* and *y* (corrected for fixed point, of course), and uses *z* to set the priority: 1 if it's on the near side (before the central pillar), and 2 if it's on the

far side (behind the pillar). It *also* does something funky with the palette, which is the first hack in the function, shortly followed by the second one.

Hack 1. I've arranged the object palette in such a way that the reds are in palette banks 4 and 5, and the cyans in banks 6 and 7 (table 24.2). This means that I can switch between the light and dark versions by toggling the first pal-bank bit, attr2 bit 12.

Immediately after this is the second hack, creating the sort key.

Hack 2. The sort key is a combination of the priority (2 bits) and the depth *z* (the rest). The lower 30 bits of `zz` work as a **signed** offset for the priority levels, so that each priority has its own depth range of [-2<<30,2<<30⟩ if one is necessary. The problem is that the keys are also signed, which would mean that priorities 2 and 3 would count as negative and therefore be sorted in front of prio 0 and 1, which would be bad. To remedy this, I subtract 0x60000000, which shifts the range of priority 0 to the most negative range where it should be.

bank	color
4	light red
5	dark red
6	light cyan
7	dark cyan

Table 24.2: object palette banks.

The last part of the function updates the OAM shadow to OAM, either with or without sorting.

SORTING DISABLED OBJECTS

Incidentally, you could easily modify the sort-key creation to account for disabled/hidden objects. All you'd have to do is assign the highest (signed) value to the sort-key, in this case 0x7FFFFFFF.

```
if( (oe->attr0&ATTR0_MODE_MASK) != ATTR0_HIDE )
    *key++= (prio<<30) + (zz>>2) - 6<<28;
else
    *key++= 0x7FFFFFFF;
```

Rest of code

The rest of the code is just `main()` and the initializer code. Most of the initializer code is pretty standard stuff: loading graphics, register inits and so on. The only interesting part is the object initialization, which sets the pal-banks to 0x4000 and 0x6000 for the red and cyan orbs. And because the sorting uses an index table instead of changing the object buffer itself, this is all I'll ever have to keep the primary colors correct.

```c
#define S_AUTO  0x0001
#define S_SORT  0x0002

const VECTOR P_ORG= { 112<<8, 0<<8, 0<<8 };

int g_phi= 0;                    // phase, integration of omega over time
int g_omega= OMEGA0;             // rotation velocity (.8)
u32 g_state= S_AUTO | S_SORT;    // state switches


void main_init()
{
    int ii;
    // --- init gfx ---
    // bgs
    memcpy32(pal_bg_mem, cagePal, cagePalLen/4);
    pal_bg_mem[0]= CLR_BLACK;
    memcpy32(tile_mem[1], cageTiles, cageTilesLen/4);
    // Hacx 3: there are 3 maps in cageMap, which have to be extracted
manually
    // front part, priority 0
    memcpy32(se_mem[5], &cageMap[ 1*32], 20*32/2);
    REG_BG1CNT= BG_CBB(1) | BG_SBB(5) | BG_8BPP | BG_PRIO(0);
    // center, priority 1
    memcpy32(se_mem[6], &cageMap[22*32], 20*32/2);
    REG_BG2CNT= BG_CBB(1) | BG_SBB(6) | BG_8BPP | BG_PRIO(1);
    // back, priority 2
    memcpy32(se_mem[7], &cageMap[43*32], 20*32/2);
    REG_BG3CNT= BG_CBB(1) | BG_SBB(7) | BG_8BPP | BG_PRIO(2);

    // object
    memcpy32(&tile_mem[4][1], ballTiles, ballTilesLen/4);
    memcpy32(pal_obj_mem, ballPal, ballPalLen/4);

    // -- init vars ---
    // init sort list
    for(ii=0; ii<SPR_COUNT; ii++)
        sprites[ii].id= sort_ids[ii]= ii;

    // --- init sprites and objects ---
    oam_init();
    for(ii=0; ii<SPR_COUNT/2; ii++)
    {
        obj_set_attr(&obj_buffer[ii], 0, ATTR1_SIZE_16, 0x4001);
        obj_set_attr(&obj_buffer[ii+SPR_COUNT/2], 0,
            ATTR1_SIZE_16, 0x6001);
    }

    spr_helix(&P_ORG, 0);
    spr_update();
```

```
    REG_DISPCNT= DCNT_BG_MASK | DCNT_OBJ | DCNT_OBJ_1D;
    int_init();
    int_enable_ex(II_VBLANK, NULL);
    txt_init_std();
    txt_init_se(0, BG_CBB(3)|BG_SBB(31), 0, 0xEC00021F, 0xEE);
}

int main()
{
    char str[32];

    main_init();

    while(1)
    {
        VBlankIntrWait();
        // kery handling
        key_poll();
        if(key_hit(KEY_START))
            g_state ^= S_AUTO;
        if(key_hit(KEY_SELECT))
            g_state ^= S_SORT;

        // movement
        if(g_state & S_AUTO)
        {
            g_omega += key_tri_shoulder()<<4;
            g_phi += g_omega;
        }
        else
            g_phi += g_omega*key_tri_shoulder();

        // sprite/obj update
        spr_helix(&P_ORG, g_phi);
        spr_update();

        // print omega
        siprintf(str, "%6d", g_omega);
        se_puts(8, 136, str, 0);
    }

    return 0;
}
```

The main loop checks for state changes, advances and updates the sprites and objects and prints the current angular velocity.

There are two state switches in `g_state`, one that toggles the sorting procedure (`S_SORT`, with the **Select** Button), and one that sets the rotation to automatic or not (`S_AUTO`, with the **Start** Button). Toggling the sorting is interesting because you can see what happens if you just set the priorities. This has two effects (see fig 24.3: first, the orb-order in each strand would be fixed and every object
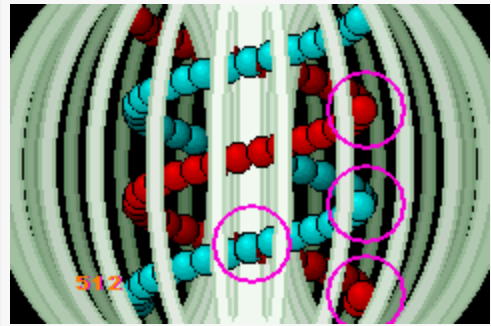
Fig 24.3: Sorting switched off.

would partially obscure the one on its left, which is incorrect for the receding parts of the strands. This is most visible at the right-most side, where the strands seem broken. The second effect is the object priority/number order bug where the deeper object can show through the background that's supposed to be occluding it.

The start button toggles between automatic and manual rotation. During automatic mode, you can change ω with the **L** and **R** Buttons. In manual mode, L and R update the phase with the current angular speed. By setting the speed really low, you can examine what happens in more detail. For example, you can clearly see that the objects in the vertical centerline are in front of both their left and right neighbors, exactly what one would expect. Unless the sorting is off, that is.

And that concludes the topic of priorities and object sorting. Remember that the priorities of objects and backgrounds aren't the only thing that determine the rendering order, the obj or bg number is also important for each priority level. Once you start mixing objects and background priorities, make sure that the object numbers follow the same order as their priorities, and that often means object sorting.

I've discussed a simple and flexible sorting method, but I warn you that it does take its time. If it's good enough, by all means use it. If it's not, faster methods can certainly be created. Linked lists, range checks, handcrafted assembly (see *id_sort_shell2.s* in the *prio_demo* directory for example) can all help make it faster. but the final implementation will be up to you.

A. Numbers, bits and bit operations

- Numbers
- Of bits and bytes
- Bit operations

Numbers

The true meaning of symbols

> "There are 10 kinds of people in the world, those that understand binary and those that don't."

If you don't get the joke, you belong in the latter category. Just like everyone else, in your youth you've probably learned that the combination of numerals '1' and '0' means ten. Not so—not exactly. The primary problem here is the meaning of symbols. Now, what I'm about to tell you is key to understanding mystifying stuff out there, so gather around and let me tell you something about what they really mean. Listening? All right then. Your basic everyday symbol, like '1' and '0' and such, your basic symbol means exactly **SQUAT**!

That's right: zilch, zip, nada, noppes, dick, and all the other synonyms you can think of that mean 'nothing'. In and of themselves, symbols have no meaning; rather, meaning *is imposed* on them by us humans. Symbols are means of communication. There's a lot of stuff in the world—objects, people, feelings, actions—and we label these things with symbols to tell them apart. The symbols themselves mean nothing; they're just *representations*, labels that we can make up as we see fit. Unfortunately, this is something that is rarely mentioned when you're growing up, the only thing

they tell you is which symbol is tied to which concept, which can lead people to confuse the thing itself with its representation. There are people that do just this, but still realise that the symbols are just social constructs, and start believing that the things they represent, stuff like gravity and the number π, are just social constructs too. (Yet these same people aren't willing to demonstrate this by, say, stepping out of a window on the 22nd floor.)

As a simple example of symbol(s), consider the word "chair". The word itself has no intrinsic relationship with a "piece of furniture for one person to sit on, having a back and, usually, four legs." (Webster's); it's just handy to have a word for such an object so that we know what we're talking about in a conversation. Obviously, this only works if all parties in the conversation use the same words for the same objects, so at some point in the past a couple of guys got together and decided on a set of words and called it the English language. Since words are just symbols with no intrinsic meaning, different groups can and have come up with a different set of words.

Such an agreement between people for the sake of convenience is called a ***convention*** (basically, a fancy word for standard). Conventions can be found everywhere. That's part of the problem: they are so ubiquitous that they're usually taken for granted. At some point in time, a convention has become so normal that people forget that it was merely an agreement made to facilitate communication, and will attach real meaning to the thing convened upon: the convention is now a "tradition".

Back to numbers. Numbers are used for two things: quantities and identifications (cardinal and ordinal numbers, respectively). It's primarily quantities we're concerned with here: one banana, two bananas, three bananas, that sort of thing. The way numbers are written down—represented by symbols—is merely a convention; for most people, it's probably even a tradition. There are a couple of different ways to represent numbers: by words (one, two, three, four, five) by carvings (*I, II, III, IIII, ⅢⅢ*), Roman numerals (I, II, III, IV, V). You have all seen these at some point or another. The system most commonly used, however, is a variant of what's called the ***base-N positional system***.

The Base-*N* Positional System

"So, Mike, what is the base-n positional system?" Well, it's probably the most convenient system to use when you have to write down long numbers and/or do arithmetic! The basic idea is that you have *N* symbols—numerals—at your disposal, for 0 up to *N*−1, and you represent each possible number by a *string* of *m* numerals. The numeral at position *i* in the string, a_i, is a multiplier of the *i*-th power of the base number. The complete number *S* is the sum of the product of the powers N^i and their multipliers a_i.

(A.1)	$S = \Sigma a_i N^i$

Another way of thinking about the system is by looking at these numbers as a set of counters, like old-style odometers in cars and old cassette players. Here you have a number of revolving wheels with *N* numerals on each. Each wheel is set so that they will increment the counter before it after a revolution has been completed. You start with all zeros, and then begin to turn the last wheel. After *N* numbers have passed, you will have a full revolution: this counter will be back to zero, and the one next to it will increase by one. And again after *N* more counts, and after N^2 the second counter will be full as well and so a third counter will increase, etc, etc.

Here's an example using the familiar case of *N* is ten: the decimal system. Base-ten means ten different symbols (digits): 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. Note that the form of these symbols is arbitrary, but this is how we got/stole them from the Arabs centuries ago. Note also the zero symbol. The zero is one of the key discoveries in mathematics, and makes the positional system possible. Now, for our sample string of numerals, consider "1025", which is to be read as:

1025_{ten}	=	$1 \cdot 10^3_{ten}$	+	$0 \cdot 10^2_{ten}$	+	$2 \cdot 10^1_{ten}$	+	$5 \cdot 10^0_{ten}$
	=	$1 \cdot 1000_{ten}$	+	$0 \cdot 100_{ten}$	+	$2 \cdot 10_{ten}$	+	$5 \cdot 1$
	=	one thousand twenty five						

You may have noticed I'm using words for numbers a lot of the time. The thing is that if you write the '*N*' in 'base-*N*' in its own base, you will always write 'base-10', because the string "10" *always* denoted the base number. That's kind of the point. To point out which "10" you're talking about, I've followed the usual convention and subscripted it with the word "ten". But because it's a big hassle to subscript every number, I'll use another convention that if the number isn't subscripted, it's a base-ten number. Yes, like everyone has been doing all along, only I've taken the effort of explicitly mentioning the convention.

base-2: binary

What you have to remember is that there's nothing special about using 10 (that is, ten) as the base number; it could have just as well been 2 (binary), 8 (octal), 16 (hexadecimal). True story: in the later decades of the 18th century, when the French were developing the metric system to standardize, well, everything, there were also proposals for going to a duodecimal (base-12) system, because of its many factors. The only reason base-ten is popular is because humans have ten fingers, and that's all there is to it.

As an example, let's look at the binary (base-2) system. This system is kinda special in that it is the simplest base-*N* system, using only two numbers 0 and 1. It is also perfect for clear-cut choices: on/off, black/white, high/low. This makes it ideal for computer-systems and since we're programmers here, you'd better know something about binary.

As said, you only have two symbols (BInary digiTs, or bits) here: 0 and 1. In the decimal system, you have ten symbols before you have to add a new numeral to the string: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. But in a binary system you'll already need a second numeral for two: 0, 1, 10 (with two represented by '10'). This means that you get large strings fairly quickly. For example, let's look the number 1025 again. To write this down in binary we have to find the multipliers for the powers of two that will add up to 1025. First, of course, we need the powers of two themselves. The first 11 are:

exponent	binary	decimal

0	1	1
1	10	2
2	100	4
3	1000	8
4	1,0000	16
5	10,0000	32
6	100,0000	64
7	1000,0000	128
8	1,0000,0000	256
9	10,0000,0000	512
10	100,0000,0000	1024

Table A.1: powers of two

As you can see, the length of binary numbers rises really quickly. With longer numbers it's often difficult to see the actual size of the critter, so I comma-separated them into numeral groups of 4. If you're serious about programming, you *need* to know the powers of two, preferably up to 16. The nice thing about binary is that you won't have to worry much about the multiplication factors of the powers, as the only possibilities are 0 and 1. This makes decimal⇔binary conversions relatively easy. For 1025, it is:

1025_{ten}	=	1024	+	1
	=	2^{10}	+	2^0
	=	$100,0000,0001_{bin}$		

An interesting and completely fortuitous factoid about binary is that 2^{10}=1024 is almost 10^3=1000. Because of this, you will often find powers of 1024 indicated by metric prefixes: kilo-, mega-, giga- etc. The correspondence isn't perfect, of course, but it is a good approximate. It also gives salesmen a good swindling angle: since in the computer world powers of 2 reign supreme, one Megabyte (MB) is 1.05 bytes, but with some justification you could also use the traditional 1M = one million in memory

sizes, and thus make it *seem* that your product has 5% more memory. You will also see both notations used randomly in Windows programs, and it's almost impossible to see whether or not that file that Explorer says is 1.4MB will fit on your floppy disk or not.

For this reason, in 1999, the IEC began to recommend a separate set of binary prefixed units based on powers of 1024. These include kibibyte (KiB) for 1024 bytes, mebibyte (MiB) for 1048576 bytes, and gibibyte (GiB) for 1073741824 bytes.

base-16, hexadecimal

In itself, binary isn't so difficult, it's just that the numbers are so large! The solution for this given above was using commas to divide them into groups of four. There is a better solution, namely hexadecimal.

Hexadecimal is the name for the base-16 system, also known as *hex*. That an abbreviation exists should tell you something about its prevalence. As you should be able to guess by now, there are 16 symbols in hex. This presents a small problem because we only have 10 symbols associated with numbers. Rather than invent new symbols, the first letters of the alphabet are used, so the sequence becomes: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. Hex is more concise than binary. In fact, since 16 is 2^4, you can exactly fit four bits into one hex digit, so hex is exactly 4 times as short as binary. This is also why I used groups of four earlier on. If you know the powers of 2, then you automatically know the powers of 16 too, but rather than decompose numbers into powers of 16, it's often easier to go to binary first, make groups and convert those to hex.

1025_{ten}	$=$	$100,0000,0001_{bin}$		
	$=$	$401_{bin}{\cdot}16^2$	$+$	$1{\cdot}16^0$
	$=$	401_{hex}		

A hexadecimal digit is often called a ***nybble*** or a nibble, which fits in nicely with the bit and the byte. Speaking of bytes, bytes are conventionally made up of 8 bits, and

hence 2 nybbles. So you can conveniently write down bytes and multiple byte types in nybbles. My personal preference in dealing with hex numbers in to always use an even number of nybbles, to correspond with the whole bytes, but that's just me. Hexadecimal is so engrained in the computer world that it not only has an abbreviation, but also a number of shorthand notations indicating numbers are indeed hex: C uses the prefix `0x` , in assembly you might find `\$` , and in normal text the affix `h` is sometimes used.

Depending on how low-level you do your programming, you will see any of the three systems mentioned above. Aside from decimal, binary and hexadecimal, you might also encounter octal (C prefix `0`) from time to time. Now, even if you know never intend to use octal, you might use it accidentally. If you would like to align your columns of numbers by padding them with zeros, you are actually converting them to octal! Yet one more of those fiendish little bugs that will have you tearing your hair out.

Using the positional system

Using a base-*N* positional system has a number of advantages over the other number systems. For starters, numbers don't get nearly as long as the carving system; and you don't have to invent new symbols for higher numbers, like in the Roman system. It's also easier to compare two numbers using either the lengths of the strings or just the first number. There's also a tie with probability theory: each individual digit has *N* possibilities, so a number-string with length *m* has N^m possibilities.

Table A.2: counting to twenty in decimal, binary, hex and octal. Note the alternating sequences in the binary column.

dec	bin	hex	o
0	0	0	
1	1	1	
2	10	2	
3	11	3	
4	100	4	
5	101	5	
6	110	6	
7	111	7	
8	1000	8	
9	1001	9	
10	1010	a	
11	1011	b	
12	1100	c	
13	1101	d	
14	1110	e	
15	1111	f	
16	10000	10	
17	10001	11	
18	10010	12	
19	10011	13	
20	10100	14	

Where it really comes into its own is arithmetic. The positions in a number-string are equivalent, so the steps for adding '3+4' are the same for '30+40'. This will allow you to break up large calculations into smaller, easier ones. If you can do calculations for single-symbol numbers, you can do them all. What's more, the steps themselves are the same, regardless of which base you use. I won't show you how to do addition in binary or hex, as that's rather trivial, but I will demonstrate multiplication. Here's an example of calculating '123 × 456', in decimal and hexadecimal. I've also given the multiplication tables for convenience.

Table A.3a: decimal multiplication table

x	1	2	3	4	5	6	7	8	9	10
1	1	2	3	4	5	6	7	8	9	10
2	2	4	6	8	10	12	14	16	18	20
3	3	6	9	12	15	18	21	24	27	30
4	4	8	12	16	20	24	28	32	36	40
5	5	10	15	20	25	30	35	40	45	50
6	6	12	18	24	30	36	42	48	54	60
7	7	14	21	28	35	42	49	56	63	70
8	8	16	24	32	40	48	56	64	72	80
9	9	18	27	36	45	54	63	72	81	90
10	10	20	30	40	50	60	70	80	90	100

Table A.3b: hex multiplication table

x	1	2	3	4	5	6	7	8	9	A	B	C
1	1	2	3	4	5	6	7	8	9	A	B	C
2	2	4	6	8	A	C	E	10	12	14	16	18
3	3	6	9	C	F	12	15	18	1B	1E	21	24
4	4	8	C	10	14	18	1C	20	24	28	2C	30
5	5	A	F	14	19	1E	23	28	2D	32	37	3C
6	6	C	12	18	1E	24	2A	30	36	3C	42	48
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54

8	8	10	18	20	28	30	38	40	48	50	58	60
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C
A	A	14	1E	28	32	3C	46	50	5A	64	6E	78
B	B	16	21	2C	37	42	4D	58	63	6E	79	84
C	C	18	24	30	3C	48	54	60	6C	78	84	90
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8
F	F	1E	2D	3C	4B	5A	69	78	87	96	A5	B4
10	10	20	30	40	50	60	70	80	90	A0	B0	C0

123 × 456, base ten

×	100	20	3	sum
400	40000	8000	1200	49200
50	5000	1000	150	6150
6	600	120	18	738
Result				56088

123 × 456, base 16

×	100	20	3
400	40000	8000	C0
50	5000	A00	F0
6	600	c0	12
Result			

In both cases, I followed exactly the same procedure: break up the big numbers into powers of N, lookup the individual multiplications in the tables and stick the right number of zeros behind them, and then add them all up. You can check with a calculator to see that these numbers are correct. Hexadecimal arithmetic isn't any harder than decimal; it just *seems* harder because they haven't drilled it into your brain at a young age.

I should point out that $4EDC2_{sixteen}$ is actually 323010_{ten}, and not 56088_{ten}. And it shouldn't be, because the second multiplication was *all* in hex: $123_{sixteen} \times 456_{sixteen}$, which actually corresponds to $291_{ten} \times 1110_{ten}$. This is why implicit conventions can cause trouble: in different conventions, the *same* number-string can mean completely *different* things. Please keep that in mind. (Incidentally, facts like this also disprove that mental virus known as numerology. Of course, it doesn't in the eyes of its

adherents, because that's one of the characteristics of belief systems: belief actually grows as evidence mounts against them, instead of diminishing it.)

Look, it floats!

Something that is only possible in a positional system is the use of a floating point. Each numeral in a number-string represents a multiplier for a power of N, but why use only positive powers? Negative powers of x are successive multiplications of $1/x$: $x^{-n} = (1/x)^n$. For example, π can be broken down like this:

exp	3	2	1	**0**	-1	-2	-3	-4	...
pow	1000	100	10	1	$\frac{1}{10}$	$\frac{1}{100}$	$\frac{1}{1000}$	$\frac{1}{10000}$	...
π	0	0	0	**3**	1	4	1	6	...

You can't simply use a number-string for this; you need to know where the negative powers start. This is done with a period: $\pi \approx 3.1416$. At least, the English community uses the period; over here in the Netherlands, people use a comma. That's yet another one of those convention mismatches, one that can *seriously* mess up your spreadsheets.

Since each base-N system is equivalent, you can do this just as well in binary. π in binary is:

exp	3	2	1	**0**	-1	-2	-3	-4	...
pow	8	4	2	1	$\frac{1}{2}$	$\frac{1}{4}$	$\frac{1}{8}$	$\frac{1}{16}$	...
π	0	0	1	**1**	0	0	1	0	...

So π in binary is 11.0010_{two}. Well, yes and no. Unfortunately, 11.0010_{two} is actually 3.1250, not 3.1416. The problem here is that with 4 bits you can only get a precision to the closest $1/16 = 0.0625$. For 4 decimals of accuracy you'd need about 12 bits ($11.001001000100 \approx 3.1416$). You could also use hex instead of binary, in which case the number is $3.243F_{sixteen}$.

Conversion between bases

You might wonder how I got these conversions. It's actually not that hard: all you have to do is divide by the base number and strip off the remainders until you have nothing left; the string of the remainders is the converted number. Converting decimal 1110 to hex, for example, would go like this:

num	/ 16	%16
1110	69	6
69	4	5
4	0	4
result:	456h	

This strategy will also work for floating point numbers, but it may be smart to break the number up in an integer and fractional part first. And remember that dividing by a fraction is the same as multiplying by its reciprocal. Grab your calculator and try it.

There are actually a number of different ways you can convert between bases. The one given using divisions is the easiest one to program, but probably also the slowest. This is especially true for the GBA, which has no hardware division. You can read about another strategy in "Binary to Decimal Conversion in Limited Precision" by Douglas W. Jones.

Scientific notation

Another thing that a positional system is useful for is what is known as the *scientific notation* of numbers. This will help you get rid of all the excess zeros that plague big and large numbers, as well as indicate the number of significant figures. For example, if you look in science books, you might read that the mass of the Earth is 5,974,200,000,000,000,000,000,000 kg. There are two things wrong with this number. First, the value itself is incorrect: it isn't 59742 followed by 20 zeros kilograms, right down to the last digit: that kind of accuracy just isn't possible in physics (with the possible exception of Quantum Mechanics, where theory can be accurate to up to a

staggering 14 decimals. That's right, that 'fuzzy' stuff actually has the highest degree of accuracy of *all* fields of science). When it comes to planetary masses, the first 3 to 5 numbers may be accurate, the rest is usually junk. The second problem is more obvious: the number is just too damn long to write!

The scientific notation solves both problems. Multiplying with a power of 10 effectively moves the floating point around and thus can rid you of the zeros. The mass of the Earth can then be written concisely as $5.9742 \cdot 10^{24}$, that is, 5.9742 times 10 to the power 24. You can also come across the even shorter notation of 5.9742e+24, where the "·10^" is replaced by an 'e' for exponent. Don't misread it as a hexadecimal number. And yes, I am aware that this is a shorthand notation of a shorthand notation. What can I say, math people are lazy bastards. Additionally, this number also indicates that you have 5 significant digits, and any calculation you do afterwards needs to respect that.

Of course, this notation will work for any base number, just remember that conversion between bases require the whole number.

It ain't as hard as you think

The concepts laid out in this section may seem difficult, but I assure you they are actually quite easy to understand. All of this stuff is taught in elementary or high school; the only thing is that they only use the decimal system there. Like I said, the workings of the positional system is equivalent for all base numbers, the only difference is that you've had lots and lots of *practice* with the decimal system, and hardly any with the others. If you had memorised the multiplication tables in hex instead of in decimal, you'd have found the latter awkward to use.

Of bits and bytes

Any self-respecting programmer knows that the workings of a computer are all about little switches that can be on or off. This means that computers are more suited to a binary (or maybe hex) representation than a decimal one. Each switch is called a **bit**; computer memory is basically a sea of millions upon millions of bits. To make things a little more manageable, bits are often grouped into **bytes**. 1 byte = 8 bits is the standard nowadays, but some older systems had 6-, 7-, or 9-bit bytes.

Since *everything* is just 1s and 0s, computers are the best example on the meaning of symbols: it's all about interpretation here. The bits can be used to mean anything: besides switches and numbers you can interpret them as letters, colors, sound, you name it. In this section, I will explain a few ways that you can interpret bits. I will often use a mixture of binary and hex, switching between them for convenience.

Integer number representations

An obvious use of bits would be numbers, especially integers. With 8 bits, you have 2^8=256 different numbers running from 0 to $1111,1111_{two}$ (FFh in hex and 255 decimal). That's not much, so there are also groupings of 16 bits (10000h or 65536 numbers) and 32 bits (10000:0000h or 4,294,967,296 numbers). In the late 2000s decade, PCs made the transition to 64 bits CPUs; I'm not even going to write down how much that is. The C types for these are `short` (16 bits), `int` or `long` (32 bits), and `long long` (64 bits). The size of an `int` or `long` is actually system dependent, but on a GBA, both are 32 bits.

Negative numbers

That you have *n* bits to represent a number does not necessarily mean that you have to use them for the range $[0, 2^n-1]$, that is, positive integers. What about negative numbers? Well, there are a number of ways you can represent negative numbers. A very simple way could be to use one of the bits as a **sign bit**: 0 for positive numbers

and 1 for negative numbers. For example, binary 1000,0001 could be '−1'. Some systems use this, but the GBA doesn't, because there's a smarter way.

Let's bring out our odometers again. In an three-digit odometer, you could go from 0 to 999. Forget what a three-digit odometer says about the quality of the car, just focus on the numbers. At 999, *every* digit will roll over and you'll be back at 0 again. You could also argue that the number *before* 0 is 999. In other words, '999' would be the representation of −1. You could split the full one thousand range into one half for the first positive five hundred (0 to 499), and the other for the first negative five hundred (−500 to −1), as counting backward from 0, using the roll-over. This type of numbering is called be ***ten's complement***. The table below shows how this works for 3 digits.

Number	-500	-499	-498	...	-2	-1	0	1	...	4!
Representation	500	501	502	...	998	999	0	1	...	4!

Table A.4: ten's complement for 3 digits

That's the practice, now the theory behind it. Negative numbers are deeply tied to subtraction; you could almost consider it part of their definition. Basically, for every number x, the following should be true:

$$(A.2) \qquad 0 = {}^*x^* + (-{}^*x^*)$$

This could be considered zeros' complement: the number $(-x)$ is the number you need to add to x to get 0. In ten's complement, they need to add up to 10 or a power of 10. In our odometer, 1000 will have the same representation as 0, and counting back from one thousand will be the same as counting back from zero. However, you *must* know the number of digits beforehand; otherwise it won't work. The actual representation of $-x$ using m digits, can be derived as follows:

$$(A.3) \qquad \begin{aligned} 0 &= x + (-x) \\ &\cong \\ 10^m &= x + (-x) \\ (10^m - 1) - x + 1 &= -x \end{aligned}$$

Don't panic, these equations aren't as nasty as they seem. Remember that for m digits, the highest number is $10^m - 1$. If $m = 3$, then that'd be 999 in decimal, 111 in binary or FFF in hex. This will allow you to do the subtraction by x without borrowing. There's more to 10's complement then a way to express negative numbers; it'll also turn subtraction into a form of addition: *subtraction* by y is equivalent to *addition* by its 10's complement. That feature was part of the system from the start, and the other schemes of negative number representations don't have that property. Checking this is left as an exercise for the reader.

The binary version of 10's complement is ***two's complement***. Finding the two's complement of a number is actually easier than in other cases: the subtraction of $10^m - 1$ by x is just the inversion of all the bits of x. Take 76, for example:

255:	1111 1111	
76:	0100 1100	−
179:	1011 0011	

The 8bit −76 would be 179+1=180 (10110100_{two}) and you will indeed see that 180+76 = 256 = 2^8, exactly as it should be.

Signed is not unsigned

I've already mentioned this before, but it's important enough to state it again: when using 10's complement, you *must* know the number of digits ahead of time, otherwise you won't know what to subtract x from. Most of the time you can remain blissfully ignorant of this fact, but there are a few instances where it really does matter. In C or assembly programming, you have two types of integer numbers: ***signed*** and ***unsigned***, and only the *signed* types are in two's complement. The difference manifests itself in the interpretation of the most significant bit: in unsigned numbers, it's just another bit. But in signed numbers, it acts as a sign-bit, and as such it needs to be preserved in certain operations as *type-casting* or *shifting*. For example, an 8-bit

$FF_{sixteen}$ is a signed '−1' or an unsigned '255'. When converting to 16 bits, the former should become $FFFF_{sixteen}$, while the latter would remain $00FF_{sixteen}$. If you ever see stuff go completely bonkers when numbers become negative, this might be why.

Here are a few guidelines for choosing signed or unsigned types. Intrinsically signed types are numbers that have a physical counterpart: position, velocity, that kind of stuff. A key feature of these is that you're supposed to do arithmetic on them. Variables that act as switches are usually unsigned, the bitflags for enabling features on a GBA are primary examples. These usually use logical operations like masking and inverting (see the section on bit operations). Then there are quantities and counters. These can be either signed or unsigned, but consider starting with signed, then switch to unsigned if you really have to. Again, these are just recommendations, not commandments that will land you in eternal damnation if you break them.

Unsigned and signed types can behave differently under type casting, comparison and bit-operations. A byte *x* containing FFh could mean a signed −1 or an unsigned 255. In that case:

FFh	signed	unsigned
comparison x<0	true	false
conversion to 16 bit	FFFFh (-1)	00FFh (255)
shift right by 1	FFh (-1)	7Fh (127)

Characters

No, I'm not talking about GBA-tiles, but the letter variety (this possible confusion is why I'm not fond of the name 'character' for tiles). For everyday purposes you would need 2×26 letters, 10 numerals, a bunch of punctuation signs and maybe a few extra things on the side: that's about 70 characters at least, so you'd need 7 bits to indicate them all (6 would only allow 2^6=64 characters). Better make it 8 bits for possible future expansion, and because it's a nice round number. In binary that is. That's part of the reason why the byte is a handy grouping: one character per byte.

ASCII

Knowing which characters you need is only part of the story: you also need to assign them to certain numbers. The order of any alphabet is, again, just a convention (well, there are orders that are more logical than others, see Tolkien's *Tengwar*, "The Lord of the Rings", Appendix E, but the Latin alphabet is completely random). One possible arrangement is to take a normal keyboard and work your way through the keys. Fortunately, this isn't the standard. The common code for character assignments is *ASCII*: *American Standard Code for Information Interchange*.

The lower 128 characters of ASCII are given below. The first 32 are control codes. Only a few of these are still of any importance: 08h (backspace, `\b`), 09h (tab, `\t`), 0Ah (Line Feed, `\n`) and 0Dh (Carriage Return, `\r`). If you have ever downloaded text files from Unix/Linux servers, you might have noticed that all the line breaks have been removed: this is because CP/M, MS-DOS, and Windows use CRLF (`\r\n`) as the line break, while Unix environments just use the line feed.

The real characters start at 20h, the space character. Note how the numeric, uppercase and lowercase characters are located sequentially and in a logical fashion. Numbers start at 30h, uppercase at 41h, lowercase at 61h. The alphabetical order of the letters makes for easy alphabetizing, although I should point out that the 32 difference between uppercase and lowercase may cause problems.

The ASCII set also has an upper 128 characters, but these can be different for different language settings. Normally, these will include accented characters that are frequent in non-English languages. In a DOS environment, they also contained a number of purely graphical characters for borders and the like. ASCII isn't the only character set available. Chinese and Japanese languages usually use the 16bit *Unicode*, as the 8bit ASCII simply isn't sufficient for thousands of characters. ASCII is basically a subset of Unicode.

The C type for the character is called *char*. A **char** is actually a *signed* 8bit integer. I mention this because I distinctly remember being sent on a long bughunt long ago

because of this little fact. To be perfectly honest, I think that the default signing of the char-type is actually platform dependent, so consider yourself warned.

dec	hex	Char	dec	hex	Char	dec	hex	Char	dec
0	00h	NUL	32	20h	sp	64	40h	@	96
1	01h		33	21h	!	65	41h	A	97
2	02h		34	22h	"	66	42h	B	98
3	03h		35	23h	#	67	43h	C	99
4	04h		36	24h	$	68	44h	D	100
5	05h		37	25h	%	69	45h	E	101
6	06h	ACK	38	26h	&	70	46h	F	102
7	07h	BELL	39	27h	'	71	47h	G	103
8	08h	BS	40	28h	(	72	48h	H	104
9	09h	HT	41	29h	)	73	49h	I	105
10	0Ah	LF	42	2Ah	*	74	4Ah	J	106
11	0Bh		43	2Bh	+	75	4Bh	K	107
12	0Ch	FF	44	2Ch	,	76	4Ch	L	108
13	0Dh	CR	45	2Dh	-	77	4Dh	M	109
14	0Eh		46	2Eh	.	78	4Eh	N	110
15	0Fh		47	2Fh	/	79	4Fh	O	111
16	10h		48	30h	0	80	50h	P	112
17	11h		49	31h	1	81	51h	Q	113
18	12h		50	32h	2	82	52h	R	114
19	13h		51	33h	3	83	53h	S	115
20	14h		52	34h	4	84	54h	T	116
21	15h		53	35h	5	85	55h	U	117
22	16h		54	36h	6	86	56h	V	118
23	17h		55	37h	7	87	57h	W	119

24	18h		56	38h	8	88	58h	X	120
25	19h		57	39h	9	89	59h	Y	121
26	1Ah	^Z	58	3Ah	:	90	5Ah	Z	122
27	1Bh	ESC	59	3Bh	;	91	5Bh	[	123
28	1Ch		60	3Ch	<	92	5Ch	\	124
29	1Dh		61	3Dh	=	93	5Dh	]	125
30	1Eh		62	3Eh	>	94	5Eh	^	126
31	1Fh		63	3Fh	?	95	5Fh	_	127

Table A.5: ASCII 0-127

IEEE(k)! Floating points

The last of the most common types is the floating point. Having, say, 32bits for a number is nice and all, but it still means you are limited to around 4 billion characters. This may seem like a big number, but we've already seen numbers that are much bigger. The floating-point types provide a solution, using the scientific notation in binary. I already described floating point numbers (even in binary), as well as the scientific notation, so I won't repeat how they work.

Describing floating-point numbers on a computer is done according to the *IEEE/ANSI* standard (Institute of Electrical and Electronic Engineers / American National Standards Institute). The floating-point format consists of 3 parts, a sign bit s, an exponent e and a fractional part f. The following table and equation is the formatting and meaning of a normal, 32bit float

```
IEEE format for 32bit float
```

1F	1E 1D 1C 1B 1A 19 18 17	16 15 14 13 12 11 10 F E D C B A 9 8 7 6
s	e	f

bits	name	description
00-16	f	**Fractional** part (23 bits)

17-1E	e	Exponent (8 bits)
1F	s	Sign bit.

$$\text{(A.4)} \qquad x = (-1)^s \cdot 1.f \cdot 2^{e-127}$$

Note that unlike signed integers, there *is* a real sign bit this time. Furthermore, the number always starts with 1, and the fractional part *f* really is the fractional part of the number. This makes sense, because sense, since if it weren't, you can always move the point around until you get a single 1 before the point. The exponent is subtracted by 127 to allow for negative powers (similar, but not exactly like you'd get in a 2s' complement number). Two examples:

x	s	e	f
1.0	0	01111111	000 0000 0000 0000 0000 0000
−1.0	1	01111111	000 0000 0000 0000 0000 0000

Eq 4 will hold for the usual causes, but there are a few exceptions to this rule.

- If **e = f = 0**, then $x = 0$. Note that the sign-bit can still be set to indicate a left-limit to zero.

- If **e = 0** and **f ≠ 0**, then the number is too small to be normalized, $x = (-1)^S \times 0.f \times 2^{-127}$

- If **e = 255** and **f = 0**, then the $x = +\infty$ or $x = -\infty$

- If **e = 255** and **f ≠ 0**, then $x =$ NaN, or *Not a Number*. $\sqrt{-1}$ would be NaN, for example.

The 32bit **float** has a 23bit fractional part, meaning 24 bits of precision. Each 10 bits mean roughly one decimal, so that 24 bits give around 7 decimals of precision, which may or may not be enough for your purposes. If you need more, there are also the 8 byte **double** and and 10 byte **long double** types, which have more exponent and fractional bits.

As you can probably tell, the floating-point format isn't nearly as easy to understand as an integer. Both arithmetic and int-float conversion is tricky. This isn't just for us humans, but computers can have a hard time with them too. PCs usually have a separate floating-point unit (FPU) for just these numbers. The GBA, however, does not. As such, the use of floating-point numbers is *strongly* discouraged on this system. So does that mean that, if we want to use fractions and decimals and such, we're screwed? No, the solution to this particular problem is called fixed-point math, and I'll explain that here.

AAaagghhh! The endians are coming!

There is one convention I have completely overlooked throughout this chapter: *endianness*. This is about the reading order numbers, bits and bytes. I have always just assumed that in a number, the *left*most digit is the most significant number, that is, the highest power of *N*. So 1025 is read as one thousand twenty-five. That's **big-endian**, so named because the big-end (the highest power) goes first. There is also **little-endian**, in which the little-end (lowest power) goes first. In that case, 1025 would be read as five thousand two hundred and one. Once again, it's a trivial convention, but it matters greatly which one you use. Both have their merits: speech is usually big-endian and our number system reflects that (except in a few countries which place the ones before the tens (five and twenty), which can be quite confusing). Arithmetic, however, usually starts at the little-end, as do URLs.

Computer endianness plays a part in two areas: bit-order in a byte and byte-order in a multi-byte type such as an int. Since the byte is usually the smallest chunk you can handle, the bit-order is usually of little concern. As a simple example, look at the int 0x11223344. This will be stored differently on different systems, see the table below. Try to think of what would happen if you save this in a file and then transfer that to a computer with a different endian-scheme.

memory	00	01	02	03
big	11	22	33	44
little	44	33	22	11

Table A.6: storing 0x11223344

So what should we use then? Well, that's just it: there is no real answer. A benefit of big-endian is that if we see a memory dump, the numbers will be in the human reading-order. On the little-endian side, lower powers are in lower memory, which makes more sense mathematically. Additionally, when you have a 16bit integer x = 0x0012, when you cast its address to a 8bit pointer, the value will be preserved which, personally, I think is a good thing.

```
u8 *pc;
short i= 0x0012;
pc= (u8*)&i;
// little endian: *pc = 0x12, fine
//    big endian: *pc = 0x00, whups
```

There is actually one place where you can see the bits-in-byte order: bitmaps. In particular, bitmaps with a bit depth less than 8. A byte in a 4bpp bitmap will represent two pixels. In a BMP, the high-nybbles are the even pixels and low-nybbles the odd ones. GBA graphics work exactly the other way around. One could say that BMP bits are big-endian and GBA bits are little-endian (*bytes*, however, are little-endian on both PCs and GBA). Another endianness-related thing about bitmaps is the color order, RGB (red-green-blue), or BGR (blue-green-red). There are so many pitfalls here that I don't even want to get into this.

Interestingly, there's one other field where endianness mucks things up: dates. In Europe we use a little-endian scheme: day-month-year. China, Japan, and the ISO 8601 standard use big-endian dates: year-month-day. And then there's the American English scheme, which just had to make things difficult for themselves by using a month-day-year scheme. This could be called middle endian, I suppose.

In the end it's not a matter of which is 'better', but rather of which system you're working on. PCs and the GBA are little-endian; I hear that PowerPC Macs and a lot of other RISC chips are big-endian (but I may be wrong here). Don't get dragged into any holy wars over this, just be aware that the different schemes exist and be careful when porting code.

Bit operations

As the name implies, bit operations (bit-ops) work at the individual bit level and are therefore the lowest operations you can think of. Most Real World applications have little need for bit-fiddling and therefore use bit-ops sparingly, if at all. A good number of programming languages don't even have them. Assembly and C (and Java) belong to the ones that do, but if you look at course books, bit operations are usually moved to the back pages (yes, I am aware that I'm doing this too, but remember that Tonc isn't meant as a general programming tutorial; you should know this stuff already. Most of it, anyway). As GBA programming is done very close to the hardware, with effects taking place depending on whether individual bits are set (1) or clear (0), a good understanding of bit operations is *essential*!

The basic list of bit-ops is: OR, AND, NOT, XOR, shift left/right, rotate left/right. That's 8 operations, though someone proficient with Occam's Razor could cut this list down to 5, perhaps even four items. Of these, only OR, AND and XOR are 'true' bit operations: they can be used to change the value of a single bit. The rest change all the bits of a variable.

True bitwise bit operations

There are 3 bitwise operators: OR ((inclusive or, symbol '&'), AND (symbol '|') and XOR (exclusive or, symbol '^')). These are binary operators, as in 'taking two arguments as their inputs'. They're called *bitwise* operators because that the nth bit of the result is only affected by the nth bits of the operands. AND and OR work pretty much as their logical counterparts (&& and ||). In *c=a&b*, a bit in *c* will be `1` only if that bit is `1` in both *a and b*. For OR, the *a*-bit *or b*-bit (or both) must be `1` . XOR doesn't have a logical counterpart, but it is more closely linked to the Real Word definition of 'or': XOR is `1` if *either* the *a*-bit *or* the *b*-bit is `1` (but not both).

There is a fourth operation that is often included in this group, namely NOT (ones' complement, symbol '~). NOT is a unary operator, and inverts all bits of the operand, which is basically XORring with −1 (which is all `1` s in binary). The bitwise NOT is

~~similar to the logical not ('!'). There is an important difference between the logical operations ('&&', '||' and '!') and their bitwise counterparts ('&', '|', '')~~, try not to confuse them.

What these four operations do is usually written down in truth tables, which list all possible input combinations and their results. Note that the truth tables look at each bit individually, not the variable as a whole, even though the operators themselves always act on variables. Table 8 shows examples of these operators on bytes 0Fh and 35h.

a b	a&b	a|b	a^b		a	~a
0 0	0	0	0		0	1
0 1	0	1	1		1	0
1 0	0	1	1			
1 1	1	1	0			

Table A.7: bit operations

Table A.8a: bit-ops examples

AND				OR				XOR		
0Fh	00001111			0Fh	00001111			0Fh	00001111
35h	00110101	&		35h	00110101	|		35h	00110101
05h	00000101			3Fh	00111111			3Ah	00111010

I hope you've noticed that some of the bits were colored. Yes, there was a point to this. Knowing what the bit-ops do is one thing; knowing how to *use* them is another. A bit is a binary switch, and there are four things you can do to a switch: leave it alone, flip it, turn it on, and turn it off. In other words, you can:

- **keep** the current state,
- **toggle** it (0→1, 1→0),
- **set** it (x→1), and
- **clear** it (x→0)

If you look at the truth tables and the examples, you may already see how this can work. OR, AND, XOR are binary operators, and you can think of the two operands as a source variable *s* and a **mask** variable *m* which tells you which of the bits are affected. In table 8a I used *s*=35h and *m*=0Fh; the mask consists of the set bits (in blue), the red bits were the ones that were affected. If you examine the table, you'll see that an OR sets bits, a XOR toggles it and an AND keeps bits (i.e., clears the unmasked bits). To clear the masked bits, you'd need to invert the mask first, so that would be an *s* AND NOT *m* operation. Note that the first three are commutative (*s* OP *m* = *m* OP *s*), but the last one isn't. This masking interpretation of the bit operations is very useful, since you'll often be using them to change the bits of certain registers in just this way, using C's assignment operators like '|='.

AND (keep bits)			OR (set bits)			XOR (flip bits)		
$s \& m$			$s \mid m$			$s \wedge m$		
35h	00110101		35h	00110101		35h	00110101	
0Fh	00001111	&	0Fh	00001111	|	0Fh	00001111	
05h	00000101		3Fh	00111111		3Ah	00111010	

Table A.8b: bit-ops examples encore, using source *s*=35h and

Non-bitwise bit operations

And then there are the shift and rotate operations. In contrast to the earlier operations, these act on a variable as a whole. Each variable is a string of bits and with the shift and rotate operations you can move the bits around. Both have left and right variants and are binary operations, the first operand is the source number, and the second is the amount of bits to move. I'll refer to shift left/right as SHL and SHR and rotate left/right as ROL and ROR for now. These sound like assembly instructions, but they're not. At least, not ARM assembly. Shift left/right have C operators '<<' and '>>', but there are no C operators for a bit-rotate, although you can construct the effect using shifts. As said, shift and rotate move bits around a variable, in pretty much the way you'd expect:

name	symbol	example	result
shift left	SL, <<	00110101 << 2	11010100 , D4h
shift right	SR, >>	00110101 >> 2	00001101 , 0Dh
rotate left	ROL	00110101 ROL 3	10101001 , A9h
rotate right	ROR	00110101 ROR 3	10100110 , A6h

Table A.9: shift / rotate operations on byte 35h (00110101)

Shifting has two uses. First of all, you can easily find the *n* bit, or the *n*th power of 2 by using `1<< `*n*. Speaking of powers, shifting basically comes down to adding zeros or removing bits, which is essentially multiplying or dividing by 10. Binary 10, that is. So you could use shifting to quickly multiply or divide by 2. The latter is especially useful, since division if very, very costly on a GBA, while shifting is a one-cycle operation. I can't really thing of a use for rotation right now but I'm sure they're there.

OK, that's what they do in theory. In *practice*, however, there's a lot more to it. One thing that is immediately obvious is that the size of the variable is important. A rotate on an 8bit variable will be very different then a rotate on a 16bit one. There is also the possibility of including the carry bit in the rotation, but that doesn't really matter for the moment because bit rotation is purely an assembly matter, and that's beyond the scope of this page.

What does matter is a few nasty things about shifting. Shift-left isn't much of a problem, unless you shift by more than the amount of bits of the variable. Shift-right, however, has one particular nasty issue for negative numbers. For example, an 8bit −2 is represented in twos' complement by `FEh` . If you shift-right by one, you'd get `7Fh` , which is 128, and not −2/2 = −1. The problem here is that the first bit acts as a sign bit, and should have special significance. When shifting- right, the sign-bit needs to be preserved and extended to the other bits, this will ensure that the result is both negative and represents a division by a power of two. There are actually two right-shift instructions, the *arithmetic* and the *logical* shift right (ASR and LSR); the former extends the sign bit, the latter doesn't. In C, the signing of the variable type determines which of these instructions is used.

Take the interesting case of the 8bits 80h, which is both the unsigned 128 as the signed −128. A right-shift by 3 should result in 16 and −16, respectively. This would be 10h for the unsigned and F0h for the signed case, and lo and behold, that is exactly what you'd get by sign-bit extension or not.

type char	unsigned	signed
1000 0000	128	−128
80h>>3	0001 0000	1111 0000
	16	−16

Table A.10: signed and unsigned `80h>>3`

I know this seems like such a small and trivial issue, and indeed, it usually is. But when it isn't, you could be looking at a long bughunt. This isn't limited to just shifting, by the way, *all* bit operations can suffer from this problem.

Arithmetic with bit operations

The shift operators can be used to divide and multiply by powers of two. The other bit-ops also have arithmetic interpretations.

For example, a modulo of a power of two basically cuts away the upper bits, which can be done with an AND operation: $x \% 2^n = x$ AND $2^n - 1$. For example, $x \% 8 = x \& 7$.

An OR operation can be used as an addition, but *only* if the affected bits were 0 to start with. F0h | 01h = F1h, which is the same as F0h+01h. However, F0h | 11h = F1h too, but F0h+11h is actually 101h. Be careful with this one, and make note of it when you see it in other people's code.

Thanks to [two's''s complement](#), we can use XOR as a subtraction: $(2^n - 1) - x = (2^n - 1)$ XOR x. This can be used to reverse the traversal order of loops, for example, which can be useful when you want collision detection with flipped tiles. Yes, it's a bit of a hack, but so what?

```
int ii, mask;

for(ii=0; ii<8; ii++)
{
    // array direction based on mask
    // mask=0 -> 0,1,2,3,4,5,6,7
    // mask=7 -> 7,6,5,4,3,2,1,0
    ... array[ii^mask] ...
}
```

OR and XOR are only very rarely used in their arithmetic form, but the shifts and AND can be seen with some regularity. This is especially true on a system with no hardware division (like the GBA), in which case division and modulo are expensive operations. That is why powers of two are preferred for sizes and such, the faster bit operations can then be used instead. Fortunately, the compiler is smart enough to optimize, say, division by 8 to a right-shift by 3, so you don't have to write down the bit-op version yourself if you don't want to. Mind you, this will only work if a) the second operand is a constant and b) that constant is a power of two.

bit-op	arithmetic function	example
SHL	x<<$n = x * 2^n$	x<<$3 = x * 8$
SHR	x>>$n = x / 2^n$	x>>$3 = x / 8$
AND	x&$(2^n - 1) = x \% 2^n$	x&$7 = x \% 8$

Table A.11 Arithmetic bit-ops summary

And now for my final trick of the day, let's take a closer look at the most basic of arithmetic operations, addition. The addition of 2 bits to be precise, and the truthtable of that can be found in table 12 below. If you've paid attention so far (well done! I didn't think anyone would make it this far :P), there should be something familiar about the two columns that make up the result. The right column is just a XOR b and the left column is a AND b. This means that you can create a 1-bit adder with just an AND and a XOR port, electric components that can be found in any Radio Shack, or its local equivalent. String 8 of these together for an 8-bit adder, and you'll have yourself the foundation of an 8bit computer, cool huh?

a b	a+b
0 0	00
0 1	01
1 0	01
1 1	10

Table A.12: 1–bit adder

Beware of bit operations

There are two things you should *always* remember when you're using bit operations. I've already mentioned the first, that they can mess with the sign of the variables. This is only relevant for signed integers, though.

The second problem is concerns the level of precedence of the bit operations. Except for NOT (~), the precedence is very low; lower than addition, for example, and even lower than conditional operators in some cases. Your C manual should have a precedence list, so I'll refer you to that for details. In the mean time, be prepared to drown your code in parentheses over this.

B. Fixed-Point Numbers and LUTs

- What are fixed-point numbers
- Fixed-point math
- Faking division (optional)
- Look-up Tables

What are fixed-point numbers

Roughly put, there are two types of numbers: integers and floating-points. For most serious math you would get nowhere with integers because, by definition, they don't allow fractions. So for 3D games you'd use floating-point math. Back in the old days, before the arrival of specialized floating-point hardware, that stuff was very slow! Or at least slower than integer arithmetic. Fortunately, there is a way of faking numbers with decimal points with integers. This is known as *fixed-point math*.

General fixed-point numbers

Here's an example. Say you have $10.78 (ten dollars and seventy-eight cents) in wallet. If you want to write this amount as an integer you have a problem, because you'd either have to leave off the fractional part ($10) or round it to $11. However, you could also write it down not in dollars, but in *cents*. That way you'd write 1078, which is an integer, problem solved.

That's the way fixed-point math works. Instead of counting units, you count *fractions*. In the previous example, you count in cents, or hundredths. Fixed-points have a in integer part (the "10"), and a fractional part ("78"). Since we have 2 digits for the fractional part, we call this a fixed-point number in an *x.2* format.

Note that PCs have floating-point units (FPU) since the mid-1990s. This makes floating-point arithmetic just as fast as integer arithmetic (sometimes even faster) so using fixed-point math is not really worth the trouble except, perhaps, in rasterization, since the conversion from `float` s to `int` s is still slow. However, the GBA doesn't do floating-point stuff well, so it's fixed math all the way.

GBA fixed-point usage

Because computers use the binary system, using decimals would be silly as a basis for fixed-points would be silly. Fortunately, you can do fixed-point math in any base, including binary. The basic format is *i.f*, where *i* is number of integer bits, and *f* the number of fractional bits. Often, only the fractional is important to know, so you'll also come across just the indication '.*f*'.

The GBA uses fixed-point math in a number of cases. The affine parameters, for example, are all .8 fixed-point numbers ("***fixeds***", for short). Effectively, this means you're counting in $1/2^8 = 1/256$ths, giving you a 0.004 accuracy. So when you write 256 to a register like REG_BG2PA, this is actually interpreted as 256/256=1.00. REG_BG2PA=512 would be 2.00, 640 is 2.50, et cetera. Of course, it is a little hard to see in the decimal system, but grab a calculator and you'll see that it's true. For this reason, it is often more convenient to write them down as hex numbers: 256=0x100→1.00, 512=0x200→2.00, 640=0x280→2.50 (remember that 8 is 16/2, or one half).

```
// .8 fixed point examples : counting in fractions of 256

int a= 256;         // 256/256 = 1.00
int a= 1 << 8;      // Ditto
int a= 0x100;       // Ditto

int b= 0x200;       // 0x200/256 = 512/256 = 2.00
int c= 0x080;       // 0x080/256 = 128/256 = 0.50
int d= 0x280;       // 0x280/256 = 640/256 = 2.50
```

The affine registers aren't the only places fixed-points are used, though that's where they are the most recognizable. The blend weights are essentially fixed-point

numbers as well, only they are 1.4 fixeds, not .8 fixeds. This is an important point, actually: the position you set the fixed-point to is arbitrary, and you can even switch the position as you go along. Now, the numbers themselves won't tell you where the point is, so it is important to either remember it yourself or better yet, write it down in the comments. Trust me, you do not want to guess at the fixed-point position in the middle of a lengthy algorithm.

> **COMMENT YOUR FIXED-POINT POSITION**
>
> When you use fixed-point variables, try to indicate the fixed-point format for them, especially when you need them for longer calculations, where the point may shift position depending on the operations you use.

Fixed-point and signs

Fixed-point numbers are supposed to be a poor man's replacement for floating-point numbers, which would include negative numbers as well. This means that they're supposed to be *signed*. Or at least, usually. For example, the affine registers use signed 8.8 fixeds, but the blend weights are unsigned 1.4 fixeds. You may think it hardly matters, but signs can really mess things up if you're not careful. Say you're using fixed-points for positions and velocities. Even if your positions are always positive, the velocities won't be, so signed numbers would be more appropriate. Furthermore, if your fixed-point numbers are halfwords, say 8.8 fixeds, a signed '−1' will be used as `0xFFFFFFFF`, i.e. a proper '−1', but an unsigned '−1' is `0x0000FFFF`, which is actually a positive number. You won't be the first person to trip over this, nor would be the last. So signed fixeds, please.

Another point of notice is the way signed fixeds are often indicated. You may see things of the form '1.*n*.*f*'. This is meant to indicate one sign bit, *n* integer bits and *f* fractional bits. Technically speaking, this is **false**. Fixed-point numbers are just plain integers, just interpreted as fractions. That means they follow two's complement and that, while a set top bit does indicate a negative number, it isn't *the* sign bit. As I

mentioned, '−1' in two's complement is `0xFFFFFFFF`, not `0x80000001` as is the case with sign and magnitude. You might not think much of this distinction and that it's obvious that it's still two's complement, but considering that floating-point formats *do* have a separate sign bit, I'd say it's worth remembering.

> **SIGNED FIXED FORMAT NOTATION**
>
> Signed fixed-point formats are sometimes indicated as '1.*n*.*f*'. From that, you might think they have a separate sign bit like floating-point formats, but this is **not correct**. They're still regular integers, using two's complement for negative numbers.

Fixed-point math

Knowing what fixed-point numbers are is one thing, you still have to use them somehow. Three things concern us here.

- Converting between regular integers or floats and fixed-point numbers.
- Arithmetical operations.
- Overflow.

None of these items are difficult to understand, but each does have its awkward issues. In fact, overflow *is* merely an issue, not really an item. This section will focus on 24.8 signed fixeds, for which I will use a "FIXED" typedef'ed int. Although it only uses this fixed-point format, the topics covered here can easily be applied to other formats as well.

Converting to and from fixed-points

I'm not really sure is "conversion" is even the right word here. The only difference between fixed-point numbers and normal ones is a scaling factor *M*. All that's

necessary to go from a FIXED to an int or float is account for that scale by either multiplication or division. Yes, it really is that simple. As we're using power-of-two's for the scales, the integer⇔FIXED conversion can even be done with shifts. You can add the shifts in the code yourself, but the compiler is smart enough to convert power-of-two multiplications and divisions to shifts itself.

```c
typedef s32 FIXED;            //! 32bit FIXED in 24.8 format

// For other fixed formats, change FIX_SHIFT and the rest goes with it.

//! Convert an integer to fixed-point
INLINE FIXED int2fx(int d)
{    return d<<FIX_SHIFT;     }

//! Convert a float to fixed-point
INLINE FIXED float2fx(float f)
{    return (FIXED)(f*FIX_SCALEF);    }


//! Convert a fixed point value to an unsigned integer.
INLINE u32 fx2uint(FIXED fx)
{    return fx>>FIX_SHIFT;    }

//! Get the unsigned fractional part of a fixed point value (orly?).
INLINE u32 fx2ufrac(FIXED fx)
{    return fx&FIX_MASK; }

//! Convert a FIXED point value to an signed integer.
INLINE int fx2int(FIXED fx)
{    return fx/FIX_SCALE;     }

//! Convert a fixed point value to floating point.
INLINE float fx2float(FIXED fx)
{    return fx/FIX_SCALEF;    }
```

Rounding off and negative number inconsistencies

The conversions are almost as simple as described above. The two places where things may be problematic are round-off inconsistencies and negative fractions. Note that I said they *may* be problematic; it depends on what you had in mind. I am not going to explain all the ins and out here, because they generally won't be much of a problem, but you need to be aware of them.

If you're not new to programming, you will undoubtedly be aware of the problem of round-off from floats to ints: a simple cast conversion truncates a number, it does not really round it off. For example, '(int)1.7' gives 1 as a result, not 2. The earlier macros have the same problem (if you can call it that). Float-to-int rounding is done by adding one half (0.5) to the float before rounding, which we can also apply to fixed-point conversion. In this case, of course, the value of one half depends on the number of fixed-point bits. For example, .8 fixeds, ½ is 0x80=128 (256/2), for .16 fixeds it is 0x8000=32768. Add this before shifting down and it'll be rounded off properly. There are actually multiple ways of rounding off, which you can read about in "An Introduction to Fixed Point Math" by Brian Hook.

And then there are negative numbers. Frankly, division on negative integers is always a bitch. The basic problem here is that they are always rounded towards zero: both +3/4 and −3/4 give 0. In some ways this makes sense, but in one way it doesn't: it breaks up the sequence of outputs around zero. This is annoying on its own, but what's worse is that right-shifting *doesn't* follow this behaviour; it always shifts towards negative infinity. In other words, for negative integer division, the division and right-shift operators are *not* the same. Which method you choose is a design consideration on your part. Personally, I'm inclined to go with shifts because they give a more consistent result.

x	-8	-7	-6	-5	-4	-3	-2	-1	0	1	2	3	4
$x/4$	-2	**-1**	**-1**	**-1**	-1	**0**	**0**	**0**	0	0	0	0	1
x>>2	-2	**-2**	**-2**	**-2**	-1	**-1**	**-1**	**-1**	0	0	0	0	1

Table B.1: Division and right-shifts around zero.

The negative division nasty is even worse when you try to deal with the fractional part. Masking with AND effectively destroys the sign of a number. For example, a 8.8 −2¼ is −0x0240 = 0xFDC0. Mask that with 0xFF and you'll get 0xC0 = ¾, a positive number, and the wrong fraction as well. On the other hand 0xFDC0>>8 is −3, for better or for worse, and −3 + ¾ is indeed −2¼, so in that sense it does work out. The question whether or not it works for *you* is something you'll have to decide on your own. If you want to display the fixed numbers somehow (as, say -2.40 in this case), you'll have to

be a little more creative than just shifts and masks. Right now, I'm not even touching that one.

> **CONVERTING NEGATIVE FIXED-POINT NUMBERS**
>
> The conversion from negative fixed-point numbers to integers is a particularly messy affair, complicated by the fact that there are multiple, equally valid solutions. Which one you should choose is up to you. If you can, avoid the possibility; the fixed→int conversion is usually reserved for the final stages of arithmetic and if you can somehow ensure that those numbers will be positive, do so.

Arithmetical operations

Fixed-point numbers are still integers, so they share their arithmetic operations. However, some caution needs to be taken to keep the fixed point in its proper position at times. The process is the same as arithmetic on decimals. For example, 0.01+0.02 = 0.03 ; what you will usually do for this sum is remove the decimal point, leaving 1 and 2, adding those to give 3, and putting the decimal point back. That's essentially how fixed-points work as well. But when adding, say, 0.1 and 0.02, the fixed decimals aren't 1 and 2, but **10** and 2. The key here is that for addition (and subtraction) the point should be in the same place.

A similar thing happens for multiplication and division. Take the multiplication 0.2×0.3. 2×3 equals 6, then put the point back which gives 0.6, right? Well, if you did your homework in pre-school you'll know that the result should actually be 0.06. Not only do the decimals multiply, the *scales* multiply as well.

Both of these items apply to fixed-point arithmetic as well. If you always use the same fixed point, addition and subtractions will pose no problem. For multiplication and division, you'll need to account for extra scaling factor as well. A fixed-fixed multiply required a division by the scale afterwards, whereas a fixed-fixed division needs a scale multiply *before* the division. In both cases, the reason of the place of the scale

correction is to keep the highest accuracy. Equations 1 and 2 show this in a more mathematical form. The fixed-point numbers are always given by a constant times the fixed scale *M*. Addition and subtraction maintain the scale, multiplication and division don't, so you'll have to remove or add a scaling factor, respectively.

$$
\text{(B.1)} \quad
\begin{aligned}
fa &= a \cdot M \\
fb &= b \cdot M
\end{aligned}
$$

$$
\text{(B.2)} \quad
\begin{aligned}
fc &= fa + fb &= (a+b) \cdot M \\
fd &= fa - fb &= (a-b) \cdot M \\
fe &= fa \cdot fb &= (a \cdot b) \cdot M^2 \\
ff &= fa / fb &= a / b
\end{aligned}
$$

```c
//! Add two fixed point values
INLINE FIXED fxadd(FIXED fa, FIXED fb)
{   return fa + fb;          }

//! Subtract two fixed point values
INLINE FIXED fxsub(FIXED fa, FIXED fb)
{   return fa - fb;          }

//! Multiply two fixed point values
INLINE FIXED fxmul(FIXED fa, FIXED fb)
{   return (fa*fb)>>FIX_SHIFT;               }

//! Divide two fixed point values.
INLINE FIXED fxdiv(FIXED fa, FIXED fb)
{   return ((fa)*FIX_SCALE)/(fb);            }
```

Over- and underflow

This is actually a subset of the scaling problems of multiplication and division. Overflow is when the result of your operation is higher that the about of bits you have to store it. This is a potential problem for any integer multiplication, but in fixed-point math it will occur much more often because not only are fixed-point numbers scaled upward, multiplying fixeds scales it up *twice*. A .8 fixed multiplication has its 'one' at 2^{16}, which is already out of range for halfwords.

One way of covering for the extra scale is not to correct after the multiplication, but before it; though you will lose some accuracy in the process. A good compromise would be to right-shift both operands by half the full shift.

Fixed divisions have a similar problem called underflow. As a simple example of this, consider what happens in integers division a/b if $b>a$. That's right: the result would be zero, even though a fraction would be what you would like. To remedy this behaviour, the numerator is scaled up by M first (which may or may not lead to an overflow problem `:P`).

As you can see, the principles of fixed-point math aren't that difficult or magical. But you do have to keep your head: a missed or misplaced shift and the whole thing crumbles. If you're working on a new algorithm, consider doing it with floats first (preferably on a PC), and convert to fixed-point only when you're sure the algorithm itself works.

Faking division (optional)

MATH HEAVY AND OPTIONAL

This section is about a sometimes useful optimization technique. It not only introduces the technique, but also derives its use and safety limits. As such, there is some nasty math along the way. Chances are you're perfectly safe without detailed knowledge of this section, but it can help you get rid of some slow divisions if the need is there.

You may have heard of the phrase "division by a constant is multiplication by its reciprocal". This technique can be used to get rid of division and replace it with a much faster multiplication. For example $x/3 = x \cdot (1/3) = x \cdot 0.333333$. At first glance, this doesn't seem to help your case: the integer form of $1/y$ is always zero by definition;

the alternative to this is floating-point, which isn't so hot either, and you *still* need a division to get even there! This is all true, but the important thing is that these problems can be avoided. The integer/floating-point problem can be solved by using fixed-point instead. As for the division, remember that we're talking about division by a *constant*, and arithmetic on constants is done at compile-time, not runtime. So problems solved, right? Uhm, yeah. Sure. The *superficial* problems are solved, but now the two age-old problems of overflow and round-off rear their ugly heads again.

Below is the code for the evaluation of "$x/12$". The ARM-compiled code creates a .33 fixed-point for 1/12, then uses a 64bit multiplication for the division. On the other hand, the Thumb version doesn't (and indeed can't) do this and uses the standard, slow division routine. If you want to get rid of this time consuming division, you will have to take care of it yourself. for the record, yes I know that even if you know ARM assembly, why it does what it does may be hard to follow. That's what this section is for.

```
@ Calculating y= x/12

@ === Thumb version ===
    ldr     r0, .L0     @ load numerator
    ldr     r0, [r0]
    mov     r1, #12     @ set denominator
    bl      __divsi3    @ call the division routine
    ldr     r1, .L0+4
    str     r0, [r1]
.L0:
    .align  2
    .word   x
    .word   y

@ === ARM version ===
    ldr     r1, .L1         @ Load M=2^33/12
    ldr     r3, .L1+4
    ldr     r3, [r3]        @ Load x
    smull   r2, r0, r1, r3  @ r0,r2= x*M (64bit)
    mov     r3, r3, asr #31     @ s = x>=0 ? 0 : -1 (for sign
correction)
    rsb     r3, r3, r0, asr #1  @ y= (x*M)/2 - s = x/12
    ldr     r1, .L1+8
    str     r3, [r1]        @  store y
.L1:
    .align  2
    .word   715827883   @ 0x2AAAAAAB (≈ 2^33/12 )
    .word   x
    .word   y
```

The remainder of this section is on recognizing and dealing with these problems, as well as deriving some guidelines for safe use of this technique. But first, we need some definitions.

Integer division; positive integers p, q, r

$$(B.3) \qquad r = \lfloor p / q \rfloor \iff p = r \cdot q + p \% q$$

Approximation; positive integers x, y, a, m, n and real error term δ

$$(B.4) \qquad y = \lfloor x / a \rfloor = \lfloor (x \cdot m) / n \rfloor + \delta$$

I'm using the floor ($\lfloor p/q \rfloor$) to indicate integer division, which is basically the rounded down version of real division. As usual, modulo is the remainder and calculated usually calculated with $p - r \cdot q$. The key to the approximation of $1/a$ is in terms m and n. In our case n will be a power of two $n=2^F$ so that we can use shifts, but it need not be. δ is an error term that is inherent in any approximation. Note that I'm only using positive integers here; for negative numbers you need to add one to the result if you want to mimic a 'true' division. (Or, subtract the sign bit, which work just as well as you can see in the ARM assembly shown above.)

> **FAKING NEGATIVE DIVISIONS AND ROUNDING**
>
> This section is about positive numbers. If you want the standard integer-division result (round toward zero), you will have to add one if the numerator is negative. This can be done quickly by subtracting the sign-bit.
>
> ```
> // pseudo-code for division by constant M
> int x, y;
> y= fake_div(x, M);  // shift-like div
> y -= y>>31;         // convert to /-like division
> ```
>
> If you want to round to minus infinity you'll have to do something else. But I'm not quite sure what, to be honest.

Theory

There are two things we need to have for success. First, a way of finding m. Second, a way of determining when the approximation will fail. The latter can be derived from eq B.4. The error in the approximation is given by $\lfloor \varepsilon/n \rfloor$, so as long as this is zero you're safe.

$$(B.5) \qquad \begin{array}{c} x \cdot m - n \cdot \lfloor x / A \rfloor = \varepsilon \\ \text{Fail if: } \varepsilon \geq n \end{array}$$

As for finding m. Recall that $\lfloor 1/A \rfloor = \lfloor (n \cdot A)/n \rfloor$, so that it'd appear that using $m = \lfloor n/A \rfloor$ would be a good value. However, it's not.

This is probably a good time for a little example. Consider the case of $A = 3$, just like at the start. We'll use .8 fixed numbers here, in other words $k = 8$ and $n=256$. Our trial m is then $m = \lfloor n/A \rfloor = 85 = 0x55$, with 1 as the remainder.

An alternative way of looking at it is to go to hexadecimal floating point and taking the first F bits. This is not as hard as you might think. The way you find a floating-point number of a fraction is to multiply by the base, write down the integral part, multiply the remainder by the base, write down the integral part and so forth. The table below has the hex version of 1/7 (I'm not using 1/3 because that's rather monotonous). As you can see 1/7 in hex is 0.249249…h. Do this for one third and you'll find 0.5555…h.

x	x·B	x·$\lfloor B/7 \rfloor$	x·B%7
1	16	2	2
2	32	4	4
4	64	9	1
1	16	2	2
2	32	4	4

Table B.2: Floating-point representation of 1/7 in base B=16

So 1/3 in hex is zero, followed by a string of fives, or just m=0x55 in truncated .8 fixed-point notation. Now look what happens when you do the multiplication by reciprocal thing. I'm using hex floats here, and $y=\lfloor (x \cdot m)/n \rfloor$, as per eq B.4. The result you actually get is just the integer part, ignore the (hexi)decimals

x	0	1	2	3	4	5	6	7
y= (x·m)>>F	0.00h	0.55h	0.AAh	**0.FFh**	1.54h	1.A9h	**1.FEh**	2.53
true x/3	0	0	0	1	1	1	2	2

Table B.3: x/3, using $m= \lfloor 256/3 \rfloor = 0x55$. Bad at 3, 6, …

As you can see, problems arise almost *immediately*! You can't even get up to *x=A* without running into trouble. This is *not* a matter of accuracy: you can use a .128 fixed-point numbers and it'll still be off. This is purely a result of **round-off error**, and it'd happen with floats just as well. When you use reciprocal division, *m* should be rounded *up*, not down. You can use the alignment trick here: add *A−1*, then divide. Now *m*=0x56, and you'll be safe. At least, for a while.

$$(B.6) \qquad m = \lfloor (n + A - 1) / A \rfloor$$

x	0	1	2	3	4	5	6	7
y= (x·m)>>F	0.00h	0.56h	0.ACh	1.02h	1.58h	1.AEh	2.04h	2.5A
true x/3	0	0	0	1	1	1	2	2

Table B.4: x/3, using m= $\lfloor (256+2)/3 \rfloor$ = 0x56. Still good at 3, 6, …

Yes, you're safe. But for how long? Eventually, you'll reach a value of *x* where there will be trouble. This time around, it does concern the accuracy. Fortunately, you can derive safety limits for *x* and *n* that spell out when things can go badly. It is possible that the true range is a little bit better due to the way the error condition of eq B.5 jumps around, but better safe than sorry. The derivations start at eq B.5 and make use of eq B.3 and a trick concerning modulo, namely that $p\%q \in [0, q)$.

$$(B.7) \qquad \begin{aligned}
x \cdot m - n \lfloor x / A \rfloor &< n \\
x \cdot m \cdot a - n \lfloor x / A \rfloor A &< n \cdot A \quad [\text{insert } \lfloor x / A \rfloor A = x - x \% A] \\
x \cdot m \cdot A - n \cdot x + n(x \% A) &< n \cdot A \quad [\text{insert } \max (x \% A) = A - 1] \\
x(m \cdot A - n) + n(A - 1) &< n \cdot A \\
x(m \cdot A - n) &< n
\end{aligned}$$

From this result, we can easily calculate the maximum valid *x* for given *A* and *n*:

$$(B.8) \qquad x < n / (m{\cdot}A - n)$$

The lower-limit for *n* follows from the fact that, by (6), max(m·A) = n+A−1, so that:

$$\text{(B.9)} \qquad n > x(A-1)$$

And that's basically it. There's a little more to it, of course. As you'll be multiplying, the product $m \cdot A$ must fit inside a variable. The practical limit of numbers will therefore be around 16 bits. You can sometimes ease this limitation a little bit by shifting out the lower zero-bits of A. For example, for $A=10=5 \cdot 2$, you can right-shift x once before doing the whole calculation. Even 360 is $45 \cdot 8$, and you can save three bits this way. Also, note that even if you surpass the limits, there's a good chance that the thing is still correct or only off by a small amount (check eq B.5). You should be able to find the true answer relatively quickly then.

> **ARM 'INT/CONST INT' DIVISION IS ALWAYS SAFE**
>
> We can now see why GCC can always safely optimize 32bit divisions. The maxima of 32bit x and A are, of course, 2^{32}. The safety limit for this is $2^{64}-2^{32}$, which will always fit in the 64bit result of `smull` .

Of course, you don't want to have to type in these things all the time. So here are two macros that can do the work for you. They look horrible, but the preprocessor and compiler know how to handle them. I'd advise against converting these to inline functions, because for some reason there is a good chance you will lose any advantages the code is supposed to bring.

```
// Division by reciprocal multiplication
// a, and fp _must_ be constants

//! Get the reciprocal of \a a with \a fp fractional bits
#define FX_RECIPROCAL(a, fp)    ( ((1<<(fp))+(a)-1)/(a) )

//! Division of x/a by reciprocal multiplication
#define FX_RECIMUL(x, a, fp)    ( ((x)*((1<<(fp))+(a)-1)/(a))>>(fp) )
```

Summary

Never forget that this is something of a hack and **only** works when *A* is constant. The whole point was to have the division at compile time rather than runtime, and that is only possible if *A* is constant. One nice thing about constants is that they're known beforehand, by definition. Negative values and powers of two may be resolved at compile-time too if desired.

The reciprocal multiplier *m* is *not* merely $\lfloor n/A \rfloor$, for reasons of round-off error. Always round up. In other words:

$$m = \lfloor (n+A-1) / A \rfloor$$

Then there's the matter of failed divisions, i.e. where the approximation differs from the 'true' $\lfloor x/A \rfloor$. The exact condition doesn't really matter, but it is useful to know the safe ranges of *x*, and conversely what *n* you need for a given *x*-range. Again, because the important terms are constant they can be figured out in advance. Note that the relations given below represent *A* limit, not *the* limit. The actual numbers for failure may be a bit looser, but depend on the circumstances and as such, relations for those would be more complex.

$$x < n / (m \cdot a - n)$$

$$n > x(A-1)$$

Lastly, if you have absolutely no idea what this whole section was about, I'd advise against using this strategy. It is a partially safe optimisation technique for division and while it can be a good deal faster that the normal division, it might not be worth it in non-critical areas. Just, use your judgement.

> **ALTENATIVE METHOD**
>
> There is an alternative method for reciprocal multiplication: instead of rounding n/A up, you can also add 1 to x for
>
> $$y = \lfloor x / A \rfloor \approx (x+1) \times \lfloor N / A / N \rfloor$$
>
> This will also get rid of the problems described by table B.3. The safety conditions are almost the same as before, but there is some difference for division of negative x. If you *really* must know, details are available on request.

Look-up Tables

A ***look-up table*** (or ***LUT***) is, well, it's a table that you use to look stuff up. That was rather obvious, wasn't it? The important point is that you can do it really *quickly*. As a simple example, what is 2^5, and what is 3^5? Real Programmers (should) know the answer to the first one instantly, but the other might take a little longer to find. Why?

Because any self-respecting programmer knows the powers of 2 by heart, up to 2^{10} at least. The powers of 2 crop up so often in programming that you've got the answers memorized – you see as much as see the question "2^5", you don't calculate the answer via repeated multiplication, your mind simply **looks it up** in its memory and you give the answer instantly, almost without thinking. The same goes for (decimal) multiplication tables: 7×8? 56, just like that. But move to, say, powers of 3 or hexadecimal multiplications and the process fails and you have to do it the hard and long way. What I'm trying to say here is that things can go a lot faster when you can simply look them up, rather than having to do the proper calculations.

The concept of look-up tables works for computers as well, otherwise I wouldn't have brought it up. In this case, the look-up table is simply an array that you stuff with whatever you think you might need to look up.

Example: sine/cosine LUTs

Classic examples are trigonometry LUTs. Sines and cosines are expensive operations, especially on the GBA, so it would be best to make a table of them so you only have to spend a memory access instead of going through two (expensive) type conversions, floating-point function. A simple way to do this is to create two FIXED arrays of, say, 360 elements each (one for every degree) and fill it at the start of your game.

```c
#define PI 3.14159265
#define DEGREES    360      // Full circle

FIXED sin_lut[DEGREES], cos_lut[DEGREES];

// A really simple (and slow and wasteful) LUT builder
void sincos_init()
{
    const double conv= 2*PI/DEGREES;
    for(int ii=0; ii<DEGREES; ii++)
    {
        sin_lut[ii]= (FIXED)(sin(conv*ii)*FIX_SCALEF);
        cos_lut[ii]= (FIXED)(cos(conv*ii)*FIX_SCALEF);
    }
}
```

However, this particular method is deeply flawed. Yes, it works, yes, it's easy, but there is definitely room for improvement. To start with an issue that would be immediately visible if you were to use this function, it actually takes a few *seconds* to complete. Yes, that's how slow the standard trig routines are. This is a fairly mild issue as you only have to call it once, but still. Additionally, because the arrays are not constant, they are put in IWRAM. That's 10% of IWRAM basically wasted on something that is never actually changed except during initialization. There are a number of ways of improving on these two points like using the sine-cosine symmetries to cut down on calculation time and having the tables overlap, but why calculate them inside the game at all? It is just as easy to precalculate the tables on a PC, then export that data to arrays: then they will be constant (i.e., not hogging IWRAM), and the GBA won't have to spend a cycle on their initialization.

A second improvement would be to use a higher fixed-point fraction. The range of sine and cosine is [−1, +1]. This means that by using 8.8 fixeds for the LUT, I am actually wasting 6 bits that I could have used for a higher accuracy. So what I'm going to do is use 4.12 fixed-point. Yes, you could go up to .14 fixeds, but 12 is a nicer number.

And for the final improvement, I'm not going to use 360 units for a circle, but a power of two; 512 in this case. This has two benefits:

- For wrapping ($\alpha<0$ or $\alpha>2\pi$), I can use a bitmask instead of if-statements or *gasp* modulo.
- Since the cosine is just shifted sine, and because of point one, I now only need one table for both waves, and can use an offset angle and wrap-by-masking to get one wave from the other.

Both these points can make life a lot easier.

For the record, it is perfectly alright to this. The forms of sine and cosine stem from travelling along the circumference of the unit circle; the number of divisions along that path is arbitrary. The number 360 has historical significance, but that's it. Let's face it, you wouldn't be able to tell how much a degree is anyway, the thing that matters is circle divisions. 360° is a full circle, 90° is a quarter circle, et cetera. Now it's 512 for a full circle, 128 (512/4) for a quarter, and so on. A quick and dirty sin LUT generator might look something like this. Summing up:

- Precalculate the LUT outside the GBA, and link it in like a normal const array.
- Use 4.12 fixeds instead of 4.8.
- Divide the LUT into a power-of-two (like 512), instead of 360.

```
// Example sine LUT generator
#include <stdio.h>
#include <math.h>

#define M_PI 3.1415926535f
#define SIN_SIZE 512
#define SIN_FP 12

int main()
{
    int ii;
    FILE *fp= fopen("sinlut.c", "w");
    unsigned short hw;

    fprintf(fp, "//\n// Sine LUT; %d entries, %d fixeds\n//\n\n",
        SIN_SIZE, SIN_FP);
    fprintf(fp, "const short sin_lut[%d]=\n{", SIN_SIZE);
    for(ii=0; ii<SIN_SIZE; ii++)
    {
        hw= (unsigned short)(sin(ii*2*M_PI/SIN_SIZE)*(1<<SIN_FP));
        if(ii%8 == 0)
            fputs("\n\t", fp);
        fprintf(fp, "0x%04X, ", hw);
    }
    fputs("\n};\n", fp);

    fclose(fp);
    return 0;
}
```

It creates a file sinlut.c which contains a 512 halfword array called `sin_lut` . Note that while I'm creating a C file here, you could just as well create a table in an assembly file, or even just as a binary file that you then somehow link to the project. Actually finding the sine and cosine values goes through the `lu_sin()` and `lu_cos()` functions.

```
// Sine/cosine lookups.
// NOTE: theta's range is [0, 0xFFFF] for [0,2π), just like the
// BIOS functions

//! Look-up a sine value
INLINE s32 lu_sin(u32 theta)
{   return sin_lut[(theta>>7)&0x1FF];   }

//! Look-up a cosine value
INLINE s32 lu_cos(u32 theta)
{   return sin_lut[((theta>>7)+128)&0x1FF]; }
```

Presenting excellut

I haven't actually used the generator shown above for the LUTs in libtonc. Rather, I've used my own excellut. This is not a program, but an Excel file. Yes, I did say Excel. The thing about using a spreadsheet program for building LUTs is that you can make *any* kind of mathematical table with it, test whether it has the desired accuracy and plot it and everything. Then after you're satisfied, you can then just export part of the spreadsheet in the manner of your choice. How's that for flexibility?

Accuracy and resolution

These are the two main things to consider when creating your LUT. *Accuracy* concerns the number of significant bits of each entry; *resolution* is how far apart each entry is in the argument space. Bigger is better in both cases, but there is a space trade-off, of course. A compromise is necessary, and once again, it depends very much on what you intend to do with it.

For accuracy, you need to think of the range of the function. As said, the sine range is [−1, +1] and using 8.8 fixeds would waste 6 bits that could have been used for more significant bits.. For a division LUT like the one I'm using for the first mode 7 chapter, I need 1/1 up to 1/160, which would *not* work well with .8 fixeds, so I'm using .16 fixeds there, which may still not be enough, but more might give overflow problems.

The second issue, resolution, is tied to how many entries you have. Even if you have all the accuracy in the world, it wouldn't do you much good if they're spread out too thin. It's similar to screen resolutions: even with 32bit color, things will look mighty hideous if you only have a 17 inch monitor with a 320×240 resolution. On the other hand, an overly high resolution doesn't help you if the accuracy isn't there to support it. Most LUT-worthy functions will be smooth curves and for any given accuracy, you will reach a point where increasing resolution will only add identical values to the LUT, which would be a waste of space. And remember, if you really need to you can always do an interpolation if necessary.

The first few values of my 512, 8.8 fixeds sine-lut, for example, read "0x0000, 0x0003, 0x0006, 0x0009" That is where the derivative is maximal, so these are the largest differences between neighbours you will find. If I were to increase the resolution fourfold, the differences would be in the final bit; going any further would be useless unless I increased the accuracy as well.

So it's actually a three-way compromise. There needs to be a balance between accuracy and resolution (the derivative of the function would be helpful to find this), and pitted against those two is the amount of ROM space you want to allot to the LUT. Again, the only person who can judge on the right balance is you.

Linear interpolation of LUTs

Look-up tables are essentially a collection of points sampled from a function. This is fine if you always access the arrays at those points, but what if you want to retrieve value between points? An example of this would be a fixed-point angle, like `theta` of the (co)sine inline functions. Usually, the lower bits of a fixed-point number are just cut off and the point before it is used. While fast, this does have some loss of accuracy as its result.

A more accurate solution would be to use the surrounding points and interpolate to the desired point. The easiest of these is *linear interpolation* (or *lerp*). Say you have a point x_a and x_b, with function values y_a and y_b, respectively. This can be used to define a line. The function value of point x can then be interpolated by:

$$(B.10) \quad y = \frac{y_b - y_a}{x_b - x_a}(x - x_a) + y_a$$

Fig B.1 gives an example of the difference that linear interpolation can make. Here I have a sine function sampled at 16 points and at .12f precision. The blue line represents the actual sine function. The magenta like is the direct look-up using the preceding point, and the lerp is given by the yellow line. Note that the blue

Fig B.1: approximating a sine by direct look-up or linear interpolation.

and yellow lines are almost the same, but the magenta line can be a good deal off. Consider $x = 4.5$, given in red. The LUT value is off by 8.5%, but the lerp value by only 0.5%: that's 16 times better! True, this is an exaggerated case, but lerping can make a huge difference.

So how do we implement this? Well, essentially by using eq B.10. The division in it may look nasty, but remember that the difference between successive points is always 1 – or a power-of-two for fixed point numbers. An efficient implementation would be:

```
//! Linear interpolator for 32bit LUTs.
/*! A LUT is essentially the discrete form of a function, f(\i x).
*    You can get values for non-integer \i x via (linear)
*    interpolation between f(x) and f(x+1).
*    \param lut  The LUT to interpolate from.
*    \param x    Fixed point number to interpolate at.
*    \param shift    Number of fixed-point bits of \a x.
*/
INLINE int lu_lerp32(const s32 lut[], int x, const int shift)
{
    int xa, ya, yb;
    xa=x>>shift;
    ya= lut[xa]; yb= lut[xa+1];
    return ya + ((yb-ya)*(x-(xa<<shift))>>shift);
}
```

That's the version for 32-bit LUTs, there is also a 16-bit version called `lu_lerp16()`, which has the same body, but a different declaration. In C++, this would make a nice template function.

These functions work for every kind of LUT, expect for a little snag at the upper boundary. Say you have a LUT of N entries. The functions use $x+1$, which is likely not to exist for the final interval between $N-1$ and N. This could seriously throw the interpolation off at that point. Rather than covering that as a special case, add an extra point to the LUT. The `sinlut` actually has 513 points, and not 512. (Actually, it has 514 points to keep things word-aligned, but that's beside the point.)

> **LERPING AT THE UPPER BOUNDARY**
>
> Linear interpolation needs the sampling point above and below x, which can cause problems at the upper boundary. Add an extra sampling point there to "finish the fence', as it were.

The direct look-up is also known as 0-order interpolation; linear interpolation is first order. Higher orders also exists but require more surrounding points and more and complexer calculations. Only attempt those if you really, really have to.

Non mathematical LUTs

While the most obvious use of lookup tables is for precalculated mathematical functions, LUTs aren't restricted to mathematics. In my text systems, for example, I'm using a look-up table for the character→tile-index conversion. This offers me a greater range in the distribution of tiles that otherwise would be possible. The default font uses ASCII characters 32-127, and the tiles for these are usually in tiles 0 through 95. But if for some reason I would only need the number tiles, I could set-up the character LUT for only numbers, and the text system would take it from there. the rest of the tiles would then be free for other purposes.

Another use would be flag lookup. The libraries that come with Visual C++ use LUTs for the character type routines, `isalpha()`, `isnum()` and the like. There is a table with bitflags and when you use the routines, they just grab the appropriate element and do some bit testing. You can find something similar in game programming too, like a table with bitflags for the tile types: background, walkable, trap, etc. Instead of massive switch-blocks, you might only have to do an array-lookup, which is a lot faster.

C. Vectors and Matrices

Vectors

Before I go into what a vector is, I'll first tell you what it isn't. Generally, you divided physical quantities into *scalars* and *vectors*. A *scalar* gives the magnitude of a quantity. It's a single number, like the ones you use every day. Mass, energy and volume are examples of scalars. A *vector* is something with both a magnitude *and* direction, and is usually represented by multiple numbers: one for every dimension. Position, momentum and force are prime examples. Also, note that velocity is a vector, while speed is not. 50 kph is not a vector. 50 kph down Highway 60 is. The notation of a vector is as a bold character, usually lowercase, and either as a set of numbers enclosed by parentheses, **u** = (1, 4, 9), or as an M×1 column. And yes, I do mean a column, not a row; we'll see why when we get to matrices.

$$(C.1) \qquad \mathbf{u} \equiv \begin{bmatrix} u_1 \\ \vdots \\ u_m \end{bmatrix} \equiv (u_1, \cdots, u_m) \equiv\!\!\!/ \begin{bmatrix} u_1 & \cdots & u_m \end{bmatrix}$$

If you have a coordinate system, vectors are usually used to represent a spatial point in that system, with vectors' elements as the coordinates. However, there is a crucial difference between points and vectors. Points are always related to an origin, while vectors can are independent of any origin. Fig C.1 on the right illustrates this. You have points *P* and *Q*, and vectors **u, v, w**. Vectors **u** and **v** are equal (they have equal

lengths and directions). However, while **u** and the point it points to (*P*) have the same coordinates, this isn't true for **v** and *Q*. In fact, *Q* = **u** + **w**. And, to be even more precise, *Q* = *O* + **u** + **w**, which explicitly states the origin (*O*) in the equation.
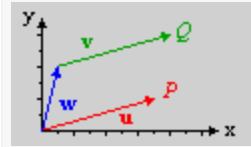
Fig C.1: the difference between vectors and points.

Vector operations

Vector operations are similar to scalar operations, but the multi-dimensionality does add some complications, especially in the case of multiplications. Note that there are no less than *three* ways of vector-multiplication, so pay attention. On the right you can see examples of vector addition and scalar-vector multiplication. **u**= (8, 3), **v**= (-4 ,4). With the definitions of the operations given below, you should be able to find the other vectors.

Vector-vector addition and subtraction

When it comes to addition and subtraction, both operands must be M-dimensional vectors. The result is another vector, also M-dimensional, which elements are the sum or difference of the operands' elements: with **w** = **u** + **v** we have $w_i = u_i + v_i$.
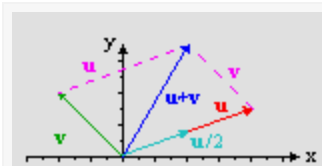
Fig C.2: vector addition and scalar-vector multiplication.

$$(C.2) \qquad w = u + v \equiv \begin{bmatrix} u_1 + v_1 \\ \vdots \\ u_m + v_m \end{bmatrix}$$

Scalar-vector multiplication

This is the first of the vector multiplications. If you have a scalar *a* and a vector **u**, the elements of resultant vector after scalar-vector multiplication are the original elements, each multiplied with the scalar. So if **v** = *c* **u**, then $v_i = c * u_i$. Note that **u** and **v**

lie on the same line – only the length is different. Also, note that subtraction can also be written as $\mathbf{w} = \mathbf{u} - \mathbf{v} = \mathbf{u} + (-1)*\mathbf{v}$.

$$(C.3) \qquad \mathrm{v} = c\mathrm{u} \equiv \begin{bmatrix} c \cdot u_1 \\ \vdots \\ c \cdot u_m \end{bmatrix}$$

The dot-product (aka scalar product)

The second vector-multiplication is the dot-product, which has two vectors as input, but a *scalar* as its output. The notation for this is $c = \mathbf{u} \cdot \mathbf{v}$, where $\mathbf{u}$ and $\mathbf{v}$ are vectors and c is the resultant scalar. Note the operator is in the form of a dot, which gives this type of multiplication its name. To do the dot-product, multiply the elements of both vectors piecewise and add them all together. In other words:

$$(C.4) \qquad c = \mathrm{u} \cdot \mathrm{v} = \sum u_i \cdot v_i = u_1 \cdot v_1 + \cdots + u_m \cdot v_m$$

Now, this may seem like a silly operation to have, but it's actually very useful. For one thing, the length of the vector is calculated via a dot-product with itself. But you can also find the projection of one vector onto another with the dot-product, which is invaluable when you try to decompose vectors in terms of other vectors or determine the base-vectors of an M-dimensional space (do what to the whaaat?!? Don't worry, I'll explain later). One of the most common uses of the dot-product is finding the angle between two vectors. If you have vectors $\mathbf{u}$ and $\mathbf{v}$, $|\mathbf{u}|$ and $|\mathbf{v}|$ their lengths and α the angle between the two, the cosine can be found via

$$(C.5) \qquad \cos(\alpha) = \frac{\mathrm{u} \cdot \mathrm{v}}{|\mathrm{u}|\,|\mathrm{v}|}$$
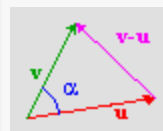
Fig C.3: dot product.

Why does this work? Well, you can prove it in a number of ways, but here's is the most elegant (thanks Ash for reminding me). Remember that the square of the length of a vector is given by the dot-product with itself. This means that $|\mathbf{v}-\mathbf{u}|^2 = |\mathbf{v}|^2 + |\mathbf{u}|^2 - 2 \cdot \mathbf{u} \cdot \mathbf{v}$. From the cosine rule

for the triangle in fig C.3, we also have $|\mathbf{v}-\mathbf{u}|^2 = |\mathbf{v}|^2 + |\mathbf{u}|^2 - 2 \cdot |\mathbf{v}| \cdot |\mathbf{u}| \cos(\alpha)$. Combined, these relations immediately result in eq C.5. And people say math is hard.

By the way, not only can you find the angle with this, but it also provides a very simply way to see if something's behind you or not. If **u** is the looking-direction and **v** the vector to an object, **u** · **v** is negative if the angle is more than 90°. It's also useful for field-of-view checking, and to see if vectors are perpendicular, as **u** · **v** = 0. You also find the dot-product by the truck-load in physics when yo do things like force decomposition, and path-integrals over force to find the potential energy. Basically, every time you find a cosine in an equation in physics, it's probably the result of a dot-product.

The cross-product (aka vector-product)

The cross product is a special kind of product that only works in 3D space. The cross-product takes two vectors **u** and **v** and gives the vector perpendicular to both, **w**, as a result. The length of **w** is the area spanned by the two operand vectors. The notation for it is this: **w** = **u** × **v**, which is why it's called the cross-product. The elements of **w** are $w_i = \varepsilon_{ijk} \cdot u_j \cdot v_k$, where ε_{ijk} is the Levi-Cevita symbol (+1 for even permutations of i,j,k, −1 for odd permutations, and 0 if any of the indices are equal). Since you've probably never even seen this thing (for your sanity, keep it that way), it's written down in full in eq C.4.

$$
(C.6) \qquad \mathbf{w} = \mathbf{u} \times \mathbf{v} \equiv \begin{bmatrix} u_y v_z - u_z v_y \\ u_z v_x - u_x v_z \\ u_x v_y - u_y v_x \end{bmatrix}
$$
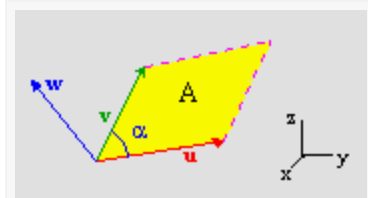
Fig C.4: cross product.

In fig C.4 you can see a picture of what the cross-product does; it's a 3D picture, so you have to use your imagination a bit. Vectors **u** and **v** define a parallelogram (in yellow). The cross-product vector **w** is perpendicular to both of these, a fact that follows from **u**·**w** and **v**·**w**. The length of **w** is the area of this parallelogram, A and if you remember your area-calculations, you'll realize that

$$(C.7) \qquad A = | \mathbf{u} \times \mathbf{v} | = |\mathbf{u}| \cdot |\mathbf{v}| \cdot \sin(\alpha)$$

meaning that you can find the sine of the angle between two vectors with the cross-product. Note that the cross-product is that it is *anti*-commutative! That means that **u** × **v** = −**v** × **u**. Notice the minus sign? This actually brings up a good point: the plane defined by **u** and **v**, the normal vector to this plane is pointing up; but how do you determine what 'up' is? What I usually do is take a normal 3D coord-system (like the one in the lower-right part of fig C.4), put the *x*-axis on **u**, rotate till the *y*-axis is along **v** (or closest to it), and then **w** will be along the *z*-axis. Eq C.6 has all of this sorted out already. I do need a right-handed system for this, though, a left-handed one messes up my mind so bad.

Now, when the vectors are parallel, **u** × **v** = 0, which means that **w** is the null-vector **0**. It also means that if **u** is your view direction, the object with vector **v** is dead-center in your sights. However, if **u** is the velocity of a rocket and **v** is the relative vector to *you*, prepare to respawn. Basically, whereas the dot-product tells you whether an object is in front or behind (along the tangent), the cross-product gives you the offset from center (the normal). Very useful if you ever want to implement something like red shells (and by that I mean the *original* SMK red shells, not the wussy instant-homing shells in the later Mario Karts, booo!!). The cross-product also appears abundantly in physics in things like angular momentum (**L** = **r** × **p**) and magnetic induction.

That was the 3D case, but the cross-product is also useful for 2D. Everything works exactly the same, except that you only need the *z*-component of **w**.

The norm (or length)

I've used this already a couple of times but never actually defined what the length of a vector is. The norm of vector **u** is defined as the square root of the dot-product with itself, see eq C.8. The age-old Pythagorean Theorem is just the special case for 2D.

The length or norm of a vector is a useful thing to have around. Actually, you often start with the length and use the sine and cosine to decompose the vector in *x* and *y* components. A good example of this is the speed. One other thing where the length

plays a role is in the creation of *unit-vectors*, which have length 1. Many calculations require the length in some way, but if that's one, you won't have to worry about that anymore. To create a normal vector, simply define it by its length: û = **u** / |**u**|.

$$\text{(C.8)} \qquad |u| = \sqrt{(u \cdot u)} = \left(\sum u_i^2 \right)^{\frac{1}{2}}$$

Algebraic properties of vectors

What follows is a list of algebraic properties of vectors. Most will seem obvious, but you need to see them at least once. Take directly from my linear algebra textbook: let **u, v, w** be M-dimensional vectors and *c* and *d* scalars, then:

u + **v** = **v** + **u**	Commutativity
(**u** + **v**) + **w** = **u** + (**v** + **w**)	Associativity
u + **0** = **0** + **u** = **u**	
u + (−**u**) = −**u** + **u** = **0**	where −**u** denotes (−1)**u**
c·(**u** + **v**) = c·**u** + c·**v**	Distributivity
(c + d)·**u** = c·**u** d·**u**	Distributivity
c·(d·**u**) = (c·d)·**u**	Associativity
1·**u** = **u**	

And on the products:

u · (**v** + **w**) = (**u** + **v**) · **w**	
u · (c·**v**) = (c·**u**) · **v** = c·(**u** · **v**)	
u × **v** = −(**u** × **v**)	Anti-commutativity
u × (**v** + **w**) = **u** × **v** + **u** × **w**	
(**u** + **v**) × **w** = **u** × **w** + **v** × **w**	
u × (c·**v**) = c·**u** × **v** = (c·**u**) × **v**	

$\mathbf{u} \cdot (\mathbf{u} \times \mathbf{v}) = 0$	
$\mathbf{u} \cdot (\mathbf{v} \times \mathbf{w}) = (\mathbf{u} \times \mathbf{v}) \cdot \mathbf{w}$	Triple scalar product, gives the volume of parallelepiped defined by **u, v, w**.
$\mathbf{u} \times (\mathbf{v} \times \mathbf{w}) = \mathbf{u}(\mathbf{v} \cdot \mathbf{w}) - \mathbf{w}(\mathbf{u} \cdot \mathbf{v})$	Triple vector product

Matrices

In a nutshell, a matrix is a 2-dimensional grid of numbers. They were initially used as shorthand to solve a system of linear equations. For example, the system using variables *x, y, z*:

$$(C.9a) \quad \begin{array}{rcrcrcr} x & - & 2y & + & z & = & 0 \\ & & 2y & - & 8z & = & 8 \\ -4x & + & 5y & + & 9z & = & -9 \end{array}$$

can be written down more succinctly using matrices as:

$$(C.9b) \quad \begin{bmatrix} 1 & -2 & 1 \\ 0 & 2 & -8 \\ -4 & 5 & 9 \end{bmatrix} \quad \text{or} \quad (C.9c) \quad \begin{bmatrix} 1 & -2 & 1 & 0 \\ 0 & 2 & -8 & 8 \\ -4 & 5 & 9 & -9 \end{bmatrix}$$

Eq C.9b is called the ***coefficient matrix***, in which only the coefficients of the variables are written down. The ***augmented matrix*** (eq C.9c) also contains the right-hand side of the system of equations. Note that the variables themselves are nowhere in sight, which is more or less the point. Mathematicians are the laziest persons in the world, and if there's a shorthand to be exploited, they will use it. If there isn't, they'll make one up.

Anyway, a matrix can be divided into ***rows***, which run horizontally, or ***columns***, which run vertically. A matrix is indicated by its size: an M×N matrix has M rows and N columns. Note that the number of rows comes first; this in contrast to image sizes,

where width is usually given first. Yeah I know, that sucks, but there's not a lot I can do about that. The coefficient matrix of eq C.9b is a 3x3 matrix, and the augmented matrix of eq C.9c is 3x4. The whole matrix itself is usually indicated by a bold, capital; the columns of a matrix are simply vectors (which were M×1 columns, remember?) and will be denoted as such, with a single index for the column-number; the *elements* of the matrix will be indicated by a lowercase (italic) letter with a double index.

$$(C.10) \qquad A = [\ a_1 \ \cdots \ a_n \ ] \equiv \begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Most computer languages also have the concept of matrices, only they don't always agree in how the things are ordered. Indexing in Visual Basic and C, for example, is row-based, just like eq C.10 is. Fortran, on the other hand, is vector-based, so the indices need to be reversed. Thanks to the C's pointer-type, you can also access a matrix as an array.

```
mat(i, j)      // VB matrix

mat[i][j]      // C matrix
mat[i+N*j]     // C matrix, in array form

mat(j, i)      // Fortran matrix
```

Let's return to (eq C.9) for a while, if we use **x** = (x, y, z), **b**= $(0, 8, -9)$, and **A** for the coefficient matrix, we can rewrite (eq C.9a) to

$$(C.9d) \qquad a_1 \cdot x + a_2 \cdot y + a_3 \cdot z = b = A \cdot x$$

I've used the column-vector notation on the left of **b**, and the full matrix notation on the right. You will do well to remember this form of equation, as we'll see it later on as well. And yes, that's a matrix-multiplication on the right-hand side there. Although I haven't given a proper definition of it yet, this should give you some hints.

Matrix operations

Transpose

To transpose a matrix is to mirror is across the diagonal. It's a handy thing to have around at times. The notation for the transpose is a superscript uppercase 'T', for example, $B = A^T$. If **A** is an M×N matrix, its transpose **B** will be N×M, with the elements $b_{ij} = a_{ji}$. Like I said, mirror it across the diagonal. The diagonal itself will, of course, be unaltered.

Matrix addition

Matrix addition is much like vector addition, but in 2 dimensions. If **A, B, C** are all M×N matrices and **C = A + B**, then the elements of **C** are $c_{ij} = a_{ij} + b_{ij}$. Subtraction is no different, of course.

Matrix multiplication

Aahhh, and now things are getting interesting. There are a number of rules to matrix multiplication, which makes it quite tricky. For our multiplication, we will use **C = A ·** **B**. The thing is that the number of columns of the first operand (**A**) *must* equal the number of rows of the second (**B**). So if **A** is a *p×q* matrix, **B** should be a *q×r* matrix. The size of **C** will then be *p×r*. Now, the elements of **C** are given by

$$(C.11) \qquad c_{ij} \equiv \sum_k a_{ik} \cdot b_{kj}$$

In other words, you take row *i* of **A**, column *j* of **B** and take their dot-product. *k* in eq C.11 is the summation-index for this dot-product. This is also the reason why the columns of **A** and the rows of **B** must be of equal size; if not you'll have a loose end at either vector. Another way of looking at it is this: The whole of **A** forms the coefficient matrix of a linear system, similar to that of eq C.9b. The columns of **B** are all vectors of variables which, when processed by the linear system, gives the columns of **C**:

$$(C.12) \qquad C = A \cdot B \equiv A[\ b_1 \quad \cdots \quad b_r\ ] \equiv [\ A \cdot b_1 \quad \cdots \quad A \cdot b_r\ ]$$

The value of this way of looking at it will become clear when I discuss coordinate transformations. Also, like I said, for matrix-multiplication you take the dot-product of a row of **A** and a column of **B**. Since a vector is basically an M×1 matrix, the normal dot-product is actually a special case of the matrix-multiplication. The only thing is that you have to take the transposed of the first vector:

$$(C.13) \qquad c = u \cdot v = [\ u\ ]^T \cdot [\ v\ ]$$

There's a wealth of other things you can do with matrix-multiplication, but I'll leave it with the following two notes. First, the operation is *not commutative*! What that means is that **A · B ≠ B · A**. you may have guessed that from the row-column requirement, but even if those do match up it is still not commutative. My affine sprite demo kind of shows this: a rotation-then-scale does not give the same results as scale-then-rotate (which is probably what you wanted). Only in very special cases is **A · B** equal to **B · A**.

The other note is that matrix multiplication is expensive. You have to do a dot-product (*q* multiplications) for each element of **C**, which leads to *p**q*r multiplications. That's an O(3) operation, the nastiest ones around. OK, so for 2x2 matrices it doesn't amount to much, but when you deal with 27x18 matrices (like I do for work), this becomes a problem. Fortunately there are ways of cutting down on the number of calculations, but that's beyond the scope of this tutorial.

Determinant

The determinant is a scalar that you get when you combine the elements of a *square matrix* (of size N×N) a certain way. I've looked everywhere for a nice, clear-cut definition of the determinant, but with very little luck. It seems it has a number of uses, but it is most often used as a simple check to see if a equations of a system (or a set of vectors) is linearly independent, and thus if the coefficient matrix is invertible.

The mathematical definition of the determinant of N×N matrix **A** is a recurrence equation and looks like this.

$$(C.14) \qquad \det A = |A| = \sum_{j} (-1)^{1+j} \, a_{1j} \, \det A_{1j}$$

I could explain this in more detail, but there's actually little point in doing that. I'll just give the formulae for the 2x2 and 3x3 case. Actually, I've already done so: in the cross product. If you have matrix **A** = [**a₁ a₂ a₃**], then |**A**| = **a₁** · (**a₂** × **a₃**). For a 2×2 matrix, **B** = [**b₁ b₂**] it's $b_{11} \cdot b_{22} - b_{12} \cdot b_{21}$, which in fact also uses the cross product. This is not a mere coincidence. Part of what the determinant is used for is determining whether a matrix can be inverted. Basically, if |**A**| = 0, then there is no inverse matrix. Now, remember that the cross-product gives is involved in the calculation of the area between vectors. This can only be 0 if the vectors are colinear. And linear independence is one of the key requirements of having an inverse matrix. Also, notice the notation for the determinant: det **A** = |**A**|. Looks a bit like the norm of a vector, doesn't it? Well, the related cross-product *is* related to the area spanned between vectors, so I guess it makes sense then.

Matrix inversion

Going back to eq C.9 (yet again), we have a system of equations with variables **x**= (*x, y, z*) and matrix **A** such that **A** · **x** = **b**. We'll that's nice and all, but most of the times it's **x** that's unknown, not **b**. What we need isn't the way from **x** to **b** (which is **A**), but its *inverse*. What we need is **x** = **A**$^{-1}$ · **b**. **A**$^{-1}$ is the notation for the inverse of a matrix. The basic definition of it is **A** · **A**$^{-1}$ = **I**, where **I** is the *identity matrix*, which has 1s on its diagonal and 0s everywhere else. There are a number of ways of calculating an inverse. There's trial-and-error, of course (don't even think about it!), but also the way one usually solves linear systems: through row reduction. Since I haven't mentioned how to do that, I'll resort to just giving a formula for one, namely the 2x2 case:

$$(C.15) \qquad A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \qquad A^{-1} \equiv \frac{1}{ad-bc} \begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$$

This is the simplest case of an inverse. And, yup, that's a determinant as the denominator. You can see what happens if that thing's zero. Now, some other things you need to know about matrix inverses. Only square matrices have a chance of being invertible. You can use the determinant to see if it's actually possible. Furthermore, the inverse of the inverse is the original matrix again. There's more, of course (oh gawd is there more), but this will have to do for now.

Algebraic properties of matrices

A and **B** are M×N matrices; **C** is N×P; **D** and **E** are N×N. $\mathbf{e}_i$ are the column vectors of **E**. c is a scalar.

A + **B** = **B** + **A**
c·(**A** + **B**) = c**B** + c**A**
A·**I** = **I**·**A** = **A**
A·**C** = **C**·**A** *only* if M=P, and then *only* under very special conditions
If **E**·**F** = **I**, then $\mathbf{E}^{-1}$ = **F** and $\mathbf{F}^{-1}$ = **E**
$(\mathbf{A}^T)^T$ = **A**
$(\mathbf{A}\cdot\mathbf{C})^T = \mathbf{C}^T \cdot \mathbf{A}^T$
$(\mathbf{A}\cdot\mathbf{C})^{-1} = \mathbf{C}^{-1} \cdot \mathbf{A}^{-1}$
If $\mathbf{a}_i \cdot \mathbf{a}_j = \delta_{ij}$, then $\mathbf{A}^{-1} = \mathbf{A}^T$ (in other words, if the vectors are unit vectors and mutually perpendicular, the inverse is the transposed.)

Spaces, bases, coordinate transformations

The collection of all possible vectors is called a ***vector space***. The number of dimensions is given by the amount of numbers of the vectors (or was it the other way around?). A 2D space has vectors with 2 elements, 3D vectors have 3, etc. Now,

usually, the elements of a vector tell you where in the space you are, but there's more to it than that. For a fully defined position you need

- a base
- an origin
- coordinates

The vectors you're used to cover the coordinates part, but without the other two coordinates mean nothing, they're just numbers. A set of coordinates like (2, 1) means as little as, say, a speed of 1. You need a frame of reference for them to mean anything. For physical quantities, that means units (like km/h or miles/h or m/s, see what a difference that makes for speed?); for spaces, that means a base and an origin.

Coordinate systems
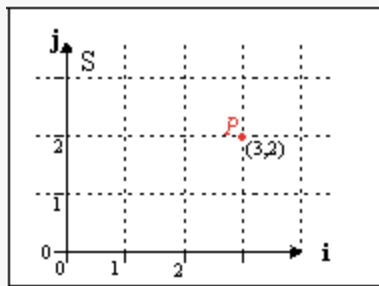
Fig C.5a: a standard coordinate system S. Point *P* is given by coordinates (3, 2).
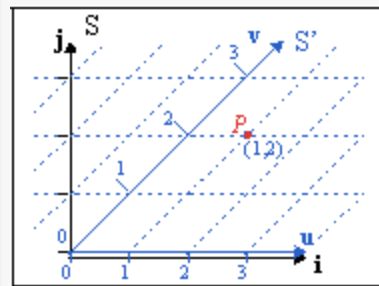
Fig C.5b, a sheared coordinate system S'. Point *P* is given by coordinates (1, 2).

Fig C.5a shows the 2D Cartesian coordinates system you're probably familiar with. You have an horizontal x-axis ($\mathbf{i}$ = (1, 0)) and a vertical y-axis ($\mathbf{j}$ = (0, 1)). And I have a point *P* in it. If you follow the gridlines, you'll see that x=3 and y=2, so *P*= (3, 2), right? Well, yes. And no. In my opinion, mostly no.

The thing is that a point in space *has* no real coordinates, it's just there. The coordinates depend on your frame of reference, which is basically arbitrary. To illustrate this, take a look at fig C.5b; In this picture I have a coordinate system S', which still has a horizontal x-axis ($\mathbf{u}$ = (1, 0)), but the y-axis ($\mathbf{v}$ = (1, 1)) is sheared 45°.

And in this system, point *P* is given by coordinates (1, 2), and not (3, 2). If you use the coordinates of one system directly into another system, bad things happen.

Two questions now emerge: why would anyone use a different set of coordinates, and how do we convert between two systems. I'll cover the latter in the rest of this article. As for the former, while a Cartesian system is highly useful, there are many instances where real (or virtual) world calculations are complicated immensely when you stick to it. For one thing, describing planetary orbits or things involving magnetism considerably easier in spherical or cylindrical coordinates. For another, in texture mapping, you have a texture with texels which need to be applied to surfaces that in nearly all cases do not align nicely with your world coordinates. The affine transformations are perfect examples of this. So, yeah, using non-Cartesian coordinates are very useful indeed.

Building a coordinate base

Stating that there are other coordinate systems besides the Cartesian one is nice and all, but how does one really use them? Well, very easily, actually. Consider what you are really doing when you're using coordinates in a Cartesian system. Look at fig C.5a again. Suppose you're given a coordinate set, like (x, y)= (3, 2). To find its location, you move 3 along the x-axis, 2 along the y-axis, and you have your point *P*. Now, in system S' (fig C.5b) we have (x', y') = (1, 2), but the procedure we used in S doesn't work here since we don't have an y-axis. However, we *do* have vectors **u** and **v**. Now if you move 1 along **u** and 2 along **v**, we're at point *P* again. Turning back to system S again, the x and y axes are really vectors *i* and *j*, respectively, so we've been using the same procedure in both systems after all. Basically, what we do is:

(C.16a)	$P = i \cdot x + j \cdot y$
(C.16b)	$P = u \cdot x' + v \cdot y'$

Now, if you've paid attention, you should recognize the structure of these equations. Yes, we've seen them before, in eq C.9d. If we rewrite our vectors and coordinates, to matrices and vectors, we get

$$M = [\ i \quad j\ ] = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad x = \begin{bmatrix} x \\ y \end{bmatrix}; \quad M' = [\ u \quad v\ ] = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad x' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

(C.16c) $\qquad P = M \cdot x = M' \cdot x'$

Vectors **x** and **x'** contain the coordinates, just like they always have. What's new is that we have now defined the coordinate system in the form matrices **M** and **M'**. The vectors that the matrices are made of are the **base vectors** of the coordinate system. Of course, since the base vectors of system S are the standard unit vectors, the matrix that they form is the identity matrix **M == I**, which can be safely omitted (and usually is), but don't forget it's there behind the curtains. Actually, there's one more thing that's usually implicitly added to the equation, namely the origin *O*. The standard origin is the null vector, but it need not be.

Eq C.17 is the full equation for the definition of a point. *O* is the origin of the coordinate system, **M** defines the base vectors, **x** is a coordinate set in that base, starting at the origin. Note that each of these is completely arbitrary; the **M** and **x** in the preceding discussion are just examples of these.

(C.17) $\qquad P = O + M \cdot x$

Last notes

It really is best to think of points in terms of eq C.17 (that is, an origin, a base matrix, and a coordinate vector), rather than merely a set of coordinates. You'll find that this technique can be applied to an awful lot of problems and having a general description for them simplifies solving those problems. For example, rotating and scaling of sprites and backgrounds is nothing more than a change of coordinate systems. There's no magic involved in *pa-pd*, they're just the matrix that defines the screen→texture space transformation.

Be very careful that you understand what does what when dealing with coordinate system changes. When transforming between two systems, it is very easy to write

down the exact inverse of what you meant to do. For example, given the systems S and S' of the previous paragraph, we see that $x = M \cdot x'$, that is **M** transforms from S' to S. But the base vectors of **M** are *inside* system S, so you may be tempted to think it transforms from S to S'. Which it doesn't. A similar thing goes on with the **P** matrix that the GBA uses. The base vectors of this matrix lie inside texture space (see fig 5 in the affine page), meaning that the transformation it does goes from screen to texture space and *not* the other way around.

The base matrix need not be square; you can use any M×N matrix. This corresponds to a conversion from N dimensions to M dimensions. For example, if M=3 and N=2 (i.e., two 3D vectors), you would have a flat plane inside a 3D world. If N>M, you'd have a projection.

D. More on makefiles and compiler options

- Introduction
- My standard makefile
- Common compiler flags

> **THIS CHAPTER MAY BE OUTDATED**
>
> This part may need an overhaul and some of the suggested tools or practices may be deprecated.

Introduction

Although I gave a quick introduction to makefiles and compiler flags in the setup section, a more complex look into these items may prove useful as well. So I'll present and explain the makefiles that Tonc uses in more detail, as well as some other little things about makefiles and compiler/linker options. I hope that this will give you enough ammo to understand the makefiles that are out there and allow you to figure out the more complicated aspects of the make process yourself. This page is hardly a substitute for the full documentation on the maketool make, the assembler as, compiler gcc and the linker ld, but it'll have to do for now. You can get the full documentation on these tools at GNU Manuals Online. You may also be interested in MrMrIce's make tutorial, which can be found in gbadev.org's documentation section.

By the way, I'm no expert at this stuff. I know a few tricks about makefiles and compiler options but that's it. If you have suggestions on improving my makefiles, do tell.

My standard makefile

What follows is the makefile for my int_demo demo. This is a moderately complex makefile, using the assembler, implicit rules and pattern substitution. The things you'll see here should be sufficient for most everyday makefiles. Two notes before we begin: this is a makefile for devkitARM. Instructions for getting it to work on DKA are indicated by comments.

```
#
# int_demo.mak
#
# makefile for a simple interrupt demo


# --- Project details ---
PROJ    := int_demo
EXT     := gba
UDIR    := ../toncllib

SFILES  := $(UDIR)/single_ints.s
CFILES  := int_demo.c gba_pic.c \
    $(UDIR)/core.c $(UDIR)/interrupt.c $(UDIR)/keypad.c $(UDIR)/vid.c

SOBJS   := $(SFILES:.s=.o)
COBJS   := $(CFILES:.c=.o)
OBJS    := $(SOBJS) $(COBJS)

#--- Tool settings ---
CROSS   := arm-none-eabi-                        # use arm-agb-elf- for DKA
AS      := $(CROSS)as
CC      := $(CROSS)gcc
LD      := $(CROSS)gcc
OBJCOPY := $(CROSS)objcopy


MODEL   := -mthumb-interwork -mthumb
SPECS   := -specs=gba.specs

ASFLAGS := -mthumb-interwork
CFLAGS  := -I./ -I$(UDIR) $(MODEL) -O2 -Wall
LDFLAGS := $(SPECS) $(MODEL)

#--- Build steps ---
build : $(PROJ).$(EXT)

$(PROJ).$(EXT) : $(PROJ).elf
    @$(OBJCOPY) -v -O binary $< $@
    -@gbafix $@

$(PROJ).elf : $(OBJS)
    @$(LD) $^ $(LDFLAGS) -o $@

#COBJS compiled automatically via implicit rules
#$(COBJS) : %.o : %.c
#    $(CC) -c $< $(CFLAGS) -o $@

$(SOBJS) : %.o : %.s
    $(AS) $(ASFLAGS) $< -o $@
```

```
# --- Clean ---
.PHONY : clean
clean :
    @rm -fv $(COBJS) $(SOBJS)
    @rm -fv $(PROJ).$(EXT)
    @rm -fv $(PROJ).elf
```

As you can see, I've divided the file into four sections: project details, tool settings, building and clean. I'll go through these in order of appearance.

1: Project details

```
PROJ    := int_demo
EXT     := gba
UDIR    := ../tonclibs

SFILES  := $(UDIR)/single_ints.s
CFILES  := int_demo.c gba_pic.c \
    $(UDIR)/core.c $(UDIR)/interrupt.c $(UDIR)/keypad.c $(UDIR)/vid.c

SOBJS   := $(SFILES:.s=.o)
COBJS   := $(CFILES:.c=.o)
OBJS    := $(SOBJS) $(COBJS)
```

These are all just variable definitions. Variables can be defined in two ways (see make manual, 7.2: "The Two Flavors of Variables"):

```
XX  = yy
AA := bb
```

The first flavour (`=`) is a *recursively expanded* variable; the second (`:=`) is a *simply expanded* variable. In either case, whenever you now write `$(XX)` the make tool will substitute it by `yy` . And yes, the parentheses are mandatory. The difference between the two can be made clear by looking what happens if you do this.

```
XX = $(XX) -c
AA := $(AA) -c
```

You would like this to behave as the C operator `+=` , but in the first case the expansion is done recursively, meaning that you get an endless loop. The second version does what you expect to happen. Simply expanded variables make things more predictable, which is a good thing. See the make manual for more details on this. Oh, in case you were wondering, the assignment operator is available for makefiles as well.

In this case I've defined variables for the project's name (int_demo), the extension (gba) and the directory where I keep all my utility routines (../libtonc). It's a good practice to do this, because you can modify and use it to suit another project without too much trouble.

The second part defines the source files (not the object files, but the actual C and assembly files) of the project. Note the use of `$(UDIR)` in many of the names. Note also that the definition of `CFILES` is split over two lines using a backslash (`\`). When you do this, though, make *absolutely* sure it's the last character on the line. If you put, say, a space behind it, you'll regret it. Some editors have an option with which you can show non-printable characters; try it if you suspect these kinds of errors (will work for the tab requirement as well).

And the third part is where it gets interesting. The form

```
$(var:a=b)
```

is called **substitution reference**, one of the many forms of pattern substitution. In this case it looks at variable *var* and if it finds the string *a* at the end of a word, it'll be replaced by string *b*. I've used this to turn the lists of .s and .c files into lists of object files. GNU Make is full of string-transformation commands such as this. Look at libtonc.mak for some others.

2: Tools settings

```
CROSS := arm-none-eabi-                           # use arm-agb-elf- for DKA
AS      := $(CROSS)as
CC      := $(CROSS)gcc
LD      := $(CROSS)gcc
OBJCOPY := $(CROSS)objcopy


MODEL   := -mthumb-interwork -mthumb
SPECS   := -specs=gba.specs

ASFLAGS := -mthumb-interwork
CFLAGS  := -I./ -I$(UDIR) $(MODEL) -O2 -Wall
LDFLAGS := $(SPECS) $(MODEL)
```

More variables. First, I list the tools I use for assembling (`arm-none-eabi-as`),
compiling (`arm-none-eabi-gcc`) and linking (`arm-none-eabi-gcc`). Note that I'm
using the same program for compiling and linking. You can also use the command
that does the actual linking (`arm-none-eabi-ld`), but if you do that you have to tell it
what standard libraries to use and where to find them. gcc does that for us, which
saves us a lot of hassle. To indicate it really is a different step conceptually, I'm using a
different variable name for the link-step. Now, in principle the variable names are
yours to choose, you can call them HUEY, LOUIS and DEWEY for all I care, but AS, CC
and LD are conventional, so you'd do the world a favour by sticking to that. And
there's actually a second reason why using these names are preferred, which I'll go
into later. Additionally, using a separate variable for the command prefix (the `CROSS`
variable) makes switching to another devkit easier. Abstraction is your friend.

The rest are lists of assembler, compiler and linker flags. I want to tell you what these
do later, since it has nothing to do with the make-process in itself. It's standard
practice to do something like this, though. Again, by using variables for this stuff
(especially with these precise names) rather than adding them to the actual build
commands, makes it easier to switch to something that requires other flags.
Abstraction is a very good friend.

3: The build commands

```
build : $(PROJ).$(EXT)

$(PROJ).$(EXT) : $(PROJ).elf
    @$(OBJCOPY) -v -O binary $< $@
    -@gbafix $@

$(PROJ).elf : $(OBJS)
    @$(LD) $^ $(LDFLAGS) -o $@

#COBJS compiled automatically via implicit rules
#$(COBJS) : %.o : %.c
#    $(CC) $(CFLAGS) -c $< -o $@

$(SOBJS) : %.o : %.s
    $(AS) $(ASFLAGS) $< -o $@
```

And now for the real work. The actual build process is composed of a number of rules. If you've forgotten what a rule looks like, here it is again:

```
target : prerequisite
    command
```

One thing to remember here is that the command *must* be preceded by a TAB, *not* spaces! Anyway, the commands will run only when the target is out of date. This is true when the target doesn't exist or is older than the prerequisites. By default, the first rule in the makefile starts the build-chain, but you can start at another rule in the command line (or the Project Settings). Let's trace through the rules one by one.

It starts at the `build` rule, which has one prerequisite, `int_demo.gba` . This has a rule too, and one that requires `int_demo.elf` , which in turn requires `the object  list` . The objects list is composed of two parts, `COBJS` and `SOBJS` . The percentage signs ('%') in their rules make them *pattern rules*. Taking `SOBJS` as an example, the rule says that for every file in the list that ends in '.o', the prerequisite is its '.s' counterpart. Here ends the `build` chain, as the sources have prerequisites. Now the commands come into play, in an stack-unwind manner.

In almost all the commands, you'll see unknown things with dollar signs: `$^` , `$<` and `$@` . These are *automatic variables*. These refer to the full prerequisite, a single item in the prerequisite and the target, respectively. Other things to not about some commands are the hyphen ('-') and the at sign ('@') in front of them. The '@' suppresses echoing that line. The hyphen lets make ignore errors. I'm using it in the gbafix command to keep the makefile running, even if you don't have the tool.

An observant reader may have noticed that the lines for compiling the C files have been commented out. So how can the files be compiled without a rule? Via *implicit rules*. For a good number of suffices GNUmake knows how to build them. For example, if you need an object file *foo.o* and *foo.c* is nearby, it'll use the rule

```
$(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@
```

There's an implicit rule for assembly files too, only it uses `AS` and `ASFLAGS` , which is why I used those names. You can find a full list of implicit rules and the variables they use in the make manual.

4: cleaning up

```
# --- Clean ---
.PHONY : clean
clean :
    @rm -fv $(COBJS)
    @rm -fv $(PROJ).$(EXT)
    @rm -fv $(PROJ).elf
```

This rule is separate from the others and is used to remove the output and intermediaries of the project (but not the utility objects, because they may be used in another project as well). It's really simple: rm is the command for removing stuff, the flags tells it to keep going even if the file doesn't exist (`-f`) and to display what it's doing (`-v`). And that's it. Well, almost. There's one more thing, namely the `.PHONY` directive. Remember that I said that the commands are only run when the target doesn't exist or is older that its prerequisites. Since the target (clean) doesn't exist, it's always out of date and the commands always run. But what happens if there *is* a file

called clean? Because there are no prerequisites the commands will never run. The `.PHONY` directive is used to indicate that the target is a target in name only and that the commands should always be executed.

There's a lot more fun to be had with makefiles. You can use makefiles that run other makefiles (which is actually how tonc.mak is set up) or include them in other makefiles. This last one can make your life a lot easier. For example, by proper use of variables, steps 3 and 4 will rarely change between projects. This means that you could put them into a master makefile and include them in all your project-makefiles, in which you will only have to write down the things that are really specific to the current project (for an example of this, see HAM). Abstraction wants to have your babies.

With the pattern substitution and wildcard rules you can practically make makefiles that write themselves! (see the devkitARM sample code). The full extent of makefile capabilities it beyond the scope of this tutorial, but trust me, there's a lot more cool stuff here.

Common compiler flags

Knowing how to write a working makefile is only part of the problem of getting the GNU tools to work. What's even more important is knowing what options you can use with the assembler, compiler and linker. In an IDE, you can enable these by selecting them in check- and list-boxes and such. No such luck for command line tools, though, here you have to set all the options by including certain flags. The key is knowing which flags to use. I'm not going to list each and every one of these since there are literally hundreds of flags. But I am going to list the ones you're most likely to see in GBA programming.

- `-c` : (gcc) Compile to object file, but do not link.
- `-E` : (gcc) Stop after the preprocessor stage.

- `-g` : (as, gcc) Generates debug-information for the gdb debugger. Haven't used myself this yet.
- `-Idir` : (gcc) Add the directory dir to the list of directories to be searched for header files. (That's a capital 'i', by the way)
- `-llibrary` : (gcc, ld) Search the library named *library* or *liblibrary.a* when linking. Important libraries are libm (math library), libgcc, libc and libstdc++; the last three are linked automatically when you use `gcc` as a linker, rather than calling `ld` directly. And that's a lowercase 'L', by the way. "lI1", "oO0", I do so hate the Latin alphabet sometimes.
- `-Ldir` : (gcc, ld) Add directory dir to the list of directories to be searched for code libraries.
- `-M` : (gcc) The family of `-M` flags generate dependency information for header files. Normally when you create rules, you only mention the source files, which are recompiled when they've been modified. But when you modify the headers that that file includes, the file itself is still considered up-to-date. You can either create a rule for the headers yourself or let make do it for you with these flags. Unfortunately, I haven't been able to make them work for me yet.
- `-Map mapfile` : (ld) Creates a ***map-file***, which indicates where the linker puts your functions and global variables. Since it is a pure linker option, you need to use -Wl,-Map,filename when linking with gcc.
- `-marm, -mthumb, -mthumb-interwork` : (as, gcc, ld) Indicates the CPU model to write object files for (ARM or Thumb). The default is ARM. With `-mthumb-interwork` you allow mixing between ARM and Thumb code, which you'll want to allow for even when you're not actually using it. This flags it actually *required* under devkitARM.
- `-nostartfiles` : (gcc, ld) Do not use the standard system start-up files when linking. If you want to link a custom *crt0.o* you want this *so* bad. (Whether you want a custom crt0.o is another matter, though.)
- `-o file` : (as, gcc, ld) Place output in file file.
- `-Onum` : (gcc) Enables optimisation level `num`, where `num` is usually `g`, `1`, `s`, `2`, or `3`. If you want to use inline functions, you need at least one level of

optimisation. See the gcc manual for details.

- `-S` : (gcc) compile, but not assemble. This gives you an assembly file of the C file you just compiled. Very useful for finding out how ARM assembly works, you should do this at least once.

- `-specs=specfile` : (gcc) use specfile to determine what switches need to be passed to gcc's subprocesses (`as` , `cc1` , `cc1plus` , `ld`) instead of the default specs. (gcc.info, line 5556. Fer IPU's sake, people, don't you guys read manuals? It's only 26k lines you know).

- `-T scriptfile` : (ld) Use scriptfile as the linker script. (Like Jeff Frohwein's lnkscript.)

- `-Wall` : (as, gcc) Enable common warnings. Options of the form `-Wfoo` are used for all kinds of warnings actually.

- `-Wl,opts` : (gcc) passes options to the linker; opts is a comma-separated list.

E. Make via editors

- [Introduction](#)
- [Make via ConTEXT](#)
- [Make via Programmer's Notepad 2](#)
- [Make via MS Visual C++ 6](#)

> **THIS CHAPTER MAY BE OUTDATED**
>
> This part may need an overhaul and some of the suggested tools or practices may be deprecated.

Introduction

As good as makefiles are, they're still command-line driven processes, with all the problems attached to it. If you're in a Unix/Linux environment those problems are usually fairly small, but on a pure Windows system we have to work in a DOS-like Windows command prompt, which has a number of very ghastly flaws that can make your life miserable: not only is wringing through the directory structure not much fun, the non-intuitive way to copy/paste text and the inability to arrow through your commands to fix a typo are somewhat annoying too. Another thing that will wear your patience very quickly is not being able to scroll through the list of compilation errors that speeds across your itty-bitty Windows command prompt like a thundering herd of rabid elephants in a China shop. And you *know* it'd all be okay if you could just find the one at the start of the list and fix that. Now, you can get around the gross inadequacies of a Windows command prompt by using an MSYS-box instead. You'd still need to learn how to use the Bash shell to make the most of it, though. And you'd still have the extra window for the command line box.

Fortunately, there are ways to avoid any kind of command line box altogether. Unless you're using something as dreadful as the standard Windows Notepad, there is a good chance you can run make or any other tool directly from your code editor. In this case, I'd like to take a look at three of them

- ConTEXT
- Programmer's Notepad. Yes, the one that comes with devkitARM.
- Microsoft Visual C++

ConTEXT and PN are basically advanced forms of text editors, of which there are quite a few. Most of these will allow tabbed files, search&replace, customizable syntax highlighting, macros and shell commands. If you're still using Windows Notepad for, well, *anything* really, you owe it to yourself to download one of the more advanced text editors and use that as a replacement. The glorified edit-box known as Notepad should not be allowed anywhere near any kind of plain text file unless there is no way around it. Every one of the programmers' editors you can google up is likely to be superior to Notepad in every single way, and some of them even allow you to replace the actual notepad.exe. While this has become harder since Windows XP Service Pack 2 because system file protection keeps resurrecting it, in my opinion it's well worth the effort to shoot it down permanently.

Ahem, sorry about that. Sometimes I get a little carried away when I remember how much the standard Windows tools suck at times. Anyway, onto the show. In the remaining part of this chapter, I'll show how you can get ConTEXT and PN to run make for the currently open makefile. The last section of this chapter will cover setting up MSVC for the job. If you're not interested in any of this, feel free to skip to the next chapter at any time.

Make via ConTEXT

ConTEXT is a lightweight free text editor that I use for most of plain text editing. It can do all the things that programmer's editors are supposed to do, it has a Notepad

replacer and a tool that allows me to export code to an html format, which has been very useful indeed for writing tonc. It does have one or two minor flaws, but none that I particularly mind.

The shell commands manager can be found under Options->Environment Options…->Execute Keys (fig E.1), and works on an extension basis. In my case, that means .mak. ConTEXT allows 4 commands per extension, and I'm using F9 to make the 'build' target and F10 for a clean operation.

F9 : make build

- **Execute:** `make.exe` (add full path if necessary)
- **Parameters:** `-f %f build`
- **Capture output:** yes

F10 : make clean

- **Execute:** `make.exe`
- **Parameters:** `-f %f clean`
- **Capture output:** yes

Be sure that the devkitARM and msys bin directories are in the system path, or context won't be able to find make.exe or the compiler tools.
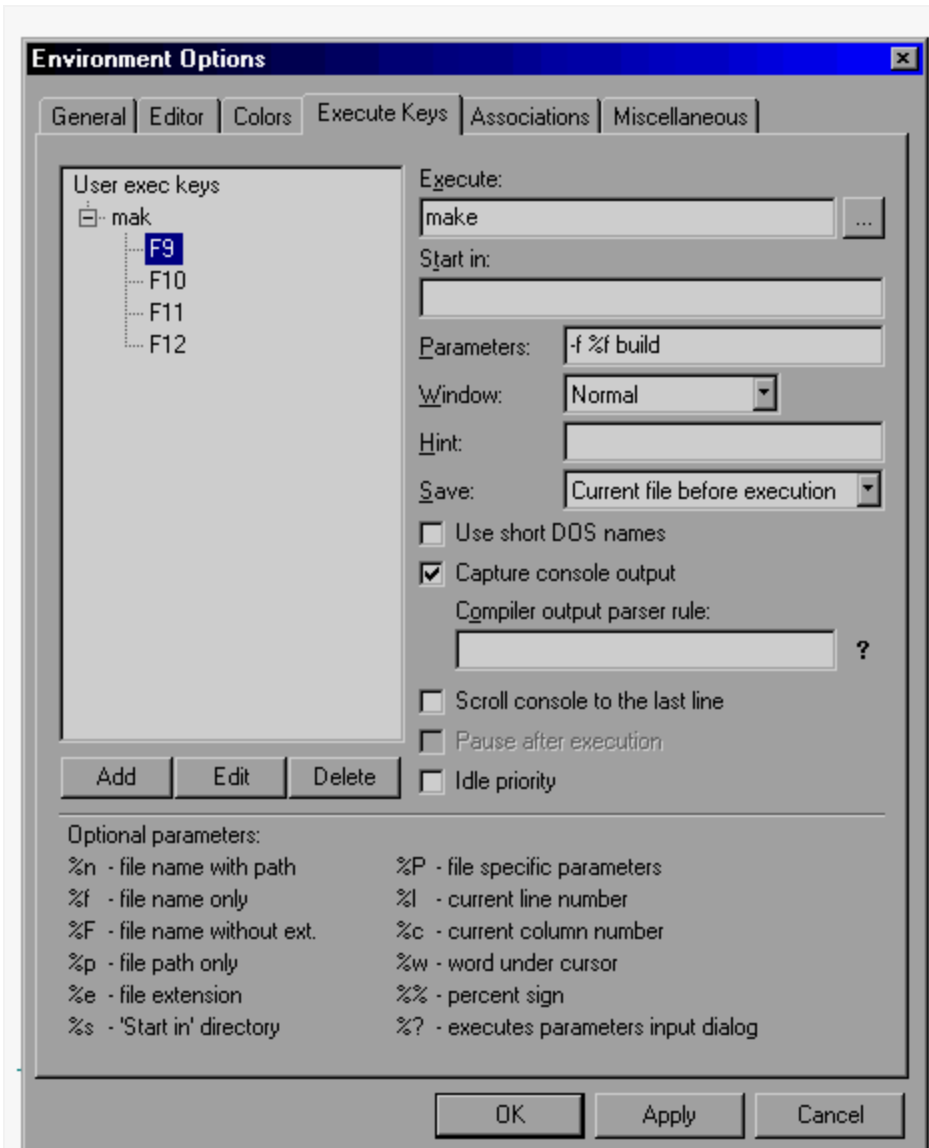
Fig E.1: ConTEXT shell commands.

Make via Programmer's Notepad 2

I never really knew about PN until it started coming with devkitARM, but it looks really good. I haven't used it that much myself, but only because I am still content with context. That said, PN is probably the better editor, and as it may come with the toolchain, chances are you'll have it already.

For all its benefits, I should say this though: by default, it seems to ignore the desktop color scheme. This may not sound like a big deal, but because the background color defaulted to a hard white, I literally couldn't even look at the thing for more than a minute. When I first tried to fix this in the options, it seemed that you could only change this on a type-by-type basis instead of globally. Took me a while to figure out I'd been looking in the wrong place `:P` all along. Look under Tools->Options->Styles, not under Tools->Options->Schemes.

To add commands for makefiles, go to Tools->Options->Tools (fig E.2), and select the 'Make'. Then add 2 commands for 'make build' and 'make clean'

F9 : make build

- **Name:** `mk build`
- **Command:** `E:\dev\devkitPro\msys\bin\make.exe`
- **Folder:** `%d` (the makefile's directory)
- **Parameters:** `-f %f build`
- **Shortcut:** F9

F10 : make clean

- **Name:** `mk clean`
- **Command:** `E:\dev\devkitPro\msys\bin\make.exe`
- **Folder:** `%d` (the makefile's directory)
- **Parameters:** `-f %f clean`
- **Shortcut:** F10

The name and shortcut can be different, of course; the rest should be as above. It is possible that you have to make sure the .mak extension is tied to the 'Make' scheme.
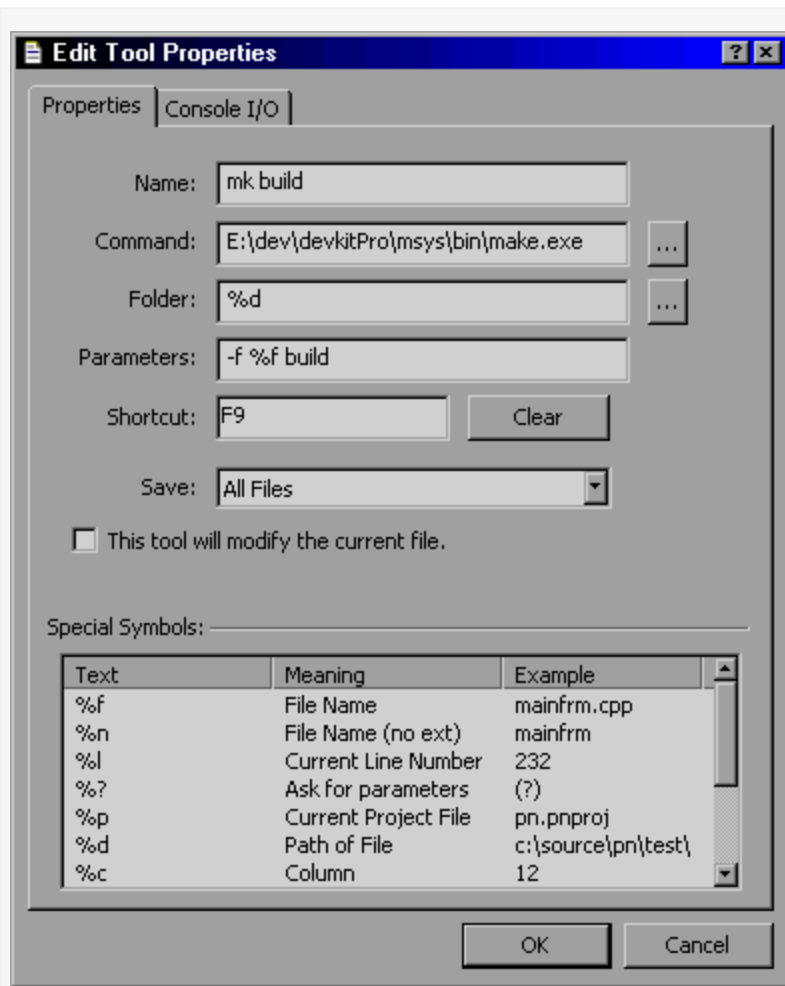
Fig E.2: Programmer's Notepad shell commands.

By adding make commands to your editor, you should be able to run the makefile of every tonc demo. If you encounter problems, you probably forgot to set a path somewhere.

Make via MS Visual C++ 6

I'm sure a lot of you will have gotten your hands on some version of Visual Studio one way or the other, officially, via school or … other methods. MSVC actually works with its own kind of makefiles and maketool called NMAKE, but we're going to ignore that one and use GNU's make instead. The instructions in this section work for versions 5

and 6, but I'm not sure about later versions. From what I hear, they changed a lot in those so if you have one of those you might have to do some digging of your own. I know that there are also add-ons that can create GBA projects via wizards, but again you'll have to find them yourself.

VC and makefile projects

Phase 1: setting the path

The first thing you need to do, if you haven't done so already, is setting the path so that Visual C can find the tools. Open the [Tools/Options] dialog and go to the [Directories] tab, then select the [Executable files] list from the [Show Directories for] box (see fig E.3 below). Now you need to add the bin directories of MSYS and dkARM. You can also set these directories to autoexec.bat as well. The devkitARM directory can also be set inside the makefiles themselves, but since I use 4 different computers to write Tonc, I prefer not to do this.
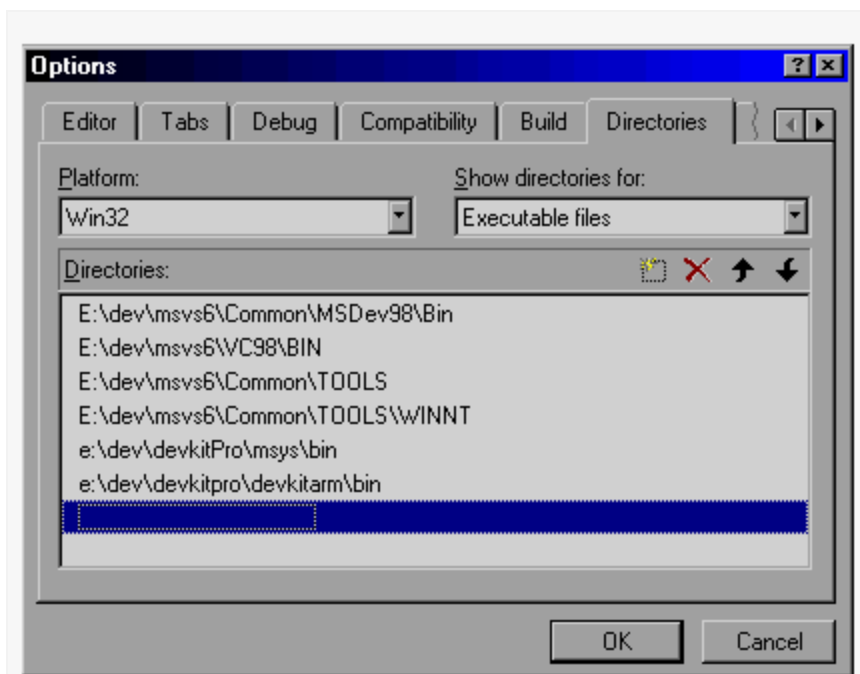
Fig E.3: adding the the dkARM paths to the executable list.

Phase 2: Creating a makefile project

The second step is creating a project/workspace that uses custom makefiles. This is called, what else, a *makefile project*. Go to the [Projects] tab of the [File/New] dialog (shown in fig E.4 below), select Makefile, give it a name and press OK. Mind you, this does *not* create the makefile, only the project! Also, the project's name I use here is 'tonc', change this to the name of your own project.
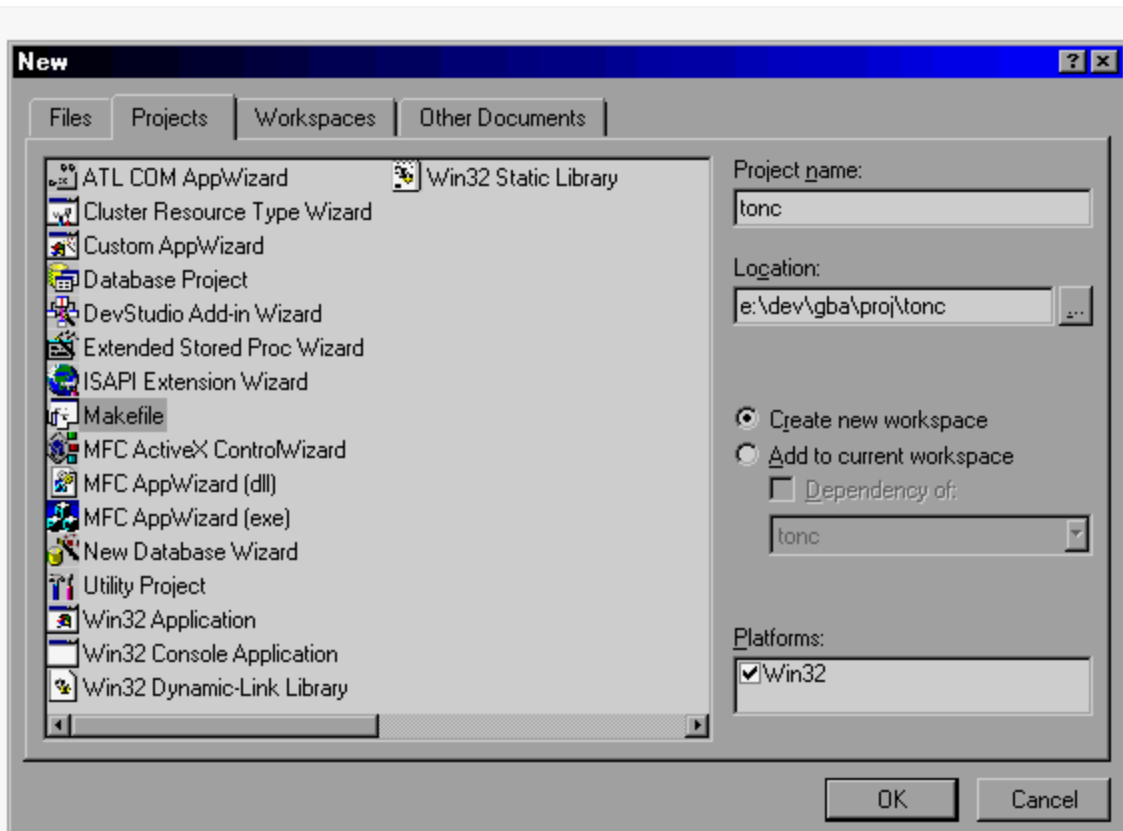
Fig E.4: creating a makefile project.

Phase 3: Profit!^H^H^H^Hject settings!

After you click OK, you will be asked to go to the Project Settings. Do so and you'll see the dialog from fig 6. The first thing you will find is the [Build command line] edit box. Right now, this reads something like

726 / 757

```
NMAKE /f tonc.mak
```

Change it to

```
make -f tonc.mak build
```

Why? Because we won't be using the standard VC make (NMAKE), but the GNU make (make). Why? Because it's free, platform-independent and usually comes with the devkit, making your project more portable, is more powerful and better documented as well. Why? Because … just because, OK? This is the command that is executed when you press Rebuild (F7). The -f flag says which makefile to use. Inside a makefile you can have multiple sub-projects; in this case the one called build is the active one.

The other settings aren't important for our purposes so leave them as they are. Yes, the output filename too; the makefile will take care of that. By the way, note that the workspace in fig E.5 shows three projects: tonc and libtonc for actual tonc stuff, and a vault project. A standard practice of mine to have one vault project where I can store source-files I don't want compiled but do want to have available for reference (such as templates and examples). All my workspaces have one and I can highly recommend them.
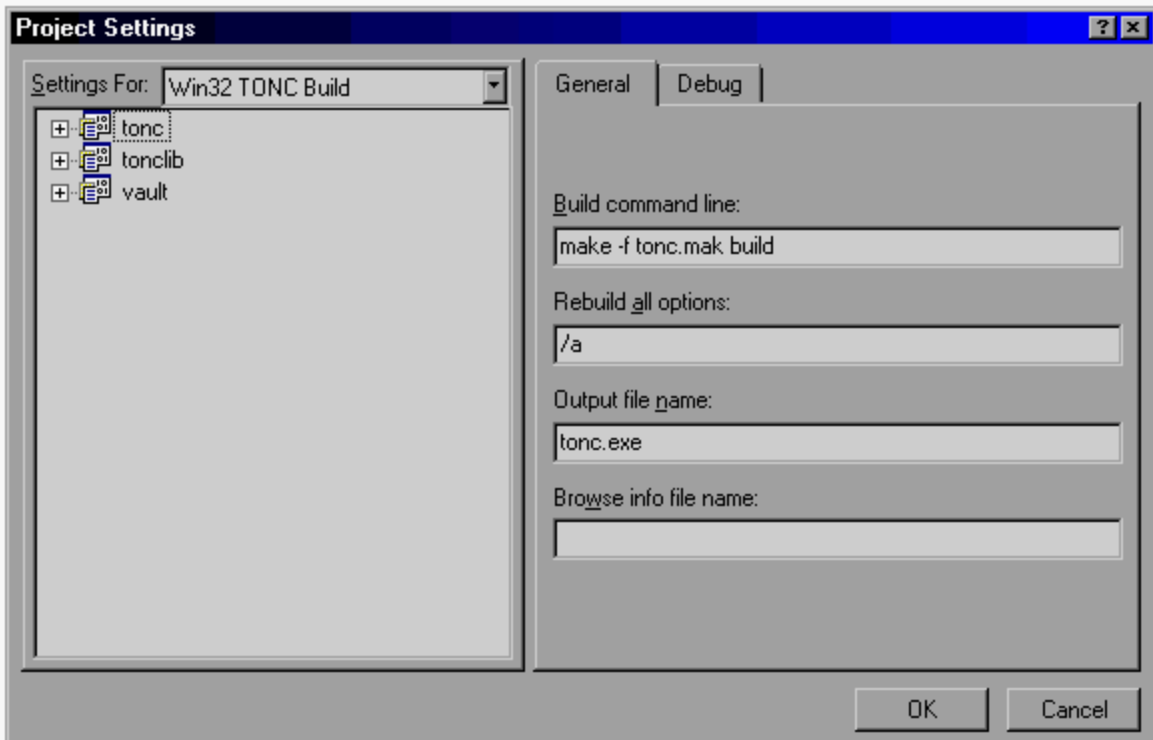
Fig E.5: project settings.

CONVERTING GCC REPORTS TO MSVC REPORTS

When you build a normal MSVC project, it will report and errors and warnings and double-clicking on these will bring to to the line that spawned it. This does not work for devkitARM because GCC has a slightly different reporting format.

```
# GCC error: {filename}:{line}: error: ...
foo.c:42: error: 'bar' undeclared (first use in this function)
# MSVC error: {dir}\{filename}(line): error ...
dir\foo.c(42) : error C2065: 'bar' : undeclared identifier
```

Because of the difference in line-number formatting, MSVC gets confused and can't find the line, or even the file. Fortunately, we can change this by piping the output of make through sed, the bash-shell string editor that comes with msys. To do this, change the build invocation to:

```
make -f tonc.mak build 2>&1 | sed -e 's|\(\w\+\):\([0-
9]\+\):|\1(\2):|'
```

The `2>&1 |` feeds the standard output of make to the standard input of the sed. The rest is a sed command that finds the parts before the first two colons, and converts them to the parenthesized format the MSVC expects. Note that tonc's build line is slightly more complicated because of its directory structure but the line above is what really matters.

Phase 3b: Build configurations

This one isn't strictly necessary, but may be useful. In Visual C++ you can have multiple *build configurations*, each with its own project settings. You're probably familiar with the Debug and Release Builds, but you can add your own as well with the [Build/Configurations] dialog (shown in fig E.6). The tonc project has five configurations, which all drive different targets in tonc.mak. `Build` builds the current demo; `Clean` removes all intermediary and output files (.O, .ELF and.GBA). In order to build/clean a specific demo you'd have to change the project settings or, preferably, set the `DEMO` variable inside tonc.mak to the name of that demo. `Build All` and `Clean All` run `Build` and `Clean` for all demos, respectively. The 'Utils' configuration creates the tonc library required for some of the later examples.
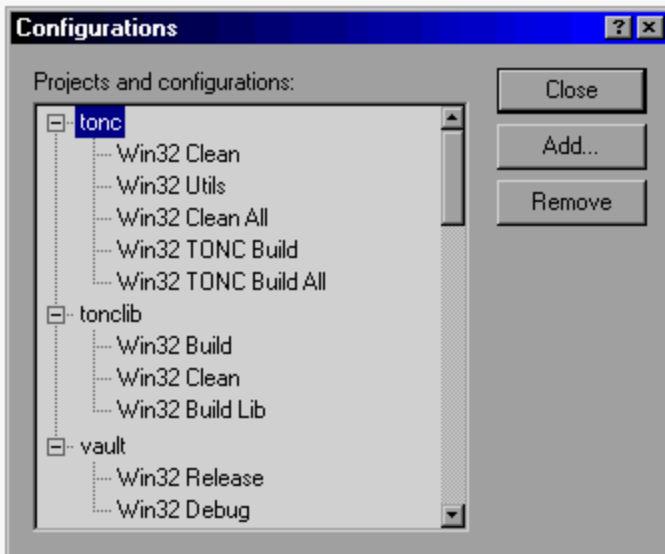
Fig E.6: Build Configurations.

And that's about it as far as Visual C++ is concerned. You still have to actually create the referenced makefile (tonc.mak in this case). You know how to create a textfile, don't you? Another thing to remember about makefile projects is that all build commands are inside the makefile; the files mentioned in the File Viewer are just for show and are not compiled by themselves like 'normal' VC projects.

EASY SWITCHING BETWEEN DEVKITS IN TONC.MAK

Tonc's makefiles are of such nature that each can stand on its own, but can also be called from a central makefile tonc.mak, with the `DEMO` variable. I've also put a `CROSS` (which houses the prefix) variable in there which overrides `CROSS` of the individual makefiles. Changing it in tonc.mak effectively changes it everywhere.

GETTING RID OF MSVC 6.0'S USELESS DIRECTORIES

It appears that Visual Studio 6 (and higher too?) has a very annoying habit of creating all kinds of extra directories for each project added to a workspace and for each project configuration. Directories that you probably never intend to

use, and *certainly* never asked for, and which clutter up your project. Removing them from disk doesn't solve the problem, because they'll just reappear merely by selecting the project/configuration.

grumble

Well, the good news is that for normal projects you can just remove them from the project settings, then remove them from disk and everything will be clean again. The bad news is that we're not using normal projects but makefile projects, which don't have the settings-tab in question. So what you have to do is go to the .DSP in a text editor, and remove everything resembling the following lines

```
# PROP BASE Output_Dir [DIR]
# PROP BASE Intermediate_Dir [DIR]
# PROP Output_Dir [DIR]
# PROP Intermediate_Dir [DIR]
```

No, I don't exactly know what I'm doing, but yes when you remove the directories now they *stay* gone. In fact, I'm pretty sure a lot of lines can be removed from the DSP, but as there is no manual for the commands in a project file, I'm not taking any chances there.

Now, if anyone does have a reference guide for DSP files, or can tell me whether this obnoxious behaviour is still present in later MSVC iterations, I'm all ears.

F. References and links

- General sites
- Documents
- Tools
- Books

General sites

Essentials

- www.devkitpro.org. Home of **devkitARM**, the toolchain of choice for GBA development. And NDS and more. Updated regularly and you can find a **libgba** and sample code here too.
- www.gbadev.org. GBA development hub. Tools, documents, tutorials can all be found here. A visit to the forum is highly recommended if you're just starting, whether you have problems or not. If you do have a problem, chances are you're not the first and that it has been solved before here. Just remember the rules of posting before you start a topic.
- Nocash. Martin Korth's site. You can find the immensely complete (though Spartan) GBATEK reference document and the No$gba emulator, both of which are insanely great.
- vba.ngemu.com, The **VisualBoy Advance** emulator. Not as accurate as no$gba when it comes to timings, but still very, very good, and has a more friendly interface and all kinds of cool viewers for tiles, maps, IO registers and the like.

Alternative dev environments

- www.ngine.de, Host of **HAM**. HAM is a full C developer environment for the GBA, complete with IDE, palette and map editor and, of course, compiler. There is also

an extension library called HEL with extra (and optimized) code. Taking a look at that would be a good idea.

- **DragonBasic**. If you don't like all the intricacies of C/asm, you might try this BASIC-like environment. The project is a little, uhm, asleep right now, though.
- Catapult. Don't know too much about Catapult, but from what I've seen, it seems to work a little bit like Gamemaker: you create images/sound and scripts that Catapult ties together into a ROM image. Catapult comes complete with graphic, map and sound editors, tutorials, samples, emulator and probably more.

Personal sites

A few sites of (high-ranked) forum-dwellers. These guys have been around for a while and you can learn a lot from playing their demos and browsing through their source code.

- darkfader.net. Darkfader's site, with information, tools, demos, code not only for GBA development, but many other systems as well.
- deku.rydia.net. DekuTree64's site has more than just the sound mixer; there's also some demos (with source) and tools like **quither**, a quantizer / ditherer for 16 color tiles.
- Headspin had put together this overview of various items, incluning the different compression routines and music players available.
- www.thingker.com. Scott Lininger's site with a number of demos, including **multiplayer** code, which seems very hard to come by.
- www.console-dev.de. Peter Schaut's site, with VisualHam, the HAMlib IDE; HEL, the HAM addon library; katie, a data-management tool and more.
- www.pineight.com. Site of gbadev faq maintainer, tepples. There are a number of interesting things here. Special mentions for **Tetanus on Drugs**, a zonked-out version of tetris (can't call it a clone as it is so much more), and **GBFS**, a file system for the GBA.

Documents

Tutorials

- www.belogic.com. Pretty much *the* site on GBA sound programming. Has info on all the registers, and a set of *very* complete demos.
- If you're looking for **C/C++ tutorials**, there seems to be some good stuff here
- DekuTree's sound mixing tutorial. Whereas Belogic shows the basics of sound programming, this sight guides you through the steps of making a sound/music mixer.
- www.drunkencoders.com. This is the new home of **the PERN project**, the original series of tutorials for gba development. PERN was set for a complete renewal, but that seems to have been deprioritised in favor of the DS, which you will also find a lot about there.
- jake2431 has been gathering **NDS / C / GBA tutorial links** on the gbadev forum:8353

Reference documents

- The **comp.lang.c FAQ**. Pretty long, but very useful if you're learning C as well as GBA programming.
- A document on **C coding standards**, one of many floating around. If you've based your code on any of the non-tonc tutorials out there you *need* to read this. The standard doesn't have to be followed religiously, but adopting most of them would be a good idea and would solve a lot of the bad habits the other sites teach.
- Mr Lee has a few things to say about optimization. These are simple optimization that cost nothing or little in readability.
- The **gbadev forum FAQ**. Essential reading, whether you're new or not. Bookmark it, make a local copy, print it out; I don't care, but get FAQed.
- The **GBATEK**. reference document. This is basically the GBA-coders' bible (only this one *is* a worthwhile read). The information density is very high and maybe a

little perplexing if you're just starting, but when you get the hang of it, it's pretty much all you'll require. Is also part of HAMLib's documentation.

- The CowBite Spec, a another reference document. At least partially based on GBATEK. Not as rich, but probably more understandable.
- www.gnu.org. **GCC documentation** in various formats. These sites have manuals on the GCC toolchains and other things. Get the files for the assembler (**AS**), compiler (**GCC**), linker (**LD**) and preferably the maketool (**make**) as well. The preprocessor manual (**cpp**) may be useful as well.

ARM docs

Naturally, the ARM site itself also has useful documents. Note that most of these are pdfs.

- miscPDF 8031. The **Arm Architecture Procedure Call Standard** (AAPCS). Explains how parameters are passed between functions. Required reading if you want to do assembly.
- PDF DAI0034A. **Writing efficient C for ARM**. Although it's written with ARM's own compiler in mind, some tips will work for other toolchains as well.
- PDF DDI0210B The big one: the complete **technical reference manual** for the ARM7TDMI.
- **Instruction set reference sheets**. ARM + Thumb combined.
- Support faqs on **alignment issues**: faqdev 1228, faqdev 1469, and faqip 3661.

Tools

Source code tools

If you're still using Notepad to write your GBA code, don't. Do yourself a favor and just … don't, OK? Though I personally use Visual C for writing code, there are some other very nice tools for the job, both in terms of general text editors as IDEs.

- **ConTEXT**. A while back there was a thread where someone asked for a replacement editor for Notepad since, and I quote, "Notepad SUCKS!". The name ConTEXT popped up a couple of times, and I find it very nice indeed, and not just for coding purposes. It allows for custom highlighters, integrated shell commands (to run makefiles for example) and attachable help files

-

Programmer's Notepad (PN). Good and versatile text editor. Comes with the devkitPro installation.

- **Eclipse IDE**. While I haven't had time to work with it firsthand, a good number of gbadev forum-dwellers swear by it. You can read how to set it up for GBA development in forum:5271.
- **Dev-C++**. Dev-C++ is another IDE that comes up often and maybe worth a look. forum:1736 has info on how to set it up, but it's an old thread so you may have to do a little extra work.

Graphics tools

Just as Notepad sucks for coding (and anything apart from the simplest text editing), MS-Paint is hell on Earth when it comes to the kind of graphics you need in GBA games. What you need is a tool that allows full control over the bitmap's palette, and MS-Paint fails spectacularly in that respect. So, I might add, does Visual C's native bitmap editor. And even big and bulky photo-editing tools like PhotoShop and Paint Shop Pro have difficulty here, or so I'm told. So here are some tools that do allow the kind of control that you need. Whatever tool you plan on using: **make sure it doesn't screw up the palette**! Some editors are known to throw entries around.

- **gfx2gba**. Command-line converter of graphics with interesting features such as tile-stripping, palette merging and supports all bitdepths and BIOS compression routines. Note that there are two converters named gfx2gba; you'll want the one my Markus. The HAM distribution includes this tool.
- **The GIMP**. Very complete GNU-based bitmap/photo editor.

- **Graphics Gale** is a very complete graphics editor. It has all the tools you would expect a bitmap editor to have, a proper palette editor and an animation tool.
- **Usenti**. This is my own bitmap editor. It may not be as advanced as Graphics Gale, but that does make the interface a lot easier. Aside from that it has some very interesting palette tweaking options like a palette swapper and sorter, and can export to GBA formats in binary, ASM and C code.

Map Editors

While the maps that I've used in Tonc were created on the fly, for any serious work you need a map editor. Here are a few.

- **MapEd**, by Nessie. Allows multiple layers, collision tiles and custom exporters. Yum.
- **Mappy**. This is a general purpose map editor which can be used for a lot of different types of maps
- **Mirach**. This is my own map editor, but lack of time means that I haven't been able to get all the tools that I wanted in yet `:(` .

Misc tools

- **excellut**. One thing you do not want in GBA programming is to call mathematical functions. You want look-up tables to get the proper values. Excellut sets up MS Excel to enable you to create any kind of LUT you can think of within seconds (well, OK, minutes). If you haven't created a LUT builder of your own (and maybe even if you have)it's definitely worth a look.

Books

- Douglas Adams, "*The Hitchhiker's Guide to the Galaxy*". OK, so this isn't exactly a reference, but recommended nonetheless. If only to know the significance of the number 42 and the origin of the Babel Fish.

- Edward Angel, "*Interactive Computer Graphics with Open GL*". Though this is a book on 3D, Lots of the linear algebra can be applied to 2D as well. Relevant chapters are 4 (matrix transformations) and 5 (perspective (Mode 7 anyone?)).

  Make sure you have the 3rd edition, there are too many embarassing errors in the second.

- George B. Arfken & Hans J. Weber, "*Mathematical Methods for Physicists*" If physics were an RPG, this would be the Monster's Manual. Chapters 1-3 deal with vectors and matrices in great detail.

- André LaMothe, "*Black Art of 3D Game Programming*". For the DOS-era, so may be hard to find. Deals with 3D programming under heavy hardware constraints (just like the GBA). Very nice.

- André LaMothe "*Tricks of the Windows Game Programming Gurus*". Another 1000+ page tome by Mr LaMothe; one of many), an excellent guide to game programming in general, and to DirectX in particular.

- David C. Lay, "*Linear Algebra and its Applications*". Nearly everything on my matrix page comes out of this book.

- O'Reilly pocket references for "*CSS*" and "*HTML*" by Eric Meyer and Jennifer Niederst, respectively. Absolute lifesavers for something like this site.

- Steve Oualline, "*How Not to Program in C++*". The cover features a computer sticking its tongue out at you; the first sentence of the introduction is "Pain is a wonderful learning tool". You just know this is gonna be good. This book gives you 111 broken code problems to solve, ranging from obvious to crafted by the Dark Lord himself. If you can't recognize yourself in at least half of these problems, you haven't been coding in C for very long.

G. Change Log

Dec 27, 2023 - Initial content port and mdbook setup

This pre-release includes all the contents from the TONC tutorial ported faithfully to the markdown format, rendered by mdbook. Some features are implemented with additional mdbook preprocessors.

The "Install" chapter also received a complete overhaul and the new version is included in this release. The original port of this part is anyway available in commit 7a3ac73.

The porting was curated by @exelotl @avivace @LunarLambda @pinobatch @mtthgn @copyrat90 @gwilymk .

Mar 2013 (1.4.1)

Maintaince update. also includes things from the errata page

- ⚠ Changed from `arm-eabi` to `arm-none-eabi` .
- ⊞ Little html fixes here and there. Thank Glod for directory search-and-replace tools.
- ⚠ `all code` : since GCC 4.7 broke my assembly functions, I've recompiled all code with the latest devkitArm (currently 40) for asm compatilibity. The examples and libtonc should all work again. I still have to adjust the text to match though.
- `asm.htm` : fixed non-matching variable names in data sections's code snippet. Thanks, Gdogg
- `gfx.htm` : removed a lost semicolon in the blending demo.
- `hardware.htm` : IO-ram upper limit was given as `0401:03FF` , which should be `0400:03FF` . Thanks, G M.
- code: Fixed links to grit for `m7_demo` , `m7_ex` , `tte_demo` .

- [objaff.htm 11.5](): fixed spurious `sina` and `cosa` calculations in `obj_rotscale_ex()` and `oac_rotscale()`. Thanks, dasi.
- GNU assembler manual has moved to [http://sourceware.org/binutils/docs/as/index.html]() (thanks, Joseph).
- Code snippet at §23.2.1:Basic operations. "x68 asm" should, of course, be "x86 asm" (thanks, Wladimir).
- Some of the memory map entries in §1.3 were … imprecise (thanks Pius).

Oct 2008 (1.4)

… Or maybe not. Silly little errors.

- ⚠ text: the `se_index_fast()` function in [regbg:map-layout]() was wrong; the second condition should have used `` `(bgcnt&BG_REG_64x64) == BG_REG_64x64` ``. Fixed.
- ⚠ text: the `ldm` example in [asm:memory]() did not list the right values for `ldmda` and `ldmdb`. Fixed.
- ⚠ code: removed `void*` arithmetic in `tonc_surface.h` (hopefully) and fixed `berk.c` from the timer demo so that it compiles now. Thanks for noticing, elwing and Ealdor.
- Fixed a few random typos here and there

May 2008 (1.4)

- ⚠ text: Last batch of spelling/grammar fixes. Thanks guys. Especially Jake.
- text: changed the stuff on #include for less hyperbole and more explanations.

I think that'll be all then.

May 2008 (1.3)

- ⊞ code: All demos that use text now use TTE to do it. Variable width text just looks prettier than fixed width.
- ⊞ code: Some of the advanced demos use grit to convert the graphics.

- ⭐ text: added TTE chapter with information about creating (fast) text renderers for every occasion.
- 🔲 text: I've revamped the setup chapter. It now covers template makefiles and some potential problems with the installation.
- 🌐 text/code: the irq chapter and its demo uses the new master ISR.
- ⚠️ text: Fixed the brown text in the pdf. For anyone who has the same problem with CutePDF, go to `CutePDF printer->Properties->Paper/Quality->Advanced->Graphic->Image Color Management` and make sure `ICM Method` is not set to `Host system`. How silly of me not to look there first.
- text: Finally removed the obsolete section for IRQ-handling with older devkits.
- ⭐ libtonc: There is a new system for text called TTE. It's pretty cool. Read more about it in tte.htm.
- ⭐ libtonc: New rendering functions. There are now 'TSurface' structs defining a rendering surface and basic primitive renderers for different surface types. Functionality includes: pixel, line rectangle renderers, a blitter and floodfill. Key surface types: 16bpp bitmap, 8bpp bitmap and 4bpp tiles.
- ⭐ libtonc: Color adjustment functions. Fading, blending, brightness and more.
- ⭐ libtonc: Added tonccpy and toncset, memcpy and memset replacements that actually work for VRAM as well.
- ⭐ libtonc: Put libtonc documentation online: http://www.coranac.com/man/tonclib/index.htm.
- ⭐ libtonc: Added `tonc_libgba.h`, a header with most of the libgba constants and functions names mapped to tonc equivalents.
- 🌐 libtonc: Changed the master ISR to one that doesn't automatically enable interrupt nesting. It's a bit of a downgrade, but it's probably more appropriate. This shouldn't affect anyone that didn't use nested interrupts. The old version is still available, it's just not the default.

Dec 2007 (1.3b)

- ⭐ Upgraded the recommendations section with a longer list and examples. If you've read other tutorials then **please read this**!
- ⚠️ More spelling and grammar fixes (thanks Patater)

- ⚠ Fix in the template makefile for c++. It's `-fno-exceptions` , not `-fno-expections` , you silly boy. (Thanks muff).
- 🌐 All projects now default to cart-boot, not multiboot. This is partially because multiboot doesn't work in devkitPro r21 (at least, not directly), but also because that's how it's normally done anyway.
- Changed `git` to `grit` where appropriate. Also fixed the download links of everything to point to the new site.

Feb 2007 (v1.3b)

As every programmer knows, you're supposed to write down the changes you make while making them. As every programmer also knows, this has a tendency to be forgotten `^_^;;` . I probably missed a few things here.

Text:

- ⭐ There's now a PDF version too, made by CutePDF. It's a nice tool, but apparently sometimes messes up pictures a little. Page-breaks also occur in unfortunate places, but this is a browser problem. It's *supposed* to be countered by CSS's `page-break-inside` , but I guess that's not widely supported yet. If anyone knows of a potential fix, let me know. Additionally, if anyone knows of a html→PDF converter that keeps headers for the table of contents, I'd be very interested in that. Mind you, it needs to be able to print a 1.4 MB file, and print it **correctly**! Some PHP html2pdf tools don't render correctly. Word/OpenOffice are probably no good either, as they have a problem with floating divs and `pre` tags. Also, Word nearly crashes on reading the file. Hehehehe.
- ⭐ hardware: GBA pictures and capabillity description. Anyone have a GBA Micro pic I can borrow?
- ⭐ first: hardware pictures.
- ⭐ bitmaps: new demo discussion for drawing primitives in modes 3/5. The page flipping demo has been moved forward in the chapter, and the mode 3/4/5 demo moved back to after the data discussion.

- ⭐ objbg: note and pic on reading tiles as bitmaps, as this this still happens to some occasionally.
- ⭐ regbg: pic to show what the offset registers really do.
- ⭐ affine: added inverse 2x2 matrix equation.
- ⭐ affbg: new structs for affine backgrounds, plus new typedefs and a very nifty method of initializing the affine parameters.
- ⭐ mode7ex: across the board upgrades and new stuff. It now uses proper graphics, making everything look a lot nicer. New background, new fade, sprite rotation-animation and sorting and different methods of motion.
- ⭐ asm: the proper form of the chapter is materializing. New structure of sections, beginning with a one on general ASM. More examples and more ways of doing the same thing for comparative purposes. Has a section on common constructs now too.
- ⭐ New subsection on linear interpolation of luts.
- ⊞ Chapter indexing. All references are how of the form 'ch.foo'.
- ⊞ Some chapters have been renamed. *tonctonc* is now *intro*, *toncmake* is now *makefile*. Also, *luts* has been merged into *fixed*, and the parts on makefiles and editors in *setup* has been moved to a separate file called *edmake*. Might move that into *makefile* too.
- ⊞ All register and register-like tables now use alternating background colors for easier reading.
- ⊞ regobj: Different structure for `obj_demo` discussion.
- ⊞ regbg: deleted `BGINFO` stuff, as it was never used much and impractical to boot.
- ⊞ regbg: New graphics besed on Super Metriod's Brinstar instead of the original Norfair. 's prettier now. Also rearranged stuff.
- ⊞ affine: updated 'finishing up' for new routines.
- ⊞ dma: discussion of upgraded DMA demo
- ⊞ interrupts: discussions of the new and (much) improved interrupt handler and its demo.
- ⊞ Inline functions for fixed-point functionality.
- ⚠ Every chapter has been checked for spelling and grammar. Again. Sigh.

Code:

- ⭐ libtonc: All new libtonc, with new file structure. All files are prefixed with `tonc` so the don't interfere with outside files. The types, memory map and register #defines are centralized in *types*, *memmap* and *memdef*. The main file to include is now `tonc.h`.

- ⭐ libtonc: Doxygen comments all around. The resulting documentation can be found in `tonclib.chm`.

- ⭐ libtonc: a few of the new items. A brand new interrupt handler for nested, prioritized interrupts. Mode 3/5 line drawers. A new .12f sine LUT with support functions as well as lerping functions. All fixed-point macros are now inlines.

- ⭐ libtonc: The `BGINFO` struct and functions are gone. Wasn't worth much anyway. Also removed are internal OAM shadows; it's better that you can define them when needed and can save IWRAM by potentially storing them in EWRAM. All OAM functions now use general object pointers, rather than buffers.

- ⭐ libtonc: yet another Great Renaming. Among other things: The leading underscore for zero-#defines are gone. I thought it was a good way if guarding against potential unsafe operations, but they just look too weird to use. And there was much rejoicing. Some macros have lots their `_ON` prefix when it's obvious that that's what they do. OAM structs are now `OBJ_ATTR` and `OBJ_AFFINE` and supporting functions are now prefixed `obj_` and `obj_aff_`. `BGAFF_EX` is now `BG_AFFINE` and used in most affine BG functions. A complete list can be found in `tonc_legacy.h`, which you can #include to keep compatibility with older code.

- ⭐ projects: the structure of the projects hierarchy has been altered. The demos have been categorized as basic, extended or advanced, which correspond with the tonc-text parts. Basic demos are simpler, with simple makefiles. They are completely self-sufficient, which should help learning the ropes. The extended demos have more complete makefiles and make use of libtonc. The advanced

demos have devkitPro-like makefiles. As much as I'd like to, the actual DKP templates don't quite suit my purposes (sorry, Dave `:P`) so I rolled my own. The advanced demos also make use of assembly files for data.

The project folders also contain `.pnproj` files, which can be opened and run from Programmer's Notepad.

- ⭐ projects: New projects. `m3_demo` , for drawing in mode 3. There are also a couple of new ones in the `lab` folder. They don't have discussions yet, but they're worth checking out. `bigmap` should be of particular interest.

- ⊞ projects: Update projects. All projects have been updated to the new libtonc. The DMA, irq and mode 7 demos have had drastic changes in content. `dma_demo` is now about using HDMA effects, in this case making a circular window. `irq_demo` uses the new irq handler to uts fullests with nested interrupts and changing irq priorities. As for `mode7ex` , well, you'd better just see for yourself.

Jul 23, 2006 (v1.2.4)

- ⭐ Added a rather long chapter on ARM/Thumb assembly. This is still a draft version, though. Most of the content is there, but I still need to reshuffle sections and spell/grammer check the whole thing.
- And yet more spell fixes `>_<` .

Jun 3, 2006 (v1.2.3)

- ⊞ Changed makefiles and build instructions to use devkitARM r19.
- ⊞ All sections and subsections are now numbered, w00t!
- ⚠ Added alignment attributes to most structs, as those are now pretty much *required* if you want struct-copies to work properly. For more, see here

Apr 28, 2006 (v1.2.2)

- ⭐ Finally realized what caused the 1 pixel offset I've been seeing in affine objects sometimes (thanks NEiM0D). Updated affobj.htm and `obj_aff` for it.
- 🌐 Moved the new off-center affine object stuff to its proper place.
- ⊞ Some small sed usage to convert from GCC error-reports to Visual C++ format, based on this.
- ⊞ Now that my html auto-numbering system works (at least a first trial), text.htm is now de-reffed. Yay.
- Small changes to interrupts and gfx.
- Added Javascript to make the id attributes visible. Will probably add more later.

Apr 28, 2006 (v1.2.1)

- ⚠ Apparently no$gba doesn't like it if you use section mirroring, like I did for REG_INTMAIN and REG_IFBIOS. These now use the proper addresses.
- ⚠ Spelling fixes in intro.htm. Thanks again, Mick.
- ⊞ New makefiles for extended and advanced projects. It does mean that makefile.htm is now pretty much behind the times.
- ⊞ Updated setup.htm for devkitARM changes. Changed the figures a little too.
- ⭐ New chapter called the lab, where I'll place new stuff that's almost, but not quite, ready. Currently contains text for priorities and sprite sorting, and a discussion on affine transformations around a non-center reference point. Both come with new demos called `prio_demo` and `oacombo` .
- ⭐ Added instructions on how to run makefiles via context or PN in setup.htm.
- ⭐ Added gfx2gba and grit conversion instructions in a few places.
- More notes in bitmaps.htm's data section.

Probable upcoming changes

I intend to make a few changes in tonc's code. First, I'll try to decouple the code in the basic demos from libtonc, which should make them easier to understand as you won't have to browse through all the other stuff. Second, this will allow me to rework and

optimize libtonc, which is now hampered in some areas by me having to keep a number of things simpler than I'd like to. Now, this is what I'd *like* to do; I can't really tell when (if) I will get round to it.

Also, I have half a mind of changing the current DMA demo to this one, which simply looks a lot cooler, even though there's is a lot more magic going on. Meh, we'll see.

Mar 21, 2006 (v1.2)

More non-final updates. Quite a lot actually.

- ⚠ More typos fixed.
- ⚠ `bg_init()` never initialized the `BGINFO` position. Oops.
- ☢ Tossed chapter order around a bit. I've moved keys up to right after bitmaps, which is a much better place for it anyway.
- ⊞ Updated First Demo, Bitmap Modes, Regular sprites, Regular Backgrounds, Affine Sprites, Affine Backgrounds, Graphic effects, and Timers with full or nearly full code of their demos.
- ⊞ First Demo now has two demos, one purely with hardcode numbers (muwahaha!), and one according to more sound programming principles. Also described these in much more detail.
- ⊞ Added two demos to Regular Backgrounds, one of which introduces libtonc and its text functions, which will come back a lot lateron. Speaking of which …
- ⊞ Recoded `bld_demo` , `m7_demo` , `mos_demo` , `obj_aff` , and `tmr_demo` to use libtonc's text so that it's clearer what you're changing.
- ⊞ Replaced copiers/fillers with libtonc's `memcpy16/32` and `memset16/32` in most demos after its introduction.
- ⊞ Added field defines to a lot of register tables.
- ⊞ Restructed part of keys.htm for better explanations of the various functions I use.
- ✪ Added a nasty piece on division by reciprocal multiplication. Not for the squeemish.
- Added a simple template makefile and discussion for the second demo.

- ⭐ Added a subsection on tribool keystates. In fact, nearly all of the demos that might benefit from these have been altered to use them. One line of code instead of four lines, and faster to boot. Seems like a win to me.
- ⭐ Added a subsection on the proper build procedure, which was still missing from that whole section. This is pretty much **required reading** for anyone how has been following non-tonc tutorials and adopted their coding standards.
- Merged the fixed-point and LUT chapters, and rewrote most of both.
- jake2431 has been gathering a lot of useful links in this thread: forum:8353. If you're new to C and/or GBA/NDS programming, I recommend you check it out.

Jan 27, 2006 (v1.1)

Heh, so that wasn't not the final update after all `:P`.

- Added a little note to setup on how to get rid of them useless directories that MSVC 6.0 insists on creating all the time.
- ⚠️ Fixed devkitARM URL and revision. (don't know why I bother with that, though, as there will be a new version the second I post this. Gawddammit, Dave, quit it! `>_<`)
- ⊞ More code in the text. At least for the earlier pages.
- Two new chapters: one on Text system fundamentals and producing beeps. The latter isn't quite finished yet, but should be enough to get you going. There are 5 new demos that go with these: 4 for text, one for sound.
- 😵 More name changes. This time in demo-names only, though, so don't worry there.
- Added/moved a section on How to deal with data. This explains some of the nastiness you may or may not encounter. But if you do, it's nice to know why they happen and how to fix it, no?

Jun 28, 2005 (v1.0)

Final update. Probably. Not because I'm done with this thing, but because there is so much to change and to add that it's easier to start from scratch again. When I started tonc I still knew very little about GBA programming and tried to do the best I could as I

went along. Now I'm a little older and wiser (well, older at any rate), know a lot more about proper procedure, what's useful to have and what isn't and also where people can get stuck on (thank you newbie forum dwellers for all your questions!) From the ideas I'm having, tonc 2 will be a *lot* bigger, better, and have more explosions! Errr, demos. Tonclib will get a major overhaul with new names and new, optimised functions including text for all modes, memory routines and more. But it'll take a while to get there, so I thought I'd update the original one last time.

- ⚠ Many, many spelling and grammar fixes. Too many in fact. C'mon people, tell me about these things!
- ⚠ `DMA_SRC_RESET` is `0x01800000`, not `0x00600000`. This is what made the outcome of `dma_demo` so weird. Also fixed `sbb_aff`'s black cross-hairs, which had its *x* and *y* values swapped in OAM. Stupid attribute x,y order.
- ⊕ Name changes. Lots of them. This partially falls under keeping to GBA community standards (`OBJ_ATTR`, charblock, screenblock, swi naming) and other classification issues (`DCNT_x` for `REG_DISPCNT` and such; prefix underscore for bit-defines that are zero, trust me this is a good thing). I've taken the liberty of creating a `legacy.h` that redefines all those old names into the newer ones so that you won't have to do all the renaming yourself if you don't want to. The older names are depreciated, though. This renaming is only part of the full tonc2 renaming, but I can't do functions yet because that *would* break old code.
- ⊞ Some small functionality changes. Most notably, `key_poll()` now *already inverts* `REG_KEYINPUT` (formerly `REG_P1`). This is a good thing, because now the synchronous functions will actually make more sense. Also, `m4_plot()` (formerly `_vid_plot8`) really does plot per pixel, not per two.
- ⊞ The memory routines `memcpy16/32` and `memset16/32` are optimised in assembly, and probably the fastest you'll come across. Rivals the speed of `CpuFastSet`, but none of the alignment / size requirements.
- ⊞ `swi.s` has calls for all BIOS routines. Some extras have been relocated to `swi_ex.s`

- Added `x_BUILD` macros for setting bit-flags in clusters. May be useful, maybe not.
- Added rectangle drawers for the bitmap modes. Fairly optimised.
- ⚠ Fixed list-margins for Firefox. Or rather, fixed list-margins to what the standard requires, but which MSIE doesn't follow. (Now if I can only figure out what to do about that <col> tag)
- ⊞ Restyled the register tables.
- ⊞ I finally realised how I could do matrices in pure html instead of images so pretty much *all* equations are now html. I expect it's quite close to MathML, but since MSIE doesn't support it natively and I don't want to worry you with an extra download (which may not even work on older versions), this'll do for now. (Now if you'll excuse me, I going to lie down to get the feeling in my brain back)
- ⊞ All sections, equations, tables etc now have id's for linking too and (maybe) automatic numbering if I figure out how.
- ⊞ `int_demo` now uses a separate file for the direct isr and makes proper use of sections and ARM/Thumb code. See the demo description for more.

That's about it I think, but it should be enough. I hve bits and pieces of the tonc2 text, examples and lib, though maybe not in their final forms. They are available, but only on request. If anyone has suggestions or requests I'll see what I can do. This also goes for mistakes (the ones that the compiler/linker can't catch) you've made that you think others might make too. I know a good number of them already from the forum (like that you should NOT use bytes or halfwords for local variables, since it can really kill performance, `int` or `u32` only, *please!*. Pretty please. With sugar on top. And frosting and whipped-cream.) Don't need to know every little thing though, especially if it's already covered by the well-done gbadev forum FAQ or already covered in here somewhere.

If anyone knows how I can keep track of all the header/equation/figure numbering automatically (without CSS2, which isn't properly supported by MSIE `:(`) that would be *very* helpful. Actually, the numbering itself isn't the problem, *referencing* them is.

Also, I could use more real-life examples of tile-map/sprite collision detection *and* response. I know the bounding box stuff and the basics of detection (even pixel-perfect), but no matter what I do I seem to be getting stuck on some of the particulars of diagonal movement and when things move at more than one pixel/frame. I'd very much like to see how it's done in real platform games for complex scenes that have multiple sprite-sprite and sprite-background collisions, not just single sprite-bg.

Dec 5, 2004 (v0.99.6)

Added the numbers page about number systems, bits and bit operations. I should warn you that it's rather large. It's been a while since I added something and I think I got a little carried away `:\`. I may break it up into smaller pieces later. Maybe.

DragonBASIC is in the process of being transferred to a new domain, so the old URL is invalid now. At the moment you can still find the forums here, but the compiler itself is still in limbo for the time being.

Aug 3, 2004 (v0.99.5)

- ⚠ Made some minor corrections all round. The first mode7 demo and page now use a different name for the camera position, so it won't clash with **v** from mode7d.
- Added a rudimentary text demo in the form of `txt_demo`.

I think this will be the last update for a while, for a number of reasons. Firstly, I think I have to actually use some of this stuff, to see what's wrong with it. Secondly, I think I may have to put in some more work into a converter and how to add pure binary files to the demos in a friendly way. Thirdly, PERN's back with a vengeance. As such, there seems little point in developing Tonc any further right now, as it seems that the new PERN is going to be very, very complete.

Jul 16, 2004 (v0.99.5)

- ⚠ Fixed `aff_rotscale2` , which should *not* have shrunk the source-angle, but rather a copy of it. Defined `MAPBLOCK` to contain 1024 (=32*32) tegels, not 512. Was confused with tile-blocks.
- ⊞ Made a *lot* of small changes to tiles/map functionality. All map/tile structures are now simple typedefs so you can access their internals via a simple array-access rather than (inline) functions. The inline functions themselves have been removed.
- Changed then `BGINFO` struct a bit, and added some map functions.

I'm sorry if any of these changes causes you any inconvenience, but I think it's better in the long run.

Jul 11, 2004 (v0.99.4)

- Deprioritized MSVC makefile projects in setup.
- Mopped up several minor and not so minor errors in mode7ex. Well, I did say there'd be some I hadn't discovered yet.
- ⊕ Renamed the interrupt requests for registers `REG_DISPSTAT` from `X_INT` to `X_IRQ` , which is more proper.
- ⊞ Modified swi.htm to show how to use pure assembly for this purpose and added a small section on the aapcs. Also renamed some of the affine structs and functions; you have been warned.
- And yet more typo fixes, where the h311 do they keep coming from? I swear if I find one more "it's"/"its" mixup I'm going to scream. [Later that day] Right, that's it: *AAAAAAAARRRRHRHRHRRGGGHHHH!!!*
- ⚠ Fixed a number of rot-scale equations that had the rotation and scaling ops wrong in the intermediate steps. Oops.
- ⚠ Fixed a window control macros (forgot some shifts). Should work properly now. Should.
- Added `geom.h|.c` to the library, as I intend to use points and rectangles more often. Also added `ABS` , `SGN` and `SWAP` macros.

- All multiboot demos (i.e., all of them) now have the extension `mb.gba` to indicate them as such.
- Renamed `key_pressed()` to `key_hit()`, which should cause less confusion about what the function actually does (thank's^H^Hs for the name Dark Angel (see? The apostrophe occurs almost automatically `:(`).

I'm working on a nice text system right now. If anyone has any requests I'll see what I can do.

June 27, 2004 (v0.99.3)

Ahhh, home at last, where I have a proper computer and Kink-FM blasting through my stereo, excccellent! `=)`

- ⊞ Added `-Map` and `-Wl` command-line options to the flags list.
- ⊕ Moved the graphics data that is only used once into the demo-folder where they are used; the gfx directory now only has shared graphics in it.
- ⊕ Had to write `swi.s` again because the `utils clean' command destroys all .s files if there is a .c with the same title. Make sure that there is no `swi.c` in your `utils` directory when copying the the new stuff.
- ⊞ The mode7ex.htm page is finally complete. Yes I know it's long and full of nasty linear algebra; if you have trouble getting through it and/or have suggestions on making it more readable, plz, do tell.
- ⊞ The accompanying `mode7d` demo is just about where I wanted it.Sure there are still some minor problems, but it should be enough to get you started.

June 21, 2004 (v0.99.2)

- ⊞ I made a lot of changes to `mode7d`; all the real mode 7 code is now in separate files so using it in other projects is easier now. Though mode7ex.htm still needs a lot of work, you can find most of the text in draft-form in m7theory.zip. Yes, it's a Word document; yes, I know that sucks; yes, I will convert it to html when I the text is stable and understandable (please tell me what I need to change in this

respect); and yes, I will do this conversion manually, since Word should be allowed to approach HTML to within 500 yards. Perhaps more.

- ⚠ Made some minor fixes to the matrix page. Silly me, I got the cross-product definition all wrong.
- ⊞ Added info on `REG_P1CNT` to the keypad page. Yet another thing which only this site covers `:)`.

Devving on a P2-300 with 24MB RAM: VBA runs at 50% (and 23% for mode7d) and minimizing a window takes a few seconds. Man, this sucks.

June 11, 2004 (v0.99.1)

- ⚠ Fixed the style sheets so that the background image, colors and borders, etc, etc, appear as I had intented on Mozilla. Sorry about that, didn't know the the wrong comments would screw it up so much. Made a vallidation run and got rid of all nonvalidities … except one: the <nobr> tags that I need to keep certain things together.
- ☻ All BIOS calls are now inside `swi.s`, in assembly. Which is where they belongs, really.

June 3, 2004 (v0.99)

- ⚠ Found out about the wrapping artifact, and changed the sprite pages accordingly. `obj_aff` now allows moving the sprite so you can see this artifact for yourself.
- ☻ Finally got over my dislike of near-empty directories (the files get so lonely that way) and put all the demo-code in separate directories. Now, if only I could get over my ifphobia as well…
- Added a section on vsyncing with interrupts in swi.htm and an accompanying demo, `swi_vsync`. You need to see these.
- Added `int_enable_ex` and `int_disable_ex`, which should make working with interrupts easier. However, I am not 100% sure if I got all the registers and flags right.

- ⚠ C++ doesn't like it when you try to use a struct-copy on a volatile variable, like `bga_update_ex` does. Or did, I should say.
- Learned some new CSS tricks and am busy updating and structuring the layout of *all* pages. It's mostly subtle stuff though, like standardizing the equation layout and giving code and register listings a subtle border that makes it stand out in printing. Non-subtle is that ever image should have a caption now.
- ⊞ Resumed work on mode7ex.htm and its accompanying demo, `mode7d`. Adding variable pitch turned out to be easier than I thought, w00t! It's still a little buggy, though.

With all these changes, it is adviced to save or remove older Tonc stuff when upgrading to avoid double files and other inconsistencies.

May 24, 2004 (v0.98.5)

- ⊞ devkitARM is now the primary devkit for Tonc. Makefiles and text are updated to match the change.
- ⊞ Using a separate interrupt file now instead of a custom crt0.S and got The Point® of the critter in the process. The text is modified to reflect newer insights, as is `int_demo.c`
- Started work on a glossary.
- Added instructions on how to run makefiles without Visual C++ in the the Tonc-code readme. Silly that I never thought of that before.
- ⚠ `REG_IF` is at `0400:0202`, not `0400:0200`, doh!
- As you can see, I'm trying to use context-specific bullets for log entries. I still need to figure out what images to use for what purpose, though.
- Rewritten the `build_all` and `clean_all` targets in `tonc.mak`.They're quite nasty now, but act more correctly and allow me to switch to a "one demo, one dir" structure if people finds having everything in the `examples` folder a bit messy.

I am soooo tired right now so I wouldn't be surprised if I messed up somewhere. I'll get it fixed when I've had a chance to sssslssszzzzzzz….

May 16, 2004 (v0.98)

- ⚠ Lot's of changes. First of all, I finally have a means to test on hardware, Wheeeee!! However, it did point out that you can't use the object tileblocks for backgrounds after all `:(` . Added my early experiences with hardware tests on a number of the pages.
- ⚠ Including in the windowing section. It seems that you need to be really careful with the vertical settings of windowing. Updated the windowing demo to not use u16 arithmetic for the window size (which is given in bytes), and more precise movement.
- ⊕ Threw the DMA code around. I'm now using `dma_memcpy()` for general copies, and renamed the old `DMA_CPY()` macro to `DMA_TRANSFER` , and rearranged the order of arguments to match `memcpy` . Makes more sense that way.
- ⊕ Also changed `oi_set` to `oi_set_attr` and `oi_pos` to `oi_set_pos` .
- The `Tonc Utils` configuration now compiles the utility code into a library. Required for `mode7d` .
- Created a `build_all` rule in `tonc.mak` . Rebuilding everything by hand was really getting on my nerves.
- More random clean-ups. The images on the entrance page are now links as well. Important notes are now in red boxes, for extra visibility. Added the demo-code of `bm_mode.c` , to show the basic steps of loading a picture and using keys. I should still post full code that earlier in the tutorials. Added an example of a fixed point identity matrix on affine.htm to make sure people don't try to use floats.
- ⚠ Fixed swi.htm. *Again!* I swear, if I find one more error here some somebody is gonna get hurt. And it won't be me. This time the range of arctan2 was wrong (should be full circle). What makes this error even worse is that I should (and did) know it's supposed to be the full-circle all along; it is the raison d'être of arctan2 after all.
- Matrix page is done.
- Practically fell out of my chair laughing at villainsupply.com. Ouch.
- Nearly drowned in my own drool after watching the Nintendo stuff at E3. Gargle.

- The picture used in `key_demo` had the palette-indices of `KEY_START` and `KEY_SELECT` switched. I never really noticed because emulators don't have real start and select buttons. So once again, hardware testing saves the day.

Apr 29, 2004 (v0.97)

- Added links to devkitARM as well as instructions on how to get it working. There is a very real possibility that I'll switch to this toolchain in the future.
- Converted most of the macros to inline functions. Safer, easier to read and just as fast. Yes, plz.
- Some more diagrams about tile-counting and the affine transformations.
- Still to do: finishing the matrix page (and perhaps the mode7ex page). And now that I have my tile-map editor Mirach, I may be able to do something with that as well. And I really, really need to get working on a text-system.

Apr 9, 2004 (v0.97)

The object affine functions have a background counterpart now, and `mode7d` is coming along nicely.

Apr 4, 2004 (v0.96)

Renamed OAM structs and related items. *Again*. When is this gonna be final?!? Also, thanks to Lupin's problems with sprite-placement in 3D I finally got my matrix-transform sense back. Now that I get it again, I hope to expand the mode7 section in the near future. I already got a working example for 3d-sprite placement already in the form of `mode7d`.

Mar 31, 2004 (v0.96)

Working on a vector/matrix page, some reshuffling of page-order and yet more random little cleanups.

Mar 24, 2004 (v0.96)

Replaced the assembly listing in swi.htm with the proper Thumb listing. I'd forgotten I wasn't using ARM code anymore.

Mar 20, 2004 (v0.96)

And just when you think you're finished you find another 2 things you can improve upon. Argh. Anyway, I've changed the way sine and cosine are retrieved. They're both macros now, using one 512-entry long `s16` sine-LUT. Also, I finally realized how to XOR the `vid_page` directly for page-flipping. And, oh yeah, the compiler flag for compile, but not assemble should be `-S`, not `-s`. Oops.
I think I finally know how I can modify my affine functions to apply to backgrounds without having to use the `OBJ_AFFINE` structure, but it may be a while before I actually do that.

Mar 17, 2004 (v0.95)

Added ArcTan2 function to `swi_demo` and fixed the errors that `swi.htm` still contained. Argh, and I thought I'd been thorough in weeding out all inconsistencies after the recent name and code modifications.

Mar 14, 2004 (v0.95)

First entry in the log. I've rewritten the parts about sprites and backgrounds, changed glyphs to tiles and tiles to tegels (hope I got them all `:]`), updated all the code one last time and written sections on how to set up DKA, MSVC and makefiles. I think Tonc's ready for use now, wheee!