cortex-m3

Alessandro Rubini (rubini@gnudd.com)

Table of Contents

Introduction

The Cortex-M3 is an interesting microcontroller that can be bought mounted in very-low cost evaluation boards. Sometimes the examples provided by the manufacturer are very low quality or just unusable for free people, as they depend on a bizarre proprietary compiler or a Windows environment. This software package and documentation shows how to work with the M3 with only free tools, in a GNU/Linux environment.

I personally enjoy building my stuff starting from the basics, especially with microcontrollers, so this document explains how I've proceeded with them. You'll most likely find a zillion similar documents, and likely some are better than this one. I just make it available in the hope that it is useful for people like me. On the other hand, if you are really like me, you won't read this as you'd rather roll your own than look around.

1 Getting Started

This chapter is a *getting started* description for this package.

1.1 Prerequisites

This document assumes you know have some solid experience with C language and the typical Unix and GNU/Linux build environments. For some code fragments, ARM-V7 assembly is used, but not explained; similarly, linker scripts are used but not explained.

You are also expected to have some experience with cross compilation. Here, the usual `CROSS_COMPILE` prefix is inherited from the environment. You need a recent enough version of *gcc*; *armv7* and *Thumb-2* were introduced, in *gcc-4.3*, so version 4.2 and earlier will not work for this processor.

Being able to use *git* is a plus, but if you ignore it and you are looking for an interesting tool to study, please study *git*. This package is available for *git* download from `git://gnudd.com/cortex-m3.git`

1.2 Information and code layout

In this document, when I say *this package* I refer to, well, this package. For me it is the tar file you downloaded, including both the code and the documentation.

The material in here is built from scratch and all steps are shown, *git* is used for revision management and all releases are shown. If you don't have *git*, just untar the package and live with the latest version; if you have *git*, you can checkout each version.

Note, however, that I'm not rewriting history to keep it clean, so the document increases over time, sometimes before the code it describes and sometimes after it. If you checkout older versions, please keep the documentation from the *master* branch in a safe place.

I make a *tag* with the release date every time I publish the package, and I won't change the history after that. So you may branch and merge and rebase and whatever you want with published versions.

The document, hoverer, increases over time and I don't care about keeping a clean history, I'm more concerned in having this out soon for my mates whom I gave boards to play with.

1.3 Hardware

The Cortex-M3 exists in several flavors, what I personally bought is the LPC-1343 by NXP in a 13-EUR-worth board built by Olimex. I'd love to not name my suppliers and avoid advertising, but some of the information is specific to their products, so you'd better know that.

I won't make the extra effort of separating the general information from the specific information, as this package aims to be as light as possible (at least as far as my effort is concerned).

2 Quick overview of the device.

The device being used here has 8kB or RAM memory and 32kB of flash storage. According to how you jumper it at boot, it either executes the program it has in flash or it allows to be programmed.

It can be programmer either by USB (default) or UART. Here I am starting with UART programming.

In order to force programming you need to pull P0-1 low (there is a jumper on the board I bought). If that signal is low, P0-3 selects whether UART or USB is used.

To do that I added a second jumper, in addition to the level translator needed to access the UART port. The pins for serial communication are P1-6 and P1-7.

3 How to program the code

The *tools* directory in this package includes two programs: *program* is programming a binary file to RAM, while *progrom* programs it to ROM. It's stuff I've been using on ARM7, where the UART was the only way to program the device. The tools work for me, but they are not complete nor very configurable. For example, the load address (which is the run address for RAM-based programs) is fixed to 0x10000400 – 1kB within RAM, like it was fixed for a different address when I used them on ARM7. I'll make them autoselect the address, but it's not there, yet.

Grep for *getenv* to see what are the configurable parameters (the serial port being used, for example).

If you prefer to use other tools you find on the net, that fine. This is what I use and make available, but I won't be surprised if something better exists.

Initially, we are programming to RAM (so the program must be smaller than 7kB, as 1kB is used for stack space). This an example run of the *program* tool, whic correspond to the first binary built (the one described in the next chapter):

```
favonio% ./tools/program m3c.ram.bin
Opening serial port /dev/ttyUSB0
Forcing boot loader mode
Syncronizing... done
Identifying... done
part number: 3d00002b
LPC1343, 32kB Flash, 8kB RAM
size is 900
W 268436480 900
0
...................OK
stuff
stuff
[...]
```

4 An initial program working from RAM

This chapter introduces my first step on the device: a program that prints stuff the serial port and moves the 4 port-3 GPIO bits. All code parts are quickly described.

4.1 The Makefile

The Makefile builds a program that runs in RAM and one that runs from ROM (flash), even though by now we are only using RAM. The program is called *m3c*, which doesn't mean anything. The source files is uses are *vectors.S*, *boot.S*, *io.c* and *main.c*.

At this point *vectors.S* is unused, so it is empty. Vectors are not used when you execute from RAM and use no interrupts. *boot.S* is described in the next section, *io.c* is the serial configuration and output (but we are not configuring the serial port by now, as it has already been configured by the internal ROM to run the programming protocol).

4.2 Boot.S

When the device start (after programming), you need to build a C language environment for your C program to run. This means directing the stack pointer to a sane place and clearing the BSS. To do this you need some assembly code, and what you find here is in *boot.S*.

The file defines three ELF sections: one which is used when booting from RAM, one that is used when booting from ROM and one that is used in every case. The linker script selects what sections are used and what sections are discarded at link time, so having all of them in the same source file is not a problem – I find it clearer, as you can look at the whole of your boot code in a single place.

4.3 The Linker Script

We have two linker scripts: *ram.lds* (used in this chapter) and *rom.lds* (not used by now). They are used to describe how the final executable is built: the address where code and data must be placed and (in *rom.lds* alone) the address where data must be stored.

The *Makefile* creates the ELF file *m3c.ram* using the *ram.lds* file. The ELF file is what you use to disassemble your code and and pass information to *gdb* when debugging.

finally, *objcopy* creates the *m3c.ram.bin* binary file, which is what you transfer to the physical device.

4.4 Serial Output

The file *io.c* defines *putc* and *puts*. It is the only output we need at this point. It also declares an empty *serial_setup* function because *boot.S* calls it. Setting up the port will be needed when booting from flash memory; at this point we'll piggy-back on the UART configuration that has been used for programming.

To write a string we repeatedly call *putc*; to write a character we write to the transmit register when the transmitter is not busy with the previous character.

4.5 The main Function

The *main* function, in *main.c* is a simple infinite loop that prints a message on the screen, flips a led and waits a while. The CPU registers being used have been extracted from the "user manual" for the processor.

5 Writing an useful program to Flash

Writing to flash is needed in order to have the application run at system boot, without interaction. There are two ways to write code to Flash memory: from the UART or from the USB slave device.

In this chapter we are going to write flash memory, and replace the led-flashing application with a minimal memory monitor.

5.1 Writing Flash from the UART

If you have an UART on the device and you already tried the *tools/program* thing introduced in the previous package, you may want to run *tools/progrom*. It works in the same way, but it programs to flash memory (I call it *rom* for symmetry with *ram*, and you'll forgive me for the associated inconsistencies). The program at this point is still suboptimal, so you may just switch to the next section.

This is a sample run of *progrom*:

```
favonio% ./tools/progrom m3c.rom.bin
Opening serial port /dev/ttyUSB0
Forcing boot loader mode
Syncronizing... done
Identifying... done
part number: 3d00002b
LPC1343, 32kB Flash, 8kB RAM
size is 2300 (xfer 2700)
W 268436480 2700
0
...................OK
...................OK
...................OK
position 0x00000:
   prepare: 0
   erase: 0
   prepare: 0
   copy: 0
Sent memmap code: OK
mimmo.c: Ready to get input
```

As you see, the program finally jumped to the reset vector, so the program you just sent to flash is already executing. This is shown but the `mimmo.c: ready` line.

The tool, just like *tools/program*, continues reading and writing the serial port and stdin/stdout until killed, but please note that it is not perfect at all. Buffering and timeouts are interfering with operation, so you may prefer to fire *minicom* or something similar instead.

The tools has other drawbacks, for example it is limited in the size of binary code it can program, and it shows it's ARM7 origins in several places, but this version is not working with ARM7 any more. I want to make it more portable and fix stuff over time, but I don't know when it will happen.

5.2 Writing Flash from USB

Another option for flash programming is using the USB slave. to program through USB you need to force programming (or "boot loader") mode by using the jumper on P0-1, also known as BLD_E, but without plugging the jumper on P0-3 – or not having it prepared at all. In this case, the Cortex registers as a storage device with the host computer. You'll find a 32kB-worth file called `firmware.bin` in the device.

Official documentation says to just remove the *firmware.bin* and write your own stuff on the storage device. But it doesn't work if you run Linux. Moreover, the official documents say you

can write any file name to the Cortex, but in practice it must be a 8.3 old-fashioned DoS name, or it won't work.

Why mounting and copying doesn't work. When you copy a file to external storage in Windows, the operating system allocates blocks sequentially and writes them sequentially. This is why they have fragmentation problem and the storage performance sucks. In Linux we have smarted algorithms and the LPC ROM is not ready to accept writes that are not ordered. Similarly, named not matching the old 8.3 limit (like names with two dots) are laid out differently on the FAT partition, and the ROM can't understand that. I'll save you my rants about why USB storage is completely wrong in design...

Even though sometimes mounting and copying does actually work, it is not reliable, so I advise against even trying that. What you should do, instead of mounting, is running the user-space suite *mtools*. To this aim, I added this line in my `/etc/mtools.conf`:

```
drive c: file="/dev/disk/by-id/usb-NXP_LPC134X_IFLASH_ISP000000000-0:0" exclusive
```

And thus I can run the following commands:

```
mdel c:firmware.bin
mcopy m3c.rom.bin c:new.bin
```

At the next reboot, after removing the jumper on P0-1 (BLD_E), you'll have your application running.

I am extremely grateful to Peter Stuge who first hinted to use *mtools*, thus saving me hours of swearing and sweating.

5.3 Filling vectors.S

In order for a program to run from flash you need to fill the reset vectors. In the Cortex-M3 the first word of memory is the initial stack pointer and the second word of memory is the pointer to the first instructions to be executed plus 1.

After these two words there are the interrupt vectors, but we are not using interrupts at this point so we can save the space. However, the LPC internal ROM calculates the checksum of the first 8 words in order to validate the user program – if it not valid it will go in ISP mode, irrespective of the P0-1 jumper.

Our new *vector.S*, thus, reserves space for the other vectors, and the *tools/fix-checksum* tools is automatically run by our *Makefile* in order to prepare a correct `.rom.bin` file.

The linker script already uses the `.vectors` ELF section, so nothing more is needed.

5.4 Initial Device Configuration

When booting from flash, unlike what happens when you interact with the ROM using the UART, nothing in the device gets configured. To use the serial port, of the GPIO or whatever else you need to configure it first.

The `boot.S` used in the previous chapter calls *serial_setup* before calling *main*, so we only need to fill this function, without touching the assembly code. The function, in *io.c*, must set up the divisors for 115200 baud (we assume we run at 12MHz, no PLL is used at this point) and other parameters; it also turns on a few internal peripherals of the device.

5.5 Serial Input

To make a useful application (for some meaning of *useful* we also need serial input. Thus, *getc* and *gets* are added to *io.c*. The code is pretty trivial and doesn't need any explanation.

5.6 mimmo.c

The application being run is called "MIni Memory MOnitor", or *mimmo* for short. The file *mimmo.c* runs the application, which can read or write any 32-bit address. For example, we can read the initial vectors and turn the leds on:

```
r 0
read 00000000
100003fc
r 4
read 00000004
00000021
w 50038000 f
write 50038000 = 0000000f
w 5003003c 0
write 5003003c = 00000000
```

In the excerpt above, `r` and `w` are the commands, while the other lines are messages from mimmo.

If you don't have a serial port, you can change "`if (1)`" to `if (0)` in *main.c* and run the usual led-flipping application instead of mimmo.

5.7 The Current Code Base

If you checked out the *git* tree, you'll find the following commits since the previous chapter (oldest commit on top):

```
c539db6 tools: added fix-checksum
2a71781 Makefile: fix checksum when making rom.bin
14a828c tools/progrom (and lib): use lpc13 values, not lp21
98f2d50 vectors.S: add the reset vectors, so it can go to flash
258a00d io.c: added serial_setup
3cc1269 io: added gets and puts, move prototypes to an header
89fcb91 mimmo.c: mini memory monitor, new file
e457a82 main, Makefile: really use mimmo
```

Whether you got the *git* tree or the *tar* file, this is the list of files that make up this example, excluding the tools:

```
-rw-rw-r-- 1 rubini staff 1134 Feb 22 10:13 Makefile
-rw-rw-r-- 1 rubini staff 1169 Feb 20 11:07 boot.S
-rw-rw-r-- 1 rubini staff 2443 Feb 22 10:13 io.c
-rw-rw-r-- 1 rubini staff  591 Feb 22 10:13 main.c
-rw-rw-r-- 1 rubini staff 1627 Feb 22 10:13 mimmo.c
-rw-rw-r-- 1 rubini staff  406 Feb 22 10:13 vectors.S
```

And this is the size of the compiled ELF files:

```
text    data    bss     dec     hex filename
2240       0     16    2256     8d0 m3c.ram
2300       0     16    2316     90c m3c.rom
```

6 Using GPIO pins

The GPIO pins in this device are especially strange. Some abstraction is definitely needed, and I personally prefer something that is as portable as possible.

To this aim, I want a `gpio.h` header file and a `gpio.c` source file. The functions are as simple as possible, but currently they are not designed to be inlined. The *gpio* argument is checked for

correctness only in the configuration function, which is expected to be executed at least once before using the bit, and possibly only once in the life of your program.

6.1 Design Choices for GPIO

I personally made a few choices that you may agree or disagree with.

As a prerequisite, the code introduces the standard functions *readl* and *writel*, so I can reuse this *gpio* material in other projects. Similarly, A few types are defined in `types.h`, the most important being `u32`.

GPIO Numbering

Most other processors I work with have their GPIO pins divided in "ports", each port being 32 bits wide. Both ports and bits are counted starting from 0 – sometimes ports start from A and use alphabetic letters.

In the operating systems I use and enjoy, gpio numbers are just numbers, whether or not the hardware docs talk about ports or not. So bit 10 of port 2 is GPIO 74 (2 * 32 + 10), irrespective of how hardware is. Here I use the same approach; it eases development of generic device drivers, like a bit-bang I2C driver, based on two GPIO pins.

As a practical result, we can use GPIO 0 through 11, 32 through 43 and so on. Macros to convert to and from port-and-bit are provided nonetheless, as they are hardware-independent conversions.

Alternate Functions

Most processors have alternate functions, and GPIO is usually function 0. Here we have up to 7 "alternate functions" and one GPIO function. Then we have other features (like hysteresis) that I'm not supporting by now.

The LPC13 device is set up strangely, in that the PIO function is not always function 0 – sometimes it is at function 1. But, for the sake of portability, we need function 0 to always represent PIO.

For example, a generic LED or key driver (or bit-bang I2C or whatever) needs to configure its own bits as PIO, irrespective of what the host hardware is (most likely, the bit numbers come from a data structure, so the driver ignores how to configure them). For this reason, `AF0` is always the PIO function. I define bit masks in order for the GPIO code to swap AF0 with AF1 for those bits where this is needed.

Accessing Configuration Registers

Even though the GPIO configuration registers are all alike, their placement in the memory map is absolutely random. Here the need is describing this placement in the smallest possible space.

Please note that configuration registers are used very rarely, because the PIO operations (including switching the direction of one bit) are performed on different registers.

To keep the code as small as possible, I define an array of offsets, one per GPIO bit, stating where the relevant register lives in the associated memory area. Such offsets are 8 bits long, and are the index of the register, so they are shifted by two bits before being added to the base register. This is saves storage space in exchange for some calculation, but as said pins are configured usually once for the whole uptime of the system.

In my experience, people looking at documentation are used to look for relevant symbolic names in the headers, grepping for the hex address. For this reason, my header prefers compile-time calculation to build the offsets starting from the complete hex number. Moreover, this avoid users from checking and rechecking the table when looking for other bugs.

Changing several bits at the same time

Although the LPC13 allows to atomically change an arbitrary set of bits as long as they are
part of the same port, I offer no support for this at API level – I plan to add it later, but it's
not there yet.

6.2 The GPIO API

This is the programming interface I'm using for GPIO. Both `gpio.h` and `gpio.c` are part of
this release, and *mimmo* has been modified to access GPIO in input and output.

*Please note that release 2011-03-20 of this package had a bug and bits 8-11 of each port were
not working. Thanks to Marcello Torchio for finding the problem.*

```
GPIO_NR(port, bit);
GPIO_PORT(nr);
GPIO_BIT(nr);
```
These macros convert from port+bit to number and back. They are not expected
to be used often, but they may be useful.

```
void gpio_init(void);
```
Initialization is needed, in that the GPIO and pin-connect clocks of the chip must
be powered for the following functions to work. Even though here initialization of
the UART is already doing the required setup, it's in general good practice to have
an init function and call it before using the module.

```
int gpio_dir_af(int gpio, int output, int value, int afnum);
int gpio_dir(int gpio, int output, int value);
```
The former function sets the alternate function for a bit, and it configures it as
input or output. The latter function only changes the direction (this is useful since
changing the mode is much more costly in term of machine instructions. The *value*
argument is needed to change the GPIO output bit right after changing the mode
(as setting value beforehand won't work with this device.

```
int gpio_get(int gpio);
u32 __gpio_get(int gpio);
```
The functions return the current input value. The former return 0 or 1, while the
latter returns 0 or non-0. As usual in several context, the double underscore means
the function is "internal" or "lower level".

```
void gpio_set(int gpio, int value);
void __gpio_set(int gpio, u32 value);
```
The functions set an output bit. The former receives 0 or 1, while the latter receives
0 or the bit value – i.e., the same value that is returned by *__gpio_get*, to save a few
instructions in some common situations,

6.3 Adding GPIO to mimmo.c

In order to test the GPIO functions, the next step is adding a few gpio-related commands to
mimmo, our main test application. However, *mimmo* has no real user interaction (we have no
printk or *printf*, yet). Thus, I only added two commands: *p* for PIO and *a* for AF.

The following table shows the complete list of *mimmo* commands. Please note that all arguments
are hex, while all documentation about GPIO pins uses integer numbers. So for example in
mimmo P0_11 is GPIO11 which must be passed as `b`, and P3_0 is GPIO96 which must be
passed as `60` (each port is 32 bits, i.e. 0x20).

```
g <address>
```
Go to an address. The call is not expected to return.

`r <address>`
> Reads a memory address. Any address is allowed, whether it lives in Flash, RAM, or register space. An address in a reserved area will freeze the CPU, as we have no fault handler at this point.

`w <address> <value>`
> Write a memory address. Again, wrong addresses will freeze the CPU.

`p <pionr> [<value>]`
> The command reads or writes a GPIO pin (also called *parallel I/O. value*, if present, must be 0 or 1 and forces the pin as an output. If no *value* is passed, the program configures the pin as input and returns the current value.

`a <pionr> <afnum>`
> Sets an alternate function value for the specific GPIO pin.

6.4 The Current Code Base

If you checked out the *git* tree, you'll find the following commits since the previous chapter (oldest commit on top):

```
2a67019 doc: small fixes
cdda737 header files: add some standard stuff used in gpio.c
a1ac985 gpio: new source file and header
c4a75ad Makefile: added gpio.o
a5e92d8 mimmo.c: add 'p' and 'a' commands
7d5f2a2 gitignore additions
1776a96 docs: added gpio chapter
2b43905 gpio.h: fix __GPIO_DAT port address
```

The code base is not very clean, as the *writel* and *readl* primitives are not used in all I/O operations – the GPIO files use them, but the previous code still uses direct volatile access, and some cleanup is needed in other places. However, such cleaning is not planned, at this point.

The only thing I miss before I consider this project complete is *printf*. And that is the topic of the next chapter.

7 Adding Printf

Code-wise, the worst thing in the *mimmo* thing is the output messages. They are build by successive calls to *putc* and *puts*, intermixed with calls to *inthex*, the local function to create hex strings. This is fixed by relying on a true *printf* implementation.

7.1 pp_printf

Unfortunately, most such implementation are bulky; some other implementations, like the widely used *mprintf* function, are small but incomplete. My personal choice has always been the code base from an old Linux kernel version – recent kernels include extra non-standard features, like looking up the symbol table for pointers; that is very useful stuff, but too much Linux-specific.

When I initially picked the *printf* implementation from *U-Boot* (which in turn picked it from the kernel many years ago), I found it very clean and understandable, but still too long for some applications. Therefore, I chose to keep the same parser, but remove some of the actual formatting. In this way, you can code your application with full-featured format strings, and choose to build against the full *printf* or the reduced one; your values will be printed nonetheless, but they may be missing alignment or other presentation details if you build the small code base.

After using the thing for a while, I had to leverage it to a real project, and thus I first made it a standalone package, called *pp-printf*, for *Poor Programmer's Printf*, published at `git://gitorious.org/rubi/pp-printf.git`.

The code in the *pp_printf* subdirectory of this package is the current master of the other project, copied here as a single commit. The linker scripts of this package now provide the *printf* symbol at link time, mapping it to *pp_printf*; this means the code can use the conventional name. Finally, the *mimmo* application is simplified by using the new function.

7.2 The Current Code Base

If you checked out the *git* tree, you'll find the following commits since the previous chapter (oldest commit on top):

```
6378e69 tools: update and bugfix
b8de310 pp_printf: upstream commit c94f55f3
127bbfd Makefile: build pp_printf (xint version)
035ea8b r*m.lds: provide printf as alias to pp_printf
ff920e4 io.h: prototype printf; fix puts
180ff03 mimmo.c: use printf to get simpler
```

7.3 Coding Further

To code or not to code, this is the question.

Initially I was planning to proceed with this package, with some clean-up, possibly USB support. This was mainly meant to help my students, to give them a working base for their coding projects.

Three years passed, and I found myself working on other stuff, which is actually more fun for me and a better starting point for my students: the other project is modular and can make good use of their commits, when their work is good enough (the code is GPL in any case, and I've nothing against merging GPL work by other authors.

So I consider this project over. If you want something more, I think you can check what *libopencm3* offers, or one ot the several operating systems out there.

My personal work turned out to be an operating system I can write from scratch in two hours: `git://gnudd.com/thos.git`, and on my own *gnudd.com* as well. You can also find a video of me writing it live at a conference, where I made a mistake with the wristwatch and completed it in 75 minutes (quite distressing minutes, I admit). The more-real spin-off of this "two hour OS" is "born again THOS", *bathos* (which means depth, in Greek language); that one is ported to a few different CPU cores and is going to have task preemption pretty soon. You can find it at `git://gnudd.com/bathos.git`. If you can't access the *git* port, `http://` works as well, and a copy is available from *gitorious.org* too.