Blinking an LED like a pro

Deeper look into open source embedded toolchain

Anton Gerasimov

Embedded software engineer

Advanced Telematic SYSTEMS

app.atsgarage,com

ENDOCODE

Where am I?

Today we’re going to learn how to build deeply embedded software from scratch, without following the workflow recommended by specific vendor.

1. What constitutes an embedded toolchain.
2. What is special about programming for microcontrollers (MCUs).
3. What you will still need to get from MCU vendor.

You’ll need to have some experience with C and Linux (Unix, POSIX) command line though.

Deeply embedded?

In the good old days (ca 10 years ago) it was the only “embedded” thing we had.

- No hardware abstraction layer **=>** we need to know our hardware
- No virtual memory **=>** we need to know what is located where
- No bootloader **=>** care needs to be taken of how variables are initialized, stack and/or heap is allocated, clock and other devices are set up
- Diskless **=>** dedicated tools for delivering software to the device are needed
- Real time requirements **=>** simple scheduling schemes, full control of execution

Motivation: deeper understanding

You've learned how to program an Arduino and want to go further

OR

Recommended procedure for your board just didn't work

OR

You want to do something trickier than standard workflow implies

Motivation: vendor independence

Most MCU vendors will provide you with some getting started documentation with step-by-step instructions on how to use their board. However, there are always good reasons to go further that that.

- Sometimes you want to reuse your code from another MCU/board.
- Sometimes your vendor’s IDE doesn’t support your operating system.
- Sometimes you just think that five different toolchains is already enough.

Let’s get started

We will need

- Toolchain: compiler, linker, binutils, standard library
- Documentation on board, MCU, processor core
- Headers, startup and examples

Let’s get started

We will need

- Toolchain: compiler, linker, binutils, standard library
- Documentation on board, MCU, processor core
- Headers, startup and examples

Compiler

Compiler takes a single source file (*.c), produces a single object file (*.o).

```c
int exp(int b,unsigned e)
{
   int res = 1;
   int i;

   for(i = 0; i < e; i++)
      res *= b;

   return res;
}
```

```asm
     push {r7}
     sub  sp, sp, #20
     add  r7, sp, #0
     str  r0, [r7, #4]
     str  r1, [r7]
     movs r3, #1
     str  r3, [r7, #12]
     movs r3, #0
     str  r3, [r7, #8]
     b    .L2
.L3:
     ldr  r3, [r7, #12]
     ldr  r2, [r7, #4]
     mul  r3, r2, r3
…
```

```
7f454c46010101000000000000
00000000100280001000000000
0000000000000f8040000000000
00053400000000000028001300
100080b485b000af786039600
123fb600023bb6007e0fb687a
6802fb03f3fb60bb680133bb6
0ba683b689a42f3d3fb681846
1437bd4680bc70477c0000000
400000000000401
0d0000000c430000004d00000
000000003800000000000000
026e0000000102710000000000
000003800000019c71000000
036200010271000000002
916c03650001027800000002
```

Example#1

Linker

- Object files are only half-baked executables.
- Object file contains *symbol table* with variable and function names defined in other object files and names accessible from other object files.
- All data and code in object file resides in *sections*, which combine data of the same type.
- Linker’s job is to match each external reference to its definition in another object file and produce executable by properly placing sections in memory.

More details: http://www.skyfree.org/linux/references/ELF_Format.pdf

Linker script

- Linker script defines placement of sections in the memory.
- LS granularity is a section of an object file, you can’t address individual symbols (variables and functions) from there.
- Most common sections are *.text* for executable code, *.bss* for zero-initialized variables and *.data* for variables that should be initialized to nonzero values.
- There are also supplementary sections like *.eh_frame* or *.ARM.exidx* for stack unwinding, *.ctors* and *.dtors* for static objects’ constructors and destructors etc.
- You can define your own custom sections.

Example#2

Binutils

Sometimes you'll encounter this term, so you'd better know what it means.

- The most important utility in binutils is **ld**, the linker.
- GNU assembler (translates assembly to an object file), **as**, is also a part of binutils.
- **objdump** converts ELF executable to loadable binary format.
- **ar**, **strip**, **strings**, **nm**, **size**...

More details: https://www.gnu.org/software/binutils/

Standard library

Implements functions of C standard library: strlen, memcpy, printf...

There are multiple different implementations:

1. glibc - most popular on Linux.
2. newlibc and uClibc - competing implementations for MCUs.
3. A lot more: musl, bionic, dietlibc.

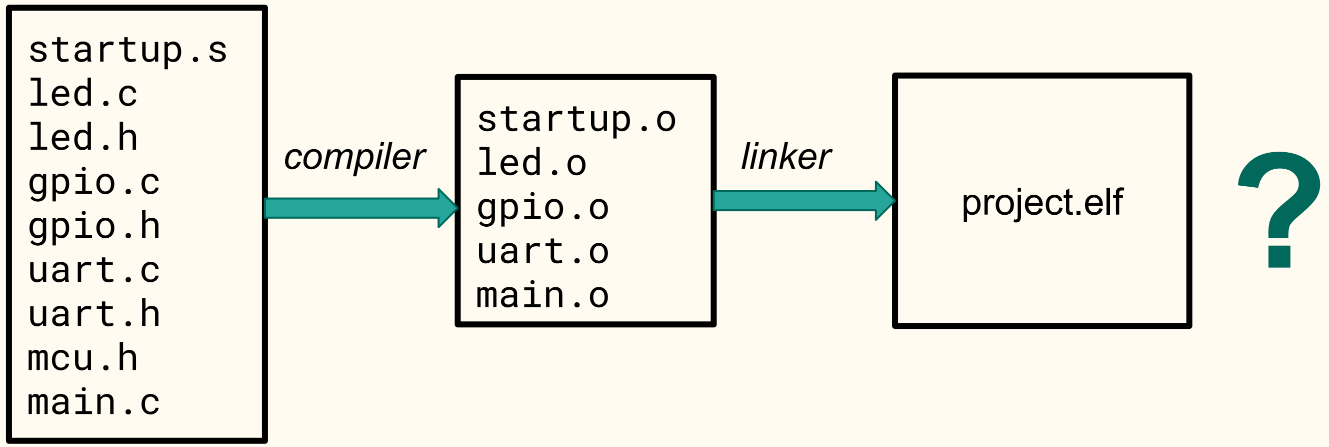
More details: http://www.linux.org/threads/a-variety-of-c-standard-libraries.7876/

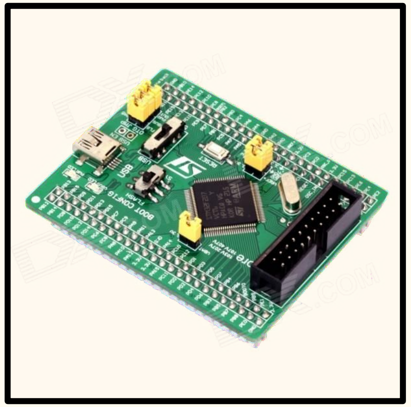
Available toolchains

- Most popular one is GNU toolchain.
- It is open-source, so you can build it from source.
- There are pre-built toolchains available, i.e. Linaro toolchain for ARM.
- Those who like doing it in alternative way may try clang.
- There are of course commercial compiler vendors like Keil and IAR.

Linaro toolchain: https://www.linaro.org/downloads/
Clang: https://clang.llvm.org/

Firmware loader/debugger (1)





Firmware loader/debugger (2)

Two ways to flash the MCU

- Via serial interface
  - Hardware required: COM port (who remembers what it is?) or USB ↔ Serial (FTDI or competitors). Should be supported on your board of course.
  - Software required: vendor-specific bootloader. bootstm32, lpc21isp, Flash Magic etc.
- Via JTAG/SWD
  - Hardware required: JTAG debugger (emulator): J-Link, ST-Link, ARM-USB-OCD etc. Some boards have on-board USB-JTAG adapter.
  - Software required: openocd (the most universal tool), jlink, stlink.
  - Bonus: with JTAG you can not only upload the software, you can also debug your program directly on the MCU with **gdb**. Using gdb is outside the scope of this meetup.

More details: http://openocd.org/; https://www.sourceware.org/gdb/

Let’s get started

We will need

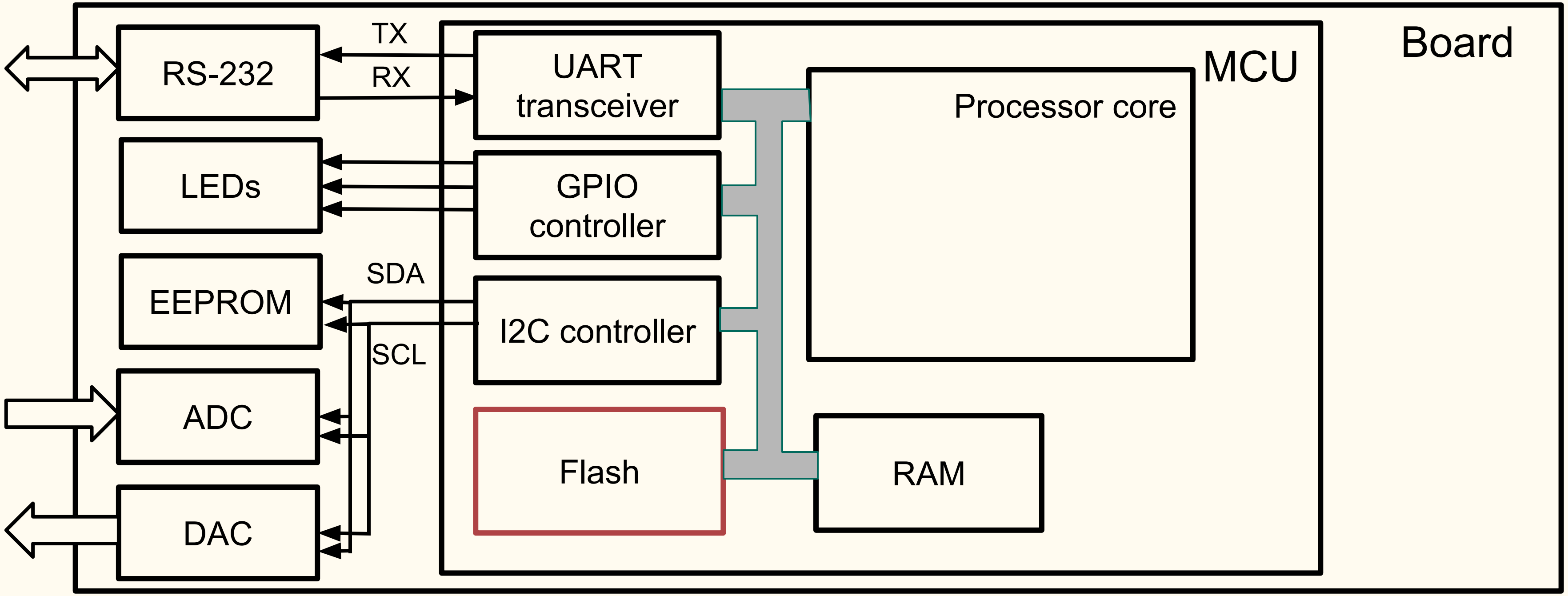
- Toolchain: compiler, linker, binutils, standard library
- Documentation on board, MCU, processor core
- Headers, startup and examples

Docs

What is (hopefully) documented.

- Processor architecture
- MCU
- Board

Intermezzo: structure of an embedded system



Docs: processor

- Most of details of processor architecture are abstracted out by C compiler.
- Basic info about interrupt processing can be found in documentation on processor. Most interrupts are generated by peripherals, and details of how they are generated, acknowledged etc. should be looked for in MCU docs.
- You will definitely need to look into processor manual if you want to write assembly code.
- Surprisingly, ARM Cortex-M processors have system timer (SysTick) integrated into processor core.
- Unsurprisingly, processor manual(s) can be found on processor designer's website, i.e. *http://infocenter.arm.com* for ARM.

Example#3

Docs: MCU

- Documentation on MCU describes the most important things you'll want to know: how all the peripherals work, how the system is clocked, how much flash and RAM does MCU have, memory map, pin multiplexing etc.
- It can be one document ("User manual", "User guide", "Technical reference guide", even "Datasheet") or a set of documents ("Clocking guide", "UART guide", "Ethernet guide" etc.).
- May contain some not so important for a programmer info like electrical characteristics, dimensions etc.
- Found on MCU vendor's website.

Example#4

Docs: board

- Main document on the board is the schematics.
- When reading schematics try to locate MCU first. If there are too many pins it may be divided into several blocks for convenience.
- Schematics will tell you what parts are connected to what pins of MCU, what is the frequency of external oscillator (if any), where you can find test points to ease debugging.
- Docs for complex parts can be found on part vendor’s website. Or in Google by part number (part number can be found in schematics).
- Possible external devices: memories (RAM, Flash, EEPROM), transceivers (RS-232/422/485, Ethernet, CAN), LEDs, buttons, LED/GPIO controllers etc.

Let’s get started

We will need

- Toolchain: compiler, linker, binutils, standard library
- Documentation on board, MCU, processor core
- Headers, startup and examples

Intermezzo: how processors work

RISC pipeline:

- Instruction fetch: take one instruction from instruction memory.
- Instruction decode: set control logic according to opcode, prepare arguments for the computation.
- Execute: perform the actual computation (addition, multiplication, comparison).
- Memory access: read/write to/from system bus (memory, peripherals).
- Writeback: write computation results back to register file.

The only interface between processor core and peripherals is system bus ("Memory"). Addresses that peripherals expose to system bus are called peripheral registers, as opposed to processor's internal registers.

Example#6

Headers

- Formally speaking, toolchain and hardware documentation are enough to program an MCU.
- Headers contain peripheral register address definitions and masks/bit numbers for individual bits/groups of bits in registers.
- Headers are made for specific MCUs or MCU families, so you'll need to look for them on MCU vendor's website.
- Unfortunately, you can rarely find a dedicated "headers" page. Usually they are contained in archives with example software for some evaluation board.
- Sometimes you'll even need to install vendor's IDE to get the headers.

Example#7

Startup

- Startup code is executed before main and prepares the code to execution: initializes variables in RAM, tunes system stack. Interrupt vectors are also normally defined in startup file.
- Startup code is mostly written in assembly, but C is also possible.
- On some architectures and on some toolchains this functionality is hidden. ARM toolchains are explicit.
- Again, processor manual is enough to write your own startup code, but a ready-made one will save you time and effort.

Example#8

Examples

- Most vendors provide example applications/drivers for their MCUs
- Don’t expect them to be bug-less.
- Don’t expect them even to work on your board.
- Apart from being buggy, they are often also hard to debug.
- They can be used as a reference in addition to docs when you’re stuck.
- Vendor’s examples are natural source of MCU headers and startup files.

Example#9

I’m confused

- You don’t need to fully understand all the details of how your toolchain, MCU or processor core works.
- You don’t need to read the whole documentation set (don’t do it, really).
- Approximate algorithm to get started:
  1. Get the toolchain.
  2. Get docs, headers, examples for your device.
  3. Choose **one** peripheral device to work with. Ideally, a GPIO pin connected to an LED.
  4. Write a simple program using the peripheral, use ready-made startup and headers.
  5. Make it compile. You will learn a lot about your toolchain on this stage.
  6. Make it work. You will learn a lot about your MCU, processor, toolchain and maybe some electronics.
  7. Congratulations! You are an embedded software developer now.

Hands-on

Let’s try it out.

Toolchain for x86_64 machines:
https://releases.linaro.org/components/toolchain/binaries/5.3-2016.05/arm-eabi/gcc-linaro-5.3.1-2016.05-x86_64_arm-eabi.tar.xz

Toolchain for 32-bit machines:
https://releases.linaro.org/components/toolchain/binaries/5.3-2016.05/arm-eabi/gcc-linaro-5.3.1-2016.05-i686_arm-eabi.tar.xz

Project template: https://github.com/OYTIS/mcu-workshop

Extra: make

GNU make helps you automate the build process.

Let’s go directly to the example.

Example#A