

# Building Bare-Metal ARM Systems with GNU

## Part 5: Fine-tuning the Application

Miro Samek

In this part I describe the options for fine-tuning the application by selective ARM/Thumb compilation and by placing hot-spot parts of the code in RAM. I also mention the

### 1. ARM/THUMB compilation

The compiler options discussed in the previous part of this article (the `CCFLAGS` symbol) specifically do not include the instruction set option (`-marm` for ARM, and `-mthumb` for THUMB). This option is selected individually for every module in the `Makefile`. For example, in the following example the module `low_level_init.c` is compiled to THUMB and module `blinky.c` is compiled to THUMB:

```
$(BINDIR)\low_level_init.o: $(BLDDIR)\low_level_init.c $(APP_DEP)
$(CC) -marm $(CCFLAGS) $(CCINC) $<

$(BINDIR)\blinky.o: $(BLDDIR)\blinky.c $(APP_DEP)
$(CC) -mthumb $(CCFLAGS) $(CCINC) $<
```

### 2. Reducing the Overhead of C++

The compiler options controlling the C++ dialect discussed in the previous part of this article (the `CPPFLAGS` symbol) are closely related to reducing the overhead of C++. However, disabling RTTI and exception handling at the compiler level is still not enough to prevent the GNU linker from pulling in some 50KB of library code. This is because the standard `new` and `delete` operators throw exceptions and therefore require the library support for exception handling. (The `new` and `delete` operators are used in the static constructor/destructor invocation code, so are linked in even if you don't use the heap anywhere in your application.)

Most low-end ARM-based MCUs cannot tolerate 50KB code overhead. To eliminate that code you need to define your own, non-throwing versions of global `new` and `delete`, which is done in the module `mini_cpp.cpp` located in the directory `cpp_blinky`.

```
#include <stdlib.h> // for prototypes of malloc() and free()

//.....
(1) void *operator new(size_t size) throw() { return malloc(size); }
//.....
(2) void operator delete(void *p) throw() { free(p); }
//.....
(3) extern "C" int __aeabi_atexit(void *object,
                                void (*destructor)(void *),
                                void *dso_handle)
{
    return 0;
}
```

**Listing 1** The `mini_cpp.cpp` module with non-throwing `new` and `delete` as well as dummy version of `__aeabi_atexit()`.

Listing 1 shows the minimal C++ support that eliminates entirely the exception handling code. The highlights are as follows:

- (1) The standard version of the operator `new` throws `std::bad_alloc` exception. This version explicitly throws no exceptions. This minimal implementation uses the standard `malloc()`.
- (2) This minimal implementation uses the standard `free()`.
- (3) The function `__aeabi_atexit()` handles the static destructors. In a bare-metal system this function can be empty because application has no operating system to return to, and consequently the static destructors are never called.

Finally, if you don't use the heap, which you shouldn't in robust, deterministic applications, you can reduce the C++ overhead even further. The module `no_heap.cpp` provides dummy empty definitions of `malloc()` and `free()`:

```
#include <stdlib.h> // for prototypes of malloc() and free()

//.....
extern "C" void *malloc(size_t) {
    return (void *)0;
}
//.....
extern "C" void free(void *) {
}
```

### 3. Placing the Code in RAM

As mentioned in part 1 of this article, placing strategic parts of the hot-spot code in RAM can significantly improve performance and reduce power dissipation of most ARM-based MCUs. The startup code and the linker script discussed in parts 2 and 3 of this article support the `.fastcode` section that is located in RAM, but is loaded to ROM and copied to RAM upon startup.

You have two options to assign individual functions to the `.fastcode` section:

1. Because each function is located in a separate section (see the `-ffunction-sections` compiler option described in part 4), you can explicitly locate the code for every function in the linker script for your applications. The linker scripts `blinky.ld` for the Blinky application provide an example how to locate the `Blinky_shift()` function in RAM.
2. You can assign any function to the `.fastcode.text` section, by means of the `__attribute__((section(".text.fastcode")))` directive. The module `blinky.c` provides an example for the `Blinky_flash()` function.

```
__attribute__((section(".text.fastcode")))
void Blinky_flash(Blinky *me, uint8_t n) {
    . . .
}
```

### 4. Coming Up Next

The provided code examples and the discussion should provide you with a head-start on any bare-metal ARM-based project with the GNU toolchain. The second part of this article will describe ARM exceptions and interrupt handling.

### 5. References

[1] Lewin A.R.W. Edwards, "Embedded System Design on a Shoestring", Elsevier 2003.

- [2] GNU Toolchain for ARM, CodeSourcery, [http://www.codesourcery.com/gnu\\_toolchains/arm](http://www.codesourcery.com/gnu_toolchains/arm).
- [3] GNU ARM toolchain, <http://www.gnuarm.com>.
- [4] GNU X-Tools™, Microcross, <http://www.microcross.com>.
- [5] ARM Projects, [http://www.siwawi.arubi.uni-kl.de/avr\\_projects/arm\\_projects](http://www.siwawi.arubi.uni-kl.de/avr_projects/arm_projects)

**Miro Samek** is the President of [Quantum Leaps, LLC](#), the provider of lightweight, state machine-based, event-driven frameworks for embedded systems. He is the author of "Practical Statecharts in C/C++" (CMP Books, 2002), has written numerous articles for magazines, and is a regular speaker at the Embedded Systems Conferences. He welcomes contact at [miro@quantum-leaps.com](mailto:miro@quantum-leaps.com).