# Embedded System Design

Booting a Cortex-M3 system from scratch

TaeWook Kim & SeokHyun Hong

Hanyang University

# Installing a compiler toolchain

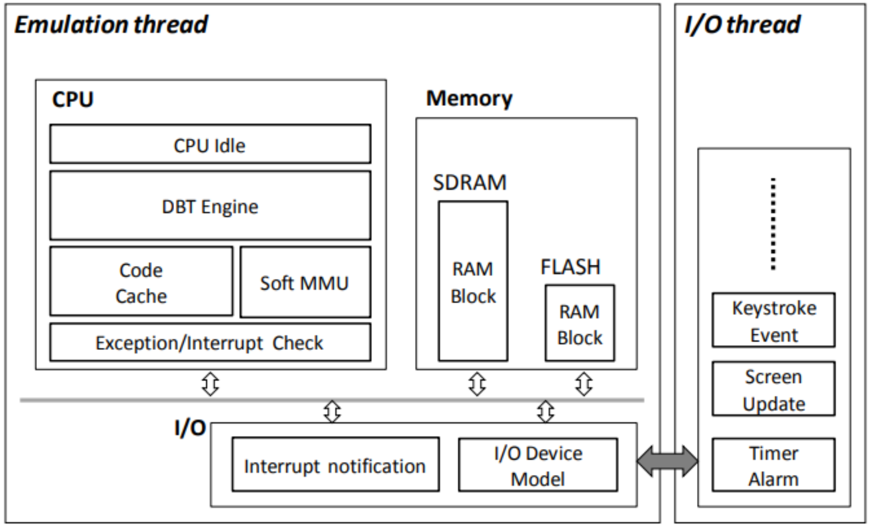
- $) apt search gcc-arm
  - lists available gcc-arm packages
- $) sudo apt install gcc-arm-none-eabi
  - installs the cross-compiler for Cortex-M processors

- FYI: naming of cross-compilers
  - [arch]-[vendor]-[os]-[abi] gcc/as/ld/objcpy/…
    - arch : target architecture
    - vendor : toolchain supplier
    - os : target OS
    - abi : ABI (Application Binary Convention)
  - Example
    - arm-linux-gnueabi-*
    - aarch64-linux-gnueabi-*
    - i686-apple-darwin10-*

# Installing QEMU

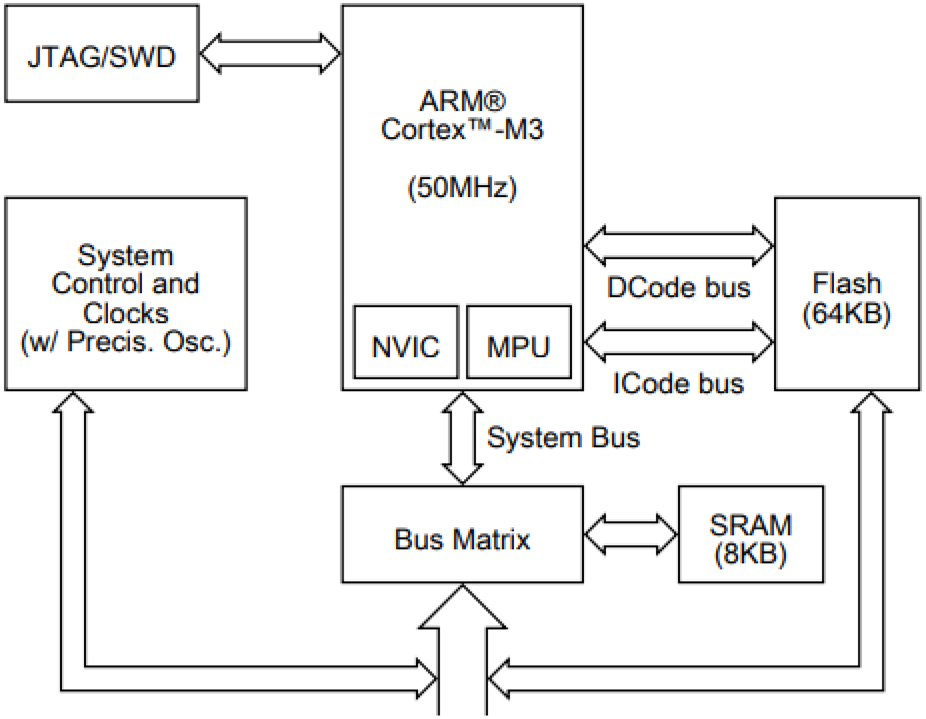
- $) apt search qemu-system
  - lists available qemu packages
- $) sudo apt install qemu-system-arm
  - installs qemu for arm architectures

- FYI: QEMU (Quick EMUlator)
  - DBT (Dynamic Binary Translation)-based system emulator

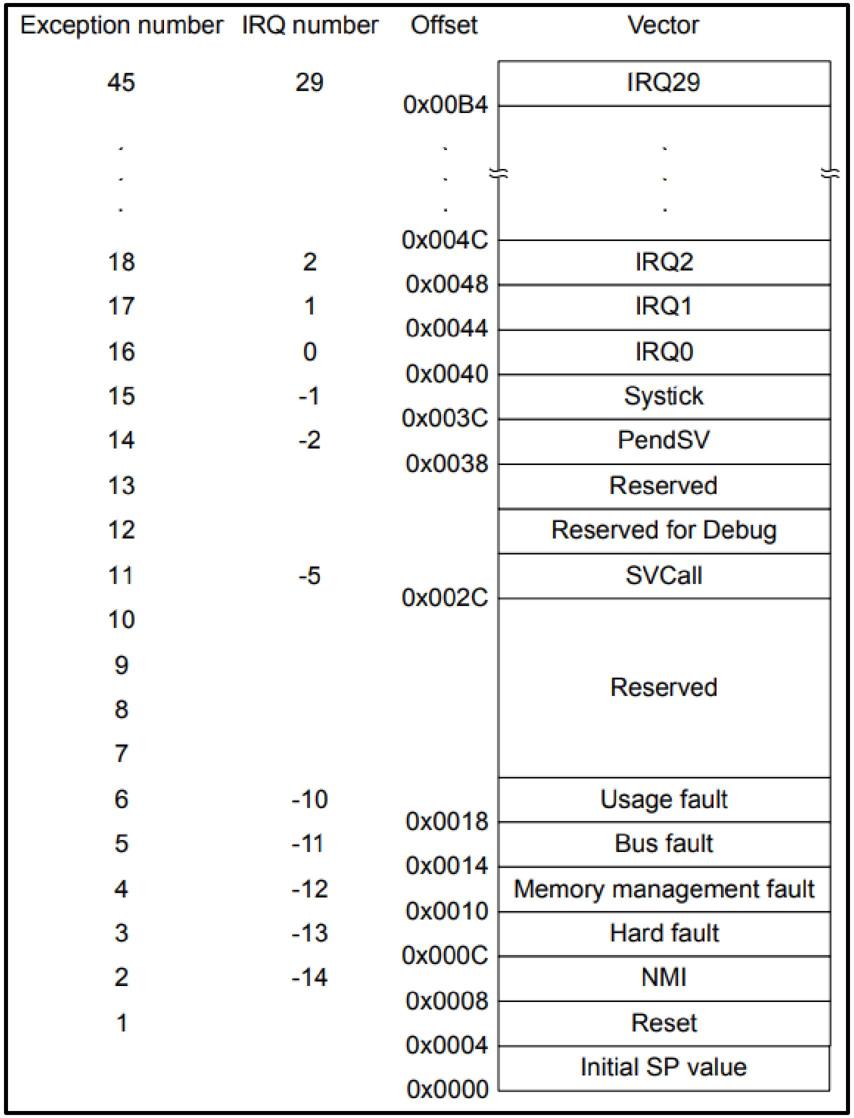


# Installing QEMU

- FYI: LM3S811
  - Cortex-M3 50MHz
  - 64KB Flash
  - 8KB SRAM



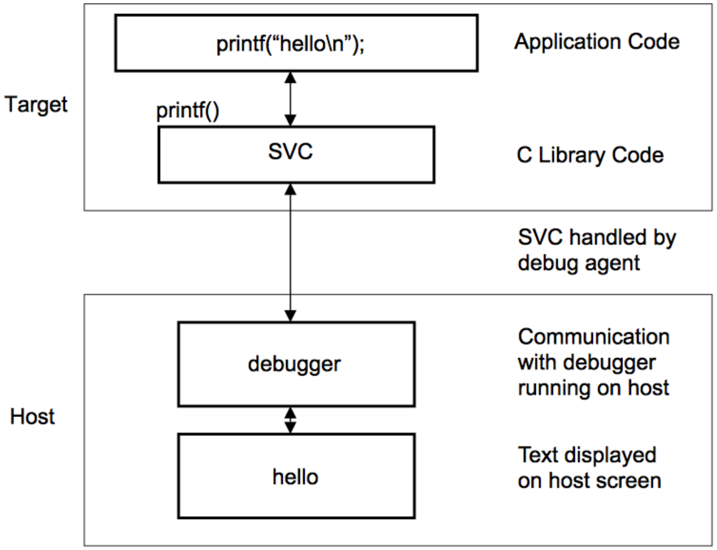
# The vector table of LM3S811



| Exception number | IRQ number | Offset | Vector |
|---|---|---|---|
| 45 | 29 | 0x00B4 | IRQ29 |
| .<br>.<br>. | | .<br>.<br>.<br>0x004C | .<br>.<br>. |
| 18 | 2 | 0x0048 | IRQ2 |
| 17 | 1 | 0x0044 | IRQ1 |
| 16 | 0 | 0x0040 | IRQ0 |
| 15 | -1 | 0x003C | Systick |
| 14 | -2 | 0x0038 | PendSV |
| 13 | | | Reserved |
| 12 | | | Reserved for Debug |
| 11 | -5 | 0x002C | SVCall |
| 10 | | | Reserved |
| 9 | | | |
| 8 | | | |
| 7 | | | |
| 6 | -10 | 0x0018 | Usage fault |
| 5 | -11 | 0x0014 | Bus fault |
| 4 | -12 | 0x0010 | Memory management fault |
| 3 | -13 | 0x000C | Hard fault |
| 2 | -14 | 0x0008 | NMI |
| 1 | | 0x0004 | Reset |
| | | 0x0000 | Initial SP value |

# Semihosting

- Semihosting enables code running on an ARM target to use IO facilities of a host.
- We can use this mechanism in a similar to system calls.
  - SVC 0x123456 : In ARM state for all architectures
  - BKPT 0xAB : For ARMv6-M and ARMv7-M, Thumb state only
    - r0: the operation type
    - r1: points to other parameters


printf("hello\n");
Application Code
Target
printf()
SVC
C Library Code
SVC handled by debug agent
debugger
Communication with debugger running on host
Host
hello
Text displayed on host screen

# Semihosting

- Examples
  - SYS_WRITE (0x05)
    - writes the buffered data to a file opened with SYS_OPEN
    - params (passed by r1)
      - word 1
        - a file descriptor opened with SYS_OPEN
      - word 2
        - a start memory address of the data
      - word 3
        - the length of the data
    - return
      - r0 is zero if there is no error.
  - SYS_TIME (0x11)
    - returns the number of seconds since 00:00 Jan 1, 1970.
    - return
      - r0 contains the number of seconds

# Example: A “Hello World “ using semihosting

- startup.c

```
#include <stdint.h>

extern void main(void);
void reset_handler(void)
{
        /* jump to C entry point */
        main();
}

__attribute((section(".isr_vector")))
uint32_t *isr_vectors[] = {
        0x10000,
        (uint32_t *) reset_handler,    /* code entry point */
};
```

# Example: A “Hello World “ using semihosting

- semi.c

```
#include <stdint.h>

static int semihost_call(int service, void *opaque)
{
        register int r0 asm("r0") = service;
        register void *r1 asm("r1") = opaque;
        register int result asm("r0");
        asm volatile("bkpt 0xab"
                : "=r" (result) : "r" (r0), "r" (r1));
        return result;
}

enum SEMIHOST_SVC {
        SYS_WRITE = 0x05,
};

void main(void)
{
        char message[] = "Hello World!\n";
        uint32_t param[] = { 1, (uint32_t) message, sizeof(message) };
        semihost_call(SYS_WRITE, (void *) param);
        while (1);
}
```

# Example: A “Hello World “ using semihosting

- semi.ld

```
ENTRY(reset_handler)

MEMORY
{
        FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 64K
}

SECTIONS
{
        .text :
        {
                KEEP(*(.isr_vector))
                *(.text)
        } >FLASH
}
```

# Example: A “Hello World “ using semihosting

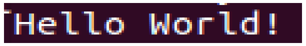
- Makefile

```
CROSS_COMPILE ?= arm-none-eabi-
CC := $(CROSS_COMPILE)gcc
CFLAGS = -fno-common -O0 -std=gnu99 \
	-mcpu=cortex-m3 -mthumb \
	-T semi.ld -nostartfiles \

TARGET = semi.bin
all: $(TARGET)

$(TARGET): semi.c startup.c
	$(CC) $(CFLAGS) $^ -o semi.elf
	$(CROSS_COMPILE)objcopy -Obinary semi.elf semi.bin
	$(CROSS_COMPILE)objdump -S semi.elf > semi.list

qemu: $(TARGET)
	@qemu-system-arm -M ? | grep lm3s811evb >/dev/null || exit
	@echo "Press Ctrl-A and then X to exit QEMU"
	@echo
	qemu-system-arm -M lm3s811evb -semihosting -nographic -kernel semi.bin

clean:
	rm -f *.o *.bin *.elf *.list
```

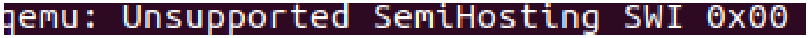
# Build and run the program

- $) make clean
  - deletes existing files
- $) make
  - builds the binary
- $) make qemu
  - runs the binary on QEMU

- Unfortunately, it doesn’t work!
  - The result we want to see is …, but ..



```
Hello World!
```



```
qemu: Unsupported SemiHosting SWI 0x00
```

- What is the matter?
  - Wrong memory layout

# Abnormal stack location

- $) cat semi.list
  - We can find push/pop instructions
    - This binary must use the stack.
    - Where was the stack created?
- Let’s look at startup.c again

```
#include <stdint.h>

extern void main(void);
void reset_handler(void)
{
        /* jump to C entry point */
        main();
}

__attribute((section(".isr_vector")))
uint32_t *isr_vectors[] = {
        0x10000,
        (uint32_t *) reset_handler,    /* code entry point */
};
```

start address of the stack

# Debugging

- Let's check though gdb
- gdb wants the binary to contain debugging symbols.
- Note that we can run gdb on the host, and the target binary runs on QEMU.
  - Remote gdb debugging mechanism is required.
    - Run the gdbserver on QEMU
    - Connect to the gdbserver from the gdb cline in the host through a TCP connection.

# Debugging

- Fix Makefile

```
CROSS_COMPILE ?= arm-none-eabi-
CC := $(CROSS_COMPILE)gcc
CFLAGS = -fno-common -O0 -std=gnu99 \
          -mcpu=cortex-m3 -mthumb \
          -T semi.ld -nostartfiles -g \

TARGET = semi.bin
all: $(TARGET)

$(TARGET): semi.c startup.c
         $(CC) $(CFLAGS) $^ -o semi.elf
         $(CROSS_COMPILE)objcopy -Obinary semi.elf semi.bin
         $(CROSS_COMPILE)objdump -S semi.elf > semi.list

qemu: $(TARGET)
         @qemu-system-arm -M ? | grep lm3s811evb >/dev/null || exit
         @echo "Press Ctrl-A and then X to exit QEMU"
         @echo
         qemu-system-arm -M lm3s811evb -semihosting -nographic -kernel semi.bin

gdb: $(TARGET)
         @qemu-system-arm -M ? | grep lm3s811evb >/dev/null || exit
         @echo "Press Ctrl-A and then X to exit QEMU"
         @echo
         qemu-system-arm -M lm3s811evb -s -S -semihosting -nographic -kernel semi.bin

clean:
         rm -f *.o *.bin *.elf *.list
```

# Debugging

- $) make gdb(The terminal will probably stop when you run it.)
  - run the program
- $) gdb-multiarch(Run on new terminal)
  - run gdb
  - Installation: $) apt install gdb-multiarch
- (gdb) file semi.elf
  - Load debugging symbols from the ELF file.
- (gdb) target remote:1234
  - Establish a connection to the gdbserver on QEMU.

- $) gdb-multiarch semi.elf -ex="target remote:1234“
  - Altogether the above.

# Debugging

- Other ways
  - $) gdb-multiarch semi.elf -ex="target remote:1234“

  or

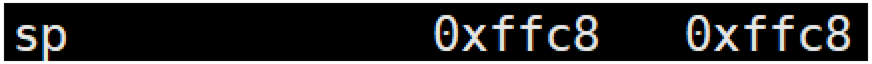
  - $) gdb-multiarch -x semi.gdb
    - semi.gdb

```
file semi.elf
target remote:1234
```

# Debugging

- (gdb) break main
  - Sets break point at main()
- (gdb) continue
  - Continues the execution.
- (gdb) info reg
  - Shows the values of all registers.

- ‘sp’ is pointing to an invalid memory region(FLASH).



```
sp                     0xffc8    0xffc8
```

- We need to locate the stack in the SRAM.

# Set up stack

- Fix startup.c

```
#include <stdint.h>

extern uint32_t _estack;

extern void main(void);
void reset_handler(void)
{
        /* jump to C entry point */
        main();
}

__attribute((section(".isr_vector")))
uint32_t *isr_vectors[] = {
        (uint32_t *) &_estack,
        (uint32_t *) reset_handler,   /* code entry point */
};
```

# Set up stack

- Fix semi.ld

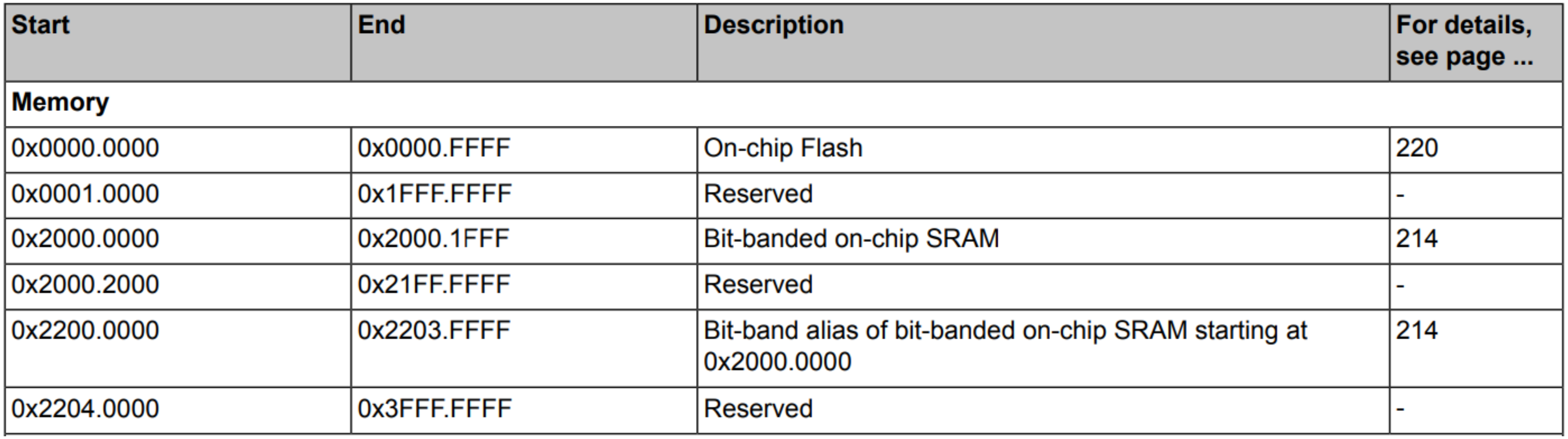
```
ENTRY(reset_handler)

MEMORY
{
        FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 64K
        RAM (rwx) : ORIGIN = 0x20000000, LENGTH = 8K
}

SECTIONS
{
        .text :
        {
                KEEP(*(.isr_vector))
                *(.text)
        } >FLASH

        _estack = ORIGIN(RAM) + LENGTH(RAM);
}
```

  - KEEP: do not discard this section

# Set up stack

- LM3S811’s Memory map

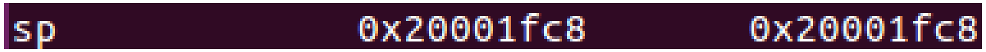
Table 2-4. Memory Map



| Start | End | Description | For details, see page ... |
|---|---|---|---|
| **Memory** | | | |
| 0x0000.0000 | 0x0000.FFFF | On-chip Flash | 220 |
| 0x0001.0000 | 0x1FFF.FFFF | Reserved | - |
| 0x2000.0000 | 0x2000.1FFF | Bit-banded on-chip SRAM | 214 |
| 0x2000.2000 | 0x21FF.FFFF | Reserved | - |
| 0x2200.0000 | 0x2203.FFFF | Bit-band alias of bit-banded on-chip SRAM starting at 0x2000.0000 | 214 |
| 0x2204.0000 | 0x3FFF.FFFF | Reserved | - |

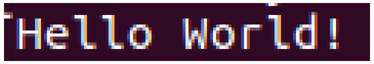
# Set up stack

- $) make gdb
- $) gdb-multiarch -ex="target remote:1234“
- (gdb) file semi.elf
- (gdb) break main
- (gdb) continue
- (gdb) info reg



```
sp             0x20001fc8       0x20001fc8
```

- Now, ‘sp’ is within a valid memory region, i.e., SRAM



```
Hello World!
```

- And, the program runs correctly!