Keep learning.
It's the Washington Way.

EMBSYS 320
Programming with Embedded & Real-Time Operating Systems

Instructor: Nick Strathy, nstrathy@uw.edu
TA: Gideon Lee, gideonhlee@yahoo.com

1/11/2021

PROFESSIONAL & CONTINUING EDUCATION
UNIVERSITY of WASHINGTON

Introductions

when u first start talking to someone
and u act all proper bc u ain't sure
when u can start being weird

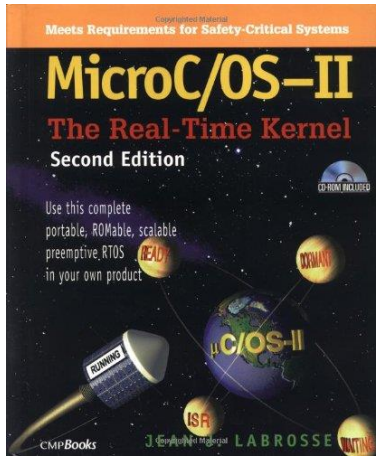
Course requirements

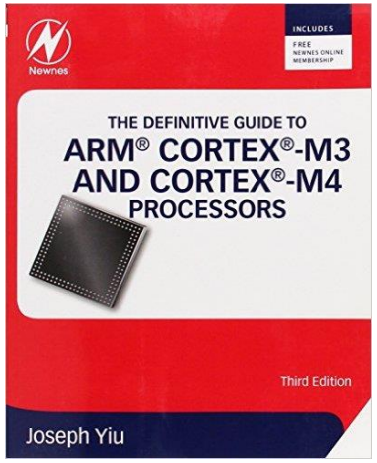
- Participation - must attend at least 6 of the 10 lectures
  - Attendance is verified using a weekly class quiz which does NOT count for your grade
- 60% for 5 assignments (individual effort)
- 40% for course project (individual effort)

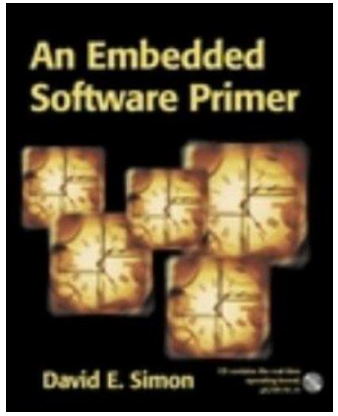
- Don’t post solutions in the discussion forum
- Using code that you find on the web is fine – just mention your source.

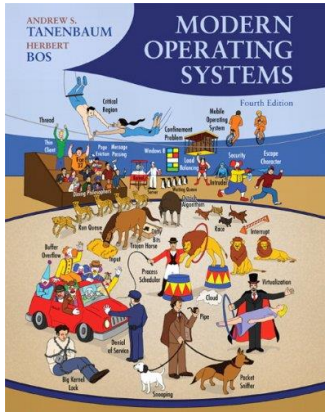
Textbooks

- Required
  - MicroC/OS-II The Real-Time Kernel, Second Edition, 2002, Jean J. Labrosse. The PDF is available online: https://doc.micrium.com/download/attachments/10753158/100-uC-OS-II-002.pdf
  - The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors 3rd ed., Joseph Yiu, Elsevier, 2014.
- Useful for reference
  - An Embedded Software Primer, David E. Simon, Addison-Wesley 1999
  - Modern Operating Systems Fourth Edition, Andrew S. Tanenbaum, Herbert Bos, Prentice Hall, 2014









Looking ahead

- Assignments are due Sundays
- 1/24/2021 11:59 PM Assignment 1 due
- We have two holidays:
  - January 18 no class
  - February 15 – we will have a class – attendance not compulsory – will be recorded – allows us to finish the course one week earlier.
- Please make sure your Canvas settings are configured to receive email announcements and discussions for this course

Handy page of reference links on Canvas

https://canvas.uw.edu/courses/1424641/pages/embsys-320-resources

- Contains
  - Links to the reference documents for this course
  - Link to the version of IAR EWARM for this course:
    - https://1drv.ms/u/s!Ahm13wmWGad-jCcuCl4eTXwHQ5iQ?e=6eFvZy
    - Use that link if you need to (re)install IAR EWARM
    - Reason to use that link: if you get a newer version from IAR, your projects will not load into my older version.

EMBSYS 320 Goals

- Embedded Operating Systems
  - Port MicroC/OS-II to the STM32L4 Discovery development board
  - Program a multitasking system
  - Develop event-driven systems
  - Develop hardware/software APIs
- Device Drivers
  - Understand and use a standard device driver API
  - Add a driver to a device driver framework
- Course Project
  - Put it all together to document, design, develop, test and debug an MP3 player.

Week 1 Overview

- Embedded Operating System Concepts Part 1 (Labrosse Chapter 2)
  - Foreground/Background Systems
  - Shared resources
  - Critical sections
  - Multitasking
  - Tasks
  - Context switching
- ARM Review (Yiu Chapters 4, 8)
  - States and Modes
  - Registers
  - Assembly language instructions
  - Interrupt semantics
- Assignment 1 – uDebugger tool (due in 1 week)

Embedded operating system concepts

- What is an operating system (OS)?
  - Fundamentally it provides a layer of software abstraction above the hardware
  - i.e. provides a set of software interfaces between applications and the hardware

Embedded operating system concepts

- Before we get to Embedded OSes, what is a “Platform OS”?
  - Examples: Linux, Mac OS X, Windows, etc.
  - General purpose platform for application programs to run on
  - File system
  - Multitasking
  - Memory management
  - Security – user mode vs kernel mode
  - etc.

  - Cost of generality/flexibility: no hard timing guarantees

Embedded operating system concepts

- An Embedded OS is some subset of a Platform OS
  - Features of the OS are restricted by
    - Limited hardware – small memory, small processor, limited I/O capabilities, etc.
    - Only a narrow range of functionality is required for such devices as thermostat, microwave oven, wrist watch, etc.
  - Hard real time commitments may be required
    - Requires a real-time OS
      - Guaranteed interrupt latency
      - Examples: medical devices, flight controls, antilock brakes, etc.

Embedded operating system concepts

- Example: Foreground/Background OS
  - A simple embedded OS
  - The OS is modeled by two components, a “foreground” and “background” component
  - The background component consists of a “super loop” that runs the ongoing monitoring and functionality of the system in an infinite loop
  - The foreground component consists of interrupt service routines (ISRs) that provide timely responses to external inputs
  - In the simplest case where nested interrupts are not supported we have at most 2 “threads of execution” – the background thread and the foreground (ISR) thread.

Embedded operating system concepts

Foreground/Background OS example: Thermostat

- Super loop runs in background, interrupt service routines run in foreground

```
uint32_t currentTemperature;
uint32_t temperatureSetting;
while (1) {  /* super loop - runs in background */
    /* update currentTemperature from sensor */
    /* update display from currentTemperature */
    /* update display from temperatureSetting */
    /* update heater on/off signal */
}
ISR_ButtonUpArrow { /* runs in foreground */
    temperatureSetting++;
}
ISR_ButtonDownArrow { /* runs in foreground */
    temperatureSetting--;
}
```

Embedded operating system concepts

Shared resource problems in multithreaded systems

- Any system that allows interrupts can have multiple threads of execution
- Problems can easily arise when multiple threads access the same resource e.g. global variable race conditions
- The integrity of the shared resource can be corrupted if interrupts occur while multiple threads are operating on the shared resource
- Thermostat example
  - Shared resource: temperatureSetting (global variable)
  - Say we want to enforce a saturation value of 90 degrees on temperatureSetting to prevent it from being set any higher
  - Need to correctly handle updates on the temperatureSetting global variable
  - Let's look at a wrong way of handling the updates …

Embedded operating system concepts

- Example of buggy update of a shared resource
- The intention was to stop temperatureSetting from exceeding 90

```
uint32_t temperatureSetting;
uint32_t i;
while (1) {
    if (temperatureSetting == 91) {
        for (i=10000;i;i--); /* artificial delay */
        temperatureSetting = temperatureSetting - 1;
    }
}

ISR_ButtonUpArrow { /* runs in foreground */
    temperatureSetting++;
}
```

If the setting happens to be 90 and we click the up-arrow button rapidly we can in principle interrupt the if block three times in quick succession and increment the setting up to 92. We've even added an artificial delay to make sure it happens!

Embedded operating system concepts

A fix for this instance of buggy resource sharing

```
uint32_t i;
while (1) { /* super loop */
    if (temperatureSetting == 91 > 90) {
        for (i=10000;i;i--); /* artificial delay */
        temperatureSetting = temperatureSetting - 1;
        temperatureSetting = 90;
    }
}
ISR_ButtonUpArrow { /* runs in foreground */
    temperatureSetting++;
}
```

Now, even though the if block gets interrupted it will always ensure that the setting is 90 or less.

- however this problem and its solution are ad hoc, so …

Embedded operating system concepts

Let's get more formal about handling shared resources.

Critical section:

- Definition: a critical section is a section of code that accesses a shared resource and may be executed by only one thread at a time.
- Critical sections are said to be *atomic*, meaning a CS cannot be executed by more than one thread at a time.
- A CS enforces *mutual exclusion* meaning any thread executing in a CS excludes all other threads from entering the CS.
- Even if a thread is interrupted during a critical section, no other thread may enter the CS until the first thread has resumed and exited it.

Embedded operating system concepts

Example of code that requires a critical section

- Code that may be executed by multiple threads and accesses a shared resource needs to be thread safe

```
int gCount = 0; // global var
...
    // should be in a critical section:
    gCount++;
```

Embedded operating system concepts

Some ways of enforcing mutual exclusion in critical sections

- Disable interrupts: this is the easiest way and is fine if critical sections are short and don't interfere with real time constraints.

```
int gCount = 0; // global var
...
    disableInterrupts(); // call a function that disables interrupts
        gCount++;
    enableInterrupts();  // call a function that enables interrupts
```

Embedded operating system concepts

Some ways of enforcing mutual exclusion in critical sections

Test-and-set

- To enter the CS, a thread polls a shared variable x whose default value is 0
- Any thread desiring to enter the CS should only do so if x==0
- If x==0, the thread sets x=1 and enters the CS otherwise tries again later
- On exit from the CS the thread sets x=0
- Problem: the thread can’t get interrupted between testing and setting
- Typically implemented at the assembly language level using an atomic SWAP instruction (ARM7 provides SWP – swap a word from memory with a register without interruption)

Embedded operating system concepts

Some ways of enforcing mutual exclusion in critical sections

Sample critical section code using test-and-set “spin lock”

```
int gCount = 0; // global var to be protected
int gTSLock = 0; // global test-and-set lock var
...
    while (testAndSet(&gTSLock) != 0) { /* spin */ };
        // if we arrive here we are in critical section
        gCount++;
    gTSLock = 0;
```

Embedded operating system concepts

Some ways of enforcing mutual exclusion in critical sections

test-and-set implementation using swap instruction (not on Cortex-M)

```
/* uint32_t testAndSet(uint32_t *gTSLock) */
.func TestAndSet
testAndSet:
    ldr     r2, =1          /* get ready to swap a 1 into memory */
    swp     r1, r2, [r0]    /* swap: *gTSLock = r2, r1 = *gTSLock */
    mov     r0, r1          /* r0 = previous value of *gTSLock */
    bx      lr
    .endfunc
.end
```

Embedded operating system concepts

Some ways of enforcing mutual exclusion in critical sections

Test-and-set implementation for Cortex M4

- Roughly the same sequence but more complicated to allow for a multi-processor environment
- Instead the atomic swap is achieved with an instruction pair
  - LDREX – load exclusive – gets current lock value
  - STREX – store exclusive – stores value to lock and returns status

Embedded operating system concepts

Some ways of enforcing mutual exclusion in critical sections

Cortex-M4 does not have swap instruction. Instead: LDREX/STREX

```
get_lock
   LDR R0, =Lock_Variable ; Get address of
                          ; lock variable
   LDREX R1, [R0] ; Read current lock value
   DMB ; Data Memory Barrier, Make sure
       ; memory transfer completes before
       ; next one starts
   CBZ R1, try_lock
   B get_lock ; Lock was set by another task,
              ; try again
```

```
try_lock
   MOVS R1, #1
   STREX R2, R1, [R0] ; Try lock by writing 1, if
     ; exclusive access failed, return status is 1 and
     ; the write is not carried out
   DMB ; Data Memory Barrier
   CBZ R2, lock_done ; Return status is 0,indicates
                     that lock process succeeded
   B get_lock ; locking process failed, retry
lock_done
   BX LR ; return
```

Reference: https://www.arm.com/files/pdf/Cortex-M3_programming_for_ARM7_developers.pdf

Embedded operating system concepts

Some ways of enforcing mutual exclusion in critical sections

- Semaphore (more later, but for now):
  - A more comprehensive thread synchronization mechanism invented by Edsger Dijkstra in the mid-1960s.
  - An operating system service consisting of a counter variable, a thread queue, and methods for operating on them.
  - The counter is initialized to 1 for a mutually exclusive critical section
  - If the counter is greater than 0, a thread may atomically decrement it and enter the CS
  - If the counter is 0, any thread wishing to enter the CS is suspended and added to the semaphore's queue
  - When a thread exits the CS, it attempts to atomically increment the counter
  - If there are any threads in the queue, the first is resumed and enters the CS, otherwise the counter is incremented

Embedded operating system concepts

Some ways of enforcing mutual exclusion in critical sections

Sample critical section code using a mutual exclusion semaphore

```
int gCount = 0; // global var
Semaphore mySem;
…
    semInitialize(mySem, 1); // initialize mySem’s counter to 1
…
    semWait(mySem);  // decrement mySem’s counter
        gCount++;    // safely update the shared resource
    semSignal(mySem); // increment mySem’s counter
```

Embedded operating system concepts

Thread, Process, Task

- Thread
  - A light weight unit of data and code that typically shares data and code with other threads
- Process
  - a heavy weight unit of data and code found typically in a platform OS rather than an embedded OS
  - May spawn multiple threads of its own which execute within the process
  - Memory and other resources are *not shared* between processes
  - Example: a web browser application
- Task
  - Depending on the context, either a process or a thread. In the context of this course it is a thread. MicroC/OS-II uses “task”, so we will usually use that term.

Embedded operating system concepts

Multitasking

- A disciplined way of *scheduling* and *switching* the CPU between multiple *tasks*.
- Same as multithreading
- Simplifies the design, implementation and management of complex interconnected threads of execution

Embedded operating system concepts

Task

- Same as *thread* in this course
- A unit of data and code that can be scheduled to execute on the CPU
- Typically, an infinite loop that handles one of the subdivisions of work in the overall embedded system. Each cycle through the loop handles one instance of the work unit after which the thread is suspended till it is needed again
- Task Control Block (TCB)
  - The data structure used by a multitasking system to manage each task
  - At a minimum it contains
    - Stack pointer for the task's own stack
    - Status of the task

Embedded operating system concepts

Task status (using MicroC/OS-II terminology)

- dormant – in memory but not available for scheduling
- ready – available for scheduling
- running – has control of the CPU – only one task at a time is ever in this state on a single processor system
- waiting – known to the scheduler but waiting for an event
- ISR (interrupted) – was running but currently waiting for ISR to complete

Embedded operating system concepts

Context switching

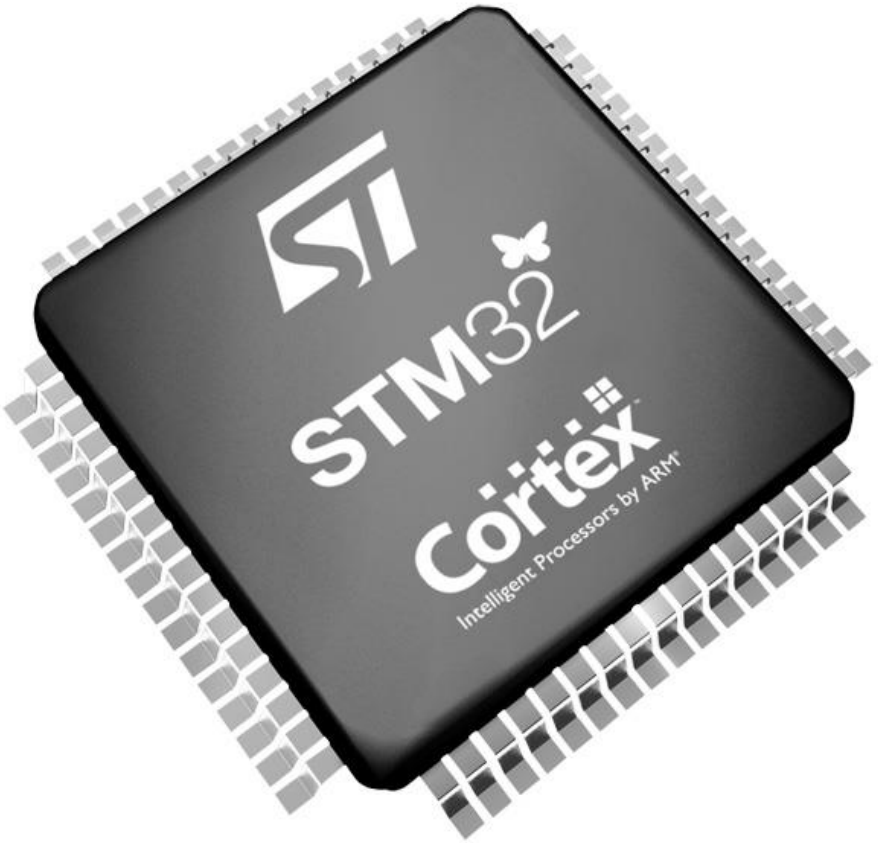
Term for the sequence of steps for transferring control of the CPU from one task to another

1. Store all the registers (context) of the current task in the task's private memory area
2. Restore the registers of the new task from its private memory area and resume execution of the new task.

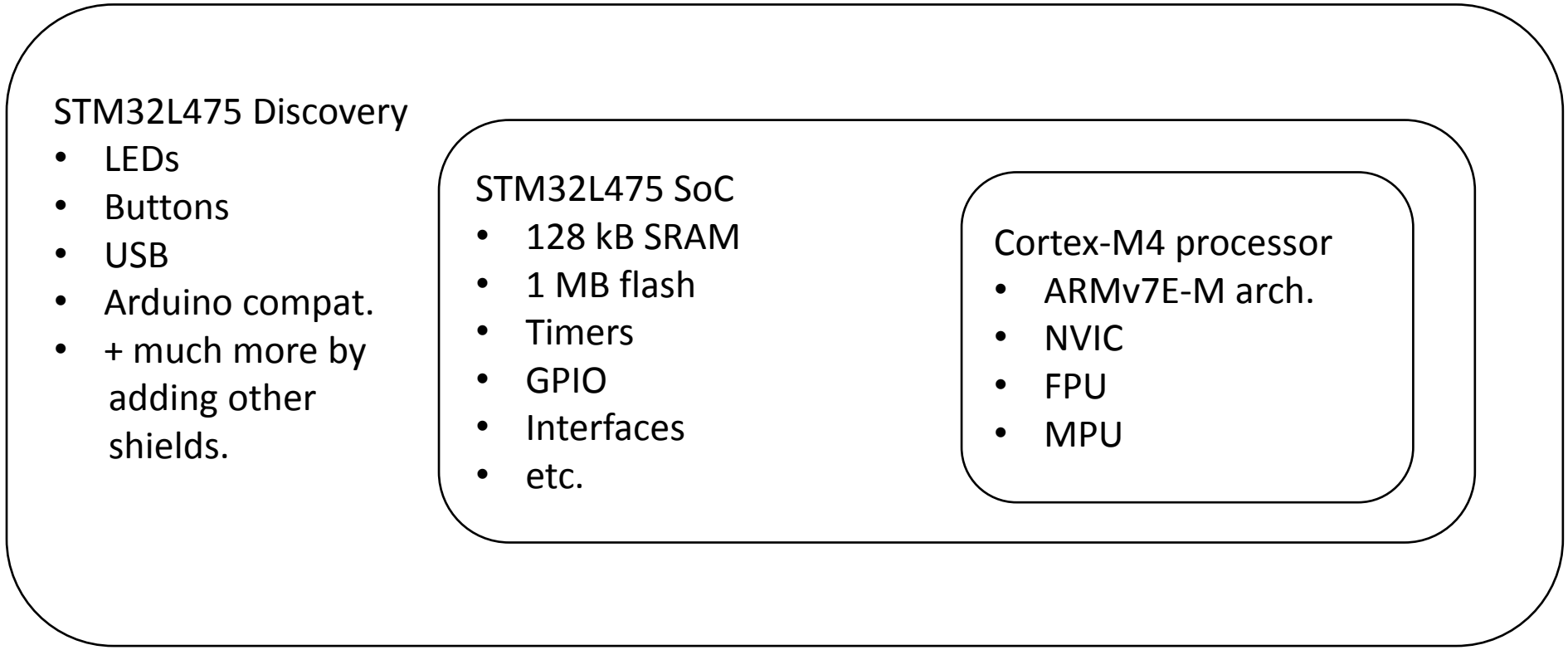
Quiz and break

- Do this evening’s online quiz in Canvas
- Then take a brief break

ARM Cortex-M4 review



STM32 Discovery Kit



ARM Cortex-M4 processor

- 2010 original release
- Three-stage instruction pipeline: fetch, decode, execute
- Harvard bus architecture: instruction-fetch, data-read/write in parallel
- 32-bit addressing, 4GB memory space
- Nested Vectored Interrupt Controller (NVIC): automatically saves a large subset of the context when handling interrupts
- Thumb (16-32-bit) instruction set: denser code than ARM (32-bit-only) instruction set
- Many more features

Cortex-M4 States and Modes

The processor is always in one of two possible operation states, and one of two possible operation modes. Additionally, it can be in one of two access levels which controls which registers and instructions are accessible.

Operation States

- Debug state: the processor is halted by the debugger after hitting a breakpoint.
- Thumb state: the processor is executing instructions. It only executes Thumb (16-32-bit) instructions. Other ARM processors can execute ARM (32-bit) instructions.

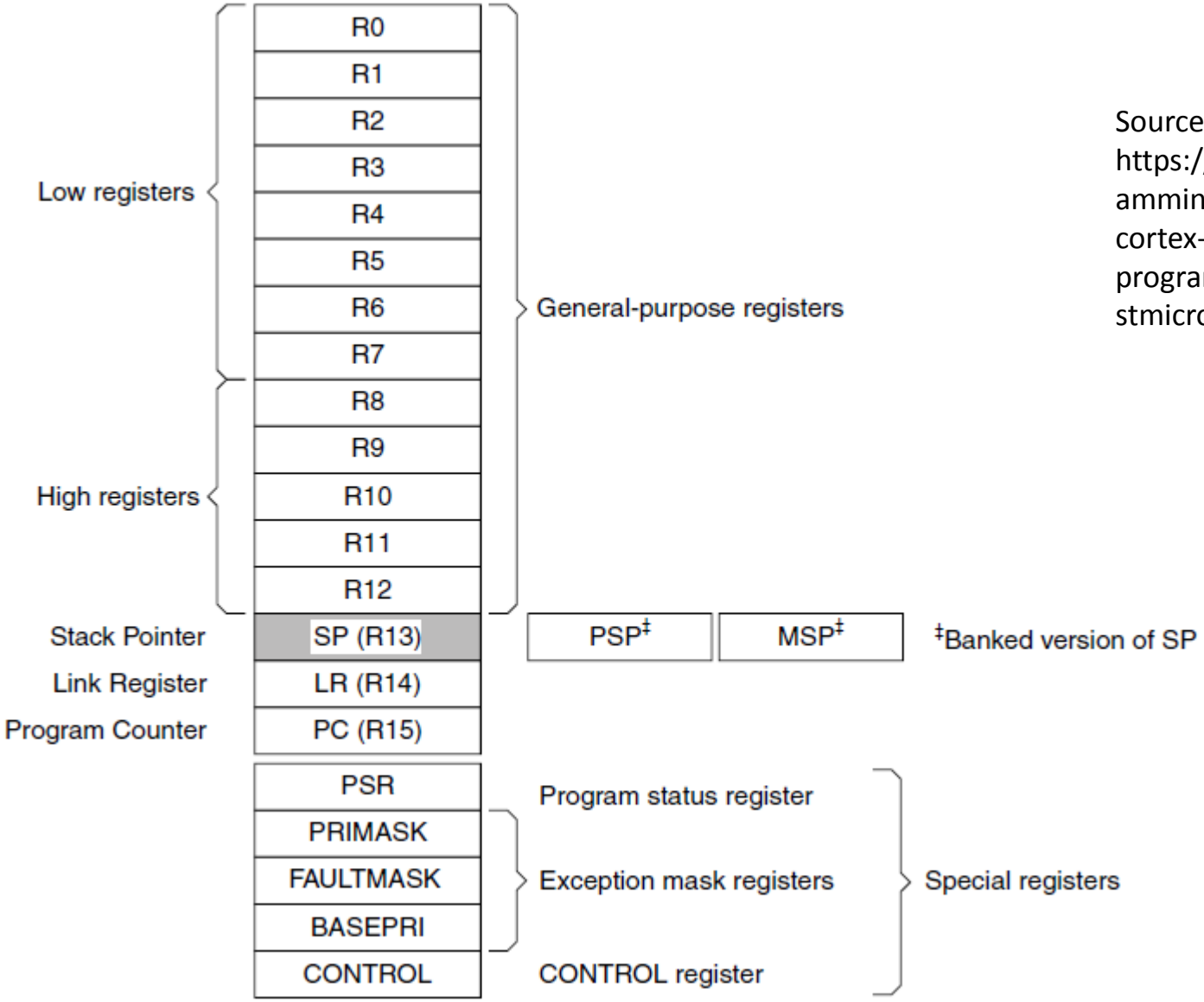
Operation Modes

- Handler mode: This mode is entered when an interrupt is triggered.
- Thread mode: this mode is entered on reset and upon returning from an interrupt handler.

Access Levels

- Privileged level: This level is always entered on reset and whenever Handler mode is entered.
- Unprivileged level: this level may be entered when in privileged level however the only way to enter privileged level from unprivileged is by triggering an interrupt.

Cortex-M4 Core Registers



Source: https://www.st.com/resource/en/programming_manual/dm00046982-stm32-cortex-m4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf

Cortex-M4 Core Registers

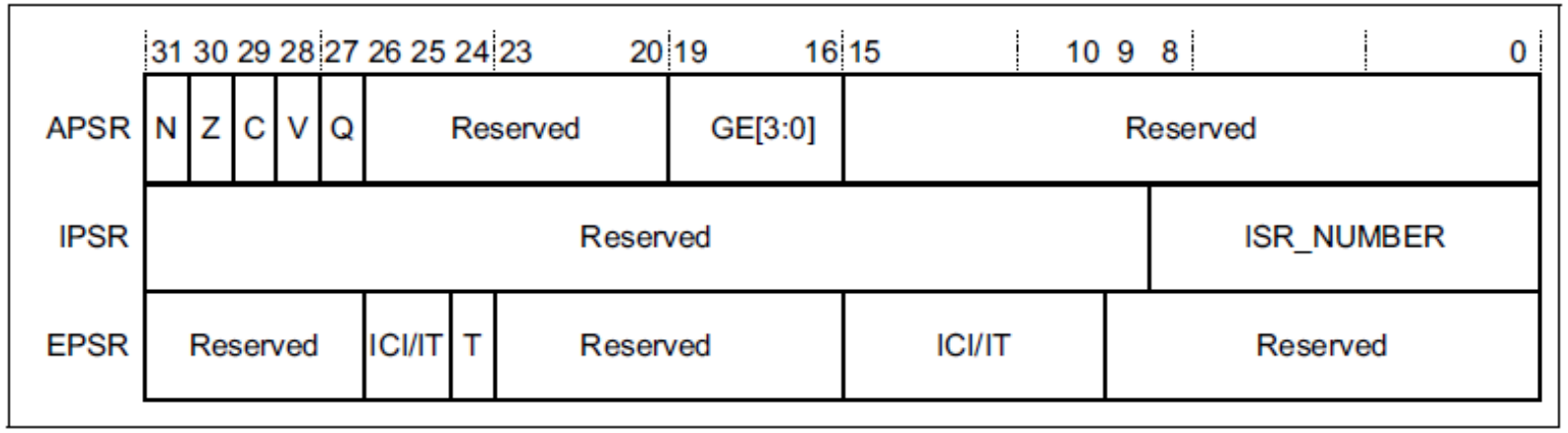
General Purpose registers (R0-R12)

- Accessible in Thread and Handler mode and privileged and unprivileged level

Stack Pointer (SP)

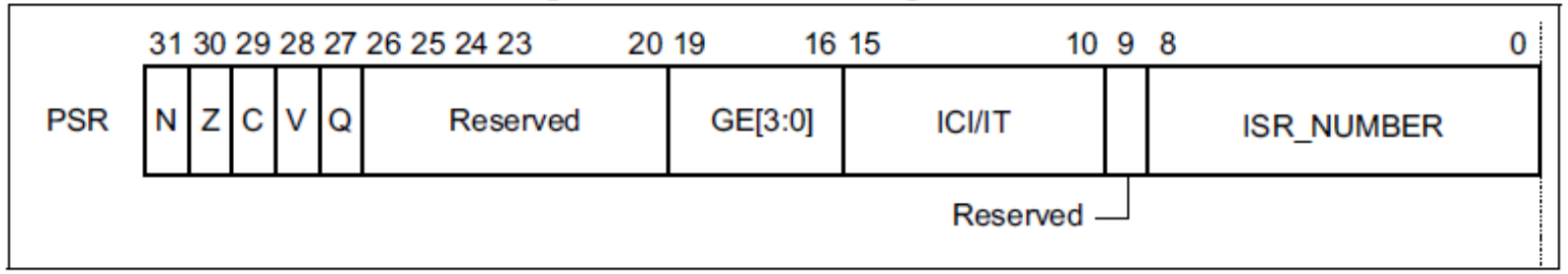
- MSP, SP_main: accessible only in privileged level, so always accessible in Handler mode, only accessible in privileged Thread mode
- PSP, SP_process: accessible in either privileged or unprivileged level. This is the stack pointer used when in unprivileged Thread mode.
- Unless your application is a multi-user system you probably don’t need to run in unprivileged thread mode and thus don’t need to use SP_process.
- For our applications we will only run at privileged level using the main stack pointer. This means handlers and application threads will all use the main stack pointer.

Cortex-M4 Core Registers



Source: https://www.st.com/resource/en/programming_manual/dm00046982-stm32-cortex-m4-mcus-and-mpus-programming-manual-stmicroelectronics.pdf

Figure 4. PSR bit assignments



PSR reset value: 0x01000000 which means thumb state, no interrupts active, no flags set

Cortex-M4 Core Registers

Exception Mask Registers (PRIMASK, FAULTMASK, BASEPRI)

- Privileged level required to access

Priority Mask (PRIMASK)

- 1-bit: 0 means no effect, 1 means all exceptions with configurable priority are prevented. Processor faults are still enabled, eg bad memory address.

Fault Mask (FAULTMASK)

- 1-bit: 0 means no effect, 1 means all exceptions except for non-maskable interrupt (NMI) are prevented.

Base Priority Mask (BASEPRI)

- 8-bits. 0x00 means no effect, otherwise the processor does not process any exception with a priority value greater than or equal to BASEPRI.

Cortex-M4 Core Registers

Control Register (CONTROL)

- Privileged level required to access

Bits	Function
Bits 31:3	Reserved
Bit 2	**FPCA:** Indicates whether floating-point context currently active: 0: No floating-point context active 1: Floating-point context active. The Cortex-M4 uses this bit to determine whether to preserve floating-point state when processing an exception.
Bit 1	**SPSEL:** Active stack pointer selection. Selects the current stack: 0: MSP is the current stack pointer 1: PSP is the current stack pointer. In Handler mode this bit reads as zero and ignores writes. The Cortex-M4 updates this bit automatically on exception return.
Bit 0	**nPRIV:** Thread mode privilege level. Defines the Thread mode privilege level. 0: Privileged 1: Unprivileged.

ARMv7-M Assembly Language

Review of a subset of instructions

- Moving data within the processor
- Moving data to/from memory
- Stack operations
- Arithmetic operations
- Branch instructions
- Conditional execution suffixes

ARMv7-M Assembly Language

Moving data within the processor

MOV *destination*, *source* ; copy source to destination

```
MOV   R0, R1   ; copy R1 to R0
MOVS R0, R1   ; copy R1 to R0, update APSR flags

MOV    R7, #1  ; copy the value 1 to R7
```

ARMv7-M Assembly Language

Moving data to/from memory

```
LDR  register, memory        ; load register contents from memory
STR  register, memory        ; store register contents to memory

LDR    R0, [R1]     ; copy memory contents to R0, address is given by R1
STR    R0, [R1]     ; copy R0 to memory, address is given by R1

LDR    R0, [R1, #4]   ; address is R1+4 (pre-index), R1 is not updated
LDR    R0, [R1, #4]!  ; address is R1+4 (pre-index), R1 is updated to R1+4 (write back)
LDR    R0, [R1], #4   ; address is R1, then R1 is updated to R1+4 after memory access (post-index)
```

ARMv7-M Assembly Language

Stack operations

```
PUSH   {register list}          ; push the list of registers to memory using SP, SP is updated
POP    {register list}          ; pop the list of registers from memory using SP, SP is updated

PUSH   {R0-R4,R12}              ; push the given registers to memory using SP, SP is updated
POP    {R0-R4,R12}              ; pop the given registers from memory using SP, SP is updated

STMFD SP!,  {R0-R4,R12}         ; equivalent to the above PUSH
LDMFD SP!, {R0-R4,R12}          ; equivalent to the above POP
```

- **Full Descending** stack convention is the ARM default.
- Registers are stored such that higher register numbers are placed in higher memory addresses

STM and LDM instructions for stack operations

- FD = Full Descending
  - STMFD/LDMFD = STMDB/LDMIA (DB = decrement before, IA = increment after)
- ED = Empty Descending
  - STMED/LDMED = STMDA/LDMIB
- FA = Full Ascending
  - STMFA/LDMFA = STMIB/LDMDA
- EA = Empty Ascending
  - STMEA/LDMEA = STMIA/LDMDB
- In practice anything other than full descending convention is rare!

Cortex-M4 Assembly Language

Arithmetic operations (add, subtract, multiply, divide)

ADD *destination*, *operand1*, *operand2*

```
ADD     R0, R0, R1    ; R0 = R0 + R1
ADD     R0, R1, #25   ; R0 = R1 + 25
ADDS    R0, R1, #25   ; R0 = R1 + 25, then update APSR flags
```

Cortex-M4 Assembly Language

Branch instructions

B: simple branch

- B MyLabel

BL: branch and link for procedure calls, return address is stored in LR

- BL MyProcedure

BX: branch and exchange updates PSR, used for return from procedure/ISR

- BX LR

Conditional Mnemonics

Mnemonic extension	Meaning	Condition flag state
EQ	Equal	Z set
NE	Not equal	Z clear
CS/HS	Carry set/unsigned higher or same	C set
CC/LO	Carry clear/unsigned lower	C clear
MI	Minus/negative	N set
PL	Plus/positive or zero	N clear
VS	Overflow	V set
VC	No overflow	V clear
HI	Unsigned higher	C set and Z clear
LS	Unsigned lower or same	C clear or Z set
GE	Signed greater than or equal	N set and V set, or N clear and V clear (N == V)
LT	Signed less than	N set and V clear, or N clear and V set (N != V)
GT	Signed greater than	Z clear, and either N set and V set, or N clear and V clear (Z == 0,N == V)
LE	Signed less than or equal	Z set, or N set and V clear, or N clear and V set (Z == 1 or N != V)
AL	Always (unconditional)	-

Exception semantics

- On entry to an exception the hardware is in Thumb state, Handler Mode, privileged level.
- The following 8 registers are stacked using SP:
- **R0-R3, R12, LR, PC, PSR**
- Lazy stacking: if FPU is active, the FPU registers are also stacked
- Return from exception is done by
  - BX EXC_RETURN
  - EXC_RETURN is the value loaded into LR on entry to the exception

Exception semantics

EXC_RETURN[31:0]	Description
0xFFFFFFF1	Return to Handler mode, exception return uses non-floating-point state from the MSP and execution uses MSP after return.
0xFFFFFFF9	Return to Thread mode, exception return uses non-floating-point state from MSP and execution uses MSP after return.
0xFFFFFFFD	Return to Thread mode, exception return uses non-floating-point state from the PSP and execution uses PSP after return.
0xFFFFFFE1	Return to Handler mode, exception return uses floating-point-state from MSP and execution uses MSP after return.
0xFFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.

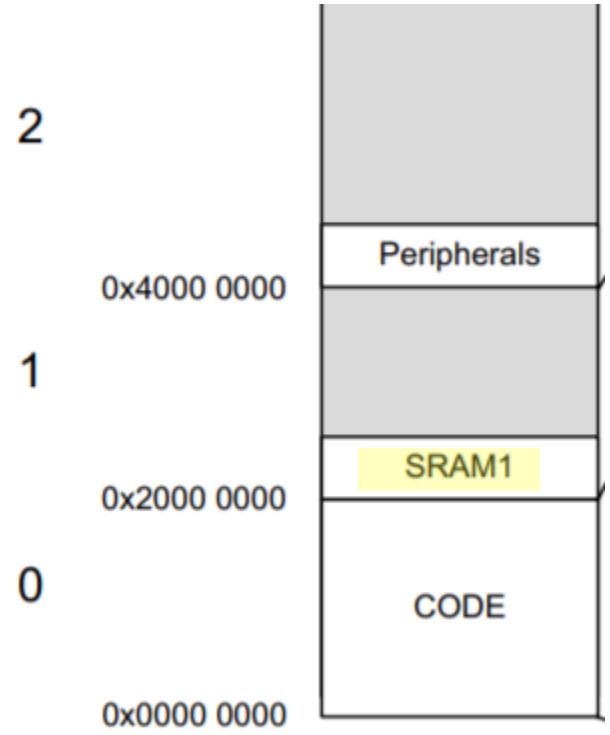
ARM procedure call standard

Register	Synonym	Special	Role in the procedure call standard
r15		PC	The Program Counter.
r14		LR	The Link Register.
r13		SP	The Stack Pointer.
r12		IP	The Intra-Procedure-call scratch register.
r11	v8		Variable-register 8.
r10	v7		Variable-register 7.
r9		v6 SB TR	Platform register. The meaning of this register is defined by the platform standard.
r8	v5		Variable-register 5.
r7	v4		Variable register 4.
r6	v3		Variable register 3.
r5	v2		Variable register 2.
r4	v1		Variable register 1.
r3	a4		Argument / scratch register 4.
r2	a3		Argument / scratch register 3.
r1	a2		Argument / result / scratch register 2.
r0	a1		Argument / result / scratch register 1.

Review Cortex-M4 exception handling in uDebugger homework

- Download uDebugger.zip
- Unzip
- Load the workspace into EWARM
- Start up Tera Term with baud rate 38400
- Run the project – notice the “Fail” message in serial terminal
- Refer to the following memory map to find an invalid address to substitute in GenerateFault() in main.c
- Explore the exception handler HardFaultIrqHandler in startup.s
- Your assignment for next week is to fill in missing code as specified in the assignment instructions.

Memory Map (detail)



SRAM1: 96 Kbytes located at address 0x2000 0000

Source: https://www.st.com/resource/en/datasheet/stm32l475vg.pdf