UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

The ARM[v7-M] Architecture

HISTORY, CPU MODES, INSTRUCTION SETS, EABI, & MORE

The ARM Way

- Designed and sold by ARM Limited
  - Fabless company
  - Licenses cores to other parties
    - AMD, Apple, NXP, Qualcomm, Samsung, TI
- Different architecture versions
  - ARMv7 comes in three profiles
    - A: Applications processors
    - R: Real time processors
    - M: Embedded processors
  - ARMv8 is latest architecture
    - Incorporates TrustZone into the M profile
- Cores can be licensed alongside other ARM IP

ARMv7 Architecture

- Defines 32 bit RISC CPU
- 16 integer registers
- Two instruction sets
  - ARM (Aarch32):
    - 32 bit instructions
    - full access to register file
  - Thumb-2 (T32)
    - 16 bit instructions
    - Most instructions operate on half the register set

The ARMv7-M Profile

- Supports only Thumb-2 instructions
- Optional Floating Point Unit (FPU)
- Optional Memory Protection Unit
  - This is not a Memory Management Unit

The ARMv7-M Profile

- Two Operating States
  - **Privileged** state
    - Default state of CPU at reset
    - Often used by RTOS kernels
    - CPU switches to this state when handling exceptions
    - All instructions can be executed
    - All memory regions can be accessed (unless disallowed by MPU)
  - **Non-privileged** state
    - Often used for RTOS tasks
    - Some instructions not available
    - Some memory regions inaccessible

The ARMv7-M Profile

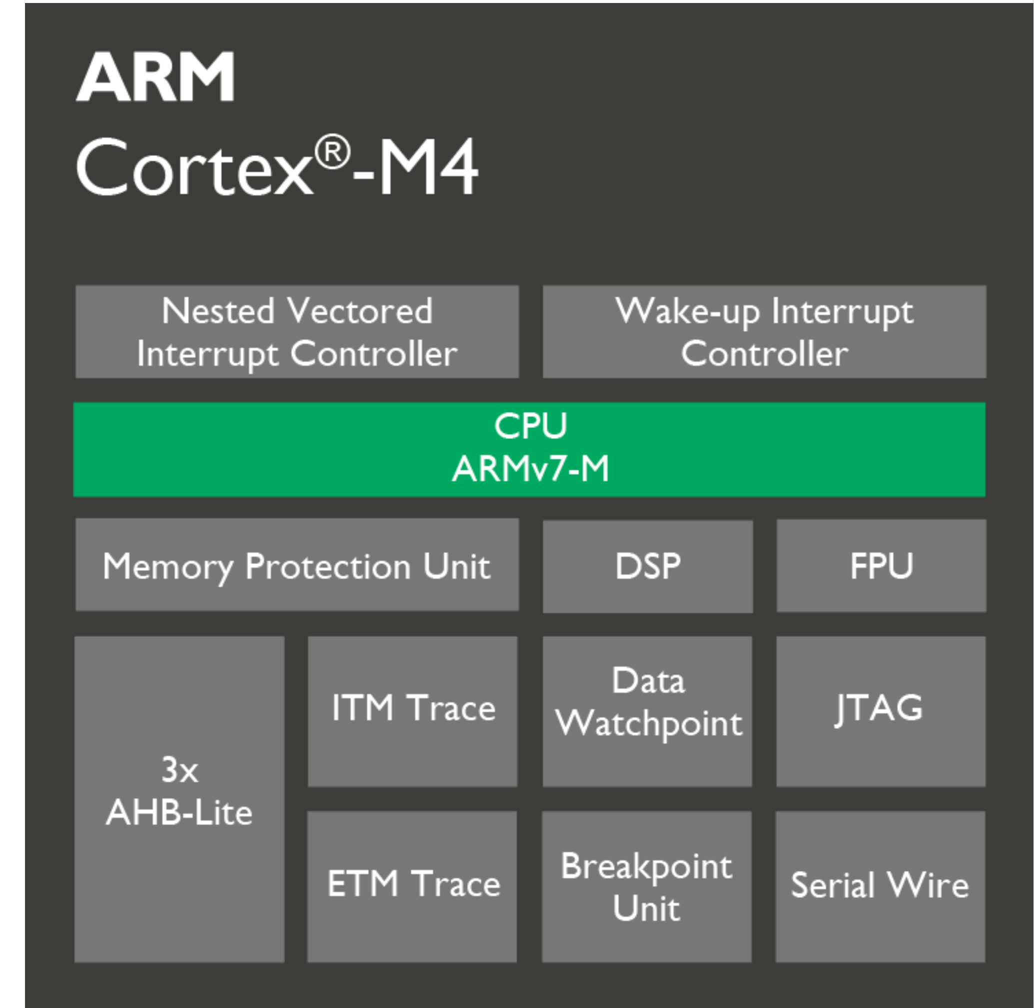
- Two Operating Modes
  - **Thread** mode
    - Standard operating mode
    - Entered upon reset or at exception return
    - CPU can use *process stack* or *main stack*
  - **Handler** mode
    - Entered as a result of an exception
    - Code runs in privileged environment
    - CPU uses *main stack*

ARMv7-M States

- Obtain state by reading `control` register
  - Operation is permissible only in privileged state
- Switch to non-privileged mode
  - Write to control register
  - Can only be done from privileged software
- Switch to privileged mode
  - Must be done through service request

The Cortex-M4[F] Core

- F = have floating point unit
- Implements ARMv7-M
- Used in the MSP432 Microcontroller (and many others)
- 3 Stage RISC Pipeline
- Branch Prediction
- 12 cycle interrupt latency
  - Very important for real time systems



The Cortex-M4 Memory Model

- Memory-mapped architecture
- Single, flat address space
- Instruction fetches are always half-word aligned (16 bit)
- All addressed in ARMv7-M are physical addresses
  - No MMU Fairy to help you with your out of bounds accesses
- Unaligned accesses may trigger a fault
- The ARM Architecture is bi-endian
  - Cortex-M4 in your board is configured as little-endian

The Cortex-M4 Memory Model

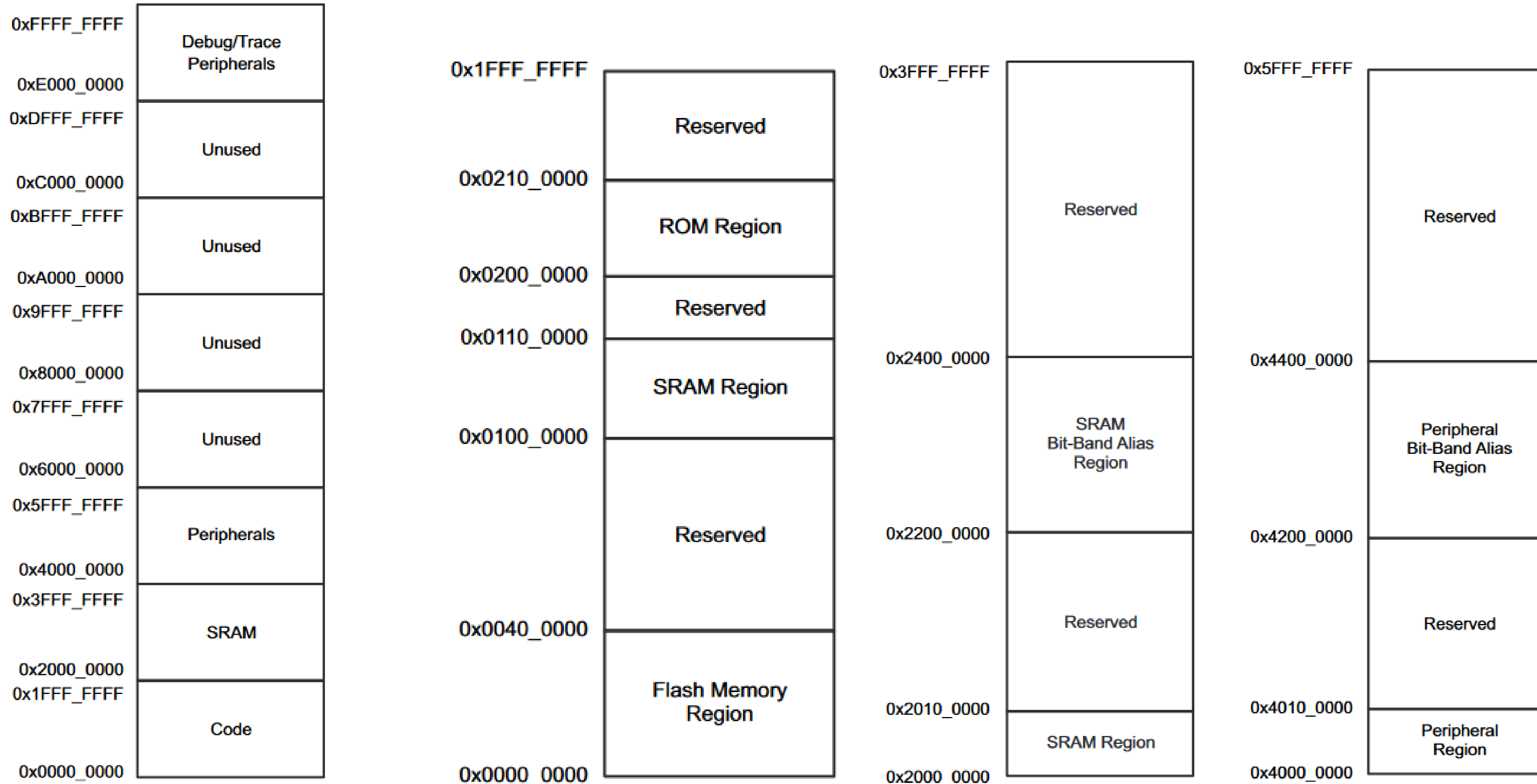
Address	Name	Type	NX?	Cache	Description
0x00000000-0x1fffffff	Code	Normal Memory	-	Write Through	Typically ROM or flash memory
0x20000000-0x3fffffff	SRAM	Normal Memory	-	Write Back, Write Allocate	SRAM region typically used for on-chip RAM
0x40000000-0x5fffffff	Peripheral	Device	NX	-	On-chip peripheral address space
0x60000000-0x7fffffff	RAM	Normal memory	-	Write Back, Write Allocate	Memory with write-back, write allocate cache attribute for L2/L3 cache support
0x80000000-0x9fffffff	RAM	Normal memory	-	Write Through	Memory with write-through cache attribute
0xa0000000-0xbfffffff	Device	Device, shareable	NX	-	Shared device space
0xc0000000-0xdfffffff	Device	Device, non-shareable	NX	-	Non-shared device space
0xe0000000-0xffffffff	System		NX	-	System segment for PPB and vendor peripherals

Regions may not be used to its fullest.

UF Nelms Institute for the Connected World UNIVERSITY of FLORIDA

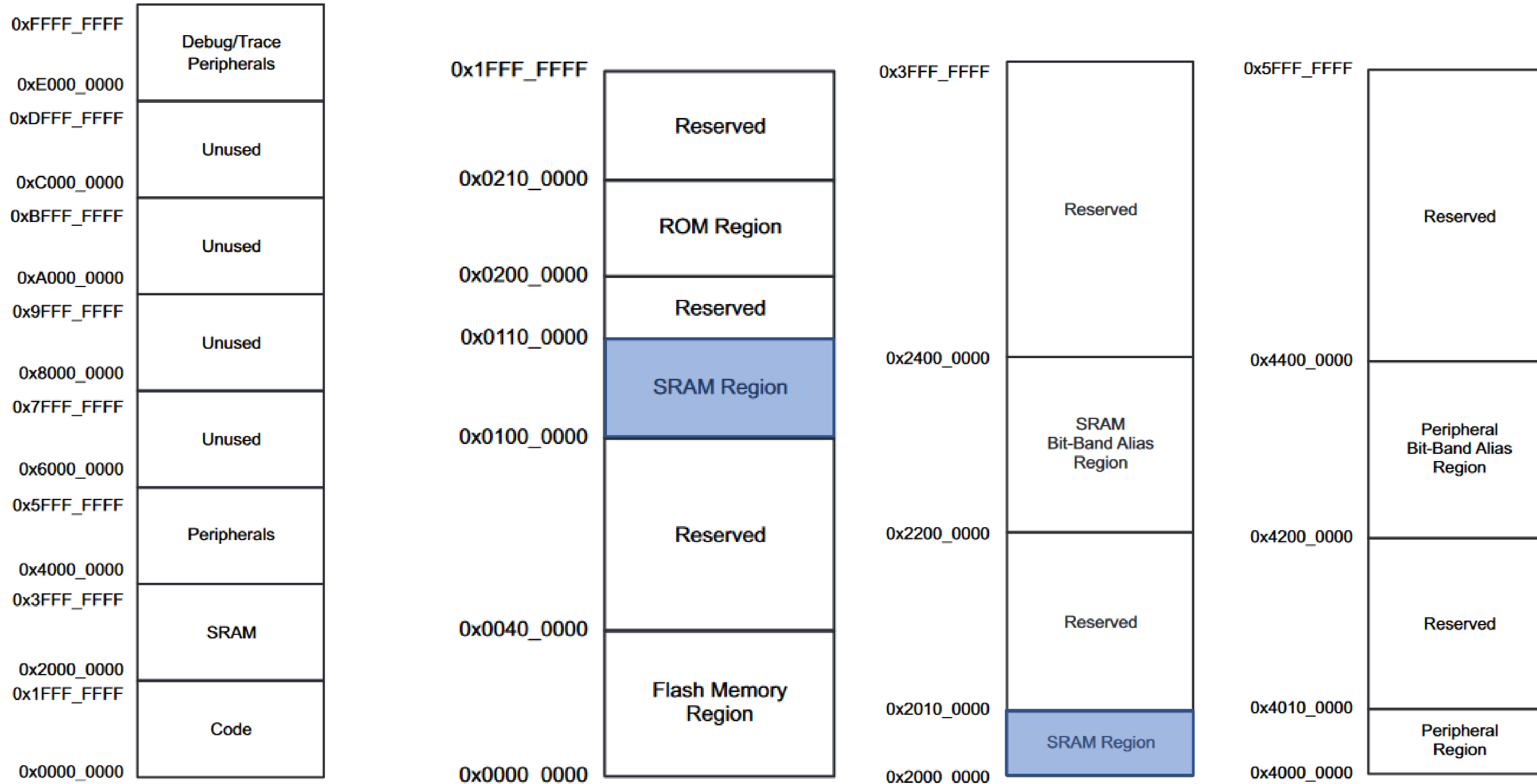
The MSP432 Memory Map

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA



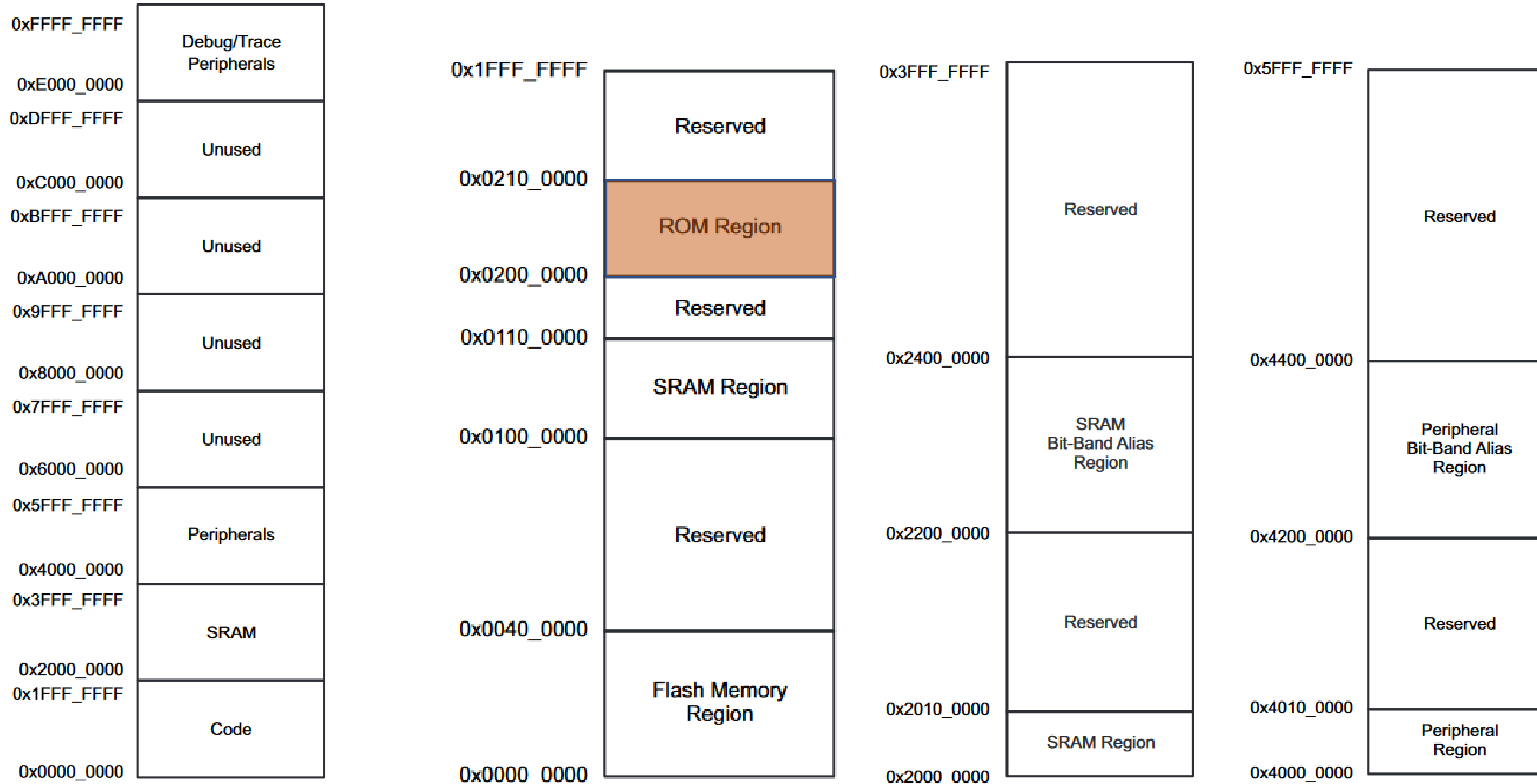
The MSP432 Memory Map

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA



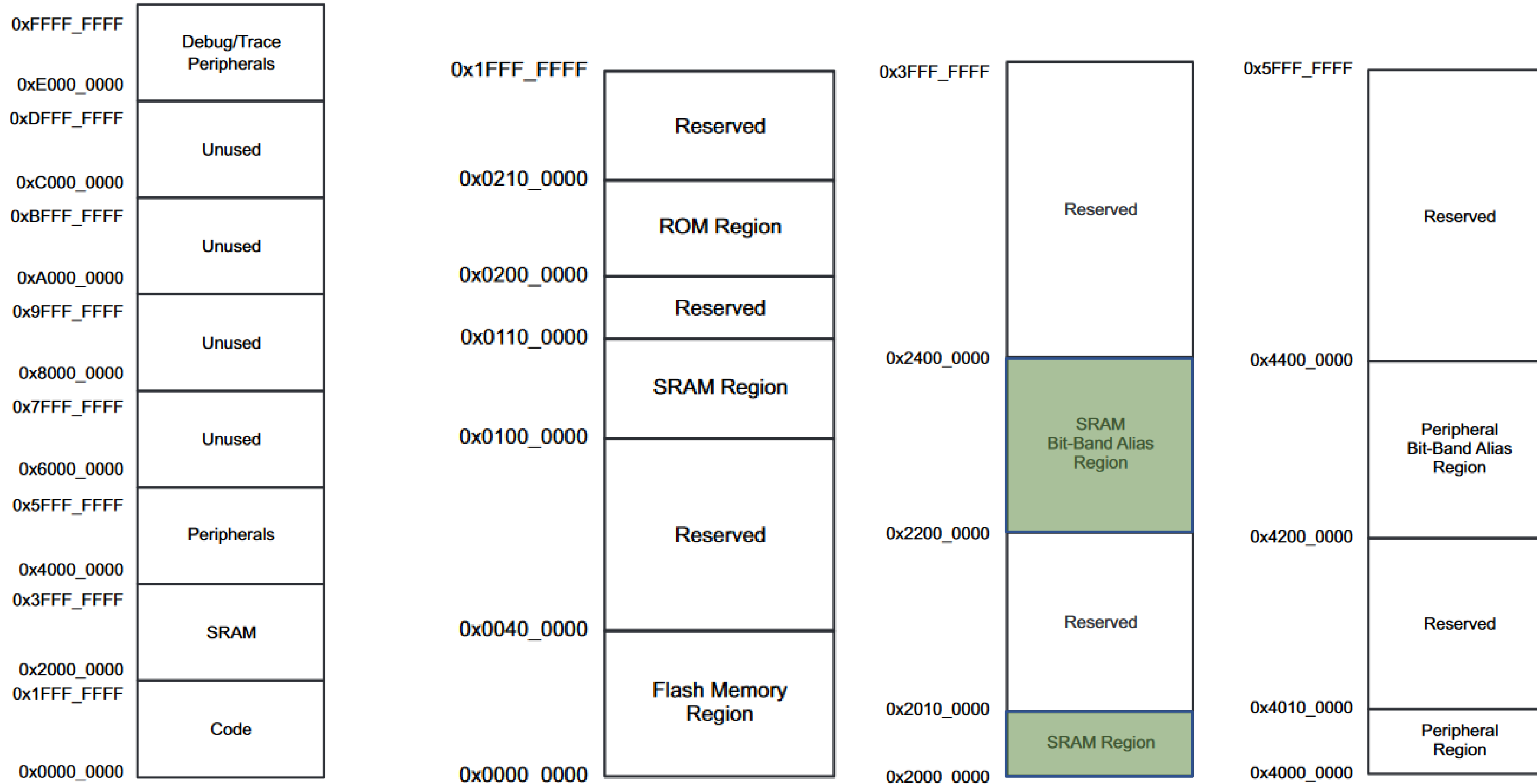
The MSP432 Memory Map

Nelms Institute for the Connected World
UNIVERSITY of FLORIDA



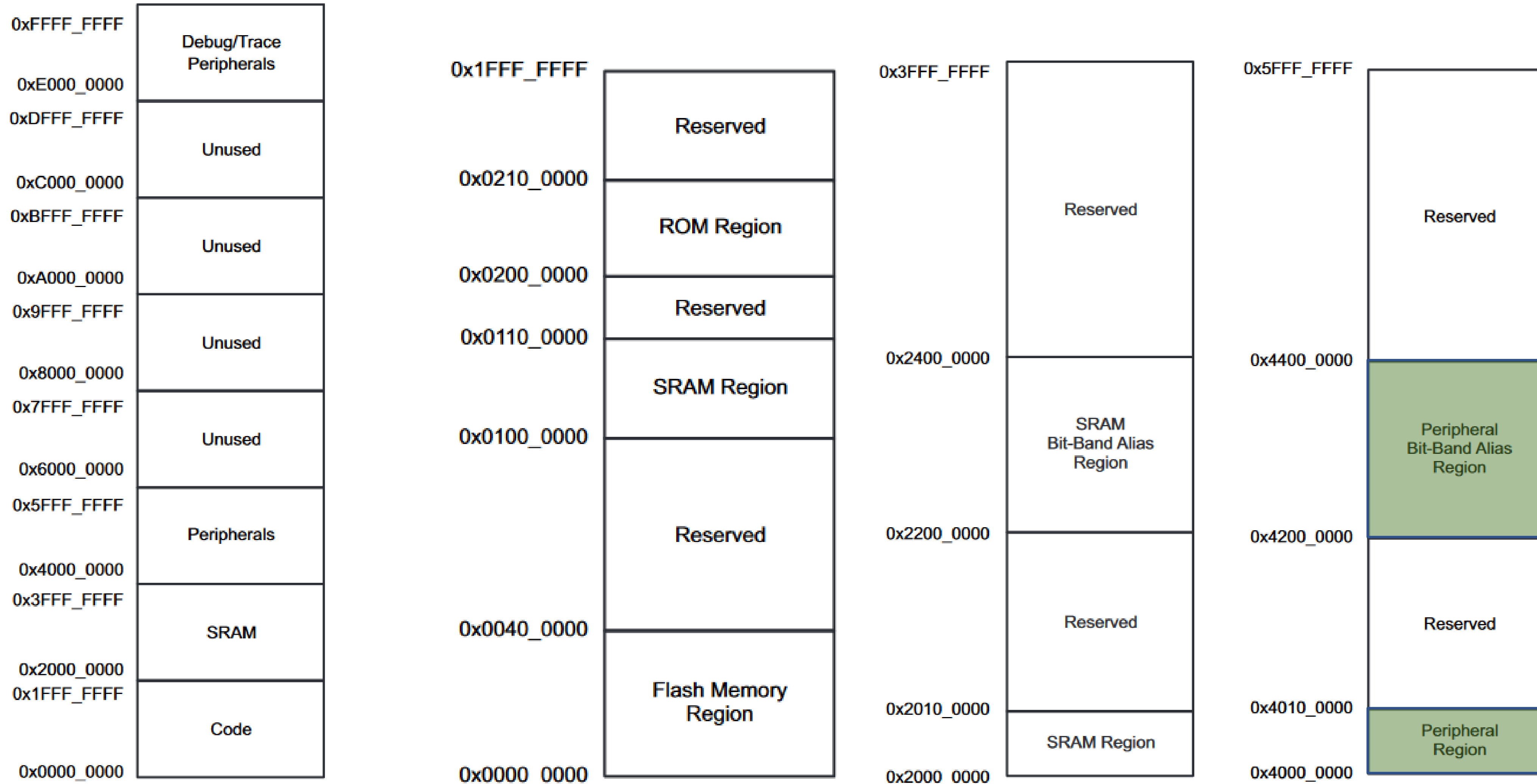
The MSP432 Memory Map

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA



The MSP432 Memory Map

Nelms Institute for the Connected World
UNIVERSITY of FLORIDA



More on Bit-band Aliasing

Each bit in SRAM is aliased to a word in the SRAM bit-band alias area

Address	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0x20000000	0	0	0	0	0	1	1	0

Address	Value
0x2200001c	00000000
0x22000018	00000000
0x22000014	00000000
0x22000010	00000000
0x2200000c	00000000
0x22000008	00000001
0x22000004	00000001
0x22000000	00000000

More on Bit-band Aliasing

Each bit in SRAM is aliased to a word in the SRAM bit-band alias area

Address	7	6	5	4	3	2	1	0
0x20000000	0	0	0	0	0	1	1	0

Address	Value
0x2200001c	00000000
0x22000018	00000000
0x22000014	00000000
0x22000010	00000000
0x2200000c	00000000
0x22000008	00000001
0x22000004	00000001
0x22000000	00000000

More on Bit-band Aliasing

Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Each bit in SRAM is aliased to a word in the SRAM bit-band alias area

Address	Bits							
0x20000000	0	0	0	0	0	1	1	0

Address	Value
0x2200001c	00000000
0x22000018	00000000
0x22000014	00000000
0x22000010	00000000
0x2200000c	00000000
0x22000008	00000001
0x22000004	00000001
0x22000000	00000000

More on Bit-band Aliasing

Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Each bit in SRAM is aliased to a word in the SRAM bit-band alias area

Address	Bits
0x20000000	0 0 0 0 0 1 1 0

Address	Value
0x2200001c	00000000
0x22000018	00000000
0x22000014	00000000
0x22000010	00000000
0x2200000c	00000000
0x22000008	00000001
0x22000004	00000001
0x22000000	00000000

More on Bit-band Aliasing

Each bit in SRAM is aliased to a word in the SRAM bit-band alias area

0x20000000

0	0	0	0	0	1	1	0

Address	Value
0x2200001c	00000000
0x22000018	00000000
0x22000014	00000000
0x22000010	00000000
0x2200000c	00000000
0x22000008	00000001
0x22000004	00000001
0x22000000	00000000

More on Bit-band Aliasing

Each bit in SRAM is aliased to a word in the SRAM bit-band alias area

0x20000000

0	0	1	0	0	1	1	0

Address	Value
0x2200001c	00000000
0x22000018	00000000
0x22000014	00000001
0x22000010	00000000
0x2200000c	00000000
0x22000008	00000001
0x22000004	00000001
0x22000000	00000000

More on Bit-band Aliasing

Each bit in SRAM is aliased to a word in the SRAM bit-band alias area

Address	Value
0x2200001c	00000000
0x22000018	00000000
0x22000014	00000001
0x22000010	00000000
0x2200000c	00000000
0x22000008	00000001
0x22000004	00000001
0x22000000	00000001

Address								
0x20000000	0	0	1	0	0	1	1	1

```
ldr r0, =0x20000000
ldrb r1, [r0]        /* read */
orr r1, r1, #1       /* modify */
strb r1, [r0]        /* write */
```

More on Bit-band Aliasing

Each bit in SRAM is aliased to a word in the SRAM bit-band alias area

0x20000000	0	0	1	0	0	0	1	1

Address	Value
0x2200001c	00000000
0x22000018	00000000
0x22000014	00000001
0x22000010	00000000
0x2200000c	00000000
0x22000008	00000000
0x22000004	00000001
0x22000000	00000001

```
ldr r0, =0x22000008
eor r1, r1, r1
str r1, [r0]          /* write */
```

ARMv7-M Registers and EABI

- EABI: Embedded Application Binary Interface
  - Set of rules for register usage
  - Important when combining C and Assembly code

Register	Name	Usage
`ip` (`r12`)	Intra-procedure-call scratch register	Usage depends on tools and environment.
`sp` (`r13`)	Stack pointer	Points to bottom of stack • `msp`: used when CPU is in privileged state • `psp`: used when CPU is in non-privileged state
`lr` (`r14`)	Link Register	Points to subroutine return address
`pc` (`r15`)	Program Counter	Points to next instruction

ARMv7-M Registers and EABI

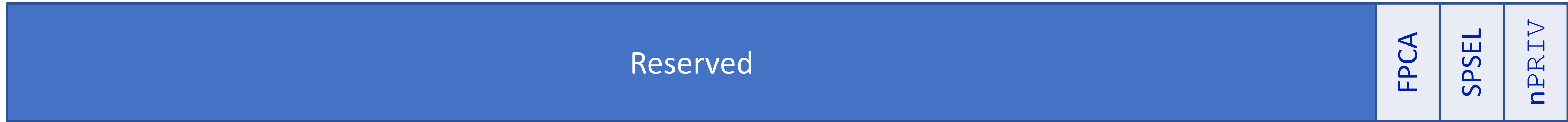
UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Register	Usage
r0	First function argument, function return, scratch register 1
r1	Second function argument, function return, scratch register 2
r2	Third function argument, scratch register 3
r3	Fourth function argument, scratch register 4
r4	Variable register 1
r5	Variable register 2
r6	Variable register 3
r7	Variable register 4
r8	Variable register 5
r9	Platform register. Usage depends on platform.
r10	Variable register 7
r11	Variable register 8

ARMv7-M Registers and EABI

- Variable registers (`r4—r11`) are callee saved
  - Function being called is responsible for saving them
- Scratch registers (`r0—r3`) are caller saved
  - Function doing the call is responsible for saving them
- Banked registers (`msp`/`psp`) must be accessed by special instructions
- Most Thumb2 instructions can only operate on lower half of register file (`r0` – `r7`)

The control register



Bit	Description	
nPRIV	Defines execution privilege in thread mode	
	0	Thread mode has privileged access
	1	Thread mode has unprivileged access
SPSEL	Defines the stack pointer to be used	
	0	Use msp as the current stack pointer (sp has the value of msp)
	1	Use psp as the current stack pointer (sp has the value of psp)
	Value of 1 is reserved in Handler mode.	
FPCA	Defines whether the Floating Point extension is active in the current context	
	0	FP extension is not active
	1	FP extension is active

UF Nelms Institute for the Connected World UNIVERSITY of FLORIDA

The `xPSR` Program Status Register

- Composed of three subregisters
  - Application Program Status Register (`apsr`)
    - Holds flags
    - Can be written by unprivileged software
  - Interrupt Program Status Register (`ipsr`)
    - Used when CPU is in handler mode
    - Holds exception number, otherwise value is 0
  - Execution Program Status Register (`epsr`)
    - Holds execution state bits

The xPSR Program Status Register

Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Bit	Description
N	Negative Flag
Z	Zero Flag
C	Carry Flag
V	Overflow Flag
Q	DSP overflow and Saturation Flag
GE	Greater than or Equals Flag

The xPSR Program Status Register

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Bit	Description
ICI	Interruptible-continuable instruction
IT	Execution state bits of the it instruction
T	Thumb state bit (must be preserved as 1)

More on `ICI` bits

- When interrupt occurs during execution of `ldm`, `stm`, `push`, `pop`, `vldm`, `vstm`, `vpush`, or `vpop` the CPU will
  - Temporarily halt the load/store multiple operation
  - Stores the next register operand in the multiple operation to `ICI`
- After servicing the interrupt the CPU will
  - Return to the register pointed to by `ICI`
  - Resumes execution of the load/store multiple instruction

UF | Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

More on `IT` bits

- Used with the `it` instruction
- Four instructions per block
- Each instruction can have the same condition
- Or have inverse condition of each other
- `IT` functionality supersedes `ICI`
  - If interrupt occurs in an `it` block over an interruptible-continuable instruction, the instruction will be restarted from scratch after the interrupt is serviced

ARM CMSIS

- The Cortex Microcontroller Software Interface Standard is a set of *platform-independent* API that allows for easy access to a Cortex MCU and its peripherals.
- CMSIS-Core: Startup and peripheral access
- CMSIS-RTOS: Generic RTOS interfaces
- CMSIS-DSP: DSP functions for Cortex processors
- CMSIS-DAP: Standard interface to the Cortex Debug Access Port

ARM CMSIS vs TI

- At its time of release, MSP432 was not CMSIS compliant
  - TI had their own set of drivers based on their MSP430's driverlib
- Today TI provides support for both CMSIS and driverlib
  - Use CMSIS to avoid platform lockdown!

ARM CMSIS-Core

File/Directory	Contents
CMSIS/Documentation/Core	All of CMSIS Core documentation
CMSIS/Core/Include	Header files for CMSIS-Core • `core_cm4.h` for Cortex-M4 devices
Devices	Reference implementation for Cortex-M devices
Devices/_Template_Vendor	CMSIS-Core Device Templates for extension by vendors

Board Support Packages

- Additional libraries provided by vendor
- Contains functions and extra drivers for peripherals
- Contains toolchain support files
  - Platform-specific linker scripts
  - Platform-specific header files
  - Platform-specific device trees
- BSPs are usually licensed so that they can be readily linked in by software and redistributed as part of it without restriction

Thumb-2 Instruction Set

- Reduces most ARM instructions from 32 bit to 16 bit
- Reduced number of bits to encode registers
- Reduced number of bits for immediate constants
- Remove conditional field from instructions (uses `it` instead)
- Remove complex instructions

Chapter A4 of the ARMv7-M Architecture Reference Manual contains the reference for the Thumb-2 Instruction Set.

Unified Assembler Language

- Common syntax for ARM and Thumb instructions

```
.syntax unified
```

- Instructions take three operands
  - Two registers and a literal
  - Three registers
- Instructions can be postfixed with `s`
  - Updates flags in `apsr`

```
eor r0, r0, r0     /* 0 -> r0 */
eors r0, r0, r0    /* 0 -> r0, apsr_z = 1 */
```

Addressing Mode: Offset

```
ldr r0, [r1, #4]
str r2, [r3, #-4]
```

- Offset is added or subtracted from base register
- Result is used as effective address for memory access

```
mem[r1 + 4] -> r0
r2 -> mem[r3 - 4]
```

Addressing Mode: Pre-indexed

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

```
ldr r0, [r1, #4]!
str r2, [r3, #-4]!
```

- Offset is applied to base register
- Result is used as effective address for memory access
- Result is written back to base register

```
mem[r1 + 4] -> r0; r1 + 4 -> r1
r2 -> mem[r3 - 4]; r3 - 4 -> r3
```

Addressing Mode: Pre-indexed

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

```
ldr r0, [r1], #4
str r2, [r3], #-4
```

- Address from base register is used as effective memory address
- Offset is applied to base register then written back

```
mem[r1] -> r0; r1 + 4 -> r1
r2 -> mem[r3]; r3 - 4 -> r3
```

- Operation is equivalent to a load, then add
- Machine encoding is smaller with pre-indexed operation

Addressing Mode: Immediate

```
add r0, r0, #256
orr r0, r0, #(1 << 8)
```

- Immediate constant is used as an operand

```
r0 + 256 -> r0
r0 | 256 -> r0
```

- Instruction may be 32 bits wide

Addressing Mode: Register


```
add r0, r0, r1
ldr r0, [r1, r2]
```

- Contents of register is used as an operand

```
r0 + r1 -> r0
mem[r1 + r2] -> r0
```

Addressing Mode: Shifted Index

```
add r0, r0, r1, lsl #3
add r0, r0, r1, lsr #3
add r0, r0, r1, asr #3
```

- Contents of register is shifted, then used
  - `lsl`: Logical shift left
  - `lsr`: Logical shift right (pad left with 0)
  - `asr`: Arithmetic shift right (pad left with sign bit)

```
r0 + 2^3*r1 -> r0
r0 + unsigned(r1)/2^3 -> r0
r0 + signed(r1)/2^3 -> r0
```

The If-Then (`it`) Instruction

- Substitute for conditional execution in A and R profiles

```
it{x{y{z}}} condition
```

- `x`, `y`, `z` parameters express execution clause
  - `t`: then
  - `e`: else
- `condition`: any condition on the set
  - `lt`, `gt`, `eq`, `ne`, `ge`, `le`
  - If condition not met instruction becomes a `nop`

The If-Then (`it`) Instruction

```c
if(r0 == 0) {
      r1 = r1 + 1;
} else {
      r1 = r1 - 1;
}
```

Can be compiled to

```
tst r0, 0          /* r0 & 0, result discarded, flags set */
ite eq             /* if equal (zero flag set), then */
addeq r1, r1, #1   /* perform addition */
subne r1, r1, #1   /* else, perform subtraction */
```

The If-Then (`it`) Instruction

- C Language Runtime Requirements
  - Variables with permanent store that are *explicitly* initialized go in `.data`
  - Variables with permanent store that are not *implicitly* initialized to 0, placed in `.bss`
- The problem
  - Embedded devices have no program loader
  - Must initialize own data sections
- Program must
  - Copy `.data` section from ROM/Flash to RAM
  - Clear `.bss` section

The If-Then (it) Instruction

Symbol	Use
__data_start	Address of .data section in RAM
__data_end	Address of end of .data section in RAM
__data_rom_start	Address of initialization data in ROM/Flash
__bss_start	Address of .bss section in RAM
__bss_end	Address of end of .bss section in RAM

Symbols are provided by linker script, names may change accordingly.

The If-Then (it) Instruction

```
__do_copy_data:
        ldr r4, =__data_start           /* load start of .data section */
        ldr r5, =__data_end             /* load end of .data section */
        ldr r6, =__data_rom_start       /* load start of initialization */
1b:     cmp r4, r5                      /* check to see if we are done */
        ittt lt                         /* if not done */
        ldrlt r7, [r6], #4              /* load word from init data */
        strlt r7, [r4], #4              /* and place it in .data, and */
        blt 1b                          /* continue in loop until done */
```


UF Nelms Institute for the Connected World UNIVERSITY of FLORIDA

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

The If-Then (it) Instruction

```
__do_clear_bss:
        ldr r4, =__bss_start            /* load start of .bss section */
        ldr r5, =__bss_end              /* load end of .bss section */
        eor r6, r6, r6                  /* clear register r6 */
1b:     cmp r4, r5                      /* check to see if we are done */
        itt lt                          /* if not done */
        strlt r6, [r4], #4              /* clear word in .bss, and */
        blt 1b                          /* continue in loop until done */
```

The If-Then (`it`) Instruction

- Not allowed in an IT block
  - `cbz`, `cbnz`, `tbb`, `tbh`, `cps`, `cpsid`, `cpsie`, `setend`
  - Assembler directives
- Branch or other instruction that modifies `pc` are only allowed at end of `IT` block
- Can not branch to any instruction inside `IT` block unless returning from exception handler

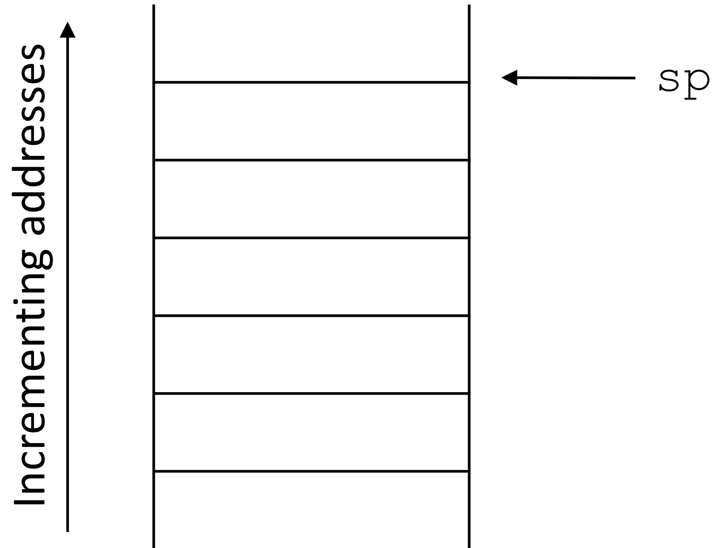
Register Lists

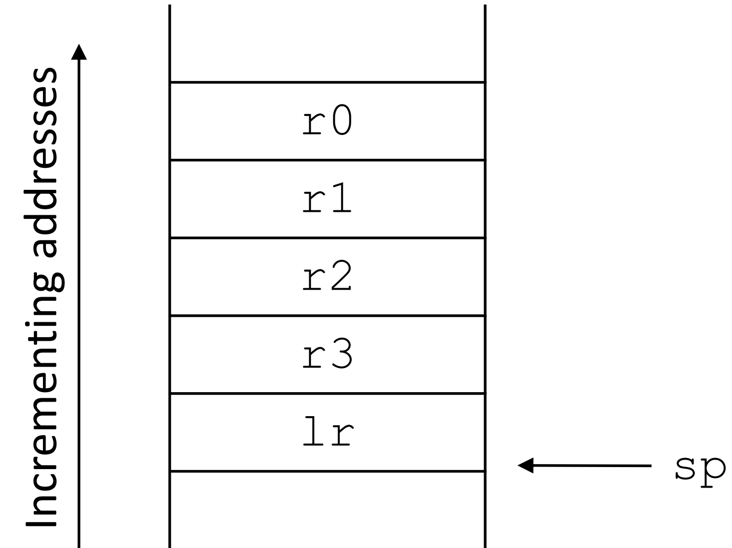
- Loading and storing multiple registers simultaneously
- Register range must be written in ascending order

```
push {r0, r1, r2, r3, lr}
pop {r0-r3, pc}
```

- Pushes `r0` to `r3`, and `lr` into stack, with r0 at highest address; decrements `sp`
- Pops `pc`, and `r3` to `r0` from the stack with the value at the lowest address going into `pc`; increments `sp`

Register Lists






```
push {r0, r1, r2, r3, lr}
stmdb sp!, {r0, r1, r2, r3, lr}
```

Register Lists

```
stmdb r0!, {r4, r5, r6, r7, r8, r9, r10, r11}
ldmia r0!, {r4-r11}
```

- **S**tore **m**ultiple, **d**ecrement **b**efore
  - Save register list starting at `r0` minus eight words (32 bytes)
  - Write back into `r0` new value of `r0` (`r0 – 32 -> r0`)
  - `r4` stored at highest address
- **L**oa**d** **m**ultiple, **i**ncrement **a**fter
  - Load register list starting at `r0`
  - Value at lowest address is placed into `r11`
  - Increment `r0` by eight words (32 bytes)
  - Save new value of `r0` (`r0 + 32 -> r0`)

Special Purpose Registers

- Registers that are not part of the integer (or floating point) register file
  - `control`
  - `psp`
  - `msp`
  - `xPSR`
- Can not be manipulated with conventional load/store or arithmetic/logic instructions
  - `mrs`: copy special purpose register into integer register
  - `msr`: copy integer register into special purpose register

Special Purpose Registers

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

- Dropping privileges and switching stacks

```
__exit_execve:
        ldr r4, task_stack_ptr         /* load address of task stack */
        ldr r5, entry_point            /* load address of entry point */
        mov lr, r5                     /* save entry point into lr */
        msr psp, r4                    /* set process stack pointer */
        mrs r4, control                /* load control register */
        orr r4, r4, #3                 /* set nPRIV and SPSEL bits */
        msr control, r4                /* save new control register */
        eor r4, r4, r4                 /* clear r4 */
        eor r5, r5, r5                 /* clear r5 */
        bx lr                          /* and jump to task entry point */
```

The Floating Point Unit

- Optional component in ARMv7-M
- IEEE 754 Compliant
- Single precision only!
  - Supports only `float`
  - Any `double` will be emulated in software ☹
- Must be enabled before using
  - Otherwise an Usage Fault interrupt will be raised
- Compiler must be aware you want to use the FPU

The Floating Point Unit

- Enable FPU using ARM CMSIS-Core
  - Cortex Microcontroller Software Interface Standard
  - Vendor Independent Hardware Abstraction Layer
  - Included with microcontroller vendor's Board Support Package (BSP)

```
/* in core_cm4.h, simplified */
#ifdef __FPU_PRESENT    /* __FPU_PRESENT defined if there is an FPU */
#define __FPU_USED      /* define this macro if the FPU is used */
#endif
```

- The `SystemInit()` function will enable the FPU

The Floating Point Unit

- Enable the FPU using only C
  - Enable coprocessors `CP11` and `CP10` in the Coprocessor Access Control Register (`cpacr`)
  - Memory mapped register at address `0xe000ed88`

```
#define cpacr (*((volatile unsigned int*)(0xe000ed88)))

void enable_fpu(void) {
        cpacr |= (0xf << 20);
        __dsb();
        __isb();
}
```

The Floating Point Unit

- Enable the FPU using only Assembly
  - Coprocessors CP11 and CP10 must be enabled in cpacr

```
enable_fpu:
      ldr r0, =0xe000ed88
      ldr r1, [r0]
      orr r1, r1, #(0xf << 20)
      str r1, [r0]
      dsb                           /* data synchronization barrier */
      isb                           /* instruction sync barrier */
      bx lr
```

The MSP432 and Board Components

PERIPHERALS, SERIAL BUSES, & LP3943

MSP432 Peripherals

UF Nelms Institute for the Connected World UNIVERSITY of FLORIDA

- Range of peripherals include
  - MSP430-compatible peripherals
  - MSP432-specific peripherals
- Supported by both TI's driverlib and CMSIS-Core
- Can be programmed directly in C or assembly without the need for libraries

MSP432 Peripherals

- General purpose I/O
- Port Mapping Controller
- Timer_A module (MSP430 compatible peripheral)
- Timer32 module (ARM-specific timer)
- Enhanced Universal Serial Communications Interface (eUSCI, MSP430 compatible)
- Real Time Clock module (RTC_C, MSP430 compatible)
- Watchdog Timer, ADC…

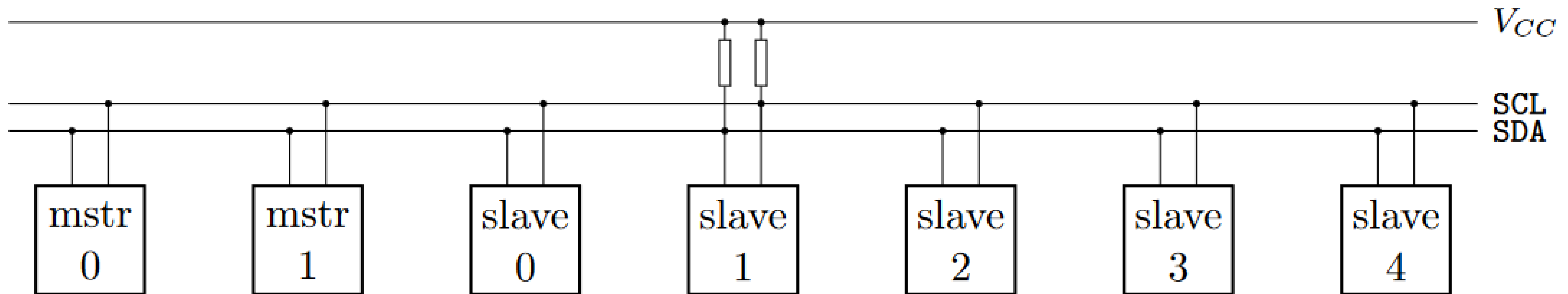
MSP432 Peripherals

- Enhanced Universal Serial Communication Interface (eUSCI)
  - Can handle UART, I2C, SPI, IrDA
- UART mode described in document number *slau423*
- I2C mode described in document number *slau425*

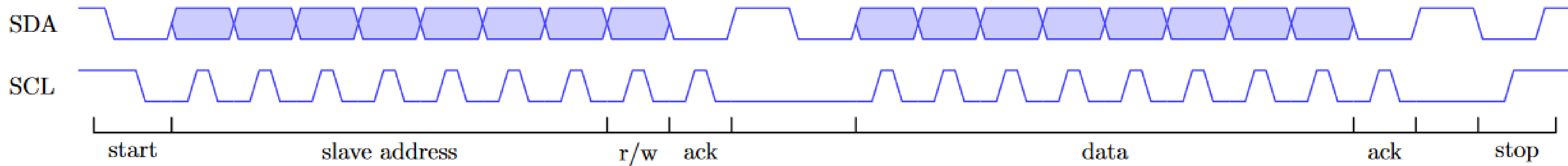
Inter-Integrated Circuit Bus

- Two Wire communication protocol
  - SDA: serial data
  - SCL: serial clock
- Multi-master, multi-slave system
- Master device always initiates transmission
- Slave device responds to requests from master
  - Slaves are identified using a 7 bit address

Inter-Integrated Circuit Bus



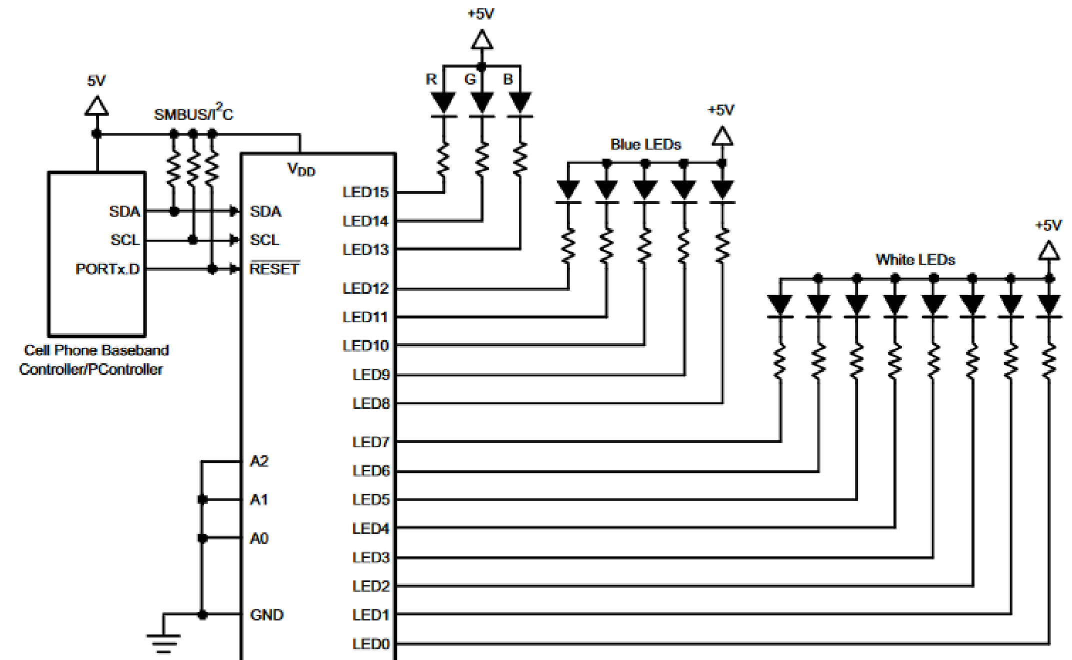
Inter-Integrated Circuit Bus



- Transmission is started by master asserting start condition
- Send 7 bit slave address and r/~w bit
- Slave sends ACK driving SDA low
- Communication is established, data communicated depends on device
  - Receiving endpoint must always send an ACK after 8 bits.
- Stop condition always generated by master

The TI LP3943 LED Driver

- Independently control 16 LEDs
- I2C interface to MCU
  - Continuous write allowed by setting bit 4 of register address to 1.
- Two Internal PWM controls



UF Nelms Institute for the Connected World UNIVERSITY of FLORIDA

The TI LP3943 LED Driver

- LED control on LS*n* registers

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0
B1	B0	B1	B0	B1	B0	B1	B0
LED3		LED2		LED1		LED0	

The LS0 Register (address 0x06)

B1	B0	Configuration
0	0	Output is high impedance (LED is off)
0	1	Output set to ON state
1	0	Use PWM0/PSC0 for waveform
1	1	Use PWM1/PSC1 for waveform

LED output modes

- Duty Cycle control in PWM*n*

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0

The PWM0 Register (address 0x03)

- Frequency control in PSC*n*

7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0

The PSC0 Register (address 0x02)

The TI LP3943 LED Driver

- PWMn and PSCn registers are integer registers
  - No floating point values allowed. Round to nearest.
- Driver can do 8 bit PWM, only 256 steps available
  - Increment in steps of 1/256
- Waveform period can range from 0.625 ms to 1.6 s
  - Compute the PSCn value for period T using $T = (PSCn + 1)/160$

The TI LP3943 LED Driver

- LEDs 0 to 7,1 Hz, 25%
- LEDs 8 to 12, 5 Hz, 25%
- LEDs 13 to 15 off

PSC0 (0x02)

7	6	5	4	3	2	1	0
1	0	0	1	1	1	1	1

PWM0 (0x03)

7	6	5	4	3	2	1	0
0	1	0	0	0	0	0	0

PSC1 (0x04)

7	6	5	4	3	2	1	0
0	0	0	1	1	1	1	1

PWM1 (0x05)

7	6	5	4	3	2	1	0
1	0	0	0	0	0	0	0

LS0 (0x06)

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0
LED3		LED2		LED1		LED0	

LS1 (0x07)

7	6	5	4	3	2	1	0
1	0	1	0	1	0	1	0
LED7		LED6		LED5		LED4	

LS2 (0x08)

7	6	5	4	3	2	1	0
1	1	1	1	1	1	1	1
LED11		LED10		LED9		LED8	

LS3 (0x09)

7	6	5	4	3	2	1	0
0	0	0	0	0	0	1	1
LED15		LED14		LED13		LED12	

The ARMv7-M Architecture

EXCEPTION MODEL, NVIC, SYSTICK, & PENDSV

Exception Model

- Exception == Interrupt
- Different in the M profile
- Hardware saves and restores key states on exception entry and exit
- Vectored exceptions
  - Each exception has its own unique vector
- Interrupts can be nested
  - Higher priority interrupts can preempt the servicing of lower priority interrupts
  - Lower priority interrupts must wait for higher priority interrupts to be serviced
  - Nest too deep and you will find yourself with a Stack Overflow

The Interrupt Vector Table

- Initial table located at `0x00000000`
- Can be relocated to any other address
  - `VTOR`: Vector Table Offset Register, initially 0
  - Can only be done by privileged software
- 16 core interrupts, up to 240 external vectors

16+*N*	`64+4*N`	External Interrupt *N*
15	`0x0000003c`	`SysTick`
14	`0x00000038`	`PendSV`
13	`0x00000034`	Reserved
12	`0x00000030`	`DebugMonitor`
11	`0x0000002c`	`SVCall`
10	`0x00000028`	
9	`0x00000024`	
8	`0x00000020`	Reserved
7	`0x0000001c`	
6	`0x00000018`	`UsageFault`
5	`0x00000014`	`BusFault`
4	`0x00000010`	`MemManage` (MPU exception)
3	`0x0000000c`	`HardFault`
2	`0x00000008`	`NMI` (Non-Maskable Interrupt)
1	`0x00000004`	`Reset`
0	`0x00000000`	Initial Stack Pointer

The Cortex-M4 Startup

- CPU enters handler mode
- CPU reads address `0x00000000`, stores value in `msp`
- CPU reads address `0x00000004`, stores value in `pc`
- CPU leaves handler mode, enters thread mode
- CPU starts instruction fetches from `pc`

Servicing Exceptions

- CPU finishes executing current instruction
  - If instruction is of interruptible-continuable kind, state is saved in `epsr.ici` and instruction is interrupted
  - If instruction is of interruptible-continuable kind and inside an it block, instruction state is discarded and instruction is interrupted
- CPU enters handler mode
- CPU saves current execution state into stack
- Exception number is stored in `ipsr`
- CPU reads address `VTOR + 4*ipsr`
- Register `lr` is loaded with a `EXC_RETURN`
- CPU stores read value in `pc`, resumes execution from new `pc`

Saved Context

- Context saved into stack
  - Stack pointer used depends on CPU mode at time of exception
- Resulting stack pointer always aligned to a double word

Element	Description
xPSR	Processor status register at time of exception
ReturnAddress	Next pc at time of exception. Current pc if instruction is interruptible and was interrupted.
lr	Procedure link register (r14)
r12	Intra-procedure-call scratch register
r3	Argument register 4/scratch register 4
r2	Argument register 3/scratch register 3
r1	Argument register 2/scratch register 2
r0	Argument register 1/scratch register 1

Exception Returns

- Exception returns occur when a the CPU is in handler mode and a `EXC_RETURN` value is loaded into `pc`
- CPU restores context from stack and resumes execution from `ReturnAddress`
- CPU may remain in handler mode after an exception return
  - e.g. nested interrupts

The EXC_RETURN values

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	F	M	S	0	1

Bit	Description
F	If 0, floating-point context is present and to be loaded If 1, floating-point context may be present but not to be loaded
M	If 0, return from exception and resume execution in Handler mode If 1, return from exception and resume execution in Thread mode
S	If 0, load saved context from msp, resume execution using msp as sp If 1, load saved context from psp, resume execution using psp as sp

- Loading an EXC_RETURN value from outside handler mode into pc will cause an instruction fetch from that location
  - Location is NX (No-eXecute), CPU will raise exception

Exception Priority

- Exception priority is configurable (for the most part)
- Reset, NMI, and HardFault have fixed priority
- Higher priority exception will preempt lower priority exceptions
- Highest configurable priority value is 0

Exception	Priority
Reset	-3
NMI	-2
HardFault	-1
MemManage	0
BusFault	0
UsageFault	0
…	0

Exception Priority

- Configuring Priority
  - System Handler Priority Registers (`SHPRx`) for core exceptions with configurable priority
  - Interrupt Priority Registers (`NVIC_IPRx`) for external interrupt sources
- Registers divided into four 8 bit fields which hold priority for exception

System Handler Priority Registers

	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SHPR1	Reserved								PRI_6								PRI_5								PRI_4							
SHPR2	PRI_11								Reserved																							
SHPR3	PRI_15								PRI_14								Reserved															

Field	Description
PRI_4	Priority of system handler 4, MemManage
PRI_5	Priority of system handler 5, BusFault
PRI_6	Priority of system handler 6, UsageFault
PRI_11	Priority of system handler 11, SVCall
PRI_14	Priority of system handler 14, PendSV
PRI_15	Priority of system handler 15, SysTick

Interrupt Priority Registers

- Vendor specific configuration
- Vendor defined allowed priority values
- Check your microcontroller datasheet for details

MSP432 Interrupt Priority Registers

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

- TI defines 3 bit priority
  - Only highest three bits of the octet are used
  - Lower five bits are reserved (RAZ/WI)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0			Reserved					0			Reserved				
RW			RAZ/WI					RW			RAZ/WI				
15	**14**	**13**	**12**	**11**	**10**	**9**	**8**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0			Reserved					0			Reserved				
RW			RAZ/WI					RW			RAZ/WI				

If configuring registers directly in C or assembly, allowable priority values are `0x00`, `0x20`, `0x40`, `0x60` , `0x80` , `0xa0` , `0xc0` , `0xe0`

MSP432 Interrupt Priority Registers

- Setting Interrupt Priority with CMSIS-Core
  - Use `IRQ_SetPriority(number, priority)`
- `number`: Interrupt number
- `priority`: Interrupt priority
  - Valid values are 0, 1, 2, 3, 4, 5, 6, 7
  - `IRQ_SetPriority()` will shift these values automatically

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Service Requests

- Issued by application software to request actions from privileged software
- Useful when implementing an operating system
  - Keep direct hardware access from software
  - Allow OS to mediate access between software and hardware
  - OS in charge of distributing resources

Service Requests

- Execute the `svc` instruction

```
svc<c> #<imm>
```

`c`	Optional condition code
`imm`	Optional 8 bit immediate

- Hardware does not check imm field
- Value is a hint to supervisor software to take an action

The `SVCall` Handler

- Executed when CPU encounters a svc instruction
- Passing arguments to `SVCall` is ABI dependent
  - Your system's design, your choice
  - Recommendation
    - use `imm` for service number
    - pass four arguments in `r0`—`r3`

The SVCall Handler

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Example assumptions:

- Functions `sys_write()`, `sys_read()`, `sys_open()`, and `sys_close()` are defined
- Privilege separation is enabled, process uses `psp`, system uses `msp`
- FPU not used

The `SVCall` Handler

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

```
svcall_vect:
        push {r4, r5, r6, lr}          /* push temporary registers and link register */
        mrs r4, psp                    /* get process stack pointer */
        ldr r5, [r4, #24]              /* get program counter after svc instruction */
        ldrh r5, [r5, #-2]             /* load svc instruction (halfword, two bytes) */
        and r5, r5, #0xff              /* extract imm from svc instruction */
        lsl r5, #2                     /* make it into a word */
        ldr r6, syscall_table_size     /* load the size of the service call table */
        cmp r5, r6                     /* ensure we are within bounds */
        ittet lt                       /* if so, then */
        ldrlt r6, =syscall_table       /* load pointer to service call table and */
        ldrlt r6, [r5, r6]             /* get address of handler function */
        movge r0, #-1                  /* otherwise, move error into return */
        blxlt r6                       /* or dispatch handler */
        str r0, [r4]                   /* store return value into process stack */
        pop {r4, r5, r6, pc}           /* restore temporary registers and return */
        .align 4
syscall_table:
        .word sys_write, sys_read, sys_open, sys_close
syscall_table_size:
        .word . - syscall_table
```

Wrapping the Service Call in C


```
/* C Library Implementation, write.c */
extern int errno;
extern ssize_t __write(int fildes, const void* buf, size_t count);

ssize_t write(int fildes, const void* buf, size_t count) {
	size_t ret;
	ret = __write(fildes, buf, count);
	return ret < 0 ? -!!(errno = -ret) : ret;
}

/* Assembly wrapper, __write.S */
	.globl __write
	.type __write, %function
__write:
	svc #0
	bx lr
```

Kernel-Side Operation

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

```c
/* sys_write.c */
ssize_t sys_write(int fildes, const void* buf, size_t count) {
	ssize_t (*write_fn)(const void*, size_t);
	if(!fildes_valid_for_task(fildes)) {
		return -EBADF;
	}
	if(!is_fildes_writable(fildes)) {
		return -EINVAL;
	}
	if(!(write_fn = get_write_fn(fildes))) {
		return -EBADF;
	}
	return write_fn(buf, count);
}
```

System Timer

- The System Timer (SysTick) is a 24 bit decrementing counter
- Can be employed in different ways
  - RTOS scheduler
  - Track time
- Configured using the memory mapped `SYST_*` registers starting at address `0xe000e010`
- Triggers the SysTick vector

System Timer

Register	Description
SYST_CSR	SysTick Control and Status Register • COUNTFLAG (bit 16): If set, timer has counted to zero since the last time this register was read. This bit is cleared on software reads to this register and writes to SYST_CVR. • CLKSOURCE (bit 2): If set, SysTick uses the processor clock, otherwise SysTick uses an *implementation defined* external reference clock. If no external clock is implemented, this bit reads as 1 and ignores writes. • TICKINT (bit 1): If set, a count to 0 changes the SysTick exception status to pending, triggering an interrupt request. Writing a 0 to SYST_CVR does not change the status of the SysTick exception to pending. • ENABLE (bit 0): If set, counter is enabled and counts down from SYST_RVR. Upon counter reaching 0, the value in SYST_RVR is reloaded into SYST_CVR.
SYST_RVR	SysTick Reload Value Register. Holds the 24 bit reload value used by the SysTick counter. SysTick counts down from this value.
SYST_CVR	SysTick Current Value Register. Reads or clears the 24 bit current counter value. Any writes to this register clears it.

System Timer

Register	Description
SYST_CALIB	SysTick Calibration Value Register. Contains calibration parameters for the SysTick implementation. • NOREF (bit 31): If set, indicates that the *implementation defined* reference clock is not implemented. If this bit is set, then the CLKSOURCE bit in SYST_CSR is forced to 1 and can not be cleared. • SKEW (bit 30): If set, indicates that the stored 10 ms calibration value is inexact because of the clock frequency used. • TENMS (bits 23 to 0): Holds a reload value to be used for 10 ms (100 Hz) subject to system clock skew errors. If the value stored is 0, the calibration value is unknown.

System Timer

- Enabling the System Timer through CMSIS-Core
  - Use the `SysTick_Config()` function

```
int ret = SysTick_Config(ticks_to_count);
if(!ret) {
      /* handle error */
}
```

System Timer

- Enabling SysTick from C without CMSIS

```
#define SYST_CSR (*((volatile unsigned int*)(0xe000e010)))
#define SYST_RVR (*((volatile unsigned int*)(0xe000e014)))
#define SYST_CALIB (*((volatile unsigned int*)(0xe000e01c)))
#define SYST_ENABLE (1 << 0)
#define SYST_TICKINT (1 << 1)

void systick_init(void) {
	SYST_RVR = SYST_CALIB & ((1 << 24) - 1);
	SYST_CSR = SYST_ENABLE | SYST_TICKINT;
}
```

SysTick and Scheduling

- Trigger `SysTick` vector to initiate a context switch?
- Problems:
  - Scheduler runs at lowest priority to avoid missing critical interrupts
  - Set SysTick interrupt to lowest priority
  - SysTick can be used for more than just scheduling
  - We miss SysTick events
- Solution:
  - Don't run the scheduler on `SysTick` vector
  - Use `SysTick` vector to signal that the scheduler should be run

Pendable Services

- ARMv7-M provides the Pendable Service interrupt for this purpose
- Scheduling, take 2:
  - Configure `PendSV` to lowest priority
  - Allow `SysTick` to trigger at normal priority
  - Let SysTick set the `PendSV` flag
  - Allow `PendSV` to run at its own time
  - No critical interrupts missed!
- PendSV is configured through the `ICSR` register

Requesting PendSV

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

```c
/* 100 SysTick interrupts per CPU quantum */
#define TICKS_PER_QUANTUM 100

#define ICSR (*((volatile unsigned int*)(0xe000ed04)))
#define ICSR_PENDSVSET (1 << 28)

static volatile unsigned int ticks_count;

void systick_handler(void) {
        ticks_count++;
        if(ticks_count == TICKS_PER_QUANTUM) {
                /* time for the scheduler to run! */
                ticks_count = 0;
                ICSR |= ICSR_PENDSVSET;
        }
        /* handle other SysTick related events */
}
```

PendSV Interrupt and Scheduling

- The job of the scheduler
  - Save the execution context of the current running task
  - Pick a new runnable task, this is the newly scheduled task
  - Restore the context of the newly scheduled task
  - Resume execution of the newly scheduled task
- The way tasks are handled by the operating system depends on the implementation
- Commonly, this is done using a Process Control Block (PCB) or Task Control Block (TCB) to store task information
- At least a portion of the scheduler must be written in assembly

PendSV Interrupt and Scheduling

UF Nelms Institute for the Connected World
UNIVERSITY of FLORIDA

Example assumptions:

- Kernel keeps a `current` pointer for the current task
- The `schedule()` function takes the current process's stack pointer and returns the newly scheduled task's stack pointer

```
void* schedule(void* sp);
```

- Context switches will occur only from non-privilege to privilege mode
- Applications use their own process stack (`psp`)
- FPU is not used
- Registers are saved in the process's stack

PendSV Interrupt and Scheduling


```
        .globl pendsv_vect
        .type pendsv_vect, %function
        .align 4
pendsv_vect:
        push {lr}
        mrs r0, psp
        stmdb r0!, {r4-r11}
        msr psp, r0
        bl schedule
        ldmia r0!, {r4-r11}
        msr psp, r0
        pop {pc}
```