Embedded Operating Systems

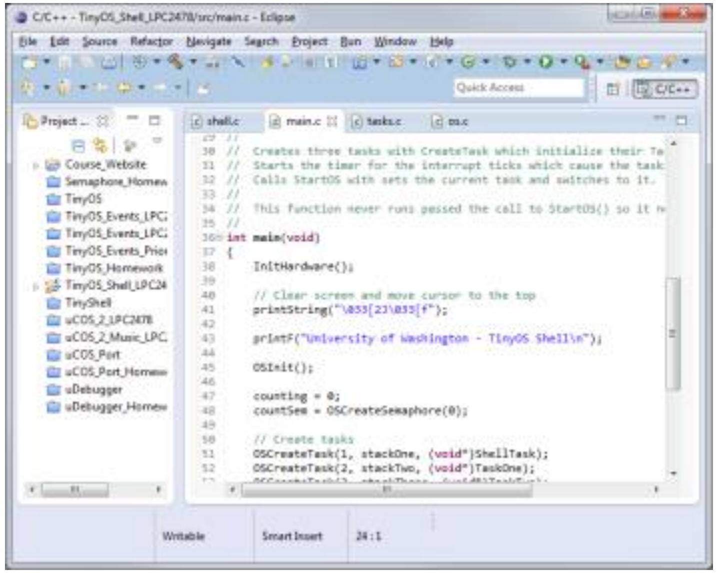
Condensed version of Embedded Operating Systems course.

Or how to write a TinyOS

Part 1

John Hatch

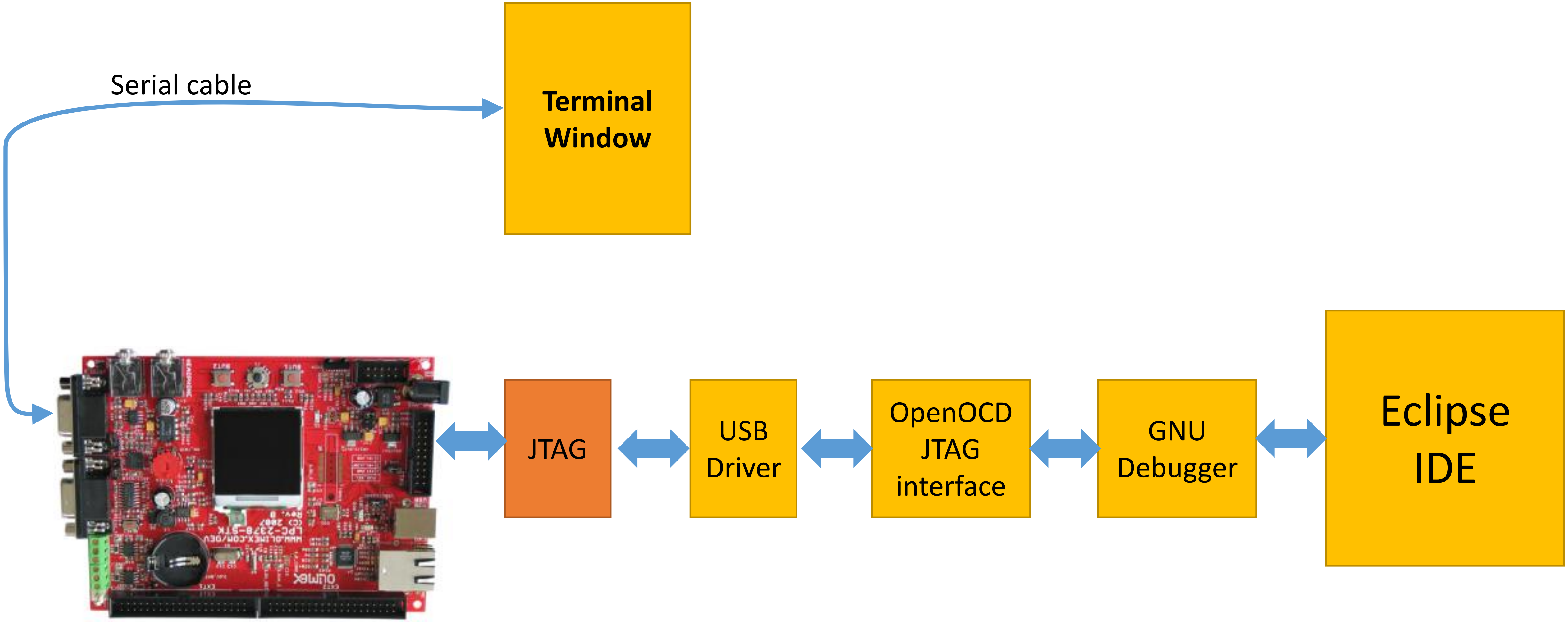
Tool Chain and Hardware



```
//
//  Creates three tasks with CreateTask which initialize their Te
//  Starts the timer for the interrupt ticks which cause the task
//  Calls StartOS with sets the current task and switches to it.
//
//  This function never runs passed the call to StartOS() so it n
//
int main(void)
{
    InitHardware();

    // Clear screen and move cursor to the top
    printString("\033[2J\033[f");

    printf("University of Washington - TinyOS Shell\n");

    OSInit();

    counting = 0;
    countSem = OSCreateSemaphore(0);

    // Create tasks
    OSCreateTask(1, stackOne, (void*)ShellTask);
    OSCreateTask(2, stackTwo, (void*)TaskOne);
```

- Eclipse IDE for C++
  - With Zylin CDT – adds support for embedded projects
- GCC built for ARM
  - Yagarto - 'Yet Another GNU ARM toolchain' http://www.yagarto.de/
- OpenOCD
  - JTAG debugger interface
- Olimex JTAG dongle
- Olimex LPC2378_STK or LPC2478_STK Arm7 board

Development Setup



Serial cable

Terminal Window

JTAG

USB Driver

OpenOCD JTAG interface

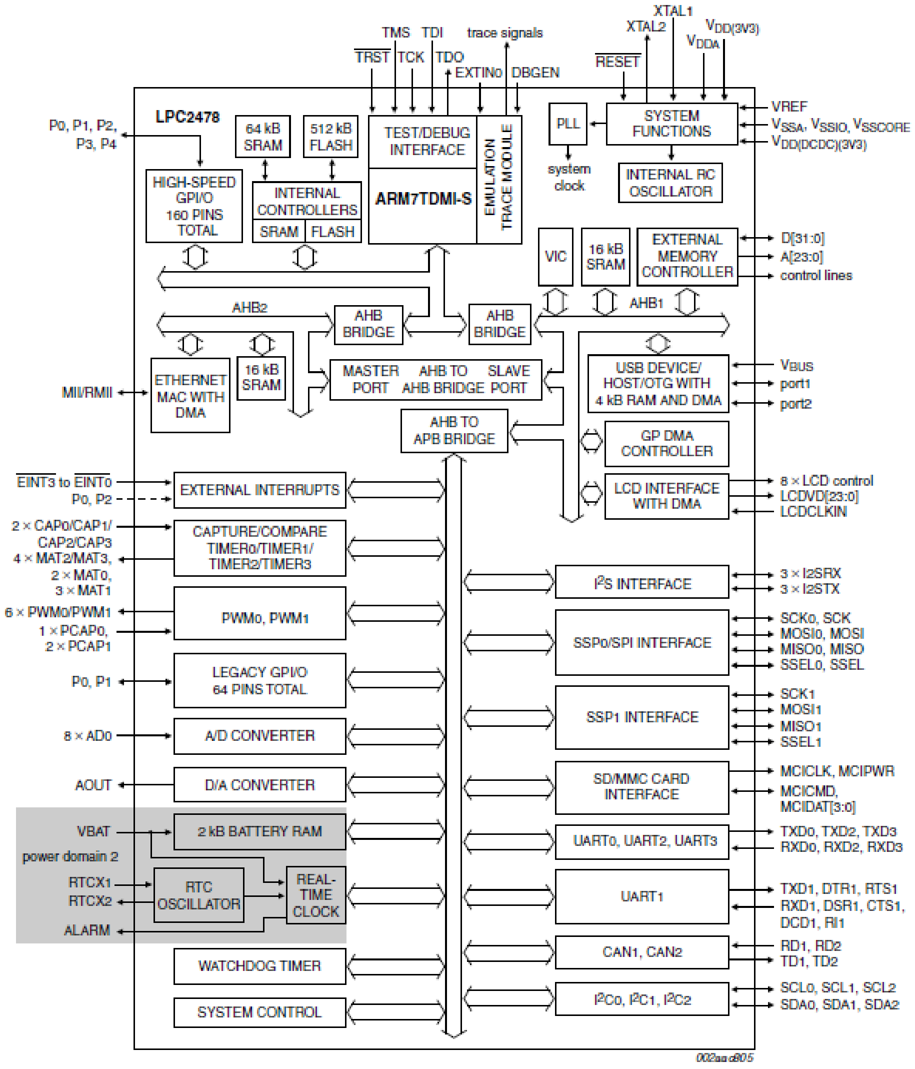
GNU Debugger

Eclipse IDE

ARM CPU Overview

- ARM is a RISC machine
  - RISC – Reduce Instruction Set Computer
  - Relies on registers and shorter instructions to do work.
- ARM Limited is a IP based company in Cambridge England that designs and licenses the ARM architecture.
  - What is licensed is called hard IP and it is the blueprints to how to build ARM cores into custom processors.
  - Licensees include Samsung, Philips, TI, Qualcomm, and many others.
- ARM cores are combined with hard IP from different vendors to produce System on a Chip processors.
- The industry is driving towards putting all mobile device functionality into a single processor.
  - Drives down costs
  - Increases reliability

The LPC2478 Processor



- Small system on a chip
- ARM 7 CPU
- LCD controller
- Ethernet
- 512 kb flash memory
- SD Card interface
- Serial Ports
- Can buses
- SPI and I^{2}C interfaces
- A/D and D/A converters

ARM Instruction Set

- Load and Store instruction set
  - Memory is only read or written
  - Actual work is done only in the registers
- An instruction is executed every clock cycle
  - Accomplished with by pipeline that is working on multiple instructions
  - Pipeline stages are roughly load, decode, then execute
- Instruction have many addressing modes
  - Addressing modes allow a register to behave as numerical values, as pointers, or as indexes.
- Most instructions can be conditional
  - Instructions can be coded to check conditional flags when executing
  - For example ‘subtract if not zero’ is one instruction
- Special instructions for status registers

ARM 7 Architecture

- 15 Regular Registers
- 1 Current Program Status Register
- 7 modes
  - User – Unprivileged with regular registers
  - System – Privileged with regular registers
  - Supervisor – Privileged mode for software interrupts
  - Abort – Privileged mode for data and instruction aborts
  - Undefined – Privileged mode for undefined instructions. Used for emulation.
  - IRQ – Privileged mode for hardware interrupts
  - FIQ – Privileged mode for fast hardware interrupts
- Some modes have special registers that are only visible in that mode.
  - They mask out the regular registers when they are active.
  - Retain their values even when not visible.
  - Used to handle exceptions.

ARM Registers and Modes

User/System	Supervisor	Abort	Undefined	IRQ	FIQ
R0					
R1					
R2					
R3					
R4					
R5					
R6					
R7					
R8					R8_FIQ
R9					R9_FIQ
R10					R10_FIQ
R11					R11_FIQ
R12					R12_FIQ
R13	R13_SVC	R13_ABT	R13_UND	R13_IRQ	R13_FIQ
R14	R14_SVC	R14_ABT	R14_UND	R14_IRQ	R14_FIQ
PC					
CPSR					
	SPSR_SVC	SPSR_ABT	SPSR_UND	SPSR_IRQ	SPSR_FIQ

Most modes have special registers that are hidden from the other modes.

When the mode becomes active its registers become visible hiding the similar registers of the other modes.

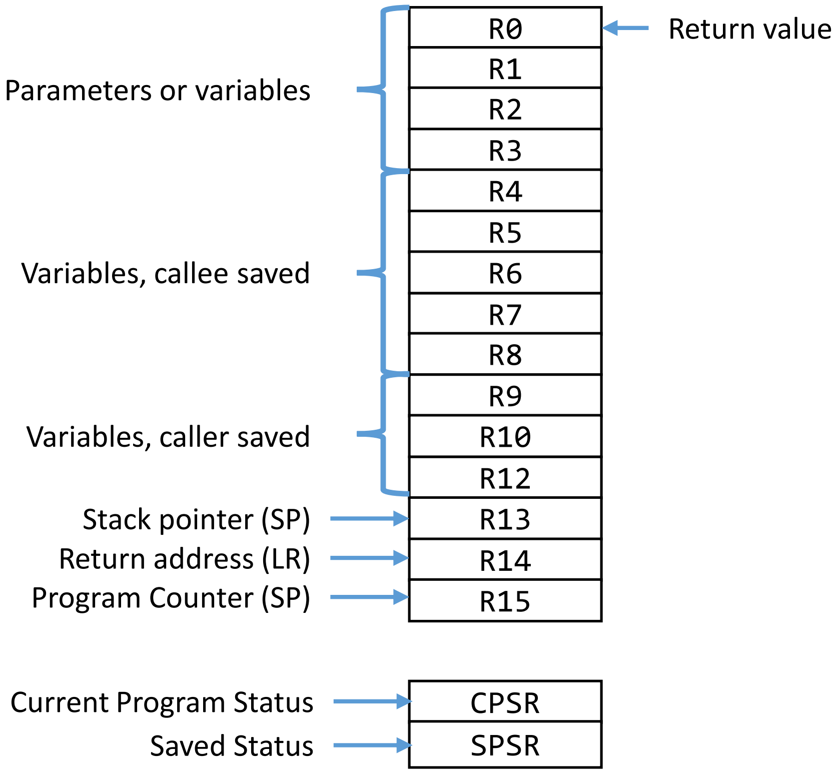
That is, when in abort mode the abort modes R13 is accessible and all the other R13s are hidden.

The values in the hidden registers are preserved and are available the next time the CPU is in the mode.

Calling Standard

- The calling standard defines a convention for how the registers are used and how function calls are written on the platform.
  - The convention is follow by all the tools.
  - Compilers, assemblers, linkers, and debuggers
- Following the rules lets you write code that can call C functions from assembler and assembler functions from C.
- Each CPU family has it own calling convention.
  - X86, ARM, Mips, SH, PPC

ARM Register Convention



Parameters or variables

Variables, callee saved

Variables, caller saved

Stack pointer (SP)

Return address (LR)

Program Counter (SP)

Current Program Status

Saved Status

R0

R1

R2

R3

R4

R5

R6

R7

R8

R9

R10

R12

R13

R14

R15

CPSR

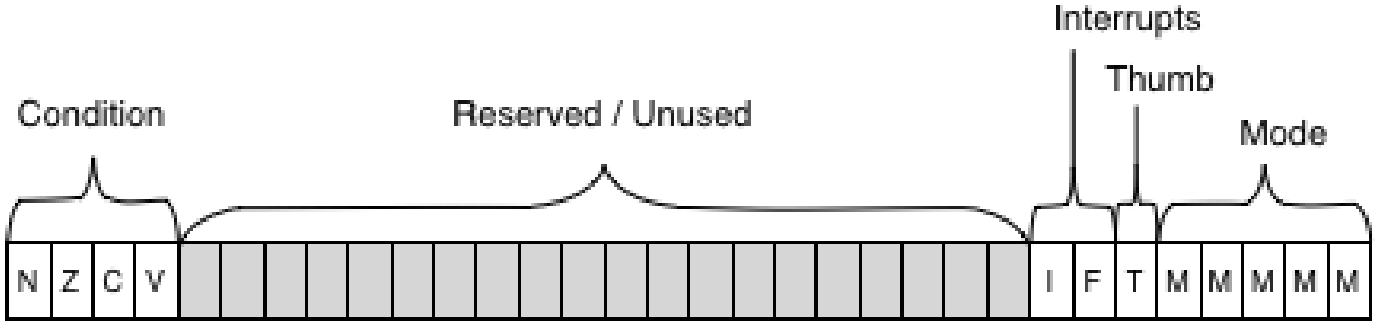
SPSR

Return value

- R0-R3 are used to pass parameters
  - Typically one to four int sized parameters
  - What doesn’t fit goes on the stack.
- R0 holds the return value
- R4-R8 – Are used as variables
  - If used by callee, callee must save and restore them.
- R9-R12 – Are used as variables
  - Saved by caller if important.
- R13 is the stack pointer (SP)
- R14 is the return address (LR)
- R15 is current address or Program Counter (PC)

CPSR – the current program status register

ARM 7 CPSR

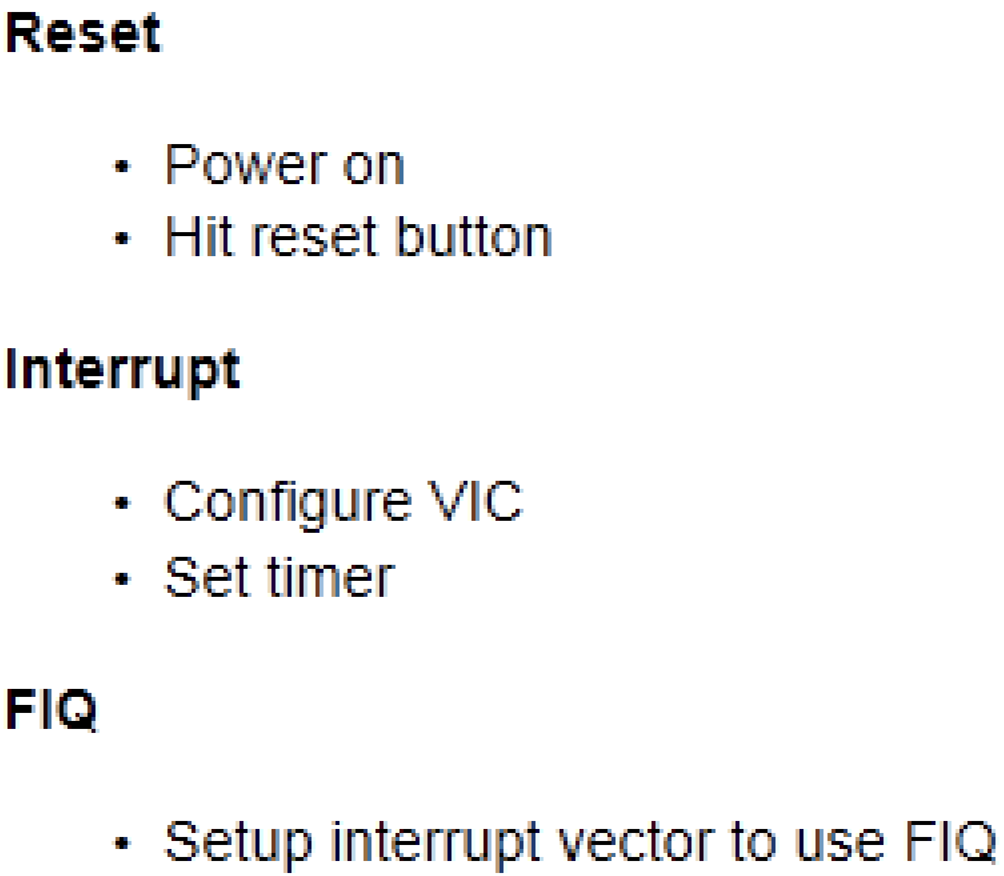


- The CPSR is a special purpose register that holds the condition codes, CPU mode, interrupt control flags, and other special purpose bits.
  - Conditions codes are result flags from previous instructions that can be tested by the current instruction.
  - Interrupt control flags are used to turn interrupts off or on.
  - Mode bits show what mode the CPU is in and can be used to change modes.
  - The T bit controls whether the CPU is using 32 bit instructions or special 16 bit Thumb instructions.
  - Variations in ARM cores add other special purpose bits.
- The SPRS is used by the CPU to save the current CPSR during exception handling.

Exceptions

- Exceptions are CPU events that interrupt the current program.
  - External hardware interrupt
    - timers, serial port, touch panel
    - Up to 32 external interrupt sources
  - Software interrupt
    - Special SWI instruction for system calls
  - Data and code aborts
    - bad pointers
    - bad jumps
  - Undefined instructions
    - Invalid instruction
    - Or instructions for a co-processor such as a floating point co-processor

Exceptions from Hardware Examples



Reset

- Power on
- Hit reset button

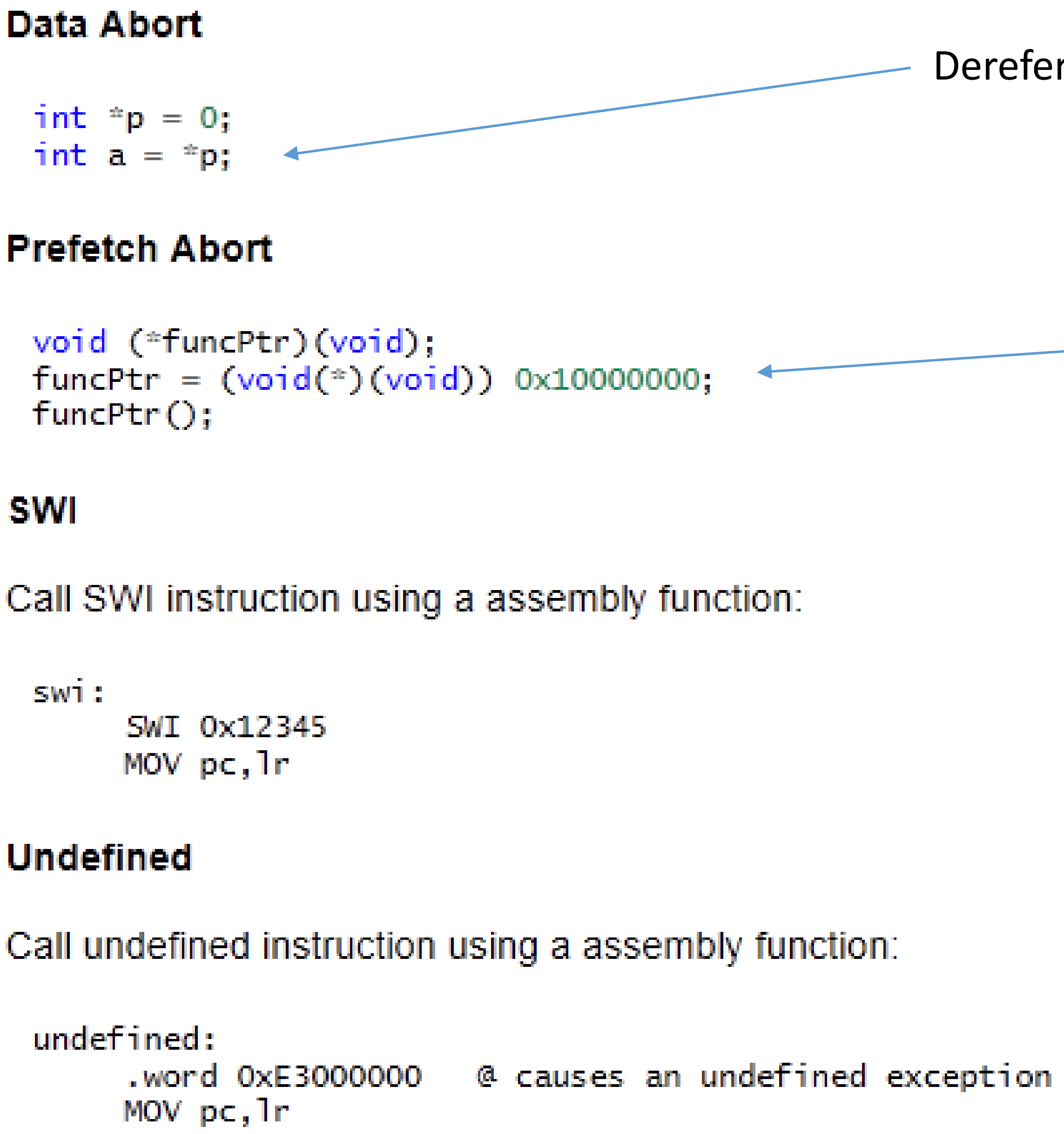
Interrupt

- Configure VIC
- Set timer

FIQ

- Setup interrupt vector to use FIQ

Exceptions from Code Examples



Data Abort

```
int *p = 0;
int a = *p;
```

Dereferencing a null pointer

Prefetch Abort

```
void (*funcPtr)(void);
funcPtr = (void(*)(void)) 0x10000000;
funcPtr();
```

Known bad address on the LPC2478

SWI

Call SWI instruction using a assembly function:

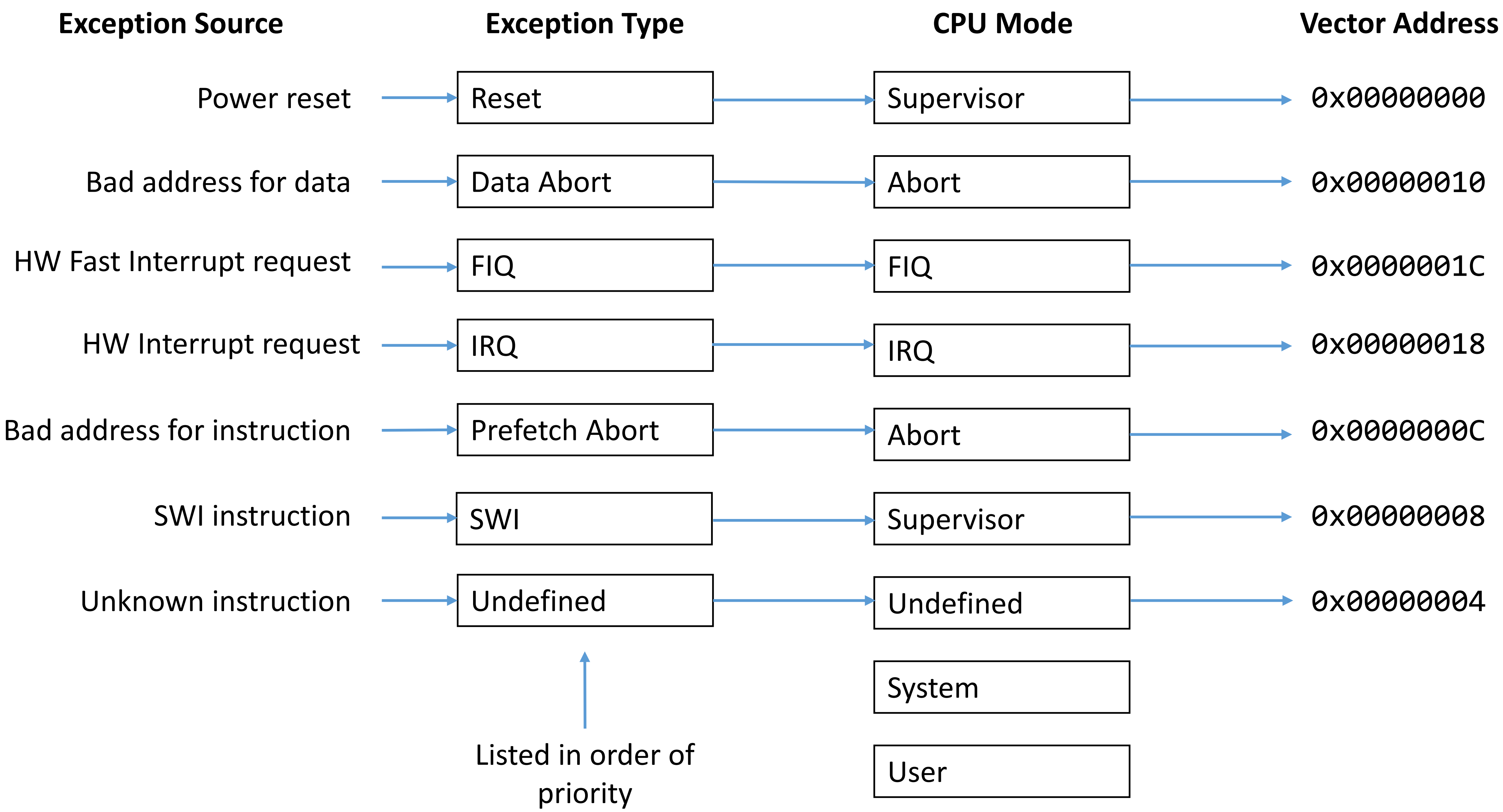
```
swi:
     SWI 0x12345
     MOV pc,lr
```

Undefined

Call undefined instruction using a assembly function:

```
undefined:
     .word 0xE3000000   @ causes an undefined exception
     MOV pc,lr
```

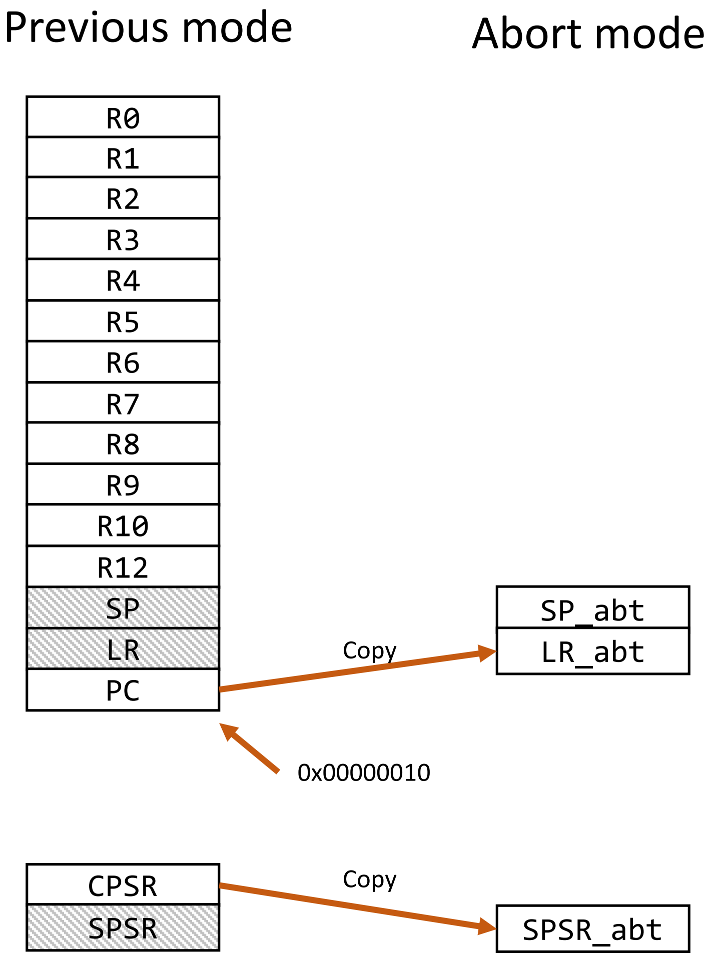
Mapping of Exceptions to Mode



Exception Source	Exception Type	CPU Mode	Vector Address
Power reset	Reset	Supervisor	0x00000000
Bad address for data	Data Abort	Abort	0x00000010
HW Fast Interrupt request	FIQ	FIQ	0x0000001C
HW Interrupt request	IRQ	IRQ	0x00000018
Bad address for instruction	Prefetch Abort	Abort	0x0000000C
SWI instruction	SWI	Supervisor	0x00000008
Unknown instruction	Undefined	Undefined	0x00000004
		System	
		User	

Listed in order of priority

Exception and mode switch



When an exception occurs the CPU will always perform the following actions:

1. Turn off interrupts
2. Switch to the mode that of the exception type if not already in it.
3. Unmask the mode's special registers hiding the other mode's special registers.
4. Copy the CPSR value into the mode's SPSR
5. Copy the PC (R15) into the mode's LR (R14)
6. Set the PC to the mode's vector address

□ Active register

▨ Hidden register

Vector Table

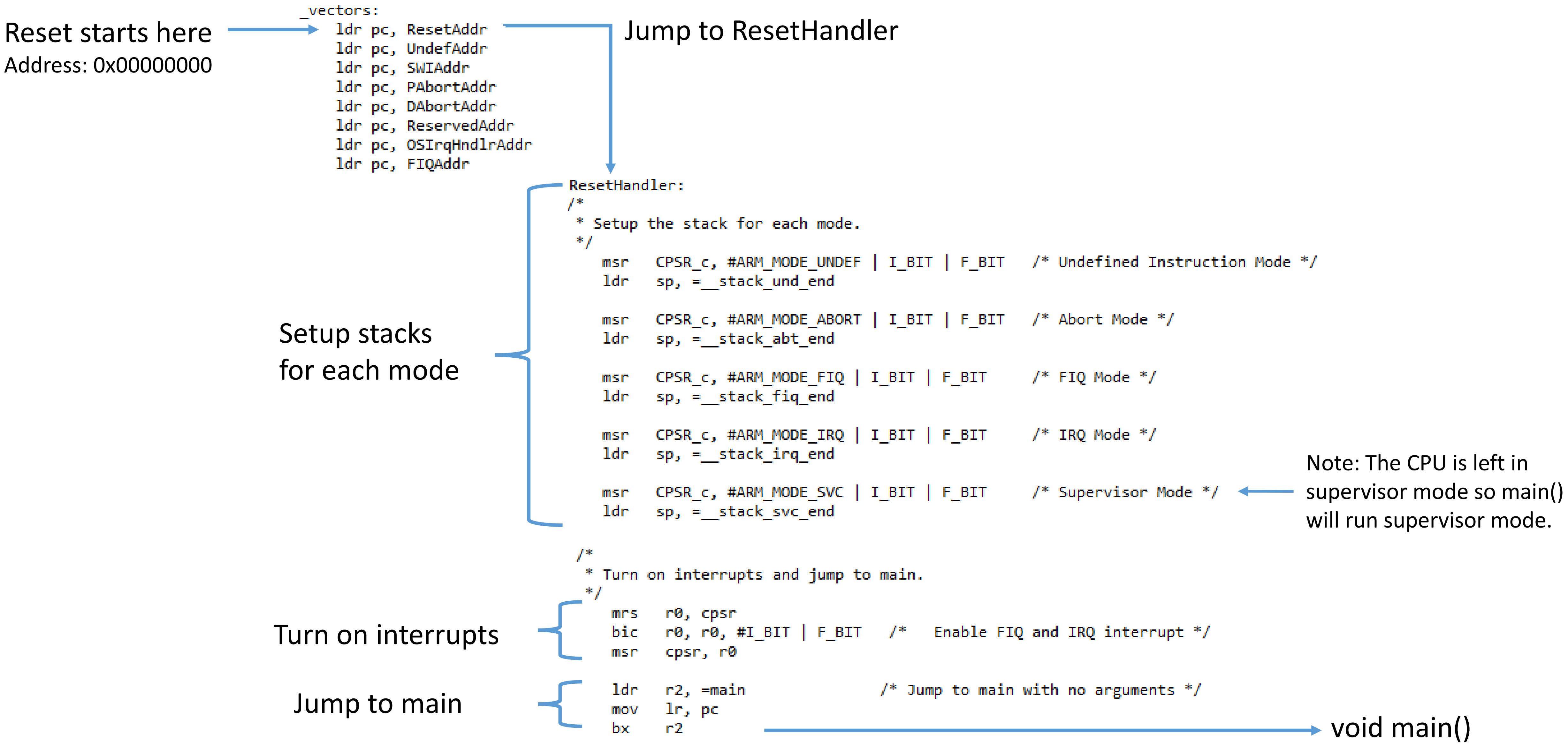
The vector table is set of instructions at fixed addresses where the CPU will jump to on an exception.

Vector table at the top of memory:

```
0x00000000   ldr pc, ResetAddr    /* Reset                 */
0x00000004   ldr pc, UndefAddr    /* Undefined instruction */
0x00000008   ldr pc, SWIAddr      /* Software interrupt    */
0x0000000C   ldr pc, PAbortAddr   /* Prefetch abort        */
0x00000010   ldr pc, DAbortAddr   /* Data abort            */
0x00000014   ldr pc, ReservedAddr /* Reserved              */
0x00000018   ldr pc, IRQAddr      /* IRQ interrupt         */
0x0000001C   ldr pc, FIQAddr      /* FIQ interrupt         */
```

In this example the vectors contain instructions to load the addresses of handler routines into the PC. When executed, they will cause a jump to the handling routine. Basically function pointers.

ARM Start Up – Starts with Reset exception



Reset starts here

Address: 0x00000000

```
_vectors:
    ldr pc, ResetAddr
    ldr pc, UndefAddr
    ldr pc, SWIAddr
    ldr pc, PAbortAddr
    ldr pc, DAbortAddr
    ldr pc, ReservedAddr
    ldr pc, OSIrqHndlrAddr
    ldr pc, FIQAddr
```

Jump to ResetHandler

Setup stacks for each mode

```
ResetHandler:
/*
 * Setup the stack for each mode.
 */
    msr   CPSR_c, #ARM_MODE_UNDEF | I_BIT | F_BIT   /* Undefined Instruction Mode */
    ldr   sp, =__stack_und_end

    msr   CPSR_c, #ARM_MODE_ABORT | I_BIT | F_BIT   /* Abort Mode */
    ldr   sp, =__stack_abt_end

    msr   CPSR_c, #ARM_MODE_FIQ | I_BIT | F_BIT     /* FIQ Mode */
    ldr   sp, =__stack_fiq_end

    msr   CPSR_c, #ARM_MODE_IRQ | I_BIT | F_BIT     /* IRQ Mode */
    ldr   sp, =__stack_irq_end

    msr   CPSR_c, #ARM_MODE_SVC | I_BIT | F_BIT     /* Supervisor Mode */
    ldr   sp, =__stack_svc_end
```

Note: The CPU is left in supervisor mode so main() will run supervisor mode.

Turn on interrupts

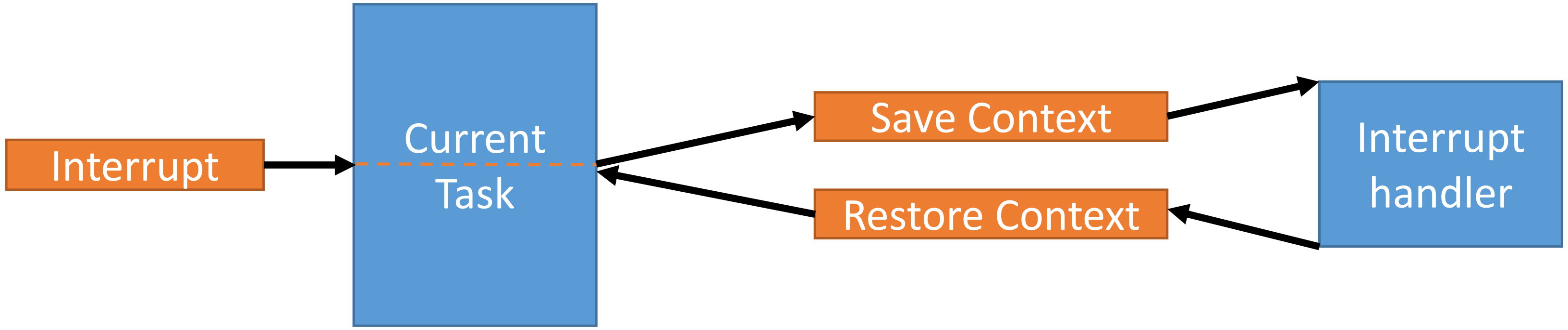
Jump to main

```
 /*
  * Turn on interrupts and jump to main.
  */
     mrs   r0, cpsr
     bic   r0, r0, #I_BIT | F_BIT   /*   Enable FIQ and IRQ interrupt */
     msr   cpsr, r0

     ldr   r2, =main                /* Jump to main with no arguments */
     mov   lr, pc
     bx    r2
```

void main()

Context Switch

- The context is current state of the CPU.
  - It includes all the values in the registers and the value of the CPRS.
  - R13 has the stack pointer
  - R15 has the address of the current instruction
  - The context can be saved and restored later.
- Saving the context allows the CPU to switch from the current task to another task and return to exactly where it was by restoring the context.
- The context is typically saved to the stack of the current task



Saving Registers to the Stack

- Context is typically saved the stack
- The SP (R13) register is the stack pointer. It has the address of the top of the stack.
- The stack typically starts at high memory address and descends.
- The stack pointer typically points to the last filled position on the stack.
- ARM also supports ascending stacks and pointers to the next empty position.
- ARM has special instructions to make working with the stack easy
  - STM – Store multiple
  - LDM – Load multiple

To push the registers R0 through R12 to the current stack and update the stack pointer (SP) at the same time:

```
STMFD SP!, {R0-R12}
```

The ! means update register

The instruction reads store multiple registers R0 to R12 to a descending stack pointed to by SP and automatically update SP to point to the new full position on stack.

Restoring registers from the stack

- Restoring context is the opposite operation from saving it.
- The values on the stack are copied back into the registers.
- The Load multiple instruction is used to restore the registers.

To restore the registers R0 through R12 from the current stack and update the stack pointer (SP) at the same time:

```
LDMFD SP!, {R0-R12}
```

The instruction reads load multiple registers R0 to R14 from a descending stack pointed to by SP and automatically update SP to point to the new full position on stack.

Saving and Restoring Context with CPSR and PC

- In practice the Save Multiple and Load Multiple instructions can be used to not only to save and restore the general registers but also restore the CPSR and return to the original interrupted task.
- To do this we save the LR (R14) as the last register. It has the return address.
- On restore we make sure the value of LR goes directly into the PC. This causes the a jump back to exactly where the task was interrupted.
- And we use a special assembly directive (^) to indicate the SPSR should be copied into the CPSR. This restores the state of the CPU as it was before it was interrupted.

LR has the return address

```
SLMFD SP!, {R0-R12, LR}
```

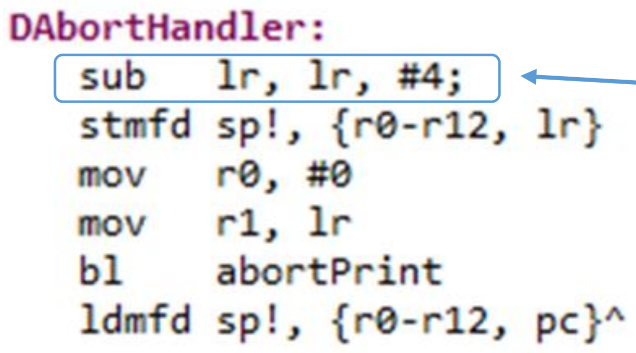
The return address is restored directly to the PC

The ^ means copy SPSR to CPSR

```
LDMFD SP!, {R0-R12, PC}^
```

Adjusting the return address

- Because of the CPU’s instruction pipeline, the PC is always ahead of the current instruction.
- When an exception occurs the PC is already pointing one or two instructions ahead depending on the exception type.
- The return address needs to be adjusted before it can be used.
- By subtracting either 4 bytes or 8 bytes.



```
DAbortHandler:
   sub   lr, lr, #4;
   stmfd sp!, {r0-r12, lr}
   mov   r0, #0
   mov   r1, lr
   bl    abortPrint
   ldmfd sp!, {r0-r12, pc}^
```

Subtracting 4 bytes so that the handler will return to the instruction immediately after the one that caused the data abort.

Data Abort Example - a µDebugger

This example will loop continuously causing data aborts.

If the data abort exception handler works correctly it will catch the abort, print out the address where the abort happened, and return back to the next line.

The code runs in supervisor mode.

```c
#include <stdint.h>
#include "lpc2378.h"
#include "init.h"
#include "print.h"

int main(void)
{
    initHardware();

    while(1)
    {
        printString("\nTesting Data Abort\n");

        int *p = (int*) 0x10000000;
        *p = 1;

        printString("\nAbort handled.\n");
    }

    return 0;
}
```

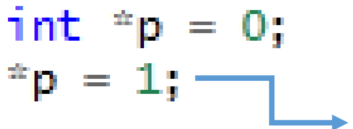
Bad address

Data abort!

Handler should return here

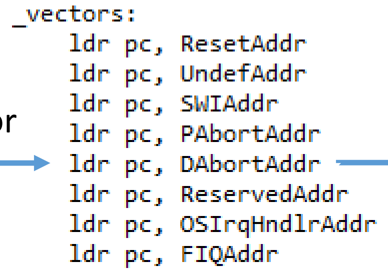
Handling a data abort exception

Causes Data Abort



```
int *p = 0;
*p = 1;
```

Switch to Abort mode
Jump to Data Abort vector
Address: 0x00000010



```
_vectors:
    ldr pc, ResetAddr
    ldr pc, UndefAddr
    ldr pc, SWIAddr
    ldr pc, PAbortAddr
    ldr pc, DAbortAddr
    ldr pc, ReservedAddr
    ldr pc, OSIrqHndlrAddr
    ldr pc, FIQAddr
```

Jump to
Data Abort
Handler

R0 is the first param
R1 is the second param

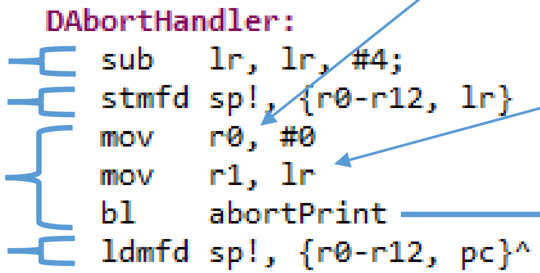
Address of where the
abort happened is in
the LR (R13) register

Adjust return address

Save Context

Call C routine

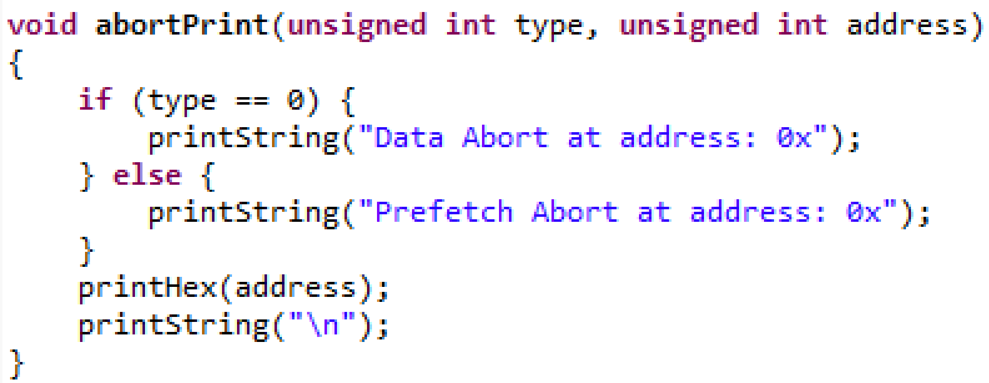
Restore Context



```
DAbortHandler:
    sub     lr, lr, #4;
    stmfd   sp!, {r0-r12, lr}
    mov     r0, #0
    mov     r1, lr
    bl      abortPrint
    ldmfd   sp!, {r0-r12, pc}^
```

Return to Supervisor mode
Jump back to the next line
in the original code with
interrupts re-enabled.

Prints the type of abort
and the address of
where it happened



```
void abortPrint(unsigned int type, unsigned int address)
{
    if (type == 0) {
        printString("Data Abort at address: 0x");
    } else {
        printString("Prefetch Abort at address: 0x");
    }
    printHex(address);
    printString("\n");
}
```

Observation about the exception handler

- Main is running in Supervisor mode. The exception handler runs in Abort mode
- The handler only saves a partial context to the Abort mode's stack.
  - It saves R0 through R12 and LR
  - But it does not save the original CPSR value saved in the SPSR
- Saving a partial context works because the Abort handler never leaves the Abort mode except to return back to the original task.
  - The value in SPSR is safe while in Abort mode.
  - The SPRS is restored back to the CPSR by the ^ directive on the LDMFD instruction.
- To switch another task, the value in the SPSR must be saved somewhere else such as the stack.
  - There are no instructions to move from the SPSR directly to memory.
  - The SPSR must first be moved to a general register then save to memory.
- The stack pointer of each task must be saved somewhere.
  - The right stack must be used to restore a given task.
- Saving the full context and switching between tasks in Supervisor requires a little bit more planning and setup.
  - To be covered in the next deck.

Summary

- ARM registers and modes
- ARM calling standard
- Exception types
- Mode switching on exception
- Vector table
- Context saving and restoring
- Adjusting the return address for exceptions
- Data abort handler example

Not covered – Switching tasks

Switching between tasks requires:

- Setting up stacks for each task
- Tracking stack pointers for each task
- Priming the stacks with an initial context
- Saving the full context
- Saving the task’s stack pointer
- Switching tasks by switching stack pointers
- Restoring the full context for next task
- Using the timer to drive context switches

To Be Continued in Part 2…

University of Washington Continuing Education

http://www.pce.uw.edu/

Embedded Operating Systems Class

https://courses.washington.edu/CP105

TinyOS

http://tinyos.rockfishnw.com