Použití mikropočítačů pro vývoj embedded aplikací

Ing. Jaroslav Lepka
Ing. Pavel Grasblum, Ph.D.

11. – 12. listopadu 2010

Tato prezentace je spolufinancována Evropským sociálním fondem a státním rozpočtem České republiky.

Calling Conventions & Stack Frame

Calling Conventions & Stack Frames

- Calling conventions
  - Language dependent
  - Processor dependent
    - Number of registers available for storing the parameters
    - Efficiency of data storage – byte, word (16-bit) , long (32-bit)

- Passing values to functions
  - Compiler scans the list of parameters from left to right
  - Compiler uses the registers of a processor to pass the values to functions
  - Compiler passes remaining parameter values on the stack
    - Stack has to be handled properly in terms of alignment, incrementing and decrementing

- Returning values from functions
  - Compiler returns function results in registers, stack or …..

Calling Conventions – DSC56F8025

- Passing values to functions – 56800E core
  - Compiler uses the following registers to pass parameter values to functions – A, B, R1, R2, R3, R4, Y0 and Y1
  - Upon a function call, the compiler scans the list of parameters from left to right and uses the registers as follows:
    - The first two 8-bit or 16-bit integer/fractional values – Y0 and Y1
    - The first two 32-bit integer/fractional values – A and B
    - The first four pointer parameter values – R2, R3, R4 and R1 (in that order)
    - The third and fourth 8-bit or 16-bit integer/fractional value – A and B (provided that the compiler does not use these registers for 32-bit parameter values)
    - The third 8-bit or 16-bit integer/fractional value – B (provided that the compiler does not use these registers for 32-bit parameter value)
    - The remaining parameters – compiler passes on the stack
      - The system increments the stack by the total amount of space required for memory parameters
      - This incrementing must be an even number of words, as the stack pointer (SP) must be continuously long-aligned
      - The system moves parameter values to the stack from left to right, beginning with
      - the stack location closest to the SP
      - Because a long parameter must begin at an even address, the compiler introduces one-word gaps before long parameter values, as appropriate

Calling Conventions – DSC56F8025 cont'd

- Returning values from functions
  - Compiler returns function results in registers
    - 8-bit integer values — Y0.
    - 16-bit integer values — Y0.
    - 32-bit integer or float values — A.
    - All pointer values — R2.
    - Structure results — R0 contains a pointer to a temporary space allocated by the caller. (The pointer is a hidden parameter value.)
- Volatile and non-volatile registers
  - Non-Volatile registers – C0, C1, C10, D0, D1, D10 and R5 (in some cases)
    - Values in volatile registers can be saved across functions calls
    - Another term for such registers – SOC (Saved Over a Call) registers
  - Volatile registers – all ALU, AGU except Non-Volatile registers
    - Values in volatile registers cannot be saved across functions calls
    - Another term for such registers – non-SOC registers

Calling Convention - Example

- Frac16 Fcn_4(Frac16 f16A, Frac16 f16B, Frac16 f16C, Frac16 f16D);

Y0 Y0 Y1 A B

- Frac16 Fcn_3(Frac16 f16A, Frac16 f16B, Frac16 f16C, Frac16 f16D, Frac32 f32A, Frac32 f32B);

Y0 Y0 Y1 SP SP-1 SP+2

A B

- Frac16 **Fcn_5**(Frac16 f16A, Frac16 f16B, Frac16 f16C, Frac16 f16D, Frac32 f32A, Frac32 f32B, Frac32 f32C, Frac16 *pF16A, Frac16 *pF16B, Frac16 *pF16C, Frac16 *pF16D, Frac16 *pF16E);

Y0 Y0 Y1 SP SP-1 A B SP+6

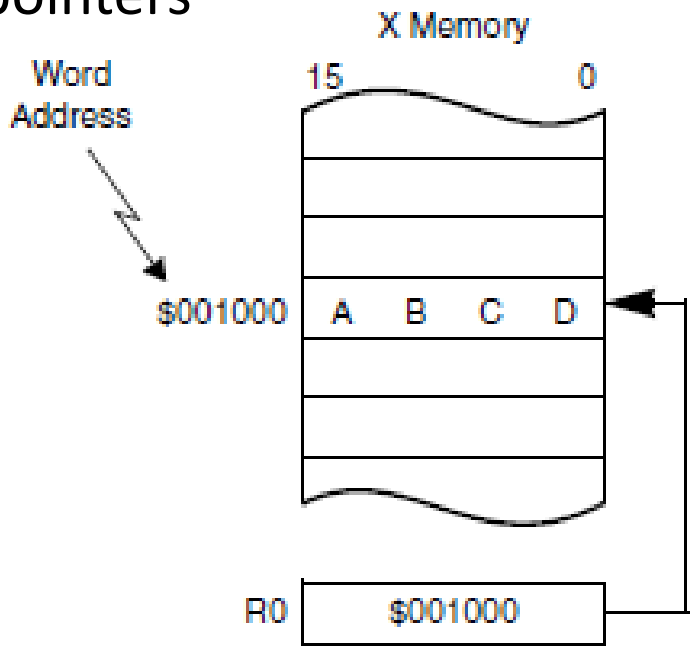
SP-2 R2 R3 R4 R1 SP-4

Memory Access and Pointers – Data Alignment

- The DSP56800E core was designed to operate as a word-addressable machine
- Instruction set allows to access: byte, word and long-word
- Accessing word values using word pointers

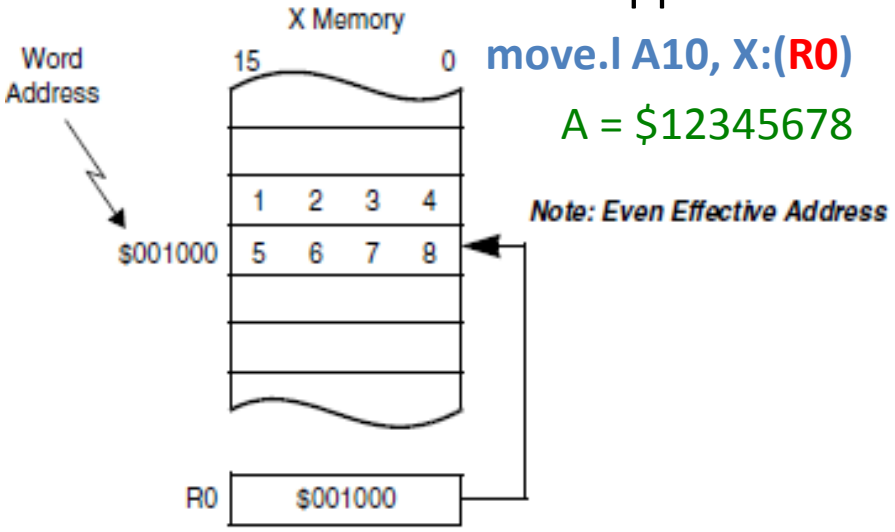
move.w A1, X:(R0)

A1 = $ABCD



Memory Access and Pointers – Data Alignment cont'd

- Accessing a long word in memory – always word address
- Always aligned to even word address
- Even address holds the lower word
- Odd address holds the upper word

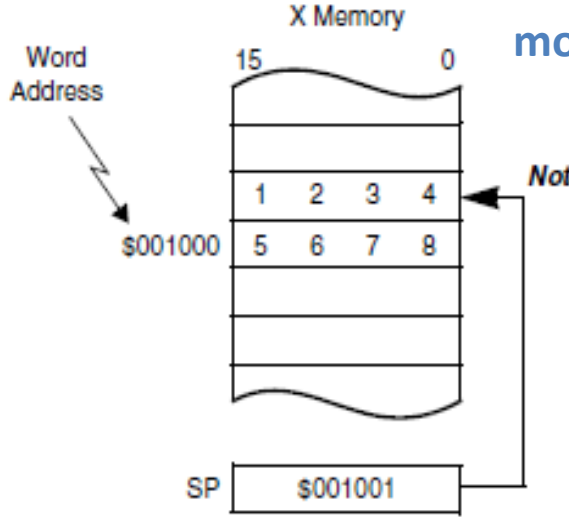


move.l A10, X:(R0)

A = $12345678

Note: Even Effective Address

- Accessing a long word on stack – always word address
- Always aligned to even word address
- Even address holds the lower word
- Odd address holds the upper word



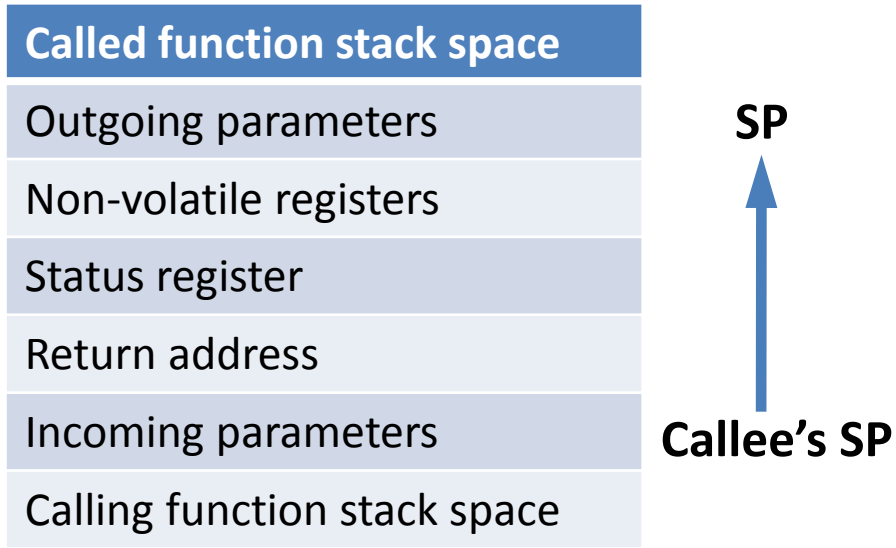
move.l A10, X:(SP)

A = $12345678

Note: Odd Effective Address

Stack Frame and Alignment

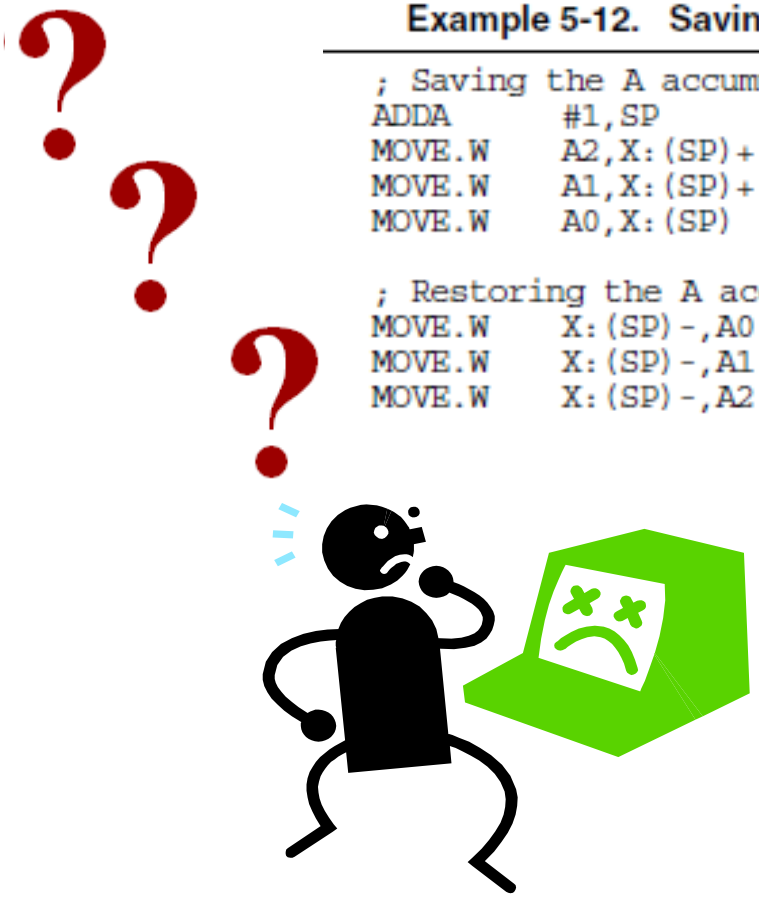
- Compiler always must operate with the SP long aligned!!!!!!!
- Start-up code in runtime firs initializes the SP to an odd value
- At all times after, the SP must point to an odd word address
- The compiler never generates instruction that adds or subtracts an odd value from SP
- The compiler never generates:
  - move.w or moveu.w with X:(SP)+ or X:(SP)-



Example from the DSC Manual

Example 5-12. Saving and Restoring an Accumulator—Word Accesses

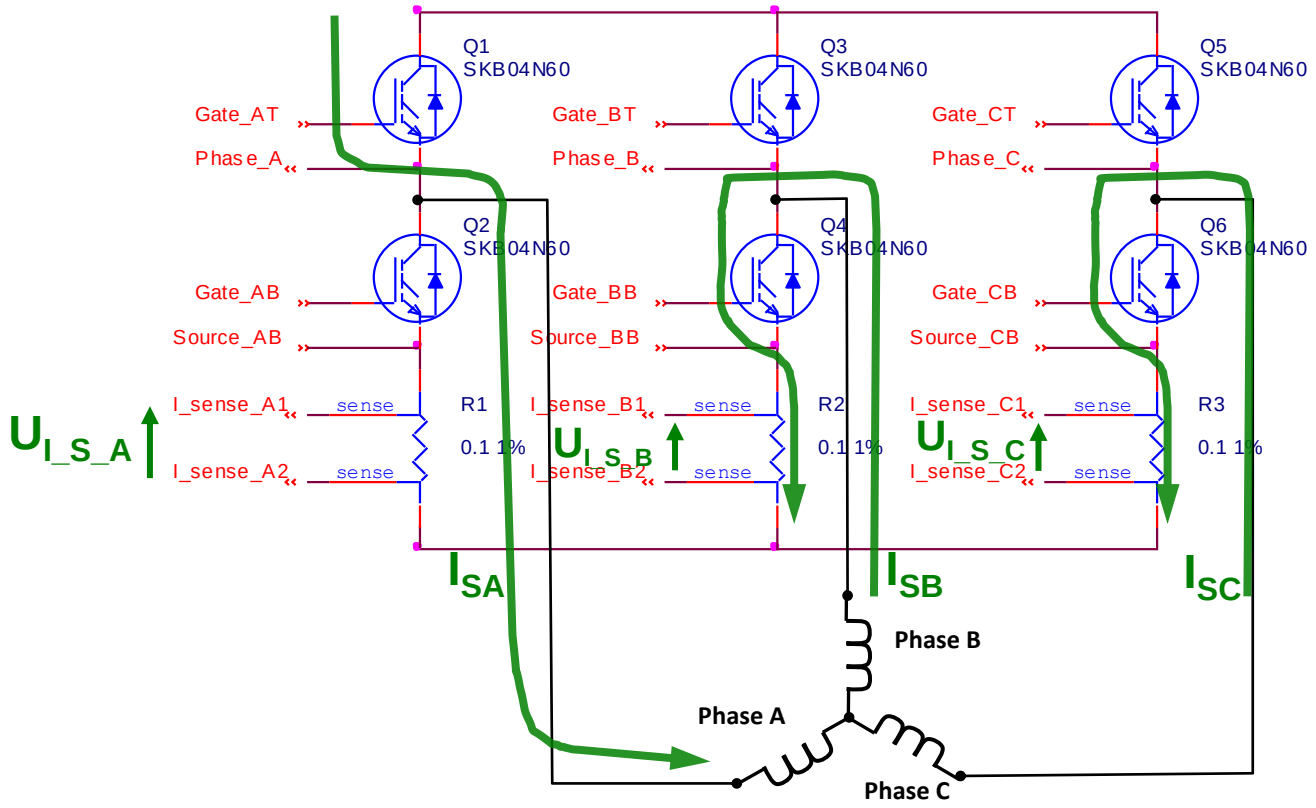
```
; Saving the A accumulator to the stack
ADDA      #1,SP          ; Point to first empty location
MOVE.W    A2,X:(SP)+     ; Save extension register
MOVE.W    A1,X:(SP)+     ; Save A1 register
MOVE.W    A0,X:(SP)      ; Save A0 register

; Restoring the A accumulator from the stack
MOVE.W    X:(SP)-,A0     ; Restore A0 register
MOVE.W    X:(SP)-,A1     ; Restore A1 register
MOVE.W    X:(SP)-,A2     ; Restore extension register
```



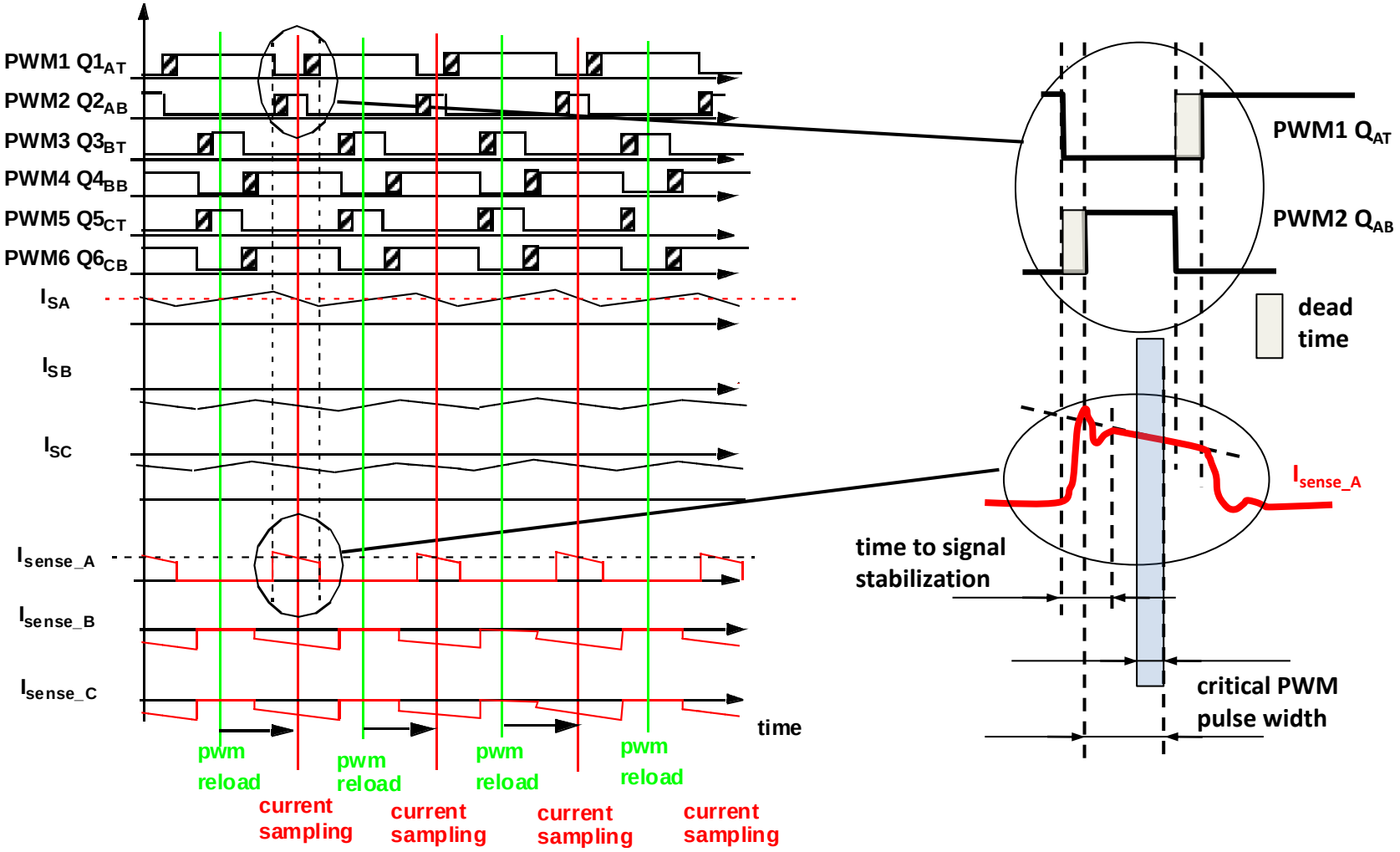
Risk of misaligned data access

PWM to ADC Synchronization

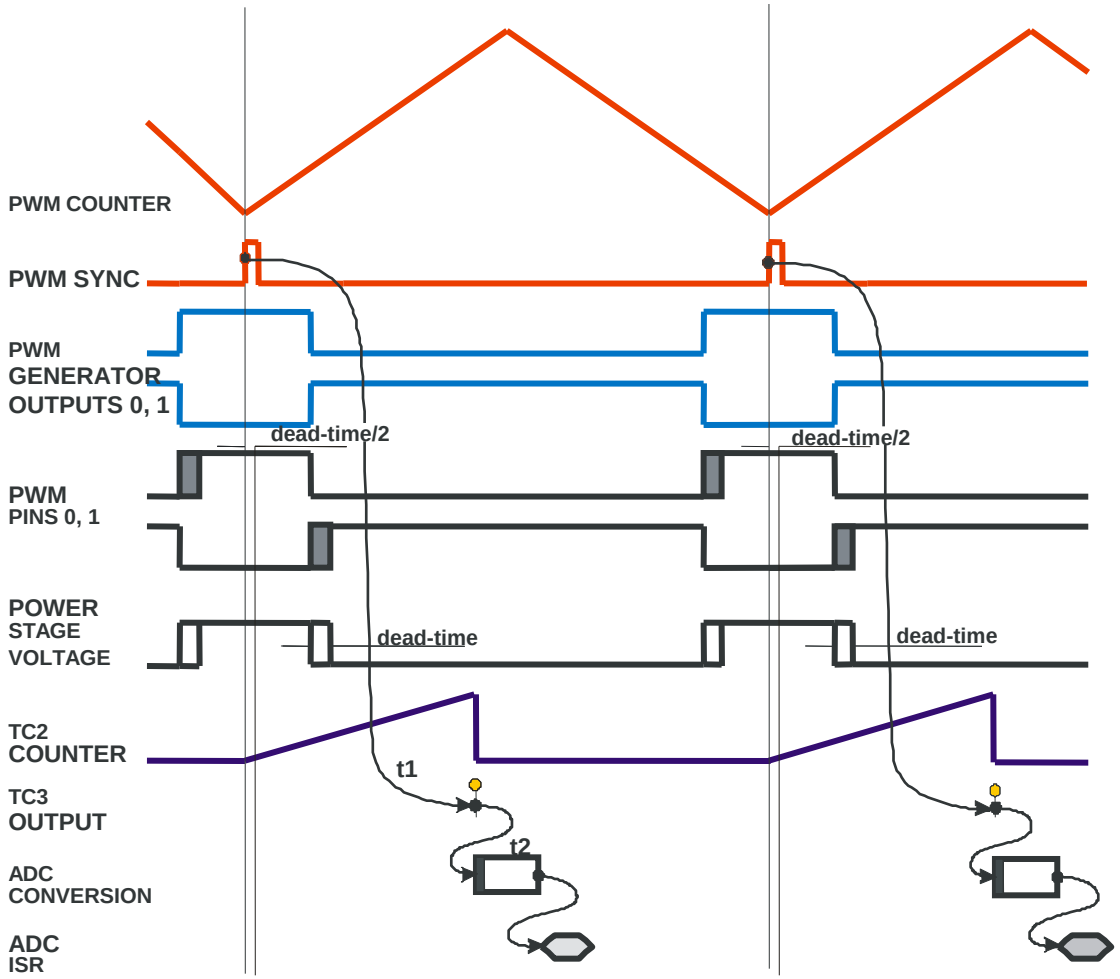
PWM to ADC Synchronization - Motivation



PWM to ADC Synchronization – Current Sensing



PWM to ADC Synchronization – Detailed Analysis



- Implementation on MC56F8025
- t1 - time between PWM SYNC signal and TC3 output.
- TC3 output starts ADC conversion immediately.
- t2 – ADC conversion time

PWM & ADC Synchronization - Create a Project

1) Create new empty project based on the Quick Start stationery
   - Connect PC (Metrowerks CodeWarrior) and EVM (processor)
     - Two options – either parallel port to JTAG (parallel command convertor) or USB to JTAG (USB TAP interface)
       - Parallel port to JTAG (parallel command convertor) – Target Settings (SDM_pFlash Settings) -> Debugger -> Remote Debugging -> 56800E Local Hardware Connection
       - USB to JTAG (USB TAP interface) - Target Settings (SDM_pFlash Settings) -> Debugger -> Remote Debugging -> 56800E Local USBTAP Connection
   - Compile, link and download the new project – to check if the project can be compiled, linked and downloaded without errors

FreeMaster Communication Set-Up

2) FreeMaster communication set-up – two possible communication channels - either RS232 (SCI peripheral) or JTAG

- Let's use RS232 (SCI peripheral)
  - Open GCT (Graphical Configuration Tool)
    - Select – "FreeMaster Software Driver" item
      - » Communication Interface - select "SCI_0" from the drop down list box
      - » Interrupt Processing options – select "Polling Mode"
      - » Enable Scope Feature – tick the tick box
      - » Enable Recorder Feature – tick the tick box
      - » Save settings
    - Select – "SCI_0 – Queued Serial Communication Interface" item
      - » Set "Divisor" – number 208 relates to baud rate equal to 9615 bps
      - » "Enable Receiver" – tick the tick box
      - » "enable Transmitter" – tick the tick box

FreeMaster Communication Set-Up cont'd

2) FreeMaster communication set-up – two possible communication channels - either RS232 (SCI peripheral) or JTAG
   - Let's use RS232 (SCI peripheral) cont'd
     - Continue GCT setting
       - Select – "GPIO_B –General Purp[ose I/O" item
         - » Configure pin 6 to RxD0 mode
         - » Configure pin 7 to TxD0 mode
       - Select – "SYS – System Support Control" item
         - » Enable clock to SCI_0 module – Peripheral Clock Enable (PCE) -> tick the SCI_0 tick box
       - Save GCT settings – "Save" icon or "Save & Exit" icon
         - » Processor setting is stored into "appconfing.h" file

FreeMaster Communication Set-Up cont'd

2) FreeMaster communication set-up – two possible communication channels - either RS232 (SCI peripheral) or JTAG
   - Let's use RS232 (SCI peripheral) cont'd
     - Application source code modification
       - Type – ioctl(GPIO_B, GPIO_INIT, NULL); - GPIO B port initialization according to the GCT setting
       - Type – ioctl(SCI_0, SCI_INIT, NULL); - SCI 0 peripheral module initialization according to the GCT setting
       - Type - FMSTR_Init(); - FreeMaster driver initialization according to the GCT setting
       - Type - FMSTR_Poll(); - somewhere in the endless loop; must be called periodically
       - Compile, link and download to the processor
       - Run the application

FreeMaster Communication Set-Up cont'd

2) FreeMaster communication set-up – two possible communication channels - either RS232 (SCI peripheral) or JTAG
   - Let's use RS232 (SCI peripheral) cont'd
     - Open FreeMaster tool – PC side application
       - Save empty FreeMaster project to the directory where new *.mcp project is stored – create new *.pmp project
       - Select – Project -> Options -> Comm -> Communication -> select "Direct RS232", Port "COM1" and Speed and tick the tick box "Save settings to project file"
       - Now the communication is established
       - Select – Project -> Options -> MAP Files -> Default symbol file and select corresponding *.elf file; then choose File format – Binary ELF with DWARF1 or DWARF2 dbg format
       - Now we can monitor and edit global variables from the C project

PWM Module Setting

3) Assumptions

- PWM frequency – 20 kHz
- Complementary mode
- Center aligned
- Deadtime 0 – 1us
- Deadtime 1 – 1us
- Positive polarity for top and bottom outputs
- Fault disabled
- PWM reload frequency – every PWM opportunity

PWM Module Setting cont'd

3) PWM

- GCT
  - PWM - the peripheral setting according to the assumptions
    - General Settings
      - » Select PWM Module Enable
      - » Load OK
  - SYS
    - Select PWM in Peripheral Clock Enable (PCE)
  - GPIO_A
    - Change functionality of Pin 0 up to Pin 5 from GPIO to PWM
- Source code
  - Type - ioctl(GPIO_A, GPIO_INIT, NULL);- GPIO A initialization according to GCT setting
  - Type - ioctl(PWM, PWM_INIT, NULL);- PWM module initialization according to GCT setting
  - Define variable – pwm_sComplmentaryValues sPwmCompl;
  - Initialize sPwmCompl variable – safety feature
  - Call the ioctl command - ioctl(PWM, PWM_UPDATE_VALUE_REGS_COMPL, &sPwmCompl); - periodically
  - Enable PWM output pads - ioctl(PWM, PWM_OUTPUT_PAD, PWM_ENABLE);

PWM Module Setting cont'd

3) PWM

- FreeMaster – monitor and edit the variables below
  - sPwmCompl.pwmChannel_0_Value
  - sPwmCompl.pwmChannel_2_Value
  - sPwmCompl.pwmChannel_4_Value

Quad Timer Module Setting

4) Quad Timer

- The PWM_reload_sync signal can be connected to the Timer's Channel 3 input and the Timer's Channels 2 and 3 outputs are connected to the ADC sync inputs
- SYNC0 – Timer Channel 3
- SYNC1 – Timer Channel 2
- GCT
  - QT_A3 - Quad Timer Channel3 settings
    - Count mode -Triggered mode, edge of secondary source triggers primary count till compare
    - Primary source – Prescale (IPB clock/1)
    - Secondary – Counter #3 input pin
    - Counter length – Count until compare and reinitialize
    - Disable tick box – Timer Channel Enabled
    - Output mode – Set on compare, cleared secondary source input edge
    - Define Compare value 1 – defines ADC sample&hold instant and conversion initialization
    - OFLAG / Timer Pin Control – select Output enable (OFLAG to pin) – to see SYNC0 signal on the output pin; debug reason

Quad Timer Module Setting cont'd

4) Quad Timer

- GCT
  - SYS
    - Peripheral Clock Enable (PCE) for TMR A3
    - DAC and Timer Internal Peripheral Source Select (IPS) -> Timer A3 and form drop down list box select - PWM reload sync. signal
  - GPIO_B
    - Pin 3 – select TA3
- Source code
  - Type - ioctl(GPIO_B, GPIO_INIT, NULL);
  - Type - ioctl(QTIMER_A3, QT_INIT, NULL);

ADC Module Setting

5) ADC

- GCT
  - ADC
    - ANA and ANB Operation
      - » Start-up mode – Normal
      - » Triggered source – SYNC0 input
      - » Power control – disable all options
      - » Interrupts – select ANA Conversion complete, define ISR name and priority
  - SYS
    - Peripheral Clock Enable (PCE) – select ADC
  - Select INTC

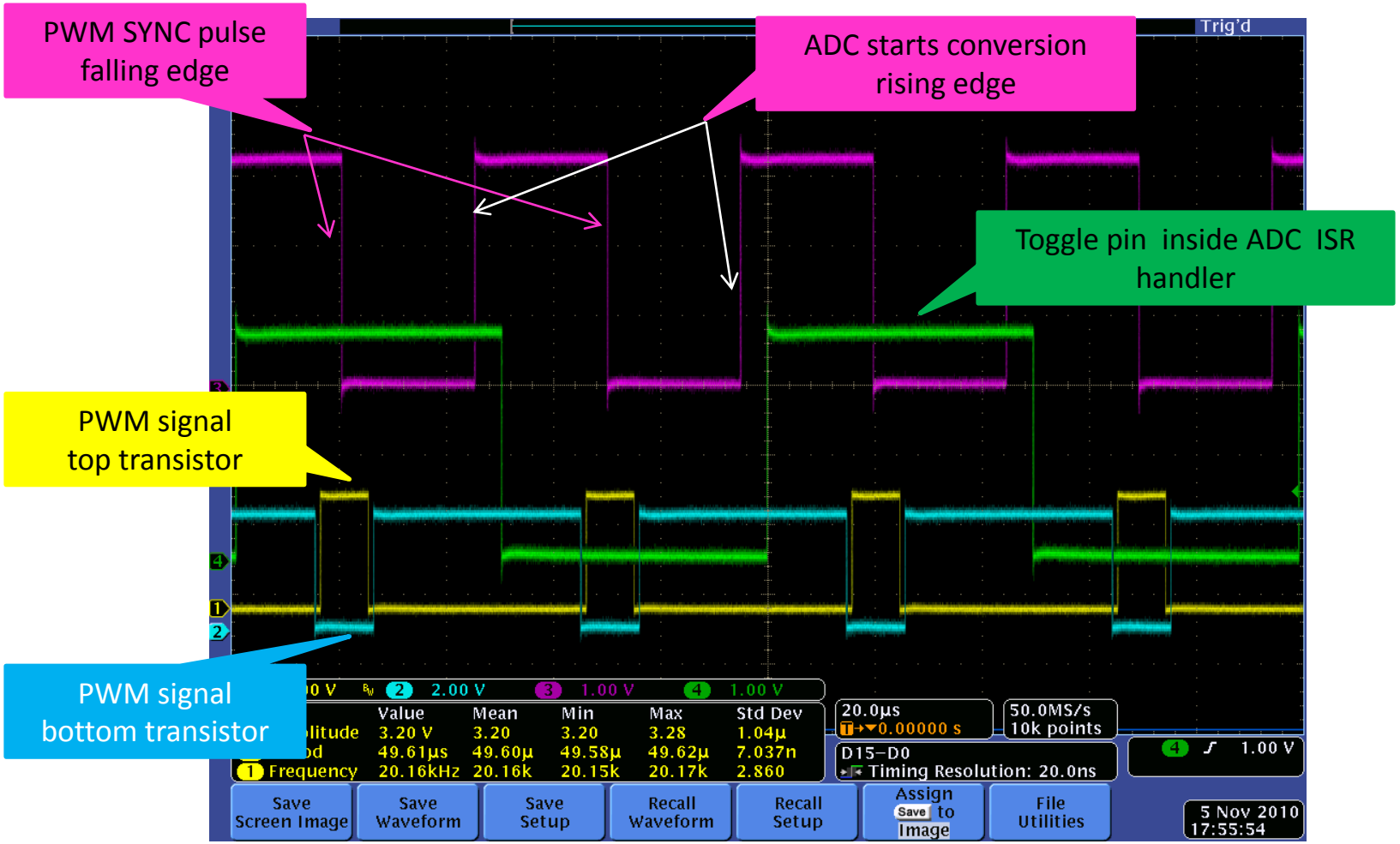
- Source code
  - Type - ioctl(ADC, ADC_INIT, NULL);
  - Define ISR function – void ?????(void)
    - Clear EOSI flag - ioctl(ADC, ADC_CLEAR_STATUS_EOSI, NULL);

Controller Board with MC56F8025

- Pins connection
  - PWM
    - PWM0/PA0
      - DSC – pin 40
      - Controller board header J10 – pin 9
    - PWM1/PA1
      - DSC – pin 39
      - Controller board header J10 – pin 11
    - PWM2/PA2
      - DSC – pin 32
      - Controller board header J10 – pin 30
    - PWM3/PA3
      - DSC – pin 33
      - Controller board header J10 – pin 32
    - PWM4/PA4
      - DSC – pin 31
      - Controller board header J10 – pin 34
    - PWM5/PA5
      - DSC – pin 27
      - Controller board header J10 – pin 36

- Pins connection
  - Quad Timer Channel 3
    - TA3/PB3
      - DSC – pin 22
      - Controller board header J10 – pin 17
  - ADC ISR check – GPIO B0 toggle
    - PB0
      - DSC – pin 30
      - Controller board header J10 – pin 28

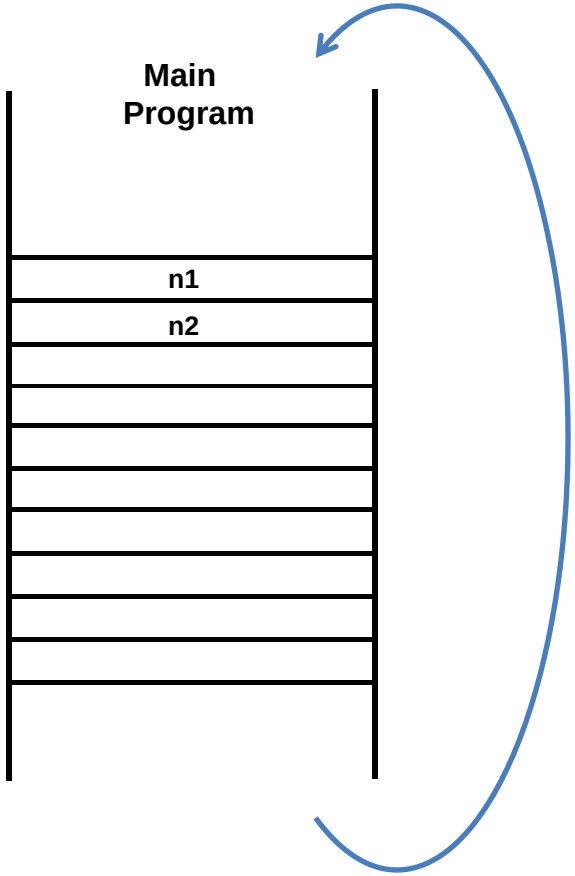
PWM & ADC Synchronization Result



Interrupt Processing

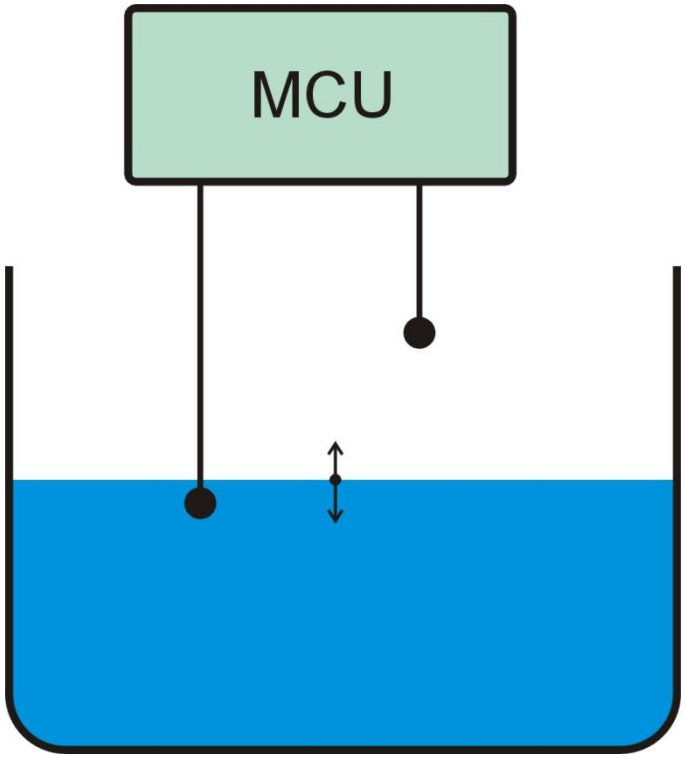
Why Do We Need Interrupts?

- The MCU executes instructions (n1, n2,…) step by step in never-ending loop (sequential machine)
- If no interrupts the main program serves all external event and exceptions



Why Do We Need Interrupts? – Example 1

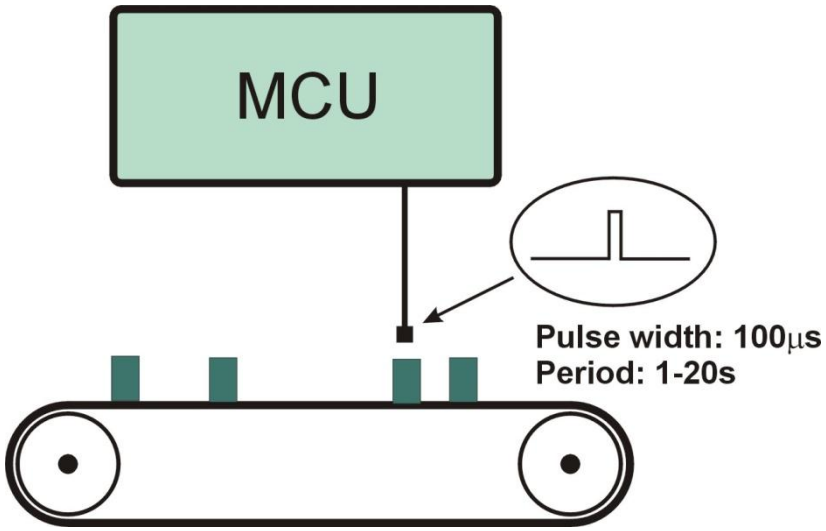
- Water level is changing slowly
- The main program checks the sensors every e.g. 1s or 10s
- This can be achieved **very easily** in the main program
- No significant load for MCU



Water level sensing

Why Do We Need Interrupts? – Example 2

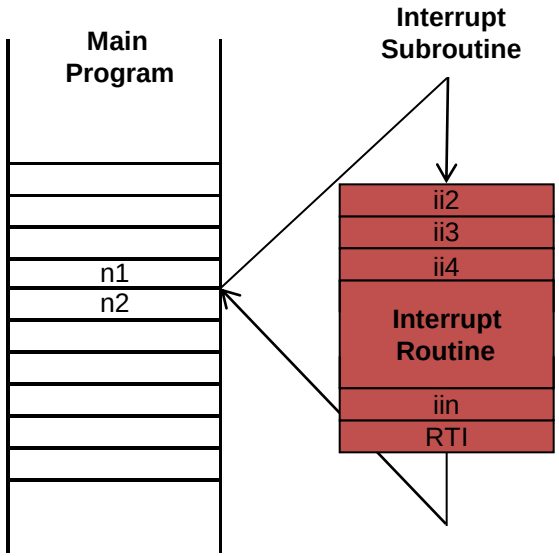
- Frequency of the pulses not too high, but pulse with is very slow
- To ensure correct pulses counting the sensor **must be checked faster than 100 μs**
- The main program spends **significant time** on this simple task
- **Difficult to program**
- How to manage it more effectively?



Product Counting

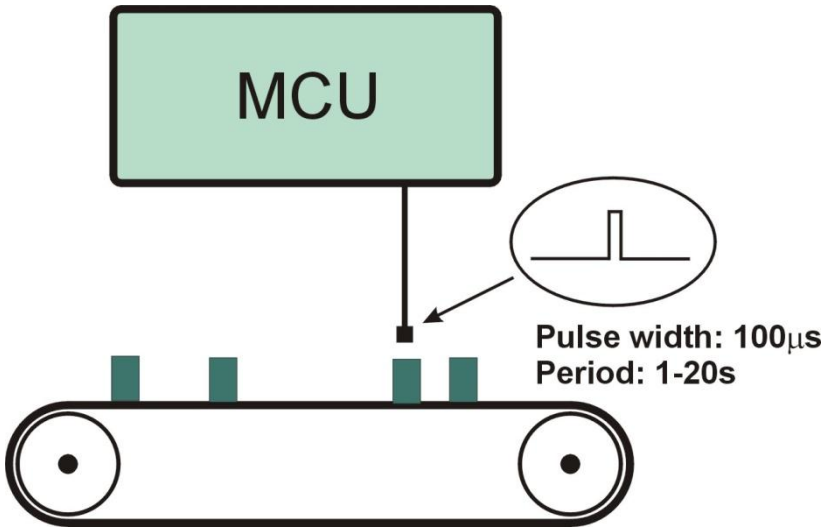
What Is the Interrupt?

- The main program execution can be interrupted by some event
- On the event the MCU core **stops main program** execution and **starts interrupt routine** execution
- Once the interrupt routine finished, the MCU core **continue in main program** execution



Usage of Interrupt Example

- The interrupt routine is executed every pulse from the sensor
- The interrupt routine counts pulses
- The interrupt routine is **very short** and executed **once per pulse** only
- The main program is **pulse counting independent**
- **Very simple to program**



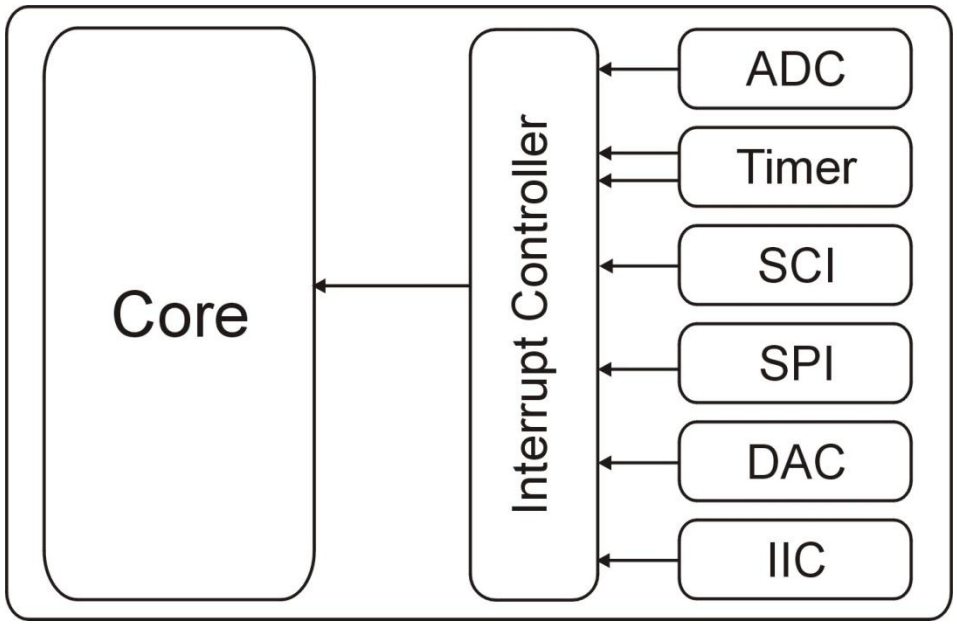
Products Counting

Interrupt Sources

- External Hardware Interrupt Sources
  - Peripherals (ADC Conversion Complete, Timer Overflow, Timer Input Capture, SCI Char Received, etc.)
  - Interrupt Request Signals (IRQ pins)
- Internal Hardware Interrupt Sources (within core)
  - Illegal instruction interrupts
  - Hardware stack overflow interrupts
  - Misaligned data access interrupts
  - Debugging (Enhanced OnCE) interrupts
- Software Interrupt Sources
  - Interrupt Instructions (SWI)

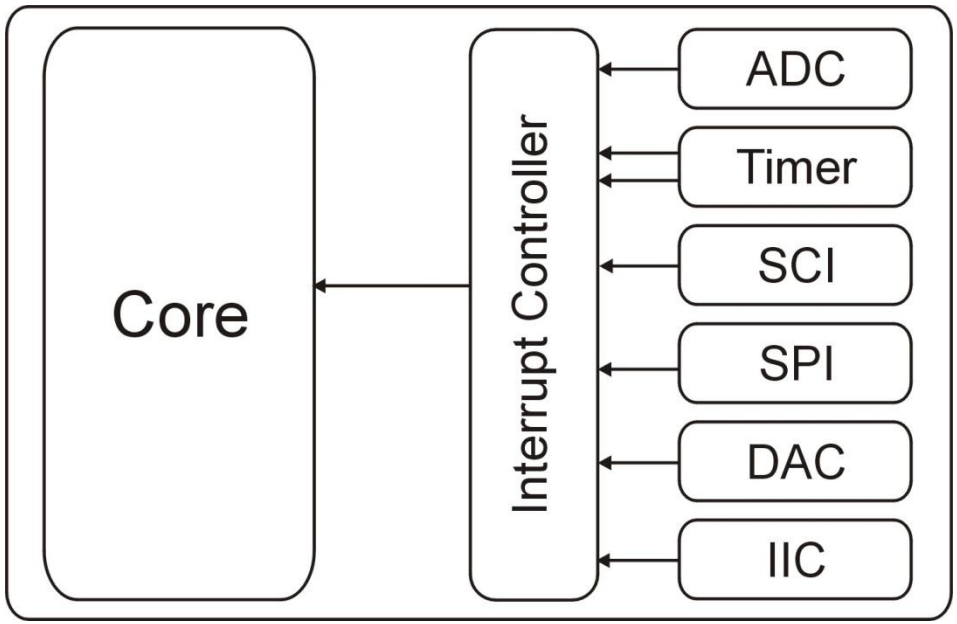
Interrupt Controller

- The MCU core has limited number interrupt inputs (1, 2, 4, 8, 16)
- The **Interrupt Controller** connects more interrupt sources into MCU core



Interrupt Controller

- The interrupt controller solves also **priority** of interrupt sources, if there is more interrupt requests at the same time
- The priority can be **fixed** or **user defined**



Interrupt Vectors Table

- The interrupt vectors table is **part of the memory**, where interrupt routines **addresses** are stored
- Each interrupt source or group of sources has **own position** in this table (interrupt vector)
- Position in this table usually defines **priority** of interrupt source

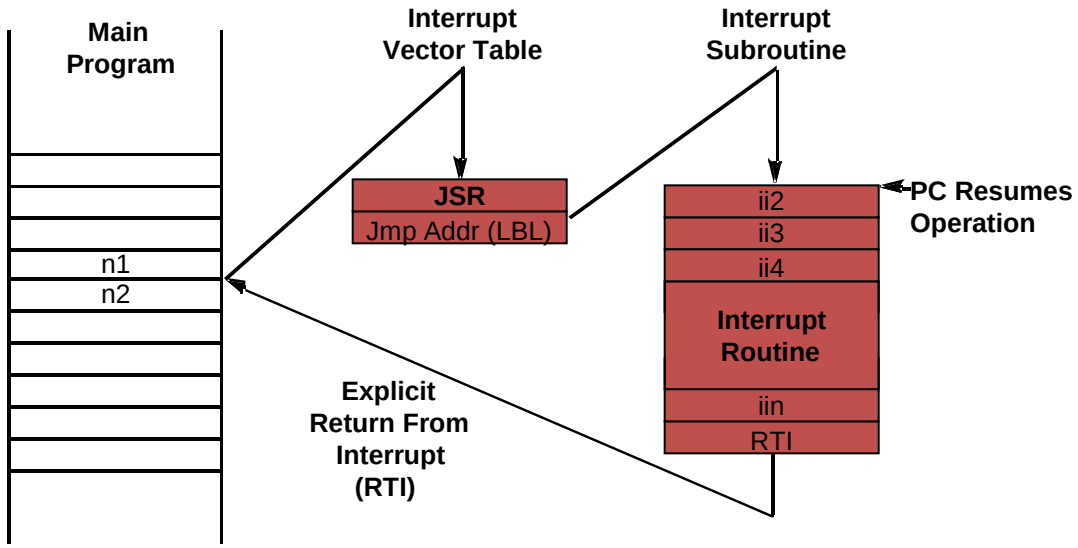
Vector Priority	Address (High/Low)	Vector Name	Module	Source	Enable	Description
Lower ↑	$FFC0/FFC1 through $FFCA/FFCB	Unused Vector Space (available for user program)				
	$FFCC/FFCD	Vrti	System control	RTIF	RTIE	Real-time interrupt
	$FFCE/FFCF	Viic	IIC	IICIS	IICIE	IIC control
	$FFD0/FFD1	Vatd	ATD	COCO	AIEN	AD conversion complete
	$FFD2/FFD3	Vkeyboard	KBI	KBF	KBIE	Keyboard pins
	$FFD4/FFD5	Vsci2tx	SCI2	TDRE TC	TIE TCIE	SCI2 transmit
	$FFD6/FFD7	Vsci2rx	SCI2	IDLE RDRF	ILIE RIE	SCI2 receive
	$FFD8/FFD9	Vsci2err	SCI2	OR NF FE PF	ORIE NFIE FEIE PFIE	SCI2 error

Interrupt Processing #1

1. When interrupt event occurs, the interrupt controller evaluates interrupt request
2. The interrupt request with highest priority continues into MCU core
3. If the interrupt are enabled, the interrupt request is accepted the MCU core

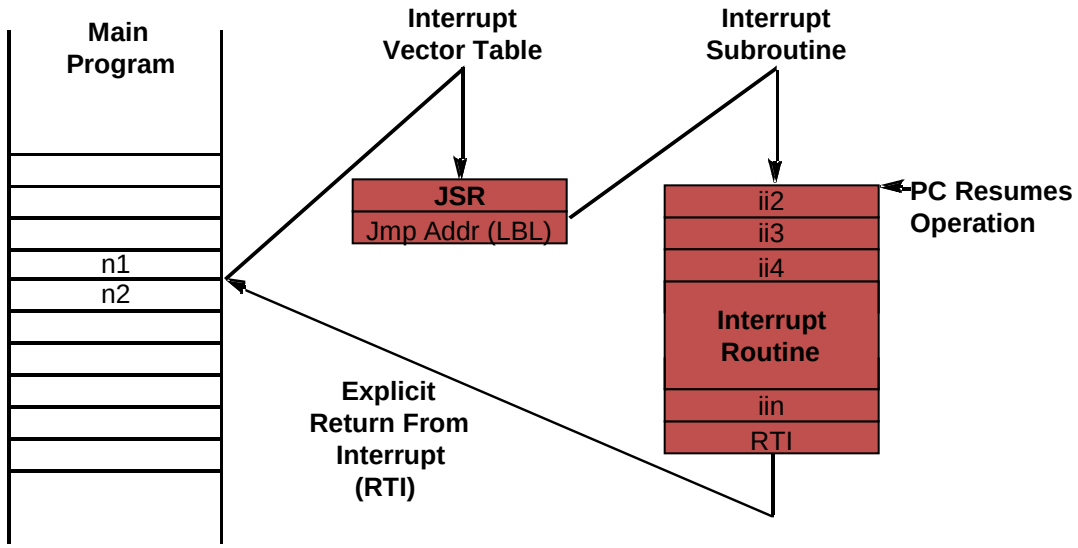
Interrupt Processing #2

4. The MCU core completes currently executed instruction
5. The MCU Core jumps to address defined in the interrupt vector table
6. The actual PC and status registers are save on the stack
7. The MCU core starts to execute interrupt routine code



Interrupt Processing #3

8. At the and of interrupt routine the MCU core restores the PC and status registers
9. The MCU core continues to execute main program



DSC56800E
Interrupt Processing

Core Interrupt Priority Structure

- The DSC56F800E core supports 5 interrupt priority levels
- The current core interrupt priority level (CCPL) is defined by I0, I1 bits in SR register (Status register)

IPL	Description	Priority	Interrupt Sources
LP	Maskable	Lowest	SWILP Instruction
0	Maskable	*	On-chip peripherals, IRQA and IRQB, SWI #0 Instruction
1	Maskable	*	On-chip peripherals, IRQA and IRQB, SWI #1 Instruction
2	Maskable	*	On-chip peripherals, IRQA and IRQB, SWI #2 Instruction
3	Non-maskable	Highest	Illegal instruction, hardware stack overflow, SWI instruction, EOnCE Interrupts, misaligned data access

Core Interrupt Priority Structure

- The new interrupt is accepted if its interrupt priority level (IPL) is higher than current core interrupt priority level (CCPL)

I1	I0	CCPL	Exceptions Accepted	Exceptions Masked	Comments
0	0	0	IPL 0,1,2,3, and SWILP	None	All interrupts are allowed including the SWILP
0	1	1	IPL 1,2,3	IPL 0 and SWILP	All non-maskable interrupts and maskable interrupts that are programmed at level 1 or 2 are allowed
1	0	2	IPL 2,3	IPL 0 , 1 and SWILP	All non-maskable interrupts and maskable interrupts that are programmed at level 1 are allowed
1	1	3	IPL 3	IPL 0, 1, 2 and SWILP	Only non-maskable interrupts are allowed

Interrupt Controller

- The Interrupt Controller can service 64 interrupt requests (IRQ)
- Each IRQ has user defined priority (disabled, 0, 1, 2)
  - Interrupt Priority Registers (IPR0 – IPR6)
- If more IRQ assigned to one level, the position in interrupt vector table defines priorities in given level (0 – highest priority, 63 lowest priority
- The Interrupt Controller also supports two Fast Interrupts
  - Fast Interrupt 0 Vector Address Registers (FIVAL0, (FIVAH0)
  - Fast Interrupt 1 Vector Address Registers (FIVAL1, (FIVAH1)
  - Fast Interrupt Match 0 Register (FIM0)
  - Fast Interrupt Match 1 Register (FIM1)

Interrupt Control in Peripherals

- Every interrupt source has **one control** and **one status bit** (interrupt flag)
- The control bit **enable/disable** interrupt
- The status bit shows **actual state** of interrupt request. This bit is set, when interrupt event occurs
- User has to **clear** the **status bit** during interrupt routine execution
- Every interrupt source has own **interrupt vector** or shares interrupt vector with more sources in peripheral

Base + $7	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Read	TCF	TCFIE	TOF	TOFIE	IEF	IEFIE	IPS	INPUT	CAPTURE MODE		MSTR	EEOF	VAL	0	OPS	OEN
Write														FORCE		
Reset	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

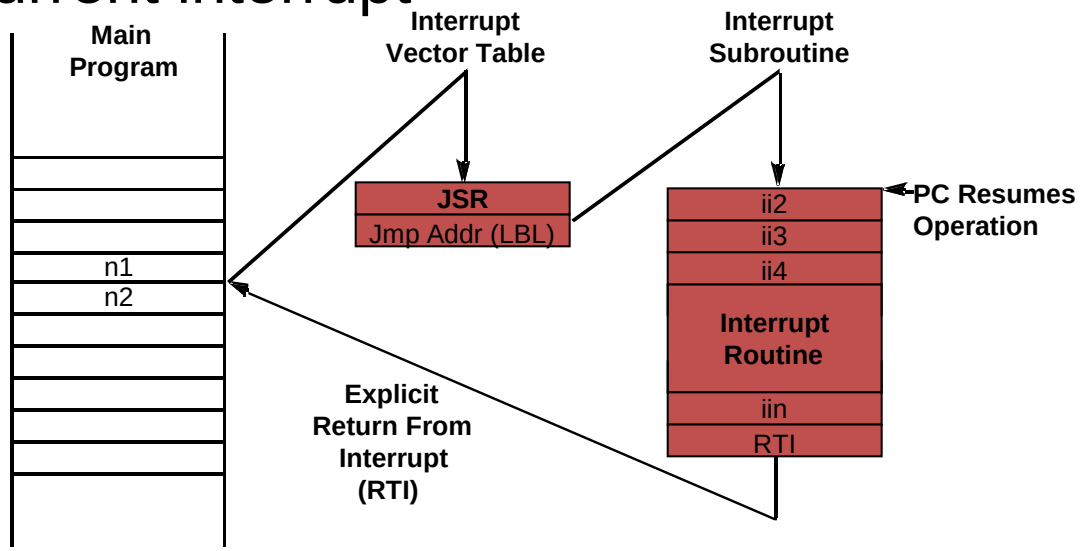
Figure 13-16. Status and Control (SCTRL) Registers

Control bit

Status bit

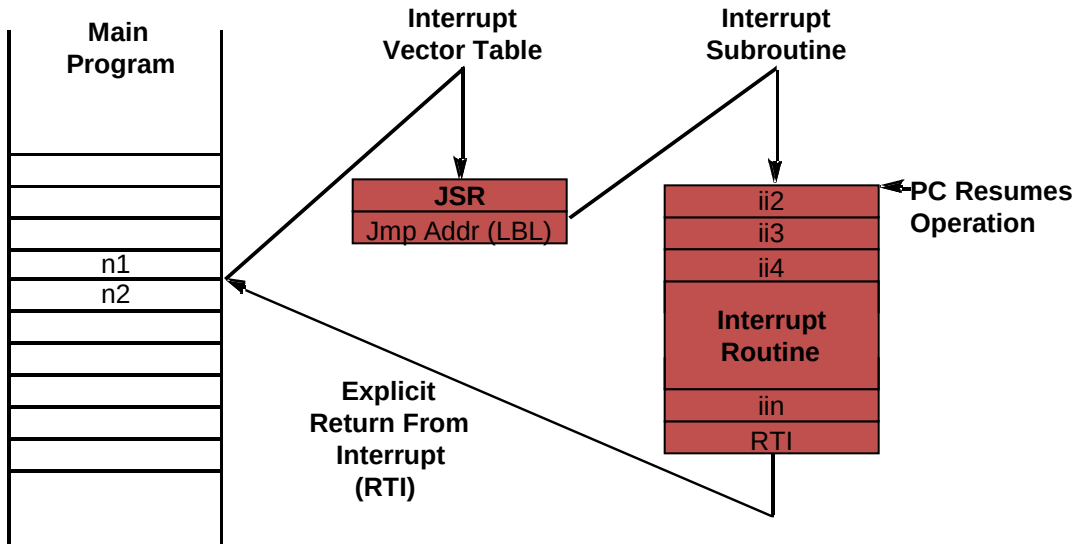
Standard Interrupt Processing #1

1. The currently executing instruction is allowed to complete, and all subsequent instructions are flushed from the pipeline
2. The program counter is frozen
3. The CCPL is raised to be one higher than the level of the current interrupt



Standard Interrupt Processing #2

3. The CCPL is raised to be one higher than the level of the current interrupt
4. The program controller fetches the JSR instruction that is located at the vector for this interrupt, and then it unfreezes the PC

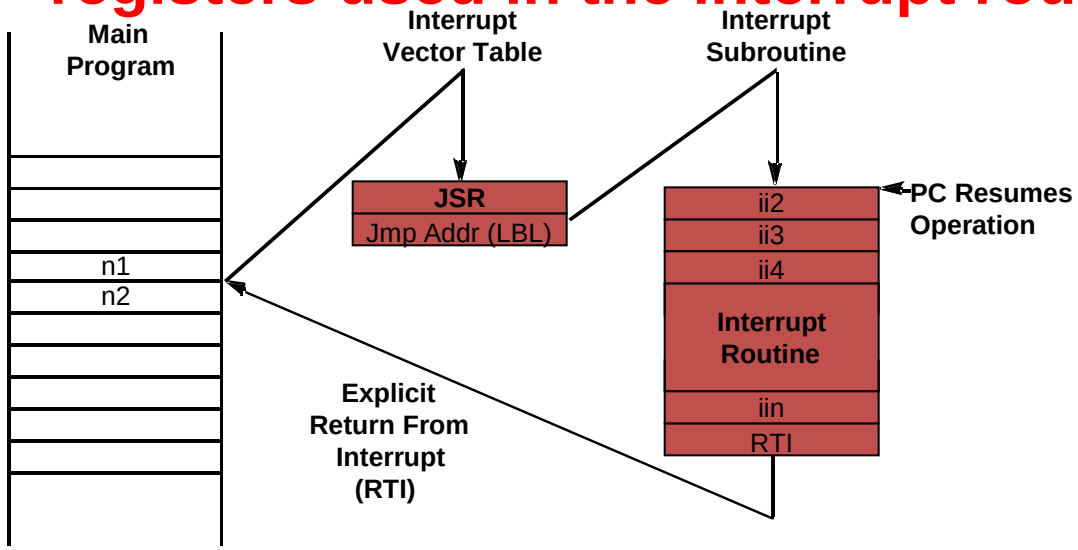


Standard Interrupt Processing #3

5. The JSR instruction is executed, saving the original program counter and status register on the software stack

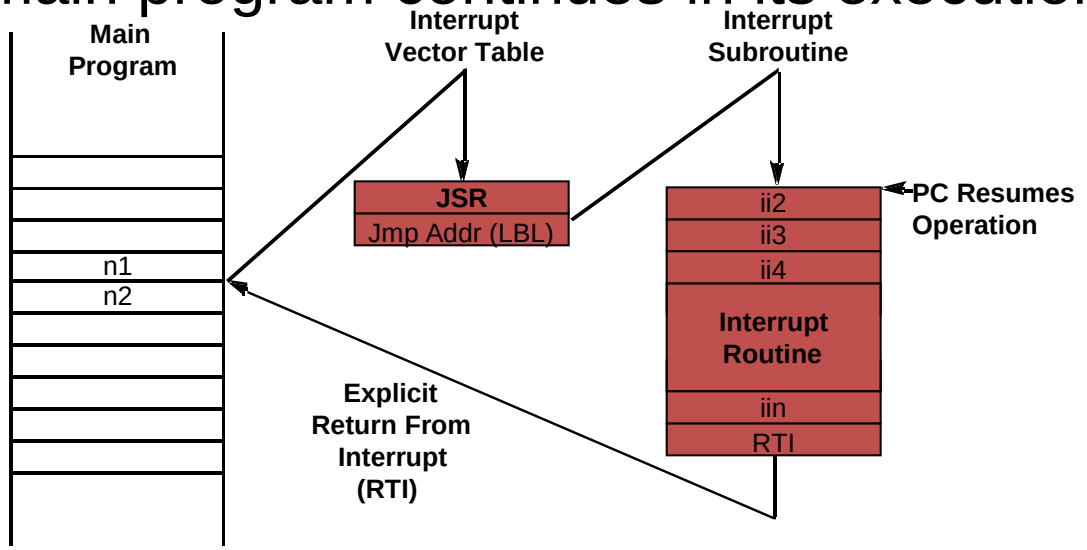
!!! CAUTION !!!

The user is responsible to save and restore all registers used in the interrupt routine



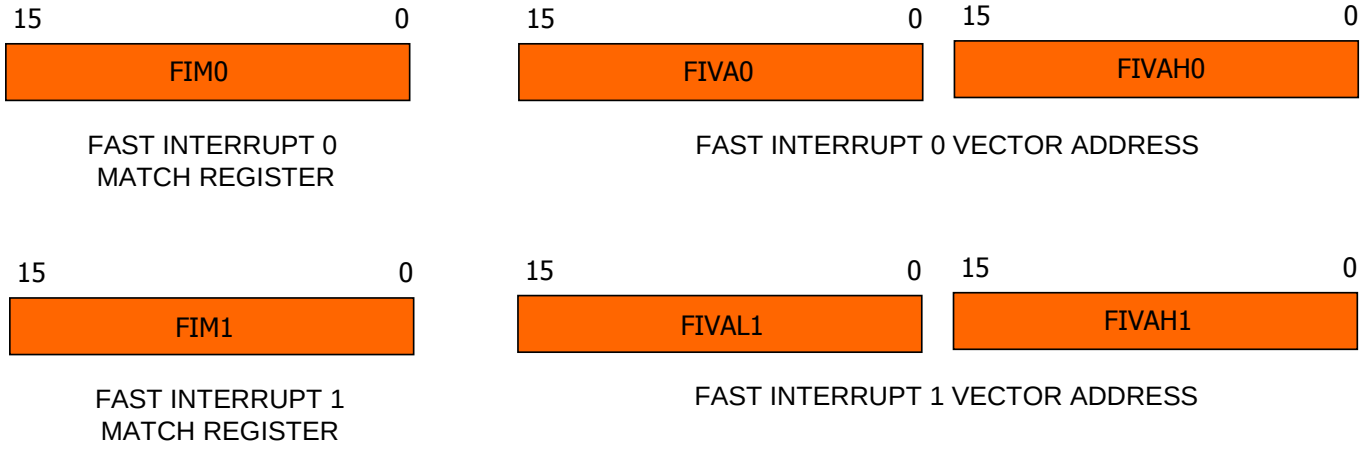
Standard Interrupt Processing #4

6. At the end of interrupt routine, the RTI restore program counter and status register to its original value
7. The CCPL is decreased by one to the original level of the interrupt
8. The main program continues in its execution



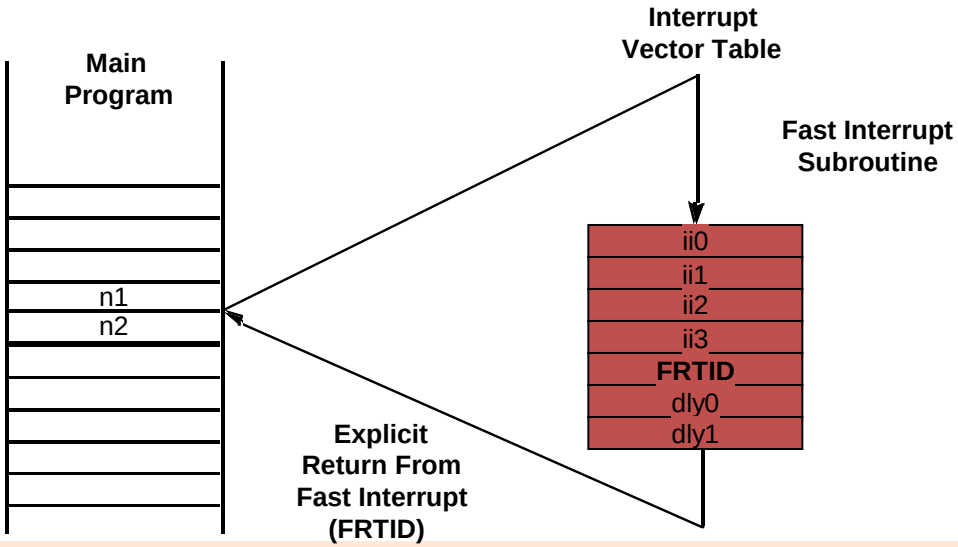
Fast Interrupt Processing #1

- The fast interrupt must be set to priority level 2
- The fast interrupt does not use interrupt vector table The number of interrupt vector has to be set to FIM0, FIM1 registers
- The interrupt routine address must be set in FIVAL0,FIVAH0 or FIVAL1, FIVAH1 registers



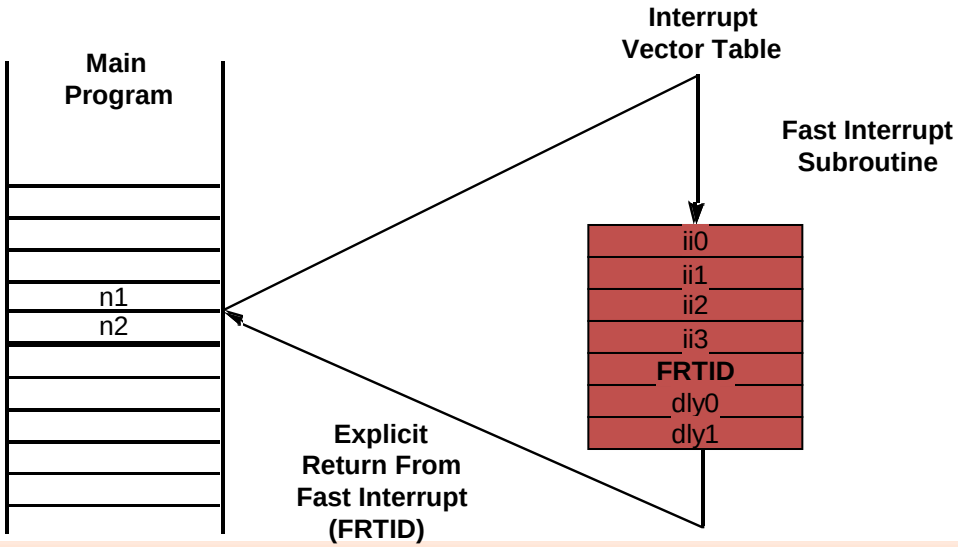
Fast Interrupt Processing #2

1. The frozen program counter (return address) is copied to the fast interrupt return address register (FIRA)
2. The status register (with the exception of the P4–P0 bits) and the NL bit in the operating mode register are copied to the fast interrupt status register (FISR)
3. The stack pointer (SP) is aligned for long-word accesses



Fast Interrupt Processing #3

4. The frozen program counter (return address) is copied to the fast interrupt return address register (FIRA)
5. The Y register is pushed onto the stack, and the stack pointer is advanced to an empty 32-bit location
6. The R0, R1, N, and M01 registers are swapped with their shadows

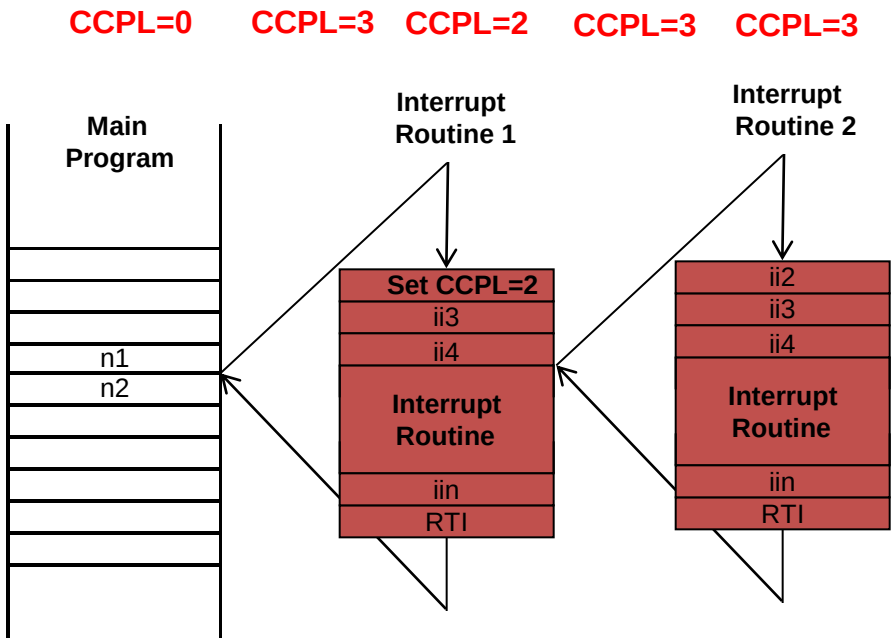


Nested Interrupts

- **Nested interrupt** = interrupt request comes when **another interrupt** is **executed**
- If the application uses maximal **three** interrupts, the interrupt nesting is **automatically managed** by the DSC core (each priority level has assigned one interrupt source)
- If **more** interrupt sources are assigned to the one interrupt priority level, the nesting **has to be managed** by **user** (user has to set CCPL (I1,I0 bits is SR) to the proper level in interrupt routine, which can be interrupted)

Nested Interrupts - Example

- Let's have two interrupt routines in level 2
- The routine 1 can be interrupted by another interrupt routine level 2
- The routine 2 is uninterruptable



Programming Interrupts

- The interrupt routine is routine, which **ends** by
  - **RTI** for standard interrupt
  - **RTFID** for fast interrupt
- If the interrupt routine **uses** any **core register**, the **user** is **responsible** to **save and restore** the used core registers
- If the interrupt routine calls another **subroutine** also this subroutine has to **save and restore** used core registers

Interrupt does’nt work?

- If your interrupt doesn’t work, please check if
  1. The interrupt source is enabled in the peripheral
  2. The interrupt has defined priority
  3. The interrupt routine uses #pragma interrupt
  4. The interrupts are enabled (archEnableInt())
  5. The interrupt flag is cleared in the interrupt routine

C Compiler Support for Interrupts

- The Metrowerks C compiler supports the interrupt routines by following #pragma:
  #pragma interrupt [<options>] [<mode>] [on|off|reset]
- <options>
  - alignsp
  - comr
- <mode>
  - saveall
  - fast
  - called

C Compiler Support for Interrupts

- **`#pragma interrupt`**
  - This pragma preserves all registers used in the routine
  - The routine ends by RTI instruction
- `#pragma interrupt saveall`
  - This pragma preserves all core registers
  - The routine ends by RTI instruction
- **`#pragma interrupt called`**
  - This pragma preserves all registers used in the routine
  - The routine ends by RTS instruction
  - This pragma should be used for every subroutine called from interrupt

C Compiler Support for Interrupts

- **`#pragma interrupt fast`**
  - This pragma preserves all registers used in the routine
  - The routine ends by RTFID instruction
- Option **alignsp**
  - Aligns the stack pointer register correctly to allow long values to be pushed on to the stack

C Compiler Support for Interrupts

- Option **comr**
  - The Operating Mode Register (OMR) is set to:
    - 36-bit values used for condition codes (CM bit cleared)
    - Convergent Rounding (R bit cleared)
    - No Saturation mode (SA bit cleared)
    - Instructions fetched from P memory (XP bit cleared)

QuickStart Support for Interrupts

- **`void archEnableInt(void);`**
  - This macro enables all interrupts by clearing bits I1 (Bit 9) and I0 (Bit 8) in the SR register
- **`void archDisableInt(void);`**
  - This macro disables all maskable interrupts by setting bits I1 and I0 (Bits 9 - 8) in the SR register
- **`void archEnableIntLvl123(void);`**
  - This macro enables interrupts at levels 1, 2 and 3 while masking the interrupts at level 0. It clears bit I1 (Bit 9) and sets bit I0 (Bit8) in the SR register
- **`void archEnableIntLvl23(void);`**
  - This macro enables interrupts at levels 2 and 3 while masking interrupts at levels 0 and 1. It sets bit I1 (Bit 9) and clears I0 (Bit 8) in the SR register

Standard Interrupt – Example 1

- Configure the QTimer channel 3 to generate compare event every 62.5 μs
- At compare event the QTimer the output is set to 1
- At the beginning the interrupt routine the QTimer output is cleared to 0
- At the end of the interrupt routine set GPIO B0 to 1
- The GPIO B0 is periodically cleared to 0 in background
- Use **#pragma interrupt**
- Measure the enter time, leave time and complete execution time of the interrupt

Standard Interrupt – Example 1

- Open the empty project with FreeMaster
- Open Configuration tool
- OCCS Module settings
  - Unselect *COP Disable*
- SYS module settings
  - Enable Clocks to *TMR A3*
- Check off box at INTC module
- QT_A3 module settings
  - *Counting mode* = **Counting, count rising edges of primary source**

Standard Interrupt – Example 1

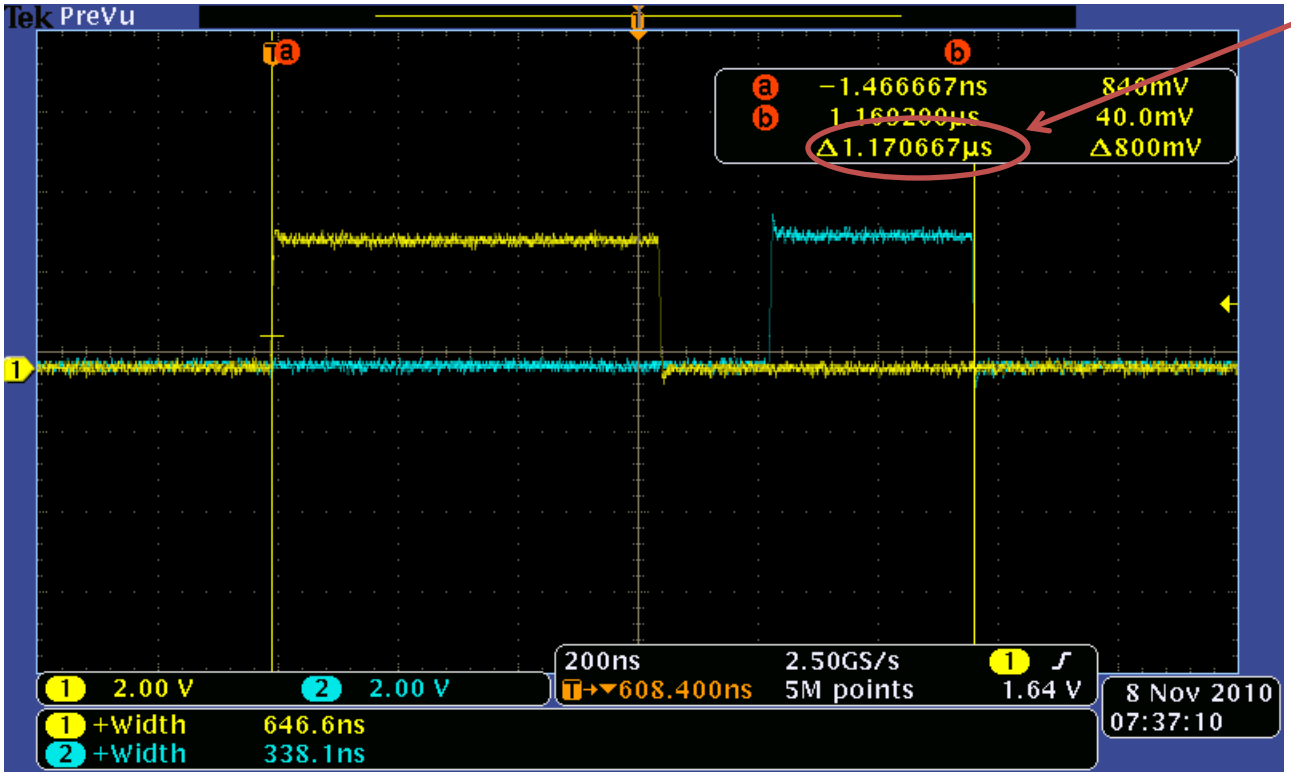
- QT_A3 module settings
  - *Counting mode* = **Counting, count rising edges of primary source**
  - *Primary source* = **Prescaler (IPB clock/1)**
  - *Output mode* = **Set OFLAG output on successful compare**
  - Check off **Output enable** (OFLAG on pin)
  - Set *Compare Value 1* to **2000**
  - Check off **Compare** interrupt
  - Set *Isr Name* to **IsrQT3Compare**
  - Set *Priority* to **2**

Standard Interrupt – Example 1

- GPIO_B module settings
  - PIN0
    - Set *Direction* to **Output**
  - PIN3
    - Set *Mode* to **TA3**
- Write the interrupt routine and main routine
- Compile the project and download it into evaluation board
- Run the code
- Measure again the enter time, leave time and complete execution time of the interrupt

Standard Interrupt – Example 1

- Results



Execution time

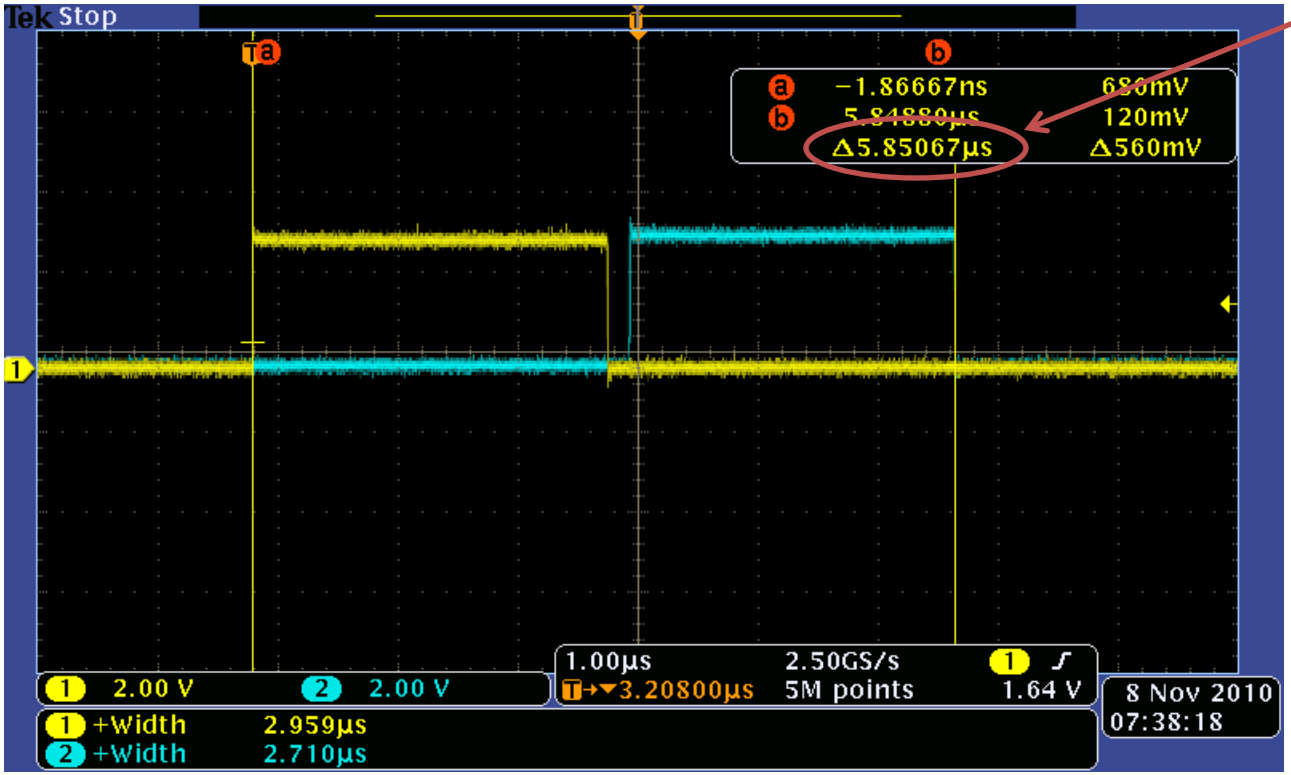
Yellow trace – Enter to the interrupt
Blue trace – Leave the interrupt

Standard Interrupt – Example 2

- Use the same example but with **#pragma interrupt saveall**
- Re-compile the project and download it into evaluation board
- Run the code
- Measure again the enter time, leave time and complete execution time of the interrupt

Standard Interrupt – Example 2

- Results



Execution time

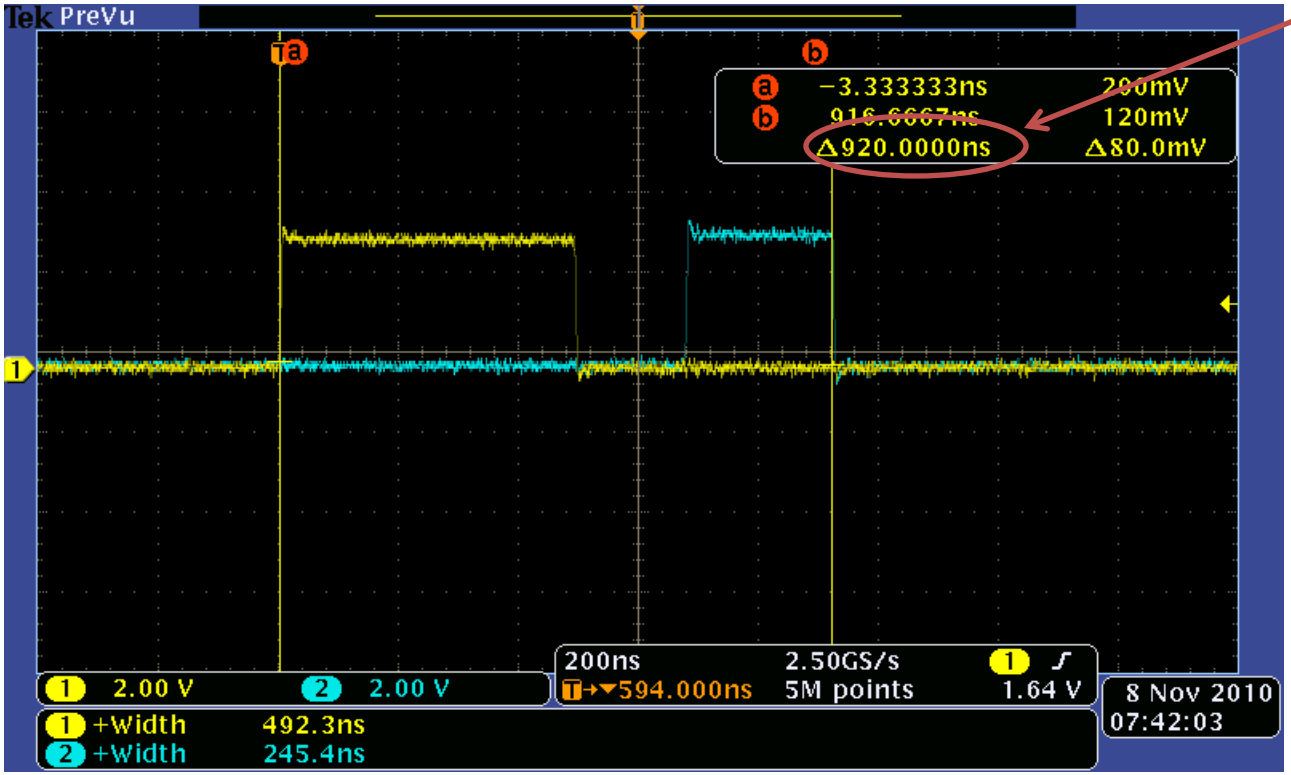
Yellow trace – Enter to the interrupt
Blue trace – Leave the interrupt

Fast Interrupt – Example

- Use previous example
- Change #pragma to **#pragma interrupt fast**
- ITCN settings
  - Set *Match Register* to **48**
  - Set *Fast ISR Name* to **IsrQT3Compare**
- Re-compile the project and download it into evaluation board
- Run the code
- Measure again the enter time, leave time and complete execution time of the interrupt

Fast Interrupt – Example

- Results



Execution time

Yellow trace – Enter to the interrupt
Blue trace – Leave the interrupt

Fast Interrupt – Example

- Interrupts Comparison

Type	Enter Time	Leave Time	Execution time
# pragma interrupt	647 ns	338 ns	1171 ns
#pragma interrupt saveall	2959 ns	2710 ns	5851 ns
#pragma interrupt fast	492 ns	245	920 ns

Thank you