Interrupt System

Introduction

This module is used to explain the interrupt system of the F2833x Digital Signal Controller.

So what is an interrupt?

Before we go into the technical terms, let us start with an analogy: Think of a nice evening and you are working at your desk, preparing the laboratory experiments for the next day. Suddenly the phone rings, you answer it and then you get back to work (after the interruption). The shorter the phone call, the better! Of course, if the call comes from your girlfriend you might have to re-think your next step due to the “priority” of the interruption… Anyway, sooner or later you will have to get back to the preparation of the task for the next day; otherwise you might not pass the next exam.

This analogy touches some basic definitions for interrupts;

- interrupts appear “suddenly”: in technical terms, this is called “asynchronous”
- interrupts might be more or less important: they have a “priority”
- they must be dealt with before the phone stops ringing: “immediately”
- the laboratory preparation should be continued after the call - the “interrupted task is resumed”
- the time spent to search the phone should be as small as possible – “interrupt latency”.
- after the call, you should continue your work from the exact place where you left it - “context save” and “context restore”

To summarize the technical terms:

Interrupts are defined as asynchronous events, generated by an external or internal hardware unit. An event causes the controller to interrupt the execution of the current program and to start a service routine, which is dedicated to this event. After the execution of this interrupt service routine, the program that was interrupted will be resumed.

The quicker a CPU performs this “task-switch”, the more this controller is suited for real-time control. After going through this chapter, you will be able to understand the F2833x interrupt system.

At the end of this chapter, we will perform an exercise with a program controlled by interrupts that uses one of the 3 core timers of the CPU. The core timer’s period interrupt will be used to perform a periodic task.

Module Topics

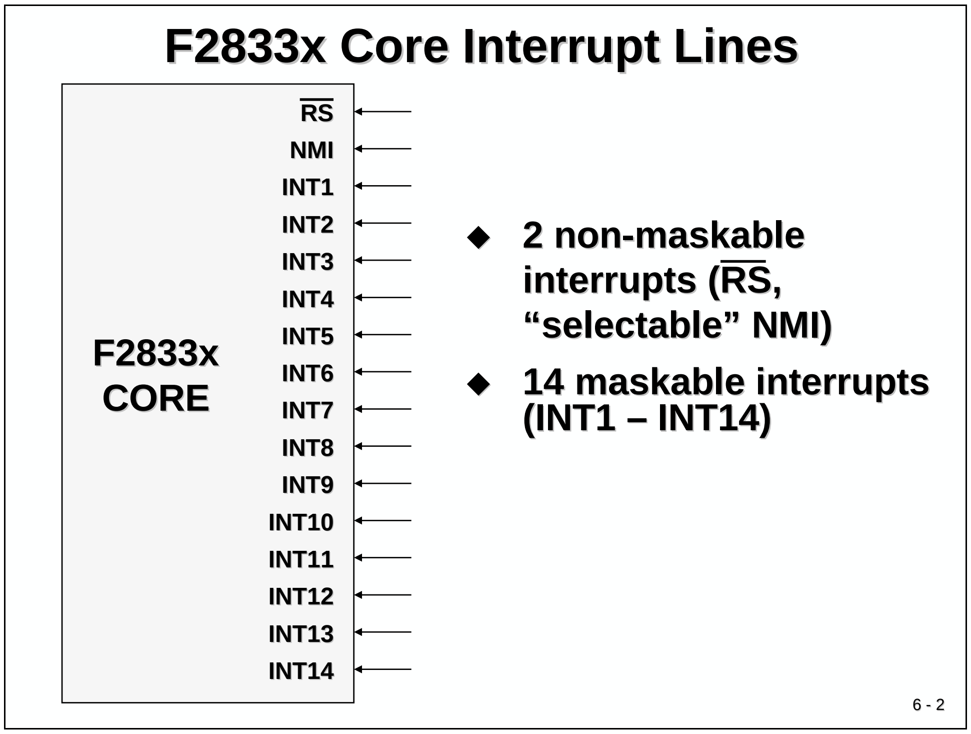
F2833x Core Interrupt Lines

The core interrupt system of the F2833x consists of 16 interrupt lines; two of them are called "Non-Maskable" (RESET, NMI). The other 14 lines are 'maskable' - this means the programmer can allow or disable interrupts from these 14 lines.

What does the phrase "mask" stand for?

A "mask" is a binary combination of '1' and '0'. A '1' stands for an enabled interrupt line, a '0' for a disabled one. By loading the mask into register "IER" we can select, which interrupt lines will be enabled to request an interrupt service from the CPU.

For a "non-maskable" interrupt, we cannot disable an interrupt request. Once the signal line goes active, the running program will be suspended and the dedicated interrupt service routine will start. Generally, "non-maskable" interrupts are used for high priority and safety based events e.g. emergency stop.



F2833x Core Interrupt Lines

- 2 non-maskable interrupts ($\overline{RS}$, "selectable" NMI)
- 14 maskable interrupts (INT1 – INT14)

6 - 2

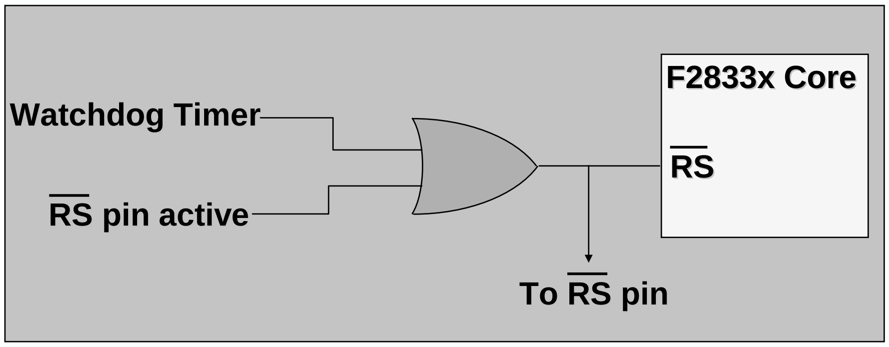
All 16 lines are connected to a table of 'interrupt vectors', which consists of 32 bit memory locations per interrupt. It is the responsibility of the programmer to fill this table with the start addresses of dedicated interrupt service routines. However, in case of the F2833x, this table is in ROM and filled with addresses, defined by Texas Instruments in such a way, that "RESET ($\overline{RS}$)" points to address 0x00 0040, NMI to address 0x00 0042 an so on. All these addresses are in RAM, so the programmer has to fit a single 32-bit instruction into these memory locations.

The F2833x RESET

A high to low transition at the external “RESET ($\overline{RS}$)” pin will cause a reset of the Digital Signal Controller. The next rising edge of $\overline{RS}$ will force the CPU to read the code start address from address 0x3F FFC0 in code memory. This event is not an ‘interrupt’ in the sense that the old program will be resumed. A reset is generated during powering up the device.

Another source for a reset is the overflow of the watchdog timer. To inform all other external devices that the CPU has acknowledged a reset, the device itself drives the reset pin active low. This means that the reset pin must be bi-directional!

F2833x Reset Sources



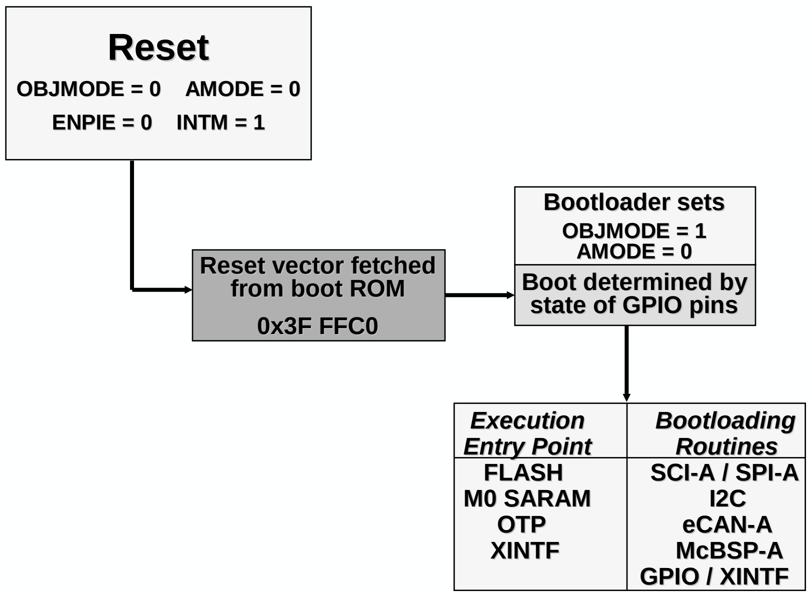
6 - 3

Reset will force the controller not only to start from address 0x3F FFC0, but it will also clear all internal operation registers, reset a group of CPU-Flags to initial states and disable all 16 interrupt lines. We will not go into details about all the flags and registers for now, please refer to the data sheet for the F2833x.

Reset Bootloader

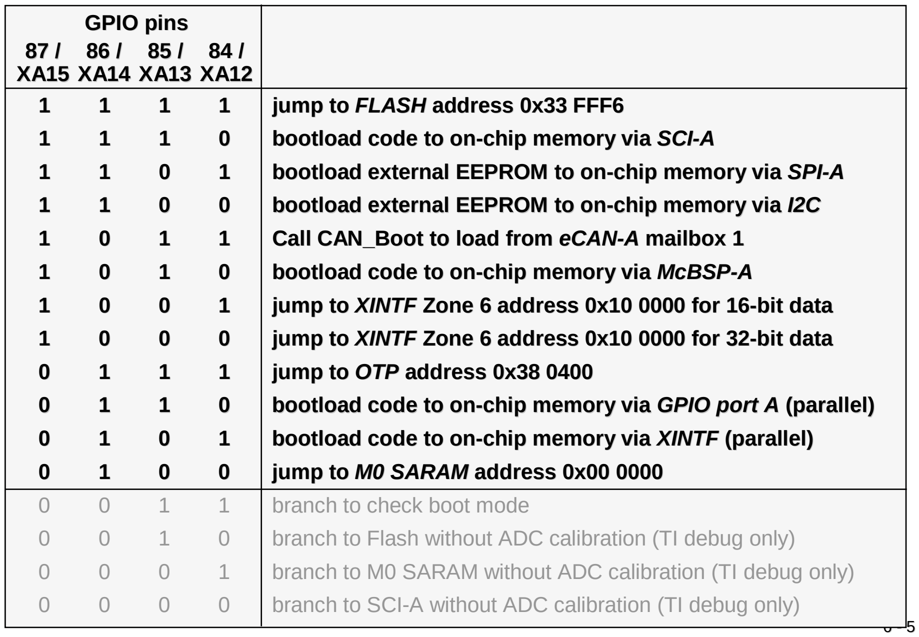
After a RESET signal has been released, the CPU starts the execution of a first code section in ROM, called “boot loader”. This function determines the next step, depending on the status of four GPIO -pins (GPIO87, 86, 85 and 84).

Reset – Bootloader



6 - 4

Bootloader Options



GPIO pins				
87 / XA15	**86 / XA14**	**85 / XA13**	**84 / XA12**	
1	1	1	1	jump to *FLASH* address 0x33 FFF6
1	1	1	0	bootload code to on-chip memory via *SCI-A*
1	1	0	1	bootload external EEPROM to on-chip memory via *SPI-A*
1	1	0	0	bootload external EEPROM to on-chip memory via *I2C*
1	0	1	1	Call CAN_Boot to load from *eCAN-A* mailbox 1
1	0	1	0	bootload code to on-chip memory via *McBSP-A*
1	0	0	1	jump to *XINTF* Zone 6 address 0x10 0000 for 16-bit data
1	0	0	0	jump to *XINTF* Zone 6 address 0x10 0000 for 32-bit data
0	1	1	1	jump to *OTP* address 0x38 0400
0	1	1	0	bootload code to on-chip memory via *GPIO port A* (parallel)
0	1	0	1	bootload code to on-chip memory via *XINTF* (parallel)
0	1	0	0	jump to *M0 SARAM* address 0x00 0000
0	0	1	1	branch to check boot mode
0	0	1	0	branch to Flash without ADC calibration (TI debug only)
0	0	0	1	branch to M0 SARAM without ADC calibration (TI debug only)
0	0	0	0	branch to SCI-A without ADC calibration (TI debug only)

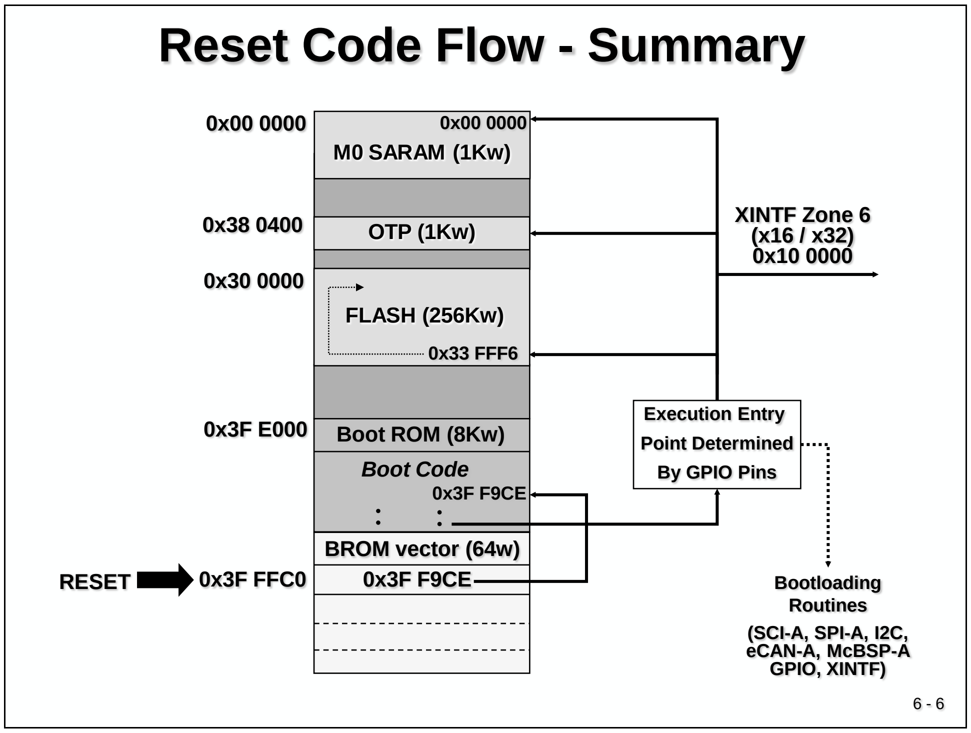
6 - 5

The F28335ControlCard pulls all four GPIO - input lines to ‘1’, so by default the start option “jump to FLASH address 0x3F FFF6” is selected. This will force the controller to continue the code sequence in FLASH memory. However, we do not currently have anything programmed into FLASH memory. So why did all of our previous labs work? The answer is: we over-ruled the hardware - sequence and forced the DSC into our own code entry point by using three of Code Composer Studio Debug commands:

• Reset CPU	-	force the DSC to Reset Address 0x3F FFC0
• Restart	-	force the DSC directly to code entry point “c_int00”, bypassing the hardware start sequence
• Go Main	-	finish the “c_int00”, call “main()” and stop at the first instruction of “main()”.

With the help of jumper J18 (SCI - Boot) on the Peripheral Explorer Board, we could change the hardware sequence. If this jumper is closed, GPIO84 will be ‘0’ and the start sequence is: “boot load code to on-chip memory via SCI-A”. In this operation mode, the chip would wait for a serial communication stream from a host, which is of no use for us for now. This mode will be used in chapter 15.

The next flowchart summarises the reset code flow for all start options of the F2833x.

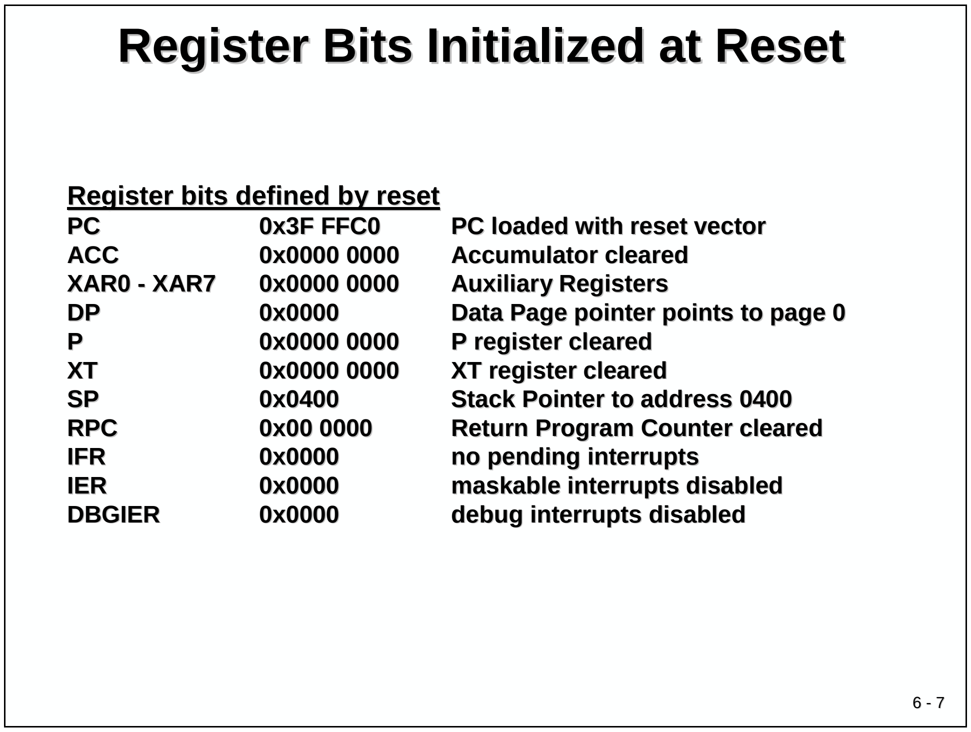


The option ‘Flash Entry’ is usually used at the end of a project development phase when the software flow is bug free. To load a program into the flash you will need to use a specific program, available either as Code Composer Studio plug in or as a stand-alone tool. For our current lab exercises we will refrain from loading (or ‘burning’) the flash memory.

The boot loader options via serial interface (SPI / SCI / I2C / eCAN / McBSP) or parallel port (GPIO / XINTF) are usually used to download the executable code from an external host or to update the contents of the flash memory. For these modes, please refer to chapters 15 and 16.

OTP-memory is a ‘one time programmable’ memory; there is no second chance to fit code into this non-volatile memory. This option is usually used for company specific startup procedures only. Again, to program this portion of memory you would need to use a Code Composer Studio plug in. You might assess your experimental code to be worth storing forever, but for sure your teacher will not. So, PLEASE do not upset your supervisor by using this option, he want to use the boards for future classes!

The next two slides show the status of important core registers and status bits after a reset.



Register Bits Initialized at Reset

Register bits defined by reset

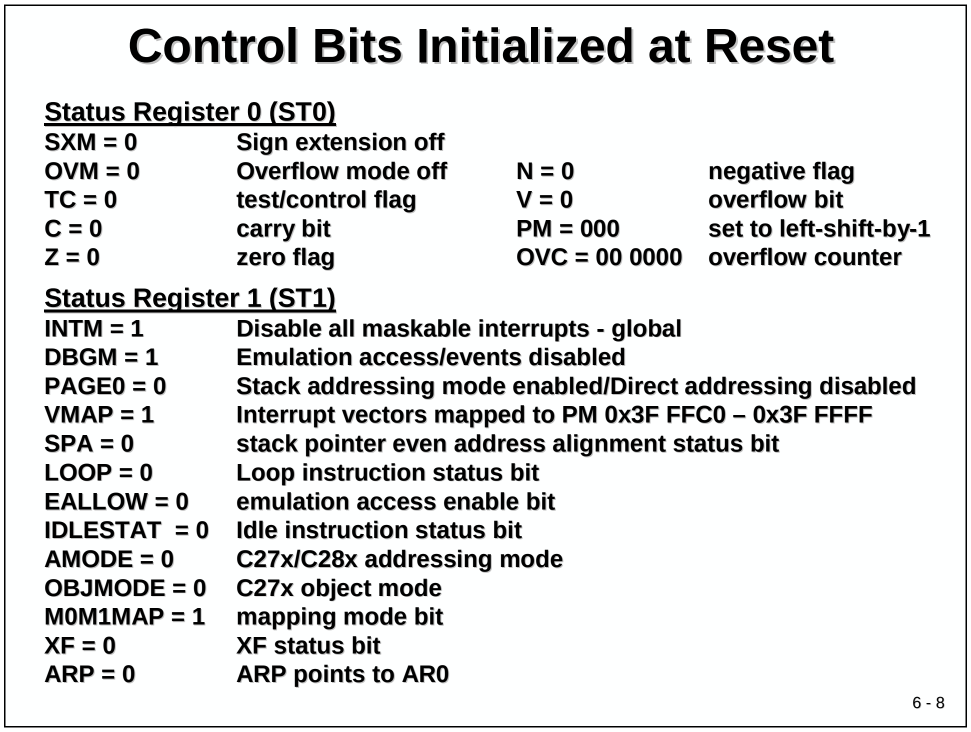
PC	0x3F FFC0	PC loaded with reset vector
ACC	0x0000 0000	Accumulator cleared
XAR0 - XAR7	0x0000 0000	Auxiliary Registers
DP	0x0000	Data Page pointer points to page 0
P	0x0000 0000	P register cleared
XT	0x0000 0000	XT register cleared
SP	0x0400	Stack Pointer to address 0400
RPC	0x00 0000	Return Program Counter cleared
IFR	0x0000	no pending interrupts
IER	0x0000	maskable interrupts disabled
DBGIER	0x0000	debug interrupts disabled

6 - 7

All internal math registers (ACC, P, XT) and auxiliary registers (XAR0 to XAR7) are cleared, interrupts are disabled (IER) and pending interrupts, which have been requested before RESET, are cancelled (IFR). The stack pointer (SP) is initialized to address 0x400 and the program counter (PC) points to hardware start address 0x3F FFC0.

The two registers ST0 and ST1 combine all control and status flags of the CPU. Slide 6-8 explains the reset status of all the bits. ST0 contains all math bits such as zero (Z), carry (C) and negative (N), whereas ST1 covers some more general operating mode bits.

We will postpone the discussion of the individual meaning of the bits until later chapters.



Control Bits Initialized at Reset

Status Register 0 (ST0)

SXM = 0	Sign extension off		
OVM = 0	Overflow mode off	N = 0	negative flag
TC = 0	test/control flag	V = 0	overflow bit
C = 0	carry bit	PM = 000	set to left-shift-by-1
Z = 0	zero flag	OVC = 00 0000	overflow counter

Status Register 1 (ST1)

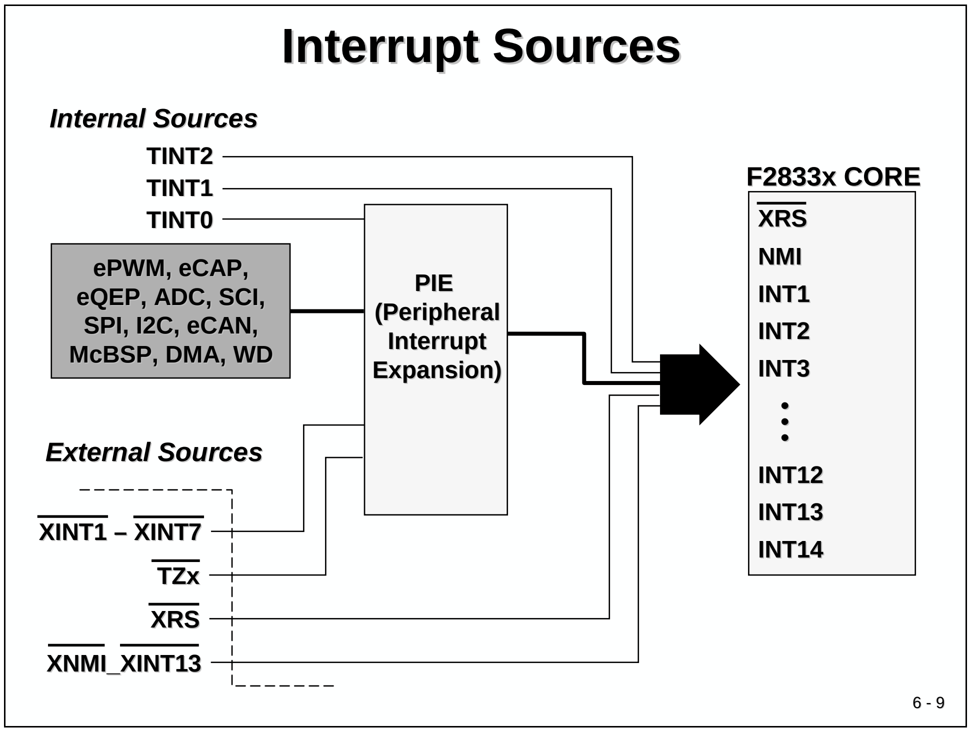
INTM = 1	Disable all maskable interrupts - global
DBGM = 1	Emulation access/events disabled
PAGE0 = 0	Stack addressing mode enabled/Direct addressing disabled
VMAP = 1	Interrupt vectors mapped to PM 0x3F FFC0 – 0x3F FFFF
SPA = 0	stack pointer even address alignment status bit
LOOP = 0	Loop instruction status bit
EALLOW = 0	emulation access enable bit
IDLESTAT = 0	Idle instruction status bit
AMODE = 0	C27x/C28x addressing mode
OBJMODE = 0	C27x object mode
M0M1MAP = 1	mapping mode bit
XF = 0	XF status bit
ARP = 0	ARP points to AR0

6 - 8

Interrupt Sources

As you can see from the next slide the F2833x has a large number of interrupt sources (96 at the moment) but only 14 maskable interrupt inputs. The question is: How do we handle this ‘bottleneck’?

Obviously we have to use a single INT-line for multiple sources. Each interrupt line is connected to its interrupt vector, a 32-bit memory space inside the vector table. This memory space holds the address for the interrupt service routine. In case of multiple interrupts this service routine must be used for all incoming interrupt requests. This technique forces the programmer to use a software based separation method on entry of this service routine. This method will cost additional time that is often not available in real time applications. So how can we speed up this interrupt service?



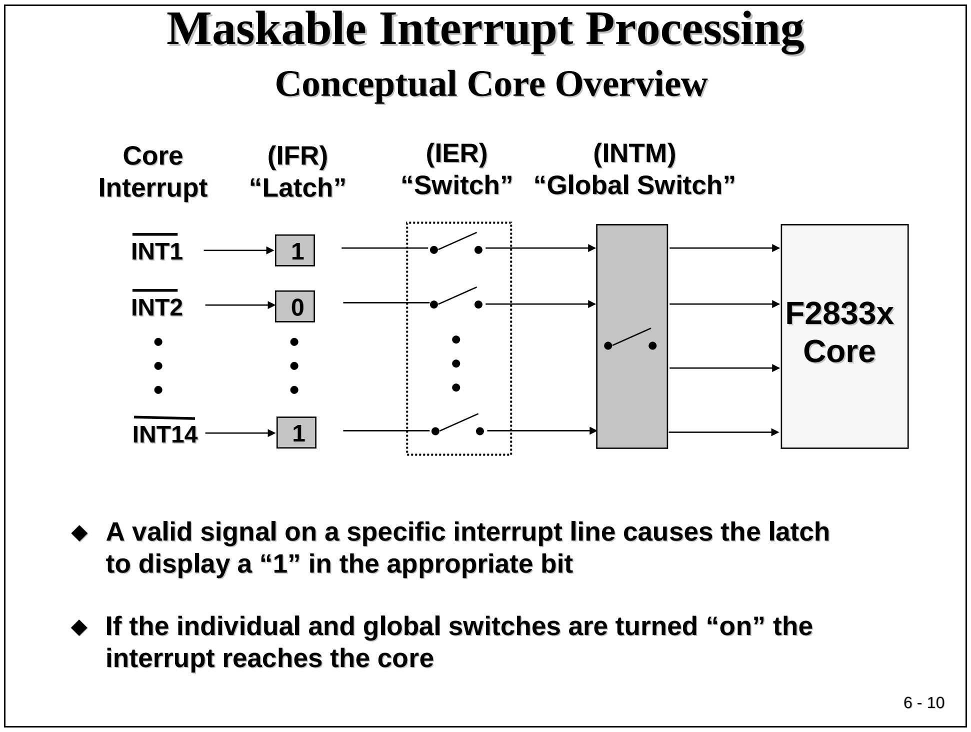
The answer from Texas Instruments is sweet, they simply used a pie. PIE stands for Peripheral Interrupt Expansion unit.

This unit ‘expands’ the vector address table into a larger scale, reserving individual 32 bit entries for each of the 96 possible interrupt sources. An interrupt response with the help of this unit is much faster than without it. To use the PIE we will have to re-map the location of the interrupt vector table to address 0x 00 0D00. This is in volatile memory! Before we can use this memory we will have to initialise it.

Do not worry about the PIE-procedure for the moment, we will exercise all this during Lab6.

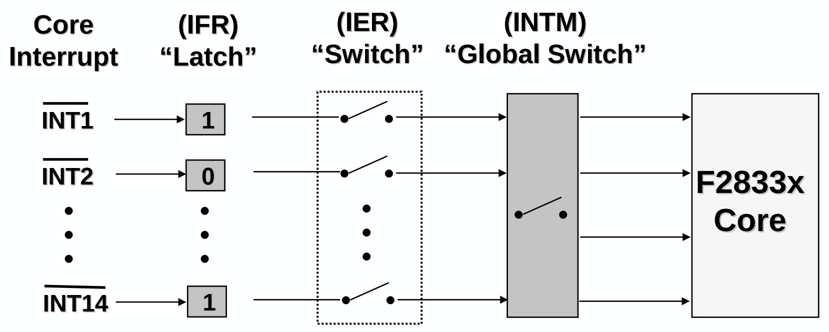
Maskable Interrupt Processing

Before we dive into the PIE-registers, we have to discuss the remaining path from an interrupt request to its acknowledgement by the DSC. As you can see from the next slide, we have to close two more switches to allow an interrupt request.



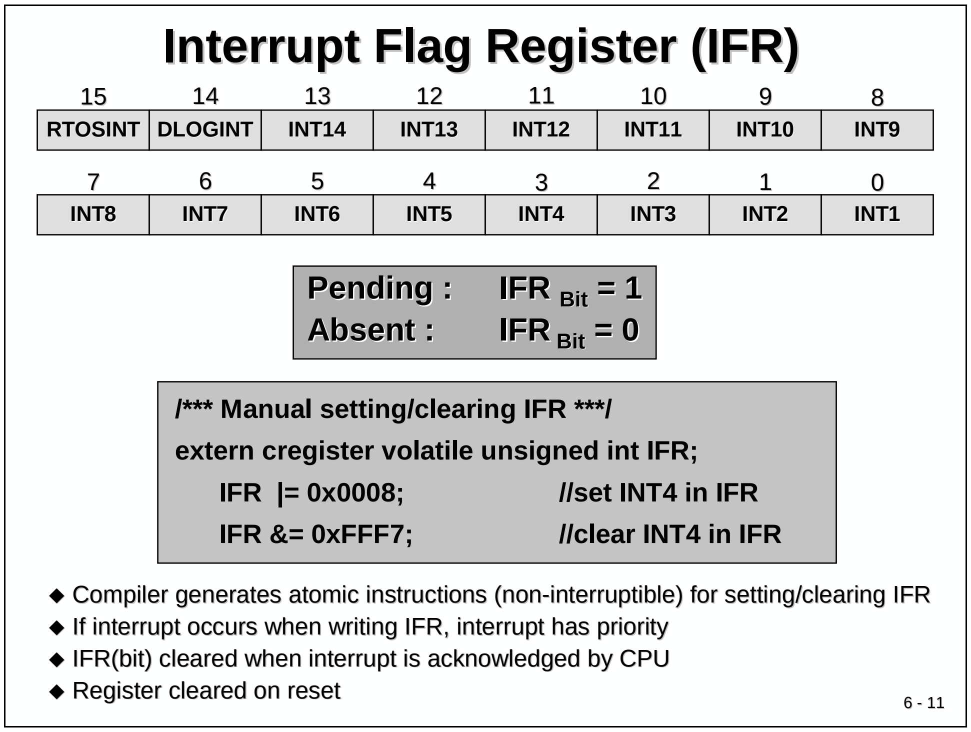
Maskable Interrupt Processing

Conceptual Core Overview



- A valid signal on a specific interrupt line causes the latch to display a "1" in the appropriate bit
- If the individual and global switches are turned "on" the interrupt reaches the core

6 - 10



Interrupt Flag Register (IFR)

15	14	13	12	11	10	9	8
RTOSINT	DLOGINT	INT14	INT13	INT12	INT11	INT10	INT9

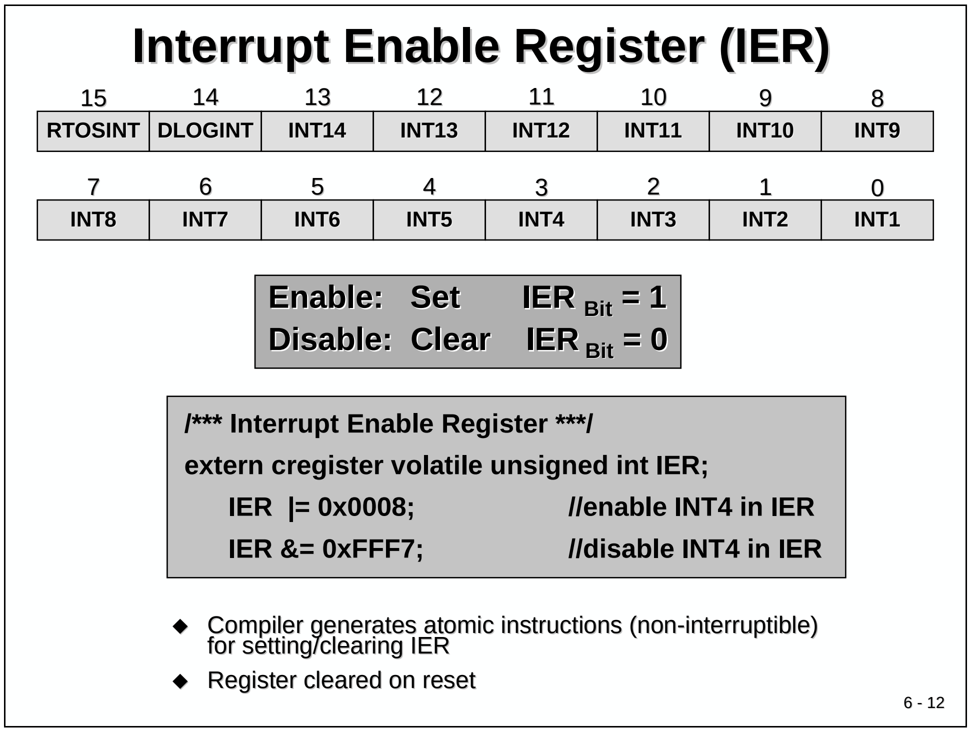
7	6	5	4	3	2	1	0
INT8	INT7	INT6	INT5	INT4	INT3	INT2	INT1

Pending : $IFR_{Bit} = 1$
Absent : $IFR_{Bit} = 0$

```
/*** Manual setting/clearing IFR ***/
extern cregister volatile unsigned int IFR;
    IFR  |= 0x0008;              //set INT4 in IFR
    IFR &= 0xFFF7;               //clear INT4 in IFR
```

- Compiler generates atomic instructions (non-interruptible) for setting/clearing IFR
- If interrupt occurs when writing IFR, interrupt has priority
- IFR(bit) cleared when interrupt is acknowledged by CPU
- Register cleared on reset

6 - 11



Interrupt Enable Register (IER)

15	14	13	12	11	10	9	8
RTOSINT	**DLOGINT**	**INT14**	**INT13**	**INT12**	**INT11**	**INT10**	**INT9**

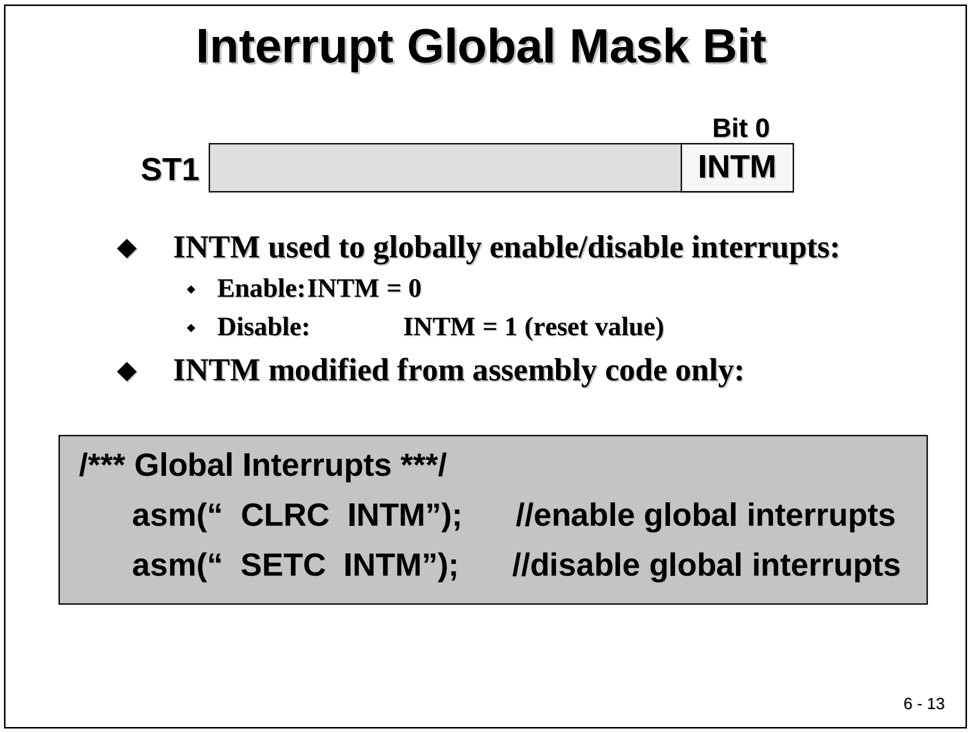
7	6	5	4	3	2	1	0
INT8	**INT7**	**INT6**	**INT5**	**INT4**	**INT3**	**INT2**	**INT1**

Enable: Set IER $_{Bit}$ = 1
Disable: Clear IER $_{Bit}$ = 0

```
/*** Interrupt Enable Register ***/
extern cregister volatile unsigned int IER;
    IER  |= 0x0008;            //enable INT4 in IER
    IER &= 0xFFF7;             //disable INT4 in IER
```

- Compiler generates atomic instructions (non-interruptible) for setting/clearing IER
- Register cleared on reset

6 - 12



Interrupt Global Mask Bit

		Bit 0
ST1		**INTM**

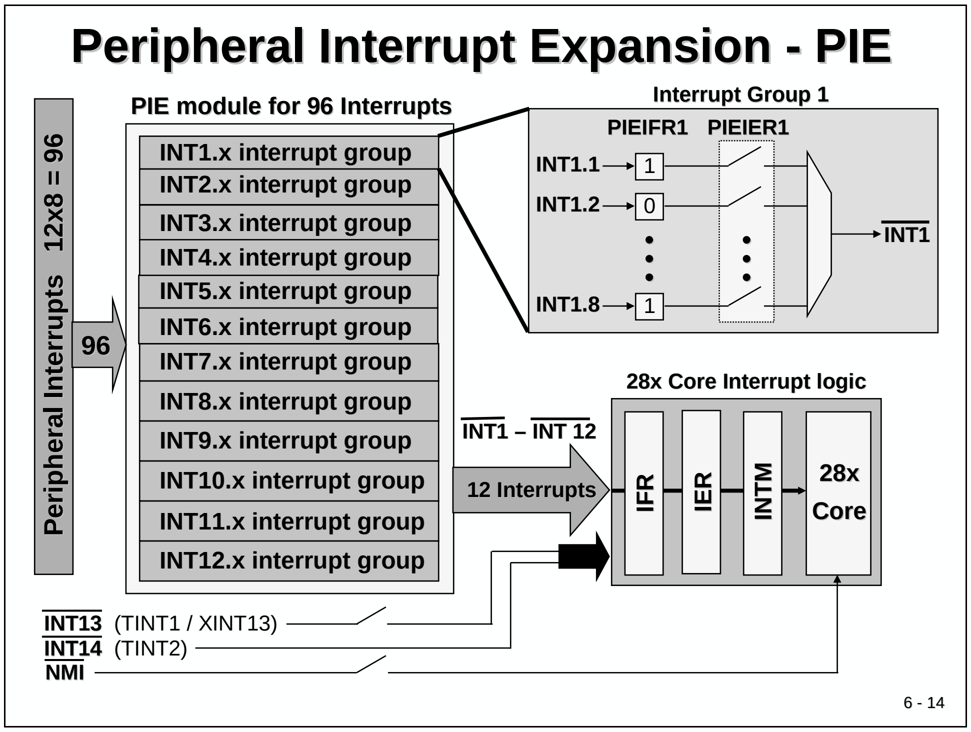
- **INTM used to globally enable/disable interrupts:**
  - **Enable: INTM = 0**
  - **Disable: INTM = 1 (reset value)**
- **INTM modified from assembly code only:**

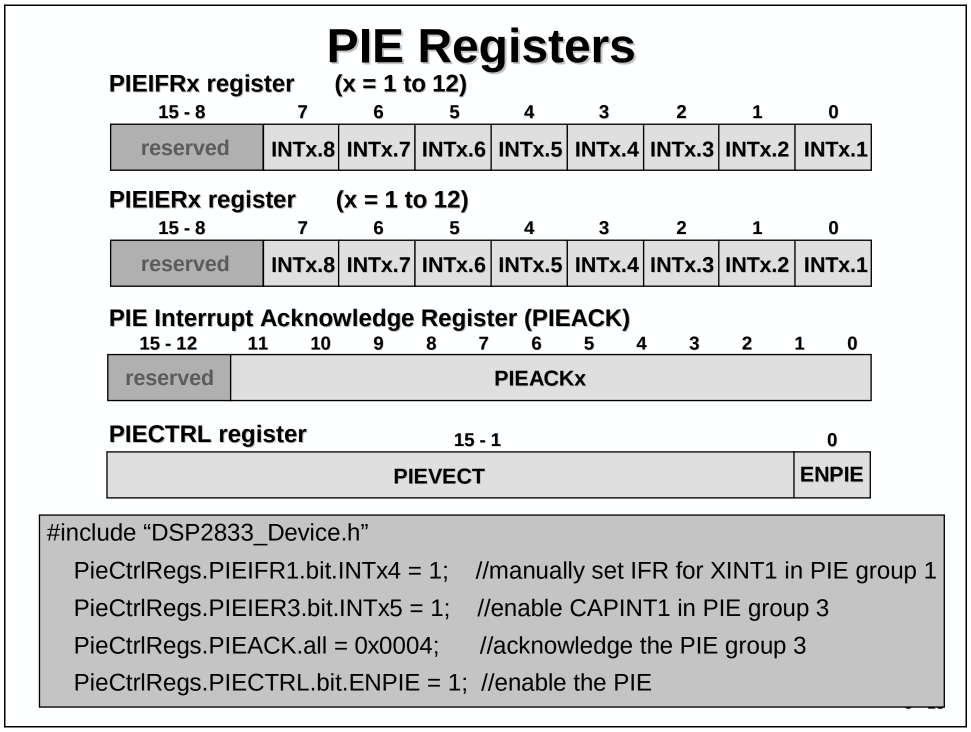
```
/*** Global Interrupts ***/
    asm(" CLRC  INTM");      //enable global interrupts
    asm(" SETC  INTM");      //disable global interrupts
```

6 - 13

Peripheral Interrupt Expansion

All 96 possible sources are grouped into 12 PIE-lines, 8 sources per line. To enable/disable individual sources we have to program another group of registers: ‘PIEIFRx’ and ‘PIEIERx’.





PIE Registers

PIEIFRx register (x = 1 to 12)

15 - 8	7	6	5	4	3	2	1	0
reserved	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1

PIEIERx register (x = 1 to 12)

15 - 8	7	6	5	4	3	2	1	0
reserved	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1

PIE Interrupt Acknowledge Register (PIEACK)

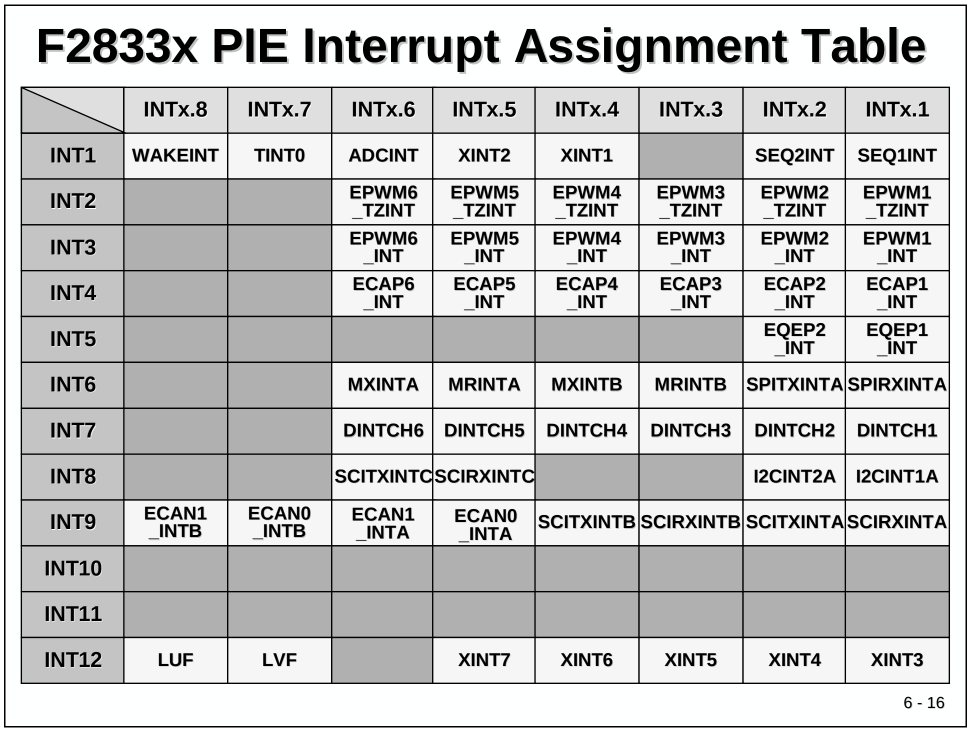
15 - 12	11 - 0
reserved	PIEACKx

PIECTRL register

15 - 1	0
PIEVECT	ENPIE

```
#include “DSP2833_Device.h”
    PieCtrlRegs.PIEIFR1.bit.INTx4 = 1;    //manually set IFR for XINT1 in PIE group 1
    PieCtrlRegs.PIEIER3.bit.INTx5 = 1;    //enable CAPINT1 in PIE group 3
    PieCtrlRegs.PIEACK.all = 0x0004;      //acknowledge the PIE group 3
    PieCtrlRegs.PIECTRL.bit.ENPIE = 1;  //enable the PIE
```

All interrupt sources are connected to interrupt lines according to this assignment table:



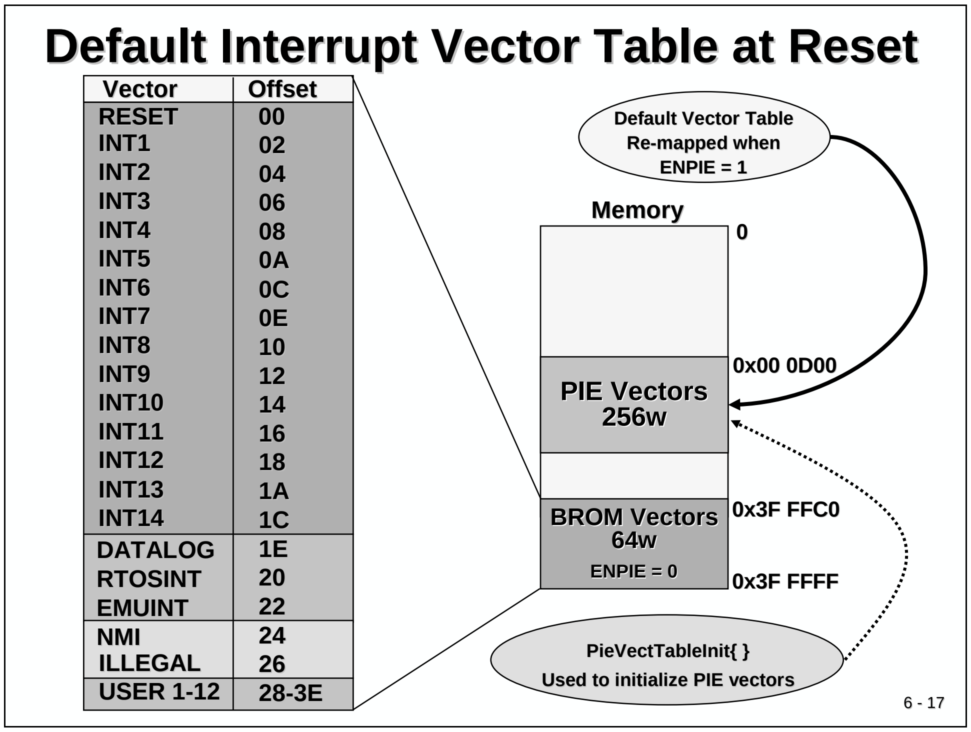
F2833x PIE Interrupt Assignment Table

	INTx.8	INTx.7	INTx.6	INTx.5	INTx.4	INTx.3	INTx.2	INTx.1
INT1	WAKEINT	TINT0	ADCINT	XINT2	XINT1		SEQ2INT	SEQ1INT
INT2			EPWM6 _TZINT	EPWM5 _TZINT	EPWM4 _TZINT	EPWM3 _TZINT	EPWM2 _TZINT	EPWM1 _TZINT
INT3			EPWM6 _INT	EPWM5 _INT	EPWM4 _INT	EPWM3 _INT	EPWM2 _INT	EPWM1 _INT
INT4			ECAP6 _INT	ECAP5 _INT	ECAP4 _INT	ECAP3 _INT	ECAP2 _INT	ECAP1 _INT
INT5							EQEP2 _INT	EQEP1 _INT
INT6			MXINTA	MRINTA	MXINTB	MRINTB	SPITXINTA	SPIRXINTA
INT7			DINTCH6	DINTCH5	DINTCH4	DINTCH3	DINTCH2	DINTCH1
INT8			SCITXINTC	SCIRXINTC			I2CINT2A	I2CINT1A
INT9	ECAN1 _INTB	ECAN0 _INTB	ECAN1 _INTA	ECAN0 _INTA	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10								
INT11								
INT12	LUF	LVF		XINT7	XINT6	XINT5	XINT4	XINT3

6 - 16

Examples: ADCINT = INT1.6; T2PINT = INT3.1; SCITXINTA = INT9.2

The vector table location at reset is:

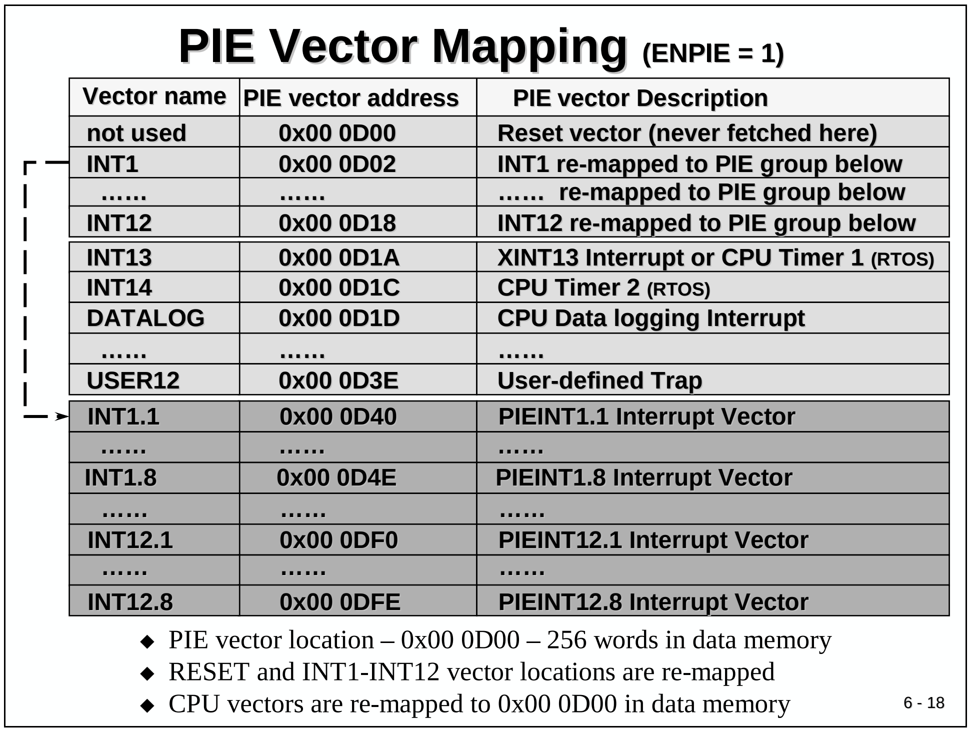


Default Interrupt Vector Table at Reset

Vector	Offset
RESET	00
INT1	02
INT2	04
INT3	06
INT4	08
INT5	0A
INT6	0C
INT7	0E
INT8	10
INT9	12
INT10	14
INT11	16
INT12	18
INT13	1A
INT14	1C
DATALOG	1E
RTOSINT	20
EMUINT	22
NMI	24
ILLEGAL	26
USER 1-12	28-3E

6 - 17

The PIE re-maps the location like this:



PIE Vector Mapping (ENPIE = 1)

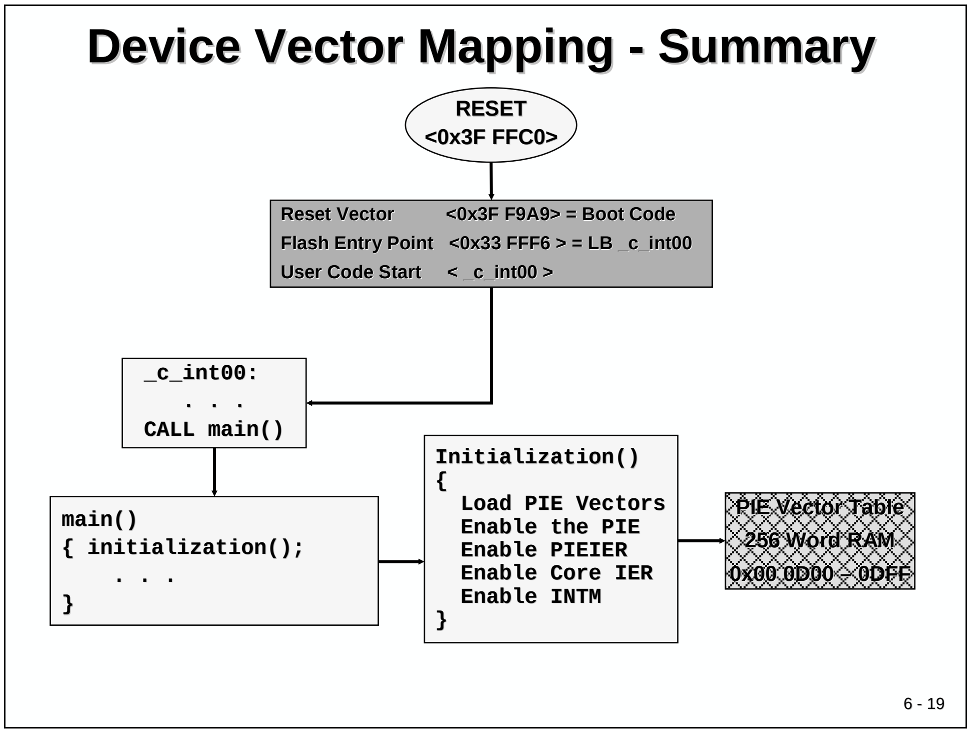
Vector name	PIE vector address	PIE vector Description
not used	0x00 0D00	Reset vector (never fetched here)
INT1	0x00 0D02	INT1 re-mapped to PIE group below
……	……	…… re-mapped to PIE group below
INT12	0x00 0D18	INT12 re-mapped to PIE group below
INT13	0x00 0D1A	XINT13 Interrupt or CPU Timer 1 (RTOS)
INT14	0x00 0D1C	CPU Timer 2 (RTOS)
DATALOG	0x00 0D1D	CPU Data logging Interrupt
……	……	……
USER12	0x00 0D3E	User-defined Trap
INT1.1	0x00 0D40	PIEINT1.1 Interrupt Vector
……	……	……
INT1.8	0x00 0D4E	PIEINT1.8 Interrupt Vector
……	……	……
INT12.1	0x00 0DF0	PIEINT12.1 Interrupt Vector
……	……	……
INT12.8	0x00 0DFE	PIEINT12.8 Interrupt Vector

- PIE vector location – 0x00 0D00 – 256 words in data memory
- RESET and INT1-INT12 vector locations are re-mapped
- CPU vectors are re-mapped to 0x00 0D00 in data memory

6 - 18

As you can see from Slide 6-18, the addresses 0x00 0D40 to 0x00 0DFF are used as the expansion area. Now we do have 32 bits for each individual interrupt vector PIEINT1.1 to PIEINT12.8.

Device Vector Mapping - Summary


RESET
<0x3F FFC0>
Reset Vector <0x3F F9A9> = Boot Code
Flash Entry Point <0x33 FFF6 > = LB _c_int00
User Code Start < _c_int00 >
PIE Vector Table
256 Word RAM
0x00 0D00 – 0DFF

```
_c_int00:
   . . .
CALL main()
```

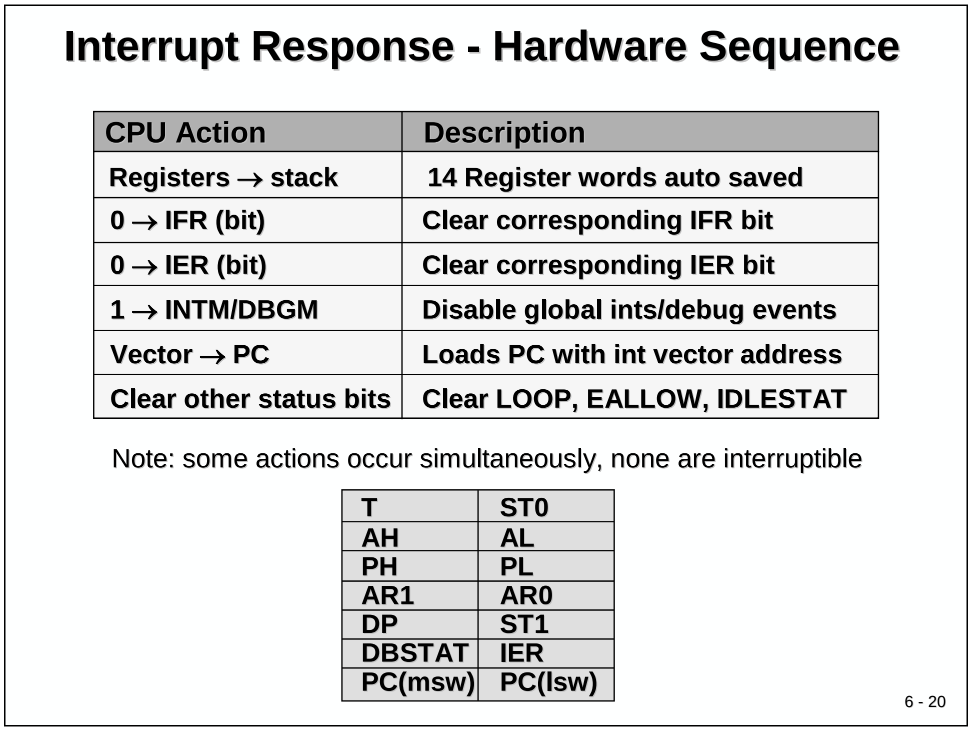
```
main()
{ initialization();
    . . .
}
```

```
Initialization()
{
  Load PIE Vectors
  Enable the PIE
  Enable PIEIER
  Enable Core IER
  Enable INTM
}
```

6 - 19

Hardware Interrupt Response

After an interrupt has been acknowledged by the CPU, an automatic hardware context switch sequence is started. It includes an auto-save of 14 internal registers with the all-important internal control and status bits, and loads the program counter (PC) with the address of the ISR.



Interrupt Response - Hardware Sequence

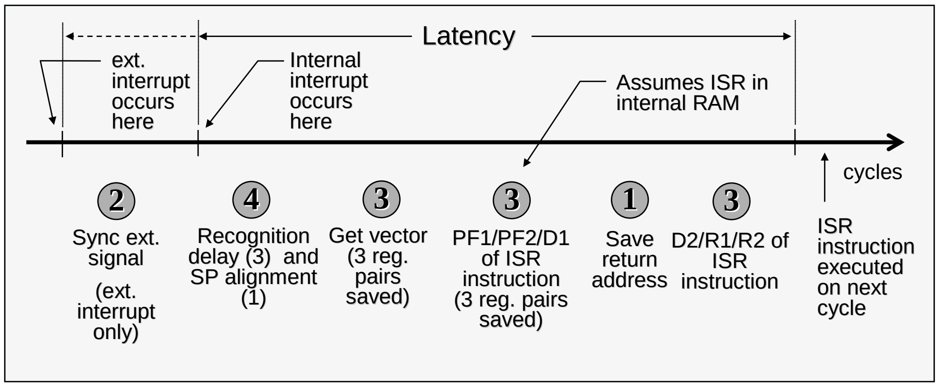
CPU Action	Description
Registers → stack	14 Register words auto saved
0 → IFR (bit)	Clear corresponding IFR bit
0 → IER (bit)	Clear corresponding IER bit
1 → INTM/DBGM	Disable global ints/debug events
Vector → PC	Loads PC with int vector address
Clear other status bits	Clear LOOP, EALLOW, IDLESTAT

Note: some actions occur simultaneously, none are interruptible

T	ST0
AH	AL
PH	PL
AR1	AR0
DP	ST1
DBSTAT	IER
PC(msw)	PC(lsw)

6 - 20

Interrupt Latency


Latency
ext. interrupt occurs here
Internal interrupt occurs here
Assumes ISR in internal RAM
cycles
2
Sync ext. signal (ext. interrupt only)
4
Recognition delay (3) and SP alignment (1)
3
Get vector (3 reg. pairs saved)
3
PF1/PF2/D1 of ISR instruction (3 reg. pairs saved)
1
Save return address
3
D2/R1/R2 of ISR instruction
ISR instruction executed on next cycle

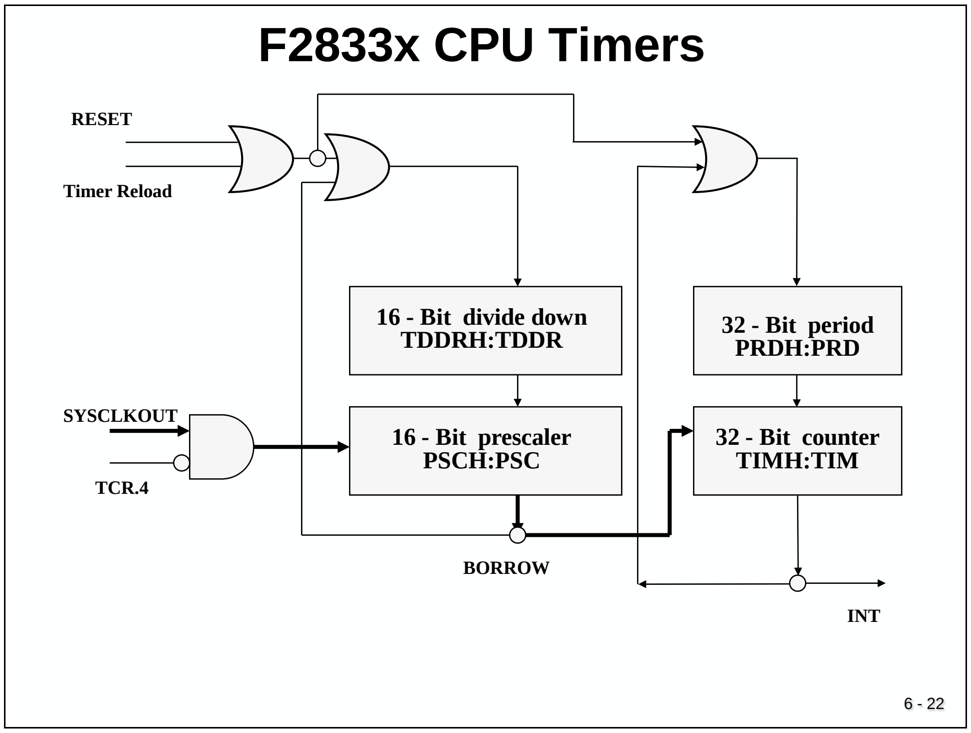
Above is for PIE enabled or disabled

- **Minimum latency (to when real work occurs in the ISR):**
  - **Internal interrupts: 14 cycles**
  - **External interrupts: 16 cycles**
- **Maximum latency: Depends on wait states, ready, INTM, etc.**

6 - 21

F2833x CPU Timers

The F2833x features 3 independent 32-bit core timers. The block diagram for one timer is shown below in Slide 6-22:

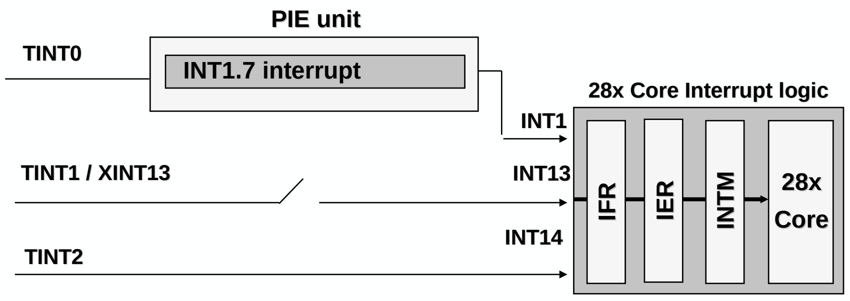


As you can see, the clock source is the internal clock “SYSCLKOUT”, which is usually 150MHz, assuming an external oscillator of 30MHz and a PLL-ratio of 10/2. Once the timer is enabled (TCR-bit 4 = 0), the incoming clock counts down a 16-bit prescaler (PSCH: PSC). On underflow, its borrow signal is used to count down the 32-bit counter (TIMH: TIM). At the end, when this timer underflows, an interrupt request is transferred to the CPU.

The 16-bit divide down register (TDDRH: TDDR) is used as a reload register for the prescaler. Each times the prescaler underflows, the value from the divide down-register is reloaded into the prescaler. A similar reload function for the counter is performed by the 32-bit period register (PRDH_PRD).

Timer 1 and Timer 2 are usually used by Texas Instruments for the real time operation system “DSP/BIOS”, whereas Timer 0 is generally free for general usage. Lab 6 will use Timer 0. This will not only preserve Timer 1 and 2 for later use together with DSP/BIOS, but also help us to understand the PIE-unit, because Timer 0 is the only timer of the CPU that goes through the PIE, as can be seen in the following slide, Slide 6-23:

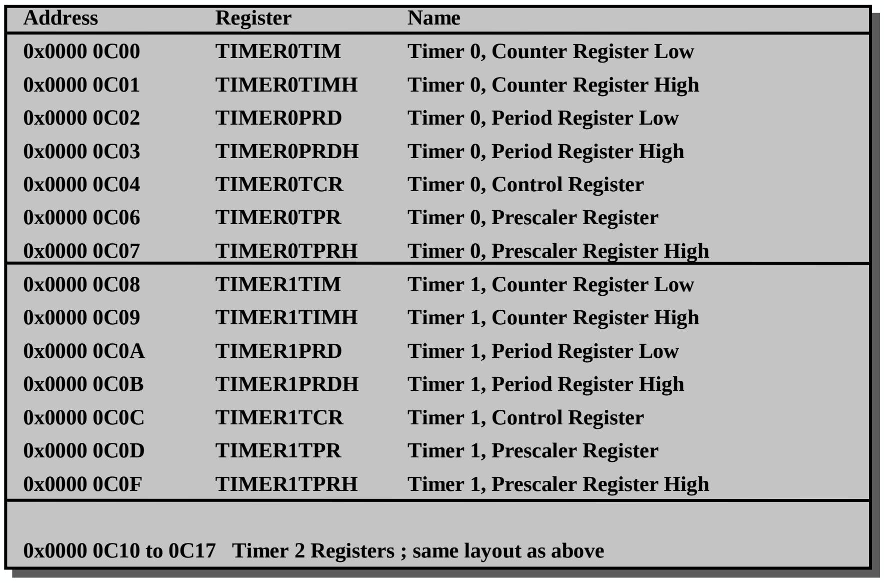
F2833x Timer Interrupt System



6 - 23

A timer unit is usually initialized by a set of registers. In Lab6, we will perform an exercise with the registers of CPU Timer 0. However, instead of setting every single bit by ourselves, we will use a hardware abstraction function, for which we only have to specify the desired timer period and the clock speed of our processor. This function is provided by Texas Instruments as part of a set of such functions.

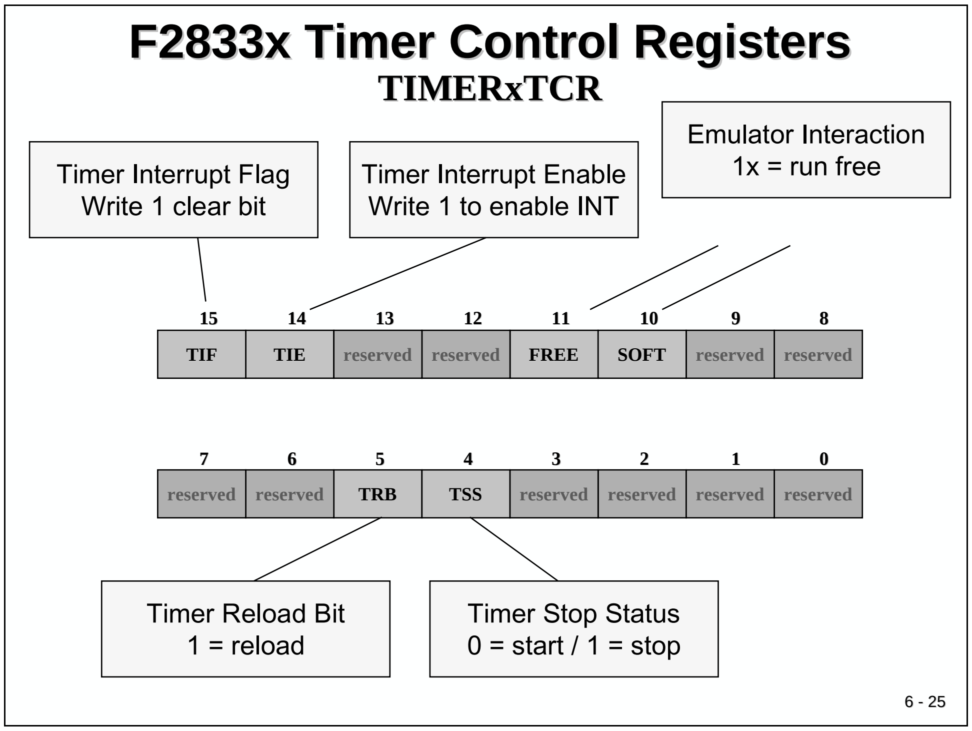
F2833x Timer Registers



Address	Register	Name
0x0000 0C00	TIMER0TIM	Timer 0, Counter Register Low
0x0000 0C01	TIMER0TIMH	Timer 0, Counter Register High
0x0000 0C02	TIMER0PRD	Timer 0, Period Register Low
0x0000 0C03	TIMER0PRDH	Timer 0, Period Register High
0x0000 0C04	TIMER0TCR	Timer 0, Control Register
0x0000 0C06	TIMER0TPR	Timer 0, Prescaler Register
0x0000 0C07	TIMER0TPRH	Timer 0, Prescaler Register High
0x0000 0C08	TIMER1TIM	Timer 1, Counter Register Low
0x0000 0C09	TIMER1TIMH	Timer 1, Counter Register High
0x0000 0C0A	TIMER1PRD	Timer 1, Period Register Low
0x0000 0C0B	TIMER1PRDH	Timer 1, Period Register High
0x0000 0C0C	TIMER1TCR	Timer 1, Control Register
0x0000 0C0D	TIMER1TPR	Timer 1, Prescaler Register
0x0000 0C0F	TIMER1TPRH	Timer 1, Prescaler Register High
0x0000 0C10 to 0C17	Timer 2 Registers ; same layout as above	

6 - 24

It is worthwhile to inspect the control register, as this is the most important register of a timer unit.



Summary:

Sounds pretty complicated, doesn't it? Well, nothing is better suited to understand the PIE unit than a lab exercise. In Lab 6 you will add the initialization of the PIE vector table to re-map the vector table to address 0x00 0D00. You will also use CPU Timer 0 as a clock time base for the source code of Lab 5_1 ("4 bit LED-counter").

Remember, so far we generated time periods with a software-loop in function "delay_loop()". This was quite a waste of processor time, not very precise and poor programming technique.

The procedure on the next page will guide you through the necessary steps to modify the source code step by step.

Take your time, no pain no gain!

We will use functions, pre-defined by Texas Instruments as often as we can. This principle will save us a lot of development time; we do not have to re-invent the wheel again and again!

Lab 6: CPU Timer 0 Interrupt and 4 LEDs

Objective

The objective of this lab is to include a basic example of the interrupt system in the “LED-counter” project of Lab5_1. Instead of using a software delay loop to generate the time interval between the output steps, which is a poor use of processor time, we will now use one of the 3 core CPU timers to do the job. One of the simplest tasks for a timer is to generate a periodic interrupt request. We can use its interrupt service routine to perform periodic activities OR to increment a global variable. This variable will then contain the number of periods that are elapsed from the start of the program.

CPU Timer 0 is using the Peripheral Interrupt Expansion (PIE) Unit. This gives us the opportunity to exercise this unit as well. Timer 1 and 2 bypass the PIE-unit and they are usually reserved for Texas Instruments real-time operating system, called “DSP/BIOS”. Therefore we implement Timer 0 as the core clock for this exercise.

Procedure

Create a Project File

1. Using Code Composer Studio, create a new project, called **Lab6.pjt** in C:\DSP2833x_V4\Labs (or in another path that is accessible by you; ask your teacher or a technician for an appropriate location!).
2. Define the size of the C system stack. In the project window, right click at project “Lab6” and select “Properties”. In category “C/C++ Build”, “C2000 Linker”, “Basic Options” set the C stack size to 0x400.
3. Open the file Lab5_1.c from C:\DSP2833x_V4\Labs\Lab5 and save it as Lab6.c in C:\DSP2833x_V4\Labs\Lab6.

Next, we will take advantage of some useful files, which have been created and provided by Texas Instruments and should be already available on your hard disk drive C as part of the so-called "Header File" package (sprc530.zip). If not, ask a technician to install that package for you!

4. In the C/C++ perspective, right click at project “Lab6” and select “**Link Files to Project**”. Go to folder “*C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\source”* and link:

   - **DSP2833x_GlobalVariableDefs.c**

   This file defines all global variable names to access memory mapped peripheral registers.

5. Repeat the “**Link Files to Project**” step. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* add:

   - **DSP2833x_CodeStartBranch.asm**
   - **DSP2833x_SysCtrl.c**

- **DSP2833x_ADC_cal.asm**
- **DSP2833x_usDelay.asm**

6. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\cmd* link to project "Lab6":

   - **DSP2833x_Headers_nonBIOS.cmd**

   This linker command file will connect all global register variables to their corresponding physical addresses.

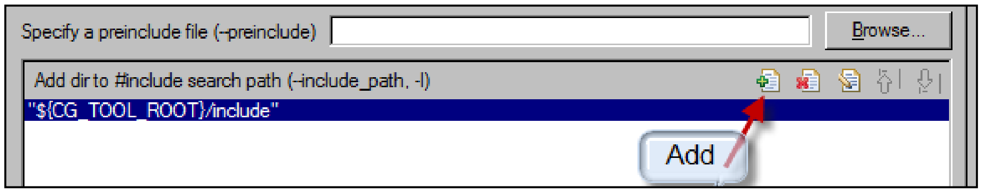
Project Build Options

7. We also have to extent the search path of the C-Compiler for include files. Right click at project "Lab6" and select "Properties". Select "C/C++ Build", "C2000 Compiler", "Include Options". In the box: "Add dir to #include search path", add the following lines:

   C:\tidcs\C28\dsp2833x\v131\DSP2833x_headers\include

   C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\include

   Note: Use the "Add" Icon to add the new paths:



Specify a preinclude file (--preinclude)

Add dir to #include search path (--include_path, -I)

"${CG_TOOL_ROOT}/include"

   Close the Property Window by Clicking <**OK**>.

Modify the Source Code

8. Open Lab6.c to edit: double click on "Lab6.c" inside the project window. At the start of your code, add the function prototype statement for the external function "InitSysCtrl()":

   extern void InitSysCtrl(void);

9. Remove the function prototype for the local function "InitSystem()" at the beginning and the whole function definition at the end of Lab6.c

10. In main replace the function call "InitSystem()" by "InitSysCtrl()".

11. Since "InitSysCtrl()" disables the watchdog, but we would like the watchdog to be active, we have to re-enable the watchdog. Add the following lines just after the call of function "InitSysCtrl()":

```
EALLOW;
SysCtrlRegs.WDCR = 0x00AF;
EDIS;
```

Build, Load and Test

12. Click the “Rebuild Active Project ” button or perform:

    Project → Rebuild All (Alt +B)

    and watch the tools run in the build window. If you get errors or warnings debug as necessary.

13. Load the output file in the debugger session:

    Target → Debug Active Project

    and switch into the “Debug” perspective.

14. Verify that in the debug perspective the window of the source code “Lab6.c” is highlighted and that the blue arrow for the current Program Counter position is placed under the line “void main(void)”.

15. Perform a real time run.

    Target ➔ Run

16. Verify that the LEDs behave as expected. In this case you have successfully finished the first part of Lab6. Halt the Device (Target ➔ Halt). Switch back into the “C/C++” – Perspective.

Modify Source Code - Part 2

17. At the beginning of “Lab6.c” add a function prototype for a new interrupt service function for CPU Timer 0:

    interrupt void cpu_timer0_isr(void);

18. In “main()”, directly after the function call "Gpio_select()", add a function call to:

    InitPieCtrl();

This is a function that is provided by TI’s header file examples. We use this function “as it is”. The purpose of this function is to clear all pending PIE-Interrupts and to disable all PIE interrupt lines. This is a useful step when we would like to initialize the PIE-unit. Function “InitPieCtrl ()” is defined in the source code file “DSP2833x_PieCtrl.c”; we have to link this file to our project:

19. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* link to project:

    DSP2833x_PieCtrl.c

    Also, add an external function prototype at the beginning of Lab6.c:

    extern void InitPieCtrl(void);

20. Inside "main()", directly after the function call "InitPieCtrl();", add a function call to:

```
InitPieVectTable();
```

This function will initialize the PIE-memory to an initial state. It uses a predefined interrupt table "PieVectTableInit()" - defined in source code file "DSP2833x_PieVect.c" and copies this table to the global variable "PieVectTable" - defined in "DSP2833x_GlobalVariableDefs.c". Variable "PieVectTable" is linked to the physical memory of the PIE area.

Also, add an external function prototype at the beginning of Lab6.c:

```
extern void InitPieVectTable(void);
```

To be able to use "InitPieVectTable()", we need to link two more code files to our project:

21. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source,* link to project:

DSP2833x_PieVect.c and

DSP2833x_DefaultIsr.c

The code file "DSP2833x_DefaultIsr.c" will add a set of interrupt service routines to our project. When you open and inspect this file, you will find that all ISRs consist of an endless for-loop and a specific assembler instruction "ESTOP0". This instruction behaves like a software breakpoint. This is a security measure. Remember, at this point we have disabled all PIE interrupts. If we were to now run the program, we should never see an interrupt request. If, for some reason, for example a power supply glitch, noise interference or just a software bug, the DSP calls an interrupt service routine, then we can catch this event by the "ESTOP0" break.

22. Now we have to re-map the entry for CPU-Timer0 Interrupt Service from the "ESTOP0" operation to a real interrupt service. Editing the source code of TI's code "DSP2833x_DefaultIsr.c" would be one way to do this. Of course this would not be a wise decision, because we would modify the original code for this single Lab exercise. **SO DO NOT DO THAT**! A much better way is to modify the entry for CPU-Timer0 Interrupt Service directly inside the PIE-memory. This is done in main by adding the next 3 lines after the function call of "InitPieVectTable();":

```
EALLOW;
PieVectTable.TINT0 = &cpu_timer0_isr;
EDIS;
```

EALLOW and EDIS are two macros to enable and disable the access to a group of protected registers; the PIE is part of this area. The name of our own interrupt service routine for Timer0 is "cpu_timer0_isr()". We created the prototype statement earlier in the procedure for this Lab. Please be sure to use the same name as you used in the prototype statement!

23. Inside "main()", directly after the re-mapping instructions from above, add the function call "InitCpuTimers();". This function will set the core Timer0 to a known state and it will stop this timer.

```
InitCpuTimers();
```

Also, add an external function prototype at the beginning of Lab6.c:

extern void InitCpuTimers(void);

Again, we use a predefined function. To do so, we have to link the source code file "DSP2833x_CpuTimers.c" to our project.

24. From *C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source* link to project:

    DSP2833x_CpuTimers.c

25. Now we have to initialize Timer0 to generate a period of 100ms. TI has provided a function "ConfigCpuTimer()". All we have to do is to pass 3 arguments to this function. Parameter 1 is the address of the core timer structure, e.g. "CpuTimer0"; Parameter 2 is the internal speed of the DSP in MHz, e.g. 150 for 150MHz; Parameter 3 is the period time for the timer overflow in microseconds, e.g. 100000 for 100 milliseconds. The following function call will setup Timer0 to a 100ms period:

    ConfigCpuTimer(&CpuTimer0, 150, 100000);

Add this function call in "main()" directly after the line "InitCpuTimers();"

Again, add an external function prototype at the beginning of Lab6.c:

extern void ConfigCpuTimer(struct CPUTIMER_VARS *, float, float);

26. Before we can start timer0 we have to enable its interrupt masks. We have to take care of 3 levels to enable an individual interrupt source. Level 1 is the PIE unit. To enable it, we have to set bit 7 of PIEIER1 to 1. Why? Because the Timer0 interrupt is directly connected to group INT1, Bit7. Add the following line to your code after the call of "ConfigCpuTimer()" in step 25:

    PieCtrlRegs.PIEIER1.bit.INTx7 = 1;

27. Next, enable interrupt core line 1 (INT1). Modify the register IER accordingly.

    IER |= 1;

28. Next, enable control – interrupts (EINT) and debug – interrupts (ERTM) globally. This is done by adding the two code macros:

    EINT; and
    ERTM;

29. Finally, we have to start Timer 0. The bit TSS inside register TCR will do the job. Add:

    CpuTimer0Regs.TCR.bit.TSS = 0;

30. After the end of "main()", we have to add our new interrupt service routine "cpu_timer0_isr()". Remember, we have prototyped this function at the beginning of our modifications. Now we have to add its body. Inside this function we have to perform two activities:

    1st - increment the interrupt counter "**CpuTimer0.InterruptCount**". This way we will have global information about how often this 100 milliseconds task was called.

2nd - acknowledge the interrupt service as the last line before return. This step is necessary to re-enable the next Timer 0 interrupt service. It is done by:

PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

31. Now we are almost done. Inside the endless while(1) loop of “main()” we have to delete the function call: “delay_loop(1000000);”. We do not need this function any longer; we can also delete its prototype at the top of our code and its function body, which is still present after the code of “main()”.

32. Inside the endless loop “while(1)“, after the “if-else”-construct, we have to implement a statement to wait until the global variable “CpuTimer0.InterruptCount” has been incremented to 1, which corresponds to the interval of 100 milliseconds. Remember to reset the variable “CpuTimer0.InterruptCount” to zero when you continue after the wait statement. Note: The global variable “CpuTimer0.InterruptCount” has been defined in the file “DSP2833x_CpuTimers.c” as a global and volatile variable, which also has been initialized to zero when we called the function “ConfigCpuTimer()”.

33. Done?

34. No, not quite! We forgot the watchdog! It is still alive and we removed the service instructions together with the function “delay_loop()”. So we have to add the watchdog reset sequence somewhere into our modified source code. Where? A good strategy is to service the watchdog not in a single portion of our code. Our code now consists of two independent tasks: the while-loop of main and the interrupt service routine of timer 0. Place one of the two reset instructions for WDKEY into the ISR and the other one into the while(1)-loop of main.

    If you are a little bit fearful about being bitten by the watchdog, then disable it first; try to get your code running without it. Later, when the code works as expected, you can re-think the watchdog service part again.

Build, Load and Test

35. Click the “Rebuild Active Project ” button or perform:

    Project → Rebuild All (Alt +B)

    If you get errors or warnings debug as necessary.

36. Load the output file in the debugger session:

    Target → Debug Active Project and switch into the “Debug” perspective.

37. Perform a real time run.

    Target → Run

38. Verify that the LEDs behave as expected. You have successfully finished Lab6. Halt the Device (Target → Halt). Switch back into the “C/C++” – Perspective.

End of Lab6.