Interrupt System

Introduction

This module is used to explain the interrupt system of the C28x DSP.

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 working at your desk to prepare the next day laboratory experiments. 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 next day task; otherwise you might not pass the next exam.

This analogy touches some basic definitions for interrupts;

- interrupts appear "suddenly": in technical terms it 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 with the phone call should be as small as possible - "interrupt latency"
- after the call you should continue your work at the very position 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. This event causes the DSP 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 C28x interrupt system.

At the end of this chapter we will exercise with an interrupt controlled program 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

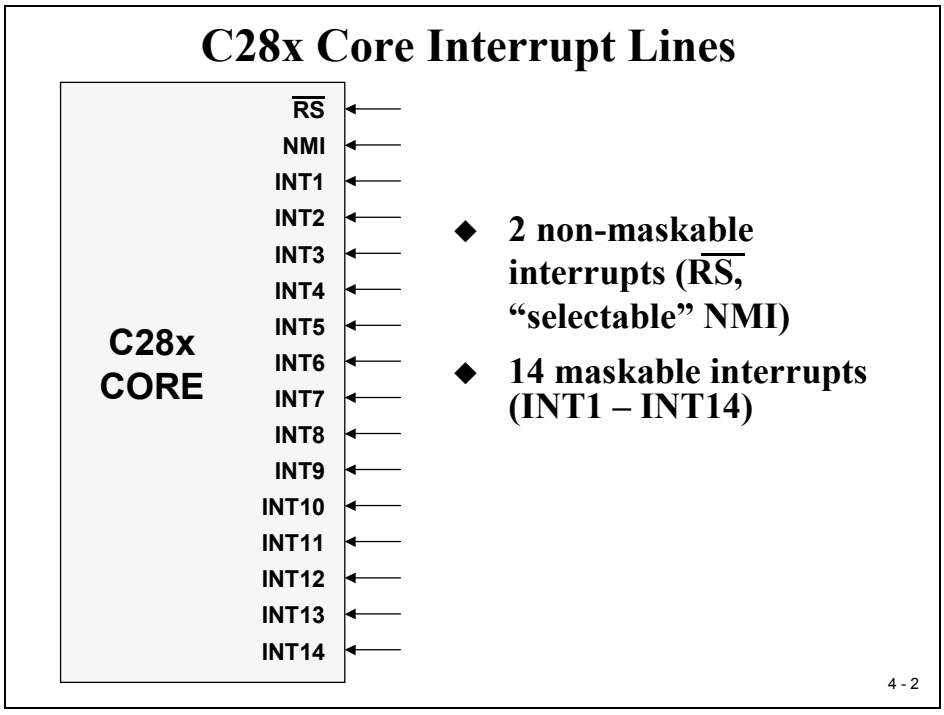
C28x Core Interrupt Lines

The core interrupt system of the C28x 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 dis-allow interrupts from these 14 lines.

What does “maskable” or “non-maskable” mean?

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 allowed to request an interrupt.

For a “non-maskable” interrupt we can’t deny an interrupt request. Once the signal line goes active, the running program will be suspended and the dedicated interrupt service routine will start.

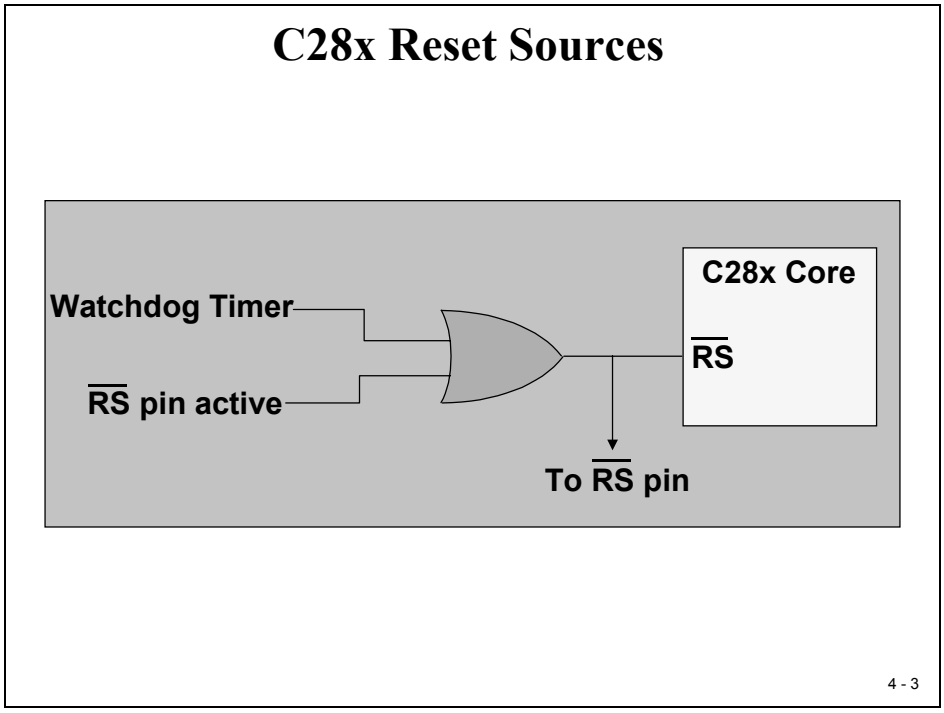


All 16 lines are connected to a table of ‘interrupt vectors’, which consists of 32 bit memory locations per interrupt. It is the responsibility of the programmer to fill this table with the start addresses of dedicated interrupt service routines.

The C28x RESET

A high to low transition at the external "/RS" pin will cause a reset of the DSP. This event will force the DSP to start from its reset address (code memory 0x3F FFC0). This event is not an 'interrupt' in the sense that the old program will be resumed. A reset is generated during powering up the DSP.

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

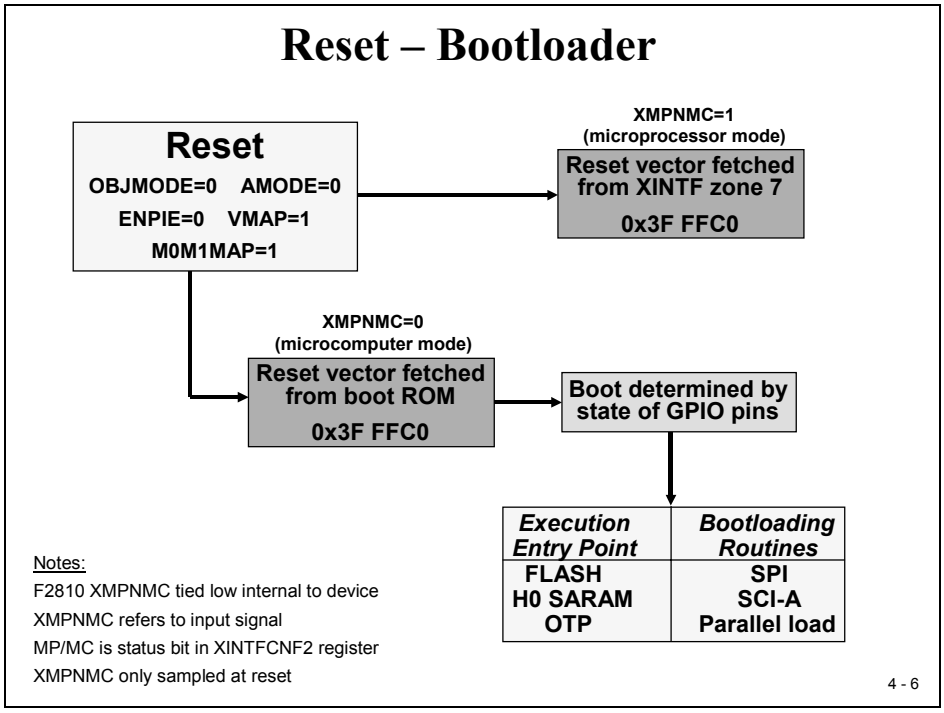


Reset will force the DSP not only to start from address 0x3F FFC0, it will also clear all internal operation registers, all CPU-Flags and will disable all 16 interrupt lines. We will not go into details about all the flags and registers for now; we will leave this for the assembler-based chapters.

Let's have a look now into the start procedure triggered by a reset. Remember, the memory map of the C28x allows us to have two memories at physical address 0x3F FFC0, the internal ROM and the external memory. Another physical pin, called Microprocessor/Microcontroller-Mode (XMP/MC) makes the decision as to which one will be used. Setting this pin to 1 will select the external memory and disable the internal address. Connecting this pin to zero will select the internal ROM to be used as the start address area. The status of XMP/MC will be copied into a flag 'XMP/MC' that can be used by software later.

Reset Bootloader

When internal ROM is selected, bootloader software is started. This function determines the next step, depending on the status of four GPIO –pins. On the eZdsp we have 5 jumpers to setup the start condition (JP1, JP7, JP8, JP11, and JP12 – see manual).



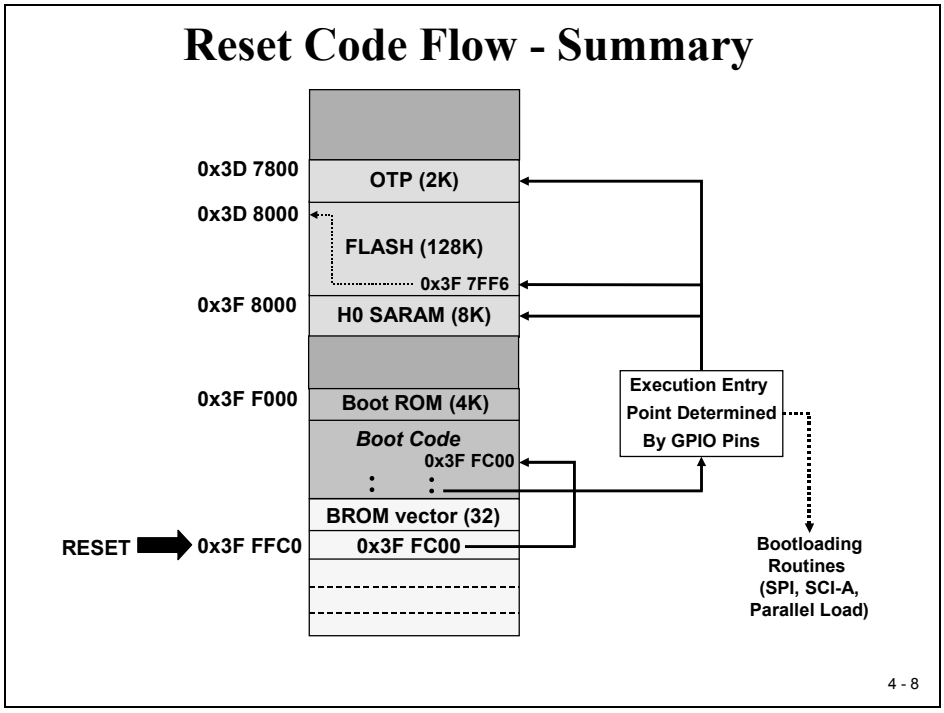
Bootloader Options

GPIO pins				
F4	F12	F3	F2	
1	x	x	x	jump to *FLASH* address 0x3F 7FF6 *
0	0	1	0	jump to *H0 SARAM* address 0x3F 8000 *
0	0	0	1	jump to *OTP* address 0x3D 7800 *
0	1	x	x	bootload external EEPROM to on-chip memory via *SPI* port
0	0	1	1	bootload code to on-chip memory via *SCI-A* port
0	0	0	0	bootload code to on-chip memory via *GPIO port B* (parallel)

*** Boot ROM software configures the device for C28x mode before jump**

4 - 7

For our Lab exercises we use H0 SARAM as execution entry point. Make sure that the eZdsp's jumpers are set to: JP1 - 2:3; JP7 – 2:3; JP8 – 2:3; JP11 – 1:2 and JP12 – 2:3. The next slide summarises the reset code flow for the 6 options in microcontroller mode.



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 lab exercises we will refrain from loading (or 'burning') the flash memory.

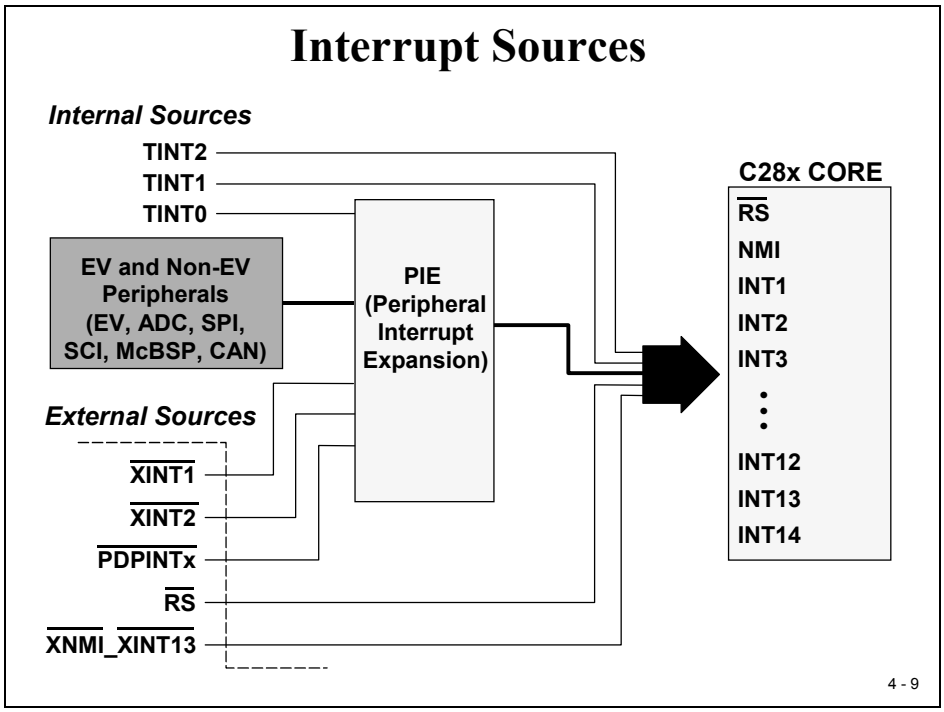
The bootloader options via serial interface (SPI / SCI) or parallel port are usually used to download the executable code from an external host or to field update the contents of the flash memory. We will not use these features during this tutorial.

OTP-memory is a 'one time programmable' memory; there is no second chance to fill 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 Code Composer Studio's 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!

Interrupt Sources

As you can see from the next slide the DSP 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 is the PIE (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.

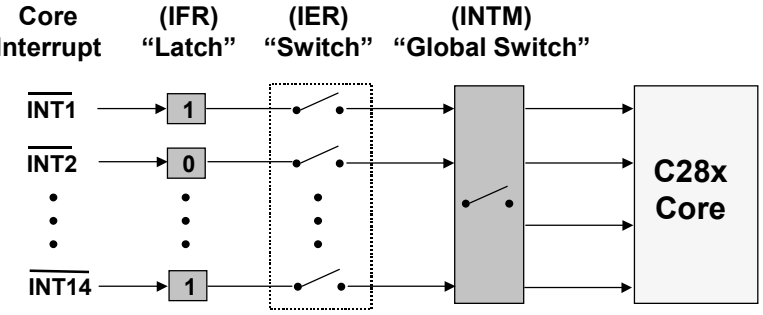
Don't worry about the PIE-procedure for the moment, we will exercise all this during Lab4.

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 DSP. 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

4 - 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

4 - 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

4 - 12

Interrupt Global Mask Bit

		Bit 0
ST1		INTM

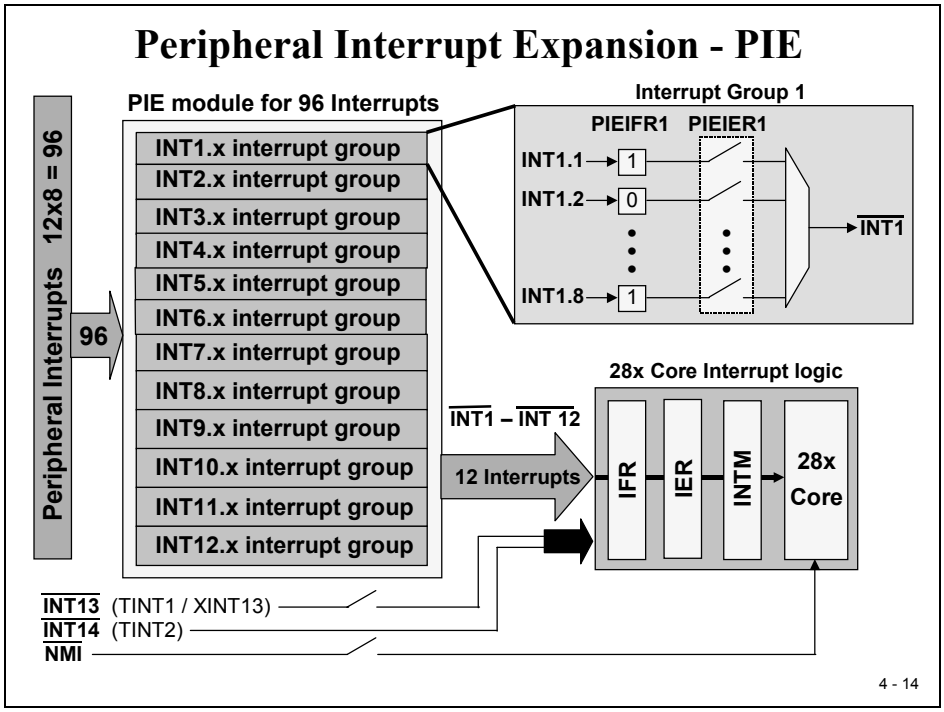
- **INTM used to globally enable/disable interrupts:**
  - **Enable: INTM = 0**
  - **Disable: INTM = 1 (reset value)**
- **INTM modified from assembly code only:**

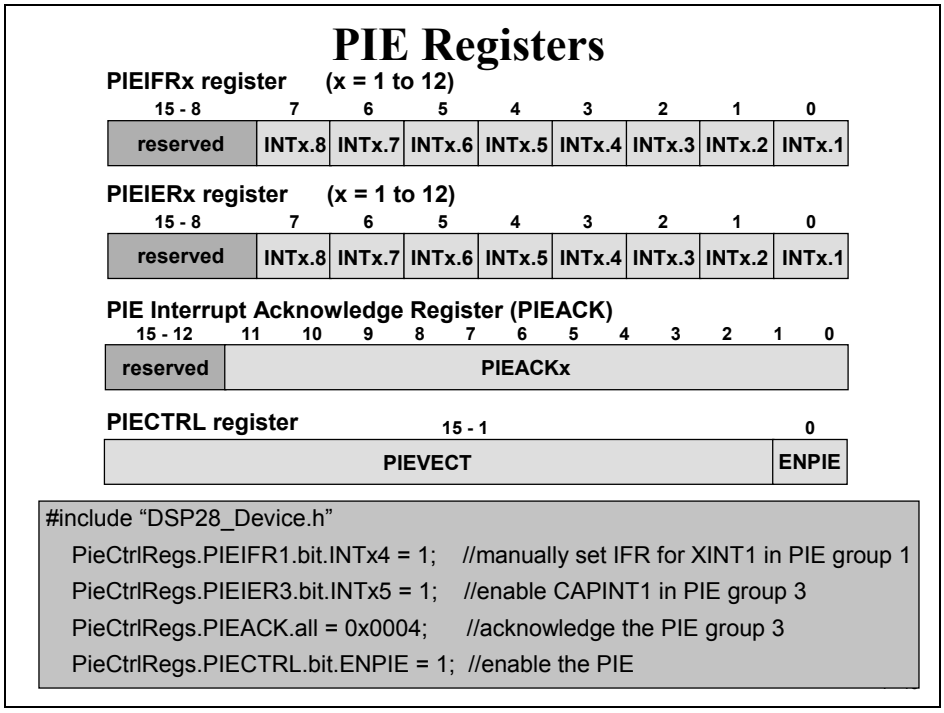
```
/*** Global Interrupts ***/
    asm(“  CLRC  INTM”);     //enable global interrupts
    asm(“  SETC  INTM”);     //disable global interrupts
```

4 - 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’.





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

F2812/10 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		PDPINTB	PDPINTA
INT2		T1OFINT	T1UFINT	T1CINT	T1PINT	CMP3INT	CMP2INT	CMP1INT
INT3		CAPINT3	CAPINT2	CAPINT1	T2OFINT	T2UFINT	T2CINT	T2PINT
INT4		T3OFINT	T3UFINT	T3CINT	T3PINT	CMP6INT	CMP5INT	CMP4INT
INT5		CAPINT6	CAPINT5	CAPINT4	T4OFINT	T4UFINT	T4CINT	T4PINT
INT6			MXINT	MRINT			SPITXINTA	SPIRXINTA
INT7								
INT8								
INT9			ECAN1INT	ECAN0INT	SCITXINTB	SCIRXINTB	SCITXINTA	SCIRXINTA
INT10								
INT11								
INT12								

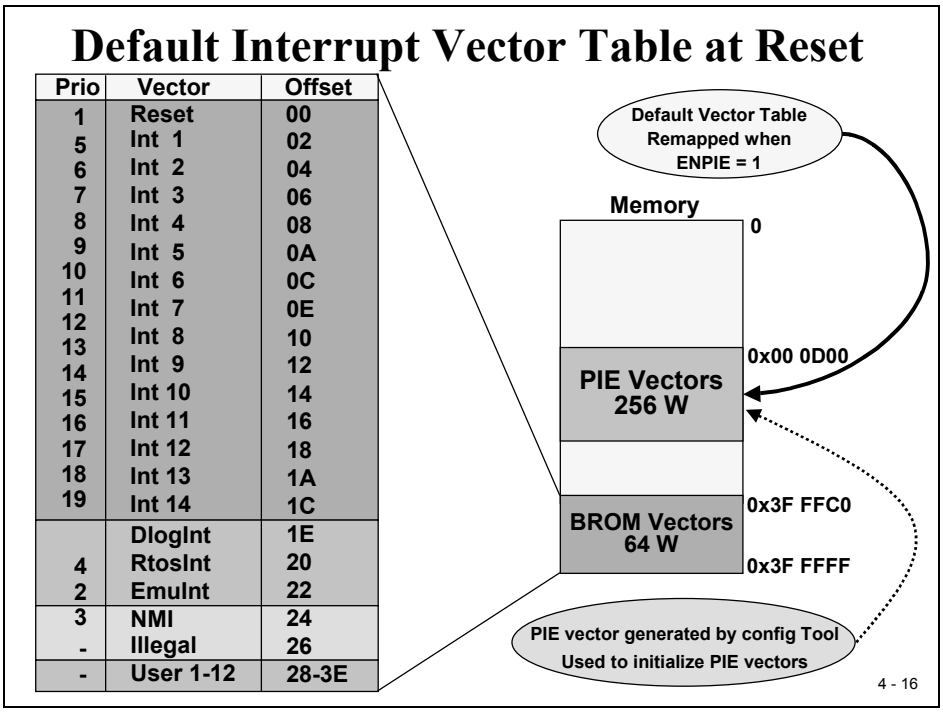
4 - 18

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

The vector table location at reset is:

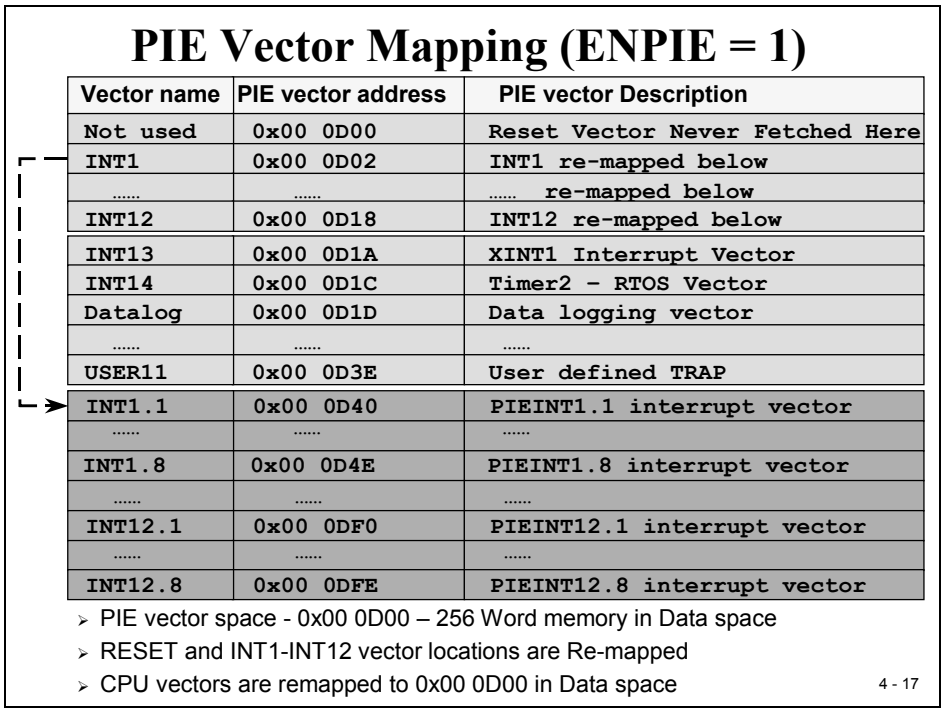
Default Interrupt Vector Table at Reset

Prio	Vector	Offset
1	Reset	00
5	Int 1	02
6	Int 2	04
7	Int 3	06
8	Int 4	08
9	Int 5	0A
10	Int 6	0C
11	Int 7	0E
12	Int 8	10
13	Int 9	12
14	Int 10	14
15	Int 11	16
16	Int 12	18
17	Int 13	1A
18	Int 14	1C
19	DlogInt	1E
	RtosInt	20
4	EmuInt	22
2	NMI	24
3	Illegal	26
-	User 1-12	28-3E
-		

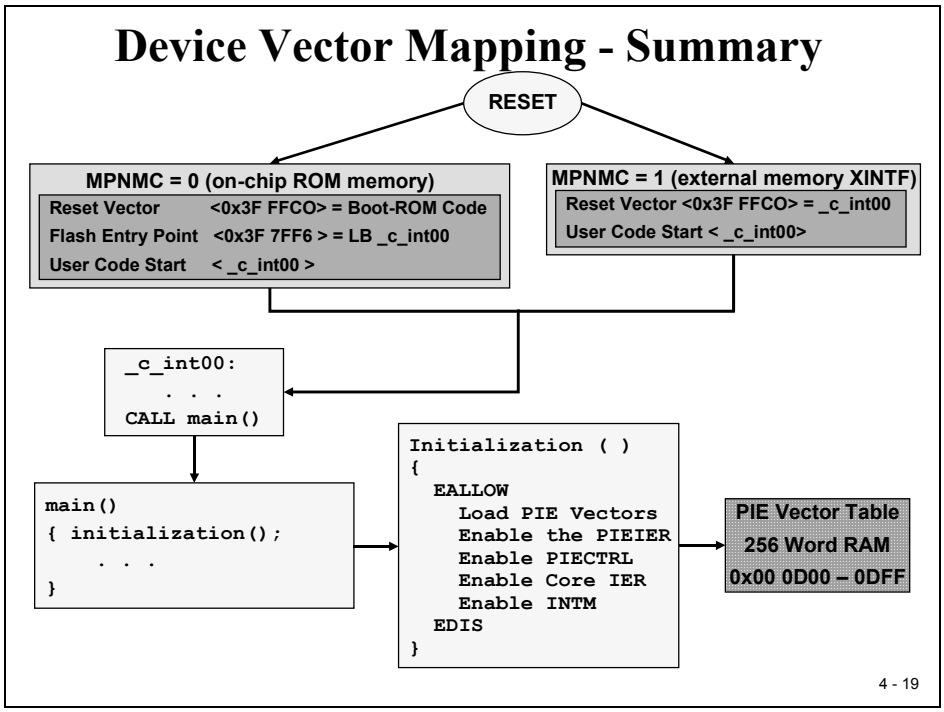


4 - 16

The PIE re-map location looks like this:

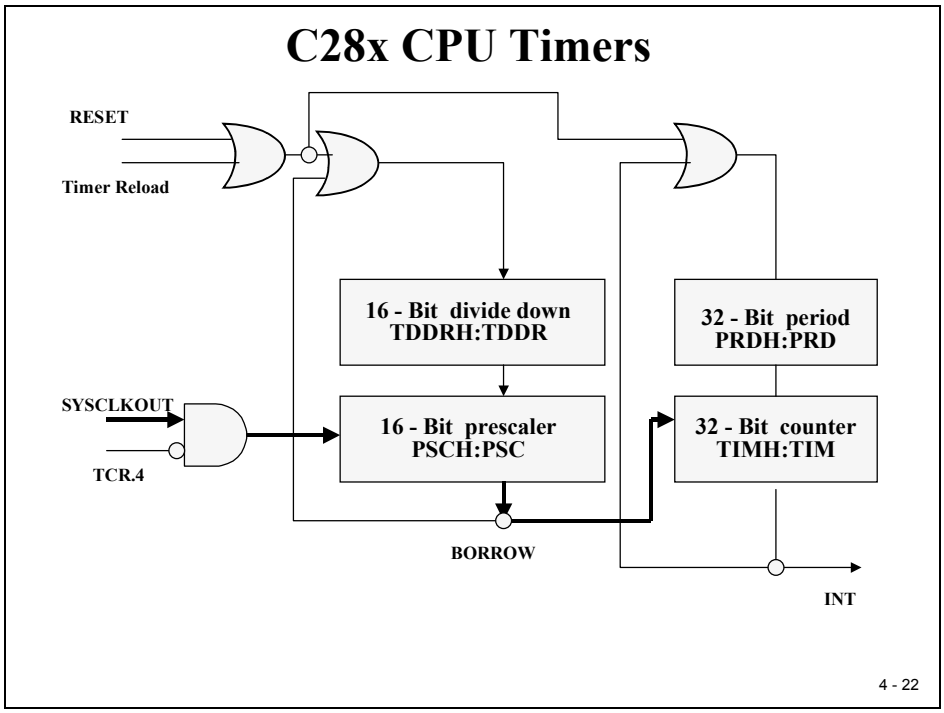


As you can see from the slide, the address area 0x00 0D40 to 0x00 0DFF is used as the expansion area. Now we do have 32 bits for each individual interrupt vector PIEINT1.1 to PIEINT12.8.



C28x CPU Timers

The C28x has 3 32-Bit CPU Timers. The block diagram for one timer is shown here:



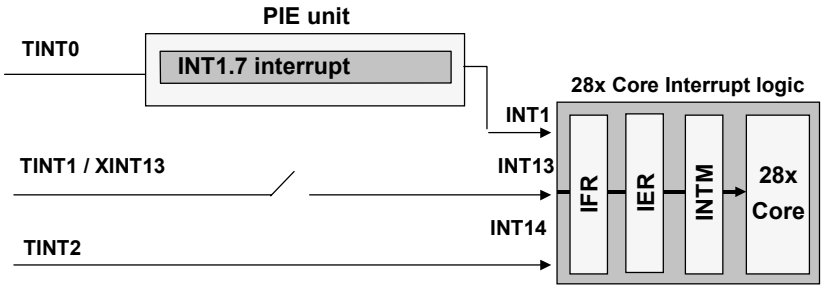
As you can see the clock source is the internal clock “SYSCLKOUT” which is usually 150MHz, assuming an external oscillator of 30MHz and a PLL-ratio of 10/2. Once the timer is enabled (TCR-Bit 4) the incoming clock counts down a 16-Bit prescaler (PSCH: PSC). On underflow, its borrow signal is used to 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 real time operation system “DSP/BIOS” whereas Timer 0 is free for general usage. Lab 4 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. Note: The Event Manager Timer T1, T2, T3 and T4 are explained later. Do not mix up the Core Timer group with the Event Manager (EVA and EVB)!

When the DSP comes out of RESET all 3-core timers are enabled.

C28x Timer Interrupt System

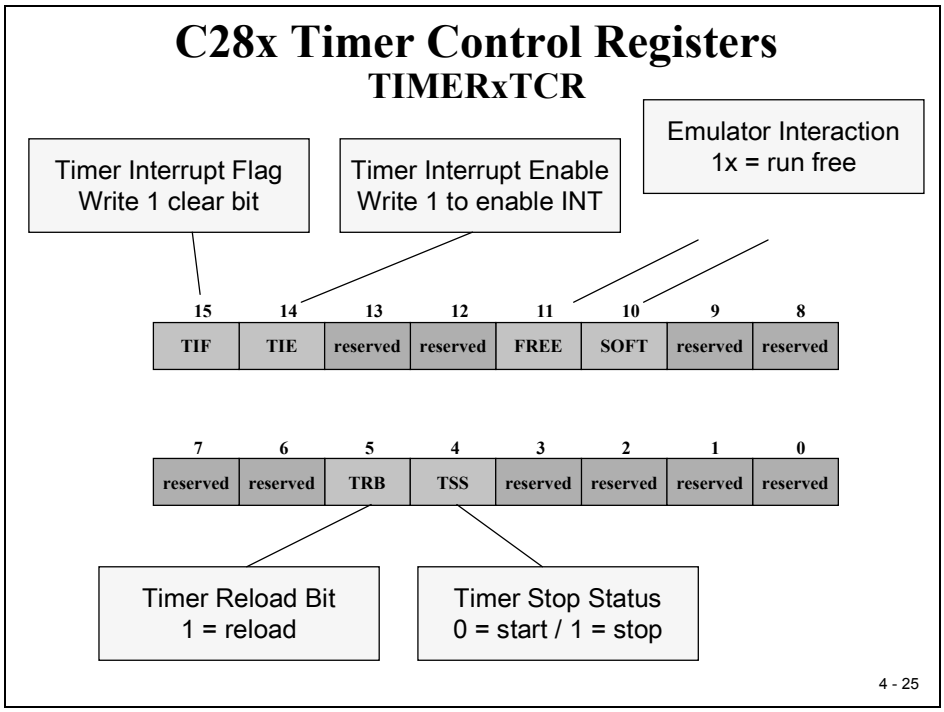


4 - 23

C28x 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	

4 - 24



Summary:

Sounds pretty complicated, doesn't it? Well, nothing is better suited to understand the PIE unit than a lab exercise. Lab 4 is asking you to add the initialization of the PIE vector table, to re-map the vector table to address 0x00 0D00 and to use CPU Timer 0 as a clock base for the source code of Lab 2 ("Knight Rider").

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

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

Take your time!

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

Lab 4: CPU Timer 0 Interrupt & 8 LED's

Objective

The objective of this lab is to include a basic example of the interrupt system into the "Knight Rider" project of Lab2. 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 show 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 with this unit as well. Timer 1 and 2 are bypassing 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

Open Files, Create Project File

1. Create a new project, called **Lab4.pjt** in E:\C281x\Labs.

2. Open the file Lab2.c from E:\C281x\Labs\Lab2 and save it as Lab4.c in E:\C281x\Labs\Lab4.

3. Add the source code file to your project:
   - **Lab4.c**

4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:
   - **DSP281x_GlobalVariableDefs.c**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:
   - **F2812_EzDSP_RAM_lnk.cmd**

   From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:
   - **F2812_Headers_nonBIOS.cmd**

   From *C:\ti\c2000\cgtoolslib* add:
   - **rts2800_ml.lib**

Modify Source Code

5. Open Lab4.c to edit: double click on "Lab4.c" inside the project window. At the start of your code add the function prototype statement for CPU Timer0 Interrupt Service:

   interrupt void cpu_timer0_isr(void);

6. Inside main, direct after the function call "Gpio_select()" add the 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'd like to initialize the PIE-unit. Function "InitPieCtrl ()" is defined in the source code file "DSP281x_PieCtrl.c"; we have to add this file to our project:

7. From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

   DSP281x_PieCtrl.c

8. Inside main, direct after the function call "InitPieCtrl();" add the function call to:

   InitPieVectTable();

   This TI-function will initialize the PIE-memory to an initial state. It uses a predefined interrupt table "PieVectTableInit()" – defined in source code file "DSP281x_PieVect.c" and copies this table to the global variable "PieVectTable" – defined in "DSP281x_GlobalVariableDefs.c". Variable "PieVectTable" is linked to the physical memory of the PIE area. To be able to use "InitPieVectTable" we have to add two more code files to our project:

9. From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

   DSP281x_PieVect.c and

   DSP281x_DefaultIsr.c

   Code file "DSP281x_DefaultIsr.c" will add a lot of interrupt service routines to our project. When you open and inspect this file you will find that all ISR's 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 would now run the program we should never see an interrupt request. If, for some reason like 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.

10. 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 "DSP281x_DefaultIsr.c" could do this. Of course this is not a good choice, because we'd modify the original code for this single Lab exercise. **SO DON'T DO THAT**!

A much better way is to modify the entry for CPU-Timer0 Interrupt Service directly inside the PIE-memory. This is done 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 and the PIE is part of this area. "cpu_timer0_isr" is the name of our own interrupt service routine for Timer0. We made the prototype statement earlier in the procedure of this Lab. Please be sure to use the same name as you used in the prototype statement!

11. Inside main, directly after the re-map instructions from procedure step 10 add the function call "InitCpuTimers();". This function will set the core Timer0 to a known state and it will stop this timer.

```
InitCpuTimers();
```

Again, we use a predefined TI-function. To do so, we have to add the source code file "DSP281x_CpuTimers.c" to our project.

12. From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

DSP281x_CpuTimers.c

13. Now we have to initialize Timer0 to generate a period of 50ms. 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. 50000 for 50 milliseconds. The following function call will setup Timer0 to a 50ms period:

```
ConfigCpuTimer(&CpuTimer0, 150, 50000);
```

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

14. Before we can start timer0 we have to enable its interrupt masks. We have to care about 3 levels to enable an individual interrupt source. Level 1 is the PIE unit. To enable we have to set bit 7 of PIEIER1 to 1. Why? Because the Timer0 interrupt is hard connected to group INT1, Bit7. Add the following line to your code:

```
PieCtrlRegs.PIEIER1.bit.INTx7 = 1;
```

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

16. Next, enable interrupts globally. This is done by adding the two macros:

    EINT; and

    ERTM;

17. Finally we have to start the timer 0. The bit TSS inside register TCR will do the job. Add:

    CpuTimer0Regs.TCR.bit.TSS = 0;

18. After the end of main we have to add our new interrupt service routine "cpu_timer0_isr". Remember, we've 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 50 milliseconds task was called.

    2nd – acknowledge the interrupt service as last line before return. This step is necessary to re-enable the next timer0 interrupt service. It is done by:

    PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;

19. 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".

20. 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 reached a predefined value, which is the multiple of 50 milliseconds. Setup a wait-statement for 150 milliseconds. Remember to reset the variable "CpuTimer0.InterruptCount" to zero when you continue after the wait statement.

21. Done?

22. 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 only 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.

Project Build Options

23. We need to setup the search path to include the peripheral register header files. Click:

    Project → Build Options

    Select the Compiler tab. In the preprocessor Category, find the Include Search Path (-i) box and enter:

    C:\tidcs\C28\dsp281x\v100\DSP281x_headers\include;..\include

24. Setup the stack size: Inside Build Options select the Linker tab and enter in the Stack Size (-stack) box:

    400

    Close the Build Options Menu by Clicking <**OK**>.

Build and Load

25. Click the "Rebuild All" button or perform:

    Project → Build

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

26. Load the output file down to the DSP Click:

    File → Load Program and choose the desired output file.

Test

27. Reset the DSP by clicking on:

    Debug → Reset CPU followed by
    Debug → Restart

28. Run the program until the first line of your C-code by clicking:

    Debug → Go main.

29. Debug your code as you've done in previous labs.