

Digital Input / Output

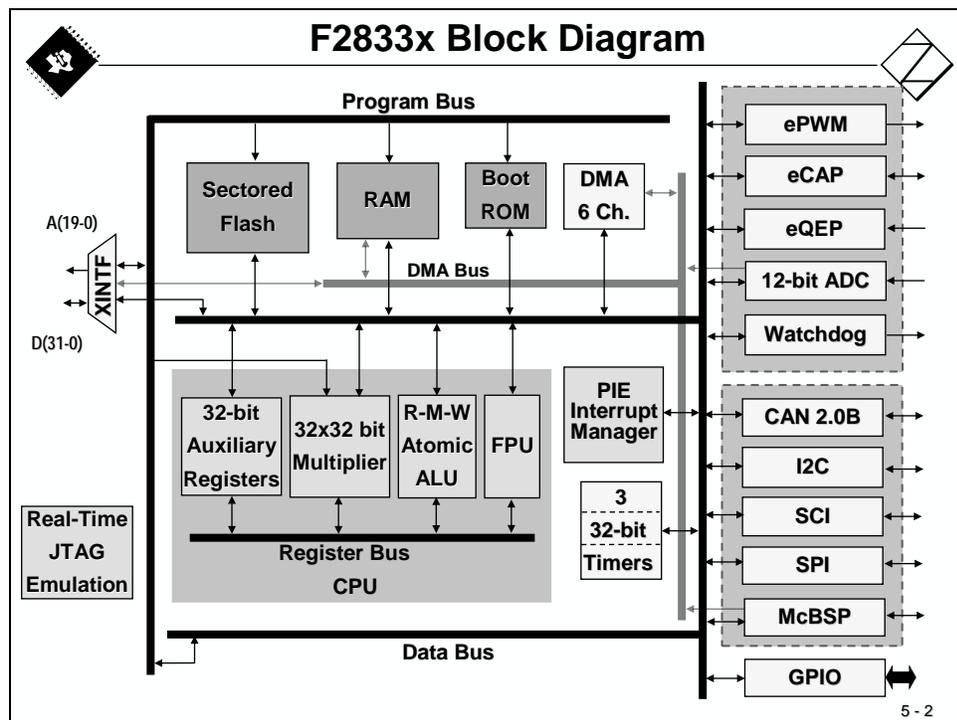
Introduction

This module introduces the first integrated peripherals of the F2833x Digital Signal Controller. The device has not only a 32-bit processor core, but also all of the peripheral units needed to build a single chip control system (SOC-“System on Chip”). These integrated peripherals give the F2833x an important advantage over other processors.

We will start with the simplest peripheral unit-Digital I/O. At the end of this chapter we will exercise input lines (switches, buttons) and output lines (LEDs).

Data Memory Mapped Peripherals

All the peripheral units of the F2833x are memory mapped into the data memory space of its Harvard Architecture Machine. This means that we control peripheral units by accessing dedicated data memory addresses. The following slide shows these units:

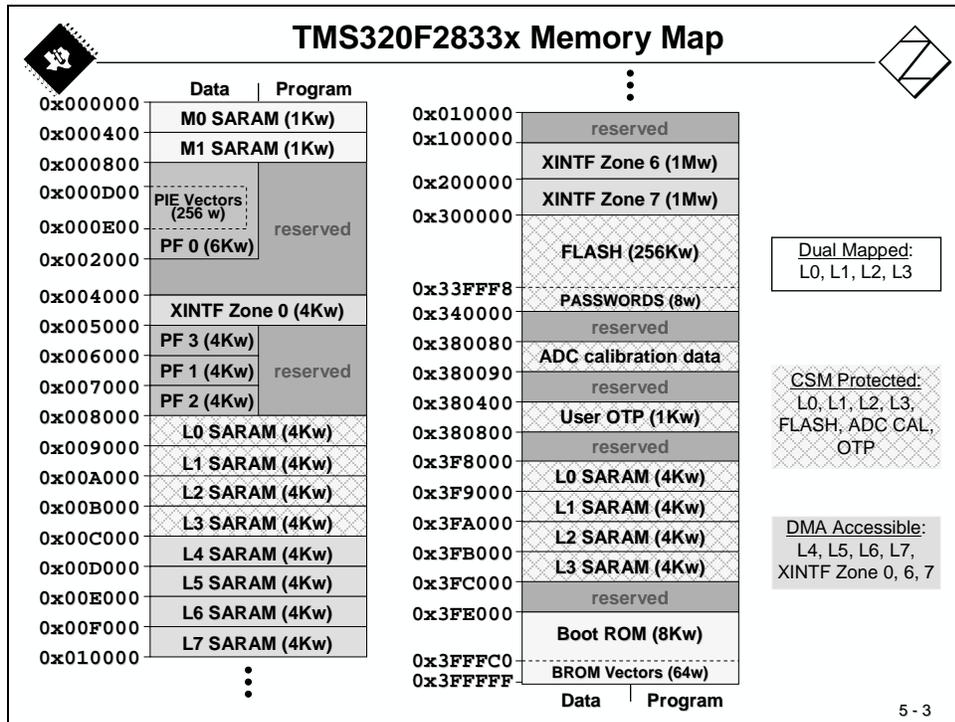


Module Topics

Digital Input / Output.....	5-1
<i>Introduction.....</i>	<i>5-1</i>
<i>Data Memory Mapped Peripherals.....</i>	<i>5-1</i>
<i>Module Topics.....</i>	<i>5-2</i>
<i>The Peripheral Frames</i>	<i>5-3</i>
<i>Digital I/O Unit.....</i>	<i>5-5</i>
F2833x Pin Assignment.....	5-7
GPIO Input Qualification.....	5-10
Summary GPIO-Registers	5-11
<i>F2833x Clock Module</i>	<i>5-12</i>
<i>Watchdog Timer.....</i>	<i>5-14</i>
<i>System Control and Status Register</i>	<i>5-17</i>
<i>Low Power Mode</i>	<i>5-17</i>
<i>Lab 5_1: Digital Output at 4 LEDs.....</i>	<i>5-20</i>
Objective.....	5-21
Procedure	5-21
Create a Project File.....	5-21
Project Build Options.....	5-22
Modify the Source Code	5-23
Setup the control loop	5-23
Build and Load.....	5-24
Test	5-24
Enable Watchdog Timer	5-24
Service the Watchdog Timer.....	5-25
<i>Lab 5_2: Digital Output (modified).....</i>	<i>5-27</i>
Procedure	5-27
Modify Code and Project File.....	5-27
<i>Lab 5_3: Digital Input.....</i>	<i>5-28</i>
Objective.....	5-28
Procedure	5-28
Modify Code and Project File.....	5-28
Build, Load and Test.....	5-29
<i>Lab 5_4: Digital In- and Output</i>	<i>5-30</i>
Objective.....	5-30
Modify Code and Project File.....	5-30
Modify Lab5_4.C.....	5-30
Build, Load and Test.....	5-31
<i>Lab 5_5: Digital In- and Output Start / Stop</i>	<i>5-32</i>
Objective.....	5-32
Modify Code and Project File.....	5-32
Modify Lab5_5.c	5-32
Build, Load and Test.....	5-33

The Peripheral Frames

All peripheral registers are grouped together into what are known as “Peripheral Frames”- PF0, PF1, PF2 and PF3. These frames are mapped in data memory only. Peripheral Frame PF0 includes register sets to control the internal speed of the FLASH memory, as well as the timing setup for external memory devices, direct memory access unit registers, core CPU timer registers and the code security module control block. Flash is internal non-volatile memory, usually used for code storage and for data that must be present at boot time. Peripheral Frame PF1 contains most of the peripheral unit control registers, such as ePWM, eCAP, Digital Input/Output control and the CAN register block. CAN-“Controller Area Network” is a well-established network widely used inside motor vehicles to build a network between electronic control units (ECU). Peripheral Frame PF2 combines the core system control registers, the Analogue to Digital Converter and all other communication channels other than McBSP, which has been allocated to PF3.



The detailed mapping of peripherals into data memory is as follows:

PF0:	PIE:	PIE Interrupt Enable and Control Registers plus PIE Vector Table
	Flash:	Flash Wait state Registers
	XINTF:	External Interface Registers
	DMA:	DMA Registers
	Timers:	CPU-Timers 0, 1, 2 Registers
	CSM:	Code Security Module KEY Registers
	ADC:	ADC Result registers (dual-mapped)

PF1:	eCAN:	eCAN Mailbox and Control Registers
	GPIO:	GPIO MUX Configuration and Control Registers
	ePWM:	Enhanced Pulse Width Modulator Module and Registers (dual mapped)
	eCAP:	Enhanced Capture Module and Registers
	eQEP:	Enhanced Quadrature Encoder Pulse Module and Registers

PF2:	SYS:	System Control Registers
	SCI:	Serial Communications Interface (SCI) Control and RX/TX Registers
	SPI:	Serial Port Interface (SPI) Control and RX/TX Registers
	ADC:	ADC Status, Control, and Result Register
	I2C:	Inter-Integrated Circuit Module and Registers
	XINT:	External Interrupt Registers

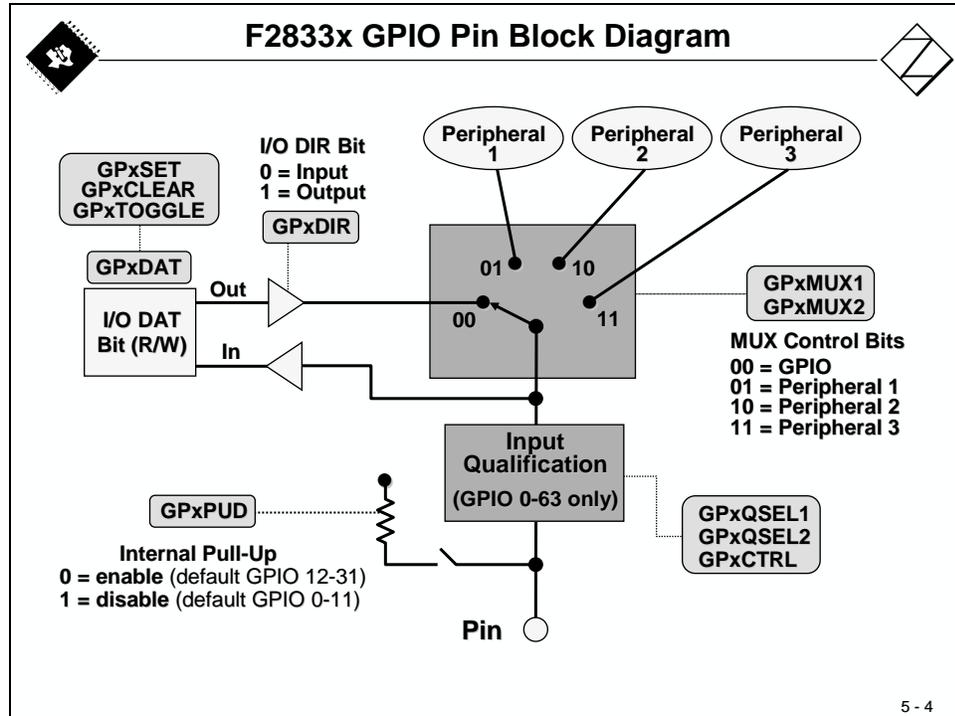
PF3:	McBSP:	Multichannel Buffered Serial Port Registers
	ePWM:	Enhanced Pulse Width Modulator Module and Registers (dual mapped)

Some of the memory areas are password protected by the “Code Security Module” (check patterned areas of the slide above). This is a feature to prevent reverse engineering. Once the password area is programmed, any access to the secured areas is only granted when the correct password is entered into a special area of PF0.

Now let us start with a discussion of the Digital Input/Output unit.

Digital I/O Unit

All digital I/O's are grouped together into "Ports", called GPIO-A, B and C. Here GPIO means "general purpose input output". The F2833x features a total of 88 I/O-pins, called GPIO0 to GPIO87. But there's more. The device comes with so many additional internal units, that not all features could be connected to dedicated pins of the device package at any one time. The solution is: multiplex. This means, one single physical pin of the device can be used for up to 4 different functions and it is up to the programmer to decide which function is selected. The next slide shows a block diagram of one physical pin of the device:



The term "Input Qualification" refers to an additional option for digital input signals at GPIO0-63. When this feature is used, an input pulse must be longer than the specified number of clock cycles to be recognized as a valid input signal. This is useful for removing input noise.

Register Group "GPxPUD" can be used to disable internal pull-up resistors to leave the voltage level floating or high-impedance.

When a digital I/O function is selected, then register group GPxDIR defines the direction of the Input or Output. Clearing a bit position to zero configures the line as an input, setting the bit position to 1 configures the line as an output.

A data read from an input line is performed with a set of GPxDAT registers.

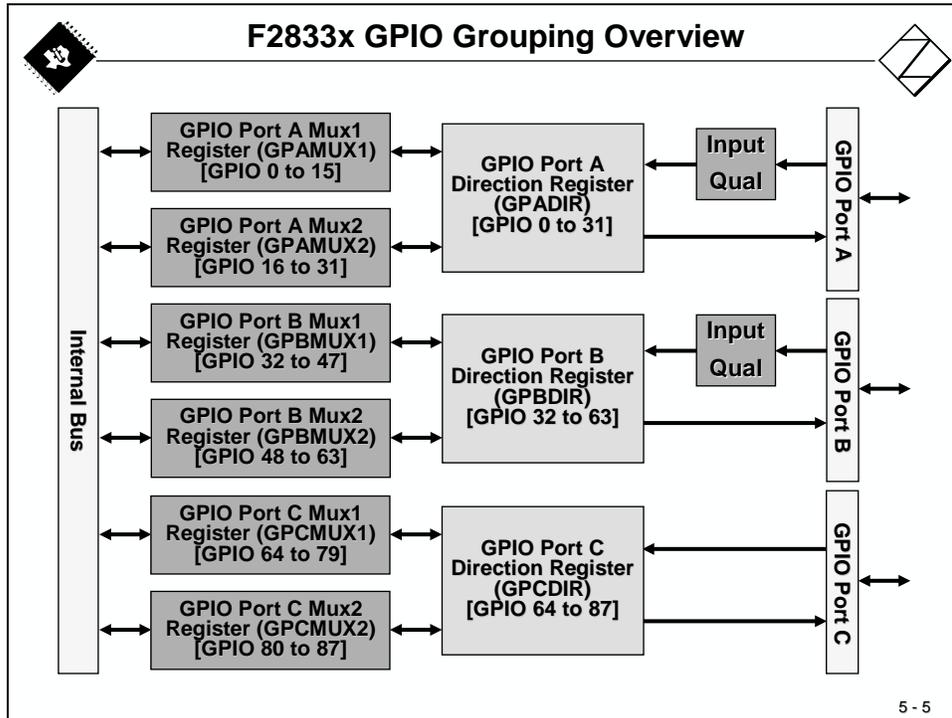
A data write to an output line can also be performed with registers GPxDAT. Additionally, there are 3 more groups of registers:

- GPxSET
- GPxCLEAR
- GPxTOGGLE

The objective of these registers is to use a mask technique to set, clear or toggle those output lines, which correspond to a bit set to 1 in the mask in use. For example, to clear line GPIO5 to 0, one can use the instruction:

- `GpioDataRegs.GPACLEAR.bit.GPIO5 = 1;`

The following slide summarizes the I/O control register set:



F2833x Pin Assignment

The next five slides show the multiplex assignment for all 88 I/O-lines:

F2833x GPIO Pin Assignment				
GPIO - A Multiplex Register GPAMUX1				
GPAMUX1 - Bits	00	01	10	11
1,0	GPIO0	EPWM1A	-	-
3,2	GPIO1	EPWM1B	ECAP6	MFSRB
5,4	GPIO2	EPWM2A	-	-
7,6	GPIO3	EPWM2B	ECAP5	MCLKRB
9,8	GPIO4	EPWM3A	-	-
11,10	GPIO5	EPWM3B	MFSRA	ECAP1
13,12	GPIO6	EPWM4A	EPWMSYNCl	EPWMSYNCO
15,14	GPIO7	EPWM4B	MCLKRA	ECAP2
17,16	GPIO8	EPWM5A	CANTXB	/ADCSOCA0
19,18	GPIO9	EPWM5B	SCITXDB	ECAP3
21,20	GPIO10	EPWM6A	CANRXB	/ADCSOCB0
23,22	GPIO11	EPWM6B	SCIRXDB	ECAP4
25,24	GPIO12	/TZ1	CANTXB	SPISIMOB
27,26	GPIO13	/TZ2	CANRXB	SPISOMIB
29,28	GPIO14	/TZ3_/XHOLD	SCITXDB	SPICLKB
31,30	GPIO15	/TZ4_/XHOLDA	SCIRXDB	/SPISTEB

5 - 6

F2833x GPIO Pin Assignment				
GPIO - A Multiplex Register GPAMUX2				
GPAMUX2 - Bits	00	01	10	11
1,0	GPIO16	SPISIMOA	CANTXB	/TZ5
3,2	GPIO17	SPISOMIA	CANRXB	/TZ6
5,4	GPIO18	SPICLKA	SCITXDB	CANRXA
7,6	GPIO19	/SPISTEA	SCIRXDB	CANTXA
9,8	GPIO20	EQEP1A	MDXA	CANTXB
11,10	GPIO21	EQEP1B	MDRA	CANRXB
13,12	GPIO22	EQEP1S	MCLKXA	SCITXDB
15,14	GPIO23	EQEP1I	MFSXA	SCIRXDB
17,16	GPIO24	ECAP1	EQEP2A	MDXB
19,18	GPIO25	ECAP2	EQEP2B	MDRB
21,20	GPIO26	ECAP3	EQEP2I	MCLKXB
23,22	GPIO27	ECAP4	EQEP2S	MFSXB
25,24	GPIO28	SCIRXDA	/XZCS6	/XZCS6
27,26	GPIO29	SCITXDA	XA19	XA19
29,28	GPIO30	CANRXA	XA18	XA18
31,30	GPIO31	CANTXA	XA17	XA17

5 - 7

F2833x GPIO Pin Assignment

GPIO - B Multiplex Register GPBMUX1

GPBMUX1 - Bits	00	01	10	11
1,0	GPIO32	SDAA	EPWMSYNCI	/ADCSOCA0
3,2	GPIO33	SCLA	EPWMSYNCO	/ADCSOCB0
5,4	GPIO34	ECAP1	XREADY	XREADY
7,6	GPIO35	SCITXDA	XR/W	XR/W
9,8	GPIO36	SCIRXDA	/XZCS0	/XZCS0
11,10	GPIO37	ECAP2	/XZCS7	/XZCS7
13,12	GPIO38	-	/XWE0	/XWE0
15,14	GPIO39	-	XA16	XA16
17,16	GPIO40	-	XA0/XWE1	XA0/XWE1
19,18	GPIO41	-	XA1	XA1
21,20	GPIO42	-	XA2	XA2
23,22	GPIO43	-	XA3	XA3
25,24	GPIO44	-	XA4	XA4
27,26	GPIO45	-	XA5	XA6
29,28	GPIO46	-	XA6	XA6
31,30	GPIO47	-	XA7	XA7

5 - 8

F2833x GPIO Pin Assignment

GPIO - B Multiplex Register GPBMUX2

GPBMUX2 - Bits	00	01	10	11
1,0	GPIO48	ECAP5	XD31	XD31
3,2	GPIO49	ECAP6	XD30	XD30
5,4	GPIO50	EQEP1A	XD29	XD29
7,6	GPIO51	EQEP1B	XD28	XD28
9,8	GPIO52	EQEP1S	XD27	XD27
11,10	GPIO53	EQEP1I	XD26	XD26
13,12	GPIO54	SPISIMOA	XD25	XD25
15,14	GPIO55	SPISOMIA	XD24	XD24
17,16	GPIO56	SPICLKA	XD23	XD23
19,18	GPIO57	/SPISTEAA	XD22	XD22
21,20	GPIO58	MCLKRA	XD21	XD21
23,22	GPIO59	MFSRA	XD20	XD20
25,24	GPIO60	MCLKRB	XD19	XD19
27,26	GPIO61	MFSRB	XD18	XD18
29,28	GPIO62	SCIRXDC	XD17	XD17
31,30	GPIO63	SCITXDC	XD16	XD16

5 - 9



F2833x GPIO Pin Assignment



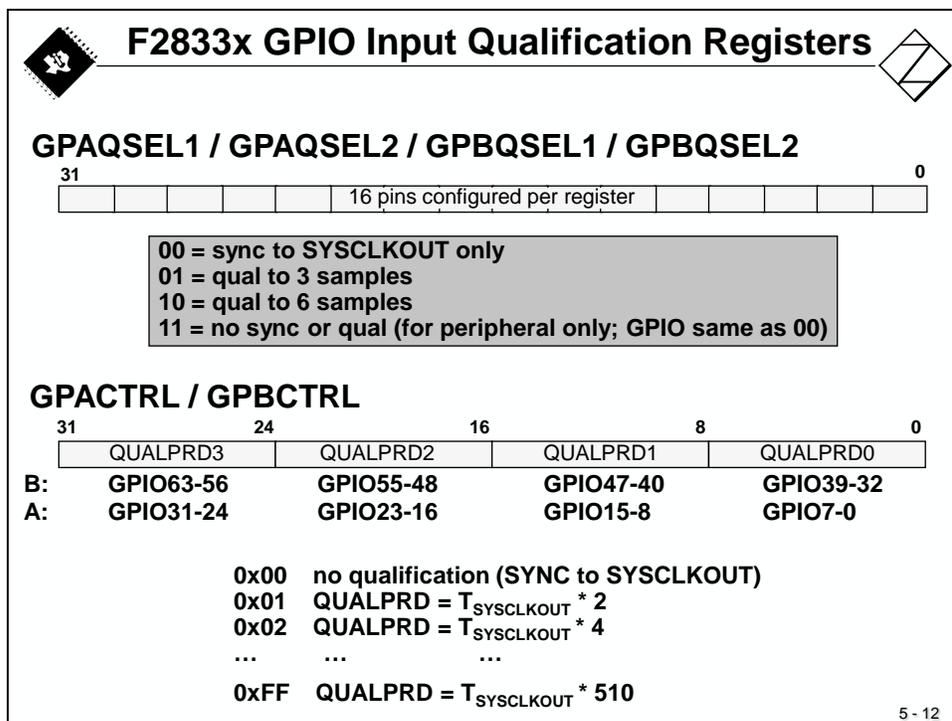
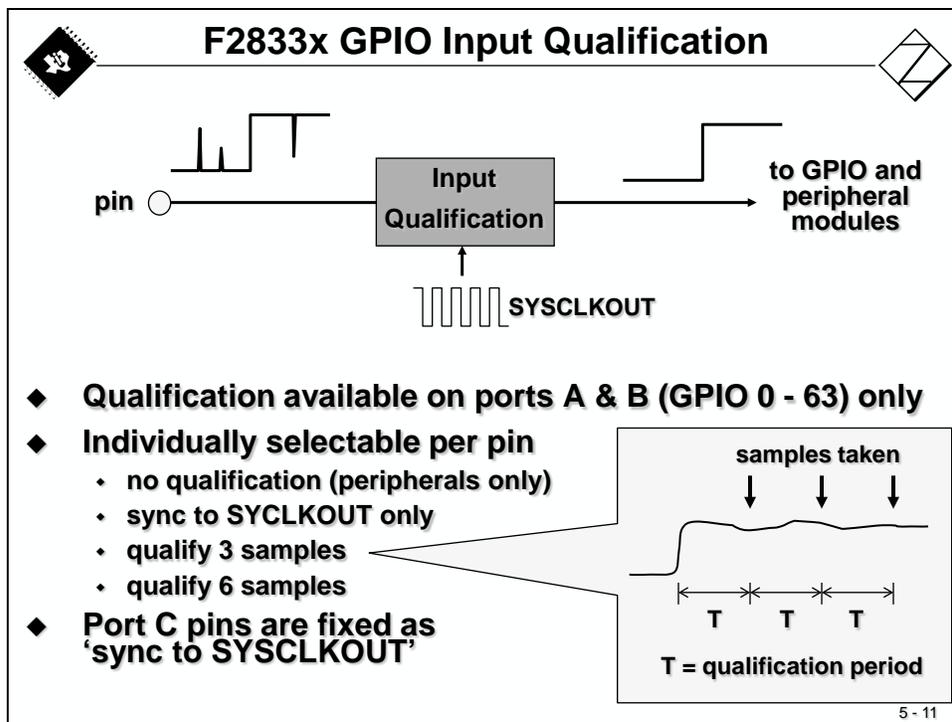
GPIO - C Multiplex Register

GPCMUX1 - Bits	00 or 01	10 or 11	GPCMUX2 - Bits	00 or 01	10 or 11
1,0	GPIO64	XD15	1,0	GPIO80	XA8
3,2	GPIO65	XD14	3,2	GPIO81	XA9
5,4	GPIO66	XD13	5,4	GPIO82	XA10
7,6	GPIO67	XD12	7,6	GPIO83	XA11
9,8	GPIO68	XD11	9,8	GPIO84	XA12
11,10	GPIO69	XD10	11,10	GPIO85	XA13
13,12	GPIO70	XD9	13,12	GPIO86	XA14
15,14	GPIO71	XD8	15,14	GPIO87	XA15
17,16	GPIO72	XD7	17,16	-	-
19,18	GPIO73	XD6	19,18	-	-
21,20	GPIO74	XD5	21,20	-	-
23,22	GPIO75	XD4	23,22	-	-
25,24	GPIO76	XD3	25,24	-	-
27,26	GPIO77	XD2	27,26	-	-
29,28	GPIO78	XD1	29,28	-	-
31,30	GPIO79	XD0	31,30	-	-

5 - 10

GPIO Input Qualification

As has already been stated, this feature on GPIO0-63 behaves like a low-pass input filter on noisy input signals. It is controlled by a pair of additional registers.



Summary GPIO-Registers

The next two slides will summarize all registers of the GPIO-unit.

C2833x GPIO Control Registers	
Register	Description
GPACTRL	GPIO A Control Register [GPIO 0 – 31]
GPAQSEL1	GPIO A Qualifier Select 1 Register [GPIO 0 – 15]
GPAQSEL2	GPIO A Qualifier Select 2 Register [GPIO 16 – 31]
GPAMUX1	GPIO A Mux1 Register [GPIO 0 – 15]
GPAMUX2	GPIO A Mux2 Register [GPIO 16 – 31]
GPADIR	GPIO A Direction Register [GPIO 0 – 31]
GPAPUD	GPIO A Pull-Up Disable Register [GPIO 0 – 31]
GPBCTRL	GPIO B Control Register [GPIO 32 – 63]
GPBQSEL1	GPIO B Qualifier Select 1 Register [GPIO 32 – 47]
GPBQSEL2	GPIO B Qualifier Select 2 Register [GPIO 48 – 63]
GPBMUX1	GPIO B Mux1 Register [GPIO 32 – 47]
GPBMUX2	GPIO B Mux2 Register [GPIO 48 – 63]
GPBDIR	GPIO B Direction Register [GPIO 32 – 63]
GPBPUD	GPIO B Pull-Up Disable Register [GPIO 32 – 63]
GPCMUX1	GPIO C Mux1 Register [GPIO 64 – 79]
GPCMUX2	GPIO C Mux2 Register [GPIO 80 – 87]
GPCDIR	GPIO C Direction Register [GPIO 64 – 87]
GPCPUD	GPIO C Pull-Up Disable Register [GPIO 64 – 87]

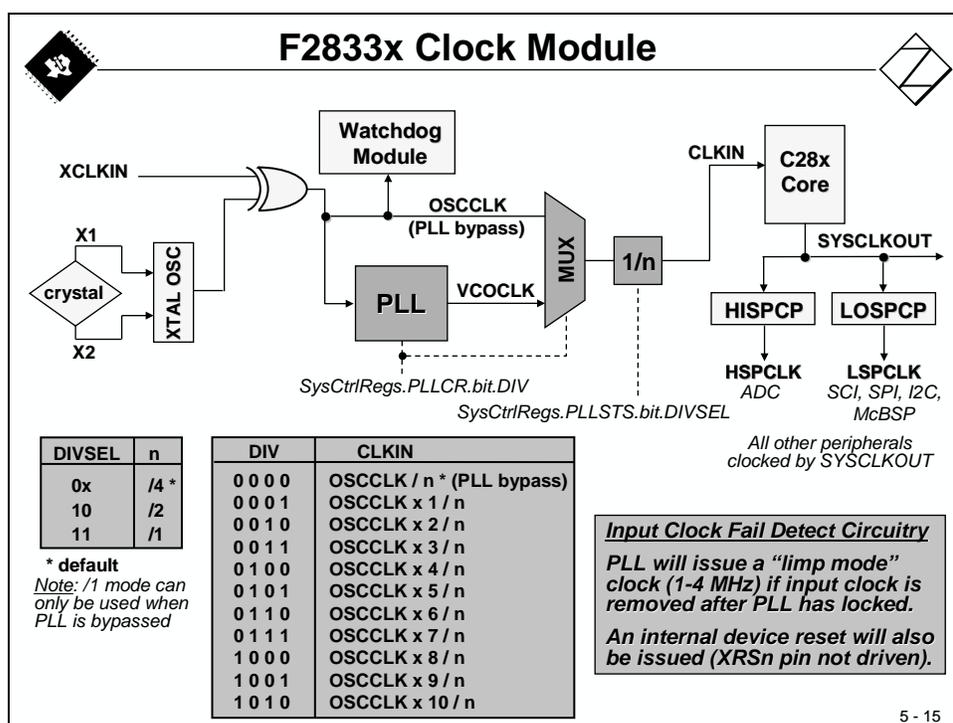
5 - 13

C2833x GPIO Data Registers	
Register	Description
GPADAT	GPIO A Data Register [GPIO 0 – 31]
GPASET	GPIO A Data Set Register [GPIO 0 – 31]
GPACLEAR	GPIO A Data Clear Register [GPIO 0 – 31]
GPATOGGLE	GPIO A Data Toggle [GPIO 0 – 31]
GPBDAT	GPIO B Data Register [GPIO 32 – 63]
GPBSET	GPIO B Data Set Register [GPIO 32 – 63]
GPBCLEAR	GPIO B Data Clear Register [GPIO 32 – 63]
GPBTOGGLE	GPIO B Data Toggle [GPIO 32 – 63]
GPCDAT	GPIO C Data Register [GPIO 64 – 87]
GPCSET	GPIO C Data Set Register [GPIO 64 – 87]
GPCCLEAR	GPIO C Data Clear Register [GPIO 64 – 87]
GPCTOGGLE	GPIO C Data Toggle [GPIO 64 – 87]

5 - 14

F2833x Clock Module

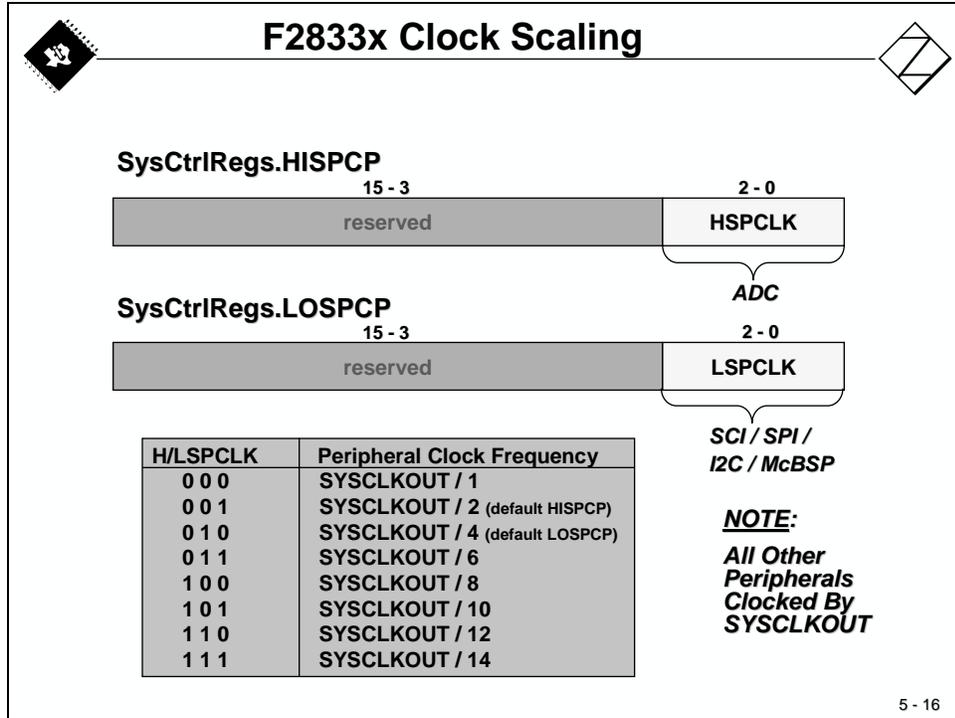
Before we can start using the digital I/Os, we need to setup the F2833x Clock Module. Like all modern processors, the F2833x is driven externally by a much slower clock generator or oscillator to reduce electromagnetic interference. An internal PLL circuit generates the internal speed. The F28335 ControlCard in our Labs is running at 20MHz externally. To achieve the internal frequency of 100 MHz, we have to use the multiply by a factor of 10, followed by a divide by 2. This is implemented by programming the PLL control register (PLLCR).



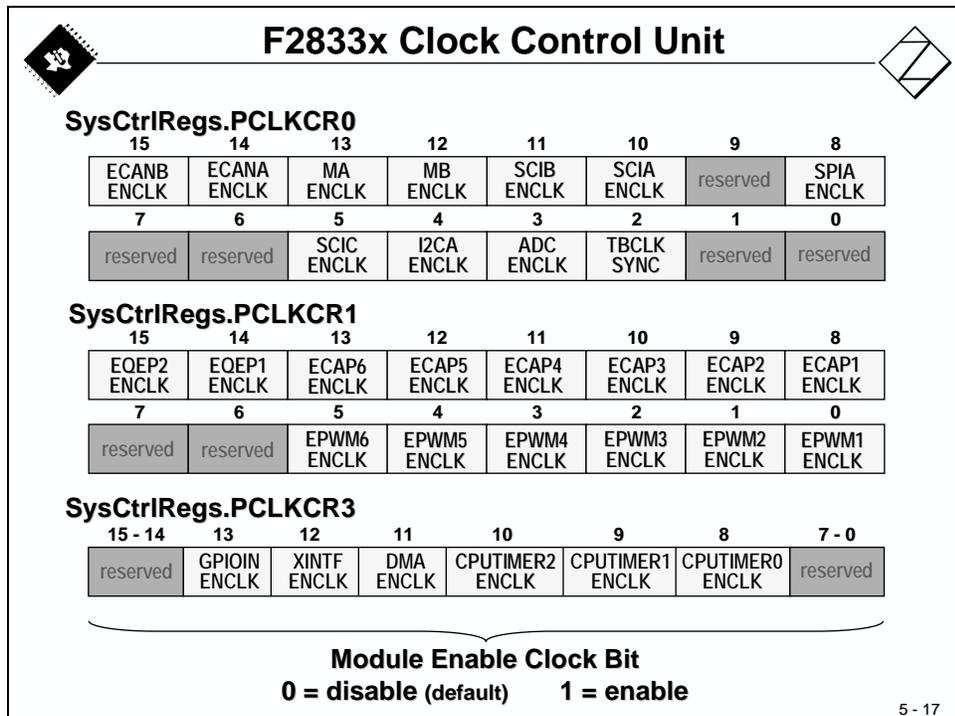
High-speed Clock Pre-scaler (HISPCP) and Low speed Clock Pre-scaler (LOSPCP) are used as additional clock dividers. The outputs of the two pre-scalers are used as the clock source for the peripheral units. We can set up the two pre-scalers individually and independently.

Note that:

- (1) the signal "CLKIN" is of the same frequency as the core output signal "SYSCLKOUT", which is used for the external memory interface, for clocking the ePWMs and the CAN-unit.
- (2) the Watchdog Unit is clocked directly by the external oscillator.
- (3) the maximum frequency for the external oscillator is 35MHz.



To use a peripheral unit, we have to enable its clock distribution by setting individual bit fields of the PCLKCRx register. Bit field “GPIOIN_ENCLK” enables the clock signal for the input qualification filter. If input qualification is not used, then it is not necessary to enable this bit.



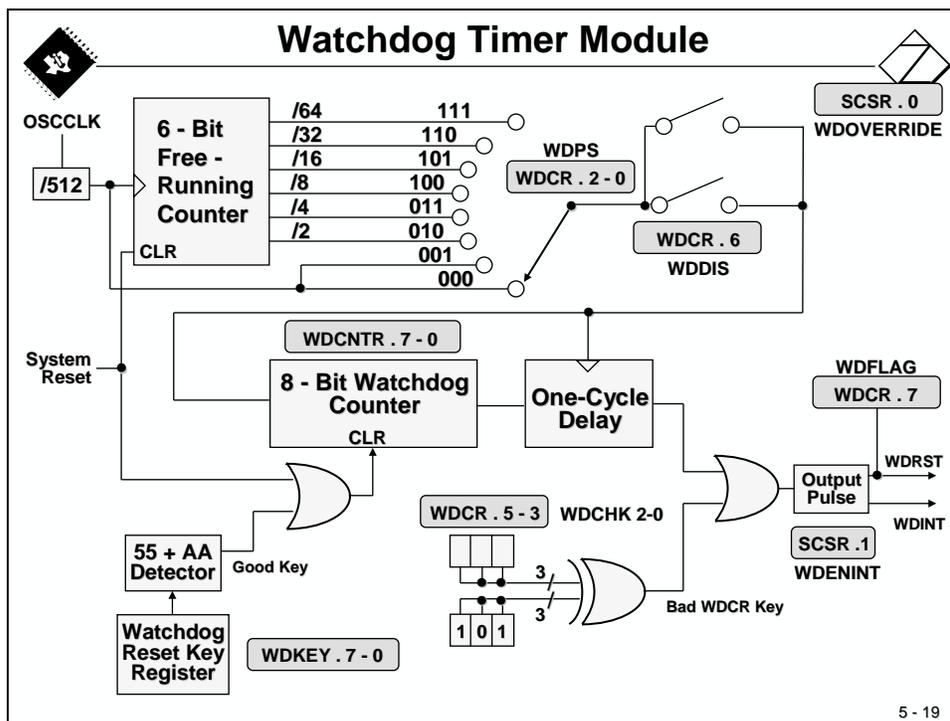
Watchdog Timer

A “Watchdog Timer” is a free running counter unit that triggers a reset if it is not cleared periodically by a specific instruction sequence. It is used to recognize events where the program leaves its designated sequence of execution, for example, if the program crashes.

Watchdog Timer

- ◆ **Resets the F2833x if the CPU crashes**
 - ◆ Watchdog counter runs independent of CPU
 - ◆ If counter overflows, a reset or interrupt is triggered (user selectable)
 - ◆ CPU must write correct data key sequence to reset the counter before overflow
- ◆ **Watchdog must be serviced or disabled within 4.37ms after reset (assuming a 30 MHz OSCCLK)**
- ◆ **This time period translates into 645000 instructions, if CPU runs at 150MHz!**

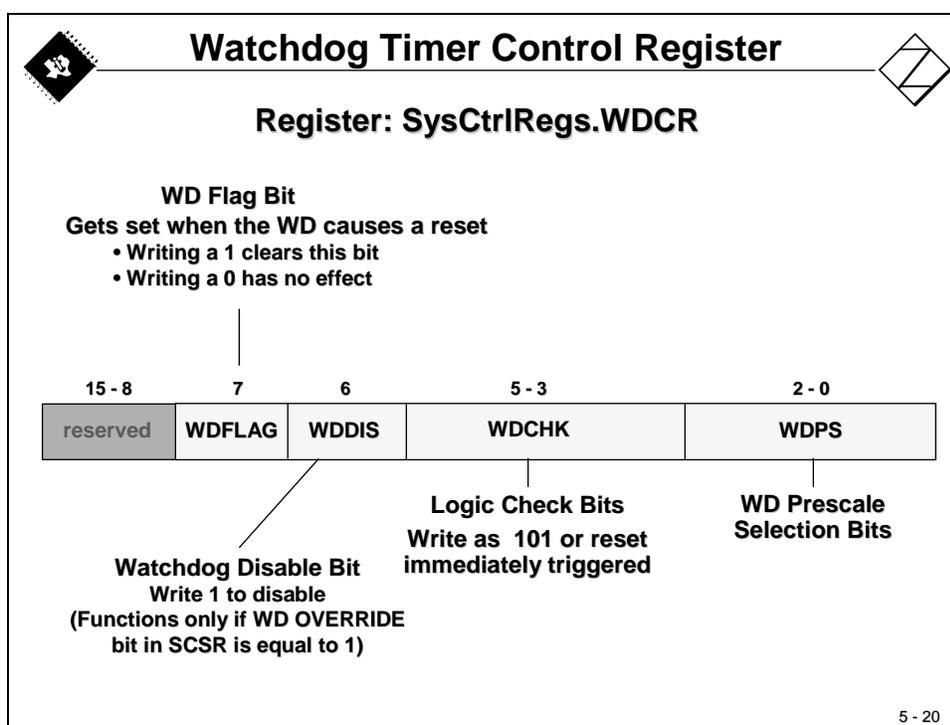
5 - 18



The Watchdog is always alive when the DSP is powered up! When we do not take care of the Watchdog periodically, it will trigger a RESET. One of the simplest methods to deal with the Watchdog is to disable it. This is done by setting bit 6 of register WDCR to 1. Of course this is not a wise decision, because a Watchdog is a security feature and a real project should always include as much security as possible or available.

The Watchdog Pre-scaler can be used to increase the Watchdog's overflow period. The Logic Check Bits (WDCHK) is another security bit field. All write accesses to the register WDCR must include the bit combination "101" for this 3 bit field, otherwise the access is denied and a RESET is triggered immediately.

The Watchdog Flag Bit (WDFLAG) can be used to distinguish between a normal power on RESET (WDFLAG = 0) and a Watchdog RESET (WDFLAG = 1). NOTE: To clear this flag by software, we have to write a '1' into this bit!



Note: if for some reason the external oscillator clock fails, the Watchdog stops incrementing. In an application we can catch this condition by reading the Watchdog counter register periodically. In the case of a lost external clock, this register will not increment any longer. The F2833x itself will still execute if in PLL mode, since the PLL will output a clock between 1 and 4 MHz in a so-called "limp"-mode.

How do we clear the Watchdog counter register, before it overflows? Answer: By writing a “valid key” or “good key” sequence into register WDKEY:

Resetting the Watchdog

- ◆ **WDKEY write values:**
 - 0x55 - counter enabled for reset on next 0xAA write
 - 0xAA - counter set to zero if reset enabled
- ◆ **Writing any other value has no effect**
- ◆ **Watchdog should not be serviced solely in an ISR**
 - ◆ If main code crashes, but interrupt continues to execute, the watchdog will not catch the crash
 - ◆ Could put the 0x55 WDKEY in the main code, and the 0xAA WDKEY in an ISR; this catches main code crashes and also ISR crashes

5 - 21

WDKEY Write Results

Sequential Step	Value Written to WDKEY	Result
1	AAh	No action
2	AAh	No action
3	55h	WD counter enabled for reset on next AAh write
4	55h	WD counter enabled for reset on next AAh write
5	55h	WD counter enabled for reset on next AAh write
6	AAh	WD counter is reset
7	AAh	No action
8	55h	WD counter enabled for reset on next AAh write
9	AAh	WD counter is reset
10	55h	WD counter enabled for reset on next AAh write
11	23h	No effect; WD counter not reset on next AAh write
12	AAh	No action due to previous invalid value
13	55h	WD counter enabled for reset on next AAh write
14	AAh	WD counter is reset

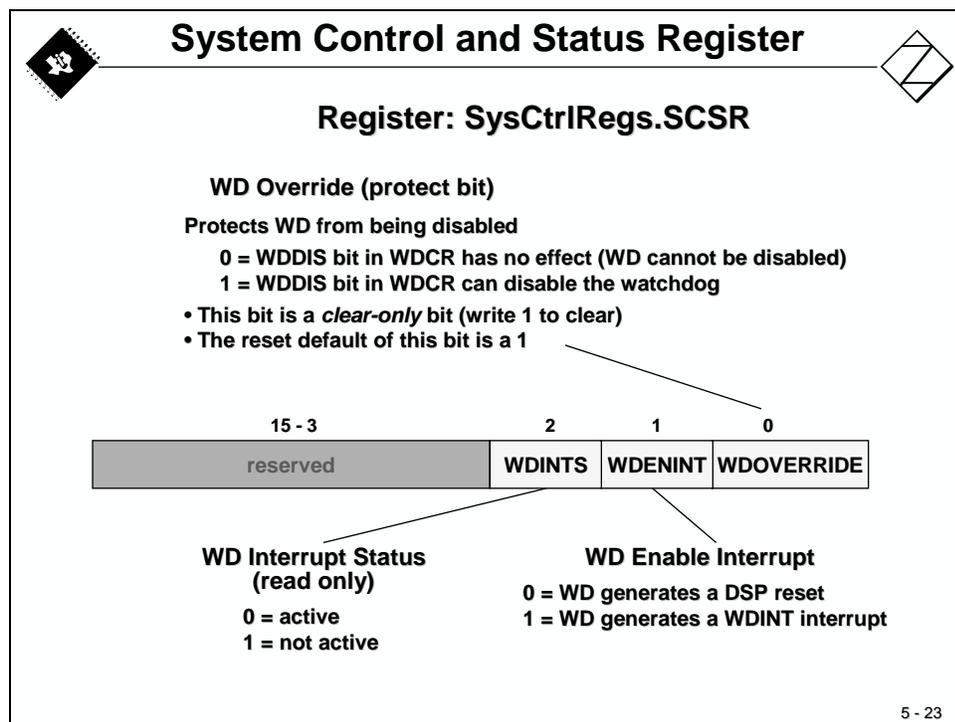
5 - 22

System Control and Status Register

Register SCSR controls whether the Watchdog causes a RESET (WDENINT = 0) or an Interrupt Service Request (WDENINT = 1). The default state after RESET is to trigger a RESET.

The WDOVERRIDE bit is a “clear only” bit, that means, once we have closed this switch by writing a 1 into the bit, we cannot re-open this switch again (see block diagram of the Watchdog). At this point the WD-disable bit is ineffectual, so there is no way to disable the Watchdog!

Bit 2 (WDINTS) is a read only bit that flags the status of the Watchdog Interrupt.



Low Power Mode

To reduce power consumption, the F2833x is able to switch into 3 different low-power operating modes. We will not use this feature in this chapter; therefore we can treat the Low Power Mode control bits as “don’t care”. The Low Power Mode is entered by execution of the dedicated Assembler Instruction “IDLE”. As long as we do not execute this instruction, the initialization of the LPMCR0 register has no effect.

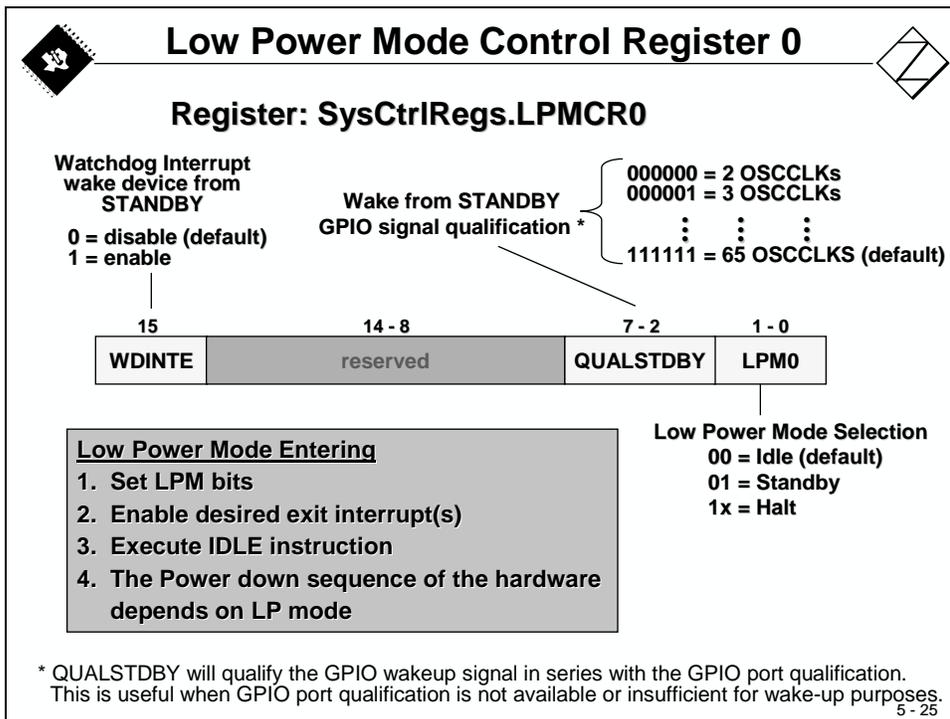
The next four slides explain the Low Power Modes in detail.

Low Power Modes

Low Power Mode	CPU Logic Clock	Peripheral Logic Clock	Watchdog Clock	PLL / OSC
Normal Run	on	on	on	on
IDLE	off	on	on	on
STANDBY	off	off	on	on
HALT	off	off	off	off

See device datasheet for power consumption in each mode

5 - 24



Low Power Mode Exit

Exit Interrupt Low Power Mode	RESET or XNMI	GPIO Port A Signal	Watchdog Interrupt	Any Enabled Interrupt
IDLE	yes	yes	yes	yes
STANDBY	yes	yes	yes	no
HALT	yes	yes	no	no

5 - 26

GPIO Low Power Wakeup Select

Register: SysCtrlRegs.GPIOLPMSSEL

31	30	29	28	27	26	25	24
GPIO31	GPIO30	GPIO29	GPIO28	GPIO27	GPIO26	GPIO25	GPIO24
23	22	21	20	19	18	17	16
GPIO23	GPIO22	GPIO21	GPIO20	GPIO19	GPIO18	GPIO17	GPIO16
15	14	13	12	11	10	9	8
GPIO15	GPIO14	GPIO13	GPIO12	GPIO11	GPIO10	GPIO9	GPIO8
7	6	5	4	3	2	1	0
GPIO7	GPIO6	GPIO5	GPIO4	GPIO3	GPIO2	GPIO1	GPIO0

Wake device from
 HALT and STANDBY mode
 (GPIO Port A)
 0 = disable (default)
 1 = enable

5 - 27

Lab 5_1: Digital Output at 4 LEDs



Lab 5_1: “Binary Counter” at 4 LEDs



Objective:

- Display the 4 least significant bits of a counter variable at LED LD1(GPIO9), LD2(GPIO11), LD3(GPIO34) and LD4(GPIO49) of the Peripheral Explorer Board.
- Increment variable “counter” every 100 milliseconds
- Use a software delay loop to generate the interval of 100 milliseconds

Project - Files :

1. C - source file “Lab5_1.c”
2. Start assembly code file: “DSP2833x_CodeStartBranch.asm”
2. Register Variable Definition File: “DSP2833x_GlobalVariableDefs.c”
3. Linker Command File: “28335_RAM_Ink.cmd”
“DSP2833x_Headers_nonBIOS.cmd”
4. Runtime Library “rts2800_fpu32.lib”

5 - 28



“DSP2833x_GlobalVariableDefs.c”



- Definition of global variables for all memory mapped peripheral registers based on predefined structures
- Master Header File is “DSP2833x_Device.h”
- Example GpioDataRegs:


```
volatile struct GPIO_DATA_REGS GpioDataRegs;
```
- This structure variable combines all registers, which belong to this peripheral group, e.g.:


```
GpioDataRegs.GPADAT
```
- Each register is declared as a union to allow 32-bit- (“all”) and single bit field -accesses (“bit”), e.g.:


```
GpioDataRegs.GPADAT.bit.GPIO9 = 1;
GpioDataRegs.GPADAT.all = 0x0000FFFF;
```
- Steps to be done are:
 1. Add “DSP2833x_GlobalVariableDefs.c” to project
 2. Include “DSP2833x_Device.h” into your C-code

5 - 29



“Lab 5_1 Register usage”



Registers involved in LAB 5_1:

- **Core Initialisation:**
 - Watchdog - Timer - Control : WDCR
 - PLL Clock Register : PLLCR
 - High Speed Clock Pre-scaler: HISPCP
 - Low Speed Clock Pre-scaler : LOSPCP
 - Peripheral Clock Control : PCLKCRx
 - System Control and Status : SCSR
- **Access to LED's (GPIO9, GPIO11,GPIO34,GPIO49):**
 - GPA and GPB Multiplex Register:
 - GPAMUX1, GPAMUX2, GPBMUX1, GPBMUX2
 - GPA and GPB Direction Register:
 - GPADIR and GPBDIR
 - GPA and GPB Data Register:
 - GPASET, GPACLEAR, GPBSET, GPBCLEAR

5 - 31

Objective

The objective of this lab is to practice using basic digital I/O-operations. GPIO9, GPIO11, GPIO34 and GPIO49 are connected to 4 Leds (LD1-4) at the Peripheral Explorer Board; a digital output value of '1' will switch on a light, a digital '0' will switch it off. Lab5_1 will use register GPAMUX1, GPBMUX1, GPADIR, GPBDIR and the data registers GPADAT, GPBDAT, GPASET, GPACLEAR, GPBSET and GPBCLEAR.

The code of Lab5_1 will continuously increment an integer variable "counter" and display the current value of its 4 least significant bits on LD1 to LD4. For this first hardware based lab we will not use any interrupts. The Watchdog-Timer unit and the core registers to set up the controller speed are also used in this exercise.

Procedure

Create a Project File

1. Using Code Composer Studio, create a new project, called **Lab5.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 "Lab5" and select "Properties". In category "C/C++ Build", "C2000 Linker", "Basic Options" set the C stack size to 0x400.
3. Copy the provided source code file "Lab5_1.c" in the project folder "C:\DSP2833x_V4\Labs\Lab5". This step will automatically include the file in project "Lab5".

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!

3. In the C/C++ perspective, right click at project "Lab5" 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.

4. Repeat the "Link Files to Project" step. From `C:\tidcs\c28\dsp2833x\v131\DSP2833x_common\source` add:

- **DSP2833x_CodeStartBranch.asm**

This file contains a single Long Branch assembly instruction and must be placed into the code entry point section "BEGIN" in code space. The Linker will that do for us, based on the file that is added in the next step.

5. From `C:\tidcs\c28\dsp2833x\v131\DSP2833x_headers\cmd` link to project "Lab5":

- **DSP2833x_Headers_nonBIOS.cmd**

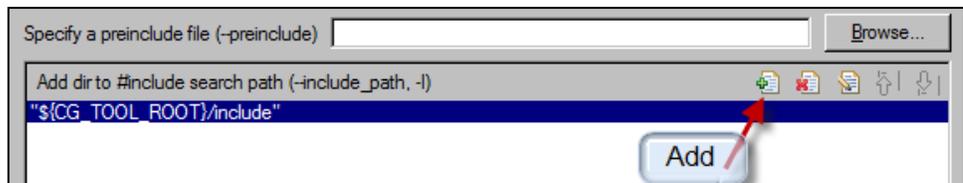
This linker command file will connect all global register variables to their corresponding physical addresses.

Project Build Options

6. We also have to extent the search path of the C-Compiler for include files. Right click at project "Lab5" and select "Properties". Select "C/C++ Build", "C2000 Compiler", "Include Options". In the box "Add dir to #include search path", add the following line:

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

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



Close the Property Window by Clicking <OK>.

Modify the Source Code

After we have prepared our project, it is time to inspect and change the provided C-source code file "Lab5_1.c". To find the correct setup for the registers, use the information from the PowerPoint slides in the presentation of this chapter!

7. Open Lab5_1.c and search for the local function **"InitSystem()"**. You will find several question marks in this code. Your task is to replace all the question marks to complete the code.
 - Set up the Watchdog-Timer (WDCR): disable the Watchdog and clear the WD Flag bit.
 - Set up the SCSR to generate a RESET out of a Watchdog event (WDENINT)
 - Setup the Clock-PLL (PLLCR)-multiply by 10/2. Assuming we use an external 30 MHz oscillator this will set the DSP to 150 MHz internal frequency. Set bit field "DIV" in PLLCR to 10 and field DIVSEL in register PLLSTS to 2!
 - Initialize the High speed Clock Pre-scaler (HISPCP) to "divide by 2", the Low speed Clock Pre-scaler (LOSPCP) to "divide by 4".
 - Enable the GPIO-Clock bit "GPIOINENCLK" in register PCLKCR3. Disable all other peripheral clock units in register: PCLKCR0, PCLKCR1 and PCLKCR3.
8. Search for the local function **"Gpio_select()"** and modify the code in it to:
 - Set up all multiplex register to digital I/O.
 - Set up GPADIR: lines GPIO9 and GPIO11 to output and all other lines to input.
 - Set up GPBDIR: lines GPIO34 and GPIO49 to output and all other lines to input.
 - Set up GPCDIR: all lines to digital input.

Setup the control loop

9. In "Lab5_1.c" look for the endless "while(1)" loop. After the increment of the variable "counter" add some instructions to analyze the current value in "counter":
 - If bit 0 of counter is 1, set GPIO9 to 1, otherwise clear GPIO9 to 0
 - If bit 1 of counter is 1, set GPIO11 to 1, otherwise clear GPIO11 to 0
 - If bit 2 of counter is 1, set GPIO34 to 1, otherwise clear GPIO34 to 0
 - If bit 3 of counter is 1, set GPIO49 to 1, otherwise clear GPIO49 to 0

Note: The GPIO data registers are accessible using a set of 4 registers ('x' stands for A, B or C):

- GpioDataRegs.GPxDAT - access to data register
- GpioDataRegs.GPxSET - set those lines, which are marked with a 1
- GpioDataRegs.GPxCLEAR - clear the lines, which are marked with a 1

- `GpioDataRegs.GPxTOGGLE` - invert the level at lines, which are marked as 1

Example to set pin GPIO5 to 1:

```
GpioDataRegs.GPASET.bit.GPIO5 = 1;
```

Build and Load

10. 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.

11. Load the output file in the debugger session:

Target → Debug Active Project

and switch into the “Debug” perspective.

Test

12. Verify that in the debug perspective the window of the source code “Lab5_1.c” is highlighted and that the blue arrow for the current Program Counter position is placed under the line “void main(void)”.

13. Perform a real time run.

Target → Run

14. Verify that the LEDs behave as expected. In this case you have successfully finished the first part of Lab5_1. Halt the Device (Target → Halt).

Enable Watchdog Timer

15. Now let us improve our Lab5_1 towards a more realistic scenario. Although it was very easy to disable the watchdog for the first part of this exercise, it is not a good practice for a ‘real’ hardware project. The watchdog timer is a security hardware unit; it is an internal part of the F2833x and it should be used in all projects. So let us modify our code:

16. Switch back to the “C/C++” perspective. In file “Lab5_1.c” search the function “InitSystem()” and modify the WDCR - register initialization

- Now do **NOT disable** the watchdog.

17. What will be the result?

- Answer: If the watchdog is enabled, our program will stop operations after a few milliseconds somewhere in our while(1) loop. Depending on the preselected boot-

mode, the watchdog will force the controller into the hardware start sequence, usually into the FLASH entry point. Since our program has been loaded in RAM rather than in FLASH, it will not start again. As a result, our LED program will not run any more!

- Note: The BOOT - Mode sequence of F2833x is selected with 4 GPIOs (GPIO87, 86, 85 and 84), which are sampled during startup. In case of the F28335ControlCard all 4 pins are resistor pulled up to 3.3V, thus the "Jump to FLASH entry point" option is selected by default. At the Peripheral Explorer Board pin GPIO84 can be forced to GND by closing jumper J3 ("Boot-2") at the XDS100 module ("M1") of the Peripheral Explorer Board; this will select the option "SCI-A boot loader". All remaining boot start options are not available for the combination F28335ControlCard + Peripheral Explorer Board.

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

Project → Rebuild Active Project

19. Load the output file in the debugger session:

Target → Debug Active Project

and switch back into the "Debug" perspective.

20. Perform a real time run.

Target → Run

Our LED code should not work any more! This is a sign that the F2833x has been RESET by a watchdog overflow.

Service the Watchdog Timer

21. To enable the watchdog timer was only half of the task to use it properly. Now we have to deal with it in our code. This means that if our control loop runs as expected, the watchdog, although it is enabled, should never trigger a RESET. How can we achieve this? Answer: We have to execute the watchdog reset key sequence somewhere in our control loop. The key sequence consists of two write instructions into the WDKEY-register, a 0x55 followed by a 0xAA.

- Switch back to "C/C++" perspective and inspect file "Lab5_1.c". Look for function "delay_loop()" and uncomment the four lines:

```
EALLOW;  
SysCtrlRegs.WDKEY = 0x55;  
SysCtrlRegs.WDKEY = 0xAA;  
EDIS;
```

Note: The C-Macro "EALLOW" will open the access to certain CPU core registers, including the Watchdog-Registers. The Macro "EDIS" will disable this access.

22. Click the “Rebuild All” button or perform:

Project → Rebuild Active Project

23. Load the output file in the debugger session:

Target → Debug Active Project

and switch back into the “Debug” perspective.

24. Perform a real time run.

Target → Run

25. Now our LED control code should run again as expected. The watchdog is still active but due to our key sequence it will not trigger a RESET unless the F2833x code crashes. Hopefully this will never happen!

END of Lab 5_1

Lab 5_2: Digital Output (modified)

Let's modify the code of Lab5_1. Instead of showing the four least significant bits of variable "counter" as in Lab5_1, let us now produce a "running" LED from left to right and vice versa (known as a "Knight Rider"):



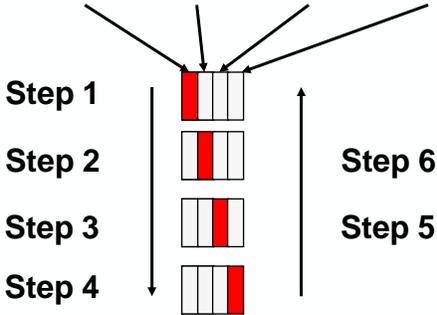
Lab Exercise 5_2



Modify the C -source – code to:

- **switch 4 LEDs at GPIO9, GPIO11, GPIO34 and GPIO49 sequentially on and off**
- **use a software time delay from Lab5_1**

GPIO9 GPIO11 GPIO34 GPIO49



5 - 32

Procedure

Modify Code and Project File

1. Open the source code "Lab5_1.c" from project Lab5.pjt in C:\DSP2833x_V4\Labs\Lab5 and save it as "**Lab5_2.c**".
2. Exclude file "**Lab5_1.c**" from build. Right click at Lab5_1.c in the project window and enable "**Exclude File(s) from Build**". Add the new source code file to your project:
3. Modify the code inside the "Lab5_2.c" according to the new objective. Variable "counter" is no longer needed, so remove it.
4. Rebuild and test as you have done in Lab5_1.

END of Lab 5_2

Lab 5_3: Digital Input

Objective

Now let us add some digital input function to our code. On the Peripheral Explorer Board, the digital lines GPIO12 to GPIO15 are inputs from a 4-bit hexadecimal encoder device (SW2). This device generates a 4-bit number between binary “0000” and “1111”, depending on its position.

The objective of Lab5_3 is to read the status of this hexadecimal encoder and display it at LEDs LD1 (GPIO9), LD2 (GPIO11), LD3 (GPIO34) and LD4 (GPIO49) of the Peripheral Explorer Board.



Lab 5_3: Digital Input (GPIO 15...12)



Objective:

- a 4 bit hex encoder connected to GPIO15...GPIO12
- 4 LED's connected to GPIO9, GPIO11, GPIO34 and GPIO49
- read the status of encoder and display it at the LEDs

Project - Files :

1. C - source file: “Lab5_3.c”
2. Register Definition File:
“DSP2833x_GlobalVariableDefs.c”
3. Linker Command File:
“28335_RAM_Ink.cmd”
4. Runtime Library: “rts2800_fpu32.lib”

5 - 33

Procedure

Modify Code and Project File

1. Open the source code “Lab5_1.c” from project “Lab5” in C:\DSP2833x_V4\Labs\Lab5 and save it as “**Lab5_3.c**”.
2. Exclude file “**Lab5_2.c**” from build. Right click at Lab5_2.c in the project window and select “**Exclude File(s) from Build**”.
3. Modify Lab5_3.c. Remove variable "counter". Keep the function calls to “InitSystem()” and “Gpio_select()”. Inside the endless while(1)-loop, modify the control loop as needed. Just copy the current value from input GPIO12 (encoder bit 0) to output GPIO9 (LED1) and so on.

4. What about the watchdog? Recall that we serviced the watchdog inside “delay_function()” - it would be unwise to remove this function call from our control loop!

Build, Load and Test

5. Build, Load and Test as you have done in previous exercises.

When the code is running, turn the hex-encoder switch at the Peripheral Explorer Board. Each clockwise turn should increment the binary pattern at the 4 LEDs, an anti clockwise turn should decrement the pattern.

END of Lab 5_3

Lab 5_4: Digital In- and Output

Objective

Now let us combine Lab5_1 and Lab5_3! That means your task is to control the speed of your “LED”- counter code (Lab5_1) by the current status of the 4-bit hex encoder. It inputs a value between 0 and 15. For example, we can use this number to change the input parameter for function “delay_loop()” to generate a time interval between 100 milliseconds (hex-encoder = 0) and 1.6 seconds (hex-encoder = 15).



Lab 5_4: Digital In- and Output



- Mix between Lab5_1 and LAB5_3:
 - change the loop – speed of Lab5_1 depending of the status of the hex – encoder.
 - If hex – encoder reads “0000”, set the time period for the LED update to approximately 100 ms.
 - If hex - encoder reads “1111”, set the time period for the LED update to approximately 1.6 seconds.
 - Adjust the period for all other encoder values accordingly.

5 - 33

Modify Code and Project File

1. Open the source code “Lab5_1.c” from project Lab5.pjt in C:\DSP2833x_V4\Labs\Lab5 and save it as “**Lab5_4.c**”.
2. Exclude file “**Lab5_3.c**” from build. Right click at Lab5_3.c in the project window and select “**Exclude File(s) from Build**”.

Modify Lab5_4.C

4. In “main()”, modify the input parameter of the function “delay_loop()”. This parameter defines the number of iterations of the for-loop. All you have to do is to change the current parameter using the GPIO-inputs GPIO15...GPIO12.
5. The best position to update the parameter for the delay loop time is inside the endless loop of “main()”, between two steps of the LED-sequence. Recall, that the 4-bit encoder will give you a number between 0 and 15. The task is to

generate a delay period between 100 milliseconds and 1.6 seconds. You need to do a little bit of maths here. Assuming your DSP runs at 100 MHz, one loop of the “for()” loop -instruction in function “delay_loop()” takes approximately 173 nanoseconds, so you need to scale the value accordingly.

Build, Load and Test

6. Build, Load and Test as you have done in previous exercises.

END of Lab 5_4

Lab 5_5: Digital In- and Output Start / Stop

Objective

As a final exercise in this chapter, let us add some start/stop functionality to our project. The Peripheral Explorer Board is equipped with two push-buttons PB1 and PB2. If pushed, the corresponding input line reads '0'; if not, it reads as '1'. Button PB1 is wired to GPIO17 and PB2 to GPIO48.

The Task is:

- (1) to start the LED counting sequence from Lab5_4, if PB1 has been pushed.
- (2) to suspend the LED counting sequence, if PB2 has been pushed.
- (3) to resume the LED counting, if PB1 has been pushed again



Lab 5_5: Start - /Stop Control



- **Add a start/stop function to Lab5_4:**

- **Peripheral Explorer Board Pushbuttons:**
 - **PB1 (GPIO17) to start/restart control code**
 - **PB2 (GPIO48) to stop/suspend control code**

- **If PB1 is pushed, LED counting should start / resume**
- **If PB2 is pushed, LED counting should stop.**

5 - 35

Modify Code and Project File

1. Open the source code "Lab5_4.c" from project Lab5.pjt in C:\DSP2833x_V4\Labs\Lab5 and save it as "**Lab5_5.c**".
2. Exclude file "**Lab5_4.c**" from build. Right click at Lab5_4.c in the project window and select "**Exclude File(s) from Build**".

Modify Lab5_5.c

4. Inspect function "Gpio_select()" and make sure that GPIO17 and GPIO48 are initialized as input lines.

5. At the beginning of Lab5_5.c add two definitions:

```
#define START      GpioDataRegs.GPADAT.bit.GPIO17
```

```
#define STOP      GpioDataRegs.GPBDAT.bit.GPIO48
```

Now we can use the symbols “START” and “STOP” instead of the long bit variable names.

6. At the beginning of function “delay_loop()”, add a definition for a static variable “run” and initialize it with 0:

```
static unsigned int run = 0;
```

This variable will later be used as a control switch. If run = 0, the control code loop execution is stopped; If run = 1, the control code loop is enabled.

7. Inside the for()-loop of function “delay_loop()”, add a code sequence to postpone the loop-execution as long as PB1 has not been pushed. One option is to use a do-while construction:

```
do
{
    EALLOW;
    SysCtrlRegs.WDKEY = 0x55;
    SysCtrlRegs.WDKEY = 0xAA;           // service watchdog
    EDIS;
    if (START == 0 && STOP == 1) run = 1; // run control code if PB1=0
} while (!run);
```

Note: You will have to adjust the calculation of the input parameter for the function “delay_loop()”!

8. After leaving this do-while loop, we need to check, if PB2 has been pushed. If so, all we have to do is to set variable run = 0.

```
if(STOP == 0)          run = 0;      // suspend
```

With the next repetition of the for() -loop the processor will re-enter the do-while construction and wait for a second START command.

Procedure step 7 and 8 are only one option to solve the task. You might find other solutions even better suited.

Build, Load and Test

9. Build, Load and Test as you have done in previous exercises.

END of Lab 5_5

Blank Page