

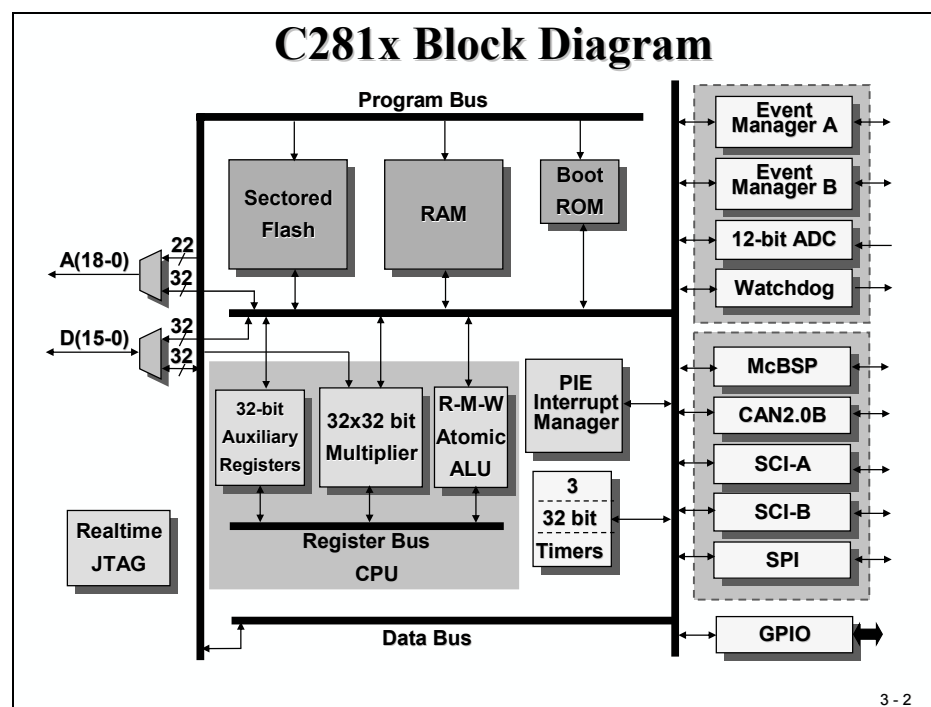
Introduction

This module introduces the integrated peripherals of the C28x DSP. We have not only a 32-bit DSP 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 C28x 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 (LED’s).

Data Memory Mapped Peripherals

All the peripheral units of the C28x 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:

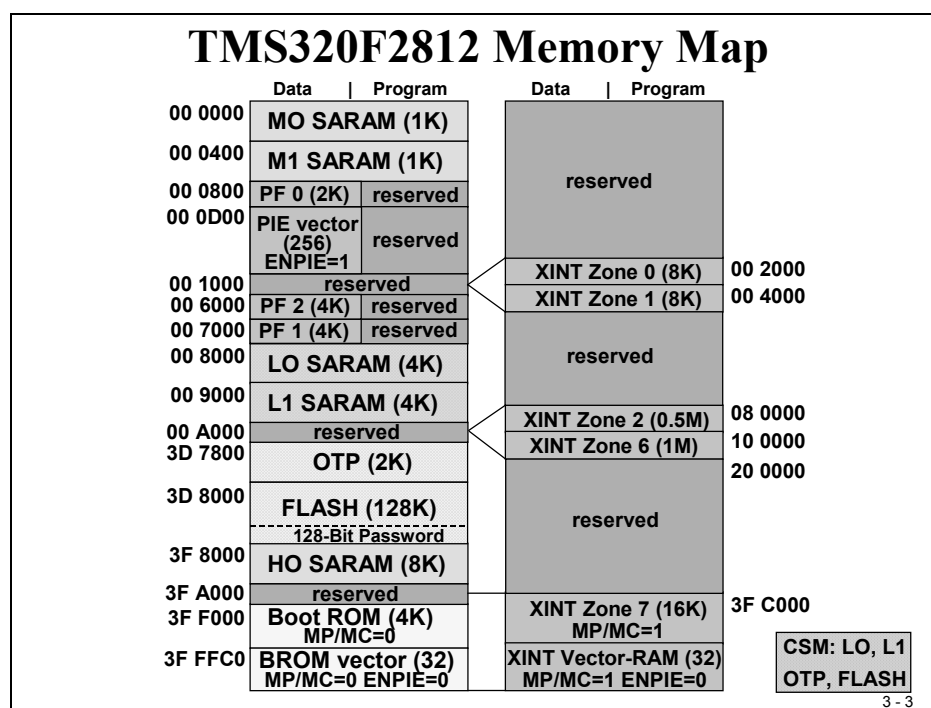


Module Topics

Digital I/O	3-1
<i>Introduction</i>	<i>3-1</i>
<i>Data Memory Mapped Peripherals</i>	<i>3-1</i>
<i>Module Topics.....</i>	<i>3-2</i>
<i>The Peripheral Frames</i>	<i>3-3</i>
<i>Digital I/O Unit.....</i>	<i>3-4</i>
<i>Digital I/O Registers</i>	<i>3-6</i>
<i>C28x Clock Module.....</i>	<i>3-7</i>
<i>Watchdog Timer.....</i>	<i>3-9</i>
<i>System Control and Status Register</i>	<i>3-12</i>
<i>Low Power Mode</i>	<i>3-12</i>
<i>Lab 2: Digital Output – 8 LED's</i>	<i>3-15</i>
<i>Lab 2A: Digital Output – 8 LED's (modified)</i>	<i>3-22</i>
<i>Lab 3: Digital Input</i>	<i>3-23</i>
<i>Lab 3A: Digital Input + Output</i>	<i>3-26</i>
<i>Lab 3B: Start / Stop Option.....</i>	<i>3-29</i>

The Peripheral Frames

All peripheral registers are grouped together into what are known as “Peripheral Frames” – PF0, PF1 and PF2. These frames are data memory mapped only. Peripheral Frame PF0 includes register sets to control the internal speed of the FLASH memory, as well as the access timing to the internal SARAM. SARAM stands for “Single Access RAM”, that means we can make one access to this type of memory per clock cycle. Flash is the 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, whereas Peripheral Frame PF2 is reserved for the CAN register block. CAN – “Controller Area Network” is a well-established network widely used inside cars to build a network between electronic control units (ECU).



Some of the memory areas are password protected by the “Code Security Module” (check patterned areas at 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’s start with the discussion of the Digital I/O unit.

Digital I/O Unit

All digital I/O's are grouped together into "Ports", called GPIO-A, B, D, E, F and G. Here GPIO means "general purpose input output". The C28x is equipped with so many 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 2 (sometimes 3) different functions and it is up to the programmer to decide which function is selected. The next slide shows the options available:

C28x GPIO Pin Assignment		
GPIO A	GPIO B	GPIO D
GPIOA0 / PWM1	GPIOB0 / PWM7	GPIOD0 / T1CTrip_PDPINTA
GPIOA1 / PWM2	GPIOB1 / PWM8	GPIOD1 / T2CTrip7_EVASOC
GPIOA2 / PWM3	GPIOB2 / PWM9	GPIOD5 / T3CTrip_PDPINTB
GPIOA3 / PWM4	GPIOB3 / PWM10	GPIOD6 / T4CTrip7_EVBSOC
GPIOA4 / PWM5	GPIOB4 / PWM11	
GPIOA5 / PWM6	GPIOB5 / PWM12	GPIO E
GPIOA6 / T1PWM_T1CMP	GPIOB6 / T3PWM_T3CMP	GPIOE0 / XINT1_XBIO
GPIOA7 / T2PWM_T2CMP	GPIOB7 / T4PWM_T4CMP	GPIOE1 / XINT2_ADCSOC
GPIOA8 / CAP1_QEP1	GPIOB8 / CAP4_QEP3	GPIOE2 / XNMI_XINT13
GPIOA9 / CAP2_QEP2	GPIOB9 / CAP5_QEP4	
GPIOA10 / CAP3_QEP1	GPIOB10 / CAP6_QEP2	
GPIOA11 / TDIRA	GPIOB11 / TDIRB	
GPIOA12 / TCLKINA	GPIOB12 / TCLKINB	
GPIOA13 / C1TRIP	GPIOB13 / C4TRIP	
GPIOA14 / C2TRIP	GPIOB14 / C5TRIP	
GPIOA15 / C3TRIP	GPIOB15 / C6TRIP	
GPIO F	GPIO G	
GPIOF0 / SPISIMOA	GPIOG4 / SCITXDB	
GPIOF1 / SPISOMIA	GPIOG5 / SCIRXDB	
GPIOF2 / SPICLKA		
GPIOF3 / SPISTEA		
GPIOF4 / SCITXDA		
GPIOF5 / SCIRXDA		
GPIOF6 / CANTXA		
GPIOF7 / CANRXA		
GPIOF8 / MCLKXA		
GPIOF9 / MCLKRA		
GPIOF10 / MFSXA		
GPIOF11 / MFSRA		
GPIOF12 / MDXA		
GPIOF13 / MDRA		
GPIOF14 / XF		

Note: GPIO are pin functions at reset

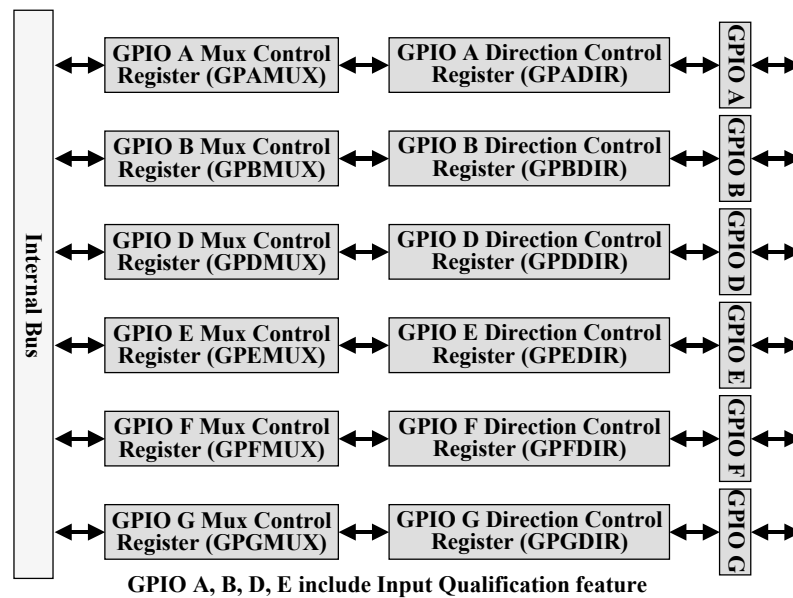
GPIO A, B, D, E include Input Qualification feature

3 - 5

The next slide explains the initialization procedure. All six GPIO-Ports are controlled by their own multiplex register, called GPxMUX (where x stands for A to F). Clearing a bit position to zero means selecting its digital I/O function, setting a bit to 1 means selecting the special function (TI calls this the "primary" function).

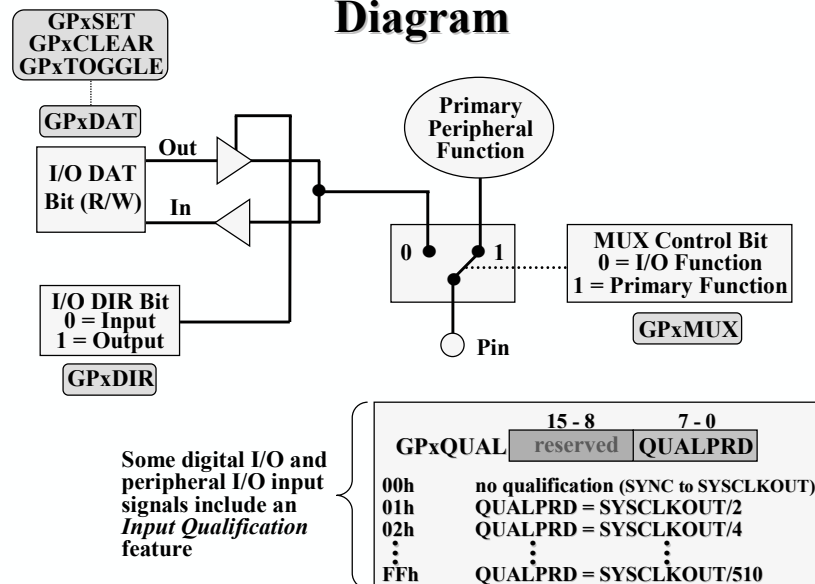
When digital I/O function is selected, then register group GPxDIR defines the direction of I/O. Clearing a bit position to zero configures the line as an input, setting the bit position to 1 configures the line as an output. Some of the input ports are equipped with an "Input Qualification Feature". With this option we can define a time length, which is used to exclude spikes or pulses of a shorter duration from being acknowledged as valid input signals.

C28x GPIO Register Structure



3 - 4

C28x GPIO Functional Block Diagram



3 - 6

Digital I/O Registers

The next two slides summarize the digital I/O control registers:

C28x GPIO MUX/DIR Registers

Address	Register	Name
70C0h	GPAMUX	GPIO A Mux Control Register
70C1h	GPADIR	GPIO A Direction Control Register
70C2h	GPAQUAL	GPIO A Input Qualification Control Register
70C4h	GPBMUX	GPIO B Mux Control Register
70C5h	GPBDIR	GPIO B Direction Control Register
70C6h	GPBQUAL	GPIO B Input Qualification Control Register
70CCh	GPDMUX	GPIO D Mux Control Register
70CDh	GPDDIR	GPIO D Direction Control Register
70CEh	GPDQUAL	GPIO D Input Qualification Control Register
70D0h	GPEMUX	GPIO E Mux Control Register
70D1h	GPEDIR	GPIO E Direction Control Register
70D2h	GPEQUAL	GPIO E Input Qualification Control Register
70D4h	GPFMUX	GPIO F Mux Control Register
70D5h	GPFDIR	GPIO F Direction Control Register
70D8h	GPGMUX	GPIO G Mux Control Register
70D9h	GPGDIR	GPIO G Direction Control Register

3 - 7

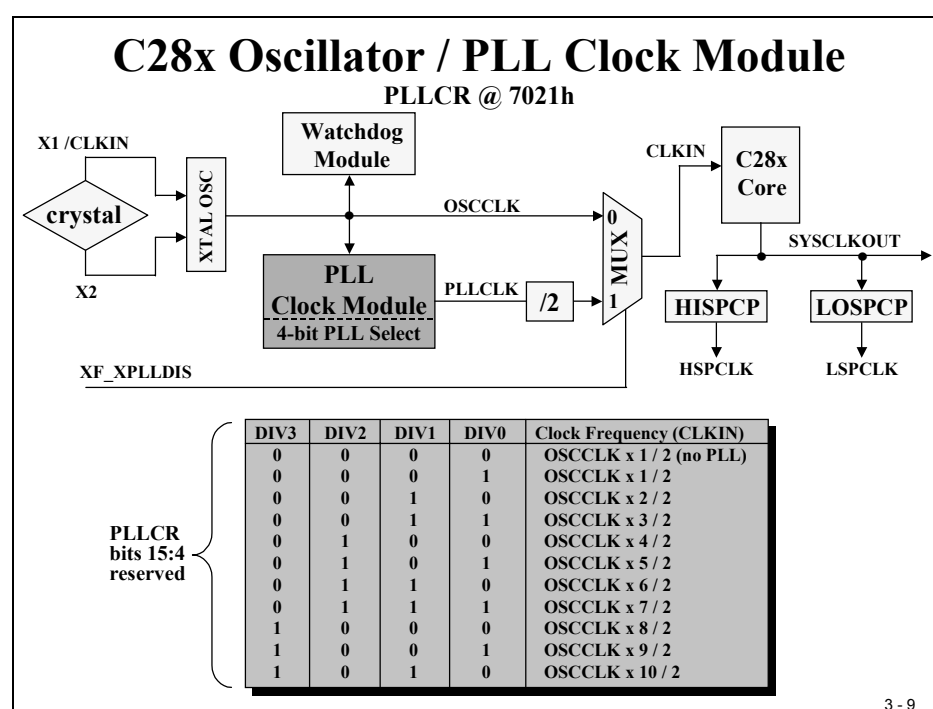
C28x GPIO Data Registers

Address	Register	Name
70E0h	GPADAT	GPIO A Data Register
70E1h	GPASET	GPIO A Set Register
70E2h	GPACLEAR	GPIO A Clear Register
70E3h	GPATOGGLE	GPIO A Toggle Register
70E4h	GPBDAT	GPIO B Data Register
70E5h	GPBSET	GPIO B Set Register
70E6h	GPBCLEAR	GPIO B Clear Register
70E7h	GPBTOGGLE	GPIO B Toggle Register
70ECh	GPDDAT	GPIO D Data Register
70EDh	GPDSET	GPIO D Set Register
70EEh	GPDCLEAR	GPIO D Clear Register
70EFh	GPDTOGGLE	GPIO D Toggle Register
70F0h	GPEDAT	GPIO E Data Register
70F1h	GPESET	GPIO E Set Register
70F2h	GPECLEAR	GPIO E Clear Register
70F3h	GPETOGGLE	GPIO E Toggle Register
70F4h	GPFDAT	GPIO F Data Register
70F5h	GPFSET	GPIO F Set Register
70F6h	GPFCLEAR	GPIO F Clear Register
70F7h	GPFTOGGLE	GPIO F Toggle Register
70F8h	GPGDAT	GPIO G Data Register
70F9h	GPGSET	GPIO G Set Register
70FAh	GPGCLEAR	GPIO G Clear Register
70FBh	GPGTOGGLE	GPIO G Toggle Register

3 - 8

C28x Clock Module

Before we can start using the digital I/Os, we need to setup the C28x Clock Module. Like all modern processors, the C28x is driven outside by a slower external oscillator to reduce electromagnetic disturbances. An internal PLL circuit generates the internal speed. The eZdsp in our Labs is running at 30MHz externally. To achieve the internal frequency of 150 MHz we have to use the multiply by 10 factor with divide by 2. This can be done by programming the PLL control register (PLLCR).



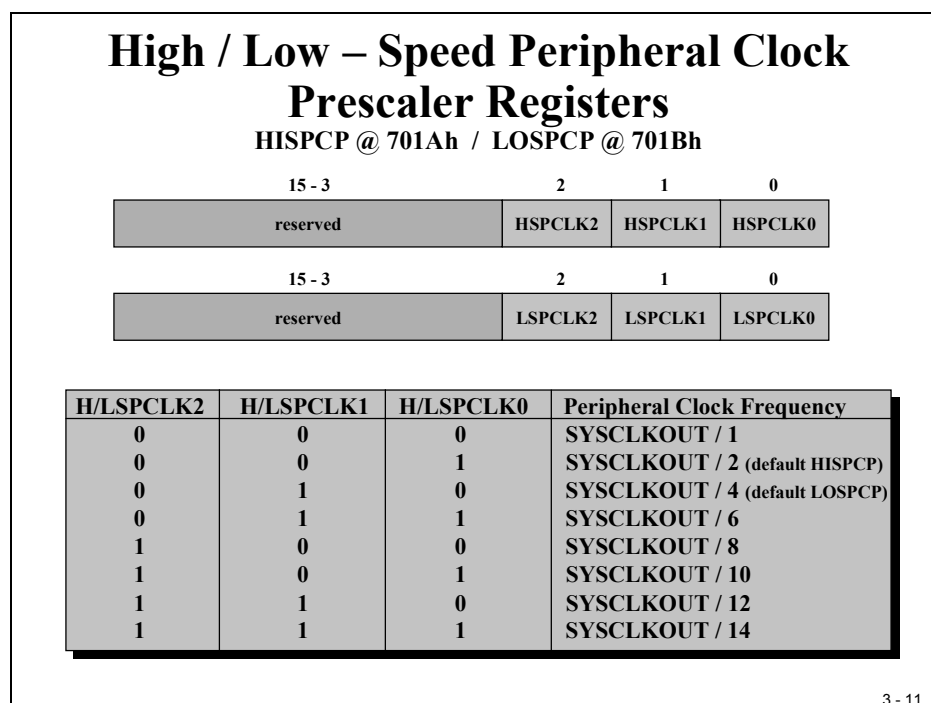
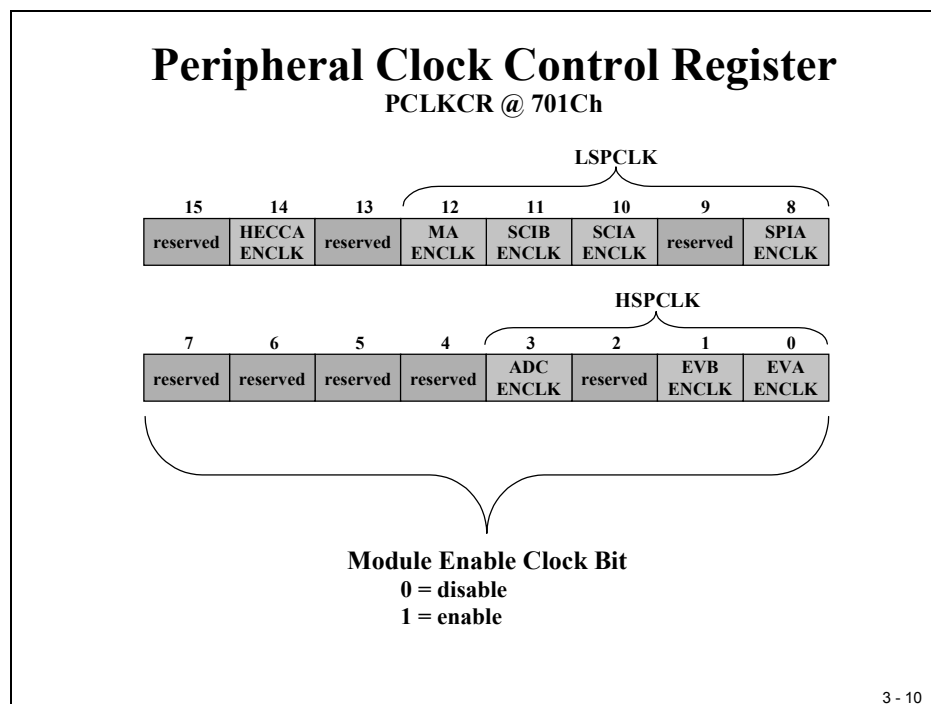
High-speed Clock Pre-scaler (HSPCP) and Low speed Clock Pre-scaler (LOSPCP) are used as additional clock dividers. The outputs of the two pre-scalers are used as the clock source for the peripheral units. We can set up the two pre-scalers individually to our needs.

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 and for clocking the CAN – unit.

Also, that (2) the Watchdog Unit is clocked directly by the external oscillator.

Finally, that (3) the maximum frequency for the external oscillator is 35MHz.

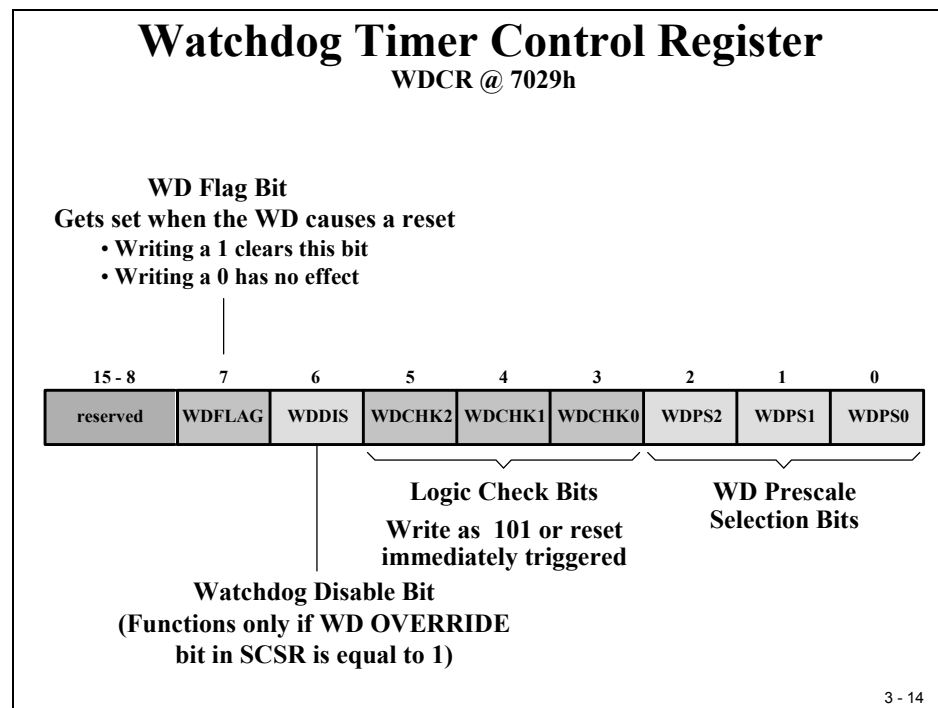
To use a peripheral unit, we have to enable its clock distribution by individual bit fields of register PCLKCR. Digital I/O does not have a clock enable feature.



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 (WDFLAG) 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 situation by reading the Watchdog counter register periodically. In case of a lost external clock this register will not increment any longer. The C28x 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? By writing a “valid key” sequence into register WDKEY:

Resetting the Watchdog

WDKEY @ 7025h

15-8	7	6	5	4	3	2	1	0
reserved	D7	D6	D5	D4	D3	D2	D1	D0

- ◆ **Allowable write values:**
 - 55h - counter enabled for reset on next AAh write
 - AAh - counter set to zero if reset enabled
- ◆ **Writing any other value immediately triggers a CPU reset**
- ◆ **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 55h WDKEY in the main code, and the AAh WDKEY in an ISR; this catches main code crashes and also ISR crashes

3 - 15

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	CPU reset triggered due to improper write value

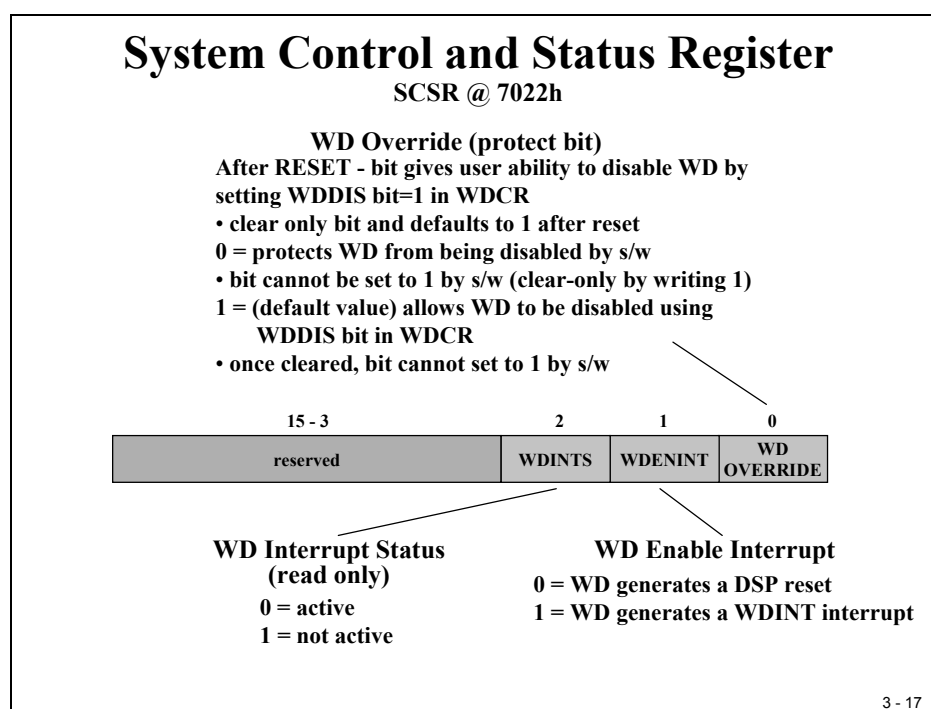
3 - 16

System Control and Status Register

Register SCSR controls whether the Watchdog causes a RESET (WDENINT = 0) or an Interrupt Service Request (WDENINT = 1).

The WDOVERRIDE bit is a “clear only” bit, that means, once we have closed this switch by writing a 1 into the bit, we can’t reopen this switch again (see block diagram of the Watchdog). At this point the WD-disable bit is ineffectual, 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 C28x is able to switch into 3 different low-power operating modes. We will not use this feature for 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 Register LPMCR0 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

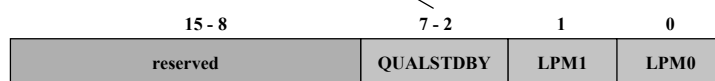
3 - 18

Low Power Mode Control Register 0

LPMCR0 @ 701Eh

Qualify before waking from STANDBY mode

000000 = 2 OSCCLKs
 000001 = 3 OSCCLKs
 ⋮
 111111 = 65 OSCCLKs



Low Power Mode Selection

00 = Idle
 01 = Standby
 1x = Halt

Low Power Mode Entering

1. Set LPM bits
2. Enable desired exit interrupt(s)
3. Execute IDLE instruction
4. The Power down sequence of the hardware depends on LP mode

3 - 19

Low Power Mode Control Register 1

LPMCR1 @ 701Fh

15	14	13	12	11	10	9	8
CANRXA	SCIRXB	SCIRXA	C6TRIP	C5TRIP	C4TRIP	C3TRIP	C2TRIP

7	6	5	4	3	2	1	0
C1TRIP	T4CTRIP	T3CTRIP	T2CTRIP	T1CTRIP	WDINT	XNMI	XINT1

Wake device from
STANDBY mode
0 = disable
1 = enable

3 - 20

Low Power Mode Exit

<div>Exit Interrupt</div> <div>Low Power Mode</div>	RESET	External or Wake up Interrupts	Enabled Peripheral Interrupts
IDLE	yes	yes	yes
STANDBY	yes	yes	no
HALT	yes	no	no

Note: External or Wake up include XINT1, PDPINT, TxCTRIP, CxTRIP NMI, CAN, SPI, SCI, WD

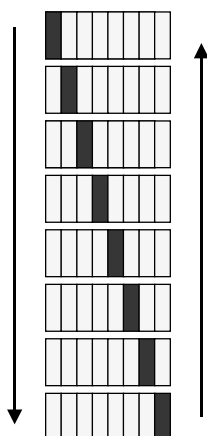
3 - 21

Lab 2: Digital Output – 8 LED's

Lab 2: Digital Output on Port B0...B7

Aim :

- Use the 8 LED's connected to GPIO- outputs B0-B7 to show a 'running light' moving from left to right and reverse



Project - Files :

1. C - source file: "Lab2.c"
2. Register Definition File: "DSP281x_GlobalVariableDefs.c"
3. Linker Command File : F2812_EzDSP_RAM_Ink.cmd
4. Runtime Library "rts2800_ml.lib"

- Use a software delay loop to generate the pause interval

3 - 22

Lab 2: Digital Output on Port B0...B7

Registers to be used in LAB 2 :

• Initialise DSP:

- | | | |
|----------------------------------|---|--------|
| • Watchdog - Timer - Control | : | WDCR |
| • PLL Clock Register | : | PLLCR |
| • High Speed Clock Prescaler | : | HISPCP |
| • Low Speed Clock Prescaler | : | LOSPCP |
| • Peripheral Clock Control Reg. | : | PCLKCR |
| • System Control and Status Reg. | : | SCSR |

• Access to LED's (B0...B7):

- | | | |
|------------------------------|---|---------|
| • GPB Multiplex Register | : | GPBMUX |
| • GPB Direction Register | : | GPBDIR |
| • GPB Qualification Register | : | GPBQUAL |
| • GPB Data Register | : | GPBDAT |

3 - 23

Register Definition File 'DSP281x_GlobalVariableDefs.c'

- This File defines global variables for all memory mapped peripherals.
- The file uses predefined structures (see ..\include) and defines instances , e.g. "GpioDataRegs" :

```
#pragma DATA_SECTION(GpioDataRegs,"GpioDataRegsFile");  
volatile struct GPIO_DATA_REGS GpioDataRegs;
```

or "GpioMuxRegsFile" :

```
#pragma DATA_SECTION(GpioMuxRegs,"GpioMuxRegsFile");  
volatile struct GPIO_MUX_REGS GpioMuxRegs;
```
- The structures consist of all the registers, that are part of that group , e.g. : **GpioDataRegs.GPBDAT**
- For each register exists a union to make a 16bit-access ("all") or a bit-access ("bit") , e.g. :

```
GpioDataRegs.GPBDAT.bit.GPOIB4 = ....  
GpioDataRegs.GPBDAT.all = ....
```

3 - 24

Register Definition File 'DSP281x_GlobalVariableDefs.c'

- The name of the DATA_SECTION ("GpioDataRegsFile") is used by the linker command file to connect the section's variable ("GpioDataRegs") to a physical memory address.
- The master header -file '**DSP281x_Device.h**' includes all the predefined structures for all peripherals of this DSP.
- All that needs to be done is :
 - (1) make 'DSP281x_GlobalVariableDefs.c' part of your project
 - (2) include 'DSP281x_Device.h' in your main C file.

3 - 25

Objective

The objective of this lab is to practice using basic digital I/O – operations. GPIO-Port B7 to B0 are connected to 8 LEDs, a digital ‘1’ will switch on a light, a digital ‘0’ will switch it off. GPIO-Port B15 to B8 are connected to 8 input switches; an open switch will be red as digital ‘1’, a closed one as digital ‘0’. Lab2 uses register GPBMUX, GPBDIR and GPBDAT.

First in Lab2 we will generate a running light (“Knight Rider”). This lab will be expanded into Labs 2A, 3 and 3A. For the time being we will not use any Interrupts. The Watchdog-Timer as well as the core registers to set up the DSP-speed are involved in this exercise.

Procedure

Open Files, Create Project File

1. Using Code Composer Studio, create a new project, called **Lab2.pjt** in E:\C281x\Labs (or another working directory used during your class, ask your teacher for specific location!).
2. Add the provided source code file to your new project:
 - **Lab2.c**
3. 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_Ink.cmd**

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

- **F2812_Headers_nonBIOS.cmd**

From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

Project Build Options

4. 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

5. 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>.

Modify the Source Code

6. Open Lab2.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 (for now) 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 5, assuming we use an external 30 MHz oscillator this will set the DSP to 150 MHz internal frequency.
 - Initialize the High speed Clock Pre-scaler (HISPCP) to “divide by 2“, the Low speed Clock Pre-scaler (LOSPCP) to “divide by 4”.
 - Disable all peripheral units (PCLKCR) for now.
7. 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 Ports A, D, E, F and G as inputs.
 - Set up Port B15 to B8 as input and B7 to B0 as output.
 - Set all input qualifiers to zero.

Verify the control loop

8. Inside “Lab2.c” look for the endless “while(1)” loop and verify the operation of this test program. The provided solution is based on a look-up table “LED[8]”. All the code does is to take the next value out of this table and move it to the LED's. In between the steps, the function “delay_loop()” is called to generate a pause interval.

Build and Load

9. 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.

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

File → Load Program

and choose the desired output file.

Test

11. Reset the DSP by clicking on:

Debug → Reset CPU
Debug → Restart

followed by

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

Debug → Go main.

Verify that in the working area the window of the source code “Lab2.c” is highlighted and that the yellow arrow for the current Program Counter is placed under the line “void main(void)”.

13. Perform a real time run.

Debug → Run

Verify that the LED's behave as expected. If yes, then you have successfully finished the first part of Lab2.

Enable Watchdog Timer

14. Now let's improve our Lab2 a little bit. Although it is quite simple 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 28x and it should be used in all projects. Let's change our code:
15. Look again for the function "InitSystem()" and modify the WDCR – line to NOT disable the watchdog.
16. What will be the result? Answer: If the watchdog is enabled after RESET, our program will stop operations somewhere in our while(1) loop and will start all over again and again. How can we verify this? Answer: by setting a breakpoint to the first line of "main" our program should hit this breakpoint periodically. AND: Our "Knight-Rider" program will never reach its full period.
17. 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.

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

File → Load Program and choose the desired output file.

19. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart

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

Debug → Go main.

21. Perform a real time run.

Debug → Run

Now our "Knight-Rider" code should start all over again before the last LED has been switched on. This is a sign that the DSP starts from RESET periodically.

22. To verify the watchdog operation we can use a breakpoint at line "InitSystem()".

To do so, click right mouse and select "Toggle Breakpoint". A red dot will mark this active breakpoint. Under normal circumstances this line would be passed only once before we enter the while(1) loop. Now, with an active watchdog timer, this breakpoint will be hit periodically.

Serve the Watchdog Timer

23. 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. That means, 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. Look for the function “delay_loop()” and uncomment the two lines:

SysCtrlRegs.WDKEY = 0x55;

SysCtrlRegs.WDKEY = 0xAA;

24. 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.

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

File → Load Program and choose the desired output file.

26. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart

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

Debug → Go main.

28. Perform a real time run.

Debug → Run

Now our “Knight Rider”-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 DSP code crashes. Hopefully this will never happen!

Lab 2A: Digital Output – 8 LED's (modified)

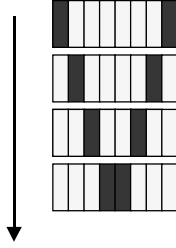
Objective

Let's modify the code of Lab2. Instead of switching on one LED at a time, as we have done in Lab2, let's now switch on 2 LED's at a time, according to the diagram in the following slide:

Lab Exercise 2A

Modify the C -source - code :

- switch 2 LED's on (B7 and B0)
- let the 'light' move one step to the centre of the LED-bar (B6 and B1 switched on)
- continue the move until the 'lights' touch each other
- 'move' the in the opposite direction



B7 and B0 = on
B6 and B1 = on
B5 and B2 = on
B4 and B3 = on

3 - 26

Procedure

Modify Code and Project File

1. Open the source code "Lab2.c" from project Lab2.pjt in E:\C281x\Labs and save it as "**Lab2A.c**".
2. Remove the file "**Lab2.c**" from the project Lab2.pjt. Right click at Lab2.c in the project window and select "**Remove from project**".
3. **Add** the file "**Lab2A.c**" to the project "Lab2.pjt".
4. Modify the code inside the "Lab2A.c" according to the objective of this Lab2A. Take into account the lookup table and the control loop in main.
5. Rebuild and test as you've done in Lab2.

Lab 3: Digital Input

Objective

Now let's add some digital input functions to our code. On the Zwickau Adapter Board, the digital I/O lines GPIO-B15 to B8 are connected to 8 input switches. When a switch is closed it will be red as digital '0', if it is open as '1'.

The objective of Lab3 is to copy the status of the 8 switches to the 8 LED's as the only task of the main loop. Hopefully our DSP will not complain about the simplicity of this task!

Lab 3: Digital Input (GPIO B15..B8)

Aim :

- 8 DIP-Switches connected to GPIO-Port B (B15...B8)
- 8 LED's connected to B7...B0
- read the switches and show their status on the LED's

Project - Files :

1. C - source file: "Lab3.c"
2. Register Definition File:
"DSP281x_GlobalVariableDefs.c"
3. Linker Command File :
F2812_EzDSP_RAM_Ink.cmd
4. Runtime Library "rts2800_ml.lib"

3 - 27

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab3.pjt** in E:\C281x\Labs.
2. Open the file Lab2.c from E:\C281x\Labs\Lab2 and save it as Lab3.c in E:\C281x\Labs\Lab3.
3. Modify Lab3.c. Remove the lookup table "LED[8]". Keep the function calls to "InitSystem()" and "Gpio_select()". Inside the endless while(1)-loop modify the control loop as needed. Do you still need the for-loop? How about the watchdog?

Remember, we served the watchdog inside “delay_function()” – it would be unwise to remove this function call from our control loop!

4. Add the source code file to your project:

- **Lab3.c**

5. 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_Ink.cmd**

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

- **F2812_Headers_nonBIOS.cmd**

From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

Project Build Options

6. 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

7. 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

8. 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.

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

File → Load Program and choose the desired output file.

Test

10. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart

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

Debug → Go main.

12. Start the code

Debug → Run

and test the operation of your code. When you change the status of the switch-line you should see the new value immediately shown at the LED's.

If not, your modification of the code (Step 3 of the procedure) was not correct. In this case try to find out why by using the debug tools that you've learned about in Lab1 (Breakpoint, Step, Watch Variables...).

Lab 3A: Digital Input + Output

Objective

Now let's combine Lab3 and Lab2! That means I'd like you to control the speed of your "Knight Rider" (Lab2) depending on the status of the input switches. Question: What's the minimum / maximum value that can be produced by the 8 input switches? Use the answer to calculate the length of function "delay_loop()" depending on the input from GPIO B15...B8.

Lab 3A

"Knight - Rider" plus frequency control :

- **modify Lab 2 :**
 - **read the input switches (B15-B8)**
 - **modify the frequency of the 'running light' (B7-B0) subject to the status of the input switches, e.g. between 10sec and 0.01 sec per step of the LED-sequence**
- **enable the watchdog timer !**
 - **Verify that , ones your program is in the main loop, the watchdog causes a reset periodically.**

3 - 28

Procedure

Create Project File

1. Create a new project, called **Lab3A.pjt** in E:\C281x\Labs.
2. Open the file Lab2.c from E:\C281x\Labs\Lab2 and save it as Lab3A.c in E:\C281x\Labs\Lab3A.
3. Add the source code file to your project:
 - **Lab3A.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_Ink.cmd**

From `C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd` add:

- **F2812_Headers_nonBIOS.cmd**

From `C:\ti\c2000\cgtoolslib` add:

- **rts2800_ml.lib**

Setup Build Options

5. 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

6. 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>**.

Modify Lab3A.C

7. Modify the run time of function “delay_loop”. The input parameter of this function defines the run time of the software delay loop. All you have to do is to adjust the actual parameter using the GPIO-input’s B15...B8.
8. The best position to update the parameter for the delay loop time is inside the endless loop of main, between two steps of the “Knight Rider” sequence.

Lab 3A (cont.)

Serve the watchdog :

- **do not disable the watchdog timer !**
- **Inside the main-loop execute the watchdog-reset instructions (WDKEY) to prevent the watchdog timer from overflow.**
- **Place the software-delay in a function and experiment with different delay period's.**
What is the period when the watchdog-timer does reset the DSP ?

3 - 29

9. Remember, it is always good practice to work with an enabled watchdog! Eventually for a large parameter for the period of delay_loop() you will have to adjust your watchdog good key sequence instructions to prevent the watchdog from causing a RESET.

Build, Load and Test

10. Build, Load and Test as you've done in previous exercises.

Lab 3B: Start / Stop Option

Objective

A last improvement of our Lab is to add a START/STOP option to it. The Zwickau adapter board has two momentary push buttons connected to GPIO-D1 and GPIO-D6. If a button is pushed, the input line reads '0'; if it is not pushed it reads '1'. The objective is now to use D1 as a start button to start the 'Knight Rider' sequence and D6 to stop it.

Lab 3B

Add start/stop control:

- use Lab 2 to start:
 - GPIO-D1 and D6 are connected to two push-buttons. If they are pushed, the input level reads 0, if released 1.
 - Use D1 to start the LED "Knight-rider" and D6 to halt it. If D1 is pushed again the sequence should continue again.
 - To do so, you also need to add the instructions to initialise GPIO-D

3 - 30

Procedure

Create Project File

1. Create a new project, called **Lab3B.pjt** in E:\C281x\Labs.
2. Open the file Lab3A.c from E:\C281x\Labs\Lab3A and save it as Lab3B.c in E:\C281x\Labs\Lab3B.
3. Add the source code file to your project:
 - **Lab3B.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_Ink.cmd**

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

- **F2812_Headers_nonBIOS.cmd**

From *C:\ti\c2000\cgtoolslib* add:

- **rts2800_ml.lib**

Setup Build Options

5. **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

6. 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>.

Modify Lab3B.C

7. Take into account to modify the endless while(1) loop of main. The for-loop should run after D1 is pushed and freeze when D6 is pushed. With the next D1 the procedure should resume from its frozen status.

Build, Load and Test

8. Build, Load and Test as you've done in previous exercises.