

TMS320F2812

DIGITAL SIGNAL PROCESSOR

IMPLEMENTATION TUTORIAL

Introduction

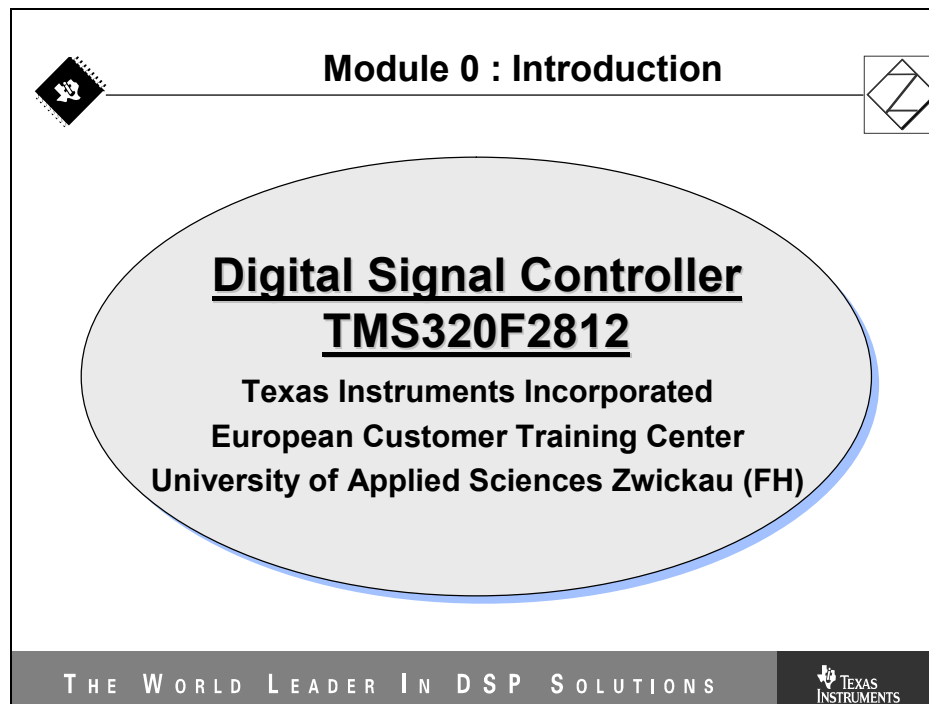
Welcome to the C28x - Tutorial

Welcome to the Texas Instruments TMS320F2812 Tutorial. This material is intended to be used as a student guide for a series of lessons and lab exercises dedicated to the TMS320F2812 Digital Signal Controller. The series of modules will guide you through the various elements of this device, as well as train you in using Texas Instruments development tools and additional resources from the Internet.

The material should be used for undergraduate classes at university. A basic knowledge of microprocessor architecture and programming microprocessors in language C is necessary. The material in Part I (Modules 0 to 9) are to be used in one semester, accompanied by lab exercises in parallel. Each module includes a detailed lab procedure to be used by students during their lab sessions.

The experimental lab sessions are based on the eZdsp TMS320F2812, the Code Composer Studio IDE that is supplied with the eZdsp and some additional hardware (The “Zwickau Adapter Board”). Copies of this add-on board are available from the author. The schematic of the board is also part of this CD-ROM, so that you can build one yourself as well.

Part II (Modules 10 to 15) of the series goes deeper into details of the TMS320F2812. It covers more advanced subjects and can be seen as an optional series of lessons.



Module Topics

Introduction	0-1
<i>Welcome to the C28x - Tutorial</i>	<i>0-1</i>
<i>Module Topics.....</i>	<i>0-2</i>
<i>CD – ROM Structure</i>	<i>0-3</i>
Modules Part I	0-3
Modules Part II	0-3
<i>Template Files for Laboratory Exercises.....</i>	<i>0-4</i>
<i>What is a Digital Signal Controller?</i>	<i>0-6</i>
The Intel 80x86: A typical Microprocessor	0-7
The Desktop – PC: a Micro Computer	0-8
The Microcontroller : a single chip computer.....	0-9
A Digital Signal Processor	0-10
The “Sum of Product” – Equation	0-11
A SOP executed by a DSP.....	0-13
A Digital Signal Controller.....	0-14
<i>DSP Competition</i>	<i>0-15</i>
<i>Texas Instruments DSP – Portfolio.....</i>	<i>0-16</i>
<i>TMS320F28x Roadmap</i>	<i>0-18</i>

CD – ROM Structure

Modules Part I

Chapter 0: Introduction to DSP

Chapter 1: TMS320F2812 Architecture

Chapter 2: Software Development Tools

Chapter 3: Digital Input/Output

Chapter 4: Understanding the F2812 Interrupt System

Chapter 5: Event Manager

Chapter 6: Analogue to Digital Converter

Chapter 7: Communication I: Serial Peripheral Interface

Chapter 8: Communication II: Serial Communication Interface

Chapter 9: Communication III: Controller Area Network (CAN)

Modules Part II

Chapter 10: Flash Programming

Chapter 11: IQ – Math Library

Chapter 12: DSP/BIOS

Chapter 13: Boot – ROM

Chapter 14: FIR – Filter



Chapter 15: Digital Motor Control

Template Files for Laboratory Exercises

All modules are accompanied by laboratory exercises. For some of the modules template files are provided with the CD (“lab template files”), for other modules the students are expected to develop their own project files out of previous laboratory sessions. In these cases the lab description in the textbook chapter explains the procedure. A 2nd group of project files (“solution files”) provides a full solution directory for all laboratory exercises. This group is intended to be used by teachers only. Instead of a single zip-file for the whole CD-ROM we decided to use separate archive files for the individual modules. This gives the teacher the opportunity to select parts of the CD to be used in his classes.

The zip-files should be extracted to a working directory of your choice. However, the textbook assumes that the files are located in: “E:\C281x\labs” for group #1 and “E:\C281x\solutions” for group #2. When extracted, a subfolder named with the exercise number will be added.

The CD-ROM lab template files are archived as follows:

 Laboratory template files 	
Location: E:\C281x\Labs	
Laboratory Exercise	Template archive file
Lab1	“lab_chapter_2.zip”
Lab2	“lab_chapter_3.zip”
Lab5A	“lab_chapter_5.zip”
Lab7A	“lab_chapter_7.zip”
Lab14	“lab_chapter_14.zip”

0 - 3

The laboratory exercises are:

- | | |
|-------------------------------------|--------------------------------------------------------------------------|
| Lab1: „Beginner’s project“ | - essentials of Code Composer Studio |
| Lab2: “Digital Output” | - 8 LED’s, perform a “running light” – sequence |
| Lab3: “Digital Input” | - read 8 input switches and copy status to LED’s |
| Lab3A: “Digital I/O” | - control speed of Lab2 by 8 input switches |
| Lab3B: “Digital I/O” | - add a start and a stop button to Lab3A |
| Lab4: “Core Timer 0 and Interrupts” | - add a hardware timer unit to Lab2 and use an interrupt service routine |
| Lab5: “Pulse Width Modulation” | - Let’s play a tune |
| Lab5A: “Sine Wave PWM” | - generate a sine wave signal by using the Boot-ROM lookup table |

Lab6: “Analogue Digital Converter”	- read two analogue voltages and visualizes the digital values as a “light-beam”
Lab6A: “Analogue Control”	- use one analogue input channel to control the speed of Lab4
Lab7: “SPI – DAC TLV5617A”	- generate a rising and falling saw tooth voltage at the two output channels of the dual DAC
Lab7A: “CCS Graph Tool”	- feedback the two DAC signals into two ADC channels and visualize the signal shape with Code Composer Studio’s graphical tool
Lab7B: “SPI _EEPROM M95080”	- store the status of 8 input switches in address 0x40 of the external EEPROM. Read the EEPROM and display the value on 8 LED’s
Lab8: “SCI-Transmission”	- send a string from DSP to a PC’s COM-port
Lab8A: “SCI-Transmit Interrupt”	- adds SCI-Transmit Interrupt and Core Timer 0 to Lab8
Lab8B: “SCI-FIFO Transmission”	- use SCI-FIFO mode to transmit the string
Lab8C: “SCI Transmit & Receive”	- when string “Texas” is received from a PC the DSP answers by transmitting “Instruments”
Lab9: “CAN – Transmission”	- Transmit the status of 8 input switches with 100KBPS and Identifier 0x1000 0000 (extended mode) periodically.
Lab10: “CAN – Receive”	- Receive identifier 0x1000 0000 with 100KBPS and display the one-byte-message at the 8 LED’S
Lab11: “FLASH Boot Mode”	- modify Lab4 to start out of internal Flash. Program Flash memory using CCS
Lab12: “DSP-BIOS”	- modify Lab2 to use BIOS functions and configuration data base
Lab14: “FIR – Filter”	- Filter a square wave signal with a digital filter.
Lab15: “Digital Motor Control”	- use TI’s library to control a 3phase PMSM – motor.

Laboratory solution files	
Location: E:\C281x\Solutions	
Exercise	Archive file
Lab1	“solution_chapter_2.zip”
Lab2, Lab3, Lab3A, Lab3B	“solution_chapter_3.zip”
Lab4	“solution_chapter_4.zip”
Lab5, Lab5A	“solution_chapter_5.zip”
Lab6, Lab6A	“solution_chapter_6.zip”
Lab7, Lab7A, Lab7B	“solution_chapter_7.zip”
Lab8, Lab8A, Lab8Aopt, Lab8B, Lab8C	“solution_chapter_8.zip”
Lab9, Lab10	“solution_chapter_9.zip”
Lab11	“solution_chapter_10.zip”
Lab12	“solution_chapter_12.zip”
Lab14_1, Lab14_2	“solution_chapter_14.zip”


0 - 4

What is a Digital Signal Controller?


First we have to discuss some keywords that are quite often used when we speak about digital control or computing in general. The TMS320F2812 belongs to a group of devices that are called “Digital Signal Controller (DSC)”. In computing, we use words like “Microprocessor”, “Microcomputer” or “Microcontroller” to specify a given sort of electronic device. When it comes to digital signal processing, the preferred name is “Digital Signal Processors (DSP)”.

To begin with, let's introduce some definitions:

- Microprocessor (μ P)
- Micro Computer
- Microcontroller (μ C)
- Digital Signal Processor (DSP)
- Digital Signal Controller (DSC)



What is a Digital Signal Controller ?



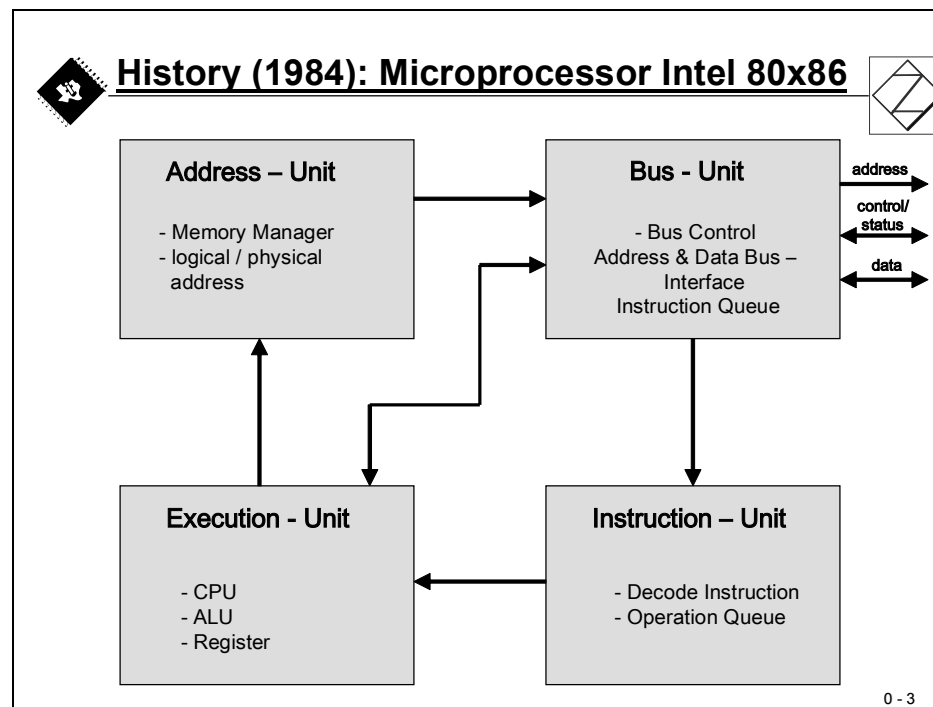
1. Microprocessor (μ P):

- Central Device of a multi chip Micro Computer System
- Two basic architectures:
 - » „Von Neumann“- Architecture
 - » „Harvard“ – Architecture
- „Von Neumann“ - Architecture:
 - » Shared memory space between code and data
 - » Shared memory busses between code and data
 - » Example: Intel's x86 Pentium Processor family
- „Harvard“ – Architecture:
 - » Two independent memory spaces for code and data
 - » Two memory bus systems for code and data
- A μ P to operate needs additional devices

0 - 2

Microprocessors are based on a simple sequential procedural approach: Read next machine code instruction from code memory, decode instruction, read optional operands from data memory, execute instruction and write back result. This series of events runs in an endless manner. To use a μ P one has to add memory and additional external devices to the Microprocessor.

The Intel 80x86: A typical Microprocessor



The Intel 8086 can be considered to be the veteran of all microprocessors. Inside this processor four units take care of the sequence of states. The bus-unit is responsible for addressing the external memory resources using a group of unidirectional digital address signals, bi-directional data lines and control and status signals. Its purpose is to fill a first pipeline, called the “instruction queue” with the next machine instructions to be processed. It is controlled by the Execution unit and the Address-Unit.

The Instruction unit reads the next instruction out of the Instruction queue decodes it and fills a second queue, the “Operation queue” with the next internal operations that must be performed by the Execution Unit.

The Execution Unit does the ‘real’ work; it executes operations or calls the Bus Unit to read an optional operand from memory.

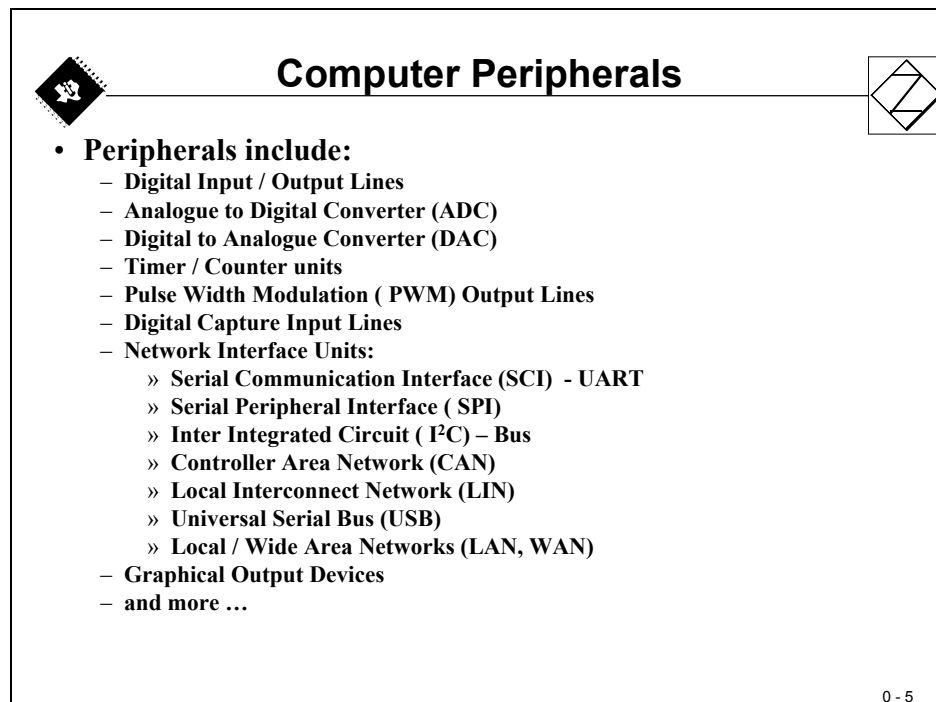
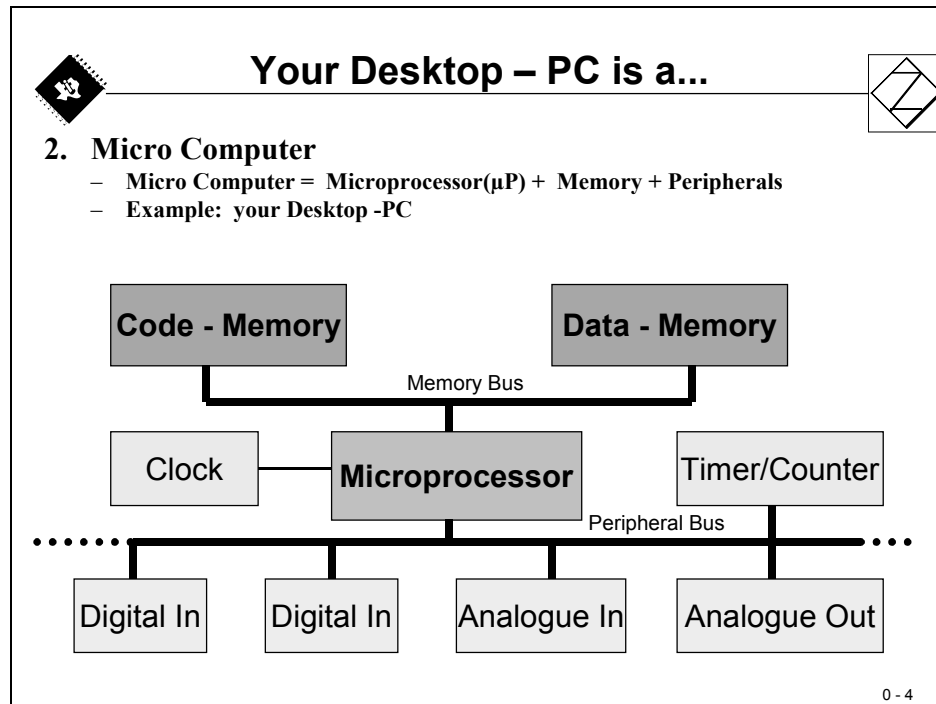
Once an instruction is completed, the Execution Unit forces the Address Unit to generate the address of the next instruction. If this instruction was already loaded into the Instruction queue, the operational speed is increased. This principle is called “cache”.

We could go much deeper into the secrets of a Microprocessor; eventually you can book another class at your university that deals with this subject much more in detail, especially into the pros and cons of Harvard versus Von-Neumann Machines, into RISC versus CISC, versions of memory accesses etc.

For now, let’s just keep in mind the basic operation of this type of device.

The Desktop – PC: a Micro Computer

When we add external devices to a Microprocessor, we end up with the set up for a computer system. We need to add external memory both for instructions (“code”) and data to be computed. We also have to use some sort of connections to the outside world to our system. In general, they are grouped into digital input/outputs and analogue input/outputs.



The Microcontroller : a single chip computer

As technology advances, we want the silicon industry to build everything that is necessary for a microcomputer into a single piece of silicon, and we end up with a microcontroller ("μC"). Of course nobody will try to include every single peripheral that is available or thinkable into a single chip – because nobody can afford to buy this "monster"-chip. On the contrary, engineers demand a microcontroller that suits their applications best and – for (almost) nothing. This leads to a huge number of dedicated microcontroller families with totally different internal units, different instruction sets, different number of peripherals and internal memory spaces. No customer will ask for a microcontroller with an internal code memory size of 16Mbytes, if the application fits easily into 64Kbytes.

Today, microcontrollers are built into almost every industrial product that is available on the market. Try to guess, how many microcontrollers you possess at home! The problem is you can't see them from outside the product. That is the reason why they are also called "embedded" computer or "embedded" controller. A sophisticated product such as the modern car is equipped with up to 80 microcontrollers to execute all the new electronic functions like antilock braking system (ABS), electronic stability program (ESP), adaptive cruise control (ACC), central locking, electrical mirror and seat adjustments, etc. On the other hand a simple device such as a vacuum cleaner is equipped with a microcontroller to control the speed of the motor and the filling state of the cleaner. Not to speak of the latest developments in vacuum cleaner electronics: the cleaning robot with lots of control and sensor units to do the housework – with a much more powerful μC of course.

Microcontrollers are available as 4, 8, 16, 32 or even 64-bit devices, the number giving the amount of bits of an operand that are processed in parallel. If a microcontroller is a 32-bit type, the internal data memory is connected to the core unit with 32 internal signal lines.



System on Chip




3. Microcontroller (μC)

- **Nothing more than a Micro Computer as a single silicon chip!**
- **All computing power AND input/output channels that are required to design a real time control system are „on chip“**
- **Guarantee cost efficient and powerful solutions for embedded control applications**
- **Backbone for almost every type of modern product**
- **Over 200 independent families of μC**
- **Both μP – Architectures („Von Neumann“ and „Harvard“) are used inside Microcontrollers**


0 - 6

A Digital Signal Processor

A Digital Signal Processor is a specific device that is designed around the typical mathematical operations to manipulate digital data that are measured by signal sensors. The objective is to process the data as quickly as possible to be able to generate an output stream of ‘new’ data in “real time”.




Digital Signal Processor




4. Digital Signal Processor (DSP)

- **Similar to a Microprocessor(μP), e.g. core of a computing system**
- **Additional Hardware Units to speed up computing of sophisticated mathematical operations:**
 - » **Additional Hardware Multiply Unit(s)**
 - » **Additional Pointer Arithmetic Unit(s)**
 - » **Additional Bus Systems for parallel access**
 - » **Additional Hardware Shifter for scaling and/or multiply/divide by 2ⁿ**

0 - 7



What are the typical DSP algorithms?



- **The Sum of Products (SOP) is the key element in most DSP algorithms:**

Algorithm	Equation
Finite Impulse Response Filter	$y(n) = \sum_{k=0}^M a_k x(n-k)$
Infinite Impulse Response Filter	$y(n) = \sum_{k=0}^M a_k x(n-k) + \sum_{k=1}^N b_k y(n-k)$
Convolution	$y(n) = \sum_{k=0}^N x(k)h(n-k)$
Discrete Fourier Transform	$X(k) = \sum_{n=0}^{N-1} x(n) \exp[-j(2\pi / N)nk]$
Discrete Cosine Transform	$F(u) = \sum_{x=0}^{N-1} c(u) \cdot f(x) \cdot \cos\left[\frac{\pi}{2N} u(2x+1)\right]$

0 - 8


The “Sum of Product” – Equation

We won't go into the details of the theory of Digital Signal Processing now. Again, look out for additional classes at your university to learn more about the math's behind this amazing part of modern technology. I highly recommend it. It is not the easiest topic, but it is worth it. Consider a future world without anybody that understands how a mobile phone or an autopilot of an airplane does work internally – a terrible thought.


To begin with, let's scale down the entire math's into one basic equation that is behind almost all approaches of Digital Signal Processing. It is the “Sum of Products”- formula. A new value ‘y’ is calculated as a sum of partial products. Two arrays “data” and “coeff” are multiplied as pairs and the products are added together. Depending on the data type of the input arrays we could solve this equation in floating point or integer mathematics. Integer is most often also called “fixed-point” math's (see Chapter 11).

Because of the TMS320F2812 is a fixed-point device, let's stay with this type of math's. If you look into chapter 1 of Texas Instruments C6000 Teaching CD-ROM, you will find a detailed discussion of pros and cons of fixed point versus floating point DSPs.

In a standard ANSI-C we can easily define two arrays of integer input data and the code lines that are needed to calculate the output value ‘y’:



Doing a SOP with a µP




$$y = \sum_{i=0}^3 data[i] * coeff[i]$$

- Task : use a Desktop - PC and code the equation into a common C-compiler system, e.g. Microsoft Visual Studio.Net**
- A C-Code Solution could look like this:**


```
#include <stdio.h>
int data[4]={1,2,3,4};
int coeff[4]={8,6,4,2};
int main(void)
{
    int i;
    int result =0;
    for (i=0;i<4;i++)
        result += data[i]*coeff[i];
    printf("%i",result);
    return 0;
}
```

0 - 9

If we look a little bit more in detail into the tasks that needs to be solved by a standard processor we can distinguish 10 steps. Due to the sequential nature of this type of processor, it can do only one of the 10 steps at one time. This will consume a considerable amount of computing power of this processor. For our tiny example, the processor must loop between step 3 and step 10 a total of four times. For real Digital Signal Processing the SOP – procedure is going to much higher loop repetitions – forcing the standard processor to spend even more computing power.




6 Basic Operations of a SOP




$$y = \sum_{i=0}^3 data[i] * coeff[i]$$

- **What will a Pentium be forced to do?**
 1. Set a Pointer1 to point to data[0]
 2. Set a second Pointer2 to point to coeff[0]
 3. Read data[i] into core
 4. Read coeff[i] into core
 5. Multiply data[i]*coeff[i]
 6. Add the latest product to the previous ones
 7. Modify Pointer1
 8. Modify Pointer2
 9. Increment i;
 10. If i<3 , then go back to step 3 and continue
- Steps 3 to 8 are called “6 Basic Operations of a DSP”
- A DSP is able to execute all 6 steps in one single machine cycle!

0 - 10



SOP machine code of a µP




Address	M-Code	Assembly - Instruction
10:	for (i=0;i<4;i++)	
00411960	C7 45 FC 00 00 00 00	mov dword ptr [i],0
00411967	EB 09	jmp main+22h (411972h)
00411969	8B 45 FC	mov eax,dword ptr [i]
0041196C	83 C0 01	add eax,1
0041196F	89 45 FC	mov dword ptr [i],eax
00411972	83 7D FC 04	cmp dword ptr [i],4
00411976	7D 1F	jge main+47h (411997h)
11:	result += data[i]*coeff[i];	
00411978	8B 45 FC	mov eax,dword ptr [i]
0041197B	8B 4D FC	mov ecx,dword ptr [i]
0041197E	8B 14 85 40 5B 42 00	mov edx,dword ptr[eax*4+425B40h]
00411985	0F AF 14 8D 50 5B 42 00	imul edx,dword ptr[ecx*4+425B50h]
0041198D	8B 45 F8	mov eax,dword ptr [result]
00411990	03 C2	add eax,edx
00411992	89 45 F8	mov dword ptr [result],eax
00411995	EB D2	jmp main+19h (411969h)


0 - 11

A SOP executed by a DSP

If we apply the SOP-task to a Digital Signal Processor of fixed-point type the ANSI-C code looks identical to the standard processor one. The difference is the output of the compilation! When you compare slide 13 with slide 11 you will notice the dramatic reduction in the consumption of the memory space and number of execution cycles. A DSP is much more appropriate to calculate a SOP in real time! Ask your professor about the details of the two slides!



Doing a SOP with a DSP




$$y = \sum_{i=0}^3 data[i] * coeff[i]$$


- **Now: use a DSP-Development System and code the equation into a DSP C-compiler system, e.g. Texas Instruments Code Composer Studio**
- **C-Code Solution is identical:**

```
int data[4]={1,2,3,4};
int coeff[4]={8,6,4,2};
int main(void)
{
    int i;
    int result =0;
    for (i=0;i<4;i++)
        result += data[i]*coeff[i];
    printf("%i",result);
    return 0;
}
```

0 - 12



DSP-Translation into machine code



Address	MCode	Assembly Instruction
0x8000	FF69	SPM 0
0x8001	8D04 0000R	MOVL XAR1,#data
0x8003	76C0 0000R	MOVL XAR7,#coeff
0x8005	5633	ZAPA
0x8006	F601	RPT #1
0x8007	564B 8781	DMAC ACC:P,*XAR1++,*XAR7++
0x8009	10AC	ADDL ACC,P<<PM
0x800A	8D04 0000R	MOVL XAR1,#y
0x800B	1E81	MOVL *XAR1,ACC

Example: Texas Instruments TMS320F2812
 Space : 12 Code Memory ; 9 Data Memory
 Execution Cycles : 10 @ 150MHz = 66 ns

0 - 13

A Digital Signal Controller

Finally, a Digital Signal Controller (DSC) is a new type of microcontroller, where the processing power is delivered by a DSP – a single chip device combining both the computing power of a Digital Signal Processor and the embedded peripherals of a single chip computing system.

For advanced real time control systems with a high amount of mathematical calculations, a DSC is the first choice.

Today there are only a few manufacturers offering DSC's. Due to the advantages of DSC's for many projects, a number of silicon manufacturers are developing this type of controller.

This tutorial is based on the Texas Instruments TMS320F2812, a 32-bit fixed point Digital Signal Controller (DSC).



Digital Signal Controller (DSC)



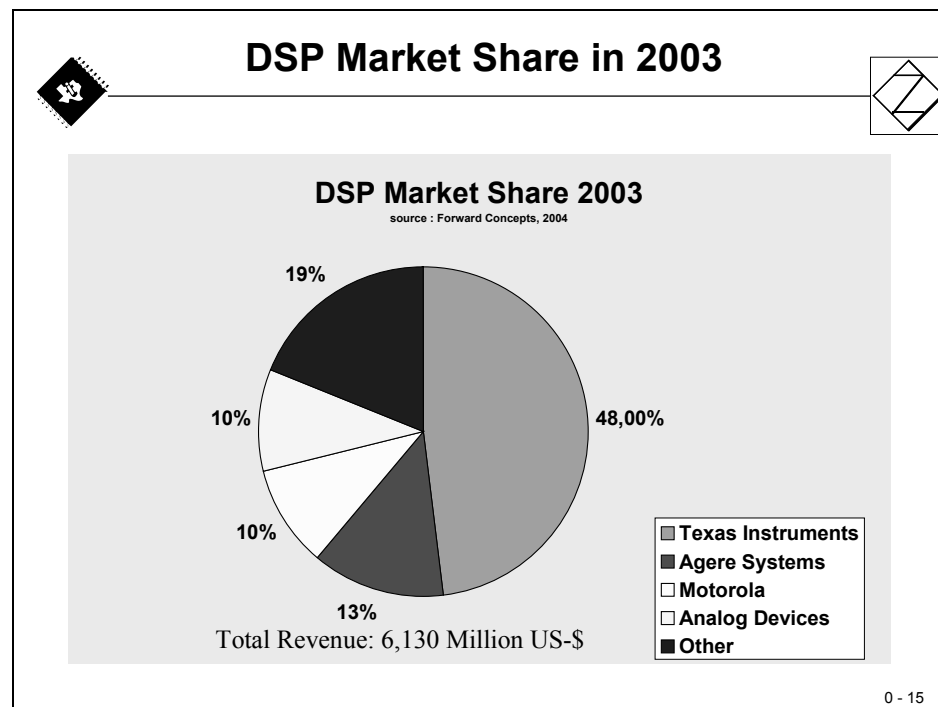
5. Digital Signal Controller (DSC)

- **recall: a Microcontroller(μ C) is a single chip Microcomputer with a Microprocessor(μ P) as core unit.**
- **Now: a Digital Signal Controller(DSC) is a single chip Microcomputer with a Digital Signal Processor(DSP) as core unit.**
- **By combining the computing power of a DSP with memory and peripherals in one single device we derive the most effective solution for embedded real time control solutions that require lots of math operations.**
- **DSC –Example: Texas Instruments C2000 family.**

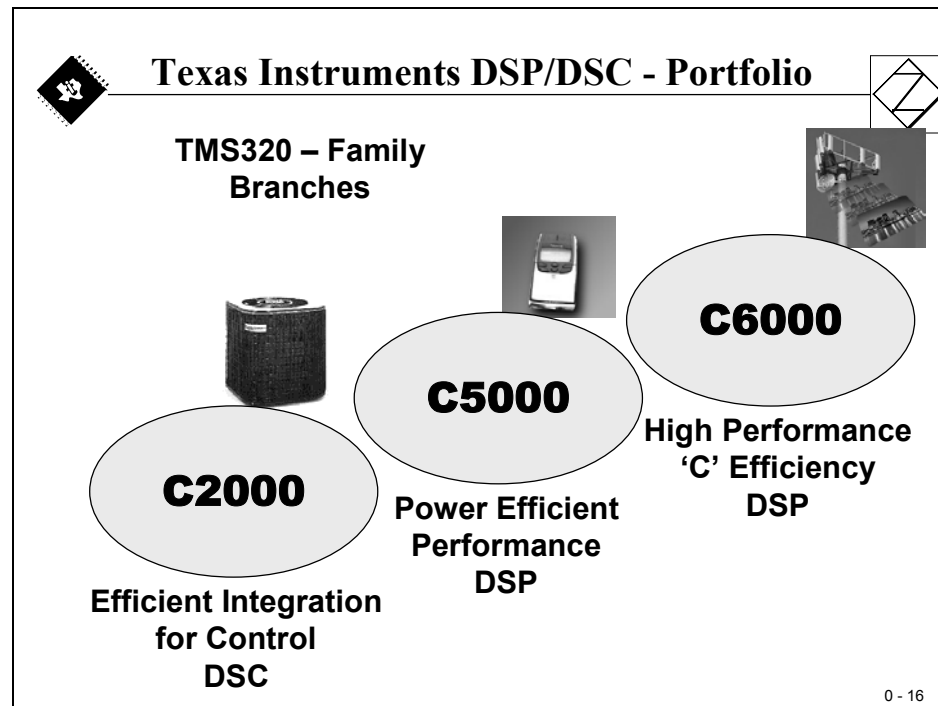
0 - 14

DSP Competition

There are only a few global players in the area of DSP and DSC. As you can see from the next slide (for more details, go to: www.fwdconcepts.com), Texas Instruments is the absolute leader in this area. A working knowledge of TI-DSP will help you to master your professional career.



Texas Instruments DSP – Portfolio



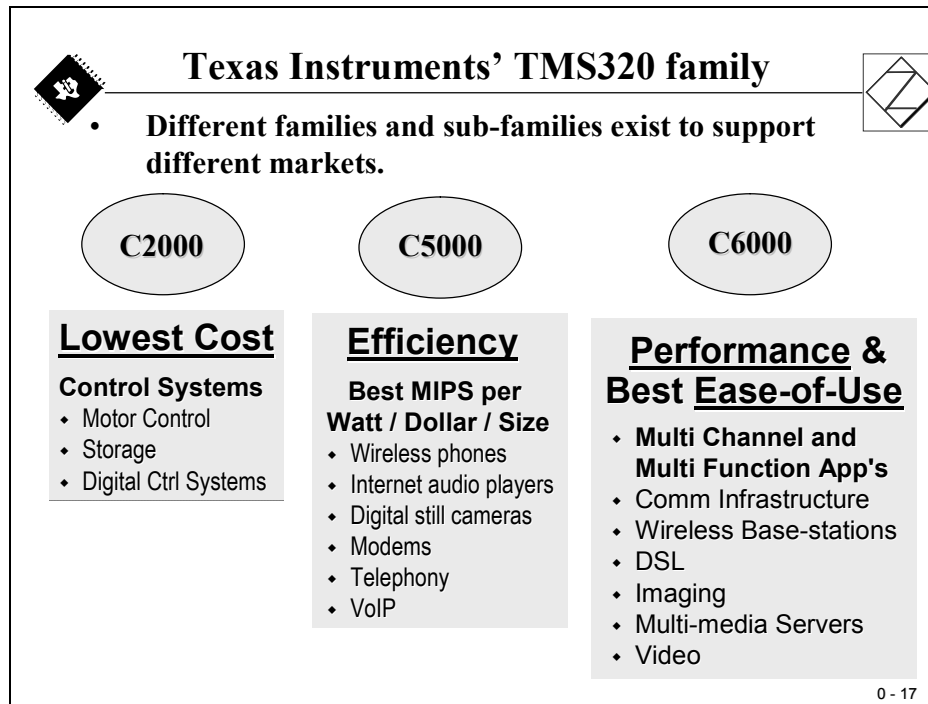
The DSP / DSC – portfolio of Texas instruments is split into three major device families, called C2000, C5000 and C6000.

The C6000 branch is the most powerful series of DSP in computing power. There are floating – point as well as fixed – point devices in this family. The application fields are image processing, audio, multimedia server, base stations for wireless communication etc.

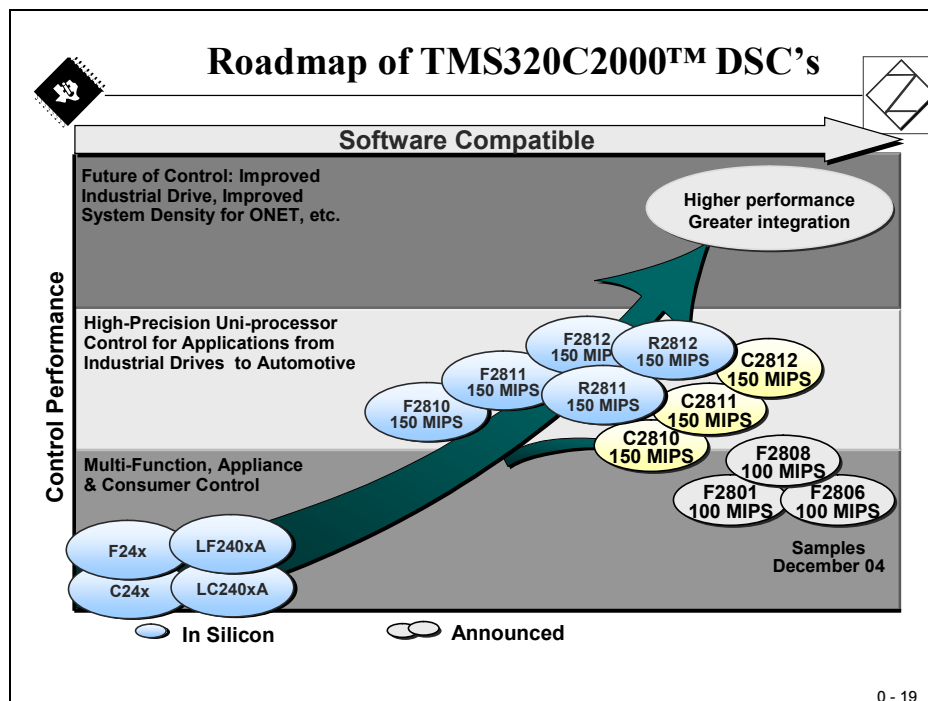
The C5000 family is focused on mobile systems with very efficient power consumption per MIPS. Its main application area is cell phone technology.

The C2000 – group is dedicated to Digital Signal Control (DSC), as you have learned from the first slides and is a very powerful solution for real time control applications.


The next slide summarizes the main application areas for the 3 Texas Instruments families of DSP.




For the C2000 – branch we can distinguish between 2 groups of fixed-point DSC's: a 16-bit group, called TMS320C24x and a 32-bit group, called TMS320C28x.




TMS320F28x Roadmap




Broad C28x™ Application Base







Digital Power Supply
Provides control, sensing, PFC, and other functions




Optical Networking
Control of laser diode



Printer
Print head control
Paper path motor control




Evaluating Other Segments
eg. Musical Instruments




Non-traditional Motor Control
Many new cool applications to come

0 - 19



TI C2000: Portfolio for Embedded Applications



	F2812	F2810	LF2407A	LF2406A	LF2403A	LF2402A	LF2401A	LC2406A	LC2404A	LC2402A	LC2401A	F243	F241	F240	C242
CPU	32bit	32 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16 bit	16bit	16bit
MIPS	150	150	40	40	40	40	40	40	40	40	40	20	20	20	20
RAM (words)	18K	18K	2.5K	2.5K	1.0K	1.0K	1.0K	2.5K	1.5K	544	1.0K	544	544	544	544
ROM (words)								32K	16K	6K	8K				4K
Flash (words)	128K	64K	32K	32K	16K	8K	8K					8K	8K	16K	
BootROM (words)	4K	4K	256	256	256	256	256								
Event Manager															
CAP/QEP	6/6	6/6	6/4	6/4	3/2	3/2	1/0	6/4	6/4	3/2	1/0	3/2	3/2	4/2	3/2
PWM(CMP)	16	16	16	16	8	8	7	16	16	8	7	8	8	12	8
TIMER	7	7	4	4	2	2	2	4	4	2	2	2	2	3	2
ADC Resolution	12-bit	12-bit	10-bit	10-bit	10-bit	10-bit	10-bit	10 bit	10-bit	10-bit	10-bit	10-bit	10-bit	10-bit	10-bit
# ofChan	16	16	16	16	8	8	5	16	16	8	5	8	8	16	8
Conv time	200ns	200ns	500ns	500ns	500ns	500ns	500ns	375ns	375ns	425ns	500ns	900ns	900ns	6.1us	900ns
McBSP	✓	✓													
EXMIF	✓	✓	✓									✓		✓	
Watch Dog	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SPI	✓	✓	✓	✓	✓			✓	✓			✓	✓	✓	
SCI (UART)	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1
CAN	✓	✓	✓	✓	✓			✓				✓			
Volts (V)	1.8 core 3.3 I/O	1.8core 3.3 I/O	3.3	3.3	3.3	3.3	3.3	3.3	3.3	3.3	3.3	5.0	5.0	5.0	5.0
# I/O	56	56	41	41	21	21	13	41	41	21	13	32	26	28	26
Package	176LQFP 179u" BGA	128LQFP	144LQFP	100LQFP	64LQFP	64PQFP	32LQFP	100LQFP	100LQFP	64PQFP	32LQFP	144LQFP	64PQFP 68PLCC	132PQFP	64PQFP 68PLCC

0 - 20

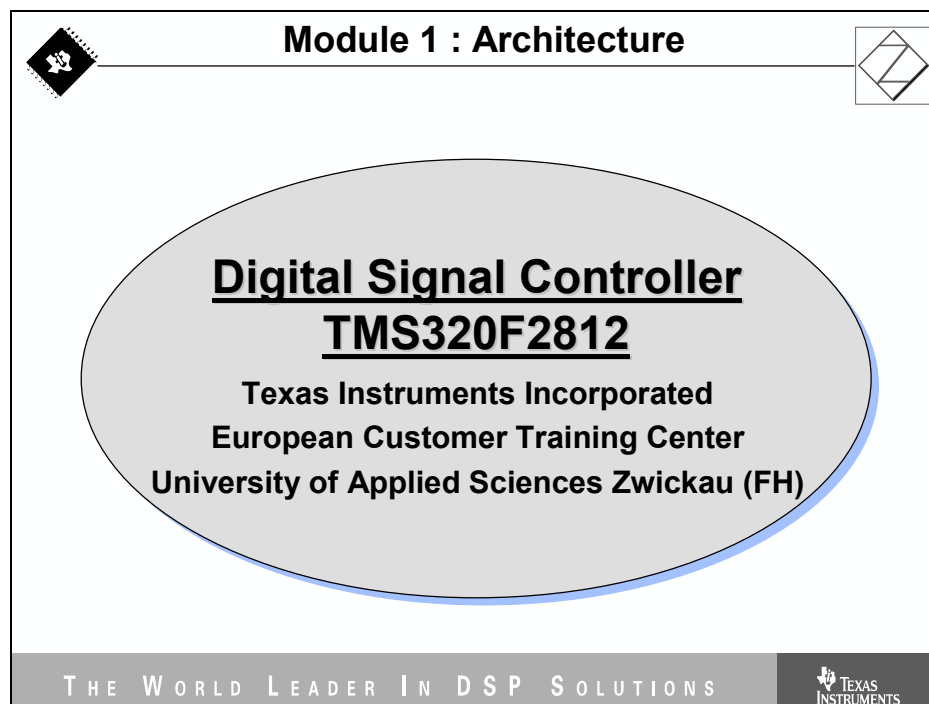
Architecture

Introduction

Since a Digital Signal Processor is capable of executing six basic operations in a single instruction cycle, the architecture of the TMS320F2812 must reflect this feature in some way. Recall this key point when we look into the details of this Digital Signal Controller (DSC). It will help you to understand the ‘philosophy’ behind the device with its different hardware units. Doing six basic math’s operations is no magic; we will find all the hardware modules that are required to do so in this chapter.

Among other things, we will discuss the following parts of the architecture:

- Internal bus structure
- CPU
- Hardware Multiplier, Arithmetic-Logic-Unit, Hardware-Shifter
- Register Structure
- Memory Map



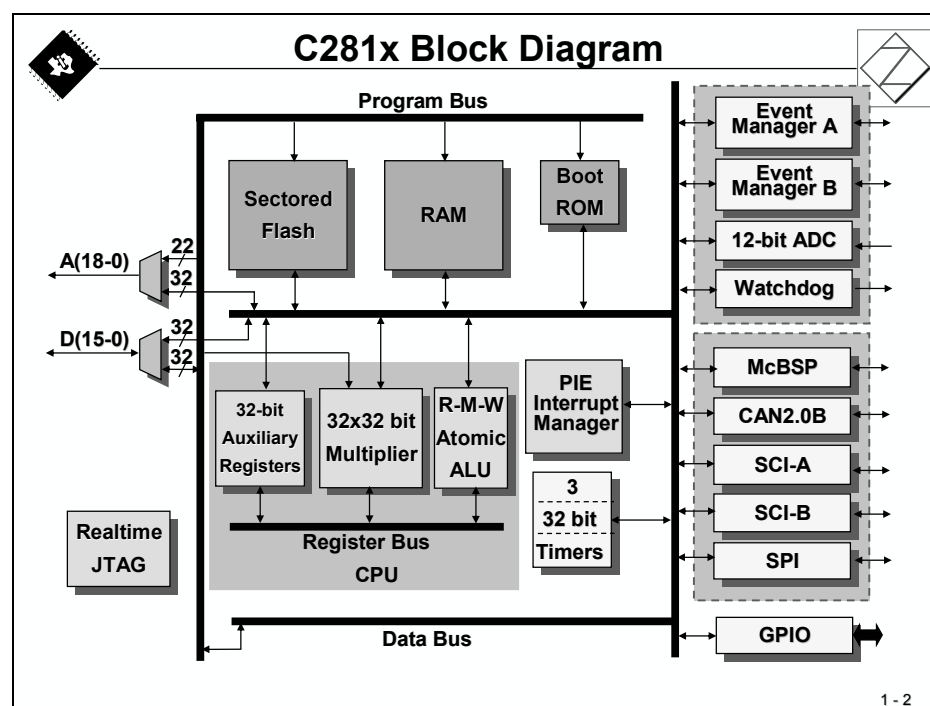
Module Topics

Architecture	1-1
<i>Introduction</i>	<i>1-1</i>
<i>Module Topics.....</i>	<i>1-2</i>
TMS320F2812 Block Diagram	1-3
The F2812 CPU	1-4
F2812 Math Units.....	1-5
Data Memory Access.....	1-6
Internal Bus Structure.....	1-7
Atomic Arithmetic Logic Unit (ALU).....	1-8
Instruction Pipeline.....	1-9
Memory Map.....	1-10
Code Security Module	1-11
Interrupt Response.....	1-12
Operating Modes	1-13
Reset Behaviour.....	1-14
Summary of TMS320F2812 Architecture	1-15

TMS320F2812 Block Diagram

The TMS320F2812 Block Diagram can be divided into 4 functional blocks:

- Internal & External Bus System
- Central Processing Unit (CPU)
- Memory
- Peripherals



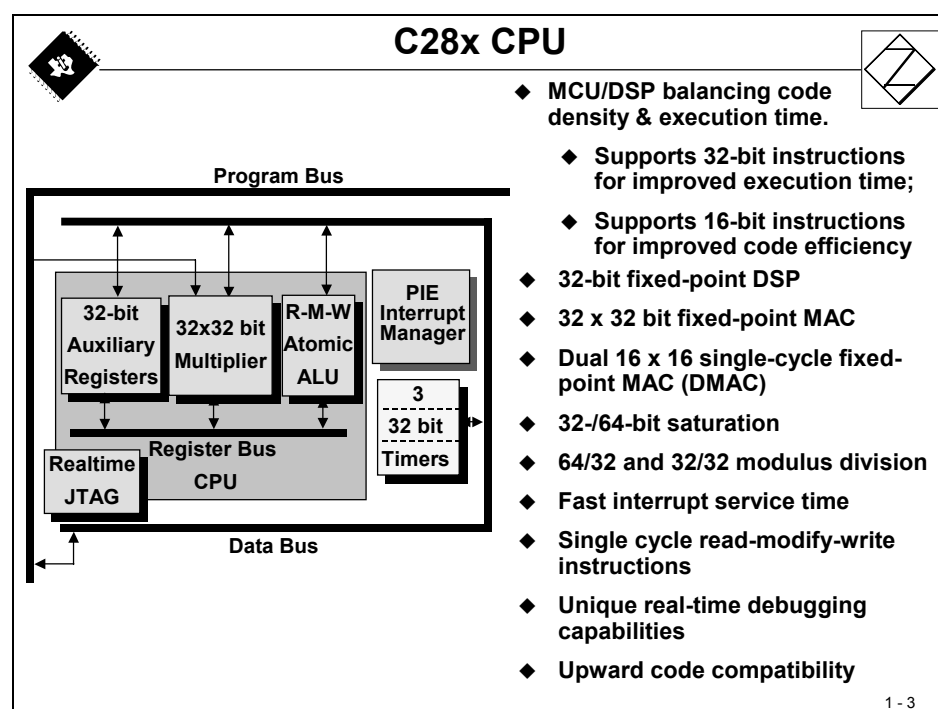
To be able to fetch two operands from memory into the central processing unit in a single clock cycle, the F2812 is equipped with two independent bus systems – Program Bus and Data Bus. This type of machine is called a “Harvard-Architecture”. Due to the ability of the F2812 to read operands not only from data memory but also from program memory, Texas Instruments calls this device a “modified Harvard-Architecture”. The “bypass”-arrow in the bottom left corner of slide 1-2 indicates this additional feature.

On the left side of the slide you will notice two multiplexer blocks for data (D15-D0) and address (A18-A0). It is an interface to connect external devices to the F2812. Please note that (1) the width of the external data bus is only 16 bits and that (2), you can’t access the external program bus data and the data bus data at the same time. Compared to a single cycle for internal access to two 32-bit operands, it takes at least 4 cycles to do the same with external memory!

The F2812 CPU

The F2812 –CPU is able to execute most of the instructions to perform register-to-register operations and a range of instructions that are commonly used by micro controllers, e.g. byte packing and unpacking and bit manipulation in a single cycle. The architecture is also supported by powerful addressing modes, which allow the compiler as well as the assembly programmer to generate compact code that almost corresponds one-to-one with the C code.

The F2812 is as efficient in DSP math tasks as it is in the system control tasks that are typically handled by microcontroller devices. This efficiency removes the need for a second processor in many systems.



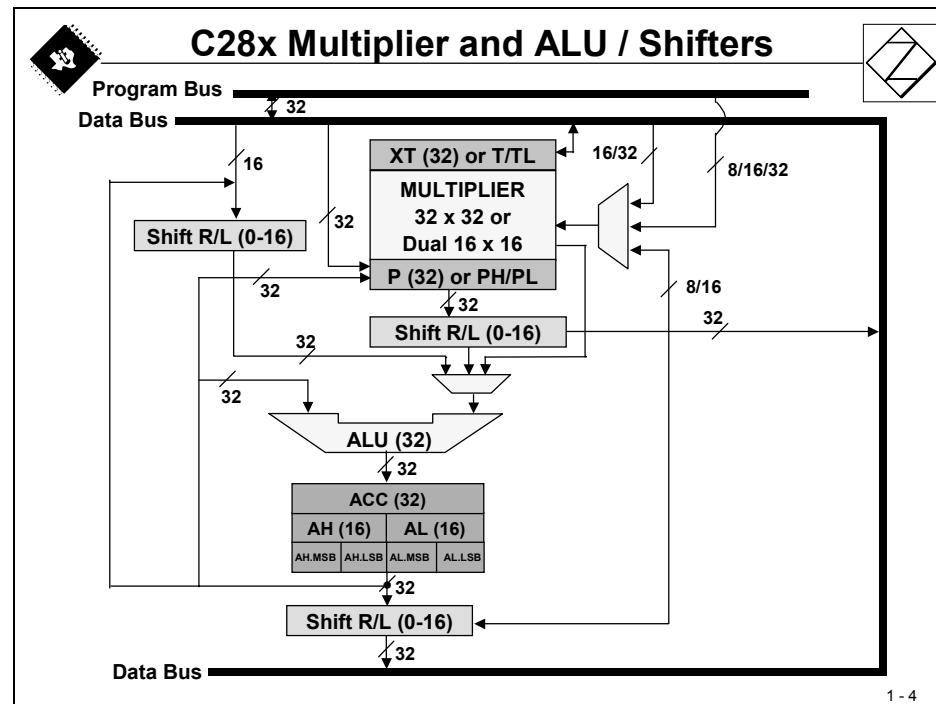
Three 32-bit timers can be used for general timing purposes or to generate hardware driven time periods for real time operating systems. The Peripheral Interrupt Expansion Manager (PIE) allows fast interrupt response to the various sources of external and internal signals and events. The PIE-Manager covers individual interrupt vectors for all sources.

A 32 by 32 bit hardware multiplier and a 32-bit arithmetic logic unit (ALU) can be used in parallel to execute a multiply and an add operation simultaneously. The auxiliary register bank is equipped with its own arithmetic logic unit (ARAU) – also used in parallel to perform pointer arithmetic.

The JTAG-interface is a very powerful tool to support real-time data exchange between the DSC and a host during the debug phase of project development. It is possible to watch variables while the code is running in real time, without any delay to the control code.

F2812 Math Units

The 32 x 32-bit Multiply and Accumulate (MAC) capabilities of the F2812 and its internal 64-bit processing capabilities, enable this DSC to efficiently handle higher numerical resolution problems that would otherwise demand a more expensive floating-point processor solution. Along with this is the capability to perform two 16 x 16-bit multiply and accumulate instructions simultaneously or Dual MAC's (DMAC).



Multiplication uses the XT register to hold the first operand and multiply it by a second operand that is loaded from memory. If XT is loaded from a data memory location and the second operand is fetched from a program memory location, a single-cycle multiply operation can be performed. The result of a multiplication is loaded into register P (product) or directly into the accumulator (ACC). Recall, if you multiply 32 x 32 bit numbers, what is the size of the result? Answer: 64-bit. The F2812 instruction set includes two groups of multiply operations to load both halves of the result into P and ACC.

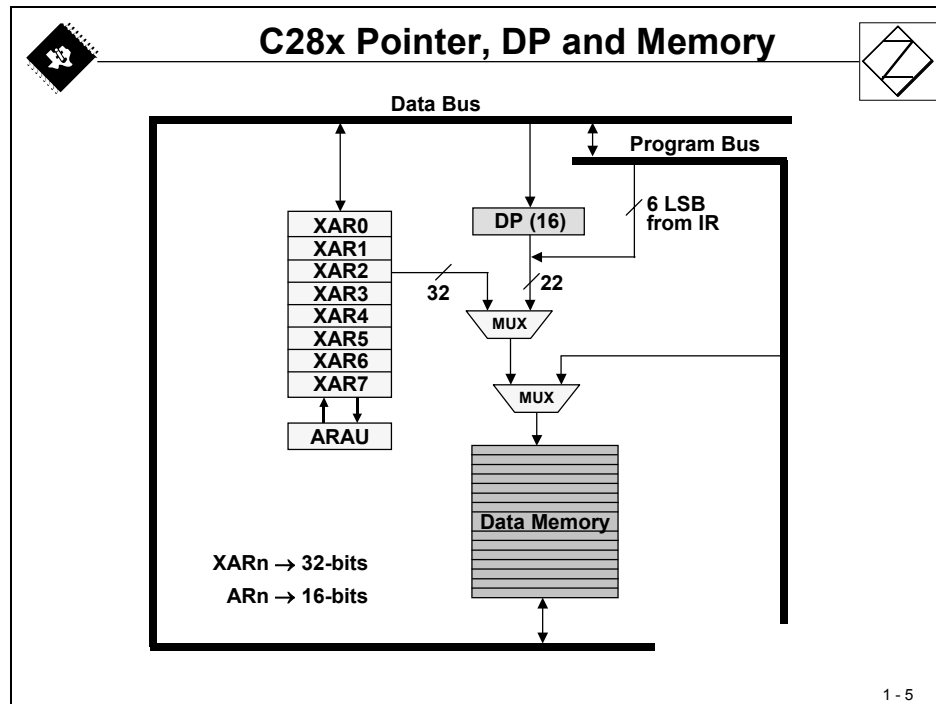
Three hardware shifters can be used in parallel to other hardware units of the CPU. Shifters are usually used to scale intermediate numbers in a real time control loop or to multiply/divide by 2^n .

The arithmetic logic unit (ALU) is doing the 'rest' of the math's. The first operand is always the content of the Accumulator (ACC) or a part of it. The second operand for an operation is loaded from data memory, from program memory, from the P register or directly from the multiply unit.

Data Memory Access

Two basic methods are available to access data memory locations:

- Direct Addressing Mode
- Indirect Addressing Mode



Direct addressing mode generates the 22-bit address for a memory access from two sources – a 16-bit register “Data Page (DP)” for the highest 16 bits plus another 6 bits taken from the instruction. Advantage: Once DP is set, we can access any location of the selected page, in any order. Disadvantage: If the code needs to access another page, DP must be adjusted first.

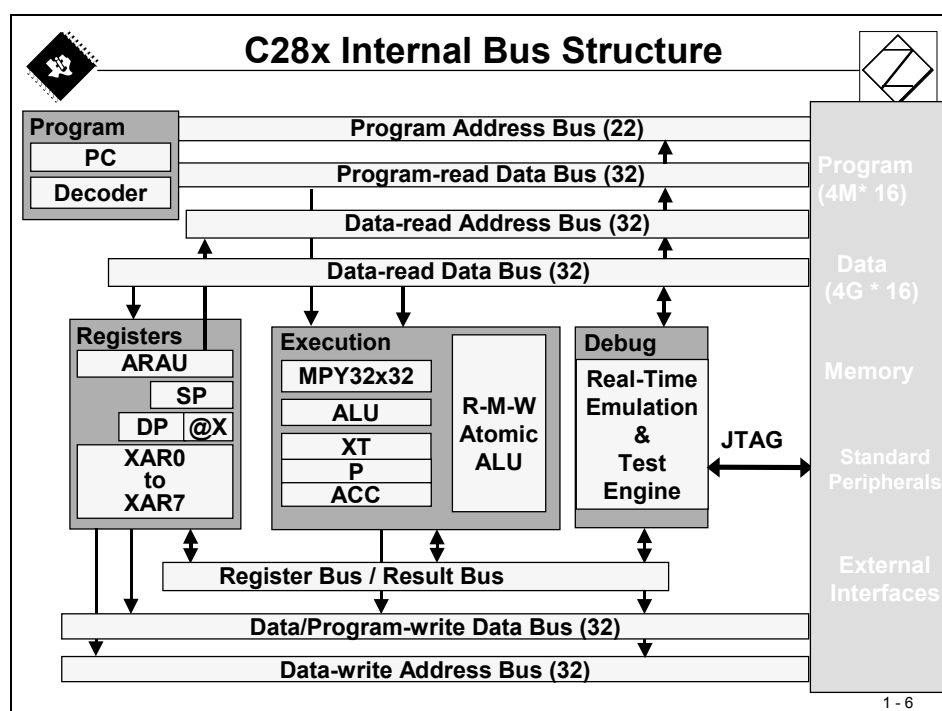
Indirect addressing mode uses one of eight 32-bit XARn registers to hold the 32-bit address of the operand. Advantage: With the help of the ARAU, pointer arithmetic is available in the same cycle in which an access to a data memory location is made. Disadvantage: A random access to data memory needs a new setup of the pointer register.

The auxiliary register arithmetic unit (ARAU) is able to perform pointer manipulations in the same clock cycle as access is made to a data memory location. The options for the ARAU are: post-increment, pre-decrement, index addition and subtraction, stack relative operation, circular addressing and bit-reverse addressing with additional options.

Internal Bus Structure

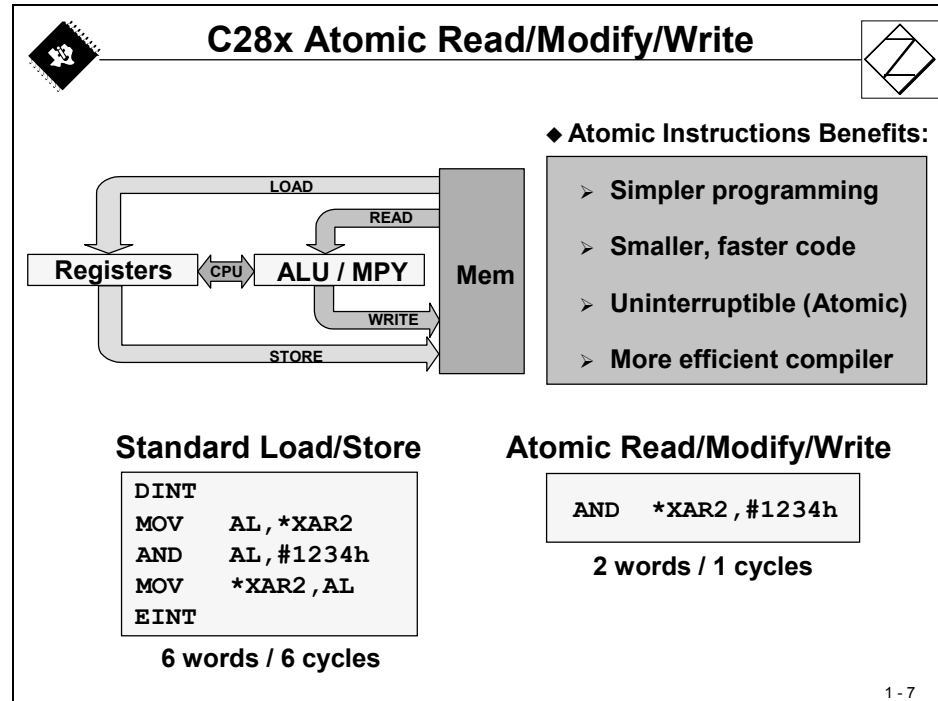
As with many DSP type devices, multiple busses are used to move data between memory locations, peripheral units and the CPU. The F2812 memory bus architecture contains:

- A program read bus (22 bit address line and 32 bit data line)
- A data read bus (32 bit address line and 32 bit data line)
- A data write bus (32 bit address line and 32 bit data line)



The 32-bit-wide data busses enable single cycle 32-bit operations. This multiple bus architecture, known as a Harvard Bus Architecture enables the C28x to fetch an instruction, read a data value and write a data value in a single cycle. All peripherals and memories are attached to the memory bus and will prioritise memory accesses.

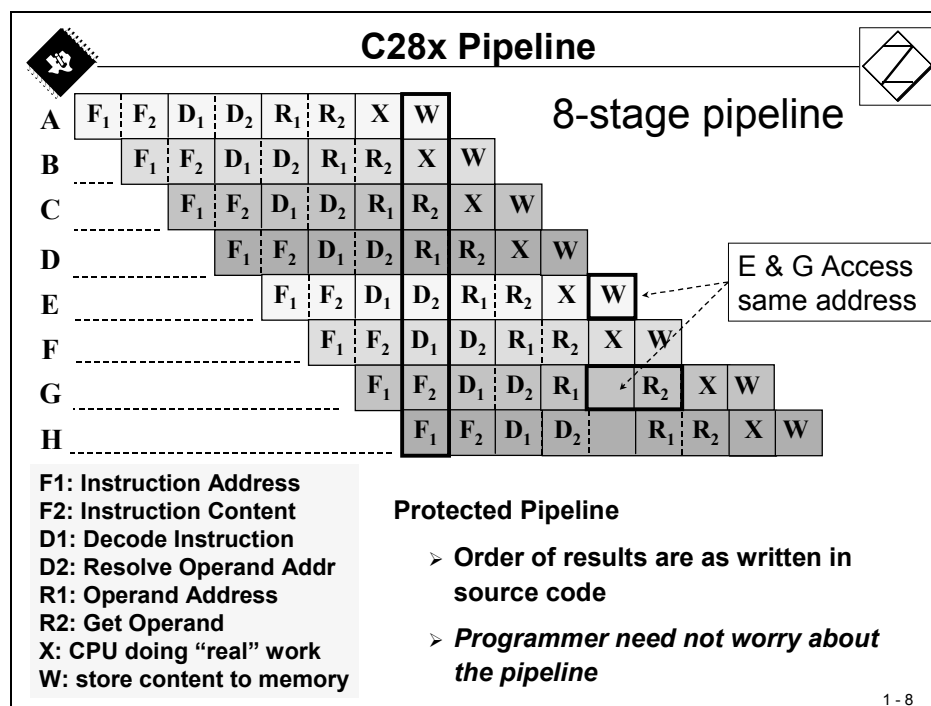
Atomic Arithmetic Logic Unit (ALU)



Atomics are small common instructions that are non-interruptible. The atomic ALU capability supports instructions and code that manages tasks and processes. These instructions usually execute several cycles faster than traditional coding.

Instruction Pipeline

The F2812 uses a special 8-stage protected pipeline to maximize the throughput. This protected pipeline prevents a write to and a read from the same location from occurring out of sequence. This pipelining also enables the C28x to execute at high speeds without resorting to expensive high-speed memories. Special branch-look-ahead hardware minimizes the delay when jumping to another address. Special conditional store operations further improve the system performance.

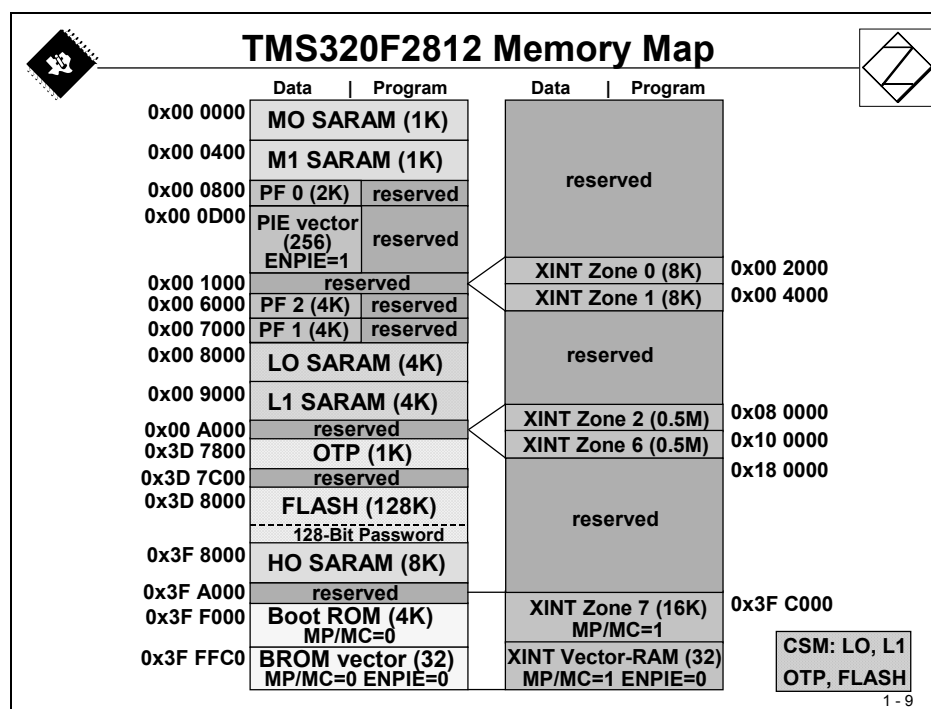


Each instruction goes through 8 stages until final completion. Once the pipeline is filled with instructions, one instruction is executed per clock cycle. For a 150MHz device, this equates to 6.67ns per instruction. The stages are:

- F1: Generate Instruction Address at program bus address lines.
- F2: Read the instruction from program bus data lines.
- D1: Decode Instruction
- D2: Calculate Address information for operand(s) of the instruction
- R1: Load operand(s) address to data and/or program bus address lines
- R2: Read Operand
- X: Execute the instruction
- W: Write back result to data memory

Memory Map

The memory space on the F2812 is divided into program and data space. There are several different types of memory available that can be used as both program or data space. They include flash memory, single access RAM (SARAM), expanded SARAM, and Boot ROM which is factory programmed with boot software routines or standard tables used in math related algorithms. Memory space width is always 16 bit.



The F2812 can access memory both on and off the chip. The F2812 uses 32-bit data addresses and 22-bit program addresses. This allows for a total address reach of 4G words (1 word = 16 bits) in data space and 4M words in program space. Memory blocks on all F2812 designs are uniformly mapped to both program and data space.

The memory map above shows the different blocks of memory available to the program and data space.

The non-volatile internal memory consists of a group of FLASH-memory sections, a boot-ROM for up to six reset-startup options and a one-time-programmable (OTP) area. FLASH and OTP are usually used to store control code for the application and/or data that must be present at reset. To load information into FLASH and OTP one need to use a dedicated download program, that is also part of the Texas Instruments Code Composer Studio integrated design environment.


Volatile Memory is split into 5 areas (M0, M1, L0, L1 and H0) that can be used both as code memory and data memory.

PF0, PF1 and PF2 are Peripheral Frames that cover control and status registers of all peripheral units ("Memory Mapped Registers").


Code Security Module

There is an internal security module available in all F28x family members. It is based on a 128-bit password that is written by the *software developer* into the last 8 memory spaces of the internal FLASH (0x3F 7FF8 to 0x3F 7FFF). Once a pattern is written into this area, all further accesses to any of the memory areas covered by this Code Security Module (CSM) are denied, as long as the *user* does not write an identical pattern into password registers of frame PF0.

NOTE: If you write any pattern into the password area *by accident*, there is no way to get access to this device any more! So please be careful and do not upset your laboratory technician!



Code Security Module



- ◆ Prevents reverse engineering and protects valuable intellectual property

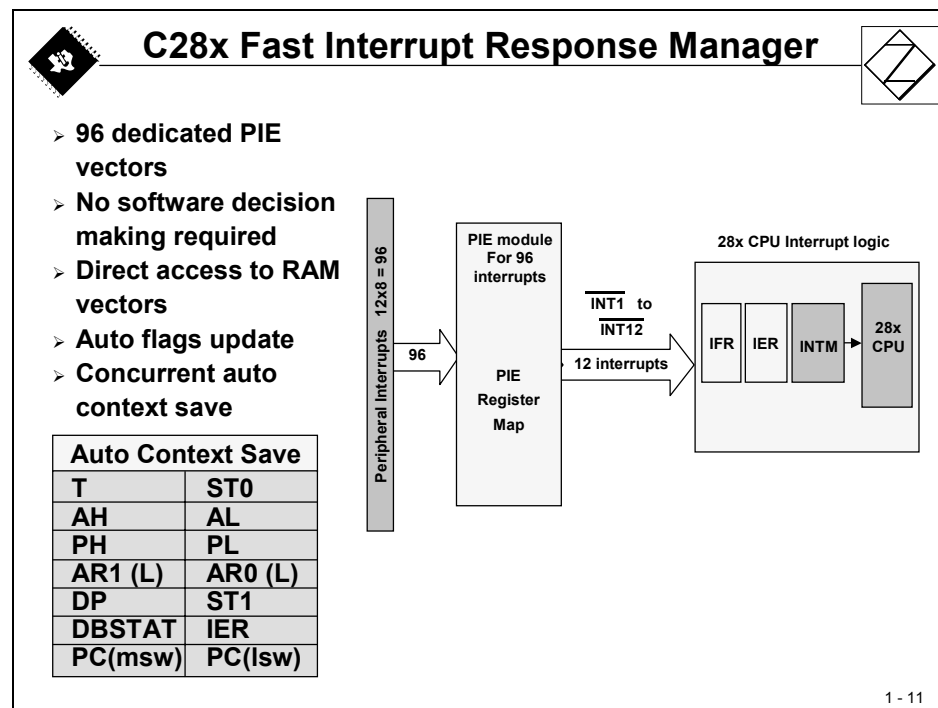
0x00 8000	LO SARAM (4K)
0x00 9000	L1 SARAM (4K)
0x00 A000	reserved
0x3D 7800	OTP (1K)
0x3D 7C00	reserved
0x3D 8000	FLASH (128K)
	128-Bit Password

- ◆ 128-bit user defined password is stored in Flash
- ◆ 128-bits = $2^{128} = 3.4 \times 10^{38}$ possible passwords
- ◆ To try 1 password every 2 cycles at 150 MHz, it would take at least 1.4×10^{23} years to try all possible combinations!

1 - 10

Interrupt Response

The fast interrupt response, with automatic “context” save of critical registers, resulting in a device that is capable of servicing many asynchronous events with minimal latency. Here “context” means all the registers you need to save so that you can go away and carry out some other process, then come back to exactly where you left. F2812 devices implement a zero cycle penalty to save and restore the 14 registers during an interrupt. This feature helps to reduce the interrupt service routine overheads.



We will look in detail into the F2812’s interrupt system in Module 4 of this tutorial. The Peripheral Interrupt Expansion (PIE) – Unit allows the user to specify individual interrupt service routines for up to 96 internal and external interrupt events. All possible 96 interrupt sources share 14 maskable interrupt lines (INT1 to INT14), 12 of them are controlled by the PIE – module.

The auto context save loads 14 important CPU registers, shown at the slide above, into a stack memory, which is pointed to by a stack pointer (SP) register. The stack is part of the data memory and must reside in the lower 64K words of data memory.

Operating Modes

The F2812 is one of several members of the fixed-point generations of digital signal processors (DSP's) in the TMS320 family. The F2812 is source-code and object-code compatible with the C27x. In addition, the F2812 is source code compatible with the 24x/240x DSP and previously written code can be reassembled to run on a F2812 device. This allows for migration of existing code onto the F2812.

C28x / C24x Modes			
Mode Type	Mode Bits		Compiler Option
	OBJMODE	AMODE	
C24x Mode	1	1	-v28 -m20
C28x Mode	1	0	-v28
Test Mode (default)	0	0	-v27
Reserved	0	1	

> **C24x source-compatible mode:**
 > Allows you to run C24x source code which has been reassembled using the C28x code generation tools (need new vectors)
 > **C28x mode:**
 > Can take advantage of all the C28x native features

Actually the F28x silicon is able to operate in three different modes:

- C28x – Mode - takes advantage of all 32-bit features of the device
- C24x – Mode - source code compatibility to the 16-bit family members
- C27x – Mode - intermediate operating mode, test purposes only.

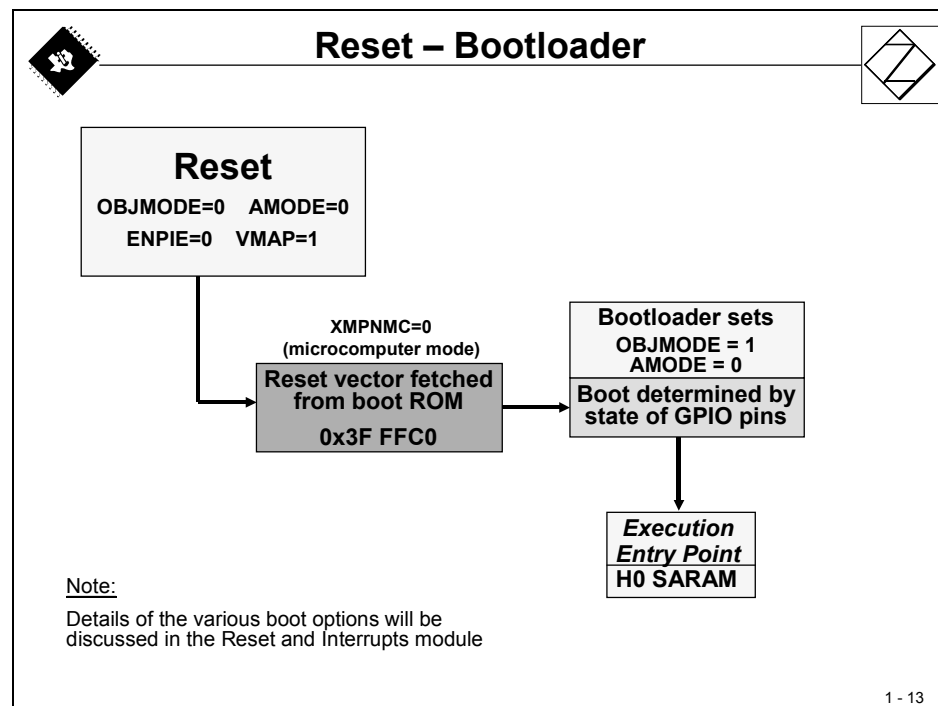
After RESET, the device behaves like a C27x device. To take advantage of the full computing power of a C28x device, the control flag “OBJMODE” must be set to 1. If you are using a C-compiler generated program, the start function of the C environment takes care of setting OBJMODE to 1. But, if you decide to develop an assembler language based solution, your first task after reset is to bring the device into C28x – mode manually.

Reset Behaviour


After a valid RESET-signal is applied to the F2812, the following sequence depends on some external pins on this DSC.

If the pin “XMPNMC” is connected to ‘1’, the F2812 tries to load the next instruction from address 0x3F FFC0 from an *external memory* at this position. This is the “Microprocessor”-Mode, loading instructions from external code memory.


If the pin “XMPNMC” is connected to ‘0’, the F2812 jumps directly into the *internal boot code* memory at address 0x3F FFC0. We call this mode “Microcontroller”-Mode. The code in this memory location has been developed by TI to be able to distinguish between 6 different start options for the F2812. The actual option is selected by the status of 4 more pins during this phase. For our tutorial we use the volatile memory H0 as code memory and its first address as the execution entry point.



Summary of TMS320F2812 Architecture



Summary



- ◆ High performance 32-bit DSP
- ◆ 32 x 32 bit or dual 16 x 16 bit MAC
- ◆ Atomic read-modify-write instructions
- ◆ 8-stage fully protected pipeline
- ◆ Fast interrupt response manager
- ◆ 128Kw on-chip flash memory
- ◆ Code security module (CSM)
- ◆ Two event managers
- ◆ 12-bit ADC module
- ◆ 56 shared GPIO pins
- ◆ Watchdog timer
- ◆ Communications peripherals

1 - 14

This page was intentionally left blank.

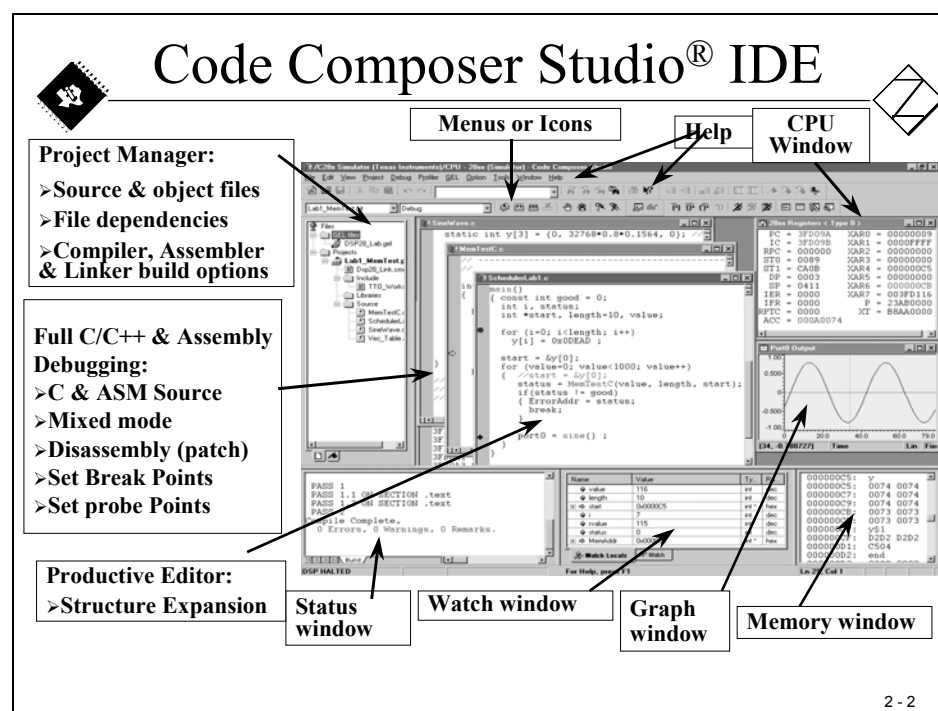
Program Development Tools

Introduction

The objective of this module is to understand the basic functions of the Code Composer Studio® Integrated Design Environment for the C2000 - Family of Texas Instruments Digital Signal Processors. This involves understanding the basic structure of a project in C and Assembler - coded source files, along with the basic operation of the C-Compiler, Assembler and Linker

Code Composer Studio IDE

Code Composer Studio is the environment for project development and for all tools needed to build an application for the C2000-Family.

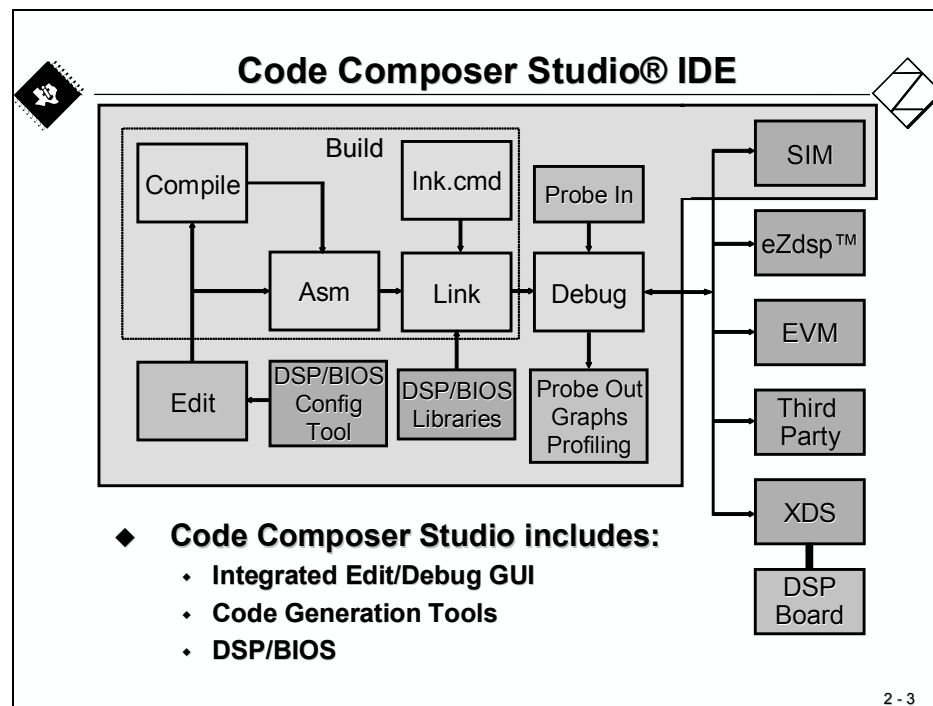


Module Topics

Program Development Tools.....	2-1
<i>Introduction</i>	<i>2-1</i>
<i>Code Composer Studio IDE.....</i>	<i>2-1</i>
<i>Module Topics.....</i>	<i>2-2</i>
<i>The Software Flow</i>	<i>2-3</i>
<i>Code Composer Studio - Basics.....</i>	<i>2-4</i>
<i>Lab Hardware Setup</i>	<i>2-7</i>
<i>Code Composer Studio – Step by Step</i>	<i>2-9</i>
Create a project.....	2-10
Setup Build Options.....	2-12
Linker Command File.....	2-13
Download code into DSP	2-15
<i>Lab 1: beginner's project.....</i>	<i>2-23</i>
Objective	2-23
Procedure.....	2-23
Open Files, Create Project File.....	2-23

The Software Flow


The following slide illustrates the software design flow within Code Composer Studio. The basic steps are: edit, compile and link, which are combined into “build”, then debug. If you are familiar with other Integrated Design Environments for the PC such as Microsoft’s Visual Studio, you will easily recognize the typical steps used in a project design. If not, you will have to spend a little more time to practice with the basic tools shown on this slide. The major difference to a PC design toolbox is shown on the right-hand side – the connections to real-time hardware!




You can use Code Composer Studio with a Simulator (running on the Host – PC) or you can connect a real DSP system and test the software on a real “target”. For this tutorial, we will rely on the eZdsp (TMS320F2812eZdsp – Spectrum Digital Inc.) as our “target” and some additional hardware. Here the word “target” means the physical processor we are using, in this case a DSP.

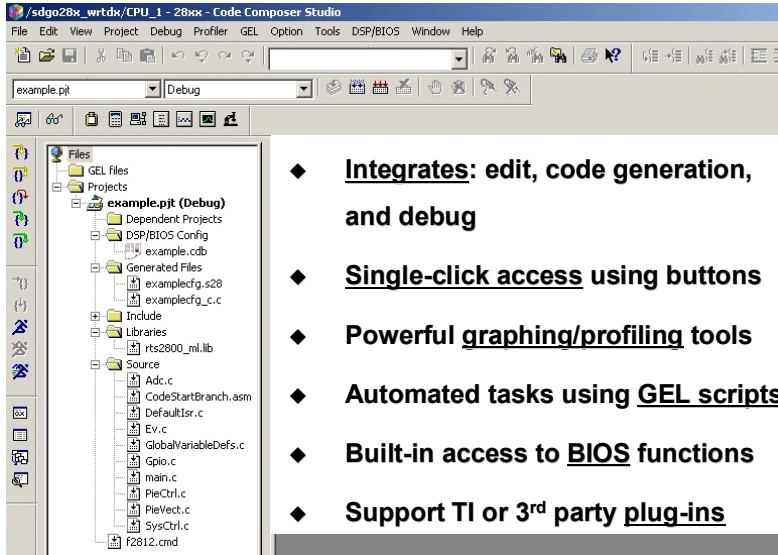
The next slides will show you some basic features of Code Composer Studio and the hardware setup for lab exercises that follow.

Code Composer Studio - Basics




Code Composer Studio® IDE






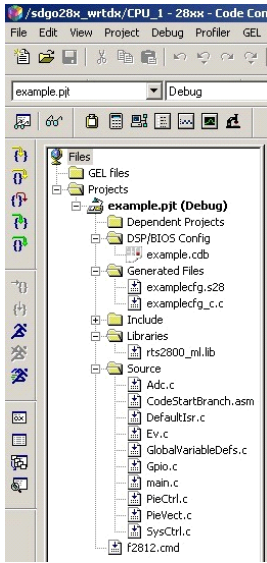
- ◆ **Integrates:** edit, code generation, and debug
- ◆ **Single-click access** using buttons
- ◆ **Powerful graphing/profiling tools**
- ◆ **Automated tasks using GEL scripts**
- ◆ **Built-in access to BIOS functions**
- ◆ **Support TI or 3rd party plug-ins**

2 - 4



The CCS Project





Project (.pjt) files contain:

- ◆ **Source files (by reference)**
 - Source (C, assembly)
 - Libraries
 - DSP/BIOS configuration
 - Linker command files
- ◆ **Project settings:**
 - Build Options (compiler and assembler)
 - Build configurations
 - DSP/BIOS
 - Linker

2 - 5

Build Options GUI - Compiler

◆ GUI has 8 pages of categories for code generation tools

◆ Controls many aspects of the build process, such as:

- Optimization level
- Target device
- Compiler/assembly/link options

2 - 6

Build Options GUI - Linker

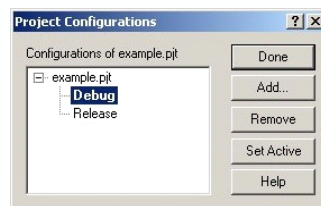
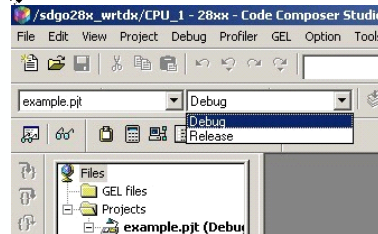
◆ GUI has 2 categories for linking

◆ Specifies various link options

◆ “.\\Debug\\” indicates on subfolder level below project (.pjt) location

2 - 7

Default Build Configurations

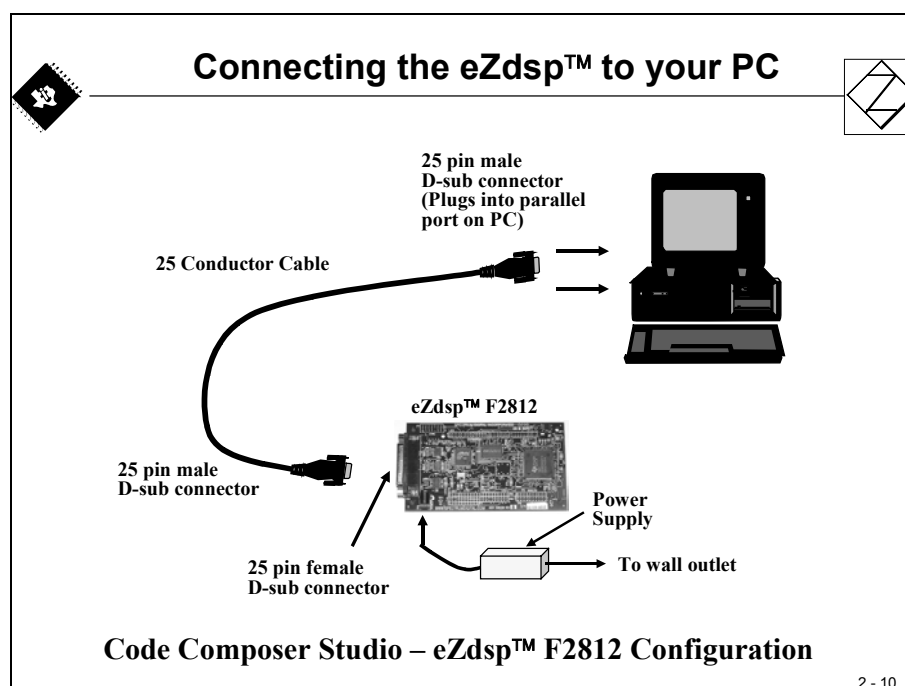
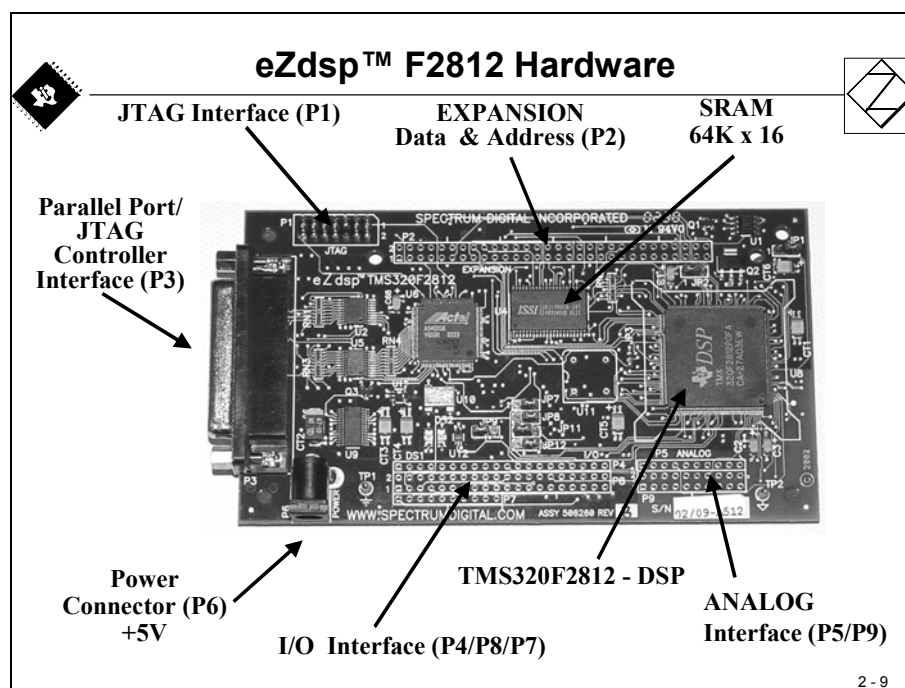


- ◆ For new projects, CCS automatically creates two build configurations:
 - **Debug** (unoptimized)
 - **Release** (optimized)
- ◆ Use the drop-down menu to quickly select the build configuration
- ◆ Add/Remove your own custom build configurations using Project Configurations
- ◆ Edit a configuration:
 1. **Set it active**
 2. **Modify build options**
 3. **Save project**

2 - 8

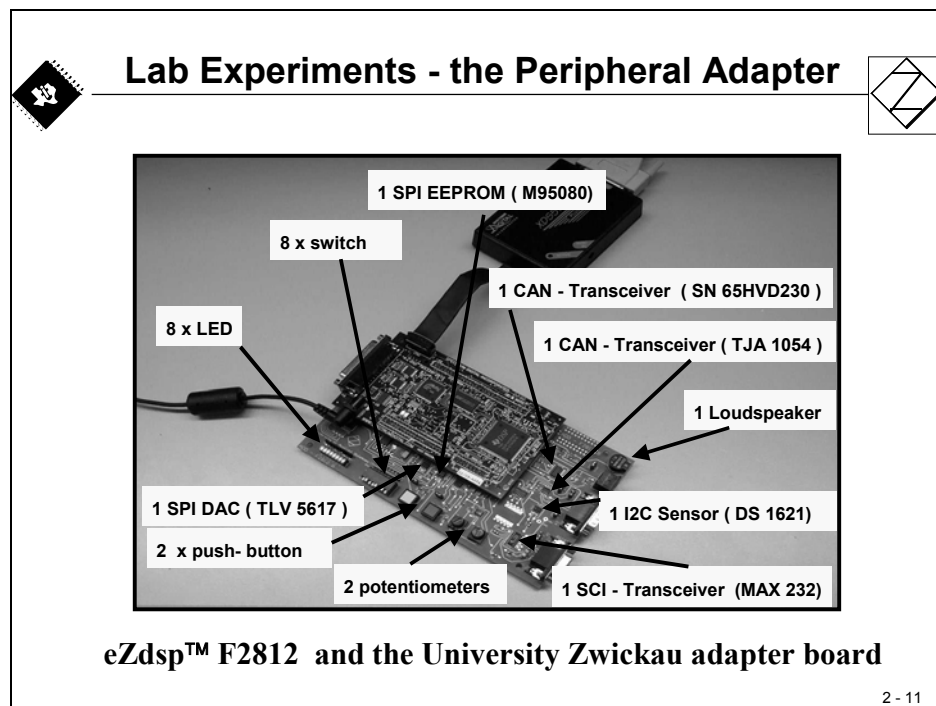
Lab Hardware Setup

The following slides show you the hardware target that is used during our lab exercises in the following chapters. The core is the TMS320F2812 32-bit DSP on board of Spectrum Digital's eZdspF2812. All the internal peripherals are available through connectors. The on board JTAG – emulator connected to the PC using a parallel printer cable.



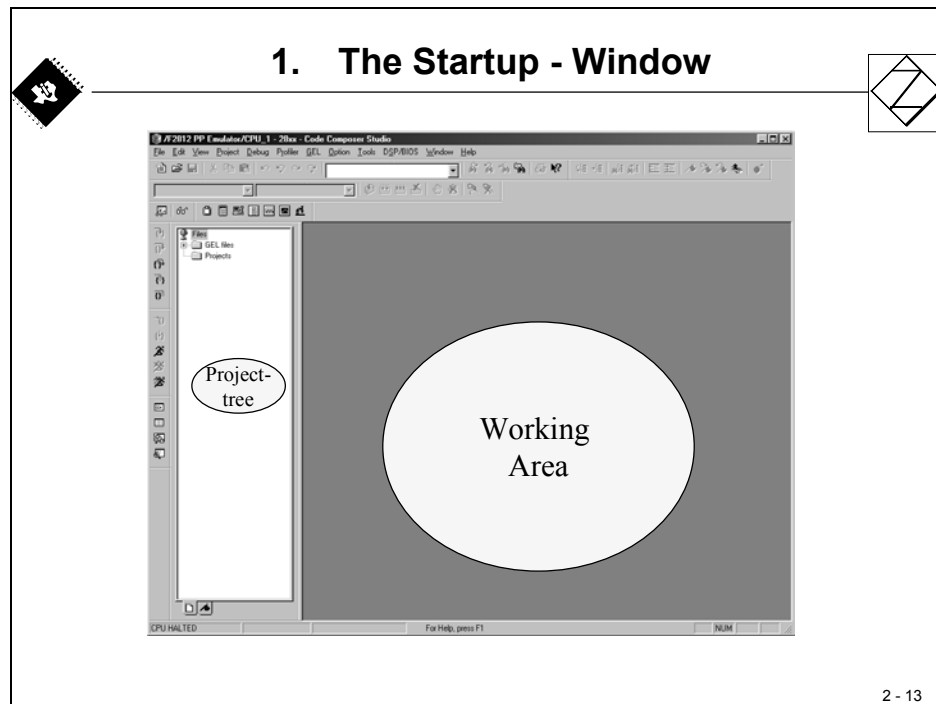
To be able to practice with all the peripheral units of the DSP and some 'real' process hardware, we've added an adapter board, which fits underneath the eZdspF2812. The Zwickau Adapter Board provides:

- 8 LED's for digital output (GPIO B7...B0)
- 8 switches (GPIO B15...B8) and 2 push buttons (GPIO D1, D6) for digital input
- 2 potentiometers (ADCINA0, ADCINB0) for analog input
- 1 loudspeaker for analogue output via PWM - Digisound F/PWC04A
- 1 dual SPI Digital to Analogue Converter (DAC) - Texas Instruments TLV5617A
- 1 SPI EEPROM 1024 x 8 Bit - ST Microelectronics M95080
- 1 CAN Transceiver - Texas Instruments SN 65HVD230 (high speed)
- 1 CAN Transceiver - Philips TJA 1054 (low speed)
- 1 I²C – Temperature Sensor Dallas Semiconductor DS1621
- 1 SCI-A RS 232 Transceiver – Texas Instruments MAX232D
- 2 dual OpAmp's Texas Instruments TLV 2462 – analogue inputs



Code Composer Studio – Step by Step

Now let's start to look a little closer at the most essential parts of Code Composer Studio that we need to develop our first project. Once you or your instructor has installed the tools and the correct driver (Setup CCS2000), you can start Code Composer Studio by simply clicking on its desktop icon. If you get an error message, check the power supply of the target board. If everything goes as expected, you should see a screen, similar to this:



The step-by-step approach for Lab1 will show how to do the following:

- Open Code Composer Studio
- Create a F28x – Project, based on C
- Compile, Link, Download and Debug this test program
- Watch Variables
- Real time run versus single-step test
- Use Breakpoints and Probe Points
- Look into essential parts of the DSP during Debug

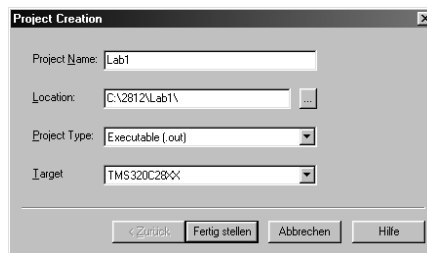
Before we start to go into the procedure for Lab1, let's discuss the steps using some additional slides:

Create a project

2. Create a F28x - project

- **Project → New**

give your project a name : “Lab1”, select a target and a suitable location of your hard disk:



Note : the project file (“Lab1.pjt”) is a simple text file (ASCII) and stores all set-ups and options of the project. This is very useful for a version management.

2 - 14

The first task is to create a project. This is quite similar to most of today’s design environments with one exception: we have to define the correct target, in our case “TMS320C28xx”. The project itself generates a subdirectory with the same name as the project. Ask your instructor for the correct location to store your project.

2. Create a F28x - project (cont.)

Write your C-Source Code :

→File →New →Source File

```
unsigned int k;
void main (void)
{
    unsigned int i;
    while(1)
    {
        for (i=0;i<100;i++)
            k=i*i;
    }
}
```

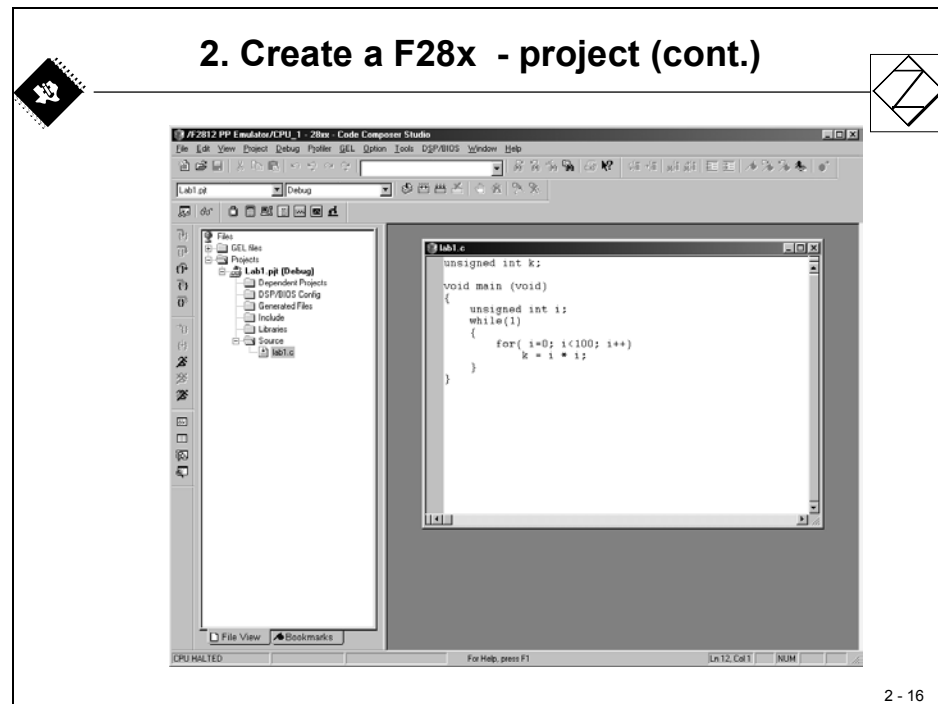
→File →Save as : “lab1.c”

(use the same path as for the project, e.g. C:\C28x\Lab1)

2 - 15

Next, write the source code for your first application. The program from the slide above is one of the simplest tasks for a processor. It consists of an endless loop, which counts variable *i* from 0 to 99, calculates the current product of *i* * *i* and stores it temporarily in *k*. It seems to be an affront to bother a sophisticated Digital Signal Processor with such a simple task! Anyway, we want to gain hands-on experience of this DSP and our simple program is an easy way for us to evaluate the basic commands of Code Composer Studio.

Your screen should now look like this:



Your source code file is now stored on the PC's hard disk but it is not yet part of the project. Why does Code Composer Studio not add this file to our project automatically? Answer: If this were an automatic procedure, then all the files that we touch during the design phase of the project would be added to the project, whether we wanted or not. Imagine you open another code file, just to look for a portion of code that you used in the past; this file would become part of your project.

To add a file to your project, you will have to use an explicit procedure. This is not only valid for source code files, but also for all other files that we will need to generate the DSP's machine code. The following slide explains the next steps for the lab exercise:

Setup Build Options

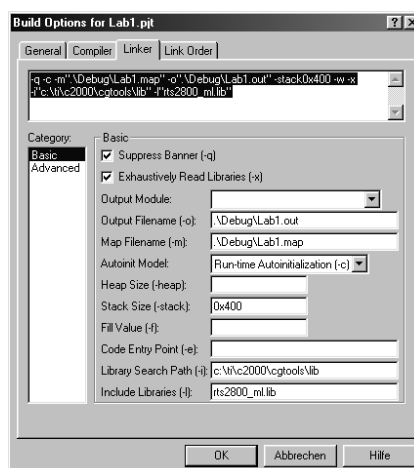
2. Create a F28x - project (cont.)

- ◆ **Add your code file to the project :**
 - ➔ Project ➔ Add files to project : “lab1.c”
- ◆ **Compile your source code :**
 - ➔ Project ➔ Compile File
 - active window will be compiled
 - in the event of syntax errors : modify your source code as needed
- ◆ **Add the C-runtime-library to your project :**
 - ➔ Project ➔ Build Options ➔ Linker ➔ Library Search Path :
c:\ti\c2000\cgtools\lib
 - ➔ ➔ Project ➔ Build Options ➔ Linker ➔ Include Libraries :
rts2800_ml.lib
- ◆ **Add the stack - size of 0x400**
 - ➔ Project ➔ Build Options ➔ Linker ➔ Stack Size : 0x400

2 - 17

When you've done everything correctly, the build options window should look like this:

2. Create a F28x - project (cont.)



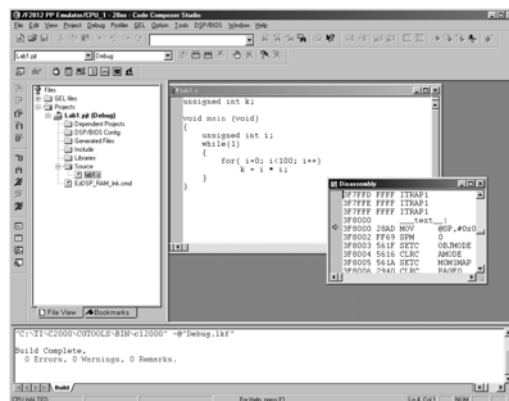
Close the build-window by 'OK'

2 - 18

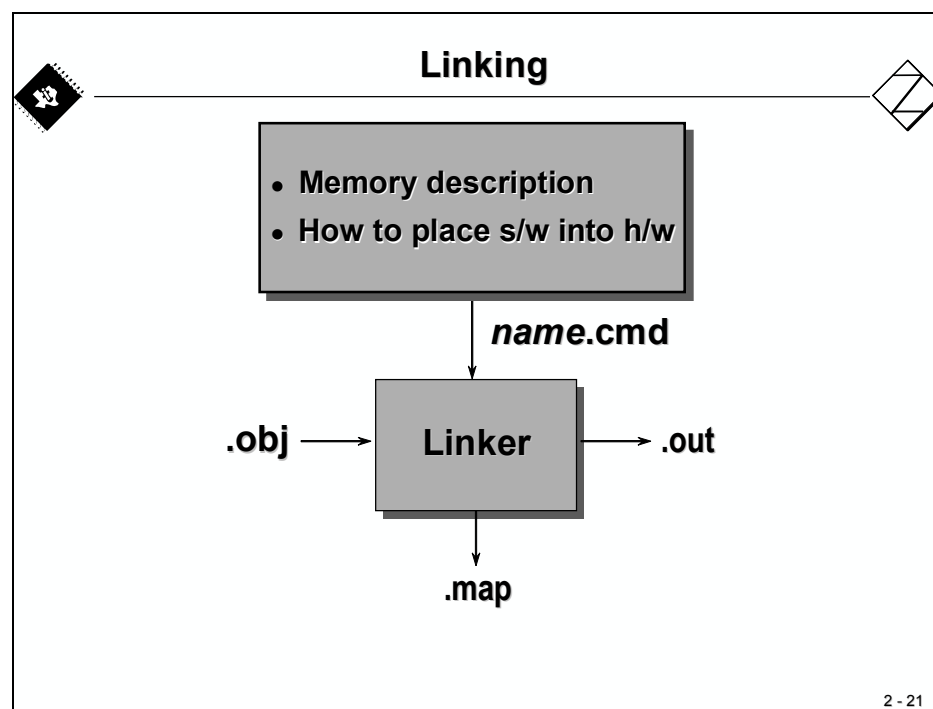
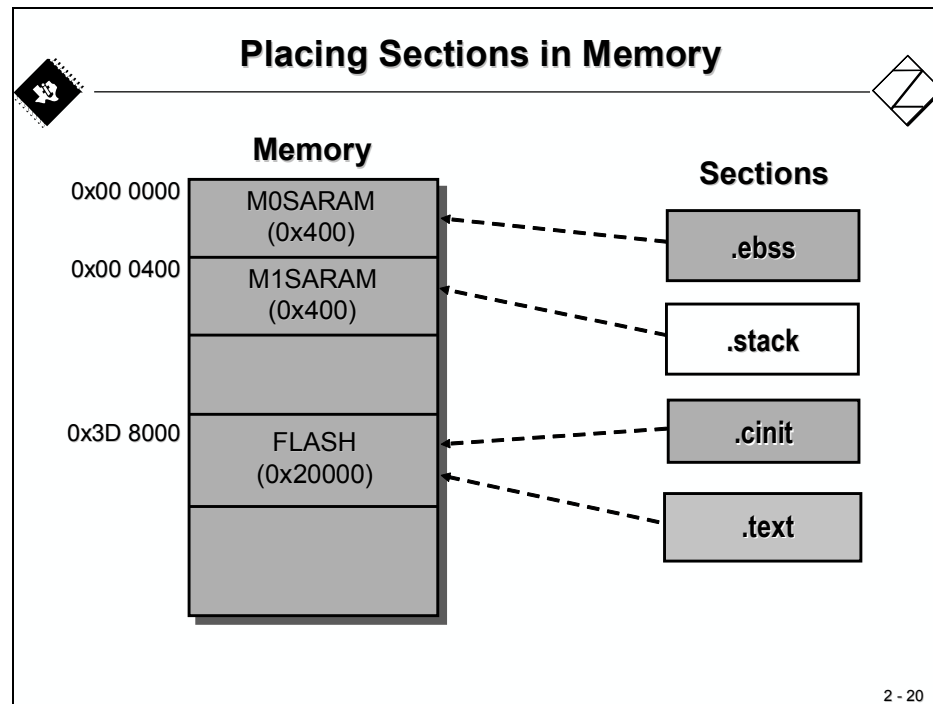
Linker Command File

Now we have to control the “Linker”. The “Linker” puts together the various building blocks we need for a system. This is done with the help of a so-called “Linker Command File”. Essentially, this file is used to connect physical parts of the DSP’s memory with logical sections created by our software. We will discuss this linker procedure later in detail. For now, we will use a predefined Linker Command File “F2812_EzDSP_RAM_lnk.cmd”. This file has been designed by Texas Instruments and is part of the Code Composer Studio Software package.

-



2 - 19



The procedure of linking connects one or more object files (*.obj) into an output file (*.out). This output file contains not only the absolute machine code for the DSP, but also information used to debug, to flash the DSP and for more JTAG based tasks. Do NEVER take the length of this output file as the length of your code! To extract the usage of resources we always use the MAP file (*.map).

Linker Command File

```

MEMORY
{
    PAGE 0:          /* Program Space */
    FLASH:           org = 0x3D8000, len = 0x20000
    PAGE 1:          /* Data Space */
    MOSARAM:         org = 0x000000, len = 0x400
    M1SARAM:         org = 0x000400, len = 0x400
}

SECTIONS
{
    .text:           >      FLASH           PAGE 0
    .ebss:           >      MOSARAM          PAGE 1
    .cinit:          >      FLASH           PAGE 0
    .stack:          >      M1SARAM          PAGE 1
}

```

2 - 22

Download code into DSP

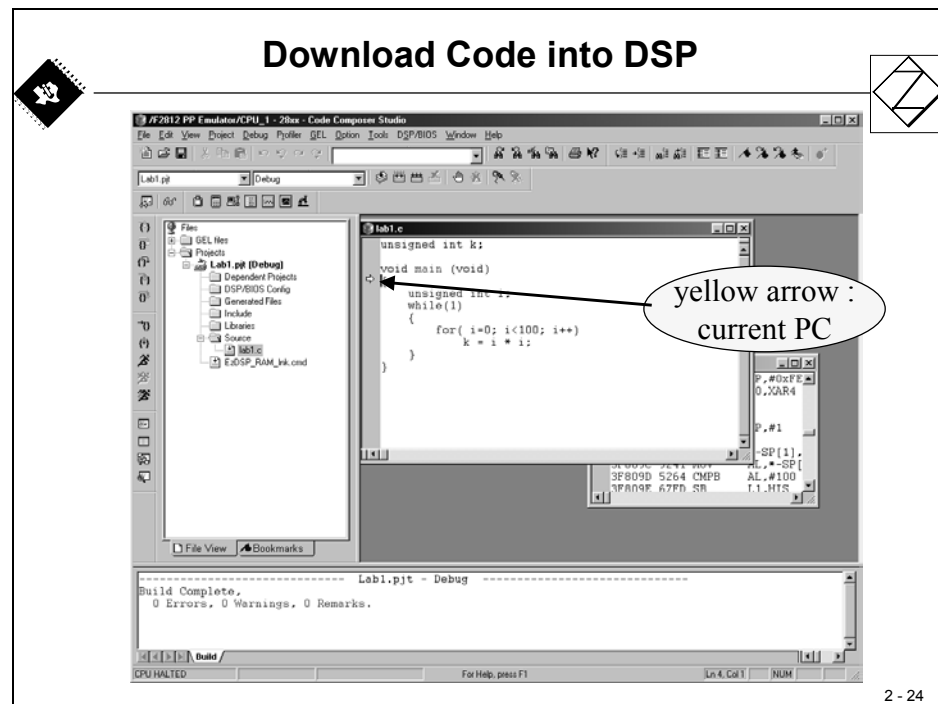
Now it is time to download the code into the F2812. We have two options: manual or automatic download after a successful build.

Download Code into DSP

- ◆ **Load the binary code into the DSP :**
 - File → Load Program → Debug\Lab1.out
- Note: a new binary code can be downloaded automatically into the target. This is done by → Option → Customize → Program Load Options → Load Program after Build. This setup will be stored for permanently.*
- ◆ **Run the program until label “main”**
 - Debug → Go main

2 - 23

After → **Debug** → **Go main**, a yellow arrow shows the current position of the Program Counter (PC). This register points always the next instruction to be executed.



2 - 24

When we start to test our first program, there is no hardware activity to be seen. Why not? Well, our first program does not use any peripheral units of the DSP. Go through the steps, shown on the next slide.

3. Debug your code !

- ◆ **Perform a real time run :**
 - Debug → Run (F5)
 - Note 1:** *the bottom left corner will be marked as : “DSP Running”. You’ll see no activity on the peripherals of the Adapter Board because our first example program does not use any of them!*
 - Note 2:** *the yellow arrow is no longer visible – that’s another sign of a real time run.*
- ◆ **Stop the real time run :**
 - Debug → Halt
- ◆ **Reset the DSP :**
 - Debug → Reset CPU
 - Debug → Restart
- ◆ **Run again to main :**
 - Debug → Go Main

2 - 25

To watch the program's variables, we can use a dedicated window called the "Watch Window". This is probably the most used window during the test phase of a software project.

4. Watch your variables

◆ Open the Watch Window :

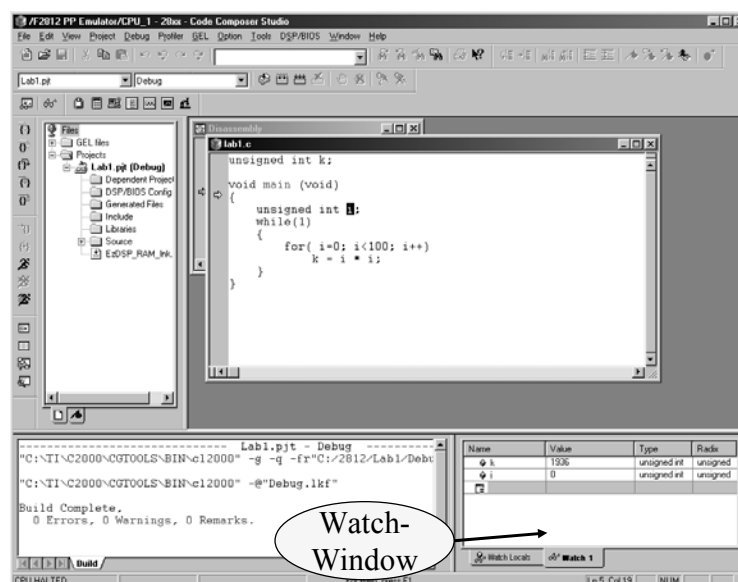
➔ View ➔ Watch Window

- The variable 'i' is already visible inside the "Watch Locals"-window .
- To see also the global 'k' we need to add this variable manually. This can be done inside window 'Watch 1'. In the column 'name' we just enter 'k' and in the second line 'i'.
- *Note : another convenient way is to mark the variables inside the source code with the right mouse button and then select "Add to watch window"*

- ◆ *note : with the column 'radix' one can adjust the data format between decimal, hexadecimal, binary etc.*


2 - 26

4. Watch your variables




2 - 27

Another useful part of a debug session is the ability to debug the code in portions and to run the program for a few instructions. This can be done using a group of single-step commands:




5. Perform a Single Step Debug




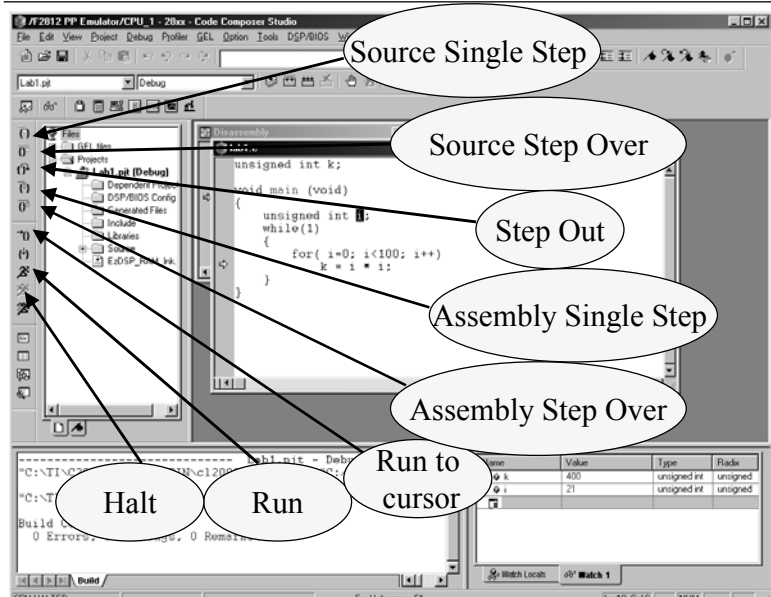
- ◆ **Perform a single step through the program :**
 ➔ Debug ➔ Step Into (or F8)
- ◆ **Watch the current PC (yellow arrow) and the numerical values of i and k in Watch Window while you single step through the code !**
- ◆ **There are more debug - commands available, see next slide**

2 - 28



5. Perform a Single Step Debug





2 - 29

When you'd like to run the code through a portion of your program that you have tested before, a Breakpoint is very useful. After the 'run' command, the JTAG debugger stops automatically when it hits a line that is marked with a breakpoint.

6. Add a breakpoint

◆ Set a breakpoint:

- Place the Cursor in Lab1.c on line : `k = i * i;`
- Click right mouse and select 'Toggle Breakpoint'
- the line is marked with a red dot (= active breakpoint)

Note : most Code Composer Studio Commands are also available through buttons or trough Command -Keys (see manual, or help)

◆ Reset the Program:

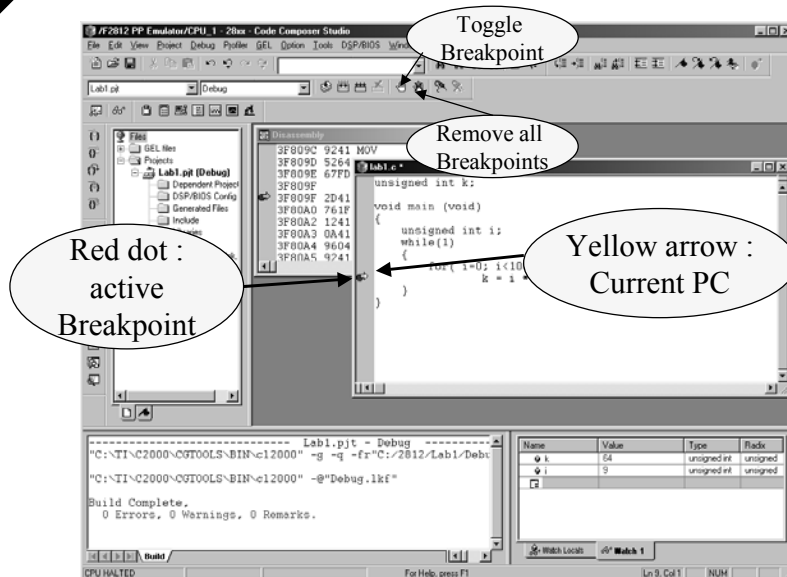
- Debug → Reset CPU
- Debug → Restart

◆ Perform a real time run:

- Debug → Run (or F5)
 - DSP stops when reaching an active breakpoint
 - repeat 'Run' and watch your variables
 - remove the breakpoint (Toggle again) when you're done.


2 - 30

6. Add a breakpoint (cont.)




2 - 31

A slightly different tool to stop the execution is a 'Probe Point'. While the Break Point forces the DSP to halt permanently, the Probe Point is only a temporary stop point. During the stop status, the DSP and the JTAG-emulator exchange information. At the end, the DSP resumes the execution of the code.



7. Set a Probe Point



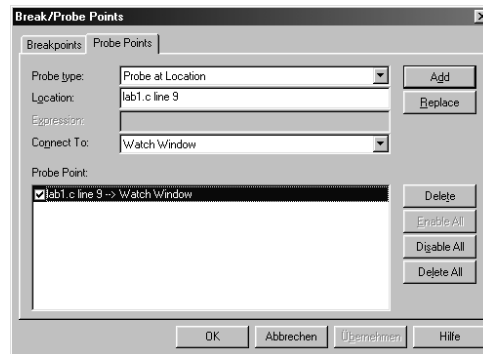
- ◆ **Causes an update of a particular window at a specific point in your program.**

When a window is created it is updated at every hit of a breakpoint. However, you can change this so the window is updated only when the program reaches the connected Probe Point. When the window is updated, execution of the program is continued.
- ◆ **To set a Probe - Point :**
 - Click right mouse on the line 'k = i*i;' in the program first.c
 - select : 'Toggle Probe Point ' (indicated by a blue dot)
 - select ➔ Debug ➔ Probe Points...
 - In the Probe Point Window click on the line 'first.c line 13 -> no Connection'
 - in the 'Connect to' - selector select 'Watch Window'
 - exit this dialog with the 'Replace' and 'OK' - Button
- ◆ **Run the program and verify that the watch window is updated continuously.**

2 - 32

NOTE: There is a more advanced method to interact with the DSP in real time, called 'Real Time Debug'. We will skip this option for the time being and use this feature during later chapters.

7. Set a Probe Point (cont.)



2 - 33

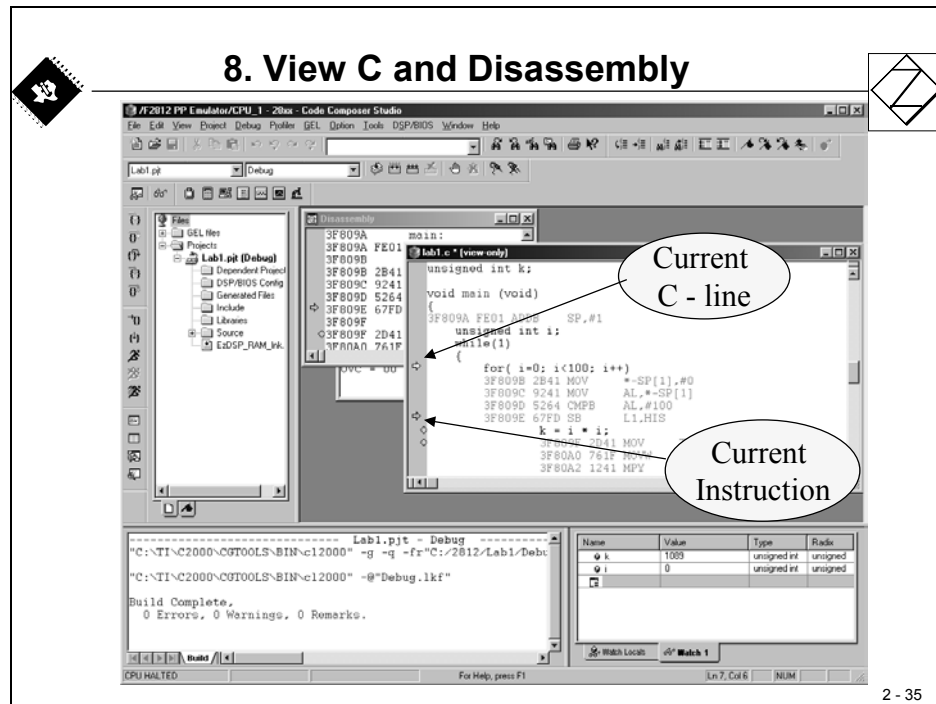
When you are more familiar with the F2812 and with the tools, you might want to verify the efficiency of the C compiler or to optimize your code at the Assembler Language level.

8. Other View Commands (cont.)

- ◆ **To view both the Assembler code and the C Source Code :**
 - click right mouse inside “Lab1.c” and select “Mixed Mode”
 - The Assembler Instruction Code generated by the Compiler is added and printed in grey colour
- ◆ **Single Step (‘Assembly Step Into’) is now possible on instruction level:**
 - ➔ Debug ➔ Reset DSP
 - ➔ Debug ➔ Restart
 - ➔ Debug ➔ Go Main
 - ➔ Debug ➔ Step Into (F8)
 - You’ll see two arrows , a yellow one on C-lines and a green one for assembler instruction-lines

2 - 34

8. View C and Disassembly



The General Extension Language (GEL) is a high-level script language. Based on a *.gel – file one can expand the features of Code Composer Studio or perform recurrent steps automatically.

9. GEL - General Extension Language

- ◆ language similar to C
- ◆ lets you create functions to extend Code Composer's features
- ◆ to create GEL functions use the GEL grammar
- ◆ load GEL-files into Code Composer
- ◆ With GEL, you can:
 - ◆ access actual/simulated target memory locations
 - ◆ add options to Code Composer's GEL menu
- ◆ GEL is useful for automated testing and user workspace adjustment .
- ◆ GEL - files are ASCII with extension *.gel

Lab 1: beginner's project

Objective

The objective of this lab is to practice and verify the basics of the Code Composer Studio Integrated Design Environment.

Procedure

Open Files, Create Project File

1. Using Code Composer Studio, create a new project, called **Lab1.pjt** in E:\C281x\Labs (or another working directory used during your class, ask your instructor for specific location!)
2. Write a new source code file by clicking: File → New → Source File. A new window in the workspace area will open. Type in this window the following few lines:

```
unsigned int k;
void main (void)
{
    unsigned int i;
    while(1)
    {
        for (i = 0; i < 100; i++)
            k = i * i;
    }
}
```

Save this file by clicking File → Save as and type in: **Lab1.c**

3. Add the Source Code Files: **Lab1.c** and the provided linker command file: **\cmd\F2812_EzDSP_RAM_lnk.cmd** (it is in E:\2812\cmd\) to your project by clicking: Project → Add Files to project
4. Add the C-runtime library to your project by clicking: Project → Build Options → Linker → Library Search Path: 'c:\ti\c2000\cgtools\lib'. Then Add the library by clicking: Project → Build Options → Linker → Include Libraries: '**rts2800_ml.lib**'

5. Verify that in Project → Build Options → Linker the Autoinit-Field contains: 'Run-time-Autoinitialisation [-c]
6. Set the stack size to 0x400: Project → Build Options → Linker → Stack Size
7. 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. Debug as necessary. To open up more space, close any open files or windows that you do not need at this time.
9. Load the output file onto the eZdsp. Click: File → Load Program and choose the output file you generated. Note: The binary output file has the same name as the project with the extension .out. The file is stored under the "Debug" subfolder.

Note: Code Composer can automatically load the output file after a successful build. To do this by default, click on the menu bar: Option → Customize → Program Load Options and select: "Load Program After Build", then click OK.

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. Verify that in the working area the source code "Lab1.c" is highlighted and that the yellow arrow for the current Program Counter is placed on the line 'void main (void)'.
12. Perform a real time run by clicking: Debug → Run
13. Verify the note at the bottom left corner of the screen: "DSP Running" to mark a real time run. Because we are doing nothing with peripherals so far, there is no other visible activity.
14. Halt the program by clicking: Debug → Halt, reset the DSP (Debug → Reset CPU, followed by → Debug → Restart) and go again until main (Debug → Go main)
15. Open the Watch Window to watch your variables. Click: View → Watch Window. Look into the window "Watch locals". Once you are in main, you

should see variable `i`. Variable `k` is a global one. To see this variable we have to add it to the window 'Watch 1'. Just enter the name of variable '`k`' into the first column 'Name'. Use line 2 to enter variable `i` as well. Exercise also with the 'Radix' column.

16. Perform a single-step through your program by clicking: Debug → Step Into (or use function Key F8). Repeat F8 and watch your variables.
17. Place a Breakpoint in the Lab1.c – window at line "`k = i * i;`". Do this by placing the cursor on this line, click right mouse and select: "Toggle Breakpoint". The line is marked with a red dot to show an active breakpoint. Perform a real-time run by Debug → Run (or F5). The program will stop execution when it reaches the active breakpoint. Remove the breakpoint after this step (click right mouse and "Toggle Breakpoint").
18. Set a Probe Point. Click right mouse on the line "`k=i*i;`". Select "Toggle Probe Point". A blue dot in front of the line indicates an active Probe-Point. From the menu-bar select "Debug → Probe Points...". In the dialog window, click on the line "Lab1.c line 13 → No Connection". Change the "connect to"-selector to "Watch Window", click on 'Replace' and 'OK'. Run the program again (F5). You will see that the probe point updates the watch window each time the program passes the probe point.
19. Have a look into the DSP-Registers: View → Registers → CPU Register and View → Registers → Status Register. Right mouse click inside these windows and select "Float in Main Window". Double click on the line ACC in the CPU-Register and select 'Edit Register'. Modify the value inside the Accumulator.
20. You might want to use the workspace environment in further sessions. For this purpose, it is useful to store the current workspace. To do so, click: File → Workspace → Save Workspace and save it as "Lab1.wks"
21. Delete the active probe by clicking on the button "Remove all Probe Points", close the project by Clicking Project → Close Project and close all open windows that you do not need any further.

End of Exercise Lab1

This page was intentionally left blank.

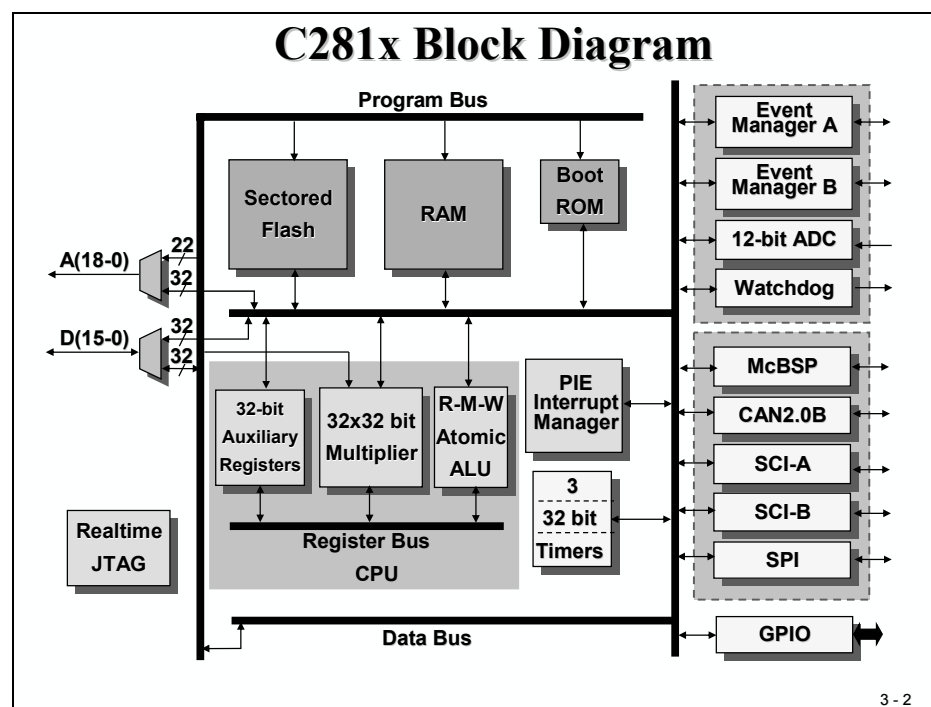
Introduction

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:

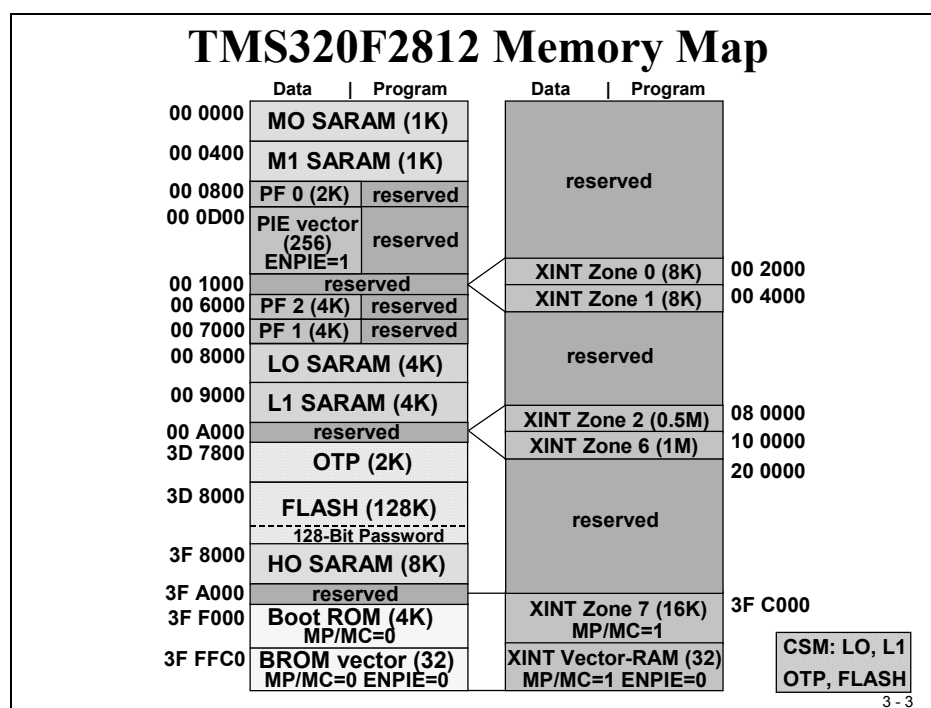


Module Topics

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

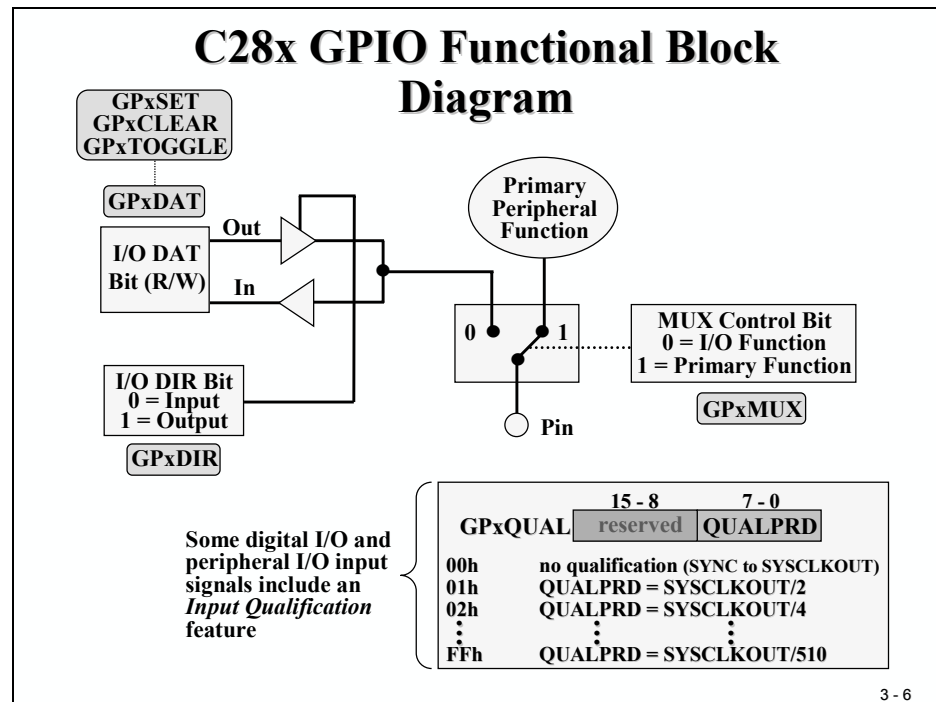
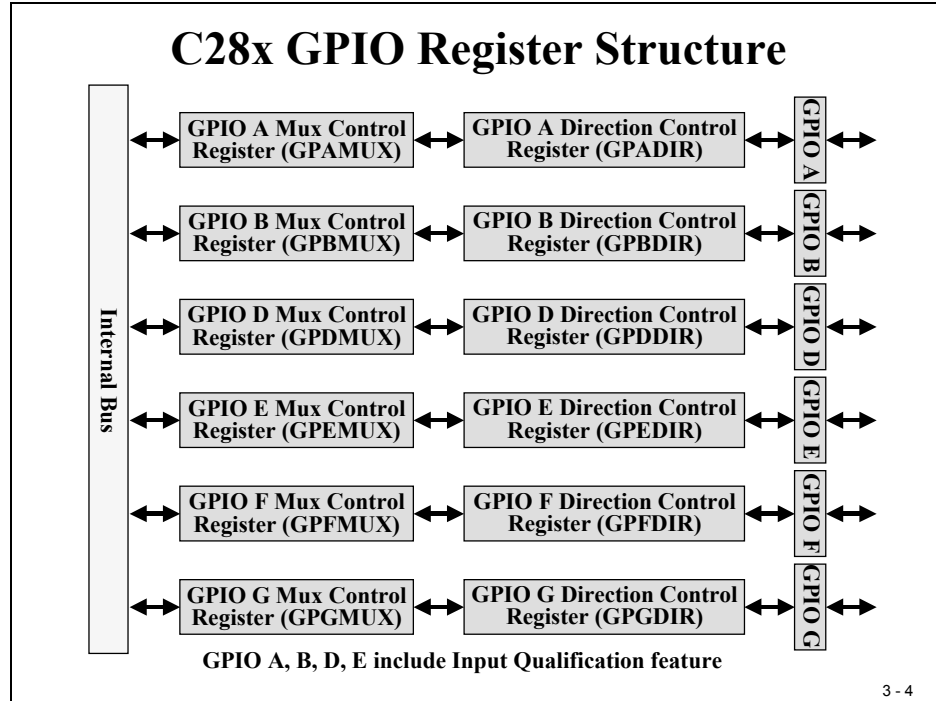
GPIO A, B, D, E include Input Qualification feature

3 - 5

The term "Input Qualification feature" refers to an additional option for digital input signals at Ports A, B, D and E. 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.

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.



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

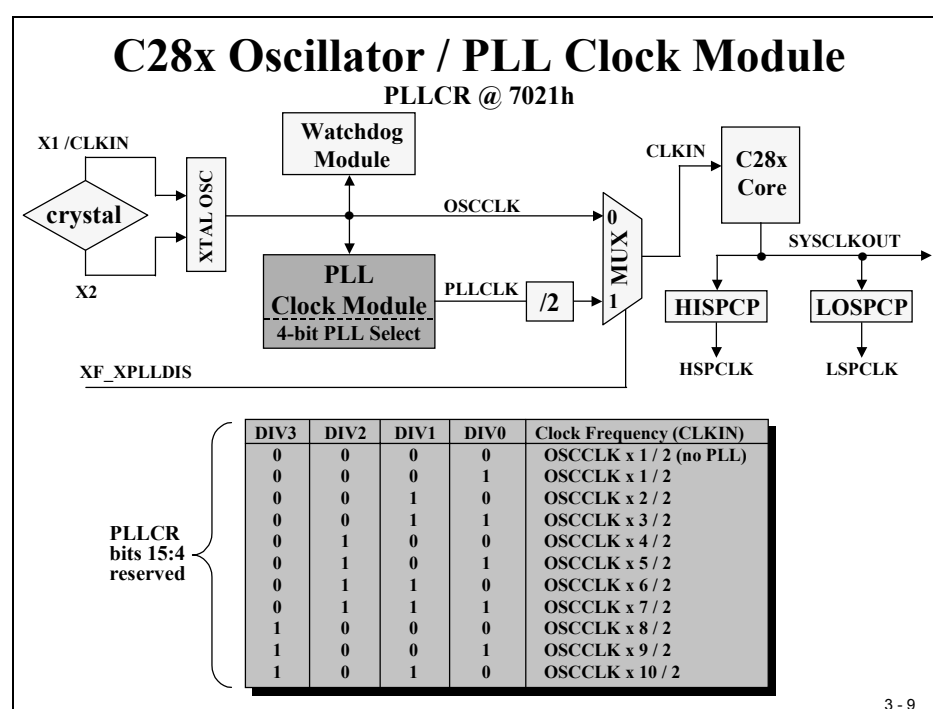
C28x GPIO Data Registers

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).



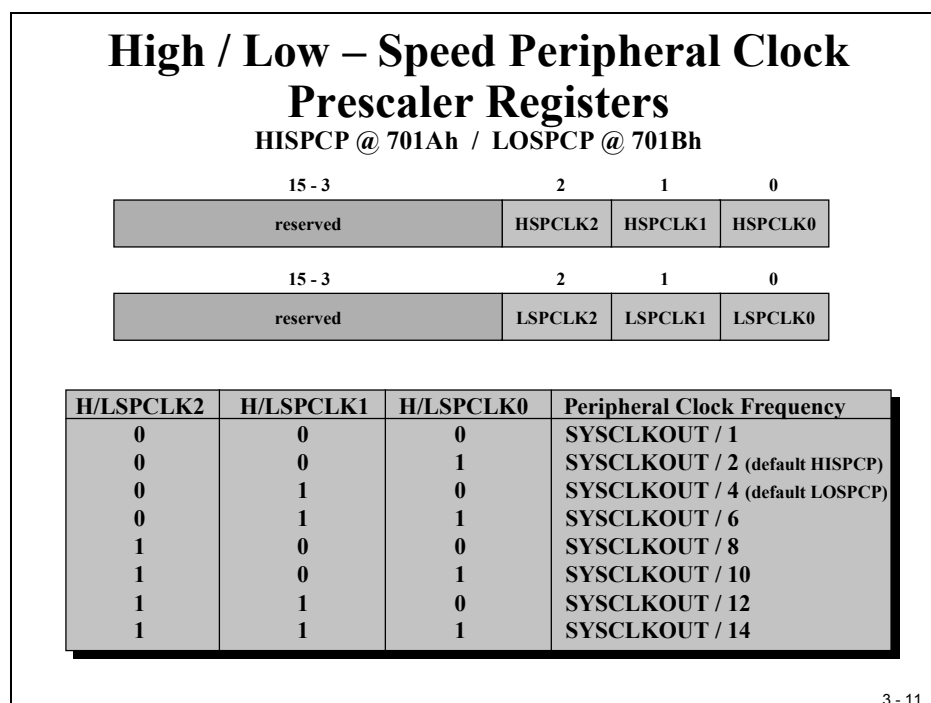
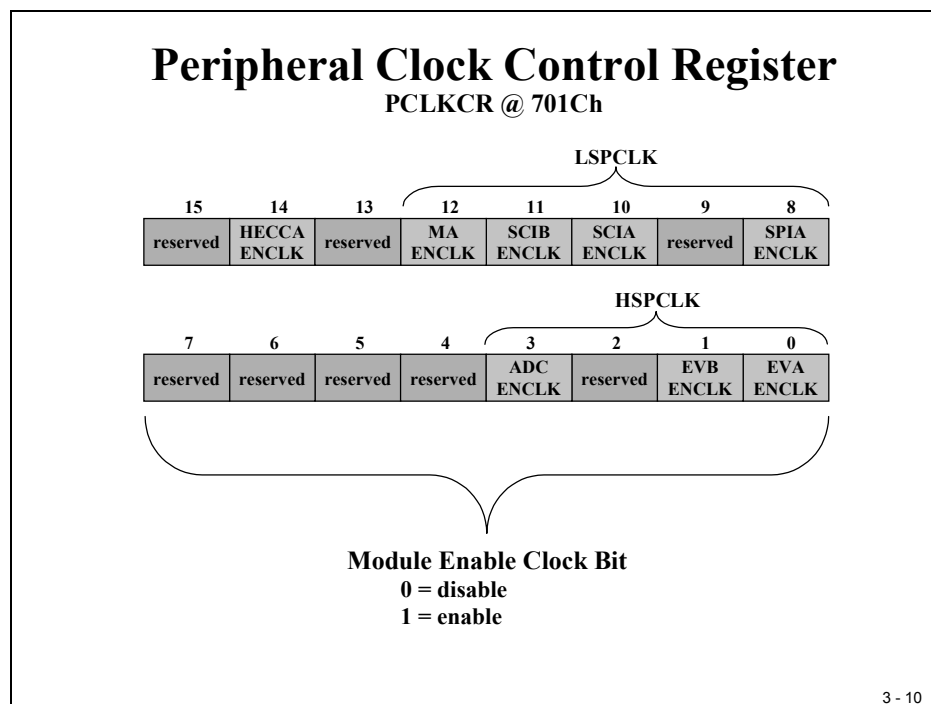
High-speed Clock Pre-scaler (HSPCP) and Low speed Clock Pre-scaler (LOSPCP) are used as additional clock dividers. The outputs of the two pre-scalers are used as the clock source for the peripheral units. We can set up the two pre-scalers individually to our needs.

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.

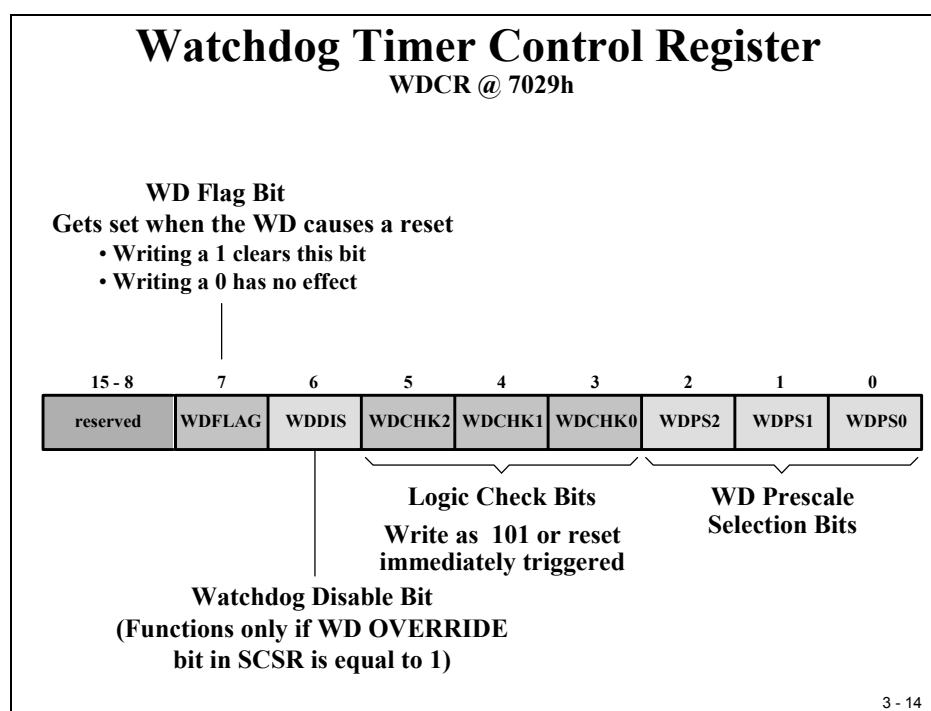
To use a peripheral unit, we have to enable its clock distribution by individual bit fields of register PCLKCR. Digital I/O does not have a clock enable feature.



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

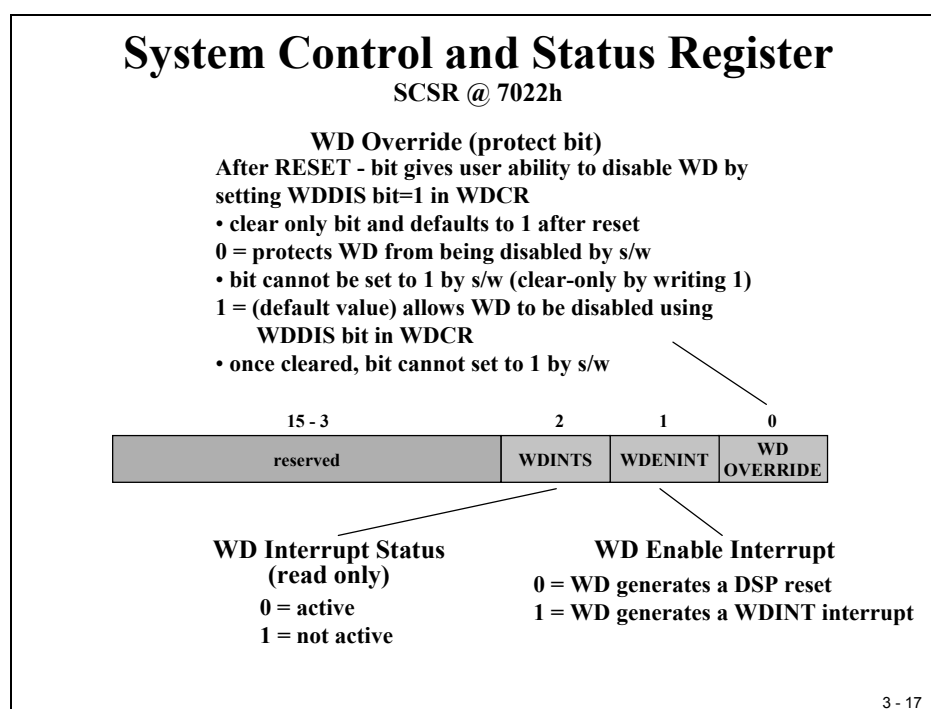
3 - 16

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

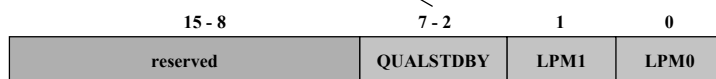
3 - 18

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

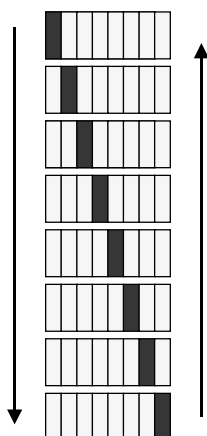
3 - 21

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 read 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)

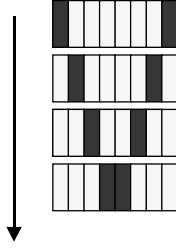
Objective

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.

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

Interrupt System	4-1
<i>Introduction</i>	<i>4-1</i>
<i>Module Topics.....</i>	<i>4-2</i>
<i>C28x Core Interrupt Lines</i>	<i>4-3</i>
<i>The C28x RESET.....</i>	<i>4-4</i>
<i>Reset Bootloader.....</i>	<i>4-5</i>
<i>Interrupt Sources</i>	<i>4-7</i>
<i>Maskable Interrupt Processing.....</i>	<i>4-8</i>
<i>Peripheral Interrupt Expansion</i>	<i>4-10</i>
<i>C28x CPU Timers</i>	<i>4-13</i>
<i>Summary:</i>	<i>4-15</i>
<i>Lab 4: CPU Timer 0 Interrupt & 8 LED's.....</i>	<i>4-16</i>

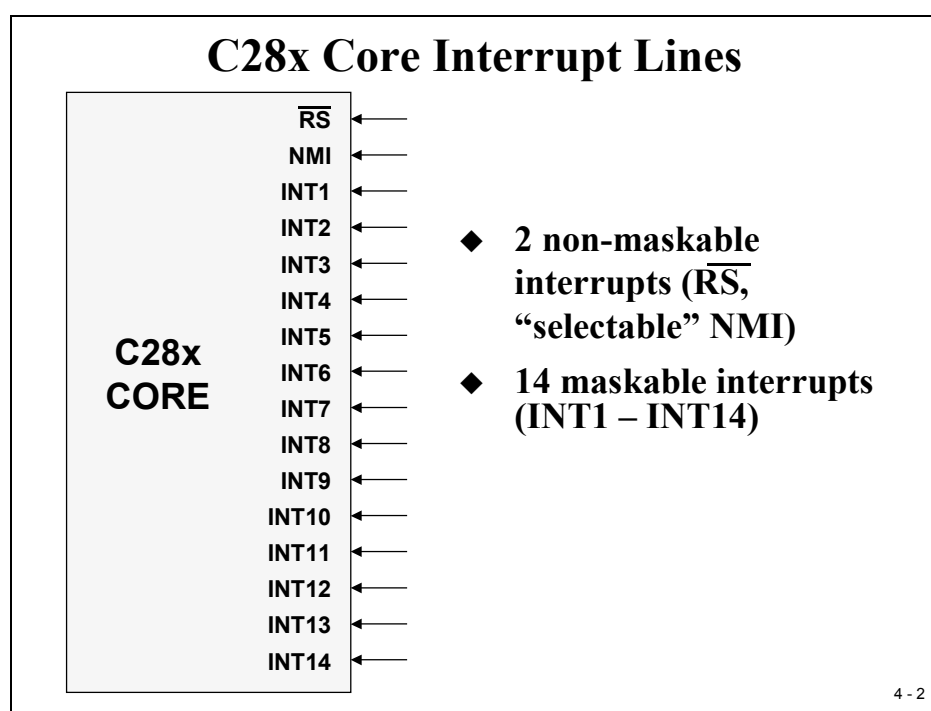
C28x Core Interrupt Lines

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.

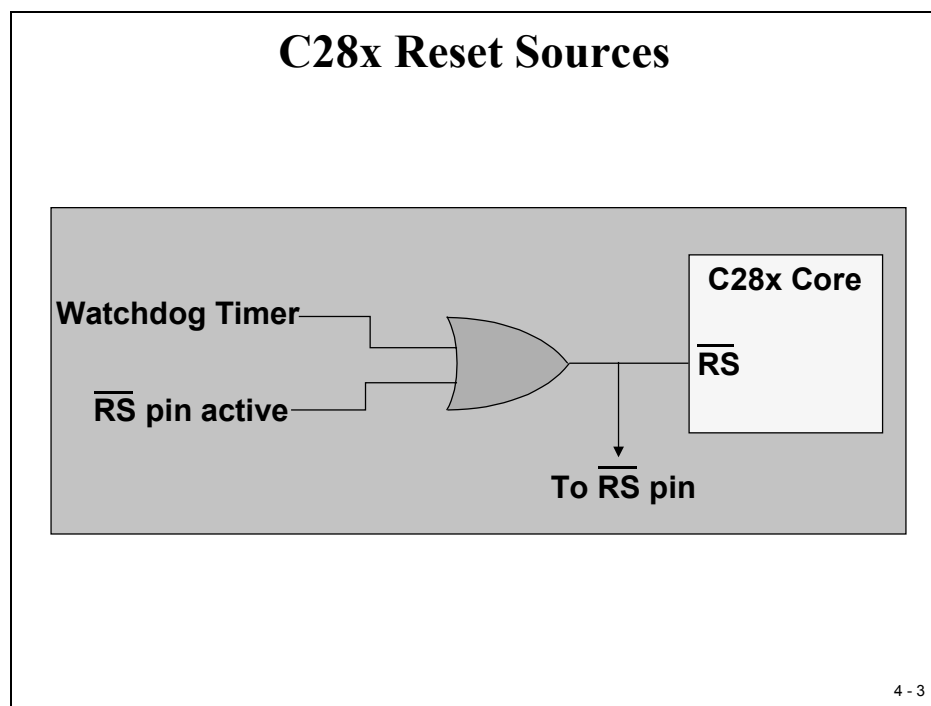


All 16 lines are connected to a table of ‘interrupt vectors’, which consists of 32 bit memory locations per interrupt. It is the responsibility of the programmer to fill this table with the start addresses of dedicated interrupt service routines.

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!

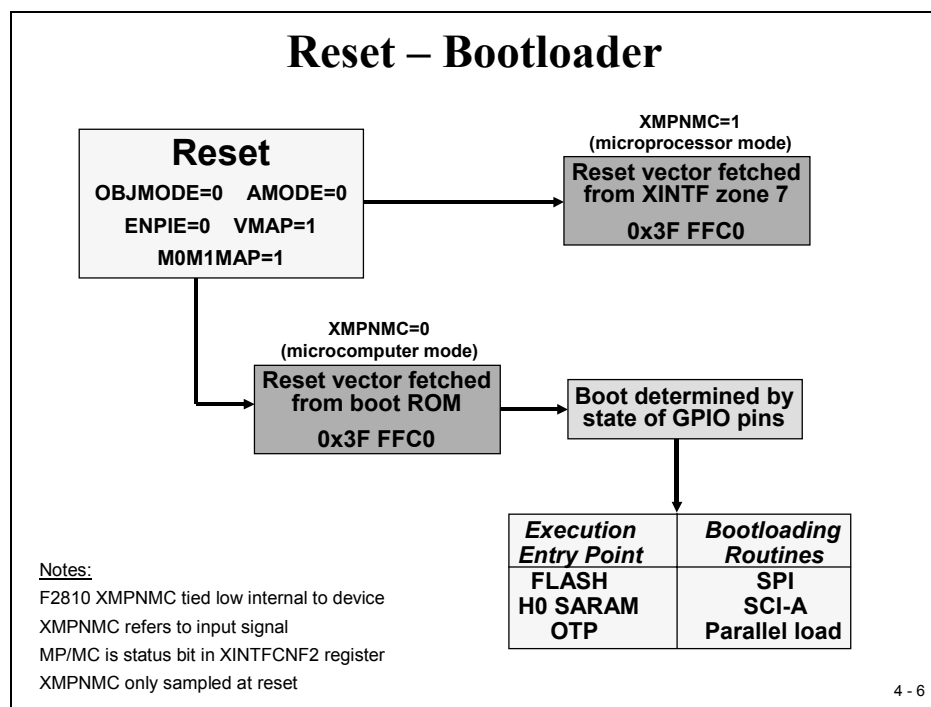


Reset will force the DSP not only to start from address 0x3F FFC0, it will also clear all internal operation registers, reset a group of CPU-Flags to initial states and disable all 16 interrupt lines. We will not go into details about all the flags and registers for now, please refer to the data sheet of the F28x.

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).



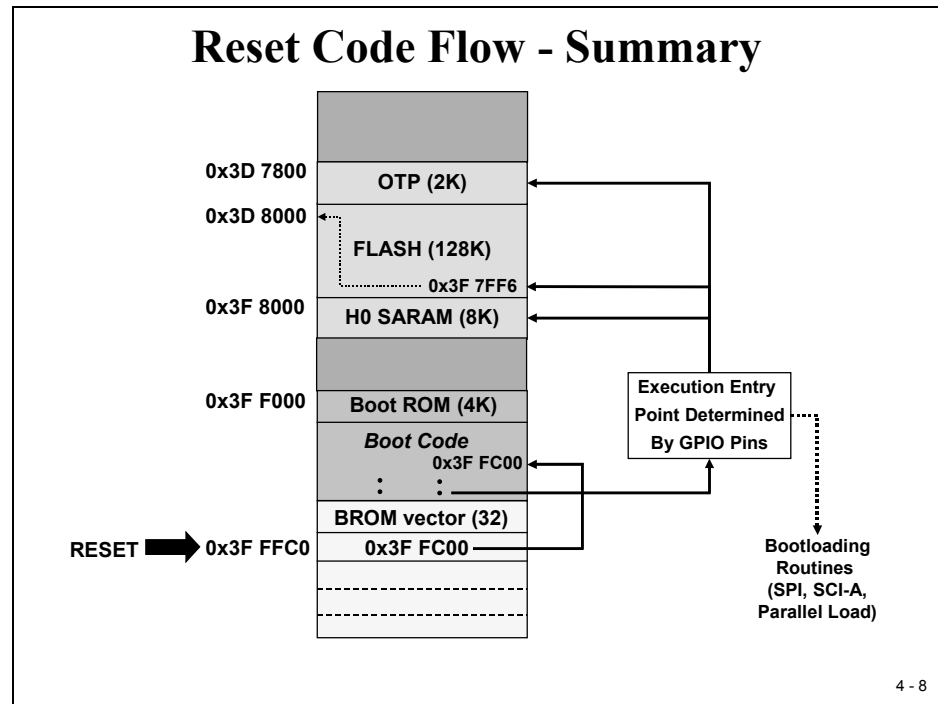
Bootloader Options

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

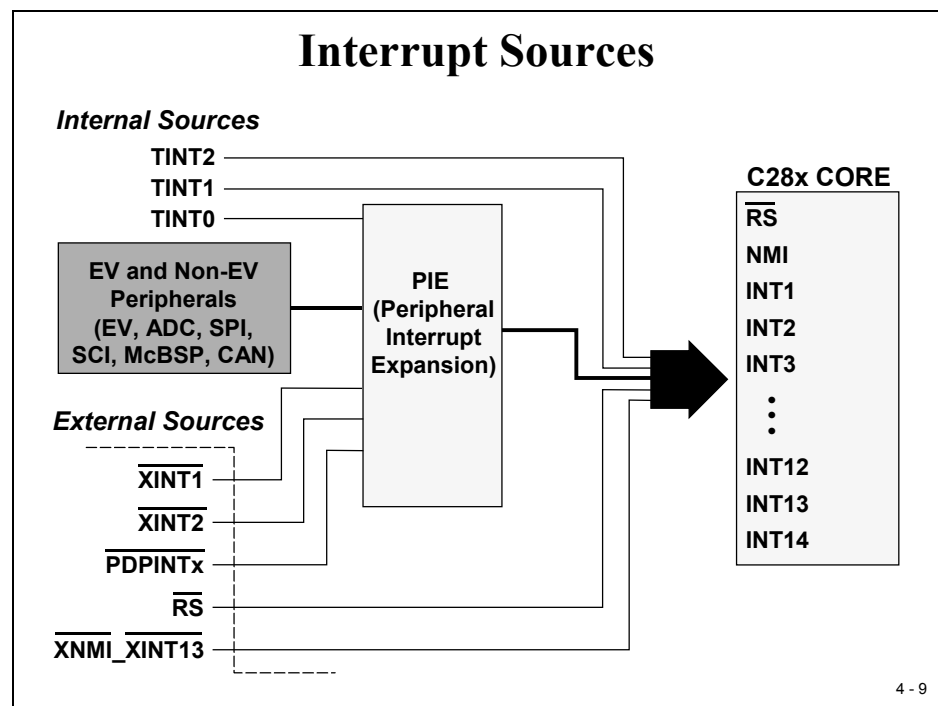
The bootloader options via serial interface (SPI / SCI) or parallel port are usually used to download the executable code from an external host or to field update the contents of the flash memory. We will not use these features during this tutorial.

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?



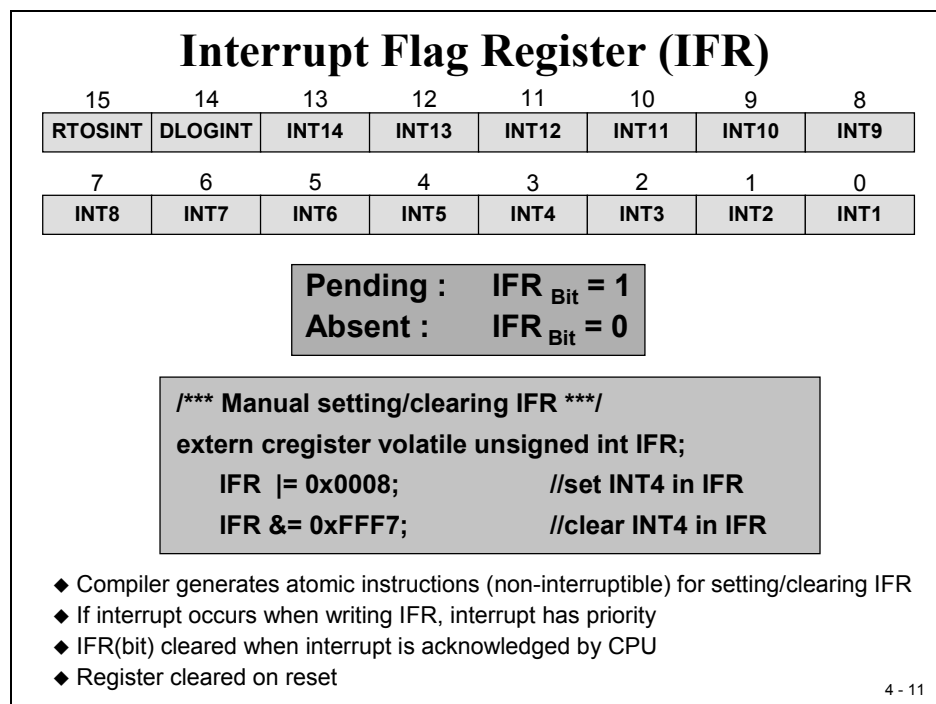
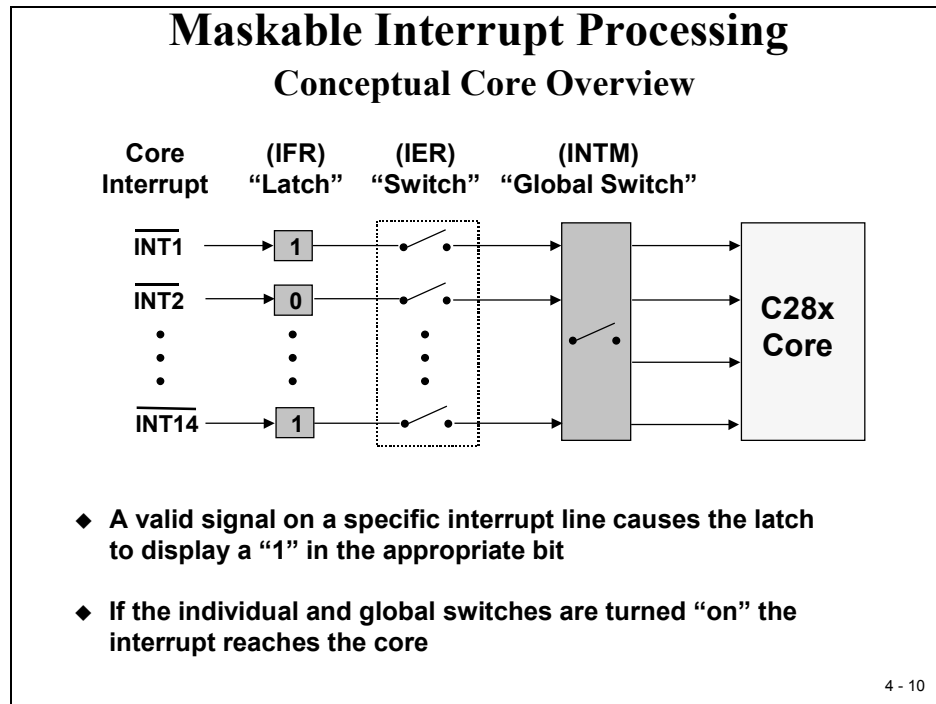
The answer is the PIE (Peripheral Interrupt Expansion)-unit.

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.



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 register volatile unsigned int IER;
IER |= 0x0008;           //enable INT4 in IER
IER &= 0xFFFF;          //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:

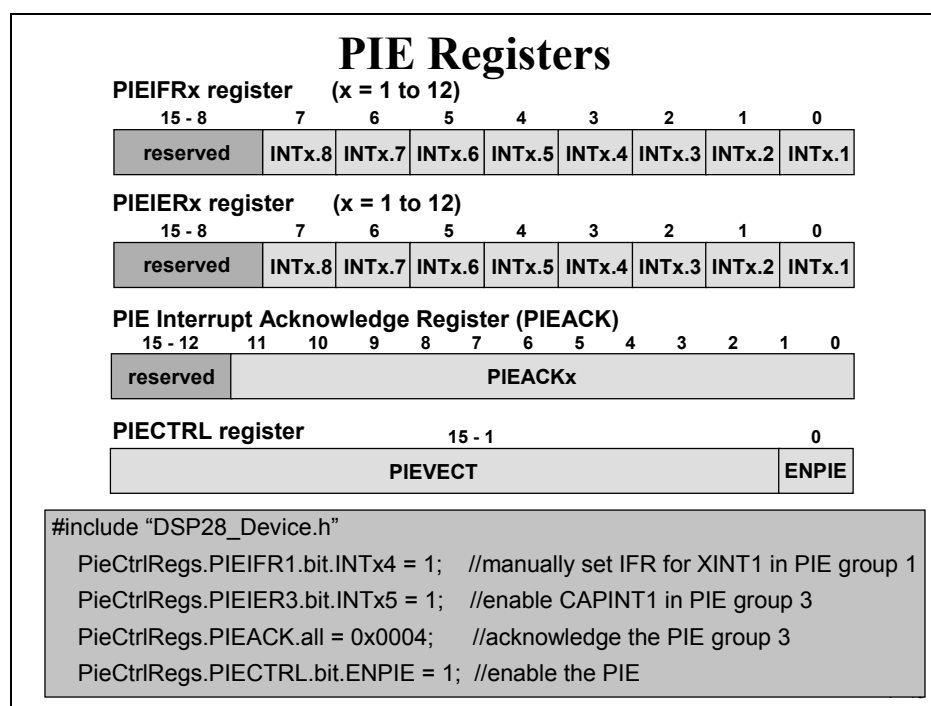
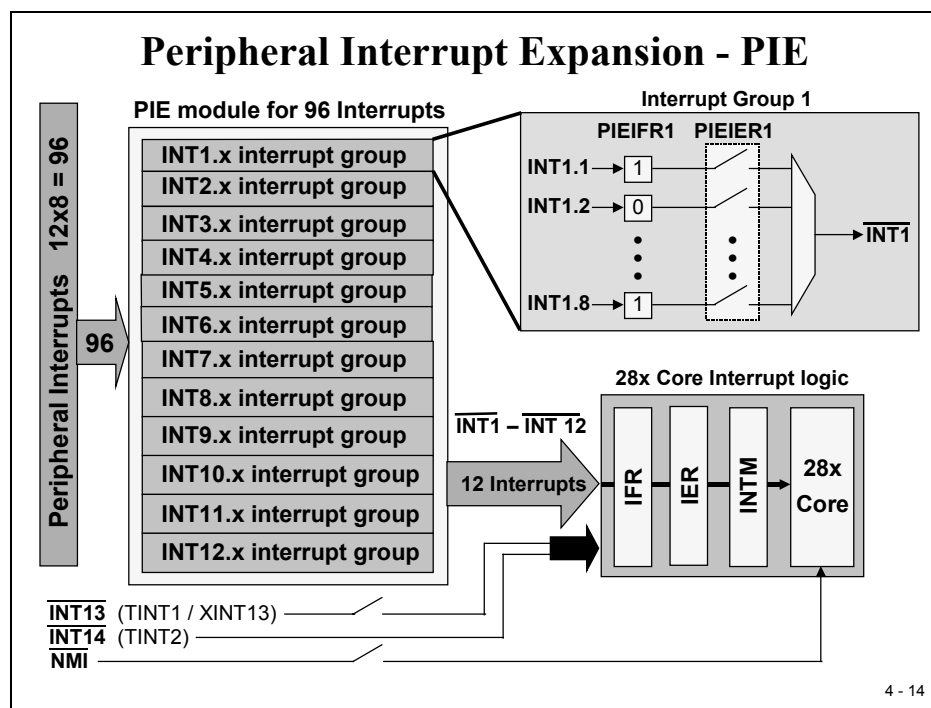
```

/** 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'.



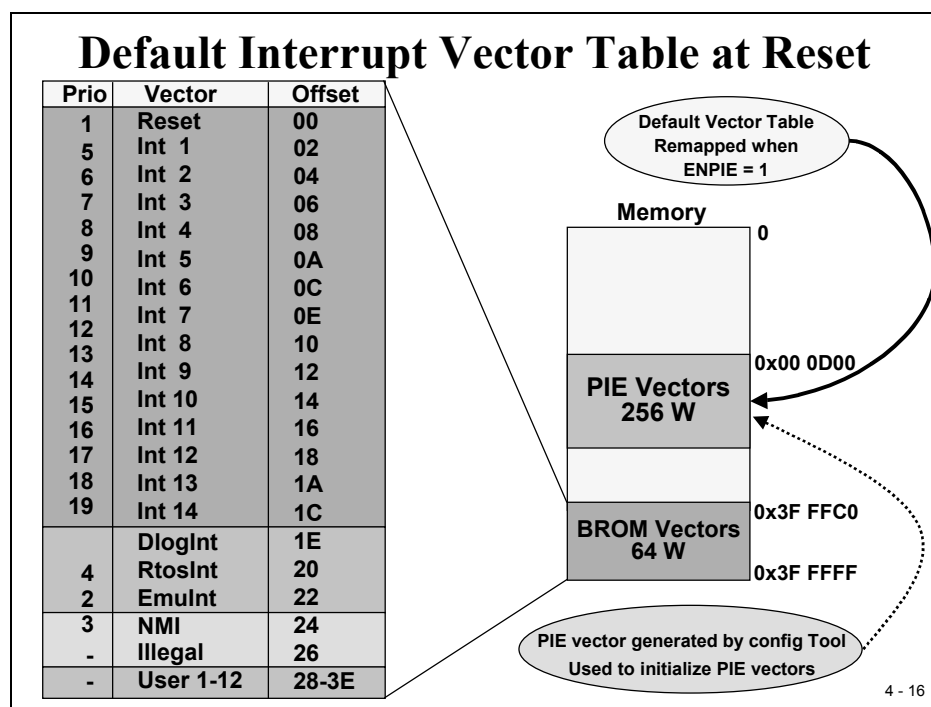
All interrupt sources are connected to interrupt lines according to this assignment table:

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:



4 - 16

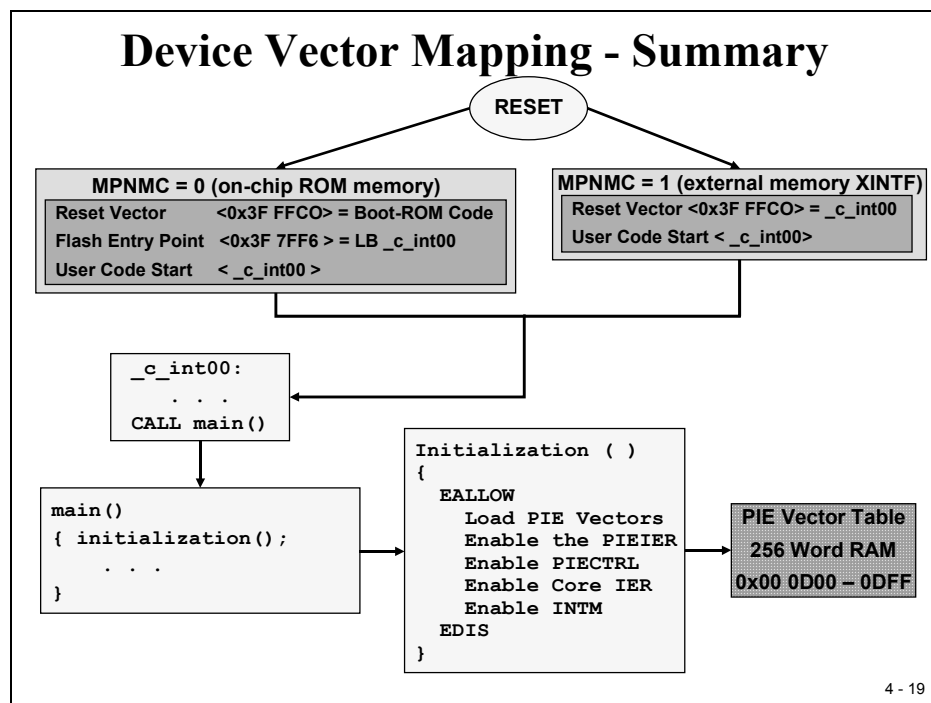
The PIE re-map location looks like this:

PIE Vector Mapping (ENPIE = 1)		
Vector name	PIE vector address	PIE vector Description
Not used	0x00 0D00	Reset Vector Never Fetched Here
INT1	0x00 0D02	INT1 re-mapped below
..... re-mapped below
INT12	0x00 0D18	INT12 re-mapped below
INT13	0x00 0D1A	XINT1 Interrupt Vector
INT14	0x00 0D1C	Timer2 - RTOS Vector
Datalog	0x00 0D1D	Data logging vector
.....
USER11	0x00 0D3E	User defined TRAP
INT1.1	0x00 0D40	PIEINT1.1 interrupt vector
.....
INT1.8	0x00 0D4E	PIEINT1.8 interrupt vector
.....
INT12.1	0x00 0DF0	PIEINT12.1 interrupt vector
.....
INT12.8	0x00 0DFE	PIEINT12.8 interrupt vector

➤ PIE vector space - 0x00 0D00 – 256 Word memory in Data space
 ➤ RESET and INT1-INT12 vector locations are Re-mapped
 ➤ CPU vectors are remapped to 0x00 0D00 in Data space

4 - 17

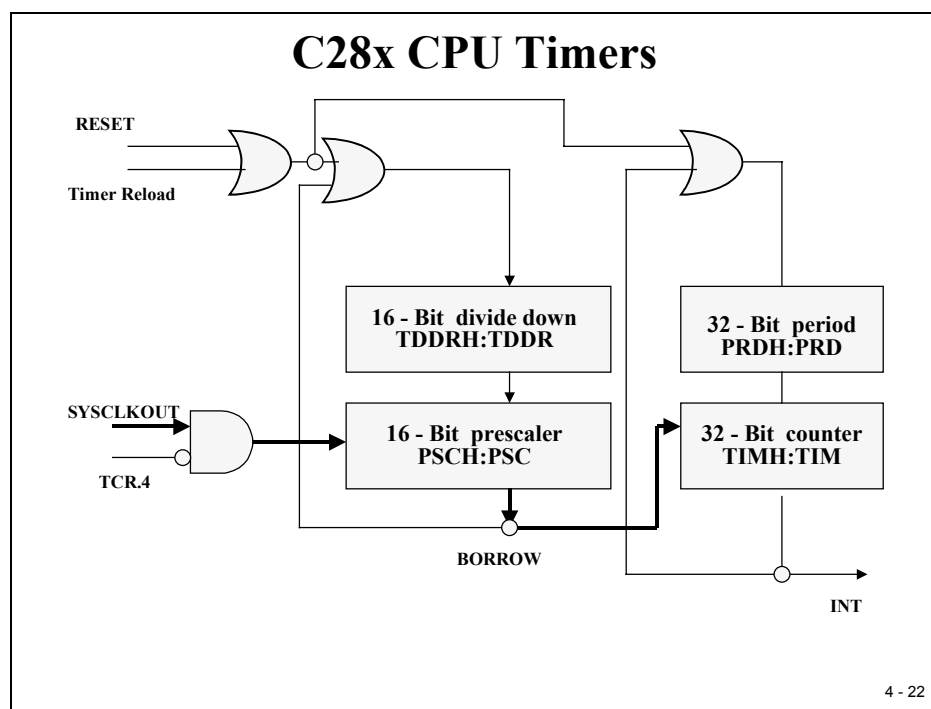
As you can see from the slide, the address area 0x00 0D40 to 0x00 0DFF is used as the expansion area. Now we do have 32 bits for each individual interrupt vector PIEINT1.1 to PIEINT12.8.



4 - 19

C28x CPU Timers

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



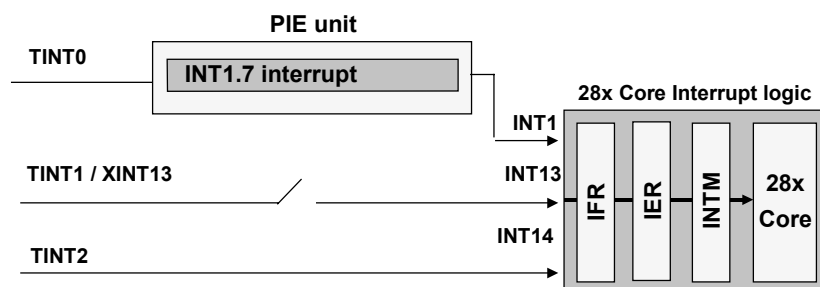
As you can see the clock source is the internal clock “SYSCLKOUT” which is usually 150MHz, assuming an external oscillator of 30MHz and a PLL-ratio of 10/2. Once the timer is enabled (TCR-Bit 4) the incoming clock counts down a 16-Bit prescaler (PSCH: PSC). On underflow, its borrow signal is used to 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 (TDDR: 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

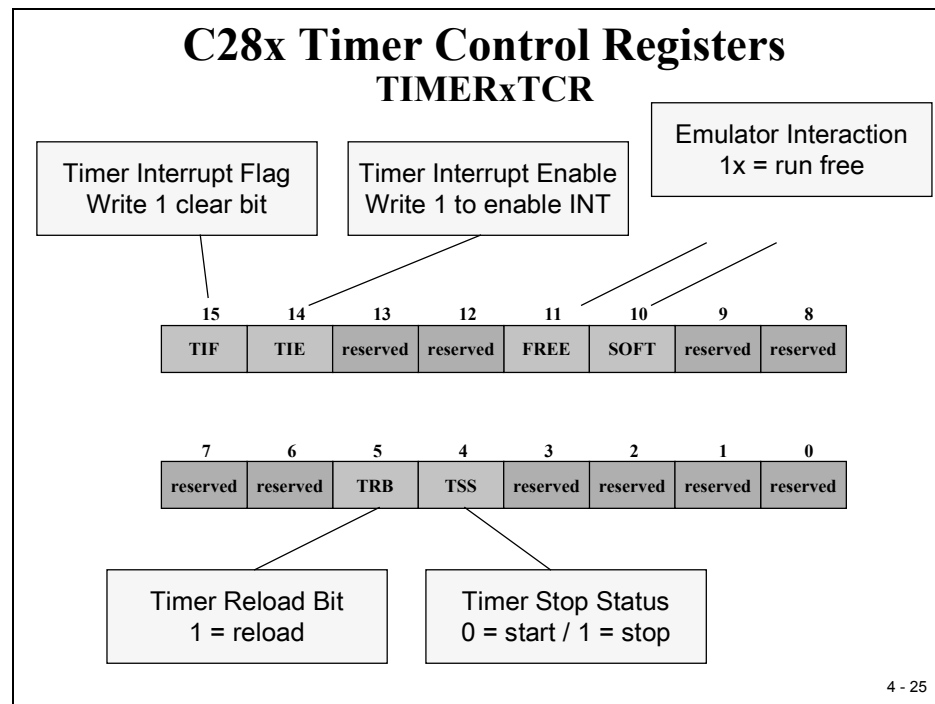


4 - 23

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_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**

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:\tides\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.

C28x Event Manager

Introduction

Now it is time to discuss one of the most powerful hardware modules of the C28x, called the 'Event Manager (EV)'. An EV is a unit that is able to deal with different types of time-based procedures. The core of this manager is a little bit similar to the DSPs core timer units Timer 0, 1 and 2. Although the Event Manager Timer units are also called "Timer 1, 2, 3 and 4", these timers are totally independent of the three core timers. So please do not mix them up! From now, when we speak of a timer unit, we have to clarify if it is a Core timer or an Event Manager timer!

The Event Manager Timer unit is a 16-bit counter/timer unit, whereas a Core Timer is a 32-bit register. The most important difference between the Event Manager and the Core timers is its input/output system. An EV is able to produce hardware signals directly from an internal time event. Thus this unit is most often used to generate time based digital hardware signals. This signal is a digital pulse with binary amplitude (0, 1). With the help of the EV-logic we can modify the frequency and/or the pulse width of these output signals. When we apply an internal control scheme to modify the shape of the signals during run time, we call this 'Pulse Width Modulation' (PWM).

PWM is used for two main purposes:

- Digital Motor Control (DMC)
- Analogue Voltage Generator

We will discuss these two main areas a little bit later. The C28x is able to generate up to 16 PWM output signals.

The Event Manager is also able to perform time measurements based on hardware signals. With the help of 6 edge detectors, called 'Capture Unit's' we can measure the time difference between two hardware signals to determine the speed of a rotating shaft in rotations per minute.

The third part of the Event Manager is called 'Quadrature Encoder Pulse' –unit (QEP). This is a unit that is used to derive the speed and direction information of a rotating shaft directly from hardware signals from incremental encoders or resolvers.

The C28x is equipped with two Event Managers, called EVA and EVB. These are two identical hardware units; two 16-bit timers within each of these EVs generate the time base for all internal operations. In case of EVA the timers are called 'General Purpose Timer' T1 and T2, in case of EVB they are called T3 and T4.

This module includes also two lab-exercises 'Lab5' and 'Lab5A' based on the eZdsp and the Zwickau Adapter board. To perform Lab5A you will need a simple analogue oscilloscope.

Module Topics

C28x Event Manager.....	5-1
<i>Introduction</i>	<i>5-1</i>
<i>Module Topics.....</i>	<i>5-2</i>
<i>Event Manager Block Diagram</i>	<i>5-3</i>
<i>General Purpose Timer.....</i>	<i>5-4</i>
<i>Timer Operating Modes</i>	<i>5-5</i>
<i>Interrupt Sources</i>	<i>5-6</i>
<i>GP Timer Registers.....</i>	<i>5-7</i>
<i>GP Timer Interrupts.....</i>	<i>5-12</i>
<i>Lab 5: Let's play a tune!.....</i>	<i>5-14</i>
<i>Event Manager Compare Units</i>	<i>5-20</i>
<i>Capture Units.....</i>	<i>5-31</i>
<i>Quadrature Encoder Pulse Unit (QEP)</i>	<i>5-36</i>
<i>Lab 5A: Generate a PWM sine wave</i>	<i>5-39</i>
<i>Optional Exercise.....</i>	<i>5-50</i>

Event Manager Block Diagram

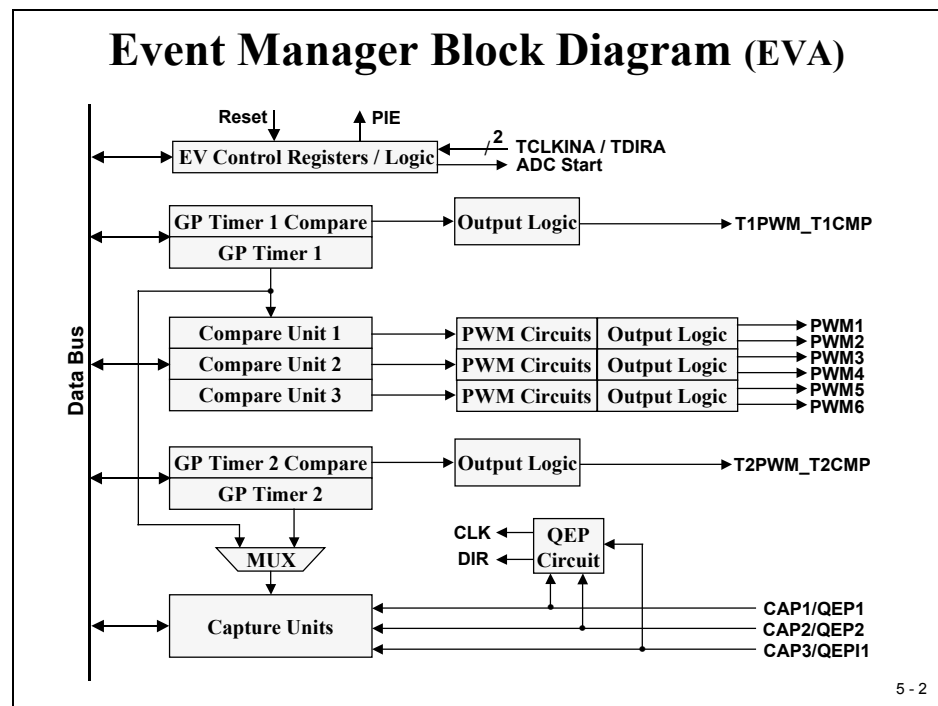
Each Event Manager is controlled by its own logic block. This logic is able to request various interrupt services from the C28x PIE unit to support its operational modes. Two external input signals 'TCLKINA' and 'TDIRA' are optional control signals and are used in some specific operational modes. A unique feature of the Event Manager is its ability to start the Analogue to Digital Converter (ADC) from an internal event. A large number of common microprocessors would have to request an interrupt service to do the same – the C28x does this automatically. We will use this feature in the next module!

The GP Timers 1 and 2 are two 16-bit timers with their own optional output signals T1PWM/T1CMP and T2PWM/T2CMP. We can also use the two timers for internal purposes only. Recall: to use any of the C28x units we have to set the multiplex registers for the I/O ports accordingly!

Compare Unit 1 to 3 are used to generate up to 6 PWM signals using GP Timer 1's time base. A large number of technical applications require exactly 6 control signals, e.g. three phase electrical motors or three phase electrical power converters.

Three independent capture units CAP1, 2, and 3 are used for speed and time estimation. An incoming pulse on one of the CAP lines will take a 'time stamp' from either GP Timer 1 or 2. This time stamp is proportional to the time between this event and the previous one.

The QEP-unit redefines the 3 input lines CAP1, 2, and 3 to be used as sensed edge pulses (QEP1, 2) and a zero degree index pulse (QEP11) for an incremental encoder.

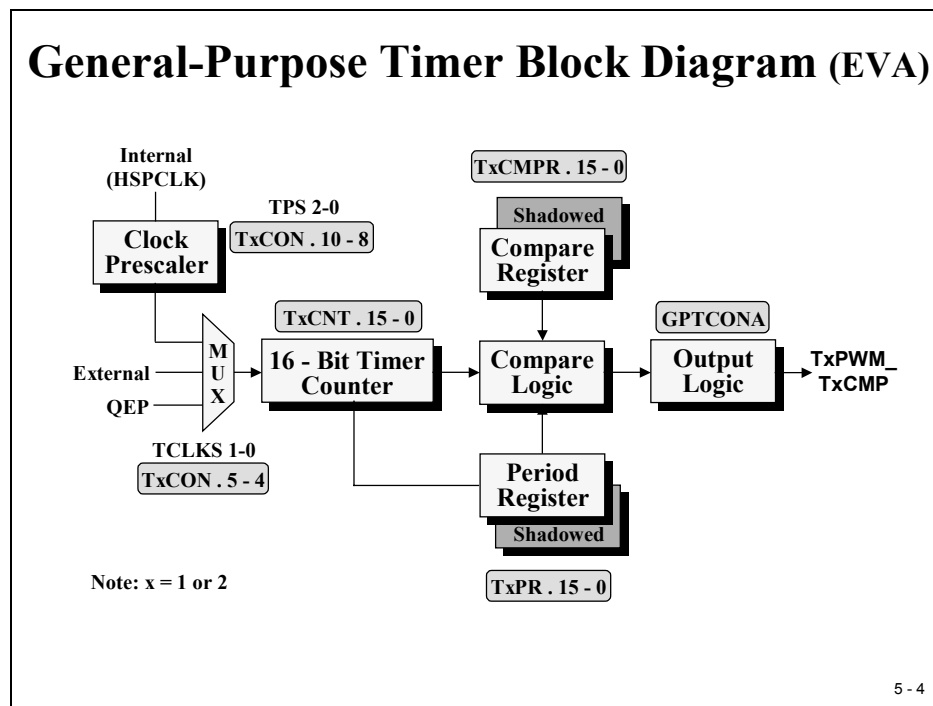


General Purpose Timer

The central logic of a General Purpose Timer is its Compare Block. This unit continually compares the value of a 16-bit counter (TxCNT) against two other registers: Compare (TxCMPR) and Period (TxPR). If there is a match between counter and compare, a signal is sent to the output logic to switch on the external output signal (TxPWM). If counter matches period, the signal is switched off. This is the basic operation in “asymmetric” mode. The second basic operating mode - “symmetric” mode – will be explained a little bit later. Register GPTCONA controls the shape of the physical output signal.

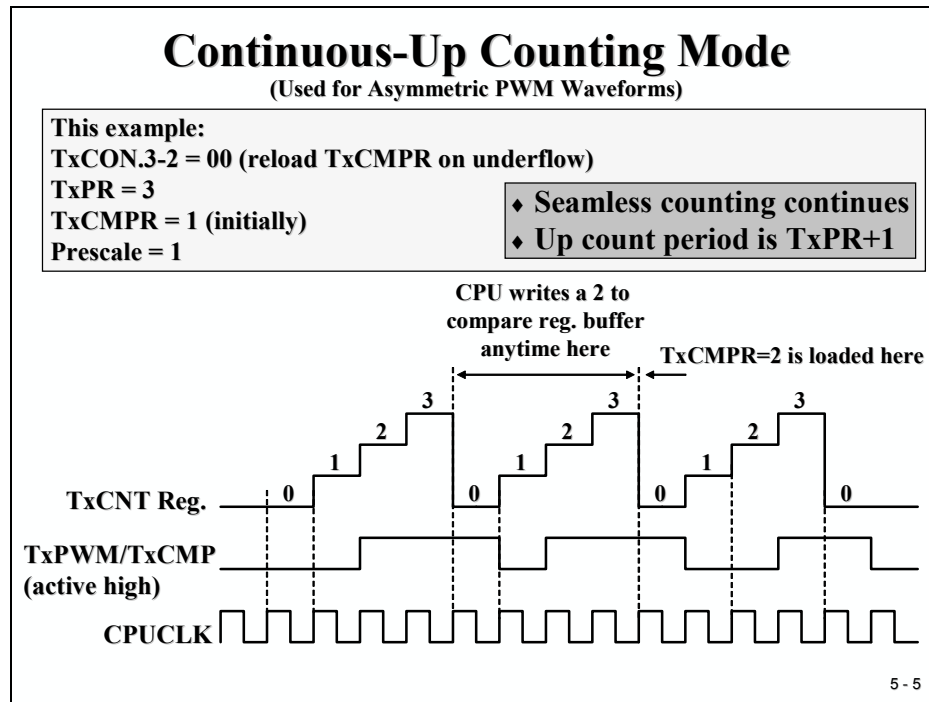
The timer’s clock source is selectable to be an external signal (TCLKIN), the QEP-unit or the internal clock. TxCON-bits5 and 4 control the multiplexer. In case of internal clock selection the clock is derived from the high-speed clock prescaler (HSPCLK). When you calculate the desired period you will have to take into account the setup of register HISPCP! To adjust the period of a General Purpose Timer one can use an additional prescaler (TPS, TxCON2-0), which gives a scaling factor between 1 and 128.

The direction of counting depends on the selected operation mode.

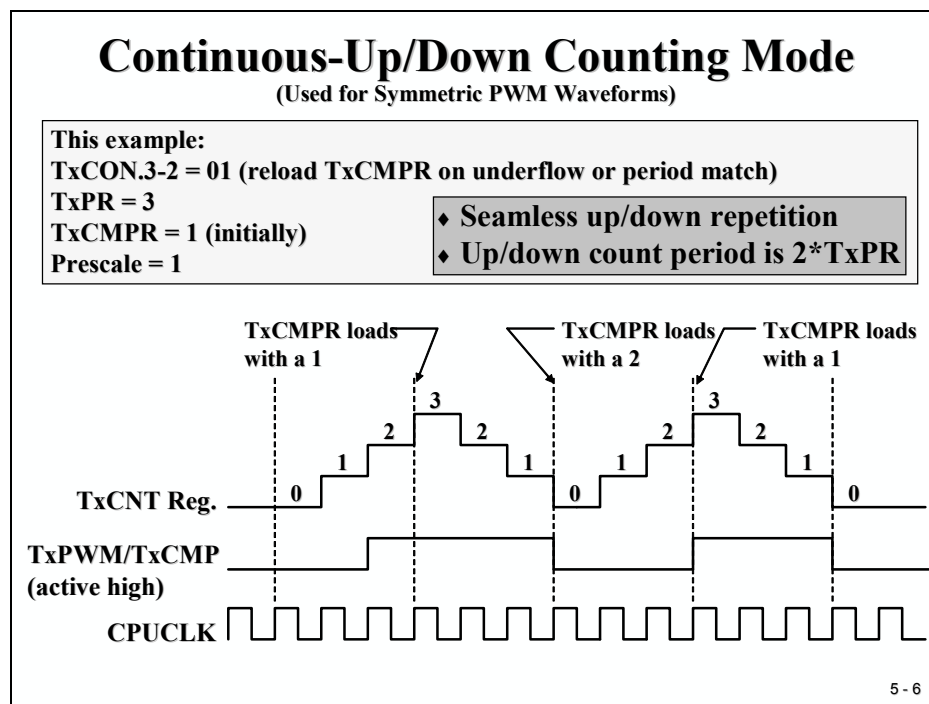


Another unique feature of the C28x is its “shadow” functionality of operating registers, in the case of GP Timers 1 and 2 available for compare register and period register. For some applications it is necessary to modify the values inside a compare or period register every period. The advantage of the background registers is that we can prepare the values for the next period in the previous one. Without a background function we would have to wait for the end of the current period, and then trigger a high prioritized interrupt. Sometimes this principle will miss its deadline...

Timer Operating Modes

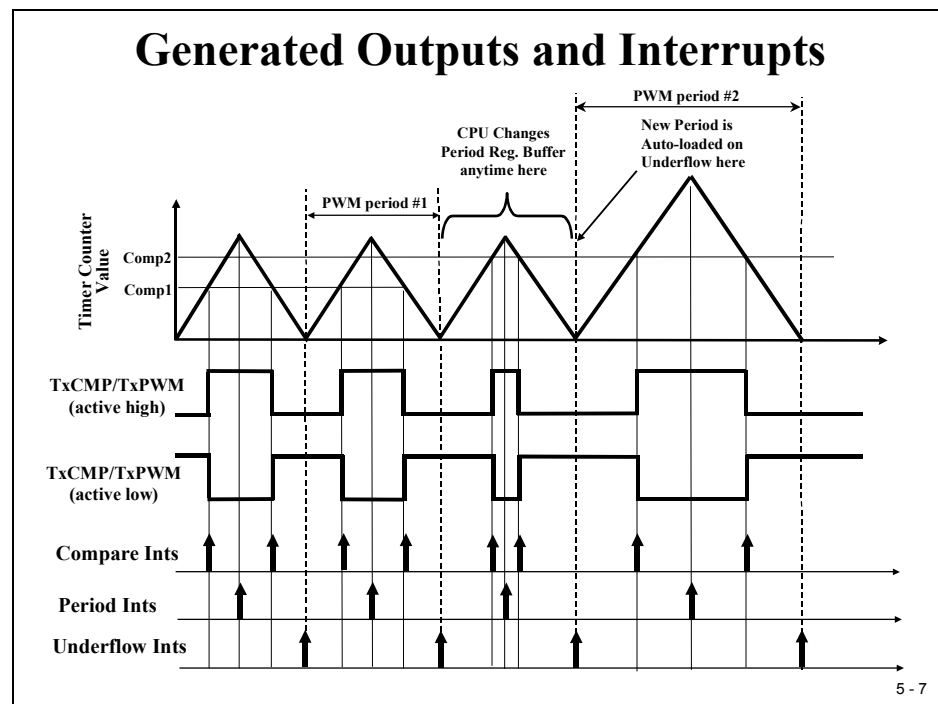


The slides give two examples of the two most used operating modes. Note: There are two more modes – see data sheet.



Interrupt Sources

Each of the two timers of Event Manager A (EVA) is able to generate four types of interrupt requests: timer underflow (counter equals zero), timer compare (counter equals compare register), timer period (counter equals period register) and timer overflow (counter equals 0xFFFF – not shown on slide). The slide also shows two options for the physical shape of the output signal (TxPWM) – “active high” and “active low”. The two options not shown are “forced low” and “forced high”. All four options are controlled by register GPTCONA.



The example on the slides assumes “Counting up/down Mode” and that the timer starts with value “Comp1” loaded into TxCMPCR and “period #1” in TXPR. At some point in period 2 our code changes the value in TxCMPCR from “Comp1” to “Comp2”. Thanks to the compare register background (or “shadow”) function this value is taken into foreground with the next reload condition. This leads to the new shape of the output signal for period 3. Somewhere in period 3 our code modifies register TxPR – preparing the shape of period 4. The answer is the PIE (Peripheral Interrupt Expansion)-unit.

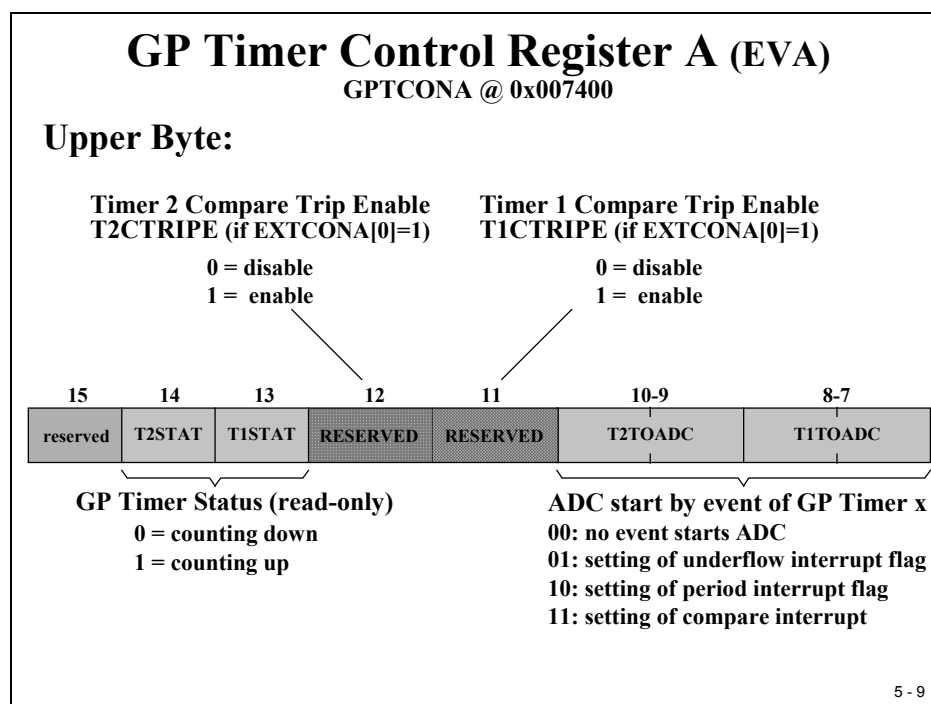
Please note that we have two points for Compare Interrupts within each period. Question: How do we distinguish between them? When we go through all timer control registers on the next pages please remember this question. There must be a way to specify whether we are in the first or second half of a period.

GP Timer Registers

To set up an Event Manager Timer we have to configure five registers per timer. If we'd like to use one or more timer interrupt sources, then we have to set up a few more registers. The next slides are going through the registers step by step, at the end a lab exercise is waiting for you!

GP Timer Registers		
Register	Address	Description
EVA	GPTCONA	0x007400 General Purpose Timer Control Register A
	T1CNT	0x007401 Timer 1 Counter Register
	T1CMPR	0x007402 Timer 1 Compare Register Buffer
	T1PR	0x007403 Timer 1 Period Register Buffer
	T1CON	0x007404 Timer 1 Control Register
	T2CNT	0x007405 Timer 2 Counter Register
	T2CMPR	0x007406 Timer 2 Compare Register Buffer
	T2PR	0x007407 Timer 2 Period Register Buffer
EVB	T2CON	0x007408 Timer 2 Control Register
	GPTCONB	0x007500 General Purpose Timer Control Register B
	T3CNT	0x007501 Timer 3 Counter Register
	T3CMPR	0x007502 Timer 3 Compare Register Buffer
	T3PR	0x007503 Timer 3 Period Register Buffer
	T3CON	0x007504 Timer 3 Control Register
	T4CNT	0x007505 Timer 4 Counter Register
	T4CMPR	0x007506 Timer 4 Compare Register Buffer
	T4PR	0x007507 Timer 4 Period Register Buffer
	T4CON	0x007508 Timer 4 Control Register
EXTCONA 0x007409 / EXTCONB 0x007509 ;Extension Control Register		

5 - 8

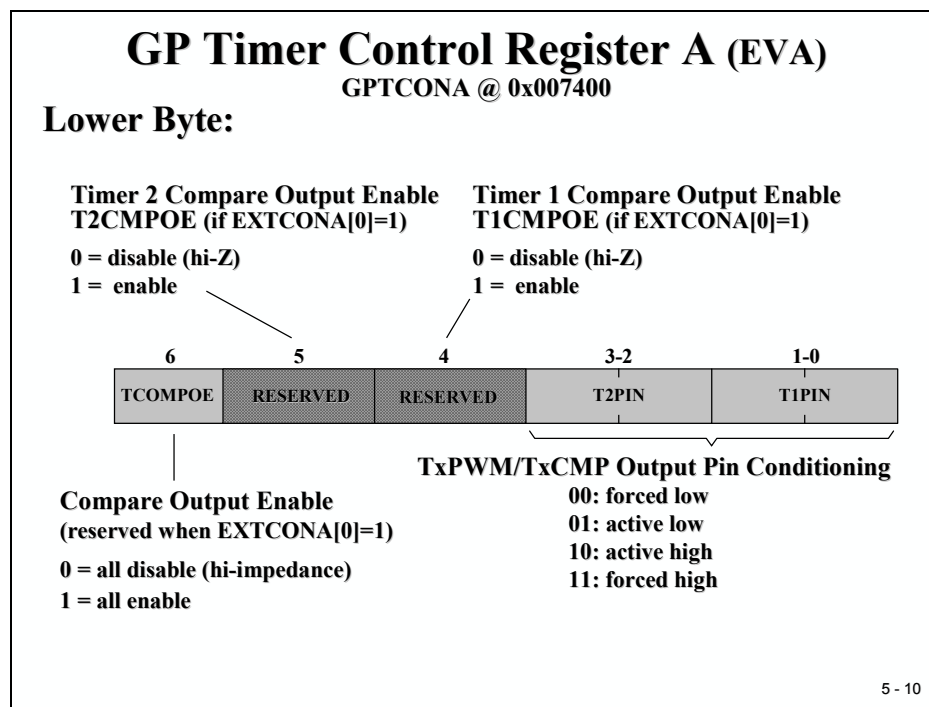


5 - 9

GP Timer Control Register GPTCONA

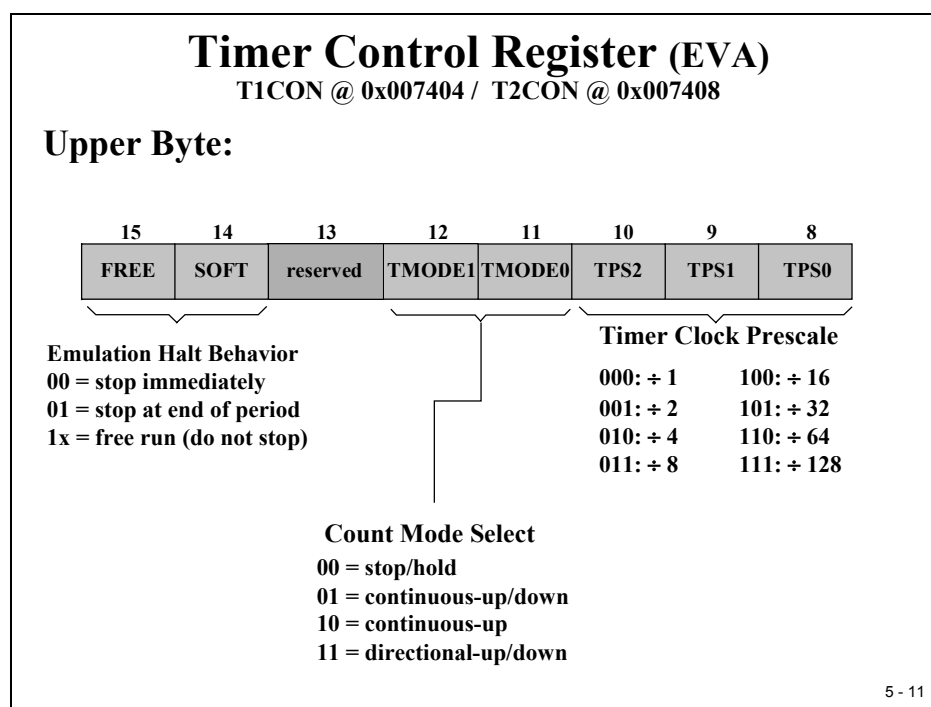
Bits 14 and 13 are status bits used to report if the timer is counting up or down. Bits 10 to 7 are used to perform the automatic start of the ADC from the specified timer event. Bits 3 to 0 define the shape of the output signal. Bit 6 is used to enable the two physical output signals for timer 1 and timer 2 simultaneously.

Note: There is an enhanced operating mode available. This extended mode is switched on by setting bit 0 in register EXTCONA to 1. In this case the definition of some of the register bits will change. Bit 6 is no longer used; instead bits 5 and 4 are used to enable/disable the output signals separately for timer 1 and timer2. Bits 12 and 11 are now used to enable a new power electronic safety feature called “Timer Compare Trip”. We will not go into these extended operating modes during this tutorial, so just treat all these new control bits as “reserved”. Reserved means for a write operation into registers you can set the bit position as a “don’t care”.



Timer Control Register TxCON

Next, we have to set up the individual timer control registers. The layout is shown on the next two slides. Bits 15 and 14 are responsible for the interaction between the timer unit and a command executed by the JTAG-Emulator unit. We will find this control bits pair for all other peripheral units of the C28x. It is very important to be able to set up a definite behaviour when, for example, the execution of our code hits a breakpoint. In case of real hardware connected to the outputs it could be very dangerous to stop the outputs in a random fashion. For the timer unit we can specify to stop its operation immediately, at the end of a period, or not at all. Of course this depends on the hardware project, for our lab exercises its good practice to stop immediately.



Bits 12-11 select the operation mode. We've discussed the two most important modes before; the two remaining modes are the "Directional Up/Down" mode and the "Stop/Hold" mode. The first mode uses an external input (TDIRA) to specify the counting direction; the latter just halts the timer in its current status, no re-initialization needed to resume afterwards.

Bits 10 to 8 are the input clock prescaler to define another clock division factor. Recall that the counting frequency is derived from:

- The external oscillator (30MHz)
- The internal PLL-status (PLLCR: multiply by 10/2 = 150 MHz)
- The High speed clock prescaler (HSPCP = divide by 2: 75 MHz) and
- The Timer Clock Prescale factor (1 to 128)

This gives us the option to specify the desired period for a timer. For example, to setup a timer period of 100 milliseconds, we can use this calculation:

$$\text{Timer input pulse} = (1/\text{ext_clock_freq}) * 1/\text{PLL} * \text{HSPCP} * \text{Timer TPS}$$

$$1,7067 \mu\text{s} = (1/30 \text{ MHz}) * 1/5 * 2 * 128$$

$$100 \text{ ms} / 1,7067 \mu\text{s} = 58593.$$

➔ Load TxPR with 58593 to set the timer period to 100 milliseconds.

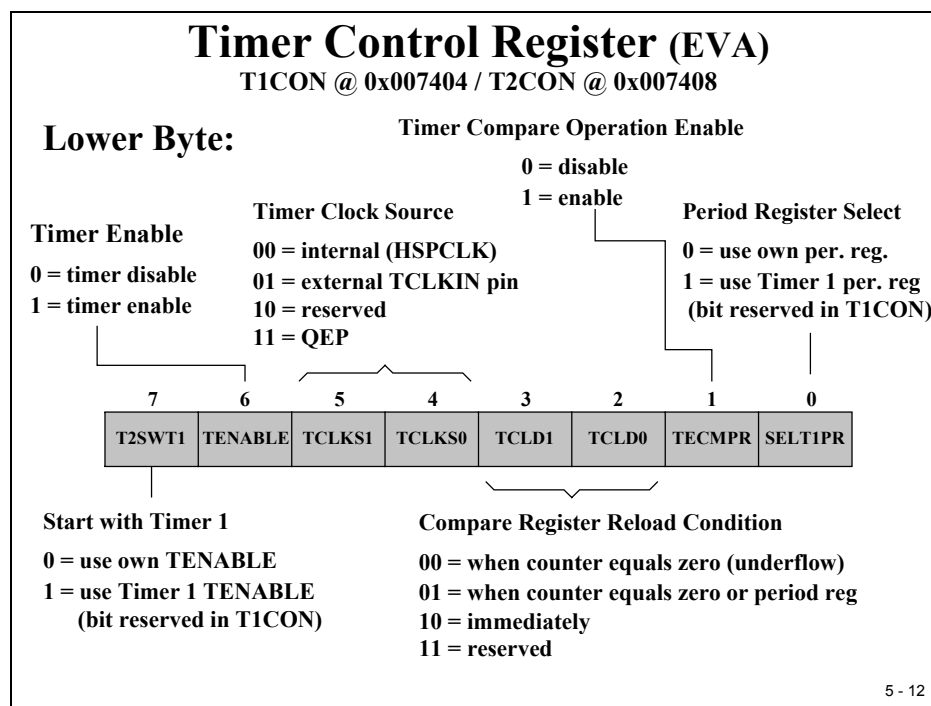
Bit 6 enables the timer operation. At the end of an initialization procedure we will have to set bit 6 to 1 to start the timer.

Bits 5 and 4 select the timer clock source; Bits 3 and 2 define the point of time to reload the value out of the background register into the foreground compare register.

Bit 1 is used to enable the compare operation. Sometimes you'll need an internal period generator only, for these applications you can switch off the compare operation that is the generation of switch patterns for the output lines.

Bits 7 and 0 are timer 2 specific bit fields. They are "don't cares" for register T1CON. With the help of bit 7 we can force a start of timer 1 and 2 simultaneously. In this case, both timers start if bit 6 (TENABLE) of T1CON is set.

Bit 0 forces timer 2 to use period register of timer 1 as base to generate a synchronized period for timer 2 and timer 1.



Now let's make a calculation. Assume that your task is to setup a PWM signal with a period of 50 kHz and a pulse width of 25%:

GP Timer Compare PWM Exercise

Symmetric PWM is to be generated as follows:

- 50 kHz carrier frequency
- Timer counter clocked by 30Mhz,
- PLL: multiply by 10/2,
- HSPCLK = divide by 2
- Use the ÷1 prescale option
- 25% duty cycle initially
- Use GP Timer Compare 1 with PWM output active high
- T2PWM/T2CMP pins forced low

Determine the initialization values needed in the GPTCONA, T1CON, T1PR, and T1CMPR registers

5 - 14

Solution :

- PLLCR =
- HSPCLK =
- GPTCONA =
- T1CON =
- T1PR =
- T1CMPR =

GP Timer Interrupts

To enable one of the interrupt sources of Event Manager A we have to set a bit inside EVAIMRA, B or C.

EVAIMRA Register							
@ 0x742C							
15	14	13	12	11	10	9	8
-	-	-	-	-	T1OFINT	T1UFINT	T1CINT
7	6	5	4	3	2	1	0
T1PINT	-	-	-	CMP3INT	CMP2INT	CMP1INT	PDPINT

Interrupt Mask Bits	Bit	Event
0 = disable interrupt	10:	Timer 1 Overflow
1 = enable interrupt	9:	Timer 1 Underflow
	8:	Timer 1 Compare match
	7:	Timer 1 Period match
	3:	Compare Unit 3, Compare match
	2:	Compare Unit 2, Compare match
	1:	Compare Unit 1, Compare match
	0:	Power Drive Protect input, EVA

5 - 16

EVAIMRB Register							
@ 0x742D							
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	-	-	-	T2OFINT	T2UFINT	T2CINT	T2PINT

Interrupt Mask Bits	Bit	Event
0 = disable interrupt	3:	Timer 2 Overflow
1 = enable interrupt	2:	Timer 2 Underflow
	1:	Timer 2 Compare match
	0:	Timer 2 Period match

5 - 17

EVAIMRC Register

@ 0x742E

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	CAP3INT	CAP2INT	CAP1INT

Interrupt Mask Bits

0 = disable interrupt

1 = enable interrupt

Bit

2:

1:

0:

Event

Capture Unit 3 input

Capture Unit 2 input

Capture Unit 1 input

5 - 18

An interrupt event will be marked by the DSP in Register EVAIFRA, B and C.

EVAIFRx Register

EVAIFRA
@ 0x742F

Read:
0 = no event
1 = flag set

15	14	13	12	11	10	9	8
-	-	-	-	-	T1OFINT	T1UFINT	T1CINT

7	6	5	4	3	2	1	0
T1PINT	-	-	-	CMP3INT	CMP2INT	CMP1INT	PDPINT

EVAIFRB
@ 0x7430

Write:
0 = no effect
1 = reset flag

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	T2OFINT	T2UFINT	T2CINT	T2PINT

EVAIFRA
@ 0x7431

15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-

7	6	5	4	3	2	1	0
-	-	-	-	-	CAP3INT	CAP2INT	CAP1INT

5 - 19

Lab 5: Let's play a tune!

Objective

The Zwickau Adapter board has a small loudspeaker connected to output T1PWM (a small on-board amplifier is also necessary) - close Jumper JP3 of the adapter board. The task for this lab exercise is to play 8 basic notes of an octave. Optionally, you can improve this lab to play a real tune. To keep it simple we will generate all notes as simple square waves. Of course, for a real musician this would be an offence because a real note is a pure sine wave and harmonics. To generate a sine wave one would have to adjust the pulse width of the PWM signal to the instantaneous voltage of the sine. This scheme is a basic principle to generate sine waves with the help of a PWM output, but as I said, let's keep it simple. If you've additional time in your laboratory and you'd like to hear pure notes, then try it later.

Lab 5: Let's play a tune !

Aim:

- Exercise with Event Manager A General Purpose Timer 1
- Use Lab 4 as a starting point. In Lab 4 we initialised Core Timer 0 to request an interrupt every 50 ms. We can use this ISR to load the next note to T1PWM.
- Timer1 output 'T1PWM' is connected to a loudspeaker

Basic Tune Frequencies:

c ¹	: 264 Hz
d	: 297 Hz
e	: 330 Hz
f	: 352 Hz
g	: 396 Hz
a	: 440 Hz
h	: 495 Hz
c ²	: 528 Hz

5 - 20

The result of Lab4 is a good starting point for Lab5. Recall that we initialized the core timer 0 to request an interrupt service every 50 milliseconds. Now we can use this interrupt service routine to load the next note into the period and compare register of T1. A time of 50 milliseconds is a little bit too fast, but we have a 50ms variable "CpuTimer0.InterruptCount". If we wait until the value of this variable is 10 we know that an interval of 500ms is over. After this period we can play the next note starting with c¹ and go to c² in an endless loop. Or, try to play the notes alternately as an ascending and descending sequence (or: recall a nursery rhyme).

New Registers involved in Lab 5:

• General Purpose Timer Control A	:	GPTCONA
• Timer 1 Control Register	:	T1CON
• Timer 1 Period Register	:	T1PR
• Timer 1 Compare Register	:	T1CMPR
• Timer 1 Counter Register	:	T1CNT
• EV- Manager A Interrupt Flag A	:	EVAIFRA
• EV- Manager A Interrupt Flag B	:	EVAIFRB
• EV-Manager A Interrupt Flag C	:	EVAIFRC
• EV- Manager A Interrupt Mask A	:	EVAIMRA
• EV- Manager A Interrupt Mask B	:	EVAIMRB
• EV- Manager A Interrupt Mask C	:	EVAIMRC
• Interrupt Flag Register	:	IFR
• Interrupt Enable Register	:	IER

5 - 21

Procedure**Open Files, Create Project File**

1. Create a new project, called **Lab5.pjt** in E:\C281x\Labs.
2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab5.c in E:\C281x\Labs\Lab5.
3. Add the source code file to your project:
 - **Lab5.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**

From `C:\tidcs\c28\dsp281x\v100\DSP281x_common\source` add to project:

- **DSP281x_CpuTimers.c**
- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**

Project Build Options

5. 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>**.

Build and Load

7. 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.

So far we just generated a new project ‘Lab5’ with the old code from Lab4. If you run the code now you should see the ‘Knight Rider’ of Lab4. Now we can start to modify our code in ‘Lab5.c’.

Modify Source Code

8. Open Lab5.c to edit: double click on "Lab5.c" inside the project window. First we have to cancel the parts of the code that we do not need any longer. The definition of array "LED [8]" in function "main" is of no use for this lab – cancel it.
9. Next we have to change the GPIO multiplex status; we need T1PWM as signal at the pin. Go into function "Gpio_select()" and modify the multiplex register setup.
10. Go into your local function "InitSystem" and enable the clock for Event Manager A.
11. Inside "main", just before the line:

CpuTimer0Regs.TCR.bit.TSS = 0;

we have to initialize the Event Manager Timer 1 to produce a PWM signal. This involves the registers "GPTCONA", "T1CON", "T1CMPR" and "T1PR".

For register "GPTCONA" it is recommended to use the bit-member of this predefined union to set bit "TCMPOE" to 1 and bit field "T1PIN" to "active low".

For register "T1CON" set

- The "TMODE"-field to "counting up mode";
 - Field "TPS" to "divide by 128";
 - Bit "TENABLE" to **"disable timer"** (we will enable it later)
 - Field "TCLKS" to "internal clock"
 - Field "TCLD" to "reload on underflow" and
 - Bit "TECMPR" to "enable compare operation"
12. Last question is: how do we initialize "T1PR"? Well, obviously we need 8 different values for our 8 basic notes. So let's define a new integer array "frequency [8]" as a local variable in main!
 13. How do we initialize array "frequency [8]"?

We can initialize the array together with the definition inside main:

int frequency [8] = {?, ?, ?, ?, ?, ?, ?, ?};

A basic octave is a fixed series of 8 frequencies. AND: there is a relationship between the basic note c1 (264 Hz) and the next notes:

264 Hz (c1)	396Hz (g) = $3/2 * c1$
297Hz (d) = $9/8 * c1$	440Hz (a) = $5/3 * c1$
330Hz (e) = $5/4 * c1$	495Hz (b) = $15/8 * c1$
352Hz (f) = $4/3 * c1$	528 Hz (c2) = $2 * c1$

What is the relationship between these frequencies and T1PWM? Answer: We have to setup T1PR to generate a PWM period according to this list. The equation is:

$$\mathbf{T1_PWM_Freq = 150MHz / (HISPCP * TPS * T1PR)}$$

For c1 = 264 Hz we get: $T1PR = 150MHz / (2 * 128 * 264 \text{ Hz}) = 2219$.

For d = 297 Hz we use: $T1PR = 150MHz / (2 * 128 * 297 \text{ Hz}) = 1973$.

Calculate the 8 initial numbers and complete the initial part for array “frequency”!

14. Next step: Modify the endless while(1) loop of main! Recall:

- i. The Core Timer T0 requests an interrupt every 50 milliseconds.
- ii. The Watchdog Timer is alive! It will trigger a reset after $33ns * 512 * 256 * WDPS$. If WDPS was initialized to 64 this reads as 280ms.
- iii. Timer T0 Interrupt Service Routine increment variable “CpuTimer0.InterruptCount” every 50 milliseconds

We have to reset the watchdog every 200 milliseconds and we should play the next note after 500 milliseconds. Two tasks within this while(1) loop. Later we will learn that this type of multi tasking is much better solved with the help of “DSP/BIOS” – Texas Instruments Real Time Operating System. For now we have to do it by our self.

How can we find out, if a period of 200ms is over?

We just have to test if “CpuTimer0.InterruptCount” is a multiple of 4. In language C this could be done by modulo division with 4 → remainder is zero:

if ((CpuTimer0.InterruptCount%4)==0)

If it is TRUE then we have to perform the second half of the watchdog re-trigger sequence:

EALLOW;

SysCtrlRegs.WDKEY = 0xAA;

EDIS;

In a similar technique we can wait for 10 times 50 ms = 500 ms before we apply the next note into T1PR and T1CMPR. In Lab4 we did a reset of variable “CpuTimer0.InterruptCount” every time a period was over. Doing so, we limited the values for this variable between 0 and 3, which was fine for this single task exercise. When we have to take care of more activities with different periods, it is not a good recommendation to reset this variable. A better approach is to build a time interval out of two read operations of “CpuTimer0.InterruptCount”. With the first access we gather the actual time and store this value in a local unsigned long variable

“time_stamp”. With a second access to “CpuTimer0.InterruptCount” we can read the new time information and the difference between this value and the value of “time_stamp” is the elapsed time in multiples of 50ms.

A wait instruction for 500ms could now look like this:

if ((CpuTimer0.InterruptCount – time_stamp) > 10)

If TRUE, then:

- Load “time_stamp” with “CpuTimer0.InterruptCount”
- Load the next note into EvaRegs.T1PR
- Load $\text{EvaRegs.T1CMPR} = \text{EvaRegs.T1PR}/2$
- Enable T1PWM, set $\text{EvaRegs.T1CON.bit.TENABLE} = 1$
- Implement and handle a status counter (variable i) to loop through array “frequency[8]”

Build and Load

15. 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.

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

File → Load Program and choose the desired output file.

Test

17. Reset the DSP by clicking on:

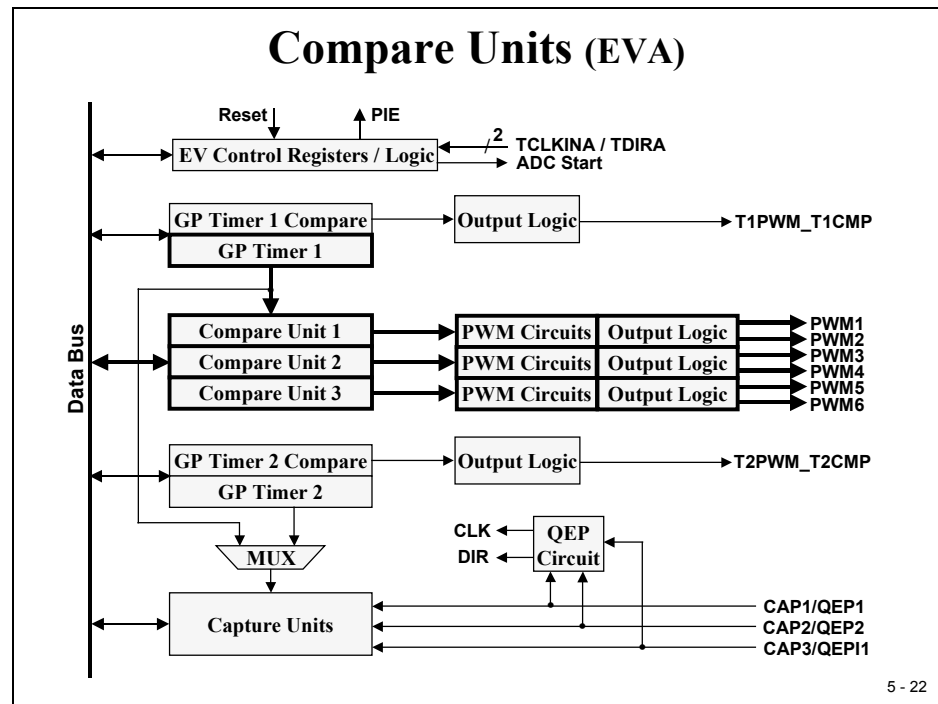
Debug → Reset CPU followed by
Debug → Restart

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

Debug → Go main.

19. Debug your code as you’ve done in previous labs.

Event Manager Compare Units



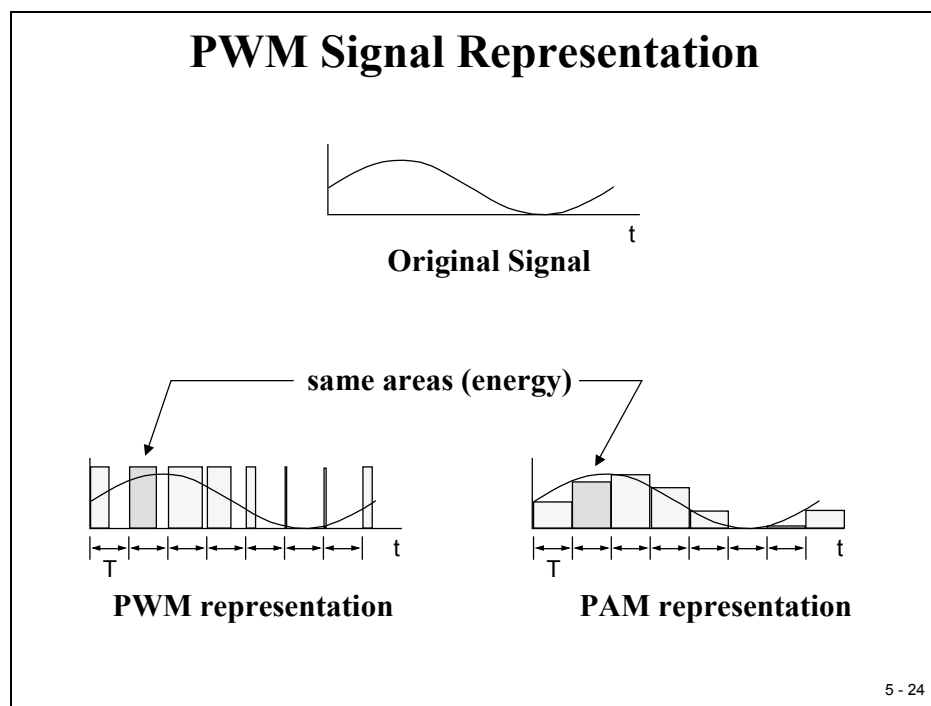
A compare unit is a peripheral that is designed to generate pulse width modulated (PWM) output signals. What is a PWM signal and what is it used for?

What is Pulse Width Modulation?

- ◆ **PWM is a scheme to represent a signal as a sequence of pulses**
 - fixed carrier frequency
 - fixed pulse amplitude
 - pulse width proportional to instantaneous signal amplitude
 - PWM energy \approx original signal energy
- ◆ **Differs from PAM (Pulse Amplitude Modulation)**
 - fixed width, variable amplitude

5 - 23

With a PWM signal we can represent any analogue output signal as a series of digital pulses! All we need to do with this pulse series is to integrate it (with a simple low pass filter) to imitate the desired signal. This way we can build a sine wave shaped output signal. The more pulses we use for one period of the desired signal, the more precisely we can imitate it. We speak very often of two different frequencies, the PWM-frequency (or sometimes “carrier frequency”) and the desired signal frequency.



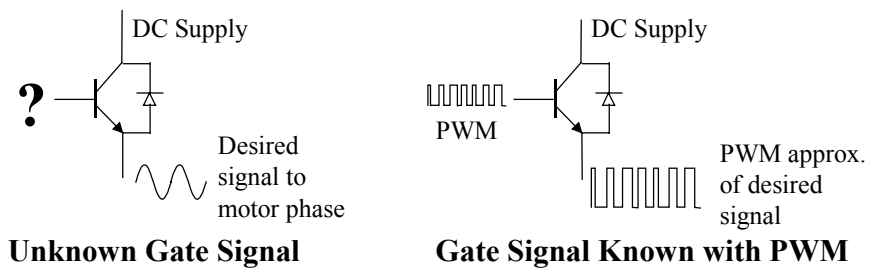
A lot of practical applications have an internal integrator, for example the windings of an electrical motor are perfectly suited to behave as a low-pass filter.

One of the most used applications of PWM is digital motor control. Why is that? Answer: The overall goal is to control electrical drives by imprinting harmonic voltages and currents into the windings of the motor. This is done to avoid electromagnetic distortions of the environment and to achieve a high power factor. To induce a sine wave shaped signal into the windings of a motor we would have to use an amplifier to achieve high currents. The simplest amplifier is a standard NPN or PNP transistor that proportionally amplifies the base current into the collector current. Problem is, for high currents we can't force the transistor into its linear area; this would generate a lot of thermal losses and for sure exceed its maximal power dissipation.

The solution is to use this transistor in its static switch states only (On: $I_{ce} = I_{cesat}$, Off: $I_{ce} = 0$). In this states a transistor has its smallest power dissipation. AND: by adapting the switch pattern of a PWM (recall: amplitude is 1 or 0 only) we can induce a sine wave shaped current!

Why Use PWM in Digital Motor Control?

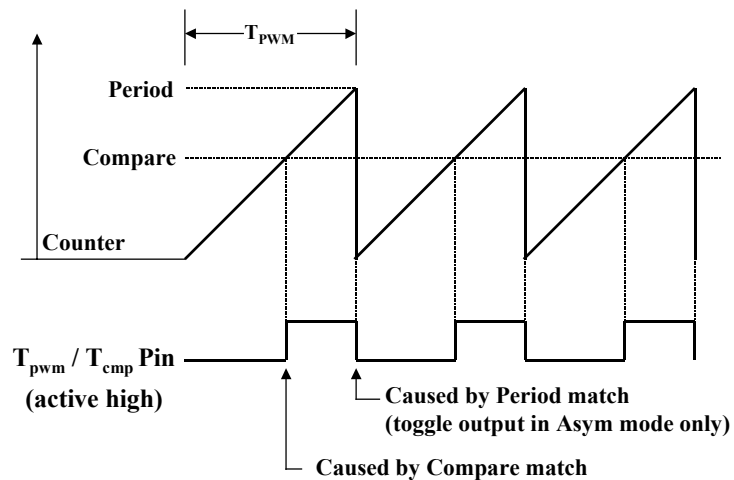
- ◆ Desired motor phase currents or voltages are known
- ◆ Power switching devices are transistors
 - Difficult to control in proportional region
 - Easy to control in saturated region
- ◆ PWM is a digital signal \Rightarrow easy for DSP to output



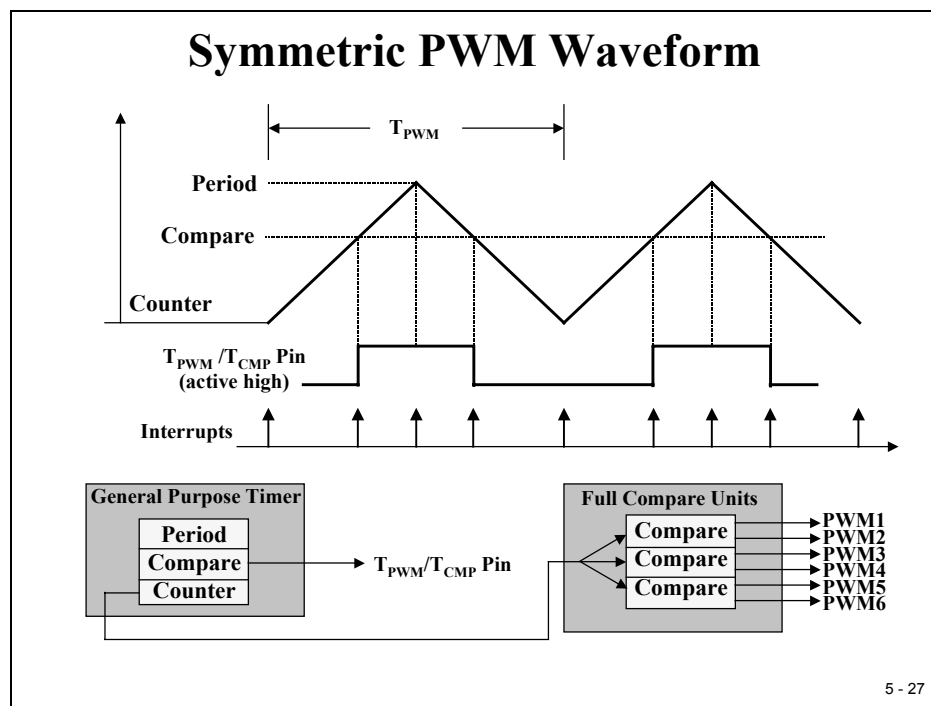
5 - 25

We have two different options to generate a PWM-signal, asymmetric and symmetric PWM.

Asymmetric PWM Waveform



5 - 26



5 - 27

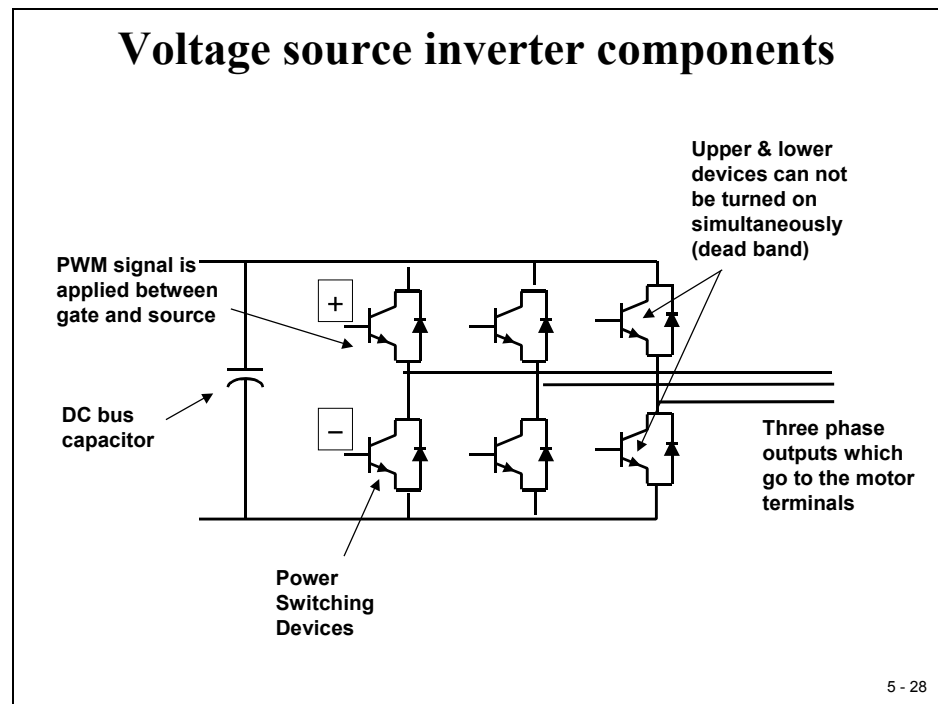
NOTE: The value in T1PR defines the length of a period “TPWM” in asymmetric operating mode. For symmetric mode the value of TxPR defines only half of the length of a period “TPWM”.

The Compare Unit consists of 6 output signals “PWM1” to “PWM6”. The time base is derived from Event Manager Timer1, e.g. register “T1PR” together with the setup for T1 (Register “T1CON”) defines the length of a PWM-period for all six output signals. Register “T1CNT” is used as the common counter register.

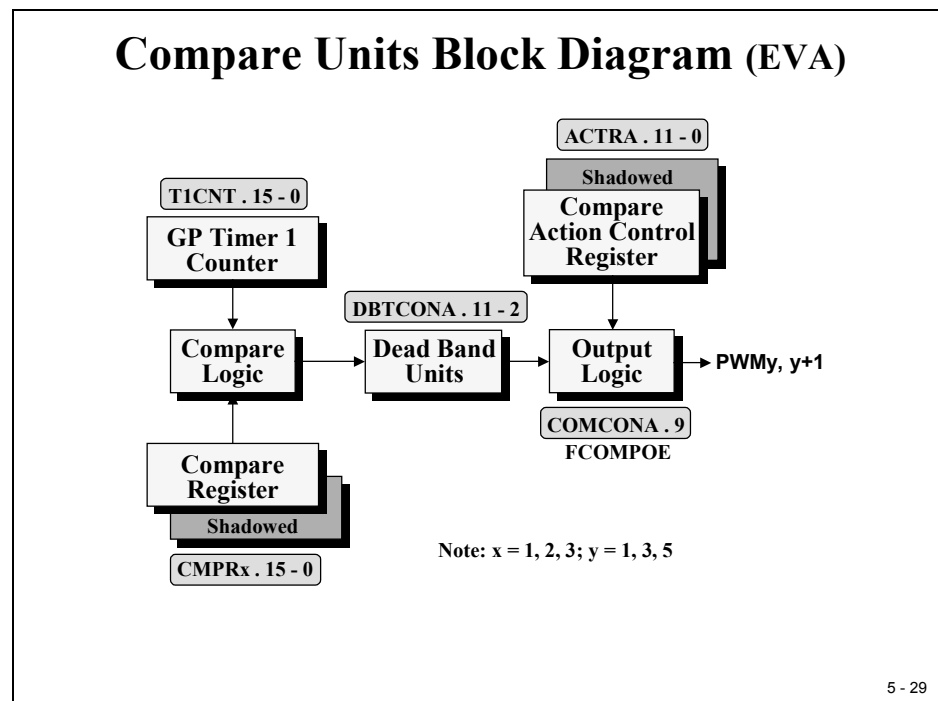
With 3 new registers “CMPR1”, “CMPR2” and “CMPR3” we can specify 3 different switch pattern based on T1PR. Obviously this leads to a 3-phase control pattern for 3 phase electrical motors.

Each Compare Unit is able to drive a pair of two output signals. With the help of its own output logic we usually define the two lines to be opposite or 180-degree out of phase to each other - a typical pattern for digital motor control.

The next slide shows a typical layout for a three-phase power-switching application.



Compare Units Block Diagram



The central block of the Compare Unit is a compare logic that compares the value of Event Manager Timer 1 counter register “T1CNT” against Compare Register “CMPRx”. If there is a first match, a rising edge signal goes into the next block called “Dead Band Unit”. With the second match between “T1CNT” and “CMPRx” in symmetric PWM mode a falling edge signal is generated. We will discuss this “Dead Band Unit” a little bit later. We do have three Compare Units available.

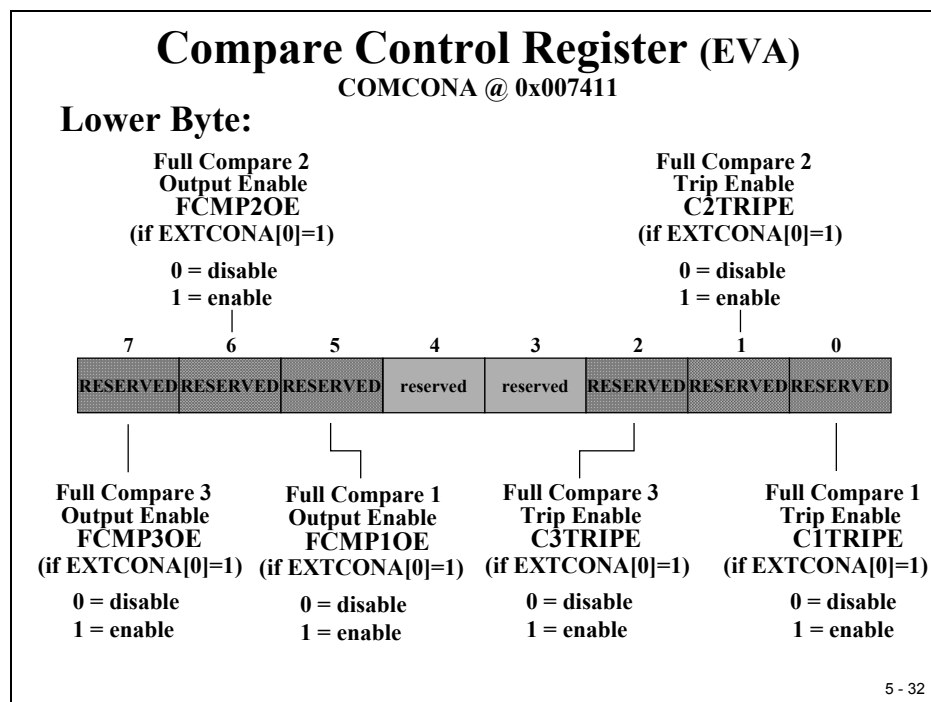
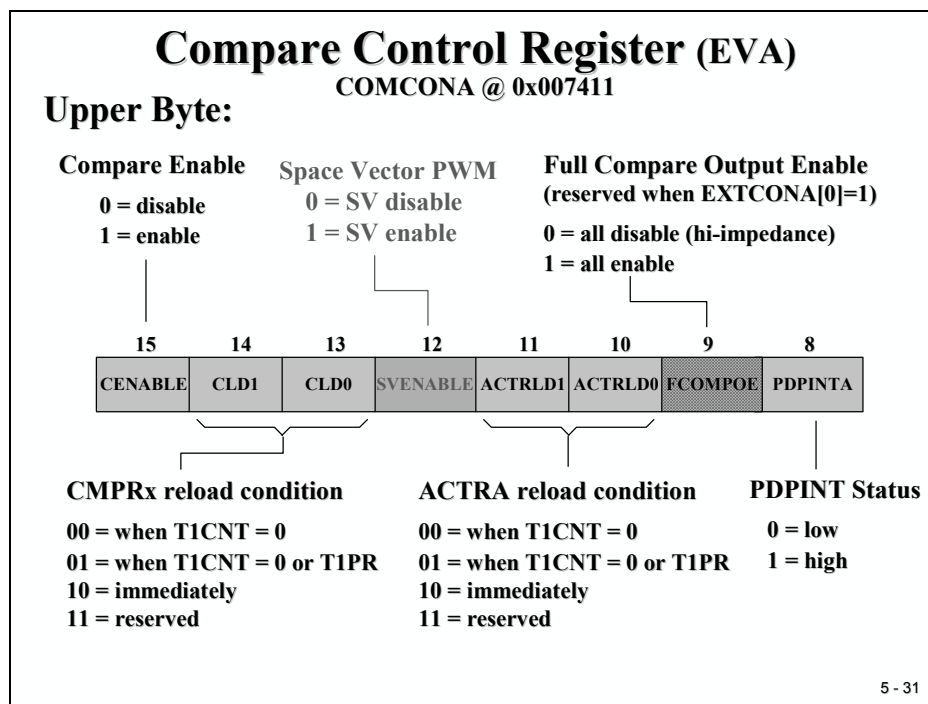
The output logic is controlled by means of a register, called “Action Control Register – ACTRA” and register “COMCONA”. With the help of this register set we can adjust the shape of the physical PWM output signal to our needs. We can specify four types for all 6 output lines:

- Active High:
 - First CMPRx match switches PWM output from 0 to 1. After second CMPRx match the signal is set back to 0.
- Active Low:
 - First CMPRx match switches PWM output from 1 to 0. After second CMPRx match the signal is setback to 1.
- Forced High:
 - PWM output always at 0.
- Forced Low:
 - PWM output always at 1.

Compare Unit Registers		
	Register	Address Description
EVA	COMCONA	0x007411 Compare Control Register A
	ACTRA	0x007413 Compare Action Control Register A
	DBTCONA	0x007415 Dead-Band Timer Control Register A
	CMPR1	0x007417 Compare Register 1
	CMPR2	0x007418 Compare Register 2
	CMPR3	0x007419 Compare Register 3
EVB	COMCONB	0x007511 Compare Control Register B
	ACTRB	0x007513 Compare Action Control Register B
	DBTCONB	0x007515 Dead-Band Timer Control Register B
	CMPR4	0x007517 Compare Register 4
	CMPR5	0x007518 Compare Register 5
	CMPR6	0x007519 Compare Register 6
	EXTCONA 0x007409 / EXTCONB 0x007509 ;Extension Control Register	

5 - 30

The next two slides explain the set up for the individual bit fields of COMCONA. Most of the bits are reserved in basic operation mode (EXTCONA [0] = 0).



COMCONA [15] is the enable bit for the three phase compare units.

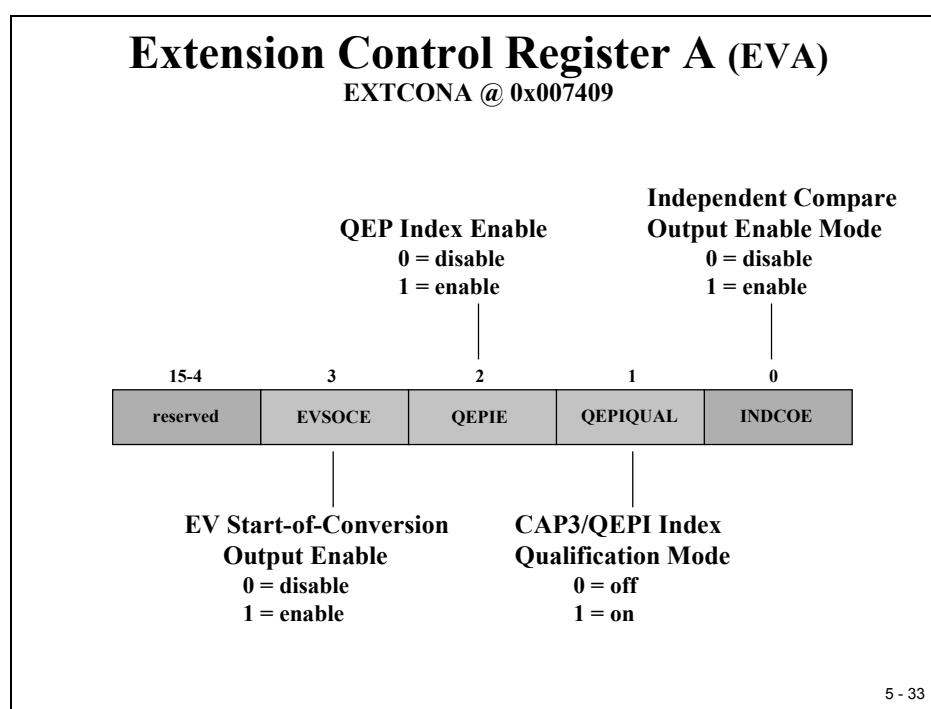
With COMCONA [14:13] and COMCONA [11:10] we specify the point in time when the compare registers and action control registers are reloaded (shadow register content into foreground). As we have seen with the timers we can prepare the next period in the current running period.

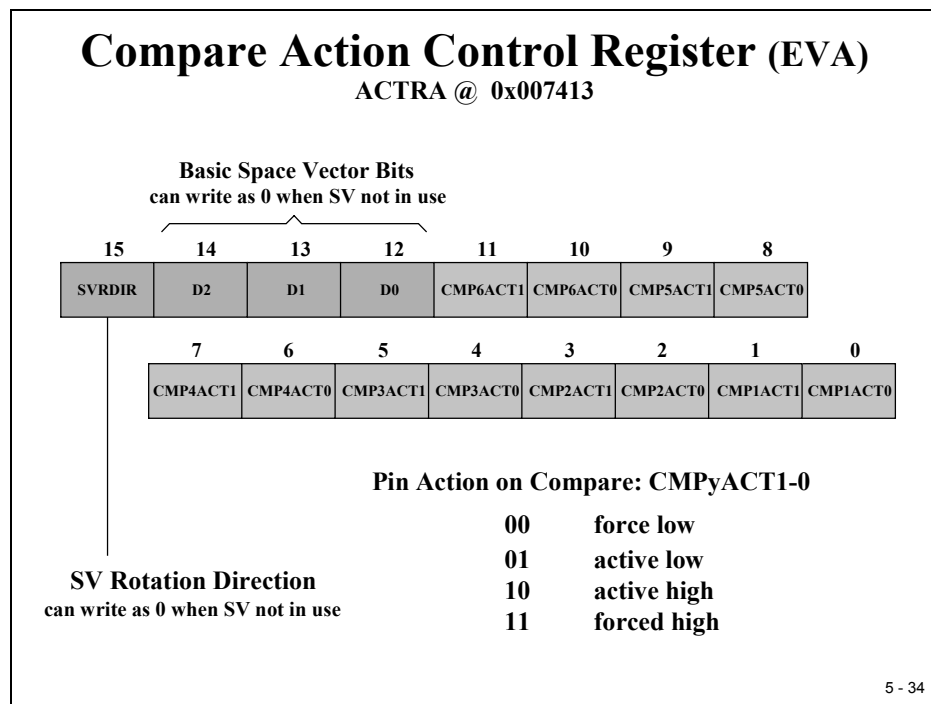
COMCONA [8] shows the status of the power drive protection flag. If it is a 1, the DSP has seen an interrupt request from its over-current input PDPINT.

With EXTCONA [0] =1 all three pairs of compare output lines can be enabled independently of each other.

If EXTCONA [0] =0 then all six lines are enabled with COMCONA [9] =1. If EXTCONA [0] =1 we can use three more individual over current inputs signals. To use these over current signals we can enable or disable this feature using bits COMCONA [2:0].

COMCONA [12] = 1 enables a special switch pattern for digital motor control, called “Space Vector Modulation” (SVM). This feature is built-in hardware support for one specific theoretical control algorithm. For details see literature or your lectures about power electronics.





ACTRA [11:0] define the shapes of the six PWM output signals as discussed before.

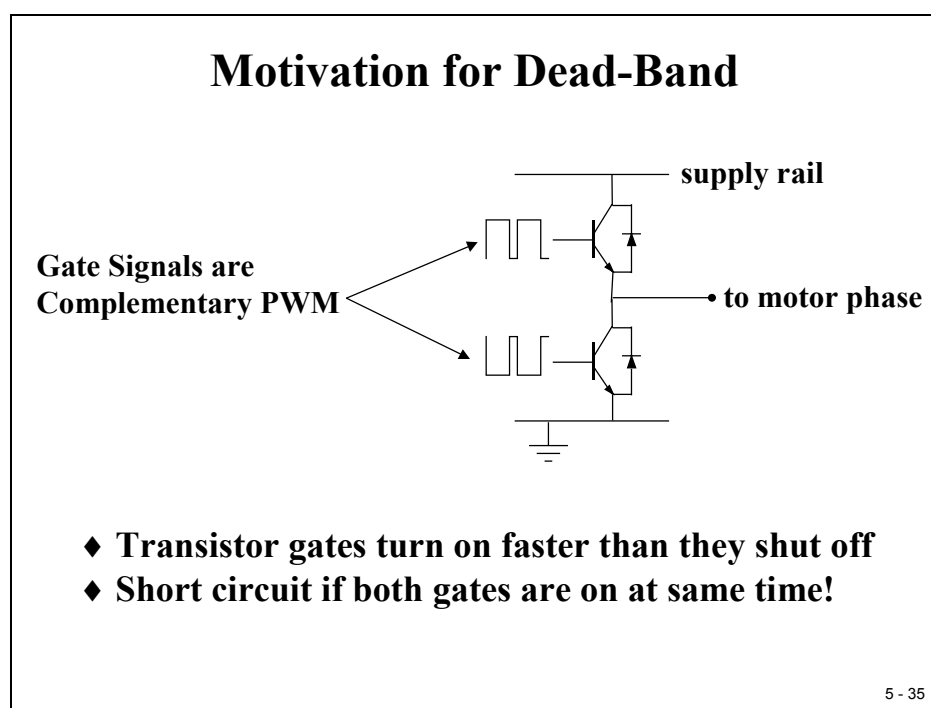
ACTRA [15:12] are used to support Space Vector Modulation. Bit 15 defines the rotation direction of the resulting electromagnetic vector as clockwise or anti clockwise.

ACTRA [14:12] declare the Basic Space Vector for the next PWM periods. A Basic Space Vector is a 60-degree section of the unit circle. This gives 6 vectors per rotation plus two virtual vectors with no current imprint.

If SVM is not used, one can initialize bits 15:12 to zero.

Hardware Dead Band Unit

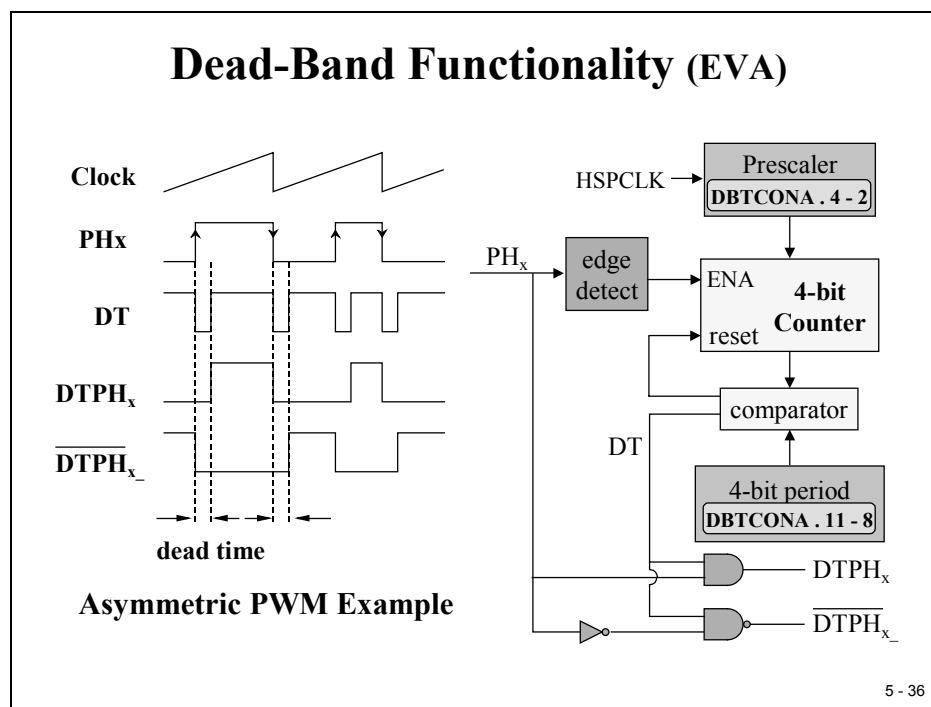
Dead-band control provides a convenient means of combating current “shoot-through” problems in a power converter. “Shoot-through” occurs when both the upper and lower transistors in the same phase of a power converter are on simultaneously. This condition shorts the power supply and results in a large current draw. Shoot-through problems occur because transistors (especially FET’s) turn on faster than they turn off, and also because high-side and low-side power converter transistors are typically switched in a complimentary fashion. Although the duration of the shoot-through current path is finite during PWM cycling, (i.e. the transistor will eventually turn off), even brief periods of a short circuit condition can produce excessive heating and stress the power converter and power supply.



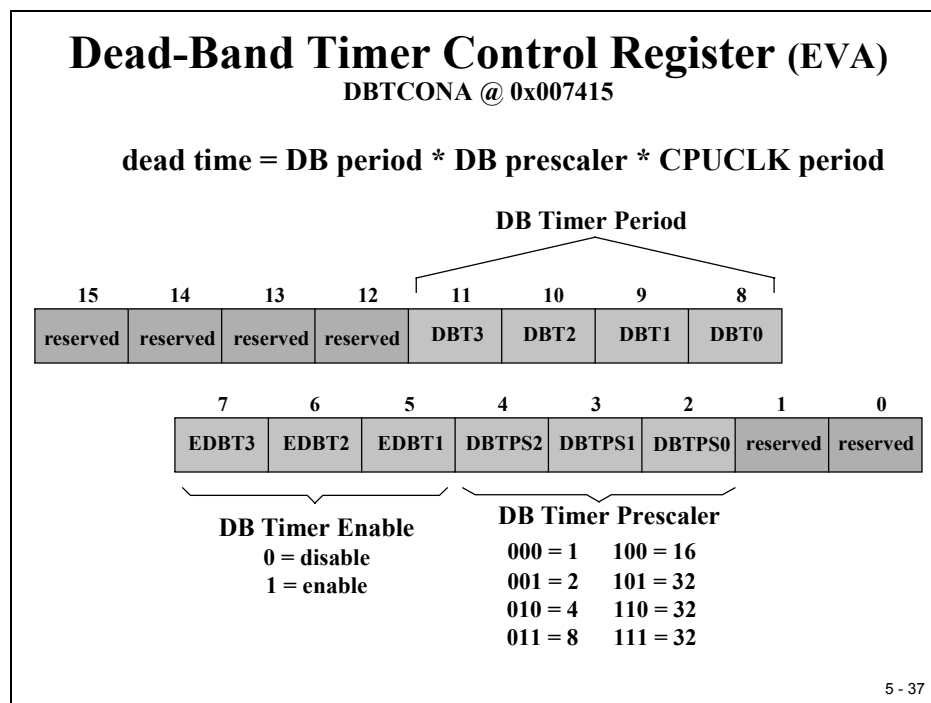
Two basic approaches exist for controlling shoot-through: modify the transistors, or modify the PWM gate signals controlling the transistors. In the first case, the switch-on time of the transistor gate must be increased so that it (slightly) exceeds the switch-off time.

The hard way to accomplish this is by adding a cluster of passive components such as resistors and diodes in series with the transistor gate to act as low-pass filter to implement the delay.

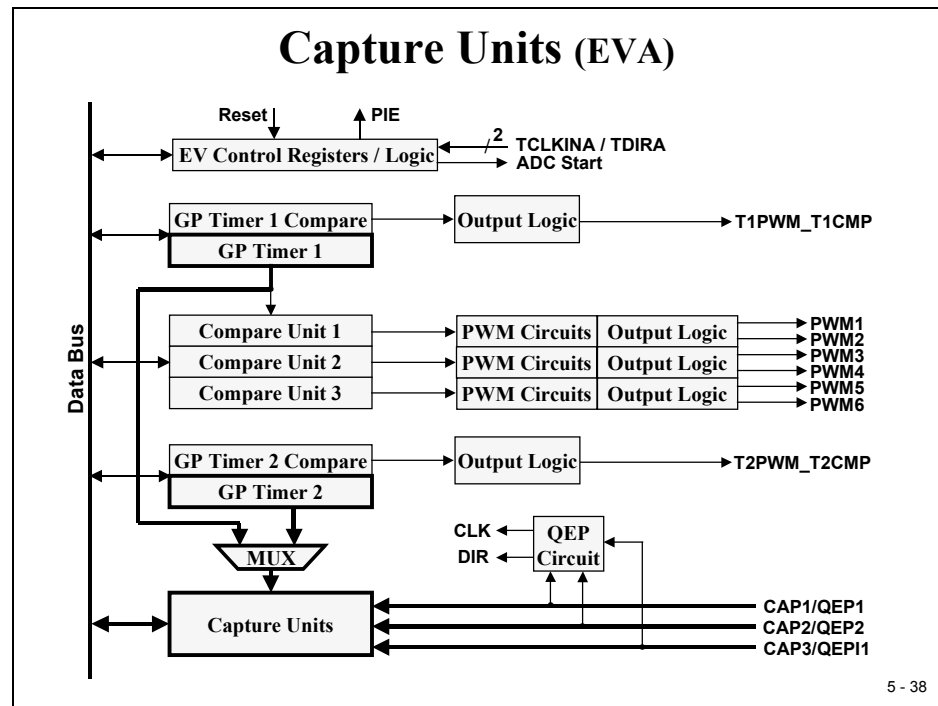
The second approach to shoot-through control separates transitions on complimentary PWM signals with a fixed period of time. This is called dead-band. While it is possible to perform software implementation of dead-band, the C28x offers on-chip hardware for this purpose that requires no additional CPU overhead. Compared to the passive approach, dead-band offers more precise control of gate timing requirements.



Each compare unit has a dead-band timer, but shares the clock prescaler unit and the dead-band period with the other compare units. Dead-band can be individually enabled for each unit.



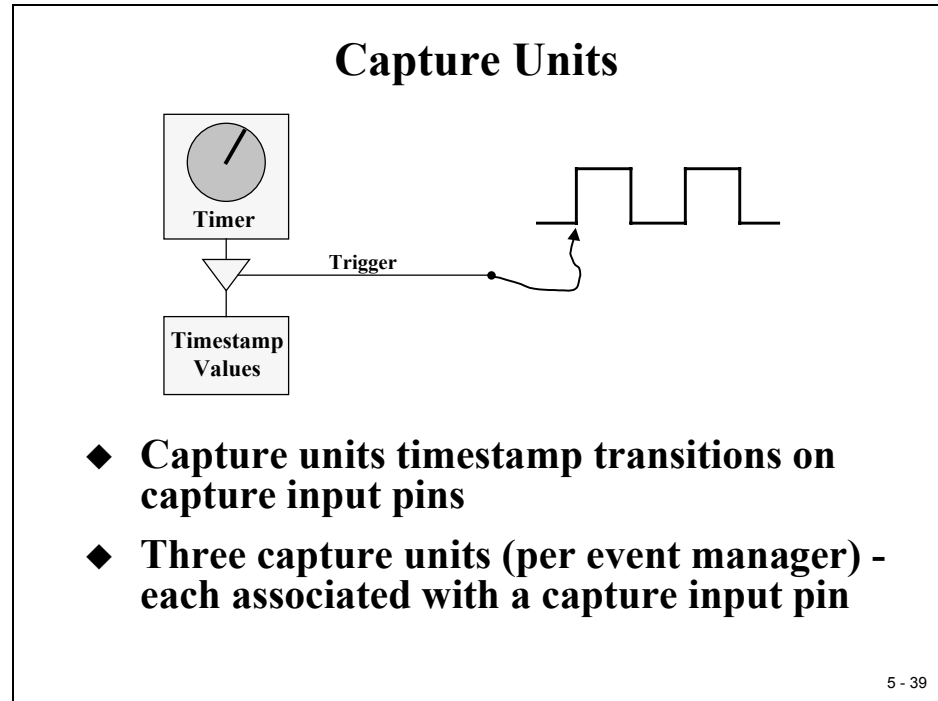
Capture Units



The capture units allow time-based logging of external logic level signal transitions on the capture input pins.

Event Manager A has three capture units, and each is associated with a capture input pin. The time base is selectable to be either GP timer 1 or 2. The timer value is captured and stored in the corresponding 2-level-deep FIFO stack when a specified transition is detected on a capture input pin.

Capture Unit 3 can be configured to trigger an A/D conversion that is synchronized with an external signal of a capture event.



Three potential uses for the Capture Units are:

- Measurement of the width of a pulse or a digital signal
- Automatic start of the AD – Converter by a Capture Event from CAP3
- Low speed estimation of a rotating shaft. A potential advantage for low speed estimation is given when we use “time capture” (16Bit resolution) instead of position pulse counting (poor resolution in slow mode).

Some Uses for the Capture Units

- ◆ Synchronized ADC start with capture event
- ◆ Measure the time width of a pulse
- ◆ Low speed velocity estimation from incr. encoder:

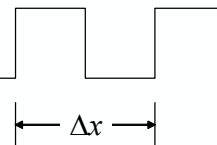
Problem: At low speeds, calculation of speed based on a measured position change at fixed time intervals produces large estimate errors

$$v_k \approx \frac{x_k - x_{k-1}}{\Delta t}$$

Alternative: Estimate the speed using a measured time interval at fixed position intervals

$$v_k \approx \frac{\Delta x}{t_k - t_{k-1}}$$

Signal from one
Quadrature
Encoder Channel

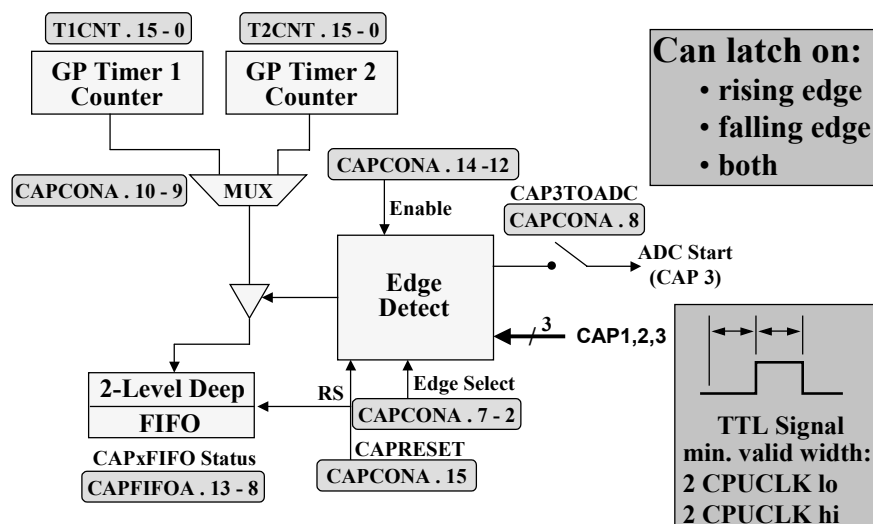


5 - 40

Capture Units Block Diagram

The edge detector stores the current value of the time base counter into a FIFO-buffer.

Capture Units Block Diagram (EVA)



5 - 41

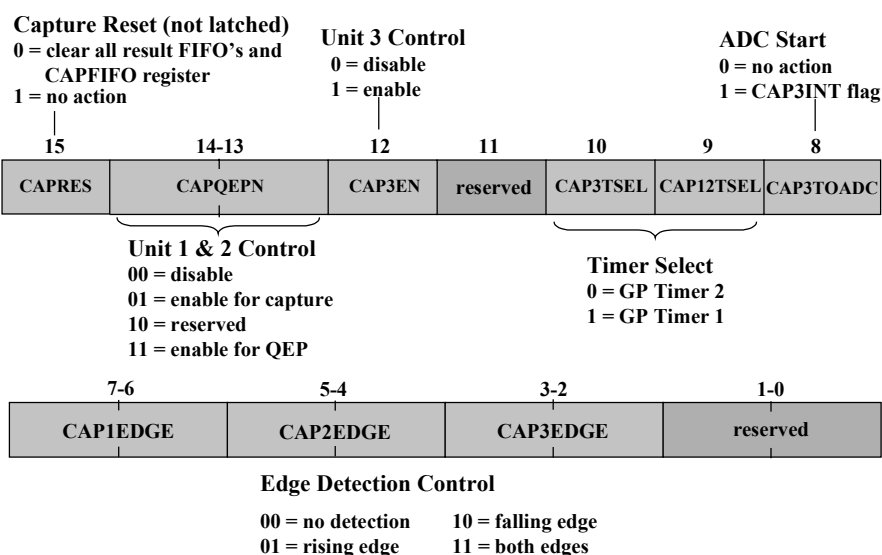
Capture Units Registers

Capture Units Registers		
Register	Address	Description
EVA	CAPCONA	0x007420 Capture Control Register A
	CAPFIFOA	0x007422 Capture FIFO Status Register A
	CAP1FIFO	0x007423 Two-Level Deep FIFO 1 Stack
	CAP2FIFO	0x007424 Two-Level Deep FIFO 2 Stack
	CAP3FIFO	0x007425 Two-Level Deep FIFO 3 Stack
	CAP1FBOT	0x007427 Bottom Register of FIFO 1
EVB	CAP2FBOT	0x007428 Bottom Register of FIFO 2
	CAP3FBOT	0x007429 Bottom Register of FIFO 3
	CAPCONB	0x007520 Capture Control Register B
	CAPFIOB	0x007522 Capture FIFO Status Register B
	CAP4FIFO	0x007523 Two-Level Deep FIFO 4 Stack
	CAP5FIFO	0x007524 Two-Level Deep FIFO 5 Stack
	CAP6FIFO	0x007525 Two-Level Deep FIFO 6 Stack
	CAP4FBOT	0x007527 Bottom Register of FIFO 4
	CAP5FBOT	0x007528 Bottom Register of FIFO 5
	CAP6FBOT	0x007529 Bottom Register of FIFO 6
EXTCONA 0x007409 / EXTCONB 0x007509 ;Ext. Cntrl Reg.		

5 - 42

Capture Control Register (EVA)

CAPCONA @ 0x007420



5 - 43

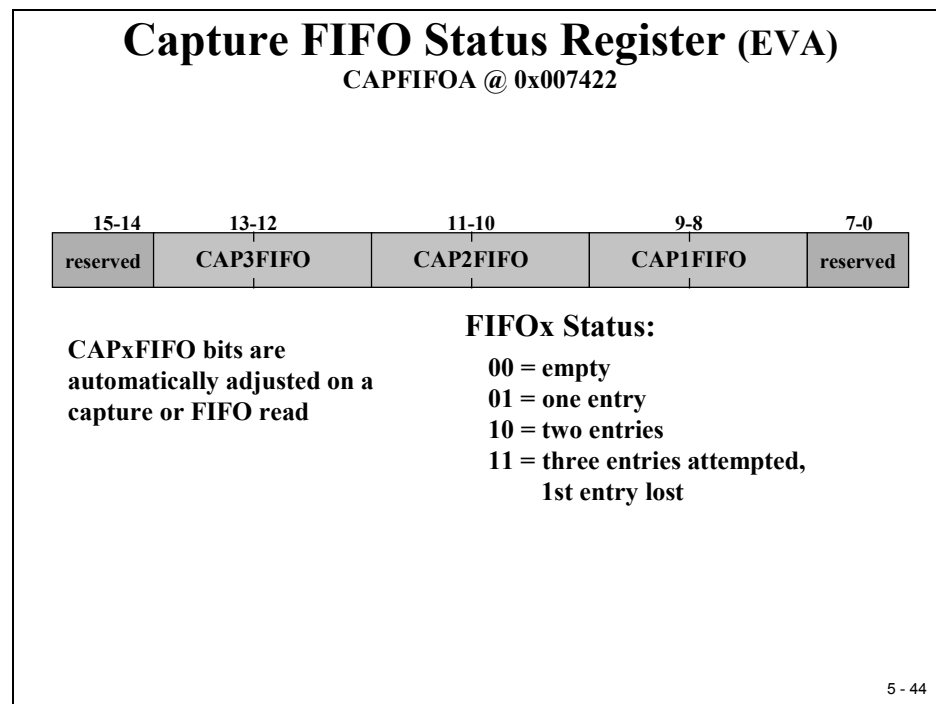
CAPCONA [15] is a reset bit for the Capture state machine and the status of the FIFO. It should be used in a single instruction to reset the Capture units during initialization. Note: to execute reset you will have to apply a zero!

With CAPCONA [14-12] the Capture Units are enabled. Please note that CAP1 and CAP2 are enabled jointly, whereas CAP3 has its own enable bit.

CAPCONA [10-9] are used to select the clock base for the capture units. Again, for CAP1 and CAP2 we have to select the same GP timer.

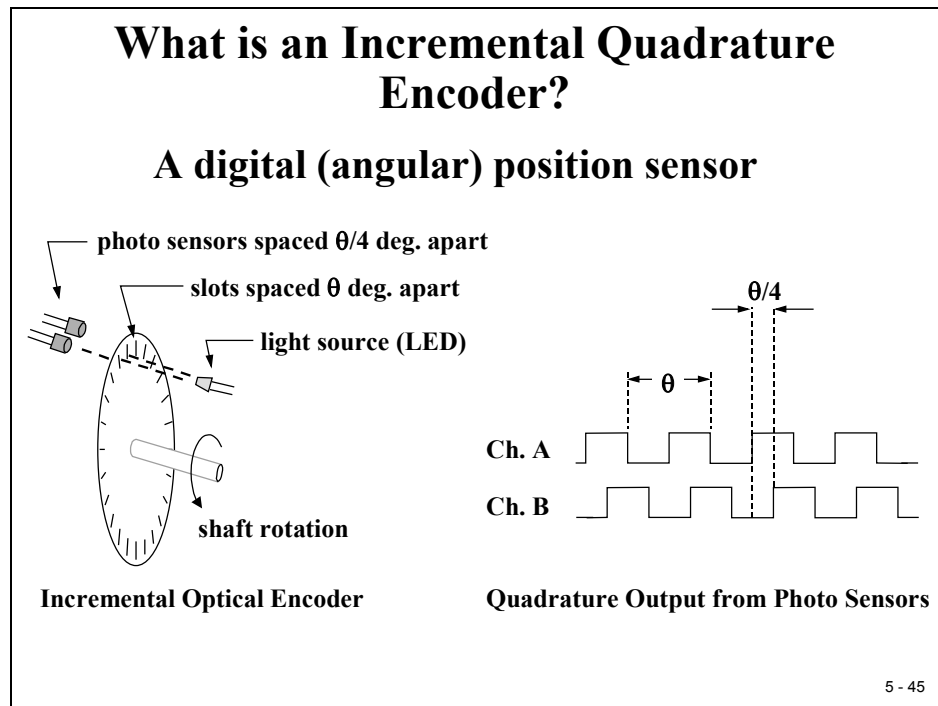
CAPCONA [8] allows CAP3 to start an AD conversion. Of course, before we use this option, we have to initialize the ADC. This will be explained in the next chapter.

CAPCONA [7-2] specify if the capture units are triggered with a rising or falling edge or with both edges.

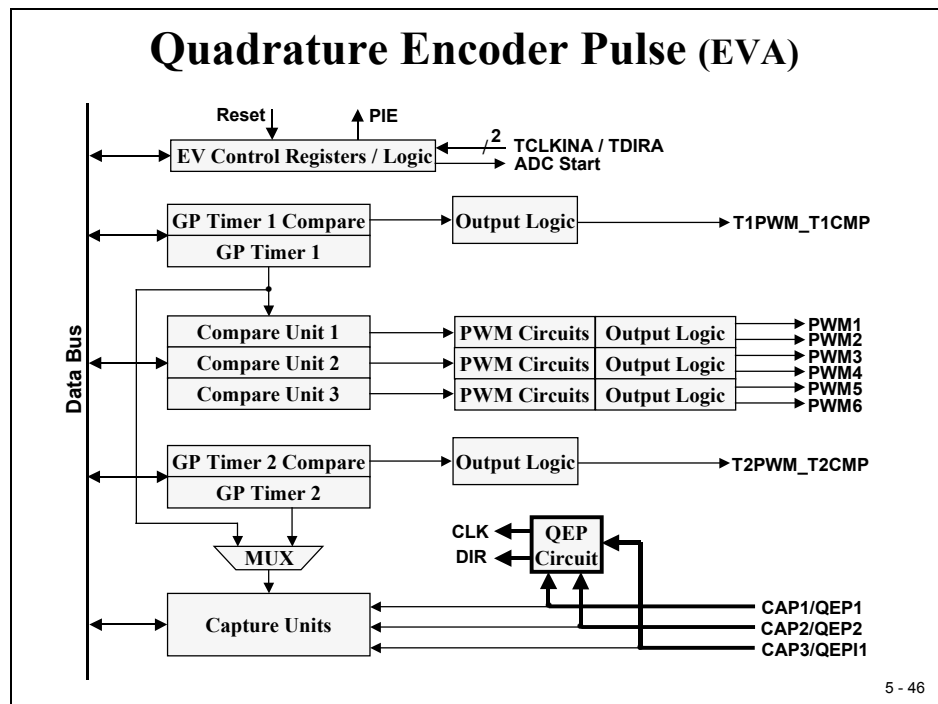


Register CAPFIFOA reflects the filling status of the three result register FIFO's. In case of an overflow the oldest entry will be lost. This principle ensures that a capture unit stores the two latest measurement results. If our program performs a read access to one of the FIFO result registers the status value in the corresponding CAPFIFOA bit field is decremented.

Quadrature Encoder Pulse Unit (QEP)



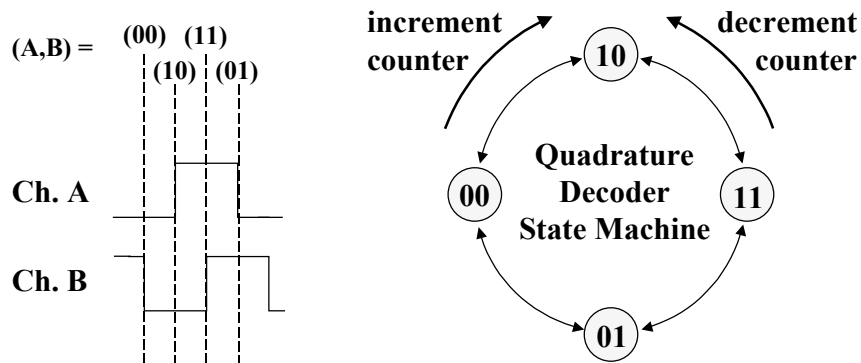
A QEP unit is normally used to derive direction and speed information from an incremental encoder circuit mounted on a rotating shaft. As shown on the previous slide two sensor signals are used to generate two digital pulse streams “Channel A” and “Channel B”.



The time relationship between A and B lead to a state machine with four states. Depending on the sequence of states and the speed of alternation, the QEP unit timer is decremented or incremented. By reading and comparing this timer counter information at fixed intervals, we can obtain speed and/or position information.

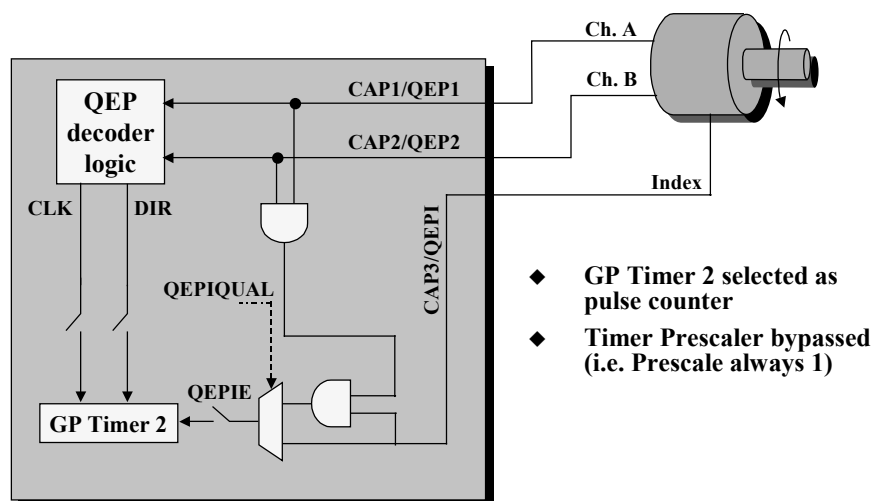
How is Position Determined from Quadrature Signals?

Position resolution is $\theta/4$ degrees.

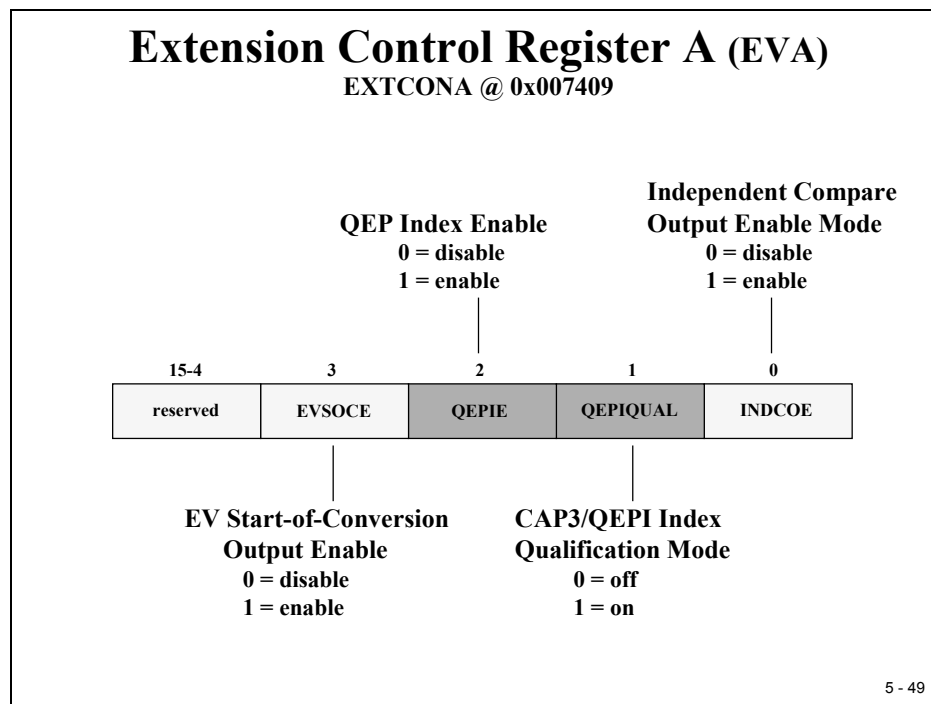


5 - 47

Incremental Encoder Connections (EVA)



5 - 48



The third capture input pin “QEPI1” can be used as an absolute position information signal for a zero degree crankshaft position. This signal is then used to reset the QEP timer to its initial state. To enable the “QEPI1” index function we have set EXTCONA [2] to 1. Then we have two more options, selected with EXTCONA [1]:

- Use index pulse “QEPI1” independent from the state of QEP1 and QEP2
- Use index pulse “QEPI1” as a valid trigger pulse only if this event is qualified by the state of QEP1 = 1 AND QEP2 = 1.

Lab 5A: Generate a PWM sine wave

Objective

So far we generated square wave signals to drive a loudspeaker. To ask a musician to listen to this type of music would be impudent. So let's try to improve the shape of our output signals. A musical note is a pure - or harmonic - sine wave signal of a fixed frequency. The objective of this lab exercise is to generate a harmonic sine wave signal out of a series of pulse width modulated digital pulses (PWM).

Remark: The first generation of cell phones used the square wave technology to generate ringing sounds. Compare this with today's latest cell phones!

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab5A.pjt** in E:\C281x\Labs.
2. Open the file Lab5.c from E:\C281x\Labs\Lab5 and save it as Lab5A.c in E:\C281x\Labs\Lab5A.
3. Add the source code file to your project:
 - **Lab5A.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:\tidcs\c28\dsp281x\v100\DSP281x_common\source add to project:
 - **DSP281x_PieCtrl.c**
 - **DSP281x_PieVect.c**
 - **DSP281x_DefaultIsr.c**From C:\ti\c2000\cgtoolslib add:

- **rts2800_ml.lib**

Project Build Options

5. 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; C:\tidcs\C28\IQmath\clIQmath\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 Source Code

7. Open Lab5A.c to edit: double click on “Lab5A.c” inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the CPU core timer 0 in this exercise; therefore we do not need the prototype of interrupt service routine “cpu_timer0_isr()”. Instead, we need a new ISR for GP Timer1-Compare-Interrupt. Add a new prototype interrupt function: “interrupt void T1_Compare_isr(void)”.
8. We do not need the variables “i”, “time_stamp” and frequency[8]” from Lab5 - delete their definition lines at the beginning of function “main”.
9. Next, modify the re-map lines for the PIE entry. Instead of “PieVectTable.TINT0 = & cpu_timer0_isr” we need to re-map:

PieVectTable.T1CINT = &T1_Compare_isr;

10. Delete the next two function calls: “InitCpuTimers();” and “ConfigCpuTimer(&CpuTimer0, 150, 50000);” and add an instruction to enable the EVA – GP Timer1 – Compare interrupt. Recall Module 4 “Interrupt System” and verify that the EVA – GP Timer1 – Compare interrupt is connected to PIE Group2 Interrupt 5. Which Register do we have to initialize? Answer:

PieCtrlRegs.PIEIER2.bit.INTx5 = 1;

Also modify the set up for register IER into:

IER = 2;

11. Next we have to initialize the Event Manager Timer 1 to produce a PWM signal. This involves the registers “GPTCONA”, “T1CON”, “T1CMPR” and “T1PR”.

For register “GPTCONA” it is recommended to use the bit-member of this predefined union to set bit “TCMPOE” to 1 and bit field “T1PIN” to “active low”.

For register “T1CON” set

- The “TMODE”-field to “counting up mode”;
- Field “TPS” to “divide by 1”;
- Bit “TENABLE” to “**disable timer**”;
- Field “TCLKS” to “internal clock”
- Field “TCLD” to “reload on underflow”
- Bit “TECMPR” to “enable compare operation”

12. Remove the 3 lines before the while(1)-loop in main:

- “CpuTimer0Regs.TCR.bit.TSS = 0;”
- “i = 0;”
- “time_stamp = 0;”

and add 4 new lines to initialise T1PR, T1CMPR, to enable GP Timer1 Compare interrupt and to start GP Timer 1:

EvaRegs.T1PR = 1500;

EvaRegs.T1CMPR = EvaRegs.T1PR/2;

EvaRegs.EVAIMRA.bit.T1CINT = 1;

EvaRegs.T1CON.bit.TENABLE = 1;

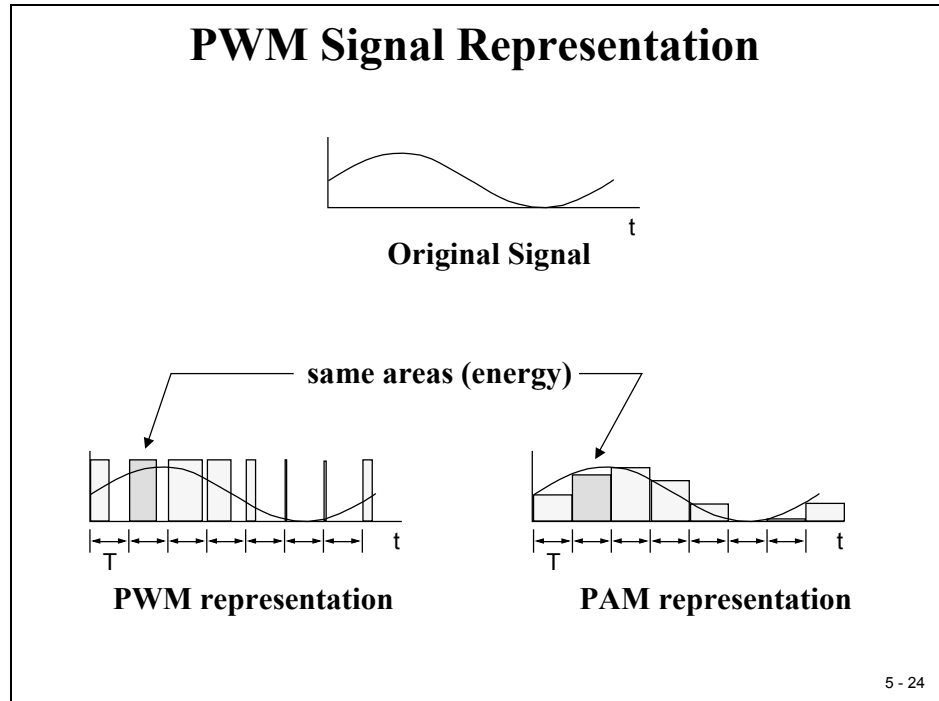
What is this number 1500 for? Well, it defines the length of a PWM period:

$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP}$$

with $TPS_{T1}=1$, $HISCP = 2$, $f_{CPU} = 150\text{MHz}$ and a desired $f_{PWM} = 50\text{kHz}$ we derive: $T1PR = 1500$!

T1CMPR is preloaded with half of T1PR. Why’s that? Well, in general T1CMPR defines the width of the PWM-pulse. Our start-up value obviously defines a pulse width of 50%.

Recall slide 5-24: "PWM-Representation"



A duty cycle of 50% represents a sine angle of 0 degrees! And, it makes sense to initialize the PWM unit for this angle. From the bottom left of the slide we can derive:

<u>Degree</u>	<u>Sin</u>	<u>Duty – Cycle</u>
0°	0	50%
90°	1	100%
180°	0	50%
270°	-1	0 %
360°	0	50%

13. Modify the endless while(1) loop of main! We will perform all activities using GP Timer 1 Compare Interrupt Service. Therefore we can delete almost all lines of this main background loop, we only have to keep the watchdog service:

```
while(1)
{
    EALLOW;
    SysCtrlRegs.WDKEY = 0xAA;
    EDIS;
}
```

14. Rename the interrupt service routine “cpu_timer0_isr” into “T1_Compare_isr”. Remove the line “CpuTimer0.InterruptCount++;” and replace the last line of this routine by:

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP2;
```

Before this line add another one to acknowledge the GP Timer 1 Compare Interrupt Service is done. Remember how? The Event Manager has 3 interrupt flag registers “EVAIFRA”, “EVAIFRB” and “EVAIFRC”. We have to clear the T1CINT bit (done by setting of the bit):

```
EvaRegs.EVAIFRA.bit.T1CINT = 1;
```

Build and Load

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

Project → Build

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

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

File → Load Program and choose the desired output file.

Test

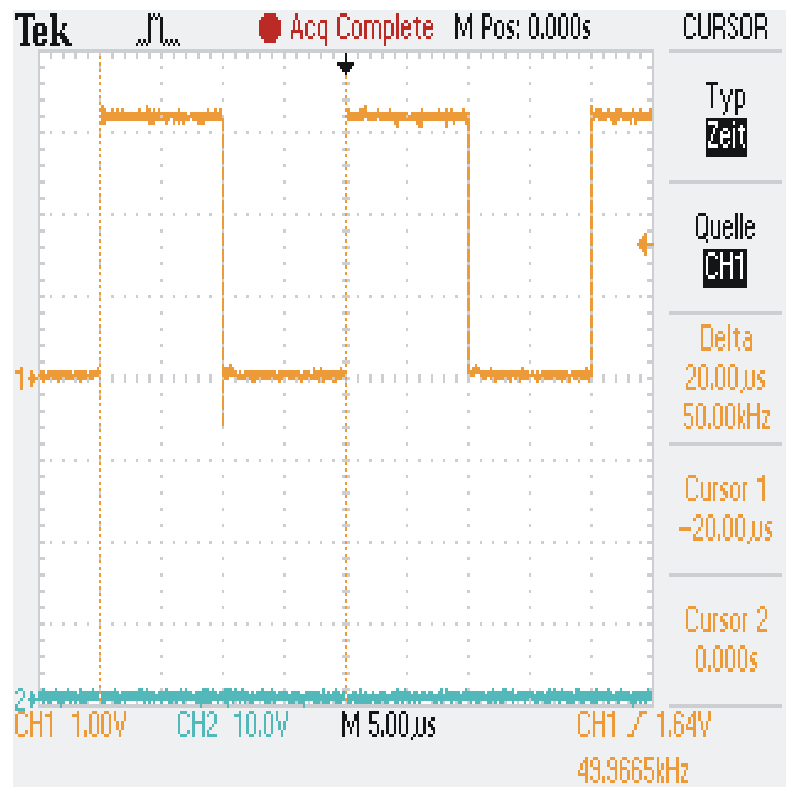
17. Reset the DSP by clicking on:

Debug → Reset CPU	followed by
Debug → Restart	and
Debug → Go main.	

18. When you now run the code the DSP should generate a 50 kHz PWM signal with a duty cycle of 50% on T1PWM. If you have an oscilloscope you can use jumper JP7 (in front of the loudspeaker) of the Zwickau Adapter board to measure the signal.

If your laboratory can't provide a scope, you can set a breakpoint into the interrupt service routine of T1 Compare at line "PieCtrlRegs.PIEACK.all = PIEACK_GROUP2; Verify that your breakpoint is hit periodically, that register T1PR holds 1500 and register T1CMPR is initialized with 750. Use the watch window to do so.

Do not continue with the next steps until this point is reached successfully! Instead go back and try to find out, what went wrong during the modification of your source code.



Generate the sine wave

So far we generated a pure square wave PWM signal of 50 KHz. Our goal is to produce a sine wave signal which is build up from a series of this 50 KHz carrier period cycles. We have to modify the pulse width according to the current sine angle. Obviously we have to increment the angle from 0° to 360° in a couple of steps.

Question is: how do we calculate the sine-values?

- **Use the $\sin(x)$ function**

This function is part of the C compilers “math.lib”. All we would need to do is to add the header file “math.h” to our project. Problem: $\sin(x)$ is a floating-point function; our DSP is a fixed point processor. That means the compiler has to generate quite a lot of assembler instructions to calculate the sine values. This will cost us a quite a lot of CPU time, just to calculate the same series of sine values over and over.

➔ Feasible, but not recommended

- **Use a lookup table with pre calculated sine values**

We do not need a floating-point precision; our goal is to adjust the 16-bit register “T1CMPR”. It is much quicker to prepare an array with precalculated sine values. Instead of calculating the next value during runtime we access a table with pre-defined results of sine calculations. This principle is called “Lookup Table Access” and is widely used in embedded control. Almost all control units for automotive electronics are using one or more of these lookup tables, not only for trigonometric functions but also for control parameters.

➔ Highly recommended

Next Question:

How do we generate a lookup table? Well, use a calculator, note all results and type them into an array! How many values? Well, the more values we have the better we can approximate the analogue sine wave shape! Recall, we need sine values from 0° to 360° . Sounds like a lot of boring work, doesn't it?

Answer: Texas Instruments has already done the work for you. The C28x – DSP comes with a “BOOT-ROM” (see memory map – module 1). A part of this memory area is a sine wave table!

From Address 0x3F F000 to 0x3F F3FF we find 512 values for $\sin(x)$. The numbers are stored as 32 Bit – numbers in “Q30”-notation. With 512 entries we have an angle step of 0.703° ($360^\circ/512$) for a unit circle.

But what is “Q30”?

“Q30” or “I2Q30” is a fractional fixed-point representation of 32-Bit numbers. We will discuss the advantage of fractional numbers for embedded control in detail in Part 2 of this DSP course. So far, let’s try to understand the basics:

The data format “Q30” separates a 32-bit number into an integer part and a fractional part. The integer part is the usual sequence of positive powers of 2; the fractional part is the sequence of negative powers of 2. For a “Q30” number we get the following binary representation:

Bit 31	Bit 30	Bit 29	Bit 28	Bit 27	Bit 26	Bit 25
$(-1)*2^1$	$1*2^0$	$1*2^{-1}$	$1*2^{-2}$	$1*2^{-3}$	$1*2^{-4}$	$1*2^{-5}$

The decimal range of a “Q30” number is $-2\dots+1.9999$:

Most negative number:	0x8000 0000	-2
Decimal minus 1:	0xC000 0000	-1
Smallest negative number	0xFFFF FFFF	-9.31322e-10
Zero	0x0000 0000	0
Smallest positive number	0x0000 0001	+9.31322e-10
Decimal plus 1:	0x4000 0000	+1
Most positive number:	0x7FFF FFFF:	+1.999999999

IQ-Math Library

Texas Instruments has built a whole library of fixed-point math operations based on this “Q”-format. This library called “IQ-Math” is widely used in closed control applications like digital motor control, FAST FOURIER TRANSFORM (FFT) or digital filters (FIR, IIR). The library is free, no royalties and can be downloaded from TI’s web. The appendix of this CD contains the current version of IQ-Math. We will discuss and use this library in a specific module in Part 2 of this DSP course.

Boot ROM Table

The following table shows the contents of the sine wave area of the Boot ROM:

Address	Low word	High Word	Angle in °	Sine value in IQ30
0x3F F000	0x0000	0x0000	0	0.0
0x3F F002	0x0E90	0x00C9	0.703	0.01227153838
0x3F F004	0x155F	0x0192	1.406	0.02454122808
0x3F F006	0x0CAF	0x025B	2.109	0.03680722322
...				
0x3F F100	0x0000	0x4000	90	1.0
....				
0x3F F200	0x0000	0x0000	180	0.0
....				
0x3F F300	0x0000	0xC000	270	-1.0
....				
0x3F F3FE	0xF170	0xFF36	359,3	-0.01227153838

Resume Lab Exercise 5A

Let's resume the procedure for Lab5A:

- How do we get access to this boot ROM sine wave table?

We have to add some basic support from the "IQ-Math" library to our project. At the top of your code, just after the line "#include "DSP281x_Device.h" add the next lines:

```
#include "IQmathLib.h"

#pragma DATA_SECTION(sine_table, "IQmathTables");

_iq30 sine_table[512];
```

The #pragma statement declares a specific data memory area, called “IQmathTables”. This area will be linked in the next procedure step to the address range of the Boot ROM sine table. The global variable “sine_table[512]” is an array of this new data type “IQ30”.

20. Add an additional Linker Command file to your project. From E:\C281x\Labs\Lab5A add:

Lab5A.cmd

Open and inspect this file. You will see that we add just one entry for the physical memory location (“ROM”) in data page 1 and that we connect the memory area “IQmathTables” to address “ROM”. The attribute “NOLOAD” assures that the debugger will not try to download this area into the DSP when we load the program – because it is already there, it is ROM – read only memory.

21. Modify “T1_Compare_isr()”

This interrupt service routine is a good point to modify the pulse width of the PWM signal. Recall, we do have now a global array “sine_table[512]” that holds all the sine values we need for our calculation.

Now we have to do a little bit of math’s.

What is the relationship between this sine value and the value of T1CMPR?

Answer:

(1) We know that the difference between T1PR and T1CMPR defines the pulse width of the current PWM period. So the goal is to calculate a new value for T1CMPR.

(2) Next, we have to take into account that the sine table delivers signed values between +1 and –1. Therefore we have to add an offset of +1.0 to this value.

(3) This shifted sine value has to be multiplied with T1PR/2.

Summary:

$$T1CMPR = T1PR - \left((\sin e_table[index] + 1.0) * \frac{T1PR}{2} \right)$$

To code this into the “IQ-Math” form we use:

$$T1CMPR = T1PR - _IQ30mpy(\sin e_table[index] + _IQ30(0.9999), T1PR/2)$$

“_IQ30mpy(a,b)” is an intrinsic function call to do a multiplication in IQ30-Format. The value from sine_table is already in IQ30-format, whereas the constant 1.0 has to be translated into it by function call “_IQ30(0.9999)”.

To avoid fixed-point overflows we can embed the calculation into a saturation function “_IQsat(x,max,min)” to limit the result between T1PR and 0. This leads to our final instruction:

```
EvaRegs.T1CMPR =
    EvaRegs.T1PR - _IQsat(
        _IQmpy( sine_table[index] + _IQ30(0.9999), EvaRegs.T1PR/2),
        EvaRegs.T1PR,0) ;
```

Add this line just after the “EDIS” instruction that follows the service of the Watchdog timer.

22. Setup the sine wave frequency.

Recall that the Boot ROM sine table consists of 512 entries for a unit circle. The frequency of the sine wave is given by:

$$f_{SIN} = \frac{f_{PWM}}{\text{Number of PWM - periods per } 360^\circ}$$

For example, if we use all 512 entries of the Boot ROM table we get:

$$f_{SIN} = \frac{50KHz}{512} = 97,6Hz$$

If we use only one lookup table entry out of 4, we end up with:

$$f_{SIN} = \frac{50KHz}{128} = 390,6Hz$$

Let's use this last set up. It means we have to increment the index by 4 to make the next access to the lookup table. Add the next line after the update line for T1CMPR:

```
index += 4;
```

Do not forget to:

- (1) Reset variable index to 0 if it is increased above 511.
- (2) Declare the integer variable index to be static.

Build and Load

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

Project → Build

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

24. Load and test the final version of the output file as you've done before. With the help of the oscilloscope we should see now a change of the pulse width of the PWM signal 'on the fly'.
25. Optional: Low pass filter:

The low pass filter capacity of the tiny loudspeaker is not strong enough to integrate the pulse sequence to a sine wave shaped signal. We can improve this by adding a simple low-pass filter between the two connectors of jumper JP7-2 (DSP-T1PWM) and JP7-1 (Loudspeaker). Build a passive low pass filter of first order with a frequency of 25 KHz:

$$f_{Filter} = \frac{1}{2 \cdot \pi \cdot R \cdot C} = 25 KHz$$

Optional Exercise

How about other frequencies?

In the previous exercise we generated a modulated sine wave of 390 Hz. We used the 512-point look-up table and stepped through it using an increment of 4.

How do we generate other frequencies?

Answer: when we change the step size for variable "index" we can generate more (or less PWM-periods per 360°. More means we slow down the sine frequency, less means we increase the sine wave frequency. The following table shows the different sine wave frequencies for a PWM carrier frequency of 50 KHz and a lookup table of 512 points per 360°:

$$f_{SIN} = \frac{f_{PWM}}{\text{Number of PWM - periods per } 360^\circ}$$

Incremental step of "index"	Number of PWM periods per 360°	Sine wave frequency In Hz
1	512	97.6
2	256	195.3
3	171	293
4	128	390
5	102	488
10	51	976
15	34	1,460
20	26	1,950
50	10	4,880

We can't increase the step size much above 50 because this gives us only 10 points per 360° to synthesize the sine wave.

What about frequencies that we do not match with any of these incremental steps? Recall our Lab Exercise 5 with the range of 8 notes; it started with a note of 264Hz.

How do we generate a sine wave of 264 Hz?

The answer is: We have to modify the PWM frequency itself. So far we did all experiments with a fixed PWM signal of 50 KHz. Let's fix now the number of points taken out of the look up table to 128 (that is an index increment by 4). To get a sine wave of 264Hz we calculate:

$$f_{SIN} = \frac{f_{PWM}}{128} = 264Hz$$

$$f_{PWM} = 264Hz * 128 = 33,792KHz$$

To setup a PWM signal of 33,792 KHz we have to re-calculate T1PR:

$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP}$$

$$T1PR = \frac{150MHz}{33.792KHz \cdot 1 \cdot 2} = 2,219.46$$

T1PR has to be loaded with an integer value, so we have to round the result to 2219.

Test:
$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP} = \frac{150MHz}{2219 * 1 * 2} = 33.799KHz$$

$$f_{SIN} = \frac{f_{PWM}}{128} = \frac{33.799KHz}{128} = 264.05Hz$$

That's a reasonable result; the intended frequency of 264Hz is missed by an error of 0.02%.

26. Try to setup your code to generate a sine wave of 264Hz!
27. If you have additional time in your laboratory try to improve Lab5 to generate all 8 notes with sine wave modulated PWM's!

End of Lab 5A

This page was intentionally left blank.

C28x Analogue Digital Converter

Introduction

One of the most important peripheral units of an embedded controller is the Analogue to Digital Converter (ADC). This unit provides an important interface between the controller and the real world. Most physical signals such as temperature, humidity, pressure, current, speed and acceleration are analogue signals. Almost all of these can be represented as an electrical voltage between V_{\min} and V_{\max} , e.g. 0...3V, which is proportional to the original signal. The purpose of the ADC is to convert this analogue voltage in a digital number. The relationship between the analogue input voltage (V_{in}), the number of binary digits to represent the digital number (n) and the digital number (D) is given by:

$$V_{in} = \frac{D * (V_{REF+} - V_{REF-})}{2^n - 1} + V_{REF-}$$

V_{REF+} and V_{REF-} are reference voltages and are used to limit the analogue voltage range. Any input voltage beyond these reference voltages will deliver a saturated digital number. NOTE: Of course all voltages must stay inside the limits of the maximum ratings according to the data sheet.

In case of the C28x and especially with the eZdsp2812 and the Zwickau adapter board voltage V_{REF-} is set to 0V, V_{REF+} to +3.0V. The C28x internal ADC has a 12 Bit resolution ($n=12$) for the digital number D . This gives:

$$V_{in} = \frac{D * 3.0V}{4095}$$

Most applications require not only one analogue input signal to be converted into a digital value; their control loop usually needs several different sensor input signals. Therefore, the C28x is equipped with 16 dedicated input pins to measure analogue voltages. These 16 signals are multiplexed internally, that means they are processed sequentially. To do the conversion, the ADC has to make sure that during the conversion procedure there is no change of the analogue input voltage V_{in} . Otherwise the digital number would be totally wrong. An internal “sample and hold unit(s&h)” takes care of this. The C28x is equipped with two s&h-units, which can be used in parallel. This allows us to convert two input signals (e.g. two currents) at the same time.

But there's more: The C28x ADC has an “auto-sequencer” capability of 16 stages. That means that the ADC can automatically continue with the conversion of the next input channel after the previous ones are finished. Thanks to this enhancement we do not have to fetch the digital results in the middle of a measurement sequence, one single interrupt service routine call at the end of the sequence will do it.

Module Topics

C28x Analogue Digital Converter	6-1
<i>Introduction</i>	<i>6-1</i>
<i>Module Topics.....</i>	<i>6-2</i>
<i>ADC Module Overview</i>	<i>6-3</i>
<i>ADC in Cascaded Mode.....</i>	<i>6-4</i>
<i>ADC in Dual Sequencer Mode.....</i>	<i>6-5</i>
<i>ADC Conversion Time</i>	<i>6-6</i>
<i>ADC Register Block</i>	<i>6-7</i>
<i>Example: 3 phase measurement.....</i>	<i>6-12</i>
<i>ADC Result Register Set</i>	<i>6-13</i>
<i>Lab 6: Two Potentiometer Voltages.....</i>	<i>6-14</i>
<i>Lab 6A: Speed Control of 'Knight Rider'</i>	<i>6-22</i>

ADC Module Overview

Before we go into the details how to program the internal ADC let's summarize some details of the ADC Module. It was said that the digital resolution of the converted number is 12 bit. Assuming an input voltage range from 0...+3V we get a voltage resolution of $3.0V/4095 = 0.732mV$ per bit.

We have two s&h units, which can be used in parallel ("simultaneous sampling"). Each sample and hold is connected to 8 multiplexed input lines. The auto sequencer is a programmable state machine and is able to automatically convert up to 16 input signals. Each state of the auto sequencer puts a measurement into its own result register.

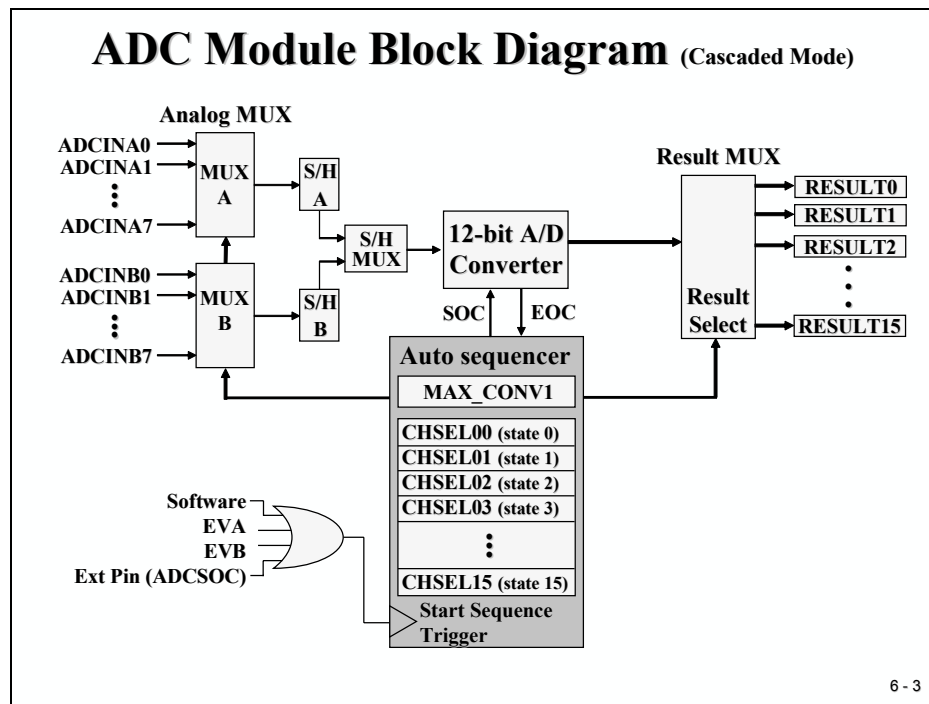
The fastest conversion time is 80ns per sample in a sequence and 160ns for the very first sample.

<p style="text-align: center;">ADC Module</p> <ul style="list-style-type: none"> ◆ 12-bit resolution ADC core ◆ Sixteen analog inputs (range of 0 to 3V) ◆ Two analog input multiplexers <ul style="list-style-type: none"> • Up to 8 analog input channels each ◆ Two sample/hold units (for each input mux) ◆ Sequential and simultaneous sampling modes ◆ Auto sequencing capability - up to 16 auto conversions <ul style="list-style-type: none"> • Two independent 8-state sequencers <ul style="list-style-type: none"> • "Dual-sequencer mode" • "Cascaded mode" ◆ Sixteen individually addressable result registers ◆ Multiple trigger sources for start-of-conversion <ul style="list-style-type: none"> • External trigger, S/W, and Event Manager events 	6 - 2
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------

A start of a conversion sequence can be initiated from four sources:

- By software - just set a start bit to 1
- By an external signal "ADCSOC"
- By an event (period, compare, underflow) of Event Manager A
- By an event (period, compare, underflow) of Event Manager B

ADC in Cascaded Mode



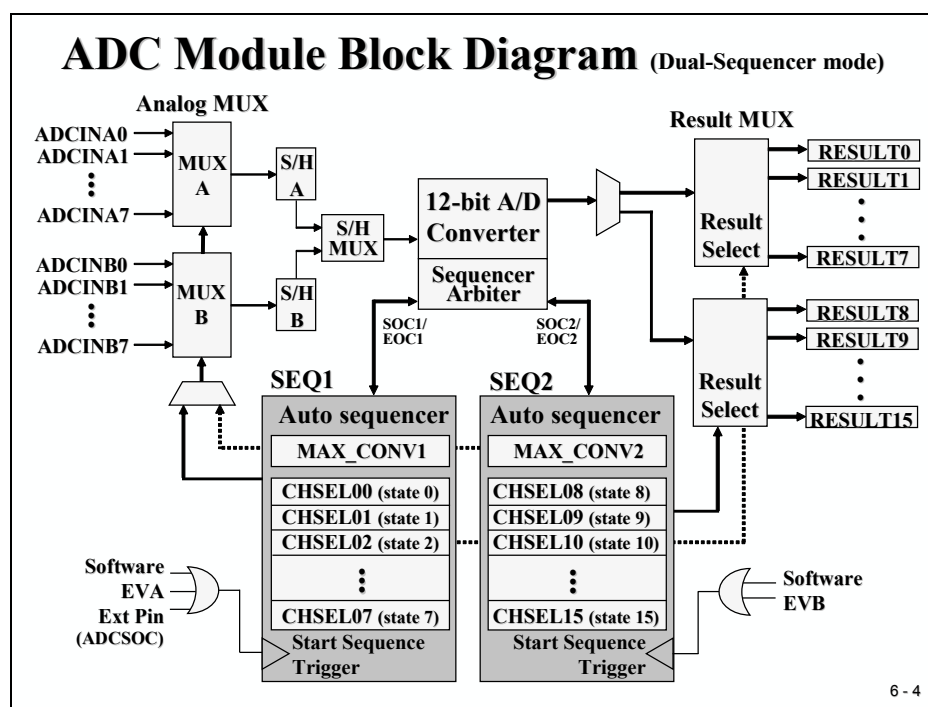
The slide shows the block diagram for the ADC operating in “cascaded mode”. One Auto sequencer controls the flow of the conversion. Before we can start a conversion, we have to setup the number of conversions (“MAX_CONV1”) and which input line should be converted in which stage (“CHSELxx”). The results are buffered in individual result registers (“RESULT0” to “RESULT15”) for every stage.

We can choose between two more options: “Simultaneous” and “Sequential” sampling. In the first case, both s&hs are used in parallel. Two input lines with the same input code (for example ADCINA3 and ADCINB3) are converted at the same time by stage CHSEL00. In “Sequential mode” the input lines can be connected to any of the states of the auto sequencer.

To trigger a conversion sequence we can use a software start by setting a particular bit. We also have three more start options using hardware events. Especially useful is the hard-wired output of a timer event, which leads to very precise sample periods. This is a necessity for correct operation of digital signal processing algorithms. No need to trigger an interrupt service (with its possible jitter due to interrupt response delays) to switch the input channel between subsequent conversions – the auto sequencer will do it.

We can use the ADC’s interrupt after the end of a sequence (or for some applications at the end of every other sequence) to read out the result register block.

ADC in Dual Sequencer Mode

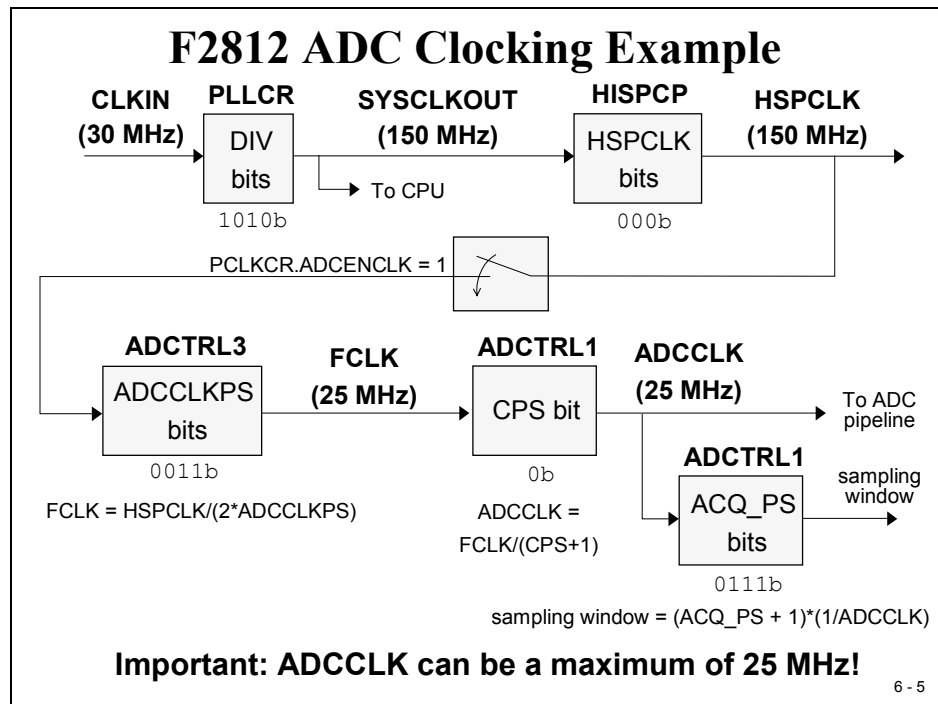


The second operating mode of the ADC “Dual Sequencer Mode” splits the auto sequencer into two independent state machines (“SEQ1” and “SEQ2”). This mode uses EVA as the hardware trigger for SEQ1 and EVB for SEQ2. To code the input channels for the individual states of the two sequencers we are free to select any of the 16 inputs for any of the 2x8 states. RESULT0 to RESULT7 cover the values from SEQ1 and RESULT8 to RESULT15 do it for SEQ2.

The reason for this split mode is to have two independent ADC’s, triggered by their own hardware timer units, GP Timer 1 and 2 for SEQ1 and GP Timer 3 and 4 for SEQ2.

In case of a simultaneous start of SEQ1 and SEQ2 the Sequencer Arbiter takes care of this situation. In this event SEQ1 has higher priority; the start of SEQ2 will be delayed after the end of SEQ1.

ADC Conversion Time



There are some limitations for the set-up of the ADC conversion time. First, the basic clock source for the ADC is the internal clock HSPCLK – we cannot use any clock speed we like. This clock is derived from the external oscillator, multiplied by PLLCR and divided by HISPCP. We discussed these bit fields in earlier modules; so just in case you do not recall their meanings, look back.

The second limitation is the maximum frequency for “FCLK” as the internal input signal for the ADC unit. At the moment this signal is limited to 25MHz. To adjust this clock we have to initialise the bit field “ADCCLKPS” accordingly. Bit “CPS” gives the option for another divider by 2. The clock “ADCCLK” is the time base for the internal processing pipeline of the ADC.

A third limitation is the sampling window controlled by the field “ACQ_PS”. This group of bits defines the length of the window that is used between the multiplexer switch and the time when we sample (or “freeze”) the input voltage. This time depends on the line impedance of the input signal. So it is hardware dependent - we can’t specify an optimal period for all applications. For our lab exercises in this chapter, it is a ‘don’t care’ because we sample DC-voltages taken from two potentiometers of the Zwickau adapter board.

ADC Register Block

Three control registers “ADCTRL1 to 3” are used to set-up one of the various operating conditions of the ADC unit. Register “ADCST” covers the current status of the ADC.

Analog-to-Digital Converter Registers

Register	Address	Description
ADCTRL1	0x007100	ADC Control Register 1
ADCTRL2	0x007101	ADC Control Register 2
ADCMAXCONV	0x007102	ADC Maximum Conversion Channels Register
ADCCHSELSEQ1	0x007103	ADC Channel Select Sequencing Control Register 1
ADCCHSELSEQ2	0x007104	ADC Channel Select Sequencing Control Register 2
ADCCHSELSEQ3	0x007105	ADC Channel Select Sequencing Control Register 3
ADCCHSELSEQ4	0x007106	ADC Channel Select Sequencing Control Register 4
ADCASEQSR	0x007107	ADC Auto sequence Status Register
ADCRESULT0	0x007108	ADC Conversion Result Buffer Register 0
ADCRESULT1	0x007109	ADC Conversion Result Buffer Register 1
ADCRESULT2	0x00710A	ADC Conversion Result Buffer Register 2
: :	: :	: : : :
ADCRESULT14	0x007116	ADC Conversion Result Buffer Register 14
ADCRESULT15	0x007117	ADC Conversion Result Buffer Register 15
ADCTRL3	0x007118	ADC Control Register 3
ADCST	0x007119	ADC Status and Flag Register

6 - 6

ADC Control Register 1 - Upper Byte

ADCTRL1 @ 0x007100

ADC Module Reset

0 = no effect

1 = reset (set back to 0 by ADC logic)

Acquisition Time Prescale (S/H)

Value = (binary+1)

* Time dependent on the “Conversion Clock Prescale” bit (Bit 7 “CPS”)

15	14	13	12	11	10	9	8
reserved	RESET	SUSMOD1	SUSMOD0	ACQ_PS3	ACQ_PS2	ACQ_PS1	ACQ_PS0

Emulation Suspend Mode

00 = [Mode 0] free run (do not stop)

01 = [Mode 1] stop after current sequence

10 = [Mode 2] stop after current conversion

11 = [Mode 3] stop immediately

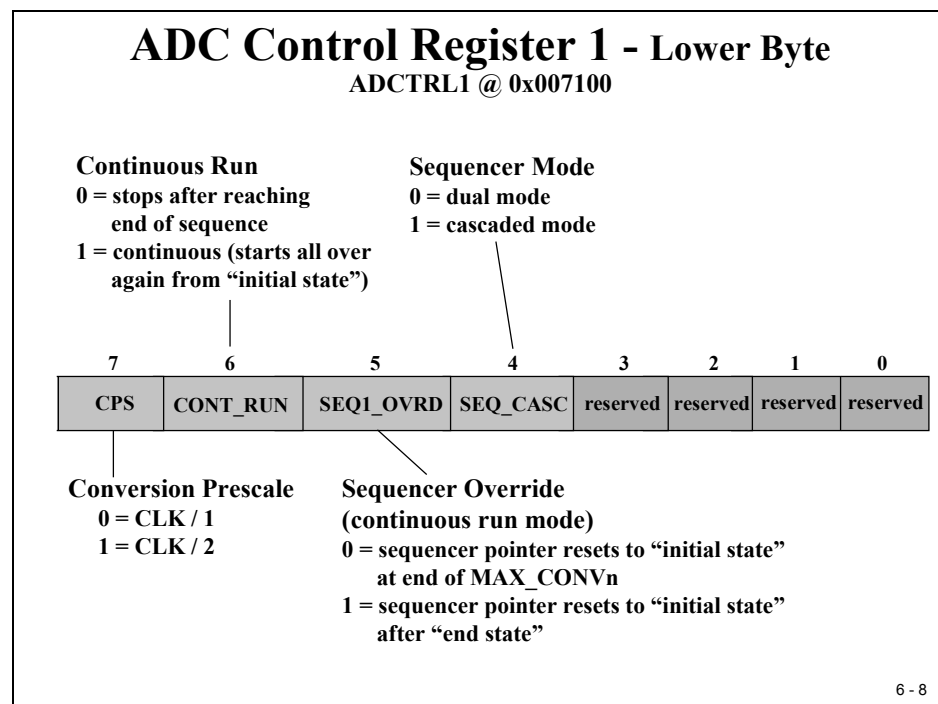
6 - 7

ADC Control Register 1

Bit 14 (“RESET”) can be used to reset the whole ADC unit into its initial state. It is always good practice to apply a RESET command before you initialise the ADC.

Bits 13 and 12 define the interaction between the ADC and an emulator command, similar to the behaviour that we discussed in the event manager module.

The next 4 bits define the length of the sample window.



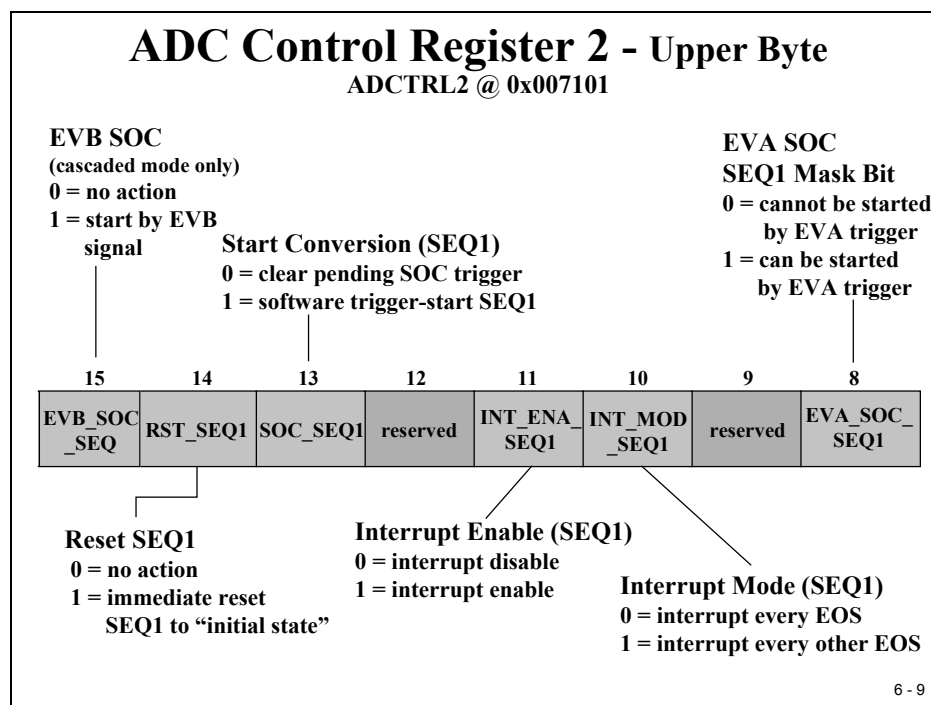
“CPS” is used to divide the input frequency by 1 or 2.

Bit 6 (“CONT_RUN”) defines if the auto sequencer starts at the end of a sequence (=0) and waits for another trigger or if the sequence should start all over again immediately (= 1).

Bit 5 (“SEQ1_OVRD”) defines two different options for continuous mode. We will not use this mode during our labs, so it is a ‘don’t care’.

Finally Bit 4 defines the Sequencer Mode to be a state machine of 16 (=1) or to operate as two independent state machines of 8 states.

ADC Control Register 2



The upper half of register ADCTRL2 is responsible to control the operating mode of sequencer 1.

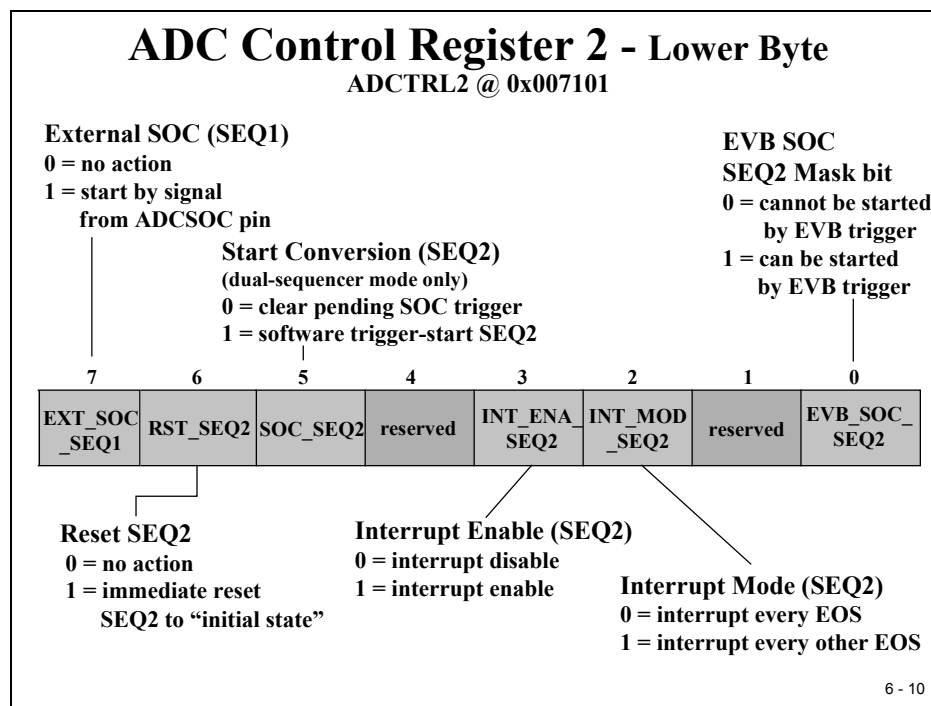
Bit 15 "EVB_SOC_SEQ" flags if Event Manager B has triggered the conversion. It is a 'read only' flag.

With Bit14 "RST_SEQ1" we can reset the state machine of SEQ1 to its initial state. That means that the next trigger will start again from CHSELSEQ1.

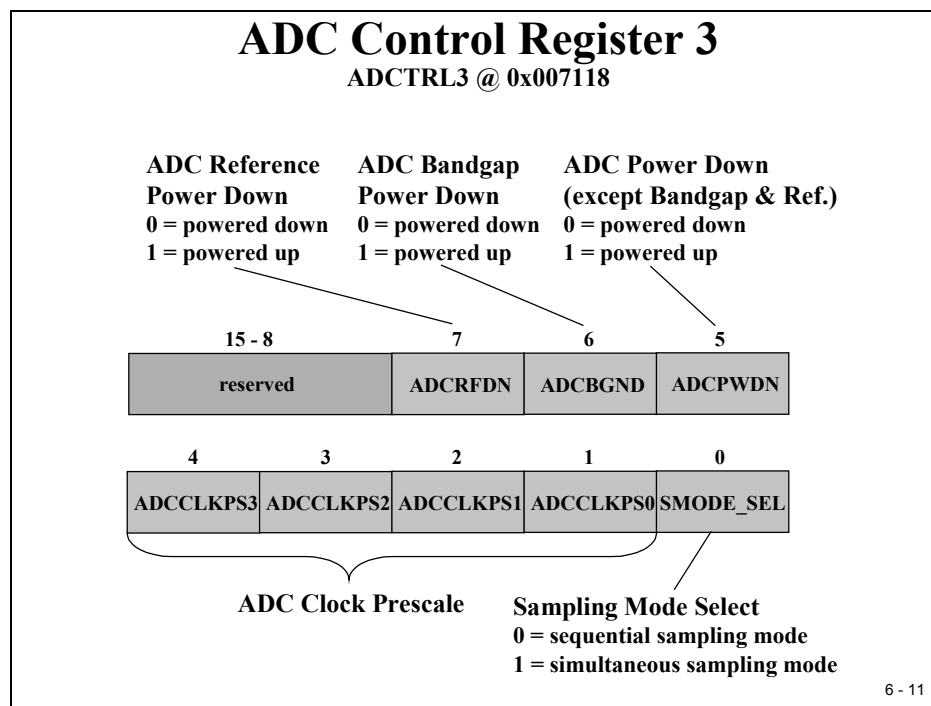
When we set Bit 13 "SOC_SEQ1" to 1 we perform a software start of the conversion.

Bits 11 and 10 define the interrupt mode of SEQ1. We can specify to whether we have an interrupt request for every "End of Sequence" (EOS) or every other (EOS).

Bit 8 "EVA_SOC_SEQ1" is the mask bit to enable or disable Event Manager A's ability to trigger a conversion. In Lab6 we will make use of this start feature, so please remember to enable this start option during the procedure of Lab6!

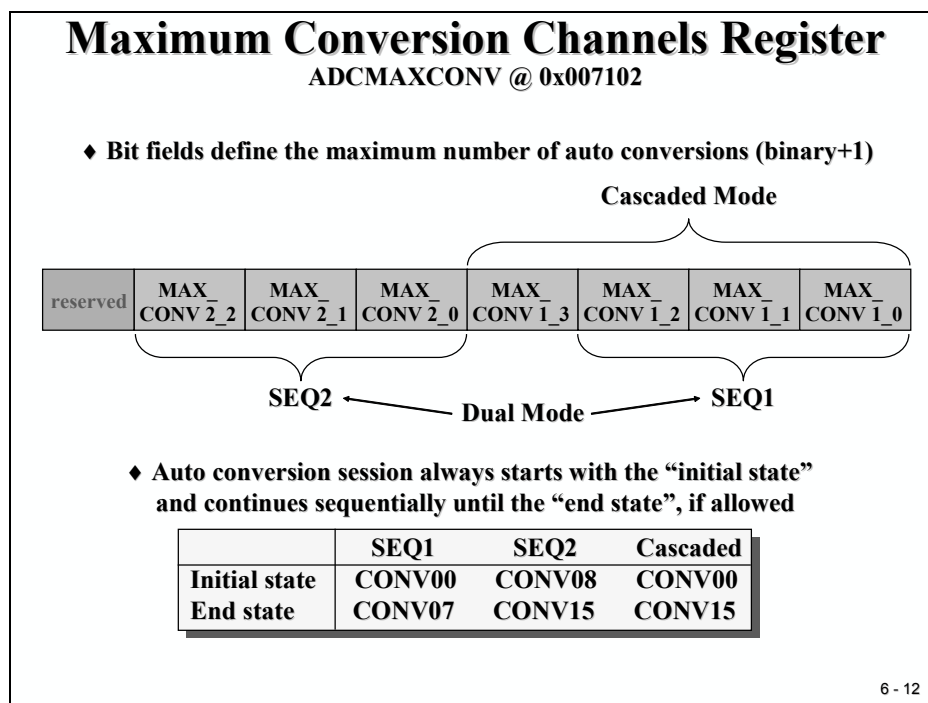


The lower byte of ADCTRL2 is similar to its upper half: it controls sequencer SEQ2. Bit 7 flags the event that the external pin “ADC SOC” has caused the conversion. The rest is identical to the upper half.



ADC MAXCONV Register

”MAXCONV” defines the number of states per trigger.



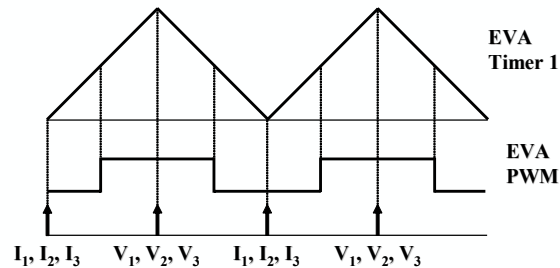
ADC Input Channel Select Sequencing Control Register

	Bits 15-12	Bits 11-8	Bits 7-4	Bits 3-0	
0x007103	CONV03	CONV02	CONV01	CONV00	ADCCHSELSEQ1
0x007104	CONV07	CONV06	CONV05	CONV04	ADCCHSELSEQ2
0x007105	CONV11	CONV10	CONV09	CONV08	ADCCHSELSEQ3
0x007106	CONV15	CONV14	CONV13	CONV12	ADCCHSELSEQ4

6 - 13

Example: 3 phase measurement

Example - Sequencer “Start/Stop” Operation



System Requirements:

- Three auto conversions (I_1, I_2, I_3) off trigger 1 (Timer underflow)
- Three auto conversions (V_1, V_2, V_3) off trigger 2 (Timer period)

Event Manager A (EVA) and SEQ1 are used for this example with sequential sampling mode

6 - 14

The two slides give a typical example of a 3-phase control system for digital motor control.

Example - Sequencer “Start/Stop” Operation (Continued)

- MAX_CONV1 is set to 2 and Channel Select Sequencing Control Registers are set to:

Bits →	15-12	11-8	7-4	3-0	
0x007103	V_1	I_3	I_2	I_1	ADCCHSELSEQ1
0x007104	x	x	V_3	V_2	ADCCHSELSEQ2

- Once reset and initialized, SEQ1 waits for a trigger
- First trigger three conversions performed: CONV00 (I_1), CONV01 (I_2), CONV02 (I_3)
- MAX_CONV1 value is reset to 2 (unless changed by software)
- SEQ1 waits for second trigger
- Second trigger three conversions performed: CONV03 (V_1), CONV04 (V_2), CONV05 (V_3)
- End of second auto conversion session, ADC Results registers have the following values:

RESULT0	I_1	RESULT3	V_1
RESULT1	I_2	RESULT4	V_2
RESULT2	I_3	RESULT5	V_3

- User can reset SEQ1 by software to state CONV00 and repeat same trigger 1, 2 session
- SEQ1 keeps “waiting” at current state for another trigger

6 - 15

Lab 6: Two Potentiometer Voltages

Lab 6: Two Channel Analogue Conversion initiated by GP Timer 1

AIM :

- ◆ AD-Conversion of ADCIN_A0 and ADCIN_B0 initiated by GPT1-period of 0.1 sec.
- ◆ ADCIN_A0 and ADCIN_B0 are connected to two potentiometers to control analogue input voltages between 0 and 3,0V.
- ◆ no GPT1-interrupt-service → Auto-start of ADC with T1TOADC-bit !!
- ◆ Use ADC-Interrupt Service Routine to read out the ADC results
- ◆ Use main loop to show alternately the two results as light-beam on LED's (GPIO port B7..B0)

6 - 18

Additional Registers to initialize Lab 6:

General Purpose Timer Control :	GPTCONA
Timer 1 Control :	T1CON
Timer 1 Period :	T1PR
Timer 1 Compare :	T1CMPR
Timer 1 Counter :	T1CNT
Interrupt Flag :	IFR
Interrupt Enable ask :	IER
ADC – Control 3 :	ADCTRL3
ADC – Control 2 :	ADCTRL2
ADC – Control 1 :	ADCTRL1
Channel Select Sequencer 1 :	CHSELSEQ1
Max. number of conversions :	MAXCONV
ADC - Result 0 :	ADCRESULT0
ADC - Result 1 :	ADCRESULT1

6 - 19

Objective

The objective of this exercise is to practice using the integrated Analogue-Digital Converter of the C28x. The Zwickau Adapter board is equipped with 2 potentiometers at ADCIN_A0 and ADCIN_B0. The two voltages can be changed between 0 and 3.0Volt. The goal of this lab is to read the current status of the potentiometers and to show the voltages as 'Light-Beam' on 8 LED's (GPIO Port B0...B7).

GP Timer 1 generates the sample period of 100msec. The conversion is triggered automatically by a GP Timer 1 period event. The ADC interrupt service routine is the only interrupt needed in this example.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab6.pjt** in E:\C281x\Labs.
2. Open the file Lab5A.c from E:\C281x\Labs\Lab5A and save it as Lab6.c in E:\C281x\Labs\Lab6.
3. Add the source code file to your project:
 - **Lab6.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:\tidcs\c28\dsp281x\v100\DSP281x_common\source add to project:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**
- **DSP281x_Adc.c**
- **DSP281x_usDelay.asm**

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

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Open Lab6.c to edit: double click on “Lab6.c” inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the IQ-Math Library for this exercise:

At the beginning of the code, delete the lines:

```
#include “IQmathLib.h”  
#pragma DATA_SECTION(sine_table, “IQmathTables”);  
_iq30 sine_table[512];
```

8. We do not need the interrupt service routine “T1_Compare_isr()”; instead, we need a new one for the ADC, called “adc_isr()”.

Change the prototype declaration for the interrupt service routine into:

```
interrupt void adc_isr(void);
```

9. Next, just after the function prototype section include the definition of two global integer variables, called “Voltage_A0” and “Voltage_B0”. The two variables will be used to pass the current measurement values from the ADC interrupt service routine to the main loop. Add:

```
int Voltage_A0;  
  
int Voltage_B0;
```

10. Inside “main”, after the function call “InitPieVectTable()”, add the following line to call the basic ADC initialization:

InitAdc();

11. Change the three lines that re-map the PIE – entry and that enable the ADC interrupt into:

PieVectTable.ADCINT = &adc_isr;

PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

IER = 1;

12. Although we initialized the ADC in step 10, we still have to configure its operating mode. Place the corresponding lines after the global interrupt enable instruction “ERTM”. Take into account the following setup:

Dual Sequencer Mode:

AdcRegs.ADCTRL1.bit.SEQ_CASC = ?

No continuous run:

AdcRegs.ADCTRL1.bit.CONT_RUN = ?

Conversion Prescale = CLK/1:

AdcRegs.ADCTRL1.bit.CPS = ?

2 conversions (ADCIN0 and ADCINB0) out of a GP Timer 1 start:

AdcRegs.ADCMAXCONV.all = ?

Setup the channel sequencer to ADCIN0 and ADCINB0:

AdcRegs.ADCSELSEQ1.bit.CONV00 = ?

AdcRegs.ADCSELSEQ1.bit.CONV01 = ?

Enable the Event Manager A to start the conversion:

AdcRegs.ADCTRL2.bit.EVA_SOC_SEQ1 = ?

Enable the ADC interrupt with every end of sequence:

AdcRegs.ADCTRL2.bit.INT_ENA_SEQ1 = ?

We also have to initialize the speed of the ADC (see also slide 6-5). In function “InitSystem()” the high-speed clock prescaler HISPCP is set to ‘divide by 2’. Assuming a SYSCLOCKOUT of 150MHz we have an ADC input clock of 75MHz.

For 'FCLK' the maximum frequency is 25MHz, so we have to setup the ADC clock prescaler "ADCCLKPS" to '0010' → FCLK = 18.75 MHz.

AdcRegs.ADCCTRL3.bit.ADCCLKPS = 2;

13. We also have to adjust the Event Manager A configuration. We do not need to drive any output signal from the GP Timer 1. For Register "GPTCONA" we can disable the two outputs and set the polarity to "forced low":

EvaRegs.GPTCONA.bit.TCMPOE = 0;

EvaRegs.GPTCONA.bit.T1PIN = 0;

To enable the auto start of the ADC with every timer period we have to enable this feature:

EvaRegs.GPTCONA.bit.T1TOADC = 2;

14. Modify the setup for GP Timer 1(Register T1CON)! Take into account to setup:

- "Continuous up count mode",
- "Internal clock source",
- "Stop on emulation suspend" and
- "Disable Compare Operation".

15. Setup the GP Timer 1 Period:

$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP}$$

To setup the timer for a sample period of 100 ms we calculate:

- $f_{CPU} = 150 \text{ MHz}$,
- $HISCP = 2$ and
- $f_{PWM} = 10\text{Hz}$
- $TPS_{T1} = 128$
- $T1PR = 58594$

EvaRegs.T1CON.bit.TPS = 7;

EvaRegs.T1PR = 58594;

16. Delete the line "EvaRegs.EVAIMRA.bit.T1CINT = 1;" – we do not need a Timer Interrupt for this lab.

17. Inside function “Gpio_select()” do not enable T1PWM as pin function.
18. Delete the whole interrupt function “T1_Compare_isr()”.
19. Add a new interrupt service routine “adc_isr()” to your code. Inside this function, do:

- Service the watchdog, part 1:

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

- Read the two ADC result register and load the value into variables “Voltage_A0” and “Voltage_B0”:

```
Voltage_A0 = AdcRegs.ADCRESULT0 >> 4;  
Voltage_B0 = AdcRegs.ADCRESULT1 >> 4;
```

- Reset ADC Sequencer1 (Register ADCCTRL2):

```
AdcRegs.ADCTRL2.bit.RST_SEQ1 = 1;
```

- Clear Interrupt Flag ADC Sequencer 1 (Register ADCST)

```
AdcRegs.ADCST.bit.INT_SEQ1_CLR = 1;
```

- Acknowledge PIE Interrupt:

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP1;
```

20. Inside function “InitSystem()” enable the clock system for the ADC:

```
SysCtrlRegs.PCLKCR.bit.ADCENCLK = 1;
```

Build and Load

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

Project → Build

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

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

File → Load Program and choose the desired output file.

Test

23. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart and
Debug → Go main.

24. Set a breakpoint into interrupt service routine “adc_isr()” at the last line of its code.

Run

25. When you’ve modified your code correctly and you execute a real time run, this breakpoint should be hit periodically. If not, you missed one or more steps in your procedure for this lab exercise. In this case try to review your modifications. If you do not spot a mistake immediately try to test systematically:

- A good start is to temporarily disable the watchdog timer
- Verify that GP Timer1 is counting (T1CNT)
- Verify that the clock system is enabled (PCLKCR) for EVA and ADC
- Inspect the Interrupt Registers (IER, PIEIER, INTM)
- Inspect the ADC Register Set (ADCTRL1-3)

If nothing helps, ask your instructor for advice. Please do not ask questions like “It is not working” or “I do not know what’s wrong...” Instead, summarize your test strategy and show intermediate results for inspection.

26. After you verified that the interrupt service routine “adc_isr()” is called periodically, check the ADC results. Add variables “Voltage_A0” and “Voltage_B0” to your watch window. With the breakpoint still set, modify the analogue input voltages with the two potentiometers “R1” and “R2” of the Zwickau Adapter board. You should be able to get values between 0 and 4095 for the leftmost and rightmost positions of R1 and R2.

Add the display code (LED beam)

27. So far we verified that the GP Timer 1 every 100 ms triggers the ADC to convert the two input voltages periodically. Now we need to add a next portion of code to display the current status of Voltage_A0 and Voltage_B0. To display it, we have to use the 8 LEDs at GPIO – B0 to B7.

A good point to add this code is the while(1) loop of main. After we served the watchdog we can easily add our display code. Question is: how do we display two values with 8 LED’s only?

One option could be to alternate every 2 seconds from display “Voltage_A0” to “Voltage_B0”. Recall, the digital number is in the range 0 to 4095. The rule is simple: the bigger the number the more LED’s should be switched on.

To generate the 2 seconds alternation you can use a simple loop counter in main, or you could use GP Timer 1, now enabled for period interrupt, and count the number of 100ms periods up to 20 before you alternate the display value.

Try to finish this portion of Lab6 by yourself!

END of LAB 6

Optional Lab6A

Modify Lab-Exercise 4 (‘Knight-Rider’) :

- use the Analogue Input ADCIN0 to change the frequency for the LED’s
- to add the ADC-setup use Lab6 as a start
- use a LED-frequency range between 50Hz and 1 Hz
- use (1) a linear or (2) a logarithm scale between F_{\min} and F_{\max} .

6 - 20

Lab 6A: Speed Control of 'Knight Rider'

Objective

Now that we have exercised both with the ADC (Lab 6) and the CPU hardware Timer 0 (LAB 4) we can combine the two exercises. The objective is to control the speed step of the LED's sequence of Lab4 ("Knight Rider") with the ADC-Input ADCIN_A0 → the higher the voltage ADCIN_A0 the higher the speed of the LED-sequence.

We will need two interrupts along with the main function, interrupt 1 for the ADC results and interrupt 2 for core CPU Timer 0.

Use your code from Lab4 and Lab6 as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab6A.pjt** in E:\C281x\Labs.
2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab6A.c in E:\C281x\Labs\Lab6A.
3. Add the source code file to your project:
 - **Lab6A.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:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**

- **DSP281x_Adc.c**
- **DSP281x_CpuTimers.c**
- **DSP281x_usDelay.asm**

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

- **rts2800_ml.lib**

Project Build Options

7. 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**

8. 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 Source Code

9. Open Lab6A.c to edit: double click on “Lab6A.c” inside the project window. Also open the file “Lab6.c” and copy the portions of code that are necessary to use the ADCIN_A0 input voltage to control the period of CPU core Timer 0. In detail:

At the beginning of “Lab6A.c” add the prototype declaration for the interrupt service routine of the ADC:

interrupt void adc_isr(void);

10. Add directly after the prototype section a global integer variable “Voltage_A0”:

int Voltage_A0;

11. Next, in function “main” after the line which calls “InitPieVectTable()” add the function call to initialize the ADC:

InitAdc();

12. In function “main”, after the re-map instruction for PieVectTable.TINT0 add the second re-map instruction:

PieVectTable.ADCINT = &adc_isr;

13. Inside “main”, after PIEIER1.bit.INTx7 is enabled, enable also INTx6 for the ADC-interrupt:

PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

14. Before the line “CpuTimer0Regs.TCR.bit.TSS = 0” add the code from Lab6.c to configure the ADC. Take into account some minor modifications: Only 1 conversion (ADCIN_A0) needed, Channel Select Sequencer.CONV00 to ADCIN_A0.

15. Also before the line “CpuTimer0Regs.TCR.bit.TSS = 0” add another part from “Lab6.c” to initialize the GP Timer 1 (GPTCONA, T1CON, T1PR). No GP Timer 1 period interrupt is needed now. In case you have included this interrupt service in your last part of Lab6, delete these interrupt enable lines.

14. In function “InitSystem()” enable the clock distribution for EVA and ADC:

SysCtrlRegs.PCLKCR.bit.EVAENCLK = 1;

SysCtrlRegs.PCLKCR.bit.ADCENCLK = 1;

15. At the end of the code “Lab6A.c” add the interrupt service routine for the ADC “adc_isr” from Lab6. Delete the line: “Voltage_B0 = AdcRegs.ADCRESULT1 >>4;”

Build and Load

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

Project → Build

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

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

File → Load Program and choose the desired output file.

Test

18. Reset the DSP by clicking on:

Debug → Reset CPU
Debug → Restart
Debug → Go main.

followed by
and

Run

19. Run the code. The LED's should do the "Knight Rider".
20. So far we have reached the same result as in Lab4, a 200msec (50ms *4) period between the steps of the LED-sequence. But additionally now we have an active ADC in the background!
21. Open a watch window to watch variable "Voltage_A0". Click right mouse inside the watch window and select "Refresh". When you modify the potentiometer you should be able to see values between 0 and 4095, assuming you repeat the "Refresh" mouse click.

Modify the main loop

22. All we have to do now is to use variable "Voltage_A0" to control delay line that we so far used in the main loop:

```
while(CpuTimer0.InterruptCount < 3);
```

Now we can replace the constant '3' by a variable that is modified by "Voltage_A0". It is also recommended to include the watchdog service into this while-loop:

```
while(CpuTimer0.InterruptCount < x)
{
    EALLOW;
    SysCtrlRegs.WDKEY = 0xAA;
    EDIS;
}
```

All you have to do now is to find a useful construction to calculate an expression for x out of Voltage_A0. Hint: InterruptCount is a multiple of 50µs, so let's try to limit the value for x between 1 (= 50µs period) and 20(=1 sec period).

END of LAB 6A

This page was intentionally left blank.

C28x Serial Peripheral Interface

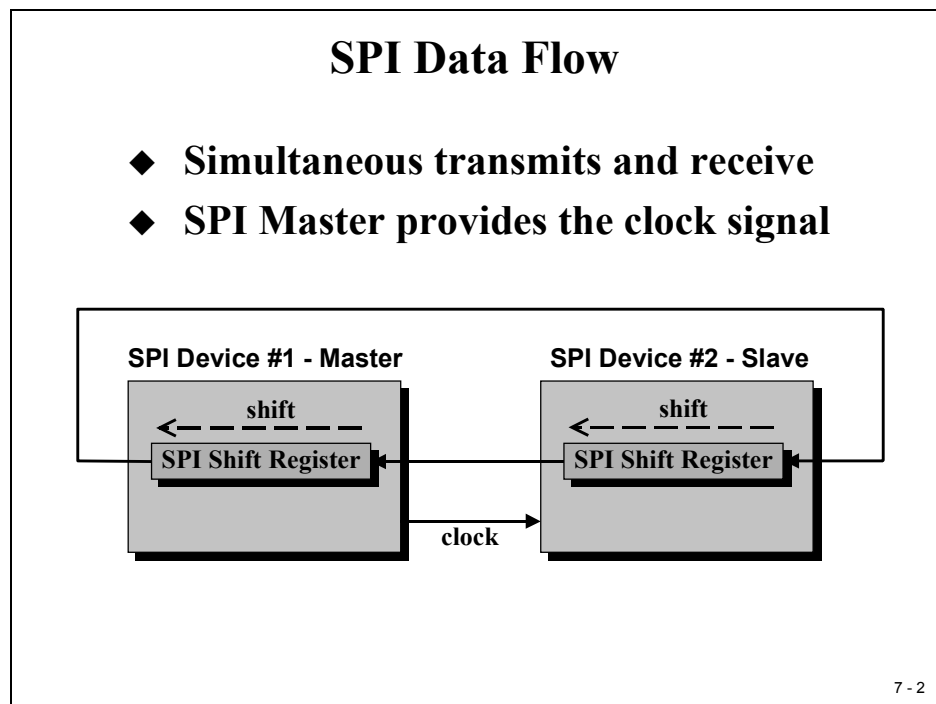
Introduction

The TMS320C28x contains built-in features that allow several methods of communication and data exchange between the C28x and other devices. This chapter deals with the Serial Peripheral Interface (SPI). Two other interface techniques (SCI and CAN) will be discussed in later chapters.

The SPI module is a synchronous serial I/O port that shifts a serial bit stream of variable length and data rate between the 'C28x' and other peripheral devices. Here "synchronous" means that the data transmission is synchronized to a clock signal.

During data transfers, one SPI device must be configured as the transfer MASTER, and all other devices configured as SLAVES. The master drives the transfer clock signal for all SLAVES on the bus. SPI communication can be implemented in any of three different modes:

- MASTER sends data, SLAVES send dummy data
- MASTER sends data, one SLAVE sends data
- MASTER sends dummy data, one SLAVE sends data



Module Topics

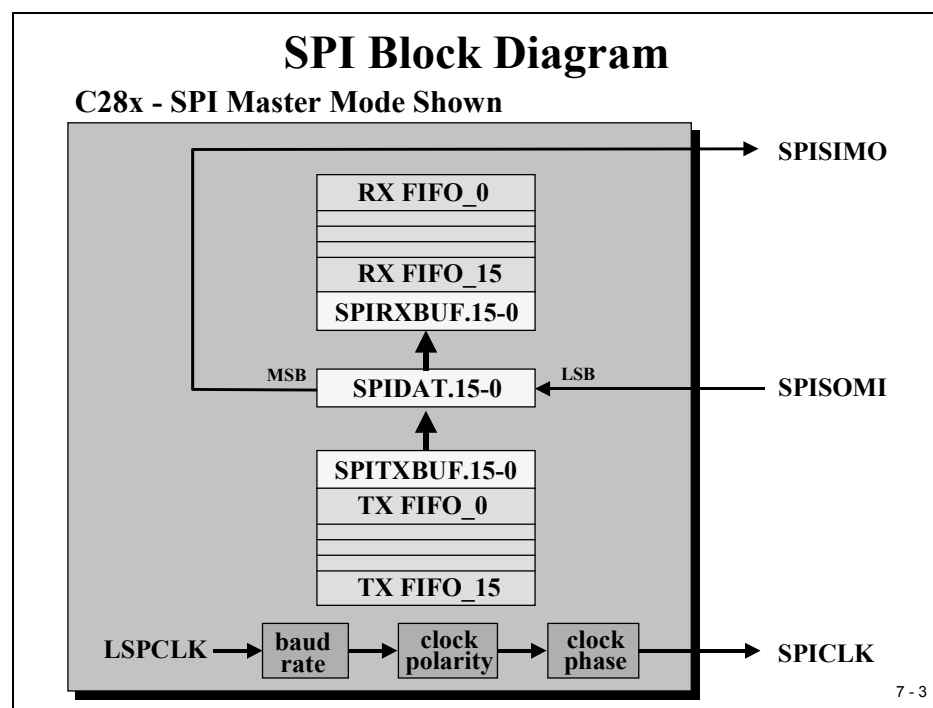
C28x Serial Peripheral Interface	7-1
<i>Introduction</i>	<i>7-1</i>
<i>Module Topics.....</i>	<i>7-2</i>
<i>Serial Peripheral Interface (SPI) – Overview.....</i>	<i>7-4</i>
<i>SPI Data Transfer</i>	<i>7-5</i>
<i>SPI Register Set.....</i>	<i>7-6</i>
SPI Configuration Control Register - SPICCR.....	7-7
SPI Operation Control Register – SPICTL.....	7-7
SPI Baud Rate Register – SPIBRR.....	7-8
SPI Status Register – SPISTS.....	7-8
SPI FIFO Transmit Register	7-9
<i>SPI Summary.....</i>	<i>7-10</i>
<i>Lab 7: SPI – Dual DAC Texas Instruments TLV5617A</i>	<i>7-11</i>
Objective	7-11
Aim of Lab7:	7-12
DAC TLV5617A Data Format	7-13
Procedure.....	7-14
Open Files, Create Project File.....	7-14
Project Build Options	7-14
Modify Source Code.....	7-15
Build and Load	7-15
Test	7-16
Run	7-16
Add the SPI initialization code.....	7-16
Add the DAC – update code.....	7-18
Build, Load and Run.....	7-19
<i>Lab 7A: Code Composer Studio Graph Tool.....</i>	<i>7-20</i>
Objective	7-20
Procedure.....	7-20
Open Files, Create Project File.....	7-20
Project Build Options	7-21
Modify Source Code.....	7-21
Build and Load	7-22
Test	7-23
Run	7-23
Add a graphical window.....	7-23
<i>Lab 7B: SPI – EEPROM M95080.....</i>	<i>7-25</i>
Objective	7-25
Aim of Lab 7B:.....	7-26
M95080 Status Register.....	7-27
M95080 Instruction Set	7-28
Procedure Lab7B	7-31
Open Files, Create Project File.....	7-31

Project Build Options	7-31
Modify Source Code.....	7-32
Build and Load	7-33
Test	7-33
Run	7-33
Add the SPI initialization code.....	7-33
Create EEPROM access functions.....	7-35
Finalize the main loop	7-36
Build, Load and Run.....	7-37

Serial Peripheral Interface (SPI) – Overview

In its simplest form, the SPI can be thought of as a programmable shift register. Data bits are shifted in and out of the SPI through the SPIDAT register. Two more registers set the programming interface. To transmit a data frame, we have to write the 16-bit message into the SPITXBUF buffer. A received frame will be read by the SPI directly into the SPIRXBUF buffer. For our lab exercises, this means we write directly to SPITXBUF and we read from SPIRXBUF.

There are two operating modes for the SPI: “basic mode” and “enhanced FIFO-buffered mode”. In “basic mode”, a receive operation is double-buffered, that is the CPU need not read the current received data from SPIRXBUF before a new receive operation can be started. However, the CPU must read SPIRXBUF before the new operation is complete or a receiver-overflow error will occur. Double-buffered transmit is not supported in this mode; the current transmission must be complete before the next data character is written to SPITXDAT or the current transmission will be corrupted. The Master can initiate a data transfer at any time because it controls the SPICLK signal.

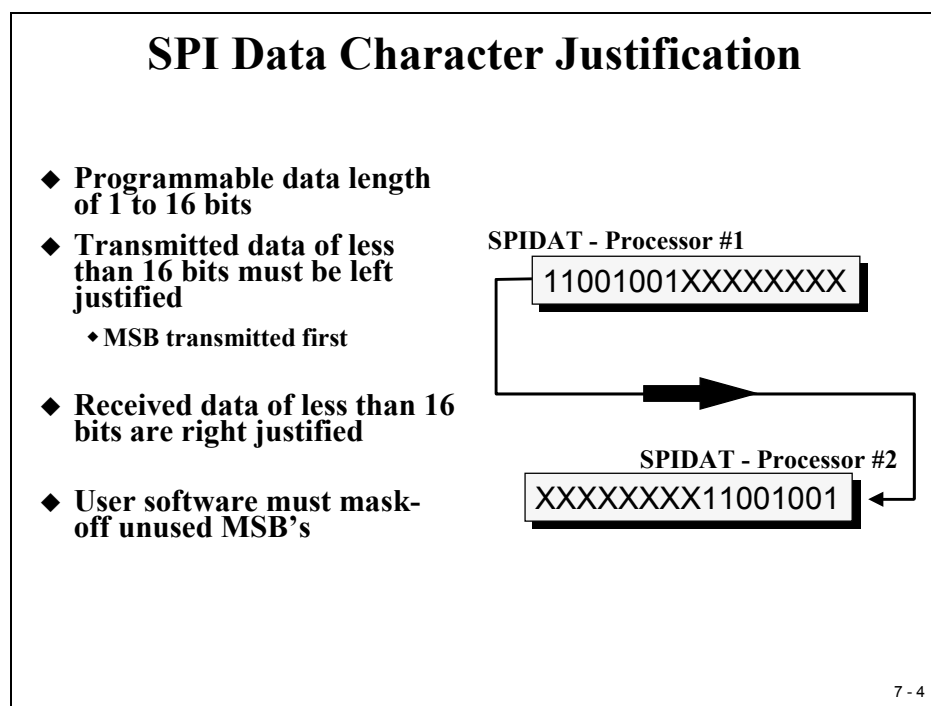


In “enhanced FIFO – buffered mode” we can build up to 16 levels of transmit- and receive FIFO memory. Again, our program interfaces to the SPI unit are the registers SPITXBUF and SPIRXBUF. This will expand the SPI’s buffer capacity for receive and transmit to up to 16 times. In this mode we are also able to specify an interrupt level that depends on the filling state of the two FIFO’s.

SPI Data Transfer

As you can see from the previous slide, the SPI master is responsible for generating the data rate of the communication. Derived from the internal low speed clock prescaler (LSPCLK), we can specify an individual baud rate for the SPI. Because not all SPI devices are interfaced in the same way, we can adjust the shape of the clock signal by two more bits, “clock polarity” and “clock phase”. Strictly speaking, the SPI is not a standard; slave devices like EEPROM’s, DAC’s, ADC’s, Real Time Clocks, temperature sensors do have different timing requirements for the interface timing. For this reason TI includes options to adjust the SPI timing.

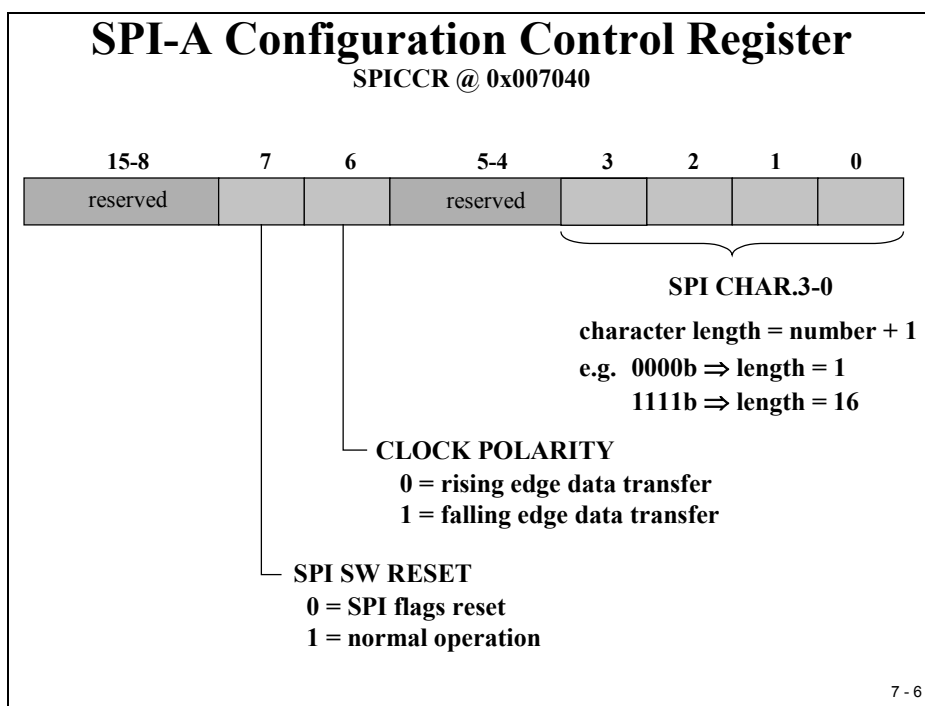
A transmission always starts with the MSB (most significant bit) out of SPIDAT first and received data will be shifted into the device, also with MSB first. Both transmitter and receiver perform a left shift with every SPI clock period. For frames of less than 16 bits, data to be transmitted must be left justified before transmission starts. Received frames of less than 16 bits must be masked by user software to suppress unused bits.



SPI Register Set

The next slide summarizes all SPI control registers. In future devices of the C28x family we will have a second SPI, for this reason the names of the first SPI are expanded with an 'A'.

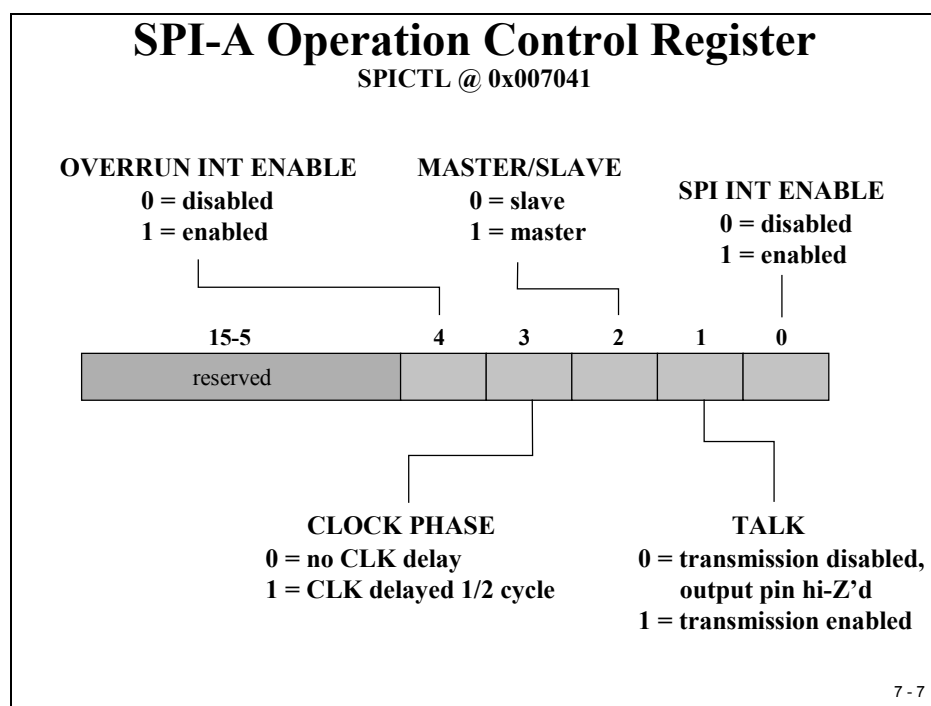
SPI-A Registers		
Address	Register	Name
0x007040	SPICCR	SPI-A configuration control register
0x007041	SPICTL	SPI-A operation control register
0x007042	SPISTS	SPI-A status register
0x007044	SPIBRR	SPI-A baud rate register
0x007046	SPIEMU	SPI-A emulation buffer register
0x007047	SPIRXBUF	SPI-A serial receive buffer register
0x007048	SPITXBUF	SPI-A serial transmit buffer register
0x007049	SPIDAT	SPI-A serial data register
0x00704A	SPIFFTX	SPI-A FIFO transmit register
0x00704B	SPIFFRX	SPI-A FIFO receive register
0x00704C	SPIFFCT	SPI-A FIFO control register
0x00704F	SPIPRI	SPI-A priority control register



SPI Configuration Control Register - SPICCR

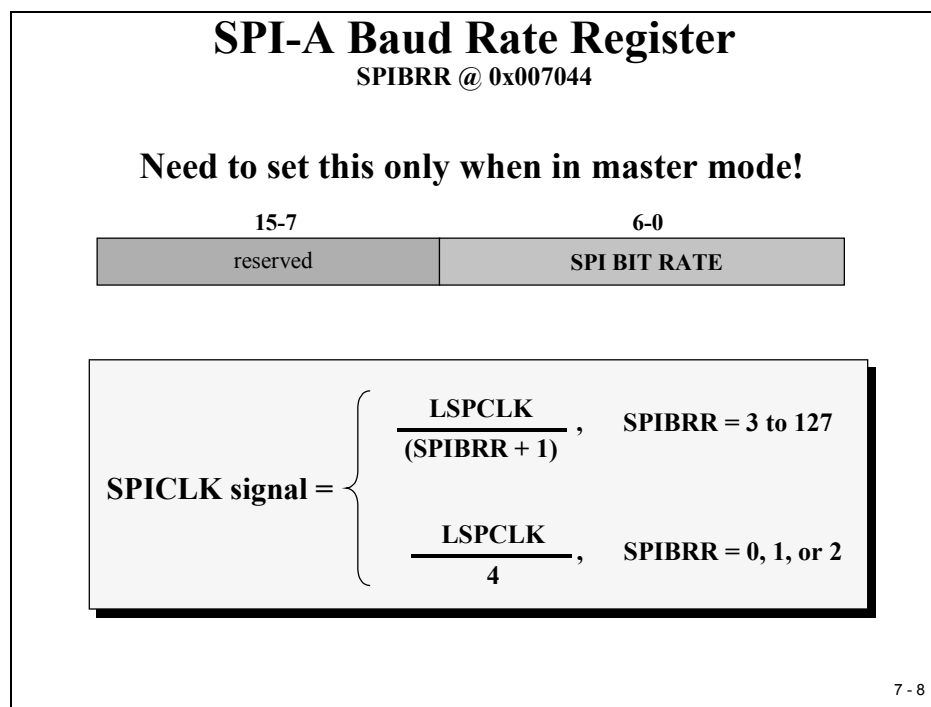
It is good practice to RESET the SPI unit at the beginning of the initialization procedure. This is done by clearing Bit 7 (SPI SW RESET) to 0 followed by setting it to 1. Bit 6 selects the active clock edge to declare the data as valid. This selection depends on the particular SPI – device (see the two examples used in the labs of this chapter). Bits 3...0 define the character length of the SPI-frame.

SPI Operation Control Register – SPICTL



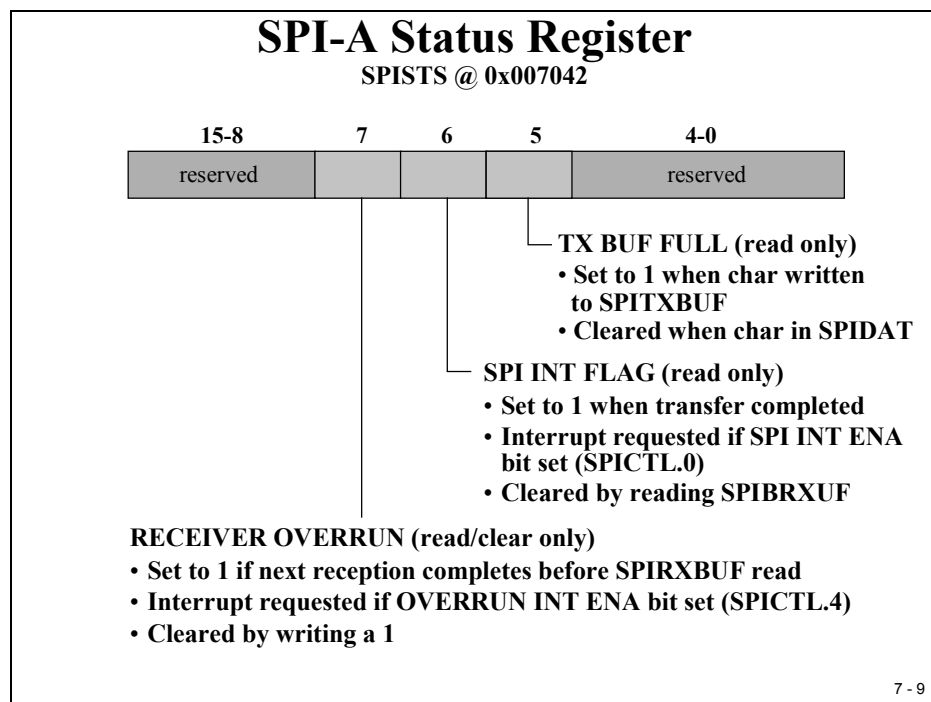
Bit 4 and 0 enable or disable the SPI- interrupts; Bit 4 enables the receivers overflow interrupt. Bit 2 defines the operating mode for the C28x to be master or slave of the SPI-chain. With the help of bit 3 we can implement another half clock cycle delay between the active clock edge and the point of time, when data are valid. Again, this bit depends on the particular SPI-device. Bit 1 controls whether the C28x listens only (Bit 1 = 0) or if the C28x is initialized as receiver and transmitter (Bit 1 = 1).

SPI Baud Rate Register – SPIBRR

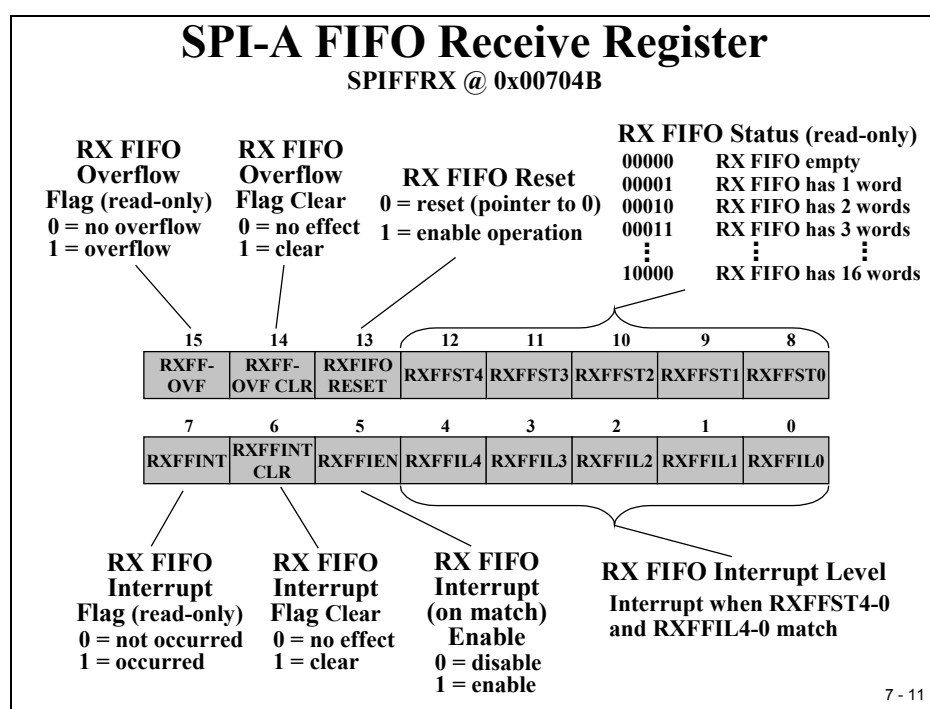
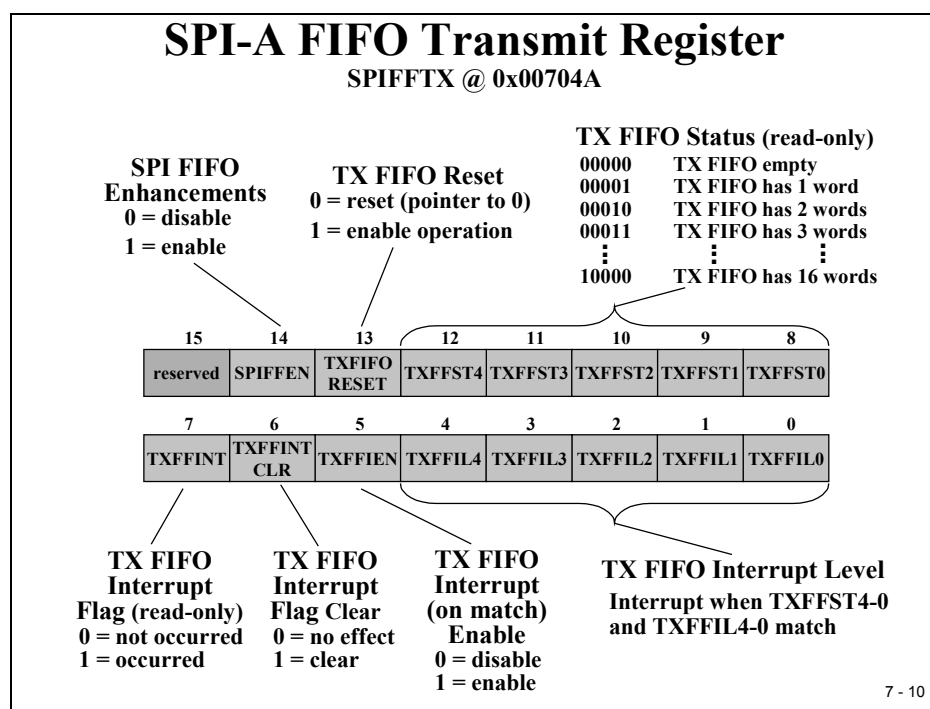


Clock base for the SPI baud rate selection is the Low speed Clock Prescaler (LSPCLK).

SPI Status Register – SPISTS



SPI FIFO Transmit Register



The FIFO operation of the SPI is controlled with Bit 14 as a master switch. The SPI-Transmit FIFO interrupt service call depends now on the match between TX FIFO Status and TX FIFO Interrupt Level. The TX FIFO Reset can be used to reset the FIFO state machine (Bit13= 0) and to re-enable it (Bit 13=1).

SPI Summary

SPI Summary

- ◆ **Provides synchronous serial communications**
 - ◆ Two wire transmit or receive (half duplex)
 - ◆ Three wire transmit and receive (full duplex)
- ◆ **Software configurable as master or slave**
 - ◆ C28x provides clock signal in master mode
- ◆ **Data length programmable from 1-16 bits**
- ◆ **125 different programmable baud rates**

7 - 12

Lab 7: SPI – Dual DAC Texas Instruments TLV5617A

SPI Example 1: DAC TLV 5617

- ◆ **Texas Instruments Digital to Analogue Converter (DAC) TLV 5617A**
 - ◆ **10 MBPS SPI Data Communication**
 - ◆ **Dual Channel Analogue Output (Out A + B)**
 - ◆ **10 Bit resolution**
 - ◆ **/CS is connected to C28x GPIO – D0 at the Zwickau Adapter Board**
 - ◆ **REF – Voltage defines Analogue Range / 2**
 - ◆ **SOIC-8**
 - ◆ **Operating Voltage : 0 to 3.3V**

7 - 13

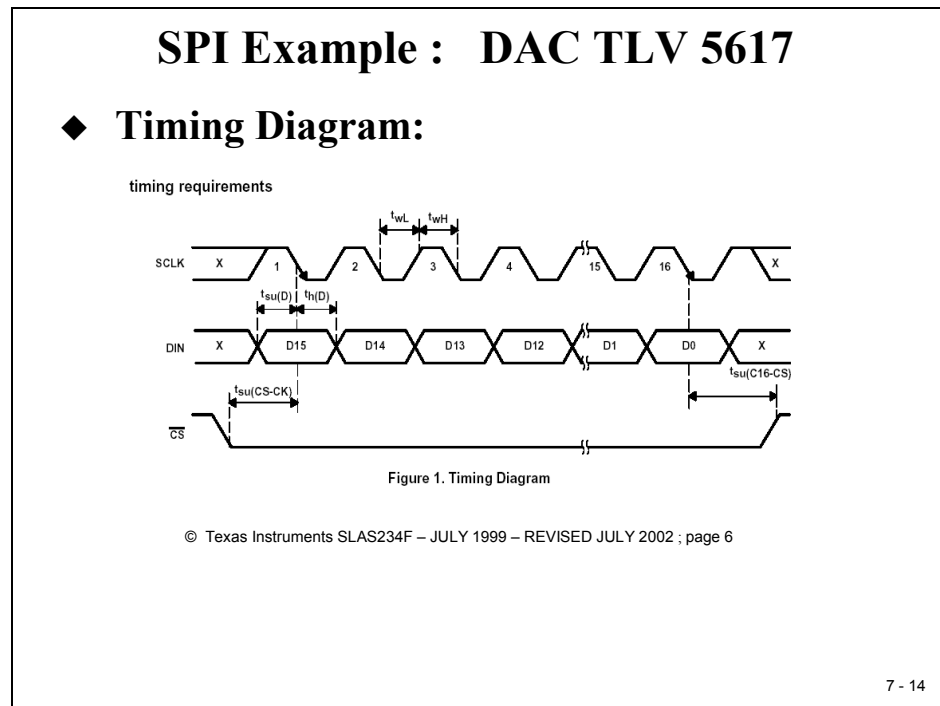
Objective

The objective of this lab is to establish an SPI communication between the C28x and a serial DAC TLV 5617. This DAC has two output channels and a resolution of 10 bits. The datasheet of this device is available from TI's website, search for document number 'SLAS234'. The interface between the C28x and the DAC is defined as follows:

TLV5617 – Signal Name	Pin – No.	Description	Connected to...
AGND	5	Ground	AGND
/CS	3	Chip - Select	C28x- GPIO D0
DIN	1	Input Data	C28x-SPISIMO
SCLK	2	SPI Clock	C28x-SPICLK
REF	6	Analogue Reference input	3.3V
VDD	8	Power supply	3.3V
OUT A	4	DAC output A	JP7 – 1
OUT B	7	DAC output B	JP8 – 1

The chip-select (/CS) of the DAC is connected to the GPIO – D0. The TLV5617 is a ‘listen only’ SPI device; it sends no data back to the C28x. The REF voltage defines the full-scale value for the analogue output voltages. It is connected to 3.3V. Although the TLV5617 is a 10 Bit DAC it uses an internal multiplier by 2, therefore the digital input values lies in the range 0 to 511.

The TLV 5617 has the following timing requirements:



The active frame is covered with an active /CS-Signal, MSB comes first, the data bit leads the falling edge of the SCLK-Signal by half of a clock period. The DAC’s internal operation starts with the rising edge of /CS.

Aim of Lab7:

The Aim of the Lab7 is to generate two saw tooth signals at the outputs of the SPI-DAC: DAC-A - “rising saw tooth” and DAC-B” - falling saw tooth between 0V and 3.3V. With the help of a scope meter, if available, we should be able to measure the two voltages. In an optional exercise (“Lab7A”) we also will make use of the graphical tool of Code Composer Studio to visualize the two voltages. To do this we have to reconnect the DAC signals back to two ADC-Input lines. This is done by closing the jumpers JP7 and JP8 on the Zwickau Adapter board.

Lab 7: DAC TLV 5617

◆ Objective:

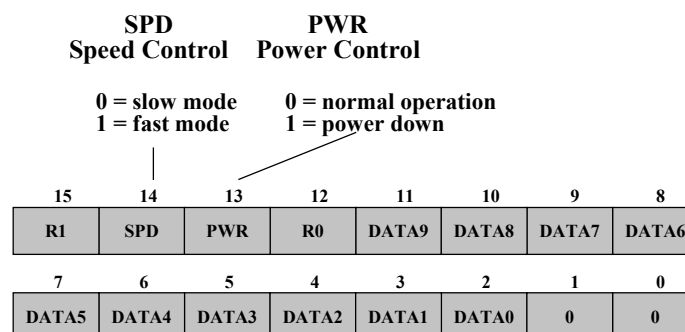
- ◆ Generate a rising saw-tooth (0V...3.3V) at channel OUTA and a falling saw-tooth (3.3V...0V) at channel OUTB
- ◆ GPIO – D0 is DAC's chip select (/CS) at the Zwickau Adapter Board
- ◆ To measure the DAC outputs:
 - ◆ Use JP7 for OUTA
 - ◆ Use JP8 for OUTB (Zwickau Adapter Board)
- ◆ REF = 3.3V
- ◆ Feedback the voltages into the C28x ADC:
 - ◆ JP7 closed: OUTA → ADCINA1
 - ◆ JP8 closed: OUTB → ADCINB1

7 - 16

DAC TLV5617A Data Format

SPI Example : DAC TLV 5617

◆ Serial Data Format:



R1 , R0 Register Select

- 00: Write to DACB and Buffer
- 01: Write to Buffer
- 10: Write to DACA and update DACB with Buffer
- 11: reserved

7 - 15

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab7.pjt** in E:\C281x\Labs.
2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab7.c in E:\C281x\Labs\Lab7.
3. Add the source code file to your project:
 - **Lab7.c**
4. From C:\ti\dcs\c28\dsp281x\v100\DSP281x_headers\source add:

- **DSP281x_GlobalVariableDefs.c**

From C:\ti\dcs\c28\dsp281x\v100\DSP281x_headers\cmd add:

- **F2812_Headers_nonBIOS.cmd**

From C:\ti\dcs\c28\dsp281x\v100\DSP281x_common\cmd add:

- **F2812_EzDSP_RAM_Ink.cmd**

From C:\ti\dcs\c28\dsp281x\v100\DSP281x_common\source add to project:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**
- **DSP281x_CpuTimers.c**

From C:\ti\c2000\cgtoolslib add:

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Open Lab7.c to edit: double click on “Lab7.c” inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the main variables “LED[8]” and “i” for this exercise:

At the beginning of main, delete the lines:

```
unsigned int i;  
unsigned int LED[8]= {0x0001,0x0002,0x0004,0x0008,  
0x0010,0x0020,0x0040,0x0080};
```

8. Next, inside the “while(1)”-loop of main reduce the code to just the following lines (we will add some more code later):

```
while(1)  
{  
  
while(CpuTimer0.InterruptCount < 3); // wait for Timer 0  
CpuTimer0.InterruptCount = 0;  
EALLOW;  
SysCtrlRegs.WDKEY = 0xAA; // and service watchdog #2  
EDIS;  
}
```

9. Before we continue to add the SPI modifications lets test if the project in its preliminary stage runs as expected. Recall, with the start code of Lab4 we initialized the CPU core timer 0 to generate an interrupt request every 50ms. The interrupt service routine “cpu_timer0_isr()” increments a global variable “CpuTimer0.InterruptCount” with every hit. If everything works as expected the DSP should hit the line

```
CpuTimer0.InterruptCount = 0;
```

in the while(1) – loop (procedure step 8) every 3*50ms = 150 ms.

Build and Load

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

Project → Build

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

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

File → Load Program and choose the desired output file.

Test

12. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart and
Debug → Go main.

13. In the “while(1)”-loop of main set a breakpoint at line:

CpuTimer0.InterruptCount = 0;

Run

14. Verify, that the breakpoint is hit periodically when you start the DSP by:

Debug → Run (F5).

Remove the breakpoint when you are done.

Add the SPI initialization code

15. So far we verified that the CPU Core Timer0 generates a period of 50 ms and that our main-loop waits for 3 periods of Timer0 before it moves to the next instruction. Now we need to add the code for the SPI to control the DAC TLV 5617A. First function that is called is “InitSystem()”. Do we have to adjust this function? YES! We have to enable the SPI clock unit! Inside “InitSystem()” do:

SysCtrlRegs.PCLKCR.bit.SPIENCLK=1;

16. The next function that is called in main is “Gpio_select()”. Inside this function we have to modify the multiplex register to use the four SPI-signals:

GpioMuxRegs.GPFMUX.all = 0xF;

Bits 3...0 control the setup for SPISTEA, SPICLKA, SPISOMIA and SPISIMOA. We also prepare GPIO-signal D0 and D5 to be digital outputs. D0 will be used as chip-select for the TLV5617A and D5 for the EEPROM M95080 (used in Lab7B). To do this, setup GPDDIR register:

`GpioMuxRegs.GPDDIR.all=0;`

`GpioMuxRegs.GPDDIR.bit.GPIO0 = 1; // /CS for DAC TLV5617A`

`GpioMuxRegs.GPDDIR.bit.GPIO5 = 1; // /CS for EEPROM`

As an initial state we should switch off both /CS-signals as well as the LED's at B7...B0:

`GpioDataRegs.GPBDAT.all = 0x0000; // Switch off LED's (B7...B0)`

`GpioDataRegs.GPDDAT.bit.GPIO0 = 1; // /CS for DAC off`

`GpioDataRegs.GPDDAT.bit.GPIO5 = 1; // /CS for EEPROM off`

17. In main, just before we enter the “while(1)”-loop add a function call to function “SPI_Init()”. Also add a function prototype at the start of your code.

At the end of your code, add the definition of function “SPI_Init()”.

Inside this function include the following steps:

- SPICCR:
 - Clock polarity =1: data output at falling edge of clock
 - 16 bit per data frame
- SPICTL:
 - No SPI interrupts for this exercise
 - Master on
 - Talk enabled
 - Clock phase = 1: one half cycle delay
- SPIBRR:
 - $BRR = LSPCLK / SPI_Baudrate - 1$
 - Example: assuming $LSPCLK = 37.5\text{MHz}$ and $SPIBRR = 124$ the SPI-Baud rate is 300 kbps.

Add the DAC – update code

18. Now we can add code to update the DAC-outputs with every loop of our main code. Recall, the objective of this lab was to generate a rising saw tooth voltage at DAC-output A and a falling saw tooth at DAC-output B.

Obviously we need two integer variables “Voltage_A” and “Voltage_B” to store the current digital value for the two DAC-output lines. “Voltage_A” (rising saw tooth) starts with initial value 0, “Voltage_B” (falling saw tooth) with the maximum digital value 511. Add the two lines at the beginning of “main”:

```
int Voltage_A = 0;
```

```
int Voltage_B = 511;
```

To update the DAC it would be a good solution to write a function called “DAC_Update” with two input parameters (channel_number and value). Add a prototype “void DAC_Update (char channel, int value)” at the beginning of the code and the definition at the end of your code:

```
void DAC_Update(char channel, int value);
```

In main, call the function “DAC_Update (channel, value)” inside the “while(1)”-loop of main twice, just after the line:

```
CpuTimer0.InterruptCount = 0;
```

```
DAC_Update('B', Voltage_B);
```

```
DAC_Update('A', Voltage_A);
```

After the calls we need to increment “Voltage_A” and decrement “Voltage_B” and we have to reset “Voltage_A” if it exceeds the maximum of 511. Same if “Voltage_B” is decremented below 0:

```
if (Voltage_A++ > 511) Voltage_A = 0;
```

```
if (Voltage_B-- < 0) Voltage_B = 511;
```

19. Now, what should be done inside “DAC_Update(char channel, int value)”?

Obviously, the activity depends on the selected channel. If channel == ‘B’ we have to load “value” into DAC-Buffer (see Slide 7-15, command R1, R0 = 01), if channel == ‘A’ we have to load “value” directly on DAC-output channel A and update DAC-output channel B with value out of buffer (slide 7-15, command R1, R0 = 10). By doing so, we can make sure that both outputs are updated synchronously.

Before we can load SPITXBUF with a data frame we have to enable the DAC’s chip select. For this purpose we defined port GPIO-D0. After the end of the transmission we have to disable it again.

How can we find out if the transmission from the SPI into the DAC is completed? We did not enable any SPI-interrupts, so all we can do is to poll the SPI-interrupt flag to check if the SPI communication has finished. Note, that there will be still one bit to be transmitted after the SPI-interrupt has been set; therefore it is recommended

to add another small wait loop before we switch off the Chip-Select signal of the DAC.

To reset the SPI-Interrupt Flag we have to do a dummy-read from SPIRXBUF.

Adding all the tiny bits of procedure step 19 together, your function “DAC_Update” should include this sequence:

```
int i;
GpioDataRegs.GPDDAT.bit.GPIOD0 = 0;    // activate /CS
if (channel == 'B')
    SpiaRegs.SPITXBUF = 0x1000 + (value<<2);
    // transmit data to DAC-Buffer
if (channel == 'A')
    SpiaRegs.SPITXBUF = 0x8000 + (value<<2);
    // transmit data to DAC-A and update DAC-B with Buffer
while (SpiaRegs.SPISTS.bit.INT_FLAG == 0) ;
// wait for end of transmission
for (i=0;i<100;i++);                // wait for DAC to finish off
GpioDataRegs.GPDDAT.bit.GPIOD0 = 1;    // deactivate /CS
i = SpiaRegs.SPIRXBUF;                // read to reset SPI-INT
```

Build, Load and Run

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

```
Project → Build
File → Load Program
Debug → Reset CPU
Debug → Restart
Debug → Go main
Debug → Run(F5)
```

21. With the help of a scope meter or a scope you should use jumper JP7 and JP8 at the Zwickau Adapter board to verify the two saw tooth voltages. Recall that we used CPU Core Timer0, initialized to a time base of 50ms. In main, we wait for 3 increments of CpuTimer0.InterruptCount (150ms). We have 511 steps of increments for the DAC; therefore the period of the whole saw tooth is $511 * 150\text{ms} = 76.65\text{s}$.
22. If you like it faster, modify the frequency of core timer0!

END of LAB 7

Lab 7A: Code Composer Studio Graph Tool

Objective

At the Zwickau Adapter Board we can re-connect the two saw tooth voltages of the SPI-DAC into the internal ADC. Doing so, we can verify the results of our DAC exercise (Lab7). Code Composer Studio has the ability to show the values of a memory area as a graphical image. The objective of exercise 7A is to use this feature.

The SPI-DAC channel A is connected to ADCINA1 and SPI-DAC channel B to ADCINB1 (Zwickau Adapter Board JP7 and JP8 closed). Note that any voltage above 3.0 to 3.3V will be saturated inside the ADC.

Use your code from Lab7 as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab7A.pjt** in E:\C281x\Labs.
2. Open the file Lab7.c from E:\C281x\Labs\Lab7 and save it as Lab7A.c in E:\C281x\Labs\Lab7A.

3. Add the source code file to your project:

- **Lab7A.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:\tidcs\c28\dsp281x\v100\DSP281x_common\source add to project:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**

- **DSP281x_Adc.c**
- **DSP281x_CpuTimers.c**
- **DSP281x_usDelay.asm**

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

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Open Lab7A.c to edit: double click on “Lab7A.c” inside the project window. Inside function “InitSystem” enable the ADC clock system:

SysCtrlRegs.PCLKCR.bit.ADCENCLK=1;

8. In main, just after the function call of “SPI_Init()” call the initialization for the ADC:

InitAdc();

This will initialize the ADC’s internal voltages.

9. Direct after the call of “InitAdc” add the remaining steps to initialize the ADC. Eventually you can use your ADC – setup from lab exercise 6. Take into account:

- Setup Dual Sequencer Mode
- No Continuous run
- Adc - Prescaler = 1 (CPS)

- 2 conversions per start of conversion
- Channels ADCINA1 & ACINB1
- Disable EVASOC to start SEQ1
- Enable SEQ1 interrupt every End of Sequence
- ADCCLKPS = 2 (Divide HSPCLK by 4)

10. Next, add an overload for the PIE - vector table (ADCINT) to point to a new function “ADC_ISR()”:

EALLOW;

PieVectTable.ADCINT = &ADC_ISR;

EDIS;

Also, add the enable instruction for the ADC-Interrupt:

PieCtrlRegs.PIEIER1.bit.INTx6 = 1;

11. Add the provided source code for function “ADC_ISR” to your project. From *E:\C281x\Labs\Lab7A* add to project:

- **ADC_ISR.c**

12. Add a function prototype for interrupt service routine “ADC_ISR” at the beginning of your code in “Lab7A.c”:

interrupt void ADC_ISR(void);

13. Inside the while(1)-loop of main, just after the line “CpuTimer0.InterruptCount = 0;” add the instruction to start a conversion of the ADC:

AdcRegs.ADCTRL2.bit.SOC_SEQ1 = 1;

Build and Load

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

Project → Build

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

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

File → Load Program and choose the desired output file.

Test

16. Reset the DSP by clicking on:

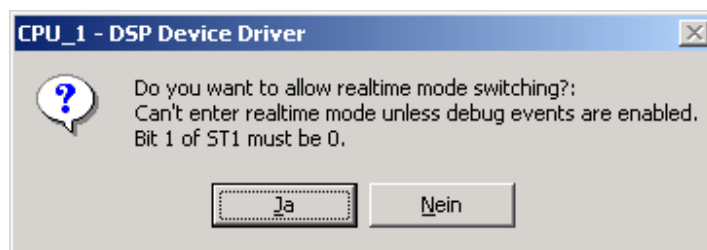
Debug → Reset CPU followed by
Debug → Restart and
Debug → Go main.

Run

17. Enable Real Time Debug. Click on

→ Debug → Real Time mode

Answer the following window with: Yes



18. Open the Watch – Window and add the ADC – Result Register “ADCRESULT0” and “ADCRESULT1” to it. Note: you can do this manually, or (recommended) use the GEL-Menu:

→ GEL → Watch ADC Registers → ADCRESULT_0_to_3

Inside the Watch Window Click right mouse and select “Continuous Refresh”.

19. Run the Code:

→ Debug → Run (F5)

ADCRESULT0 should show rising values, ADCRESULT 1 falling values. Stop the DSP.

Add a graphical window

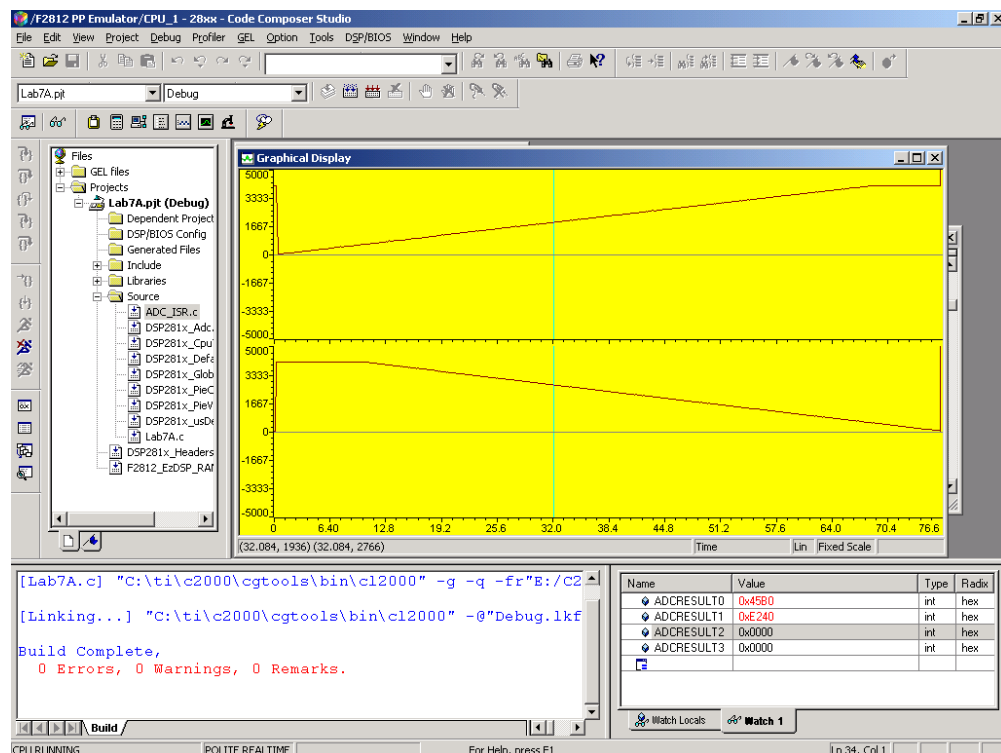
20. To visualize the measured saw tooth voltages open a graph window:

→ View → Graph → Time/Frequency

and enter the following properties:

Display Type	Dual Time
Start Address Upper Display	AdcBuf_A
Start Address Lower Display	AdcBuf_B
Acquisition Buffer Size	512
Display Data Size	512
Sampling Rate (Hz)	6.67
DSP Data Type	16-Bit unsigned integer
Auto scale	OFF
Time Display Unit	s

Click right mouse inside the graph window and select “Continuous Refresh” and run the code again.



END of LAB 7A

Lab 7B: SPI – EEPROM M95080

SPI Example 2: EEPROM M95080

- ◆ **ST Microelectronics EEPROM M95080**
 - ◆ **10 MBPS SPI Data Communication**
 - ◆ **Capacity: 1024 x 8 Bit**
 - ◆ **/CS is connected to C28x GPIO – D5 (Zwickau Adapter Board)**
 - ◆ **6 Instructions:**
 - ◆ **Write Enable, Write Disable**
 - ◆ **Read Status Register, Write Status Register**
 - ◆ **Read Data, Write Data**
 - ◆ **SOIC-8**
 - ◆ **Single Power Supply : 3.3V**

7 - 17

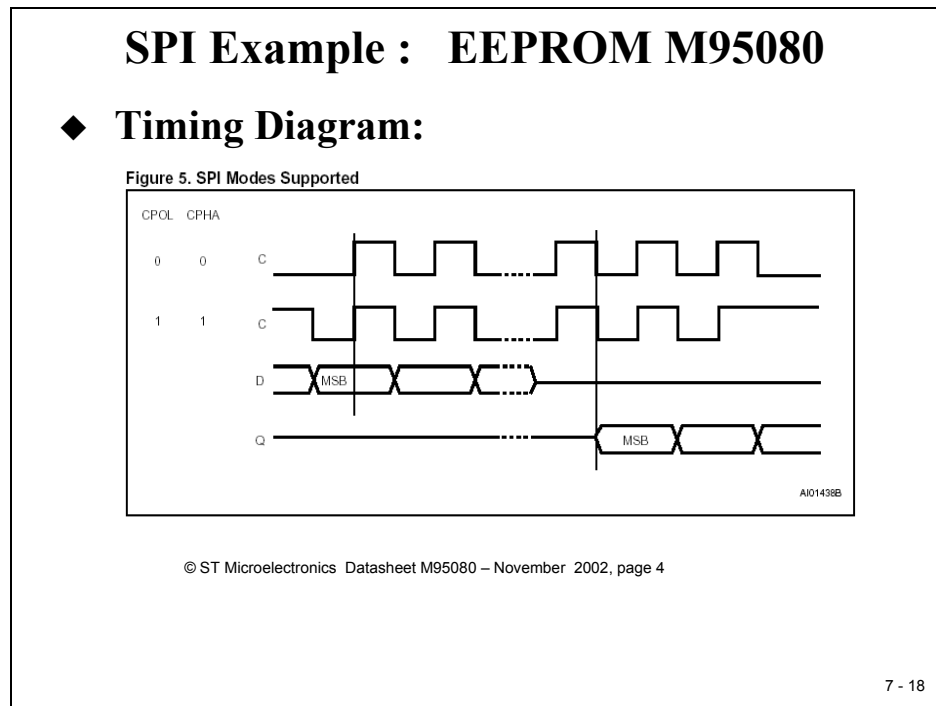
Objective

The objective of this lab is to establish an SPI communication between the C28x and a serial EEPROM ST M95080. The interface between the C28x and this 1024 x 8 Bit - EEPROM is the standard SPI and uses the following connections:

M95080 – Signal	Pin #	Description	Connected to...
VSS	4	Ground	GND
VCC	8	3.3V	3.3V
C	6	SPI Clock	C28x - SPICLK
D	5	SPI Data In	C28x –SPISIMO
Q	2	SPI Data OUT	C28x - SPISOMI
/S	1	Chip Select	C28x – GPIO D5
/W	3	Write protect	3.3V
/HOLD	7	Hold Communication	3.3V

The chip-select (/CS) of the EEPROM is connected to the GPIO – D5. The EEPROM is able to store data non-volatile. This means we need to setup a closed SPI – loop to write and to read data. The EEPROM data input ‘D’ is connected to the DSP’s ‘SIMO’ (Slave In - Master Out) and the EEPROM’s output line ‘Q’ drives serial signals to ‘SOMI’ (Slave Out – Master In). The signals ‘Write Protect’ and ‘/HOLD’ are not used for this experiment.

The M95080 has the following timing requirements:



To write data into the EEPROM the DSP has to generate the data bit first; with a clock delay of $\frac{1}{2}$ cycles the rising edge is the strobe pulse for the EEPROM to store the data. When reading the EEPROM the falling clock edge causes the EEPROM to send out data. With the rising clock edge the DSP can read the valid data bit. The passive state for the clock line is selectable to be high or low.

Aim of Lab 7B:

The Aim of the Lab 7B is to store the data byte derived from the 8 input switches (GPIO B15...B8) in EEPROM address 0x40 when button GPIO-D1 (yellow) is pushed. When button GPIO-D6(red) is pushed the data byte from EEPROM address 0x40 should be read back and shown at GPIO B7...B0 (8 LED's). The program should sample the two command buttons D1 and D6 every 200 ms, forced by CPU Timer0.

Lab 7B: EEPROM M95080

◆ Objective:

- Based on hardware of Zwickau Adapter Board
- Store the value of 8 input switches (GPIO – B15...B8) into EEPROM – Address 0x40 when command input button GPIO-D1 is pressed (low active).
- Read EEPROM-Address 0x40 and show its content on 8 LED's (GPIO-B7...B0) when command input button GPIO-D6 is pressed (low active).
- GPIO – D5 is EEPROM's chip select (/CS) at the Zwickau Adapter Board

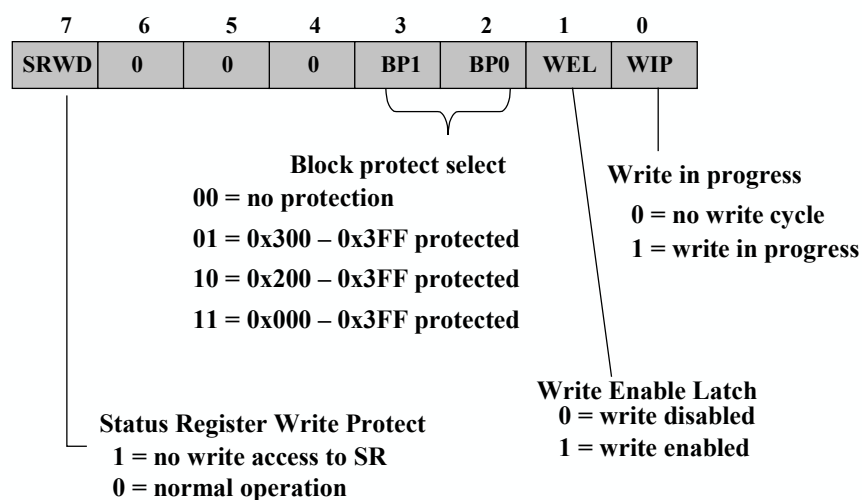
7 - 25

M95080 Status Register

The M95080 Status Register controls write accesses to the internal memory.

SPI Example : EEPROM M95080

◆ M95080 Status Register:



7 - 19

It also flags the current status of the EEPROM. Bit 0 (“WIP”) flags whether an internal write cycle is in progress or not. Internal write cycles are started at the end of a command sequence and last quite long (maximum 10ms). To avoid the interruption of a write cycle in progress any other write access should be delayed as long as WIP=1.

Bit 1(“Write Enable Latch”) is a control bit that must be set to 1 for every write access to the EEPROM. After a successful write cycle this bit is cleared by the EEPROM.

Bits 3 and 2(“Block Protect Select”) are used to define the area of memory that should be protected against any write access. We will not use any protection in our lab, so just set the two bits to ‘00’.

Bit 7 (“Status Register Write Protect”) allows us to disable any write access into the Status Register. For our Lab we will leave this bit cleared all the time (normal operation).

M95080 Instruction Set

To communicate with the M95080 we have to use the following table of instructions. An instruction is the first part of the serial sequence of data between the DSP and the EEPROM.

SPI Example : EEPROM M95080		
◆ M95080 Instruction Set:		
Instruction	Description	Code
WREN	Write Enable	0000 0110
WRDI	Write Disable	0000 0100
E	Read Status Register	0000 0101
WDSR	Write Status Register	0000 0001
READ	Read Data	0000 0011
WRITE	Write Data	0000 0010

7 - 20

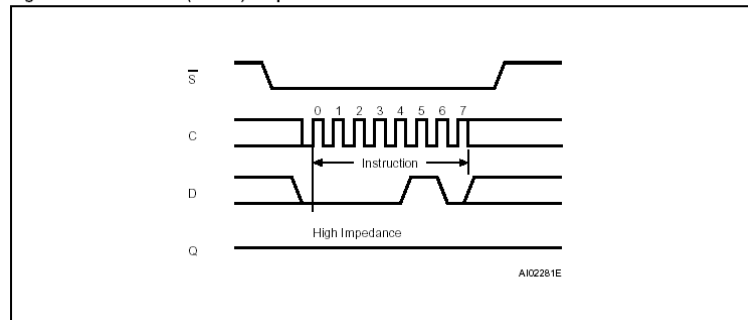
Before we can start our Lab procedure we have to discuss these instructions a little bit more in detail.

The **WREN** command must be applied to the EEPROM to the Write Enable Latch (WEL) **prior to each** WRITE and WRSR instruction. The command is an 8-clock SPI- sequence shown on the next slide:

SPI Example : EEPROM M95080

◆ Timing Diagram WREN:

Figure 7. Write Enable (WREN) Sequence



© ST Microelectronics ; Datasheet (8028.pdf) – November 2002; Page 8

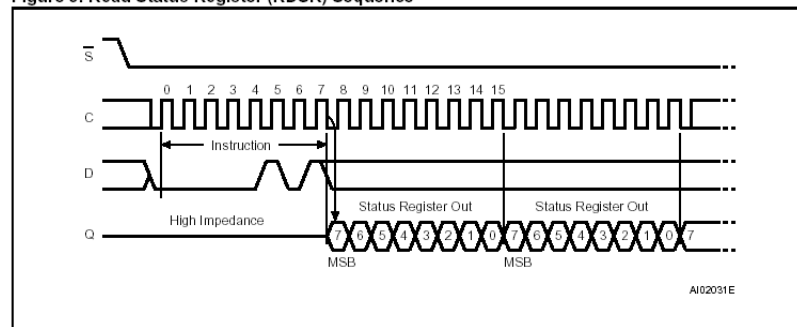
7 - 21

The **RDSR** instruction allows the Status Register to be read. The Status Register may be read any time. It is recommended to use this instruction to check the "Write In Progress" (WIP) bit **before** sending a new instruction to the EEPROM. This is also possible to read the Status Register continuously.

SPI Example : EEPROM M95080

◆ Timing Diagram RDSR:

Figure 9. Read Status Register (RDSR) Sequence



© ST Microelectronics ; Datasheet (8028.pdf) – November 2002; Page 10

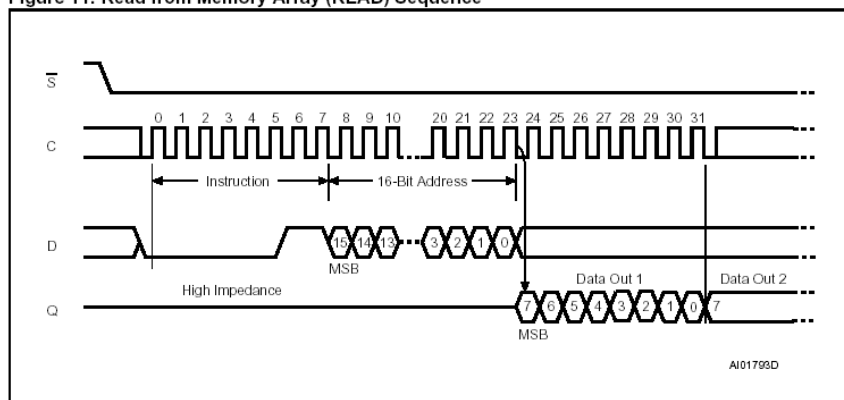
7 - 22

The **READ** instruction is used to read data out of the EEPROM. The address range of the M95080 is from 0 to 1023. The address is given as a full 16 bit address; bits A15 to A10 are don't cares. As shown with the next figure, an internal address counter is incremented with each READ instruction, if chip select (/S) continues to be driven low:

SPI Example : EEPROM M95080

◆ Timing Diagram READ:

Figure 11. Read from Memory Array (READ) Sequence



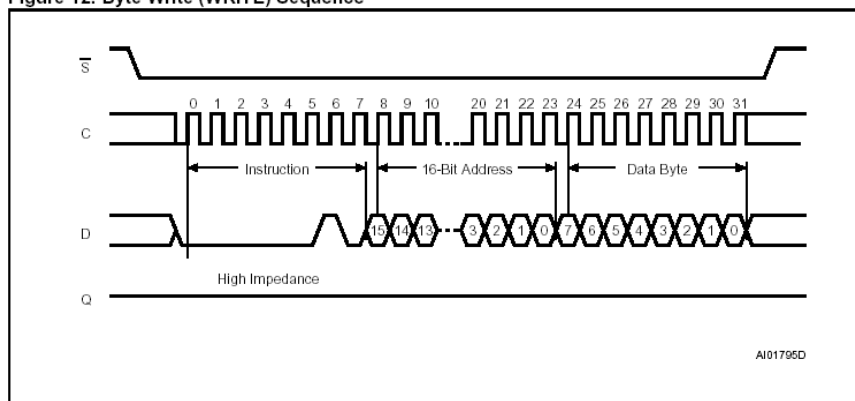
© ST Microelectronics ; Datasheet (8028.pdf) – November 2002; Page 13

7 - 23

SPI Example : EEPROM M95080

◆ Timing Diagram WRITE:

Figure 12. Byte Write (WRITE) Sequence



© ST Microelectronics ; Datasheet (8028.pdf) – November 2002; Page 14

7 - 24

The **WRITE** instruction is used to write data into the EEPROM. The instruction is terminated by driving Chip Select (/S) high. At this point the internal self timed write cycle actually starts, at the end of which the "Write In Progress "(WIP) bit of the Status Register is reset to 0.

Procedure Lab7B

Open Files, Create Project File

1. Create a new project, called **Lab7B.pjt** in E:\C281x\Labs.
2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab7B.c in E:\C281x\Labs\Lab7B.

3. Add the source code file to your project:

- **Lab7B.c**

4. From C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source add:

- **DSP281x_GlobalVariableDefs.c**

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

- **F2812_Headers_nonBIOS.cmd**

From C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd add:

- **F2812_EzDSP_RAM_Ink.cmd**

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

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**
- **DSP281x_CpuTimers.c**

From C:\ti\c2000\cgtoolslib add:

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Open Lab7B.c to edit: double click on “Lab7B.c” inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the main variables “LED[8]” and “i” for this exercise:

At beginning of main, delete the lines:

**unsigned int i;
unsigned int LED[8]= {0x0001,0x0002,0x0004,0x0008,
0x0010,0x0020,0x0040,0x0080};**

8. Next, inside the “while(1)”-loop of main reduce the code to just the following lines (we will add some more code later):

**while(1)
{

 while(CpuTimer0.InterruptCount < 3); // wait for Timer 0
 CpuTimer0.InterruptCount = 0;
 EALLOW;
 SysCtrlRegs.WDKEY = 0xAA; // and service watchdog #2
 EDIS;

}**

9. Before we continue to add the SPI modifications lets test if the project in its preliminary stage runs as expected. Recall, with the start code of Lab4 we initialized the CPU core timer 0 to generate an interrupt request every 50ms. The interrupt service routine “cpu_timer0_isr()” increments a global variable “CpuTimer0.InterruptCount” with every hit. If everything works as expected the DSP should hit the line

CpuTimer0.InterruptCount = 0;

in the while(1) – loop (procedure step 8) every 3*50ms = 150 ms.

Build and Load

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

Project → Build

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

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

File → Load Program and choose the desired output file.

Test

12. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart and
Debug → Go main.

13. In the “while(1)”-loop of main set a breakpoint at line:

CpuTimer0.InterruptCount = 0;

Run

14. Verify, that the breakpoint is hit periodically when you start the DSP by:

Debug → Run (F5).

Remove the breakpoint when you are done.

Add the SPI initialization code

15. So far we verified that the CPU Core Timer0 generates a period of 50 ms and that our main-loop waits for 3 periods of Timer0 before it moves to the next instruction. Now we need to add the code for the SPI to control the EEPROM M95080. First function that is called is “InitSystem()”. Again we have to enable the SPI clock unit. Inside “InitSystem()” set:

SysCtrlRegs.PCLKCR.bit.SPIENCLK=1;

16. The next function that is called in main is “Gpio_select()”. Inside this function we have to modify the multiplex register to use the four SPI-signals:

GpioMuxRegs.GPFMUX.all = 0xF;

Bits 3...0 control the setup for SPISTE, SPICLK, SPISOMI and SPISIMO. We also prepare GPIO-signal D0 and D5 to be digital outputs. D0 will be used as chip-select for the TLV5617A and D5 for the EEPROM M95080 (used in Lab7B). To do this, setup GPDDIR register:

`GpioMuxRegs.GPDDIR.all=0;`

`GpioMuxRegs.GPDDIR.bit.GPIOD0 = 1; // /CS for DAC TLV5617A`

`GpioMuxRegs.GPDDIR.bit.GPIOD5 = 1; // /CS for EEPROM`

As an initial state we should switch off both /CS-signals as well as the LED's at B7...B0:

`GpioDataRegs.GPBDAT.all = 0x0000; // Switch off LED's (B7...B0)`

`GpioDataRegs.GPDDAT.bit.GPIOD0 = 1; // /CS for DAC off`

`GpioDataRegs.GPDDAT.bit.GPIOD5 = 1; // /CS for EEPROM off`

17. In main, just before we enter the “while(1)”-loop add a function call to function “SPI_Init()”. Also add a function prototype at the start of your code.

At the end of your code, add the definition of function “SPI_Init()”.

Inside this function, include the following steps:

- SPICCR:
 - Clock polarity =1: data output at falling edge of clock
 - 16 bit per data frame
- SPICTL:
 - No SPI interrupts for this exercise
 - Master on
 - Talk enabled
 - Clock phase = 0: no phase shift
- SPIBRR:
 - $BRR = LSPCLK / SPI_Baudrate - 1$
 - Example: assuming $LSPCLK = 37.5MHz$ and $SPIBRR = 124$ the SPI-Baud rate is 300 kbps.

Create EEPROM access functions

18. Now we have to develop the code to access the SPI – EEPROM. A good method would be to write four specific functions to:

- Read the EEPROM Status Register:
 - `int SPI_EEPROM_Read_Status(void)`
- Set the Write Enable Latch:
 - `void SPI_EEPROM_Write_Enable(void)`
- Write 8 bit into the EEPROM:
 - `void SPI_EEPROM_Write(int address, int data)`
- Read 8 bit out of the EEPROM:
 - `int SPI_EEPROM_Read(int address)`

19. Function “**SPI_EEPROM_Read_Status**”

- At the beginning of the function activate the chip select (D5) signal for the EEPROM:

`GpioDataRegs.GPDDAT.bit.GPIOD5 = 0;`

- Next, load the code for “Read Status Register” into the SPI-Transmit buffer “SPITXBUF”. Take care of the correct alignment when loading this 8 bit code into “SPITXBUF”.
- Before we can continue we will have to wait for the end of the SPI transmission. Because we did disable all SPI interrupts we can’t use an interrupt driven synchronization, all we can do now is to ‘poll’ the SPI INT flag:

`while(SpiaRegs.SPISTS.bit.INT_FLAG == 0);`

- Now we can read the status out of SPIRXBUF and return it to the calling function.
- At the end of the function we have to deactivate the EEPROM’s chip select (D5).

20. Function “**SPI_EEPROM_Write_Enable**”

- This function is used to set the Write Enable Latch of the EEPROM and must be called before every write access.
- At the beginning of the function activate the chip select (D5) signal for the EEPROM.

- Next, load the code for “Write Enable” into the SPI-Transmit buffer “SPITXBUF”. Take care of the correct alignment when loading this 8 bit code into “SPITXBUF”.
- ‘Poll’ the SPI INT flag to wait for the end of SPI transmission.
- To reset the SPI INT flag we have to execute a “dummy”-read from SPIRXBUF:

i = SpiaRegs.SPIRXBUF;
- Switch off the EEPROM’s chip select (D5).

21. Function “SPI_EEPROM_Write”

- This function is used to write a character of 8 bit into the EEPROM. The input parameters are (1) the EEPROM – Address (16 bit integer) and (2) the data (16-bit integer – only the 8 least significant bits (LSB’s) are used)
- At the beginning of the function, activate the chip select (D5) signal for the EEPROM.
- Next, load the code for “Write” and the upper half of the address parameter into the SPI-Transmit buffer “SPITXBUF”.
- ‘Poll’ the SPI INT flag to wait for the end of SPI transmission.
- To reset the SPI INT flag do a “dummy”-read from SPIRXBUF.
- Next, load the lower half of the address parameter and the lower half of the data parameter into the SPI-Transmit buffer “SPITXBUF”.
- ‘Poll’ the SPI INT flag to wait for the end of SPI transmission.
- To reset the SPI INT flag once more, do a “dummy”-read from SPIRXBUF.
- Switch off the EEPROM’s chip select (D5).

22. Function “SPI_EEPROM_Read”

- This function returns a 16-bit integer (only lower 8 bits are used) to the calling function. The input parameter is the EEPROM address to be read (16-bit integer).
- At the beginning of the function, activate the chip select (D5) signal for the EEPROM.
- Next, load the code for “Read” and the upper half of the address parameter into the SPI-Transmit buffer “SPITXBUF”.
- ‘Poll’ the SPI INT flag to wait for the end of SPI transmission.
- Reset the SPI INT flag by a “dummy”-read from SPIRXBUF.
- Next, load the lower half of the address parameter into the SPI-Transmit buffer “SPITXBUF” (take care of correct alignment).
- ‘Poll’ the SPI INT flag to wait for the end of SPI transmission.
- Read SPIRXBUF and return its value to the calling function
- Switch off the EEPROM’s chip select (D5).

Finalize the main loop

23. Finally we have to add some function calls to our main loop to use the EEPROM. The objective of this lab exercise is to store the status of the 8 input switches (GPIO

B15-B8) into EEPROM address 0x40 when a first command button (GPIO D1) is pushed(D1 = 0). If a second command button (GPIO D6) is pushed(D6 = 0) the program should read EEPROM address 0x40 and load the value onto 8 LED's (GPIO B7-B0).

- To write into address 0x40:
 - Call function “SPI_EEPROM_Write_Enable”
 - Verify that flag “WEL” is set (= 1) by calling function “SPI_EEPROM_Read_Status”
 - Call function “SPI_EEPROM_Write”
 - Wait for the end of write by polling flag “WIP”. It will be cleared at the very end of an internal EEPROM write sequence. Use function “SPI_EEPROM_Read_Status” to poll “WIP” periodically.
- To read from address 0x40:
 - Call function “SPI_EEPROM_Read()” and transfer the return value onto GPIO B7-B0 (8 LED's).

Build, Load and Run

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

Project → Build
File → Load Program
Debug → Reset CPU
Debug → Restart
Debug → Go main
Debug → Run (F5)

25. Modify the input switches and push GPIO-D1. Next, push GPIO-D6. The LED's should mirror the last state of the input switches.
26. Close Code Composer Studio and switch off the eZdsp. After a few seconds re-power the board and start Code Composer Studio. Download the project into the DSP, run it and push the read button D6 first. Now the LED's should display the last value that has been stored inside the EEPROM before the power has been switched off. (An EEPROM is a non volatile memory that keeps the information also when power supply has been switched off).

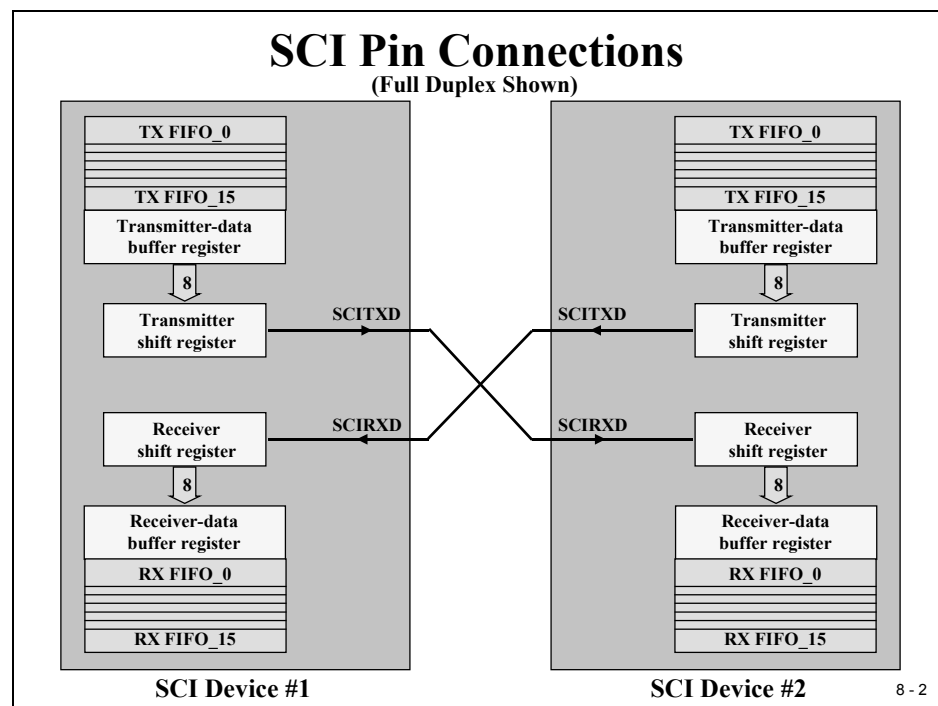
END of LAB 7B

This page was intentionally left blank.

C28x Serial Communication Interface

Introduction

The Serial Communication Interface (SCI) module is a serial I/O port that permits asynchronous communication between the C28x and other peripheral devices. It is usually known as a UART (Universal Asynchronous Receiver Transmitter) and is used according to the RS232 standard. To allow for efficient CPU usage, the SCI transmit and receive registers are both FIFO-buffered to prevent data collisions. In addition, the C28x SCI has a full duplex interface, which provides for simultaneous data transmit and receive. Parity checking and data formatting can also be done by the SCI port hardware, further reducing the software overhead.



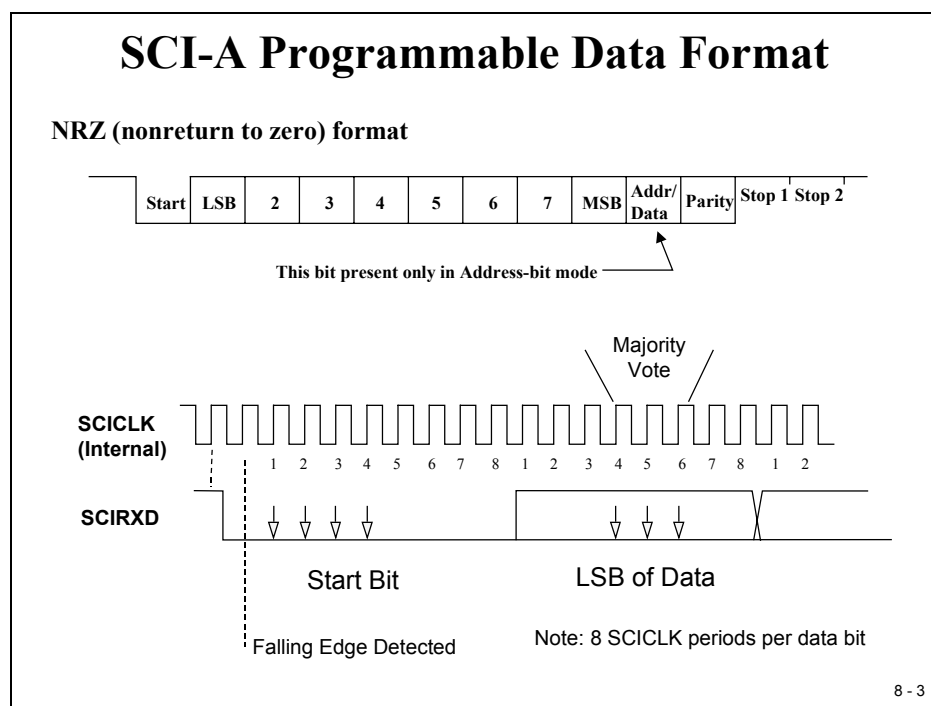
Module Topics

C28x Serial Communication Interface	8-1
<i>Introduction</i>	<i>8-1</i>
<i>Module Topics.....</i>	<i>8-2</i>
<i>SCI Data Format.....</i>	<i>8-3</i>
<i>SCI Multi Processor Wake Up Modes</i>	<i>8-4</i>
<i>SCI Register Set</i>	<i>8-6</i>
SCI Communications Control Register (SCICCR).....	8-7
SCI Control Register 1(SCICTL1)	8-7
SCI Baud Rate Register	8-8
SCI Control Register 2 – SCICTL2.....	8-9
SCI Receiver Status Register – SCIRXST	8-10
SCI FIFO Mode Register.....	8-11
<i>Lab 8: Basic SCI – Transmission.....</i>	<i>8-13</i>
Objective	8-13
Procedure	8-14
Open Files, Create Project File.....	8-14
Project Build Options	8-14
Modify Source Code.....	8-15
Add the SCI initialization code.....	8-15
Finish the main loop	8-16
Build, Load and Run.....	8-16
<i>Lab 8A: Interrupt SCI – Transmission.....</i>	<i>8-17</i>
Procedure.....	8-17
Open Files, Create Project File.....	8-17
Project Build Options	8-18
Modify Source Code.....	8-18
Build and Load	8-20
Test & Run	8-21
Optional Exercise	8-21
<i>Lab 8B: SCI – FIFO Transmission</i>	<i>8-22</i>
Procedure.....	8-22
Open Files, Create Project File.....	8-22
Project Build Options	8-23
Modify Source Code.....	8-23
Build, Load and Test	8-24
<i>Lab 8C: SCI – Receive & Transmit.....</i>	<i>8-25</i>
Procedure.....	8-25
Open Files, Create Project File.....	8-25
Project Build Options	8-26
Modify Source Code.....	8-26
Build, Load and Test	8-28
Optional Exercise	8-28

SCI Data Format

The basic unit of data is called a **character** and is 1 to 8 bits in length. Each character of data is formatted with a start bit, 1 or 2 stop bits, an optional parity bit, and an optional address/data bit. A character of data along with its formatting bits is called a **frame**. Frames are organized into groups called blocks. If more than two serial ports exist on the SCI bus, a block of data will usually begin with an address frame, which specifies the destination port of the data as determined by the user's protocol.

The start bit is a low bit at the beginning of each frame, which marks the start of a frame. The SCI uses a NRZ (Non-Return-to-Zero) format, which means that in an inactive state the SCIRX and SCITX lines will be held high. Peripherals are expected to pull the SCIRX and SCITX lines to a high level when they are not receiving or transmitting on their respective lines.



SCI Multi Processor Wake Up Modes

Multiprocessor Wake-Up Modes

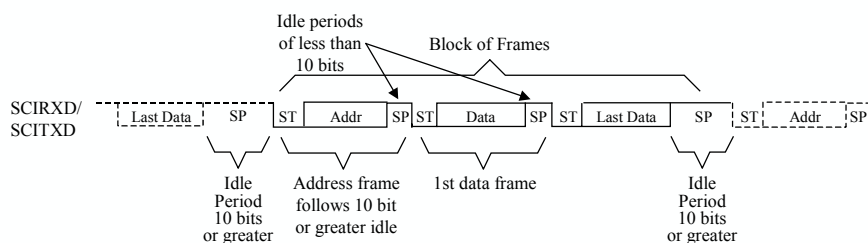
- ◆ **Allows numerous processors to be hooked up to the bus, but transmission occurs between only two of them**
- ◆ **Idle-line or Address-bit modes**
- ◆ **Sequence of Operation**
 1. Potential receivers set SLEEP = 1, which disables RXINT except when an address frame is received
 2. All transmissions begin with an address frame
 3. Incoming address frame temporarily wakes up all SCIs on bus
 4. CPUs compare incoming SCI address to their SCI address
 5. Process following data frames only if address matches

8 - 4

Although a SCI data transfer is usually a point-to-point communication, the C28x SCI interface allows two operation modes to communicate between a master and more than one slave.

Idle-Line Wake-Up Mode

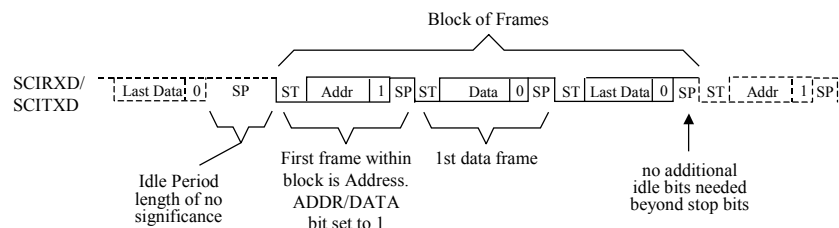
- ◆ **Idle time separates blocks of frames**
- ◆ **Receiver wakes up with falling edge after SCIRXD was high for 10 or more bit periods**
- ◆ **Two transmit address methods**
 - ◆ **deliberate software delay of 10 or more bits**
 - ◆ **set TXWAKE bit to automatically leave exactly 11 idle bits**



8 - 5

Address-Bit Wake-Up Mode

- ◆ All frames contain an extra address bit
- ◆ Receiver wakes up when address bit detected
- ◆ Automatic setting of Addr/Data bit in frame by setting TXWAKE = 1 prior to writing address to SCITXBUF



8 - 6

SCI Summary

- ◆ Asynchronous communications format
- ◆ 65,000+ different programmable baud rates
- ◆ Two wake-up multiprocessor modes
 - Idle-line wake-up & Address-bit wake-up
- ◆ Programmable data word format
 - 1 to 8 bit data word length
 - 1 or 2 stop bits
 - even/odd/no parity
- ◆ Error Detection Flags
 - Parity error; Framing error; Overrun error; Break detection
- ◆ FIFO-buffered transmit and receive
- ◆ Individual interrupts for transmit and receive

8 - 7

SCI Register Set

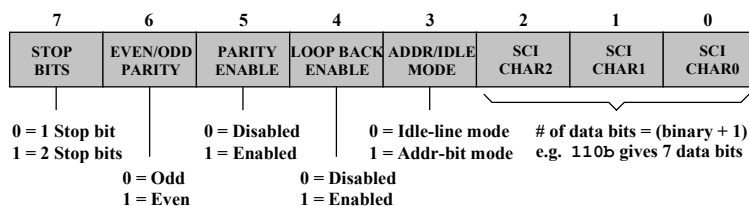
The next slide summarizes all SCI control registers for SCI channel A. Note that there is a second SCI channel B available in the C28x.

SCI-A Registers		
Address	Register	Name
0x007050	SCICCR	SCI-A commun. control register
0x007051	SCICTL1	SCI-A control register 1
0x007052	SCIHBAUD	SCI-A baud register, high byte
0x007053	SCILBAUD	SCI-A baud register, low byte
0x007054	SCICTL2	SCI-A control register 2 register
0x007055	SCIRXST	SCI-A receive status register
0x007056	SCIRXEMU	SCI-A receive emulation data buffer
0x007057	SCIRXBUF	SCI-A receive data buffer register
0x007059	SCITXBUF	SCI-A transmit data buffer register
0x00705A	SCIFFTX	SCI-A FIFO transmit register
0x00705B	SCIFFRX	SCI-A FIFO receive register
0x00705C	SCIFFCT	SCI-A FIFO control register
0x00705F	SCIPRI	SCI-A priority control register

8 - 8

SCI-A Communication Control Register

Communications Control Register (SCICCR) – 0x007050



[SCI-B Communications Control Register (SCICCR) – 0x007750]

8 - 9

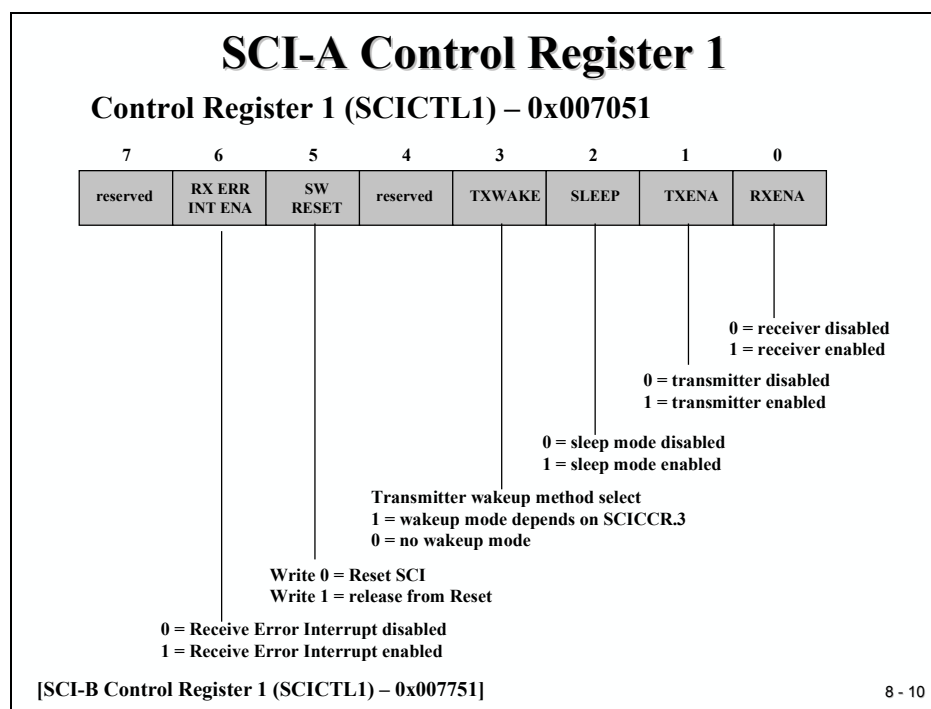
SCI Communications Control Register (SCICCR)

The previous slide explains the setup for the SCI data frame structure. If Multi Processor Wakeup Mode is not used, bit 3 should be cleared. This avoids the generation of an additional address/data selection bit at the end of the data frame (see slide 8-3). Some hosts or other devices are not able to handle this additional bit.

The other bit fields of SCICCR can be initialized, as you like. For our lab exercises in this chapter we will use:

- 8 data bit per character
- no parity
- 1 Stop bit
- loop back disabled

SCI Control Register 1(SCICTL1)



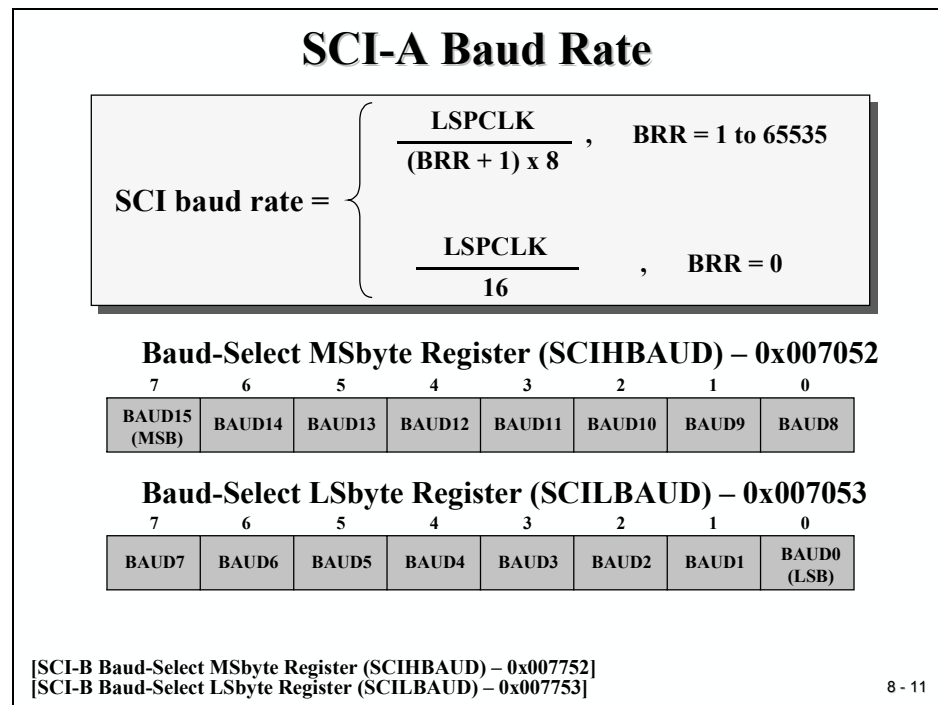
When configuring the SCICCR, the SCI port should first be held in an inactive state. This is done using the SW RESET bit of the SCI Control Register 1 (SCICTL1.5). Writing a 0 to this bit initializes and holds the SCI state machines and operating flags at their reset condition. The SCICCR can then be configured. Afterwards, re-enable the SCI port by writing a 1 to the SW RESET bit. At system reset, the SW RESET bit equals 0.

For our Lab exercises we will not use wakeup or sleep features (SCICTL1.3 = 0 and SCICTL1.2 = 0).

Depending on the direction of the communication we will have to enable the transmitter (SCICTL1.1 = 1) or the receiver (SCICTL1.0 = 1) or both.

For a real project we would have to think about potential communication errors. The receiver error could then be allowed to generate a receiver error interrupt request (SCICTL1.6 = 1). Our first labs will not use this feature.

SCI Baud Rate Register



The baud rate for the SCI is derived from the low speed pre-scaler (LSPCLK).

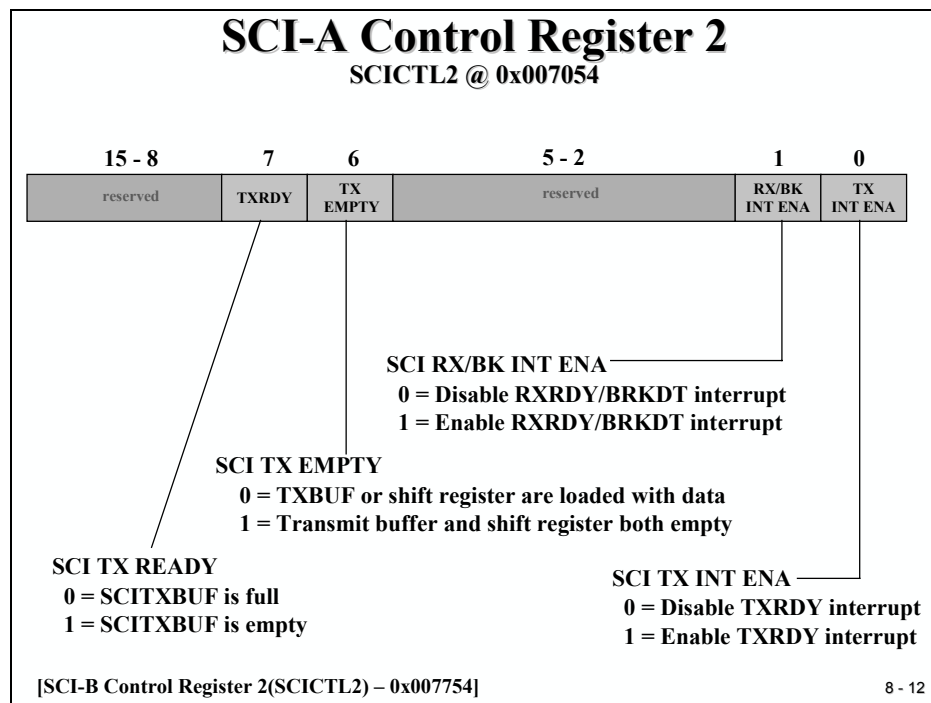
Assuming a SYSCLK frequency of 150MHz and a low speed pre-scaler initialized to “divide by 4” we can calculate the set up for BRR to initialize, let’s say a baud rate of 9600 baud:

$$9.600\text{Hz} = \frac{37.5\text{MHz}}{(\text{BRR} + 1) * 8}$$

$$\text{BRR} = \frac{37.5\text{MHz}}{9.600\text{Hz} * 8} - 1 = 487.2$$

BRR must be an integer, so we have to round the result to 487. The reverse calculation with BRR = 487 leads to the real baud rate of 9605.5 baud (error = 0,06 %).

SCI Control Register 2 – SCICTL2

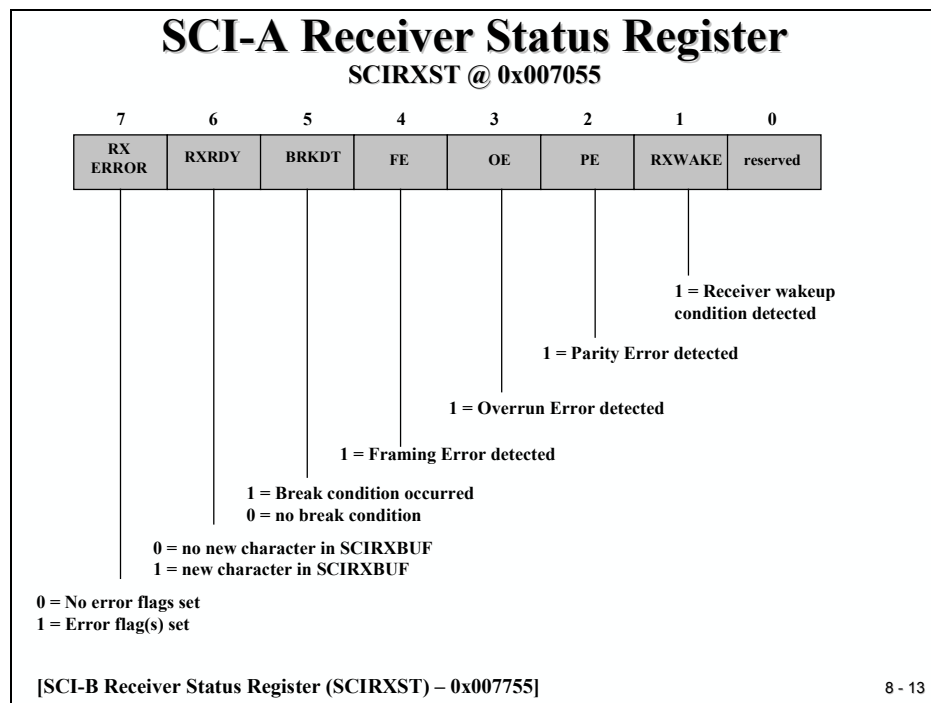


Bit 1 and 0 enable or disable the SCI- transmit and receive interrupts. If interrupts are not used, we can disable this feature by clearing bits 1 and 0. In this case we have to apply a polling scheme to the transmitter status flags (SCICTL2.7 and SCICTL2.6). The flag SCITXEMPTY waits until the whole data frame has left the SCI output, whereas flag SCITXREADY indicates the situation that we can reload the next character into SCITXBUF before the previous character was physically sent.

The status flags for the receiver part can be found in the SCI receiver status register (see next slide).

For the first basic lab exercise we will not use SCI interrupts. This means we have to rely on a polling scheme. Later we will include SCI interrupts in our experiments.

SCI Receiver Status Register – SCIRXST

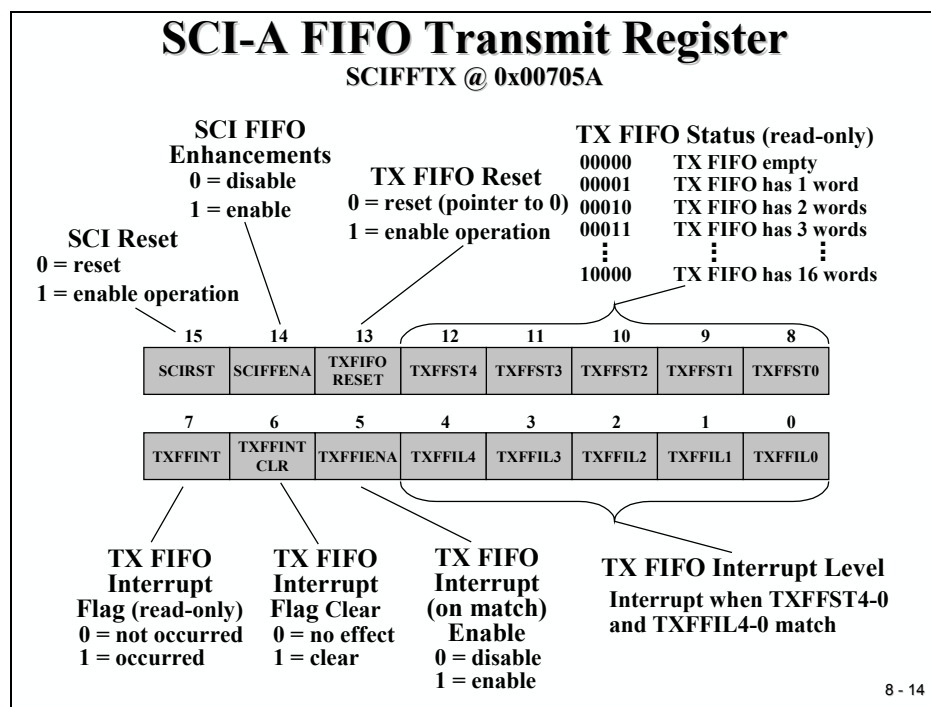


The SCI interrupt logic generates interrupt flags when it receives or transmits a complete character as determined by the SCI character length. This provides a convenient and efficient way of timing and controlling the operation of the SCI transmitter and receiver. The interrupt flag for the transmitter is TXRDY (SCICTL2.7), and for the receiver RXRDY (SCIRXST.6). TXRDY is set when a character is transferred to TXSHF and SCITXBUF is ready to receive the next character. In addition, when both the SCIBUF and TXSHF registers are empty, the TX EMPTY flag (SCICTL2.6) is set.

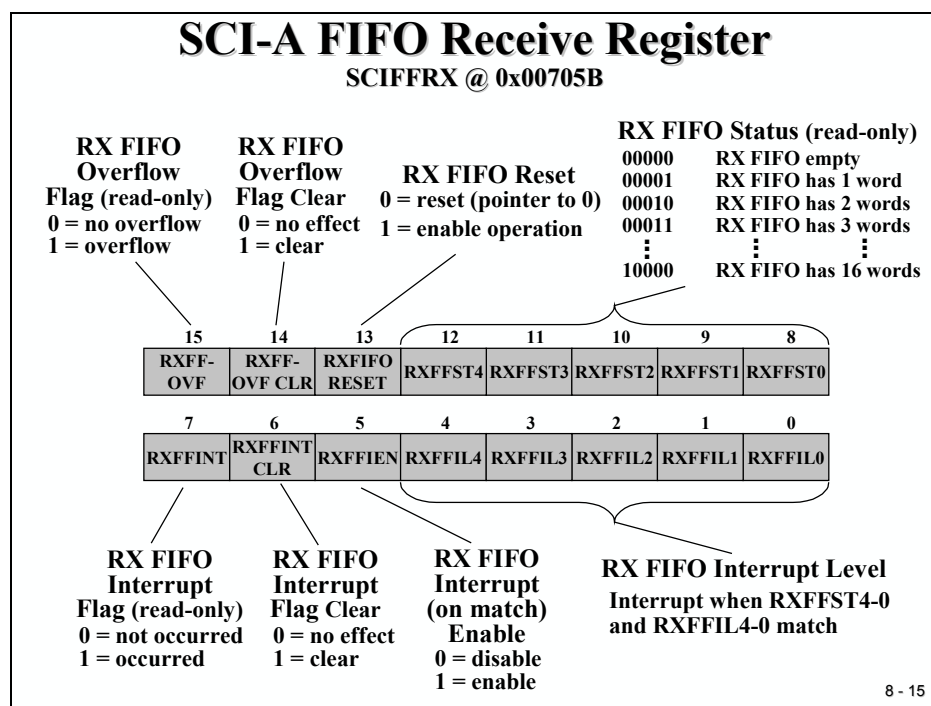
When a new character has been received and shifted into SCIRXBUF, the RXRDY flag is set. In addition, the BRKDT flag is set if a break condition occurs. A break condition is where the SCIRXD line remains continuously low for at least ten bits after a stop bit has been missed. The CPU to control SCI operations can poll each of the above flags, or interrupts associated with the flags can be enabled by setting the RX/BK INT ENA (SCICTL2.1) and/or the TX INT ENA (SCICTL2.0) bits active high.

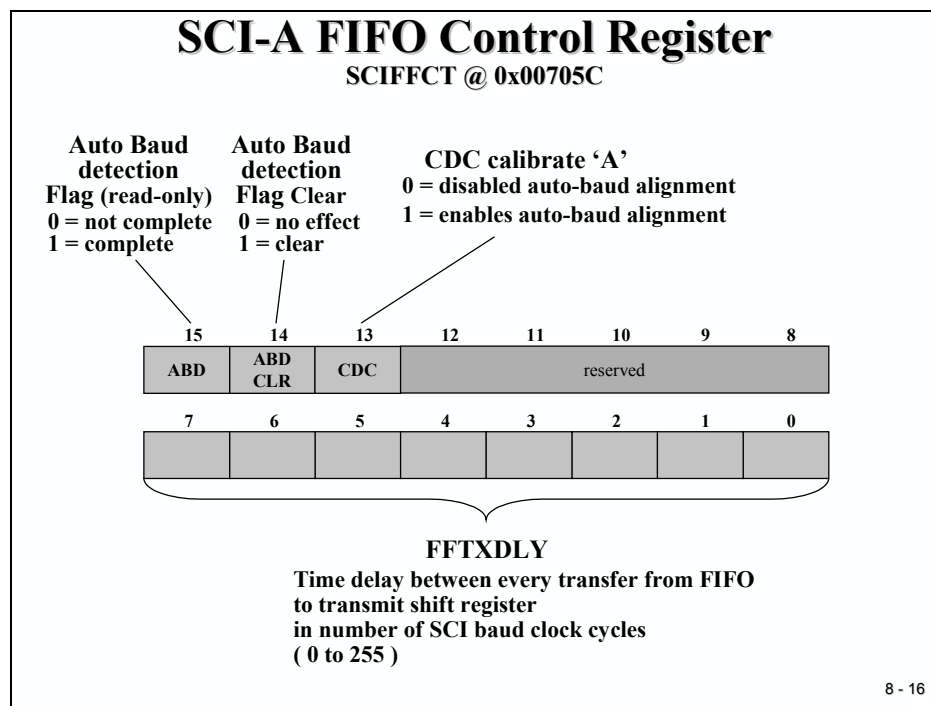
Additional flag and interrupt capability exists for other receiver errors. The RX ERROR flag is the logical OR of the break detect (BRKDT), framing error (FE), receiver overrun (OE), and parity error (PE) bits. RX ERROR high indicates that at least one of these four errors has occurred during transmission. This will also send an interrupt request to the CPU if the RX ERR INT ENA (SCICTL1.6) bit is set.

SCI FIFO Mode Register



The C28x SCI is equipped with an enhanced buffer mode with 16 levels for the transmitter and receiver. We will use this enhanced mode at the end of the lab exercise series of this chapter.





Lab 8: Basic SCI – Transmission

SCI Example 1: transmit a text - string

◆ Lab 8: Basic SCI Communication

- ◆ Send a string from DSP to a PC's COM-port.
- ◆ Connect the RS232 - Connector of the Zwickau adapter board with a standard DB9 - cable (1:1) to a serial port of the PC (COM1 or COM2).
- ◆ DSP shall transmit a string from the DSP to the PC periodically.
- ◆ No SCI interrupt services in this lab
- ◆ After transmission of the first character we just poll the transmission ready flag (TXEMPTY) before loading the next character into the transmit buffer - and wait again.
- ◆ The Windows-Hyper Terminal program is used as the counterpart from the PC's-side and must be initialized properly for correct function (Baud rate, Parity, no protocol).

8 - 17

Objective

The objective of this lab is to establish an SCI transmission between the C28x and a serial port of a PC. The Zwickau adapter board converts the serial signals of the DSP into a standard RS232 format. The DB9 female connector (X1) has to be connected with a standard serial cable (1:1 connection) to a serial COM-port of the PC.

The DSP shall transmit a string, e.g. "The F2812-UART is fine!\n\r" periodically. No interrupt services are used for this basic test.

As the counterpart at the PC we will use Windows® - hyper terminal program. This program can be found under Windows - OS → start → programs → Accessories → Communication → HyperTerminal. Create a new connection, select a symbol and name it "SCI-Test".

In the "connect to" field select the COM-port of your PC, e.g. COM1.

Setup:

- data rate to 9600,
- 8 bits per character,
- no parity bit,
- 1 stopbit,
- no protocol.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8.pjt** in E:\C281x\Labs.
2. Open the file Lab2.c from E:\C281x\Labs\Lab2 and save it as Lab8.c in E:\C281x\Labs\Lab8.
3. Add the source code file to your project:
 - **Lab8.c**
4. From C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source add:

- **DSP281x_GlobalVariableDefs.c**

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

- **F2812_Headers_nonBIOS.cmd**

From C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd add:

- **F2812_EzDSP_RAM_Ink.cmd**

From C:\ti\c2000\cgtoolslib add:

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Open Lab8.c to edit: double click on “Lab8.c” inside the project window. First we have to cancel the parts of the code that we do not need any longer. We will not use the main variables “LED[8]” and “i” for this exercise:

At the beginning of main, delete the lines:

```
unsigned int i;  
unsigned int LED[8]= {0x0001,0x0002,0x0004,0x0008,  
0x0010,0x0020,0x0040,0x0080};
```

8. Next, empty the “while(1)”-loop of main, we will add some more code later:

```
while(1)  
{  
  
}
```

9. Delete the function “delay_loop” and its prototype declaration.

Add the SCI initialization code

10. Inside function “InitSystem()” enable the SCI-A clock unit:

```
SysCtrlRegs.PCLKCR.bit.SCIAENCLK=1;
```

11. Inside “Gpio_select()” modify multiplex register GPFMUX to use the 2 SCI-signals:

```
GpioMuxRegs.GPFMUX.bit.SCIRXDA_GPIOF5 = 1;
```

```
GpioMuxRegs.GPFMUX.bit.SCITXDA_GPIOF4 = 1;
```

12. At the beginning of main define a string variable with the following text in it:

```
"The F2812-UART is fine !\n\r"
```

13. In main, just before entering the “while(1)”-loop add a function call to function “SCI_Init()”. Also add a function prototype at the start of your code.

14. At the end of your code, add the definition of function “SCI_Init()”.

Inside this function, include the following steps:

- SCICCR:
 - 1 stop bit, no loop back, no parity, 8 bits per character
- SCICTL1:
 - Enable TX, RX -output
 - Disable RXERR INT, SLEEP and TXWAKE

- SCIHBAUD / SCILBAUD:
 - $BRR = (LSPCLK / (SCI_Baudrate * 8)) - 1$
 - Example: assuming $LSPCLK = 37.5\text{MHz}$ and $SCI_Baudrate = 9600$ the SCIBRR must be set to 487.

Finish the main loop

15. Now we can finalize the while(1)-loop of main. Recall, we have to add the following:

- Load the next character out of the string variable into SCITXBUF
- Wait (poll) bit TXEMPTY of register SCICTL2. It will be set to 1 when the character has been sent. The bit will be cleared automatically when the next character is written into SCITXBUF.
- Increment an array pointer to point to the next character of the string. Also include a test if the whole string has been sent. In this case reset the pointer to prepare the next transmission sequence.
- Include a software loop before the start of the next transmission sequence of approximately 2 seconds:

`for(i=0;i<15000000;i++);` *// Software - delay approx. 2 sec.*

- Service the watchdog periodically

Build, Load and Run

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

Project → Build
File → Load Program
Debug → Reset CPU
Debug → Restart
Debug → Go main
Debug → Run (F5)

17. In the hyper terminal window you should see the received string every 2 seconds.

If not → Debug!

END of LAB 8

Lab 8A: Interrupt SCI – Transmission

The objective of the next lab exercise is to improve Lab 8 by including both the SCI – Transmit interrupt service to service an empty transmit buffer and the CPU Core Timer 0 interrupt service to trigger the transmission of the first character of the string every 2 seconds.

Use your code from Lab8 as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8A.pjt** in E:\C281x\Labs.
2. Open the file Lab8.c from E:\C281x\Labs\Lab8 and save it as Lab8A.c in E:\C281x\Labs\Lab8A.
3. Add the source code file to your project:
 - **Lab8A.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:\tidcs\c28\dsp281x\v100\DSP281x_common\source add to project:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**
- **DSP281x_CpuTimers.c**
- **DSP281x_usDelay.asm**

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

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Open Lab8A.c to edit: double click on “Lab8A.c” inside the project window. In main, after the function call “Gpio_select()” add a call of function:

InitPieCtrl();

and:

InitPieVectTable();

Next, re-map the Interrupt table entry for CPU Core Timer 0 Interrupt

EALLOW;

PieVectTable.TINT0 = &cpu_timer0_isr;

EDIS;

Initialize the CPU Core Timer group by calling:

InitCpuTimers();

and configure CPU-Timer 0 to interrupt every 50 ms:

ConfigCpuTimer(&CpuTimer0, 150, 50000);

Now enable the CpuTimer0 PIE interrupt line:

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

```
IER = 1;
```

Finally enable the global interrupt flag and reset start the CPU Core Timer 0:

```
EINT; // Enable Global interrupt INTM
```

```
ERTM; // Enable Global real-time interrupt DBGM
```

```
CpuTimer0Regs.TCR.bit.TSS = 0;
```

8. Add a CPU Core Timer0 Interrupt service routine “cpu_timer0_isr()” at the very end of your source code. You can use the same code that we used in Lab7B.c for this function. Do not forget to add a function prototype at the beginning of your code.
9. Next, we have to modify the SCI initialization function “SCI_Init()”. Not a big change, the only modification is that for this project we have to enable the SCI-Transmit Interrupt:

```
SciaRegs.SCICTL2.bit.TXINTENA = 1;
```

10. The SCI Transmit Interrupt must be also enabled inside the PIE and the address of the interrupt service routine must be written into the PIE vector table. Before the global interrupt enable line “EINT” add the following code:

```
EALLOW;
```

```
PieVectTable.TXAINT = &SCI_TX_isr;
```

```
EDIS;
```

```
PieCtrlRegs.PIEIER9.bit.INTx2 = 1;
```

```
IER |= 0x100;
```

11. If the SCI-TX interrupt is enabled we have to provide an interrupt service routine “SCI_TX_isr()”. At the top of your code add a function prototype and at the very end of the code add the definition of this function. What should be done inside this function? Answer:
 - Load the next character of the string into SCITXBUF, if the string pointer has not already reached the last character of the string and increment this pointer.
 - If the string pointer points beyond the last character of the string, do NOT load anything into SCITXBUF. Transmission of the string is finished.

- In every single call of this function acknowledge it's call by resetting the PIEACK-register:

PieCtrlRegs.PIEACK.all = 0x0100;

12. Because the string variable and the variable "index" are now used out of main and "SCI_TX_isr" they must now be declared as global variables. Remove the definition from main and add them as global variables at the beginning of your code, outside any function:

char message[]= {"The F2812-UART is fine !\n\r"};

int index =0; // pointer into string

13. Now it is time to think about the while(1) – loop of main. Remove everything that is inside this loop. We do not need the old code from Lab8.

Instead, let's add new code:

First, we will get a CPU Timer0 Interrupt every 50ms. According to the setup of our interrupt service routine "cpu_timer0_isr()" variable "CpuTimer0.InterruptCount" will be incremented every 50ms. To trigger a SCI Transmission every 2 seconds we just have to wait until this variable has reached 40. If the value is less than 40 we just have to wait and do nothing, right? NO! Our watchdog is alive and we have to serve it! If the first reset key is applied inside function "cpu_timer0_isr()" we have to apply the second key while we wait for variable "CpuTimer0.InterruptCount" to reach 40.

This could be the portion of code:

```
while(CpuTimer0.InterruptCount < 40) // wait for 2000ms
{
    EALLOW;
    SysCtrlRegs.WDKEY = 0xAA;    // service watchdog #2
    EDIS;
}
```

What's next? Well, these actions are required now:

- Reset variable "CpuTimer0.InterruptCount" to 0
- Reset variable "index" to 0
- Load first character out of message into SCITXBUF and
- Increment variable index afterwards.

Build and Load

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

Project → Build

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

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

File → Load Program and choose the desired output file.

Test & Run

16. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart and
Debug → Go main
Debug → Run (F5)

As we've done in Lab8 open a Hyper Terminal Session (use 9600, n, 1 and no protocol as parameters). Every 2 seconds you should receive the string from the DSP.

If your code does not work try to debug systematically.

- Does the CPU core timer work?
- Is the CPU core timer interrupt service called periodically?
- Is the SCITX interrupt service called?

Try to watch important variables and set breakpoints as needed.

17. The result of this lab does not differ that much from lab8. All we do is to send a string every 2 seconds to the PC. The big difference however is that the time interval is now generated by a hardware timer instead of a software delay loop. This is a big improvement, because the period is now very precise and the DSP is not overloaded by such a stupid task to count a variable from x to y. For real projects we would gain a lot of CPU time by using a hardware timer.

Optional Exercise

18. Instead of transmitting the string to the PC your task is now to transmit the current status of the input switches (GPIO B15-B8), which is an integer, to the PC. Recall, to use Windows Hyper Terminal to display data, you must transmit ASCII-code characters. To convert a long integer into an ASCII-string we can use function "ltoa"(see help).

END of LAB 8A

Lab 8B: SCI – FIFO Transmission

The objective of this lab is to improve Lab 8A by using the transmit FIFO capabilities of the C28x. Instead of generating a lot of SCI – transmit interrupts to send the whole string we now will use a type of ‘burst transmit’ technique to fill up to 16 characters into the SCI transmit FIFO. This will reduce the number of SCI-interrupt services from 16 to 1 per string transmission!

Use your code from Lab8A as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8B.pjt** in E:\C281x\Labs.
2. Open the file Lab8A.c from E:\C281x\Labs\Lab8A and save it as Lab8B.c in E:\C281x\Labs\Lab8B.
3. Add the source code file to your project:
 - **Lab8B.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:\tidcs\c28\dsp281x\v100\DSP281x_common\source add to project:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**
- **DSP281x_CpuTimers.c**
- **DSP281x_usDelay.asm**

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

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Open Lab8B.c to edit: double click on “Lab8B.c” inside the project window.

Modify the SCI Initialization in function “SCI_Init()”. Add the Initialization for register “SCIFTX”. Include the following:

- Relinquish FIFO unit from reset
 - Enable FIFO- Enhancements
 - Enable TX FIFO Operation
 - Clear TXFFINT-Flag
 - Enable TX FIFO match
 - Set FIFO interrupt level to interrupt, if FIFO is empty (0)
8. Change the content of variable “message[]” from “The F2812-UART is fine !\n\r” into “BURST-Transmit\n\r”. The length of the string is now limited to 16 characters and using the TX-FIFO we can transmit the whole string in one single SCI interrupt service routine.

9. Go into function “SCI_TX_isr()” and modify it. Recall that this service will be called when the FIFO interrupt level was hit. Due to our set up of this level to 0 we can load 16 characters into the TX-FIFO:

for(i=0;i<16;i++) SciaRegs.SCITXBUF = message[i];

Note: Variable i should be a local variable inside “SCI_TX_isr()”. Also, do NOT remove the PIEACK- reset instruction at the end of this function!

10. Go into the while(1)-loop of main. We still will use the CPU Core Timer 0 as our time base. It is still initialized to increment variable “CpuTimer0.InterruptCount” every 50ms. No need to change our wait construction to wait for 40 increments (equals to 2 seconds).

Delete the next two lines of the old code:

index = 0;

SciaRegs.SCITXBUF = message[index++];

The difference between Lab8A.c and Lab8B.c is the initialization of the SCI-unit. In this lab we enabled the TX-FIFO interrupt to request a service when the FIFO-level is zero. This will be true immediately after the initialization of the SCI-unit and will cause the first TX-interrupt! The next TX-interrupt will be called only after setting the TX FIFO INT CLR – bit to 1, clears the TX FIFO INT FLAG. If we execute this clear instruction every 2 seconds we will allow the next TX FIFO transmission to take place at this very moment. To do so, add the following instruction:

SciaRegs.SCIFFTX.bit.TXINTCLR = 1;

That’s it.

Build, Load and Test

11. Apply all the commands needed to translate and debug your project. Meanwhile you should be familiar with the individual steps to do so; therefore we skip a detailed procedure. If you are successful, you should receive the string every 2 seconds at the hyper terminal window. If not – debug!
12. Again, the big improvement of this Lab8B is that we reduced the number of interrupt services to transmit a 16-character string from 16 services to 1 service. This adds up to a considerable amount of time that can be saved! The exercise has shown the advantage of the C28x SCI-transmit FIFO enhancement compared to a standard UART interface.

END of LAB 8B

Lab 8C: SCI – Receive & Transmit

The final project in this module asks you to include the SCI receiver in our lab exercises. The objective of this lab is to receive a string “Texas” from the PC and to answer it by transmitting “Instruments” back to the PC.

Use your code from Lab8B as a starting point.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab8C.pjt** in E:\C281x\Labs.
2. Open the file Lab8B.c from E:\C281x\Labs\Lab8B and save it as Lab8C.c in E:\C281x\Labs\Lab8C.
3. Add the source code file to your project:
 - **Lab8C.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:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**

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

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Open Lab8C.c to edit.

First we have to remove everything that deals with CPU Core Timer0 – we do not need a timer for this exercise.

- Remove prototype and definition of function “cpu_timer0_isr”.
- In main, remove the overload instruction:

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

- Remove the function calls:

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

and the interrupt enable lines:

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

- Remove the start instruction for CpuTimer0:

```
CpuTimer0Regs.TCR.bit.TSS = 0;
```

- In the while(1)-loop of main remove everything. Replace it by:

```
while(1)
{
    EALLOW;
    SysCtrlRegs.WDKEY = 0x55;    // service watchdog #1
    SysCtrlRegs.WDKEY = 0xAA;    // service watchdog #2
    EDIS;
}
```

All activities will be done by interrupt service routines, nothing to do in the main loop but to service the watchdog!

8. Change the content of variable “message[]” into “ Instruments\n\r”.
9. Now we need to introduce a new interrupt service routine for the SCI receiver, called “SCI_RX_isr”. Declare it’s prototype at the beginning of your code:

interrupt void SCI_RX_isr(void);

10. Add an overload instruction for this function inside the PIE vector table. Add this line directly after the overload for TXAINT:

PieVectTable.RXAINT = &SCI_RX_isr;

11. Enable the PIE interrupt for RXAINT:

PieCtrlRegs.PIEIER9.bit.INTx1 = 1;

12. Modify the initialization function for the SCI: “SCI_Init()”.

- Inside register “SCICTL2” set bit “RXBKINTENA” to 1 to enable the receiver interrupt.
- For register “SCIFFTX” do NOT enable the TX FIFO operation (bit 13) yet. It will be enabled later, when we have something to transmit.
- Add the initialization for register “SCIFFRX”. Recall, we wait for 5 characters “Texas”, so why not initialize the FIFO receive interrupt level to 5? This setup will cause the RX interrupt when at least 5 characters have been received.

13. At the end of your source code add interrupt function “SCI_RX_isr()”.

- What should be done inside? Well, this interrupt service will be requested if 5 characters have been received. First we need to verify that the 5 characters match the string “Texas”.
- With five consecutive read instructions of register “SCIRXBUF” you can empty the FIFO into a local variable “buffer[16]”.

- The C standard function “strncmp” can be used to compare two strings of a fixed length. The lines:

```
if( strncmp(buffer, “Texas” , 5) == 0)
{
    SciaRegs.SCIFFTX.bit.TXFIFORESET = 1;
    SciaRegs.SCIFFTX.bit.TXINTCLR = 1;
}
```

will compare the first 5 characters of “buffer” with “Texas”. If they match the two next instructions will start the SCI Transmission of “ Instruments\n\r” with the help of the TX-interrupt service.

- At the end of interrupt service routine we need to reset the RX FIFO, clear the RX FIFO Interrupt flag and acknowledge the PIE interrupt:

```
SciaRegs.SCIFFRX.bit.RXFIFORESET = 0; // reset pointer
SciaRegs.SCIFFRX.bit.RXFIFORESET = 1; // enable op.
SciaRegs.SCIFFRX.bit.RXFFINTCLR = 1; // reset RX int
PieCtrlRegs.PIEACK.all = 0x0100; // acknowledge PIE
```

- That’s it.

Build, Load and Test

14. Apply all the commands needed to translate and debug your project.
15. Start Windows Hyper Terminal and type in the text “Texas”. The C28x will respond with the string “ Instruments\n\r”. If not → debug!

Optional Exercise

16. DSP – Junkies only! → Remote Control of the C28x by a PC!

Try to combine the “Knight-Rider” exercise (Lab4) with Lab8C. Let the PC send a string with a numerical value and use this value to control the speed of the “Knight-Rider”!

If a new value was received by the DSP it should answer back to the PC with a text like “control value xxx received”.

Note: The C standard function “atoi” can be used to convert an ASCII-string into a numerical value (see Code Composer Studio Help for details).

C28x Controller Area Network

Introduction

One of the most successful stories of the developments in automotive electronics in the last decade of the 20th century has been the introduction of distributed electronic control units in passenger cars. Customer demands, the dramatic decline in costs of electronic devices and the amazing increase in the computing power of microcontrollers has led to more and more electronic applications in a car. Consequently, there is a strong need for all those devices to communicate with each other, to share information or to co-ordinate their interactions.

The “Controller Area Network” was introduced and patented by Robert Bosch GmbH, Germany. After short and heavy competition, CAN was accepted by almost all manufacturers. Nowadays, it is the basic network system in nearly all automotive manufacturers’ shiny new cars. Latest products use CAN accompanied by other network systems such as LIN (a low-cost serial net for body electronics), MOST (used for in-car entertainment) or Flexray (used for safety critical communication) to tailor the different needs for communication with dedicated net structures.

Because CAN has high and reliable data rates, built-in failure detection and cost-effective prices for controllers, nowadays it is also widely used outside automotive electronics. It is a standard for industrial applications such as a “Field Bus” used in process control. A large number of distributed control systems for mechanical devices use CAN as their “backbone”.

What is “CAN”

→ what does CAN mean ?

it stands for : Controller Area Network

- it is a dedicated development of the automotive electronic industry
- it is a digital bus system for the use between electronic systems inside a car
- it uses a synchronous serial data transmission

→ why is it important to know about CAN ?

among the car network systems it is the market leader

- it is the in car backbone network of BMW, Volkswagen , Daimler-Chrysler , Porsche and more manufacturers
- CAN covers some unique internal features you can’t find elsewhere..
- there is an increasing number of CAN-applications also outside the automotive industry

9 - 2

Module Topics

C28x Controller Area Network.....	9-1
<i>Introduction</i>	<i>9-1</i>
<i>Module Topics.....</i>	<i>9-2</i>
<i>CAN Requirements.....</i>	<i>9-4</i>
<i>Basic CAN Features.....</i>	<i>9-5</i>
<i>CAN Implementation.....</i>	<i>9-6</i>
<i>CAN Data Frame</i>	<i>9-7</i>
<i>CAN Automotive Classes</i>	<i>9-9</i>
<i>ISO Standardization.....</i>	<i>9-10</i>
<i>CAN Application Layer.....</i>	<i>9-11</i>
<i>CAN Bus Arbitration – CSMA/CA</i>	<i>9-12</i>
<i>High Speed CAN</i>	<i>9-14</i>
<i>CAN Error Management.....</i>	<i>9-15</i>
<i>C28x CAN Module</i>	<i>9-18</i>
<i>C28x Programming Interface</i>	<i>9-19</i>
CAN Register Map	9-20
Mailbox Enable – CANME Mailbox Direction - CANMD.....	9-20
Transmit Request Set & Reset - CANTRS / CANTRR.....	9-21
Transmit Acknowledge - CANTA.....	9-21
Receive Message Pending - CANRMP	9-22
Remote Frame Pending - CANRFP.....	9-22
Global Acceptance Mask - CANGAM.....	9-23
Master Control Register - CANMC.....	9-24
<i>CAN Bit - Timing</i>	<i>9-25</i>
Bit-Timing Configuration - CANBTC	9-26
<i>CAN Error Register</i>	<i>9-28</i>
Error and Status - CANES.....	9-28
Transmit & Receive Error Counter - CANTEC / CANREC	9-29
<i>CAN Interrupt Register.....</i>	<i>9-30</i>
Global Interrupt Mask - CANGIM	9-30
Global Interrupt 0 Flag – CANGIF0	9-31
Global Interrupt 1 Flag – CANGIF1	9-31
Mailbox Interrupt Mask - CANMIM.....	9-32
Overwrite Protection Control - CANOPC	9-32
Transmit I/O Control - CANTIOC	9-33
Receive I/O Control - CANRIOC.....	9-33
<i>Alarm / Time Out Register</i>	<i>9-34</i>
Local Network Time - CANLNT	9-34
Time Out Control - CANTIOC.....	9-35
Local Acceptance Mask - LAMn	9-35
Message Object Time Stamp - MOTSn.....	9-36
Message Object Time Out - MOTOn	9-36

<i>Mailbox Memory</i>	9-37
Message Identifier - CANMID.....	9-37
Message Control Field - CANMCF.....	9-37
Message Data Field Low - CANMDL.....	9-38
Message Data Field High - CANMDH.....	9-38
<i>Lab Exercise 9</i>	9-39
Preface	9-39
Objective	9-40
Procedure.....	9-40
Open Files, Create Project File.....	9-40
Project Build Options	9-41
Modify Source Code.....	9-42
Build, Load and Run.....	9-43
Modify Source Code Cont.	9-43
Add the CAN initialization code	9-43
Prepare Transmit Mailbox #5	9-45
Add the Data Byte and Transmit	9-45
Build, Load and Run.....	9-46
<i>Lab Exercise 10</i>	9-47
Preface	9-47
Objective	9-48
Procedure.....	9-48
Open Files, Create Project File.....	9-48
Project Build Options	9-49
Modify Source Code.....	9-49
Build, Load and Run.....	9-50
Modify Source Code Cont.	9-50
Add the CAN initialization code	9-51
Prepare Receiver Mailbox #1	9-52
Add a polling loop for a message in mailbox 1	9-53
Build, Load and Run again	9-53
<i>What's next?</i>	9-55

CAN Requirements

Why a car network like CAN?

➔ what are typical requirements of an in car network?

- low cost solution
- good and high performance with few overhead transmission
- high volume production in excellent quality
- high reliability and electromagnetic compatibility (EMC)
- data security due to a fail-safe data transmission protocol
- short message length, only a few bytes per message
- an 'open system'

➔ what are customer demands ?

- reduce pollution
- reduce fuel consumption
- increase engine performance
- higher safety standards , active & passive systems
- add more & more comfort into car
 - lots of electronic control units (ECU) necessary !!!
 - lots of data communication between ECU's.

9 - 3

ECU's of a car

The number of microcontrollers inside a car :



break control ABS (1 + 4)
keyless entry system(1)
active wheel drive control (4)
engine control (2)
airbag sensor(6++)
seat occupation sensors(4)
automatic gearbox(1)
electronic park brake(1)
diagnostic computer(1)
driver display unit(1)
air conditioning system(1)
adaptive cruise control(1)
radio / CD-player(2)
collision warning radar(2)
rain/ice/snow sensor systems (1 each)
dynamic drive control(4)
active damping system (4)
driver information system(1)
GPS navigation system(3)

9 - 4

Basic CAN Features

Features of CAN

- developed by Robert Bosch GmbH, Stuttgart in 1987
- licensed to most of the semiconductor manufacturers
- meanwhile included in most of the microcontroller-families
- today the most popular serial bus for automotive applications
- competitors are : VAN (France) , J1850 (USA) and PALMNET (Japan)
- a lot of applications in automation & control (low level field bus)

Features :

- multi master bus access
- random access with collision avoidance
- short message length , at max. 8 Bytes per message
- data rates 100KBPS to 1MBPS
- short bus length , depending on data rate
- self-synchronised bit coding technology
- optimised EMC-behaviour
- build in fault tolerance
- physical transmission layers : RS485, ISO-high-speed(differential voltage), ISO-low-speed (single voltage), fibre-optic, galvanic isolated

9 - 5

CAN does not use physical addresses to address stations. Each message is sent with an identifier that is recognized by the different nodes. The identifier has two functions – it is used for message filtering and for message priority. The identifier determines if a transmitted message will be received by CAN modules and determines the priority of the message when two or more nodes want to transmit at the same time.

The bus access procedure is a multi-master principle, all nodes are allowed to use CAN as a master node. One of the basic differences to Ethernet is the adoption of non-destructive bus arbitration in case of collisions, called “Carrier Sense Multiple Access with Collision Avoidance”(CSMA/CA). This procedure ensures that in case of an access conflict, the message with higher priority will not be delayed by this collision.

The physical length of the CAN is limited, depending on the baud rate. The data frame consists of a few bytes only (maximum 8), which increases the ability of the net to respond to new transmit requests. On the other hand, this feature makes CAN unsuitable for very high data throughputs, for example, for real time video processing.

There are several physical implementations of CAN, such as differential twisted pair (automotive class: CAN high speed), single line (automotive class: CAN low speed) or fibre optic CAN, for use in harsh environments.

CAN Implementation

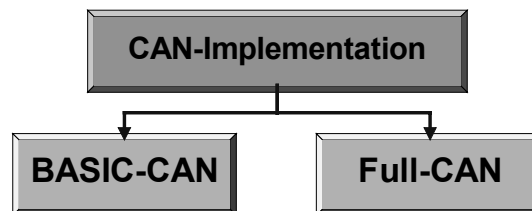
Implementation / Classification of CAN

The Implementation of CAN in Silicon



Don't get confused !

Communication is identical for all implementations of CAN. However, there are two principal hardware implementations and two additional versions of data formats :



9 - 6

There are two versions of how the CAN-module is implemented in silicon, called “BASIC” and “Full” – CAN. Almost all new processors with a built-in CAN module offer both modes of operation. BASIC-CAN as the only mode is normally used in cost sensitive applications.

BASIC-CAN and FULL-CAN

BASIC-CAN

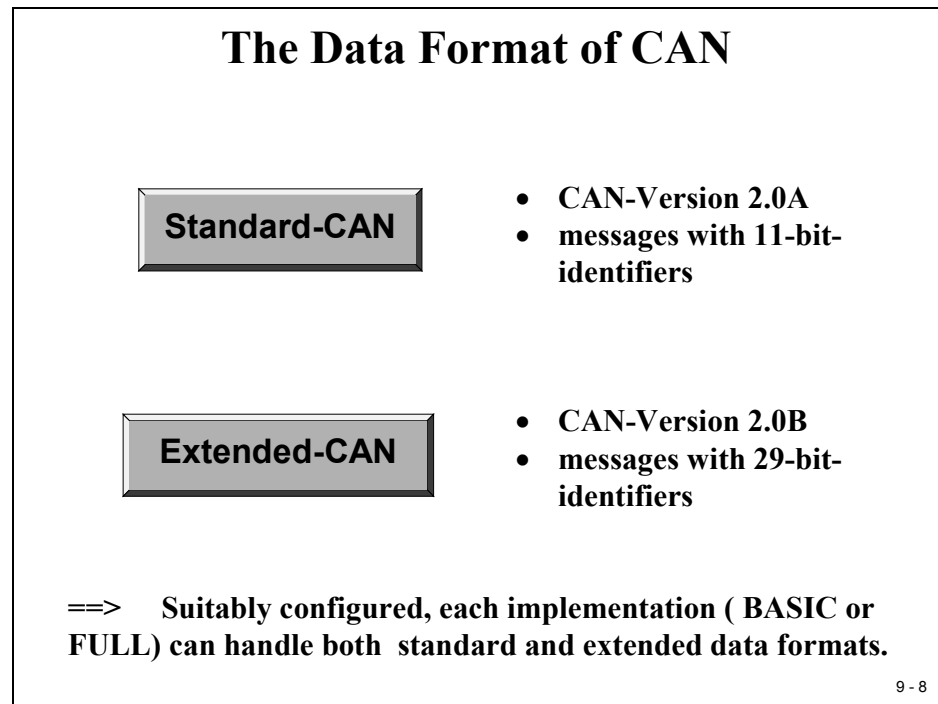
- Close loop between MCU-core and CAN
- only one transmit buffer
- only two receive buffer
- only one filter for incoming messages
- Software routines are needed to select between incoming messages

Full-CAN

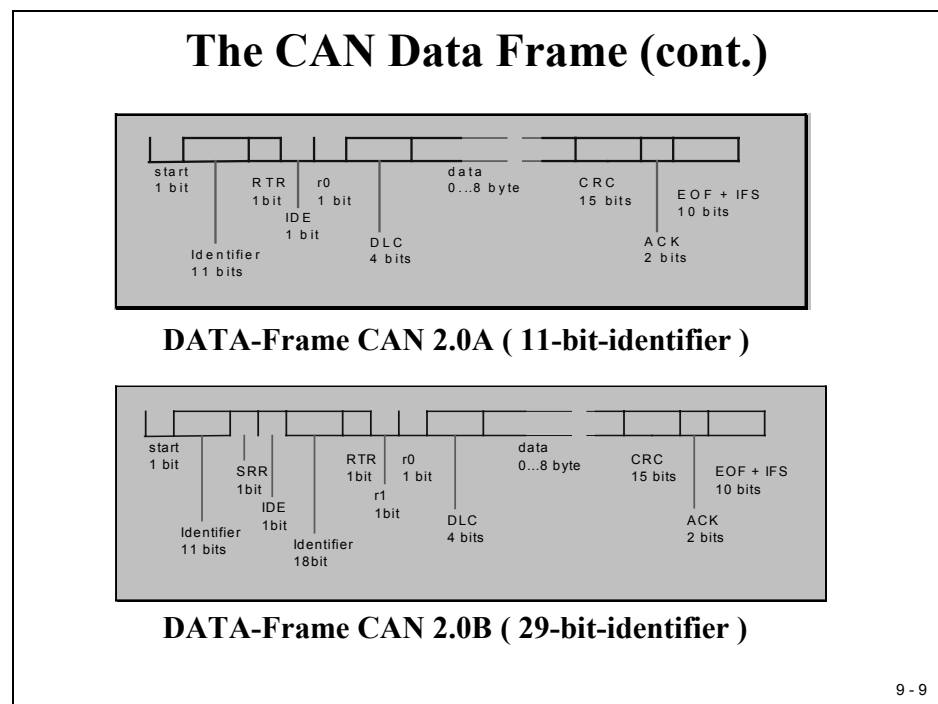
- provide a message server
- extensive acceptance filtering on incoming messages
- user configurable mailboxes
- mailbox memory area , size of mailbox areas depends on manufacturer
- advanced error recognition

9 - 7

CAN Data Frame



The two versions of the data frame format allow the reception and transmission of standard frames and extended frames in a mixed physical set up, provided the silicon is able to handle both types simultaneously (CAN version 2.0A and 2.0B respectively).



The CAN Data Frame

each data frame consists of four segments :

(1) arbitration-field :

- denote the priority of the message
- logical address of the message (identifier)
- Standard frame , CAN 2.0A : 11 bit-identifier
- Extended frame (CAN 2.0B) : 29 bit-identifier

(2) data field :

- up to 8 bytes per message ,
- a 0 byte message is also permitted

(3) CRC field:

- cyclic redundancy check ; contains a checksum generated by a CRC-polynomial

(4) end of frame field:

- contains acknowledgement , error-messages, end of message

9 - 10

The CAN Data Frame (cont.)

start bit	(1 bit - dominant) : flag for the begin of a message; after idle-time falling-edge to synchronise all transmitters
identifier	(11 bit) : mark the name of the message and its priority ;the lower the value the higher the priority
RTR	(1 bit) : remote transmission request ; if RTR=1 (recessive) no valid data's inside the frame - it is a request for receivers to send their messages
IDE	(1 bit) : Identifier Extension ; if IDE=1 then extended CAN-frame
r0	(1 bit) :reserved
CDL	(4 bit) : data length code, code-length 9 to 15 are not permitted !
data	(0..8 byte) : the data's of the message
CRC	(15 bit) : cyclic redundancy code ; only to detect errors, no correction ; hamming-distance 6 (up to 6 single bit errors)
ACK	(2 bit) : acknowledge ; each listener, which receive a message without errors (including CRC !) has to transmit an acknowledge-bit in this time-slot !!!
EOF	(7 bit = 1 , recessive) : end of frame ; intentional violation of the bit-stuff-rule ; normally after five recessive bits one stuff-bit follows automatically
IFS	(3 bit = 1 recessive) : inter frame space ; time space to copy a received message from bus-handler into buffer
Extended Frame only :	
SRR	(1 bit = recessive) : substitute remote request ; substitution of the RTR-bit in standard frames
r1	(1 bit) : reserved

9 - 11

CAN Automotive Classes

The Automotive Classification of CAN

There are four classes of CAN-systems in use :

- | | |
|---|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ➡ | Class A: chassis electronics, e.g. mirror adjust, light & bulb control
10 KBPS ; 1 data transmission line , chassis used for ground |
| ➡ | Class B: distribution of information, e.g. central driver-display; 40 KBPS |
| ➡ | Class C: real-time information exchange in and between control-loops e.g. engine-control(ignition, injection), brake-systems (ABS, ASR); dynamic drive control, damping ; steering-control ; 1 MBPS |
| ➡ | Class D: network with large number of data's (> 10KB/frame) , e.g. radio, telephone, navigation-systems |

9 - 12

The four automotive CAN classes are used to specify different groups of electronic control units in a car. There are also different specifications for Electromagnetic Compatibility (EMC) compliances and tailored versions of physical transceivers available for the four classes in use. Class A and B are quite often specified as “Low Speed CAN” with a data rate of 100 kbps. Class C usually is implemented as “High Speed CAN”, commonly with a baud rate of 500 kbps.

For more details on automotive electronics, look out for additional classes in your university. A highly recommended textbook about CAN in automotive applications is:

“CAN System Engineering”
Wolfhard Lawrenz
Springer N.Y. 1997
ISBN: 0-387-94939-9

ISO Standardization

The Standardisation of CAN

- The CAN is an open system
- The European ISO has drafted equivalent standards
- The CAN-Standards follow the ISO-OSI seven layer model for open system interconnections
- In automotive communication networks only layer 1, 2 and 7 are implemented
- Layer 7 is not standardised

The ISO-Standards :

- CAN : ISO 11519 - 2 : layer 2 , layer 1 (top)
- CAN : ISO 11898 : layer 1 (bottom)
- VAN : ISO 11519 - 3 : layer 2 , layer 1
- J1850 : ISO 11519 - 4 : layer 2 , layer 1

9 - 13

ISO Reference Model

Open Systems Interconnection (OSI):

Layer 7 Application Layer	
Layer 6 Presentation Layer	void
Layer 5 Session Layer	void
Layer 4 Transport Layer	void
Layer 3 Network Layer	void
Layer 2 Data Link Layer	
Layer 1 Physical Layer	

Layer 1 : Interface to the transmission lines

- differential two-wire-line, twisted pair with/without shield
- IC's as integrated transceiver
- Optional fibre optical lines (passive coupled star, carbon)
- Optional Coding : PWM, NRZ, Manchester Code

Layer 2 : Data Link Layer

- message format and transmission protocol
- CSMA/CA access protocol

Layer 7 : Application Layer

- a few different standards for industry, no for automotive
- but a must : interfaces for communication, network management and real-time operating systems

9 - 14

CAN Application Layer

CAN Layer 7

1. CAN Application Layer (CAL):
 - European CAN user group "CAN in Automation (CiA)"
 - originated by Philips Medical Systems 1993
 - CiA DS-201 to DS-207
 - standardised communication objects, -services and -protocols (CAN-based Message Specification)
 - Services and protocols for dynamic attachment of identifiers (DBT)
 - Services and protocols for initialise, configure and obtain the net (NMT)
 - Services and protocols for parametric set-up of layer 2 & 1 (LMT)
 - Automation, medicine, traffic-industry
2. CAN Kingdom
 - Swedish , Kvaser ;
 - toolbox
 - "modules serves the net , not net serves for the modules"
 - off-road-vehicles ; industrial control , hydraulics
3. OSEK/VDX
 - European automotive industry , supplier standard
 - include services of a standardised real-time-operating system

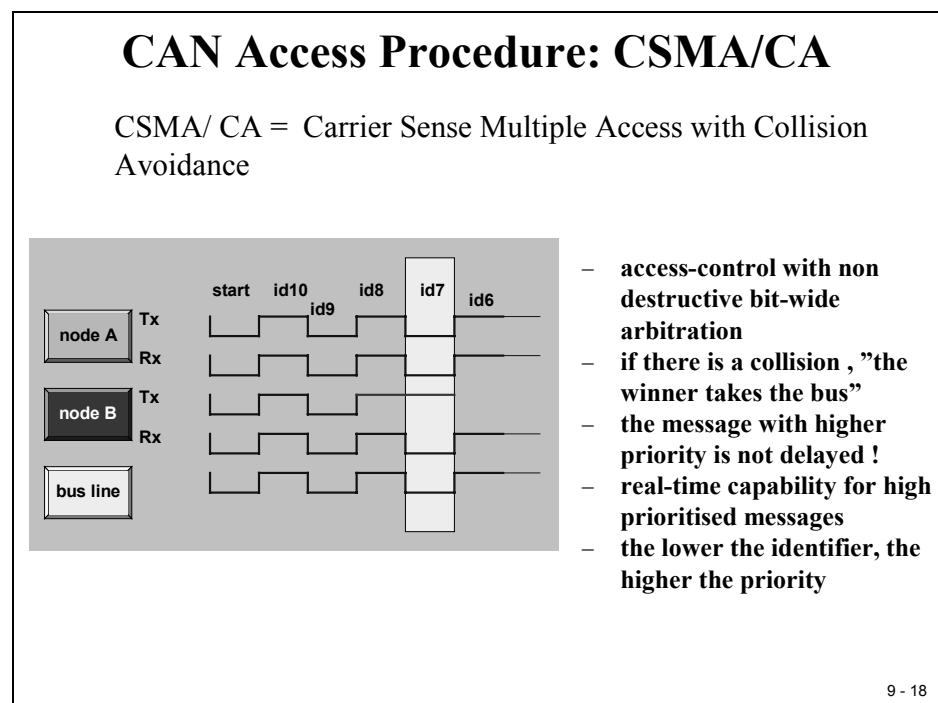
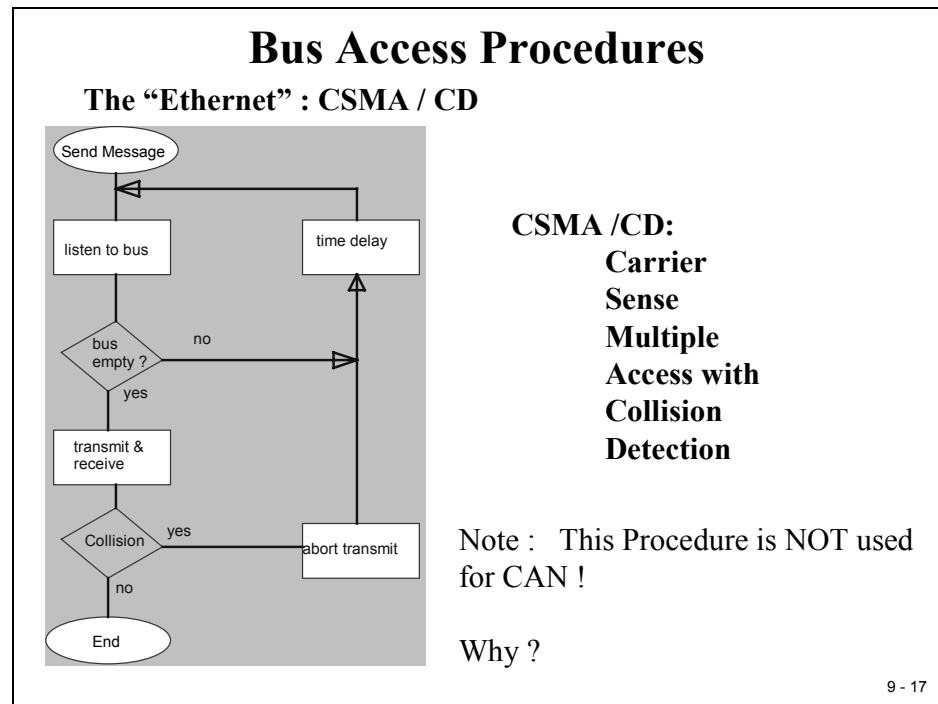
9 - 15

CAN Layer 7(cont.)

4. CANopen :
 - European Community funded project "ESPRIT"
 - 1995 : CANopen profile : CiA DS-301
 - 1996 : CANopen device profile for I/O : CiA DS-401
 - 1997 : CANopen drive profile
 - industrial control , numeric control in Europe
5. DeviceNet :
 - Allen-Bradley, now OVDA-group
 - device profiles for drives, sensors and resolvers
 - master-slave communication as well as peer to peer
 - industrial control , mostly USA
6. Smart Distributed Systems (SDS)
 - Honeywell , device profiles
 - only 4 communication functions , less hardware resources
 - industrial control , PC-based control
 - US-food industry
 - Motorola 68HC05 with SDS on silicon available now
7. other profile systems
 - J1939 US truck and bus industry
 - LBS Agricultural bus system, Germany, DIN)
 - M3S : European manufacturers of wheelchairs

9 - 16

CAN Bus Arbitration – CSMA/CA



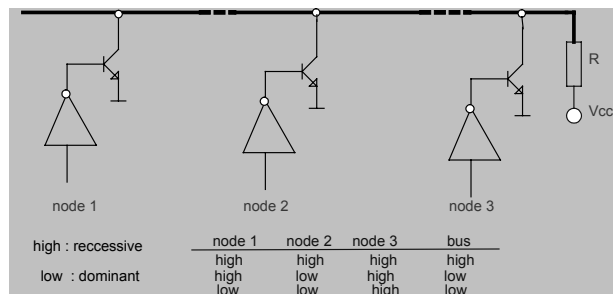
CSMA/CA (cont.)

CSMA / CA =

"bit - wide arbitration during transmission with simultaneous receiving and comparing of the transmitted message"

means :

- **if there is a collision within the arbitration-field, only the node with the lower priority cancels its transmission.**
- **The node with the highest priority continues with the transmission of the message.**



9 - 19

As you can see from the previous slide the arbitration procedure at a physical level is quite simple: it is a “wired-AND” principle. Only if all 3 node voltages (node 1, node2 or node3) are equal to 1 (recessive), the bus voltage stays at V_{cc} (recessive). If only one node voltage is switched to 0 (dominant), the bus voltage is forced to the dominant state (0).

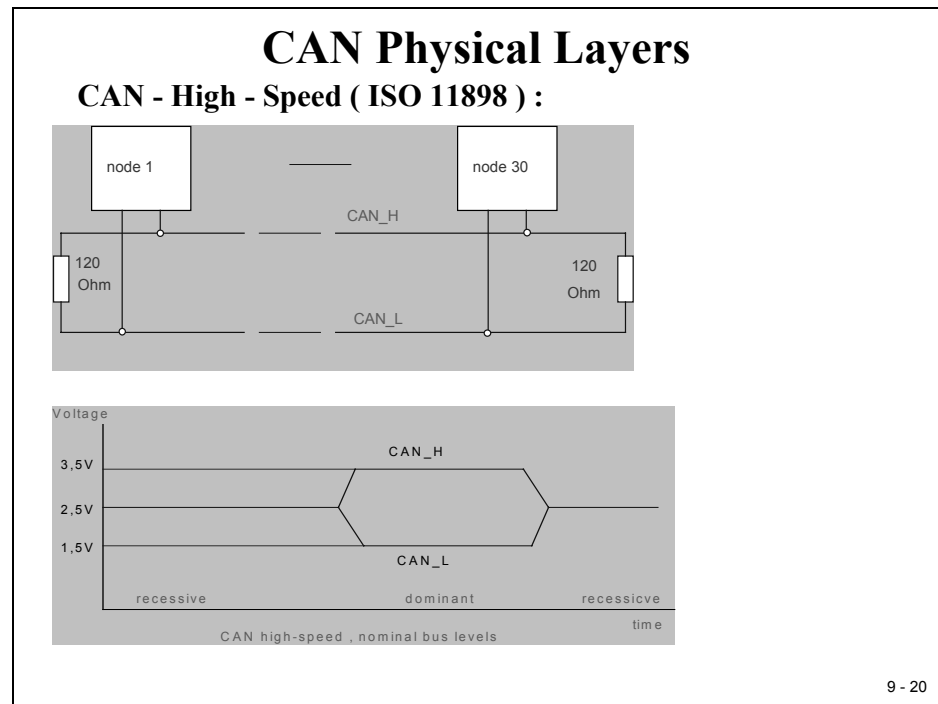
The beauty of CAN is that the message with highest priority is not delayed at all in case of a collision. For the message with highest priority, we can determine the worst-case response time for a data transmission. For messages with other priorities, to calculate the worst-case response time is a little bit more complex task. It could be done by applying a so-called “time dilatation formula for non-interruptible systems”:

$$R_i^{n+1} = C_i + B_{\max i} + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n - C_i}{T_j} \right\rceil * C_j$$

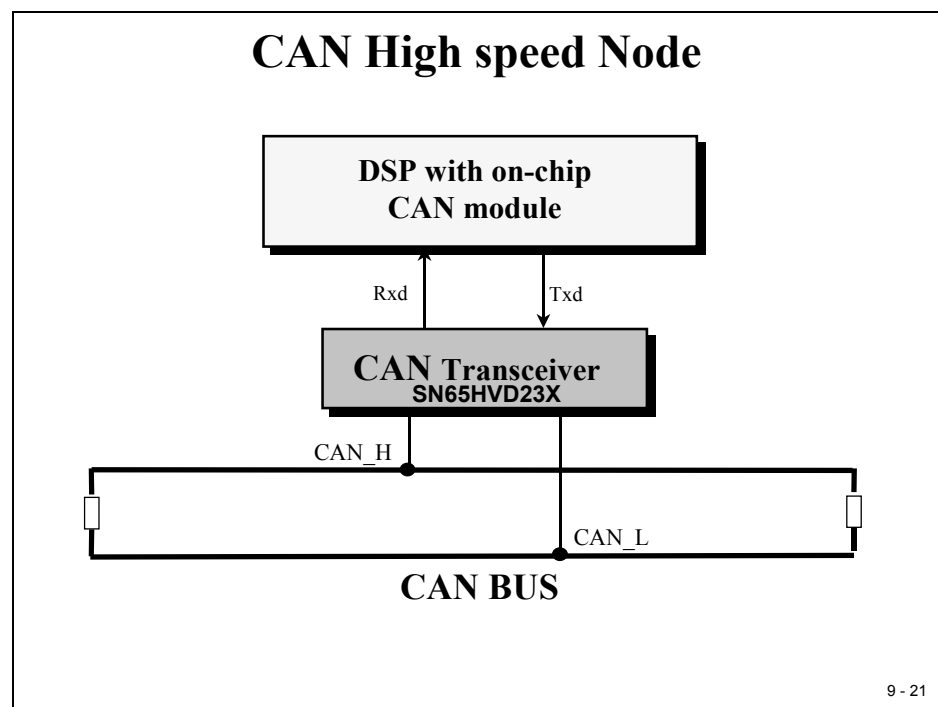
HARTER, P.K: “Response Times in level structured systems” Techn. Report, Univ. of Colorado, 1991

In detail, the hardware structure of a CAN-transceiver is more complex. Due to the principle of CAN-transmissions as a “broadcast” type of data communication, all CAN-modules are forced to “listen” to the bus all the time. This also includes the arbitration phase of a data frame. It is very likely that a CAN-module might lose the arbitration procedure. In this case, it is necessary for this particular module to switch into receive mode immediately. This requires every transceiver to provide the actual bus voltage status permanently to the CAN-module.

High Speed CAN

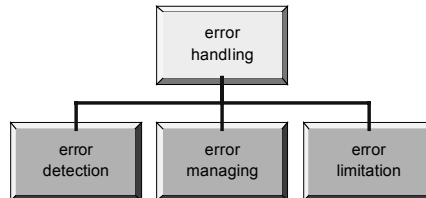


To generate the voltage levels for the differential voltage transmission according to CAN High Speed we need an additional transceiver device, e.g. the SN65HVD23x.



CAN Error Management

CAN Error & Exception Management



How does it work ?

- most of errors should be detected and self-corrected by the CAN-Chip itself
- automatic notification to all other nodes, that an error has been seen :

Error-Frame = deliberate violation of code-law's)

(6-bit dominant = passive error frame)

(12-bit dominant = active error frame)

- all nodes have to cancel the last message they have received
- transmission is repeated automatically by the bus - handler

9 - 22

CAN Error Recognition

• Bit-Error

the transmitted bit doesn't read back with the same digital level (except arbitration and acknowledge- slot)

• Bit-Stuff-Error

more than 5 continuous bits read back with the same digital level (except 'end of frame'-part of the message)

• CRC-Error

the received CRC-sum doesn't match with the calculated sum

• Format-Error

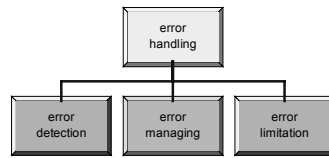
Violation of the data-format of the message , e.g.: CRC-delimiter is not recessive or violation of the 'end -of-frame'-field

• Acknowledgement-Error

transmitter receives no dominant bit during the acknowledgement slot, i.e. the message was not received by any node.

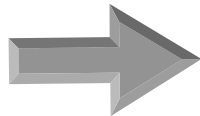
9 - 23

CAN Error Sequence



After detection of an error by a node every other node receives a particular frame, the Error -Frame :

This is the violation of the stuff-bit-rule by transmission of at least 6 dominant bits. The Error-Frame causes all other nodes to recognise an Error Status of the bus.

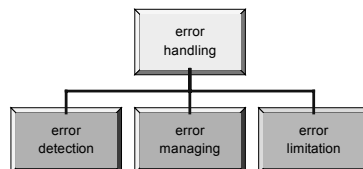


Error Management Sequence :

- error is detected
- error-frame will be transmitted by all nodes, which have detected this error
- The last message received will be cancelled by all nodes
- Internal hardware error-counters will be increased
- The original message will be transmitted again.

9 - 24

CAN Error Status



* Purpose: avoid persistent disturbances of the CAN by switching off defective nodes

* three Error States :



Error Active : normal mode, messages will be received and transmitted. In case of error an active error frame will be transmitted

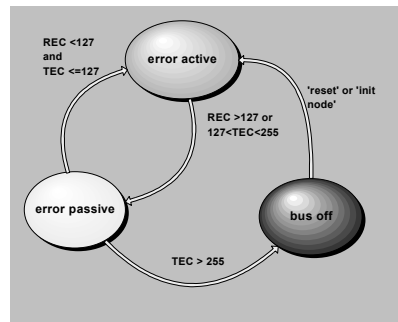
Error Passive : after detection of a fixed number of errors , the node reaches this state. messages will be received and transmitted, in case of error the node sends a passive error frame.

Bus Off : the node is separated from CAN , neither transmission nor receive of messages is allowed, node is not able to transmit error frame's . leaving this state is only possible by reset !

9 - 25

CAN Error Counter

State - Diagram :



- transitions will be carried out automatically by the CAN-chip
- states are managed by 2 Error Counters :
Receive Error Counter (REC)
Transmit Error Counter (TEC)
- Possible situations :
 - a transmitter recognises an error:
 $TEC := TEC + 8$
 - a receiver sees an error : $REC := REC + 1$
 - a receiver sees an error, after transmitting an error frame:
 $REC := REC + 8$
 - if an 'error active'-node finds a bit-stuff-error during transmission of an error frame:
 $TEC := TEC + 1$
 - successful transmission:
 $TEC := TEC - 1$
 - successful receive :
 $REC := REC - 1$

9 - 26

C28x CAN Module

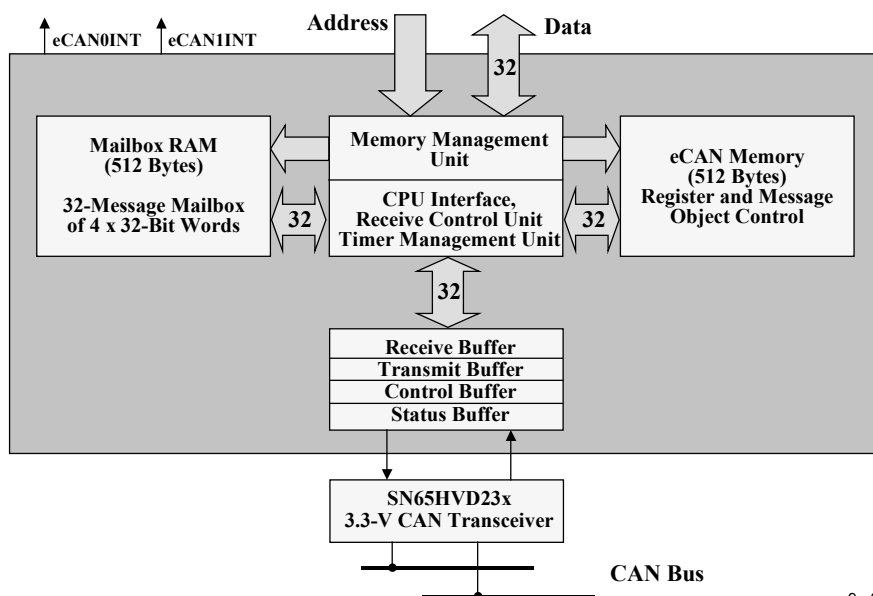
C28x CAN Features

- ◆ Fully CAN protocol compliant, version 2.0B
- ◆ Supports data rates up to 1 Mbps
- ◆ Thirty-two mailboxes
 - ◆ Configurable as receive or transmit
 - ◆ Configurable with standard or extended identifier
 - ◆ Programmable receive mask
 - ◆ Supports data and remote frame
 - ◆ Composed of 0 to 8 bytes of data
 - ◆ Uses 32-bit time stamp on messages
 - ◆ Programmable interrupt scheme (two levels)
 - ◆ Programmable alarm time-out
- ◆ Programmable wake-up on bus activity
- ◆ Self-test mode

9 - 27

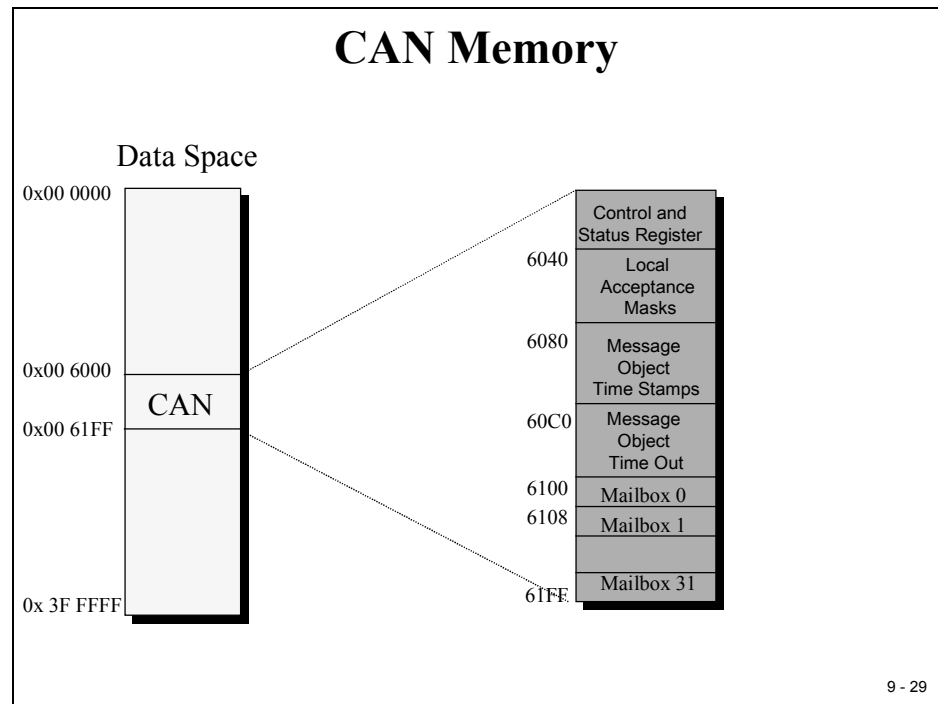
The DSP CAN module is a full CAN Controller. It contains a message handler for transmission, a reception management and frame storage. The specification is CAN 2.0B Active – that is, the module can send and accept standard (11-bit identifier) and extended frames (29-bit identifier).

CAN Block Diagram



9 - 28

C28x Programming Interface



The CAN controller module contains 32 mailboxes for objects of 0- to 8-byte data lengths:

- configurable transmit/receive mailboxes
- configurable with standard or extended identifier

The CAN module mailboxes are divided into several parts:

- MID – contains the identifier of the mailbox
- MCF (Message Control Field) – contains the length of the message (to transmit or receive) and the RTR bit (Remote Transmission Request – used to send remote frames)
- MDL and MDH – contains the data

The CAN module contains registers, which are divided into five groups. These registers are located in data memory from 0x006000 to 0x0061FF. The five register groups are:

- Control & Status Registers
- Local Acceptance Masks
- Message Object Time Stamps
- Message Object Timeout
- Mailboxes

It is the responsibility of the programmer to go through all those registers and set every single bit according to the designated operating mode of the CAN module. It is also a challenge for a student to exercise the skills required to debug. So let's start!

First, we will discuss the different CAN registers. If this chapter becomes too tedious, ask your teacher for some practical examples how to use the various options. Be patient!

CAN Register Map

CAN Control & Status Register	
	31 0
6000	CANME
6002	CANMD
6004	CANTRS
6006	CANTRR
6008	CANTA
600A	CANAA
600C	CANRMP
600E	CANRML
6010	CANRFP
6012	CANGAM
6014	CANMC
6016	CANBTC
6018	CANES
601A	CANTEC
601C	CANREC
601E	CANGIF0
	31 0
6020	CANGIM
6022	CANGIF1
6024	CANMIM
6026	CANMIL
6028	CANOPC
602A	CANTIOC
602C	CANRIOC
602E	CANLNT
6030	CANTOC
6032	CANTOS
6034	reserved
6036	reserved
6038	reserved
603A	reserved
603C	reserved
603E	reserved

9 - 30

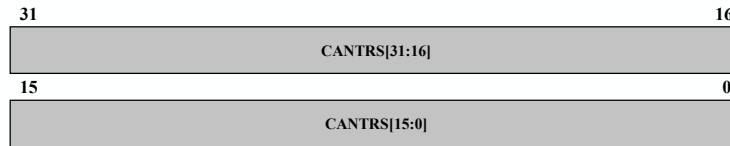
Mailbox Enable – CANME Mailbox Direction - CANMD

CAN Mailbox Enable Register (CANME) – 0x006000	
31	16
CANME[31:16]	
15	0
CANME[15:0]	
Mailbox Enable Bits 0 = corresponding mailbox is disabled 1 = The corresponding mailbox is enabled. A mailbox must be disabled before writing to the contents of any mailbox identifier field.	
CAN Mailbox Direction Register (CANMD) – 0x006002	
31	16
CANMD[31:16]	
15	0
CANMD[15:0]	
Mailbox Direction Bits 0 = corresponding mailbox is defined as a transmit mailbox. 1 = corresponding mailbox is defined as a receive mailbox.	

9 - 31

Transmit Request Set & Reset - CANTRS / CANTRR

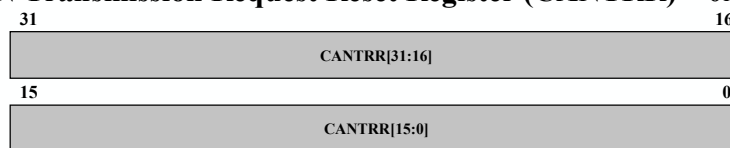
CAN Transmission Request Set Register (CANTRS) – 0x006004



Mailbox Transmission Request Set Bits (TRS)

0 = no operation. NOTE: Bit will be cleared by CAN-Module logic after successful transmission.
1 = Start of transmission of corresponding mailbox. Set to 1 by user software;
OR by CAN logic in case of a Remote Transmit Request.

CAN Transmission Request Reset Register (CANTRR) – 0x006006



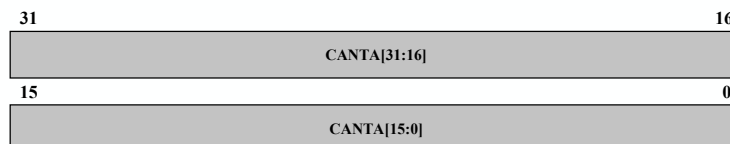
Mailbox Transmission Reset Request Bits (TRR)

0 = no operation.
1 = setting TRRn cancels a transmission request, if not currently being processed.

9 - 32

Transmit Acknowledge - CANTA

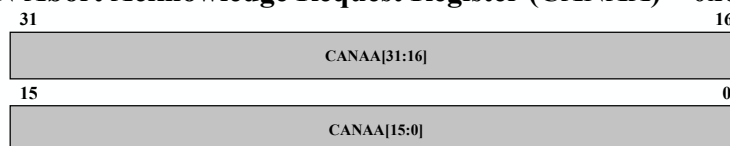
CAN Transmission Acknowledge Register (CANTA) – 0x006008



Mailbox Transmission Acknowledge Bits (TA)

0 = the message is not sent.
1 = if the message of mailbox n is sent successfully, the bit n of this register is set.
Note: To reset a TA bit by software: write a '1' into it!!

CAN Abort Acknowledge Request Register (CANAA) – 0x00600A



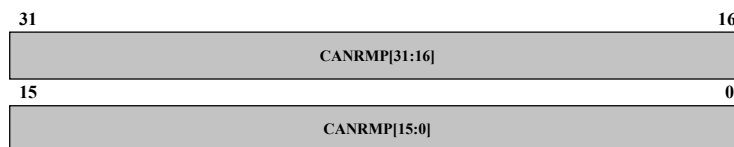
Mailbox Abort Acknowledge Bits (AA)

0 = The transmission is not aborted.
1 = The transmission of mailbox n is aborted.
Note: To reset a AA bit by software: write a '1' into it!!

9 - 33

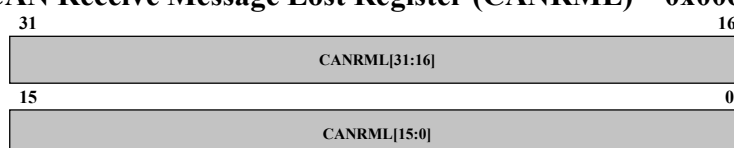
Receive Message Pending - CANRMP

CAN Receive Message Pending Register (CANRMP) – 0x00600C



Mailbox Receive Message Pending Bits (RMP)
 0 = the mailbox does not contain a message.
 1 = the mailbox contains a valid message.
 Note: To reset a RMP bit by software: write a '1' into it!!

CAN Receive Message Lost Register (CANRML) – 0x00600E

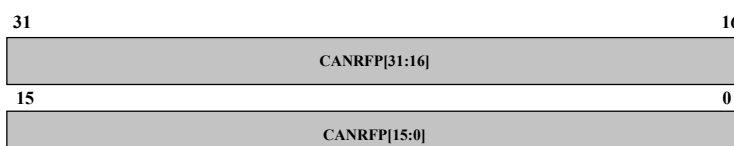


Mailbox Receive Message Lost Bits (RML)
 0 = no message was lost.
 1 = an old unread message has been overwritten by a new one in that mailbox.
 Note: To reset a RML bit by software: write a '1' into it!!

9 - 34

Remote Frame Pending - CANRFP

CAN Remote Frame Pending Register (CANRFP) – 0x006010

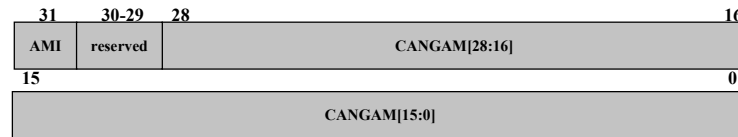


Mailbox Remote Frame Pending Bits (RFP)
 0 = no remote frame request was received.
 1 = a remote frame request was received by the CAN module.
 Note: To reset a RFP bit by software: write a '1' into the corresponding TRR bit!!

9 - 35

Global Acceptance Mask - CANGAM

CAN Global Acceptance Mask Register (CANGAM) – 0x006012



Note : This Register is used in SCC mode only for mailboxes 6 to 15, if the AME bit (MID.30) of the corresponding mailbox is set. It is a “don’t care” for HECC – Mode!

Acceptance Mask Identifier Bit (AMI)

0 = the identifier extension bit in the mailbox determines which messages shall be received. Filtering is not applicable.

1 = standard and extended frames can be received. In case of an extended frame all 29 bits of the identifier and all 29 bits of the GAM are used for the filter. In case of a standard frame only bits 28-18 of the identifier and the GAM are used for the filter.

Note: The IDE bit of a receive mailbox is a “don’t care” and is overwritten by the IDE bit of the transmitted message.

Global Acceptance Mask (GAM)

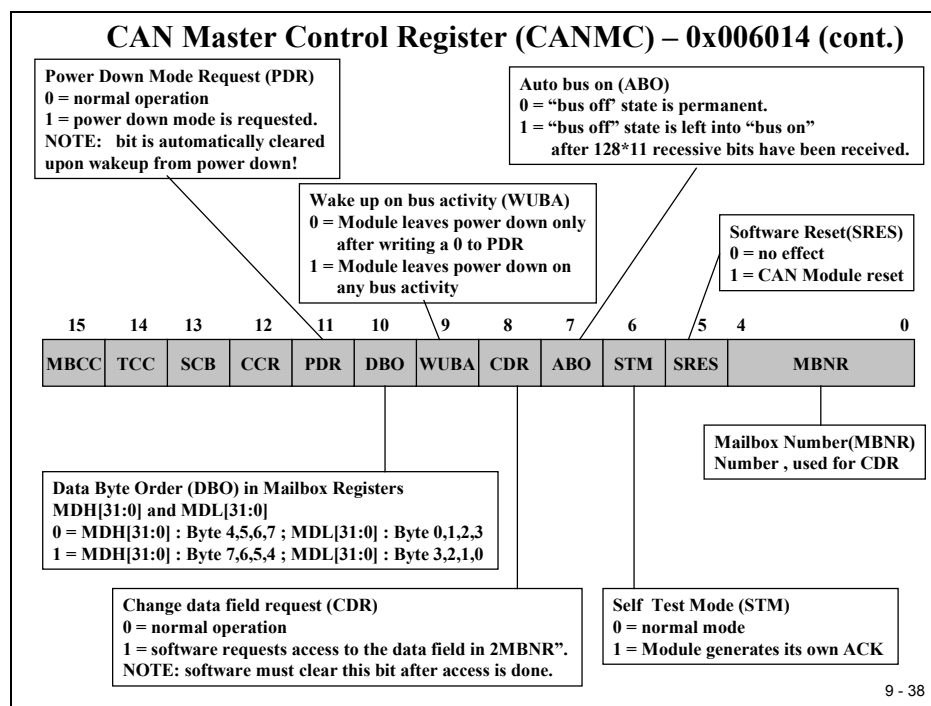
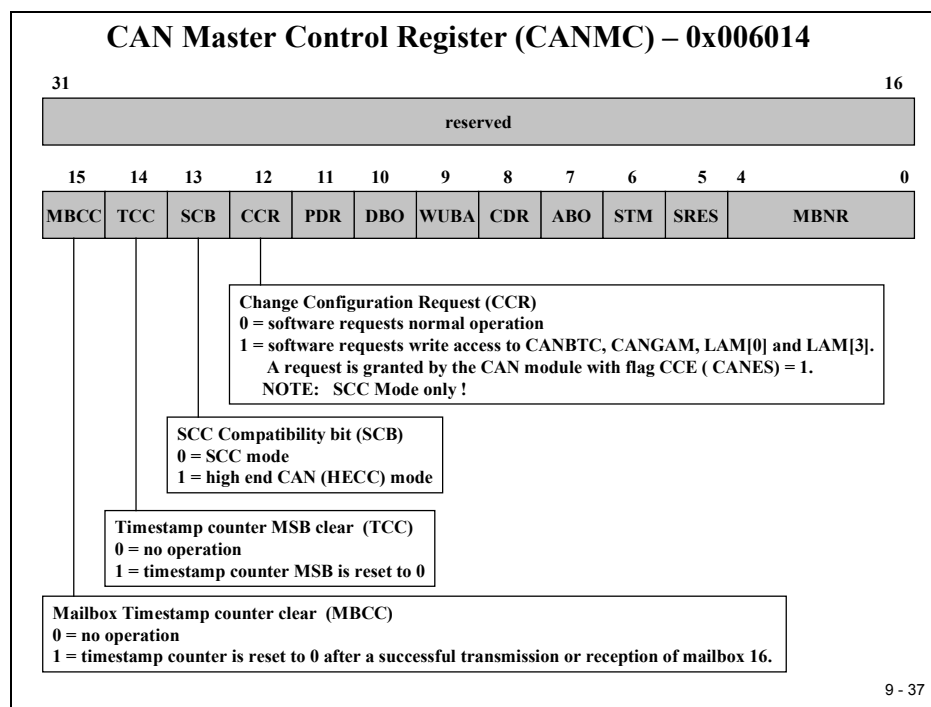
0 = bit position must match the corresponding bit in register CANMIDn.

1 = bit position of the incoming identifier is a “don’t care”.

Note: To reset a RFP bit by software: write a ‘1’ into the corresponding TRR bit!!

9 - 36

Master Control Register - CANMC



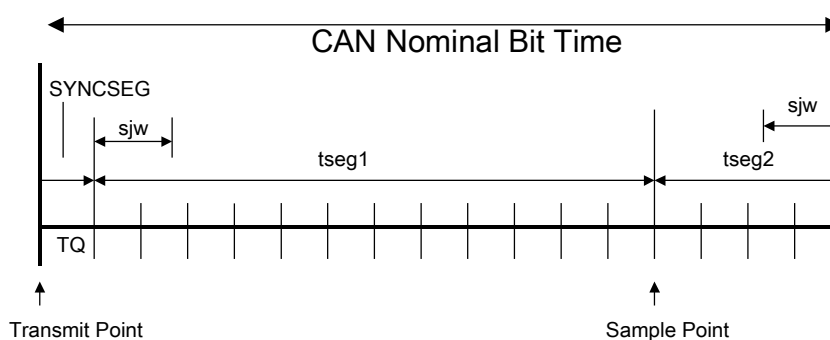
CAN Bit - Timing

CAN Bit-Timing Configuration

- ◆ CAN protocol specification splits the nominal bit time into four different time segments:
 - ◆ SYNC_SEG
 - ◆ Used to synchronize nodes
 - ◆ Length : always 1 Time Quantum (TQ)
 - ◆ PROP_SEG
 - ◆ Compensation time for the physical delay times within the net
 - ◆ Twice the sum of the signal's propagation time on the bus line, the input comparator delay and the output driver delay.
 - ◆ Programmable from 1 to 8 TQ
 - ◆ PHASE_SEG1
 - ◆ Compensation for positive edge phase shift
 - ◆ Programmable from 1 to 8 TQ
 - ◆ PHASE_SEG2
 - ◆ Compensation time for negative edge phase shift
 - ◆ Programmable from 2 to 8 TQ

9 - 39

CAN Bit-Timing Configuration



- ◆ tseg1 : PROP_SEG + PHASE_SEG1
- ◆ tseg2 : PHASE_SEG2
- ◆ TQ : SYNCSEG
- ◆ CAN Nominal Bit Time = TQ + tseg1 + tseg2

9 - 40

CAN Bit-Timing Configuration

- ◆ According to the CAN – Standard the following bit timing rules must be fulfilled:
 - ◆ $t_{seg1} \geq t_{seg2}$
 - ◆ $3/BRP \quad t_{seg1} \quad 16 TQ$
 - ◆ $3/BRP \quad t_{seg2} \quad 8 TQ$
 - ◆ $1 TQ \quad sjw \quad MIN[4 * TQ, t_{seg2}]$
 - ◆ $BRP \geq 5$ (if three sample mode is used)

9 - 41

Bit-Timing Configuration - CANBTC

CAN Bit-Timing Configuration Register (CANBTC) – 0x006016

31																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																										
----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--

Baud Rate Prescaler (BRP)
Defines the Time Quantum (TQ):

$$TQ = \frac{BRP + 1}{SYSCLK}$$

Note: with an external clock of 30MHz and a PLL * 5:
 SYSCLK = 150MHz

9 - 42

CAN Bit-Timing Configuration Register (CANBTC) – 0x006016

15	11	10	9	8	7	6	3	2	0
reserved				SBG	SJW	SAM	TSEG1		TSEG2

Synchronisation Jump Width (SJW)

$$sjw = TQ * (SJW + 1)$$

Synchronisation Edge Select (SBG)

0 = re synchronisation with falling edge only
 1 = re-sync. with rising & falling edge

Time Segment 1(tseg1)

$$tseg1 = TQ * (TSEG1 + 1)$$

Time Segment 2(tseg2)

$$tseg2 = TQ * (TSEG2 + 1)$$

Sample Points (SAM)

0 = one sample at sample point
 1 = 3 samples at sample point – majority vote

9 - 43

CAN Bit-Timing Examples

◆ Bit Configuration for SYSCLK = 150 MHz

◆ Sample Point at 80% of Bit Time :

CAN-Baudrate	BRP	TSEG1	TSEG2
1 MBPS	9	10	2
500 KBPS	19	10	2
250 KBPS	39	10	2
125 KBPS	79	10	2
100 KBPS	99	10	2
50 KBPS	199	10	2

◆ Example 50 KBPS:

$$TQ = (199+1)/150 \text{ MHz} = 1.334 \text{ ns}$$

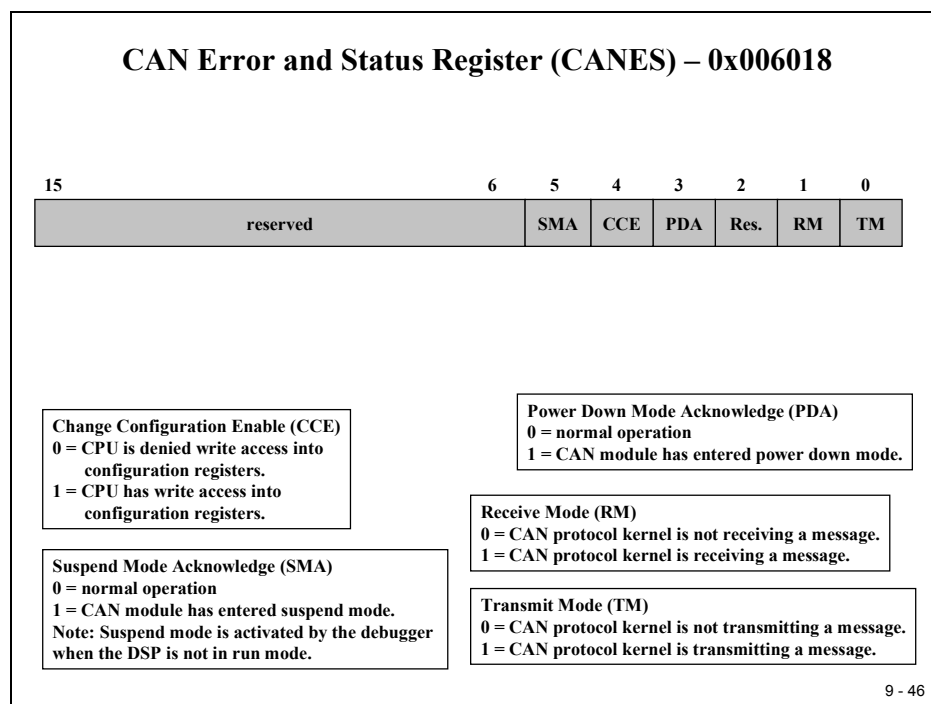
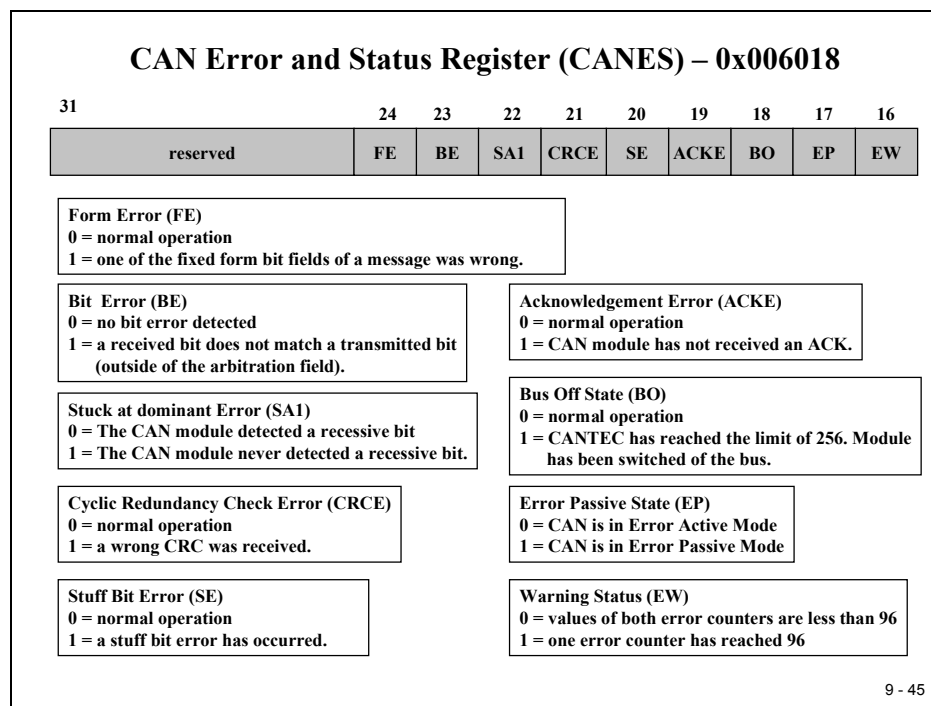
$$tseg1 = 1.334 \text{ ns} * (10 + 1) = 14.674 \text{ ns} \rightarrow t_{CAN} = 20.010 \text{ ns}$$

$$tseg2 = 1.334 \text{ ns} * (2 + 1) = 4.002 \text{ ns}$$

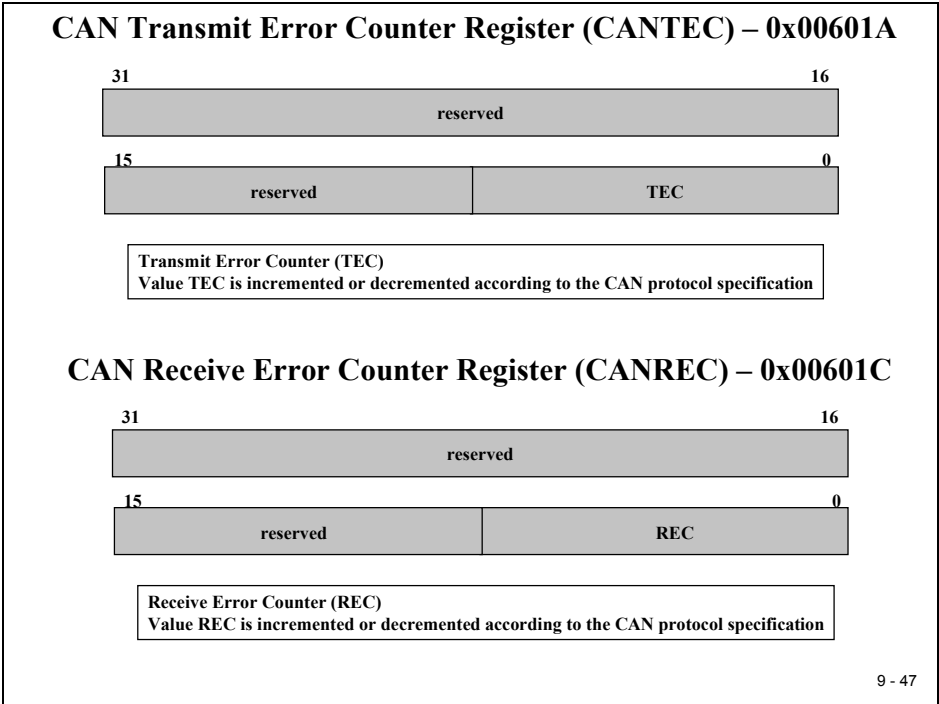
9 - 44

CAN Error Register

Error and Status - CANES

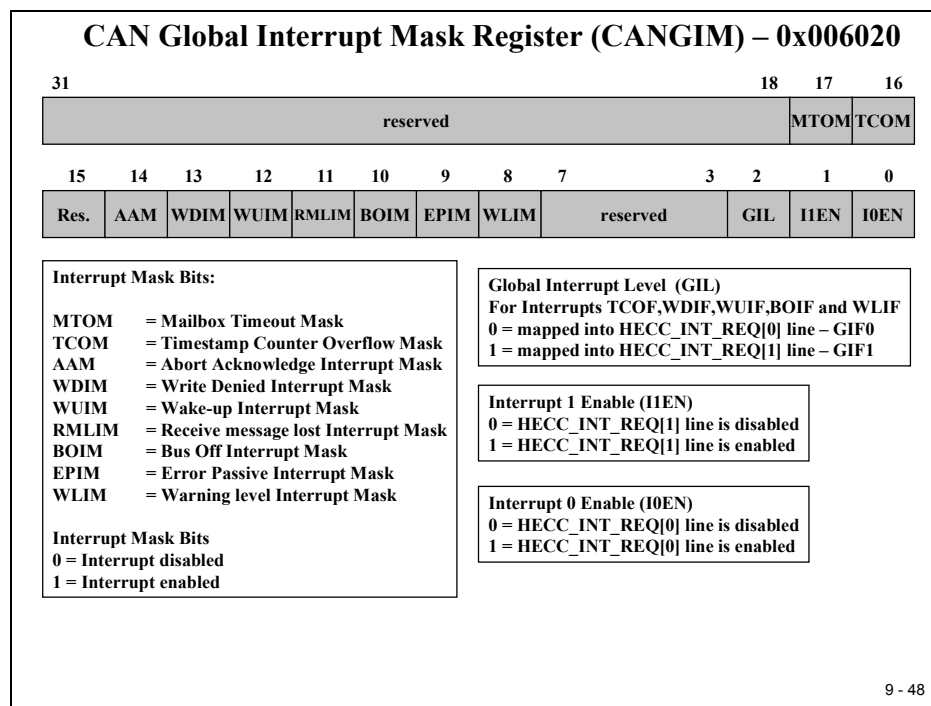


Transmit & Receive Error Counter - CANTEC / CANREC



CAN Interrupt Register

Global Interrupt Mask - CANGIM



CAN Global Interrupt Flag 0 Register (CANGIF0) – 0x00601E

31															18				17		16						
reserved																			MTOF0		TCOF0						
15		14		13		12		11		10		9		8		7-5		4		3		2		1		0	
GMIF0		AAIF0		WDIF0		WUIF0		RMLIF0		BOIF0		EPIF0		WLIF0		Res.		MIV0.4		MIV0.3		MIV0.2		MIV0.1		MIV0.0	

Interrupt Flag Bits:

MTOF0	= Mailbox Timeout Flag
TCOF0	= Timestamp Counter Overflow Flag
GMIF0	= Global Mailbox Interrupt Flag
AAIF0	= Abort Acknowledge Interrupt Flag
WDIF0	= Write Denied Interrupt Flag
WUIF0	= Wake-up Interrupt Flag
RMLIF0	= Receive message lost Interrupt Flag
BOIF0	= Bus Off Interrupt Flag
EPIF0	= Error Passive Interrupt Flag
WLIF0	= Warning level Interrupt Flag

Mailbox Interrupt Vector (MIV0)	Indicates the number of the message object that set the global mailbox interrupt flag (GMIF0)
----------------------------------------	-----------------------------------------------------------------------------------------------

Interrupt Flag Bits
0 = Interrupt has not occurred
1 = Interrupt has occurred

9 - 49

CAN Global Interrupt Flag 1 Register (CANGIF1) – 0x006022

31															18					17		16					
reserved																				MTOF1		TCOF1					
15		14		13		12		11		10		9		8		7-5		4		3		2		1		0	
GMIF1		AAIF1		WDIF1		WUIF1		RMLIF1		BOIF1		EPIF1		WLIF1		Res.		MIV1.4		MIV1.3		MIV1.2		MIV1.1		MIV1.0	

Interrupt Flag Bits:

MTOF1	= Mailbox Timeout Flag
TCOF1	= Timestamp Counter Overflow Flag
GMIF1	= Global Mailbox Interrupt Flag
AAIF1	= Abort Acknowledge Interrupt Flag
WDIF1	= Write Denied Interrupt Flag
WUIF1	= Wake-up Interrupt Flag
RMLIF1	= Receive message lost Interrupt Flag
BOIF1	= Bus Off Interrupt Flag
EPIF1	= Error Passive Interrupt Flag
WLIF1	= Warning level Interrupt Flag

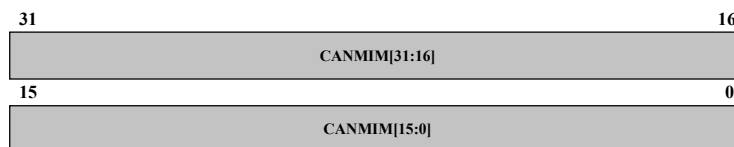
Mailbox Interrupt Vector (MIV1)
Indicates the number of the message object that set the global mailbox interrupt flag (GMIF1)

Interrupt Flag Bits
0 = Interrupt has not occurred
1 = Interrupt has occurred

9 - 50

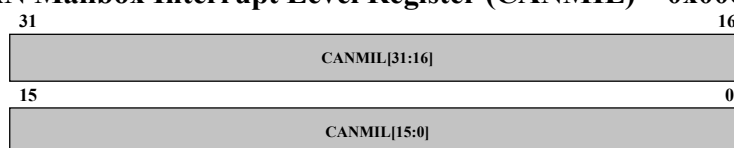
Mailbox Interrupt Mask - CANMIM

CAN Mailbox Interrupt Mask Register (CANMIM) – 0x006024



Mailbox Interrupt Mask Bits (MIM)
 0 = mailbox interrupt is disabled.
 1 = mailbox interrupt is enabled. An Interrupt is generated if a message has been transmitted successfully or if a message has been received without an error.

CAN Mailbox Interrupt Level Register (CANMIL) – 0x006026

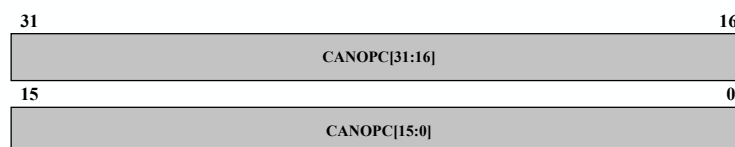


Mailbox Interrupt Level Bits (MIL)
 0 = mailbox interrupt is generated on HECC_INT_REQ[0] line.
 1 = mailbox interrupt is generated on HECC_INT_REQ[1] line.

9 - 51

Overwrite Protection Control - CANOPC

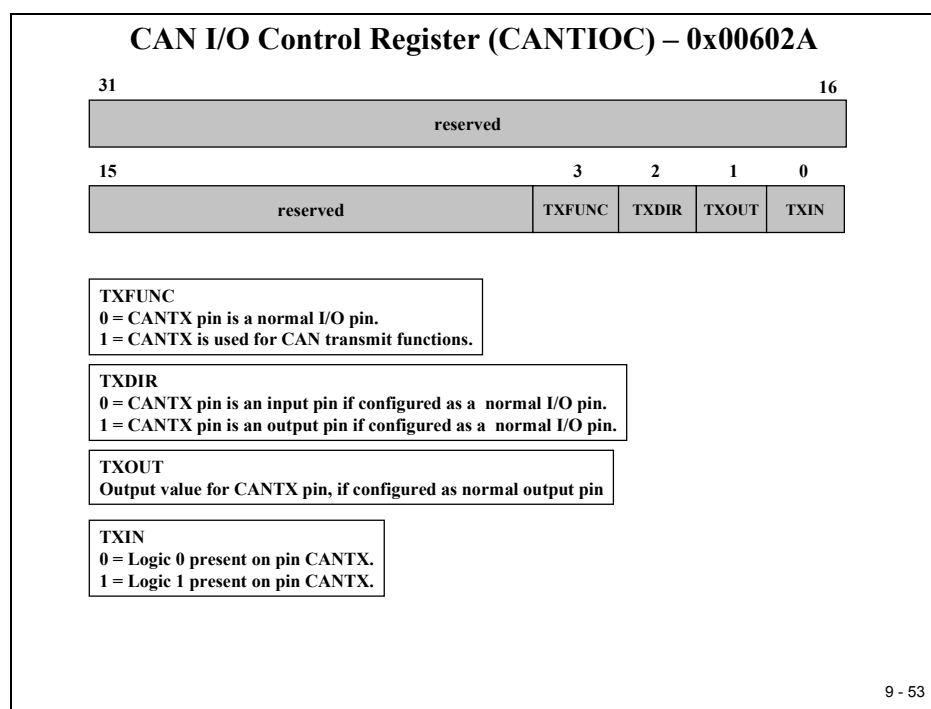
CAN Overwrite Protection Control Register (CANOPC) – 0x006028



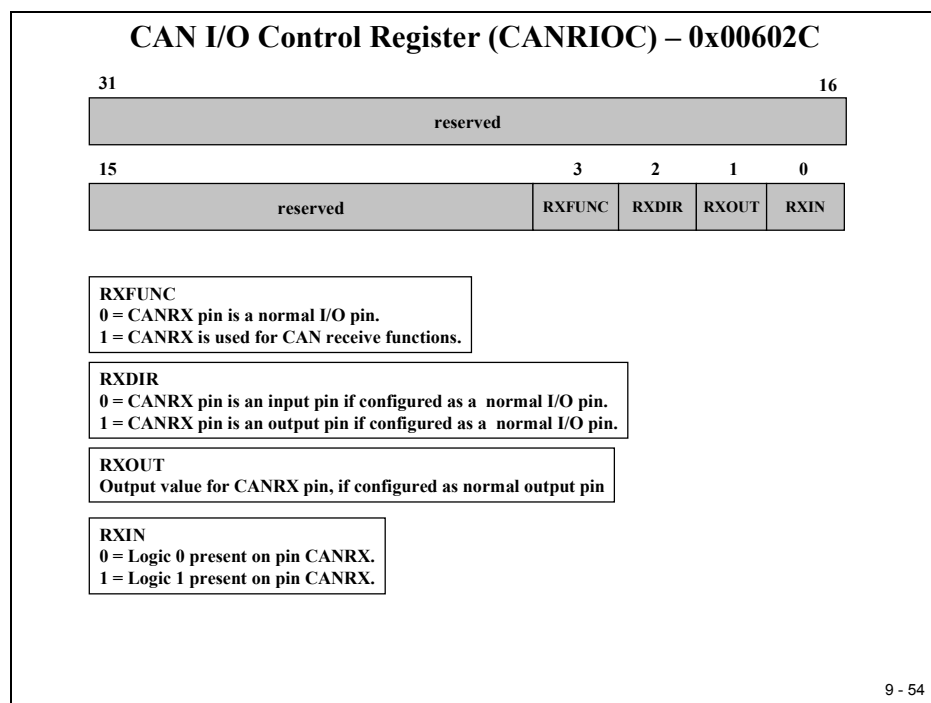
Overwrite Protection Control Bits (MIM)
 0 = the old message in mailbox n may be overwritten by a new one.
 This will be notified by the receive message lost bit RML[n].
 1 = an old message in mailbox n is protected against being overwritten by a new one.
 Thus, the next mailboxes are checked for a matching ID.
 If no other mailbox is found, the new message is lost.

9 - 52

Transmit I/O Control - CANTIOC



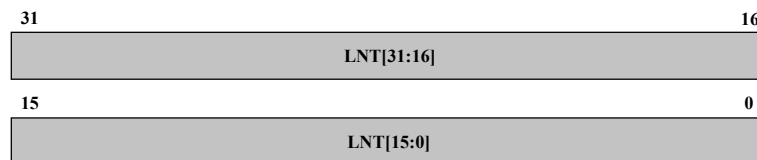
Receive I/O Control - CANRIOC



Alarm / Time Out Register

Local Network Time - CANLNT

CAN Local Network Time Register (CANLNT) – 0x00602E

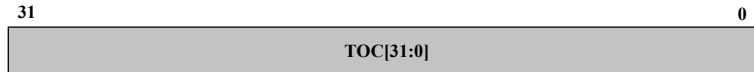


- ◆ LNT is a Free Running Counter, Clocked from the bit clock of the CAN module.
- ◆ LNT is written into the time stamp register (MOTS) of the corresponding mailbox when a received message has been stored or a message has been transmitted.
- ◆ LNT is cleared when mailbox #16 is transmitted or received. Thus mailbox #16 can be used for a global network time synchronization.

9 - 55

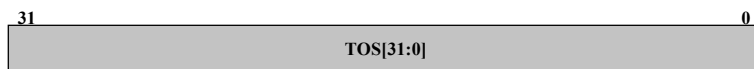
Time Out Control - CANTIOC

CAN Time Out Control Register (CANTOC) – 0x006030



Time Out Control Bits (TOC)
 0 = Time Out function is disabled for mailbox n.
 1 = Time Out function is enabled for mailbox n.
 If the corresponding MOTO register is greater than LNT a time out event will be generated

CAN Time Out Status Register (CANTOS) – 0x006032



Time Out Status Flags (TOS)
 0 = No Time Out occurred for mailbox n.
 1 = The value in LNT is greater or equal to the value in the corresponding MOTO register

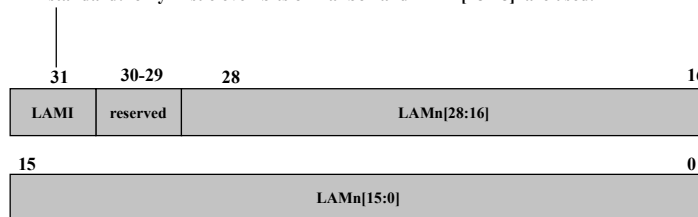
9 - 56

Local Acceptance Mask - LAMn

CAN Local Acceptance Mask Register

0x00 6040 - 0x00 607F

0 = IDE bit of mailbox determines which message shall be received
 1 = extended or standard frames can be received.
 extended: all 29 bit of LAM are used for filter against all 29 bit of mailbox .
 standard: only first eleven bits of mailbox and LAM [28-18] are used.



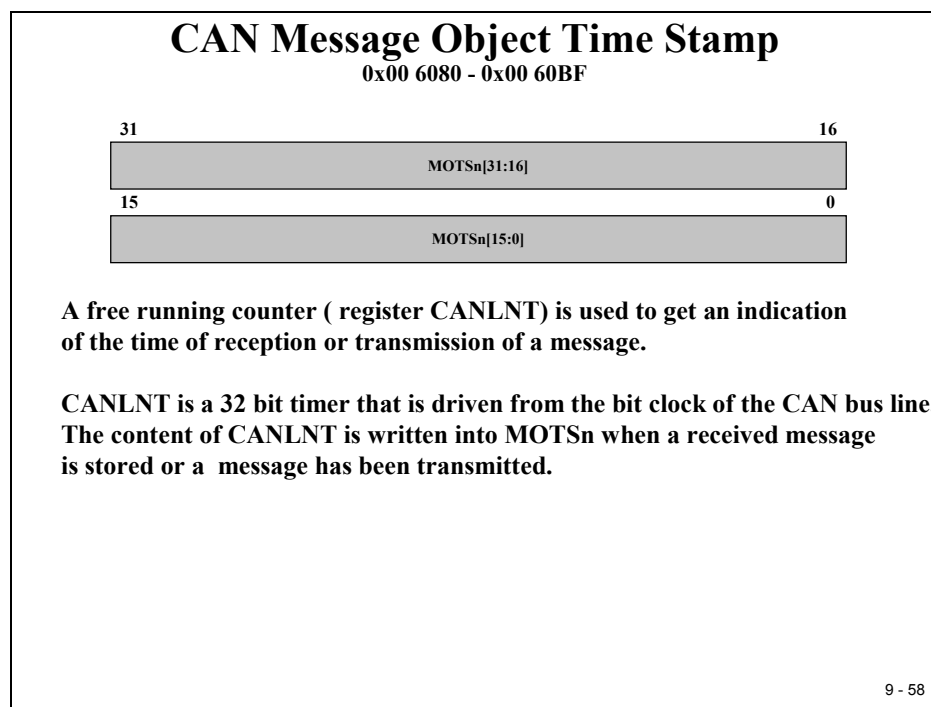
LAMn[28-0]: Masking of identifier bits of incoming messages
 1 = don't care (accept 1 or 0 for this bit position) of incoming identifier.
 0 = received identifier bit must match the corresponding message identifier bit (MID).

Note: There are two operating modes of the CAN module : "HECC" and "SCC".
 In "SCC" (default after reset) LAM0 is used for mailboxes 0 to 2, LAM3 is used for mailboxes 3 to 5 and the global acceptance mask (CANGAM) is used for mailboxes 6 to 15.

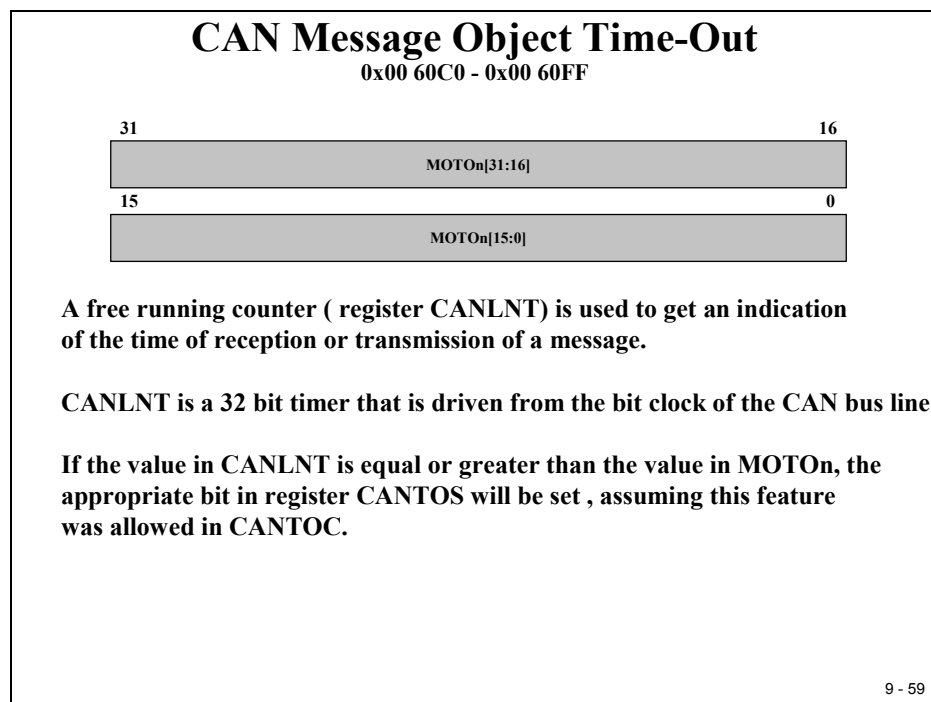
In "HECC" (CANMC:13 = 1) each mailbox has its own mask register LAM0 to LAM31.

9 - 57

Message Object Time Stamp - MOTSn

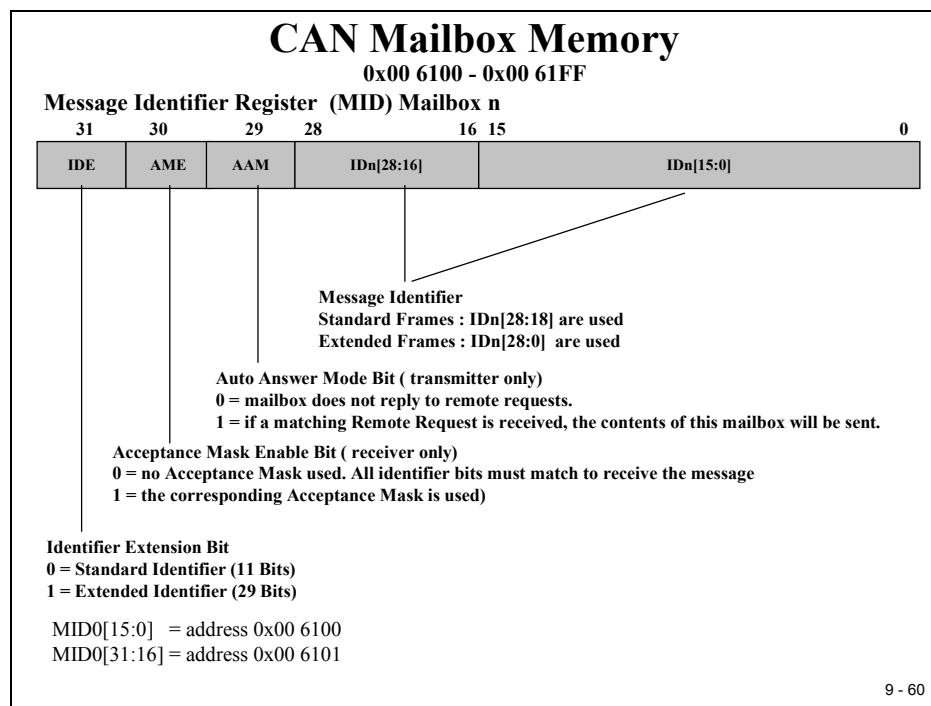


Message Object Time Out - MOTOn

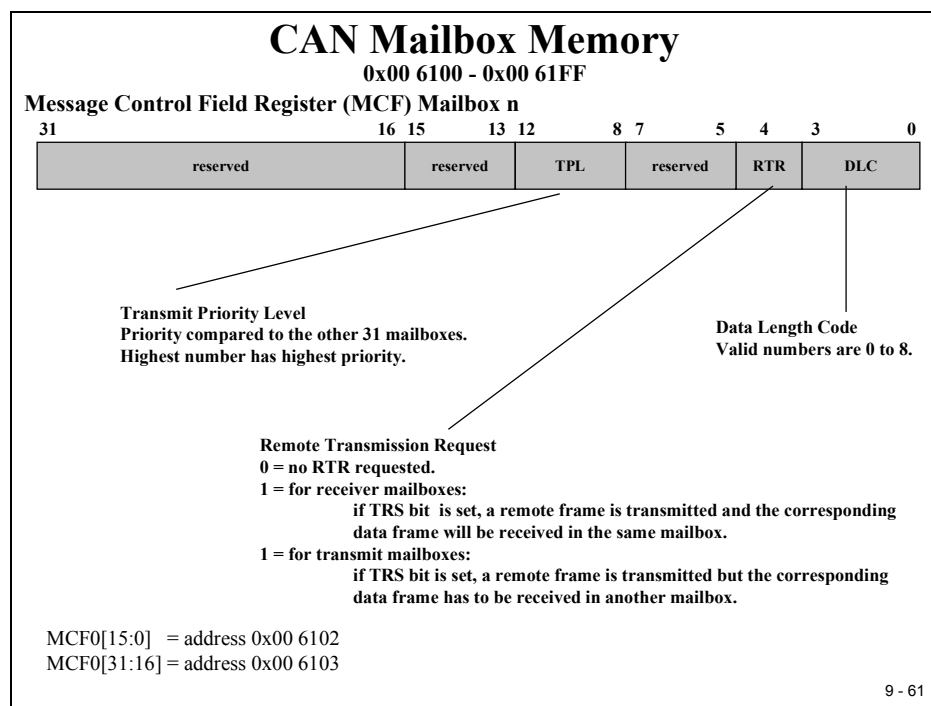


Mailbox Memory

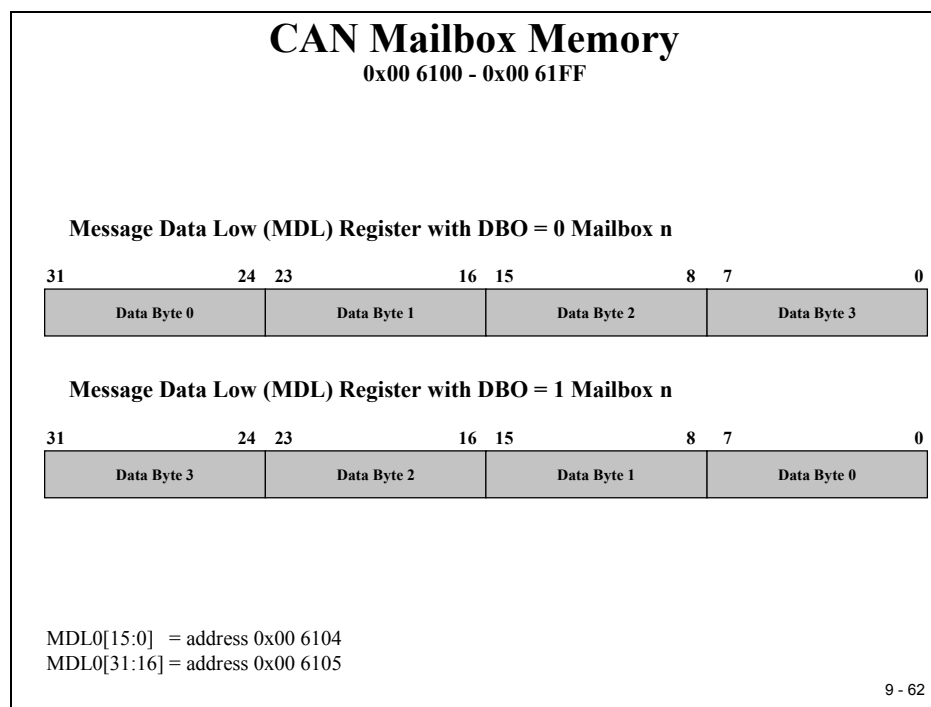
Message Identifier - CANMID



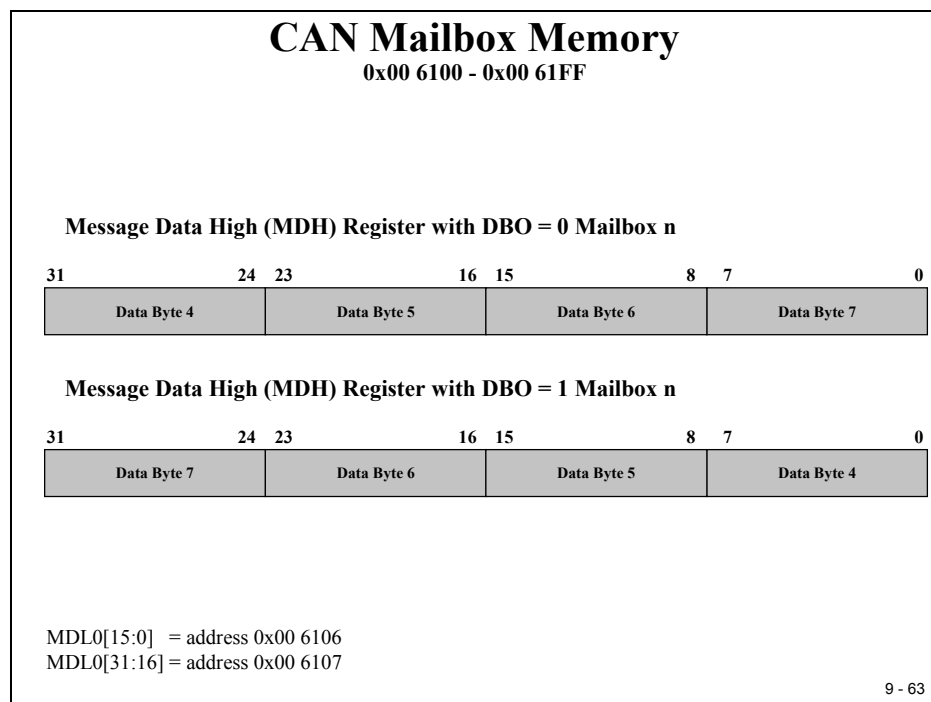
Message Control Field - CANMCF



Message Data Field Low - CANMDL



Message Data Field High - CANMDH



Lab Exercise 9

CAN Example : transmit a frame

◆ Lab 9: Transmit a CAN message

- CAN baud rate : 100 Kbps (CAN low speed)
- Transmit a one byte message every second
- Message Identifier 0x 1000 0000 (extended frame)
- Use Mailbox #5 as transmit mailbox
- Message content: status of the input switches (GPIO B15-B8)
- CAN transceiver SN 65 HVD 230 (Zwickau Adapter Board) :
 - Set jumper JP5 and JP6 to 1-2
 - Set jumper JP4 to 2-3 (enables on board line terminator of 120 Ohm)
- DB9 (male) to connect the Adapter Board to CAN
 - Pin 2 : CAN_L ; Pin 7 : CAN_H ; Pin 3 : GND

9 - 64

Preface

After this extensive description of all CAN registers of the C28x, it is time to carry out an exercise. Again, it is a good idea to start with some simple experiments to get our hardware to work. Later, we can try to refine the projects by setting up enhanced operation modes such as “Remote Transmission Request”, “Auto Answer Mode”, “Pipelined Mailboxes” or “Wakeup Mode”. We will also refrain from using the powerful error recognition and error management, which of course would be an essential part of a real project. To keep it simple, we will also use a polling method instead of an interrupt driven communication between the core of the DSP and the CAN mailbox server. Once you have a working example, it is much simpler to improve the code in this project by adding more enhanced operating modes to it.

The CAN requires a transceiver circuit between the digital signals of the C28x and the bus lines to adjust the physical voltages. The Zwickau Adapter Board is equipped with two different types of CAN transceivers, a Texas Instruments SN65HVD230 for high speed ISO 11898 applications and a Phillips TJA1054, quite often used in the CAN for body electronics of a car. With the help of two jumpers (JP5, JP6), we can select the transceiver in use. For Lab 9 we will use the SN65HVD230.

The physical CAN lines for ISO 11898 require a correct line termination at the ends of the transmission lines by 120 Ohm terminator resistors. If the C28x is placed at one of the end positions in your CAN network, you can use the on board 120 Ohm terminator by setting jumper JP4 to position 2-3. If the physical structure of the CAN in your laboratory does

not require the C28x to terminate the net, set JP4 to 1-2. Ask your teacher which set up is the correct one.

To test your code, you will need a partner team with a second C28x doing Lab 10. This lab is an experiment to receive a CAN message and display its data at GPIO B7-B0 (8 LED's on the Zwickau Adapter Board). If your laboratory does not provide any CAN infrastructure, it is quite simple to connect the two boards. Use two female DB9 connectors, a twisted pair cable to connect pins 2-2 (CAN_L), 7-7 (CAN_H) and eventually 3-3 (GND) and plug them into the DB9 connectors of the Zwickau Adapter Board.

Before you start the hard wiring, ask your teacher or a laboratory technician what exactly you are supposed to do to connect the boards!

Objective

- The objective of Lab 9 is to transmit a one byte data frame every second via CAN.
- The actual data byte should be taken from input lines GPIO-B15 to B8. In case of the Zwickau Adapter Board, these 8 lines are connected to 8 digital input switches.
- The baud rate for the CAN should be set to 100 kbps.
- The exercise should use extended identifier 0x1000 0000 for the transmit message. You can also use any other number as identifier, but please make sure that your partner team (Lab 10) knows about your change. If your classroom uses several eZdsp's at the same time, it could be an option to set-up pairs of teams sharing the CAN by using different identifiers. It is also possible that due to the structure of the laboratory set-up at your university, not all identifier combinations might be available to you. You surely don't want to start the ignition of a motor control unit that is also connected to the CAN for some other experiments. Before you use any other ID's → ask your teacher!
- Use Mailbox #5 as your transmit mailbox
- Once you have started a CAN transmission wait for completion by polling the status bit. Doing so we can refrain from using CAN interrupts for this first CAN exercise.
- Use CPU core timer 0 to generate the one second interval

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab9.pjt** in E:\C281x\Labs.

2. A good point to start with is the source code of Lab4, which produces a hardware based time period using CPU core timer 0. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab9.c in E:\C281x\Labs\Lab9.
3. Add the source code file to your project:
 - **Lab9.c**
4. From C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source add:
 - **DSP281x_GlobalVariableDefs.c**

From C:\tidcs\c28\dsp281x\v100\DSP281x_common\source add:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**
- **DSP281x_CpuTimers.c**

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

- **F2812_Headers_nonBIOS.cmd**

From C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd add:

- **F2812_EzDSP_RAM_Ink.cmd**

From C:\ti\c2000\cgtoolslib add:

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

- Before we can start editing our own code we have to modify a portion of the Texas Instruments Header file “**DSP281x_ECan.h**”, Version 1.0. This file has a bug inside the structures “CANMDL_BYTES” and “CANMDH_BYTES”. The order of bytes is not correct.

Search and edit struct CANMDL_BYTES and CANMDH_BYTES:

```
struct CANMDL_BYTES {    // bits  description
```

```
    Uint16    BYTE3:8;    // 31:24
```

```
    Uint16    BYTE2:8;    // 23:16
```

```
    Uint16    BYTE1:8;    // 15:8
```

```
    Uint16    BYTE0:8;    // 7:0
```

```
};
```

```
struct CANMDH_BYTES {    // bits  description
```

```
    Uint16    BYTE7:8;    // 63:56
```

```
    Uint16    BYTE6:8;    // 55:48
```

```
    Uint16    BYTE5:8;    // 47:40
```

```
    Uint16    BYTE4:8;    // 39:32
```

```
};
```

- Open Lab9.c to edit. First, we have to adjust the while(1) loop of main to perform the next CAN transmission every 1 second. Recall that we initialized the CPU core timer 0 to interrupt every 50ms and to increment variable “CpuTimer0.InterruptCount” with every interrupt service. To generate a pause period of 1 second, we just have to wait until “CpuTimer0.InterruptCount” has reached 20. BUT, while we wait, we have to deal with the watchdog! One second without any watchdog service will be far too long; the watchdog will trigger a reset! Modify the code accordingly!

Build, Load and Run

9. Before we continue to modify our source code, let us try to compile the project and run a test. If everything goes as expected, the DSP should perform the LED Knight-Rider from Lab4, now with a pause interval of one second between the steps.

Click the “Rebuild All” button or perform:

Project → Build
File → Load Program
Debug → Reset CPU
Debug → Restart
Debug → Go main
Debug → Run(F5)

Modify Source Code Cont.

10. Congratulations! Now the tougher part is waiting for you. You will have to add code to initialize the CAN module. Let’s do it again using a step-by-step approach.

First, delete the variables “i” and “LED[8]” of main. Next, add a new structure “ECanaShadow” as a local variable in main:

struct ECAN_REGS ECanaShadow;

This structure will be used as a local copy of the original CAN registers. A manipulation of individual bits is done inside the copy. At the end of the access, the whole copy is reloaded into the original CAN structures. This operation is necessary because of the inner structure of the CAN unit; some registers are only accessible by 32-bit accesses and by copying the whole structure, we make sure to generate 32 bit accesses only.

11. In function “InitSystem()” enable the clock unit for the CAN module.
12. Next, inside function “Gpio_select()” enable the peripheral function of CANTxA and CANRxA:

GpioMuxRegs.GPFMUX.bit.CANTXA_GPIOF6 = 1;
GpioMuxRegs.GPFMUX.bit.CANRXA_GPIOF7 = 1;

Add the CAN initialization code

13. Add a new function “InitCan()” at the end of your source code to initialize the CAN module. Inside “InitCan()” add the following steps:
 - In Register “ECanaRegs.CANTIOC” and “ECanaRegs.CANRIOC” configure the two pins “TXFUNC” and “RXFUNC” for CAN.

- Enable the HECC mode of the CAN module (Register “ECanaRegs.CANMC”).
- To set-up the baud rate for the CAN transmission, we need to get access to the bit timing registers. This access is requested by setting bit “CCR” of register “ECanaRegs.CANMC” to 1.
- Before we can continue with the initialisation, we have to wait until the CAN module has granted this request. In this case the flag “CCE” of register “ECanaRegs.CANES” will be set to 1 by the CAN module. Install a wait construct into your code.
- Now we are allowed to set-up the bit timing parameters “BRP”, “TSEG1” and “TSEG2” of register “ECanaRegs.CANBTC”. Use the 100 kbps set-up from the following table:

CAN Bit-Timing Examples

◆ Bit Configuration for SYSCLK = 150 MHz

◆ Sample Point at 80% of Bit Time :

CAN-Baudrate	BRP	TSEG1	TSEG2
1 MBPS	9	10	2
500 KBPS	19	10	2
250 KBPS	39	10	2
125 KBPS	79	10	2
100 KBPS	99	10	2
50 KBPS	199	10	2

◆ Example 50 KBPS:

$$TQ = (199+1)/150 \text{ MHz} = 1.334 \text{ ns}$$

$$t_{seg1} = 1.334 \text{ ns} (10 + 1) = 14.674 \text{ ns} \rightarrow t_{CAN} = 20.010 \text{ ns}$$

$$t_{seg2} = 1.334 \text{ ns} (2 + 1) = 4.002 \text{ ns}$$

9 - 44

- After the access to register “ECanaRegs.CANBTC”, we have to re-enable the CAN modules access to this register. This is done by clearing bit “Change Configuration Request (CCR)” of register “ECanaRegs.CANMC”. Again we have to apply a wait loop until this command is acknowledged by the CAN module (Flag “CCE” of register “ECanaRegs.CANES” will be cleared by the CAN module as acknowledgement).
- Finally, we have to disable all mailboxes to exclude them from data communication and to allow write accesses into the message identifier registers of the mailbox of our choice. To disable all mailboxes we have to write a ‘0’ into all bit fields of register “ECanaRegs.CANME”.

14. In main, just before the CpuCoreTimer0 is started, add the function call of “InitCan()”.
15. At the beginning of your code, add a function prototype for “InitCan()”

Prepare Transmit Mailbox #5

16. In main, after the function call to “InitCan()”, add code to prepare the transmit mailbox. In this exercise, we will use mailbox #5, an extended identifier of 0x10000000 and a data length code of 1. Add the following steps:

- Write the identifier into register “ECanaMboxes.MBOX5.MSGID”.
- To transmit with extended identifiers set bit “IDE” of register “ECanaMboxes.MBOX5.MSGID” to 1.
- Configure Mailbox #5 as a transmit mailbox. This is done by setting bit MD5 of register “ECanaRegs.CANMD” to 0. Caution! Due to the internal structure of the CAN-unit, we can’t execute single bit accesses to the original CAN registers. A good principle is to copy the whole register into a shadow register, manipulate the shadow and copy the modified 32 bit shadow back into its origin:

ECanaShadow.CANMD.all = ECanaRegs.CANMD.all;

ECanaShadow.CANMD.bit.MD5 = 0;

ECanaRegs.CANMD.all = ECanaShadow.CANMD.all;

- Enable Mailbox #5:

ECanaShadow.CANME.all = ECanaRegs.CANME.all;

ECanaShadow.CANME.bit.ME5 = 1;

ECanaRegs.CANME.all = ECanaShadow.CANME.all;

- Set-up the Data Length Code Field (DLC) in Message Control Register “ECanaMboxes.MBOX5.MSGCTRL” to 1.

Add the Data Byte and Transmit

17. Now we are almost done. The only thing that’s missing is the periodical loading of the data byte into the mailbox and the transmit request command. This must be done inside the while(1)-loop of main. Locate the code where we wait until variable “CpuTimer0.InterruptCount” has reached 20. Here add:

- Load the current status of the 8 input switches at GPIO-Port B (Bits 15 to 8) into register “ECanaMboxes.MBOX5.MDL.byte.BYTE0”
- Request a transmission of mailbox #5. Init register “ECanaShadow.CANTRS”. Set bit TRS5=1 and all other 31 bits to 0. Next, load the whole register into “ECanaRegs.CANTRS”

- Wait until the CAN unit has acknowledged the transmit request. The flag “ECanaRegs.CANTA.bit.TA5” will be set to 1 if your request has been acknowledged.
- Clear bit “ECanaRegs.CANTA.bit.TA5”. Again the access must be made as a 32 bit access:

ECanaShadow.CANTA.all = 0;

ECanaShadow.CANTA.bit.TA5 = 1;

ECanaRegs.CANTA.all = ECanaShadow.CANTA.all;

Build, Load and Run

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

Project → Build
File → Load Program
Debug → Reset CPU
Debug → Restart
Debug → Go main
Debug → Run(F5)

19. Providing you have found a partner team with another C28x connected to your laboratory CAN system and waiting for a one-byte data frame with identifier 0x10000000 you can do a real network test. Modify the status of your input switches. The current status should be transmitted every second via CAN.

If your teacher can provide a CAN analyser you should be able to trace your data frames at the CAN. Your partner team should be able to receive your frames and use the information to update their LED's.

If you end up in a fight between the two teams about whose code might be wrong, ask your teacher to provide a working receiver node. Recommendation for teachers: Store a working receiver code version in the internal Flash of one node and start this node out of flash memory.

END of LAB 9

Lab Exercise 10

CAN Example : receive a frame

◆ Lab 10: Receive a CAN message

- CAN baud rate : 100 Kbps (can low speed)
- Receive a one byte message and show it on GPIO-Port B7...B0 (8 LED's)
- Message Identifier 0x 1000 0000 (extended frame)
- Use Mailbox #1 as receive mailbox
- CAN Transceiver SN 65 HVD 230 (Zwickau Adapter Board) :
 - Set jumper JP5 and JP6 to 1-2
 - Set jumper JP4 to 2-3 (enables on board line terminator of 120 Ohm)
- DB9 (male) to connect the Adapter Board to CAN
 - Pin 2 : CAN_L ; Pin 7 : CAN_H ; Pin 3 : GND

9 - 65

Preface

This laboratory experiment is the second part of a CAN-Lab. Again we have to set up the physical CAN-layer according to the layout of your laboratory.

The CAN requires a transceiver circuit between the digital signals of the C28x and the bus lines to adjust the physical voltages. The Zwickau Adapter Board is equipped with two different types of CAN transceivers, a Texas Instruments SN65HVD230 for high speed ISO 11898 applications and a Phillips TJA1054, quite often used in the CAN for body electronics of a car. With the help of two jumpers (JP5, JP6), you can select the transceiver in use. For Lab 10 we will use the SN65HVD230.

The physical CAN lines for ISO 11898 require a correct line termination at the ends of the transmission lines by 120 Ohm terminator resistors. If the C28x is placed at one of the end positions in your CAN network, you can use the on-board terminator of 120 Ohms by setting jumper JP4 to position 2-3. If the physical structure of the CAN in your laboratory does not require the C28x to terminate the net, set JP4 to 1-2. Ask your teacher which set-up is the correct one.

To test your code you will need a partner team with a second C28x doing Lab 9, e.g. sending a one byte message with identifier 0x10 000 000 every second.

Before you start the hard wiring, ask your teacher or a laboratory technician what exactly you are supposed to do to connect the boards!

Objective

- The objective of Lab 10 is to receive a one byte data frame every time it is transmitted via CAN, and update the status of the 8 output lines GPIO-B7...B0 (8 LED's) with the data information..
- The baud rate for the CAN should be set to 100 KBPS.
- The exercise should use extended identifier 0x1000 0000 for the receive filter of mailbox 1. You can also use any other number as identifier, but please make sure that your partner team (Lab 9) knows about your change. If your classroom uses several eZdsp's at the same time, it could be an option to set up pairs of teams sharing the CAN by using different identifiers. It is also possible, that due to the structure of the laboratory set-up of your university, not all identifier combinations might be available to you. You surely don't want to start the ignition of a motor control unit that is also connected to the CAN for some other experiments. Before you use any other ID's → ask your teacher!
- Use Mailbox #1 as your receiver mailbox
- Once you have initialized the CAN module wait for a reception of mailbox #1 by polling the status bit. Doing so we can refrain from using CAN interrupts for this first CAN exercise.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab10.pjt** in E:\C281x\Labs.
2. A good point to start with is the source code of Lab4, which produces a hardware based time period using CPU core timer 0. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab10.c in E:\C281x\Labs\Lab10.
3. Add the source code file to your project:
 - **Lab10.c**
4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:
 - **DSP281x_GlobalVariableDefs.c**From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\cmd* add:
 - **F2812_Headers_nonBIOS.cmd**From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\cmd* add:
 - **F2812_EzDSP_RAM_Ink.cmd**

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

- **rts2800_ml.lib**

Project Build Options

5. 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 Source Code

7. Before we can start editing our own code we have to modify a portion of the Texas Instruments Header file “**DSP281x_ECan.h**”, Version 1.0. This file has a bug inside the structures “CANMDL_BYTES” and “CANMDH_BYTES”. The order of bytes is not correct.

Search and edit struct CANMDL_BYTES and CANMDH_BYTES:

```
struct CANMDL_BYTES {    // bits  description
```

```
    Uint16    BYTE3:8;    // 31:24
```

```
    Uint16    BYTE2:8;    // 23:16
```

```
    Uint16    BYTE1:8;    // 15:8
```

```
    Uint16    BYTE0:8;    // 7:0
```

```
    };
```

```
struct CANMDH_BYTES {    // bits  description
```

```
    Uint16    BYTE7:8;    // 63:56
```

```
    Uint16    BYTE6:8;    // 55:48
```



```
    Uint16    BYTE5:8;    // 47:40

    Uint16    BYTE4:8;    // 39:32

};
```

8. Open Lab10.c to edit.

Remove the function prototype and the definition of function “**cpu_timer0_isr()**.” We do not use the CPU core timer in this lab exercise.

In “**main()**”, remove the local variables “**i**” and “**LED[8]**”.

Between the start of “**main**” and the “**while(1)**-loop of “**main()**”, remove all function calls apart from “**InitSystem()**” and “**Gpio_select()**”.

Inside the **while(1)**-loop remove all old lines, just keep the service instructions for the watchdog:

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

Build, Load and Run

9. Before we continue to modify our source code lets try to compile the project in this stage to find any syntax errors.

Click the “Rebuild All” button or perform:

```
Project → Build
File → Load Program
Debug → Reset CPU
Debug → Restart
Debug → Go main
Debug → Run(F5)
```

If everything went like expected you should end up with 0 errors, 0 warnings and 0 remarks.

Modify Source Code Cont.

10. Congratulations! Now let’s install the CAN – receiver part.

First, add a new structure “**ECanaShadow**” as a local variable in **main**:

```
    struct ECAN_REGS ECanaShadow;
```

This structure will be used as a local copy of the original CAN registers. A manipulation of individual bits is done inside the copy. At the end of the access the whole copy is reloaded into the original CAN structures. This principle of operation is necessary because of the inner structure of the CAN unit; some registers are only accessible by 32-bit accesses and by copying the whole structure, we make sure to generate 32-bit accesses only.

11. In function “InitSystem()” enable the clock unit for the CAN module.
12. Next, inside function “Gpio_select()”, enable the peripheral function of CANTxA and CANRxA:

```
GpioMuxRegs.GPFMUX.bit.CANTXA_GPIOF6 = 1;  
GpioMuxRegs.GPFMUX.bit.CANRXA_GPIOF7 = 1;
```

Add the CAN initialization code

13. Add a new function “InitCan()” at the end of your source code to initialize the CAN module. Inside “InitCan()”, add the following steps:
 - In Register “ECanaRegs.CANTIOC” and “ECanaRegs.CANRIOC” configure the two pins “TXFUNC” and “RXFUNC” for CAN.
 - Enable the HECC mode of the CAN module (Register “ECanaRegs.CANMC”).
 - To set-up the baud rate for the CAN transmission we need to get access to the bit timing registers. This access is requested by setting bit “CCR” of register “ECanaRegs.CANMC” to 1.
 - Before we can continue with the initialization we have to wait until the CAN module has granted this request. In this case, the flag “CCE” of register “ECanaRegs.CANES” will be set to 1 by the CAN module. Install a wait construct into your code.
 - Now we are allowed to set-up the bit timing parameters “BRP”, “TSEG1” and “TSEG2” of register “ECanaRegs.CANBTC”. Use the 100 kbps set-up from the following table:

CAN Bit-Timing Examples

◆ **Bit Configuration for SYSCLK = 150 MHz**

◆ **Sample Point at 80% of Bit Time :**

CAN-Baudrate	BRP	TSEG1	TSEG2
1 MBPS	9	10	2
500 KBPS	19	10	2
250 KBPS	39	10	2
125 KBPS	79	10	2
100 KBPS	99	10	2
50 KBPS	199	10	2

◆ **Example 50 KBPS:**

$$TQ = (199+1)/150 \text{ MHz} = 1.334 \text{ ns}$$

$$t_{seg1} = 1.334 \text{ ns} (10 + 1) = 14.674 \text{ ns} \rightarrow t_{CAN} = 20.010 \text{ ns}$$

$$t_{seg2} = 1.334 \text{ ns} (2 + 1) = 4.002 \text{ ns}$$

9 - 44

- After the access to register “ECanaRegs.CANBTC” we have to re-enable the CAN modules access to this register. This is done by clearing bit “Change Configuration Request (CCR)” of register “ECanaRegs.CANMC”. Again, we have to apply a wait loop until this command is acknowledged by the CAN module (Flag “CCE” of register “ECanaRegs.CANES” will be cleared by the CAN module as acknowledgement).
- Finally, we have to disable all mailboxes to exclude them from data communication and to allow write accesses into the message identifier registers of the mailbox of our choice. To disable all mailboxes we have to write a ‘0’ into all bit fields of register “ECanaRegs.CANME”.

14. In main, just before we enter the while(1)-loop, add the function call to “InitCan()”.

15. At the beginning of your code, add a function prototype for “InitCan()”

Prepare Receiver Mailbox #1

16. In main, after the function call of “InitCan()” add code to prepare the receiver mailbox. In this exercise, we will use mailbox #1, an extended identifier of 0x10000000 and a data length code of 1. Add the following steps:

- Write the identifier into register “EcanMboxes.MBOX1.MSGID”.
- To transmit with extended identifiers set bit “IDE” of register “EcanMboxes.MBOX1.MSGID” to 1.
- Configure Mailbox #1 as a receive mailbox. This is done by setting bit MD1 of register “ECanaRegs.CANMD” to 1. Caution! Due to the internal structure of the CAN-unit, we can’t execute single bit accesses to the original CAN registers. A

good principle is to copy the whole register into a shadow register, manipulate the shadow and copy the modified 32 bit shadow back into its origin:

ECanaShadow.CANMD.all = ECanaRegs.CANMD.all;

ECanaShadow.CANMD.bit.MD1 = 1;

ECanaRegs.CANMD.all = ECanaShadow.CANMD.all;

- Enable Mailbox #1:

ECanaShadow.CANME.all = ECanaRegs.CANME.all;

ECanaShadow.CANME.bit.ME1 = 1;

ECanaRegs.CANME.all = ECanaShadow.CANME.all;

Add a polling loop for a message in mailbox 1

17. Now we are almost done. The only thing that's missing is the final modification of the while(1)-loop of main. All we have to add is a polling loop to wait for a received message in mailbox #1. The register "ECanaRegs.CANRMP" – Bit field "RMP1" will be set to 1 if a valid message has been received. All we have to do is to wait for this event to happen in a sort of "do-while" loop.

NOTE1: It is highly recommended to copy ECanaRegs.CANRMP into the local variable "ECanaShadow.CANRMP" before any logical test of bit RMP1 is made.

NOTE2: Do not forget to include the watchdog-service code lines into your wait construct!

18. If Bit RMP1 was set to 1 by the CAN – Mailbox we can take the data byte 0 out of the mailbox and load it onto the GPIO-B7...B0 (8 LED's):

GpioDataRegs.GPBDAT.all = ECanaMboxes.MBOX1.MDL.byte.BYTE0;

19. Finally, we have to reset bit RMP1. This is done by writing a '1' into it:

ECanaShadow.CANRMP.bit.RMP1 = 1;

ECanaRegs.CANRMP.all = ECanaShadow.CANRMP.all;

Build, Load and Run again

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

Project → Build
File → Load Program
Debug → Reset CPU
Debug → Restart
Debug → Go main
Debug → Run(F5)

21. Providing you have found a partner team with another C28x connected to your laboratory CAN system and transmitting a one-byte data frame with identifier 0x10000000 you can do a real network test. Ask your partner team to modify their input switches and transmit it every second via CAN.

If your teacher can provide a CAN analyzer you can also generate a transmit message out of this CAN analyzer.

If you end up in a fight between the two teams about whose code might be wrong, ask your teacher to provide a working transmitter node.

Recommendation for teachers: Store a working transmitter code version in the internal Flash of one node and start this node out of flash memory.

END of LAB 10

What's next?

Congratulations! You've successfully finished your first two lab exercises using Controller Area Network. As mentioned earlier in this chapter these two labs were chosen as a sort of "getting started" with CAN. To learn more about CAN it is necessary to book additional classes at your university.

To experiment a little bit more with CAN, choose one of the following **optional exercises**:

Lab 10A :

Combine Lab9 (CAN – Transmit) and Lab10 (CAN-Receive) into a bi-directional solution. The task for your node is to transmit the status of the input switches (B15...B8) to CAN every second (or optional: every time the status has changed) with a one-byte frame and identifier 0x10 000 000. Simultaneously, your node is requested to receive CAN messages with identifier 0x11 000 000. Byte 1 of the received frame should be displayed at the GPIO-Port pins B7...B0, which in case of the Zwickau Adapter board are connected to 8 LED's.

Lab 10B:

Try to improve Lab9 and Lab10A by using the C28x Interrupt System for the receiver part of the exercises. Instead of polling the "CANRMP-Bit field" to wait for an incoming message your task is to use a mailbox interrupt request to read out the mailbox when necessary.

Lab 10C:

We did not consider any possible error situations on the CAN side so far. That's not a good solution for a practical project. Try to improve your previous CAN experiments by including the service of potential CAN error situations. Recall, the CAN error status register flags all possible error situations. A good solution would be to allow CAN error interrupts to request their individual service routines in case of a CAN failure. What should be done in the case of an error request? Answer: Try to use the PWM – loudspeaker at output line T1PWM to generate a sound. By using different frequencies, you can signal the type of failure.

Another option could be to monitor the status of the two CAN – error counters and show their current values with the help of the 8 LED's at GPIO-B7...B0.

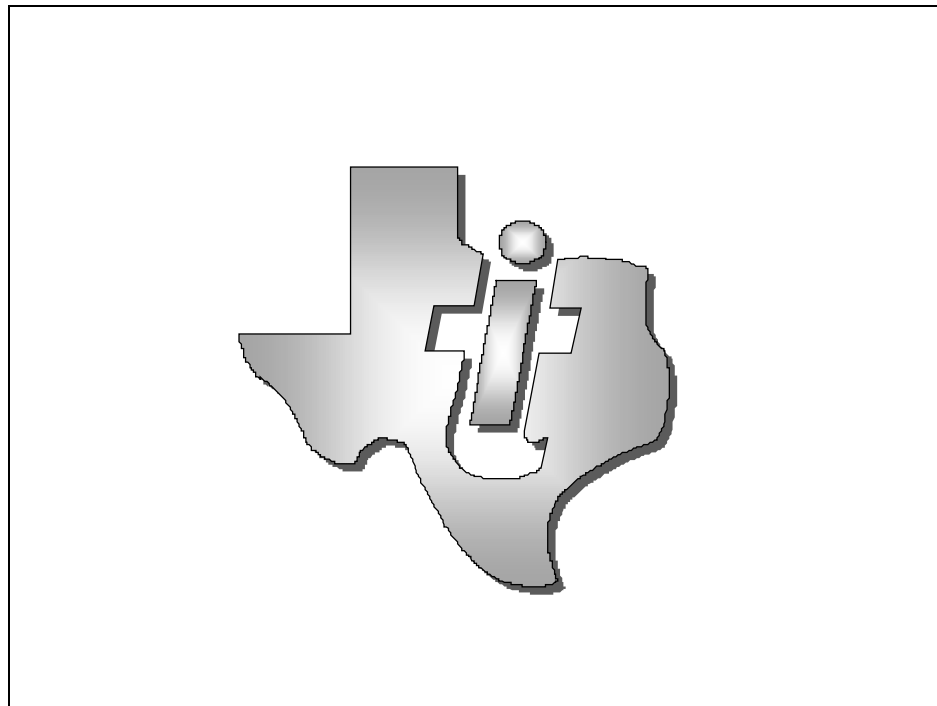
If your laboratory is equipped with a CAN failure generator like "CANstress" (Vector Informatik GmbH, Germany) you can generate reproducible disturbance of the physical layer, you can destroy certain messages and manipulate certain bit fields with bit resolution. Ask your laboratory technician whether you are allowed to use this type of equipment to invoke CAN errors.

Lab 10D:

An enhanced experiment is to request a remote transmission from another CAN-node. An operating mode, that is quite often used is the so-called "automatic answer mode". A transmit mailbox, that receives a remote transmission request ("RTR") answers automatically by transmitting a predefined frame. Try to establish this operating mode for the transmitter node

(Lab9 or Lab10B). Wait for a RTR and send the current status of the input switches back to the requesting node. The node that has requested the remote transmission should be initialized to wait for the requested answer and display byte 1 of the received data frame at the 8 LED's (GPIO B7...B0).

There are a lot more options for RTR operations available. Again, look out for additional CAN classes at your university!



C28x Flash Programming

Introduction

So far we have used the C28x internal volatile memory (H0 – SARAM) to store the code of our examples. Before we could execute the code we used Code Composer Studio to load it into H0-SARAM (“File” → “Load Program”). This is fine for projects in a development and debug phase with frequent changes to parts and components of the software. However, when it comes to production versions with a standalone embedded control unit based on the C28x, we no longer have the option to download our control code using Code Composer Studio. Imagine a control unit for an automotive braking system, where you have to download the control code first when you hit the brake pedal (“Do you really want to brake? ...”).

For standalone embedded control applications, we need to store our control code in NON-Volatile memory. This way it will be available immediately after power-up of the system. The question is, what type of non-volatile memory is available? There are several physically different memories of this type: Read Only Memory (ROM), Electrically Programmable Read Only Memory (EPROM), Electrically Programmable and Erasable Read Only Memory (EEPROM) and Flash-Memory. In case of the F2812, we can add any of the memory to the control unit using the external interface (XINTF).

The F2812 is also equipped with an internal Flash memory area of 128Kx16. This is quite a large amount of memory and more than sufficient for our lab exercises!

Before we can go to modify one of our existing lab solutions to start out of Flash we have to go through a short explanation of how to use this memory. This module also covers the boot sequence of the C28x - what happens when we power on the C28x?

Chapter 10 also covers the password feature of the C28x code security module. This module is used to embed dedicated portions of the C28x memory in a secure section with a 128bit-password. If the user does not know the correct combination that was programmed into the password section any access to the secured areas will be denied! This is a security measure to prevent reverse-engineering.

At the end of this lesson we will do a lab exercise to load one of our existing solutions into the internal Flash memory.

CAUTION: Please do not upset your teacher by programming the password area! Be careful, if you program the password by accident the device will be locked for ever! If you decide to make your mark in your university by locking the device with your own password, be sure to have already passed all exams.

Module Topics

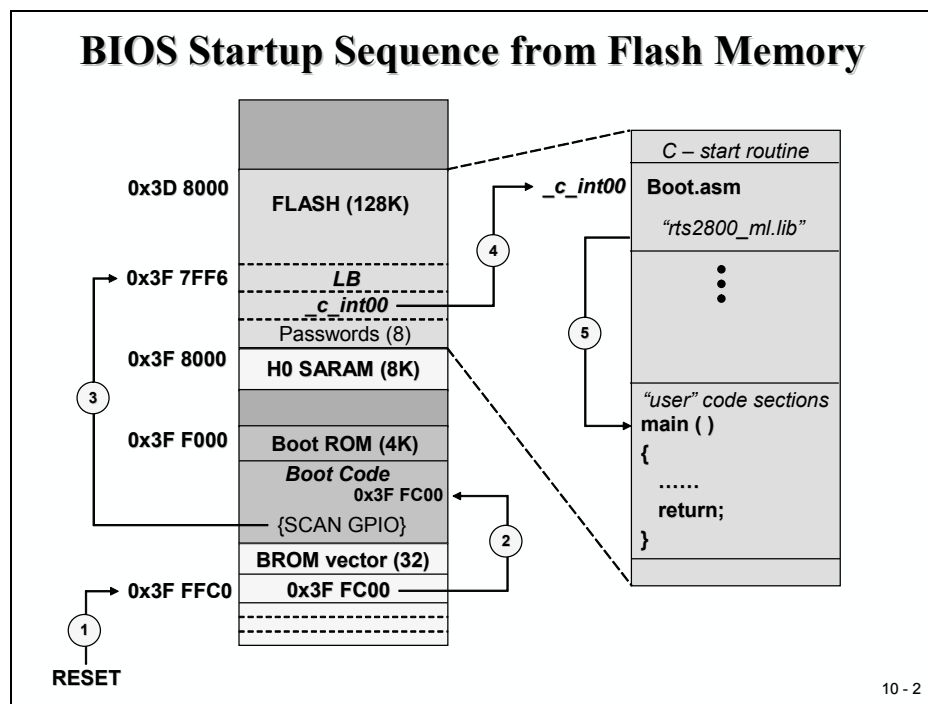
C28x Flash Programming	10-1
<i>Introduction</i>	<i>10-1</i>
<i>Module Topics.....</i>	<i>10-2</i>
<i>C28x Start-up Sequences</i>	<i>10-3</i>
<i>C28x Flash Memory Sectors</i>	<i>10-4</i>
<i>Flash Speed Initialization</i>	<i>10-5</i>
<i>Flash Configuration Registers</i>	<i>10-7</i>
<i>Flash Programming Procedure</i>	<i>10-8</i>
<i>CCS Flash Plug-In.....</i>	<i>10-10</i>
<i>Code Security Mode.....</i>	<i>10-11</i>
<i>Lab Exercise 11.....</i>	<i>10-15</i>
Objective	10-15
Procedure.....	10-16
Open Files, Create Project File.....	10-16
Project Build Options	10-17
Add Additional Source Code Files	10-17
Modify Source Code to Speed up Flash memory	10-18
Build project.....	10-19
Verify Linker Results – The map - File	10-20
Use CCS Flash Program Tool.....	10-21
Shut down CCS & Restart eZdsp	10-22

C28x Start-up Sequences

There are 6 different options to start the C28x out of power-on. The options are hard-coded by 4 GPIO-Inputs of Port F (F4, F12, F3 and F2). The 4 pins are sampled during power-on. Depending on the status one of the following options is selected:

F4	F12	F3	F2	
1	x	x	x	: FLASH address 0x3F 7FF6 (see slide 10-2)
0	0	1	0	: H0 – SARAM address 0x3F 8000
0	0	0	1	: OTP address 0x3D 7800
0	1	x	x	: boot load from SPI
0	0	1	1	: boot load from SCI-A
0	0	0	0	: boot load from parallel GPIO – Port B

To switch from H0-SARAM mode to Flash mode we have to change F4 from 0 to 1. At the eZdsp this is done using jumper JP7 (1-2 = Flash; 2-3 = H0-SARAM). Please note that the C28x must also run in Microcomputer-Mode (JP1 = 2-3). The following slide shows the sequence that takes place if we start from Flash.



1. RESET-address is always 0x3F FFC0. This is part of TI's internal BOOT-ROM.
2. BOOT-ROM executes a jump to address 0x3F FC00 (Boot Code). Here basic initialization tasks are performed and the type of the boot sequence is selected.
3. If GPIO-F4 = =1, a jump to address 0x3F 7FF6 is performed. This is the Flash-Entry-Point. It is only a 2 word memory space and this space is not filled yet. One of our tasks to use the Flash is to add a jump instruction into this two-word-space. If we use a project based on C language we have to jump to the C-start-up function "c_int00", which is part of the runtime library "rts2800_ml.lib".

CAUTION: Do never exceed the two word memory space for this step. Addresses 0x3F 7FF8 to 0x3F 7FFF are reserved for the password area!!

4. Function "c_int00" performs initialization routines for the C-environment and global variables. For this module we will have to place this function into a specific Flash section.
5. At the very end "c_int00" branches to a function called "main", which also must be loaded into a flash section.

C28x Flash Memory Sectors

TMS320F2812 Flash Memory Map	
Address Range	Data & Program Space
0x3D 8000 – 0x3D 9FFF	Sector J ; 8K x 16
0x3D A000 – 0x3D BFFF	Sector I ; 8K x 16
0x3D C000 – 0x3D FFFF	Sector H ; 16K x 16
0x3E 0000 – 0x3E 3FFF	Sector G ; 16K x 16
0x3E 4000 – 0x3E 7FFF	Sector F ; 16K x 16
0x3E 8000 – 0x3E BFFF	Sector E ; 16K x 16
0x3E C000 – 0x3E FFFF	Sector D ; 16K x 16
0x3F 0000 – 0x3F 3FFF	Sector C ; 16K x 16
0x3F 4000 – 0x3F 5FFF	Sector B ; 8K x 16
0x3F 6000 – 0x3F 7F7F	Sector A ; (8K-128) x 16
0x3F 7F80 – 0x3F 7FF5	Program to 0x0000 when using Code Security Mode !
0x3F 7FF6 – 0x3F 7FF7	Flash Entry Point ; 2 x 16
0x3F 7FF8 – 0x3F 7FFF	Security Password ; 8 x 16

10 - 3

The 128k x 16 Flash is divided into 10 portions called "sectors". Each sector can be programmed independently from the others. Please note that the highest 128 addresses of sector A (0x3F7F80 to 0x3F 7FFF) are not available for general purpose. Lab 11 will use sections A and D.

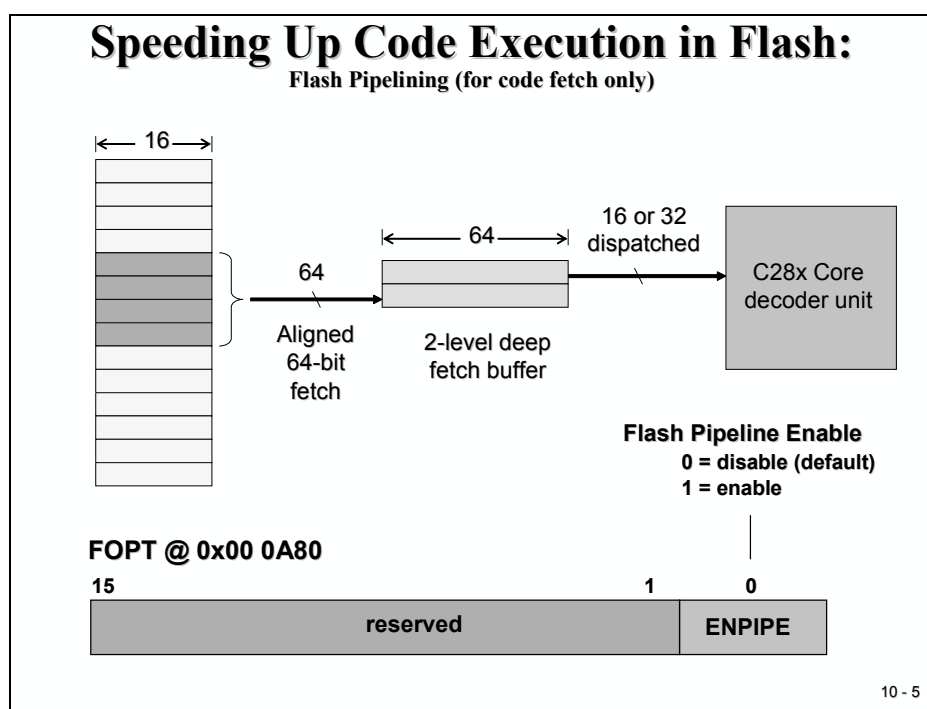
For a one-cycle instruction machine like the C28x, the 25 MHz translate into 25MIPS. This is pretty slow compared to the original system frequency of 150 MHz! Is this all we can expect from Texas Instruments? No! The hardware solution is called “pipeline”, see next slide!

Instead of reading only one 16 bit instruction out of Flash code memory TI has implemented a 64 bit access – reading up to 4 instructions in 1+5 cycles. This leads to the final estimation for the speed of internal Flash:

$$4 \text{ instructions} / 6 \text{ cycles} * 150 \text{ MHz} = 100 \text{ MHz.}$$

Using the Flash Pipeline the real **Flash speed is 100 MIPS!**

To use the Flash pipelining code fetch method we have to set bit “ENPIPE” to 1. By default after RESET, this feature is disabled.



Flash Configuration Registers

There are some more registers to control the timing and operation modes of the C28x internal Flash memory. For our lab exercise and most of the ‘real’ C28x applications it is sufficient to use the default values after RESET.

Texas Instruments provides an initialization function for the internal Flash, called “**InitFlash()**”. This function is part of the Peripheral Register Header Files, Version 1.00 that we already used in our previous labs. The source code of this function is part of file “**DSP281x_SysCtrl.c**”. All we have to do to use this function in our coming lab is to add this source code file to our project.

Other Flash Configuration Registers

Address	Name	Description
0x00 0A80	FOPT	Flash option register
0x00 0A82	FPWR	Flash power modes registers
0x00 0A83	FSTATUS	Flash status register
0x00 0A84	FSTDBYWAIT	Flash sleep to standby wait register
0x00 0A85	FACTIVEWAIT	Flash standby to active wait register
0x00 0A86	FBANKWAIT	Flash read access wait state register
0x00 0A87	FOTPWAIT	OTP read access wait state register

- ◆ **FPWR:** Save power by putting Flash/OTP to ‘Sleep’ or ‘Standby’ mode; Flash will automatically enter active mode if a Flash/OTP access is made
- ◆ **FSTATUS:** Various status bits (e.g. PWR mode)
- ◆ **FSTDBYWAIT:** Specify number of cycles to wait during wake-up from sleep to standby
- ◆ **FACTIVEWAIT:** Specify number of cycles to wait during wake-up from standby to active

Defaults for these registers are often sufficient – See “*TMS320F28x DSP System Control and Interrupts Reference Guide*,” *SPRU078*, for more information.

10 - 6

Flash Programming Procedure

The procedure to load a portion of code into the Flash is not as simple as loading a program into the internal RAM. Recall that Flash is non-volatile memory. Flash is based on a floating gate technology. To store a binary 1 or 0 this gate must load / unload electrons. Floating gate means this is an isolated gate with no electrical connections. Two effects are used to force electrons into this gate: 'Hot electron injection' or 'electron tunnelling' done by a charge pump on board of the C28x.

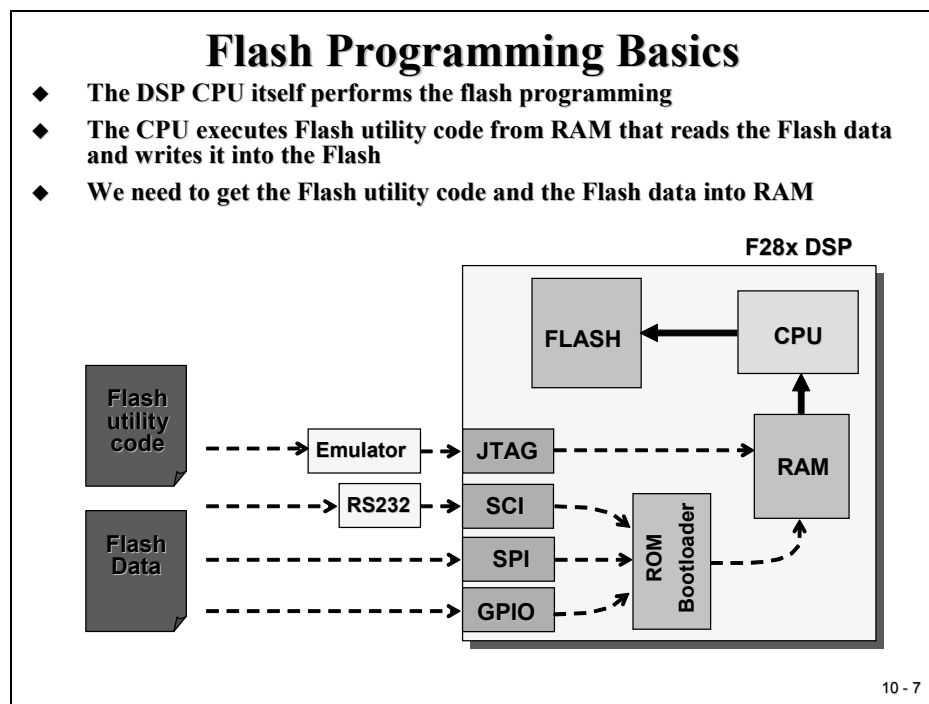
How do we get the code into the internal Flash?

The C28x itself will take care of the Flash programming procedure. Texas Instruments provides the code to execute the sequence of actions. The Flash Utility code can be applied in two basic options:

1. Code Composer Studio Plug-in Tool
➔ Tools ➔ F28xx On Chip Flash Programmer
2. Download both the Flash Utility code and the Flash Data via one of the 3 boot load options SCI-A, SPI or GPIO-B.

For our lab we will use the CCS-Tool.

Please note that the Flash Utility code must be executed from a SARAM portion of the C28x.



The steps “Erase” and “Program” to program the Flash are mandatory; “Verify” is an option but is highly recommended.

Flash Programming Basics

◆ Sequence of steps for Flash programming:

Algorithm	Function
1. Erase	- Set all bits to zero, then to one
2. Program	- Program selected bits with zero
3. Verify	- Verify flash contents

- ◆ **Minimum Erase size is a sector**
- ◆ **Minimum Program size is a bit!**
- ◆ **Important not to lose power during erase step: If CSM passwords happen to be all zeros, the CSM will be permanently locked!**
- ◆ **Chance of this happening is quite small! (Erase step is performed sector by sector)**

10 - 8

Flash Programming Utilities

- ◆ **Code Composer Studio Plug-in (uses JTAG) ***
- ◆ **Serial Flash loader from TI (uses SCI boot) ***
- ◆ **Gang Programmers (use GPIO boot)**
 - ◆ BP Micro programmer
 - ◆ Data I/O programmer
- ◆ **Build your own custom utility**
 - ◆ Use a different ROM bootloader method than SCI
 - ◆ Embed flash programming into your application
 - ◆ Flash API algorithms provided by TI

* Available from TI web at www.ti.com

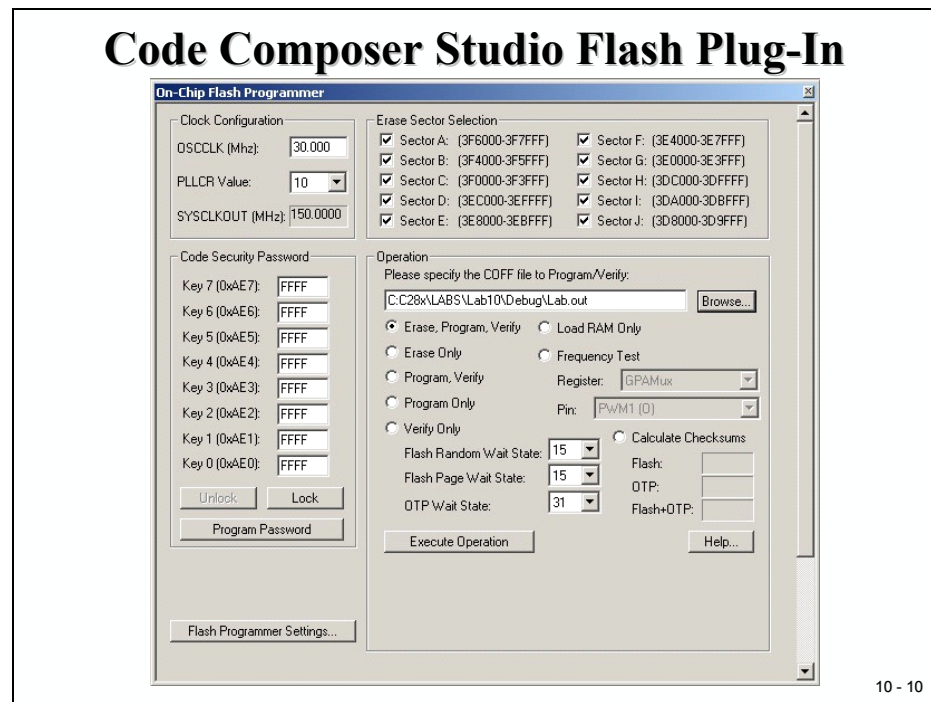
10 - 9

CCS Flash Plug-In

The Code Composer Studio Flash Plug-in is called by:

➔ Tools ➔ F28xx On Chip Flash Programmer

and opens with the following window:



First verify that the OSCCLK is set to 30MHz and the PLLCR to 10 which gives a SYSCLKOUT frequency of 150MHz. This is equivalent to the physical set up of the eZdsp2812.

NEVER use the buttons “Program Password” or “LOCK”!

Leave all 8 entries for Key 0 to Key 7 filled with “FFFF”.

On the top of the right hand side, we can exclude some of the sectors from being erased.

The lower right side is the command window. First we have to specify the name of the projects out-file. The Plug-In extracts all the information needed to program the Flash out of this COFF-File.

Before you start the programming procedure it is highly recommended to inspect the linker map-file (*.map) in the “Debug”-Subfolder. This file covers a statistical view about the usage of the different Flash sections by your project. Verify that all sections are used as expected.

Start the programming sequence by clicking on “Execute Operation”.

Code Security Mode

Before we go into our next lab let's discuss the Code Security feature of the C28x. As mentioned earlier in this module, dedicated areas of memory are password protected. This is valid for memory L0, L1, OTP and Flash.

Code Security Module (CSM)

- ◆ **Access to the following on-chip memory is restricted:**

0x00 8000	LO SARAM (4K)
0x00 9000	L1 SARAM (4K)
0x00 A000	reserved
0x3D 7800	OTP (1K)
0x3D 7C00	reserved
0x3D 8000	FLASH (128K)
- ◆ **Data reads and writes from restricted memory are only allowed for code running from restricted memory**
- ◆ **All other data read/write accesses are blocked:**
 JTAG emulator/debugger, ROM bootloader, code running in external memory or unrestricted internal memory

10 - 11

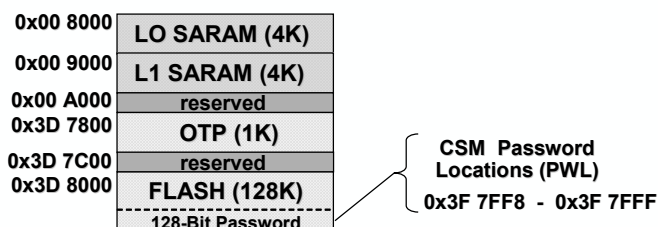
Once a password is applied, a data read or write operation from/to restricted memory locations is only allowed from code in restricted memory. All other accesses, including accesses from code running from external or unrestricted internal memories as well as JTAG access attempts are denied.

As mentioned earlier the password is located at address space 0x3F 7FF8 to 0x3F 7FFF and covers a 128-bit field. The 8 key registers (Key0 to Key7) are used to allow an access to a locked device. All you need to do is to write the correct password sequence in Key 0 -7 (address space 0x00 0AE0 – 0x00 0AE7).

The password area filled with 0xFFFF in all 8 words is equivalent to an unsecured device.

The password area filled with 0x0000 in all 8 words locks the device **FOREVER!**

CSM Password



- ◆ 128-bit user defined password is stored in Flash
- ◆ 128-bit Key Register used to lock and unlock the device
 - ♦ Mapped in memory space 0x00 0AE0 – 0x00 0AE7
 - ♦ Register “EALLOW” protected

10 - 12

CSM Registers

Key Registers – accessible by user; EALLOW protected

Address	Name	Reset Value	Description
0x00 0AE0	KEY0	0xFFFF	Low word of 128-bit Key register
0x00 0AE1	KEY1	0xFFFF	2 nd word of 128-bit Key register
0x00 0AE2	KEY2	0xFFFF	3 rd word of 128-bit Key register
0x00 0AE3	KEY3	0xFFFF	4 th word of 128-bit Key register
0x00 0AE4	KEY4	0xFFFF	5 th word of 128-bit Key register
0x00 0AE5	KEY5	0xFFFF	6 th word of 128-bit Key register
0x00 0AE6	KEY6	0xFFFF	7 th word of 128-bit Key register
0x00 0AE7	KEY7	0xFFFF	High word of 128-bit Key register
0x00 0AEF	CSMSCR	0xFFFF	CSM status and control register

PWL in memory – reserved for passwords only

Address	Name	Reset Value	Description
0x3F 7FF8	PWL0	user defined	Low word of 128-bit password
0x3F 7FF9	PWL1	user defined	2 nd word of 128-bit password
0x3F 7FFA	PWL2	user defined	3 rd word of 128-bit password
0x3F 7FFB	PWL3	user defined	4 th word of 128-bit password
0x3F 7FFC	PWL4	user defined	5 th word of 128-bit password
0x3F 7FFD	PWL5	user defined	6 th word of 128-bit password
0x3F 7FFE	PWL6	user defined	7 th word of 128-bit password
0x3F 7FFF	PWL7	user defined	High word of 128-bit password

10 - 13

Locking and Unlocking the CSM

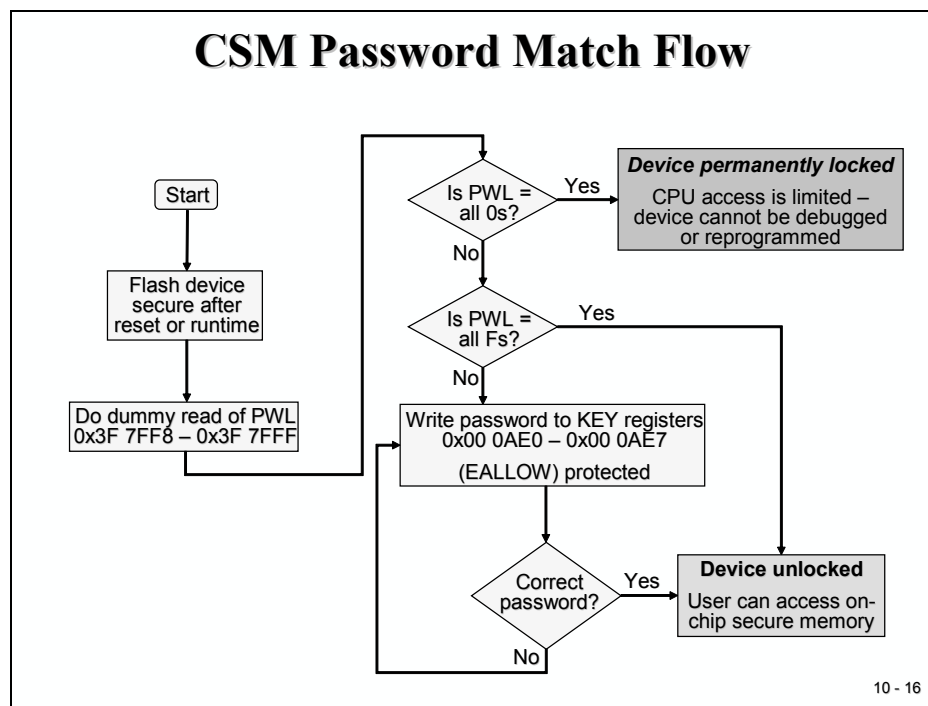
- ◆ **The CSM is locked at power-up and reset**
- ◆ **To unlock the CSM:**
 - ◆ Perform a dummy read of each password in the Flash
 - ◆ Write the correct passwords to the key registers
- ◆ **New Flash Devices (PWL are all 0xFFFF):**
 - ◆ When all passwords are 0xFFFF – only a read of the PWL is required to bring the device into unlocked mode

10 - 14

CSM Caveats

- ◆ **Never program all the PWL's as 0x0000**
 - ◆ *Doing so will permanently lock the CSM*
- ◆ **Flash addresses 0x3F7F80 to 0x3F7FF5, inclusive, must be programmed to 0x0000 to securely lock the CSM**
- ◆ **Remember that code running in unsecured RAM cannot access data in secured memory**
 - ◆ Don't link the stack to secured RAM if you have any code that runs from unsecured RAM
- ◆ **Do not embed the passwords in your code!**
 - ◆ Generally, the CSM is unlocked only for debug
 - ◆ Code Composer Studio can do the unlocking

10 - 15



CSM C-Code Examples

Unlocking the CSM:

```

volatile int *PWL = &CsmPwl.PSWD0; //Pointer to PWL register file
volatile int i, tmp;

for (i = 0; i<8; i++) tmp = *PWL++; //Dummy reads of PWL locations

asm (" EALLOW"); //KEY regs are EALLOW protected
CsmRegs.KEY0 = PASSWORD0; //Write the passwords
CsmRegs.KEY1 = PASSWORD0; //to the Key registers
CsmRegs.KEY2 = PASSWORD2;
CsmRegs.KEY3 = PASSWORD3;
CsmRegs.KEY4 = PASSWORD4;
CsmRegs.KEY5 = PASSWORD5;
CsmRegs.KEY6 = PASSWORD6;
CsmRegs.KEY7 = PASSWORD7;
asm (" EDIS");
  
```

Locking the CSM:

```

asm(" EALLOW"); //CSMSCR reg is EALLOW protected
CsmRegs.CSMSCR.bit.FORCESEC = 1; //Set FORCESEC bit
asm ("EDIS");
  
```

10 - 17

Lab Exercise 11

Lab 11: Load an application into Flash

- Use Solution for Lab4 to begin with
- Modify the project to use internal Flash for code
- Add “DSP281x_CodeStartBranch.asm” to branch from Flash entry point (0x3F 7FF6) to C - library function “_c_int00”
- Add TI - code to set up the speed of Flash
- Add a function to move the speed-up code from Flash to SARAM Adjust Linker Command File
- Use CCS plug-in tool to perform the Flash download
- Disconnect emulator, set eZdsp into MC-mode (JP1) and re-power the board!
- Code should be executed out of Flash
- For details see procedure in textbook!

10 - 18

Objective

The objective of this laboratory exercise is to practice with the C28x internal Flash Memory. Let us assume your task is to prepare one of your previous laboratory solutions to run as a stand alone solution direct from Flash memory after power on of the C28x. You can select any of your existing solutions but to keep it easier for your supervisor to assist you during the debug phase let us take the ‘knight rider’ (Lab 4) as the starting point.

What do we have to modify?

In Lab 4 the code was loaded by CCS via the JTAG-Emulator into H0-SARAM after a successful build operation. The linker command file “F2812_EzDSP_RAM_Ink.cmd” took care of the correct connection of the code sections to physical memory addresses of H0SARAM. Obviously, we will have to modify this part. Instead of editing the command file we will use another one (“F2812.cmd”), also provided by TI’s header file package.

Furthermore we will have to fill in the Flash entry point address with a connection to the C environment start function (“c_int00”). Out of RESET the Flash memory itself operates with the maximum number of wait states – our code should reduce this wait states to gain the highest possible speed for Flash operations. Unfortunately we can’t call this speed up function when it is still located in Flash – we will have to copy this function temporarily into any code SARAM before we can call it.

Finally we will use Code Composer Studio's Flash Programming plug in tool to load our code into Flash.

Please recall the explanations about the Code Security Module in this lesson, be aware of the password feature all the time in this lab session and do NOT program the password area!

A couple of things to take into account in this lab session, as usual let us use a procedure to prepare the project.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab11.pjt** in E:\C281x\Labs.
2. Open the file Lab4.c from E:\C281x\Labs\Lab4 and save it as Lab11.c in E:\C281x\Labs\Lab11.
3. Add the source code file to your project:
 - **Lab11.c**
4. From *C:\tidcs\c28\dsp281x\v100\DSP281x_headers\source* add:

- **DSP281x_GlobalVariableDefs.c**

From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**
- **DSP281x_CpuTimers.c**

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

- **F2812_Headers_nonBIOS.cmd**

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

- **F2812.cmd**

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

- **rts2800_ml.lib**

Project Build Options

5. 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>**.

Add Additional Source Code Files

7. To add the machine code for the Flash entry point at address 0x3F 7FF6 we have to add an assembly instruction “LB _c_int00” and to link this instruction exactly to the given physical address. Instead of writing our own assembly code we can make use of another of TI’s predefined functions (“code_start”) which is part of the source code file “DSP218x_CodeStartBranch.asm”.

From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add:

- **DSP281x_CodeStartBranch.asm**

If you open the file “F2812.cmd” you will see that label “code_start” is linked to “BEGIN” which is defined at address 0x3F 7FF6 in code memory page 0.

8. The function to speed up the internal Flash (“InitFlash()”) is also available from TI as part of the source code file “DSP281x_SysCtrl.c”.

From *C:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add:

- **DSP281x_SysCtrl.c**

Modify Source Code to Speed up Flash memory

9. Open Lab11.c to edit.

In main, after the function call “InitSystem()” we have to add code to speed up the Flash memory.

This will be done by function “InitFlash”. But, as already mentioned, this code must run out of SARAM. When we finally run the program out of Flash and the C28x reaches this line all code is still located in Flash. That means, before we can call “InitFlash” the C28x has to copy it into SARAM. Standard ANSI-C provides a memory copy function “memcpy(*dest,*source, number)” for this purpose.

What do we use for “dest”, “source” and “number”?

Again, the solution can be found in file “DSP281x_SysCtrl.c”. Open it and look at the beginning of this file. You will find a “#pragma CODE_SECTION” – line that defines a dedicated code section “ramfuncs” and connects the function “InitFlash()” to it. Symbol “ramfuncs” is used in file “F2812.cmd” to connect it to physical memory “FLASHD” as load-address and to memory “RAML0” as execution address. The task of the linker command file “F2812.cmd” is it to provide the physical addresses to the rest of the project. The symbols “LOAD_START”, “LOAD_END” and “RUN_START” are used to define these addresses symbolically as “_RamfuncsLoadStart”, “_RamfuncsLoadEnd” and “_RamfuncsRunStart”.

Add the following line to your code:

```
memcpy(&RamfuncsRunStart, &RamfuncsLoadStart,  
      &RamfuncsLoadEnd - &RamfuncsLoadStart);
```

Add a call of function “InitFlash()”, now available in RAML0:

```
InitFlash();
```

At the beginning of Lab11.c declare the symbols used as parameters for memcpy as externals:

```
extern unsigned int RamfuncsLoadStart;
```

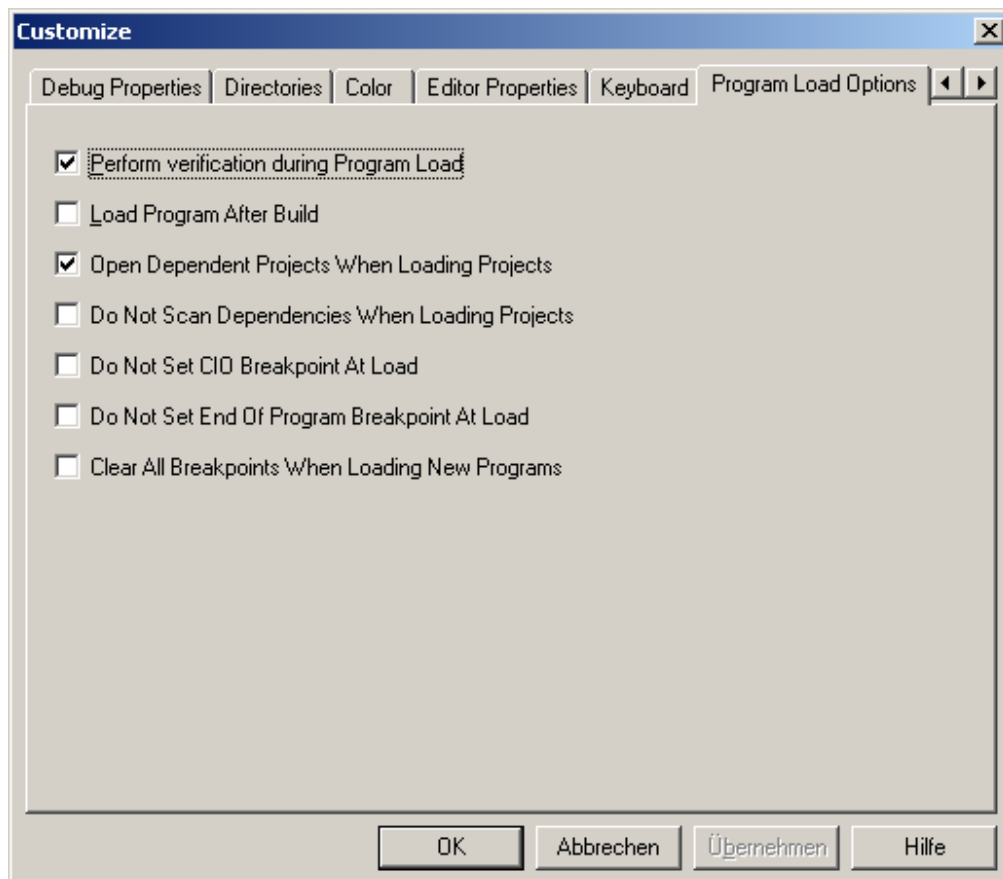
```
extern unsigned int RamfuncsLoadEnd;
```

```
extern unsigned int RamfuncsRunStart;
```

Build project

10. Our code will be compiled to be located in Flash. In our previous lab exercises you probably used the option to download the program after successful build” in CCS → Option → Customize → Load Program After Build. We can’t use this feature for this exercise because the destination is Flash.

Please make sure, that this option is disabled now!



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

Project → Build

If build was successful you’ll get:

Build Complete,

0 Errors, 0 Warnings, 0 Remarks.

Verify Linker Results – The map - File

12. Before we actually start the Flash programming it is always good practice to verify the used sections of the project. This is done by inspecting the linker output file 'lab11.map'
13. Open file 'lab11.map' out of subdirectory ..\Debug

In 'MEMORY CONFIGURATION' column 'used' you will find the amount of physical memory that is used by your project.

Verify that only the following four parts of PAGE 0 are used:

RAML0	00008000	00001000	00000016	RWIX
FLASHD	003ec000	00004000	00000016	RWIX
FLASHA	003f6000	00001f80	0000056b	RWIX
BEGIN	003f7ff6	00000002	00000002	RWIX

The number of addresses used in FLASHA might be different in your lab session. Depending on how efficient your code was programmed by you, you will end up with more or less words in this section.

Verify also that in PAGE1 the memory RAMH0 is not used.

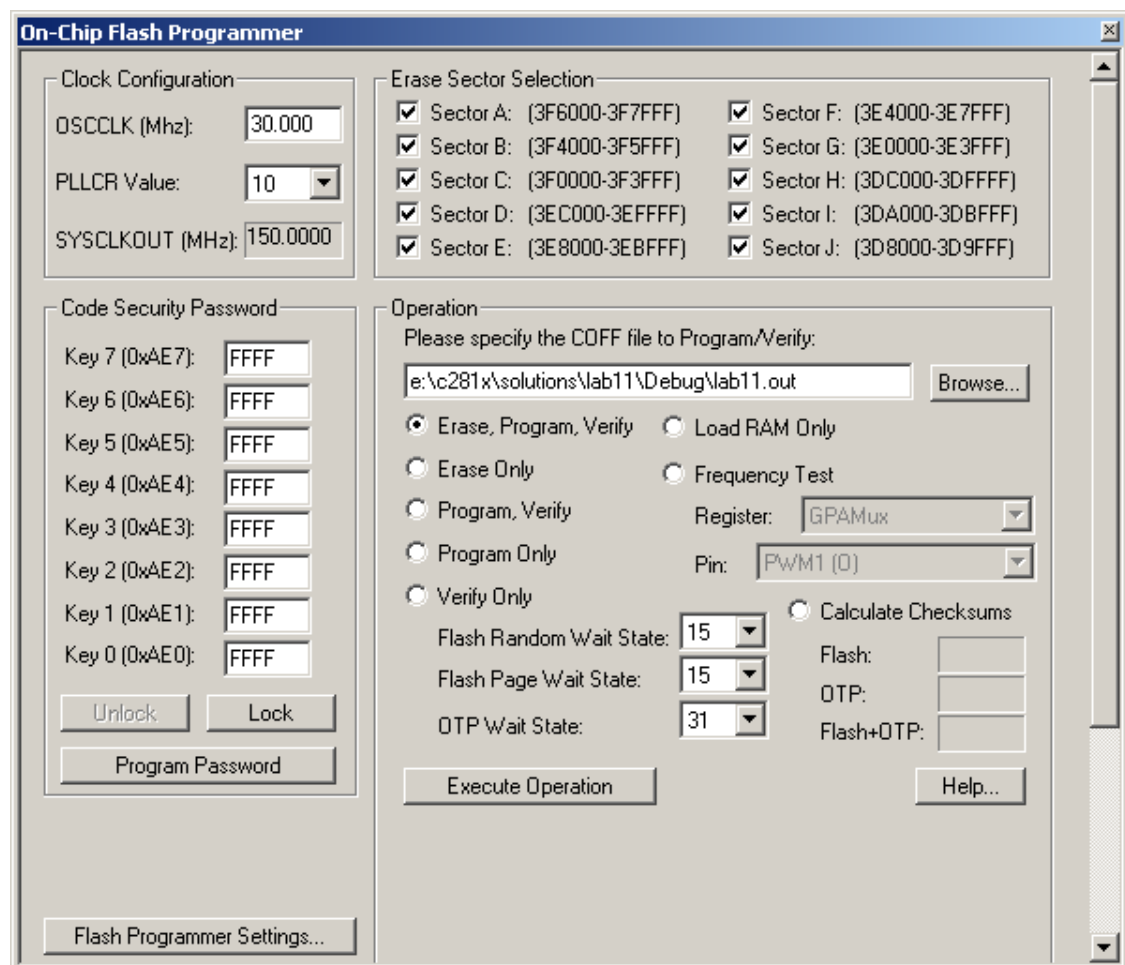
In the SECTION ALLOCATION MAP you can see how the different portions of our projects code files are distributed into the physical memory sections. For example, the .text-entry shows all the objects that were concatenated into FLASHA.

Entry 'codestart' connects the object 'CodeStartBranch.obj' to physical address 0x3F7FF6 and occupies two words.

Use CCS Flash Program Tool

13. Next step is to program the machine code into the internal Flash. As mentioned in this lesson there are different ways to accomplish this step. The easiest way is to use the Code Composer Studio plug-in tool:

Tools → F28xx On-Chip Flash Programmer



- Please make sure that OSCCLK is set to 30MHz and PLLCR to 10.
- Do **NOT** change the Key 7 to Key 0 entries! They should all show “FFFF”!
- Select the current COFF – out file: ..\Debug\lab11.out
- Select the operation “Erase, Program, Verify
- Hit button “Execute Operation”

If everything went as expected you should get these status messages:

```
**** Begin Erase/Program/Verify Operation. ***  
Erase/Program/Verify Operation in progress...  
Erase operation in progress...  
Erase operation was successful.  
Program operation in progress...  
Program operation was successful.  
Verify operation in progress...  
Verify operation successful.  
Erase/Program/Verify Operation succeeded  
**** End Erase/Program/Verify Operation. ***
```

Now the code is stored in Flash!

Shut down CCS & Restart eZdsp

14. Close your CCS session.
15. Disconnect the eZdsp from power.
16. Verify that eZdsp Jumper JP1 is in position 2-3 (Microcomputer Mode).
17. Verify that eZdsp Jumper JP7 is in position 1-2 (Boot from Flash)
18. Reconnect eZdsp to power.

Your code should be executed immediately out of Flash, showing the LED-sequence at GPIO-port B.

C28x IQ – Math Library

Introduction

One of the most important estimations in embedded control is the calculation of computing time for a given task. Since embedded control has to cope with these tasks in a given and fixed amount of time, we call this ‘Real Time Computing’. And, as you know, time goes very quickly.

Therefore, one of the characteristics of a processor is the ability to do mathematical calculations in an optimal and efficient way. In recent years, the size of mathematical algorithms that have been implemented in embedded control units has increased dramatically. Just take the number of pages for the requirement specification of an electronic control unit for a passenger car:

- 1990: 50 pages,
- 2000: 3100 pages (Source: Volkswagen AG)

So, how does a processor operate with all these mathematical calculations? And, how does the processor access process data?

You know that the ‘native’ numbering scheme for a digital controller uses binary numbers. Unfortunately, all process values are either in the format of integer or real numbers. Depending on how a processor deals with these numbers in its translation into binary numbers, we distinguish between two basic types of processor core:

- Floating-point Processors
- Fixed-point Processors

This chapter will start with a brief comparison between the two types of processor.

Because the C28x belongs to the fixed-point type we will focus on this type more in detail. After a brief discussion about binary numbers we will have a look into the different options to use the fixed-point unit of the C28x. It can perform various types of mathematical operations in a very efficient way, using only a few machine clock cycles.

The secret behind this approach is called “IQ-Math”. In case of the C28x Texas Instruments provides a library that uses the internal hardware of the C28x in the most efficient way to operate with 32bit fixed-point numbers. Taking into account that all process data usually do not exceed a resolution of 16 bits, the library gives enough headroom for advanced numerical calculations. The latest version of Texas Instruments “IQ-Math” - Library can be found with literature number “SPRC087” at www.ti.com.

Module Topics

C28x IQ – Math Library.....	11-1
<i>Introduction</i>	<i>11-1</i>
<i>Module Topics.....</i>	<i>11-2</i>
<i>Floating-point, Integer and Fixed-point</i>	<i>11-3</i>
<i>IEEE 754 Floating-point Format.....</i>	<i>11-4</i>
<i>Integer Number Basics.....</i>	<i>11-7</i>
Two's Complement representation.....	11-7
Binary Multiplication	11-7
<i>Binary Fractions</i>	<i>11-9</i>
Multiplying Binary Fractions	11-9
<i>The “IQ” – Format.....</i>	<i>11-11</i>
<i>Sign Extension.....</i>	<i>11-14</i>
<i>Correcting the redundant sign bit.....</i>	<i>11-15</i>
<i>IQ – Math – Library.....</i>	<i>11-17</i>
Standard ANSI – C 16-Bit Mathematics	11-18
Standard ANSI – C 32-Bit Mathematics	11-19
32-Bit IQ – Math Approach.....	11-20
IQ – Math Library Functions.....	11-24
IQ- Math Application : Field Orientated Control	11-25

Floating-point, Integer and Fixed-point

All processors can be divided into two groups, “floating-point” and “fixed-point”. The core of a floating-point processor is a hardware unit that supports floating-point operations according to international standard IEEE 754. Intel’s x86 – family of Pentium processors is a typical example of this type. Floating-point processors are very efficient when operating with floating-point data and allow a high dynamic range for numerical calculations. They are not so efficient when it comes to control tasks (bit manipulations, input/output control, interrupt response) – and, they are expensive.

Floating Point, Integer and Fixed Point

- ◆ **Two basic categories of processors:**
 - ◆ **Floating Point**
 - ◆ **Integer/Fixed Point**
- ◆ **What is the difference?**
- ◆ **What are advantages / disadvantages ?**
- ◆ **Real – Time Control : Fixed Point !**
- ◆ **Discuss fixed-point math development limitations**
- ◆ **Compare/contrast floating-point and IQ representation**
- ◆ **Texas Instruments IQ-Math approach**

11 - 2

Fixed-point Processors are based on internal hardware that supports operations with integer data. The arithmetic logic unit and, in case of digital signal processors, the hardware multiply unit, expect data to be in one of the fixed-point types. There are limitations in the dynamic range of a fixed-point processor, but they are inexpensive.

What happens, when we write a program for a fixed-point processor in C and we declare a floating-point data type ‘float’ or ‘double’? A number of library functions support this data type on a fixed-point machine. These standard ANSI-C functions consume a lot of computing power. Recalling the time constraints in a real time project, we just can’t afford to use these data types in most of embedded control applications.

The solution, in case of the C28x is “IQ-Math”. The IQ-Math Library is a collection of highly optimised and high precision mathematical functions used to seamlessly port floating-point algorithms into fixed-point code. In addition, by incorporating the ready-to-use high precision functions, the IQ-Math library can shorten significantly an embedded control development time.

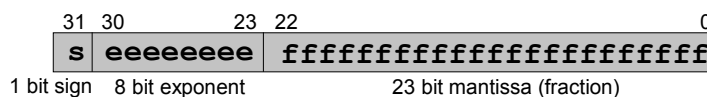
Processor Types

- ◆ **Floating Point Processors**
 - ◆ Internal Hardware Unit to support Floating Point Operations
 - ◆ Examples : Intel's Pentium Series , Texas Instruments C 6000 DSP
 - ◆ High dynamic range for numeric calculation
 - ◆ Rather expensive
- ◆ **Integer / Fixed – Point Processors**
 - ◆ Fixed Point Arithmetic Unit
 - ◆ Almost all embedded controllers are fixed point machines
 - ◆ Examples: all microcontroller families, e.g. Motorola HC68x, Infineon C166, Texas Instruments TMS430, TMS320C5000, C2000
 - ◆ Lowest price per MIPS

11 - 3

IEEE 754 Floating-point Format

IEEE Standard 754 Single Precision Floating-Point



Case 1: if $e = 255$ and $f \neq 0$, then $v = \text{NaN}$

Case 2: if $e = 255$ and $f = 0$, then $v = [(-1)^s] * \text{infinity}$

Case 3: if $0 < e < 255$, then $v = [(-1)^s] * [2^{(e-127)}] * (1.f)$

Case 4: if $e = 0$ and $f \neq 0$, then $v = [(-1)^s] * [2^{(-126)}] * (0.f)$

Case 5: if $e = 0$ and $f = 0$, then $v = [(-1)^s] * 0$

Advantage \Rightarrow Exponent gives large dynamic range
Disadvantage \Rightarrow Precision of a number depends on its exponent

11 - 4

Floating-point definitions:

- **Sign Bit (S):**

- Negative: bit 31 = 1 / Positive: Bit 31 = 0

- **Mantissa (M):**

- $$M = 1 + m_1 \cdot 2^{-1} + m_2 \cdot 2^{-2} + \dots = 1 + \sum_{i=1}^{23} m_i \cdot 2^{-i}$$

- Mantissa is tailored to $m_0 = 1$; m_0 will not be stored in memory!

$$1 \leq M < 2$$

- **Exponent (E):**

- 8 Bit signed exponent, stored with offset “+127”

- **Summary:**

- $$Z = (-1)^S \cdot M \cdot 2^{E-OFFSET}$$

Example1:

0x 3FE0 0000 = 0011 1111 1110 0000 0000 0000 0000 0000 B

$$S = 0$$

$$E = 0111\ 1111 = 127$$

$$M = (1).11000 = 1 + 0.5 + 0.25 = 1.75$$

$$Z = (-1)^0 * 1,75 * 2^{127-127} = 1.75$$

Example2:

0x BFB0 0000 = 1011 1111 1011 0000 0000 0000 0000 0000 B

$$S = 1$$

$$E = 0111\ 1111 = 127$$

$$M = (1).011 = 1 + 0.25 + 0.125 = 1.375$$

$$Z = (-1)^1 * 1.375 * 2^{127-127} = -1.375$$

Example3:

Z = -2.5 S = 1

$$2.5 = 1.25 * 2^1$$

$$1 = E - \text{OFFSET}$$

$$E = 128$$

$$M = 1.25 = (1).01 = 1 + 0.25$$

$$\text{Binary : } 1100\ 0000\ 0010\ 0000\ 0000\ 0000\ 0000\ 0000\ \text{B} = 0x\ \text{C020}\ 0000$$

Floating - Point does not solve everything!

Example: x = 10.0 (0x41200000)
 + y = 0.000000238 (0x347F8CF1)

$$z = 10.000000238$$

WRONG!

You cannot represent 10.000000238 with
single-precision floating point

0x412000000 = 10.000000000
 10.000000238 \leftarrow can't represent!
0x412000001 = 10.000000950

So z gets rounded down to 10.000000000

11 - 5

Integer Number Basics

Two's Complement representation

The next slides summarize the basics of the two's complement representation of signed integer numbers.

Integer Numbering System Basics

◆ Binary Numbers

$$0110_2 = (0*8)+(1*4)+(1*2)+(0*1) = 6_{10}$$

$$11110_2 = (1*16)+(1*8)+(1*4)+(1*2)+(0*1) = 30_{10}$$

◆ Two's Complement Numbers

$$0110_2 = (0*-8)+(1*4)+(1*2)+(0*1) = 6_{10}$$

$$11110_2 = (1*-16)+(1*8)+(1*4)+(1*2)+(0*1) = -2_{10}$$

11 - 6

Binary Multiplication

Now consider the process of multiplying two two's complement values, which is one of the most often used operations in digital control. As with “long hand” decimal multiplication, we can perform binary multiplication one “place” at a time, and sum the results together at the end to obtain the total product.

Note: The method shown at the following slide is not the method the C28x uses to multiply numbers — it is merely a way of observing how binary numbers work in arithmetic processes.

The C28x uses 32-bit operands and an internal 64-bit product register. For the sake of clarity, consider the example below where we shall investigate the use of 4-bit values and an 8-bit accumulation:

Four-Bit Integer Multiplication

$ \begin{array}{r} 0100 \\ \times 1101 \\ \hline 00000100 \\ 00000000 \\ 00010000 \\ 11100000 \\ \hline 11110100 \end{array} $	$ \begin{array}{r} 4 \\ \times -3 \\ \hline \\ \\ \\ \hline -12 \end{array} $
--------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------

Accumulator 11110100

Data Memory ?

Is there another (superior) numbering system? 11 - 7

In this example, consider the following:

- 4 multiplied by (-3) gives (-12) in decimal
- Size of the product is twice as long as the input values (4 bit * 4 bit = 8 bit)
- If this product is to be used in a next loop of a calculation, how can the result be stored back to memory in the same length as the inputs?
 - Store back upper 4 Bit of Accumulator? → -1
 - Store back lower 4 Bit of Accumulator? → +4
 - Store back all 8 Bit of Accumulator? → overflow of length
- Scaling of intermediate results is needed!

From this analysis, it is clear that integers do not behave well when multiplied.

Might some other type of number system behave better? Is there a number system where the results of a multiplication have bounds?

Binary Fractions

In order to represent both positive and negative values, the two's complement process will again be used. However, in the case of fractions, we will not set the LSB to 1 (as was the case for integers). When we consider that the range of fractions is from -1 to $\sim+1$, and that the only bit which conveys negative information is the MSB, it seems that the MSB must be the “negative ones position”. Since the binary representation is based on powers of two, it follows that the next bit would be the “one-half” position, and that each following bit would have half the magnitude again.

Yes: Binary Fractions

1	.	0	1	1
-1			1/2	1/4

$$= -1 + 1/4 + 1/8 = -5/8$$

Fractions have the nice property that
fraction x fraction = fraction

11 - 9

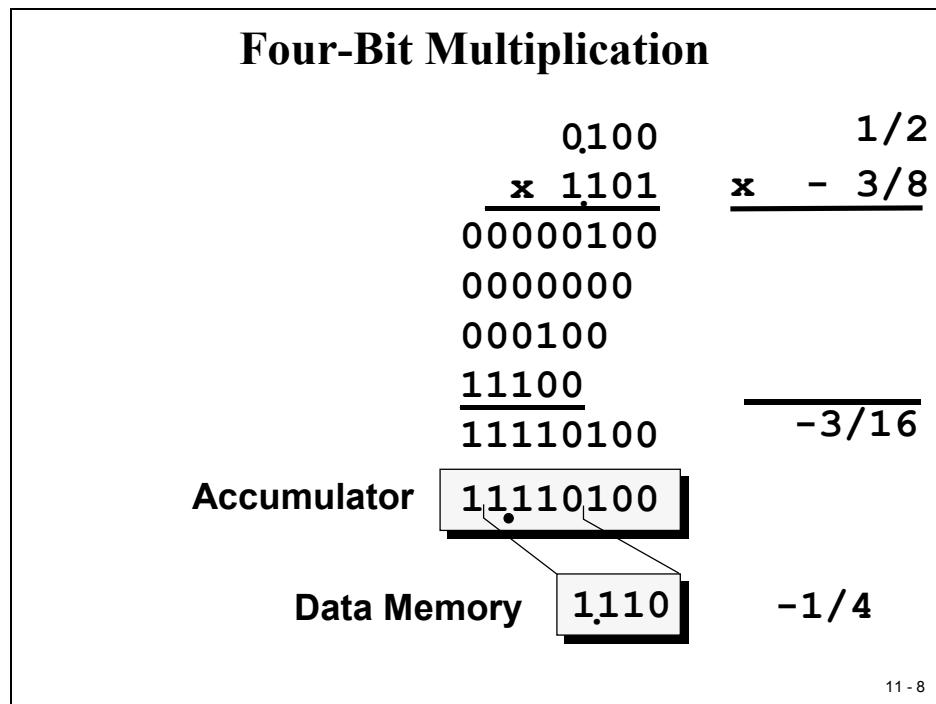
Multiplying Binary Fractions

When the C28x performs multiplication, the process is identical for all operands, integers or fractions. Therefore, the user must determine how to interpret the results. As before, consider the 4-bit multiply example:

The input numbers are now split into two parts – integer part (I) and fractional part (Q – quotient). These type of fixed-point numbers are often called “IQ”-numbers, or for simplicity just Q-numbers.

The example above shows 2 input numbers in I1Q3 - Format. When multiplied the length of the result will add both I and Q portions (see next slide):

$$\text{I1Q3} * \text{I1Q3} = \text{I2Q6}$$



If we store back the intermediate product with the four bits around the binary point we keep the data format (I1Q3) in the same shape as the input values. No need to re-scale any intermediate results!

Advantage: With Binary Fractions we will gain a lot of speed in closed loop calculations.

Disadvantage: The result might not be the exact one. As you can see from the slide above we will end up with $(-4/16)$ stored back to Data Memory. Bits 2^{-4} to 2^{-6} are truncated. The correct result would have been $(-3/16)$.

Recall that the 4-bit input operand multiplication operation is not the real size for the C28x, which operates on 32-bit input values. In this case, the truncation will affect bits 2^{-32} to 2^{-64} . Given the real size of process data with let's say 12-bit ADC measurement values, there is plenty of room left for truncation.

In most cases we will truncate noise only. However, in some feedback applications like IIR-Filters the small errors can add and lead to a given degree of instability. It is designer's responsibility to recognize this potential source of failure when using binary fractions.

The “IQ” – Format

So far, we have discussed only the option to use fractional numbers with the binary point at the MSB-side of the number. In general, we can place this point anywhere in the binary representation. This gives us the opportunity to trade off dynamic range against resolution:

Fractional Representation



$$-2^I + 2^{I-1} + \dots + 2^1 + 2^0 \bullet 2^{-1} + 2^{-2} + \dots + 2^{-Q}$$

“IQ” – Format

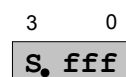
“I” \Rightarrow INTEGER – Fraction
 “Q” \Rightarrow QUOTIENT – Fraction

Advantage \Rightarrow Precision same for all numbers in an IQ format
Disadvantage \Rightarrow Limited dynamic range compared to floating point

11 - 10

IQ - Examples

I1Q3 – Format:



Most negative decimal number: -1.0	= 1.000 B
Most positive decimal number: + 0.875	= 0.111 B
Smallest negative decimal number: $-1 \cdot 2^{-3}$ (0.125)	= 1.111 B
Smallest positive decimal number: 2^{-3} (0.125)	= 0.001 B

Range: -1.0 0.875 ($\approx + 1.0$)
Resolution: 2^{-3}

11 - 11

IQ - Examples

I3Q1 – Format:

$\begin{matrix} 3 & 0 \\ \text{SII} \cdot \text{f} \end{matrix}$

Most negative decimal number:	-4.0	= 100.0 B
Most positive decimal number:	+ 3.5	= 011.1 B
Smallest negative decimal number:	$-1 * 2^{-1}$	= 111.1 B
Smallest positive decimal number:	2^{-1}	= 000.1 B

Range:	-4.0 +3.5 ($\approx + 4.0$)
Resolution:	2^{-1}

11 - 12

IQ - Examples

I1Q31 – Format:

$\begin{matrix} 31 & & & & & & & & & & & & & & & & & & & & & & & & & & & & & & & & & 0 \\ \text{S} \cdot \text{fff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} & \text{ffff} \end{matrix}$

Most negative decimal number:	-1.0	1.000 0000 0000 0000 0000 0000 0000 B
Most positive decimal number:	$\approx + 1.0$	0.111 1111 1111 1111 1111 1111 1111 B
Smallest negative decimal number:	$-1*2^{-31}$	1.111 1111 1111 1111 1111 1111 1111 B
Smallest positive decimal number:	2^{-31}	0.000 0000 0000 0000 0000 0000 0001 B

Range:	-1.0 (+1.0)
Resolution:	2^{-31}

11 - 13

IQ - Examples

I8Q24 – Format:

³¹ 0
S III IIII.ffff ffff ffff ffff ffff

Most negative decimal number: -128
 1000 0000. 0000 0000 0000 0000 0000 0000 B

Most positive decimal number: $\approx +128$
 0111 1111. 1111 1111 1111 1111 1111 1111 B

Smallest negative decimal number: $-1 \cdot 2^{-24}$
 1111 1111. 1111 1111 1111 1111 1111 1111 B

Smallest positive decimal number: 2^{-24}
 0000 0000. 0000 0000 0000 0000 0000 0001 B

Range:	-128 (+128)
Resolution:	2^{-24}

11 - 14

And to come back to the failing floating-point example from the beginning of this module; IQ-Math can do much better:

IQ-Math can do better!

I8Q24 Example:	x = 10.0	(0x0A000000)
	+ y = 0.000000238	(0x00000004)
	z = 10.000000238	
		(0x0A000004)

Exact Result (this example)

11 - 15

Sign Extension

When working with signed numbers it is important to keep the sign information when expanding an operand in its binary representation, for example from 4-bit to 8-bit, as shown in the next slide.

The C28x can operate on either unsigned binary or two's complement operands. The so-called "Sign Extension Mode (SXM)" identifies whether or not the sign extension process is used automatically when a number is processed internally. It is a good programming practice to always select the desired operating mode of SXM at the beginning of a subroutine or a module.

What is Sign Extension?

- ◆ When moving a value from a narrowed width location to a wider width location, the sign bit is extended to fill the width of the destination
- ◆ Sign extension applies to signed numbers only
- ◆ It keeps negative numbers negative!
- ◆ Sign extension controlled by SXM bit in ST0 register; When SXM = 1, sign extension happens automatically

4 bit Example: Load a memory value into the ACC

memory 1101 = $-2^3 + 2^2 + 2^0 = -3$

↓ Load and sign extend

←

ACC 1111 1101 = $-2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^0$

= $-128 + 64 + 32 + 16 + 8 + 4 + 1$

= -3

11 - 16

The SXM-Bit is part of ST0, one of the C28x status- and control registers. It can be accessed in assembly language only. To set or to clear it out of a C environment one can use the inline assembly function:

```
asm("      SETC SXM");

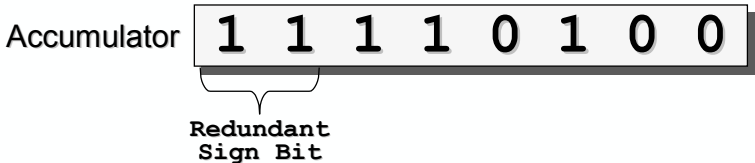
asm("      CLRC SXM");
```

Correcting the redundant sign bit

As we have already seen, when we multiply two I1Q3-numbers we end up with an I2Q6-result. Or, by multiplying two I1Q15 – numbers we end up with an I2Q30 – result. The second sign bit is always redundant. To adjust the result back to the format of the inputs we would have to apply shift operations, as shown in the next slide in a C environment. The shift operator (\gg 15) will shift the intermediate result 15 times before it is typecast back to the data format of result variable z.

Texas Instruments “IQ-Math”-library, which will be explained in the rest of this module, takes care of this shift procedure internally. Again, we gain speed by using “IQ-Math”.

Correcting Redundant Sign Bit



Accumulator **1 1 1 1 0 1 0 0**

Redundant Sign Bit

◆ **Correcting Redundant Sign Bit**

- ◆ **IQmath: automatically handled (*next topic*)**
- ◆ **Q math in “C”, shift in software:**

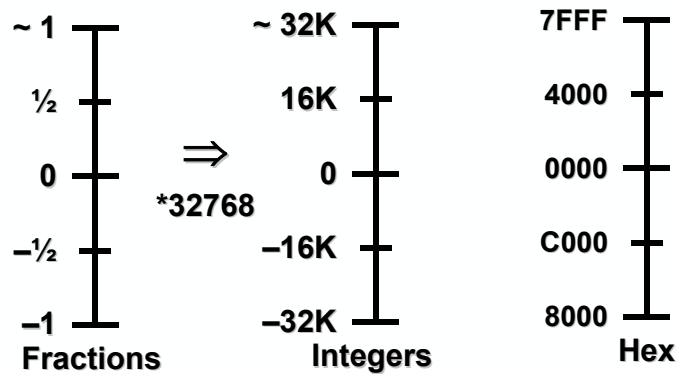
```
int x,y,z;
z = ((long)x * (long)y) >> 15;
```

11 - 17

How do we code fractions in an ANSI-C environment? We do not have a dedicated data type, called ‘fractional’. There is a new ANSI- standard under development, called “embedded C”, which will eventually use this type.

For now we can use the following trick, see next slide:

How is a fraction coded?



- ◆ Example: represent the fraction number 0.707

```
void main(void) {
    int coef = 32768 * 707 / 1000;
}
```

11 - 18

Fractional vs. Integer

- ◆ Range
 - ◆ Integers have a maximum range determined by the number of bits
 - ◆ Fractions have a maximum range of ± 1
- ◆ Precision
 - ◆ Integers have a maximum precision of 1
 - ◆ Fractional precision is determined by the number of bits

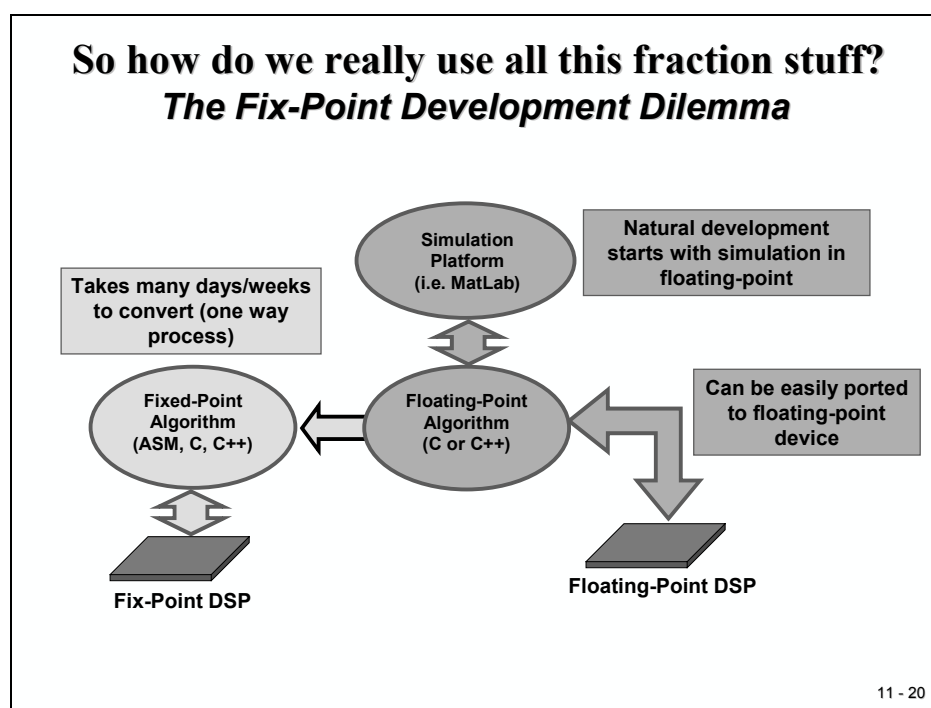
11 - 19

IQ – Math – Library

Implementing complex digital control algorithms on a Digital Signal Processor (DSP), or any other DSP capable processor, typically we come across the following issues:

- Algorithms are typically developed using floating-point math's
- Floating-point devices are more expensive than fixed-point devices
- Converting floating-point algorithms to a fixed-point device is very time consuming
- Conversion process is one way and therefore backward simulation is not always possible

The diagram below illustrates a typical development scenario in use today:

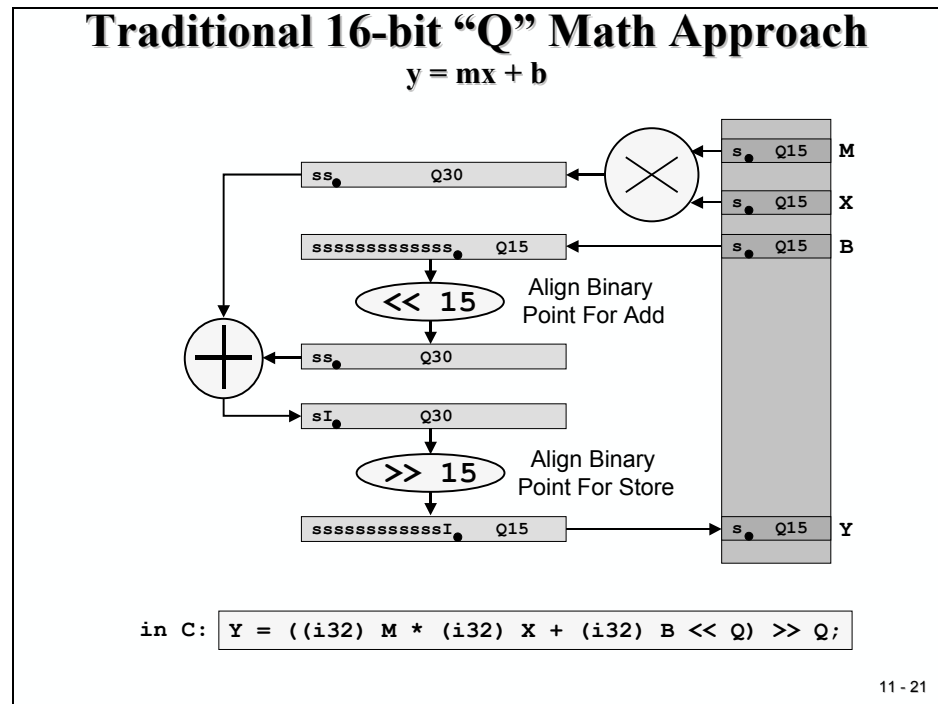


The design may initially start with a simulation (i.e. MatLab) of a control algorithm, which typically would be written in floating-point math (C or C++). This algorithm can be easily ported to a floating-point device. However, because of the commercial reality of cost constraints, most likely a 16-bit or 32-bit fixed-point device would be used in many target systems.

The effort and skill involved in converting a floating-point algorithm to function using a 16-bit or 32-bit fixed-point device is quite significant. A great deal of time (many days or weeks) would be needed for reformatting, scaling and coding the problem. Additionally, the final implementation typically has little resemblance to the original algorithm. Debugging is not an easy task and the code is not easy to maintain or document.

Standard ANSI – C 16-Bit Mathematics

If the processor of your choice is a 16-bit fixed-point and you don't want to include a lot of library functions in your project, a typical usage of binary fractions is shown next. We assume that the task is to solve the equation $Y = MX + B$. This type of equation can be found in almost every mathematical approach for digital signal processing.



The diagram shows the transformations, which are needed to adjust the binary point in between the steps of this solution. We assume that the input numbers are in $I1Q_{15}$ -Format. After M is multiplied by X , we have an intermediate product in $I2Q_{30}$ -format. Before we can add variable B , we have to align the binary point by shifting b 15 times to the left. Of course we need to typecast B to a 32-bit long first to keep all bits of B . The sum is still in $I2Q_{30}$ -format. Before we can store back the final result into Y we have to right shift the binary point 15 times.

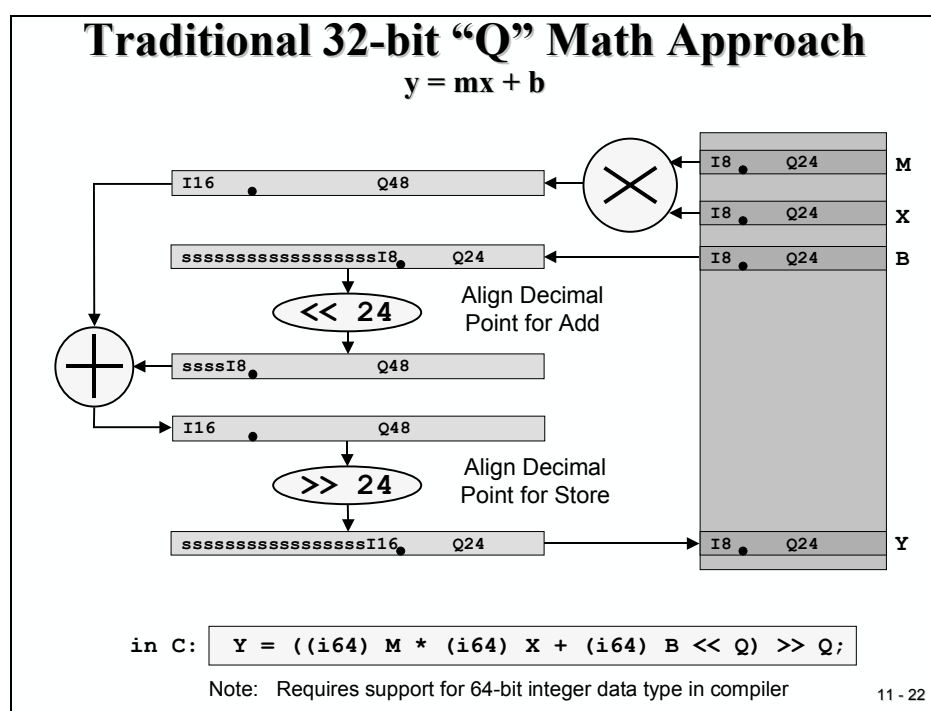
The last line of the slide shows the equivalent syntax in ANSI-C. “i32” stands for a 32-bit integer, usually called ‘long’. ‘Q’ is a global constant and gives the number of fractional bits; in our example Q is equal to 15.

The disadvantage of this Q_{15} – approach is its limitation to 16 bits. A lot of projects for digital signal processing and digital control will not be able to achieve stable behavior due to the lack of either resolution or dynamic range.

The C28x as a 32-bit processor can do better – we just have to expand the scheme to 32-bit binary fractions!

Standard ANSI – C 32-Bit Mathematics

The next diagram is an expansion of the previous scheme to 32-bit input values. Again, the task is to solve equation $Y = MX + B$. In the following example the input numbers are in an I8Q24-format.



The big problem with the translation into ANSI-C code is that we do not have a 64-bit integer data type! Although the last line of the slide looks pretty straight forward, we can't apply this line to a standard C-compiler!

What now?

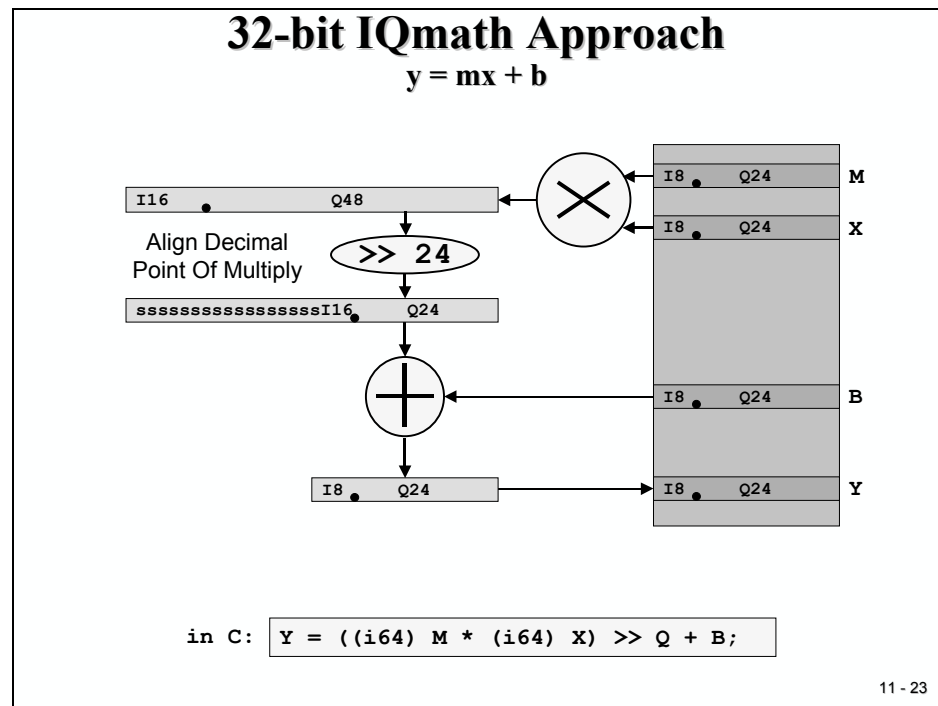
The rescue is the internal hardware arithmetic (Arithmetic Logic Unit and 32-bit by 32-bit Hardware Multiply Unit) of the C28x. These units are able to deal with 64-bit intermediate results in a very efficient way. Dedicated assembly language instructions for multiply and add operations are available to operate on the integer part and the fractional part of the 64-bit number.

To be able to use these advanced instructions, we have to learn about the C28x assembly language in detail. Eventually your professor offers an advanced course in C28x assembly language programming -

OR, just use Texas Instruments "IQ-Math"-library, which is doing nothing more than using these advanced assembly instructions!

32-Bit IQ – Math Approach

The first step to solve the 64-bit dilemma is to refine the last diagram for the 32-bit solution of $Y = MX + B$. As you can see from the next slide the number of shift operations is reduced to 1. Again, the C-line includes a 64-bit ‘long’, which is not available in standard C.



The “IQ”-Math approach ‘redefines’ the multiply operation to use the advantages of the internal hardware of the C28x. As stated, the C28x is internally capable of handling 64-bit fixed-point numbers with dedicated instruction sets. Texas Instruments provides a collection of intrinsic functions, one of them to replace the standard multiply operation by an `_IQmpy(M,X)` –line. Intrinsic means, we do not ‘call’ a function with a lot of context save and restore; instead the machine code instructions are directly included in our source code.

As you can see from the next slide the final C-code looks much better now without the cumbersome shift operations that we have seen in the standard C approach.

AND: The execution time of the final machine code for the whole equation $Y = MX + B$ takes only 7 cycles – with a 150MHz C28x this translates into 46 nanoseconds!

IQmath Approach

Multiply Operation

$$Y = ((i64) M * (i64) X) >> Q + B;$$

Redefine the multiply operation as follows:

$$_IQmpy(M,X) == ((i64) M * (i64) X) >> Q$$

This simplifies the equation as follows:

$$Y = _IQmpy(M,X) + B;$$

C28x compiler supports “_IQmpy” intrinsic; assembly code generated:

```

MOVL    XT,@M
IMPYLL  P,XT,@X      ; P = low 32-bits of M*X
QMPYLL  ACC,XT,@X     ; ACC = high 32-bits of M*X
LSL64   ACC:P,#(32-Q) ; ACC = ACC:P << 32-Q
                        ; (same as P = ACC:P >> Q)
ADDL    ACC,@B        ; Add B
MOVL    @Y,ACC        ; Result = Y = _IQmpy(M*X) + B
; 7 Cycles

```

11 - 24

Let's have a closer look to the assembly instructions used in the example above.

The first instruction 'MOVL XT,@M' is a 32-bit load operation to fetch the value of M into a temporary register 'XT'.

Next, 'XT' is multiplied by another 32-bit number taken from variable X ('IMPYLL P,XT,@X'). When multiplying two 32-bit numbers, the result is a 64-bit number. In the case of this instruction, the lower 32-bit of the result are stored in a register 'P'.

The upper 32 bits are stored with the next instruction ('QMPYLL ACC,XT,@X') in the 'ACC' register. 'QMPYLL' is doing the same multiplication once more but keeps the upper half of the result only. At the end, we have stored all 64 bits of the multiplication in register combination ACC:P.

What follows is the adjustment of the binary point. The 64-bit result in ACC:P is in I16Q48-fractional format. Shifting it 32-24 times to the left, we derive an I8Q56-format. The instruction 'ADDL ACC,@B' uses only the upper 32 Bits of the 64-bit, thus reducing our fractional format from I8Q56 to I8Q24 – which is the same format as we use for B and all our variables!

The whole procedure takes only 7 cycles!

The next slide compares the different approaches. The IQ-Math library also defines a new data type ‘_iq’ to simplify the definition of fractional data. If you choose to use C++ the floating-point equation and the C++ equation are identical! This is possible due to the overload feature of C++. The floating-point multiply operation is overloaded with its IQ-Math replacement – the code looks ‘natural’.

IQmath Approach

It looks like floating-point!

Floating-Point	<pre>float Y, M, X, B; Y = M * X + B;</pre>
Traditional Fix-Point Q	<pre>long Y, M, X, B; Y = ((i64) M * (i64) X + (i64) B << Q)) >> Q;</pre>
“IQmath” In C	<pre>_iq Y, M, X, B; Y = _IQmpy(M, X) + B;</pre>
“IQmath” In C++	<pre>iq Y, M, X, B; Y = M * X + B;</pre>

**Taking advantage of operator overloading feature in C++,
“IQmath” looks like floating-point math (looks natural!)**

11 - 25

This technique opens the way to generate a unified source code that can be compiled in a floating-point representation as well as into a fixed-point output solution. No need to translate a floating-point simulation code into a fixed-point implementation – the same source code can serve both worlds.

IQmath Approach

GLOBAL_Q simplification

User selects “Global Q” value for the whole application

● GLOBAL_Q

based on the required dynamic range or resolution, for example:

GLOBAL_Q	Max Val	Min Val	Resolution
28	7.999 999 996	-8.000 000 000	0.000 000 004
24	127.999 999 94	-128.000 000 00	0.000 000 06
20	2047.999 999	-2048.000 000	0.000 001

```
#define GLOBAL_Q 18    // set in "IQmathLib.h" file
_iq Y, M, X, B;
Y = _IQmpy(M,X) + B;    // all values are in Q = 18
```

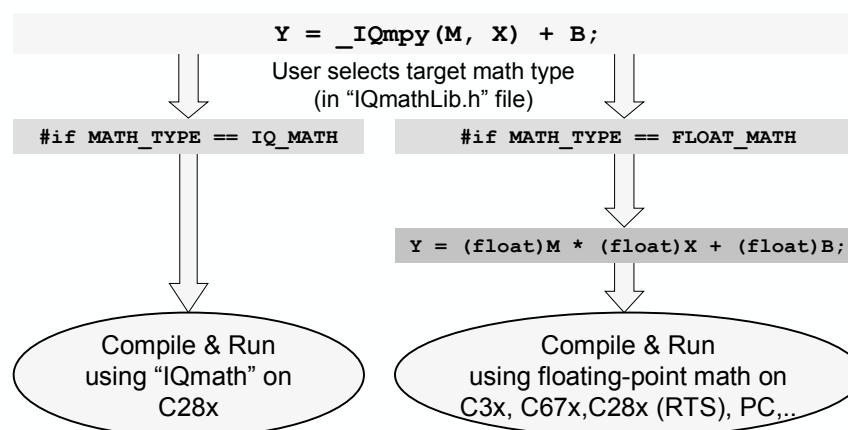
The user can also explicitly specify the Q value to use:

```
_iq20 Y, M, X, B;
Y = _IQ20mpy(M,X) + B;    // all values are in Q = 20
```

11 - 26

IQmath Approach

Targeting Fixed-Point or Floating-Point device



All “IQmath” operations have an equivalent floating-point operation

11 - 27

IQ – Math Library Functions

The next two slides summarize the existing library functions of IQ-Math.

IQmath Library: math & trig functions (v1.4)

Operation	Floating-Point	“IQmath” in C	“IQmath” in C++
type	float A, B;	_iq A, B;	iq A, B;
constant	A = 1.2345	A = _IQ(1.2345)	A = IQ(1.2345)
multiply	A * B	_IQmpy(A, B)	A * B
divide	A / B	_IQdiv(A, B)	A / B
add	A + B	A + B	A + B
subtract	A - B	A - B	A - B
boolean	>, >=, <, <=, ==, !=, &&,	>, >=, <, <=, ==, !=, &&,	>, >=, <, <=, ==, !=, &&,
trig functions	sin(A), cos(A) sin(A*2pi), cos(A*2pi) atan(A), atan2(A, B) atan2(A, B)/2pi sqrt(A), 1/sqrt(A) sqrt(A*A + B*B)	_IQsin(A), _IQcos(A) _IQsinPU(A), _IQcosPU(A) _IQatan(A), _IQatan2(A, B) _IQatan2PU(A, B) _IQsqrt(A), _IQisqrt(A) _IQmag(A, B)	IQsin(A), IQcos(A) IQsinPU(A), IQcosPU(A) IQatan(A), IQatan2(A, B) IQatan2PU(A, B) IQsqrt(A), IQisqrt(A) IQmag(A, B)
saturation	if(A > Pos) A = Pos if(A < Neg) A = Neg	_IQsat(A, Pos, Neg)	IQsat(A, Pos, Neg)

Accuracy of functions/operations approx ~28 to ~31 bits

11 - 28

IQmath Library: Conversion Functions (v1.4)

Operation	Floating-Point	“IQmath” in C	“IQmath” in C++
iq to iqN	A	_IQtoIQN(A)	IQtoIQN(A)
iqN to iq	A	_IQNtoIQ(A)	IQNtoIQ(A)
integer(iq)	(long) A	_IQint(A)	IQint(A)
fraction(iq)	A - (long) A	_IQfrac(A)	IQfrac(A)
iq = iq*long	A * (float) B	_IQmpyI32(A, B)	IQmpyI32(A, B)
integer(iq*long)	(long) (A * (float) B)	_IQmpyI32int(A, B)	IQmpyI32int(A, B)
fraction(iq*long)	A - (long) (A * (float) B)	_IQmpyI32frac(A, B)	IQmpyI32frac(A, B)
qN to iq	A	_QNtoIQ(A)	QNtoIQ(A)
iq to qN	A	_IQtoQN(A)	IQtoQN(A)
string to iq	atof(char)	_atoiQ(char)	atoiQ(char)
IQ to float	A	_IQtoF(A)	IQtoF(A)

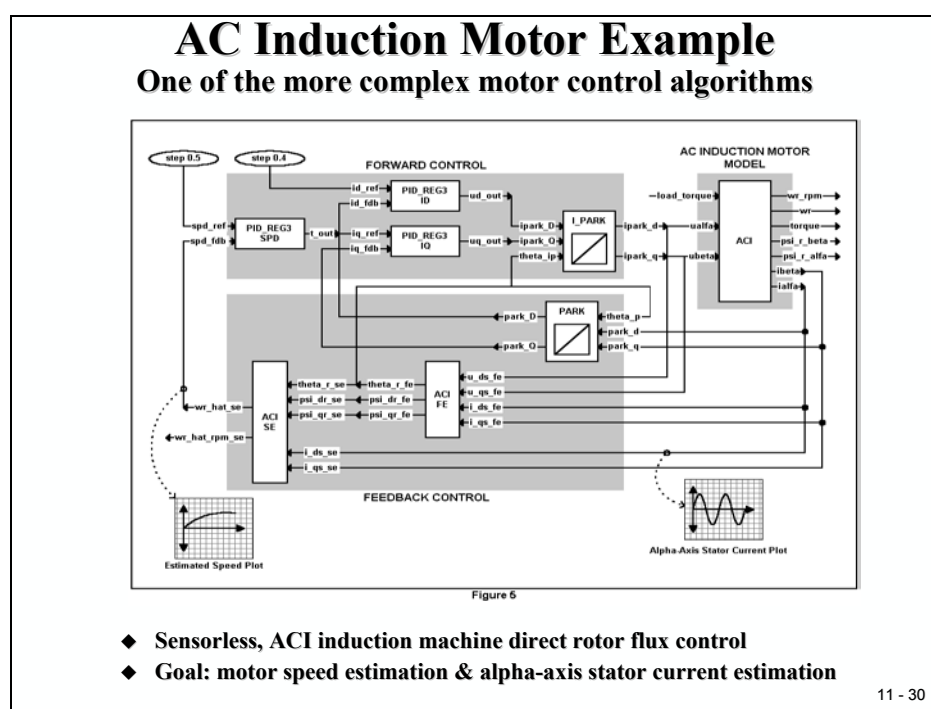
IQmath.lib > contains library of math functions
 IQmathLib.h > C header file
 IQmathCPP.h > C++ header file

11 - 29

IQ- Math Application : Field Orientated Control

The next slides are just to demonstrate the ability of “IQ-Math” to solve advanced numeric calculations in real time. The example is taken from the area of digital motor control. We will not go into the details of the control scheme and we will not discuss the various options to control an electrical motor. If you are a student of an electrical engineering degree you might be familiar with these control techniques. Eventually your university also offers additional course modules with this topic. The field of motor and electrical drive control is quite dynamic and offers a lot of job opportunities.

The next slide is a block diagram of a control scheme for an alternating current (AC) induction motor. These types of motors are based on a three-phase voltage system. Modern control schemes are introduced these days to improve the efficiency of the motor. One principle, called “Space Vector Modulation” or “Field Orientated Control” is quite popular today. In fact this theory is almost 70 years old now, but in the past it was impossible to realize a real time control due to the lack of computing power. Now with a controller like the C28x, it can be implemented!



The core control system consists of three digital PID-controllers, one for the speed control of the motor (“PID_REG3 SPD”), one to control the torque (“PID_REG3 IQ”) and one for the flux (“PID_REG3 ID”). Between the control loops and the motor two coordinate transforms are performed (“PARK” and “I_PARK”).

Let’s have a look into a standard C implementation of the PARK transform, which converts a 3-D vector to a 2-D vector. For now, it is not necessary to fully understand this transform, just have a look into the mathematical operations involved.

All variables are data type “float” and the functions included are:

- Six multiply operations,
- Two trigonometric function calls,
- An addition and
- A subtraction.

This code can easily be compiled by any standard C compiler and downloaded into a simulation or into any processor, for example the C28x. It will work, but it will not be the most efficient way to use the C28x because it will involve floating-point library function calls that will consume a considerable amount of computing time.

AC Induction Motor Example

Park Transform - floating-point C code

```
#include "math.h"
#define TWO_PI 6.28318530717959
void park_calc(PARK *v)
{
    float cos_ang , sin_ang;
    sin_ang = sin(TWO_PI * v->ang);
    cos_ang = cos(TWO_PI * v->ang);

    v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
    v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

11 - 31

With the “IQ-Math” library we can improve the code for the C28x, as shown at the next slide. Of course, we have to replace all float function calls by “IQ-Math” intrinsics.

All variables are now of data type “_iq”, the sine and cosine function calls are replaced by their intrinsic replacements as well as the six multiply operations.

The constant “TWO_PI” will be converted into the standard IQ-format with the conversion function “_IQ()”. This way the number 6.28 will be translated into the correct fixed-point scale before it is used during compilation.

The resulting code will be compiled into a much denser and faster code for the C28x. Of course, a little bit of coding is still needed to convert an existing floating-point code into the “IQ-Math” C-code.

Fortunately, the structure of the two program versions is identical, which helps to keep a development project consistent and maintainable, for both the floating-point and a fixed-point implementations.

AC Induction Motor Example

Park Transform - converting to “IQmath” C code

```
#include "math.h"
#include "IQmathLib.h"
#define TWO_PI _IQ(6.28318530717959)
void park_calc(PARK *v)
{
    _iq  cos_ang , sin_ang;
    sin_ang = _IQsin(_IQmpy(TWO_PI , v->ang));
    cos_ang = _IQcos(_IQmpy(TWO_PI , v->ang));

    v->de = _IQmpy(v->ds , cos_ang) + _IQmpy(v->qs , sin_ang);
    v->qe = _IQmpy(v->qs , cos_ang) - _IQmpy(v->ds , sin_ang);
}
```

11 - 32

If we go further on and use a C++ compiler to translate the “IQ-Math” code, we can take advantage of the overload technique of C++. The result for this PARK-transform is shown at the next slide.

AC Induction Motor Example

Park Transform - converting to "IQmath" C++ code

```
#include "math.h"
extern "C" { #include "IQmathLib.h" }
#include "IQmathCPP.h"

#define TWO_PI  IQ(6.28318530717959)

void park_calc(PARK *v)
{
    iq    cos_ang , sin_ang;
    sin_ang = IQsin(TWO_PI * v->ang);
    cos_ang = IQcos(TWO_PI * v->ang);

    v->de = (v->ds * cos_ang) + (v->qs * sin_ang);
    v->qe = (v->qs * cos_ang) - (v->ds * sin_ang);
}
```

11 - 33

The multiply operation looks identical in floating-point and in fixed-point implementation. It is quite a simple and fast procedure to take any floating-point algorithm and convert it to an "IQ-Math" algorithm.

The complete system was coded using "IQ-Math". Based on analysis of coefficients in the system, the largest coefficient had a value of 33.3333. This indicated that a minimum dynamic range of 7bits (+/-64 range) was required. Therefore, this translated to a GLOBAL_Q value of $32-7 = 25(Q25)$. Just to be safe, the initial simulation runs were conducted with GLOBAL_Q = 24 (Q24) value.

Next, the whole AC induction motor solution was investigated for stability and dynamic behavior by changing the global Q value. With a 32-bit fixed-point data type we can modify the fractional part between 0 bit ("Q0") and 31 bits ("Q31"). The results are shown below. As you can see, there is an area, in which all tests led to a stable operating mode of the motor. The two other areas showed an increasing degree of instability, caused by either not enough dynamic range in the integer part or not enough fractional resolution of the numbering system.

AC Induction Motor Example

Q stability range

Q range	Stability Range
Q31 to Q27	Unstable (not enough dynamic range)
Q26 to Q19	Stable
Q18 to Q0	Unstable (not enough resolution, quantization problems)

The developer must pick the right GLOBAL_Q value!

11 - 34

Where Is IQmath Applicable?

Anywhere a large dynamic range is not required

Motor Control (PID, State Estimator, Kalman,...)
 Servo Control
 Modems
 Audio (MP3, etc.)
 Imaging (JPEG, etc.)
 Any application using 16/32-bit fixed-point Q math

Where it is not applicable

Graphical applications (3D rotation, etc.)
 When trying to squeeze every last cycle

11 - 35

IQmath Approach Summary

“IQmath” + fixed-point processor with 32-bit capabilities =

- ◆ Seamless portability of code between fixed and floating-point devices
 - User selects target math type in “IQmathLib.h” file
 - `#if MATH_TYPE == IQ_MATH`
 - `#if MATH_TYPE == FLOAT_MATH`
- ◆ One source code set for simulation vs. target device
- ◆ Numerical resolution adjustability based on application requirement
 - Set in “IQmathLib.h” file
 - `#define GLOBAL_Q 18`
 - Explicitly specify Q value
 - `_iq20 X, Y, Z;`
- ◆ Numerical accuracy without sacrificing time and cycles
- ◆ Rapid conversion/porting and implementation of algorithms

***IQmath library is freeware - available from TI DSP website
<http://www.dspvillage.ti.com> (follow C2000 DSP links)***

11 - 36

Introduction

Chapter 12 introduces Texas Instruments Built-In Operating System (“BIOS”) for Digital Signal Processors - “DSP/BIOS”. This firmware will allow you to use a range of a Real Time Operating System (“RTOS”) features in an embedded control application. It is not the intention of this module to cover the theory of Operating Systems and the specific demands of a real time control applications. To be able to work with all options of DSP/BIOS it will be necessary for you to enrol in additional classes about this topic at your university. When you have completed these classes, it is highly recommended to return to this module after you are familiar with the theory of multi processing and real time control. With this knowledge, you can enhance the examples shown in this chapter to their full extent.

We will now have a brief look into the essentials of DSP/BIOS. We shall start with the introduction of core terms and definitions of a real time system:

- What is a RTOS?
- What is a task?
- What is the task state model?
- What is a scheduler?

Then we will discuss and use the BIOS system configuration tools to simplify the setup of a C28x code project.

Next, we will take a closer look to some of the basic concepts of DSP/BIOS:

- Scheduler
- Hardware Interrupts (HWI)
- Software Interrupts (SWI)
- Periodic Functions (PRD)

At the end of this module we will return to the laboratory and modify one of our existing programs to use both the DSP/BIOS configuration tool and a periodic function.

Module Topics

C28x DSP/BIOS.....	12-1
<i>Introduction</i>	<i>12-1</i>
<i>Module Topics.....</i>	<i>12-2</i>
<i>Real Time Operating System.....</i>	<i>12-3</i>
<i>A RTOS - Task.....</i>	<i>12-4</i>
<i>Task State Model.....</i>	<i>12-5</i>
<i>The RTOS - Scheduler.....</i>	<i>12-7</i>
<i>Texas Instruments DSP/BIOS</i>	<i>12-8</i>
<i>DSP/BIOS Configuration Tool.....</i>	<i>12-10</i>
<i>DSP/BIOS Task Groups</i>	<i>12-11</i>
Hardware Interrupts (HWI)	12-14
Software Interrupts (SWI)	12-16
Periodic Functions (PRD).....	12-17
<i>Real-Time Analysis Tool.....</i>	<i>12-19</i>
<i>DSP/BIOS API.....</i>	<i>12-20</i>
<i>Lab Exercise 12.....</i>	<i>12-21</i>
Objective	12-21
Procedure.....	12-21
Open Files, Create Project File.....	12-21
Project Build Options	12-22
Modify Source Code.....	12-23
Edit DSP/BIOS Configuration.....	12-23
Build the project	12-25
Test the code.....	12-25
Potential Solution for function “led_runner”.....	12-26

Real Time Operating System

You are probably familiar with the basics of the operating system (OS) of your PC. One of the typical features of this OS is the ability to start and work on different processes simultaneously. You can open your word processor to edit a document and at the same time you can browse the web in another application process or you can check your emails. The term for this type of OS is “multi tasking”. The more processes you open the more difficult it will become for your OS to respond in time. Sometimes one of your processes will be ‘blocked’ for a while before it continues. It becomes more difficult for the OS if dedicated hardware interactions are involved, like burning a CD/DVD. In these cases, your application probably gives out a meek recommendation like: “Do not start any other application now...” to assure that its own procedure will be completed in time.

In short, this type of OS does not guarantee any completion time for a process! Sometimes a process is not completed at all; you are no doubt familiar with the keys “CTRL-ALT-DEL”.

Now imagine an embedded control application such as an autopilot of an airplane or a car braking system controlled by this type of operating system.

For this type of application, we need another type of OS that will guarantee a time line for all processes involved. We also need to know a “worst case response time” for all interactions and event processing of the control system - a “Real Time Operating System”

Real Time Operating Systems (RTOS)

What is a RTOS?

- ◆ **Particular class of operating systems for digital processor systems**
- ◆ **Capable of serving multiple user tasks at one time (“multi-task OS”)**
- ◆ **For all tasks in a running system it is guaranteed, that random external events in the environment of each individual task will be serviced in a given time (“Worst Case Response Time”).**
- ◆ **All tasks must be served simultaneously (“multi-task OS”) AND timely (“RTOS”)**
- ◆ **RTOS are very popular in embedded control**

12 - 2

A RTOS - Task

One of the basic terms of a RTOS is 'task'. A task is an independent portion of the whole control solution that is driven by its own program code. This owns a predefined set of resources and interacts with other tasks only by means of the RTOS. A task is characterised by its own set of state variables, its own local stack and program counter.

Real Time Operating Systems (RTOS)

What is a Task?

- ◆ **A running or executable program object, that:**
 - **is controlled by a portion of machine code**
 - **'owns' a given set of operating resources to start/resume its course of actions**
 - **is characterized by a set of state variables (registers, program counter, local stack variables, semaphores, mailboxes)**
- ◆ **Tasks are programmed and debugged independently from each other**
- ◆ **Accesses to peripherals or data transfers between tasks are performed by RTOS - system functions calls.**

12 - 3

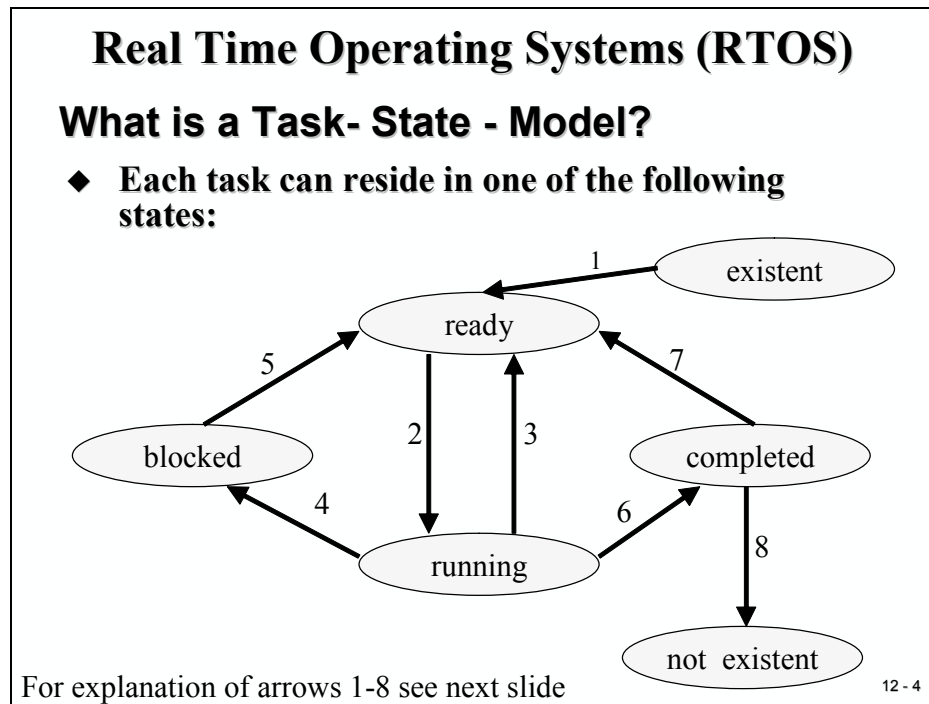
Tasks are designed and debugged independently from other activities of the control unit. Each task follows the design rules of the RTOS. To estimate the workload of the computing system, each task is characterized by the following timing information:

- Best case execution time (BCET) in number of CPU-cycles
- Worst case execution time (WCET) in number of CPU-cycles
- Period of task or time interval between two consecutive requests to start the task
- Amount of time for resource usage, block time
- Task "Dead Line" - latest time when the task must be finished, measured from the point of requesting the task.

For a "hard"-RTOS, no task must fail to meet its Dead Line under any circumstances, throughout the whole lifetime of the control system. Most embedded systems are of this type.

Task State Model

Each task will be in one of the states given in the task state model (see slide 12-4). The RTOS takes control of the current status and possible transitions for each individual state.



The states are:

- Existent - The task is installed in the control system but not yet activated.
- Ready - The task is activated and ready for running. It owns all necessary resources but the processor.
- Running - The task is currently executing
- Blocked - The task is waiting for a resource, a message, a signal or an event, which has not yet occurred.
- Completed – The task has terminated and awaits a new activation.
- Not Existent – The task has been cancelled permanently.

The transitions between the states of a task are explained at the next slide. It is important to remember that some of the transitions are initiated by the RTOS-Scheduler, whereas others are performed by the task that is currently in state “running”.

Real Time Operating Systems (RTOS)

When does the task – state change?

1. **A task is created by an initialization function**
 2. **A task is selected by the scheduler to use the CPU**
 3. **The scheduler performs a task change according to the scheduling rules of the RTOS**
 4. **The running task is waiting for an external event, a message from another task or for a signal**
 5. **The event that was blocking a task has occurred**
 6. **The task has completed its program**
 7. **The task is re-activated by another task or by an event**
 8. **The task will never be used again (as long as the embedded system is not switched off)**
- All other task state transitions are illegal.**

12 - 5

The transitions are:

- (1) - A task is activated by another task. This is usually done by an initialization function during startup. Alternately, a running task needs to be supported by a new 'child' task and therefore activates a new one.
- (2) - According to the rules of the RTOS, one of the tasks in the “ready” state is selected by the scheduler to use the processor.
- (3) - The scheduler decides to swap the task in the “running” state with another task in the “ready” state according to the scheduling rules of the RTOS.
- (4) - The “running task” waits for an external signal, for a message or for any other resource that is not available at this time.
- (5) - The scheduler administrates all blocked tasks with their blocking events. If one event occurs, the blocked task will be freed and merge back into the “ready” state.
- (6) - The running task has terminated.
- (7) - The task is re-activated by the running task.
- (8) - The task is permanently deleted from the system by the running task.

The RTOS - Scheduler

One of the core modules of a RTOS is the scheduler. As the name implies, it schedules the execution of all tasks in a control system. It uses the task state model and the rules of the RTOS to work out how to handle multiple requests at one time. The main task of a RTOS-Scheduler is to interleave all tasks in state “ready”, so that each individual task is able to meet its specified dead line.

Real Time Operating Systems (RTOS)

What is a Task - Scheduler?

- ◆ **An important part of the RTOS that schedules the sequence of task execution phases and the change of task states**
- ◆ **Two basic operating modes for schedulers:**
 - **time slice mode - computing time is assigned to tasks in a predefined amount of processor time**
 - **priority mode – computing time is assigned to tasks according to the priority of each task in the system. If a task with higher priority gets into status ‘ready’ the running task will be pre-empted.**
 - **combined versions between pre-emptive and time slice schedulers are also possible**

12 - 6

A scheduler is no magic trick; it follows strict rules that were defined during the system configuration of the embedded system. Depending on the ability of the RTOS, we can choose between different options to tailor the scheduler. In general, there are two basic operating modes for the scheduling rules:

- **Time Slice Mode** - CPU time is granted to the tasks in fixed amounts of time. If the slice was consumed by the task before it could finish its operation the next “ready” task will be selected by the scheduler. The old task is suspended and will be added back at the end of the waiting list in state “ready”. It will resume its operation when the next slice is granted by the scheduler.
- **Priority Mode** - The scheduler uses the priority of a task to perform task switches. If there is a task in state “ready” with a higher priority than the running task, the scheduler will swap the two tasks. Some RTOS allow dynamic change of the priority depending on the current situation of the control system, others use static priorities. Common rules are “Dead Line Monotonic”, “Rate Monotonic”, “Earliest Dead Line First” or “Least Laxity”.

Texas Instruments DSP/BIOS

For the TMS320 family of Digital Signal Processors Texas Instruments offers a Built-In Operating System (BIOS) to support the setup of embedded control systems and to help designers to build control schemes according to the “philosophy” of Real Time Operating Systems. DSP/BIOS allow the designer to split a control project into independent sub-modules and to control the interaction between the “tasks” by means of the operating system. Instead of writing your own code to initialize and administrate interrupt vector tables, service routines and internal hardware modules, you can use standardized accesses to do so.

DSP/BIOS is available as a scalable real time kernel, it can be tailored in its size to the needs of the developer. It features all the typical functions of a RTOS and is equipped with a pre-emptive priority driven scheduler.

During debug a new real time analysis tool is available to verify the correct functionality of the system solution.

Texas Instruments DSP/BIOS

What is DSP/BIOS?

- ◆ **BIOS = “Build In Operating System”**
- ◆ **Texas Instruments firm ware RTOS kernel for the TMS320 family of DSP's**
- ◆ **A full-featured, scalable real-time kernel**
 - ◆ **System configuration tools**
 - ◆ **Preemptive multi-threading scheduler**
 - ◆ **Real-time analysis tools**

12 - 7

DSP/BIOS is an integrated part of Code Composer Studio and can be included to any CCS project as simply as adding another source code file to an existing project. Lab12 will prove this.

The next slide summarizes some reasons to take into account when starting a new project. Especially with the C28x and its powerful math units, it is worth considering the use of DSP/BIOS.

For most car manufacturers it is mandatory to have electronic control units equipped with RTOS functionality. If you decide to build your professional career in the automotive industry, you won't be able to skip this point.

Texas Instruments DSP/BIOS

Why use DSP/BIOS?

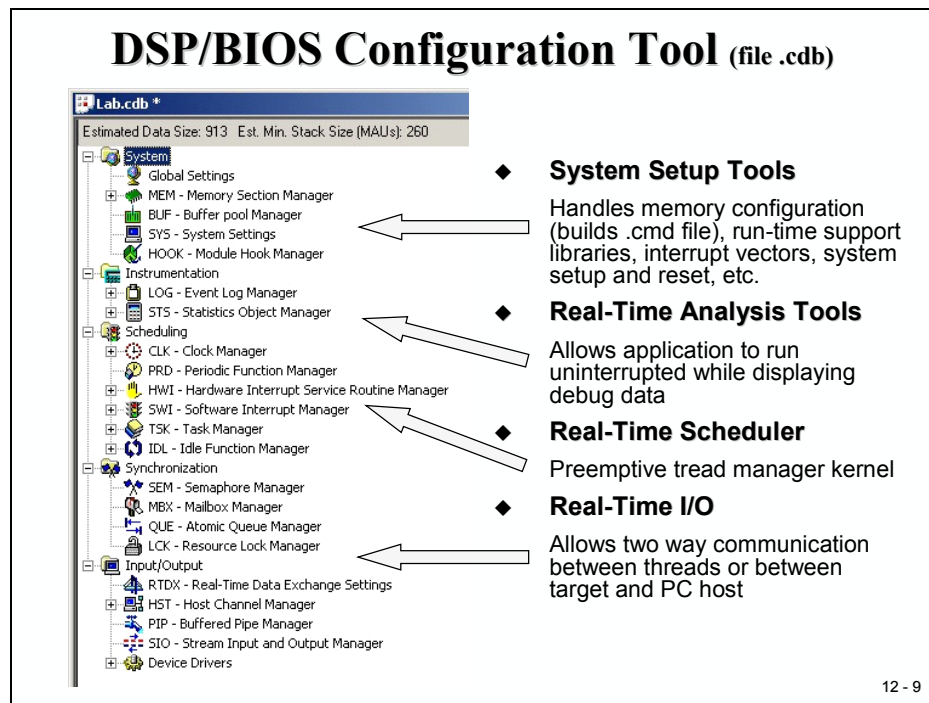
- ◆ **Helps manage complex C28x system resources**
- ◆ **Allows to develop and test tasks in a multiple task control environment independently**
- ◆ **Reduces software project development time**
- ◆ **Eases software maintenance & documentation**
- ◆ **Integrated with Code Composer Studio IDE**
 - ◆ **Requires no runtime license fees**
 - ◆ **Fully supported by TI and is a key component of TI's eXpressDSP™ real-time software technology**
- ◆ **Uses minimal MIPS and memory (2-8K)**

12 - 8

DSP/BIOS Configuration Tool

The following slide highlights all major blocks of DSP/BIOS. They are:

- The System Configuration Tool (“System”)
- The Real-Time Scheduler (“Scheduling”)
- The Real-Time Analysis Tool (“Instrumentation”)
- The Task Synchronisation Module (“Synchronisation”) and
- The Real-Time Data Exchange Support (“Input/Output”)

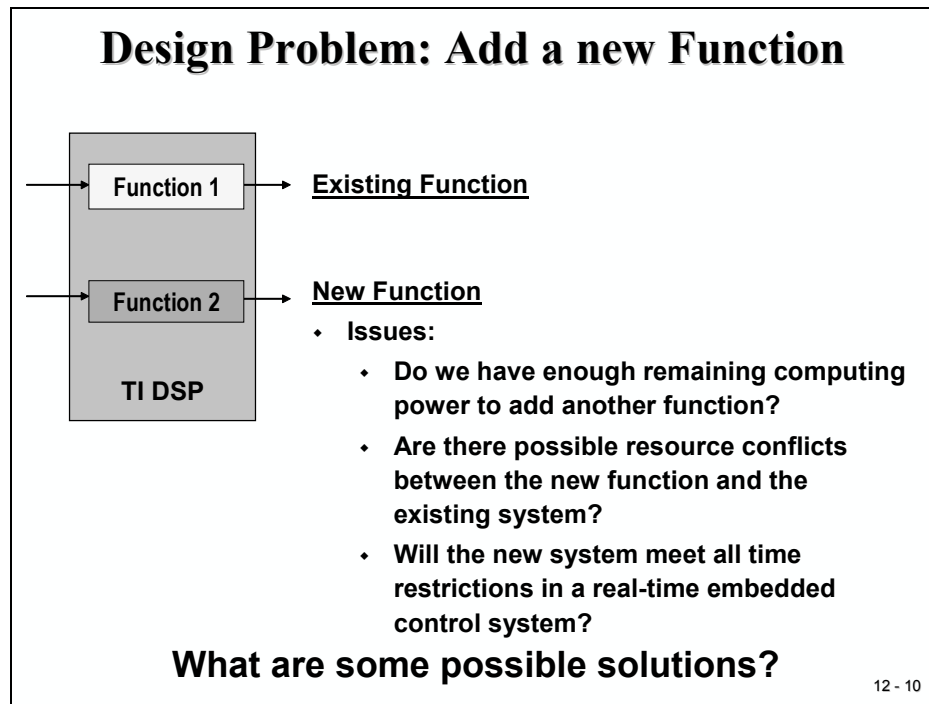


The first part of DSP/BIOS that will be useful during the set-up of a CCS project is the System Configuration Tool. In general, this tool will replace the handwritten linker command file that we have used so far in our lab exercises. With the help of the Configuration Tool, we can define new sections of physical memory and connect logical software segments to memory locations (“MEM”). Global settings, like defining the stack size, are also done within this tool. In Lab exercise 12, we will use this tool to setup the linker environment.

With the help of the scheduler group, we can configure the different tasks that are part of the project. There are four different groups of tasks available: Hardware – Interrupts (“HWI”), Software Interrupts (“SWI”), Tasks (“TSK”) and Periodic Functions (“PRD”). We will use the scheduler part to connect RTOS-function to our software modules.

DSP/BIOS Task Groups

If we look into a typical development cycle of a project that is not driven by an OS, we will face the following design problem:



A new function is to be added to our existing project. A professional embedded system designer would NOT start immediately with the new coding and hope that afterwards everything will work as expected. That would be the 'trial and error' method and is no good at all for safety critical applications. Instead, a serious developer would try to answer the following questions first:

- How much computing power is still available BEFORE we add a new function?
- How much storage capacity will the new function need? Is there enough code memory left?
- What type of resources will be needed by the new function? Are there potential conflicts with other functions in the system?
- Are there any interactions, synchronisations or message transfers between the new function and the existing functions needed?
- What is the deadline of the new function, once it is called? What is the estimated computing time for the new function?
- Do I have to prioritize the new function against others? Are there consequences for the execution of other functions, when I have to modify the priorities of existing functions?

If the new system seems feasible and we do not have an OS-support, we could add the new function as shown at the next slide:

Solution 1: extend the main-loop

```
Main()
{
  while(1)
  {
    Function 1
    Function 2
  }
}
```

- ◆ **Call each function from an *endless loop* within main**
- ◆ **Potential Problems:**
 - What if Algorithms run at different rates:
 - motor current loop at 20 kHz
 - respond to keypad input at 2 Hz
 - What if one algorithm consumes enough MIPS to force the other algorithm to miss its real-time deadlines / delays its response?

12 - 11

This is a simple solution that might work for a few projects. But with this scheme we can face new problems:

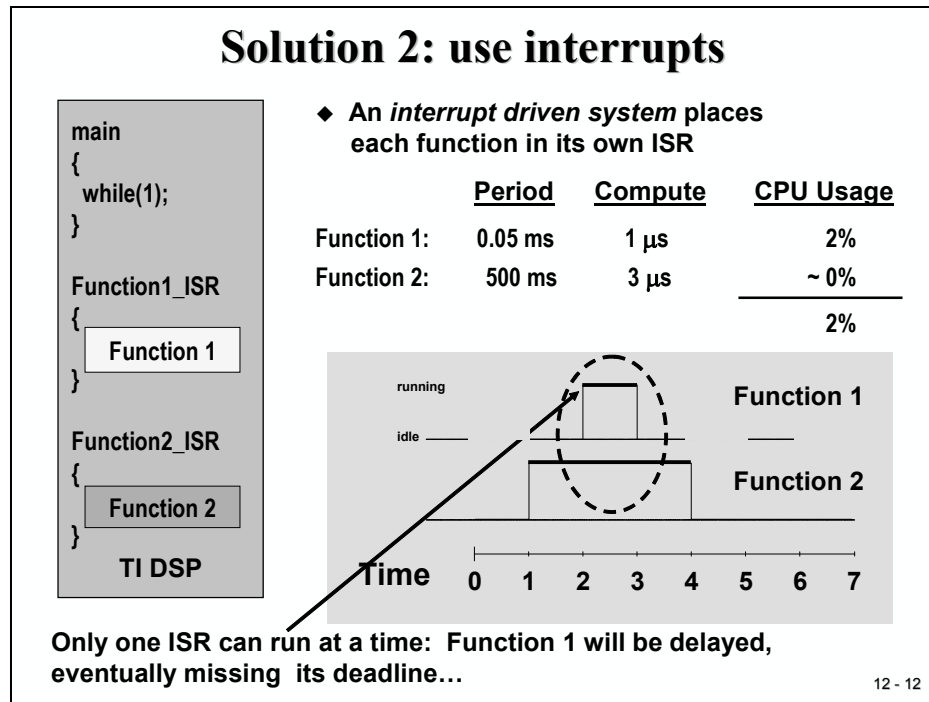
- What if function 1 has to be executed more frequently than function 2? Answer: we could call function 1 more than once before we call function 2:

```
while(1)
{
    function_1();
    function_1();
    ...
    function_1();
    function_2()
}
```

- But what if function2 takes too long to execute? We might reach the next call of function1 too late for its real time interactions! In this case we would have to completely re-write function2, split it into several parts and interleave calls to function1. The resulting code would not look structured at all!

Obviously there must be a more appropriate way for this type of real-time programming!

If you already have some basic experience with microprocessors, you may have suggested using interrupts instead of the function loop in solution1. This principle is shown with the next slide:



We distribute all functions into dedicated Interrupt Service Routines (ISR). The main function will perform basic initialization routines for the project like system setup, memory initialization, peripheral unit setups, enabling of desired interrupts, prepare timer periods etc. At the end of main the code will stay in an idle-loop, in C usually coded as “while(1);” or “for(; ;)”. When one of the hardware units, for example a timer, calls its ISR, the assigned function will be executed. After finishing this ISR the processor will return to main’s idle-loop and await the next interrupt.

Our previous example could use a first timer that is initialized to a period of 50 μ s and a second timer with a period of 0.5s. All we would have to do is to connect timer 1 interrupt service to function 1 and timer 2 to function 2.

This principle will work fine for most of the time, but sometimes our control unit will behave unexpectedly. Why is this?

If the processor executes function 2 and at the same time function 1 is requested by timer 1 ISR, the processor will finish function 2 first before it deals with the next request. In this case the execution of function 1 will be delayed. In best cases our system will behave a little bit sluggishly, in worst cases the motor that is controlled by function 1 will change from full forward to full reverse....

Hardware Interrupts (HWI)

The usual back door way out of this dilemma is using interrupts that can be interrupted by other interrupts. This principle is called ‘nested’ interrupts and is usually the best solution for this type of real-time projects. Nesting interrupts requires the save of the environment (registers, stack pointer, flags etc) and status of the running interrupt before the processor can switch to the next interrupt service. Before the processor can switch back to the interrupt that was paused, the status must be restored. This principle is called ‘context save & restore’. If you do not have the support of an OS, you would be responsible to take care of these additional steps. Or, your C-compiler adds context save & restore functions automatically – usually when you mark a function with keyword “interrupt”.

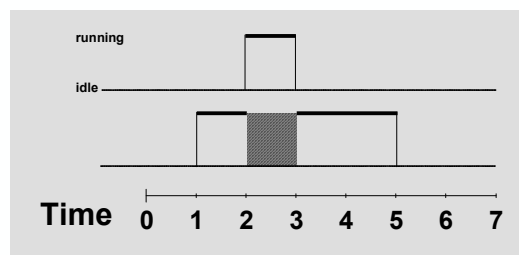
Solution 3: nested hardware interrupts (HWI)

```
main
{
    return;
}

Function1_ISR
{
    Function 1
}

Function2_ISR
{
    Function 2
}
```

- ◆ **Nested interrupts allow hardware interrupts to preempt each other**



- ◆ Use DSP/BIOS HWI dispatcher for context save/restore, and allow preemption
- ◆ Reasonable approach if you have limited number of interrupts/functions
- ◆ one HWI function for each interrupt

12 - 13

In case of DSP/BIOS the OS itself takes care of context switch. The Texas Instruments syntax for nested interrupts is called “Hardware Interrupt (HWI)”. For the C28x we can assign one HWI function for each physical interrupt line INT1-INT14.

NOTE: To use DSP/BIOS as background task system, we have to leave the main function after all initialization is done! Make sure that you do not have any endless loop construct still in your main – code.

DSP/BIOS will call its own idle – function if there are no other activities necessary at the moment.

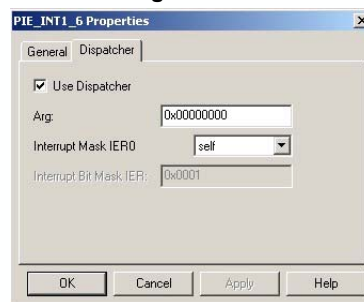
To adapt a non-DSP/BIOS code for interrupt service routines you will have to delete the keyword “interrupt” from an existing service routine.

DSP/BIOS - HWI Dispatcher for ISRs

- ◆ For non-BIOS code, we use the *interrupt* keyword to declare an ISR
 - tells the compiler to perform context save/restore

```
interrupt void MyHwi(void)
{
}
```

- ◆ For DSP/BIOS code, the dispatcher will perform the save/restore
 - Remove the *interrupt* keyword from the MyHwi()
 - Check the “Use Dispatcher” box when you configure the interrupt vector in the DSP/BIOS config tools



12 - 14

Next you will have to tell the DSP/BIOS configuration tool to take care of context switches. Open the configuration database file (*.cdb) and click on

➔ Scheduling ➔ HWI Hardware Interrupt Service Routine Manager

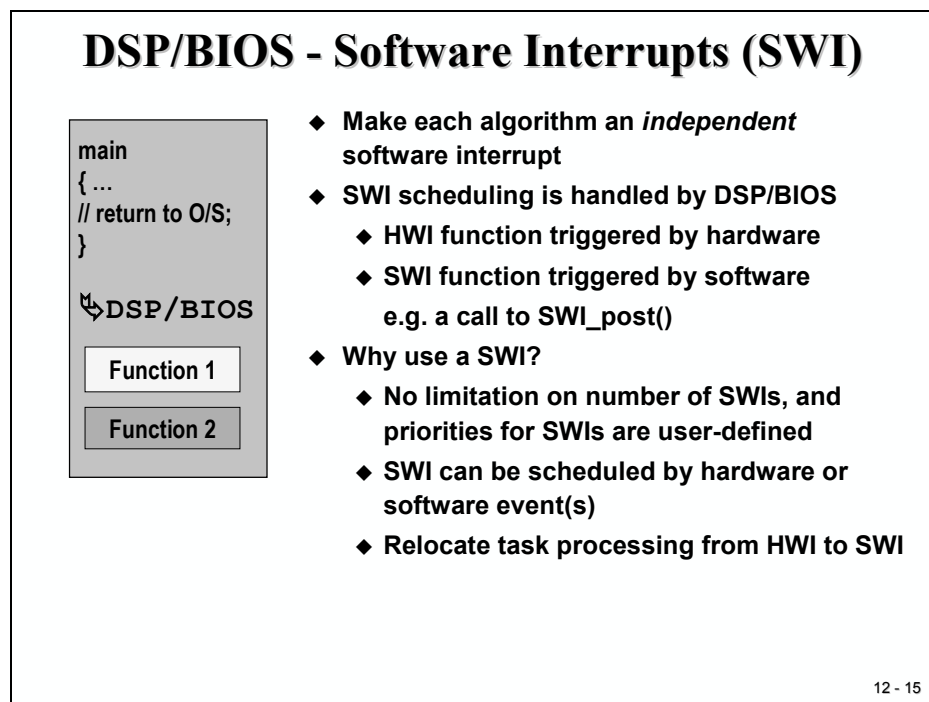
Expand the ‘+’-sign in front of HWI, click right on the HWI that you want to modify (e.g. “HWI_INT1”) and select “Properties”

Under “General” you can specify the name of the function that you’d like to connect to this HWI, e.g. `_MyHwi`. The underscore in front of the name of the function is mandatory to indicate a C code function.

Under “Dispatcher” enable the box “Use Dispatcher”, this will include the context save and restore functions that are needed to allow nested hardware interrupts.

Software Interrupts (SWI)

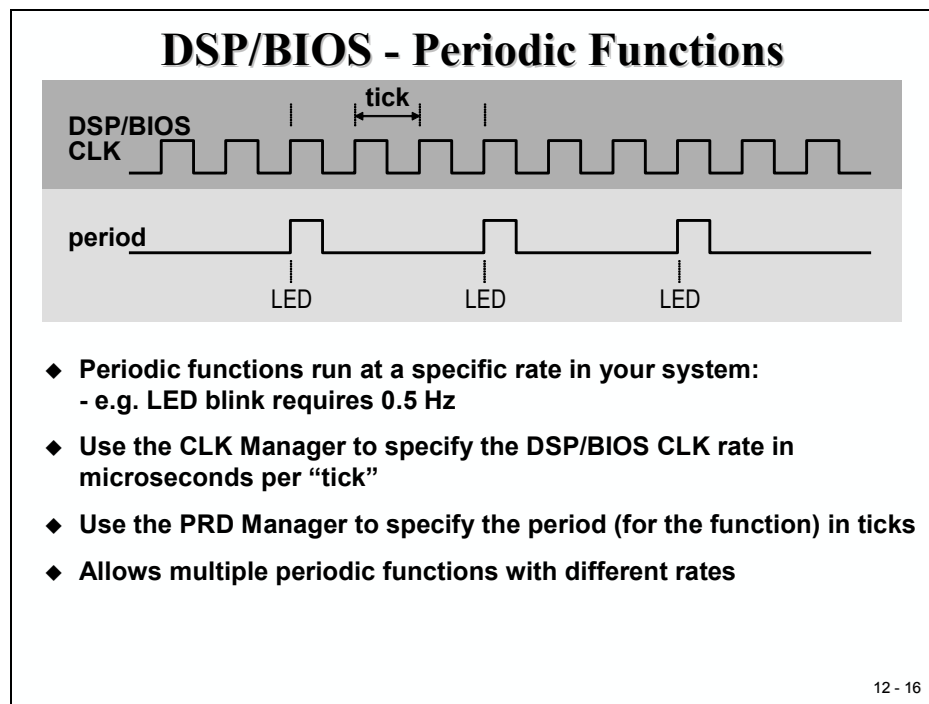
The next group of DSP/BIOS functions is called “Software Interrupts (SWI)”. As the name says, this group of functions is initiated by software requests. The difference to HWI’s is that the user has more flexibility in the system design when using software interrupts. Whilst the priority of all hardware interrupts is given by the C28x and can’t be changed at all, the user has full flexibility to assign priorities to software interrupts as static or dynamic priorities. The number of software interrupts is not limited by DSP/BIOS; of course the number of feasible software interrupts depends on the computing time that is consumed by all software interrupt functions.



A recommended principle is to limit the length of hardware interrupt service routines to an absolute minimum, just what is necessary to deal with the hardware event. All other activities that must be executed due to the hardware event are relocated into software interrupt services. The HWI schedules all actions that should follow the hardware event. Thus the DSP/BIOS controlled project is most flexible.

Periodic Functions (PRD)

Another group of DSP/BIOS objects are called “Periodic Functions”. They are based on multiples of the DSP/BIOS system clock and are usually used to call a function at equally spaced time intervals, e.g. blinking a LED. Recall our previous lab exercises where we used one of the C28x core timers or an event manager timer just to create a period for a next event. We can simplify these exercises by replacing all the timer functions by a periodic DSP/BIOS function!



First we have to specify the DSP/BIOS clock rate (“ticks”) in microseconds per tick. This is done with the CLK Manager of the Configuration Tool.

Next we can use the Periodic Function Manager (“PRD”) to define the period for a specific function in number of ‘ticks’. We also have to connect the PRD-event with a function from our source code.

The procedure for creating a periodic event in DSP/BIOS is shown at the next slide. Later in lab exercise 12 we will use a periodic function to modify the status of our 8 LED output lines periodically.

The two steps to prepare a periodic function are shown at the slide:

Creating a Periodic Function

CLK - Clock Manager Properties

General

Object Memory: LOSARAM

☒ Enable CLK Manager

☒ Use high resolution time for internal timings

Microseconds/Int.: 1000.0000

☐ Directly configure on-chip timer registers

☐ Fix TDDR

TDDR Register: 2

PRD Register: 43999

Instructions/Int.: 150000

OK Cancel Apply

Lab.cdb

Estimated Data Size: 948 Est. Min. Stack Si

- System
- Instrumentation
- Scheduling
- CLK - Clock Manager
- PRD - Periodic Function Manager
- LedBlink_PRD**
- HWI - Hardware Interrupt Service
- SWI - Software Interrupt Manage
- TSK - Task Manager
- IDL - Idle Function Manager
- Synchronization
- Input/Output

LedBlink_PRD Properties

General

comment: <add comments here>

period (ticks): 500

mode: continuous

function: _LedBlink

arg0: 0x00000000

arg1: 0x00000000

period (ms): 500.0

OK Cancel Apply

12 - 17

Finally, to start DSP/BIOS we have to terminate the main function:

Enabling BIOS – Return from main()

```

main
{ ...
// return to BIOS
}
        
```

➡ DSP BIOS

Function 1

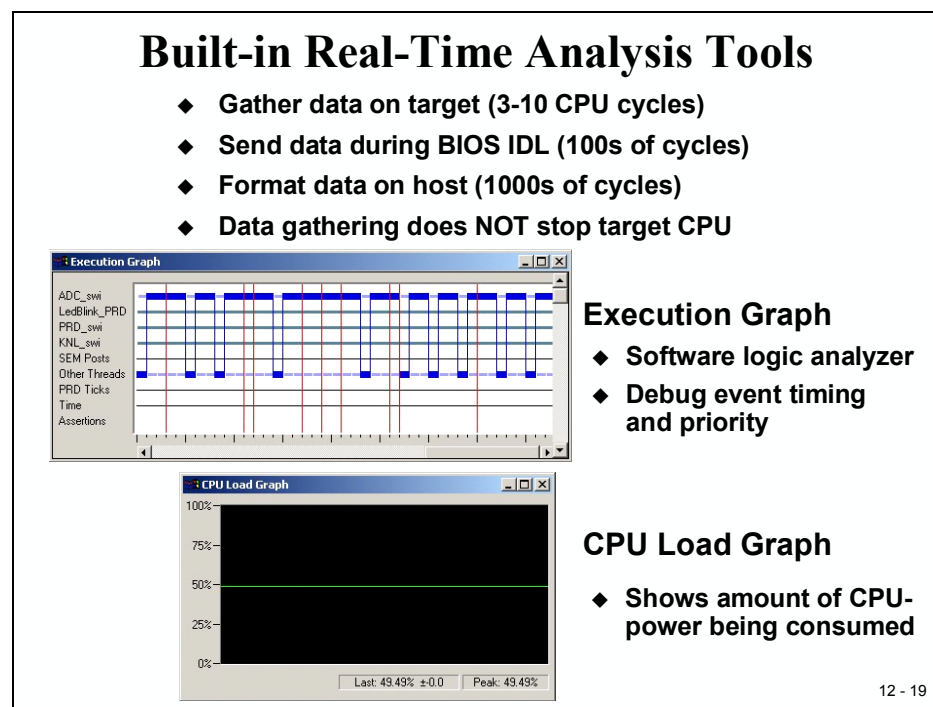
Function 2

- ◆ Must delete the endless while() loop
- ◆ main() returns to BIOS IDLE thread, allowing BIOS to schedule events, transfer info to host, etc.
- ◆ An endless while() loop in main() will not allow BIOS to activate

12 - 18

Real-Time Analysis Tool

Important test support tools for RTOS based projects are analysis tools to measure and document the efficiency and the feasibility of the implementation. This includes graphical displays for resource usage, CPU load, the sequence of task changes during test runs and measurement tools for blocking times of resources by individual tasks. At the end of the day the designer must answer all the questions that we have discussed at beginning of this module with solid facts. A widely used tool that is used to support these debug and test phases is a logic analyzer, used in combination with a real time data logging system.



Texas Instruments provides some of these support tools as integrated parts of DSP/BIOS. The Execution Graph shows the flow and interleaving of tasks and interrupts on a time base. The CPU Load Graph gives a graphical view of the C28x computing time that is used by the whole project, excluding the DSP/BIOS idle function.

Another tool is the “Event Log Manager (LOG)”. In order to send debug information about the sequence of instructions to a terminal window a designer quite often uses ‘printf’- instructions. The disadvantage is the modification of the code by these additional instructions. With DSP/BIOS we can use the LOG – window for this purpose. All logging instructions will be executed by the OS in background, hence not delaying the original flow of program execution.

DSP/BIOS API

The next slide is a summary of all API Modules that are part of DSP/BIOS. In the next lab exercise we will use the Memory Manager (MEM), System Clock Manager (CLK) and Periodic Function Manager (PRD).

DSP/BIOS - API Modules	
Instrumentation/Real-Time Analysis	
LOG	Message log manager
STS	Statistics accumulator manager
TRC	Trace manager
RTDX	Real-Time Data eXchange manager
Thread Types	
HWI	Hardware interrupt manager
SWI	Software interrupt manager
TSK	Multi-tasking manager
IDL	Idle function & process loop manager
Clock and Periodic Functions	
CLK	System clock manager
PRD	Periodic function manager
TSK Communication/Synchronization	
SEM	Semaphores manager
MBX	Mailboxes manager
LCK	Resource lock manager
Device-Independent Input/Output	
PIP	Data pipe manager
HST	Host input/output manager
SIO	Stream I/O manager
DEV	Device driver interface
Memory and Low-Level Primitives	
MEM	Memory manager
SYS	System services manager
QUE	Queue manager
ATM	Atomic functions
GBL	Global setting manager

12 - 20

Lab Exercise 12

Lab 12: Modify Lab 2 to use BIOS

- Use your solution for Lab2 to begin with
 - Modify the project to use DSP/BIOS functionality
 - Let DSP/BIOS create the necessary Linker Command Files
 - Replace the software delay loop from Lab2 by a periodic function ("PRD") of DSP/BIOS
 - Create a new function "led_runner()" that will activate the next state of the LED-line
 - Call this function every 500ms with a PRD-function out of DSP/BIOS
-
- For a detailed procedure see textbook!

12 - 21

Objective

The objective of this laboratory exercise is to implement the code from Lab2, this time using DSP/BIOS periodic functions rather than a simple software delay loop to generate the time separation.

Instead of using the linker command file from Lab2 we will use the DSP/BIOS Configuration Tool to auto generate the linker information

We will add a periodic function (PRD) to the DSP/BIOS system that will be used to execute the next state of our control sequence.

Procedure

Open Files, Create Project File

1. Create a new project, called **Lab12.pjt** in E:\C281x\Labs.
2. Open the file Lab2.c from E:\C281x\Labs\Lab2 and save it as Lab12.c in E:\C281x\Labs\Lab12.

3. Add the source code file to your project:

- **Lab12.c**

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

- **DSP281x_GlobalVariableDefs.c**

5. This is the first exercise in which we are using the DSP/BIOS. This requires a new linker command file to be added to our project. The new command-file, also provided by Texas Instruments with the peripheral header files, excludes the PieVectorTable structure from the project.

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

- **F2812_Headers_BIOS.cmd**

6. The DSP/BIOS itself will include the C environment library files into the project. There is no need to add the file “rts2800_ml.lib” manually.
7. Next we have to create a DSP/BIOS Configuration File (*.cdb). Select:

File → New → DSP/BIOS Configuration

From the next window select the template “c28xx.cdb” to begin with. Save the new configuration file as “lab12.cdb”.

8. Add the configuration file to your project:

Project → Add Files to Project → lab12.cdb

9. The DSP/BIOS configuration has automatically generated a new linker command file “lab12cfg.cmd”, which must be added to our project manually:

Project → Add Files to Project → lab12cfg.cmd

Project Build Options

10. 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**

Close the Build Options Menu by Clicking <OK>.

NOTE: Do NOT setup the stack size inside the Build Options! It will be done with the help of the DSP/BIOS configuration tool later.

Modify Source Code

11. Open Lab12.c to edit.

Disable the Watchdog Timer:

In function “InitSystem” make sure to disable the watchdog timer (register WDCR). For the first DSP/BIOS – Test we would like to keep the code as simple as possible. Later, if the code works, we can easily expand our test setup.

Delete function “delay_loop”:

Recall, in lab2 we used a software delay function for the interval between the LED-output instructions. For lab12, all timing will be done by DSP/BIOS.

Delete the endless “while(1)” construct in “main”:

At the end of “main” DSP/BIOS will take care about the background activities and the timing of all active tasks. If we do not return from main, DSP/BIOS will not start.

Add a new function “led_runner()” at the end of your code:

This new function will be called by DSP/BIOS periodically. The code inside this function should copy the next value from array LED[8] to the LED-lines.

Remove variable “i” and “LED[8]” from “main” and add them at the beginning of this new function. Declare them as “static” and initialize “i” to 0.

Increment “i” with each call of function “led_runner” and load the next value out of “LED[8]” into register “GpioDataRegs.GPBDAT.all”.

Do not forget to reset i to 0 after the end of the sequence was reached!

Edit DSP/BIOS Configuration

12. Open File “lab12.cdb”.

“MEM – Memory Section Manager”

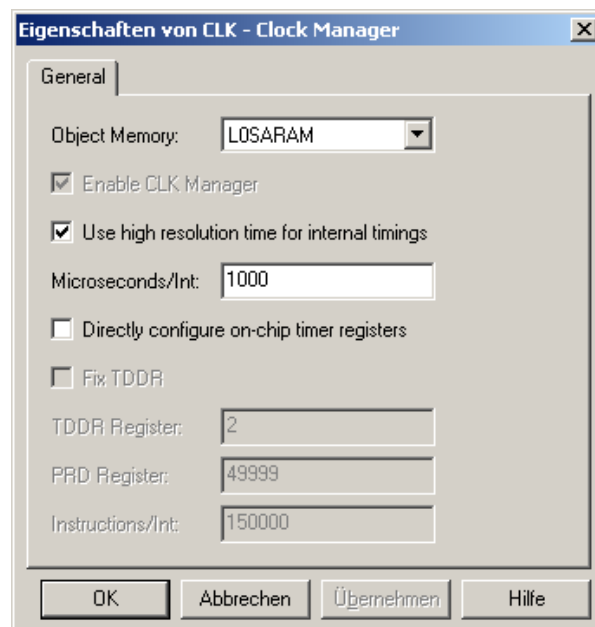
Inspect the “System” Category by expanding the ‘+’ sign in front. Click right on “MEM – Memory Section Manager” and select “Properties”. Verify that in the “General”- Folder the Stack Size has already been defined as 0x0200. This is a default value for the stack size that was initialized when we created the configuration file. If necessary for larger project, this size can be adjusted here. For now, just keep the default value.

Next, move to the “Compiler Sections” tab. This table controls the connection of code sections to physical memory in an identical way, to which we used in our handmade linker command files. When we edit the configuration, this table will be used by CCS to auto-generate the linker command file “lab12cfg.cmd”, which is already part of our project. For now, just keep the table with its default values.

Close the “MEM” – menu by clicking <OK>.

“CLK” – Clock Manager

Right click on “CLK” and select “Properties”. Enable “Use high resolution time for internal timings” and set the number of microseconds per interrupt to 1000.

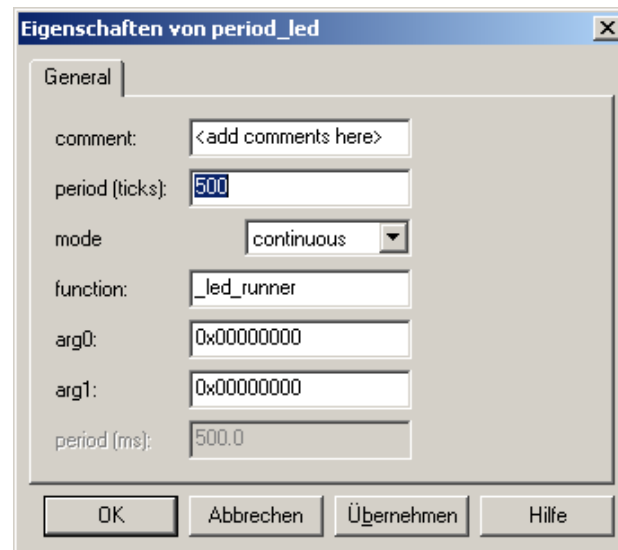


“PRD – Periodic Function Manager”

Expand the “Scheduling” Category. For lab12 we would like to add a periodic function to our project. Click right on “PRD – Periodic Function Manager” and select “Insert PRD”. A function “PRD0” will be created.

Rename “PRD0” to “LED_Line”.

Next, click right on “LED_Line”, select “Properties” and edit the general properties of this object. Edit the number of ticks from 65535 to 500. This is the period of the function call in milliseconds. In field “function” enter “_led_runner”. Do not forget the leading underscore!



Build the project

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

Project → Build . If build was successful you’ll get:

Build Complete, 0 Errors, 0 Warnings, 0 Remarks.

Test the code

14. Test the project with the debugger by hitting <F5>. The LED - line should show the next value after 500ms. The amazing thing is that our code never calls this function; it is called by the scheduler-task of DSP/BIOS every 500ms.
15. Let’s have a final look into the source code. The main function consists of only 2 function calls:

```
void main(void)
{
    InitSystem();
    Gpio_select();
}
```

After returning from “Gpio_select()” main itself terminates! For a standard embedded systems program this would be most strange because leaving “main” would definitely cause a system crash.

In case of lab12 with DSP/BIOS supporting a background task, the DSP/BIOS-scheduler, takes care of further activities.

16. You can experiment with different time intervals for the call of function “led_runner” by changing the period property of PRD-function “LED_Line” in the DSP/BIOS configuration file “Lab12.cdb”.

Potential Solution for function “led_runner”:

```
void led_runner(void)
{
    static unsigned int i=0;
    static unsigned int LED[8]= {0x0001,0x0002,0x0004,0x0008,
                                0x0010,0x0020,0x0040,0x0080};

    if(i<7) GpioDataRegs.GPBDAT.all = LED[i];
    else GpioDataRegs.GPBDAT.all = LED[14-i];
    if (i++ > 13) i=0;
}
```

C28x Boot ROM

Introduction

In chapter 10 we already discussed the option to start our embedded control program directly from the C28x internal Flash memory. We also looked briefly into other options for starting the code execution. We saw that it is also possible to start from H0 – SARAM, OTP and that we can select a ‘boot load’ operating mode that engages a serial or parallel download of the control code before it is actually executed.

In module 13 we will have a closer look into what is going on in these different modes and into the sequence of activities that is performed by the C28x boot firmware before your very own first instruction is touched. This chapter will help you to understand the start-up procedures of the C28x and the power-on problems of an embedded system in general.

We start with a summary of the six options to start the C28x out of RESET, followed by a look into the firmware structure inside the C28x Boot-ROM. This includes some lookup tables for mathematical operations, a generic interrupt vector table and the code that is used to select one of the six start options.

Because we have already dealt with the Flash start option in chapter 10, we can now focus on the serial boot loader options. Two options are available: Serial Communication Interface (SCI) and Serial Peripheral Interface (SPI). Both interfaces were discussed in detail in Modules 7 and 8. If you have finished the lab exercises of these two modules successfully, you should be able to develop your own code to download code from a PC as host into the SARAM of the C28x and start it from there.

A typical application for the serial download of new code into the C28x is a field update of the internal Flash memory that contains the control code for the embedded system. It would be much too expensive to use the JTAG-Emulator to download the new code. Instead, Texas Instruments offers a Flash API that uses exactly the same SCI boot load option to transmit the new code and/or data into the C28x. This API - a portion of code that will be part of your project will take care of the code update. For more details refer to “TMS320F2810, TMS320F2811 and TMS320F2812 Flash API v1.00”, document number: SPRC125 on TI’s website.

Another typical application is the use of the SPI boot load option. In this case, an external serial SPI-EEPROM or Flash holds the actual code. Before it is executed on the C28x, it is downloaded into the C28x. This is a useful option for the ROM version of the C28x or for an R28x, which do not have any internal non-volatile memory at all.

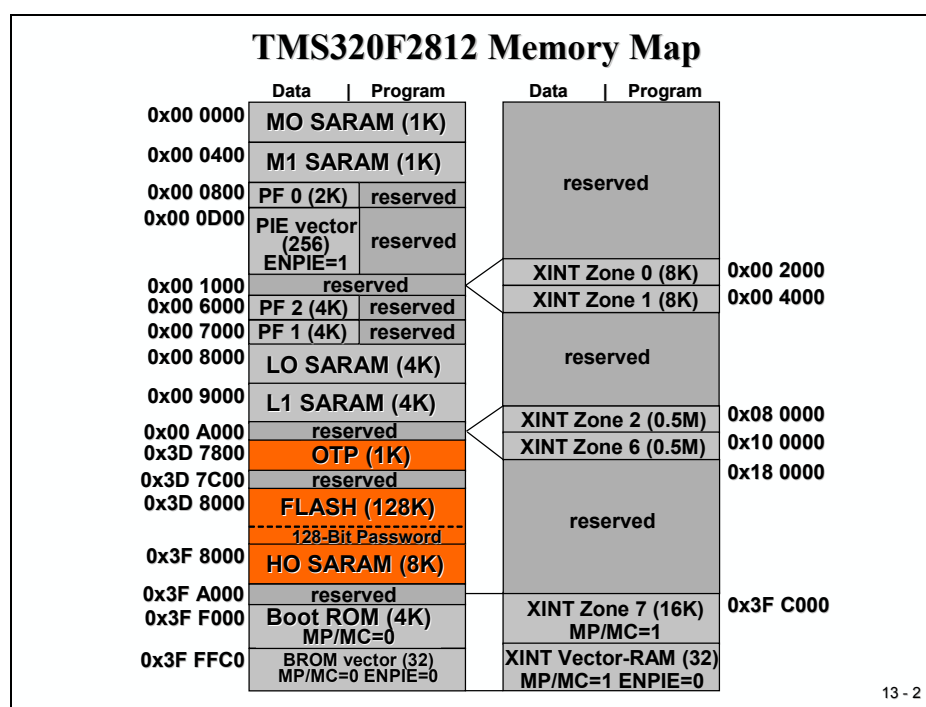
Finally, we will discuss a parallel boot load option that uses the GPIO port B to download code and/or data into the C28x.

Module Topics

C28x Boot ROM	13-1
<i>Introduction</i>	<i>13-1</i>
<i>Module Topics.....</i>	<i>13-2</i>
<i>C28x Memory Map</i>	<i>13-3</i>
<i>C28x Reset Boot Loader</i>	<i>13-4</i>
Timeline for Boot Load:	13-5
<i>Boot – ROM Memory Map.....</i>	<i>13-6</i>
SINE / COSINE Lookup Table	13-6
Normalized Square Root Table	13-8
Normalized ArcTan Table	13-8
Rounding and Saturation Table	13-8
Boot Loader Code.....	13-8
C28x Vector Table	13-9
<i>Boot Loader Data Stream</i>	<i>13-10</i>
Boot Loader Data Stream Example	13-11
Boot Loader Transfer Function	13-12
<i>Init Boot Assembly Function</i>	<i>13-13</i>
<i>SCI Boot Load.....</i>	<i>13-14</i>
SCI Hardware Connection.....	13-14
SCI Boot Loader Function.....	13-15
<i>Parallel Boot Loader</i>	<i>13-16</i>
Hardware Connection	13-16
C28x Software Flow	13-17
Host Software Flow	13-18
<i>SPI Boot Loader.....</i>	<i>13-19</i>
SPI Boot Loader Data Stream.....	13-20
SPI Boot Loader Flowchart	13-20

C28x Memory Map

To begin with, let us recall the C28x memory map. We have a choice of starting our program from Flash, OTP and H0-SARAM, as highlighted in the slide:



We have six different options to start the C28x out of power-on. The options are hard-coded by 4 GPIO-Inputs of Port F (F4, F12, F3 and F2). The 4 pins are sampled during power-on; depending on the status one of the following options is selected:

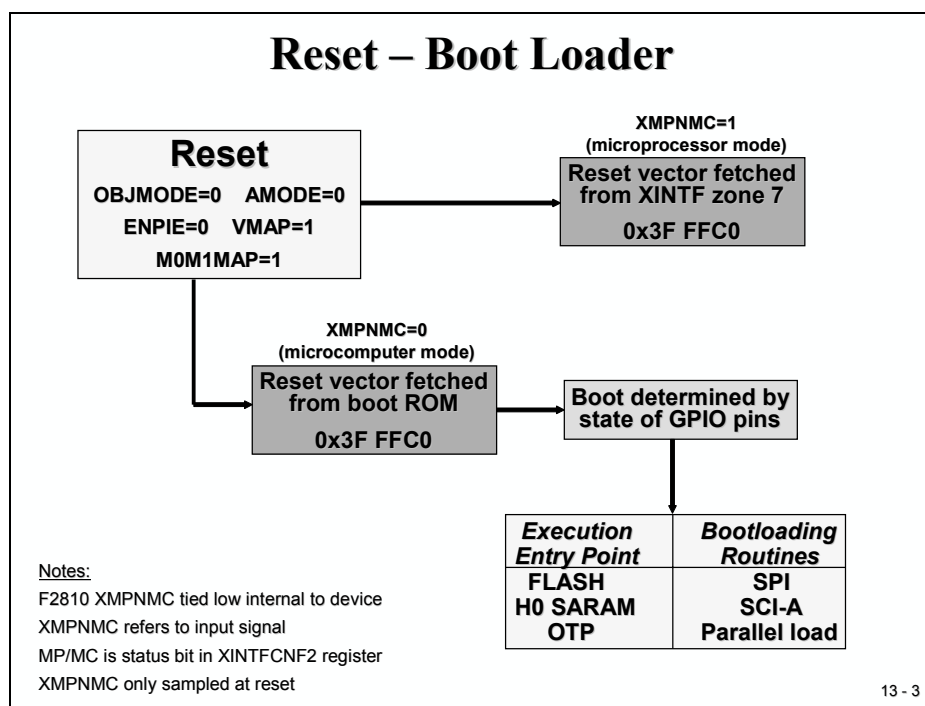
F4	F12	F3	F2	
1	x	x	x	: FLASH address 0x3F 7FF6 (see slide 10-2)
0	0	1	0	: H0 – SARAM address 0x3F 8000
0	0	0	1	: OTP address 0x3D 7800
0	1	x	x	: boot load from SPI
0	0	1	1	: boot load from SCI-A
0	0	0	0	: boot load from parallel GPIO – Port B

The F2812eZdsp controls the four lines F2, F3, F4 and F12 by four jumpers: JP7 (F4), JP8 (F12), JP11 (F3) and JP12 (F2). A ‘1’ in the table above is coded as 1-2 and a ‘0’ as 2-3 jumper set.

Jumper JP1 selects “Microcomputer-Mode” (2-3) or “Microprocessor-Mode” (1-2).

C28x Reset Boot Loader

The next two slides summarize the RESET options of the C28x.



Boot Loader Options

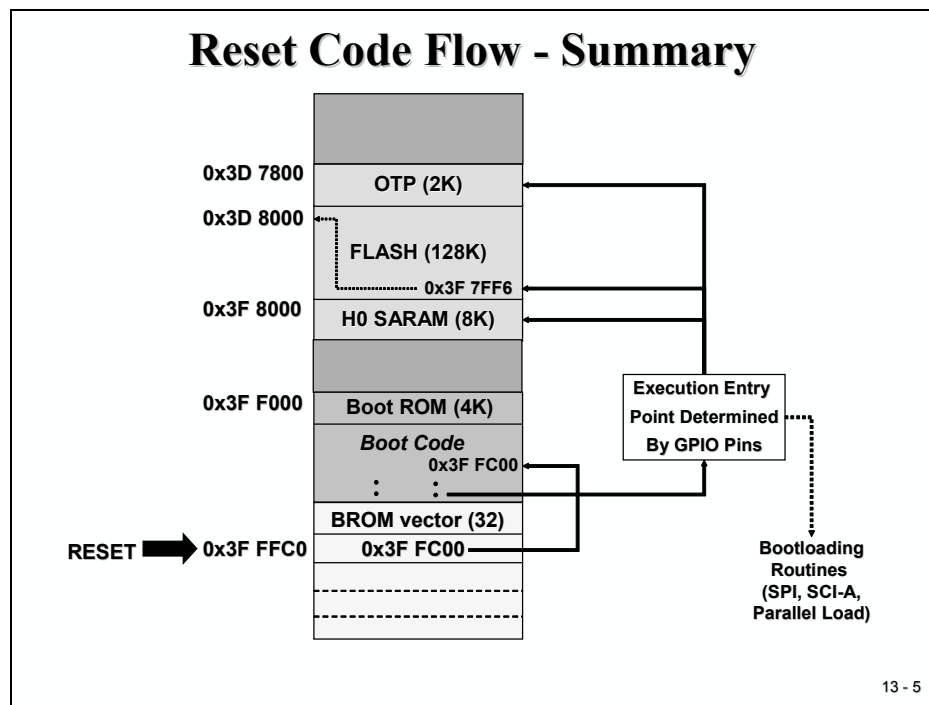
GPIO pins				
F4	F12	F3	F2	
1	x	x	x	jump to <i>FLASH</i> address 0x3F 7FF6 *
0	0	1	0	jump to <i>H0 SARAM</i> address 0x3F 8000 *
0	0	0	1	jump to <i>OTP</i> address 0x3D 7800 *
0	1	x	x	bootload external EEPROM to on-chip memory via <i>SPI</i> port
0	0	1	1	bootload code to on-chip memory via <i>SCI-A</i>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

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

13 - 4

Timeline for Boot Load:

1. RESET-address is always 0x3F FFC0. This is part of TI's internal BOOT-ROM.
2. BOOT-ROM executes a jump to address 0x3F FC00 (the Boot Code). Here basic initialization tasks are performed and the type of the boot sequence is selected.
3. Next, still as part of the Boot Code, the execution entry point is determined by the status of the four GPIO-pins.
4. If one of the three boot loading options is selected, another dedicated part of the Boot Code is executed to establish a standard communication path for SCI, SPI or parallel port B. We will have a closer look into the three options in later slides.



Boot – ROM Memory Map

Before we go into the boot load options let us have a closer look into the partitioning of the boot-ROM area. The size of the area is 4K x 16bit and it is mapped both into code and data memory, using a unified memory map.

TMS320F2812 BOOT-ROM Memory Map

Address Range	Data & Program Space
0x3F F000 – 0x3F F501	SIN/COS; 641 x 32(Q30)
0x3F F502 – 0x3F F711	Normal. Inverse; 264 x 32(Q29)
0x3F F712 – 0x3F F833	Normal. Sqrt; 145 x 32(Q30)
0x3F F834 – 0x3F F9E7	Normal. Arctan; 218 x 32(Q30)
0x3F F9E8 – 0x3F FB4F	Round/Sat. 180 x 32(Q30)
0x3F FB50 – 0x3F FBFF	reserved
0x3F FC00 – 0x3F FFBF	Bootloader ; 960 x 16
0x3F FFC0 – 0x3F FFC1	RESET – Vector; 2 x 16
0x3F FFC2 – 0x3F FFFF	Int. Vectors; 62 x 16

13 - 6

You can look at this memory with Code Composer Studio, providing you have started your eZdsp-board in “Microcomputer-Mode”:

→ View → Memory

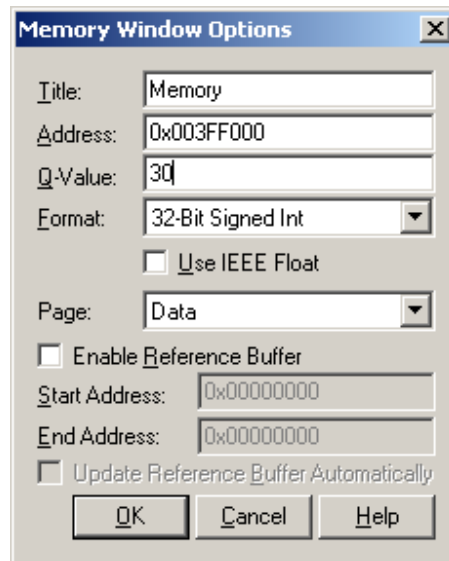
SINE / COSINE Lookup Table

Let's begin with the first 1282 addresses (0x3F F000 to 0x3F F501). This area includes a SIN/COS – Lookup-table and consists of 641 32bit-numbers. The first 512 numbers are for a 360-degrees unit circle with an increment angle of $360/512 = 0.703$ degree. The remaining 128 elements repeat the first 90-degree angle.

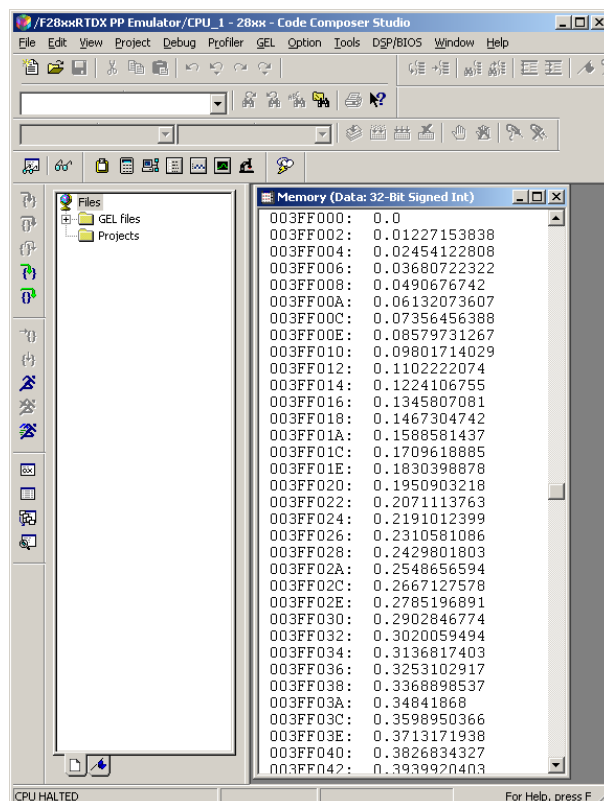
To visualize the SIN/COS -values setup the memory window properties like this:

→ View → Memory

Memory Window Options:



Numbers are in “IQ-Format” with 2 Integer and 30 Fractional Bits. CCS uses the binary content of the memory to display it in the correct format:



Compare: $\sin(1 * 360/512) = 0.012271538285719926079408261951003$

$\sin(2 * 360/512) = 0.024541228522912288031734529459283$

Normalized Inverse Table

The next section of the Boot-ROM includes a lookup table for the Newton-Raphson inverse algorithm. It spans 528 addresses (0x3F F502 to 0x3F F711) and covers 264 32-bit numbers in IQ29-Format.

Normalized Square Root Table

From address 0x3F F712 to 0x3F F833 145 32-bit numbers are stored as a lookup table for estimates of the Newton-Raphson square root algorithm. Data format is IQ30.

Normalized ArcTan Table

A lookup table for the iterative estimation of the Normalized Arc Tangent follows from 0x3F F834 to 0x3F F9E7 in IQ30-format.

Rounding and Saturation Table

Finally memory area 0x3F F9E8 to 0x3F FB4F is used for rounding and saturation subroutines of Texas Instrument library function, like IQ-math or digital motor control libraries (dmclib). The format is also of IQ30.

Boot Loader Code

The last (1K – 64) of memory addresses is used for the Boot Loader Code. When the C28x is coming out of RESET and is running in “Microcomputer Mode” this portion of code will be executed first. As mentioned earlier it derives the actual entry point or the boot loader option from the status of four input pins.

C28x Vector Table

The very last 64 addresses are reserved for 32 Entries of 32-bit address information for interrupt service routine entry points. The layout is shown at the following slide. Each interrupt core line is hard linked to its individual entry in this memory area. In the case where an interrupt is acknowledged by the C28x the assigned 32-bit-information (shown in the next slide as “Content”) is used as entry point for the dedicated interrupt service routine. Because we can’t change the content of this TI-ROM we have to use the fixed entry points in M0-SARAM (0x00 0040 to 0x00 007F) to place a 32-bit assembly branch instruction into our dedicated interrupt service routines. If we come out of RESET, all interrupts are disabled, so we don’t have to do anything. If we decide to use interrupts, which is a wise decision for embedded control, we can use M0-SARAM as vector table – or – we use the Peripheral Interrupt Expansion (PIE) Unit – see Chapter 4.

C28x BOOT-ROM Vector Table

Vector	Address	Content	Vector	Address	Content
RESET	0x3F FFC0	0x3F FC00	RTOSINT	0x3F FFE0	0x00 0060
INT1	0x3F FFC2	0x00 0042	reserved	0x3F FFE2	0x00 0062
INT2	0x3F FFC4	0x00 0044	NMI	0x3F FFE4	0x00 0064
INT3	0x3F FFC6	0x00 0046	ILLEGAL	0x3F FFE6	0x00 0066
INT4	0x3F FFC8	0x00 0048	USER 1	0x3F FFE8	0x00 0068
INT5	0x3F FFCA	0x00 004A	USER 2	0x3F FFEA	0x00 006A
INT6	0x3F FFCC	0x00 004C	USER 3	0x3F FFEC	0x00 006C
INT7	0x3F FFCE	0x00 004E	USER 4	0x3F FFEE	0x00 006E
INT8	0x3F FFD0	0x00 0050	USER 5	0x3F FFF0	0x00 0070
INT9	0x3F FFD2	0x00 0052	USER 6	0x3F FFF2	0x00 0072
INT10	0x3F FFD4	0x00 0054	USER 7	0x3F FFF4	0x00 0074
INT11	0x3F FFD6	0x00 0056	USER 8	0x3F FFF6	0x00 0076
INT12	0x3F FFD8	0x00 0058	USER 9	0x3F FFF8	0x00 0078
INT13	0x3F FFDA	0x00 005A	USER 10	0x3F FFFA	0x00 007A
INT14	0x3F FFDC	0x00 005C	USER 11	0x3F FFFC	0x00 007C
DLOGINT	0x3F FFDE	0x00 005E	USER 12	0x3F FFFE	0x00 007E

13 - 7

Boot Loader Data Stream

The following two slides show the structure of the data stream incoming into the boot loader. The basic structure is the same for all the boot loaders and is based on the C28x hex utility. The tool is called “hex2000.exe (C:\ti\c2000\cgtools\bin)” and is used to convert the project’s out-file, which is in “COFF”-format into the necessary hex-format.

The first 16-bit word in the data stream is known as the key value. The key value is used to tell the boot loader the width of the incoming stream: 8 or 16 bits. Note that not all boot loaders will accept both 8 and 16-bit streams. The SPI boot loader is 8-bit only. Please refer to the detailed information on each loader for the valid data stream width. For an 8-bit data stream, the key value is 0x08AA and for a 16-bit stream it is 0x10AA. If a boot loader receives an invalid key value, then the load is aborted. In this case, the entry point for the Flash memory will be used.

Boot Loader Data Stream Structure

1	0x10AA : Key for memory width = 16 bit
2-9	Reserved for future use
10	Entry Point PC[22:16]
11	Entry Point PC[15:0]
12	Block Size (words); if 0 then end of transmission
13	Destination Address of block ; Addr[31:16]
14	Destination Address of block ; Addr[15:0]
15	First word of block
⋮	
N	Last word of block
N+1	Block Size (words)
N+2	Destination Address of block ; Addr[31:16]
N+3	Destination Address of block ; Addr[15:0]
⋮	

13 - 8

The next eight words are used to initialize register values or otherwise enhance the boot loader by passing values to it. If a boot loader does not use these values then they are reserved for future use and the boot loader simply reads the value and then discards them. Currently, only the SPI boot loader uses one word to initialize registers.

The next 10th and 11th words comprise the 22-bit entry point address. This address is used to initialize the PC after the boot load is complete. This address is most likely the entry point of the program downloaded by the boot loader.

The twelfth word in the data stream is the size of the first data block to be transferred. The size of the block is defined for both 8 and 16-bit data stream formats as the number of 16-bit words in the block. For example, to transfer a block of twenty 8-bit data values from an 8-bit data stream, the block size would be 0x000A to indicate ten 16-bit words.

The next two words tell the loader the destination address of the block of data. Following the size and address will be the 16-bit words that makeup that block of data.

This pattern of block size/destination address repeats for each block of data to be transferred. Once all the blocks have been transferred, a block size of 0x0000 signals to the loader that the transfer is complete. At this point, the loader will return the entry point address to the calling routine, which in turn will cleanup and exit. Execution will then continue at the entry point address as determined by the input data stream contents.

Boot Loader Data Stream Example

Next is an example of a boot loader data stream that is used to load two blocks of data into two different memory locations of the C28x. Five words (1,2,3,4,5) are loaded into address 0x3F 9010 and two words are loaded into address 0x3F 8000.

Boot Loader Data Stream Example

```

10AA    ; Key for 16-Bit memory stream
0000
0000
0000
0000
0000
0000
0000
0000
0000
003F    ; PC - starting point after load is complete: 0x3F 8000
8000
0005    ; 5 words in block 1
003F
9010    ; First block is loaded into 0x3F 9010
0001    ; first data word
0002
0003
0004
0005    ; last data
0002    ; Second block is two words long
003F    ; Second block is loaded into 0x3F 8000
8000
7700    ; first data
7625    ; last data
0000    ; next block zero length = end of transmission

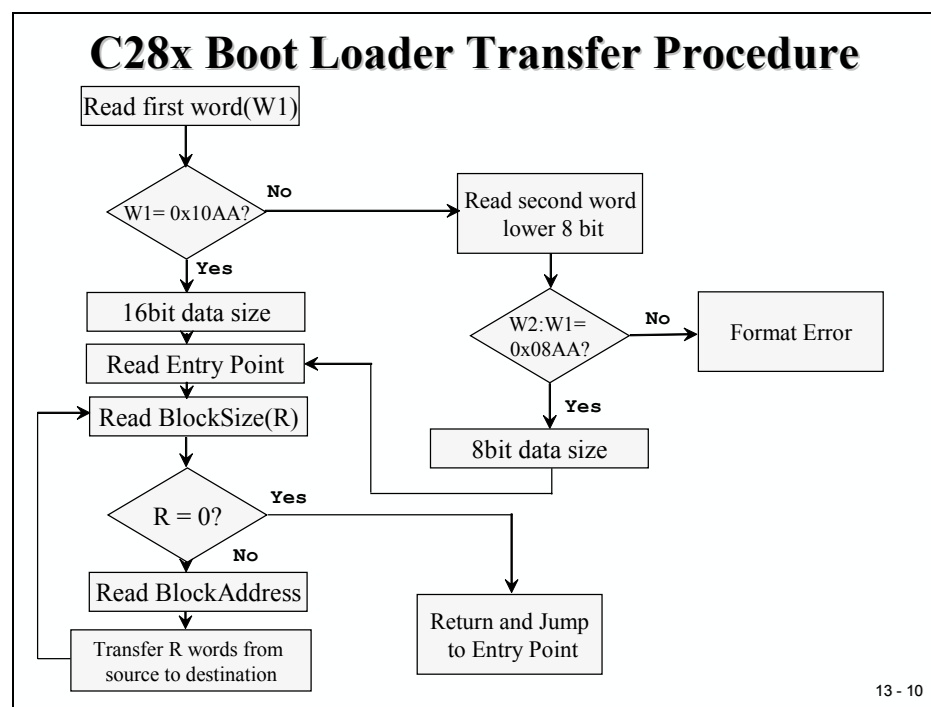
```

13 - 9

Boot Loader Transfer Function

The next flowchart illustrates the basic process a boot loader uses to determine whether 8-bit or 16-bit data stream has been selected, transfer that data, and start program execution. This process occurs after the boot loader finds the valid boot mode selected by the state of the GPIO pins.

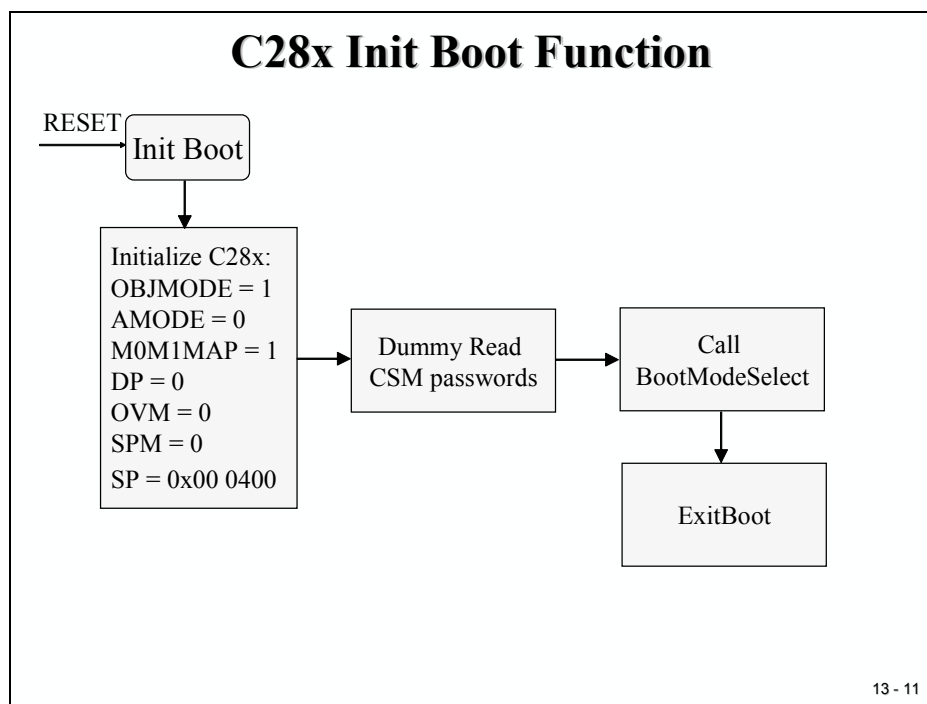
The loader compares the first value sent by the host against the 16-bit key value of 0x10AA. If the value fetched does not match then the loader will read a second value. This value will be combined with the first value to form a word. This will then be checked against the 8-bit key value of 0x08AA. If the loader finds that the header does not match either the 8-bit or 16-bit key value, or if the value is not valid for the given boot mode then the load will abort. In this case the loader will return the entry point address for the flash to the calling routine.



Init Boot Assembly Function

The first routine of the Boot-ROM that is called after RESET is the InitBoot assembly routine. This routine initializes the device for operation in C28x object mode. Next it performs a dummy read of the Code Security Module (CSM) password locations. If the CSM passwords are erased (all 0xFFFFs) then this has the effect of unlocking the CSM. Otherwise, the CSM will remain locked and this dummy read of the password locations will have no effect. This can be useful if you have a new device that you want to boot load.

After the dummy read of the CSM password locations, the InitBoot routine calls the SelectBootMode function. This function will then determine the type of boot mode desired by the state of certain GPIO pins. Once the boot is complete, the SelectBootMode function passes back the EntryAddr to the InitBoot function. InitBoot then calls the ExitBoot routine that then restores CPU registers to their reset state and exits to the EntryAddr that was determined by the boot mode.



SCI Boot Load

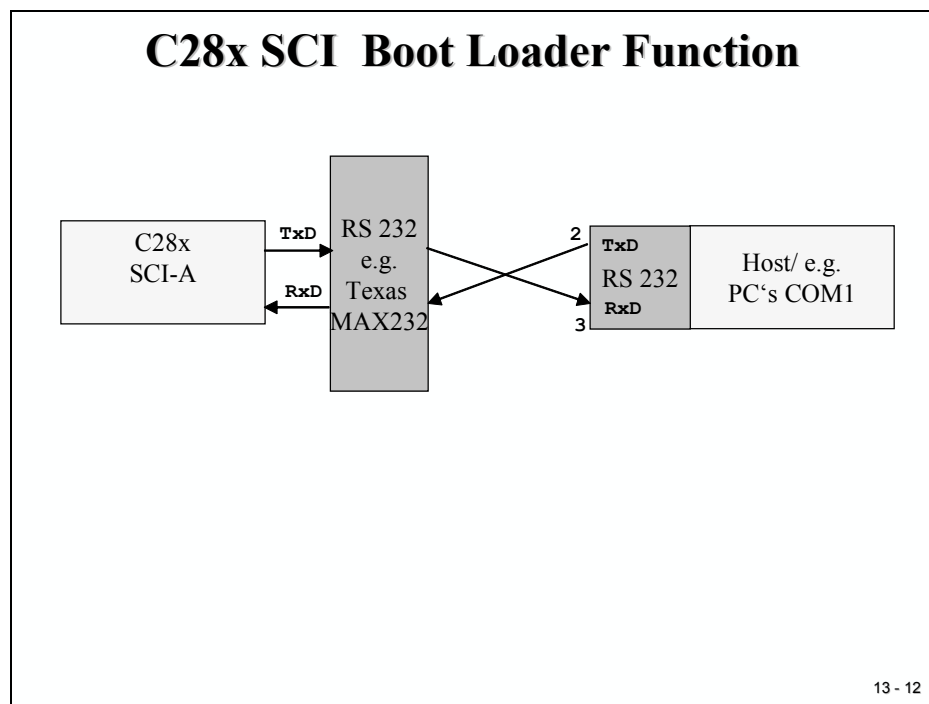
SCI Hardware Connection

The SCI boot mode asynchronously transfers code from SCI-A to the C28x. It only supports an incoming 8-bit data stream and follows the same data flow as outlined before.

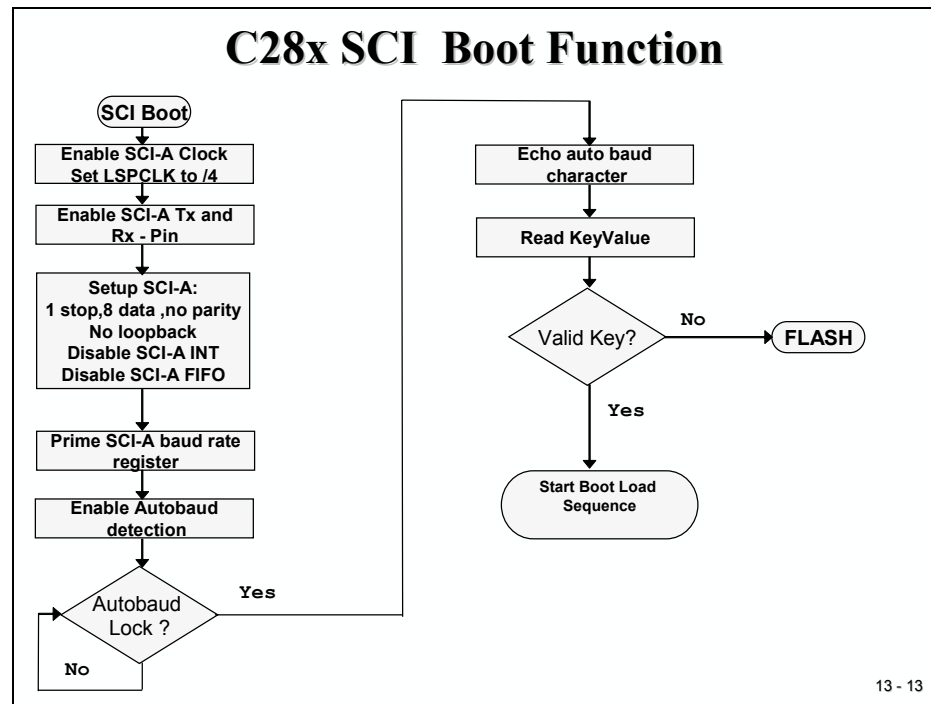
Note:

It is important to understand that, if you want to connect a PC via its serial COM-port to a C28x you will need to have a RS-232 transceiver device in front of the C28x to generate the necessary voltages. If you connect the C28x direct into the 2 PC-COM lines you will eventually destroy the C28x!

The 2812eZdsp does NOT have such a device on the board. The Zwickau adapter board, which was used in Module 8, is equipped with a TI MAX232. Ask your teacher about the actual set up in your laboratory!



SCI Boot Loader Function



The F2810/12 communicates with the external host device by communication through the SCI-A Peripheral. The auto baud feature of the SCI port is used to lock baud rates with the host. For this reason the SCI loader is very flexible and the user can use a number of different baud rates to communicate with the DSP.

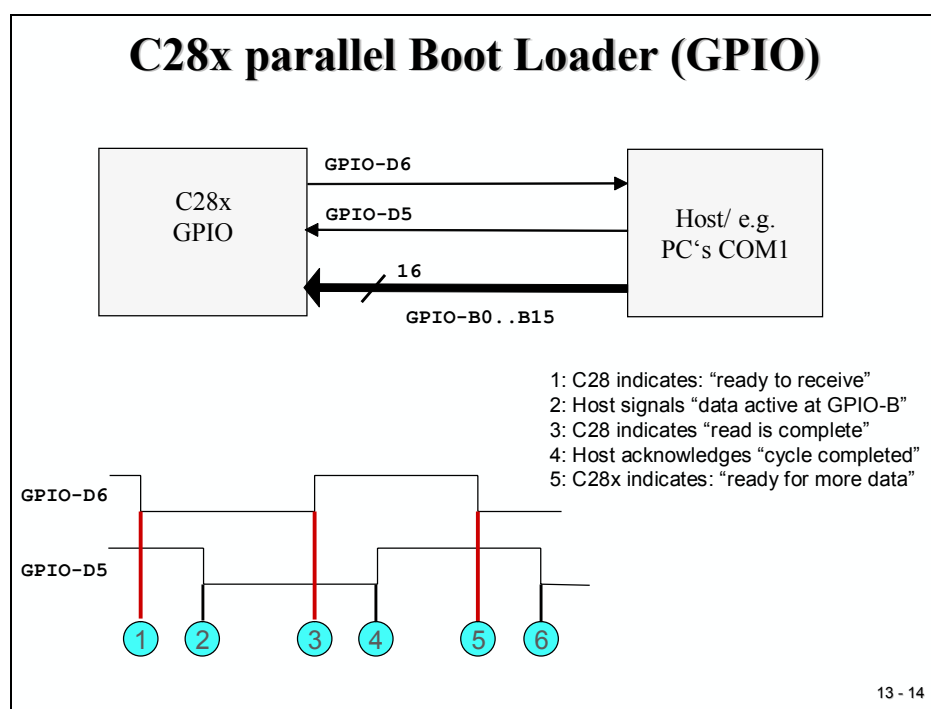
After each data transfer, the DSP will echo back the 8-bit character received to the host. In this manner, the host can perform checks that each character was received by the DSP.

At higher baud rates, the slew rate of the incoming data bits can be affected by transceiver and connector performance. While normal serial communications may work well, this slew rate may limit reliable auto-baud detection at higher baud rates (typically beyond 100 kbaud) and cause the auto-baud lock feature to fail.

Parallel Boot Loader

Hardware Connection

The parallel general purpose I/O (GPIO) boot mode asynchronously transfers code from GPIO port B to internal or XINTF memory. Each value can be 16 bits or 8 bits long and follows the same data flow as outlined in Data Stream Structure.



The F2810/12 communicates with the external host device by polling/driving the GPIOD5 and GPIOD6 lines. The handshake protocol shown above must be used to successfully transfer each word via GPIO port B. This protocol is very robust and allows for a slower or faster host to communicate with the F2810/12 device.

If the 8-bit mode is selected, two consecutive 8-bit words are read to form a single 16-bit word. The most significant byte (MSB) is read first followed by the least significant byte (LSB). In this case, data is read from the lower eight lines of GPIO port B ignoring the higher byte.

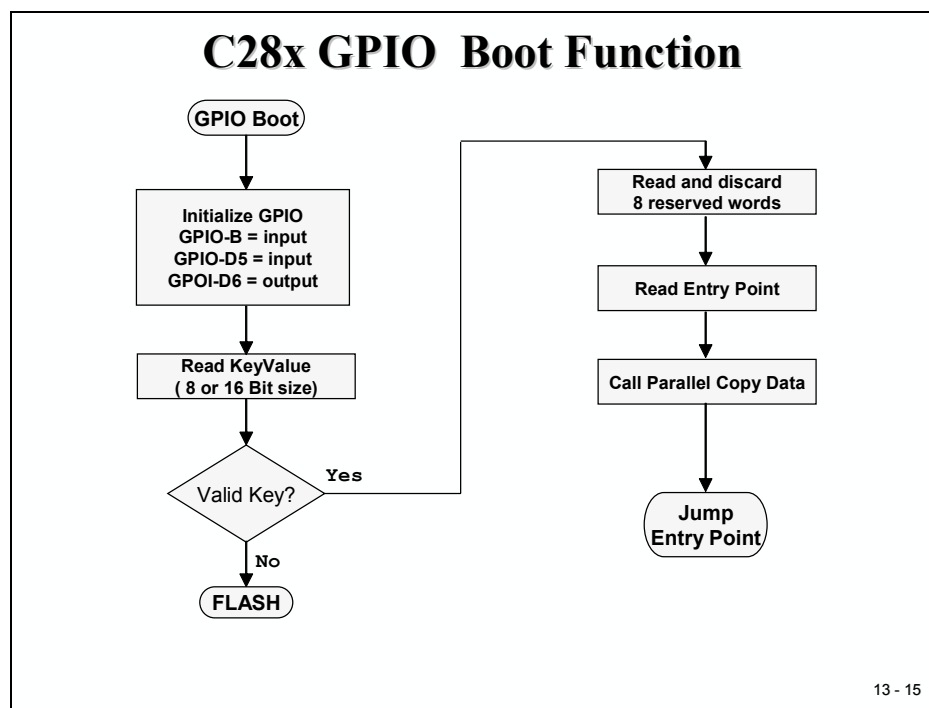
The DSP first signals to the host that the DSP is ready to start a data transfer by pulling the GPIOD6 pin low. The host load then initiates the data transfer by pulling the GPIOD5 pin low. The complete protocol is shown in the slide above.

C28x Software Flow

Slide 13-15 shows a flowchart for the Parallel GPIO boot loader inside the C28x. After parallel boot is selected during RESET, GPIO-port B is initialized as an input port. The two handshake lines GPIO-D5 and D6 are initialized as input and output respectively.

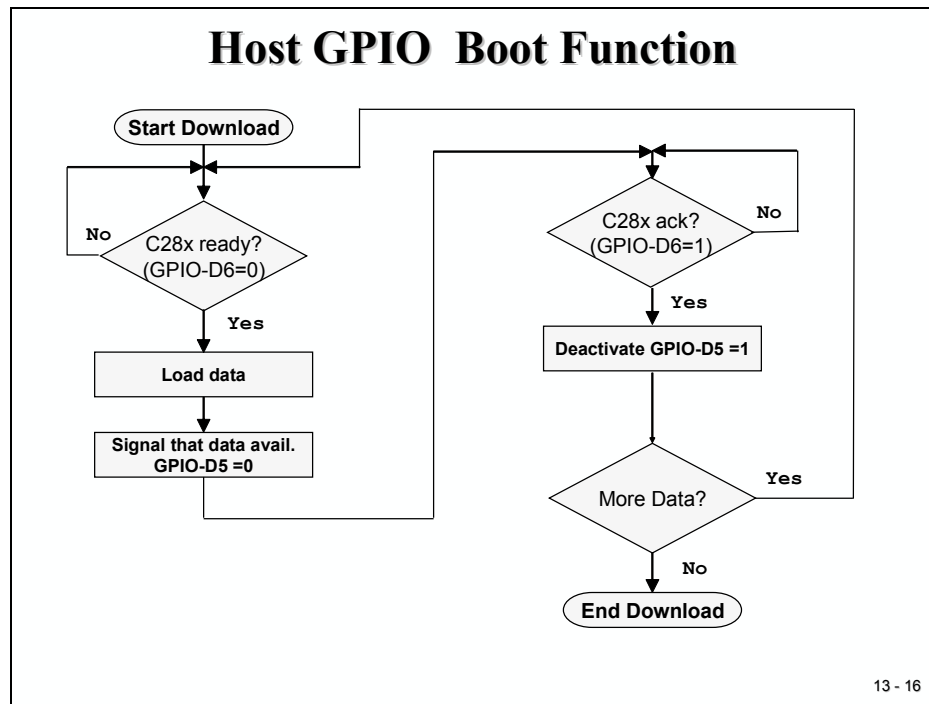
Next, the first character is polled from GPIO-port B. If it was a valid 8- (0x08AA) or 16-bit (0x10AA) key, the procedure continues to read eight more reserved words and discards them. Next, the code entry point and all following blocks are polled according to the diagram at slide 13-14.

If all blocks are received successfully, the routine jumps to the entry point address that was received during the boot load sequence.



Host Software Flow

Slide 13-16 shows the transfer flow from the Host side. The operating speed of the C28x and Host are not critical in this mode as the host will wait for the C28x and the C28x will in turn wait for the host. In this manner the protocol will work with both a host running faster and a host running slower than the C28x.



First, the host waits for a handshake signal (GPIO-D6) to be activated (= 0) by the C28x.

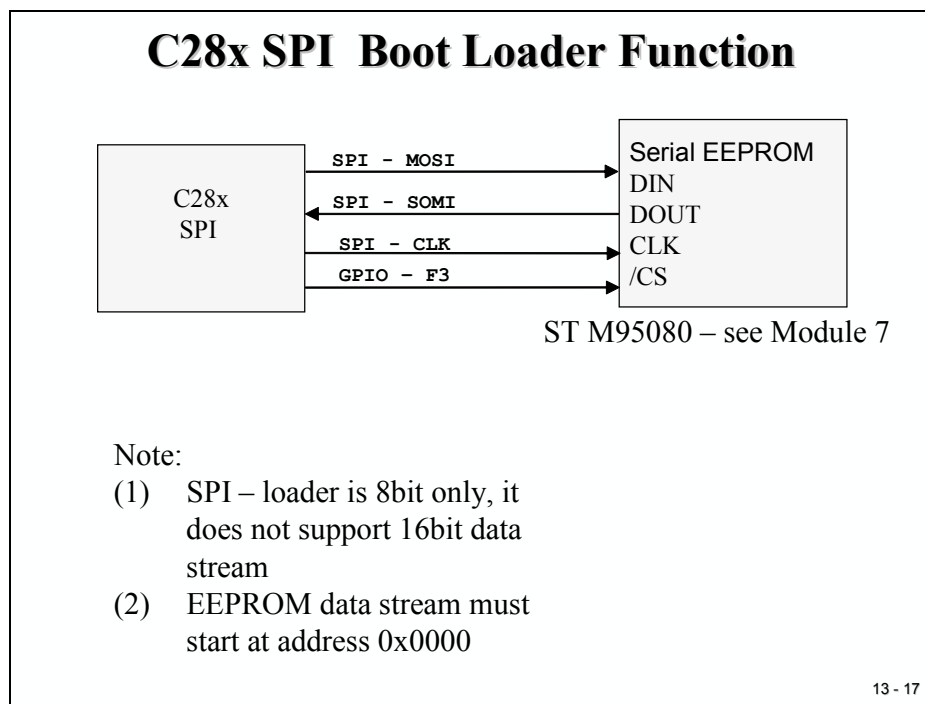
Next, the host has to load the next character onto its parallel output port. A valid character is then acknowledged by the host by activating (=0) a signal that is connected to the C28x GPIO-D5 input line.

The C28x has now all the time it requires to read the data from GPIO-port B. Once this is done, the C28x deactivates its output line GPIO-D6 to inform the host that the transfer cycle is completed.

The host acknowledges this situation by deactivating its handshake line (D5). If the algorithm has more data to transmit to the C28x, the procedure is repeated once more. If not, the download is finished.

SPI Boot Loader

The SPI loader expects an 8-bit wide SPI-compatible serial EEPROM device to be present on the SPI pins as indicated in Figure 21. The SPI boot loader does not support a 16-bit data stream.



The SPI boot ROM loader initializes the SPI module to interface to a serial SPI EEPROM. Devices of this type include, but are not limited to, the Microchip M95080 (1K x 8), the Xicor X25320 (4Kx8) and Xicor X25256 (32Kx8). The Zwickau adapter board is equipped with a M95080, which can be used to experiment with the SPI boot load mode.

The SPI boot ROM loader initializes the SPI to the following settings: FIFO enabled, 8-bit character, internal SPICLK master mode and talk mode, clock phase = 0, polarity = 0 and slowest baud rate.

If the download is to be preformed from an SPI port on another device, then that device must be setup to operate in the slave mode and mimic a serial SPI EEPROM. Immediately after entering the SPI Boot function, the pin functions for the SPI pins are set to primary and the SPI is initialized. The initialization is done at the slowest speed possible. Once the SPI is initialized and the key value read, the user could specify a change in baud rate or low speed peripheral clock.

SPI Boot Loader Data Stream

The following slide shows the sequence of 8-bit data expected by the Boot Loader.

C28x SPI Boot Loader Data Stream	
Byte	Content
1	LSB = 0xAA (Key for 8bit transfer)
2	MSB = 0x08 (Key for 8bit transfer)
3	LSB = LSPCLK value
4	MSB = SPIBRR value
5-18	reserved
19	Entry Point [23:16]
20	Entry Point [31:24]
21	Entry Point [7:0]
22	Entry Point [15:8]
23 ...	Blocks of data: block size/destination/data as shown

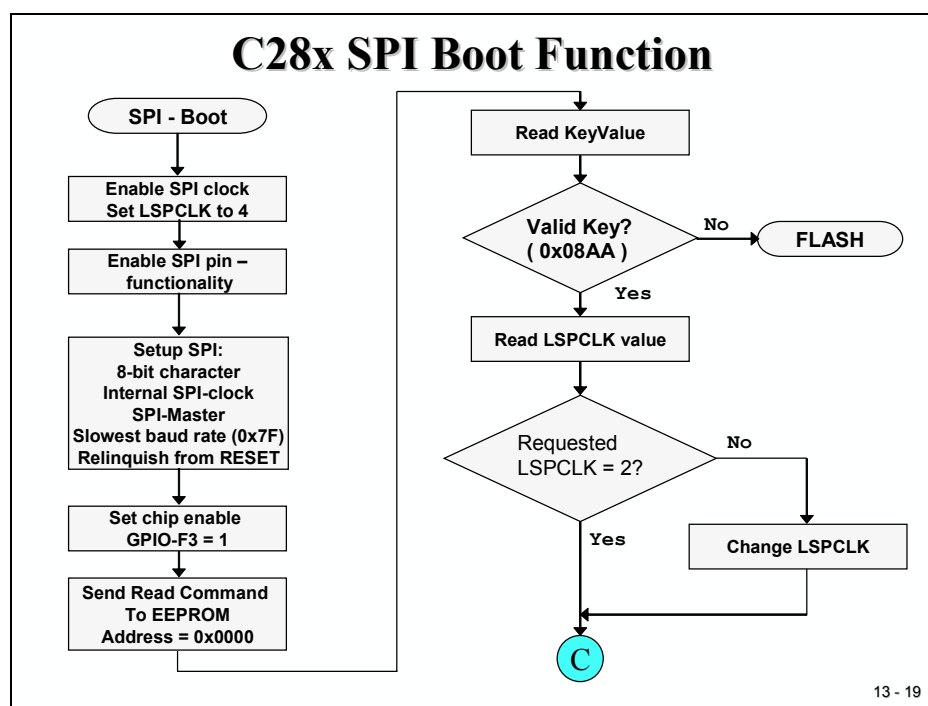
13 - 18

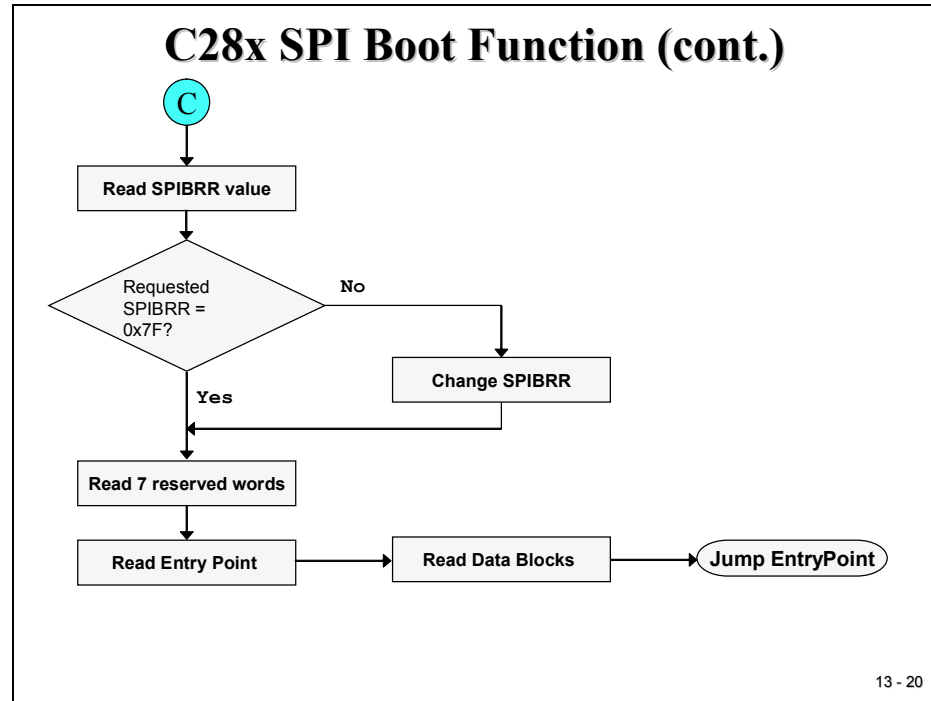
SPI Boot Loader Flowchart

The chart is shown on the next page. The data transfer is done in “burst” mode from the serial SPI EEPROM. The transfer is carried out entirely in byte mode (SPI at 8 bits/character). A step-by step description of the sequence follows:

- 1) The SPI-A port is initialized
- 2) The GPIOF3 pin is now used as a chip-select for the serial SPI EEPROM
- 3) The SPI-A outputs a read command to the serial SPI EEPROM

- 4) The SPI-A sends the serial SPI EEPROM address 0x0000; that is, the host requires that the EEPROM must have the downloadable packet starting at address 0x0000 in the EEPROM.
- 5) The next word fetched must match the key value for an 8-bit data stream (0x08AA). The most significant byte of this word is the byte read first and the least significant byte is the next byte fetched. This is true of all word transfers on the SPI. If the key value does not match then the load is aborted and the entry point for the Flash (0x3F 7FF6) is returned to the calling routine.
- 6) The next two bytes fetched can be used to change the value of the low speed peripheral clock register (LOSPCP) and the SPI Baud rate register (SPIBRR). The first byte read is the LOSPCP value and the second byte read is the SPIBRR value. The next seven words are reserved for future enhancements. The SPI boot loader reads these seven words and discards them.
- 7) The next two words makeup the 32-bit entry point address where execution will continue after the boot load process is complete. This is typically the entry point for the program being downloaded through the SPI port.
- 8) Multiple blocks of code and data are then copied into memory from the external serial SPI EEPROM through the SPI port. The blocks of code are organized in the standard data stream structure presented earlier. This is done until a block size of 0x0000 is encountered. At that point in time, the entry point address is returned to the calling routine that then exits the boot loader and resumes execution at the address specified.





Introduction

This chapter looks into one of the most common applications for Digital Signal Processors: “Digital Filters”. As we have seen before, there are two basic classes of computing: “off – line” and “real – time”. This is also valid for digital signal processing. Because we are dealing with the C28x controller, which is designed for embedded control, calculations are usually done in a real time environment. For a digital filter, this means that all internal processing of the current state of a system must be finished before a next input value is sampled. Again, computing time is most precious! The faster we calculate the algorithm of a digital filter, the more samples we can take from an input channel. It also means that the frequency of the input signal that is to be processed depends directly on the efficiency of the controller.

We will start with some mathematical basics of Digital Filters, but we will not go too much into the theoretical background. To learn more about the mathematics behind Digital Filters, you will have to join other courses at your university. The Texas Instruments C6000- Teaching CD-ROM is highly recommended to learn more about the design of digital filters. Beginning with chapter 14 of this CD, you will be introduced to techniques for filter coefficient estimation and to basic approaches of windowing. Although this C6000 CD is based on the C6000 family the chapters about Filters and Fourier Transform are also valid for the C28x.

Next we will look into the structure of “Finite Impulse Response (FIR)”-filters and their properties. Because of their simplicity and stability, these types of filters are often used in digital signal processing. We will calculate some examples for low-pass (LPF) and high-pass (HPF) – FIR-filters before we look into a C implementation of such a filter algorithm for the C28x. The use of IQ-Math C-functions leads to a much faster execution then using standard ANSI-C implementation. As we have already seen, IQ-Math is a unique feature of the C28x, which is based on its internal hardware units.

If the computing speed of the IQ-Math implementation is still not fast enough, we can do better! By switching to Assembly Language coding and by using Texas Instruments Digital Filter Library (sprc082.zip), we can bring the C28x to its top speed. By means of two examples, we will have a look into the implementation of a FIR in Assembly Language and into the usage of the C-callable library function “FIR16”, provided by TI.

Finally we will perform a laboratory experiment using a FIR-Filter. After we generate a 2 kHz square wave signal, we will sample it back into the C28x by means of the internal ADC. We will use the samples to calculate a low-pass function with a 4th order FIR-Filter in real-time. Code Composer Studio’s Graph Tool will be used to visualize the behaviour of both the unfiltered and filtered signal in parallel.

Module Topics

C28x FIR - Filter	14-1
<i>Introduction</i>	<i>14-1</i>
<i>Module Topics.....</i>	<i>14-2</i>
<i>Basics of Digital Filter Theory</i>	<i>14-3</i>
Time Domain Equation	14-3
Frequency Domain Equation	14-5
<i>Finite Impulse Response Filter</i>	<i>14-8</i>
Properties of a FIR - Filter.....	14-9
<i>FIR Examples.....</i>	<i>14-10</i>
<i>FIR Implementation in C.....</i>	<i>14-14</i>
<i>FIR Implementation in Assembly Language</i>	<i>14-16</i>
Circular Addressing Mode.....	14-17
FIR- Filter Code	14-19
<i>Texas Instruments C28x Filter Library.....</i>	<i>14-21</i>
MATLAB Filter Script.....	14-22
FIR16 Library Function.....	14-23
<i>Lab 14: FIR – Filter for a square-wave signal</i>	<i>14-25</i>
Objective	14-25
Procedure.....	14-26
Open Files, Create Project File.....	14-26
Project Build Options	14-27
Modify Source Code.....	14-27
Build and Load	14-29
Test	14-30
Feedback the Signal into ADC	14-31
Set up ADC sample period (Timer 2).....	14-31
Connect T1PWM to ADCIN2	14-32
Build, Load and Test	14-32
Inspect and Visualize the FIR.....	14-33
CCS Graphical Tool	14-34

Basics of Digital Filter Theory

Time Domain Equation

The following equation for a Linear Time-Invariant (LTI) system is the starting point to derive a representation of a Digital Filter:

Basics of Digital Filter Theory

- ◆ Digital Filter Algorithms are probably the most used numerical operations of a Digital Signal Processor
- ◆ Digital Filters are based on the common difference equation for Linear Time-Invariant (LTI) – systems:

$$\sum_{m=0}^{N-1} a_m \cdot y[n-m] = \sum_{k=0}^{N-1} b_k \cdot x[n-k]$$

- ◆ $y(n)$ = output signal
- ◆ $x(n)$ = input signal
- ◆ a_m, b_k = coefficients
- ◆ N = number of coefficients (order of system)
- ◆ Normalized to $a_0 = 1$ we derive the basic equation in time domain:

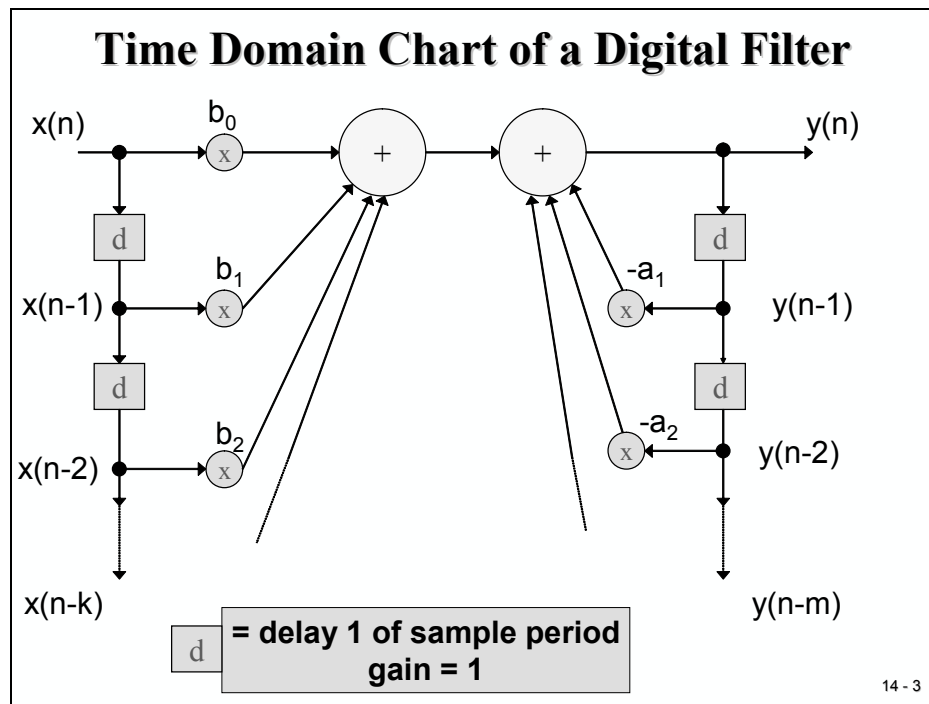
$$y(n) = \sum_{k=0}^{N-1} b_k \cdot x[n-k] - \sum_{m=1}^{N-1} a_m \cdot y[n-m]$$

14 - 2

It is an equation for discrete input ($x(n)$) and output ($y(n)$) signals that are processed by a transforming system. The properties of the transformation are expressed by coefficients (a_m and b_k). Terms like $x[n-k]$ or $y[n-m]$ are used to express the status of the input and output signal k or m times before the current sample time n . For causal systems, all samples before time $t = 0$ are zero.

These types of equations represent the modification of the output signal $y(n)$ on a time base – we call it “Time Domain Representation”. The smallest amount of time that is used in these equations is the sampling period. After the input signal is sampled, the continuous time scale is replaced by a sequence of numbers. According to Shannon’s sampling theorem, the sampling frequency must be at least twice as fast as the highest frequency component of the input signal.

The calculation of the normalized equation for $y(n)$ can be visualized graphically, as shown with the next slide:



This flow can be used to calculate $y(n)$ from the current input sample $x(n)$ and samples of the input signal, taken one ($x(n-1)$), two ($x(n-2)$) or k ($x(n-k)$) samples before. We call this part the “forward” section of the calculation. If we include the status of the output signal delayed by one period ($y(n-1)$), two periods ($y(n-2)$) up to k periods ($y(n-k)$) into the calculation of the new value of $y(n)$, we add a “feedback” section to the computing scheme.

To translate this flowchart into a computer program, we would have to store not only the current input $x(n)$ and output $y(n)$, but also information about their previous states. How do we code this in a programming environment? Usually, with two arrays that are big enough to store all the previous states of x and y . This type of array is usually called a “buffer”. It functions as delay-line, hence the term “Delay-Line-Buffer”.

So what happens when the code has calculated a new value of $y(n)$? Obviously, $y(n)$ must be presented to the outside world as a new result of our calculation. Fine, but what is next? To perform a calculation in real-time, the code must read the next sample from input signal x and store it at buffer position $x(n)$. But $x(n)$ is still occupied by the sample from one period earlier! Before we can store the new sample in $x(n)$, the code must move all entries in array x to the next position, $x(n)$ to $x(n-1)$, $x(n-1)$ to $x(n-2)$ and so on. During this procedure the oldest sample will be discarded. At the output side, the code has to shift all y – values in a similar manner.

Consider the sequence of shift operations! In practice, we have to shift the second oldest first, followed by the next oldest. If not, we fill the entire buffer with $x(n)$!

Frequency Domain Equation

The second interpretation of the behavior of a LTI-system is done in terms of frequency – the “Frequency Domain” - Equation.

The basic operation to transfer a time discrete signal in frequency domain is called “Z-Transformation” (ZT). The transformation follows these rules:

Transfer Function of a Digital Filter

- ◆ The Z-Transform of the original input signal $x(n)$ is defined as:

$$ZT\{x(n)\} = \underline{X}(z) = \sum_{n=0}^{\infty} x(n) \cdot z^{-n}$$

- ◆ with

$$z = e^{pT} \text{ and } p = \sigma + j\omega$$

p = complex angular frequency

- ◆ One property of the Z-Transform is that the ZT of a time-shifted signal is equal to the ZT of the original signal except of a factor z^{-k} :

$$ZT\{x(n-k)\} = z^{-k} \cdot \underline{X}(z)$$

14 - 4

The slide shows how the series of discrete input samples $x(n)$ is converted into a complex series $\underline{X}(z)$. Instead of representing the signal as sequence “number over time” we can represent the signal as sequence “complex number over frequency”.

One important property of the ZT is that the ZT of a time shifted input signal $x(n-k)$ is identical to the ZT of the non-time shifted signal $x(n)$, except for a multiplier z^{-k} . This feature reduces the workload to calculate the complex series for $\underline{X}(z)$ dramatically.

How do we use this ZT to convert the time-domain equation for an LTI-system into its frequency representation? Well, we have to apply the ZT to both sides of the time-domain equation, shown at the next slide:

Transfer Function of a Digital Filter

- ◆ Z-Transform is applied to both sides of the time domain equation of a Digital Filter :

$$ZT\left\{y(n) + \sum_{m=1}^{N-1} a_m \cdot y[n-m]\right\} = ZT\left\{\sum_{k=0}^{N-1} b_k \cdot x[n-k]\right\}$$

$$\underline{Y}(z) + \sum_{m=1}^{N-1} a_m z^{-m} \underline{Y}(z) = \sum_{k=0}^{N-1} b_k z^{-k} \underline{X}(z)$$

$$\underline{Y}(z) \left[1 + \sum_{m=1}^{N-1} a_m z^{-m} \right] = \underline{X}(z) \left[\sum_{k=0}^{N-1} b_k z^{-k} \right]$$

14 - 5

The final equation is called “Transfer Function” of the Digital Filter. It is a frequency domain representation of the influence that is exerted to an input signal by the Digital Filter.

Transfer Function of a Digital Filter

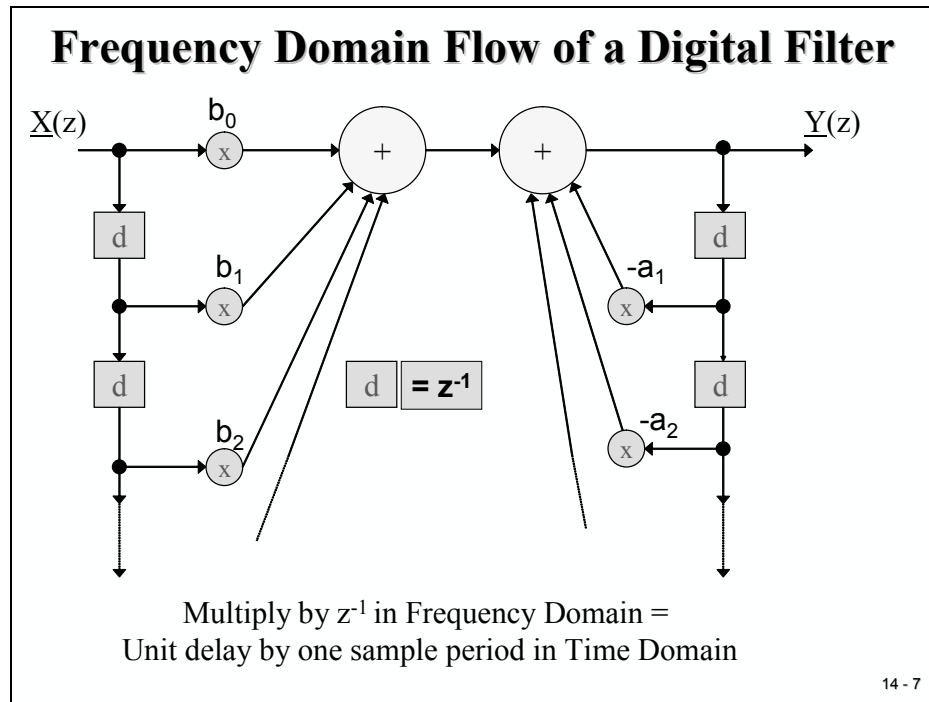
- ◆ Finally we derive the Transfer Function of a Digital Filter of order N in frequency domain:

$$\underline{H}(z) = \frac{\underline{Y}(z)}{\underline{X}(z)} = \frac{\sum_{k=0}^{N-1} b_k z^{-k}}{1 + \sum_{m=1}^{N-1} a_m z^{-m}}$$

14 - 6

For a given spectrum of input frequencies $\underline{X}(z)$ the transfer function defines the shape of the output spectrum $\underline{Y}(z)$. Complex frequency numbers are represented as magnitude and phase per frequency line.

In a similar way as we have seen for the time-domain flow, we can draft a calculation scheme for the transfer function in the frequency domain. Each delay-unit in the time domain is replaced by a multiplication by complex number z^{-1} . The basic principle to calculate a new $\underline{Y}(z)$ is similar to the time-domain flow, except the complex multiplication. The algorithm still needs two arrays to store $\underline{X}(z)$ and $\underline{Y}(z)$, except that they have to now use complex numbers with a real and imaginary part for each number. The Frequency Domain Calculation of the frequency response of a system is normally used to analyse an incoming signal for its frequency components.



Finite Impulse Response Filter

If a Digital Filter does not have any feedback components (all $a_m = 0$), we call this system a “Finite Impulse Response” (FIR). It can be shown that the response of such a system to a single input impulse will eventually vanish.

Finite Impulse Response (FIR) - Filter

- ◆ If all feedback coefficients a_m are equal to zero we derive the equation system for a “Finite Impulse Response (FIR)” – Filter:

$$\underline{H}(z) = \frac{\underline{Y}(z)}{\underline{X}(z)} = \sum_{k=0}^{N-1} b_k z^{-k} \quad \text{Frequency Domain}$$

- ◆ and:

$$y(n) = \sum_{k=0}^{N-1} b_k x[n-k] \quad \text{Time Domain}$$

14 - 8

If feedback components exist, the system is called an “Infinite Response Filter” (IIR).

Infinite Impulse Response (IIR) - Filter

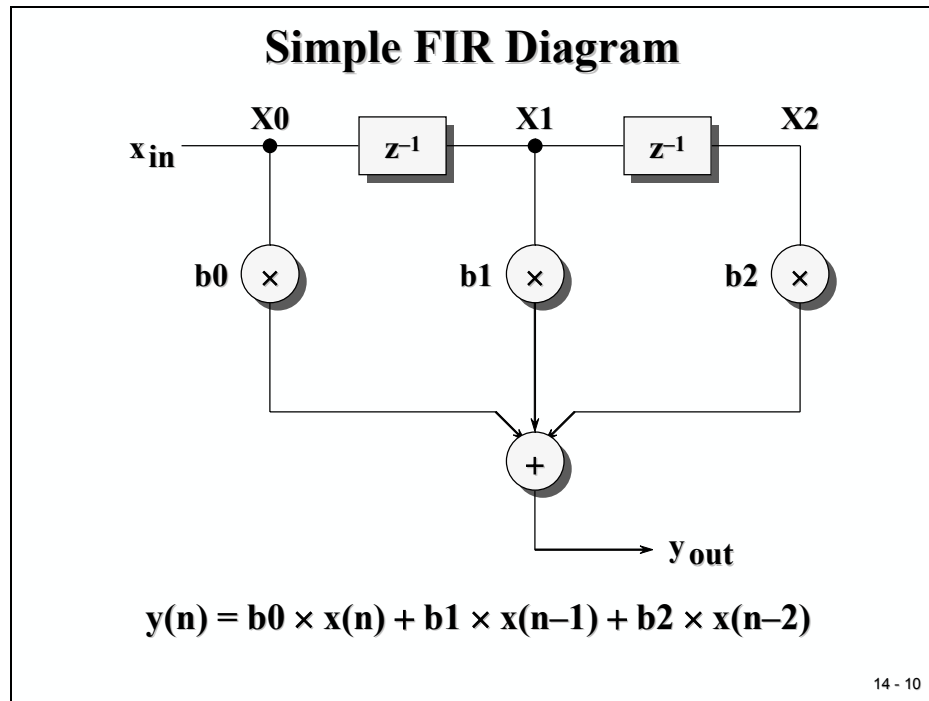
- ◆ If coefficients a_m are present we call this type of filter “Infinite Impulse Response (IIR). In this case the equation with feedback part must be used for the filter calculation.
- ◆ Obviously the “feedback” terms $a_m \cdot y(n-m)$ deliver some amount of energy back into the calculation.
- ◆ Under particular circumstances this feedback system will respond to a finite input impulse infinite in time – hence the name.

$$\underline{H}(z) = \frac{\underline{Y}(z)}{\underline{X}(z)} = \frac{\sum_{k=0}^{N-1} b_k z^{-k}}{1 + \sum_{m=1}^{N-1} a_m z^{-m}} \quad \text{IIR – Filter}$$

$$y(n) = \sum_{k=0}^{N-1} b_k \cdot x[n-k] - \sum_{m=1}^{N-1} a_m \cdot y[n-m]$$

14 - 9

Properties of a FIR - Filter



One typical property of the FIR-Transfer Function is its periodicity by 2π :

Properties of a FIR Filter

- ◆ Replacing z by it's original definition:

$$z = e^{pT} = e^{(\sigma + j\omega)T}$$
 disregarding σ (loss – less filter) and normalizing to $T=1$:

$$\underline{H}(z) \Big|_{z=e^{j\omega}} = \underline{H}(e^{j\omega}) = \sum_{k=0}^{N-1} b_k e^{-jk\omega}$$
- ◆ Since $e^{-j2\pi k} = 1$:

$$\underline{H}(e^{j(\omega+2\pi)}) = \sum_{k=0}^{N-1} b_k e^{-jk(\omega+2\pi)} = \sum_{k=0}^{N-1} b_k e^{-jk\omega} e^{-j2\pi k} = \underline{H}(e^{j\omega})$$
- ◆ FIR filters have a periodic frequency response of 2π !
- ◆ We need to limit the spectrum!

14 - 11

FIR Examples

Let us calculate the frequency response of the following filter system. As the diagram shows it lacks feedback components – it is of FIR-type. It is a first-order filter, the filter coefficients are

- $b_0 = +0.5$
- $b_1 = +0.5$

What will the magnitude of the output signal look like? Which frequency components will pass the filter, which one will be damped?

FIR – Example 1

$b_0 = 0.5 \quad b_1 = 0.5$

- Frequency Response ?
- Type of Filter ?

$$\underline{H}(z) = b_0 z^0 + b_1 z^{-1}$$

$$\underline{H}(z) = 0.5(1 + z^{-1})$$

$$\underline{H}(j\omega) = 0.5(1 + e^{-j2\pi \frac{f}{f_A}})$$

$$\underline{H}(j\omega) = 0.5(1 + \cos(2\pi \frac{f}{f_A}) - j \sin(2\pi \frac{f}{f_A}))$$

$$|\underline{H}(j\omega)| = \sqrt{\text{Re}^2 + \text{Im}^2}$$

$$z = e^{pT}; \quad p = \sigma + j\omega; \quad \omega = 2\pi f; \quad T = \frac{1}{f_A}$$

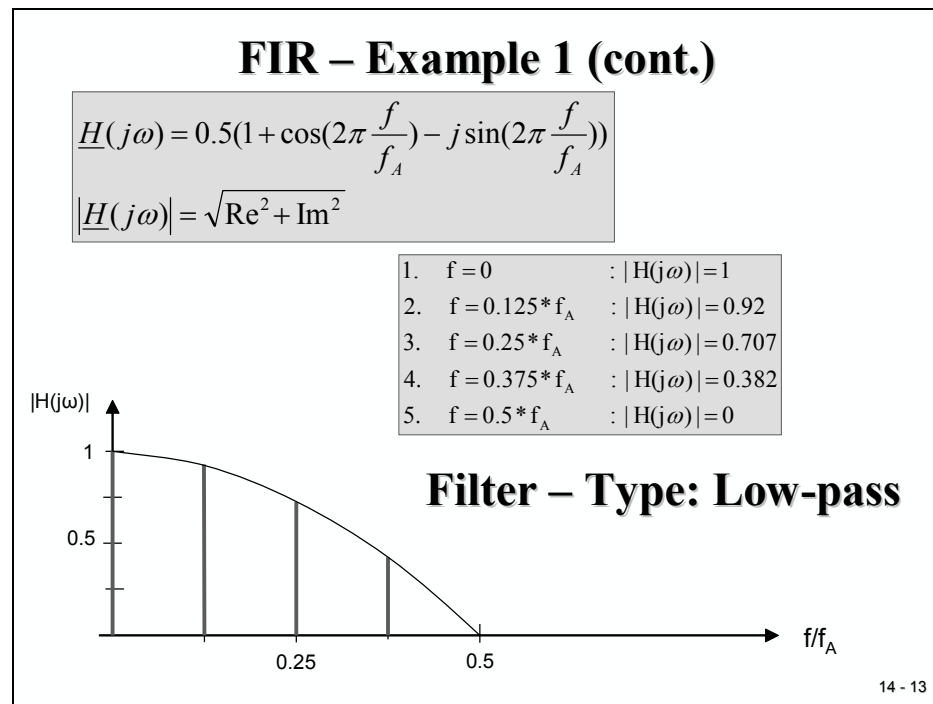
$f_A = \text{sampling frequency}$

14 - 12

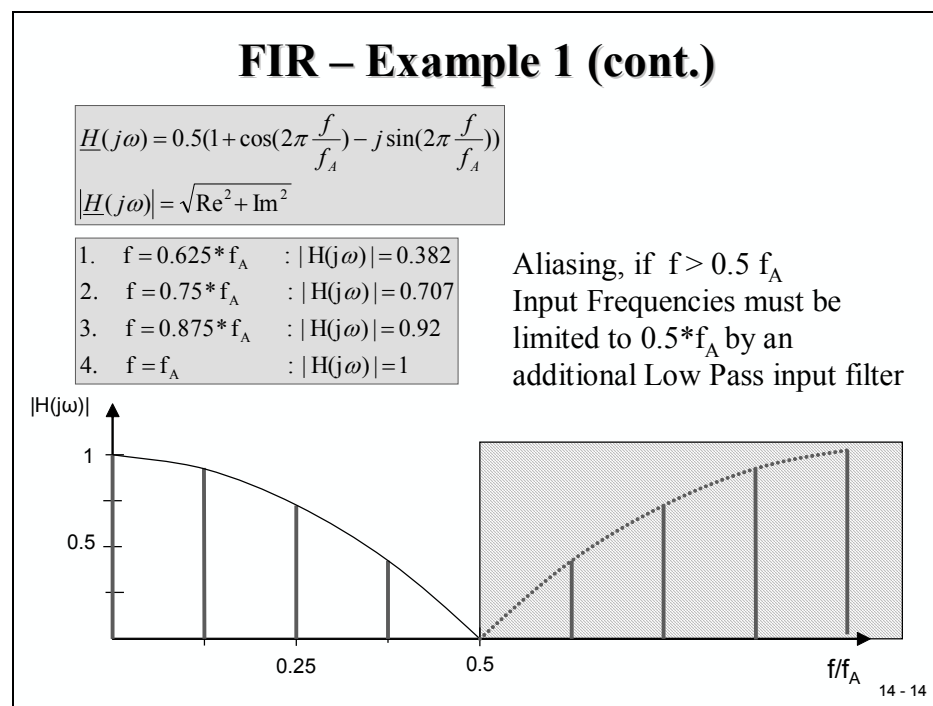
If we calculate the magnitude for frequencies $f = 0$, $f = f_A/8$, $f = f_A/4$, $f = f_A \cdot 3/4$ and $f = f_A/2$ we get:

$$|H(j\omega)| = 1, 0.92, 0.707, 0.382 \text{ and } 0$$

The graph is shown next. Low frequencies are amplified by 1, the more we approach $0.5 \cdot f_A$, the more the magnitude is damped and finally reaches 0. This is a low-pass filter!



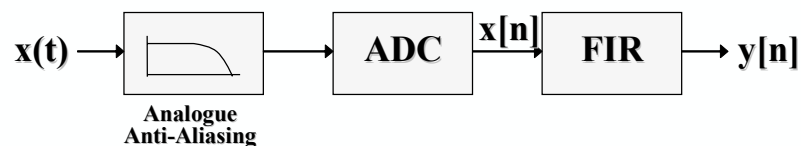
What happens, if the input frequency f goes beyond $f_A/2$, violating the SHANNON-theorem? The magnitude rises from 0 to 1, introducing false (“aliased”) frequency components!



The solution to suppress any frequencies that violate the sampling theorem is to introduce an anti-aliasing filter before the samples are taken. This way, no frequency component beyond $f_A/2$ will disturb the digital processing. The anti-alias filter is an analogue low-pass filter.

FIR – Example 1 (cont.)

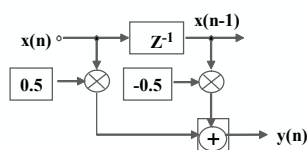
- ◆ **Solution: Use an anti-aliasing filter at input to limit all input frequencies to $f_A/2$.**



14 - 15

In the next example for an FIR-Filter of order 1 we only change coefficient b_1 from $+0.5$ into -0.5 .

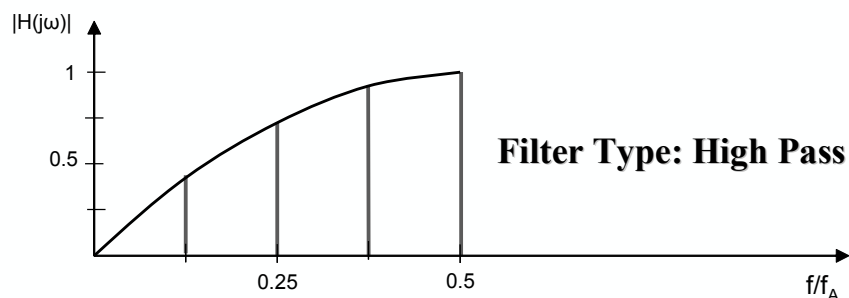
FIR – Example 2



$$b_0 = 0.5 \quad b_1 = -0.5$$

- Frequency Response ?
- Type of Filter ?

Note : We only changed b_1 from $+0.5$ to **-0.5** !

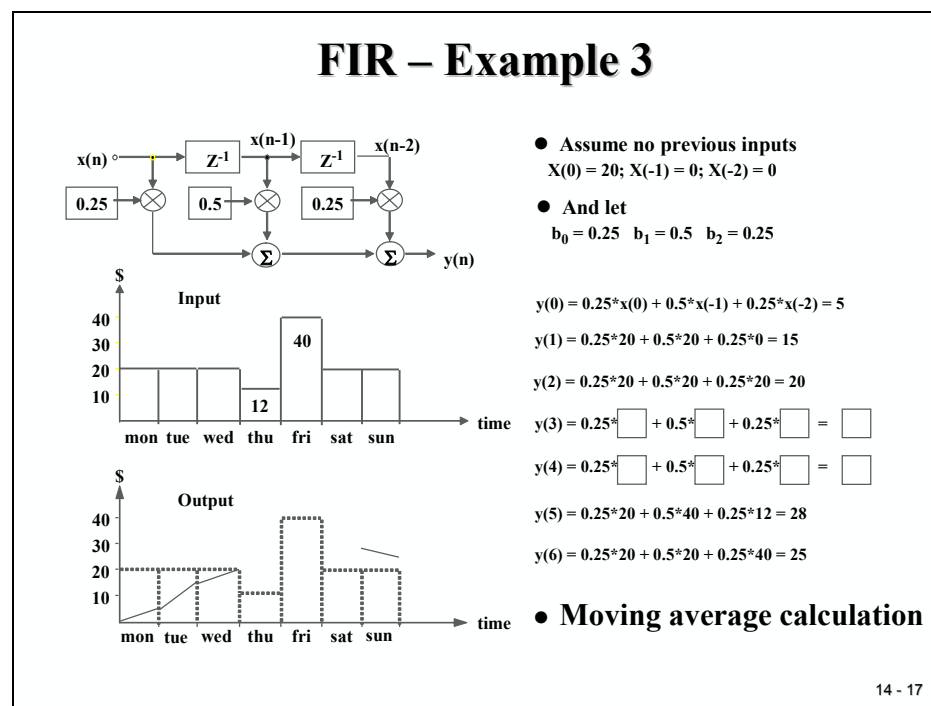


14 - 16

What happens? The digital FIR-filter now damps frequencies around $f=0$ and amplifies an input frequency of $f = f_A/2$ by 1. The filter type has changed from low-pass into high-pass. By modifying coefficients we can change the behaviour of the filtering system. Compare this feature with an analogue filter, where you would have to change resistors or capacitors with a soldering iron.

If the modification of filter coefficients is done in real-time by the controller code itself, we call this an “Adaptive Filter”.

We can also use digital filters to non-technical topics. The next example of a 2nd order FIR-Filter shows how the average stock price per week is calculated using the “moving average calculation”:



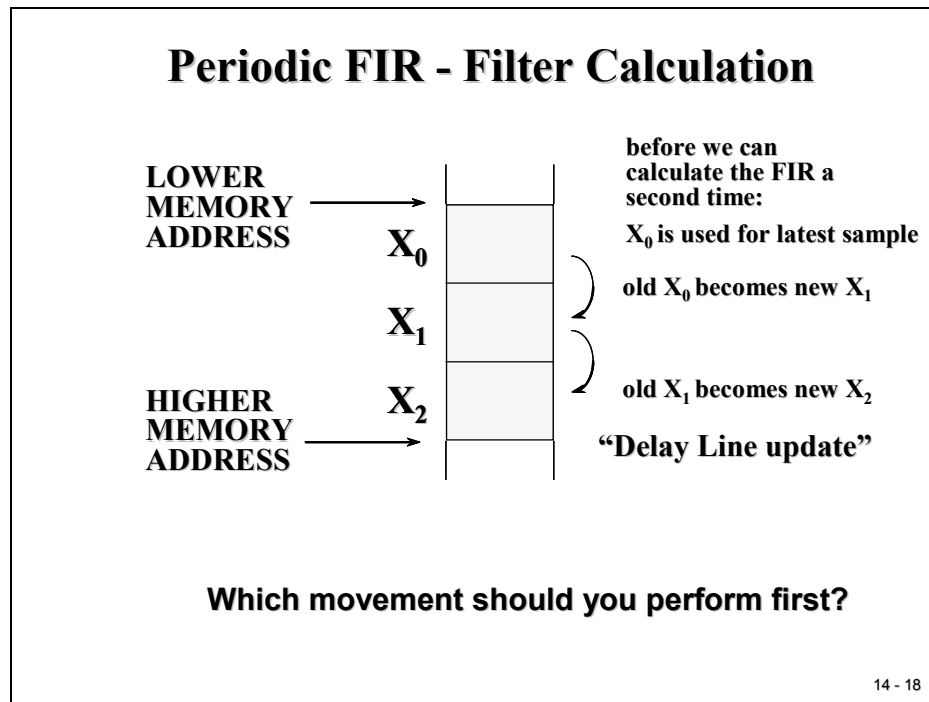
The input is the daily stock price; the output is the moving average. Due to the nature of causally determined systems, we assume that we do not have previous inputs when we start on Monday (time = zero). Of course a broker would know the prices from last week and would request us to add them to our calculation – technically speaking, the stock market is a non causal system, probably practically too.

Anyway, if our calculation advances to Wednesday everybody will be pleased with our results of a moving average.

Let's go back to more technical issues and leave the stock market brokers without further DSP-support.

FIR Implementation in C

Before we proceed to the implementation of a FIR filter using the C28x, let us recall one important step between the periodic calculations. The sample buffer must be prepared to include the latest sample value at the start of the buffer by shifting all elements by one position. Quite often this is done in ascending order:



The following code is taken from a Texas Instruments example to implement FIR-Code for the C28x in IQ-Math-format. You have seen that this fixed-point math's is much better adapted to the C28x than any standard ANSI-C solution – in terms of computing power.

The name of the function “IQssFIR” relates to “IQ – single source FIR” – only one stream of input numbers is computed. Input parameters are two pointers to the array of input samples and to the coefficients and the number of taps – that's order minus 1.

The processing is based on the data type “_iq” which is defined in “IQmathLib.h”. The return parameter is the new output value $y(k)$, also in “_iq”-format.

When you inspect the code, you will notice that it operates from back to front, placing the two pointers to the end of the two buffers and post-decrementing them after any single multiplication. The shift operation on the delay line is done immediately after the current tap has been processed with the help of a temporary pointer “xold”.

The accumulation is done with a simple add-operation using local variable y .

C sourcecode FIR - IQmath

FIR Filter Implementation in C

```

/*****
* Function: IQssfir()
* Description: IQmath n-tap single-sample FIR filter.
*
*           $y(k) = a(0)*x(k) + a(1)*x(k-1) + \dots + a(n-1)*x(k-n+1)$ 
*
* DSP: TMS320F2812, TMS320F2811, TMS320F2810
* Include files: DSP281x_Device.h, IQmathLib.h
* Function Prototype: _iq IQssfir(_iq*, _iq*, Uint16)
* Useage: y = IQssfir(x, a, n);
* Input Parameters: x = pointer to array of input samples
*                  a = pointer to array of coefficients
*                  n = number of coefficients
* Return Value: y = result
* Notes:
* 1) This is just a simple filter example, and completely
*    un-optimized. The goal with the code was clarity and
*    simplicity,
*    not efficiency.
* 2) The filtering is done from last tap to first tap. This
*    allows
*    more efficient delay chain updating.
*****/

```

14 - 19

FIR Filter Implementation in C

```

_iq IQssfir(_iq *x, _iq *a, Uint16 n)
{
    Uint16 i;           // general purpose
    _iq y;              // result
    _iq *xold;          // delay line pointer

    /*** Setup the pointers ***/
    a = a + (n-1);      // a points to last coefficient
    x = x + (n-1);      // x points to last buffer element
    xold = x;           // temporary buffer

    /*** Last tap has no delay line update ***/
    y = _IQmpy(*a--, *x--);

    /*** Do the other taps from end to beginning ***/
    for(i=0; i<n-1; i++)
    {
        y = y + _IQmpy(*a--, *x); // filter tap
        *xold-- = *x--;          // delay line update
    }
    return(y);
}

```

14 - 20

FIR Implementation in Assembly Language

Although the previous example of a C based implementation of a FIR algorithm used IQ-Math function calls to calculate the next value for the output, there is still headroom to optimize the FIR code for the C28x. As we have seen from the C example, basic mathematical operations in this algorithm are multiply instructions for each “coefficient” and “sample” in the delay chain and add operations to sum the partial products. To prepare the next filter calculation cycle, all sample values have been shifted by one position after they have been processed.

DSP's have a unique group of assembly instructions that take advantage of the parallel hardware units: “Arithmetic Logic Unit (ALU)” – for the sum operation and “Hardware Multiplier (MUL)” – for the multiplication. Thanks to the Harvard Architecture of DSP's, two operands can be read simultaneously – one from the data bus and the other from the program-bus.

The assembly language instruction set supports these types of operations in single clock cycle with the “Multiply and Accumulate” (MAC) instruction. In case of the C28x, two groups of assembly instructions are available:

- MAC for 16-bit and 32-bit operands
- DMAC for 16-bit operands

The DMAC – “Dual Mac” instruction takes advantage of the 32-bit width of the internal busses and processes four 16-bit operands (two coefficients and two samples) in a single cycle.

FIR Filter Implementation in ASM

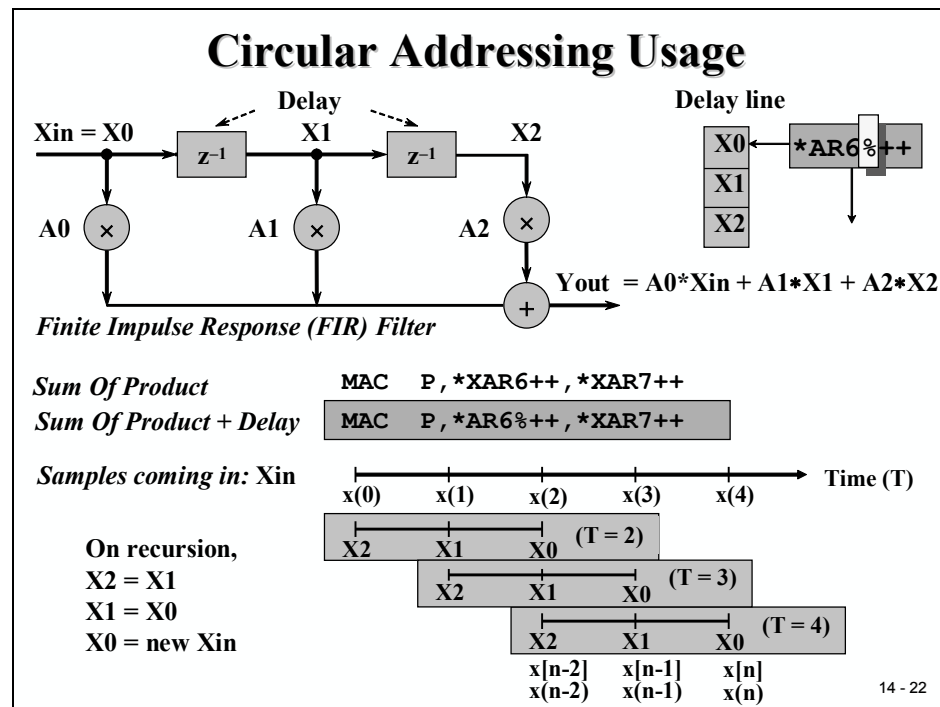
◆ FIR Filter Optimization:

- Previous C solution is a generic one, coded in standard ANSI-C, can be compiled for every microprocessor or embedded microcomputer. Works well.
- But is not optimized for Digital Signal Processors like the C28x. In case more computing power for the real time calculation of a FIR is needed, one can take advantage of internal parallel hardware resources of a DSP.
- ASM-coding of a FIR allows to reduce the number of clock cycles needed to calculate one loop of the FIR algorithm.
- A new Addressing Mode is used to avoid the shift operations of the delay-line for input samples: “Circular Addressing Mode”
- Describe the Circular Addressing function
- Implement FIR filters using Circular Addressing Mode

14 - 21

Circular Addressing Mode

Knowing that MAC or DMAC will accelerate FIR-code the last portion to be optimized is the shift operation of samples, after they have been processed. Assembly language addressing modes of operands include one particular mode that is used to avoid these shift operations altogether. We won't dive too deep into assembly language programming yet, but to explain this addressing mode, let us take an example.



The new addressing mode is called “Circular Addressing Mode” and it is coded with the percentage (%) –sign in front of the pointer register name:

The instruction

MAC P,*XAR6++,*XAR7++

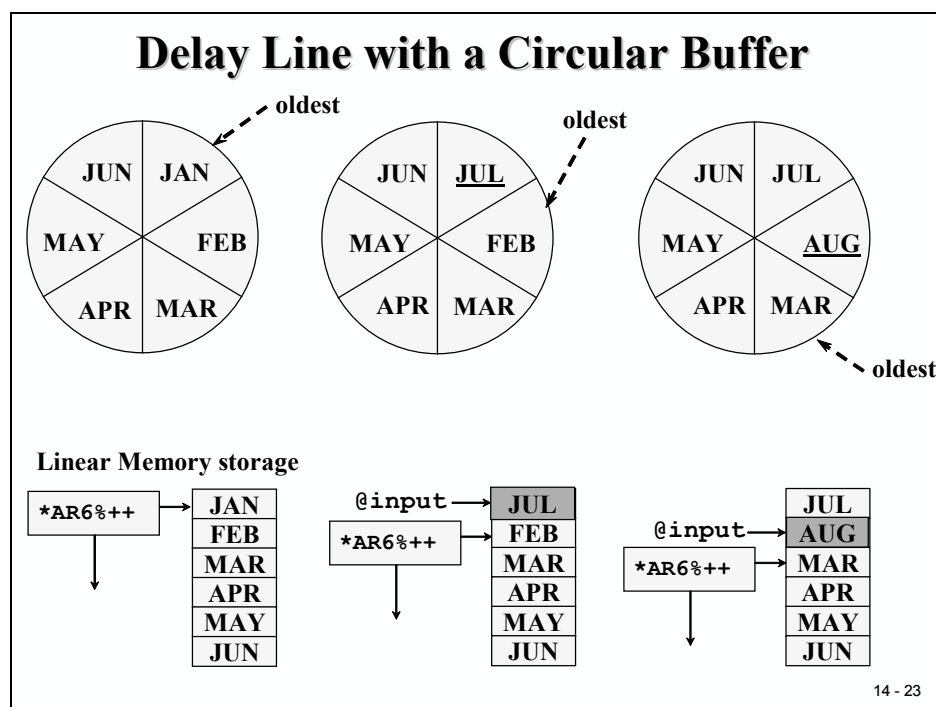
- Adds the previous product (stored in P) to the pre-accumulated sum in register ACC
- Multiplies the data memory operand, pointed to by XAR6, by the program memory operand, pointed to by XAR7
- Post-increments the two pointers XAR6 and XAR7.

If we introduce a new syntax:

MAC P,*AR6%++,*XAR7++

The first pointer XAR6, which points to the sample array in data memory, is used in a circular fashion. Once the pointer has reached the end of the array it will store the next value at start of the buffer. We call this “Circular Buffer”.

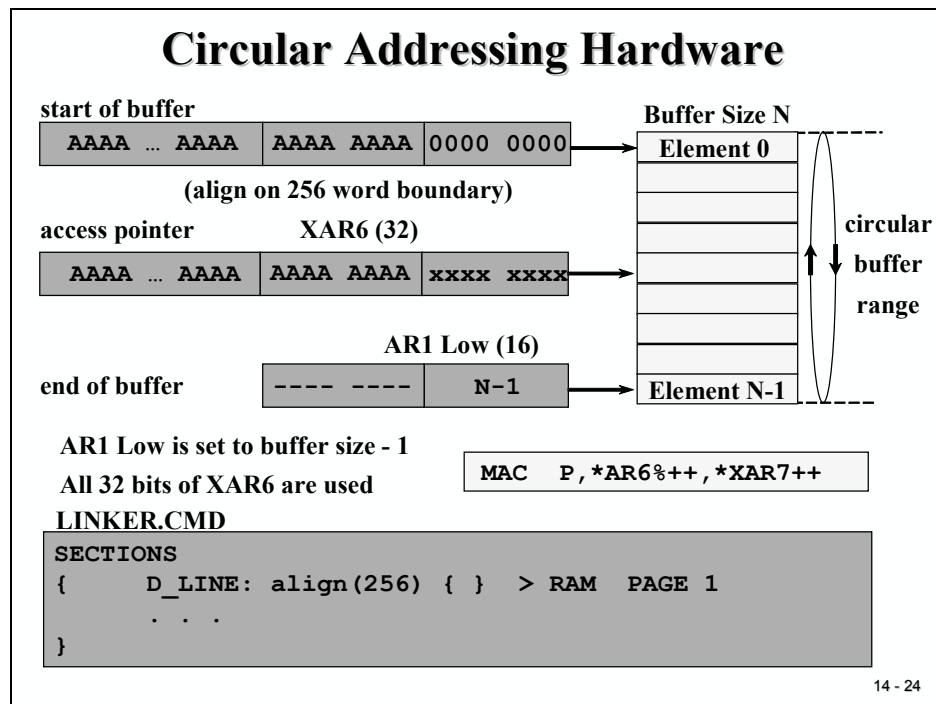
The next slide explains this with a six element buffer, which is used to store the name of month. If the last space in buffer was used, the month “July” will replace the oldest entry “January”



Before we use this circular addressing mode, register XAR7 must be aligned to point to the first element of the coefficient array.

XAR7 must be initialized to point to the start of the circular buffer. This start address of the circular buffer must be aligned to a 256-word boundary (8 LSB's = 0000 0000), which is usually done with the help of a linker command file instruction (see next slide).

The lower 8 bits of register XAR1 are used to specify the size of the circular buffer. This is an implicit usage of register XAR1 by the circular addressing mode; it is not shown in the assembly code!



FIR- Filter Code

The next slide is an assembly language implementation of a 3rd order FIR filter (4 taps). It can be adapted to higher orders by changing the constant “TAPS” and the number of coefficients in array “tbl”. All operands are 16-bit wide in Q15-format.

The directive “.usect” defines un-initialized data memory of length “TAPS” and assigns it to symbol “xn”. The linker command file will connect this section “D_LINE” to physical memory. The directive “.data” defines initialized code memory and assigns it to symbol “tbl”. The four coefficients in this array are in I1Q15-format.

The directive “.text” opens the code-section for assembly instructions. After setting core op-mode bits (SXM, OVM, PM) the two pointers $XAR6$ and $XAR7$ are initialized to point to symbol “xn” and “tbl” respective. $AR1$ is loaded with the buffer size.

Next, a new sample is read from the external ADC at data memory address “0:adc” and stored at first place in the circular buffer. We also could have used the internal ADC. The “%++” operator will set the circular buffer pointer to its next element.

After clearing register ACC, P and OV (“ZAPA”) the following two instructions will do the entire filter work. The repeat instruction (“RPT”) tells the C28x to repeat the following instruction #number plus 1 times, in our case twice. The instruction “DMAC” will perform two 16x16-bit MAC operations, reading and processing two members of “xn” and “tbl” per cycle. The 3rd order FIR filter will be calculated in two clock cycles.

Finally the two halves of the result are added using the instruction “ADDL ACC:P” and the new result is loaded in I1Q15-format to an external DAC at address “0:dac”.

FIR Filter – Dual MAC - Operation

```

TAPS .set      4                      ; FIR - Order +1
xn   .usect    "D_LINE",TAPS          ; sample array in I1Q15
      .data                      ; FIR - Coeffs in I1Q15
tbl  .word     32768*707/1000          ; 0.707
      .word     32768*123/1000         ; 0.123
      .word     32768*(-175)/1000      ; -0.175
      .word     32768*345/1000         ; 0.345
      .text
FIR:  SETC     SXM                      ; 2's complement math
      CLRC     OVM                      ; no clipping mode
      SPM      1                        ; fractional math
      MOVL     XAR7,#tbl                ; coefficient pointer
      MOVL     XAR6,#xn                 ; circular buffer pointer
      MOV      AR1,#TAPS-1              ; buffer offset
      MOV      *XAR6%++,*(0:dac)        ; get new sample ( x(n) )
      ZAPA                      ; clear ACC,P,OVC
      RPT      #(TAPS/2)-1              ; RPT next instr.(#+1)times
||    DMAC     ACC:P,*XAR6%++,*XAR7++   ; multiply & accumulate 2pairs
      ADDL     ACC:P                    ; add even & odd pair-sums
      MOV      *(0:dac),AH              ; update output ( y(n) )
      RET

```

14 - 25

Circular Addressing Summary

Buffer Size

- ◆ Up to 256 words
- ◆ Break larger arrays into ≤ 256 word blocks.

Buffer Alignment

- ◆ Always align on 256-word boundaries, regardless of size. Unused space can be used for other purposes.
- ◆ Let the linker assign addresses. Link largest blocks first.

Usage

- ◆ XAR6 is the only circular pointer.
- ◆ AR1 must be set to the size minus one (0 - 255).
- ◆ Pointer update is post-increment by one (*XAR6%++).
- ◆ 32-bit access causes post-increment by two. Make sure XAR6 and AR1 are even to avoid jumping past end of buffer.

14 - 26

Texas Instruments C28x Filter Library

So for some types of applications, it seems to make sense to code in Assembly Language. But, as usual, there is no need to re-invent the wheel. Better than developing your own code is – use a library. Texas Instruments is offering a variety of libraries for free, one of them us dedicated to Digital Filters.

Texas Instruments C28x Filter Library

- ◆ **MATLAB script to calculate Filter Coefficients for FIR and IIR, includes windowing**
- ◆ **Filter Modules:**
 - ◆ **FIR16: 16-Bit FIR-Filter**
 - ◆ **IIR5BIQ16: Cascaded IIR-Filter (16bit-biquad)**
 - ◆ **IIR5BIQ32: Cascaded IIR-Filter (32bit-biquad)**
- ◆ **C-callable Assembly (“CcA”) Functions**
 - ◆ **Adapted to internal Hardware features of the C28x**
 - ◆ **Uses the Dual –MAC instruction**
 - ◆ **Interface according to ANSI-C standard**

Available from TI-web as document “sprc082.zip”

14 - 27

To make it easier to use this library in a C-based program environment, all library functions are equipped with an interface structure. Thus any library function can be called like an ordinary C subroutine.

An important step in designing a Digital Filter is to calculate Filter Coefficients. This task involves a lot of theoretical background. Without this knowledge, you won’t be able to profile the set of coefficients for a given transfer function. As recommended earlier, join additional courses at your university to understand the math behind Digital Signal Processing.

The library package includes also a MATLAB script to calculate filter coefficients, including windowing techniques.

MATLAB Filter Script

The MATLAB filter script allows you to initialize essential parts of the transfer function, like sampling frequency, filter type, order, type of window and corner frequency.

MATLAB Filter Script

FIR Filter Design Example: Low-pass Filter of Order 50
 LPF Specification:
 FIR Filter Order : 50
 Type of Window : Hamming
 Sampling frequency : 20KHz
 Filter Corner Frequency : 3000Hz

ezFIR FILTER DESIGN SCRIPT
 Input FIR Filter order(EVEN for BS and HP Filter) : 50
 Low Pass : 1
 High Pass : 2
 Band Pass : 3
 Band Stop : 4
 Select Any one of the above Response : 1
 Hamming : 1
 Hanning : 2
 Bartlett : 3
 Blackman : 4
 Select Any one of the above window : 1
 Enter the Sampling frequency : 20000
 Enter the corner frequency(Fc) : 3000
 Enter the name of the file for coeff storage : lpf50.dat

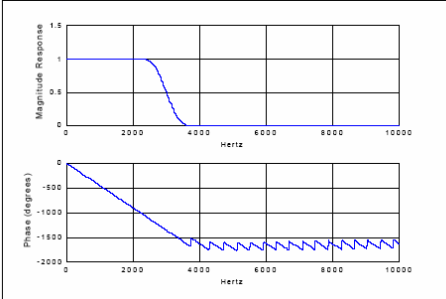
14 - 28

The output of the MATLAB calculation is a list of coefficients and a graph of magnitude and phase of the filter response.

MATLAB Filter Script

MATLAB – Output File for Filter Coefficients:

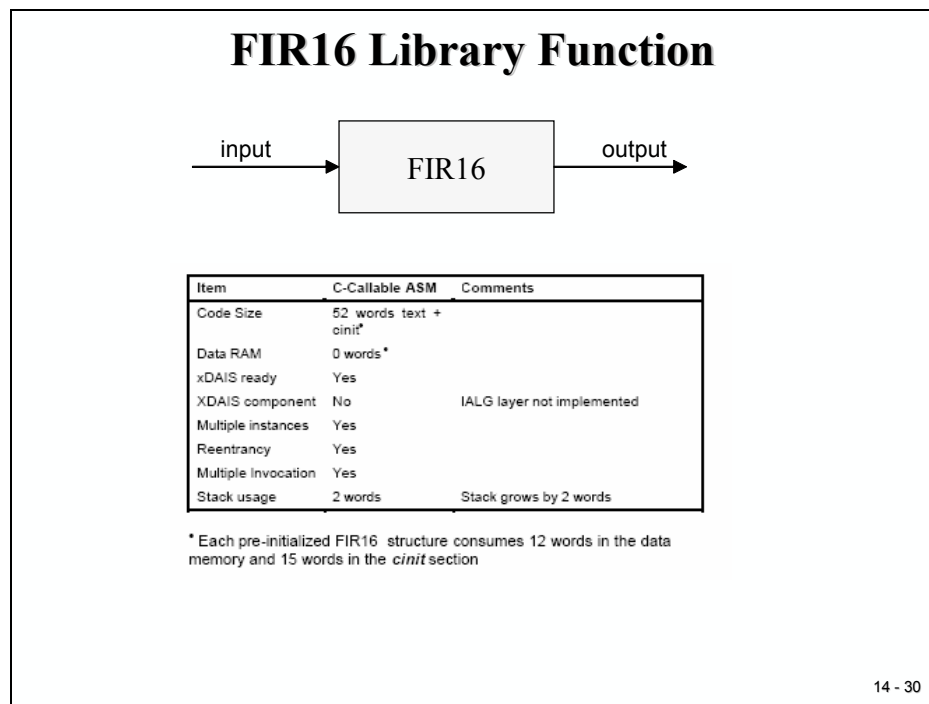
```
#define FIR16_COEFF {\
9839,-2219809,-1436900,853008,3340889,3668111,-896,\
-5963392,-8977456,-3669326,8585216,18152991,13041193,\
-8257663,-30867258,-31522540,131,45285320,64028535,\
25231269,-58654721,-124846025,-94830542,68157453,\
320667626,551550942}
```



14 - 29

FIR16 Library Function

FIR16 is one of the library functions of sprc082. It processes a single stream of input samples in Q15-format into a new output value of the same format. One instance of this function occupies 52 words of code memory and processing time is 350 ns with a 150 MHz C28x.



The format of the FIR16 functions object structure is shown next. Interface parameters are

Input:

- 2 pointers to coefficients and samples
- Order of filter
- 1 pointer to an initialize function of the filter
- 1 pointer to the calculation function

Output:

- 1 new filter output value.

To guarantee the ability to operate with more than 1 instance of the filter, you should call the function by its function parameters (init, calc) only.

FIR16 Library Function

Object Definition:

```
typedef struct {
    long *coeff_ptr;      /* Pointer to Filter coeffs */
    long *dbuffer_ptr;    /* Delay buffer pointer */
    int cbindex;          /* Circular Buffer Index */
    int order;            /* Order of the Filter */
    int input;            /* Latest Input sample */
    int output;           /* Filter Output */
    void (*init)(void *); /* Pointer to Init function */
    void (*calc)(void *); /* Pointer to calc function */
}FIR16;
```

coeff_ptr: Pointer to the Filter coefficient array.
 dbuffer_ptr: Pointer to the Delay buffer.
 cbindex: Circular buffer index, computed internally by initialization function based on the order of the filter.
 order: Order of the Filter. Q0-Format, range 1 – 255
 input: Latest input sample to the Filter. Q15-Format (8000-7FFF)
 output: Filter output value. Q15-Format (8000-7FFF)

14 - 31

The next slide is an example for the usage of the filter function. An instance of FIR16, called “lpf” has been defined in a DATA_SECTION “firfilt”. All accesses to parameters and functions are made by this instance.

FIR16 Library Usage Example

```
#define FIR_ORDER 50          /* Filter Order */

#pragma DATA_SECTION(lpf, "firfilt");
FIR16 lpf = FIR16_DEFAULTS;
#pragma DATA_SECTION(dbuffer,"firlldb");
long dbuffer[(FIR_ORDER+2)/2];

const long coeff[(FIR_ORDER+2)/2]= FIR16_LPF50;
main()
{
    lpf.dbuffer_ptr=dbuffer;
    lpf.coeff_ptr=(long *)coeff;
    lpf.order=FIR_ORDER;
    lpf.init(&lpf);
}
void interrupt isr20khz()
{
    lpf.input=xn;
    lpf.calc(&lpf);
    yn=lpf.output;
}
```

14 - 32

Lab 14: FIR – Filter for a square-wave signal

Objective

Lab 14: LP -Filter of a square wave

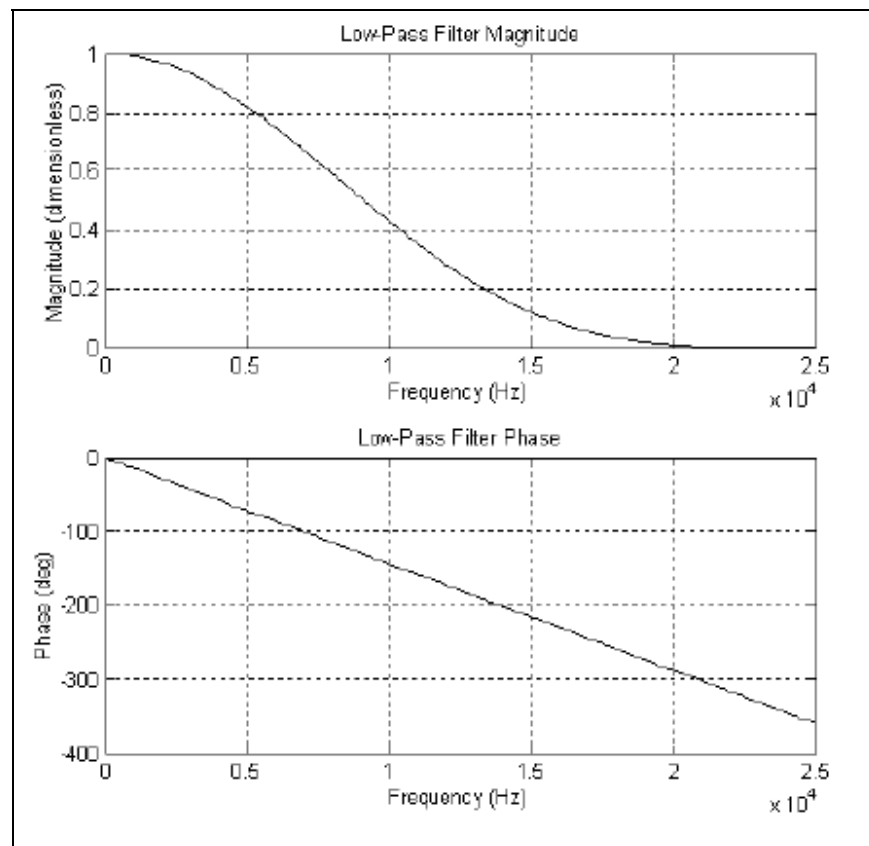
Objective:

- ◆ Generate a square wave signal of 2 KHz at EVA-T1PWM
- ◆ Asymmetric PWM , duty cycle 50%
- ◆ Use T1-Compare Interrupt Service to serve the watchdog
- ◆ Wire - Connect T1PWM to ADC-input ADCIN2
- ◆ Sample the square wave signal at 50KHz
- ◆ Sample period generated by EVA-Timer 2
- ◆ Store samples in buffer “AdcBuf”
- ◆ Filter the input samples with a FIR – Low pass 4th order
- ◆ Store filtered output samples in buffer “AdcBufFiltered”
- ◆ Visualize “AdcBuf” and “AdcBufFiltered” graphically by Code Composer Studio’s Graph Tool

14 - 33

The lab experiment consists of four parts:

- First we will generate a 2 kHz square wave signal at output T1PWM. At the Zwickau adapter board this signal can be connected to a loudspeaker (JP3 closed). Or, use a scope to visualize the signal.
- Second we will feedback this signal back into one channel of the internal ADC and store the digital samples in a data memory buffer “AdcBuf”.
- Next, we call a Low-Pass Filter of FIR-Type with order 4 to wave-shape the signal edges. The coefficients were calculated with MATLAB as:
 - 1/16, 4/16, 6/16, 4/16 and 1/16
 - The sampling frequency is set to 50 kHz
- The filtered numbers will be stored in “AdcBufFiltered”
- Finally we will use Code Composer Studios graphical tool to visualize the contents of “AdcBuf” and “AdcBufFiltered”. We will take advantage of the real time debug capabilities to display the data without interrupting or delaying the C28x while it is running!



Procedure

Open Files, Create Project File

1. Create a new project, called **Lab14.pjt** in E:\C281x\Labs.
2. Open the file Lab5.c from E:\C281x\Labs\Lab5 and save it as Lab14.c in E:\C281x\Labs\Lab14.
3. Add the source code file to your project:
 - **Lab14.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:\tidcs\c28\dsp281x\v100\DSP281x_common\source* add to project:

- **DSP281x_PieCtrl.c**
- **DSP281x_PieVect.c**
- **DSP281x_DefaultIsr.c**

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

- **rts2800_ml.lib**

Project Build Options

5. 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; C:\tidcs\C28\IQmath\clIQmath\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 Source Code

7. Open Lab14.c to edit: double click on “Lab14.c” inside the project window. First we have to remove the parts of the code that we do not need any longer. We will not use the CPU core timer 0 in this exercise; therefore we do not need the prototype of interrupt service routine “cpu_timer0_isr()”. Instead, we need a new ISR for EVA-Timer1-Compare-Interrupt. Add a new prototype interrupt function: “interrupt void T1_Compare_isr(void)”.
8. We do not need the variables “i”, “time_stamp” and frequency[8]” from Lab5 - delete their definition lines at the beginning of the function “main”.

9. Next, modify the re-map lines for the PIE entry. Instead of “PieVectTable.TINT0 = &cpu_timer_isr” we need to re-map:

PieVectTable.T1CINT = &T1_Compare_isr;

10. Delete the next two function calls: “InitCpuTimers();” and “ConfigCpuTimer(&CpuTimer0, 150, 50000);” and add an instruction to enable the EVA-Timer1-Compare interrupt. According to the interrupt chapter this source is wired to PIE-group 2 , interrupt 5:

PieCtrlRegs.PIEIER2.bit.INTx5 = 1;

Also modify the set up for register IER into:

IER |= 2;

11. Next we have to initialize the Event Manager Timer 1 to produce a PWM signal. This involves the registers “GPTCONA”, “T1CON”, “T1CMPR” and “T1PR”.

For register “GPTCONA” it is recommended to use the bit-member of this predefined union to set bit “TCMPOE” to 1 and bit field “T1PIN” to “active low”.

For register “T1CON” set

- The “TMODE”-field to “counting up mode”;
- Field “TPS” to “divide by 1”;
- Bit “TENABLE” to “**disable timer**”;
- Field “TCLKS” to “internal clock”
- Field “TCLD” to “reload on underflow”
- Bit “TECMPR” to “enable compare operation”

12. Remove the 3 lines before the while(1)-loop in main:

- “CpuTimer0Regs.TCR.bit.TSS = 0;”
- “i = 0;”
- “time_stamp = 0;”

and add 4 new lines to initialise T1PR, T1CMPR, to enable GP Timer1 Compare interrupt and to start GP Timer 1:

EvaRegs.T1PR = 37500;

EvaRegs.T1CMPR = EvaRegs.T1PR/2;

EvaRegs.EVAIMRA.bit.T1CINT = 1;

EvaRegs.T1CON.bit.TENABLE = 1;

What is this number 37500 for? Well, it defines the length of a PWM period:

$$f_{PWM} = \frac{f_{CPU}}{T1PR \cdot TPS_{T1} \cdot HISCP}$$

with $TPS_{T1}=1$, $HISCP = 2$, $f_{CPU} = 150\text{MHz}$ and a desired $f_{PWM} = 2\text{ kHz}$ we derive: $T1PR = 37500!$

T1CMPR is preloaded with half of T1PR. Why's that? Well, in general T1CMPR defines the width of the PWM-pulse. Our start-up value obviously defines a pulse width of 50%.

13. Modify the endless while(1) loop of main! We will perform all activities out of GP Timer 1 Compare Interrupt Service. Therefore we can delete almost all lines of this main background loop, we only have to keep the watchdog service:

```
while(1)
{
    EALLOW;
    SysCtrlRegs.WDKEY = 0xAA;
    EDIS;
}
```

14. Rename the interrupt service routine “cpu_timer0_isr” into “T1_Compare_isr”. Remove the line “CpuTimer0.InterruptCount++;” and replace the last line of this routine by:

```
PieCtrlRegs.PIEACK.all = PIEACK_GROUP2;
```

Before this line add another one to acknowledge the GP Timer 1 Compare Interrupt Service is done. Remember how? The Event Manager has 3 interrupt flag registers “EVAIFRA”, “EVAIFRB” and “EVAIFRC”. We have to clear the T1CINT bit (done by setting of the bit):

```
EvaRegs.EVAIFRA.bit.T1CINT = 1;
```

Build and Load

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

Project → Build

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

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

File → Load Program and choose the desired output file.

Test

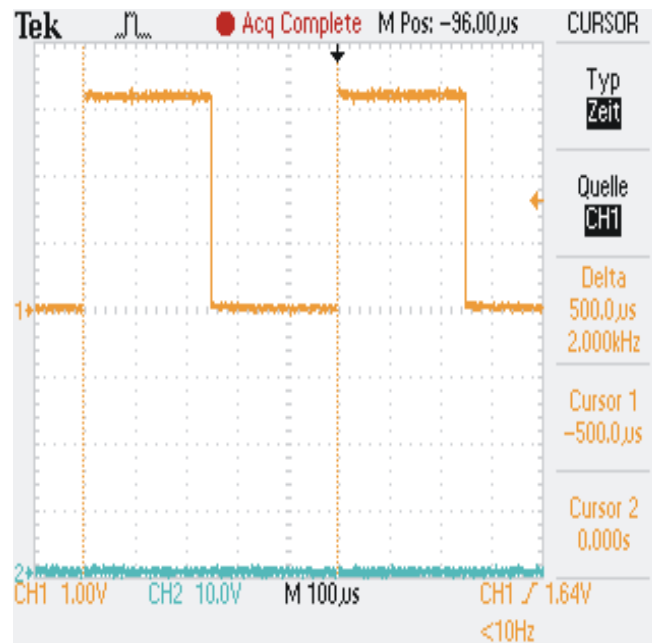
17. Reset the DSP by clicking on:

Debug → Reset CPU followed by
Debug → Restart and
Debug → Go main.

18. When you now run the code the DSP should generate a 2 kHz PWM signal with a duty cycle of 50% on T1PWM. If you have an oscilloscope you can use jumper JP7 (in front of the loudspeaker) of the Zwickau Adapter board to measure the signal.

If your laboratory can't provide a scope, you can set a breakpoint into the interrupt service routine of T1 Compare at line "PieCtrlRegs.PIEACK.all = PIEACK_GROUP2; Verify that your breakpoint is hit periodically, that register T1PR holds 37500 and register T1CMPR is initialized with 18750. Use the Watch Window to do so.

Do not continue with the next steps until this point is reached successfully! Instead, go back and try to find out what went wrong during the modification of your source code.



End of Lab 14 Part 1

Feedback the Signal into ADC

19. Three files have been provided to this lab to add the ADC functionality. Add the two files “Adc.c”, “Adc_isr.c” and “filter.c” to your project.
20. In function “InitSystem” of Lab14.c enable the ADC-clock:

SysCtrlRegs.PCLKCR.bit.ADCENCLK = 1;

21. In “main”, just after the call of “InitPieVectTable()” add a call to initialize the ADC:

InitAdc();

This function will setup the ADC to one conversion per trigger. ADCIN2 will be converted by SEQ1 out of Event Manager A trigger. An interrupt will be requested with every end of sequence. Inspect the code of “InitAdc()”.

22. Next, we have to connect the ADC interrupt to a new function: “ADC_FIR_INT_ISR()”. This function is defined in the new source code file “Adc_isr”. All we have to do is to replace the entry in the PieVectTable by this new address. Look in “main” and locate the line, where we already overload the PieVectTable with T1_Compare_isr. Add a new line:

PieVectTable.ADCINT = &ADC_FIR_INT_ISR;

The new function “ADC_FIR_INT_ISR” is not declared yet in “Lab14.c”. Therefore we have to add a new prototype statement at the beginning of “Lab14.c”:

interrupt void ADC_FIR_INT_ISR(void);

Register IER must be modified to enable INT1 (ADC) and INT2 (T1-Compare):

IER |= 3;

Set up ADC sample period (Timer 2)

23. EVA-Timer 2 will be used to generate the sample period for the ADC. Each period event of T2 will trigger a start of an ADC sequence automatically, if we enable this option:

EvaRegs.GPTCONA.bit.T2TOADC = 2;

Add this line in front of the while(1)-loop of “main”.

Before the code enters the while(1)-loop we have to initialize EVA-Timer2 to produce a sample period of 50 kHz. Register T2CON defines the operating mode. Let's select:

- Continuous up – Mode
- Timer – Prescaler : 1

- Enable Timer (TENABLE = 1)
- No Timer Compare Operation Enable

Register T2PR must define the time period. According to:

$$f_{PWM} = \frac{f_{CPU}}{T2PR \cdot TPS_{T2} \cdot HISCP}$$

and with a given 150MHz CPU frequency, HISCP =2, TPST2 = 1 and 50 KHz as output frequency we derive:

$$T2PR = 1500$$

Add the necessary instructions for T2PR and T2CON!

Connect T1PWM to ADCIN2

24. Connect T1PWM (eZdsp Pin P8 -15) to ADCIN2 (eZdsp Pin P9 -6) with a wire or a 1000 Ohm resistor provided by your laboratory technician.

Caution: Be careful when connecting pins while the eZdsp is powered on. By plugging the wire into wrong pins you can damage the board!

To be safe, ask your technician for assistance before you connect anything!

Build, Load and Test

25. Finalize the Project and prepare a test: :

Project	→	Build
File	→	Load Program
Debug	→	Reset CPU
Debug	→	Restart
Debug	→	Go main.

If all code was modified correct, the 2 kHz-Signal should still be audible at the loudspeaker.

To verify that our ADC sampling is operating as expected, place a breakpoint at the beginning of the ADC's interrupt service routine ("ADC_FIR_INT_ISR") in file "Adc_isr.c". If you run the code in real time (F5), it should hit the breakpoint periodically.

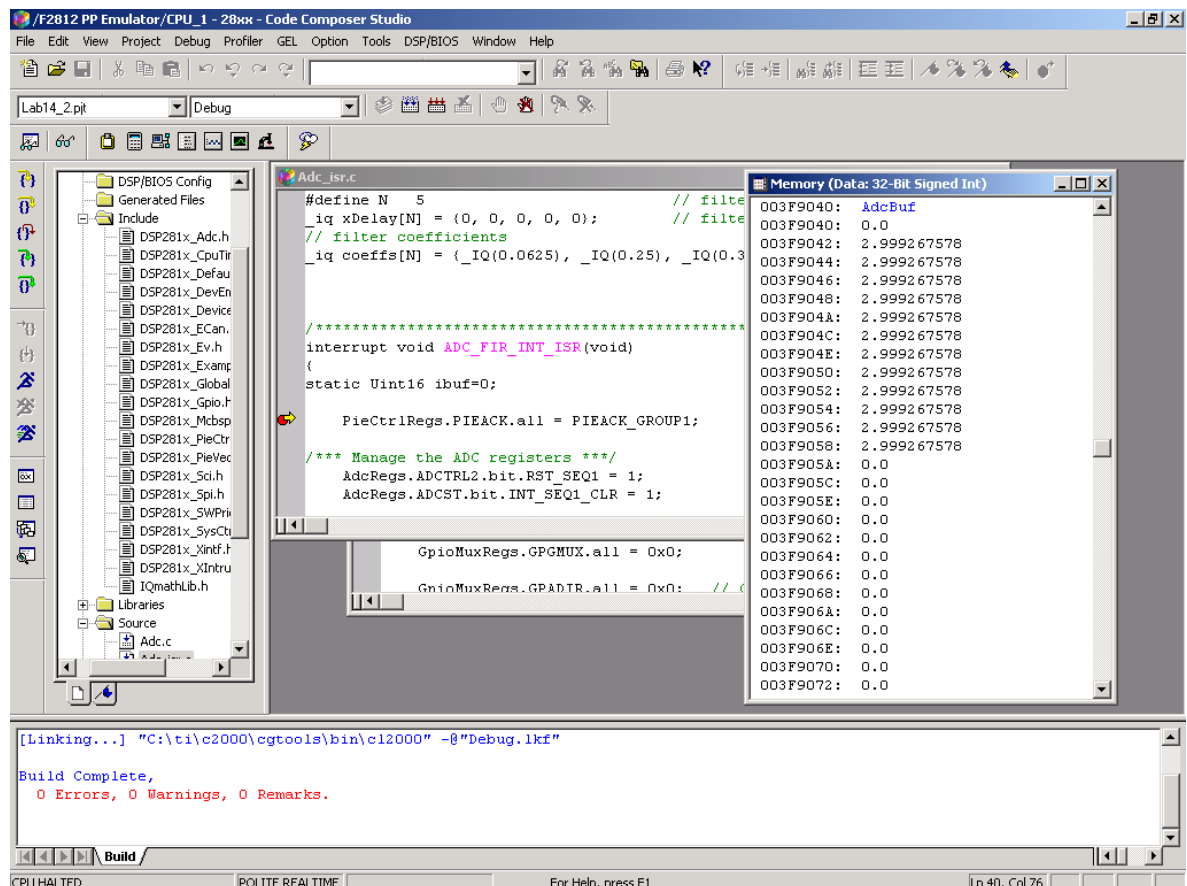
Inspect and Visualize the FIR

26. Let us inspect the code in this ISR. The sample (AdcRegs.ADCRESULT0) is stored in a buffer “AdcBuf” after it is scaled to 3.0V and converted into an IQ-number. The default value is I8Q24, defined in “IQmathLib.h”.

We can display the content of this buffer in a memory window:

View → Memory

- Address: AdcBuf
- Q-Value: 24
- Format: 32_Bit Signed Int
- Page: Data



27. Back to “ADC_FIR_INT_ISR”. After the sample is stored in `AdcBuf`, it is also placed as latest sample in a filter-array “`xDelay`”. Then a function “`IQssfir`” is called and its return value is stored in a new buffer “`AdcBufFiltered`”. Obviously, the return value is the output signal from the FIR-Filter.
28. Now inspect the filter-function “`IQssfir`” in “`filter.c`”. Input parameters are the samples, the coefficients and number of taps. The filter implementation is a C-based solution with no optimization. The difference to the solution that was presented at the beginning of the chapter is that it is a tailored solution for the C28x IQ-Math Library, running much faster than any ANSI-C solution with float variables.

You have also learned about Texas Instruments Filter Library. Knowing that these library functions are based on an assembly language implementation we could move on and increase the speed of the FIR-calculation further by replacing the “`IQssfir`”-function with one from the library.

CCS Graphical Tool

29. Now let’s visualize both the square wave and the filtered signal. CCS has a build in tool to visualize the content of an area of code or data memory graphically. We can use this tool to plot the content of “`AdcBuf`” and “`AdcBufFiltered`”:

View → Graph → Time/Frequency

Select the properties:

- Display Type : Dual Time
- Start Address upper display: `AdcBuf`
- Start Address lower display: `AdcBufFiltered`
- Page: Data
- Acquisition Buffer Size: 50
- Display Data Size: 50
- DSP Data Type: 32-bit signed integer
- Q-Value 24
- Sampling Rate: 50000
- Time Display Unit: μ s

When you close the property window with <OK> a Graphical Display with a yellow background should pop up.

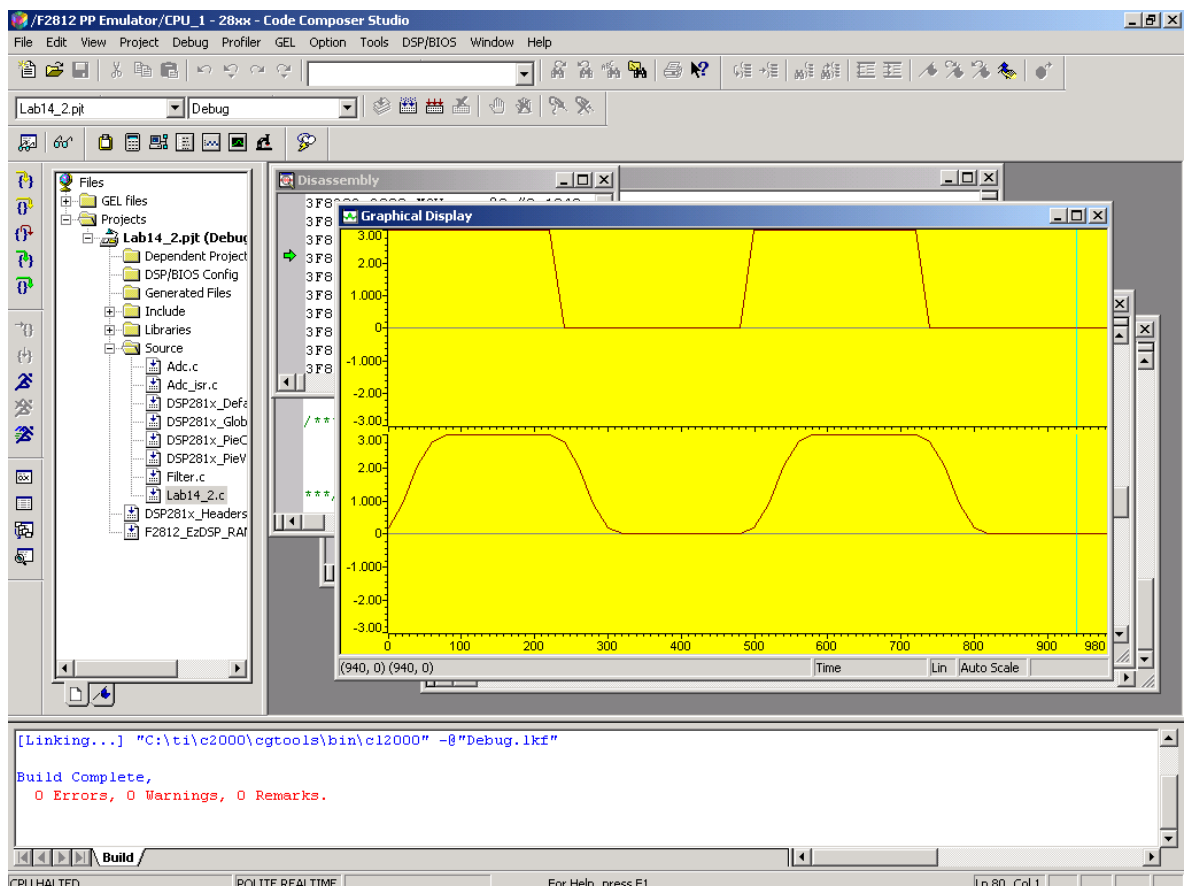
30. Enable Real Time Mode:

Debug → Real Time Mode

Click right in the graph window and select “Continuous Refresh”

31. Run the code in Real Time (F5)

The Graph window should look like this:



End of Lab 14

This page has intentionally been left blank.

C28x Digital Motor Control

Introduction

In this module, we will look into an application area that is not usually the domain of Digital Signal Processors: real-time control of electrical motors. In the old days, control of the speed and torque of electrical motors was done purely using analog technology. Since the appearance of microprocessors, more and more control units have been designed digitally, using the advantages of digital systems. This improves the degree of efficiency and allows the implementation of more advanced control schemes, thanks to increase real-time computing power. It is a natural progression to use the internal hardware computing units of a DSP to transfer the calculation from a standard microprocessor to a DSP. This way, we can implement more advanced algorithms in a given period of time.

However, to use a digital controller for motor control, the system needs a little more than computing power. The output signals of the digital controller to the power electronic are usually generated as pulse width modulated signals (PWM). It would be most cost-effective if the controller could be equipped with an internal PWM-unit. To control the operation of the motor we need to do some measurement for currents and voltages – analogue to digital converters (ADC) will be helpful as well. A typical unit to perform a position/speed measurement is an optical encoder; quite often, we build in a Quadrature Encoder (QEP). Recalling all parts of the C28x we discussed in chapters 1-9, you can imagine that the C28x is an ideal device for Digital Motor Control (DMC).

The chapter will not impart a deep knowledge of electrical motors and drives. Instead, it will give you a sense what needs to be done to use the C28x to control the motor of a vacuum cleaner or the motor of an electrical vehicle. To fully understand the principles, it needs a lot more classes at university. If you are in a course of electrical engineering that focuses on drives and power engineering, you might be familiar with most of the technical terms. If not, see this chapter as a challenge for you to open up another application field for a Digital Signal Processor.

Module 15 is based on a Texas Instruments Application Note (SPRC129, “C28x & F28x PMSM3_1: 3-Phase sensored Field Oriented Control“, Version 3.0, May-17-2003). Depending on the laboratory equipment at your university, you might be offered the chance to attend a laboratory session to build a working solution for such a motor control.

After a general discussion of the basics of motor control, we will focus on the computing steps necessary to build a control scheme for Field Oriented Control (FOC). This will be done in an incremental way, to help you to understand the computing steps. We will not discuss any power electronic hardware requirements or design constraints.

A complete working reference design is available in the appendix of this CD-ROM, based on a small 24V 3-phase PMSM, the eZdspTMS320F2812 and the DMC550, a power electronic adapter board available from Spectrum Digital.

Module Topics

C28x Digital Motor Control	15-1
<i>Introduction</i>	<i>15-1</i>
<i>Module Topics.....</i>	<i>15-2</i>
<i>Basics of Electrical Motors.....</i>	<i>15-3</i>
Motor Categories	15-3
3 – phase Motor	15-4
Permanent Magnet Synchronous Motor	15-6
Brushless Direct Current Motor (BLDC)	15-7
3 – Phase Power Switches	15-8
<i>Motor Control Principles.....</i>	<i>15-9</i>
Scalar Control (“V/Hz”)	15-9
Field Oriented Control (FOC)	15-10
FOC Control Scheme.....	15-11
<i>FOC Core Math Operations</i>	<i>15-12</i>
PARK Transform.....	15-12
CLARKE Transform	15-14
PARK Transform Summary	15-15
<i>Texas Instruments Digital Motor Control Library.....</i>	<i>15-16</i>
Library Modules	15-18
FOC for PMSM	15-19
Hardware Laboratory Setup.....	15-20
PMSM Library Modules.....	15-22
PMSM Software Flowchart.....	15-23
<i>Lab 15: PMSM control project</i>	<i>15-24</i>
Build Level 1	15-24
Build Level 2	15-28
Build Level 3	15-30
Build Level 4	15-32
Build Level 5	15-35

Basics of Electrical Motors

Motor Categories

In order to classify the different electrical motors families, we can distinguish motors driven by direct current (DC) and motors driven by an alternating current (AC). DC motors are the most popular ones: both stators and rotors carry an excitation created by coils or windings in which DC current circulates. In order to ensure the motor rotation by commutating the windings, brushes are permanently in contact with the rotor.

Electrical Motor families

Motor Classification:

- ◆ **Direct Current Motors (DC)**
- ◆ **Alternating Current Motors (AC)**
 - ◆ **Asynchronous Induction Motor (ACI)**
 - ◆ **Permanent Magnet Synchronous Motor (PMSM)**
 - ◆ **Synchronous Brushless DC Motor (BLDC)**

15 - 2

Under the classification of AC motors, we have synchronous motors and asynchronous motors; both motor types are induction machines.

Asynchronous machines require a sinusoidal voltage distribution on the stator phases in order to induce current on the rotor, which is not fed by any currents nor carries any magnetic excitation.

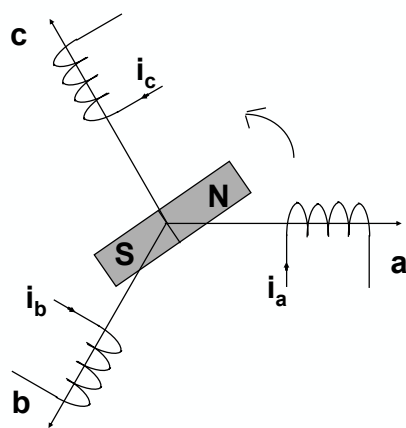
Synchronous motors are usually called “Brushless DC Motors” (BLDC) but can also be called “Permanent Magnet Synchronous Motors” (PMSM) depending on their construction and the way of being controlled. In this type of motor, we have one sinusoidal or trapezoidal excitation on the stator and one constant flux source on the rotor (usually a magnet).

Because of the selected application note as the base of this module, we will focus on the PMSM – type for the rest of this module.

3 – phase Motor

Let us start with a brief discussion of a 3-phase motor. We can start with a magnet rotating in front of an electrical winding. This can be expanded into a three phase winding approach. Windings a, b and c are physically spaced 120° apart from each other. While the magnet is rotating with a mechanical speed Ω , each winding sees a varying flux that induces voltage and current at their ends. Applying the Faraday law to each of the phases, we can obtain a three-phase equation system that will be the base of our rotating machine study.

3 – Phase - Motor (PMSM)



$$\begin{cases} i_a = I_s \cdot e^{j\omega t} \\ i_b = I_s \cdot e^{j\omega t - \frac{2\pi}{3}} \\ i_c = I_s \cdot e^{j\omega t - \frac{4\pi}{3}} \end{cases}$$

$$\begin{cases} e_a = E \cdot e^{jp\Omega t} \\ e_b = E \cdot e^{jp\Omega t - \frac{2\pi}{3}} \\ e_c = E \cdot e^{jp\Omega t - \frac{4\pi}{3}} \end{cases}$$

- ◆ For most three phase machines, the winding is stationary, and magnetic field is rotating
- ◆ Three phase machines have three stator windings, separated 120° apart physically
- ◆ Three phase stator windings produce three magnetic fields, which are spaced 120° in time

15 - 3

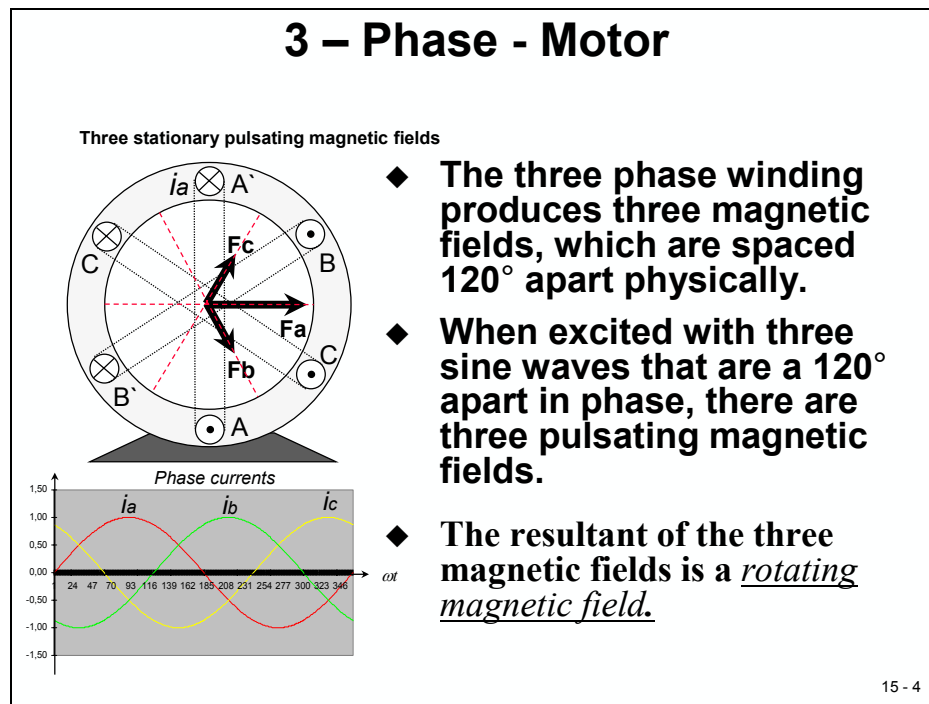
Depending on the number of pole pairs (p) of the rotating magnet, the electrical pulsation of the induced voltage will change. For a two pole pairs rotating magnetical excitation, the electrical pulsation is twice the mechanical pulsation. For a complete 360° turn of the magnet, the windings see two North and South poles. This observation for two pole pair magnet can be generalized to $\omega = p \cdot \Omega$ with electrical (ω) and mechanical (Ω) pulsation. Parameter e is the inductive voltage, called „Back Electromotive Force (Bemf)“.

We can represent the system using the following equations (in real notation as an example):

$$\begin{aligned} e_a &= \Phi \omega \sqrt{2} \sin(\omega t) = E \sqrt{2} \sin(p\Omega t) \\ e_b &= \Phi \omega \sqrt{2} \sin\left(\omega t - \frac{2\pi}{3}\right) = E \sqrt{2} \sin\left(p\Omega t - \frac{2\pi}{3}\right) \\ e_c &= \Phi \omega \sqrt{2} \sin\left(\omega t - \frac{4\pi}{3}\right) = E \sqrt{2} \sin\left(p\Omega t - \frac{4\pi}{3}\right) \end{aligned}$$

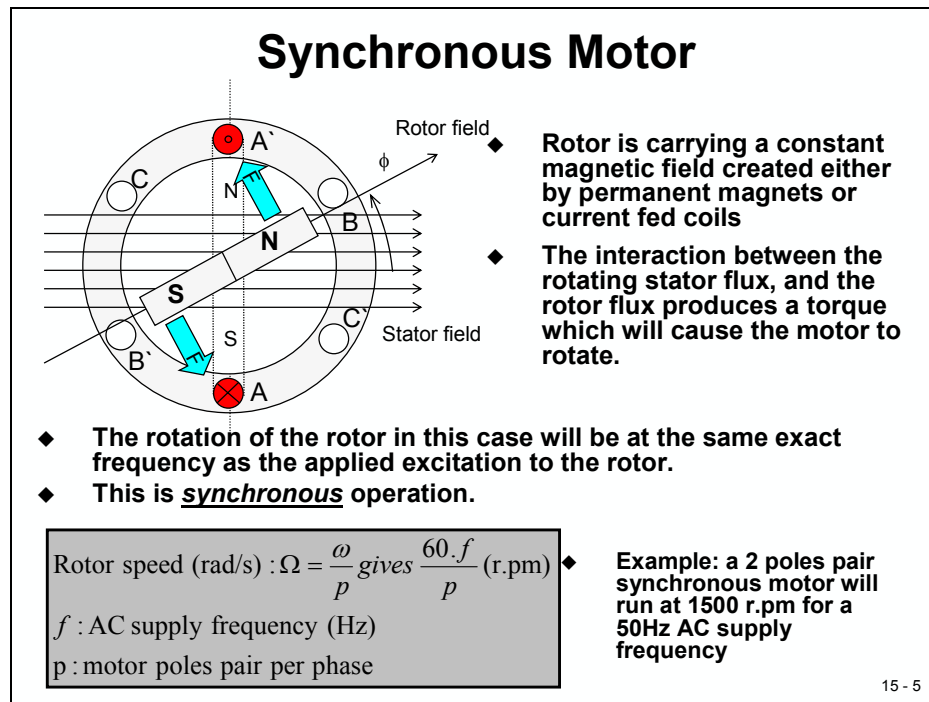
Instead of rotating the magnet in front of windings and inducing a flux variation and a B_{emf} inside the three phases we can turn the principle around:

We can place a sinusoidal source current distribution on the stator that creates a rotating magnetic field. In this case, we apply a rotating force to a magnetic field placed in the center of the stator. This leads us to the principle of the synchronous motor.



Permanent Magnet Synchronous Motor

Synchronous motor construction: Permanent magnets glued or screwed tightly to the rotating axis create the rotor flux (constant). When excited, the stator windings create electromagnetic poles. By controlling the stator currents, we control the stator magnetic field and the revolution of the rotor.



The permanent magnet approach is suitable for synchronous machine ranging up to a few kilo Watts (kW). For higher power ranges, the rotor is composed of windings in which a DC current circulates with the appropriate sequence of North and South poles, in order to get the desired pole pair number.

This action of the rotor chasing after the electromagnet poles on the stator is the fundamental action used in synchronous permanent magnet motors.

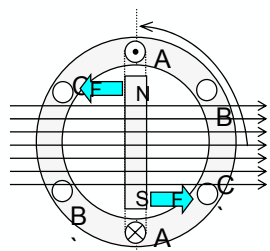
The phase difference between the rotor and the rotating stator field must be controlled to produce torque and to achieve a high degree of efficiency. This implies:

- The rotating stator field must revolve at the same frequency as the rotor permanent magnetic field. If not, the rotor will stop chasing after the stator field.
- The angle between the rotor field and the stator field must be equal to 90° to obtain the highest mutual torque production. This synchronization requires knowing the rotor position in order to generate the right stator field.

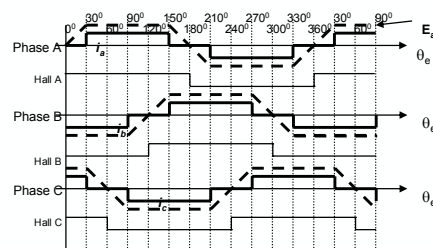
Brushless Direct Current Motor (BLDC)

We can distinguish two types of synchronous motor: Brushless Direct Current motor (BLDC) and Permanent Magnet Synchronous Motor (PMSM). This terminology defines the shape of the E_{mf} of the synchronous motor. Both a BLDC and a PMSM motor can have permanent magnets on the rotor, but different E_{mf} shapes. It is mainly linked to the way of mounting and disposing of the magnets on the rotor. To get the best performances out of the synchronous motor, it is important to identify the type of motor in order to apply the most appropriate type of control scheme.

Synchronous Motors: BLDC and PMSM

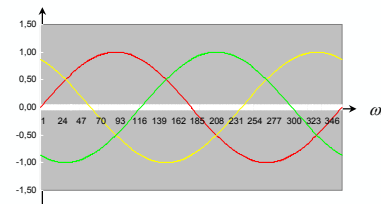


Back EMF of BLDC Motor



- ◆ Both (typically) have permanent-magnet rotor and a wound stator
- ◆ BLDC (Brushless DC) motor is a permanent-magnet brushless motor with trapezoidal back EMF
- ◆ PMSM (Permanent-magnet synchronous motor) is a permanent-magnet brushless motor with sinusoidal back EMF

Back EMF of PMSM



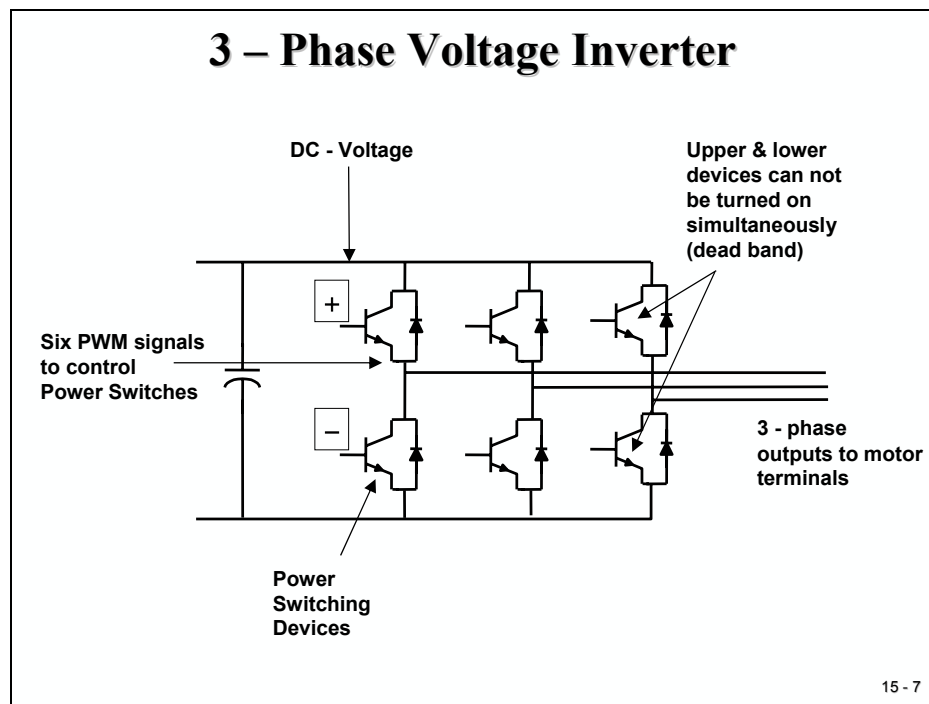
15 - 6

3 – Phase Power Switches

As we saw in the previous basic diagrams, we need to apply three 120° phase shifted excitation signals into the power circuitry of the motor. As you have seen in Module 5 („Event Manager“) a PWM signal can be used to modulate sine wave shaped signals. With three independent switching pattern streams and six power switches, we can deliver the necessary phase voltages to generate the required torque imposed by the load. The goal is to build the correct IGBT's conduction sequences to deliver sine wave shaped currents to the motor to transform it in a mechanical rotation.

This is traditionally achieved by comparing a three-phase sinusoidal waveform with a triangular carrier. In the digital world, on the DSP processor, we compute a sinusoidal command and apply it to the PWM units that generate the appropriate PWM outputs usually connected to gate drivers of the IGBT's from the inverter.

Basically we are “chopping” a DC voltage, carried by the DC bus capacitor, in order to build the appropriate voltage shapes to the stator phases, with the goal of having a good efficiency during this energy conversion process. This is a power electronics concern: we need to minimize the noise introduced by these conducting sequences source of harmonics.



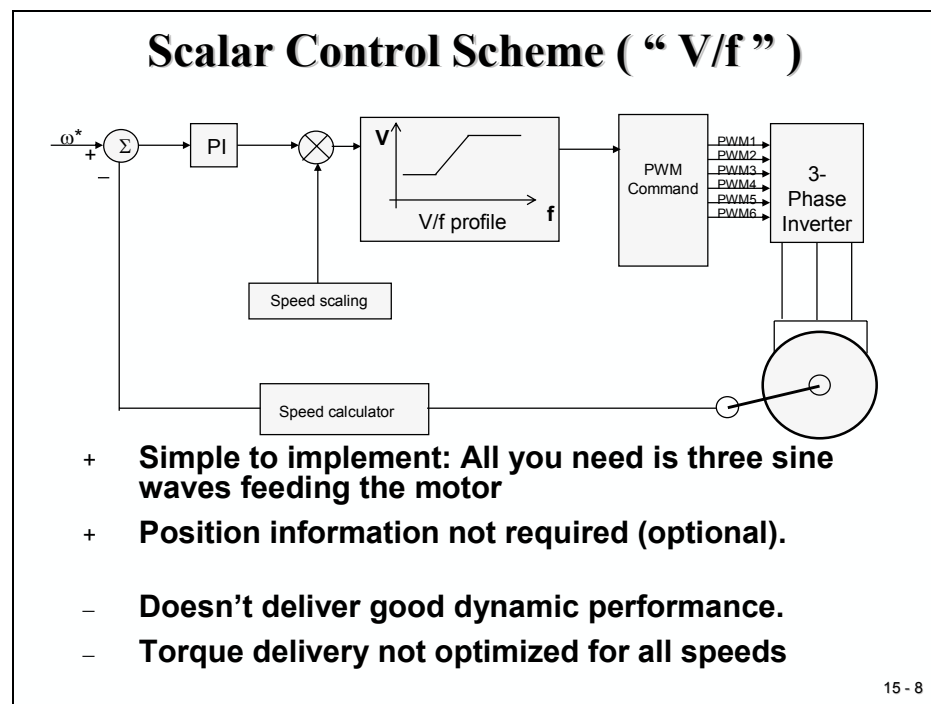
Motor Control Principles

Scalar Control (“V/Hz”)

The V/Hz regulation scheme is the simplest one that can be applied for an **asynchronous motor**. The goal is to work in an area where the rotor flux is constant (Volts proportional to speed).

In practical solutions, the speed sensor is optional as the control is tuned to follow a predefined “speed-profile versus load“- table, assuming the load characteristics are known in advance.

Obviously, this type of control bases itself on the steady electrical characteristic of the machine and assumes that we are able to work with a constant flux on the complete speed range the application targets. This is why this type of control does not deliver a good dynamic performance and a good transient response time; the V/Hz profile is fixed and does not take into account conditions other than those seen in a steady state. The second point is the problem at startup AC induction motors, which cannot deliver high torques at zero speed; in this case, the system cannot maintain a fixed position. In practice for low speed, we need to increase the delivered voltage to the stator compared to the theoretical V/Hz law.



Field Oriented Control (FOC)

Instead of using a pure sine wave shaped modulation of the PWM stage, in recent years the space vector theory has demonstrated some improvements for both the output crest voltage and the harmonic copper loss. The maximum output voltage based on the space vector theory is 1.155 times larger than the conventional sinusoidal modulation. It makes it possible to feed the motor with a higher voltage than the simpler sub-oscillation modulation method. This modulator enables higher torque at high speeds, and a higher efficiency. Torque distortion is also reduced.

The space vector PWM technique implemented into the existing TI DMC library reduces the number of transistor commutations. It therefore improves EMI behavior.

Field Oriented Control (FOC)

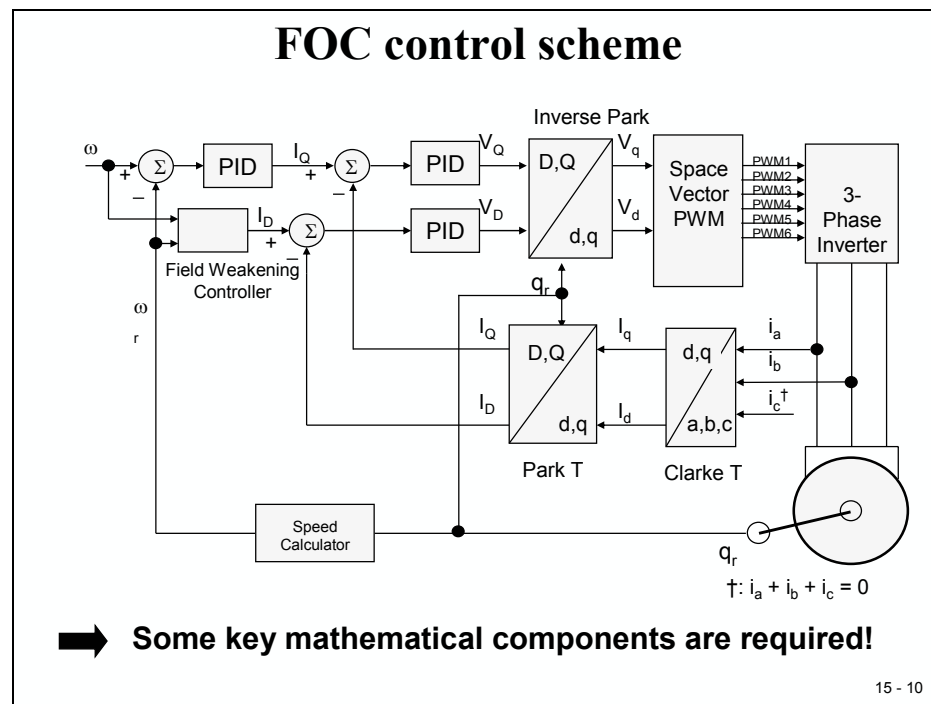
- ◆ **Field Oriented Control (FOC) or Vector Control, is a control strategy for 3-phases induction motors where the torque producing and magnetizing components of the stator flux are separately controlled.**
- ◆ **The approach consists in imitating the DC motors' operation**
- ◆ **FOC will be possible with system information: currents, voltages, flux and speed.**

15 - 9

A typical characteristic of FOC - PWM command strategy is that the envelope of the generated signal is carrying the first and the third harmonics. We can interpret this as a consequence of the special PWM sequence applied to the power inverters. Literature also mentions the third harmonic injection to boost out the performance we get out of the DC bus capacitor. This third-harmonic exists in the phase to neutral voltage but disappears in the phase-to-phase voltage.

FOC Control Scheme

The overall system for implementation of the 3-phase PMSM control is depicted in the next slide. The PMSM is driven by the conventional voltage-source inverter. The C28x is generating six pulse width modulation (PWM) signals by means of space vector PWM technique for six power-switching devices in the inverter. Two input currents of the PMSM (i_a and i_b) are measured from the inverter and they are sent to the C28x via two analog-to-digital converters (ADCs).



Theoretically, the field oriented control for the PMSM drive allows the motor torque be controlled independently with the flux-like DC motor operation. In other words, the torque and flux are decoupled from each other. The rotor position is required for variable transformation from stationary reference frame to synchronously rotating reference frame. As a result of this transformation, the so called Park transformation, the q-axis current will be controlling torque while the d-axis current is forced to zero for a PMSM. Here “d” means direct and “q” means Quadrature.

Therefore, an important key module of this system is the information of rotor position from QEP encoder.

Note that the same control scheme with an additional field weakening controller can be used to control 3-phase asynchronous AC induction motors.

FOC Core Math Operations

PARK Transform

The PARK transform is not something new. This theory has been around for 75 years. As we will see, this technique requires a large amount of mathematical calculations involving in particular matrix multiplications. Thanks to new control processor technologies, it becomes now possible to use this real-time control technique.

Let us consider the voltage vector (V_s) applied to the stator of the three phase machine we are working on (ACI or PMSM).

$$(V_s) = \begin{bmatrix} v_{s1} \\ v_{s2} \\ v_{s3} \end{bmatrix}$$

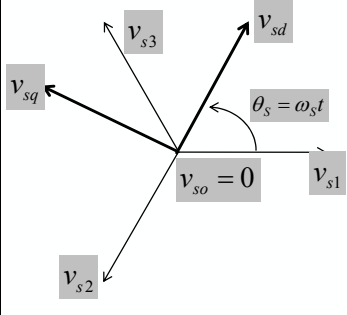
This theory is not limited to sinusoidal distribution and can be applied to any kind of vector. For now, we will consider the general case.

The PARK transform is only a referential change defined as below:

PARK transform (1929):

- ◆ **(V_s): voltage vector applied to motor stator (index s)**
- ◆ **Park transform is a referential change**

$$(V_s) = \begin{bmatrix} v_{s1} \\ v_{s2} \\ v_{s3} \end{bmatrix}$$



$$\begin{bmatrix} v_{s1} \\ v_{s2} \\ v_{s3} \end{bmatrix} = \begin{bmatrix} \cos \theta_s & -\sin \theta_s & 1 \\ \cos(\theta_s - \frac{2\pi}{3}) & -\sin(\theta_s - \frac{2\pi}{3}) & 1 \\ \cos(\theta_s - \frac{4\pi}{3}) & -\sin(\theta_s - \frac{4\pi}{3}) & 1 \end{bmatrix} \begin{bmatrix} v_{sd} \\ v_{sq} \\ v_{so} \end{bmatrix}$$

$$\begin{bmatrix} v_{s1} \\ v_{s2} \\ v_{s3} \end{bmatrix} = [P(\theta_s)] \begin{bmatrix} v_{sd} \\ v_{sq} \\ v_{so} \end{bmatrix} \text{ and } \begin{bmatrix} v_{sd} \\ v_{sq} \\ v_{so} \end{bmatrix} = [P(\theta_s)]^{-1} \begin{bmatrix} v_{s1} \\ v_{s2} \\ v_{s3} \end{bmatrix}$$

15 - 11

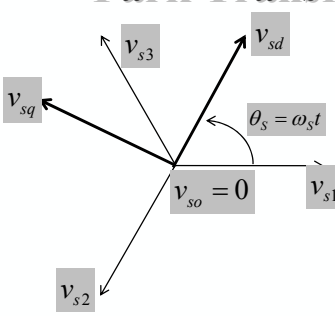
The index “S” indicates that we work with the stator parameters. This referential change transforms the V_s vector in a new vector as written below:

$$\begin{bmatrix} v_{s1} \\ v_{s2} \\ v_{s3} \end{bmatrix} = [P(\theta_s)] \begin{bmatrix} v_{sd} \\ v_{sq} \\ v_{s0} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} v_{sd} \\ v_{sq} \\ v_{s0} \end{bmatrix} = [P(\theta_s)]^{-1} \begin{bmatrix} v_{s1} \\ v_{s2} \\ v_{s3} \end{bmatrix}$$

PARK coordinates will play an important role in the FOC control from a regulation point of view. Instead of using the general PARK transform, literature prefers the normalized PARK transform for which inverted matrix is much easier to calculate and which allows building an orthogonal referential change. The normalized PARK transform is defined as follows:

$$[P(\theta_s)] = \sqrt{\frac{2}{3}} \begin{bmatrix} \cos \theta_s & -\sin \theta_s & \frac{1}{\sqrt{2}} \\ \cos(\theta_s - \frac{2\pi}{3}) & -\sin(\theta_s - \frac{2\pi}{3}) & \frac{1}{\sqrt{2}} \\ \cos(\theta_s - \frac{4\pi}{3}) & -\sin(\theta_s - \frac{4\pi}{3}) & \frac{1}{\sqrt{2}} \end{bmatrix}$$

Park Transform key components



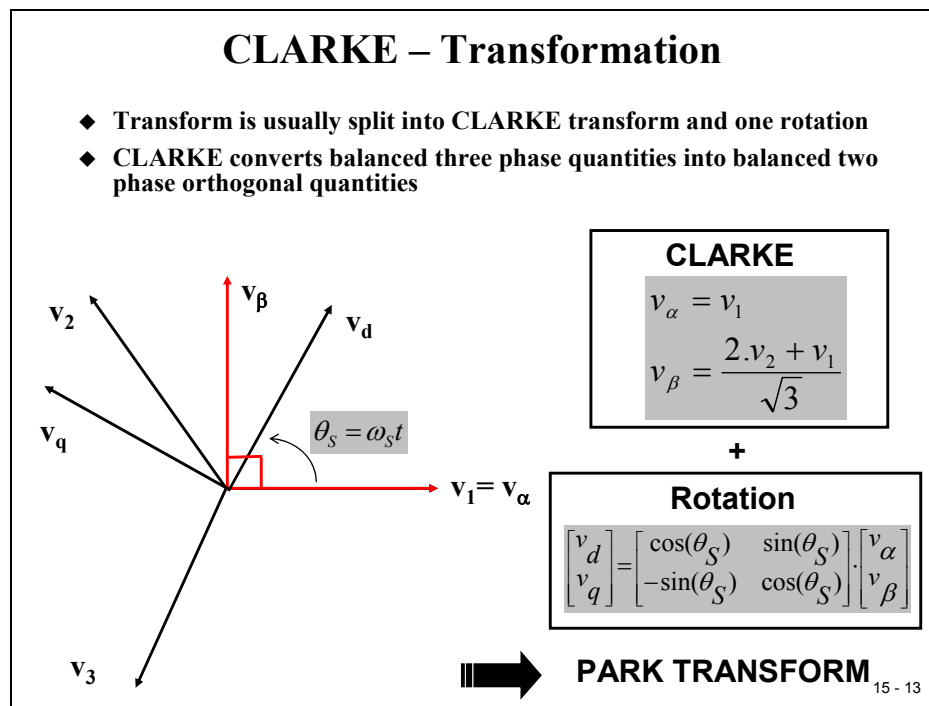
$\begin{cases} \vec{v}_{s1} + \vec{v}_{s2} + \vec{v}_{s3} = \vec{0} \quad (\text{tri-phases balanced system}) \\ \vec{v}_{sd} \cdot \vec{v}_{sq} = 0 \\ \vec{v}_{sd} \cdot \vec{v}_{so} = 0 \\ \vec{v}_{sq} \cdot \vec{v}_{so} = 0 \end{cases}$

- ◆ (v_{sd}, v_{sq}, v_{so}) are called the Park coordinates
- ◆ v_{sd} : direct Park component
- ◆ v_{sq} : squaring Park component
- ◆ v_{so} : homo-polar Park component
- ◆ v_{so} is null for a three-phases balanced system
- ◆ Each pair of components is perpendicular to each other

15 - 12

CLARKE Transform

The normalized PARK can be seen as the result of the combination of the CLARKE transform combined with a rotation. Literature sometimes refers to PARK in this way: this is the case for the TI Digital Motor Control library. This gives an intermediate step that helps to build the regulation scheme.

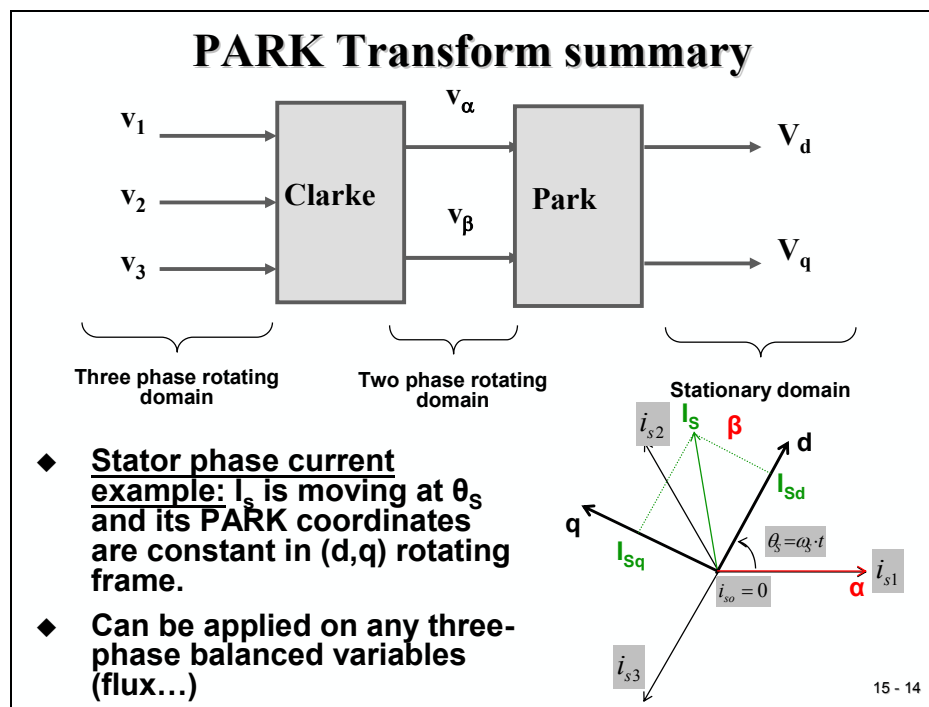


We start from a three-phase balanced system that we first transform in a two-phase balanced system: this is the role of the CLARKE transform that defines a two-phase rotating frame (α , β) that rotates at the speed of the rotating stator magnetic field (ω_s). If we “sit” on the rotating frame (α , β) then we see stationary variables. This is the role of the θ_s angle rotation that allows us to move from the rotating domain to the stationary domain.

PARK Transform Summary

In the next slide, the two modules “PARK” and “CLARKE” are shown in a “one-piece” transform. Texas Instruments DMC library uses two C-callable functions to perform the coordinate transform. The library functions are based on the fixed point math „IQ-Math“.

Combining the CLARKE and PARK transforms as defined above, we move from the three phase rotating domain to the stationary domain: we just need to control DC quantities in real-time.



To drive the power switches with new calculated values we have to re-transform these stationary control values back into the three phase rotating domain. This is done with a similar transform function called “Inverse PARK”, shown in slide 15-10.

The control itself is done with 3 instances of a C-callable function “PID” (see slide 15-10). This function implements a discrete proportional (P)-integral (I)-derivative (D) control scheme with an additional anti-windup feedback. To learn more about the theory refer to file „pid_reg3.pdf“ in the appendix.

Texas Instruments Digital Motor Control Library

Texas Instruments Digital Motor Control (DMC) Library is available free of charge and can be downloaded from the website. It consists of a number of useful functions for motor control applications. Among those functions, there are pure motor control modules (Park and Clark transforms, Space Vector PWM, ...) as well as traditional control modules (PID controller, ramp generator, ...) and peripherals drivers (for PWM, ADC, ...)

Based on this DMC library, Texas Instruments has developed a number of application notes for different types of electrical motors. All applications examples are specially designed for the C2000 platform and come with a working example of the corresponding software, background information and documentation.

One branch of this library is dedicated to the C28x and takes advantage of the 32 Bit IQ-Math data format.

The following slide shows the examples available for the C28x:

Texas Instruments Motor Control Solution

“C2000 - Digital Motor Control Library (DMC)” :

- ◆ **Single Phase ACI Motor Control Using Constant V/Hz**
- ◆ **3-Phase ACI Motor Constant V/Hz Control**
- ◆ **3-Phase ACI Motor Field Oriented Control**
- ◆ **3-Phase Sensored Field Oriented Control (PMSM)**
- ◆ **3-Phase Sensorless Field Oriented Control (PMSM)**
- ◆ **3-Phase Sensored Trapezoidal Control (BLDC)**
- ◆ **3-Phase Sensorless Trapezoidal Control (BLDC)**

15 - 15

In the remaining part of this chapter we will focus on the “C28x & F28x PMSM3_1: 3-phase Sensored Field Oriented Control” – Literature Number SPRC129. The laboratory setup will be based on the following hardware modules:

- TMS320F2812 eZdsp (Spectrum Digital Inc.) with 5V DC power supply
- DMC550 power drive adapter board (Spectrum Digital Inc.),
- A 24V 3-Phase PMSM (Applied Motion Inc.)
- External 24V – DC power supply unit

Digital Motor Control Library (DMC-Lib)

- ◆ **The DMC-Library is a collection of most commonly used algorithms and function blocks for motor control systems**
- ◆ **For every algorithm and function:**
 - **Essential theoretical background information**
 - **Data types for input/output parameters with numerical range and precision**
 - **Function prototypes and calling conventions**
 - **code size (program and data memory)**
 - **Build Level based code examples**

15 - 16

NOTE:

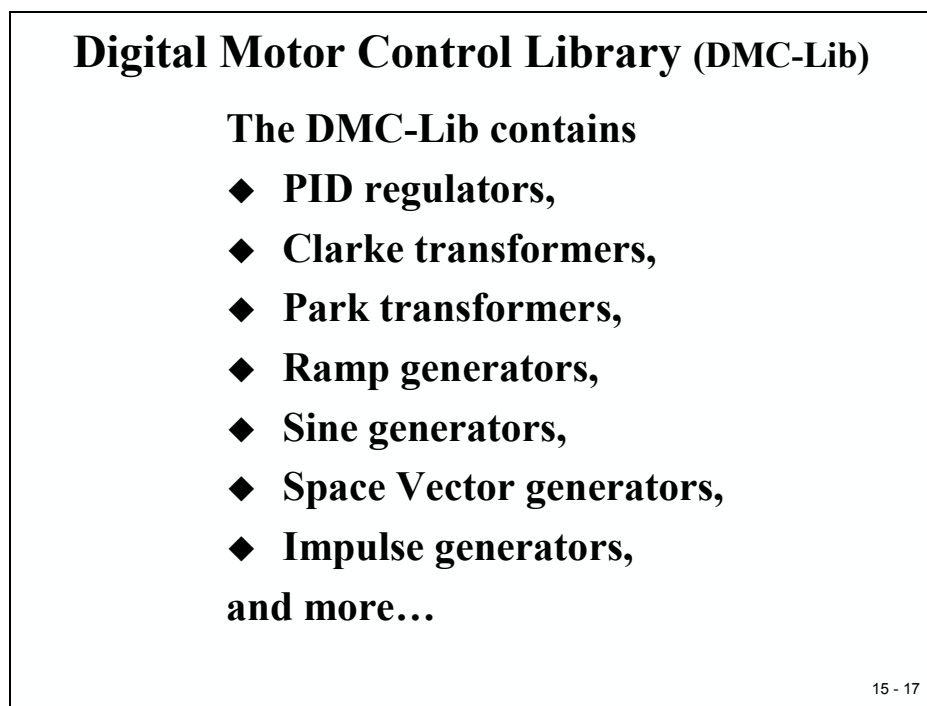
Depending on existing equipment at your university laboratory, you might be able to attend sessions based on other motor types. The procedure for all library solutions will be similar and is based on TI’s modular approach for each of the motor solutions.

To accomplish the lab exercises you will need an additional student user guide, which is tailored to your university lab.

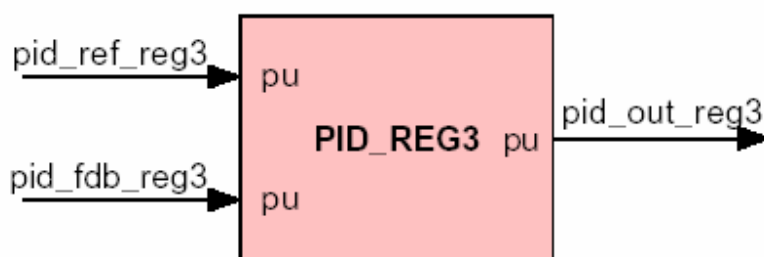
Ask your professor if this optional laboratory exercise is available to you.

Library Modules

The following slide lists all groups of C-callable modules which are part of the library. Each block is explained in detail with its accompanying documentation. After the installation of the library, a separate subfolder (“C:\tides\dmc\c28\lib\dmclib\doc”) includes the documentation.



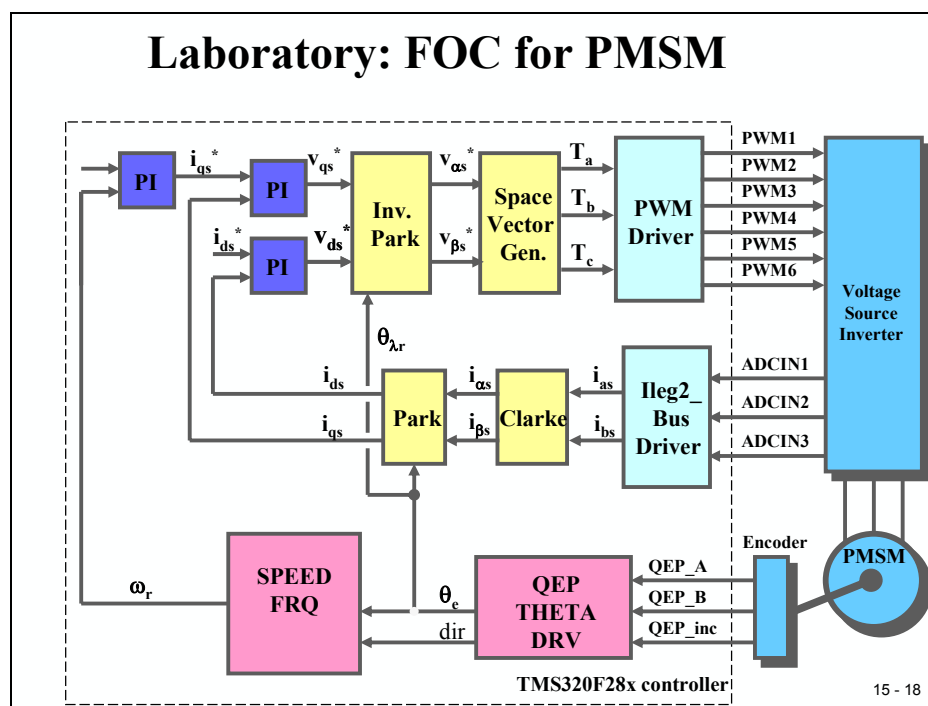
For example, the file “pid_reg3.pdf” explains the interface and the background of the PID-controller:



All functions are coded for 32-Bit variables in IQ-Math-format, which was explained in module 11. Successful completion of module 11 is necessary to be up to speed with the following software modules. All functions are used as instances of a predefined object class, based on a structure definition in a header file.

FOC for PMSM

The following slide is the data flow chart of the complete application for a 3-Phase PMSM. This software project will be implemented step-by-step during the laboratory.



The diagram shows all library modules, which are part of the C28x real time control solution for this type of motor.

It consists of three PID-controller loops that are executed periodically. This period is defined to 20 (in kHz) by parameter “ISR_FREQUENCY” in file “parameter.h” and can be adapted to the needs of the motor, which is used in your laboratory.

PID-controller 1 (top left) controls the rotating speed of the motor, controller 2 (top) is used to optimize the torque and controller 3 is leveling the flux.

The coordinate transform modules, which we discussed previously, form part of the control scheme as well as the QEP – support module to estimate speed and position of the rotor.

Hardware Laboratory Setup

Before we busy ourselves with the software project, let's summarize the hardware equipment that we need to continue:

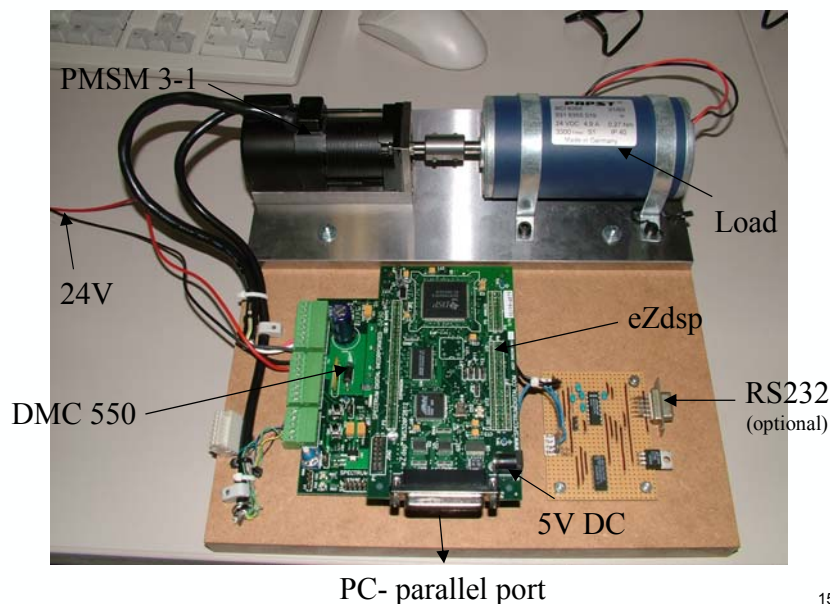
TI Library Solution “PMSM 3-1” (sprc129)

Hardware for Laboratory setup:

- Spectrum Digital eZdsp TMS320F2812
- Spectrum Digital DMC550 drive platform
- 3-phase PMSM with a QEP encoder
 - Applied Motion 40mm Alpha Motor
 - Type: A0100-104-3-100
- 24V DC power supply (DC bus voltage)
- load , e.g. DC Motor as Generator
- PC parallel port to JTAG
- 5V DC (eZdsp)
- RS232 (optional)
- Oscilloscope

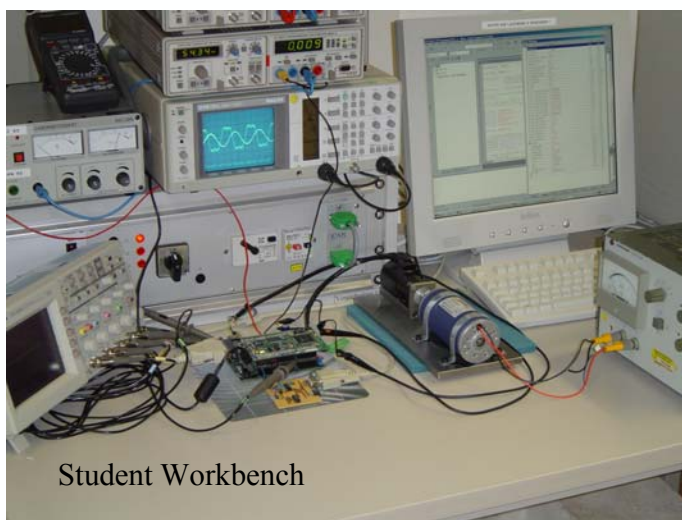
15 - 19

TI Library Solution “PMSM 3-1” (sprc129)



15 - 20

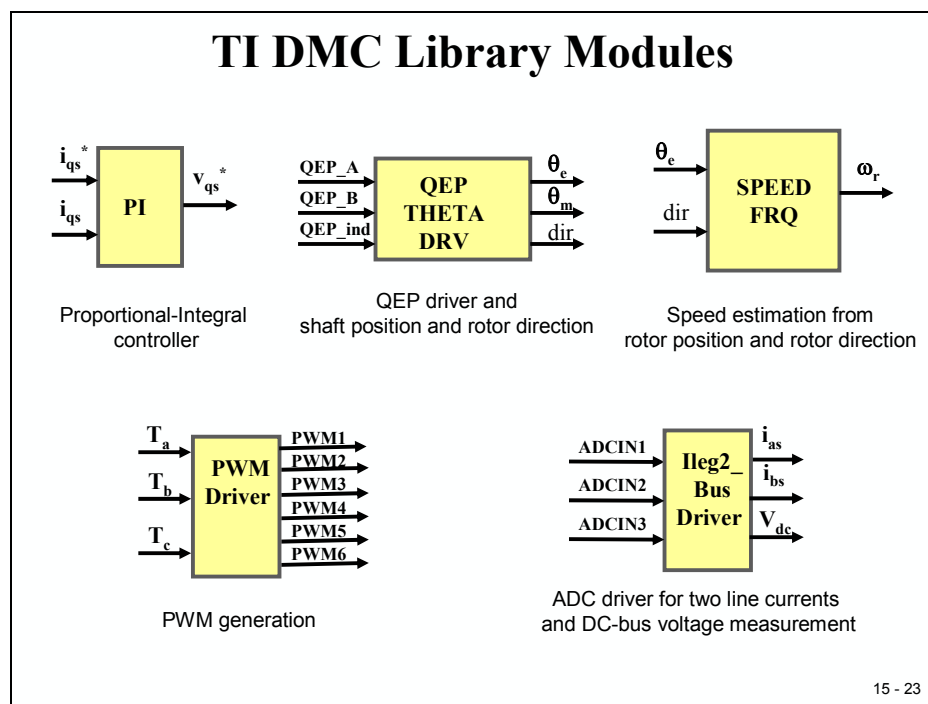
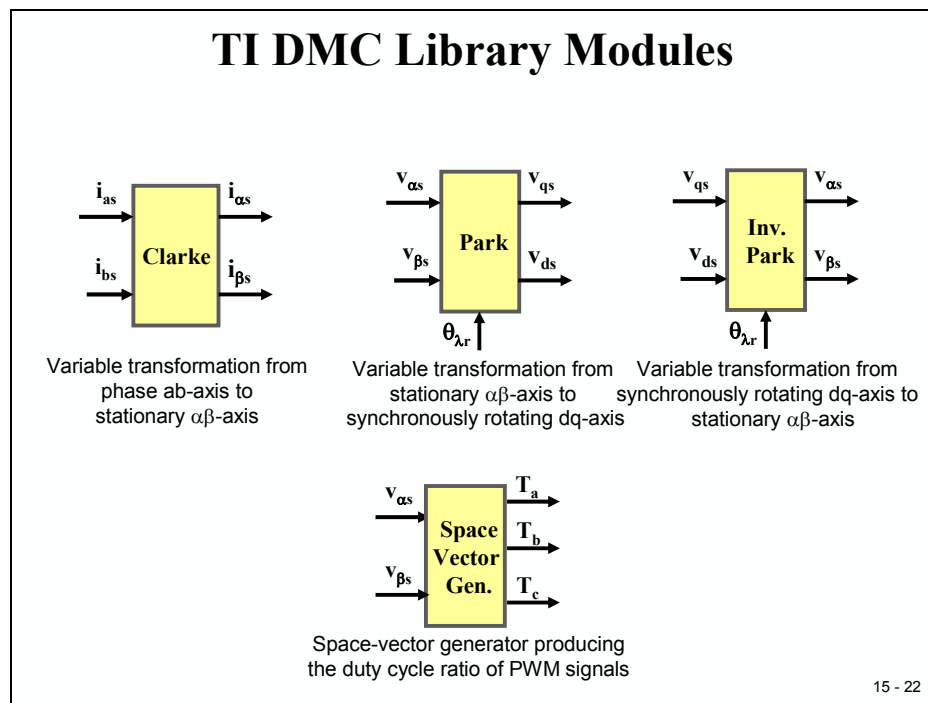
PMSM 3-1 Laboratory



15 - 21

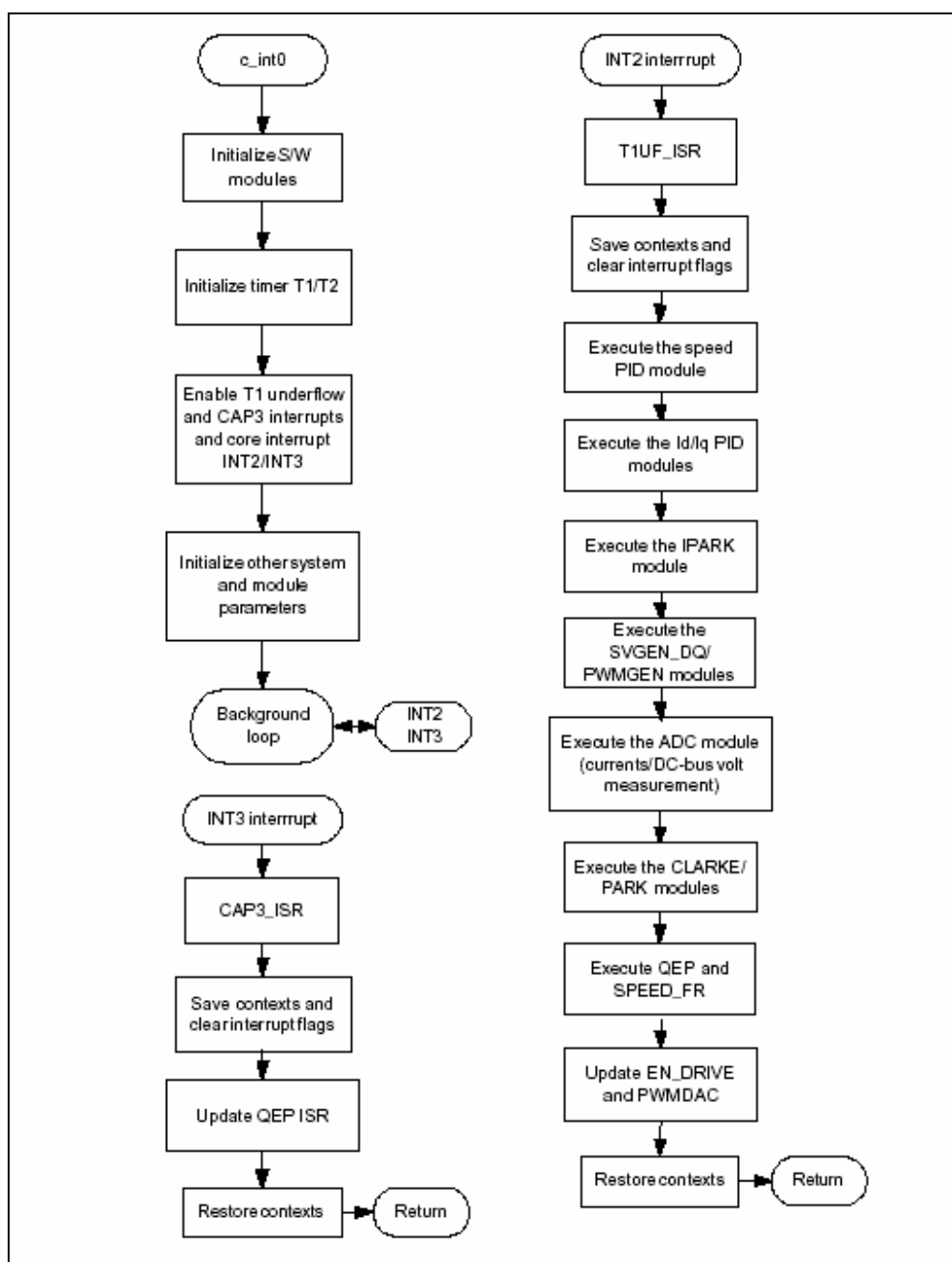
PMSM Library Modules

The next two slides are a summary of all the library modules used in the software project.



PMSM Software Flowchart

The flow of the control software consists of an initialize part (c_int00 plus main) and two interrupt service routines (INT2, INT3). After main() has done the primary initialization, it stays in an endless loop. All further activities are done by the two ISR's. INT3 belongs to the QEP-unit and is called when the rotor passes through zero degrees. INT2 belongs to Timer 1 underflow and executes the control loop periodically.



Lab 15: PMSM control project

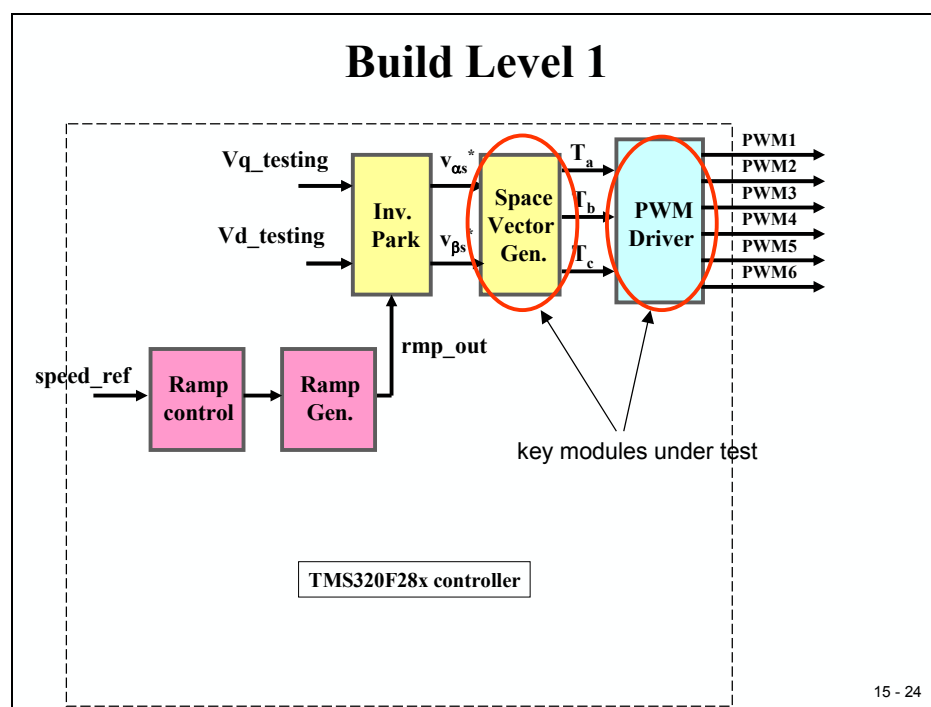
The laboratory experiment is gradually built-up in order that the final system can be confidently operated. Five phases of the incremental system build are designed to verify each of the major software modules used in the system.

Build Level 1

This first level describes the steps for a “minimum” system check-out, which confirms correct operation of system interrupts, the peripheral and target independent I_PARK and SVGEN_DQ modules and the peripheral dependent PWM_DRV module.

Notice that only the x2812 eZdsp is used in this phase. The PMSM and DMC550 are not to be connected yet.

In the **build.h** header file located under *../pmsm3_1/cIQmath/include* directory, select phase 1 incremental build option by setting the parameter “build level” to “level 1”. Use the ‘Rebuild All’ feature of CCS to save the program, compile it and load it to the target.

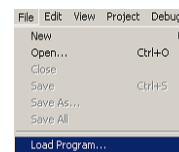


Code Composer Studio with Real Time Mode

1. Load a workspace file '*pmsm3_1.wks*'
2. In *build.h*, `#define BUILDLEVEL LEVEL1`
3. Rebuild all



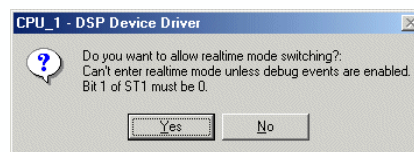
4. Load program to target
(..\pmsm3_1.out)



15 - 25

Code Composer Studio with Real Time Mode

5. In Debug menu, "Reset CPU" and then set "Real-time Mode". Then, click "Yes" when the message box pops up.



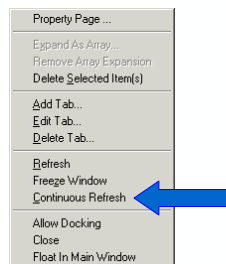
6. Click "Run" icon 

15 - 26

After running and setting real time mode, set variable "enable_flg" to 1 in the Watch Window in order to enable interrupt T1UF. The variable named "isr_ticker" will be increased incrementally, as can be seen in Watch Window to confirm the interrupt working properly.

Code Composer Studio with Real Time Mode

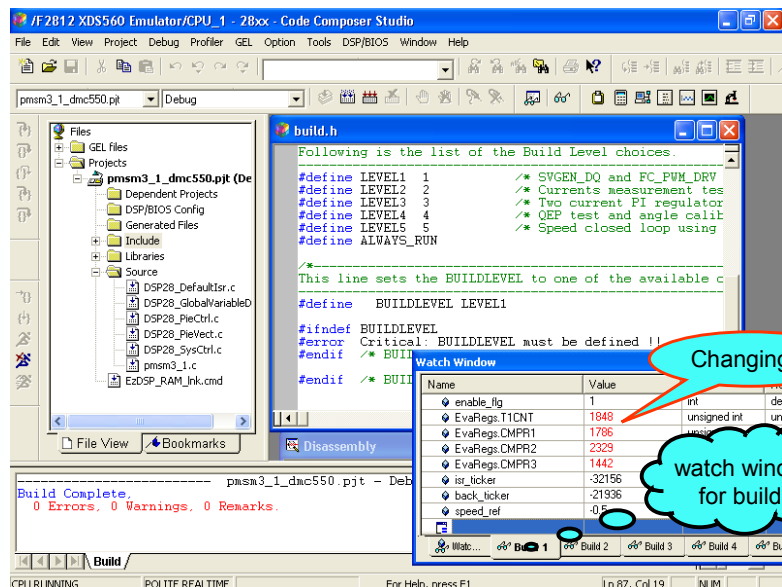
7. Right click on watch window. Then, check “Continuous Refresh”.



8. Set “enable_flg” to 1 in watch window.
(to enable T1UF interrupt and PWM drive on DMC550)

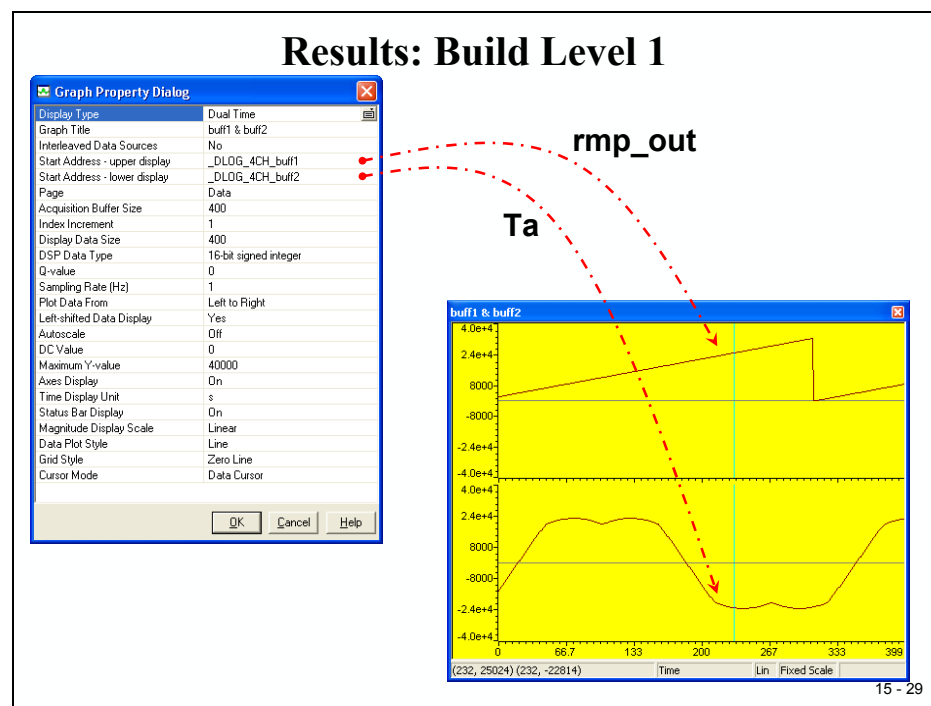
15 - 27

Code Composer Studio Level 1



15 - 28

The speed_ref value is specified to the RAMP_GEN module via RAMP_CNTL module. The I_PARK module is generating the outputs to the SVGEN_DQ module. Three outputs from SVGEN_DQ module are monitored via the PWMDAC module with external low-pass filter and an oscilloscope. The expected output waveform can be seen in the next slide. Waveforms Ta, Tb, and Tc are 120° apart from each other. Specifically, Tb lags Ta by 120° and Tc leads Ta by 120°.

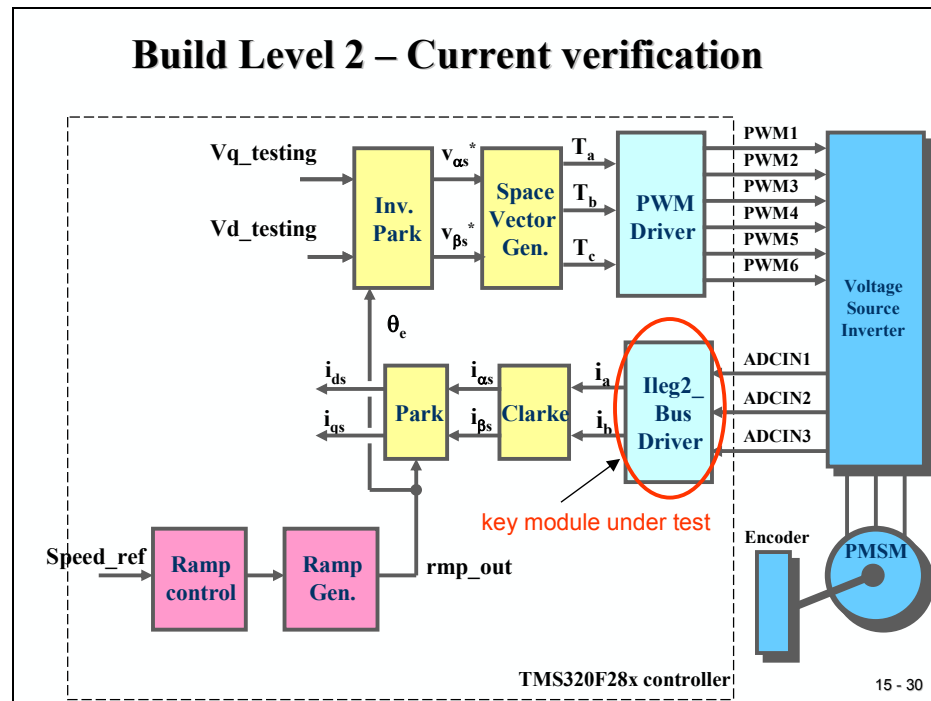


Next, the PWM_DRV module is tested by looking at the six PWM output pins.

Note:

A simple 1st – order low-pass filter RC circuit must be created to visualize the integral of the PWM-signals with an oscilloscope. Ask your laboratory technician about provisions or additional recommendations.

Once the low-pass filter is connected to the PWM pins of the x2812 eZdsp, the filtered version of the PWM signals are monitored by oscilloscope. The waveform shown on the oscilloscope should appear as same as one shown above. It is emphasized that the Ta waveform may be out of phase comparing with the filtered PWM1 signal.



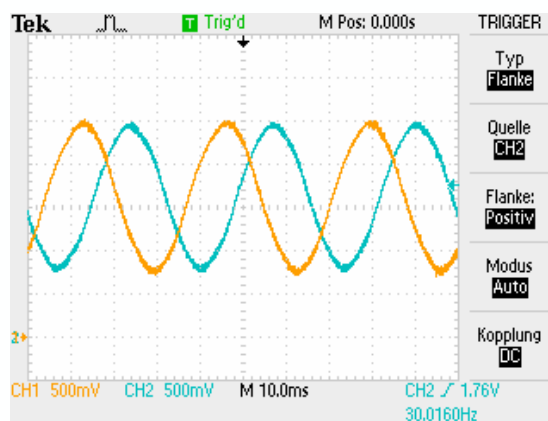
Instructions: Build Level 2

1. Tune the 24V Power Supply to 10 Volts with 1 Amp limit
2. Load a workspace file 'pmsm3_1.wks'
3. In build.h, #define BUILDLEVEL LEVEL2
4. Reset CPU, Compile, Load, start RTM and Run
5. Switch on 24V Power Supply
6. Set variable "enable_flg" to 1 in watch window.
7. Try to change motor speed by setting "speed_ref" (p.u.) in watch window. Then, motor should change its speed accordingly.

15 - 31

Results: Build Level 2

1. PMSM should run open-loop smoothly
2. The currents in the motor phases should be sinusoidal.



15 - 32

The slide above shows the measured signals „ia“ (yellow) and „ib“ (blue) from module „ILEG2_DCBUS_DRV“.

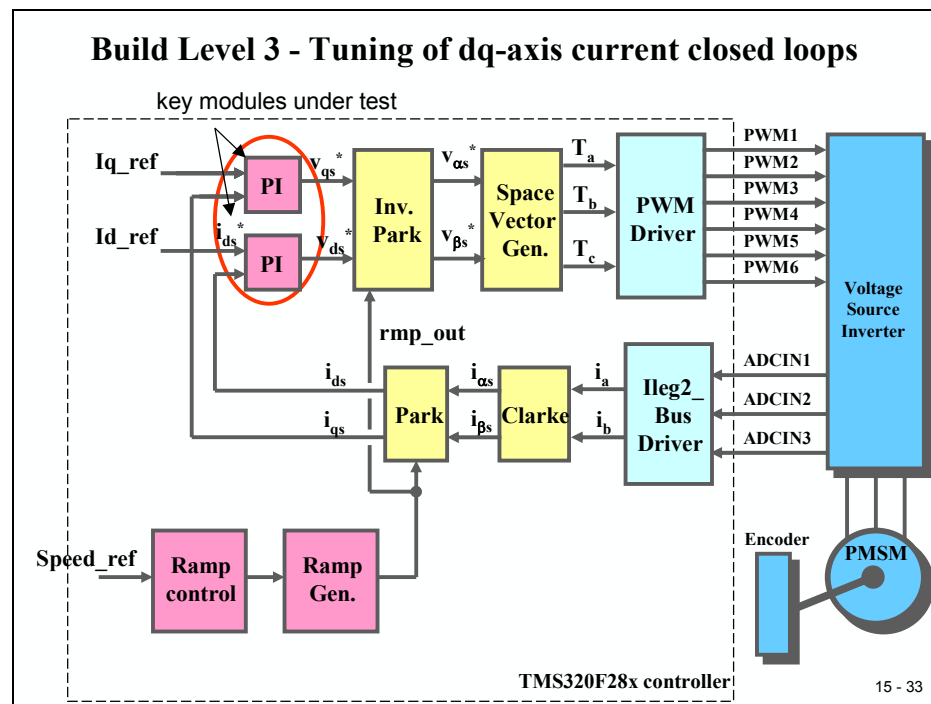
At this stage, the motor should rotate already, but without any control. Therefore we can't connect the load yet. We can modify the speed of the motor by changing variable "speed_ref" between -0.15 (anti-clockwise), 0 (stop) and +0.15 (clockwise) rotation.

With variable "rc1.rmp_dly_max", we can modify the acceleration of the motor with values between 0 (fast) and 100 (slow).

Variable "Vq_testing" will be replaced in the following levels by a control value to control the torque of the motor. Now we can use this variable to experiment with different torque control values.

Build Level 3

At this level, we close the two inner loops of the control scheme by enabling the dq-axis current regulation performed by PID_REG3 modules. To confirm the operation of current regulation, the gains of these two PID controllers are necessarily tuned for proper operation.



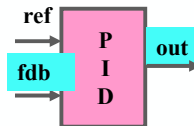
In the **build.h** header file located under `../pmsm3_1/cIQmath/include` directory, select phase 3 incremental build option by setting the build level to level 3. Use the 'Rebuild All' feature of CCS to save the program, compile it and load it to the target.

After running and setting real time mode, set “enable_flg” to 1 in the Watch Window, in order to enable interrupt T1UF. The variable named “isr_ticker” will be incrementally increased as seen in watch windows to confirm the interrupt working properly.

In this build level, the motor is supplied by AC input voltage and the motor current is dynamically regulated by using PID_REG3 module through the park transformation on the motor currents.

Instructions: Build Level 3

1. In build.h, **#define** BUILDLEVEL LEVEL3
2. Compile, Load, start RTM and Run
3. Set “enable_flg” to 1 in watch window.
4. Tune-up the PI
 - Observe dq-axis current regulations at PI inputs (i.e., reference and feedback). For example, “pid1_iq.pid_ref_reg3” and “pid1_iq.pid_fdb_reg3”.
 - Try to change motor speed by setting “speed_ref” (p.u.) in watch window . Then, observe dq-axis current regulations.

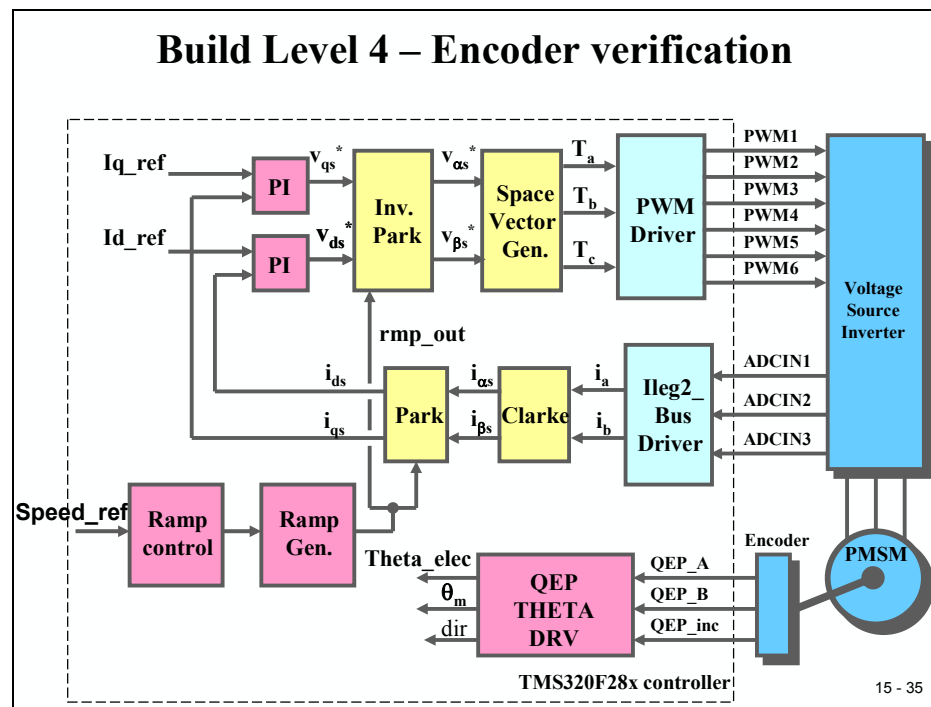


15 - 34

Build Level 4

The objective of this level is to verify the QEP driver and its speed calculation. The number of poles (p) must be set in file “parameter.h” according to the motor used in your laboratory. Next, parameter “mech_scaler” must be set according to the number of encoder-pulses per 360°-rotation:

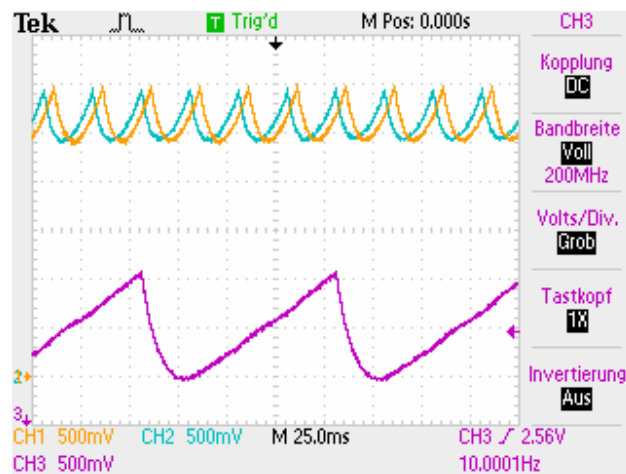
$$\text{mech-scaler} = 1 / \text{encoder-pulses} \quad (\text{in Q30 - Format})$$



Next, we have to adjust the offset angle between the index pulse of the encoder and the physical zero degrees angle of the rotor. At the end of this step, the offset will be stored as parameter “qep1.cal_angle”. With control variable “locktr_flg=0” we can activate the ramp generator in real time mode. If the motor is running, register T2CNT counts the number of pulses since the last index pulse. When we set “Locktr_flg=1”, the ramp is disconnected and the motor stops. T2CNT holds the offset between rotor position zero degrees and the QEP index pulse.

After tuning the QEP, the signals “qep1.theta_elec” and “rg1.rmp_out” should be similar in frequency and amplitude (see slide 15-37).

With the number of pole-pairs $p=4$ the frequency of “qep1.theta_mech” must be exactly $\frac{1}{4}$ of signal “qep1.theta_elec”:



Note: the signals are saw-tooth signals. Due to the low pass filter at the signal output lines the shape is smoothed.

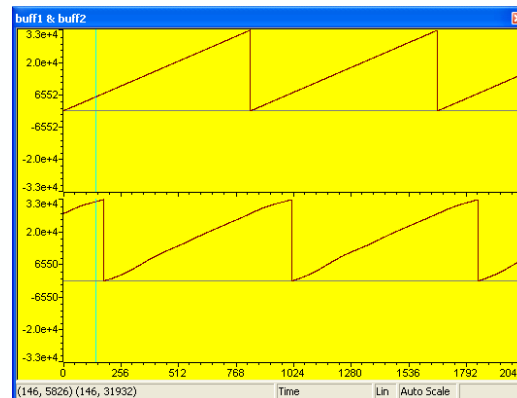
Instructions: Build Level 4

1. In `build.h`, `#define BUILDLEVEL LEVEL4`
2. Compile, Load, start RTM and Run
3. Set the DC-bus to 24 Volts 1 Amp
4. Set “enable_flg” to 1 in watch window.

15 - 36

Results: Build Level 4

◆ Emulated angle VS sensed angle

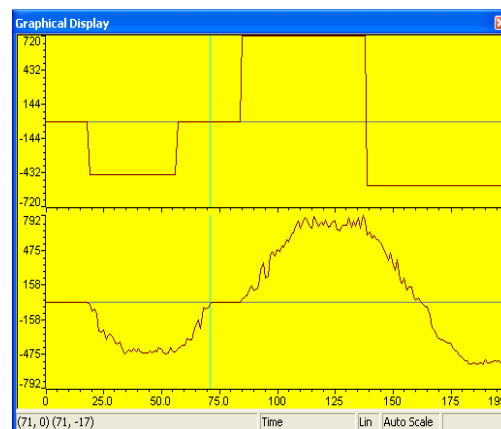


15 - 37

If you compare the changes in reference speed against real motor speed (slide 15-38), you will see that the motor follows any changes sluggishly. The reason is that we do not have closed the speed control loop yet. This will be done in the last build level 5.

Results: Build Level 4

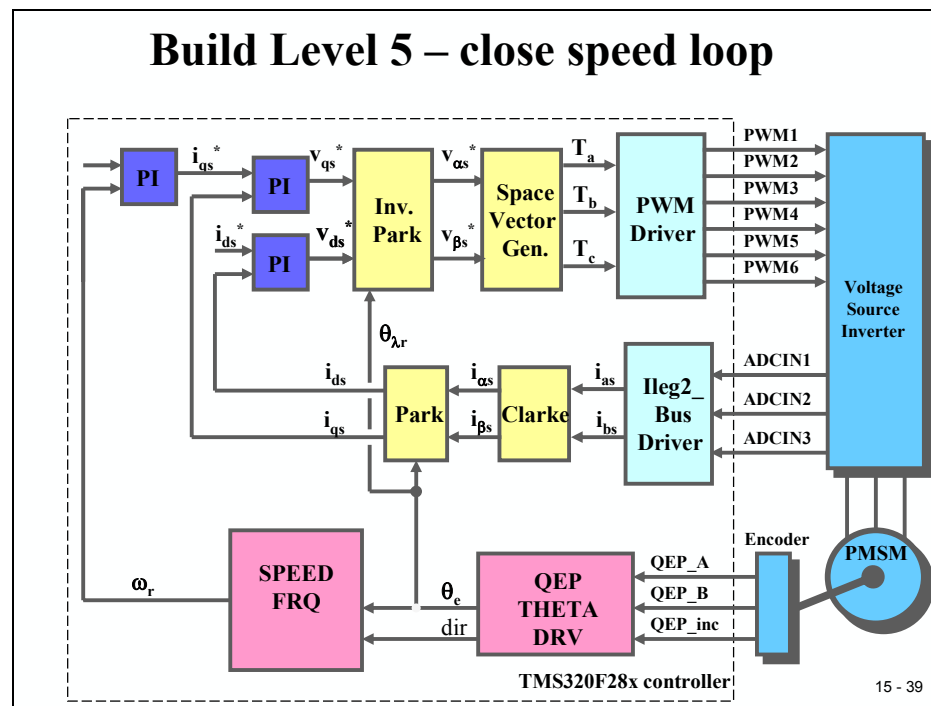
◆ Speed reference VS real speed



15 - 38

Build Level 5

After tuning the Encoder unit, we will use this feedback information as position information in our control loops. The simulated ramp modules from level 1 to 4 are no longer needed. We will also close the speed control loop.

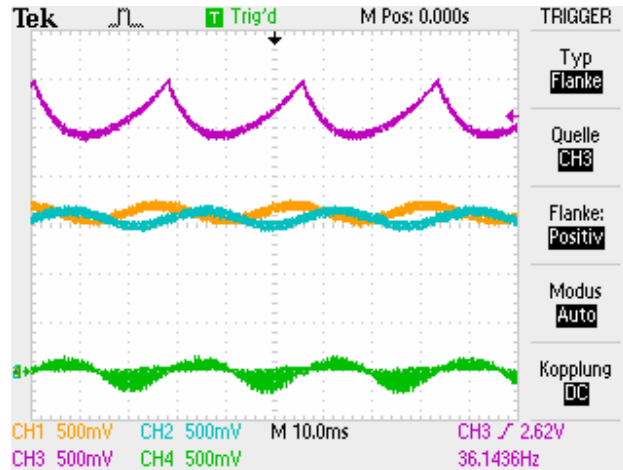


To adjust the speed reference we can use potentiometer R66 of the DMC550. Its analogue value can be connected to ADCINA7 and the converted result can be used for a calculation of signal “speed_ref”. To do this we need to modify the Interrupt Service Routine of EVA-Timer1.

Note:

This last step depends on the hardware setup in your laboratory. Ask your technician if this option is available in your case.

To verify the correct operation of module “speed_ref” we can do some measurements:



The first signal (top) is “speed1.theta_elec”. For this example and knowing that $p=4$ we can calculate the mechanical speed to:

$$\text{Speed1.theta_mech} = (36.1436 \text{ s}^{-1} / 4) * 60 = 542 \text{ rpm}$$

This value should be verified with a stroboscope, if available.

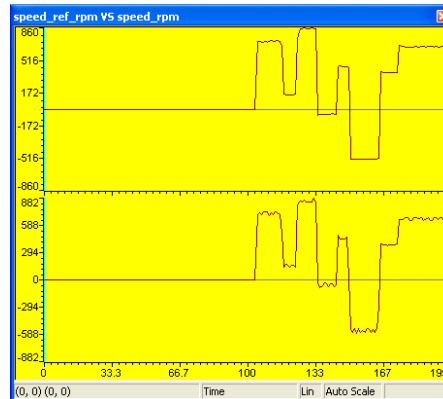
Instructions: Build Level 5

1. In build.h, **#define** BUILDLEVEL LEVEL5
2. Compile, Load, start RTM and Run
3. Set the DC-bus to 24 Volts
4. Set “enable_flg” to 1 in watch window.

If we measure the change in reference speed against real motor speed we should see a much better response of the control system (see slide 15-41).

Results: Build Level 5

- ◆ Fastest response time with the closed loop!



15 - 41

A thesis project at Zwickau University was the implementation of the C28x control scheme for a PMSM in a tram generator.

Application of PMSM 3-1:



img GmbH Nordhausen

15 - 42

This page has intentionally been left blank.

eZdspTM F2812

Technical Reference

eZdsp™ F2812

Technical Reference

506265-0001 Rev. A
May 2002

SPECTRUM DIGITAL, INC.
12502 Exchange Dr., Suite 440 Stafford, TX. 77477
Tel: 281.494.4505 Fax: 281.494.5310
sales@spectrumdigital.com www.spectrumdigital.com

IMPORTANT NOTICE

Spectrum Digital, Inc. reserves the right to make changes to its products or to discontinue any product or service without notice. Customers are advised to obtain the latest version of relevant information to verify data being relied on is current before placing orders.

Spectrum Digital, Inc. warrants performance of its products and related software to current specifications in accordance with Spectrum Digital's standard warranty. Testing and other quality control techniques are utilized to the extent deemed necessary to support this warranty.

Please be aware, products described herein are not intended for use in life-support appliances, devices, or systems. Spectrum Digital does not warrant, nor is it liable for, the product described herein to be used in other than a development environment.

Spectrum Digital, Inc. assumes no liability for applications assistance, customer product design, software performance, or infringement of patents or services described herein. Nor does Spectrum Digital warrant or represent any license, either express or implied, is granted under any patent right, copyright, or other intellectual property right of Spectrum Digital, Inc. covering or relating to any combination, machine, or process in which such Digital Signal Processing development products or services might be or are used.

WARNING

This equipment is intended for use in a laboratory test environment only. It generates, uses, and can radiate radio frequency energy and has not been tested for compliance with the limits of computing devices pursuant to subpart J of part 15 of FCC rules, which are designed to provide reasonable protection against radio frequency interference. Operation of this equipment in other environments may cause interference with radio communications, in which case the user, at his own expense, will be required to take any measures necessary to correct this interference.

TRADEMARKS

eZdsp is a trademark of Spectrum Digital, Inc.

Contents

1	Introduction to the eZdsp™ F2812	1-1
	<i>Provides a description of the eZdsp™ F2812, key features, and board outline.</i>	
1.0	Overview of the eZdsp™ F2812	1-2
1.1	Key Features of the eZdsp™ F2812	1-2
1.2	Functional Overview of the eZdsp™ F2812	1-3
2	Operation of the eZdsp™ F2812	2-1
	<i>Describes the operation of the eZdsp™ F2812. Information is provided on the DSK's various interfaces.</i>	
2.0	The eZdsp™ F2812 Operation	2-2
2.1	The eZdsp™ F2812 Board	2-2
2.1.1	Power Connector	2-3
2.2	eZdsp™ F2812 Memory	2-3
2.2.1	Memory Map	2-4
2.3	eZdsp™ F2812 Connectors	2-5
2.3.1	P1, JTAG Interface	2-6
2.3.2	P2, Expansion Interface	2-6
2.3.3	P3, Parallel Port/JTAG Interface	2-8
2.3.4	P4,P8,P7, I/O Interface	2-8
2.3.5	P5,P9, Analog Interface	2-10
2.3.6	P6, Power Connector	2-12
2.3.7	Connector Part Numbers	2-13
2.4	eZdsp™ F2812 Jumpers	2-13
2.4.1	JP1, XMP/MCn Select	2-14
2.4.2	JP2, Flash Power Supply Select	2-15
2.4.3	JP6, Voltage Control Select	2-15
2.4.4	JP7,JP8,JP11,JP12, Boot Mode Select	2-16
2.4.5	JP9, PLL Select	2-16
2.4.6	JP10, Connect XF Bit to LED DS2	2-17
2.5	LEDS	2-17
2.6	Test Points	2-17
A	eZdsp™ F2812 Schematics	A-1
	<i>Contains the schematics for the eZdsp™ F2812</i>	
B	eZdsp™ F2812 Mechanical Information	B-1
	<i>Contains the mechanical information about the eZdsp™ F2812</i>	

List of Figures

Figure 1-1, Block Diagram eZdsp™ F2812	1-3
Figure 2-1, eZdsp™ F2812 PCB Outline	2-2
Figure 2-2, eZdsp™ F2812 Memory Space	2-4
Figure 2-3, eZdsp™ F2812 Connector Positions	2-5
Figure 2-4, Connector P1 Pin Locations	2-6
Figure 2-5, Connector P2 Pin Locations	2-6
Figure 2-6, Connector P4/P8/P7 Connectors	2-8
Figure 2-7, Connector P5/P9 Pin Locations	2-10
Figure 2-8, Connector P6 Location	2-12
Figure 2-9, eZdsp™ F2812 Power Connector	2-12
Figure 2-10, eZdsp™ F2812 Jumper Positions	2-14

List of Tables

Table 2-1, eZdsp™ F2812 Connectors	2-5
Table 2-2, P1, JTAG Interface Connector	2-6
Table 2-3, P2, Expansion Interface Connector	2-7
Table 2-4, P4/P8, I/O Connectors	2-9
Table 2-5, P7, I/O Connector	2-10
Table 2-6, P5/P9, Analog Interface Connector	2-11
Table 2-7, Connector Part Numbers	2-13
Table 2-8, eZdsp™ F2812 Jumpers	2-13
Table 2-9, JP1, XMP/MCn Select	2-14
Table 2-10, JP2, Flash Programming Voltage Select	2-15
Table 2-11, JP6, Test Mode Select	2-15
Table 2-12, JP7,JP8, JP11, JP12, Boot Mode Select	2-16
Table 2-13, JP9, PLL Disable	2-16
Table 2-14, JP10, Connect XF Bit to LED DS2	2-17
Table 2-15, LEDs	2-17
Table 2-16, Test Points	2-17

About This Manual

This document describes board level operations of the eZdsp™ F2812 based on the Texas Instruments TMS320F2812 Digital Signal Processor.

The eZdsp™ F2812 is a stand-alone module--permitting engineers and software developers evaluation of certain characteristics of the TMS320F2812 DSP to determine processor applicability to design requirements. Evaluators can create software to execute onboard or expand the system in a variety of ways.

Notational Conventions

This document uses the following conventions.

The “eZdsp™ F2812” will sometimes be referred to as the “eZdsp”.

Program listings, program examples, and interactive displays are shown in a special italic typeface. Here is a sample program listing.

equations
!rd = !strobe&rw;

Information About Cautions

This book may contain cautions.

This is an example of a caution statement.

A caution statement describes a situation that could potentially damage your software, hardware, or other equipment. The information in a caution is provided for your protection. Please read each caution carefully.

Related Documents

Texas Instruments TMS320F2801 Users Guide
Texas Instruments TMS320C28XX Fixed Point Assembly Language Users Guide
Texas Instruments TMS320C28XX Fixed Point C Language Users Guide
Texas Instruments TMS320C28XX Code Composer Users Guide

Chapter 1

Introduction to the eZdsp™ F2812

This chapter provides a description of the eZdsp™ for the TMS320F2812 Digital Signal Processor, key features, and block diagram of the circuit board.

Topic	Page
1.0 Overview of the eZdsp™ F2812	1-2
1.1 Key Features of the eZdsp™ F2812	1-2
1.2 Functional Overview of the eZdsp™ F2812	1-3

1.0 Overview of the eZdsp™ F2812

The eZdsp™ F2812 is a stand-alone card--allowing evaluators to examine the TMS320F2812 digital signal processor (DSP) to determine if it meets their application requirements. Furthermore, the module is an excellent platform to develop and run software for the TMS320F2812 processor.

The eZdsp™ F2812 is shipped with a TMS320F2812A DSP. The eZdsp™ F2812 allows full speed verification of F2812A code. Two expansion connectors are provided for any necessary evaluation circuitry not provided on the as shipped configuration.

To simplify code development and shorten debugging time, a C2000 Tools Code Composer driver is provided. In addition, an onboard JTAG connector provides interface to emulators, operating with other debuggers to provide assembly language and 'C' high level language debug.

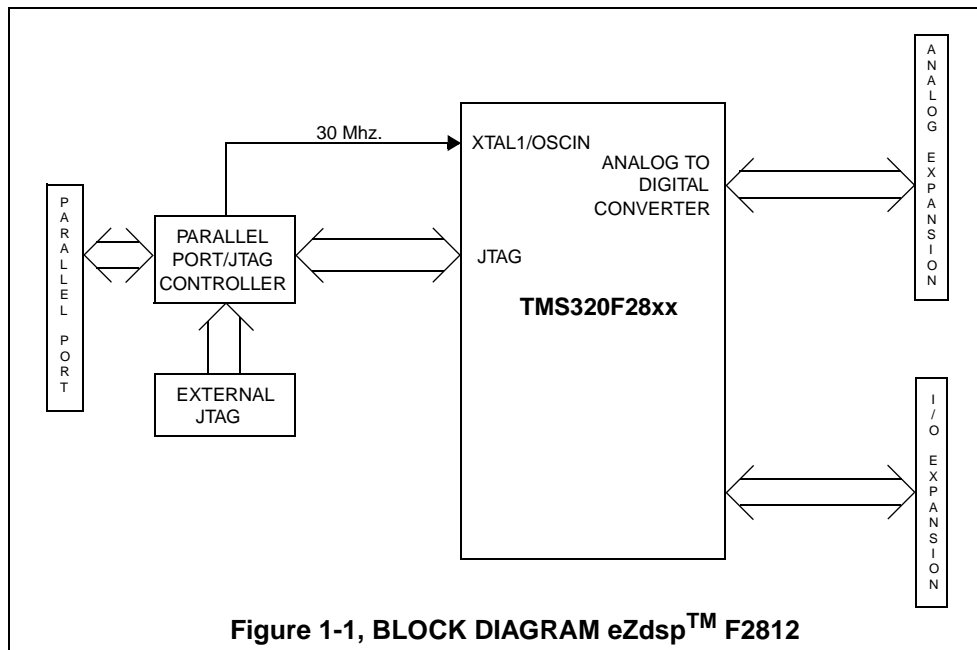
1.1 Key Features of the eZdsp™ F2812

The eZdsp™ F2812 has the following features:

- TMS320F2812A Digital Signal Processor
- 150 MIPS operating speed
- 1K words on-chip RAM
- 128K words on-chip Flash memory
- 64K words off-chip SRAM memory
- Onboard 30 MHz. CPU clock from gate array
- 2 Expansion Connectors (analog, I/O)
- Onboard IEEE 1149.1 JTAG Controller
- 5-volt only operation with supplied AC adapter
- TI F28xx Code Composer Studio tools driver
- On board IEEE 1149.1 JTAG emulation connector

1.2 Functional Overview of the eZdsp™ F2812

Figure 1-1 shows a block diagram of the basic configuration for the eZdsp™ F2812. The major interfaces of the eZdsp are the JTAG interface, and expansion interface.



Chapter 2

Operation of the eZdsp™ F2812

This chapter describes the operation of the eZdsp™ F2812, key interfaces and includes a circuit board outline.

Topic	Page
2.0 The eZdsp™ F2812 Operation	2-2
2.1 The eZdsp™ F2812 Board	2-2
2.1.1 Power Connector	2-3
2.2 eZdsp™ F2812 Memory	2-3
2.2.1 Memory Map	2-4
2.3 eZdsp™ F2812 Connectors	2-5
2.3.1 P1, JTAG Interface	2-6
2.3.2 P2, Expansion Interface	2-6
2.3.3 P3, Parallel Port/JTAG Interface	2-8
2.3.4 P4,P8,P7, I/O Interface	2-8
2.3.5 P5,P9, Analog Interface	2-10
2.3.6 P6, Power Connector	2-12
2.3.7 Connector Part numbers	2-13
2.4 eZdsp™ F2812 Jumpers	2-13
2.4.1 JP1, XMP/MCn Select	2-14
2.4.2 JP2, Flash Power Supply Select	2-15
2.4.3 JP6, Voltage Control Select	2-15
2.4.4 JP7,JP8,JP11,JP12, Boot Mode Select	2-15
2.4.5 JP9, PLL Disable	2-16
2.4.6 JP10, Connect XF Bit to LED DS2	2-17
2.5 LEDs	2-17
2.6 Test Points	2-17

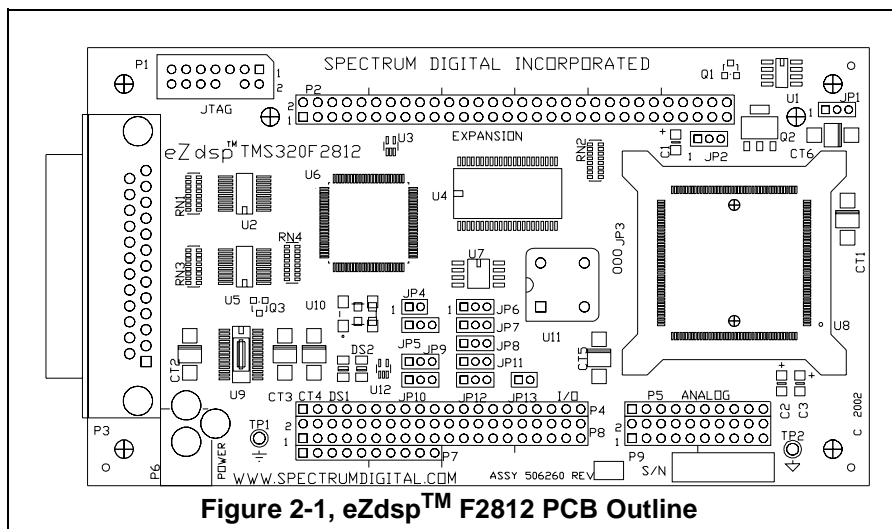
2.0 The eZdsp™ F2812 Operation

This chapter describes the eZdsp™ F2812, key components, and operation. Information on the eZdsp's various interfaces is also included. The eZdsp™ F2812 consists of four major blocks of logic:

- Analog Interface Connector
- I/O Interface Connector
- JTAG Interface
- Parallel Port JTAG Controller Interface

2.1 The eZdsp™ F2812 Board

The eZdsp™ F2812 is a 5.25 x 3.0 inch, multi-layered printed circuit board, powered by an external 5-Volt only power supply. Figure 2-1 shows the layout of the F2812 eZdsp.



2.1.1 Power Connector

The eZdsp™ F2812 is powered by a 5-Volt only power supply, included with the unit. The unit requires 500mA. The power is supplied via connector P6. If expansion boards are connected to the eZdsp, a higher amperage power supply may be necessary. Section 2.3.6 provides more information on connector P6.

2.2 eZdsp™ F2812 Memory

The eZdsp includes the following on-chip memory:

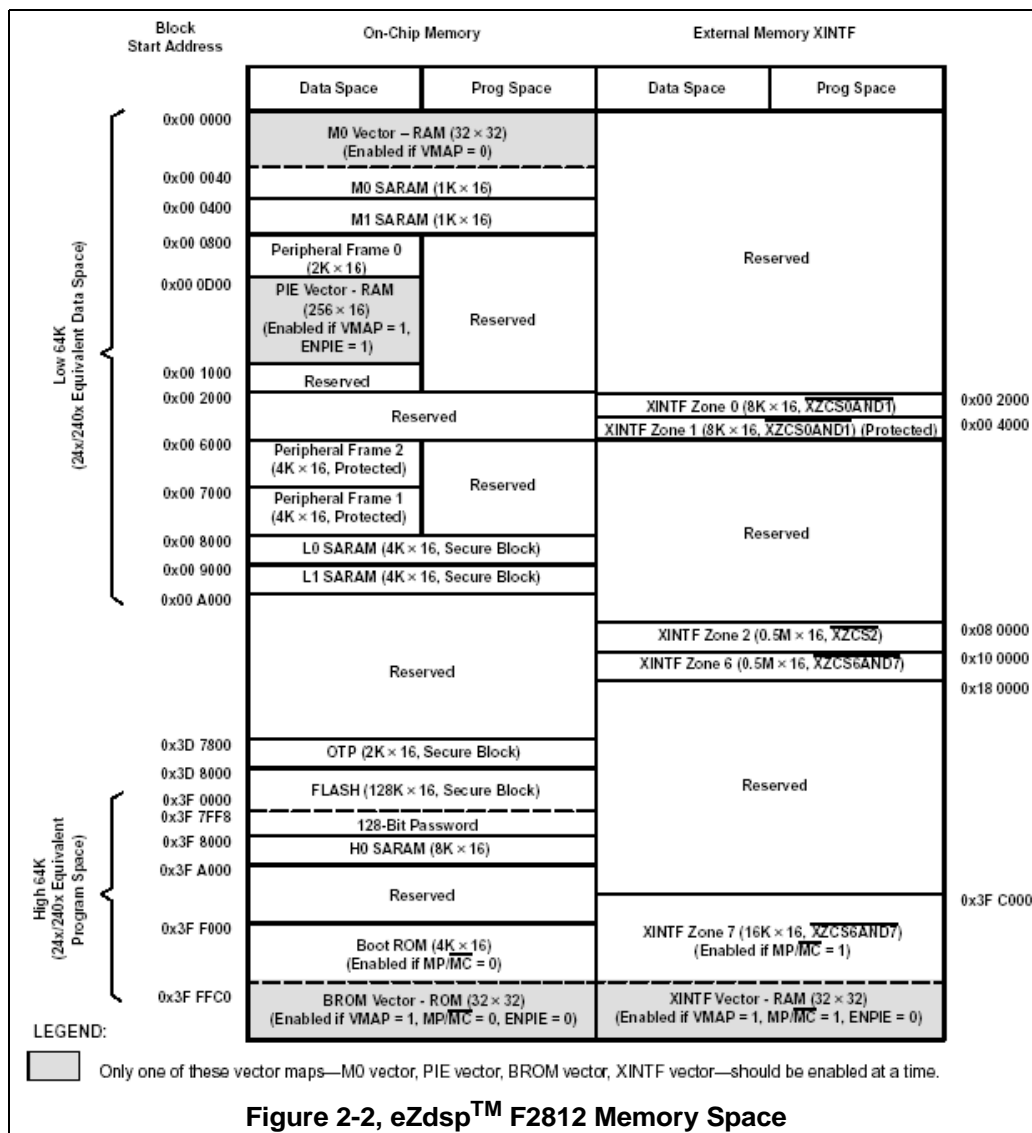
- 128K x 16 Flash
- 2 blocks of 4K x 16 single access RAM (SARAM)
- 1 block of 8K x 16 SARAM
- 2 blocks of 1K x 16 SARAM

In addition 64K x 16 off-chip SRAM is provided. The processor on the eZdsp can be configured for boot-loader mode or non-boot-loader mode.

The eZdsp can load ram for debug or FLASH ROM can be loaded and run. For larger software projects it is suggested to do a initial debug with on eZdsp F2812 module which supports a total RAM environment. With careful attention to the I/O mapping in the software the application code can easily be ported to the F2812.

2.2.1 Memory Map

The figure below shows the memory map configuration on the eZdsp™ F2812.



Note: The on-chip flash memory has a security key which can prevent visibility when enabled,

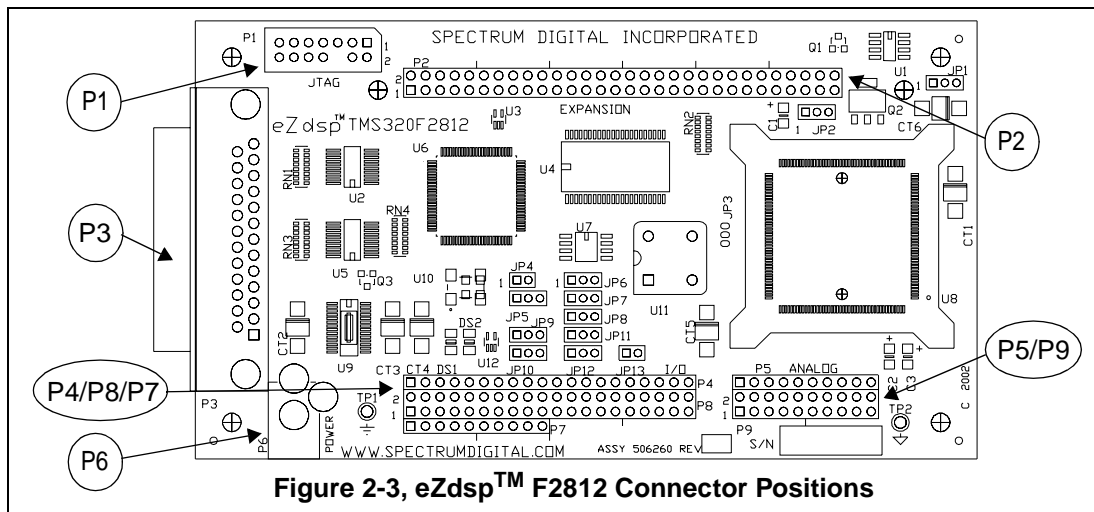
2.3 eZdsp™ F2812 Connectors

The eZdsp™ F2812 has five connectors. Pin 1 of each connector is identified by a square solder pad. The function of each connector is shown in the table below:

Table 1: eZdsp™ F2812 Connectors

Connector	Function
P1	JTAG Interface
P2	Expansion
P3	Parallel Port/JTAG Controller Interface
P4/P8/P7	I/O Interface
P5/P9	Analog Interface
P6	Power Connector

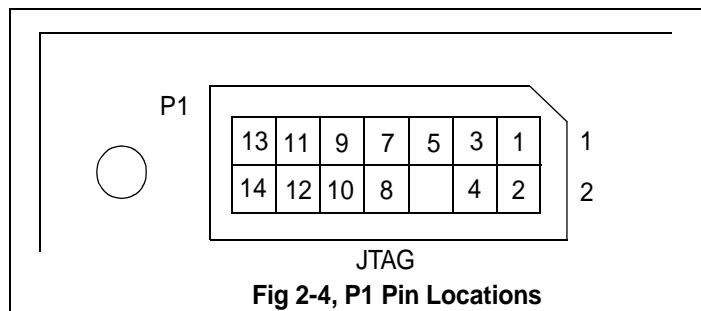
The diagram below shows the position of each connector



2.3.1 P1, JTAG Interface

The eZdsp™ F2812 is supplied with a 14-pin header interface, P1. This is the standard interface used by JTAG emulators to interface to Texas Instruments DSPs.

The positions of the 14 pins on the P1 connector are shown in the diagram below as viewed from the top of the eZdsp.



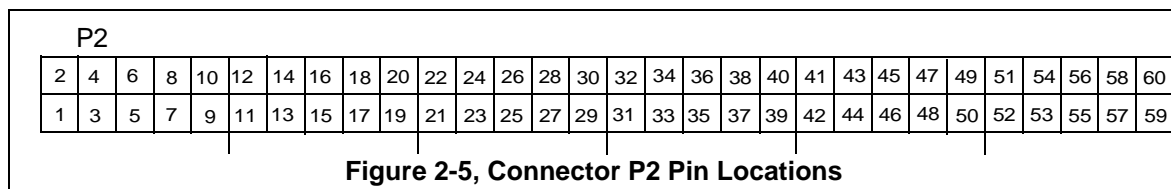
The definition of P1, which has the JTAG signals is shown below.

Table 2: P1, JTAG Interface Connector

Pin #	Signal	Pin #	Signal
1	TMS	2	TRST-
3	TDI	4	GND
5	PD (+5V)	6	no pin
7	TDO	8	GND
9	TCK-RET	10	GND
11	TCK	12	GND
13	EMU0	14	EMU1

2.3.2 P2, Expansion Interface

The positions of the 60 pins on the P2 connector are shown in the diagram below as viewed from the top of the eZdsp.



The definition of P2, which has the I/O signal interface is shown below.

Table 3: P2, Expansion Interface Connector

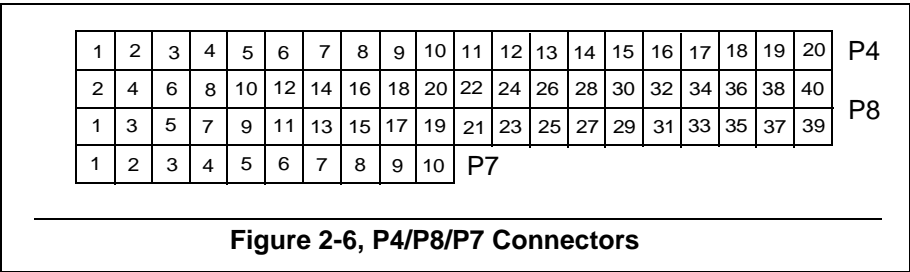
Pin #	Signal	Pin #	Signal
1	+5V	2	+5V
3	XD0	4	XD1
5	XD2	6	XD3
7	XD4	8	XD5
9	XD6	10	XD7
11	XD8	12	XD9
13	XD10	14	XD11
15	XD12	16	XD13
17	XD14	18	XD15
19	XA0	20	XA1
21	XA2	22	XA3
23	XA4	24	XA5
25	XA6	26	XA7
27	XA8	28	XA9
29	XA10	30	XA11
31	XA12	32	XA13
33	XA14	34	XA15
35	GND	36	GND
37	XZCS0AND1n/PSn	38	XZCS2n/DSn
39	XREADY	40	ISn
41	XRnW	42	STRBn
43	XWE	44	XRnD
45	+3.3V/BR-	46	XNMN/INT3
47	XRSn/RSn	48	
49	GND	50	GND
51	GND	52	GND
53	XA16	54	XA17
55	XA18	56	XHOLDn
57	XHOLDAn	58	
59		60	

2.3.3 P3, Parallel Port/JTAG Interface

The eZdsp™ F2812 uses a custom parallel port-JTAG interface device. This device incorporates a standard parallel port interface that supports ECP, EPP, and SPP8/bidirectional communications. The device has direct access to the integrated JTAG interface. Drivers for C2000 Code Composer tools are shipped with the eZdsp modules

2.3.4 P4/P8/P7, I/O Interface

The connectors P4, P8, and P7 present the I/O signals from the DSP. The layout of these connectors are shown below.



The pin definition of P4/P8 connectors are shown in the table below.

Table 4: P4/P8, I/O Connectors

P4 Pin #	P4 Signal	P8 Pin #	P8 Signal	P8 Pin #	P8 Signal
1	+5 Volts	1	+5 Volts	2	+5 Volts
2	XINT2/ADCSOC	3	SCITXDA	4	SCIRXDA
3	MCLKXA	5	XINT1n/XBIO _n	6	CAP1/QEP1
4	MCLKRA	7	CAP2/QEP2	8	CAP3/QEP11
5	MFSXA	9	PWM1	10	PWM2
6	MFSRA	11	PWM3	12	PWM4
7	MDXA	13	PWM5	14	PWM6
8	MDRA	15	T1PWM/T1CMP	16	T2PWM/T2CMP
9		17	TDIRA	18	TCLKINA
10	GND	19	GND	20	GND
11	CAP5/QEP4	21		22	XINT1N/XBIO _n
12	CAP6/QEP12	23	SPISOMA	24	SPIOMA
13	T3PWM/T3CMP	25	SPICLKA	26	SPISTEA
14	T4PWM/T4CMP	27	CANTXA	28	CANRXA
15	TDIRB	29	XCLKOUT	30	PWM7
16	TCLKINB	31	PWM8	32	PWM9
17	XF/XPLLDIS _n	33	PWM10	34	PWM11
18	SCITXDB	35	PWM12	36	CAP4/QEP3
19	SCIRXDB	37	T1CTRIP/PDPINT	38	T3CTRIP/PDPINTB _n
20	GND	39	GND	40	GND

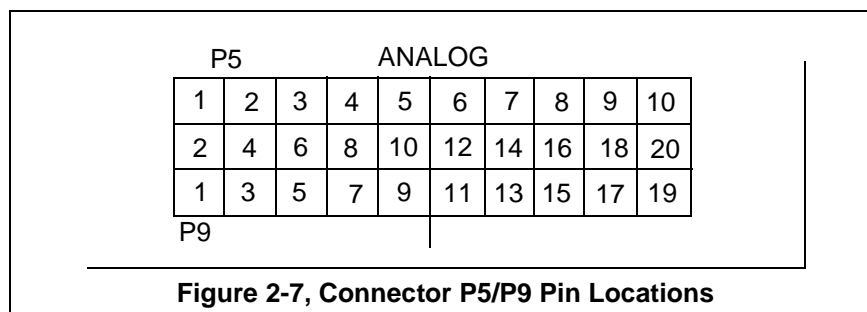
The pin definition of P7 connector is shown in the table below.

Table 5: P7, I/O Connector

P7 Pin #	P7 Signal
1	C1TRIPn
2	C2TRIPn
3	C3TRIPn
4	T2CTRIPn/EVASOCn
5	C4TRIPn
6	C5TRIPn
7	C6TRIPn
8	T4REPn/EVBSOCn
9	
10	GND

2.3.5 P5/P9, Analog Interface

The position of the 30 pins on the P5/P9 connectors are shown in the diagram below as viewed from the top of the eZdsp.



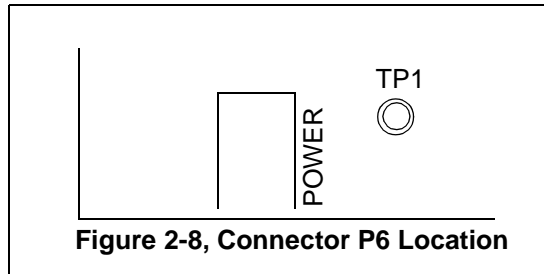
The definition of P5/P9 signals are shown in the table below.

Table 6: P5/P9, Analog Interface Connector

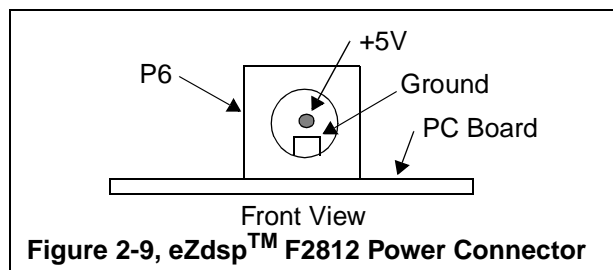
P5 Pin #	Signal	P9 Pin #	Signal	P9 Pin #	Signal
1	ADCINB0	2	GND	2	ADCINA0
2	ADCINB1	4	GND	4	ADCINA1
3	ADCINB2	6	GND	6	ADCINA2
4	ADCINB3	8	GND	8	ADCINA3
5	ADCINB4	10	GND	10	ADCINA4
6	ADCINB5	12	GND	12	ADCINA5
7	ADCINB6	14	GND	14	ADCINA6
8	ADCINB7	16	GND	16	ADCINA7
9	ADCREFM	18	GND	18	VREFLO
10	ADCREFP	20	GND	20	

2.3.6 P6, Power Connector

Power (5 volts) is brought onto the eZdsp™ F2812 via the P6 connector. The connector has an outside diameter of 5.5 mm. and an inside diameter of 2 mm. The position of the P6 connector is shown below.



The diagram of P6, which has the input power is shown below.



2.3.7 Connector Part Numbers

The table below shows the part numbers for connectors which can be used on the eZdsp™ F2812. Part numbers from other manufacturers may also be used.

Table 7: eZdsp™ F2812 Suggested Connector Part Numbers

Connector	Male Part Numbers	Female Part Numbers
P1	SAMTEC TSW-1-10-07-G-T	SAMTEC SSW-1-10-01-G-T
P2	SAMTEC TSW-1-20-07-G-T	SAMTEC SSW-1-20-01-G-T

*SSW or SSQ Series can be used

2.4 eZdsp™ F2812 Jumpers

The eZdsp™ F2812 has 9 jumpers available to the user which determine how features on the eZdsp™ F2812 are utilized. The table below lists the jumpers and their function. The following sections describe the use of each jumper.

Table 8: eZdsp™ F2812 Jumpers

Jumper #	Size	Function	Position As Shipped From Factory
JP1	1 x 3	XMP/MCn	2-3
JP2	1 x 3	Flash Power Supply	1-2
JP6	1 x 3	Test Mode Select	2-3
JP7	1 x 2	Boot Mode 3	2-3
JP8	1 x 3	Boot Mode 2	2-3
JP9	1 x 3	PLL Disable	1-2
JP10	1 x 3	Connect XF to LED DS2	1-2
JP11	1 x 3	Boot Mode 1	1-2
JP12	1 x 3	Boot Mode 0	2-3

WARNING!

Unless noted otherwise, all 1x3 jumpers must be installed in either the 1-2 or 2-3 position

The diagram below shows the positions of the seven jumpers on the eZdsp™ F2812.

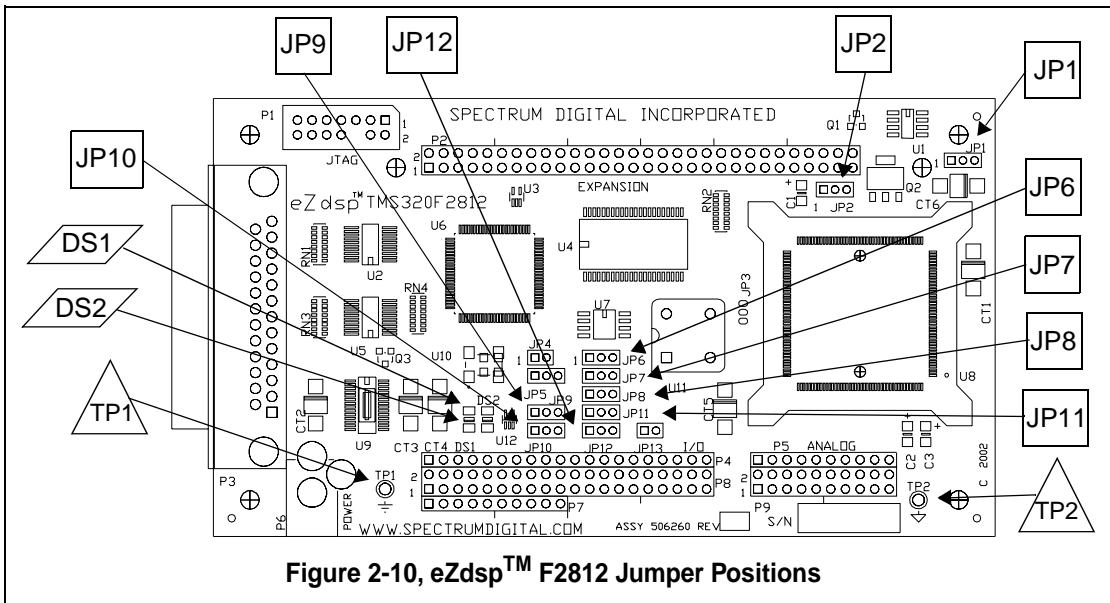


Figure 2-10, eZdsp™ F2812 Jumper Positions

2.4.1 JP1, XMP/MCn Select

Jumper JP1 is used to select the XMP/MCn option. The 1-2 selection allows the DSP to operate in the Microcontroller mode. The 2-3 selection allow the DSP to operate in the Microprocessor mode. The positions are shown in the table below.

Table 9: JP1, XMP/MCn Select

Position	Function
1-2	Microcontroller mode
2-3 *	Microprocessor mode

* as shipped from factory

2.4.2 JP2, Flash Power Supply

Jumper JP2 is used to supply voltage to the DSP for programming the on-chip Flash memory. This jumper is always in the 1-2 position. The is shown in the table below.

Table 10: JP2, Flash Programming Voltage Select

Position	Function
1-2 *	Programming voltage supplied to DSP
2-3	Programming voltage not supplied to DSP

* always in this position

2.4.3 JP6, Test Mode Select

Jumper JP2 is used to determine if the eZdsp will operate in Test mode or User mode. The 1-2 selection puts the eZdsp in Test Mode. If position 2-3 is selected the eZdsp is in the User Mode. The 2-3 position should **always** be used, do not change. The positions are shown in the table below.

Table 11: JP6, Test Mode Select

Position	Function
1-2	Test mode
2-3 *	User Mode

* always use this position

2.4.4 JP7, JP8, JP11, JP12, Boot Mode Select

Jumpers JP7, JP8, JP11, JP12 are used to determine what mode the DSP will use for bootloading on power up. The options are shown in the table below.

Table 12: JP7, JP8, JP11, JP12, Boot Mode Select

JP7, BOOT3 SCITXDA	JP8, BOOT2 MDXA	JP8, BOOT2 MDXA	JP8, BOOT2 MDXA	MODE
1	X	X	X	FLASH
0	1	X	X	SPI
0	0	1	1	SCI
0	0	1	0	H0 *
0	0	0	1	OTP
0	0	0	0	PARALLEL

* factory default

2.4.5 JP9, PLL Disable

Jumper JP9 is used to enable/disable the use of the Phase Lock Loop (PLL) logic on the DSP. The selection of the 1-2 position enables the use of the PLL. If the 2-3 position is used the PLL is disabled. The positions are shown in the table below.

Table 13: JP9, PLL Disable

Position	Function
1-2 *	PLL Enabled
2-3	PLL disabled

* as shipped from the factory

2.4.6 JP10, Connect XF Bit to LED DS2 Select

Jumper JP10 is used to determine if the XF bit from the processor is connected to LED DS2. If the 1-2 position is selected the XF bit is connected to LED DS2 through a buffer. The 2-3 selection disconnects the XF bit from driving LED DS2. The table below shows these options.

Table 14: JP10, Connect XF Bit to LED DS2

Position	Function
1-2 *	XF bit connected to LED DS2
2-3	XF bit not connected to LED DS2

* as shipped from the factory

2.5 LEDs

The eZdsp™ F2812 has two light-emitting diodes. DS1 indicates the presence of +5 volts and is normally 'on' when power is applied to the board. DS2 is under software control and is tied to the XF pin on the DSP through a buffer. These are shown in the table below:

Table 15: LEDs

LED #	Color	Controlling Signal
DS1	Green	+5 Volts
DS2	Green	XF bit (XF high = on)

2.6 Test Points

The eZdsp™ F2812 has two test points. The signals they are tied to are shown in the table below.

Table 16: Test Points

Test Point	Signal
TP1	Ground
TP2	Analog Ground

Appendix A

eZdsp™ F2812

Schematics

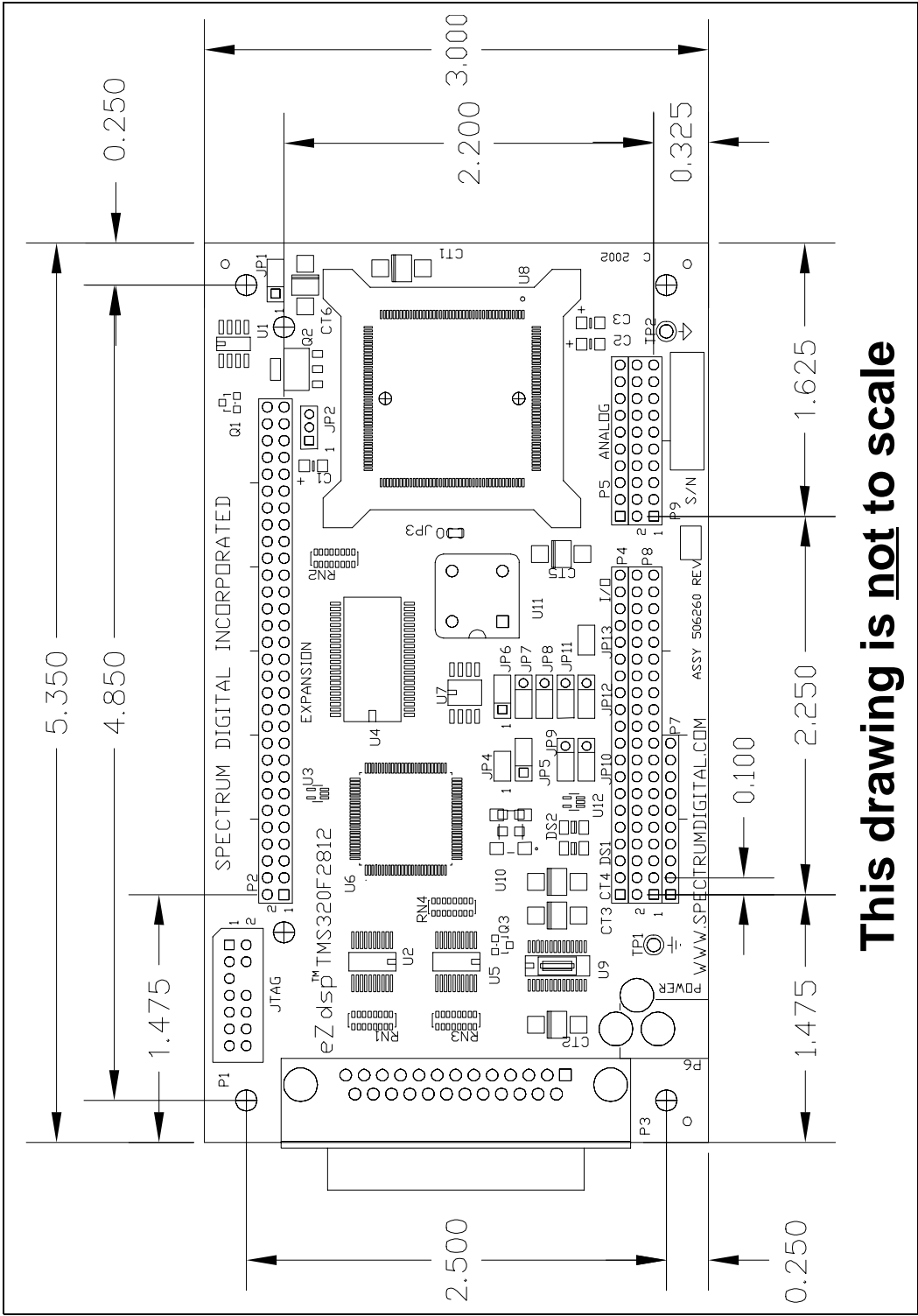
The schematics for the eZdsp™ F2812 can be found on the CD-ROM that accompanies this board. The schematics were drawn on ORCAD.

Appendix B

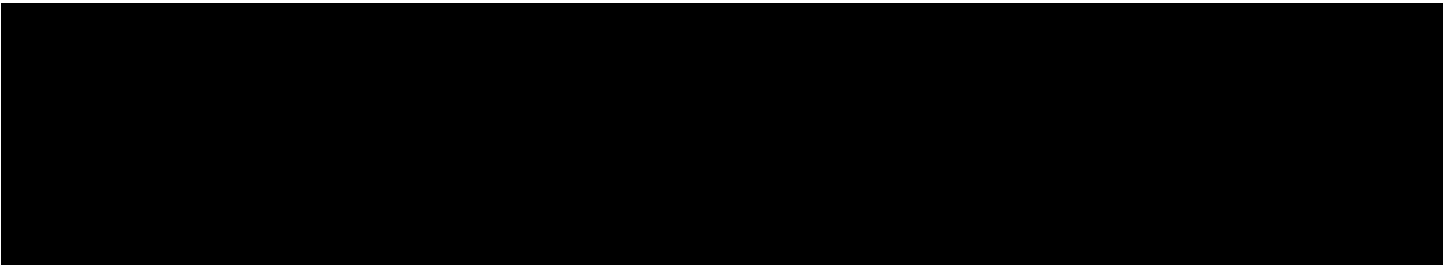
eZdsp™ LF2401

Mechanical Information

This appendix contains the mechanical information about the eZdsp™ LF2401



This drawing is not to scale



Printed in U.S.A., May 2002
506265-0001 Rev. A