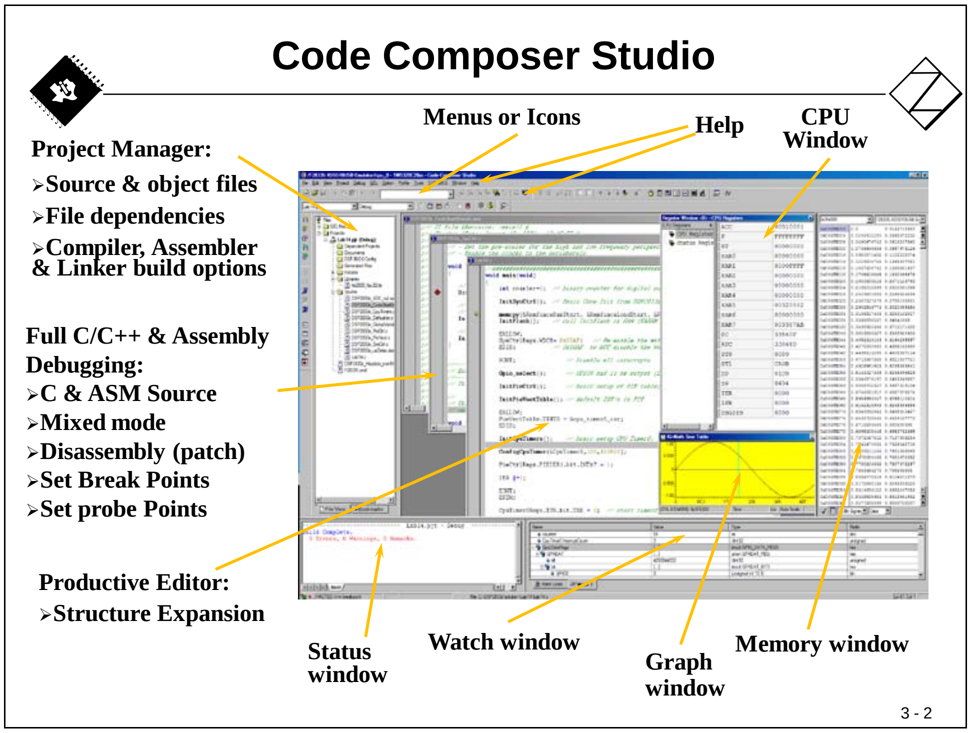
Program Development Tools

Introduction

The objective of this module is to understand the basic functions of the Code Composer Studio® Integrated Design Environment (IDE) for the C2000 Family of Texas Instruments Digital Signal Processors and Microcontrollers. 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.



Code Composer Studio

Project Manager:

- **Source & object files**
- **File dependencies**
- **Compiler, Assembler & Linker build options**

Full C/C++ & Assembly Debugging:

- **C & ASM Source**
- **Mixed mode**
- **Disassembly (patch)**
- **Set Break Points**
- **Set probe Points**

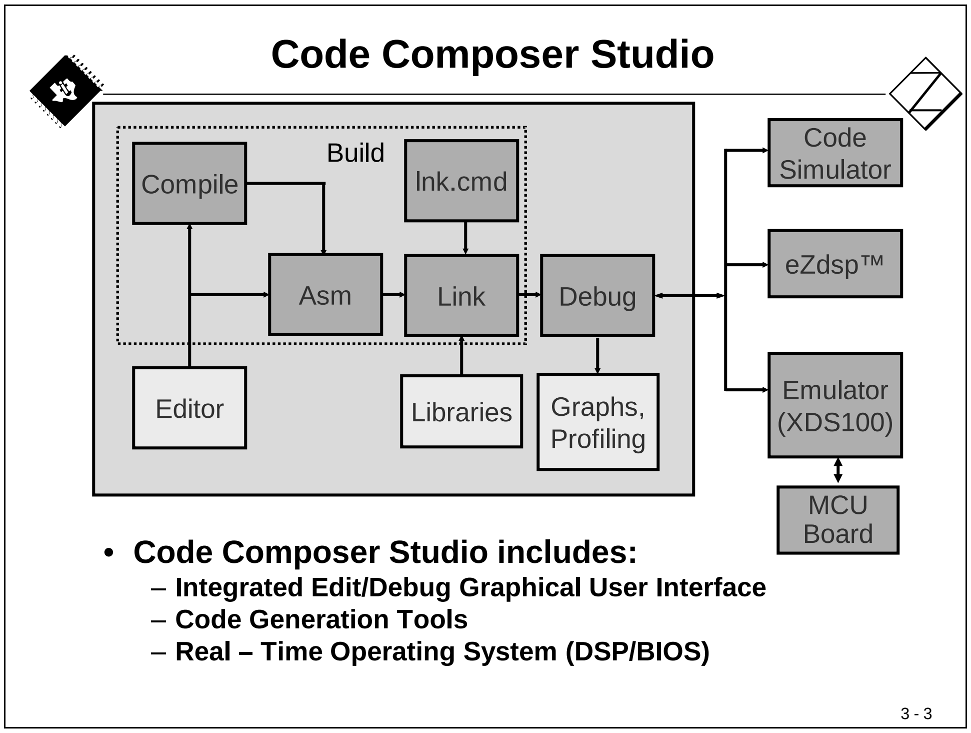
Productive Editor:

- **Structure Expansion**

Module Topics

The Software Flow

The following slide (Slide 3-3) 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 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!



Code Composer Studio

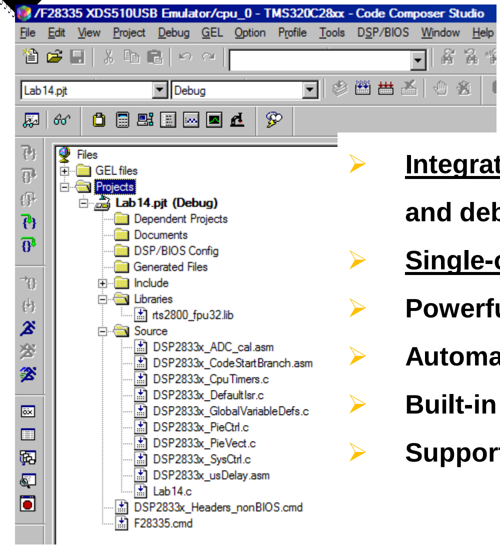
- **Code Composer Studio includes:**
  - **Integrated Edit/Debug Graphical User Interface**
  - **Code Generation Tools**
  - **Real – Time Operating System (DSP/BIOS)**

You can use Code Composer Studio with a Simulator (running on the Host - PC) or you can connect a microcontroller system and test the software on a real “target”. For this tutorial, we will rely on the Peripheral Explorer Board and the TMS320F28335 Control Card as our “target”. Here the word “target” means the physical processor we are using, in this case a TMS320F28335.

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

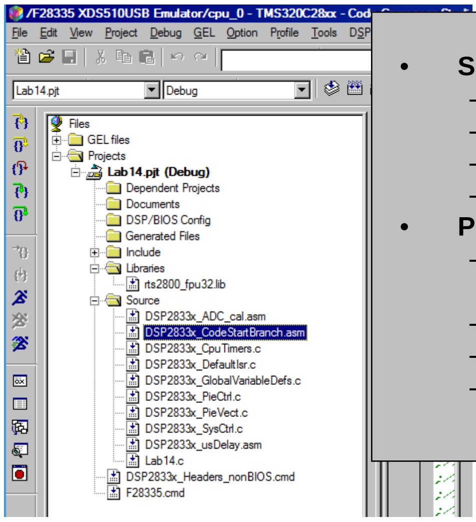
Code Composer Studio: IDE



- **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**

3 - 4

The Project - File

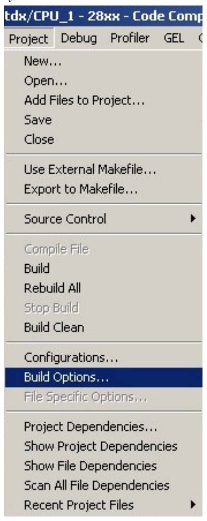


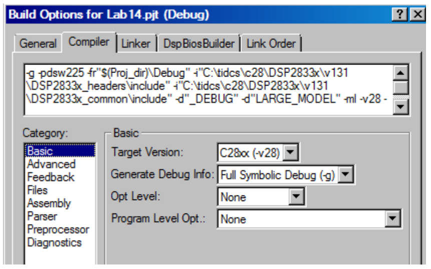
Project (.pjt) files contain:

- **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**

3 - 5

Build Options GUI - Compiler





```
-g -pdsw225 -fr"$(Proj_dir)\Debug" -i"C:\tidcs\c28\DSP2833x\v131
\DSP2833x_headers\include" -i"C:\tidcs\c28\DSP2833x\v131
\DSP2833x_common\include" -d"_DEBUG" -d"LARGE_MODEL" -ml -v28 -
```

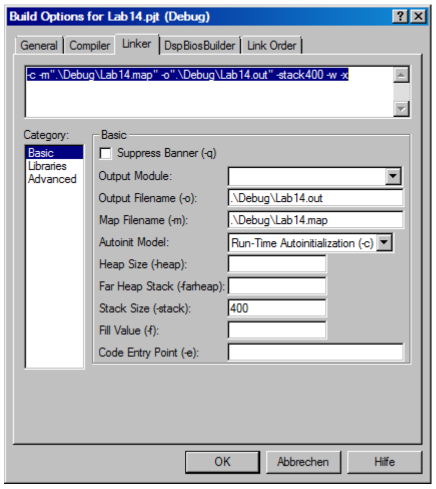
Basic

Target Version:	C28xx (-v28)
Generate Debug Info:	Full Symbolic Debug (-g)
Opt Level:	None
Program Level Opt.:	None

- 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

3 - 6

Build Options GUI - Linker



```
-c -m".\Debug\Lab14.map" -o".\Debug\Lab14.out" -stack400 -w -x
```

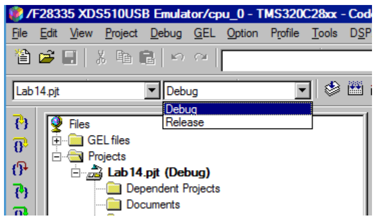
Basic

Suppress Banner (-q)	☐
Output Module:	
Output Filename (-o):	.\Debug\Lab14.out
Map Filename (-m):	.\Debug\Lab14.map
Autoinit Model:	Run-Time Autoinitialization (-c)
Heap Size (-heap):	
Far Heap Stack (-farheap):	
Stack Size (-stack):	400
Fill Value (-f):	
Code Entry Point (-e):	

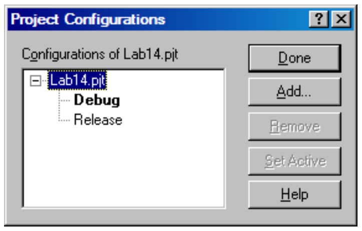
- GUI has 3 categories for linking
- Specifies various link options
- ".\Debug\" indicates on subfolder level below project (.pjt) location

3 - 7

Default Build Configurations


/F28335 XDS510USB Emulator/cpu_0 - TMS320C28xx - Cod
File Edit View Project Debug GEL Option Profile Tools DSP
Lab14.pjt
Debug
Debug
Release
Files
GEL files
Projects
Lab14.pjt (Debug)
Dependent Projects
Documents

- For new projects, CCS automatically creates two build configurations:
  - Debug (unoptimized)
  - Release (optimized)
- Use the drop-down menu to quickly select the build configuration

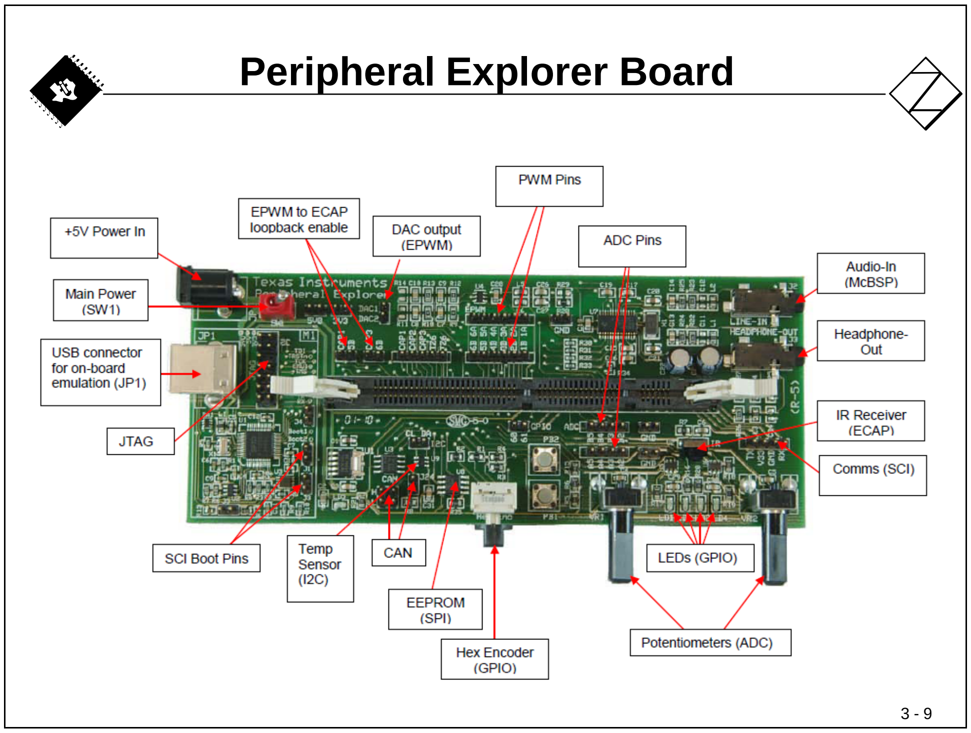


- Add/Remove your own custom build configurations using Project Configurations
- Edit a configuration:
  1. Set it active
  2. Modify build options
  3. Save project

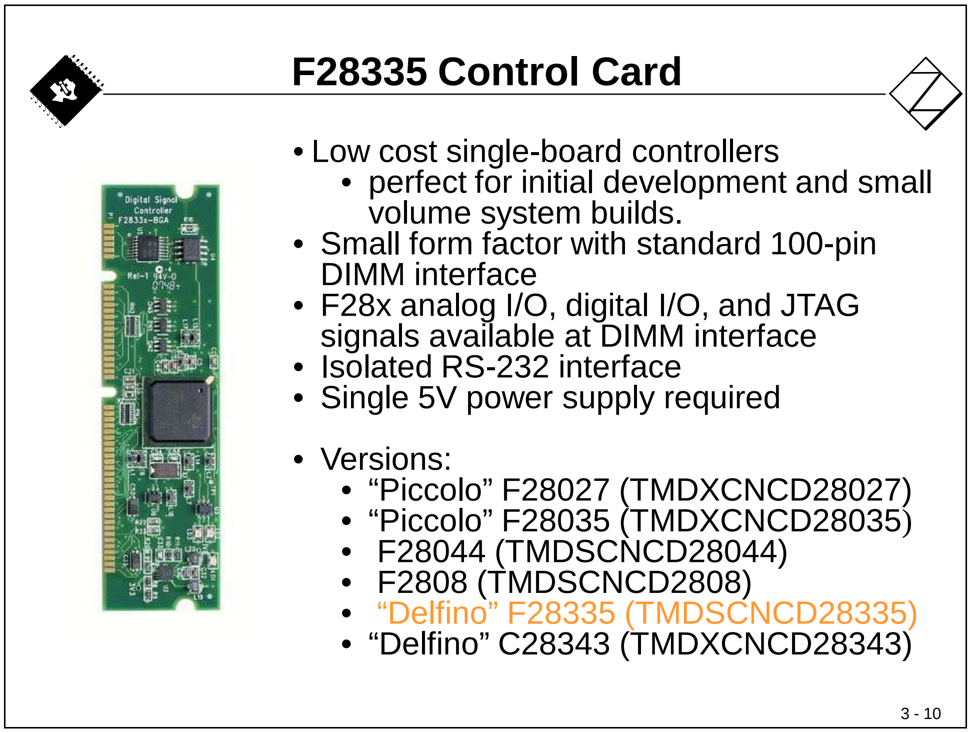
3 - 8

Lab Hardware Setup

The following slides show you the hardware target that is used during our lab exercises in the chapters that follow. The core is the TMS320F2335 32-bit Digital Signal Controller on board of a Texas Instruments “Peripheral Explorer Board”. All the major internal peripherals are available through connectors. The JTAG interface connects the board to the PC via a USB link.



Peripheral Explorer Board



F28335 Control Card

- Low cost single-board controllers
  - perfect for initial development and small volume system builds.
- Small form factor with standard 100-pin DIMM interface
- F28x analog I/O, digital I/O, and JTAG signals available at DIMM interface
- Isolated RS-232 interface
- Single 5V power supply required

- Versions:
  - “Piccolo” F28027 (TMDXCNCD28027)
  - “Piccolo” F28035 (TMDXCNCD28035)
  - F28044 (TMDSCNCD28044)
  - F2808 (TMDSCNCD2808)
  - “Delfino” F28335 (TMDSCNCD28335)
  - “Delfino” C28343 (TMDXCNCD28343)

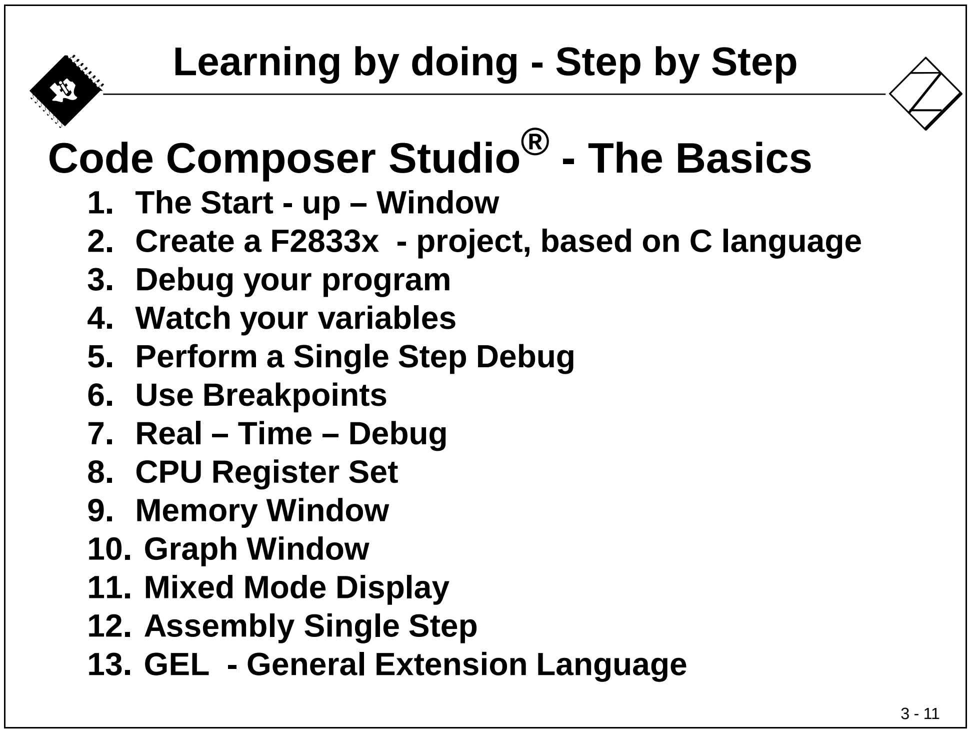
3 - 10

To be able to practice with all the peripheral units of the Digital Signal Controller and some ‘real’ process hardware, the Peripheral Explorer Board provides:

- 4 LEDs for digital output (GPIO9, GPIO11, GPIO34 and GPIO49)
- A 4 - bit hexadecimal input encoder (GPIO12…GPIO15) and 2 push buttons (GPIO 34 and GPIO17) for digital inputs
- 2 potentiometers (ADCINA0, ADCINA1) for analog inputs
- 1 stereo audio codec AIC23B for line -in and headphone -out (connected via McBSP and SPI)
- 1 SPI 256k - Atmel AT25C256 EEPROM (connected via McBSP)
- 1 CAN Transceiver - Texas Instruments SN 65HVD230 (high speed)
- 1 I^{2}C - Temperature Sensor Texas Instruments TMP100
- 1 SCI-A RS232 Transceiver - Texas Instruments MAX232D
- 1 Infrared Receiver TSOP32230 (connected to eCAP)

Code Composer Studio - Step by Step

Now let us start to look a little closer at the main parts of Code Composer Studio that we need to develop our first project. We will perform the following steps:



Learning by doing - Step by Step

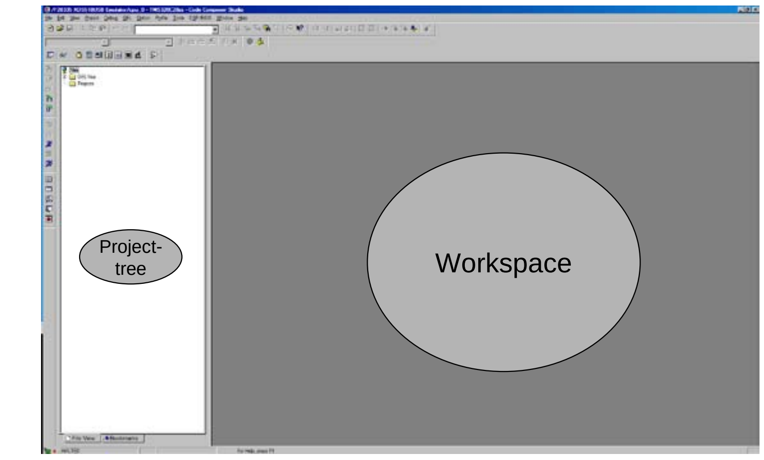
Code Composer Studio® - The Basics

1. The Start - up – Window
2. Create a F2833x - project, based on C language
3. Debug your program
4. Watch your variables
5. Perform a Single Step Debug
6. Use Breakpoints
7. Real – Time – Debug
8. CPU Register Set
9. Memory Window
10. Graph Window
11. Mixed Mode Display
12. Assembly Single Step
13. GEL - General Extension Language

3 - 11

Once you or your instructor has installed the tools and the correct driver (Setup CCS3.3), you can start Code Composer Studio by simply clicking on its desktop icon. If you get an error message, check the USB connection of the target board. If everything goes as expected, you should see a screen, similar to this:

1. The Start up - Window



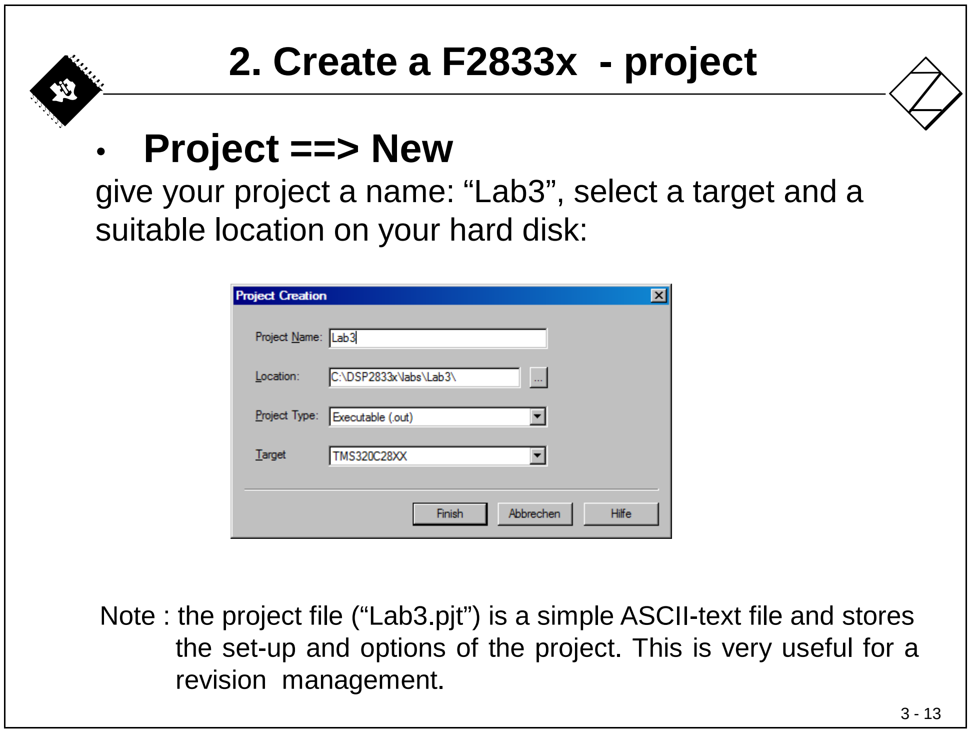
3 - 12

The step-by-step approach for Lab3 will show how to do the following:

- Open Code Composer Studio
- Create a F2833x - Project, based on C
- Compile, Link, Download and Debug this test program
- Watch Variables
- Continuous run and single - step mode
- Use of Breakpoints
- Use of “Real - Time - Debug” - mode
- View registers
- Mixed Mode (C and Assembler Language)
- General Extension Language (GEL)

Before we start to go into the procedure for Lab3, let us discuss the individual steps using some additional slides:

Create a project



2. Create a F2833x - project

- **Project ==> New**

give your project a name: “Lab3”, select a target and a suitable location on your hard disk:

Project Creation

Project Name: Lab3

Location: C:\DSP2833x\labs\Lab3\

Project Type: Executable (.out)

Target TMS320C28XX

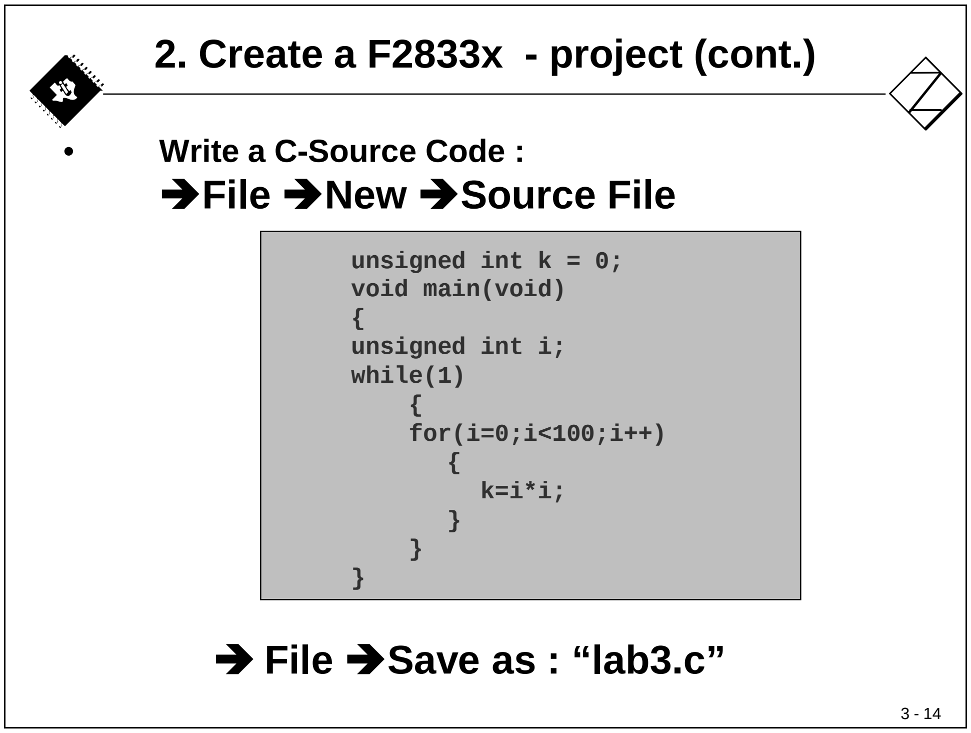
Note : the project file (“Lab3.pjt”) is a simple ASCII-text file and stores the set-up and options of the project. This is very useful for a revision management.

3 - 13

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.

Write C - code

Next, write the source code for your first application. The program from the slide below is one of the simplest tasks for a processor.



2. Create a F2833x - project (cont.)

- **Write a C-Source Code :**

➔File ➔New ➔Source File

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

➔ File ➔Save as : "lab3.c"

3 - 14

The code example consists of an endless while(1) - loop, which contains a single for - loop - instruction.

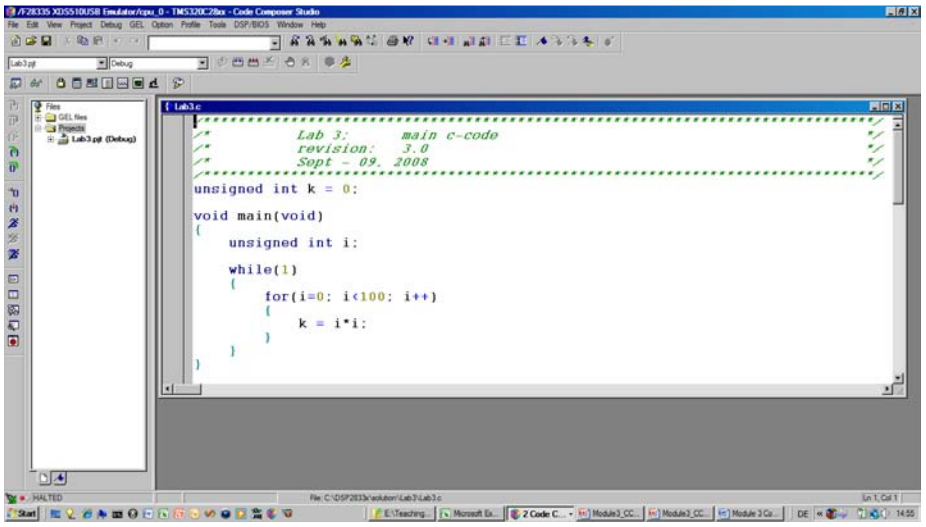
In that for-loop we:

- increment variable i from 0 to 99,
- calculate the current product of i * i and
- store it temporarily in variable k.

It seems to be an affront to bother a sophisticated Digital Signal Controller with such a simple task! However, we want to gain hands-on experience of this DSC 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 the following slide:

2. Create a F2833x - project (cont.)



/F28335 XDS510USB Emulator/cpu_0 - TMS320C28xx - Code Composer Studio
File Edit View Project Debug GEL Option Profile Tools DSP/BIOS Window Help
Lab3.pjt
Debug
Files
GEL files
Projects
Lab3.pjt (Debug)
Lab3.c
HALTED
Ln 1, Col 1

```
/*************************************************************************/
/*            Lab 3:       main c-code                                   */
/*            revision:    3.0                                           */
/*            Sept - 09. 2008                                            */
/*************************************************************************/
unsigned int k = 0;

void main(void)
{
    unsigned int i;

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

3 - 15

The C - source code file is now stored on the hard disk of the PC 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 to 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 F2833x - project (cont.)

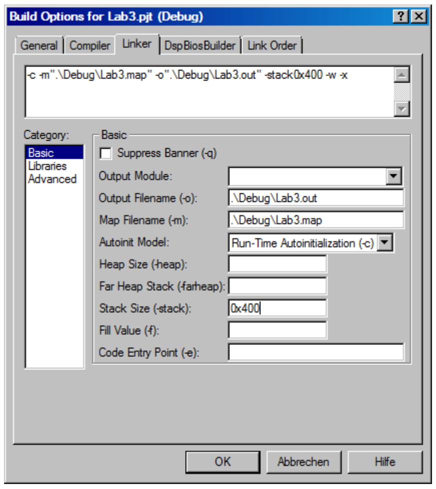
- **Add your file to the project :**
  ➔ Project ➔ Add files to project:
  Add: "lab3.c"
- **Compile your source code :**
  ➔ Project ➔ Compile File
  The active window will be compiled. In the event of syntax errors: modify your source code as needed.
- **Add the C-runtime-library to your project :**
  ➔ Project ➔ Add files to project:

  C:\CCStudio_v3.3\C2000\cgtools\lib\rts2800_ml.lib
- **Add the stack - size of 0x400**
  ➔ Project ➔ Build Options ➔ Linker ➔ Stack Size : 0x400

3 - 16

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

2. Create a F2833x - project (cont.)



```
-c -m".\Debug\Lab3.map" -o".\Debug\Lab3.out" -stack0x400 -w -x
```

Basic

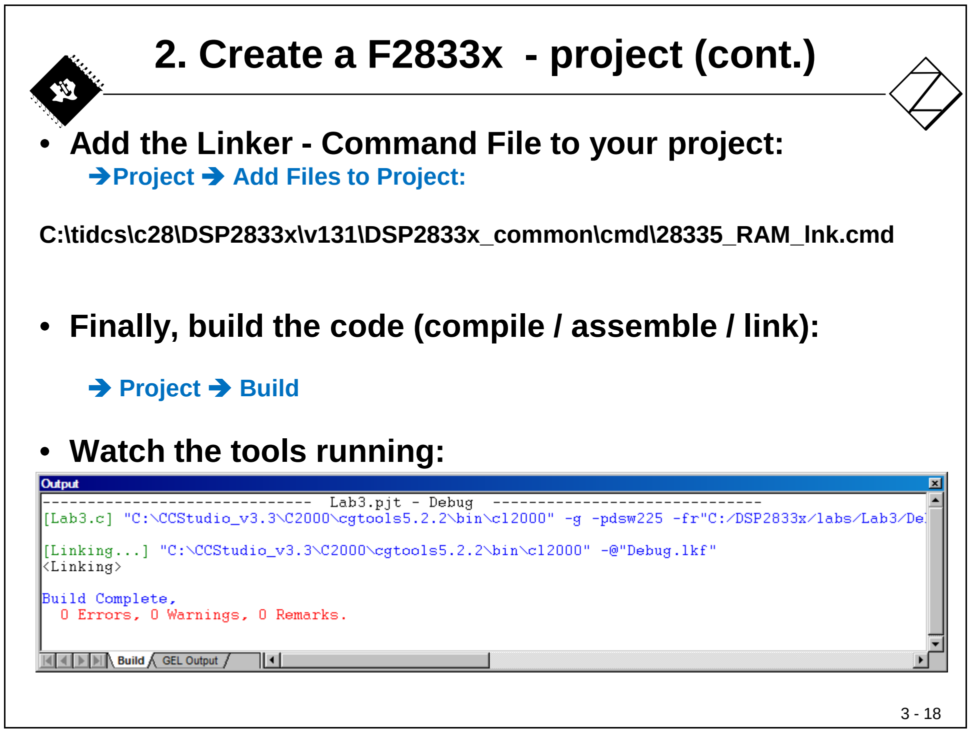
☐ Suppress Banner (-q)	
Output Module:	
Output Filename (-o):	.\Debug\Lab3.out
Map Filename (-m):	.\Debug\Lab3.map
Autoinit Model:	Run-Time Autoinitialization (-c)
Heap Size (-heap):	
Far Heap Stack (-farheap):	
Stack Size (-stack):	0x400
Fill Value (-f):	
Code Entry Point (-e):	

Close the build-window by 'OK'

3 - 17

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 “28335_RAM_lnk.cmd”. This file has been provided by Texas Instruments and is part of the “Header - File” support package.



2. Create a F2833x - project (cont.)

- **Add the Linker - Command File to your project:**
  - **➔Project ➔ Add Files to Project:**

C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\cmd\28335_RAM_lnk.cmd

- **Finally, build the code (compile / assemble / link):**
  - **➔ Project ➔ Build**

- **Watch the tools running:**

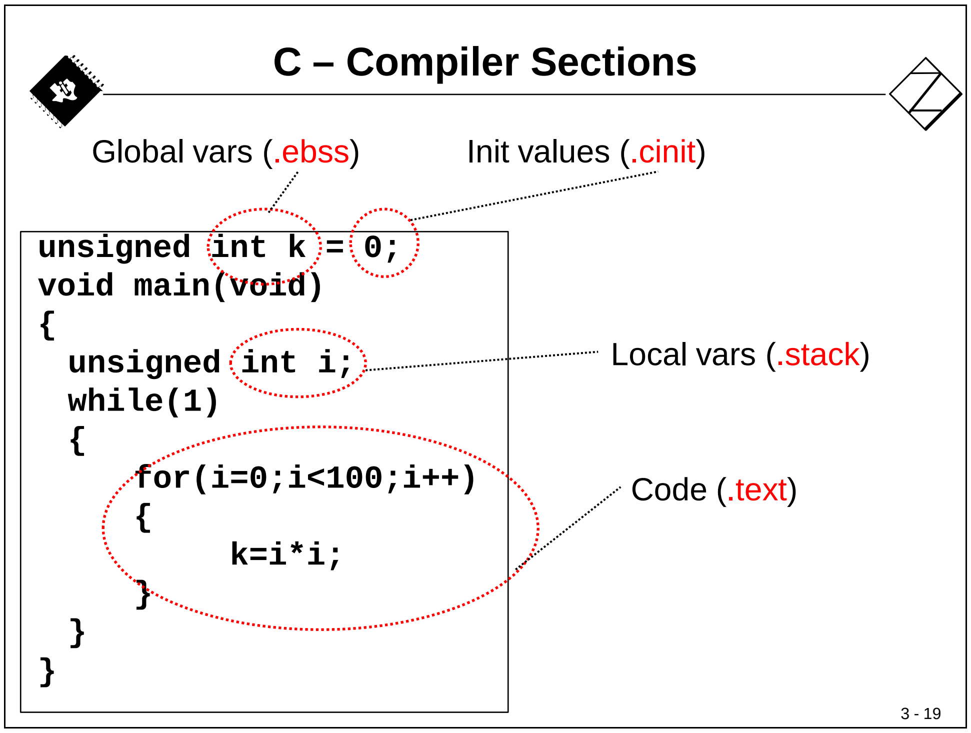
```
------------------------------  Lab3.pjt - Debug  ------------------------------
[Lab3.c] "C:\CCStudio_v3.3\C2000\cgtools5.2.2\bin\cl2000" -g -pdsw225 -fr"C:/DSP2833x/labs/Lab3/De

[Linking...] "C:\CCStudio_v3.3\C2000\cgtools5.2.2\bin\cl2000" -@"Debug.lkf"
<Linking>

Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

3 - 18

C - Compiler Sections

When we compile our tiny code from Lab3, the C - compiler will generate 4 so-called “sections”. These sections cover different parts of the object module, which must be “linked” to physical memory. Our four sections are:

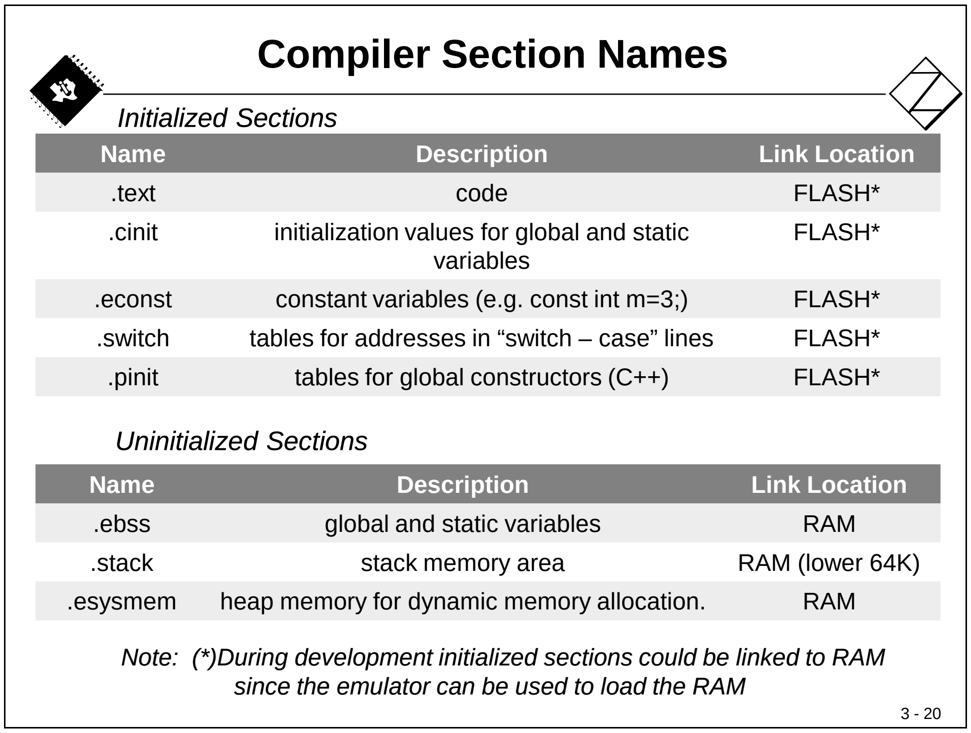
• .text	This section collects all assembly code instructions
• .ebss	The section covers all global and static variables
• .cinit	This section is used for initial values
• .stack	The stack memory for local variables, return addresses, parameters



C – Compiler Sections

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

The linker will connect these sections to physical memory. For this task we pass information to the linker with the help of “Linker - command - files” (extension *.cmd). But before we look at the details of this procedure, let us finish the C compiler sections. As you can probably guess, when we use a slightly more complex program than Lab3, the C compiler will generate more sections. The following slide will summarize all possible C sections:



Compiler Section Names

Initialized Sections

Name	Description	Link Location
.text	code	FLASH*
.cinit	initialization values for global and static variables	FLASH*
.econst	constant variables (e.g. const int m=3;)	FLASH*
.switch	tables for addresses in “switch – case” lines	FLASH*
.pinit	tables for global constructors (C++)	FLASH*

Uninitialized Sections

Name	Description	Link Location
.ebss	global and static variables	RAM
.stack	stack memory area	RAM (lower 64K)
.esysmem	heap memory for dynamic memory allocation.	RAM

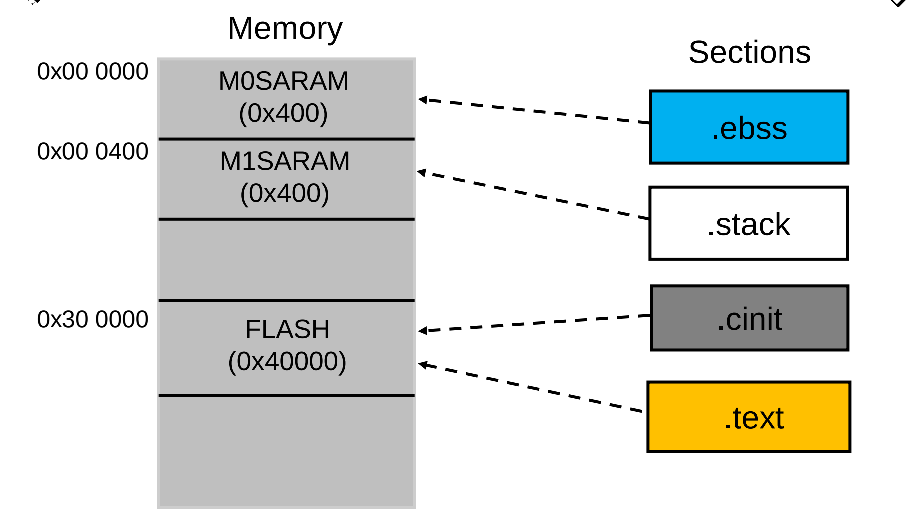
Note: (*)During development initialized sections could be linked to RAM since the emulator can be used to load the RAM

3 - 20

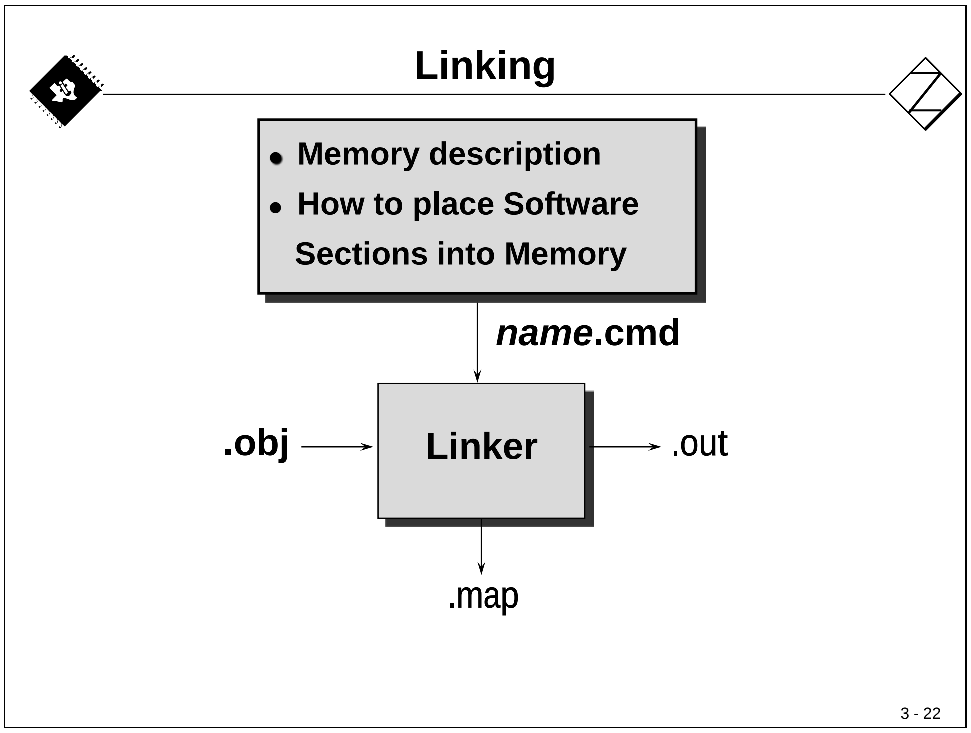
Linking Sections to Memory

The following slide is an example, how we could link the four sections from Lab3 into parts of physical memory. For a standalone embedded system, all constants, initialization values and code must be stored in non-volatile memory, such as FLASH. Un-initialized data (variables) are linked to RAM (Note: For our first labs, we will use only RAM sections).

Placing Sections in Memory

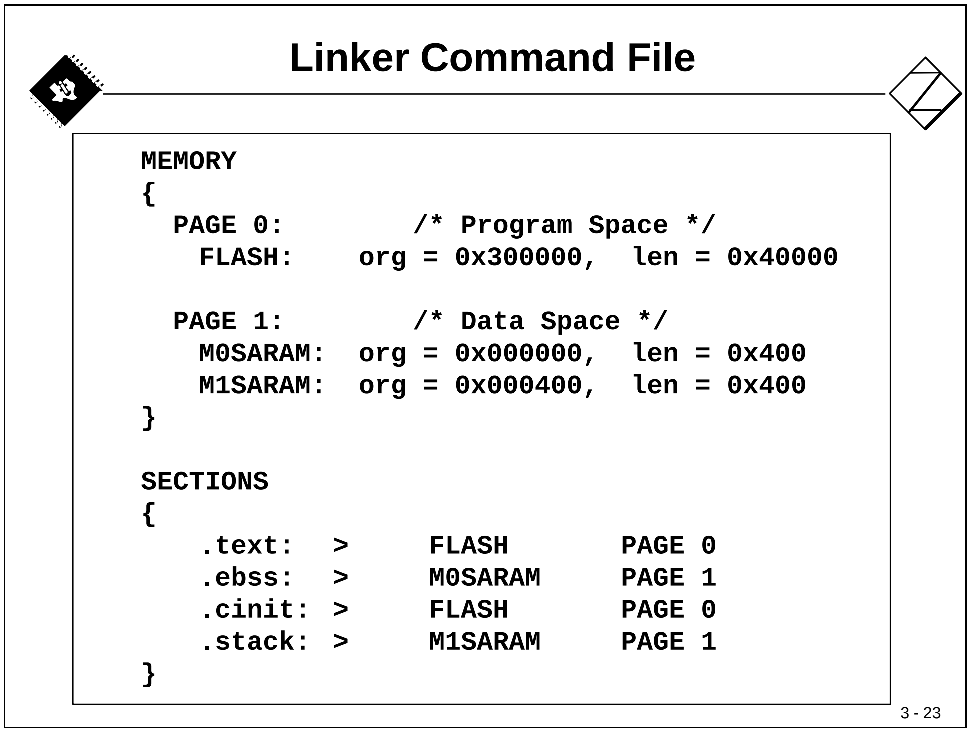


3 - 21



Linking

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 DSC, but also information used to debug, to flash the controller and for more JTAG based tasks. 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

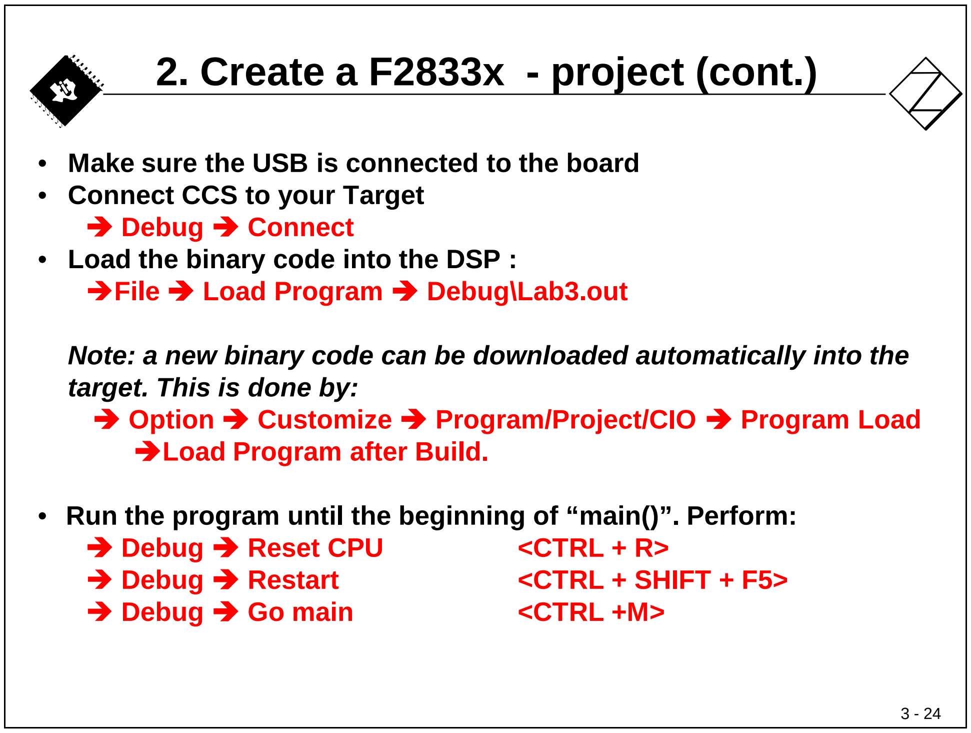
```
MEMORY
{
  PAGE 0:         /* Program Space */
    FLASH:    org = 0x300000,  len = 0x40000

  PAGE 1:         /* Data Space */
    M0SARAM:  org = 0x000000,  len = 0x400
    M1SARAM:  org = 0x000400,  len = 0x400
}

SECTIONS
{
    .text:   >     FLASH       PAGE 0
    .ebss:   >     M0SARAM     PAGE 1
    .cinit:  >     FLASH       PAGE 0
    .stack:  >     M1SARAM     PAGE 1
}
```

Note: Slide 3-23 is the solution for the example from Slide 3-21 and not for lab3!

Download code into the controller

Now it is time to download the code into the F2833x. We have two options: manual or automatic download after a successful build. Note: If our code would contain FLASH - parts, we would need to use a FLASH - programming module. This option will be discussed later in Chapter 14.

For projects, which are intended to run both with code and data from RAM, we can use Code Composer Studio to directly download all sections into their physical memory locations.



2. Create a F2833x - project (cont.)

- **Make sure the USB is connected to the board**
- **Connect CCS to your Target**
  ➔ Debug ➔ Connect
- **Load the binary code into the DSP :**
  ➔File ➔ Load Program ➔ Debug\Lab3.out

  Note: a new binary code can be downloaded automatically into the target. This is done by:
  ➔ Option ➔ Customize ➔ Program/Project/CIO ➔ Program Load
  ➔Load Program after Build.

- **Run the program until the beginning of "main()". Perform:**

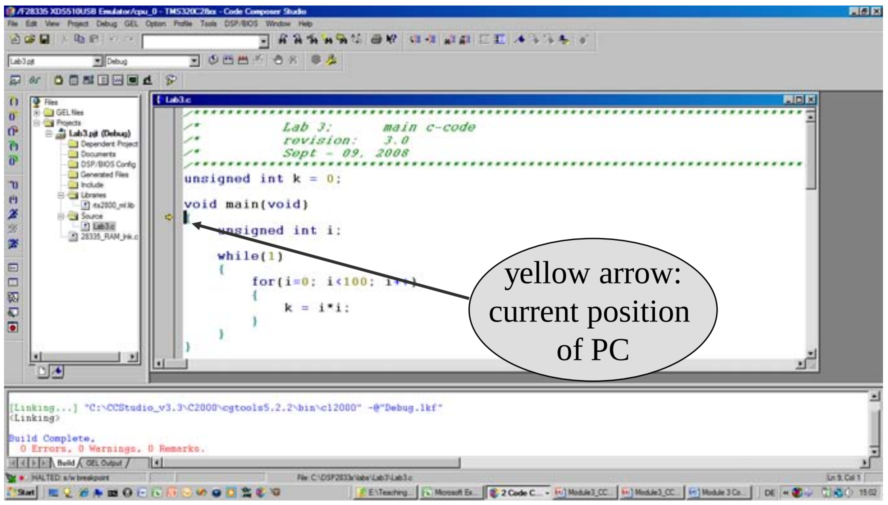
➔ Debug ➔ Reset CPU	**<CTRL + R>**
➔ Debug ➔ Restart	**<CTRL + SHIFT + F5>**
➔ Debug ➔ Go main	**<CTRL +M>**

3 - 24

Test the Code

After ➔ *Debug* ➔ *Go main*, a yellow arrow shows the current position of the Program Counter (PC). The purpose of register "PC" is to always point the next machine code instruction to be executed.

2. Create a F2833x - project (cont.)



```
/**********************************************************************
/*            Lab 3:       main c-code
/*            revision:    3.0
/*            Sept - 09, 2008
/**********************************************************************
unsigned int k = 0;

void main(void)
{
    unsigned int i;

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

```
[Linking...] "C:\CCStudio_v3.3\C2000\cgtools5.2.2\bin\cl2000" -@"Debug.lkf"
<Linking>

Build Complete,
  0 Errors, 0 Warnings, 0 Remarks.
```

/F28335 XDS510USB Emulator/cpu_0 - TMS320C28xx - Code Composer Studio
Lab3.c
yellow arrow: current position of PC
HALTED: s/w breakpoint

3 - 25

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 controller.

To exercise the different start and stop options, go through the steps, shown on the next slide.

3. Run and Halt

- **Perform a real time run:**
  - ➔ Debug ➔ Run <F5>

  Note 1: the bottom left corner will be marked as : "Running". You'll see no activity at the Peripheral Explorer Board, because the first example program does not use any of them!

  Note 2: the yellow arrow is no longer visible, because the F2833x runs at full speed.

- **Stop the real time run:**
  - ➔ Debug ➔ Halt <SHIFT + F5>

- **Repeat the steps to reach "main()" again:**
  - ➔ Debug ➔ Reset CPU <CTRL + R>
  - ➔ Debug ➔ Restart <CTRL + SHIFT + F5>
  - ➔ Debug ➔ Go main <CTRL +M>

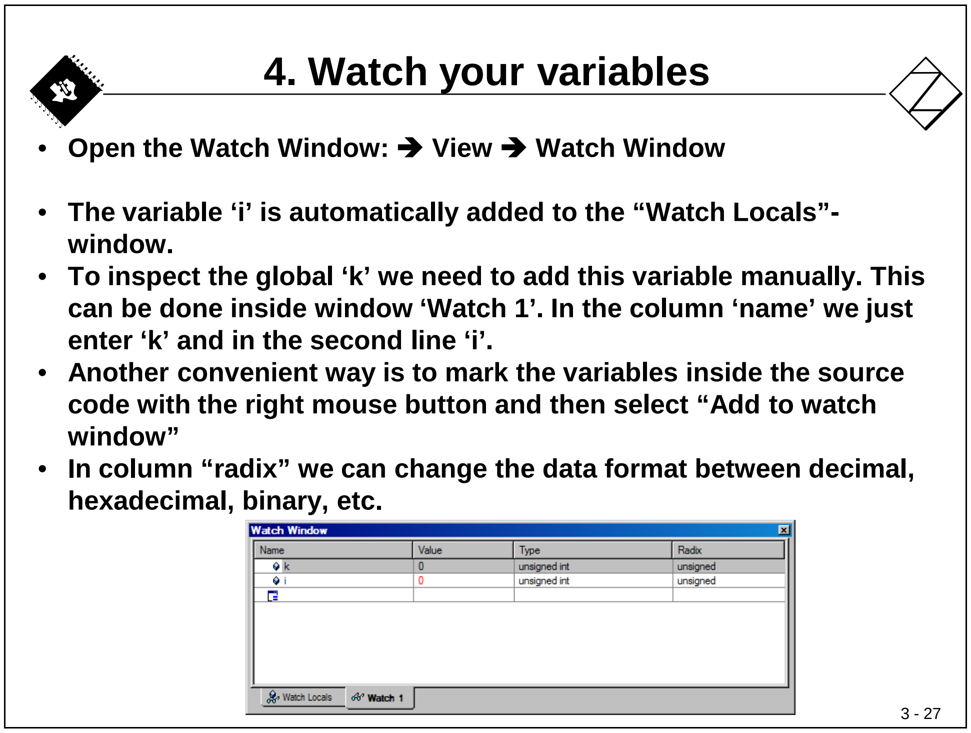
3 - 26

The Watch Window

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. It is good engineering practice to carefully test parts and modules of a software project. For an embedded system we need to ‘look’ into internal parts of the controller, such as variables and function stacks and monitor their changes.

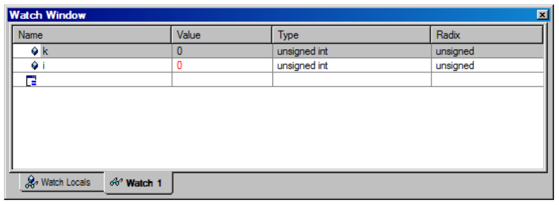
Here the Watch Window is of great use. Instead of hitting the ‘run’ - key F5 and hoping that the software behaves as expected, it is much better to test it systematically. That means:

- Predict, what will happen in the next instruction
- Single step the critical code instruction
- Monitor the variables of that code snippet and compare the results with your expectations.
- Proceed with the next line under test.



4. Watch your variables

- **Open the Watch Window: ➔ View ➔ Watch Window**
- **The variable ‘i’ is automatically added to the “Watch Locals”-window.**
- **To inspect 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’.**
- **Another convenient way is to mark the variables inside the source code with the right mouse button and then select “Add to watch window”**
- **In column “radix” we can change the data format between decimal, hexadecimal, binary, etc.**



Name	Value	Type	Radix
k	0	unsigned int	unsigned
i	0	unsigned int	unsigned

3 - 27

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

5. Perform a Single Step Debug

- **Perform a single step through the program:**

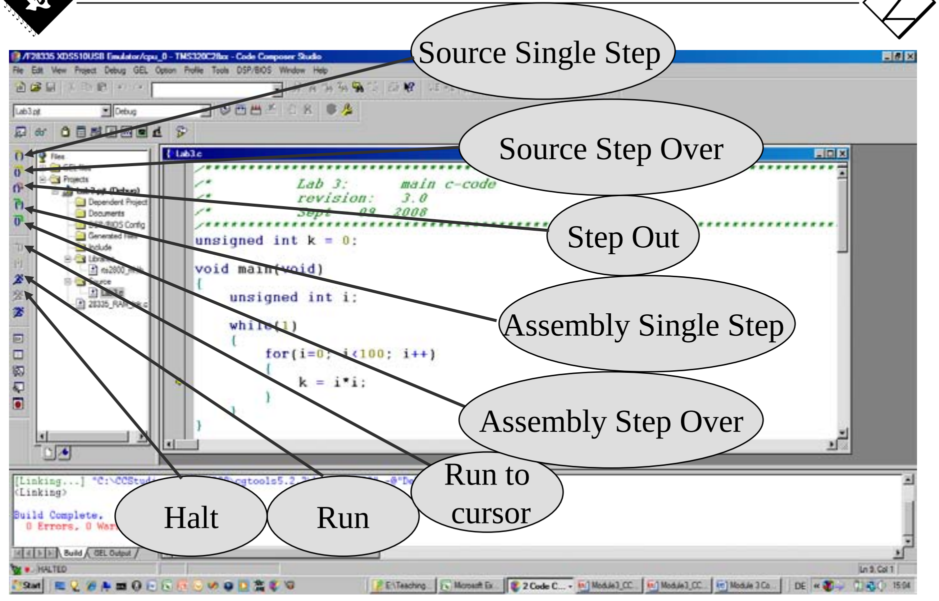
  ➔ Debug ➔ Step Into (or use function key "F11")

- **Watch the current PC (yellow arrow) and the values of variables 'i' and 'k' in Watch Window while you single step through the code !**

- **More debug - commands are shown at the following slide:**

3 - 28

5. Perform a Single Step Debug



```
/*      Lab 3:      main c-code
/*      revision:   3.0
/*      Sept   09   2008
unsigned int k = 0;
void main(void)
{
   unsigned int i;
   while(1)
   {
      for(i=0; i<100; i++)
      {
         k = i*i;
      }
   }
}
```

```
[Linking...] "C:\CCStudi
<Linking>
Build Complete,
  0 Errors, 0 War
```

3 - 29

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

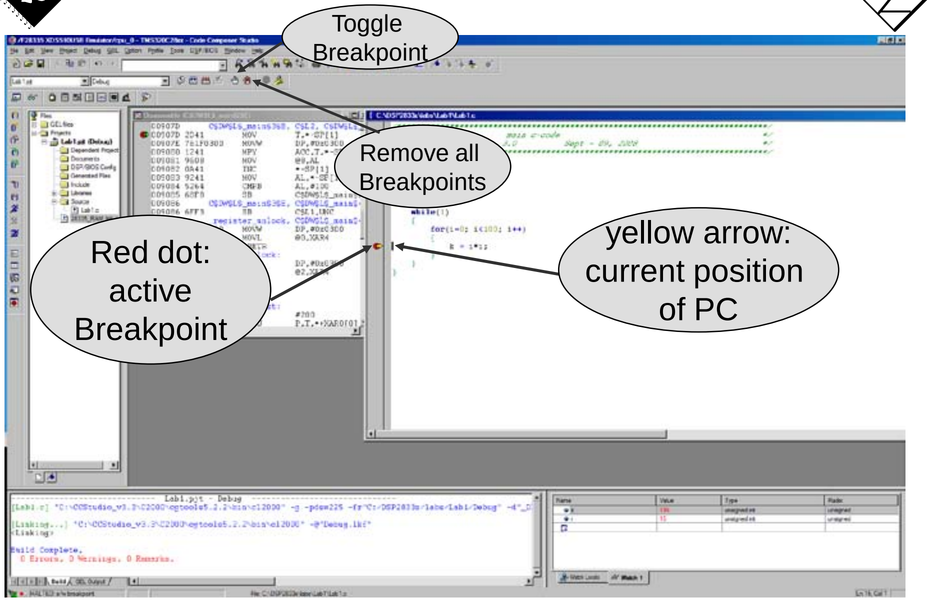
6. Adding a Breakpoint

- **Set a Breakpoint :**
  - **Place the Cursor in Lab3.c at line: k = i * i;**
  - **Click right mouse and select ‘Toggle Software Breakpoint’**
  - **the line is marked with a red dot to indicate an active breakpoint.**
  - **Another option to toggle a breakpoint is the “hand” – icon on the top menu of CCS.**
  - **Or, use a left mouse double click at the grey left hand side of window “Lab3.c” to toggle the breakpoint.**
- **Reset the Program**
  - **➔ Debug ➔ Reset CPU** **<CTRL+R>**
  - **➔ Debug ➔ Restart** **<CTRL+SHIFT+F5>**
- **Perform a real time run**
  - **➔ Debug ➔ Run** **<F5>**

  The F2833x stops after hitting an active breakpoint
- **repeat ‘Run’ and watch your variables**
- **remove the breakpoint (Toggle again) when you’re done.**

3 - 30

6. Adding a Breakpoint (cont.)


Toggle Breakpoint
Remove all Breakpoints
Red dot: active Breakpoint
yellow arrow: current position of PC

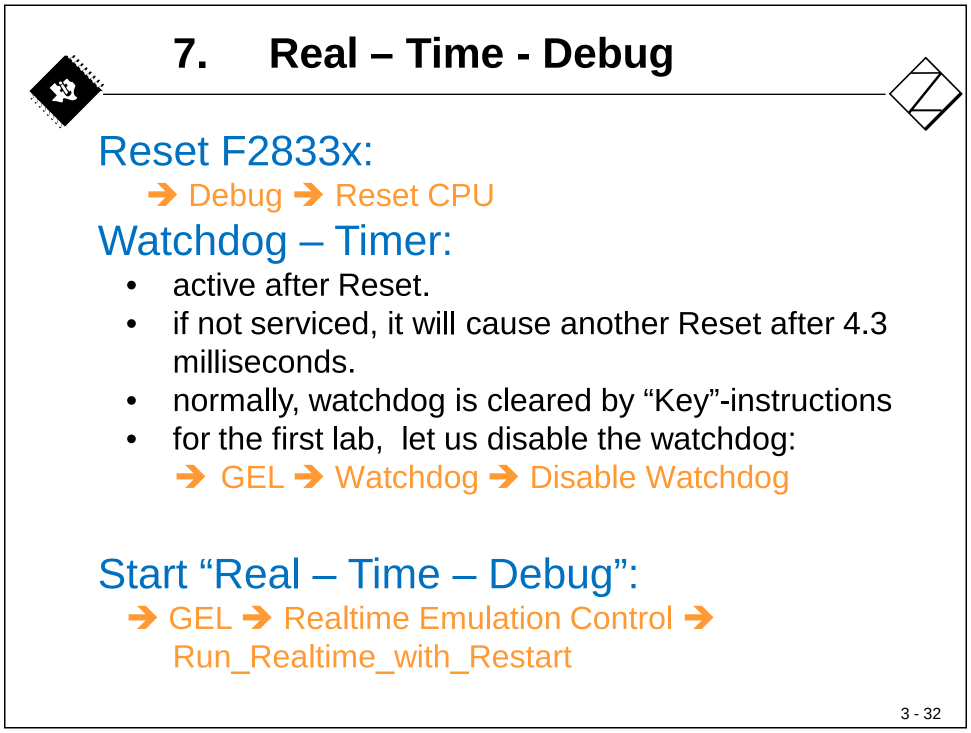
3 - 31

Real - Time Debug Mode

A critical part of a test session is any interference between the control code and the test functions. Imagine, what would happen with PWM output lines for power inverters, if we would just place a breakpoint in a certain point in our code. If the processor hits the breakpoint, the execution would stop immediately, including all dynamic services of the PWM lines. The result would be: fatal, because a permanent open switch will destroy the power circuits.

The best solution would be to have an operating mode, in which the control code is not disturbed at all by any data exchange between Code Composer Studio and the running control code. This test mode is called **"Real - Time - Debug"**. It is based on a large set of internal registers in the JTAG - support module of the F2833x. At this stage we will not discuss the internal functionality; we will just use its basic features.

It is important to delete or disable all breakpoints in a CCS - session, before you switch ON the Real - Time - Debugger. So please make sure, that no breakpoints are left from previous tests!



7. Real – Time - Debug

Reset F2833x:

➔ Debug ➔ Reset CPU

Watchdog – Timer:

- active after Reset.
- if not serviced, it will cause another Reset after 4.3 milliseconds.
- normally, watchdog is cleared by "Key"-instructions
- for the first lab, let us disable the watchdog:
  ➔ GEL ➔ Watchdog ➔ Disable Watchdog

Start "Real – Time – Debug":

➔ GEL ➔ Realtime Emulation Control ➔ Run_Realtime_with_Restart

3 - 32

To switch into Real - Time - Debug, it is good practice to first reset the device. This step ensures that all previous core and peripheral initialization is cancelled.

We have not discussed so far the internal watchdog unit. This unit is nothing more than a free running counter. If it is not serviced, it will cause a hardware reset. The purpose of this unit is to monitor the correct flow of control code. There are two dedicated clear instructions, which are normally executed from time to time, if the code is still running as expected. If not, because the code hangs somewhere, the watchdog will bring the device back into a safe passive state. It operates similar to a "dead man's handle" in a railway locomotive.

We will discuss and use the watchdog in detail in chapter 5. However, the watchdog is active after power on, so we cannot neglect it! For now, we can use a CCS GEL - command to disable the watchdog. We would never do that in a real project!

To use “Real - Time - Debug” perform:

➔ Debug ➔ Reset CPU

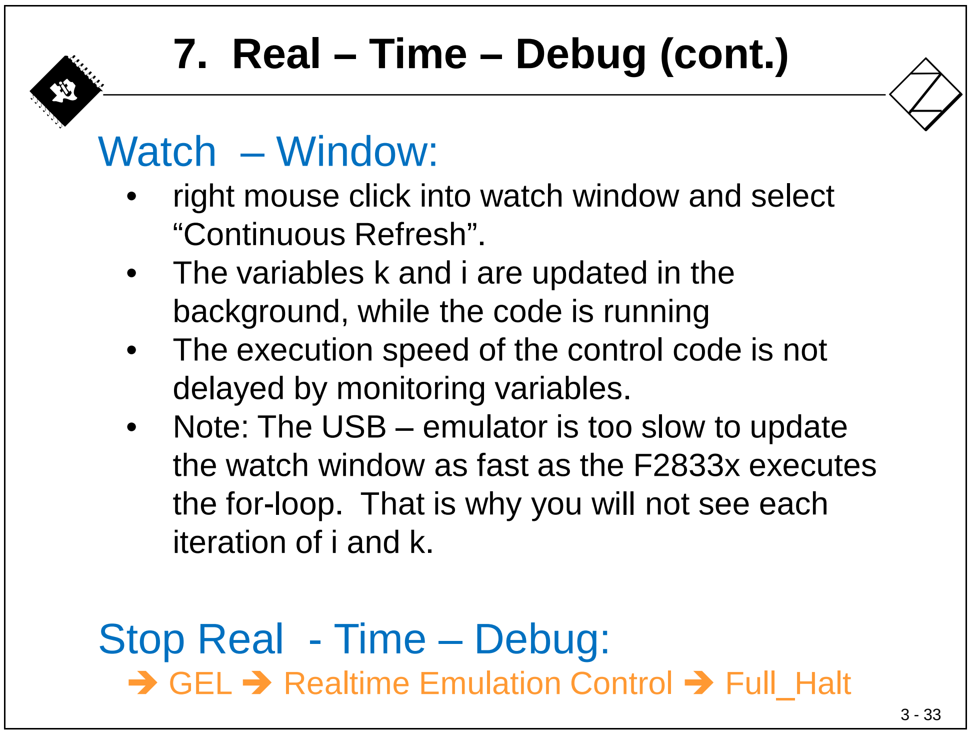
➔ GEL ➔ Watchdog ➔ Disable Watchdog

➔ GEL ➔ Realtime Emulation Control ➔ Run_Realtime_with_Restart

Now the code is running in real-time. The new feature is that we can interact with the device, while the code is running. To practice this:

- Right click into the Watch Window and enable the option “Continuous Refresh”

The content of the watch window is updated frequently. The JTAG - controller uses cycles, in which the core of the device does not access the variables to “steal” the information needed to update the window. This real - time mode is called “Polite”, as you can see in the lower left corner of CCS.



7. Real – Time – Debug (cont.)

Watch – Window:

- right mouse click into watch window and select “Continuous Refresh”.
- The variables k and i are updated in the background, while the code is running
- The execution speed of the control code is not delayed by monitoring variables.
- Note: The USB – emulator is too slow to update the watch window as fast as the F2833x executes the for-loop. That is why you will not see each iteration of i and k.

Stop Real - Time – Debug:

➔ GEL ➔ Realtime Emulation Control ➔ Full_Halt

3 - 33

When you are done, you should stop the real - time test by:

➔ GEL ➔ Realtime Emulation Control ➔ Full_Halt

Note: the test mode **“Run_Realtime_with_Reset”** will first perform a reset followed by a direct start of the device from its reset address. The device will follow its hardware boot sequence (see Chapter 15) to begin the code execution. Since the Peripheral Explorer Board sets the coding pins to “Branch to FLASH” by default, it would start code stored in FLASH.

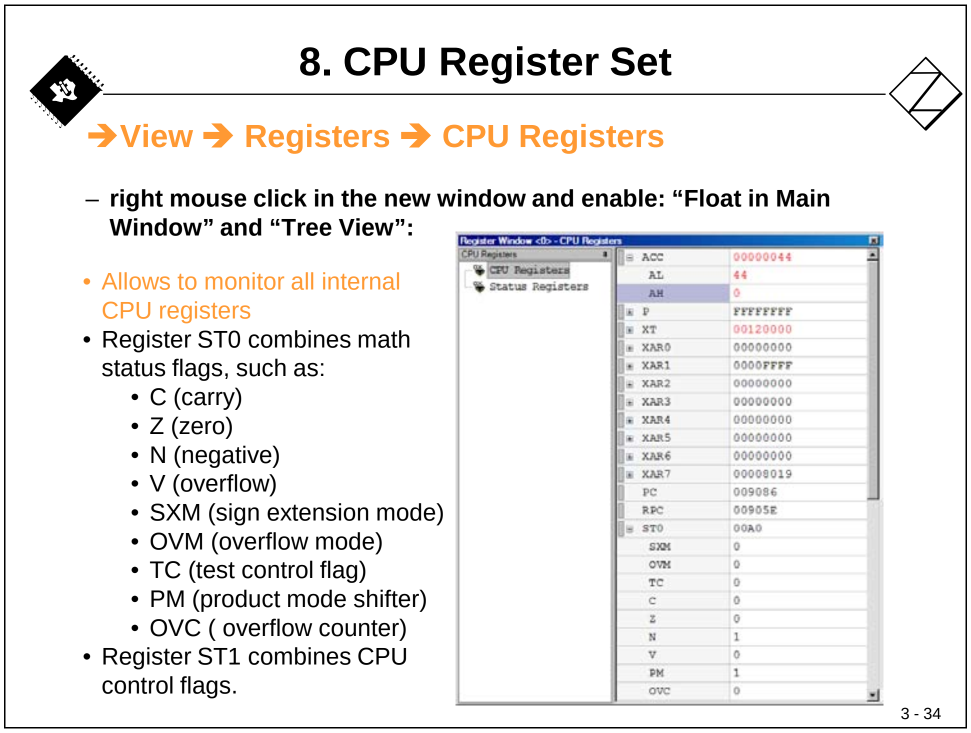
The problem is, that so far we have not stored anything in FLASH (we will do this in Chapter 14). By using **"Run_Realtime_with_Restart",** we force CCS to place the program counter at the start address of our project in RAM (a function called "c_int00") and to start from this position.

CPU Register Set

When you are more familiar with the F2833x and with the tools, you might want to verify the efficiency of the C compiler or to optimize your code at the Assembly Language level. As a beginner you are probably not in the mood to do this advanced step now, but a brief look would not be amiss.

Open a register window:

➔ View ➔ Registers ➔ CPU Registers



8. CPU Register Set

➔View ➔ Registers ➔ CPU Registers

– **right mouse click in the new window and enable: "Float in Main Window" and "Tree View":**

- Allows to monitor all internal CPU registers
- Register ST0 combines math status flags, such as:
  - C (carry)
  - Z (zero)
  - N (negative)
  - V (overflow)
  - SXM (sign extension mode)
  - OVM (overflow mode)
  - TC (test control flag)
  - PM (product mode shifter)
  - OVC (overflow counter)
- Register ST1 combines CPU control flags.

Register Window <0> - CPU Registers

CPU Registers
- CPU Registers
- Status Registers

ACC	00000044
AL	44
AH	0
P	FFFFFFFF
XT	00120000
XAR0	00000000
XAR1	0000FFFF
XAR2	00000000
XAR3	00000000
XAR4	00000000
XAR5	00000000
XAR6	00000000
XAR7	00008019
PC	009086
RPC	00905E
ST0	00A0
SXM	0
OVM	0
TC	0
C	0
Z	0
N	1
V	0
PM	1
OVC	0

3 - 34

The option "Tree View" allows you to inspect details of registers more in detail. At this early stage of the tutorial it is not important to understand the meaning of all the bit fields and registers, shown in this window. But you should get the feeling, that with the help of such windows, you can obtain control information about internal activities of the device.

There are two core registers, ST0 and ST1, which combine all core control switches and flags of the CPU, such as carry, zero, negative, overflow, sign extension mode, interrupt enable and so on. An inspection of these flags and bits allows you to immediately monitor the status of the CPU in a certain spot of code.

The 32-bit registers ACC ("accumulator"), P ("Product") and XT ("eXtended Temp") are the core math registers of the fixed - point arithmetic unit.

The 32-bit registers XAR0 to XAR7 are general purpose registers, often used as pointer registers or auxiliary registers for temporary results.

The register PC ("Program Counter") points always the address of the next machine code instruction in code memory. The register RPC ("return program counter") stores the return address of a function, which has called a sub-routine.

Watch Memory Contents

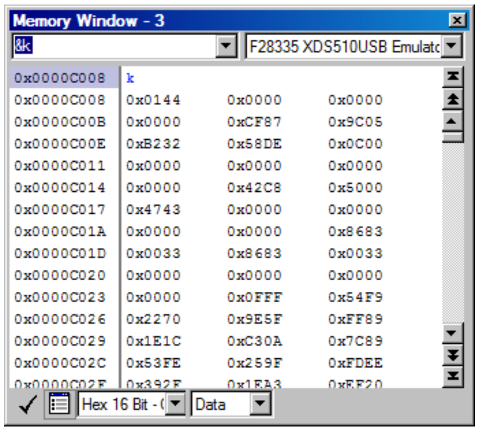
Let us open another control window, the "Memory Window". This window allows us to inspect any physical memory locations of the device, including RAM, FLASH, OTP and Boot - ROM. Since the core of this device is a Digital Signal Processor, we have always to remember that there are two memory spaces, code and data. To inspect variables, we have to select "data space". For machine code instructions inspection we have to look into "code space". The selection is made in the center box at the bottom of this window.

9. Memory Window

➔View ➔ Memory

– **make sure to inspect data space (bottom centre box)**
– **enter address &k in the top left corner box:**

- the current value in memory location of variable k is displayed
- change the display type from "Hex – 16 bit" to "16 Bit Unsigned Integer"
- change the display type into "16 Bit Binary"



```
0x0000C008  k
0x0000C008  0x0144  0x0000  0x0000
0x0000C00B  0x0000  0xCF87  0x9C05
0x0000C00E  0xB232  0x58DE  0x0C00
0x0000C011  0x0000  0x0000  0x0000
0x0000C014  0x0000  0x42C8  0x5000
0x0000C017  0x4743  0x0000  0x0000
0x0000C01A  0x0000  0x0000  0x8683
0x0000C01D  0x0033  0x8683  0x0033
0x0000C020  0x0000  0x0000  0x0000
0x0000C023  0x0000  0x0FFF  0x54F9
0x0000C026  0x2270  0x9E5F  0xFF89
0x0000C029  0x1E1C  0xC30A  0x7C89
0x0000C02C  0x53FE  0x259F  0xFDEE
0x0000C02F  0x392F  0x1EA3  0xEF20
```

3 - 35

The bottom left box allows us to specify the display mode of the 16-bit memory locations in different form, such as:

- Hexadecimal
- Integer, signed and unsigned
- Binary
- Float
- Character

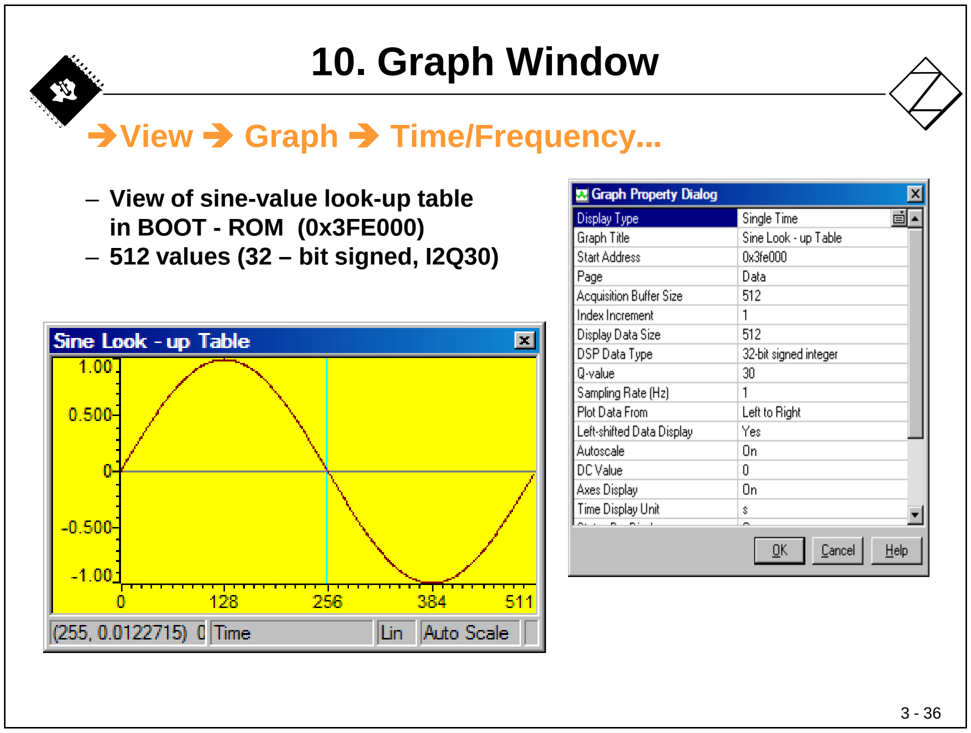
The number of memory windows is not limited, you can open as many as you like!

Graphical View

A unique feature of Code Composer Studio (CCS) is the ability to display any region of memory in a graphical form. This is very helpful for inspection of data, such as sampled analogue signals. We can display such memory values as raw data on a time - axis or even better, we can display the samples a frequency axis. In the 2nd option CCS is performing a FOURIER - transform, before it displays the data.

Let us inspect this graph feature. The BOOT-ROM contains a sine - value lookup table of 512 samples for a unit circle of 360 degree. The data format is 32-bit signed integers in fractional I2Q30 - format. The start address of this table is 0x3FE000.

Open a graph window and enter the properties from the left hand side of Slide 3 - 36:



10. Graph Window

➔View ➔ Graph ➔ Time/Frequency...

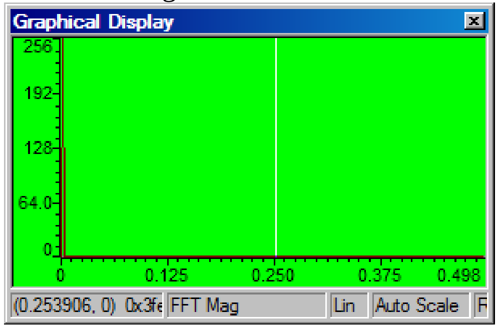
– View of sine-value look-up table in BOOT - ROM (0x3FE000)
– 512 values (32 – bit signed, I2Q30)

Graph Property Dialog

Display Type	Single Time
Graph Title	Sine Look - up Table
Start Address	0x3fe000
Page	Data
Acquisition Buffer Size	512
Index Increment	1
Display Data Size	512
DSP Data Type	32-bit signed integer
Q-value	30
Sampling Rate (Hz)	1
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s

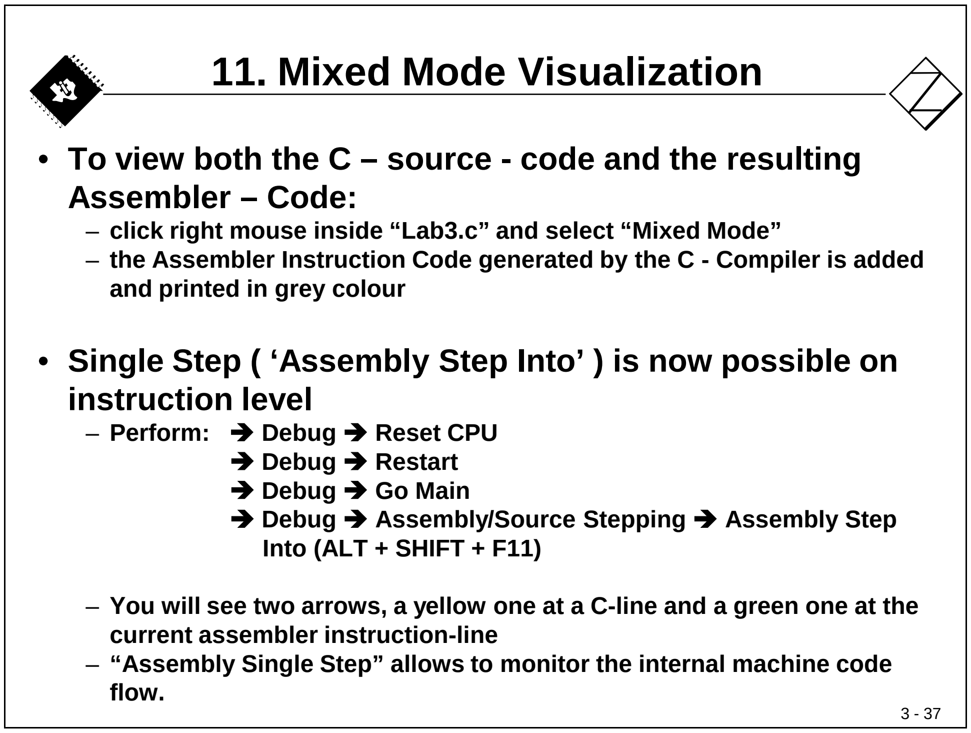
3 - 36

Optionally, right click into the graph window, select “Properties” and change the display type from “Single Time” into “FFT Magnitude”:



Mixed Mode C and Assembly

An important test method for inspecting both C and the resulting assembly code from the C - compilation is called "Mixed Mode" - display. This option allows us not only to inspect and verify the device steps both on C high-level language, but also on the device native environment assembly language.



11. Mixed Mode Visualization

- **To view both the C – source - code and the resulting Assembler – Code:**
  - **click right mouse inside "Lab3.c" and select "Mixed Mode"**
  - **the Assembler Instruction Code generated by the C - Compiler is added and printed in grey colour**

- **Single Step ('Assembly Step Into') is now possible on instruction level**
  - **Perform: ➔ Debug ➔ Reset CPU**
    ➔ Debug ➔ Restart
    ➔ Debug ➔ Go Main
    ➔ Debug ➔ Assembly/Source Stepping ➔ Assembly Step Into (ALT + SHIFT + F11)

  - **You will see two arrows, a yellow one at a C-line and a green one at the current assembler instruction-line**
  - **"Assembly Single Step" allows to monitor the internal machine code flow.**

3 - 37

Although this test method is not always required, especially not at the beginning of a tutorial, it allows us to benchmark the efficiency of the C compiler.

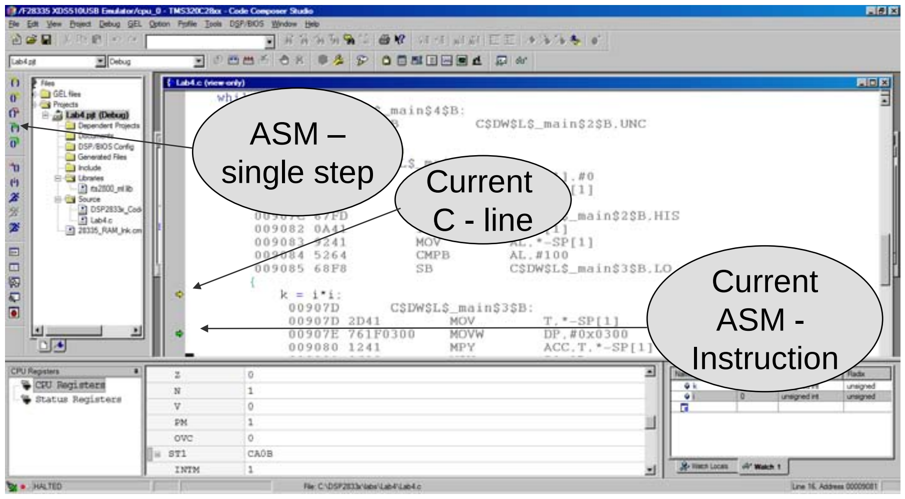
Also later, when we have to use assembly optimized libraries and to design time critical control loops, it will be important to optimize programs. For high speed control loops, for example in Digital Power Supply, where sometimes the control loop (e.g. sample measurement values, compute new output values and adjust PWM - outputs) runs at 500 kHz or even at 1 MHz, we deal with time intervals of 1/1MHz = 1µs. Assuming that the device runs at 150MHz (= 6.667 Nanoseconds clock period), we can execute just 150 machine code instructions in such a loop. In such circumstances an option to monitor a code flow on assembly language level is very helpful.

Assembly Single Step Mode

In mixed - mode visualization of C - modules we can perform test steps both on C and Assembly Language level. Code Composer Studio supports the 2nd test mode by two more icons (green icons at the left hand side):

- Assembly Single Step
- Assembly Step Over

12. Assembly Single Step



```
    009082 0A41
    009083 9241          MOV          AL,*-SP[1]
    009084 5264          CMPB         AL,#100
    009085 68F8          SB           C$DW$L$_main$3$B,LO
    {
        k = i*i;
     00907D         C$DW$L$_main$3$B:
     00907D 2D41          MOV          T,*-SP[1]
     00907E 761F0300      MOVW         DP,#0x0300
     009080 1241          MPY          ACC,T,*-SP[1]
```

Note: When you are done with mixed mode, switch the visualization back to „Source Mode“ (right mouse click).

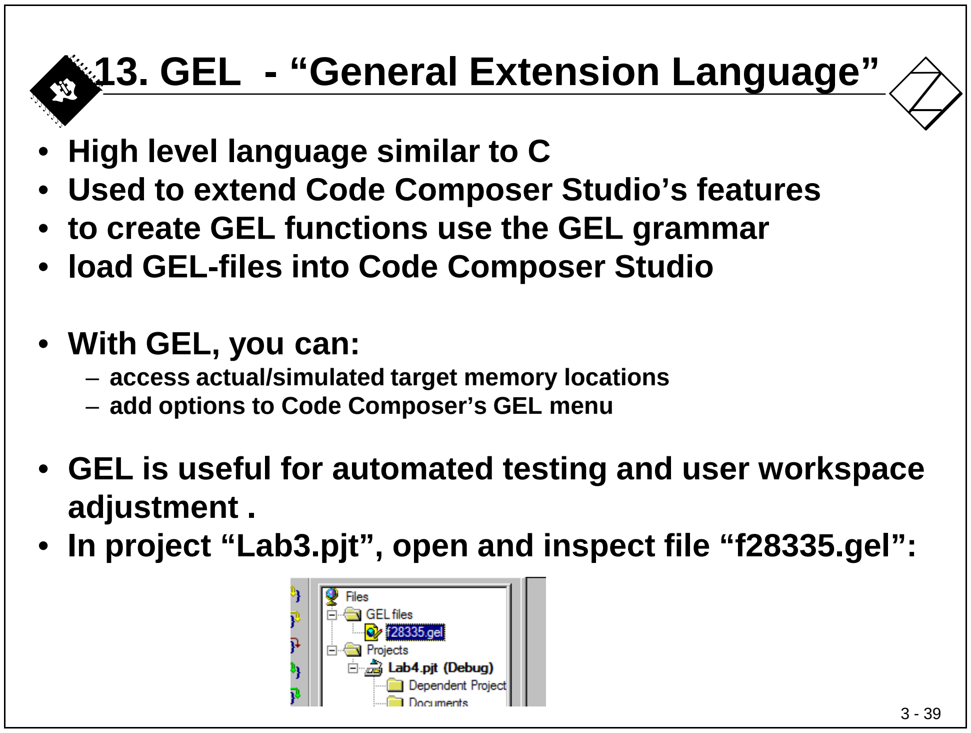
3 - 38

If you use “Assembly Single Step”, the code is executed machine code line by machine code line. The green arrow marks the next following assembly line. The yellow arrow remains at the corresponding C - line, as long as we deal with the assembly results of that line.

When you have finished the mixed mode tests, please switch back to “Source Mode” (right mouse click).

GEL General Extension Language

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



13. GEL - “General Extension Language”

- **High level language similar to C**
- **Used to extend Code Composer Studio’s features**
- **to create GEL functions use the GEL grammar**
- **load GEL-files into Code Composer Studio**

- **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 .**
- **In project “Lab3.pjt”, open and inspect file “f28335.gel”:**

3 - 39

By default, a start GEL - file is automatically added to a new project by Code Composer Studio. Expand the “GEL” - directory and open file “F28335.gel”. Since we are still in an early stage of the classes, please do not change anything inside this file.

We can also add more GEL - Files to a project. To manually add a 2nd GEL - file use:

➔ File ➔ Load GEL… ➔
C:\tidcs\C28\DSP2833x\v131\DSP2833x_headers\gel\DSP2833x_Peripheral.gel

This file will extend the options in the “GEL” - menu of CCS by adding a new entry “Watch DSP2833x Peripheral Structures”. This new entry allows us to extend the Watch - Window options, which we will use in the following chapters.

Lab 3: beginner's project

Objective

The objective of this lab is to practice and verify the basics of the Code Composer Studio Integrated Design Environment. The following procedure will summarize all the steps discussed in this chapter.

Procedure

Open Files, Create Project File

1. Using Code Composer Studio, create a new project, called **Lab3.pjt** in C:\DSP2833x\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 = 0;
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: **Lab3.c**

3. Add the following files to your project:

   ➔ Project ➔ Add Files to project

   - Lab3.c
   - C:\CCStudio_v3.3\C2000\cgtools\lib\rts2800_ml.lib
   - C:\tidcs\c28\DSP2833x\v131\DSP2833x_common\cmd\28335_RAM_lnk.cmd

Note1: The file "rts2800_ml.lib" is part of the Code Composer Studio environment and has been stored at the hard disk of your PC during installation time. The file location might be different at your computer. If you cannot locate the correct path, ask your teacher.

Note2: The file "28335_RAM_lnk.cmd" is part of the header file package for the F28335 (sprc530.zip). If you cannot find that file on your PC, ask your teacher to install the package for you. The zip-file can be found in the "Software" folder of this Teaching CD-ROM.

4. Verify that in Project → Build Options → Linker the Autoinit- Model field is set to: "Run-time-Autoinitialisation [-c]"

5. Set the stack size to 0x400: Project → Build Options → Linker → Stack Size

   Note: The stack memory is used by the compiler to store local variables, parameters and the processors context in case of hardware interrupts. It is our task to specify a certain amount of memory for this purpose and 0x400 is sufficient for this lab.

6. 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. Debug as necessary. To open up more space, close any open files or windows that you do not need at this time.

8. Load the output file to the device. Click: File → Load Program and choose the output file "Lab3.out", which you just generated. Note: the file is stored in the "Debug" subdirectory in the project folder.

   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

9. Reset the DSP by clicking on → Debug → Reset CPU, followed by → Debug → Restart

10. 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 "Lab3.c" is highlighted and that the yellow arrow for the current Program Counter is placed on the first line of function "main()".

11. Disable the watchdog: ➔ GEL ➔Watchdog ➔Disable_WD

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 variable. To display 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. Try changing the value in the "Radix" column.

16. Perform a single-step through your program by clicking: Debug → Step Into (or use function Key F11). Repeat F11 and watch your variables.

17. Place a Breakpoint in the Lab3.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. Now we will exercise with the real-time debug mode. Make sure that all breakpoints have been deleted or disabled. Next, reset the device:

    ➔ Debug ➔ Reset

    This reset will set the device in its default state, including the watchdog unit, which is enabled after reset. To exercise without this unit, do:

    ➔ GEL ➔Watchdog ➔Disable_WD

    Start the real-time debug:

    ➔ GEL ➔ Realtime Emulation Control ➔ Run_Realtime_with_Restart

    Now the code is running in real-time. The new feature is that we can interact with the device, while the code is running. To practice using this:

    ➔ Right click into the Watch Window and enable the option "Continuous Refresh"

    The contents of the Watch Window are updated frequently. The JTAG - controller uses cycles, in which the core of the device does not access the variables to "steal" the information needed to update the window. This real-time mode is called "Polite", as you can see in the lower left corner of CCS.

    When you are done, stop the real - time mode by:

    ➔ GEL ➔ Realtime Emulation Control ➔ Full_Halt

    Note: the test mode **"Run_Realtime_with_Reset"** will first perform a reset followed by a direct start of the device from its reset address. The device will follow its hardware boot sequence (see Chapter 15) to begin the code execution. Since the Peripheral Explorer Board sets the coding pins to "Branch to FLASH" by default, it would start code stored in FLASH. The problem is, that so far we have not stored anything in FLASH (we will do this in Chapter 14). By using **"Run_Realtime_with_Restart"** we force CCS to place the program counter at the start address of our project in RAM (a function called "c_int00") and to start from this position.

19. Inspect the internal device registers:

    ➔ View ➔ Registers ➔ CPU Registers

    The option "Tree View" (right mouse click) allows you to inspect registers more in detail. At this early stage of the tutorial it is not important to understand the meaning of all the bit fields and registers, shown in this window. However, you should get the feeling, that with the help of such windows, you can get control information on all internal activities of the device.

    There are two core registers, ST0 and ST1, which combine all core control switches and flags of the CPU, such as carry, zero, negative, overflow, sign extension mode, interrupt enable and so on. An inspection of these flags and bits allows you to immediately monitor the status of the CPU in a certain spot of code.

    The 32-bit registers ACC ("accumulator"), P ("Product") and XT ("eXtended Temp") are the core math registers of the fixed - point arithmetic unit.

    The 32-bit registers XAR0 to XAR7 are general-purpose registers, often used as pointer registers or auxiliary registers for temporary results.

    The register PC ("Program Counter") points always the address of the next machine code instruction in code memory. The register RPC ("return program counter") stores the return address of a function, which has called a sub-routine.

20. Let us open another control window, the "Memory Window":

    ➔ View ➔ Memory

    Enter the address for variable k (&k) in the address box (top left hand side).

    This window allows us to inspect any physical memory location of the device, including RAM, FLASH, OTP and Boot - ROM. Since the core of this device is a Digital Signal Processor, we have always to remember that there are two memory spaces, code and data. To inspect variables, we have to select "data space". For machine code instructions inspection we have to look into "code space". The selection is made in the center box at the bottom of this window.

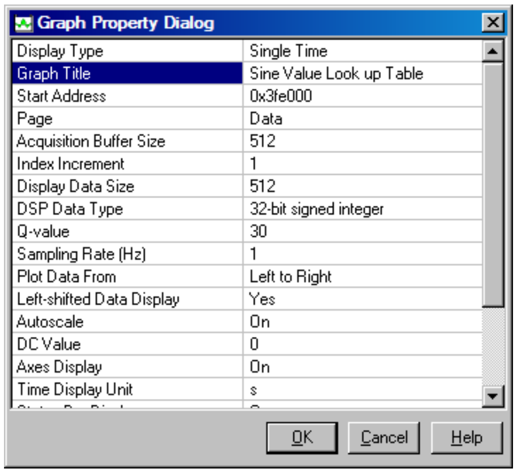
    The bottom left box allows us to specify the display mode of the 16-bit memory locations in different form. Try using the different formats available: 16-bit hexadecimal, signed integer, unsigned integer and binary.

21. A unique feature of Code Composer Studio (CCS) is the ability to display any region of memory in graphical form. This is very helpful for inspection of data, such as sampled analogue signals. We can display such memory values as raw data on a time - axis or even better, we can display the samples a frequency axis. In the 2nd option CCS is performing a FOURIER - transform, before it displays the data.

Let us inspect this graph feature:

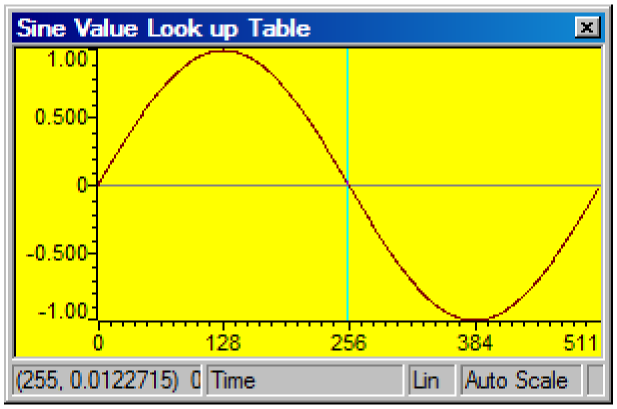
➔ View ➔ Graph ➔ Time/Frequency

The BOOT-ROM contains a sine value lookup table of 512 samples for a unit circle of 360 degree. The data format is 32-bit signed integers in fractional I2Q30 - format. The start address of this table is 0x3FE000. Enter the following parameters:

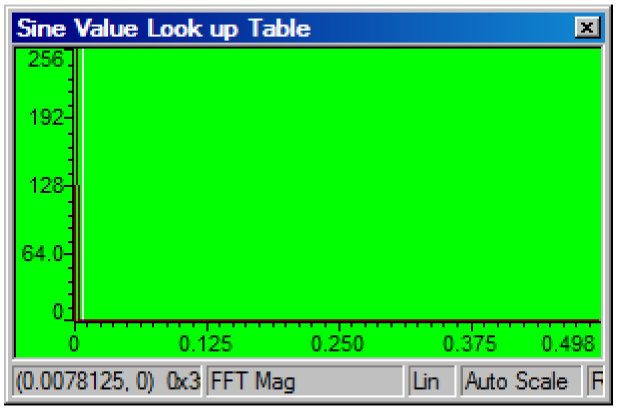


Display Type	Single Time
Graph Title	Sine Value Look up Table
Start Address	0x3fe000
Page	Data
Acquisition Buffer Size	512
Index Increment	1
Display Data Size	512
DSP Data Type	32-bit signed integer
Q-value	30
Sampling Rate (Hz)	1
Plot Data From	Left to Right
Left-shifted Data Display	Yes
Autoscale	On
DC Value	0
Axes Display	On
Time Display Unit	s

As a result, the graph window should display a single sinusoidal signal:



Change the properties from "Single Time" to "FFT Magnitude". Since the signal is a pure harmonic signal, the FFT will show just one single frequency line:



Close the graphical window, when you are done.

22. Now let us exercise the mixed mode test. An important test method for inspecting both C and the resulting assembly code from the C - compilation is called "Mixed Mode" - monitoring. This option allows us to inspect and verify the device steps both on C high-level language, and also in the device native environment assembly language. Right click into "Lab3.c" and select "Mixed Mode".

    Perform:

    ➔ Debug ➔ Reset CPU

    ➔ Debug ➔ Restart

    ➔ Debug ➔ Go Main

    ➔ Debug ➔ Assembly/Source Stepping ➔ Assembly Step Into (ALT+SHIFT + F11)

    Optionally, use the green icon "Assembly Single Step" on left hand side of CCS.

    If you use "Assembly Single Step", the code is executed machine code line by machine code line. The green arrow marks the next following assembly line. The yellow arrow remains at the corresponding line of C code, as long as we deal with the assembly results of that line.

    When you have finished the mixed mode tests, please switch back to "Source Mode" (right mouse click).

23. Inspect the GEL - script files. The General Extension Language (GEL) is a high-level script language. Using a *.gel file, the user can expand the features of Code Composer Studio or perform recurrent steps automatically. By default, a start GEL - file is automatically added to a new project by Code Composer Studio.

    Expand the "GEL" - directory and open the file "F28335.gel".

    Since we are still in an early stage of the classes, please do not change anything inside this file.

    We can also add more GEL - Files to a project. To manually add a 2nd GEL - file use:

➔ File ➔ Load GEL… ➔
C:\tidcs\C28\DSP2833x\v131\DSP2833x_headers\gel\DSP2833x_Peripheral.gel

This file will extend the options in the "GEL" - menu of CCS by adding a new entry "Watch DSP2833x Peripheral Structures". This new entry allows us to extent the Watch - Window options, which we will use in the following chapters.

24. 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 "Lab3.wks"

25. Close the project by Clicking Project → Close Project and close all open windows that you do not need any further.

End of Lab 3

Blank page