Program Development Tools

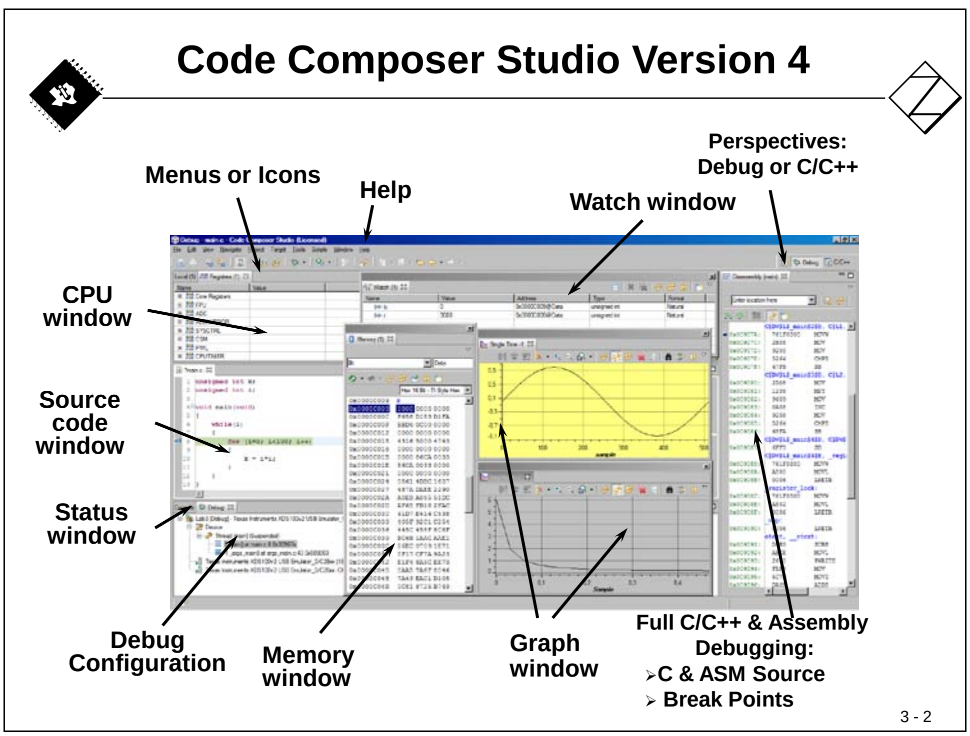
Introduction

The objective of this module is to understand the basic functions of the Code Composer Studio® (CCS) 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, Version 4

Note: This chapter explains the use of Code Composer Studio, Version 4 and later. This revision is based on Eclipse and introduced a major change of the design environment compared to earlier CCS versions. If you use an older version, please refer to the previous releases of this teaching CD-ROM.

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



Module Topics

CCS 4: Eclipse Concepts

With CCS version 4 Texas instruments moved the Integrated Design Environment to an Eclipse (www.eclipse.org) open source software framework. Hence understanding some of the basic concepts of Eclipse will lead to a better understanding of CCSv4. Some of the more commonly referenced concepts are described below.

CCS4 Eclipse Concepts

CCS 4 - based on Eclipse

- **Open source framework (www.eclipse.org)**

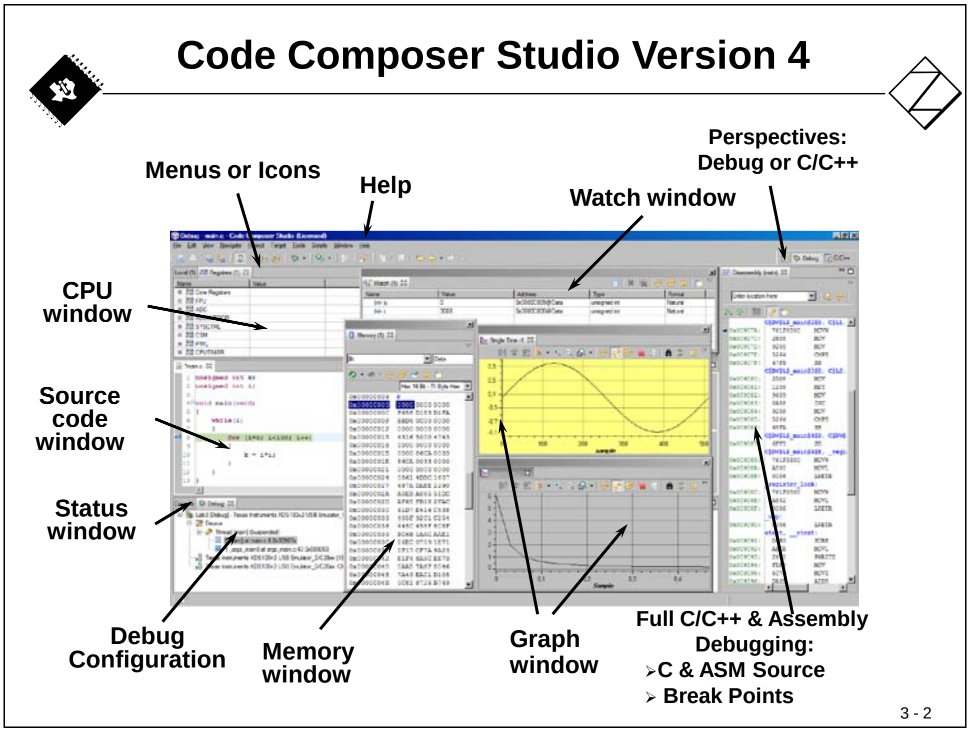
Commonly referenced categories:

- **Workbenches**
- **Workspaces**
- **Perspectives**
- **Views**
- **Resources**
  - **Projects**
  - **Files**

3 - 3

Workbench

A Workbench contains all the various views and resources used for development and debug. Multiple CCSv4 Workbench windows can be opened ('Window->New Window'). While each Workbench window can differ visually (arrangement of views, toolbars and such), all windows refer to the same workspace and the same running instance of CCSv4 - if a project is opened from one Workbench, that same project will be open in all the Workbench windows.



Workspace

The workspace is the main working folder for CCSv4 and where it stores project information to manage all the projects that you define to it. This is the case even if the projects themselves do not physically reside inside the workspace folder. CCSv4 Workspaces are not to be confused with CCSv3 workspace files (*.wks), which have more in common with CCSv4 Perspectives than they do with CCSv4 workspaces. The default location of any new projects created in CCSv4 will be within the workspace folder. Once a project has been defined to the workspace, it will be visible in the 'C/C++ Projects' view and can be opened and closed and such. To define an existing CCSv4 project to the workspace, it will need to be imported into CCSv4.

CCSv4 will prompt the user for the workspace folder location when launching CCSv4. The workspace folder is also used by CCSv4 to store other information such as user preferences, custom perspectives, cached data for plug-ins, etc.

Multiple workspaces may be maintained (for example, one for each user), however only one can be active within each CCSv4 instance. The 'File->Switch Workspace...' option can be used to switch between the workspaces. Each workspace would have its own stored user preferences and projects associated with it.

Perspective

A perspective (compare Slide 3-2) defines the initial set and layout of views in the Workbench window. Each perspective provides a set of functionality aimed at accomplishing a specific type of task. For example, the default 'C/C++' perspective displays views most commonly used during code development, such as the 'C/C++ Projects' view, 'Outline' view and the Editor. When a debug session is started, CCSv4 will automatically switch to the 'Debug' perspective, which (by default) displays the 'Debug' view, 'Watch' view and 'Local' view. Also in the 'Debug' perspective, menus and toolbars associated with debugging (such as target connect, load program, reset target, etc) are now available. Users can also manually switch between perspectives. Any changes made to a perspective will be preserved (but can be reset to the default arrangement via ➔ Window➔ Reset Perspective). New perspectives can be created simply by saving the current perspective as a new name (➔ Window ➔Save Perspective As...).

Perspectives can be easily switched between perspectives by clicking on the perspective icons in the upper right corner.

Views

Views are windows within the main Workbench window that provide visual representation of some specific information. The Workbench window mainly consists of the editor and a collection of views. Examples of some views are “C/C++ Projects”, “Debug”, “Outline”, “Memory”, “Disassembly”, etc.

Most of the views in CCSv4 are available from the main “View” menu.

Resources

“Resources” is a collective term for the projects, folders, and files that exist in the Workbench.

Projects

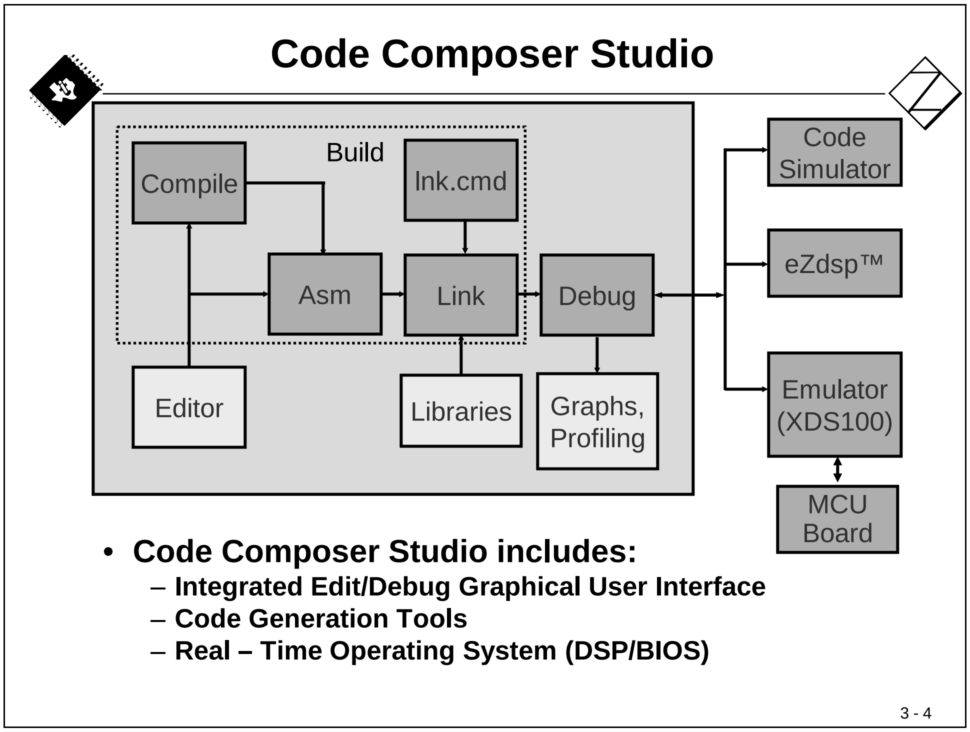
“Projects” typically contain folders and files. Like the workspace, projects map to directories in the file system.

Files

“Files” can either be added or linked to a project. When a file is added to a project, the file is copied to the root location of the project directory. This differs from the concept of “adding” a file to a CCSv3 project, where it would not make a local copy, but simply make a reference to where the file is located (you were *adding* a reference to the file in your project). To achieve the same functionality with CCSv4 projects, there is also the option to “link” a file to a project. This will simply have the project create a reference to the file instead of copying the file into the project directory.

The Software Flow

The following slide (Slide 3-4) 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 Eclipse based Integrated Design Environments, 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 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.

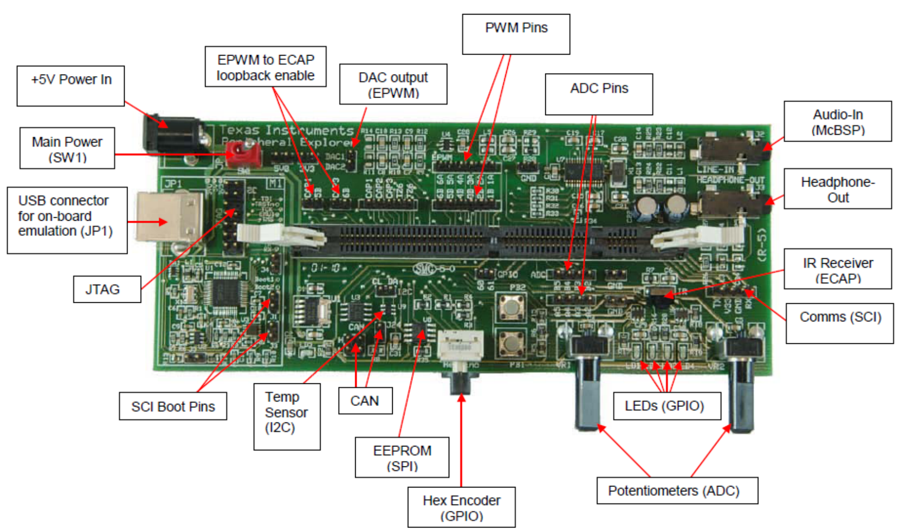
Before we inspect some basic features of Code Composer Studio Version 4 more in detail, we will first discuss the hardware setup for lab exercises that follow.

Lab Hardware Setup

The following slides illustrate the hardware target that will be 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.

Slide 3-5 reveals all peripheral units, which are populated at the Peripheral Explorer Board (Texas Instruments part number: TMDSPREX28335).

Peripheral Explorer Board



3 - 5

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)

Slide 3-6 shows the F28335 Control Card, which we will use for our teaching course.

Teachers: Please note that the F28335ControlCard is already bundled with the Peripheral Explorer Board. In case that you need additional spare modules of this control card, the part number is TMDSCNCD28335.

Other versions of C2000 Control Cards are also available.

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 - 6

Code Composer Studio Version 4 - Step by Step

Now let us start to look a little closer at the main parts of Code Composer Studio Version 4 that we need to develop our first project. We will perform the following steps:

Learning by doing - Step by Step

Code Composer Studio V4 - The Basics

1. **CCS workspace and welcome 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 - 7

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

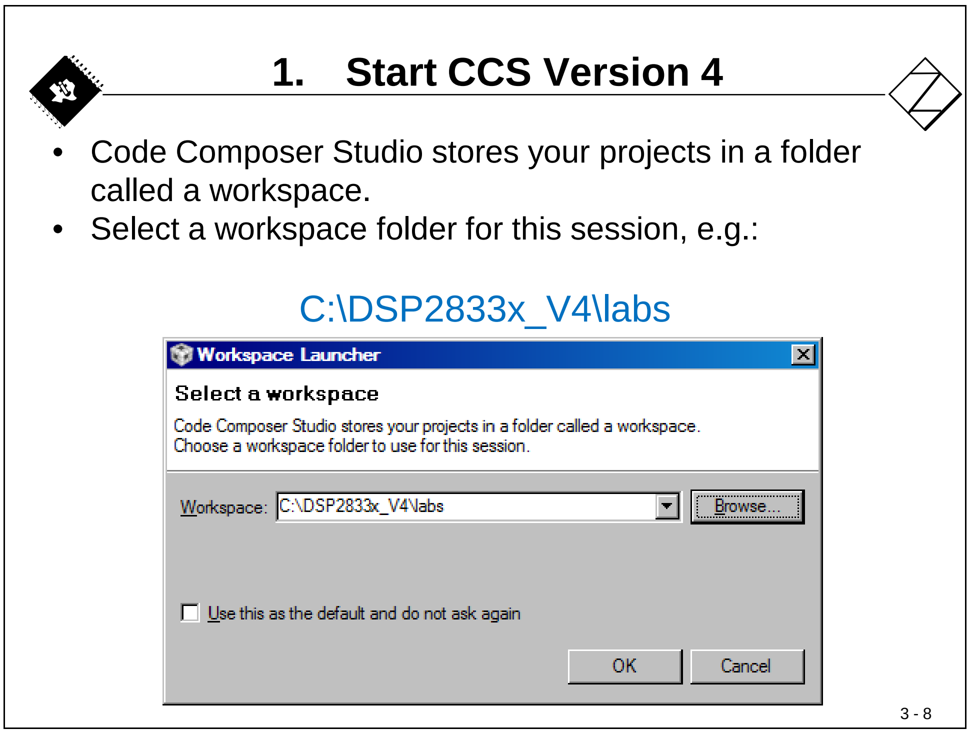
- Open Code Composer Studio
- Create a F2833x - Project, based on C
- Compile, Link, Download and Debug this test program
- Watch Variables
- Continuous run and single - step mode
- Use of Breakpoints
- Use of “Real - Time - Debug” - mode
- View registers
- Mixed Mode (C and Assembler Language)
- General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

Before we start to go into the procedure for Lab3 at the end of this chapter, let us discuss the individual steps using some additional slides.

Start Code Composer Studio Version 4

Once you or your laboratory technician have installed the software tools and the correct Emulator driver for CCS4.1, you can start Code Composer Studio by simply clicking on its desktop icon. If you get an error message, check the correct USB connection of the target board. If everything goes as expected, a message will pop up, asking you to select a "workspace". Code Composer Studio stores your projects in a folder called a workspace. Now you have to choose a workspace folder for this session.

You might have to ask your teacher, which folder you should use in your classroom. For this tutorial, I assume that we store the projects in "C:\DSP2833x_V4\labs".

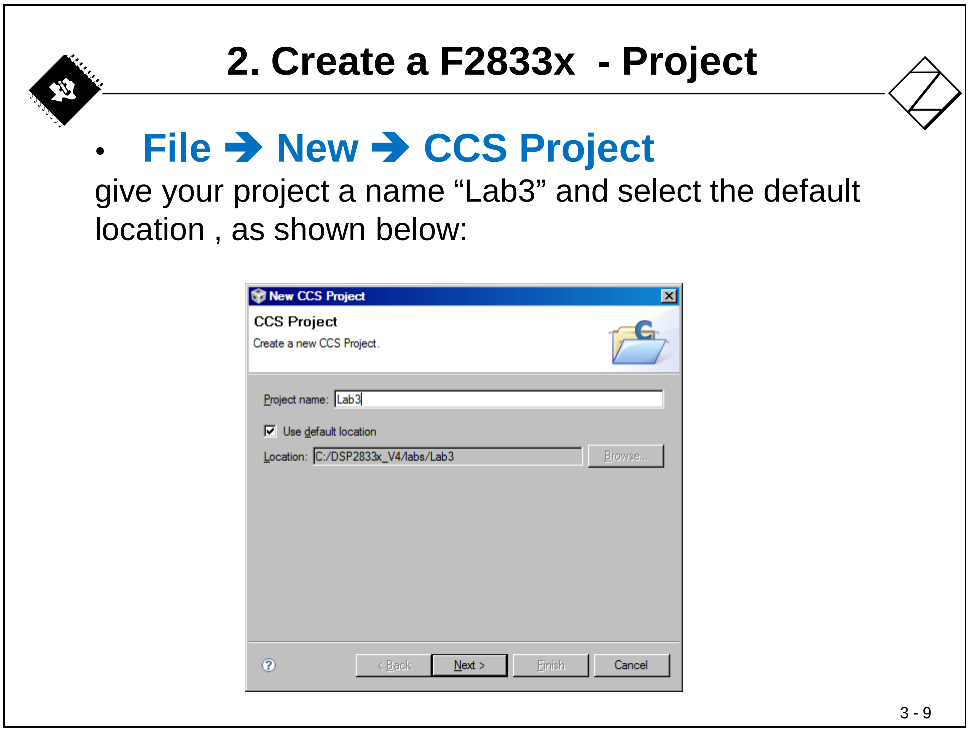


Next, a "welcome" window will appear. As the name suggests, this window shows you essential menus for CCS, such as "Getting Started", "Examples", "What's new" and "Device Information". Although all these information might be very interesting, we will concentrate on our task to generate our first project from scratch.

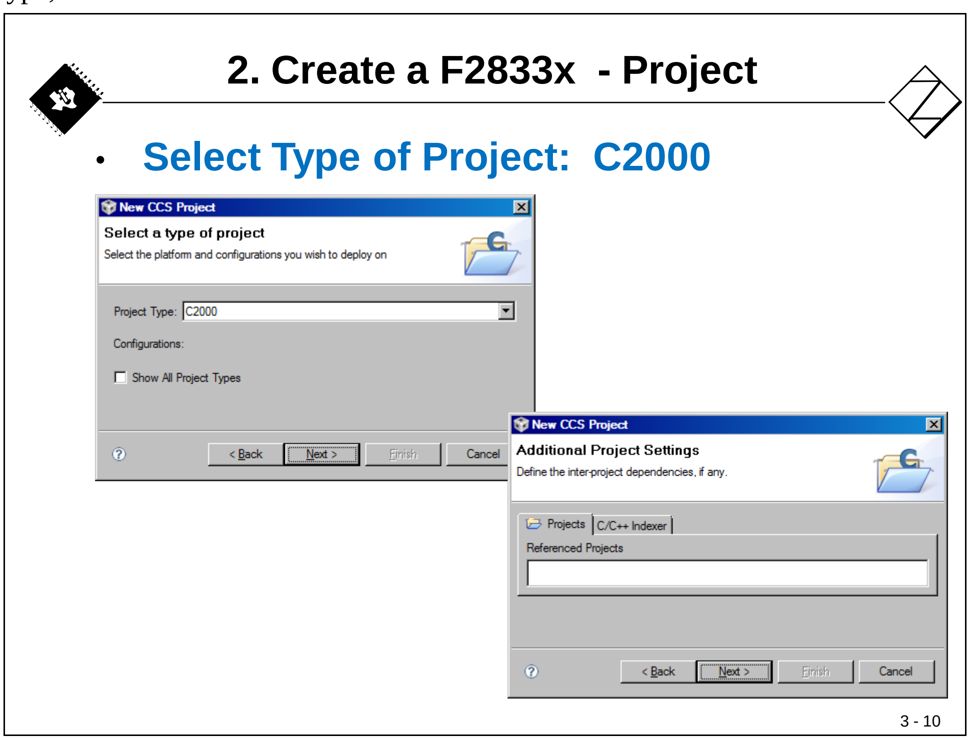
Later you can always return to this welcome page (➔ Help ➔ Welcome).

Create a project

Let us now create our first project. Click on File ➔ New ➔ CCS Project and enter “Lab3”:

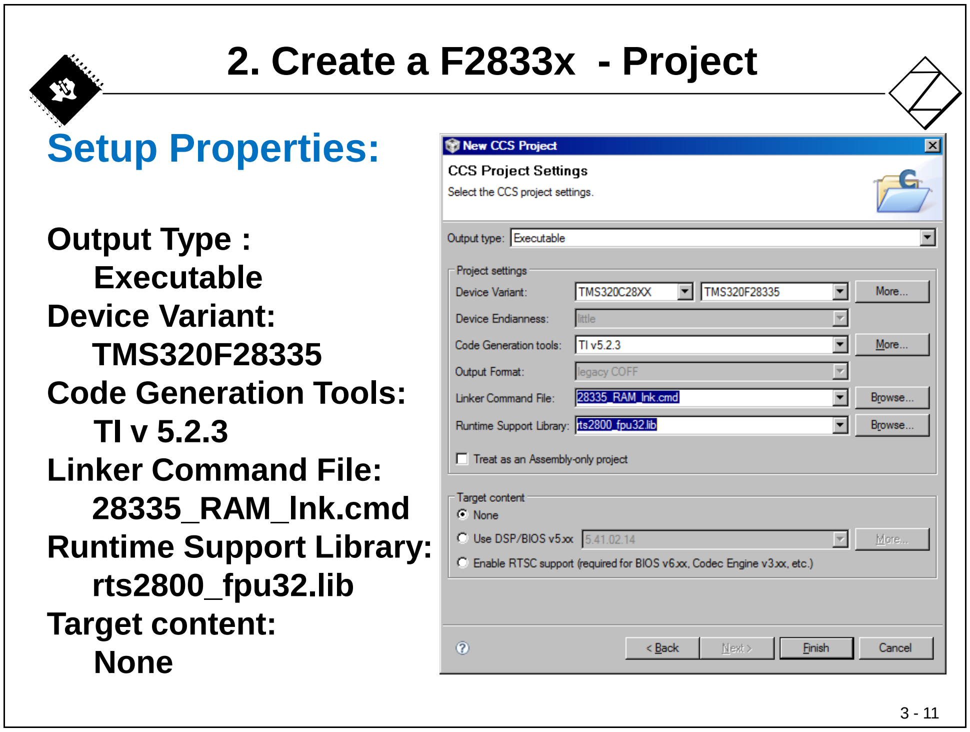


This step is quite similar to most of today’s design environments with one exception. Because CCS4 is also used for C6000, C5000, MSP430 and ARM processors, we have also to define the project type, in our case “C2000”:

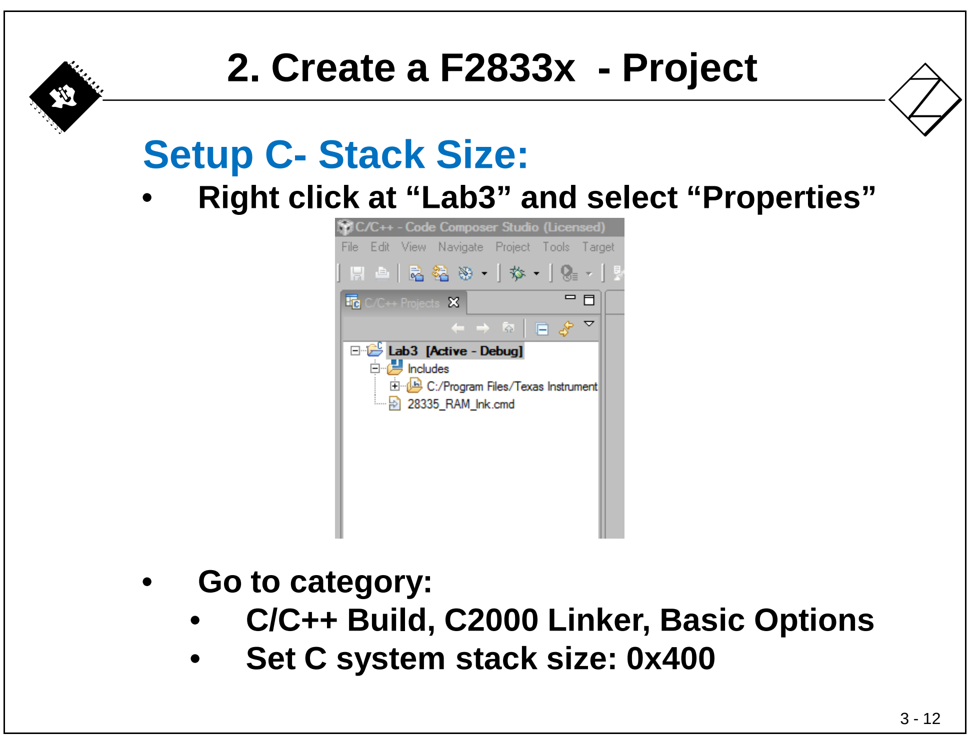


We do not use any inter-project dependencies so far, so click “Next” twice.

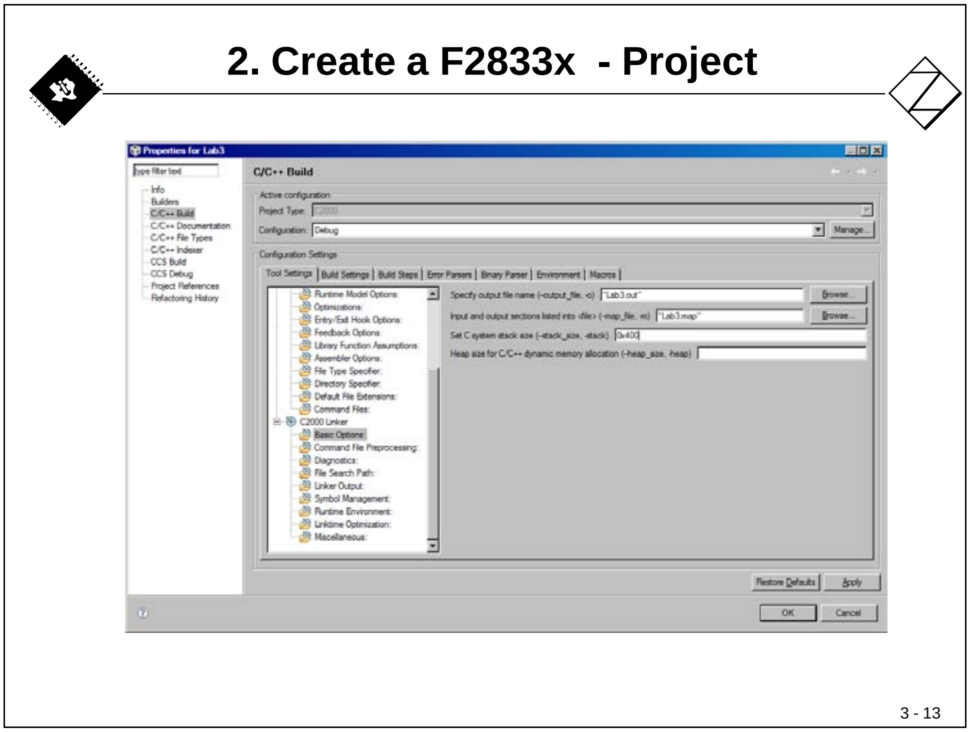
Now we have to set the project properties according to the following Slide 3-11:



Close the project setup by clicking on “Finish”. We are almost done. Cancel the “welcome” window to show the project layout. All C code based programs need a system stack. We have to define its size:



The following Slide 3-13 shows the setup for the stack size. The selected size of 0x400 is a first “rule of thumb” number. Later we can be more specific about the stack usage of code examples.



Do not change the remaining parts of this property window.

Close the window by clicking “OK”.

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.

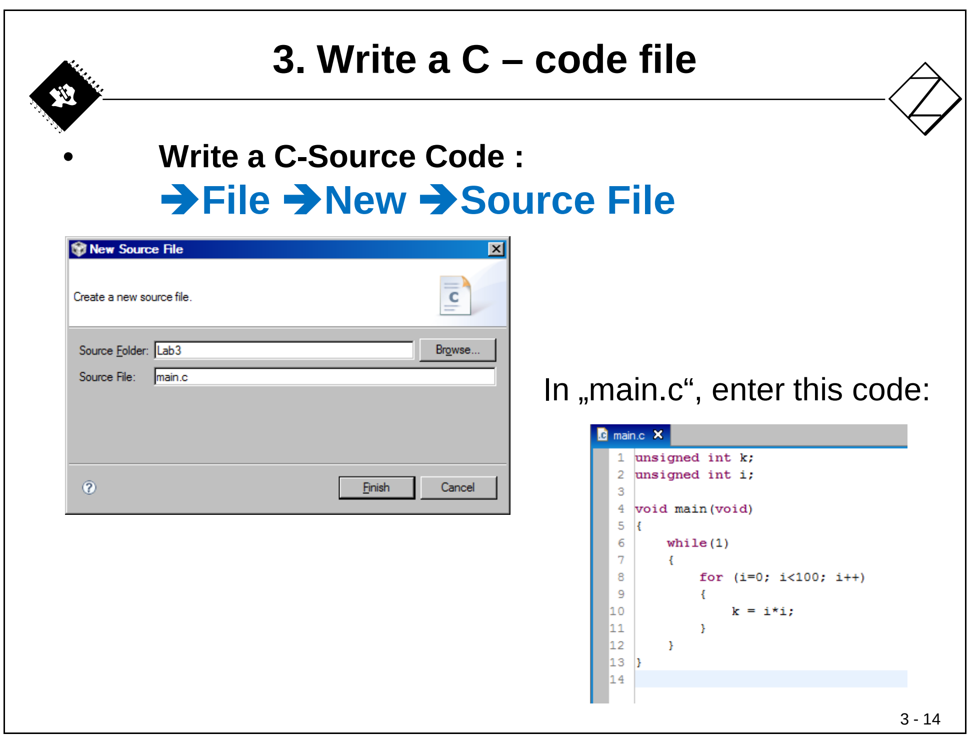
```c
unsigned int k;
unsigned int i;

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

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 the product 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.



Linker Command File

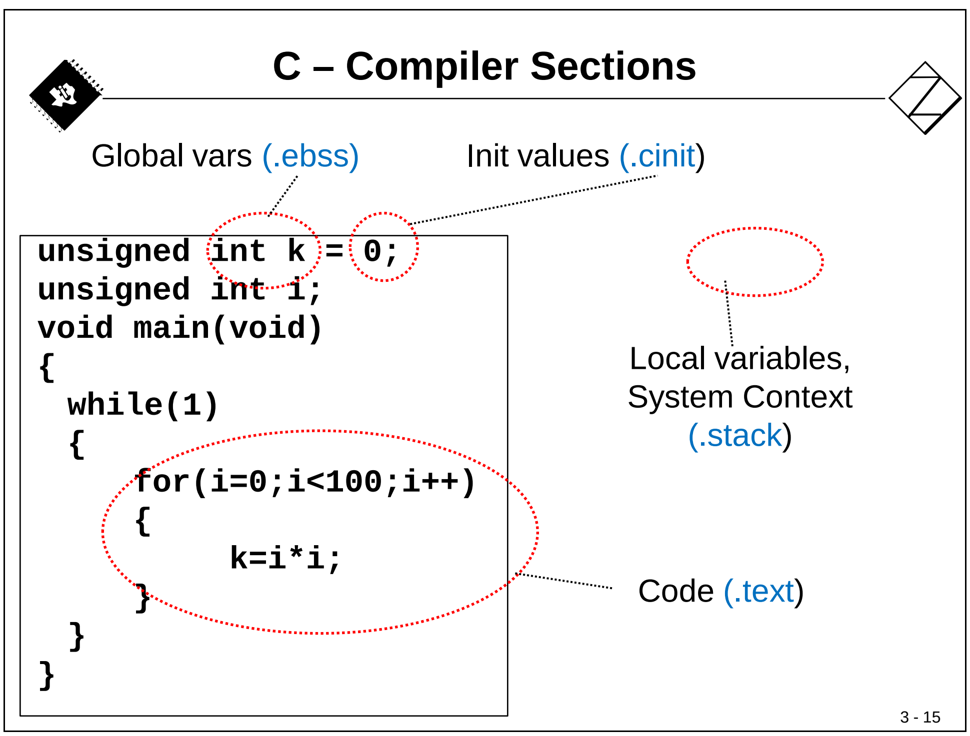
Before we continue with our project, let us first discuss why we added the file "28335_RAM_lnk.cmd" to our project (see Slide 3-11). This file is used to control the "Linker".

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 CCS Version 4 support package.

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



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

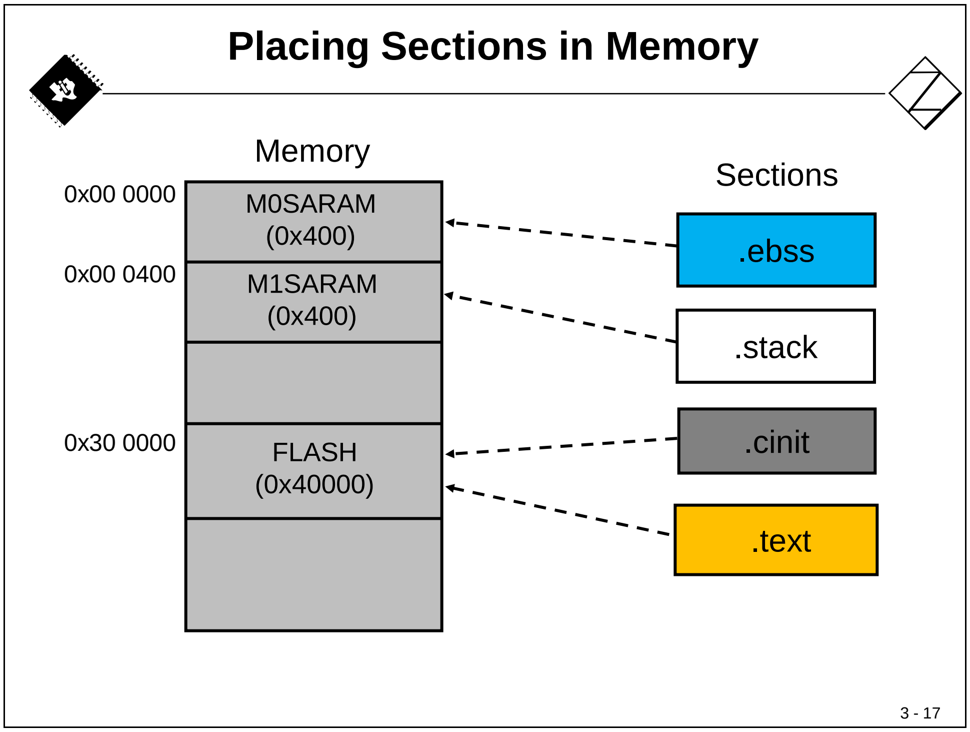
Note: (*)During development initialized sections could be linked to RAM since the emulator can be used to load the RAM

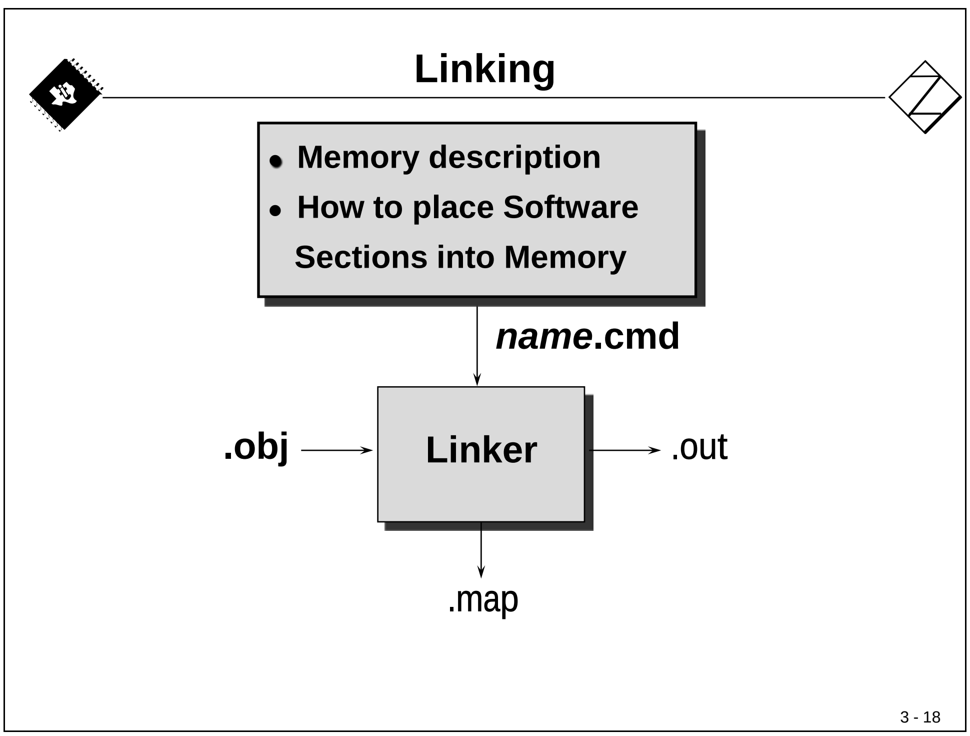
3 - 16

Linking Sections to Memory

The following Slide (3-17) gives an example on 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: Our lab “lab3” will be based on volatile memory usage only, so the following explanation is a more generic one.



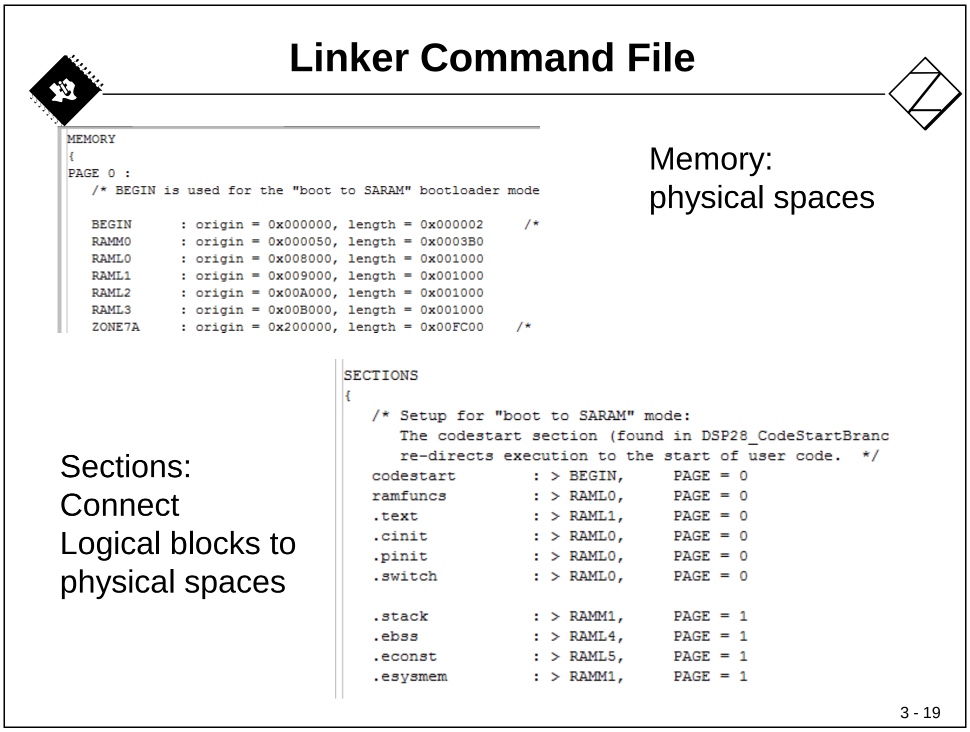


The procedure of linking connects one or more object files (*.obj) into an output file (*.out). This output file contains not only the absolute machine code for the Digital Signal Controller, but also information used to debug, to flash the controller and for more JTAG based tasks. NEVER take the length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

Now let us inspect the linker command file “28335_RAM_lnk.cmd”. Basically the file consists of two parts, “MEMORY” and “SECTIONS”.

“MEMORY” declares all available physical memory of the device. The declaration is split in “PAGE 0” – for code memory and “PAGE 1” for data memory.

Please recall that the F28335 is a Digital Signal Controller and that one of the properties of DSPs is to have a “Harvard”-Architecture, which has two memory spaces, one for code and one for data.



When you inspect the file, you will find that our sections are actually allocated in:

- .text is allocated in code address space 0x9000 (RAML1)
- .cinit is allocated in code address space 0x8000 (RAML0)
- .ebss is allocated in data address space 0x0C000 (RAML4)
- .stack is allocated in data address space 0x0400 (RAMM1)

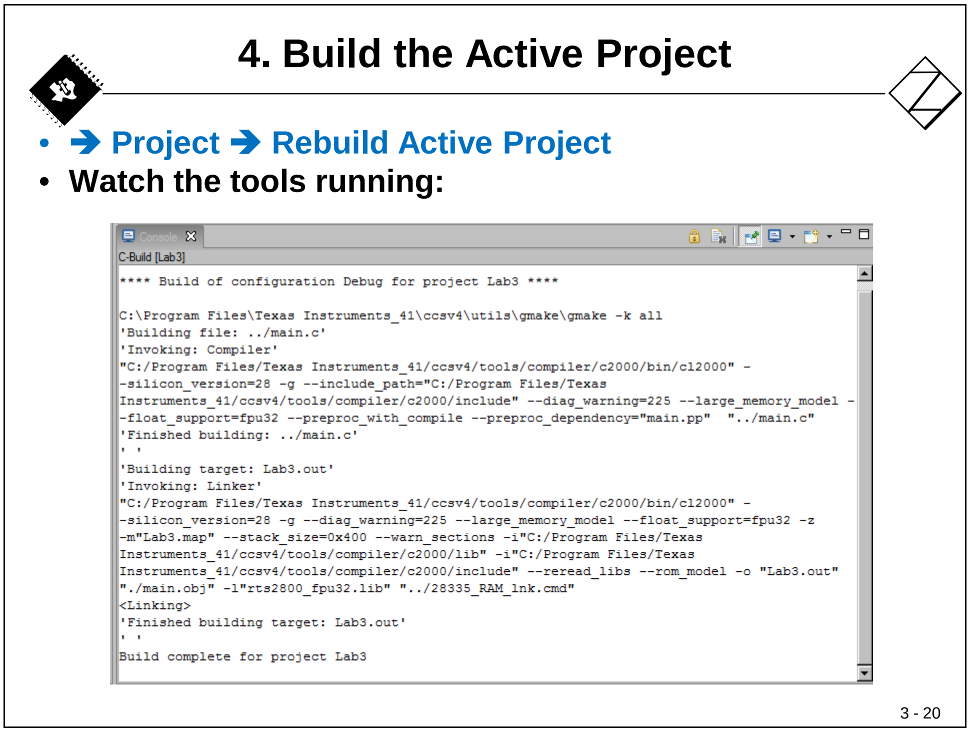
Close the file “28335_RAM_lnk.cmd”, when you are done.

Build the active project

Now let us resume or lab and build the machine code. This step includes the compilation of all source code files (C and Assembler) and the linking of all modules and libraries, which are part of the project, into a single output file. This file contains a lot of information, including the machine code for all sections.

➔ Project ➔ Rebuild Active Project

And watch the tools output in the console window:



Hopefully you have the same console output as shown in Slide 3-20 above. If you have error messages or warning, both in red colors, you will have to find out what went wrong. In most cases, not always, the error comment gives you an indication about the cause of the error/warning.

And, you still have the option to ask your teacher!

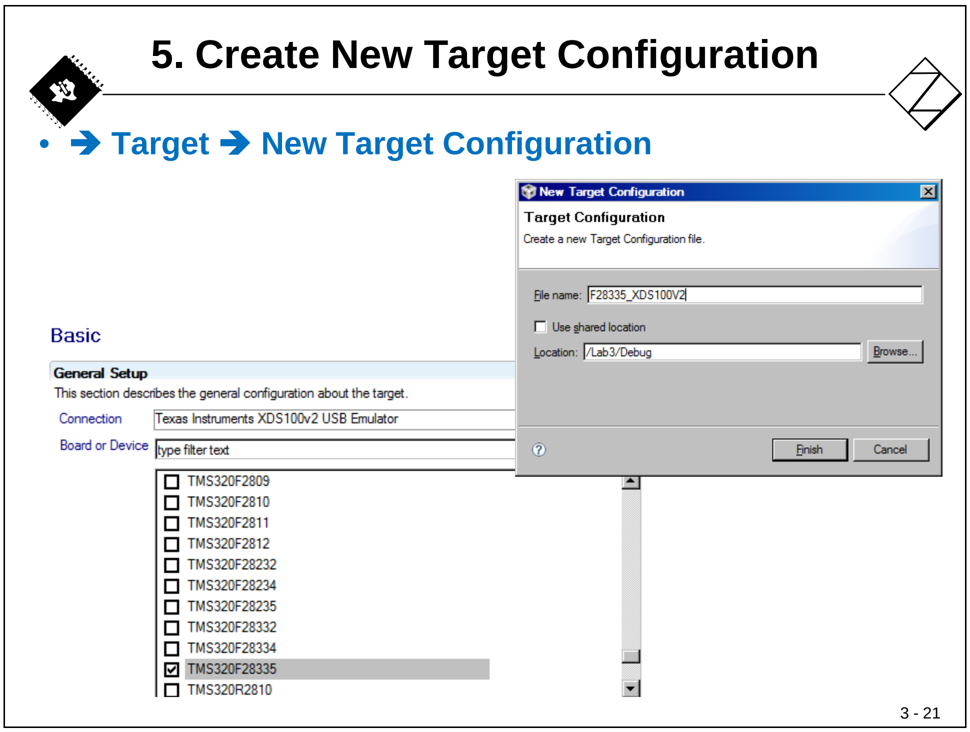
Please do NOT continue with the next steps in case of errors/warnings!

Create a new Target Configuaration

Before we can download the machine code into the F28335, we have to define the “target configuration”.

➔ Target ➔ New Target Configuration

Type a name for the target configuration file in box “File name”. You can use any name here but it makes sense to indicate the JTAG-emulation interface, which we will use for the download later. In case of the Peripheral Explorer Board we use the XDS100V2, so let us call the file “F28335_XDS100V2. The suffix “.ccxml” will be added automatically.



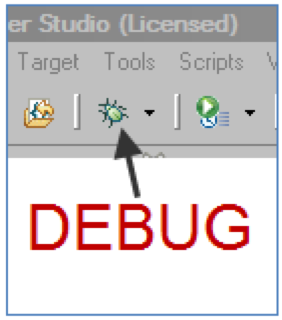
In the window that appears, select the emulator “Texas Instruments XDS100v2 USB Emulator” via the “Connection” pull-down list and select the “TMS320F28335” device checkbox.

Download code into the controller

Now it is time to download the code into the F2833x.

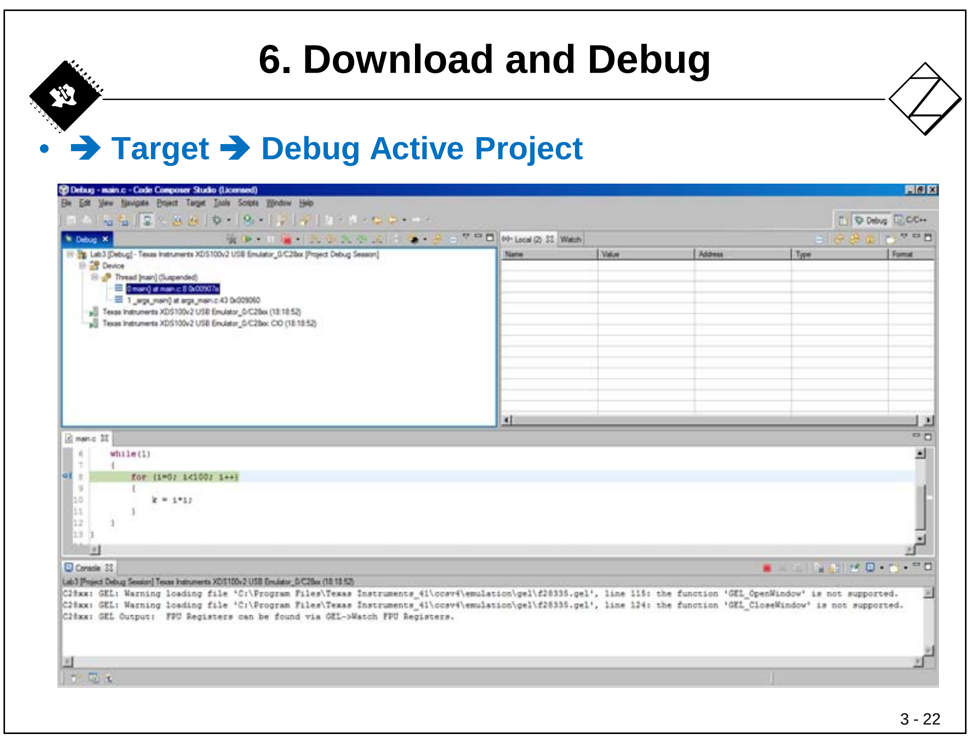
➔ Target ➔ Debug Active Project

This command is also available on the green bug:



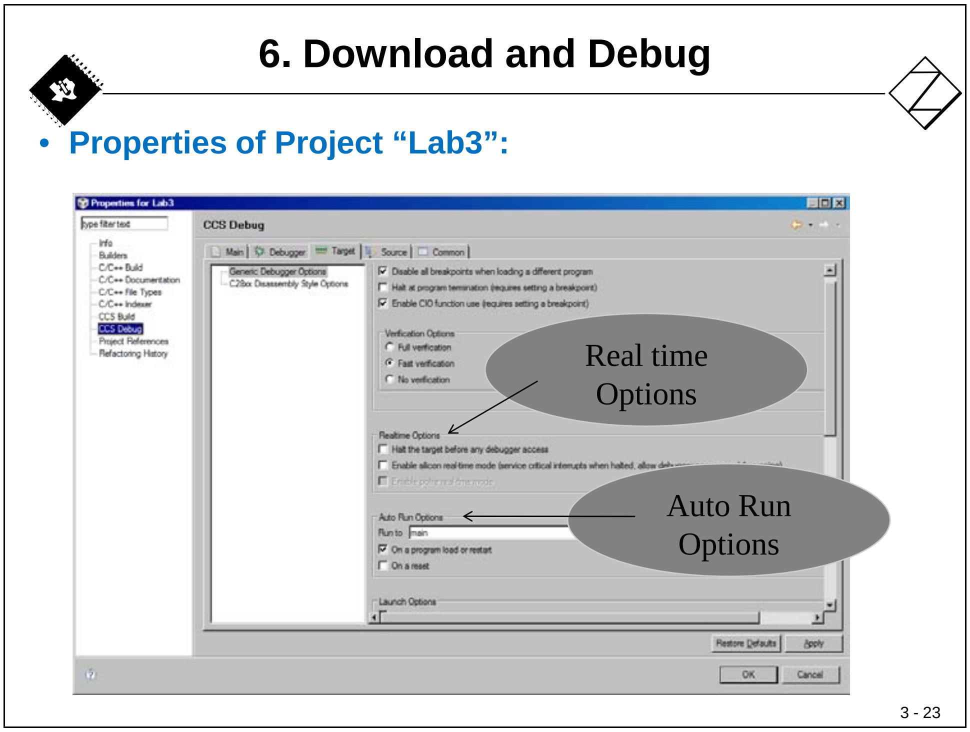
This button combines the following “single action” commands:

- Rebuild Active Project
- Connect Target
- Load Program



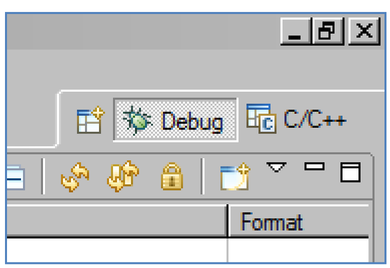
A blue arrow should now point to the “for” – line in code file “main.c”. This is an indication that the machine code has been downloaded properly into the F28335.

Note: The automatic procedure of connecting the target, download code and run the code to the entry point of main can be controlled by the project properties. Right click at the project “Lab3” and select “Properties”. In the “CCS Debug” category, go to the “Target” properties and verify, that “Run to main” on a program load is enabled. Next, close the property window.



Debug Perspective

Code Composer Studio Version 4 allows inspecting a project from different perspectives. All available perspectives are show in the top right corner of CCS. You can always change your perspective of looking into the project. There are at least two perspectives, “C/C++” and “Debug”. For the following tests please make sure that you have selected perspective “Debug”:



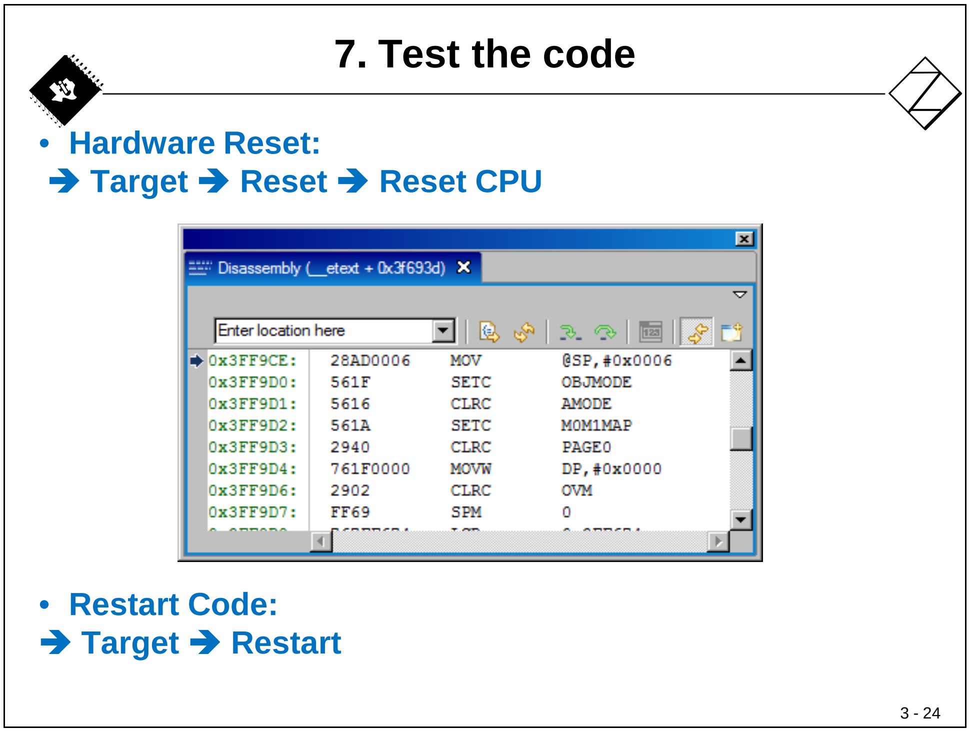
Test the Code

Now that we have successfully downloaded our code into the target, we can perform some basic test commands.

RESET CPU

The most important hardware command for the target is “RESET”. This command will always force the device to a default RESET condition, including all internal peripheral units.

➔ Target ➔ Reset ➔ Reset CPU



A new window, the “Disassembly Window” will open. This window shows the machine code that will be executed in the next clock cycles. However, the JTAG – Emulator has frozen the controller, so that we can take our time to inspect all parts of the CPU and the peripherals. The blue arrow shows the current position of the Program Counter (PC), which is now loaded with the hardware - reset address 0x3FF9CE in Boot-ROM. The purpose of register “PC” is to always point the next machine code instruction to be executed.

We will not discuss the content of the Boot-ROM now; let us postpone its details for a later chapter.

Restart CPU

Another important command is

➔ Target ➔ Restart

This command is often used directly after a RESET command. Its purpose is to bypass the Boot – code and to load the Program Counter (PC) directly with the “entry point address” for the code.

This entry point address can be specified in the project options. For C-language based projects the default address is the environment preparation function "_c_int00" (from library rts2800_fpu.lib"

However, because we have enabled the auto run option to "main()", the restart command will run through "_c_int00" and stop at the beginning of "main()". If this auto run option would have been disabled, we could use ➔ Target ➔ "Go to Main" as a 3rd command.

The Watch Window

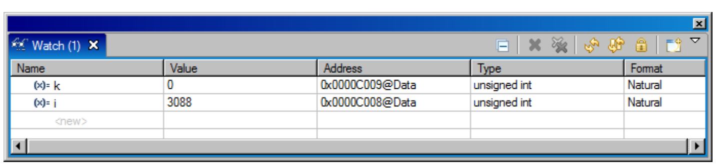
To watch the program's variables, we can use a dedicated window called the "Watch Window". This is probably the most used window during the test phase of a software project. It is good engineering practice to carefully test parts and modules of a software project. For an embedded system we need to 'look' 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 F8 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.

8. Watching variables

- **If not already open, open a Watch Window: ➔ View ➔ Watch**
- **To inspect the global variables 'i' and 'k' we have to add this variables 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 Watch expression"**
- **In column "Format" we can change the data format between decimal, hexadecimal, binary, etc.**



3 - 25

Note that the physical addresses for 'i' and 'k' in column "Address" are shown as 0xC009 and 0xC008 respectively. Can you explain why these two addresses have been used? (*Answer: The linker command file, which we inspected earlier, allocated section .ebss (global variables) to data memory "RAML4" at address block 0x00C000.)*

Code Step Comands

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. In the “Debug” perspective, perform:

9. Perform a Single Step Debug

- **Perform a single step of the target - code:**

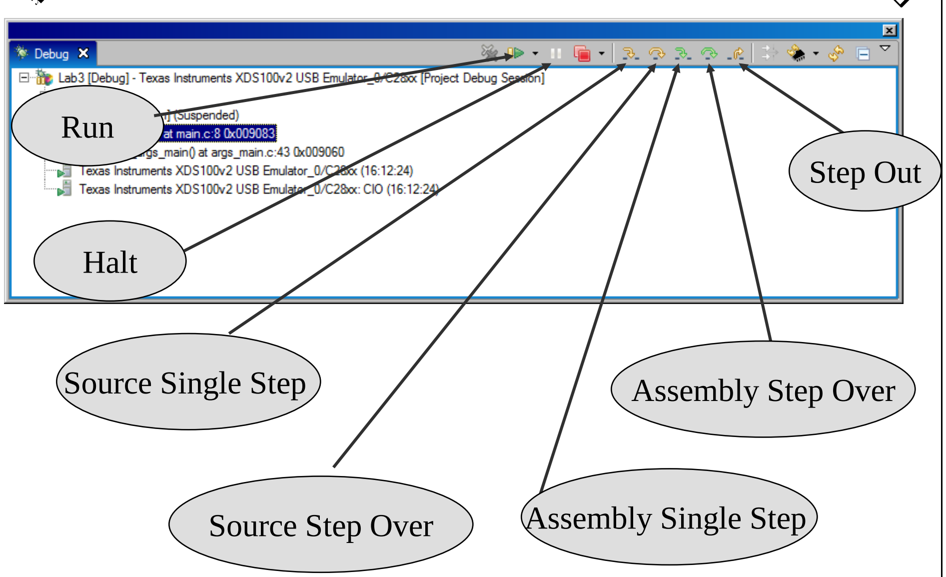
  ➔ Target ➔ Step Into (or use function key “F5”)

- **Watch the current PC (blue arrow) and the values of variables ‘i’ and ‘k’ in Watch Window while you single step the code !**

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

3 - 26

9. Perform Code Stepping



3 - 27

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.

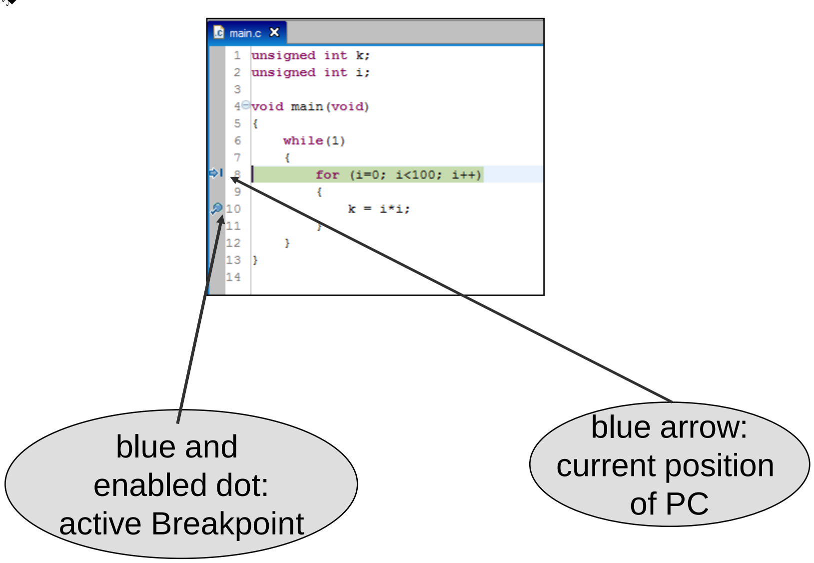
10. Set a Breakpoint

- **Set a Breakpoint:**
  - **Place the Cursor in Lab3.c at line: k = i * i;**
  - **Click right mouse and select “Toggle Breakpoint”**
  - **the left hand side of the line is marked with a blue dot to indicate an active breakpoint.**
  - **A 2nd option is to use a left mouse double click at the grey left hand side of window “Lab3.c” to toggle the breakpoint.**
- **Reset the Program**
  - **→ Target → Reset → Reset CPU**
  - **→ Target → Restart**
- **Perform a real time run**
  - **→ Target → Run (or F8)**

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

3 - 28

10. Set a Breakpoint (cont.)



3 - 29

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!

11. Real – Time - Debug

Reset F2833x:

➔ **Target** ➔ **Reset** ➔ **Reset CPU**

Watchdog – Timer:

- always active after a 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 just disable the watchdog:

➔ **Scripts** ➔ **Watchdog** ➔ **Disable Watchdog**

Start "Real – Time – Debug":

➔ **Scripts** ➔ **Realtime Emulation Control** ➔ **Run_Realtime_with_Restart**

3 - 30

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 script command to disable the watchdog. We would never do that in a real project!

To use “Real - Time - Debug” perform:

➔ Target ➔ Reset ➔ Reset CPU

➔ Scripts ➔ Watchdog ➔ Disable Watchdog

➔ Scripts ➔ Real time Emulation Control ➔ Run_Realtime_with_Restart

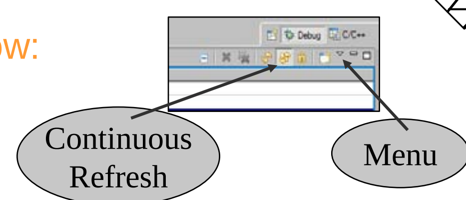
Now the code is running in real-time. The new feature is that we can interact with the device, while the code is running. To practice this:

- In the upper right-hand corner of the watch window, click on the white down-arrow and select “Customize Continuous Refresh Interval”. Change the “Continuous Refresh Interval” to 1 second instead of the default 5 seconds.
- In the upper right-hand corner of the watch window, click on the yellow arrows rotating in a circle over a pause sign to enable continuous refresh mode for the watch window.

The content of the watch window is now 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. However, the USB-JTAG emulator is too slow to update the watch window in the same frequency as our F2833x executes the for-loop. That is why we do not see each increment of variables ‘i’ and ‘k’.

11. Real – Time – Debug (cont.)

In the Watch – Window:



- Enable Continuous Refresh
- In the “menu” open “Customize Continuous Refresh Interval” and change the “Continuous Refresh Interval” to 1 second.
- 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 the real time - Debug:

➔ GEL ➔ Realtime Emulation Control ➔ Full_Halt

3 - 31

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

➔ Scripts ➔ Real time Emulation Control ➔ Full_Halt

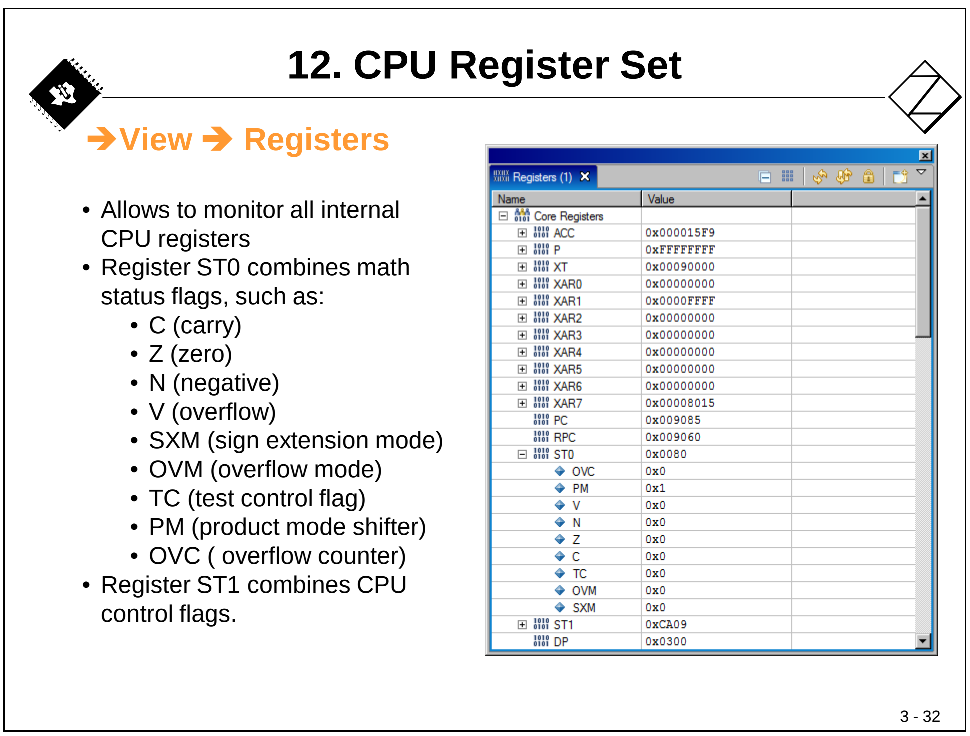
Note: the test mode **"Run_Realtime_with_Reset"** will first perform a reset followed by a direct start of the device from its reset address. The device will follow its hardware boot sequence (see Chapter 15) to begin the code execution. Since the Peripheral Explorer Board sets the coding pins to "Branch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



When you expand the plus signs, for example for register ST0, you can inspect details of a particular register 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 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.

Watch Memory Contents

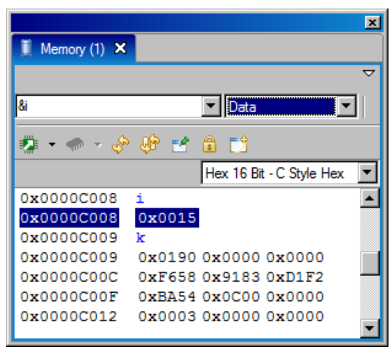
Let us open another control window, the "Memory Window". This window allows us to inspect any physical memory locations of the device, including RAM, FLASH, OTP and Boot - ROM. Since the core of this device is a Digital Signal Processor, we have always to remember that there are two memory spaces, code 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.

13. Memory Window

➔View ➔ Memory

- **make sure to inspect data space (top right hand side selection box)**
- **enter address &i in the top left corner box:**

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



3 - 33

The right-hand side selection box allows us to specify the display mode of the 16-bit memory location in different form, such as:

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

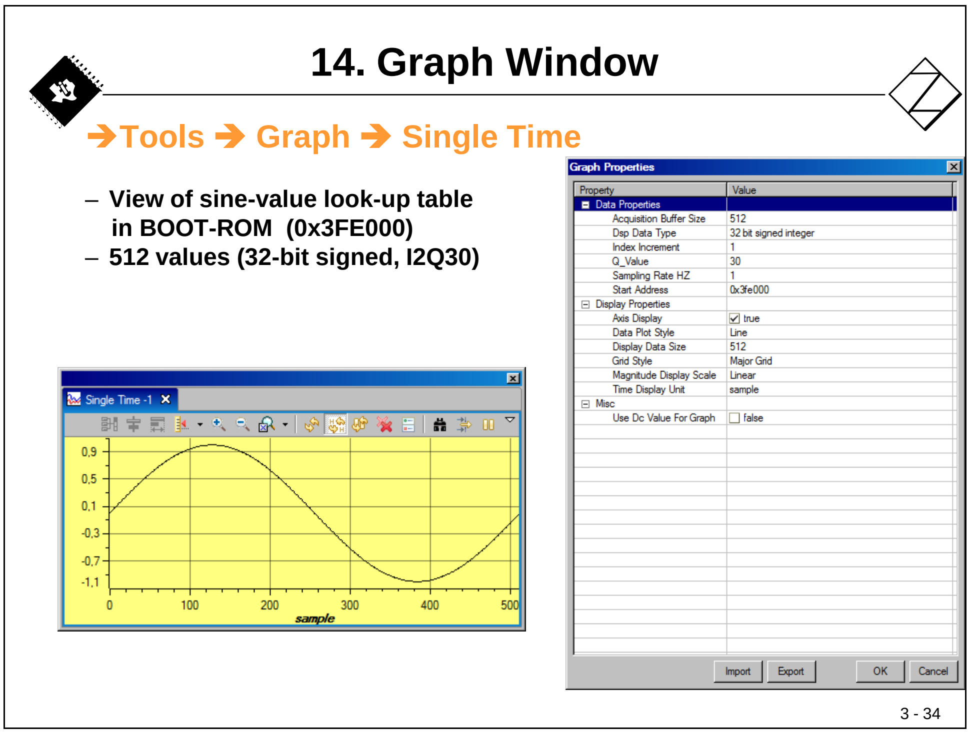
The number of memory windows is not limited, you can open as many as you like!

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.

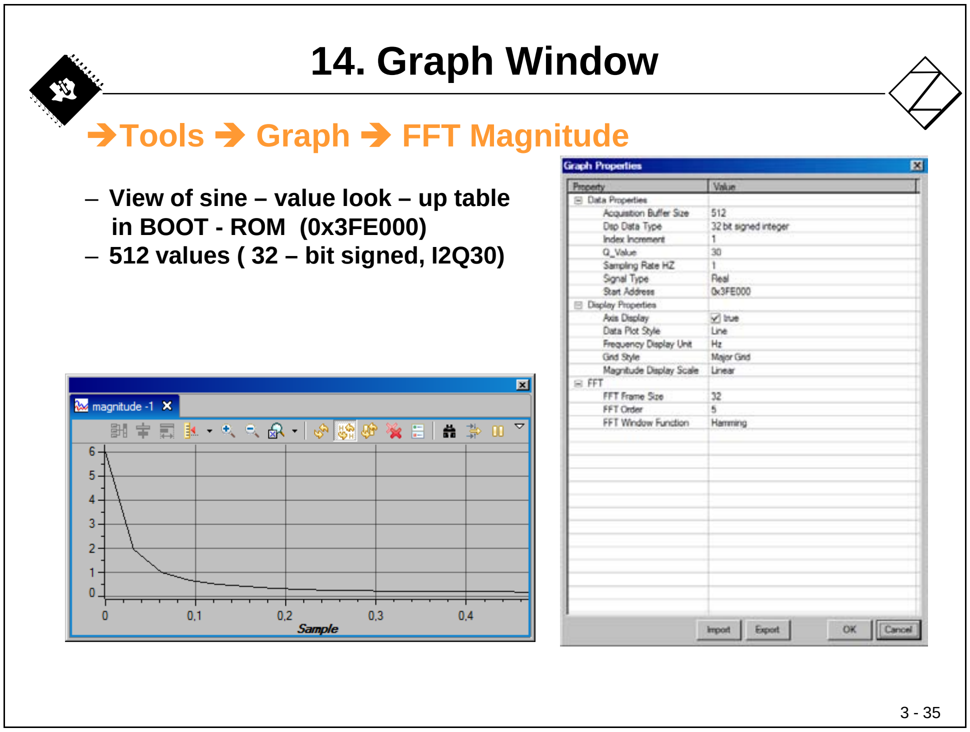
Let us inspect this graph feature. The BOOT-ROM contains a sine - value lookup table of 512 samples for a unit circle of 360 degree. The data format is 32-bit signed integers in fractional I2Q30 - format. The start address of this table is 0x3FE000.

Open a graph window and enter the properties from the left hand side of Slide 3 - 34:



Optionally, you can open a second window to show the fast FOURIER transform (FFT) of the sinusoidal lookup table in Boot – ROM.

Open a graph window and enter the properties from the left hand side of Slide 3 - 35:

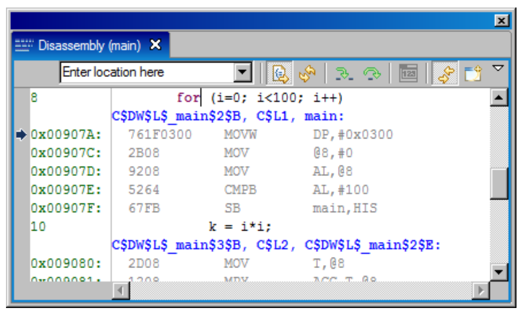


Mixed Mode C and Assembly

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.

15. Mixed Mode Visualization

- **To view both the C – source - code and the resulting Assembler – Code:**
  - **click right mouse inside window “Disassembly” and enable “Show Source”**
  - **Both the C-source code (bold) and the Assembler Instruction Code generated by the C - Compiler (grey) are displayed**



3 - 36

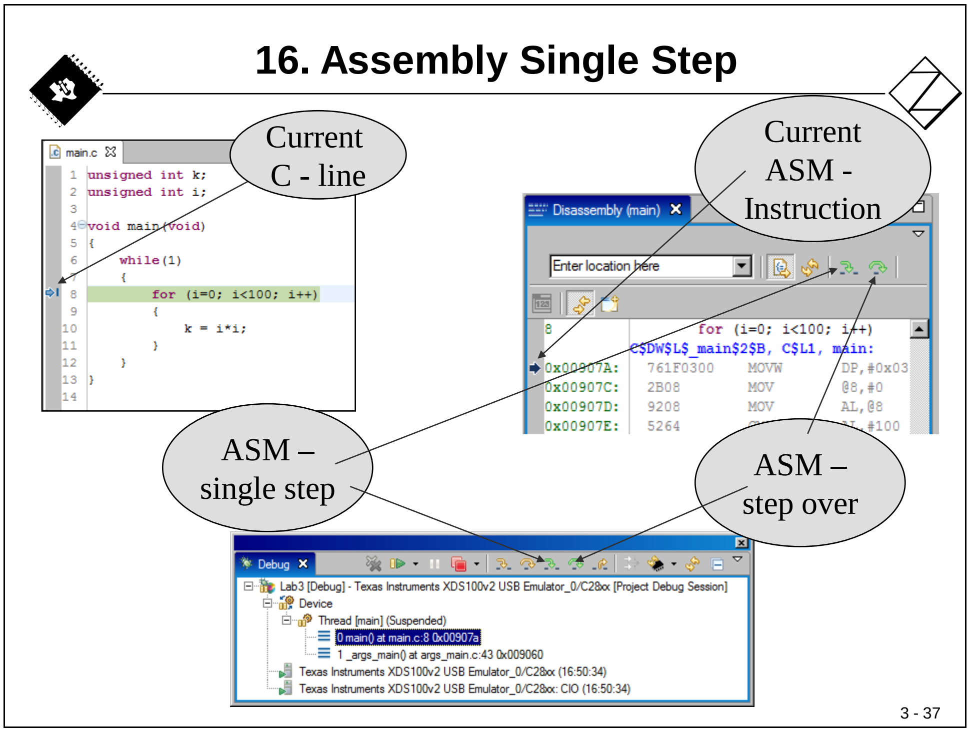
Although this test method is not always required, especially not at the beginning of a tutorial, it allows us to benchmark the efficiency of the C compiler.

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 arrows in the “Debug” and “Disassembly” windows):

- Assembly Single Step
- Assembly Step Over



If you use “Assembly Single Step”, the code is executed machine code line by machine code line. The dark blue arrow in the “Disassembly” window marks the next following assembly line. The light blue arrow in the C-code window (“main.c”) remains at the corresponding C - line, as long as we deal with the assembly results of that line.

At this point it is not important to understand what happens in this assembly code snippet. We will deal later with assembly coding and optimization. However, it is never a fault to question your teacher!

For Assembly Language freaks only:

Here is an explanation of the first C-line: ***for(i=0; i<100; i++)***

MOVW DP ,#0x300	*; sets the direct address pointer DP to address 0xC000*
	; 0x300 left shifted 6 times gives 0xC000
MOV @8,#0	*; loads constant 0 into address 0xC008 (which is ‘i’)*
MOV AL,@8	*; read value from address 0xC0008 into register AL*
CMPB AL,#100	*; compares AL against constant 100*
SB main, HIS	*; short branch (SB) to the beginning of main, if AL was “HIgher*
	; or Same (HIS)”

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.

17. GEL - “General Extension Language”

- GEL = high level language, similar to C
- Used to extend Code Composer Studio’s features
- to create GEL functions use the GEL syntax
- 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 .
- Startup GEL files defined in the target configuration file will be automatically loaded when a debug session is started
- Additional GEL files can be loaded in the CCS from the 'GEL Files' dialog (via 'Tools->GEL Files' menu)

3 - 38

By default, startup GEL – files defined in the target configuration file are automatically loaded when a debug session is started.

To open and inspect the default GEL - file, select:

➔ Tools ➔ GEL Files

Right click at file “F28335.gel” and select “Open”. Inspect the file to get a view of the syntax of this GEL language.

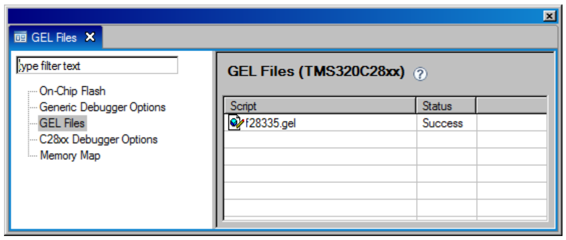
For example search function “OnReset”. This function will be executed every time we perform a command ➔ Target ➔ Reset ➔ Reset CPU.

```
OnReset(int nErrorCode)
{
    C28x_Mode();
    Unlock_CSM();
    ADC_Cal();
}
```

This function itself calls 3 more functions to switch the device into C28x operating mode, to unlock a code security module (CSM) and to calibrate the internal Analogue to Digital Converter (ADC).

17. GEL - “General Extension Language”

➔Tools ➔ GEL files



Right-click in the view and select ‘Open’ to inspect the default GEL-File “F28335.gel”

Example Function: “OnReset”:

```
OnReset(int nErrorCode)
{
    C28x_Mode();
    Unlock_CSM();
    ADC_Cal();

}
```

3 - 39

Lab 3: beginner's project

The following procedure is a summary of the steps, which we discussed in the previous part of this chapter. The following procedure will help you to build your first project for the F2833x device under Code Composer Studio Version 4.1.

Objective

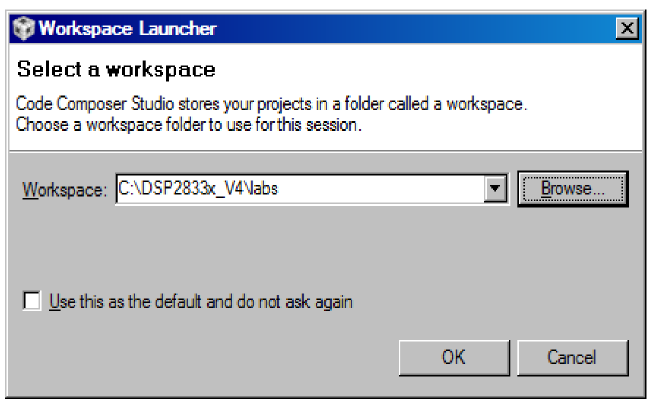
The objective of this lab is to practice and verify the basics of the Code Composer Studio Integrated Design Environment. The following procedure will summarize all the steps discussed in this chapter.

Procedure

Open Files, Create Project File

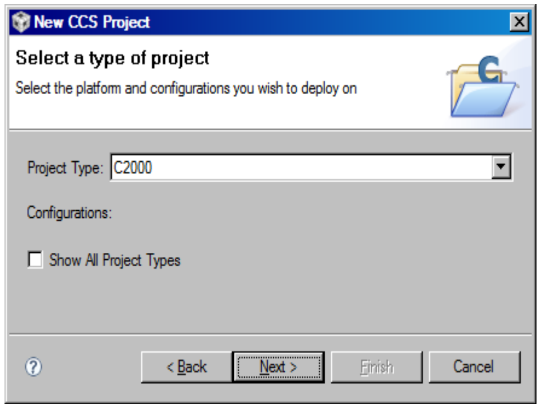
1. To open Code Composer Studio Version 4.1 click the corresponding desktop icon:

2. Next, select a workspace. Ask your teacher about the correct directory of the laboratory PC. The example below uses the folder "C:\DSP2833x_V4\labs"

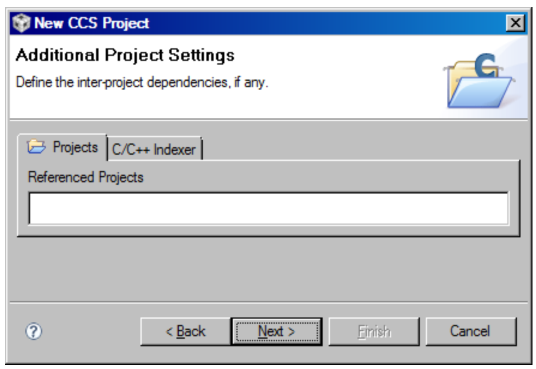


3. Create a new project "**Lab3**" **(➔ File ➔ New ➔ CCS Project):**

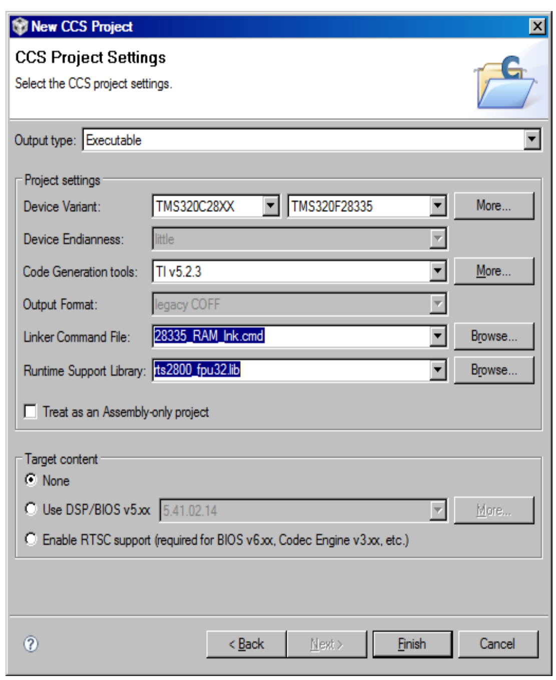
In the following window, select "C2000" for the project type.



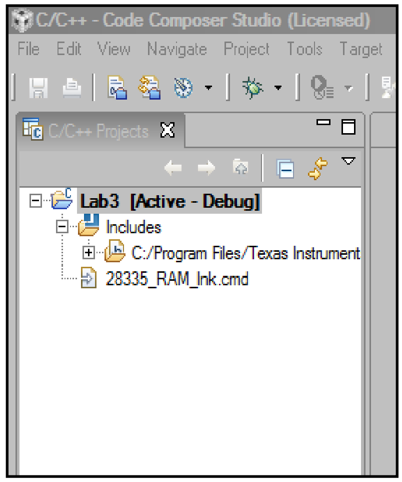
In the next window, do not add any additional settings:



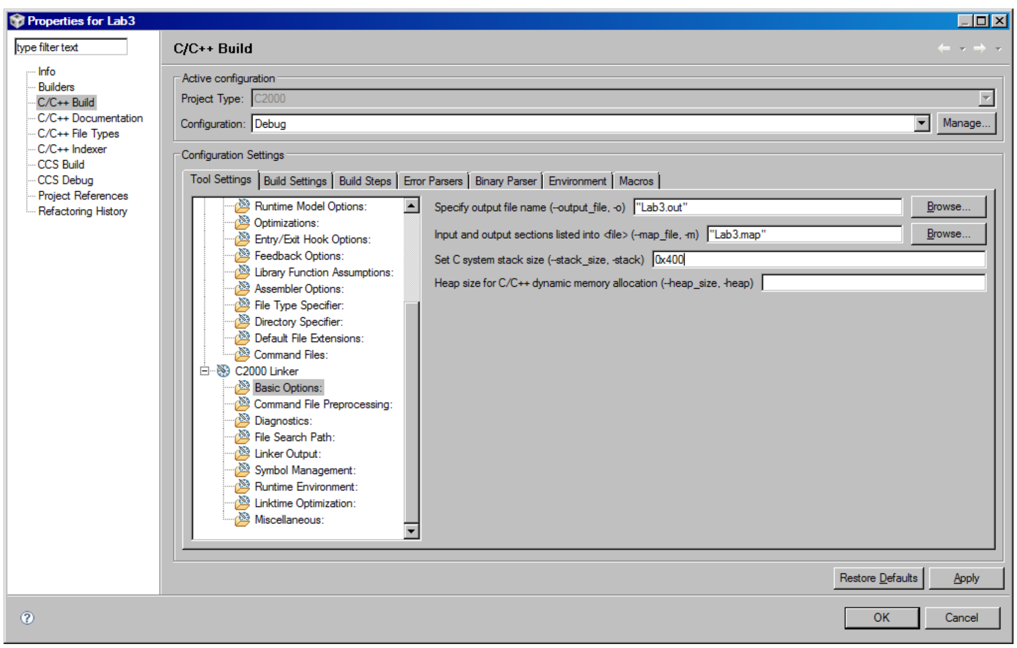
Fill in the project properties according to the following picture:



4. Define the size of the C system stack. In the project window, right click at "Lab3" and Select properties:



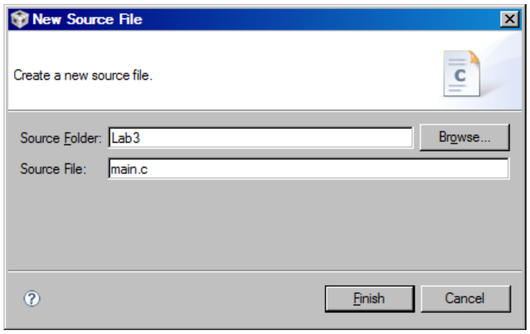
In category "C/C++ Build", "C2000 Linker", "Basic Options" set the C stack size to 0x400:



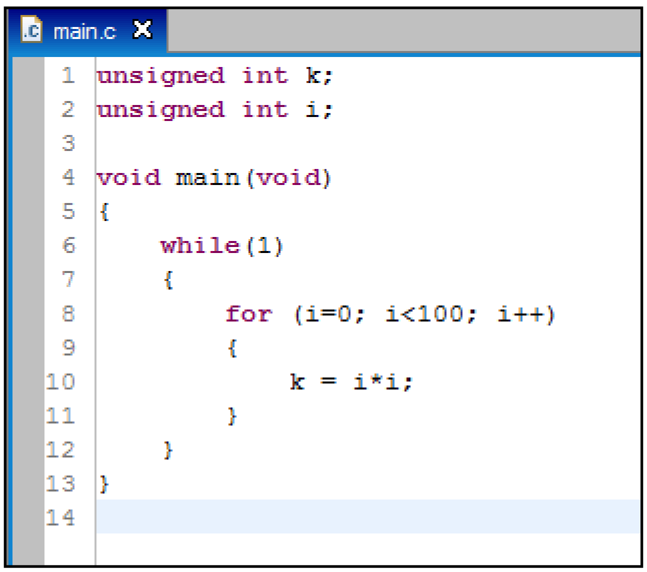
Note: The stack memory is used by the compiler to store local variables, parameters and the processors context in case of hardware interrupts. It is our task to specify a certain amount of memory for this purpose and 0x400 is sufficient for this lab.

Write C - code

5. Write a new source code file by clicking: File → New → Source File. A new window will open. Enter the file name "main.c":



In the file "main.c" enter the following few lines:



Save this file by clicking File → Save as and type in: **Lab3.c**

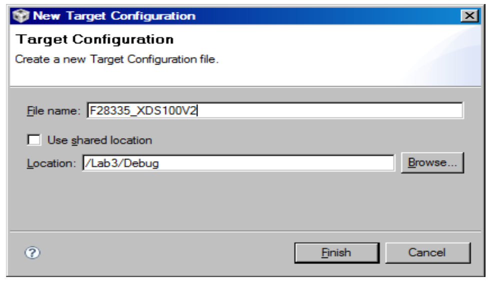
Build and Load

6. Click the "Rebuild Active Project" button or perform: Project → Rebuild Active Project 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.

Create a Target Configuration

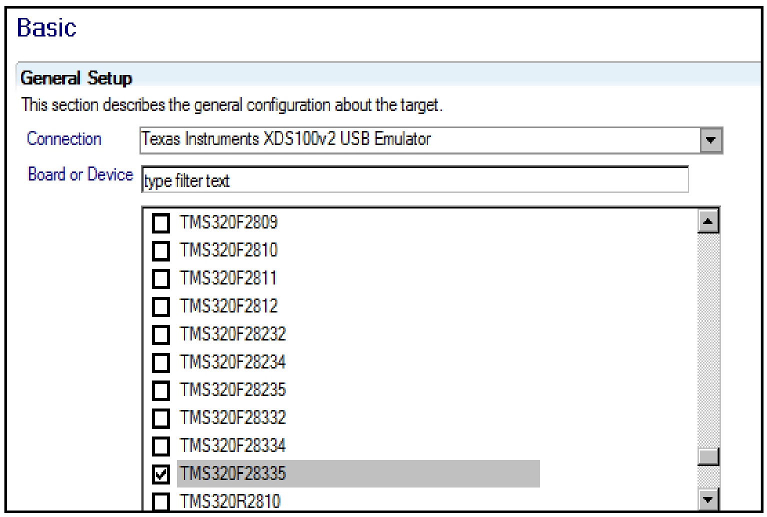
7. Before we can download the machine code into the F28335, we have to define the "target configuration".

   ➔ Target ➔ New Target Configuration



Type a name for the target configuration file in box "File name". You can use any name here but it makes sense to indicate the JTAG-emulation interface, which we will use for the download later. In case of the Peripheral Explorer Board we use the XDS100V2, so let us call the file "F28335_XDS100V2. The suffix ".ccxml" will be added automatically.

In the window that appears next, select the emulator "Texas Instruments XDS100v2 USB Emulator" via the "Connection" pull-down list and select the "TMS320F28335" device checkbox.



Load Code into Target

8. Load the machine code into the device. Click:

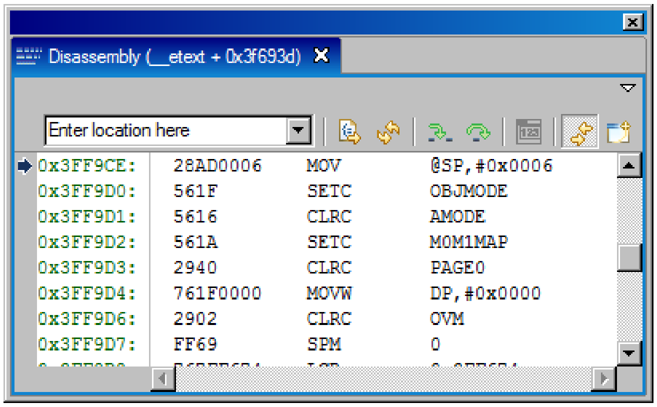
   ➔ Target ➔ Debug Active Project

   Or use the "Debug" icon: 

   A blue arrow should now point to the "for" – line in code file "main.c". This is an indication that the machine code has been downloaded properly into the F28335.

Test

9. Reset the DSP by clicking on ➔ Target ➔ Reset ➔ Reset CPU



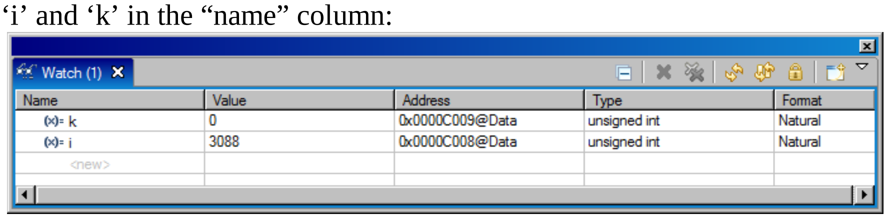
The blue arrow shows the current position of the Program Counter (PC), which is now loaded with the hardware - reset address 0x3FF9CE in Boot-ROM.

10. Run the program until the first line of your C-code by clicking: Target ➔ Restart.

This command is often used directly after a RESET command. Its purpose is to bypass the Boot – code and to load the Program Counter (PC) directly with the "entry point address" for the code. This entry point address can be specified in the project options. For C-language based projects the default address is the environment preparation function "_c_int00" (from library rts2800_fpu.lib"

However, because we have enabled the auto run option to "main()", the restart command will run through "_c_int00" and stop at the beginning of "main()". If this auto run option would have been disabled, we could use ➔ Target ➔ "Go to Main" as a 3rd command.

11. Open the Watch Window to watch your variables. Click: View → Watch. Add the two variables 'i' and 'k' in the "name" column:



Code Step Comands

12. Perform a single-step through your program by clicking: Target → Step Into (or use function Key F5). Repeat F5 and watch your variables.

13. 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 blue dot to mark an active breakpoint. Perform a real- time run by Target → Run (or F8). The program will stop execution when it reaches the active breakpoint. Remove the breakpoint after this step (click right mouse and "Toggle Breakpoint").

Real Time Mode

14. Now we will exercise with the real-time debug mode. Make sure that all breakpoints have been deleted or disabled.

    Next, reset the device: ➔ Target ➔ Reset ➔ Reset CPU

    This reset will set the device in its default state, including the watchdog unit, which is enabled after reset. For this test we will have to disable the watchdog:

    ➔ Scripts ➔Watchdog ➔Disable Watchdog

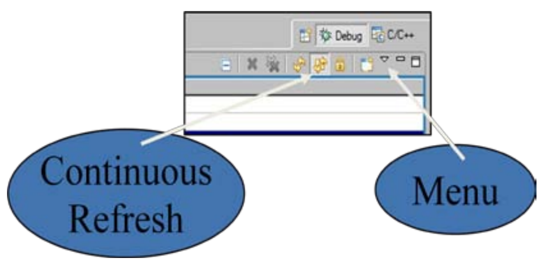
    Now start the real-time debug:

    ➔ Scripts ➔ Real time 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:

    ➔ In the upper right-hand corner of the watch window, click on the white down-arrow and select "Customize Continuous Refresh Interval". Change the "Continuous Refresh Interval" to 1 second instead of the default 5 seconds.

    ➔ In the upper right-hand corner of the watch window, click on the yellow arrows rotating in a circle over a pause sign to enable continuous refresh mode for the watch window.



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. However, the USB-JTAG emulator is too slow to update the watch window at the same frequency as our F2833x executes the for-loop. That is why we do not see each increment of 'i' and 'k'.

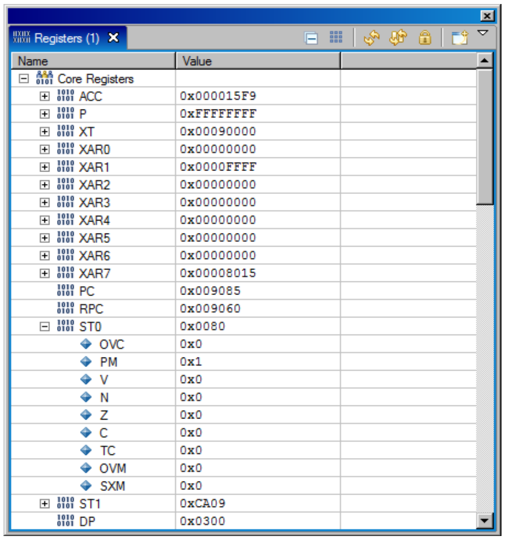
When you are done, stop the real - time mode by:

➔ Scripts ➔ Real time 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.

15. Inspect the internal device registers:

➔ View ➔ Registers



When you expand the plus signs, for example for register ST0, you can inspect details of the particular register 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 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.

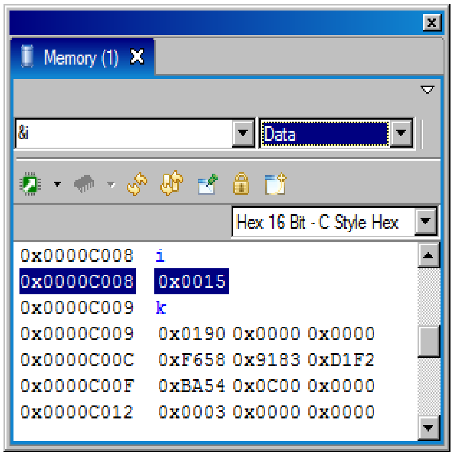
Watch Memory Contents

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

➔ View ➔ Memory

Enter the address for variable k ("&k") in the address box (top left corner box).

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 top right corner box of this window.



The right center 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.

Graphical Views

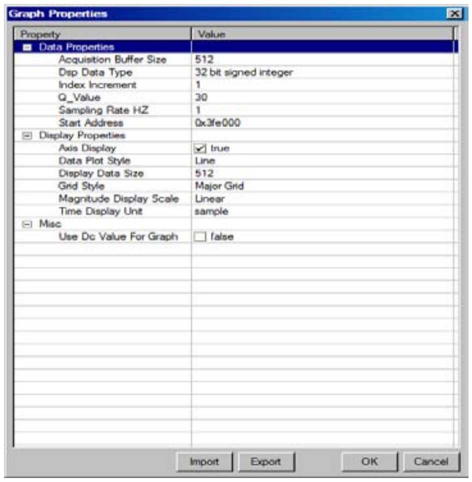
Time Domain Graph

17. 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.

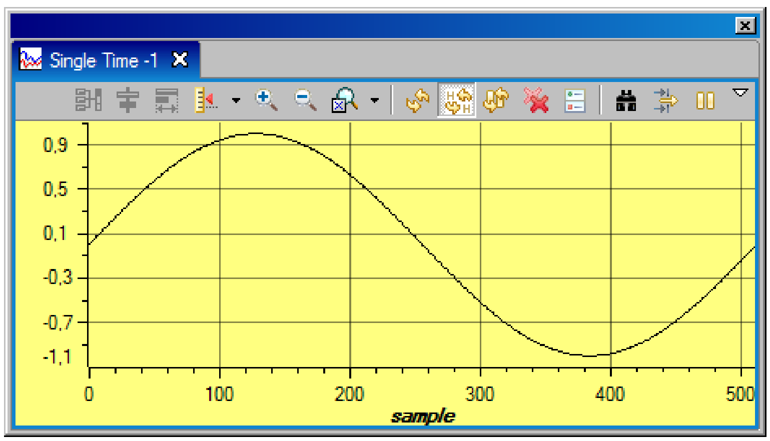
    Let us inspect this graph feature:

    ➔ Tools ➔ Graph ➔ Single Time

    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:



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

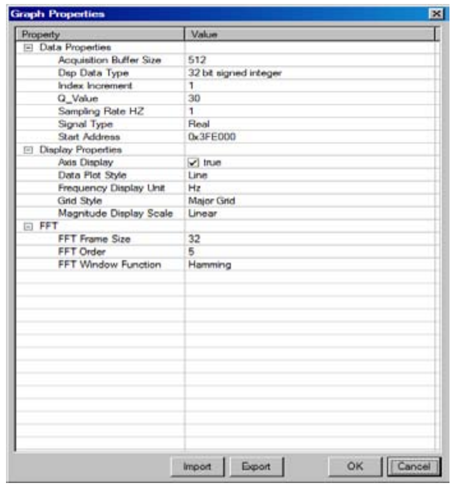


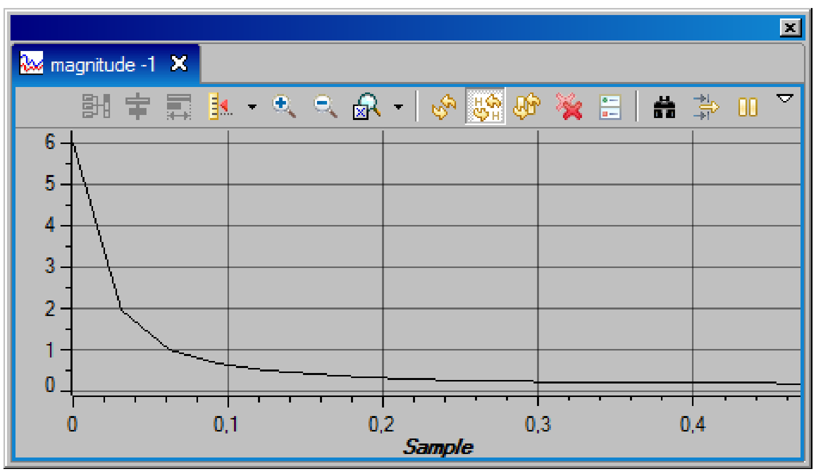
Frequency Domain Graph

Now open a second graph:

➔ Tools ➔ Graph ➔ FFT Magnitude

Enter the following properties:





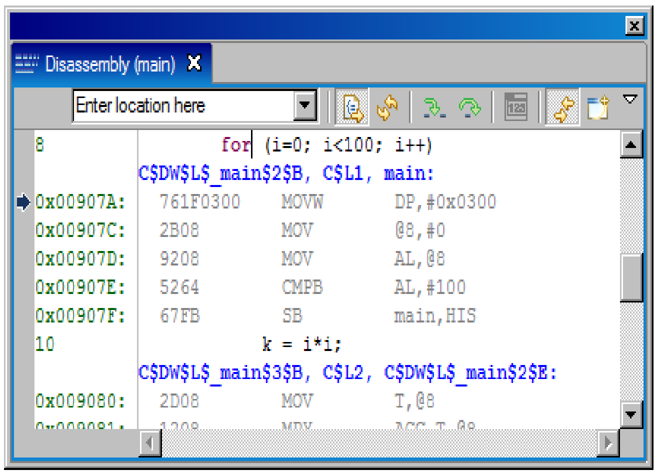
Close the graphical windows, when you are done.

Mixed Mode C and Assembly Language

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.

18. To visualize C and Assembler:

Right mouse click into the "Disassembly Window" and enable "Show Source"



Optionally, use the green icon "Assembly Single Step" on the top window line.

If you use "Assembly Single Step", the code is executed machine code line by machine code line. The dark blue arrow marks the next following assembly line. The light blue 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).

End of Exercise Lab3