

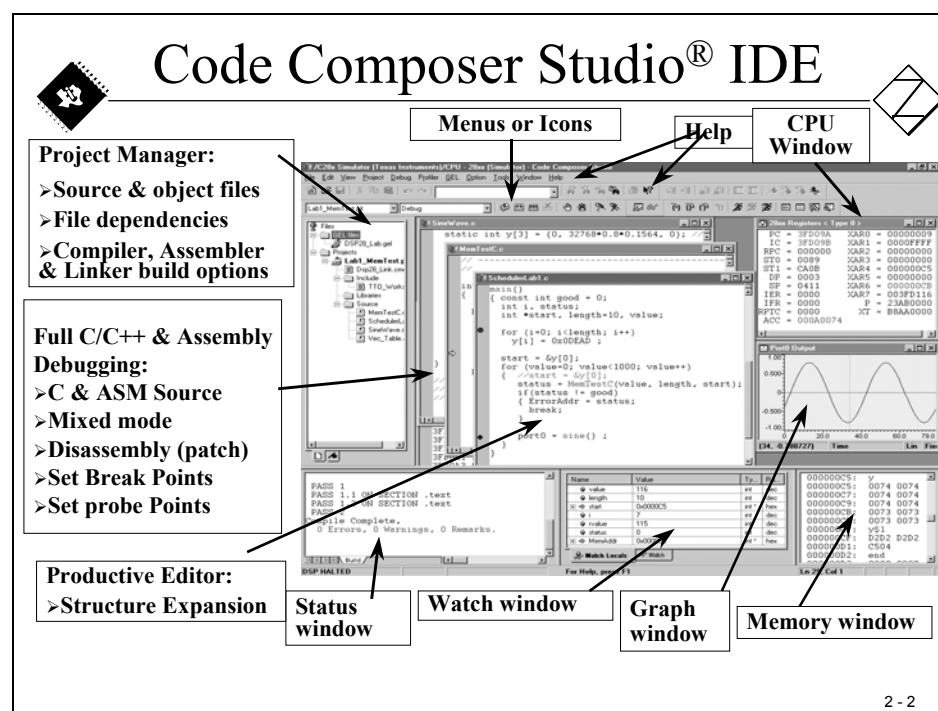
# Program Development Tools

## Introduction

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

## Code Composer Studio IDE

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

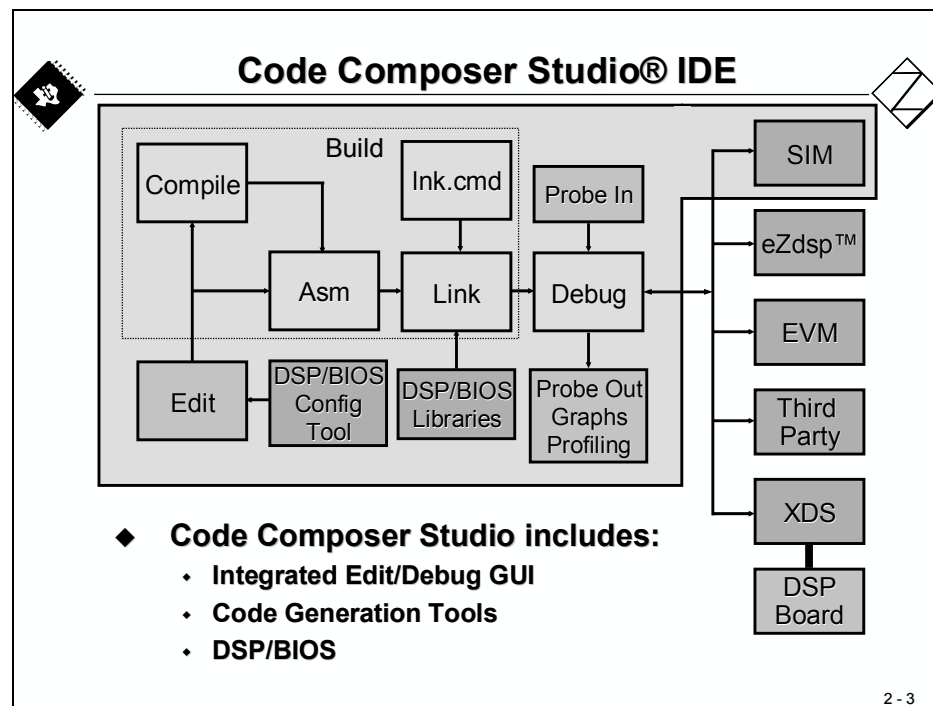


## Module Topics

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

## The Software Flow

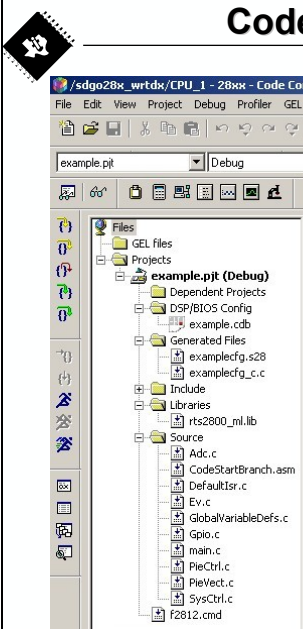
The following slide illustrates the software design flow within Code Composer Studio. The basic steps are: edit, compile and link, which are combined into “build”, then debug. If you are familiar with other Integrated Design Environments for the PC such as Microsoft’s Visual Studio, you will easily recognize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



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

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

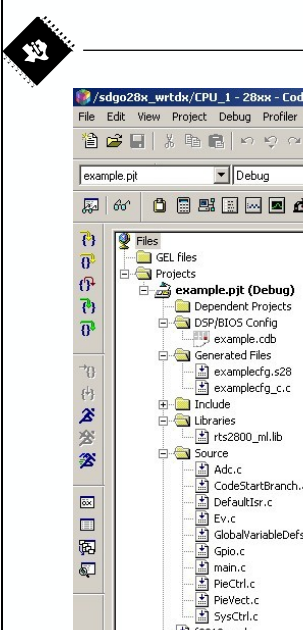
## Code Composer Studio - Basics



The screenshot shows the Code Composer Studio IDE. The title bar reads '/sdgo28x\_wrtx/CPU\_1 - 28xx - Code Composer Studio'. The menu bar includes File, Edit, View, Project, Debug, Profiler, GEL, Option, Tools, DSP/BIOS, Window, and Help. The toolbar contains various icons for file operations, project management, and debugging. The left pane shows a tree view with 'example.pjt (Debug)' selected, containing sub-items like 'Dependent Projects', 'DSP/BIOS Config', 'example.cdb', 'Generated Files', 'Include', 'Libraries', 'Source', and 'F2812.cmd'. The right pane is empty.

- ◆ **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 3<sup>rd</sup> party plug-ins**

2 - 4



The screenshot shows the Code Composer Studio IDE. The title bar reads '/sdgo28x\_wrtx/CPU\_1 - 28xx - Code Con'. The menu bar includes File, Edit, View, Project, Debug, Profiler, GEL. The toolbar contains various icons for file operations, project management, and debugging. The left pane shows a tree view with 'example.pjt (Debug)' selected, containing sub-items like 'Dependent Projects', 'DSP/BIOS Config', 'example.cdb', 'Generated Files', 'Include', 'Libraries', 'Source', and 'F2812.cmd'. The right pane is empty.

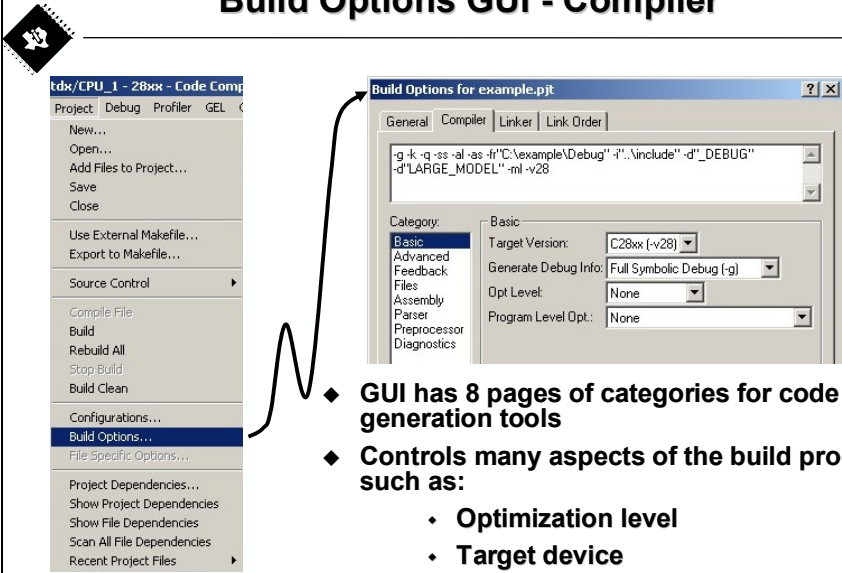
### The CCS Project

**Project (.pjt) files contain:**

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

2 - 5

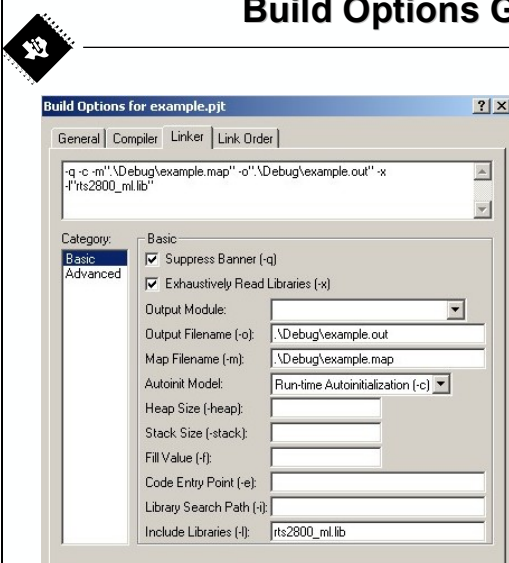
## Build Options GUI - Compiler



- ◆ GUI has 8 pages of categories for code generation tools
- ◆ Controls many aspects of the build process, such as:
  - Optimization level
  - Target device
  - Compiler/assembly/link options

2 - 6

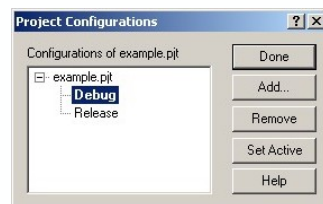
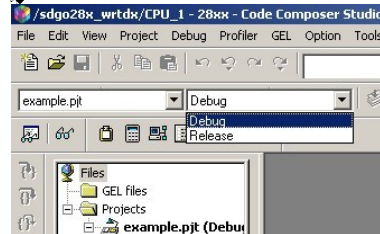
## Build Options GUI - Linker



- ◆ GUI has 2 categories for linking
- ◆ Specifies various link options
- ◆ “.\\Debug\\” indicates on subfolder level below project (.pjt) location

2 - 7

## Default Build Configurations

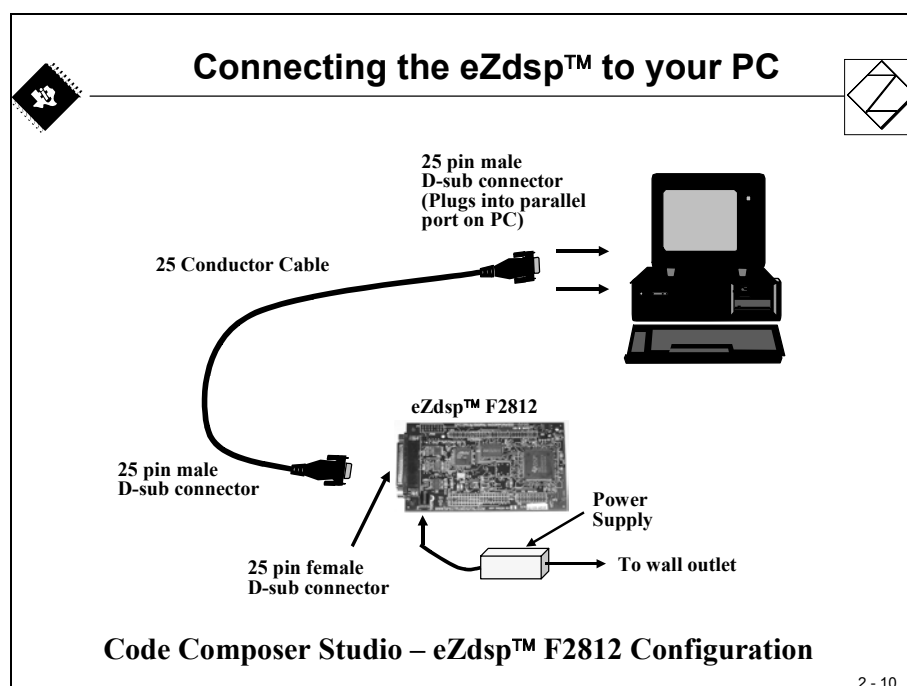
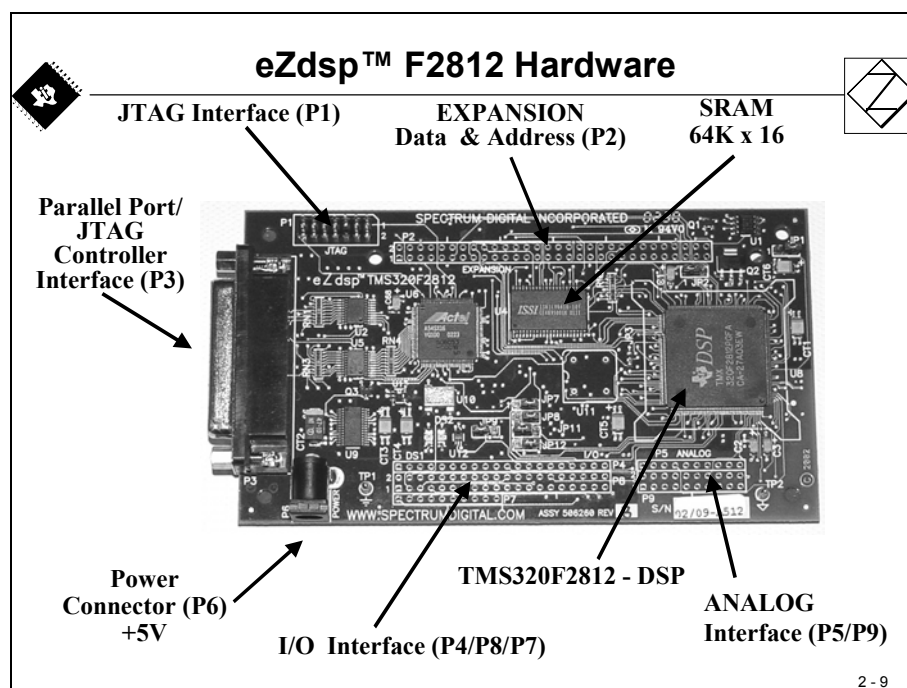


- ◆ For new projects, CCS automatically creates two build configurations:
  - Debug (unoptimized)
  - Release (optimized)
- ◆ Use the drop-down menu to quickly select the build configuration
- ◆ Add/Remove your own custom build configurations using Project Configurations
- ◆ Edit a configuration:
  1.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

2 - 8

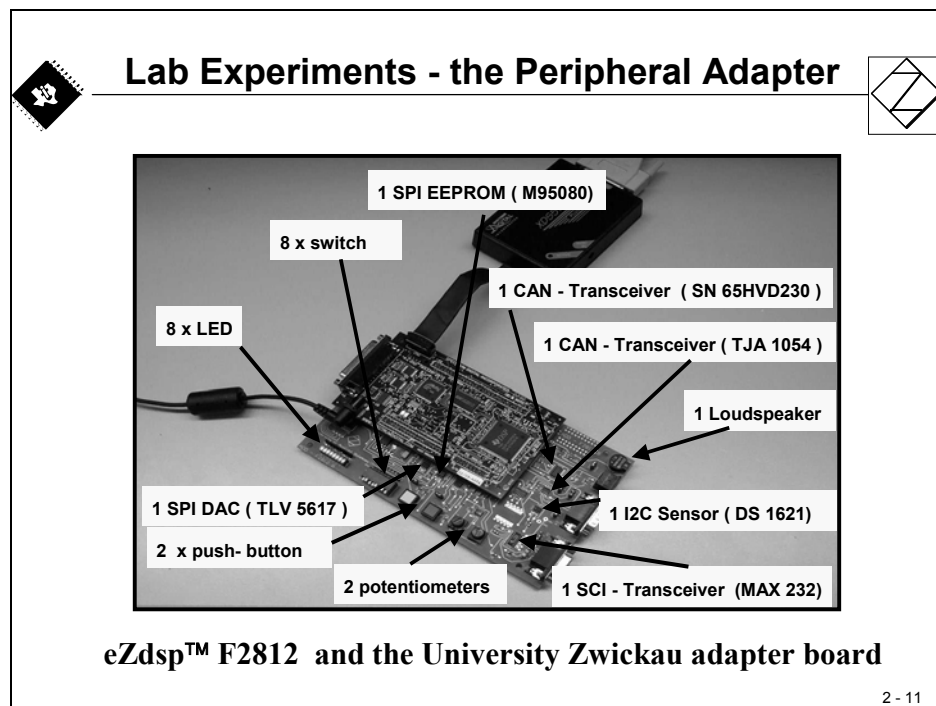
## Lab Hardware Setup

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



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

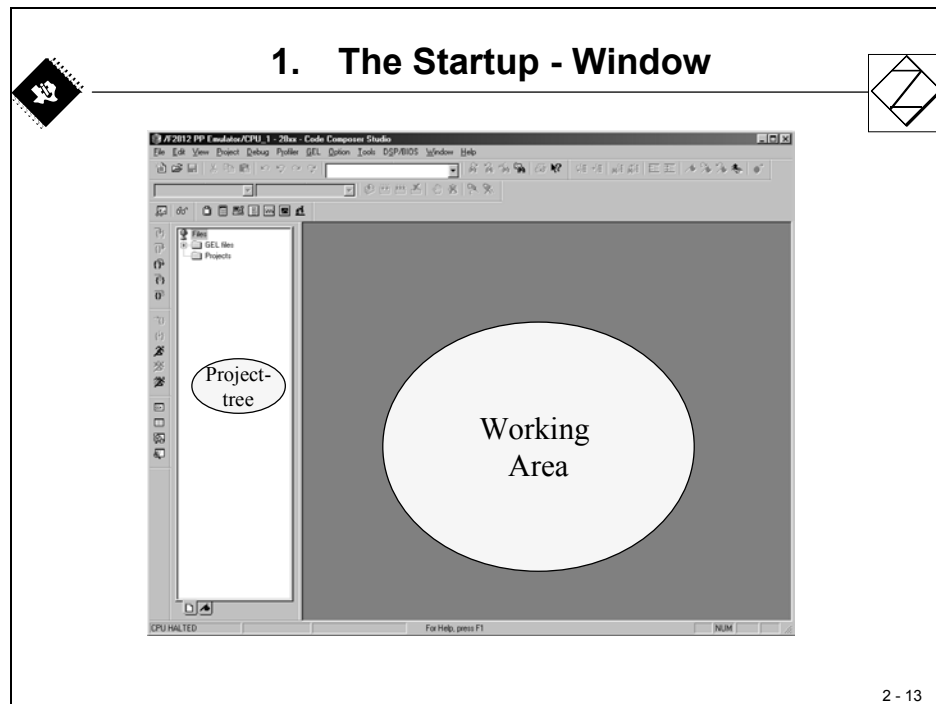
- 8 LED's for digital output (GPIO B7...B0)
- 8 switches (GPIO B15...B8) and 2 push buttons (GPIO D1, D6) for digital input
- 2 potentiometers (ADCINA0, ADCINB0) for analog input
- 1 loudspeaker for analogue output via PWM - Digisound F/PWC04A
- 1 dual SPI Digital to Analogue Converter (DAC) - Texas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





## Code Composer Studio – Step by Step

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




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


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

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

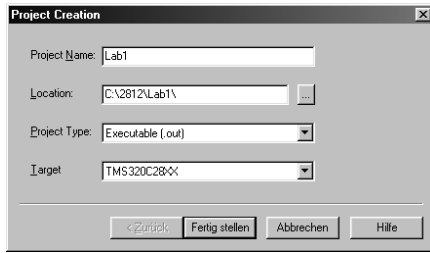
## Create a project



### 2. Create a F28x - project




- **Project → New**  
give your project a name : “Lab1”, select a target and a suitable location of your hard disk:




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

2 - 14

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



### 2. Create a F28x - project (cont.)



**Write your C-Source Code :**  
→File →New →Source File

```

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

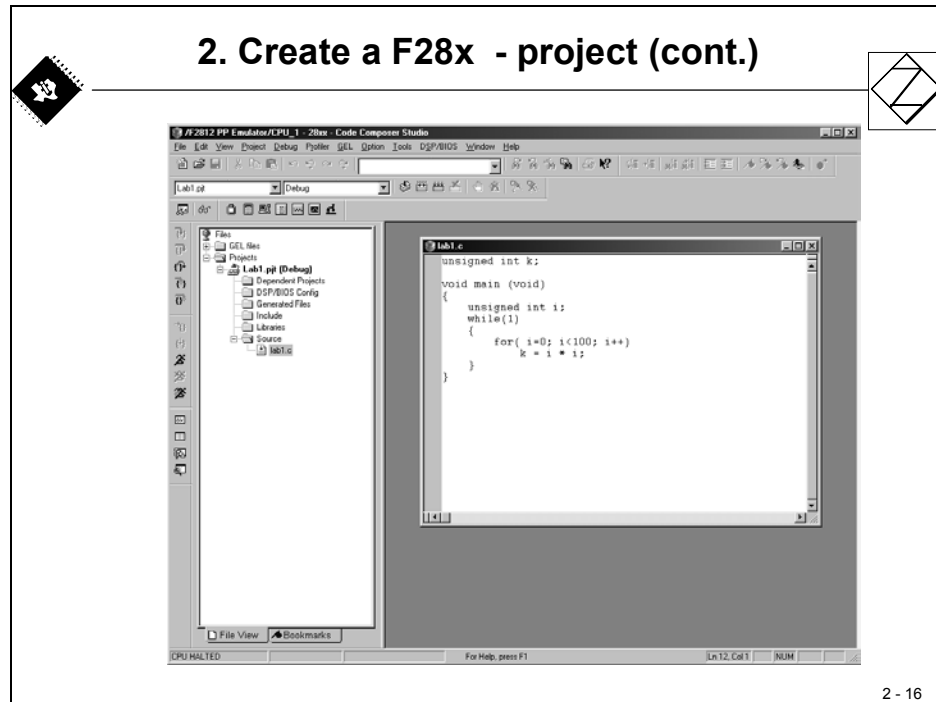
```

→File →Save as : “lab1.c”  
( use the same path as for the project, e.g. C:\C28x\Lab1)

2 - 15

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

Your screen should now look like this:



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

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

## Setup Build Options

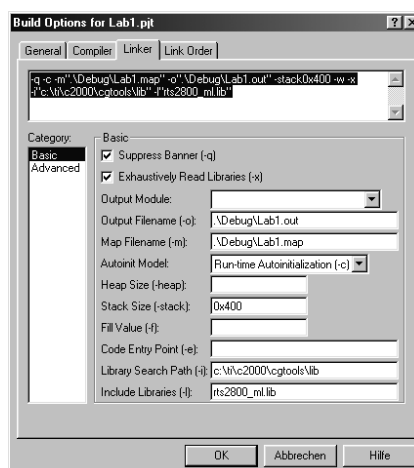
### 2. Create a F28x - project (cont.)

- ◆ **Add your code file to the project :**
  - ➔ Project ➔ Add files to project : “lab1.c”
- ◆ **Compile your source code :**
  - ➔ Project ➔ Compile File
    - active window will be compiled
    - in the event of syntax errors : modify your source code as needed
- ◆ **Add the C-runtime-library to your project :**
  - ➔ Project ➔ Build Options ➔ Linker ➔ Library Search Path :  
c:\ti\c2000\cgttools\lib
  - ➔ ➔ Project ➔ Build Options ➔ Linker ➔ Include Libraries :  
rts2800\_ml.lib
- ◆ **Add the stack - size of 0x400**
  - ➔ Project ➔ Build Options ➔ Linker ➔ Stack Size : 0x400

2 - 17

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

### 2. Create a F28x - project (cont.)



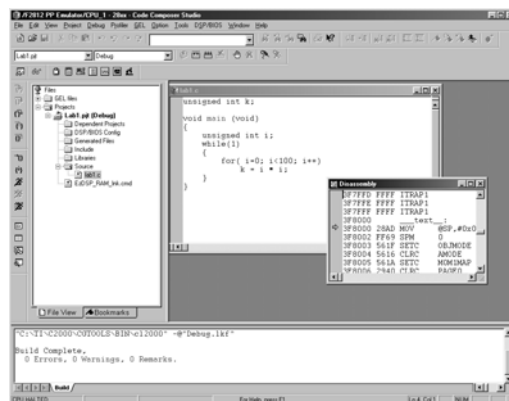
Close the build-window by 'OK'

2 - 18

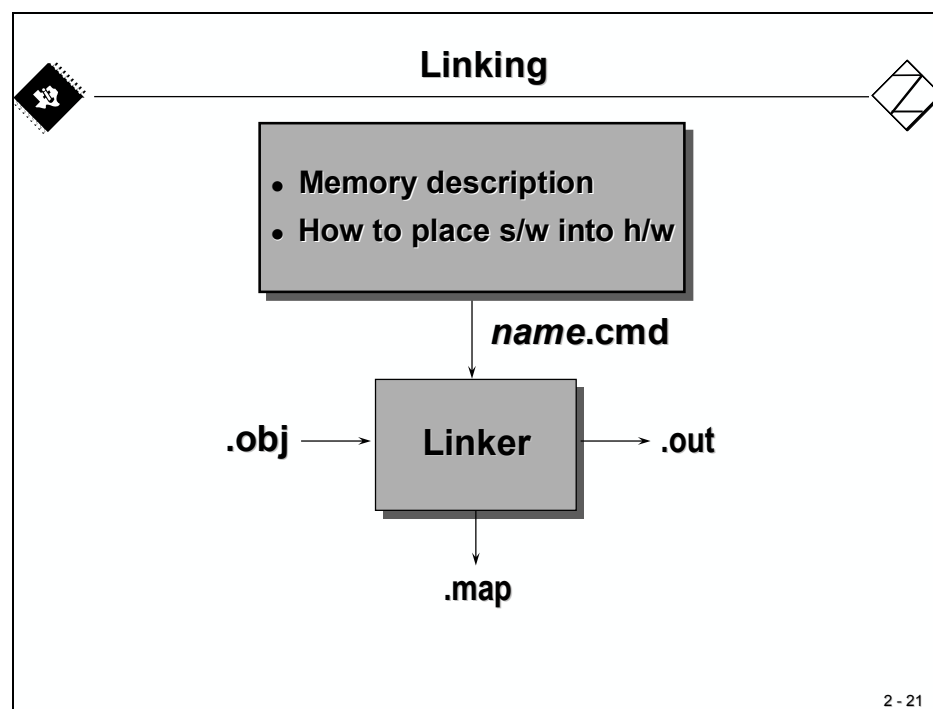
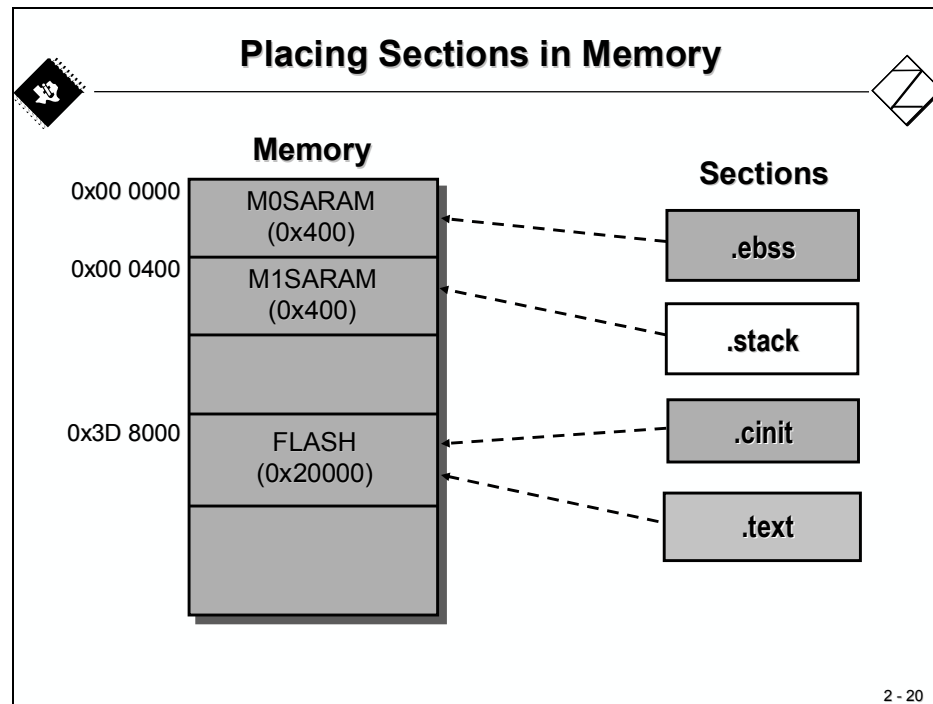
## Linker Command File

Now we have to control the “Linker”. The “Linker” puts together the various building blocks we need for a system. This is done with the help of a so-called “Linker Command File”. Essentially, this file is used to connect physical parts of the DSP’s memory with logical sections created by our software. We will discuss this linker procedure later in detail. For now, we will use a predefined Linker Command File “F2812\_EzDSP\_RAM\_lnk.cmd”. This file has been designed by Texas Instruments and is part of the Code Composer Studio Software package.

- 



2 - 19



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

## Linker Command File

```

MEMORY
{
    PAGE 0:          /* Program Space */
    FLASH:           org = 0x3D8000, len = 0x20000
    PAGE 1:          /* Data Space */
    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
}

```

2 - 22

## Download code into DSP

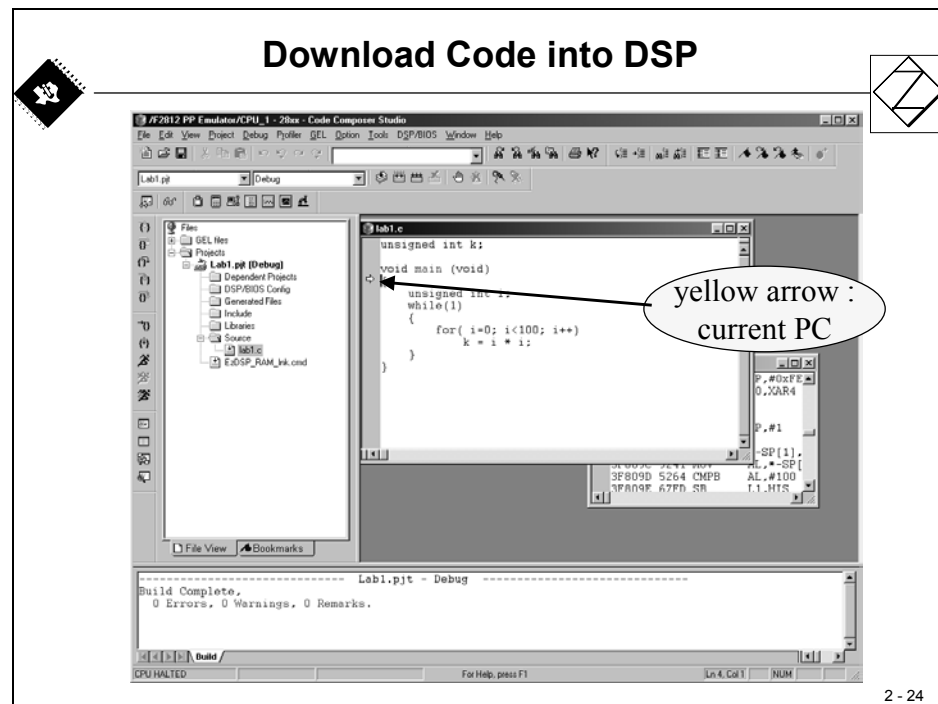
Now it is time to download the code into the F2812. We have two options: manual or automatic download after a successful build.

## Download Code into DSP

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

2 - 23

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



2 - 24

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

### 3. Debug your code !

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

2 - 25



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

## 4. Watch your variables

### ◆ Open the Watch Window :

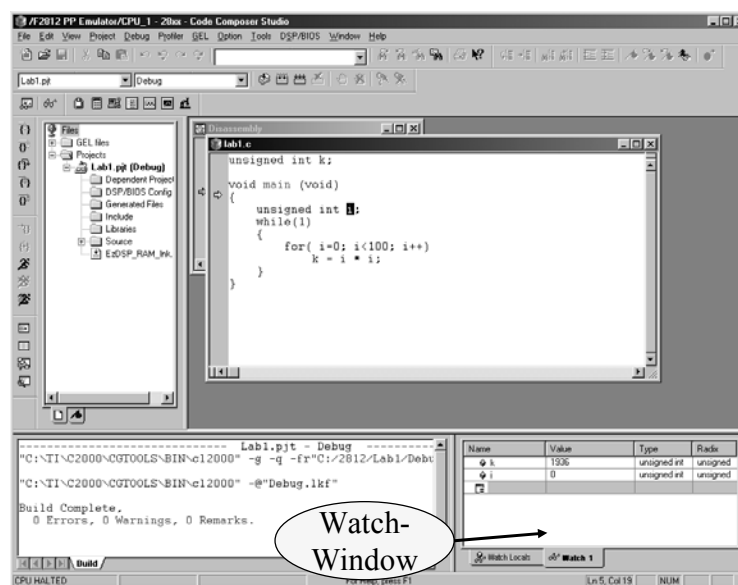
#### ➔ View ➔ Watch Window

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

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


2 - 26

## 4. Watch your variables




2 - 27

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




## 5. Perform a Single Step Debug




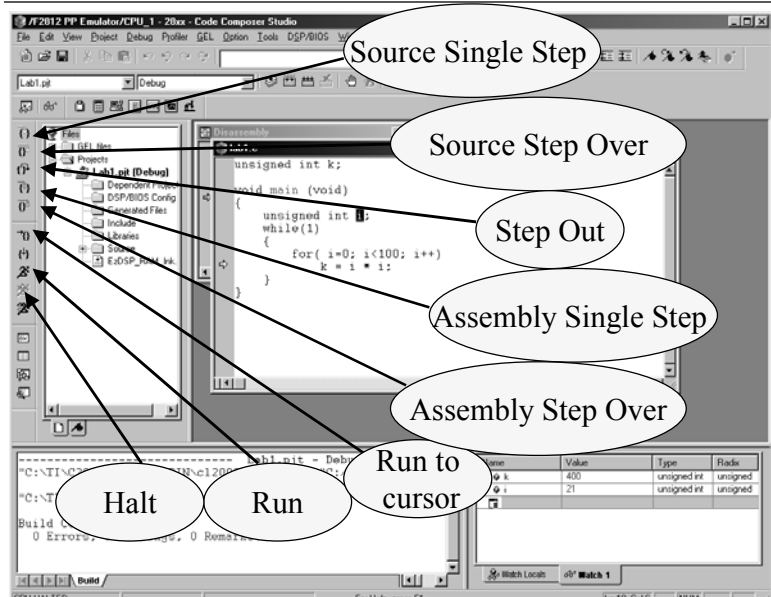
- ◆ **Perform a single step through the program :**
  - ➔ Debug ➔ Step Into ( or F8 )
- ◆ **Watch the current PC ( yellow arrow) and the numerical values of i and k in Watch Window while you single step through the code !**
- ◆ **There are more debug - commands available, see next slide**

2 - 28



## 5. Perform a Single Step Debug





2 - 29

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

## 6. Add a breakpoint

### ◆ Set a breakpoint:

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

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

### ◆ Reset the Program:

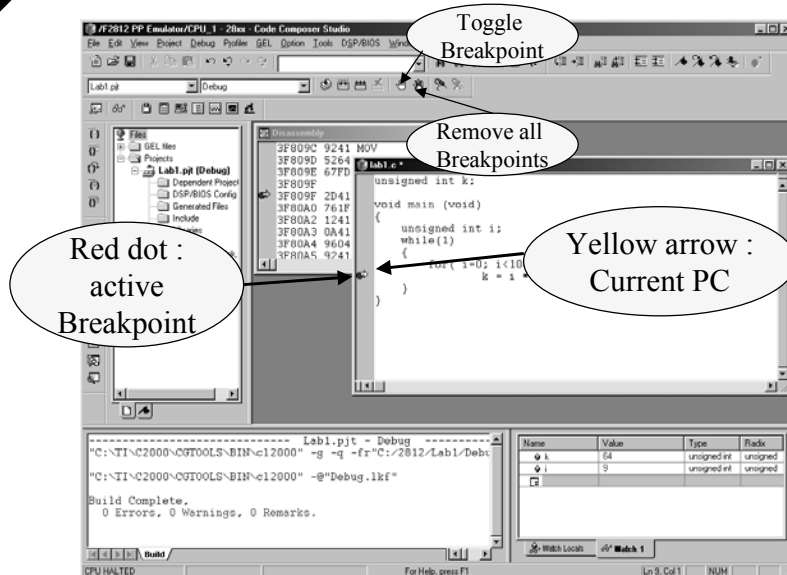
- Debug → Reset CPU
- Debug → Restart

### ◆ Perform a real time run:

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


2 - 30

## 6. Add a breakpoint (cont. )




2 - 31

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



## 7. Set a Probe Point



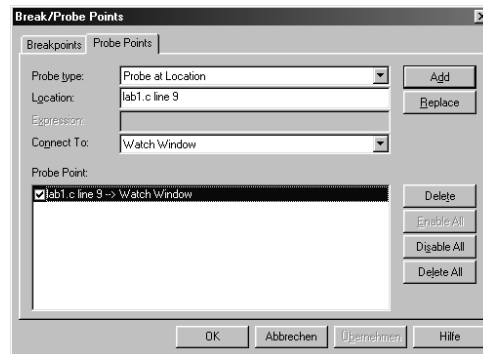
- ◆ **Causes an update of a particular window at a specific point in your program.**  

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

2 - 32

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

## 7. Set a Probe Point (cont.)



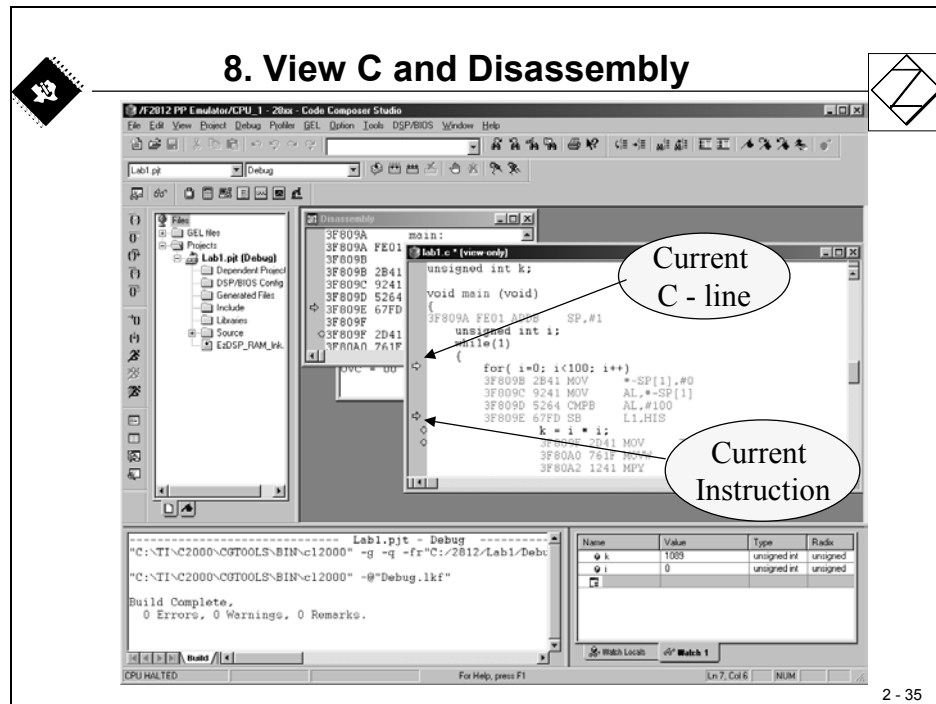
2 - 33

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

## 8. Other View Commands (cont.)

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

2 - 34



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

## 9. GEL - General Extension Language

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

2 - 36

## Lab 1: beginner's project

### Objective

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

### Procedure

#### Open Files, Create Project File

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

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

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

3. Add the Source Code Files: **Lab1.c** and the provided linker command file: **\cmd\F2812\_EzDSP\_RAM\_lnk.cmd** (it is in E:\2812\cmd\) to your project by clicking: Project → Add Files to project
4. Add the C-runtime library to your project by clicking: Project → Build Options → Linker → Library Search Path: 'c:\ti\c2000\cgtools\lib'. Then Add the library by clicking: Project → Build Options → Linker → Include Libraries: '**rts2800\_ml.lib**'

5. Verify that in Project → Build Options → Linker the Autoinit-Field contains: 'Run-time-Autoinitialisation [-c]
6. Set the stack size to 0x400: Project → Build Options → Linker → Stack Size
7. Close the Build Options Menu by clicking OK

## Build and Load

8. Click the "Rebuild All" button or perform: Project → Build and watch the tools run in the build window. Debug as necessary. To open up more space, close any open files or windows that you do not need at this time.
9. Load the output file onto the eZdsp. Click: File → Load Program and choose the output file you generated. Note: The binary output file has the same name as the project with the extension .out. The file is stored under the "Debug" subfolder.

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

## Test

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



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

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

End of Exercise Lab1

This page was intentionally left blank.