

Software Development Tools

The goal of this chapter is to present the main characteristics of the MSP430 Integrated Development Environments (IDEs), both from TI and from third parties. Due to the wide range of hardware platforms available, special attention will be given to Code Composer Essentials and IAR Embedded Workbench IDE. The basic functions and step-by-step project development will be given for each tool. Topics covered will be the structure and management (source files, compiling, assembling and linking operations) of projects developed in both C (mainly) and/or Assembly language.

Topic	Page
2.1 Introduction	2-2
2.2 Code Composer Essentials IDE.....	2-2
2.2.1 Installing CCE.....	2-3
2.2.2 Introduction to CCE IDE	2-7
2.2.3 Lab1: "Hello World" Beginner's project	2-37
2.3 IAR Embedded Workbench IDE.....	2-59
2.3.1 IAR EWB main features	2-59
2.3.2 Lab1: "Hello World" Beginner's project	2-59
2.4 Third party MSP430 IDEs	2-69

2.1 Introduction

During the development of the laboratory exercises, the software development tools will be used to make use of the microcontroller features. The introduction to the MSP430 microcontroller makes use of the TI's Code Composer Essentials (CCE v3) and IAR Systems' Embedded Workbench - Kickstart Version. These Integrated Development Environments (IDEs) allow applications to be written, compiled, assembled, linked, debugged and run on MSP430 hardware.

2.2 Code Composer Essentials IDE

TI recently launched Code Composer Essentials v3. This IDE's latest version (version 3) supports all available MSP430 devices. It is available as a:

- Free upgrade for existing v2 users;
- Professional version (\$499), the main features being:
 - Unlimited code size;
 - Can be ordered from the MSP430 web page;
 - Supported by TI Software Support.
- Evaluation Version (Free):
 - 16 kB limit on C / unlimited ASM code size;
 - Download from MSP430 web page;
 - Supported by TI Software Support.

The new features of CCE v3 include:

- Free 16 kB code-limited version;
- Supports the large memory model (Place data >64k);
- Enhanced Compatibility with IAR C-code:
 - `#pragma` (ISR declarations), most intrinsics.
- GDB Debugger replaced by TI proprietary debugger that allows faster single stepping;
- Hardware Multiplier libraries (16-bit and 32-bit multiplies);
- CCE v2 project support (auto convert);
- Breakpoints:
 - Extended Emulation Module (EEM) support via unified breakpoint manager;
 - Using of EEM (predefined Use Cases);
 - Unlimited Breakpoints.

2.2.1 Installing CCE

Eclipse is a software development platform, developed in Java, which allows it to be used on different operating systems. One of its main features is that it is fully based on plug-ins, which gives it great versatility. This tool was originally developed by IBM, who have applied considerable financial resources and afterwards released it to the Open Source community. This makes it one of the most universal software development tools. There are a large number of institutions, both public and private, that support the development of this tool. A typical release of Eclipse comes with the components needed for the development of JAVA applications (JDT - Java Development Tools). In addition, others plug-ins are also part of the default version, the important ones being:

- ❑ Concurrent Version System (CVS): For control of code versions in production.
- ❑ Plug-in Development Environment (PDE): Relevant for those who want to expand the functionality of IDE through plug-ins.
- ❑ JUnit: Framework for code validation and test.

In addition to allowing the development of Java applications, Eclipse also allows the development in other programming languages. This requires the proper installation of their plug-ins or, alternatively, choosing a release that already includes them. The plug-in CDT (C/C++ Development Tools) enables the development of C/C++ code, but there are other extensions for Python or Cobol programming languages. Information relevant to the use of Eclipse can be found on its homepage located at <http://www.eclipse.org/>. On this site, in addition to the latest releases of Eclipse, resources to support those who are starting up using this tool can also be found.

The modular feature of Eclipse encourages its use as a basis for rapid and effective development of other tools. Using a common platform, the learning curve is rapid and simultaneously provides the reuse of modules already developed. Users of Eclipse can be divided into three communities:

- ❑ Committers: Community responsible for the official tool's development;
- ❑ Plug-in developers: Community that expands the capabilities of the tool through the development of plug-ins.
- ❑ Users: Community that uses the tool developed by the two previous communities.

Code Composer Essentials (CCE) version 3 is based on Eclipse release 3.2 (Callisto). On the market there are hundreds of plug-ins that can be added in order to enhance or optimize a particular aspect of CCE. One of the available repositories for plug-ins developed for the Eclipse is the Eclipse Plugin Central located at <http://www.eclipseplugincentral.com/>. This development tool has advanced capabilities to support the development of applications for the MSP430 family. Among them are the support for the use of

breakpoints, either hardware or software. CCE supports code debugging activities, with support for features such as code step-by-step execution, or fast and efficient access to registers and memory locations. There is complete compatibility between the C programming language syntax used and the great diversity of code examples available.

CCE supports all elements of the MSP430 family. It also includes the expanded memory devices, MSP430X. Different programming languages such as assembly, C and C++ are supported.

CCE installation

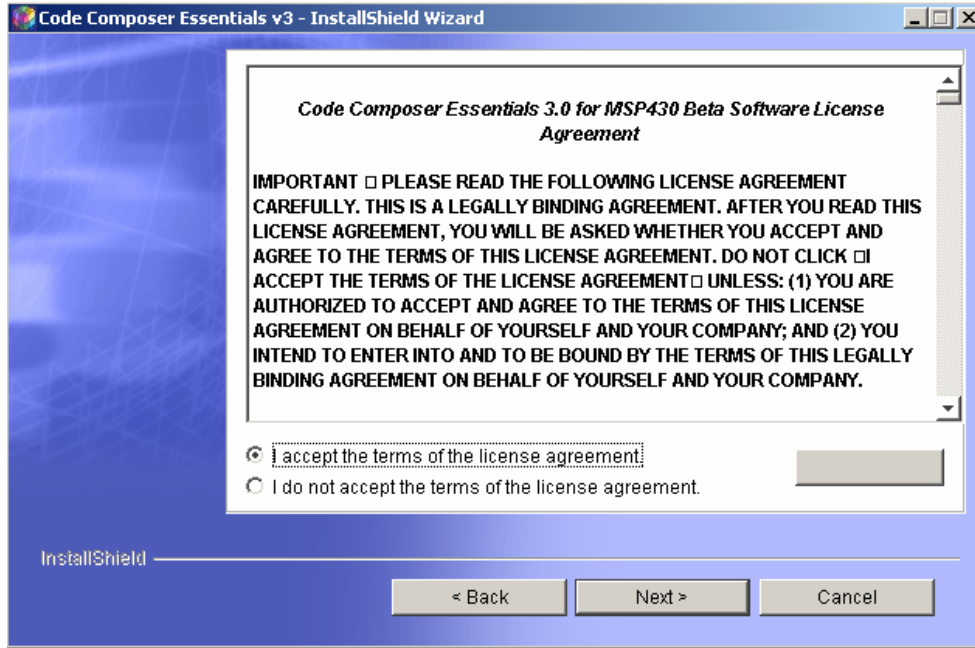
Most of the installation of CCE is automated. It is only necessary to provide some user indications as to how the program installation should continue. The installation process begins with the welcome window shown in *Figure 2-1*.

Figure 2-1. CCE installation process - Welcome window.



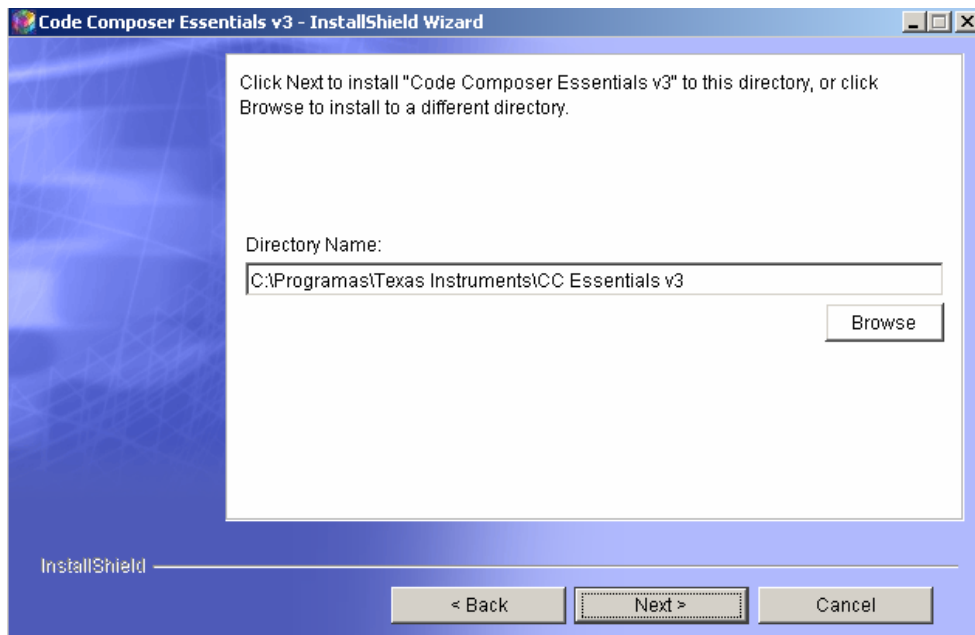
The installation process starts with the acceptance of the software license agreement, as shown in *Figure 2-2*.

Figure 2-2. CCE installation process – Software license agreement window.



The first question asked by the software installer is the directory on the local disc where to install the software. It is recommended to accept the directory suggested by the application.

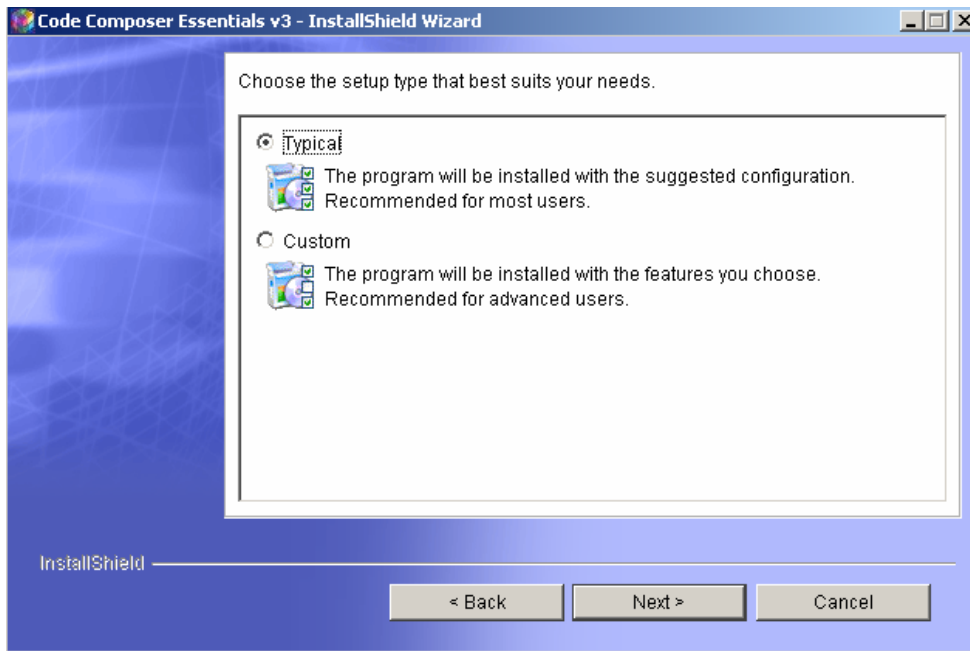
Figure 2-3. CCE installation process – Installation directory window.



Two different installation procedures are available. In the **Typical** installation procedure, the tool is installed with all the default options. The installation procedure **Custom** allows the user to select

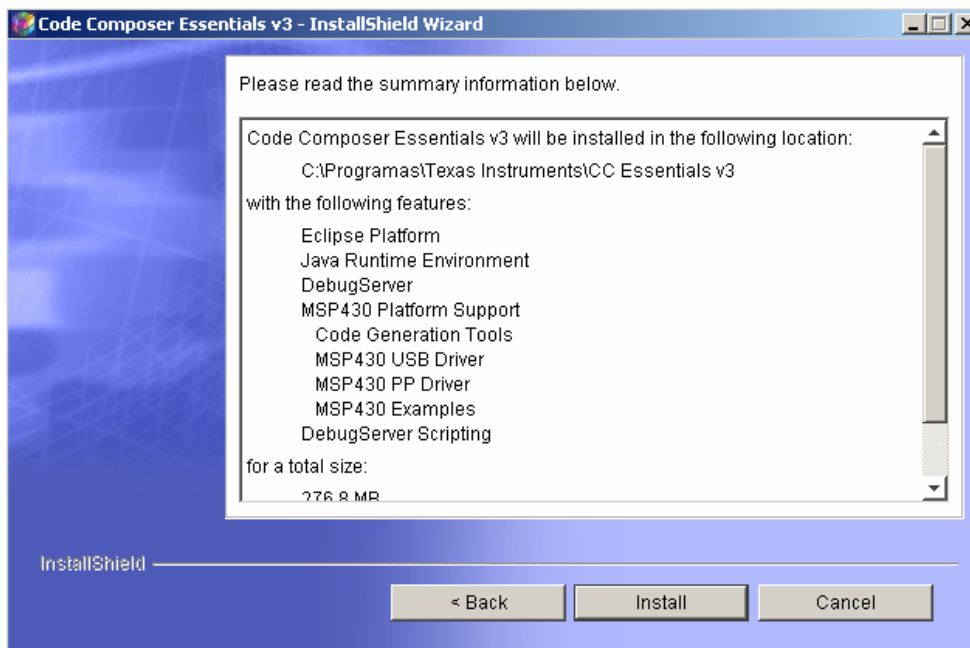
the components to install. It is recommended to select the typical installation, as shown in *Figure 2-4*.

Figure 2-4. CCE installation process – Installation procedure window.



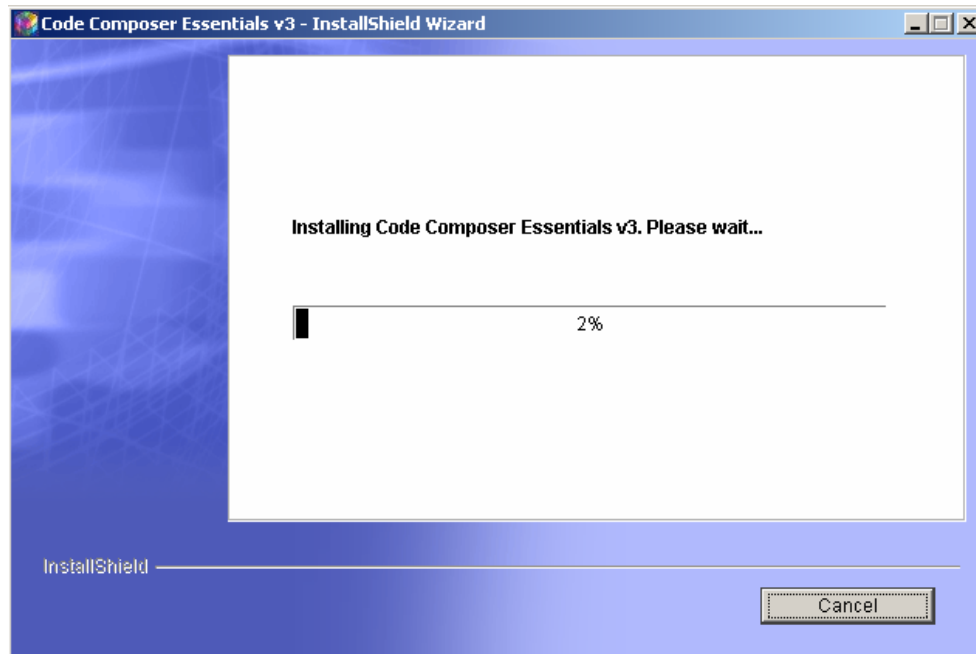
Whichever installation procedure was chosen in the previous window, the components installed are listed in the next installation window (see *Figure 2-5*).

Figure 2-5. CCE installation process – Installed components window.



The installation process begins as shown in *Figure 2-6*.

Figure 2-6. CCE installation process – Installation process evolution window.



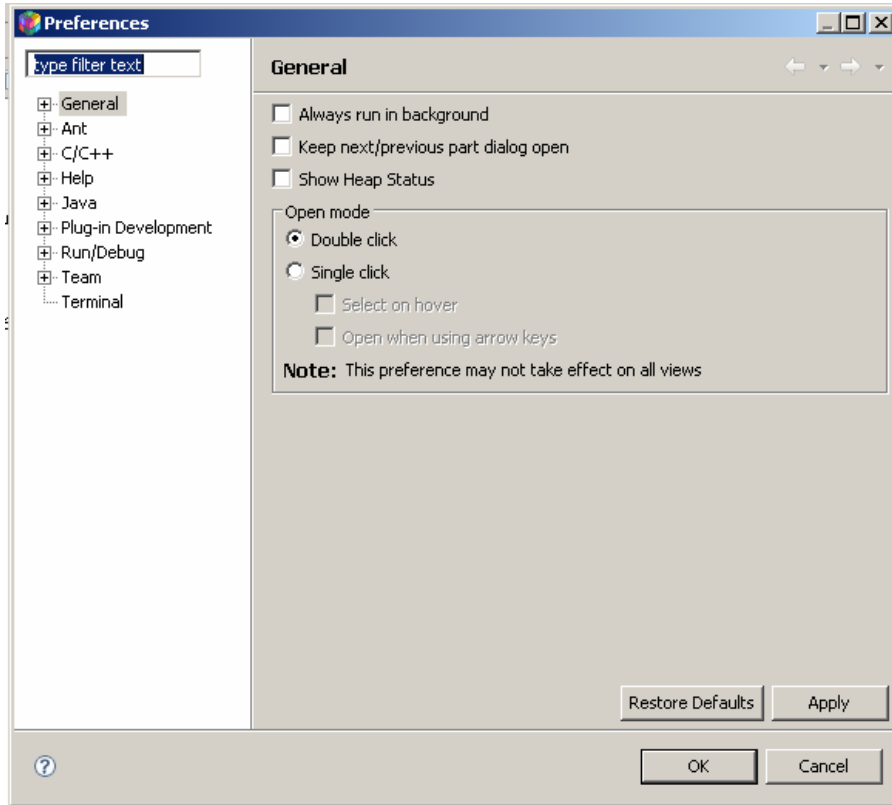
2.2.2 Introduction to CCE IDE

The introductory overview in the use of CCE will continue with a practical example, addressing some of its main features. Let us begin by building a project. This project will be configured with respect to the hardware, i.e., the MSP430 family device.

Launching the workbench

The term “Workbench” refers to the integrated development environment of all tools necessary for the development and management of projects. When CCE is started, it asks the user where they want to locate the work directory (workspace). If this will be always the same, this question can be inhibited in future openings of CCE by choosing the inhibit option for this window. If it is necessary to change the location of the workspace in future projects, select **Window > Preferences**. This menu allows access to the CCE preferences configuration. CCE’s start and stop can be configured in the **General > Start and Shutdown** option. The organization of the various configuration options presented depends on to the modules installed. Some time should be spent here, opening the various options and identifying where to set general aspects of the tool such as: general appearance, editor, shortcut keys, etc. The CCE preferences window is shown in *Figure 2-7*.

Figure 2-7. CCE workbench – Preferences window.



After choosing the location where the workspace will be stored, it opens by default in the project construction perspective. The concept associated with a perspective is important for the correct understanding of CCE operation. A perspective provides that for a given task there is an organization of windows most appropriate to its implementation. Changing perspective involves reformulating the workspace for a new Windows configuration that promotes the development of particular task. There are two major perspectives: **C/C++** for editing, management and compilation of projects, and **Debug** for debugging the applications. The working perspective is selected in the upper right hand corner of the application.

























By default, the windows included in the **C/C++** perspective are: **C/C++ Projects** (to manage the projects); **Editor** (to edit files); **Outline** (to view data); **Console** (to send messages); **Problems** (identifies problems found in the project). The icons associated with the various tasks that can be performed in this perspective are shown together in *Table 2-1*.

Table 2-1. Tasks icons and description – C/C++ perspective.

Button	Description	Button	Description
	Open a new perspective		Save the active editor contents
	Save the contents of all editors		Save editor contents under a new name or location
	Opens the search dialog		Print editor contents
	Open a resource creation wizard (New)		Open a file creation wizard
	Open a folder creation wizard		Open a project creation wizard
	Open the import wizard		Open the export wizard
	Run incremental build (Build All)		Run a program
	Debug a program		Run an external tool
	Cut selection to clipboard		Copy selection to clipboard
	Paste selection from clipboard		Undo most recent edit
	Redo most recent undone edit		Navigate to next item in a list
	Navigate to previous item in a list		Navigate forwards
	Navigate backwards		Navigate up one level
	Add bookmark or task		Open a view's drop down menu
	Close view or editor		Pin editor to prevent automatic reuse
	Filter tasks or properties		Go to a task, problem, or bookmark in the editor
	Restore default properties		Show items as a tree
	Refresh view contents		Sort list in alphabetical order
	Cancel a running operation		Delete selected item or content
	Last edit location		Toggle Mark Occurrences
	Assembly instruction only		

By default, the windows included in the **Debug** perspective are: **Debug** (provides information concerning the debug process); **Editor** (to edit files); **Variables/Expressions** (to evaluate variables and expressions values during debug); **Console** (console to send messages); **Registers/Breakpoints** (to evaluate the contents of registers and to define code breakpoint); and **Disassembly/Memory** (to evaluate the assembly code and memory map occupation). The icons associated with the various tasks that can be performed in this perspective are shown together in *Table 2-2*.

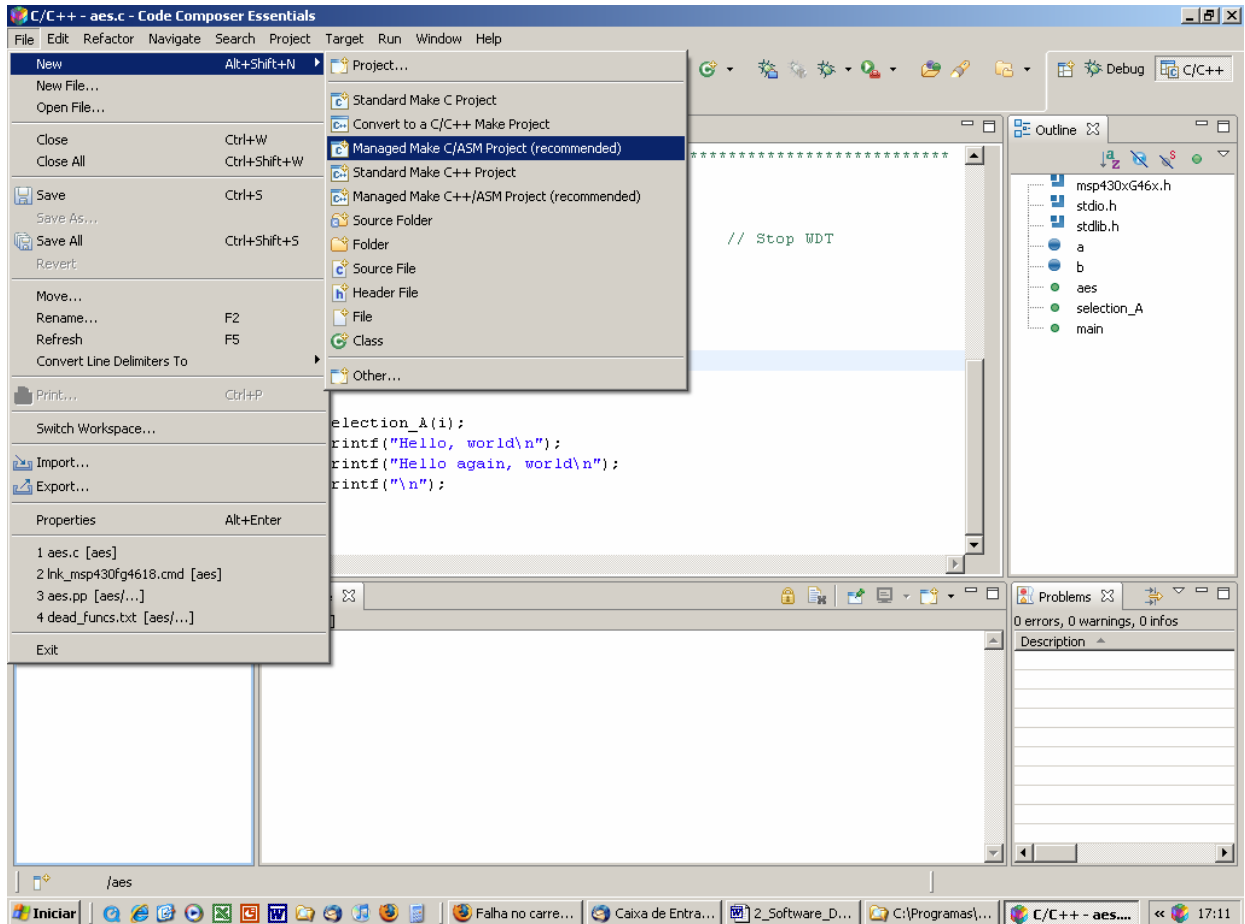
Table 2-2. Tasks icons and description – Debug perspective.

Icon	Command	Description
	Create New	Create a new project, folder, or file.
	Save	Save the content of the current editor. Disabled if the editor does not contain unsaved changes.
	Print	Prints the content of the current editor.
	Build All	Compiles all files for all projects in workbench.
	Enable/Disable Breakpoints	Enables or disables a breakpoint at the specified location.
	Toggle Breakpoint	Toggles a breakpoint at a specific address selected in the Edit window.
	Change Build Configuration	Lists available build configurations to choose.
	New C/C++ Project	Creates a new C/C++ project.
	New C/C++ Source Folder	Creates a source folder within the current project.
	New C/C++ Source File	Creates a source file within the current project.
	New C/C++ Class	Creates a C++ class within the current project.
	Debug Active Project	Debugs the current active project.
	Launch TI Debugger	Launches the TI specific debugger.
	Debug	Launches the Debug dialog box.
	Run	Launches the Run dialog box
	External Tools	Launches the External Tools dialog box
	Open Type	Brings up the Open Type selection dialog to open a type in the editor. The Open Type selection dialog shows all types existing in the workspace.
	Search	Launches the C/C++ Search dialog box
	Select Working Sets	Selects a working set from the list to be the active one. Working sets group elements for display in views or for operations on a set of elements.
	Next Annotation	Selects the next annotation in the resource that is currently active in the editor area. Supported in the Java editor.
	Previous Annotation	Selects the previous annotation in the resource that is currently active in the editor area. Supported in the Java editor.
	Go to Last Edit Location	Returns editor view to the last line edited, if the file that was last edited was closed it will be re-opened.
	Back	Navigates back through open files.
	Forward	Navigates forward through open files.

Creating a Project

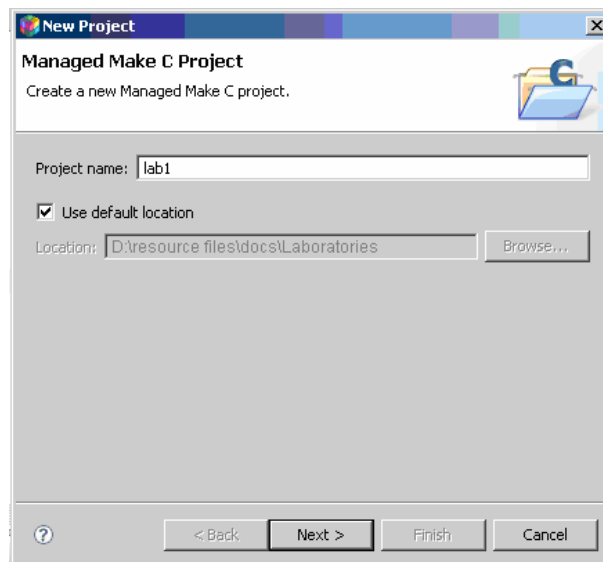
Select the option **File> New > Managed Make C\ASM Project** (recommended) to create a project. Other project options are available, but with the above option, the project process creation is more automated. The *Figure 2-8* shows the window where the option should be selected.

Figure 2-8. CCE workbench – Project creation process window.



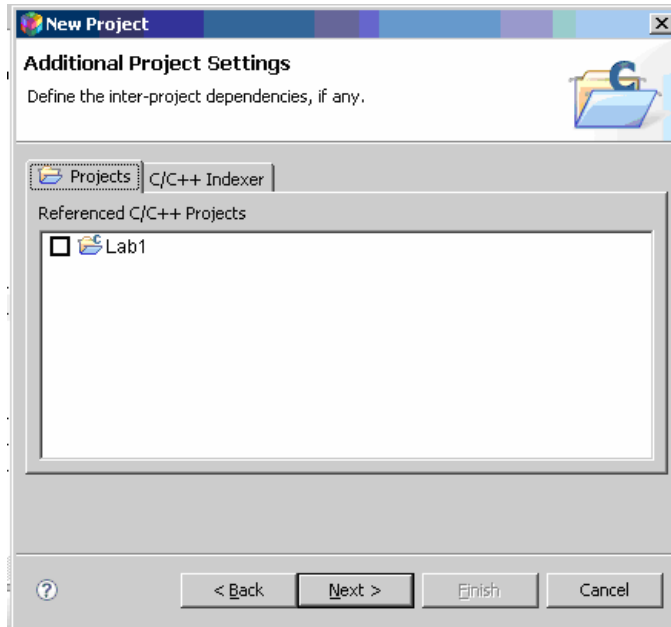
After choosing this option, a procedure for the creation of projects for the MSP430 family of microcontrollers is provided. The user must answer the first question concerning the project's name. By default, all the project files are stored within a folder, with the name of the project in the directory chosen for the *workspace*. The New Project window is shown in the *Figure 2-9*.

Figure 2-9. CCE workbench – New project name window.



Afterwards, some additional settings are made to the project, such as whether there is any dependency of this project on another. If this condition is true, the dependency should be established through the window shown in *Figure 2-10*.

Figure 2-10. CCE workbench – Project dependency window.



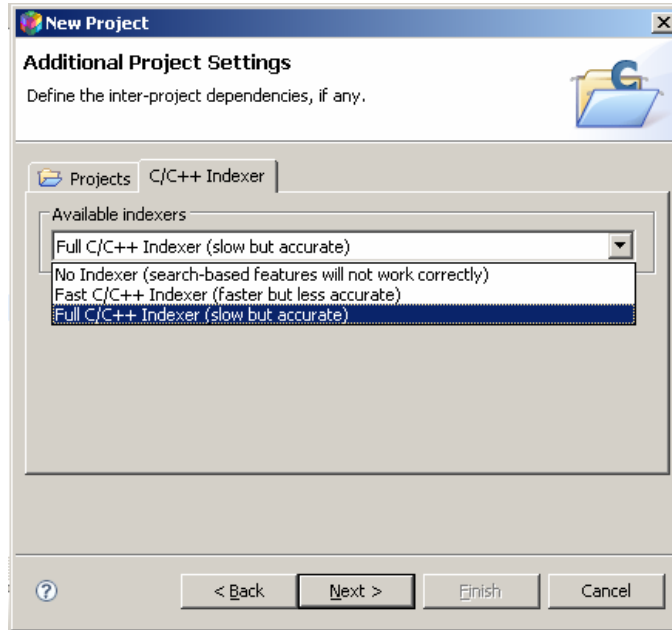
Information indexing functionality is part of the C/C++ Project. It uses a parser to create a database of the contents of the project files. This feature is used during the information search, the project navigation, and in the content assistant. The indexing task is performed in the background and reacts to changes in content such as: C/C++ project creation or deletion; file creation or deletion; file import; content of files changes.

There are three options for setting up the operation of this functionality:

- Without Project contents indexing (No Indexer);
- Fast C/C++ or,
- Full Indexer C/C++ Indexer).

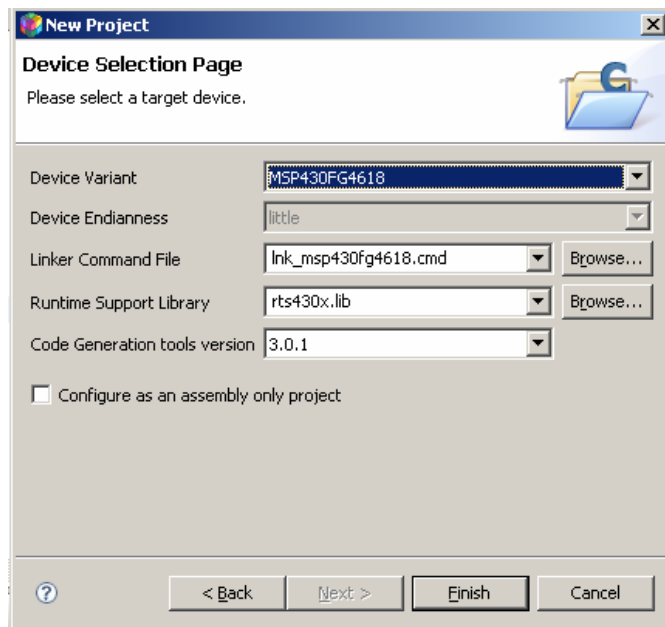
These two last options differ mainly in the required processing time of the indexing task and results quality. The configuration window of this feature is displayed in *Figure 2-11*.

Figure 2-11. CCE workbench – Project indexing window.



In the final window displayed during the project's creation procedure (see *Figure 2-12*), the device with which the project is being developed must be chosen. By choosing the device, the appropriate linker command file and supporting libraries are selected automatically.

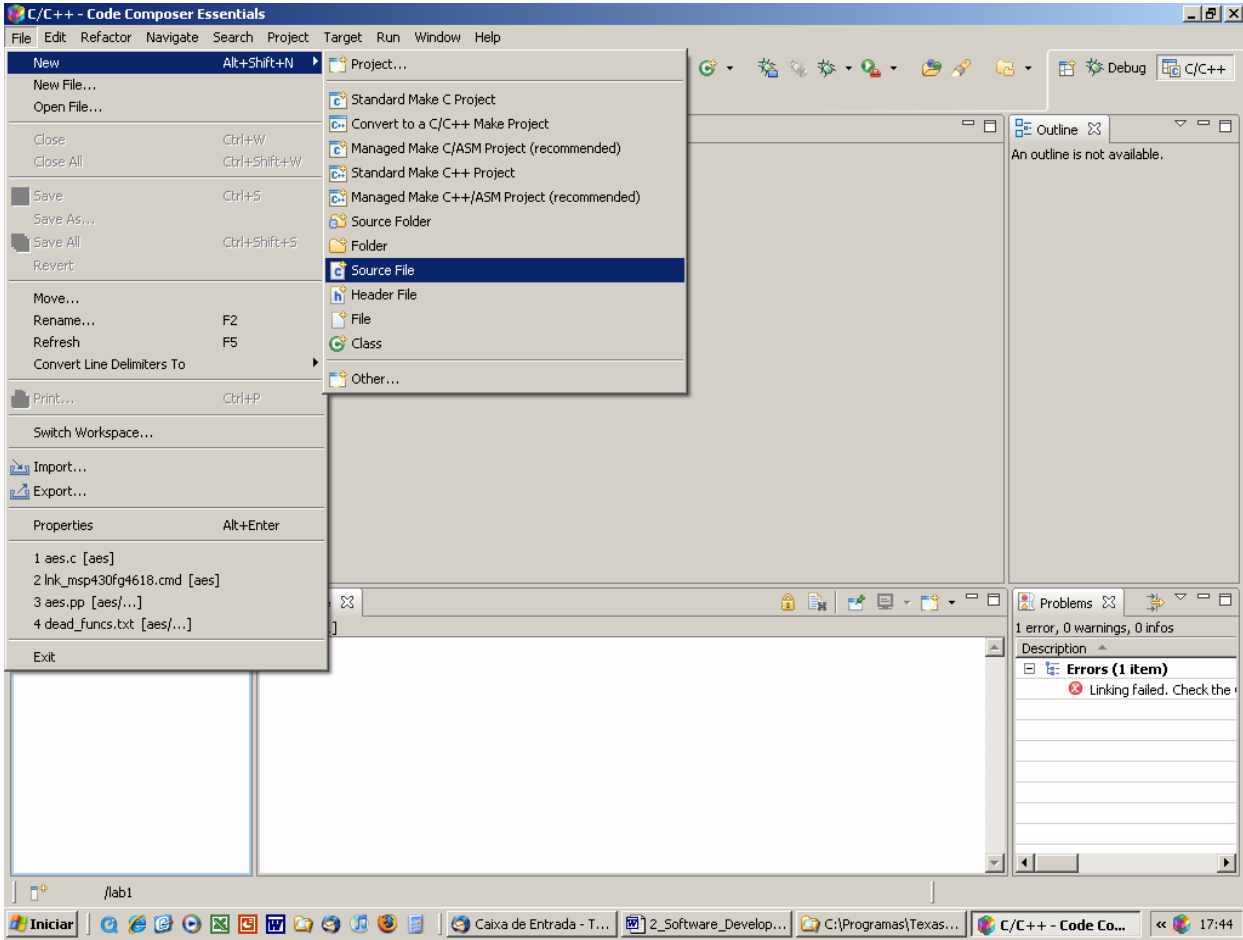
Figure 2-12. CCE workbench – Device selection window.



The project's creation can then be finalised by choosing the option **Finish**. At any time, it is possible to go back to previous windows by choosing the option **Back**.

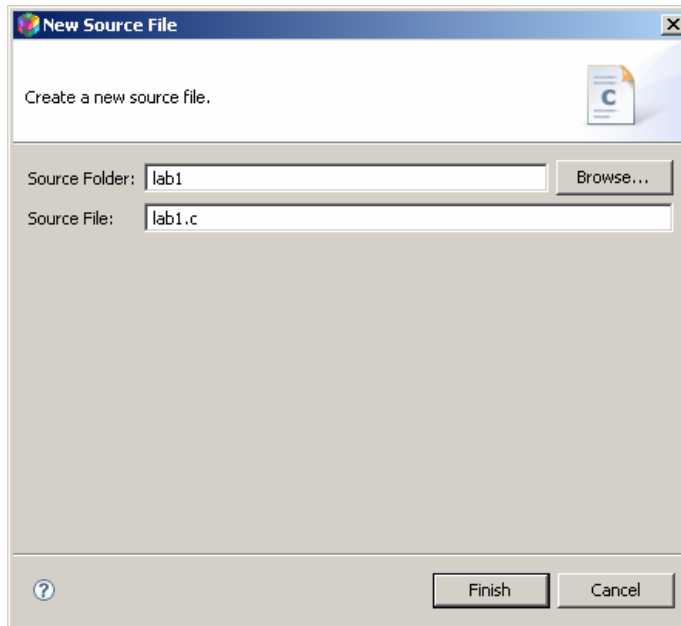
The next step is to add the source code file to the project. Choose **File > New > Source File**. In this menu the option to create `.C` type file should be selected, as shown in *Figure 2-13*.

Figure 2-13. CCE workbench – Source code file creation procedure.



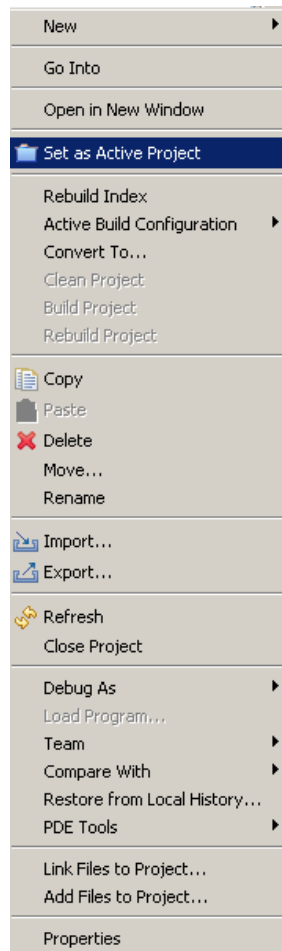
The name of the file is then requested in the window as shown in Figure 2-14. Do not forget to add the file extension such as "myfile.c" so that it is recognized as a C file.

Figure 2-14. CCE workbench – Source code file creation window.



The project is automatically selected as the default project. Although the workspace allows several projects to be opened simultaneously, it allows only one of them to be active. To select an active project, its name must be selected with the mouse's right button in **C/C++ Project** view, in order to show the context menu. Then the option **set as active project** must be selected. From here, the expression **[Active-Debug]** will appear. In the context menu there are other options to manage the project: add or remove files, import or export resources, edit the properties and so on.

Figure 2-15. CCE workbench – Set as an active project window.

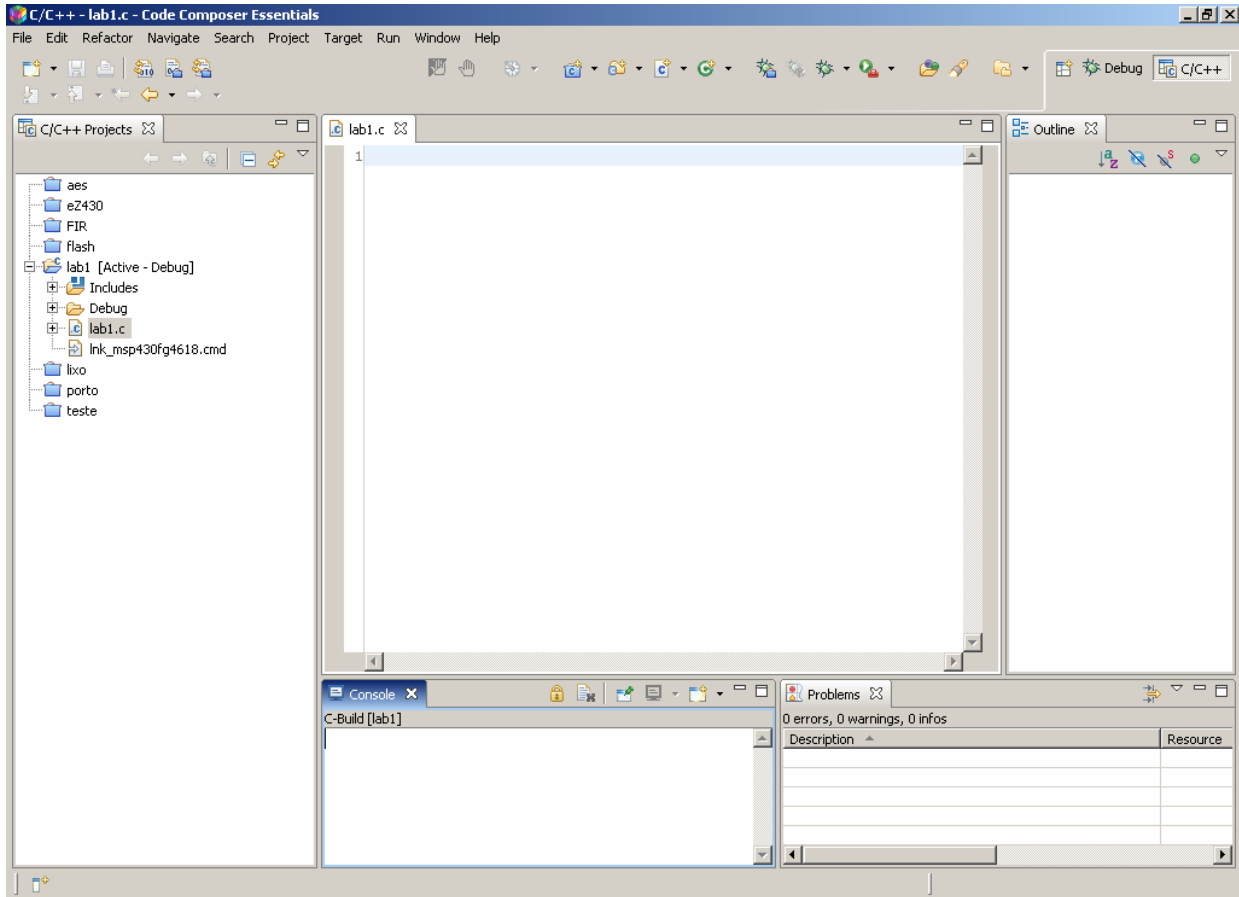


At this point, it is possible to start editing the project's source code. CCE has all the capabilities inherited from the Eclipse edition. Adding the file lab1.c, which already exists, is done through the option **add file to project**. This file can be found in **Project > add file to project**, as in the context menu of the view **C/C++ Projects**. The file lab1.c can be removed from the project by simply selecting it in the view and selecting the option **delete**. Note that when the file is removed, it will be cleared from the directory.

Code Editor

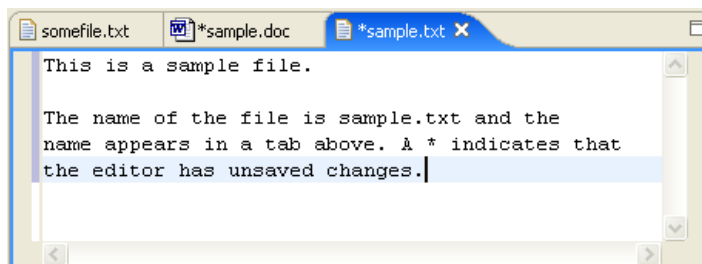
The text editor included in CCE is a versatile tool and very effective for helping with the code editing task. The C/C++ perspective is shown in *Figure 2-16*.

Figure 2-16. CCE workbench – C/C++ perspective.










The text editor has a set of features that allow speeding up the code editing process. An overview of the text editor is shown in *Figure 2-17*.

Figure 2-17. CCE workbench – Text editor window.



On the left hand side of the text editor there is a bar. Several colourful icons indicate different options. The text editor identifies a

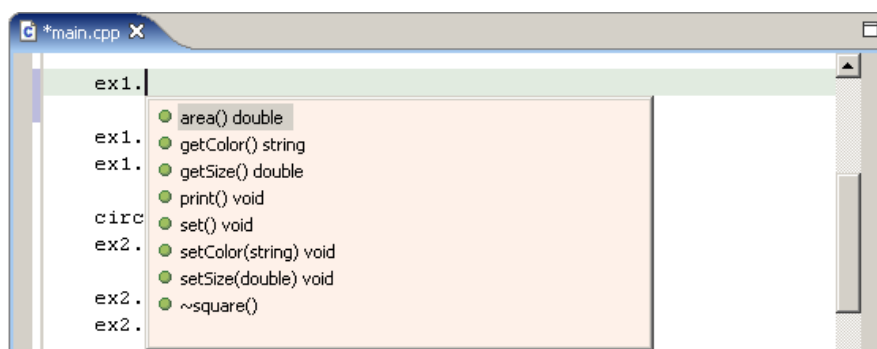
bookmark , a breakpoint , add a task , search , a error mark , a warning , and a information .

Code editing is greatly facilitated using features such as search and replace. To accomplish this task, the user must select **Edit > Find/Replace**. In addition to the normal features of search and replace, the option **Search > File** allows the use of more elaborate expressions. For example, it provides the global replacement in all files of a specific directory. The search and replace tasks previously performed can be found on the **drop-list**.

CCE can regularly save the opened files for editing in order to prevent losses caused by system crashes. To use this function, select **Window > Preferences > General > Workspace** and specify the time interval at which this task should be performed automatically. The project can also be saved whenever it goes through project build.

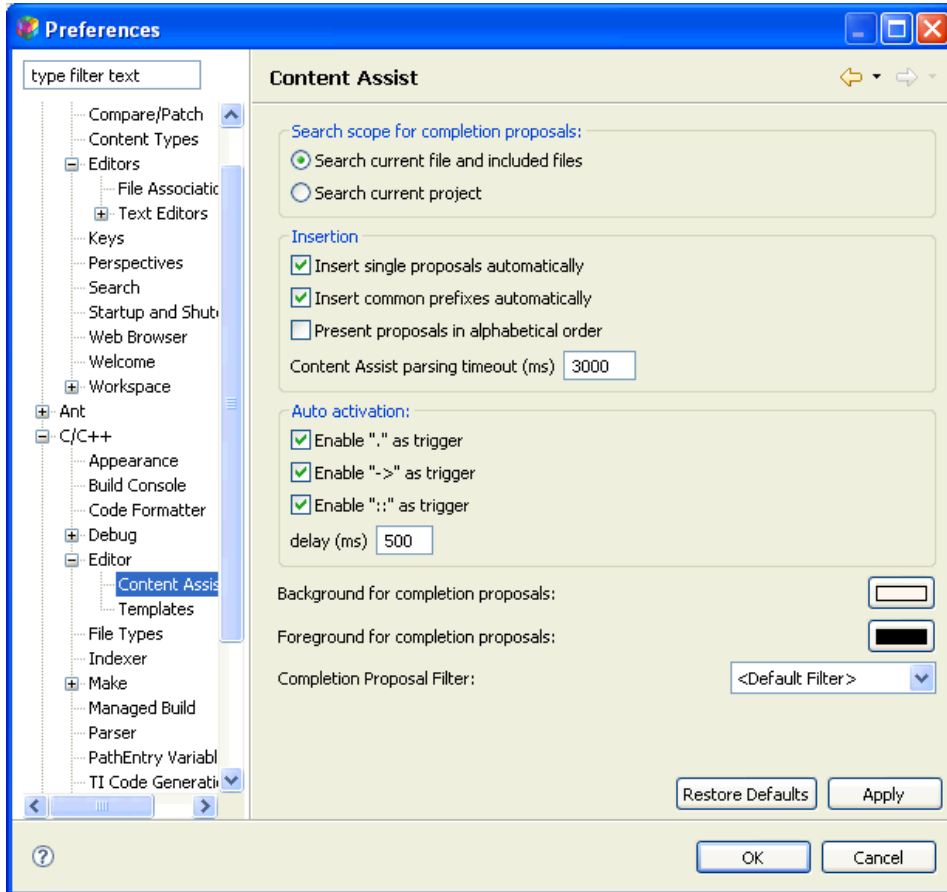
The content wizard is a very effective tool to support the writing of code. It is possible to automatically insert a code structure model, previously defined, as an alternative to writing it out completely (see an example in *Figure 2-18*). To insert a model of a structure, it is only necessary to write the first letters in the text editor and then press the **Ctrl + Space** keys in order to display a list of the corresponding models. The options in the list can be reduced by continuing writing the structure name. The **Arrow Up** and **Arrow Down** keys can be used to select the desired model and by pressing the **Enter** key to accept the selection. At any time the **Esc** key allows editing to continue without the use of the content wizard.

Figure 2-18. CCE workbench – Content wizard.



The behaviour of this feature can be configured in **Window > Preferences**. In *Figure 2-19* shows the configuration page of the content wizard.

Figure 2-19. CCE workbench – Preferences window.

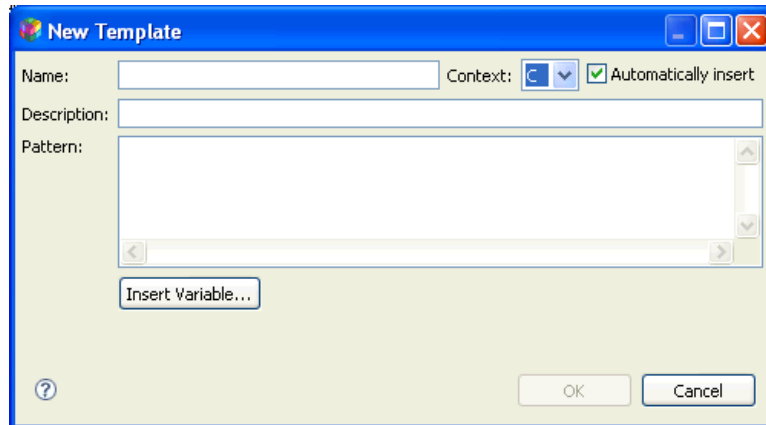


The search range can be restricted to only the edited file and to the files included therein (**Search current file and included files**), or alternatively a search can be in the whole project (**Search current project**). Automatic model insertion is allowed, as long as it is the only one at the options list (**Insert single proposals automatically**). The user may also request that the suggestions list is presented in alphabetical order (**Present proposals in alphabetical order**). Another aspect that can be configured is related to the time (in milliseconds) that the content wizard delays to suggest a list (**delay**), or the duration of the validity of the suggestion (**Content Assist Parsing timeout**).

In addition to the sequence of **Ctrl + Space** keys, the content wizard can also be set automatically when the following characters are typed: ".", "->" or "::".

CCE is already provided with a set of models. However, it is possible to create new models by opening the models editor. Expand the C/C++ perspective in **Window > Preferences**, and select **Editor > Templates**. The option **New** must be selected to create a new model (see *Figure 2-20*).

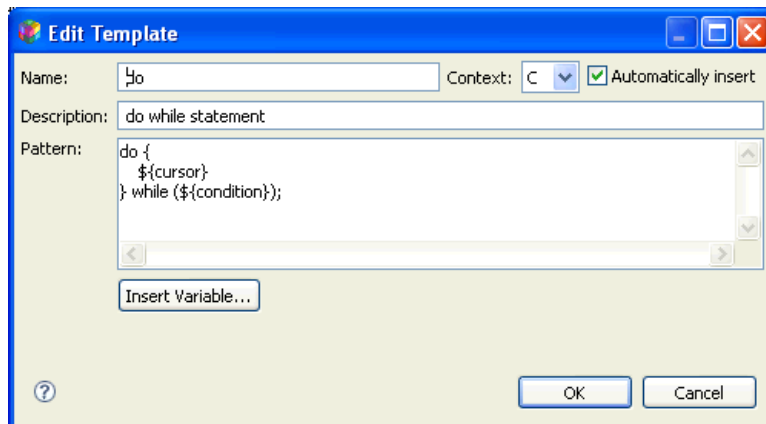
Figure 2-20. CCE workbench – New template window.



A name must be given for the new model. The context in which the model is valid should be selected. In the **Description** field a brief description of the model can be added. The model itself is described in the **Pattern** field. To insert a variable, use the **Insert Variable** option.

One way to learn how to create models, or even how to customize existing models, can be achieved using the model editing feature (see Figure 2-21). To access this feature, the **Editor > Templates** option must be chosen, and is visible after expanding the C/C++ perspective in **Window > Preferences**.

Figure 2-21. CCE workbench – Edit template window.



The procedures to check on this page are identical to those described earlier for building new models.

The CCE supports the following intrinsic functions for the MSP430 family devices:

- void __no_operation(void);
- void __enable_interrupt(void);

- ❑ void __disable_interrupt(void);
- ❑ unsigned short __get_interrupt_state(void);
- ❑ void __set_interrupt_state(void);
- ❑ void __op_code(unsigned short);
- ❑ unsigned short __swap_bytes(unsigned short);
- ❑ void __bic_SR_register(unsigned short);
- ❑ void __bis_SR_register(unsigned short);
- ❑ unsigned short __get_SR_register(void);
- ❑ void __bic_SR_register_on_exit(unsigned short);
- ❑ void __bis_SR_register_on_exit(unsigned short);
- ❑ unsigned short __get_SR_register_on_exit(void);
- ❑ void __set_SP_register(unsigned short);
- ❑ unsigned short __get_SP_register(void);
- ❑ unsigned short __bcd_add_short(unsigned short, unsigned short);
- ❑ unsigned long __bcd_add_long(unsigned long, unsigned long);
- ❑ void __data20_write_char(unsigned long, unsigned char);
- ❑ void __data20_write_short(unsigned long, unsigned short);
- ❑ void __data20_write_long(unsigned long, unsigned long);
- ❑ unsigned char __data20_read_char(unsigned long);
- ❑ unsigned short __data20_read_short(unsigned long);
- ❑ unsigned long __data20_read_long(unsigned long);

File history

Another of the features included in CCE allows comparisons between two files or previous versions of it, using file history stored during the work sessions. This feature allows searching and integrating the different versions between files.

The file to compare with the local history must be selected in one of the navigation views. In the context menu (select the file, mouse right button click), choose the **Compare With > Local history** option. In response to this selection, the **Compare With Local History** window is opened. A previous state presented in the **Local History** list can be chosen. The text comparison editor will then be open. The navigation between changes is made through the buttons **Select Next Change** and **Select Previous Change**.

It is possible to recover a resource that has been cleared of the workspace. The procedure is as follows: the project to which is to be restored to a previous state must be chosen in the navigation view. In the context menu, the **Restore from Local History...** option should be chosen. The **Restore From Local History** window will open on the right hand side of the screen. It will display all the files

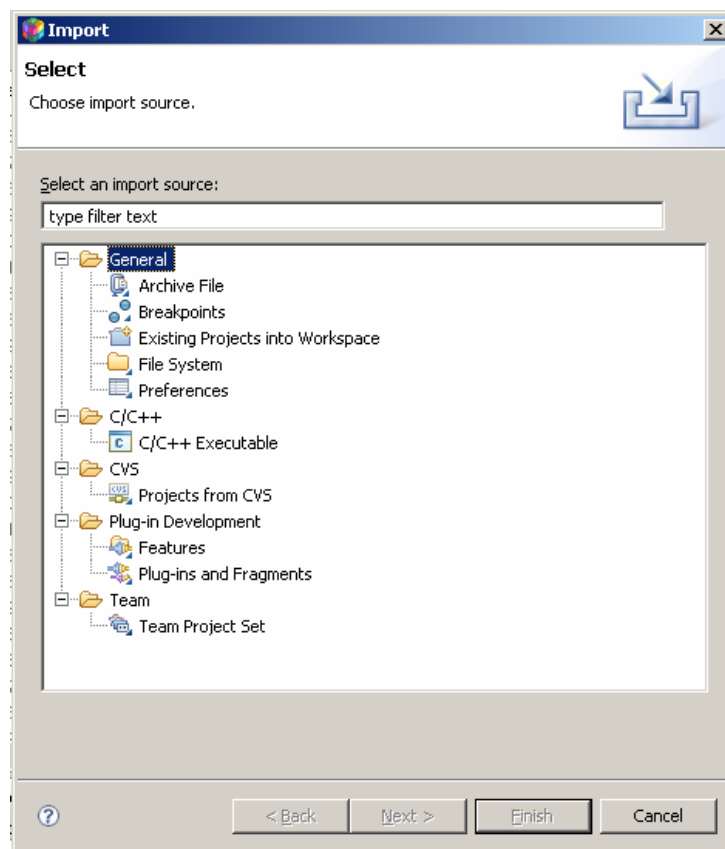
that were previously part of the project. The last file version, or any of those previous, can be fully recovered by choosing it in the **Local History** list. The restore will be done after clicking **Restore**.

The file history feature can be configured according to Project needs. In the preferences page **General > Workspace > Local History**, it is possible to establish the number of days that a particular file history should remain available and the maximum number of entries per file. If the defined value is exceeded, the oldest changes are removed in order to provide memory space for the latest. The maximum size available to store the file history can also be defined. If its size is exceeded, the file history ceases to be performed.

Import and Export functionality

CCE has the capability to import and export different types of information. In the context menu of the view **C/C++ Projects** it is possible to activate the import process choosing the option **Import**. This process allows importing resources such as those listed in the following figure (*Figure 2-22*).

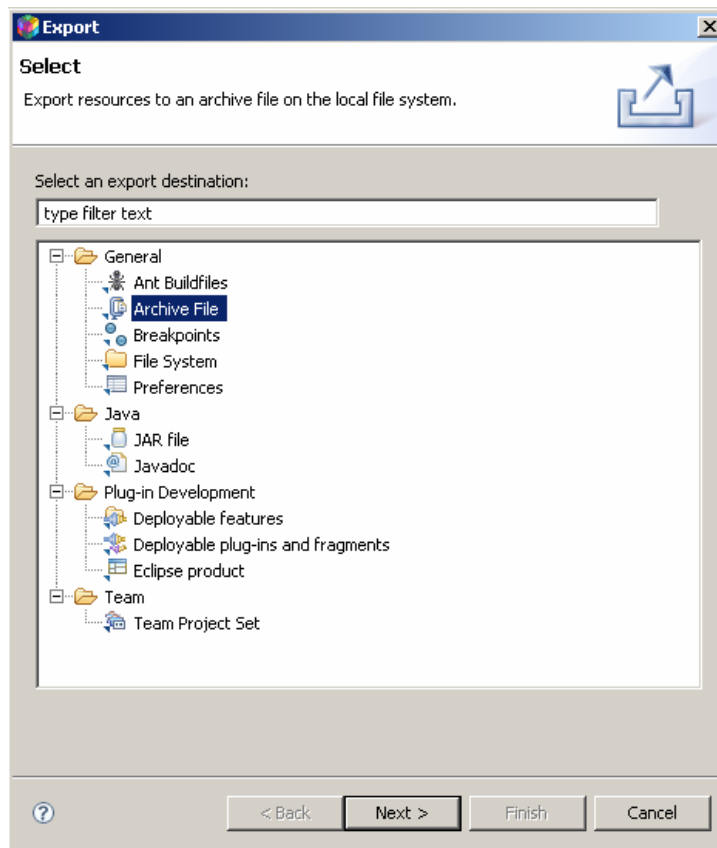
Figure 2-22. CCE workbench – Import options window.



Following the instructions given by the import wizard: **Archive File** (imports files stored in a compressed file); **Import Breakpoints** (imports a breakpoints scenario previously defined in another or in the same project); **Existing Project into Workspace** (imports a project into the actual workspace); **File System** (imports a file); **Preferences** (imports the CCE configuration preferences), etc.

When the **Export** option from the context menu is selected, the window with the export procedures is displayed, as shown in *Figure 2-23*.

Figure 2-23. CCE workbench – Export options window.



Similar to the import procedure, the resources belonging to a Project can be exported: **Archive File**; **Export Breakpoints**; **File System**, **Preferences**, etc...

Project Configuration details

The project configuration defines a set of options to build it. The options defined at this level are applied to all the files of the project. CCE allows setting different options for building at different stages of the project.

Building a project is a process that generates new features starting from the existing ones, or updates them if they already exist. In the workspace, different builds for different types of projects, or for different stages of development can be invoked. The different build types are:

- ❑ An **Incremental build** uses a build held earlier. Thus, from a past build state, it applies the necessary changes to the resources that have been changed;
- ❑ A **Clean Build** ignores all previous builds as well as problems or errors that led to them. This type of build will transform all resources in accordance with the set of rules in the project build configuration.

The project builds can be done in two different ways. The behaviour configuration can be defined in **Window > Preferences > General > Workspace**:

- ❑ Automatic builds are always incremental and are always carried out throughout the workspace. Whenever there is resource alteration, it will initiate a build process. This option may be disabled in **Window > Preferences > General > Workspace**;
- ❑ A manual build is always triggered by the user. This type of project build option can be clean or incremental, and can be applied on a group of project files, or to the whole workspace.

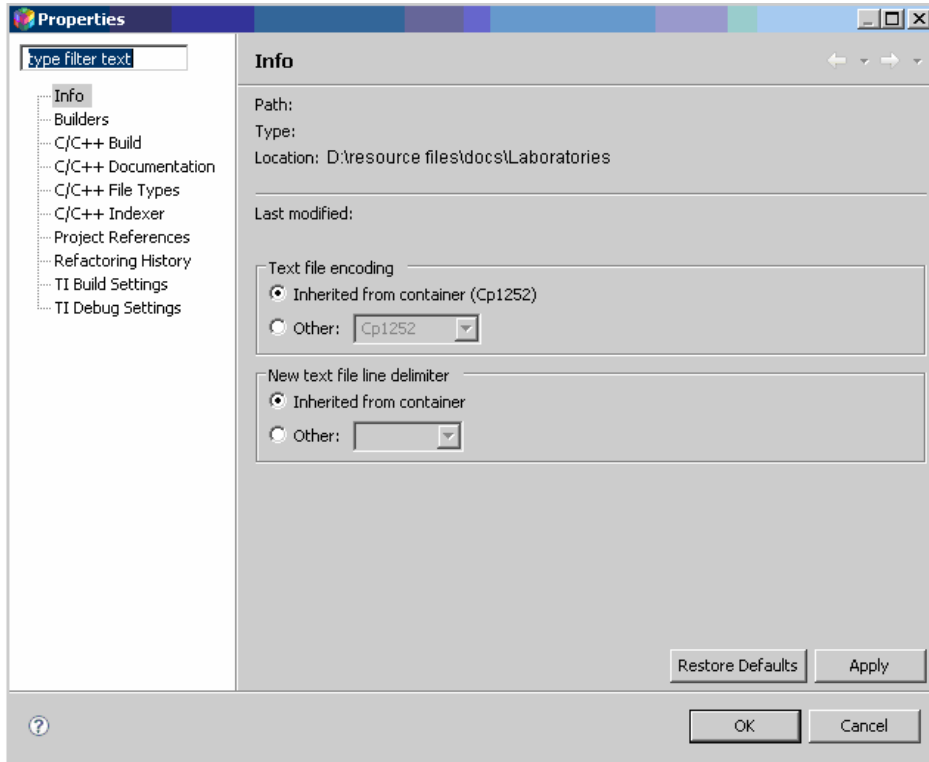
The order in which the build is processed is configurable. If the project contains mentions to another project, the CDT (C/C++ Development Tools) must first build the initial project. The order in which the build takes place may be selected in **Window > Preferences > General > Workspace > Build Order**.

In order to bring the various parts of a project together, it is necessary to build the project using a configuration stored in a file. There are several build files available, giving different build alternatives, so the build file most appropriate to the stage of the project must be selected. The CDT can automatically generate build files whenever a **Managed Make C project** or **Managed Make C++ project** is created. Each project is therefore created with two default settings: Debug and Release. Other additional settings can be configured. Whenever a project is created or an existing project is opened, the first configuration in the list of alphabetically sorted items, is taken as active.

The project's compiler and linker definition options are complex. Therefore, it is recommended to carefully read the documentation related to the compiler and to the assembler/linker.

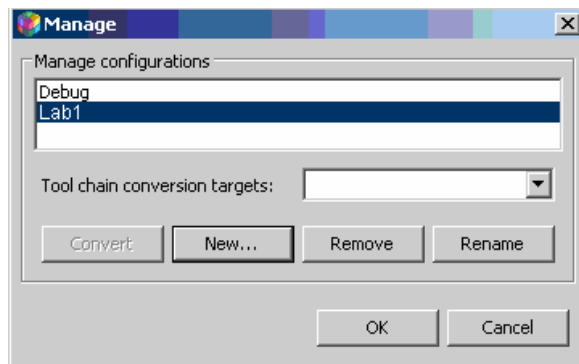
After the project's creation, it must be configured for the appropriate compiler, linker and debugging options. By selecting the option **Properties** from the context menu of the view **C/C++ Project**, the project's configuration window is displayed (*Figure 2-24*). The compiler, linker and debug options can be defined here.

Figure 2-24. CCE workbench – Configuration window.



The management of build configurations is found under the option **C/C++ Build**, accessed via the **Manage** button. Through it the management features can be accessed (see *Figure 2-25*).

Figure 2-25. CCE workbench – Manage window.



It is possible to create new build configurations, delete the existing ones or modify their names. The name of the modified configuration is selected in **configuration**.

The C/C++ compiler used by the project is controlled by the project's properties. To view the project properties in the dialog box that appears, the page **C/C++ Build** allows control of the variety of configurations, including:

❑ **Build Options:** specifies the options that affect all project files. This dialog page allows selection of the appropriate options, including those for compiling and linking. It is also possible to specify whether the compiler uses **Stop On Error** or **Keep Going On Error**. The second option allows the compiler to build all projects referenced, even if the current project contains errors. The build command specifies the make file to use.

The MSP430X devices allow data to be located anywhere in the 20-bit address space. By enabling this option, the compiler will use instructions that need a larger space for their storage. Hence, the memory space occupied by the final program will be greater. The option (- large_memory_model) is valid only when the project is intended as a MSP430X device defined by the compile option (- vmspx). The programs for MSP430X processors should be compiled with RTS libraries supplied for that purpose (rts430xl.lib and rts430xl_eh.lib).

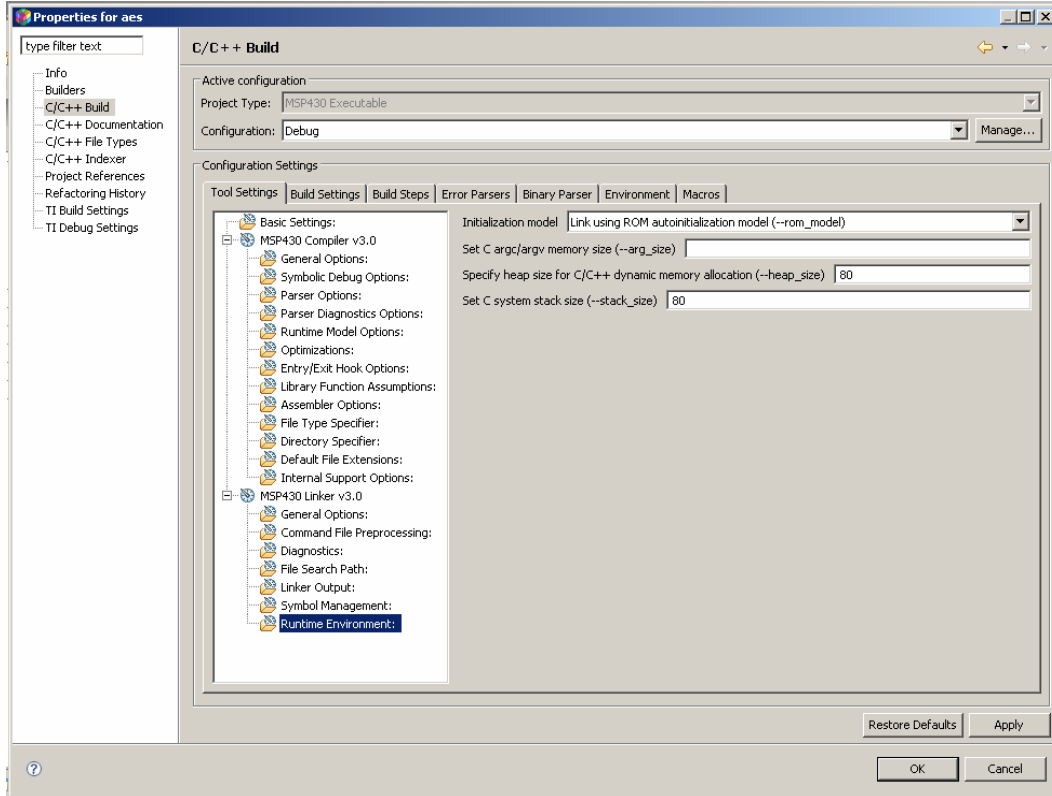
The compilation option (- silicon_version) selects the CPU version using the 4 least significant processor's identification digits. If this option is not used, the compiler will construct the default code for the device.

In the process of linking, if all references to the multiplication routines of integers are to be replaced by the routines versions that use the hardware multiplier option (- use_hw_mpy), the device multiplier's length must be specified. To use the 16-bit hardware multiplier, present in most devices, choose the option 16. For devices belonging to the F4xx family, which has a 32-bit multiplier module, chose the option 32. Finally, for the new 5xx family, which also has a 32-bit multiplier, use the F5 option. The default option is 16-bit hardware multiplier module.

The model used for the initialization of static variables in the C program can be specified as: None, Link using ROM autoinitialization model (- rom_model), or link using RAM autoinitialization model (- ram_model). The C/C++ compiler produces tables of data for automatic initialization of global variables. These tables are placed in the section identified by .cinit.

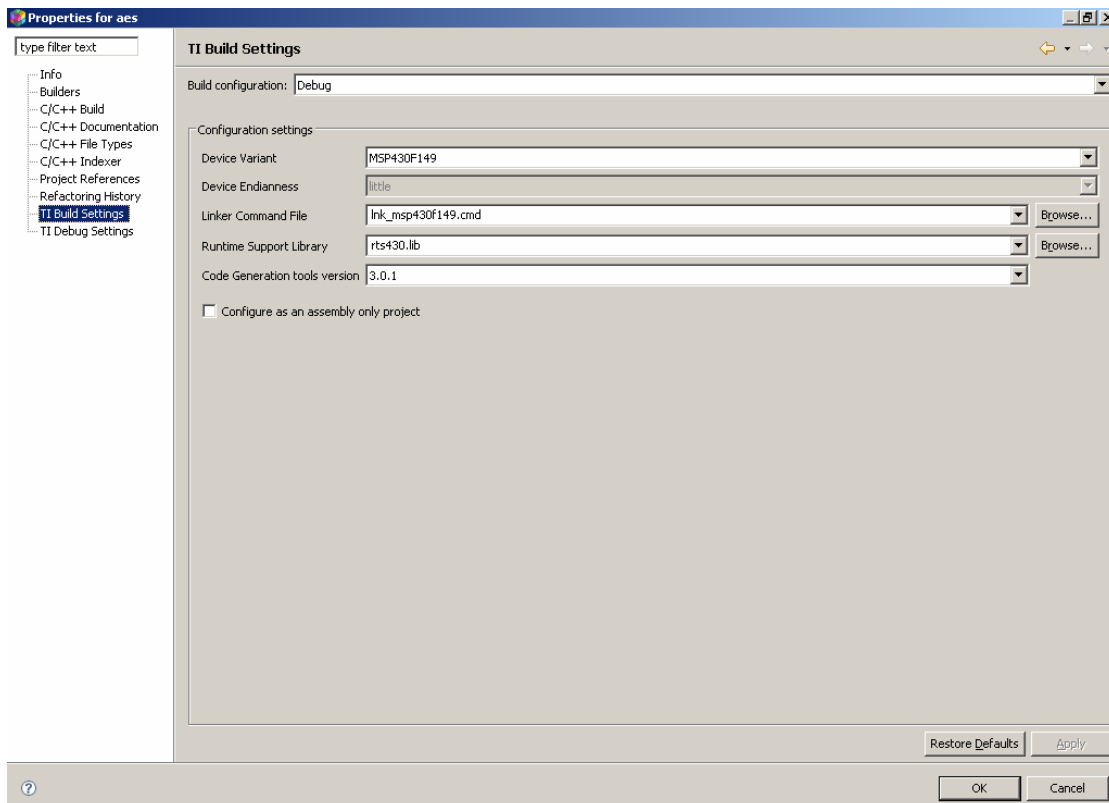
The memory space reserved for the passing of arguments by the C routines is defined in (--arg_size). The space reserved for the dynamic allocation of memory by the program is defined in the option (--heap_size). The system stack size used by the program is set by the option (--stack_size). See *Figure 2-26*.

Figure 2-26. CCE workbench – Memory space configuration.



The device for which the project is intended is configured in the **Properties> TI Building Setting**. The window is in every way identical to that presented in the project’s creation (Figure 2-27).

Figure 2-27. CCE workbench – Device configuration.



The project debugging is carried out as specified in the window **TI Debug Settings**. With the **Setup** tab, using the option **connection**, the method of connecting to the device is established, either parallel port or USB port. The **Debugger** tab can be used to specify whether to load the all application (**Load program**) or just load the project's symbols (**Load symbols only**). These options can be used to choose between loading the entire program, or just the symbols. This last option is valid when the development environment cannot load the software, such as in the case of the software runs in ROM.

Using the **Target** tab, it is possible to define various aspects related to the device. Thus, it is possible to enable the use of IO functions in **Enable CIO functions** use, or establish the starting point for the code execution when a reset occurs or a program is loaded. In the MSP430 properties, it is possible to specify the supply voltage and the types of breakpoints: software or hardware. The memory storage process can also be defined using this tab (Figures 2-28 to 2-30).

Figure 2-28. CCE workbench – Device options configuration.

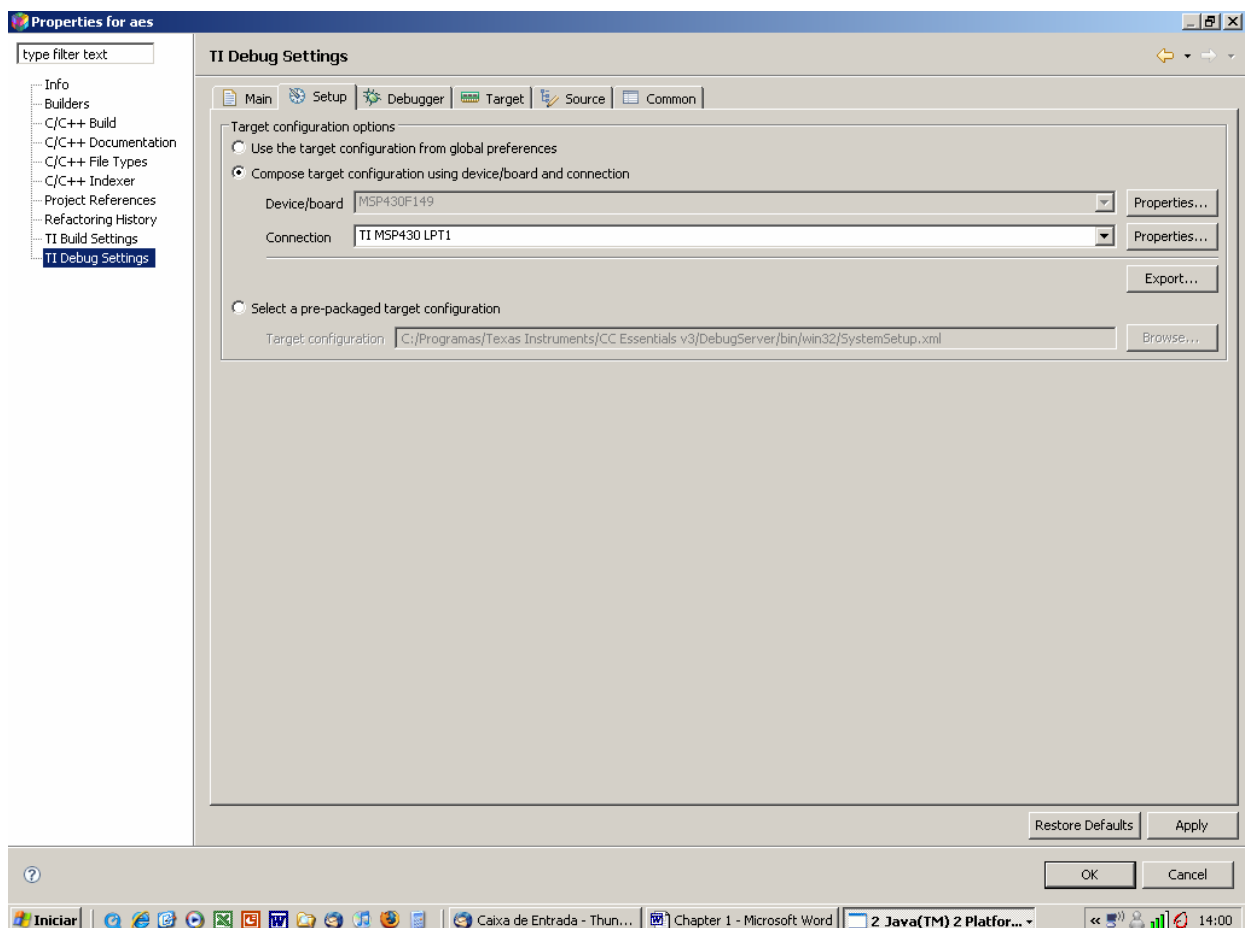


Figure 2-29. CCE workbench – Device options configuration: TI Debug Settings – Target: Generic.

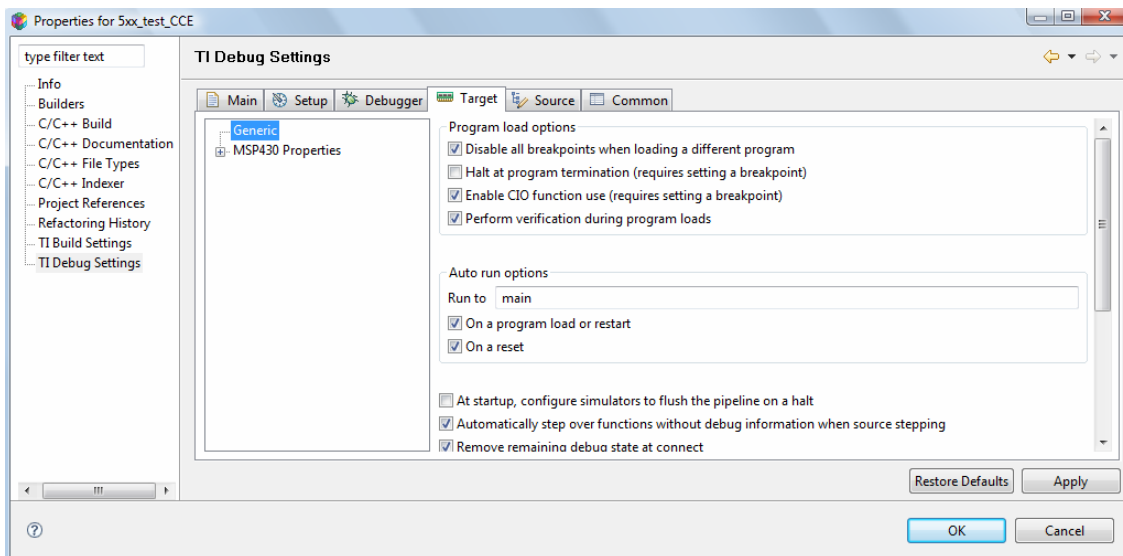
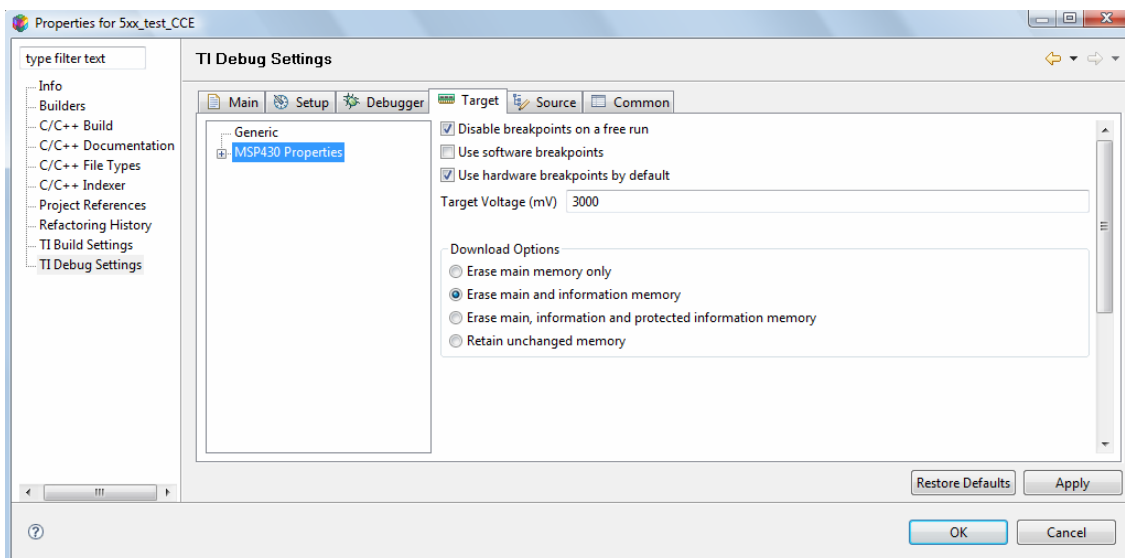


Figure 2-30. CCE workbench – Device options configuration: TI Debug Settings – Target: MSP430 properties.



The first time that the project is built, the **Project > Build All** option must be selected. The project build status can be examined in the **Console** window. If there is a problem, the **Problems** window will list them all. After a successful build of the project, the output file can be automatically loaded into the device.

Alternatively, a project can be built at the beginning of the debug session. The option **Debug Active Project** will recompile the project and launch the debugger, using the device information defined in the project options.

Note that an attempt to update the **firmware** can occur when the debugger is used for the first time, after a software release has been installed or a new USB interface is used.






Finally, the active perspective must be switched to the **Debug** perspective. This operation can be carried out with the perspective selection buttons located on upper right corner of the workspace window, or alternatively, by selecting **Window > Open Perspective > Debug**.







When the project is debugged, the errors are identified on the right hand side of the editor as red marks while the problems are identified as white marks. A mark is added on the left hand side of the editor to the lines that contain an error. When this mark is selected, the compiler provides information about the error.

When the project is made (make option), the resources used can be accessed on **Properties > C/C++ Build > MSP430 Linker V3.0 > Linker Output** in the option **Produce list of input and output sections**.

Introduction to Debug with CCE

TI's debugger generates an output file as a result of building the project. To debug a project after a build, it is only necessary to perform the following steps:

- ❑ Select the project as active, or click **Run > Debug Active Project** or select the icon . The debug perspective is open and it is possible to debug the code;
- ❑ This resets and suspends the execution of code on the device. Running this command, the content of all status registers is modified to the power-up state defaults in accordance with the device specifications. The reset command  is enabled by **> Reset CPU**;
- ❑ To start a program execution, once loaded into the device's memory, select the option **Run > Run (F8)**, or click on the button . Program execution will take place until the program finds a breakpoint;
- ❑ The program execution may be suspended at any time by using the command **Run > Halt**, or by clicking the button ;
- ❑ To re-start the execution of the application use the command **Run > Restart**, or click . This action does not modify the execution stage of the device. It only restores the PC to the application's starting point loaded into memory;
- ❑ The **Set PC to Cursor** feature moves the execution of the application to a particular point in program memory. The execution of this command only modifies the contents of the PC register. No instruction will be executed in order to reach this point. The command can be found in the context menu of the C/C++ perspective in **Set PC to Cursor**;

- ❑ There are several different ways to run the code until a specific point:
 - Use a breakpoint to specify that when this point is reached the program execution must be halted;
 - Use the command **Run > Run to Line**, or click on , available in the context menu of C/C++ perspective, to run the code until the specified location;
 - A special case is to run the code until the main function is reached . This feature enables a temporary breakpoint at the beginning of the main routine and starts the execution of the application. The breakpoint is removed and execution is suspended once the location is reached. This command provides a convenient method of starting C applications.
- ❑ The stepping commands execute each instruction step-by-step. When a function is called, it is possible to move the execution to the function (step into) or perform the function and pass to the following instruction (step over). Once inside a function, the user can continue to execute each instruction individually, or run the rest of the function code until it ends (step out);
- ❑ The execution of the next instruction is performed through **Run > Step Into** (F5), or by clicking the icon . The next instruction is executed when this command is used. If the next statement is a function call, the debugger passes the execution to the first instruction of the function, and suspends execution at this point;
- ❑ When the execution is on top of a function call, the step over operation can be enabled by selecting the **Run > Step Over** (F6) or by clicking the icon . The debugger then performs the function and then suspends execution when it returns. If it finds a breakpoint somewhere in the function, the execution may be suspended at this point. If the **Step Over** is executed on an instruction that is not a function call, the debugger response will be the same as **Step Into** command;
- ❑ If the application is being executed inside a function in response to a function call, it is possible to force the return of this function through the command **Run > Step Return** (F7), or by clicking the button . The debugger will execute the rest of the function code and return the calling point. The execution will be suspended at this point;
- ❑ The command **Terminate**  allows finishing of the application's debugging.

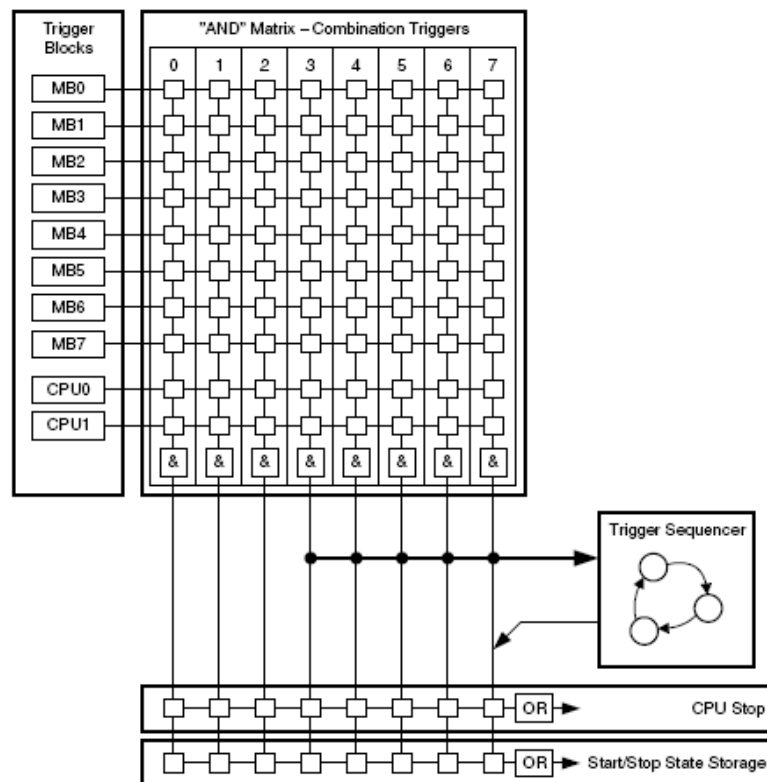
All the MSP430 family of devices have an advanced code debugging module (EEM - Enhanced Emulation Module). This module allows CCE to monitor the device's operation in a non-intrusive way, and without using any resources. Thus, it facilitates the development of the application through the verification of its operation. Depending on the device, the EEM module implementations differ. Generally, the following features are present:

- ❑ Up to 8 hardware breakpoints;
- ❑ Operates in all range of frequencies and clock sources;

- ❑ Ability to set more complex breakpoints through association of triggers;
- ❑ Suspend the execution of the application on the occurrence of a program or data bus access;
- ❑ Access protection to protected data and program memory areas;
- ❑ All timers and counters can be inhibited (depending on the device);
- ❑ Inhibits PWM signals generation on the occurrence of the application's suspension;
- ❑ Allows real-time execution of the applications in the modes: single step, step into; run to cursor; step over;
- ❑ Supports all low power modes.

The *Figure 2-31* represents a simplified block diagram of one of the most complete implementations of EEM module.

Figure 2-31. CCE workbench – EEM module block diagram.



Events within the device can generate triggers. These triggers can be classified as the event that causes them to:

- ❑ Access to addressing and data buses;
- ❑ Access to CPU registers.

Depending on the device, it is possible to associate two or more of these triggers, in order to build complex event detectors that help the detection of incorrect operation of applications. Generally, a

trigger can be used to control the following functional blocks of the EEM: breakpoints, trace, and sequencer. The activation of a trigger is conditioned to an access to the data and program busses or access to CPU registers.

A breakpoint is set through one or more triggers. Through these it is possible to establish the following types of breakpoints:

- Address breakpoint;
- Data breakpoint;
- Register Breakpoint;
- Mask Register;
- Range breakpoint.

A simple breakpoint is defined using a trigger associated with an instruction read operation by the CPU. It is necessary to specify the instruction address where the trigger should occur.

By combining two triggers, it is possible to establish a Data Breakpoint. While one of the triggers is used to detect the occurrence of a particular address on the address bus, the other is used to detect the occurrence of a read or write operation at that address. It is possible to force the suspension of the execution of the application to only occur when there is a match between the value written or read and the one specified.

When the application is written in assembly language, it is sometimes necessary to analyse the accesses to some of the microcontroller's registers. A Register Break Point uses a trigger to detect the access to a register. A Mask Register should be used when the register is composed of several fields, since it can apply a mask and test specific bits only.

An application in certain operating conditions may occasionally try to access to invalid or protected memory regions. Using a range breakpoint, it is possible to detect the occurrence of these events. It is thus possible to suspend the execution of the application on the occurrence of:

- Write to flash;
- Invalid access to memory;
- Access to an instruction in invalid program space;
- Access to data in invalid data space.

The hardware breakpoint properties are established through different fields. The action to make when all triggers are true can be defined in the Action option of the Hardware Configuration field. One of the following options can be chosen:

- Halt;
- Trigger storage;
- Halt and trigger storage.

In the trigger field, specify through various options, the check condition for a true trigger. The trigger can be:

- Memory Address bus;
- Memory Data bus;
- Register Write.

Depending on the type of trigger chosen, the options to specify may be:

- Memory Address Bus:
 - Location: Address of the program code line (e.g.: aes.c, line 30) or data memory address (e.g.: &a);
 - Mask: the information introduced in this field is used in a logic AND operation with the contents;
 - Operator: Logic operation with the data (==, <=, >=, !=);
 - Access: Memory access type:
 - Instruction fetch;
 - Instruction fetch and hold trigger;
 - No instruction fetch;
 - Don't care;
 - No Instruction fetch and read;
 - No instruction fetch and write;
 - Read;
 - Write;
 - No instruction fetch and no DMA access;
 - DMA access (read or write);
 - No DMA access;
 - Write and no DMA access;
 - No instruction fetch and read and no DMA access;
 - Read and no DMA access;
 - Read and DMA access;
 - Write and DMA access.

- ❑ Memory Data Bus:
 - Value: A mask and compare will be applied to the data on the bus and to value added here, to determine if the trigger is true;
 - Mask: The information introduced in this field is used in a logic AND operation with the contents;
 - Operator: Logic operation with the data (`==`, `<=`, `>=`, `!=`);
 - Access: Memory access type (on Memory Address Bus).

- ❑ Miscellaneous:
 - Group: Group to which the breakpoint belongs;
 - Name: Name assigned to the breakpoint.




There is a predefined breakpoint that can be set to:



- ❑ Break in program range: Generates a suspension of the execution of the application in a range of program memory addresses. It uses two triggers that define the range of addresses;
- ❑ Break in DMA transfer: Generates the suspension of the execution of the application, whenever a DMA read or write operation at the specified program address occurs. This breakpoint is implemented using only one trigger;
- ❑ Break in DMA transfer range: Generates the suspension of the execution of the application, whenever a DMA read or write operation at the specified address range occurs. This breakpoint is implemented using two triggers;
- ❑ Break in stack overflow: Generates the suspension of the execution of the application whenever the SP register value assumes a lower value than the specified one. This breakpoint is implemented using only one trigger;
- ❑ Breakpoint: Generates the suspension of the execution of the application whenever the memory bus address takes the value specified. This breakpoint is implemented using only one trigger;
- ❑ Hardware breakpoint: Generates the suspension of the execution of the application whenever the memory bus address takes the value specified. This breakpoint is implemented using only one trigger;
- ❑ Watch on data address range: Generates a suspension of the execution of the application whenever the specified data memory addresses range is accessed. It uses two triggers to define the range of addresses;
- ❑ Watchpoint: Generates the suspension of the execution of the application whenever a specified data memory address is accessed. It uses a trigger to generate the watchpoint;
- ❑ Watchpoint with data: Generates the suspension of the execution of the application whenever a specified data memory address is accessed and the value of the address is equal to specified value. Two triggers are used to implement this watch.

In order to verify the code execution, it is necessary to use support tools to complete this task. CCE provides a set of features with this aim.

A breakpoint suspends the execution of the application in order to check the status of the system. The activation, deactivation and configuration of these breakpoints are possible through CCE.

There are two types of breakpoints: software and hardware. While the first type of breakpoint is implemented through the insertion of code in the application, in a way invisible to the user, the second type is implemented internally by the device's hardware. Although the software breakpoints are not limited, the hardware breakpoints, depending on the device, have a limit of 2 to 4 breakpoints.

Thus, an active breakpoint is identified by the symbol  in the sidebar of the code window. The symbol  will identify a hardware breakpoint. A disabled breakpoint will be identified by the symbol . Note that the CCE will by preference use hardware breakpoints and then software breakpoints.

Placing the cursor at a particular code line and then using the command  will activate a breakpoint. The command  enables or disables a breakpoint at the cursor's location. An alternative way is to use the context menu with the option **toggle breakpoint**.

The application debugging process often requires access to the actual values of the variables. The **Variable** view allows the user to monitor the application's local and global variables. In this view, the CCE automatically displays the name and contents of the local variables of the function that is being executed. It is also possible to add the name of other local variables or global variables to be monitored in the debugging process.

The values of the local variables can be modified. The values of the variables that have been changed during the last instruction execution are displayed in red. However, the variable names cannot be modified. It is allowed to change the representation format of the variable: Natural, decimal or hexadecimal. The variables that contain more than one element, such as arrays, structures, or pointers are presented with a (+) sign immediately after the name. This signal means that the variable has elements that can be seen through the expansion of the (+) sign, passing this signal (-), which allows the structure to be collapsed.

The local variables cannot be added or removed from the Variables view. However, global variables can be added or removed. The local variables can be disabled in order to freeze their value as the program is executed.

The **Expressions** view accepts the entry of expressions to evaluate them as the program is executed. These expressions are written in syntax similar to that used by the C programming language.

The commands accessible through the context menu can:

- Specify the number of elements of the array to be displayed in the Expression view: The command **Display as Array** can be used to display the elements of any pointer or array. The command **Remove Array Expansion** is used to return an expanded variable back to its original state;
- Change value: Changes the content of the variable;
- Cast to type: Performs a promotion (cast) for a different type of variable;
- Restore Original Type: Restores the expression for the original data type.

The **Memory** window of the Debug perspective allows the user to monitor and modify the device's memory. The memory is provided with a list of **Memory Monitors**. Each monitor represents a section of memory specified by its named location base address. Each memory monitor may be represented in different data formats (memory renderings). The debugger allows four different types of rendering:

- Hex (default);
- Ascii;
- Integer signed;
- Unsigned integer.

The Memory view has two panels:

- Memory Monitors;
- Memory Renderings.

The first panel displays the memory monitors list added to the debug session. The second panel is controlled by selection in the first one and consists of tabs that display the rendering. This panel can be configured to display both renderings.

Expanding CCE capabilities with Eclipse JDT and PDE

The CCE can be expanded beyond its default capabilities. There is a set of Eclipse plug-ins that can be used. Features can be installed to allow Java programming and the development of Eclipse plug-ins. These updates should be made only where these features are required. Alternatively, CCE and Eclipse can always co-exist in the same system, one CCE installation and other from Eclipse.

The CCE is based on the Eclipse's release 3.2. Hence, all the plug-ins that are installed must be compatible with this version. The installation of plug-ins is extremely easy. Just unpack the releases to the correct directories.

2.2.3 Laboratory 1: "Hello World" Beginner's project

The following laboratories provide an overview of the features of CCE.

Lab1.1 – Introduction to the application debug

Project files

□ C source files: **Chapter 2 > Lab1_CCE > Lab1a.c**

Overview

This laboratory shows how to use the basic features of CCE to allow applications to be built and debugged. All the steps needed to create a project, including its configuration, as described above, are illustrated using an example. The application is downloaded to the device after the project has been successfully compiled and built. Several debugging techniques are used, such as: step-by-step execution, analysis of the contents of local and global variables, resetting the device, etc.

Step 1: Creation of the project

A project is to be developed that sends a set of numbers corresponding to the Fibonacci series to the CCE console. The Fibonacci series can be recursively defined by the expression:

$$F(n) = \begin{cases} 0, & \text{if } n=0; \\ 1, & \text{if } n=1; \\ F(n-1) + F(n-2) & \text{other cases} \end{cases}$$

This sequence was originally used by Leonardo of Pisa (also known as Fibonacci, circa 1200 A.D.) to describe the growth of a rabbit population. With this formula, the sequence of Fibonacci numbers can be determined and answer to the question "how many rabbits were born in the sixth month?", in which the correct answer is, "in the sixth month 8 rabbits were born."

The algorithm that solves this problem is simple to develop. Beginning with the first two values of the sequence, the remaining values are successively calculated.

The sequence of actions to build the project is described below.

❑ A. Creating the project

- Create a new project in **Project > New Managed C/ASM Project**;
- In project name write **Project1**;
 - Accept the default settings in **Select a type of project**;
- There should not be any dependency on other existing projects;
- In **Device Selection Page**, choose in the option **Device Variant**, the device **MSP430FG4618**;
- Automatically, the CCE will select the appropriate debug file (lnk_msp430fg4618.cmd) and the support library (rts430x.lib);
- Complete the creation of the project by clicking the button **finish**.

❑ B. Add a source code file

- The project created is visible in the **C/C++ Projects** window using the **C/C++** perspective;
- Add a file to the project where the source code will be written, by selecting **File > New > Source File**;
- The file created should be named **Lab1a.c**;
- Write the code below that solves the problem:

```
//*****
// Basic debug introduction using CCE. Application conditional execution
//*****
#include <msp430xG46x.h>
#include <stdio.h>
//*****
// Global data
//*****
unsigned int a, b, i;
//*****
// Main routine
//*****
void main (void)
{
// Stop watch dog
WDTCTL = WDTPW + WDTHOLD;           // Stop WDT
```

```

// Global data initialization
a = 0;
b = 1;

// First message to CIO
printf("Lab 1 - Introduction to Debug with CCE V3\n");
printf("Fibonacci sequence computation\n");
printf("Number n = 0 - %d\n", a);
printf("Number n = 1 - %d\n", b);

for(i = 0; i < 7; i++){
    int c;

    c = a + b;
    a = b;
    b = c;
    printf("Number n = %d - %d\n", i, c);
}
}

```

- ❑ The code starts with a descriptive header of the contents of the file and other essential aspects, such as: author, date and version. Other contents may be added;
- ❑ Next, the file `msp430xG46xx.h` is included. It contains the definitions necessary for programming the device;
- ❑ The global variables ***a***, ***b*** are defined as unsigned integer, and are used to store the Fibonacci numbers of order *N* and *N*-1, respectively. These variables are initialized with the values {0, 1}, respectively;
- ❑ The control variable ***i*** is used to control the number of iterations used in the sequence calculation;
- ❑ The first message is sent to the console using the instruction ***printf***;
- ❑ The message consists of a sequence of text lines, so it uses the escape character ***\n***, causing a change of line in the console;
- ❑ Within the code structure, the numbers of the Fibonacci sequence are calculated successively using the above mathematical expression;
- ❑ Finally, the code ends with the directive ***_NOP()***, corresponding to the execution of an operation, which has no effect other than to take execution time.

Step 2. Configuration

The project must be configured after writing the code.

- ❑ Go to **Project > Properties** and select the field **MSP430 Linker 3.0 Runtime Environment** in the option **C/C++ Build**;
- ❑ At this location, define the size of the stack using the directive **Specify heap Size for C/C++ dynamic memory allocation** (***--heap_size***) with the value 400 and in **set C system stack size** (***--stack_size***) with the value 200;

- ❑ Since the *printf* library function is used, the type of implementation in use must be specified;
- ❑ In **MSP430 Compiler V3.0** on the option **Library Function Assumption** choose to *full*;
- ❑ Finally, confirm in the **Run Time Model Options** if the silicon version (*--silicon_version*) is defined as **msp**, because the application runs on an extended memory device.

In the **TI Debug Settings** field, specify the connection type between the PC and hardware:

- ❑ Choose the link **TI MSP430 USB1**;
- ❑ At **Debugger** tab confirm the selection of options **Connect to exact CPU** and **load Program** in **loading options**;
- ❑ At **Target** tab ensure through **Program load options** that the use of the IO console CIO is active in **Enable CIO Function Use** and that the program written to the device is verified using **Perform Verification During Program Load**;
- ❑ Ensure that in **Auto run options**, the program begins to run in the **main** routine. Select the two options **On the Load or Restart Program** and **On reset**.

Step 3. Compilation



Once the configuration has been set up, the project must be compiled through the option **Project > Build Active Project**.

Step 4. Project debug

To debug the code it is necessary to download it to the device through the function **Run > Debug Active Project**.

The CCE will automatically switch to the **Debug** perspective;



The following sequence of tasks will allow verification of the operation of the application, while utilizing the CCE features for code debugging.

- ❑ **A. Observe the contents of variables and expressions**
 - In the **Variables** window press the icon  and mark the global variables *a*, *b*, and *i* to highlight them;
 - The contents of these variables can be monitored immediately;
 - As the program has not yet begun to run, the values not yet valid;
 - In the **Expressions** window, add an expression that the debug will automatically calculate through the debugging process. For example, using the context menu option , the

global variables addresses can be monitored through the expressions **&a**, **&b** and **&i**;

- Request the number of Fibonacci numbers remaining to calculate using the expression **7-i**.

□ B. Run the code step-by-step

- Execute the application step-by-step;
- By pressing the key **F6**, the first instruction is executed;
- The program's status indicator will indicate the next instruction to be executed and will stop the watchdog timer;
- Note that after the execution of the lines that changes the values of variables **a** and **b**, the **Variables** window displays the new values.
- Observe messages being sent to the CIO console;
- Display the CIO console pressing the icon  , choosing Console 3;
- To ensure that it remains visible, press the icon .
- Later, as the application runs step-by-step (F6), the messages will appear in the CIO console:

```
Lab 1 - Introduction to Debug with CCE V3
Fibonacci sequence computation
Number n = 0 - 0
Number n = 1 - 1
```

- At the end of this sequence of instructions, the variables should have the following values of a = 0, b = 1.

□ C. Run the application until a specific code line

- Put the cursor on line of code 48, corresponding to: `printf("Number n = %d - %d\n", i, c)`
- Select the command **Run to line** execution, to lead the program execution status indicator to that line;
- The sequence of instructions that led up to this point has been executed, modifying the program variables.
- The local variable **c** was automatically added to the **Variables** window.

- ❑ **D. Restart the debugging task**
 - Once the application has finished running, or whenever it is desired to repeat the code debug, it is possible to restart the device;
 - Choose the **reset CPU** option in **Run > Reset**;
 - Note that the memory of the processor remains unchanged;
 - Re-run the application. Place the cursor at the code line 36 and order it to run the program until the cursor position;
 - Using the **Variables** window, change the contents of variables **a** and **b** to **3** and **4**, respectively;
 - Execute the application until line of code 52;
 - Observe the new sequence of values for the Fibonacci numbers.

- ❑ **E. Include the project in the workspace**
 - The project built during this laboratory can be found in code folder **Lab1**. To include it in the workspace, choose the project through **Project > Open Existing Project**;
 - Perform this laboratory starting at point B.

Lab1.2 – Using breakpoint to save and load data to and from file

Project files

- ❑ C source files: **Chapter 2 > Lab1_CCE > Lab1a.c**

Overview

The debugging of an application can benefit from the ability to exchange information with files on disk. This laboratory will exploit this capability using the project developed in Lab1. Now, the two initial values required to determine the sequence of Fibonacci numbers are collected from a data file. The numbers are successively calculated and are also stored in a data file.

Step 1. Include the project in the workspace

This task is identical to that carried out Step E of Lab 1.1. Compile the laboratory with the file **Lab1a.c**.

Step 2. Creating input and output data files

The code debugger can access binary files in COFF (Common Object File Format) format or files in text format. The latter format is used in this laboratory. This file has a header line followed by several lines with one value per line. The data can be stored in the following formats: Hexadecimal, integer, long, float. The information contained in the header line has the following syntax:

```
MagicNumber Format InitialAddress PageNumber Length
```

where:

- ❑ **MagicNumber:** constant value equal to 1651;
- ❑ **Format:** Value between 1 and 4, indicating the format used in file samples;
- ❑ **InitialAddress:** Address of the start of the data block;
- ❑ **PageNumber:** Page number from which the data block was obtained;
- ❑ **Length:** Number of samples in the block.

All the numbers represented in the header are in hexadecimal format.

❑ A. Create the data files

- The CCE can create this file through **File > New > File**;
- Include the name *DataIn_a.txt* on file dialog box to create the file containing the data to be allocated to the variable **a**:

```
1651 1 135c 0 2
2
```

- Repeat the process for the data file of variable **b**:

```
1651 1 135c 0 2
3
```

- Note that CCE expects that the values read are represented using 5 values, each with 4 digits. If the data values are represented in hexadecimal, then CCE expects the first digit to be zero.

❑ B. Associate the input data files to the project

- The project built during this laboratory can be found in the code folder **Lab1** and can be added to the project using two different procedures:

- Copy the file to the project directory: This solution has the disadvantage of copying the file to the directory of the project. It loses any freedom to control the file contents if it is used in several projects because there can be multiple sources of that file;
 - Link the file to the project: This option allows the file to be stored in an appropriate location and connect it with the project without copying it to the project directory. The same file can be used on different projects, and a change to its content will be observed by all projects that use it.
- In this example, the files are copied into the project directory.

Step 3. Add breakpoints to the application and associate them with files

The easiest way to set a breakpoint is through the **Breakpoint** window. This window allows the choice of action, including the reading or writing data on file. Choosing one of these actions allows the linking of a file to the breakpoint.

□ A. Activate the Breakpoint window

- If the breakpoint window is not already open, do so in **Window > Show View > Breakpoints**.

□ B. Create Breakpoints

- Create two breakpoints, one at line of code 36 and another at line of code 37;
- Click on the line of code and add the breakpoint using the left mouse button, in order to provide access to the C editor content menu;
- In this menu choose the option **Toggle Breakpoint**.

□ C. Setting the Breakpoint

- In **Breakpoint** window, after selecting the line of code 28 breakpoint, edit its properties;
- Choose the option **Action** and open the list of options to choose **Read Data from File**.

□ D. Filling the Breakpoint options

- It is required to fill out the following fields:
 - File: location and file name from where the data will be read;
 - Wrap around: mark this selection to start reading at the beginning of the file once it reaches the end;
 - Start Address: Location to send the data read from the file. This address can be changed, since it is always re-evaluated at the beginning of each read;
 - Length: The length of memory. This parameter can also be modified by the debug process.

- To set the Breakpoint associated with the line of code 36, configure:
 - File: DataIn_a.txt
 - Wrap around: Yes
 - Start Address: &a
 - Length: 1
 - Name: Breakpoint Load a

- To set the Breakpoint associated with the line of code 37, configure:
 - File: DataIn_b.txt
 - Wrap around: Yes
 - Start Address: &b
 - Length: 1
 - Name: Breakpoint Load b

- For the file used to write the values of variable **c** (Fibonacci series of numbers), create a breakpoint to associate the file to line of code 48;

- Follow the same steps for setting the previous breakpoint, edit its properties and choose **Write Data to File** in the option **Action**;

- The data values to configure are:
 - File: DataOut_c.txt
 - Format: Hex
 - Start Address: &b
 - Length: 1
 - Name: Breakpoint Write c

- Finally, add a breakpoint at line 52 named Final Point, to suspend the execution of the application at the end of the calculation of the Fibonacci numbers.

Step 4. Save file and load files with the breakpoint information

The breakpoints created in this laboratory can be written and read. This feature is useful in order to reuse the breakpoint configuration.

A. Export the breakpoints to file





- After creating and setting up the breakpoints, as described in the previous steps, the breakpoint information can be saved by exporting it to a file;
- Go to **File > Export**;
- In the **General** item choose the option **Breakpoints**;
- All breakpoints defined in debugger will be present. Select them all;
- Specify the file name to make, name it **Lab1_breakpoint.bkpt**.


B. Import breakpoints from the file

- To import the breakpoint use the **Import** feature;
- Indicate the file name and select the two options to automatically create and update the breakpoint imported.

Step 5. Code debugging:

The data file exchange operation can be verified during the debug process. To debug the code, perform the following tasks:

- Verify that the execution state indicator points to line of code 27. If this does not happen, order a **restart**  of the device;
- Verify that the CIO console is clean and visible. If these conditions are not fulfilled, using the context menu, order its cleaning  and activate the icon  to turn the console to always visible;
- Execute the application step-by-step  until the variables **a** and **b** are initialized. Verify that in the **Variables** window that they take the values 0 and 1, respectively;
- When the code line 36 is pointed to (before being executed), the variable **a** takes the value 2, read from the respective data file. The same thing will happen when it reaches code line 29, since when it is pointed to, it leads to the execution of **breakpoint Load B**, resulting in reading the value 3 for the variable **b**;

- ❑ The Fibonacci number calculation process, using these initial conditions, is initiated within the computing cycle. Whenever the code line is reached, the **breakpoint Save c** runs. Successive data values are stored in the data file associated with the breakpoint. Order the execution of the application using the command **run** . The execution takes place until the **breakpoint Final Point** is reached. In this execution state, the application already determined and stored all the Fibonacci numbers in the file;
- ❑ Switching to the **C/C++** perspective, in the **C/C++ Projects** window, it is now possible to see that the data file **DataOut_c** has been created. Edit this file to see the results. Note that the calculation process returned the sequence: 2, 3, 5, 8, 13, 21, 34, 55, 89.

Lab1.3 - Advanced breakpoints with triggers

Project files

- ❑ C source files: **Chapter 2 > Lab1_CCE > Lab1b.c**



Overview


The use of breakpoints has been demonstrated in the previous laboratory, either to run the code until a certain point or to identify access to the data in the files, under specific conditions. The correct use of these features assists the debugging of complex applications. This laboratory shows how breakpoints can be set, with the appropriate use of triggers. We will continue to use the previous example, because only a small change will be introduced. The breakpoints will be configured to be active under special conditions.

Step 1: Preliminary conditions

The change introduced in the source code of this project is the addition of a data array to store the calculated results. The file **Lab1b.c** incorporates this modification. Compile the project with this new source file, and debug it.

There is a wide range of available triggers that can generate an appropriate response by the debugger. To understand the logic associated with the configuration of these features, let us perform some actions for that purpose.

- ❑ Remove all existing breakpoints by selecting the icon  in the **Breakpoint** window.
- ❑ **A. Read access to a variable**
 - Set up a new breakpoint using the icon  in the **Breakpoint** window;

- Chose the option **watchpoint**;
 - Introduce the following configuration:
 - Location: &a
 - Access type: Read
 - Execute the application with **Run**;
 - Note that at the first access reading of the variable **a**, the application execution is suspended.
- **B. Access to a variable using a condition**
- Specify and provide that the suspension should only occur if a particular value is read;
 - Remove the last breakpoint and implement the following actions:
 - Set up a new breakpoint using the icon  in the **Breakpoint** window;
 - Chose the option **watchpoint with Data**;
 - Introduce the following configuration:
 - Location: &a
 - Data value: 5
 - Access type: Read
 - Execute the application with **Run**;
 - Note that at the first access, the variable **a** is equal to 5 and the application execution is suspended.

Lab1.4 - Memory and usage Register

Project files

- C source files: **Chapter 2 > Lab1_CCE > Lab1c.c**

Overview

The assessment of memory resources used by the application is crucial to the development of applications. This laboratory illustrates how to access to the device's memory and registers. The use of the system stack by C/C++ is described. The verification of this process is done through an example. For this, the translation from the C/C++ source code to assembly language is observed and some of the tools used to analyze programs in assembly language are discussed briefly.

Step 1. Using the stack with C/C++

The C/C++ compiler uses a stack of data in memory to:

- ❑ Store local variables;
- ❑ Pass necessary arguments to execute a task,
- ❑ Safeguard the register contents for subsequent replacement.

The system stack grows towards the lower value addresses.

Its management is done using the device's register R1 (commonly referred as the Stack Pointer - SP), which is used to point to the available memory address.

The size of the system stack is specified in the process of building the application, during which the overall symbol **_stack_size** is created and given the value (in bytes) of the stack size. The default value is 128 bytes. This value can be modified during the project build by using the option **_stack_size**.

As for MSP430X device, the size of registers is greater, therefore saving and restoring their contents requires the use of a 32-bit stack (2 words) operation, for each register to be saved. To reuse code originally written for 16-bit devices, it is necessary to increase the system stack size.

At the application's boot time, the SP points to the address at the top of the system stack. This address is the first location after the end of the section reserved for the system stack. This area is defined only at the time of building the application. Before executing a function, the C/C++ automatically decreases the SP to reserve the space needed for the return address and local variables. The SP is increased when it returns from the function, the system stack being restored to a state similar to the one it had before the function executed.

By convention, some registers are associated with specific operations in the C/C++ environment. To interface applications written in C/C++ with code written in assembly language, it is necessary to have a thorough understanding of how the registers are used. A convention stipulates how the registers are used and how their contents are preserved during functions calls. *Table 2-3* shows the registers are affected.

Table 2-3. Registers description.

Register type	Description
Argument register	Passes arguments during a function call
Return register	Holds the return value from a function call
Expression register	Holds a value
Argument pointer	Used as a base value from which a function's parameters (incoming arguments) are accessed
Stack pointer	Holds the address of the top of the software stack
Program counter	Contains the current address of code being executed

Table 2-4 shows the use of each register by the debugger. Also described are how the values are preserved during function calls.

Table 2-4. Register usage and preservation.

Register	Alias	Usage	Preserved by function ⁽¹⁾
R0	PC	Program counter	N/A
R1	SP	Stack pointer	N/A ⁽²⁾
R2	SR	Status register	N/A
R3		Constant generator	N/A
R4 – R10		Expression register	Child
R11		Expression register	Parent
R12		Expression register, Argument pointer, return register	Parent
R13		Expression register, Argument pointer, return register	Parent
R14		Expression register, Argument pointer	Parent
R15		Expression register, Argument pointer	Parent

⁽¹⁾ The parent function refers to function making the function call. The child function refers to function being called.

⁽²⁾ The SP is preserved by the convention that everything pushed on the stack is popped before returning.

Any function (Parent function) that calls or is called by another function (Child function) must follow a set of rules. Any violation of these rules may destabilise the C/C++ environment and cause the application to fail.

The terminology used in the description of how a function call is processed by the C/C++ is:

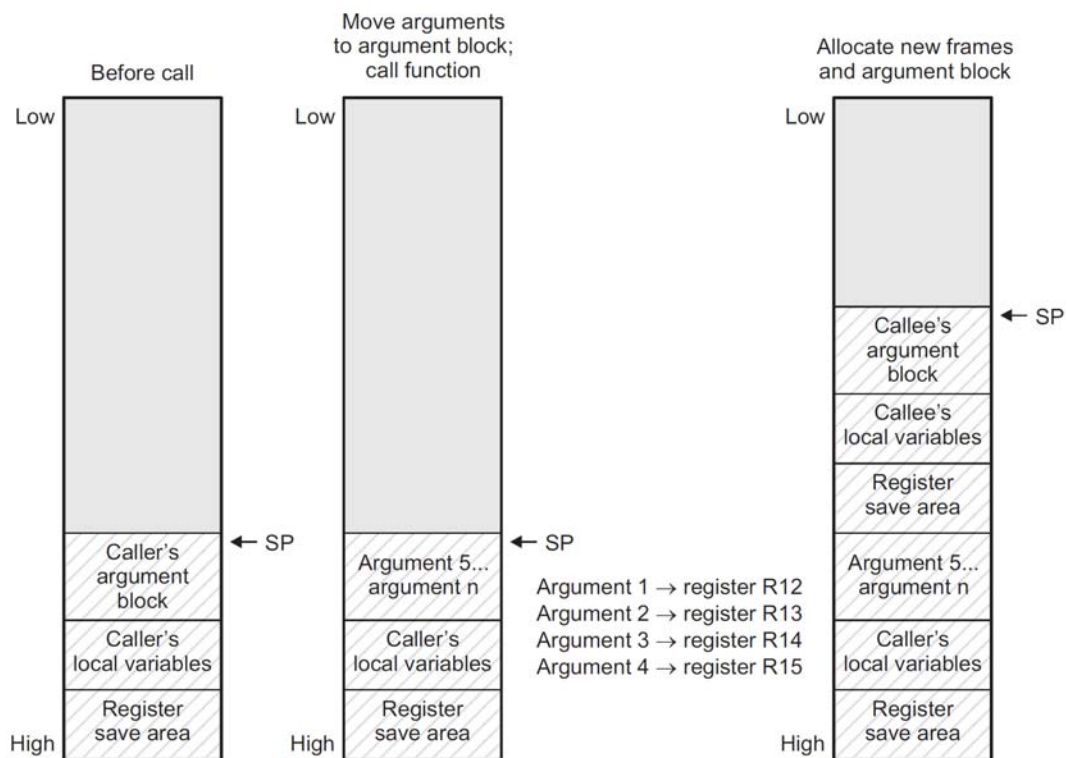
❑ **Argument block:** The part of the local frame used to pass arguments to other functions. The arguments are passed to the functions by moving them to this data block, instead of placing them on the system stack. A local frame and the Argument block are reserved simultaneously;

❑ **Register save area:** Part of the local frame used to store the register's contents when the application invokes a function, allowing it to be restored on return from the call;

- ❑ **Save-on-call register:** Registers R11-R15. The function invoked does not preserve the values of these registers, so the function that performs the invocation must save their contents and restore them, if necessary;
- ❑ **Save-on-entry registers:** Registers R4-R10. It is the responsibility of the function to preserve the values of these registers. If the function invoked modifies the contents of these registers, its contents need to be stored and restored when the function returns.

Figure 2-32 illustrates a typical function invocation. Arguments to the function are provided, and it uses local variables. It invokes, in turn, another function. The first four arguments are passed to the function using the registers R12-R15. This example also shows the space reserved for a local frame and an argument block used to invoke another function. Functions that do not have local variables and therefore do not require an argument block, do not reserve space for the local frame.

Figure 2-32. Typical function invocation.



Legend: SP: stack pointer

A function (known as the parent) performs the following tasks when it invokes another function (referred to as the child). The parent function is responsible for the preservation of any save-on-call registers used by the child function. If the child function returns a structure, the parent function reserves space for this structure and passes the address of that space to the child function as the first argument. The parent function locates the argument to pass to the child function in register R12-R15, in that order. The remaining

arguments are placed in the argument block, in reverse order. The child function begins to execute.

Before starting to execute the task assigned to it, the child function should take the following actions.

If the child function is to be invoked with a variable number of arguments, the parent function locates the arguments on the stack if the following criteria are met:

- ❑ The argument includes or follows the last argument declared;
- ❑ The argument is passed in a register.

The child function locates the contents of all registers that are modified by it and that must be restored immediately after its return. Usually, these are the save-on-entry registers (R4-R10).

If the function services an interrupt, there may be additional requirements to preserve the contents of other registers. The functions reserve space for local variables and for the argument block by performing a subtraction from the SP. This constant is determined from the sum of the size of all local variables and the maximum value that is necessary to reserve storage for all the parameters in each call that this function. Then, the child function executes the code associated with it.

If this function returns a value, it is placed in R12 (or R12 and R13). If the function returns a structure, it will be copied to the memory block pointed by the argument R12.

If the function does not use the value returned, R12 is reset to 0x00h. The child function frees space for the local frame and forms the argument block by adding the constant determined previously. Then follows the restore of all saved registers by the child function. The child function returns to the calling point.

This brief introduction serves to illustrate an advanced analysis of the application developed. In the following items, the tasks for the process described above are analysed using the tools provided by CCE.

Step 2. Include the project in the workspace

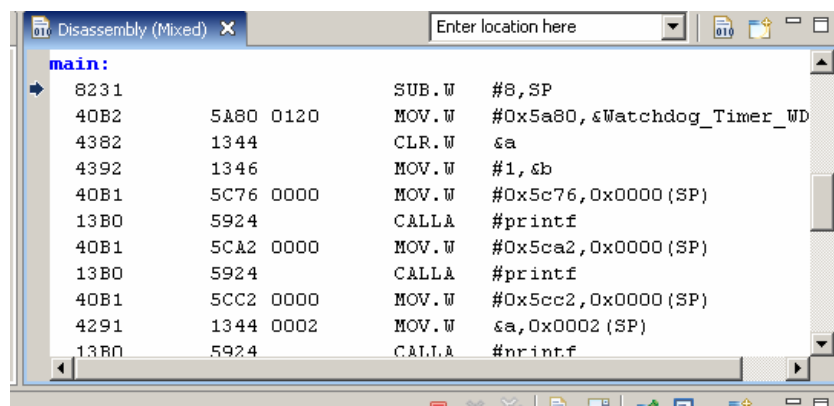
Start the task by including the file ***Lab1c.c*** in the compilation process. Alternatively, the project may be created from the beginning, following the steps already described in previous laboratories. The source code may be imported or written from scratch to a new file.


The task performed by this application is similar to that performed earlier. In this case, the determination of the new number of the Fibonacci sequence is performed by a function. This function receives the addresses of the variables *a* and *b*, and returns the result of the calculation.

Step 3. Program and assembly observation

Once the compiling process is started, CCE changes to the Debug perspective. It is possible to view the assembly code in the Disassembly window resulting from the compilation of the application code written in C. This is shown in *Figure 2-33*.

Figure 2-33. Disassembly window.



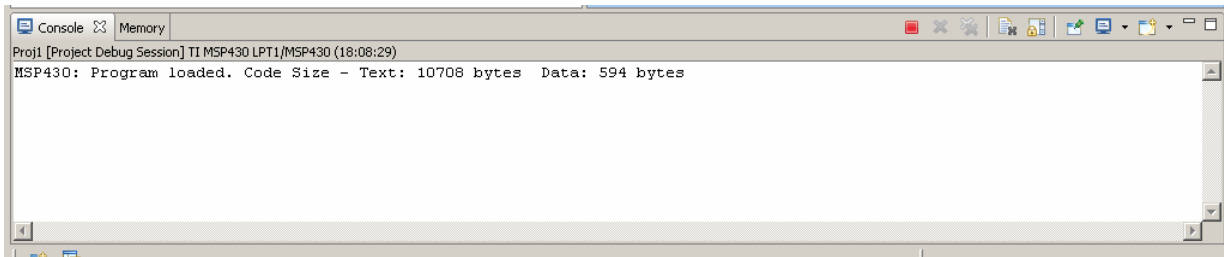
This window can display the contents in two ways. Through the icon  it is possible to switch between them. One shows the assembly code, while the other interleaves assembly code lines with the C source code line that generates them.

For each assembly instruction, the window displays the disassembly instruction, the memory address, and the corresponding opcode. To produce this list of instructions, the debugger collects the opcodes in the device's memory, disassembles them, then adds the available symbolic information. The next instruction to be executed is identified by a mark on the left hand side of the window, similar to the window that displays the C code. It is possible to run the application using the same set of features previously described for C/C++ debugging.

Step 4. Analysis of the device's memory

The contents of the device's memory are accessible through the **Memory** window. It is recommended to move this window to the same console windows group. Drag the window and drop it in the Console group window to obtain to the windows arrangement shown in *Figure 2-34*.

Figure 2-34. Window arrangement.



The **Memory** window helps the task to monitor and modify the device's memory contents. This window is shown with a **Memory Monitors** list. Each monitor is a memory section specified by its location and called as a base address. Each monitor can be displayed in different formats.

The Memory window has two panels:

- Memory Monitors panel: Collects the memory monitors added during the debug session;
- Memory Renderings panel: The content of this panel is controlled by the selection made in Memory Monitors panel and lists the contents of the desired memory address.

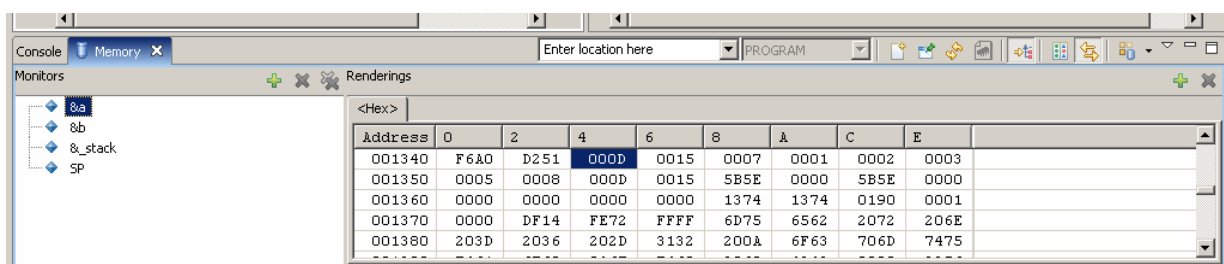
Both panels can be viewed simultaneously by selecting the icon

Add memory addresses to observe some of the application's variables. Use the button to add the address to be monitored. Examine:

- Global Variables *a* and *b*: Insert *&a* and *&b*
- Top C system stack: Insert *&_stack*
- Address pointed by SP: Insert *SP*

The result of this task is similar to that shown in *Figure 2-35*.

Figure 2-35. Memory window example.



Use this feature to determine the position of the variables. From the project compilation results:

- Address 0x01344 is used for the variable **a**;
- Address 0x01346 is used for variable **b**.

Figure 2-36. Global variable addresses in disassembly window.


```

Disassembly (Mixed) x
Enter location here
c = addData(&a,&b);
C$DW$main$2$B, C$1:
403C 1344      MOV.W  #0x1344, R12
403D 1346      MOV.W  #0x1346, R13
13B0 509E      CALLA  #addData
4C81 0006      MOV.W  R12, 0x0006(SP)
array[i] = c;
421F 1348      MOV.W  &i, R15
025F      RLAM.W #1, R15
419F 0006 134A  MOV.W  0x0006(SP), 0x134a(R15)
printf("Number n = %d - %d\n", i, c);
40B1 5CEA 0000  MOV.W  #0x5cea, 0x0000(SP)
4291 1348 0002  MOV.W  &i, 0x0002(SP)
4191 0006 0004  MOV.W  0x0006(SP), 0x0004(SP)
13B0 5924      CALLA  #printf
for(i = 0; i < 7; i++){
5392 1348      INC.W  &i
90B2 0007 1348  CMP.W  #0x0007, &i
2BE1      JLO   (C$11)
_NOP();
C$12, C$DW$main$2$E:
4303      NOP
}
5231      ADD.W  #8, SP
0110      RETA
<source line is not available>
setvbuf:
143A      PUSHM.A #4, R10
0FC9      MOVA  R15, R9
0EC7      MOVA  R14, R7
0DC8      MOVA  R13, R8
0CCA      MOVA  R12, R10
433F      MOV.W  #-1, R15
9A2F      CMP.W  @R10, R15

```

Step 5. Interface to a function

During the introduction to this laboratory, there was a brief description of the procedure used by the system to pass and receive data from a function. Use the CCE supported debugging tools to see an example of this process.

- A. Reinitiate the debug process**
 - The first task will be to ensure that the application is ready to be launched;
 - The starting point is the **main** position;
 - Proceed with device restart through the icon .

❑ **B. Preparation to call a function**


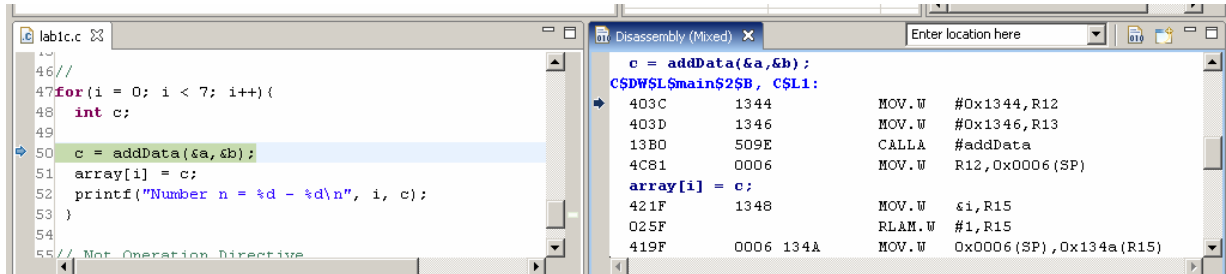
- Put the cursor at line 50 and then execute the application until this line is reached, using the feature **run to line** ;
- In the **Disassembly** window, the instructions before the **addData** function execution call are shown in *Figure 2-37*.

Figure 2-37. addData function example.

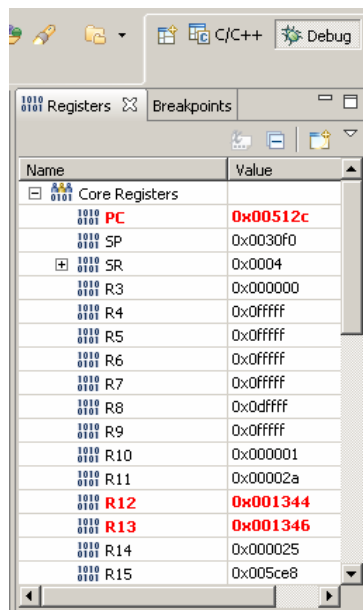


- The address of the variable **a** is 0x1344 while the address of variable **b** is 0x1346;
- The compiler begins by loading the addresses of these variables into registers R12 and R13 respectively, with the instructions:

```
MOV.W    #0x1344, R12
MOV.W    #0x1346, R13
```

- The execution of these instructions can be verified in the **Registers** window;
- The system stack address pointed to by SP is 0x0030F0;
- Since the system stack begins at 0x03038, it already supports 180 bytes of data.

Figure 2-38. Register addresses.



- Using the instruction:

```
CALLA #addData
```

- The function is then invoked. Following the call, the processor automatically puts the return address on the system stack;
- It requires two bytes to store the return address, here being positions 0x30EC and 0x30ED.

□ C. Analysis of the execution in assembly language

- The function begins by reserving space on the system stack;
- It subtracts the constant value `#6` from the register `SP`, pointing to the address 0x30EC;
- The system stack base address is 0x030E6;
- On the system stack it saves the parameters passed from the registers `R12` and `R13` in `SP+0` and `SP+2`, corresponding to the address of variables *a* and *b*, respectively;
- These actions are carried by the code:

```
SUB.W      #0x0006, SP
MOV.W R13, 0x0002(SP)
MOV.W R12, 0x0000(SP)
```

- The new Fibonacci number is now calculated;
- It adds the contents of memory addresses sent;
- This is accomplished with the following sequence of instructions:

```
MOV.W      0x0002(SP), R15
MOV.W      @R15, R15
ADD.W      @R12, R15
MOV.W      R15, 0x0004(SP)
```

- The processor begins by storing the value on the system stack pointed to by `SP+2` (variable *b* address) in `R15`, followed by a data transfer to register `R15` of the memory address pointed to by `R15`;
- Following this statement, register `R15` has the address of variable *b*;
- An add operation is then performed by adding the value contained in the address stored in register `R12` (variable *a* address) with the value contained in register `R15`;
- The result of the sum in `R15` is placed on the system stack at the address `SP+4`, which is the space reserved for the local variable *c*;

- The data management in order determine the next Fibonacci number is carried out using the following code to update the variable **a**:

```
MOV.W      0x0002(SP),R15
MOV.W      @SP,R14
MOV.W      @R15,0x0000(R14)
```

- This sequence of instructions begins by copying the value stored on the system stack at position SP+2 (variable **a** address) to register R15;
- The value contained on the system stack at the position SP+0 (variable **b** address) is copied to register R14;
- Copy the contents of the address stored in R15 to the address stored in register R14;
- Using this sequence of operations, the value in variable **a** will be equal to the value of variable **b**;
- The variable **b** is updated with the result stored in local variable **c**;
- This operation is performed using only two instructions;
- Initially, the contents stored on the system stack at SP+2 are copied to register R15;
- This register will hold the address of variable **b**;
- The value of the system stack stored in SP+4, corresponding to the local variable **c**, is copied to the variable **b**:

```
MOV.W      0x0002(SP),R15
MOV.W      0x0004(SP),0x0000(R15)
```

□ D. Returning the results function

- Finally, after this sequence of operations, the result is returned in register R12;
- The code begins by moving the value stored on the system stack at SP+4 to the register R12;
- It restores the system stack state reserved for this function, by adding the constant originally subtracted from the SP at the beginning;
- The register SP will point back to the original position 0x030EC;
- The execution of the application is returned to the main function by the last instruction:

```
MOV.W      0x0004(SP),R12
ADD.W      #0x0006,SP
RETA
```

□ E. Function result storage

- After returning to the main function, the result is again stored on the restored system stack using the instruction:

```
MOV.W R12, 0x0006 (SP)
```

2.3 IAR Embedded Workbench IDE

The IAR Embedded Workbench™ (EWB) is an Integrated Development Environment (IDE) that allows developing and managing projects for embedded applications. It is available for several microprocessors and 8-bit, 16-bit and 32-bit microcontroller. Among these is the Texas Instruments' MSP430 Microcontroller Family.

The information provided in this section resumes the detailed information provided in the "MSP430 IAR Embedded Workbench® IDE User Guide". Additional information can be obtained on the IAR Systems web site www.iar.com.

2.3.1 IAR EWB main features

The tools included in the IAR EWB for the MSP430 are:

- C/C++ Compiler;
- Assembler;
- XLINK Linker;
- XAR Library Builder and the XLIB Librarian;
- Editor;
- Project manager;
- Command line build utility;
- C-SPY™ debugger.

Integrating these tools together in a unique workspace development environment facilitates efficient programming, providing a reduced development time.

There follows a short presentation on the procedure used to create a project. The "Hello world" example demonstrates the typical development cycle and how the compiler and the linker are used in order to create an application for the MSP430.

2.3.2 Laboratory 1: "Hello World" Beginner's project

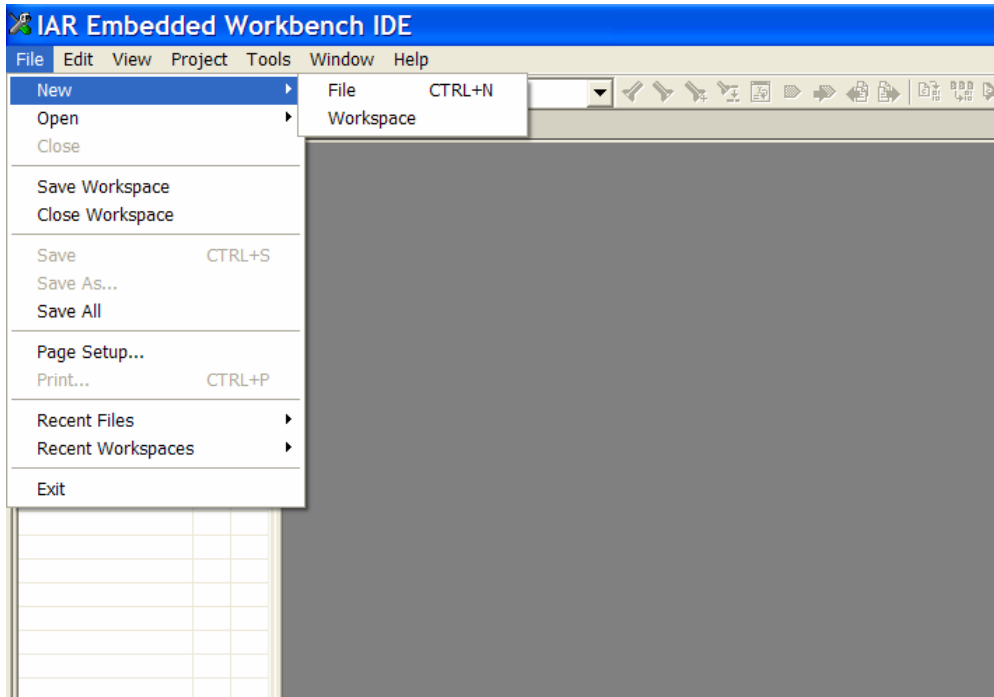
Setting up a new project

- The IDE methodology is based on a workspace concept;
- Modules can contain one or more projects;
- The projects are made up of one or more source code files needed for the binary file, which allow the simulation and/or programming to take place;
- Before creating a project, it is necessary to create a workspace.
- When IAR EWB starts for the first time, it creates a default workspace.

Step 1. Creating a workspace window

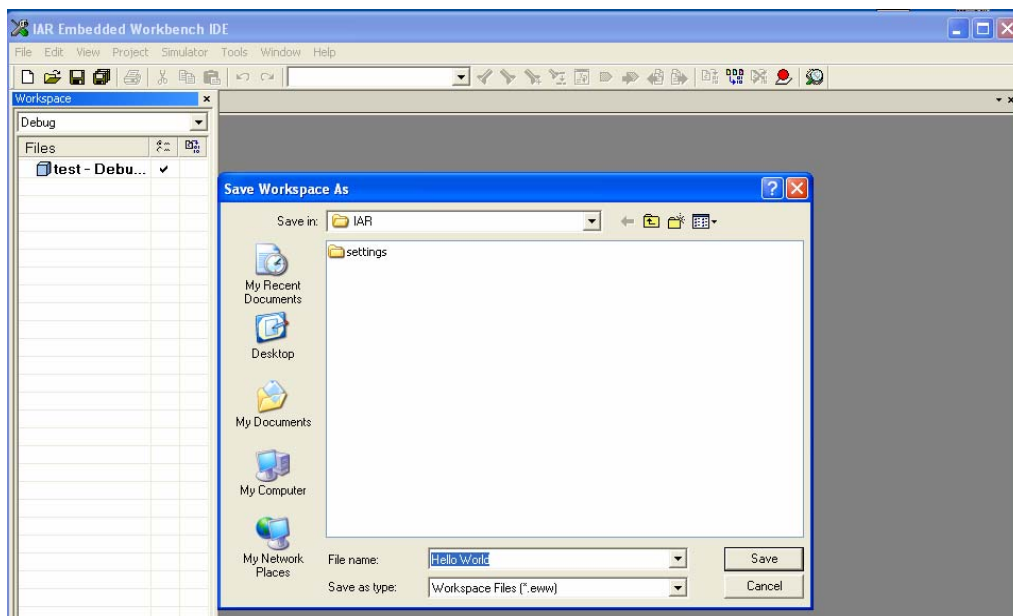
- Choose **File > New > Workspace** (Figure 2-39);
- Pressing **OK** will generate a window with an empty workspace.

Figure 2-39. IAR EWB – Creating a workspace.



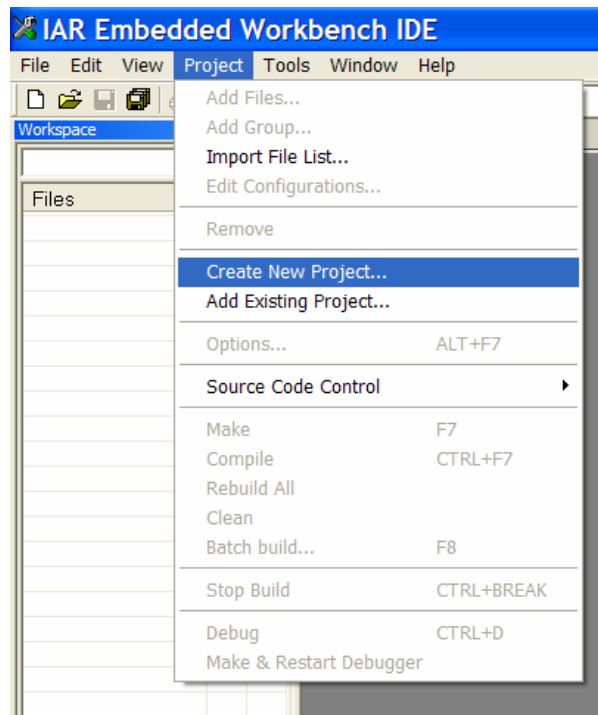
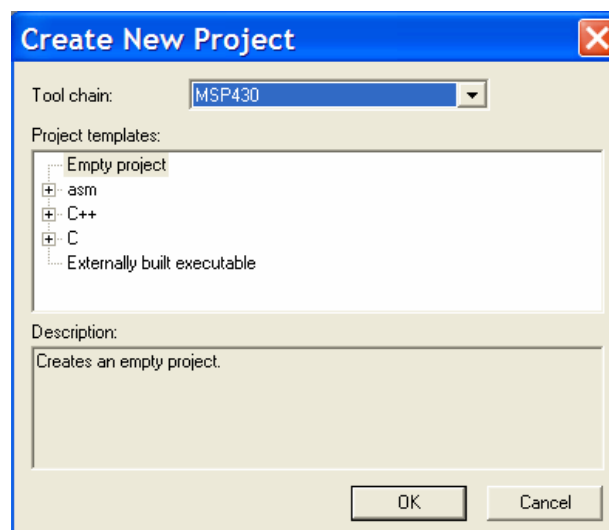
- Save the workspace: Choose **File > Save Workspace**.
- Specify where the workspace file should be saved.
- Name: HelloWorld in **File name > Save** to create the new workspace (file extension: .eww) as present in Figure 2-40.

Figure 2-40. IAR EWB – Saving a workspace.



Step 2. Creating the new project

- ❑ Choose **Project > Create New Project**(see *Figure 2-41*);
- ❑ The **Create New Project** dialog box lets the new project to be based on a project template;
- ❑ Select the project template **Empty project** to creates an empty project that uses default project settings, as shown in *Figure 2-42*;
- ❑ Set the **Tool chain** to **MSP430**, then click **OK**.

Figure 2-41. IAR EWB – Creating a new project.*Figure 2-42. IAR EWB – Empty project.*

- ❑ In the **Save As** dialog box, specify the location of the project file (newly created projects directory);
- ❑ Name **Project1** in the File name box, and click **Save** to create the new project (see *Figure 2-43*).
- ❑ The project will appear in the workspace window (*Figure 2-44*).

Figure 2-43. IAR EWB – Saving the project.

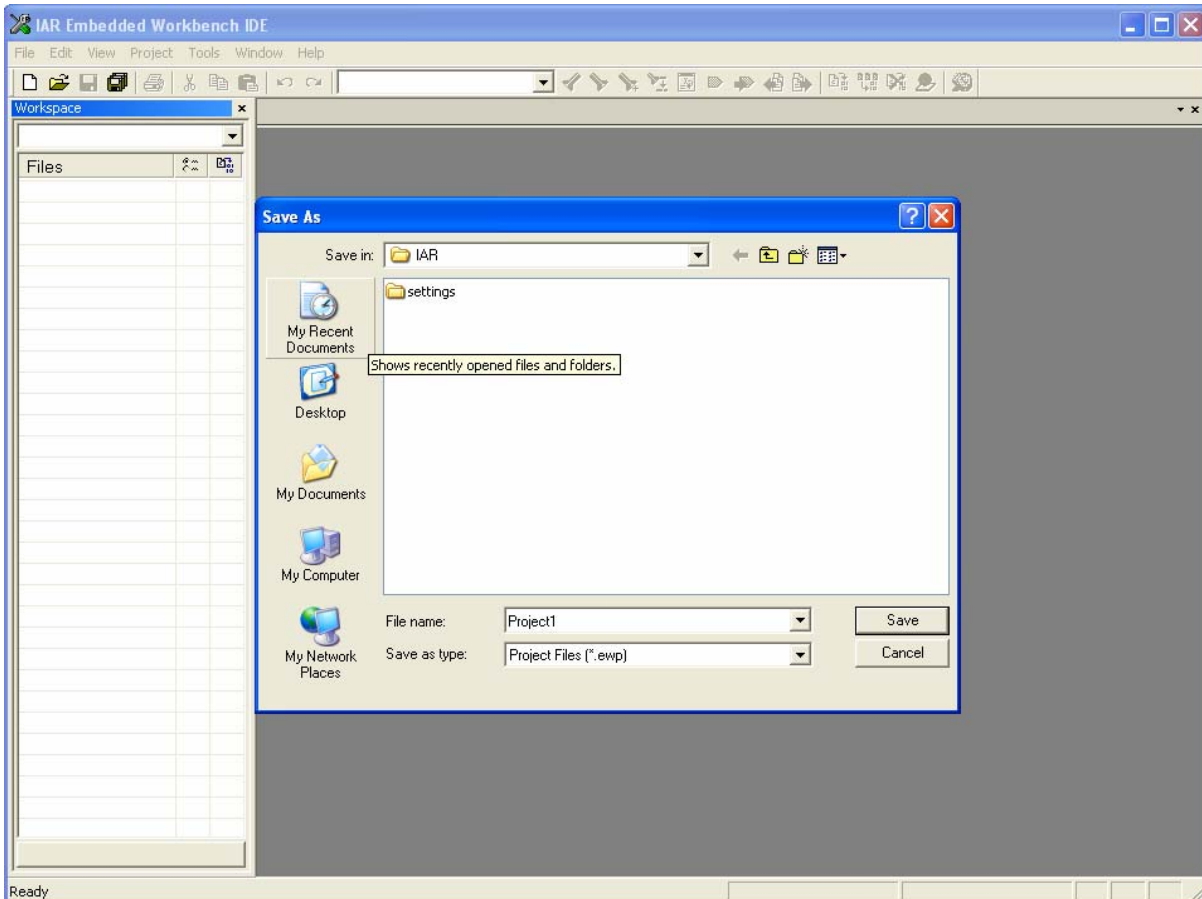
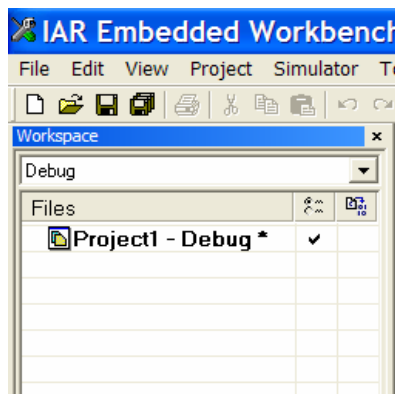


Figure 2-44. IAR EWB – Workspace window.



Step 3. Adding files to the project

- ❑ In the Workspace window, select the where you want to add a source file, in this case directly to the project;
- ❑ To create new source files, choose **File>New** and select **Source/Text**;
- ❑ Choose **Project > Add Files** (see Figure 2-45);
- ❑ Locate the file **Lab2C_solution.c** and click **Open** to add it to **Project1** (see Figure 2-46).

Figure 2-45. IAR EWB – Add files to the project.

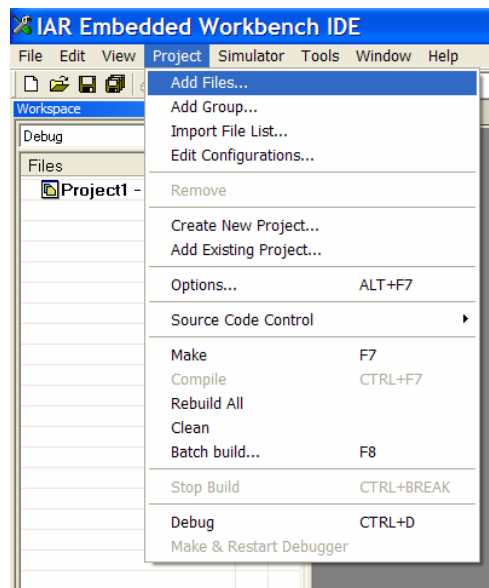
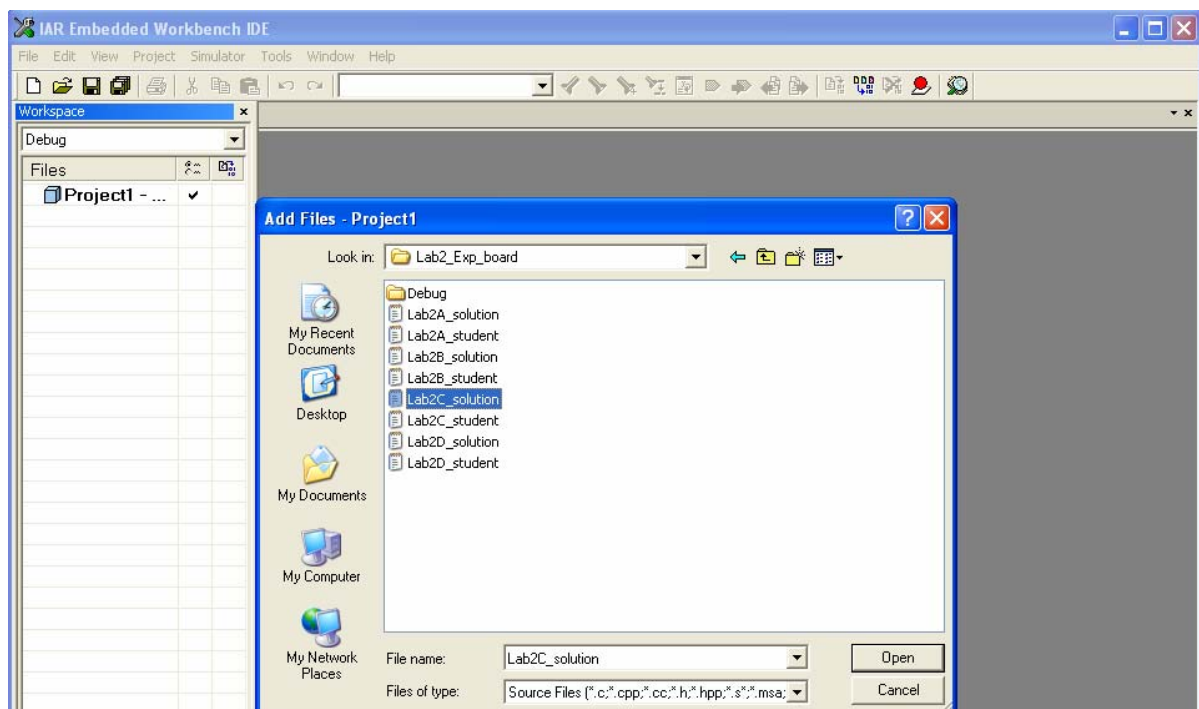


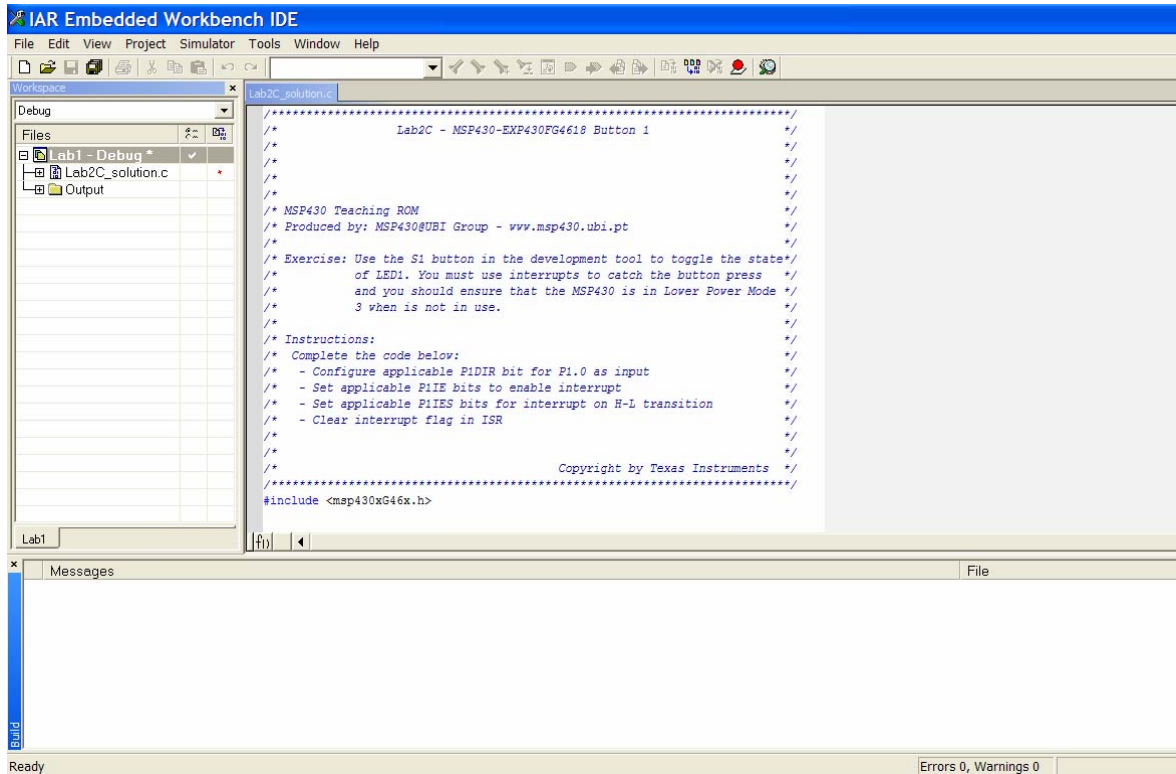
Figure 2-46. IAR EWB – Add files to the project.



The Integrated Development Environment (IDE), as shown in *Figure 2-47*, is composed of three windows:

- Workspace: Project and associates files;
- Text editor: File(s) source code;
- Debug Log: Compilation information.

Figure 2-47. IAR EWB – IDE window.



Step 4. Setting project options

- Select the project folder icon **Project1 - Debug** in the Workspace window and choose **Project > Options** (see *Figure 2-48*);
- The **Target** options page in the **General Options** category is displayed;
- Settings:
 - MSP430 device: MSP430FG4619 (see *Figure 2-49*);
 - Output file: Executable;
 - Library: CLIB.

Figure 2-48. IAR EWB – Project options.

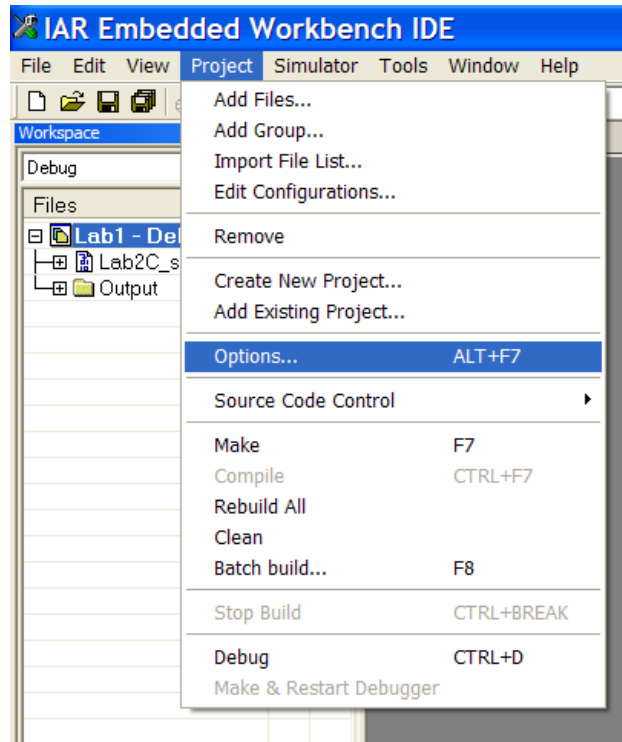
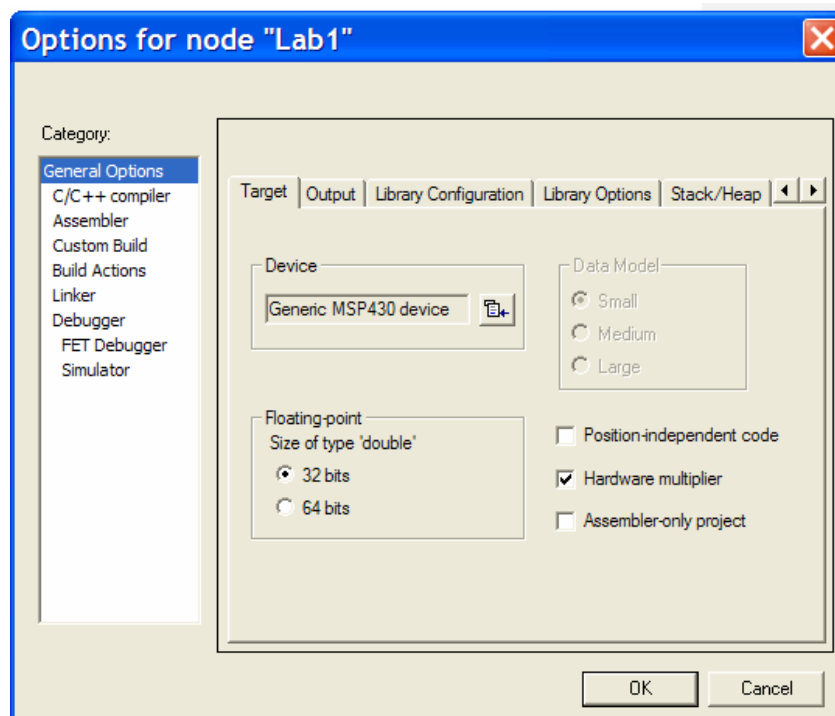


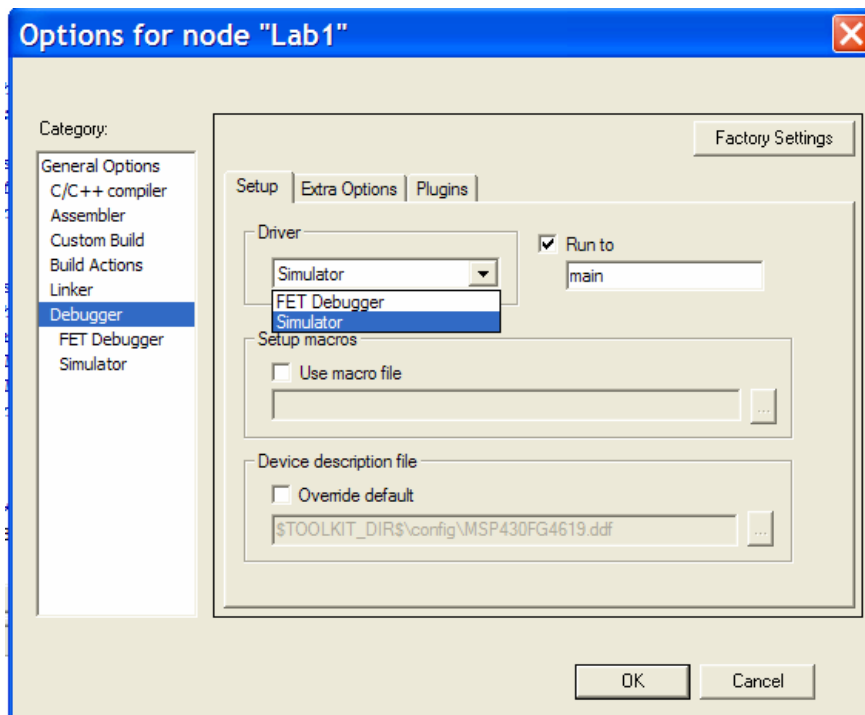
Figure 2-49. IAR EWB – General options.



- ❑ Select **C/C++ Compiler** in the **Category** list to display the compiler option pages:
 - Language: Details specifications for the interpretation of the C language;
 - Code: Configuration of code optimization;
 - Output: Configuration of output files type;
 - List: Configuration of list files created by the compiler (containing both C and generated assembly code).

- ❑ Select **Debugger** in the **Category** list to display the debugger option pages:
 - Simulate in the PC: **Setup -> Simulator**.
 - Testing the application at the μ C: **Setup -> FET Debugger**.
 - After choosing one of the options (for this example **FET Debugger**), click **OK**, as shown in *Figure 2-50*.

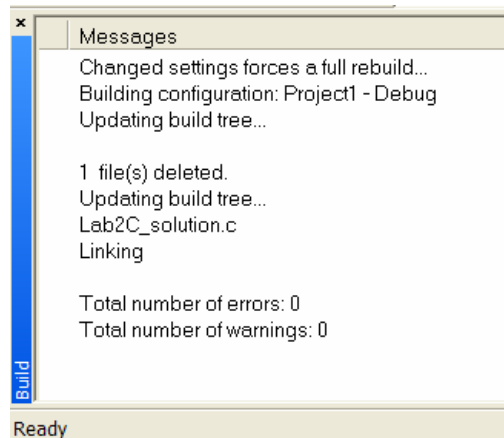
Figure 2-50. IAR EWB – Debugger options.



Step 5. Compiling and linking

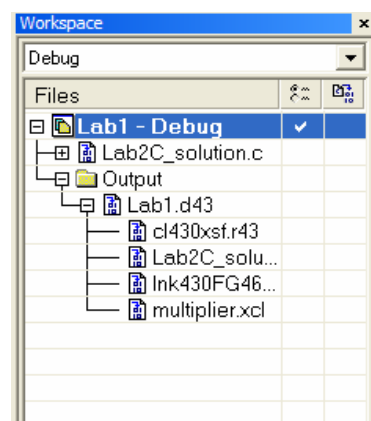
- ❑ To compile the file **Lab2_solution.c**, select it in the Workspace window;
- ❑ Choose **Project > Compile**;
- ❑ The progress will be displayed in the Build messages window as shown in *Figure 2-51*.

Figure 2-51. IAR EWB – Compile messages window.



- New directories in the Debug directory have been created containing the directories List, Obj, and Exe (see Figure 2-52):
 - List: Destination directory for the list files (.lst).
 - Obj: Destination directory for the object files (.r43) from the compiler and the assembler (input to the IAR Linker).
 - Exe: Destination directory for the executable file (.d43) that will be used as input to the IAR Debugger.

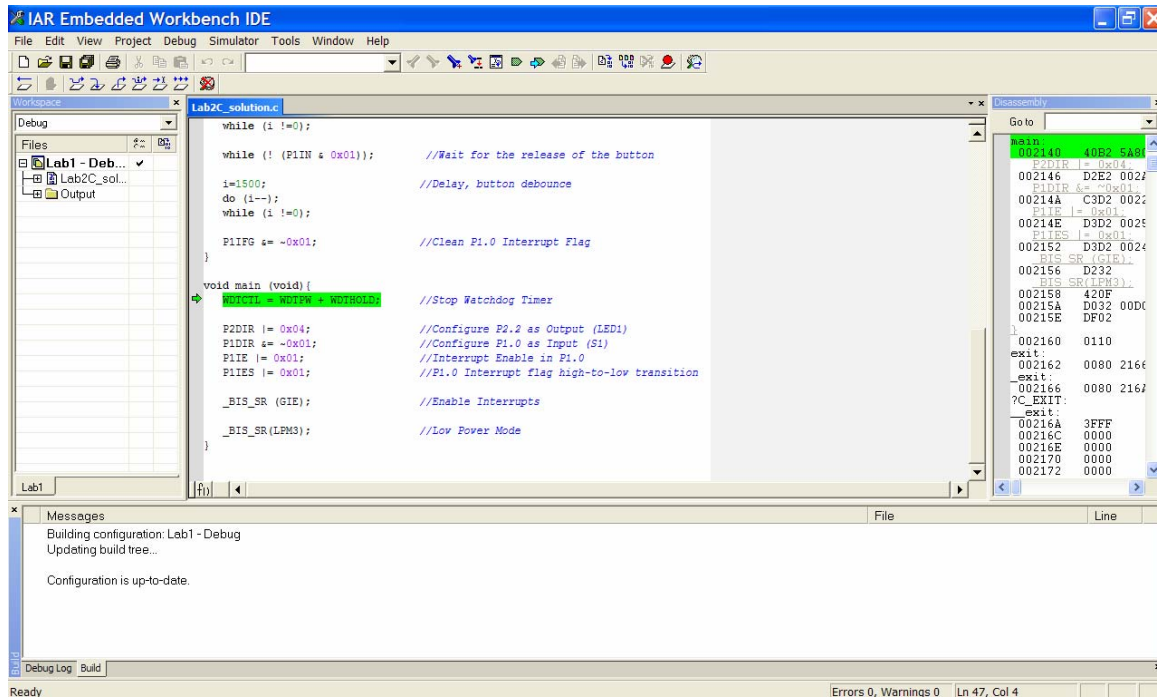
Figure 2-52. IAR EWB – Directories tree.



Step 6. Debugging the application

- Choose **Project > Debug**;
- The workspace configuration is changed to the project execution mode (see Figure 2-53).

Figure 2-53. IAR EWB – Debug view window.



□ Workspace configuration for monitoring (**View**>) the windows of particular interest to the user:

- **Disassembly:** Hexadecimal format of the memory content interleaved with the C source code;
- **Memory:** Memory usage with several data viewing options;
- **Register:** Monitoring CPU and SFR registers in the RAM;
- **Watch:** Monitor expressions or variables;
- **Locals:** Local variables inside a routine or function;
- **Live Watch:** Expressions or variables for real-time monitoring;
- **Call Stack:** Monitor stack content;
- **Terminal I/O:** Monitoring the I/O ports state.

□ To continue debugging the application, select **Debug > Go**.

This brief presentation of the IAR EWB IDE only provides an initial overview. It is recommended to read and carry out the tutorials included in the “MSP430 IAR Embedded Workbench™ - IDE User Guide for Texas Instruments’ MSP430 Microcontroller Family” manual for a detailed description of the IAR IDE.

2.4 Third party MSP430 IDEs

Some third parties have developed Software Development Tools to configure and program the MSP430 hardware development tools. Previously described were packages provided by Texas Instruments (Code Composer Essentials) and by IAR Systems (IAR Embedded Workbench - Kickstart Version). In addition, other parties have developed IDEs, such as Rowley Associates (CrossStudio) and the MSPGCC, which is a compiler developed by the Open-Source Community.

Following the information provided on the TI webpage, below is a list of the third parties that have developed IDEs for the MSP430:

Forth, Inc provides SwiftX cross-development system for the MSP430 featuring a multitasking kernel with build-in support for low-power mode and a library of several hundred functions. SwiftX supports fully interactive development including the Forth high-level language and assembler, to produce extremely compact, fast, low-power applications. They also offer a full line of services, including programming courses, software and hardware design and development, and even board layout and production to take your idea all the way to a finished product.

HI-TECH software provides HI-TECH for MSP430 is an advanced C compiler with a fast and flexible programming environment for the Texas Instruments MSP430 devices. HI-TECH for MSP430 makes use of specific MSP430 features and using an intelligent optimizer and can generate high-quality code easily rivalling hand written assembler.

ImageCraft's ANSI C tools offer quality code generation wrapped in an easy-to-use GUI. ImageCraft claim to provide excellent customer support, which other companies cannot match. ImageCraft also claim to have a low cost factor, and the best deal in C tools for the innovative TI MSP430.

Version 7 C Compiler Tools with Windows IDE for TI MSP430 / MSP430X Microcontrollers.

Phyton, Inc. provides Project-430 is a set of hardware and software tools for developing MSP430 applications under control of one integrated development environment (IDE). A full package includes MCA-430 macro assembler, PDS-430 software debugger/simulator, and PICD-430 in-circuit debugger integrated under control of the Project-430 IDE, so this toolset provides a complete development cycle, from editing source texts, to getting debugged code, and "burning" it into a target microcontroller or memory device. Selected third parties' C compilers can also be bundled with the Phyton's tools to make it capable for the C-level development. Instead of using the Phyton PICD-430 debugger the IDE can drive very popular TI's MSP-FET430 flash-emulation tools for JTAG debugging. The following Project-430 configurations are available:

Quadravox provides the AQ430: C- Compiler and C/ASM-Debugger.

Quadravox, Inc. has designed development tools for embedded systems since 1996. Archelon, Inc. has, since 1982, developed tools for microcoding and, in particular, have pioneered the implementation of standard high level languages on micro programmed hardware, signal processor chips and other unique architectures.

The AQ430 development tools are a fully integrated software environment for all flash based MSP430 micro-controllers. They allow the user to create projects, edit files, compile, assemble, link, make / build and debug an MSP430 application within a single Windows™ IDE.

All debugging operations occur on the actual target system, controlled via a JTAG interface, like the MSP430 Flash Emulation Tool. AQ430 allows an unlimited number of "software" breakpoints on C or assembly language instructions for non-real time debugging, and the usual number of hardware breakpoints for real time operation. The user can switch from the "C source view" to the "asm view" to see (and step through) the assembly code generated by the C compiler. There is no limitation (other than the physical address space of the MSP430 controller used) on the size of C programs used with AQ430.

Find additional details on www.ti.com.