Introl−CODE User Guide

Introl Corporation

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Table of Contents

Introl−CODE User Guide

This document describes CODE, its user interface, and using it to build embedded systems.

Quick Start
: General information about CODE; how to start it, a brief tutorial, common CODE menus, and keyboard shortcuts.

The Directory/Project Browser
: Set the current project directory and manipulate a project and its files.

CODE Projects
: What is a project? The project editor window and how to set up a project.

The Editor
: The source file editor.

The Debugger
: Execute and debug target applications.

The Manual Browser
: View on−line manuals and other HTML documents.

Preferences
: Setting default colors and fonts, etc.

Searching and Bookmarking Files
: Find strings or patterns in text and optionally replace them. Set bookmarks in text that can be found later. Go to previously set bookmarks.

The Command Window
: A command line interface to the CODE scripting language and the CODE command line programs.

Version numbers
: What CODE version numbers mean.

Installation
: Install CODE on your computer.

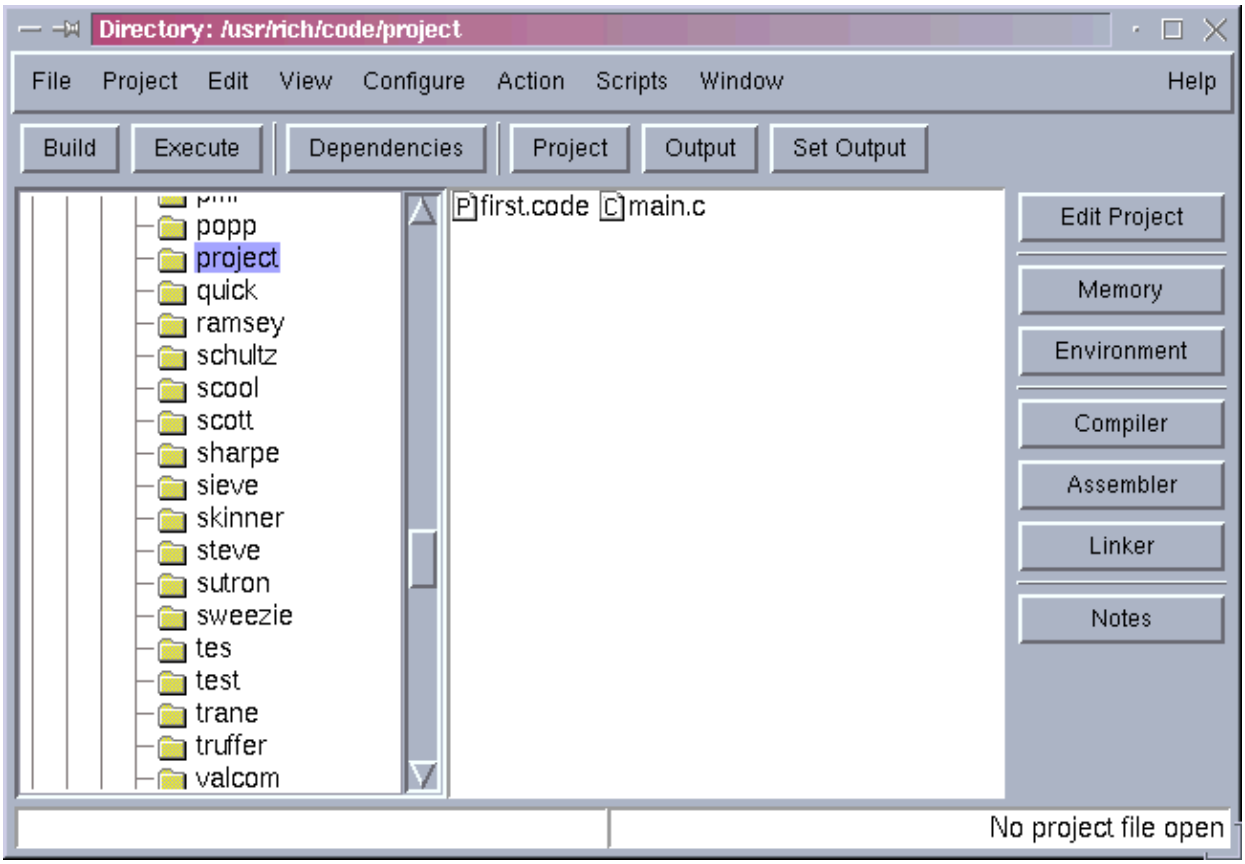
Application Notes
: Miscellaneous topics that may be interesting to CODE programmers.

Demonstration Programs
: Projects that demonstrate various wasys to use CODE.

Quick Start

If you are reading this from the CODE distribution CD−ROM, you may want to install CODE first. See the Installation section for information about installing CODE on your system.

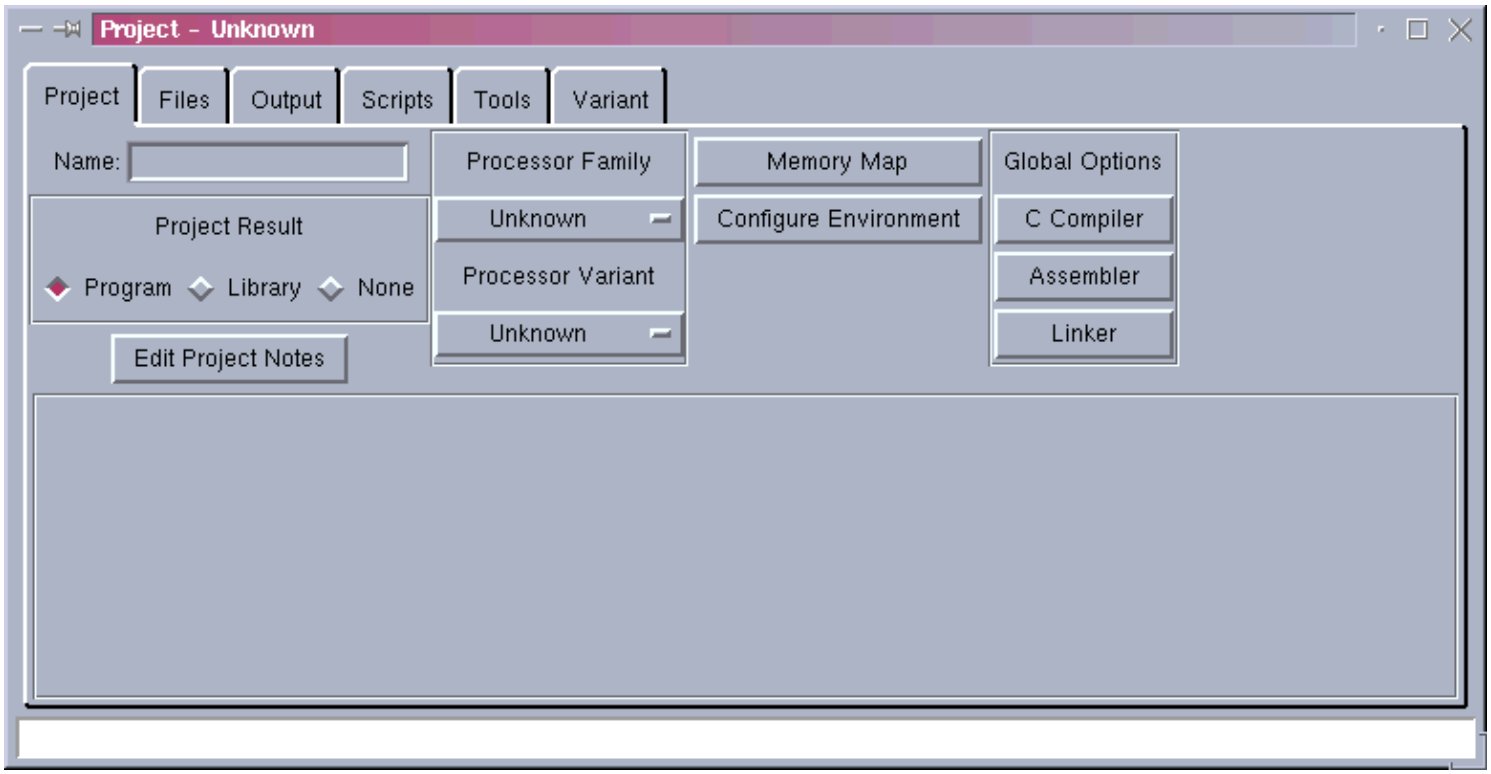
When you first start CODE you see the on−line manual browser. If you are reading this from CODE, go to the **Window** menu and select **New Window**, or press **F11**. This will open a new window that will allow you to follow along with this tutorial and see what's happening. Both windows show the manual browser but you can change either of them into another CODE window. Select the new window now and choose **Directory** from the **Window** menu or press **F3**. This will change the window into the directory/project browser. The directory browser looks like this:



The directory browser is an important window that allows you to navigate around your system and create and modify *projects*. A project is a set of files and rules used to build a program, among other things. A program is built by taking one or more source files, in either C or assembly language, translating them to object files, and linking them, along with *startup code* and one or more libraries, to create a single executable file.

Use the directory browser to change to the directory you'd like to use to create a project by clicking on a folder in the directory tree on the left side or double clicking on a folder on the right side of the window. Now, select **New Folder** from the **File** menu and give your project folder a name. Double click on your new folder to go into the project directory. The window shown above displays a new, empty project directory.

Now you're ready to make your first project. Select **New Project** from the **Project** menu to bring up the project editing notebook. This will bring up the project editor window:



The next step is to choose a processor family and variant for which you want to write a program. You can use the menus in the project notebook to select them now. CODE needs to know the processor family to compile and assemble your source files for the appropriate processor. The processor variant is used to determine which low level input/output functions should be used when a program is built, among other things.

The right side of the directory browser shows files that are in the current directory as well as other files that are associated with the project. Select **Save Project As...** from the **Project** menu to save your new project. You can give it any name you choose as long as the name is unique in this directory. After you've saved the project, you'll see a file in the directory window with an extension of **.code**: this is your project file.

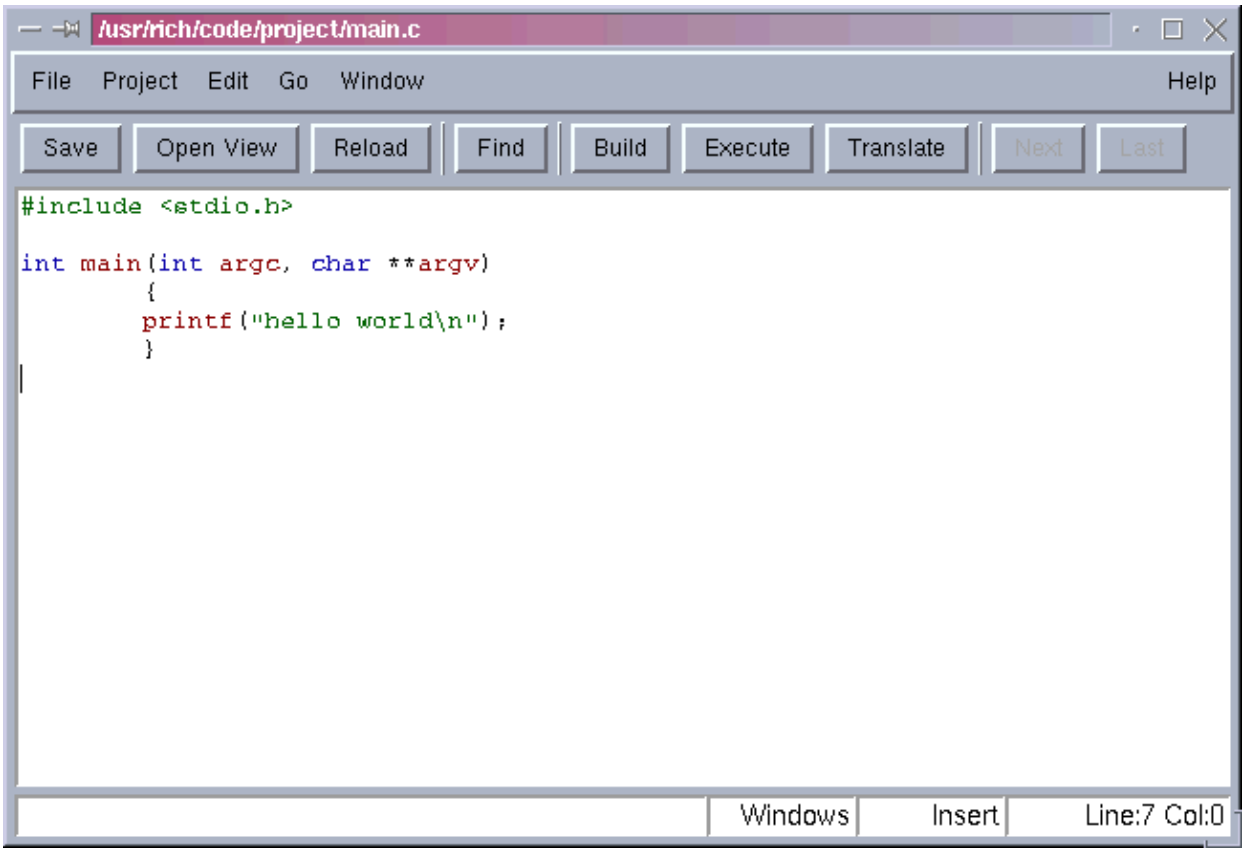
The name of the project file is also used to name other things generated by the project, for example, the final executable file and the CODE generated startup code both include the project name as part of their name.

Now you are ready to create your first program. From the **Window** menu, select **Editor** or press **F4**. This opens the Editor where you can create a new source file. Type in the following program (you can use **Copy** and **Paste** from the **Edit** menus of the Manual and Editor window to copy this):

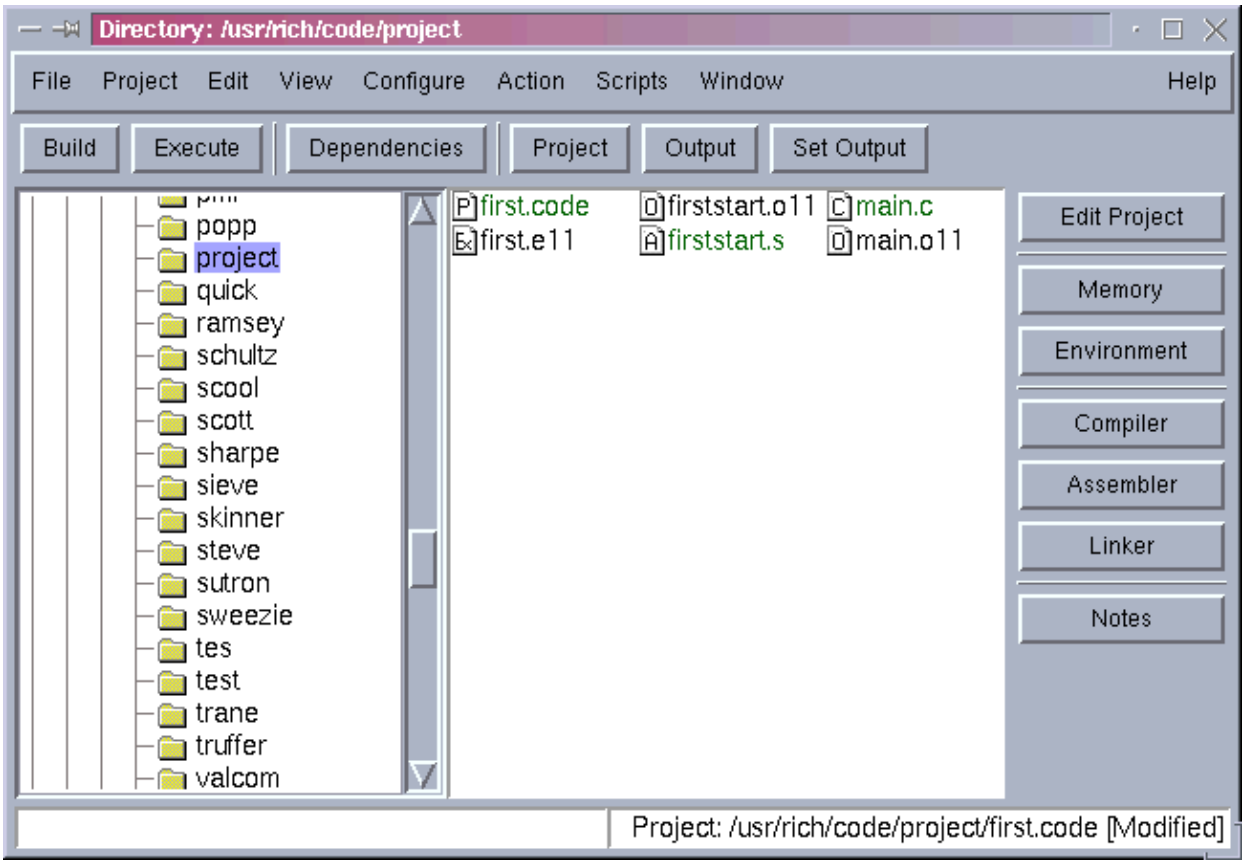
```
#include <stdio.h>

int main(int argc, char **argv)
        {
        printf("hello world\n");
        }
```

Now choose **Save As** from the File menu and save the file with a **.c** extension, **main.c** for example, to tell CODE it is a C source file. You'll notice that the source code in the Editor window is now colored appropriately for a C program. The editor window should look like this:

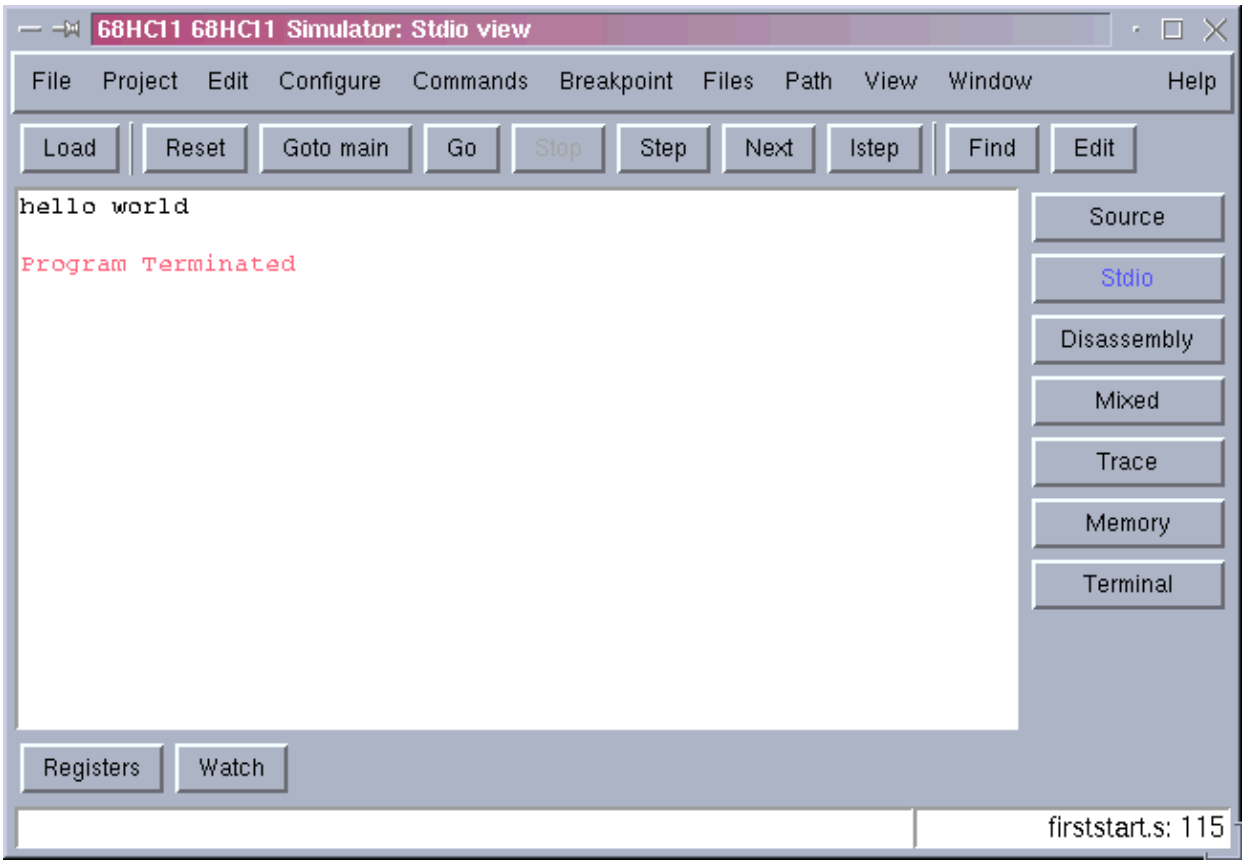


You can now add this file to your current project by selecting **Add to Project** from the Editor's **File** menu. Now go back to the directory window by selecting **Directory** from the **Window** menu or pressing **F3**. Press the **Build** button below the menu bar. This will create a program from your project. Your directory window should look something like this:



Notice that two files were added to the directory: a **.o** file and a **.e** file each with two digits in the extension. The **.o** file is the object file created by compiling your **.c** file. The **.e** file is the executable program created by linking your program with the standard runtime libraries. The two digits in these extensions represent the processor family you have chosen for the project. Two files with **start** in their names were also added: The **.s** file is the generated assembly startup code and the **.o** file is the assembled object file from the startup code. The files which are colored green in the directory window are part of your project.

Now press the **Execute** button, next to the **Build** button, and your program will execute in the debugger. The Debugger window will be opened and the program output will be shown, followed by the message "Program Terminated":



You are now looking at the debugger's **Stdio** view. To run the program again you can press the **Reset** button and **Go**. You can use the buttons on the right side of the debugger window to see other debbugging views.

Go back to the directory window. You'll notice that the status line at the bottom of the window will indicate what CODE knows about a file when you pass the mouse cursor over the file. You can right click on a file to perform operations on it. Double clicking on a file will perform the first operation in the right click menu. For example, double clinking on a **.c** or **.s** file will open it in the editor.

The buttons on the right side of the directory window allow you to modify the project configuration. You can find more information about the directory/project browser by reading the Directory/Project section of this manual.

Now let's add an assembly language file to the project (call it **donothing.s**):

```
                section .text           ; place the program code
donothing:                              ; a colon causes a symbol to be exported
                rts                     ; really do nothing!
```

Remember to add the file to the project. Modify the C file to call it:

```
#include <stdio.h>

extern void donothing(void);            // assembly language function

int main(int argc, char **argv)
        {
        printf("hello world\n");
```

```
donothing();
}
```

Now, pressing the **Build** button in either the directory or editor window will build a program containing both the C program and the assembly language file.

You can see the effect of an error that occurs during the build by editing the C file and removing the closing quote from "hello world\n". Press **Build** and you see the first error line highlighted and the error message in the status line at the bottom of the window. The **Next** and **Last** buttons will show you other errors that the CODE has found in the file.

There are a set of demonstration projects to help you get used to using CODE. You can use the directory window to go into the demonstration directories and double click on the various project files to open them.

General Navigation

The **File**, **Project**, **Window** and **Help** menus exist in all CODE windows.

The **File** menu will always have an entries to close a CODE window and to exit CODE The **File** menu also contains an entry to set CODE preferences, such as fonts and colors used in CODE windows. Other windows may add additional entries to the **File** menu.

The **Project** menu performs actions on the current project. The **Project** menu has entries to build the current project, edit information about it, add a file to it, etc.

You can use the **Project** menu to create a new project file or open an existing project file in the current or another directory.

The **Window** menu is used to navigate between CODE windows. The first entries in the **Window** menu show the titles of all CODE windows you have open besides the current one. You can select one of these entries to raise that window to the top. The middle entries in the **Window** menu change the contents of the current window. The keyboard shortcuts in the **Window** menu can be used in any CODE window to change to another window. The **New Window** entry of the **Window** menu will make another window into the current CODE session. You can use this to get multiple views of a file you are editing or a program you are debugging. You can create an entirely independent CODE session by using the **New CODE** menu entry.

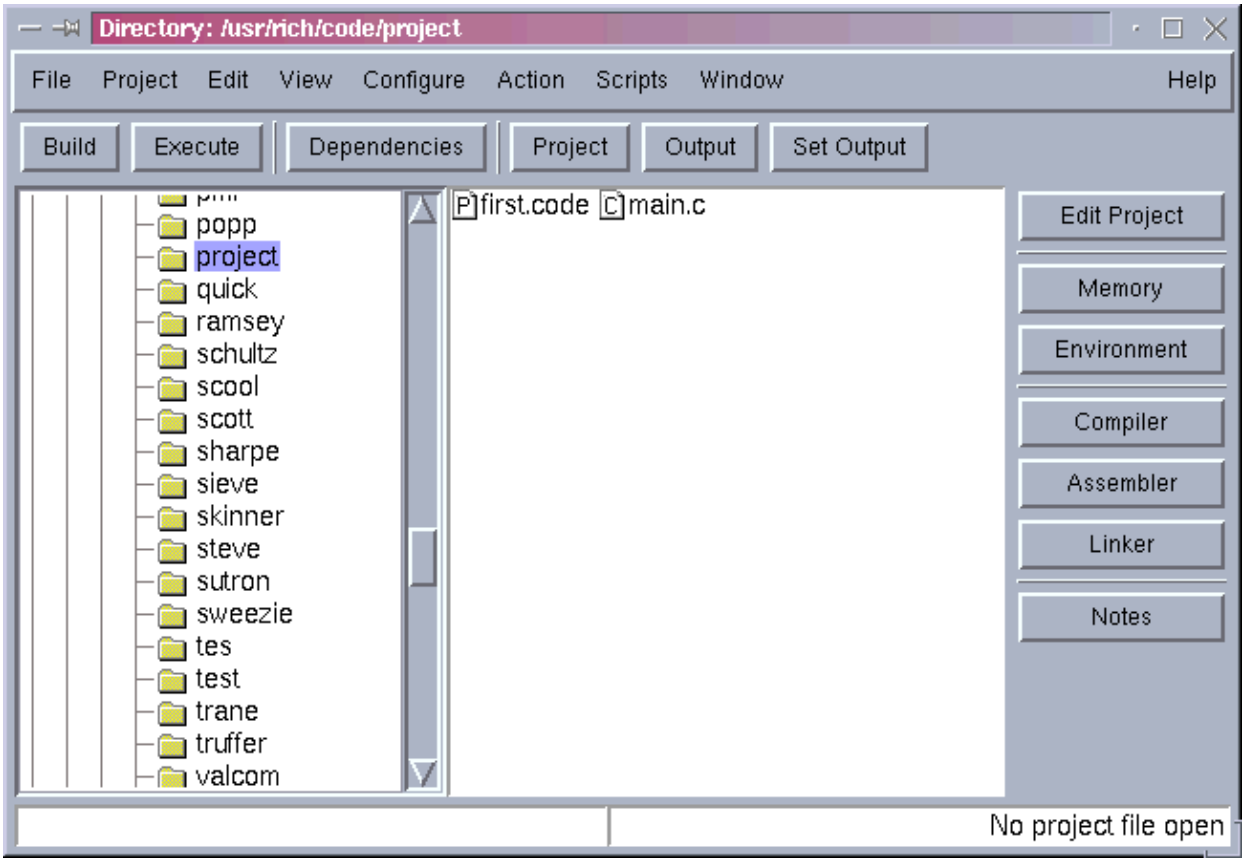
The **Help** menu allows you to bring up infomation like this about all the CODE windows and to access the CODE reference manual. The first entry in the **Help** menu always brings up information about the current window. The **F1** function key can be used as a shortcut.

In your first tour of CODE you should visit all the CODE windows. If there are aspects of the windows you don't understand, you can use the **Help** menu (or the **F1** key) to bring up more information on each window.

The bottom of all CODE windows contains a status bar that will display information about what CODE is doing. CODE will also display help information about items in a window in the status bar when the mouse pointer moves over the item.

The Directory/Project Browser

You enter the directory window by selecting **Directory** from the **Window** menu or pressing **F3** in a main CODE window. The directory window looks like this:



The directory window adds **View**, **Configure**, **Action**, and **Scripts** menus to the standard menus. The **View** menu allows you to reload the current directory display. The **Configure** menu is a shortcut way of changing the projects current processor family and variant. The **Action** menu will display operations that can be performed on a selected file or files, just as the right mouse button does as described below. The **Action** menu will be empty if no file is selected. The **Scripts** menu allows you to view and modify build scripts that are associated with a project.

The directory window allows you to navigate through directories and build simple applications or whole projects. The directory window consists of two parts: the left side shows the current directory you are in and can be used to change directories. The right side shows the files in the current directory and any additional files that might be included in the project. The files that exist in the directory are shown at the top of the window. Below the directory files are any files that might have been *imported*, or added to the project from other directories. They will be displayed with their full path name. Libraries used by the current project will be displayed below the imported files, one per line. When a project is built the project manager searches the library files in the project in the order they are displayed in the directory window.

Files in the current directory will be displayed in black, unless they are in the current project, in which case they will be displayed in green. Imported files will also be displayed in green. Libraries used by the project will be displayed in blue. Any files that are part of the project but can't be found are displayed in red. This can happen if you delete a project file and don't remove it from a project. These and many other colors and

fonts that are used in CODE can be controlled in the Preferences window.

Another view of the files in a project is available in the **Files** tab of the Project Editor window. That view allows you to specify the order that files in your project are processed.

CODE recognizes many file types and can perform operations on files for you. As you move the mouse pointer around in the Directory window the status bar will indicate the type of file under the pointer.

The right mouse button will bring up a menu of operations that can be performed on a file. For example, C files can be compiled by right clicking on them and choosing **Compile**. You can also right click on a project source file to change its build options or attach build scripts.

In some cases, CODE can determine all the information it needs to operate on a file from its name. An assembly language file with a **.s*XX*** extension will be assembled with the appropriate assembler, for example. Some files, like **.c** files, don't have enough information or CODE to determine exactly how to perform a certain operation. CODE will use information that you can configure to decide how to operate on a file. A crucial bit of information that CODE needs to compile a C file (and to perform other operations on C and other files) is the default target microcontroller. This can be set in the Project section of the Project window.

Project files have a **.code** file extension and can be opened by double clicking on them in the directory window. CODE can only have one project file open at a time.

The buttons below the menu bar allow you to build and manipulate the current project. The **Build** button will build the current project, if it has changed. The **Execute** button will make sure the project is up to date and will then run the project's program file in the debugger.

The **Dependencies** button will search all the project's C files and determine which **#include** files they use. This information is used by CODE to tell if a C file should be rebuilt if an included file has changed.

The **Project** button will change the directory to the current project's *home* directory, that is, the directory that contains the **.code** file. The **Output** button will change the directory to the current *output* directory, the directory where object files and project results are placed. You can use the **Set Output** button to set the output directory for the current project.

The buttons on the right side of the directory window allow you to change important parts of the project configuration. You can also access these in the project editor.

CODE Projects

A CODE *project* is a collection of files and rules for performing operations on those files. The files in a project can be source files, in either C or assembly language, or can be project files, known as *subprojects*, that are built before the source files in a project are built.

Creating a Project
: Creating a new project, configuring the project runtime environment and memory map.

Manipulating Project Source Files
: Manipulating project source files and controlling link order.

Specifying Project Output
: Specifying alternate object file formats for the result of a project.

Using Build Scripts
: Setting up build scripts that will be run at various times while a project is being built.

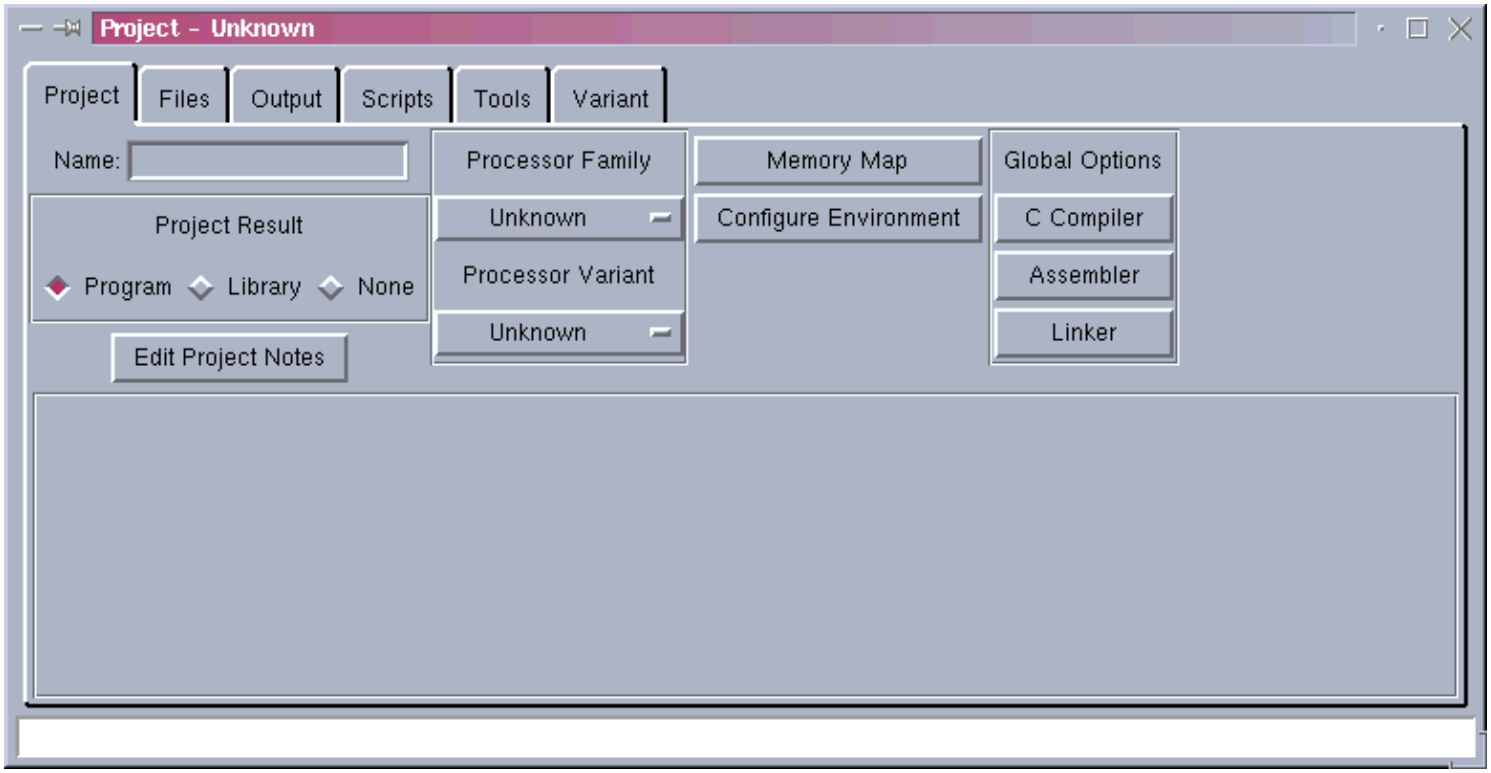
Setting Tool Defaults
: Setting up the build environment's default commands and options.

Adding a Processor Variant
: Adding a new processor variant to CODE.

Creating a Project

The first thing to do when creating a new project is to use the Directory/Project browser to go to a directory that will contain the project. You can use the directory browser to create a new folder, if necessary.

To create a new project you can select **New Project** from the **Project** menu in any CODE window. A project editor window like this will pop up:



The first tab of the project editor is used to set up basic information about the project. The main thing you need to set up about a project initially is the processor family and variant your target system is using. These values are used to configure the project build tools for the specific microcontroller that you are using.

Now you should select the result of the project you are making. Most projects build *programs*, self−contained executable files that can be run on your target system. A project that builds a *library* builds a collection of program parts that can be used by other projects to build programs. Library projects are used to build the standard C and assembly runtime libraries, for example. A project's result can be set to none, which means the project doesn't build anything directly, but rather relies on *build scripts* and *subprojects* to produce one or more results.

The **Memory Map** button brings up a dialog that lets you set up the memory configuration of your target system.

The **Configure Environment** button brings up a dialog that lets you configure the startup code and libraries that are used in your project.

The **Edit Project Notes** button allows you to add arbitrary notes to your project. The first few lines of the project notes will show up below the button for easy identification of the project.

The **Global Options** buttons set up the default options used to compile, assemble, and link the files in your project into a program. It is also possible to set the build options of individual files in your project in the directory browser or under the Files tab of the project editor.

The **Files**, **Output**, **Scripts**, **Tools**, and **Variant** tabs are described in their own help pages.

When you have finished the initial configuration of your project, you should select **Save Project As...** from the **Project** menu of any main CODE window. This save your project file, with a **.code** extension, and also fill in the Name entry. The project name is used to name certain files created when a project is built, such as the final executable file and the startup code assembly source file.

Setting up the Environment

The *environment* of a CODE program project controls how the program starts. A CODE program usually starts by executing *startup code*, a short assembly language program, which is generated automatically by the CODE project builder. The tabs in the environment configuration window control how CODE generates the program's startup code. Some common tabs in the environment configuration window are:

Startup
: Configure the project's basic startup code generation.

IO
: Configure the standard input/output devices used by the library.

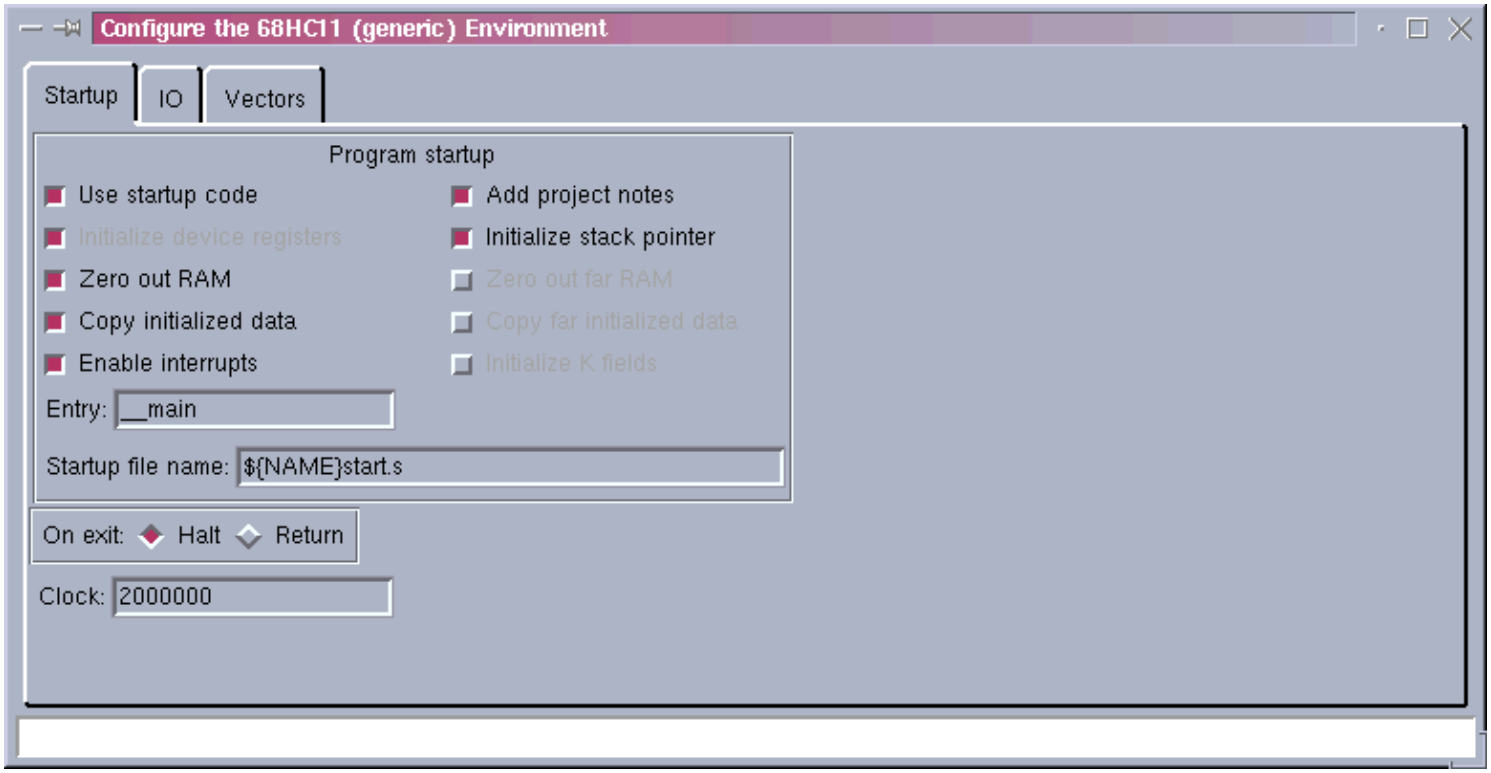
Vectors
: Configure the projects reset, interrupt, and exception vectors.

Registers
: Initial values of the various on−chip registers.

There may be additional tabs in the environment configuration window depending on the processor and variant selected for your project.

Startup

The **Startup** tab of the environment editor dialog box varies depending on the processor and variant selected, but looks something like this:



The **Use startup code** checkbox tells CODE you want to use the automatically generated startup code. The startup code will be generated into a file defined by **Startup file name**, with **${NAME}** replaced with the name of the project. If you turn off the **Use startup code** checkbox, then no startup code will be generated for your project. You can specify your own manually created startup code file in the **Tools** tab of the project editor window. Of course, you can have code CODE generate the startup code the first time, and then use and modify that startup code file directly.

The **Add project notes** checkbox, when enabled, causes the project notes to be placed in the generated startup code as comments.

The other checkboxes in this window control what standard startup code functions are included in the startup code. The **Initialize device registers** checkbox controls whether code should be generated to set the on−chip peripheral registers to known initial values. You can control these initial values by using the **Registers** tab or one of the other tabs that might exist for the processor and variant you have chosen. The device register initialization code is the first code executed in the startup code, so things like chip selects can be set up immediately.

Some processors need an initial stack pointer set, if it is not set by resetting the processor. The **Initialize stack pointer** checkbutton causes the startup code to contain the stack pointer initialization code.

C programs expect global, uninitialized variables to be set to zero when the program starts. The **Zero out RAM** checkbutton places code in the startup code to do this.

Initialized, writeable C variables have to be handled specially in an embedded system. They are copied from ROM to RAM when the program starts. The **Copy initialized data** checkbutton controls this.

The **far** variations of the previous two checkbuttons apply to 68HC16 large model programs. The **Initialize K fields** checkbutton also applies to the 68HC16: it causes code to be added to the startup code which sets the processor's K fields to the near bank.

The **Enable interrupts** check button causes the startup code to enable the processor interrupts before the main program is started.

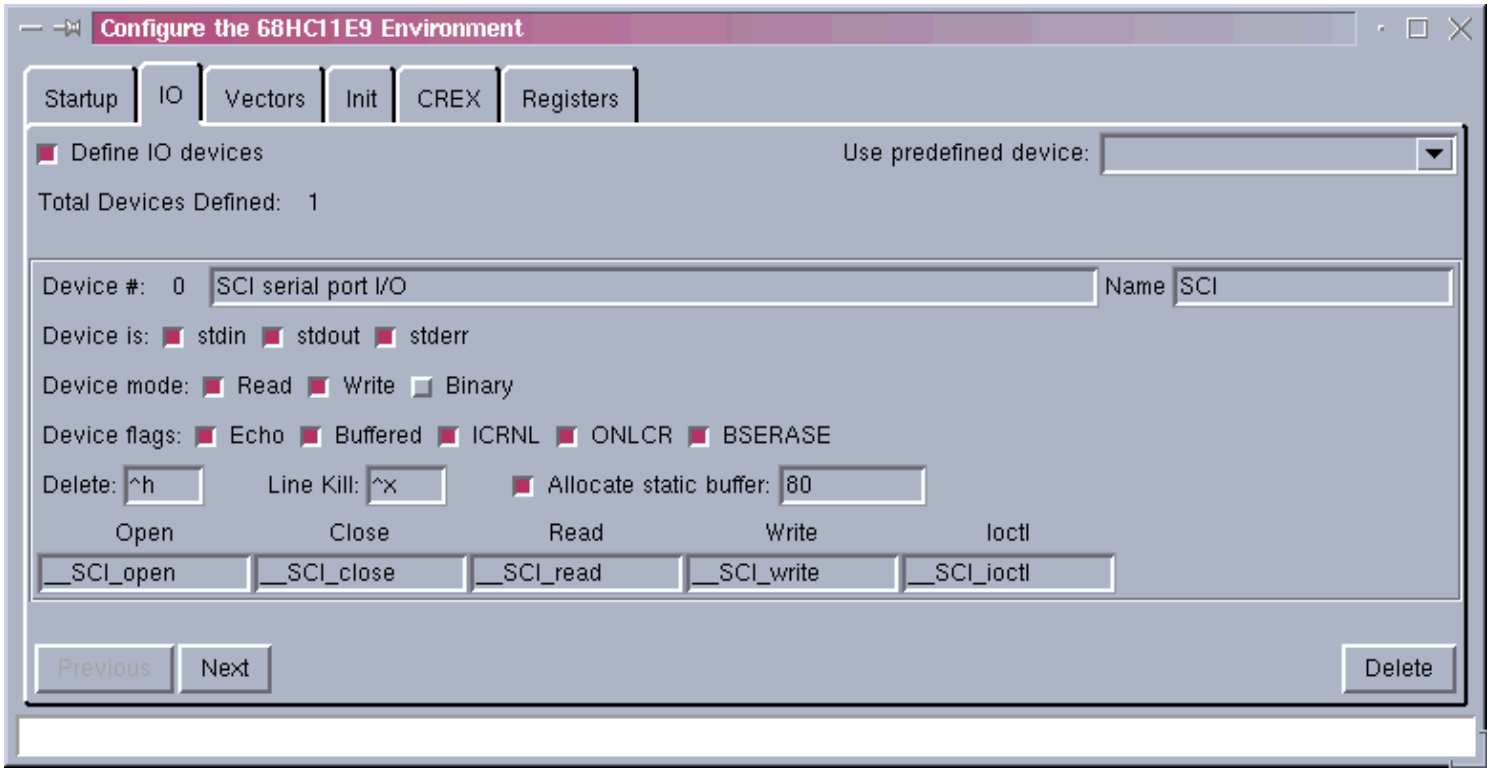
When the startup code is finished, a call is made to the entry point of the project. The name of the entry point is defined in the **Entry** box. By default, the entry point is the library function **__main()**, which initializes the standard C library and calls the function **main()**.

The **On exit** control in the **Startup** tab controls how a return from the main program or a call to **exit()** should be handled. The default action is to halt, or disable all interrupts and enter an infinite loop. If you select **Return**, the stack pointer value at program start up is restored and the program that called your program, typically a monitor or operating system, is returned to.

The **Clock** entry tells CODE the clock speed of the processor in your project. The clock speed is used by CODE to determine things like baud rates of SCI ports.

IO

The **IO** tab of the environment editor dialog box lets you set up devices that can be accessed using the **stdio** functions in the C standard library. The **IO** tab looks like this:



The **Define IO devices** checkbox causes code to be generated in the startup code file to build the device table

and initialize the standard devices. By default, one device is defined for a project, but you can add additional devices for which drivers exist in the standard library or that you write yourself. You can find more information about creating your own I/O devices in The Runtime Environment. The **Use predefined device** selector lets you choose one of the predefined drivers for a particular processor and variant.

For each device that is defined, a number, description, and name are shown at the top. The device number and description are informational. The name can be used to open the device using the standard library's **fopen()** function.

The **stdin**, **stdout**, and **stderr** checkbuttons control which device is assigned to these standard I/O streams. Only one device can be used for each of these streams. Changing the device attached to **stdout** will, for example, control where data written with **printf()** will go.

The **Device mode** checkbuttons control whether a device can be opened and used for read or write, and whether a device is character oriented or binary.

The **Echo** flag causes all device input to be echoed. The **Buffered** flag causes device input to be line buffered. The **ICRNL** flag causes input carriage returns to be converted to C newline characters. The **ONLCR** flag causes output C newlines to be converted to carriage return and linefeed pairs. The **BSERASE** flag causes a backspace input character to be echoed as backspace space backspace.

The **Delete** and **Line Kill** characters specify the character and line delete input characters respectively and are enabled if the input is line buffered.

The **Allocate static buffer** checkbutton allows you to specify a buffer to be used by the device driver. This buffer will be allocated in the startup code in the **.bss** section. By default **stdin**, when buffered, is allocated a static buffer since it is opened automatically in the startup code.

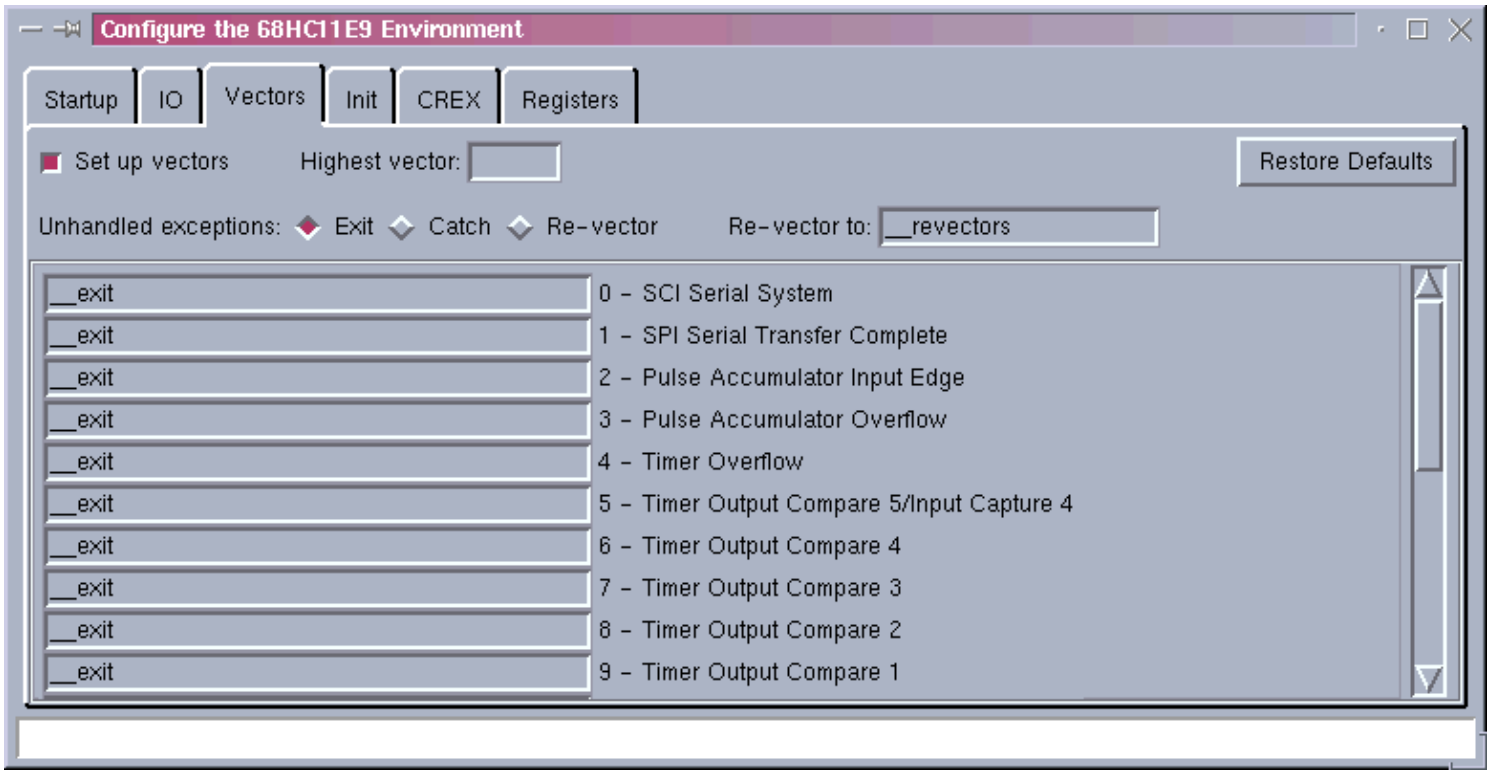
The **Open**, **Close**, **Read**, **Write**, and **Ioctl** entries specify the names of the functions that do the actual device input/output and control.

You can add a device to the device table by pressing the **Next** button until a blank device entry is shown and filling in the fields. You can fill in the fields by selecting a device from the **Use predefined device** menu or by making your own custom device driver.

A displayed device can be deleted from the device table by pressing the **Delete** button.

Vectors

The **Vectors** tab of the environment editor dialog box looks similar to this, but the actual window will vary depending on the processor and variant chosen:



The **Vectors** tab allows you to change how the current assignment of the processors reset and exception vectors. The **Set up vectors** checkbutton causes code to be generated in the startup code to make the processor vector table.

All vectors for a processor are initialized by default. If you set the **Highest vector** field you can limit the number of vectors initialized to less that the total number the processor supports to save space. This is usually useful only on processors with many vectors, such as the 68HC16 and 68K family.

Each processor vector is associated with the name of a C or assembly language function. The default value for all vectors except the Reset vector is **__exit**, the program exit point. The reset vector's default value is **__start**, the beginning of the standard startup code.

CODE considers vectors that point to the default **__exit** exit point as *unhandled exceptions*. You can control how unhandled exceptions are treated in the runtime environment with the **Unhandled exceptions** radio buttons. The **Exit** button, when selected, leaves all unhandled exceptions pointing to **__exit**. In other words, the program will disable interrupts and enter an infinite loop if an unhandled exception occurs. If you select the **Catch**, all vectors defined as going to **__exit** will be redirected to individual trap handling functions that disable interrupts and sit in an infinite loop. This is handy for catching unexpected interrupts and determining their source because each trap handling function has a unique address. Finally, you can select Re−vector to handle unhandled exceptions. This causes each unhandled exception to be re−vectored through another vector table that you can place somewhere else. This is handy for placing a vector table in RAM that can be changed at run time.

The **Re−vector** to entry allows you to specify where the unhandled exceptions should be re−vectored. By default, CODE sets this address to the symbol **__revectors**. This symbol is not defined by CODE and will result in an unresolved symbol at program build time if you don't define it. You can replace **__revectors** with another symbol or expression or with the absolute address where the vectors should be placed, e.g. 0xFFC0. The symbol **__revectorsend** will be defined if unhandled execptions are re−vectored. It is the address one

past the end of the re−vector table. If the **Re−vector** entry does not contain the symbol **__revectors**, the symbol **__revectors** will also be defined to be the value placed in the **Re−vector** entry.

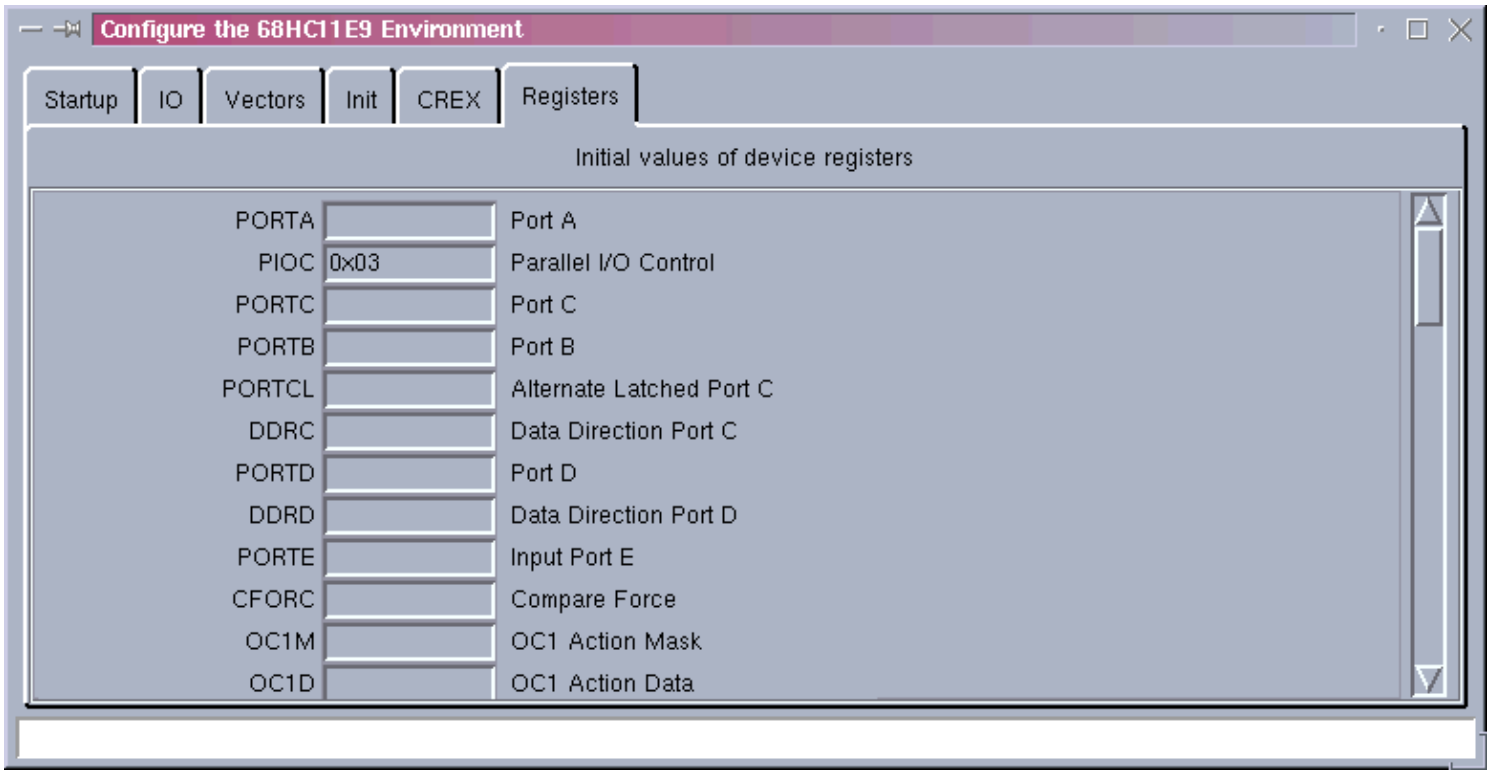
You can change the value of any of the vectors but the function called must be an interrupt handling function. In assembly language the function must end with a return from interrupt instruction and preserve any registers it uses on those processors that don't automatically save the processor's registers. In C, an interrupt handling function must be declared with the __interrupt storage class modifier:

```
__interrupt void handler(void)
        {
        ....     /* your code here */
        }
```

The **Restore Defaults** button will restore the processor default vector values.

Registers

The **Registers** tab of the environment editor dialog box looks similar to this, depending on the processor and variant selected:

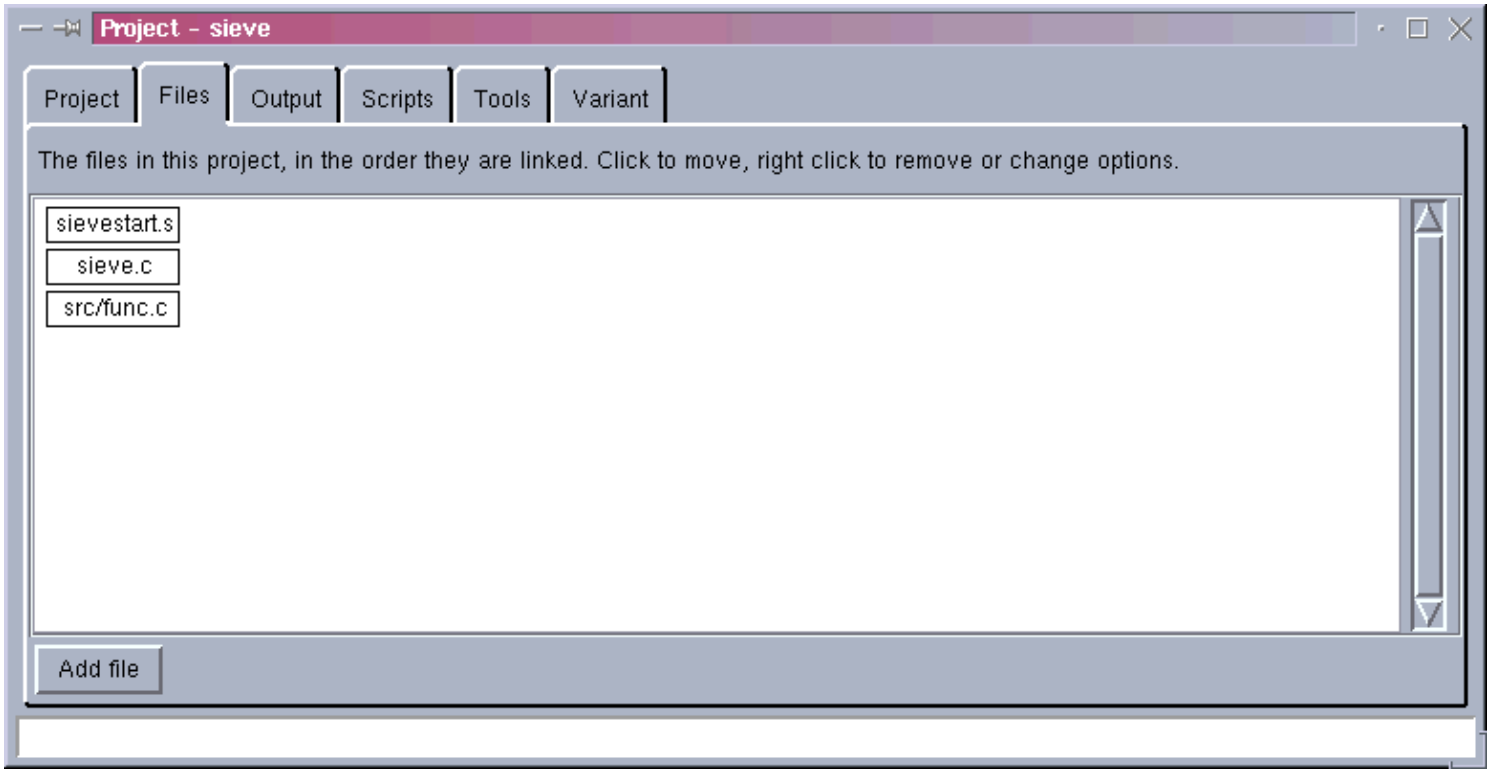


Any register values that you change will have code generated in the startup code to initialize the register to the specified value. Register initialization is done very early in the startup code so that important registers, such as time protected registers and chip select registers, are initialized right away.

Depending on the processor and variant you are using, there may be other tabs in the environment editor that modify the initial values of various registers with a more user−friendly interface.

Manipulating Project Source Files

Part of the process of defining a project is telling CODE what source files should be compiled or assembled and made into the final project result. The **Files** tab of the project editor dialog box is used to manipulate the source files that are part of the project. The **Files** tab looks like this:



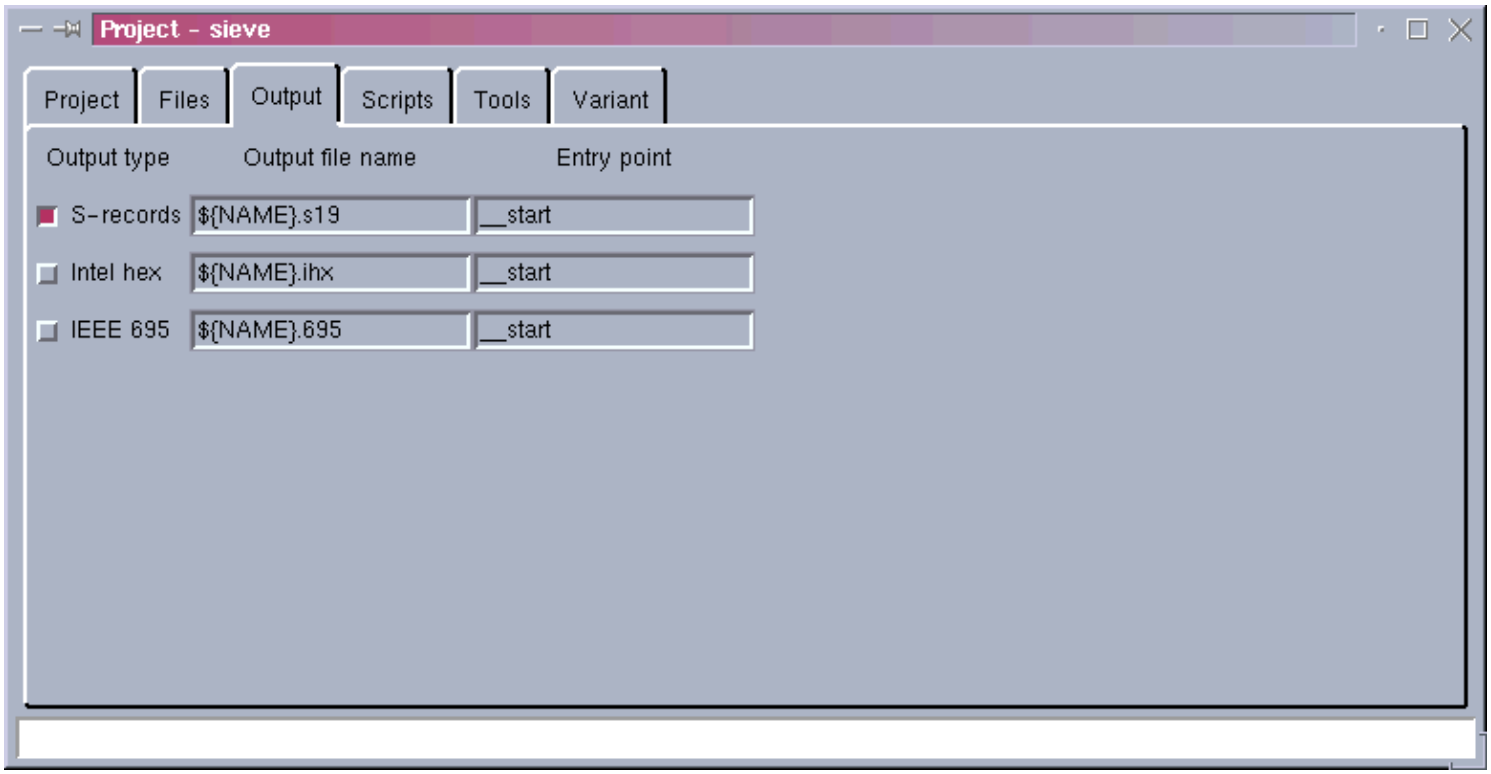
The source files in the project are shown. You can right click on a source file to perform operations on it, just like you can in the Directory window. Right clicking lets you remove a file from the project, add file specific compiler or assembler options, etc.

The files are shown in the order they will be linked together. You can click on a source file and drag it to a different spot in the file list to control the order in which your project's files are linked together. If you just click on a source file without dragging it, it will be moved above the source file immediately above it.

The **Add File** button brings up a dialog box that lets you add another source file to the project.

Specifying Project Output

The **Output** tab of the project editor dialog box is used to specify which optional output files should be created when a project is built. The Output tab looks like this:



You can select one or more of the output file types. Each output file is created using the substituted name in the **Output file name** entry. The default names create files using the name of the project and an appropriate extension for the file type (The project name is substituted for **${NAME}**). The **Entry point** entry specifies the symbolic entry point of the program and normally defaults to **__start**.

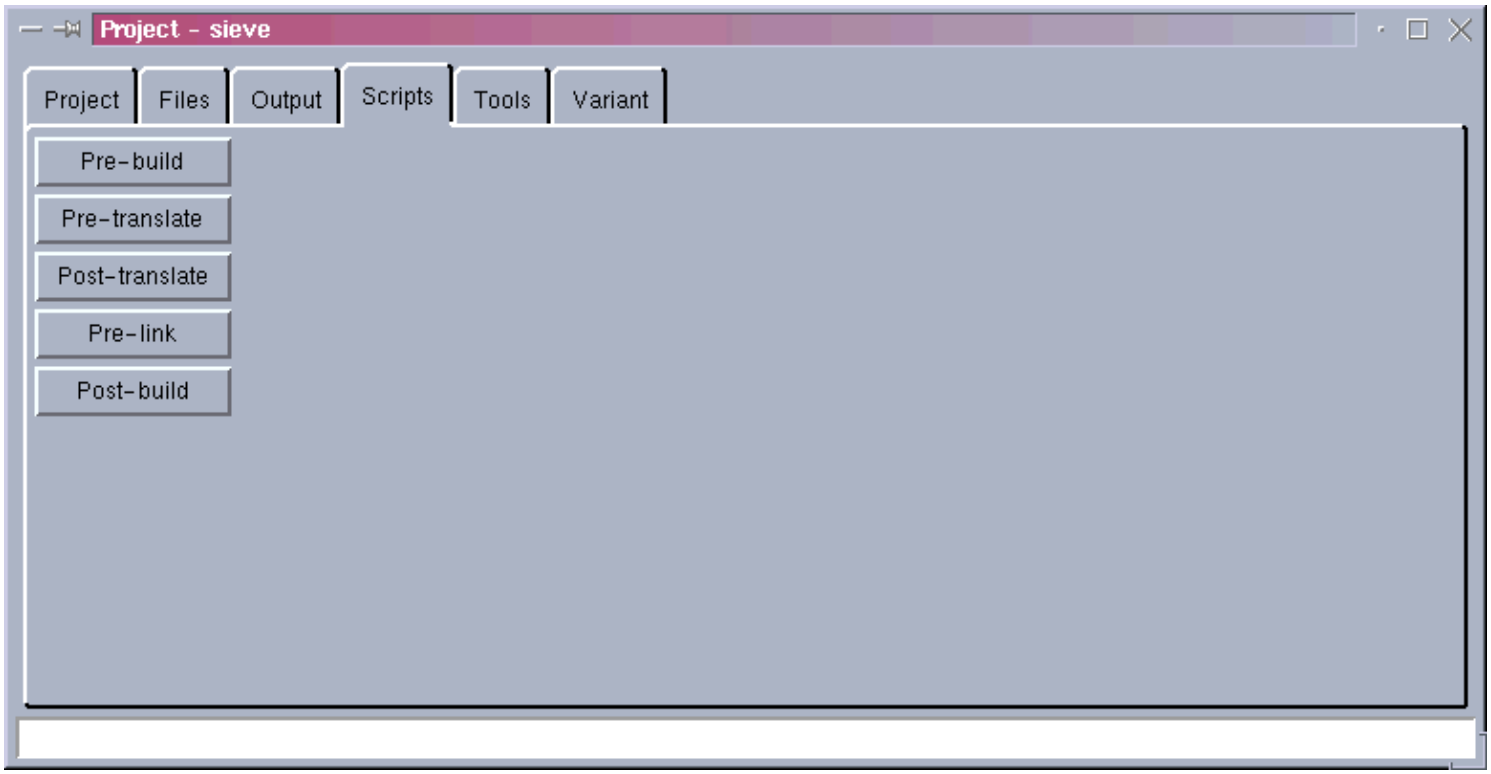
Using Build Scripts

CODE can execute several user defined *build scripts* when it builds a project. Build scripts are written in the Tcl/Tk scripting language. Build scripts can be used to process files and data before, during and after a build to perform special operations that the CODE project builder does not do directly. There are several times during the build process that you can run scripts:

- Before the build starts.
- Before a source file is compiled or assembled.
- After a source file is compiled or assembled.
- Before a program is linked.
- After a project is built.

In addition, each project source file can have file specific scripts associated with it that are run before and after the file is compiled or assembled. You can associate a script with a source file by right clicking on the file in the directory window or in the **Files** tab of the project editor.

The **Scripts** tab of the project editor is used to set a project's global build scripts, those scripts that apply to the entire project or to project files that have no specific scripts associated with them. The **Scripts** tab looks like this:



The scripts of each build share the same Tcl interpreter so variables and procedures that are defined in earlier scripts can be used by scripts run later in the build.

There are several variables set to special values during the build process that are accessible in build scripts. Some have fixed values throughout the entire build:

INTROL
: The full name of the directory where CODE was installed.

PROJECT
: The full name of the project directory.

NAME
: The name of the project.

OBJECT
: The full name of the object file directory.

These are the only variables with values when the pre−build script is run.

When a script is run before and after a source file is compiled or assembled, the following variables are set:

PROCESSOR
: The name of the processor for which the build is being done.

FILES
: The name of the source file being compiled or assembled, relative to the project directory.

ROOTNAME
: The name of the source file, without its extension.

TARGET
: The full name of the object file created.

During the pre−link script the previous variables have the following meaning:

PROCESSOR

The name of the processor for which the build is being done.

FILES

The full names of the object files being linked.

TARGET

The full name of the executable file to be created.

The post−build script defines the variables as follows:

PROCESSOR

The name of the processor for which the build is being done.

TARGET

The full name of the executable file created or empty if the project was up to date.

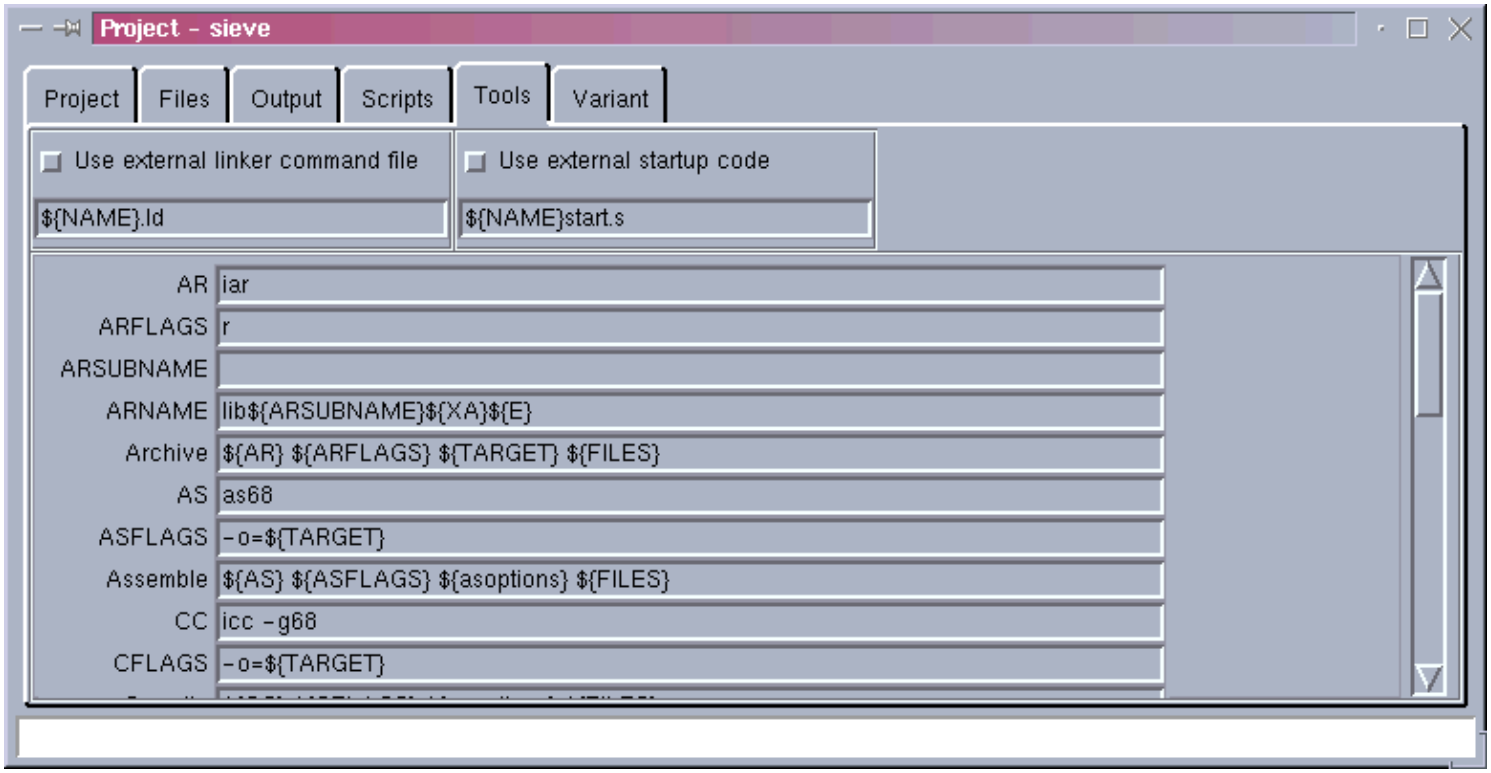
A build script can modify a build variable. This can be handy, for example, to rename a source file's object file. In the pre−translate script for a source file, which you can set by right clicking on the file in the directory window, you can have somthing like:

```
set TARGET new${TARGET}
```

which will name the object file for **foo.c newfoo.o11** rather than the default **foo.o11**.

Setting Tool Defaults

CODE *tools* are the command line programs that are used to build projects. The **Tools** tab of the project editor dialog box is used to define how the tools are used by CODE. The **Tools** tab looks like this:



The checkbox in the upper left lets you specify a pre−existing linker command file for a project. This might be handy if you have already created a linker command file or have special requirements for your linker command file that the CODE Memory Map dialog doesn't handle. If you choose to use a pre−existing linker command file, the memory map dialog is disabled since its contents are not used when building a project. You can specify you own name for the linker command file, or use the default value, which is ***project name.*ld**.

The checkbox in the upper right lets you specify a pre−existing startup code file for a project. If you choose to use a pre−existing startup code file, the **Environment** dialog is disabled since its contents are not used when building a project. You can specify you own name for the linker command file, or use the default value, which is ***project name*start.s**.

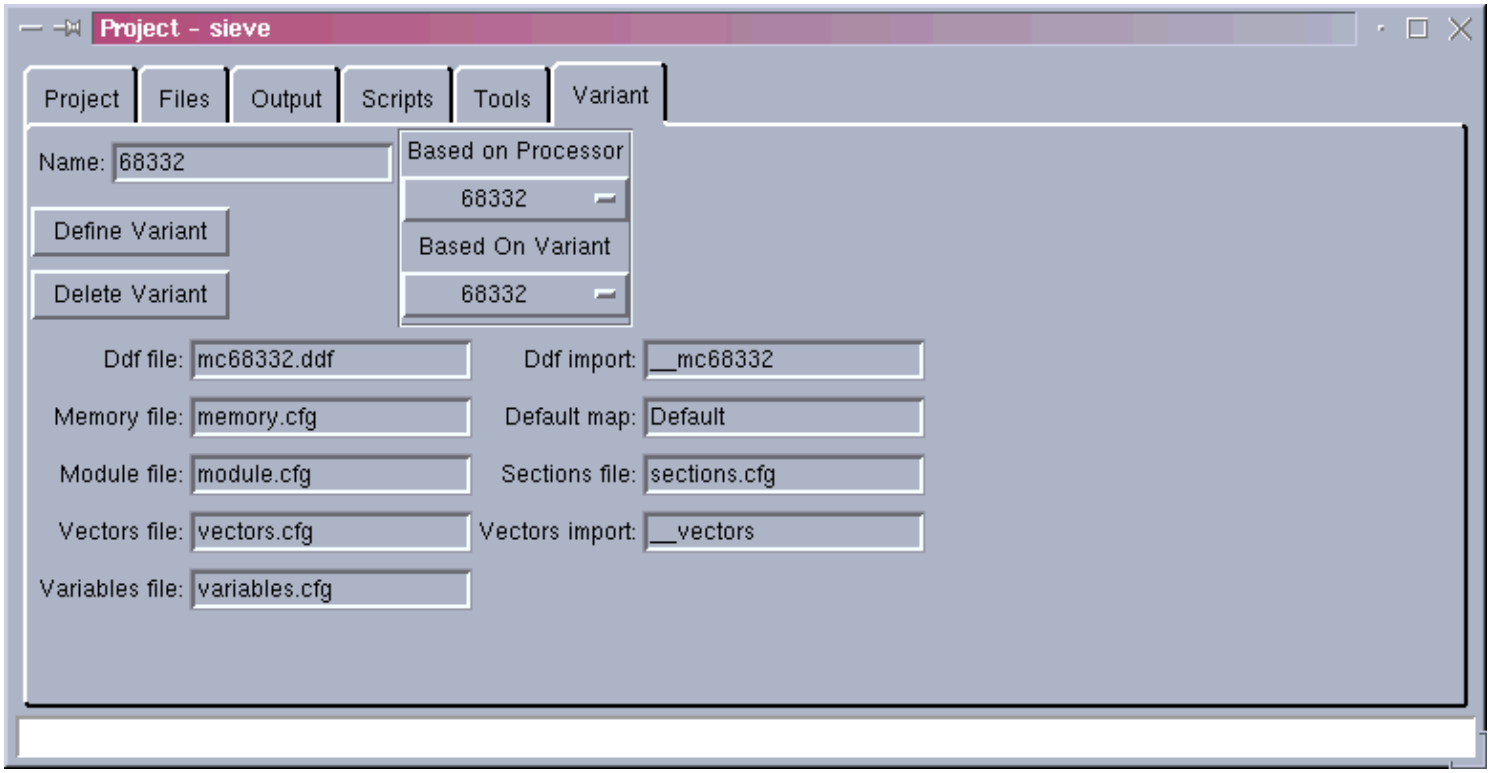
You should never have to change the remaining tool definitions in general. Sometimes it is necessary, however, to override the CODE default values to handle special build situations.

Adding a Processor Variant

Many of the microcontrollers and microprocessors supported by CODE come in several *variants* that have different complements of on−chip peripherals, RAM, ROM, etc. It is possible to add a variant that has not been supplied with CODE.

When a variant is defined, CODE can automatically generate symbolic register definitions, both for C and assembly programs, and also generate register initialization code to set the processor up in a known state at reset. You can also add a user interface to a variant definition that lets you change the register initialization values with nice dialog boxes rather than raw hexadecimal values.

The **Variant** tab of the project editor looks like this:



You add a new variant by basing it on a pre−existing variant. Many times a pre−existing variant will be very close to the new variant and minimal changes will be required. The **Name** field defines the variant name that is displayed in variant menus. This name is arbitrary, but should be unique and ideally should be the actual part number of the chip.

The **Define Variant** button will define a new variant or replace the definition of an existing variant. You can only define new variants: CODE predefined variants cannot be changed. The **Delete Variant** button will

delete the currently selected variant. Again, you cannot delete predefined variants.

The various files used to configure a variant are all text files. The predefined variant files are in the directories **Libraries/Assembly/gen*XX***. A new variant can use some or all of the files that a preexisting variant uses. If you need to change any of the preexisting files, you can copy them from the library directory to your current project directory and modify them with the editor.

The **Ddf file** is the main variant *device definition file*. It is processed by CODE to produce register symbol definitions, C header files, user interface dialogs, etc.

A new variant will probably need a new **.ddf** file. You can add a **.ddf** file to your project and CODE will do all the processing required to use the **.ddf** file.

The **Ddf import** is a symbol that is used to import the symbol definitions of a **.ddf** file if they are placed in a library.

The **Memory file** is the default memory maps for the processor and variant. The memory file does not generally have to be changed for a new variant: You can change the project's memory map in the memory map window. The memory file just defines one or more default memory maps that can be used in the memory map window.

The **Default map** is the name of the map that is considered the normal map for a variant.

The **Module file** is used to define configuration parameters for certain modules in the runtime library. You generally don't have to change this for a new variant.

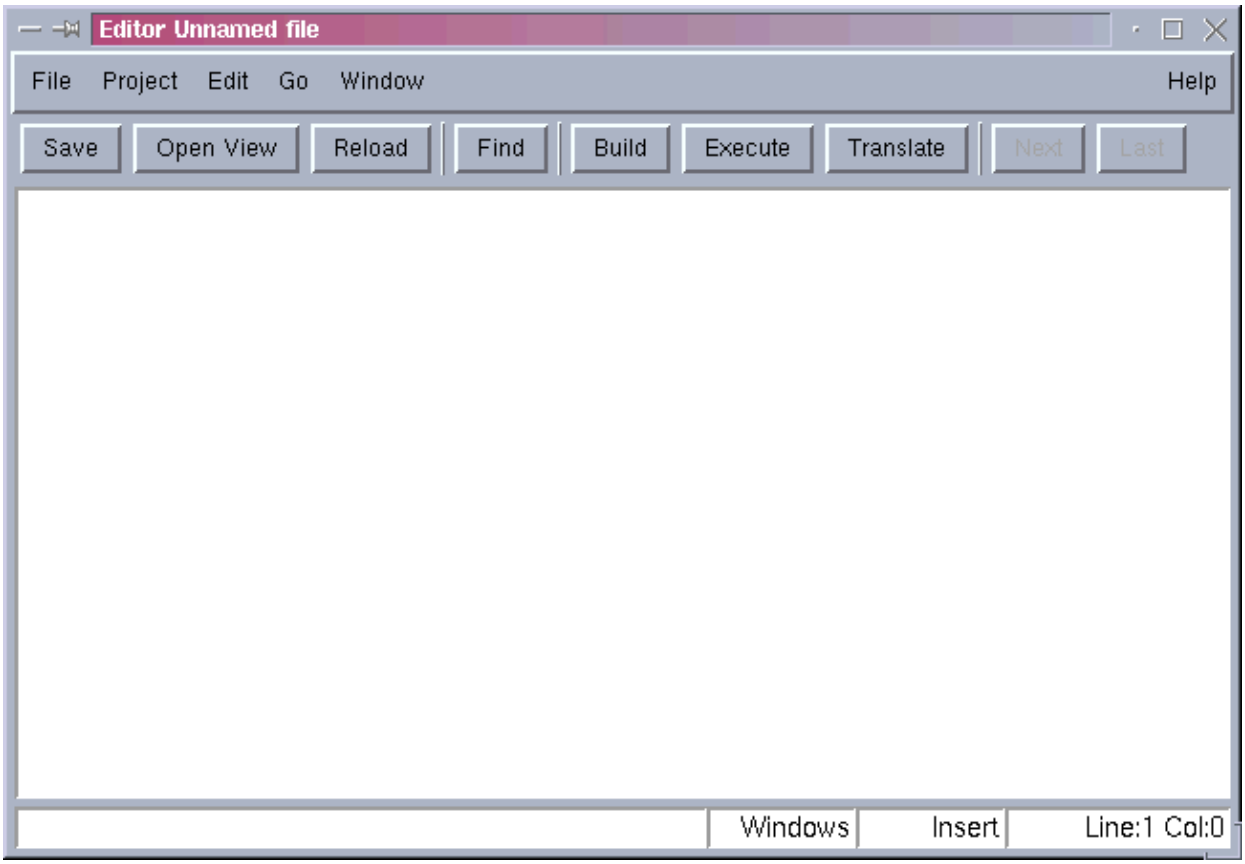
The **Sections file** defines the common sections used by CODE for the processor and variant. These definitions rarely change.

The **Vectors file** defines the processor vectors. This file will only need to be changed if your new variant has different reset, exception, or interrupt vectors than any existing variant. The **Vectors import** symbol is used to import the vector table if it is placed in a library.

The **Variables file** defines linker variables that are defined when a program for the processor and variant is linked. This file should never have to change for a new variant.

The Editor

The CODE editor is a multi−windowed editor for creating and modifying source files. You can also translate source files into object files from the editor and it will walk you through the source to correct any errors that may have occurred. When the editor window is first opened, it looks like this:



Most functions that Editor adds to its menus are self explanatory but a few need to described.

In the **File** menu several new entries have been added. The **Reload** entry, and the Reload button on the button bar, will reload the current file. The **Open View** entry, and the **Open View** button on the button bar, will create another Editor window open to the same file. Any changes made to any editor window will be made in all windows that are open to the same file.

The **Save** and **Save As...** entries work normally, the **Save Copy As...** entry will save a copy of the current file being edited without changing the name of the current file.

The **Add to Project** and **Remove from Project** entries add or remove the file being edited from the current project.

In the **Edit** menu, The **Undo** entry will undo the last change to the file, the **Redo** entry will undo the last **Undo**. The **Find...**, **Replace...**, **Goto...**, and **Mark...** entries will bring up the standard text searching dialogs.

The **Match Brace** entry will find the next **}**, **)**, or **]** if the cursor is to the left of a **{**, **(**, or **[**, respectively or the last **{**, **(**, or **[** if the cursor is to the left of a **}**, **)**, or **]** on the same line. Whenever a brace is matched, the text between the braces is momentarily highlighted.

The **ToUpper** and **ToLower** entries convert any selected text to upper or lower case respectively.

The **Go** menu contains has entries for recently edit files and for all the source files in the current project. If you select one of the Go menu entries, the selected file will be open.

The editor keeps track of your last position in a file and will position the cursor at that position is a file is reopened.

The **Check** button will check the current file for errors. If any errors occur when the program is being checked, the first error is displayed in the status bar and you can use the **Next** and **Last** buttons to navigate through the source to examine (and fix!) the errors.

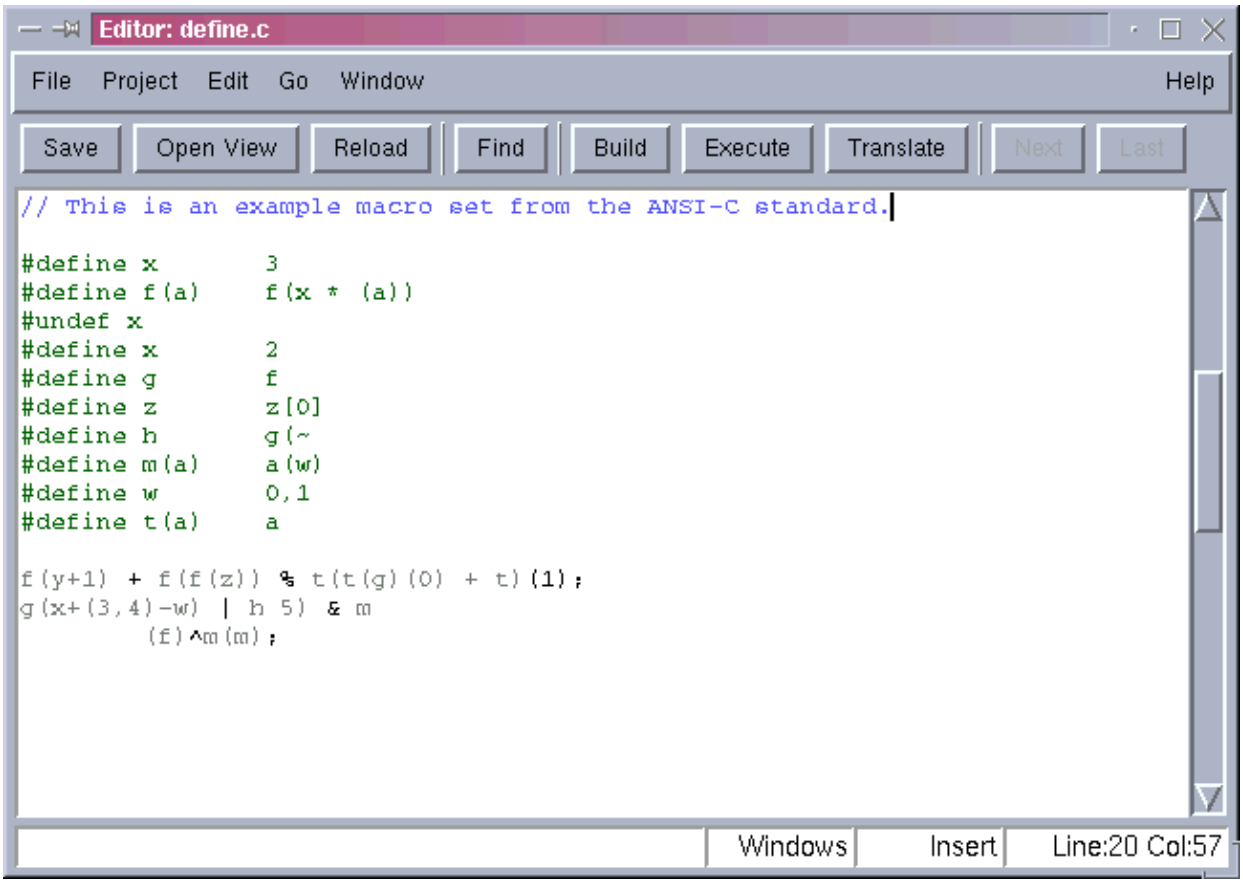
Build will build the current project. If the file you are editing is a part of the current project, it will be translated also. If errors occur during translation **Build** lets you navigate through the errors like **Check**.

The **Execute** button will build the current project and then execute the project's program output in the debugger.

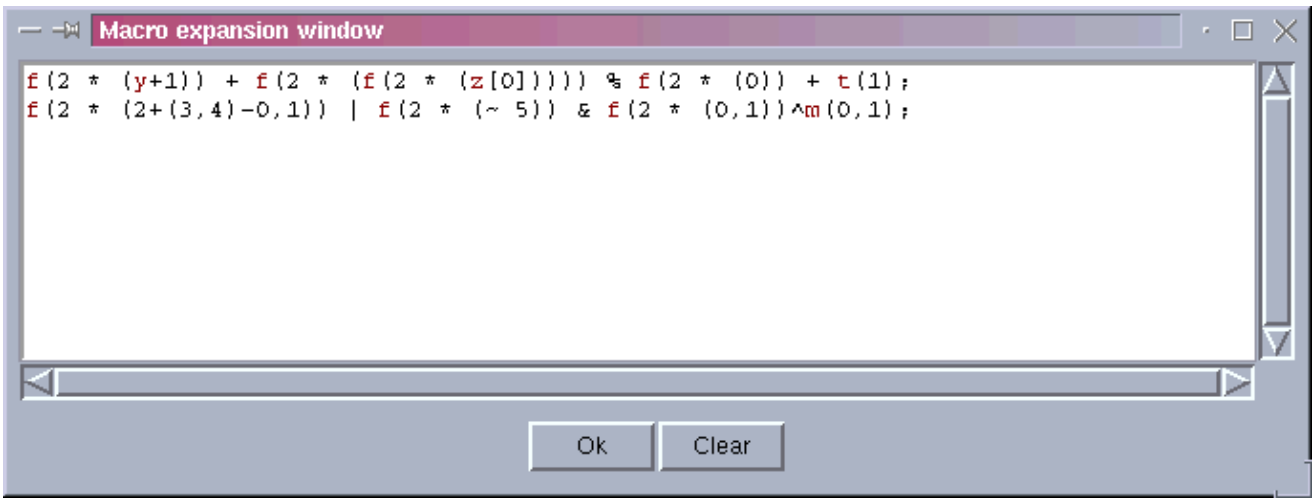
When you are entering text using the editor, a line will automatically indented to the level of a previous line when you press **Enter**. An undo, via the **Edit** menu or **Control−Z**, after an **Enter** will move the cursor back to the beginning of the new line.

You can right click in the editor window to perform useful operations like deleting a line and inserting a new line. In addition, when you are editing a C source file, for example, right clicking on the word *printf* or any other standard library function name will allow you to open the manual page for that function. In assembly language, right clicking on an assembly opcode or directive will bring up the manual page for that opcode or directive.

You can also right click on a C macro call to view its expansion. For example, this window has some complicated macros and macro calls:



If you right click on a line containing a macro call one of the menu entries is **Macro expand line**. If you select that entry for both lines in this example, a Macro expansion window appears showing the expanded lines:



You can also view the expansion of an individual macro in the status line by passing the mouse cursor over the macro call.

The Debugger

The debugger window lets you execute and debug target applications. The debugger has several *views* that can be displayed. You can open multiple Debugger windows and each one can display its own view of the application. The debugger supports several *targets*, including simulators, the Introl Monitor Interface (IMI), and various others depending on the processor selected.

Overview
: Using the debugger, its menus and controls.

Views
: The debugger's views and how to use them.

Breakpoints
: Setting and editing breakpoints.

Registers
: Viewing and modifying the processor registers.

Goto
: Executing the program to a certain point.

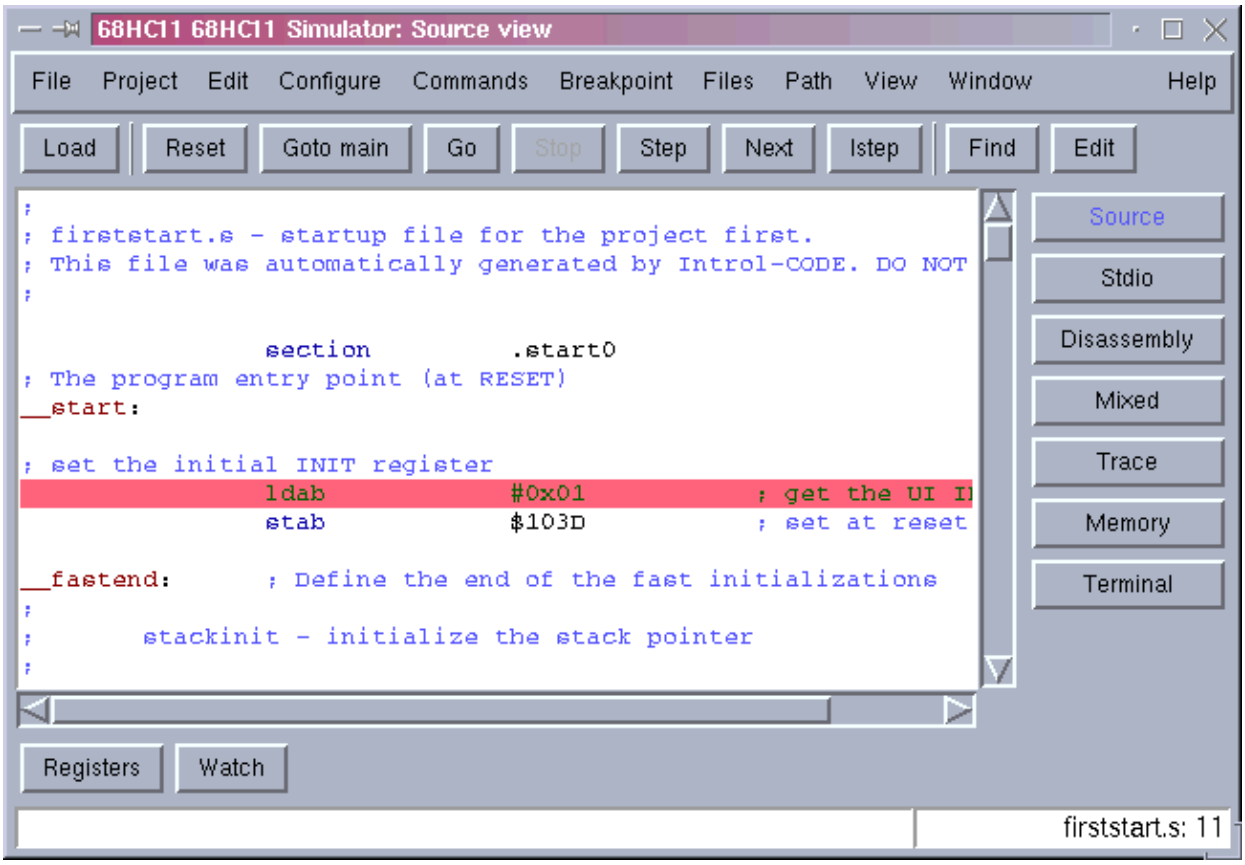
Watch
: Watching the value of program variables and expressions.

Modules
: Creating and using debugger scripts.

Targets
: The targets supported by the debugger and how to use them.

Overview

The CODE debugger can be used to execute and debug programs for various microcontrollers and target environments. The debugger supports source level debugging of both C and assembly language programs. When you first enter the debugger after building a project you'll see a window that looks something like this:



This window shows the program loaded but not yet running, since the **Stop** button is disabled and the **Go** button is enabled. The first few lines of the startup code for the project are displayed, with the current line shown in green. The title bar of the window tells which processor (68HC11) and debugging target (68HC11 Simulator) is being used as well as which debugger view is shown (Source view). The debugger view can be selected from the buttons on the right side of the debugger window, the **View** menu, or the by the keyboard shortcut for each view which is displayed in the **View** menu.

Various aspects of the Debugger's operation can be modified by selecting **Preferences** under the **File** menu to open the Preferences window and choosing the **Debugger** tab.

If you are viewing the debugger window manually you can use the **Open...** entry in the **File** menu to open an executable program. An program has to be loaded into the target environment with the **Load** button or by selecting **Load** from the **Commands** menu.

If you have a project open and the program created by that project has been built then the debugger will automatically open and load that program when you enter the debugger window. If the project program is rebuilt the debugger will automatically reload it. These automatic operations can controlled from the Preferences window.

The **Configure** menu allows you to set the processor and variant you want to debug and the target environment the debugger should use. You can also open the Debugger Modules window from the Configure menu to specify debugger scripts that should be automatically loaded by the debugger.

You can perform the following operations by pressing the appropriate button on the button bar or by choosing the same entry from the **Commands** menu.

Reset
: Reset the target processor.

Goto main
: Go to the symbol **main** in the program. In a C program this is the start of your program, after the initialization code has been executed.

Go
: Start the target processor.

Stop
: Stop the target processor and display update any open views with the application's state.

Step
: Step the processor through a single source line. **Step** will step into any functions that are called in the source line.

Next
: Step the processor through a single source line. If the source file is written in C and the line contains one or more function calls, **Next** will execute the called functions at full speed.

Istep
: Step the processor through a single machine instruction.

Edit
: This will open the Editor window with the current source file.

The first two buttons below the **Debugger** view area are **Registers** and **Watch**. The **Registers** button opens up the register window displaying the processor's registers. The **Watch** button brings up an expression watch window that lets you view and modify the value of variables and expressions.

The **Commands** menu has entries for all of the above and also has a **Goto...** entry which allows you to type in a function name or line number. **Goto...** will start the target program and execute until the point specified in the **Goto...** dialog box has been reached, another breakpoint occurs, or the **Stop** button is pressed.

All of the commands in the **Commands** menu have keyboard shortcuts that are listed next to their entry in the menu.

Debugger modules may add other buttons to the bottom button bar.

The **Breakpoint** menu allows you to set and clear program breakpoints. The **Files** menu shows the name of all the source files that were used to build the project. If you select one of the names then the debugger will display that file in the **Source view**.

The **Path** menu is used to control where the debugger will look to source files that make up a project. Normally all source file names in a project are saved without full directory paths. The debugger will the use the paths defined under the **Path** menu to find project source files for display.

Views

The debugger has several *views* that can be displayed. The current debugger view can be selected from the buttons on the right side of the debugger window, the **View** menu, or the by the keyboard shortcut for each view which is displayed in the **View** menu. You can open multiple debugger windows and each one can display its own view of the application.

Source
: View a program's source when debugging.

Stdio

View a program's input/output when debugging.

Disassembly

View a disassembly of the program.

Mixed

View the program source with each line disassembled.

Trace

View the trace buffer.

Memory

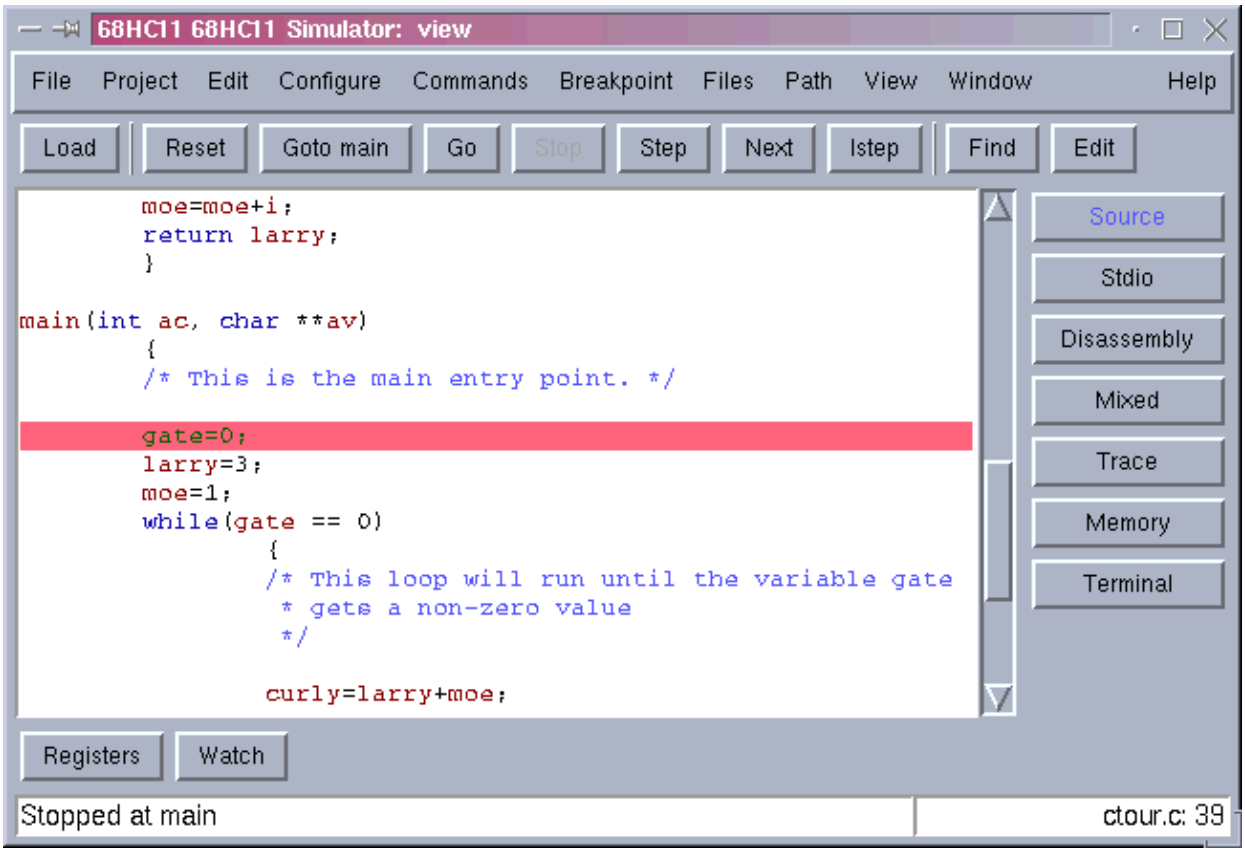
View and modify memory contents.

Terminal

Open a serial port for terminal input/output.

Source

The **Source** view of the debugger window view displays source code from the current program. Whenever program execution stops the Source view displays the source file where execution stopped with the current line highlighted in red. You can also select a line in the source window by clicking on the line with the left mouse button. This will highlight the line in green. This line is called the *selected line* and you can use the **Breakpoint** menu to go to or set a breakpoint on that line. You can view other source files that make up the project by selecting the file from the **Files** menu. The file currently in the Source view can be edited by pressing the **Edit** button.

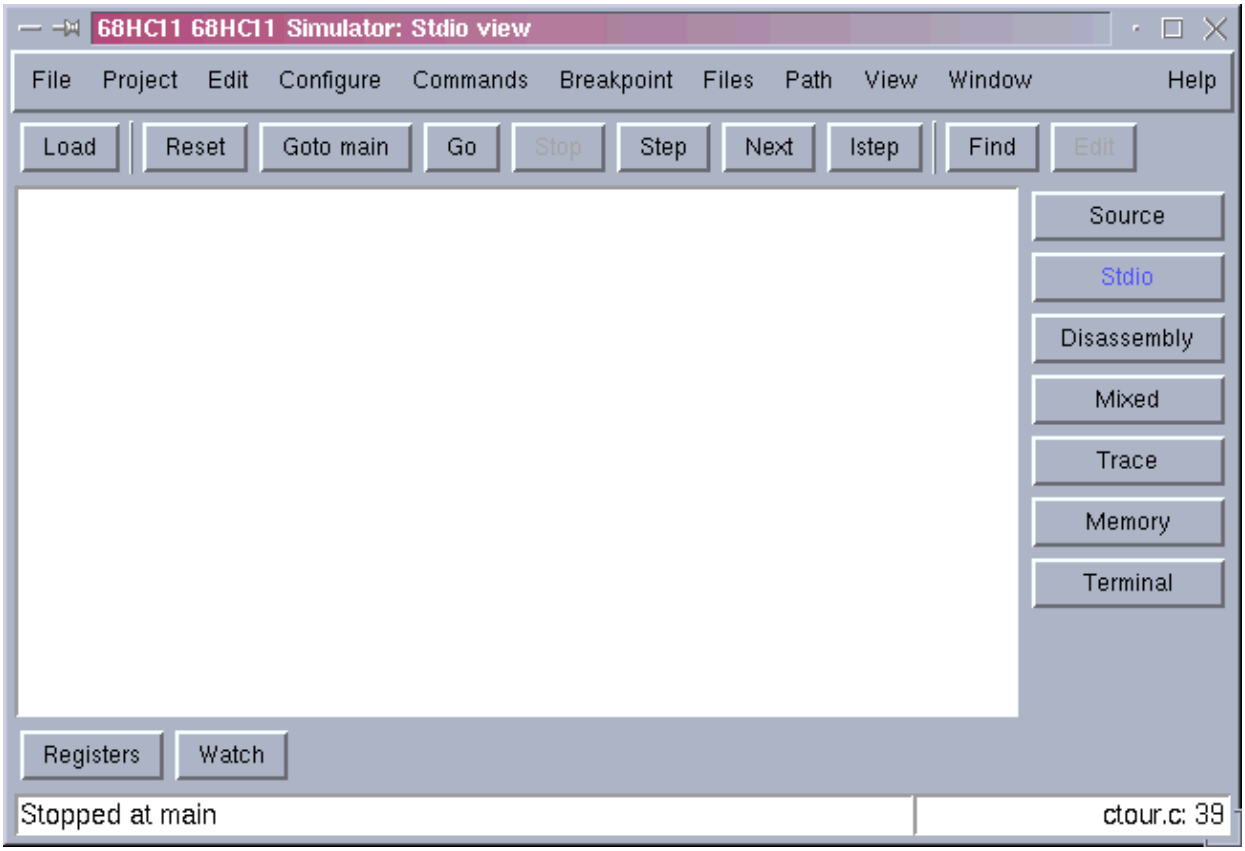


The right mouse button is very handy in the source view. You can use it to set a breakpoint on a line or go to that line. If you have selected a function name you can set a breakpoint at that function or go to that function. If you have selected an variable name or an expression you can view the value of the selected item in the status bar. You can also set the program counter to the address of the selected line.

If you right click on the name of a C standard library function or assembly opcode or directive, you also have the option of opening the appropriate manual page.

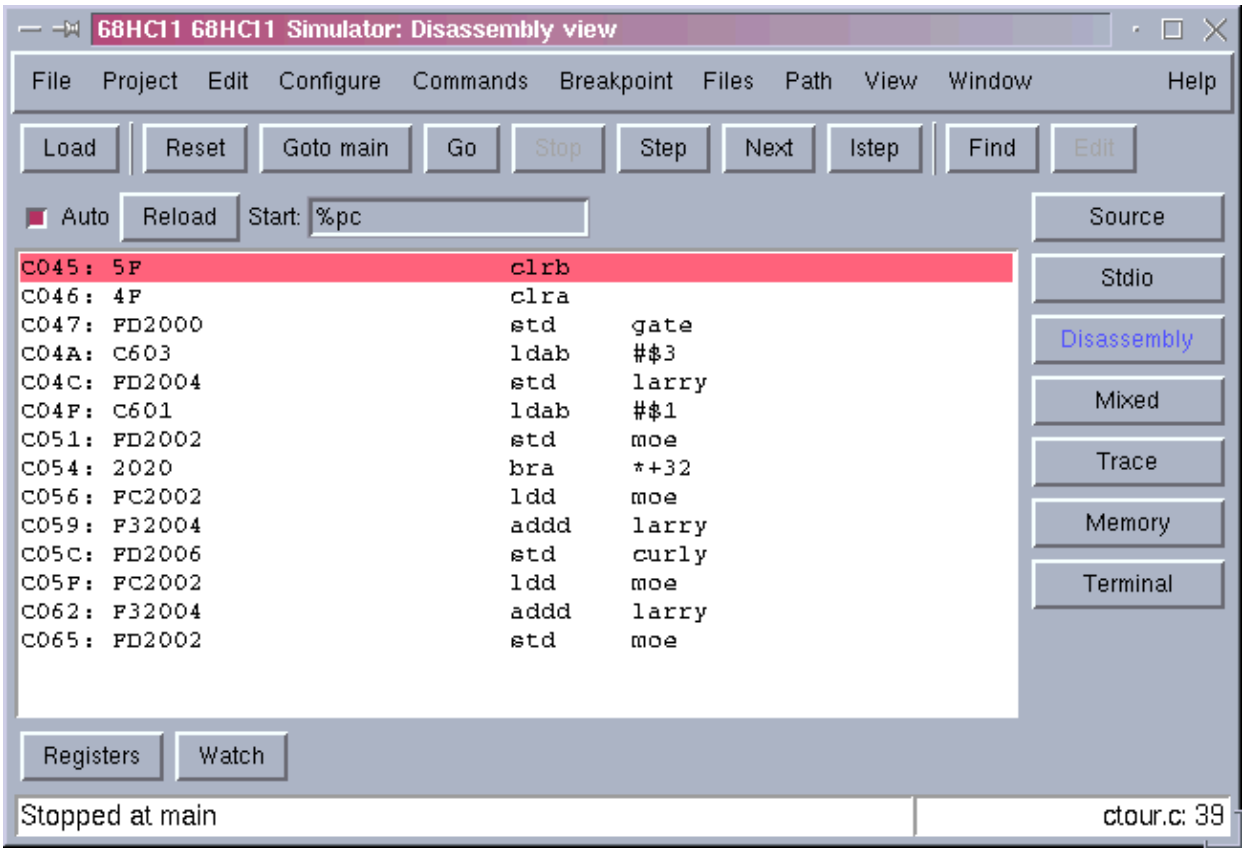
Stdio

The **Stdio** view of the debugger window will display the output of the running program and allow you to type input. This view is active only when appropriate breakpoints trap that target program's I/O requests.



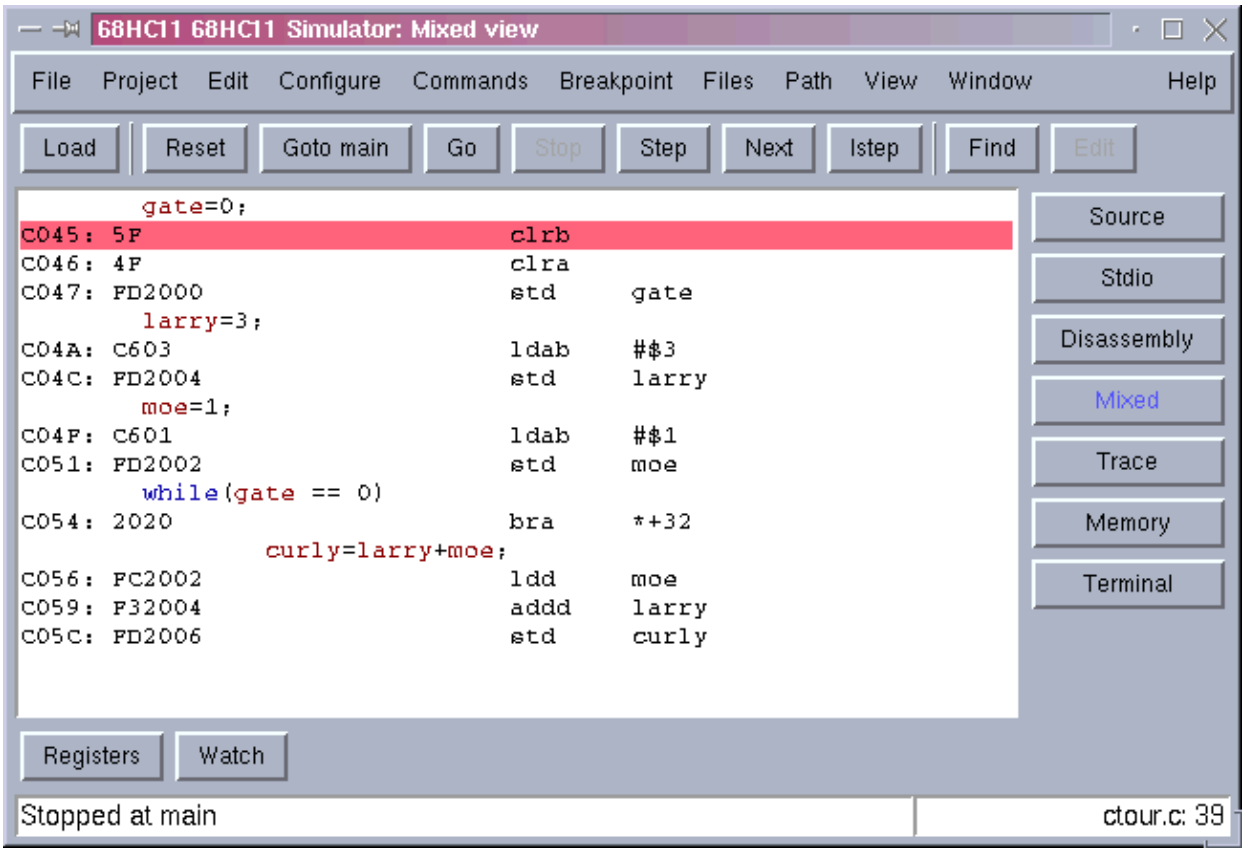
Disassembly

The **Disassembly** view of the debugger window shows the disassembly of the current program at the point where execution last stopped. The **Page Down** and down arrow keys will display the next screen full and next line of disassembly respectively.



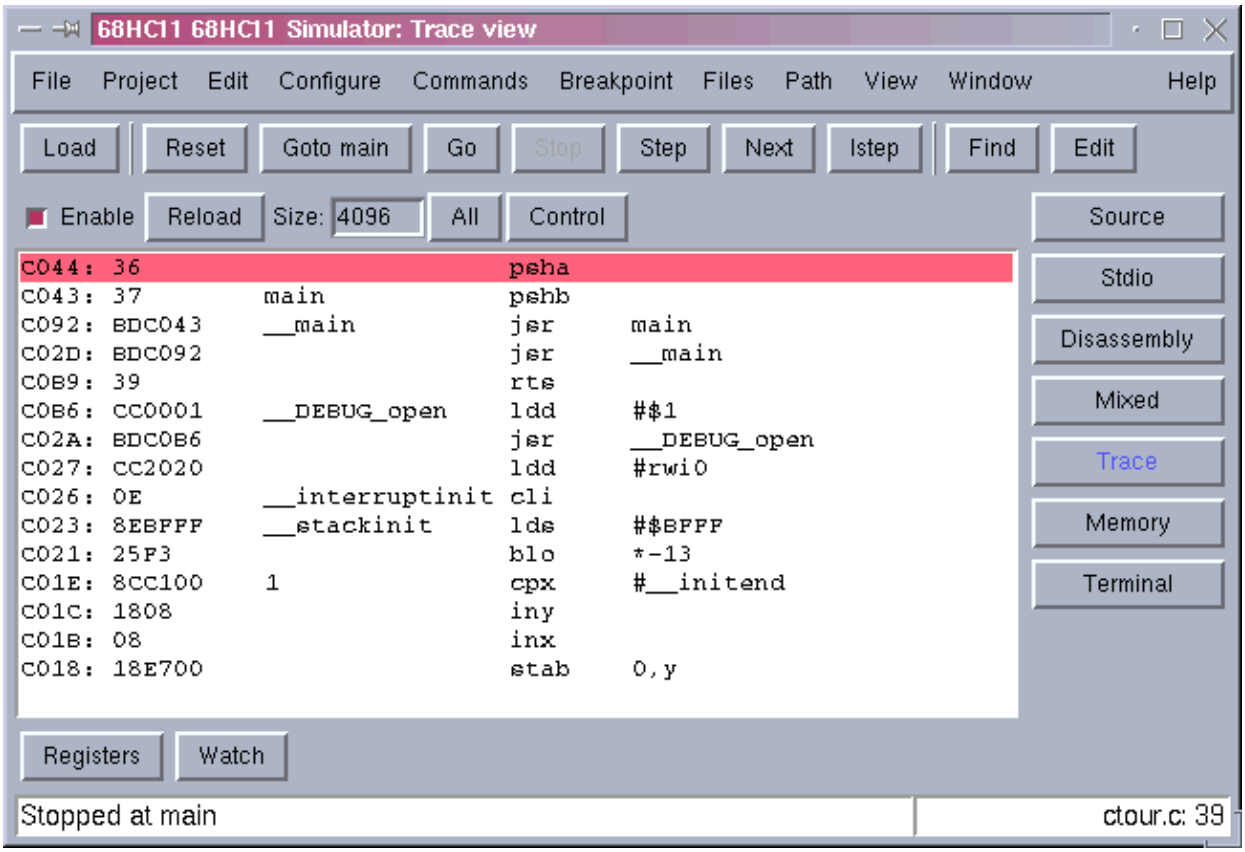
Mixed

The **Mixed** view of the debugger window shows the program source code followed by the disassembly of each line. This window is updated whenever the program stops.



Trace

The **Trace** view of the debugger window shows the contents of the target environment's *trace buffer*. Not all target environments support the trace view. The trace buffer is a circular buffer of the last instructions executed. The trace buffer size depends on the target environment. It defaults to 4096 instructions for the simulators. The trace view must be **enabled** for tracing to become active. The Trace view is updated whenever the program stops and shows a disassembly of the last instructions executed, in reverse order or the when the **Reload** button is pressed.



The **All** button loads the entire trace buffer into the Trace view.

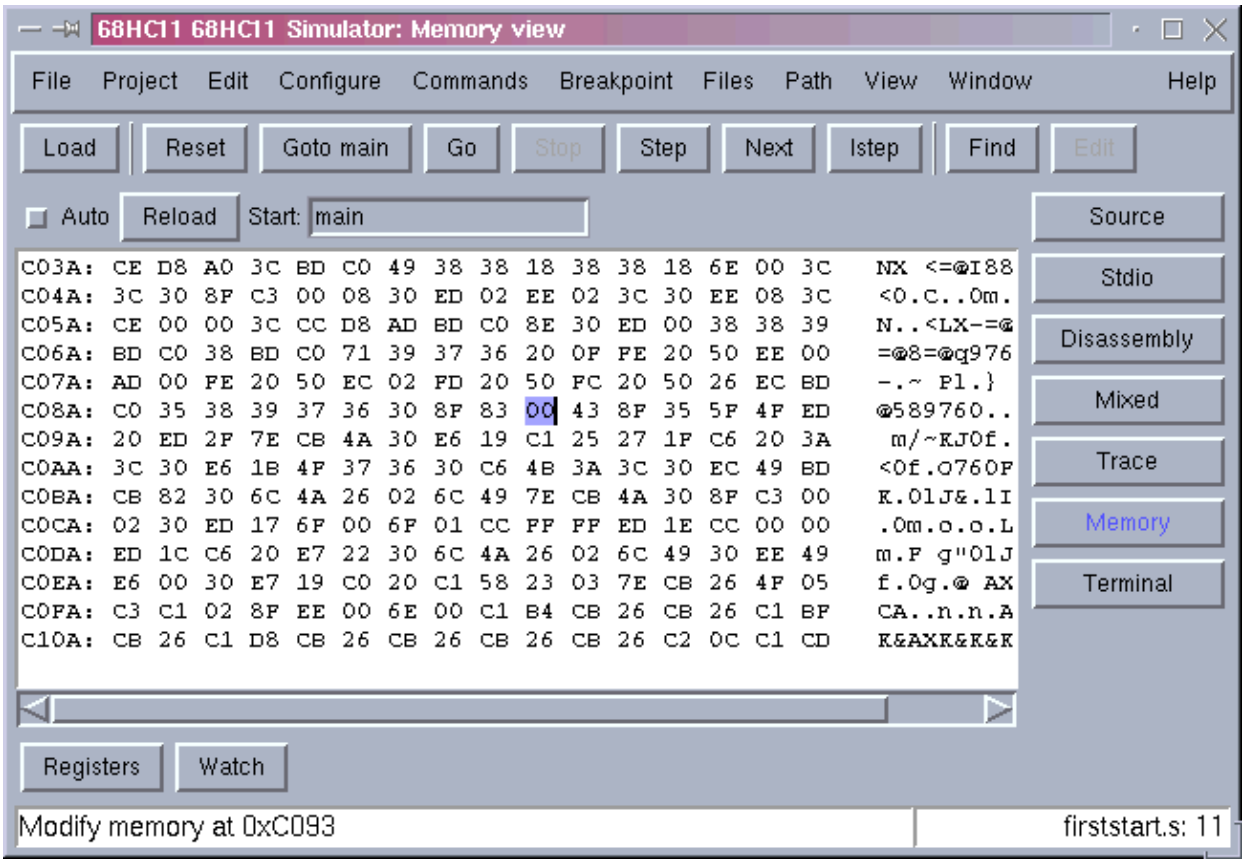
Some target environments, such as the simulators, support executing programs from the trace buffer. When these environments are active, the **Control** button brings up a window with two sets of buttons to control execution out of the trace buffer.

The **Backward** buttons execute the program backward in the trace buffer. You can single step by machine

instruction or source line. If you **Go** the program backward, execution will continue until the end of the trace buffer is reached, a breakpoint is encountered, or the **Stop** button is pressed. The **Forward** buttons can be used after backward execution to re−execute trace buffer instruction.

Memory

The **Memory** view of the debugger window displays an area of memory in hexadecimal and ASCII. If the **Auto** checkbox is on, then the memory window will be updated whenever the program stops. The Memory view is updated in **Auto** mode whenever the program stops or when the **Reload** button is pressed. It will display memory from the start address until the view is filled.



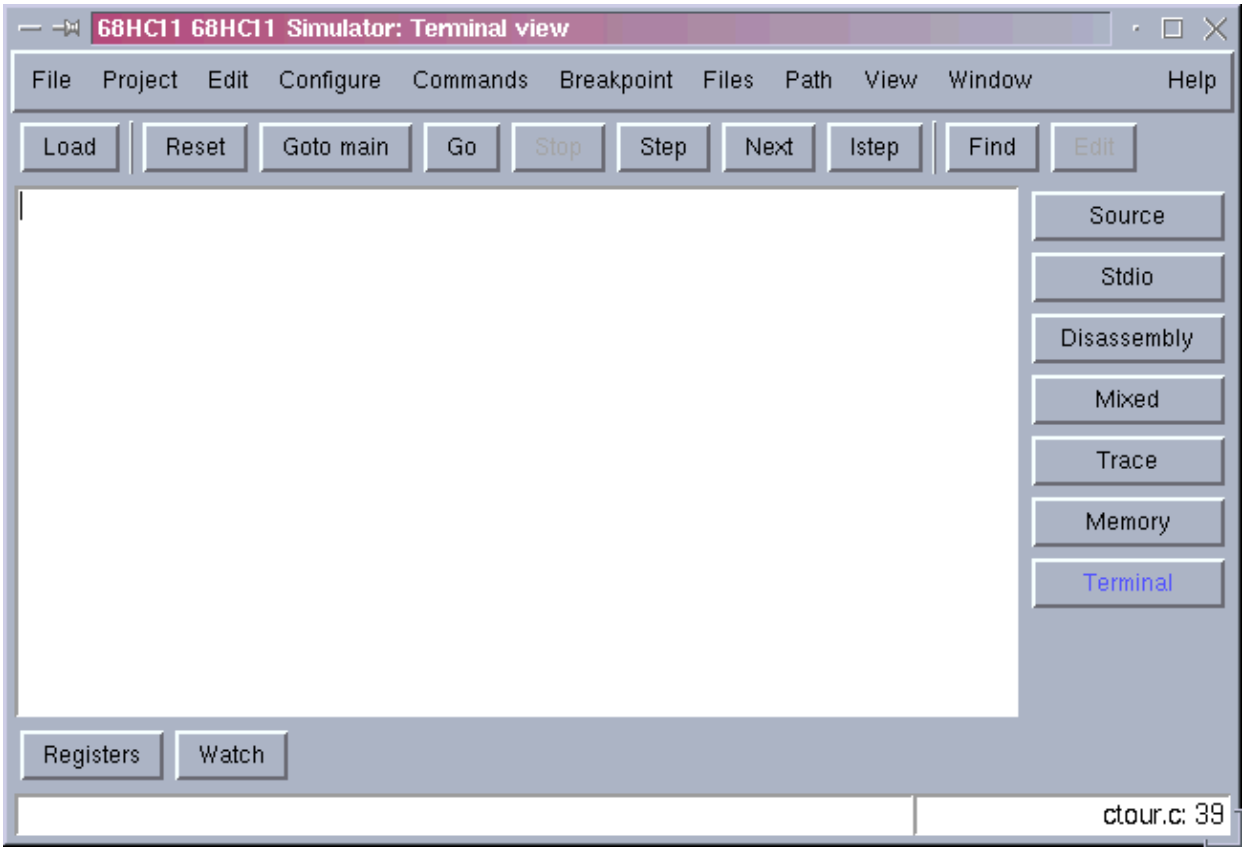
The start addresses may be a complex expression. For example to dump a named memory location or a C variable, ou can use a notation like **&name**. To display the processor stack every time the program stops, use the start expression **%s** or **%sp** depending on what the stackpointer is called on the target processor. See the description of debugger expressions for more information.

The memory window can also be used to modify memory. Place the cursor over the memory location you want to change and type the new value.

The **Page Up** and **Page Down** keys will display the last and next screenfull of memory and the up and down arrow keys will scroll the memory display one line.

Terminal

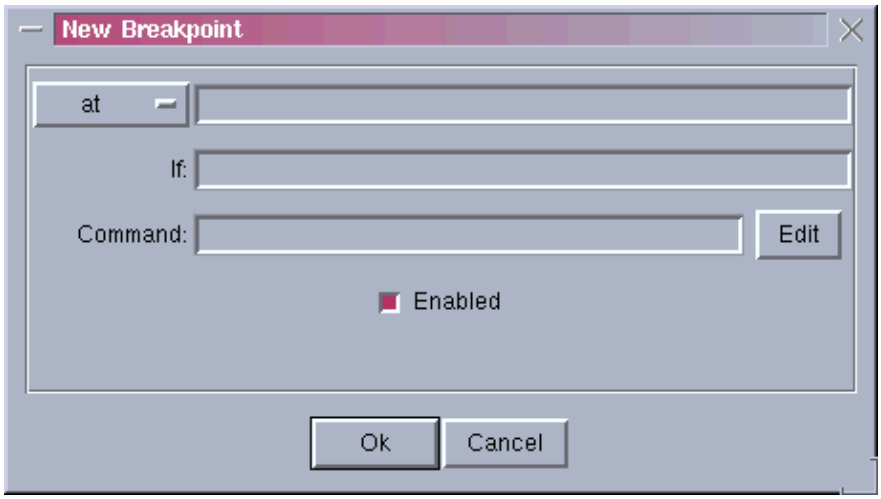
The **Terminal** view of the debugger window allows you to specify a serial port which you can use as a connection to your target.



The debugger will ask you for a serial port to use when you first enter the Terminal view. You can change the port at any time by selecting **SetupTerminal** from the **Configure** menu.

Breakpoints

The **Debugger Breakpoint** window allows you to create and edit a program *breakpoint*. A breakpoint is a condition that you define that causes your program to interrupt its operation and perform a specific action.



A breakpoint has a *type*. Depending on the target environment you are debugging with, CODE will support some or all of the possible breakpoint types. The simulators support all breakpoint types. The breakpoint types supported by CODE are:

at
: The breakpoint occurs when a program executes the specified address.

read
: The breakpoint occurs when a program reads from the specified address(es).

write
: The breakpoint occurs when a program writes to the specified address(es).

access
: The breakpoint occurs when the specified address(es) are read or written.

clock
: The breakpoint occurs after the specified number of processor clock cycles have been executed.

enabled
: The breakpoint occurs when the interrupts are enabled or disabled by the program.

The breakpoint target can be an arbitrary debugger expression. It represents an address in the case of **at**, **read**, **write**, and **access** breakpoints and a value in **clock** and **enabled** breakpoints.

When a breakpoint target is an address, an unadorned integer is assumed to be a line number in the current file. To set a breakpoint at a specific integer address, precede the integer with the address of operator:

```
&0xE0000
```

For **read**, **write**, and **access** breakpoints that are made on C program variables, the debugger will set the breakpoint on every byte on the variable.

The expression in a **clock** breakpoint expression represents an integer number of cycles.

The expression in an enabled breakpoint is evaluated and if it is odd, the breakpoint occurs when interrupts are enabled. Otherwise the interrupt occurs when the interrupts are disabled.

The **if** expression, if present, is evaluated when the breakpoint occurs. If the expression evaluates to a zero

value, the breakpoint is ignored:

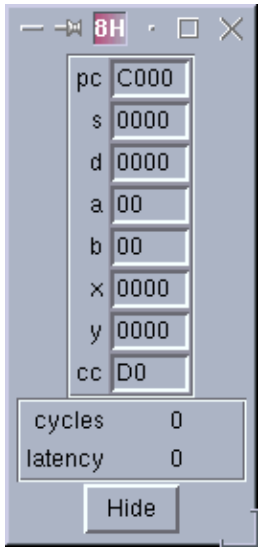
```
i == 5
```

will only breakpoint if the value of variable **i** in the program is equal to 5.

If no command is given for a breakpoint, the default action is to stop the running program. A command, if specified, can be an arbitrary Tcl script that will be executed when the breakpoint occurs.

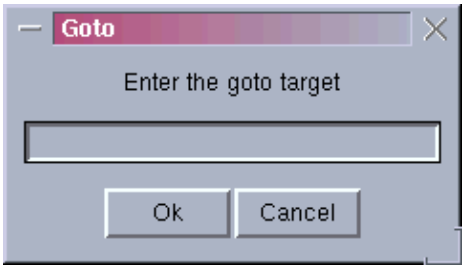
Registers

The debugger register window allows you to see and modify the value of the target processor's registers. A typical example of a register window looks like this:



This window will be updated whenever the target processor is stopped by the **Stop** or **Reset** buttons or a breakpoint. You can modify a register's value in the register window. The registers are displayed in hexadecimal. The **cycles** and **latency** counters, which are enabled for certain debugger targets such as the simulators, display the number of processor cycles that have executed and the maximum number of processor cycles that interrupts have been disabled since they were first enabled, respectively.

Goto



The **Goto** window allows you to specify a target to execute to. You can go to a line number in the current file by specifying a line number. You can go to a line number in a different file by using the @ operator:

```
filename@100
```

will go to the file that has the base name *filename*. The debugger will search in the standard places for the file and will use the first source file that matches *filename*, regardless of extension. You can specify a full path and extension by putting *filename* in quotes:

```
'filename.c'@100
```

You can go to a C function or assembly language label by giving it's name:

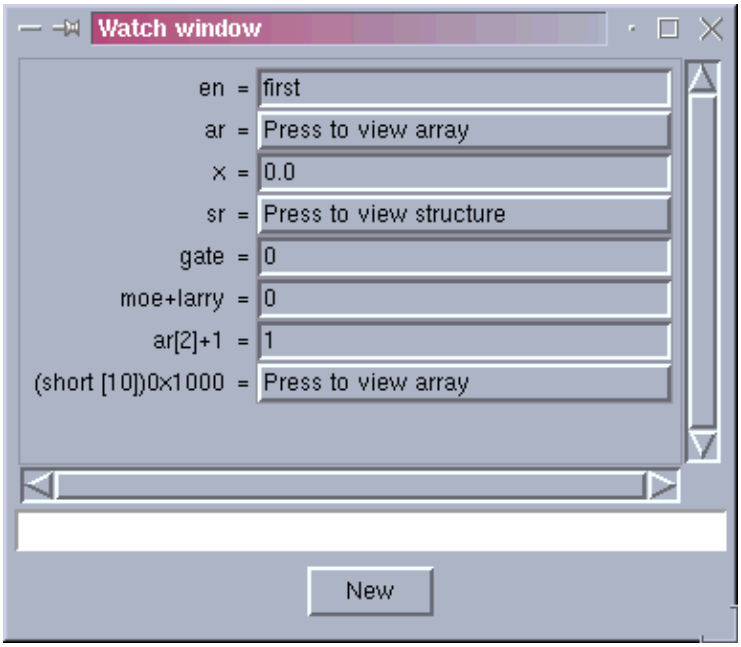
```
main
```

This will find all **extern** names and **static** names in the current file You can use the . (period) operator to specify a static name in a different file:

```
filename.function
```

Watch

The Debugger Watch window allows you to see and modify the value of variables and expressions. You create a new watch by entering a *watch expression*. A watch expression can be a simple variable name or a more complicated debugger expression. The watch expression will be evaluated whenever the target processor is stopped, like the register window, and the value of the watch expressions are displayed. If a watch expression evaluates to the address of a modifiable memory location then you can change that location by changing the value in the watch window. You can use arbitrarily complicated expressions in the watch window.



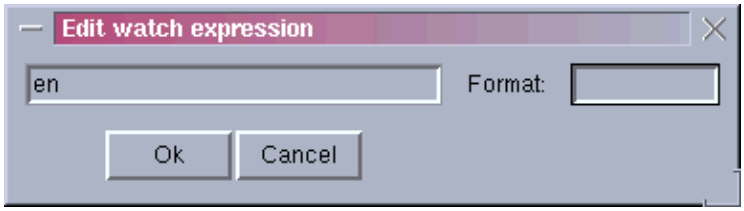
When the watch expression evaluates to an array, structure, union, or pointer, you can open a new watch window containing the elements, members, or data pointed at respectively. The example above also shows

more complex watch expressions: **a[2]+1** and **(short [10])0x1000**. The second complex expression displays memory at address 0x1000 as an array of short integers. More complex watch expressions can be any legal C expression.

An easy way to set a watch expression from the debugger source view is to select the expression or variable to be watched and then right clicking the mouse and choosing **Watch**.

You can edit or delete a watch window entry by right clicking on the watch expression for the entry.

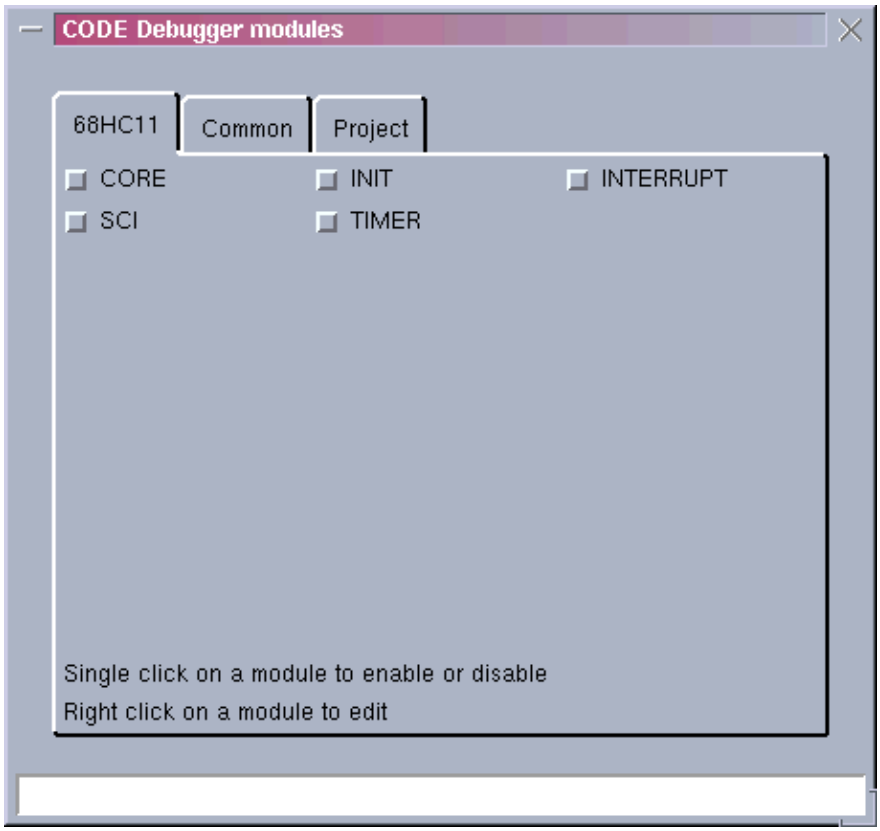
Expressions are normally displayed in a format consistant with the way they are declared in C. You can change the format of a displayed expression by right clicking on the expression and choosing **Edit**. This will display a watch expression edit window:



The format field in the edit window accepts a C **printf()** format string. For example, to display an integer as hex you can use the format string **0x%04X**.

Modules

The **Modules** window allows you to specify debugger scripts that should be loaded automatically by the debugger. There are tabs in the Debugger Modules window for modules specific to the current processor, modules common to all processors and projects, and modules specific to your project.



You can select one or modules to be load by single clicking on the module button. If you double click on the module button an Editor window will open with the contents of the script.

Debugger modules are standard Tcl/Tk scripts that have access to several commands specifically added to make writing scripts easier. In addition, the debugger calls certain predefined procedures in scripts at appropriate times to allow a script to handle different debugger and target program actions.

By convention, all module procedures and variables should be preceded with the name of the module: the CREX module has a variable named CREXregister and a procedure named CREXstart, for example.

The additional command available to debugger scripts are:

BR command ?arguments?
: Set a processor breakpoint.

BUTTON name action ?help?
: Create a button for a module.

BYTE expression ?value?
: Read/Write a byte from/to memory.

DELETEBUTTON ?buttons?
: Delate a module's button(s).

EXPR expression
: Evaluate a debugger expression.

HELP widget string
: Define balloon help for a widget in a module's user interface.

INFO short description
: Information that is displayed about a module in the Module configuration window.

LONG expression ?value

Read/Write a long from/to memory.

MODULE name
Load a debugger module.

PORT address
Set the target parallel I/O communication port.

PREFERENCE name ?value?
Read/Write a module preference.

PROCESSOR
Get the current processor being debugged.

REG register ?value?
Read/Write a processor register.

RESET
Reset the target.

SERIAL
Set the target serial I/O communication port.

STDIN handler
Set up a stdin handler.

STDOUT char
Send a character to the Stdio window.

TARGET
Get the current target type.

TK
Enable the Tk GUI toolkit in the module interpreter.

UI name window
Load a user interface into the module interpreter.

UPDATE
Update the debugger state.

VADR expression
Get the address of an expression in the target.

VAR name ?value?
Read/write a debugger variable.

VARIANT
Get the current processor variant.

WHERE expression
Get the program position for an address.

WORD expression ?value?
Read/Write a word from/to memory.

The predefined procedures that are called my the debugger are named *modulename***xxx**. Where *modulename* is the name of the module and **xxx** is added by the debugger. It is not an error for a module not to contain one of the predefined procedures.

The predefined procedures are:

modulenamestart
Called when a module is loaded by the debugger.

modulenameloaded
Called when an executable file has been loaded by the debugger.

modulenamedelete
Called when a module is being deleted by the debugger.

modulenamereset
Called when the target processor is reset.

modulenameupdate
: Called when the target environment's state may have changed, for example, after a running program is stopped.

Targets

A debugger *target* is the environment in which debugging is done. All processors are supported by simulator and IMI targets. Some processors are supported by addition targets, such as BDM (Background Debug Mode).

Simulator
: Software simulators.

IMI
: Introl Monitor Interface.

BDM
: P&E ICD or public domain BDM (**68HC16** and **68332** only).

PEBDM12
: P&E CABLE12 BDM (**68HC12** only).

SDI
: Motorola Serial Debug Interface (**68HC12**, **68HC16**, and **68332** only).

Software Simulators

Introl Monitor Interface

The Introl Monitor Interface (IMI) is an software interface between the CODE debugger and a target system. IMI communicates through a serial port on the host and target systems and lets you download a program and do full source level debugging with no additional hardware. There are projects to build IMI and programs to run under IMI in the IMI demo directory. You can choose IMI as the target environment by using the debugger **Configure** menu under **Target**.

The following descriptions will be for the 68HC11, the most common IMI target. For other processors, you can replace the 11 in the following file names with the two digit code for your processor. On different processors the memory map may change, also.

If you go into the IMI directory under the demo directory you'll see two project files: **imi11.code** and **prog11.code**.

To build the monitor, **imi.e11**, just open the **imi11.code** project and press **Build**. IMI is now in the C library, so all you'll see generated is **imi11start.s** and the **imi.e11** file. **imi.e11** is meant to be burned into an EEPROM, so you may have to create a hex file for your EEPROM burner using the **Output** tab of the project editor.

If you look at the **Memory Map** and **Configure Environment** windows you'll see several things:

1. IMI uses the on−chip RAM for R/W storage.
2. IMI's code is addressed at $E000.
3. The entry point is set to **__imimain**, to give IMI control when the processor is reset.
4. The startup code for IMI re−vectors the unused interrupt vectors to the top of the memory area called **PROGRAM**. IMI uses the SCI and SWI interrupt vectors by default.

5. The PROGRAM memory area is supposed to be RAM on your target and is where a program being debugged should be loaded.

The IMI in the C library uses the SCI port by default. It does this by using the **stderr** device defined in the **IO** tab of the **Configure Environment** window. If you want to use a different serial device, just define it as **stderr** in the **IO** tab. Don't forget to change the SCI interrupt back to **__exit** in the **imi11.code** project and set the appropriate interrupt handler for your serial device. We have supplied the source to IMI so you can modify it if necessary. In addition to imi.c, each processor has its own specific trap handling code to save and restore the processor registers. For the 68HC11 it is in Libraries/Assembly/gen11/trap.s.

The trap handers are entered when a debug port serial interrupt occurs, to interrupt a running program, and when an execution breakpoint occurs. CODE causes an execution breakpoint by replacing the instruction that is at the breakpoint address with a **SWI** (for the 68HC11) or similar instruction. Each processor's trap.s handlers save the processor registers in a specific order known by the debugger when a trap occurs. This is how the debugger gets access to the processor registers. When a program is continued, using the **g** command described below, the processor registers are restored.

Burn an EEPROM and put it in the target system. Open the **Terminal** view with the appropriate communication port. When you power up, you should see the IMI prompt. The IMI prompt looks like:

IMI *savedsp returnpoint buffersize*

Where *savedsp*, *returnpoint*, and *buffersize* are hex values used by the debugger. *savedsp* is the most interesting of these values as it is the address where the trap handler has saved the processor's stack pointer. This is used by the debugger to read and modify the copy of the processor registers saved by the trap handler.

IMI doesn't have human friendly commands, but you can try something like **r 0 10** to dump 10 (hex) bytes from location zero. All IMI commands and arguments can be entered in upper or lower case. Command arguments are hexadecimal values. The IMI command set is:

g
: Start the target processor.

p
: Pause the running program.

z
: Reset the target processor.

w address count byte byte ...
: Write *count* bytes to memory at *address*.

r address count
: Read count bytes from address.

f startaddress endaddress value
: Fill memory bytes from *fromaddress* to *toaddress* with *value*.

m toaddress fromaddress count
: Move *count* bytes of memory from *fromaddress* to *toaddress*.

In addition to these commands, IMI has binary read/write memory commands that are used by the debugger to speed up communications.

If these commands work, you are now ready to use CODE and it's debugger to build and debug a program under IMI.

The **prog11.code** project shows how to build a program to run under IMI. Its memory map is set up so that all code and data is in IMI's **PROGRAM** area. The vectors of a program that runs under IMI are placed at the

top of the **PROGRAM** memory area. Since IMI re−vectors the interrupt vectors, your program can be built as if the processor's vectors are actually at the top of the **PROGRAM** area.

This means that you can run your program under IMI (including setting up interrupt vectors with the **Vectors** tab of the **Configure Environment** window) for debugging. When you want to make your program stand−alone, just change its memory map to put the code in high memory, rebuild, and burn an EPROM.

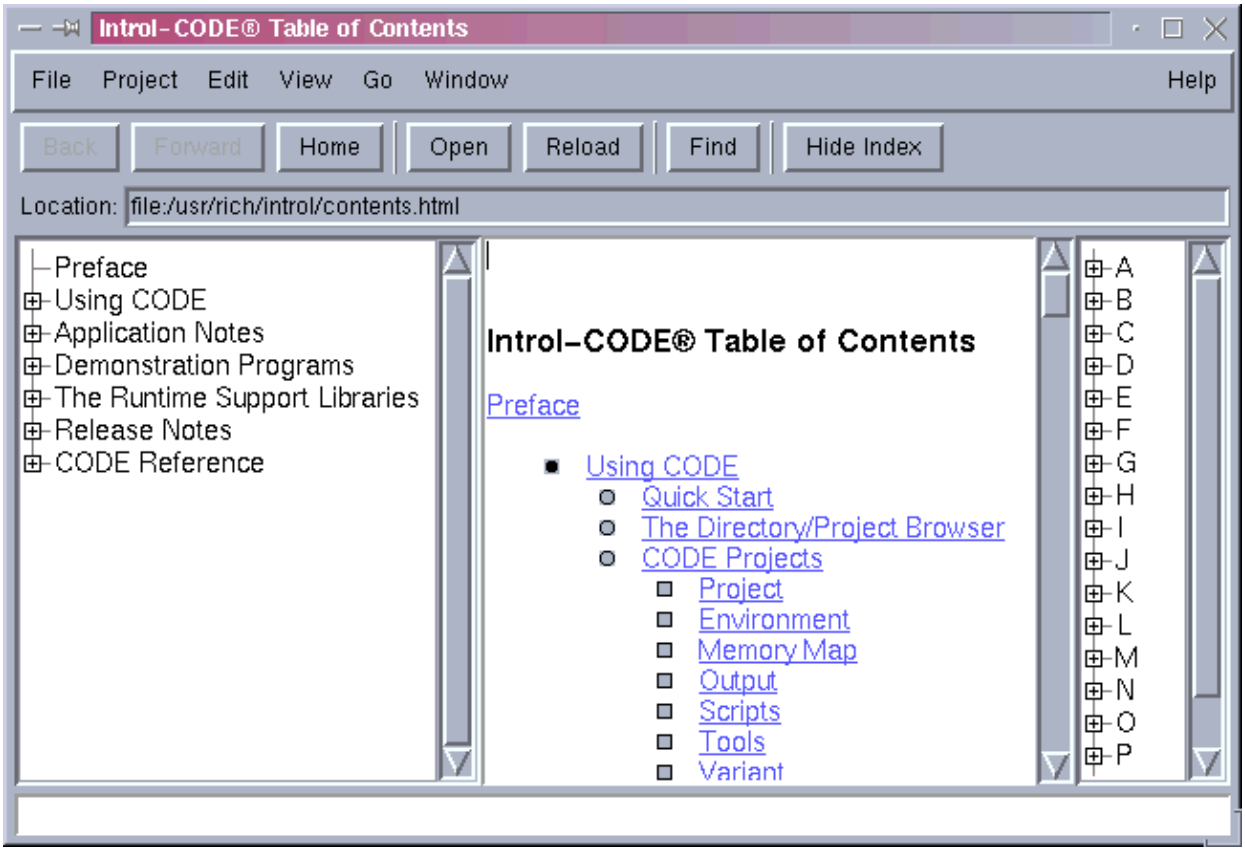
P&E ICD or public domain BDM

P&E CABLE12 BDM

Motorola Serial Debug Interface

The Manual Browser

The manual window can be used to display the CODE on−line help information and other HTML (HyperText Markup Language) documents. This is the standard format for World Wide Web documents which means you can use any WWW browser to view the CODE manual pages. The manual window adds a table of contents and index that make finding your way around the manual easier:



The manual window adds **View**, and **Go** menus to the standard CODE menus.

The **View** menu allows you to reload the current document or view the document's HTML source.

The **Go** menu lets you navigate through the Manual's history list. The current document is highlighted in the **Go** menu. The history list keeps track of the manual pages that you have visited. You can go to a previously viewed page by selecting it from the **Go** menu or you can use the **Back** and **Forward** buttons to see the previous and next pages in the history list.

If you are at the beginning of the history list the **Back** button will return you to the last CODE window you were in before entering the Manual window.

The **Hide Index** button will remove the index from the manual window to increase the viewing area.

There are several buttons below the menu bar that are shortcuts to some of the most used menu functions.

The location field shows the name of the document being displayed. You can use this field to enter the name of a document that you want to display.

The main manual window is divided into three panes by default. The leftmost pane is the table of contents. Clicking on any entry will bring up that manual page. The center pane displays the current manual page, with related links underlined and displayed in blue. This rightmost pane is the alphabetically sorted index. It can be used like the table of contents, but is made up of topics rather than page titles. If you have limited space on your display screen is is possible in the manual's preferences to set an option that causes the index and table of contents to share a pane, with a button to toggle which is being displayed.

Preferences

CODE has many *preferences*, or options that you can tailor to suit your needs. The **Preferences** dialog box is used to edit the CODE options. You access the **Preferences** dialog by choosing **Preferences...** from the **File** menu in any CODE main window.

General
: General color and font options.

Manual
: Options for the manual browser.

Directory
: Options for the directory/project browser.

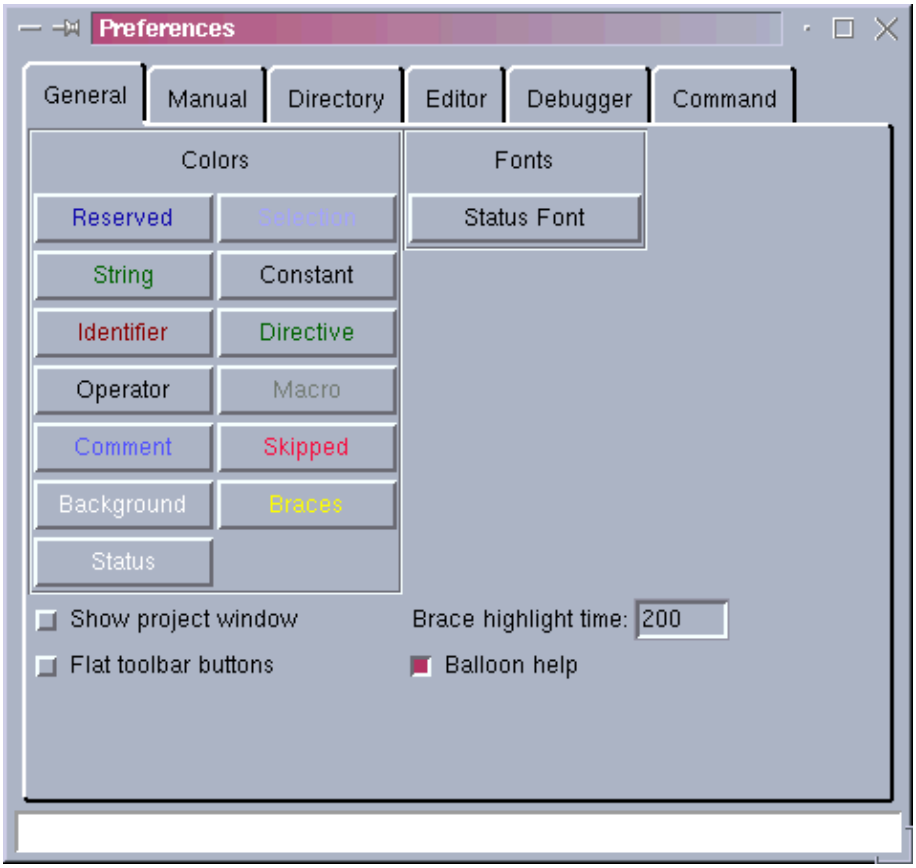
Editor
: Options for the editor.

Debugger
: Options for the debugger.

Command
: Options for the command window.

General

The **General** tab of the preferences dialog box sets options that apply to most or all CODE windows.



The **Colors** define how various parts of windows, and text within windows are displayed.

The **Status Font** button sets the font that is used in the CODE status display, which is at the bottom of most CODE windows.

The **Show project window** checkbutton, when enabled, will cause the project editor to be opened when CODE is started.

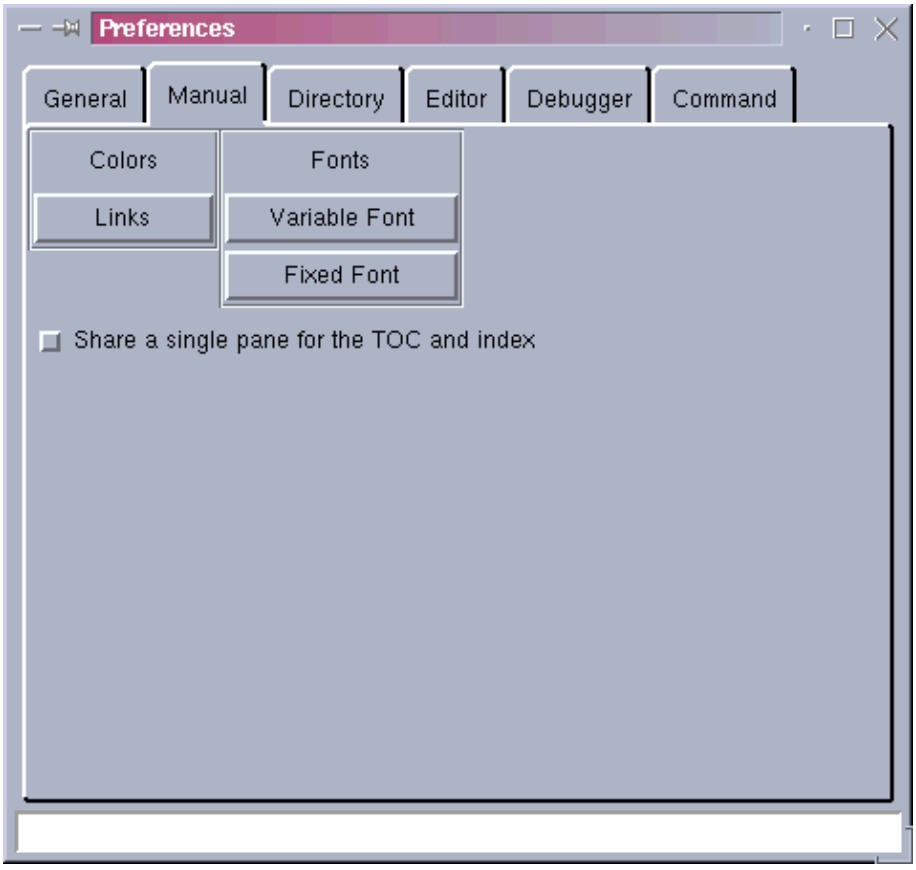
The **Flat toolbar buttons** checkbutton, when enabled, will cause CODE to display buttons in main windows as flat until the mouse cursor moves over the button.

The **Balloon help** button enables the display of help messages for a button or control in a little window that appears about a second after the mouse is placed over the button or control. If **Balloon help** is disabled, the help message will appear in the window's status bar.

The **Brace highlight time** entry sets the number of milliseconds that the text between braces, brackets and parentheses will be highlighted when you do brace matching or are using the editor or command windows.

Manual

The **Manual** tab of the preferences dialog box sets preferences that are used in the manual window.



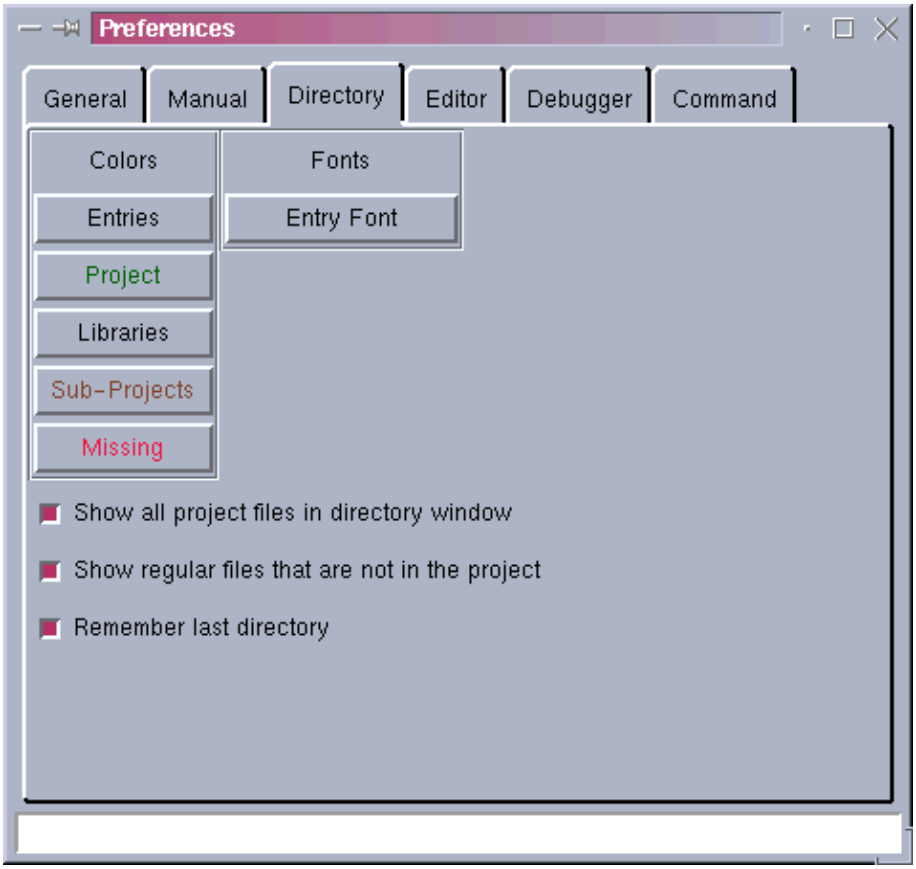
The **Links** button sets the color used for the manual's hypertext links.

The **Variable Font** button sets the proportional font used by the manual and the **Fixed Font** button sets the font that the manual should display pre formatted text with.

The **Share a single pane for the TOC and index** checkbutton, when set, will cause the manual's table of contents and index to share a single pane in the manual window, with a button to toggle between them, to save screen space.

Directory

The **Directory** tab of the preferences dialog box is used to set preferences in the directory window.



The **Colors** buttons control how various project and non project files are displayed in the directory window, while the **Entry Font** button controls the font in which directory entries are displayed.

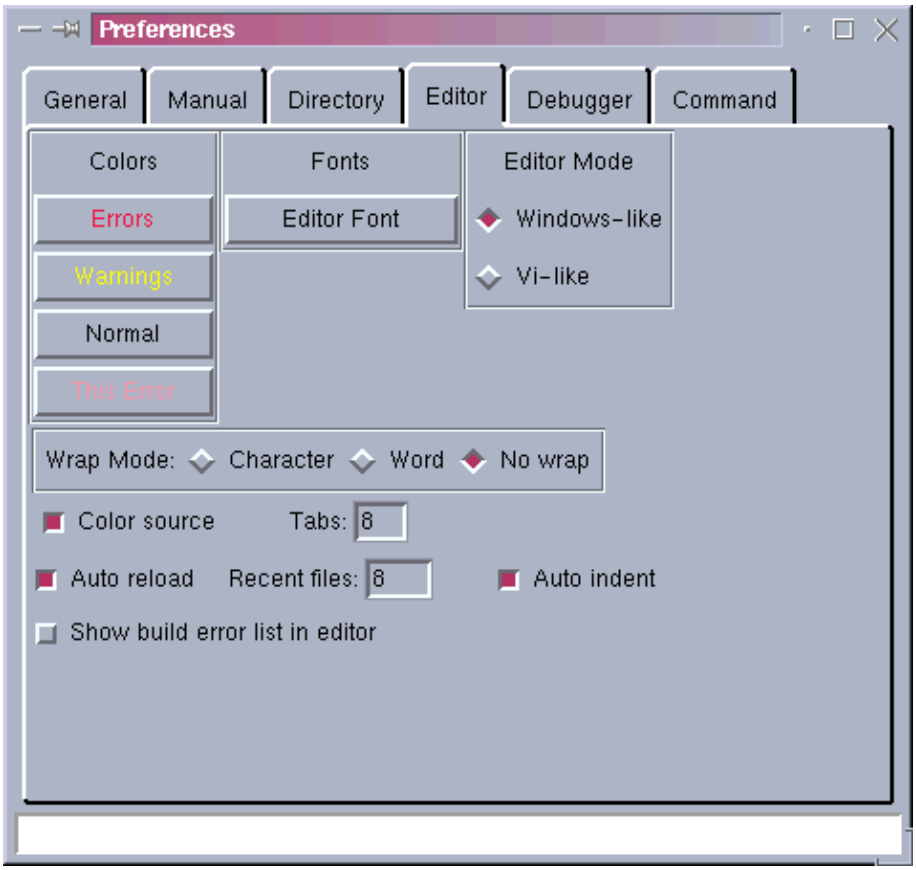
The **Show all project files in directory window** checkbutton, if enabled, will cause all project files, not just those in the current directory, to be displayed.

The **Show regular files that are not in the project** checkbutton, if enabled, will cause all files in the directory, not just project files, to be displayed.

The **Remember last directory** checkbutton, if enabled, will result in the last directory you were in being restored when you reenter CODE.

Editor

The **Editor** tab of the preferences dialog box sets up various options used in the editor window.



The **Colors** buttons control how normal text (when not colored by a syntax coloring algorithm) and build errors and warnings are displayed.

The **Editor Font** button controls the font used by the editor.

The **Wrap Mode** selector determines whether wrapping is done by the editor and how it should be done.

The **Color source** checkbutton, when enabled, turns on syntax coloring in the editor.

The **Tabs** entry is the number of characters between tab stops.

If the **Auto reload** checkbutton is enabled, the editor will automatically reload any files that are being edited if they have changed on the disk.

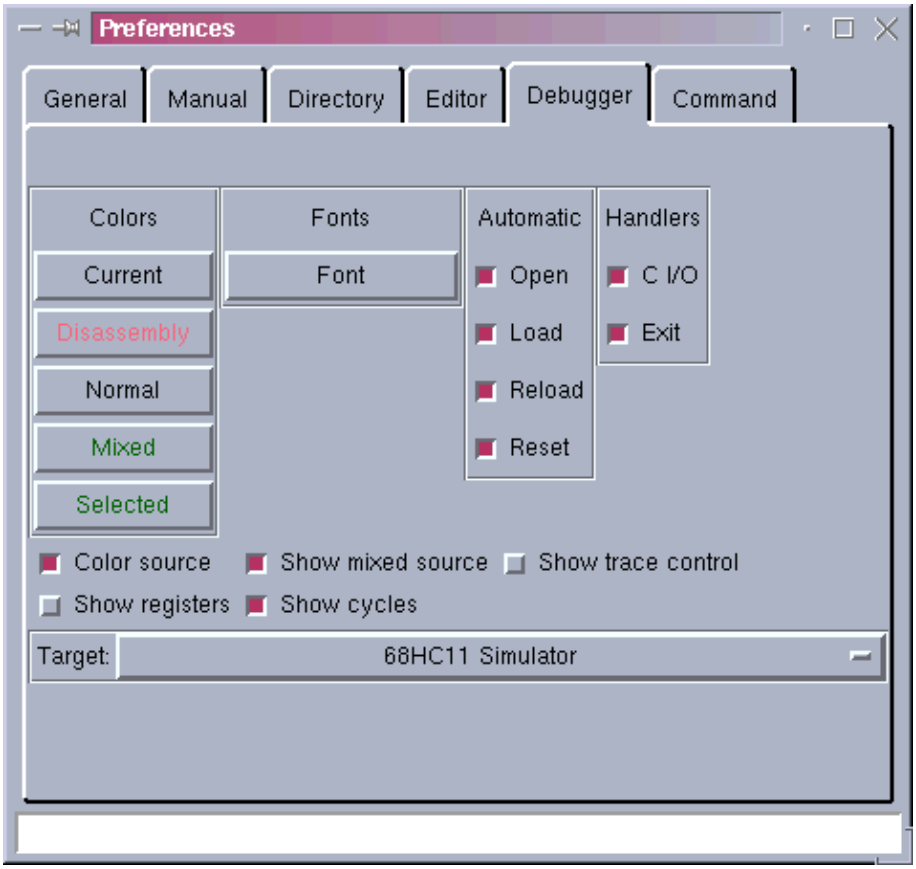
The **Recent files** entry controls how many recently edited files the editor will remember and place under the **Go** menu.

The **Auto indent** checkbutton, if enabled, will cause the editor to indent a new line the same as the line immediately preceding it.

The **Show build error list in editor** checkbutton will cause any errors that occur during a C compile or an assembly to be shown directly in an editor window, as opposed to bringing up the offending source file. This is handy when you are using an external editor and just want to see the list of build errors.

Debugger

The **Debugger** tab of the preferences dialog box controls options available in the debugger window.



The **Colors** buttons control how various text is displayed in the debugger views. Text in debugger views are displayed in the font specified with the **Font** button.

The **Automatic** checkbuttons control operations that debugger can do automatically when entered.

The **Handlers** checkbuttons control breakpoints that can be set automatically by the debugger. If **C I/O** is checked, then program input/output through *stdin* and *stdout* is trapped and sent to the Stdio view. If **Exit** is checked, then the debugger sets a breakpoint at the program termination point.

The **Color source** checkbutton enables debugger syntax coloring.

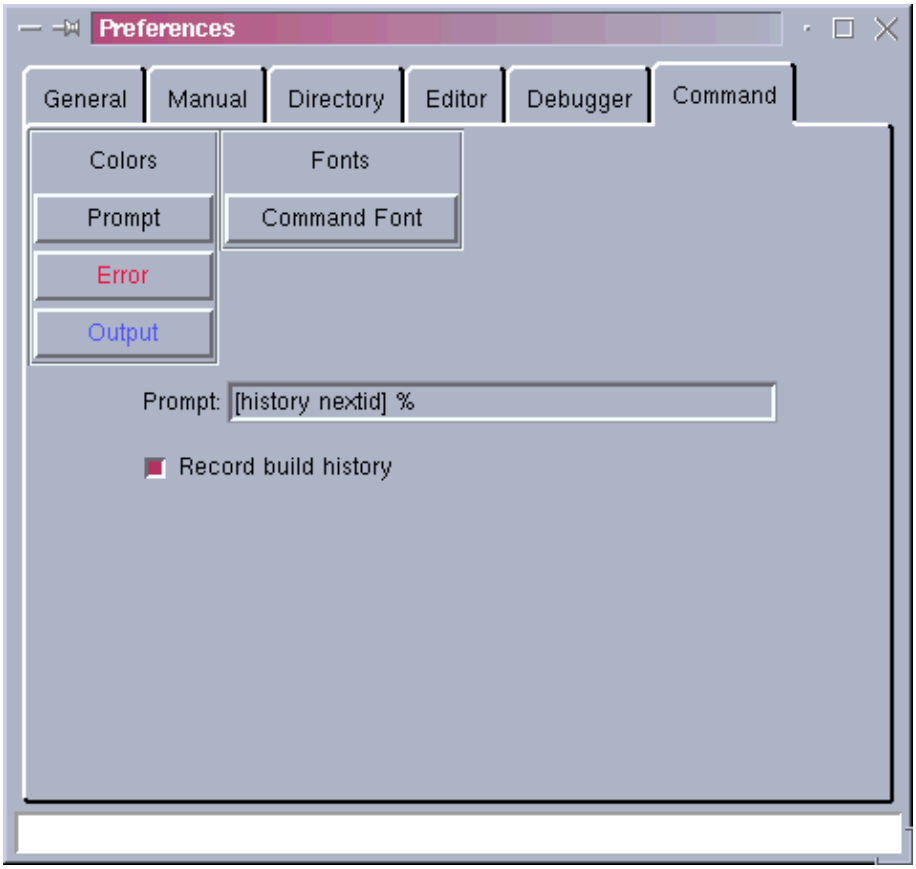
The **Show mixed source** checkbutton, when enabled, will cause the debugger to display program source in the Mixed view.
Otherwise, just the filenames and line numbers of the program source are displayed.

The **Show trace control** and **Show registers** checkbuttons, if checked, will cause the debugger to automatically open the trace control and register windows, respectively, when the debugger is started.

The **Target** menu allows you to select the default debugger target.

Command

The **Command** tab of the preferences dialog box controls options for the command window.



The **Colors** buttons control how the command prompt, program error output, and program standard output is displayed.

The **Command Font** button controls the font used in the command window.

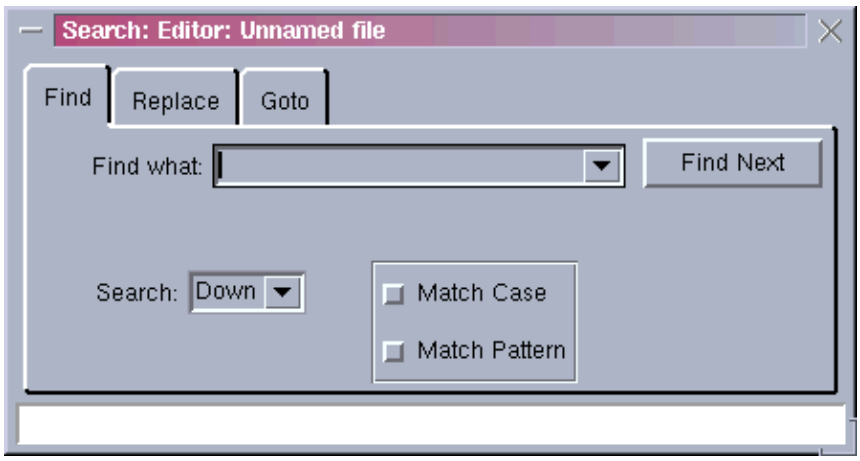
The Prompt entry specifies the command prompt. This entry is evaluated as a Tcl script, which is why **[history nextid]** becomes the command number.

If the **Record build history** checkbutton is set, CODE will remember all the commands that are executed when building projects as well as all commands entered in the command window manually.

Searching and Bookmarking Files

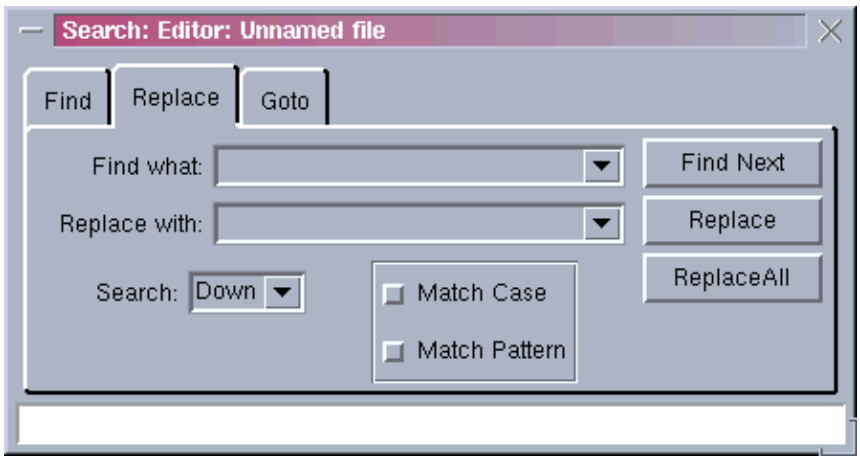
The Search window allows you to search text, replace text strings or *patterns*, and go to line numbers and text marks. There is only one **Search** window in CODE. It is associated with the last **Manual**, **Editor**, or **Debugger** window that attached to it either through the **Find** tool button, the **Find**, **Replace**, or **Goto** entries of the **Edit** menu, or by pressing **Control−F**, **Control−R**, or **Control−G** in the window. When a CODE window is attached to the **Search** window, the contents of the attached window will be displayed in the **Search** window's title bar.

Find



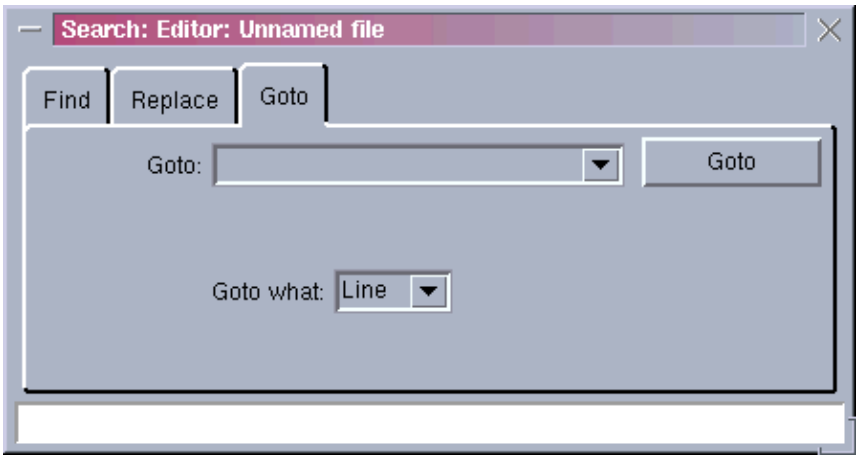
The Find tab is used to find strings or patterns in text. If you are searching for strings, you have the option of requiring the case of the strings to be identical. Patterns are specified as regular expressions.

Replace



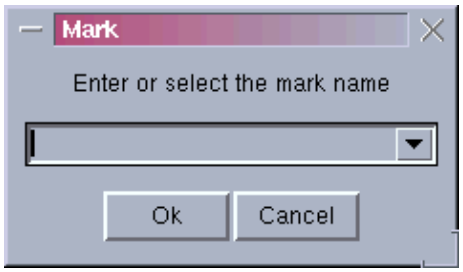
The **Replace** tab allows you to replace strings or patterns in text.

Goto



The **Goto** tab is used to go to a specific line number in a file or to a mark that has been previously set in text.

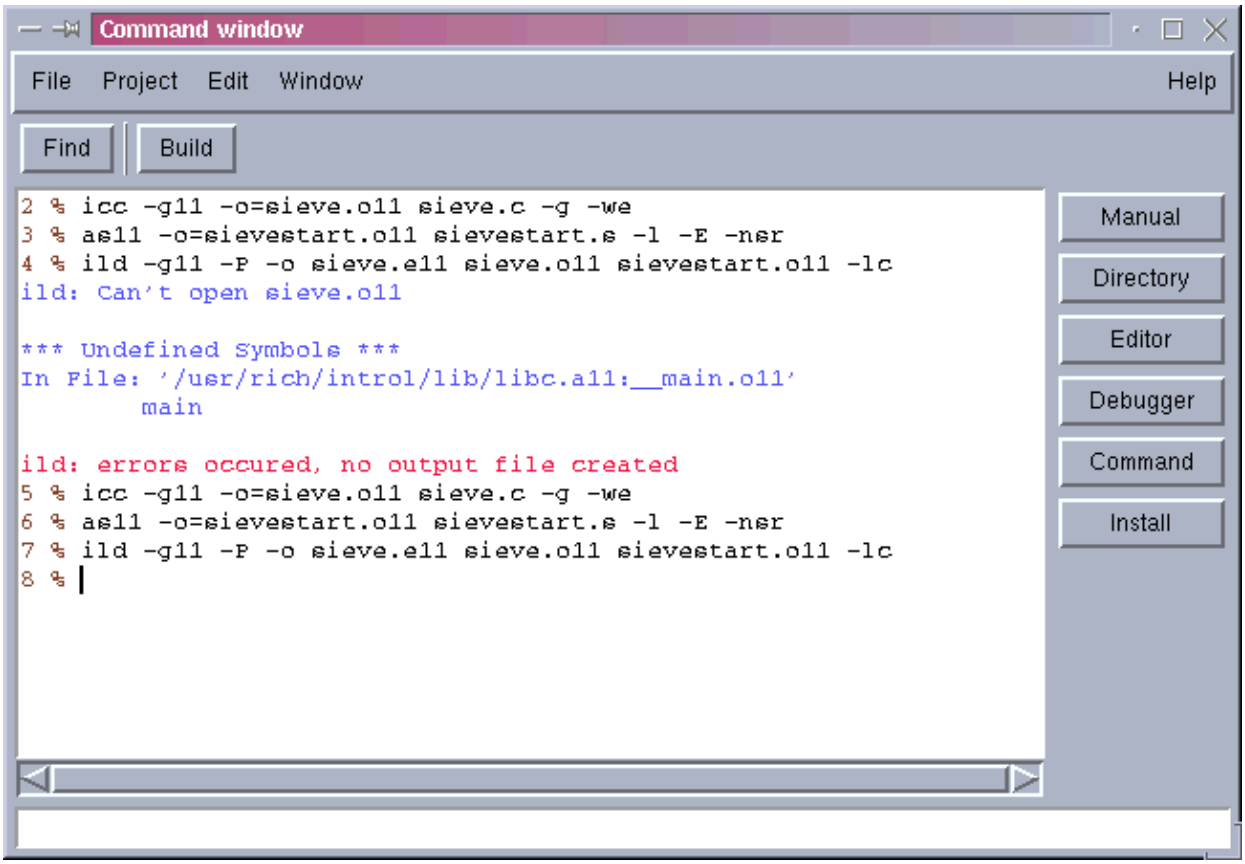
Marking Text



The **Mark** window allows you to add a mark to text. A text mark can have an arbitrary name: only the words "**anchor**", "**current**", and "**text**" are reserved. CODE will remember any marks you set on a file in the **Editor**, **Manual**, and **Debugger** windows during a session. You can then use your marks in any of those windows that has that file open. You can go to a particular text mark by using the **Goto** search window.

The Command Window

The **Command** window is a command line interface to CODE's underlying Tcl/Tk scripting language. You can also use the command window to execute command line programs that exist on your system, for example the various compilers and assemblers and other command line programs that come with CODE.



When you use CODE to build a project or any of its components the Command window is updated with the command line that CODE executed and the output that the command may have produced. This may be helpful to track down difficult build errors.

Command maintains a history of commands that have been executed. You can use the arrow keys to recall commands from the history list and re−execute them.

Version Numbers

Version numbers have the form X.XX.X (three period−separated numbers) for release software and X.XX.X.X (four period−separated numbers) for pre−release software. The first two numbers are the major and minor version numbers, e.g. 4.00. For pre−release software the third digit indicates the release status: 0 is ALPHA (not yet feature complete), 1 is BETA (feature complete but may contain bugs). The fourth number is a release number. For example, version 4.00.1.1 is the first BETA release of the 4.00 version of CODE. For release software, the third number, if it exists, represents the point release number and will be 2 or greater. For example, 4.00.3 is the first point release after the initial release of CODE 4.00.2.

CODE can be used from its Integrated Development Environment (IDE) or programs can be developed using the traditional command line approach using the commands supplied with CODE. You can find the version number of CODE from the IDE by selecting **About** from the **Help** menu. All command line programs also display version numbers when invoked without any arguments. For example, to determine the version number of the 68HC11 assembler, type **as11** at a command prompt (either the CODE command prompt or the system command prompt) and a message like the this will be displayed:

```
as11: 4.00.2 Copyright 1996−2000 Introl Corp.
```

We release all components of CODE at the same time so all components will have the same version number. Even if the entire CODE release is an ALPHA or BETA release, most of the components will be of final release quality. We are continually developing and improving CODE and generally an ALPHA or BETA designation applies only to new components that have been added. The CODE release notes will have up to date information about the components you are using.

Have your CODE version number handy when contacting Introl technical support: it helps us determine what software you are using so we can help you more efficiently.

Installation

You can install CODE from the CD−ROM by running SETUP in the main directory of the CD−ROM.

If you have downloaded CODE you can install it by placing the downloaded CODE self extracting archive file in your chosen installation directory and running it.

SETUP helps you install CODE on your computer. You can choose which components of CODE should be installed and choose the installation directory from **SETUP**.

CODE does not have to be installed to be used. You can run it directly off the CD−ROM. If you choose to install CODE it may take up from 14 MB to 60 MB of disk space depending upon which host system binaries you choose to install.

If you choose not to install the online manual from the CD−ROM, you save about 4 MB of disk space. The manual will be accessible whenever the CODE CD−ROM is in the drive.

If you have downloaded and run the self extracting CODE archive code is already installed.

If you install CODE from the CD−ROM, when the installation is finished just exit CODE and start the CODE application in the directory you specified.

To uninstall CODE, just delete the directory you installed it into.

The Quick Start section is the best place for new users to become familiar with CODE.

Application Notes

This part of the manual contains miscellaneous topics that may be interesting to CODE programmers.

The Runtime Environment
: The runtime environment of an embedded program and how to configure it using CODE.

Custom stdio Drivers
: Making custom drivers so the C standard library can access your devices.

Direct Port Input/Output
: Accessing memory mapped I/O and peripherials using C.

C Language Extensions
: The extensions Introl−C has that are not in the ANSI standard.

Symbols in C and Assembly
: Naming conventions and symbol sharing between C and assembly language files.

C Calling Conventions
: The C compiler function calling conventions and register usage.

C Memory Models
: The C memory models for those processors that have more than one.

C Position Independence
: The 68XXX and 6809 C compilers support position independent code and data.

C Hardware Floating Point
: The 68XXX C compiler can generate code to take advantage of floating point hardware.

Program Sections
: Program sections and how they are used by the CODE to control where program code and data go.

Building Standard Libraries
: How to modify and build the runtime support libraries.

The CREX Real−time Executive
: An overview of the CREX real−time executive supplied with CODE.

Assembler Only Projects
: Making a CODE project that contains only assembly language files.

Device Definition Files
: Modifying and creating Device Definition Files (.ddf).

Moving Functions to RAM
: Moving functions into RAM from Flash/ROM and executing them.

The Runtime Environment

The process of building an application for an embedded system is quite a bit different from building an application that runs on a general purpose computer. Even the simplest program requires a much more detailed knowledge of the execution environment than is needed in a general purpose computing environment.

For example, suppose we were to build the program

```
#include <stdio.h>

int main(int argc, char **argv)
        {
        printf("Hello world\n");
        }
```

in a general purpose environment. A typical command line might look like:

```
cc −o hello hello.c
```

The program could then be executed, producing the output "Hello world".

Embedded systems are not quite so simple. We need to ask several questions:

- What is the target processor?
- What on chip resources does the processor have?
- What kind of memory resources exist on the target and where are they?
- Is there any software running on the target?
- How do I set up the processor's interrupt and exception vectors?
- How do I get my program into the target processor's memory space?
- Where is "Hello world" supposed to go?

In the general purpose environment these questions can be answered by default because the operating system defines how a program that runs in the environment should be built. CODE needs answers to these questions to be able to produce a program that runs in the target environment. There are various dialog boxes in CODE that let you answer these questions in a simple, straightforward way. The information used to answer these questions are saved as part of a CODE project file. Most of the dialog boxes mentioned in this section are accessible through the CODE project editor.

What is the target processor?

The first thing you should specify in a project is the target processor you are working with. You can do this in the **Project** tab of the project editor. The **Processor Family** menu lets you select your target processor. Every microcontroller used in a embedded system has its own unique language, or instruction set, that it understands. This instruction set can be represented in several forms. At the lowest level the instruction set is represented by zeros and ones in the computer's memory. This is the processor's *machine code*. *Assembly language* is the next step up to represent the processor's instruction set. This is a human readable form of the instruction set that is very closely related to the machine code. In general, there is one line of assembly language for each machine instruction.

CODE can translate assembly source files for many Motorola microcontrollers. It does this translation using an *assembler* specific to the target processor. The assemblers, and the processors they support are listed in the following table.

Assembler	Supported Processor(s) (default is **bold**)
as05	**68HC05**
as08	**68HC08**
as09	**6809**
as11	**68HC11**, 6801, 6301
as12	**68HC12**
as16	**68HC16**
as68	**683XX**, 68000, 68010, 68020, 68030, 68EC030, 68040, 68EC040

Normally when you have an assembly language file in your project, CODE will know which assembler to use to translate the file based on the project's target processor. You can refer to the assembler documentation for more information on the assemblers, including assembler command line options, directives, and macros. CODE understands that files with a **.s*XX*** extension are assembly source files, where *XX* is the two digit processor number in the assembler's name. You do not have to name your assembly language files this way but we find it a useful to keep assembly language source files for several processors separate. CODE also understands the extensions **.s** and **.asm** for assembly language source files.

CODE also contains C compilers for several target processors. C is also a representation of a processor's instruction set, but one much more removed from the actual processor than assembly language. Introl's C compilers actually perform the translation of C to machine code in two steps: first C programs are translated to assembly language, then the assembly file is assembled to produce an object file. The C compilers, and the processors they support are listed in the following table.

C compiler	Supported Processor(s) (default is **bold**)
icc	The general compiler
cc09	**6809**
cc11	**68HC11**, 6801, 6301
cc12	**68HC12**
cc16	**68HC16**
cc68	**683XX**, 68000, 68010, 68020, 68030, 68EC030, 68040, 68EC040

If you have a C source file as part of your project, the appropriate C compiler will be used to translate it for the project's target processor. C files in a CODE project have a **.c** extension. The **cc*XX*** compilers are identical to the **icc** compiler invoked with a **−g*XX*** option. Refer to the compiler documentation for more information on the C compilers and their command line options. A compiler should also be invoked with a **−g*XX*** option if you want to generate code for a processor that is not the default processor listed in the table above.

What on chip resources does the processor have?

Most processor families have several *variants*. Each variant has a slightly different complement of on chip peripherals. The **Processor Variant** menu in the **Project** tab of the project editor allows you to choose the specific variant you are using. CODE will adjust several of its dialog boxes specifically for your selected variant, such as the **Vectors** and **Registers** tabs of the **Configure Environment** window.

If the variant you are using is not mentioned, don't panic. It is easy to set up support for another variant.

What kind of memory resources exist on the target and where are they?

Memory resources, for example RAM and ROM, that exist in the target environment are highly dependent on the target processor and the design of the target system hardware. At some point during development you have to decide where in the target system's memory your program code and data will reside.

Introl assemblers (and therefore, C compilers) produce object files in *Introl Common Object File Format*, or ICOFF. ICOFF files created by the assemblers can be either *absolute* or *relocatable*. Absolute files are created when the source file fully defines (with **org** directives) the addresses that all the program code and data will reside in in the target memory space and when all references and definitions of code and data are contained in a single source file (there are no **import** directives). A relocatable file either has external references or places its code and data in named sections without explicit origins. The compilers always produce relocatable object files.

To build a set of source files in a project into a program, the source files are first translated appropriately and then *linked* together to form the program.

A single assembly source file that produces an absolute file does not have to be linked.

Relocatable files are combined together and given absolute address using the object file linker, **ild**. When using the linker, a *linker command file* (which normally has the extension **.ld**), is used to tell the linker how to place the program code and data in memory. In particular, the linker's **section** command is used to give sections origins and content.

CODE usually makes the linker command file for you using information you specify in the **Memory Map** window. You can also configure a linker command file by hand, usually by setting up a memory map in the Memory Map window and then selecting **View LD File** from the **Project** menu. You can then modify the generated linker command file, save it to a file, and tell CODE to use it to link your project in the **Tools** tab of the project editor.

Is there any software running on the target?

This is an important question to help us determine how to configure the environment of the target system. The environment configuration tells CODE how to get a program ready for execution. If there is no software already running on the target system and your program will be the only program running then the configuration will define the entire state of the processor from reset. The configuration will define the reset vector, the initial processor stack pointer, and do any other initialization that the processor might need. The **Configure Environment** window , accessible from the project editor, allows you to set up your program for its target environment.

If there is other software running on the target, for example a monitor program or real−time executive, then

the configuration should be modified to run *under* that other software. Typically, these environments will already have set up the reset and interrupt vectors, the stack pointer, and any other chip initializations. This other software may also have certain requirements about where the program's code and data should be located. You can control all of this through the Configure Environment window.

CODE uses the information you specify in the Configure Environment window to make *startup code* for your program. When you build a project, CODE will create a file containing the startup code called ***name*start.s**, where *name* is the name of your project. Whenever you change information in the Configure Environment window and re−build you project, the startup code file will be regenerated. You can specify your own startup code file in the **Tools** tab of the project editor.

How do I set up the processor's interrupt and exception vectors?

To have a C or assembly language function handle a specific interrupt or exception go to the **Vectors** tab of the **Configure Environment** window and change the entry for that vector to the name of your handling function. The function called must do a proper return from interrupt, in C this is accomplished using the **__interrupt** keyword as described in the Language Extensions section. The vector table is generated in the startup code file described above.

How do I get my program into the target processor's memory space?

The CODE debugger will load your program into the target's address space. If you want to use a different debugger or burn an EPROM, for example, the **Output** tab of the project editor can be used to specify one or more optional output files, such as Motorola S records and IEEE−695 format.

Where is "Hello world" supposed to go?

Standard input and output are a bit of a problem on an embedded system: there isn't any. You may want to use functions like *printf* during debugging or stdio might be an intrinsic part of your embedded program. In either case you have to decide where standard output goes and standard input comes from. You may also want to specify additional input/output devices and use the standard C functions to access them. The **IO** tab of the Configure Environment window lets you define the devices your program will use. You can also use this tab to define custom devices that your hardware supports.

Custom stdio Drivers

It is very easy to add low−level drivers to the CODE runtime library. A low−level driver is used to interface the C stdio functions to a device.

The runtime libraries access low−level drivers through a dispatch table. There are two parts in a table entry for each device. The first part is contained in read−only memory and looks like this (from stdio.h):

```
/*
 *      FILE − type definition for the FILE structure
 *      This part of the FILE information is contained in ROM.
 */
typedef struct __FILE
        {
        struct __FILEv __FAR *f_v;                      // volatile (R/W) file informati
        char __FAR *f_name;                                     // device name
        __CDECL int (*f_open)(struct __FILEv __FAR *);          // device initialization
                                                                // function
        __CDECL int (*f_close)(struct __FILEv __FAR *);         // disables the device
        __CDECL int (*f_read)(struct __FILEv __FAR *);          // read from the device
        __CDECL int (*f_write)(int, struct __FILEv __FAR *);    // write to the device
        __CDECL int (*f_ioctl)(int, void __FAR *, struct __FILEv __FAR *); // device con
        } FILE;
```

The **f_v** member in the table points to read/write information for the device. The f_name member points to the name of the device which is used by **fopen()**. The remaining five function pointers are used by the runtime library to communicate with the device. **f_open** is called when the device is first opened, **f_close** is called when the device is closed, **f_read** is called to get a character from the device, **f_write** is called to send a character to the device, and **f_ioctl** is called to control certain behavior in the low−level driver. These functions will be described in detail below. All of these functions take as a parameter a pointer to the read/write information for the device, which looks like this:

```
/*
 *      __FILEv − changeable parts of a __FILE
 *      This part of the FILE information is contained in RAM.
 */
struct __FILEv
        {
        void __FAR *f_buffer;                   /* input buffer */
        char __FAR *f_buffptr;                  /* place in buffer from which
                                                 * the next character is read
                                                 */
        int f_cnt;                              /* number of character in the
                                                 * buffer */
        struct __io_params f_io_params;         /* line cooking parameters */
        unsigned char f_mflags;                 /* file mode flags */
        };
```

This information is normally used by the higher level stdio library functions but may also be used by a low−level driver on occasion.

The read−only entries in the device table are placed in the global array **__devtab**. The end of the table is pointed to by the global pointer **__devtabend**. These global variables are declared like this:

```
extern FILE __devtab[];
extern FILE *__devtabend;
```

fopen() and similar functions use **__devtab** and **__devtabend** to find the device table entry for pre−defined and user defined low−level drivers.

You can define **__devtab** yourself in your C or assembly language source, but normally CODE generates **__devtab** automatically in the program startup code. This is controlled in the **IO** tab tab of the Configure Environment dialog box. The **IO** tab lets you add entries to the device table for additional low−level drivers that are not part of the standard CODE distribution. Low−level drivers can be written in C or assembly language.

The devices marked as **stdin**, **stdout**, and **stderr** in the **IO** tab are automatically opened in the startup code.

The five functions needed by a low−level device driver can have any names that do not conflict with pre−defined library names. These are the names you put the the **IO** tab. In the following descriptions, we use names such as **open**, etc. These should be changed and made unique for each low−level driver. There are examples of low−level drivers for each processor in the C library source directories. These examples are all called **io.c** and are located in the **lib*XX*** directory for each processor. You may also want to look at the Direct Port Input/Output application note for information about accessing peripheral registers from C.

Open Device

The open device function is declared

```
int open(struct __FILEv __FAR *fp);
```

This function should do whatever is necessary to initialize the device when it is first opened. This might include resetting the device, setting an initial baud rate, etc. The **open** function should return −1 if an error occurs.

Close Device

```
int close(struct __FILEv __FAR *fp);
```

This function is called when a device is closed. The **close** function should return −1 if an error occurs.

Read Device

```
int read(struct __FILEv __FAR *fp);
```

This function is called to read a character from a device. The read function should wait for a character to be available and return it, or return −1 if an end of input condition is encountered.

Write Device

```
int write(int ch, struct __FILEv __FAR *fp);
```

This function is called to write **ch** to the device. It should return −1 if the write fails.

Control Device

```
int ioctl(int cmd, void __FAR *vp, struct __FILEv __FAR *);
```

This function is used to control the behavior of a device. Valid values of **cmd** are (from **stdio.h**):

```
#define __HASINPUT      2       // does input stream have characters ready?
#define __RECVINTERRUPT 3       // control the receive interrupt
#define __ACKRI         4       // acknowledge receive interrupt
#define __XMITINTERRUPT 5       // control the transmit interupt
#define __ACKTI         6       // acknowledge transmit interrupt
```

For **__HASINPUT**, the value of **vp** will be 0 and **ioctl** should return the number of characters available from the device, or −1 if an error occured.

For **__RECVINTERRUPT**, the receive interrupt of the device, if any should be disabled if **vp** is equal to 0 or enabled if **vp** is non−zero. ioctl should return 0 if the receive interrupt was disabled before the call and 1 if the receive interrupt was enabled befor the call.

For **__ACKRI**, the value of **vp** will be 0. **ioctl** should do whatever is necessary to acknowledge a receive interrupt.

__XMITINTERRUPT and **__ACKTI** are similar to **__RECVINTERRUPT** and **__ACKRI** except that they are for the transmitter interrupt.

Direct Port Input/Output

CODE supports several processor families and many individual processors in those families. Most of these processors have on chip peripheral registers that can be used to control I/O ports, timers, analog to digital convertors, etc. In CODE, each member of a processor family is called a *variant*. You choose the variant of a processor family you want to use using the **Project** tab of the Project editor.

Declaring Registers in C

You can access device registers in C in several different ways. In CODE, the most common way is to include a pre−defined header file for the particular processor you are using. The **include** directory has many header files, with names like "**hc11e9.h**", that can be included in your C program to define all the on−chip registers for a particular processor. Then, accessing registers is easy:

```
#include <hc11e9.h>

int main(void)
    {
    PORTA = 0x00; // set all bits in PORTA to 0
    }
```

If you don't want to use the standard header files, or you want to declare registers that are not in the standard headers, you can declare ports directly. Remember to use the **volatile** keyword when defining a port. This will insure that the compiler does not optimize out any port references. Let's say that you want to use a byte sized register that is at address 0x2000 in your target system. There are a couple of ways to accomplish this. The first way is by declaring a C variable in the usual way:

```
extern volatile unsigned char MYPORT;
```

Notice that the register is declared **extern**. This tells the compiler and linker not to allocate space for the variable. If we try to build a program with a declaration like this, the linker will complain about **MYPORT** being undefined. One way to define **MYPORT** is to add a definition to the linker command file used to build the project. If you are using a CODE generated linker command file, you can add a definition for **MYPORT** by pressing the **Variables** button in the Memory map window and adding a line like:

```
let MYPORT = 0x2000;    // set the address of MYPORT
```

If you are using an external linker command file, just put the line above in your file.

Another way to define MYPORT is to place a definition in an assembly language source file. Both of the following will work:

```
        export    MYPORT
MYPORT  equ       $2000
```

or

```
        export    MYPORT
        org       $2000
MYPORT: ds.b      1
```

Another way to access registers in C is to use casts. You cast a constant (the register address) as a pointer to a

type of the register's size and go indirect through the pointer:

```
#define MYPORT (*(volatile unsigned char *)0x2000)

int main(void)
    {
    MYPORT = 0x00;
    }
```

MYPORT can be used just like a variable when defined like this.

Manipulating Bits in C

All the processors that CODE supports have memory mapped registers. This means that it is very easy to access and manipulate these registers in C. Peripheral registers can be treated just like external variables and all operations that can be performed on variables can be used on registers. Typical device registers are one or more bytes wide and contain several bits that may need to be tested or manipulated to control the device's function. C has several operators that can be used to manipulate device registers with ease. The bitwise or ('|'), bitwise and ('&'), and bitwise exclusive or ('^'), along with their assignment update versions ('|=', '&=', and '^=') are very handy for dealing with device registers. The one's complement operator ('~') will also come in handy.

You often have to set or clear a bit in a register. The easiest way to set a bit is to use '|=' (using one of the definitions of MYPORT above):

```
int main(void)
    {
    MYPORT |= 0x02;    // set bit 1 of MYPORT
    }
```

Similarly, to clear a bit, '&=' works well:

```
int main(void)
    {
    MYPORT &= ~0x02;    // clear bit 1 of MYPORT
    }
```

Notice that the value being anded with MYPORT is the one's compliment of the bit(s) you want to clear.

A program often has to wait for one or more bits in a register to change state. The following example will wait in a loop until bit 7 is 1:

```
int main(void)
    {
    while((MYPORT & 0x80) == 0)
         ;    // wait for bit 7 of MYPORT
    }
```

C Language Extensions

The Introl−C compiler is ANSI compatible. It also supports the following extensions.

Bit fields

The ANSI C Standard states that bit fields, no matter what type, are filled in units the size of unsigned integers. Introl−C allows any integral scalar type to be used as a bit field unit. For example, if the bits in a field were defined as type **char**, they would be stored in a **char**. You may do this if you want to assign an 8 bit field to some hardware register.

The default fill order for bit fields is from least significant to most significant bit. For example:

```
struct tag
        {
        char a : 1;
        char b : 1;
        char c : 1;
        } abc;
```

the bit field would be stored as follows:

```
Bit no: 7 6 5 4 3 2 1 0
Field :           c b a
```

The Introl−C compiler has two command line options that control the fill direction of bit fields: **−gbl** (default) and **−gbh**. If the **−gbh** option is used, bit fields will be filled from high to low order:

```
Bit no: 7 6 5 4 3 2 1 0
Field : a b c
```

The actual size of the structure used to contain the bit fields will be the size of the type of the elements. In the above example, the size of the structure is equivalent to the size of its defining type of **char**. If the bit field had been declared as follows:

```
struct tag
        {
        int a : 1;
        int b : 1;
        int c : 1;
        } abc;
```

the size of the encompassing structure will be equal to the size of an **int**.

For declarations of mixed type, the code generator will start packing a structure the size of the first declaration until it sees a declaration of another type. Then, it will begin packing a new structure until it again sees a declaration of another type. If the bit field is declared as:

```
struct tag
        {
        int a : 1;
        int b : 1;
        char c : 1;
        } abc;
```

the compiler will pack the first two fields into an **int** and the last field into a **char**. If the bit field is declared as:

```
struct tag
        {
        char a : 1;
        int b : 1;
        char c : 1;
        } abc;
```

the compiler will put the first field into a **char**, the second field into an **int**, and the third field into a new **char**. In other words, the compiler will not fill a member of a certain type unless the fields were defined continuously, but rather will begin to fill a new member with each new type of declaration.

User defined storage allocation

Introl−C recognizes the extended type modifiers **_ _mod1_ _** , **_ _mod2_ _**, and **_ _base** which may be used in the same context as **const** and **volatile**. When used in non−function declarations, the effect of these modifiers is to change the name of the section to which storage is allocated as follows:

modifier used	section name
_ _mod1_ _	.mod1
_ _mod2_ _	.mod2
_ _base	.base

If an extended type modifier is used alone in a declaration whose storage class would be implicit **auto**, they behave as explicit **extern** declarations. If **_ _ base** is used to declare a variable on a processor that supports base page or direct addressing, such as the 68HC11 or 68HC12, then that addressing mode will be used to access the variable.

Function modifiers

The modifier **_ _protected** causes a function to be generated that will run with all interrupts disabled. The current interrupt state is saved when the function is entered, the interrupts are disabled and the body of the function is executed. The previous interrupt state is restored when the function returns. The disabling of interrupts only applies to those interrupts that can be disabled under software control.

The **_ _ interrupt** modifier specifies a function to be generated that can be used directly as an interrupt handler. A function declared with the **_ _interrupt** modifier must not be called directly from C since the function performs a return from interrupt when it is finished. The name of an interrupt function can be used in the **Vectors** tab of the project editor to define an exception handler. An interrupt function should return **void** and have a **void** parameter list:

```
__interrupt void handler(void)
        {
        ...
        }
```

Calling convention override

Introl−C also recognizes the extended type modifiers _ _**cdecl** and _ _**pascal** when used in function declarations and definitions. They cause the compiler to generate functions and calls to functions using, respectively, the ``caller−cleanup" function linkage or the ``return−and−deallocate" linkage See Function calling conventions.

These keywords have been provided to help migrate from versions of Introl−C prior to 3.07. Prior to version 3.07, Introl−C only supported a single calling convention. The function calling convention for the current version of Introl−C as well as examples of _ _**cdecl** and _ _**pascal** are described in Calling Conventions.

Asm keyword

The Introl−C compiler recognizes a the keyword **asm**. It can be used in two ways: as a pseudo−function and to signal the start of an multi−line assembly block.

In the first form, the compiler interprets it as a request to insert a string directly into the assembler source file. The function is used with the following form:

```
asm(string);
```

All legal C escape sequences are permitted in the *string*. For example, a table of numbers could be created as:

```
asm("table:\tdc.w\t1");
asm("\tdc.w\t2");
asm("\tdc.w\t3");
```

The same table could be created as:

```
asm("table:\tdc.w\t1\n\tdc.w\t2\n\tdc.w\t3");
```

In the second form, **asm** is followed by a **{** and all text until the next **}** is passed to the assembler:

```
asm
{
table:  dc.w    1
        dc.w    2
        dc.w    3
}
```

Note that in both forms of **asm**, a label, if present, must appear in the first column of the assembly language line. Both forms of **asm** may appear both inside and outside of C functions.

It is also possible to access C functions and data from separate assembly language files, see Calling Conventions.

Symbol name length

Symbol names (i.e. the names of functions, variables, structure members, etc.) in Introl−C may be of any length, but only the first 90 characters are significant. You can use the **−y** option to lower the limit to help in porting old source code that depends on names being truncated.

Symbols in C and Assembly

The function and variable name symbols generated by compiling a C program into an object file are unadorned. That means no additional characters are added to the symbols generated. If you name a C function **func**, it can be referred to as **func** in an assembly language program.

There is an exception to this rule. If the C identifiers that are the same as processor registers have to be "back quoted" in an assembly language program. For example, an assembly language file using a symbol named **x** (on the 68HC12) or **d0** (on the 68332) would need to refer to them as **`x`** and **`d0`** respectively.

Here is an example. The C file contains:

```
int     integer;
char    data[10];

int func(void)
        {
        ...
        }
```

and the assembly file contains:

```
        import  integer,data,func       ; import names from the C definitions

        section .text
        ldd     integer                 ; get the integer variable
        ldx     #data                   ; get the address of the array
        jsr     func                    ; cal the C function
```

C Calling Conventions

These sections describe the function calling conventions used by Introl−C including the mechanisms used for passing parameters to functions, returning values from functions as well as the ways in which the processor's registers are used. If you want to link your own assembly−written functions with a C program, you will have to follow the compiler's conventions.

6809
: The 6809 C compiler calling conventions and register usage.

68HC11
: The 68HC11 C compiler calling conventions and register usage.

68HC12
: The 68HC12 C compiler calling conventions and register usage.

68HC16
: The 68HC16 C compiler calling conventions and register usage.

68XXX
: The 68XXX family C compiler calling conventions and register usage.

6809

Parameters

Parameters are, in general, passed to a function by pushing them onto the stack. Since the processor's stack grows toward lower addresses, the last C parameter is pushed first and the first C parameter is pushed last (i.e. the parameters are pushed onto the stack in right−to−left order).

The first parameter passed to a function in passed in the **D** register (for 8 and 16 bit data) and the **U** and **D** registers for (32 bit data). For 32 bit data, **U** contains the upper 16 bits. If the first parameter is larger than 32 bits, or is a **struct** or **union**, then it is passed on the stack.

Local variables

If a function contains local variables, the memory used to store them is allocated on the stack at the beginning of the function. Since the compiler may create temporary local variables of its own, the amount of stack space allocated may be greater than the requirements for the user defined local variables.

Return values

Scalar return values

8−, 16− and 32 bit values are returned in the **D** or **U** and **D** registers. For example, the function:

```
int func()
        {
        return 5;
        }
```

would cause the compiler to generate the following assembly code for the return statement:

```
        ldd         #5
        ...
        rts
```

and the function:

```
long func()
        {
        return 0x12345678;
        }
```

would cause the compiler to generate the equivalent of:

```
        ldu         #$1234
        ldd         #$5678
        ...
        rts
```

Aggregate return values

Aggregates (structures) are returned by copying them into static storage and returning a pointer to the copy in the **D** register. Introl−C allocates the storage for aggregate return values by defining a common section named **.retm**. After the function returns, the caller will use the returned pointer to copy the data. For example,

given a function returning a 10 byte structure:

```
struct st
        {
        char array[10];
        };

struct st func(void)
        {
        struct st rval;

        return rval;
        }
```

the compiler generates code to copy the local structure rval to the common area and then loads the D0 register with the address of the common area:

```
        ...
        leax            0,s             ; address of rval
        ldu             #.retm          ; address of the common area
        ...                             ; copy 10 bytes from (X)
                                        ; to (U)
        ldd             #.retm          ; load the return value
        leas            10,s            ; remove rval
        rts
        ...
        section         .retm,comm      ; allocate the common area
        ds.b            10
```

Register use and preservation rules

Introl−C for the 6809 generates code with the assumption that no registers will be preserved across a function call. All registers except the stack pointer can be destroyed.

Stack pointer considerations

All parameters passed to a function are deallocated by the caller. Functions should not modify the stack pointer (they can use the stack pointer during their execution; its value after returning should be the same as it was upon entry).

Controlling the calling convention

The **_ _pascal** and **_ _cdecl** keywords have no effect on the 6809.

68HC11

Parameters

Parameters are, in general, passed to a function by pushing them onto the stack. Since the processor's stack grows toward lower addresses, the last C parameter is pushed first and the first C parameter is pushed last (i.e. the parameters are pushed onto the stack in right−to−left order).

The first parameter passed to a function in passed in the **D** register (for 8 and 16 bit data) and the **Y** and **D** registers for (32 bit data). For 32 bit data, **Y** contains the upper 16 bits. If the first parameter is larger than 32 bits, or is a **struct** or **union**, then it is passed on the stack.

Local variables

If a function contains local variables, the memory used to store them is allocated on the stack at the beginning of the function. Since the compiler may create temporary local variables of its own, the amount of stack space allocated may be greater than the requirements for the user defined local variables.

Return values

Scalar return values

8−, 16− and 32 bit values are returned in the **D** or **Y** and **D** registers. For example, the function:

```
int func()
        {
        return 5;
        }
```

would cause the compiler to generate the following assembly code for the return statement:

```
        ldd         #5
        ...
        rts
```

and the function:

```
long func()
        {
        return 0x12345678;
        }
```

would cause the compiler to generate the equivalent of:

```
        ldy         #$1234
        ldd         #$5678
        ...
        rts
```

Aggregate return values

Aggregates (structures) are returned by copying them into static storage and returning a pointer to the copy in the **D** register. Introl−C allocates the storage for aggregate return values by defining a common section named **.retm**. After the function returns, the caller will use the returned pointer to copy the data. For example,

given a function returning a 10 byte structure:

```
struct st
        {
        char array[10];
        };

struct st func(void)
        {
        struct st rval;

        return rval;
        }
```

the compiler generates code to copy the local structure rval to the common area and then loads the D0 register with the address of the common area:

```
        ...
        tsx                             ; address of rval
        ...
        ldd             #.retm          ; address of the common area
        ...                             ; copy 10 bytes from (X)
                                        ; to (Y)
        ldd             #.retm          ; load the return value
        pulx                            ; remove rval
        pulx
        pulx
        pulx
        pulx
        rts
        ...
        section         .retm,comm      ; allocate the common area
        ds.b            10
```

Register use and preservation rules

Introl−C for the 68HC11 generates code with the assumption that no registers will be preserved across a function call. All registers except the stack pointer can be destroyed.

Stack pointer considerations

Functions defined or declared with a prototype parameter list without an ellipsis are expected to pop their parameters off the stack before returning. All other functions should not modify the stack pointer (they can use the stack pointer during their execution; its value after returning should be the same as it was upon entry).

These two conventions for managing the parameter area on the stack are known, respectively, as ``return−and−deallocate" and ``caller−cleanup". The following paragraphs contain examples of these calling conventions. They show the code produced by Introl C for function calls and for function entry and exit. In your assembler written functions, you can use any code sequence that has the same effect (i.e. you need not use exactly the same code as the compiler).

A function defined with a prototype but without an ellipsis will use return−and−deallocate. Consider the function:

```
func(int a, int b, int c)
        {
```

```
        ...
        }
```

It will cause the compiler to generate the following code (but not the comments):

```
func:   fbegin
        pshb                            ; save the first parameter
        psha                            ;
        ...                             ; function body
        pulx                            ; remove the first parameter
        puly                            ; pop the function return address
        pulx                            ; deallocate the next 2 parameters
        pulx                            ;
        jmp             0,y             ; return from the function
```

If you write a function in assembly language for which the C compiler will see a prototype without an ellipsis, you must use the return−and−deallocate convention before your function returns. This only applies to functions that take parameters. If a function does not take any parameters, there is no difference between the two conventions.

Controlling the calling convention

The **_ _pascal** and **_ _cdecl** keywords can be used in function declarations and definitions to override the default calling convention. The **_ _cdecl** keyword will cause the compiler to use the caller−cleanup convention and the **_ _pascal** keyword will cause the compiler to use return−and−deallocate. For example, in the function:

```
void func(void)
        {
        extern void func1(int, int);
        extern void _ _cdecl func2(int, int);
        extern void _ _pascal func3(int, int);

        void (* fptr1)(int, int) = func1;
        void (* _ _cdecl fptr2)(int, int) = func2;
        void (* _ _pascal fptr3)(int, int) = func3;

        func1(1, 2);
        fptr1(1, 2);
        func2(2, 3);
        fptr2(2, 3);
        func3(3, 4);
        fptr3(3, 4);
        }
```

the call to *func1* will use return−and−deallocate because it was defined with a prototype; the call to *func2* will use return−and−deallocate because it was declared with the **_ _pascal** keyword and the call to *func3* will use caller−cleanup because it was declared with the **_ _cdecl** keyword.

68HC12

Parameters

Parameters are, in general, passed to a function by pushing them onto the stack. Since the processor's stack grows toward lower addresses, the last C parameter is pushed first and the first C parameter is pushed last (i.e. the parameters are pushed onto the stack in right−to−left order).

The first parameter passed to a function in passed in the **D** register (for 8 and 16 bit data) and the **Y** and **D** registers for (32 bit data). For 32 bit data, **Y** contains the upper 16 bits. If the first parameter is larger than 32 bits, or is a **struct** or **union**, then it is passed on the stack.

Local variables

If a function contains local variables, the memory used to store them is allocated on the stack at the beginning of the function. Since the compiler may create temporary local variables of its own, the amount of stack space allocated may be greater than the requirements for the user defined local variables.

Return values

Scalar return values

8−, 16− and 32 bit values are returned in the **D** or **Y** and **D** registers. For example, the function:

```
int func()
        {
        return 5;
        }
```

would cause the compiler to generate the following assembly code for the return statement:

```
        ldd         #5
        ...
        rts
```

and the function:

```
long func()
        {
        return 0x12345678;
        }
```

would cause the compiler to generate the equivalent of:

```
        ldy         #$1234
        ldd         #$5678
        ...
        rts
```

Aggregate return values

Aggregates (structures) are returned by copying them into static storage and returning a pointer to the copy in the **D** register. Introl−C allocates the storage for aggregate return values by defining a common section named **.retm**. After the function returns, the caller will use the returned pointer to copy the data. For example,

given a function returning a 10 byte structure:

```
struct st
        {
        char array[10];
        };

struct st func(void)
        {
        struct st rval;

        return rval;
        }
```

the compiler generates code to copy the local structure rval to the common area and then loads the D0 register with the address of the common area:

```
        ...
        leax            0,sp            ; address of rval
        ldy             #.retm          ; address of the common area
        ...                             ; copy 10 bytes from (X)
                                        ; to (Y)
        ldd             #.retm          ; load the return value
        leas            10,sp           ; remove rval
        rts
        ...
        section         .retm,comm      ; allocate the common area
        ds.b            10
```

Register use and preservation rules

Introl−C for the 68HC12 generates code with the assumption that no registers will be preserved across a function call. All registers except the stack pointer can be destroyed.

Stack pointer considerations

All parameters passed to a function are deallocated by the caller. Functions should not modify the stack pointer (they can use the stack pointer during their execution; its value after returning should be the same as it was upon entry).

Controlling the calling convention

The **_ _pascal** and **_ _cdecl** keywords have no effect on the 68HC12.

68HC16

Parameters

Parameters are, in general, passed to a function by pushing them onto the stack. Since the processor's stack grows toward lower addresses, the last C parameter is pushed first and the first C parameter is pushed last (i.e. the parameters are pushed onto the stack in right−to−left order).

The first parameter passed to a function in passed in the **D** register (for 8 and 16 bit data) and the **E** and **D** registers for (32 bit data). For 32 bit data, **E** contains the upper 16 bits. If the first parameter is larger than 32 bits, or is a **struct** or **union**, then it is passed on the stack.

Local variables

If a function contains local variables, the memory used to store them is allocated on the stack at the beginning of the function. Since the compiler may create temporary local variables of its own, the amount of stack space allocated may be greater than the requirements for the user defined local variables.

Return values

Scalar return values

8−, 16− and 32 bit values are returned in the **D** or **E** and **D** registers. For example, the function:

```
int func()
        {
        return 5;
        }
```

would cause the compiler to generate the following assembly code for the return statement:

```
        ldd         #5
        ...
        rts
```

and the function:

```
long func()
        {
        return 0x12345678;
        }
```

would cause the compiler to generate the equivalent of:

```
        lde         #$1234
        ldd         #$5678
        ...
        rts
```

Aggregate return values

Aggregates (structures) are returned by copying them into static storage and returning a pointer to the copy in the **E** and **D** registers. Introl−C allocates the storage for aggregate return values by defining a common section named **.retm**. After the function returns, the caller will use the returned pointer to copy the data. For

example, given a function returning a 10 byte structure:

```
struct st
        {
        char array[10];
        };

struct st func(void)
        {
        struct st rval;

        return rval;
        }
```

the compiler generates code to copy the local structure rval to the common area and then loads the D0 register with the address of the common area:

```
        ...
        tzx                             ; address of rval
        ldab            #(.retm>>16)    ; address of the common area
        ldy             #.retm          ; address of the common area
        ...                             ; copy 10 bytes from (X)
                                        ; to (Y)
        lde             #(.retm)>>16    ;
        ldd             #.retm          ; load the return value
        ais             #10             ; remove rval
        pulm            z,k             ;
        rts
        ...
        section         .retm,comm      ; allocate the common area
        ds.b            10
```

Register use and preservation rules

Introl−C for the 68HC16 generates code with the assumption that the **K** and **Z** registers will be preserved across a function call. All other registers except the stack pointer can be destroyed.

Stack pointer considerations

All parameters passed to a function are deallocated by the caller. Functions should not modify the stack pointer (they can use the stack pointer during their execution; its value after returning should be the same as it was upon entry).

Controlling the calling convention

The **_ _pascal** and **_ _cdecl** keywords have no effect on the 68HC16.

68XXX

Parameters

Parameters are, in general, passed to a function by pushing them onto the stack. Since the processor's stack grows toward lower addresses, the last C parameter is pushed first and the first C parameter is pushed last (i.e. the parameters are pushed onto the stack in right−to−left order).

Local variables

If a function contains local variables, the memory used to store them is allocated on the stack at the beginning of the function. Since the compiler may create temporary local variables of its own, the amount of stack space allocated may be greater than the requirements for the user defined local variables.

Return values

Scalar return values

8−, 16− and 32 bit values are returned in the **D0** register. Double precision floating point values (double) are returned in both the **D1** and **D0** registers; the **D1** register contains the most significant 32 bits. For example, the function:

```
int func()
        {
        return 5;
        }
```

would cause the compiler to generate the following assembly code for the return statement:

```
        move.l          #5,d0
        ...
        rts
```

and the function:

```
double func()
        {
        return 1.0;
        }
```

would cause the compiler to generate:

```
        move.l          #$3ff00000,d0
        move.l          #0,d1
        ...
        rts
```

Aggregate return values

Aggregates (structures) are returned by copying them into static storage and returning a pointer to the copy in the **D0** register. Introl−C allocates the storage for aggregate return values by defining a common section named **.retm**. After the function returns, the caller will use the returned pointer to copy the data. For example, given a function returning a 10 byte structure:

```
struct st
        {
        char array[10];
        };

struct st func(void)
        {
        struct st rval;

        return rval;
        }
```

the compiler generates code to copy the local structure rval to the common area and then loads the **D0** register with the address of the common area:

```
        ...
        lea             (−12,fp),a0     ; address of rval
        lea             .retm,a1        ; address of the common area
        ...                             ; copy 10 bytes from (0,A0)
                                        ; to (0,A1)
        move.l          #.retm,d0       ; load the return value
        rts
        ...
        section         .retm,comm      ; allocate the common area
        ds.b            10
```

Register use and preservation rules

Introl−C generates code with the assumption that certain registers will be preserved across a function call. Other registers can be destroyed. The following table indicates whether the compiler expects a register to be preserved across function call (as well as whether the register is used for return values − in which case it is implicitly allowed to be destroyed):

Register	Preserved	Return value
D0 .. D1		**X**
D2 .. D7	**X**	
A0 .. A1		
A2 .. A6	**X**	
A7	(see below)	
CCR		
FP0		**X**
FP1 .. FP2		
FP3 .. FP7	**X**	

Stack pointer considerations

Functions defined or declared with a prototype parameter list without an ellipsis are expected to pop their parameters off the stack before returning. All other functions should not modify the stack pointer (they can use the stack pointer during their execution; its value after returning should be the same as it was upon entry).

These two conventions for managing the parameter area on the stack are known, respectively, as ``return−and−deallocate" and ``caller−cleanup". The following paragraphs contain examples of these calling conventions. They show the code produced by Introl C for function calls and for function entry and exit. In your assembler written functions, you can use any code sequence that has the same effect (i.e. you need not use exactly the same code as the compiler).

A function defined with a prototype but without an ellipsis will use return−and−deallocate. Consider the function:

```
func(int a, int b, int c)
        {
        ...
        }
```

It will cause the compiler to generate the following code (but not the comments):

```
func:   link            fp,#0           ; create the stack frame
        ...                             ; function body
        unlk            fp              ; unlink the stack frame
        move.l          (sp)+,a0        ; pop the function return address
        lea             (12,sp),sp      ; deallocate the 3 parameters
        jmp             (a0)            ; return from the function
```

The compiler will use the **rtd** instruction for processors that support it. The code shown is for the 68000/08.

If you write a function in assembly language for which the C compiler will see a prototype without an ellipsis, you must use the return−and−deallocate convention before your function returns. This only applies to functions that take parameters. If a function does not take any parameters, there is no difference between the two conventions.

Controlling the calling convention

The **_ _pascal** and **_ _cdecl** keywords can be used in function declarations and definitions to override the default calling convention. The **_ _cdecl** keyword will cause the compiler to use the caller−cleanup convention and the **_ _pascal** keyword will cause the compiler to use return−and−deallocate. For example, in the function:

```
void func(void)
        {
        extern void func1(int, int);
        extern void _ _cdecl func2(int, int);
        extern void _ _pascal func3(int, int);

        void (* fptr1)(int, int) = func1;
        void (* _ _cdecl fptr2)(int, int) = func2;
        void (* _ _pascal fptr3)(int, int) = func3;

        func1(1, 2);
        fptr1(1, 2);
```

```
func2(2, 3);
fptr2(2, 3);
func3(3, 4);
fptr3(3, 4);
}
```

the call to *func1* will use return−and−deallocate because it was defined with a prototype; the call to *func2* will use return−and−deallocate because it was declared with the **_ _pascal** keyword and the call to *func3* will use caller−cleanup because it was declared with the **_ _cdecl** keyword.

C Memory Models

These sections describe the memory models supported by Introl−C for those processors that have models in addition to the normal flat address space model.

68HC11
: The 68HC11 C compiler memory models.

68HC12
: The 68HC12 C compiler memory models.

68HC16
: The 68HC16 C compiler memory models.

68HC11

The compiler supports both 16− and 32 bit addresses for functions. A 16 bit address points to a *near* function whereas a 32 bit address points to a *far* function. 32 bit data pointers are not supported.

Functions and pointers to functions

Far function pointers are represented by the compiler as a 32 bit value. The value of a far function pointer is packed in the 32 bits with the upper 8 bits counting zero, the next 8 bits containing the bank number and the lower 16 bits containing the bank offset of the function.

Conversions between near and far

Conversions between near and far pointers can occur in either direction. A near pointer (i.e. a 16 bit pointer) is converted to a far pointer (i.e. a 32 bit pointer) by extending it with zero. A far pointer is converted to a near pointer by truncating it to 16 bits. This causes a loss of significance and although legal and supported, does not make much sense in real programs; the compiler will generate a warning when a far pointer is truncated to a near pointer.

Implementation

For the 68HC11, bank switching in CODE is implemented using three pseudo instructions that are understood by the C compiler and 68HC11 assembler and that are implemented using the 68HC11's illegal opcode trap. The three instructions are **CALL**, **CALLX**, and **RTC**. **CALL** is a four byte instruction that implements a banked function call. The first byte of the instruction is an illegal 68HC11 opcode, the section byte contains a bank number, and the next two bytes contain the offset in the bank. The **CALLX** instruction works similarly to the **CALL** instruction except that the **X** register points to the function pointer of the banked function to be called. **CALLX** is a two byte instruction. **RTC** is a two byte instruction that implements a return from a far function.

The actual implementation of the banking pseudo instructions is done in the library function callrtc.s, which is located in the libgena.11 library.

68HC12

The compiler supports both 16− and 32 bit addresses for functions. A 16 bit address points to a *near* function whereas a 32 bit address points to a *far* function. 32 bit data pointers are not supported.

Functions and pointers to functions

Far function pointers are represented by the compiler as a 32 bit value. The value of a far function pointer is packed in the 32 bits with the upper 16 bits containing the bank offset of the function, the next 8 bits containing the page number of the function, and the lowest 8 bits containing zero.

Conversions between near and far

Conversions between near and far pointers can occur in either direction. A near pointer (i.e. a 16 bit pointer) is converted to a far pointer (i.e. a 32 bit pointer) by extending it with zero. A far pointer is converted to a near pointer by truncating it to 16 bits. This causes a loss of significance and although legal and supported, does not make much sense in real programs; the compiler will generate a warning when a far pointer is truncated to a near pointer.

Libraries

Two C libraries for the 68HC12 are supplied with CODE. The library libc.a12 is the standard library, built normally. The library libcl.a12 is build in large model: all the functions defined in it are far functions and can be used in banked systems.

68HC16

The compiler supports both 16− and 32 bit addresses for functions and variables. A 16 bit address points to a *near* object whereas a 32 bit address points to a *far* object. Near objects are assumed to be in a bank indicated by the initial value loaded into all of the **K** fields at program start−up time (i.e. at reset). The initial value loaded into all of the **K** fields is the bank number of what we call the *near bank*. Near objects reside in the near bank. Far objects can reside anywhere, including the near bank.

Compiler assumptions

All of the **K** fields point to the near bank at the beginning of every function; this eliminates the need to load a **K** field when a near object is accessed. Therefore, a function that only accesses near data will never have to reload a K field. Once a far object has been accessed, however, it is very likely that one of the K fields will have to be reloaded for subsequent accesses to near data.

Another assumption made by the compiler is that the stack is in the near bank. This limits the total size of stack and the uninitialized external near data to 64K. In practice, this assumption should be easy to satisfy for most programs by placing the external data at the bottom of a contiguous RAM region and by placing the stack at the top. The −gk and −gl command line options disable this assumption. By assuming that the stack and external near data sections are contained within the same bank, the compiler can omit code that it would otherwise have to generate to preserve the **K** register in the face of instructions such as adx or aix which might modify a **K** field. Note that this assumption doesn't impose any restrictions on the use or amount of far data in a program; it exists solely for providing the best efficiency in a program that contains mainly near data accesses. Also, by keeping the stack in the near bank, the addresses of local data will fit in a short pointer.

Functions and pointers to functions

Functions are always treated as far objects and pointers to functions are always treated as far pointers; therefore, the new declaration keywords described below do not apply to functions. This means that program code is always, in effect, far.

When a function's address is used in an expression, the compiler generates an address 2 bytes greater than its real address. This allows function pointers to be more naturally used by the **rts** instruction (which is the only instruction capable of branching to a variable address). In the following example, if the function main were at address 1000, x would be assigned the value 1002.

```
void main(void)
        {               /* assume that main is at address 1000 */
        void (*x)(void);
        x = main;       /* x now has the value of 1002 */
        }
```

Defining far and near data

There are two extended keywords the compiler recognizes to modify the near or far attributes of data. The C declaration syntax has been extended to allow the words **_ _near** and **_ _far** wherever the keyword **const** would be legal. In practice, we recommend using the C preprocessor to create more usable substitutes for these names such as:

```
#define near _ _near
#define far _ _far
```

In the absence of these keywords, the attribute is assumed to be near (unless the −gl option is specified). For example, the following two definitions both define a near integer:

```
int nearint;
int _ _near nearint;
```

Pointers may point to either near data or far data. A pointer that points to near data is called a *near pointer* whereas a pointer that points to far data is called a *far pointer*, though a far pointer can also point to near data. As is the case with the **const** modifier, these new modifiers apply to the part of the declaration that is to their right. For example, the following defines a pointer, itself near (note that the **_ _near** wouldn't normally be needed since it is implicit), to a far integer (i.e. a *far pointer*):

```
int _ _far * _ _near farptr;
```

A far pointer is 32 bits, but only the lower 20 bits are used for addressing. A near pointer is 16 bits and when using one, the compiler uses the near bank for the rest of the address.

Far ellipsis

An extension to the function prototype syntax can be used to tell the compiler that all indirection performed on the function's variable arguments should use far accesses. We call this syntax *far ellipsis*. In the following standard prototype:

```
extern void func(int, char *, ...);
```

the function takes an integer, a pointer to a character and a variable number of additional arguments. This declaration, however, doesn't tell the compiler anything about the types of the variable arguments. Introl−C will pass a near pointer as 16 bits and it will pass a far pointer as 32 bits (i.e. it will not convert them). The far ellipsis syntax allows you to make a function that will receive all of its pointer arguments as far pointers. If the function were declared as:

```
extern void func(int, char *, ... _ _far);
```

the compiler assumes that all near pointers passed as the variable arguments should first be converted to far pointers. All of the variable argument functions in the standard library are declared this way. This feature allows us, for example, to make a single printf function that can be used regardless of whether it is passed short pointers or long pointers for its **%s** format strings.

Conversions between near and far

Conversions between near and far pointers can occur in either direction. A near pointer (i.e. a 16 bit pointer) is converted to a far pointer (i.e. a 32 bit pointer) by extending it with the value of the near bank. A far pointer is converted to a near pointer by overwriting its bank value with the value of the near bank. This causes a loss of significance and although legal and supported, does not make much sense in real programs; the compiler will generate a warning when a far pointer is truncated to a near pointer.

If a function prototype exists that declares parameters as far pointers, the compiler will automatically insert conversions for near pointers that are passed to the function as parameters. Also, in expressions involving both types of pointers, the compiler will widen near pointers as appropriate. For example, the following code fragment is perfectly valid and will contain an implicit conversion of nearptr to a far pointer:

```
int _ _near * _ _near nearptr;
```

```
int _ _far * _ _near farptr;
farptr = nearptr;
```

Allocation of strings

As described in Section allocation the compiler generates data in sections that depend on the data's storage class (and also whether the variable has been initialized). Strings, however, aren't explicitly defined, the section in which they are placed is determined by their context. If a string is used in the context of a far pointer, the string's data are placed in the .fstrngs section, otherwise they are placed in the .strings section. For example, in the following code fragment, the first string is near and the second is far:

```
char _ _near *near_pointer;
char _ _far *far_pointer;

near_pointer = "strings";       /* allocated in .strings */
far_pointer = "fstrngs";        /* allocated in .fstrngs */
```

Overriding the defaults

As mentioned previously, in the absence of a **_ _near** or **_ _far** qualifier, the compiler behaves as if **_ _near** were used. The presence of either of these qualifiers will disable the compiler's default behavior. For example, the following definitions are semantically identical:

```
int ***x;
int *** _ _near x;
int ** _ _near * _ _near x;
int * _ _near * _ _near * _ _near x;
int _ _near * _ _near * _ _near * _ _near x;
```

The important feature to note is that the compiler effectively inserts **_ _near** in every allowable context in the absence of an explicit qualifier. This applies to all declaration−related code including casts, structures, arrays, etc.

This default behavior can be modified with the −gl option which causes the compiler to effectively insert **_ _far** in every allowable context (where an explicit **_ _near** or **_ _far** does not already exist) and to place all strings in the **.fstrngs** section regardless of their context. Furthermore, this option disables the compiler's assumption that the stack and uninitialized data reside in the same bank.

The assumption that the stack is in the near bank can be disabled independently of the implied qualification assumption by using the −gk option. This option is required, essentially, any time you place your stack in a different bank than the near data. It does not, however, change the compiler's behavior when assigning the address of local data to a short pointer (i.e. the compiler will still only generate the lowest 16 bits of the address and store them into the short pointer without any warning). Addresses of local data that are used in a far context, however, will work just fine; in other words, if the stack is not in the near bank (and you are using the **−gk** option), you must make sure that all uses of addresses of local variables occur in a far context − such as assigning to a far pointer.

Programming recommendations

We have implemented the **−gl** option to allow existing large programs (*large*, in this context, means that the program needs to use more than 64K of data) to be easily ported to the 68HC16 without modification. The use of this option, however, will generate much more and much slower code than a program written to use far data only when necessary.

A program designed for the mixed memory model environment provided by Introl's 68HC16 compiler should define most, if not all, of its global and static data as near (by not using any qualifier). This will greatly reduce code size and and increase execution speed. Only those data items that need to be large should be declared that way.

Libraries

All the C−written functions in the pre−compiled libraries we distribute with the compiler have been compiled with the **−gl** command line option (the assembler written functions have been written to assume far pointers). This should not have a very big impact on performance since most of the library functions that are potential bottlenecks (such as the string and memory functions) are coded in assembly and have no performance loss for assuming far pointers. You must include the proper header file for all library function you use so the compiler converts the parameters and return values appropriately (it will generate a warning for any function call that doesn't have a corresponding prototype).

C Position Independence

The 6809 and 68XXX family compilers can generate programs that have position independent code and data. The **−gr** option controls the generation of position independent code and data. The **−grd** option enables position independent data, the **−grc** option enables position independent code and, combined as **−grdc**, both position independent code and data are produced.

Position independent code is supported by using *PC−relative* addressing for all external function calls. Also, PC−relative addressing is used to compute the addresses of functions.

Position independent data is supported by using *base register* addressing for all accesses to non constant global and static data (including strings). Constant data is referenced using PC−relative addressing. The **−C** compiler option will make program strings constant and thus cause them to be accessed using PC−relative addressing also. The **A5** (for the 68XXX family) or the **Y** (for the 6809) register is used as the base and it must be loaded with an address that points to the global data area before the program is started. When linking a program to use position independent data, the linker command file should be set as if the data started at address zero. The actual data start address will be loaded into the base register at runtime.

The libraries we ship with the compiler have not been designed to support position independent code or data; however, since we ship their source code, you should be able to build a version of the library that works properly. Of all the C−written functions in the library, only the **malloc** function (as well as its internal support function **_ _sbrk**) use any undocumented static data. In the case of these two functions, they each use a static pointer.

The code generator support libraries *libgen.a09* and *libgen.a68* (and others for the 68XXX family) have not, specifically, been written to avoid directly accessing global addresses (using absolute addressing). They may contain absolute references to code via the jsr instruction. The *libgen.a09* and *libgen.a68* libraries do not use any static or global data so they have no conflict with position independent data.

C Hardware Floating Point

This chapter applies to the 68XXX family only. Introl−C can support floating point numbers by generating code for the 68881/2 coprocessor, the 68040's built in floating point instructions, or by using a software library. The method of floating point support is determined by both the command line options used to compile your program and by the libraries with which it is linked.

The **−gm** option controls the generation of floating point instructions. Without the option, the compiler generates calls to a software support library. An optional argument to the **−gm** option selects the instruction set generated as indicated in the list below:

−gm
: Generate floating point instructions for the 68881/68882.

−gm1
: Generate floating point instructions for the 68040 and also generate the **fintrz** instruction.

−gm2
: Synonymous with **−gm**.

−gm5
: Generate floating point instructions for the 68040. Use in−line code for rounding to an integer.

−gm9
: Generate floating point instructions for the 68040. Use a library function for rounding to an integer.

For the 68020 or 68030 with a 68881 or 68882, you should use the **−gm** option; use the **−gm5** option for the 68040. Since the integer argument in the **−gm** option is actually a bit mask indicating target processor capabilities as indicated in the following table, the compiler can, for example, generate code using floating point instructions that will run on both the 68881/2 and the 68040 (by using the **−gm6** option).

Bit	Bit Value	Value	Description
0	1	0	Do not generate floating point instructions.
1	Use the 68040 floating point instruction set (mutually exclusive with bit 1).		
1	2	0	Do not generate floating point instructions.
1	Use the 68881/2 floating point instruction set (mutually exclusive with bit 0).		
2	4	0	Generate the fintrz instruction for floating−point−to−integer conversions.
1	Generate an in−line code sequence instead of using the fintrz instruction for floating−point−to−integer conversions (mutually exclusive with bit 4).		
3	8	0	Generate the fintrz instruction for

			floating−point−to−integer conversions.
1	Generate a function call instead of using the fintrz instruction for floating−point−to−integer conversions (mutually exclusive with bit 3).		

Intrinsic floating point functions

If any **−gm** option is in effect, the compiler will generate a single floating point instruction for many standard library functions as long as the header file *math.h* has been included. You can disable intrinsic function generation on a per−function basis by **#undef**−ing the function. The functions generated as intrinsic are listed in the following table:

Function	Instruction
fabs	fabs
asin	fasin
cos	fcos
exp	fetox
sinh	fsinh
tanh	ftanh
log	flog
acos	facos
atan	fatan
cosh	fcosh
sin	fsin
tan	ftan
log10	flog10

For example, in the following function, the call to **sqrt** will cause the compiler to generate the **fsqrt** instruction and the call to **sin** will cause the compiler to generate a **jsr sin**:

```
#include <math.h>
#undef sin

func()
        {
        extern double a, b, c;

        a = sqrt(c);
        b = sin(c);
        }
```

The compiler generates the following code when invoked with the **−gm** option:

```
fmove.d         c,fp0           ; load c
fsqrt.x         fp0             ; compute its square root
fmove.d         fp0,a           ; store the result in a
move.l          c+4,−(sp)       ; push c as an argument to sin
move.l          c,−(sp)
jsr             sin             ; compute its sine using the function
fmove.d         fp0,b           ; store the result in b
```

Note: the previous output is not exactly the code generated by the compiler. This is, however, equivalent to what the compiler generates. It has been rearranged for clarity.

In addition to using the proper **−gm** command line option, you must also link your program with a corresponding library. We have supplied several pre−compiled and pre−assembled versions of the libraries. You must use the version of the Introl libraries that correspond to your floating point hardware and your target processor. Our sample linker command files use versions of both that target a 683XX with software floating point (i.e. the files *libc.a68* and *libgen.a68* are used).

Program Sections

These sections describe program sections and how they are used by CODE.

What are Sections?

A *section* is an atomic contiguous unit representing a memory region. Every assembly language source file, including those produced by a compiler, defines one or more sections. Sections can be *anonymous*, with only an absolute start address, or can be *named*. Named sections can be absolute, that is, given an absolute start address, or relocatable. Relocatable sections will be given an actual start address at link time.

Sections are used to partition a program between different memory regions. For example, Introl compilers put a program's executable code in a section named **.text**, constant data in a section called **.const**, strings in a section called **.strings**, writeable initialized data in a section called **.data**, and writeable uninitialized data in a section called **.bss**. Typically, the program's **.text**, **.const**, and **.strings** will be placed by the linker at an address where a read−only memory device, such as an EPROM, exists on the target system. The **.data** and **.bss** sections will usually be located in the target system's read/write memory. For more information on the compiler's use of sections, see Section allocation.

Section names are arbitrary: the Introl compilers use the previously mentioned names by convention. It is only at link time that the actual correspondence between a section name and a particular target memory region is established. See the linker manual for more information.

A named section is created by using the **section** directive. The following assembler source line creates a section called *.abc*:

```
section         .abc
```

When the assembler encounters a section directive, subsequent assembler source lines that generate code or data will place their data in that section. For example, the following is a complete assembler input file that creates a section named *.abc* containing directives to specify two constant bytes using the **dc** directive:

```
section         .abc
dc.b            5
dc.b            10
```

Objects in a section are referred to by their offset within the section. In the previous example, the constant 5 is at offset 0 of section *.abc* and the constant 10 is at offset 1.

The first time a section's name appears in a section directive, the assembler creates that section. Subsequent source lines then generate code in that section beginning at offset zero. If that same section name later appears in a section directive, the assembler will not re−create the section; instead, the assembler causes subsequent lines to generate code in that section beginning at an offset equal to the number of bytes of data currently in that section. For example, the following is completely equivalent to the previous example:

```
section         .abc
dc.b            5
section         .abc
dc.b            10
```

In this case, the second section directive causes the following **dc** directive to place its data in section *.abc* at offset 1 (since the section already contained 1 byte); in more complex programs, several sections are created. For example, the following is a complete assembler input file that creates the sections *.abc* and *.def*, both of which are two bytes long and contain the constants 5 and 10:

```
section         .abc    ; create .abc
dc.b            5       ; generate byte constant 5 in .abc
```

```
section         .def    ; create .def
dc.b            5       ; generate byte constant 5 in .def
section         .abc    ; switch back to .abc
dc.b            10      ; generate byte constant 10 in .abc
section         .def    ; switch back to .def
dc.b            10      ; generate byte constant 10 in .def
```

Section types

Sections are generic containers for code or data which you create using the **section** directive. They are considered generic containers because consumer programs such as a debugger or linker are unaware of their contents. To help a consumer program determine the content of the sections, you can mark a section with one of several flags to identify its contents.

One important property of a section is its *relocatability*. The differences between *absolute* and *relocatable* sections are described below.

Absolute sections

An absolute section is one whose final starting address (i.e. the load address) is determined at assembly time. The **addr** qualifier to the section directive is used to create an absolute section. For example the following code will create a section named *.abc* whose hexadecimal load address is $1000:

```
section         .abc,addr=$1000
dc.b            10
```

Anonymous absolute sections are created by using the **org** directive. The following example creates a section similar to the previous example, but it has no name:

```
org             $1000
dc.b            10
```

As with named sections, anonymous absolute sections are created the first time **org** is seen with a particular value. If the same value is later seen in an org directive, subsequent code will overwrite the code which was already placed in that section. For example, the following file creates two one byte sections:

```
org             $100
dc.b            1
org             $200
dc.b            2
```

whereas the following creates a single section (containing a single byte whose value is 5):

```
org             $100
dc.b            10
org             $100
dc.b            5
```

It is perfectly legal to create overlapping absolute sections. For example, the following creates two sections where the second overwrites the first:

```
org             $100
dc.b            10
dc.b            20
org             $101
```

```
                dc.b            30
```

Although overlapping sections are legal, it is bad practice to create files with such ambiguities, particularly if all the sections are anonymous. The reason is that the effect of overlapping depends solely on the behavior of the consumer of the object file. For example, if sections of the object file produced by the previous example are loaded into a target's memory by a loader program that loads each section in the order in which it appears in the object file, memory starting at $100 will contain the bytes 10 and 30. If the loader happened to load the sections in reverse order, the memory would contain the bytes 10 and 20. The assembler always creates sections in the object file corresponding to their order of definition in the assembler source file.

Relocatable sections

All sections which are not absolute are relocatable. A relocatable section's starting address is determined by the linker. At assembly time, the only thing known about the objects within a relocatable section is their offset from the section's beginning within the input file. Therefore, the value of labels defined in a relocatable section is not known at assembly time as is the case with the value of labels defined in an absolute section. For example, the following code will generate the value $1000:

```
                section         .abc,addr=$1000
abc                                             ; just a label on this line
                dc.l            abc
```

The following code will generate a value that is not known at assembly time:

```
                section         .abc
abc                                             ; just a label on this line
                dc.l            abc
```

The operand to the **dc** directive can be much more complex than just *abc*, but if it contains any relocatable symbols, the entire expression is relocatable and its value will not be known until the program is bound to an absolute address by the linker. A greater than sign (>) will appear to the left of the object code field in the assembly listing if the source line's operand contains a relocatable expression.

In the previous example, if the linker were instructed to place section *.abc* at address $1234, the **dc** directive would ultimately generate the value $1234 (because the label *.abc* is at offset 0 in the section).

Section semantics

As previously mentioned, the meaning of the contents of a section are subject to interpretation by the consumer of the object file. It is quite possible, for example, that a particular debugger program might depend on all sections named *.code* containing executable code, though Introl's debugger does not.

Various flags may be applied to a section via the **section** directive. Of all these section flags, four of them have standardized meanings in the definition of **ICOFF** files. A description of their semantics follows.

Regular sections

A regular section is one that is defined with no additional flags. All of the sections defined in the examples of this chapter so far have been regular sections. Regular sections have a starting address (if they are absolute), a size, and a number of bytes of code or data equal to the section's size.

BSS sections

It is often desirable to create a section representing an area of RAM used to hold variables, stacks, etc. Without the concept of *BSS*, or Blank Static Storage, sections, you might use the following type of code to create a 10 kilobyte section for a stack:

```
section         .stack
repeat          10240   ; repeat the next line 10240 times
dc.b            0
```

This will create a section named .stack consisting of 10240 zeroes but it is probably the least efficient way of creating an empty section. Not only is this equivalent to assembling a file with 10240 lines, but it creates a rather large object file since it must hold all the zeroes.

A BSS section does not contain any physical data; it only has a size. The contents of a BSS section, when the program is loaded into memory, depend on the program doing the loading. For example, if an object file is converted to Motorola S−records by Introl's **ihex** program, nothing will be generated for the BSS sections. Therefore when the S−records are loaded into memory, the memory corresponding to the BSS sections will not be touched. This is typical behavior and is, in fact, the normal procedure for program development.

Generally, if BSS areas need to contain some specific value, the program's initialization code handles it. For example, Introl's sample startup code for C programs clears a single BSS section to zeroes to produce the semantics required by statically allocated variables in C.

Given the concept of BSS sections, the stack section described above should be defined as follows:

```
section         .stack,bss
ds.b            10240
```

The **bss** qualifier to the **section** directive indicates that a BSS section should be created. Since BSS sections do not have any physical contents, only data storage defining directives, such as **ds**, are allowed to be placed in them. For example, the following is illegal:

```
section         .abc,bss
dc.b            5               ; illegal to put a constant
                                ; in a bss section
```

because it attempts to place a constant into a BSS section. It is not actually necessary to use the **bss** qualifier at all. The assembler will automatically generate a BSS section for any section that only contains data storage defining directives. For example, the following would also generate a 10240 byte .stack section:

```
section         .stack
ds.b            10240
```

Even though the assembler automatically generates BSS sections as appropriate, it is still recommended that you use the **bss** qualifier to catch typographical errors (such as misspelling the **ds** directive as **dc**).

Common sections

Normally, the linker concatenates the data from sections of the same name together so their data are stacked in contiguous order in memory. The data from *common* sections, however, are overlaid rather than concatenated. After linking, the size of a common section is equal to the size of the largest common section of the same name in any of the object files. The following code creates a 1024 byte common section (that will

also be a BSS section):

```
                section         .abc,comm
                ds.b            1024
```

Common sections are typically used to implement named common areas. For example, a data structure consisting of a two bytes and a word could be defined in one file as:

```
                section         .abc,comm
byte1           ds.b            1
byte2           ds.b            1
word1           ds.w            1
```

If a different assembler source file is defined:

```
                section         .abc,comm
longword1       ds.l            1
```

the effect of linking the object files together would be to overlay the loan−word with the data structure defined in the other file. The final address of *byte1* would equal that of *longword1*.

Offset sections

Offset sections are not true sections since they do not create a section of any type in the object file; instead, they create a sectionalize environment within the assembly source file that is used to create absolute symbols. An offset section is created with the **offset** directive.

As with BSS sections, only data storage defining directives may appear in an offset section. The operand to the offset directive must be an absolute expression. The expression's value is used as the base address for the section. The effect of an offset section is to assign an absolute value to any label appearing in the section. For example, the following will assign the value one to *label1* and the value four to *label2*:

```
                offset          0
                ds.b            1
label1          ds.b            3
label2
```

The following example is equivalent to the previous one except that the **equ** directive, which assigns a value to a symbol, is used instead of an offset section:

```
label1          equ             1
label2          equ             4
```

Offset sections are typically used to assign tag names to the various components of a data structure or to define variables that will exist in memory at a fixed address. You should use **offset** rather than the equivalent series of equ directives for these purposes when you know the sizes of the objects and want the assembler to calculate their offsets.

A C data structure defined as:

```
        struct xyz {
                int a;
                char b[8];
                int c;
                };
```

might be accessed in assembly code using names defined in an offset section as follows:

```
                        offset          0
member_a                ds.l            1       ; assume ?int? is 4 bytes
member_b                ds.b            8       ; assume ?char? is 1 byte
member_c                ds.l            1
size_xyz
```

The label *size_xyz* will be assigned the address following the last member and is useful in places where the size of the data structure is needed.

C Section Allocation

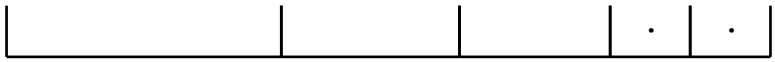
Introl−C generates assembler source files that contain code and data placed into sections. A section is named by a case sensitive string of up to 8 characters. The section into which an object is placed is summarized in the tables below:

Sections common to all compilers.

Object type	Section string	−a override	ROM	RAM
Executable code	.text	−at	.	
Constants	.const	−ac	.	
Initialized data	.data	−ad	.	
Strings	.strings	−as	.	
Uninitialized data	.bss,bss	−ab		.
_ _mod1_ _ data	.mod1	−a1	.	.
_ _mod2_ _ data	.mod2	−a2	.	.
_ _mod1_ _ _ _mod2_ _ data	.base	−ap	.	.

Sections used by compilers with memory models.

Object type	Section string	−a override	ROM	RAM
Far Executable code	.ftext	−aft	.	
Far Constants	.fconst	−afc	.	
Far Initialized data	.fdata	−afd	.	
Far Strings	.fstrngs	−afs	.	
Far Uninitialized data	.fbss,bss	−afb		.
Far _ _mod1_ _ data	.fmod1	−af1	.	.
Far _ _mod2_ _ data	.fmod2	−af2		



The ``−a override" column indicates which −a option is used to override the name of the section. For example, if you wanted executable code to be placed in a section named .code, you would use a command like:

```
cc12 −at=.code test.c
```

The column entitled ``Section string" shows the actual string that is used as an argument to the assembler's section directive. Since uninitialized data sections do not contain any physical contents, their sections use the **bss** qualifier to the **section** directive.

Note that **.mod1** and **.mod2** sections may be located in RAM or ROM since they are user defined. If you choose to put one of them in RAM, we recommend that you use the **bss** qualifier. For example, if you want to use **.mod2** as a RAM section, use the command line:

```
cc68 −a2=.mod2,bss test.c
```

Below are examples of some of the object types listed in the table and an assembler code fragment that shows what the compiler would generate.

Executable code

Executable code refers to the statements and expressions in functions. In the following examples, unless otherwise stated, data definitions and declarations that are shown without any context are assumed to be outside of any function. This means that the names they define are global and the compiler will place a colon (:) after the label to implicitly export the symbol. For example, given a source file containing:

```
func()
        {
        ...
        }
```

the compiler would generate:

```
                section         .text
func:           ...
```

Constants

Constants are generated by initializations that use the **const** qualifier. For example, the following constant:

```
long int const xyz = 5;
```

would cause the compiler to generate:

```
                section         .const
xyz:            dc.l            5
```

Initialized **static** and **extern** data is placed in the .data section. The initialization:

```
char array[ ] = {1, 2, 3};
```

would cause the compiler to generate:

```
                section                 .data
array:          dc.b                    1,2,3
```

Strings

Strings refer to C strings (anonymous arrays of char) and they are put in the **.strings** section unless they are used to initialize an array (since the string is a shorthand in that case). The string (``xyz'') in the expression:

```
func("xyz");
```

would be placed in the **.strings** section as follows:

```
                section                 .strings
                dc.b                    "xyz",0
```

whereas the same string used to initialize an array as in:

```
char array[ ] = "xyz";
```

would be generated in the **.data** section. This is because the use of the string in an initialization context is a shorthand for:

```
char array[ ] = {'x', 'y', 'z', '\0'};
```

Uninitialized data

Finally, uninitialized data refers to all **static** and **extern** data that is not initialized. For example, the array:

```
char array[100];
```

would generate:

```
                section                 .bss,bss
array:          ds.b                    100
```

Notice that this is the **ds** directive (define storage) rather than the **dc** (define constant) directive that is used if you are initializing something.

Startup Sections

CODE defines several standard sections that are used by the standard linker command files and the assembly general library to control program initialization. These sections may or may not contain anything. It depends entirely on your runtime environment configuration. There is no penalty in either execution speed or program size to have a section defined that has nothing in it.

The standard sections used by CODE are:

.start0
: Contains only the label *__start*. There is no program code in this section.

.start1
: Contains code that has to be executed immediately after reset. This section is reserved for code that must be executed within a specified number of cycles after reset. The symbol *__fastend* is defined by the standard linker command files to be the end of this section. This symbol can be used, for example when simulating using the debugger.

.start2
: Code that must be executed before memory is accessed, for example: code to set up chip select lines.

.start3
: Code to do fundamental system initialization such as zeroing the **.bss** section and copying initialized, writeable data from ROM to RAM.

.start4
: Code to initialize the stack pointer. After this section is executed programs may call subroutines.

.start5
: Code to initialize basic higher level system funtions, e.g. the raw memory manager that need to be done before later higher level system functions are initialized and before interrupts are enabled.

.start6
: Code to initialize higher level system functions, e.g. the fixed size memory manager, static threads, etc. that need to be done before interrupts are enabled.

.start7
: Code to enable interrupts.

.start8
: Thread initialization. Code that needs to be executed before a thread is started. Global constructors.

.start9
: Code to be executed immediately before the program at the entry point is executed, for example, to initialize the frame pointer or, for the 68HC16, to initialize the K fields with the value of the *near bank*.

.startX
: Code to enter the program at the entry point. This is the only program code that actually exists in the source file *nstart.s*.

.startZ
: Code to be executed when the program terminates. The section defines the symbol *__exit*.

These sections are meant to be executed in sequence and are therefore defined in sequential order in the standard linker command files. The code in these sections does not return, it just "falls off the end". This means that the various **.start** sections are executed in order before the program is started. The .start sections are designed to provide important *sequence points* during program start up. These sequence points allow you to arrange for code to be executed before a program starts at well defined times during program start up.

The sections .**start0**, **.start3**, **.start4**, .**start7**, **.start9**, **.startX**, and **.startZ** should not in general have code placed in them by user programs. You can change the functionality of these sections, but you should change them in the appropriate **genXX** library.

The sections **.start2**, **.start5**, **.start6**, and **.start8** are meant to be places where you can add code to the start up process. Section **.start2** should be reserved for time critical initialization and be kept rather small. Sections **.start5** and **.start6** can be used rather arbitrarily for things that need to be done before interrupts are enabled with the knowledge that code in **.start5** is executed before code in **.start6**. Section **.start9** can contain arbitrary code that needs to execute after interrupts have been enabled but before program startup.

The important thing to remember about these start up sections is that *within any given section* there is no way to insure that the code from *filea* will execute before or after code in *fileb*. In other words, code that is targeted for one of the start up sections should have no order of execution relationship with other code targeted for the same section. For example, the library code in **.start4** initializes the processor's stack pointer so subroutine calls can be made. If you add code to the section **.start4**, you have no guarantee that your code will be placed by the linker after the libraries code to initialize the stack. This may have disastrous results. That is why so many start up sections exist. You should pick the appropriate section out of **.start2**, **.start5**, **.start6**, and **.start8** to place any code that you need executed at program start up unless you have a compelling reason to use one of the other start up sections.

At the end of this lengthy discussion of start up sections it seems appropriate to point out a couple of important points: first, although there are quite a few start up sections defined in general only a few of them may be used by any given program; and second: the definition of a start up section in the linker command file by itself imposes no space or execution penalty on the final program unless there is code actually placed in the section. Start up sections are defined to allow the code that is executed when the program is started to be controlled at link time rather than to force you to edit and re−assemble the start up code.

Building Standard Libraries

There are two object libraries that are linked with your object files when you create a program. The C library, named **libc.a*XX*** and the assembly support library, named **libgen.a*XX***. The *XX* in both of these library names is the two digit number that specifies which processor the library is used with. The object libraries are kept in the **lib** directory.

There are several CODE project files that you can use to build or re−build the libraries. You can find these project files in the **Libraries** directory and its sub directories. The **Libraries** directory contains a project file called **libAll.code** which can be used to build the libraries for all the processors. We use **libcAll.code** at Introl to build the libraries for distribution. **libAll.code** is a project that builds two subprojects: **libcAll.code** in the **C** directory; and **libgenAll.code** in the **Assembly** directory. You can build all the **C** or **gen** libraries separately by building those projects, respectively. You can build the **C** and **gen** libraries for a particular processor separately using the individual project files in the **C** and **Assembly** directories called **libc*XX*.code** and **libgen*XX*.code**.

For example, to build just the libraries for the 68HC11, build the project **libc11.code** in the **C** directory and **libgen11.code** in the **Assembly** directory.

All of the individual library building projects create object directories called **obj*XX*** and place the individual object files for the library there. When a library is successfully built, the post build script in each library building project copies the library to the **lib** directory so it is ready for use.

You can change the following pre−processor definitions in **introl.h** to affect how a C library is compiled.

FLOATS
: Defined in ifmt.c and ofmt.c. When defined, allows the scanf() and printf() families of functions to input and output floating point. If your program does not use floating point, you could recompile the two files with FLOATS undefined for considerable code size savings.

LONGS
: Defined in ifmt.c and ofmt.c. When defined, allows the scanf and printf families of functions to input and output long integers. If your program does not use long integers, you could recompile the two files with LONGS undefined for considerable code size savings.

You also may find that the conveniences in the ANSI−C library give too much overhead to your embedded application. Some methods of controlling the size of the library are:

- alleviating overhead by not using any of the standard I/O functions and directly calling your device I/O functions. Also enter you program at **main** rather than the stdio initializing **__main** default entry point, see The Runtime Environment.
- undefining FLOATS and LONGS in the formatted I/O support functions;
- using the function version of the character type macros (ischar, isdigit, isctrl, etc.), which are slower but smaller because they do not need the 257 byte _ _ctype look−up table.

Building the libraries using the supplied Makefiles

We have included UNIX compatible makefiles to help rebuild either the ANSI−C library or the support library. In addition to standard Introl software (supplied with this package) they only use a few external commands such as **echo** and **mv**.

Working with libraries with iar

If you decide that you do not need to rebuild an entire library, you can rebuild the necessary files and use the **iar** program to manipulate files contained in archives and libraries. For example, you can add, delete, update, or create new libraries with **iar**.

Filenames inside an archive have a maximum file name length of 14 characters. If a path to an archive member is given on the **iar** command line, it is stripped, and the file name only is used.

For the duration of this sub−section, the manual will reference a library named *libc.a68*. This file can be found in the **lib** sub directory of the directory in which you installed the Introl software. We recommend that you make a copy of this file in a scratch directory before trying the following examples.

To see the members contained in an archive the following command could be used:

```
iar −t libc.a68
```

To extract all files contained in an archive into the current directory, use the following command:

```
iar −x libc.a68
```

The syntax for extracting a single source file from the library is similar to that of listing certain members from an archive. For example:

```
iar −x libc.a68 putc.o68
```

will extract a copy of the file *putc.o68* into your current directory.

To add files to an existing library, specify the names of the files that you wish to add (If *file0.o68* and *file1.o68* existed):

```
iar −a libc.a68 file0.o68 file1.o68
```

The **iar** program can replace files in a library by using its replace option and specifying the list of files that you wish to replace. This example:

```
iar −r libc.a68 putc.o68
```

will replace the contents of *putc.o68* in *libc.a68* with the contents of the *putc.o68* which resides in the current directory.

iar can also be told to only replace a file in an existing library if it has been modified more recently than the file under the same name in the library. This is an additional flag that can be given to the replace option for iar:

```
iar −ru libc.a68 putc.o68
```

Deleting a file from a library archive follows the same general format as the above commands. For example:

```
iar −d libc.a68 putc.o68
```

will delete the file *putc.o68* from the library *libc.a68*.

The iar program can also create a new library by using its create option, specifying the name of the new library, and all the files which are to be placed in it. For example:

```
iar −r name.a file.o
```

will create a library called *name.a* and place *file.o* in it as its first component.

The CREX Real−time Executive

CREX is a small, extensible *real−time executive* that comes with CODE. CREX has been ported to the 68HC11, 68HC12, 68HC16, and 68332 microcontrollers as well as the 6809 microprocessor. The source for CREX is included with CODE. You can find it in the assembly library section. CODE also includes a small CREX demonstration program that you can build and run.

A real−time executive manages *resources* on a system. System resources that can be managed by an executive include the CPU and memory. This section describes the operation of CREX as it applies to all supported processors. There are also CREX documents for each processor that describe processor specific aspects of CREX, including assembly language calling conventions, memory models, etc. The C library documentation describes the data structures and functions that can be used by a C program to access CREX facilities.

CREX was designed to provide adequate functionality on very small target systems while also allowing its functionality to be extended cleanly and easily. In addition to minimal memory requirements, CREX has been designed to minimize *interrupt latency*: the maximum time that interrupts are disabled.

CREX is made up of a *core*, or minimal set of functions, and optional add−on *modules* for extending CREX's functionality. Modules can be added to CREX to manage memory, queues, timers, etc. Some of the modules can be used independently of CREX, while other modules require the CREX core and other modules to function. The following modules are described here:

The CREX core
: The basic CREX functions.

The dynamic memory management module
: Allocation and freeing of dynamic memory areas.

The clock module
: Periodic events and time slicing.

The timeout module
: Millisecond and second timekeeping functions.

The queuing modules
: LIFO and FIFO queuing of bytes and arbitrary sized objects.

The CREX core

The CREX core manages the CPU resource of a system. The core allows you to define multiple *threads* of execution and control how those threads interact with one another and vie for control of the CPU. In CREX each thread has its own stack and a *priority*. The priority of a thread is used to determine which of possibly several threads should be executed at any given time. Once started, a thread will run until it waits for an *event*, attempts to lock a *resource lock*, is *interrupted*, or the thread terminates.

The priority of a thread can be a value between 0 and 255, with 255 being the highest priority. CREX defines a thread of priority 0, the *idle thread*, that will execute whenever no other threads are ready to execute. The idle thread can be user defined. It must never terminate and should make no CREX call that might cause it to become locked or forced to wait for an event.

An event is an arbitrary value that can be waited on. Event values are usually memory addresses. A thread can *send* an event. This wakes up any threads that might be waiting for that event.

A resource lock can be used to control access to a system resource. When a thread attempts to lock a lock and the lock has already been locked, the thread will be suspended until the lock becomes unlocked.

A thread can be interrupted by an asynchronous interrupt. When this happens, the *interrupt handler* determines how system execution should proceed. An interrupt handler can do whatever it takes to remove the source of the interrupt, process it, and return. The interrupted thread will continue executing in this case. Normally under CREX an interrupt handler will start an *interrupt thread* to process an interrupt. An interrupt thread is a normal CREX thread and can use CREX facilities to communicate with other CREX threads.

A thread terminates by returning or explicitly calling the CREX exit function.

There are several functions in the CREX core that are used for thread creation and control. They are **cx_init**, **cx_start**, **cx_suspend**, **cx_slice**, **cx_wait**, **cx_event**, **cx_lock_w**, **cx_lock_nw**, **cx_unlock**, **cx_unlock_wait**, **cx_priority**, and **cx_exit**. In addition an interrupt handler can use **_cx_interrupt_thread** to start an interrupt thread. These functions will be described in the following sections. These sections describe the CREX functions in language independent terms: for specific information on how to use this functions see the C or Assembly library sections.

cx_init and cx_start

A thread is created under CREX by using a two step process. First, a thread object is initialized. Then the thread can be started at any time. **cx_init** is used to initialize a thread from a *thread init block*. A thread init block contains the following information:

- A pointer to the thread object to be initialized.
- The address of the thread entry point.
- The address that the thread returns to (the thread's exit function).
- The address of the thread's stack.
- The size of the stack in bytes.
- The initial thread priority.

cx_init does not return anything. Its arguments are:

- An value that is passed to the thread when it is started.
- The address of the thread init block.
- An address that is passed to the thread's exit function when the thread terminates.

cx_start can be used to start a thread that has been newly initialized or it can be used to restart a thread that was suspended, see **cx_suspend** below. When a thread is started, it CREX will schedule it, along with other running threads, based on its priority. **cx_start** does not return anything and has a single argument:

- The pointer to the thread object as initialized by **cx_init**.

cx_suspend

cx_suspend will *suspend* a thread. A suspended thread will not execute again until it is restarted with **cx_start**. cx_suspend does not return anything and has a single argument:

- The pointer to the thread object or **NULL** for the current thread.

A thread may suspend itself, in which case the cx_suspend call will not return until some other thread starts the suspended thread again.

cx_slice

cx_slice will cause the next thread to be executed to be placed after other threads of equal priority in the running thread queue. This function is used by the clock module to implement time sliced scheduling of equal priority tasks. cx_slice does not return anything and takes no arguments.

cx_wait and cx_event

cx_wait and **cx_event** are used to cause threads to wait for and send events. An event is an arbitrary value that is usually a memory address associated with the event. For example, a thread that is waiting for something to happen with an I/O device might wait on that device's address. When the device interrupts, the device's interrupt thread could then wake up the sleeping thread by signaling the device's address as an event.

There can be more than one thread waiting for an event. If the event in signaled that all waiting threads are restarted.

cx_wait returns a value determined by the cx_event call that signals the event and takes one argument:

- The value (typically an address) of the event.

cx_event does not return a value and takes two arguments:

- a value to be returned to all threads awakened by the event.
- The value of the event.

cx_lock_w, cx_lock_nw, cx_unlock and cx_unlock_wait

These functions are used to manipulate resource locks. **cx_lock_w** and **cx_lock_nw** attempt to lock a resource lock. **cx_lock_w** causes the calling thread to block if the lock is already locked. **cx_lock_nw** returns immediately with value indicating whether or not the lock was locked. Both functions take a single argument:

- The address of the resource lock.

cx_unlock unlocks a resource lock and wakes up the highest priority thread that is waiting on the lock. **cx_unlock** does not return anything and takes a single argument:

- The address of the resource lock.

cx_unlock_wait unlocks a lock like **cx_unlock** but then immediately causes the calling program to wait for the event represented by the address of the resource lock. **cx_unlock_wait** returns the same value as **cx_wait** and takes the same argument as **cx_unlock**.

cx_priority

This function changes the priority of the current thread. **cx_priority** does not return anything and takes a single argument:

- The new priority value between 0 and 255.

cx_exit

This function terminates the current thread.

_cx_interrupt_thread

An interrupt handler in a program using CREX is written in assembly language and carefully crafted to keep interrupts disabled as little as possible. An interrupt handler that is to communicate with a CREX thread will do so by starting an interrupt thread with **_cx_interrupt_thread**. **_cx_interrupt_thread** never returns and takes two arguments:

- An arbitrary value that is passed to the interrupt thread.
- The address of a thread object that has been initialized by **cx_init**.

The interrupt thread object should initialized with its thread return address as **_cx_interrupt_exit**. **_cx_interrupt_exit** reinitializes the interrupt thread object, making it ready to be used for the next interrupt.

The general scheme used in interrupt handlers and interrupt threads is that the interrupt thread object is initialized with **cx_init** sometime before the interrupt is enabled. When the interrupt occurs the interrupt handler:

- Saves the current processor context, if necessary. This is required on the 68HC16 and 68332/68K and is done automatically by the processor on the other processors.
- Service the interrupt and remove the interrupt source, if necessary.
- Jump to **_cx_interrupt_thread**.

For example, the interrupt handler for a serial port received character interrupt would read the data from the port, which would remove the interrupt source, and jump to **_cx_interrupt_thread** with the data as its argument.

The interrupt thread can then use normal CREX calls to handle the interrupt. When the thread returns, **_cx_interrupt_exit** is entered with the thread initialization block as its argument. An interrupt thread can be a normal C function that takes one argument and simply returns when it finishes.

Although interrupt latency is very low in CREX, it is possible for an interrupt to occur while its interrupt thread is still active. If this happens and **_cx_interrupt_thread** is called with an active thread object, the interrupt is *lost*, or simply ignored. In the serial port example, this would mean that received characters would be missed. You can handle fast interrupts in an interrupt handler by having multiple interrupt threads that can be run. The interrupt handler can then choose an available thread to process the interrupt.

There is an example of a serial port interrupt handler and interrupt thread called **crexio.s** in the assembly library for the supported processors.

The dynamic memory management module

The dynamic memory management module are the lowest level facilities available in CREX to manage *dynamic* memory, that is, memory that can be allocated and deallocated during the course of an embedded program's execution. The dynamic memory management module should be used carefully in an embedded system. The dynamic memory management allows the allocation and deallocation of arbitrary sized memory regions. Over time this may cause *fragmentation* of the available free memory. Fragmentation occurs when the available free memory, although large enough to satisfy a memory allocation request, does not have a single free area large enough to meet the request. The memory management algorithms try to minimize fragmentation but a program can allocate and deallocate memory in such a way as to cause fragmentation that cannot be eliminated by the manager.

The dynamic memory management module is safe to use when you keep the memory fragmentation problem in mind. In general, CREX modules use the dynamic memory manager to create long−lived, if not

permanent, regions of memory that are dynamic only in the sense that they are allocated only when the system is initialized. To help insure safety, and to reduce the overhead of the dynamic memory manager, there is a version of it in the library that supports only memory allocation, not deallocation.

The dynamic memory manager maintains free memory regions in a *free list*. The free list is protected by a CREX lock so that multiple threads can make requests to the memory manager in an organized way.

There are three functions in the dynamic memory manager: **cx_addmemory**, **cx_alloc**, and **cx_free**.

cx_addmemory is used, generally at system initialization, to add a block of memory to the manager's free list. **cx_addmemory** does not return anything and has two arguments:

- The size of the memory block in bytes.
- The starting address of the memory block.

cx_memory is called implicitly when the dynamic memory manager is added to a CREX based program to add the **.heap** section to the manager's free list.

cx_alloc returns the address of an allocated block of memory and has one argument:

- The size of the desired memory region.

cx_free frees a region of memory previously allocated by **cx_alloc**. **cx_free** does not return anything and has one argument:

- The address of a memory region previously returned by **cx_alloc**.

The clock module

The CREX clock module is used to generate periodic events and to *time slice* threads. The basic unit of time measurement is a system *tick*. A tick is a re−occurring fixed period interrupt. The clock module calls a function called **_cx_millitimer** to convert a tick into milliseconds. Milliseconds can be accurately calculated for a tick or, as is the case with the default **_cx_millitimer** function, a tick may represent an arbitrary number of artificial ``milliseconds'' upon which all other system timing is based. The default **_cx_millitimer** function calls each clock tick one millisecond. This is only accurate for wall clock timing if the periodic interrupt happens to occur at every millisecond. Either you can supply or CODE can create a target system specific version of **_cx_millitimer** that can calculate the number of milliseconds in a clock tick based on the number of cycles between periodic interrupts and the number of cycles that constitute a second.

No mater what the basis of a millisecond is, a CREX second is 1000 milliseconds.

Depending on the processor speed and periodic interrupt rate, the *resolution* of the systems timing functions may vary: faster period interrupts allow single millisecond timings while slower interrupts allow 10 or 100 millisecond resolution. All CREX timing functions are based on either milliseconds or seconds.

Time slicing occurs when equal priority threads are competing for the CPU at the same time. You can specify at link time the number of milliseconds that a thread should run before it is preempted by another thread of equal priority. The preempted thread is placed after other threads of equal priority in the running queue, thus implementing a scheduling scheme called *round robin* for equal priority threads.

The clock module also maintains two counters, **_cx_ms** and **_cx_clock**. An event is sent on **_cx_ms** periodically with the number of milliseconds that have passed since the last **_cx_ms** event. An event is sent on **_cx_clock** every second or 1000 milliseconds.

There are two functions associated with the clock module: **cx_gettime** and **cx_settime**. These functions are used to read and set the system clock.

cx_gettime takes no arguments and returns the current value of the **_cx_clock** and **_cx_ms** counters. The reading of the counters is synchronized so the value returned represents on instant of system time. The **_cx_clock** value is between 0 and 4,294,967,295. If no calls to **cx_settime** were made, this value represents the number of seconds that have occurred since the system started. The **_cx_ms** value is between 0 and 999: the number of milliseconds that have occurred in the current second.

cx_settime does not return anything and takes two arguments:

- The new value to place in **_cx_clock**.
- The new value to place in **_cx_ms**.

By convention, the **_cx_clock** value is set to a number of seconds since a fixed an initial *epoch*, or time zero. The ANSI−C library functions assume that the **_cx_clock** value represents the number of seconds since 00:00:00, January 1st, 1970, Coordinated Universal Time.

The timeout module

The timeout module is used to generate events at a future time. Time−outs can be scheduled in milliseconds or seconds and can occur once or automatically repeat indefinitely. A small number of time−outs are allocated at system startup time. The number of time−outs allocated at a time is controlled by the linker variable **CX_TIMEOUTS**. If a timeout is requested when the available time−outs are all being used then the timeout module will allocate more. You can eliminate allocation of time−outs after system initialization by making **CX_TIMEOUTS** large enough to take care of all the time−outs that a system will have active at any time.

The timeout module functions are **cx_TOms**, **cx_TOms_periodic**, **cx_TOsec**, **cx_TOsec_periodic**, and **cx_TOend**. There are two internal functions, **_cx_TOmilliseconds** and **_cx_TOseconds**, that are called by the clock module to update the millisecond and second time−outs respectively.

cx_TOms, **cx_TOms_periodic**, **cx_TOsec**, and **cx_TOsec_periodic** are used to start a timeout. These functions return the address of the timeout that can be used in a subsequent **cx_TOend** call. Their arguments are:

- The number of time units to count, in milliseconds or seconds.
- The event address to send when the timeout expires.
- The argument to send to the event.

The functions ending in **periodic** set up time−outs that will repeat indefinitely.

The function **cx_TOend** must be called when a timeout is no longer needed, either because it has expired or because it should be canceled. **cx_TOend** returns an error if the specified timeout is not active or completed and takes one argument:

- The address of the timeout as returned by one of the timeout starting functions.

The functions **_cx_TOmilliseconds** and **_cx_TOseconds** are called by the clock module with one argument:

- The number of milliseconds (usually one or a small number) or the number of seconds (always one) that have occurred since the last call.

A user program should never have to call these functions.

The queuing modules

The CREX queuing modules are used to define data *queues* that are used to synchronize threads that need to pass data between themselves. Queues are protected by locks to insure that only one thread accesses them at any given time. Data may be *inserted* in a queue by a thread. When the queue is full the inserting thread either waits for the queue to have more room or the queue insertion functions will return an error indicator, depending on how the queue is configured. Data may be extracted from a queue by *removing* or *unstacking* it. Data that is removed from a queue comes out in FIFO (First In, First Out) order. Data that is unstacked from a queue comes out in LIFO (Last In, First Out) order. Otherwise these operations are identical. Depending on how the a queue is configured a thread will either wait for data in an empty queue or receive an error indication from the queue if no data is available.

N.B. The queue extracting functions return zero, along with an error flag, if the queue is empty. By scheduling an event on the queue address with a non zero argument, a thread that is waiting for input from a queue can time out after a period of time. (This will be formalized in a later CREX module).

There are two queue modules currently implemented: queues with arbitrary sized data elements and queues with byte sized data elements. Byte queues work just like arbitrary queues but have been optimized for single byte queuing − a common size when handling characters, for example.

The arbitrary sized queue functions are **cx_newQ**, **cx_Qflags**, **cx_deleteQ**, **cx_insertQ**, **cx_removeQ**, and **cx_unstackQ**.

A queue is created with **cx_newQ**, which returns the address of the initialized queue. Its arguments are:

- The maximum number of data elements in the queue.
- The size of each data element.

By default a created queue will return an error indicator if an attempt is made to insert data into a full queue or remove data from an empty one. The **cx_Qflags** function is used to modify this default behavior. The flags that can be set on a queue are:

- **CXQ_WAITIN** − wait on insert until there is room in the queue.
- **CXQ_WAITOUT** − wait on remove or unstack until data is available.

The **cx_Qflags** function does not return a value and has two arguments:

- The bitwise−or of the flags.
- The address of the queue object as returned by **cx_newQ**.

The **cx_insertQ** function inserts data into a queue. When the queue is full it either waits or returns an error, as described above, and takes two arguments:

- The address of the queue object.
- The address of the data element.

The **cx_removeQ** and **cx_unstackQ** functions extract data from the queue in FIFO and LIFO order respectively. When the queue is empty that either wait or return an error, as described above, and take two arguments:

- The address of the queue object.
- The address to put the queue data element.

The byte sized queue functions are **cx_newBQ**, **cx_BQflags**, **cx_deleteBQ**, **cx_insertBQ**, **cx_removeBQ**, and **cx_unstackBQ**.

A byte queue is created with **cx_newBQ**, which returns the address of the initialized queue. Its arguments are:

- The maximum number of data elements in the queue.

By default a created queue will return an error indicator if an attempt is made to insert data into a full queue or remove data from an empty one. The **cx_BQflags** function is used to modify this default behavior. The flags that can be set on a queue are:

- **CXQ_WAITIN** − wait on insert until there is room in the queue.
- **CXQ_WAITOUT** − wait on remove or unstack until data is available.

The **cx_BQflags** function does not return a value and has two arguments:

- The bitwise−or of the flags.
- The address of the byte queue object as returned by **cx_newBQ**.

The **cx_insertBQ** function inserts data into a queue. When the queue is full it either waits or returns an error, as described above, and takes two arguments:

- The byte to insert.
- The address of the queue object.

The **cx_removeBQ** and **cx_unstackBQ** functions extract data from the queue in FIFO and LIFO order respectively. When the queue is empty that either wait or return an error, as described above, or returns the extracted byte. They take one argument:

- The address of the queue object.

Assembler Only Projects

Device Definition Files

Moving Functions to RAM

Sometimes it is desirable to move a C function to RAM to execute it. This is useful to call a function in RAM that can be used to burn flash ROM.

A very easy solution to this is to create the flash burning function in a separate source file and then right click on the source file in the directory window and select **Options**. Choose the **Sections** tab in the options window and change the Code section from **.text** to **.data**. For the GUI impaired, this adds an **−at=.data** option to the compiler command line.

The **.data** section is placed in ROM but linked as if it was in RAM. The startup code copies the ROM into RAM when the program starts, so your flash burning function will be copied along with any other initialized writeable data you might have.

You can call the burning function in the normal way, since the linker has linked the program as if the RAM initialization has already happened.

One caveat with this technique: don't declare any static initialized variables inside a C function that is being placed in the **.data** section because the variables will end up inside of your function's executable code.

There is a slightly more complicated example project that defines two functions that are copied manually into the same RAM area in the **demo/flash** directory.

Demonstration Programs

This part of the manual describes several example projects supplied with CODE. These examples have project (**.code**) files created for them. You can select **Directory** from the **Window** menu and open the project files by double clicking on them.

Ctour
: A simple C program with various data types.

BASIC11 Example
: An example of a multi−file assembly language project.

BUFFALO Example
: An example of a 68HC11 program that runs under the BUFFALO monitor.

CREX Example
: An example using CREX (Controller Real−time EXecutive).

IO Example
: An example using parallel port access and interrupt handling functions.

IMI Example
: Building and using the Introl Monitor Interface (IMI).

MCX11 Example
: An example using the MCX11 executive for the 68HC11.

STEROID Example
: An example of a 68HC16 program that runs on the Intec Innovations Steroid Stamp.

Executing RAM Functions
: An example copying functions to RAM before executing them.

Ibuild Example
: An example using **ibuild** to make a multiple source file program.

Ctour

This example shows how to use the debugger to view and change various data types. This example is meant to built and executed from CODE. Use the CODE Directory window to enter the Ctour directory in the demo directory. Double click on the the project file **ctour.code**. To build and run the demo, press the **Execute** button.

main.c
: The main program.

BASIC11

This example shows how a multi−file assembly language project can be set up. BASIC11 was written by Gordon Doughman, who has generously allowed us to provide it as a demonstration project. BASIC11 is provided AS−IS, with no support available either from Mr. Doughman or Motorola.

This example is meant to built and executed from CODE. Use the CODE Directory window to enter the BASIC11 directory in the demo directory. Double click on the the file **basic.code**. To build and run the demo, press the **Execute** button. The BASIC11 signon message and command prompt will be displayed.

BASIC11 has been modified slightly for this demo, see the project notes for more information. One significant aspect of this demo is that BASIC11's memory usage is controlled entirely from the **Memory Map** window: you can access the **Memory Map** window from the **Project** window under the **Target** tab.

basic11.doc
: The documentation for BASIC11.

defines.lib
basiclb1.asm
basiclb2.asm
basiclb3.asm
basiclb4.asm
basiclb5.asm
command1.asm
command2.asm
inits.asm
iopkg.asm
leditor.asm
rexpres.asm
runtime1.asm
runtime2.asm
runtime3.asm
variables.asm
vectors.asm
: The source for BASIC11.

BUFFALO

This example shows how to build a simple program to run under the BUFFALO monitor on an Motorola EVB card for the 68HC11. This example is meant to built and executed from CODE. Use the CODE Directory window to enter the BUFFALO directory in the demo directory. Double click on the the project file **prog.code**.

main.c
: The main program.

CREX

This example shows how to use CREX (the Controller Realtime EXecutive) to do interrupt driven I/O and have multiple threads of execution. This example is meant to built and executed from CODE. Use the CODE Directory window to enter the CREX directory in the demo directory. Double click on one of the .code files. To build the demo, press the **Build** button.

Now go to the Debugger window by selecting **Debugger** from the **Window** menu or pressing **F5**.

You should now see the beginning of the startup code for the program. From the view select buttons on the right side of the Debugger window,select **Stdio** to view the Standard Input/Output view. Then press **Go**. The program that is running has two threads of execution that use a lock to control access to the C library stdio routines. Press **Stop** to stop the program. You can press **Go** to continue or check out some of the other debugger features.

main.c
: The main program.

IO

This example shows how to use parallel port I/O and an interrupt handling function. This example is meant to built and executed from CODE. Use the CODE Directory window to enter the IO directory in the demo directory. Double click on the the project file **io.code**. To build and run the demo, press the **Execute** button.

main.c
: The main program.

IMI

This example shows how to use parallel port I/O and an interrupt handling function. This example is meant to built and executed from CODE. Use the CODE Directory window to enter the IMI directory in the demo directory. Double click on one of the project files **imi*XX*.code**. This will create the IMI montor for the specific target.

The project files **prog*XX*.code** can be used to build a program to run under IMI.

main.c
: The main program.

MCX11

This example shows how to use the MCX11 executive with CODE. This example works only with the 68HC11. To build and run the demo, select **Open Project** from the **Project** menu. Open the file **mcx11.code** in the MCX11 directory under the demo directory. Or, you can use the CODE Directory window to enter the MCX11 directory and double click on the project file **mcx11.code**. Press the **Build** button to build the project.

Now go to the Debugger window by selecting **Debugger** from the **Window** menu or pressing **F5**.

You should now see the beginning of the startup code for the program. Press **Go**. The program supplied, **test.c**, sets up several tasks and can be simulated, but it does not do anything interesting.

test.c
: The main program.

mcxconf.c
: An example configuration.

manual.doc
: The original MCX11 documentation.

mcx.h, *mcxtypes.h*, *message.h*, *queue.h*, *sema4.h*, *tcb.h*, *timer.h*
: C header files for MCX11.

mcx.s11, *mcxif.s11*, *mcx.lib*, *mcxesr.lib*, *message.lib*, *queue.lib*, *tcb.lib*, *timer.lib*
: The source of the MCX11 executive.

scidrv.s11
: An SCI driver that is not used in the demo.

STEROID

This example shows how to build a simple program to run on the Intec Automation Steroid Stamp. This example is meant to built and executed from CODE. Use the CODE Directory window to enter the STERIOD directory in the demo directory. Double click on the the project file **steroid.code**.

main.c
: The main program.

Executing RAM Functions

This example shows how to copy functions to RAM before executing them. This example is meant to built and executed from CODE. Use the CODE Directory window to enter the flash directory in the demo directory. Double click on the the project file **flash.code**. To build and run the demo, press the **Execute** button.

main.c
: The main program.

flashburn0.c
: The first RAM function.

flashburn1.c
: The second RAM function.

Ibuild

These files are a simple example of the use of **ibuild** to make a multiple source file program.

Makefile
: The **ibuild** description file for the program.

main.c
: The main program.

func1.c
: A simple additional source file.

func2.c
: Another simple additional source file.

asfunc1.s11
: *A simple assembly file that is also linked in.*

funcs.h
: A header file defining the functions in the program.

Copyright