Getting Started

With the Raisonance 8051, XA and ST6 Development Kits

Revision 1.00

Local Distributor

Written by Andrew Ayre

Contents

Tables

Chapter 1. Introduction

The Raisonance 8051, XA and ST6 Development Kits are a complete solution to creating software for the 8051 family, XA family and ST6 family of microcontrollers. The Development Kits comprise many different tools that allow projects ranging from simple to highly complex to be developed with relative ease.

Raisonance has been developing embedded tools since 1988 and has built up many years of experience. You will find that with the Raisonance Development Kits you can rely on tools that have been tested by real users over a long period of time.

This manual has been written to introduce the first time user to the Raisonance Development Kits and guide them through many of the features available. With the aid of this manual users should be able to quickly understand the tools, how they interact with each other and how to start developing their own projects. This manual does not cover the features in great detail or cover advanced features, but provides a familiarity to the tools that will provided a basis for using more complex features.

It is assumed that the user is familiar with Windows and has at least some familiarity with the 8051, XA or ST6 microcontroller family and the C programming language.

This manual is organized into two main parts. The first part takes on a tutorial form, guiding the user through installation and the Windows front-end, demonstrating the main features and ideas. The second part takes on a reference form and provides information about each of the tools in turn.

In some places there are differences between the 8051, XA and ST6 tools and they will be pointed out. To demonstrate the RIDE simulator an 8051 will be simulated, however the simulator is identical in all the Development Kits apart from the aspects specific to the 8051.

Development Tools

The following is a list of the tools included in the Development Kit with a short overview of each one:

Tool	Overview
ANSI C Compiler	The C Compiler is an ANSI compliant compiler that takes source files and generates object files. Extensions to the C language are used to enable features of the microcontroller to be used or controlled.
Assembler	The Assembler takes source files written in assembler and generates object files. Controls are included to enable features of the microcontroller to be used or controlled.
Linker/Locator	The Linker combines the object files generated by the Compiler and Linker and produces a different kind of object file. The Linker also decides where certain types of Data and Code are located in memory.
Object-to-HEX Converter	The converter converts an object file generated by the linker and generates an Intel Hex file, compatible with most device programmers.
RIDE	The Raisonance Integrated Development Environment. RIDE is a Windows program that allows the user to create projects, easily call the Compiler, Assembler and Linker to build the project and either simulate or debug the project.
Library Manager	The Library Manager can take object files generated by the Compiler or Assembler and create a library that is included in other projects.
Monitor	The Monitor is a program that runs on hardware and transmits debugging information back to RIDE as the program executes. It also provides a means of controlling the execution of the program and debugging the program while it is executing on hardware

Table 1.1 Tools Overview

Development Tool Names

Some of the tools have names by which they are often referred to. The table below lists the names for each of the three toolsets.

Tool	8051 Tool Name	XA Tool Name	ST6 Tool Name
ANSI C Compiler	RC-51	RC-XA	RC-ST6
Assembler	MA-51	MA-XA	MA-ST6
Linker/Locator	LX-51	RL-XA	RL-ST6
Object-HEX Converter	OH51XA	OH51XA	OHST6
Library Manager	LIB-51	LIB-XA	LIB-ST6

Table 1.2 Tool Names

RIDE (Raisonance Integrated Development Environment) is common to all three tool chains. Only the installation specifies which Microcontroller family is available.

Conventions Used in this Manual

File | New	Refers to the menu item “New” on the File menu
`while(1);`	(bold, monospaced type) User input
filename	Replace the italicized text with the item it represents.
[ ]	Items inside [and] are optional.

Additional Help or Information

You may find additional documentation in the DOCS folder inside the RIDE installation. In addition help is available via the Help menu in RIDE.

Also you can visit the following web sites:

http://www.raisonance.com
http://www.amrai.com

North and South America:

Address:	American Raisonance Inc., PO Box 1784, Addison, TX 75001-1784 USA
Telephone:	1-877-315-0792
Fax:	1-972-818-0996
Email:	info@amrai.com (General information) sales@amrai.com (Sales) support@amrai.com (Customer Support)

Rest of the World:

Address:	Raisonance S.A. 755, Avenue Ambroise Croizat, 38920 Crolles, France
Telephone:	+33 4 76 08 18 16
Fax:	+33 4 76 08 09 97
Email:	info@raisonance.com

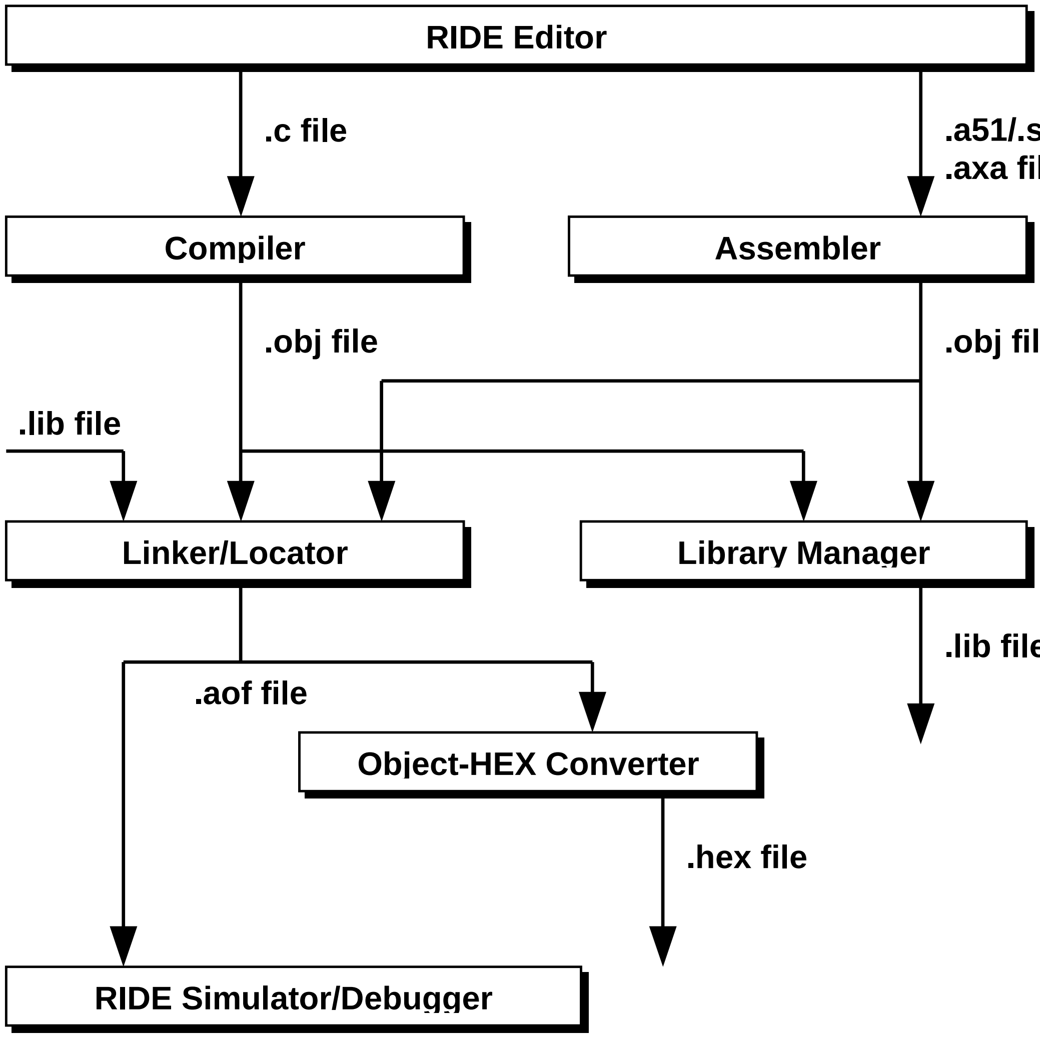
If you find any errors in this manual or omissions from it, or if you have suggestions for improving this manual, please let us know by Emailing:

support@amrai.com

Chapter 2. Development Steps

The Relationship Between the Tools

The following diagram shows the relationship between the tools.



1. RIDE provides an editor which allows the user to generate C source files (.c extension) and Assembler source files (.a51 extension for 8051, .axa extension for XA and .st6 extension for ST6).

2. Each source file is translated using the appropriate tool. The Compiler translates C source files. The Assembler translates assembler source files. Each tool generates a relocatable object file (.obj extension). If a project has more than one C source file or more than one Assembler source file, then the Compiler and Assembler are executed as many times as required.

3. If a library file is being generated then the Library Manager takes all the relocatable object files and combines them into a library file (.lib extension). The library file may then be linked in with other projects.

4. The Linker/Locator takes relocatable object files and library files and links them together resolving external references. The Linker/Locator then locates variables and code to specific addresses in the memory map. The Linker/Locator generates a single Absolute Object File (.aof extension). It also generates the same file with no extension.

5. The Absolute Object File may then be used by the simulator or debugger in RIDE, as the file can contain debugging information. Alternatively the Absolute Object File may be used with In-Circuit Emulators.

6. The Object-HEX Converter tool converts an Absolute Object File into an Intel HEX file (.hex extension) which is a representation of the pure binary code generated, without debugging information. The Intel HEX File is accepted by virtually all device programmers.

In addition to being an editor, simulator and debugger, RIDE also controls and automates the entire build process. By selecting a single menu item, RIDE will execute the correct tools to generate either a library file or an Absolute Object File and Intel HEX File.

NOTE

Each relocatable object file is referred to as a module. Each module must have a unique name. For example the source file foo.c generates the relocatable object file foo.obj. The module name is therefore “foo”. However the source file foo.a51 also generates the relocatable object file foo.obj.

The result is two modules with the same name. Therefore each source file must have a unique name, regardless of whether it is a C source file or an Assembler source file.

Listing Files

Some of the tools generate text files, collectively referred to as Listing Files, in addition to the files shown in the diagram. These listing files aid the user in understanding how the tools processed the input files and in tracking down problems.

The Compiler and Assembler generate a listing file (.lst extension) for each source file they translate. The listing file contains such information as the assembly code generated, a symbol table, the memory requirements of the module and how the tool was invoked.

The Linker generates a listing file commonly referred to as the Map File (.m51 extension for 8051, .mxa extension for XA and .mst extension for ST6). This file will be referred to as the Map File in the rest of this manual. The Map File contains a list of input modules and libraries, a memory map of the project, a summary of the memory requirements, a call tree and a symbol table. Only one Map File is generated as the Linker is only executed once.

Summary of File Extensions

The following table provides a summary of the file extensions used.

File	**8051 File Extension**	**XA File Extension**	**ST6 File Extension**
Project	.prj	.prj	.prj
Compiler Source File	.c	.c	.c
Assembler Source File	.a51	.axa	.ast
Compiler and Assembler Object Files	.obj	.obj	.obj
Compiler and Assembler Listing Files	.lst	.lst	.lst
Linker Object File	.aof	.aof	.aof
Linker Map File	.m51	.mxa	.mst
Intel Hex File	.hex	.hex	.hex
Library File	.lib	.lib	.lib

Table 2.1 Summary of File Extensions

Note

The file extensions used may be modified or additional file extensions supplied. In RIDE choose Options | Tools then select the relevant tool and click on Edit. Fields are provided to enter the input and output file extensions.

Chapter 3. Installing the Software

Minimum System Requirements

- Windows 95/98/NT/2000/Me
- Pentium Processor
- 20Mb Hard Drive Space
- 32Mb RAM

Installing the Software

If the software is being installed from CD then the installation program should automatically run when the CD is inserted.

If the CD autorun feature is turned off or you have downloaded the software from a web site then the software may be installed simply by running INSTALL.EXE.

Directory Structure

The following is the directory structure placed into the installation folder

BIN folder

This folder contains the executable files for the tools and associated library files that are required by the tools.

DOC folder

Contains manuals for each of the tools as well as manuals for various evaluation boards

EXAMPLES folder

Contains example projects for use with RIDE, including Monitor example projects. The examples are subdivided into categories.

HELP folder

Contains the on-line help files used by RIDE.

INC folder (**INCST6** for the ST6)

Contains include files that may be used by users in projects. Some of the include files define Special Function Registers for various derivatives. Include files for the Standard C Libraries may also be found in this folder.
In addition source code for the Monitor (8051 and XA), some of the Standard C Library functions, such as memory allocation (8051 and XA) and low-level I/O and the startup code may be found in the Sources sub-folder.

LIB folder

Contains libraries that the Compiler and Linker/Locator may use in the compiling of a source file or linking of a project. The libraries include such things as routines to perform mathematical operations on floating point numbers, the Standard C Library functions and the Monitor (8051 and XA).

NOTE

The Compiler will look in the INC folder for header files (INCST6 for ST6). If the Special Function Register Header File for the particular device you are using is not present in the INC folder then you can easily create your own and place it in the INC folder.

1. Find the header file of the device that is closest to the device you are using
2. Copy it and rename the copy to the name of the device
3. Using the device datasheet, add in the missing Special Function Register declarations using the same format as shown in the file.
4. Include the header file in your C source files inside “<” and “>”, for example

```
      #include <reg999.h>
```

Chapter 4. Getting Started with RIDE

Overview of RIDE

It is possible to create the source files in a text editor such as Notepad, run the Compiler on each C source file, specifying a list of controls, run the Assembler on each Assembler source file, specifying another list of controls, run either the Library Manager or Linker (again specifying a list of controls) and finally running the Object-HEX Converter to convert the Linker output file to an Intel Hex File. Once that has been completed the Hex File can be downloaded to the target hardware and debugged.

Alternatively RIDE can be used to create source files, automatically compile, link and convert using options set with an easy to use user interface and finally simulate or perform debugging on the hardware with access to C variables and memory.

Unless you have to use the tools on the command line, the choice is clear. RIDE greatly simplifies the process of creating and testing an embedded application.

Projects

The use of RIDE centers on “projects”. What is a project? A project is a list of all the source files required to build a single application, all the tool options which specify exactly how to build the application, and – if required – how the application should be simulated.

A project contains enough information to take a set of source files and generate exactly the binary code required for the application.

Because of the high degree of flexibility required from the tools, there are many options that can be set to configure the tools to operate in a specific manner. It would be tedious to have to set these options up every time the application is being built, therefore they are stored in a project file. Loading the project file into RIDE informs RIDE which source files are required, where they are, and how to configure the tools in the correct way. RIDE can then execute each tool with the correct options.

It is also possible to create new projects in RIDE. Source files are added to the project and the tool options are set as required. The project can then be saved to preserve the settings. The project also stores such things as which windows were left open in the simulator/debugger, so when a project is reloaded and the simulator or debugger started, all the desired windows are opened.

RIDE project files have the extension .prj.

Simulator/Debugger

The simulator/debugger in RIDE can perform a very detailed simulation of a microcontroller along with external signals. It is possible to view the precise execution time of a single assembly instruction, or a single line of C code, all the way up to the entire application, simply by entering the crystal frequency.

A window can be opened for each peripheral on the device, showing the state of the peripheral. This enables quick troubleshooting of mis-configured peripherals.

Breakpoints may be set on either assembly instructions or lines of C code, and execution may be stepped through one instruction or C line at a time.

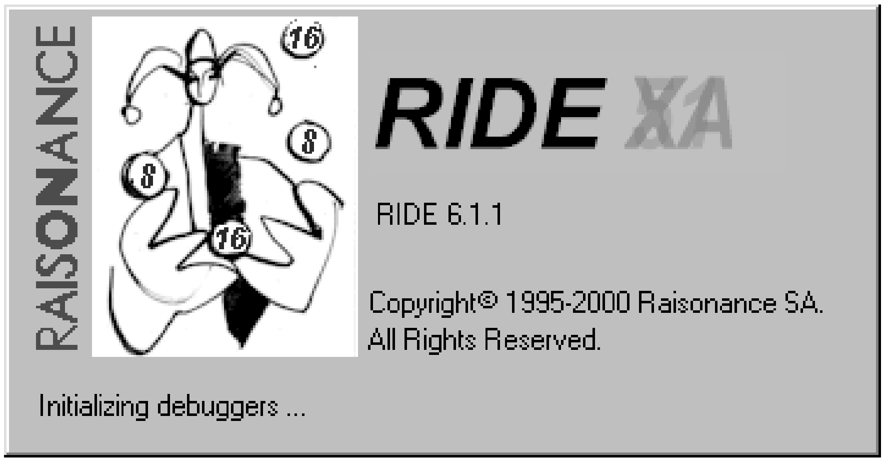
The contents of all the memory areas may be viewed along with the ability to find specific variables. In addition the registers may be viewed allowing a detailed view of what the microcontroller is doing at any point in time.

This chapter will highlight the main features of RIDE and demonstrate how they are used.

Starting RIDE

Starting RIDE is very easy. Select “Ride IDE” from the Start | Programs | Raisonance Kit menu.

You will be presented with the following splash screen:



RIDE 6.1.1

Copyright© 1995-2000 Raisonance SA.
All Rights Reserved.

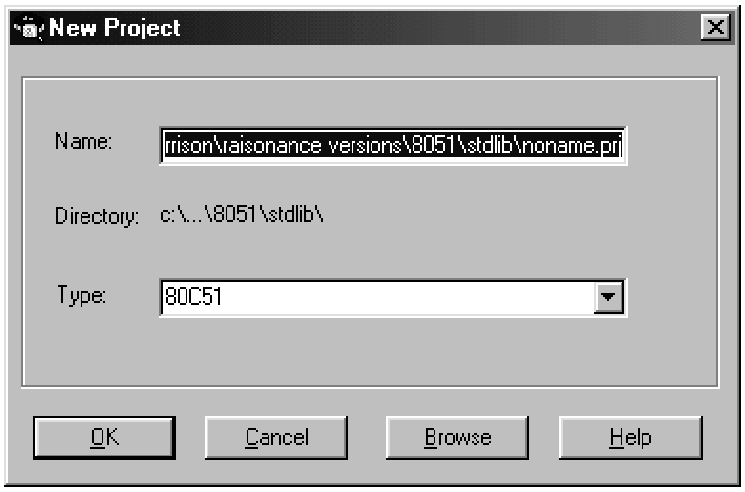
Initializing debuggers ...

After a few moments the main window will open. The main window is described in the next section.

Creating a Project

The first thing we will do is create a new project. We will then take a look at the main RIDE window and become familiar with it.

Choose Project | New and you will be presented with a window which looks like the following:



Name: rrison\raisonance versions\8051\stdlib\noname.prj

Directory: c:\...\8051\stdlib\

Type: 80C51

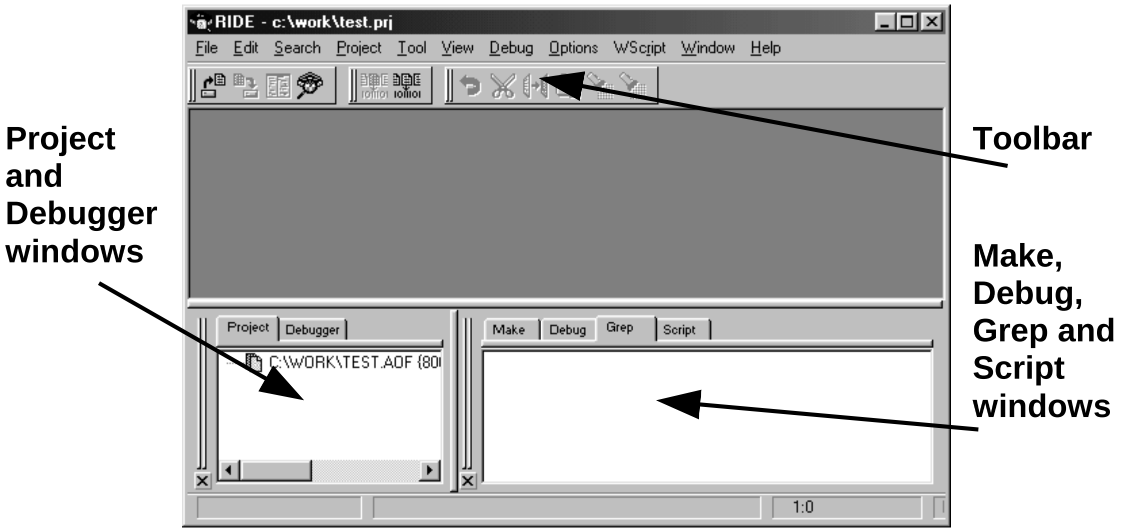
The Name field shows the path to the project file that will be created. The Type field shows the microcontroller family type the project will use. Depending on which development kit you are using the Type field will show either: 80C51, XA or ST6. If you are running the combined 51+XA development kit then you are able to choose between the 80C51 and XA at this point. Remember whether you choose 80C51 or XA as you will need that information later on.

Click on the Browse button and browse to the folder where the project is to be created. This manual will use the location:

```
C:\work
```

In the Filename field the name of the project is entered. Enter **`test.prj`** and click on the Open button then click on the OK button to create the project.

The main RIDE window should now look like the following:


RIDE - c:\work\test.prj
File Edit Search Project Tool View Debug Options WScript Window Help
Project
Debugger
C:\WORK\TEST.AOF {80
Make
Debug
Grep
Script
1:0
Project and Debugger windows
Toolbar
Make, Debug, Grep and Script windows

The Project window acts like a project manager, showing which source files are in the project and giving instant access to each one in both the editor and debugger.

If you look at the Project window you will see one entry with the following pathname:

```
C:\WORK\TEST.AOF
```

This entry represents the project as a whole and the .aof file will be the result of building the project.

Note

The .aof file (Absolute Object File) always takes its name from the name of the project. In this case the project is called “Test” so the .aof file will be called “test.aof”. Likewise the generated Intel Hex File will also be named after the project (“test.hex”).

Creating and Adding a Source File

The next step we will take is to create a new, basic source file and add it to the project. The following section will show how to build the project. We will then know that all the tools are working and you will then have a starting point for all future projects.

To create a new source file choose File | New followed by "C Files" from the pop-up menu that appears. A blank window will open.

Enter the following into the new window:

```
void main(void)
{
  while(1);
}
```

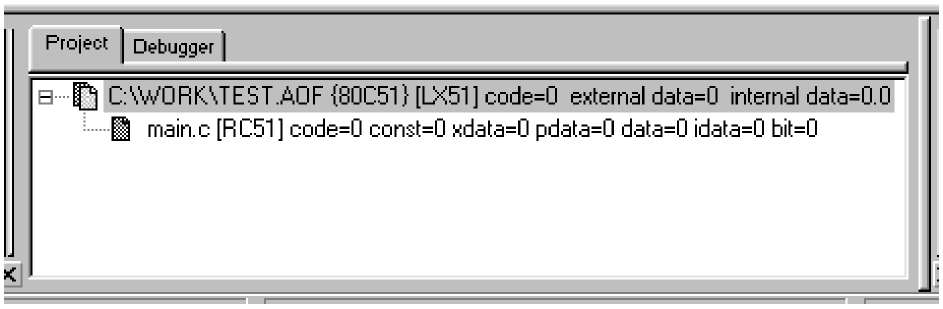
To save the source file:

- Choose File | Save. A standard Save As window will open.
- Enter **`main.c`** into the Filename field.
- Click on Save.

To add the file to the project:

- Select "`C:\WORK\TEST.AOF`" in the project window and press the right mouse button (right-click). A menu will pop up.
- Select Add Node/Source Application from the menu.
- Select main.c in the File Open window.
- Click on Open.

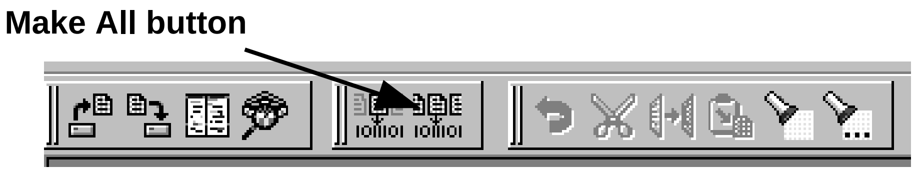
A small "+" sign should appear next to the .aof file in the Project window. Click on the "+" to expand the project tree. The source file should now appear in the Project window under the .aof file:



```
C:\WORK\TEST.AOF {80C51} [LX51] code=0  external data=0  internal data=0.0
    main.c [RC51] code=0 const=0 xdata=0 pdata=0 data=0 idata=0 bit=0
```

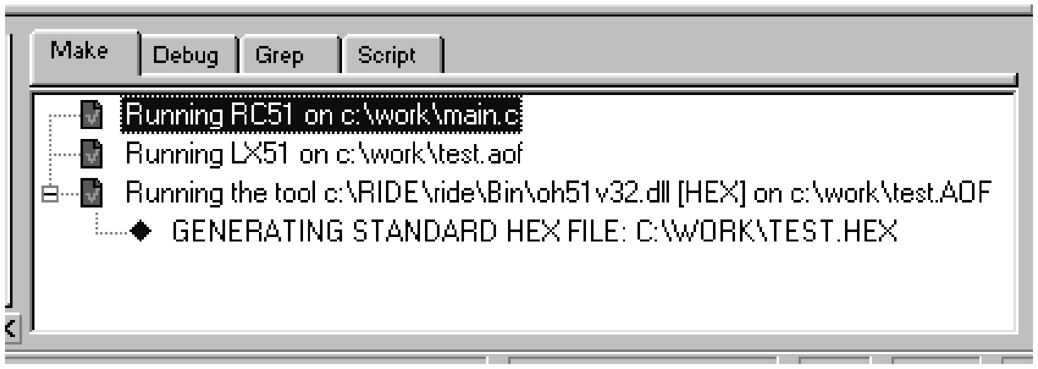
Building the Project

To build the project simply click on the Make All button on the toolbar



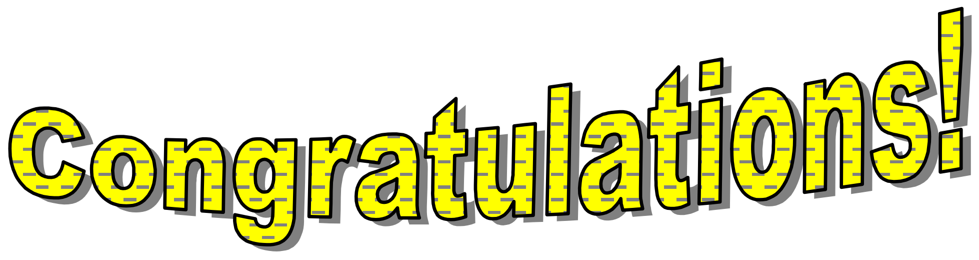
Or choose Project | Make All

Once the project has been built, the Make window will show the result of the build process in tree form:



```
Running RC51 on c:\work\main.c
Running LX51 on c:\work\test.aof
Running the tool c:\RIDE\ride\Bin\oh51v32.dll [HEX] on c:\work\test.AOF
    GENERATING STANDARD HEX FILE: C:\WORK\TEST.HEX
```

Note
If the Compiler or Linker generated any warnings or errors then it would be possible to view them in the Make window. In the case of Compiler warnings or errors, double clicking on them in the Make window will take you to the relevant point in the source code.



You have created and built your first project!

Adding More Code

Before we take a look at the simulator we need to add some more code. Simulating an infinite loop is not very exciting, unless you happen to like infinite loops a great deal.

The Raisonance Compiler features language extensions that allow aspects specific to microcontrollers to be used in C. One of those language extensions gives the ability to declare Special Function Registers (SFRs) so they may be read from and written to.

To save you from entering the SFR declarations every time you create a new project they are commonly placed in header files, with one header file per derivative.

8051 Development Kit Users:

Enter the following line at the top of the main.c source file before the main function:

```
#include <reg51.h>
```

XA Development Kit Users:

Enter the following line at the top of the main.c source file before the main function:

```
#include <regxag3.h>
```

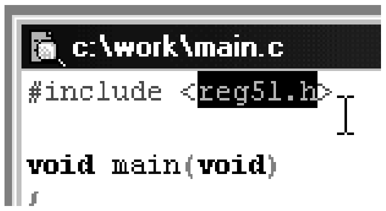
ST6 Development Kit Users:

Enter the following line at the top of the main.c source file before the main function:

```
#include <st6201c.h>
```

All Users:

- Using the mouse select just the filename of the header file (for example “reg51.h”):



```
#include <reg51.h>

void main(void)
{
```

- Press the right mouse button and a menu will pop-up.
- Choose Open Document *filename*.

RIDE will find and open the header file you are including in main.c. You can now examine the contents of the header file to see how SFRs are declared using the language extensions.

Note
For more information on header files please refer to the Header Files chapter.

Below the `#include` add the following line:

```
unsigned char counter = 0;
```

We are going to add two functions. The first, called init, initializes a timer and the timer interrupt. The second function, called timerisr, is the Interrupt Service Routine for the timer. The code for both these functions is different for all three microcontroller families, so please refer to the correct section below for the tools you are using.

8051 Development Kit Users

Add the following code before the main function, but after the line you just added declaring the counter variable:

```
void timerisr(void) interrupt 1
{
  TF0 = 0;     // clear overflow flag
  counter++;
}

void init(void)
{
  TMOD = 0x02; // 8-bit auto-reload timer
  ET0 = 1;     // enable timer interrupt
  EA = 1;      // global interrupt enable
  TR0 = 1;     // run timer
}
```

XA Development Kit Users

Add the following code before the main function, but after the line you just added declaring the counter variable:

```
void timerisr(void) interrupt 1 priority 15
{
  TF0 = 0;     // clear overflow flag
  counter++;
}

void init(void)
{
  TMOD = 0x02; // 8-bit auto-reload timer
  IPA0 = 0x70; // priority 15
  ET0 = 1;     // enable timer interrupt
  EA = 1;      // global interrupt enable
  TR0 = 1;     // run timer
}
```

ST6 Development Kit Users

Add the following code before the main function, but after the line you just added declaring the counter variable:

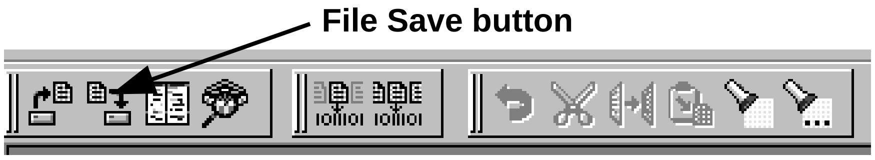
```
void timerisr(void) interrupt 3
{
  TSCR &= 0x7f;  // clear underflow flag
  counter++;
}

void init(void)
{
  TSCR = 0x48;   // timer interrupt enable, prescaler
                 // enable, divide by 1
  IOR = 0x10;    // global interrupt enable
}
```

All Users:

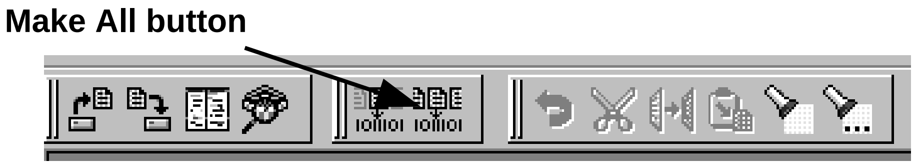
Add the following line inside the main function, just before the `while(1);`

```
init();
```

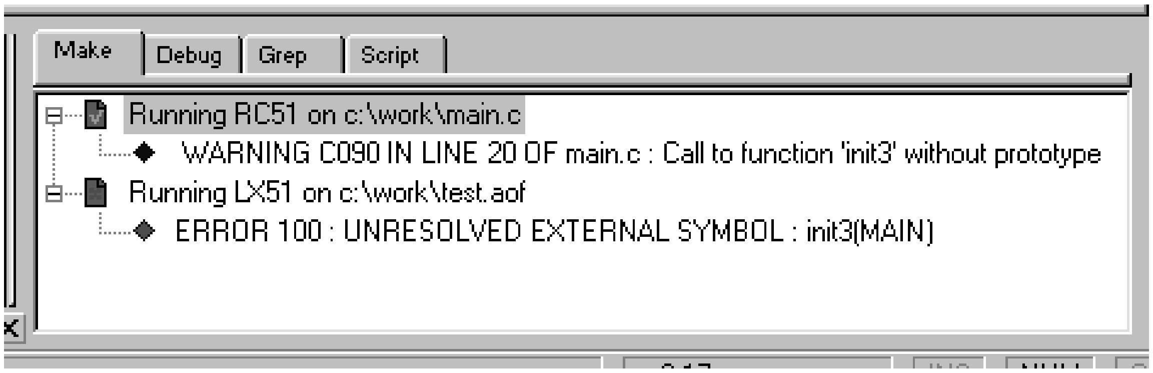
Save the source file by choosing File | Save or clicking on the Save button:



Build the project by choosing Project | Make All or clicking on the Make All button:



If “+” signs appear next to the Compiler or Linker items in the Make window, click on the “+” sign to expand the tree and view the warnings. Double-click on the Compiler warnings to jump to the relevant point in the source code and fix the problem.



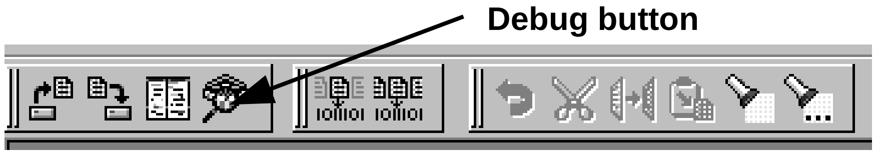
```
Running RC51 on c:\work\main.c
    WARNING C090 IN LINE 20 OF main.c : Call to function 'init3' without prototype
Running LX51 on c:\work\test.aof
    ERROR 100 : UNRESOLVED EXTERNAL SYMBOL : init3(MAIN)
```

Once you have the project successfully built we are ready to start the debugger.

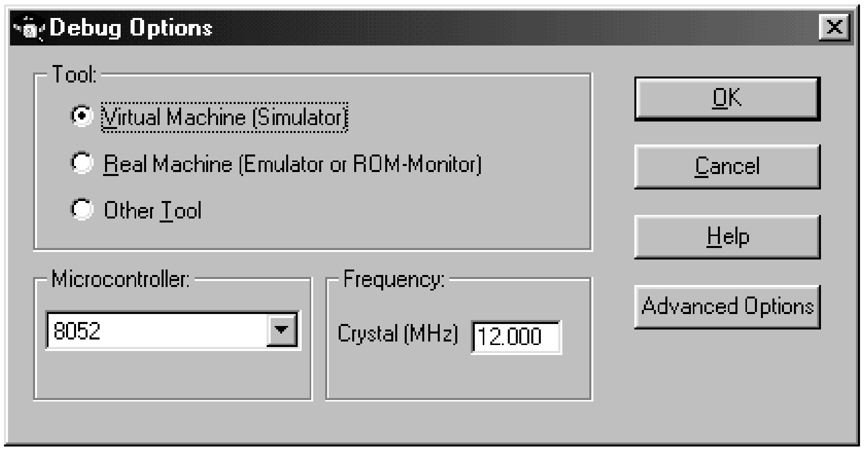
Starting the RIDE Debugger

The debugger is integrated into RIDE and is one mouse-click or menu item away.

To start the debugger either choose Debug | Start test.aof or click on the Debug button:



You will be presented with the following debug options window:

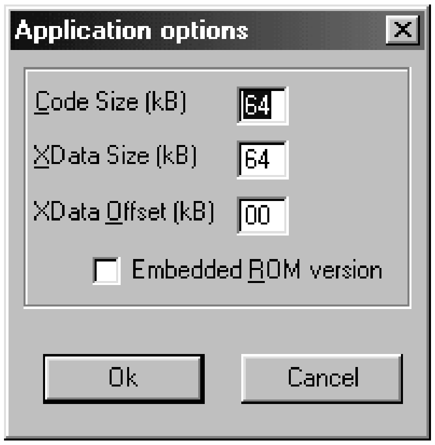


Tool:
[✓ Virtual Machine (Simulator)]
Real Machine (Emulator or ROM-Monitor)
Other Tool

Microcontroller: 8052

Frequency: Crystal (MHz) 12.000

Click on the OK button and the advanced options window will open:



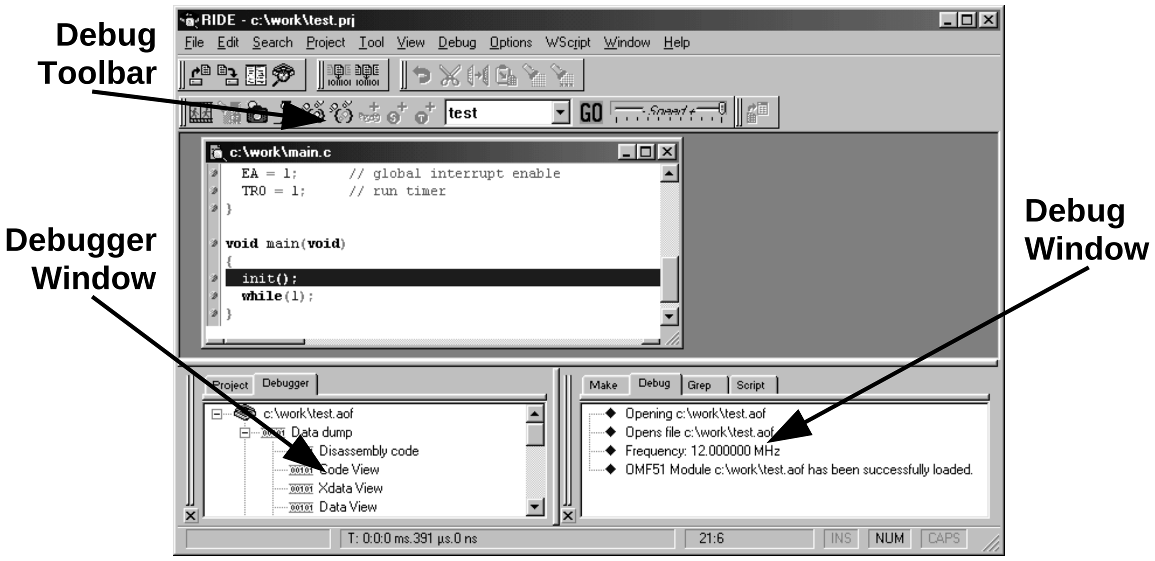
Code Size (kB) 64
XData Size (kB) 64
XData Offset (kB) 00
☐ Embedded ROM version

Click on OK to accept the advanced options then click on OK in the debug options window to accept those options.

Note

Next time the debugger is started the debug options window will not open as you have already confirmed that the options are correct. If you wish to change the debug options either before starting the debugger or while debugging then choose Options | Debug.

The RIDE window should now look something like this:



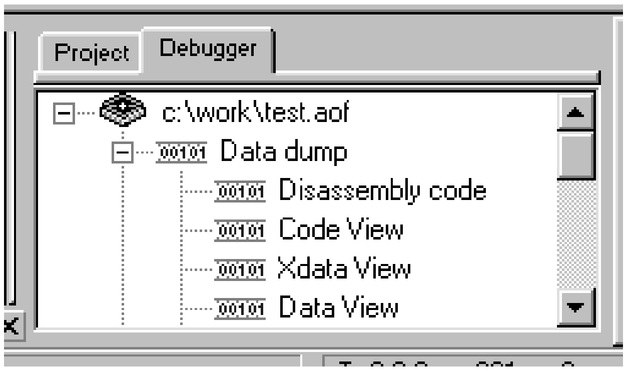
```
  EA = 1;       // global interrupt enable
  TR0 = 1;      // run timer
}

void main(void)
{
  init();
  while(1);
}
```

```
Opening c:\work\test.aof
Opens file c:\work\test.aof
Frequency: 12.000000 MHz
OMF51 Module c:\work\test.aof has been successfully loaded.
```

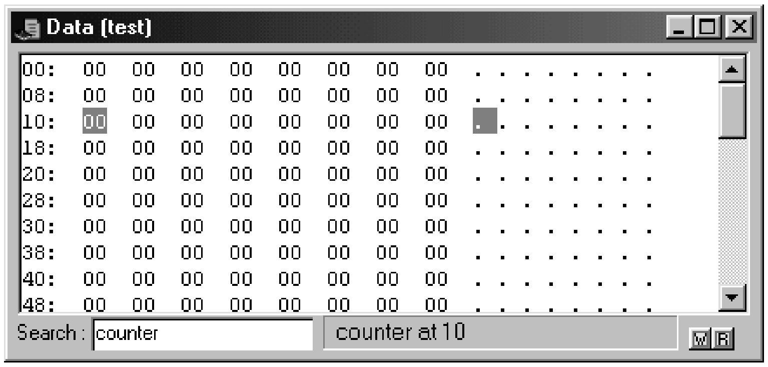
If you are currently not using the largest screen mode available to you then now might be a good time to switch to it. The debugger features many windows that may be opened.

There is a possible point of confusion over the window names in the debugger so both windows will be described.

The **Debugger window** shows a tree view of select debugger windows that may be opened by double-clicking on the time in the tree. The windows listed are memory viewing windows and peripheral windows.



Double-click on the “Data View” item. A window will open showing the contents of memory (in this case Data memory)



```
00:  00  00  00  00  00  00  00  00  . . . . . . . .
08:  00  00  00  00  00  00  00  00  . . . . . . . .
10:  00  00  00  00  00  00  00  00  . . . . . . . .
18:  00  00  00  00  00  00  00  00  . . . . . . . .
20:  00  00  00  00  00  00  00  00  . . . . . . . .
28:  00  00  00  00  00  00  00  00  . . . . . . . .
30:  00  00  00  00  00  00  00  00  . . . . . . . .
38:  00  00  00  00  00  00  00  00  . . . . . . . .
40:  00  00  00  00  00  00  00  00  . . . . . . . .
48:  00  00  00  00  00  00  00  00  . . . . . . . .
Search : counter        counter at 10
```

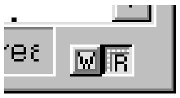
The memory is shown in rows of eight bytes, first in hexadecimal and then in ASCII. If the ASCII equivalent is a character that cannot be displayed then a period is displayed instead.

Static variables may be searched for simply by entering the variable name in the search field. Try entering

```
counter
```

and pressing return. A memory location will be highlighted indicating where the variable is stored.

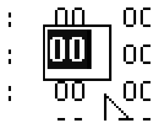
Other features of the Data view are the ability to set write and read access breakpoints (execution will stop when the memory location is written to or read from) by clicking on the "W" and "R" buttons.



Pressing the right mouse button over the Data view window opens a pop-up menu that provides the following functions:

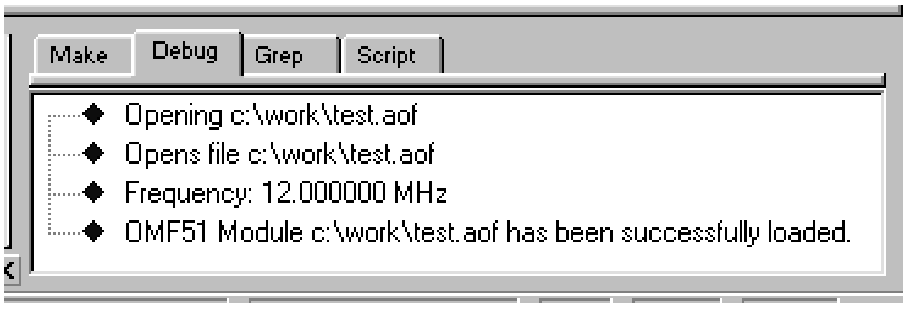
- Associate a memory location with a symbol
- Associate a memory location with a function generator
- Change the number of bytes shown per line
- Go to a specific address or symbol
- Fill memory with a value
- Toggle write and read breakpoints

Double-clicking on a memory location in the window allows the value at that location to be modified.



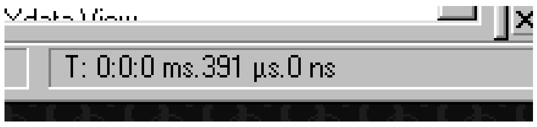
Close the Data View window, but make sure that no write or read breakpoints have been set and that any memory locations that you modified are set back to their original values.

The **Debug window** shows messages that relate to the debug session. For example if you stop and start the simulation a message will appear for each, complete with a time stamp.



```
Opening c:\work\test.aof
Opens file c:\work\test.aof
Frequency: 12.000000 MHz
OMF51 Module c:\work\test.aof has been successfully loaded.
```

The blue bar in the source code window indicates the current execution point. It highlights the next piece of code to be executed. The Compiler automatically includes some startup code (written in assembler) that is executed before the main function is reached. When you start the debugger it automatically executes the startup code and stops at the first line in the main function. If you look at the status bar at the bottom of the RIDE window you will see a time displayed:

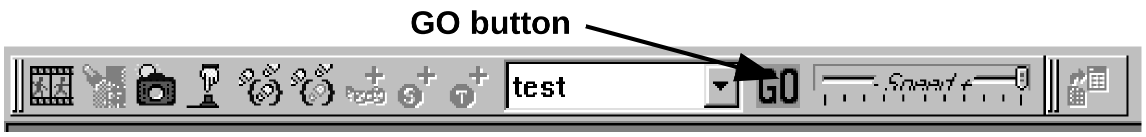

T: 0:0:0 ms.391 µs.0 ns

This is the current execution time so far. As the only code that has been executed is the startup code, this time shows how long the startup code took to execute.

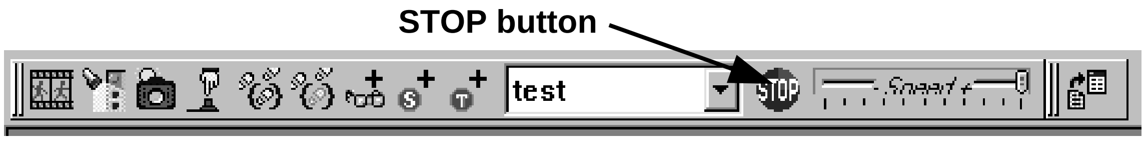
Note
During a simulation the debugger will bring the source code window to the front. If the source code window is maximized or covering other windows then it will not be possible to view those other windows. We recommend that you do not maximize the source code window and that you drag it out of the way when you want to view other windows.

Enough of the window descriptions, lets watch the code be simulated!

You may have noticed the large GO button on the toolbar. It will come as no surprise to learn that clicking on the GO button will start the simulation. Do that now.

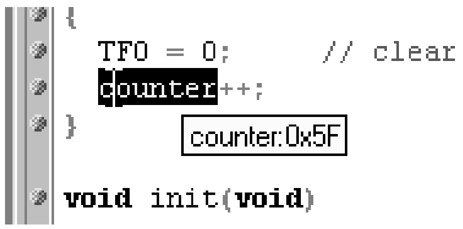


Two things indicate that the simulation is taking place: periodically the execution time at the bottom of the RIDE window will update and the GO button has changed into a STOP button:



Click on the STOP button to stop the simulation.

Using the pointer, highlight “counter” in the source code window then hover the pointer over the highlighted “counter” text for a moment. A tooltip will appear showing the current value of the variable counter in hexadecimal:



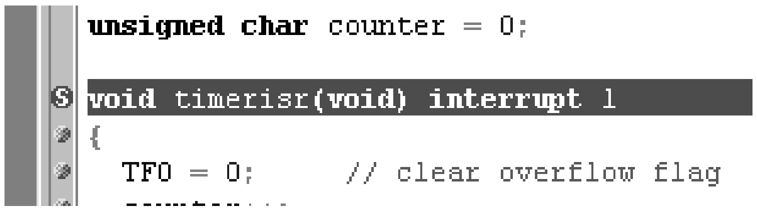
```
{
   TFO = 0;      // clear
   counter++;
}       counter:0x5F

void init(void)
```

This may be used for any identifier in the source code window. Try it for the SFR TFO and a function name.

Breakpoints and Measuring Execution Time

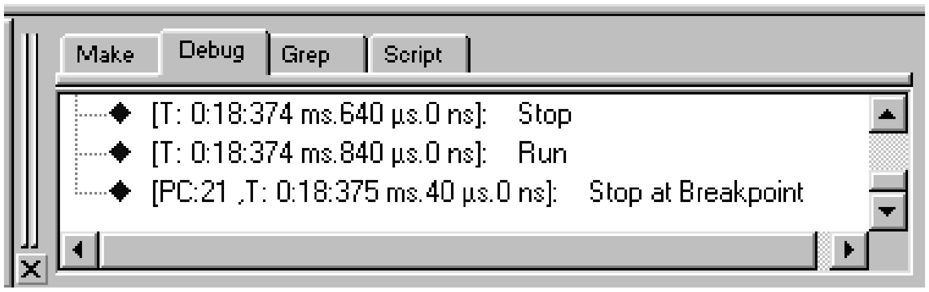
To the left of each C source line that generated code (and therefore was not optimized out) a green dot appears.

Set a breakpoint on the first line of the timer Interrupt Service Routine by clicking on the green dot. The dot will turn red with an “S” inside it and a red bar will highlight the source line.



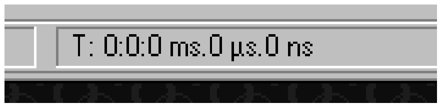
```
unsigned char counter = 0;

void timerisr(void) interrupt 1
{
  TF0 = 0;      // clear overflow flag
```

Click on the GO button. The simulation will stop almost immediately. If you look at the debug window a message will indicate that the breakpoint was reached:



```
[T: 0:18:374 ms.640 µs.0 ns]:   Stop
[T: 0:18:374 ms.840 µs.0 ns]:   Run
[PC:21 ,T: 0:18:375 ms.40 µs.0 ns]:   Stop at Breakpoint
```

We are going to measure the time between interrupts. Choose Debug | Reset Time and note that the current execution time shown at the bottom of the RIDE window has been reset to zero:



```
T: 0:0:0 ms.0 µs.0 ns
```

Now click on the GO button to execute up to the start of the next interrupt. The execution time between interrupts will now be shown at the bottom of the RIDE window.

Note

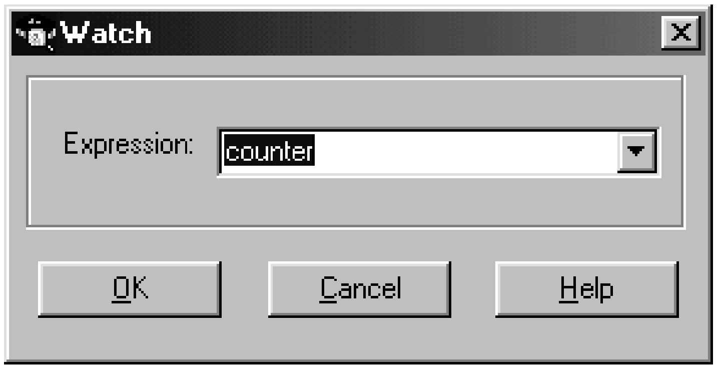
Using breakpoints and resetting the execution time makes it possible to measure the time to execute any piece of code, including parts of functions or the whole application, as well as verifying if you have correctly configured a timer to generate interrupts at a specific rate.

Setting Watchpoints

Don’t remove the breakpoint just yet.

Rather than highlighting the counter variable to see its value it would be useful if the value was always shown on the screen. To do that we need to add the counter variable to the Watch window.

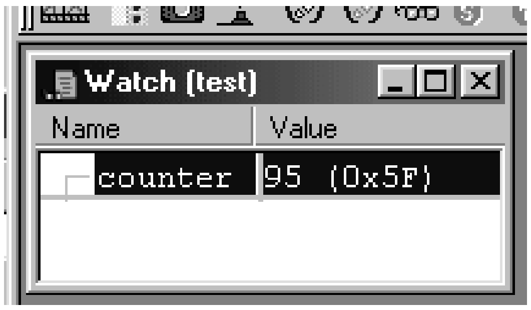
- Open the Watch window by choosing View | Watch.
- With the pointer over the Watch window press the right mouse button.
- Choose Add from the menu that pops up.
- Enter **`counter`** into the expression field.



Expression: counter

- Click on OK

The Watch window will now show the current value of the counter variable in decimal, with the hexadecimal equivalent in parentheses.



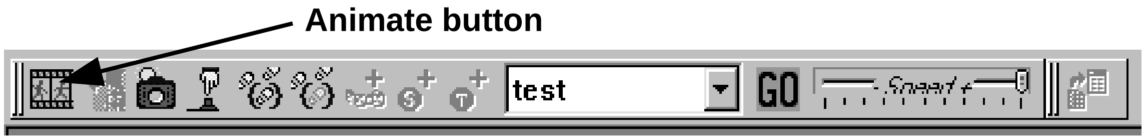
```
counter   95 (0x5F)
```

Click on the GO button to run to the next breakpoint. The value of counter will increment and turn red. The value turns red to indicate that it has changed, and therefore draws attention to itself.

Simulation Animation

It’s as fun to see the simulation animate as it is to say the title of this section.

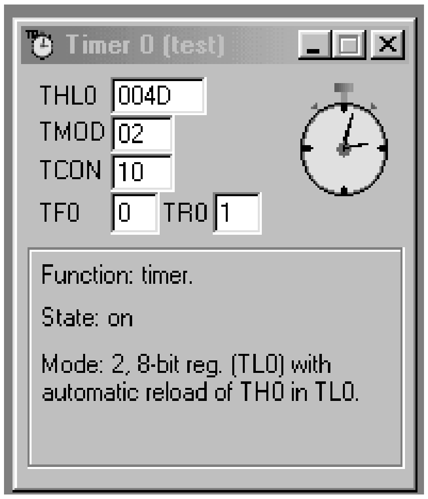
First remove the breakpoint by clicking on the red dot with an “S” inside it. The red bar will also disappear.
Choose Debug | Animated Mode or click on the Animate button.



Finally click on the GO button.

Every once in a while the blue bar will move to the Interrupt Service Routine, visit each source line inside the ISR in turn then return to the `while(1)`. At the same time the value of counter will increment in the Watch window.

In the Debugger window, scroll down to the list of peripherals and double click on “timer” or “timer0”, The timer peripheral window will open with the value of the timer count register constantly changing as the execution progresses. The screen shot below is for the 8051 timer so the one you see may look different.


Timer 0 (test)

THL0 004D
TMOD 02
TCON 10
TF0 0 TR0 1

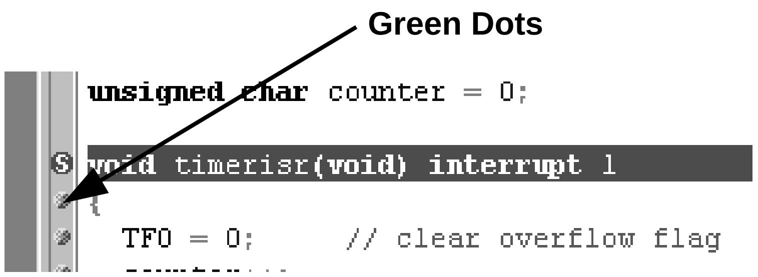
Function: timer.
State: on
Mode: 2, 8-bit reg. (TL0) with automatic reload of TH0 in TL0.

Stop the simulation by clicking on the STOP button and either choose Debug | Animated Mode or click on the Animate button to turn the animate mode off. You can also close the timer peripheral window.

Stepping Through Code

The RIDE debugger makes it possible to step through the simulation, one source line at a time or one instruction at a time.

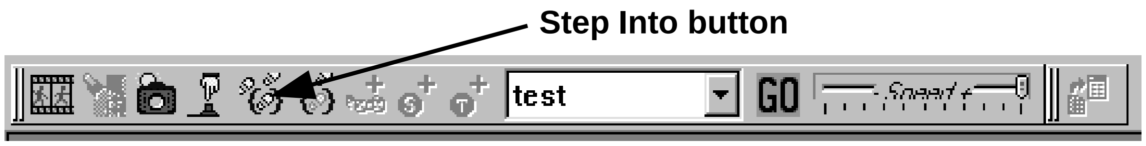
Set a breakpoint on the first line of the Interrupt Service Routine by clicking on the green dot.



```
unsigned char counter = 0;

void timerisr(void) interrupt 1
{
  TF0 = 0;      // clear overflow flag
```

Click on the GO button to simulate up to that line.

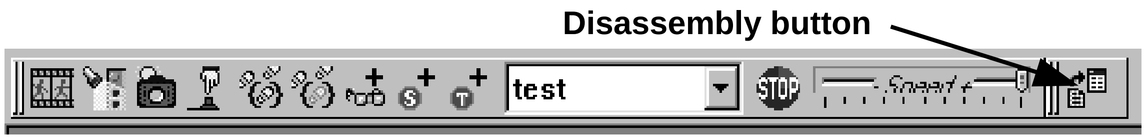
To step through the source lines one at a time choose Debug | Step Into or click on the Step Into button:



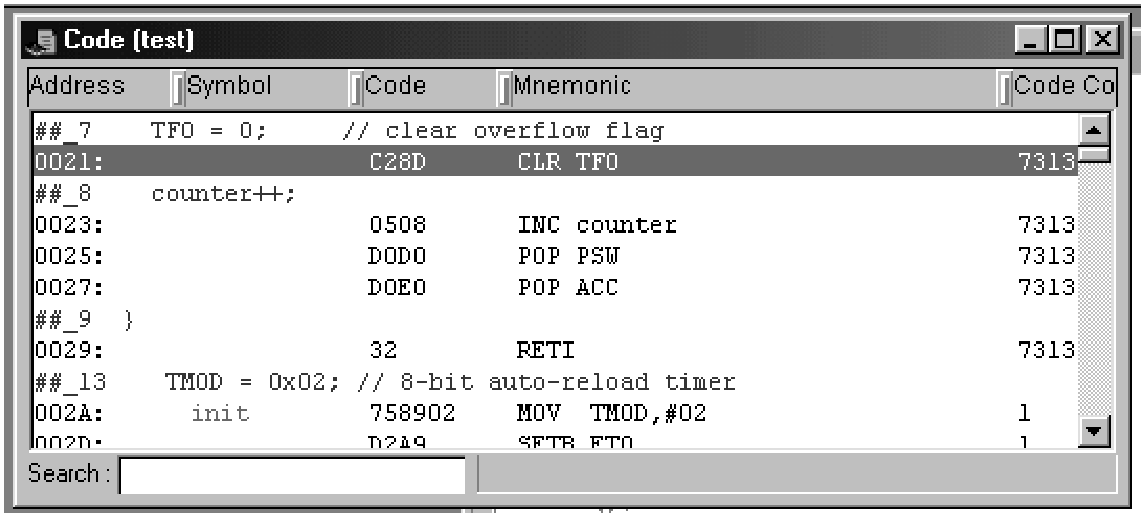
The blue bar will move onto the next source line on each press of the button or selection of the menu entry. At the same time the current execution time shown at the bottom of the RIDE window will increase.

Click on GO to execute up to the ISR again.

To view the assembly code equivalent of the C source code open the Disassembly window by choosing View | Code or clicking on the Disassembly button:



The disassembly window will look something like:



```
Address      Symbol      Code        Mnemonic                       Code Co
##_7    TFO = 0;     // clear overflow flag
0021:                C28D        CLR TFO                            7313
##_8    counter++;
0023:                0508        INC counter                        7313
0025:                D0D0        POP PSW                            7313
0027:                D0E0        POP ACC                            7313
##_9  }
0029:                32          RETI                               7313
##_13    TMOD = 0x02; // 8-bit auto-reload timer
002A:     init       758902      MOV  TMOD,#02                      1
002D:                D2A9        SETB ETO                           1
```

Source code lines are shown in purple, with the assembler equivalent immediately below.

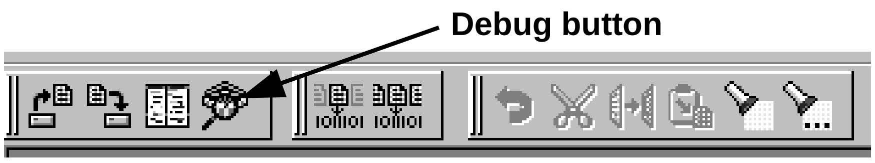
When the Disassembly window has the focus (the title bar is not gray), performing a Step Into will step through assembly code.
When the source code window has the focus, performing a Step Into will step through C source code.

Open the Register window by choosing View | Main Registers, then repeatedly press the Step Into button and watch the bar move and the registers change accordingly.

Even if the Disassembly window has the focus you can click on the GO button to execute up to the next breakpoint.

When you are finished close the Disassembly window and remove the breakpoint. Also close the Register window as well.

Either choose Debug | Terminate test.aof or click on the debug button to end the debug session.



Note

When simulating more complex applications sometimes clicking on the STOP button appears to have no effect. However if you wait for a short while the disassembly window will open and a Break button will appear on the toolbar. If you wait some more then eventually the simulation will stop.

This appears to be a bug but it is not. You tried to stop the simulation while the debugger was in the middle of a large block of assembly code that corresponded to the current C source line. Typically the assembly code will be library routines such as printf. The debugger recognized that it would take some time to simulate all the assembly code and reach the next C source line so it opened the disassembly window for you at the current execution point in the assembly code. The Break button is provided to allow you to stop the simulation immediately in the assembly code, rather than wait for the next C source line to be reached.

If you do not like this behavior of the simulator then identify the C source lines that take some time to execute and set a breakpoint on the following lines.

Final Code Additions

In order to show a couple of useful features of the debugger we need to make some additions to our code.

Change the line:

```
unsigned char counter = 0;
```

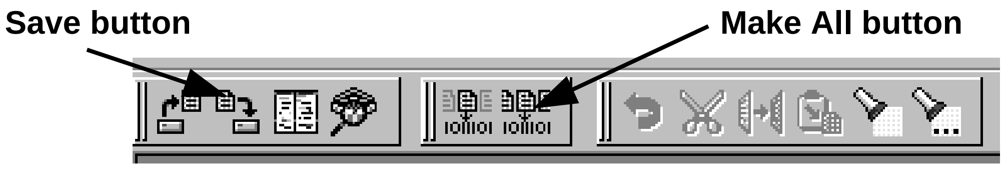
To:

```
unsigned char counter = 0, toggler = 0;
```

Just below `counter++;` inside the timerisr function, add the following line:

```
toggler = 0xFF – toggler;
```

Save the file by clicking on the Save button and rebuild the project by clicking on the Make All button



If there are any errors or warnings go back to your source code and fix them.

Tracing and Displaying Waveforms

The RIDE debugger has the ability to graphically display waveforms relating to a microcontroller pin or a variable. This allows the relationship between variables, inputs and outputs to be clearly shown.

Start the debugger by choosing Debug | Start test.aof or by clicking on the Debug button.

Open the Watch window by choosing View | Watch.

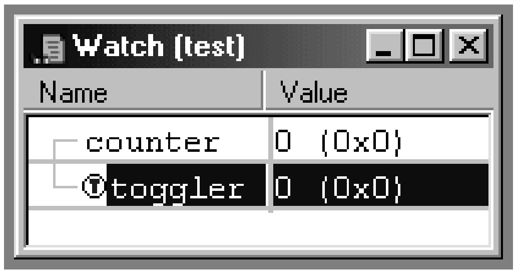
With the pointer over the Watch window press the right mouse button and choose Add from the menu that pops up.

In the expression field enter

`toggler`

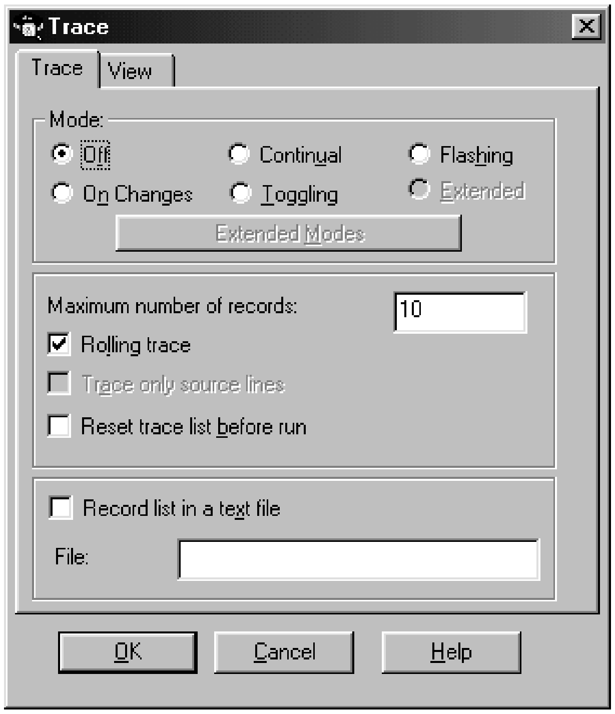
And click on OK. Toggler should appear in the Watch window.

Select toggler in the Watch window so it is highlighted in blue and press the right mouse button again. This time select Add/Delete from Trace List from the menu that pops up. A small "T" in a circle will appear next to toggler in the Watch window.



```
counter    0  (0x0)
Ⓣtoggler   0  (0x0)
```

Choose View | Trace | Options. The Trace Options window will open and look like:



Mode: [✓ Off]

Maximum number of records: 10

[✓] Rolling trace

[ ] Trace only source lines

[ ] Reset trace list before run

[ ] Record list in a text file

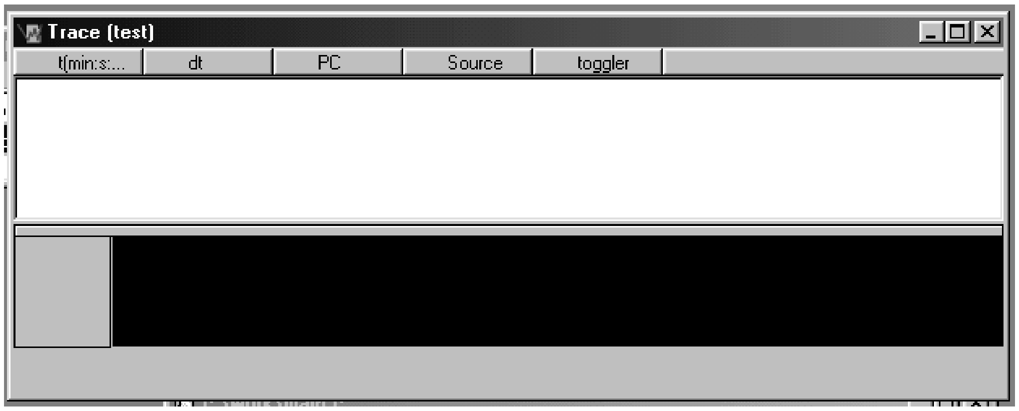
File:

Select "On Changes" and in the Maximum Number of records field enter

50

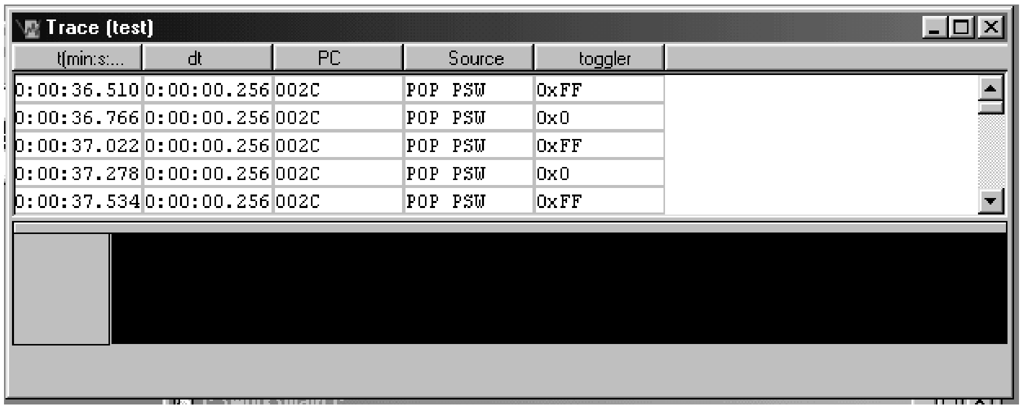
Click on OK.

To open the Trace window choose View | Trace | View.



Click on the GO button to start the simulation and allow it to run for a few moments before stopping it by clicking on the STOP button.

The top part of the Trace window will now show the trace records for the last 50 changes of toggler:



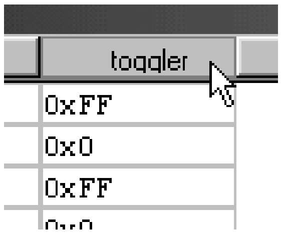
t(min:s:...	dt	PC	Source	toggler
0:00:36.510	0:00:00.256	002C	POP PSW	0xFF
0:00:36.766	0:00:00.256	002C	POP PSW	0x0
0:00:37.022	0:00:00.256	002C	POP PSW	0xFF
0:00:37.278	0:00:00.256	002C	POP PSW	0x0
0:00:37.534	0:00:00.256	002C	POP PSW	0xFF

You can scroll through the trace records. The columns are:

t	The simulation time
dt	The change in simulation time between the records
PC	The Program Counter at that point
Source	The source code to be executed at that point
toggler	The value of toggler at that point

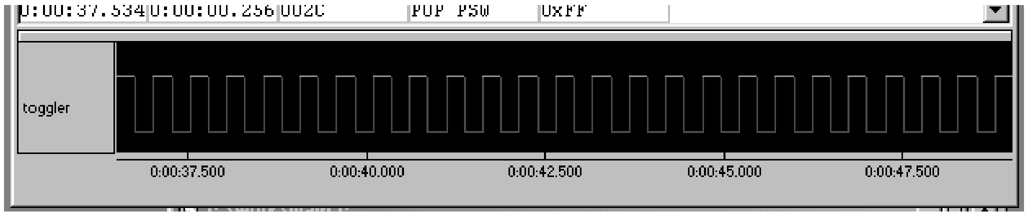
Because we selected the “On Changes” option, records are only generated when the variables being traced (in our case toggler) change.

To view a waveform of toggler click on the title button of the toggler column



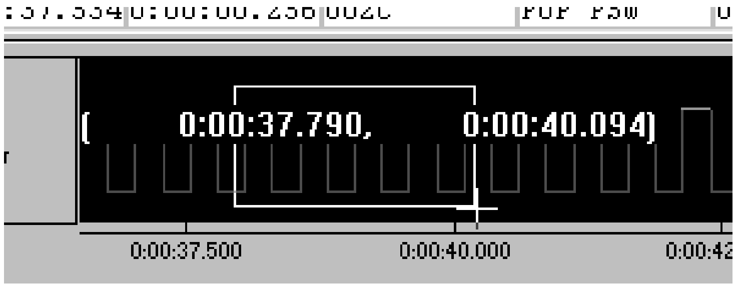
toggler
0xFF
0x0
0xFF

The lower part of the trace window will show a square wave as toggler changed between 0xFF and 0x00.



0:00:37.534	0:00:00.256	002C	POP PSW	0xFF

It is possible to zoom in on the waveform by dragging a box around the area of interest. Timing information is shown along the bottom of the window.



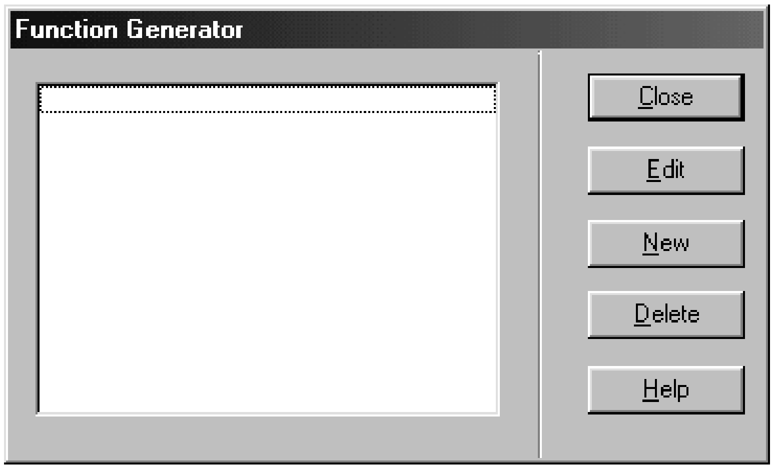
Pressing the right mouse button while the point is over the waveform will zoom out to the original view.

The trace records may be saved as a text file. Press the right mouse button over the trace records and choose Save As from the pop-up menu.
Browse to the desired folder and enter the filename. Click on Open to save.

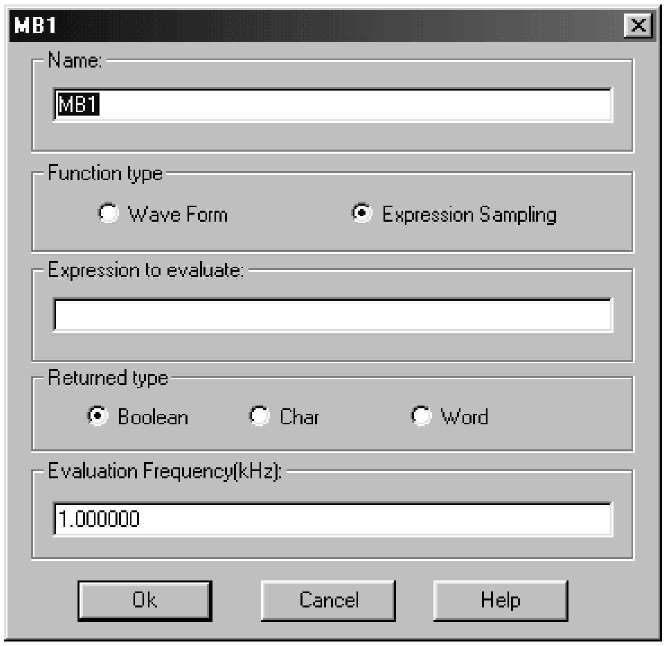
Generating Waveforms on Pins

The RIDE debugger makes it easy to generate waveforms on a specific pin of the microcontroller being simulated. Using function generators the program may be tested with external input stimuli.

Choose View | Function Generators followed by Options from the pop-up menu. The function generator main window will open.



Click on New to create a new function generator. The Function Generator Options window will open.



Name: MB1

Function type: Wave Form [✓ Expression Sampling]

Expression to evaluate:

Returned type: [✓ Boolean] Char Word

Evaluation Frequency(kHz): 1.000000

In the Name field enter

`square wave`

Select the Wave Form option.

In the Expression to Evaluate field enter

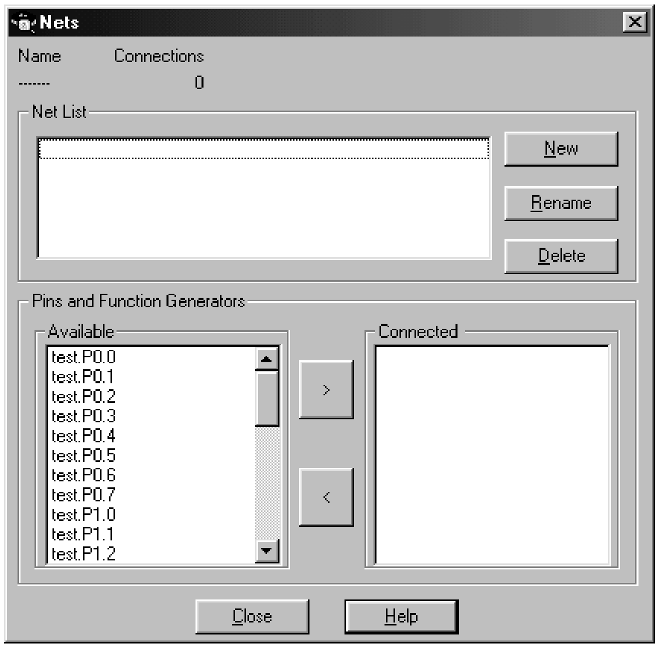
`(L100U,H200U)`

and click on OK. This will create a function generator that generates a square wave, low for 100us and high for 200us.

Click on Close in the function generator main window.

The function generator needs to be attached to a microcontroller pin, and that is achieved using a netlist. A netlist is a form of representing electrical circuits by giving each connection (called a node) a unique name, then specifying what is connected to the node.

Choose View | Nets. You will be presented with the Nets window.



Click on New to create a new node. Net0 will appear in the Net List area.

8051 and XA Users

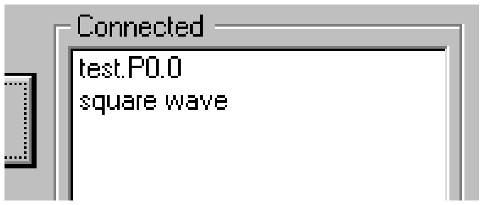
Find test.P0.0 in the Available list, select it and click on the “>” button. Test.P0.0 will appear on the right side in the Connected list to show it is connected to the node.

ST6 Users

Find test.PB0 in the Available list, select it and click on the “>” button. Test.PB0 will appear on the right side in the Connected list to show it is connected to the node.

All Users

Find square wave in the Available list, select it and click on the “>” button. Square wave will appear on the right side in the Connected list to show it is connected to the node.



Click on the Close button to close the Nets window.

Having a waveform on a pin is no good unless we can view it. We therefore need to add the pin to the watch window.

With the pointer over the Watch window press the right mouse button and choose Add from the pop-up menu.

8051 and XA Users

In the expression field enter:

`P0.0`

ST6 Users

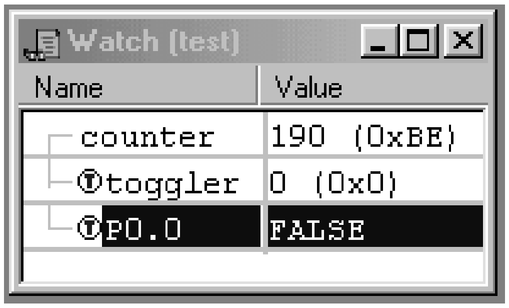
In the expression field enter:

`PB.0`

All Users

Click on OK.

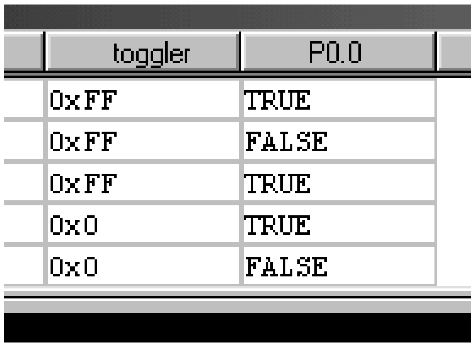
Select the pin in the watch window so it is highlighted with a blue bar and press the right mouse button. Choose Add/Delete from Trace List. A small blue "T" in a circle will appear next to the pin name.



Name	Value
counter	190 (0xBE)
Ⓣtoggler	0 (0x0)
ⓉP0.0	FALSE

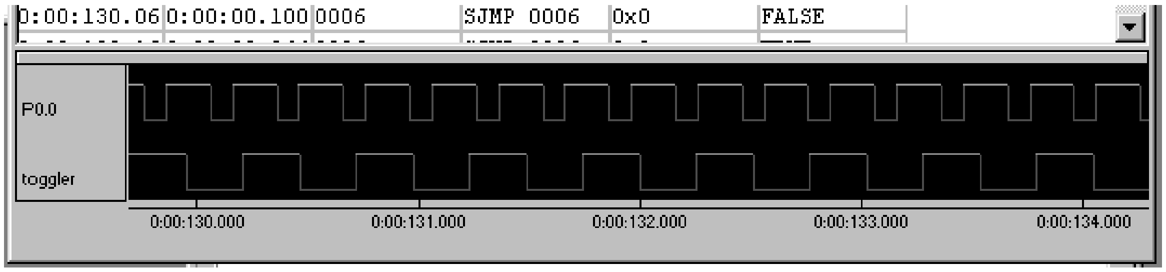
Start the simulation by pressing the GO button and allow it to run for a short while, then stop the simulation by pressing the STOP button.

The trace records in the Trace window will now have a column for the port pin, with entries either "TRUE" or "FALSE".



toggler	P0.0
0xFF	TRUE
0xFF	FALSE
0xFF	TRUE
0x0	TRUE
0x0	FALSE

Press the toggler column title button followed by the title button of the port pin column to display the waveforms of both.



```
0:00:130.06 0:00:00.100 0006        SJMP 0006    0x0        FALSE
```

You can zoom in to obtain a closer view of the waveforms.

Additional Things to Try

Add another variable to the code which is 0xFF only when toggler is 0xFF, and the port pin is high, otherwise it is 0x00.

Add the variable to the watch window and trace it to see its waveform.

Chapter 5. Compiler Listing and Linker Map Files

This section is intended as a brief introduction to the Compiler listing file and the Linker map file. For a more detailed examination of the file contents please refer to the Compiler and Linker manuals.

Understanding the Compiler Listing File

Choose View | Listing from Compiler. A window will open showing the listing file (.lst) generated by the Compiler when it processed main.c.

The Compiler listing file is very useful in understanding what the Compiler did. The listing file contains six consecutive pieces of information.

Command Line Invocation

This information shows exactly how the Compiler would be called from the command line to obtain the exact same result. This information is very useful if you want to relate the Compiler options in RIDE with the command line options, or for better understanding how the command line version of the Compiler is used.

Source Code

A listing of the input source code is included in the listing file. Each line is numbered and source lines that actually generated code have a second number which relates to the nesting level of the code.
It is possible to change how this section looks and include the source code from header files by changing the compiler options.

Generated Assembler Equivalent

The assembly code generated by the compiler is listed to indicate exactly what the compiler did when it processed the source code. This section is useful for tracking down bugs in source code and understanding better how the compiler works.
Throughout the assembly code there are comments like:

```
; SOURCE LINE # 8
```

The assembly code immediately following the comment relates to that source code line, which may be found by looking for the line number in the Source Code section of the listing file.

The assembly code is divided up into functions.

Symbol Table

The symbol table lists all the static variables and functions that were found in the source file. For each one various information is listed.

Module Information

The module information table gives a summary of the memory requirements of the module. Some memory may be overlaid.

Errors and Warnings

The final piece of information given in the listing file is a summary of the number of warnings and errors that were generated by the compiler. Note however that it does not list the actual warnings and errors.

Note

You may have noticed at the start of the listing file a line that looks like:

```
      QCW(0x00002D32)
```

The hexadecimal value is the internal compiler representation of most of the compiler options. This value is for internal use only and aids Raisonance in supporting customers, so you can safely ignore it.

Understanding the Linker Map File

Choose View | Map Report from Linker. A window will open showing the Map File (.m51 for 8051, .mxa for XA and .mst for ST6) generated by the Linker when it processed all the object files in the project.

The Linker Map File is invaluable in understanding what the Linker did. It is probably the most useful piece of information generated by the tools and understanding the contents will vastly improve the chances of a project’s success. Indeed, without understanding it many projects will never work. The listing file contains eight consecutive pieces of information (nine for the ST6).

Command Line

This information shows exactly how the linker would be called from the command line to obtain the exact same result. This information is very useful if you want to relate the linker options in RIDE with the command line options, or for better understanding how the command line version of the linker is used.

Memory Model

This section shows the memory model that was used for the project. This is useful for double-checking that the intended memory model was used.

Input Modules

A list of all the input modules is given. The input modules include the modules generated by the source files, plus any library modules if required.
After the path to each module, the module name is given in parentheses.

Link Map

The Link Map is the memory map of the project and shows exactly where every piece of code and data was located.
The memory map is given as a table, divided into the various memory areas supported by the microcontroller.

Each source file is divided up into segments. All the code in a function will be placed in a single segment. All the static DATA variables will be placed in another segment. All the static IDATA variables will be placed in yet another segment.

Each line in the link map is one segment, showing where it was located and how big the segment is.

Function segment names are given the following format:

memorytype?*functionname*?*modulename*

Data segment names are given the following format:

memorytype?*modulename*

Some special segments do not follow the above formats.

More information on segment names may be found in the Compiler and Linker manuals.

Using the Link Map you can quickly and easily see which regions of memory your project uses and how much memory your project uses. Also you can see if you configured the Linker correctly to use only specific regions of memory.

Executable Summary

The executable summary gives the total memory usage for each of the various memory areas, excluding dynamic memory requirements such as the stack.

Reference Map

The reference map shows which executable segments call other segments. This is useful in determining if the linker correctly figured out such things as indirect function calls.

Symbol Table

Every variable, function and code line are listed in the symbol table along with the address they were located at and their type. This is very useful when using the debugger as you can look up the location a variable is stored at then watch that location during the simulation.

Project Call Tree (ST6 only)

The Project Call Tree is an analysis of the stack usage by the project. The ST6 supports only six stack levels and the linker makes an attempt to determine if the project will go over the six level limit. This section indicates what the linker did to work out the stack usage of the project.

Errors and Warnings

The final piece of information given in the map file is a summary of the number of warnings and errors that were generated by the linker. Note however that it does not list the actual warnings and errors.

Note

Sometimes it will appear that a variable has been omitted from the symbol table. This isn’t a bug, but an indication that the variable was optimized out and therefore is not present in the .aof file. You cannot therefore watch that variable in the debugger.

If you need to see the variable then try declaring the variable as “volatile”. The volatile keyword will instruct the Compiler to avoid performing optimizations on the variable.

```
      volatile unsigned char foo;
```

Note however that using the volatile keyword will change the code generated and therefore may not behave in quite the same way as the final version without the volatile keyword.

Chapter 6. Header Files

Each tools installation contains a set of pre-written header files that declare Special Function Registers for many derivatives of the microcontroller family you are using.

Each header file is specific to a particular derivative, however it is easy to generate your own header files if you find that one for the derivative you are using is not included.

Follow the steps below to generate your own header file.

1. Find the header file of the device most similar to the one you are using. Alternatively find the header file of the most basic derivative. For example for the 8051 it would be reg51.h.

Header files are stored in the INC folder in 8051 and XA installations and the INCST6 folder in ST6 installations.

2. Make a copy of it in the same folder and rename the new file to the name of the derivative you are using, following the same naming format at the other header files.

3. Open the header file in RIDE or a text editor such as Notepad.

4. Using the table of SFRs from the datasheet of the device you are using, simply enter the missing SFRs following the same declaration format:

`at` *`address`* `sfr` *`name`*`;`

for example on 8051s the SFR P0 is located at 0x80, so it is declared as:

```
at 0x80 sfr P0;
```

The 8051 and XA also allow declarations of individual bits in the bit-addressable SFRs. They are declared using the following format:

`at` *`address`* `sbit` *`name`*`;`

where the address is the address of the SFR which contains the bit plus the bit position.

For example:

Address	Corresponds to
0x80	Bit 0 in the SFR at 0x80
0x81	Bit 1 in the SFR at 0x80
0x83	Bit 3 in the SFR at 0x80

Table 6.1 SFR Bit Addresses

So bit 1 in the SFR at 0x80 would be declared as:

```
at 0x81 sbit FOO;
```

As you can see, it is quick and easy to generate your own header files.

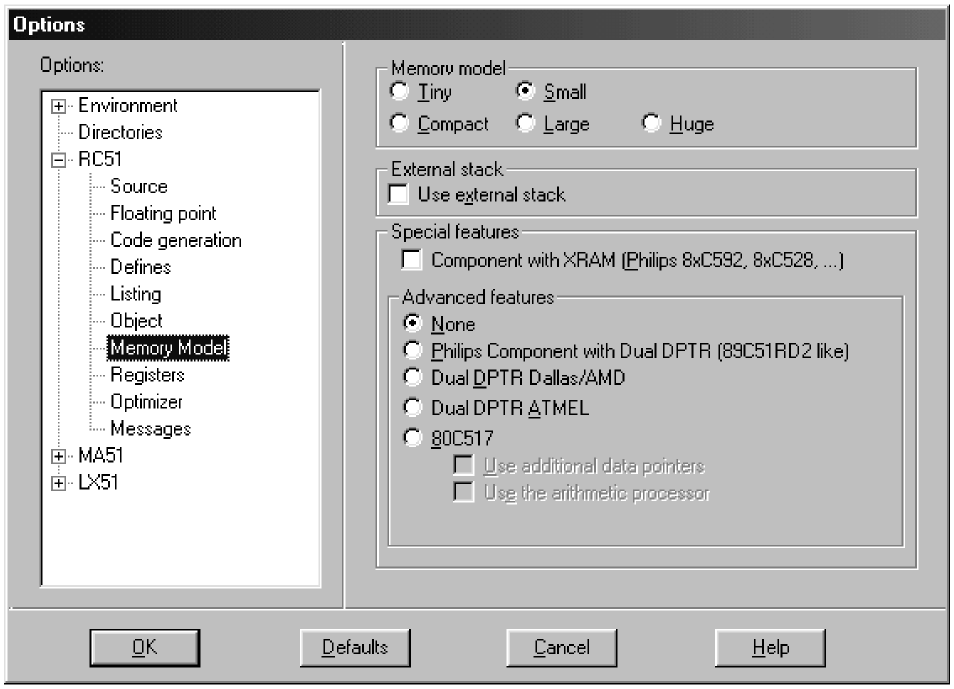
Once generated you may include them in the C source code in the usual manner. For example:

```
#include <mydevice.h>
```

Chapter 7. Compiler

Changing the Compiler Settings in RIDE

With a project loaded into RIDE the Compiler options for the project as a whole may be accessed by choosing Options | Project then expanding the tree for the Compiler and clicking on the various sections. The following screenshot is for the 8051:



Memory model: Tiny [✓ Small] Compact Large Huge

External stack: ☐ Use external stack

Special features: ☐ Component with XRAM (Philips 8xC592, 8xC528, ...)

Advanced features: [✓ None] Philips Component with Dual DPTR (89C51RD2 like) Dual DPTR Dallas/AMD Dual DPTR ATMEL 80C517

☐ Use additional data pointers
☐ Use the arithmetic processor

Once all the options have been set up accordingly click on OK to confirm them.

The Compiler options may also be set individually for each source file in the project:

- In the Project window select the source file whose options you wish to change
- Press the right mouse button.
- From the pop-up menu choose Options | Local Options. A window will open allowing you to change the Compiler options for that source file.

8051 Compiler Options Overview

The following Compiler options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Source

Option	Description
ANSI C	Strictly ANSI C only allowed. No language extensions. Directive: NOEXTEND
80C51 Specific Language Extensions	ANSI C plus language extensions allowed. Directive: EXTEND
'struct/union/enum' optional	Allows the struct, union and enum keywords to be omitted when declaring structures, unions and enumerators in certain situations. Directive: SUE_OPT

Table 7.1 8051 Compiler Source Settings

Floating Point

Option	Description
No floating point	No floating point variables are allowed in the project. Directive: FP(NOFLOAT)
IEEE: Standard	Allows single-precision floating point numbers (IEEE-754) – little endian Directive: FP(IEEE,STANDARD)
IEEE: Reversed (251 compatible)	Allows single-precision floating point numbers (IEEE-754) – big endian Directive: FP(IEEE,FP251)
BCD: All types	Allows 32-bit, 48-bit and 56-bit floating point numbers in BCD format. Directive: FP(BCD,ALL)
BCD: “float” only	Allows 32-bit floating point numbers in BCD format. Directive: FP(BCD.FLOAT)
BCD: “double” only	Allows 48-bit floating-point numbers in BCD format. Directive: FP(BCD,DOUBLE)
BCD: “long double” only	Allows 56-bit floating-point numbers in BCD format. Directive: FP(BCD,LONG)

Table 7.2 8051 Compiler Floating Point Settings

Code Generation

Option	Description
Enable ANSI Integer Promotion Rules	If selected chars will be promoted to ints before comparison. This does not generate optimal code on 8-bit microcontrollers such as the 8051, but is included for ANSI compliance. Directive: IP/NOIP
Stack Automatic Variables	With this option on all automatic variables will be located on the stack rather than in fixed memory locations. The result is reentrant but larger code. Directive: AUTO/NOAUTO
Initialize static variables to zero	Includes startup code to initialize non-initialized static variables to zero. This is a safeguard that can be used against uninitialized variables. Directive: IS/NOIS
Generic	When selected all non-memory specific pointers are generic pointers. When not selected, non-memory specific pointers point to the default memory space for the current memory model. Directive: GENERIC/NOGENERIC
Unsigned characters	When selected chars are converted to unsigned chars. Directive: UNSIGNEDCHAR/SIGNEDCHAR

Table 7.3 8051 Compiler Code Generation Settings

Note

With the generic option turned on the following pointer declaration results in a pointer that can point to any memory space, however the code to manipulate the pointer is large and involves library calls:

```
      unsigned char *foo;
```

With the generic option turned off the same pointer declaration results in a pointer that points to the default memory space for the current memory model (such as DATA in the Small memory model), however generic pointers can still be declared using the generic keyword, for example

```
      unsigned char generic *foo;
```

This makes turning the generic option off the best setting to use, however be aware of the effects.

Defines

This section allows you to enter identifiers that are defined for such things as conditional compilation.
Directive: DEFINE(*text*)

Listing

Option	Description
Generate listing	If selected a .lst listing file will be generated Directive: PR(*filename*)/NOPR
Show lines omitted from compilation	If selected then the source code listing in the listing file will include files that were not involved in the compilation Directive: CO/NOCO
Display the contents of the include files	If selected then the contents of the include files are inserted into the source code listing in the listing file Directive: LC/(none)
Generate Preprocessor listing file	If selected then a separate file will be generated showing macro expansions Directive: PP(*filename*)/(none)
Append assembly mnemonics list	If selected then the compiler generated assembly code will be included in the listing file Directive: CD/NOCD
Generate a list of symbols	If selected then a symbol table will be included in the listing file Directive: SB/NOSB
Insert form feeds at the end of pages	If selected then form feed control characters will be inserted into the listing file Directive: PL(*lines*)/(none)
Number of lines printed per page	Specifies how long a page is Directive: PL(*lines*)
Number of characters printed per line	Specifies how wide a line is Directive: PW(*characters*)

Table 7.4 8051 Compiler Listing Settings

Object

Option	Description
Generate an assembler source file	The compiler will generate a .src file which is the complete assembler equivalent of the C source code. The .src file may be assembled by the assembler. Directive: SRC
Generate an object file	The compiler will generate a relocatable object file that make be linked with other object files Directive: OBJECT(*filename*)
Debugging information – no information	No debugging information will be included in the object file. Directive: NODB
Debugging information – Standard	Includes basic debugging information in the object file, compatible with the original Intel OMF-51 specification. Directive: DB
Debugging information – Extended	Includes extended debugging information in the object file, including type information. Directive: OE(1)
Debugging information – Extended 1997 version	Includes yet more debugging information in the object file, including the temporary locations of automatic variables when located in registers. Directive: OE(2)
Generate interrupt vectors	The compiler will generate interrupt vectors automatically for interrupt functions Directive: INTVECTOR(*offset*)/NOINTVECTOR
Interval for interrupt vectors	The number of bytes between interrupt vectors Directive: INTERVAL(*bytes*)
Offset for interrupt vectors	The vector for interrupt zero will be located at this code address + 3 Directive: INTVECTOR(*offset*)

Table 7.5 8051 Compiler Object Settings

Memory Model

Option	Description
Tiny	Selects the Tiny memory model Directive: TINY
Small	Selects the Small memory model Directive: SMALL
Compact	Selects the Compact memory model Directive: COMPACT
Large	Selects the Large memory model Directive: LARGE
Huge	Selects the Huge memory model Directive: HUGE
Use external stack	Uses a simulated external stack, rather than the system stack for a reentrant stack Directive: EXTSTK/(none)
Component with XRAM	Ensures the correct startup code is used for devices with on-chip XRAM. Directive: INTXD/(none)
Advanced features – none	The compiler will use library that use a single DPTR and no arithmetic units Directive: (none)
Advanced features – Philips component with dual DPTR	For certain library functions the compiler will use the Philips dual DPTR scheme Directive: PHILIPSDDPTR
Advanced features – Dual DPTR Dallas/AMD	For certain library functions the compiler will use the Dallas/AMD dual DPTR scheme Directive: MODAMD(DP2)
Advanced features – Dual DPTR Atmel	For certain library functions the compiler will use the Atmel dual DPTR scheme Directive: MODATM
80C517	Allows the compiler to use the additional data pointers and/or arithmetic processor of the 80C517 for certain library function, depending on which 517 options are selected.
Use additional data pointers	For certain library functions the compiler will use the 80C517 multiple DPTR scheme Directive: MOD517(DP8,*otherparam*)/MOD517(NODP8,*otherparam*)

Use the arithmetic processor	For certain library functions the compiler will use the 80C517 arithmetic processor Directive: MOD517(otherparam,AU)/MOD517(otherparam,NOAU)

Table 7.6 8051 Compiler Memory Model Settings

Note

The memory models affect the compiler in several ways. The following table is a summary of the memory models

Small and Large are the preferred memory models to use. The 8051 supports up to 256 bytes of internal RAM so using the Small memory model is not always possible. In those situations the Large memory model should be used.

Memory Model	Default Data Space	Description
Tiny	DATA	Suitable for small applications. Maximum program size is 2k.
Small	DATA	Best memory model for most applications.
Compact	PDATA	Retained for backwards compatibility with preexisting applications.
Large	XDATA	Retained for backwards compatibility with preexisting applications.
Huge	PDATA	A simulated external stack is used for every stack operation giving a larger stack size however execution is slow.

Table 7.7 8051 Compiler Memory Models

Registers

Option	Description
Use absolute register address for R0-R7	Allows the compiler to use absolute register addressing for the register, generating more efficient but registerbank dependant code Directive: AREGS/NOAREGS
Pass function arguments in registers	Allows function arguments to be passed in registers therefore reducing memory requirements Directive: REGPARMS/NOREGPARMS
Registerbank	Selects which registerbank to use for the functions Directive: RB(*banknumber*)

Table 7.8 8051 Compiler Register Settings

Optimizer

Option	Description
Optimize for tight code	The optimizer will favor generating smaller code over faster code Directive: OT(*otherparam*, SIZE)
Optimize for fast code	The optimizer will favor generating faster code over smaller code Directive: OT(*otherparam*, SPEED)
Optimizer level	The level at which the optimizer will operate. The higher the level the more optimizations are performed Directive: OT(*level*, *otherparam*)
Generate post-optimizing information	This option generates information for the global optimizer. Directive: POSTOPT/(none)

Table 7.9 8051 Compiler Optimizer Settings

Messages

Option	Description
Warning level	The compiler warnings are grouped into levels, with the higher levels containing all the warnings in the lower levels. This option indicates which levels of warnings you want the compiler to generate Directive: WL(*level*)
Stop after n errors	The number of errors after which the compiler should abandon compilation Directive: MAXERR(*errornum*)
Stop after n warnings	The number of warnings after which the compiler should abandon compilation Directive: MAXWAR(*warningnum*)

Table 7.10 8051 Compiler Messages Settings

XA Compiler Options Overview

The following Compiler options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Source

Option	Description
ANSI C	Strictly ANSI C only allowed. No language extensions. Directive: NOEXTEND
XA Specific Language Extensions	ANSI C plus language extensions allowed. Directive: EXTEND
‘struct/union/enum’ optional	Allows the struct, union and enum keywords to be omitted when declaring structures, unions and enumerators in certain situations. Directive: SUE_OPT

Table 7.11 XA Compiler Source Settings

Floating Point

Option	Description
No floating point	No floating point variables are allowed in the project. Directive: FP(NOFLOAT)
IEEE: Standard	Allows single-precision floating point numbers (IEEE-754) – little endian Directive: FP(IEEE,STANDARD)

Table 7.12 XA Compiler Floating Point Settings

Code Generation

Option	Description
Enable ANSI Integer Promotion Rules	If selected chars will be promoted to ints before comparison. Directive: IP/NOIP
Initialize static variables to zero	Includes startup code to initialize non-initialized static variables to zero. This is a safeguard that can be used against uninitialized variables. Directive: IS/NOIS
Generic	When selected all non-memory specific pointers are generic pointers. When not selected, non-memory specific pointers point to the default memory space for the current memory model. Directive: GENERIC/NOGENERIC
Unsigned characters	When selected chars are converted to unsigned chars. Directive: UNSIGNEDCHAR/SIGNEDCHAR
Far access allowed	Allows access to far data in page zero mode when using the SmartXA. Directive: FARDATAALLOWED/FARDATANOTALLOWED
Load ES on far data access	When selected the ES register is reloaded before every access to far data. Turn this option off if you wish to set up and maintain the ES register manually. Directive: LOADES/NOLOADES
Load CS on far code access	When selected the CS register is reloaded before every access to far code. Turn this option off if you wish to set up and maintain the CS register manually. Directive: LOADCS/NOLOADCS
Save segment registers in interrupt handlers	When selected the CS and ES registers are saved and restored on the entry and exit of interrupt service routines Directive: SAVESEG/NOSAVESEG

Table 7.13 XA Compiler Code Generation Settings

Note

With the generic option turned on the following pointer declaration results in a pointer that can point to any memory space, however the code to manipulate the pointer is large and involves library calls:

```
      unsigned char *foo;
```

With the generic option turned off the same pointer declaration results in a pointer that points to the default memory space for the current memory model (such as DATA in the Small memory model), however generic pointers can still be declared using the generic keyword, for example

```
      unsigned char generic *foo;
```

This makes turning the generic option off the best setting to use, however be aware of the effects.

Note

If the LOADCS and LOADES directives are being used the Compiler will automatically set up the CS and ES registers before every far access. In addition the relevant bit in the SSEL register will be set and cleared before and after every far access. This results in inefficient code.

There are two steps that can be taken to remove both inefficiencies:

1. Use NOLOADES and NOLOADCS and set up the ES and CS registers manually. This is most suitable when only one far segment is being used.

2. Use SSELINIT to initialize the SSEL register manually. The Compiler will then use only the registers whose corresponding bits in the SSEL register are set for far accesses. The other registers will be used for near accesses. For example:

```
      SSELINIT(0x04)
```

Results in R2 being used for far accesses, and the other registers being used for near accesses.

Defines

This section allows you to enter identifiers that are defined for such things as conditional compilation.
Directive: DEFINE(*text*)

Listing

Option	Description
Generate listing	If selected a .lst listing file will be generated Directive: PR(*filename*)/NOPR
Show lines omitted from compilation	If selected then the source code listing in the listing file will include files that were not involved in the compilation Directive: CO/NOCO
Display the contents of the include files	If selected then the contents of the include files are inserted into the source code listing in the listing file Directive: LC/(none)
Generate Preprocessor listing file	If selected then a separate file will be generated showing macro expansions Directive: PP(*filename*)/(none)
Append assembly mnemonics list	If selected then the compiler generated assembly code will be included in the listing file Directive: CD/NOCD
Generate a list of symbols	If selected then a symbol table will be included in the listing file Directive: SB/NOSB
Insert form feeds at the end of pages	If selected then form feed control characters will be inserted into the listing file Directive: PL(*lines*)/(none)
Number of lines printed per page	Specifies how long a page is Directive: PL(*lines*)
Number of characters printed per line	Specifies how wide a line is Directive: PW(*characters*)

Table 7.14 XA Compiler Listing Settings

Object

Option	Description
Generate an assembler source file	The compiler will generate a .src file which is the complete assembler equivalent of the C source code. The .src file may be assembled by the assembler. Directive: SRC
Generate an object file	The compiler will generate a relocatable object file that make be linked with other object files Directive: OBJECT(*filename*)
Include debugging information	Includes debugging information in the object file Directive: DB/NODB
Include variable type and definition information	Includes extended debugging information in the object file Directive: OE/NOOE

Table 7.15 XA Compiler Object Settings

Memory Model

Option	Description
Tiny	Selects the Tiny memory model Directive: TINY
Small	Selects the Small memory model Directive: SMALL
Compact	Selects the Compact memory model Directive: COMPACT
Medium	Selects the Medium memory model Directive: MEDIUM
Large	Selects the Large memory model Directive: LARGE
Huge	Selects the Huge memory model Directive: HUGE
Functions in system mode	Indicates that the functions will execute in system mode – only of use to SmartXA users when using an RTOS Directive: SYSTEMFCT
Functions in user mode	Indicates that the functions will execute in user mode – only of use to SmartXA users when using an RTOS Directive: USERFCT
Functions in generic mode	Indicates that the functions will execute in a combination of user and system modes – only of use to SmartXA users when using an RTOS Directive: GENERICFCT
Use extended register set	Enables the compiler to generate code using registers R8 to R15. Directive: EXTREGS/NOEXTREGS

Table 7.16 XA Compiler Memory Model Settings

Note

The memory models affect the compiler is several ways. The following table is a summary of the memory models

Memory Model	Default Data Space	Description
Tiny (page 0)	DATA	24-bit addressing is not supported. Maximum 32k of Code space and 32k of Data space
Small (page 0)	DATA	24-bit addressing is not supported. Maximum of 64k of Code space and 64k of Data space
Compact (non page 0)	DATA	All addressing modes are supported.
Medium (non page 0)	IDATA	All addressing modes are supported.
Large (non page 0)	IDATA	All addressing modes are supported. Pointers default to 24-bit addressing.
Huge (non page 0)	-	The Huge memory model is currently not implemented.

Table 7.17 XA Compiler Memory Models

Optimizer

Option	Description
Optimize for tight code	The optimizer will favor generating smaller code over faster code Directive: OT(*otherparam*, SIZE)
Optimize for fast code	The optimizer will favor generating faster code over smaller code Directive: OT(*otherparam*, SPEED)
Optimizer level	The level at which the optimizer will operate. The higher the level the more optimizations are performed Directive: OT(*level*, *otherparam*)

Table 7.18 XA Compiler Optimizer Settings

Messages

Option	Description
Warning level	The compiler warnings are grouped into levels, with the higher levels containing all the warnings in the lower levels. This option indicates which levels of warnings you want the compiler to generate Directive: WL(*level*)
Stop after n errors	The number of errors after which the compiler should abandon compilation Directive: MAXERR(*errornum*)
Stop after n warnings	The number of warnings after which the compiler should abandon compilation Directive: MAXWAR(*warningnum*)

Table 7.19 XA Compiler Messages Settings

ST6 Compiler Options Overview

The following Compiler options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Source

Option	Description
'struct/union/enum' optional	Allows the struct, union and enum keywords to be omitted when declaring structures, unions and enumerators in certain situations. Directive: SUE_OPT

Table 7.20 ST6 Compiler Source Settings

Code Generation

Option	Description
Enable ANSI Integer Promotion Rules	If selected chars will be promoted to ints before comparison. This does not generate optimal code on 8-bit microcontrollers such as the ST6, but is included for ANSI compliance. Directive: IP/NOIP
Initialize static variables to zero	Includes startup code to initialize non-initialized static variables to zero. This is a safeguard that can be used against uninitialized variables. Directive: IS/NOIS
Generic	When selected all non-memory specific pointers are generic pointers. When not selected, non-memory specific pointers point to the default memory space for the current memory model. Directive: GENERIC/NOGENERIC
Unsigned characters	When selected chars are converted to unsigned chars. Directive: UNSIGNEDCHAR/SIGNEDCHAR
Use DRWR copy	Instructs the compiler to make a copy of the DRWR register in DRWRCOPY just before the register contents are changed. Directive: DRWRCOPY/(none)

Table 7.21 ST6 Compiler Code Generation Settings

Note

With the generic option turned on the following pointer declaration results in a pointer that can point to any memory space, however the code to manipulate the pointer is large and involves library calls:

```
      unsigned char *foo;
```

With the generic option turned off the same pointer declaration results in a pointer that points to the default memory space for the current memory model, however generic pointers can still be declared using the generic keyword, for example

```
      unsigned char generic *foo;
```

This makes turning the generic option off the best setting to use, however be aware of the effects.

Defines

This section allows you to enter identifiers that are defined for such things as conditional compilation.
Directive: DEFINE(*text*)

Listing

Option	Description
Generate listing	If selected a .lst listing file will be generated Directive: PR(*filename*)/NOPR
Show lines omitted from compilation	If selected then the source code listing in the listing file will include files that were not involved in the compilation Directive: CO/NOCO
Display the contents of the include files	If selected then the contents of the include files are inserted into the source code listing in the listing file Directive: LC/(none)
Generate Preprocessor listing file	If selected then a separate file will be generated showing macro expansions Directive: PP(*filename*)/(none)
Append assembly mnemonics list	If selected then the compiler generated assembly code will be included in the listing file Directive: CD/NOCD
Generate a list of symbols	If selected then a symbol table will be included in the listing file Directive: SB/NOSB
Insert form feeds at the end of pages	If selected then form feed control characters will be inserted into the listing file Directive: PL(*lines*)/(none)
Number of lines printed per page	Specifies how long a page is Directive: PL(*lines*)
Number of characters printed per line	Specifies how wide a line is Directive: PW(*characters*)

Table 7.22 ST6 Compiler Listing Settings

Object

Option	Description
Generate an assembler source file	The compiler will generate a .src file which is the complete assembler equivalent of the C source code. The .src file may be assembled by the assembler. Directive: SRC
Generate an object file	The compiler will generate a relocatable object file that make be linked with other object files Directive: OBJECT(*filename*)
Include debugging information	Includes debugging information in the object file Directive: DB/NODB

Table 7.23 ST6 Compiler Object Settings

Memory Model

Option	Description
Small	Selects the Small memory model Directive: SMALL
Large	Selects the Large memory model Directive: LARGE

Table 7.24 ST6 Compiler Memory Model Settings

Note

The memory models affect the compiler is several ways. The following table is a summary of the memory models

Memory Model	Description
Small	No bank switching. Maximum program size of 4k. Maximum 128 bytes of DATA.
Large	Bank switching supported. Maximum program size of 8k. Maximum 512 bytes of RAM + EEPROM.

Table 7.25 ST6 Compiler Memory Models

Optimizer

Option	Description
Optimize for tight code	The optimizer will favor generating smaller code over faster code Directive: OT(*otherparam*, SIZE)
Optimize for fast code	The optimizer will favor generating faster code over smaller code Directive: OT(*otherparam*, SPEED)
Optimizer level	The level at which the optimizer will operate. The higher the level the more optimizations are performed Directive: OT(*level*, *otherparam*)

Table 7.26 ST6 Compiler Optimizer Settings

Messages

Option	Description
Warning level	The compiler warnings are grouped into levels, with the higher levels containing all the warnings in the lower levels. This option indicates which levels of warnings you want the compiler to generate Directive: WL(*level*)
Stop after n errors	The number of errors after which the compiler should abandon compilation Directive: MAXERR(*errornum*)
Stop after n warnings	The number of warnings after which the compiler should abandon compilation Directive: MAXWAR(*warningnum*)

Table 7.27 ST6 Compiler Messages Settings

Compiler Command Line Syntax

All three compilers have the same command line syntax:

toolexename sourcefile [*directiveslist*]

toolexename:

> one of: RC51, RCXA, RCST6

sourcefile:

> an absolute or relative path to a C source file

directiveslist:

> a space separated list of directives. The directives may be listed in any order.

If each directive is not explicitly listed then defaults will be used for the missing directives.

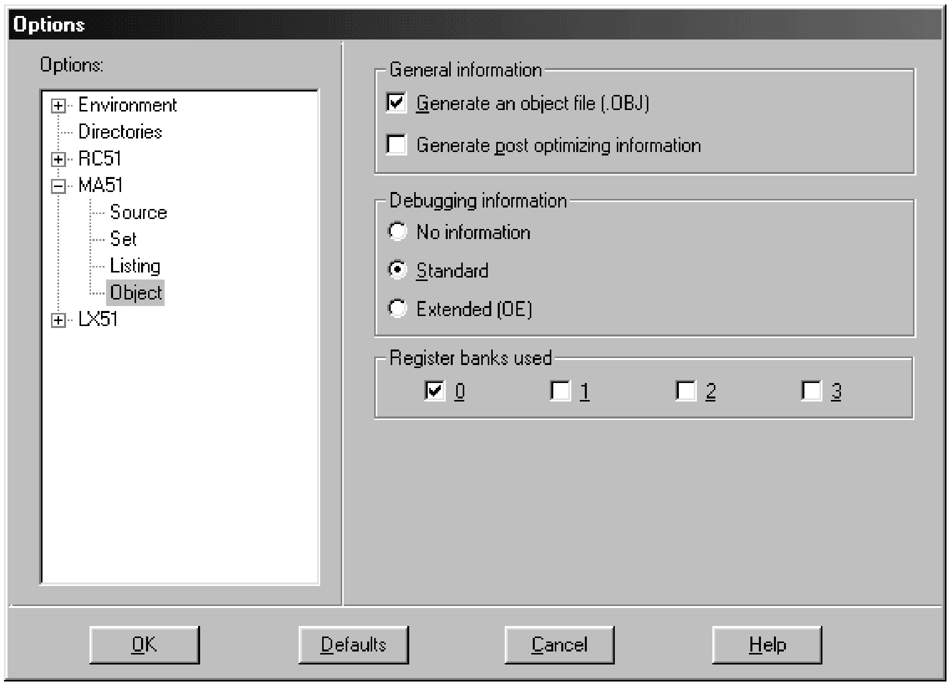
Command line examples:

```
RC51 test.c PR(test.lst) OBJECT(test.obj) CD SB

RCXA C:\work\xa.c USERFCT EXTSTK SAVESEG

RCST6 ..\foo.c DRWRCOPY

RC51 bar.c
```

Chapter 8. Assembler

Changing the Assembler Settings in RIDE

With a project loaded into RIDE the Assembler options for the project as a whole may accessed by choosing Options | Project then expanding the tree for the Assembler and clicking on the various sections. The following screenshot is for the 8051:



General information
- [✓] Generate an object file (.OBJ)
- [ ] Generate post optimizing information

Debugging information
- ( ) No information
- (●) Standard
- ( ) Extended (OE)

Register banks used
- [✓] 0
- [ ] 1
- [ ] 2
- [ ] 3

Once all the options have been set up accordingly click on OK to confirm them.

The Assembler options may also be set individually for each source file in the project:

- In the Project window select the source file whose options you wish to change
- Press the right mouse button.
- From the pop-up menu choose Options | Local Options. A window will open allowing you to change the assembler options for that source file.

8051 Assembler Options Overview

The following Assembler options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Source

Option	Description
Define symbols for the 8051 function registers	When selected a standard 8051 set of Special Function Registers will be defined, thus avoiding having to define them in the assembler file Directive: MOD51/NOMOD51
Accept Intel MPL	When selected the assembler will accept the Intel Macro programming language. Directive: MACRO(MPL)/(none)
ASM-51 Syntax	When selected the assembler will accept the Intel ASM-51 assembler syntax. Directive: SYNTAX(ASM51)
Raisonance Syntax	When selected the assembler will accept the syntax of older Raisonance assemblers, such as EMA-51. Directive: SYNTAX(EMA)

Table 8.1 8051 Assembler Source Settings

Set

By entering text with the format:

`symbol [ = value] [, symbol [ = value]]`

values can be assigned to symbols. If no value is specified then the symbol is assigned the value 0xFFFF.

Directive: SET(*text*)

Listing

Option	Description
Generate Listing	When selected the assembler will generate a .lst listing file Directive: PRINT(*filename*)/NOPRINT
Include the program source text	When selected the assembler source code will be included in the listing file Directive: LIST/NOLIST
Display the contents of the include files	When selected the contents of include files will be inserted into the source code listing in the listing file Directive: LC/(none)
Show unassembled lines of conditional constructs	Lines that were not assembled will be included in the source code listing when this option is selected Directive: COND/NOCOND
Expand assembly instructions of macros	When selected assembly code inside macro definitions will appear in the listing file Directive: GEN/NOGEN
Generate a table of the symbols	When selected a symbol table will be included in the listing file Directive: SB/NOSB
Insert form feeds at the end of pages	When selected form feed control characters will be inserted into the listing file at the end of every page Directive: EJECT/(none)
Generate a cross reference table of the symbols	Includes a cross reference table of all the symbols in the listing file Directive: XR/NOXR
Number of characters printed per line	Specifies the page width of the listing file in characters Directive: PW(*characters*)
Number of lines printed per page	Specifies the page length used in the listing file in lines Directive: PL(*lines*)

Table 8.2 8051 Assembler Listing Settings

Object

Option	Description
Generate an object file	When selected the assembler will generate a relocatable object file Directive: OBJECT(*filename*)/NOOBJECT
Generate post optimizing information	When selected the assembler will generate information used in global optimization Directive: POSTOPT/(none)
Debugging information – no information	Selection of this option will omit debugging information from the object file Directive: (none)
Debugging information – standard	This option includes basic debugging information in the object file Directive: DB
Debugging information – extended	This option includes additional debugging information in the object file Directive: OE
Register banks used	This section of the options allow you to select which register banks are to be reserved by the assembler. Directive: RB(*registerbank*[,*registerbank*])

Table 8.3 8051 Assembler Object Settings

XA Assembler Options Overview

The following Assembler options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Source

Option	Description
Define symbols for the XA function registers	When selected a standard XA set of Special Function Registers will be defined, thus avoiding having to define them in the assembler file Directive: MODXA/NOMO
Accept Intel MPL	When selected the assembler will accept the Intel Macro programming language. Directive: MACRO(MPL)/(none)
Always set code labels on even addresses	When selected the assembler will ensure that code labels are word-aligned. Directive: ECL/NOECL
Use extended register set R8 – R15	Allows the use of registers R8 – R15 when selected. Directive: EXTREGS/NOEXTREGS

Table 8.4 XA Assembler Source Settings

Set

By entering text with the format:

symbol [= *value*] [, *symbol* [= *value*]]

values can be assigned to symbols. If no value is specified then the symbol is assigned the value 0xFFFF.

Directive: SET(*text*)

Listing

Option	Description
Generate Listing	When selected the assembler will generate a .lst listing file Directive: PRINT(*filename*)/NOPRINT
Include the program source text	When selected the assembler source code will be included in the listing file Directive: LIST/NOLIST
Display the contents of the include files	When selected the contents of include files will be inserted into the source code listing in the listing file Directive: LC/(none)
Show unassembled lines of conditional constructs	Lines that were not assembled will be included in the source code listing when this option is selected Directive: COND/NOCOND
Expand assembly instructions of macros	When selected assembly code inside macro definitions will appear in the listing file Directive: GEN/NOGEN
Generate a table of the symbols	When selected a symbol table will be included in the listing file Directive: SB/NOSB
Insert form feeds at the end of pages	When selected form feed control characters will be inserted into the listing file at the end of every page Directive: EJECT/(none)
Generate a cross reference table of the symbols	Includes a cross reference table of all the symbols in the listing file Directive: XR/NOXR
Number of characters printed per line	Specifies the page width of the listing file in characters Directive: PW(*characters*)
Number of lines printed per page	Specifies the page length used in the listing file in lines Directive: PL(*lines*)

Table 8.5 XA Assembler Listing Settings

Object

Option	Description
Generate an object file	When selected the assembler will generate a relocatable object file Directive: OBJECT(*filename*)/NOOBJECT
Include Debugging information	This option includes debugging information in the object file Directive: DB/NODB
Register banks used	This option is not used by the assembler and is only included for compatibility with the 8051 toolset.

Table 8.6 XA Assembler Object Settings

ST6 Assembler Options Overview

The following Assembler options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Source

Option	Description
Define symbols for the ST6 function registers	When selected a standard ST6 set of Special Function Registers will be defined, thus avoiding having to define them in the assembler file Directive: MODST6/NOMO
Use AST6 syntax	When selected the assembler will accept files written for the ST Microelectronics AST6 assembler. Directive: PREPROST/(none)

Table 8.7 ST6 Assembler Source Settings

Set

By entering text with the format:

symbol [= *value*] [, *symbol* [= *value*]]

values can be assigned to symbols. If no value is specified then the symbol is assigned the value 0xFFFF.

Directive: SET(*text*)

Listing

Option	Description
Generate Listing	When selected the assembler will generate a .lst listing file Directive: PRINT(*filename*)/NOPRINT
Include the program source text	When selected the assembler source code will be included in the listing file Directive: LIST/NOLIST
Display the contents of the include files	When selected the contents of include files will be inserted into the source code listing in the listing file Directive: LC/(none)
Show unassembled lines of conditional constructs	Lines that were not assembled will be included in the source code listing when this option is selected Directive: COND/NOCOND
Expand assembly instructions of macros	When selected assembly code inside macro definitions will appear in the listing file Directive: GEN/NOGEN
Generate a table of the symbols	When selected a symbol table will be included in the listing file Directive: SB/NOSB
Insert form feeds at the end of pages	When selected form feed control characters will be inserted into the listing file at the end of every page Directive: EJECT/(none)
Generate a cross reference table of the symbols	Includes a cross reference table of all the symbols in the listing file Directive: XR/NOXR
Number of characters printed per line	Specifies the page width of the listing file in characters Directive: PW(*characters*)
Number of lines printed per page	Specifies the page length used in the listing file in lines Directive: PL(*lines*)

Table 8.8 ST6 Assembler Listing Settings

Object

Option	Description
Generate an object file	When selected the assembler will generate a relocatable object file Directive: OBJECT(*filename*)/NOOBJECT
Include Debug information	This option includes basic debugging information in the object file Directive: DB/NODB
ROM fill value	The ROM area not used by code or constants is filled with the ROM fill value. This is useful in ensuring the ROM area has a particular checksum. Directive: ROMFILL(*value*)

Table 8.9 ST6 Assembler Object Settings

Assembler Command Line Syntax

All three assemblers have the same command line syntax:

toolexename sourcefile [*directiveslist*]

toolexename:

one of: MA51, MAXA, MAST6

sourcefile:

an absolute or relative path to an assembler source file

directiveslist:

a space separated list of directives. The directives may be listed in any order.

If each directive is not explicitly listed then defaults will be used for the missing directives.

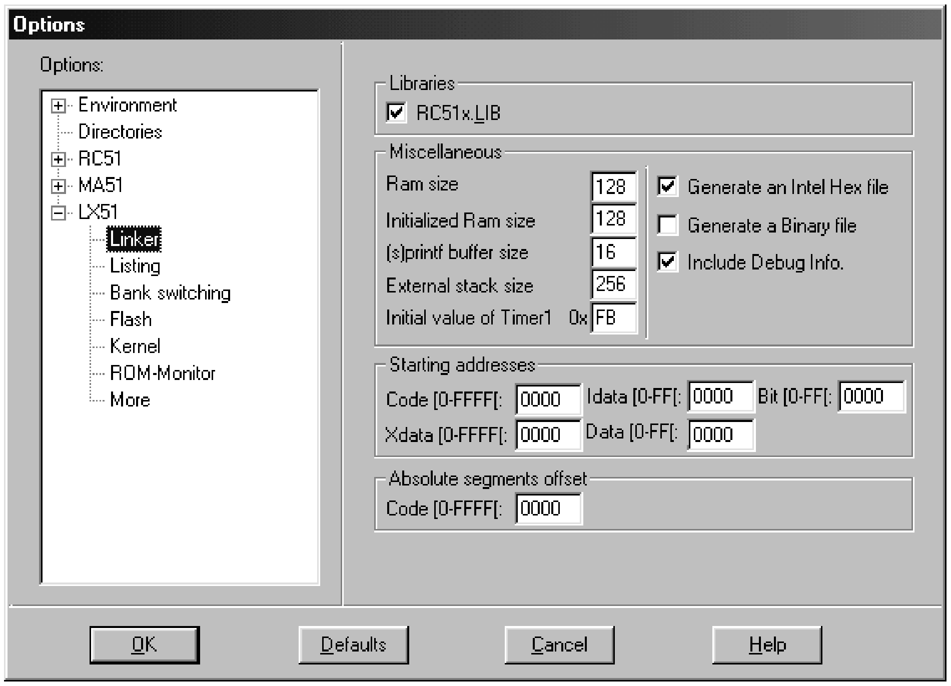
Command line examples:

```
MA51 test.a51 PR(test.lst) OBJECT(test.obj) SB

MAXA C:\work\xa.axa NOMO XR

RCST6 ..\foo.st6 ROMFILL(0xFF)

MA51 bar.a51
```

Chapter 9. Linker

Changing the Linker Settings in RIDE

With a project loaded into RIDE the Linker options for the project as a whole may be accessed by choosing Options | Project then expanding the tree for the Linker and clicking on the various sections. The following screenshot is for the 8051:



Libraries
- [✓] RC51x.LIB

Miscellaneous
- Ram size: 128
- Initialized Ram size: 128
- (s)printf buffer size: 16
- External stack size: 256
- Initial value of Timer1: 0x FB
- [✓] Generate an Intel Hex file
- [ ] Generate a Binary file
- [✓] Include Debug Info.

Starting addresses
- Code [0-FFFF[: 0000
- Idata [0-FF[: 0000
- Bit [0-FF[: 0000
- Xdata [0-FFFF[: 0000
- Data [0-FF[: 0000

Absolute segments offset
- Code [0-FFFF[: 0000

Once all the options have been set up accordingly click on OK to confirm them.

8051 Linker Options Overview

The following Linker options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Linker

Option	Description
Libraries – RC51x.LIB	When selected the libraries supplied with the toolset will be used. Directive: (none)/NLIB
Ram size	Specifies the amount of internal RAM on the device Directive: RS(*size*)
Initialized Ram size	Specifies the amount of internal RAM to initialize to 0 Directive: RSI(*size*)
(s)printf buffer size	Specifies the buffer size used to construct strings in printf and sprintf
External stack size	When using an external reentrant stack this options specifies the size of the stack.
Initial value of timer 1	In the startup code timer 1 is initialized to be used as a baud rate generator. This option allows the initial value to be changed therefore changing the baud rate.
Generate an Intel Hex file	When selected an Intel Hex file will be generated.
Generate a binary file	When selected a raw binary file will be generated.
Include debug info.	Debugging information will be included in the absolute object file when this option is selected. Directive: DL + DP + DS/NODL + NODP + NODS
Starting addresses – Code	Specifies the starting address for Code segments. Note that it does not specify a starting address for the reset vector or interrupt vectors. Directive: CODE(*address*)
Starting addresses - Xdata	Specifies the starting address for the external RAM (Xdata) segments. Directive: XDATA(*address*)
Starting addresses - Idata	Specifies the starting address of the indirect internal RAM (Idata) segments.

	Directive: IDATA(*address*)
Starting addresses - Data	Specifies the starting address of the direct internal RAM (Data) segments. Note that it does not affect the location of the registerbanks. Directive: DATA(*address*)
Starting addresses - Bit	Specifies the starting address of the internal bit-addressable area (Bdata). Directive: BDATA(*address*)
Absolute segments offset - code	Specifies an offset to apply to absolute code segments Directive: ABSCODEOFFS(*offset*)

Table 9.1 8051 Linker Settings

Note

Sometimes it is undesirable to have the project code starting at 0x0000 and another address is required. For example if two separate projects must be loaded into the same ROM at the same time.

To relocate all code to a specific address two steps must be performed:

1. Specify the address in the Starting addresses – Code box
2. Specify the address in the Absolute segments offset – code box

Always remember to check the link map of module table in the map file to verify that the memory map of your project is correct.

Listing

Option	Description
Include the cross references table XREF	When selected a cross reference table will be included in the listing file. Directive: IX/NOIX
Insert form feeds at the end of pages	When selected form feed control characters will be inserted into the listing file. Directive: EJECT/(none)
Number of lines printed per page	Specifies the page length in lines. Directive: PL(*lines*)
Number of characters printed per line	Specifies the page width in characters. Directive: PW(*lines*)

Table 9.2 8051 Linker Listing Settings

Bank Switching

Option	Description
Use bank switching mode	When selected the linker will use bank switching to provide more than 64k of code space Directive: BANKAREA(*start*, *end*)
Maximum number of banks	The maximum number of code banks used in the project (must be a power of 2)
Starting address of the code banking	The base address of the code banks Directive: BANKAREA(*address*, *otherparam*)
Ending address of the code banking	The top address of the code banks Directive: BANKAREA(*otherparam*, *address*)
Use external stack	When selected allows functions to be reentrant. Directive: BM(SMA)/BM(PLM)
Use the macro definition	The macro in the box is used to switch code banks. A custom macro may be entered into the box.
Evaluate the expression for the currently selected bank	The symbol that represents the currently selected code bank, and is used in the macro definition.
Modules Tab	The section under the Modules tab may be used to select which code banks to place various object files and libraries into, by double-clicking on each one. Note that object files are only listed if the project has previously been built. Directive: BANK*banknumber*{*objectfile*}

Table 9.3 8051 Linker Bank Switching Settings

Note
Code Banking is an involved and complex area of the linker. Please refer to the Linker manual for full and detailed information on Code Banking.

Flash

Option	Description
Use Flash mode/Start Address	Activate the Flash mode. Specify the border address between the ROM part (lowest bound) and the Flash part (highest bound) Directive: FLASH(*address*)
FLS file – locked mode	Relocate/Link modules of the Flash Part, and keep intact the ROM part. Directive: REFLASH
Reserved data space	Reserve a gap of *n* bytes for future DATA storage needs. Directive: RESERVE(DATA, *n*)
Reserved bit space	Reserve a gap of *n* bytes for future BIT storage needs. Directive: RESERVE(BIT, *n*)
Reserved Bdata space	Reserve a gap of *n* bytes for future BDATA storage needs. Directive: RESERVE(BDATA, *n*)
Object files to be in Flash	Select the modules to be located in the Flash. Directive: FLASH(*otherparam*, *objectfilelist*)

Table 9.4 8051 Linker Flash Settings

Kernel

Option	Description
Use KR-51 Kernel	When selected the KR-51 RTOS will be used
Kernel Model – KR-Tiny	Selects the Tiny model – up to 8 tasks
Kernel Model - KR-Standard	Selects the Standard model – up to 32 tasks with Xdata required.
Kernel Model - KR-Huge	Selects the Huge model – up to 256 tasks with Xdata required.
Debug	When selected the extended debug libraries will be used.
Semaphores	Select this option if semaphores are used.
Time – use the automatic definitions	When selected the clock and dividers settings in the window are used, otherwise the same settings must be provided in an assembly file.
CPU cycles/tick	Specifies the relationship between the CPU clock cycles and the RTOS tick rate.
Dividers – Group 1	The number of ticks in a single group 1 tick.
Dividers – Group 2	The number of group 1 ticks in a single group 2 tick
Dividers – Group 3	The numberof group 2 ticks in a single group 3 tick

Table 9.5 8051 Linker Kernel Settings

ROM-Monitor

Option	Description
Use the ROM-Monitor	When selected the ROM Monitor will be used to provide in-system debugging.
Communications - Standard UART	Select to use the first internal UART of the microcontroller for the monitor to communicate with RIDE
Communications - External UART	Select to use an external UART for the monitor to communicate with RIDE
Communications - ROM-Monitor	Select to use a customized method for the monitor to communicate with RIDE
Communications – Communication baud rate	Specify the baud rate of the communications between RIDE and the monitor
Dynamically modifiable code	Select if the code is located in RAM where it may be modified. This is required if the use of breakpoints is desired.
XEVA board	Select if the board is a Raisonance XEVA board
Von Neuman board	Select if the board is a Von Neuman board (code space is mapped to external RAM).

Table 9.6 8051 Linker ROM-Monitor Settings

More

Linker directives may be entered into the More box in the form of a space separated list.
Note however that you should not specify a directive that will already be specified by selection or non-selection of a linker option in one of the sections previously described.

XA Linker Options Overview

The following Linker options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Linker

Option	Description
Libraries – RCXAx.LIB	When selected the libraries supplied with the toolset will be used. Directive: (none)/NLIB
BTR initialization needed	This option should be selected if the Bus Timing Register needs to be initialized before execution.
BTR init value	The value to initialize the Bus Timing Register to. Directive: BTRINIT(*value*)
Ram size	Specifies the amount of internal RAM on the device Directive: RS(*size*)
Initialized Ram size	Specifies the amount of internal RAM to initialize to 0 Directive: RSI(*size*)
User stack size	Specifies the size of the user stack Directive: USS(*size*)/NOUSS
System stack size	Specifies the size of the system stack Directive: SSS(*size*)/NOSSS
Buffer size for printf	Specifies the buffer size used to construct strings by printf and sprintf
Initial value of timer 1	In the startup code timer 1 is initialized to be used as a baud rate generated. This option allows the initial value to be changed therefore changing the baud rate.
Generate an Intel Hex file	When selected an Intel Hex file will be generated.
Generate a binary file	When selected a raw binary file will be generated.
Include debug info.	Debugging information will be included in the absolute object file when this option is selected. Directive: DL + DP + DS/NODL + NODP + NODS
Generate crossref table	When selected a cross-reference table is included in the

file	map file. Directive: IX/NOIX
Generate an ABS file	When selected an ABS format file compatible with some emulators will be generated.
Manage stacks	Enables the stack overflows and the system stack location to be controlled.
System overflow	Specifies the number of bytes to reserve below the system stack overflow point. Directive: SSTKOV(*bytes*)
User overflow	Specifies the number of bytes to reserve below the user stack overflow point. Directive: USTKOV(*bytes*)
System location	Specifies the location of the system stack. Directive: SSTACK(*address*)
Listing – lines/page	Specifies the page length of the map file in lines Directive: PL(*lines*)
Listing – characters/line	Specifies the page width of the map file in characters Directive: PW(*characters*)

Table 9.7 XA Linker Settings

Relocation

Option	Description
Near code relocation – base address	Specifies the starting address for near code segments Directive: CO(*address*)
Far code relocation – base address	Specifies the starting address for far code segments Directive: FARCODE(*address*)
Near data relocation – Near data/idata segment	Specifies the segment to located the near data and idata into. Directive: NDTSEG(*segment*)
Near data relocation - Idata base address	Specifies the starting address for near idata segments Directive: NID(*address*)
Near data relocation – data base address	Specifies the starting address for near data segments Directive: NDT(*address*)
Far data relocation – Idata base address	Specifies the starting address for far idata segments Directive: FID(*address*)
Far data relocation – Data base address	Specifies the starting address for far data segments Directive: FDT(*address*)

Table 9.8 XA Linker Relocation Settings

Kernel

Option	Description
Use KRXA Kernel	When selected the KR-XA RTOS will be used
Debug	When selected the extended debug libraries will be used.
Semaphores	Select this option if semaphores are used.
Use the automatic definitions	When selected the clock and dividers settings in the window are used, otherwise the same settings must be provided in an assembly file.
CPU cycles/tick	Specifies the relationship between the CPU clock cycles and the RTOS tick rate.
Dividers – group 1	The number of ticks in a single group 1 tick.
Dividers – group 2	The number of group 1 ticks in a single group 2 tick
Dividers – group 3	The numberof group 2 ticks in a single group 3 tick

Table 9.9 XA Linker Kernel Settings

ROM-Monitor

Option	Description
Use the ROM-Monitor	When selected the ROM Monitor will be used to provide in-system debugging.
Communications - Standard UART 0	Select to use the first internal UART of the microcontroller for the monitor to communicate with RIDE
Communications – Standard UART 1	Select to use the second internal UART of the microcontroller for the monitor to communicate with RIDE
Communications – External UART	Select to use an external UART for the monitor to communicate with RIDE
Communications – Communication baud rate	Specify the baud rate of the communications between RIDE and the monitor
Dynamically modifiable code	Select if the code is located in RAM where it may be modified. This is required if the use of breakpoints is desired.
XEVA board	Select if the board is a Raisonance XEVA board
Von Neuman board	Select if the board is a Von Neuman board (code space is mapped to external RAM).

Table 9.10 XA Linker ROM-Monitor Settings

More

Linker directives may be entered into the More box in the form of a space separated list.
Note however that you should not specify a directive that will already be specified by selection or non-selection of a linker option in one of the sections previously described.

ST6 Linker Options Overview

The following Linker options are grouped by section, as listed in the Options window.

Where there are two directives for an option, the first directive is the one used with the option selected. The second directive is the one used when the option is not selected. In some cases the lack of a directive selects the default setting, which is indicated by (none).

Linker

Option	Description
Generate a Hex file	When selected an Intel Hex file will be generated
Generate a binary file	When selected a raw binary file will be generated
Include debug information	Select to include debugging information in the absolute object file Directive: DL + DP + DS/NODL + NODP + NODS
ROM – Fill unused areas	When selected enter a value to fill the unused areas of ROM with. Directive: ROMFILLUNUSEDVALUE(*value*)
ROM – Fill reserved areas	When selected enter a value to fill the reserved areas of ROM with. Directive: ROMFILLRESERVEDVALUE(*value*)
Use RCST6x.LIB library files	Select to use the libraries that are supplied with the toolset. Directive: (none)/NLIB
Initialized static RAM size	Number of bytes to initialize in the static RAM Directive: RAMSIZEINIT(*bytes*)
Printf argument max size	Maximum printf argument size in bytes Directive: PRSTATICSIZE(*bytes*)

Table 9.11 ST6 Linker Settings

Bank Switching

This section allows the code bank for specific object and library files to be selected. Double-click on a file to select the code bank.

More

Linker directives may be entered into the More box in the form of a space separated list.
Note however that you should not specify a directive that will already be specified by selection or non-selection of a linker option in one of the sections previously described.

Listing

Option	Description
Include the cross reference table XREF	When selected a cross reference table will be included in the map file. Directive: IX/NOIX
Insert form feeds at the end of pages	When selected form feed control characters will be inserted into the map file. Directive: EJECT/(none)
Number of lines printed per page	Specifies the page length of the map file in lines Directive: PL(lines)
Number of characters printed per line	Specifies the page width of the map file in characters. Directive: PW(characters)
Print call tree	When selected includes the call tree in the map file. Directive: CALLTREE/NOCALLTREE
Print module mapping	When selected includes the module mapping information in the map file Directive: MODULEMAP/NOMODULEMAP

Table 9.12 ST6 Linker Listing Settings

Linker Command Line Syntax

All three linkers have the same command line syntax:

toolexename objectfilelist [*directiveslist*]

toolexename:

> one of: LX51, RLXA, RLST6

objectfilelist:

> a comma separated list of object files and library files to be linked together

directiveslist:

> a space separated list of directives. The directives may be listed in any order.

If each directive is not explicitly listed then defaults will be used for the missing directives.

Command line examples:

```
LX51 test.obj, foo.lib TO(test.aof) RS(128) IX

RLXA c:\work\bar.obj SSS(256) TO(bar.aof)

RLST6 ..\baz.obj, test.obj CODESTART(80) TO(baz.aof)
```

Glossary

8051 – an 8-bit microcontroller family which is the world's most popular and features the most derivatives. Many different silicon vendors make 8051s.

Absolute Object File – an object file generated by the linker containing data to be stored in a target's ROM and RAM. Addresses are specified for all the target's data in the file. The file may also contain debugging information.

Assembler – a program which takes a source file containing a textual representation of assembly code and converts it into a binary form stored in a relocatable object file. The assembler processes symbols converting them to addresses to be fixed and performs macro processing.

Breakpoint – a code address at which execution must stop once the microcontroller reaches it.

Build process – the process involving the compilation and/or assembly of source files, followed by the linking of generated object files, and optionally followed by the processing of the linker generated absolute object file to convert it into other file formats.

Compiler – a program which takes a source file containing a C program and converts it into a binary form stored in a relocatable object file.

Debugger – software which enables high-level as well as low-level debugging to be performed, including debugging of software running on target hardware.

Execution point – the address at which the next instruction will be executed, i.e. the location where the Program Counter points to.

Intel Hex File – an ASCII file format that represents binary data stored at specific addresses.

Interrupt Service Routine – the function which is executed when a particular interrupt is generated, providing the interrupt has been correctly enabled.

I/O – input/output – for example pins on a microcontroller which allow signals to be generated or read.

Language extension – an extension to the ANSI C programming language which allows features specific to a microcontroller to be used. Language extensions take the form of new keywords and new operator syntax.

Library – a set of relocatable object files combined into a single file called a library. The library may then be linked with other relocatable object files producing the same result as if each relocatable object file in the library had been linked individually.

Library Manager – a program that allows the creation of library and the addition and removal of relocatable object files from the library.

Linker – a program that takes a set of relocatable object files and combines them into a single absolute object file. All relocatable code and data is located at specific addresses. All symbols are resolved to specific addresses.

Listing file – the text files generated by the assembler and compiler which detail what the assembler and compiler did when they processed a source file and the result of the processing.

Map file – the text file generated by the linker which details what the linker did when it processed the input files and the result of the processing.

Memory model – the memory model configures the Compiler to operate in a certain way by selecting the default memory space and the addressing modes allowed.

Microcontroller – a single-chip computer. Integrated onto one chip is a microprocessor, RAM, peripherals, such as UART, I/O ports, timers, CAN controllers, etc, usually ROM/EPROM/Flash/OTP ROM, sometimes EEPROM.

Microcontroller family – a collection of microcontrollers that feature the same instruction set, memory areas, and other core features.

Module – the code and data generated by the assembling or compiling of a single source file.

Monitor – a program that runs on the target system along with a user program and reports back debugging information.

Peripheral – a unit integrated onto a microcontroller chip with a specific function, not considered part of the core functions. For example UART, timer, CAN controller, I/O port, PCA, watchdog, I2C

Program Counter – a register which contains the address of the next instruction to be executed.

Project – a description of all the information necessary to create an absolute object file and possibly an Intel Hex File. This includes the source files required, any libraries required, a list of tools required in the build process, how to execute the tools.

Project File – a file containing all the project information.

Relocatable object file – a file containing code and data generated by the processing of a source file. However some symbols may not be fixed to specific addresses.

RIDE – Raisonance Integrated Development Environment. RIDE functions as an editor, project manager, make utility and simulator/debugger, featuring a menu driven and toolbar driven user interface.

Segment – when a source file is processed it is broken up into segments. There is a segment for each function and a segment for each of the memory areas that have static relocatable variables in the source file.

Simulator – a program that takes code for a microcontroller and executes it in exactly the same way the microcontroller would execute it – allowing detailed analysis of what would happen if the code was executed in the microcontroller.

Source file – a text file that contains either a textual representation of assembly code or a C program.

Special Function Register – a register in a microcontroller that allows control of features of the microcontroller.

ST6 – an 8-bit microcontroller family manufactured by ST-Microelectronics.

Startup code – the section of assembler code that executes before the main function is reached. Usually the startup code is automatically inserted by the linker, however it can be modified.

Symbol table – a table listing each symbol and the address it is stored at.

Timer – a peripheral that can count either up or down or count pulses on a pin.

XA – a 16-bit microcontroller family manufactured by Philips Semiconductors.

Index

Notes

Notes

Notes

Notes

Notes