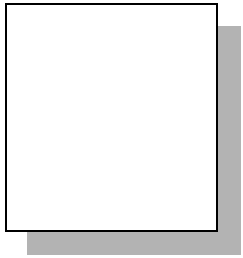
Version 1.5

C-TestIt! User's Guide for Freescale HC08/HCS08

Table of Contents



Introduction

What is C-TestIt!

C-TestIt! is a product that allows users to "***unit test***" their C code. "Unit testing" in the case of C is understood as the ability to test a function with a number of different sets of parameters and then check the results; this is usually referred to as **Black Box Testing**. **C-TestIt!** also allows users to specify a number of "assertions"; "assertions" are conditional statements that will be checked by **C-TestIt!** during execution of the function under test, this is referred to as **Gray Box Testing**.

C-TestIt! offers a unique approach for unit testing, in that it actually allows you to test functions in the very body of your own application, since the tests are done not on the C source but on the actual executable file that is your application. This is called "*In Application Unit Testing*". This approach guarantees that the function is working in its final environment, and it relieves the sometimes tedious steps of having to recompile and link the tested function with added code that will implement the test. In the case of **C-TestIt!**, no code is added to your function, no compilation or link is necessary, **C-TestIt!** directly uses your own application.

C-TestIt! also supports the traditonnal approach of "*Source Unit Testing*". In this case the test consists in generating C code that positions the arguments and more C code that tests the results. This code is then com-

piled and linked and executed and **C-TestIt!** then reports the results of the tests. In this case, a compiler and linker must be readily available to the user in order to perform all the steps.

By incorporating these two methods for unit testing **C-TestIt!** offers the user a comprehensive testing tool.

C-TestIt! Main features

- **In Application Unit Testing**
  - The code under test is the exact code you will later put into your target,
  - works directly on the application code: no need to recompile or re-link
  - Test with the same memory model and compiler options as the real project The real code is tested so all options and parameters of the code are used
  - Test the function at its real location. Since the test is done using the real application code, functions are thus tested at their real location thereby relieving the issues of function location, bank switching, etc.
- **Source Unit Testing.**
  Source code of the function is analyzed, then C source instrumented code is generated to accomplish the test. This code is then compiled, linked and executed.
- **Can check every function of the project**
  All functions (compiled in debug mode for "*In Application Testing*") can be checked with a single file load.
- **Gray Box Testing**
  The user can specify a number of conditional statements that will be evaluated during function execution.
- **Defines input parameters and expected results**
  All parameters of the function under test can be specified using C expressions.

- **Defines values for global variables**
  Global variables can be defined for every test using C expressions.

- **Inputs (arguments, globals)**
  Can be specified as a single value, or as a range or a set of values thereby allowing the same function to be tested with different inputs in the same test session.

- **Create a suite of tests (a "*Testorama*")**
  to be run together

- **Supports simulator and real hardware (BDM, ICE, JTAG or Emulators)**
  C-TestIt! offers several variants for executing the code under test.

- **Supports a large number of targets**

- **Runs interactively or in batch mode**
  Tests can be run immediately and results visualized graphically, or they can be run in batch mode with logging of output results.

- **Can produce reports for archive**

- **Can produce additional information such as Code Coverage and execution timing**

CHAPTER

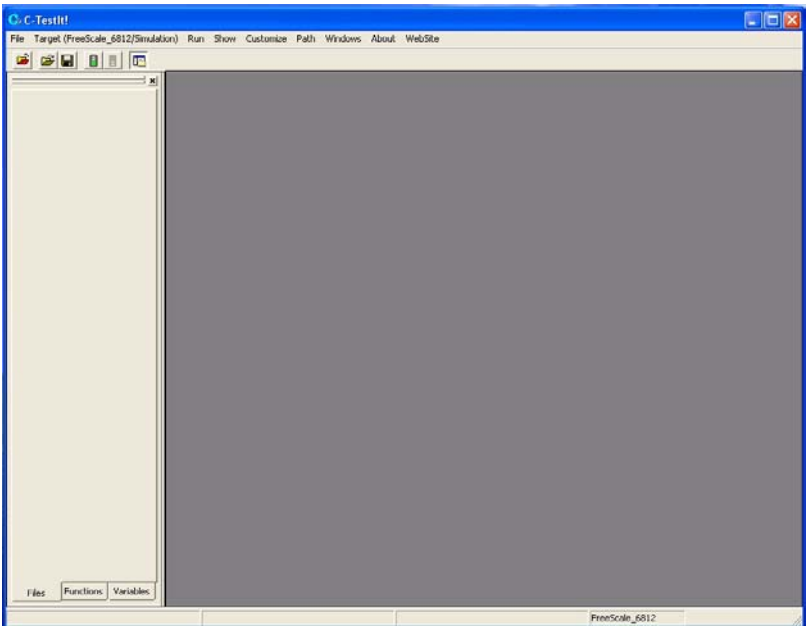
1

Starting C-TestIt!

- C-TestIt! Display Description
- Specifying the Target Processor and the Execution Engine
- Setting up the execution environment

C-TestIt! Display Description

Once you start **C-TestIt!** the screen should look like:



The main window is composed of:

- The application pane on the left which shows all the components of the executable file for which tests will be built/run.
- The test window where tests will be displayed, as well as source files if necessary.

Specifying the Target Processor and the Execution Engine

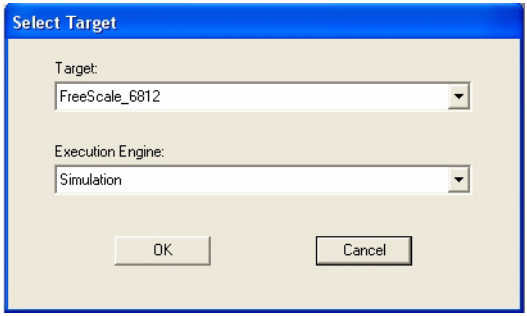
C-TestIt! supports various targets and execution environments.

So once you have started C-TestIt!, you need to specify for which target processor and which execution environment you are going to specify the tests.

Please note that the target specification becomes part of the test definition, while the execution environment is not; *i.e.* a test for a specific target can be used with any execution environment available for that specific target.

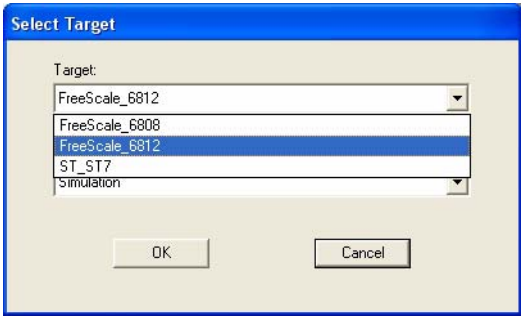
The menu shows what is the current target and environment. In the above example, the Freescale HC12 is selected and the simulation execution environment is also selected. To change these selections, use the menu entry.

You will then get the following dialog that will allow you to make your own selection:

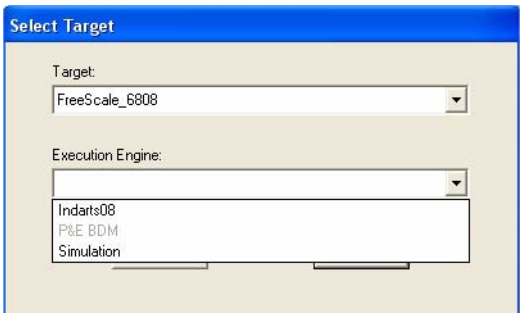


Please note that you select a target family and not a specific derivative. The derivative selection can be made when running the tests and setting up the execution environment.

You can then select the target family, the screen will look like:



And you can select the execution environment, the screen will look like:



Setting up the execution environment

For some execution engines it may be necessary to setup the target, for example it may be necessary to specify which derivative is used, or which port is used to connect to the execution engine. This is not always needed but may be necessary for example when the target processor is changed from one derivative to another.

It is possible to force a setup of the execution environment before running tests, by checking the **Customize->Force Target Setup** menu option.

By default this option is not checked, so it is the user responsibility to check it if needed.

CHAPTER
2

In Application Unit Testing

As explained earlier **C-TestIt!** can run using an executable file produced by Cosmic Tools.Tests can be created, saved, loaded and executed later, or they can be grouped in a "**Testorama**".

We are now going to see how to:

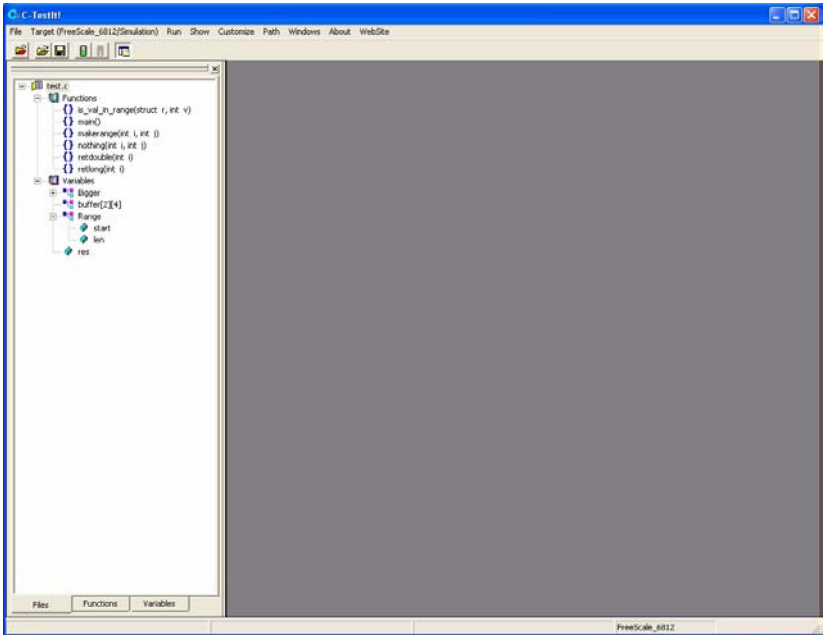
- Creating a Test
- Specifying Input and Output values
- Run a test
- Adding Assertions
- Save a Test
- Create a Testorama
- Run a Testorama
- Save a Testorama

Creating a Test

To create a test, the first step is to load the executable file that contains the function to be tested. To do so you can use either the menu or the button bar.

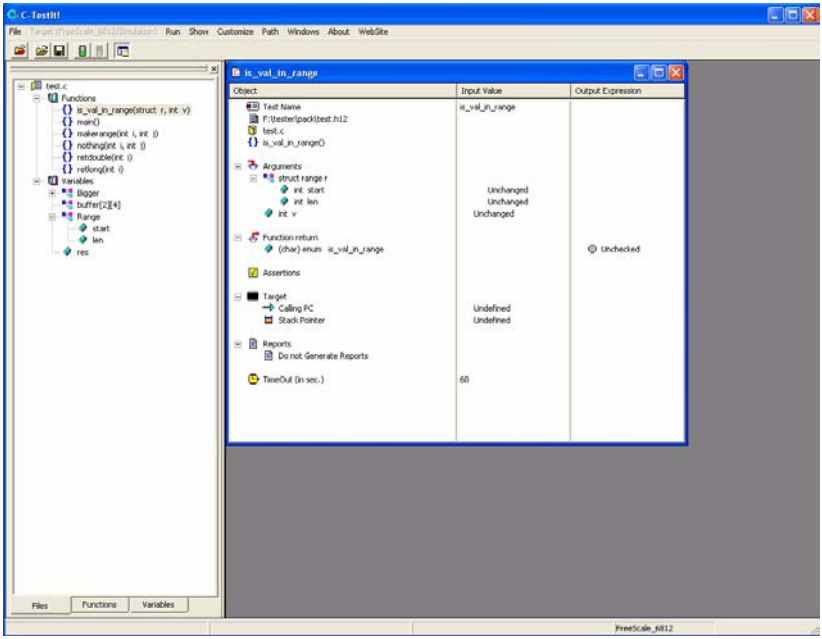
Once you have loaded the selected file the application pane will display information about the application. You will then be able to list the function names and the variable names included in your application.

Your screen will look like:



Now to create a test for a specific function, just right click on the function name in the application pane; this will open a test window with all the components of the test displayed with their default values.

Your screen will look like:



The test window is composed of three columns. The leftmost one lists all the objects that are manipulated by the test, the middle one shows input values when appropriate, and the right most one shows output values when appropriate. When a test is created all appropriate values are set to their defaults. "**Unchanged**" is used to indicate an input value that is not specified, and "**Unchecked**" is used to indicate an output value that needs not be checked for this test.

The leftmost window lists all the objects of the test in the following order:

- **Test Name**: this is a name that by default is the same as the function name under test. The user can edit this by right clicking on it.
- The Executable file name used for the test.
- The name of the source file that includes the function under test.
- The name of the function under test.

- Then we find the **Globals** entry. This entry exists if and only if the function under test uses global variables of the program. This entry can be expanded to view all the variables used as well as their components for aggregate variables. Each of these variables can receive an input value for the test by right clicking the corresponding entry in the middle window, and receive an output value by right clicking the corresponding entry in the rightmost window.

- This is then followed by the list of arguments to the function if appropriate. Each argument can receive an input value by right clicking on the corresponding entry in the middle window.

- The function return value. In the case of a function returning an aggregate, this entry can be expanded to show all components. The return value can be tested against a specified output value by right clicking the corresponding entry in the rightmost window.

- Then one finds the "**Assertions**". These are conditional expressions that will be tested during the execution of the function. To add an assertion simply right click on the Assertions icon or text in the leftmost window. This will bring up a dialog that is used to specify assertions.

- After, the **Target** entry is displayed. This entry allows the user to specify the Stack value used for the test and the address from which the function call should be executed.

- The **Reports** entry allows the user to specify whether reports should be created and where they should be saved. To modify the report status simply right click on the Reports entry in the leftmost window.

- Finally, there is the **TimeOut** entry. This entry allows the user to specify a time-out for the execution of a function. This is to cope with situations where the code being tested does not "end" execution. The TimeOut value is used to stop a test in such cases. To modify the TimeOut value simply right click on the correponding middle window entry.

Specifying Input and Output values

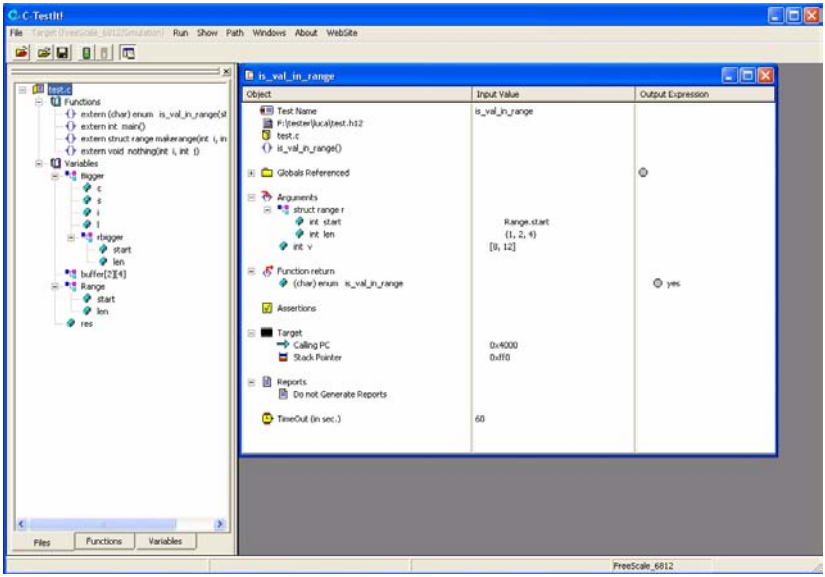
To specify an input or an output value, right click on the appropriate entry. Typing <*RETURN*> ends the editing, while typing <*ESCAPE*> cancels the editing and restores the initial value. Input and output values can be specified for globals, arguments and function return values as either simple values or valid C expressions. Additionally, inputs can be specified as a range of constant values or as a set of constant values. To specify a range the following notation is used: **`[<low_val>, <high_val>]`**, the test will be run for every value in that range; to specify a set the following notation is used: **`{<val1>, <val2>, ..., <valn>}`**, the test will be run for every value in the set.

If an output value is specified as a constant or a C expression without any comparators (*i.e.* <, > , ...), then it is taken to specify the exact value of the corresponding object; otherwise it is taken to be a C expression to be evaluated and the value thus obtained is tested for **TRUE** or **FALSE**.

For example, to ensure that the function under test returns a value greater than zero but less than 5, we could use the following **Output Expression**:

```
func() > 0 && func() < 5
```

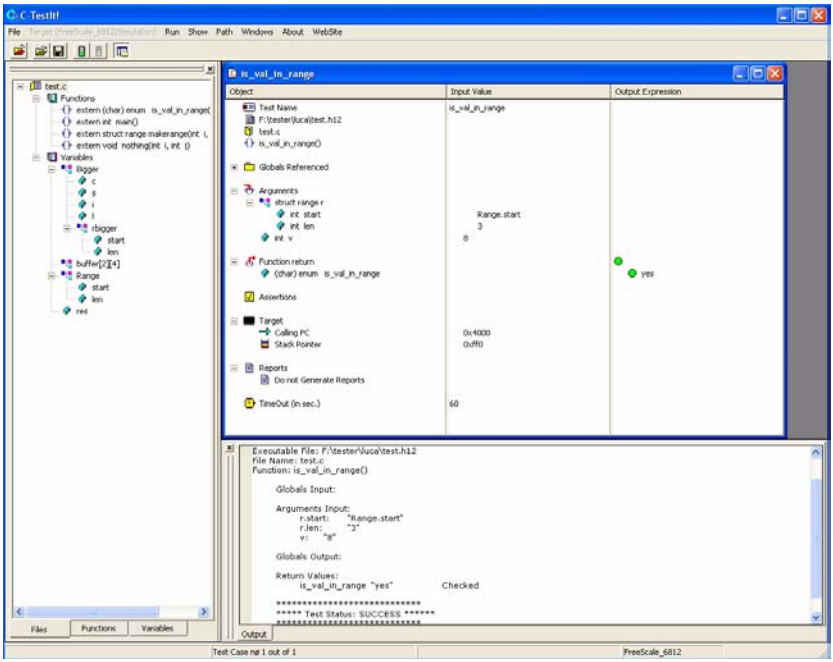
Once the inputs and outputs have been specified the screen will look like:



In this example the argument **`r.len`** is specified as a set of values, and argument **`v`** is specified as a range. This produces fifteen different sets of input values for the test, which is pretty much the same as producing fifteen different test cases.

Run a test

Once a test is completely specified, you can launch the test, and the result will be something like:



You can see the **GREEN** light icon next to the function output, which highlights the fact that the return value of the function does match the output value specified. If that was not the case the icon would be a **RED** light. You can also see the **Output** window which contains a textual report as to the test execution.

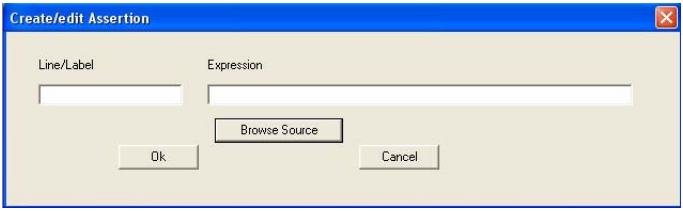
Adding Assertions

An assertion is an expression that will be evaluated while the program is running. An assertion can be attached to a particular source line in the function under test. This will allow the user to test a condition every time this line of code is executed. Please note that the assertion is evaluated **BEFORE** the line of code it is attached to is executed.

Assertions can, for example, allow the user to test that a particular variable meets some specified condition when a line of code is reached.

To add an assertion to a test, simply right click on the Assertions icon in the leftmost part of the test window.

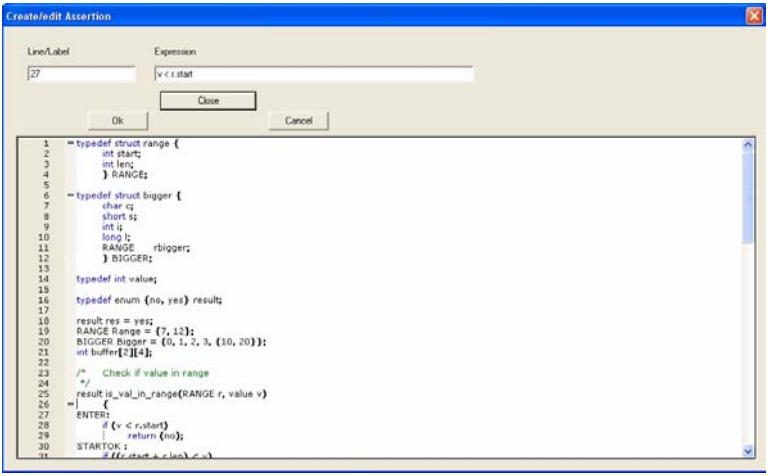
The following dialog box is then displayed:



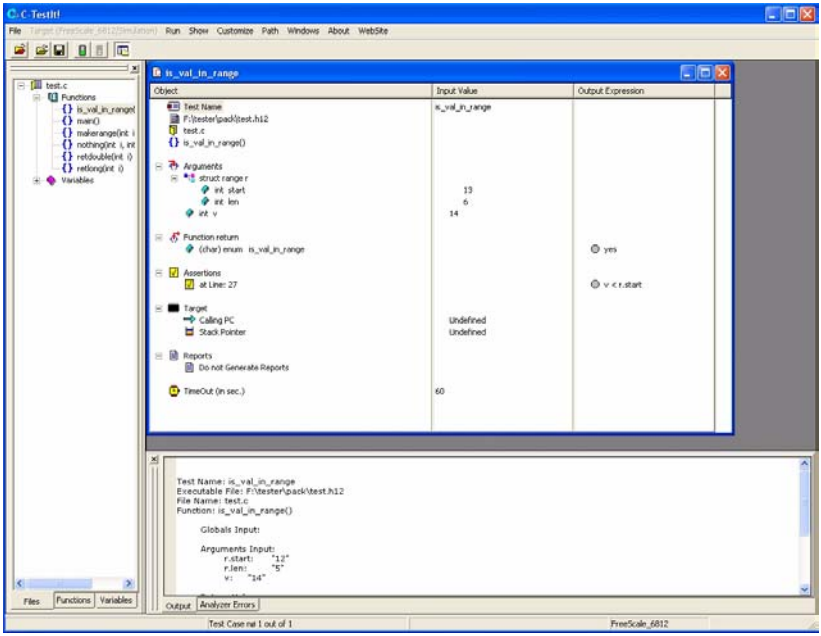
The **Browse Source** button can be used to display the code of the function under test. You must then specify a line number or a C label and a valid C expression that will be evaluated when that particular line of code is reached. The line number/Label can be specified either by typing it in the or by double-clicking on it in the browse window.

When a test is executed, assertions will be displayed with a **GREEN** light icon if their expression is **TRUE**, and will be displayed with a **RED** light icon if their expression is **FALSE**.

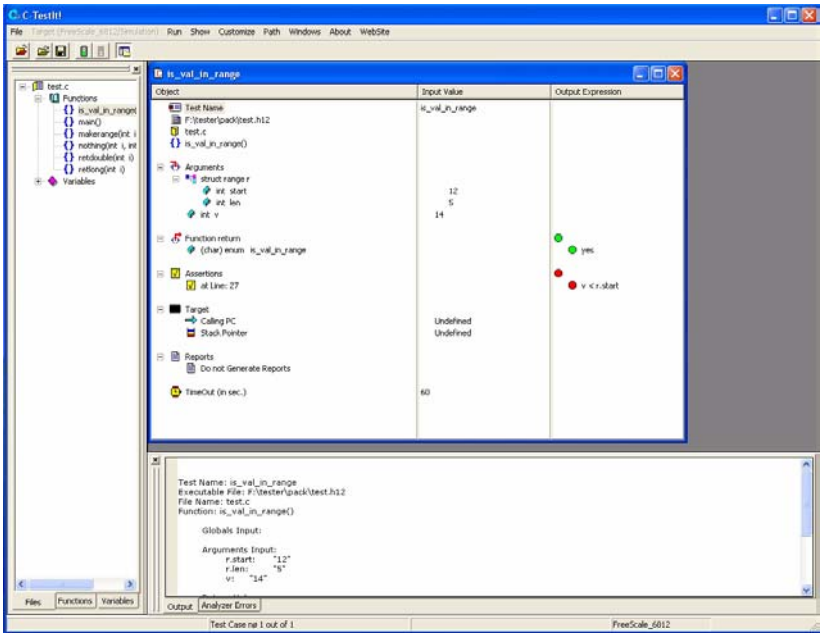
Here is an example of assertion:



Once you validate the assertion (by clicking OK), the screen will look like:



If we run a test with assertions here is what the screen may look like:



In this case you can see that the assertion is displayed with a **RED** light icon because it did not evaluate to **TRUE**.

Assertions are the means to create "***GRAY BOX Tests***", *i.e.* tests where the user can see what happens inside the function under test.

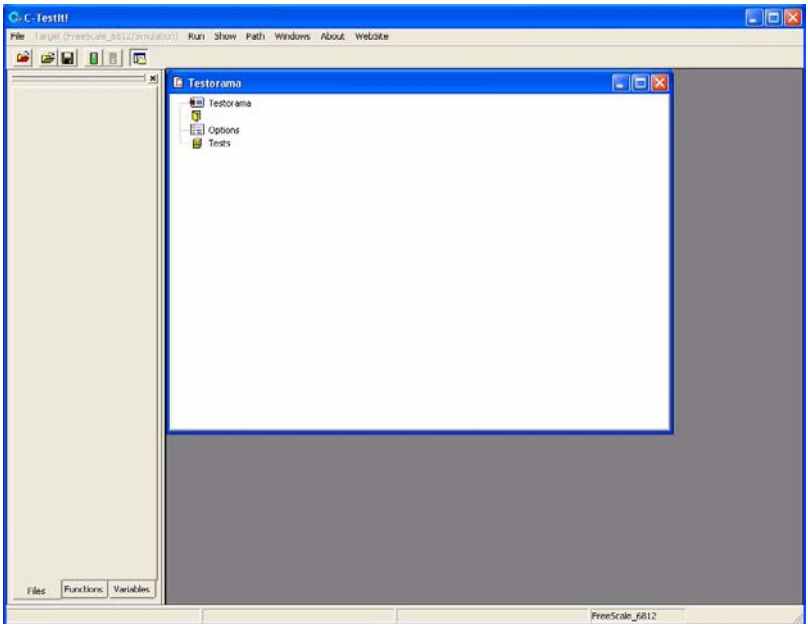
Save a Test

To save a test, select the appropriate menu option under the File menu. You will have to enter a name for the saved test. Tests are saved in **.CTH** files, and can then be later reloaded for execution or editing/ updating.

Create a Testorama

A testorama is a suite of tests, all using the same executable file. When a testorama is executed reports for all of the tests can be collated in a unique report thus allowing the user to gather test information for a set of related functions.

To create a testorama choose **File->New->Testorama**. This creates a testorama window and the screen should look like:

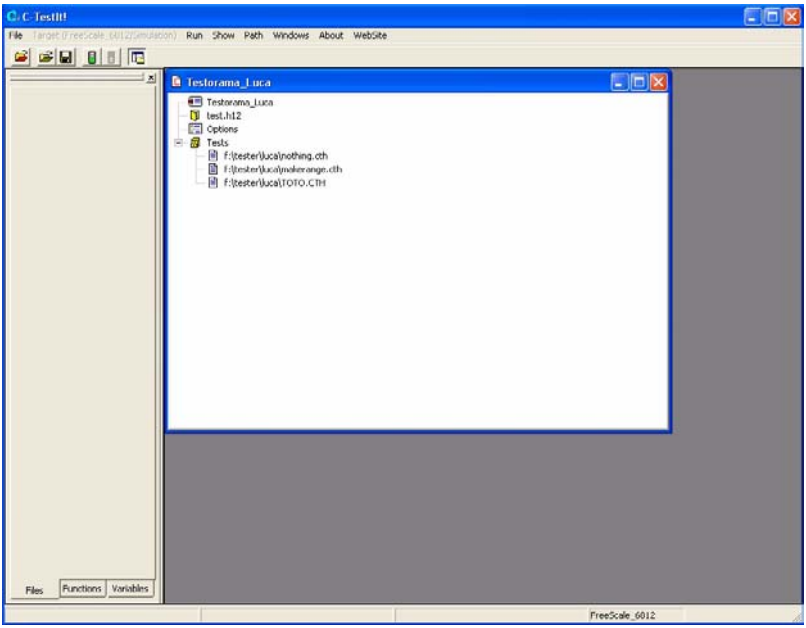


The testorama window shows:

- The testorama **name**. You can update this by right clicking on it.
- The icon for the executable file that the testorama will use. This will be updated when you add a test to the testorama.
- The **Options** icon will allow the user to specify options for the testorama.

- The set of test files for this testorama.

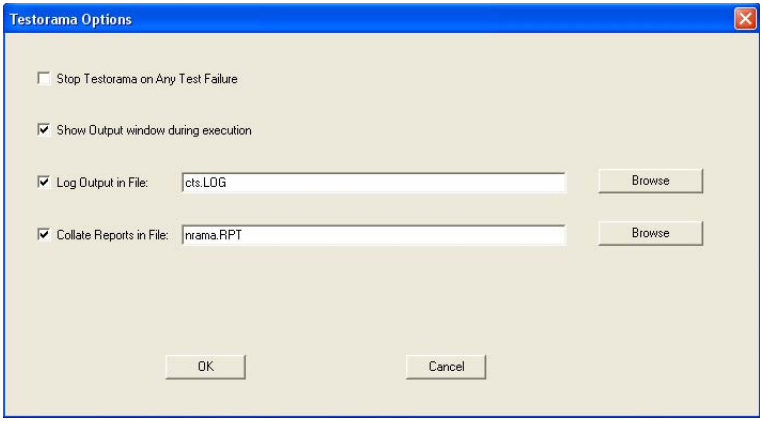
Here is what the screen may look like once you have started to specify the components of the testorama:



This testorama is based on executable file **`test.h12`**, and it will execute the following tests in sequence:

```
f:\tester\luca\nothing.cth
f:\tester\luca\makerange.cth
f:\tester\luca\TOTO.cth
```

The options of a testorama can be viewed/specified by right clicking, the **Options** icon. The following dialog is then displayed:

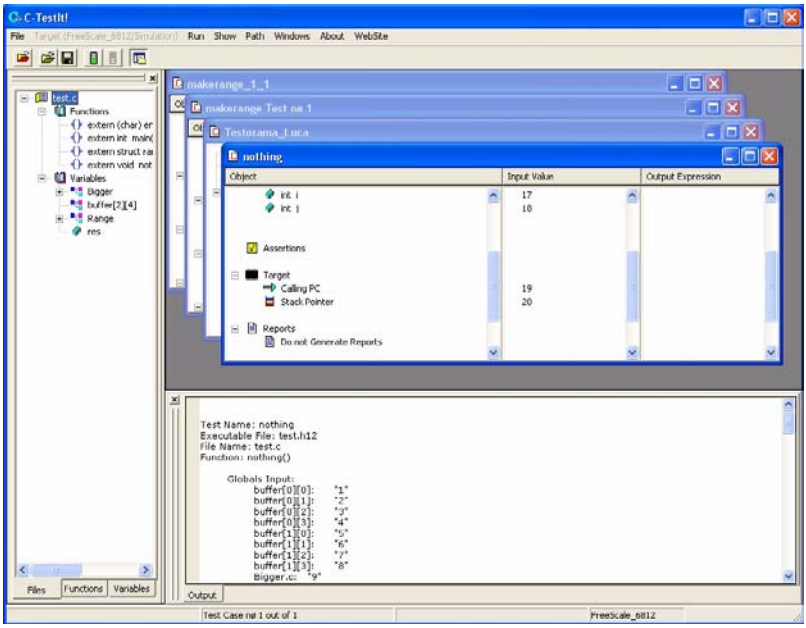


The options are:

- **Stop Testorama on Any Test Failure**.
- **Show Output window during execution**. This will open the Output window to show the test's execution.
- **Log Output in File**. This allows the user to log the output produced in the output window in a file, which can be archived or examined later.
- **Collate Report in File**. This allows the user to collate all the report files produced by the individual tests in a unique file.

Run a Testorama

Once a testorama is completely specified, you can launch the test, and the result will be something like:



A test window is opened for every test specified in the testorama. This window shows the results of each particular test.

Save a Testorama

To save a testorama, select the appropriate menu option under the File menu. You will have to enter a name for the testorama. Testoramas are saved in **.CTS** files, and can then be later reloaded for execution or editing/updating.

CHAPTER
3

Source Unit Testing

As explained earlier **C-TestIt!** can run using a C source file.

We are now going to see how to:

- Customizing C-TestIt!
- Creating a Source Test
- Specifying Input and Output values
- Run a test
- Adding Assertions

Customizing C-TestIt!

When using **Source Testing**, it is possible to customize some parts of the product.

C-TestIt! uses name mangling to create the variables it needs in the generated C source code. This mangling is done by prefixing the user names with predefined strings, namely **__argument__** and **__result__**. These predefined strings can be changed if the user uses some similar convention in his own code. To do this use the **Customize->Source Options** menu.

C-TestIt! can either generate simple test code, or fully instrumented test code. The later not only includes the code to achieve the tests but also includes code that checks the result of the test and uses the printf channel to output test results. This can be configured via the **Customize->Source Options** menu.

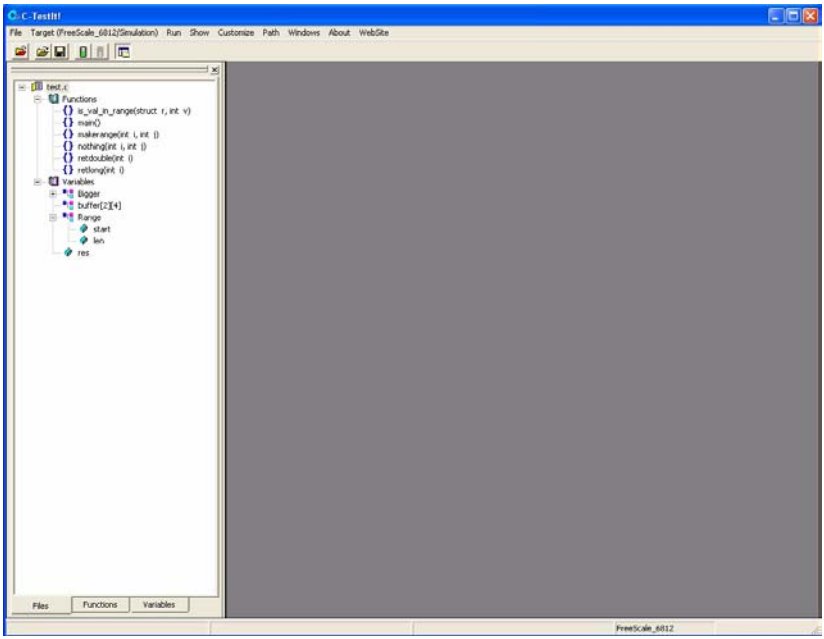
Finally it is also possible to configure the options used to compile the test, thus allowing to build the tests with the same options than those that will be used for the real code. This is achieved via the **Customize->Compiler Options** menu.

Creating a Source Test

To create a test, the first step is either to load a C source file that contains the function to be tested. To do so you can use either the menu or the button bar.

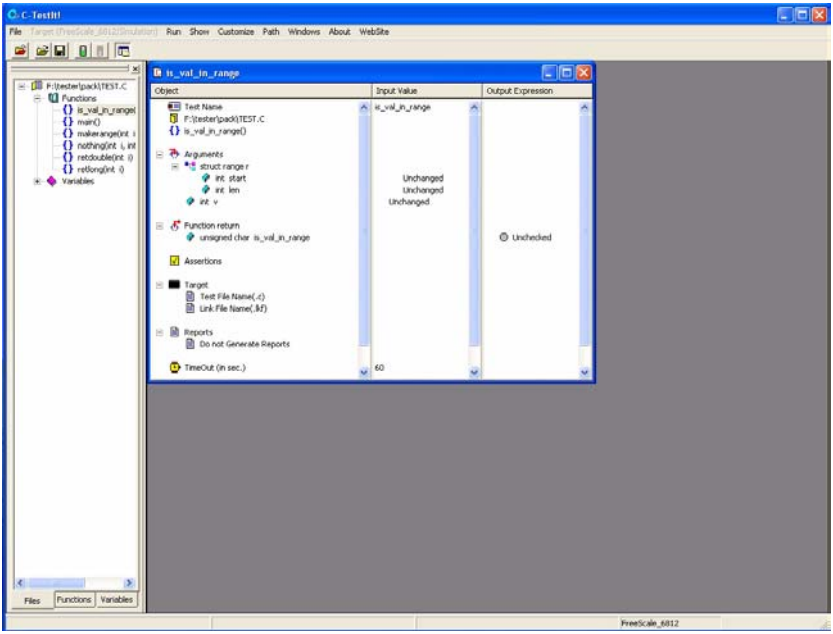
Once you have loaded the selected file the application pane will display information about the application. You will then be able to list the function names and the variable names included in your application.

Your screen will look like:



Now to create a test for a specific function, just right click on the function name in the application pane; this will open a test window with all the components of the test displayed with their default values.

If you have loaded a source file your screen will look like:



The test window is composed of three columns. The leftmost one lists all the objects that are manipulated by the test, the middle one shows input values when appropriate, and the right most one shows output values when appropriate. When a test is created all appropriate values are set to their defaults. "**Unchanged**" is used to indicate an input value that is not specified, and "**Unchecked**" is used to indicate an output value that needs not be checked for this test.

The leftmost window lists all the objects of the test in the following order:

- **Test Name**: this is a name that by default is the same as the function name under test. The user can edit this by right clicking on it.
- The name of the source file that includes the function under test.
- The name of the function under test.

- Then we find the **Globals** entry. This entry exists if and only if the function under test uses global variables of the program. This entry can be expanded to view all the variables used as well as their components for aggregate variables. Each of these variables can receive an input value for the test by right clicking the corresponding entry in the middle window, and receive an output value by right clicking the corresponding entry in the rightmost window.

- This is then followed by the list of arguments to the function if appropriate. Each argument can receive an input value by right clicking on the corresponding entry in the middle window.

- The function return value. In the case of a function returning an aggregate, this entry can be expanded to show all components. The return value can be tested against a specified output value by right clicking the corresponding entry in the rightmost window.

- Then one finds the "**Assertions**". These are conditional expressions that will be tested during the execution of the function. To add an assertion simply right click on the Assertions icon or text in the leftmost window. This will bring up a dialog that is used to specify assertions.

- After, the **Target** entry is displayed. It allows the user to specify the name of the C source file generated for the test, and the linker command file.

  - **These use internal default values if none are specified.**

    - Specifying a source file name allows you to keep copies of the tests and archive them if needed; if the source file is not specified **C-TestIt!** uses a file called t_code.c.

    - Specifying a link command file gives you better control on the test; if no link command file is specified **C-TestIt!** uses a typical default link file**.**

  - **Please note that C-TestIt! needs to find a label called _exit which is used to break execution when the test is ended;** so it is highly recommended that the C run time header file provided with compiler be used.

- The **Reports** entry allows the user to specify whether reports should be created and where they should be saved. To modify the report status simply right click on the Reports entry in the leftmost window.

- Finally, there is the **TimeOut** entry. This entry allows the user to specify a time-out for the execution of a function. This is to cope with situations where the code being tested does not "end" execution. The TimeOut value is used to stop a test in such cases. To modify the TimeOut value simply right click on the correponding middle window entry.

Specifying Input and Output values

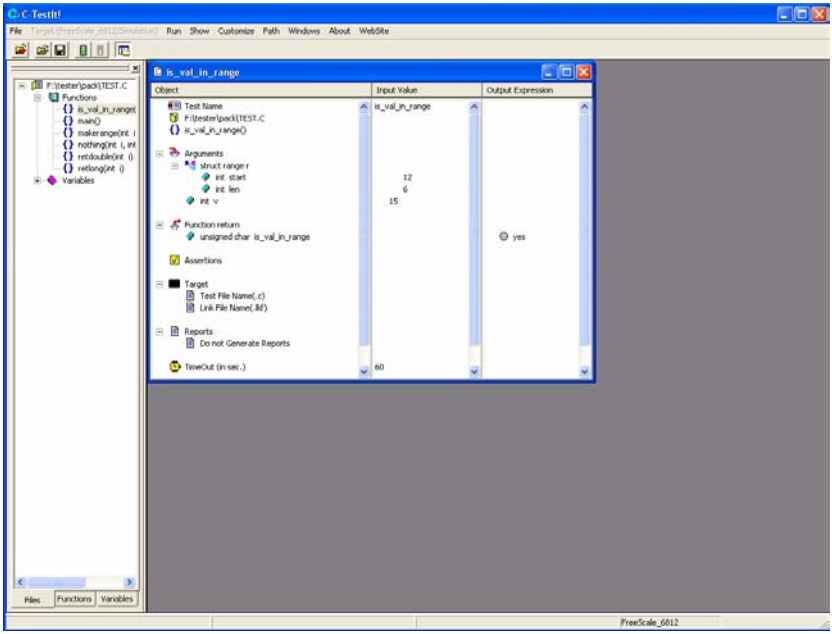
To specify an input or an output value, right click on the appropriate entry. Typing <RETURN> ends the editing, while typing <ESCAPE> cancels the editing and restores the initial value. Input and output values can be specified for globals, arguments and function return values as either simple values or valid C expressions.

If an output value is specified as a constant or a C expression without any comparators (*i.e.* `<, > , ...`), then it is taken to specify the exact value of the corresponding object; otherwise it is taken to be a C expression to be evaluated and the value thus obtained is tested for TRUE or FALSE.

For example, to ensure that the function under test returns a value greater than zero but less than 5, we could use the following **Output Expression**:

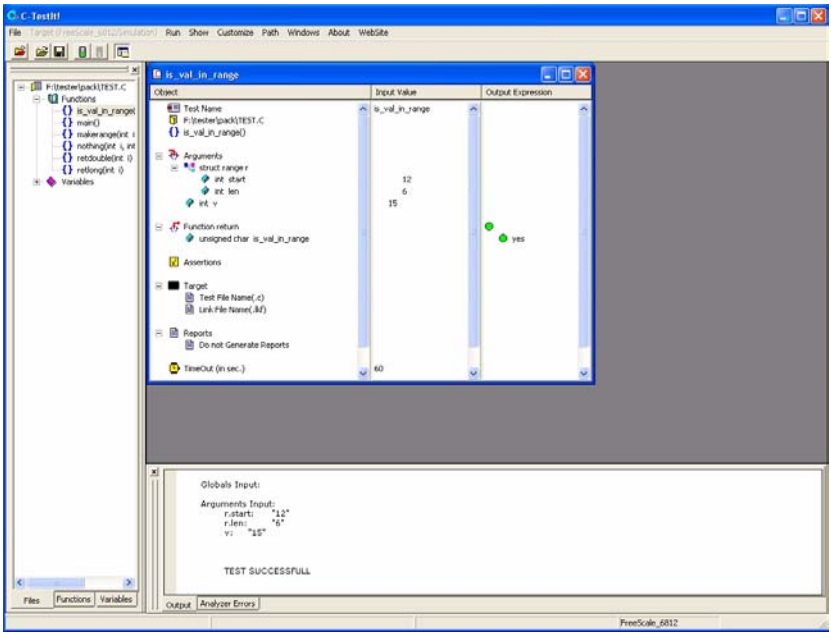
```
func() > 0 && func() < 5
```

Once the inputs and outputs have been specified the screen will look like:


C-Test!!
File
Run
Show
Customize
Path
Windows
About
WebSite
F:\tester\pack\TEST.C
Functions
is_val_in_range
main()
makerange(int i
nothing(int i, int
retdouble(int i)
retlong(int i)
Variables
Files
Functions
Variables
is_val_in_range
Object
Input Value
Output Expression
Test Name
F:\tester\pack\TEST.C
is_val_in_range()
is_val_in_range
Arguments
struct range r
int start
12
int len
6
int v
15
Function return
unsigned char is_val_in_range
yes
Assertions
Target
Test File Name(.c)
Link File Name(.lkf)
Reports
Do not Generate Reports
TimeOut (in sec.)
60
FreeScale_6812

Run a test

Once a test is completely specified, you can launch the test, and the result will be something like:



You can see the **GREEN** light icon next to the function output, which highlights the fact that the return value of the function does match the output value specified. If that was not the case the icon would be a **RED** light. You can also see the **Output** window which contains a textual report as to the test execution.

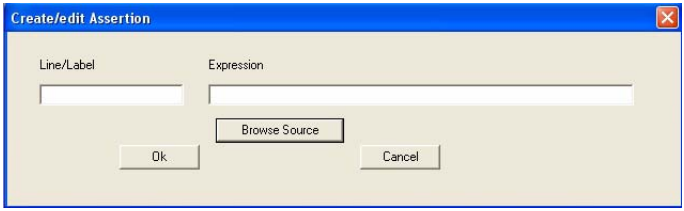
Adding Assertions

An **assertion** is an expression that will be evaluated while the program is running. An assertion can be attached to a particular source line in the function under test. This will allow the user to test a condition every time this line of code is executed. Please note that the assertion is evaluated **BEFORE** the line of code it is attached to is executed.

Assertions can, for example, allow the user to test that a particular variable meets some specified condition when a line of code is reached.

To add an assertion to a test, simply right click on the Assertions icon in the leftmost part of the test window.

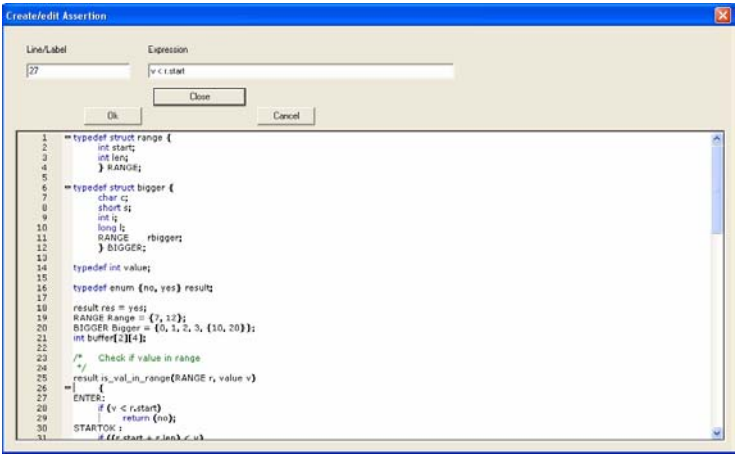
The following dialog is then displayed:



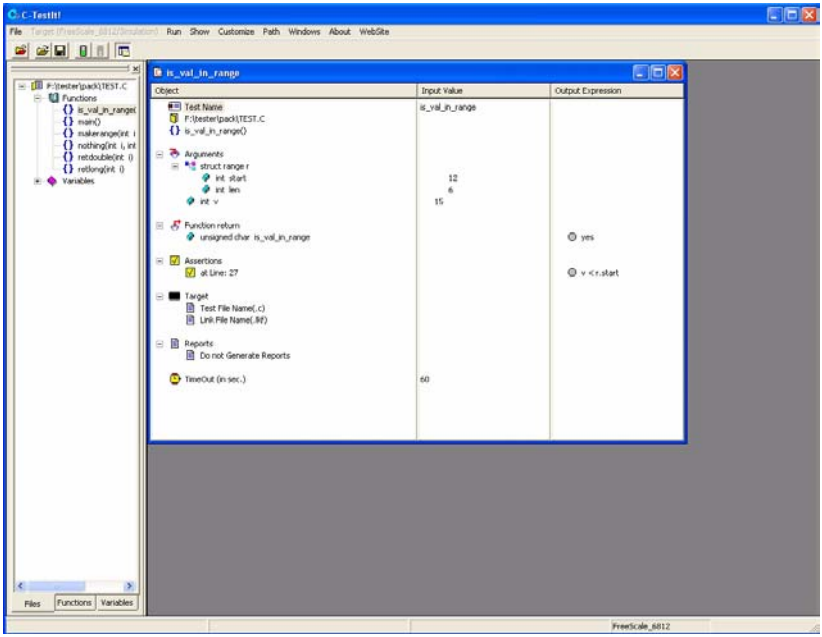
The **Browse Source** button can be used to display the code of the function under test. You must then specify a line number or a C label and a valid C expression that will be evaluated when that particular line of code is reached. The line number/Label can be specified either by typing it in the or by double-clicking on it in the browse window. **Please note that at this stage there is no check done as to the validity of the line or the Label**.

When a test is executed, assertions will be displayed with a **GREEN** light icon if their expression is **TRUE**, and will be displayed with a **RED** light icon if their expression is **FALSE**.

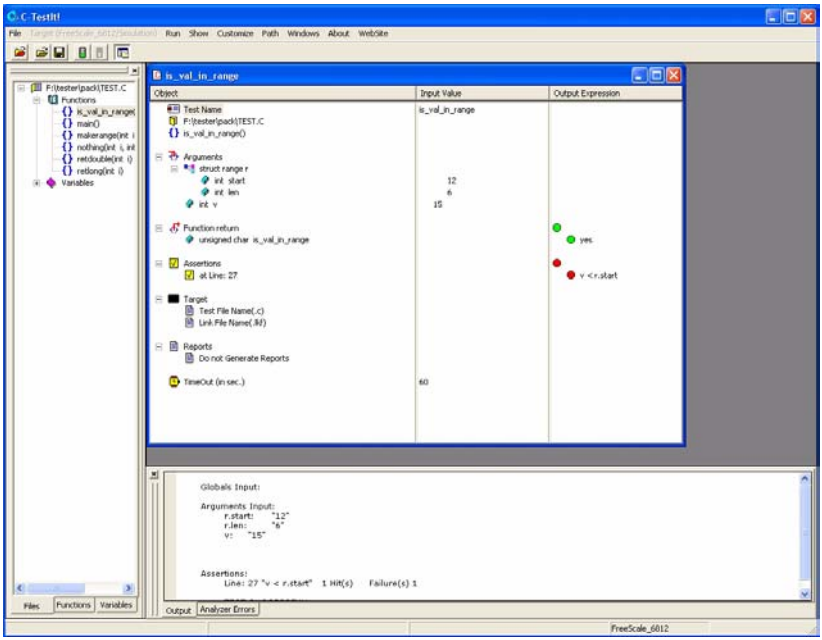
Here is an example of assertion:



Once you validate the assertion (by clicking OK), the screen will look like:



If we run a test with assertions here is what the screen may look like:



In this case you can see that the assertion is displayed with a **RED** light icon because it did not evaluate to **TRUE**.

Assertions are the means to create "**GRAY BOX Tests**", *i.e.* tests where the user can see what happens inside the function under test.

Index

U

V