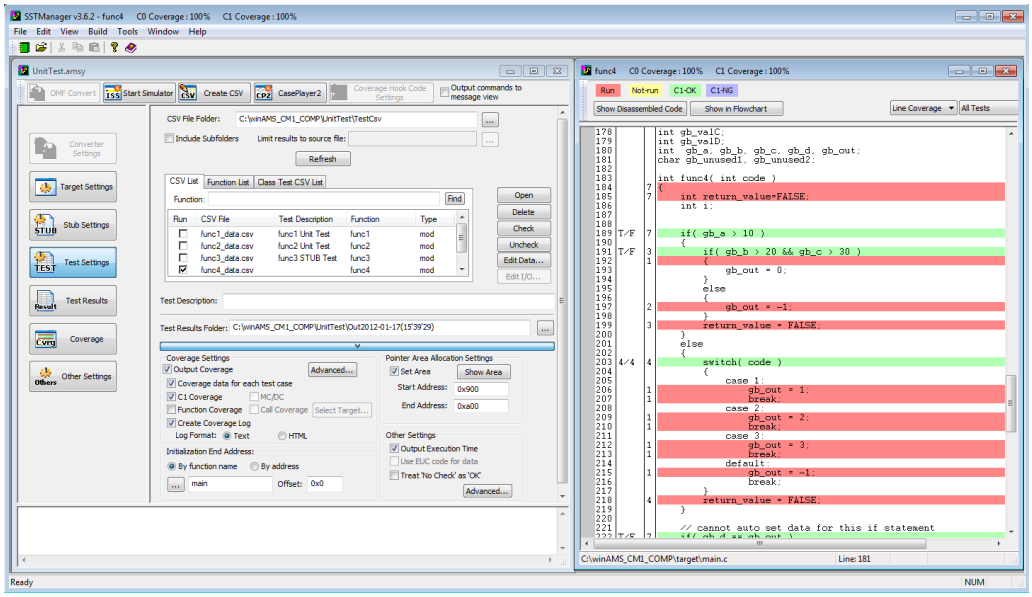


CoverageMaster winAMS Tutorial

Ver. 2.1.0
Jun 2016

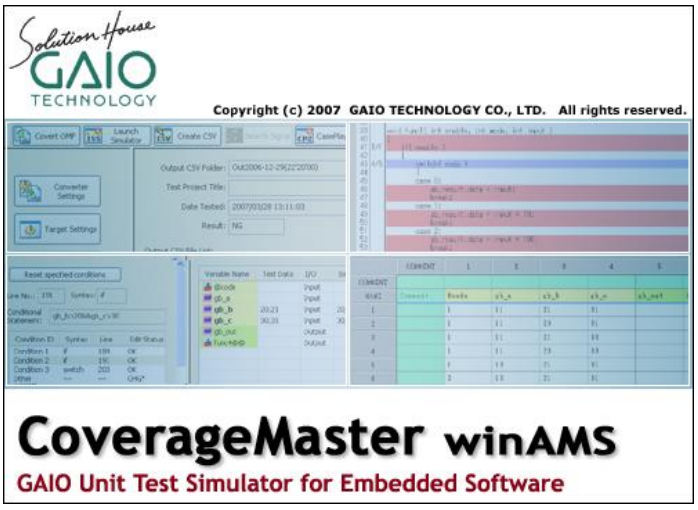
Table of Contents

CoverageMaster Tutorial

Introduction

At this time we wish to thank you for your interest in GAIO TECHNOLOGY's unit testing tool, **CoverageMaster winAMS**. CoverageMaster winAMS is a unit testing tool for evaluating and improving embedded software.

This tutorial includes practice exercises designed for first time users. By following this tutorial and completing the exercises, the user will gain an understanding of CoverageMaster winAMS's basic usage and features.



CoverageMaster Overview

Before starting the exercises, first an explanation of CoverageMaster's features and operation will be presented.

Embedded Software Unit Test Tool

CoverageMaster winAMS is a unit testing tool for embedded software. It performs testing by executing the software code (functions) using an MPU simulator (System Simulator).

The MPU simulator (System Simulator) operates by running the actual cross-compiled MPU code (object code). Because the actual target code is used in the simulation, it is possible to verify the program's behavior and also check for errors that depend on the target microprocessor.

CoverageMaster winAMS includes the following components:

SSTManager: The primary UI application used to manage the unit testing operations.
System-G: MPU Simulator (System Simulator) – includes support for a variety of MPUs.
WinAMS: the unit test simulator.

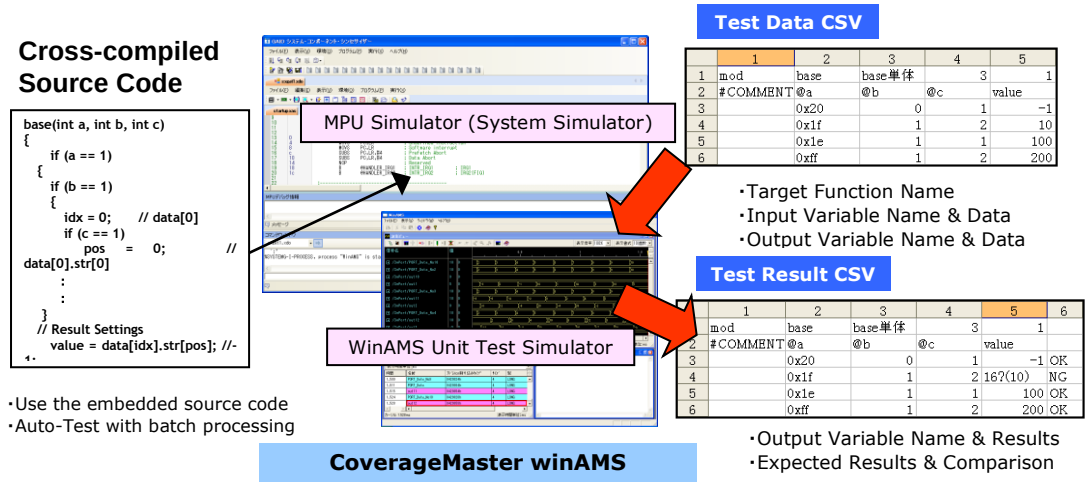
In addition to these programs a cross compiler or Integrated Development Environment (IDE) will be required to prepare the source files for testing.

CSV Format Files for Test I/O

CoverageMaster winAMS uses CSV format files during simulations, in order to increase the unit testing efficiency while maintaining accuracy.

Individual unit testing settings, as well as input / output (I/O) data for variables and stub functions are all saved in CSV files. Thus, by using CSV files it is unnecessary to include any special test code or input data into the target source code.

The diagram below illustrates the flow of CoverageMaster winAMS unit testing.



First, the cross compiled code of the target function is loaded into the MPU simulator. The code must be linked executable object code (unlinked intermediate object files cannot be used).

Next a CSV file must be created in order to test the **base()** function (in the example above). Information about the function such as the function name, input variables, output variables, test conditions, and the test data (test vector) will be included in the CSV file.

In the above example, test data is entered for the input variables ***@a, @b, @c***. In addition, the expected value is entered for the global variable ***value***.

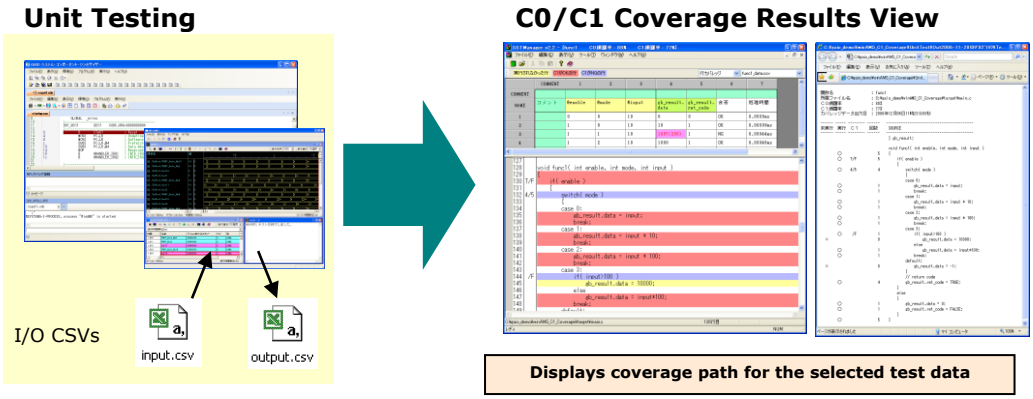
Expected values for output variables (such as *value* in the above example) may be entered to compare with the actual results. The expected value cells however may be left blank if you simply wish to output the results.

If expected values are entered, the actual test results will be compared with the expected values and reported as NG or OK in the test result column. If the expected results differ from the actual results, both values will be displayed in the cell. In the above example line 4 of the test result CSV reported a result of **NG** and **16?(10)** in the value column. In this case, the 16 represents the actual result, and the 10 represents the expected value.

Automated Unit Testing with Coverage and Test Results

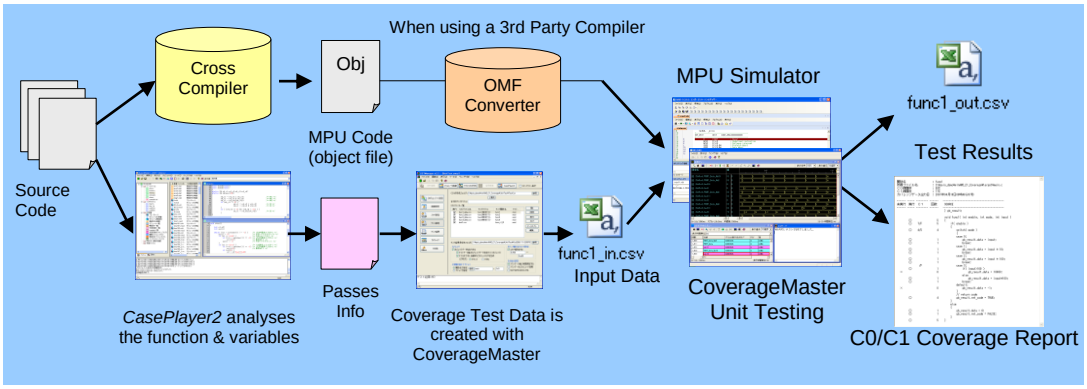
By simply creating a CSV test data file the rest of the test simulation process is automated. Simply click the **Start Simulator** button and the simulation for the input test data will be run. Upon completion, the C0/C1 Coverage (*) and test results will be output. The coverage results may be viewed anytime from the **Coverage** screen or output to a file.

(*): C1 Coverage requires the use of GAIO's "CasePlayer2" static analysis program.



Automatically Create Coverage Test Data using CasePlayer2

Input test data may be generated automatically for the unit test using the CasePlayer2 static analysis tool alongside CoverageMaster winAMS. The diagram below illustrates the simulation process.



CasePlayer2 retrieves the function's variable information and then creates test data to fulfill coverage requirements. The test data is sent to CoverageMaster winAMS, and then simulated using GAIO's MPU simulator.
Note: CasePlayer2 tool usage is discussed in further detail during exercise 4 of this tutorial.

For using GAIO's Development Environment with CoverageMaster winAMS

Getting Ready

Preparing the Sample Code

To begin the exercises, first we will locate the provided sample code and compile it using GAIO's XASS-V series cross development environment.

If a 3rd party development environment will be used, please instead refer to the instructions in the appropriate CoverageMaster winAMS*: 3rd Party Development Environment Usage Guide* found at the end of this tutorial.

Extract the Sample Project

Extract the exercise files, then open GAIO's development environment.
(6-1.winAMS_CM1.zip and other sample test projects may be downloaded from the CoverageMaster winAMS Evaluation Kit Site)

1. Extract **6-1.winAMS_CM1.zip** to the C root directory.
   (note: the root directory is used for sake of simplicity, other directories are also valid)
2. Open the **C:¥winAMS_CM1¥target** folder.
3. Double-click the **SAMP1.gxp** project file to open in the project in GAIO's development environment (GAIO Framework).

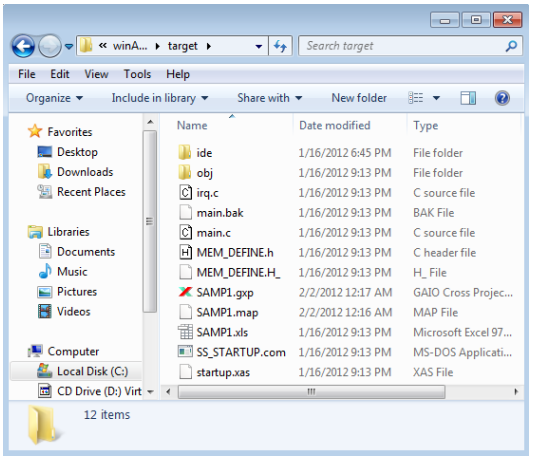
There are three source files included in the project for this tutorial:

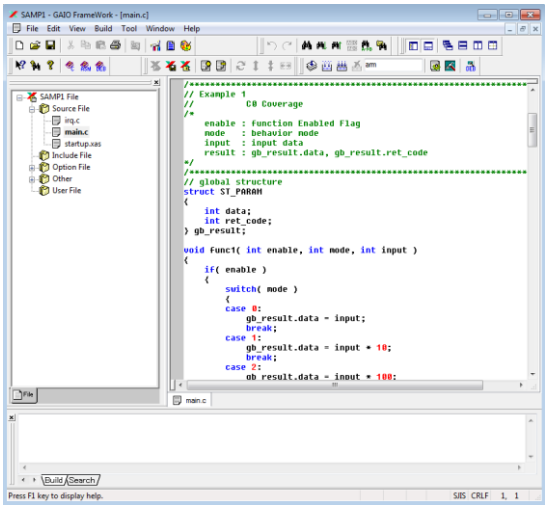
main.c: includes the functions to be tested.
startup.xas: assembly coded startup routine for ARM7.
irq.c: interrupt handler function written in C (not used in this tutorial).

The functions we will be testing during these exercises, **func1() - func4()**, are included in **main.c**. The **main()** function is also included in **main.c**, but it is left blank. This is because CoverageMaster's MPU Simulator (System Simulator) executes instructions like an actual MPU device, hence linked executable code is required.

Now let's build the sample code and create an executable object file.

4. Select **Build -> Rebuild** from the application menu to compile the source code and confirm the build is completed successfully.





Exercise File Structure

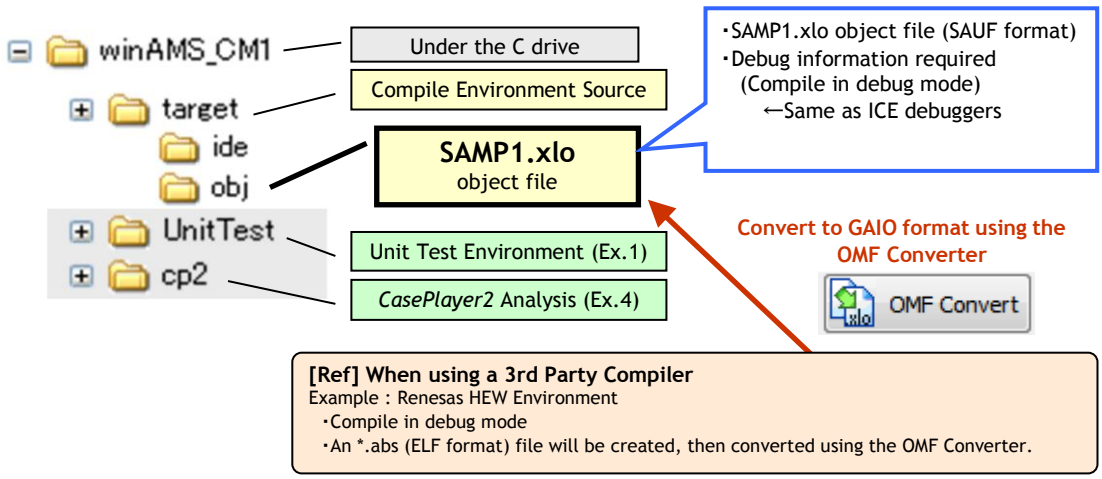
This section explains the structure of the sample files used in the exercises. First we will cover the **SAMP1.xlo** executable object file found in *c:¥winAMS_CM1¥target¥obj* after compiling.

The ***.xlo** file extension is an object file created by GAIO's XASS-V series cross development environment. Its format is according to GAIO's original SAUF file format. CoverageMaster winAMS may only perform tests using *.xlo files.

CoverageMaster's built in object conversion function (OMF Converter) may be used to convert the file format (debug information) of object files compiled using a 3rd party cross compiler into compatible *.xlo files. CoverageMaster may be configured to automatically perform OMF conversions whenever needed, eliminating the need to manually start the conversion by the user.

* Note that for those using GAIO's development environment it is unnecessary to perform an OMF conversion since the object file is already in proper *.xlo format. For further details about the OMF conversion process, please refer to the CoverageMaster winAMS*: 3rd Party Development Environment Usage Guide* for your appropriate environment found at the end of this tutorial.

Lastly when compiling, it is necessary that the executable object file include debug information in order to perform unit testing using CoverageMaster. Compiler optimizations and other compiler options however, may be used.



Exercise 1: Our First Test (Intro)

The purpose of exercise 1 is to familiarize you with fundamental operations of unit testing with CoverageMaster. Topics covered will include: creating a test project, creating a CSV test file, starting the simulator, and reviewing the test results and coverage.

Create a New Test Project

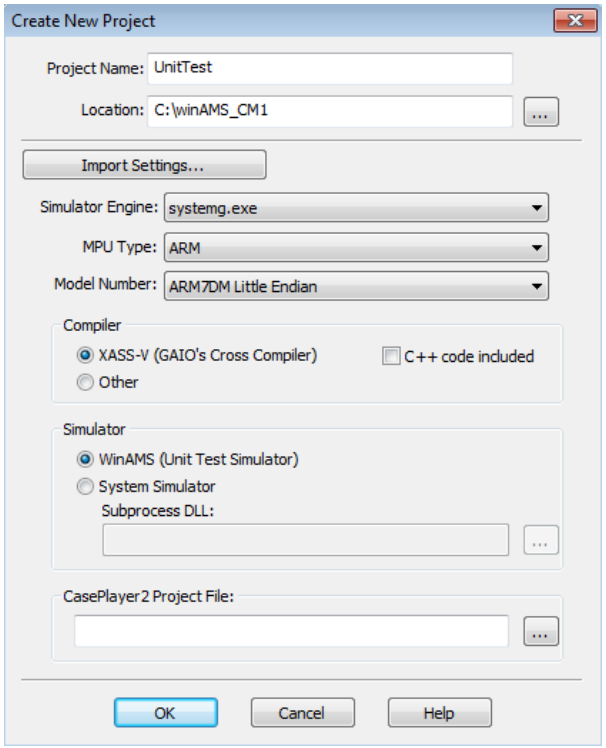
First let's create the test project that will be used throughout this tutorial (exercises 1 – 4).

1. Start **SSTManager**: Start -> All Programs -> WinAMS -> WinAMS(SSTManager)
2. Select **File -> New Project** from the SSTManager menu.

Use the following settings in the **New Project Creation Dialog**:

3. Project Name: **UnitTest**
   (a folder with this name will be created in the location set below)
4. Location: **C:¥winAMS_CM1**
5. Simulator Engine: **system-g.exe**
6. MPU Type: **ARM**
7. Model Number: **ATM7DM Little Endian**
8. Compiler: **XASS-V (Gaio's Cross Compiler)**
9. Simulator: **WinAMS (Unit Test Simulator)**
10. Leave the CasePlayer2 Project File region blank for now (will be used in Ex.4).

Click **OK** to create the new test project will the above settings.
New project folder: **C:¥winAMS_CM1¥UnitTest**



For using GAIO's Development Environment with CoverageMaster winAMS

Target Settings

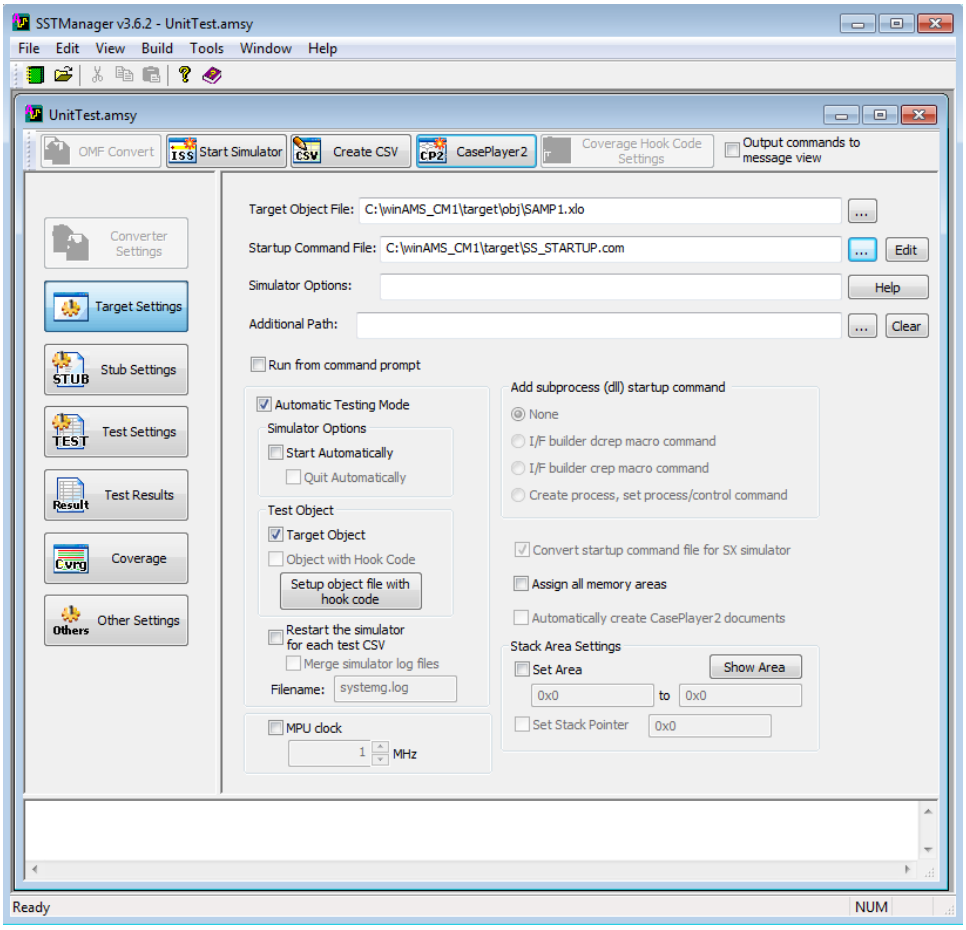
Next we will configure the test target settings.

First we must specify the object file to be simulated. For this tutorial we will use the **SAMP1.xlo** we compiled in the steps above. This single file contains all the needed information to perform the simulation such as the debug information, variable names, source file path, source code line information, etc.

1. Click the **Target Settings** button located as the second from the top of the column of buttons found on the left side.
2. Object File: **C:¥winAMS_CM1¥target¥obj¥SAMP1.xlo** (click the **…** button to select).

Next we will specify the **Startup Command File** that contains the settings for the MPU Simulator. This file is a script file that will be executed at the start of each simulation.

3. Startup Command File: **C:¥winAMS_CM1¥target¥SS_STARTUP.com** (click the **…** button to select).

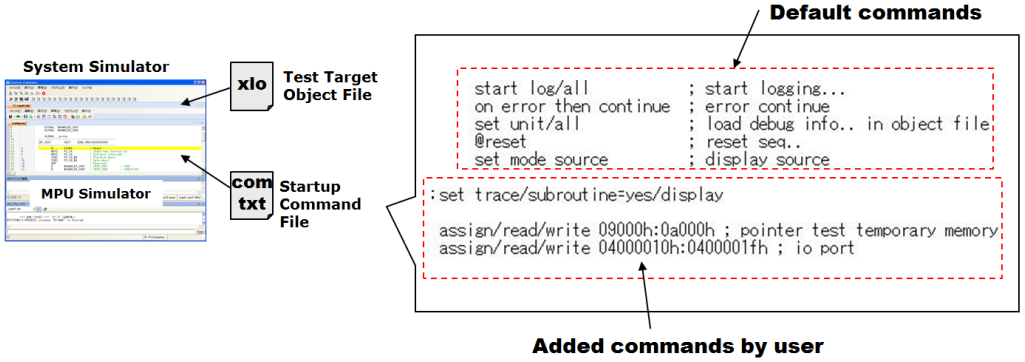


Startup Command File

The startup command file is a script file loaded when the MPU simulator starts. It is used to change the MPU simulator configuration for unit tests. For example:

- Change memory attributes (ROM -> RAM)
- Skip unnecessary loops that wait for hardware or peripheral signals
- Change local variables while executing a target function
- Initialize a large memory area before starting unit tests

The Startup Command File used in this tutorial includes the following information: (click the **Edit** button on the right in order to open the file with notepad).



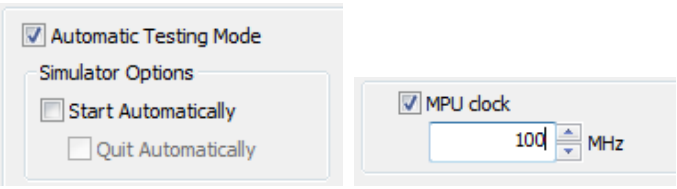
The first five lines are the basic commands for using the MPU simulator. These must not be removed. The set **trace/** command on the sixth line sets the MPU simulator to **trace mode**. Trace mode highlights the line of source code executed during the simulation.

The final two lines that begin with **assign/** are assign commands for configuring the test platform's memory attributes. Using these commands, memory errors may be tested during the simulation. If no memory attributes are configured, the memory will be assigned according to the link map information.

* The assign command was only included in this tutorial as a reference, it will not be used in the exercises and has been commented out.

Lastly we will use the following two settings for this tutorial.
From the **Target Settings** window:

1. Check the **Automatic Testing Mode** box.
2. Check the **MPU Clock** box, set to **100** MHz.



The MPU clock setting is an optional feature that may be used to output calculated execution time results of each test data set for a device of that speed. These results will be included in the test results CSV file.

The remainder of the settings may be left at their default values.

(Ref) Unit Test Flow

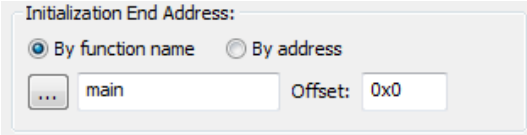
In this reference section the CoverageMaster winAMS unit test flow will be described.

CoverageMaster winAMS uses an embedded MPU simulator (System Simulator) to execute the functions coded for the target device. The MPU simulator is designed so that it will perform the same as if tested on an MPU evaluation board.

Normally when the MPU system powers up the hardware is reset, the program counter (PC) is set to the reset vector (start address), and then the operation begins. The MPU's initial program, called the startup routine, is found at the start of the program. When executed the startup routine initializes the stack pointer, initializes the MPU register, and makes other necessary settings for operation.

For a standard MPU system, the *main()* function is then called after the startup routine has been completed. When performing unit tests with CoverageMaster winAMS, the program counter is instead altered by the simulator to execute the target function.

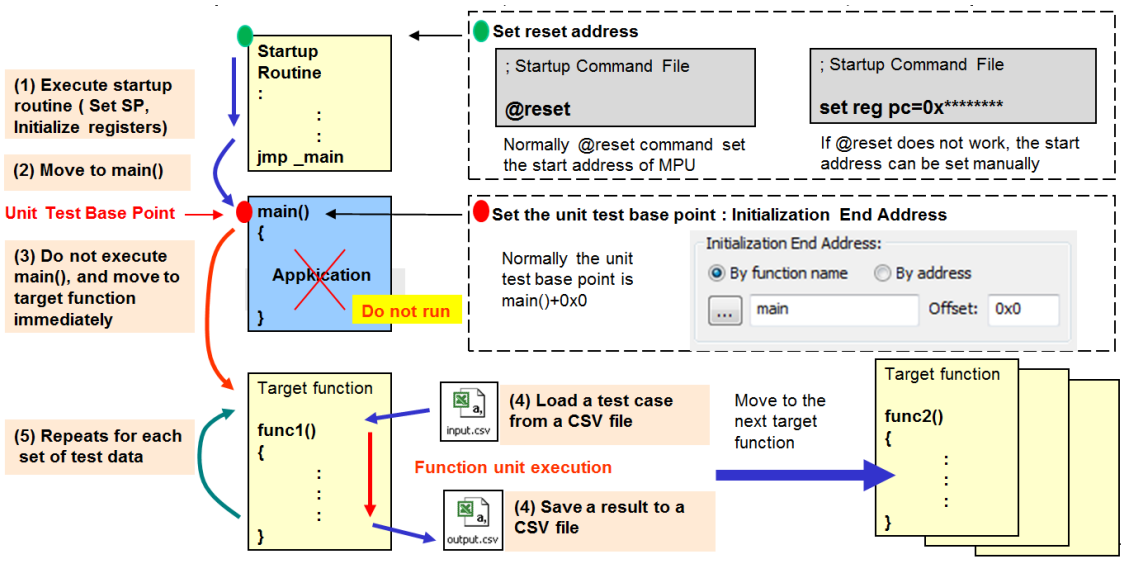
The **Specify Startup Routine End Address** setting in the **Test Settings** specifies where to start after the startup routine has ended. Normally this is set to *main()*, however it may be changed to a different address if necessary.



In order to start the startup routine when launching the MPU simulator, the reset address value should be specified to PC (program counter). The '@reset' command described in the startup command file will set the MPU's reset address value to PC automatically. However, the address may not be set correctly depending on the MPU type. In this case, the "set reg pc" command can be used to set the reset address value.

```
set reg pc = 0x********
```

Refer to the MPU hardware manual regarding the MPU reset address value.



Create a CSV File

Now let's begin practice exercise 1. The goal of exercise 1 is to create test data that will achieve 100% C0 coverage for *func1()* included in the sample code.

Below is a copy of the sample code for ***func1()***. It may also be viewed by opening **main.c** in the development environment.

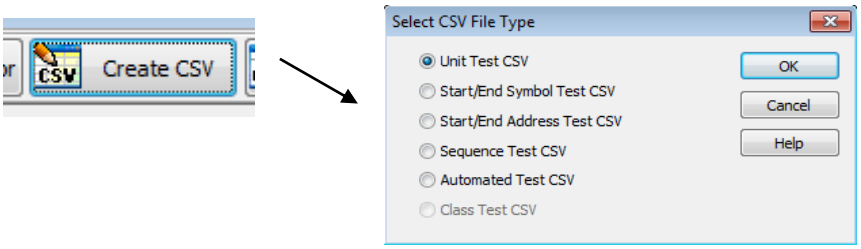
```
// Global structure
struct ST_PARAM
{
     int data;
     int ret_code;
} gb_result;

void func1( int enable, int mode, int input )
{
     if( enable )
    {
          switch( mode )
          {
          case 0:
               gb_result.data = input ;
               break;
          case 1:
               gb_result.data = input * 10;
               break;
          case 2:
               gb_result.data = input * 100;
               break;
          case 3:
               if( input >100 )
                    gb_result.data = 10000;
               else
                    gb_result.data = input *100;
               break;
          default:
               gb_result.data = -1;
          }
          // return code
          gb_result.ret_code = TRUE;
     }
     else
     {
          gb_result.data = 0;
          gb_result.ret_code = FALSE;
     }
}
```

First let's confirm the input / output conditions. The variables in red (above) are the input, giving us three input variables. There is no return value for this function; the results are stored in the global structure ***gb_result***.

Therefore, practice exercise 1's test conditions include three input variables (***enable***, ***mode***, ***input***) and two output variables (***gb_result.data***, ***gb_result.ret_code***) included in the global structure. We will use this information to create the CSV file.

Now we will create the CSV file for testing *func1().*

1. Click the **Create CSV** button in the upper portion of the SSTManager window.
2. Select **Unit Test CSV** and click the **OK** button.



The CSV creation dialog will now appear.

From this Unit Test CSV Creation dialog we will enter the test conditions.

3. Enter **func1_data** as the filename.

This will become the name of the CSV file with a **.csv** extension. The filename may be anything you wish, but since in most cases a CSV file is created for each function, including the function

name helps keeps them organized.

Next select the target function to be tested:

4. Next to the **Function** box click the **…** button to open a selection dialog, select **func1** by expanding the **f** tree and clicking on **func1**, then click **OK**.

The selection dialog includes a list of functions whose debug information is found in the **SAMP1.xlo** object file we included previously. Function names may also be typed in directly instead of by using the selection dialog.

5. Input **func1 unit test** into the **Test Description** box.

This setting is optional. However, by giving the test a simple descriptive it makes it easier to differentiate this CSV file from others. The test description will appear in the CSV file list and results report.

Next select the **INPUT** variables:

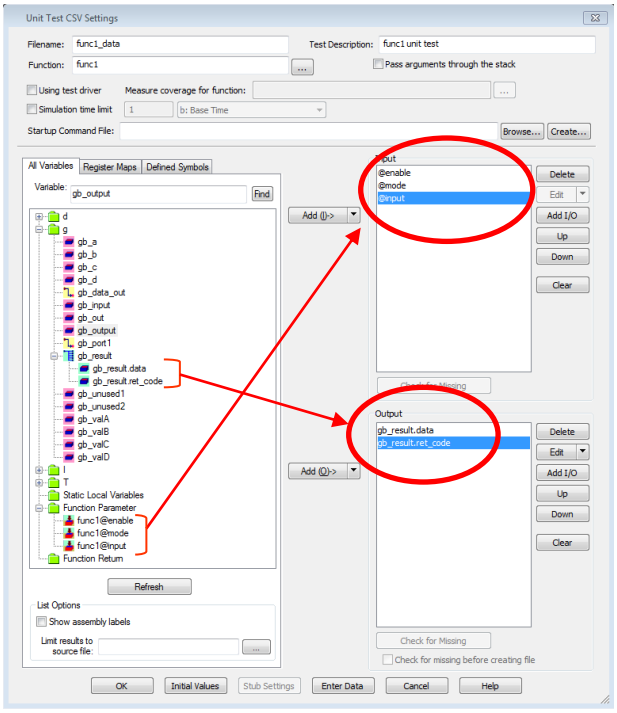
6. From the **All Variables** tab of the variable list view, expand the **Function Parameter** folder. (three function parameters for func1 will be shown)
7. Select ***Enable****, **Mode**, and **Input*** from the variable list, then add them to the INPUT list by clicking the **Add (I)->** button located next to the INPUT list.

Function Parameters appear as *function_name@variable_name* in the variable list.

Lastly, select the **OUTPUT** variables:

8. Expand the **g** folder in the variable list to display the variables starting with the letter ‘g’.
9. Expand the **gb_result** tree, then add ***gb_result.data*** and ***gb_result.ret_code*** to the **OUTPUT** list by selecting them and clicking the **Add (O)->** button near the OUTPUT list.
10. Click **OK**.

You’ve now created the *func1_data.csv* CSV file to be used in exercise 1.

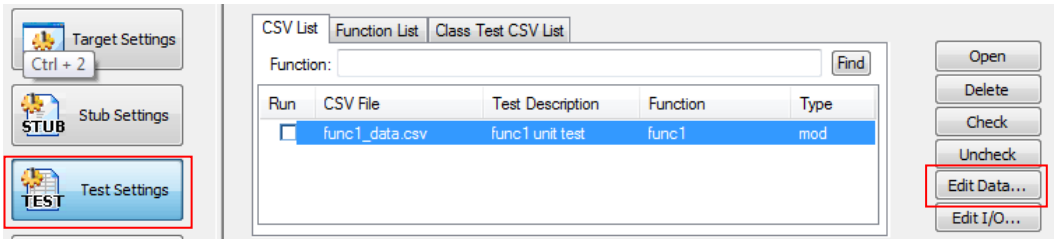


Note: When adding variables from the variable list, you may select and add multiple variables at a time by using the SHIFT or Ctrl key to make a multiple selection.

Open the CSV File You Created

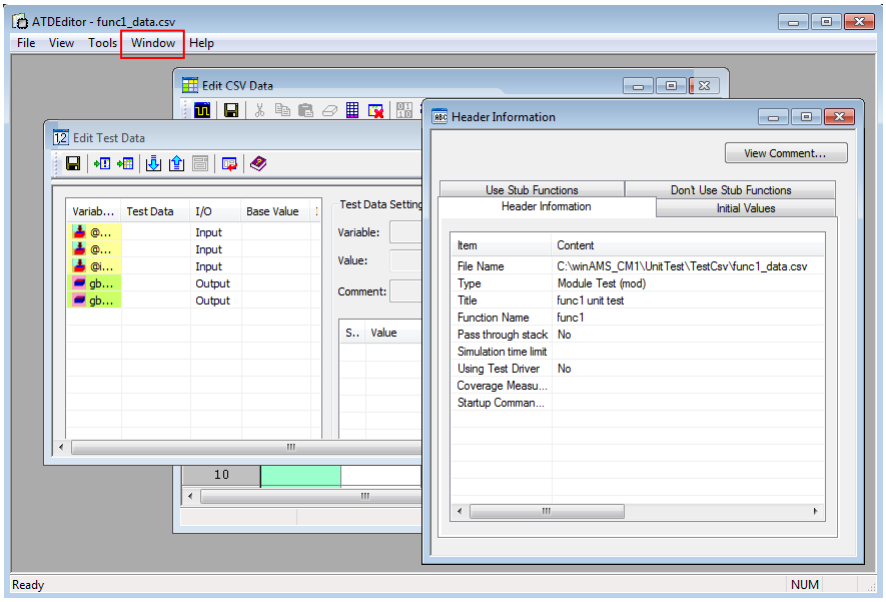
Display the **Test Settings** screen to show a list of CSV files.

11. On the left side of SSTManager click the **Test Settings** button.
12. Click on **func1_data.csv** to select it, then click the **Edit Data** button.

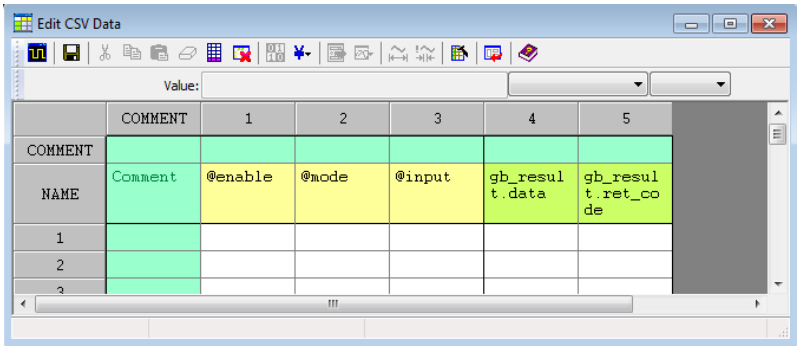


Now open the Edit CSV Data window in the test data editor (ATDEditor).

[NOTE]: By default, the more complex Test Data Analysis Editor might open instead of the basic Test Data Editor. That editor is covered later in this tutorial. In this case, disable the Test Data Analysis Editor by clicking the Tools menu, selecting “Test Data Analysis Editor Settings” and disabling the top option “Use the Test Data Analysis Editor for unit test…”. Then click OK, delete the created CSV file and re-create it from start.

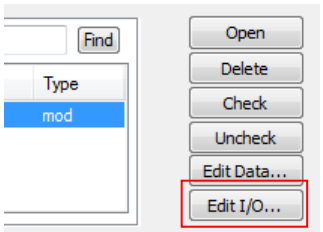


13. From the ATDEditor menu click **Window** -> **Edit CSV Data**.



CSV stands for Comma-Separated-Value, and is simply a text file that may also be opened with MS Excel or a text editor. Even so, in order to minimize errors it is recommended to use winAMS's CSV creation dialog rather than editing the file directly when changing the variable names or type.

In order to edit the CSV file settings and variables, click the **Edit I/O** button located on the right of the CSV file list.



Enter Test Data into the CSV File

Next we're going to input test data for *func1()* into our recently created CSV file we opened in the previous step. First we're going to input some test data and then see what kind of coverage results we get from the *winAMS* simulation. Each row of data represents one execution of function *func1()*, so by entering 5 rows of data the function will be executed 5 times.

1. **Enter** the test data pictured below.
2. Leave the outputs for *gb_result* blank for now.
3. **Save** the changes made to *func1_data.csv.*
4. **Close** the test data editor (ATDEditor).

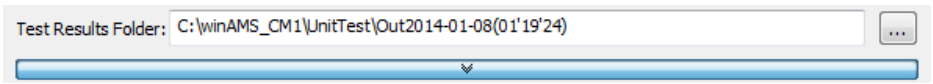
	COMMENT	1	2	3	4	5
COMMENT						
NAME	Comment	@enable	@mode	@input	gb_result.data	gb_result.ret_code
1		0	0	10		
2		1	0	10		
3		1	1	10		
4		1	2	10		
5		1	3	10		
6						

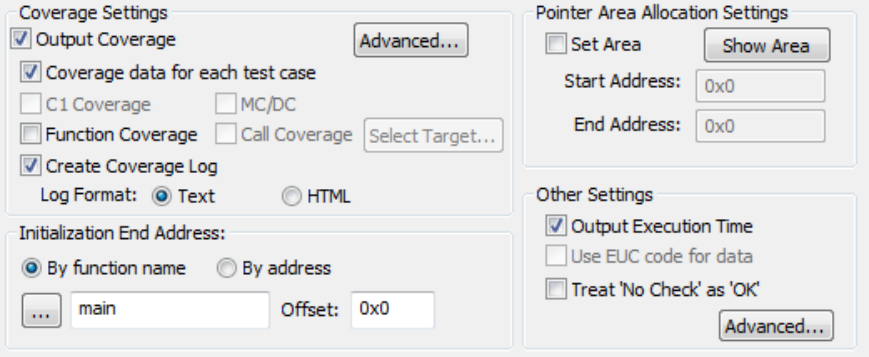
Run the func1() Test

Now let's run the test for *func1()* using the CSV file and test data we created. Before starting the simulator check the following settings in the **Test Settings** window.

1. **Check** the **Run** box to the left of *func1_data.csv* in the CSV list.
2. In the **Coverage Settings** section, check **Output Coverage**, **Show coverage path for each data set**, **Create Coverage Log**, **Log Format: Text**.
3. In the **Initialization End Address** section, select **by Function Name**, **main**, and **0x0** for the offset.
4. In the **Other Settings** section, check **Output Execution Time**.
5. All other settings in the Test Settings window should be unchecked (refer to the image below).

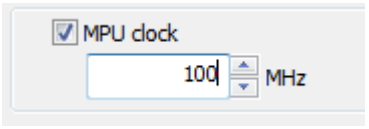
Note: If the above mention settings are not visible, click the long horizontal bar shown below:





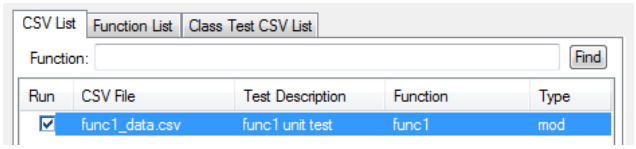
The **Output Execution Time** setting in the Other Settings section is optional. This setting when turned on outputs to the CSV file how long it would take the MPU device to execute the test data. This calculated time is only the amount of time it would take the device of set MPU clock (see Target Settings) to execute the code. Since it does not take into account other factors such as memory read/write time, caching, etc. the execution time on the actual device may differ.

The MPU clock setting set in the **Target Settings** window is used to calculate the execution time. For this exercise we are using the setting of 100 MHz (the speed of the device we selected).

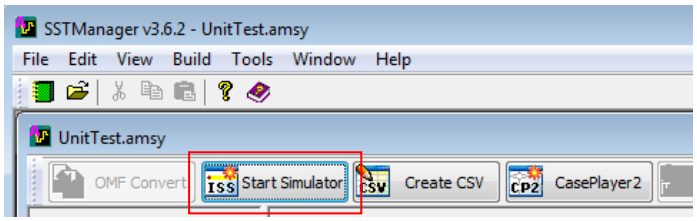


Now start the simulator:

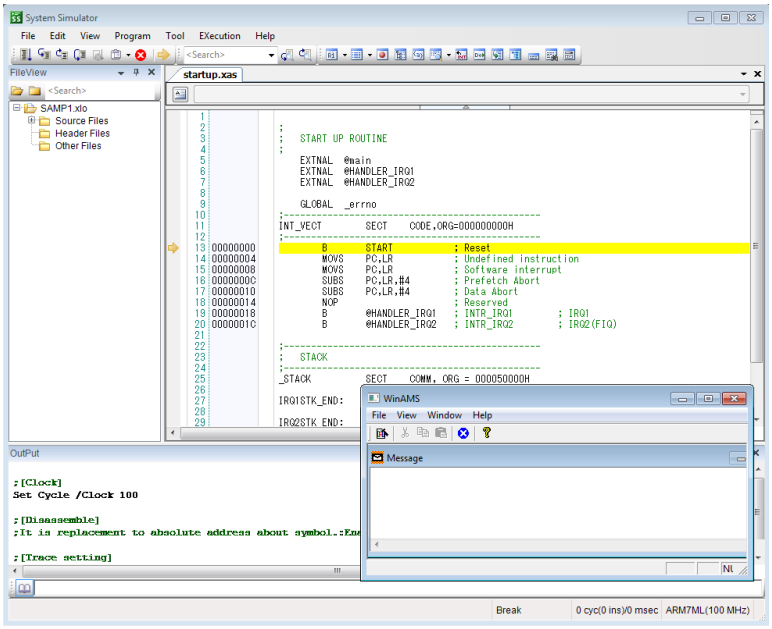
1. Check **func1_data.csv** in the **Test Settings** view.



2. Click the **Start Simulator** button located in the upper portion of SSTManager.

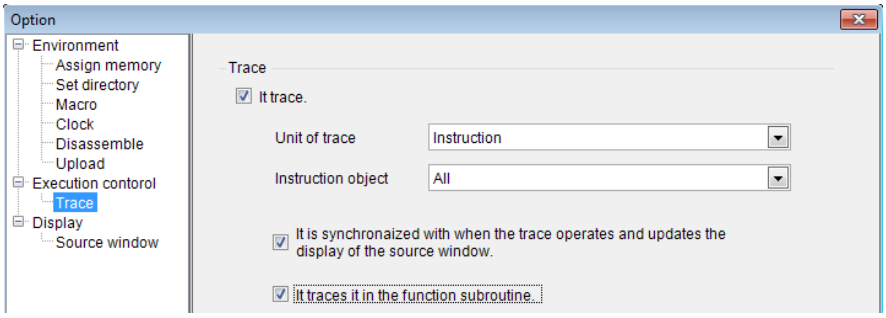


3. The **System Simulator** (MPU simulator) window and **WinAMS** (unit test application) window will appear.
4. **Minimize** the **WinAMS** window since we won't be using it at this point.



In order to trace the execution line during the testing, enable the trace feature of System Simulator.

1. From the **System Simulator** window click **Tool** -> **Option** from the application menu.
2. Click **Execution control -> Trace**.



3. Enable all checkboxes like above.
4. Press the **Application** button and close the dialog.

Let's start testing.

5. Click **Execution** -> **Execution** from the application menu.

During the simulation the trace will highlight the portion of code being executed in yellow. The simulation has been completed once the trace has stopped at the *main()* function.

Close the simulator to and check the test results:

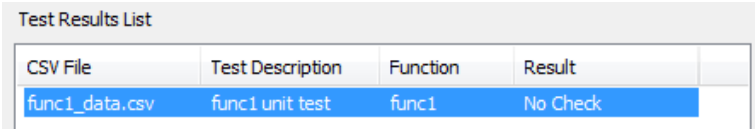
1. From the MPU simulator (System Simulator) select **File** -> **Exit**.

Running simulations in trace-mode actually slows down the simulation because the code display must be updated. We turned on the trace-mode in this example to become familiar with the feature, but for the remainder of the exercises it will be turned off.

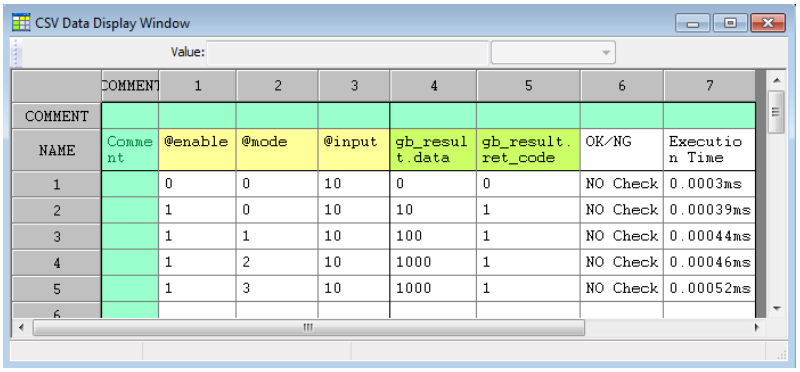
Check the Results of the *func1()* Test

Now that we've successfully completed the *func1()* test let's check the results.

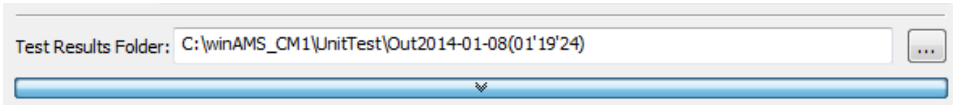
1. Click the **Test Results** button on the left side of SSTManager.
2. Double click **func1_data.csv**.
3. The test results CSV file will be opened.
   (Note: The test results CSV and input CSV filenames are the same, but are stored in different folders).



The format of the output CSV file is the same as the input CSV file. The test data for the three input variables has not changed, but now the test result values for *gb_result.data* and *gb_result.ret_code* are shown. Since no expected values were entered for the output variables in the input CSV file, the result in column seven will show **NO Check**. If expected values are entered the result will be either OK if the match or NG if they don't match.



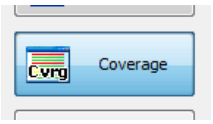
The test results CSV file folder may be changed in the **Test Settings** window. By default, the time stamp when creating the project will be applied to the folder name:

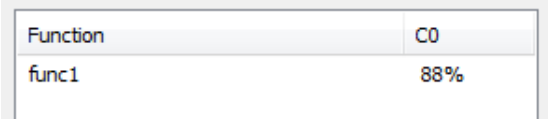


Check the Coverage for *func1()*

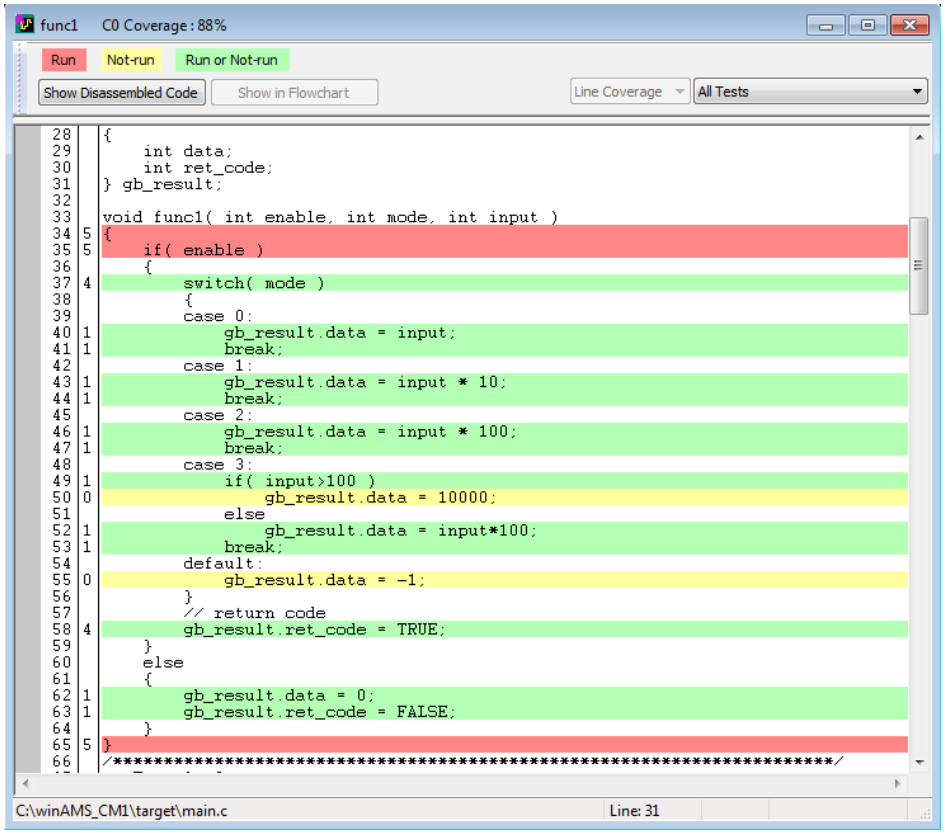
Next let's check the **C0 Coverage** for *func1()*.

1. Close the test result CSV file (*func1_data.csv*) in Excel.
2. Click the **Coverage** button on the left side of SSTManager.
3. Under the C0 column it should display **88%** for **func1**.





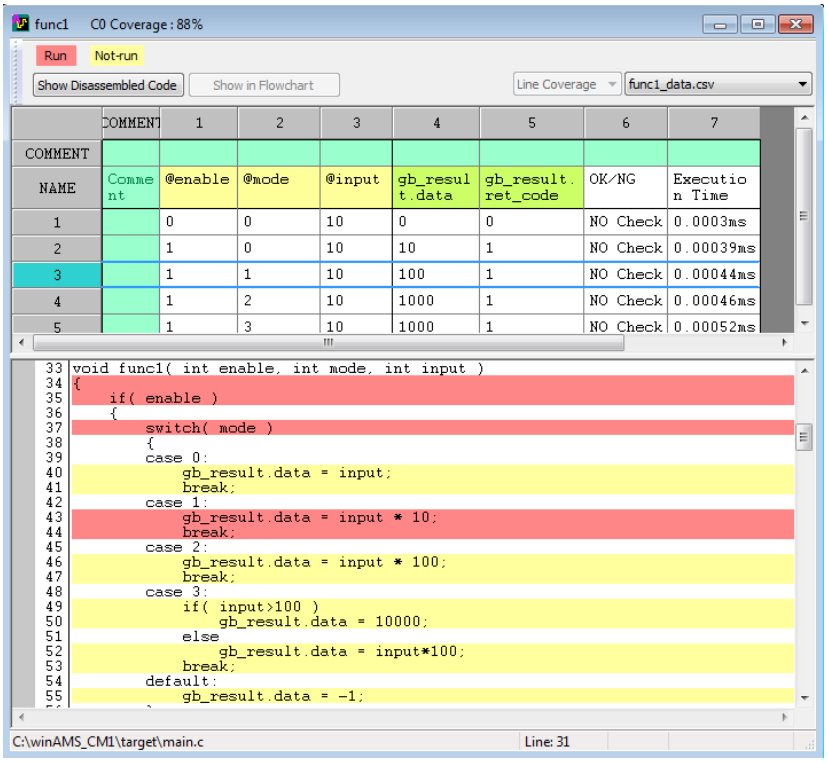
4. **Double-click** on **func1** in the function list to display the Coverage Viewer.



This is a portion of the C0 coverage result of *func1()*. As the legend at the top of the screen shows, the yellow highlighted code indicates code not executed, red indicates code that was executed by all test data sets, and green indicates code that was executed by at least one test data set. The number appearing to the right of the row number indicates the number of times the line of code was executed.

Next we're going to display the coverage view alongside the test data sets.

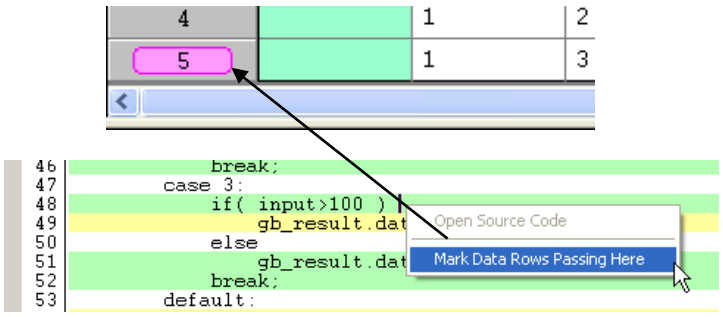
2. At the upper right portion of the Coverage View window change the selection from **All Tests** to ***func1_data.csv***.
3. This will cause the test result CSV file to appear in the upper portion of the Coverage View window (if not visible try adjusting the window size or by scrolling).



Try clicking on one of the test data rows. Once selected the test data row will become highlighted, and the coverage path for that data set will be shown in red in the coverage display. If a comment row or name row is selected the coverage for the entire test set will be displayed.

In addition, it is possible to highlight the test data that passes through a selected line of code.

4. **Right-click** on row #49, *if(input > 100)*.
5. Select **Mark Data Rows Passing Here** from the popup menu.
6. Line 5 of the test data will then be marked as shown below.



This means that test data line 5 is the only set of test data that passes through the selected if statement.

Looking at line #50 of the code we'll notice it is highlighted in yellow meaning it was not executed. Therefore, we need to come up with a value to insert into the test data so that this row will be executed. Using test data row 5 as our base (since it passed through the if statement), we simply need to enter a value greater than 100 into the input variable to make the if statement true.

The above are simple examples of analyzing features available from the coverage results. For unit testing with more complex source code (such as multiple nested condition statements), these

analyzing features become even more useful for determining what test data to enter.

The Relationship between the Code and the Coverage

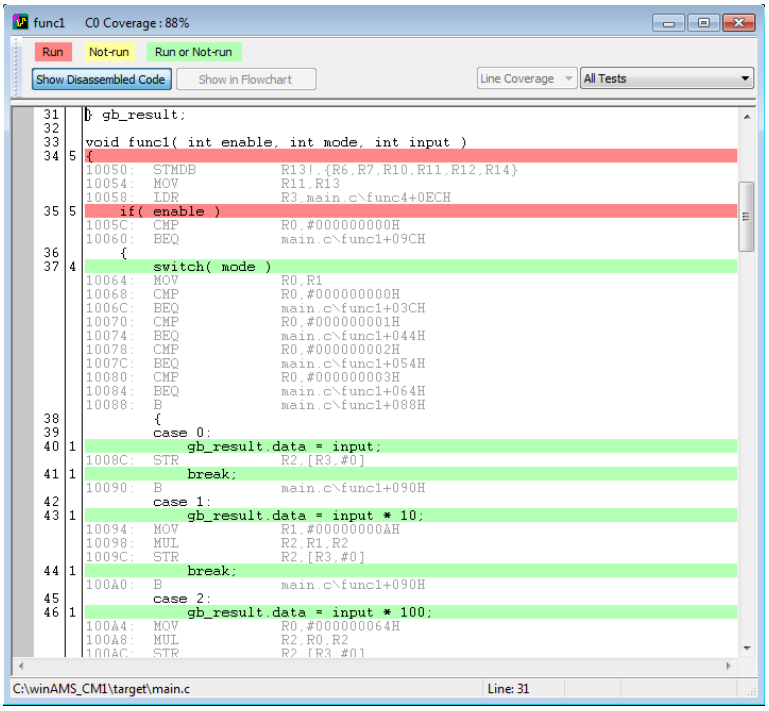
CoverageMaster winAMS executes the actual MPU code in order to perform unit testing and coverage testing. This section will explain the correlation between the MPU code and the coverage results.

1. Click the **Show disassembled code** button found in the upper portion of the coverage view.
2. The disassembled code (the MPU code) will be displayed in the coverage view.

The disassembled code comes from the debug information generated by the compiler. It is the same as a C / assembly “mixed display” of ICE debuggers and MPU simulators.

CoverageMaster winAMS displays the coverage by highlighting the C code that corresponds to the execution code (MPU code). When using compiler optimization features, the compiler may modify (optimize) the execution code in order to increase the execution speed. This may adversely affect the coverage results display.

Using the code below as an example, you’ll notice the variable ***int i*** is declared at the beginning of the function. However, since the variable *i* is not used in the function, it was removed due to compiler optimizations. In this case *int i* will simply be left blank and not receive coverage as shown below.

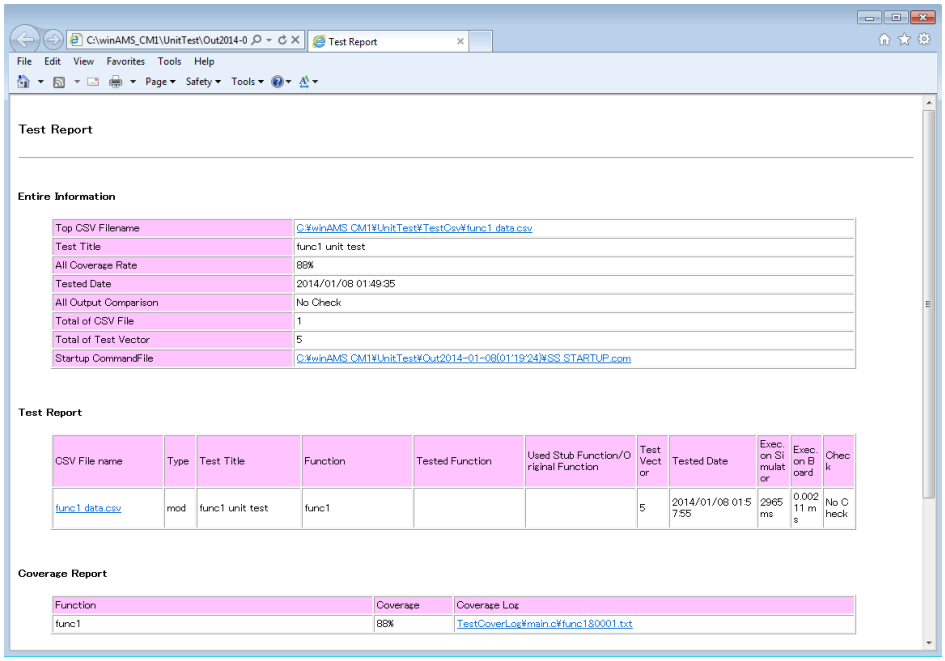


Like in the example above, it may be useful to show the disassembled code for unmarked lines when viewing the coverage results.

Open the Test Report File

Now *let's* open the coverage test report file.

1. Click the **Test Results** button on the left side of SSTManager.
2. Click the **Open Reports** button on the upper right part of the window.
3. The test report will be opened with your web browser as shown below.

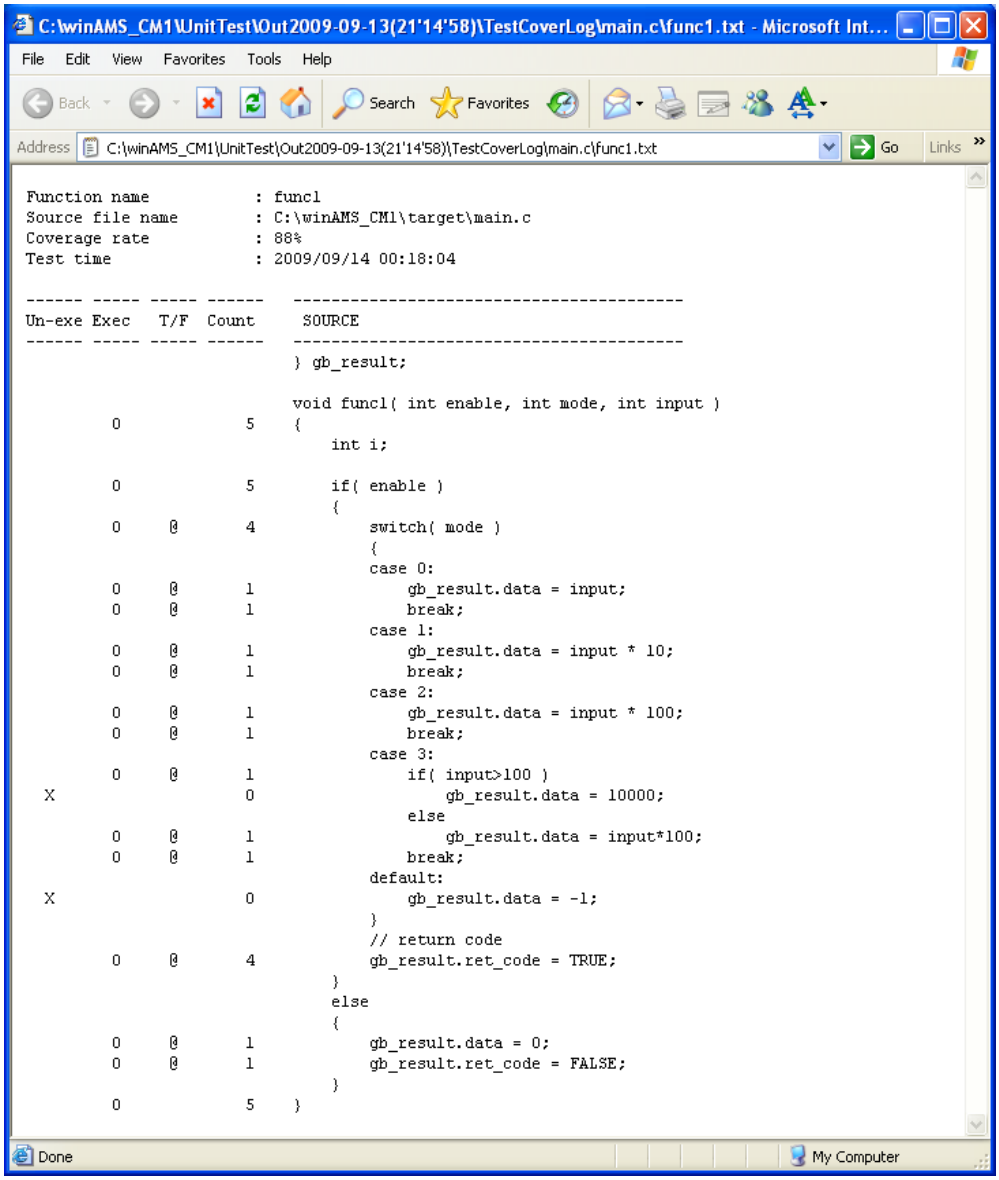


This HTML file contains a variety of information regarding the most recently performed test. These test report files are not intended for archival, rather temporarily stored for viewing the results of the most recent test.

The test report includes the following information:

Entire Information: The top CSV File, links to the input CSV files.
Test Report: CSV Files, links to the output CSV files.
Coverage Report: the tested functions and their coverage rate, links to the coverage log files.

An example of a coverage log text file is shown below. The log files may be output in either text or HTML format (configured from the Test Settings window).




```
Function name           : func1
Source file name        : C:\winAMS_CM1\target\main.c
Coverage rate           : 88%
Test time               : 2009/09/14 00:18:04

------ ----- ----- ------   ----------------------------------------
Un-exe Exec   T/F  Count     SOURCE
------ ----- ----- ------   ----------------------------------------
                            } gb_result;

                            void func1( int enable, int mode, int input )
        0             5     {
                                int i;

        0             5         if( enable )
                                {
        0     @       4             switch( mode )
                                    {
                                    case 0:
        0     @       1                 gb_result.data = input;
        0     @       1                 break;
                                    case 1:
        0     @       1                 gb_result.data = input * 10;
        0     @       1                 break;
                                    case 2:
        0     @       1                 gb_result.data = input * 100;
        0     @       1                 break;
                                    case 3:
        0     @       1                 if( input>100 )
  X                   0                     gb_result.data = 10000;
                                        else
        0     @       1                     gb_result.data = input*100;
        0     @       1                 break;
                                    default:
  X                   0                 gb_result.data = -1;
                                    }
                                    // return code
        0     @       4             gb_result.ret_code = TRUE;
                                }
                                else
                                {
        0     @       1             gb_result.data = 0;
        0     @       1             gb_result.ret_code = FALSE;
                                }
        0             5     }
```

Retest *func1()*, get 100% Coverage

Since the goal of exercise 1 is to get 100% C0 coverage, we're going to enter more test data, then run the test again. Also we'll include expected values this time so we can compare the results of the test to our expected values.

1. Click the **Test Settings** button on the left side of SSTManager to display the test settings window.
2. Select **func1_data.csv** and click the **Edit Data** button.
3. From the ATDEditor menu click **Window** -> **Edit CSV Data** to show the Edit CSV Data window.
4. Enter the new test data values and expected values as shown in the image below (outlined in red boxes).

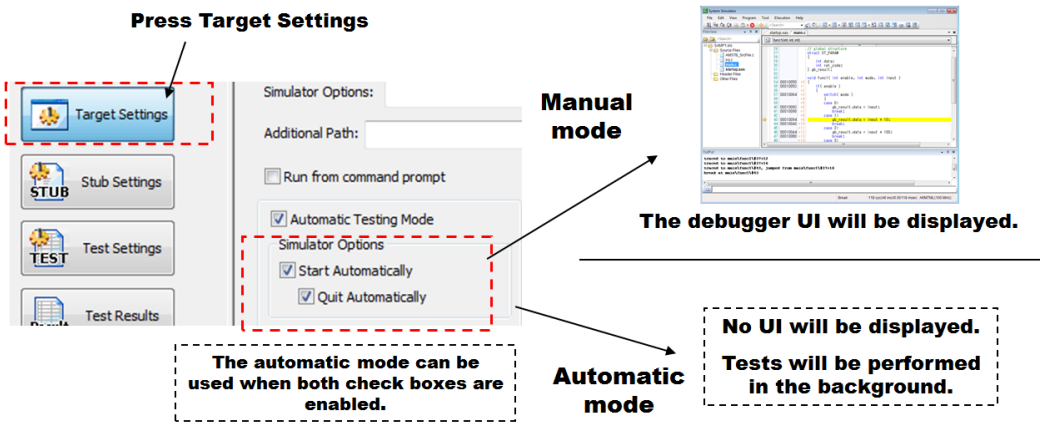
The expected values shown in the image below include some intentionally incorrect values so that you may see what the result will look like when the expected and actual values are different.

	COMMENT	1	2	3	4	5
COMMENT						
NAME	Comment	@enable	@mode	@input	gb_result.data	gb_result.ret_code
1		0	0	10	0	0
2		1	0	10	10	1
3		1	1	10	99	1
4		1	2	10	1000	0
5		1	3	10	1000	1
6		1	3	200		
7		1	4	10		

7. **Save** the close the ATDEditor.

Now *let's* run the test again. Last time we started the test manually, this time we will change the settings to run automatically.

8. Click the **Target Settings** button on the left side of SSTManager.
9. Check the **Automatic Testing Mode**, **Start Automatically**, and **Quit Automatically** settings.



Now the simulation will be run automatically from start to finish simply by clicking the **Start Simulator** button.

Start the test.

10. Click the **Start Simulator** button on the upper portion of SSTManager.
11. The test will be performed automatically and then display the test results window.

Since we turned off trace mode the simulation should completely quickly. Now let's check the test results and coverage.

12. **Double-click** on *func1_data.csv* in the **Test Results** window.

When output results differ from the expected values they will be displayed as *actual_result?(expected_result)* as shown in the image below. The test result will display *NG* if the actual/expected values differ, *OK* if they are the same, and *NO Check* if no expected values were entered.

	COMMENT	1	2	3	4	5	6	7
COMMENT								
NAME	Comment	@enable	@mode	@input	gb_result.data	gb_result.ret_code	OK/NG	Execution Time
1		0	0	10	0	0	OK	0.03ms
2		1	0	10	10	1	OK	0.039ms
3		1	1	10	100?(99)	1	NG	0.044ms
4		1	2	10	1000	1?(0)	NG	0.046ms
5		1	3	10	1000	1	OK	0.052ms
6		1	3	200	10000	1	NO Check	0.053ms
7		1	4	10	-1	1	NO Check	0.044ms

Also check the coverage results which should now be 100% for C0 coverage.

(Ref) To display the MPU simulator running in the background

As mentioned before, the MPU simulator window is not displayed when using the automatic execution mode. If the MPU simulator running in the background does not close automatically due to an error or an infinite loop, the simulator can be accessed by performing the following operations:

1. Click the '**L**' icon on the task tray.
2. Select Display from the pop-up menu.
3. Select **File** -> **Quit** in the **Lix** window to quit.

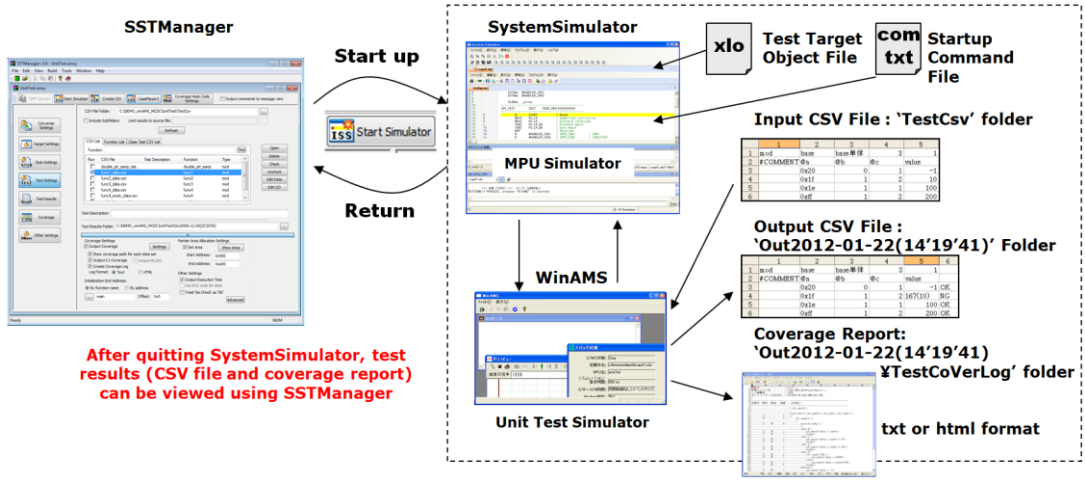
Do not use forced termination from the Windows task manager.

(Ref) Application Module Structure and Result File Location

This is a review of the unit test execution flow and test results output location. The unit test execution flow is as follows:

1. Start a unit test by pressing the **Start Simulator** button from SSTManager.
2. The MPU simulator (window name: SystemSimulator) starts up.
3. Unit test simulator (window name: WinAMS) starts up.
4. MPU simulator starts unit test execution.
   - When using automatic mode: (Automatically starts)
   - When using manual mode: Select **Execution** -> **Execution**
5. The test cases are loaded and the target function is executed.
6. The test results are output to a CSV file.
7. The coverage report file (txt/html) is created.

8. Quit the MPU simulator.
   - When using automatic mode: (Automatically quits)
   - When using manual mode: Select **File** -> **Application Exit**
9. The test results can be viewed from SSTManager.



The input/output file location is as follows:

- Input CSV File: [Test project folder]¥TestCSV
- Result CSV File: [Test project folder]¥Out[Creation Date][Creation Time]
- Coverage Report: [Test project folder]¥Out[Creation Date][Creation Time]¥TestCoverLog

Exercise 1 Conclusion

We've now completed our first test with CoverageMaster winAMS. In this exercise we learned the fundamentals of using CoverageMaster winAMS such as creating CSV files, inputting test data, running the simulation, and viewing the test results / coverage.

Exercise 2: Unit Testing a Function with Pointers

For our next practice exercise we will simulate a function that includes pointers.

Pointers are often used to hold an address for an object that resides outside of the function. Also pointers are commonly used to hold the address of a data table (array) used throughout the entire application. However, when unit testing functions it is more efficient to input data directly to the function rather than using a data table.

For this exercise we will use CoverageMaster winAMS's **Automatic Allocation** feature for pointers, and then input the necessary test data.

func2() Sample Code

For this exercise we will be testing *func2()* located in the sample file *main.c*. This function uses pointers frequently, but before worrying about that first let's identify the input and output variables.

```
// Ex. 2 : Function with Pointers
//         Get 100% C0 Coverage

// global variables
char *gb_port1 ;  // i/o port input
int  *gb_data_out; // result

void func2( int mode, int *data_in )
{
    if( *gb_port1 & 0x00000001 ) // when LSB is 1
    {
        switch( mode)
        {
            case 0:
                *gb_data_out= *data_in ;
                break;
            case 1:
                *gb_data_out= *data_in * 10;
                break;
            case 2:
                *gb_data_out= *data_in * 100;
                break;
            case 3:
                if( *data_in > 100 )
                    *gb_data_out= 10000;
                else
                    *gb_data_out= *data_in *100;
                    break;
                default:
                    *gb_data_out= -1;
        }
    }
    else
    {
        *gb_data_out= 0;
    }
}
```

The following three variables are referenced in the function:

char *gb_port1 (global port variable),
int mode (function parameter), **int *data_in** (function parameter)

Two out of the three are pointers that point to individual variables.

For the output the following variable is used to hold the result:

int *gb_data_out (global pointer variable)

Now we can use this information in the CSV file creation.

Address Values and Test Cases Required for Pointer Variables

As an example, the following settings are required to use the pointer variable "**char *gb_port1**":

1) Set the address value to the pointer variable **gb_port1**.
2) Set test cases to the memory referred by **gb_port1**.

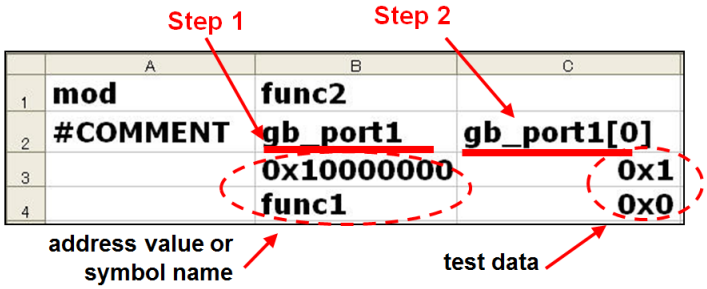
The address value may be given before func2() is called when the actual application is executed,

however it must be specified by the user before executing the func2() unit test. There are two methods to give address value to a pointer when using CoverageMaster.

Set address value to the pointer directly

In order to directly set an address value to a pointer, specify the pointer name in the CSV file as an input variable. In case of "**char *gb_port1**", enter "**gb_port1**" in the CSV cell, and enter the address value in hexadecimal (begin with "0x") as the test case value.

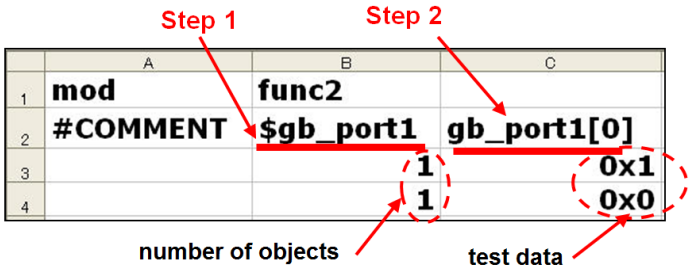
When using this method, the user must manage the address allocation according to the MPU's memory map. If the MPU type or memory map changes, the address value may need to be changed. After setting the pointer address, enter the pointer entity "**gb_port1[0]**" to the next cell to enter test cases.



Use the automatic allocation feature

CoverageMaster has an automatic pointer allocation feature for executing unit tests. This can be an easier method because the user does not need to manage the address value assigned to the pointer manually.

In order to automatically allocate memory to a pointer, add '$' in front of the pointer name when entering it in the CSV cell. In case of "**char *gb_port1**", enter "**$gb_port1**" in the CSV cell, and then enter the number of objects to allocate area for as the value. If the pointer is an array's start address, the number of array items should be entered for the number of objects. If zero is entered, the pointer will be NULL. After setting the pointer address, enter the pointer entity "**gb_port1[0]**" to the next cell to enter test cases.



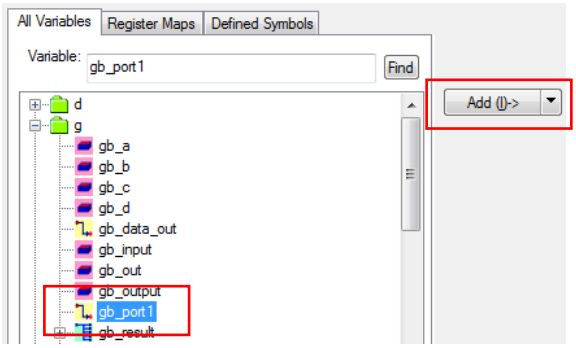
Create a CSV File with Pointer Auto Allocation

Now *let's* create the CSV file for exercise 2 using the same method we learned in exercise 1.

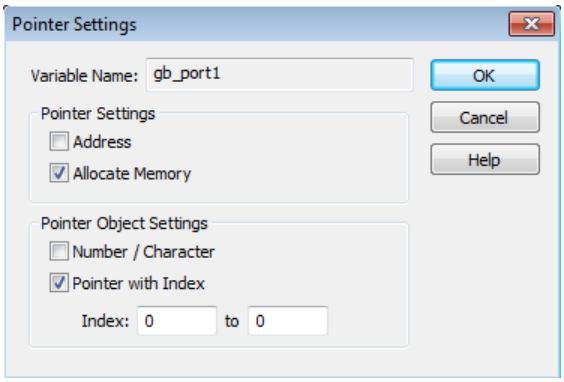
1. Click the **Create CSV** button towards the top of SSTManager.
2. Select **Unit Test CSV** and click **OK**.
3. In the Unit Test CSV Creation dialog enter **func2_data** for the filename.
4. Select **func2** as the function.

First add ***char *gb_port1*** to the INPUT list. Since this is a pointer we must also allocate memory for it.

10. Expand **g** in the variable list, then select ***gb_port1***.
11. Click the INPUT list's **Add (I) ->** button to add it to the INPUT list.



When adding a pointer the following dialog will appear.



Using *char *gb_port1* as our example each setting functions as follows:

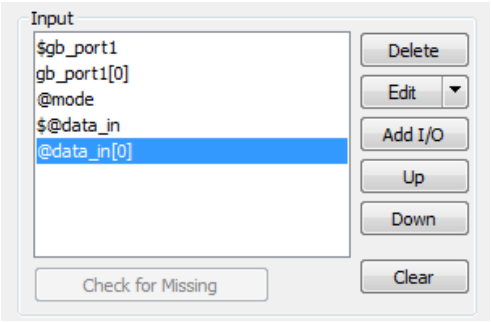
- **Address**: Use this setting when specifying an address to the pointer variable (*gb_port1* will be added).
- **Allocate memory:** Use this setting when you wish to allocate memory for the pointer ($gb_port1 will be added).
- **Number / Character**: Use this setting when the pointer points to an array and you wish to enter the entire array data into a cell. (*gb_port1 will be added).
- **Pointer with index**: Use this setting when you want to specify objects pointed to by the pointer (*gb_port1[0]*,…, will be added).

Allocate Memory and add *gb_port1* as Pointer with Index options.

12. Check both **Allocate Memory** and **Pointer with Index** (leave the index at 0 to 0), click **OK**.
13. **$gb_port1** and **gb_port1[0]** will be added to the INPUT list.

Next add the other input variables in a similar manner.

- ***@mode*** (function parameter): since it is of type int, it may be added to the INPUT list by simply selecting it and clicking the INPUT list's **Add (I) ->** button.
- ***@data_in*** (function parameter): since it is a pointer, add with **Allocate Memory** and **Pointer with Index** (leave the index at 0 to 0) settings (same as *gb_port1* above).

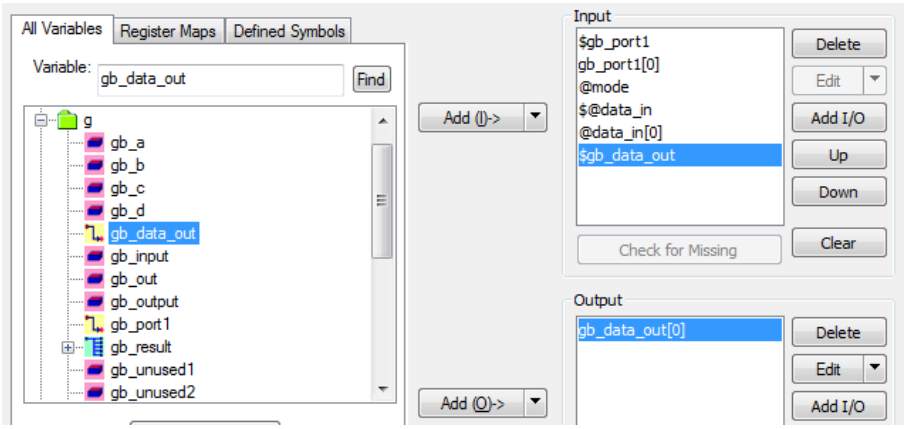


Next we're going to add our output ***int *gb_data_out*** to the OUTPUT list. But since this variable is a pointer we must first allocate memory for it in the INPUT list.

14. Expand **g** in the variable list, then select ***gb_data_out***.
15. Click the INPUT list's **Add (I) ->** button.
16. Check **Allocate Memory** and click **OK**.

Now we're going to add ***gb_data_out*** to the OUTPUT list so we can check the results and give it expected values.

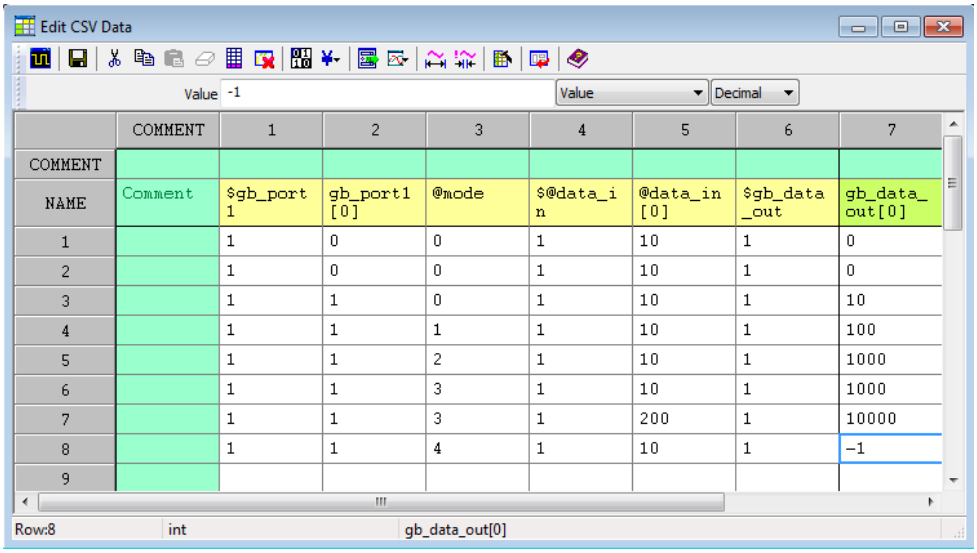
17. Expand **g** in the variable list, then select ***gb_data_out***.
18. Click the OUTPUT list's **Add (O) ->** button.
19. Check **Pointer with Index** (leave the index at 0 to 0) and click **OK**.



Now that we've completed the I/O variable settings click **OK** to create the CSV file. Next we're going to enter test data.

20. Go to the **Test Settings** window of SSTManager.
21. Select **func2_data.csv** and click the **Edit Data** button.
22. From the ATDEditor menu click **Window** -> **Edit CSV Data** to show the Edit CSV Data window.
23. Enter the test data values and expected values as shown in the image below.

Note: Because the pointers used in *func2()* are pointers to individual objects (as opposed to arrays), a ‘ 1 ‘ should be entered for the columns that begin with a dollar ‘ $ ‘ sign. These columns indicate the number of objects to allocate memory for, in *func2()*’s case only 1 is needed.



	COMMENT	1	2	3	4	5	6	7
COMMENT								
NAME	Comment	$gb_port1	gb_port1[0]	@mode	$@data_in	@data_in[0]	$gb_data_out	gb_data_out[0]
1		1	0	0	1	10	1	0
2		1	0	0	1	10	1	0
3		1	1	0	1	10	1	10
4		1	1	1	1	10	1	100
5		1	1	2	1	10	1	1000
6		1	1	3	1	10	1	1000
7		1	1	3	1	200	1	10000
8		1	1	4	1	10	1	-1
9								

For using GAIO's Development Environment with CoverageMaster winAMS

Unit Testing using the Pointer Memory Auto Allocation Feature

We're now ready to test *func2()* using CoverageMaster's pointer memory auto allocation feature.

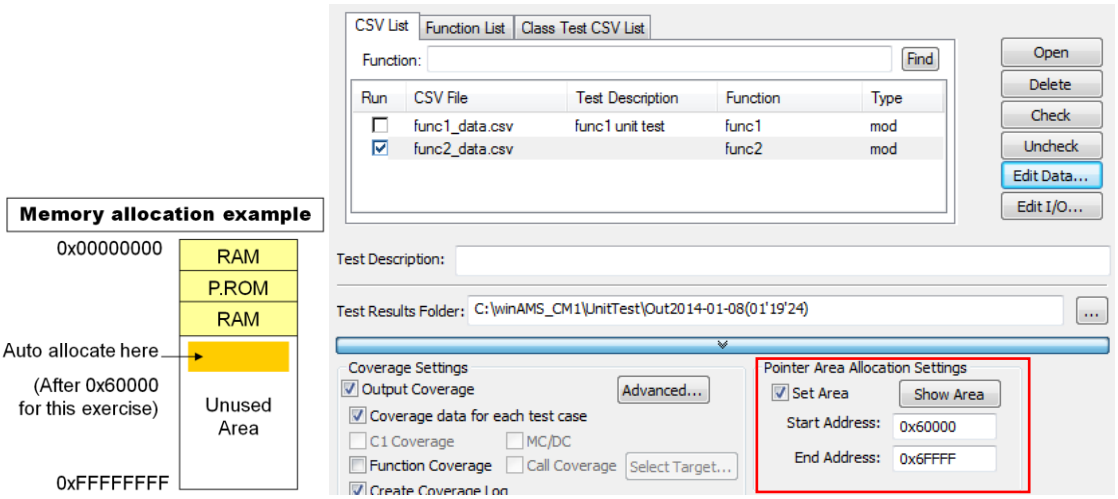
Note: Because the memory allocation settings differ according to the MPU device, please refer to the appropriate CoverageMaster winAMS*: 3rd Party Development Environment Usage Guide* when using an environment other than GAIO's.

1. On the **Test Settings** screen, **check** the Run box next to *func2_data.csv* (uncheck all other files).
2. In the **Pointer Area Allocation Settings** section, check the **Set Area** box.

Next, specify the address area used for the auto allocation feature. Set the address area to an unused address area where no memory or register is assigned (refer to the MPU device's memory map for unused areas). Even address areas not normally used by the device can be set because the area is only used for allocating area during simulation. The "Show Area" button may also be used for identifying unused memory areas.

In this example, specify 0x60000 as the start address and 0x6FFFF as the end address.

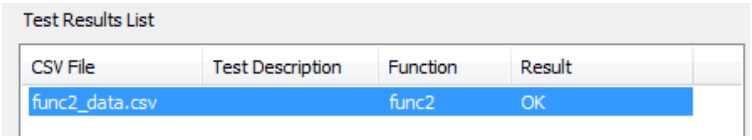
3. Start Address: **0x60000**, End Address: **0x6FFFF**.



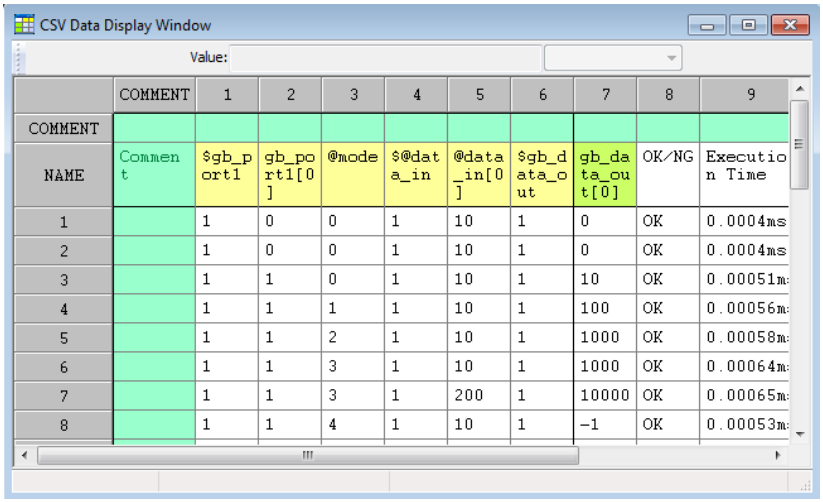
Now let's start the test.

- Click the **Start Simulator** button.

Once the simulation has been completed, the **Test Results** screen will be displayed.



Double-click on **func2_data.csv** to view the test results.



	COMMENT	1	2	3	4	5	6	7	8	9
COMMENT										
NAME	Comment	$gb_port1	gb_port1[0]	@mode	$@data_in	@data_in[0]	$gb_data_out	gb_data_out[0]	OK/NG	Execution Time
1		1	0	0	1	10	1	0	OK	0.0004ms
2		1	0	0	1	10	1	0	OK	0.0004ms
3		1	1	0	1	10	1	10	OK	0.00051m
4		1	1	1	1	10	1	100	OK	0.00056m
5		1	1	2	1	10	1	1000	OK	0.00058m
6		1	1	3	1	10	1	1000	OK	0.00064m
7		1	1	3	1	200	1	10000	OK	0.00065m
8		1	1	4	1	10	1	-1	OK	0.00053m

Since the correct expected values were entered the results should all appear as **OK**.

This concludes exercise 2: unit testing with pointers.

(Ref) How to Input Array Data into the CSV File

This section explains how to input data for an array into the CSV file. For a character array (string) the variable name begins with an asterisk ' * ' if declared as a pointer, and an ampersand ' & ' if declared as an array. Such character arrays (strings) may be entered by selecting the **Number / Character** option in the **Pointer Settings** selection dialog shown in exercise 2.

Character array (string) data must be enclosed within single-quotation marks (').
Example: 'abcdefg'

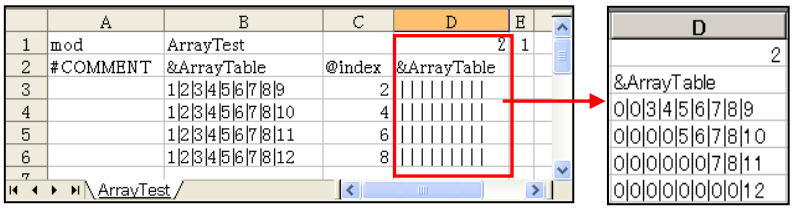
```
/*
   Character Array Data
*/
char outStr[64];

void StringTest( int
limit )
{
        outStr[limit] =
'¥0';
}
```

	A	B	C	D	E
1	mod	StringTest		2	1
2	#COMMENT	&outStr	@limit	&outStr	
3		'GAIO TECHNOLOGY'	4	'GAIO'	NO Check
4		'GAIO TECHNOLOGY'	6	'GAIO T'	NO Check
5		'GAIO TECHNOLOGY'	8	'GAIO TEC'	NO Check
6		'GAIO TECHNOLOGY'	10	'GAIO TECHN'	NO Check

StingTest

For number arrays, each value should be separated by a ' | ' mark as shown in the image below.

```
int ArrayTable[10];

void ArrayTest( int index )
{
  int i;
  if( index>10 ) index=10;
    for( i=0; i<index; i++ )
      {
          ArrayTable[i] = 0;
      }
}
```



	A	B	C	D	E
1	mod	ArrayTest		2	1
2	#COMMENT	&ArrayTable	@index	&ArrayTable	
3		1|2|3|4|5|6|7|8|9	2	|||||||||	
4		1|2|3|4|5|6|7|8|10	4	|||||||||	
5		1|2|3|4|5|6|7|8|11	6	|||||||||	
6		1|2|3|4|5|6|7|8|12	8	|||||||||	

ArrayTest

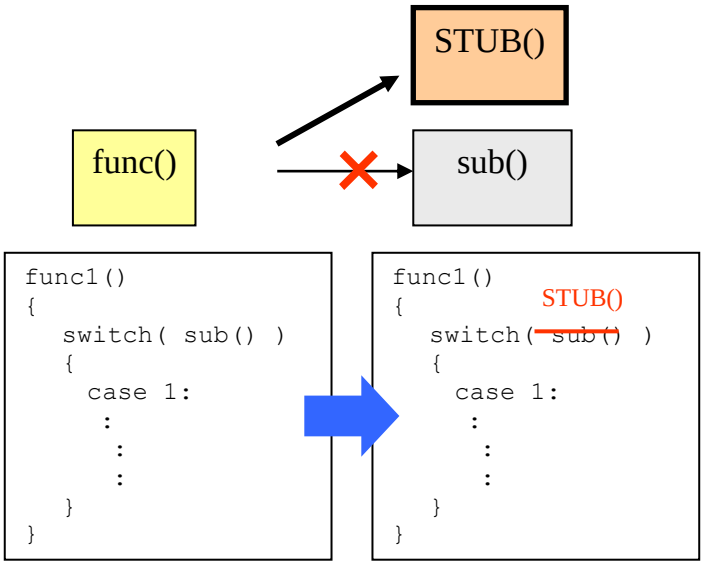
D
2
&ArrayTable
0|0|3|4|5|6|7|8|9
0|0|0|0|5|6|7|8|10
0|0|0|0|0|0|7|8|11
0|0|0|0|0|0|0|0|12

Exercise 3: Using Stub Functions

In our next exercise we're going to test a function that includes a call to another function.

What are Stub Functions?

Many functions include calls to other functions to perform various tasks. For example, say we have a function named *func()* that calls another function named *sub()* as represented visually below.



In the example above, the result of function *func()* is dependent upon the data return by the called function *sub()*. Therefore, if you wish to test *func()* as is, you will first have to find the appropriate input conditions for *sub()* in order to obtain the desired return value to be used in *func()*. In addition, because the two functions are dependent upon one another, both functions would need to be re-tested whenever either one changes. This is too complex and inefficient of a method.

In such cases when called functions exist, a stub function may be configured and used instead of the actual called function. In the example above, the function *STUB()* takes places of the function *sub()* allowing us to control the return value and focus on testing one function at a time.

CoverageMaster's Stub Feature

CoverageMaster winAMS includes features for using stub functions, including:

- Creating stub functions, and configuring them to interface with the test function.
- Replacing the called function with the stub function during testing.

The ability to substitute called functions with stub functions is an important feature in CoverageMaster. Using this feature the stub function may be called without having to modify the source code of the test function or the functions called by it.

Stub functions created with CoverageMaster winAMS will be contained in one source file. Because the stub function will be executed by the MPU simulator, it is necessary for the stub function to be compiled and linked.

Now *let's* begin the exercise and learn how to create and configure stub functions.

func3()

In exercise 3 we'll be testing ***func3()*** found in the sample code. The I/O variables are as follows:

Input Variables: *enable*, *mode* (function parameters)
Output Variables: *gb_result.data*, *gb_result.ret_code* (also a global structure member)

Within this function there is a call to another function, ***func3_sub_read_io()***. The return value of the called function is temporarily stored in the local variable ***retval*** that is then used as a switch statement condition. In order to get 100% C0 coverage for *func3()*, the return value must be modified to enter each case of the switch statement.

For this exercise we will create a stub function to replace *func3_sub_read_io()* and add the return value the INPUT list so we can control its value.

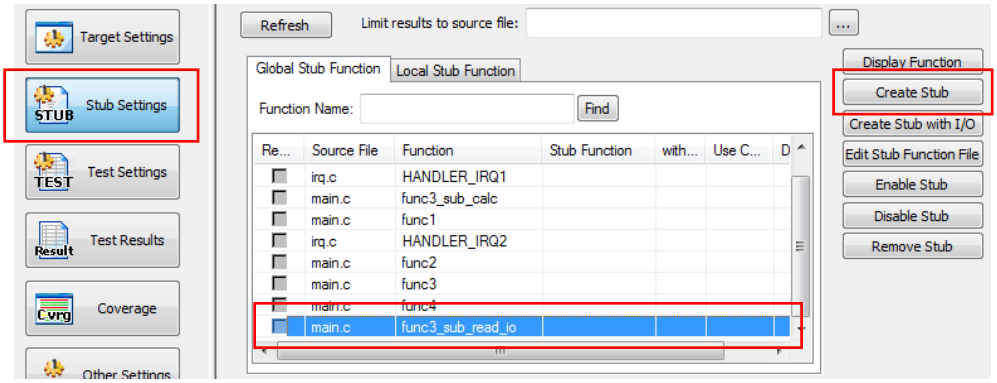
```
int func3_sub_read_io( int index )
{
  //Return value depends on data_table

    if( data_table[index]>0x7f )
    {
        return data_table[index];
    }
    else
    {
        return -data_table[index];
    }
}
```

```
void func3( int enable, int mode )
{
    int retval;

    if( enable )
    {
        // Create stub, make retval an input
        retval = func3_sub_read_io( mode );

        // Condition based on return value
        switch( retval )
        {
            case 0:
                gb_result.data = 0;
                break;
            case 1:
                gb_result.data = 50;
                break;
            case 2:
                gb_result.data = 100;
                break;
            default:
                gb_result.data = -1;
        }
        gb_result.ret_code = TRUE;
    }
    else
    {
        gb_result.data = 0;
        gb_result.ret_code = FALSE;
    }
}
```

Stub Function Creation and Settings

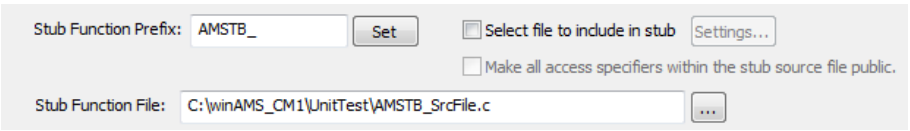
Now to create a stub function for *func3_sub_read_io()* using CoverageMaster winAMS's stub feature.

1. Click the **Stub Settings** button in SSTManager.



First let's discuss the settings. The **Stub Function Prefix** at the very top is the name that will appear before the stub function. For this exercise we will use the default of **AMSTB_** as the prefix so the stub function name will be **AMSTB_ func3_sub_read_io()**.

The next item down, **Stub Function Source File**, specifies the filename the stub function will be saved in. If the path doesn't match the image shown above, click the **…** button and change the path to **C:¥winAMS_CM1¥UnitTest**. The stub function source file **AMSTB_SrcFile.c** will now be saved in that directory.

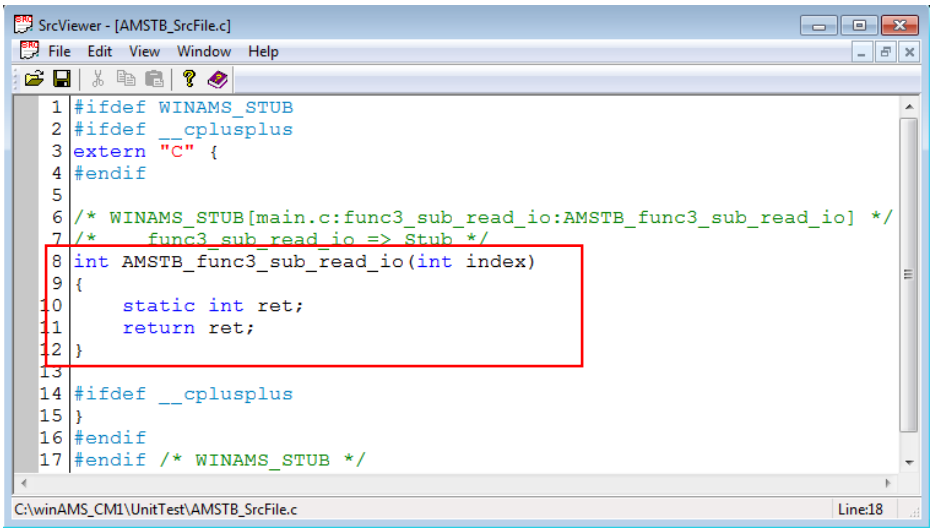


Now let's create the stub function.

2. Select **func3_sub_read_io()** and click the **Create Stub** button.

A stub function template for the function selected will be created with the specified name and then opened for editing.

Enter the code shown below into the stub function template. The code includes defining a static variable, and then returning that static variable.



Later we will add the static variable defined in the stub function ***(ret)*** to the INPUT list so we can enter test data values for it.

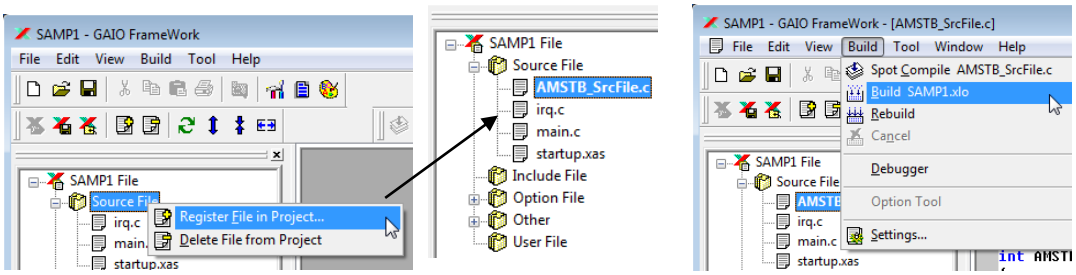
Only variables that have their addresses defined prior to the function's execution may be entered in CoverageMaster CSV files. This is why we defined the variable as static, to force the linker to allocate an address for the variable.

For using GAIO's Development Environment with CoverageMaster winAMS

Compile the Stub Function

Save the stub function. Next we're going to compile it into executable code using a cross compiler. In this section of the tutorial we'll be using GAIO's cross compiler, if you are using a 3rd party compiler please refer to the appropriate CoverageMaster winAMS*: 3rd Party Development Environment Usage Guide*.

1. Open the **C:¥winAMS_CM1¥target** folder.
2. Double-click the **SAMP1.gxp** project file to open in the project in GAIO's development environment (GAIO Framework).
3. From the file view on the left, expand **SAMP1 File** (project), right-click on the **Source File** (folder) and select **Register File in Project…**
4. Add **C:¥winAMS_CM1¥UnitTest¥*AMSTB_SrcFile.c***.
5. Note: Because *AMSTB_SrcFile.c* uses the compiler switch WINAMS_STUB, it must be added to the Preprocessor definition section of the compiler (it has already been done for you in the sample files).
6. Select **Build** -> **Rebuild** (or **Build**) from the menu, the stub information will now be compiled and linked for the test function.



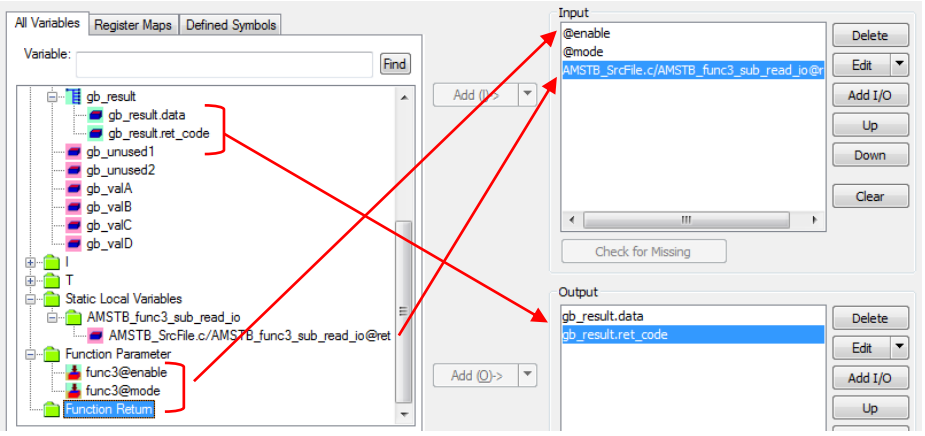
The stub function has now been added to the executable object *SAMP1.xlo*. You may now close GAIO Framework (GAIO's cross compiler).

Create *func3()* Test Data

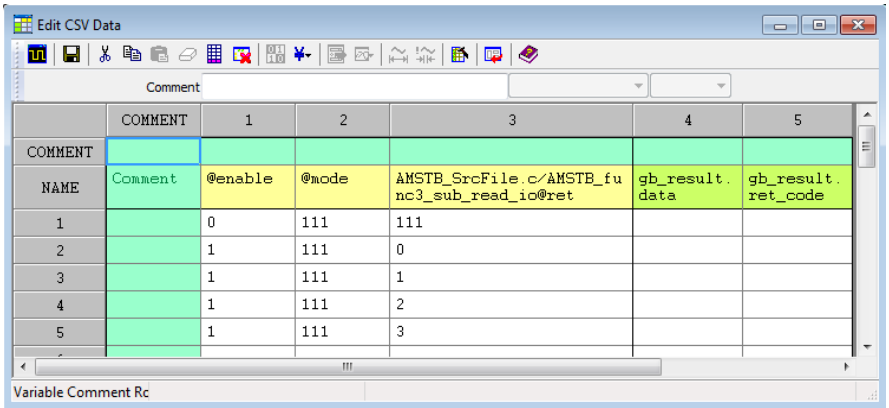
Next let's create a CSV file and enter test data for *func3()* like we did for the previous exercises. First identify the input and output variables.

Input: *int enable, int mode*, *ret*
Output: *gb_result.data, gb_result.ret_code*

1. Click the **Create CSV** button toward the top of SSTManager.
2. Select **Unit Test CSV** and click **OK**.
3. Filename: **func3_data**, Function: **func3**.
4. Add the I/O to the INPUT & OUTPUT lists as shown below.



5. Click the **OK** button to create the CSV file.
6. Go to the **Test Settings** window of SSTManager, select **func3_data.csv** and click the **Edit Data** button.
7. From the ATDEditor menu click **Window** -> **Edit CSV Data** to show the Edit CSV Data window.
8. Enter the test data values as shown in the image below.



The 3rd variable, ***AMSTB_SrcFile.c/AMSTB_func3_sub_read_io@ret***, is the stub functions return value we setup. Using this variable, we can choose what values will be returned by the subfunction.

The function parameter ***@mode*** in the original function is used to pass data to the called function. However, since we replaced the called function with a stub function, this function parameter will not actually be used and thus is optional for this test example.

'111' is entered as the value for test data cells that have no impact on the result of the test. The actual value does not matter and was chosen arbitrarily.

As you become more familiar with unit testing you'll learn to differentiate between test data that impacts the result and those that do not. This will lead to the ability to create efficient test routines that require fewer sets of test data to obtain 100% coverage.

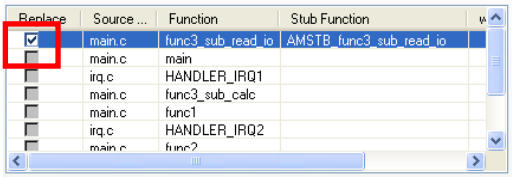
Testing using Stub Functions

Now to test *func3()* using the stub function and test data we created first we must tell CoverageMaster to use the stub function to replace the subfunction.

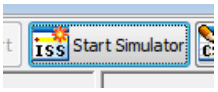
1. Go to the **Stub Settings** screen of SSTManager.
2. Check the **Replace** box for stub function ***func3_sub_read_io***.
3. Got to the **Test Settings** screen of SSTManager.
4. Select ***func3_data.csv***.

Start the test.

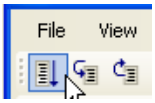
5. Click the **Start Simulator** button.



SSTManager



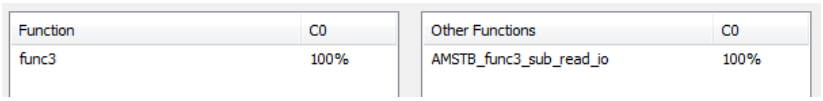
System Simulator



Go Button
(If not using the Auto Start/Quit settings)

Once the testing has been successfully completed click the **Coverage** button to go to the coverage screen.

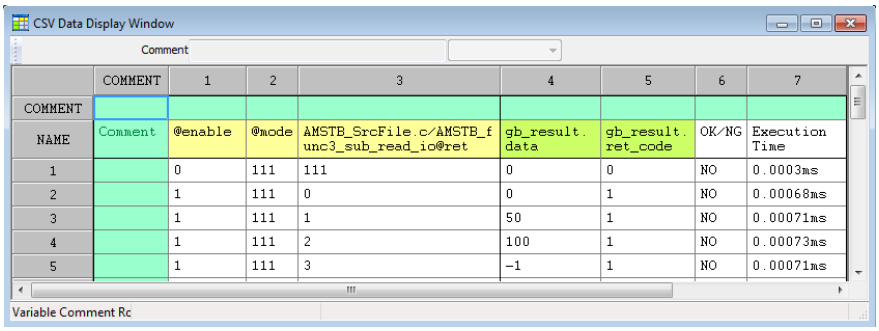
AMSTB_func3_sub_read_io should appear in the **Subfunction List** portion of the function list. The subfunction list displays subfunctions and/or stub functions called by the test function. This is a convenient way for confirming that the stub function was properly called.



Function	C0
func3	100%

Other Functions	C0
AMSTB_func3_sub_read_io	100%

Now switching to the **Test Results** screen, opening the test results CSV file should reveal the following results:



	COMMENT	1	2	3	4	5	6	7
COMMENT								
NAME	Comment	@enable	@mode	AMSTB_SrcFile.c/AMSTB_func3_sub_read_io@ret	gb_result.data	gb_result.ret_code	OK/NG	Execution Time
1		0	111	111	0	0	NO	0.0003ms
2		1	111	0	0	1	NO	0.00068ms
3		1	111	1	50	1	NO	0.00071ms
4		1	111	2	100	1	NO	0.00073ms
5		1	111	3	-1	1	NO	0.00071ms

This concludes exercise 3, using stub functions.

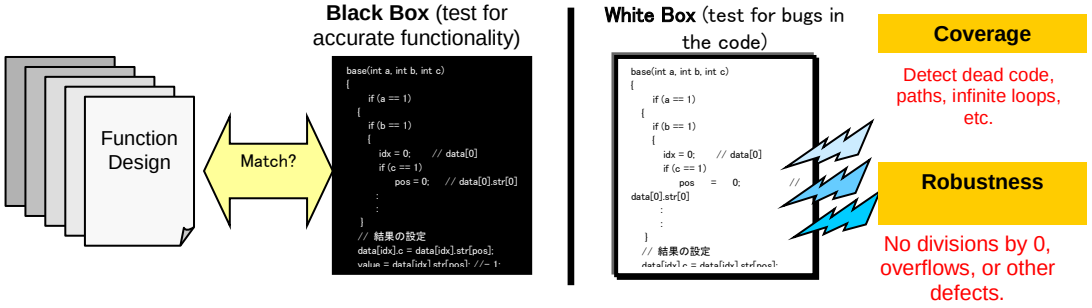
C1 / MCDC Coverage - Test Data Creation Feature

Exercise 4

In the previously 3 exercises we began by identifying the input and output variables, and then created test data for the unit test ourselves. In this exercise we will discuss a method to automatically create the test data.

When testing to see that the source code matches with the function's specifications, test data should be created by looking at the function's specifications rather than the code itself. This method is referred to as "black box" since you create test data without looking at the code inside "the box". "White box" on the other hand refers to creating test data while looking at the source code.

Thorough unit testing should also include testing for robustness such that errors are not caused by exceptional conditions. In addition, coverage testing is important to help detect "dead code" not actually necessary for the application. These verification factors are all important in finding latent bugs and improving software quality.



CoverageMaster winAMS can be linked with GAIO's **CasePlayer2** analysis tool to add the following functionalities:

- Automatically identify test input / output variables
  By analyzing the target function with CasePlayer2, the I/O variables may be automatically detected. This is a particularly useful feature for projects with a large number and variety of variables.

- Create test data to satisfy C1 Coverage
  Through the use of CasePlayer2 the test data to satisfy the C1 coverage test can be automatically generated. By using CasePlayer2's analysis data, CoverageMaster winAMS can automatically detect the conditional statements (if, switch, etc.) for the function, then create test data to execute each condition of the conditional statements (including nests).

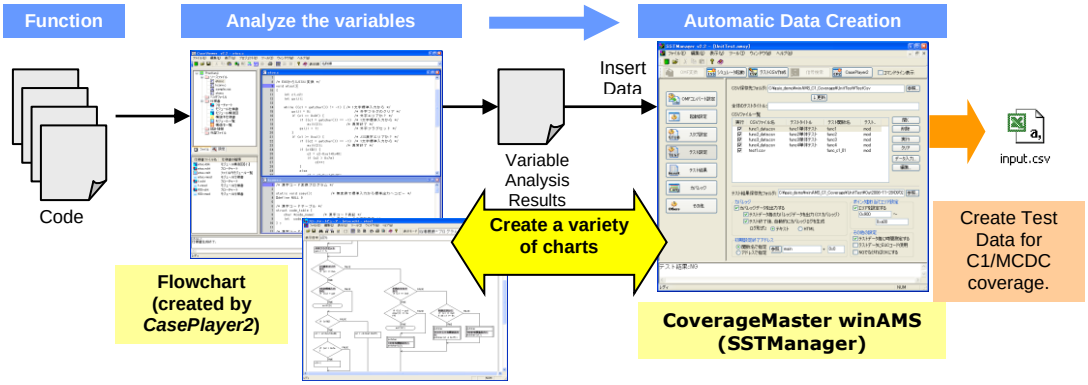
- Create test data to satisfy MCDC Coverage
  Similar to C1 coverage above, CasePlayer2 can also be used to create test data to satisfy MCDC coverage.

This type of testing is best suited for testing for dead code, unnecessary conditionals, and bugs in the C source file.

Linking with CasePlayer2

CasePlayer2 is a reverse CASE tool that can make flowcharts, module structure charts, and a variety of other specification documents by means of a source code analyzer. CasePlayer2 can also be used to create lists of global variables accessed by the function, look up variable references and assignments, and more when linked with CoverageMaster winAMS.

The CasePlayer2 to CoverageMaster winAMS linked relationship is outlined below.



In exercise 4 we're going to make use of CasePlayer2's capabilities to create C1 test data for the target function.

Exercise 4: Auto Generate Test Data with CasePlayer2

In this exercise, we'll discuss how to automatically create test data for a C0/C1 coverage test with the help of GAIO's CasePlayer2. We'll be testing *func4()* (shown below), with our goal being to obtain 100% C0/C1 coverage.

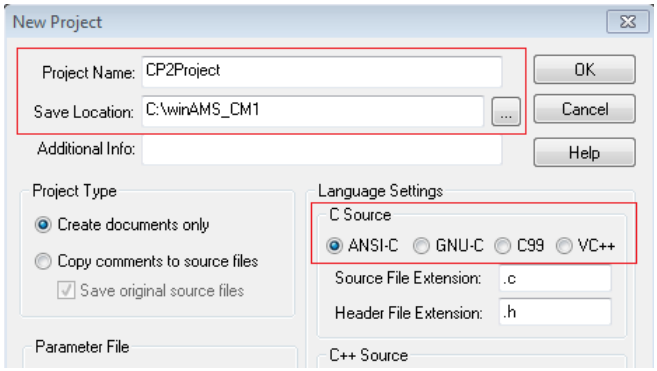
As you can see in the code below, *func4()* includes definitions for many global variables. Using CasePlayer2, first we'll identify only the global variables actually used in the function.

```
int gb_input;
int gb_output;
int gb_valA;
int gb_valB;
int gb_valC;
int gb_valD;
int  gb_a, gb_b, gb_c, gb_d, gb_out;
char gb_unused1, gb_unused2;

int func4( int code )
{
    int return_value=FALSE;
    int i;
    if( gb_a > 10 )
    {
        if( gb_b > 20 && gb_c > 30 )
        {
            gb_out = 0;
        }
        else
        {
            gb_out = -1;
        }
        return_value = FALSE;
    }
    else
    {
        switch( code )
        {
            case 1:
                gb_out = 1;
                break;
            case 2:
                gb_out = 2;
                break;
            case 3:
                gb_out = 3;
                break;
            default:
                gb_out = -1;
                break;
        }
        return_value = FALSE;
    }

// The conditional below uses the above
//  calculation result (gb_out)
    // Dynamic boundary values may not be
    //  statically analyzed.

    if( gb_d == gb_out )
    {
        gb_out = 4;
    }
    else
    {
        gb_out = 5;
    }

    return return_value;
}
```

CasePlayer2 Function Analysis

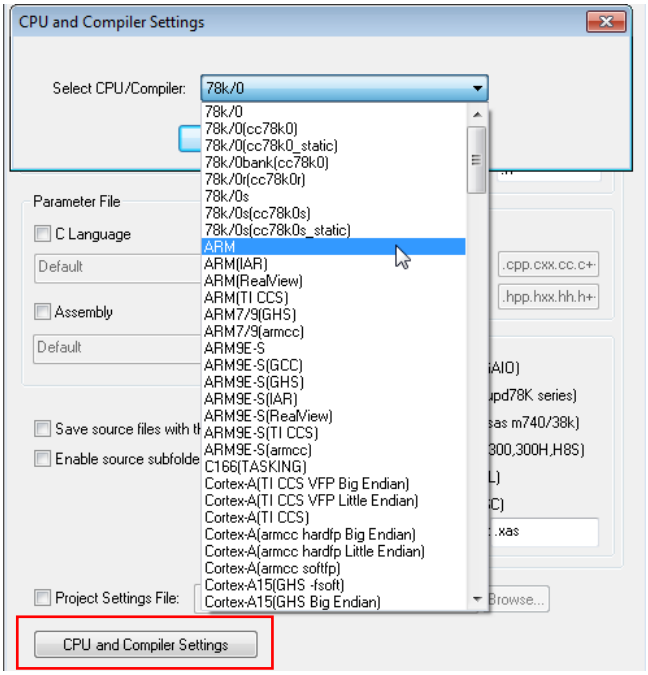
We must first create a CasePlayer2 project in order to analyze the source code.

1. Launch **CasePlayer2** by selecting: **Windows Start** -> **CasePlayer2** -> **CaseViewer.exe**
2. Create a project by selecting: **File** -> **New** - **Project**
   Project Name: **CP2Project**, Save Location: **C:¥winAMS_CM1,** Language: **ANSI-C**
3. Since we won't be using any of the other settings at this time you may leave them at their default values.



In addition, select the MPU and compiler type required for source code analysis settings.

4. Click the **CPU and Compiler Settings** button.
5. Select “ARM” from the pull-down menu.
6. Click OK to close the dialog.



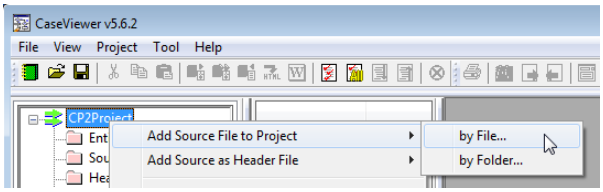
The **CPU and Compiler Settings** option is used for setting compiler specific analysis settings. CasePlayer2 cannot analyze compiler specific descriptions and will result in an analysis error if the proper analysis settings have not been set. For this purpose, CasePlayer2 has a C source code analysis customization feature “C option parameter”. By selecting the MPU and compiler type, common “C option parameter” settings used by that device will be automatically set.

This setting is not required when testing the tutorial sample code. However, this setting is recommended when testing user code so the feature is introduced here.

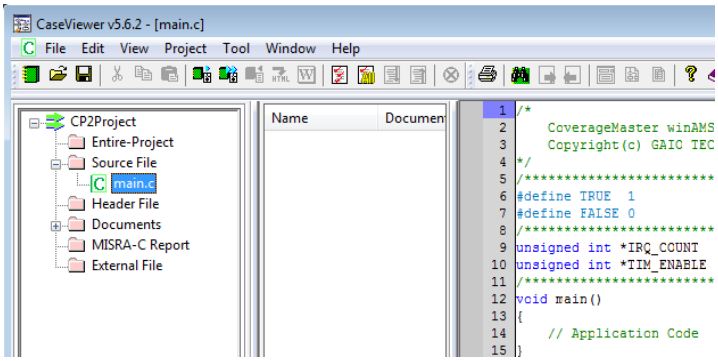
[NOTE] For more information, refer to the “C Option Parameter” section further down in this chapter.

Next select the source code to be analyzed. Since all test functions are included in *main.c* for these exercises we only need to select that file.

7. Click the **right mouse button** on **P2Project** in the project tree.
8. Select "**Add Source File to Project**" – **File**



9. Select **main.c** (C:¥winAMS_CM1¥target¥main.c).
10. Click the **Open** button to add the source file to the CasePlayer2 project.



CasePlayer2 Setting #1: Analysis Settings

Before performing a static analysis of the target function func4(), let's check the CasePlayer2 settings.

First, confirm the basic settings to use with CoverageMaster.

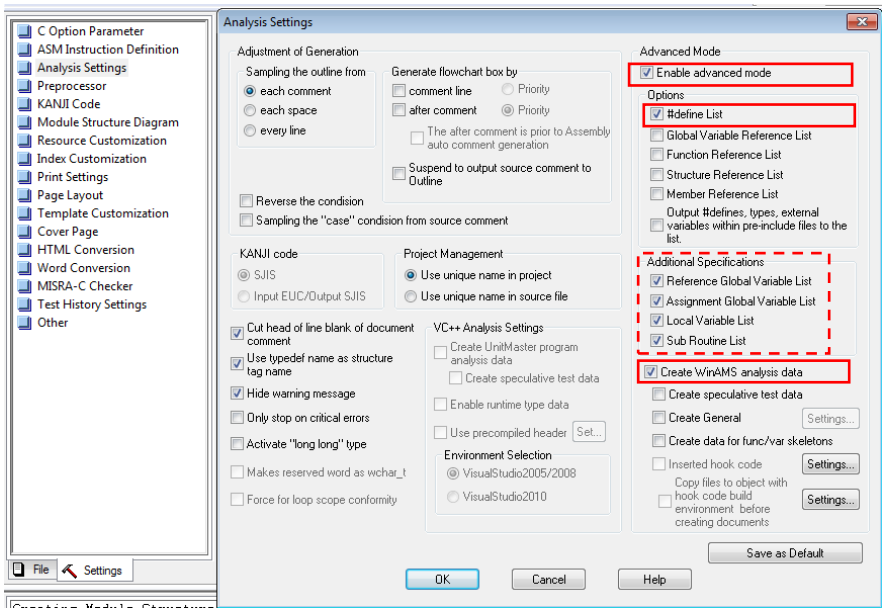
1. Double-click **Analysis Settings** in the CasePlayer2 Settings tab.
2. Enable the following options in the **Advanced Mode** settings group.

- Enable advanced mode
- #define List
- Create WinAMS analysis data

These three options must be enabled to use with CoverageMaster. Other options are settings for program document creation and can be enabled as needed. In order to create the global variable reference information for the **Module Specification** document mentioned later, add to enable the following options enclosed by the dashed line.

- Reference Global Variable List
- Assignment Global Variable List
- Local Variable List
- Sub Routine List

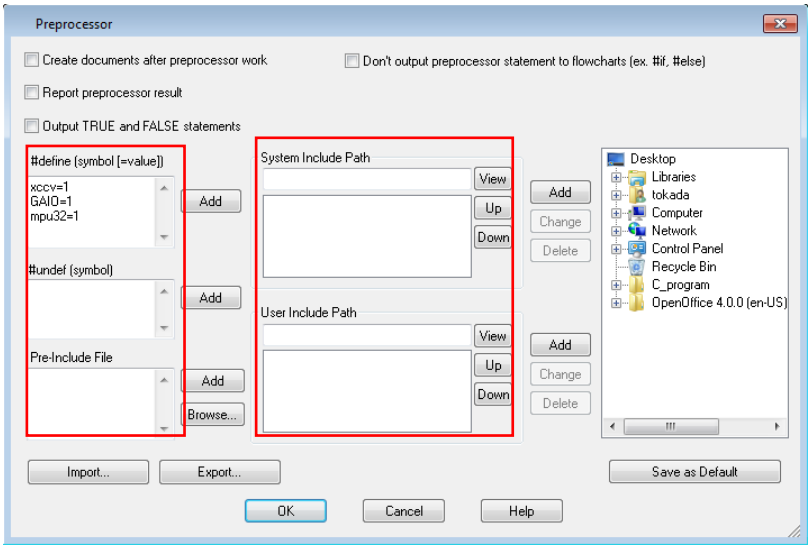
3. Press OK to close the dialog.



CasePlayer2 Setting #2: Preprocessor

Next, let's confirm the preprocessor settings to handle header files and #define parameters.

1. Double-click on "**Preprocessor**" in the CasePlayer2 Settings tab.



In order to analyze source code that includes header files, preprocessor settings configured in the IDE must be set in CasePlayer2 as well. These includes #define definitions and include path information.

#define (symbol [=value])
Define macros should be specified here. After entering an item click the Add button or press CTRL + ENTER in order to go to a new line to add additional items.

#undef (Symbol)
Existing #define macros may be undefined by entering them here.

Pre-Include File
Enter a header file here in order to include it before other source files. A pre-include file that describes implicit definitions may be entered in order to avoid CasePlayer2 analysis errors when the cross compiler has implicit key words or definitions.

System Include Path / User Include Path
Enter system include paths or user include paths in these fields. The paths may be selected using the tree view to the right then clicking the Add button. Since Subfolders are not included, each folder must be specified individually.

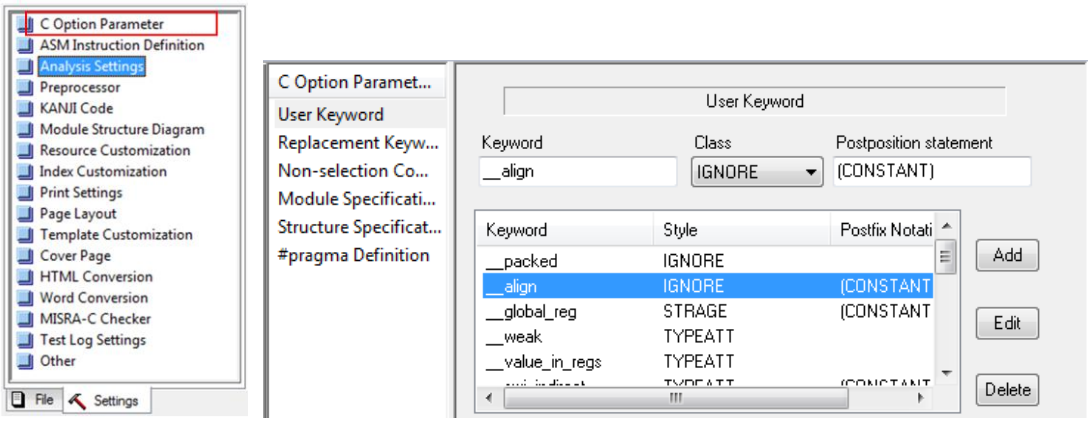
CasePlayer2 Setting #3: C Option Parameter

C-option parameters may need to be set in order to avoid analysis errors caused by incompatibilities in the code with the ANSI-C standard. CasePlayer2 analyzes C source code according to the selected language mode (ANSI-C / GNU-C / C99). However, CasePlayer2 does not recognize (reports as error) cross compiler specific code such as #pragma or language extensions.

By setting analysis methods to the C option parameter, incompatible descriptions in the code can be analyzed without source code modification.

Commonly used keywords will be automatically set in the C Option Parameter settings by selecting the MPU and compiler type when creating the CasePlayer2 project. Additional settings can be added manually as follows:

1. Double-click on **C Option Parameter** in the Settings tab.



(Ref) C Option Parameter Manual Setting

User Keyword
ANSI-C incompatibilities or compiler specific keywords may be added here. The following examples are typical settings to avoid analysis errors:

Example #1: int near p1;
Using the **TYPEATT** class for the **near** keyword will cause near keywords to be analyzed as a type instruction modifier.
Keyword: near, Class: TYPEATT, Postfix Notation: [**leave blank**]

Example #2: direct int array[100];
Using the **STRAGE** class for the **direct** keyword will cause the direct keyword to be analyzed as a storage class instruction modifier.
Keyword: direct, Class: STRAGE, Postposition statement: [**leave blank**]

Example #3: __asm (“ ..... “)
Using the **ASM** class for the __**asm** keyword will cause the __asm keyword to be analyzed as an inline assembler description.
Keyword: __asm, Class: ASM, Postposition statement: (EXPRESSION)

Example #4: __except (.....)
Using the **IGNORE** class for the __**except** keyword will cause the __except keyword to be ignored during analysis.
Keyword: __except, Class: IGNORE, Postposition statement: (EXPRESSION)

Example #5: INT32 val;
Using the **TYPE** Class for the **INT32** keyword will cause the INT32 keyword to be analyzed as a type definition.
Keyword: INT32, Class: TYPE, Postposition statement: [**leave blank**]

Replacement Keyword
Use to replace keywords in the code.

Example #1: typedef __WCHAR_T_TYPE__ _Wchart;
__**WCHAR_T_TYPE**__ can be replaced by **int** in order to avoid errors.
New Keyword: int, Existing Keyword: __WCHAR_T_TYPE__

Example #2: typedef __SIZE_T_TYPE__ _Sizet;
__**SIZE_T_TYPE**__ can be replaced by **int** in order to avoid errors.
New Keyword: int, Existing Keyword: __SIZE_T_TYPE__

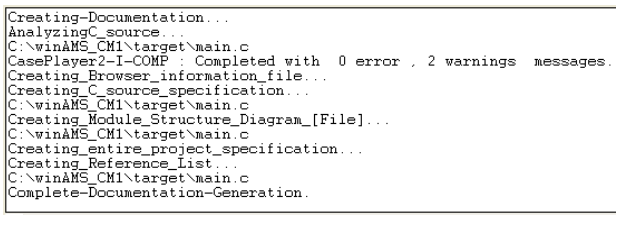
Other Settings in C Option Parameter
Most errors can be fixed using the above described User Keywords and Replacement Keywords. Additional settings found in C Option Parameter include:

Non-selection Comment
Module Specification/Sampling definition of Outline
Structure Specification/Sampling definition of Outline
#pragma Definition

Execute code analysis and program document creation

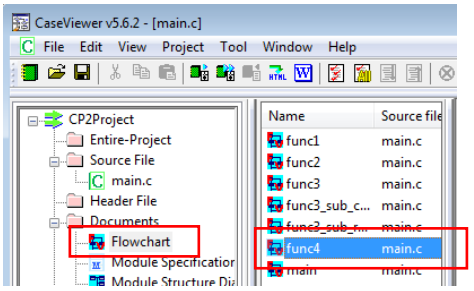
Next perform the code analysis / program document creation.

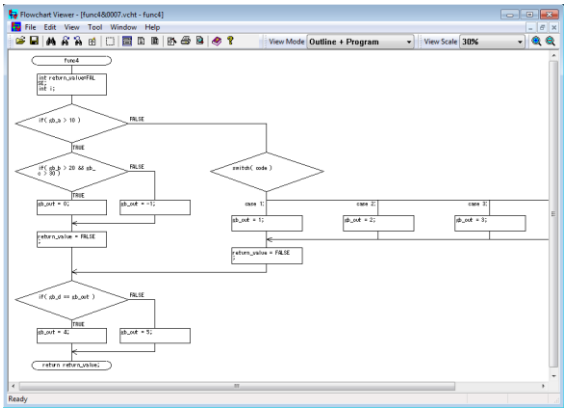
1. Select **Project** -> **Create documents** from the CasePlayer2 (CaseViewer) menu.
The following will appear in the message view at the bottom of the CaseViewer window.



Now that the analysis and program document creation has been completed, let's view some of the program documents that were created.

2. Click the File tab (lower-left part of the screen) to return to the CP2Project tree.
3. From the **Documents** folder select **Flowchart**.
4. From the list of flowcharts that appear to the right, double-click on **func4**.



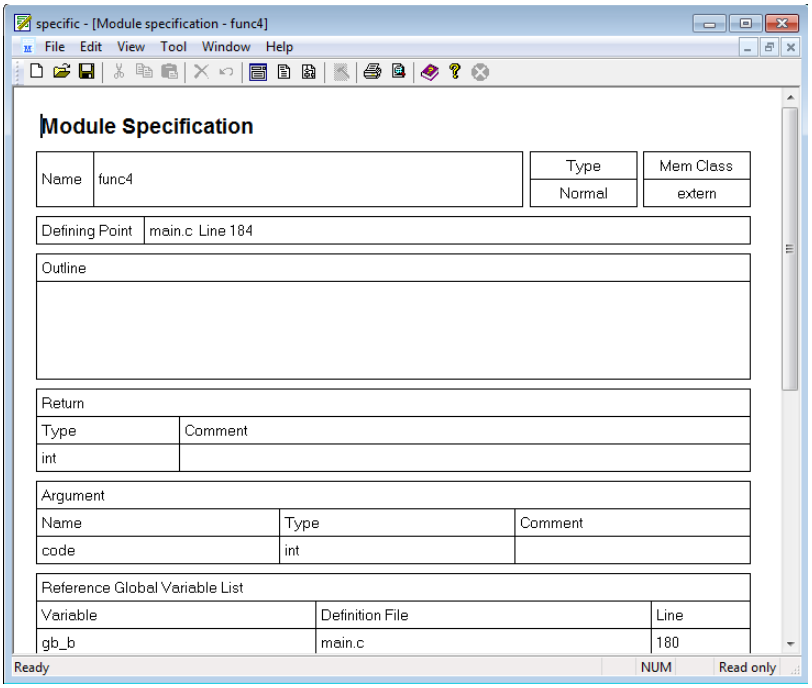


The flowchart displayed is for *func4()*, the target function for this exercise. Flowcharts are particularly useful for visualizing condition statements (included nests) of the code. Flowcharts are but one of the numerous types of specification documents creatable using CasePlayer2.

Next let's take a look at the analysis results by creating a **Module Specification**.

5. From the **Documents** folder select **Module Specification**.
6. From the list of module specifications that appear to the right, double-click on **func4**.

The Module Specification displayed contains information obtained by CasePlayer2 about *func4()*.



A list of global variables referenced and assigned within *func4()* is included in the module specification (as shown below). Now we can use this information for our test conditions in CoverageMaster winAMS. If the reference information would not be displayed, go back to **Analysis Settings** and check the necessary options.

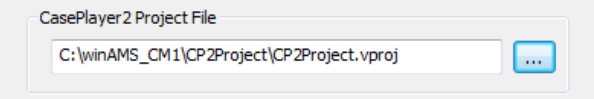
Referenced Global Variables		
Variable	Definition File	Line
gb_b	main.c	180
gb_c	main.c	180
gb_d	main.c	180
gb_out	main.c	180
gb_a	main.c	180

Assigned Global Variables		
Variable	Definition File	Line
gb_out	main.c	180

Import CasePlayer2 analysis results

Now let's return to SSTManager (CoverageMaster winAMS) and make C1 test data for *func4()*. The analysis data should have already been imported from CasePlayer2. First let's verify the analysis data file is configured correctly.

1. From the SSTManager menu select: **File** -> **Project Settings**.
2. From the Test Project Settings dialog that is displayed, check that the **CasePlayer2 Project File** is set to: ***C:¥winAMS_CM1¥CP2Project¥CP2Project.vproj***
   (if not, click the … button and select the file located in the above path).
3. Click the **OK** button.



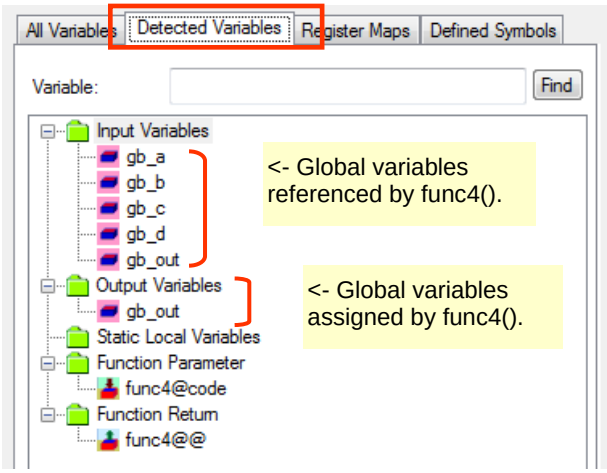
After confirming that the appropriate file has been selected we are now ready to use the CasePlayer2 analysis data.

Select Variables from the Analysis Results

Now we are ready to create the test CSV file for *func4()*. The method for creation is as follows (same as the previous exercises).

1. From SSTManager click the **Create CSV** button, select **Unit Test CSV**.
2. Filename: **func4_data**, Function: **func4**, Test Description: (optional)
3. From the variable list select the **Detected Variables** tab.

The **Detected Variables** tab is only available when CasePlayer2 analysis data is present. Using the data collected during the CasePlayer2 analysis this tab contains a list of variables referenced or assigned by *func4()*.



In order to test function *func4()*, it is necessary to input values into the global variables referenced by *func4()*. In order to do so, we will add the variables shown in the **Input Variables** folder to the INPUT list. Likewise add the **Output Variables** and **Function Return** to the OUTPUT list.

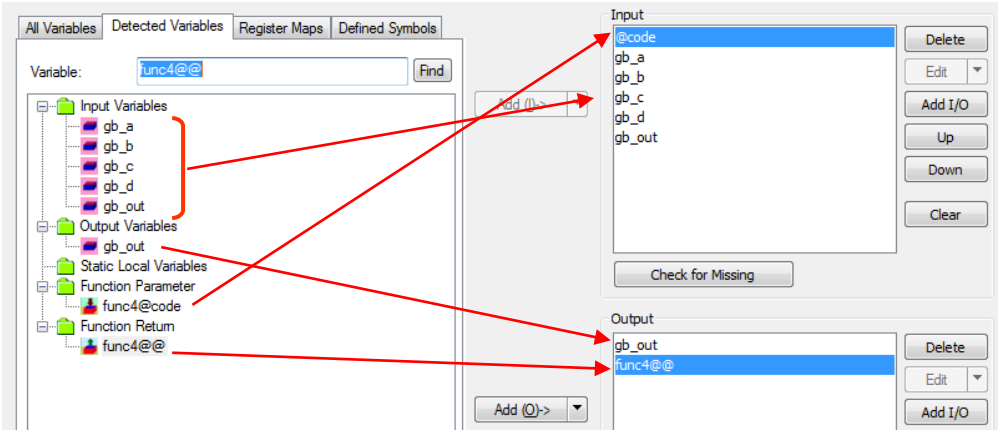
In the previous exercises since the test functions did not contain very many variables we simply selected the appropriate ones ourselves. For functions with a large number of variables detection of variables using CasePlayer2 analysis information can be very useful.

Now let's add the variables to the INPUT & OUTPUT lists. Starting with the INPUT list:

4. From the **Function Parameter** folder add ***func4@code*** to the **INPUT** list.
5. From the **Input Variables** folder add the 5 variables: ***gb_a, gb_b, gb_c, gb_d, gb_out*** to the **INPUT** list.

Now for the OUTPUT list.

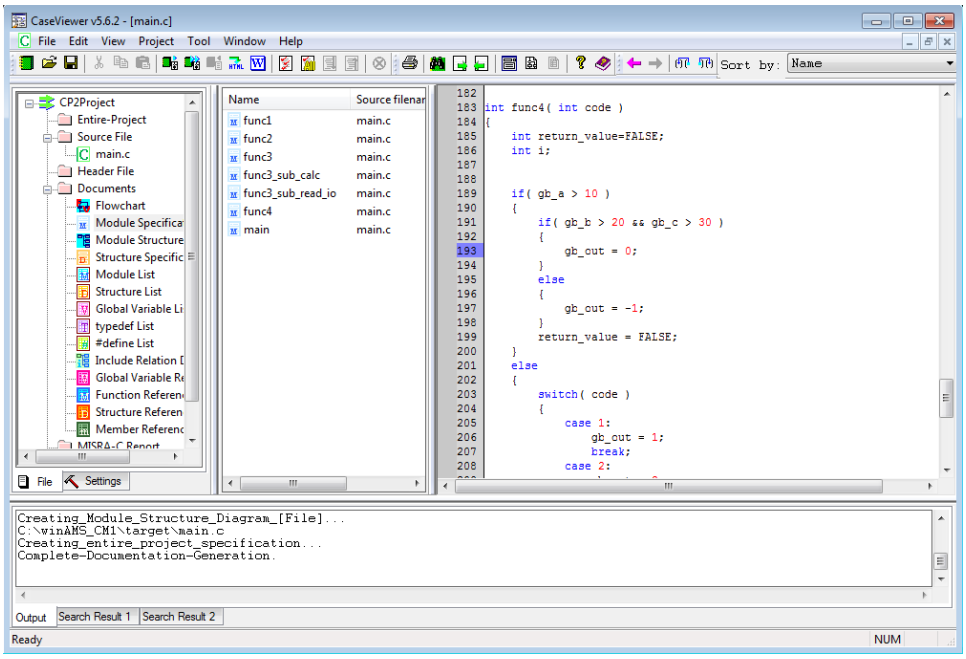
6. From the **Output Variables** folder add ***gb_out*** to the **OUTPUT** list.
7. From the **Function Return** folder add ***func4@@*** to the **OUTPUT** list.



We've now finished setting up the INPUT & OUTPUT lists for *func4()*. Notice that we added ***gb_out*** to both the INPUT and the OUTPUT list. Using a CasePlayer2 search feature let's check how this variable is being used in the source code.

8. Right-click on ***gb_out*** in the INPUT list and select **Open Source Code [Variable]** from the right-click menu.

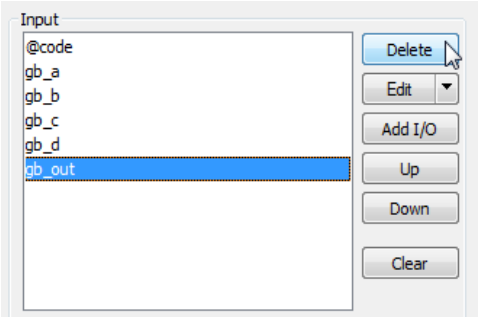
The source file will be opened in the CasePlayer2 editor and go to the first line where *gb_out* is used.



If we continue to search downward for ***gb_out*** we'll find it inside the if statement: `if( gb_d == gb_out )`. Because *gb_out* is being compared to another global variable (*gb_d*), CasePlayer2's analysis includes it as both an input and an output.

But because the value for *gb_out* is determined within *func4()* it is not necessary for us to add it to the INPUT list. Therefore, *gb_out* may be removed from the INPUT list.

9. Select ***gb_out*** from the INPUT list and click the **Delete** button.
   (*gb_out* will be removed from the INPUT list only).



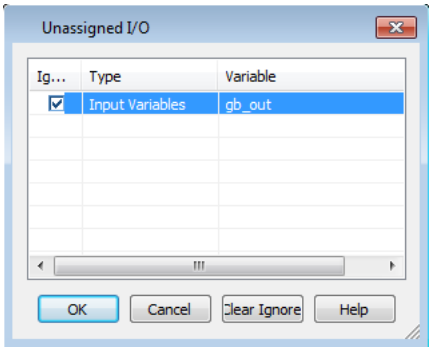
We've now finished setting up the input / output variables. Next we're going to make test data for the C1 coverage test.

10. Click the **Enter Data** button lined up on the bottom row.

Upon clicking the Enter Data button the **Unassigned I/O** dialog will be displayed. This dialog appears as a confirmation when input/output variables detected have not been added to the INPUT or OUTPUT list. In this case it is warning us that *gb_out* has not been added (the item we

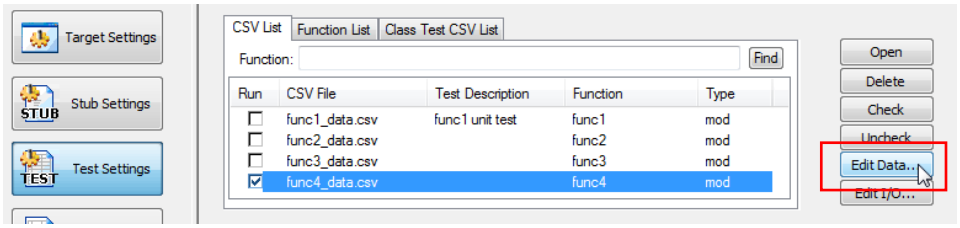
recently removed). In order to prevent the dialog from appearing you may check **Ignore** box next to the variable.

11. Check **Ignore** for ***gb_out*** in the Unassigned I/O dialog.
12. Click **OK**.



Test Data Editor (ATDEditor)

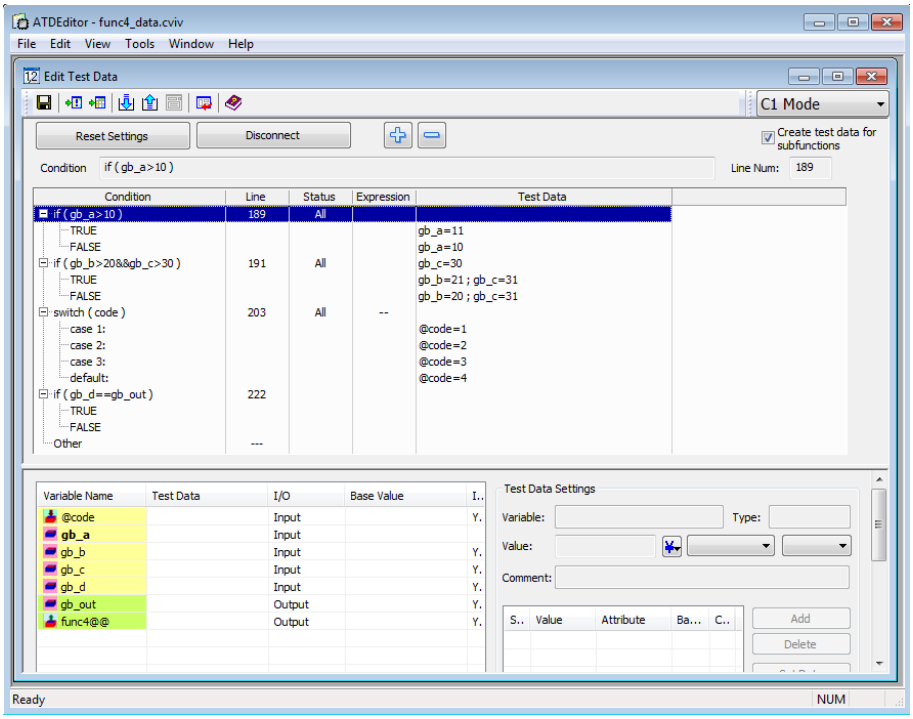
Next we'll use the test data editor (ATDEditor) to creating test data combinations from the CasePlayer2 analysis data.



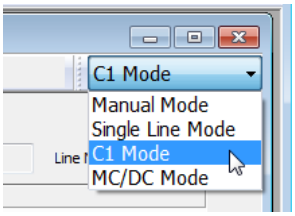
1. Select **func4_data.csv** in the CSV List in the Test Setting view.
2. Start the test data editor (ATDEditor) by pressing the **Edit Data** button in the right portion of the view.

The test data editor will start.

3. From ATDEditor's menu click **Window** -> **Edit Test Data**.
   (If the **Edit Test Data** window is not in the list, make sure that Edit Test Data Window is checked in the **View** menu).



There are 4 modes in the ATDEditor that can be selected from the pull-down menu in the upper-right portion of the screen.



The 4 modes are as follows:

- **C1 Mode**: for creating C1 (branch) coverage data.
  - Using CasePlayer2 analysis data automatically create test data to fulfill C1 coverage.
  - Automatically create test data for each conditional statement.
  - For maximum efficiency the minimum number of required test data sets will be generated.

- **MC/DC Mode**: for creating MC/DC (modified condition/decision coverage) data.
  - Same as C1 mode but for MC/DC testing.

- **Manual Mode**: for manually creating data coverage data.
  - Does not require CasePlayer2.
  - Manually set the data for each variable.
  - Data combinations' "base values" are assigned based on a "decision table". The decision table will be explained later in this document.

- **Single Line Mode**: for creating 1 line worth of CSV test data.
  - Assigning 1 value to each variable.
  - Can simply create 1 set of test data without having to worry about combinations.

The mode may be changed in order to create test data using different modes if desired, but for this exercise we will only use the C1 mode.

13. From the pull-down menu in the upper-right change the mode to **C1 Mode**.

Condition	Line	Status	Expression	Test Data
if (gb_a>10)	189	All		
TRUE				gb_a=11
FALSE				gb_a=10
if (gb_b>20&&gb_c>30)	191	All		gb_c=30
TRUE				gb_b=21 ; gb_c=31
FALSE				gb_b=20 ; gb_c=31
switch (code)	203	All	--	
case 1:				@code=1
case 2:				@code=2
case 3:				@code=3
default:				@code=4
if (gb_d==gb_out)	222			
TRUE				
FALSE				
Other	---			

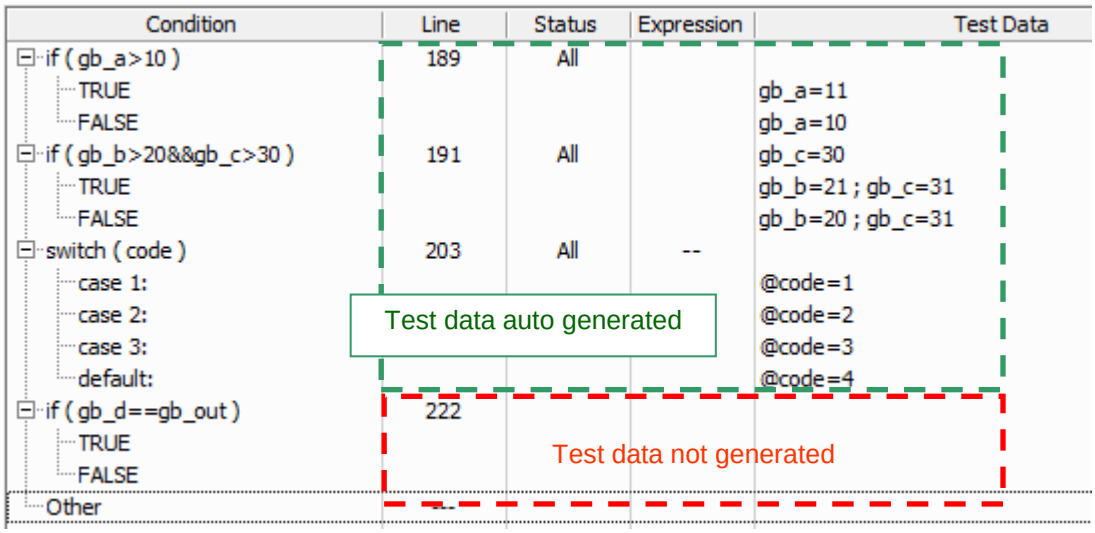
In this view the branches of the condition statements for *func4()* are listed. C1 coverage requires that each branch of every condition statement be executed. For example, for an if statement both T/F must be executed, and for a switch statement each case must be executed.

This view includes each condition statement shown independently regardless of its nested status (hierarchy) for ease of visibility and usage. For example, the 2nd condition statement, **`if(gb_b>20 && gb_c>30)`**, is actually nested in the True case of the first condition statement, **`if(gb_a >10)`**.

Check the Auto Generated Test Data

The first thing we want to confirm is the auto generated test data. When test data values were detected for all branches '**All**' will appear in the condition's **Status** column. In this exercise 'All' should appear in the status column for the top three conditions.

The last condition, **`if(gb_d == gb_out)`**, is blank because since the value of gb_out is determined during runtime, test values were not auto generated.



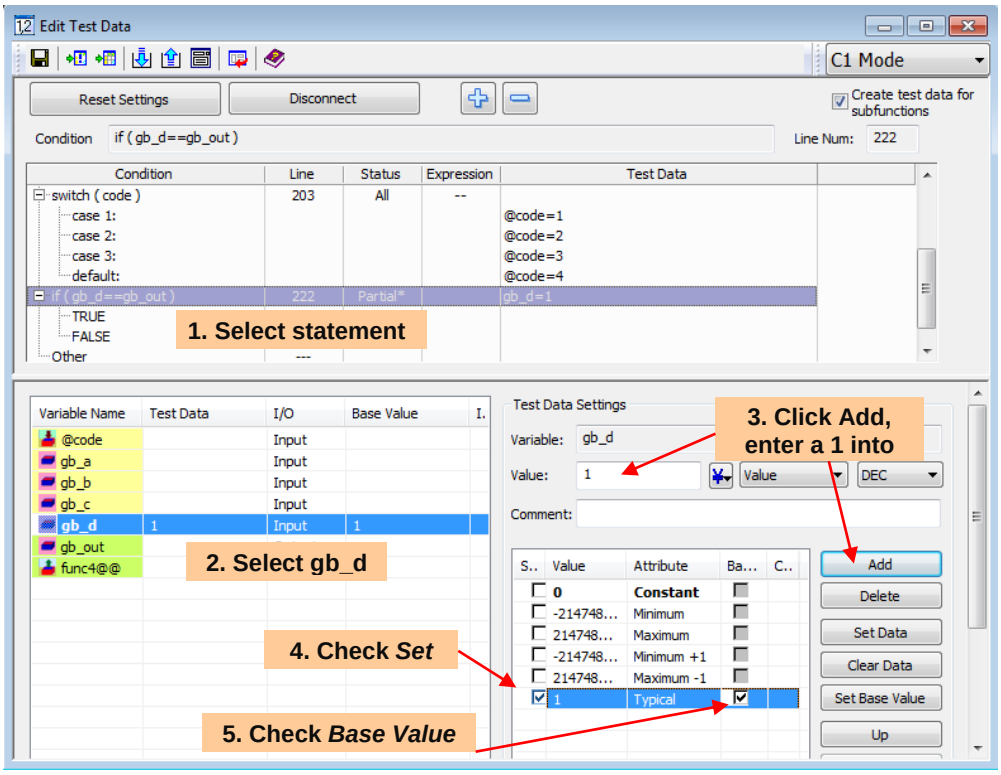
Manual Settings

Now we're going to specify the data for the last conditional statement, `if( gb_d == gb_out )`. This conditional statement is comparing the global variables *gb_d* and *gb_out*. *gb_out* is assigned a value from within the function so we need to input values for *gb_d* that will fulfill C1 coverage requirements.

The values for *gb_out* in the switch statement are as follows: case 1: *gb_out=1*, case 2: *gb_out=2*, etc. Therefore, if we simply set *gb_d* to 1, case 1 of the switch statement will result as true, and case 2 will result as false thereby fulfilling the C1 coverage requirements.

Now let's set ***gb_d*** to be constantly 1.

1. Select the ***if(gb_d == gb_out)*** conditional statement.
2. Select ***gb_d*** from the variable list (lower-left).
3. In the **Test Data Settings** region (lower-right) click the **Add** button, then input a **1** into the **Value** box.
4. Check the **Set** box for the recently add 1 value.
5. Check the **Base Value** box for the recently add 1 value.



We've now completed the settings for the conditional statements.

(Ref) Decision Tables and Base Values

Step 7 above (where we checked the base value box) is a part of the method for creating data combinations using a decision table. When creating combinations consisting of multiple elements, a decision table is used as a method for creating only the combinations that are necessary, rather than every possible combination.

For example, suppose a function contains the variables *a, b*, *c* and we assign the following test

values: *a = 9, 10, 11; b = 19, 20, 21; c= 30, 31*. Using these values we will input data combinations into the CSV file for unit testing. But if we enter every combination possible there will be 3 x 3 x 2 or 18 test data patterns (a relatively small number compared to many programs).

If the variables *a, b, c* are independent of one another, it is not necessary that we input data for every combination possible. It is however necessary that we enter each possible value for the variables at least once. But since the variables are independent of one another the values of b and c are irrelevant when testing variable a (with values 9, 10, 11).

To start with we'll define at least one value as the base value. Using the same example variables we've selected the values in brackets '[]' as the base values.

```
a:  9  [10]  11      b: 19  [20]  21      c: [30]  31
```

Using the decision table combinations will be created as follows:

- For variable a's data (9, 10, 11) the values for b and c will be the base values.
- For variable b's data (19, 20, 21) the values for a and c will be the base values.
- For variable c's data (30, 31) the values for a and b will be the base values.

The resulting test data combinations will be as follows:

```
a :  9  10  11   10  10  10   10  10
b : 20  20  20   19  20  21   20  20
c : 30  30  30   30  30  30   30  31
```
Red is the Variable's test values
Black is the Variable's Base Values

In exercise 4 since we only specified one value (1) for the variable *gb_d*, this value will be used as the base value.

Create Test Data Combinations

Now we're going to create test data (a CSV file) using the above settings we made.

When making the settings above for the conditional statements we did not pay attention to the nested (hierarchy) status of the statements, but rather each as an individual condition statement. But some of the statements are nested, for example the second if statement, `if( gb_b>20 && gb_c>30 )`, requires that the proceeding if statement, `if( gb_a >10 )`, to be true for it to be tested. Thus we need to create a test combination that satisfies these requirements.

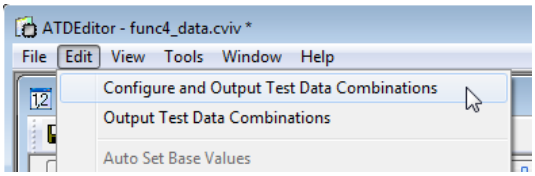
CoverageMaster winAMS uses the nest structure information provided by CasePlayer2 to automatically create test combinations to fulfill these requirements. For the above four conditional statements the following number of test data combinations must be created:

2 (paths) x 2 (paths) x 4 (paths) x 1 (path)

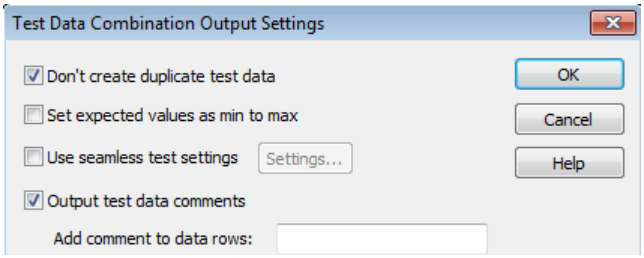
Making a total of 16 paths to make test data combinations for. Using this information CoverageMaster will automatically create the minimal number of test data sets required to satisfy the C1 coverage.

Now let's output the test data combinations.

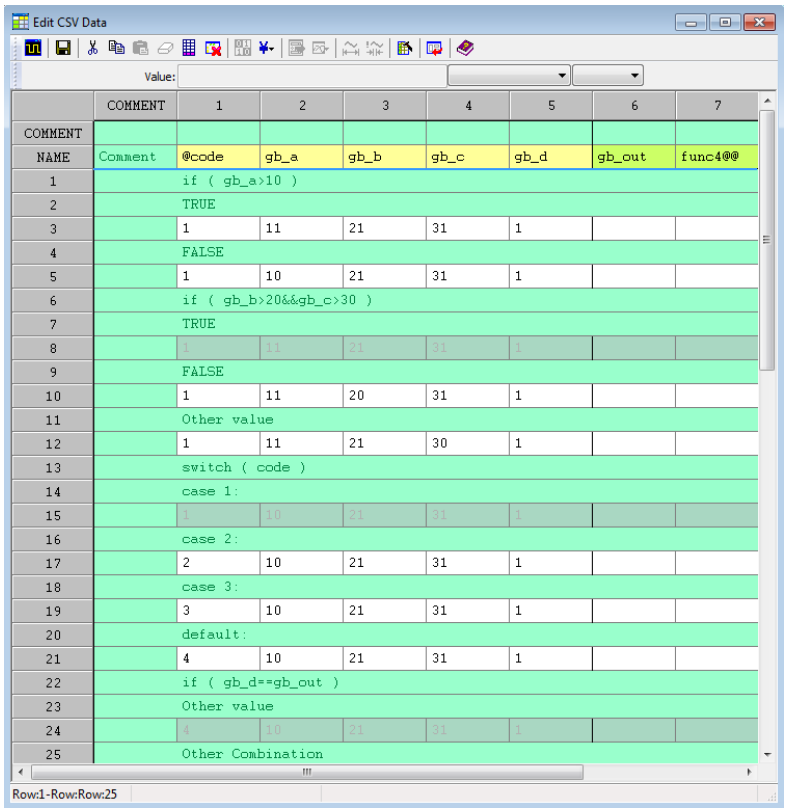
1. From the ATDEditor menu select **Edit** -> **Configure and Output Test Data Combinations**.



2. Check **Don't create duplicate test data**.
3. Check **Output test data comments**.
4. Click the **OK** button.



The test data combinations will be generated and output to the CSV file as shown in the **Edit CSV Data** window.



	COMMENT	1	2	3	4	5	6	7
COMMENT								
NAME	Comment	@code	gb_a	gb_b	gb_c	gb_d	gb_out	func4@@
1		if (gb_a>10)						
2		TRUE						
3		1	11	21	31	1		
4		FALSE						
5		1	10	21	31	1		
6		if (gb_b>20&&gb_c>30)						
7		TRUE						
8		1	11	21	31	1		
9		FALSE						
10		1	11	20	31	1		
11		Other value						
12		1	11	21	30	1		
13		switch (code)						
14		case 1:						
15		1	10	21	31	1		
16		case 2:						
17		2	10	21	31	1		
18		case 3:						
19		3	10	21	31	1		
20		default:						
21		4	10	21	31	1		
22		if (gb_d==gb_out)						
23		Other value						
24		4	10	21	31	1		
25		Other Combination						

The test data combinations generated include seven test patterns. Also because we chose to output comments, we can see which condition statement and result the test data corresponds to.

The green lines indicate comment lines. If you look at line #8 you'll notice it is the same data as line #3. These duplicate test data combinations are commented out because of the settings we made in the Test Data Combination Output Settings dialog.

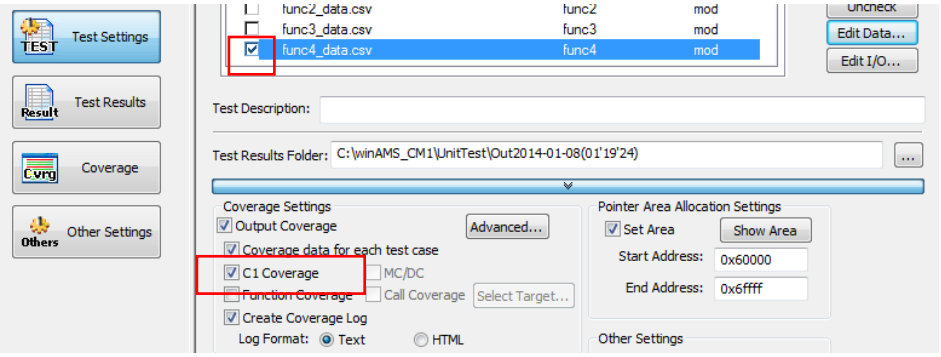
C1 Coverage Test using the Generated Test Data

Now let's save the CSV file and run the test.

1. From the **Edit CSV Data** window click **File** -> **Save**.
2. Click **File** -> **Exit** to close the ATDEditor.
3. From the Save Prompt click **OK** to save the editor setting file (*func4_data.cviv*).

Now we'll return to the Test Settings screen of SSTManager and prepare for testing.

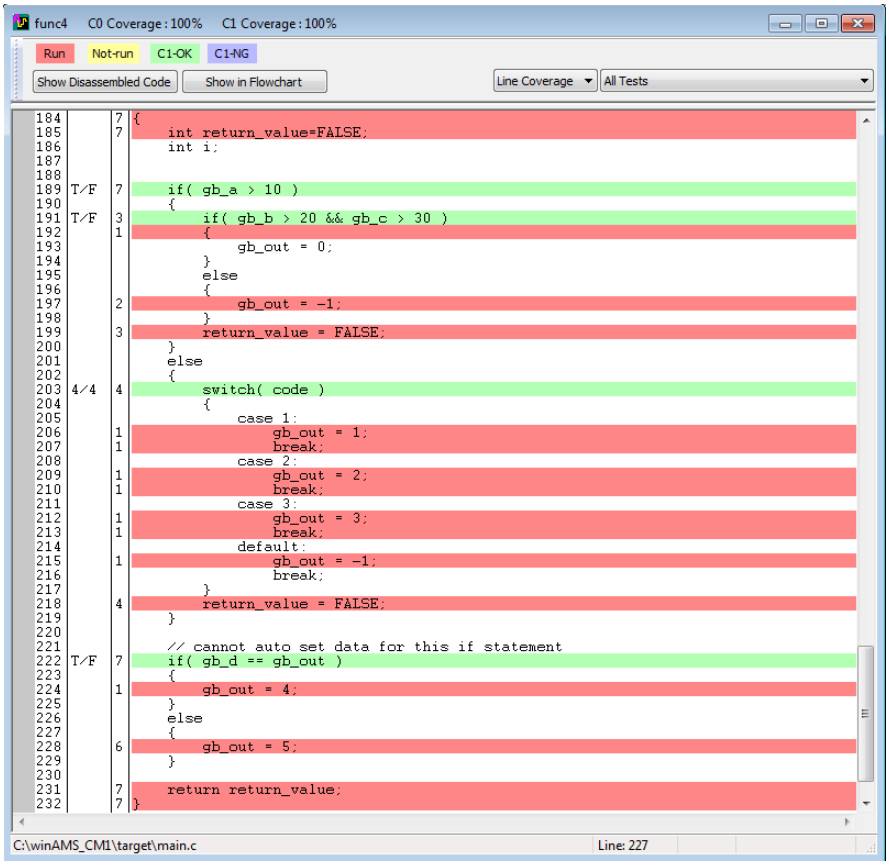
4. From the **Test Settings** screen of SSTManager check ***func4_data.csv*** (other CSV files should be unchecked).
5. Under the **Coverage Settings** section of the Test Settings screen check **Output C1 Coverage**.
6. Click the **Start Simulator** button to start the simulation.



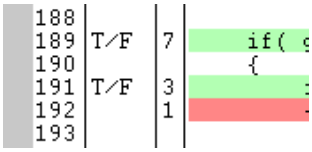
Check the Coverage for func4()

From the Coverage view screen, check that both the C0 and C1 coverage rates are at 100%. We were able to obtain 100% coverage in this exercise simply using the seven automatically created test data sets.

Function	C0	C1
func4	100%	100%



The Coverage view should appear as the image above. Green lines indicate full C1 coverage results for the highlighted branch. In the column next to the line number column (ex. Line #189) "**T/F**" is displayed to indicate that both true and false were executed for the if statement. If only true was executed it would be displayed as " **T/** ", " **/F**" for only false, and " **/** " for neither.



For **switch** statements the number of cases executed will be displayed as # *executed / total* #. In this example, all 4 cases were executed so "**4/4**" is displayed.

Conclusion

This concludes the CoverageMaster winAMS tutorial. In this tutorial we learned:

- Once we've prepared a test data CSV file, the simulation itself is automated.
- We can automatically allocate memory for functions which include pointer variables.
- When testing functions that include function calls, we can replace and control the subfunctions using stub functions without having to modify the source code.
- We can automatically create test data combinations to fulfill C1 coverage through the help of the CasePlayer2 analysis tool.

Thank you for choosing CoverageMaster winAMS. We hope that the fundamentals you have learned while performing these exercises will be useful to your company's product development.

(Reference) Technical Support Information for Users

Technical support information for users can be found on the GAIO website. This page provides information for users of CoverageMaster winAMS/General.

How to Access

Access the following URL using the link below.
(To access the link from Acrobat Reader, hold down the CTRL key and click the link.)

http://www.gaio.com/support/user/techpaper.html

Frequently asked questions about how to use this software are included in the FAQ. The page also includes setup instructions, as well as information on simulator macros (scripts) for advanced users. This technical support information is available for all users.

[Application] Creating Test Cases by Test Data Analysis

Introduction

In addition to the Auto Create Test Data feature used in Tutorial 4, CoverageMaster (v3.7 or newer) also includes a support feature for creating test cases based on function design specifications for confirming the features of a function.

In this application manual, you will learn about the feature for efficiently creating specifications-based test data and performing reviews using the Test Data Analysis Editor.

What Is Unit Test Data Analysis?

GAIO has provided unit testing services for many years. These services consist of test design based on function design documents or source code supplied by the client, taking black box and white box into account, and evaluation and delivery of the results obtained from test execution.

When performing this work, an I/O Analysis Table is created. This is a table containing I/O variables for the input/output of the function and test data values to be assigned to these variables. This is used to check that no test items are omitted and that the assigned values are suitable before creating the test cases.

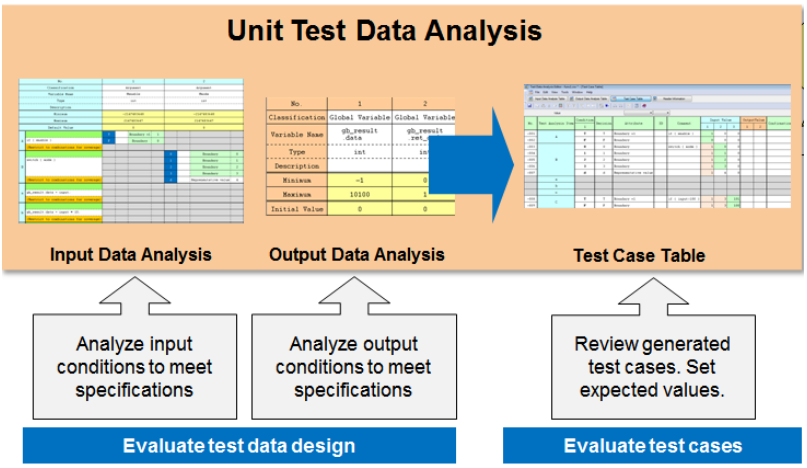
Usually when designing test cases for unit testing, the test elements required for unit testing, such as boundary values, maximum/minimum values and test values for each equivalence class, are extracted from the specifications according to predetermined design procedures. These are then combined when generating the test cases in such a manner that all of these items can be tested with no omissions.

In Unit Test Data Analysis, our unit testing workflow has been turned into a method for CoverageMaster users, based on experience that GAIO has accumulated through its unit testing services. CoverageMaster is equipped with Test Data Analysis Editor, a support feature for efficiently designing test cases using this method and confirming the suitability of test data designed while referring to function specifications.

Test Data Analysis Table for Reviewing and Cross-Checking Test Designs

When designing unit tests with CoverageMaster's unit test data analysis method, this method confirms the suitability of test input data using the Input Data Analysis Table before creating test cases to assign to the function. The test data assigned to each input variable of each test item based on the requirement specifications is compiled in a table. After confirming that the input data is sufficient and performing an evaluation, this data is combined with no omissions and test cases to be assigned to the function are generated.

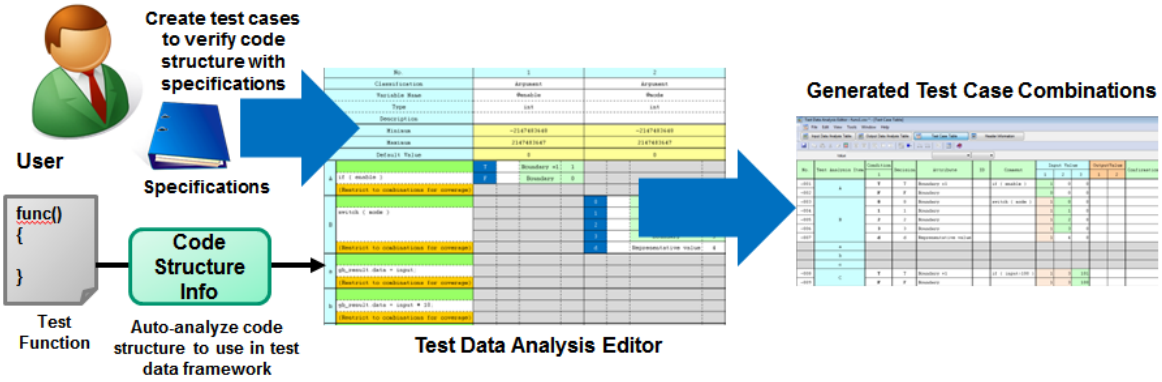
Test design for output values is performed using the Output Data Analysis Table. The output values that need to be acquired when executing the function (expected values) are confirmed and listed in this table. An evaluation is performed to confirm that the expected values for each of the designed input test cases are covered by the output values extracted from this Output Data Analysis Table. This confirms whether the test is sufficient.



Efficient Design and Evaluation of Test Cases while Referring to the Code Structure and Requirement Specifications

Essentially, specifications-based tests confirm whether a function operates according to defined function specifications. It is therefore necessary to read and understand the document, extract the necessary test items from the specifications, and create test cases when designing these tests. However, this work requires an extremely large number of labor hours, including the procedures up to the design review.

To make this work more efficient, the Test Data Analysis Editor described in this chapter provides support with a feature that divides the information into test analysis items based on the code structure information analyzed in CasePlayer2 and uses these for efficient test design.



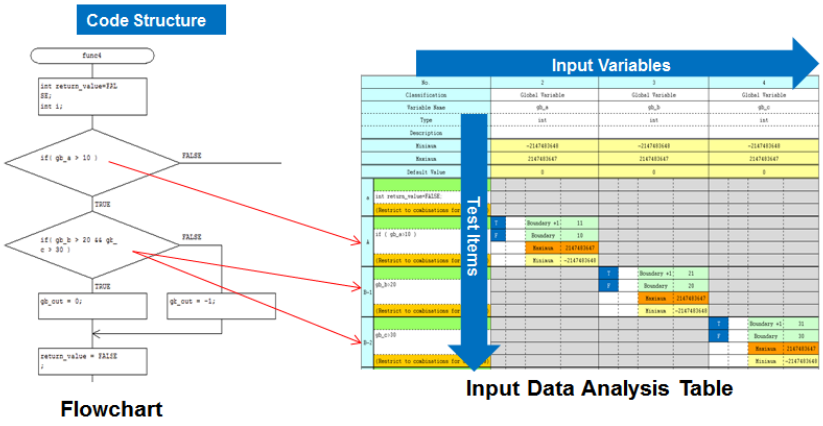
More Efficient Test Data Design Using Test Analysis Items and Automatic Extraction of Test Data

The unit test data analysis feature first parses the code structure using CasePlayer2's code analysis feature and then automatically creates a "Test Analysis Item" field corresponding to the code structure in the Input Data Analysis Table. The user can evaluate the suitability of the source code structure by confirming whether the generated "Test Analysis Item" field meets the requirement specification items. It is also possible to confirm the source code from the test analysis items and confirm the structure in the CasePlayer2 linked to the table.

After confirming that the code structure corresponds to the specification items, test data is configured to confirm detailed requirement specifications for each test analysis item. For branch structure components, CasePlayer2's code analysis feature is used to analyze variables and boundary conditions related to the branching conditions and automatically extract values such as boundary values, maximum and minimum values and singular values as factors for the unit test. The user evaluates whether it is possible to confirm the test analysis items with the automatically

extracted test data, and amends the data if it is excessive, insufficient or incorrect. This enables efficient design of test input data.

This in turn enables efficient execution of black box testing, in which the requirement specifications are confirmed, and white-box testing, in which the code structure is confirmed.



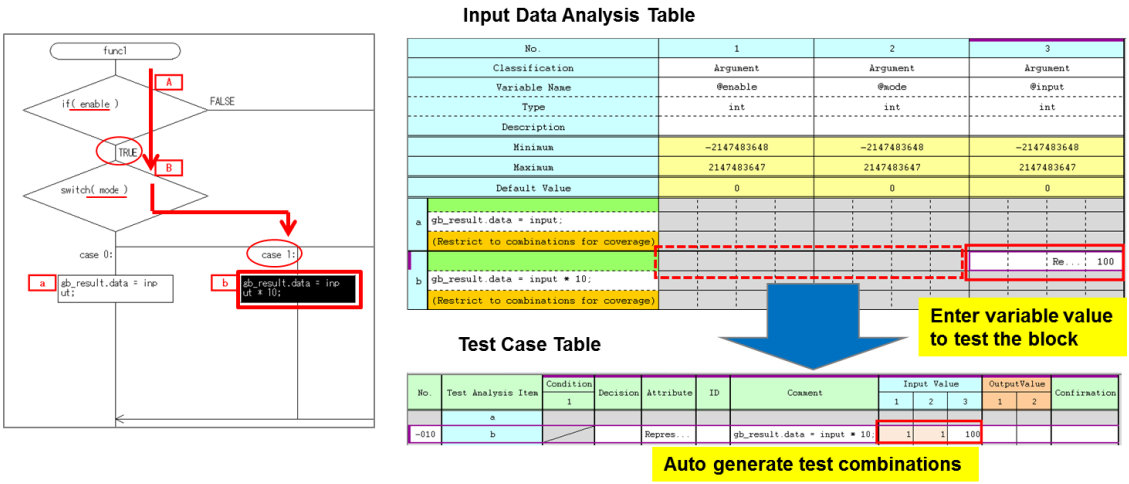
Automatic Generation of Test Cases from the Input Data Analysis Table

After confirming the suitability and coverage of the input test conditions in the Input Data Analysis Table, the test data is combined to generate test cases for the function.

One point of concern for test designers is when creating test combinations for code branches. For example, when testing the case marked as [b] in the flow chart below, the following conditions are required in order to arrive to this branch: variable “enable” = “TRUE” for branch [A] and variable “mode” = “1" for branch [B]. If designing this test data manually, the nest structure of the conditions must be considered in order to arrive at the desired branch.

Test Data Analysis Editor includes a feature that manages conditions based on the nest structure of the source code and the test data entered in the "Test Analysis Item" to automatically generate test data combinations. In the example below, the user simply enters "input = 100" as an input condition for testing block [b] and Test Data Analysis Editor automatically combines the branching conditions for this block (enable = TRUE, mode = 1) and generates them in the test cases.

This means that the user no longer needs to design complicated combinations of conditions, enabling the user to quickly design test cases for the condition needed.

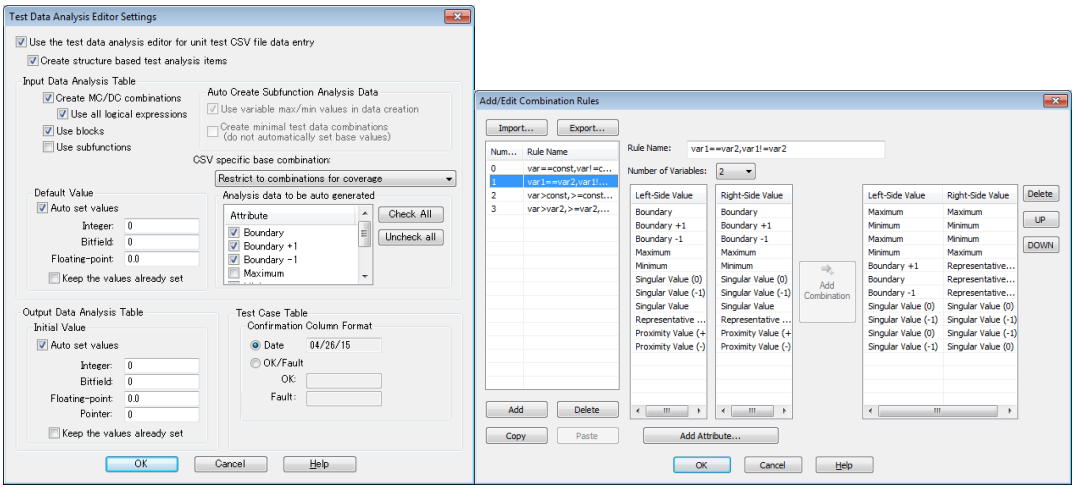


Specify Design Rules to Standardize the Precision of Test Design

When designing unit tests from requirement specifications, there is a high degree of freedom in terms of design methods, meaning that the degree of precision differs depending on the tester. During development projects in which unit tests are conducted by multiple testers, it is particularly important to set a particular test data design method as a procedure and communicate this in a manual to enable standardized design that is not dependent on a tester's skill level.

Test Data Analysis Editor includes a feature for setting design rules for test data to be assigned to functions. General methods for deriving test cases for unit cases include data such as boundary values, maximum/minimum values, representative values from equivalence partitioning and singular points such as 0 and -1. Setting rules about which data to assign by default makes it possible to prevent the precision of the data extraction from varying according to the tester, prevent omissions from occurring in test items and standardize test data design among testers.

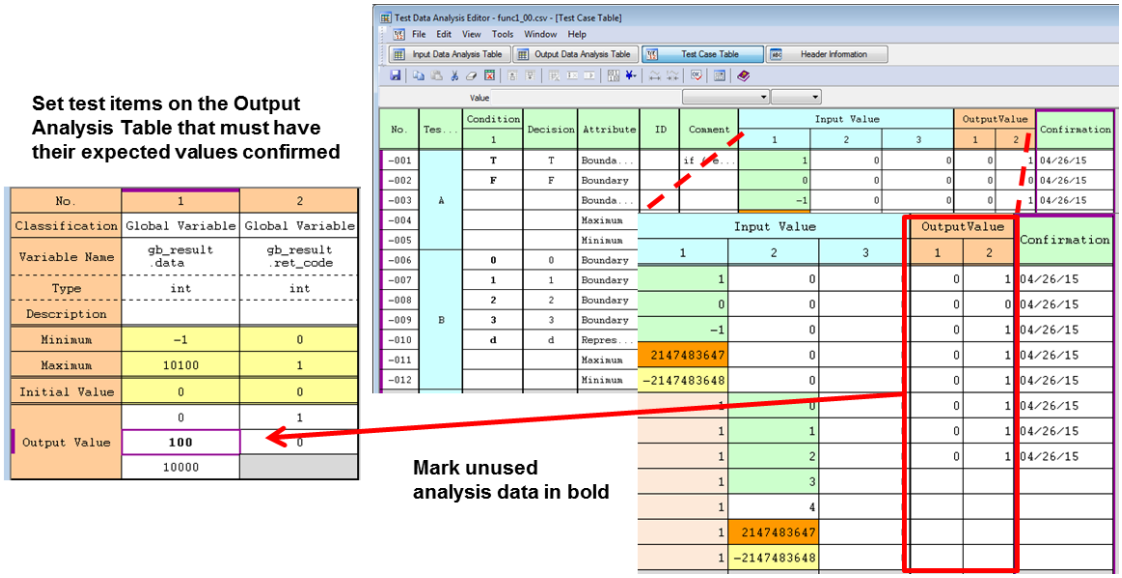
Test Data Analysis Editor also includes a feature for custom configuration of rules concerning which data attributes to combine when generating test case combinations from the test data set in the Input Data Analysis Table.



Design Confirmation in Relation to Expected Values Using the Output Data Analysis Table

It is also possible to increase the accuracy of test design by using an Output Data Analysis Table to confirm the design in relation to expected values. After setting the input conditions for confirmation of the requirement specifications, data that needs to be confirmed as output is extracted in advance and indicated in the Output Data Analysis Table.

For example, when dealing with a function for a specification with three outputs - 5, 4 and 3 - this specification needs to be covered by confirming whether input conditions that will output each of these three values are set. In this case, the user sets 5, 4 and 3 as outputs in the Output Data Analysis Table and confirms whether these three values are covered by using expected values (output values) in test cases with generated combinations. If any data is not set in the expected values for the test cases in the Output Data Analysis Table, this can be detected and displayed in bold text. This makes it possible to detect omissions in test conditions based on expected values.



Tutorial 5: Designing a func5() Test in the Test Data Analysis Editor

Now try designing a test for one function using the Test Data Analysis Editor. In this tutorial, you will perform the following tasks using the func5() function. *Tutorial 4 in the previous chapter needs to be completed before performing this tutorial.

- Configuring and confirming an Input Data Analysis Table based on a source structure
- Configuring and adding test analysis items with specific conditions
- Generating test case combinations
- Configuring an Output Data Analysis Table and setting expected values

Confirming Requirement Specifications and Source Code for the Test Function

First, confirm the specifications and code of the function to be used in the tutorial. To begin with, the specification conditions of the func5() function are defined as follows.

<Design specifications for func5 >

Determine values for the two inputs - input1 and input2 - using a mode switching value (mode)
and output to gb_result.data. Note that the entire function is turned on or off using the enable input flag.

Input: (all arguments)
enable (flag) : when feature is OFF output gb_result is (0, FALSE)
mode (variable) : mode switch 0: Output gb_result is (input1, TRUE)
1: Output gb_result is (input2, TRUE)
2: Output gb_result is (input1+input2, TRUE)
default: Output gb_result is (255, TRUE)
input1(variable) : input1 range (0-150)
input2(variable) : input2 range (0-150)

Output: (global structure)
gb_result.data : output data

```
    gb_result.ret_code : error code (TRUE when feature is on; otherwise FALSE)
---------------------------------------------------------------------------
```

The following code is created based on these specifications.

```
---------------------------------------------------------------------------
void func5( int enable, int mode, unsigned char input1, unsigned char input2 )
{
    if( enable )
    {
            switch( mode )
            {
            case 0:
                    gb_result.data = input1;
                    break;
            case 1:
                    gb_result.data = input2;
                    break;
            case 2:
                    gb_result.data = input1 + input2;
                    break;
            default:
                    gb_result.data = 255;
                    break;
            }
            gb_result.ret_code = TRUE;
    }
    else
    {
            gb_result.data = 0;
            gb_result.ret_code = FALSE;
    }
}
---------------------------------------------------------------------------
```

The above func5() is included in a main.c using the tutorial sample that was used in the previous chapters, and is included in the object code compiled in Tutorial 1. An analysis using CasePlayer2 is also performed in Tutorial 4. Use this as it is.

Confirming Test Analysis Items Based on Requirement Specifications

The following five requirement specification items have been set and numbered in this tutorial to confirm the requirement specifications of the func5 function. Test data for testing these requirement specifications will then be designed using the Test Data Analysis Editor.

<Sample of requirement specifications>

- Specification 001: The entire feature shall be switched by the enable flag.
- Specification 002: When enable is OFF (FALSE), output gb_result is (0, FALSE).
- Specification 003: When enable is ON (TRUE), output gb_resul.ret_code is TRUE.
- Specification 004: The mode shall be switched by mode.
- Specification 005: Output gb_result.data is 255 shall be fixed when mode is a value other than 0, 1 or 2.
- Specification 006: input1 shall be selected for output gb_result.data for mode 0.
- Specification 007: input2 shall be selected for output gb_result.data for mode 1.
- Specification 008: The additional values of input 1 and input2 shall be output to output gb_result.data in mode 2.

Confirming Test Indexes (Design Rules)

The following test indexes (design rules) are set in this tutorial. While more detailed and specific test indexes are set in actual operations, only a sample of simple rules will be handled in this tutorial.

<Sample of test indexes>

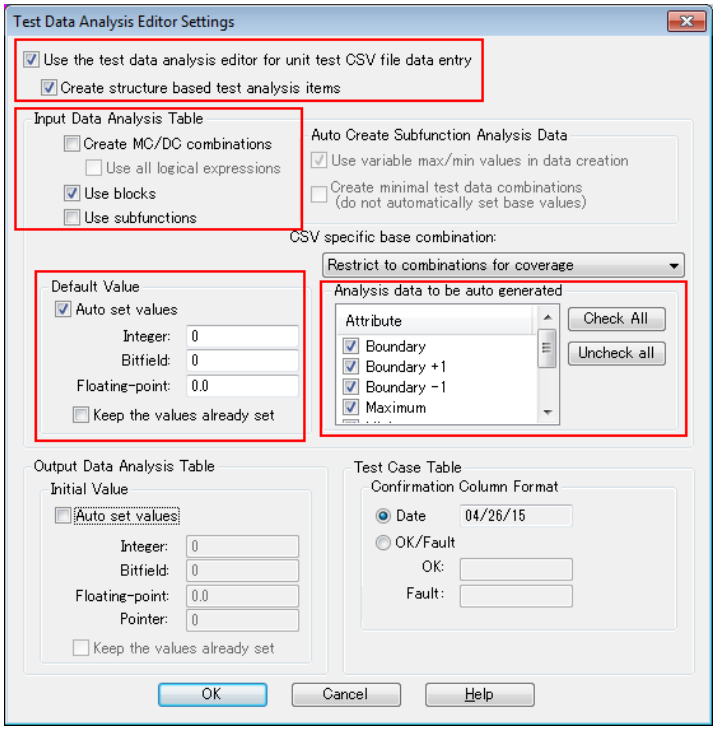
Index 1: Assign only TRUE/FALSE for flags
Index 2: Add required data and maximum value/minimum value to ordinary variables
Index 3: Assign maximum value/minimum value/median value to variables with a range index
Index 4: Assign maximum type value/minimum type value to variables without a range index
Index 5: Confirm overflow caused by the maximum value/minimum value in the operator component

---

Configuring the Test Data Analysis Editor Settings

Configure the settings for using the Test Data Analysis Editor.

1. In SSTManager, select the "Tools" menu - "Test Data Analysis Editor Settings..."
2. Set the check boxes as shown in the figure below.
3. Set the "CSV specific base combination" setting to "Restrict to combinations for coverage"
4. Select the following attributes for “Analysis data to be auto generated”: Boundary, Boundary +1, Boundary -1, Maximum, Minimum
5. Click "OK" to save and close the settings window.



[About the Options]

□ Use the test data analysis editor for unit test CSV file data entry

This is the main option for using the Test Data Analysis Editor. Clicking the "Test Input" button in the Test Settings View or the Create Model for Module Test window launches the Test Data Analysis Editor when this checkbox is checked. ATDEditor, which was used in Tutorial 4, is launched when this checkbox is unchecked.

<CAUTION>

The selection of Test Data Analysis Editor or ATDEditor is recorded as a flag in the CSV file. When a created CSV file is selected and the "Input Data" is clicked, Test Data Analysis Editor is launched if "1" is indicated in the M column (13th column) of the first row. ATDEditor is launched if this cell is blank or "0" is indicated.

□ Create structure based test analysis items

This option parses the code structure and automatically generates a Test Analysis Item field. If this is turned off, test analysis items are not generated and all analysis item settings need to be configured manually. Set this to "OFF" if setting test analysis items manually using all of the specifications information only, without referring to the code structure. The recommended setting is "ON" when using the Test Data Analysis Editor.

□ Create MC/DC combinations

This option is a switch for generating test case combinations that fulfill MC/DC for branches with composite conditions. This is used when applying MC/DC measurements. Set this to "OFF" in this tutorial.

□ Use blocks

Set this to "ON" to extract processing blocks other than branches（rectangles in a flowchart） as test analysis items when parsing code structure and automatically generating a Test Analysis Item field. The combinations of conditions for branching to the test analysis items (processing blocks) generated in this option are automatically analyzed. The recommended setting is "ON" when using the Test Data Analysis Editor.

However, apply "OFF" when you want to focus only on branching and create a specialist test for executing all paths, like in Tutorial 4. In this case, only the branches are extracted as test analysis items.

□ Use subfunctions

Use this to create test data that includes input/output conditions of subfunctions when performing combined unit testing including actual subfunctions. However, the subfunction application range extends only to subfunctions one level below the test target function (base point). In ordinary unit testing, subfunctions are stubbed and the actual subfunctions are therefore not used in the test. For this reason, this setting should be turned off for ordinary testing.

□ Default value

This is an option for setting a default value for test data to be used in combinations of variables for which data has not been set in the Test Data Analysis Table when combining data and generating a Test Case Table after creating the Test Data Analysis Table.

□ Analysis data to be auto generated

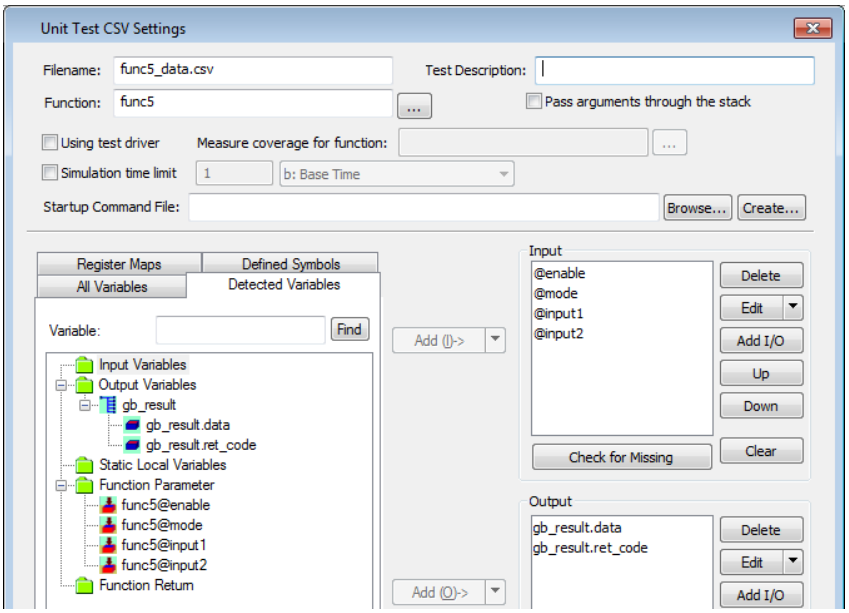
This is an option for specifying the default type of data to be created for variables extracted as variables related to condition branches. Select a type according to the test indexes (design rules). In this tutorial, data for all attributes except singular values is set as the default.

Other options will not be used in this tutorial. See the Help Manual for details.

Creating a func5() Test CSV

In this step, you will create a func5() test CSV file. The procedure for creating a CSV file is the same as Tutorial 4.

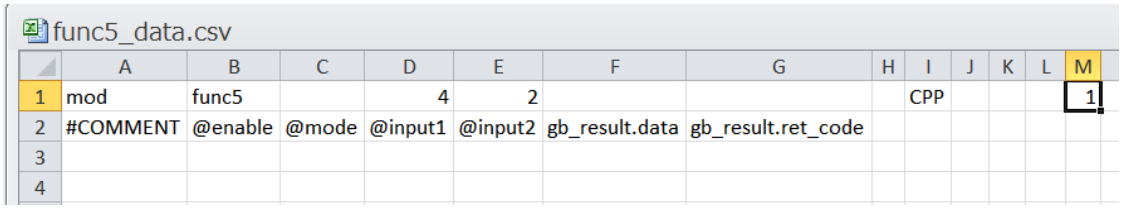
1. Click the "Create CSV" button in the SSTManager and select "Unit Test CSV".
2. Set the filename to “func5_data”, select the function “func5”.
3. Set the func5() input/output conditions from the "Detected Variables".
   INPUT: @enable, @mode, @input1, @input2
   OUTPUT: gb_result.data, gb_result.ret_code
4. Click the "OK" button to save the CSV file.



Confirm the model (format) of the created CSV file.

1. Double click "func5_data.csv" from the Test CSV list in the "Test Settings" view.
2. Confirm the CSV file opened in MS Excel.

A test CSV file is created with the I/O variable names input. The figure "1" in the M column (13th column) of the first row indicates that the mode for using the Test Data Analysis Editor is set. If this cell is blank or "0" is written, the ATDEditor used in Tutorial 4 will be used instead of the Test Data Analysis Editor.



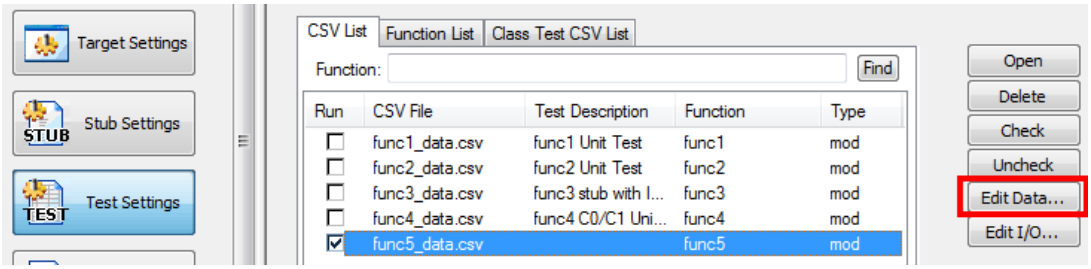
	A	B	C	D	E	F	G	H	I	J	K	L	M
1	mod	func5		4	2				CPP				1
2	#COMMENT	@enable	@mode	@input1	@input2	gb_result.data	gb_result.ret_code						
3													
4													

3. Close the CSV file after confirming this.

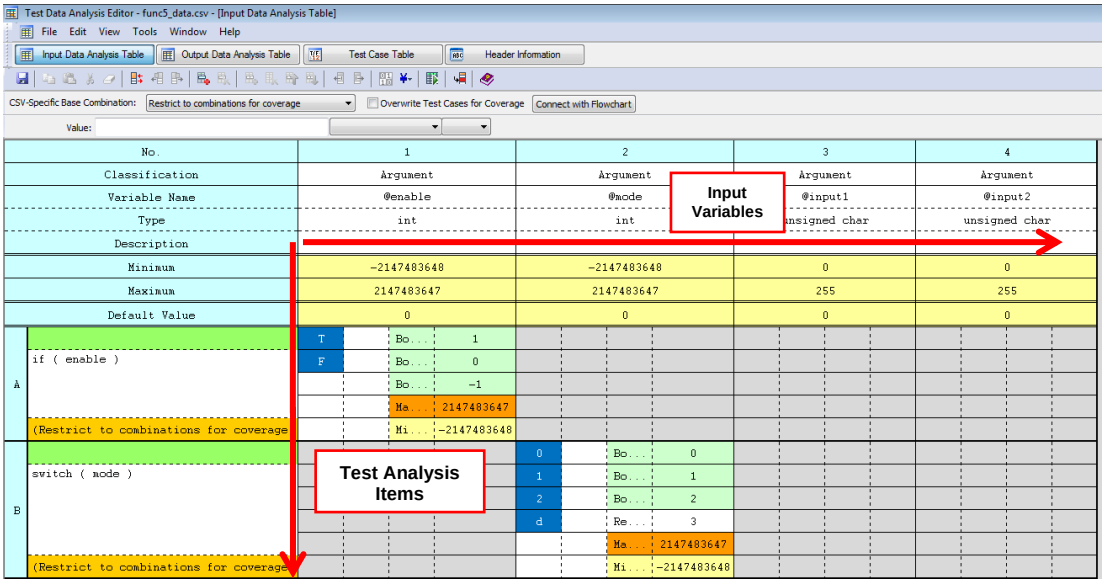
Starting the Test Data Analysis Editor

Now start the Test Data Analysis Editor.

1. Select "func5_data.csv" from the Test CSV list in the "Test Settings" view and click the "Edit Data" button.



The settings configured in "Test Data Analysis Editor Settings" are reflected and an input data analysis table is automatically created. The input data analysis table consists of a matrix with the rows indicating input variables and the columns indicating test analysis items obtained by parsing the code.



First, here is an explanation of the input variables indicated by the rows. Variables selected when creating the CSV model are automatically input in these variable fields. Information such as arguments, classifications (e.g. Global Variable), types, maximum values/minimum values and default values set in the Test Data Analysis Editor Settings is also automatically input. The maximum value/minimum value in the variable specifications can be set as the maximum value/minimum value.

The first number, "No.", is used as a variable ID in features such as the Test Case Table, which will be used later. (This number is displayed in place of the variable name in the Test Case Table, etc.)

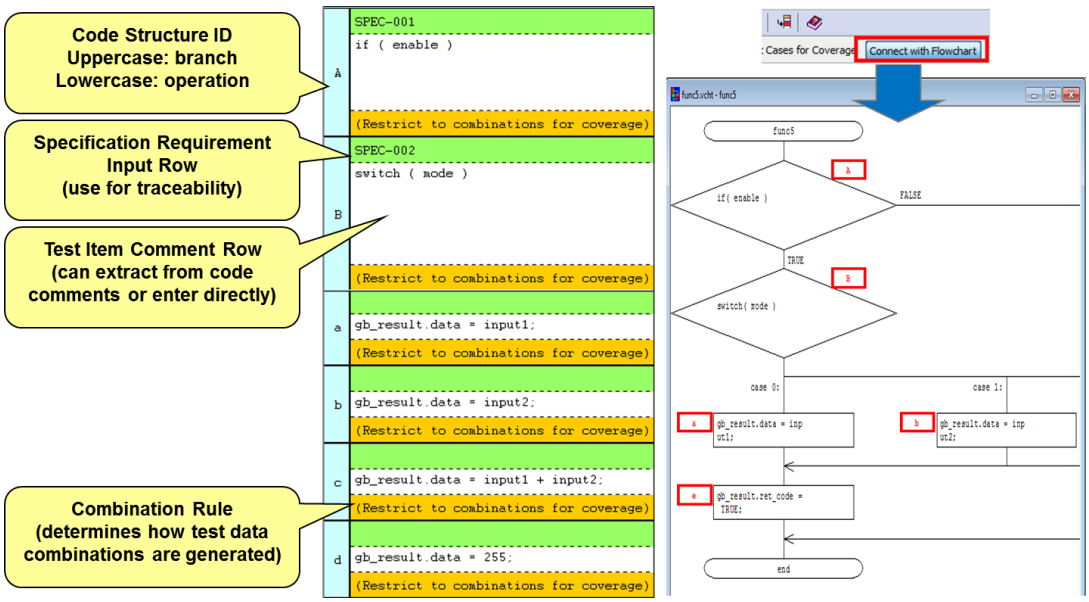
No.	1
Classification	Argument
Variable Name	@enable
Type	int
Description	
Minimum	-2147483648
Maximum	2147483647
Default Value	0

Next, here is an explanation of the test analysis items indicated by the columns. The source structure is parsed and test analysis items are automatically generated in this field based on the results. The character on the left is the ID of the parsed test analysis item. A capital letter indicates a branch block and a lower case letter indicates a processing block. When the "Coverage Flowchart" button at the top of the editor is turned on, the ID is displayed in the flowchart in CasePlayer2 and its handling is indicated in a linked display.

The green row is used for the handling of the control number of the requirement specifications assigned to the document. This can be used to manage the handling (traceability) of the requirement specifications and test analysis items required for purposes such as safety verifications of features.

The white field is a comment field for test analysis items. An extract of the source code is displayed here by default, but this can be edited to replace or add to this.

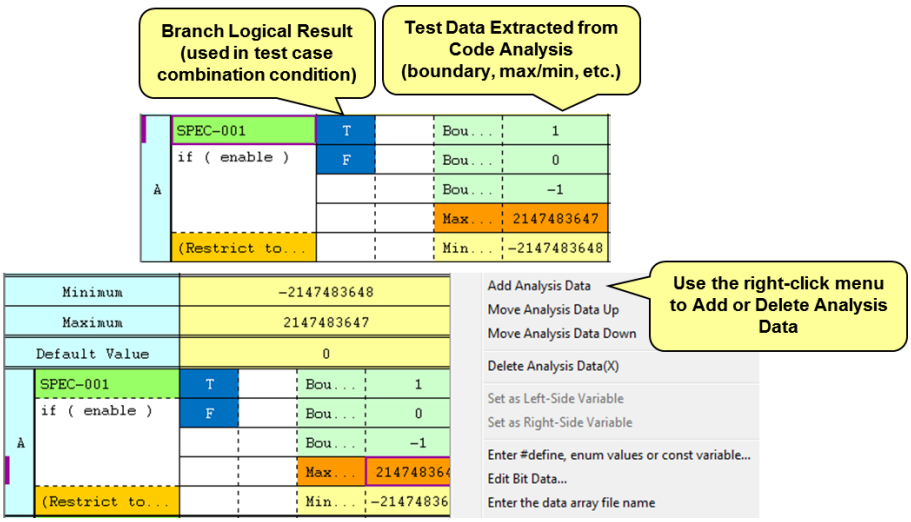
The orange row contains a combination rule selected for application when automatically generating test cases from the Input Data Analysis Table. "Restrict to combinations for coverage", a rule that generates the minimum number of combinations needed to run all branches, is applied by default, but this can be changed to generate combinations with a specific rule.



Finally, here is an explanation of the test data that is automatically set. Like in Exercise 4 when using the ATDEditor, test data, based on CasePlayer2 boundary value analysis, is automatically set in the Input Data Analysis Table.

The logic for executing the condition branches is automatically set in the blue box. When combinations of branch conditions need to be set according to the nest structure of the branches, the logic set here is used to create the combinations. However, when testing the "B" branch and the branches below it in the figure shown above, for example, "TRUE" needs to be set for the "A"

branch above it. The data for which "enable variable" is "TRUE" ("1 (boundary value +1)") is therefore used in the combinations generated for test cases of the "B" branch and below.



This test data can be edited (added to, deleted, etc.) by right clicking the applicable area to display a menu. The data set by default can be deleted if it is not necessary for the predetermined test indexes (rules) by right clicking and selecting "Delete Analysis Data".

If static analysis has not been executed for an executable statement (if statement, switch statement, etc.) of a condition branch, no test data is set for the test analysis item corresponding to the capital-letter ID that indicates that branch. In this case, the user needs to create new data (right click and select "Insert Analysis Data") and determine the logic for this condition branch by themselves.

Confirming Correspondence to the Requirement Specifications and Adding to the Input Data Analysis Table

First, confirm whether the items correspond the requirement specifications based on the automatically extracted code structure in the Input Data Analysis Table. If a code structure corresponding to an item in the requirement specifications cannot be found in the Input Data Analysis Table, this requirement specification may not be covered and needs to be verified. Conversely, if the table contains code that is not in the requirement specifications, this may mean that unnecessary code remains. This also needs to be verified.

This step is a key point for efficiently confirming that no requirement specifications are missing and that there is no unnecessary code - in other words, performing both black box testing and white-box testing - by checking the code structures against the specification requirements.

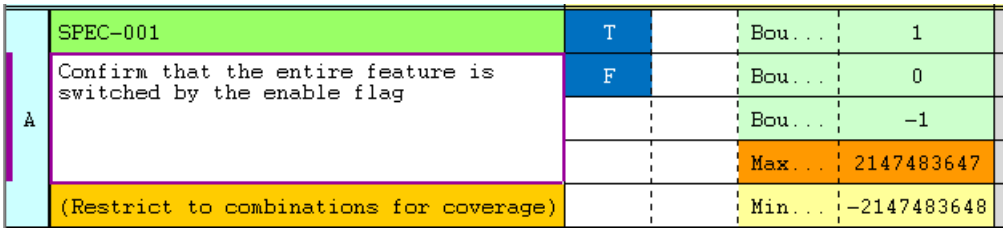
First, confirm which of the extracted test analysis items in the Input Data Analysis Table confirm to each of the numbers from 001 to 006 that were set as the requirement specifications, and add the numbers and comments in the Input Data Analysis Table.

First, it can be confirmed that,

> Specification 001: ("The entire feature shall be switched by the enable flag" (gb_result.ret_code=FALSE)")

corresponds to test analysis item "A". Add this information to the Input Data Analysis Table.

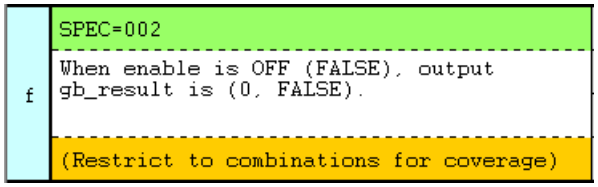
1. Click the green box in the field for test analysis item "A" and enter "SPEC-001".
2. Replace the "if (enable)" comment at the center with "Confirm that the entire feature is switched by the enable flag".



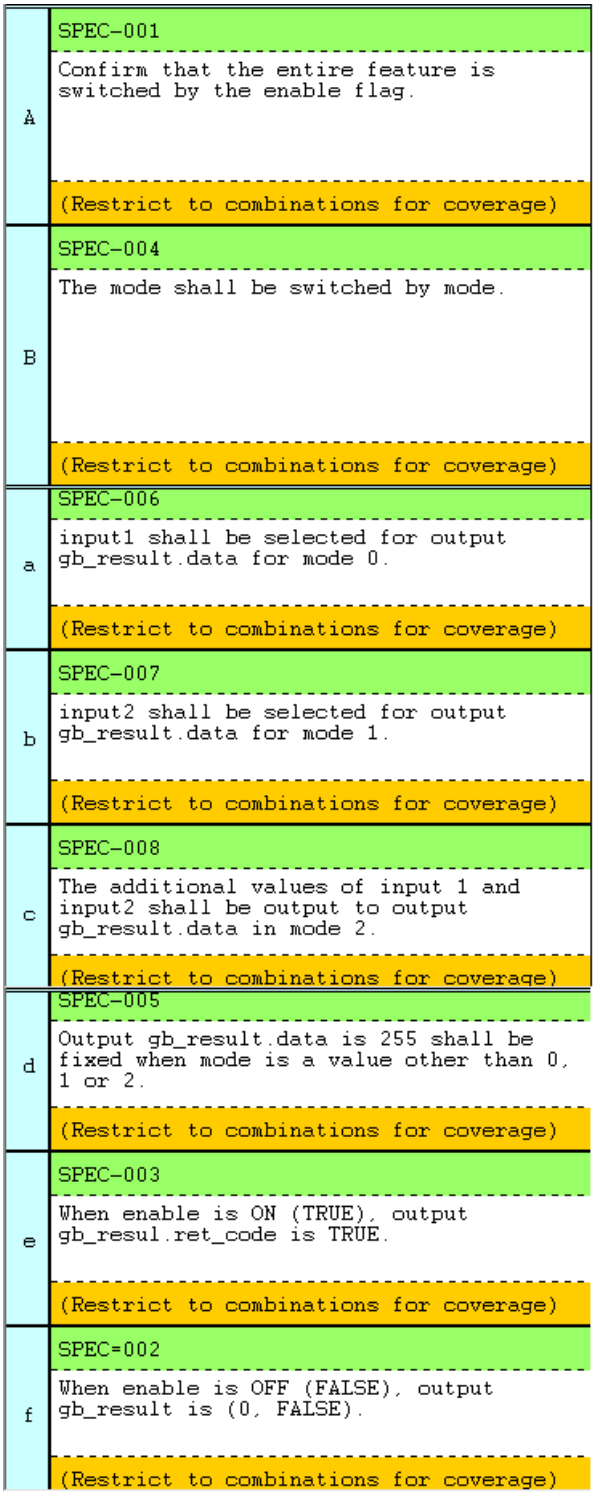
Next,

Specification 002: When enable is OFF (FALSE), output gb_result is (0, FALSE)

is included in processing block "f". Add this information to the Input Data Analysis Table.



Confirm how the remaining requirement specification numbers correspond with the Input Data Analysis Table and add them in the same way.



You have now confirmed that the code structures correspond with all of the items in the requirement specifications. This also confirms that there are no unnecessary code structures.

Editing the Test Data Based on the Test Indexes

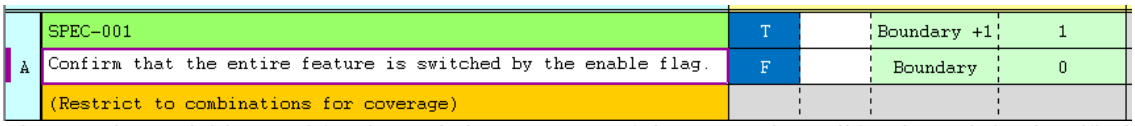
In the next step, you will set the test data necessary for confirming the requirement specifications according to previously defined test indexes (design rules).

First, edit the automatically extracted test data. To begin with, the input variable "enable" used in the if statement test of the first condition branch is classified as a "flag" in the document. Set only TRUE/FALSE data for this according to the following previously confirmed test index:

"Index 1. Assign only TRUE/FALSE for flags"

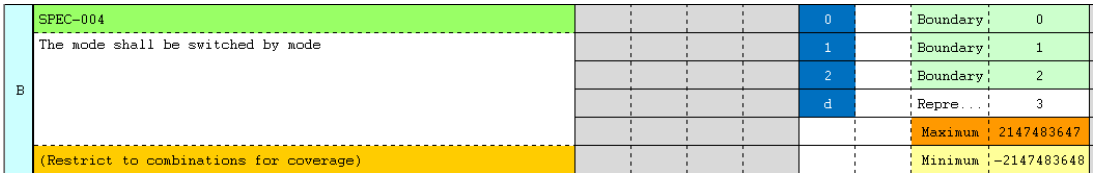
The automatically set boundary value (-1), maximum value and minimum value therefore need to be deleted from the table.

1. Right click the blue-green area containing the “Boundary -1” value for the "enable" variable and select "Delete Analysis Data".
2. Right click the orange area containing the “Maximum” value of the "enable" variable and select "Delete Analysis Data".
3. Right click the yellow area containing the “Minimum” value of the "enable" variable and select "Delete Analysis Data".



The "mode" variable used for the switch statement of the second condition branch is classified as an ordinary variable in the document. Use the extracted data as it is according to the following previously confirmed test indexes:

"Index 2. Add required data and maximum value/minimum value to ordinary variables"
"Index 4. Assign maximum type value/minimum type value to variables without a range index".

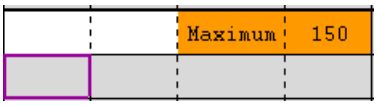


By doing this, you have set the test data for all mode switches to be executed by the "mode" variable.

Next, set the test cases to confirm the actions in each mode. First, it is necessary to verify that the value set for input1 in the test for Requirement Specification 006 is correctly output to the gb_result.data output. To do this, set 150 as the maximum value, 0 as the minimum value and 75 as the median value (representative value) according to the following test index:

"Assign maximum value/minimum value/median value to variables with a range index".

1. Right click the "input1" area of the field for test analysis item "a" and select "Insert Analysis Data".
2. Set the second cell from the right as "Maximum value" and enter "150" in the cell on the right.



Set the minimum value as 0 and the median value (representative value) as 75 in the same way. (Note: The representative value indicates standard data to be assigned as a test case.)

		Maximum	150
		Minimum	0
		Repr...	75

The settings of the test for requirement specification 007 are the same as those for requirement specification 006. The same settings as above should therefore be configured for "input2" in the field for requirement specification 007. The same settings can be configured easily by using the copy and paste feature of the editor.

1. Select the cells set for requirement specification 006 as shown in the figure below.
2. Right click and select "Copy". (Ctrl+C on the keyboard can also be used.)
3. Select the cells to which to paste the information ("input2" in the field for requirement specification 007).
4. Right click and select "Paste". (Ctrl+V on the keyboard can also be used.)
   (Note: be certain to place the cursor in the appropriate cell in order to paste the data)

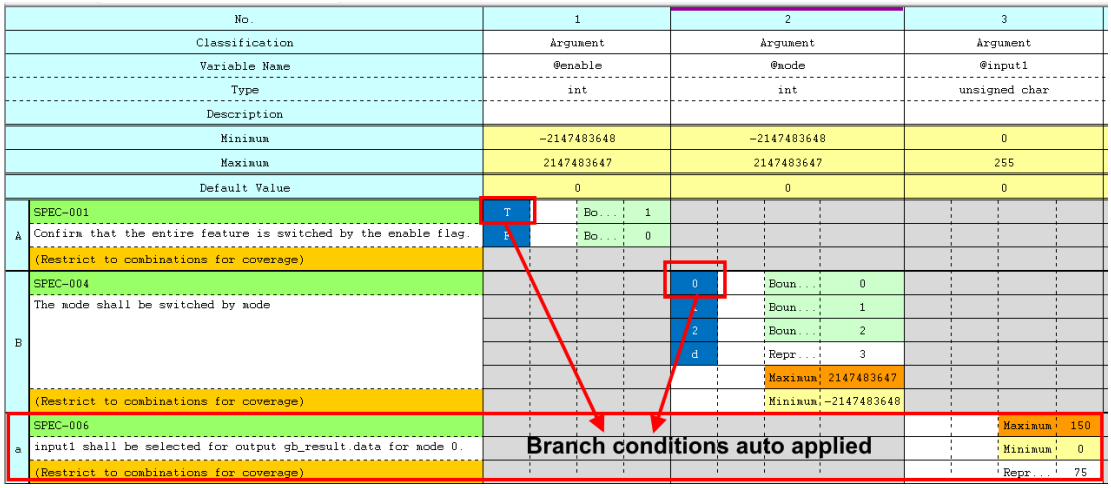
		Maximum	150
		Minimum	0
		Repr...	75

(DOWN)

		Maximum	150				
		Minimum	0				
		Repr...	75				
						Maximum	150
						Minimum	0
						Repr...	75

The test data for requirement specification 006 (test analysis item "a") and requirement specification 007 (test analysis item "b") has now been set. However, branch conditions need to be taken into account in order to branch into these processing blocks. For example, in order to run requirement specification 006 (test analysis item "a"), "enable=TRUE" and "mode=0" need to be assigned as conditions.

To design and carry out this task more efficiently, the Test Data Analysis Editor includes a feature to automate this process by managing and combining these branch conditions for the user. Analysis by CasePlayer2 has determined that requirement specification 006 (test analysis item "a") is included in the nests of the "enable=TRUE" and "mode=0" conditions. Data specified by the "T" logic of the "enable" variable and data specified by the "0" logic of the "mode" variable is therefore automatically applied when generating test case combinations later. This means that the user does not need to consider condition branches during input test data analysis.



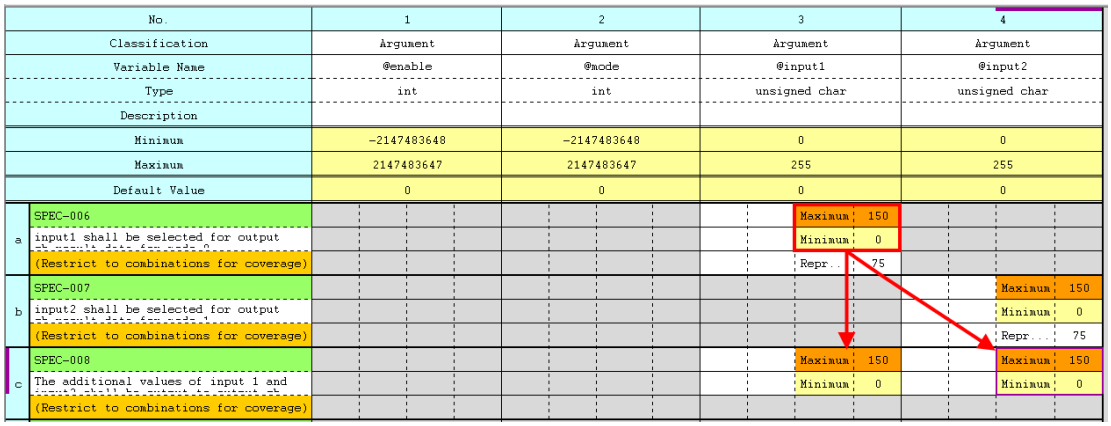
Specifying Combinations Based on Test Indexes

Next, add test data for testing test analysis item "c" (requirement specification 008): "The additional values of input 1 and input2 shall be output to output gb_result.data in mode 2." The following test indexes are applied for operation here:

Index 3: Assign maximum value/minimum value/median value to variables with a range index
Index 5: Confirm overflow caused by the maximum value/minimum value in the operator component As test analysis item "c" contains operations including addition, overflow needs to be tested according to index 5 using a maximum value and minimum value. Here you will set a maximum value and minimum value for the respective ranges of input1 and input2, create all combinations for each input and design them to be output as test cases.

First, set a maximum value and minimum value for input1 and input2 in the test analysis item field of test analysis item "c" (specification 008).

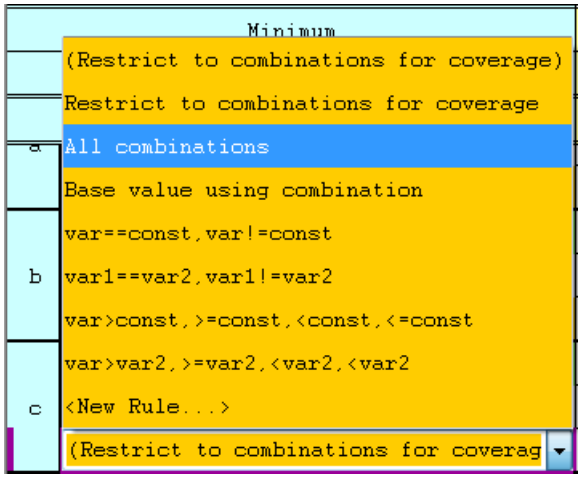
1. Select the cells containing the maximum value and minimum value input for test analysis item "a".
2. Right click and select "Copy". (Ctrl+C on the keyboard can also be used.)
3. Select the "input1" cell for test analysis item "c" (requirement specification 008).
4. Right click and select "Paste". (Ctrl+V on the keyboard can also be used.)
5. Select the "input2" cell for test analysis item "c" (requirement specification 008).
6. Right click and select "Paste". (Ctrl+V on the keyboard can also be used.)



Set the combination rule for test analysis item "c" (specification 008) to “All Combinations” so that all respective combinations of the maximum values and minimum values are generated for

generating test cases.

7. Select "All combinations" from the combination rules (orange pull-down menu) for test analysis item "c" (specification 008). This setting can be changed by using the pulldown menu to the right of the orange combination rule cell.

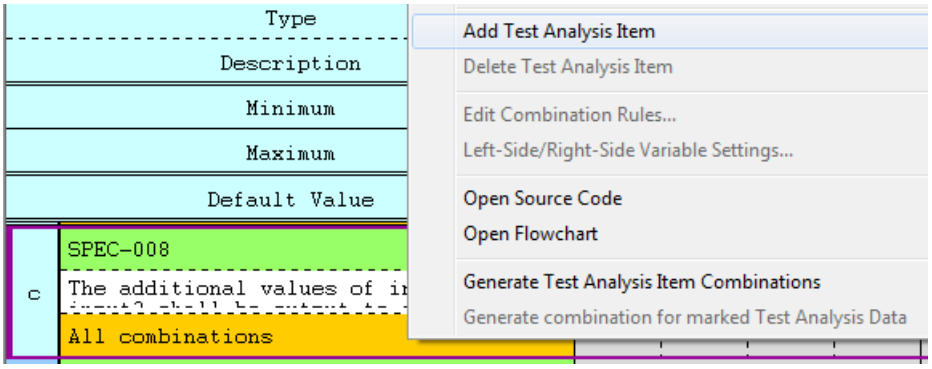


Next, add test data that complies with the following index:

Index 3: Assign maximum value/minimum value/median value to variables with a range index As all number combinations of the maximum values and minimum values have already been created, add only the test data for the median value here and design test data in different combinations from the maximum values and minimum values.

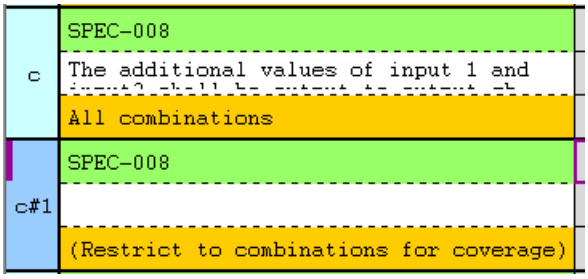
Here you will create a subtest analysis item for test analysis item "c". When a subtest analysis item field is added under a test analysis item and test data is set, other variable conditions for branching to the processing block of this item are automatically applied in the same way as for the parent test analysis item.

1. Select test analysis item "c" (requirement specification 008).
2. Right click and select "Add Test Analysis Item".
3. "c#1" is added as a sub-block of test analysis item "c".

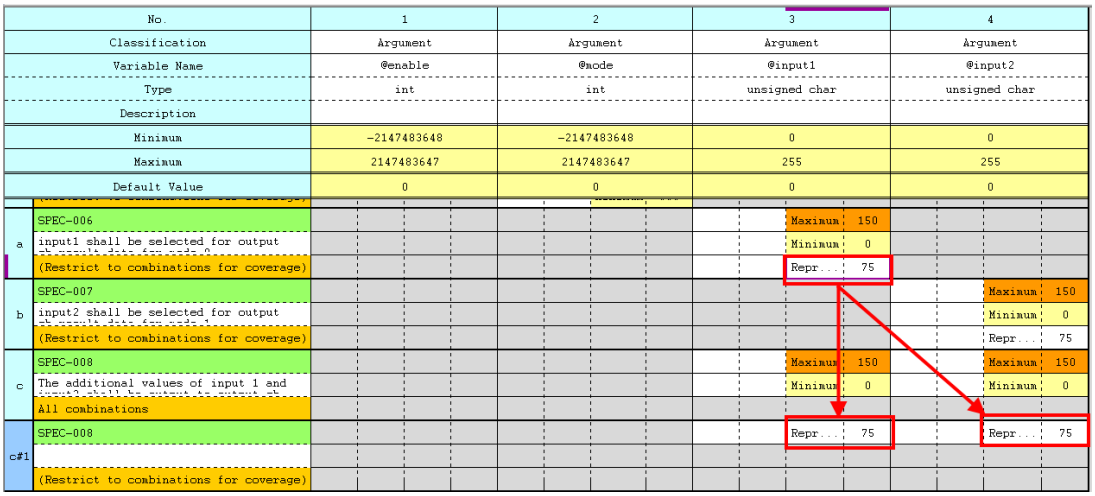


Write the requirement specification number.

4. Click the green box in the field for test analysis item "c#1" and enter "SPEC-008".

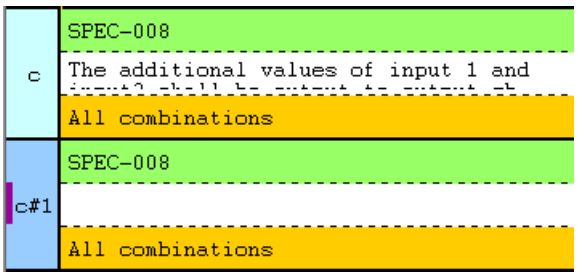


5. Select the cell containing the representative value input for test analysis item "a".
6. Right click and select "Copy". (Ctrl+C on the keyboard can also be used.)
7. Select the "input1" cell for test analysis item "c#1" (requirement specification 008).
8. Right click and select "Paste". (Ctrl+V on the keyboard can also be used.)
9. Select the "input2" cell for test analysis item "c#1" (requirement specification 008).
10. Right click and select "Paste". (Ctrl+V on the keyboard can also be used.)



Set combination rules for test analysis item "c#1" (requirement specification 008) so that combinations for the representative value are generated when generating test cases.

11. Select "All Combinations" from the combination rules (orange pull-down menu) for test analysis item "c#1" (requirement specification 008).



The settings have now been configured so that all combinations of the maximum values and minimum values and combinations of representative values for input1 and input2 are generated as test cases for the operator component.

Adding Test Data to Confirm the Remaining Operational Specifications

This leaves test analysis items "d" and "e" in the Input Data Analysis Table. Test analysis item "d" was created for the purpose of testing the operations of modes other than 0, 1 and 2 and was

assigned the requirement specification number 005. However, this test data is already set as data specifying a branch condition for test analysis item "B" (requirement specification 004). This means that test cases for testing this mode will be generated for test analysis item "B" even if test data is not set for test analysis item "d". In this tutorial, however, we will configure the settings to generate test cases as test analysis item "d" in addition to those for test analysis item "B" in order to clarify how these test cases correspond to the requirement specifications.

1. Right click the "@mode" variable in the field for test analysis item "d" (requirement specification 005) and select "Insert Analysis Data".
2. Set 3 as a representative value.

No.		1	2	
Classification		Argument	Argument	
Variable Name		@enable	@mode	
Type		int	int	
Description				
Minimum		-2147483648	-2147483648	
Maximum		2147483647	2147483647	
Default Value		0	0	
c#1	SPEC-008			
	All combinations			
d	SPEC-005		Representativ...	3
	Output gb_result.data is 255 shall be			
	(Restrict to combinations for coverage)			

This generates one test case for test analysis item "d". Similarly, while test data for test analysis items "e" and "f" has already been set for test analysis item A, we will set data for the "@enable" variable to clarify how this data corresponds with the requirement specifications. Set this as shown in the figure below.

No.		1		2	
Classification		Argument		Argument	
Variable Name		@enable		@mode	
Type		int		int	
Description					
Minimum		-2147483648		-2147483648	
Maximum		2147483647		2147483647	
Default Value		0		0	
c#1	SPEC-008				
	All combinations				
d	SPEC-005			Representativ...	3
	Output gb_result.data is 255 shall be				
	(Restrict to combinations for coverage)				
e	SPEC-003	Boundary +1	1		
	When enable is ON (TRUE), output				
	(Restrict to combinations for coverage)				
f	SPEC-002	Boundary	0		
	When enable is OFF (FALSE), output				
	(Restrict to combinations for coverage)				

You have now confirmed how each requirement specification and test analysis item correspond to each other and set the test data. The entire Input Data Analysis Table can be seen below.

No.		1			2			3			4		
Classification		Argument			Argument			Argument			Argument		
Variable Name		@enable			@mode			@input1			@input2		
Type		int			int			unsigned char			unsigned char		
Description													
Minimum		-2147483648			-2147483648			0			0		
Maximum		2147483647			2147483647			255			255		
Default Value		0			0			0			0		
A	SPEC-001	T	Boundary +1	1									
	Confirm that the entire feature is	F	Boundary	0									
	(Restrict to combinations for coverage)												
B	SPEC-004				0	Boundary	0						
	The mode shall be switched by mode				1	Boundary	1						
					2	Boundary	2						
					d	Representativ...	3						
						Maximum	###						
	(Restrict to combinations for coverage)					Minimum	###						
a	SPEC-006								Maximum	150			
	input1 shall be selected for output								Minimum	0			
	(Restrict to combinations for coverage)								Repr...	75			
b	SPEC-007											Maximum	150
	input2 shall be selected for output											Minimum	0
	(Restrict to combinations for coverage)											Repr...	75
c	SPEC-008								Maximum	150		Maximum	150
	The additional values of input 1 and								Minimum	0		Minimum	0
	All combinations												
c#1	SPEC-008								Repr...	75		Repr...	75
	All combinations												
d	SPEC-005					Representativ...	3						
	Output gb_result data is 255 shall be												
	(Restrict to combinations for coverage)												
e	SPEC-003		Boundary +1	1									
	When enable is ON (TRUE), output												
	(Restrict to combinations for coverage)												
f	SPEC-002		Boundary	0									
	When enable is OFF (FALSE), output												
	(Restrict to combinations for coverage)												

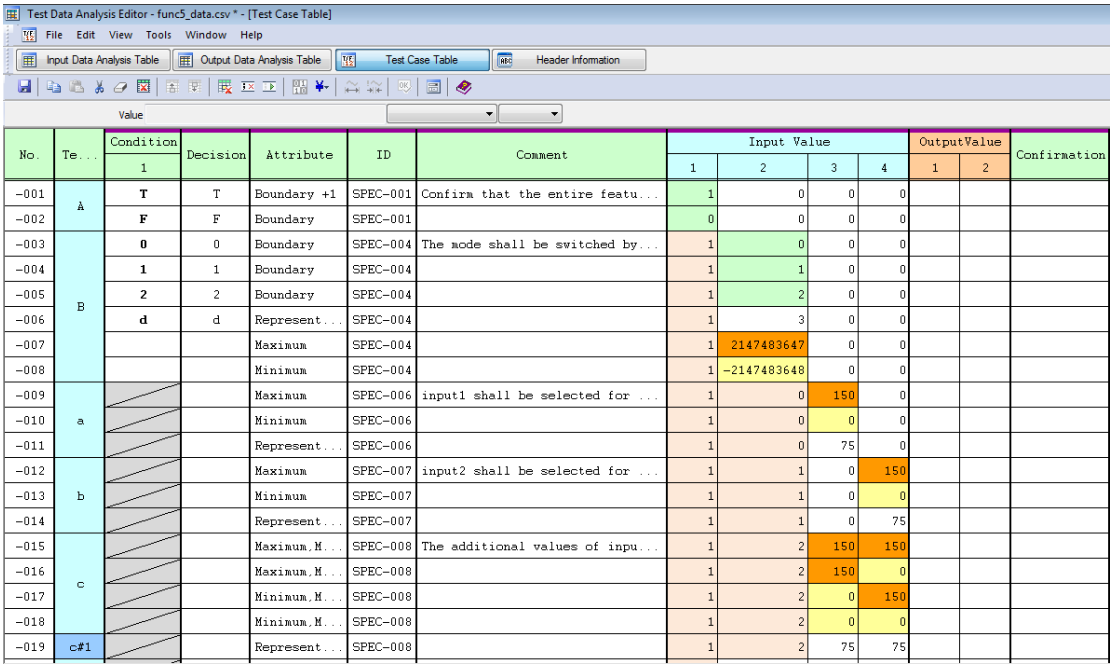
Generating Test Cases

In this step, you will generate test cases from the completed Input Data Analysis Table.

1. Select any cell in the Input Data Analysis Table (in this example, the cell saying "Argument" is selected. To generate test cases for all test analysis items, leave all of the test analysis items unselected.)

2. Select "Generate Test Analysis Item Combinations" from the "Edit" menu.

A Test Case Table is generated like below.



No.	Te...	Condition 1	Decision	Attribute	ID	Comment	Input Value 1	2	3	4	Output Value 1	2	Confirmation
-001	A	T	T	Boundary +1	SPEC-001	Confirm that the entire featu...	1	0	0	0			
-002		F	F	Boundary	SPEC-001		0	0	0	0			
-003	B	0	0	Boundary	SPEC-004	The mode shall be switched by...	1	0	0	0			
-004		1	1	Boundary	SPEC-004		1	1	0	0			
-005		2	2	Boundary	SPEC-004		1	2	0	0			
-006		d	d	Represent...	SPEC-004		1	3	0	0			
-007				Maximum	SPEC-004		1	2147483647	0	0			
-008				Minimum	SPEC-004		1	-2147483648	0	0			
-009	a			Maximum	SPEC-006	input1 shall be selected for ...	1	0	150	0			
-010				Minimum	SPEC-006		1	0	0	0			
-011				Represent...	SPEC-006		1	0	75	0			
-012	b			Maximum	SPEC-007	input2 shall be selected for ...	1	1	0	150			
-013				Minimum	SPEC-007		1	1	0	0			
-014				Represent...	SPEC-007		1	1	0	75			
-015	c			Maximum,M...	SPEC-008	The additional values of inpu...	1	2	150	150			
-016				Maximum,M...	SPEC-008		1	2	150	0			
-017				Minimum,M...	SPEC-008		1	2	0	150			
-018				Minimum,M...	SPEC-008		1	2	0	0			
-019	c#1			Represent...	SPEC-008		1	2	75	75			

The generated Test Case Table contains data attributes, execution logic for branch areas and reference information for the requirement specifications in addition to the input/output data and expected values from the final CSV file input into CoverageMaster.

No.	Test Analysis Item	Condition 1	Decision	Attribute	ID	Comment
-001	A	T	T	Boundary +1	SPEC-001	Confirm that the entire featu...
-002		F	F	Boundary	SPEC-001	
-003	B	0	0	Boundary	SPEC-004	The mode shall be switched by...
-004		1	1	Boundary	SPEC-004	
-005		2	2	Boundary	SPEC-004	
-006		d	d	Represent...	SPEC-004	
-007				Maximum	SPEC-004	
-008				Minimum	SPEC-004	
-009	a			Maximum	SPEC-006	input1 shall be selected for ...
-010				Minimum	SPEC-006	
-011				Represent...	SPEC-006	

Test Case Number

(a unique test case number)

Test Analysis Item

(The ID that appears in the Input Data Analysis Table)

Branch Condition/Decision

(the logical result of the branch to execute)

Test Case Attribute

(the test case attribute entered in the Input Data Analysis Table)

Comment

(comment entered in the Input Data Analysis Table)

The "Condition" and "Decision" items in the above Test Case Table are particularly valuable information for reviewing test cases, as they contain execution logic for the designed test cases if the test analysis items contain condition branches.

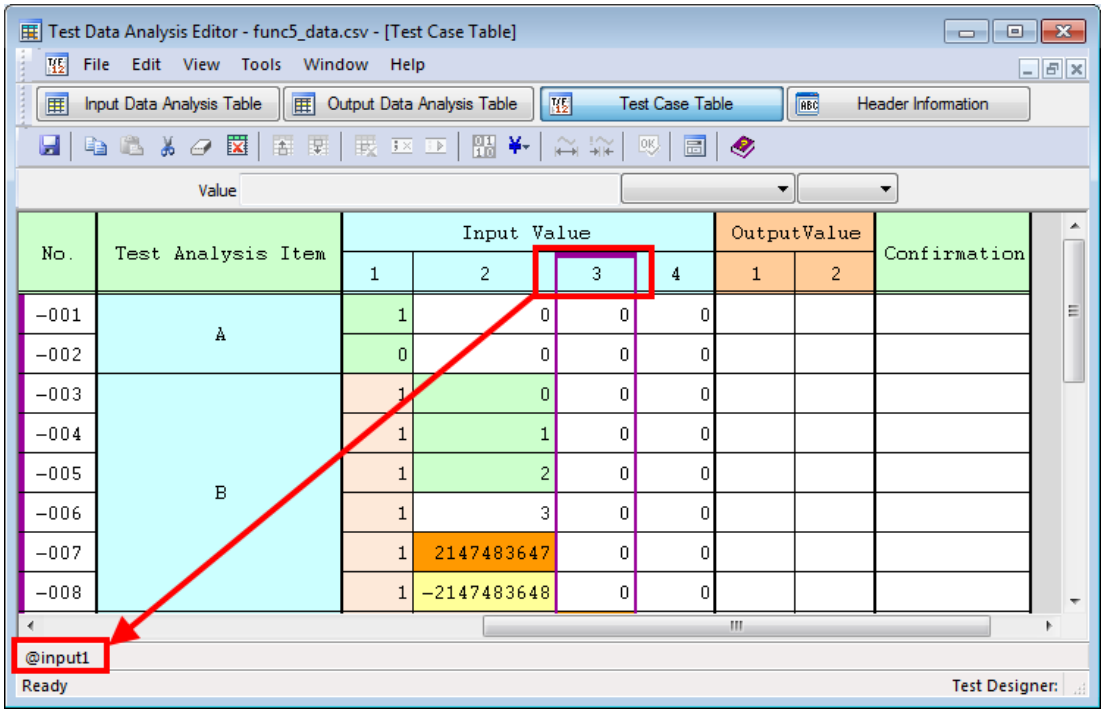
[Reference]

Condition: Execution logic in the form of a condition (in the case of compound conditions, numbers are added and logic is displayed for each number)

Decision: Logic for an entire condition (in the case of compound conditions, logic is

displayed for the condition as a whole)

To reduce the width of the Test Case Table, IDs are displayed in place of all input variable and output variable names. Variable names are displayed in the bar at the bottom of the table when the corresponding test case is selected.



No.	Test Analysis Item	Input Value				OutputValue		Confirmation
		1	2	3	4	1	2	
-001	A	1	0	0	0			
-002		0	0	0	0			
-003	B	1	0	0	0			
-004		1	1	0	0			
-005		1	2	0	0			
-006		1	3	0	0			
-007		1	2147483647	0	0			
-008		1	-2147483648	0	0			

The test data cells are displayed in the colors set for boundary values, minimum values, maximum values, representative values, etc. the same as in the Input Data Analysis Table.

Pale pink cells indicate that a value other than the user's own test analysis items is used. In this case, they indicate that the condition values for branching to the block of the analysis item has been automatically generated.

Input Value			
1	2	3	4
1	0	0	0
0	0	0	0
1	0	0	0
1	1	0	0
1	2	0	0
1	3	0	0
1	2147483647	0	0
1	-2147483648	0	0
1	0	150	0
1	0	0	0
1	0	75	0

Reviewing the Test Cases and Inputting Expected Values

In this section, you will set expected values based on the function design specifications mentioned previously. Function specifications have been provided for reference below. In this operation, the settings need to be configured based on the function specifications, not from the code. Refer to the func5 design specifications, calculate expected values and fill in the "Output Value" field of the Test Case Table with these values.

<Design specifications of func5 function>

Determine values for the two inputs - input1 and input2 - using a mode switching value (mode)

and output to gb_result.data. Note that the entire function is turned on or off using the enable input flag.

Input: (all arguments)

enable (flag) : when feature is OFF output gb_result is (0, FALSE)

mode (variable) : mode switch 0: Output gb_result is (input1, TRUE)

1: Output gb_result is (input2, TRUE)

2: Output gb_result is (input1+input2, TRUE)

default: Output gb_result is (255, TRUE)

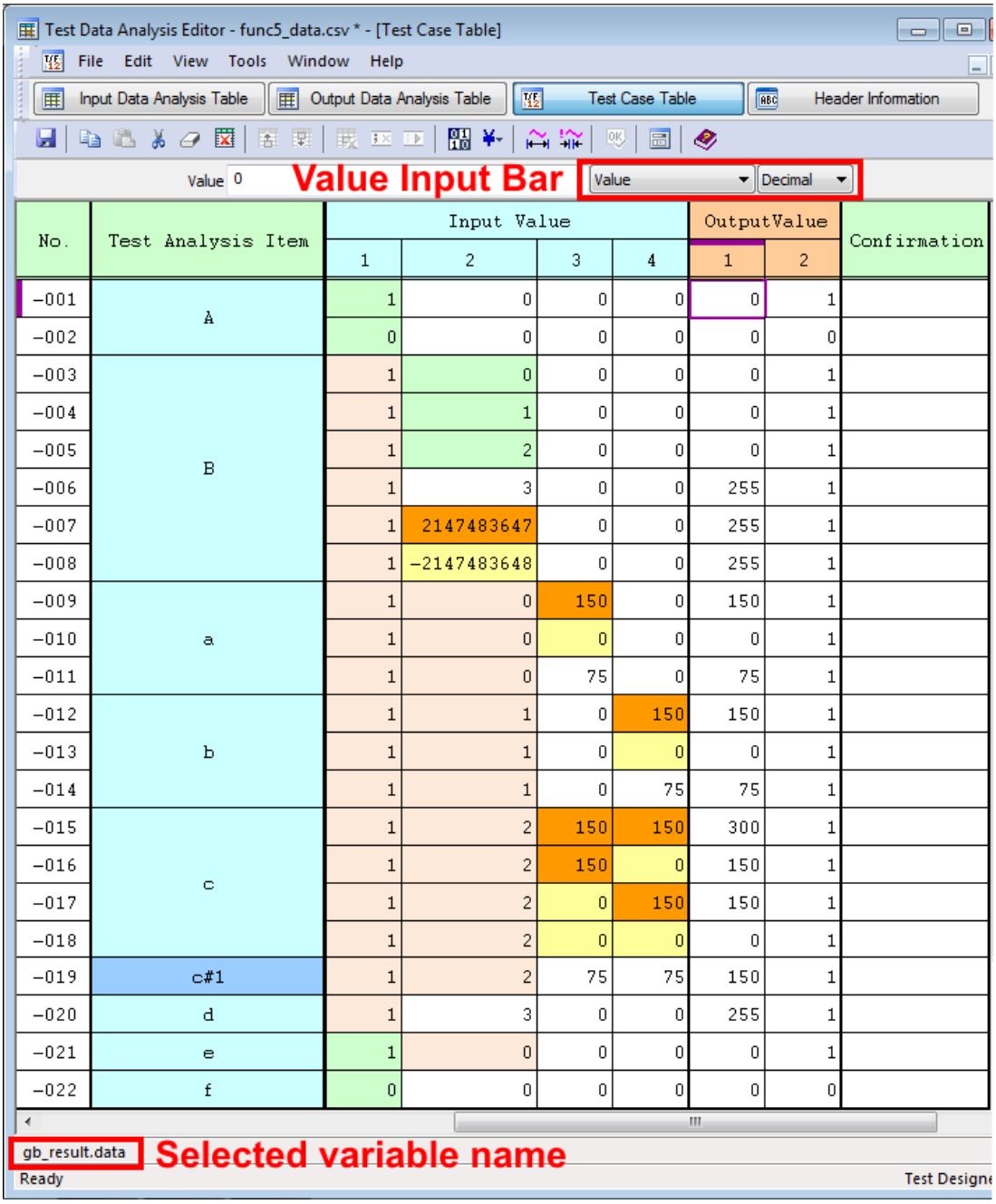
input1(variable) : input1 range (0-150)

input2(variable) : input2 range (0-150)

Output: (global structure)

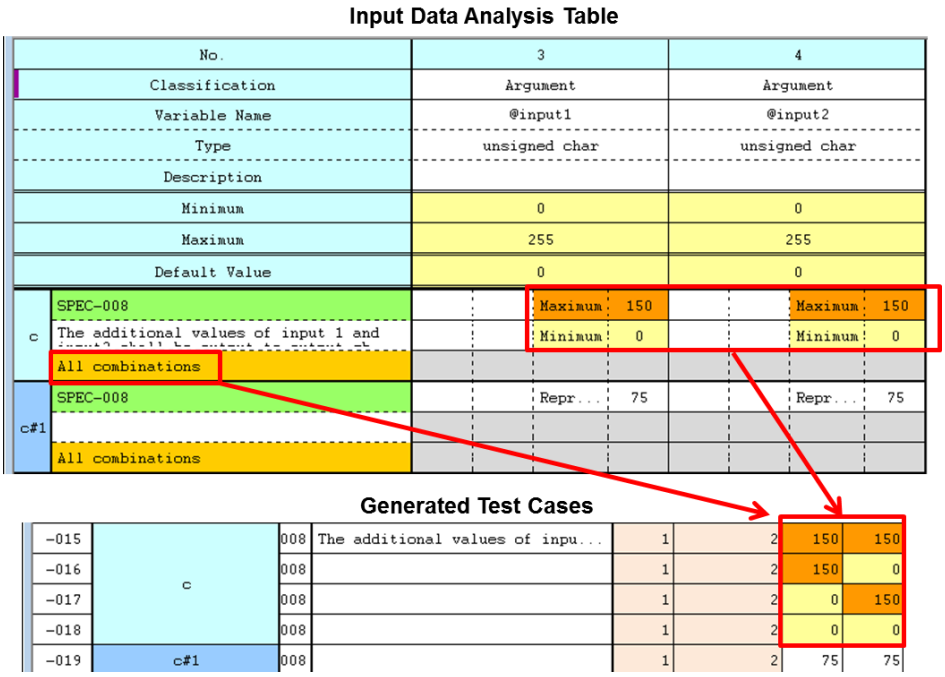
gb_result.data : output data

gb_result.ret_code : error code (TRUE when feature is on; otherwise FALSE)

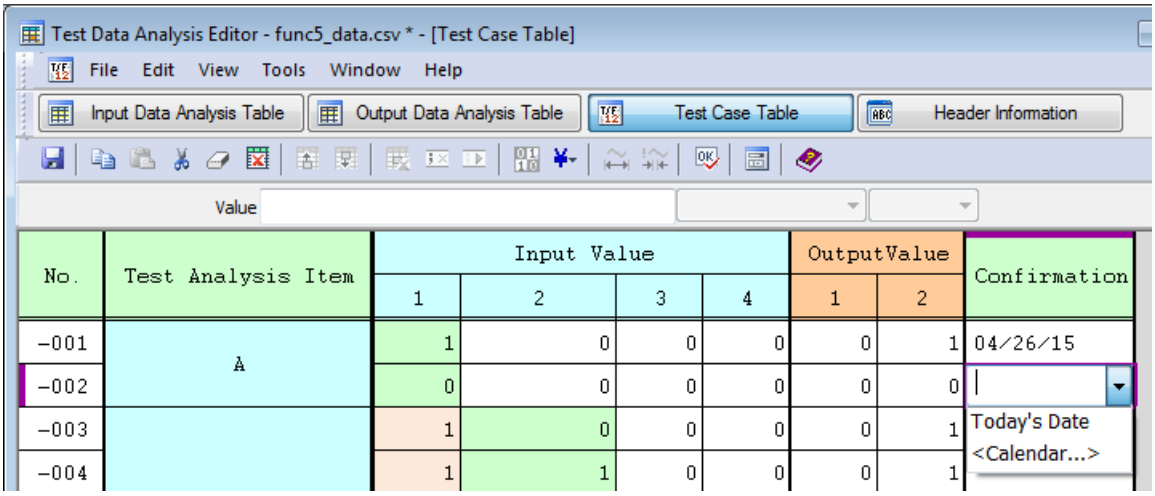


No.	Test Analysis Item	Input Value				OutputValue		Confirmation
		1	2	3	4	1	2	
-001	A	1	0	0	0	0	1	
-002		0	0	0	0	0	0	
-003	B	1	0	0	0	0	1	
-004		1	1	0	0	0	1	
-005		1	2	0	0	0	1	
-006		1	3	0	0	255	1	
-007		1	2147483647	0	0	255	1	
-008		1	-2147483648	0	0	255	1	
-009	a	1	0	150	0	150	1	
-010		1	0	0	0	0	1	
-011		1	0	75	0	75	1	
-012	b	1	1	0	150	150	1	
-013		1	1	0	0	0	1	
-014		1	1	0	75	75	1	
-015	c	1	2	150	150	300	1	
-016		1	2	150	0	150	1	
-017		1	2	0	150	150	1	
-018		1	2	0	0	0	1	
-019	c#1	1	2	75	75	150	1	
-020	d	1	3	0	0	255	1	
-021	e	1	0	0	0	0	1	
-022	f	0	0	0	0	0	0	

For test analysis item "c", this block contains a calculating (adding) process. This means that it has been set to generate combinations of all maximum values and minimum values when combining test cases. Confirm that these combinations are generated.

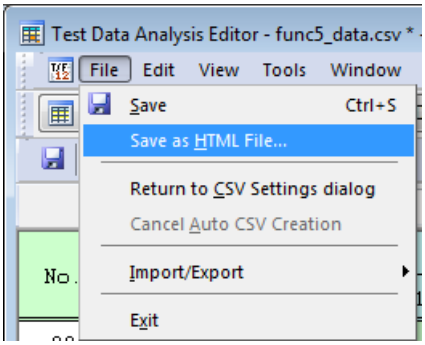


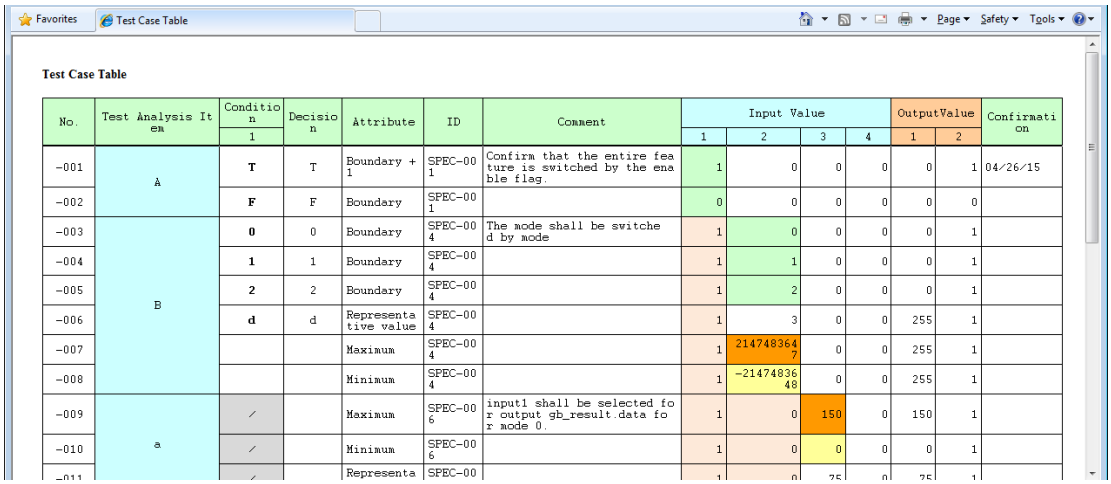
The "Confirmation" field of the Test Case Table can be used to record that the test cases have been confirmed. The current date is input automatically, but other text can be entered if desired.



[Reference] Outputting Each Analysis Table in HTML

Editor images of the Input Data Analysis Table, Output Data Analysis Table and Test Case Table can be saved as HTML files. This makes it possible to verify generated Test Case Tables, etc. offline without launching CoverageMaster.





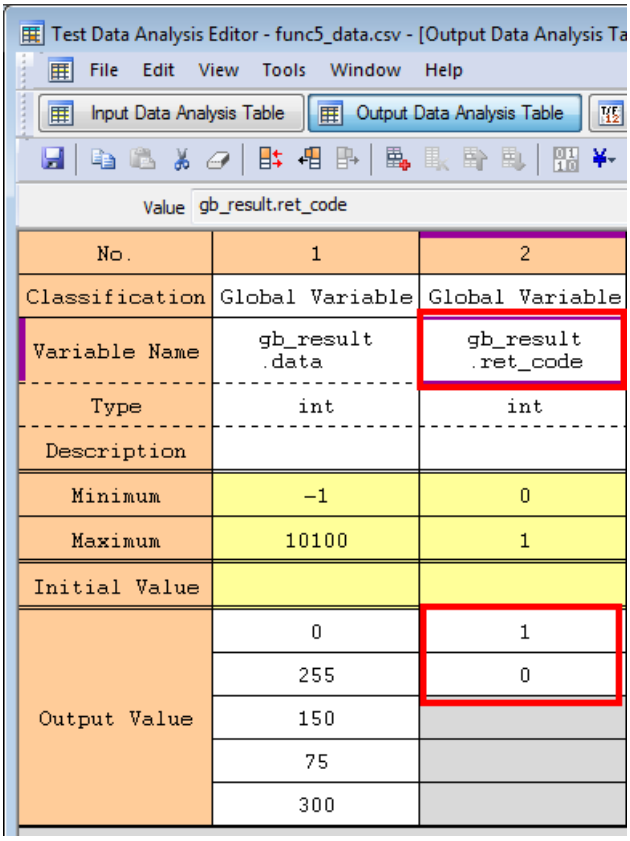
Test Case Table

No.	Test Analysis Item	Condition	Decision	Attribute	ID	Comment	Input Value				Output Value		Confirmation
		1					1	2	3	4	1	2	
-001	A	**T**	T	Boundary + 1	SPEC-001	Confirm that the entire feature is switched by the enable flag.	1	0	0	0	0	1	04/26/15
-002		**F**	F	Boundary	SPEC-001		0	0	0	0	0	0	
-003	B	**0**	0	Boundary	SPEC-004	The mode shall be switched by mode	1	0	0	0	0	1	
-004		**1**	1	Boundary	SPEC-004		1	1	0	0	0	1	
-005		**2**	2	Boundary	SPEC-004		1	2	0	0	0	1	
-006		**d**	d	Representative value	SPEC-004		1	3	0	0	255	1	
-007				Maximum	SPEC-004		1	2147483647	0	0	255	1	
-008				Minimum	SPEC-004		1	-2147483648	0	0	255	1	
-009	a	/		Maximum	SPEC-006	input1 shall be selected for output gb_result.data for mode 0.	1	0	150	0	150	1	
-010		/		Minimum	SPEC-006		1	0	0	0	0	1	
-011		/		Representa	SPEC-00		1	0	75	0	75	1	

Confirming the Output Data Analysis Table

When output values (expected values) are set in the generated Test Case Table, a list of set values is displayed in the Output Data Analysis Table. Repeating data items among the values input as output values in the Test Case Table are unified as single items in this list. In this unit test design, this table indicates data values to be tested as output values.

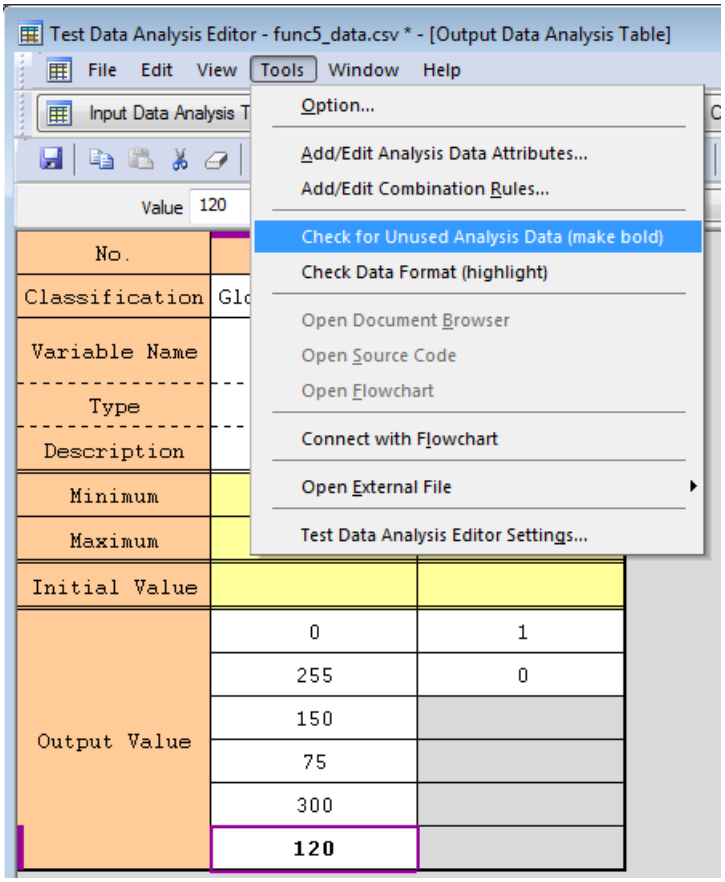
In this case, the global variable "gb_result.ret_code" returns TRUE/FALSE (1/0) as an output. However, test cases with 1 and 0 as the output value need to be created to confirm this specification. 1 and 0 are displayed for gb_result.ret_code in the Output Data Analysis Table. This confirms that there is at least one test case that tests this condition and this test case has not been omitted.



No.	1	2
Classification	Global Variable	Global Variable
Variable Name	gb_result .data	gb_result .ret_code
Type	int	int
Description		
Minimum	-1	0
Maximum	10100	1
Initial Value		
Output Value	0	1
	255	0
	150	
	75	
	300	

[Reference] Configuring an Output Data Analysis Table in Advance and Using This to Confirm Specifications

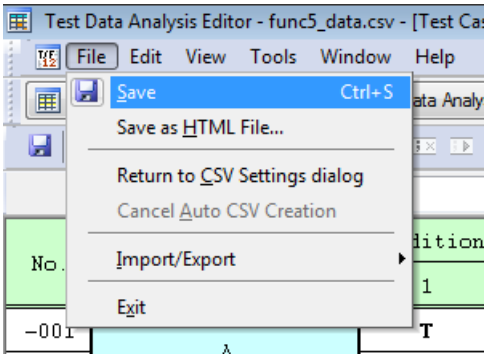
Expected output values (values that need to be tested as output values according to the requirement specifications) can be input in the Output Data Analysis Table before inputting output values (expected values) in the Test Case Table. After configuring the Output Data Analysis Table according to the requirement specifications, it is possible to check whether the output values in the Test Case Table (if output values are set in this table) cover the values in the Output Data Analysis Table. In the example below, "120" is input in the Output Data Analysis Table as an output value that must be tested, but this value does not appear as an output value in the Test Case Table. In cases like this, it is possible to highlight the missing value (display it in bold) by selecting "Check for Unused Analysis Data (make bold)" in the "Tools" menu.



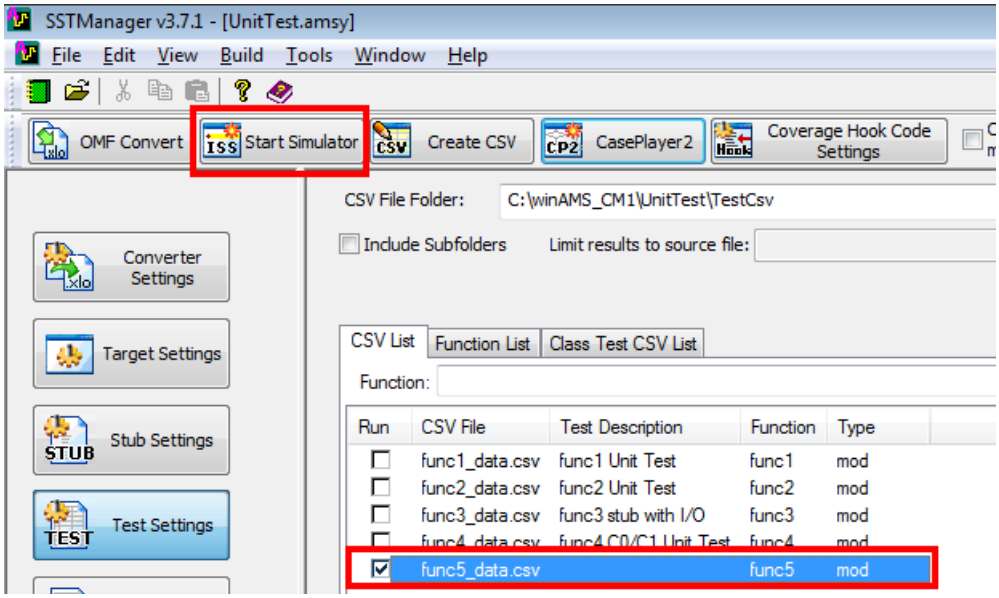
Generating a CSV File and Running the Unit Test

Finally, create a CSV file from the Test Case Table, run the test and then confirm the results.

1. Display the Test Case Table and select the "File" menu - "Save".
   (The data is written to the CSV file at this time.)

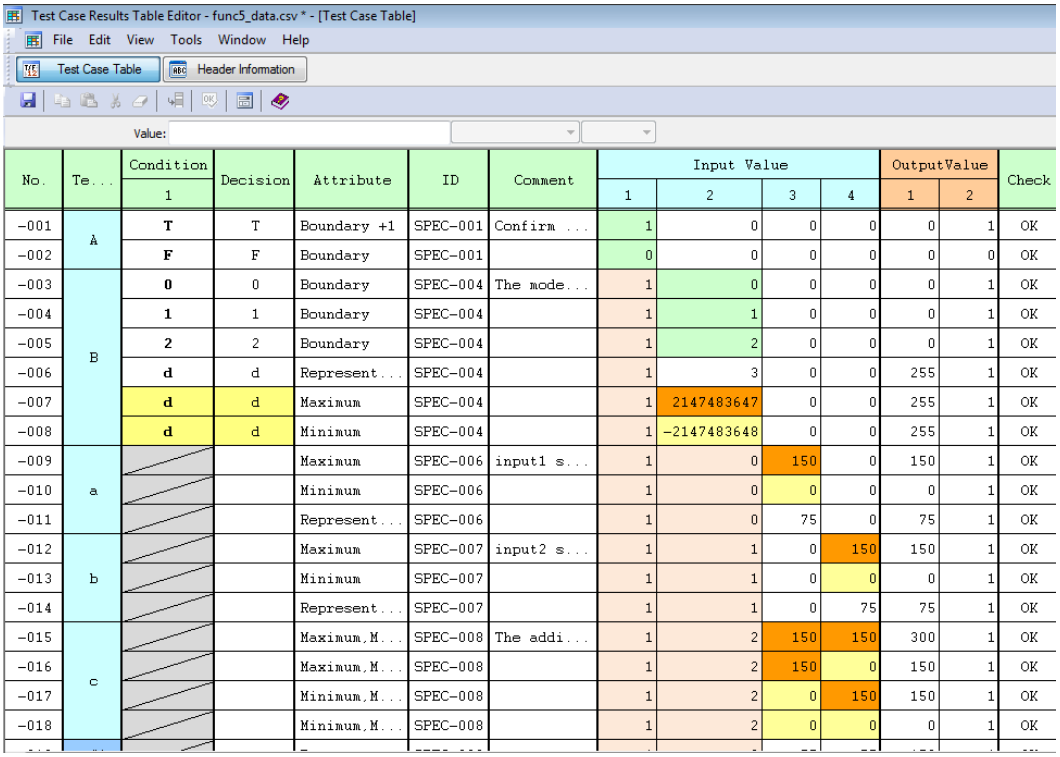


2. Exit the Test Data Analysis Editor.
3. Select "func5_data.csv" in the "Test Settings" view of SSTManager.
4. Click the "Start Simulator" button to run the test.



When the test is completed, verify the results. View the CSV file in CoverageMaster's internal viewer, not directly in Excel. Try displaying other output information along with the results.

5. In the "Other" view of SSTManager, turn off the "Open CSV test result file with external editor" option.
6. Double click "func5_data.csv" in the "Test Results" view.
7. The Test Case Table (output results) is displayed in the internal editor.



No.	Te...	Condition 1	Decision	Attribute	ID	Comment	Input Value 1	Input Value 2	Input Value 3	Input Value 4	OutputValue 1	OutputValue 2	Check
-001	A	T	T	Boundary +1	SPEC-001	Confirm ...	1	0	0	0	0	1	OK
-002		F	F	Boundary	SPEC-001		0	0	0	0	0	0	OK
-003	B	0	0	Boundary	SPEC-004	The mode...	1	0	0	0	0	1	OK
-004		1	1	Boundary	SPEC-004		1	1	0	0	0	1	OK
-005		2	2	Boundary	SPEC-004		1	2	0	0	0	1	OK
-006		d	d	Represent...	SPEC-004		1	3	0	0	255	1	OK
-007		d	d	Maximum	SPEC-004		1	2147483647	0	0	255	1	OK
-008		d	d	Minimum	SPEC-004		1	-2147483648	0	0	255	1	OK
-009	a			Maximum	SPEC-006	input1 s...	1	0	150	0	150	1	OK
-010				Minimum	SPEC-006		1	0	0	0	0	1	OK
-011				Represent...	SPEC-006		1	0	75	0	75	1	OK
-012	b			Maximum	SPEC-007	input2 s...	1	1	0	150	150	1	OK
-013				Minimum	SPEC-007		1	1	0	0	0	1	OK
-014				Represent...	SPEC-007		1	1	0	75	75	1	OK
-015	c			Maximum,M...	SPEC-008	The addi...	1	2	150	150	300	1	OK
-016				Maximum,M...	SPEC-008		1	2	150	0	150	1	OK
-017				Minimum,M...	SPEC-008		1	2	0	150	150	1	OK
-018				Minimum,M...	SPEC-008		1	2	0	0	0	1	OK

In addition to the decision outcomes of the output values and expected values, actual branch item conditions of test analysis items containing branches are output to the Test Case Table. This logic is displayed based on actual results of measurements performed by the simulator.

(Conditions are also displayed in the Test Case Table used when designing the input data; however, this logic is from designs based on analysis in CasePlayer2.)

Input Value				OutputValue		Check
1	2	3	4	1	2	
1	0	0	0	0	1	OK
0	0	0	0	0	0	OK
1	0	0	0	0	1	OK
1	1	0	0	0	1	OK
1	2	0	0	0	1	OK
1	3	0	0	255	1	OK
1	2147483647	0	0	255	1	OK
1	-2147483648	0	0	255	1	OK

Results match with Expected values

No.	Te...	Condition	Decision
		1	
-001	A	**T**	T
-002		**F**	F
-003	B	**0**	0
-004		**1**	1
-005		**2**	2
-006		**d**	d
-007		**d**	d
-008		**d**	d

Logic displayed based on actual results of measurements performed by the simulator

This concludes the tutorial on test design using the Test Case Analysis Editor (Tutorial 5).

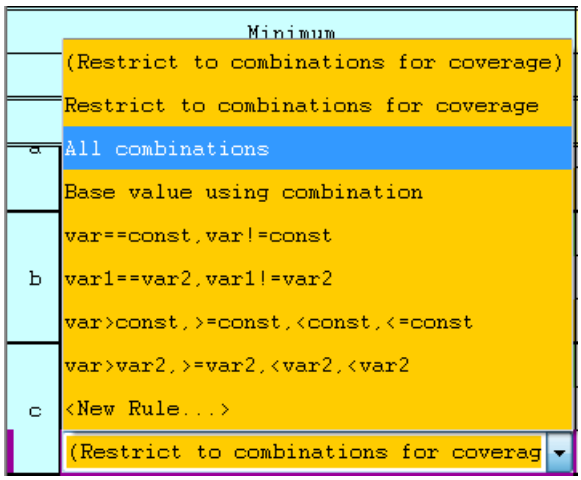
[Reference] Test Analysis Item Combination Rules

This section explains how test data is combined to make a Test Case Table from the Input Data Analysis Table. In Tutorial 5, we partially applied "All combinations" and did not handle any other details. However, the features and mechanisms concerning these combinations need to be understood in order to generate test cases to run tests as planned.

Default Combination Rules

As a general rule, test data input in the Input Data Analysis Table is always output to the Test Case Table at one point when creating combinations. All data is output to the Test Case Table, regardless of whether it is input in the Input Data Analysis Table. However, while this basic rule guarantees that the data will be output to the Test Case Table, this does not determine how it will be combined with data assigned to other variables. Applying the combination rules explained here makes it possible to generate test cases that take into account combinations with data assigned to other variables.

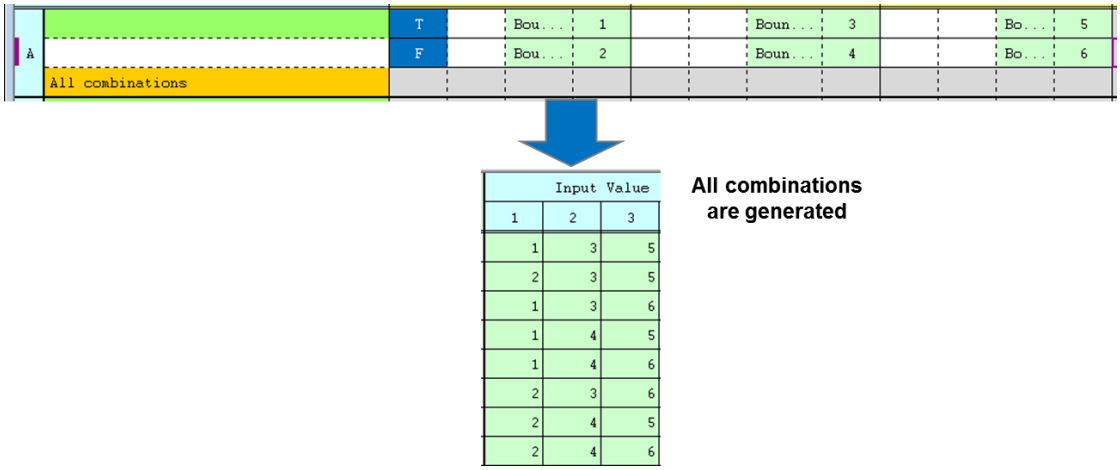
First, here is an explanation of the combination rules set in the Input Data Analysis Table by default. The following 3 options are preset as combination rules in the tools.



■ All combinations

The simplest combination rule. Combinations are created from all data, regardless of the flags and attributes assigned to each variable. For example, when using this rule to create combinations when two data items are assigned to each of three variables, 8 test cases (2x2x2) are generated to combine all of these data items.

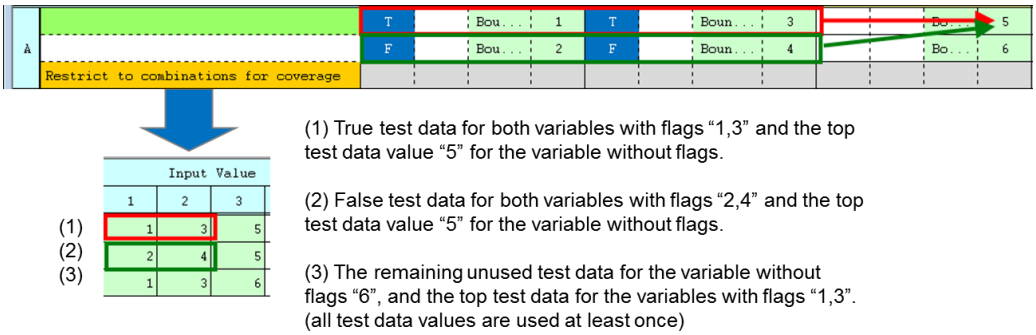
Apply this rule to test analysis items in which the variables are closely related and there is a high likelihood that defects dependent on each data combination may occur due to factors such as operator components with multiple variables or branching with complicated compound conditions. The "All combinations" rule generates the largest number of test cases but creates 100% coverage, minimizing the risk of test omissions occurring as a result of insufficient combinations.



■ Restrict to combinations for coverage

This combination rule controls combinations by using the TRUE flags, FALSE flags and case label flags assigned to branch conditions. In test analysis items containing branches, these flags are set for test cases corresponding to the logic of the branches. When creating combinations for flagged variables with this rule, one combination each is created between test data items containing the same flag. For the data of unflagged variables, combinations are decided by the tool, and combinations are generated only with the topmost data item in the data set for other variables.

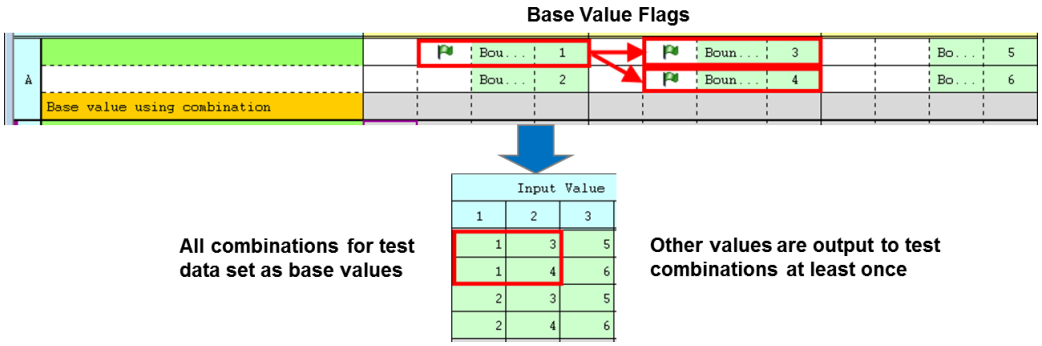
Apply this combination rule when you want to reliably create combinations of conditions that cover the code (achieve coverage) with the minimum possible number of generated test cases.



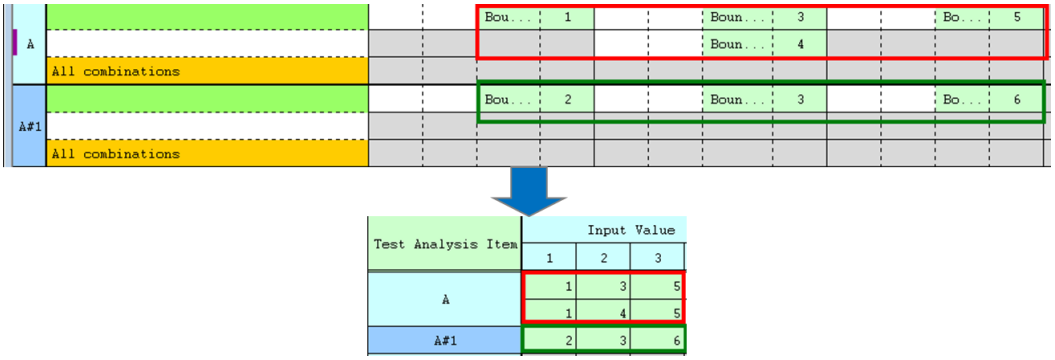
■ Base value using combination

This combination rule controls combinations using the Base Values flags set for each test data item in the Input Data Analysis Table. The tool creates all combinations of data items specified as base values in the variables set for each test analysis item, but this is not done for other data items.

Specifically, all combinations of data items specified as base values are generated, while for other analysis data items that are not set as base values, test cases are created so that each data item is output at least once. These data combinations are determined by the tool.

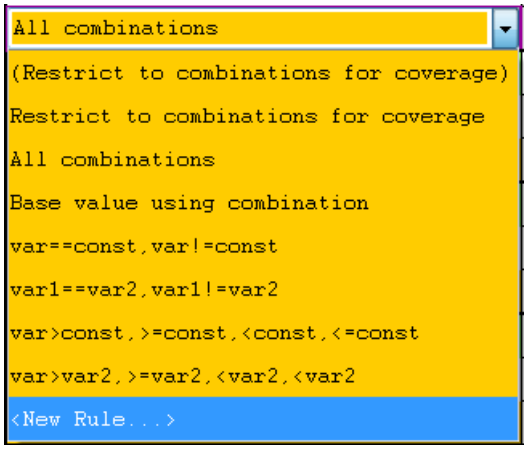


As this method uses the base values, it can be used when you want to guarantee coverage of test case combinations only for parts of the data in each test analysis item. However, it may take a long time to set base values for large amounts of test data, and it may be difficult to guarantee the coverage of the test cases that are ultimately output, as combinations of data items that are not set as base values are determined by the tool. In such cases, the coverage of test cases can be confirmed more easily by dividing the test analysis items into multiple sets and applying all combinations for each test analysis item like shown below.

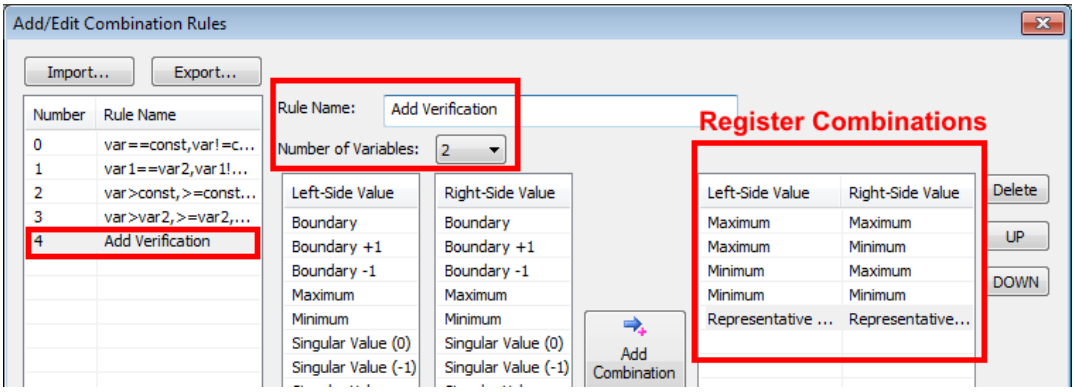


Creating Combination Rules

In addition to the three combination rules set in the Input Data Analysis Table by default, users can create their own combination rules based on attributes they have set for the data.



To add or edit a combination rule, select "<New Rule...>" from the pull-down menu for selecting combinations in the figure above, or select "Add/Edit Combination Rules" from the "Tools" menu.



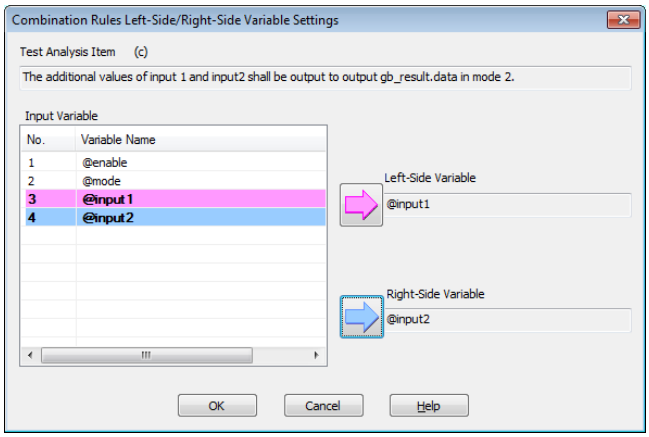
In the example in the figure above, a rule called "Add Verification" is created. This rule is set so that all combinations of data with the "maximum value" and "minimum value" attribute are created and combinations between representative values are created when creating combinations of data assigned to two variables.

This feature can be used in this way to specify data for which to create all combinations, using the attributes of the data set in the Input Data Analysis Table. Combinations of data items with attributes that are not specified for these combinations will be created so that each data item is output to a test case at least once.

For example, we will create a combination for the operator confirmation component from Tutorial 5 by applying this new rule that has been created.

No.		3			4		
Classification		Argument			Argument		
Variable Name		@input1			@input2		
Type		unsigned char			unsigned char		
Description							
Minimum		0			0		
Maximum		255			255		
Default Value		0			0		
c	SPEC-008		Maximum	150		Maximum	150
	The additional values of input 1 and		Minimum	0		Minimum	0
	Add Verification		Repr...	75		Repr...	75

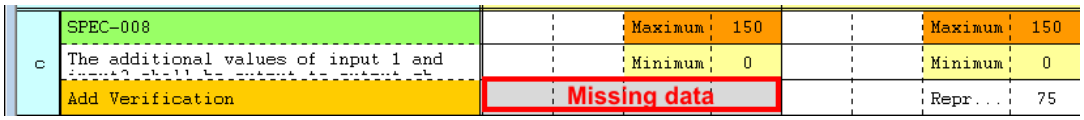
When the rule that has been created ("Add Verification") is applied, the tool first displays a screen for specifying which variables to use as the "Left-Side Value" and "Right-Side Value" set in the rule.

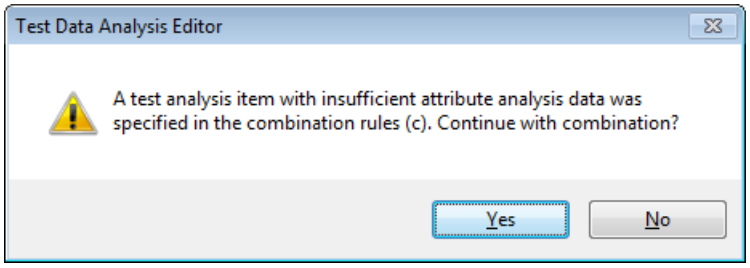


By specifying input1 and input2 respectively, the following combinations are generated.

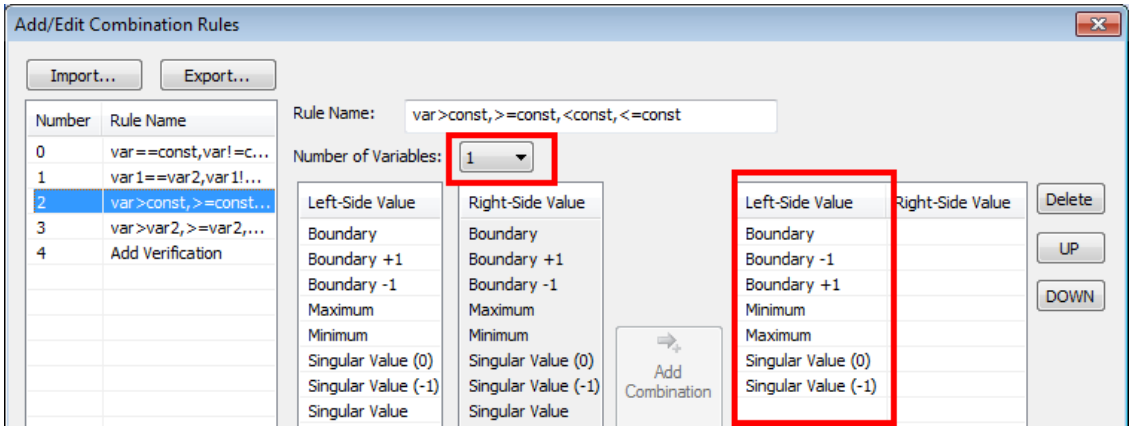
No.	Test Analysis Item	Attribute	ID	Comment	Input Value			
					1	2	3	4
-015	c	Maximum,Maximum	SPEC-008	The a...	1	2	150	150
-016		Maximum,Minimum	SPEC-008		1	2	150	0
-017		Minimum,Maximum	SPEC-008		1	2	0	150
-018		Minimum,Minimum	SPEC-008		1	2	0	0
-023		Representative value,Representative value	SPEC-008		1	2	75	75

The tool also confirms whether the data with the attributes specified for the combinations is set in the Input Data Analysis Table when the created rule is applied. If, for example, the new rule shown above is applied when no representative value data is input, a dialog appears to warn the user that there is insufficient data to create the combinations.





The number of variables can also be set as "1" in the "Add/Edit Combination Rule" window. Specifying only one variable means that the tool simply outputs the input data as it is, without creating combinations. However, a warning like the one shown above is displayed if data with the attributes specified for combinations does not exist in the Input Data Analysis Table. "1" can therefore be specified as the number of variables to check for insufficient data.



Conclusion

This tutorial explained representative ways to use this tool, particularly for confirming how test items correspond with the requirement specifications. We hope that this tool and the operations you have learned in this tutorial will be useful in solving issues you may currently be facing, such as reducing labor hours and improving the quality of test design.

If a unit test workflow has already been established, you will also need to consider how to change the existing workflow in order to create a design that makes use of the benefits of the Test Case Analysis Editor.

The Test Case Analysis Editor is a requirement specifications-based test design support feature that cannot be found in other unit testing tools. We hope that it will be useful to you as a standard tool for confirming the quality of embedded software that you have developed.

[Application] Measuring Coverage by Hook Code

Introduction

As you learned in Tutorials 1 to 4, CoverageMaster can measure C0 and C1 coverage using the "target code", that is the same code used in the actual product. In some cases, however, C0 and C1 coverage cannot be measured accurately as the logical structure of the C source code and the structure of the generated assembler code do not match, due to optimization of the cross compiler. Furthermore, it is not possible in principle to perform measurement from the object code that was used to compile the original source code when performing MC/DC measurements that evaluate not only the coverage of branches but whether execution is possible with coverage of all compound conditions.

CoverageMaster therefore contains a feature that performs coverage measurements using hook code to avoid influence from cross compiler optimization when measuring C0 and C1 coverage and evaluate the coverage of compound conditions that is required for MC/DC coverage.

In this application manual, you will learn the principles of coverage measurement using hook code, how to create an environment for this, and how to use this feature. This section is the final part of the CoverageMaster tutorial that has been described here, and is for users who have mastered the basics of using CoverageMaster.

*A license for the MC/DC Option is required in order to use the MC/DC measurement function. C0 and C1 measurement using hook code is supported as a standard feature.

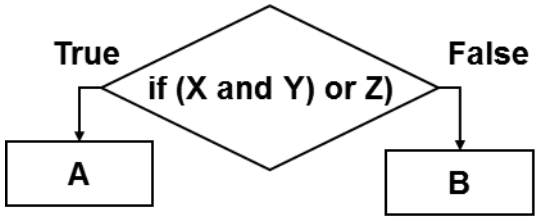
What is MC/DC? (Background Information)

An overview of MC/DC is provided as background information for CoverageMaster's coverage measurement feature using hook code. Please also take a moment to read this as a reference for C0 and C1 measurement.

Condition Coverage Covering Compound Conditions

Condition coverage (C2) is a coverage index that generally has a higher degree of coverage than branch coverage (C1). Branch coverage simply evaluates whether the TRUE/FALSE of a branch was run at least once, and does not cover combinations of the logic of each condition in a compound condition. Condition coverage can be used to test all combinations of the logic of each condition if it is determined that branch coverage provides insufficient coverage.

For example, if a branch contains a compound condition with conditions X, Y and Z, there are eight possible combinations of condition logic. Condition coverage needs to test all of these cases.



X	F	F	F	F	T	T	T	T
Y	F	F	T	T	F	F	T	T
Z	F	T	F	T	F	T	F	T

Combinations of condition logic required for condition coverage

Determining MC/DC Test Cases

However, when all combinations of condition logic are taken into account in condition coverage (C2), some invalid test cases occur among these combinations. For example, if the logic of (X and Y) in the compound condition shown above is TRUE, the overall logic of the (X and Y) or Z condition is determined at this point to be TRUE, regardless of the logic of Z. This means that there is no point in testing both of the two logic combinations at the right end of the logic table shown above (the logic enclosed in the green box). Only one of the two needs to be tested.

MC/DC (Modified Condition/Decision Coverage) functions as a method of examining each condition and confirming the effect of changes in the logic instead of using superfluous combinations of logic that occur in condition coverage (C2).

MC/DC test cases are derived as follows.

> The tool examines one condition in a compound condition and extracts two combinations of logic in which a change in the logic of this individual condition changes the logic of the entire compound condition, making these test cases for the condition that was examined.

For example, when condition X is examined, the tool finds logic combinations in which the entire logic of the compound condition is changed by changing only the TRUE/FALSE status of X, without changing Y or Z. These are set as test cases for X.

Below are examples of logic combinations that fulfill this condition.

> (X, Y, Z) is (FALSE, TRUE, FALSE) and (TRUE, TRUE, FALSE)

Logic combinations are then derived for Y and Z in the same way. The four test cases highlighted in green in the table below are ultimately determined to be test cases that are required for MC/DC evaluation of (X and Y) or Z. The test cases other than those highlighted in green are logic combinations that do not need to be tested in the MC/DC evaluation.

X	F	F	F	F	T	T	T	T
Y	F	F	T	T	F	F	T	T
Z	F	T	F	T	F	T	F	T
Outcome	F	T	F	T	F	T	T	T

2 test cases Z X Y

Mechanism of CoverageMaster's Coverage Measurement Using Hook Code

MC/DC Measurement Using Hook Code

MC/DC determines and evaluates test cases as mentioned previously. When doing this, however, it is necessary to measure whether the executed logic was TRUE/FALSE for each condition in compound conditions included in the conditional statements of a given test case.

For example, if a function includes a conditional statement like the one below:

if((x>10 && y>20) || z>30)

it is necessary to measure whether the executed logic of each condition (x>10, y>20, z>30) is TRUE or FALSE when executing the compound condition.

However, in the current status, it is not possible in principle to detect the executed logic of each condition when the above compound condition is compiled into object code. In order to determine the executed logic of each condition, it is necessary to run each of the conditions (x>10, y>20, z>30) individually to detect the execution status.

CoverageMaster measures MC/DC using a method that detects the executed logic of each of the above conditions by sending the logic of each condition to a separate function (hereinafter "hook function") as an argument. The values of the arguments sent to these functions are determined by CoverageMaster. Specifically, the above compound condition is changed to the following code for execution.

if(Hook(Hook(x>10) && Hook(y>20)) || Hook(z>30))

The Hook() function is designed so that argument values can be sent to CoverageMaster and CoverageMaster can determine the logic of the condition at that time. Hook functions of this nature need to be embedded in the test code for MC/DC measurement. The source code of the hook function is automatically generated by CoverageMaster.

```
int Hook( int condition )
{
        [mechanism for sending condition value to CoverageMaster] ;
        return condition; // logic result sent as an argument is returned as it is
}
```

In CoverageMaster, the code in which code for measuring coverage is added to the original evaluated function is referred to as "hook code". This hook code is automatically generated by CasePlayer2's source code analysis feature.

C0 and C1 Coverage Measurement Using Hook Code

It is also possible to measure C0 and C1 coverage without influence from optimization by using hook code, in which a hook function similar to the one shown above is inserted in the code. First, here is an explanation of an example in which C1 coverage is measured.

For example, a switch statement may contain the following code.

```
switch (mode){
      case 0:
            gb_out = 10;      // assignment processing
            break;
      case 1:
            ct+;
            break
      case 2:
            gb_out = 10;      //assignment processing
            break;
      default:
            break;
}
```

As the processes of case 0 and case 2 are the same, the cross compiler may merge case 0 and case 2 when compiling this code to decrease the code size. In the C-language image, the assembler code is created with the following code structure.

```
switch (mode){
    case 0:
    case 2:
        gb_out = 10;    //assignment processes are merged
        break;
    case 1:
        ct+;
        break
    default:
        break;
}
```

In this case, the same code is used for the conditions of both case 0 and case 2. As CoverageMaster measures coverage by executing compiled object code, this means that CoverageMaster cannot in principle distinguish between the two branches, and is thus unable to accurately measure the coverage.

To accurately measure which line was run, code is generated with a hook function that leaves a "footprint" in each branch position (footprint function). Below is an image of the code after this is inserted.

```
switch (mode){
    case 0:
        FootPrint(1); gb_out = 10;     // assignment processing
        break;
    case 1:
        FootPrint(2); ct+;
        break
    case 2:
        FootPrint(3); gb_out = 10;     //assignment processing
        break;
    default:
        FootPrint(4); break;
}
```

When this code is run, arguments sent by the hook function **FootPrint**() (branch numbers) are detected, making it possible to accurately measure which branch was run. Source code with a hook function is automatically generated by CoverageMaster.

While the above example is for C1 coverage measurement, CoverageMaster also includes a feature for measuring C0 coverage without influence from factors such as code omissions caused by optimization by inserting a hook function that leaves footprints not only on branches but in all source lines.

Mechanism for Keeping the Test Faithful to the Target Code

In order to measure coverage using hook code, it is necessary to use hook code, the code described previously in which a hook function is embedded for coverage measurement. However, the fact that a hook function is added to the actual tested function means that this code is now different from the actual target code used in the product. This means that CoverageMaster has lost its advantage: the ability to perform unit test testing using the target code to ensure faithful testing of the target code.

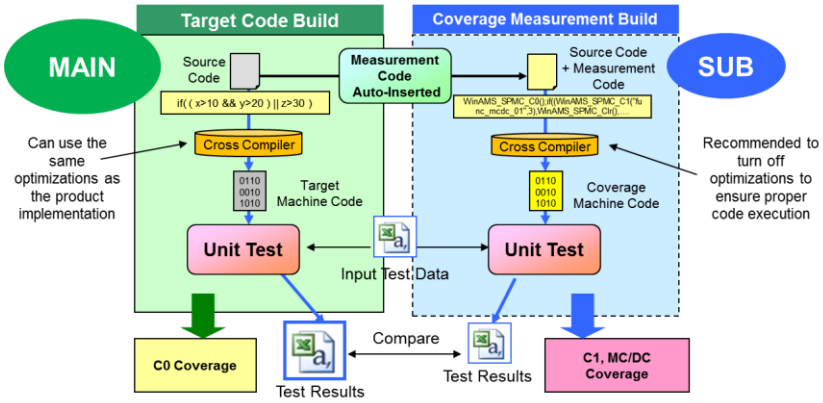
To keep the test as faithful to the target code as possible, CoverageMaster has a mechanism for executing reliable tests for coverage measurement and unit testing. Both the unmodified target code build, and the coverage measurement (hook code) build are executed and their results

automatically compared.

Simultaneous Execution of Target Code and Hook Code

The figure below indicates CoverageMaster's mechanism for measuring coverage using hook code. The "Target Code Build" on the left side indicates the unmodified build environment for the development of the product. The "Coverage Measurement Build" is created separately in addition to this, and hook code is inserted to the coverage measurement build. The build environment of the target code is duplicated folder by folder and the hook code is inserted into the duplicated build for coverage measurement. The hook code is generated automatically in CasePlayer2 based on analysis of the source code.

The coverage measurement build is compiled into an executable object file using the same cross compiler as the target code build.



Function Output Values Are Obtained from the Target Code and Only the Coverage Results Are Obtained using the Hook Code

Test cases are assigned both to the target code and the hook code when executing unit tests in CoverageMaster. The output values of each function are obtained from the target code to ensure faithful and reliable testing. Only the coverage results are obtained using the hook code. These are both integrated to create the overall test and coverage results.

The target code can be run by CoverageMaster's simulator feature at a command level to obtain running results equivalent to the actual device. The hook code is compiled using the same cross compiler and run by the same simulator. However, information on the coverage running results is obtained from the logic actions (C-source level actions) of the hook code. The compile settings applied to the target code (optimization, etc.) can be the same as those used when building the code to be implemented in the product. For the hook code, however, it is recommended to turn off compiler optimizations when building the code to ensure correct operation of the coverage measurement mechanism.

Feature for Confirming that the Hook Code Has Not Influenced the Test

CoverageMaster has a feature for confirming whether the output values of the variables set for the output conditions are the same in both the target code and the hook code, to confirm that the hook code added for coverage measurement is not affecting the essential features of the function.

If the output values match for all test cases, this means that the measurement results are still reliable and that the inserted code does not affect any features of the function itself, such as branches or operators or the function.

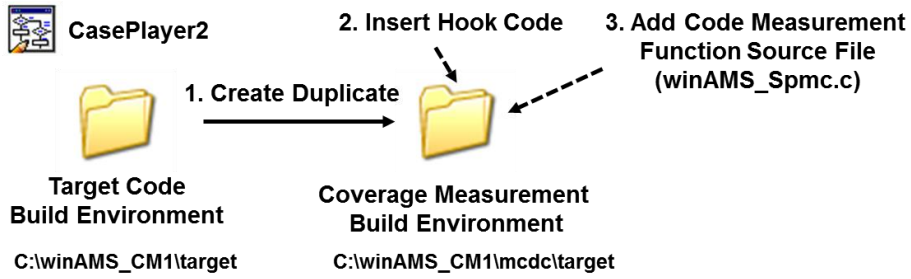
Create the Coverage Measurement Build Using Hook Code

Here is an explanation on how to create an environment for measuring coverage using hook code, based on the test environment created during the tutorial.

Workflow for Creating a Coverage Measurement Build Environment

To measure coverage using hook code, duplicate the target code build environment you created in the tutorial to create a dedicated build for coverage measurement. This is done by the following procedure.

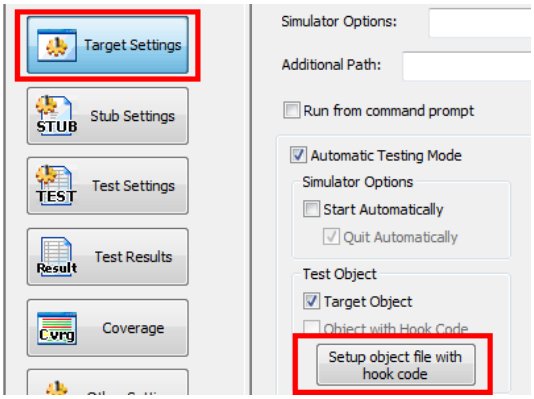
1. Duplicate the "Target Code" build environment using the features in CasePlayer2.
2. CasePlayer2 inserts code measurement hook code into the source code of the duplicated build environment.
3. Add the code measurement function source file (winAMS_Spmc.c) generated by CasePlayer2 into the coverage measurement build environment.
4. Build the coverage measurement build environment to create an executable object code for measuring the coverage.



Duplicating the "Target Code" Development Environment

Duplicate each folder of the target code build environment in the dedicated build environment for coverage measurement. Use the "Setup object file with hook code" feature in CasePlayer2 to duplicate the folders.

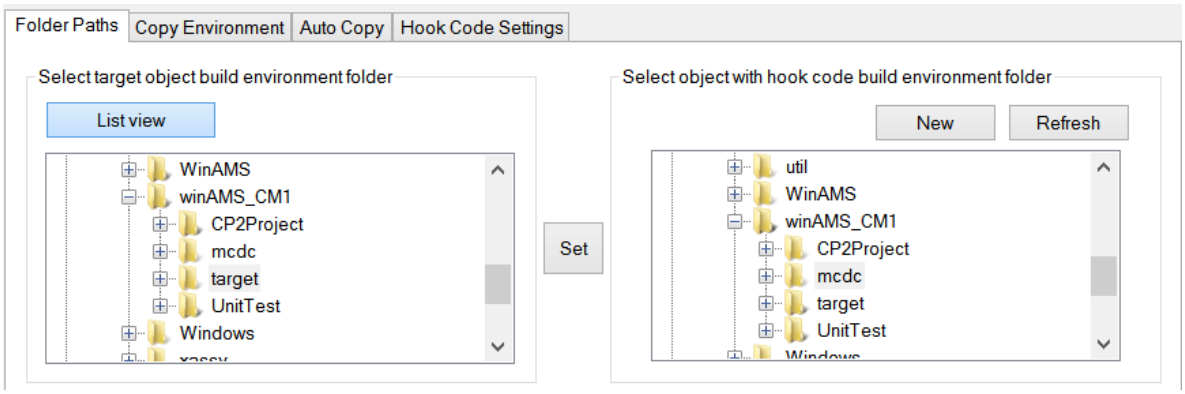
The duplicated folders should have the same folder name and structure as the folders in the target code build environment (the "target" folder in the case of the tutorial). To avoid the issue of being unable to create two folders with the same name, create a new "mcdc" folder in "c:¥winAMS_CM1" and create the duplicates in this folder. (This folder can be named as desired.)



1. Click the "Setup object file with hook code" button in the "Target Settings" view.
2. In the "Select target object build environment folder" tree view , select “C:¥winAMS_CM1¥target"
3. In the "Select object with hook code build environment folder" tree view, select "C:¥winAMS_CM1" and click the "New" button and create a folder named "mcdc".

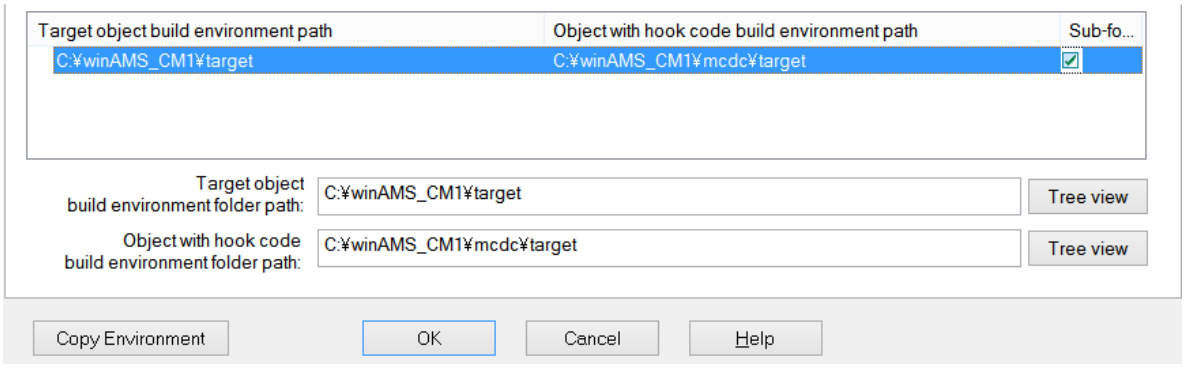
The mcdc folder is created in the winAMS_CM1 folder. (The “mcdc” folder can be created in

Windows Explorer instead.)



4. Select “C:¥winAMS_CM1¥target" on the left.
5. Select “C:¥winAMS_CM1¥target¥mcdc" on the right.
6. Click the “Set” button in the center. (The selected paths will be added to the list below.)

The target object build environment folder and object with hook code build environment folder paths are registered to CasePlayer2.

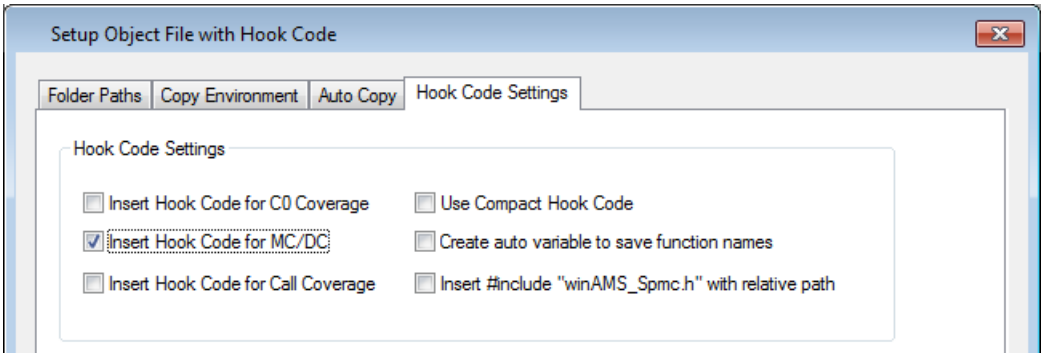


Next, confirm the type of coverage to insert hook code for.

7. Select the "Hook Code Settings" tab.
8. For MC/DC measurement, check the "Insert Hook Code for MC/DC" option. Uncheck this option if only measuring C1 coverage. (Note: Hook code for C1 coverage is output by default, regardless of whether this option is turned on or off.)

When using coverage measurement hook code, C1 coverage is measured using the hook code by default.

For C0 coverage, it is possible to select whether to obtain coverage from the target code or the hook code. C0 coverage can be obtained from the hook code when the "Insert Hook Code for C0 Coverage" option is checked.

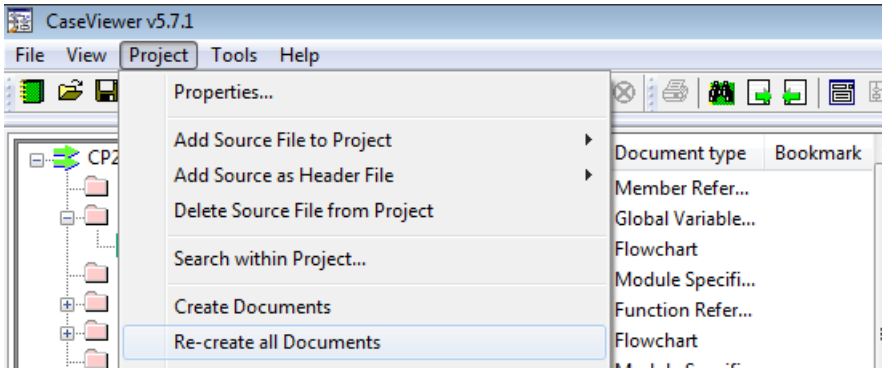


9. Check the "Insert Hook Code for C0 Coverage" option if you wish to measure C0 coverage using hook code.
10. Verify that the source files to insert hook code into (main.c in the tutorial example) are checked
11. Click the "Copy Environment" button in the "Setup Object File with Hook Code" dialog. A copy of the target folder is created in the mcdc folder.

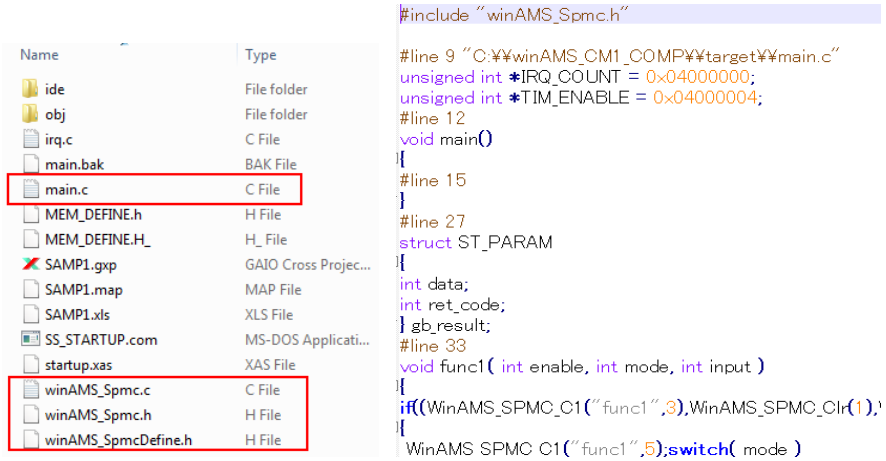
Inserting the Hook Code

Insert the hook code for coverage in the source code of the duplicated coverage measurement build environment. This process is performed in CasePlayer2.

12. Select "Re-create all Documents" from the "Project" menu in CasePlayer2.



The hook code is written to the source code files of the duplicated environment. A source file containing the coverage measurement functions (winAMS_Spmc.c) is also generated at this time in the folder.



The source file containing the coverage measurement functions (winAMS_Spmc.c) and the main.c file after the hook code is inserted

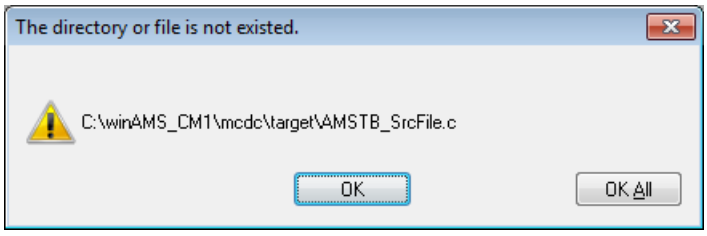
Add the Coverage Measurement Function Source File to the Build Environment

Add the source file containing the coverage measurement functions (winAMS_Spmc.c) to the coverage measurement build environment.

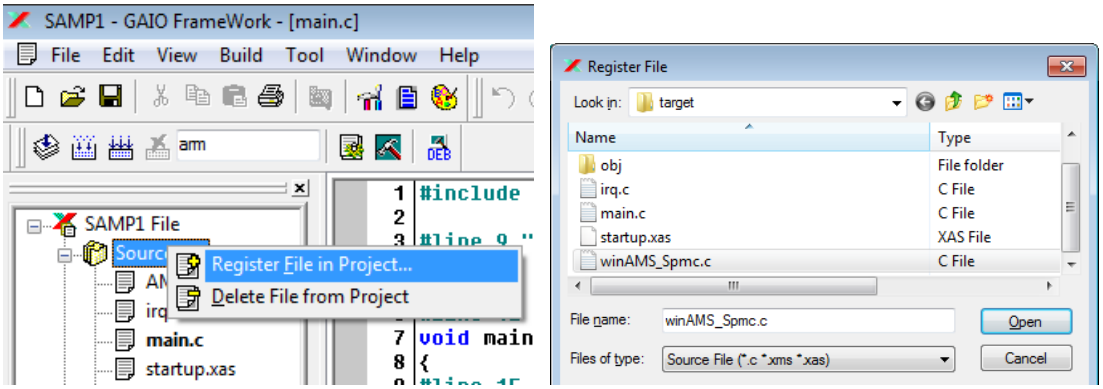
This tutorial uses GAIO Framework, GAIO's cross compiler. For practical purposes, use the development environment used to develop your product.

1. Double click "SAMP1.gxp" in "C:¥winAMS_CM1¥mcdc¥target" (the coverage measurement build) to open the build project in GAIO Framework.

The following error dialog may appear. The displayed file is the source file for stub functions, but this source file cannot be referenced because it is not included in the folder of the duplicated build environment. In this case, click the "OK" button to close the dialog and delete "AMSTB_SrcFile.c" from the Source File folder. Then re-register it from the location: "C:¥winAMS_CM1¥UnitTest¥AMSTB_SrcFile.c" using the same procedure described in step 14 below.



2. Right click on "Source File" in the project tree, select "Register File in Project" and add the coverage measurement source file "winAMS_Spmc.c" to the project.



3. Run "Rebuilt" from the "Build" menu. This generates an object file with hook code in the folder: "C:¥winAMS_CM1¥mcdc¥target¥obj".

By performing this process, you have built a dedicated build environment for coverage measurement.

Measuring Coverage Using Hook Code

Configure the settings in SSTManager for MC/DC measurement or C0/C1 coverage measurement using the object code for coverage measurement that was compiled in the dedicated build environment for coverage measurement.

Registering Object for Coverage Measurement to SSTManager

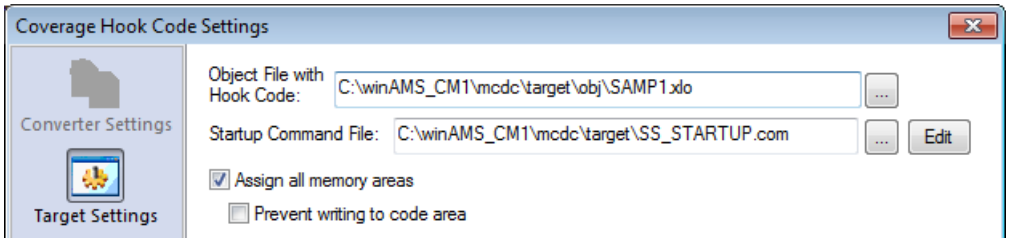
Register the object created for coverage measurement to SSTManager.

1. Click the "Coverage Hook Code Settings" button in SSTManager.
2. For the "Object file with hook code" item, specify the object code for coverage measurement that has been generated in the duplicated hook code folder. *Be careful not to confuse this with the folder in the target code environment.
   The example in this tutorial is:
   C:¥winAMS_CM1¥mcdc¥target¥obj¥SAMP1.xlo

3. For the "Startup command file" item, specify the startup command generated in the duplicated hook code folder. (*The same startup command file can be used to run both the target code and the hook code if the content is the same.)

   C:¥winAMS_CM1¥mcdc¥target¥SS_STARTUP.com

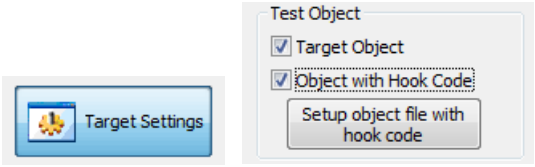
4. Check "Assign all memory areas"
5. Click "OK" to finish.



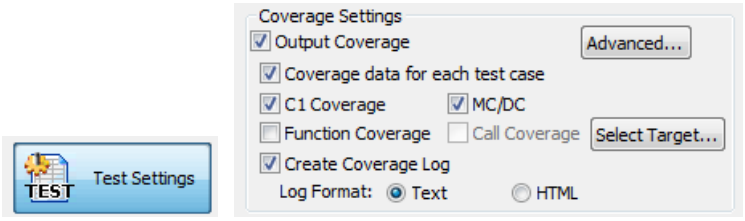
Additional Settings for Coverage Measurement in SSTManager

Enable the following settings for coverage measurement in SSTManager.

1. Check "Object file with hook code" in the "Target settings" view. This runs the hook code using the simulator. (both objects should be checked)



2. Check "MC/DC" in the "Test settings" view. This outputs MC/DC measurement results. Turn this off if measuring only C0 or C1 coverage.

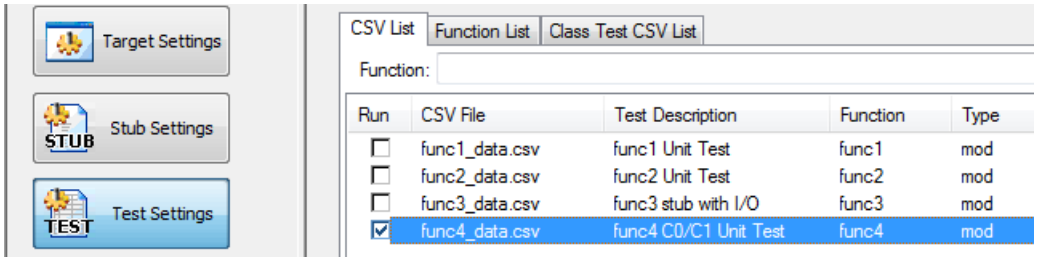


This completes configuration of the settings required for measuring coverage using an object for coverage measurement.

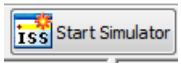
Run a Test for Coverage Measurement Using Hook Code

Run the coverage measurement using hook code and confirm the results. Select the function func4() for the test.

1. Click the "Test settings" button.
2. Check the "Run" box for "func4_data.csv" in the Test CSV List tab.



3. Click the "Start simulator" button.

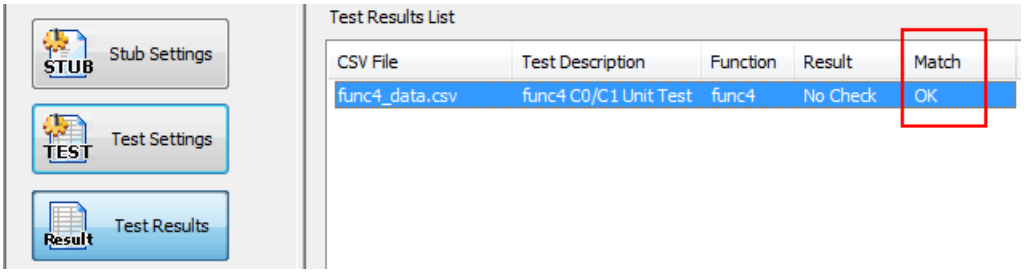


This automatically starts the simulator twice. The simulator runs the target code followed by the code for coverage measurement, and then outputs the results.

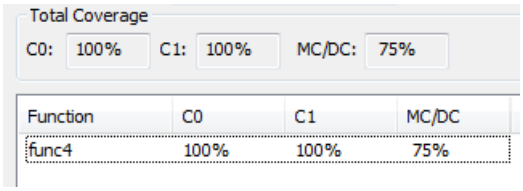
Confirming the Results of Coverage Measurement Using Hook Code

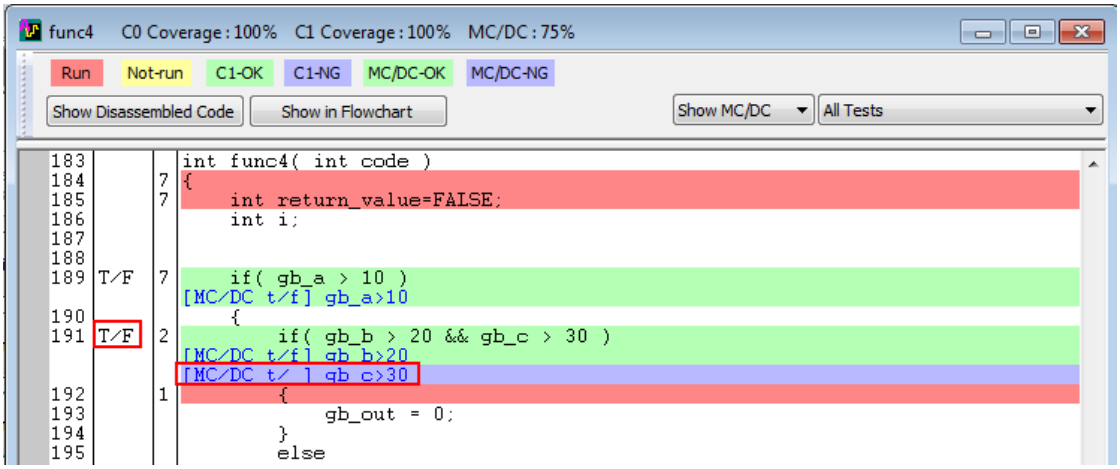
The func4() test results are displayed in the test results view. In the "Match" column, “OK” is displayed to indicate that the output variable values of the target code and the code for coverage measurement match in all of the test cases that were run. This makes it possible to confirm that

the hook code has not influenced actions such as those of the branches in the function itself.



C0, C1 and MC/DC coverage results are output in the Coverage View. (MC/DC is not displayed if no MC/DC measurements have been performed.) In this sample, the coverage rate of the C0 and C1 coverage is 100%, but MC/DC coverage is incomplete with a coverage rate of 75%. Double click the "func4" line to display the coverage view.





MC/DC evaluates whether all conditions have been run as both TRUE and FALSE. In this sample, the C1 coverage of the if statement in line 191 is 100% because the TRUE/FALSE logic of the overall statement has been run and all branches are covered. However, we can see that the FALSE condition of the "gb_c>30" part of the compound condition has not been run.

The MC/DC coverage rate is calculated according to the ratio of conditions for which both TRUE and FALSE have been run.

This concludes the explanation on how to measure coverage using hook code.

Workflow for Measuring Coverage Using Hook Code

Here is a summary of the work procedures necessary for re-running coverage measurements after the environment for coverage measurement using hook code has been completed and the test target source code has been changed.

When the Test Target Source Code Has Been Changed

If the test target source code has been changed, the hook code needs to be updated

accordingly. The object code also needs to be rebuilt for running in the simulator. This is done by the following procedure.

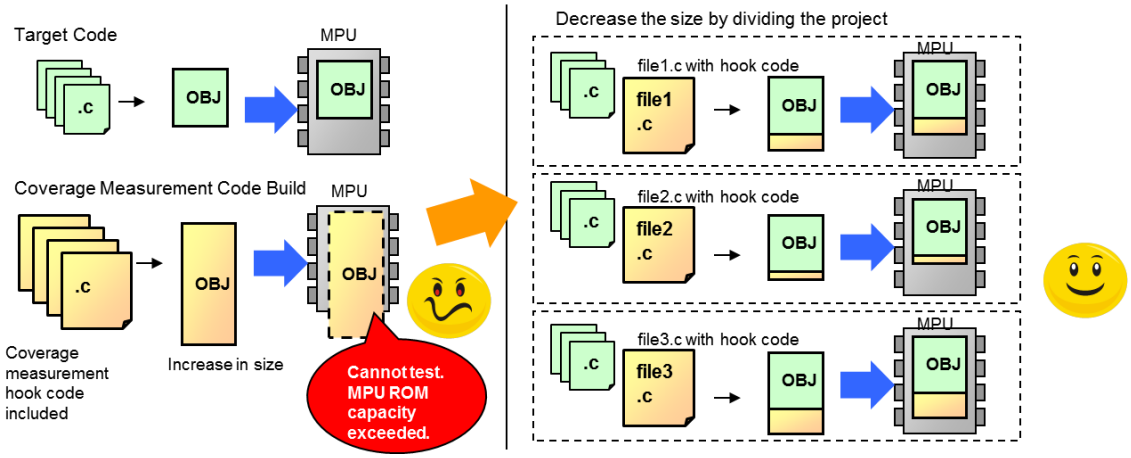
1. Compile the object in the target code build environment (using the development environment being used. Operate IDE.)
2. Run "Re-create all Documents" in CasePlayer2 to re-create the hook code.
3. Compile the generated hook code in the dedicated build environment for coverage measurement (using the development environment being used. Operate IDE.)
4. Click the "Start simulator" button in SSTManager to re-run the test.

This concludes the tutorial on how to measure coverage using hook code.

[Reference] Increase in Code Size by Hook Code

Inserting hook code for MC/DC measurement increases the size of the compiled object. Inserting hook code into all code will increase the object size by several times.

This will exceed the ROM capacity of the microprocessor making it impossible to perform the test.

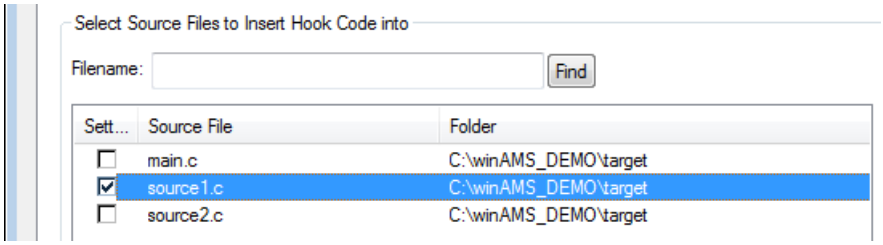


Below are some methods for avoiding this.

1) Limit the application range of the hook code

It is possible to select code for insertion of hook functions in source file units. Inserting hook functions only in source files containing test target functions makes it possible to keep the object code size from becoming too large.

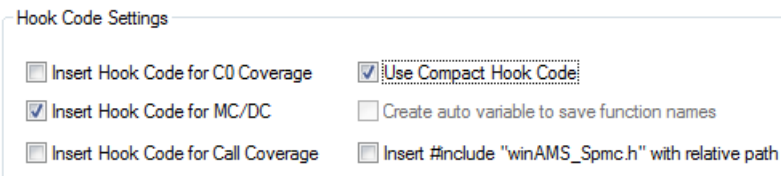
In CasePlayer2, select the [Project Menu] - [Setup object file with hook code] - [Hook Code Settings] tab and select only the sources with test target functions.



2) Use compact hook code

Versions 3.6 and later include a "Use Compact Hook Code" option to decrease the size of the hook code. Turning this option on limits elements of the hook code to decrease the size of the generated object code, although the degree to which the size can be decreased depends on the microprocessor and the compiler.

In CasePlayer2, select the [Project Menu] - [Setup object file with hook code] - [Hook Code Settings] tab and check the "Use Compact Hook Code" option.



3) Compile the hook code object by selecting another MPU type that has the same core and a larger ROM capacity.

If there is a model of microprocessor in the same series as your actual microprocessor but with a larger ROM capacity, the issue can be avoided by specifying the model with the larger ROM capacity for the hook code compilation environment and generating the hook code object on this microprocessor.

Measurements from hook code are run according to the features of the inserted hook function, and the mechanism is not dependent on the object structure of the microprocessor. This means that even if the code is run in a different model from the actual microprocessor, the MC/DC test results are not affected as long as the code is executable.

Select the model with the larger ROM capacity for the compilation environment of the hook code and generate the object here. Leave the compilation environment of the target code (the code that will be used in the product) as it is. This does not affect the results of unit testing other than the coverage measurement, as the values used for evaluation of output variables after running the function are obtained from the target code that runs parallel to the hook code, not the hook code itself.

4) If the ROM capacity is exceeded despite using the above methods, adjust the object

If the object code exceeds the ROM capacity even when hook code is inserted in only one source file, the only way to avoid this issue is to reduce the size of the object in areas that will not affect the test, by methods such as removing objects that do not affect the test from linking.

[Application] Measuring Function Coverage and Call Coverage

Introduction

This tutorial explains the feature for measuring function coverage and call coverage in CoverageMaster. There is no practical tutorial in this chapter.

*Version 3.6 and later of CoverageMaster winAMS/General includes a feature for measuring function coverage and call coverage. A license for the Function/Call Coverage option is required to use this.

Integration Testing in CoverageMaster

The C0, C1 and MC/DC coverage measurements in CoverageMaster are tests performed in the unit test phase. The function coverage and call coverage measurements described in this chapter, meanwhile, are required in the integration test phase.

In particular, structural coverage measurement at an integration level is required under "Part 6-10: Software integration and testing" in ISO26262, a standard for the safety of automobile systems. Function Coverage (1a) and Call Coverage (1b) are standardized in this method.

This feature is provide to enable efficient measurement of function coverage and call coverage in accordance with ISO26262.

Differences between Unit Testing and Integration Testing

This function is an integration test, but the basic operation method in CoverageMaster and the format of the CSV file are the same as in unit testing. For the most part, you do not need to learn a new method for operating CoverageMaster.

However, a proper understanding of the differences between this test and unit testing is required. The main differences are as follows:

- The actual subfunctions are integrated instead of using the stub functions created in unit testing.
- The tests are measured in feature units, and the tests are run with the test cases assigned to the topmost function.
- The execution of the actual subfunctions is measured from the perspectives of function coverage and call coverage.

When designing test cases:

- Integrated tests need to be designed in feature component units, taking into account the execution of the subfunctions.

Integration tests are tests that integrate actual subfunctions instead of using stubs. While the CSV file format and creation procedure are the same as those used in unit tests, it is not possible to use the same test data with stubs that is used in unit testing.

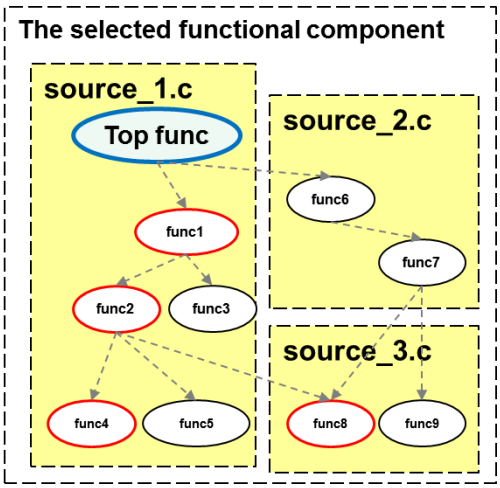
What Is Function Coverage?

Function coverage confirms whether all of the functions included in a feature component are run at least once in an integration test. For example, in the case of a feature component like that shown below, which contains subfunctions (func1-9) in addition to the top function, the test measures whether all of the subfunctions (func1-9) are run when the top function is run.

The arrows in the figure below indicate the integration in the function. However, function coverage only confirms whether the subfunctions were executed, not the function from which they

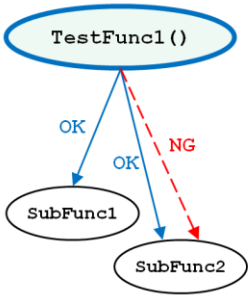
were called. For example, in the case of func1(), func1() is considered to have been run even in cases when the top function calls a function outside of the target feature component and that external function then calls func1().

This is still an indication that the subfunctions in the feature component are covered and called, but function coverage is insufficient for confirming whether the subfunctions are integrated as shown by the arrows.



What Is Call Coverage?

Call coverage confirms the integration indicated by the arrows by measuring whether calling occurs between the functions. For example, if SubFunc1() and SubFunc2() are called from TestFunc1(), this test measures whether all of these calls occurred. In this example, SubFunc2() is called in two places. These calls are measured separately.




```
TestFunc1(){
    if( val == 0)
    {
        SubFunc1(); OK
    }
    else
    {
        SubFunc2(); OK
                :
                :
        if(ctl) SubFunc2(); NG
    }
}
```

Call coverage is measured in function units. Three subfunction calls are made from TestFunc1(). This test measures whether all of these calls are made.

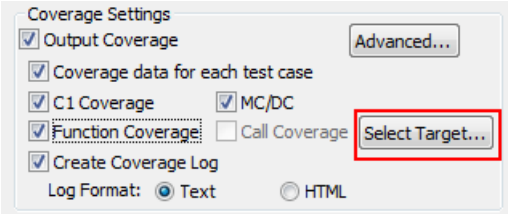
Unlike the function coverage explained above, it is possible to confirm the function integration indicated by the arrows, as call coverage measures whether a specific subfunction is called directly from a specific function. If it is confirmed that a feature component is covered by call coverage, this confirms the actual integration of each of the expected functions in the feature component.

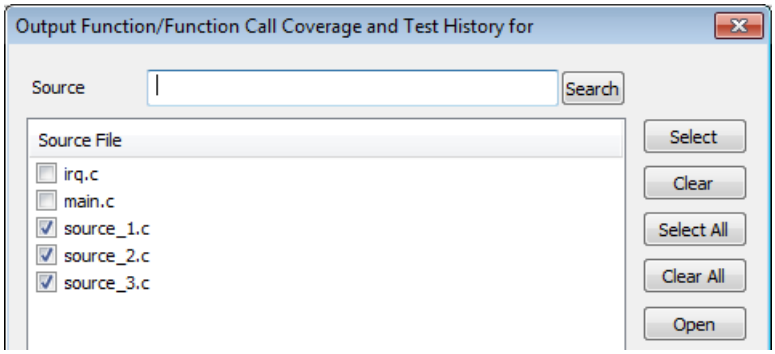
Necessary Settings for Function Coverage and Call Coverage

Selecting Source Files for Measurement

Source files can be selected for function coverage and call coverage measurement in source file units. Functions cannot be selected individually. The selection procedure is as follows.

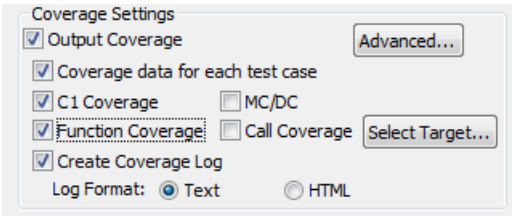
1. Click the "Select Measurement Target..." button in the coverage item of the "Test settings" view.
   (Either the function coverage checkbox or the call coverage checkbox needs to be checked.)
2. Check the checkboxes of the source files to measure.





Configuring Settings for Function Coverage Measurement

Function coverage can be obtained by running the target code. The necessary settings can be configured simply by turning on the "Function Coverage" option in the "Test settings" view. (The hook code does not need to be run if only function coverage is measured.)



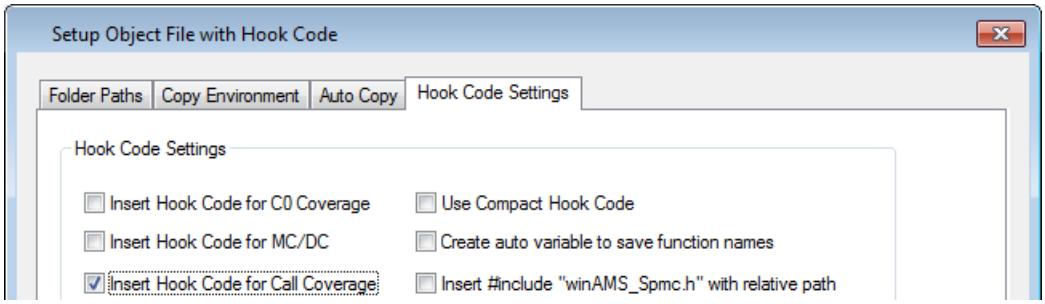
After configuring the above setting, the function coverage can be measured by clicking the "Start simulator" button in SSTManager to execute the test.

Configuring Settings for Call Coverage Measurement

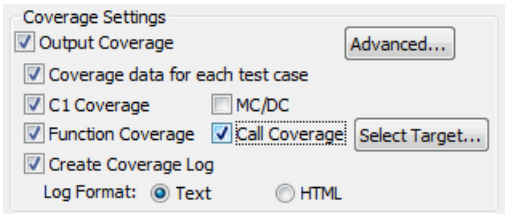
Hook code needs to be applied for call coverage measurement. A dedicated build for coverage measurement needs to be created and run along with the target code by the same method as MC/DC measurement. (See the earlier chapter on building an MC/DC measurement environment for information on how to apply hook code.)

When the MC/DC measurement environment is complete (the hook code is applied), configure the measurement settings by following the procedures below.

1. Select "Setup object file with hook code" from the "Project Menu" of CasePlayer2.
2. Check the "Insert Hook Code for Call Coverage" option in the "Hook Code Settings" tab.



3. Run "Re-create all Documents" from the "Project" menu of CasePlayer2 to create the hook code.
4. Rebuild the hook code to update the object code in the dedicated build for coverage measurement.
5. Check the "Call Coverage" checkbox in the coverage item of the "Test settings" view in SSTManager.



After configuring the above setting, the call coverage can be measured by clicking the "Start simulator" button in SSTManager to execute the test.

Function Coverage and Call Coverage Measurement Results

The function coverage and call coverage measurement results are output as a list in HTML and CSV format.

Confirming the Test Results in the HTML File

The function coverage and call coverage measurement results are output to an HTML file named "Test Report", which is generated immediately after the test is executed. This can be displayed as follows.

1. Click the "Open Reports" button at the top right of the "Test results" view in SSTManager.
2. The "Function/Call Coverage Report" is output toward the bottom of the displayed file, "Test Report.htm".

Function/Call Coverage Rate

Filename	Function Coverage Rate (by file)	Call Coverage Rate (by file)	Function Name	Execution	Call Coverage Rate (by function)	Function Call Count	Function Call Line	Execution
source_1.c	100%	100%	fc_cover_test	yes	100%	2	Line 23 : sub_func1	yes
							Line 27 : sub_func6	yes
			sub_func1	yes	100%	3	Line 35 : sub_func2	yes
							Line 36 : sub_func3	yes
							Line 40 : sub_func3	yes
			sub_func2	yes	100%	3	Line 48 : sub_func4	yes
							Line 52 : sub_func5	yes
							Line 56 : sub_func8	yes
			sub_func3	yes	N/A	0	-	-
			sub_func4	yes	N/A	0	-	-
			sub_func5	yes	N/A	0	-	-

- Filename: The filename of the source file selected as a measurement target.
- Function Coverage Rate (by file): The ratio of functions in the source file that have been run at least once
  - Calculated from the number of items marked as “yes” in the "Execution" column to the right of the “Function Name” column.
- Call Coverage Rate (by file): Execution coverage of the subfunction calls from the functions in the source file.
  - Calculated from the number of items marked as “yes” in the "Execution" column to the right of the “Function Call Line” column.
- Function Name: The name of the function that is a target for measurement.
- Execution: Marked as “yes” if the function indicated by "Function Name" has been run at least once.
- Call Coverage Rate (by function): Execution coverage of the subfunction calls from the function indicated by "Function Name".
- Function Call Count: Total number of subfunction calls for the function indicated by "Function Name".
- Function Call Line: Call function name of the source line number for the subfunction call of the function indicated by "Function Name".
- Execution: Marked as “yes” if the subfunction call of the function indicated by "Function Name" is run.

If all of the "Execution" items at the right of this list are marked as “yes”, this indicates that all of the expected subfunction calls have been run. This confirms that all of the functions in the feature module are integrated as expected.

[NOTE] The "Test Report.htm" file output as an HTML file is a temporary file. The latest test results are saved, but the file is overwritten with the new test results when the test is run again. There is no feature for saving an HTML file for the results of each test. If the results need to be kept, the user must save the HTML file under another name or in a different location.

The HTML file named "Test Report.htm" is generated in the following folder.

[test project folder]¥Out2013-08-03(10'07'39)

*2013-08-03 is the creation date of this project.

*(10'07'39) is the creation time of this project.

*The location for saving can be changed by clicking "Test Results Folder" in the "Test results" view.

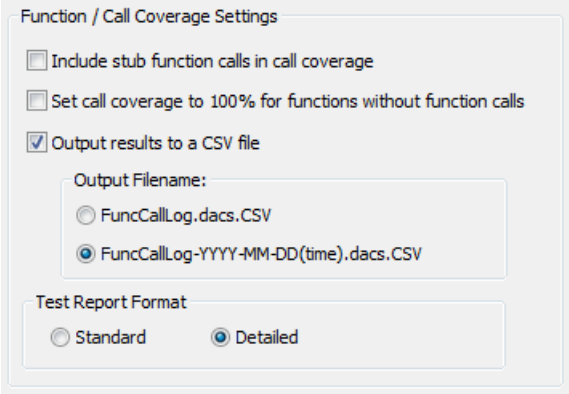
Confirming the Test Results in the CSV File

The "Function/Call Coverage Report" output to the above HTML file is also output to a CSV file at the same time.

It is possible to select whether to save the CSV file as a temporary file that saves only the latest test results, like the HTML file, or whether to add the creation date and time to the filename and save the results of every test.

The method for saving the CSV file can be selected as follows.

1. Click the "Advanced..." button located under the “Coverage Settings” section of the "Test settings" view.
2. Enable "Output results to a CSV file" in the "Function/Call Coverage Settings" section of the "Advanced Coverage Settings" dialog.



The CSV file is saved to the same folder as the HTML file mentioned above.

	A	B	C	D	E	F	G	H	I
1	All Function/Call Coverage Report								
2	Function Coverage Rate	100%							
3	Call Coverage Rate	100%							
4									
5	Function/Call Coverage Report								
6	Filename	Function Coverage Rate(by file)	Call Coverage Rate(by file	Function Name	Execution	Call Coverage Rate(by function)	Function Call Count	Function Call Line	Execution
7	source_1.c	100%	100%	fc_cover_test	yes	100%	2	Line 23 : sub_func1	yes
8								Line 27 : sub_func6	yes
9				sub_func1	yes	100%	3	Line 35 : sub_func2	yes
10								Line 36 : sub_func3	yes
11								Line 40 : sub_func3	yes
12				sub_func2	yes	100%	3	Line 48 : sub_func4	yes
13								Line 52 : sub_func5	yes
14								Line 56 : sub_func8	yes
15				sub_func3	yes	N/A	0	-	-
16				sub_func4	yes	N/A	0	-	-
17				sub_func5	yes	N/A	0	-	-

This concludes the tutorial on function and call coverage measurement.

Copyright Notice and Disclaimer