CS+ Integrated Development Environment

Introductory Guide to Using Python for Automating Debugging in CS+ (Tutorial)

Introduction

This application note is an introductory guide to using the Python functionality incorporated in the CS+ integrated development environment with RH850 microcontrollers. Descriptions in this guide are also applicable to RL78 and RX microcontrollers.

This application note includes the following two sample projects.

RH850_F1L_GoWaitBreakSetRegister

(Name of the folder: AutoDebug_MultiRounds_WriteRegister_Resume)

RH850_F1L_ExcelReadWrite

(Name of the folder: AutoFuzzTest_ExcelData)

These sample projects are based on the “RH850_F1L_Tutorial_Basic_Operation” CS+ sample project.

The operation of the sample projects which come with this application note has been confirmed with V8.06.00 of CS+ for CC. When using CS+ for CC, we recommend using the latest version. The latest version of CS+ for CC can be downloaded from the following URL.

https://www.renesas.com/en/software-tool/cs

This application note and the accompanying software are intended to describe examples of applying the Python functionality, so the content is not guaranteed.

Target Device

RH850/F1L

Documents for Reference

CS+ V8.06.00 Integrated Development Environment User’s Manual: RH850 Debug Tool

CS+ V8.06.00 Integrated Development Environment User’s Manual: Python Console

Contents

1. Getting Startedl

1.1 Outline of This Document

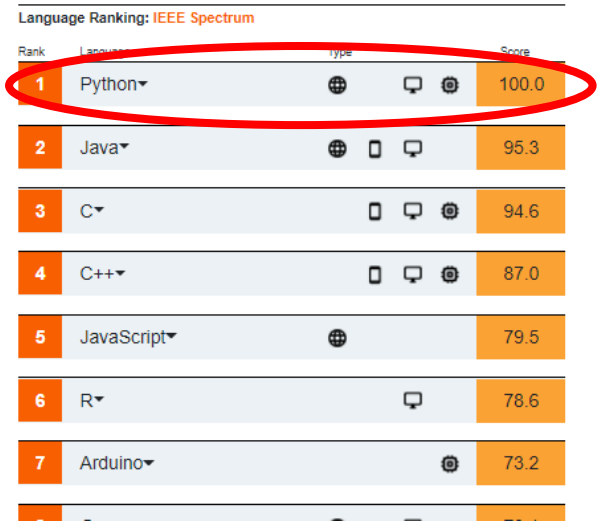
This application note provides tips for helping you in optimizing and automating debugging processes with the use of Python within CS+.

- Brief Introduction to Python
- Role of Python in CS+
- Basic debugging operation with the use of the Python functions
- Debugging operations with the use of two scripts

2. What is Python? A Brief Introduction

2.1 What is Python? A Brief Introduction

Python consistently ranks highly in surveys of the usage of general-purpose programming languages.



The figure on the left shows the top part of the results of a survey by IEEE in 2020.

Python is widely used in system control, tool and application development, scientific computing, and web systems.

The features of Python are as follows.

- Interpreted language (for sequential translation) with a structure where ease of use is given priority over performance in execution
- Lower speed of execution and larger memory usage, compared to compiled languages such as C (where code is subject to bulk translation)

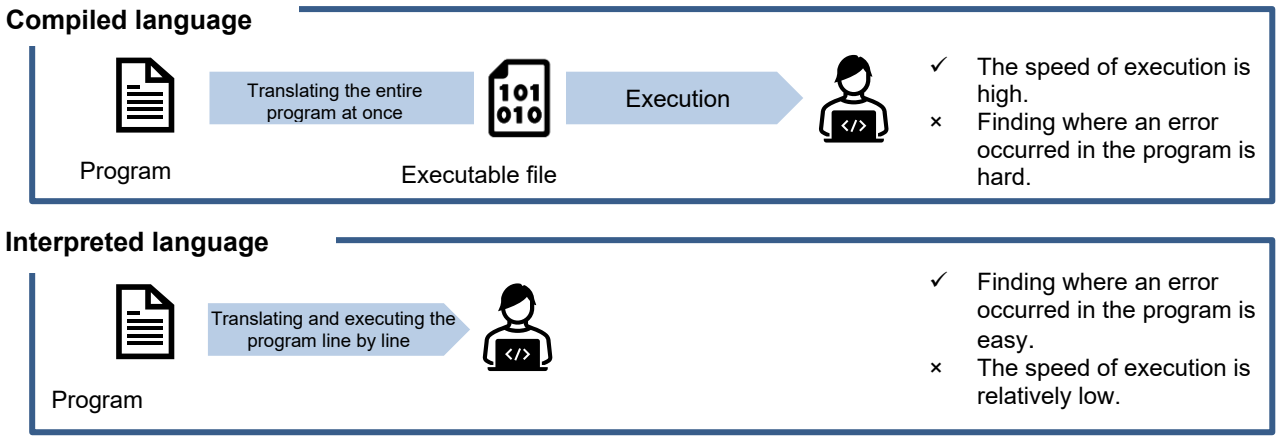
Example of employing Python: Although Python is inferior in terms of performance in execution, it is employed in Instagram.

⇒ The statement at left gives the reason for this.

[@Pycon2017 instagram keynote]

2.2 Difference between Compiled and Interpreted Languages

The following is a brief description to confirm the difference between compilers and interpreters. To actually execute a programming language on a computer, it must be translated (compiled) into machine code. Languages that have a system for translating (compiling) the entire program into machine code as a batch are referred to as compiled languages. On the other hand, languages that have a system for translating (compiling) the program line by line sequentially, like a human-language interpreter, are referred to as interpreted languages.



As described above, compilers offer the advantage of high-speed execution of programs by directly executing the executable file created through bulk translation in the computer. They have, however, disadvantages such as finding where an error occurred in the program being difficult and rebuilding being required each time the program is modified. This is likely to result in relatively long times for debugging, with longer times for development as a result.

On the other hand, using an interpreter slows the actual execution of a program because of the sequential translation, but debugging becomes relatively easier. For this reason, the times for development are likely to be shortened.

2.3 Role of Python in CS+

The role of Python in CS+ is, for example, equivalent to that of VBA for writing Excel macros.

In other words, it can be used to automate routine tasks and speed up the work itself.

The features of Python in CS+ are listed below. (See Appendices for details.)

1) The variant of Python incorporated is IronPython (Python running in the .NET Framework and compatible with Python 2.7)

- Includes control statements and so on.
- Works with external applications such as Excel (importing .NET assemblies into Python is possible).
- Restrictions apply to certain functions such as the Python debugger.

2) Functions that implement functionality that is specific to the CS+ IDE
The CS+ functionality is specifically implemented as functions for use with CS+.

- For project: project.Close, project.Open, project.GetFunctionList, etc.
- For build tool: build.All, build.ChangeBuildMode, build.Clean, etc.
- For debug tool: debugger.ActionEvent.Delete, debugger.ActionEvent.Set, etc.
- Others (for basic operation and common): ClearConsole, Save, common.GetOutputPanel, etc.

Total: 160

3. Python Console

The Python console is analogous to the Windows command prompt, and is used to enter and execute Python functions, control statements, and the CS+ Python functions. It also displays the results of executing the functions and errors.

The Python console is displayed in the Python Console panel.

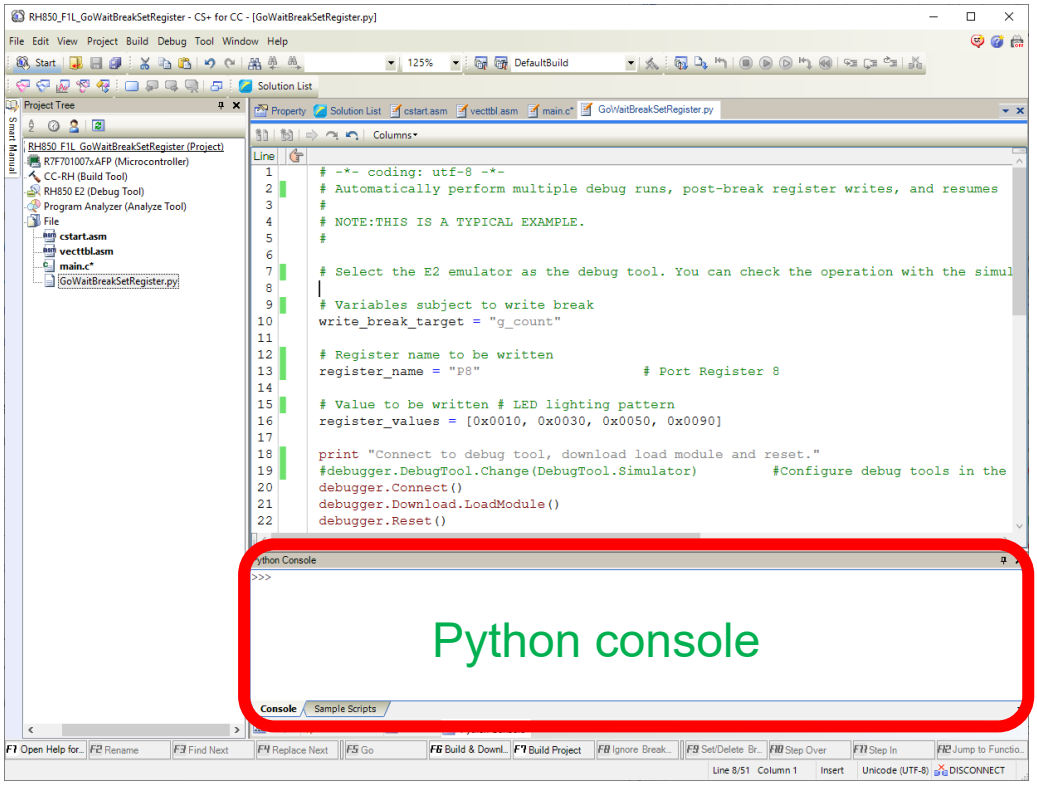
3.1 Preparing the Python Console Panel

The display of the Python Console panel consists of the console itself and sample scripts. The former is used to execute Python commands and display the results of their execution and the latter is used to display sample scripts and register their files with projects.

Display the Python Console panel.

Steps:

1) Opening a project
2) Displaying the Python Console panel

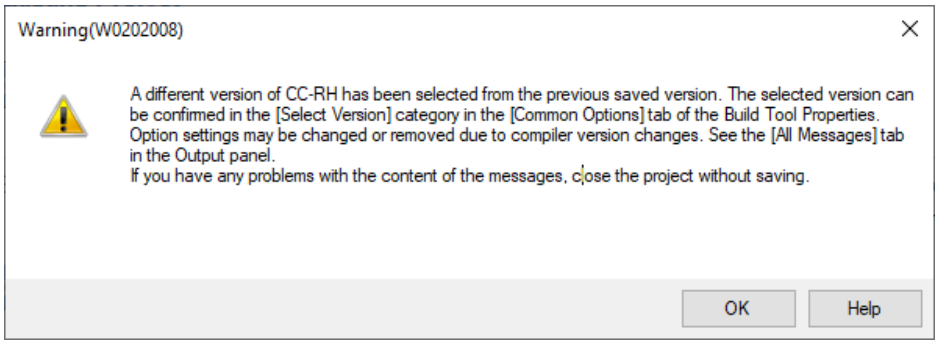


Step i. Opening a project

Use the RH850_F1L_GoWaitBreakSetRegister project.

Place the “AutoDebug_MultiRounds_WriteRegister_Resume” folder, which comes with this application note, on your desktop, and double-click on RH850_F1L_GoWaitBreakSetRegister.mtpj in the RH850_F1L_GoWaitBreakSetRegister folder under the accompanying folder to open the project.

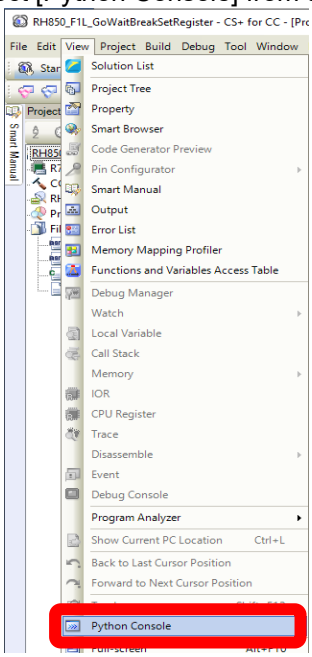
Note: The operation of the sample projects which come with this application note has been confirmed with V8.06.00 of CS+ for CC. If you are using a later version of CS+ for CC than V8.06.00 and the bundled CC-RH C compiler has been upgraded, the following warning will appear. The situation does not present a problem, so click on the [OK] button.



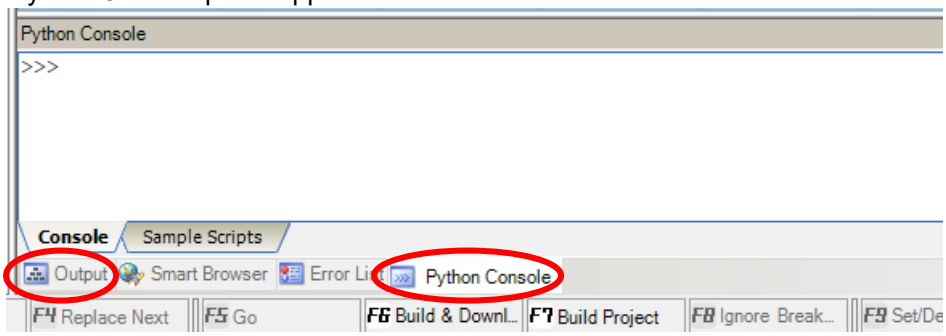
Step ii. Displaying the Python Console panel

Once the project has been opened, display the Python Console panel.

1) Select [Python Console] from the [View] menu.



2) The Python Console panel appears.



The Python Console panel opens at the same position as the Output panel, so that they fully overlap each other at that time.

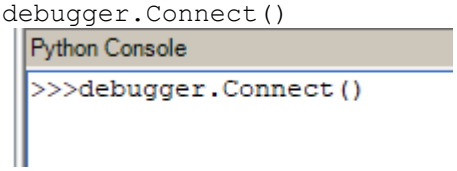
3.2 Basic Operations for Debugging with the Use of Python Commands

This section describes how to proceed with the following basic operations for debugging by directly using functions (commands) from the Python console.

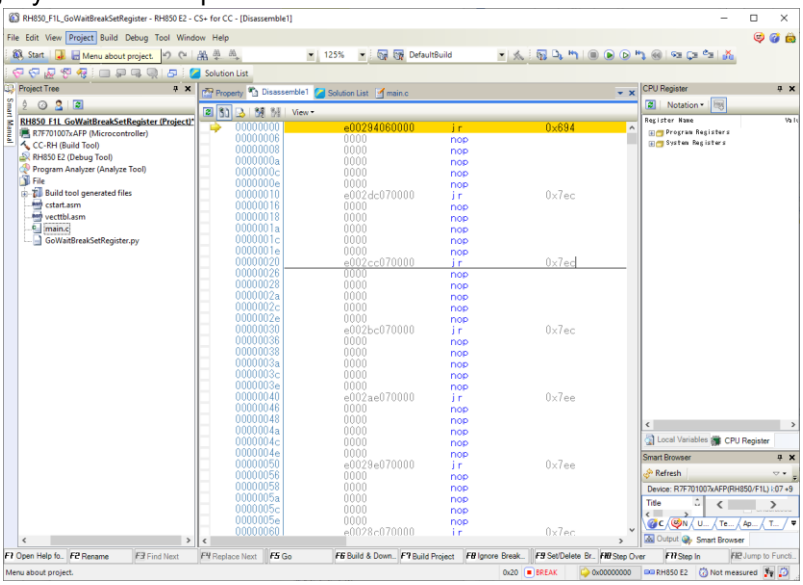
Connecting to the debugger ⇒ Downloading the program ⇒ Setting a breakpoint ⇒ Executing the program ⇒ Disconnecting from the debugger

1) Connecting to the debugger
Enter the following function (command) in the Python Console panel and press the Enter key.

```
debugger.Connect()
```



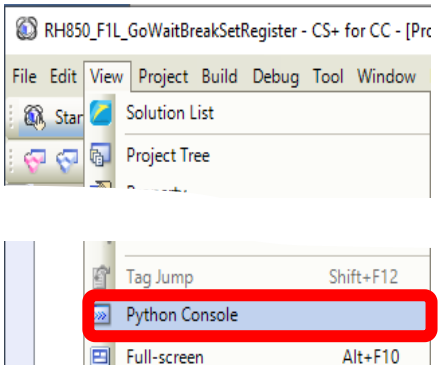
The debug layout window opens.



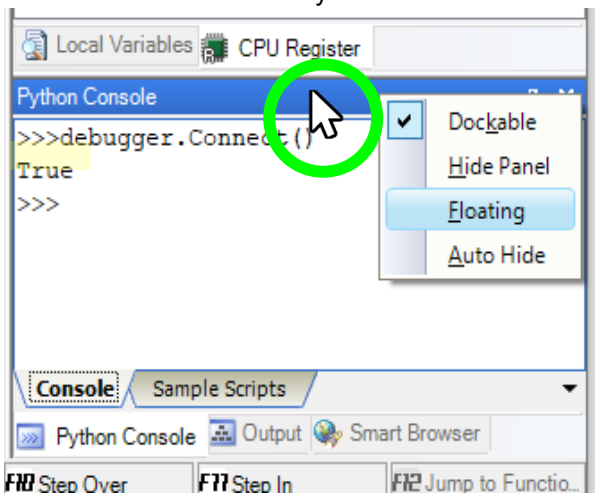
Since a program has not been downloaded yet, confirm that no code is displayed on the [Disasemble1] tabbed page.
The Python Console panel has not been displayed. As the layout is exclusively for use during debugging, setting another layout is required. Display the Python console again.

▪ Displaying the Python console

Before downloading the program, set the layout for use in debugging.
Open the Python console.
<1> Select [Python Console] from the [View] menu.

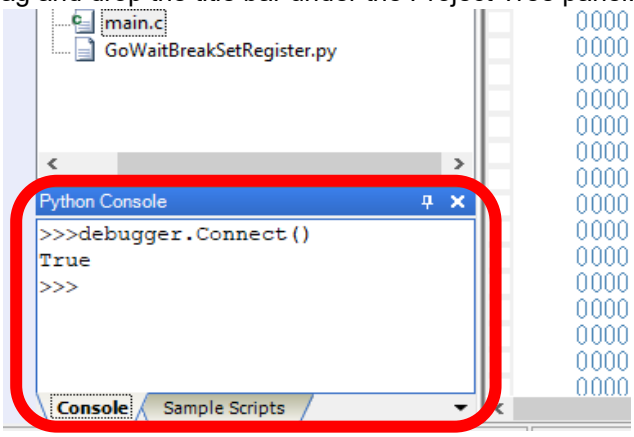


<2> Once the Python console has opened, you can confirm that “True” is set for the debugger.Connect() function that was previously entered, which indicates that it was executed successfully.
Right-click on the title bar of the Python console and select [Floating] from the pop-up menu.



* During debugging, if the Python Console and Output panels overlap each other in the display, the Output panel is given priority for display. If you want to continuously monitor the Python console, move it. To do this, temporarily place it in the floating state. Note that leaving the Python console floating may lead to its being hidden by CS+ while a script is running, and as a result you will be prevented from viewing outputs during execution. Therefore, move the Python console and then fix it in place so that it is not overlapped by the Output panel.

<3> Drag and drop the title bar under the Project Tree panel.

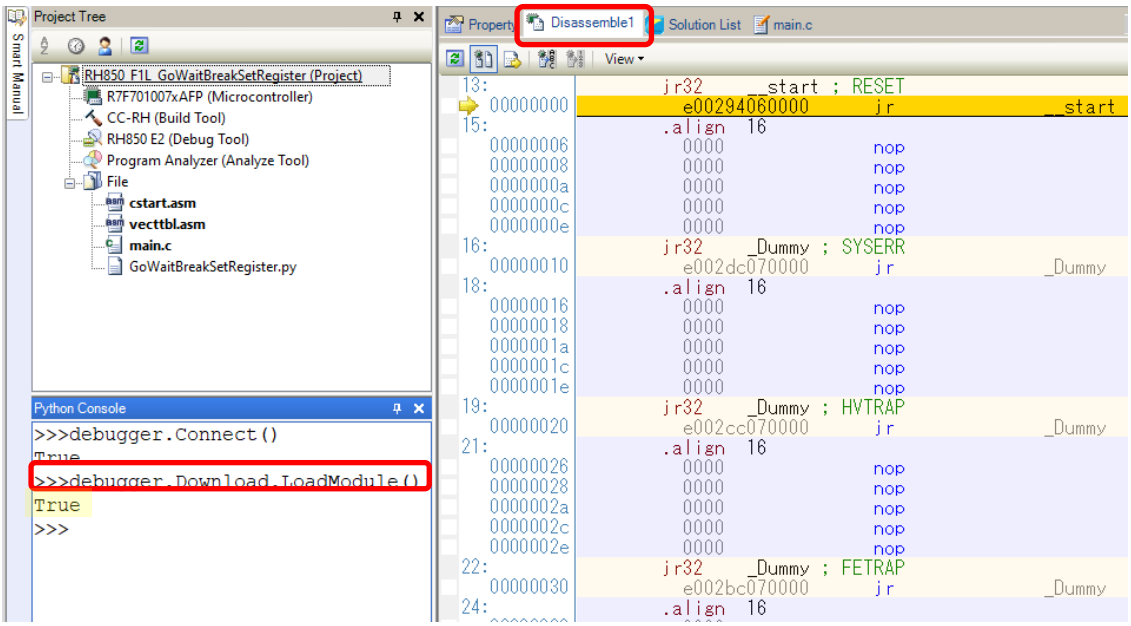


The location to which the panel is moved is as desired. Once you have displayed the Python console, the layout will be saved when you save the project.

2) Downloading the program
Enter the following function in the Python Console panel.

```
debugger.Download.LoadModule()
```

The program will be downloaded. After downloading is completed, you can confirm that “True” is returned in the Python Console panel and that code has been loaded by checking the [Disasemble1] tabbed page.



3) Setting a breakpoint

As any breakpoints set during previous evaluation may remain in the code, use the following function to delete the breakpoints just to make sure.

```
debugger.Breakpoint.Delete()
```

Set a breakpoint at the address corresponding to line 46 of main.c, in the case of this example, 0x77c.

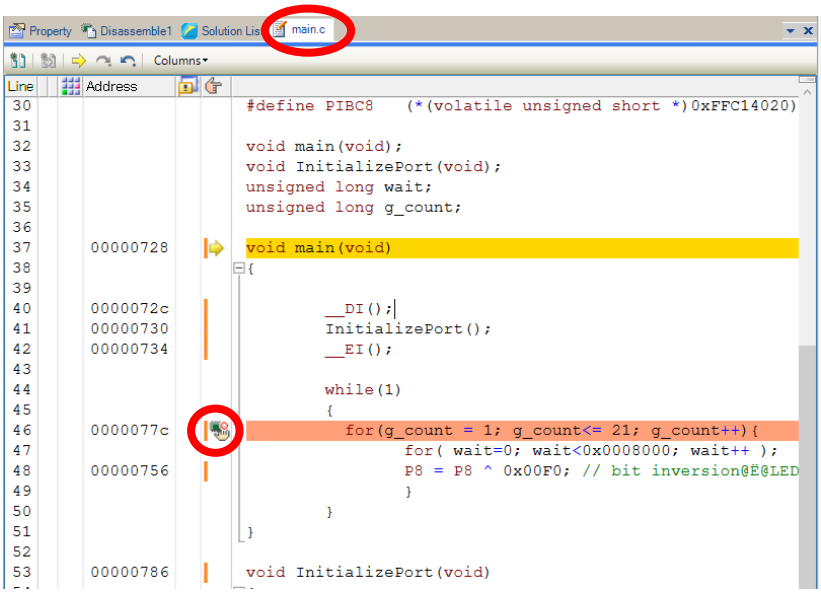
Note: The address will differ with the compiler version, setting for the optimization option, and other factors.

Enter the following statements to set a breakpoint.

```
bp = BreakCondition()
bp.Address = "0x77c"
debugger.Breakpoint.Set(bp)
```

(These statements are separately described in Explanation of the Script.)

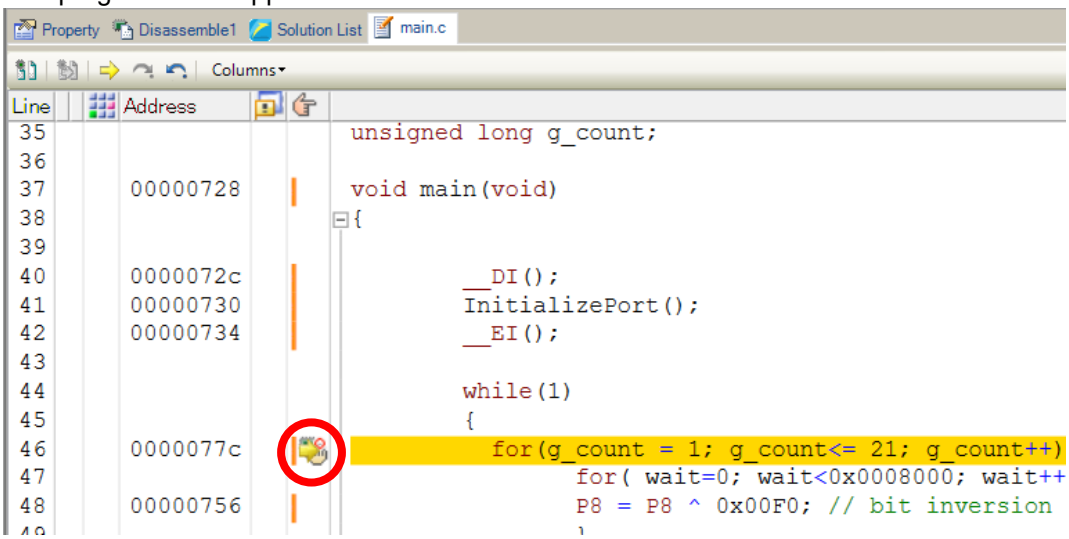
You can confirm that an event symbol (enclosed in the red circle in the figure below) is displayed and that a breakpoint has been set at address 0x77c on the [main.c] tabbed page.



4) Executing the program
Enter the following command to run the program up to the breakpoint. GoOption.WaitBreak in parentheses is an argument which indicates that execution is to proceed up to the next breakpoint.

```
debugger.Go(GoOption.WaitBreak)
```

A right-facing yellow arrow is displayed next to the value of the program counter (PC) at the breakpoint to indicate that the program has stopped there.



5) Disconnecting from the debugger
Finally, enter the following command to disconnect from the debugger and finish debugging.

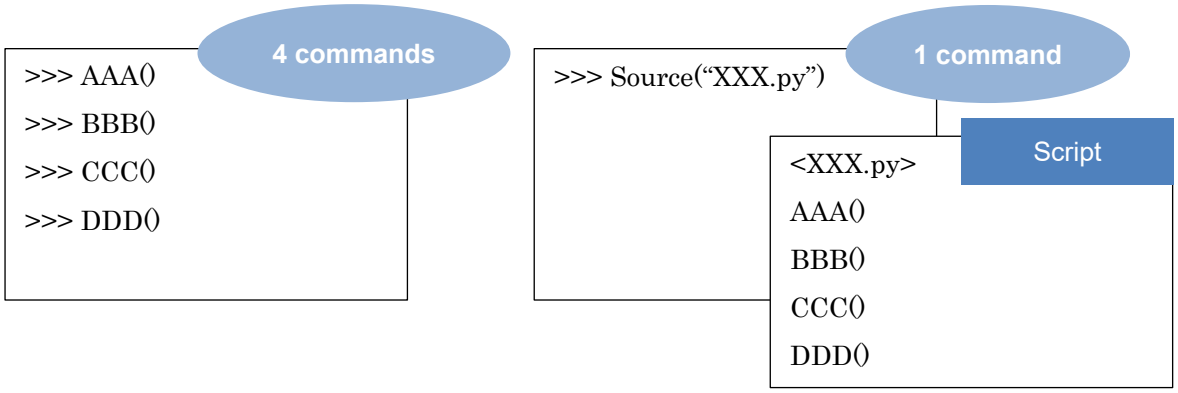
```
debugger.Disconnect()
```

3.3 Preparing Scripts to Enable Efficient Operations

Entering individual functions or commands each time you want to execute an operation is time-consuming. Furthermore, since the control statements cannot be used with this approach, advanced control is not possible. Therefore, we will describe operations to be executed in a file so that it can run as a program.

Describing operations such as functions and control statements in a script file in advance enables the use of only a single command input to achieve multiple operations. This file is referred to as a Python script. Executing such a file enables multiple operations in response to a single command input.

Schematic view



4. Automatic Debugging with the Use of Python Scripts

First of all, a brief description of the scripts of the two projects covered in this application note is given below.

The script of project 1 (referred to as script 1):

Script 1 is used to automatically proceed with multiple rounds of debugging, writing to a register after a break, and resuming.

⇒ This script facilitates unit testing by allowing the modification of program variables as desired.

The scripts of project 2 (collectively referred to as script 2):

Script 2 actually consists of three scripts for use in ”AutoFuzzTest_ExcelData” folder.

⇒ These scripts facilitate fuzzing test, which allows the data processing for rewriting a program variable with values from an Excel file for use in execution and output of the results of calculation to the given Excel file.
Fuzzing test is a form of testing for confirming the behavior of programs in response to the input of random values.

4.1 Script 1

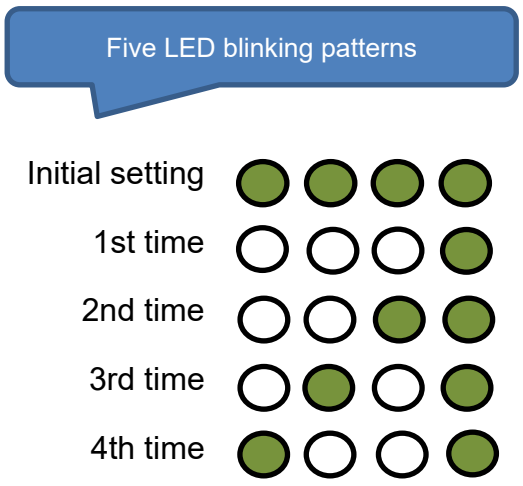
An overview of script 1 is given below.

Script 1 is used to automatically proceed with multiple rounds of debugging, writing to a register after a break, and resuming.

This script is intended for confirming the operation of LED lighting patterns under multiple conditions.

The script proceeds through the following operations.

1) Connection to a debug tool (E2 emulator or simulator).
2) Setting a breakpoint and running the program.
3) Writing an LED blinking pattern to be tested to a register following a break.
4) Restarting.
5) Repetition of steps 3 and 4 a specified number of times (five times including the initial setting).
6) Disconnection from the debug tool



The lighting patterns are fixed in the program. Those shown are with the initial settings.Python is used to rewrite the register to change the pattern four times and check whether lighting of the LEDs is in accord with the changes in the pattern.

The script starts with processing to connect to the debug tool and set a breakpoint.

It uses Python to rewrite the register in response to a break and then restart processing.

After that, it repeats the rewriting in response to a break four times, and then disconnects from the debug tool.

* Equipment to be used: E2 emulator or simulator
EB-850/F1L-176-S evaluationboard (MCU board)
from TESSERA TECHNOLOGY INC.

1) Preparing the First Project

The following is preparation in advance of using the first script.

Steps: As steps i and iii are a duplication of steps described in section 3.1, the related figures are omitted here.

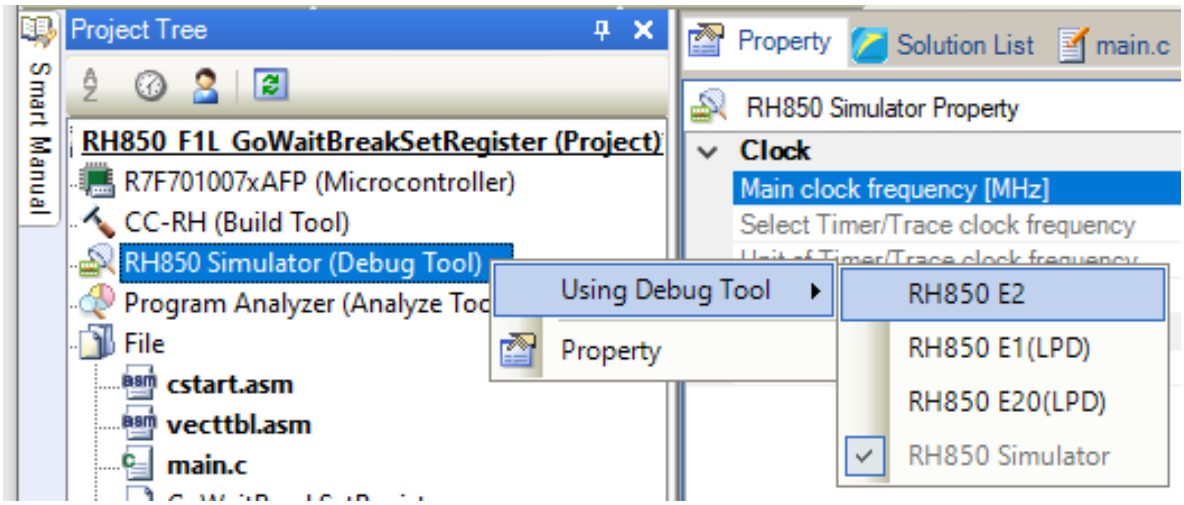
<1> Opening the project
<2> Setting up the debug tool
<3> Displaying the Python console and adjusting its position
<4> Preparing for debugging with the use of the script (setting the debug layout)

Step i Opening the project

Open the project, which comes with this application note.
Use the RH850_F1L_GoWaitBreakSetRegister project. This is packaged in the "AutoDebug_MultiRounds_WriteRegister_Resume" folder.
Extract the given folder in the location you desire. In this description, it is extracted on your desktop as an example.
Double-click on RH850_F1L_GoWaitBreakSetRegister.mtpj in the RH850_F1L_GoWaitBreakSetRegister folder under the extracted "AutoDebug_MultiRounds_WriteRegister_Resume" folder to open the project.

Step ii Setting up the debug tool

Set the E2 emulator or simulator as the debug tool.



If you will be using the E2 emulator, connect it to the evaluation board. In addition, check the conditions on the connection, such as the main clock setting and power supply, in the properties.
If you will be using the simulator, you cannot check the behavior of the LEDs. However, we provide a separate project for using Excel to monitor the behavior of the LEDs (see the relevant appendix).

Step iii Displaying the Python console and adjusting its position

<1> Select [Python Console] from the [View] menu.
<2> The Python Console panel appears.

Step iv Preparing for debugging with the use of the script (setting the debug layout)

This setting is for checking display by the Python console in real time during execution of the Python script. As the layout is exclusively for use during debugging, the setting should be made in advance of executing the script.

At the same time, make preparations for checking the variables and register values for use in debugging the script (watch expression settings)

<1> Connect to the debug tool.

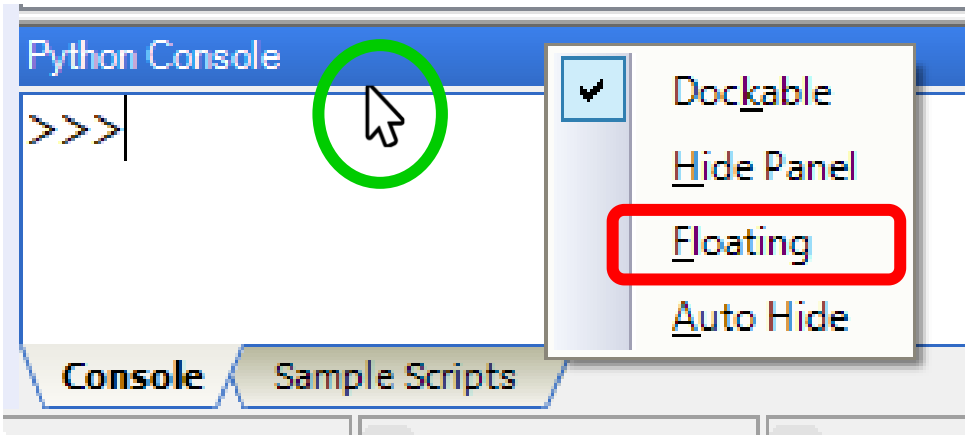
Pressing the  button (used to initiate downloading of the program to the debug tool) in the upper-right corner of the CS+ window makes a connection to the debug tool and downloads the program.

<2> Display the Python console and move it to the location you desire.

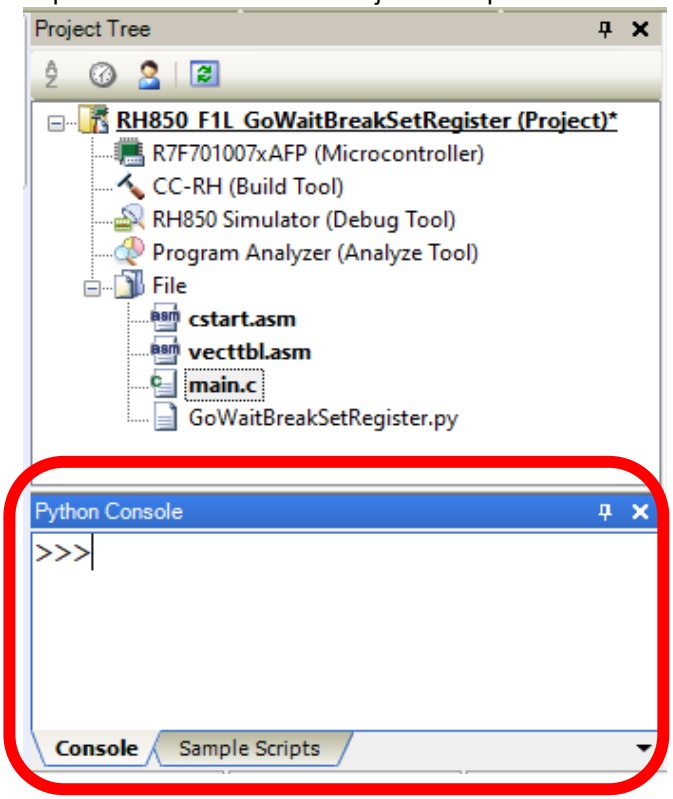
This is for checking the Python output during execution of the script. This step is partially a repetition of the descriptions in section 3.2.)

i. Select [Python Console] from the [View] menu.

ii. **Right-click** on the title bar of the Python console and select [**Floating**] from the pop-up menu.



iii. Drag and drop the title bar under the Project Tree panel.

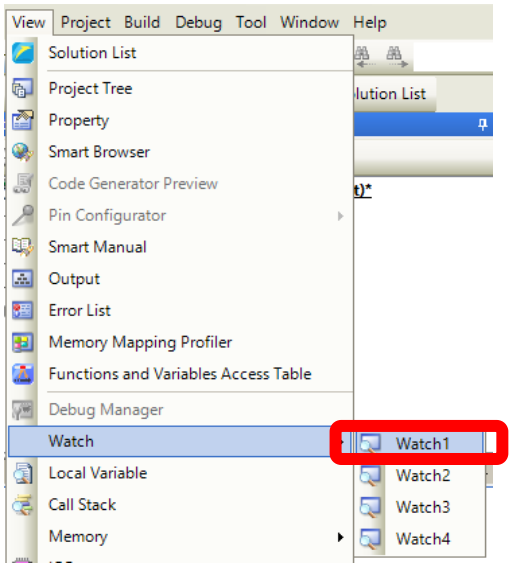


<3> To use the GUI to monitor the values of variables during execution, display the Watch panel and register watch expressions.

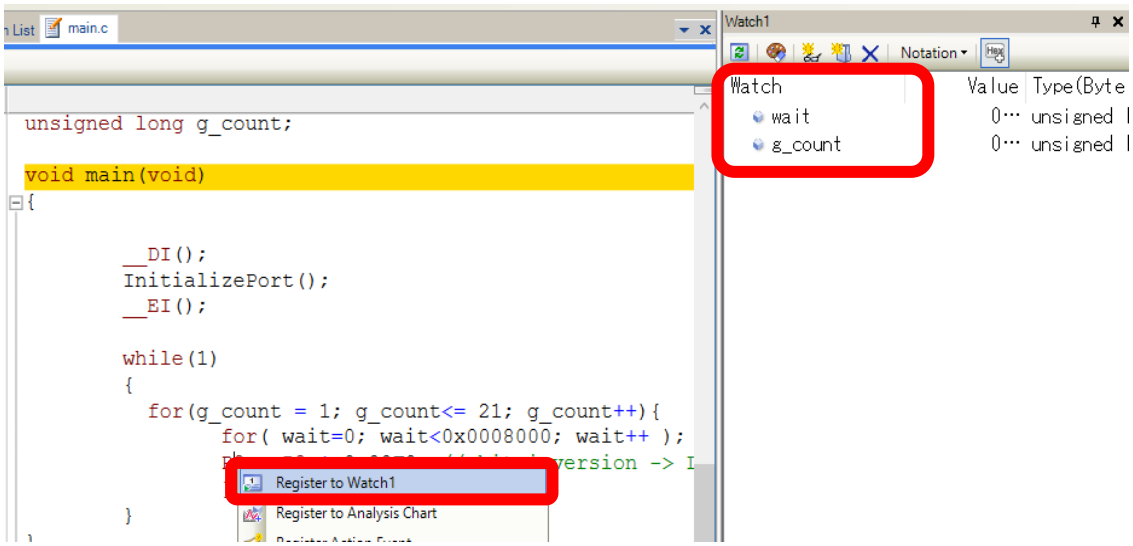
The variables to be monitored here are wait, g_count, and P8.

i. Select [Watch] ⇒ [Watch1] from the [View] menu.

ii. Register the watch expressions.



In sequence, right-click on each of the wait, g_count, and P8 variables in the main.c entry in the Editor panel. Select [Register to Watch1] at the top of the context menu that appears.
These variables are now registered with the Watch panel.



<4> Disconnect from the debug tool.

After having registered the variables, press the  button (disconnect button) in the upper-right corner of the CS+ window to disconnect from the debug tool.
The preparations are completed at this point.

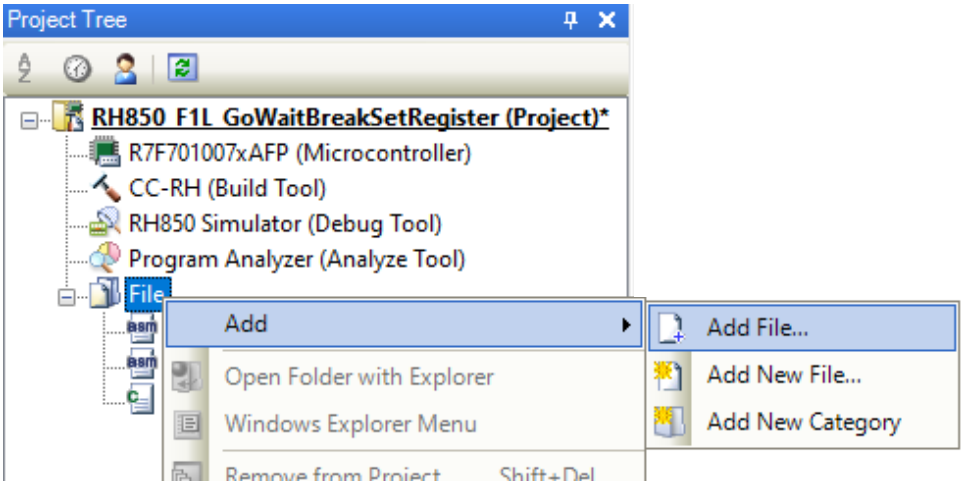
2) Executing Script 1

Before explaining the script, we will check its operation.

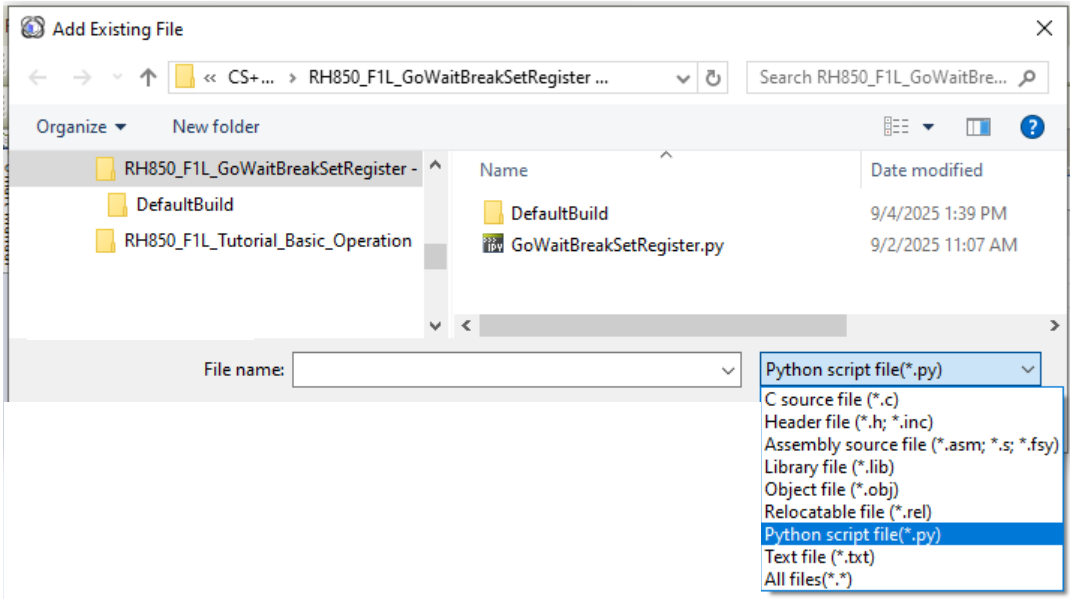
Step i Registering the script

When the project is already in use, the script will have been registered. This step is for the case where it has not been registered.

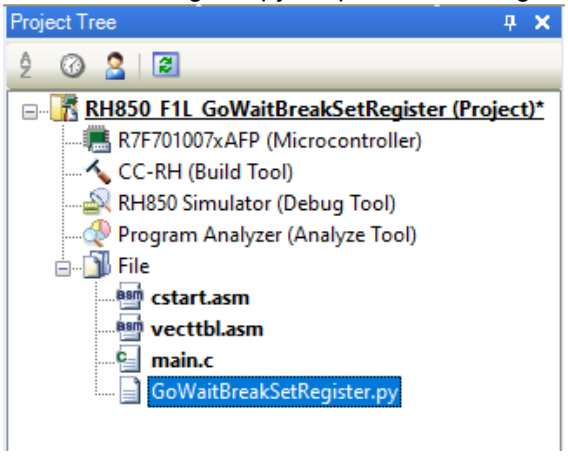
<1> Right-click on [File] in the Project Tree panel and then select [Add] ⇒ [Add Existing File] from the context menu.



<2> The [Add Existing File] dialog box opens. Select [Python script file(*.py)] as the file type. Select [GoWaitBreakSetRegister.py].

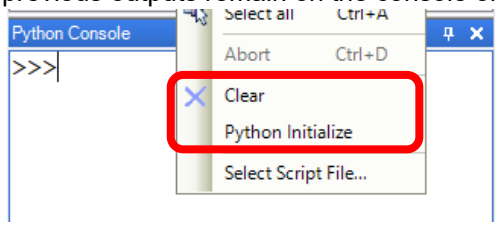


<3> The GoWaitBreakSetRegister.py script file is now registered.



<4> Before executing the script, initializing the Python console and clearing of the screen are required. Right-click on the Python Console panel.

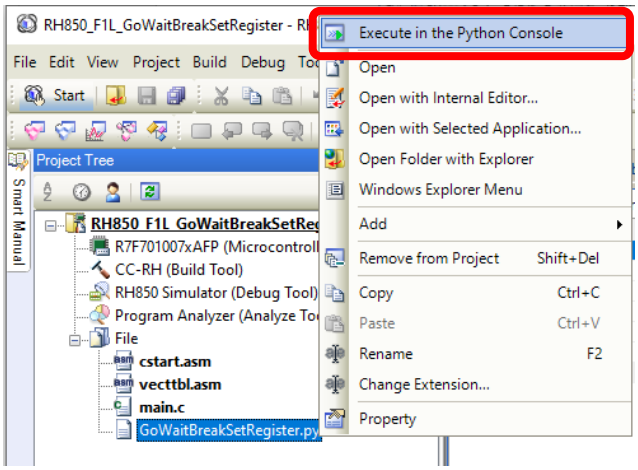
This step is for the second and subsequent rounds of execution of the script. It clears the memory when the previous outputs remain on the console or errors occurred in the first round, etc.



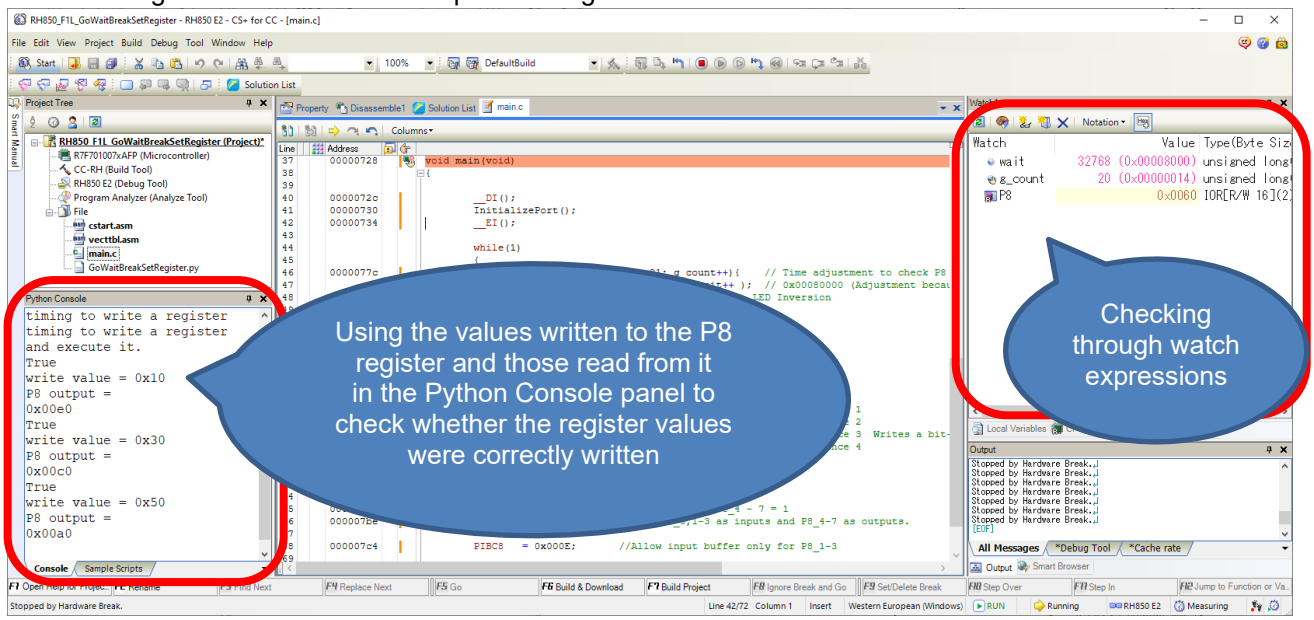
<5> Execute GoWaitBreakSetRegister.py.

Right-click on [GoWaitBreakSetRegister.py] in the Project Tree panel and select [Execute in the Python Console] at the top of the context menu.

You can confirm that the script is running and the lighting patterns for the LEDs on the board change in sequence.



The figure below is a screen capture during execution.



The script makes the Python Console panel display the values written to the P8 register and those read from it to check whether the register values were correctly written.

The Watch panel displays the values of each of the registered variables, wait, g_count, and P8 in real time.

3) Explanation of the Program and Script
This section describes the program, and then the script.

Explanation of the Program

Overview: Blinking LEDs connected to port 8

main.c

```
// Port register 8
#define P8 (*(volatile unsigned short *)0xFFC10020)


  main(){
  while(1)
  {
    for(g_count= 1; g_count<21; g_count++){


// The above loop repeats the following lines 20 times
    for( wait=0; wait<0x00020000; wait++ );


    P8 = P8 ^ 0x00F0;

    }
  }
```

The LEDs are connected to bits 4 to 7 of port 8.

- Register definition:
- P8: Output from port register 8
  Write an LED lighting pattern.
  The script changes the value on a break.
- for(g_count=1, ...): To make the LEDs blink 10 times with each LED lighting pattern, set a break at g_count = 21.
- for(wait=0; ...): Serves as a timer in control of blinking
- Invert the values of bits 4 to 7 in the P8 register.
  Bit inversion = inversion of the LED states.

This program makes the LEDs connected to bits 4 to 7 of port 8 blink. Write the LED lighting pattern to the P8 register, that is, port register 8, which sets the output of the corresponding port as a port-related register. The script rewrites the value of this register to change the lighting pattern following a break. The program makes all of the LEDs light up when the setting is 0xF0.
The program uses a for loop with counting by the wait variable as a timer to control blinking of the LEDs through inversion of the values of bits 4 to 7 in the P8 register. As a result, the P8 output is inverted and the LEDs that are on are switched off and those that are off are switched on (that is, they are made to blink).
The for loop controlled by g_count sets the number of times the LEDs are to be switched between on and off or off and on for each LED lighting pattern. This program makes them blink 10 times. To handle this, the script sets a breakpoint so that a break occurs when counting up by g_count reaches 21. Note that in this case, the for loop sets g_count to start counting up from 1. You can confirm that starting from 0 causes the values written to be read and output to the Python Console panel without change instead of being inverted.
The script rewrites the value of the P8 register to change the lighting pattern following a break.

Explanation of the Script

The script file is GoWaitBreakSetRegistexr.py.
This script handles the processing for connecting to the debugger ⇒ setting the breakpoint ⇒ repeating the processing five times ⇒ disconnecting from the debugger.
The detailed descriptions are divided into three items: 1. Control over the execution-related processing; 2. Setting the breakpoint; and 3. Repeated processing.

Note that the related code is consolidated for ease of description, so its flow differs from that of the actual code.

1) Control over the execution-related processing

```
debugger.Connect()
debugger.Download.LoadModule()
debugger.Reset()
debugger.Disconnect()
```

- Connects to the debug tool.
- Downloads the program.
- Resets the CPU.
- Disconnects from the debug tool.

2) Setting the breakpoint

```
bp =BreakCondition()
bp.Address="g_count"

bp.BreakType=BreakType.Write
bp.Data=21

debugger.Breakpoint.Set(bp)
debugger.Go(GoOption.WaitBreak)
```

- Creates an instance.
- Specifies the address for setting the break (in this case, that of g_count).
- Specifies a data write break as the break type.
- Specifies the number to set as the break condition for the data.
- Configures the breakpoint.
- Makes the script wait until the program stops.

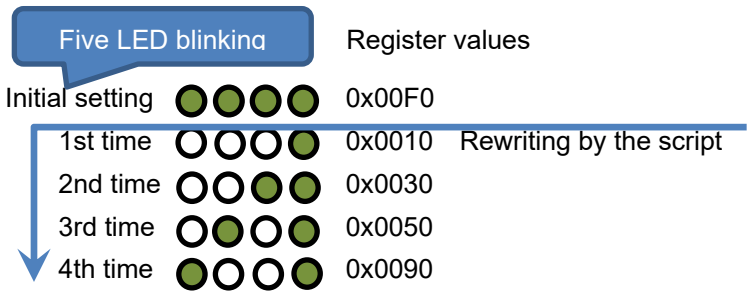
BreakCondition is a class that creates a set of break conditions. In this case, the instance is the bp object.
The BreakCondition class has four member variables: Address, BreakType, Data, and AccessSize. In this case, the three other than AccessSize are set.
As described in the section titled “Explanation of the Program”, rewriting the LED lighting pattern requires setting a breakpoint so that a break occurs after the LEDs have been made to blink 10 times. To do this, set the data write break so that a break occurs when 21 is written to the g_count variable. Set the address of the g_count variable as the point at which to generate the break.
The function debugger.Breakpoint.Set(bp) is used to set the breakpoint to reflect the conditions stated above.

3) Repeated processing (writing to the register, restarting, and output to the monitor)

The following describes the repeated processing.
In the repeated processing, writing to the register, restarting execution, and output of the register value to the monitor repeatedly proceed each time a break occurs.
Set the two variables to be used in advance. Specifically, set P8 in register_name as the destination for writing and the four LED lighting patterns to follow the initial one in the register_values array.

```
# Settings:
# Name of the destination register for writing
  register_name="P8"
# Values to be written (for changing the LED lighting pattern)
register_values = [0x0010, 0x0030, 0x0050, 0x0090]
```

- P8: Port register 8 used as an output port

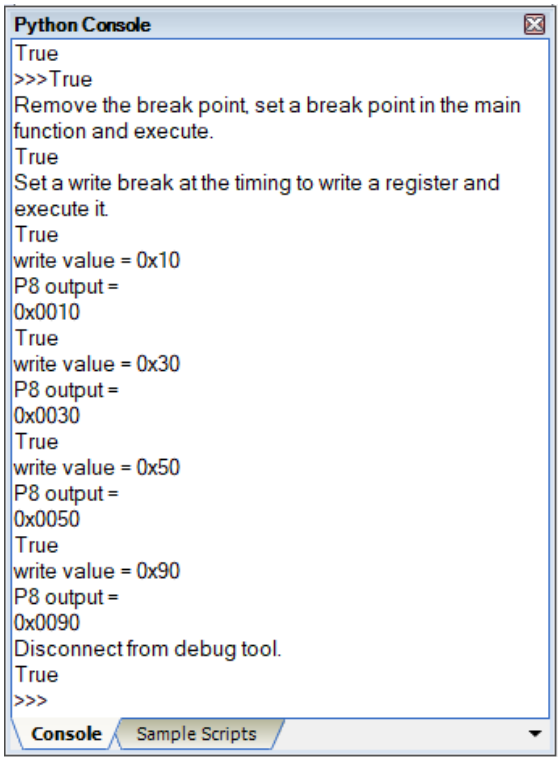


As shown in the figure to the left, the initial setting of the program, 0x00F0 is set in the program as the initial value. The other four patterns are subsequently rewritten each time a break occurs.

The following processing is repeated.

```
# Starting from the beginning of the register_values,
  values are read from the array as the variable values.
for value in register_values:
  debugger.Register.SetValue(register_name, vlue)      # Writes a value to the register.
                                                       # Sequentially writes a value from among
                                                         those in the register_values array to P8.
  debugger.Go(GoOption.WaitBreak)                      # Restarts execution (waits for the next
                                                         break).
  # Output the result of monitoring to the Python console.
  output ="write value = 0x%x"%value                   # The current value to be written
  print(output)
  # Confirm by reading P8 that writing proceeded.
  print "P8 output = "
  debugger.Register.GetValue(register_name)            # Directly specifying the name of the register
                                                         for reference is also possible.
```

In the processing for rewriting, the following steps are repeated.
The for loop is used to capture values, one by one, from the register_values array as the variable values.
Each captured value is set by the debugger.Register.SetValue function, the register is rewritten, and execution is restarted with a wait for the next break
In addition, the script uses print statements to output the value written to the P8 register and that read from it for use in monitoring to the Python console within the loop processing. The debugger.Register.GetValue function refers to the register value and returns the result to the Python console without the use of a print statement.
The list below shows the result of monitoring output on the Python console panel. This is the result of output of the values written to and those read from the P8 register.



As described in the section titled “Explanation of the Program, in this case where g_count in the for loop starts at 0, the values written and read are the same. When g_count starts with 1, the values written are inverted before being read as the values for output.
This completes the explanation of script 1.

4.2 Script 2

An overview of script 2 is given below.

Script 2 is used in automatic fuzzing test with the values for testing taken from an Excel file. Fuzzing test is a form of testing for confirming the behavior of programs in response to the input of random values.

This script is intended to demonstrate the following items.

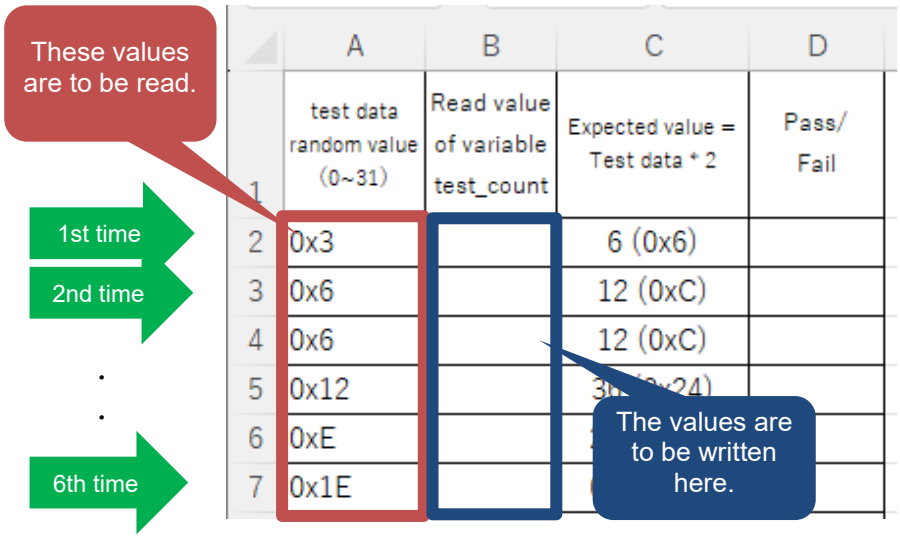
- Proceeding with fuzzing test by the input of values for use in testing (random values) to the test_count variable in the program
- Using an Excel file to set and store the values for use in testing
- Proceeding with testing by repeating the processing stated above

Specifically, the following operations proceed.

1) **Reading a value for use in testing** from the Excel file in response to a break
2) Rewriting the test_count variable with the value for testing obtained in step 1
3) Operation on the test_count variable (in the program)
4) Reading the test_count variable and **writing it to the Excel file** on detection of the next break
5) Executing steps 1 to 4 the specified number of times (16 times)

* Automatically executing the processing stated above (applied in the same way as in script 1)

The figure below shows the Excel sheet for use in setting and storing values for testing.



Script 2 reads random values for use in testing from column A of the Excel sheet, writes the results of calculation (test_count *= 2) to column B, compares the results with the expected values in column C, which were calculated beforehand, and indicates the results of judgment in column D of the sheet.

The script repeats these operations for the number of input values on the Excel sheet. It is combined with the script for automatic execution, which is applied in the same way as in the script for project 1.

As this project does not use any peripheral functions of the MCU, it runs on the RH850 simulator as the debug tool.

Configuration of files in the second project

As the configuration of files in project 2 is relatively complicated, the detailed descriptions are given below.

Program:

- main.c:
  The test_count variable and its processing (test_count*= 2) for use in testing are written in main().

Three scripts:

- ExcelReadWrite.py:
  This script handles various types of processing such as setting Excel, defining the user functions (open, close, read, write) for use in operations, and registering a hook function. It is loaded into the memory for execution before debugging.

- ExcelReadWriteHook.py:
  This script writes the hook function which is called during break. It is registered during execution of ExcelReadWrite.py and loaded into the memory along with ExcelReadWrite.py. Hook functions are executed during specific events and have fixed names. For example, AfterCpuStop() and AfterCpuReset() are executed following a break and CPU reset, respectively.

- AutoExecute.py:
  This script is for automatic execution and is partly an application of some of script 1. It handles various types of processing such as setting the breakpoint and connecting to and disconnecting from a debug tool. ExcelReadWrite.py can also be run during manual debugging.

Excel file:

- ExcelReadWrite.xlsm:
  This file is used for data processing. It is created beforehand to generate random values for testing to be written to the test_count variable, read and store the test_count values after calculation, compare them with the expected values, and judge the results of comparison.

1) Preparing the Second Project
   The following is preparation in advance of using the second script.
   Steps: As steps i and ii are a duplication of steps described in section 3.1, the related figures are omitted here.
   i. Opening the project
   ii. Displaying the Python console and adjusting its position
   iii. Preparing for debugging with the use of the scripts (setting the debug layout)

   Step i Opening the project
   Open the project, which comes with this application note.
   Use the RH850_F1L_ExcelReadWrite project. This is packaged in the “AutoFuzzTest_ExcelData” folder. Extract the given folder in the location you desire. In this description, it is extracted on your desktop as an example. Double-click on RH850_F1L_ExcelReadWrite.mtpj in the RH850_F1L_ExcelReadWrite folder under the extracted “AutoFuzzTest_ExcelData” folder to open the project.

   Step ii Displaying the Python console and adjusting its position
   <1> Select [Python Console] from the [View] menu.
   <2> The Python Console panel appears.

   Step iii Preparing for debugging with the use of the script (setting the debug layout)
   This setting is for checking display by the Python console in real time during execution of the Python script. As the layout is exclusively for use during debugging, the setting should be made in advance of executing the script.
   At the same time, make preparations for checking the variables and register values for use in debugging the script (watch expression settings).

   <1> Connect to the debug tool.
   Pressing the  button (used to initiate downloading of the program to the debug tool) in the upper-right corner of the CS+ window makes a connection to the debug tool and downloads the program.

<2> Display the Python console and move it to the location you desire.
This is for checking the Python output during execution of the script. This step is partially a repetition of the descriptions in section 3.2.

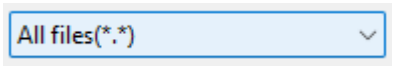
i. Select [Python Console] from the [View] menu.
ii. Right-click on the title bar of the Python console and select [Floating] from the pop-up menu.
iii. Drag and drop the title bar under the Project Tree panel.

<3> To use the GUI to monitor the values of variables during execution, display the Watch panel and register watch expressions.
The variables to be monitored here are wait, g_count, and test_count.

i. Select [Watch] ⇒ [Watch1] from the [View] menu.
ii. Register the watch expressions.
In sequence, right-click on each of the wait, g_count, and test_count variables in the main.c entry in the Editor panel. Select [Register to Watch1] at the top of the context menu that appears.
These variables are now registered with the Watch panel.

<4> Disconnect from the debug tool.
After having registered the variables, press the  button (disconnect button) in the upper-right corner of the CS+ window to disconnect from the debug tool.
The preparations are completed at this point.

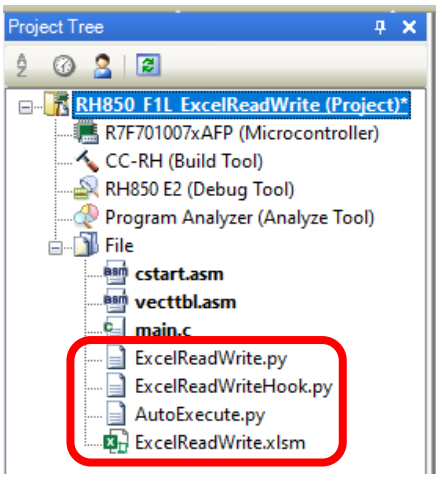
2) Executing Script 2
Before explaining the script, we will check its operation.
Script 2 actually consists of three scripts
Step i Registering the scripts
When the project is already in use, the scripts will have been registered. This step is for the case where they have not been registered.
<1> Right-click on [File] in the Project Tree panel and then select [Add] ⇒ [Add Existing File] from the context menu.
<2> The [Add Existing File] dialog box opens.
As registering the Excel file is also required in this case, select [All files (*.*)] as the file type.



- ExcelReadWrite.py ⇐ Sets Excel, etc.
- ExcelReadWriteHook.py ⇐ Called during breaks.
- ExcelReadWrite.xlsm ⇐ Excel file to handle data processing
- AutoExecute.py ⇐ Application of the parts of script 1 for automatic execution with a little extra code

Select the four files listed above.

<3> The four files enclosed in red lines shown below are now registered.



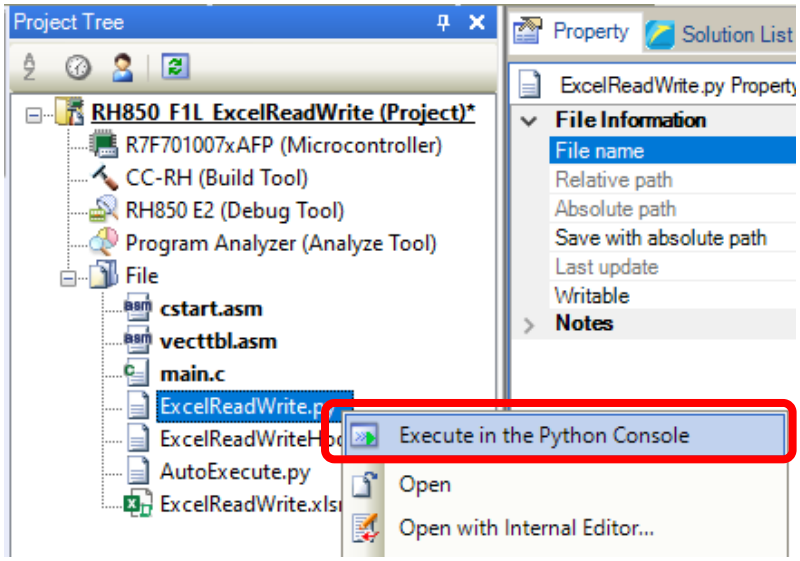
<4> Initialize the ExcelReadWrite.xlsm Excel file.
Double-click on [ExcelReadWrite.xlsm] in the Project Tree panel to open the ExcelReadWrite.xlsm file.

	A	B	C	D	E	F
1	test data random value (0~31)	Read value of variable test_count	Expected value = Test data * 2	Pass/ Fail		Random occurrence (0~31)
2	0x3		6 (0x6)			0xF
3	0x6		12 (0xC)			0x10
15	0x13		38 (0x26)			0xF
16	0x2		4 (0x4)			0x19
17	0xF		30 (0x1E)			0x9

If any values have already been written to “Values Read from the test_count Variable” column B, clear them.
To use new random values, copy F2 to F16 from column F and paste the copied values to A2 to A16 in column A. Use [Paste Values] for pasting.
* The ExcelReadWrite.xlsm Excel file included in the second project incorporates macros for clearing column B, copying random values, and updating.

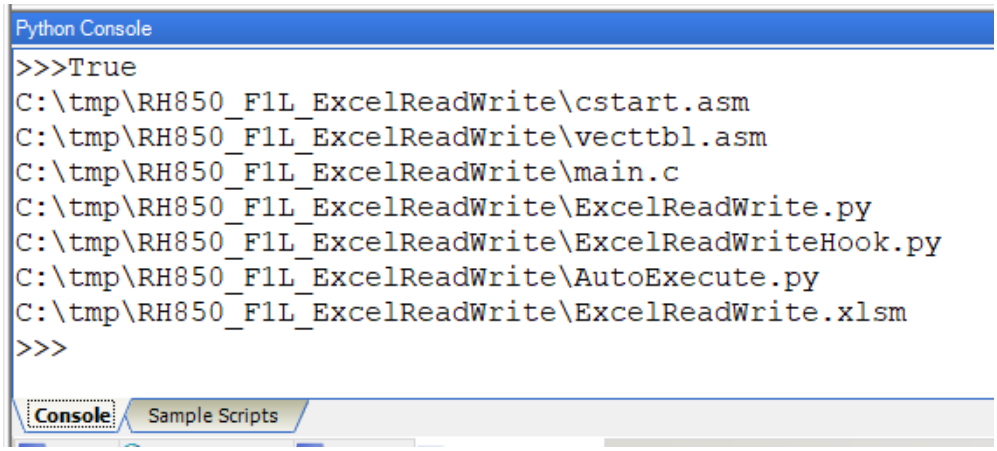
<5> Before executing the script, initializing the Python console and clearing of the screen are required.
Right-click on the Python Console panel.
Only execute this step if it is necessary.

<6> Execute the ExcelReadWrite.py script file.
This script loads the Excel settings and user-defined functions into Python.
Right-click on [ExcelReadWrite.py] in the Project Tree panel and select [Execute in the Python Console] at the top of the context menu. The script starts running at this point in time.



<7> Results of executing ExcelReadWrite.py

The figure below shows ExcelReadWrite.py searching for the Excel file in the project folder. This is how the script obtains the path to the Excel file.

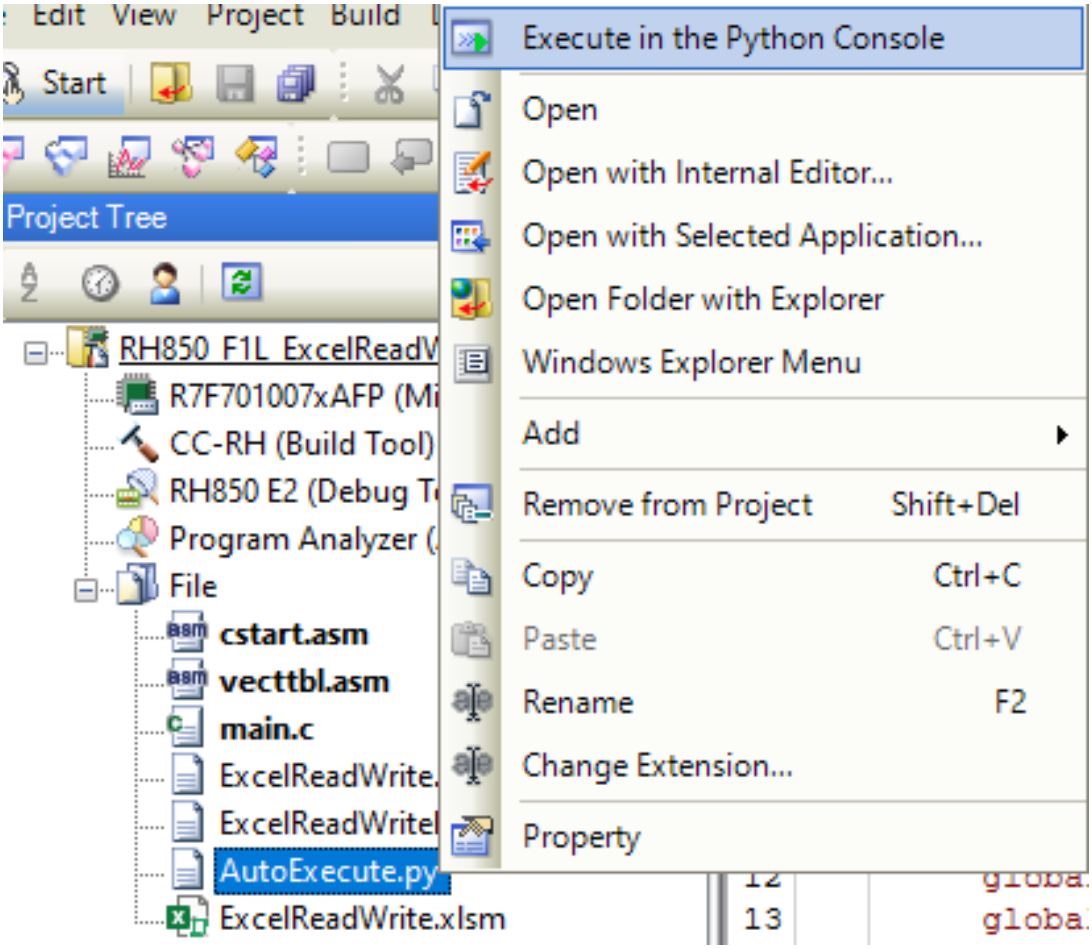


This is all that is displayed, but the Excel settings and user-defined functions are also loaded into Python.

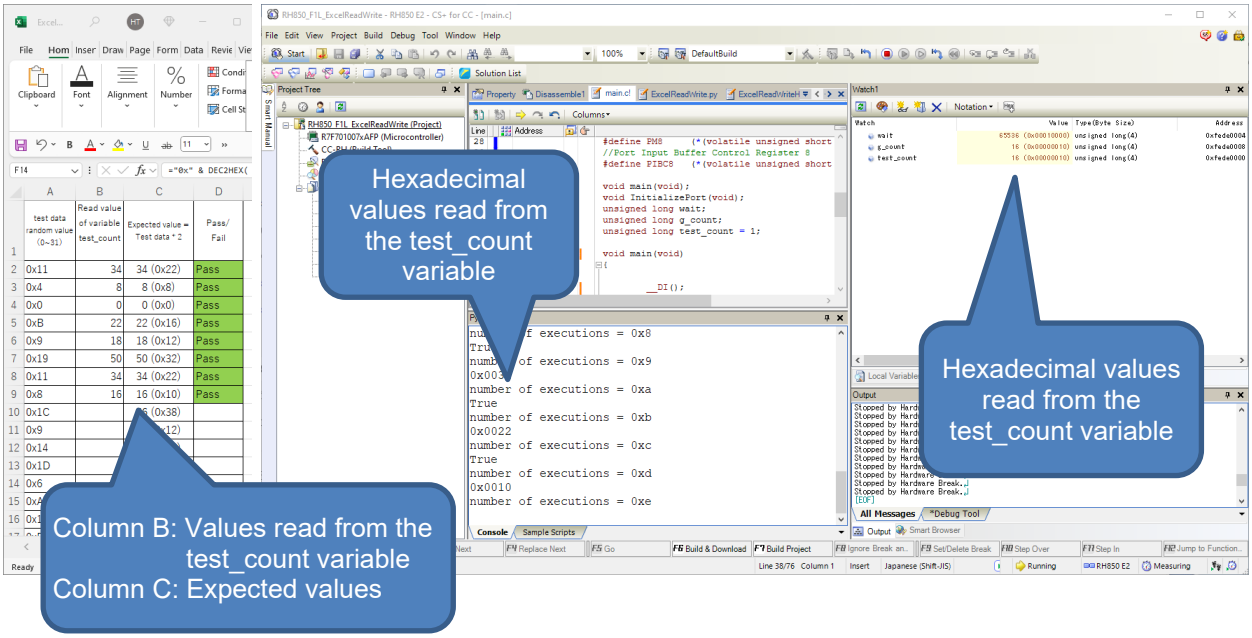
<8> Execute the AutoExecute.py script file.

This script automatically executes the processing for debugging such as connection to the debug tool and setting of the breakpoint.

Right-click on [AutoExecute.py] in the Project Tree panel and select [Execute in the Python Console] at the top of the context menu. The script starts running at this point in time.



<9> The figure below is a screenshot taken during the execution of AutoExecute.py.



The script handles processing as follows.
Following a break, the script opens the Excel file, reads a random value from column A starting with the first, and writes the given value to the test_count variable. Following the next break, the script reads the doubled test_count value and writes it to Excel column B. Excel compares the written value with the expected value in column C and indicates the result of judgment in column D. The script repeats these steps for each random value from column A, then closes Excel and disconnects from the debugger.
Set the Python Console and Watch panels for use in debugging before running the project.
The results of reading test_count are displayed in the Python Console panel of CS+.
You can use the Watch panel to monitor the behavior of variables.

3) Explanation of the Program and Scripts
This section describes the program, and then the scripts.

Explanation of the Program

Overview: Adding the test_count variable to be used in reading and writing and the expression to execute in between, test_count*= 2

main.c

```
unsigned long  test_count=1;
main(){
  :
  while(1)
 {
  for(g_count= 1; g_count<0x20;
g_count++){
   for( wait=0; wait<0x00020000;
wait++ );
   PNOT8 = 0x00F0;

   test_count*= 2;
  }
 }
}
```

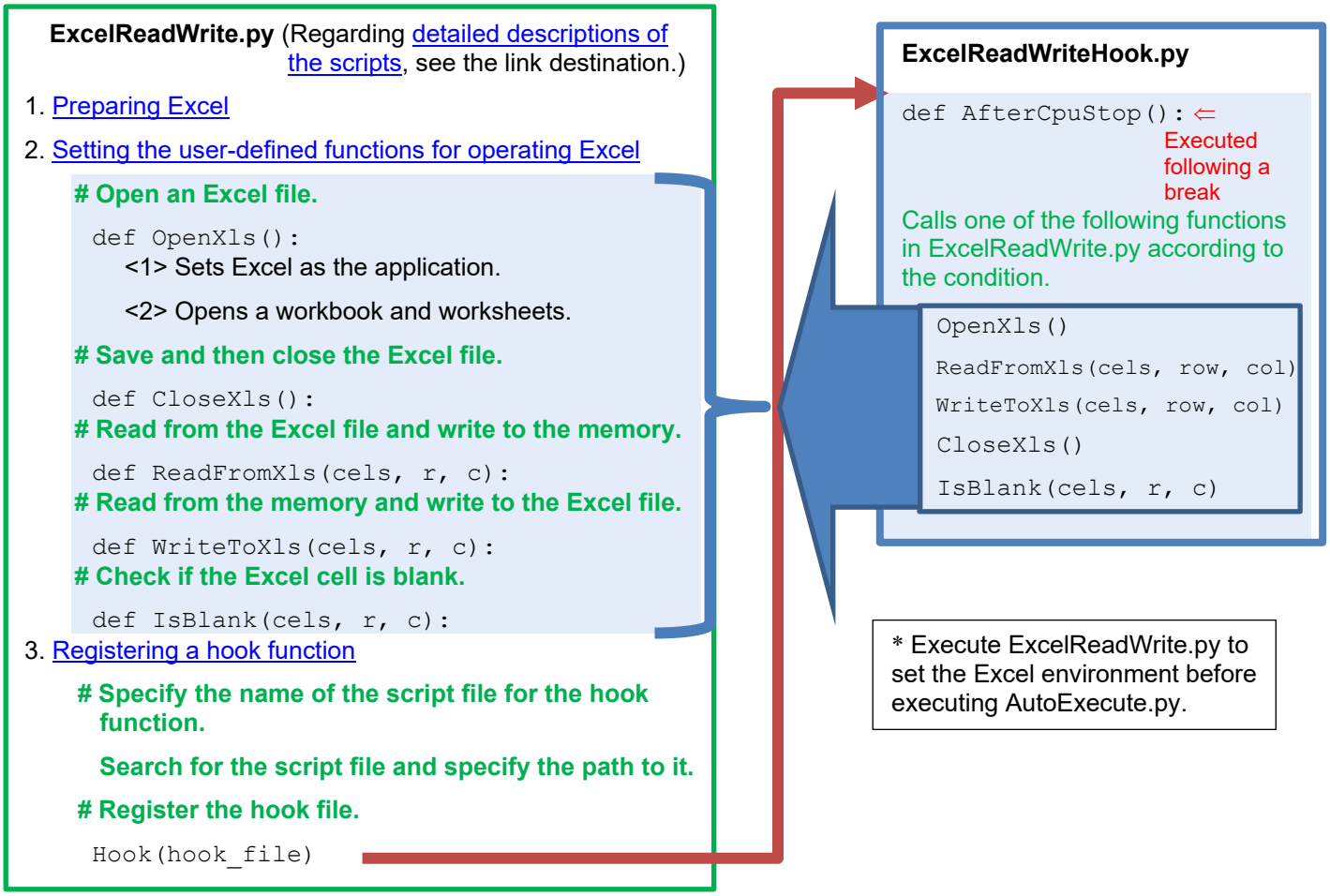
- Defines the test_count variable
- A break has been set to occur in response to writing to PNOT8.
- Process to double the value of test_count

As this project is intended for fuzzing test the test_count variable, the descriptions of the program also focus on this intention. Firstly, the test_count variable is defined, and then the processing to be tested, that is, doubling the value of test_count, is added to the program.
Using the script to write to and read from test_count requires stopping the program at a breakpoint. Set the breakpoint while the debug tool is connected. In this case, set it such that a break occurs in response to writing to PNOT8. The setting can manually be made. However, in this project, the AutoExecute.py automatic script is used to make the setting.

Explanation of the Scripts

Script 2 actually consists of three scripts: ExcelReadWrite.py, ExcelReadWriteHook.py, and AutoExecute.py. To begin with, this section describes the first two, which are used in combination with each other to control Excel by following the procedure below.
Setting Excel ⇒ registering a hook function ⇒ executing the hook function following an event
ExcelReadWrite.py prepares Excel, sets the user-defined functions for operating Excel, and registers a hook function that is triggered during a break. The user-defined functions include those for opening the Excel file, saving and closing the Excel file, and read and write access to the Excel file. Specify the ExcelReadWriteHook.py file to be registered as the hook function.
In this project, the transition to ExcelReadWriteHook.py is made following a break, and the user-defined functions in ExcelReadWrite.py are called according to the state and conditions at that time.

Relation between ExcelReadWrite.py and ExcelReadWriteHook.py



AutoExecute.py

AutoExecute.py controls automatic execution.
This script is partly an application of some of script 1 (GoWaitBreakSetRegister.py).
This script handles the following types of processing.

— Control over the execution-related processing
— Setting the breakpoint
— Repeated processing

Regarding control over the execution-related processing and setting the breakpoint, see the section titled "Explanation of the Script" for script 1. In this case, access to PNOT8 is set as the breakpoint.
For the repeated processing, a range function is used in the for loop. The range function specifies elements from 0 to 0x20 when the number entered is 0x21, thus serving as an alternative to array notation.

Repeated processing

Checking of the number of breaks, restarting, and output to the monitor repeatedly proceed.

```
The following steps are repeated.
# Use the range function for the break_count variable to specifyelements from 0 to 0x20.
  range(0x21) is equivalent to [0x0, 0x1, 0x2, …, 0x19, 0x20].
for break_count in range(0x21):
# Restart execution (wait for the next break).
   debugger.Go(GoOption.WaitBreak)
# Output the result of monitoring to the Python console.
   print("Number of rounds of execution = 0x%x"%break_count)
```

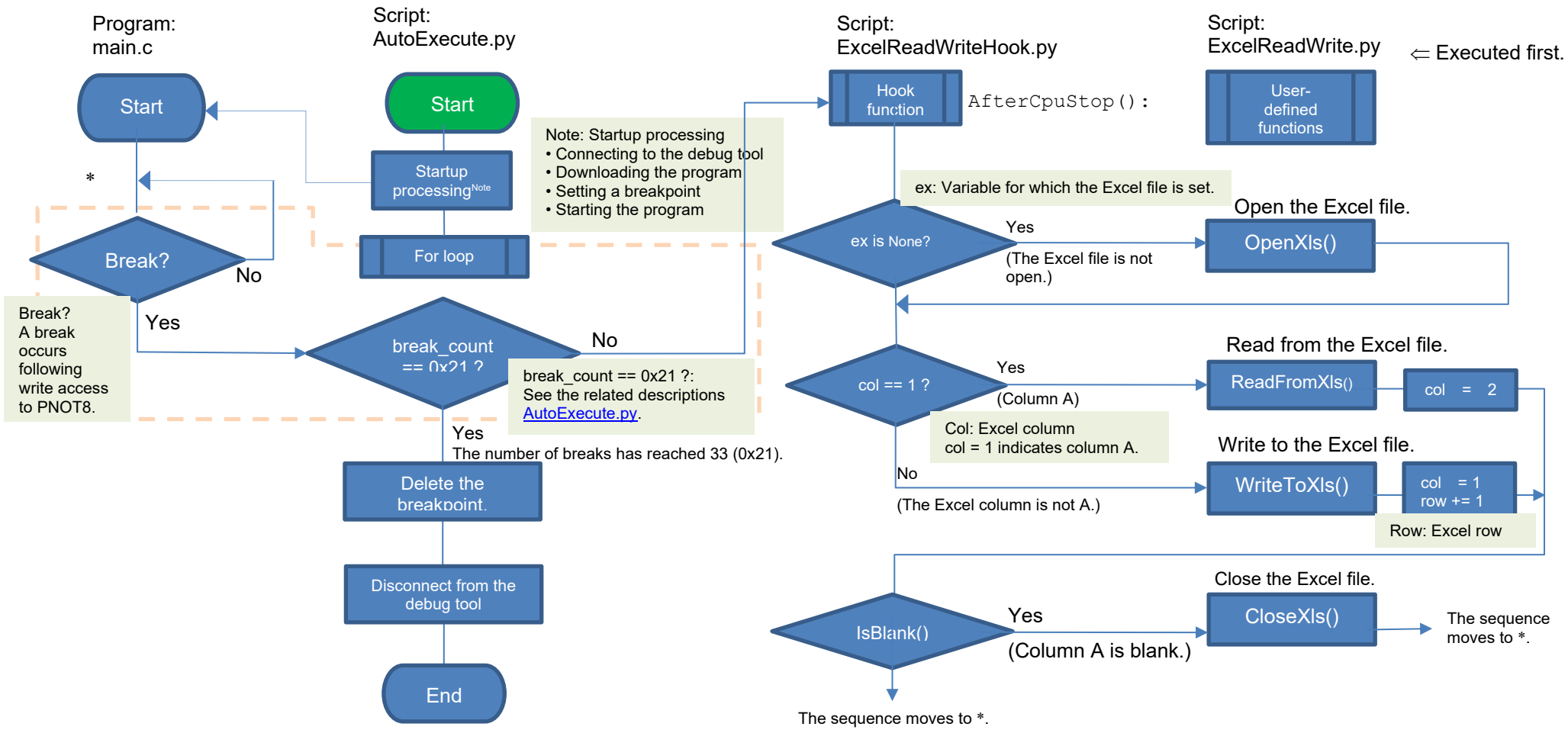
* This script has 16 (0x10) values for testing with the use of Excel.

Two breaks are required for each round of the repeated processing; one is for writing to the target variable for processing, test_count, and the other is for reading it after processing. Accordingly, break_count is set to 0x10 * 2 + 1 = 0x21.

Flowchart for Operation of the Program and Scripts in Combination

The figure on the next page illustrates the flow related to calls of the user-defined functions. It also illustrates how AutoExecute.py is involved in the testing.
AutoExecute.py connects to the debug tool, downloads the program, sets a breakpoint, and starts the program.
In the flowchart on the next page, these steps are collectively referred to as the startup processing.
After that, AutoExecute.py executes the for loop, where it counts the number of breaks and re-executes the program.
When the number of breaks reaches 0x21, or 33 times, AutoExecute.py leaves the loop, deletes the breakpoint, disconnects from the debug tool, and stops running.
The number of rounds of the loop is set slightly higher than necessary.

Flowchart for Operation of the Program and Scripts in Combination



Tips for Use in the Debugging of Python Scripts are given in an appendix.

Detailed Descriptions of the Scripts

Detailed descriptions of the individual scripts are given below.

ExcelReadWrite.py

Detailed description of ExcelReadWrite.py 1: Preparing Excel

The following summarizes the parts of ExcelReadWrite.py that are related to preparing Excel.

The figure on the left side lists the import statements, which import the module for operating the OS and the library module for use with the .NET Framework, and register the settings required to make Excel usable.

The figure on the right side lists the statements for defining and setting the variables related to the Excel file to be used. These involve making the settings for the filename, workbook, worksheets, cells, row, and column.

```
# Module for use in operating the OS above Python
Import os

# Import the .NET Framework module.
Import clr

# Load the following .NET assemblies and import them into Python.
clr.AddReference("office")
clr.AddReference("Microsoft.Office.Interop.Excel")
from Microsoft.Office.Coreimport*
from Microsoft.Office.Interop.Excelimport*
from System.Runtime.InteropServicesimport*
```

```
# Define and set the variables related to the Excel file to be used.

# Specify the Excel file as the ex variable.
ex  = ApplicationClass()

# Workbook
wb  = ex.Workbooks.Open(xls_file)

# Worksheets
wss  = wb.Worksheets

# Sheet 1
ws  = wss[1]

# Cells
cels  = ws.Cells

# Initial setting for the row
row  = 2

# Initial setting for the column
col  = 1
# Filename
xls_file_name  = "ExcelReadWrite.xlsm"
```

(Continued on next page)

ExcelReadWrite.py

Detailed description of ExcelReadWrite.py 2: Setting the user-defined functions for operating Excel

The following summarizes the following user-defined functions for operating Excel from ExcelReadWrite.py.

— OpenXls() for opening the Excel file
— CloseXls() for saving and then closing the Excel file
— ReadFromXls() for reading a variable from the Excel file and writing it to the memory
— WriteToXls() for reading a variable from the memory and writing it to the Excel file

 # Function for opening the Excel file

```
def OpenXls():
    # Search for the Excel file.
    xls_file=""
    for path inproject.File.Information():
        if os.path.basename(path) ==xls_file_name:
            xls_file=path
    # Open the Excel workbook and worksheets.
    wb=ex.Workbooks.Open(xls_file)
    wss=wb.Worksheets
    ws=wss[1]
    cels=ws.Cells
```

 # Function for saving and then closing the Excel file

```
def CloseXls():
    # Save the workbook (file).
    wb.Save()
    # Free the memory.
    Marshal.ReleaseComObject(cels)
    Marshal.ReleaseComObject(ws)
    Marshal.ReleaseComObject(wss)
    # Close the workbook (file).
    wb.Close(False)
    # Free the memory.
    Marshal.ReleaseComObject(wb)
    # Quit Excel.
    ex.Quit.()
    # Free the memory.
    Marshal.ReleaseComObject(ex)
```

```
# Specify #test_count as the target address for write access.
write_address = "test_count"  # Specifying the actual address, 0xfede0000, is also possible.
```

 # Function for reading a variable from the Excel file and writing it to the memory

```
def ReadFromXls(cels, r, c):
    cell = cels[r, c]  # Identify a cell by [r: Row, c: Column].
    # Assign the value read from the cell to "v".
    v = cell.Value[XlRangeValueDataType.xlRangeValueDefault]
    Marshal.ReleaseComObject(cell)  # Free the memory.
    # Write the value to the memory.
    debugger.Memory.Write(write_address, int(v, 0),MemoryOption.HalfWord)
```

```
# Specify #test_count as the target address for read access.
read_address = "test_count"  # Specifying the actual address, 0xfede0000, is also possible.
```

 # Function for reading a variable from the memory and writing it to the Excel file

```
def WriteToXls(cels, r, c):
    cell = cels[r, c]  # Identify a cell by [r: Row, c: Column].
    # Assign the value (variable) read from the memory to "v".
    v = debugger.Memory.Read(read_address, MemoryOption.HalfWord)
    cell.Value = v  # Write the value to the Excel cell.
    Marshal.ReleaseComObject(cell)  # Free the memory.
```

ExcelReadWrite.py & ExcelReadWriteHook.py

Detailed description of ExcelReadWrite.py 3: Statements related to the hook function and a detailed description of ExcelReadWriteHook.py

The following summarizes the information related to the hook function.

ExcelReadWrite.py registers a hook function. It does this by specifying the name of the script file, in this case ExcelReadWriteHook.py, in which the given hook function is written.

The AfterCpuStop() is specified for the hook function so that it is executed at the breaks in program execution.

The state of the variables related to the Excel file selects one of the following user-defined functions for execution.

— OpenXls() for opening the Excel file
— CloseXls() for saving and then closing the Excel file
— ReadFromXls() for reading a variable from the Excel file and writing it to the memory
— WriteToXls() for reading a variable from the memory and writing it to the Excel file

ExcelReadWrite.py

```
# Name of the script file for the hook function
hook_file_name = "ExcelReadWriteHook.py"
# Search for the script file for the hook function.
hook_file = ""
for path in project.File.Information():
 if os.path.basename(path) == hook_file_name:
       hook_file = path
# Register the hook function.
Hook(hook_file)   # Specify the filename with the path.
```

ExcelReadWriteHook.py

```
def AfterCpuStop():  ⇐ Specifies execution of the function following a break.
    # Conditional branching
    if row == 0: # Case where the initial value of the row, 2, has not been set
      return      # Do nothing.
    if ex is None:    # Case where Excel is not open.
      OpenXls()     # Open Excel.
    if col == 1: # Case where this is the first column (values for testing)
      # Read from the cell in the first column and write the result to the memory.
      ReadFromXls(cels, row, col)
      col = 2 # Set the second column (for writing the result).
    else:  # Case where this is the second column (for writing the result)
      # Read the value from the memory and write it to the cell in the second column.
      WriteToXls(cels, row, col)
      col = 1  # Return to the first column (values for testing).
      row += 1  # Move to the row with the next value for use in testing.
    if IsBlank(cels, row, 1): # Case where the cell in the first column has no value
      # Reading and writing are ended, so quit Excel.
      CloseXls()
      row = 0
      col = 0
```

5. Command-Line Operation of CS+ and Python Scripts

This section describes how to execute CS+ and Python scripts from a command line.

CS+ Python scripts can be operated without displaying the CS+ main window (GUI) by using a command input method such as starting them from a command line (Command Prompt of Windows).

This is ideal for applications like continuous device testing (CI/CD) without launching CS+ (GUI) at inspection or other sites.

Commands can be executed from the Command Prompt of Windows in the following format.

Format: Manipulate CS+ on the Command Line | V8.06.00 (renesas.com)

CubeSuite+.exe /ps [Name of the script file] [Name of the project file]

Example: Taking project 2 as an example:

CubeSuite+.exe /ps AutoExecuteWithExcelpy.py RH850_F1L_ExcelReadWrite.mtpj

With display of CS+:

CubeSuite**W**+.exe /ps AutoExecuteWithExcelpy.py RH850_F1L_ExcelReadWrite.mtpj

Here, we will use a new script referred to as AutoExecuteWithExcelpy.py.

In the descriptions of project 2 so far, it is assumed to have the three scripts.

ExcelReadWrite.py
ExcelReadWriteHook.py

AutoExecute.py

The ExcelReadWrite.py and AutoExecute.py scripts are assumed to be executed individually. The advantage of dividing the scripts is allowing you to debug them individually. We also divide the scripts in order to help you understand the interesting feature that executing them in sequence enables processing as intended.

On the other hand, individually executing the two divided scripts takes a lot of time and effort, especially from the command line.

AutoExecuteWithExcelpy.py makes AutoExecute.py call ExcelReadWrite.py by adding the following code at the top of AutoExecute.py, as a result allowing the execution of both scripts as a batch.

Source("ExcelReadWrite.py")

Actual Operation

There are two actual operation methods, which share a requirement for advance preparation.

0) Registering the folder containing the CS+ executable files with the path Windows environment variable (advance preparation)

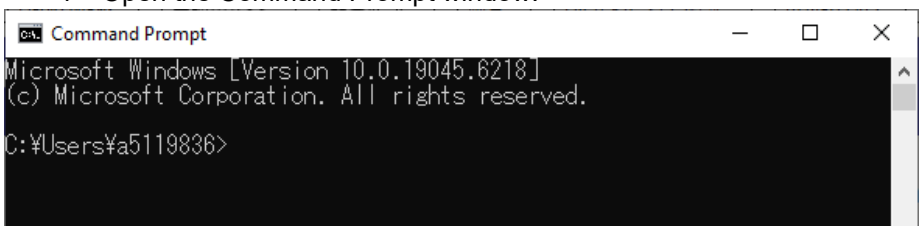
Open the [Environment Variables] dialog box from the [Environment Variables (N)] option on the [Advanced] tabbed page of the Windows 10 System Properties dialog box. Select the [Path] system environment variable, click on the [Edit] button. The [Edit Environment Variable] dialog box appears. Click on the [New] button to enter the new path.

“C:¥Program Files (x86)¥Renesas Electronics¥CS+¥CC” will normally be used.

1) Method 1: Execution from the Command Prompt

The first method is with the use of the Command Prompt.

<1> Open the Command Prompt window.

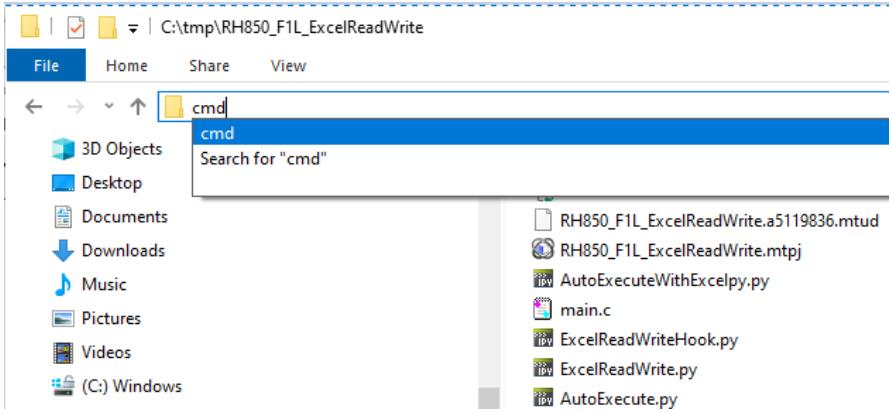


<2> Move to the folder for use with the project.

<3> Enter and execute the command.

Pressing the up-arrow key (↑) allows re-entry of the command.

Note that steps 1 and 2 above can be performed simultaneously by entering “cmd” into the address bar of an Explorer window in which the project is open.

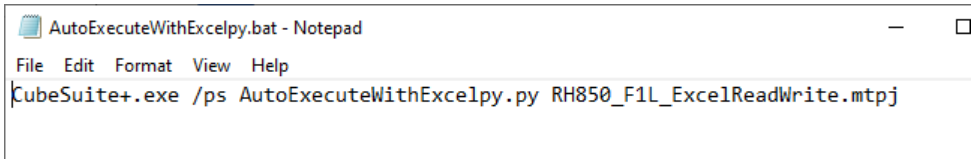


2) Method 2: Execution from a batch file

The second method is with the use of a batch file.

<1> Generate a batch file.

Write the command and save it in a file with the .bat filename extension in the project folder.



<2> Double-click on the batch file to execute it.

The batch file is also executable from the Command Prompt.

The Command Prompt will open and display the information displayed in the CS+ Python console panel. After a moment, Excel will be opened and the test will run.

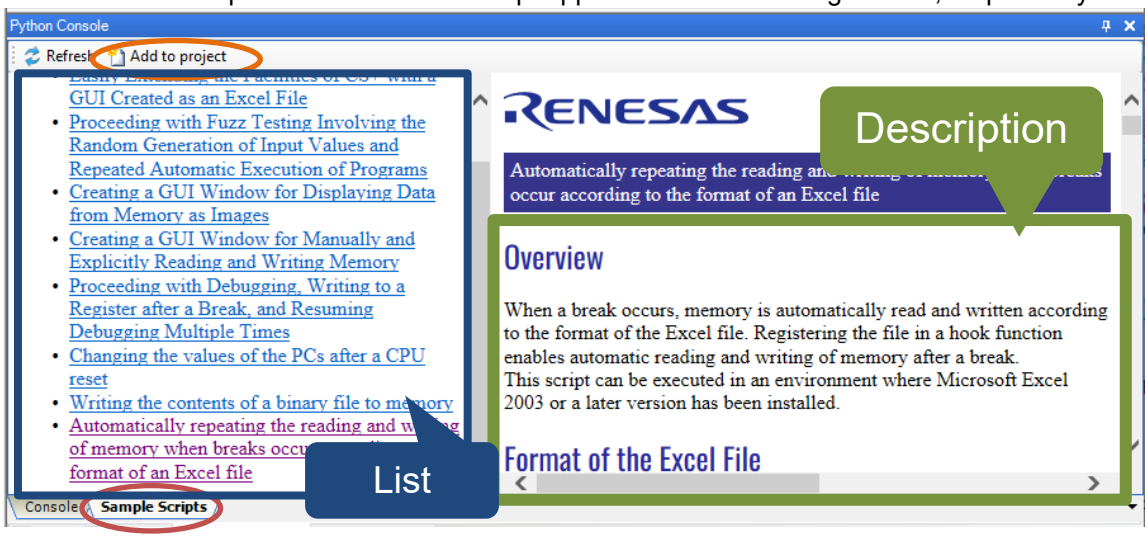
6. Approach to the Next Step

This application note covers general information on the Python functionality. Finally, it gives two pieces of information to help you in approaching the next steps: sample scripts and the “Renesas Engineering Community.”

6.1 Sample Scripts

We provide nine scripts for use with CS+ for CC for reference.

1) Click on the **[Sample Scripts] tab** to the right of the Python Console panel. The list of the sample scripts and the description of the selected script appear on the left and right sides, respectively.

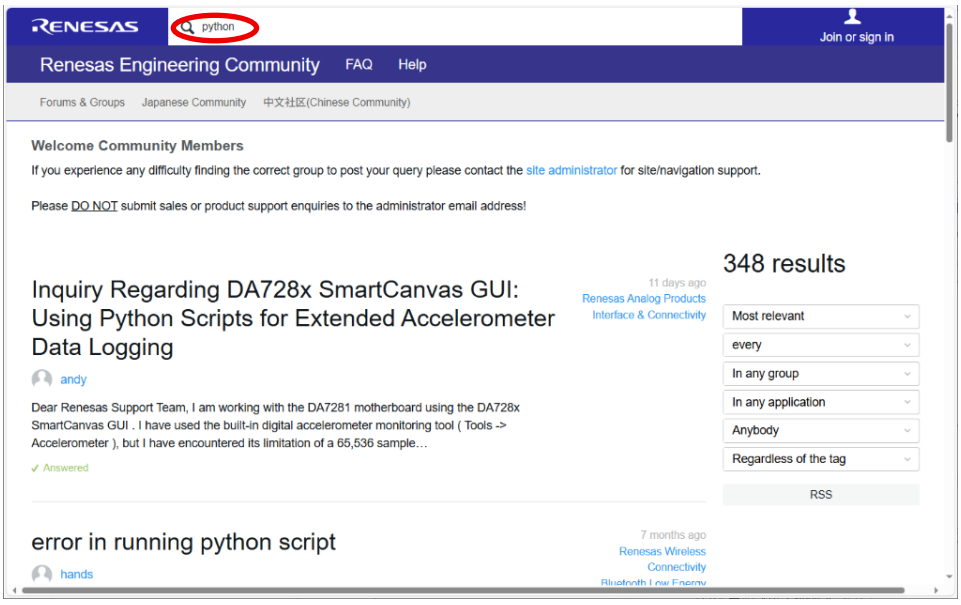


2) Select a desired sample script from the list and click on [Add to Project] in the upper-left corner. The script file or files (.py) and any other required files, such as an Excel file, will be added to the given project.

6.2 Renesas Engineering Community

The other piece of useful information is the availability of a forum for users of Renesas development tools, the Renesas Engineering Community.

Searching for Python on the Renesas Engineering Community (https://community.renesas.com/g) site allows you to view various types of information on Python, use cases, and Q&A items. You can also consult forum members.



Appendices

Python Implementation in CS+

This is an execution environment for use with IronPython (Python running in the .NET Framework and compatible with Python 2.7).

- Due to limitations on functions requiring option specifications, some functions (such as the pdb debugger function) cannot be used.
- Libraries written in any language other than Python cannot be used.
- Since IronPython is currently compatible with Python 2.7, functions added to later versions of Python cannot be used.
- Using control statements such as for and if statements enables the creation of programs as scripts.
- Register names for reference can be specified directly without defining them.
- For details on the IronPython language specification, refer to the following URL.
  http://ironpython.net/
  User's Manual:
  CS+ V8.06.00 Integrated Development Environment User's Manual: Python Console

Return to the related page.

.NET Framework

Features:

- The .NET framework is a useful component included in Windows, and can be used to run or create programs.
- Users of Windows programs can use this module as a standard function.
- Calls of .NET framework API functions are possible from any language.

For more details, refer to the following sites (Microsoft official websites).

- NET Framework Documentation| Microsoft Docs
- https://learn.microsoft.com/en-us/dotnet/

Return to the related page.

.NET Assemblies

.NET assemblies are libraries for use in the .NET Framework.

- ▪ The filename extension for a .NET assembly is *.EXE or *.DLL.

In other words,

.NET assemblies are components (provide an environment) that can be used to run or create programs included in Windows.

.NET assemblies can be loaded and imported into Python.

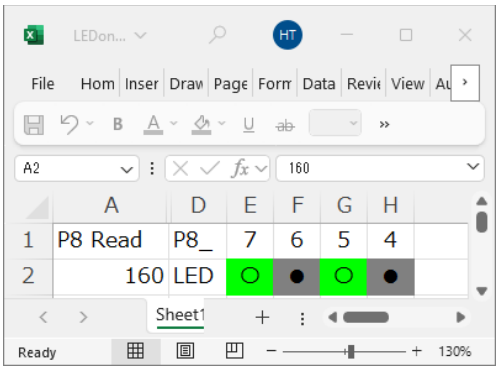
Example:

```
Import clr          ⇐ Imports the program execution engine module called “common
                      language runtime (CLR)” of the .NET Framework.
clr.AddReference("office")
clr.AddReference("Microsoft.Office.Interop.Excel")
from Microsoft.Office.Core import*
from Microsoft.Office.Interop.Excel import*
from System.Runtime.InteropServices import*
```

Return to the related page.

Project for Using Excel to Monitor the Behavior of the LEDs in the First Project

If you do not have or do not want to use an evaluation board for the first project, you can check the operation in the RH850 simulator for CS+, but this does not allow monitoring the behavior of the LEDs. Therefore, we also provide a separate project in which Excel is used to monitor the behavior of the LEDs.



Double-click on LED_by_Excel.mtpj in the LED_by_Excel_Pproject folder under the “AutoDebug_MultiRounds_WriteRegister_Resume” folder to open the project. Execute LEDonExcel.py, and then GoWaitBreakSetRegister2.py.

For details on scripts related to control over Excel, see the relevant parts of section 4.2 Script 2.

Return to the related page.

CI/CD

Continuous integration (CI) is an approach for applying automatic and continuous building and testing in software development.

Continuous delivery (CD) is a process that allows automatic processing such as merging code tested through CI and creating builds for the production environment in preparation for actual deployment to production.

In addition, “CD” is sometimes used as an abbreviation for “continuous deployment.” The concept of continuous deployment is very similar to that of continuous delivery, but the two differ in that continuous delivery only involves preparation for deployment, whereas continuous deployment actually proceeds through to the stage of deployment to production.

These methodologies for automatically and continuously proceeding with building, testing, and deploying are collectively referred to as “CI/CD.”

Return to the related page.

Tips for Use in the Debugging of Python Scripts

Commands for use in debugging

Since the CS+ implementation of Python does not support pdb, examples of the use of commands that are helpful for debugging are given below.

— Output of a message
- print: print(variable), print"message", or print("message")

— Suspending execution
- sleep: import time
  time_duration = 60
  time.sleep(time_duration)
- pause: import os
  os.system("pause")

— Forcible termination
- exit: import sys
  sys.exit()

To forcibly quit a script that is in progress, right-click on the Python console and select [Abort], or press Ctrl + D.

Configuration of a script

Scripts can be created as parts which are individually executable. As described in section 4.2 Script 2, the scripts for the project are divided into two parts: an Excel control-related script and the script for its hook function, and a script for handling automatic execution, and they are executed independently. This makes debugging easier. In actual execution, using the Source function to call one from the other allows executing the scripts in bulk.

Usage note on the Source function:

How to specify a script file to be called by using the Source function requires attention. When the script is to be executed from the CS+ project tree or Python console, an error of the following kind may occur.

For example, when Source(“sample.py”) is specified, an error message “Could not find file 'C:¥Program Files (x86)¥Renesas Electronics¥CS+¥CC¥sample.py'.” will appear. This occurs because the current directory (folder) is the directory (folder) containing the CS+ executable file, CubeSuiteW+.exe. To avoid this, the script file must be specified by using an absolute path. However, since projects to be used may have been moved or copied, use the following method to avoid the error by flexibly specifying the absolute path.

```
# Specify the name of the script file to be called.
Source_file_name = "sample.py"

# Search for the script file.
Source_file = ""
for path in project.File.Information():
    if os.path.basename(path) == Source_file_name:
        Source_file = path
Source(Source_file)
```

Note that when the script is to be executed from a batch file in the project folder, simply using Source(“sample.py”) does not create a problem. This also applies in the case of a hook function.

Return to the related page.

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Nov.24,.2025	—	First edition issued

General Precautions in the Handling of Microprocessing Unit and Microcontroller Unit Products

The following usage notes are applicable to all Microprocessing unit and Microcontroller unit products from Renesas. For detailed usage notes on the products covered by this document, refer to the relevant sections of the document as well as any technical updates that have been issued for the products.

1. Precaution against Electrostatic Discharge (ESD)

   A strong electrical field, when exposed to a CMOS device, can cause destruction of the gate oxide and ultimately degrade the device operation. Steps must be taken to stop the generation of static electricity as much as possible, and quickly dissipate it when it occurs. Environmental control must be adequate. When it is dry, a humidifier should be used. This is recommended to avoid using insulators that can easily build up static electricity. Semiconductor devices must be stored and transported in an anti-static container, static shielding bag or conductive material. All test and measurement tools including work benches and floors must be grounded. The operator must also be grounded using a wrist strap. Semiconductor devices must not be touched with bare hands. Similar precautions must be taken for printed circuit boards with mounted semiconductor devices.
2. Processing at power-on

   The state of the product is undefined at the time when power is supplied. The states of internal circuits in the LSI are indeterminate and the states of register settings and pins are undefined at the time when power is supplied. In a finished product where the reset signal is applied to the external reset pin, the states of pins are not guaranteed from the time when power is supplied until the reset process is completed. In a similar way, the states of pins in a product that is reset by an on-chip power-on reset function are not guaranteed from the time when power is supplied until the power reaches the level at which resetting is specified.
3. Input of signal during power-off state

   Do not input signals or an I/O pull-up power supply while the device is powered off. The current injection that results from input of such a signal or I/O pull-up power supply may cause malfunction and the abnormal current that passes in the device at this time may cause degradation of internal elements. Follow the guideline for input signal during power-off state as described in your product documentation.
4. Handling of unused pins

   Handle unused pins in accordance with the directions given under handling of unused pins in the manual. The input pins of CMOS products are generally in the high-impedance state. In operation with an unused pin in the open-circuit state, extra electromagnetic noise is induced in the vicinity of the LSI, an associated shoot-through current flows internally, and malfunctions occur due to the false recognition of the pin state as an input signal become possible.
5. Clock signals

   After applying a reset, only release the reset line after the operating clock signal becomes stable. When switching the clock signal during program execution, wait until the target clock signal is stabilized. When the clock signal is generated with an external resonator or from an external oscillator during a reset, ensure that the reset line is only released after full stabilization of the clock signal. Additionally, when switching to a clock signal produced with an external resonator or by an external oscillator while program execution is in progress, wait until the target clock signal is stable.
6. Voltage application waveform at input pin

   Waveform distortion due to input noise or a reflected wave may cause malfunction. If the input of the CMOS device stays in the area between V_{IL} (Max.) and V_{IH} (Min.) due to noise, for example, the device may malfunction. Take care to prevent chattering noise from entering the device when the input level is fixed, and also in the transition period when the input level passes through the area between V_{IL} (Max.) and V_{IH} (Min.).
7. Prohibition of access to reserved addresses

   Access to reserved addresses is prohibited. The reserved addresses are provided for possible future expansion of functions. Do not access these addresses as the correct operation of the LSI is not guaranteed.
8. Differences between products

   Before changing from one product to another, for example to a product with a different part number, confirm that the change will not lead to problems. The characteristics of a microprocessing unit or microcontroller unit products in the same group but having a different part number might differ in terms of internal memory capacity, layout pattern, and other factors, which can affect the ranges of electrical characteristics, such as characteristic values, operating margins, immunity to noise, and amount of radiated noise. When changing to a product with a different part number, implement a system-evaluation test for the given product.

Notice

1. Descriptions of circuits, software and other related information in this document are provided only to illustrate the operation of semiconductor products and application examples. You are fully responsible for the incorporation or any other use of the circuits, software, and information in the design of your product or system. Renesas Electronics disclaims any and all liability for any losses and damages incurred by you or third parties arising from the use of these circuits, software, or information.
2. Renesas Electronics hereby expressly disclaims any warranties against and liability for infringement or any other claims involving patents, copyrights, or other intellectual property rights of third parties, by or arising from the use of Renesas Electronics products or technical information described in this document, including but not limited to, the product data, drawings, charts, programs, algorithms, and application examples.
3. No license, express, implied or otherwise, is granted hereby under any patents, copyrights or other intellectual property rights of Renesas Electronics or others.
4. You shall be responsible for determining what licenses are required from any third parties, and obtaining such licenses for the lawful import, export, manufacture, sales, utilization, distribution or other disposal of any products incorporating Renesas Electronics products, if required.
5. You shall not alter, modify, copy, or reverse engineer any Renesas Electronics product, whether in whole or in part. Renesas Electronics disclaims any and all liability for any losses or damages incurred by you or third parties arising from such alteration, modification, copying or reverse engineering.
6. Renesas Electronics products are classified according to the following two quality grades: “Standard” and “High Quality”. The intended applications for each Renesas Electronics product depends on the product’s quality grade, as indicated below.
   "Standard": Computers; office equipment; communications equipment; test and measurement equipment; audio and visual equipment; home electronic appliances; machine tools; personal electronic equipment; industrial robots; etc.
   "High Quality": Transportation equipment (automobiles, trains, ships, etc.); traffic control (traffic lights); large-scale communication equipment; key financial terminal systems; safety control equipment; etc.

   Unless expressly designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not intended or authorized for use in products or systems that may pose a direct threat to human life or bodily injury (artificial life support devices or systems; surgical implantations; etc.), or may cause serious property damage (space system; undersea repeaters; nuclear power control systems; aircraft control systems; key plant systems; military equipment; etc.). Renesas Electronics disclaims any and all liability for any damages or losses incurred by you or any third parties arising from the use of any Renesas Electronics product that is inconsistent with any Renesas Electronics data sheet, user’s manual or other Renesas Electronics document.
7. No semiconductor product is absolutely secure. Notwithstanding any security measures or features that may be implemented in Renesas Electronics hardware or software products, Renesas Electronics shall have absolutely no liability arising out of any vulnerability or security breach, including but not limited to any unauthorized access to or use of a Renesas Electronics product or a system that uses a Renesas Electronics product. RENESAS ELECTRONICS DOES NOT WARRANT OR GUARANTEE THAT RENESAS ELECTRONICS PRODUCTS, OR ANY SYSTEMS CREATED USING RENESAS ELECTRONICS PRODUCTS WILL BE INVULNERABLE OR FREE FROM CORRUPTION, ATTACK, VIRUSES, INTERFERENCE, HACKING, DATA LOSS OR THEFT, OR OTHER SECURITY INTRUSION (“Vulnerability Issues”). RENESAS ELECTRONICS DISCLAIMS ANY AND ALL RESPONSIBILITY OR LIABILITY ARISING FROM OR RELATED TO ANY VULNERABILITY ISSUES. FURTHERMORE, TO THE EXTENT PERMITTED BY APPLICABLE LAW, RENESAS ELECTRONICS DISCLAIMS ANY AND ALL WARRANTIES, EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT AND ANY RELATED OR ACCOMPANYING SOFTWARE OR HARDWARE, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE.
8. When using Renesas Electronics products, refer to the latest product information (data sheets, user’s manuals, application notes, “General Notes for Handling and Using Semiconductor Devices” in the reliability handbook, etc.), and ensure that usage conditions are within the ranges specified by Renesas Electronics with respect to maximum ratings, operating power supply voltage range, heat dissipation characteristics, installation, etc. Renesas Electronics disclaims any and all liability for any malfunctions, failure or accident arising out of the use of Renesas Electronics products outside of such specified ranges.
9. Although Renesas Electronics endeavors to improve the quality and reliability of Renesas Electronics products, semiconductor products have specific characteristics, such as the occurrence of failure at a certain rate and malfunctions under certain use conditions. Unless designated as a high reliability product or a product for harsh environments in a Renesas Electronics data sheet or other Renesas Electronics document, Renesas Electronics products are not subject to radiation resistance design. You are responsible for implementing safety measures to guard against the possibility of bodily injury, injury or damage caused by fire, and/or danger to the public in the event of a failure or malfunction of Renesas Electronics products, such as safety design for hardware and software, including but not limited to redundancy, fire control and malfunction prevention, appropriate treatment for aging degradation or any other appropriate measures. Because the evaluation of microcomputer software alone is very difficult and impractical, you are responsible for evaluating the safety of the final products or systems manufactured by you.
10. Please contact a Renesas Electronics sales office for details as to environmental matters such as the environmental compatibility of each Renesas Electronics product. You are responsible for carefully and sufficiently investigating applicable laws and regulations that regulate the inclusion or use of controlled substances, including without limitation, the EU RoHS Directive, and using Renesas Electronics products in compliance with all these applicable laws and regulations. Renesas Electronics disclaims any and all liability for damages or losses occurring as a result of your noncompliance with applicable laws and regulations.
11. Renesas Electronics products and technologies shall not be used for or incorporated into any products or systems whose manufacture, use, or sale is prohibited under any applicable domestic or foreign laws or regulations. You shall comply with any applicable export control laws and regulations promulgated and administered by the governments of any countries asserting jurisdiction over the parties or transactions.
12. It is the responsibility of the buyer or distributor of Renesas Electronics products, or any other party who distributes, disposes of, or otherwise sells or transfers the product to a third party, to notify such third party in advance of the contents and conditions set forth in this document.
13. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
14. Please contact a Renesas Electronics sales office if you have any questions regarding the information contained in this document or Renesas Electronics products.

(Note1) “Renesas Electronics” as used in this document means Renesas Electronics Corporation and also includes its directly or indirectly controlled subsidiaries.
(Note2) “Renesas Electronics product(s)” means any product developed or manufactured by or for Renesas Electronics.

(Rev.5.0-1 October 2020)

Corporate Headquarters

TOYOSU FORESIA, 3-2-24 Toyosu,
Koto-ku, Tokyo 135-0061, Japan
www.renesas.com

Trademarks

Renesas and the Renesas logo are trademarks of Renesas Electronics Corporation. All trademarks and registered trademarks are the property of their respective owners.

6.2.‘ Contact information

For further information on a product, technology, the most up-to-date version of a document, or your nearest sales office, please visit:
www.renesas.com/contact/.