RH850 Introducing Methods of Debugging Devices Incorporating an Initially Stopped Core and Devices in Standby Mode

R20AN0558EJ0100
Rev.1.00
2020.01.08

Introduction

Devices of the RH850 family have a standby mode for controlling the power they consume and some products incorporate an initially stopped core*.

This application note describes the debugging methods for devices that incorporate an initially stopped core and applications that include transitions to standby mode.

Note: "Initially stopped core" refers to a CPU core that is not activated by release from the reset state.
"Initially stopped state" refers to the state in which an initially stopped core has not yet been activated.

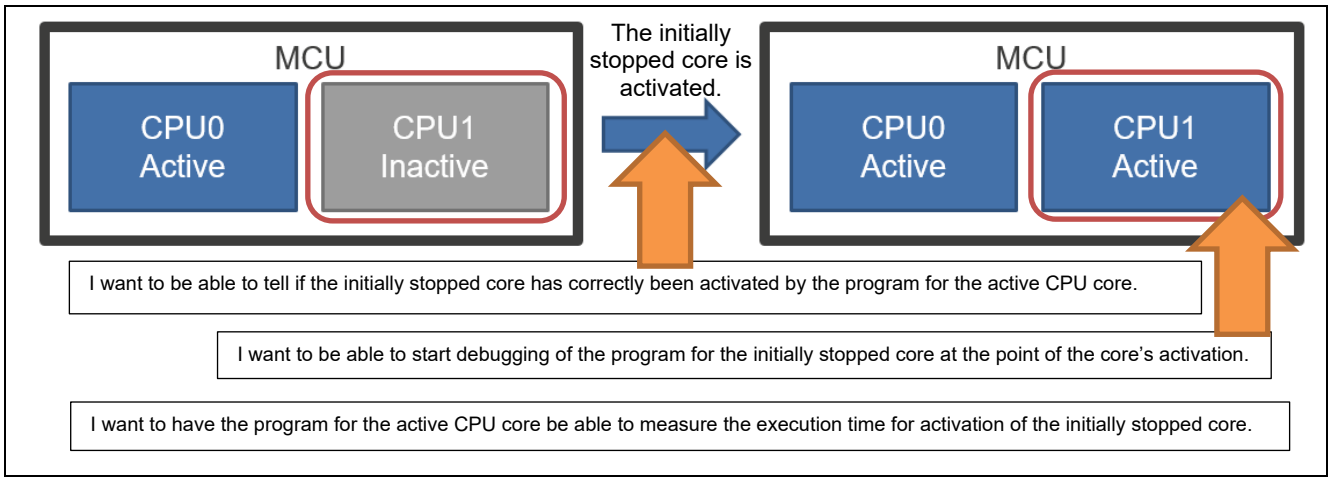

Figure 1 Requests for the Debugging of Applications for Devices that Incorporate an Initially Stopped Core

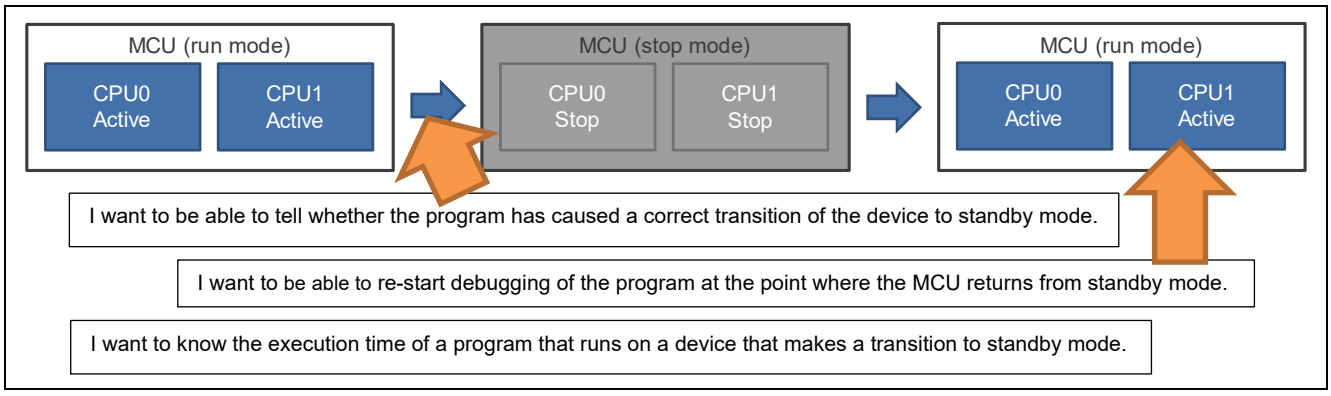

Figure 2 Requests for the Debugging of Applications that Include Transitions to Standby Mode

Target Devices

RH850/F1KM-S1

RH850/F1KM-S4

RH850/F1KH-D8

RH850/E2x (no standby modes)

RH850/U2A

Contents

Terminology

Some specific words used in this application note are defined below.

Integrated development environment (IDE):

This tool provides powerful support for the development of embedded applications for Renesas microcomputers. It has an emulator debugger function allowing the emulator to be controlled from the host machine via an interface. Furthermore, it permits a range of operations from editing a project to building and debugging it to be performed within the same application. In addition, it supports version management.

CS+:

This is an integrated development environment from Renesas.

MULTI:

This is the integrated development environment from Green Hills Software.

Emulator debugger:

This means a software tool that is started up from the integrated development environment, and controls the emulator and enables debugging.

Host machine:

This means a personal computer used to control the emulator.

Target device (MCU):

This means the device to be debugged.

User system:

This means a user's application system in which the MCU to be debugged is used.

User system interface:

This means the interface that the E1/E20/E2/IE850A emulator connects to the target device.

User system interface cable:

This means a cable that the E1/E20/E2/IE850A emulator connects to the target device.

ICU-M:

This is an abbreviation of “Intelligent Cryptographic Unit/Master”. The RH850 family includes devices that incorporate an ICU-M core as a security CPU core. The ICU-M core can be switched to enabled or disabled by a setting in an option byte.

Configuration of Manuals

The documents related to this application note consist of the following.

- E1/E20 Emulator User's Manual, E2 Emulator User's Manual, and IE850A Emulator User's Manual
- E1/E20 Emulator, E2 Emulator, IE850A Additional Documents for User's Manual
- User's manual and help for the emulator debugger
- Application Note for RH850 Introducing Methods of Debugging Devices Incorporating an Initially Stopped Core and Devices in Standby Mode (this document)

(1) E1/E20 Emulator User's Manual, E2 Emulator User's Manual, and IE850A Emulator User's Manual
These user's manuals have the following contents.
— Components of the emulator
— Hardware specifications of the emulator
— Connection to the emulator and the host machine and user system

(2) E1/E20 Emulator, E2 Emulator, IE850A Additional Documents for User's Manual
These documents describe the features of the debugger, items dependent on the given MCU, and give notes on usage.

(3) User's manual and help for the emulator debugger
The user's manual and help for the emulator debugger describe the functions of the E1/E20/E2/IE850A emulator debugger and the operating instructions.

(4) Application Note for RH850 Introducing Methods of Debugging Devices Incorporating an Initially Stopped Core and Devices in Standby Mode (this document)
Although some RH850 family devices include more than two cores, this application note covers how to debug those RH850 devices that incorporate an initially stopped core and synchronously debug programs that cause transitions of RH850 family devices to standby mode.

1. Overview

The RH850 family includes devices that incorporate an initially stopped core and a standby mode.

An initially stopped core is a CPU core that is activated by a register operation rather than automatically on release from a reset. An option-byte setting can be used to specify a CPU core as being activated on release from a reset or as an initially stopped core. For details, refer to the hardware manual for the device. Hereafter, the descriptions in this document are on the assumption that a core is set as an initially stopped core and so enters the initially stopped state on release from a reset.

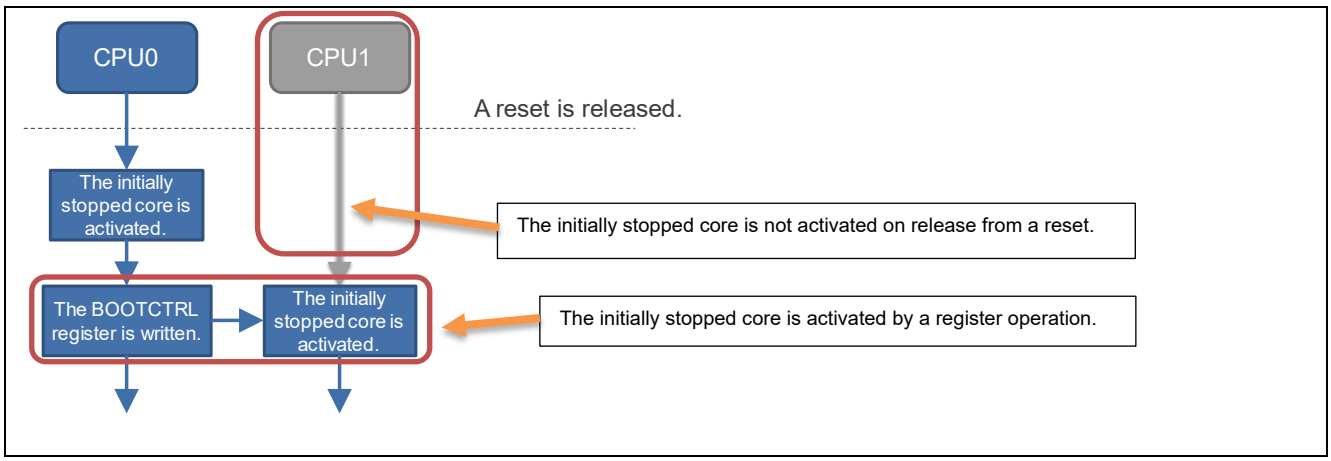

Figure 1-1 Non-Operation of an Initially Stopped Core on Release from a Reset and the Activation Sequence

The standby modes are used to control the power consumed by the device. There are four types of standby mode: stop, deep stop, cyclic run, and cyclic stop. Transitions between standby modes occur due to register operations or wakeup resources as shown in Figure 1-2. For details, refer to the hardware manual for the device.

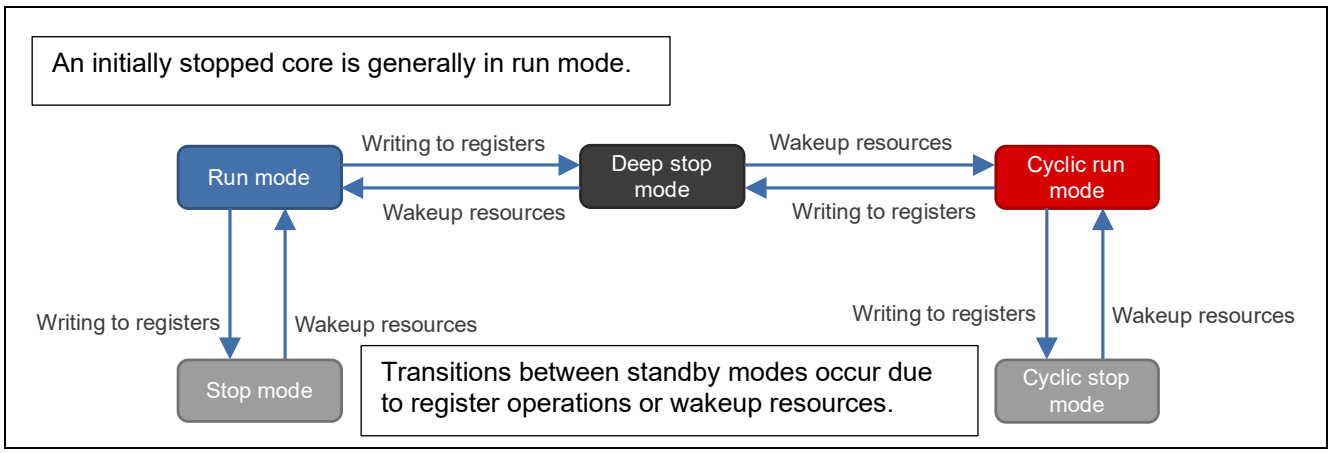

Figure 1-2 Transition between Standby Modes and Normal Operation (Run Mode)

1.1 Debugging Specifications for the Initial Setting of the Debugger

For the initial settings of the debugger, when the user uses the synchronous debugging facility* for applications that run on devices incorporating an initially stopped core or include transitions to standby mode, the respective specifications are that the debugger activates the initially stopped core when debugging is started and that caution is required regarding the timing with which inserted breaks will occur. Thus, since the initial operation of the initially stopped core is not actual operation, debugging of normal operation is not possible. Debugging of operation during standby mode is also not possible. For details, refer to the E1/E20 Emulator, E2 Emulator, IE850A Additional Document for User's Manual.

Note: Synchronous debugging is used to all CPU cores run or have a break in execution at the same time in terms of program execution and break generation. Debugging that involves the execution or generation of breaks only in a selected CPU core is called asynchronous debugging.

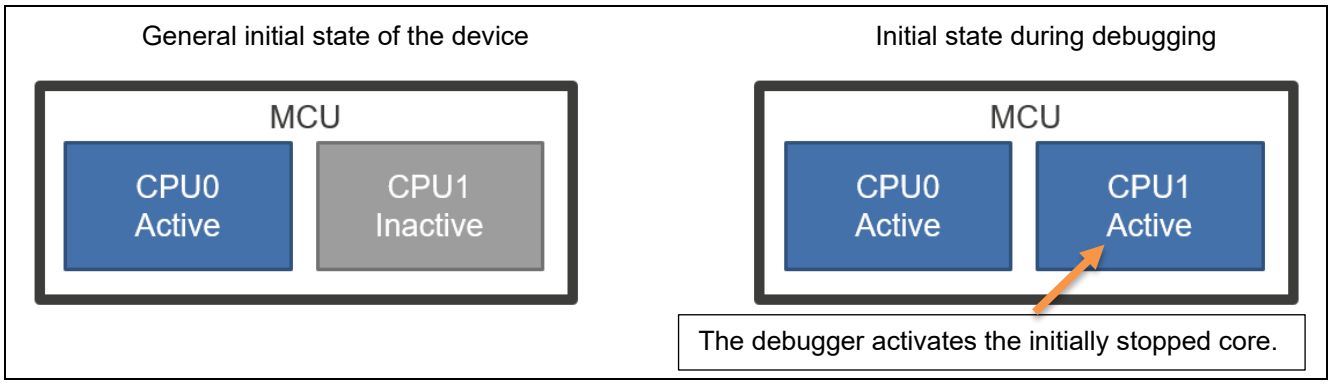

Figure 1-3 Differences between the Initial State of the Device and the Initial State for Synchronous Debugging

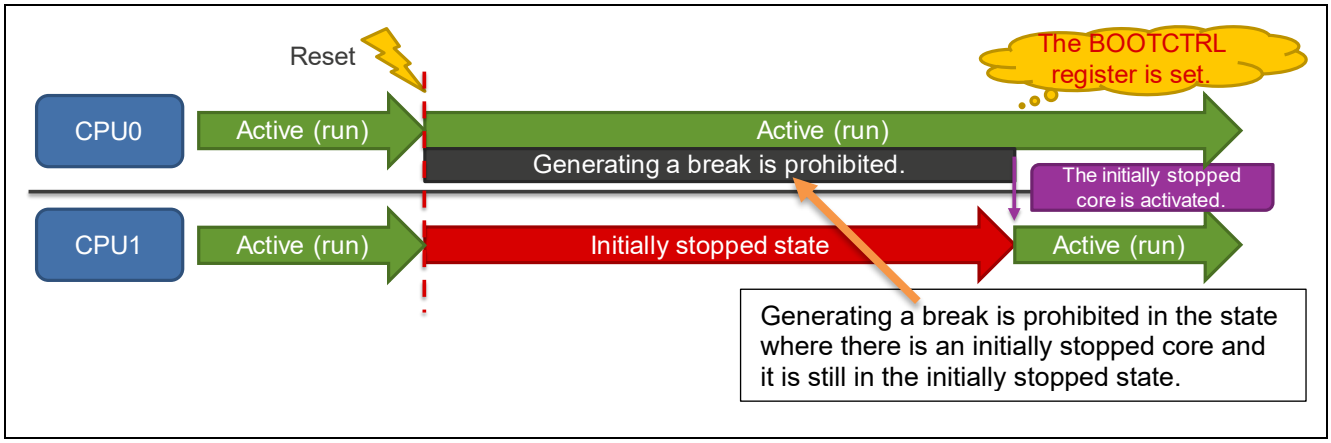

Figure 1-4 Debugging Specifications in the State where an Initially Stopped Core of a Device is in the Initially Stopped State for the Initial Setting of the Debugger

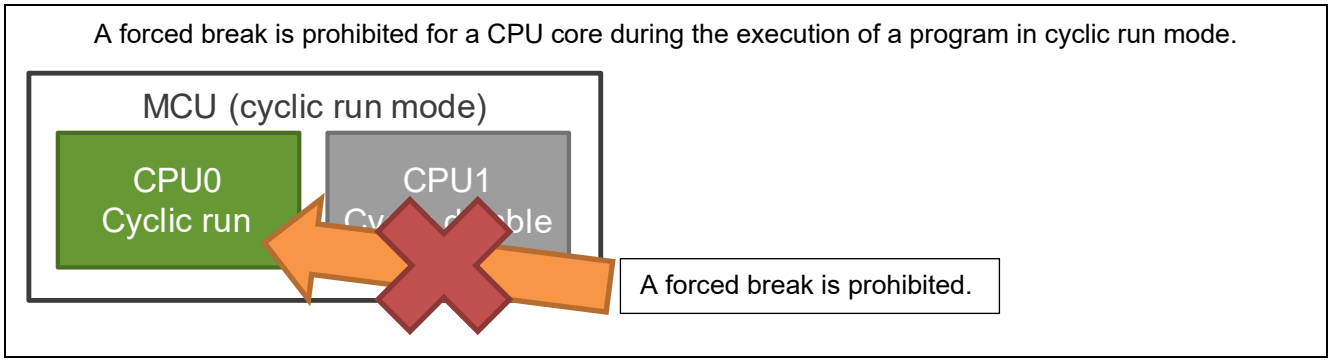

Figure 1-5 Debugging Specifications in Cyclic Run Mode for the Initial Setting of the Debugger

1.2 Debugging Devices with an Initially Stopped Core and Devices in Standby Mode

When the settings are made for a debugger to debug a device that has an initially stopped core or is running a program that includes transitions to standby mode, the user can confirm the current state of the initially stopped core and debug the actual operation of the application. The user can also confirm the standby mode of the current device and debug applications which operate on the device in standby mode.

This application note describes the method of synchronous debugging for users who want to debug applications that run on devices incorporating an initially stopped core or include transitions to standby mode. This application note also describes how to confirm the current state of the CPU core during debugging and start debugging immediately after a state transition with the use of examples of applications and programs which include transitions of the state of the device, confirm that the requirements regarding time restriction have been satisfied. It also lists points for caution on debugging of devices with an initially stopped core and applications that include transitions to standby mode.

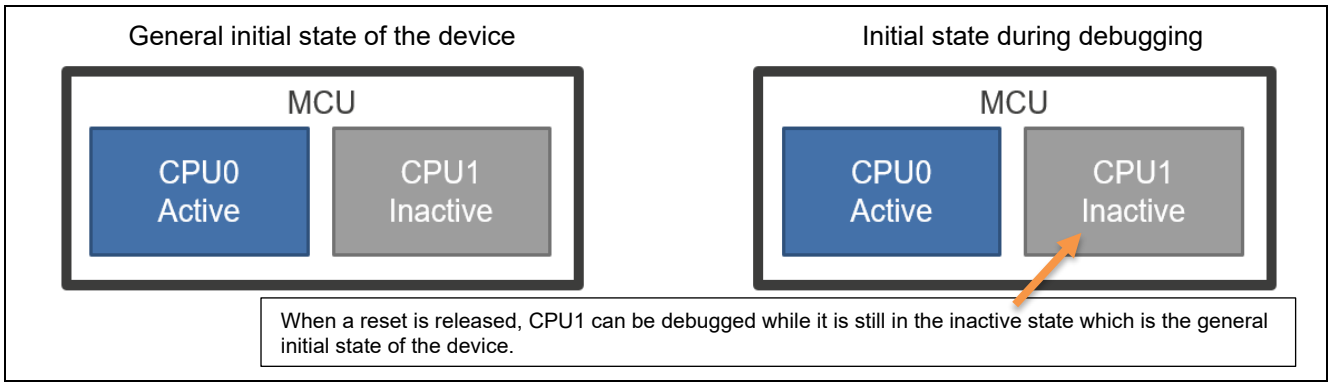

Figure 1-6 Initial State of the Device when a Device with an Initially Stopped Core or Program that Includes Transitions to Standby Mode is to be Debugged

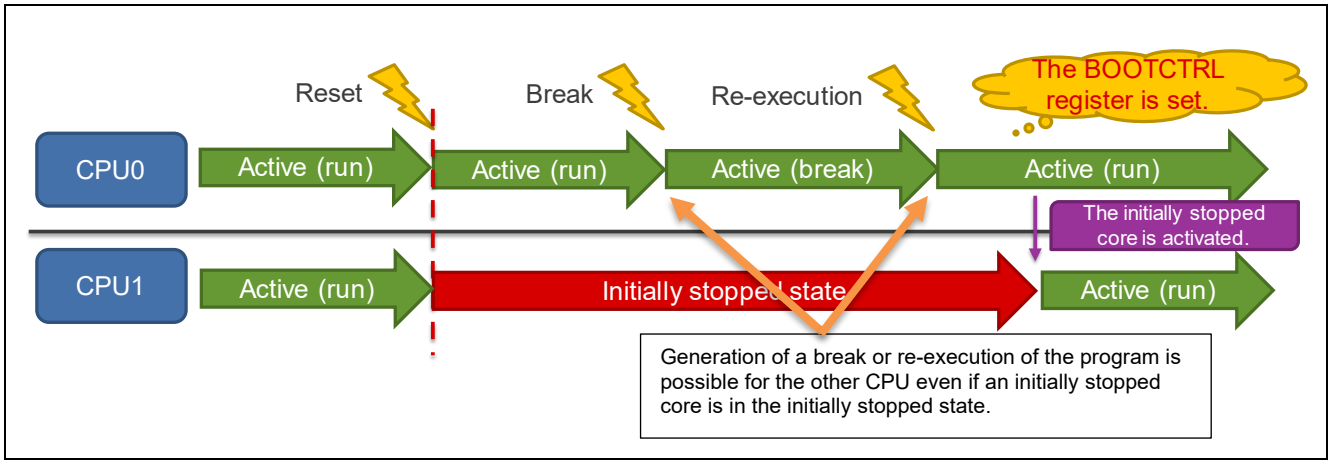

Figure 1-7 Debugging in the State where a CPU Core Enters the Initially Stopped State when Debugging is of a Device with an Initially Stopped Core or a Program that Includes Transitions to Standby Mode

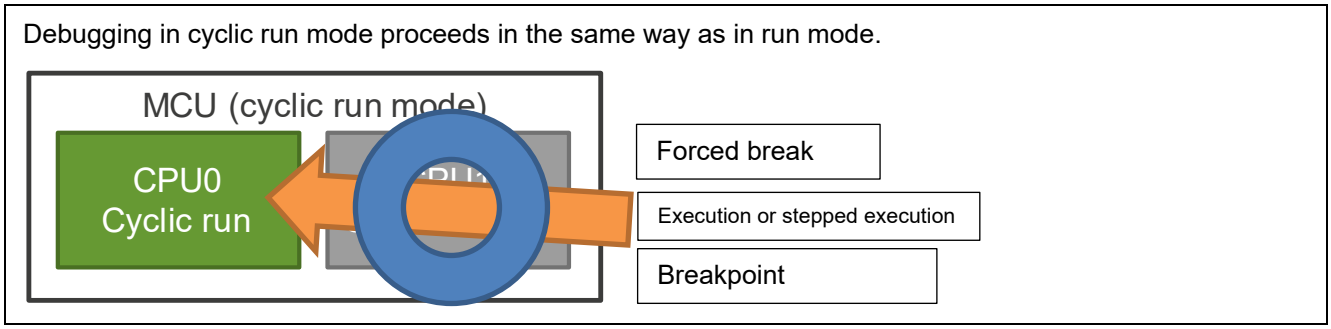

Figure 1-8 Debugging in Cyclic Run Mode when Debugging is of a Device with an Initially Stopped Core or a Program that Includes Transitions to Standby Mode

2. Setting up the Environment

This chapter describes setting up the environment for debugging an application that runs on a device incorporating an initially stopped core or includes transitions to standby mode.

2.1 System Configuration and Required Environment

This section describes the system configuration and required environment.

2.1.1 System Configuration

Figure 2-1 and Figure 2-2 show the system configuration.

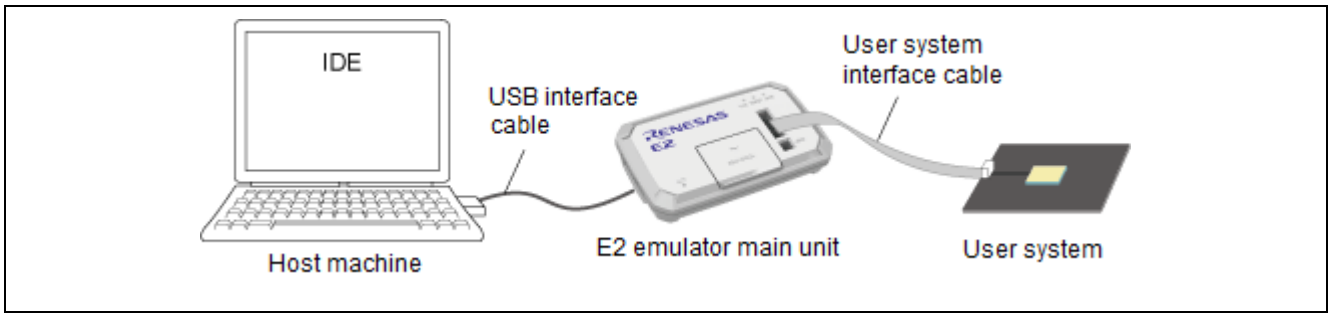

Figure 2-1 System Configuration (E1/E20/E2 Emulator)

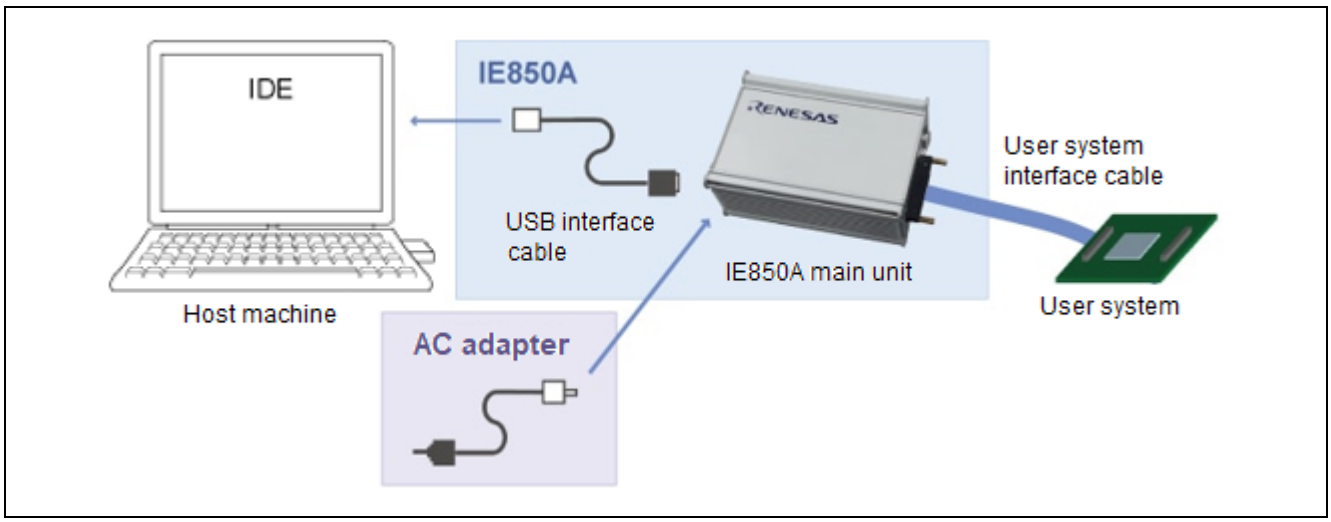

Figure 2-2 System Configuration (IE850A Emulator)

2.1.2 Required Environment

Table 2-1 shows the required environment.

Table 2-1 Required Environment

Item	Detail	
Target device and emulator	[Target devices] RH850/F1KM-S1 RH850/F1KM-S4 RH850/F1KH-D8	[Emulators] E1 emulator E20 emulator E2 emulator
	[Target devices] RH850/E2x RH850/U2A	[Emulators] E2 emulator IE850A emulator
Integrated development environment (and versions)	Renesas CS+	V8.03.00 and later versions
	Green Hills Software MULTI	Installations with 850eserv2 support*

Note: For details, ask Green Hills Software or your local Renesas Electronics sales office or distributor.

2.2 Turning on the Emulator and User System

The following describes how to start the emulator and user system.

(1) Connect the A plug of the USB interface cable to the USB interface connector of the host machine.
(2) Connect the mini-B plug of the USB interface cable to the USB interface connector of the E1/E20/E2/IE850A emulator.
(3) For the E1/E20/E2 emulator, the power of the emulator is turned on by connecting the emulator to the host machine with a USB interface cable.
(4) For the IE850A emulator, connect the AC adapter to the IE850A emulator. Turning on the power switch turns on the emulator.
(5) Turn on the user system.

3. Settings for Debugging Applications

This chapter describes how to set the integrated development environment for synchronous debugging of the actual operation of applications or applications that includes transitions to standby mode on device incorporating an initially stopped core or standby mode.

With this setting, the initially stopped core stays in the initially stopped state following release from a reset and the actual operation can be synchronously debugged. It also allows synchronously debugging of an application that includes transitions to standby mode.

3.1 Setting in CS+

In CS+, select the emulator for the debug tool to be used. Select [Yes] for [Debug the initial stop state and the standby mode] on the [Connect Settings] tabbed page of the [Property] panel for the debug tool.

After the setting was made, select [Build & Download] from the [Debug] menu and start and download the emulator debugger.

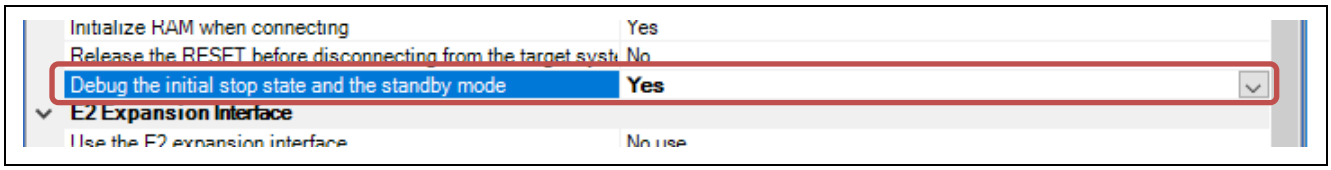

Figure 3-1 Setting for Debugging Devices with Cores in the Initially Stopped State and Operation Including Transitions to Standby Mode in CS+

3.2 Setting in MULTI

In MULTI, specify “-initstop” as an option for starting the emulator debugger.

This option sets up the connection with 850eserv2.

```
connect 850eserv2 -rh850 -e2lpd4=default -initstop -df=.¥DR7F702Z12.DVF ...
```

Figure 3-2 Setting for Debugging Devices with Cores in the Initially Stopped State and Operation Including Transitions to Standby Mode in MULTI

4. Debugging Methods

This chapter describes the methods of synchronous debugging of applications that operate on devices incorporating an initially stopped core and execution that includes transitions to standby mode.

4.1 Debugging Method for Applications Running on Devices Incorporating an Initially Stopped Core

This section describes the method of synchronous debugging of applications that operate on devices incorporating an initially stopped core. The following shows an example of debugging operations for an application.

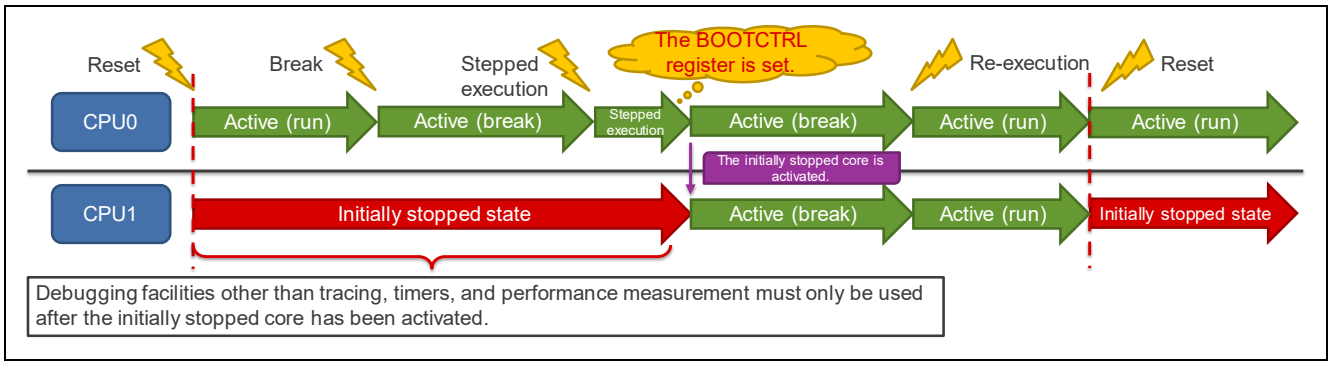

Figure 4-1 Example of Applications that Run on a Device Incorporating an Initially Stopped Core (Example Application 1) and Descriptions of Debugging Operations

The user can confirm whether the initially stopped core is currently in the initially stopped or active state and start debugging immediately after the core has been activated. The user can also confirm the requirement in terms of the time restriction of waiting for activation of the initially stopped core by measuring this time.

If you want to acquire tracing information or measure the execution time of an initially stopped core, specify the facilities related to tracing, timers, and performance measurement before executing the program. Only use the other debugging facilities after the initially stopped core has been activated.

For the example above, the following describes the items of debugging and the names of the relevant sections.

- Confirming that the initially stopped core is in the initially stopped state
  — Refer to section 4.1.1, Confirming that the Initially Stopped Core is in the Initially Stopped State.
- Confirming that the initially stopped core has been activated
- Starting debugging at the point where the initially stopped core has been activated
  — Refer to section 4.1.2, Activating the Initially Stopped Core.
- Checking the time until the initially stopped core is activated after release from a reset
  — Refer to section 4.1.3, Confirming that the Time until the Initially Stopped Core is Activated Satisfies the Requirements in Terms of Time Restrictions.
- Confirming that the initially stopped core enters the initially stopped state after a reset is issued
  — Refer to section 4.1.4, Confirming the Response to the Initially Stopped Core to a Reset.

4.1.1 Confirming that the Initially Stopped Core is in the Initially Stopped State

This section describes how to confirm that the initially stopped core is in the initially stopped state. The user can confirm that the initially stopped core is in the initially stopped state in the following way.

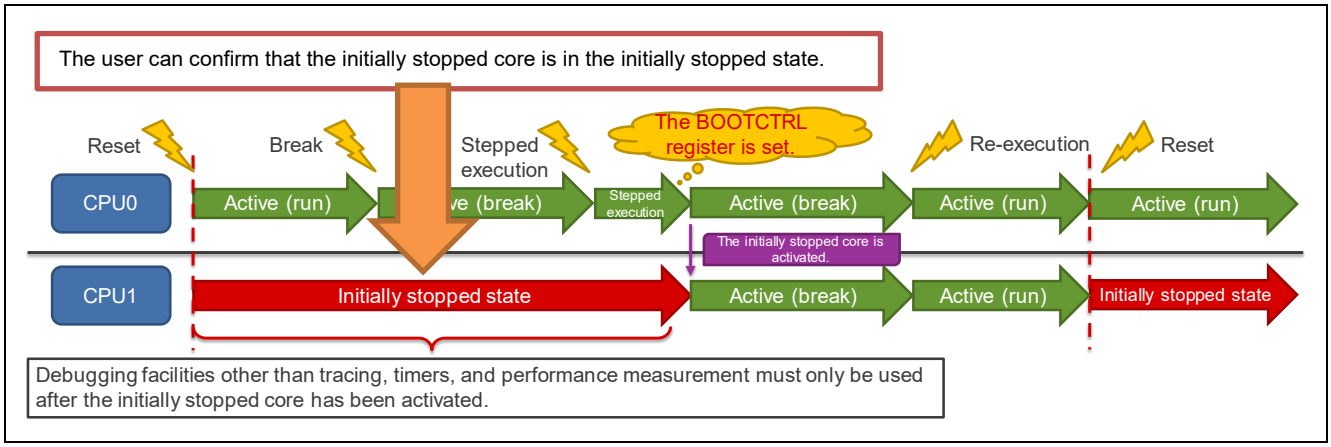

Figure 4-2 Confirming the Initially Stopped State in Example Application 1

The user can confirm that the initially stopped core is in the initially stopped state by using the debugger to acquire this information. The displays in each debugger after the state is identified are shown below.

- CS+
  CS+ shows “Initial Stop” as the state of an initially stopped core in the initially stopped state.
  CS+ shows only the state of the selected CPU core. If you want to confirm the states of the other CPU core, switch the CPU core.

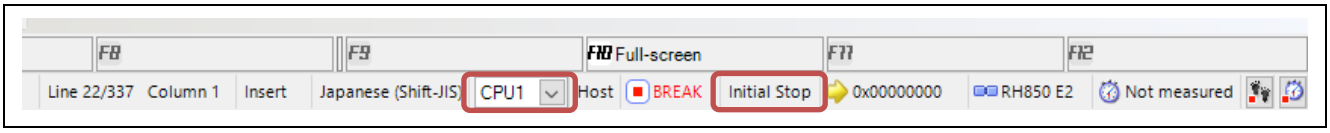

Figure 4-3 Display in CS+ Indicating that an Initially Stopped Core is in the Initially Stopped State

- MULTI
  MULTI shows “0x100” (numerals) and “FETCH-STOP” (a string) to indicate that an initially stopped core is in the initially stopped state.
  In MULTI, issue the cpustatus command to confirm the states of all CPU cores.

```
850eserv2> cpustatus
CPU0 CPU status (0x0):
Core is Stopped, PC=0xXXXX

CPU1 CPU status (0x100): FETCH-STOP
Core is Stopped, PC=0x0

CPU2 CPU status (0x100): FETCH-STOP
Core is Stopped, PC=0x0

CPU3 CPU status (0x100): FETCH-STOP
Core is Stopped, PC=0x0
```

CPU0 is not in the initially stopped state but is active.

Figure 4-4 Display in MULTI Indicating that an Initially Stopped Core is in the Initially Stopped State

Activating the Initially Stopped Core

This section describes how to confirm the state of the device when the initially stopped core has been activated and start debugging the application immediately after having done so. With the method shown below, the user can confirm that the initially stopped core is in the active state and start debugging at that point after having done so.

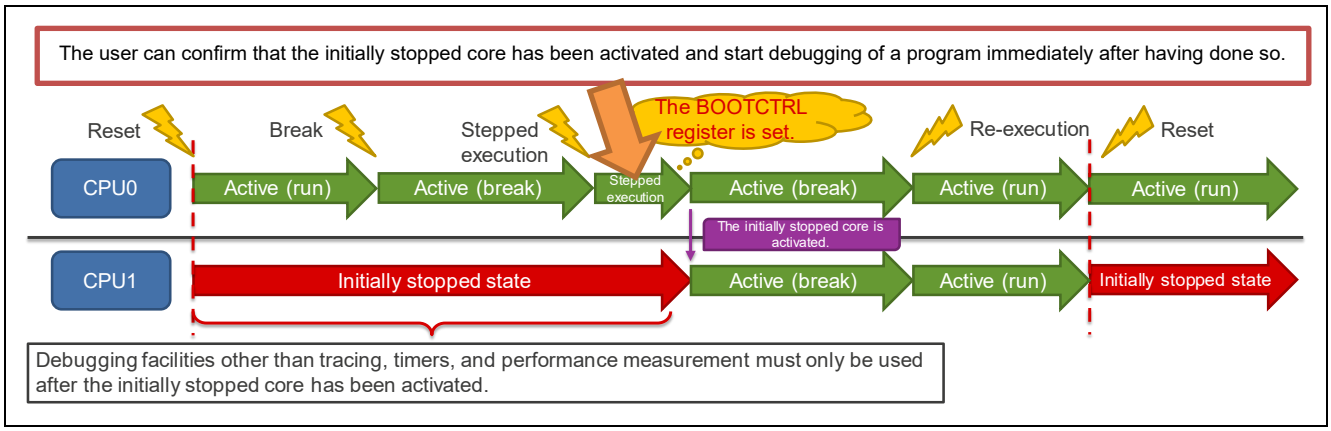

Figure 4-5 Activating the Initially Stopped Core in Example Application 1

Figure 4-6 shows an example of a program that activates the initially stopped core. Execution of this program by CPU0 in the example of Figure 4-5 activates the initially stopped core. For details on registers and programming, refer to the hardware manual for the device.

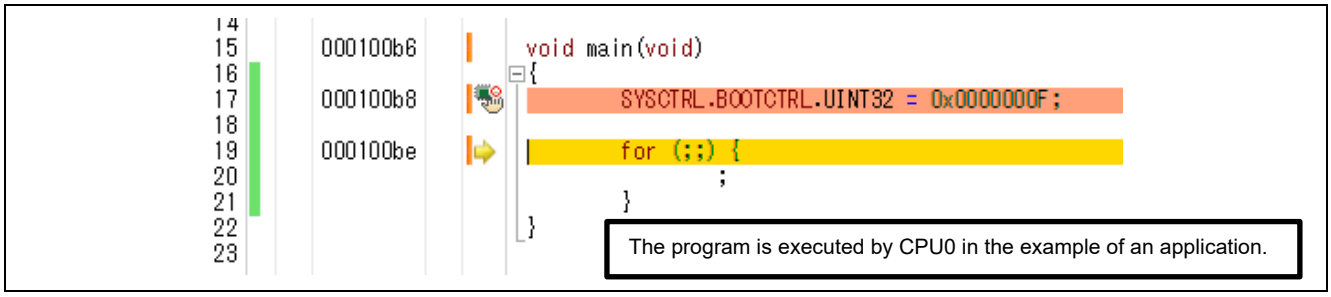

Figure 4-6 Example of a Program for Activating the Initially Stopped Core

When you want to start debugging at the point where the initially stopped core has been activated, change the selected CPU core to the initially stopped core that has been activated after step-executing programming of the BOOTCTRL register of CPU0 with the above example of a program.

The displays in each debugger after the state is identified are shown below.

The indicators of the state when the initially stopped core has been activated are the same as those for a non-initially stopped CPU core that is active.

- CS+

In CS+, the indicator of the state when the initially stopped core has been activated and selected is the same as that for a non-initially stopped CPU core that is active.

CS+ shows only the state of the selected CPU core. If you want to confirm the states of the other CPU core, switch the CPU core.

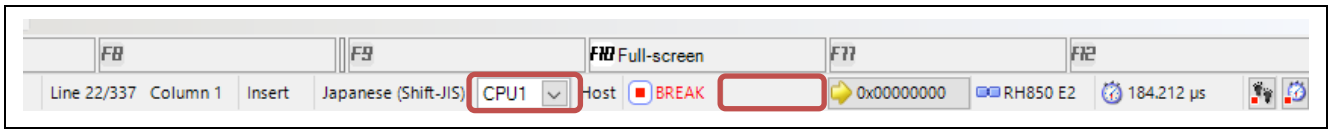

Figure 4-7 Displaying the State in CS+ of the Initially Stopped Core having been Activated so that it is Operating

- MULTI

In MULTI, the part of the list of states when the initially stopped cores have been activated and selected is the same as that for a non-initially stopped CPU core that is active.

In MULTI, issue the cpustatus command to confirm the states of all CPU cores.

```
850eserv2> cpustatus
CPU0 CPU status (0x0):
Core is Stopped, PC=0xXXXX

CPU1 CPU status (0x0):
Core is Running

CPU2 CPU status (0x0):
Core is Running

CPU3 CPU status (0x0):
Core is Running
```

CPU0 is also active.

Figure 4-8 Displaying the State in MULTI with the Initially Stopped Cores Activated and Operating

4.1.2 Confirming that the Time until the Initially Stopped Core is Activated Satisfies the Requirements in Terms of Time Restrictions

This section describes how to measure the time from CPU0 being activated following release from a reset until the program run by CPU0 activates the initially stopped core, which starts executing a program. Through the following methods, the user can confirm that the time until the initially stopped core is activated after the device has been activated satisfies the requirements in terms of time restrictions.

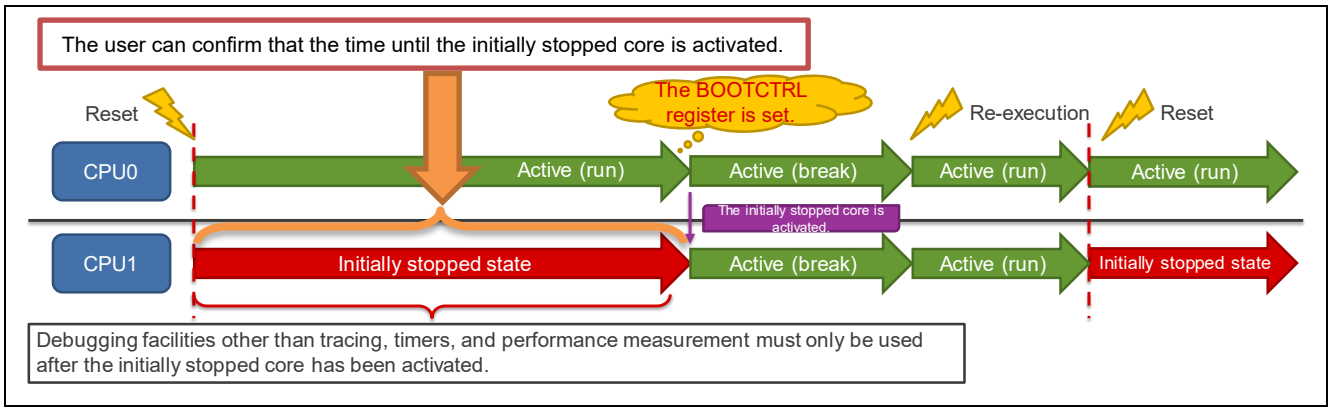

Figure 4-9 Time until the Initially Stopped Core is Activated in Example Application 1

Using the timer facility of the debugger, specify the condition for the start of time measurement as the reset vector address of CPU0 or the start of the execution of a program from the reset break state, and specify the end of time measurement as being immediately after programming of the BOOTCTRL register. Executing the program after the jump to the reset vector address of CPU0 or the start of the execution of a program by CPU0 allows measuring the time by which the initially stopped core is activated after release from a reset.

If the time until the initially stopped core is activated does not satisfy the requirements in terms of time restrictions, the user can confirm the times taken by the parts of the program until the initially stopped core is activated by measuring the times from the start to end of time measurement or using the display from tracing of times taken for processing. This clarifies how long each part of processing by the program is taking.

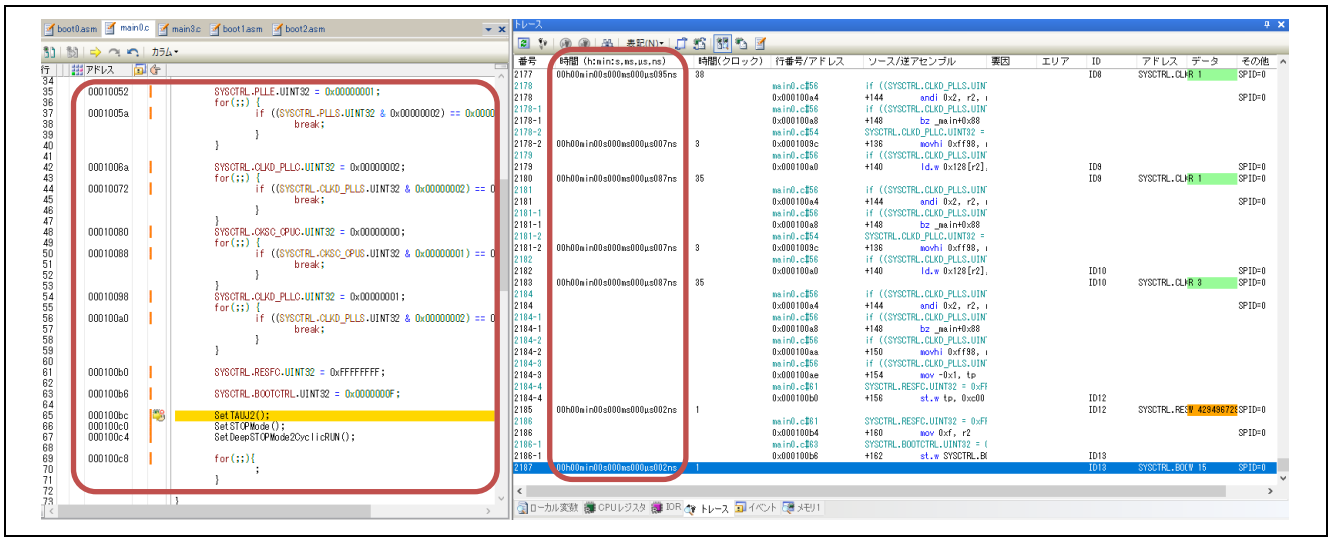

Figure 4-10 Result of Tracing a Program in CS+ until the Initially Stopped Core is Activated

4.1.3 Confirming the Response to the Initially Stopped Core to a Reset

When the user or program applies a reset while the initially stopped core is active, the initially stopped core returns to its initially stopped state.

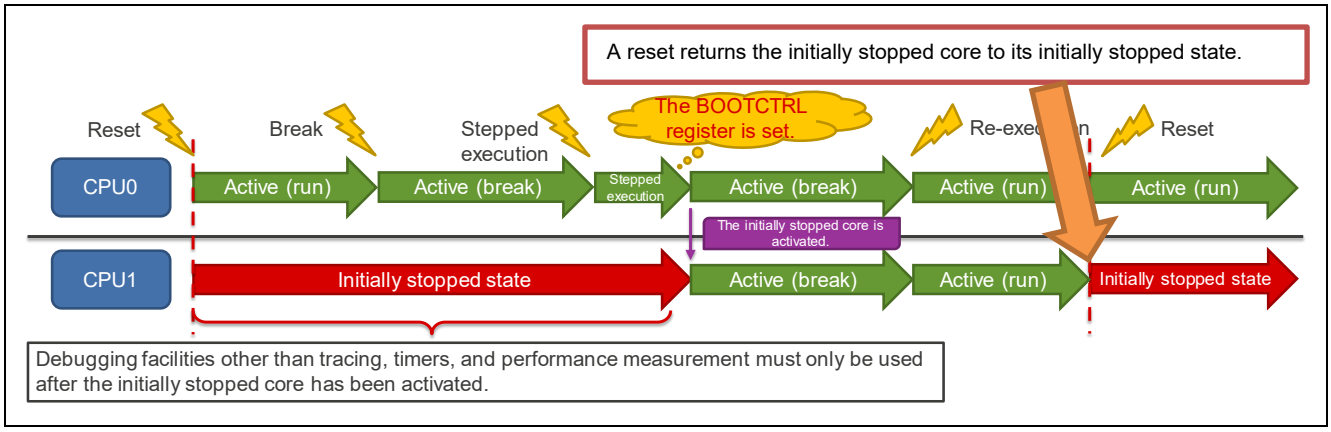

Figure 4-11 The State after a Reset in Example Application 1

For details on the display of states, refer to section 4.1.1, Confirming that the Initially Stopped Core is in the Initially Stopped State.

4.2 Debugging Method for Applications that Include Transitions to Standby Modes

This section describes the method of synchronous debugging of applications that in the case of transitions to standby modes on devices that incorporating them. The following shows an example of debugging operations for an application.

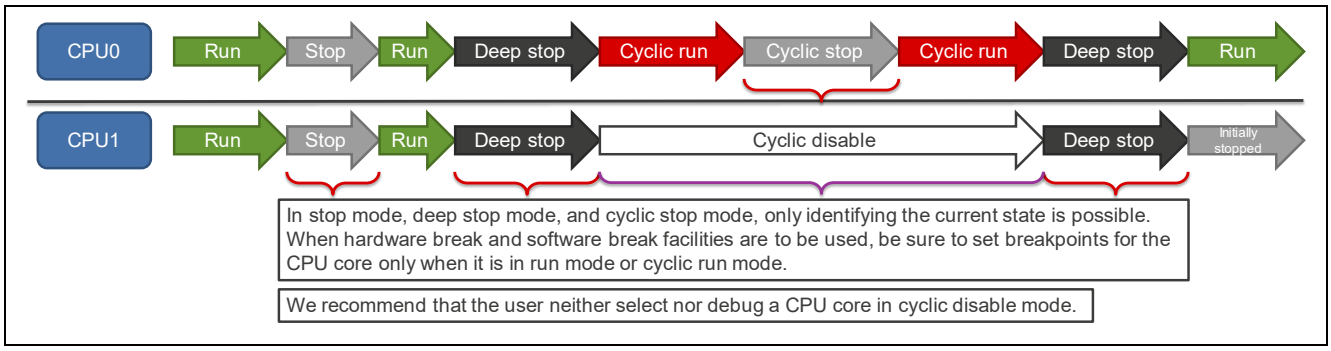

Figure 4-12 Example of Applications that Include Transitions to Standby Modes (Example Application 2) and Descriptions of Debugging Operations

The user can confirm the current standby mode. Debugging can start immediately after transitions from stop mode, deep stop mode, or cyclic stop mode to run mode or cyclic run mode. In addition, the user can also confirm that requirements in terms of time restrictions have been satisfied by measuring times over which the cores of the device are in a standby mode.

As is the case in run mode, the user can start debugging when the device is in cyclic run mode, but flash memory is not accessible.

When the device is in stop mode, deep stop mode, or cyclic stop mode, the user can only acquire state information. No other debugging facilities are available. When using hardware break and software break facilities, be sure to set breakpoints for a CPU core only when it is in run mode or cyclic run mode.

A CPU core in cyclic disable mode does not operate; the CPU core only activates when it returns to run mode through deep stop mode. We recommend that the user neither select nor debug a CPU core in cyclic disable mode.

For the example above, the following describes the items of debugging and the names of the relevant sections.

- Confirming that the device has made the transition to stop mode
- Starting debugging at the point where the device has made the transition to run mode from stop mode
- Checking the times over which the device is in stop mode
  - Refer to sections 4.2.1, Stop Mode, 4.2.1.1, Starting Debugging Immediately after the Device has Made the Transition to Run Mode from Stop Mode, and 4.2.1.2, Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Stop Mode.

- Confirming that the device has made the transition to deep stop mode
- Starting debugging at the point where the device has made the transition to run mode from deep stop mode
- Checking the times from the device entering deep stop mode until the transition to run mode
  - Refer to sections 4.2.2, Deep Stop Mode, 4.2.2.1, Starting Debugging Immediately after the Device has Made the Transition to Run Mode from Deep Stop Mode, and 4.2.2.2, Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Deep Stop Mode at the Time of the Transition to Run Mode from Deep Stop Mode.

- Confirming that the device has made the transition to cyclic run mode
- Starting debugging at the point where the device has made the transition to cyclic run mode from deep stop mode
- Checking the times from the device entering deep stop mode until the transition to cyclic run mode
- Checking the times over which the device is in cyclic run mode
  - Refer to sections 4.2.3, Cyclic Run Mode, 4.2.3.1, Starting Debugging Immediately after the Device has Made the Transition to Cyclic Run Mode from Deep Stop Mode, 4.2.3.2, Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Deep Stop Mode at the Time of the Transition to Cyclic Run Mode from Deep Stop Mode, and 4.2.3.3, Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Cyclic Run Mode.

- Confirming that the device has made the transition to cyclic stop mode
- Starting debugging at the point where the device has made the transition to cyclic run mode from cyclic stop mode
- Checking the times over which the device is in cyclic stop mode
  - Refer to sections 4.2.4, Cyclic Stop Mode, 4.2.4.1, Starting Debugging Immediately after the Device has Made the Transition to Cyclic Run Mode from Cyclic Stop Mode, and 4.2.4.2, Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Cyclic Stop Mode.

4.2.1 Stop Mode

This section describes how to cause a transition of a device to stop mode and confirm that the device is in stop mode. The user can confirm that the device is in stop mode in the following way.

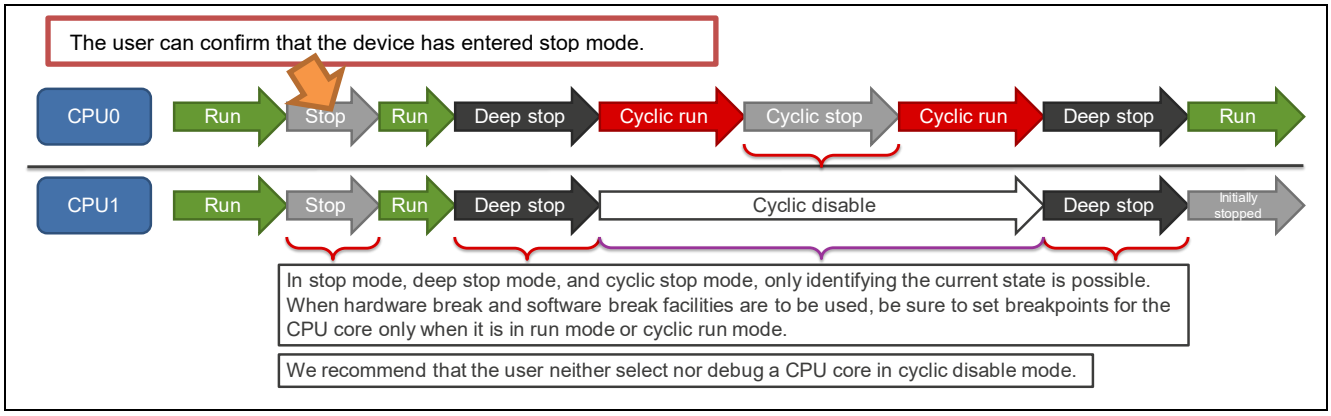

Figure 4-13 Confirming Stop Mode in Example Application 2

Figure 4-14 shows an example of a program that causes a transition to stop mode. When CPU0 in the example of an application shown in Figure 4-13 executes this program, the device enters stop mode. For details on registers and programming, refer to the hardware manual for the device.

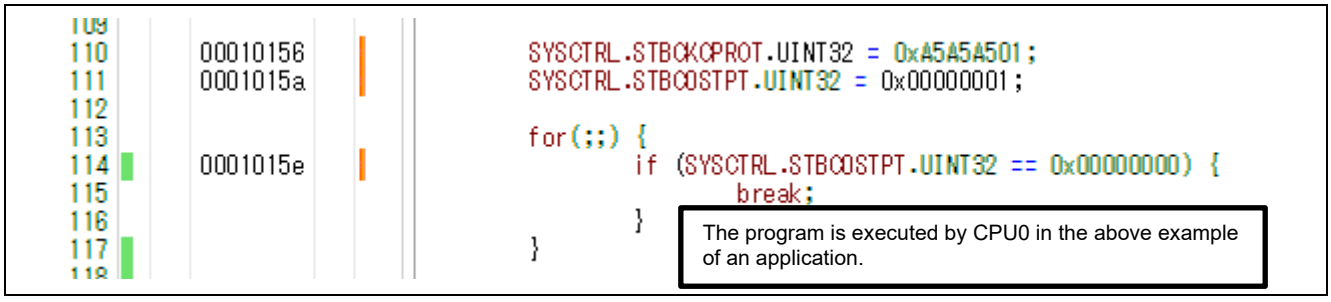

Figure 4-14 Example of a Program that Causes a Transition to Stop Mode

The user can confirm that the device is in stop mode by identifying the state. The displays in each debugger after the state is identified are shown below.

- CS+
  CS+ shows “Stop” as the state to indicate that the device is in stop mode.

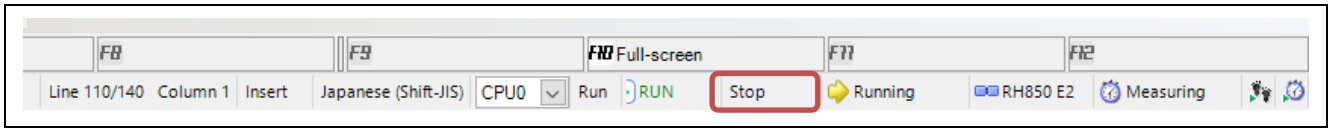

Figure 4-15 Displaying the State of the Device as Stop Mode in CS+

- MULTI
  MULTI shows “0x8” (numerals) and “HARDWARE STOP” (a string) to indicate that the state of the device is stop mode. Since the device is in this standby mode, the indicator for stop mode is also displayed for CPU0.

```
850eserv2> cpustatus
CPU0 CPU status (0x8): HARDWARE STOP
Core is Running

CPU1 CPU status (0x8): HARDWARE STOP
Core is Running

CPU2 CPU status (0x8): HARDWARE STOP
Core is Running

CPU3 CPU status (0x8): HARDWARE STOP
Core is Running
```

Figure 4-16 Displaying the State of the Device as Stop Mode in MULTI

4.2.1.1 Starting Debugging Immediately after the Device has Made the Transition to Run Mode from Stop Mode

This section describes how to generate a break in the CPU core immediately after the device has entered run mode due to wakeup resources after having been in stop mode. With the method shown below, the user can confirm the state of the device immediately after it has made the transition to run mode from stop mode and start debugging of an application immediately after the device enters run mode.

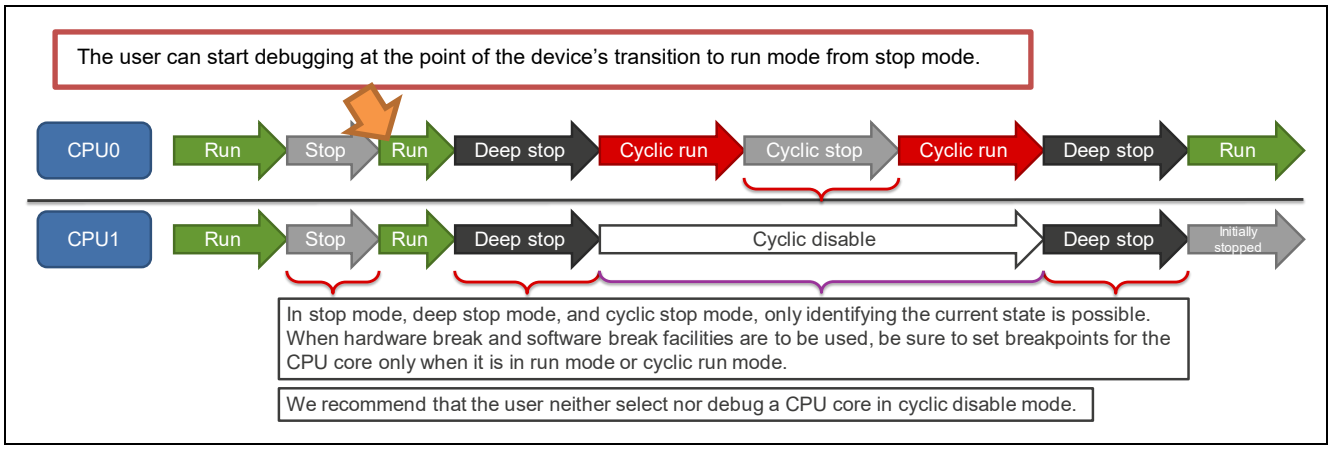

Figure 4-17 Transition to Run Mode from Stop Mode in Example Application 2

Figure 4-18 shows an example of the debugging of a program where a break occurs at the point of the CPU core's transition to run mode from stop mode. When the CPU core has made the transition to run mode from stop mode, the program is executed after checking the value of the STBC0STPT register. To start debugging at the point where the CPU core has entered run mode, set a breakpoint after checking the STBC0STPT register before the CPU core enters stop mode. This enables the generation of a break at the point where the CPU core has entered run mode.

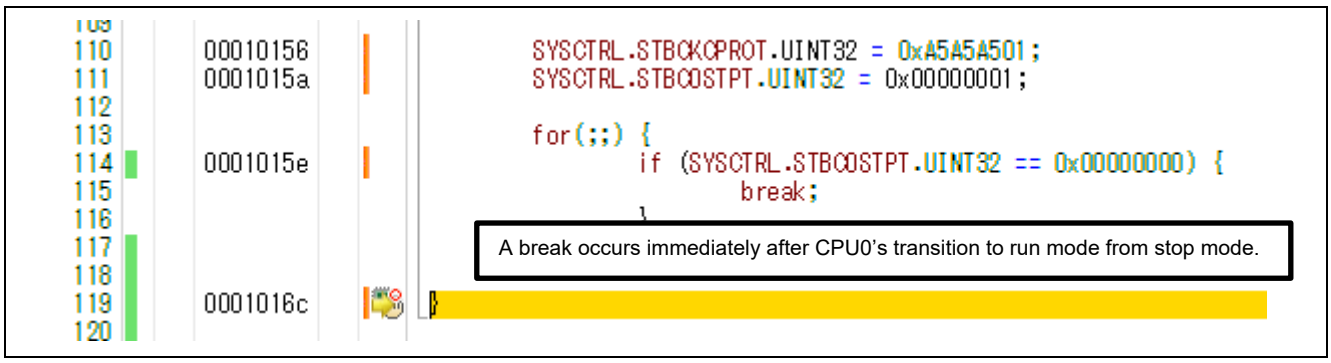

Figure 4-18 Example of a Break Following the Device's Transition to Run Mode from Stop Mode

4.2.1.2 Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Stop Mode

This section describes how to measure the time the device is in stop mode before returning to run mode. With the method shown below, the user can confirm that the requirements in terms of restrictions on time in stop mode have been satisfied.

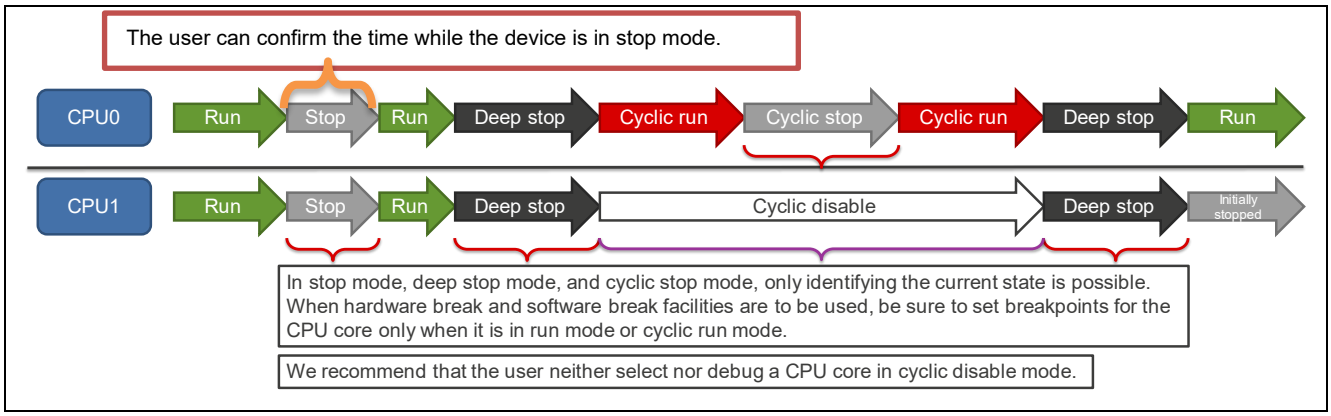

Figure 4-19 Confirming the Time in Stop Mode in Example Application 2

Use the timer facility of the debugger to specify the start of time measurement as the program address that causes the transition to stop mode and the end of time measurement as the program address immediately after the device has returned to run mode from stop mode. Executing the program will provide a measurement of the time from the program on CPU0 placing the device in stop mode until the device enters run mode.

4.2.2 Deep Stop Mode

This section describes how to cause a transition of a device to deep stop mode and confirm that the device is in deep stop mode. The user can confirm that the device is in deep stop mode in the following way.

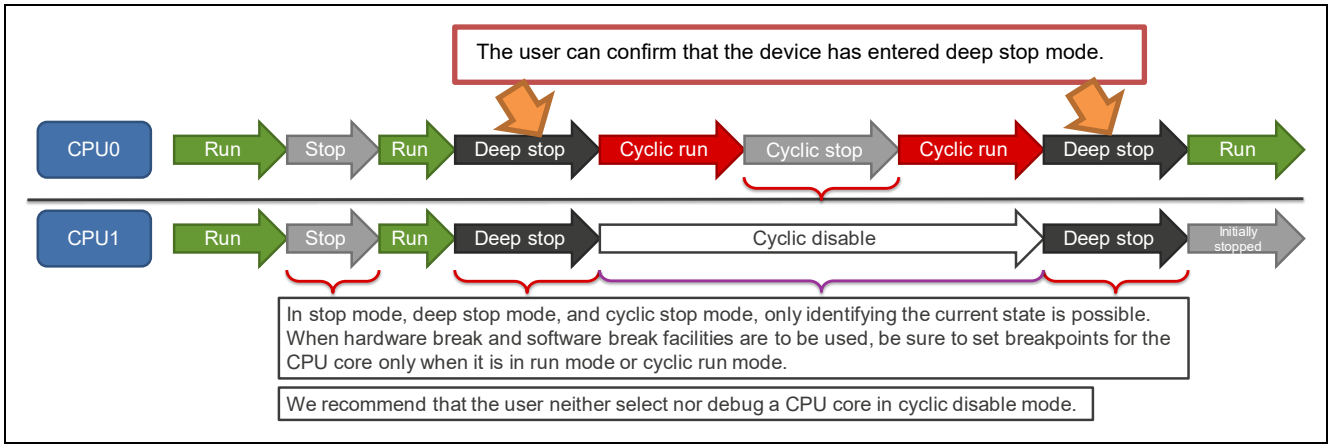

Figure 4-20 Confirming Deep Stop Mode in Example Application 2

Figure 4-21 shows an example of a program that causes a transition to deep stop mode. When CPU0 in the example of an application shown in Figure 4-20 executes this program, the device enters deep stop mode. For details on registers and programming, refer to the hardware manual for the device.

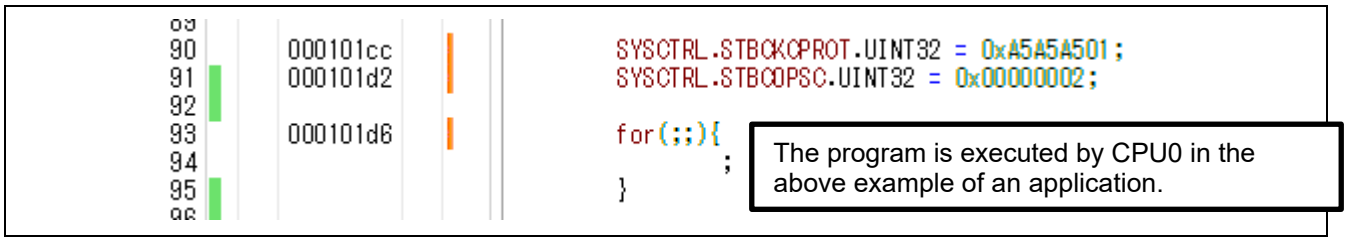

Figure 4-21 Example of a Program that Causes a Transition to Deep Stop Mode

The user can confirm that the device is in deep stop mode by identifying the state. The displays in each debugger after the state is identified are shown below.

- CS+
  CS+ shows “Deep Stop” as the state to indicate that the device is in deep stop mode.

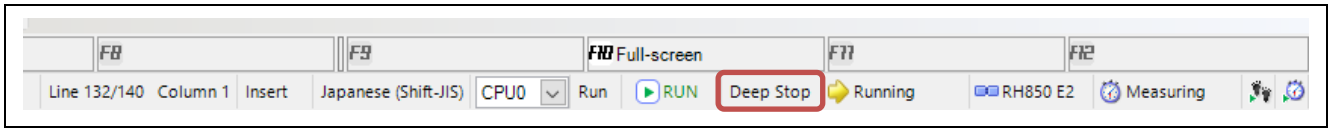

Figure 4-22 Displaying the State of the Device as Deep Stop Mode in CS+

- MULTI
  MULTI shows “0x200” (numerals) and “DEEP-STOP” (a string) to indicate that the state of the device is deep stop mode. Since the device is in this standby mode, the indicator for deep stop mode is also displayed for CPU0.

```
850eserv2> cpustatus
CPU0 CPU status (0x200): DEEP-STOP
Core is Running

CPU1 CPU status (0x200): DEEP-STOP
Core is Running

CPU2 CPU status (0x200): DEEP-STOP
Core is Running

CPU3 CPU status (0x200): DEEP-STOP
Core is Running
```

Figure 4-23 Displaying the State of the Device as Deep Stop Mode in MULTI

4.2.2.1 Starting Debugging Immediately after the Device has Made the Transition to Run Mode from Deep Stop Mode

This section describes how to generate a break in the CPU core immediately after the device has entered run mode due to wakeup resources after having been in deep stop mode. With the method shown below, the user can confirm the state of the device immediately after it has made the transition to run mode from deep stop mode and start debugging of an application immediately after the device enters run mode.

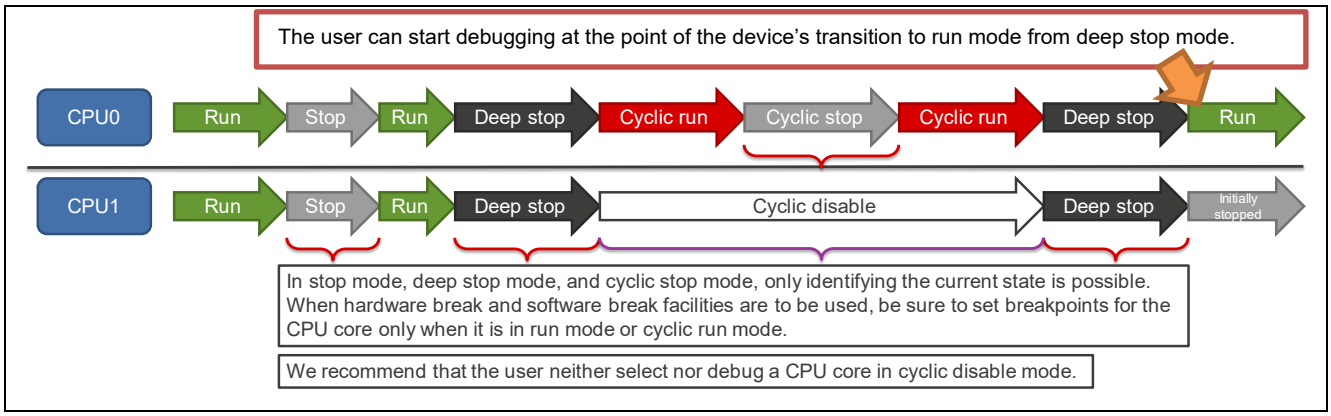

Figure 4-24 Transition to Run Mode from Deep Stop Mode in Example Application 2

Figure 4-25 shows an example of the debugging of a program where a break occurs at the point of the CPU core's transition to run mode from deep stop mode. When the CPU core has made the transition to run mode from deep stop mode, the reset that was applied to initiate release from deep stop mode leads to the start of program execution from the reset vector address. The reset vector address depends on RBASE. In the example of debugging of the program, RBASE is set to 0x00000000. To start debugging at the point where the CPU core has entered run mode, set a breakpoint at the reset vector address before the CPU core enters deep stop mode. This enables the generation of a break at the point where the CPU core has entered run mode.

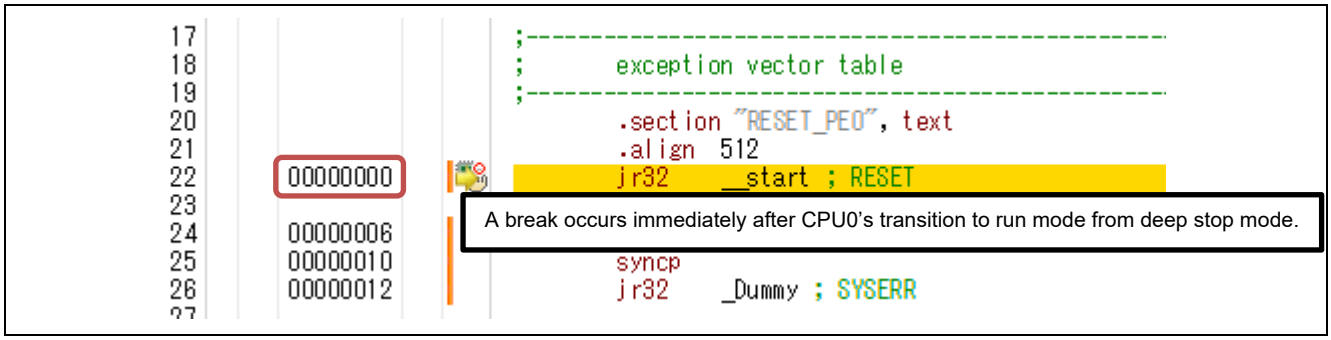

Figure 4-25 Example of a Break Following the Device's Transition to Run Mode from Deep Stop Mode

The deep stop reset generated by the transition of the device to run mode from deep stop mode makes the initially stopped core enter the initially stopped state. For the debugging of an initially stopped core, refer to section 4.1, Debugging Method for Applications Running on Devices Incorporating an Initially Stopped Core.

4.2.2.2 Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Deep Stop Mode at the Time of the Transition to Run Mode from Deep Stop Mode

This section describes how to measure the time the device is in deep stop mode before returning to run mode. With the method shown below, the user can confirm that the requirements in terms of restrictions on time in deep stop mode have been satisfied by the time of the device's return to run mode.

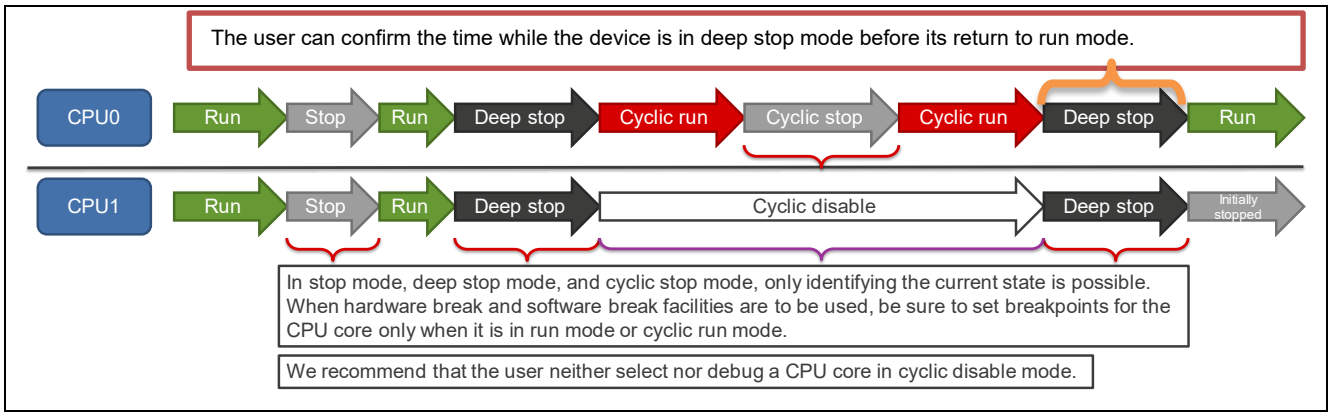

Figure 4-26 Confirming the Time in Deep Stop Mode until the Device Enters Run Mode in Example Application 2

Use the timer facility of the debugger to specify the start of time measurement as the program address that causes the transition to deep stop mode and the end of time measurement as the reset vector address immediately after the device has returned to run mode from deep stop mode. Executing the program will provide a measurement of the time from the program on CPU0 placing the device in deep stop mode until the device enters run mode.

4.2.3 Cyclic Run Mode

This section describes how to cause a transition of a device to cyclic run mode and confirm that the device is in cyclic run mode. The user can confirm that the device is in cyclic run mode in the following way. In addition, as is the case in run mode, the user can start debugging when the device is in cyclic run mode, but flash memory is not accessible.

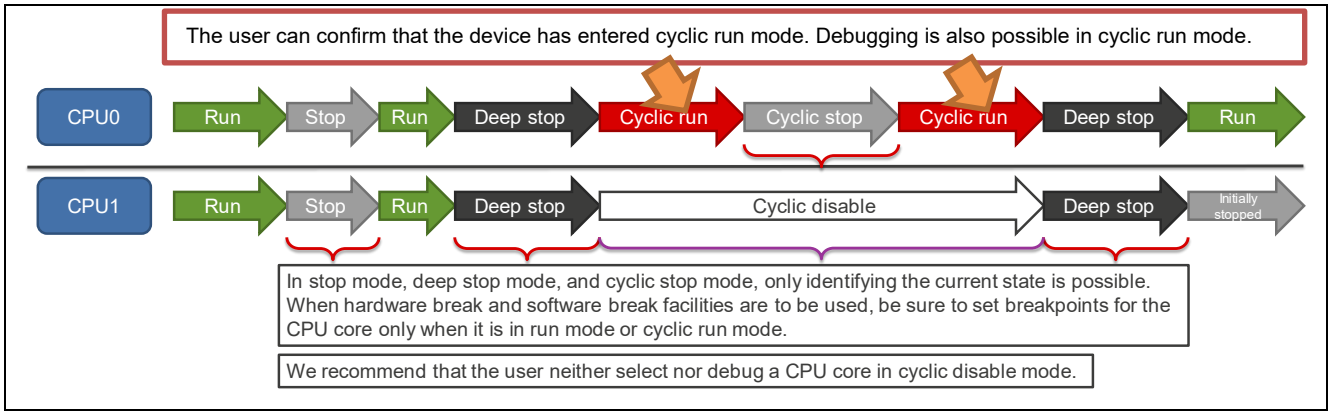

Figure 4-27 Confirming Cyclic Run Mode in Example Application 2

Figure 4-28 shows an example of a program that causes a transition to cyclic run mode. In this example, an interrupt from TAUJ2 is specified as the wakeup resource. When CPU0 in the example of an application shown in Figure 4-27 executes this program, the device enters deep stop mode, the interrupt from TAUJ2 specified as the wakeup resource is issued, and the device enters cyclic run mode. For details on registers and programming, refer to the hardware manual for the device.

When the device enters cyclic run mode, CPU1 is activated in RH850/F1KM-S1 series, F1KM-S4 series, and F1KH-D8 series devices. Only CPU0 is activated in other RH850 family devices. The activated CPU core starts executing code from the first address of the retention RAM. When the CPU core is to operate in cyclic run mode, download the program to the retention RAM in advance. For details on downloading programs to the retention RAM, refer to the user's manual and help system for the emulator debugger.

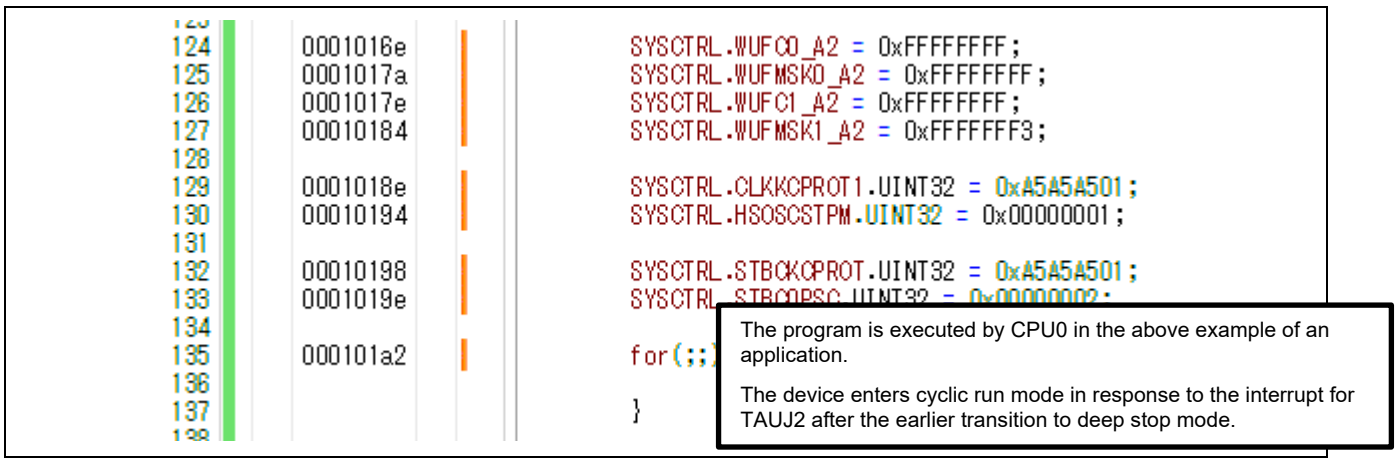

Figure 4-28 Example of a Program that Causes a Transition to Cyclic Run Mode

The user can confirm that the device is in cyclic run mode by identifying the state. The displays in each debugger after the state is identified are shown below.

- CS+
  CS+ shows “Cyclic RUN” as the state to indicate that the device is in cyclic run mode.

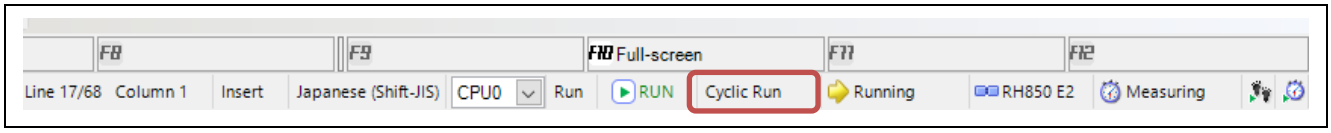

Figure 4-29 Displaying the State of the Device as Cyclic Run Mode in CS+

- MULTI
  MULTI shows “0x400” (numerals) and “CYCLE-RUN” (a string) to indicate that the state of the device is cyclic run mode. Since the device is in this standby mode, the indicator for cyclic run mode is also displayed for CPU0.

```
850eserv2> cpustatus
CPU0 CPU status (0x400): CYCLE-RUN
Core is Running

CPU1 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running

CPU2 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running

CPU3 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running
```

Figure 4-30 Displaying the State of the Device as Cyclic Run Mode in MULTI

4.2.3.1 Starting Debugging Immediately after the Device has Made the Transition to Cyclic Run Mode from Deep Stop Mode

This section describes how to generate a break in the CPU core immediately after the device has entered cyclic run mode due to wakeup resources after having been in deep stop mode. With the method shown below, the user can confirm the state of the device immediately after it has made the transition to cyclic run mode from deep stop mode and start debugging of an application immediately after the device enters cyclic run mode.

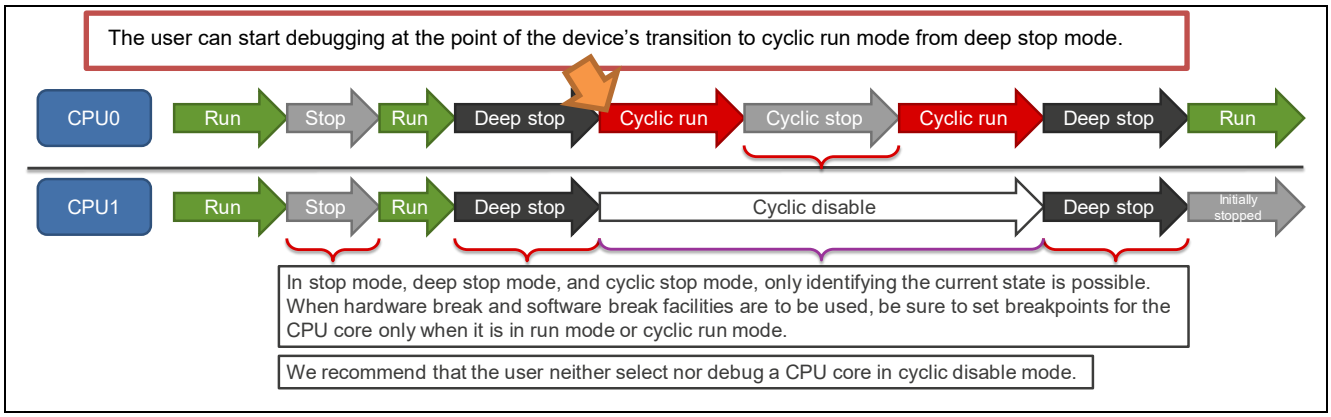

Figure 4-31 Transition to Cyclic Run Mode from Deep Stop Mode in Example Application 2

Figure 4-32 shows an example of the debugging of a program where a break occurs at the point of the CPU core's transition to cyclic run mode from deep stop mode. When the CPU core has made the transition to cyclic run mode from deep stop mode, the program is executed at the first address of the retention RAM. To start debugging at the point where the CPU core has entered cyclic run mode, set a breakpoint at the first address of the retention RAM before the CPU core enters deep stop mode. This enables the generation of a break at the point where the CPU core has entered cyclic run mode.

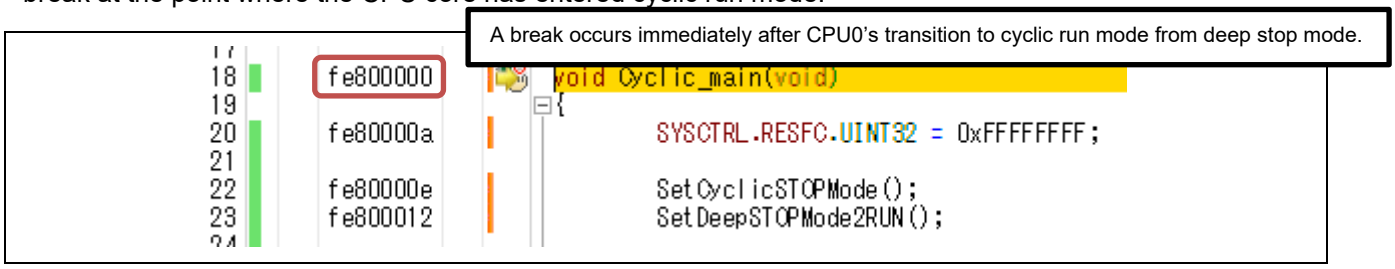

Figure 4-32 Example of a Break Following the Device's Transition to Cyclic Run Mode from Deep Stop Mode

The debuggers show that the device is in cyclic run mode and that a break occurred when the state was identified.

- CS+
  In CS+, the state indicator changes from “RUN” to “BREAK”. The display of “Cyclic RUN” does not change. The PC value is the first address of the retention RAM.

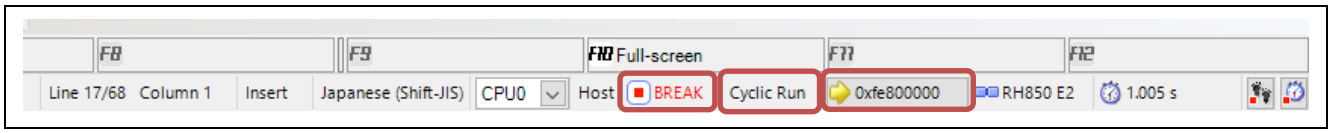

Figure 4-33 Display for the State having Changed to “BREAK” in Cyclic Run Mode for CS+

- MULTI
  In MULTI, the state indicator changes from “Core is Running” to “Core is Stopped”. The display of “CYCLE-RUN” does not change. The PC value is the first address of the retention RAM.

```
850eserv2> cpustatus
CPU0 CPU status (0x400): CYCLE-RUN
Core is Stopped, PC=0xfe800000

CPU1 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running

CPU2 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running

CPU3 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running
```

Figure 4-34 Display for the State having Changed to “Core is Stopped” in Cyclic Run Mode for MULTI

4.2.3.2 Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Deep Stop Mode at the Time of the Transition to Cyclic Run Mode from Deep Stop Mode

This section describes how to measure the time the device is in deep stop mode from cyclic run mode. With the method shown below, the user can confirm that the requirements in terms of restrictions on time in deep stop mode have been satisfied by the time of the device's return to cyclic run mode.

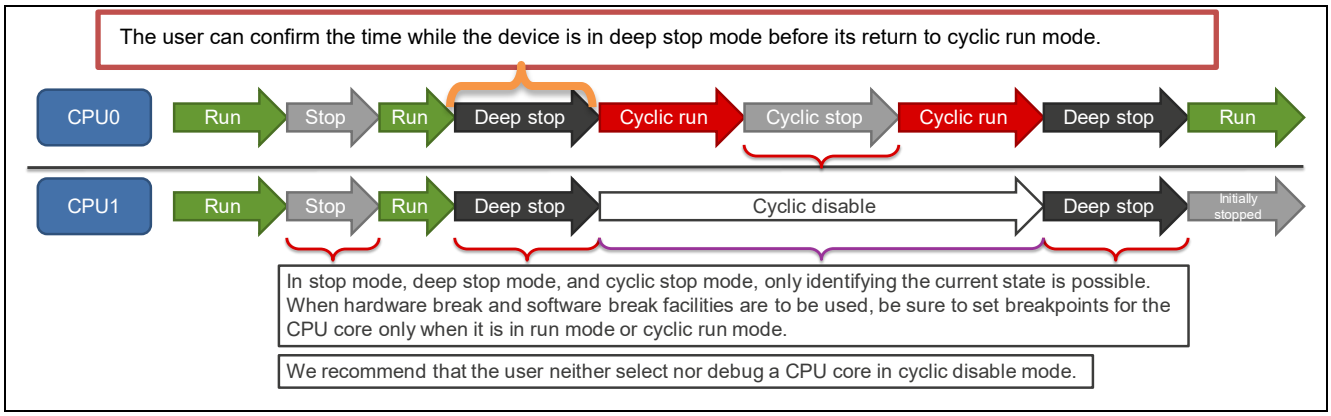

Figure 4-35 Confirming the Time in Deep Stop Mode until the Device Enters Cyclic Run Mode in Example Application 2

Use the timer facility of the debugger to specify the start of time measurement as the program address that causes the transition to deep stop mode and the end of time measurement as the first address of the retention RAM immediately after the device has returned to cyclic run mode from deep stop mode. Executing the program will provide a measurement of the time from the program on CPU0 placing the device in deep stop mode until the device enters cyclic run mode.

4.2.3.3 Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Cyclic Run Mode

This section describes how to measure the execution time of a program in cyclic run mode. With the method shown below, the user can confirm that the requirements in terms of restrictions on time in cyclic run mode have been satisfied.

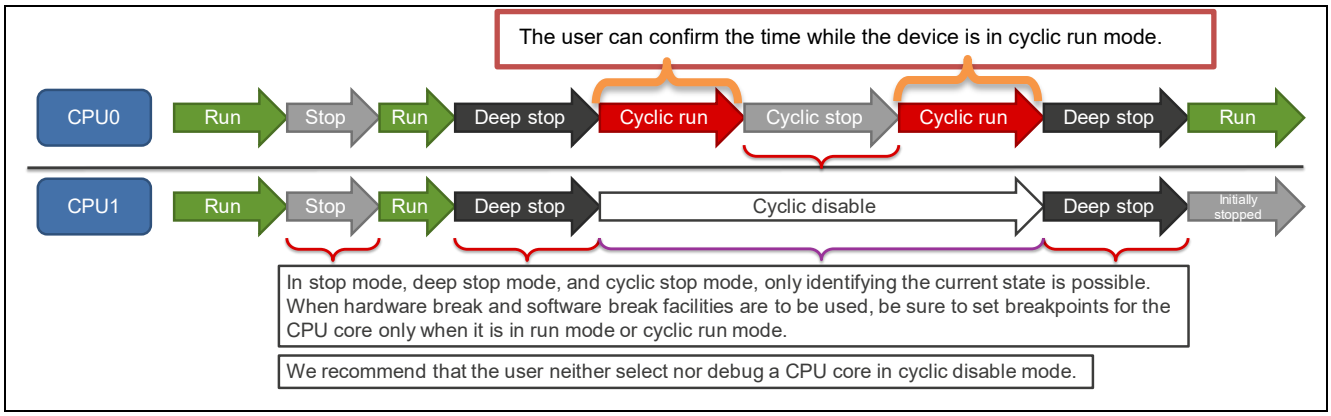

Figure 4-36 Confirming the Time in Cyclic Run Mode in Example Application 2

Use the timer facility of the debugger to specify the start of time measurement as the first address of the retention RAM that causes the transition to cyclic run mode and the end of time measurement as the program address where the device has returned to cyclic stop mode from cyclic run mode or to deep stop mode from cyclic run mode. Executing the program will provide a measurement of the time from the program on CPU0 placing the device in cyclic run mode until the device enters cyclic stop mode or deep stop mode.

If the time in cyclic run mode does not satisfy the requirements in terms of time restrictions, the user can confirm the times taken by the parts of the program in cyclic run mode by measuring the times from the start to end of time measurement or using the display from tracing of times taken for processing. This clarifies how long each part of processing by the program is taking.

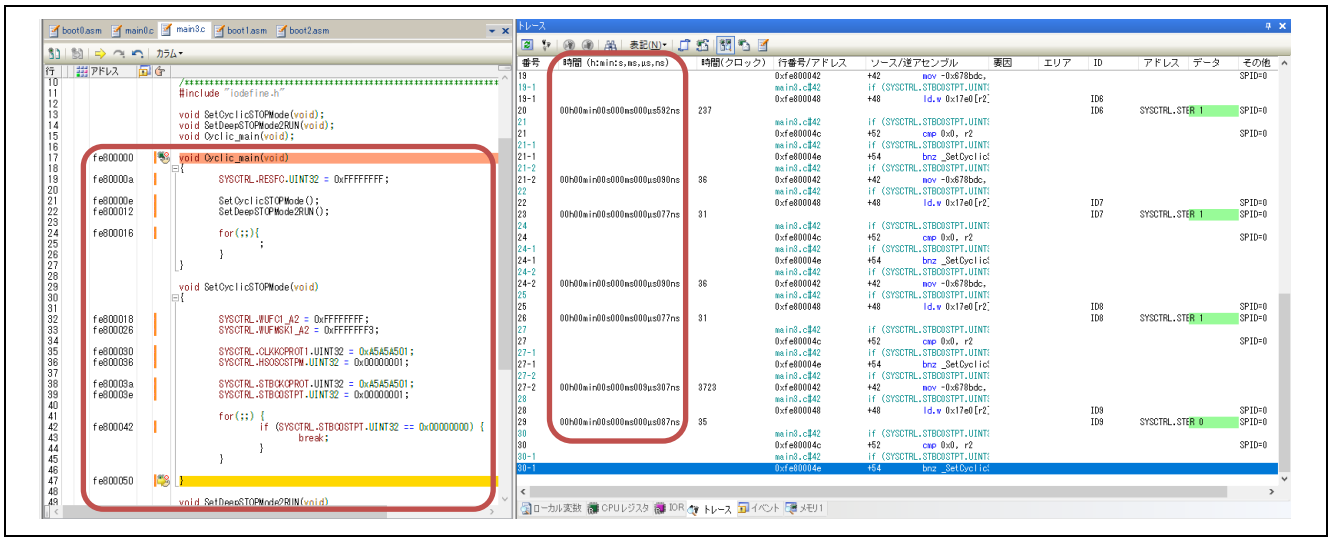

Figure 4-37 Result of Tracing a Program in CS+ in Cyclic Run Mode

4.2.4 Cyclic Stop Mode

This section describes how to cause a transition of a device to cyclic stop mode and confirm that the device is in cyclic stop mode. The user can confirm that the device is in cyclic stop mode in the following way.

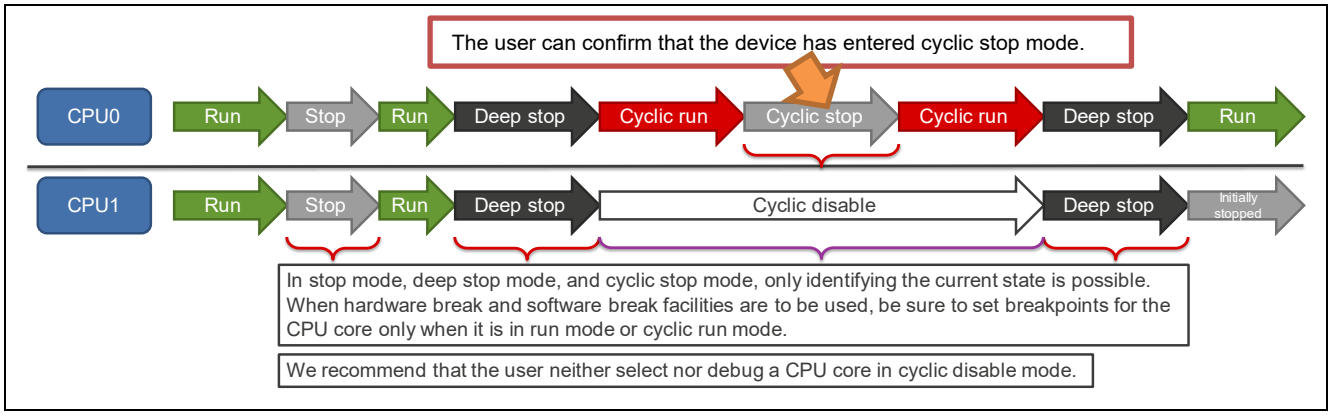

Figure 4-38 Confirming Cyclic Stop Mode in Example Application 2

Figure 4-39 shows an example of a program that causes a transition to cycle stop mode. Since the program is executed from the retention RAM when the CPU core is in cyclic run mode, a program must be downloaded to the retention RAM. When CPU0 in the example of an application shown in Figure 4-38 executes this program, the device enters cyclic stop mode. For details on registers and programming, refer to the hardware manual for the device.

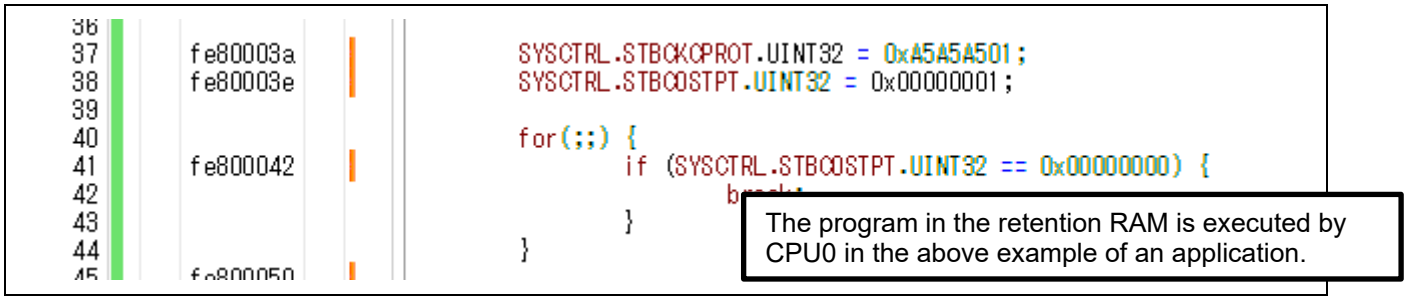

Figure 4-39 Example of a Program that Causes a Transition to Cyclic Stop Mode

The user can confirm that the device is in cyclic stop mode by identifying the state. The displays in each debugger after the state is identified are shown below.

- CS+
  CS+ shows “Cyclic Stop” as the state to indicate that the device is in cyclic stop mode.

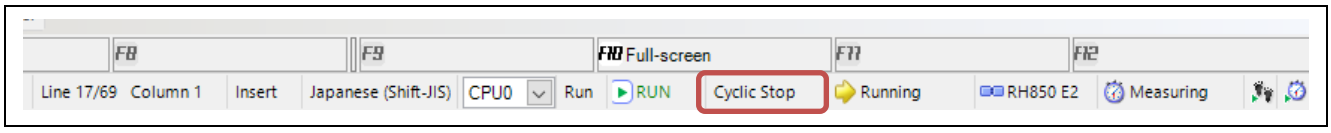

Figure 4-40 Displaying the State of the Device as Cyclic Stop Mode in CS+

- MULTI
  MULTI shows “0x800” (numerals) and “CYCLE-STOP” (a string) to indicate that the state of the device is cyclic stop mode. Since the device is in this standby mode, the indicator for cyclic stop mode is displayed for CPU0.

```
850eserv2> cpustatus
CPU0 CPU status (0x800): CYCLE-STOP
Core is Running

CPU1 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running

CPU2 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running

CPU3 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running
```

Figure 4-41 Displaying the State of the Device as Cyclic Stop Mode in MULTI

4.2.4.1 Starting Debugging Immediately after the Device has Made the Transition to Cyclic Run Mode from Cyclic Stop Mode

This section describes how to generate a break in the CPU core immediately after the device has entered cyclic run mode due to wakeup resources after having been in cyclic stop mode. With the method shown below, the user can confirm the state of the device immediately after it has made the transition to cyclic run mode from cyclic stop mode and start debugging of an application immediately after the device enters cyclic run mode.

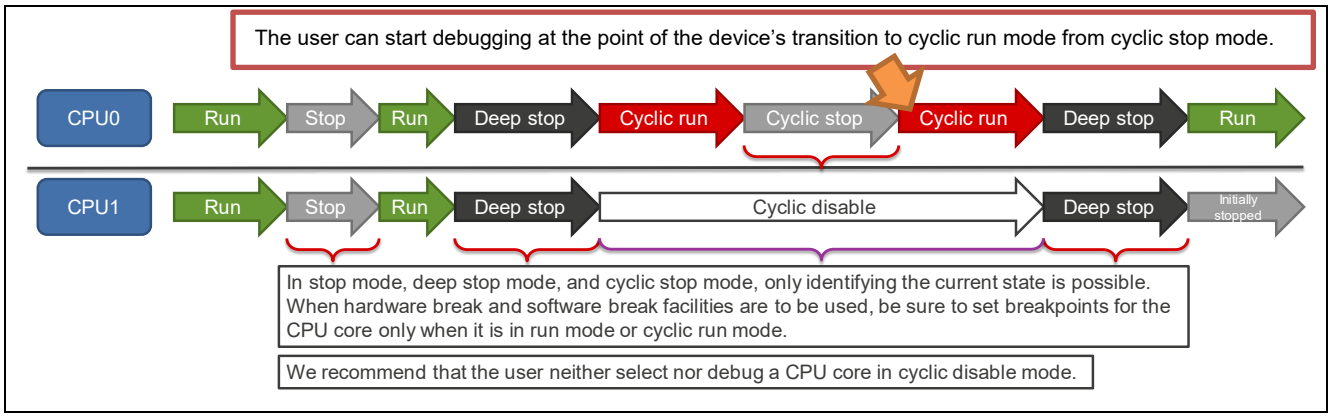

Figure 4-42 Transition to Cyclic Run Mode from Cyclic Stop Mode in Example Application 2

Figure 4-43 shows an example of the debugging of a program where a break occurs at the point of the CPU core's transition to cyclic run mode from cyclic stop mode. When the CPU core has made the transition to cyclic run mode from cyclic stop mode, the program is executed after checking the value of the STBC0STPT register. To start debugging at the point where the CPU core has entered cyclic run mode, set a breakpoint after checking the STBC0STPT register before the CPU core enters cyclic stop mode. This enables the generation of a break at the point where the CPU core has entered cyclic run mode.

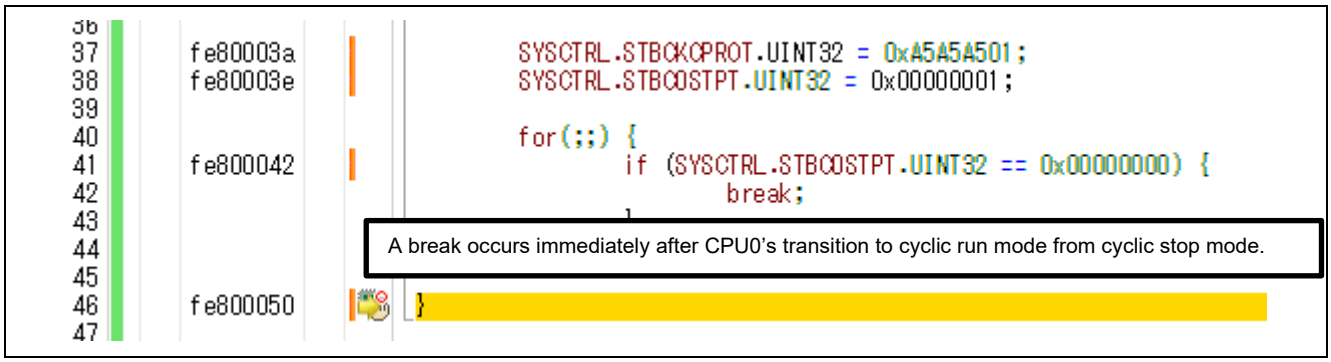

Figure 4-43 Example of a Break Following the Device's Transition to Cyclic Run Mode from Cyclic Stop Mode

4.2.4.2 Confirming Satisfaction of the Requirements in Terms of Restrictions on Time in Cyclic Stop Mode

This section describes how to measure the time the device is in cyclic stop mode before returning to cyclic run mode. With the method shown below, the user can confirm that the requirements in terms of restrictions on time in cyclic stop mode have been satisfied.

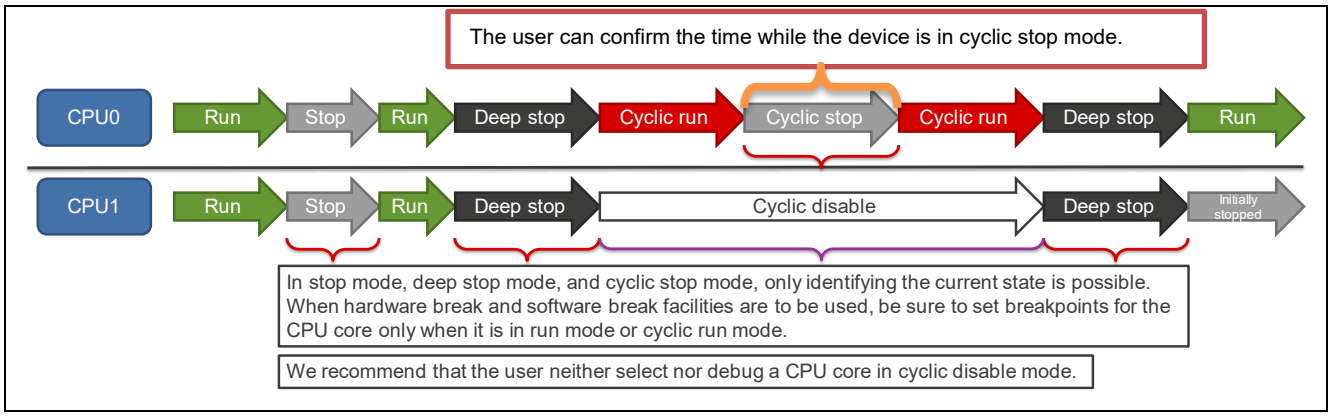

Figure 4-44 Confirming the Time in Cyclic Stop Mode in Example Application 2

Use the timer facility of the debugger to specify the start of time measurement as the program address that causes the transition to cyclic stop mode and the end of time measurement as the program address immediately after the device has returned to cyclic run mode from cyclic stop mode. Executing the program will provide a measurement of the time from the program on CPU0 placing the device in cyclic stop mode until the device enters cyclic run mode.

4.2.5 Cyclic Disable Mode

This section describes how to confirm that the CPU core is in cyclic disable mode. The user can confirm that the CPU core is in cyclic disable mode in the following way.

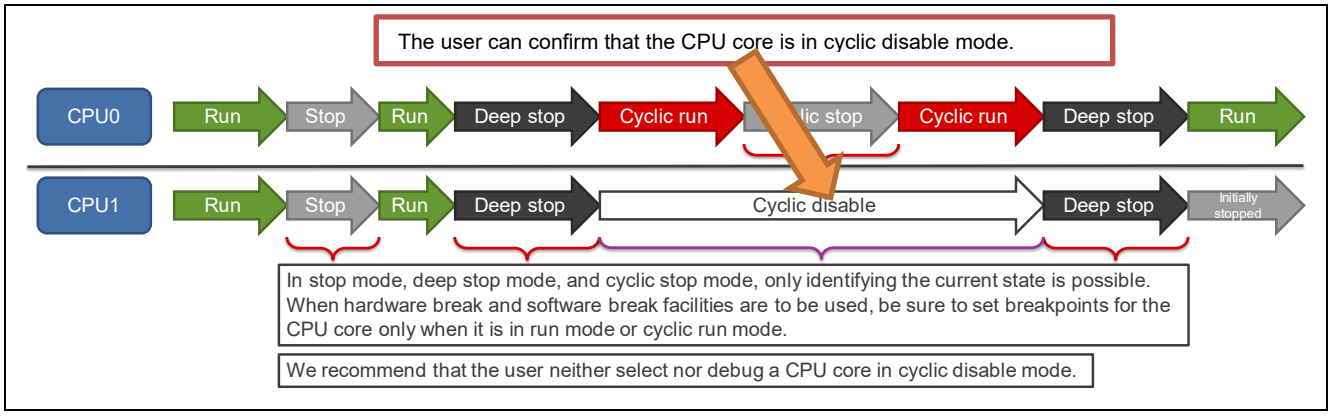

Figure 4-45 Confirming Cyclic Disable Mode in Example Application 2

The user can confirm that the CPU core is in cyclic disable mode by identifying the state. The displays in each debugger after the state is identified are shown below.

- CS+
  CS+ shows “Cyclic Disable” as the state to indicate that the CPU core is in cyclic disable mode.

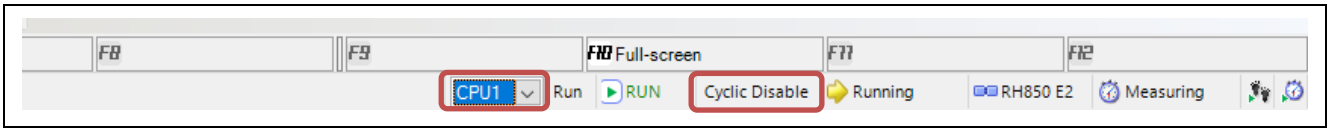

Figure 4-46 Displaying the State of the CPU Core as Cyclic Disable Mode in CS+

- MULTI
  MULTI shows “0x1000” (numerals) and “CYCLE-STOP INVALID” (a string) to indicate that the state of the CPU core is cyclic disable mode. Since the device is in this standby mode, the indicator for cyclic disable mode is for CPUs other than CPU0.

```
850eserv2> cpustatus
CPU0 CPU status (0x400): CYCLE-RUN
Core is Running

CPU1 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running

CPU2 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running

CPU3 CPU status (0x1000): CYCLE-STOP INVALID
Core is Running
```

Figure 4-47 Displaying the State of the CPU Core as Cyclic Disable Mode in MULTI

5. Points for Caution

This chapter describes points for caution in the debugging of devices incorporating an initially stopped core and devices that enter standby modes.

5.1 Executing a Program to Cause the Device to Enter Stop Mode or Cyclic Stop Mode

The following items are prohibited: starting the execution of a program after having written to the STBC0STPT register, stepped execution of writing to the STBC0STPT register, and setting a breakpoint within the loop for confirming the value of the STBC0STPT register.

When debugging writing to the STBC0STPT register, after a break has occurred in the processing before writing to the STBC0STPT register, execute a program that causes the device to enter stop mode or cyclic stop mode.

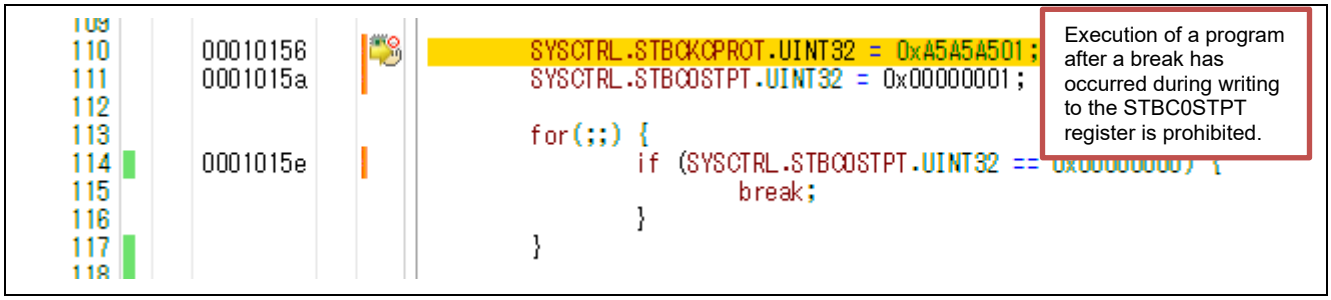

Figure 5-1 Example of a Break which Occurs before Writing to the STBC0STPT Register

5.2 Executing a Program to Cause the Device to Enter Deep Stop Mode

The following items are prohibited: starting the execution of a program after having written to the STBC0PSC register, stepped execution of writing to the STBC0PSC register, and setting a breakpoint within the unconditional loop to occur after writing to the STBC0PSC register.

When debugging writing to the STBC0PSC register, after a break has occurred in the processing before writing to the STBC0PSC register, execute a program that causes the device to enter deep stop mode.

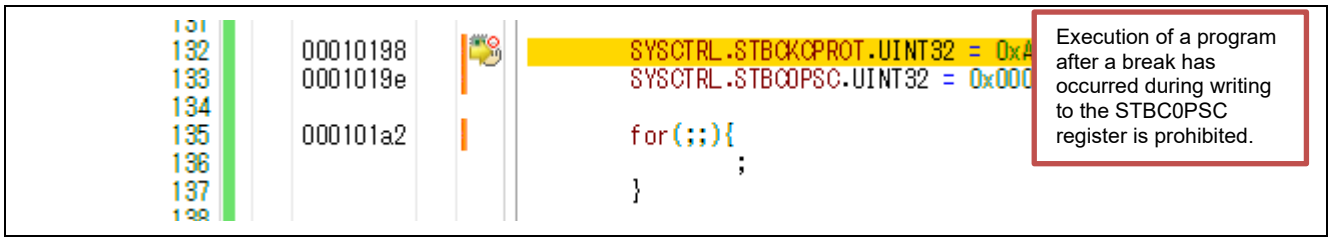

Figure 5-2 Example of a Break which Occurs before Writing to the STBC0PSC Register

5.3 Debugging an Initially Stopped Core in Devices with the ICU-M Core Enabled

For devices with the ICU-M core enabled, the ICU-M core is active by default on release form a reset, so all CPU cores are initially stopped cores. To make the initially stopped cores operate in this situation, they must be activated from the ICU-M core. Activating the initially stopped cores can be done by writing code for processing by the ICU-M core to activate the initially stopped cores or use the debugger to make selected register operations in the ICU-M core. An option byte can also be used to set whether the CPU cores are activated on release from a reset in devices that have an ICU-M core.

For main-core debugging, where only the CPU core or cores are to be debugged and the ICU-M core is not, although the ICU-M core is present, all CPU cores are initially stopped cores which enter the initially stopped state on release from a reset. Thus the ICU-M core cannot be debugged and debugging of devices is impossible. When the CPU cores are to be debugged in main-core debugging, write code for processing to activate the initially stopped cores in the program for the ICU-M core. For an example of the required program, refer to section 4.1.2, Activating the Initially Stopped Core. Since the program for the ICU-M core will be running during main-core debugging, the initially stopped cores are activated when the ICU-M core executes the required processing, which enables debugging of the device.

5.4 Debugging Standby Modes in Devices with the ICU-M Core Enabled

Synchronous debugging of devices that have been entered in standby mode is not possible for devices with the ICU-M core enabled. For this reason, debugging of cyclic run mode is not possible in devices with the ICU-M core enabled. Since the ICU-M core need not be operating in the debugging of a program in cyclic run mode, disable the ICU-M core to proceed with the debugging.

Even if a device with the ICU-M core enabled has entered standby mode, synchronous debugging becomes possible again when the device is returned to run mode by a program which was in progress.

5.5 Hot Plug-in Debugging

When the device is in cyclic run mode or cyclic stop mode, hot plug-in connection is not possible. Since hot plug-in connection is possible when the device is in run mode, stop mode, or deep stop mode, retry the hot plug-in connection after having placed the device in one of those modes.

If the device enters a standby mode during hot plug-in connection, the hot plug-in connection will fail. In such cases, retry the hot plug-in connection.

While the program is running immediately after a hot plug-in connection (hereafter referred to as “the hot plug-in run state”), only the following debugging facilities are available. If you wish to use other debugging facilities, generate a break to cause the device to make a transition to the break state from the hot plug-in run state.

- Reading or writing RAM areas
- Reading or writing peripheral I/O registers
- Forced break*
- Forced reset*

Note: If a core of the device is an initially stopped core which is still in the initially stopped state or has entered stop mode, deep stop mode, or cyclic stop mode, the forced break and forced reset facilities are not available. Figure 5-3 shows the error message which appears in CS+ when the user attempts to use the forced break facility while a core of a device in the hot plug-in run state is an initially stopped core which is still in the initially stopped state. Figure 5-4 shows the error message which appears in MULTI when the user attempts to use the forced break facility while a core of a device in the hot plug-in run state is an initially stopped core which is still in the initially stopped state. If you wish to use the forced break and forced reset facilities, check the states of the CPU cores.

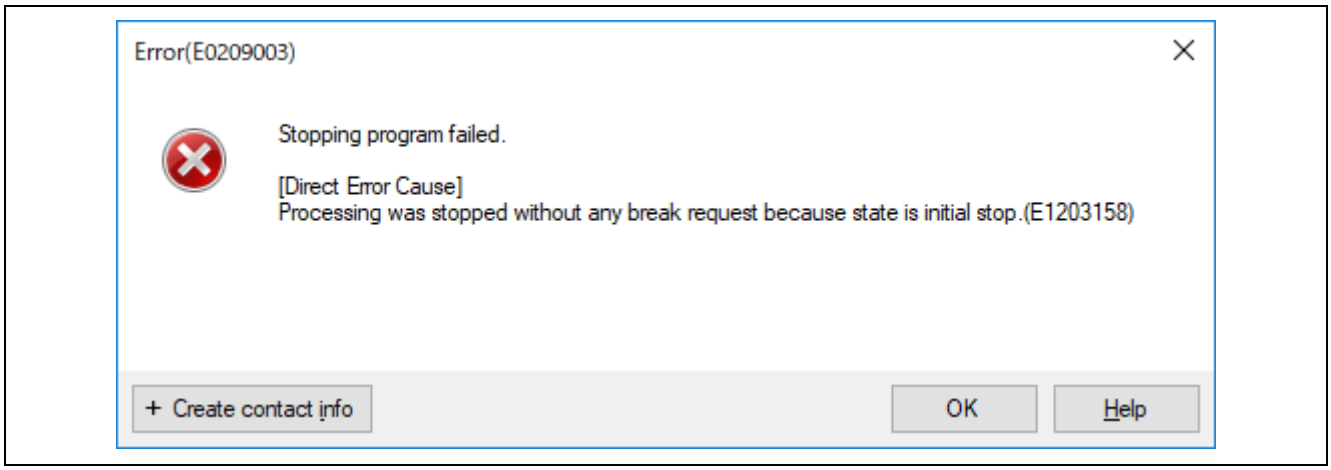

Figure 5-3 Error Message in Response to an Attempt to Use the Forced Break Facility in a Device which Has an Initially Stopped Core in the Initially Stopped State (CS+)

```
0x0c56:status err
(break request is canceled by fetch-stop)
```

Figure 5-4 Error Message in Response to an Attempt to Use the Forced Break Facility in a Device which Has an Initially Stopped Core in the Initially Stopped State (MULTI)

5.6 Operations Related to Setting and Deleting Hardware Breakpoints

When a CPU core is active and starts execution of a program, the debugger cannot set hardware breakpoints for an initially stopped core in the initially stopped state or a CPU core in cyclic disable mode. When the initially stopped core is activated or the CPU core is activated from cyclic disable mode during the execution of a program by another CPU, a break may not occur even if the instruction at an address which has been set has set as a hardware breakpoint for the newly activated CPU core is executed. Figure 5-5 shows an example of such operation. A break will occur at a hardware breakpoint that has been set for the CPU core in the case of an initially stopped core that has been activated or a CPU core that is no longer in cyclic disable mode.

With regard to the setting of hardware breakpoints, refer to sections 4.1 and 4.2. When an initially stopped core is activated or a CPU core is activated from cyclic disable mode, generate a break, set the hardware breakpoint, and start executing the program. The hardware breakpoint will have been set for the CPU core.

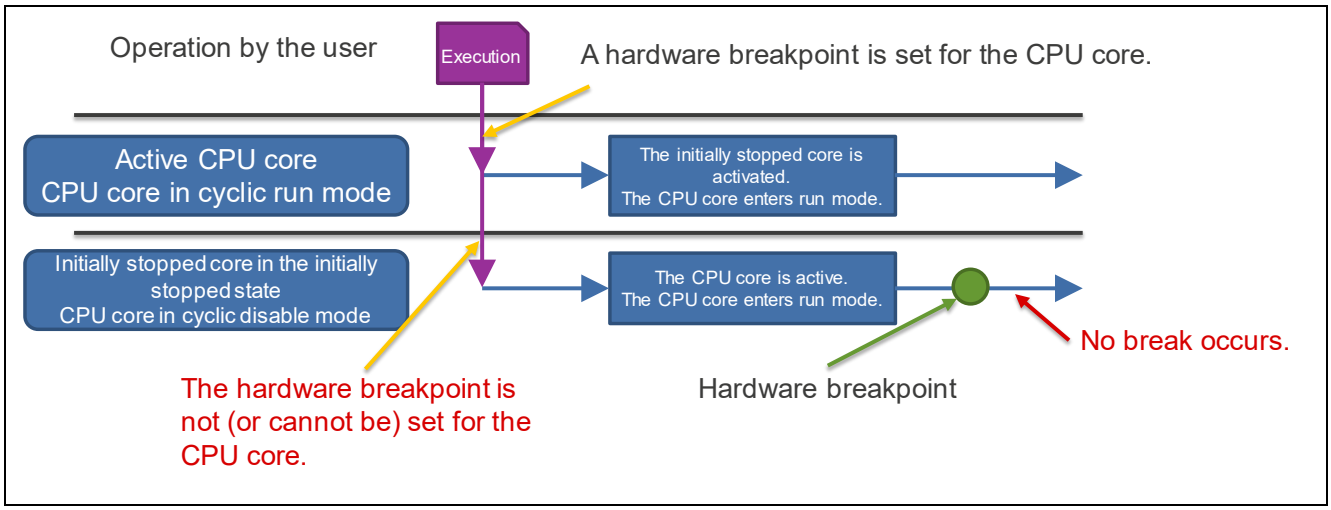

Figure 5-5 Example of Operation where a Hardware Breakpoint is Not Set for the CPU Core so a Break Does Not Occur

If a break is inserted while an initially stopped core is in the initially stopped state or the CPU core enters the cyclic disable state during execution of a program, the debugger will be unable to delete the specified hardware breakpoint when program execution is to be resumed. Thus, when the user restarts execution of the program while ignoring the breakpoint that has been specified and the initially stopped core is activated or the CPU core is activated from the cyclic disable mode during the execution of a program, the hardware breakpoint which was not deleted will be effective so that a break occurs. Figure 5-6 shows an example of such operation.

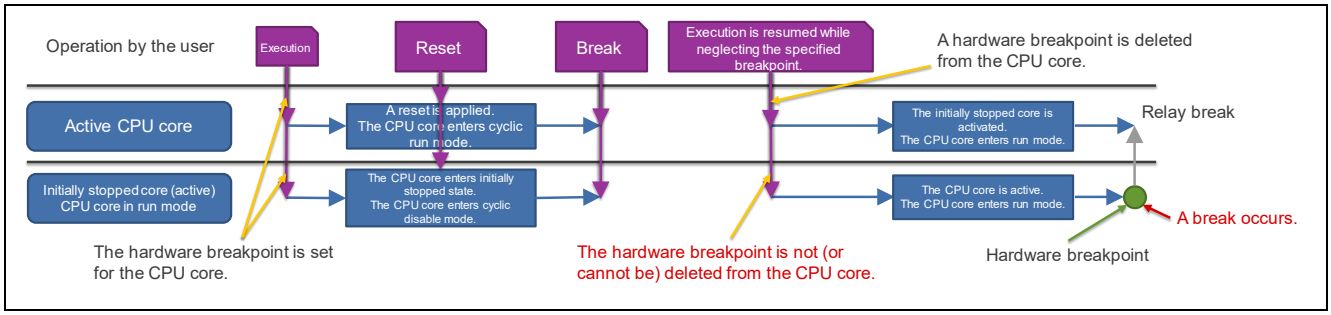

Figure 5-6 Example of Operation where a Hardware Breakpoint for a CPU Core is Not Deleted so a Break that was Not Intended Occurs

5.7 State of a Hardware Breakpoint Set for an Initially Stopped Core at the Time of Downloading a Program

If set to do so, the debugger applies a reset to the device following the downloading of a program. After that, an initially stopped core will enter the initially stopped state.

In CS+ and MULTI, a hardware breakpoint that was set for an initially stopped core in the active state the last time a program was being executed before downloading of a program will remain set. A hardware breakpoint which was set after the final execution before downloading of a program will not be set for the device.

The address to which the program has been written is set as a hardware breakpoint for the device. Even if the program has been modified before downloading, the hardware breakpoint for the program which before it was modified remains in place for the device.

Figure 5-7 shows an example of the operation. The hardware breakpoint which was set in the program that has now been modified remains set for the device by the debugger when the program is re-executed after the initially stopped core is activated and a break occurs.

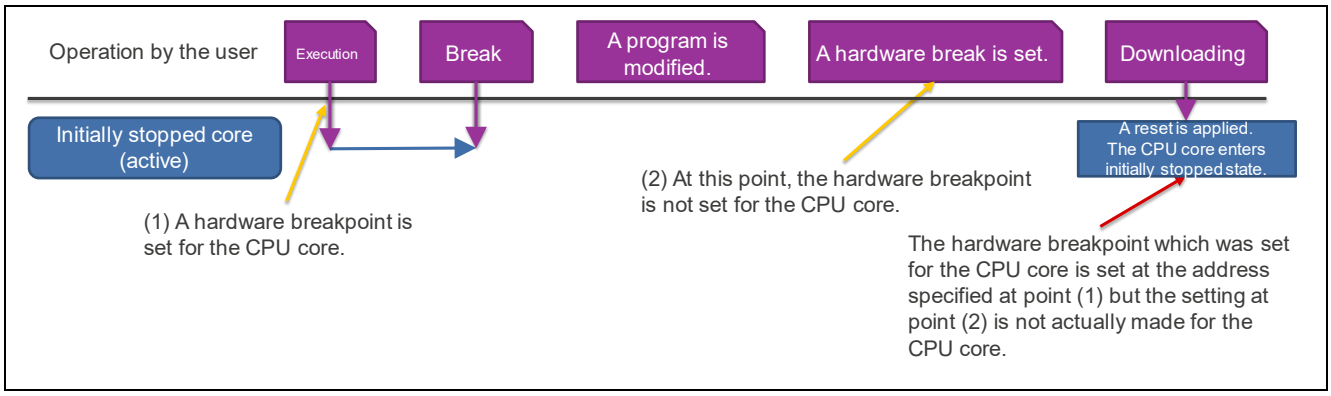

Figure 5-7 State where a Hardware Breakpoint is Set after Downloading a Program in CS+ and MULTI

Revision History

Rev.	Date	Description	
		Page	Summary
1.00	Jan.08.20	—	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 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. This document shall not be reprinted, reproduced or duplicated in any form, in whole or in part, without prior written consent of Renesas Electronics.
12. 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.4.0-1 November 2017)

Corporate Headquarters

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

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/

Trademarks

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