embOS

Real-Time
Operating System

CPU & Compiler
specifics for SH2A core
using IAR Embedded
Workbench

Document: UM01064
Software version 4.22
Revision: 0
Date: May 27, 2016

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller GmbH & Co. KG

In den Weiden 11
D-40721 Hilden

Germany

Tel.+49 2103-2878-0
Fax.+49 2103-2878-28
E-mail: support@segger.com
Internet: http://www.segger.com

Manual versions

This manual describes the current software version. If any error occurs, inform us and we will try to assist you as soon as possible.
Contact us for further information on topics or routines not yet specified.

Print date: May 27, 2016

Software	Revision	Date	By	Description
4.22	0	160527	TS	First FrameMaker version.

About this document

Assumptions

This document assumes that you already have a solid knowledge of the following:

- The software tools used for building your application (assembler, linker, C compiler)
- The C programming language
- The target processor
- DOS command line

If you feel that your knowledge of C is not sufficient, we recommend The C Programming Language by Kernighan and Richie (ISBN 0-13-1103628), which describes the standard in C-programming and, in newer editions, also covers the ANSI C standard.

How to use this manual

This manual explains all the functions and macros that the product offers. It assumes you have a working knowledge of the C language. Knowledge of assembly programming is not required.

Typographic conventions for syntax

This manual uses the following typographic conventions:

Style	Used for
Body	Body text.
`Keyword`	Text that you enter at the command-prompt or that appears on the display (that is system functions, file- or pathnames).
`Parameter`	Parameters in API functions.
`Sample`	Sample code in program examples.
`Sample comment`	Comments in program examples.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections.

Table 2.1: Typographic conventions

SEGGER Microcontroller GmbH & Co. KG develops and distributes software development tools and ANSI C software components (middleware) for embedded systems in several industries such as telecom, medical technology, consumer electronics, automotive industry and industrial automation.

SEGGER's intention is to cut software development time for embedded applications by offering compact flexible and easy to use middleware, allowing developers to concentrate on their application.

Our most popular products are emWin, a universal graphic software package for embedded applications, and embOS, a small yet efficient real-time kernel. emWin, written entirely in ANSI C, can easily be used on any CPU and most any display. It is complemented by the available PC tools: Bitmap Converter, Font Converter, Simulator and Viewer. embOS supports most 8/16/32-bit CPUs. Its small memory footprint makes it suitable for single-chip applications.

Apart from its main focus on software tools, SEGGER develops and produces programming tools for flash micro controllers, as well as J-Link, a JTAG emulator to assist in development, debugging and production, which has rapidly become the industry standard for debug access to ARM cores.

Corporate Office:
http://www.segger.com

United States Office:
http://www.segger-us.com

EMBEDDED SOFTWARE (Middleware)

emWin
Graphics software and GUI
emWin is designed to provide an efficient, processor- and display controller-independent graphical user interface (GUI) for any application that operates with a graphical display.

embOS
Real Time Operating System
embOS is an RTOS designed to offer the benefits of a complete multitasking system for hard real time applications with minimal resources.

embOS/IP
TCP/IP stack
embOS/IP a high-performance TCP/IP stack that has been optimized for speed, versatility and a small memory footprint.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and Compact-Flash cards, are available.

USB-Stack
USB device/host stack
A USB stack designed to work on any embedded system with a USB controller. Bulk communication and most standard device classes are supported.

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for micro controllers.

J-Link
JTAG emulator for ARM cores
USB driven JTAG interface for ARM cores.

J-Trace
JTAG emulator with trace
USB driven JTAG interface for ARM cores with Trace memory. supporting the ARM ETM (Embedded Trace Macrocell).

J-Link / J-Trace Related Software
Add-on software to be used with SEGGER's industry standard JTAG emulator, this includes flash programming software and flash breakpoints.

Table of Contents

Chapter 1

Using embOS

This chapter describes how to start with and use embOS. You should follow these steps to become familiar with embOS.

1.1 Installation

embOS is shipped as a zip-file in electronic form.

To install it, proceed as follows:

Extract the zip-file to any folder of your choice, preserving the directory structure of this file. Keep all files in their respective sub directories. Make sure the files are not read only after copying.

Assuming that you are using an IDE to develop your application, no further installation steps are required. You will find a lot of prepared sample start projects, which you should use and modify to write your application. So follow the instructions of section *First steps*.

You should do this even if you do not intend to use the IDE for your application development to become familiar with embOS.

If you do not or do not want to work with the IDE, you should: Copy either all or only the library-file that you need to your work-directory. The advantage is that when switching to an updated version of embOS later in a project, you do not affect older projects that use embOS, too. embOS does in no way rely on an IDE, it may be used without the IDE using batch files or a make utility without any problem.

1.2 First steps

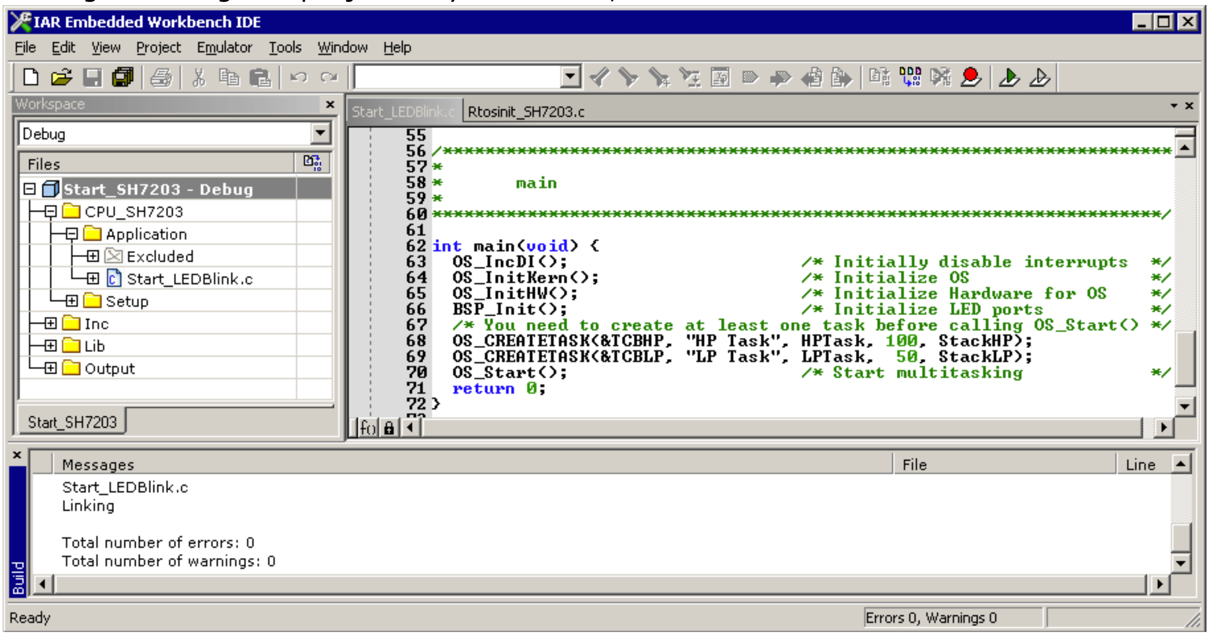
After installation of embOS you can create your first multitasking application. You have received several ready to go sample start workspaces and projects and every other files needed in the subfolder **Start**. It is a good idea to use one of them as a starting point for all of your applications. The subfolder **BoardSupport** contains the workspaces and projects which are located in manufacturer- and CPU-specific subfolders.

To start with, you may use any project from **BoardSupport** subfolder:

To get your new application running, you should proceed as follows:

- Create a work directory for your application, for example `c:\work`.
- Copy the whole folder **Start** which is part of your embOS distribution into your work directory.
- Clear the read-only attribute of all files in the new **Start** folder.
- Open one sample workspace/project in
  Start\BoardSupport\Renesas\<CPU>
  with your IDE (for example, by double clicking it).
- Build the project. It should be built without any error or warning messages.

After generating the project of your choice, the screen should look like this:



```
/**********************************************************************
*
*        main
*
**********************************************************************/

int main(void) {
  OS_IncDI();                          /* Initially disable interrupts  */
  OS_InitKern();                       /* Initialize OS                 */
  OS_InitHW();                         /* Initialize Hardware for OS    */
  BSP_Init();                          /* Initialize LED ports          */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                          /* Start multitasking            */
  return 0;
}
```

```
Start_LEDBlink.c
Linking

Total number of errors: 0
Total number of warnings: 0
```

For additional information you should open the `ReadMe.txt` file which is part of every specific project. The ReadMe file describes the different configurations of the project and gives additional information about specific hardware settings of the supported eval boards, if required.

1.3 The example application OS_StartLEDBlink.c

The following is a printout of the example application `OS_Start_LEDBlink.c`. It is a good starting point for your application. (Note that the file actually shipped with your port of embOS may look slightly different from this one.)

What happens is easy to see:

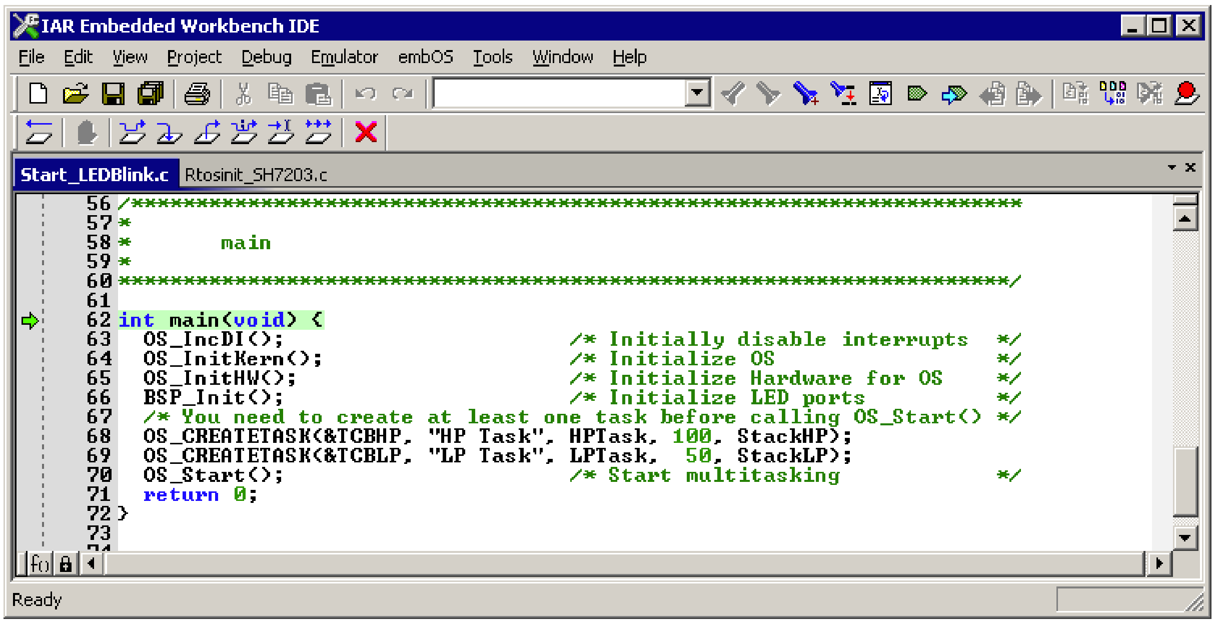
After initialization of embOS; two tasks are created and started.
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

```
----------------------------------------------------------------------
File    : OS_Start_LEDBlink.c
Purpose : embOS sample program toggling one LED in each task
--------- END-OF-HEADER ---------------------------------------------
*/

#include "RTOS.h"
#include "BSP.h"

OS_STACKPTR int StackHP[128], StackLP[128]; /* Task stacks */
OS_TASK         TCBHP, TCBLP;       /* Task-control-blocks */

static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}

/*********************************************************************
*
*       main()
*/
int main(void) {
  OS_IncDI();                      /* Initially disable interrupts  */
  OS_InitKern();                   /* Initialize OS                 */
  OS_InitHW();                     /* Initialize Hardware for OS    */
  BSP_Init();                      /* Initialize LED ports          */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking            */
  return 0;
}
```

1.4 Stepping through the sample application

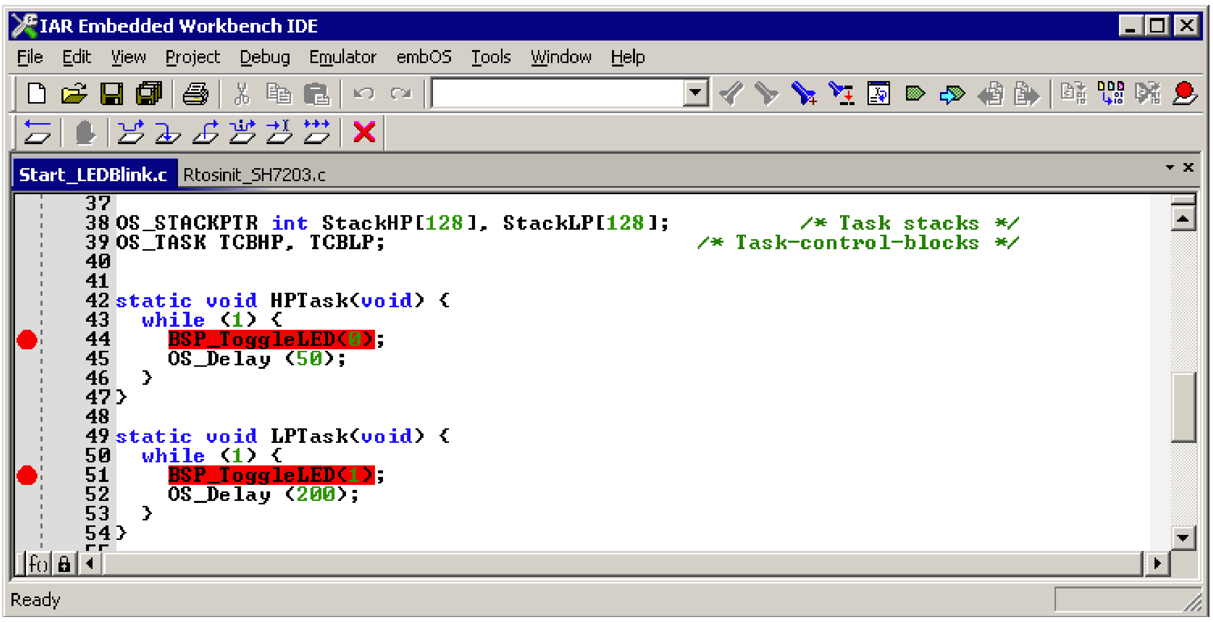
When starting the debugger, you will see the main() function (see example screenshot below). The main() function appears as long as project option **Run to main** is selected, which it is enabled by default. Now you can step through the program. OS_IncDI() initially disables interrupts.
OS_InitKern() is part of the embOS library and written in assembler; you can therefore only step into it in disassembly mode. It initializes the relevant OS variables. Because of the previous call of OS_IncDI(), interrupts are not enabled during execution of OS_InitKern().
OS_InitHW() is part of RTOSInit_*.c and therefore part of your application. Its primary purpose is to initialize the hardware required to generate the system tick interrupt for embOS. Step through it to see what is done.

OS_Start() should be the last line in main(), because it starts multitasking and does not return.


IAR Embedded Workbench IDE
File Edit View Project Debug Emulator embOS Tools Window Help
Start_LEDBlink.c
Rtosinit_SH7203.c
Ready

```
/*********************************************************************
*
*       main
*
**********************************************************************/

int main(void) {
  OS_IncDI();                      /* Initially disable interrupts  */
  OS_InitKern();                   /* Initialize OS                 */
  OS_InitHW();                     /* Initialize Hardware for OS    */
  BSP_Init();                      /* Initialize LED ports          */
  /* You need to create at least one task before calling OS_Start() */
  OS_CREATETASK(&TCBHP, "HP Task", HPTask, 100, StackHP);
  OS_CREATETASK(&TCBLP, "LP Task", LPTask,  50, StackLP);
  OS_Start();                      /* Start multitasking            */
  return 0;
}
```

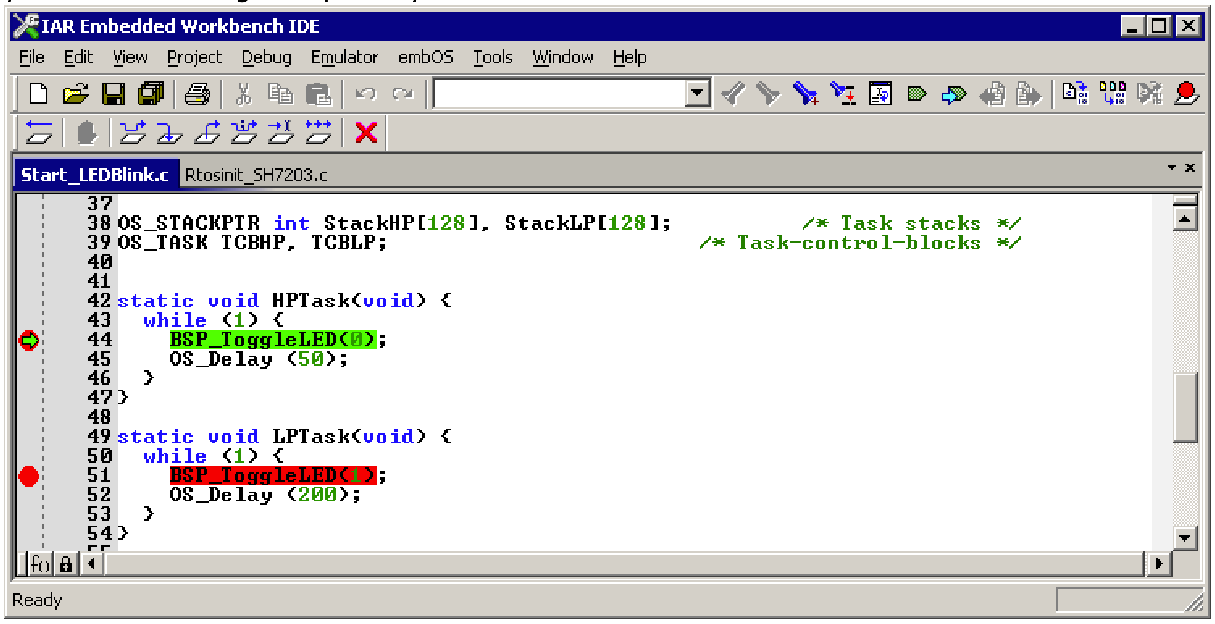
Before you step into OS_Start(), you should set two breakpoints in the two tasks as shown below.


IAR Embedded Workbench IDE
File Edit View Project Debug Emulator embOS Tools Window Help
Start_LEDBlink.c
Rtosinit_SH7203.c
Ready

```
OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
OS_TASK TCBHP, TCBLP;                        /* Task-control-blocks */


static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}
```

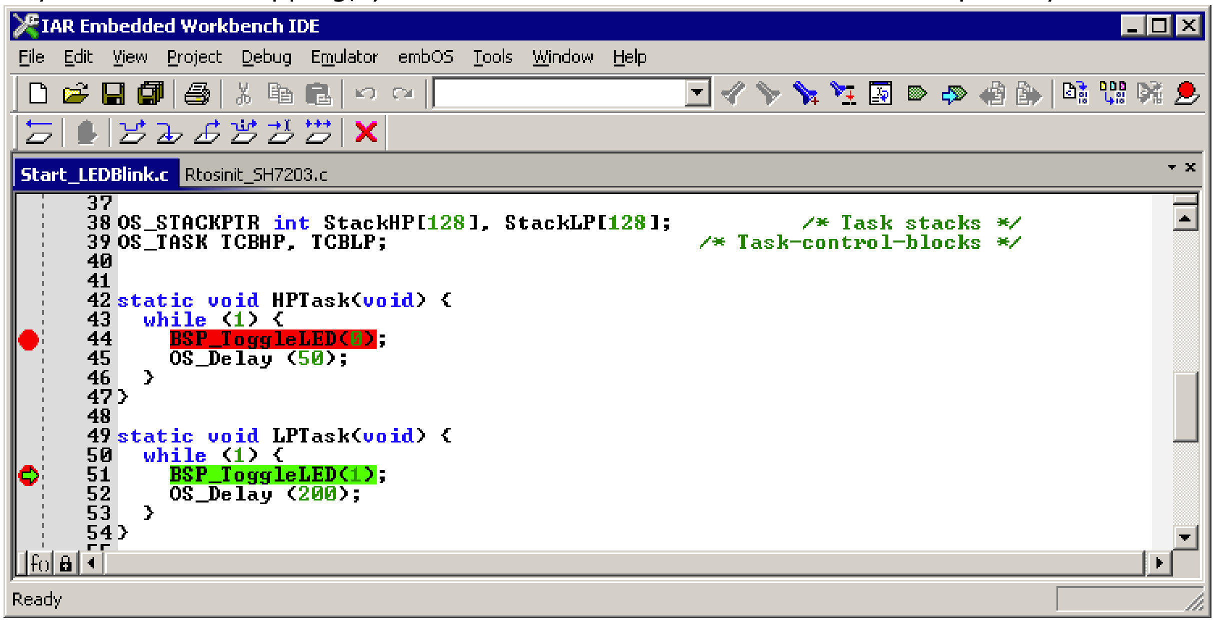
As OS_Start() is part of the embOS library, you can step through it in disassembly mode only.

Click **GO**, step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task.


IAR Embedded Workbench IDE
File Edit View Project Debug Emulator embOS Tools Window Help
Start_LEDBlink.c
Rtosinit_SH7203.c
Ready

```
OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
OS_TASK TCBHP, TCBLP;                        /* Task-control-blocks */


static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}
```

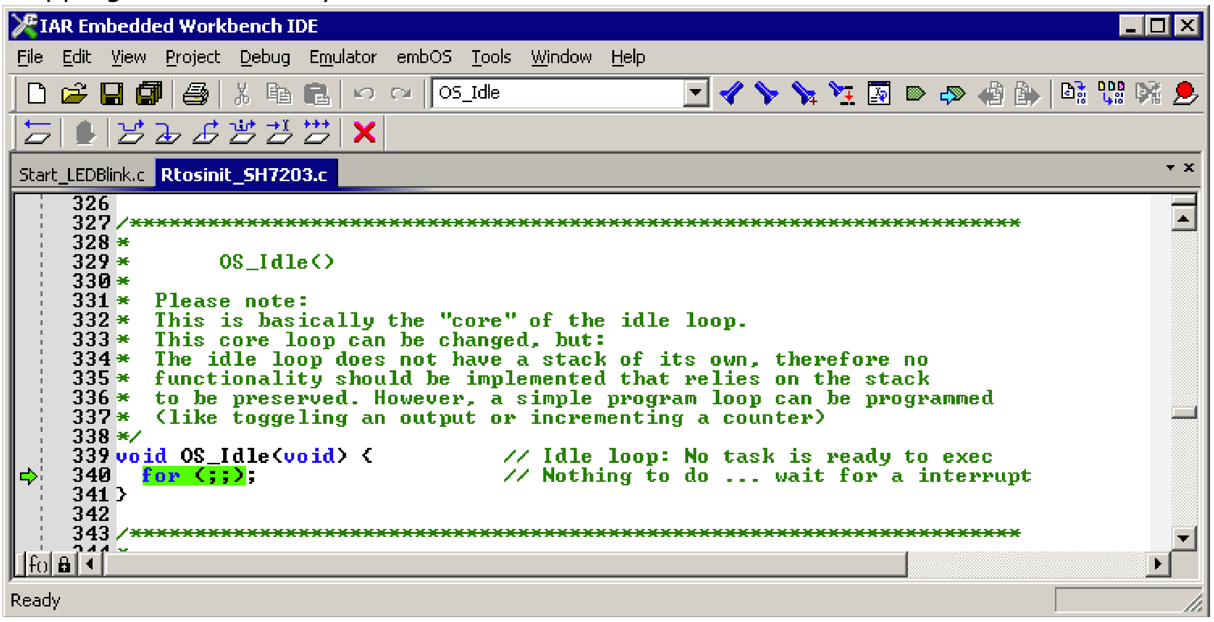
If you continue stepping, you will arrive at the task that has lower priority:


IAR Embedded Workbench IDE
File Edit View Project Debug Emulator embOS Tools Window Help
Start_LEDBlink.c
Rtosinit_SH7203.c
Ready

```
OS_STACKPTR int StackHP[128], StackLP[128];          /* Task stacks */
OS_TASK TCBHP, TCBLP;                        /* Task-control-blocks */


static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}
```

Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

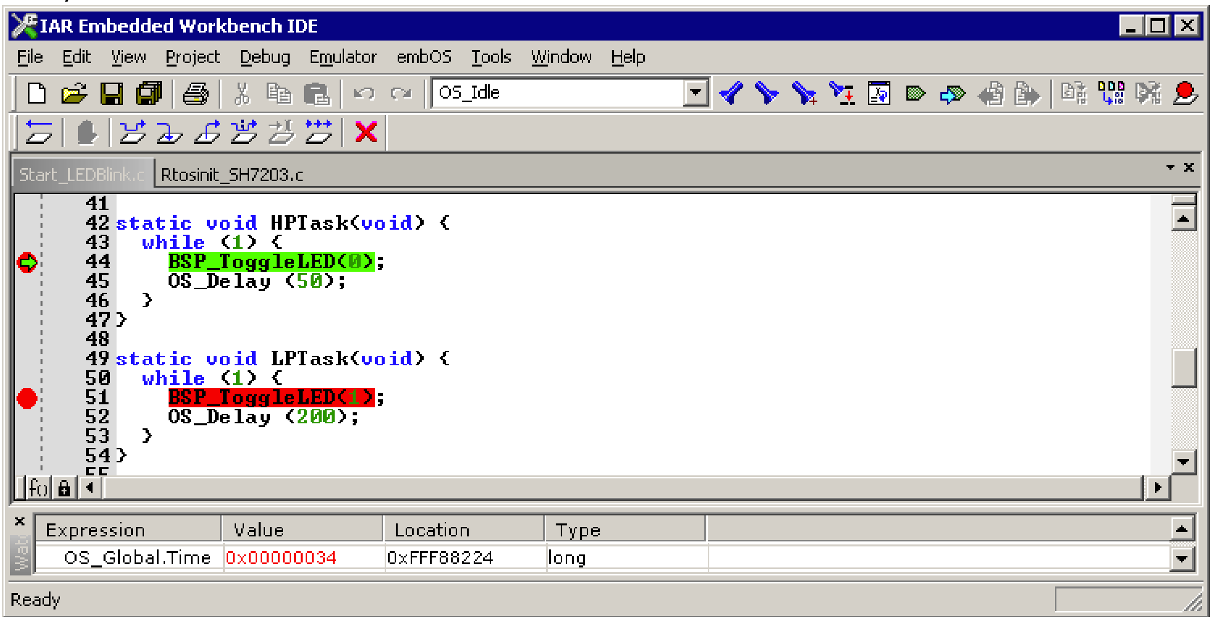
You will arrive there when you step into the `OS_Delay()` function in disassembly mode. `OS_Idle()` is part of `RTOSInit*.c`. You may also set a breakpoint there before stepping over the delay in `LPTask`.


IAR Embedded Workbench IDE
File Edit View Project Debug Emulator embOS Tools Window Help
OS_Idle
Start_LEDBlink.c
Rtosinit_SH7203.c
Ready

```
/*********************************************************************
*
*       OS_Idle()
*
*  Please note:
*  This is basically the "core" of the idle loop.
*  This core loop can be changed, but:
*  The idle loop does not have a stack of its own, therefore no
*  functionality should be implemented that relies on the stack
*  to be preserved. However, a simple program loop can be programmed
*  (like toggeling an output or incrementing a counter)
*/
void OS_Idle(void) {          // Idle loop: No task is ready to exec
  for (;;);                   // Nothing to do ... wait for a interrupt
}

/*********************************************************************
```

If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.

As can be seen by the value of embOS timer variable OS_Global.Time, shown in the Watch window, `HPTask` continues operation after expiration of the 50 system tick delay.


IAR Embedded Workbench IDE
File Edit View Project Debug Emulator embOS Tools Window Help
OS_Idle
Start_LEDBlink.c
Rtosinit_SH7203.c
Watch
Ready

```

static void HPTask(void) {
  while (1) {
    BSP_ToggleLED(0);
    OS_Delay (50);
  }
}

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
    OS_Delay (200);
  }
}
```

Expression	Value	Location	Type
OS_Global.Time	0x00000034	0xFFF88224	long

Chapter 2

Build your own application

This chapter provides all information to set up your own embOS project.

2.1 Introduction

To build your own application, you should always start with one of the supplied sample workspaces and projects. Therefore, select an embOS workspace as described in chapter *First steps* and modify the project to fit your needs. Using an embOS start project as starting point has the advantage that all necessary files are included and all settings for the project are already done.

2.2 Required files for an embOS

To build an application using embOS, the following files from your embOS distribution are required and have to be included in your project:

- `RTOS.h` from subfolder **Inc\**.
  This header file declares all embOS API functions and data types and has to be included in any source file using embOS functions.
- RTOSInit_*.c from one target specific **BoardSupport\\<Manufacturer>\\<MCU>\** subfolder.
  It contains hardware-dependent initialization code for embOS. It initializes the system timer interrupt and optional communication for embOSView via UART or JTAG.
- One embOS library from the subfolder **Lib\**.
- OS_Error.c from one target specific subfolder **BoardSupport\\<Manufacturer>\\<MCU>\**. The error handler is used if any debug library is used in your project.
- Additional CPU and compiler specific files may be required according to CPU.

When you decide to write your own startup code or use a low level init() function, ensure that non-initialized variables are initialized with zero, according to C standard. This is required for some embOS internal variables.
Your `main()` function has to initialize embOS by a call of `OS_InitKern()` and `OS_InitHW()` prior any other embOS functions are called.
You should then modify or replace the OS_`Start_LEDBlink.c` source file in the subfolder **Application\**.

2.3 Change library mode

For your application you might want to choose another library. For debugging and program development you should use an embOS debug library. For your final application you may wish to use an embOS release library or a stack check library.

Therefore you have to select or replace the embOS library in your project or target:

- If your selected library is already available in your project, just select the appropriate configuration.
- To add a library, you may add the library to the existing Lib group. Exclude all other libraries from your build, delete unused libraries or remove them from the configuration.
- Check and set the appropriate `OS_LIBMODE_*` define as preprocessor option and/ or modify the OS_Config.h file accordingly.

2.4 Select another CPU

embOS contains CPU-specific code for various CPUs. Manufacturer- and CPU-specific sample start workspaces and projects are located in the subfolders of the **BoardSupport** folder. To select a CPU which is already supported, just select the appropriate workspace from a CPU-specific folder.

If your CPU is currently not supported, examine all `RTOSInit`.c files in the CPU-specific subfolders and select one which almost fits your CPU. You may have to modify `OS_InitHW()`, OS_COM_Init(), the interrupt service routines for embOS system timer tick and communication to embOSView and the low level initialization.

Chapter 3

Libraries

This chapter describes the available embOS libraries.

3.1 Naming conventions for prebuilt libraries

embOS is shipped with different pre-built libraries with different combinations of the following features:

- Code memory model - `Code model`
- Floating point unit - `Doublesize`
- FPU present - `fpu`
- Library mode - `LibMode`

The libraries are named as follows:

`osSH<code model><doublesize><fpu>_<LibMode>.a`

Parameter	Meaning	Values
`Code model`	Specifies the code memory model.	s: Small code model m: Medium code model l: Large code model h: huge code model
`Doublesize`	Specifies the FPU mode.	f: 32bit doubles d: 64bit doubles
`fpu`	Specifies the present of FPU	f: FPU present n: No FPU
`LibMode`	Specifies the library mode.	xr: Extreme Release
		r: Release
		s: Stack check
		sp: Stack check + profiling
		d: Debug
		dp: Debug + profiling
		dt: Debug + profiling + trace

Example

osSHhff_SP.a is the embOS library for a project using an huge code model, 32bit doubles, FPU with Stack check and profiling support.

Note

embOS for SH2A for IAR compiler is delivered with libraries for all code and data memory models and other options and compiler settings. The different code and data models are described in the IAR compiler manual.

When using the IAR Embedded Workbench, please check the following points:

- One embOS library is part of your project (included in one group of your target). When a CPU with floating point unit is used, the library with floating point option has to be used.
- The appropriate define according to embOS library mode is set as compiler preprocessor option for your project. May be defined in `OS_Config.h`.

Chapter 4

CPU and compiler specifics

4.1 CPU specifics

All hardware specific functions required for embOS are located in the CPU specific Rtosinit_*.c files. Settings for CPU clock speed and UART settings for embOSView are defined with most common defaults. According to your specific hardware, these settings may have to be changed to ensure proper timer tick and UART communication with embOSView.

As far as possible, you should not modify Rtosinit_*.c, as this has the disadvantage, that this modifications have to be tracked when you update to a newer version of embOS. Various CPU derivative may be equipped with different peripherals. It may be necessary to write your own initialization code for your specific CPU derived. You may therefore copy one Rtosinit_*.c file which is closest to your CPU variant and modify this new created file to handle your CPU.

4.2 Clock settings for embOS timer interrupt

OS_InitHW() routine in Rtosinit.c derives timer init values from the constant define OS_PCLK_TIMER. Per default, the value of OS_PCLK_TIMER equals OS_FSYS / OS_PCLK_DIVIDER / 8, which defines the CPU clock of the target system. Wrong settings would result embOS timer ticks unequal to 1 ms. To adapt the embOS timer tick frequency to your CPU, you may:

- Define OS_FSYS as project option. OS_FSYS should equal your CPU clock frequency in Hertz. Note that modification of OS_FSYS may also affect the UART initialization for embOSView.
- You may alternatively define OS_PCLK_TIMER or OS_PCLK_DIVIDER as project option (compiler preprocessor option). These values are used to calculate the timer compare value used for embOS timer.

The CPU clock generator and PLL is initialized during startup in the function __low_level_init() in the source file Rtosinit_*.c.

4.3 Clock settings for UART used for embOSView

OS_COM_Init() routine in RTOSInit.c derives baudrate generator init values from the constant define OS_PCLK_UART. Per default, the value of OS_PCLK_UART equals OS_FSYS / OS_PCLK_DIVIDER.

To ensure correct time base clock for baudrate generator used for embOSView, you may:

- Define OS_FSYS as project option. OS_FSYS should equal your CPU clock frequency in Hertz. Note that modification of OS_FSYS may also affect the timer initialization for embOS tick timer.
- You may alternatively define OS_PCLK_UART as project option (compiler preprocessor option). This value is used to calculate values used to initialize UART used for communication with embOSView.

4.4 Conclusion about clock settings

- **OS_FSYS** has to be defined according to your CPU clock frequency. This should be defined as compiler preprocessor option in your project.
- **OS_PCLK_TIMER** has to be defined to fit the frequency used as peripheral clock for the embOS timer. The value defaults to OS_FSYS. It should be modified and defined as compiler preprocessor option if modification is required.
- **OS_PCLK_UART** has to be defined to fit the frequency used as peripheral clock for the UART used for communication with embOSView. The value defaults to OS_FSYS / OS_PCLK_DIVIDER. It should be modified and defined as compiler preprocessor option if modification is required.

4.5 embOS hardware timer selection

embOS for SH2A CPUs is prepared to use one Compare Match Timer (CMT) channel as time base timer. The initialization code and interrupt handler are delivered in source code and are located in `Rtosinit_*.c`.

4.6 UART for embOSView

Any SCIF UART of the SH2A CPU may be used as communication channel for embOS-View which enables profiling analysis during runtime. The initialization code and interrupt handler are delivered in source code and are located in RTOSInit_*.c. OS_UART may be defined from 0 to 3 to select, initialize and enable one of the SCIFs. When embOSView should not be used, define OS_UART to -1. This may be done in OS_Config.h. The UART used for embOSView requires four interrupt handler which are defined in Rtosinit_*.c

- `OS_ISR_RxErr()` is the reception error interrupt handler.
- `OS_ISR_RxErrB()` is the second reception error interrupt handler.
- `OS_ISR_Rx()` is the reception interrupt.
- `OS_ISR_Tx()` is the transmission interrupt which is called on Tx end condition.

Chapter 5

Stacks

This chapter describes the different stacks.

5.1 Task stack

Each task uses its individual stack. The stack pointer is initialized and set every time a task is activated by the scheduler. The stack-size required for a task is the sum of the stack-size of all routines, plus a basic stack size, plus size used by exceptions.

The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by calling embOS-routines.

The minimum basic task stack size is about 48 bytes. Because any function call uses some amount of stack and every exception also pushes at least 32 bytes onto the current stack, the task stack size has to be large enough to handle one exception too. We recommend at least 256 bytes stack as a start.

Note: The task stacks have to be aligned at EVEN addresses. To ensure proper alignment, implement the task stack as array of int.

5.2 System stack

The minimum system stack size required by embOS is about 136 bytes (stack check & profiling build). However, since the system stack is also used by the application before the start of multitasking (the call to `OS_Start()`), and because software-timers and C-level interrupt handlers also use the system-stack, the actual stack requirements depend on the application.
The size of the system stack can be changed by modifying the stack size define in your linker file. We recommend a minimum stack size of 256 bytes.

5.3 Interrupt stack

The CPUs do not support a hardware interrupt stack. All interrupts primarily run on the current stack. To reduce the amount of task-stack used by interrupts, embOS uses register bank switching mode for interrupts and supports its own interrupt stack. During the execution of the embOS ISR handler function `OS_CallISR()` and `OS_CallNestableISR()`, embOS automatically switches to the system stack. Only the first level interrupt will use some amount of task stack. At least the return address, the status register, some CPU registers and in case a CPU with FPU is used, the floating point registers are stored on the task stack once.

5.4 Reducing the stack size

The stack check libraries check the used stack of every task and the system stack also. Using embOSView the total size and used size of any stack can be examined. This may be used to reduce the stack sizes, if RAM space is a problem in your application.

If the floating point unit is not used, a CPU without floating point unit may be selected under project options and the embOS libraries without floating point support may be used to reduce the interrupt stack size.

Chapter 6

Interrupts

6.1 Interrupt mode

SH2A CPUs support a priority controlled interrupt mode and as an option an additional register bank switching mechanism. This mode supports the following features:

- Interrupt priority registers to assign 16 priority levels to peripheral interrupts.
- Priority level controlled masking.
- Interrupts with higher priority are never disabled by entering an interrupt service routine with lower priority.
- If bank switching is enabled for the interrupt priority of the current interrupt, the CPU switches to an other register bank.

6.2 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request from the interrupt controller.
- As soon as the interrupts are enabled, the interrupt is accepted and executed.
- The CPU stores the PC and the status register onto the current stack.
- The interrupt mask level in the status register of the CPU is updated from the level of the interrupting device.
- The CPU jumps to the vector address delivered by the ISR.
- If bank switching is enabled for this interrupt, the CPU switches to an other register bank.
- ISR: Save registers if register bank switching is not enabled.
- ISR: User-defined functionality.
- ISR: Restore registers, or restore register bank by switching back to previous bank.
- ISR: Execute RTE command, restoring PC and status register from the stack.
- For more details, refer to the RENESAS manuals.

6.3 Interrupt priorities

With introduction of Zero latency interrupts, interrupt priorities usable by the application are divided into two groups:

- Low priority interrupts with priorities from 1 to a user definable priority limit. These interrupts are called embOS interrupts.
- High priority interrupts with priorities above the user definable priority limit. These interrupts are called **Zero latency interrupts**.
- Interrupt handler functions for both types have to follow the coding guidelines described in the following chapters.
  The priority limit between embOS interrupts and Zero latency interrupts can be set at runtime by a call of `OS_SetFastIntPriorityLimit()`.

6.4 Defining interrupt handlers in C

Routines preceded by the keywords `__interrupt` save & restore the temporary registers and all registers they modify onto the stack and return with RTE. Because embOS enables register bank switching for all interrupts, the compiler has to be informed to add code for resetting the register bank right before the RTE command. Therefore, the option `__fast_interrupt` has to be used in the declaration of interrupt handler functions. The IAR toolchain automatically adds the interrupt vector into the interrupt vector table. The corresponding interrupt vector number has to be declared by a `#pragma` right in front of the interrupt function declaration. The interrupt handler may be implemented in any source file. The interrupt handler used by embOS are implemented in the CPU specific `RTOSInit_*.c` file.

Example of an embOS interrupt handler

embOS interrupt handler have to be used for interrupt sources running at all priorities up to the user definable interrupt priority level limit for Zero latency interrupts.

```
#pragma vector = 142
__fast_interrupt __interrupt void OS_ISR_Tick(void) {
  // __fast_interrupt option required!
  OS_CallNestableISR(_IsrTickHandler);
}
```

Any interrupt handler running at priorities from 1 to the selectable "Zero Latency interrupt" priority limit has to be written according the code example above, regardless any other embOS API function is called.

The rules for an embOS interrupt handler are as follows:

- The embOS interrupt handler **must be defined** with **__fast_interrupt** option.
- The embOS interrupt handler **must not define** any **local variables**.
- The embOS interrupt handler has to call `OS_CallISR()`, when interrupts should not be nested. It has to call `OS_CallNestableISR()`, when nesting should be allowed.
- **The interrupt handler must not perform any other operation, calculation or function call.** This has to be done by the local function called from `OS_CallISR()` or `OS_CallNestableISR()`.

Differences between OS_CallISR() and OS_CallNestableISR()

`OS_CallISR()` should be used as entry function in an embOS interrupt handler, when the corresponding interrupt should not be interrupted by another embOS interrupt. `OS_CallISR()` sets the interrupt priority of the CPU to the user definable "zero latency" interrupt priority level, thus locking any other embOS interrupt. High priority interrupts are not disabled. OS_CallNestableISR() should be used as entry function in an embOS interrupt handler, when interruption by higher prioritized embOS interrupts should be allowed. OS_CallNestableISR() does not alter the interrupt priority of the CPU, thus keeping all interrupts with higher priority enabled.

Example of a Zero Latency interrupt handler

Zero latency interrupt handler have to be used for interrupt sources running at priorities above the user definable interrupt priority limit.

```
#pragma vector = ZeroLatencyInterruptVector {
__fast_interrupt __interrupt void ZeroLatencyInterrupt (void) {
  unsigned long Count;  // local variables are allowed
  Count = TPU_TCNT0;
  HandleCount(Count);   // Any function call except embOS functions is allowed
}
```

The rules for a Zero Latency interrupt handler are as follows:

- Local variables may be used.
- Other functions may be called.
- Register bank switching by prefix __fast_interrupt may be used, but is not required.
- embOS functions must not be called, nor directly, neither indirectly.
- The priority of the interrupt has to be above the user definable priority limit for Zero latency interrupts.

6.5 Interrupt vector table

The IAR toolchain automatically generates the interrupt vector table. The location of the vector table in ROM is defined in the linker settings file. The interrupt vector number has to be assigned to the interrupt handler function by a `#pragma vector` declaration right in front of the interrupt handler function in the source code. The embOS timer interrupt handler is located in the in the source code file `RTOSInit_*.c`.

6.6 Register bank switching

The SH2A-LSI has register banks that enable register saving and restoration required in the interrupt processing to be performed at high speed. The interrupt service routine therefore does not need to push registers onto the stack, if register bank switching is enabled.
When using embOS, register bank switching has to be enabled for all interrupts. The task switch from interrupt relies on the register bank switching in interrupt handler functions.
The initialization sequence which enables register bank switching is included in the `OS_InitHW()` function.

6.7 Zero latency interrupts

Instead of disabling interrupts when embOS does atomic operations, the interrupt level of the CPU is set to a higher user definable level. Therefore all interrupts with higher levels can still be processed. These interrupts are named Zero latency interrupts. The default level limit for zero latency interrupts is set to 8, meaning, any interrupt with level 9 or above is never disabled and can be accepted anytime. You must not execute any embOS function from within a Zero latency interrupt function.

6.8 OS_SetFastIntPriorityLimit(): Set the interrupt priority limit for Zero latency interrupt

The interrupt priority limit for Zero Latency interrupts is set to 8 by default. This means, all interrupts with higher priority from 9 to 15 will never be disabled by embOS.

Description

`OS_SetFastIntPriorityLimit()` is used to set the interrupt priority limit between Zero latency interrupts and lower priority embOS interrupts.

Prototype

```
void OS_SetFastIntPriorityLimit(unsigned int Priority)
```

Parameter	Meaning
`Priority`	The highest value usable as priority for embOS interrupts. All interrupts with higher priority are never disabled by embOS. Valid range: 1 <= `Priority` <= 15

Return value

None.

Additional Information

To disable Zero latency interrupts at all, the priority limit may be set to 15 which is the highest interrupt priority for interrupts. To modify the default priority limit, OS_SetFastIntPriorityLimit() should be called before embOS was started. All interrupts with low priorities from 1 to the user definable priority limit for Zero latency interrupts have to call OS_CallISR() or OS_CallNestableISR() regardless any other embOS function is called in the interrupt handler. This is required, because interrupts with low priorities may be interrupted by other interrupts calling embOS functions. The task switch from interrupt will only work if every embOS interrupt uses the same stack layout. This can only be guaranteed when OS_CallISR() or OS_CallNestableISR() is used. Any interrupts running above the Zero latency interrupt priority limit must not call any embOS function.

Chapter 7

embOS C-Spy plug-in

This chapter gives a short overview about the embOS C-Spy plug-in for IAR Embedded Workbench®.

7.1 Overview

7.1.1 embOS C-Spy plug-in for IAR Embedded Workbench

SEGGER’s embOS C-Spy plug-in for IAR Embedded Workbench provides embOS-awareness during debugging sessions. This enables you to inspect the state of several embOS primitives such as the task list, resource semaphores, mailboxes, and timers.

Since embOS version 3.62, you can check the general-purpose registers and inspect the call stack of all available application tasks.

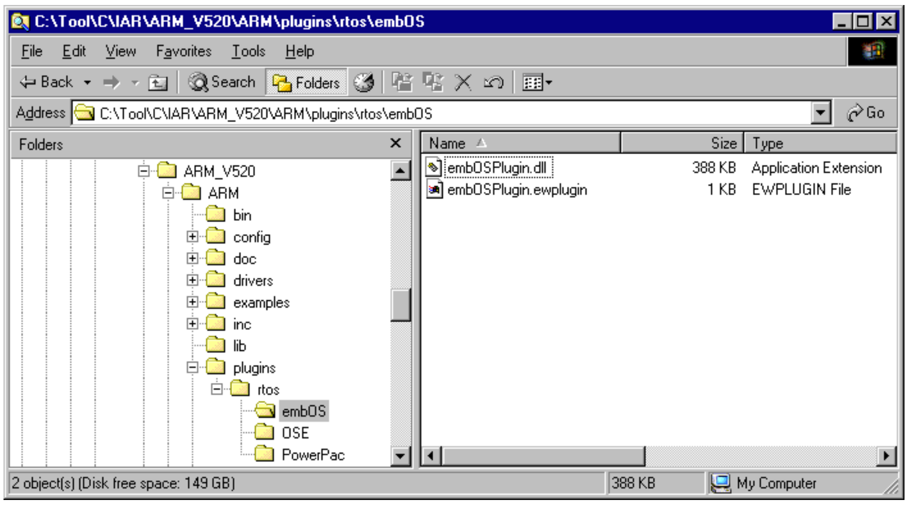
7.1.2 Requirements

To use the embOS C-Spy plug-in you need a version of IAR Embedded Workbench installed and a debug target which uses embOS. Specifically:

- An embOS version 3.62 or higher is required for complete compatibility. Older embOS versions use different internal structures and the C-Spy plug-in is therefore of limited use with version prior to 3.62.
- An IAR Embedded Workbench IDE with a C-SPY debugger version 5.x or higher is required for complete compatibility.

7.2 Installation

The installation procedure is very straightforward because it only requires you to copy the contents of the embOS C-Spy plug-in package into the IAR CPU specific plug-in folder for `rtos` plug-ins. The directory structure may look like this:

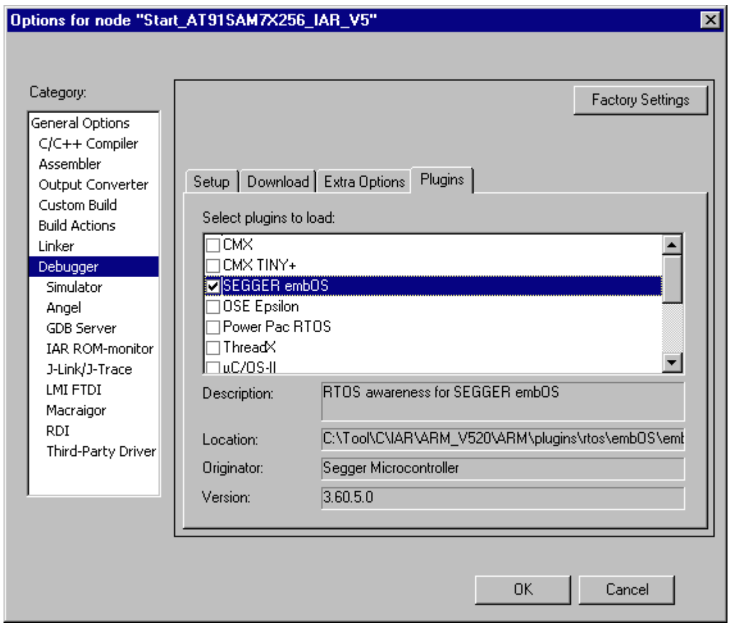

C:\Tool\C\IAR\ARM_V520\ARM\plugins\rtos\embOS
File Edit View Favorites Tools Help
Back
Search
Folders
Address C:\Tool\C\IAR\ARM_V520\ARM\plugins\rtos\embOS
Go
Folders
ARM_V520
ARM
bin
config
doc
drivers
examples
inc
lib
plugins
rtos
embOS
OSE
PowerPac
2 object(s) (Disk free space: 149 GB)
388 KB
My Computer

Name	Size	Type
embOSPlugin.dll	388 KB	Application Extension
embOSPlugin.ewplugin	1 KB	EWPLUGIN File

If not already delivered with the IAR Embedded Workbench IDE, create a directory `embOS` below the CPU specific `plugin\rtos\` folder and copy the files from the embOS folder which comes with the plug-in into that folder in your IAR installation directory. Then restart the IAR Embedded Workbench IDE.

7.3 Configuration

By default, the embOS C-Spy plug-in is not loaded during debugging. For each project configuration you have to explicitly enable the plug-in in the debugger section of the project options:



Select plugins to load:

- [ ] CMX
- [ ] CMX TINY+
- [x] SEGGER embOS
- [ ] OSE Epsilon
- [ ] Power Pac RTOS
- [ ] ThreadX
- [ ] uC/OS-II

Description: RTOS awareness for SEGGER embOS

Location: C:\Tool\C\IAR\ARM_V520\ARM\plugins\rtos\embOS\emt

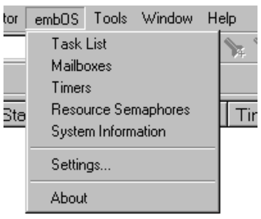
Originator: Segger Microcontroller

Version: 3.60.5.0

The embOS C-Spy plug-in is now available in debugging sessions and may be accessed from the main menu.

7.4 Using the embOS C-Spy plug-in

During your debugging session, the embOS C-Spy plug-in is accessible from the IAR Embedded Workbench IDE **main** menu. Note that if you are not running a debugging session, there is no **embOS menu** item available.

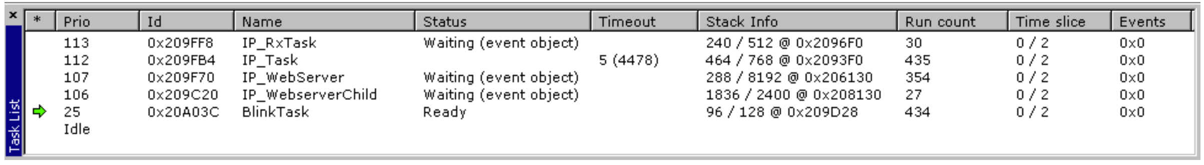


From the menu you may activate the individual windows that provide embOS related information. The sections below describe these individual windows. The amount of information available depends on the embOS build used during debugging. If a certain part is not available, the respective menu item is either greyed out or the window column shows a **N/A**.

7.4.1 Tasks

The **Task List** window lists all embOS tasks. It retrieves its information directly from the embOS task list. The green arrow points to the running task, which is the task currently executing. If no task is executing, the CPU is executing the Idle-loop. In this case, the green arrow is in the bottom row of the window, labeled “Idle”.

The bottom row of the task list window is always labeled “Idle”. It does not actually represent a task, but the Idle loop, which is executed if no task is ready for execution.



*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	113	0x209FF8	IP_RxTask	Waiting (event object)		240 / 512 @ 0x2096F0	30	0 / 2	0x0
	112	0x209FB4	IP_Task		5 (4478)	464 / 768 @ 0x2093F0	435	0 / 2	0x0
	107	0x209F70	IP_WebServer	Waiting (event object)		288 / 8192 @ 0x206130	354	0 / 2	0x0
	106	0x209C20	IP_WebserverChild	Waiting (event object)		1836 / 2400 @ 0x208130	27	0 / 2	0x0
➡	25	0x20A03C	BlinkTask	Ready		96 / 128 @ 0x209D28	434	0 / 2	0x0
	Idle								

The individual columns are described below:

Column	Description
*****	A green arrow points to the running task.
Prio	Priority of the task.
Id	The task control block address that uniquely identifies a task.
Name	If available, the task name is shown here.
Status	The task status as a short text.
Timeout	If a task is delayed, this column shows the time remaining until the delay expires and in parenthesis the time of expiration.
Stack Info	If available, this column shows the amount of used stack space, and the available stack space, as well as the value of the current stack bottom pointer.
Run count	The number of task activations.
Time slice	If round robin scheduling is available, this column shows the number of remaining time slices and the number of time slice reloads.
Events	The event mask of a task.

Table 7.1: Task list window items

7.4.1.1 Task sensitivity

The **Source Code** window, the **Disassembly** window, the **Register** window, and the **Call Stack** window of the C-Spy debugger are task sensitive since version 3.62 of the embOS C-Spy plug-in. This means that they show the position in the code, the general-purpose registers and the call stack of the selected task. By default, the selected task is always the running task, which is the normal behavior of a debugger that the user expects.

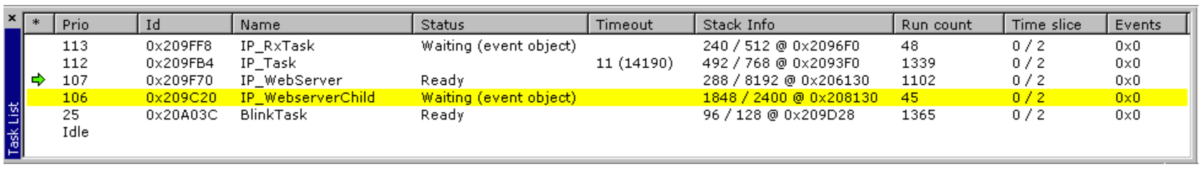
You can examine a particular thread by double-clicking on the corresponding row in the window. The selected task will be underlayed in yellow. The C-Spy Debugger rebuilds the call stack and the preserved general-purpose registers of a suspended task. Refer to *State of suspended tasks* on page 39 for detailed information about which information are available for the different task states.

Every time the CPU is started or when the Idle-row of the task window is double clicked, the selected task is switched back to this default.

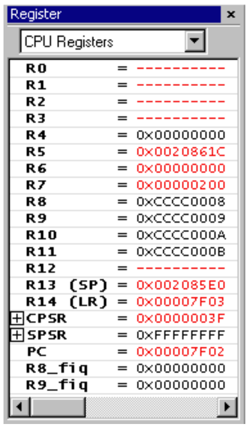
7.4.1.2 State of suspended tasks

Blocked tasks (suspended by cooperative task switch)

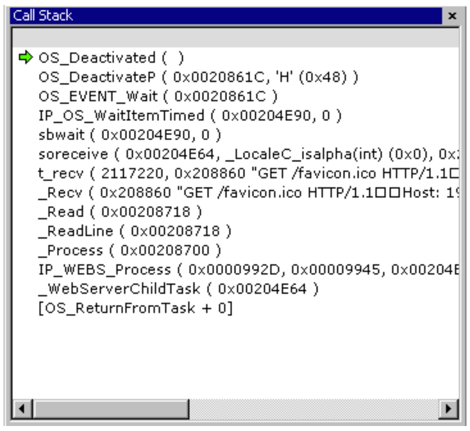
Tasks which have given up execution voluntarily by calling a blocking function, such as `OS_Delay()` or `OS_Wait_...()`. In this case, there was no need for the OS to save the scratch registers (in case of ARM R0-R3, R12).
The **Register** window will show "----------" for the content of these registers.



*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	113	0x209FF8	IP_RxTask	Waiting (event object)		240 / 512 @ 0x2096F0	48	0 / 2	0x0
	112	0x209FB4	IP_Task		11 (14190)	492 / 768 @ 0x2093F0	1339	0 / 2	0x0
➪	107	0x209F70	IP_WebServer	Ready		288 / 8192 @ 0x206130	1102	0 / 2	0x0
	106	0x209C20	IP_WebserverChild	Waiting (event object)		1848 / 2400 @ 0x208130	45	0 / 2	0x0
	25	0x20A03C	BlinkTask	Ready		96 / 128 @ 0x209D28	1365	0 / 2	0x0
	Idle								



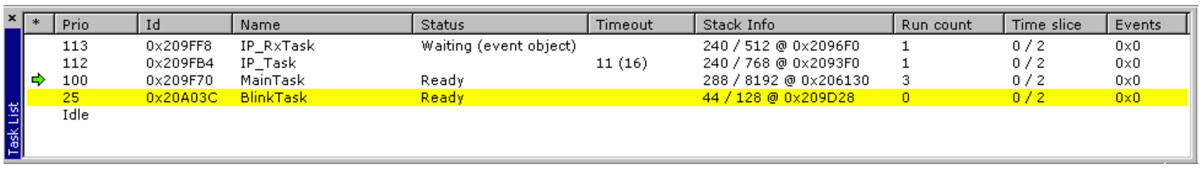
```
R0        = ----------
R1        = ----------
R2        = ----------
R3        = ----------
R4        = 0x00000000
R5        = 0x0020861C
R6        = 0x00000000
R7        = 0x00000200
R8        = 0xCCCC0008
R9        = 0xCCCC0009
R10       = 0xCCCC000A
R11       = 0xCCCC000B
R12       = ----------
R13 (SP)  = 0x002085E0
R14 (LR)  = 0x00007F03
CPSR      = 0x0000003F
SPSR      = 0xFFFFFFFF
PC        = 0x00007F02
R8_fiq    = 0x00000000
R9_fiq    = 0x00000000
```



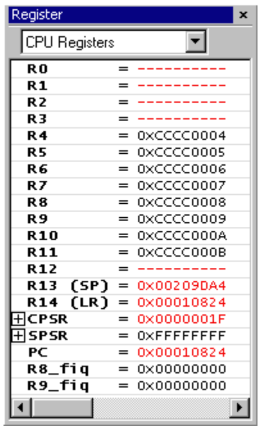
```
OS_Deactivated (  )
OS_DeactivateP ( 0x0020861C, 'H' (0x48) )
OS_EVENT_Wait ( 0x0020861C )
IP_OS_WaitItemTimed ( 0x00204E90, 0 )
sbwait ( 0x00204E90, 0 )
soreceive ( 0x00204E64, _LocaleC_isalpha(int) (0x0), 0x
t_recv ( 2117220, 0x208860 "GET /favicon.ico HTTP/1.1□
_Recv ( 0x208860 "GET /favicon.ico HTTP/1.1□□Host: 1
_Read ( 0x00208718 )
_ReadLine ( 0x00208718 )
_Process ( 0x00208700 )
IP_WEBS_Process ( 0x0000992D, 0x00009945, 0x00204
_WebServerChildTask ( 0x00204E64 )
[OS_ReturnFromTask + 0]
```

Tasks waiting for first activation

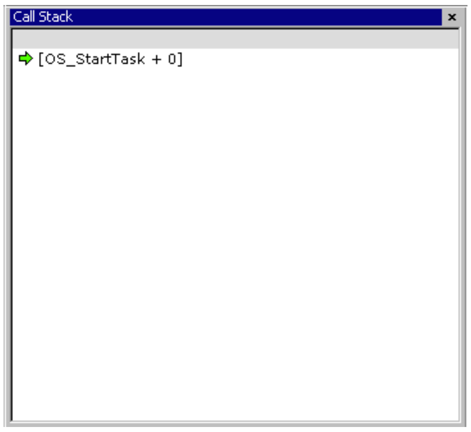
These basically fall into the same category as blocked tasks, the call stack and registers look similar to the following screenshots. Similarly, temporary registers are unknown. The **Call Stack** shows a single entry **OS_StartTask**. **Run count** is 0.



*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	113	0x209FF8	IP_RxTask	Waiting (event object)		240 / 512 @ 0x2096F0	1	0 / 2	0x0
	112	0x209FB4	IP_Task		11 (16)	240 / 768 @ 0x2093F0	1	0 / 2	0x0
➪	100	0x209F70	MainTask	Ready		288 / 8192 @ 0x206130	3	0 / 2	0x0
	25	0x20A03C	BlinkTask	Ready		44 / 128 @ 0x209D28	0	0 / 2	0x0
	Idle								



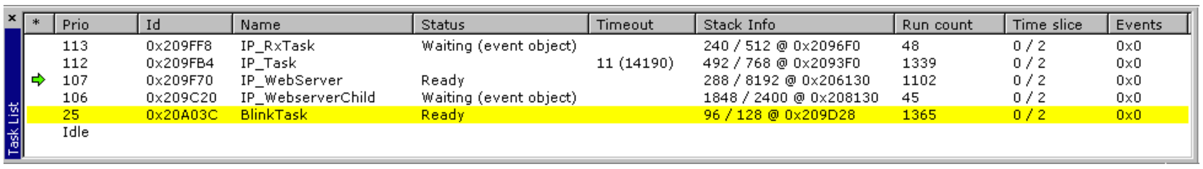
```
R0        = ----------
R1        = ----------
R2        = ----------
R3        = ----------
R4        = 0xCCCC0004
R5        = 0xCCCC0005
R6        = 0xCCCC0006
R7        = 0xCCCC0007
R8        = 0xCCCC0008
R9        = 0xCCCC0009
R10       = 0xCCCC000A
R11       = 0xCCCC000B
R12       = ----------
R13 (SP)  = 0x00209DA4
R14 (LR)  = 0x00010824
CPSR      = 0x0000001F
SPSR      = 0xFFFFFFFF
PC        = 0x00010824
R8_fiq    = 0x00000000
R9_fiq    = 0x00000000
```



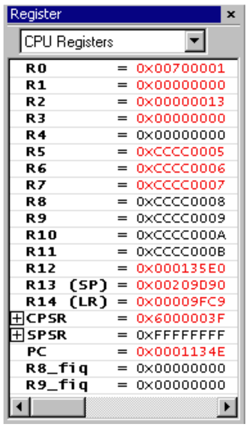
```
[OS_StartTask + 0]
```

Interrupted tasks

Tasks which have been interrupted and preempted, typically by a task with higher priority becoming ready. In this case, the OS saved all registers, including the scratch registers (in case of ARM R0-R3, R12). The **Register** window shows the values of all registers, including the scratch registers.



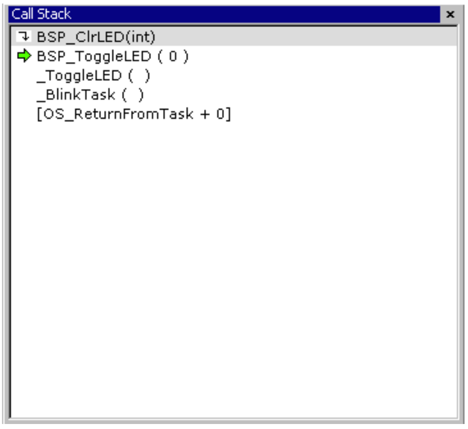
*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	113	0x209FF8	IP_RxTask	Waiting (event object)		240 / 512 @ 0x2096F0	48	0 / 2	0x0
	112	0x209FB4	IP_Task		11 (14190)	492 / 768 @ 0x2093F0	1339	0 / 2	0x0
➡	107	0x209F70	IP_WebServer	Ready		288 / 8192 @ 0x206130	1102	0 / 2	0x0
	106	0x209C20	IP_WebserverChild	Waiting (event object)		1848 / 2400 @ 0x208130	45	0 / 2	0x0
	25	0x20A03C	BlinkTask	Ready		96 / 128 @ 0x209D28	1365	0 / 2	0x0
	Idle								



Register

CPU Registers

R0	= 0x00700001
R1	= 0x00000000
R2	= 0x00000013
R3	= 0x00000000
R4	= 0x00000000
R5	= 0xCCCC0005
R6	= 0xCCCC0006
R7	= 0xCCCC0007
R8	= 0xCCCC0008
R9	= 0xCCCC0009
R10	= 0xCCCC000A
R11	= 0xCCCC000B
R12	= 0x000135E0
R13 (SP)	= 0x00209D90
R14 (LR)	= 0x00009FC9
⊞CPSR	= 0x6000003F
⊞SPSR	= 0xFFFFFFFF
PC	= 0x0001134E
R8_fiq	= 0x00000000
R9_fiq	= 0x00000000

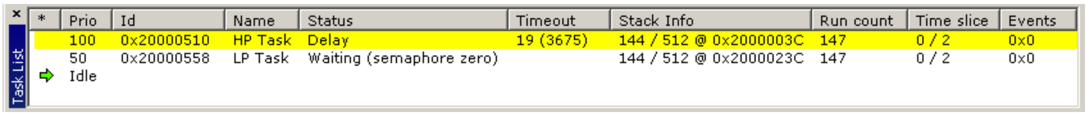


Call Stack

```
BSP_ClrLED(int)
BSP_ToggleLED ( 0 )
_ToggleLED ( )
_BlinkTask ( )
[OS_ReturnFromTask + 0]
```

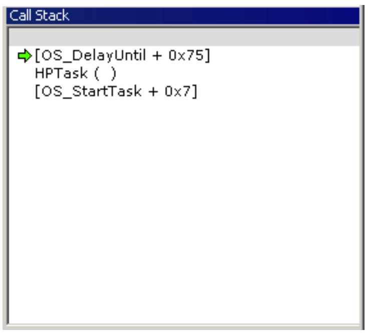
7.4.1.3 Call stack with embOS libraries

All embOS libraries are built with full optimization. Therefore it may happen that not all function calls are shown in the call stack in detail. The additional embOS library *dpl.a is built with low optimization. It may be used for application development instead of the Debug and Profiling library.
This gives the ability to see the complete detailed call stack.



*	Prio	Id	Name	Status	Timeout	Stack Info	Run count	Time slice	Events
	100	0x20000510	HP Task	Delay	19 (3675)	144 / 512 @ 0x2000003C	147	0 / 2	0x0
	50	0x20000558	LP Task	Waiting (semaphore zero)		144 / 512 @ 0x2000023C	147	0 / 2	0x0
➡	Idle								

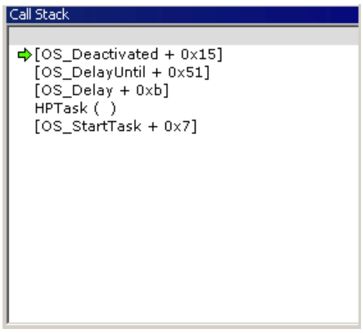
Call stack with DP library



Call Stack

```
[OS_DelayUntil + 0x75]
HPTask ( )
[OS_StartTask + 0x7]
```

Call stack with DPL library

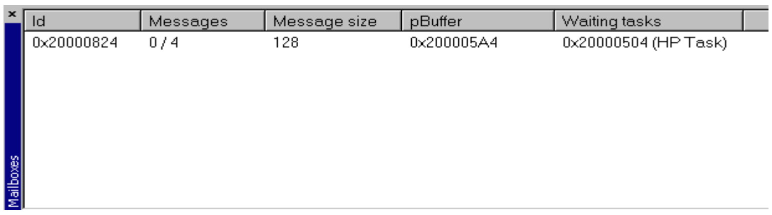


Call Stack

```
[OS_Deactivated + 0x15]
[OS_DelayUntil + 0x51]
[OS_Delay + 0xb]
HPTask ( )
[OS_StartTask + 0x7]
```

7.4.2 Mailboxes

A mailbox is a buffer that is managed by the real-time operating system. The buffer behaves like a normal buffer; you can put something (called a message) in and retrieve it later. This window shows the mailboxes and provides information about the number of messages, waiting tasks etc.



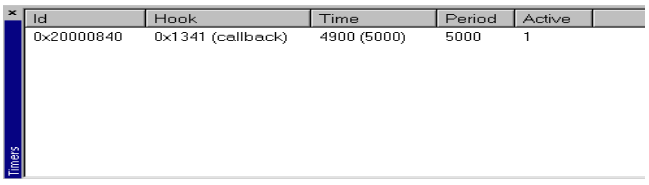
Id	Messages	Message size	pBuffer	Waiting tasks
0x20000824	0 / 4	128	0x200005A4	0x20000504 (HP Task)

Column	Description
Id	The mailbox address.
Messages	The number of messages in a mailbox and the maximum number of messages as mailbox can hold.
Message size	The size of an individual message in bytes.
pBuffer	The message buffer address.
Waiting tasks	The list of tasks that are waiting for a mailbox, that is their address and name.

Table 7.2: Mailboxes window items

7.4.3 Timers

A software timer is an object that calls a user-specified routine after a specified delay. This window provides information about active software timers.



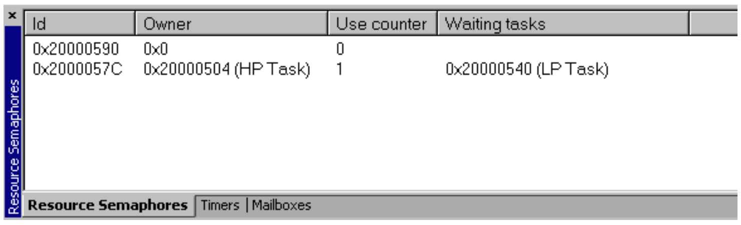
Id	Hook	Time	Period	Active
0x20000840	0x1341 (callback)	4900 (5000)	5000	1

Column	Description
Id	The timer's address.
Hook	The function (address and name) that is called after the timeout.
Time	The time delay and the point in time, when the timer finishes waiting.
Period	The time period the timer runs.
Active	Shows whether the timer is active or not.

Table 7.3: Timers window items

7.4.4 Resource semaphores

Resource semaphores are used to manage resources by avoiding conflicts caused by simultaneous use of a resource. This window provides information about available resources.



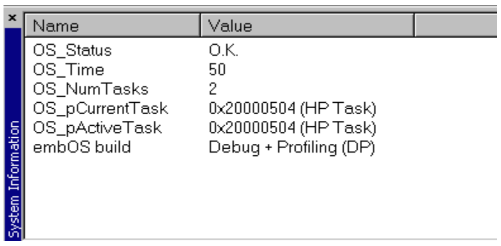
Id	Owner	Use counter	Waiting tasks
0x20000590	0x0	0	
0x2000057C	0x20000504 (HP Task)	1	0x20000540 (LP Task)

Column	Description
Id	The resource semaphore address.
Owner	The address and name of the owner task.
Use counter	Counts the number of semaphore uses.
Waiting tasks	Lists the tasks (address and name) that are waiting at the semaphore.

Table 7.4: Resource Semaphores window items

7.4.5 System information

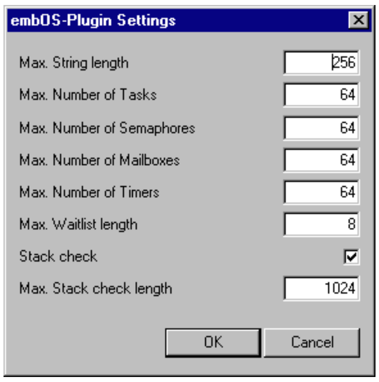
A running embOS contains a number of system variables that are available for inspection. This window lists the most important ones.



Name	Value
OS_Status	O.K.
OS_Time	50
OS_NumTasks	2
OS_pCurrentTask	0x20000504 (HP Task)
OS_pActiveTask	0x20000504 (HP Task)
embOS build	Debug + Profiling (DP)

7.4.6 Settings

To be safe, the embOS C-Spy plug-in imposes certain limits on the amount of information retrieved from the target, to avoid endless requests in case of false values in the target memory. This dialog box allows you to tweak these limits in a certain range, for example if your task names are no longer than 32 characters you may set the **Maximum string length** to 32, or if they are longer than the default you may increase that value.



embOS-Plugin Settings

Max. String length	256
Max. Number of Tasks	64
Max. Number of Semaphores	64
Max. Number of Mailboxes	64
Max. Number of Timers	64
Max. Waitlist length	8
Stack check	✓
Max. Stack check length	1024

After changing settings and clicking the **OK** button, your changes are applied immediately and should become noticeable after the next window update, for example when hitting the next breakpoint. However, the settings are restored to their default values on plug-in reload.

7.4.7 About

Finally, the **About** dialog box contains the embOS C-Spy plug-in version number and the date of compilation.

Chapter 8

Technical data

This chapter lists technical data of embOS used with the SH2A cpu.

8.1 Memory requirements

These values are neither precise nor guaranteed, but they give you a good idea of the memory requirements. They vary depending on the current version of embOS. The minimum ROM requirement for the kernel itself is about 2.000 bytes.

In the table below, which is for release build, you can find minimum RAM size requirements for embOS resources. Note that the sizes depend on selected embOS library mode.

embOS resource	RAM [bytes]
Kernel	75
Task	32
Semaphore	16
Mailbox	24
Software timer	20

Index