embOS

Real Time Operating System

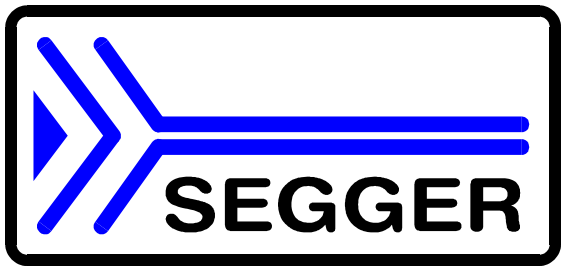
CPU & Compiler specifics for

RENESAS M16C/R8C CPUs

used

with HEW and GNU compiler

Document Rev. 1



A product of Segger Microcontroller Systeme GmbH

www.segger.com

Contents

1. About this document

This guide describes how to use ***embOS*** for M16C/R8C Real Time Operating System for the RENESAS M16C/R8C series of microcontroller using RENESAS HEW version 4 and KPIT GNU compiler.

1.1. How to use this manual

This manual describes all CPU and compiler specifics for ***embOS*** using M16C/R8C CPUs with GNU compiler. Before actually using ***embOS***, you should read or at least glance through this manual in order to become familiar with the software.
Chapter 2 gives you a step-by-step introduction, how to install and use ***embOS*** using RENESAS High-performance Embedded Workshop HEW. If you have no experience using ***embOS***, you should follow this introduction, even if you do not plan to use RENESAS HEW, because it is the easiest way to learn how to use ***embOS*** in your application.
Most of the other chapters in this document are intended to provide you with detailed information about functionality and fine-tuning of ***embOS*** for the M16C/R8C CPUs and GNU compiler.

2. Using *embOS* with RENESAS HEW

2.1. Installation

embOS is shipped on CD-ROM or as a zip-file in electronic form.

In order to install it, proceed as follows:

If you received a CD, copy the entire contents to your hard-drive into any folder of your choice. When copying, please keep all files in their respective sub directories. Make sure the files are not read only after copying.
If you received a zip-file, please extract it to any folder of your choice, preserving the directory structure of the zip-file.

Assuming that you are using RENESAS HEW to develop your application, no further installation steps are required. You will find a prepared sample workspace and sample start project for M16C CPU, which you should use and modify to write your application. So follow the instructions of the next chapter 'First steps'.

You should do this even if you do not intend to use RENESAS HEW for your application development in order to become familiar with ***embOS.***

embOS does in no way rely on RENESAS HEW, it may be used without the workbench using batch files or a make utility without any problem.

2.2. First steps

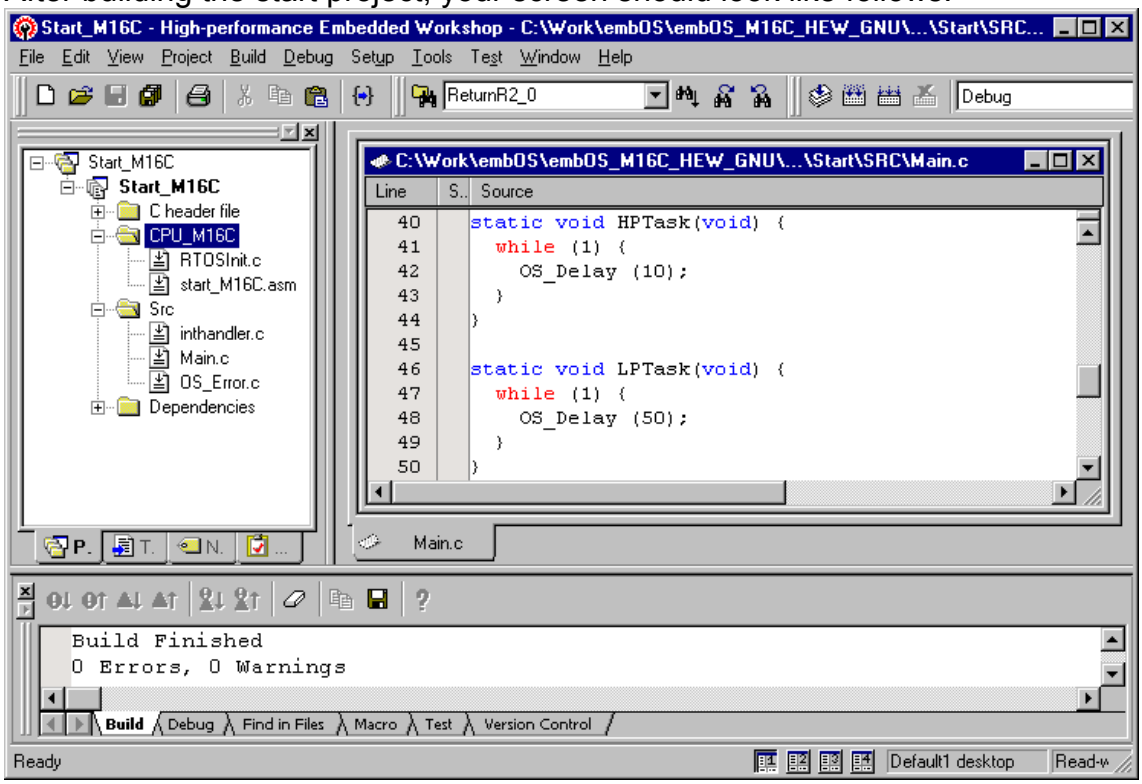
After installation of ***embOS*** (→ Installation) you are able to create your first multitasking application. You received a ready to go sample start project for M16C CPUs and it is a good idea to use this as a starting point of all your applications.

Your ***embOS*** distribution contains everything you need for NC30 compiler version 5.40 and HEW version 4. If you have to use compiler version 5.3 or lower and RENESAS Tool manager, the sample start workspace can not be used.

For NC30 compiler version 5.40 and HEW version 4 which is explained in this manual, you should:

- Create a work directory for your application, for example c:\work
- Copy all files and subdirectories from the folder 'embOS_M16C_NC30_V540_HEW' from your ***embOS*** distribution into your work directory.
- Clear the read only attribute of all files in the new 'Start'-folder in your working directory.
- Open the folder 'Start'
- Open the start workspace 'Start_M16C.hws'. (e.g. by double clicking it)
- Build the start project

After building the start project, your screen should look like follows:



2.3. The sample application Main.c

The following is a printout of the sample application main.c. It is a good starting-point for your application.
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.

```
/*********************************************************************
*               SEGGER MICROCONTROLLER SYSTEME GmbH                  *
*       Solutions for real time microcontroller applications         *
**********************************************************************
*                                                                    *
*       (C) 2006   SEGGER Microcontroller Systeme GmbH               *
*                                                                    *
*       www.segger.com     Support: support@segger.com               *
*                                                                    *
**********************************************************************

----------------------------------------------------------------------
File    : Main.c
Purpose : Skeleton program for embOS
--------  END-OF-HEADER  ---------------------------------------------
*/

#include "RTOS.H"

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

static void HPTask(void) {
  while (1) {
    OS_Delay (10);
  }
}

static void LPTask(void) {
  while (1) {
    OS_Delay (50);
  }
}

/*********************************************************************
*
*       main
*
*********************************************************************/

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

3. M16C and GNU compiler specifics

3.1. Memory models

embOS supports all memory models that the KPIT GNU compiler supports.
For M16C there is only one memory models available:

Model	Code	Data
Near	far (20 bits always)	near (16 bits)

3.2. Available libraries

The files to use are:

Memorymodel	Library type	Library	define
Near	Release	osNR	OS_LIBMODE_R
Near	Stack-check	osNS	OS_LIBMODE_S
Near	Stack-check + Profiling	osNSP	OS_LIBMODE_SP
Near	Debug	osND	OS_LIBMODE_D
Near	Debug + Profiling	osNDP	OS_LIBMODE_DP
Near	Trace + Debug	osNDT	OS_LIBMODE_DT

When using RENESAS HEW, please check the following points:

- One ***embOS*** library is part of your project (included in the library list under Project Options | LFLAGS)
- The appropriate define is set as compiler option for your project.

3.3. Distributed project files

The distribution of ***embOS*** contains one start workspace and one start project which is set up to use the Debug+Profiling library.

3.4. M306N CPU specifics

The M16C/6N group of CPUs require modification affecting RTOSInit.c. The hardware initialization routines and default settings in RTOSInit.c were designed for M16C/62 CPUs.
M306N CPUs have additional prescalers that are activated per default after reset and divide the peripheral clock for timer and UART by two.
This results in wrong settings for ***embOS*** timer tick and baudrate for UART used for embOSView.
There are different solutions to correct these settings.

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***.

You may reprogram the Peripheral function clock select register (PCLKR) at address 0x025E to disable the prescaler for timer and UART, before calling `OS_InitHW()`. This could be done during your own target specific hardware initialization. The protection register bit 0 has to be set to enable modification of PCLKR

When PCLKR is left unchanged (reset value = 0x00), CPUs internal timer A0 and UART clock is derived from CPU clock divided by two.
If you do not want to disable (reprogram) the prescaler for UART or timer, you may define different values for `OS_PCLK_TIMER` and `OS_PCLK_UART` as compiler / project option without any changes in RTOSInit.c
`OS_PCLK_TIMER` is the frequency of CPUs internal peripheral clock used for the timer. Calculations of timer reload value is derived from this define. Without modification or override, it is defined to OS_FSYS.
`OS_PCLK_UART` is the frequency of CPUs internal peripheral clock used for UARTs. Calculations of baudrate generator value is derived from this define. Without modification or override, it is defined to OS_FSYS.

3.5. M16C/62P CPU specifics

M16C/62P CPUs come with a built in PLL.
The initialization routine `OS_InitHW()` for timer initialization is written for generic M16C CPUs and can also be used for M16C/62P CPUs.
If you want to use the PLL, you will have to modify the initialization sequence for CPU clock mode setting in `OS_InitHW()`.
You might also have to modify the clock mode initialization when you decide to use the KD30 ROM monitor for debugging. If the monitor is set up to initialize the PLL, the system may stop working when the CPU clock mode is modified during `OS_InitHW()`.

3.6. Startup file start_M16C.asm

embOS. comes with a modified startup file for M16C. Minor modifications of the original startup files are required; they are documented in this file.

3.7. Section definition file start_M16C.sec

The section file is required to export information about stack sizes and stack start and end addresses which are needed for ***embOS*** stack check.
When building a new project, the linker options have to be modified to use this file as “Sections only (script file)”.
Depending on the specific CPU derivate, the RAM and ROM address and size definitions will have to be modified.

4. Stacks

4.1. Task stack for M16C

Every ***embOS*** task has to have its own stack. Task stacks can be located in any RAM memory location that can be used as stack by the M16C CPU.
As M16C CPUs have a 16bit stack pointer only, this may be any RAM located from 0x0000..0xFFFF.
The stack-size required is the sum of the stack-size of all routines plus basic stack size.
The basic stack size is the size of memory required to store the registers of the CPU plus the stack size required by ***embOS*** -routines.
For the M16C, this minimum task stack size is about 42 bytes.

4.2. System stack for M16C

The system stack size required by ***embOS*** is about 40 bytes (65 bytes in. profiling builds) 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 also use the system-stack, the actual stack requirements depend on the application.
The stack used as system stack is the one defined at startup. Its size is defined as `STACKSIZE` in the ncrt0.a30 start up file.
A good value for the system stack is typically about 80 to 200 bytes.

4.3. Interrupt stack for M16C

The M16C CPU has been designed with multitasking in mind; it has 2 stack-pointers, the USP and the ISP. The U-Flag selects the active stack-pointer. During execution of a task or timer, the U-flag is set thereby selecting the user-stack-pointer. If an interrupt occurs, the M16C clears the U-flag and switches to the interrupt-stack-pointer automatically this way. The ISP is active during the entire ISR (interrupt service routine). This way, the interrupt does not use the stack of the task and the task-stack-size does not have to be increased for interrupt-routines. Additional stack-switching as for other CPUs is therefore not necessary for the M16C CPUs.
The interrupt stack size is defined as `_ISTACKSIZE` in the section definition file.
The interrupt stack size depends on the interrupt handler functions and the nesting level of interrupts. A minimum stack size of 256 bytes is required, we recommend a a stack size of 512 bytes.

4.4. Reducing the stack size

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

5. Interrupts

5.1. What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the interrupts are enabled and the processor interrupt priority level is below or equal to the interrupt priority level, the interrupt is executed
- the CPU switches to the Interrupt stack
- the CPU saves PC and flags on the stack
- the IPL is loaded with the priority of the interrupt
- the CPU jumps to the address specified in the vector table for the interrupt service routine (ISR)
- ISR : save registers
- ISR : user-defined functionality
- ISR : restore registers
- ISR: Execute REIT command, restoring PC, Flags and switching to User stack
- For details, please refer to the RENESAS users manual.

5.2. Defining interrupt handlers in "C"

Routines declared with the keywords __attribute__ ((interrupt)) automatically save & restore the registers they modify and return with REIT.
All interrupt functions are declared in the header file “inthandler.h”.
As the interrupt sources may be device specific, this file may have to be modified according to the specific target CPU.
The interrupt handler functions itself are implemented in the interrupt functions file inthandler.c
You may add your code directly into the predefined interrupt handler functions, or may call your own interrupt handler from there.

5.3. Writing interrupt handlers which call embOS functions

Before any embOS API function is called, embOS has to be informed that an interrupt handler is running. This is required to inhibit immediate task switches and is realized by an “EnterInterrupt” function.
A pending task switch will the be executed at the end of the interrupthandler during execution of the “LeaveInterrupt” function.

Example

"Simple" interrupt-routine which calls an embOS function

```
#pragma INTERRUPT OS_ISR_tx
void OS_ISR_tx(void) {
  OS_EnterNestableInterrupt();  //  We will enable interrupts
  OS_SignalEvent(&TCBrx, 0x01);
  OS_LeaveNestableInterrupt();
}
```

5.4. Interrupt vector table

The interrupt vectors are defined in the assembly file vects.asm which may have to be modified according to the specific target CPU.

5.5. Interrupt-stack

Since the M16C CPUs have a separate stack pointer for interrupts, there is no need for explicit stack-switching in an interrupt routine. The routines `OS_EnterIntStack()` and `OS_LeaveIntStack()` are supplied for source compatibility to other processors only and have no functionality.

5.6. Fast interrupts with M16C

Instead of disabling interrupts when ***embOS*** does atomic operations, the interrupt level of the CPU is set to 4. Therefore all interrupts with level 5 or above can still be processed.
These interrupts are named *Fast interrupts*. You must not execute any ***embOS*** function from within a *fast interrupt* function.

5.7. Interrupt priorities

With introduction of *Fast interrupts*, interrupt priorities useable for interrupts using ***embOS*** API functions are limited.

- Any interrupt handler using ***embOS*** API functions has to run with interrupt priorities from 1 to 4. These ***embOS*** interrupt handlers have to start with `OS_EnterInterrupt()` or `OS_EnterNestableInterrupt()` and must end with `OS_LeaveInterrupt()` or `OS_LeaveNestableInterrupt()`.
- Any *Fast interrupt* (running at priorities from 5 to 7) must not call any ***embOS*** API function. Even `OS_EnterInterrupt()` and `OS_LeaveInterrupt()` must not be called.
- Interrupt handler running at low priorities (from 1 to 4) not calling any ***embOS*** API function are allowed, but must not re-enable interrupts!

The priority limit between *embOS* interrupts and Fast interrupts is fixed to 4 and can only be changed by modification of the source code and re-compiling *embOS* libraries!

6. STOP / WAIT Mode

Usage of the wait instruction is one possibility to save power consumption during idle times. If required, you may modify the `OS_Idle()` routine, which is part of the hardware dependent module RtosInit.c.

The stop-mode works without a problem; however the real-time operating system is halted during the execution of the stop-instruction if the timer that the scheduler uses is supplied from the internal clock. With external clock, the scheduler keeps working.

7. Technical data

7.1. 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 values in the table are for the near memory model and release build library.

Short description	ROM [byte]	RAM [byte]
Kernel	approx.1670	27
Add. Task	---	19
Add. Semaphore	---	4
Add. Mailbox	---	11
Add. Timer	---	11
Power-management	---	---

8. Files shipped with *embOS* for NC30 compiler

embOS for M16C/R8C and NC30 compiler is shipped for compiler version 5.40 and projects for HEW version 4.
This version of ***embOS*** is located in Folder “embOS_M16C_NC30_V540” and contains the following files:

Directory	File	Explanation
Start\	`Start_M16C.hws`	Start workspace for HEW 4
Start\	`PD30_SimInt21.scr`	IO script file for ***embOS*** timer interrupt simulation
Start\Start_M16c\	`*.*`	Project files for HEW
Start\INC\	`RTOS.h`	***embOS*** API header file. To be included in any file using ***embOS*** functions
Start\INC\	`CPUM16C.h`	SFR definition file for M16C CPU
Start\Lib	`*.a`	***embOS*** libraries
Start\Src\	`main.c`	Frame program to serve as a start
Start\Src\	`inthandler.*`	Interrupt handler sources and definitions
Start\Src	`OS_Error.c`	The ***embOS*** error handler, used called on runtime error occurrence in debug builds.
Start\Src\	`vects.asm`	Interrupt vector table
Start\Src\CPU_M16C	`RTTOSInit.c`	CPU specific functions required for ***embOS***
Start\Src\CPU_M16C	`start_M16C.asm`	startup code used with ***embOS***
Start\Src\CPU_M16C	`start_M16C.sec`	Section definition file required for ***embOS***

embOSView and the manuals are found in the root directory of the distribution.

9. Index