embOS

Real Time Operating System

CPU & Compiler specifics for NEC V850/V850E/V850ES cores using NEC compiler for V850

Software version 3.52
Document revision 1
Date: September 27, 2007

SEGGER

A product of SEGGER Microcontroller Systeme GmbH

www.segger.com

Disclaimer

Copyright notice

Trademarks

Contact address

SEGGER Microcontroller Systeme GmbH

Heinrich-Hertz-Str. 5
D-40721 Hilden

Germany

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

Manual versions

This manual describes the latest software version. If any error occurs, please inform us and we will try to assist you as soon as possible.

For further information on topics or routines not yet specified, please contact us.

Manual version	Date	By	Explanation
1.00	070927	OO	Initial version

Software versions

Refer to Release.html for information about the changes of the software versions.

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.
Reference	Reference to chapters, sections, tables and figures or other documents.
GUIElement	Buttons, dialog boxes, menu names, menu commands.
Emphasis	Very important sections

Table 1.1: Typographic conventions

j-link

Table of Contents

Chapter 1

Introduction

This guide describes how to use **embOS** Real Time Operating System for the NEC V850 series of microcontrollers using NEC compiler for V850 and NEC’s PM+ Workbench.

How to use this manual

This manual describes all CPU and compiler specifics of **embOS** using NEC based controllers with NEC compiler for V850 and NEC’s PM+ Workbench. 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** for V850 using NEC’s PM+ Workbench. If you have no experience using **embOS**, you should follow this introduction, 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 NEC based controllers using NEC’s PM+ Workbench.

Chapter 2

Using embOS with NEC's PM+ Workbench

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.

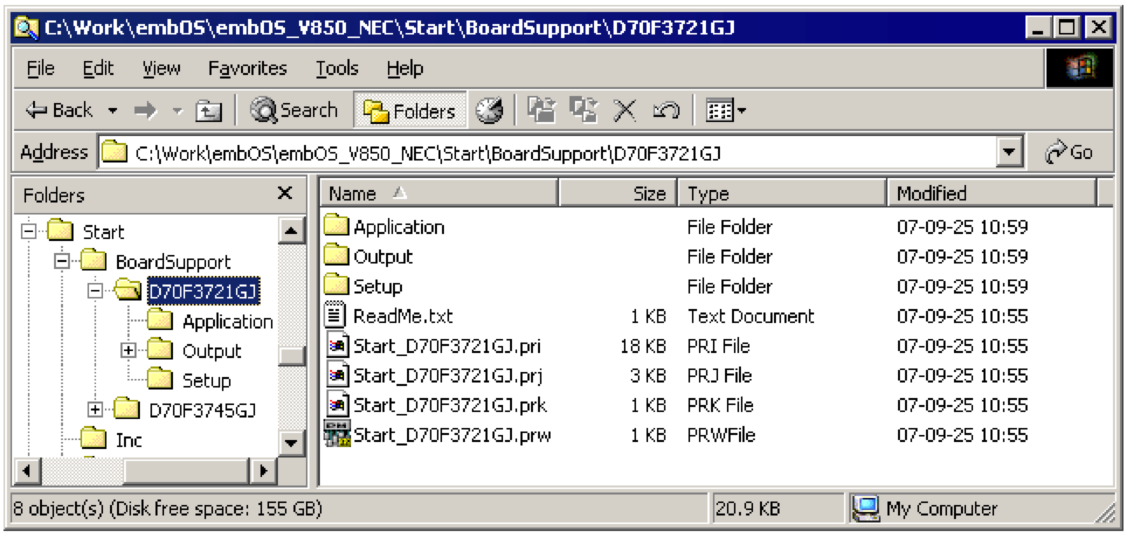
Assuming that you are using NEC's PM+ Workbench to develop your application, no further installation steps are required. You will find a prepared sample start application, which you should use and modify to write your application. So follow the instructions of the next heading *First steps* on page 11.

You should do this even if you do not intend to use NEC's PM+ Workbench for your application development in order to become familiar with **embOS**.

If for some reason you will not work with NEC's PM+ Workbench, you should:
Copy either all or only the library-file that you need to your work-directory. Also copy the entire CPU specific subdirectory and the **embOS** header file `RTOS.h`. This has the advantage that when you switch to an updated version of **embOS** later in a project, you do not affect older projects that use **embOS** also.
embOS does in no way rely on NEC's PM+ Workbench, 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** (See “Installation” on page 10.) you are able to create your first multitasking application. You 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 CPU-specific subfolders:


C:\Work\embOS\embOS_V850_NEC\Start\BoardSupport\D70F3721GJ
File Edit View Favorites Tools Help
Back Search Folders
Address C:\Work\embOS\embOS_V850_NEC\Start\BoardSupport\D70F3721GJ
Go
Folders
Start
BoardSupport
D70F3721GJ
Application
Output
Setup
D70F3745GJ
Inc

Name	Size	Type	Modified
Application		File Folder	07-09-25 10:59
Output		File Folder	07-09-25 10:59
Setup		File Folder	07-09-25 10:59
ReadMe.txt	1 KB	Text Document	07-09-25 10:55
Start_D70F3721GJ.pri	18 KB	PRI File	07-09-25 10:55
Start_D70F3721GJ.prj	3 KB	PRJ File	07-09-25 10:55
Start_D70F3721GJ.prk	1 KB	PRK File	07-09-25 10:55
Start_D70F3721GJ.prw	1 KB	PRWFile	07-09-25 10:55

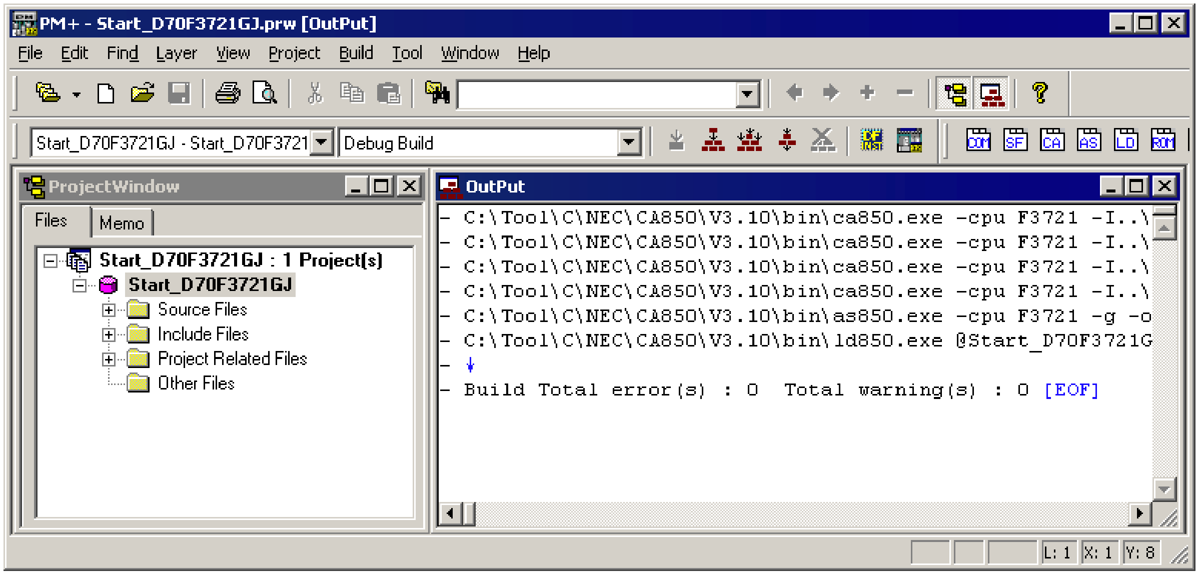
8 object(s) (Disk free space: 155 GB) 20.9 KB My Computer

The screenshot above shows the start workspace and project for an NEC D70F3721GJ, which is located under "`Start\BoardSupport\D70F3721GJ`".

To get your first multitasking 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 a sample workspace e.g. "`Start\BoardSupport\D70F3721GJ\Start_D70F3721GJ.prw`" with the NEC’s PM+ (e.g. by double clicking it)
- PM+ may complain about a missing device file (this occurs if the toolchain installation folder has changed). In this case open the menu "Project->Project Settings..." and select the proper device.
- Build the start project

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


PM+ - Start_D70F3721GJ.prw [OutPut]
File Edit Find Layer View Project Build Tool Window Help
Start_D70F3721GJ - Start_D70F3721
Debug Build
ProjectWindow
Files
Memo
Start_D70F3721GJ : 1 Project(s)
Start_D70F3721GJ
Source Files
Include Files
Project Related Files
Other Files
OutPut
L: 1 X: 1 Y: 8

```
- C:\Tool\C\NEC\CA850\V3.10\bin\ca850.exe -cpu F3721 -I..\
- C:\Tool\C\NEC\CA850\V3.10\bin\ca850.exe -cpu F3721 -I..\
- C:\Tool\C\NEC\CA850\V3.10\bin\ca850.exe -cpu F3721 -I..\
- C:\Tool\C\NEC\CA850\V3.10\bin\ca850.exe -cpu F3721 -I..\
- C:\Tool\C\NEC\CA850\V3.10\bin\as850.exe -cpu F3721 -g -o
- C:\Tool\C\NEC\CA850\V3.10\bin\ld850.exe @Start_D70F3721G
- ↓
- Build Total error(s) : 0  Total warning(s) : 0 [EOF]
```

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...

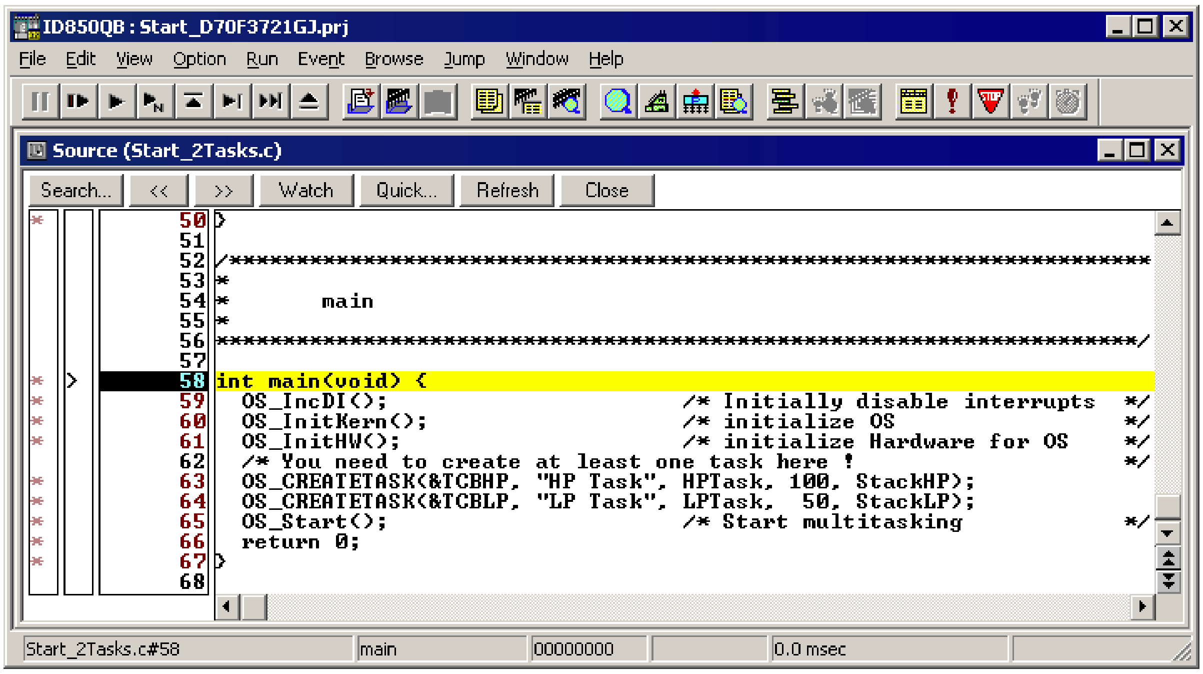
2.3 The sample application Main.c

The following is a printout of the sample application `main.c`. It is a good startingpoint for your application. (Please 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.

```
/*********************************************************
*           SEGGER MICROCONTROLLER SYSTEME GmbH          *
*  Solutions for real time microcontroller applications *
**********************************************************
File    : Main.c
Purpose : Skeleton program for OS
--------------END-OF-HEADER----------------------------*/
#include "RTOS.H"

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

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

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;
}
```

2.4 Stepping through the sample application using NEC's ID850QB debugger

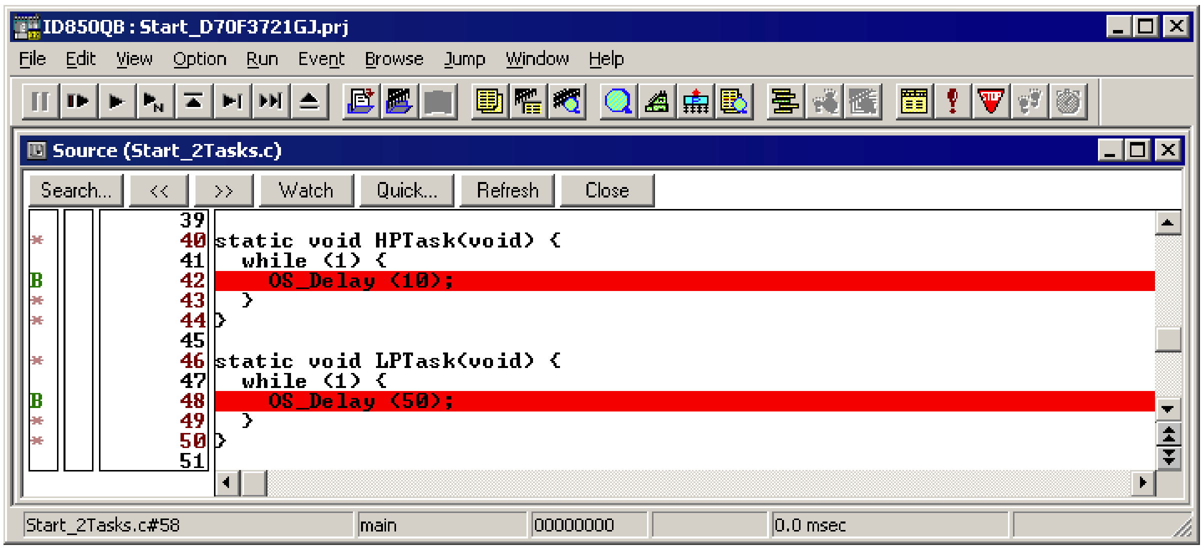
When starting the debugger, you will usually see the main function (very similar to the screenshot below). In some debuggers, you may look at the startup code and have to set a breakpoint at main. Now you can step through the program. `OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part of the **embOS** library; 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 timer-tick-interrupt for **embOS**. Step through it to see what is done.
`OS_Start()` should be the last line in main, since it starts multitasking and does not return.


ID850QB : Start_D70F3721GJ.prj
File Edit View Option Run Event Browse Jump Window Help
Source (Start_2Tasks.c)
Search... << >> Watch Quick... Refresh Close

```
}

/*********************************************************************
*
*       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;
}
```

```
Start_2Tasks.c#58    main    00000000    0.0 msec
```

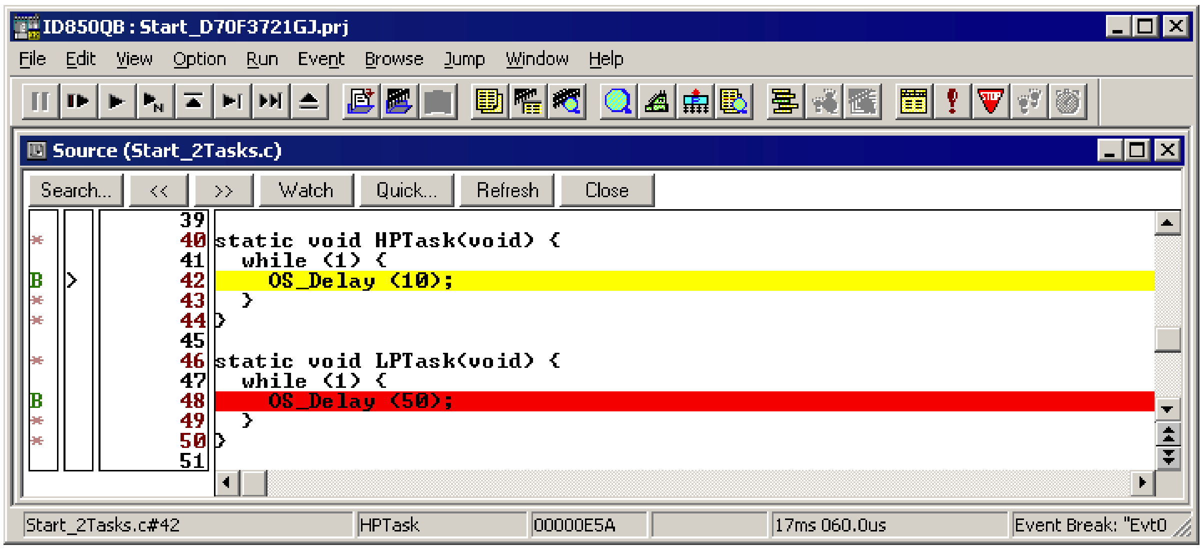
Before you step into `OS_Start()`, you should set two break points in the two tasks as shown below.


ID850QB : Start_D70F3721GJ.prj
File Edit View Option Run Event Browse Jump Window Help
Source (Start_2Tasks.c)
Search... << >> Watch Quick... Refresh Close

```

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

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

```
Start_2Tasks.c#58    main    00000000    0.0 msec
```

As `OS_Start()` is part of the **embOS** library, you can step through it in disassembly mode only. You may press `GO`, step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task.

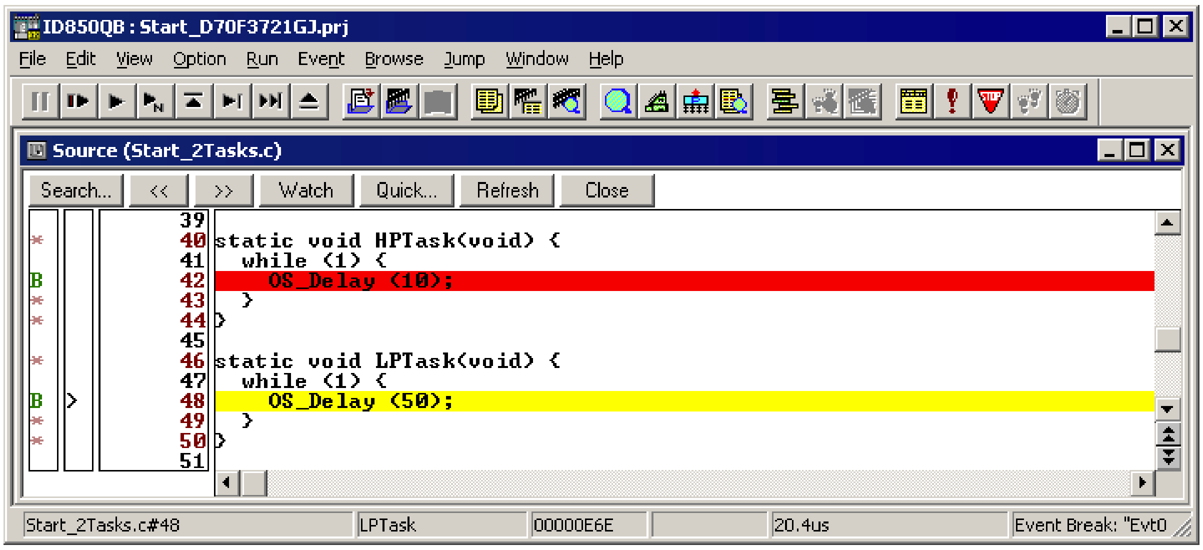

ID850QB : Start_D70F3721GJ.prj

Source (Start_2Tasks.c)

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

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

Start_2Tasks.c#42 | HPTask | 00000E5A | | 17ms 060.0us | Event Break: "Evt0

If you continue stepping, you will arrive in the task with lower priority:


ID850QB : Start_D70F3721GJ.prj

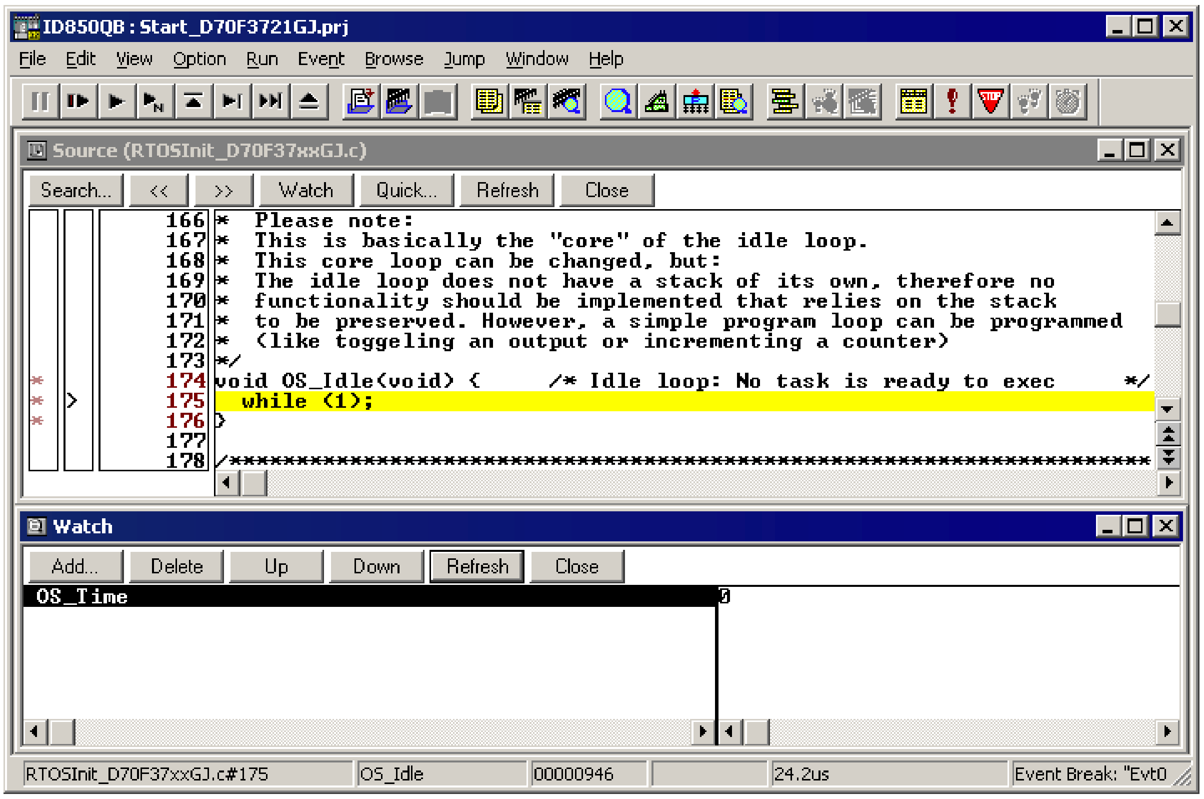
Source (Start_2Tasks.c)

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

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

Start_2Tasks.c#48 | LPTask | 00000E6E | | 20.4us | Event Break: "Evt0

Continuing 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 which is 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 you step over the delay in LPTask.

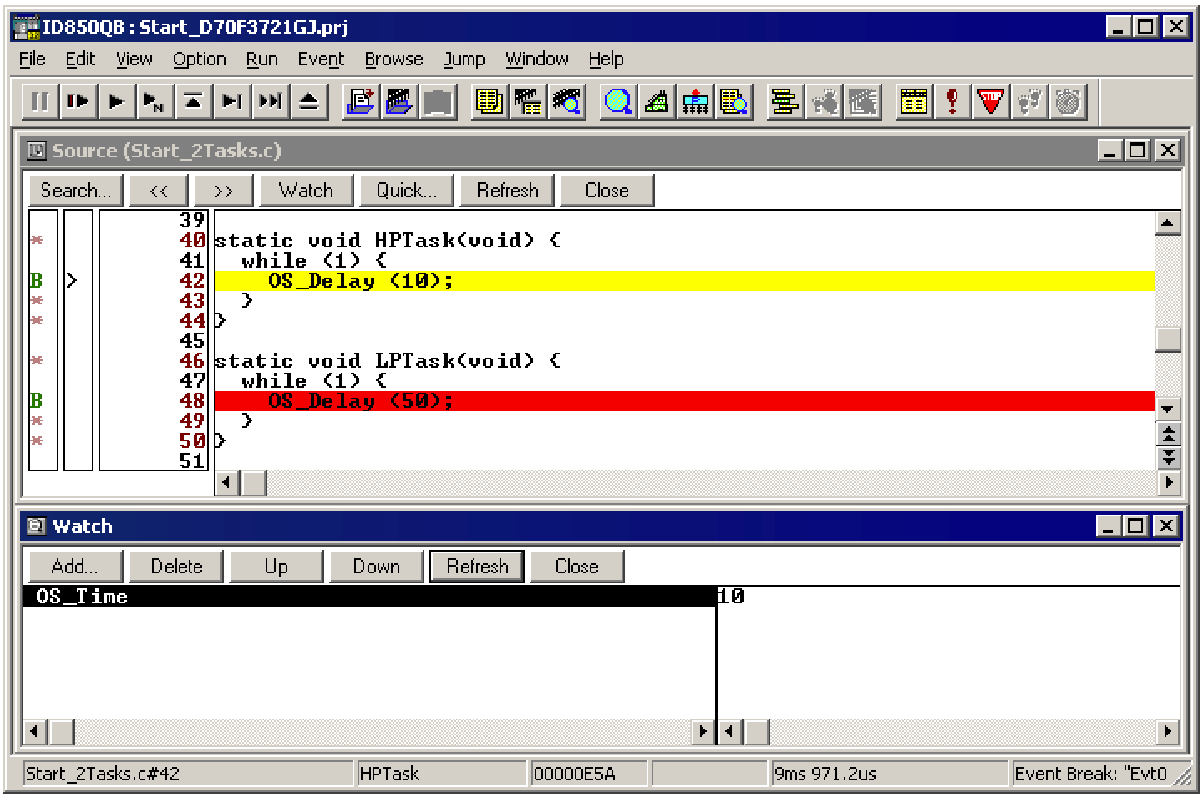

ID850QB : Start_D70F3721GJ.prj
File Edit View Option Run Event Browse Jump Window Help
Source (RTOSInit_D70F37xxGJ.c)
Search... << >> Watch Quick... Refresh Close
Watch
Add... Delete Up Down Refresh Close

```
* 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 toggling an output or incrementing a counter)
*/
void OS_Idle(void) {     /* Idle loop: No task is ready to exec      */
  while (1);
}

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

```
OS_Time   0
```

```
RTOSInit_D70F37xxGJ.c#175 | OS_Idle | 00000946 | | 24.2us | Event Break: "Evt0
```

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_Time`, shown in the watch window, HPTask continues operation after expiration of the 10 ms delay.


ID850QB : Start_D70F3721GJ.prj
File Edit View Option Run Event Browse Jump Window Help
Source (Start_2Tasks.c)
Search... << >> Watch Quick... Refresh Close
Watch
Add... Delete Up Down Refresh Close

```

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

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

```

```
OS_Time   10
```

```
Start_2Tasks.c#42 | HPTask | 00000E5A | | 9ms 971.2us | Event Break: "Evt0
```

Chapter 3

Build your own application

To build your own application, you should always start with a copy of the sample start workspace and project. Therefore copy the entire folder “Start” from your embOS distribution into a working folder of your choice and then modify the start project there. This has the advantage, that all necessary files are included and all settings for the project are already done.

3.1 Required files for an embOS application

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\*\"`-subfolder.
  It contains the hardware dependent initialization code for the specific CPU, the **embOS** timer and optional UART for `embOSView`.
- One **embOS** library from the "Lib\"-subfolder.
- **OS_Error.c** from one `target specific "BoardSupport\*\"`-subfolder. The error handler is used if any **embOS** library other than the Release build library is used in your project.

When you decide to write your own startup code, please ensure that non initialized variables are initialized with zero, according to "C" standard, as this is required by **embOS**.
Your `main()` function has to initialize **embOS** by call of `OS_InitKern()` and `OS_InitHW()` prior any other **embOS** functions except `OS_IncDI()` are called.

3.2 Change library mode

For your application you may wish to choose an other 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 wished library is already contained in your project, just select the appropriate configuration.
- To add a library, you may add the library to your project settings and exclude all other librarys from the configuration.
- Check and set the appropriate `OS_LIBMODE_*` define which you would like to use for debug and release builds and modify the OS_Config.h file accordingly.

3.3 Select an other CPU

embOS for V850 and NEC compiler contains CPU specific code for various NEC CPUs. 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 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** timer tick and communication to `embOSView`.

Chapter 4

V850 / V850E specifics

4.1 CPU modes

embOS supports all memory and code model combinations that NEC's C-Compiler supports.

4.2 Available librarys

embOS for V850 for NEC compiler comes with 14 different libraries, one for each CPU core and library type combination.
The libraries are named as follows:

libos<v><LibMode>.a

Parameter	Meaning	Values
`v`	Specifies the CPU variant	Empty: V850 core
		E : V850E core
`LibMode`	Library mode	XR: Extreme Release
		R: Release
		S: Stack check
		D: Debug
		SP: Stack check + profiling
		DP: Debug + profiling
		DT: Debug + trace

Example:

`libosEDP.a` is the library for a project using V850E core and debug and profiling features of **embOS**.

Chapter 5

Stacks

5.1 Task stack for V850

Every **embOS** task has to have its own stack. Task stacks can be located in any RAM memory location.
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 V850 CPUs, this minimum stack size is about 136 bytes to store the CPU registers. A practical minimum value is about 180 bytes.

5.2 System stack for V850

The system stack size required by **embOS** is about 40 bytes. 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 interrupts also use the system-stack, the actual stack requirements depend on the application.
Interrupt stack switching of **embOS** also uses the system stack for interrupts.
The size of the system stack is defined by the constant `STACKSIZE` in the startup file.

5.3 Interrupt stack for V850

V850 CPUs do not support a separate hardware interrupt stack. Therefore every interrupt runs on the task stack, as long as interrupt functions do not use interrupt stack switching functions.
To reduce task stack load by interrupts, **embOS** uses the system stack as interrupt stack. Interrupt handler should use `OS_EnterIntStack()` and `OS_LeaveIntstack()` to switch to interrupt stack. See "Interrupts" on page 23.

5.4 Stack specifics of the NEC V850 family

The NEC V850 family of microcontroller can address the whole memory space as stack. Therefore, stacks can be located anywhere in RAM. For performance reasons you should try to locate stacks in fast RAM.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs

- The CPU-core receives an interrupt request
- As soon as interrupts are enabled and the processor interrupt priority level is below the interrupt priority level, the interrupt is executed
- The CPU saves PC in EIPC register
- The CPU saves current processor status in EIPSW
- An exception is written into ECR
- Further interrupts are disabled, EP bit is cleared
- 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 RETI command, restoring saved processor status word and saved PC thus continuing interrupted task.

6.2 Defining interrupt handlers in "C"

Routines defined with the keyword `__interrupt` automatically save & restore the registers they modify and return with `RETI`.
The corresponding interrupt vector number may be defined by a `#pragma` directive prior the interrupt service routine.
For a detailed description on how to define an interrupt routine in "C", refer to the NEC C-Compiler's user's guide.

Interrupt-routine using embOS functions

```
#pragma interrupt INTUA0T tx
__interrupt void tx(void) {
  OS_EnterInterrupt();
  OS_EnterIntStack();
  OS_OnTx();
  OS_LeaveIntStack();
  OS_LeaveInterrupt();
}
```

Every interrupt service routine which uses **embOS** functions has to inform **embOS** that interrupt code is running. Therefore the first command in an interrupt service routine should be `OS_EnterInterrupt()`, the last command has to be `OS_LeaveInterrupt()`.

6.3 Nestable interrupt-routines

Allowing nestable interrupts requires an other function declaration to instruct the compiler to generate different code which allows nestable interrupts. The interrupt handler has to be declared with the prefix `__multi_interrupt`.
embOS has to be informed that nesting should be allowed. `OS_EnterNestableInterrupt()` and `OS_LeaveNestableInterrupt()` have to be used. The call of `OS_EnterNestableInterrupt()` re-enables the interrupt.

Nestable interrupt routine using embOS functions

```
#pragma interrupt INTUA3R OS_ISR_rx
__multi_interrupt void OS_ISR_rx(void) {
  OS_EnterNestableInterrupt();
  OS_EnterIntStack();
  _ISR_Rx_Handler();
  OS_LeaveIntStack();
  OS_LeaveNestableInterrupt();
}
```

6.4 Interrupt stack switching

Since the V850 CPUs do not have a separate stack pointer for interrupts, every interrupt runs on the current stack. To reduce stack load of tasks, **embOS** offers its own interrupt stack which is located in the system stack.
To use **embOS** interrupt stack, call `OS_EnterIntStack()` at the beginning of an interrupt handler just after the call of `OS_EnterInterrupt()` and `OS_LeaveIntStack()` at the end just before `OS_LeaveInterrupt()`.

Please note, that an interrupt handler using interrupt stack switching must not use local variables.

No calculation is allowed in an interrupt handler which uses stack switching. The interrupt handler has to call an other function which must not take any parameter.

Interrupt-routine using embOS interrupt stack

```
static void OS_ISR_Rx_Handler(void) {
  volatile unsigned char Dummy;
  if ((UA0STR & 0x07) == 0) { /* No error */
    OS_OnRx(UA0RX); /* Process data byte */
  } else {
    UA0STR &= ~0x07; /* Reset error */
    Dummy = UA0RX; /* discard Byte */
  }
}

#pragma interrupt INTUA0R rx
__interrupt void rx(void) {
  OS_EnterNestableInterrupt(); /* We will enable interrupts */
  OS_EnterIntStack(); /* We will use interrupt stack */
  OS_ISR_Rx_Handler(); /* Call to handler is required ! */
  OS_LeaveIntStack(); /* Interrupt stack switching does */
  OS_LeaveNestableInterrupt(); /* not allow local variables in ISR */
}
```

Chapter 7

STOP / WAIT mode

7.1 Saving power

Usage of the HALT-mode 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`.
As internal peripheral clock is not stopped in this mode, **embOS** keeps functioning. Any interrupt will wake up the CPU and will therefore continue suspended tasks if required.

IDLE and STOP mode stop internal peripheral clock and can only be resumed by NMI or RESET and should therefore not be used to reduce power consumption during idle times in `OS_Idle()`.

Chapter 8

Technical data

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**. In THUMB mode, the minimum ROM size required for the **embOS** kernel is about 1.870 bytes.
In the table below, you find the minimum RAM size for **embOS** resources. The sizes depend on selected **embOS** library mode; the table below is for a release build.

embOS resource	**RAM [bytes]**
Task control block	32
Resource semaphore	8
Counting semaphore	4
Mailbox	20
Software timer	20

Chapter 9

Files shipped with embOS

9.1 Files included in embOS

Directory	File	Explanation
root	*.pdf	Generic API and target specific documentation
root	embOSView.exe	Utility for runtime analysis, described in generic documentation
root	Release.html	Version control document
Start\BoardSupport\*\	*.*	Sample workspaces and project files for NEC's compiler and PM+ Workbench
Start\BoardSupport\*\Application\	*.*	Sample programs to serve as a start
Start\BoardSupport\*\Setup\	*.*	CPU specific hardware routines for various CPUs
Start\Inc\	BSP.h	Include file for BoardSupport packages, to be included in every "C"-file using BSP-functions
Start\Inc\	OS_Config.h	Include file for **embOS** library mode configuration, included by RTOS.h
Start\Inc\	RTOS.h	Include file for **embOS**, to be included in every "C"-file using embOS-functions
Start\Lib\	libos*.a	**embOS** libraries

Any additional files shipped serve as example.

Index