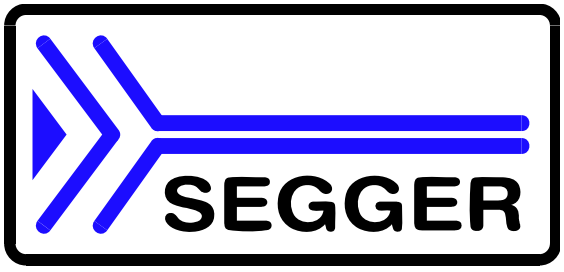
embOS

Real Time Operating System

CPU & Compiler specifics for AVR32AP cores using AVR32 Studio and GNU tool-chain

Document revision 1

Date: October 9, 2008



A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

Disclaimer

Specifications written in this document are believed to be accurate, but are not guaranteed to be entirely free of error. The information in this manual is subject to change for functional or performance improvements without notice. Please make sure your manual is the latest edition. While the information herein is assumed to be accurate, SEGGER MICROCONTROLLER GmbH & Co. KG (the manufacturer) assumes no responsibility for any errors or omissions. The manufacturer makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. The manufacturer specifically disclaims any implied warranty of merchantability or fitness for a particular purpose.

Copyright notice

Trademarks

Names mentioned in this manual may be trademarks of their respective companies.

Brand and product names are trademarks or registered trademarks of their respective holders.

Contact address

SEGGER Microcontroller GmbH & Co. KG

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	081007	TS	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

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 efficent 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 developes and produces programming tools for flash microcontrollers, 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. Starterkits, eval- and trial-versions are available.

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. The profiling PC tool embOSView is included.

emFile
File system
emFile is an embedded file system with FAT12, FAT16 and FAT32 support. emFile has been optimized for minimum memory consumption in RAM and ROM while maintaining high speed. Various Device drivers, e.g. for NAND and NOR flashes, SD/MMC and CompactFlash cards, are available.

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

SEGGER TOOLS

Flasher
Flash programmer
Flash Programming tool primarily for microcontrollers.

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

Introduction

This guide describes how to use **embOS** Real Time Operating System for the AVR32AP series of microcontrollers using AVR32 Studio and GNU toolchain.

How to use this manual

This manual describes all CPU and compiler specifics of **embOS** using AVR32AP based controllers with AVR32 Studio and GNU toolchain. 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 AVR32AP using AVR32 Studio and GNU toolchain. 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 AVR32AP based controllers using AVR32 Studio and GNU toolchain.

Chapter 2

Using embOS with AVR32 Studio and GNU toolchain

The following chapter describes how to start with and use **embOS** for AVR32AP and AVR32 Studio and GNU toolchain. You should follow these steps to become familiar with **embOS** for AVR32AP and AVR32 Studio and GNU toolchain.

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 harddrive 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 AVR32 Studio and GNU toolchain 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 the AVR32 Studio for your application development in order to become familiar with **embOS**.

If for some reason you will not work with AVR32 Studio, 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 `AVR32 Studio`, it may be used without the project manager using batch files or a make utility without any problem.

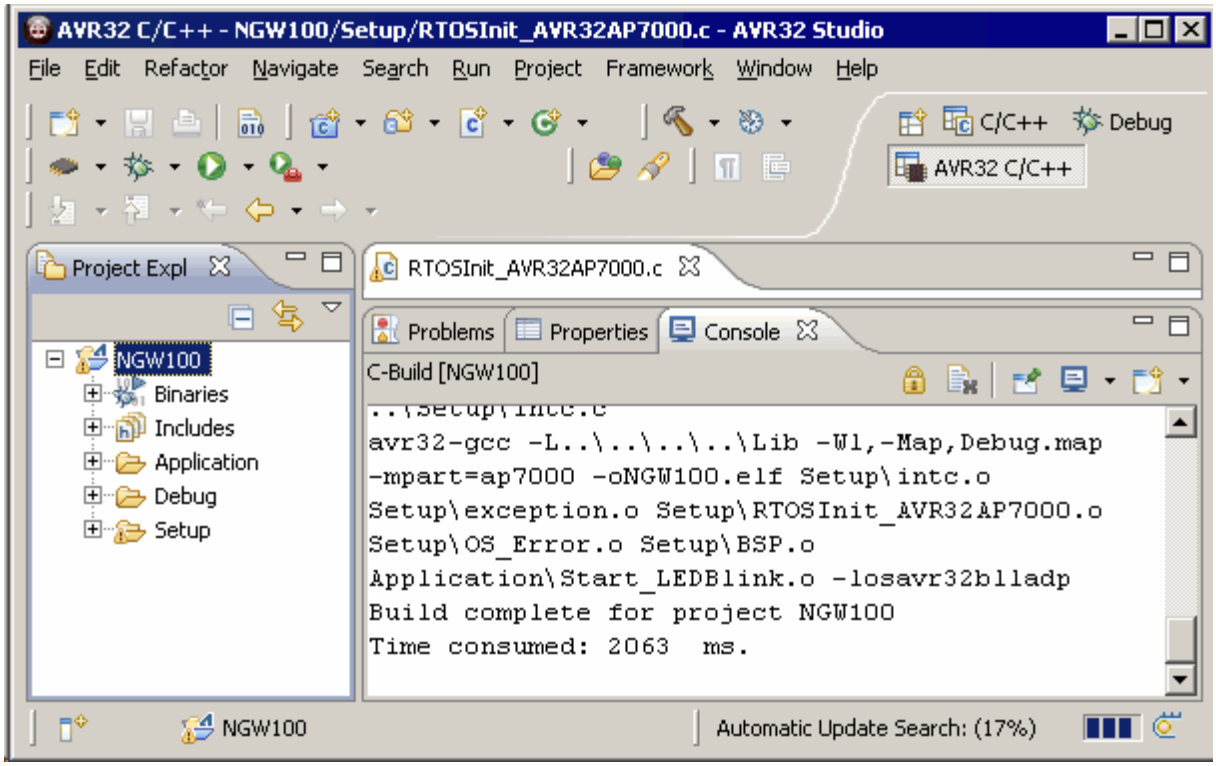
2.2 First steps

After installation of **embOS** (See "Installation" on page 10.) you are able to create your first multitasking application. You received a 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 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
- Start AVR32 Studio and switch your workspace directory to "Start\BoardSupport\Atmel\"
- Build one of the the start projects

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



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 evalboards, if required.

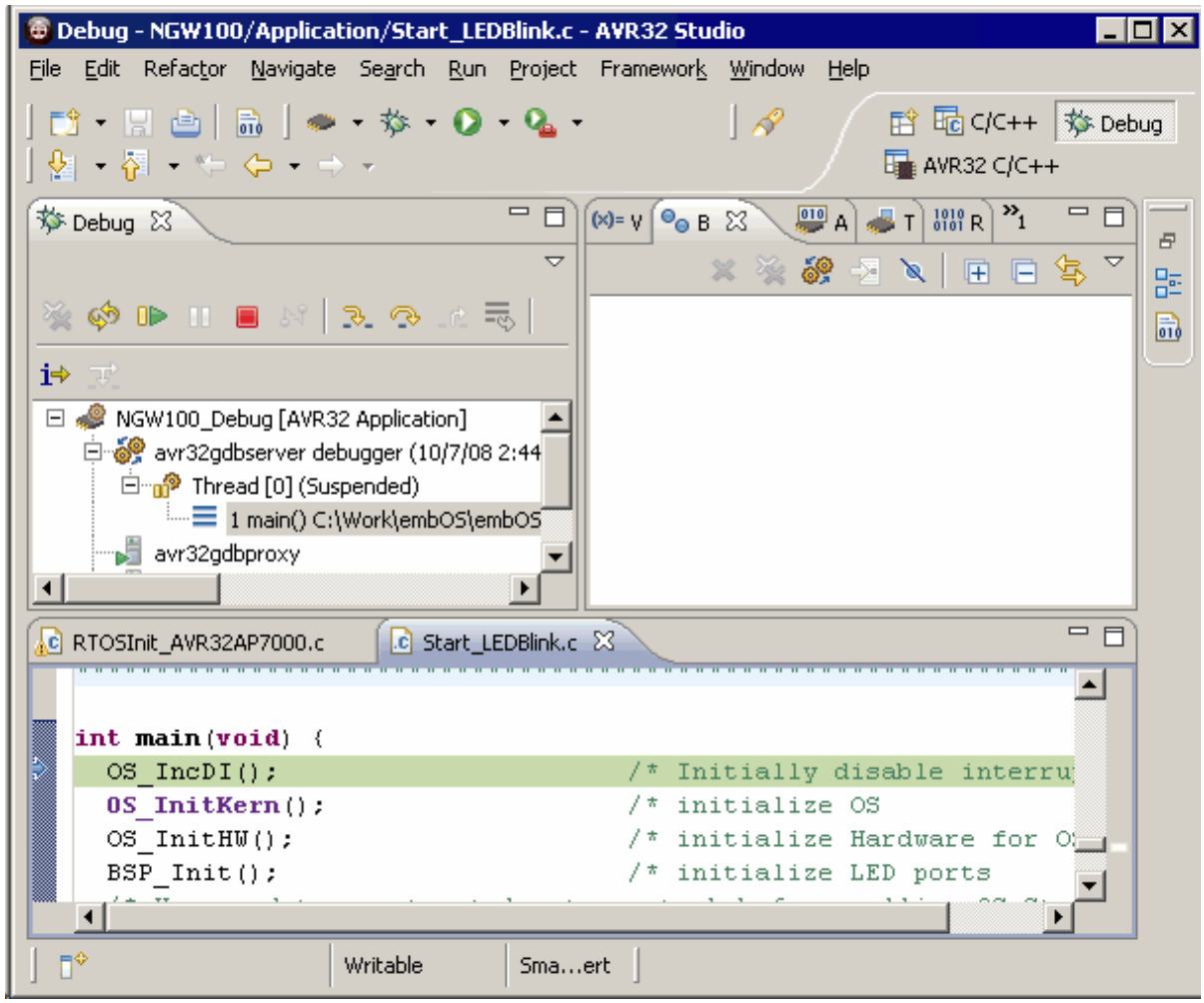
2.3 The sample application Main.c

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.

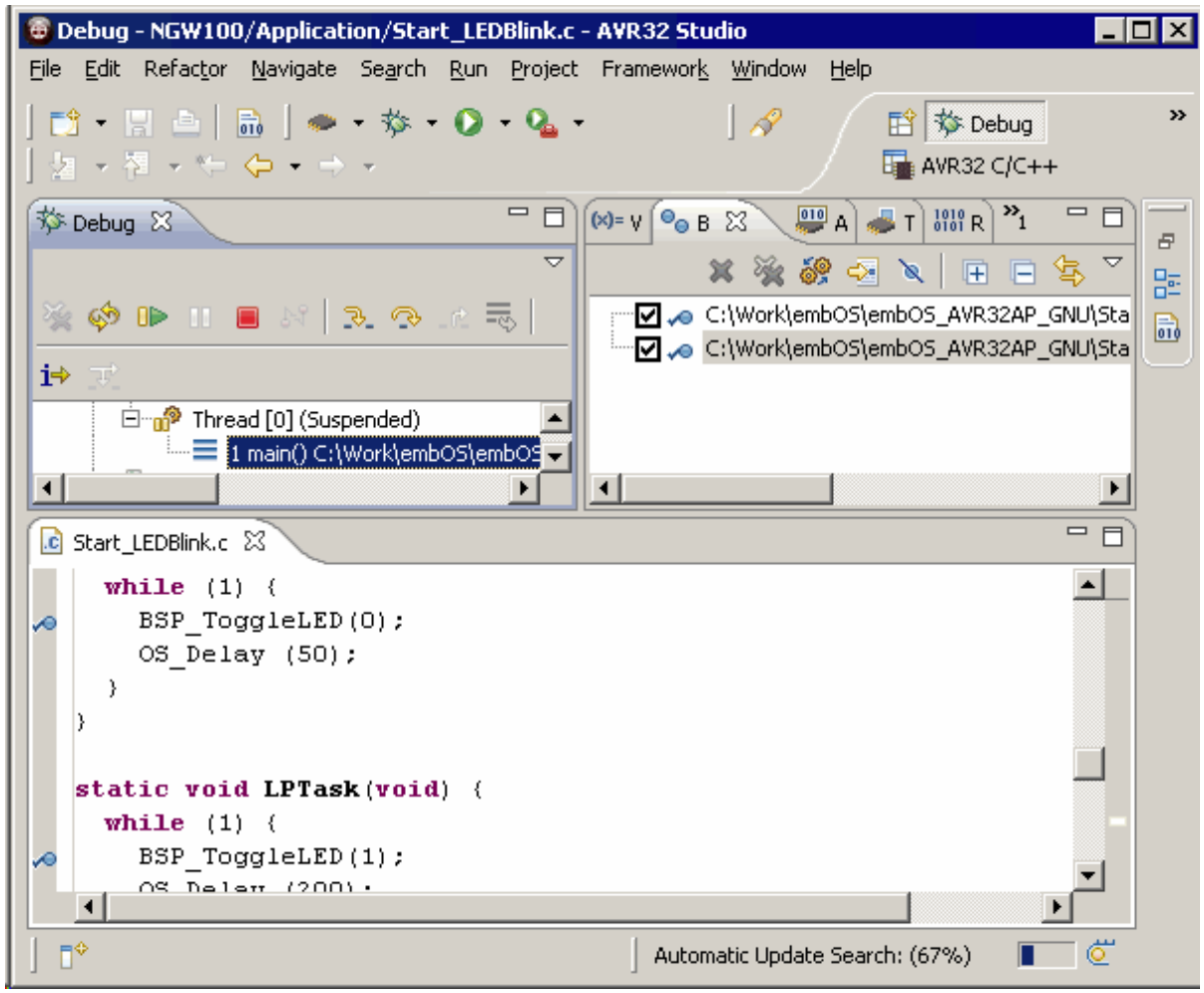
```
/*********************************************************
*          SEGGER MICROCONTROLLER GmbH & Co. KG         *
*  Solutions for real time microcontroller applications *
*********************************************************
File    : Main.c
Purpose : Skeleton program for OS
--------------END-OF-HEADER---------------------------*/
#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 AVR-JTAGICE-MKII

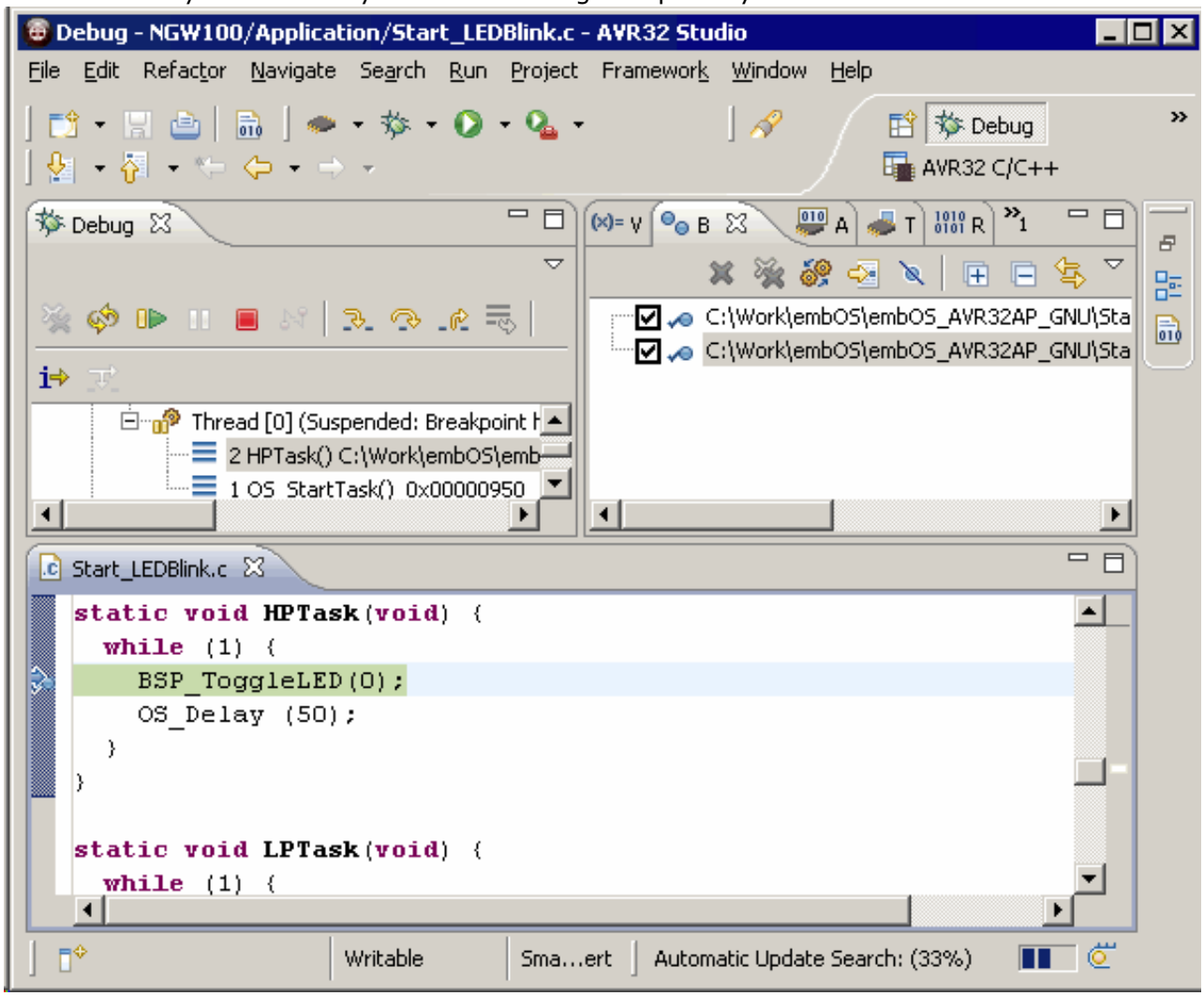
When starting the debugger, you will usually see the main function (very similar to the screenshot below). In some debuggers, you may look at the startup code and have to set a breakpoint at main. Now you can step through the program. `OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is part 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.



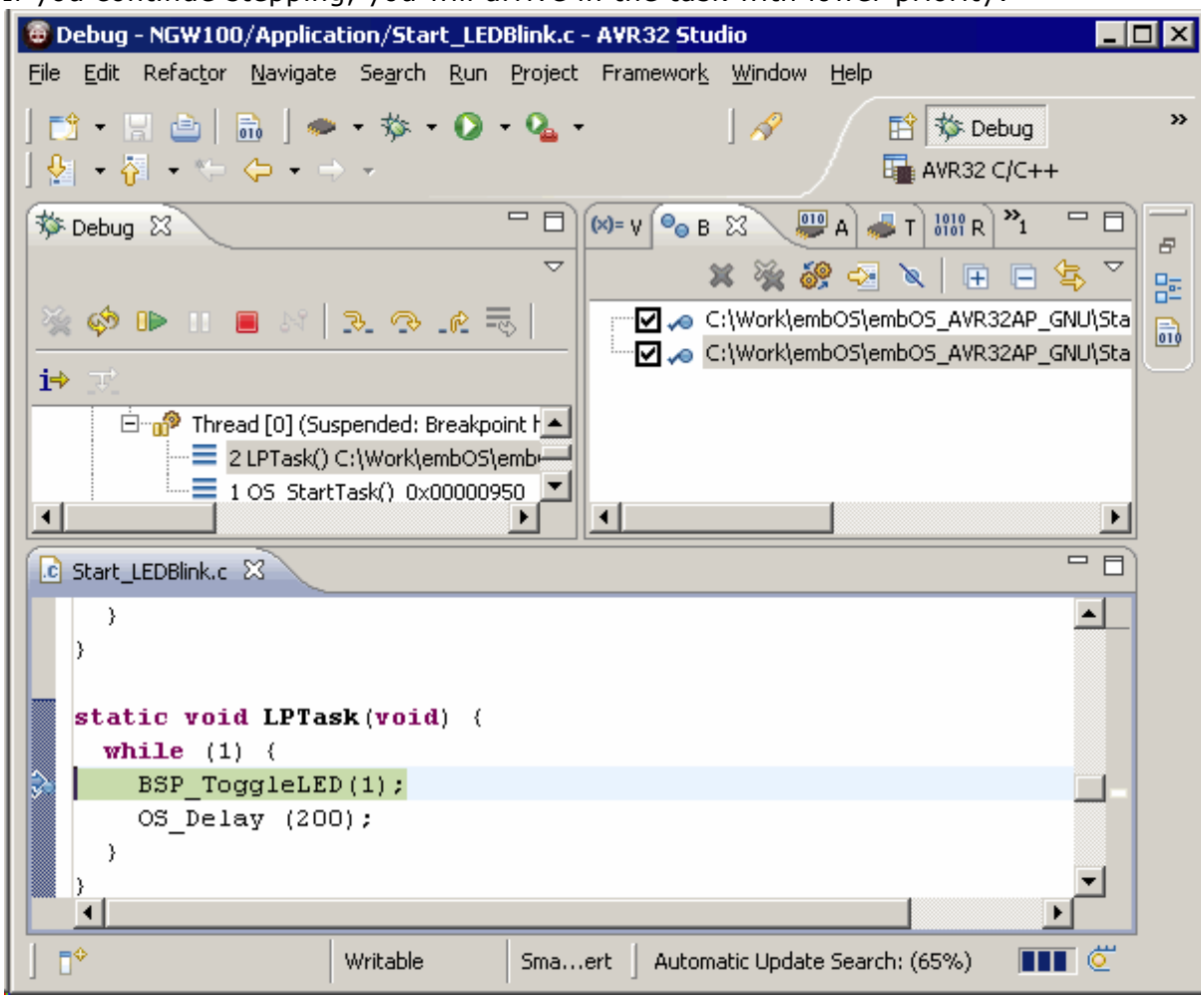
Before you step into `OS_Start()`, you should set two break points in the two tasks as shown below.



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.

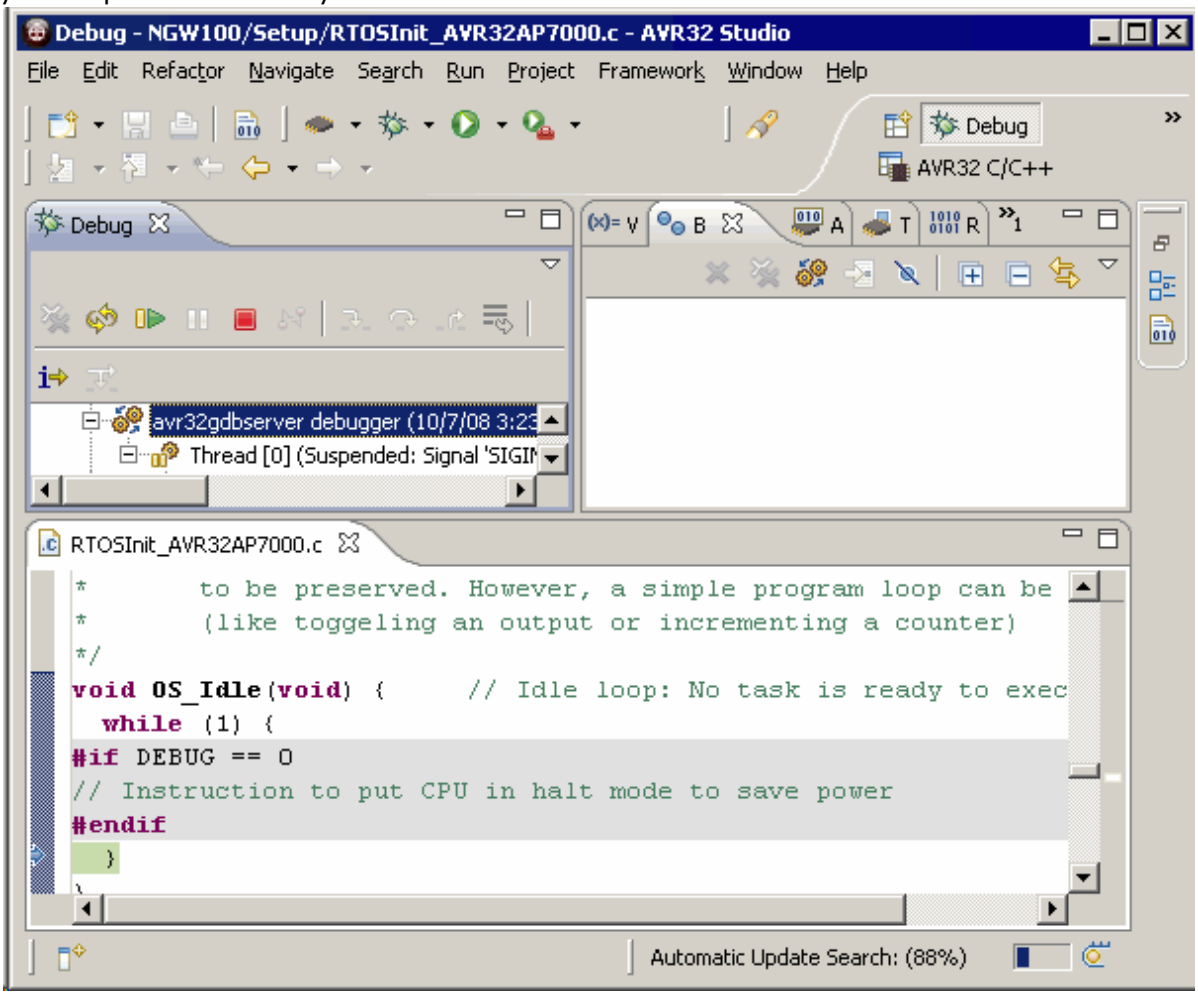


If you continue stepping, you will arrive in the task with lower priority:

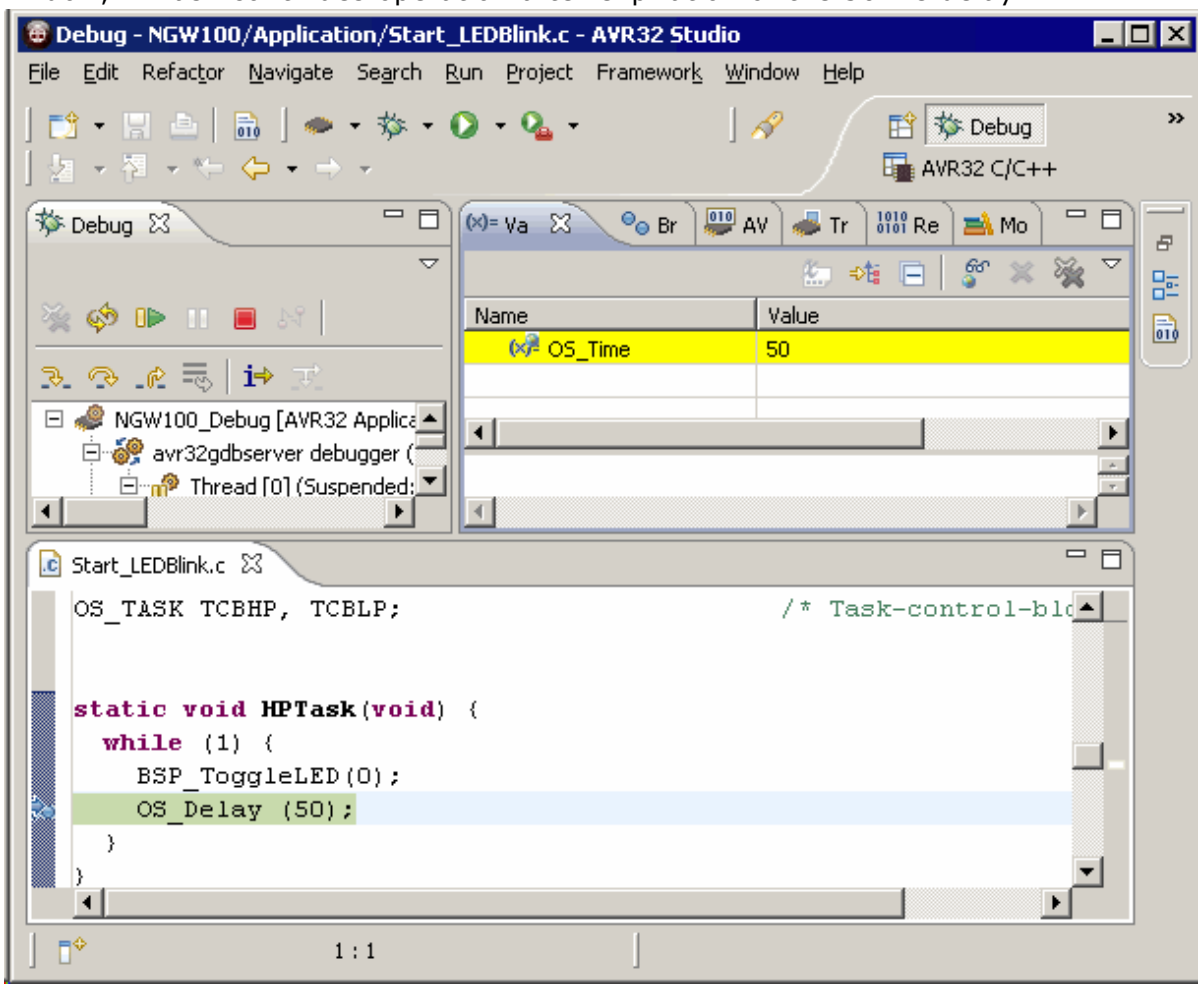


Continuing to step through the program, there is no other task ready for execution. **embOS** will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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.



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 50 ms delay.



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, 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.
Also ensure, that main is called with CPU running in supervisor mode.
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.
You should then modify or replace the `main.c` source file in one target specific "`BoardSupport\*\`"-subfolder.

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, change the project settings accordinangly.
- 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 AVR32AP and AV32 Studio and GNU toolchain contains CPU specific code for various AVR32AP 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 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` and `__lowlevel_init()`.

Chapter 4

AVR32 specifics

4.1 CPU modes

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

4.2 Available libraries

embOS for GNU compiler comes with 7 different libraries, one for each library mode. The libraries are named as follows:

libosavr32b<CodeModel><DataModel><AllowUnalignedWordAccess><LibMode>.r82

Parameter	Meaning	Values
CodeModel	Code modele	S: Small code model
		M: Medium code model
		L: Large code model
DataModel	Data model	S: Small data model
		L: Large data model
AllowUnalignedWordAccess	Allow unaligned word access	U: Allow unaligned access
		A: Do not allow unaligned access
LibMode	Library mode	XR: Extreme release
		R: Release
		S: Stack check
		D: Debug
		SP: Stack check + profiling
		DP: Debug + profiling
		DT: Debug + trace

Example:

`osavr32blladp.a` is the library for a project using AVR32AP core, large code model, large data model, do not allow unaligned word access and debug and profiling features of **embOS**.

4.3 Application mode and supervisor mode

All **embOS** tasks, interrupts and embOS system run in supervisor mode. Interrupts require space on the supervisor stack. Therefore extra space for interrupts on the task stacks is needed.

Chapter 5

Stacks

5.1 Task stack for AVR32AP

All **embOS** tasks execute in supervisor mode. Every **embOS** task has its own individual stack which can be located in any memory area. The required stacksize for a task is the sum of the stack size used by all functions for local variables and parameter passing, basic stack size and stack size for all nested interrupt routines.
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 AVR32AP, this minimum basic task stack size is about 44 bytes.

5.2 System stack for AVR32AP

The **embOS** system executes in supervisor mode. The minimum system stack size required by **embOS** is about 144 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 systemstack, the actual stack requirements depend on the application.

5.3 Interrupt stack for AVR32AP

If a normal hardware exception occurs, the AVR32AP core switches to IRQ mode,which uses the supervisor stack pointer.
Every interrupt requires 28 bytes on the supervisor stack.
The maximum supervisor stack size required by the application can be calculated as maximum interrupt nesting level * 28 bytes.

5.4 Stack specifics of the AVR32AP family

Interrupts require space on the supervisor stack. The supervisor stack is used to store contents of scratch registers, the ISR itself uses supervisor stack. The Supervisor stack is also used during startup, main(),**embOS** internal functions and software timers.
All other stacks are not initialized and not used by **embOS**. If required by the application, the startup function and linker command files have to be modified to initialize the stacks.

Chapter 6

Interrupts

6.1 What happens when an interrupt occurs

- The CPU-core receives an interrupt request
- If the interrupt is not masked the interrupt is accepted
- The hardware automatically sets the mask bits in the status register
- The Status Register and Program Counter are stored by hardware in the register set of the current context
- LR and R8-R12 are stored on the supervisor stack
- User defined functionality in interrupt service routine
- The rete instruction signals the end of the event.
- Return from interrupt.
- For details, please refer to the AVR32AP user manual

6.2 Defining interrupt handlers in "C"

Routines preceded by the keywords "__attribute__((interrupt))" return with RETE.

The interrupt handler used by embOS are implemented in the CPU specific RTOSInit_*.c file. **embOS** provides support for each of the four interrupt levels.

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 fast interrupts.

```
static void _OS_ISR_Tick (void) __attribute__((interrupt));
static void _OS_ISR_Tick (void) {
  OS_CallISR(_OS_ISR_TickHandler);
}
```

Any interrupt handler running at priorities from 1 to 2 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 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 "fast" interrupt priority level, thus locking any other embOS interrupt, Fast 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.

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

The rules for a *Fast interrupt* handler are as follows:

- Local variables may be used.
- Other functions may be called.
- **embOS** functions must not be called, nor direct, neither indirect.
- The priority of the interrupt has to be above the user definable priority limit for fast interrupts.

6.3 OS_SetFastIntPriorityLimit(): Setting the interrupt priority limit for fast interrupts

The interrupt priority limit for fast interrupts is set to 2 by default. This means, all interrupts with higher priority from 3 to 4 will never be disabled by **embOS**.

Description

OS_SetFastIntPriorityLimit() is used to set the interrupt priority limit between fast interrupts and lower priority **embOS** interrupts.

Prototype

void OS_SetFastIntPriorityLimit(unsigned int Priority)

Parameter	Meaning
Priority	The highest value useable as priority for embOS interrupts. All interrupts with higher priority are never disabled by **embOS**. Valid range: 0 < Priority <= 4

Return value

NONE.

Add. information

To disable fast interrupts at all, the priority limit may be set to 4 which is the highest interrupt priority for interrupts. To modify the default priority limit, OS_SetFastIntPriorityLimit() should be called before **embOS** was started. In the default projects, OS_SetFastIntPriorityLimit() is called from OS_IntHW() in RTOSInit_*.c. All interrupts running at low priority from 0 to the user definable priority limit for fast 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 fast interrupt priority limit must not call any **embOS** function.

6.4 OS_InitInterrupts(): Initialize the interrupt table and the interrupt vector controller

The RAM interrupt table let the user define which handler runs if an specific interrupt occur.

Description

OS_InitInterrupts() is used to initialize the interrupt handler table, which is placed in RAM.

Prototype

void OS_InitInterrupts(void)

Return value

NONE.

6.5 OS_InstallISRHandler(): Install an interrupt handler

Description

OS_InstallISRHandler() is used to install a specific interrupt handler..

Prototype

```
void OS_InstallISRHandler (OS_ISR_HANDLER* pISRHandler, int ISRIndex,
                           int ISRPrio)
```

Parameter	Meaning
pfISRHandler	Address of the interrupt handler function.
ISRIndex	Index of the interrupt source
ISRPrio	Interrupt Priority

Return value

NONE.

Add. information

None.

6.6 OS_GetIRQHandler(): Get handler for avtive interrupt source

This routine is only used internally. The user must not call this function.

Description

OS_GetIRQHandler() returns the handler address for an interrupt source.

Prototype

OS_ISR_HANDLER* OS_GetIRQHandler(int ISRPrio)

Parameter	Meaning
ISRPrio	Interrupt group priority Valid range: 0 < Priority <= 4

Return value

OS_ISR_HANDLER*: the address of the configured interrupt handler for the given interrupt source.

6.7 Interrupt stack switching

Since AVR32AP core based controllers 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. The AVR32 supervisor stack is used for interrupt handler.

Chapter 7

STOP / WAIT mode

7.1 Saving power

In case your controller does support some kind of power saving mode, it should be possible to use it also with **embOS**, as long as the timer keeps working and timer interrupts are processed. To enter that mode, you usually have to implement some special sequence in function `OS_Idle()`, which you can find in **embOS** module `RTOSIinit_*.c`.

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**. Using AVR32 mode, the minimum ROM requirement for the kernel itself is about 2.500 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	36
Resource semaphore	16
Counting semaphore	8
Mailbox	24
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
.metadata	*.*	AVR32 Studio workbench files
Start\Inc	RTOS.h	Include file for **embOS**, to be included in every "C"-file using **embOS**-functions
Start\Lib	osavr32b*.a	**embOS** libraries
Start\Setup	OS_Error.c	**embOS** runtime error handler used in stack check or debug builds
Start\Setup	RtosInit*.c	CPU specific hardware routines
Start\Application	*.c	Sample frame program to serve as a start

Any additional files shipped serve as example.

Index

