embOS

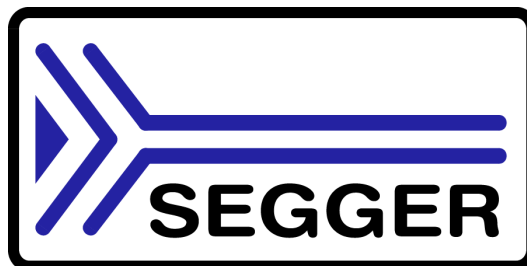## Real-Time Operating System

CPU & Compiler specifics for PIC24 using Microchip XC16 compiler

Document: UM01059

Software version 4.02a

Revision: 0

Date: October 16, 2014

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 (SEGGER) assumes no responsibility for any errors or omissions. SEGGER makes and you receive no warranties or conditions, express, implied, statutory or in any communication with you. SEGGER 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

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: October 16, 2014

Software	Revision	Date	By	Description
4.02a	0	141015	TS	First version

4

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 com-piler)
*   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 Program-ming 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 program-ming 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 programm examples.
Reference	Reference to chapters, sections, tables and figures or other docu-ments.
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 for PIC24/PIC30/dsPIC33

This chapter describes how to start with and use embOS for PIC24 cores and the Microchip XC16 compiler. You should follow these steps to become familiar with embOS. The manual only refers to PIC24 but it works the same way for PIC30 and dsPIC33 CPUs.

1.1   Installation

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

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

Assuming that you are using the MPLABX project manager 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" on page 11.

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

If you will not work with the MPLABX IDE, you should: Copy either all or only the library-file that you need to your work-directory. 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 the MPLABX project manager, it may be used without the project manager 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 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.

For the first step, you may use the project for PIC24FJ128 CPU:
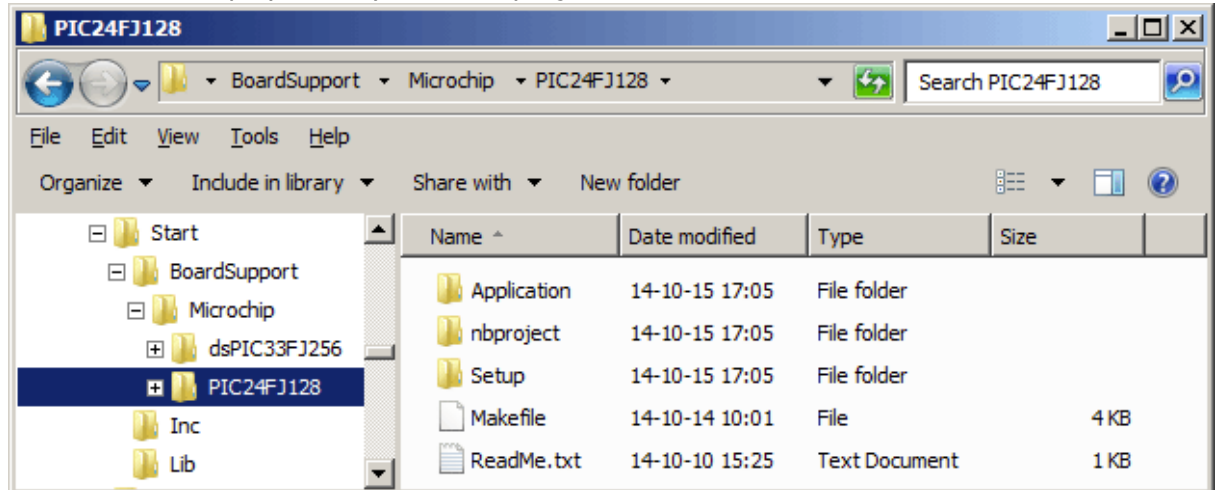
To get your new application running, you should proceed as follows:

- 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 the sample workspace
  Start\BoardSupport\Microchip\PIC24FJ128 with the MPLABX IDE project manager.
- Build the project. It should be build without any error or warning messages.

After generating the project of your choice, the screen should look like this:
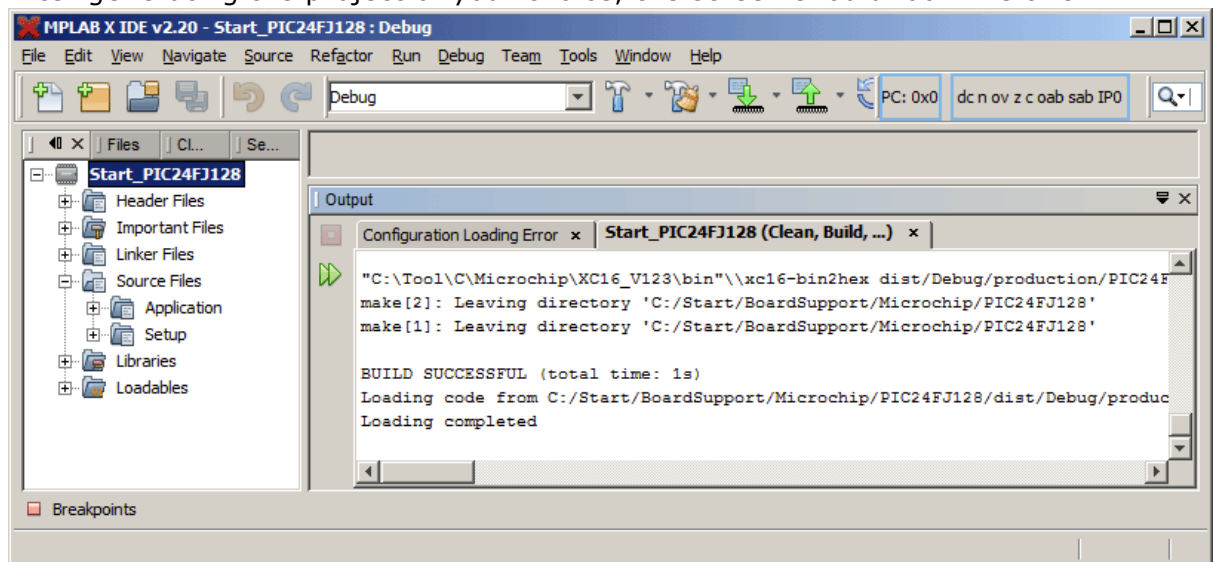
For additional information you should open the `ReadMe.txt` file which is part of every specific project. The ReadMe file describes the different configurations of the project and gives additional information about specific hardware settings of the supported eval boards, if required.

1.3   The example application Start_2Tasks.c

The following is a printout of the example application Start_2Tasks.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.

```c
/***********************************************************
* SEGGER MICROCONTROLLER SYSTEME GmbH & Co.KG
* Solutions for real time microcontroller applications
************************************************************
File    : Start2Tasks.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 */'

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

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

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

void 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;
}
```

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 the C-SPY option **Run to main** is selected, which it is 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 timer-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.
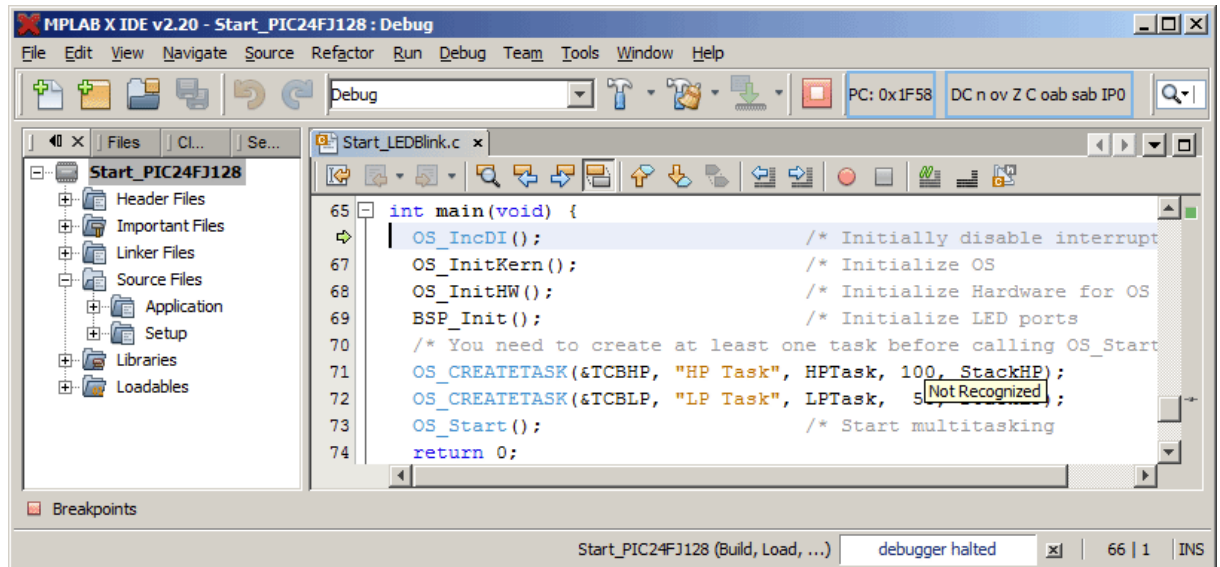
Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.
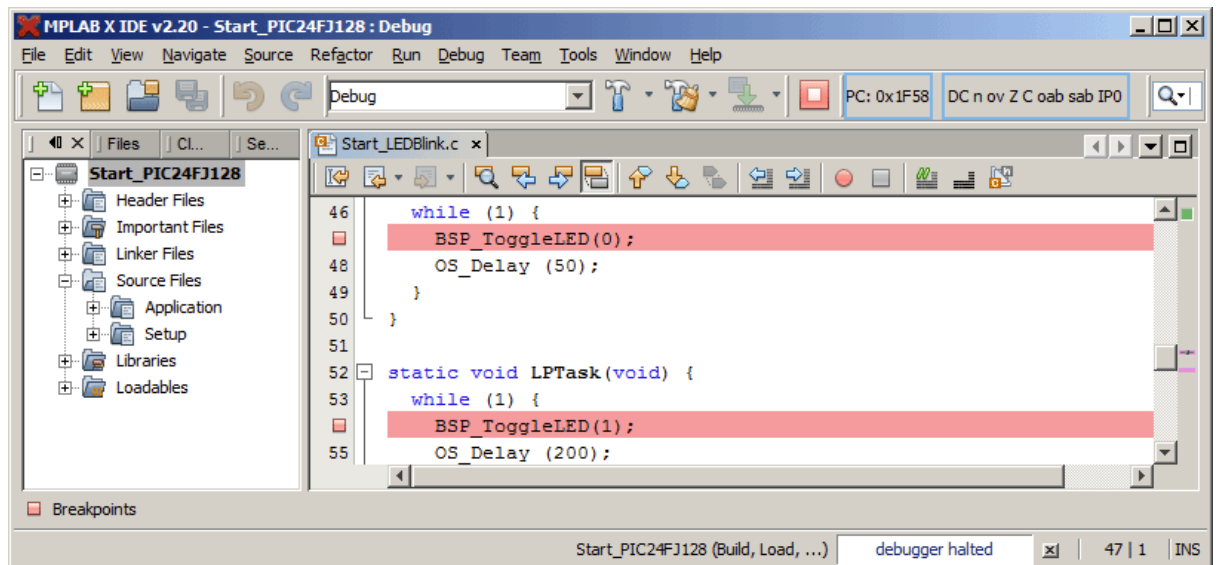
As `OS_Start()` is part of the embOS library, you can step through it in disassembly mode only.

Click **GO**, step over `OS_Start()`, or step into `OS_Start()` in disassembly mode until you reach the highest priority task.
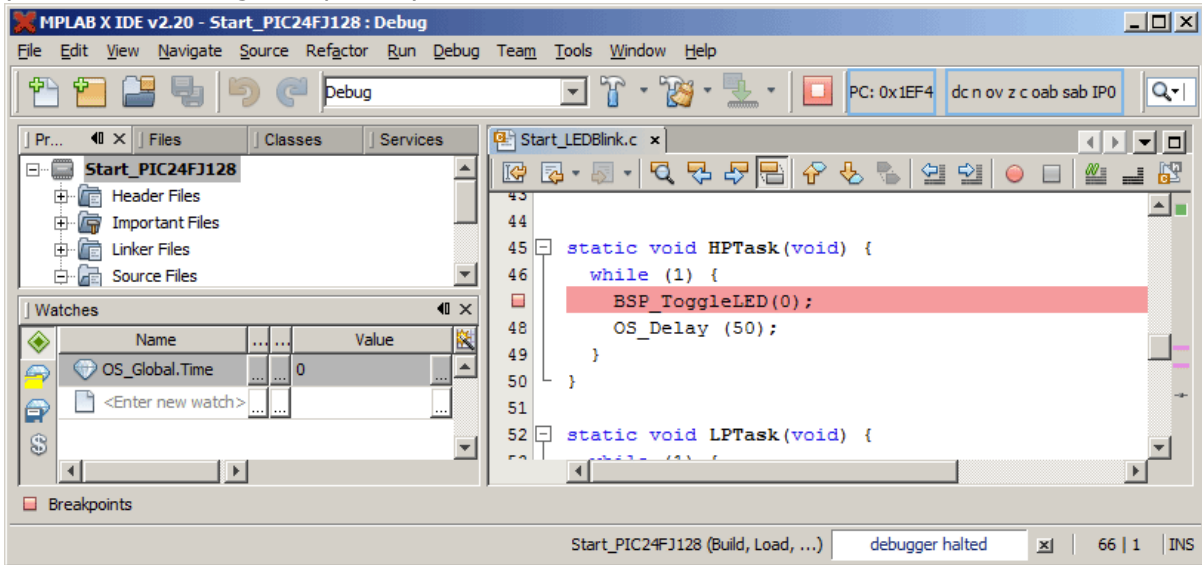
If you continue stepping, you will arrive in the task that has lower priority:
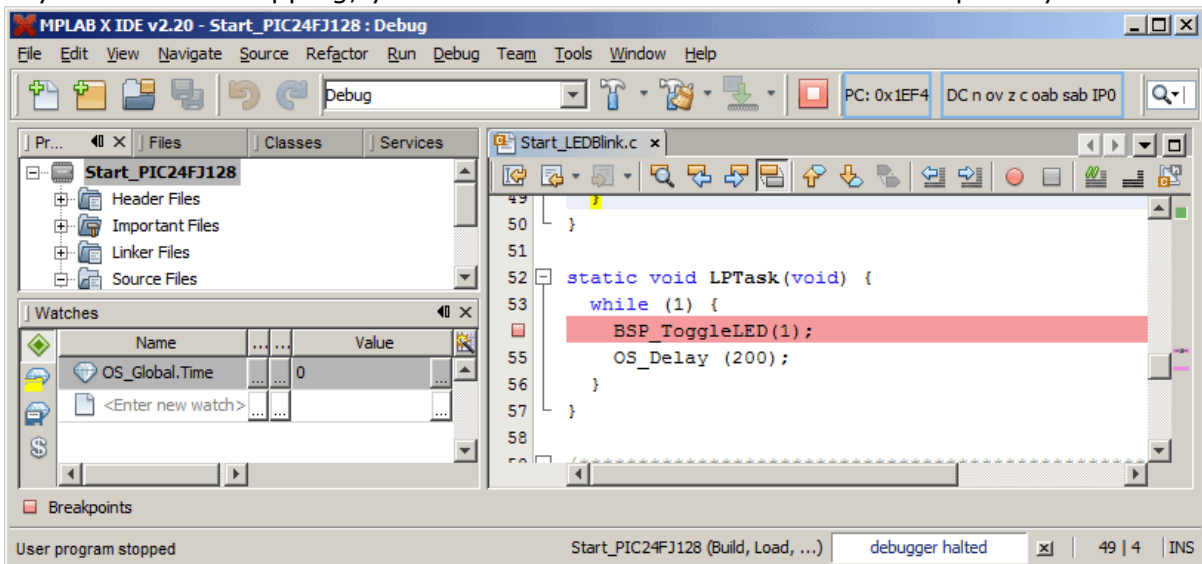
Continue to step through the program, there is no other task ready for execution. embOS will therefore start the idle-loop, which is an endless loop which is always executed if there is nothing else to do (no task is ready, no interrupt routine or timer executing).

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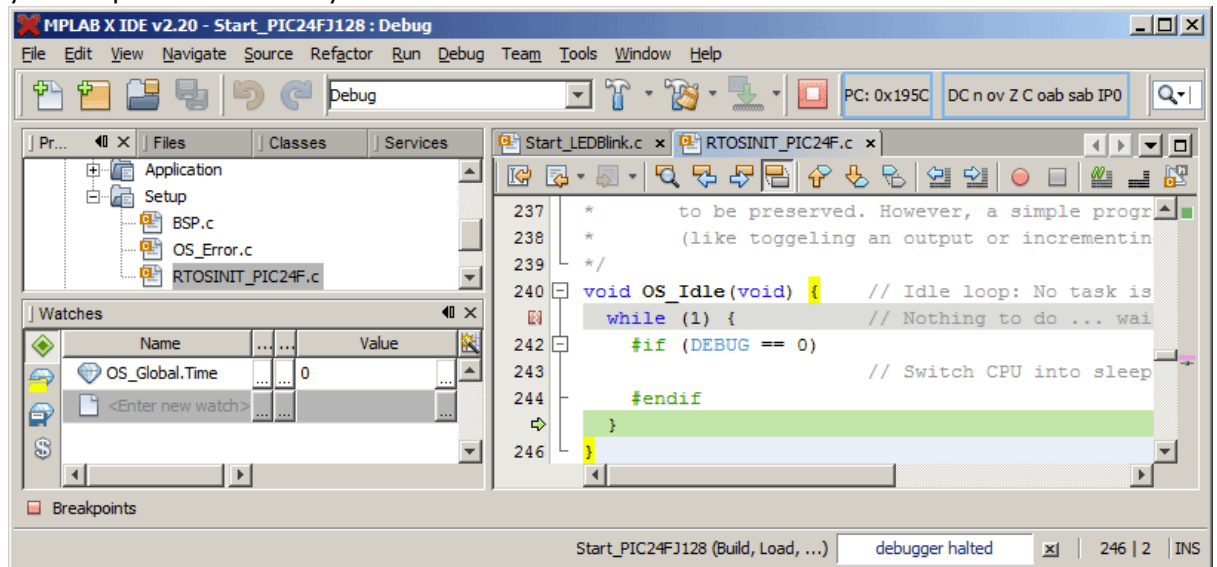
If you set a breakpoint in one or both of our tasks, you will see that they continue execution after the given delay.

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.
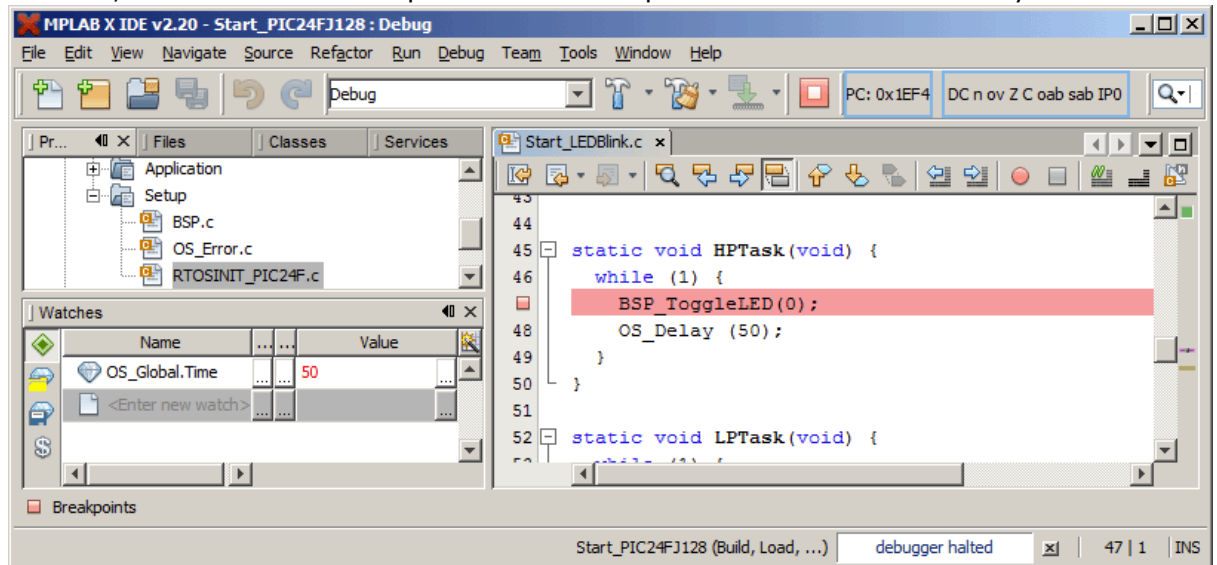
Chapter 2

Build your own application

This chapter provides all information to setup 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 First steps on page 9 and modify the project to fit your needs. Using a sample 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 for PIC24

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, 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 library other than Release build library is used in your project.
- Additional low level init code 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.
Also ensure, that main() is called with the CPU running in supervisor or system mode.
Your main() function has to initialize embOS by a call of OS_InitKern() and OS_InitHW() prior any other embOS embOS functions are called.
You should then modify or replace the Start_2Task.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 a new Lib group to your project and add this library to the new group. Exclude all other library groups from build, delete unused Lib groups 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 PIC24 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 PIC24 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()`, and the interrupt service routines for embOS timer tick and communication to embOSView and `__low_level_init()`.

Chapter 3

PIC24 specifics

3.1   CPU modes

embOS for PIC24 supports all memory and code model combinations that the Microchip XC16 compiler supports.

3.2   Available libraries

embOS for PIC24 and XC16 compiler comes with 28 different libraries, one for each memory model / library mode combination.

3.2.1　Naming conventions for prebuilt libraries

embOS PIC24 for the Microchip XC16 compiler is shipped with different prebuilt libraries with different combinations of the following features:

- Code memory model
- Data memory model
- Library mode - `LibMode`

The libraries are named as follows:

`RTOS<Code_model><Data_model><Libmode>.a`

Parameter	Meaning	Values
Code model	Specifies the code memory model	S: Small L: Large
Data model	Specifies the data memory model	S: Small L: Large
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

RtosLLR.a is the library for a project using the large code memory model, large data memory model and release build library type.

Chapter 4

Stacks

4.1 Task stack for PIC24

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.

For the PIC24 CPUs, this minimum basic task stack size is about 26 bytes. Because any function call uses some amount of stack the task stack size has to be large enough. We recommend at least 128 bytes stack as a start.

As PIC24 CPUs have a 16 bit stack pointer, the whole internal memory area can be used as task stack.

Please note, that the task stacks have to be aligned at EVEN addresses. To ensure proper alignment, implement task stack as array of int.

4.2 System stack for PIC24

The minimum system stack size required by embOS is about 44 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 project options.
We recommend a minimum stack size of 256 bytes for the CSTACK.

4.3 Interrupt stack for PIC24

The PIC CPUs have no separate interrupt stack pointer. The interrupt service routines use the task stacks or the system stack in the case no task is running.

Chapter 5

Interrupts

5.1   What happens when an interrupt occurs?

- The CPU-core receives an interrupt request.
- As soon as the processor interrupt priority level is below or equal to the interrup priority level, the interrupt is executed
- The CPU pushes status regiter and the return address onto the current stack.
- The IPL is loaded with the priority of the interrupt.
- The CPU calls the interrupt service routine, that belongs to the received interrupt. The interrupt vector table is located at address 0x0000.
- The interrupt handler is processed.
- Execute *retfie* to return from interrupt, restoring all saved registers

For more details, please refer to Microchip user manual

5.2   Defining interrupt handlers in "C"

Routines defined with the keywords OS_ISR automatically save & restore the registers they modify and return with RETFIE. OS_ISR is defined in the header file Rtos.h.

For a detailed description on how to define an interrupt routine in "C", refer to the C30 CCompiler's user's guide.

Example

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.

```
/**********************************************************************
*
* _T1InterruptHandler()
* Systick interrupt handler function
*/
static void _T1InterruptHandler(void) {
  IFS0 &= ~(1 << 3); // Reset Timer 1 interrupt pending flag
  OS_TICK_Handle(); // Call OS tick handler
}

/**********************************************************************
*
* _T1Interrupt()
* The interruput service routine
*/
void OS_ISR _T1Interrupt(void) {
  OS_CallNestableISR(&_T1InterruptHandler);
}
```

Any interrupt handler running at priorities from 1 to 5 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 nestingshould 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().

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.

```
__attribute__((no_auto_psv,__interrupt__)) void ZeroLatencyInterrupt (void) {
  unsigned long Count; // local variables are allowed
  Count = TPU_TCNT0;
  HandleCount(Count); // Any function call except embOS functions is allowed
}
```

5.3   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, zero latency 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 CPU the, thus keeping all interrupts with higher priority enabled.

5.4   Zero latency interrupts with PIC24

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 interrupt priorities 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 5, meaning, any interrupt with level 6 or above is never disabled and can be accepted anytime.

You must not execute any embOS function from within a zero latency interrupt function.

5.5   Interrupt priorities with PIC24

With introduction of zero latency interrupts, interrupt priorities useable 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 fast interrupts can be set at runtime by a call of OS_SetFastIntPriorityLimit().

5.6   OS_SetFastIntPriorityLimit()

The interrupt priority limit for fast interrupts is set to 5 by default. This means, all interrupts with higher priority 6 and 7 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	Description
Priority	The highest value useable as priority for embOS interrupts. All interrupts with higher priority are never disabled by embOS. Valid range: 1 <= Priority <= 7.

Return value

NONE.

Add. 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.
In the default projects, OS_SetFastIntPriorityLimit() is called from OS_IntHW() in RTOSInit.c.
All interrupts running at low priority from 1 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.

5.7   Interrupt Vector Table

The interrupt vectors can not be defined in "C" when using Microchips XC16 compiler. The interrupt vector table is part of the linker script file.

Chapter 6

STOP / WAIT Mode

6.1   Introduction

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 the function `OS_Idle()`, which you can find in embOS module `RTOSInit.c`.

Chapter 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 minimum ROM requirement for the kernel itself is about 2.500 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]
Task control block	20
Resource semaphore	8
Counting semaphore	4
Mailbox	12
Software timer	10

Chapter 8

Files shipped with embOS

List of files shipped with embOS

Directory	File	Explanation
root	*.pdf	Generic API and target specific documentation.
root	Release.html	Version control document.
root	embOSView.exe	Utility for runtime analysis, described in generic documentation.
Start\BoardSupport\		Sample workspaces and project files for MPLABX IDE, contained in manufacturer specific sub folders.
Start\Inc	RTOS.h BSP.h	Include file for embOS, to be included in every C-file using embOS functions.
Start\Lib	os??_*.a	embOS libraries for XC16 compiler.
Start\BoardSupport\..\Setup	OS_Error.c	embOS runtime error handler used in stack check or debug builds.
Start\BoardSupport\...\Setup\	*.*	CPU specific hardware routines for various CPUs.

Any additional files shipped serve as example.

Index