embOS

Real-Time
Operating System

CPU & Compiler
specifics for PIC32 using
XC32 compiler

Document: UM01058
Software version 4.34
Revision: 0
Date: April 3, 2017

SEGGER

A product of SEGGER Microcontroller GmbH & Co. KG

www.segger.com

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: April 3, 2017

Software	Revision	Date	By	Description
4.34	0	170331	TS	New software version.
4.12b	0	150921	TS	VFP support description added.
4.04a	0	141201	TS	New software version.
4.04	0	141124	TS	New software version.
4.02a	0	140924	TS	MZ support added.
4.02	0	140903	TS	First version.

About this document

Assumptions

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

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

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

How to use this manual

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

Typographic conventions for syntax

This manual uses the following typographic conventions:

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

Table 2.1: Typographic conventions

USB
j-link
SEGGER
www.segger.com
JTAG

Table of Contents

Chapter 1

Using embOS for PIC32

This chapter describes how to start with and use embOS for PIC32 core and the Microchip XC32 compiler. You should follow these steps to become familiar with embOS.

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 Microchip 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.

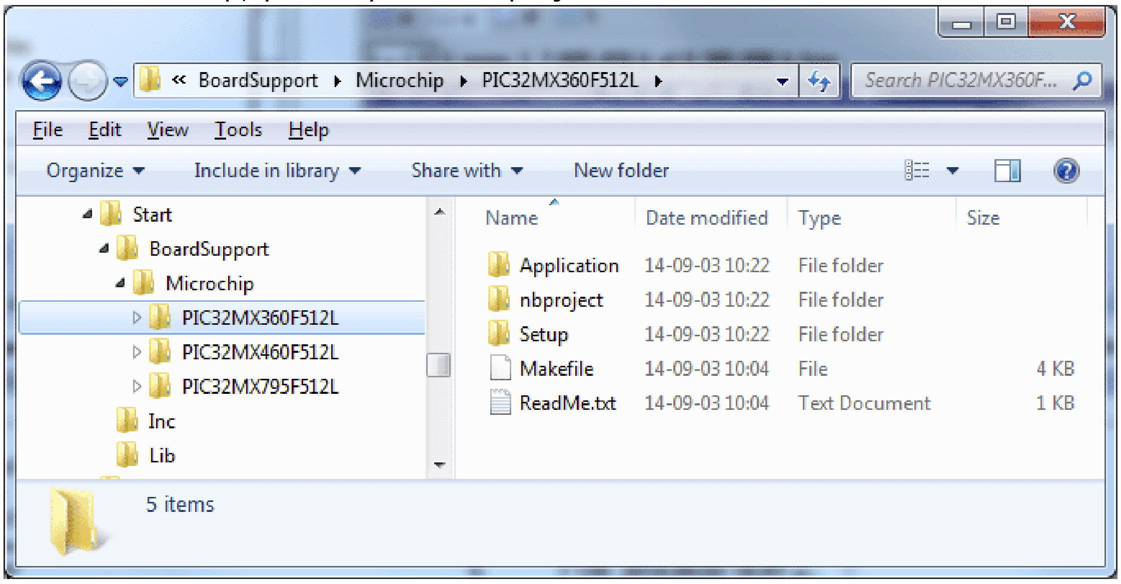
You should do this even if you do not intend to use the project manager for your application development to become familiar with embOS.

If you will not work with the embedded workbench, 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 Microchip 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 PIC32MX360F512L CPU:

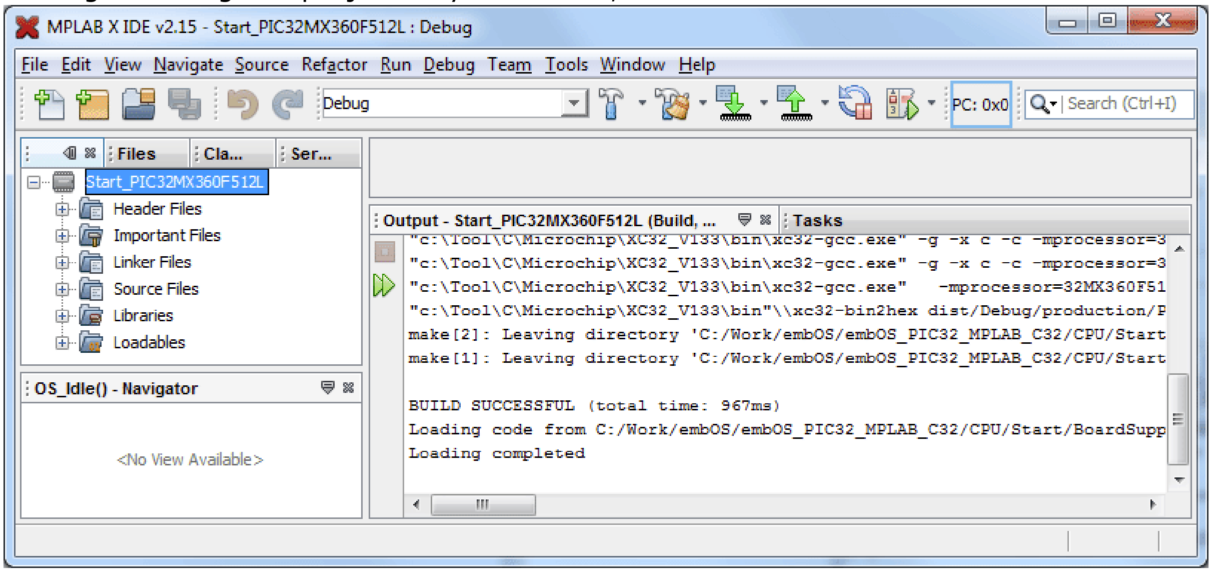


Name	Date modified	Type	Size
Application	14-09-03 10:22	File folder	
nbproject	14-09-03 10:22	File folder	
Setup	14-09-03 10:22	File folder	
Makefile	14-09-03 10:04	File	4 KB
ReadMe.txt	14-09-03 10:04	Text Document	1 KB

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\<DeviceManufactor>\<CPU>
  with the Microchip MPLABX 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:



```
"c:\Tool\C\Microchip\XC32_V133\bin\xc32-gcc.exe" -g -x c -c -mprocessor=3
"c:\Tool\C\Microchip\XC32_V133\bin\xc32-gcc.exe" -g -x c -c -mprocessor=3
"c:\Tool\C\Microchip\XC32_V133\bin\xc32-gcc.exe"   -mprocessor=32MX360F51
"c:\Tool\C\Microchip\XC32_V133\bin"\\xc32-bin2hex dist/Debug/production/P
make[2]: Leaving directory 'C:/Work/embOS/embOS_PIC32_MPLAB_C32/CPU/Start
make[1]: Leaving directory 'C:/Work/embOS/embOS_PIC32_MPLAB_C32/CPU/Start

BUILD SUCCESSFUL (total time: 967ms)
Loading code from C:/Work/embOS/embOS_PIC32_MPLAB_C32/CPU/Start/BoardSupp
Loading completed
```

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:

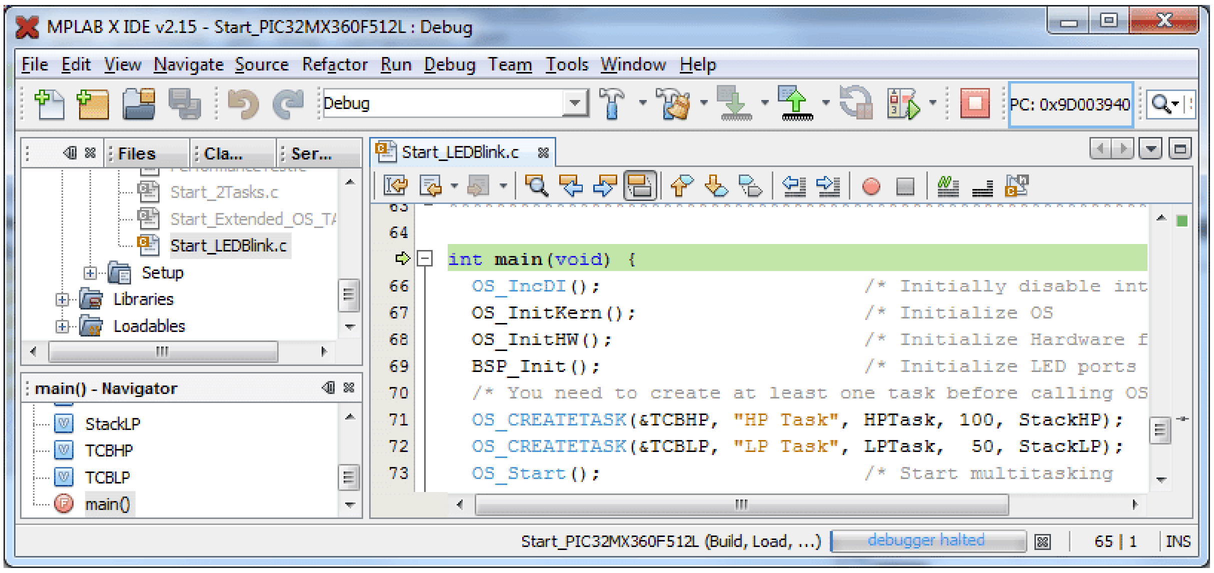
After initialization of embOS; two tasks are created and started.
The two tasks are activated and execute until they run into the delay, then suspend for the specified time and continue execution.

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

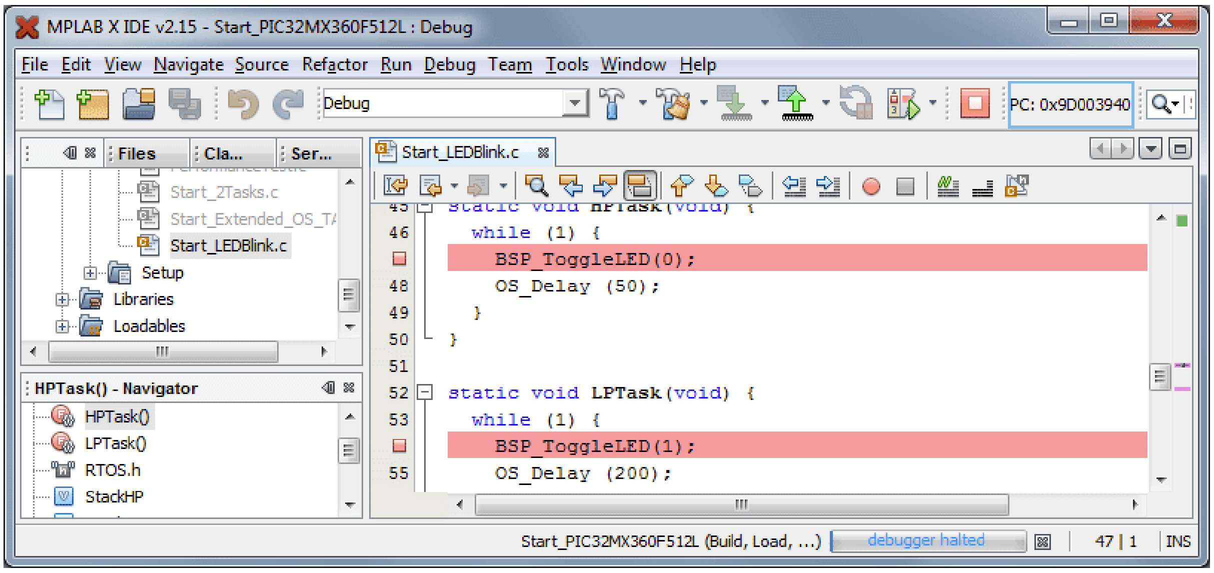
When starting the debugger, you will see the `main` function (see example screenshot below). The `main` function appears as long as the C-SPY option **Run to main** is selected, which it is by default. Now you can step through the program. `OS_IncDI()` initially disables interrupts.
`OS_InitKern()` is 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.



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

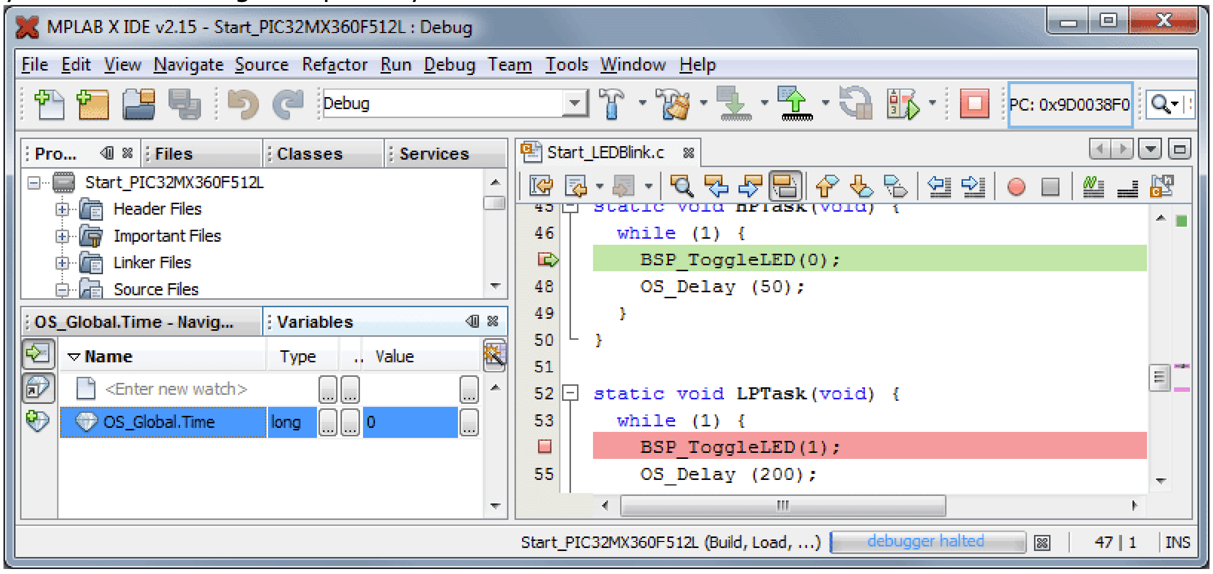
Before you step into `OS_Start()`, you should set two breakpoints in the two tasks as shown below.



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

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

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

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


MPLAB X IDE v2.15 - Start_PIC32MX360F512L : Debug

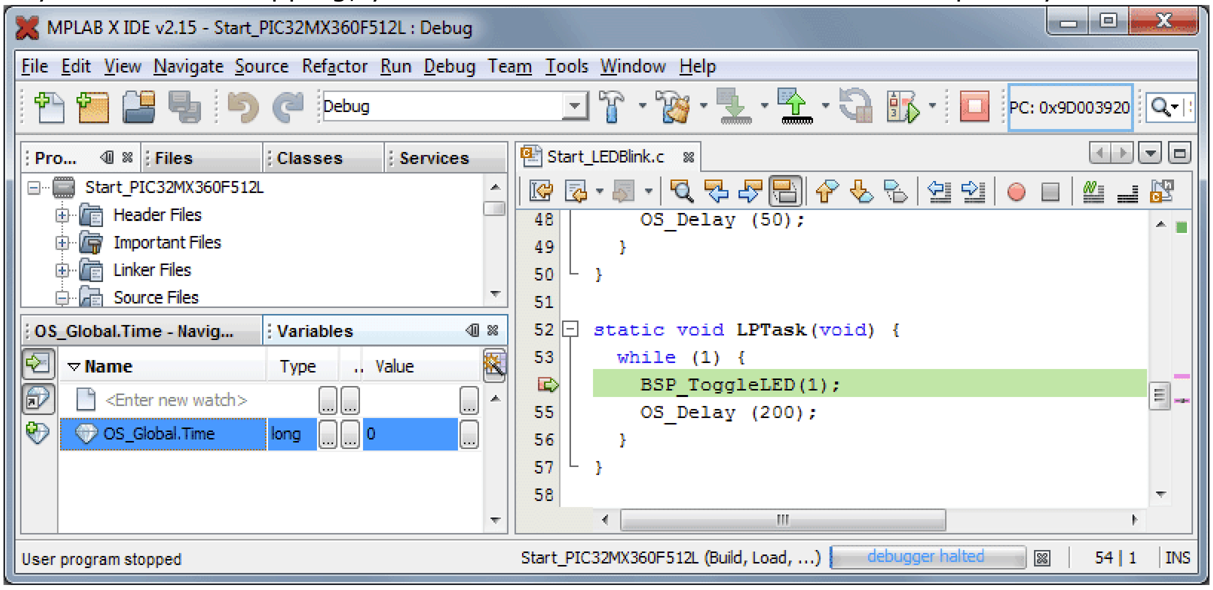
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
Debug
PC: 0x9D0038F0
Pro... Files Classes Services
Start_PIC32MX360F512L
Header Files
Important Files
Linker Files
Source Files
OS_Global.Time - Navig... Variables
<Enter new watch>
Start_LEDBlink.c

Name	Type	Value
OS_Global.Time	long	0

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

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

Start_PIC32MX360F512L (Build, Load, ...) debugger halted 47 | 1 INS

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


MPLAB X IDE v2.15 - Start_PIC32MX360F512L : Debug

File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
Debug
PC: 0x9D003920
Pro... Files Classes Services
Start_PIC32MX360F512L
Header Files
Important Files
Linker Files
Source Files
OS_Global.Time - Navig... Variables
<Enter new watch>
Start_LEDBlink.c

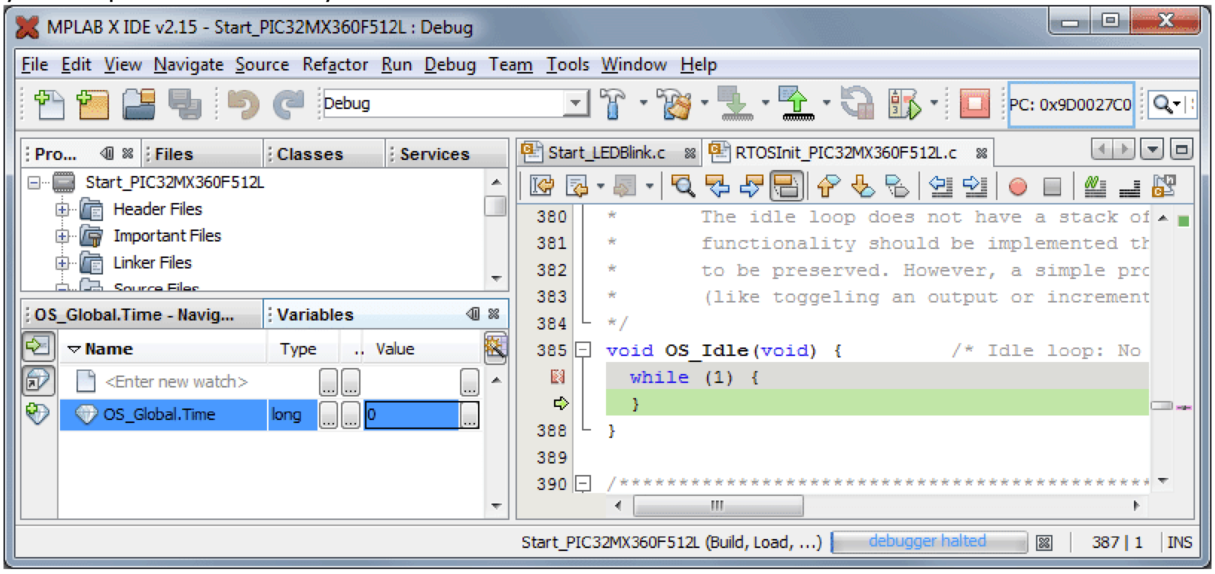
Name	Type	Value
OS_Global.Time	long	0

```
    OS_Delay (50);
  }
}

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

User program stopped Start_PIC32MX360F512L (Build, Load, ...) debugger halted 54 | 1 INS

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`.



MPLAB X IDE v2.15 - Start_PIC32MX360F512L : Debug

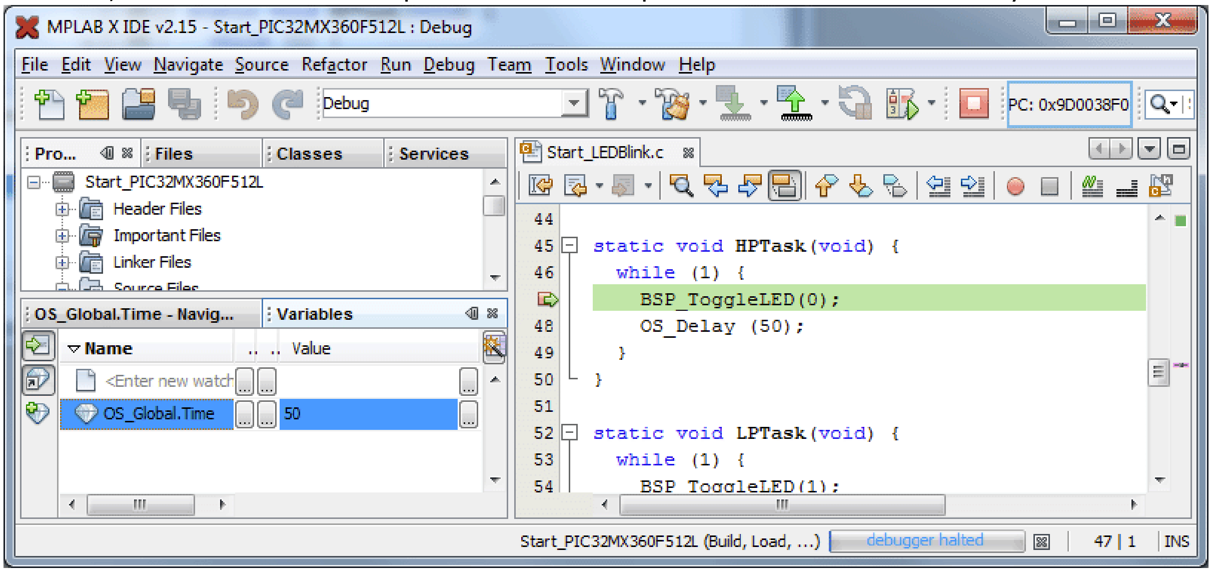
```
 *       The idle loop does not have a stack of
 *       functionality should be implemented th
 *       to be preserved. However, a simple pro
 *       (like toggeling an output or increment
 */
void OS_Idle(void) {            /* Idle loop: No
  while (1) {
  }
}

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

Name	Type	Value
OS_Global.Time	long	0

Start_PIC32MX360F512L (Build, Load, ...) debugger halted 387 | 1 INS

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.



MPLAB X IDE v2.15 - Start_PIC32MX360F512L : Debug

```

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

static void LPTask(void) {
  while (1) {
    BSP_ToggleLED(1);
```

Name	Value
OS_Global.Time	50

Start_PIC32MX360F512L (Build, Load, ...) debugger halted 47 | 1 INS

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 PIC32

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 PIC32 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 PIC32 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

PIC32 specifics

3.1 CPU modes

embOS for PIC32 supports all memory and code model combinations that the Microchip XC32 ompiler supports.

3.2 Available libraries

embOS for PIC32 and XC32 compiler comes with 35 different libraries, one for each CPU instruction mode / library mode combination.

3.2.1 Naming conventions for prebuilt libraries

embOS PIC32 for the Microchip XC32 compiler is shipped with different prebuilt libraries with different combinations of the following features:

- Instruction mode
- Library mode

The libraries are named as follows:

```
RTOS<InstructionMode><FPU><Libmode>.a
```

Parameter	Meaning	Values
`Instruction Mode`	Specifies the Mips instruction mode	Mips16: 16 bit instructions Mips32: 32 bit instructions MicroMips: 16/32 bit instructions
`FPU`	Specifies if FPU is supported.	fpu: FPU is supported
`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

RtosMips32DP.a is the library for a project using 32 bit instructions with debug and profiling support.

Note

The MicroMips embOS libraries can only be used with the PIC32MZ core.

3.3 Cache support for PIC32MZ CPUs

embOS comes with functions to support the cache of PIC32MZ.

PIC32MZ CPUs have separate data and instruction caches. embOS delivers the following functions to setup and handle the data cache.

Function	Description
`OS_PIC32MZ_DCACHE_CleanRange()`	Clean data cache
`OS_PIC32MZ_DCACHE_InvalidateRange()`	Invalidate data cache

Table 3.1: Cache handling for PIC32MZ CPUs

3.3.1 OS_PIC32MZ_DCACHE_CleanRange()

Description

`OS_PIC32MZ_DCACHE_CleanRange()` is used to clean a range in the data cache memory to ensure that the data is written from the data cache into the memory.

Prototype

```
void OS_PIC32_DCACHE_CleanRange ( void   *p,
                                  OS_U32 NumBytes );
```

Parameter	Description
p	Points to the base address of the memory area that should be updated.
NumBytes	Number of bytes which have to be written from cache to memory.

Table 3.2: OS_PIC32MZ_DCACHE_CleanRange() parameter list

Additional Information

Cleaning the data cache is needed, when data should be transferred by a DMA or other BUS master that does not use the data cache. When the CPU writes data into a cacheable area, the data might not be written into the memory immediately. When then a DMA cycle is started to transfer the data from memory to any other location or peripheral, the wrong data will be written.

Before starting a DMA transfer, a call of `OS_PIC32_DCACHE_CleanRange()` ensures, that the data is transferred from the data cache into the memory and the write buffers are drained.

3.3.2 OS_PIC32MZ_DCACHE_InvalidateRange()

Description

`OS_PIC32MZ_DCACHE_InvalidateRange()` is used to invalidate a memory area in the data cache. Invalidating means, mark all entries in the specified area as invalid. Invalidation forces re-reading the data from memory into the cache, when the specified area is accessed again.

Prototype

```
void OS_PIC32MZ_DCACHE_InvalidateRange ( void   *p,
                                         OS_U32 NumBytes );
```

Parameter	Description
p	Points to the base address of the memory area that should be updated.
NumBytes	Number of bytes which have to be written from cache to memory.

Table 3.3: OS_PIC32MZ_DCACHE_InvalidateRange() parameter list

Additional Information

This function is needed, when a DMA or other BUS master is used to transfer data into the main memory and the CPU has to process the data after the transfer.

To ensure, that the CPU processes the updated data from the memory, the cache has to be invalidated. Otherwise the CPU might read invalid data from the cache instead of the memory.

Special care has to be taken, before the data cache is invalidated. Invalidating a data area marks all entries in the data cache as invalid. If the cache contained data which was not written into the memory before, the data gets lost. Unfortunately, only complete cache lines can be invalidated.

Chapter 4

Compiler specifics

4.1 Standard system libraries

embOS for PIC32 cores and XC32 compiler may be used with XC32 standard libraries for most of all projects.

4.2 Vector Floating Point support

Some PIC32 MCUs come with an integrated vectored floating point unit.
When selecting the CPU and activating the VFP support in the project options, the compiler and linker will add efficient code which uses the VFP when floating point operations are used in the application.
With embOS, the VFP registers have to be saved and restored when preemptive or cooperative task switches are performed.
For efficiency reasons, embOS does not save and restore the VFP registers for every task automatically. The context switching time and stack load are therefore not affected when the VFP unit is not used or needed.
Saving and restoring the VFP registers can be enabled for every task individually by extending the task context of the tasks which need and use the VFP.

4.2.1 OS_ExtendTaskContext_VFP()

Description

`OS_ExtendTaskContext_VFP()` has to be called as first function in a task, when the VFP is used in the task and the VFP registers have to be added to the task context.

Prototype

`void` OS_ExtendTaskContext_VFP(`void`)

Return value

None.

Additional Information

OS_ExtendTaskContext_VFP() extends the task context to save and restore the VFP registers during context switches.
Additional task context extension for a task by calling `OS_ExtendTaskContext()` is not allowed and will call the embOS error handler `OS_Error()` in debug builds of embOS.
There is no need to extend the task context for every task. Only those tasks using the VFP for calculation have to be extended.

4.2.2 Using the VFP in interrupt service routines

Using the VFP in interrupt service routines requires additional functions to save and restore the VFP registers. embOS delivers two functions to save and restore the VFP context in an interrupt service routine.

4.2.2.1 OS_VFP_Save()

Description

`OS_VFP_Save()` has to be called as first function in an interrupt service routine, when the VFP is used in the interrupt service routine. The function saves the temporary VFP registers on the stack.

Prototype

`void` OS_VFP_Save(`void`)

Return value

None.

Additional Information

OS_VFP_Save() declares a local variable which reserves space for all floating point registers and stores the registers in the variable.
After calling the OS_VFP_Save() function, the interrupt service routine may use the VFP for calculation without destroying the saved content of the VFP registers.
To restore the registers, the ISR has to call OS_VFP_Restore() at the end.
The function may be used in any ISR regardless the priority. It is not restricted to low priority interrupt functions.

4.2.2.2 OS_VFP_Restore()

Description

`OS_VFP_Restore()` has to be called as last function in an interrupt service routine, when the VFP registers were saved by a call of OS_VFP_Save() at the beginning of the ISR. The function restores the temporary VFP registers from the stack.

Prototype

`void` OS_VFP_Restore(`void)`

Return value

None.

Additional Information

OS_VFP_Restore() restores the temporary VFP registers which were saved by a previous call of OS_VFP_Save().
It has to be used together with OS_VFP_Save() and should be the last function called in the ISR.

Chapter 5

Stacks

5.1 Task stack for PIC32

Each task uses its individual stack. The stack poiter 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 PIC32 CPUs, this minimum basic task stack size is about 48 bytes. Because any function call uses some amount of stack and all interrupts run on the task stack as well. Therefore we recommend at least 256 bytes for the task stack as a start.

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

5.2 System stack for PIC32

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 system-stack, the actual stack requirements depend on the application.

The size of the system stack can be changed by modifying the project settings.

5.3 Interrupt stack for PIC32

The PIC32 CPU has 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 6

Interrupts

The PIC32 core comes with an built in vectored interrupt controller which supports a priority controlled interrupt mode.

6.1 Interrupt processing with PIC32 CPUs

PIC32 CPUs support a priority controlled interrupt mode. This mode supports the following features:

- Interrupt priority registers to assign 7 priority levels to peripheral interrupts.
- Priority level controlled masking.
- Interrupts with higher priority are never disabled by entering an interrupt service routine with lower priority

6.2 What happens when an interrupt occurs?

- The CPU-core receives an interrupt request
- As soon as the processor interrupt priority level is below to the interrupt priority level, the interrupt is executed
- The CPU calls the interrupt service routine, that belongs to the received interrupt.
- User defined functionality in interrupt service routine
- Execute `eret` to return from interrupt
- For details, please refer to Microchip user manual

6.3 Zero latency interrupts with PIC32 CPUs

Instead of disabling interrupts when embOS does atomic operations, the interrupt level of the CPU is set to a higher user definable level. Therefore all interrupts with higher levels can still be processed.
These interrupts are named zero latency interrupts.
The default level limit for zero latency interrupts is set to 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.

6.4 Interrupt priorities with embOS for PIC32 CPUs

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()`.

6.5 Interrupt Stack

The PIC32 has no own interrupt stack pointer. The interrupts uses the task stack instead. Be sure to define your task stack size big enough, so that all nested interrupt routines can run on this stack!

6.6 Defining interrupt handlers for PIC32 CPUs

Since the PIC32 compiler generates different interrupt function entry code depending on the compiler optimization level assembler interrupt routines are necessary. These assembler interrupt routines call the C interrupt handler, which is just an usual C function. The assembler interrupt routine includes only one line of code:

```
OS_CALLISR <HandlerFunction> <Priority>
```

Parameter	Meaning
HandlerFunction	Name of the C interrupt handler function
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

The interrupt handler used by embOS are implemented in OS_ISR.S file and in the CPU specific RTOSInit_*.c files.

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

Example of an embOS assembler interrupt handler

```
# ####################################################################
# #
# #       void OS_SysTick_ISR(void)
# #
# #       Wrapper function for OS_Systick.
# #       This function saves and restores all necessary registers.
# #       Interrupts are not enabled (MPLAB compiler does this in
# #       interrupt function prolog).
# #
OS_SysTick_ISR:
         OS_CALL_ISR  OS_Systick OS_INT_PRIORITY_1
```

The assembler interrupt function has to be declared with the following prototype:

```
void __attribute__( (interrupt(<Priority>),
  vector(VectorNumber))) <AssemblerISRName>( void );
```

Example of an embOS C interrupt handler

```
void __attribute__( (interrupt(ipl1), vector(0))) OS_SysTick_ISR( void );

/*********************************************************************
*
*       OS_Systick()
*
*       Interrupt handler function for core timer
*
*/
void OS_Systick(void) {
  OS_I32 t, t1;
  IFS0CLR = 0x00000001;       // reset core timer interrupt pending flag
  OS_EnterNestableInterrupt();
  …
  …
  OS_LeaveNestableInterrupt();
}
```

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 C interrupt handler has to call `OS_EnterInterrupt()/OS_LeaveInterrupt()`, when interrupts should not be nested.
- It has to call `OS_EnterNestableInterrupt()/OS_LeaveNestableInterrupt()`, when nesting should be allowed.

Differences between OS_EnterInterrupt) and OS_EnterNestableInterrupt()

`OS_EnterInterrupt()` should be used when the corresponding interrupt should not be interrupted by another embOS interrupt. `OS_EnterInterrupt()` sets the interrupt priority of the CPU to the user definable interrupt priority level, thus locking any other embOS interrupt, Fast interrupts are not disabled.

`OS_EnterNestableInterrupt()` should be used as entry function, when interruption by higher prioritized embOS interrupts should be allowed.
`OS_EnterNestableInterrupt()` does not alter the interrupt priority of the CPU, thus keeping all interrupts with higher priority enabled.

Example of a Zero latency interrupt handler

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

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

The rules for a zero latency interrupt handler are as follows:

- Local variables may be used.
- Other functions may be called.
- 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.

6.7 OS_SetFastIntPriorityLimit()

The interrupt priority limit for fast interrupts is set to 5 by default. This means, all interrupts with higher priority from 6 to 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	Meaning
`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.

Additional information

To disable zero latency interrupts at all, the priority limit may be set to 7 which is the highest interrupt priority for interrupts.
To modify the default priority limit, `OS_SetFastIntPriorityLimit()` should be called before embOS was started.

All interrupts running at low priority from 1 to the user definable priority limit for zero latency interrupts have to call `OS_EnterInterrupt()/OS_LeaveInterrupt()` or `OS_EnterNestableInterrupt()/OS_LeaveNestableInterrupt()` 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.
Any interrupts running above the zero latency interrupt priority limit must not call any embOS function.

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	36
Resource semaphore	16
Counting semaphore	8
Mailbox	24
Software timer	20

Index