# Q▪Kernel™

## *Pic24 dsPIC Porting guide*

*Version 6.0-3363*

*Q▪Kernel™ is a product of QuasarSoft Ltd.*

## License



QuasarSoft Ltd

312-5th Avenue Suite No. 354

Cochrane Alberta T4C 2E3

Canada

Tel. +1 (403) 450 3482

*www.quasarsoft.com*

## About this Document

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

- The software tools used for building your application, mainly the compiler and linker
- The C Programming language
- The processor

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, covers the ANSI C standard.

The Q▪Kernel™ Reference Guide is available to learn the API and the Q▪Kernel™ User Guide to learn how to use Q▪Kernel™.

## How to Use this Manual

The intention of this manual is to give you a detailed description for the PIC24/dsPIC

1. Introduction to PIC24 and dsPIC ..... 5
1.1. Differences between MCU's ..... 5
1.1.1 Upper 32Kb Data Space Window ..... 5
1.1.2 Table Page (TBLPAG) ..... 5
1.1.3 DSP Unit ..... 6
1.2. Grouping the different MCU's ..... 6
1.2.1 Generic group ..... 6
1.2.2 GenericDA group ..... 6
1.2.3 GenericEP group ..... 6
1.3. Pre-build libraries ..... 7
2. Installation ..... 8
3. Adding Q-Kernel™ to your application ..... 9
3.1. Using an object library ..... 9
3.2. Using a project library ..... 9
3.3. Using your own object library ..... 9
3.4. Creating a new application ..... 10
4. Resources used by Q-Kernel™ ..... 17
4.1.1 Specific Errors ..... 17
4.1.2 Stack Limiting Features ..... 17
4.1.3 Interrupts ..... 17

# 1. Introduction to PIC24 and dsPIC

Q▪Kernel™ is a Tick-Less Dual-Mode Real Time Operating System (RTOS) sometimes referred to as a kernel. Q▪Kernel™ is specially created for the modern processors like the PIC24 and dsPIC and fully exploits the power of these processors.

Q▪Kernel™ supports all 16-bit PIC microprocessors including the dsPICs. The minimum system requirements are 2kb RAM and 32Kb Flash. Q▪Kernel™ supports the following compilers:

- MPLAB-X Version 1.8 or higher
- XC16 Compiler Version 1.1 or higher.
- C30 compiler version 3.31 or higher
- MPLAB8.x is supported but project files are not included.

The system is tested with MPLAB-X 2.00.

## 1.1. Differences between MCU's

Not all version of the PIC24 and dsPIC are the same. The EP models have a slightly different instruction set and op-code. The other models can be separated how they address the upper 32kB address space.

### 1.1.1 Upper 32Kb Data Space Window

The 16-bit PIC micro controllers divide their data address space in two 32 Kb windows. The lower window maps the Special Function Registers (SFR's) and the RAM. The second window mapping depends on the type of processor.

Most of the controllers map flash memory for constant data. This process is called program space visibility because part of the program space is visible in data memory and the register PSVPAG specifies where in flash the 32Kb is mapped. This is limited to 32Kb. If there is more than 32Kb of constant data the PSVPAG must be maintained. The user is responsible that the PSVPAG is maintained and pointers into the PSV space cannot be used by more than one thread without special facilities. Pointers in RAM don't have that problem because they always point to the lower 32 Kb for all threads and fibers. Please read the XC16 or C30 user manual that describes this process and all options.

Newer controllers implement Extended Data Space. This feature can map flash or ram in the window. The system uses separate registers for read and writes. (DSRPAG/DSWPAG)

The system will persist PSVPAG or DSRPAG/DSWPAG in the context of the thread and fibers. The Q▪Kernel™ interrupts qISR() and qISR_FAST() also persist this information. The developer is responsible for saving and restoring the information in native interrupts.

### 1.1.2 Table Page (TBLPAG)

The 16-bit PIC micro controllers have the ability to access any flash in the controller by its 24-bit address. The TBLPAG register and some special instructions are used facilitate this. This register has to be saved during context switches if multiple

threads use the TBLPAG. As described above the TBLPAG register is persisted together with the PSVPAG register on controllers that have a PSVPAG register.

Controllers with Extended Data Space don't have to use the TBLPAG but can use the far superior Extended Data Space feature instead. Using the TBLPAG in multiple threads and fibers is still possible by using it within a critical section. The TBLPAG can be used in interrupts on those controllers but the developer is responsible for saving and restoring the TBLPAG register in both Q▪Kernel™ interrupts and native interrupts.

### 1.1.3 DSP Unit

The DSP unit is only available in the dsPIC30 and dsPIC33. The DSP functionality can only be used in fibers and not in threads. This is the developers' responsibility and the system will and cannot check this.

## 1.2. Grouping the different MCU's

Q▪Kernel™ comes in source code and as an object library. Because different MCU's use different memory mapping methods and not all instructions are supported by all MCU's Microchip has divided the MCU is 3 device groups. See Readme_XC16.html in the doc directory. The following groups are defined:

- Generic core devices
- Generic "DA" devices
- Generic "EP" devices

The simplest way to check to which group your device belongs is to answer he follow questions:

1. If the device is a PIC24EP or dsPIC33EP it belongs to the GenericEP group.
2. If the device has a PSVPAG register it belongs to the Generic group
3. All other devices belong to the GenericDA group

### 1.2.1 Generic group

These are MCU's with a PSVPAG register to address the top 32KB of the address space. All older smaller MCU fall in this category. Examples of those MCU's are PIC24FJ128GA010 and dsPIC33FJ64GP204.

### 1.2.2 GenericDA group

These are MCU's with a DSWPAG and DSRPAG registers to address the top 32KB of the address space. Most of the newer larger MCU's fall in this category. Examples of those MCU's are PIC24FJ256DA210 and PIC24FJ64GA310.

### 1.2.3 GenericEP group

These are MCU's with a DSWPAG and DSRPAG registers to address the top 32KB of the address space. Their instruction set and op-code is different than the previous groups. These are the MCU's with EP in their name like PIC24EP256GU810 and dsPIC33EP64MC202.

## 1.3. Pre-build libraries

This port comes with 4 prebuild libraries. Generic, GenericDA, GenericEP and ThreadMetric. The first 3 are for the different MCU groups and are compiled with parameter checking and the small code (-s) optimization. ThreadMetric is compiled with the fastest code optimization (-3) and no parameter checking.

# 2. Installation

Q▪Kernel™ is distributed as a zip file with the name qKernelV3353.zip. The version contains all versions of Q▪Kernel™ in that build. The last 4 digits in the name is the build number. Extract the file into a main directory like C:\qKernelFree, but it can be any drive or name because all references are relative. An example of the directory structure is specified below.

```
V3353
 ----- Documentation
 ----- Source
 ----- ------Pic24_MPLAB.X (PIC24 /dsPIC port)
 ----- ------ -----Blinky.X (Test program)
 ----- ------ ----- ----- dist
 ----- ------ ----- ----- ------default
 ----- ------ ----- ----- ----------production
 ----- ------ ----- ----- nbproject
 ----- ------ -----dist (Distribution for the Q▪Kernel™ libraries)
 ----- ------ ----- ----- default
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- Generic
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- GenericDA
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- GenericEP
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- ThreadMetric
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ -----nbproject
 ----- ------ -----ThreadMetric.X
 ----- ------ ----- ----- dist
 ----- ------ ----- ----- ------default
 ----- ------ ----- ----- ----------production
 ----- ------ ----- ----- nbproject
 ----- ------ ----- ----- ------private
 ----- ------Pic32_MPLAB.X (PIC32 port)
 ----- ------ -----Blinky.X (Test program)
 ----- ------ ----- ----- dist
 ----- ------ ----- ----- ------default
 ----- ------ ----- ----- ----------production
 ----- ------ ----- ----- nbproject
 ----- ------ -----dist (Distribution for the Q▪Kernel™ libraries)
 ----- ------ ----- ----- default
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ ----- ----- Generic
 ----- ------ ----- ----- ------production (Library Pic24_MPLAB.X.a)
 ----- ------ -----nbproject
```

# 3. Adding Q▪Kernel™ to your application

Q▪Kernel™ can be included as a project library or as an object library. We recommend to use the object library because this prevents building the library every time.

Your application need to include “qKernel.h” in all source files in your project. The best way to do that is to add the path to this file in the “C include dirs.” of the compiler. Select your project properties, select xc16-gcc and select option category “PreProcessing and messages” Then add the path to “C include dirs.”. You can also just specify the full path.

## 3.1. Using an object library

As described above you can add Q▪Kernel™ as an object library. Include one of the standard configurations that is included in the distribution. The libraries are in the Generic, GenericDA or GenericEP directories. The libraries are compiled with -s as optimization level, kernel timer is TMR4/TMR5 and the full parameter checking.

## 3.2. Using a project library

You create your application and include Q▪Kernel™ as a library project. You go to properties of your project and click libraries. There you click “Add library Project” and select qKernel.X. In this case the source will be included and MPLAB will compile the source if required. You have to specify the configuration, for things like MCU, optimization level, etc.

Q▪Kernel™ consists of many files so only the code that is required will be in flash. This makes building slow and even if a full build executes sporadic it is advisable to use the full computer potential. You can improve the build time by selecting Options from the main menu, then Embedded and select the tab “Project Options” Select the option “Use parallel make” to speed up the build. It will use all your cores of your processors.

***Improve the compile speed by selecting the option “Use parallel make” and use all the available cores***

You can use all other compile and build options, like memory model, optimization levels, etc. for your project or Q▪Kernel™ so it is very flexible.

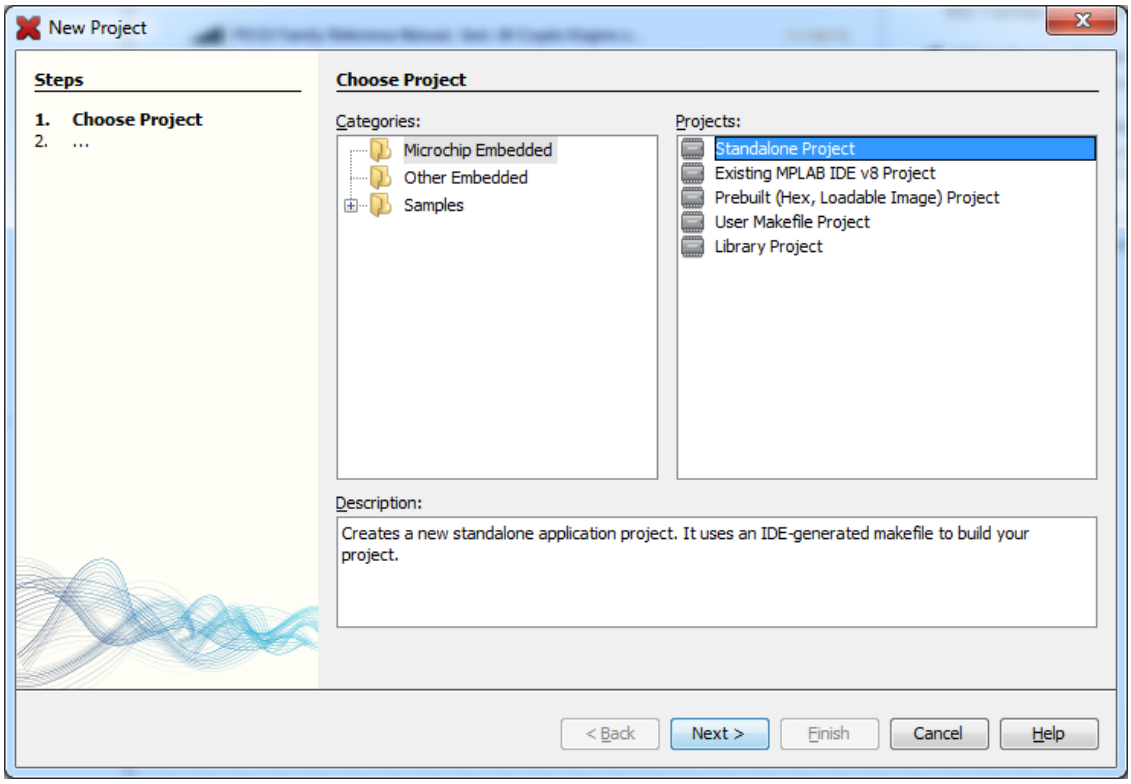
## 3.3. Using your own object library

First create a configuration within the Q▪Kernel™ distribution and set your compile options. Build the system and include the created object library in your application.

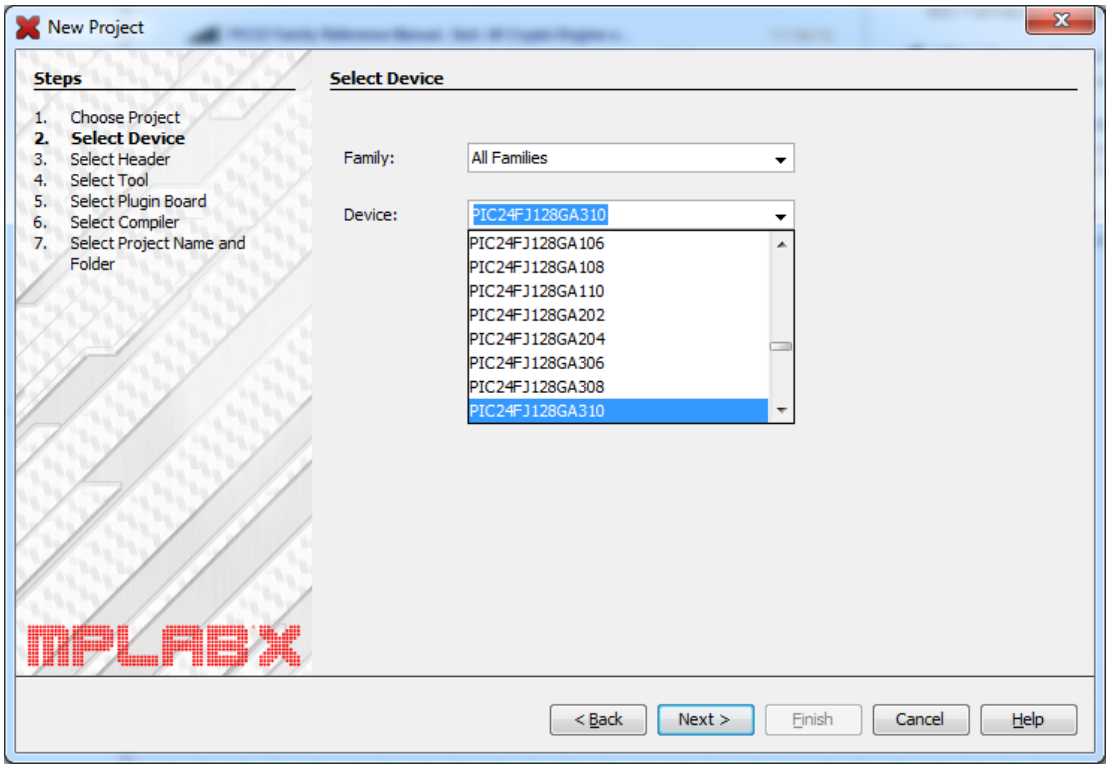
## 3.4. Creating a new application

This example creates a simple application with a . While it looks complicated the whole thing is very simple. See the following screen shots.

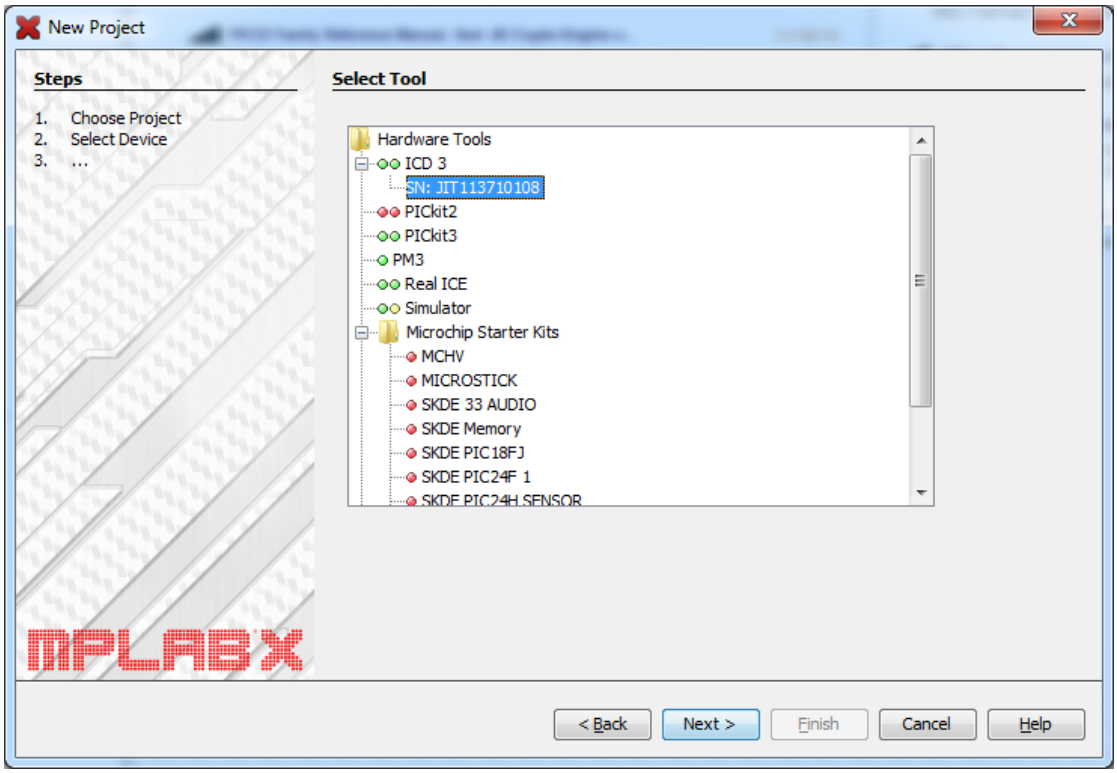
Start a new project



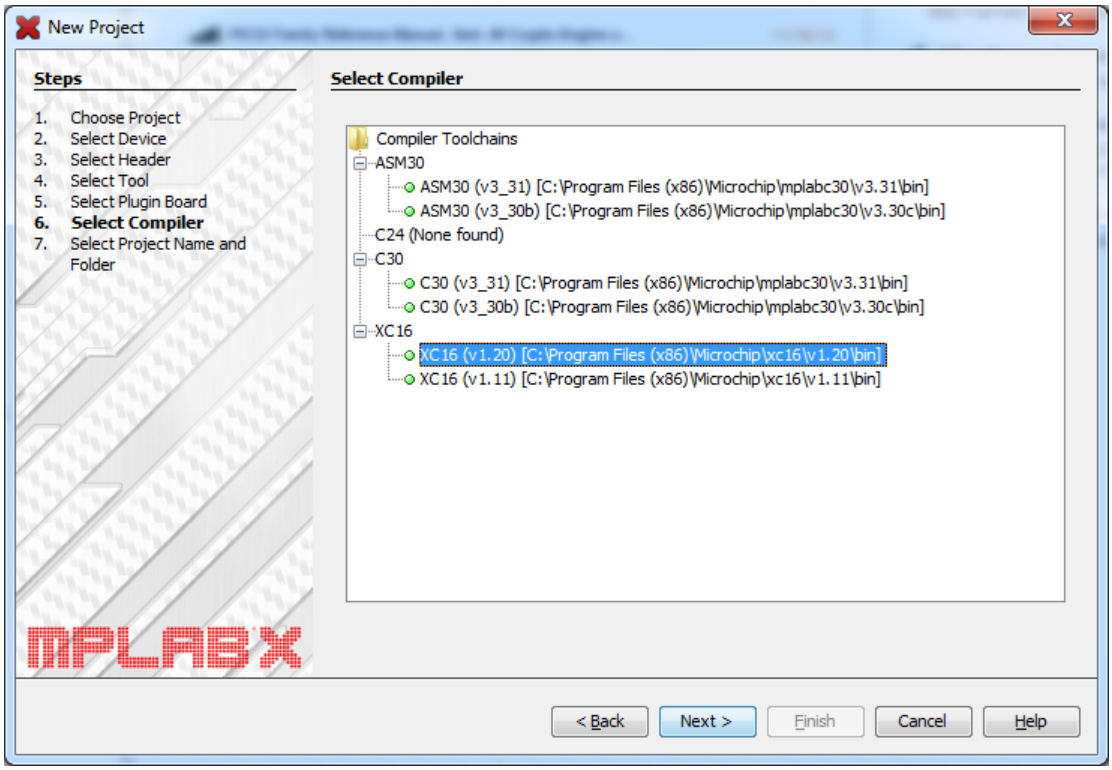
Select the processor



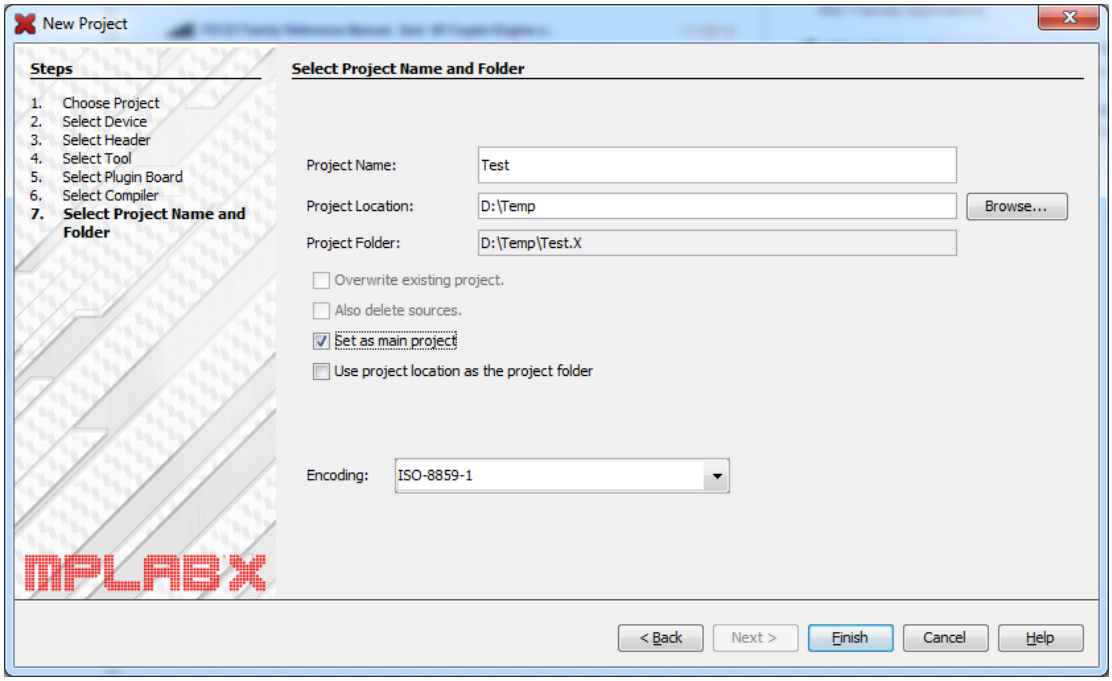

Select the debugging tool (ICD3)



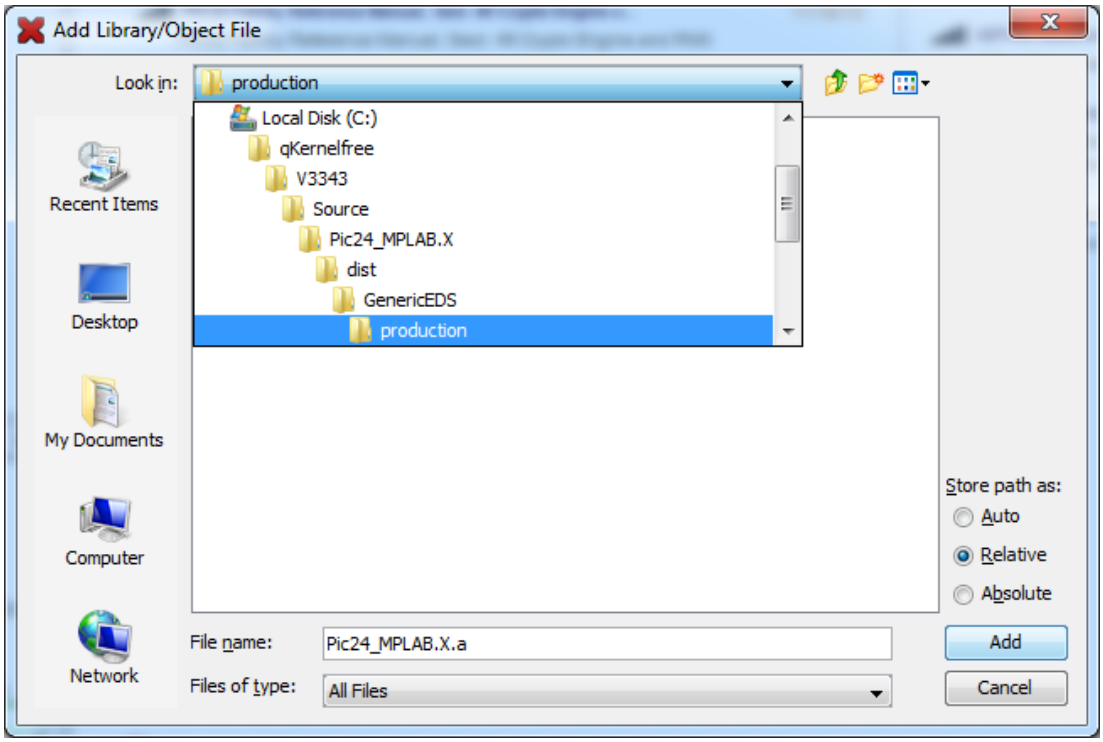
Select a compiler



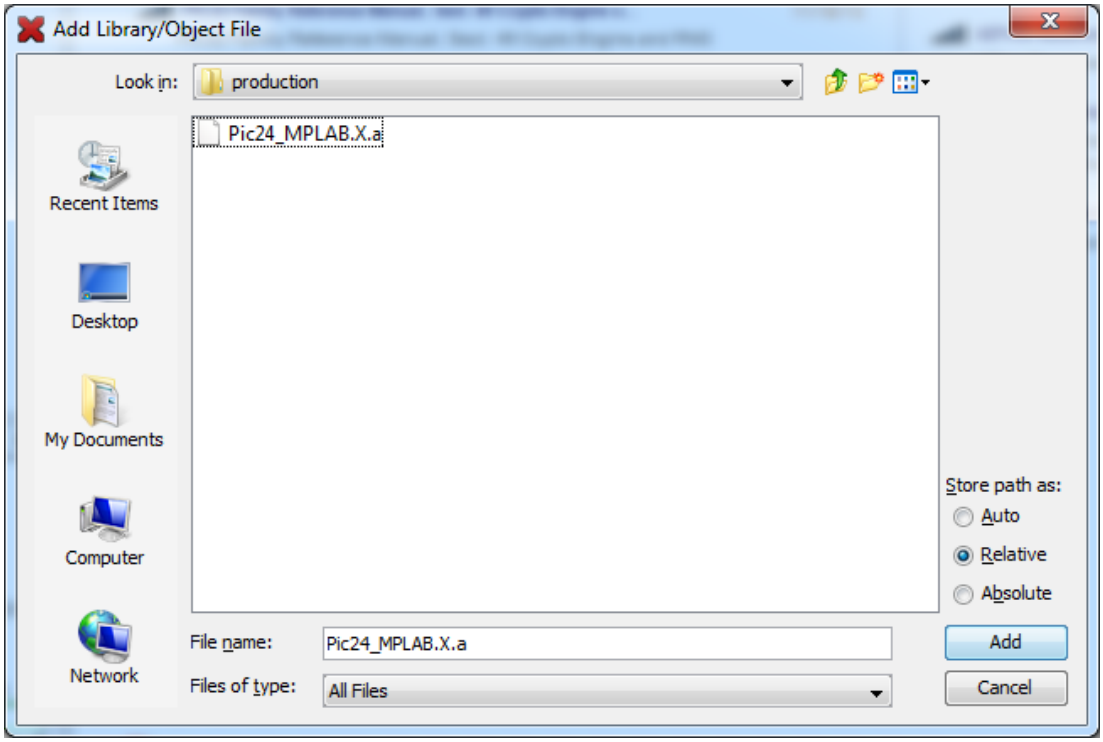

And the project name and folder (See that we use the D: drive)



Add an object library directory (We use GenericEDS because that's our processor)



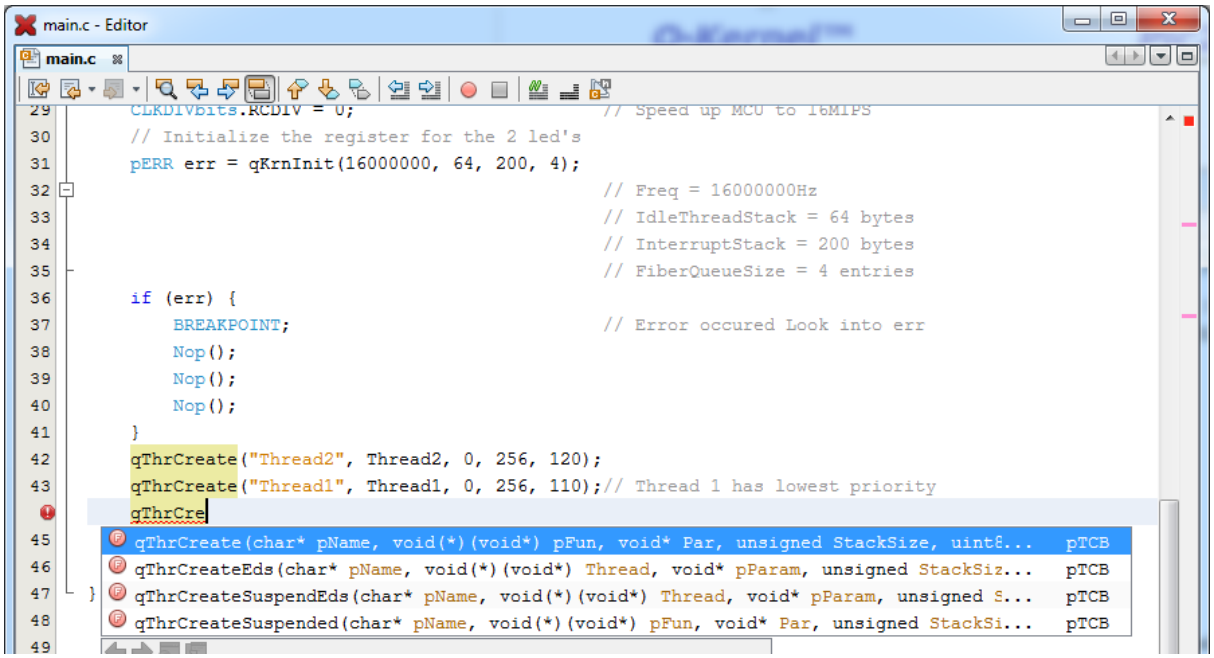

And select the file itself by pressing the Add button



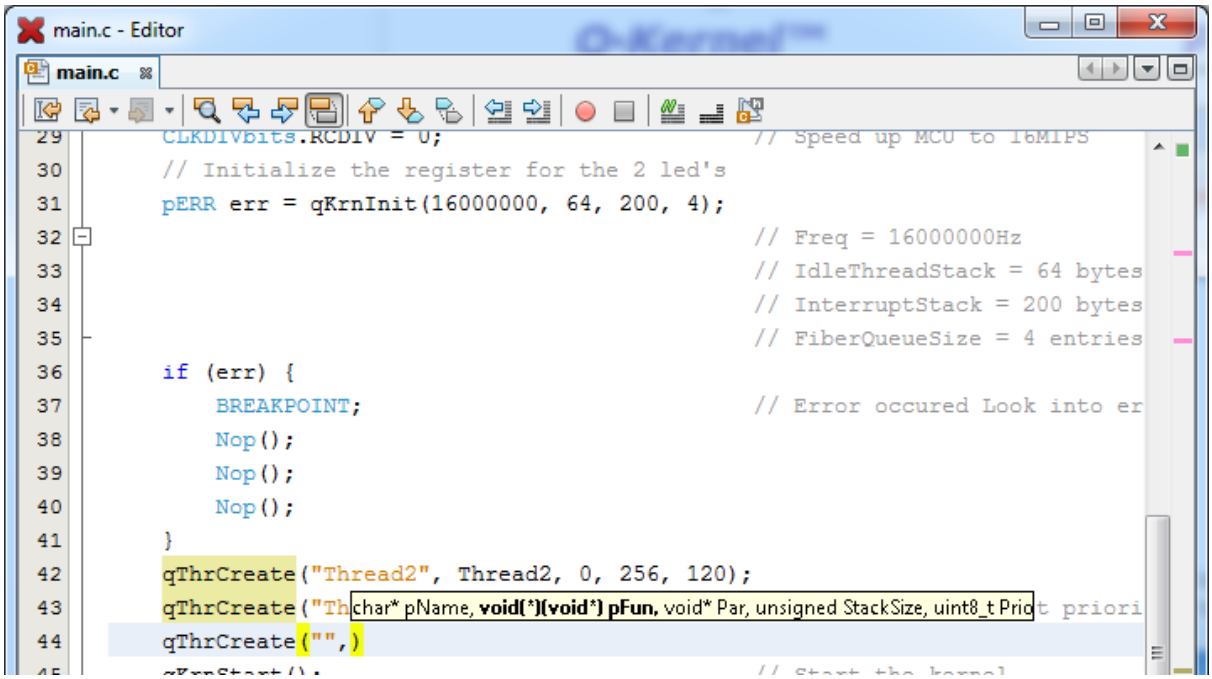
Now create a main c file and add some code like this example (See blinky)

```
#include <stdio.h>
#include <stdlib.h>
#include "C:\qKernelfree\V3343\Source\Pic24_MPLAB.X\qKernel.h"

void Thread1(void* Dummy) {
    while (1) {
        //Toggle led here
        qThrSleep(300000);
    }
}

void Thread2(void* Dummy) {
    while (1) {
        //Toggle (another) led here
        qThrSleep(100000);
    }
}

unsigned qErrNotify(unsigned Error) {           // Central error handling
    if (Error==0) {                             // If error = 0 just return
        return 0;
    }
    qKrnError(Error);                           // This will reset the system..
    return Error;                               // ..and trigger breakpoint
}


int main(int argc, char** argv) {
    CLKDIVbits.RCDIV = 0;                       // Speed up MCU to 16MIPS
    // Initialize the register for the 2 led's
    pERR err = qKrnInit(16000000, 64, 200, 4);
                                                // Freq = 16000000Hz
                                                // IdleThreadStack = 64 bytes
                                                // InterruptStack = 200 bytes
                                                // FiberQueueSize = 4 entries
    if (err) {
        BREAKPOINT;                             // Error occured Look into err
        Nop();
        Nop();
        Nop();
    }
    qThrCreate("Thread2", Thread2, 0, 256, 120);
    qThrCreate("Thread1", Thread1, 0, 256, 110);// Thread 1 has lowest priority
    qKrnStart();                                // Start the kernel
    return 0;                                   // We should never get here
}
```

During typing you can use <ctrl><space> to find the function name.




During typing MPLAB helps you with the parameters



Some remarks:

1. The #include statement (line 3) in the program needs to find file qKernel.h in the porting directory. In the Blinky.X example we use a relative specification because both Q▪Kernel™ and the project are on the C: drive. Use the “preprocessing and messages” dialog and specify the “C include dirs.”
2. Because this is a PIC24 with a DSRPAG and DSWPAG we use the GenericDA configuration. (the picture uses the old name GenericEDS)
3. While it is possible to add Q▪Kernel™ as a project we do not recommend this because it will compile and build the library every time.
4. Even if Q▪Kernel™ is included as an object library (versus a project) the debugger will find all sources related to the object and the cursor (green line) will show the Q▪Kernel™ code.

# 4. Resources used by Q·Kernel™

Q·Kernel™ will use a 32-bit timer to control its timing. The default timer is TMR4/TMR5 but this can be changed. The PIC24 timer interrupt is TMR5 is used in 32-bit mode and this will free-up TMR4 interrupt, which is used as kernel interrupt.

### 4.1.1 Specific Errors

The Q·Kernel™ code signals traps by calling the qErrNotify() function. The following traps are included as Q·Kernel™ errors:

- OscillatorFail (0x1000)
- AddressError (0x1001)
- HardTrapError (0x1002)
- StackError (0x1003)
- MathError (0x1004)
- DMACError (0x1005)
- SoftTrapError (0x1006)

### 4.1.2 Stack Limiting Features

The stack limiting features of the PIC24/dsPIC architecture are used extensively. Every individual thread and the interrupt stack keep track of its own stack-limit by manipulating the SPLIM register. This feature does not prevent stack errors but makes it simpler to detect them.

### 4.1.3 Interrupts

The Q·Kernel™ version for the PIC24/dsPIC will never disables interrupts and has a fixed interrupt latency of 4 cycles (PIC24F and H). The EP versions have a fixed or variable interrupt latency.

The qISR() function uses 6 byte on the thread stack (2 for WREG0 and 4 for the return address) and it needs 18 cycles to switch the stack and start execution of the interrupt code. This is about 6 cycles more compared to native interrupt handler. The difference is the switch to the interrupt stack. Without this all thread has to add stack-space for the interrupts.

The qISR_FAST() function uses 4 byte on the thread stack (for the return address) and it needs 15 cycles to switch the stack and start execution of the interrupt code. This is about 6 cycles more compared to native interrupt handler. The difference is the switch to the interrupt stack. Without this all thread has to add stack-space for the interrupts.