Q▪Kernel™

Pic32 Porting guide

Version 6.0-3352

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 PIC32.

1. Introduction to PIC32

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, dsPIC, PIC32 and fully exploits the power of these processors.

Q▪Kernel™ supports most of the PIC32 processors and the development environment. The minimum system requirements are 4kb RAM and 64Kb Flash. Q▪Kernel™ is tested with the following compilers and IDE:

- MPLAB-X Version 2.0 or higher
- XC32 Compiler Version 1.31 or higher.

The version should also work with

- MPLAB-X Version 1.8 or higher
- XC32 Compiler Version 1 or higher.
- C32 compiler version 2 or higher
- MPLAB8.x or higher

1.1. Supported PIC32 series

Currently all MX devices are supported. The MZ series is not supported but will be supported in the future.

2. 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. Software interrupt 0 is used as kernel interrupt.

2.1. Stack Limiting Features

The PIC32 architecture does not support stack limiting features.

3. Architecture

The PIC32 contains a MIPS core which uses a load/store architecture. Operations like Add, Subtract, logical operations, etc. are always performed on registers. This introduces some challenges for Q-Kernel because it requires atomic operations on data and the load store architecture does not provide that. The architecture provides load-linked and store-conditional instructions. Load-link returns the current value of a memory location, while a subsequent store-conditional to the same memory location will store a new value only if no updates have occurred to that location since the load-link. Together, this implements a lock-free atomic read-modify-write operation. All atomic operations are implemented with this mechanism and interrupts do not have to be disabled.

3.1. Interrupts

The Q▪Kernel™ version for the PIC32 will never[1] disables interrupts on its own but the architecture of the PIC32 will disable interrupts for a very short time during part of the prologue and epilogue of an interrupt handler. This will introduce some interrupt jitter but is very limited.

The qISR(vector) function uses zero bytes on the thread stack and it needs about 40 cycles to switch the stack and start execution of the interrupt code. This includes the interrupt prologue and saving the registers. During this process the PIC32 disables interrupts during 10 cycles.

The qISR_FAST(vector) function uses zero bytes on the thread stack and it needs about 24 cycles to switch the stack and start execution of the interrupt code. This includes the interrupt prologue and saving the registers. During this process the PIC32 disables interrupts during 10 cycles.

The vector in both function is a number. Please refer to the processor manual for a list of vectors.

3.1.1 Interrupt stack memory savings

The PIC32 has 32 main registers plus a few support registers. This means that during an interrupt the system has to save over 100 bytes. This is a minimum because the interrupt itself will use the stack for local variables and calling other functions. A value of 150 bytes per interrupt is a more realistic value. Because interrupts service routines can be interrupted a maximum of 6 times 150 bytes or 900 bytes need to be saved on the stack. Without and interrupt stack an application with 16 threads needs to reserve 900 bytes per thread which is 10,800 bytes of memory. With an interrupt stack the system just needs in interrupt stack of 900 bytes. The savings will be almost 10KB.

[1] Q-Kernels claim that it has zero interrupt latency is still valid. Zero interrupt latency is based on the fact that the RTOS does not add latency to the system other than based on the architecture. The PIC32 cannot claim zero interrupt jitter because of the architecture but limits it to an absolute minimum.

3.2. Context switching

Because of the large amount of registers of the PIC32, context switching requires more time. The minimum time for context switching is about 80 cycles but a more practical number is 120 cycles because of some extra overhead.

Systems without the segmented interrupt architecture have to disable interrupt processing for 120 cycles which introduces a lot of interrupt jitter. Because those systems disable interrupt in every critical section the system has a lot of jitter.

Q-Kernel does not add any interrupt jitter but the PIC32 architecture disables interrupts for 10 cycles so the system is not completely jitter free.

4. 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
```

5. Adding Q▪Kernel™ to your application

Q▪Kernel™ can be included as a library in your project. 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.

You 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 xc32-gcc and select option category “PreProcessing and messages” Then add the path to “C include dirs.”.

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.

5.1. Using an object library

It is also possible to add Q▪Kernel™ as an object library. There are two options to do this. Create a specific configuration for your project, including the MCU to use and build Q▪Kernel™. Then include that object library in your project.

It is also possible to include the standard Q-Kernel distribution. This library is compiled with -s as optimization level, kernel timer is TMR4/TMR5 and the full parameter checking.