Distribution	Software Documentation	MS080929001	1/9
TO: Users of the freeRTOS port for NEC microcontroller 78K0R/Kx3	**Startup Sample for 78K0R/Kx3 Microcontroller family freeRTOS port**	Date: 29. September, 2008	**NEC-Electronics (Europe) GmbH Technical Product Support ABG.TPS.STS**

Startup Sample for 78K0R/Kx3 Microcontroller family FreeRTOS port

Revision History

Item	Description/changes from the previous version	Previous Version	New Version	Date
1.	Initial Version	-	1.0	29.09.2008

Table of contents

1. Purpose of this document

This document describes the Startup sample of the freeRTOS port for the NEC microcontroller family 78K0R/Kx3. The function of the given sample is explained as well as the settings which can be changed by the user to adjust the clock frequency or the microcontroller memory model for example.

2. Abbreviations

Abbreviation	Explanation
RTOS	Real Time Operating System
TAU	Timer Array Unit
TDR	Timer Data Register
ISR	Interrupt service routine
PC	Program Counter
PSW	Program Status Word
SP	Stack Pointer
TCB	Task Control Block
LED	Light Emitting Diode
CPU	Central Processing Unit

Table 1 Abbreviations

3. Licensing

FreeRTOS is licensed under a modified GPL and can be used in commercial applications under this license. An alternative commercial license option is also available if required.
The FreeRTOS source code is licensed by the GNU General Public License (GPL) with an exception. The full text of the GPL and the text of the exception are available on the FreeRTOS.org web site.
The exception permits the source code of applications that use FreeRTOS solely through the API published on the FreeRTOS.org web site to remain closed source, thus permitting the use of FreeRTOS in commercial applications without requiring that the whole application be open sourced.

4. Introduction

The freeRTOS.org project delivers a Real Time Operating System (RTOS), which can be ported to a wide band of microcontrollers. In this function the RTOS can also be ported to the 78K0R/Kx3 microcontroller family of NEC Electronics.
The RTOS works by a scheduled task handling which uses a tick timer to control the task active time. By the kernel two operating modes are predefined and can be selected. The first one is the easiest implementation were only a task can deactivate itself to give other task processing time, which is called cooperative mode. The second mode, called preemptive mode, uses in adaption to the first mode the time triggered task interruption using the tick timer interrupt to perform a task switch.
For further information about these freeRTOS kernel specific settings refer to the freeRTOS.org homepage.
The given Startup Sample is designed as an easy introduction to this RTOS showing the functionality of tasks, queues and semaphores.

5. Programming environment

The given example is designed with the **IAR Systems Embedded Workbench for NEC 78K0R v4.60b**. It is also checked on the older IAR Systems Embedded Workbench v4.50. For the necessary software validation the NEC IECUBE and the Minicube2 using the 78K0R/KG3 target board (QB-78K0RKG3-TB) was used. The Startup sample is designed to work with the given peripherals on this target board like a Switch and 2 LEDs. It gives the user also the possibility to expand the Startup sample with more tasks and a larger functionality because all microcontroller pins are laid to connector strips where a lot more peripherals can be connected to. This gives the opportunity to perform quiet a wide band of possible actions with the freeRTOS.

6. User defined hardware settings

The Startup sample has a predefined hardware setting. This setting has to be checked with the users given environment.

6.1 Predefined hardware settings:

- Option and security byte definition
- Using a external high speed clock of 20MHz
- Using the microcontroller far memory mode
- Using TAU (Timer Array Unit) Channel 5 for the RTOS tick generation (interval timer)
- P120/INTP0/EXLVI Pin as interrupt controlled input from the switch
- P76 and P77 as output pin controlling the LEDs
- RTOS tick count of 1ms

6.2 Changing predefined settings

When there is a need to change the whole or parts of the predefined settings use the following guideline to perform these changes.

6.2.1 Option and security byte definition

To use the 78K0R/Kx3 the option and security bytes have to be set. This is done in the *main.c* file in the Startup sample project. The Option bytes are predefined in the following way.

Option Byte address	Option byte value	Option byte function
0x000C0	0x00	Disabled watchdog timer
0x000C1	0xFE	Low voltage indication enabled
0x000C2	0xFF	Reserved area
0x000C3	0x81	On Chip debugging enabled

Table 2 Option Byte settings

And the security bytes are written 0xFFFFFFFFFFFFFFFFFFFF during the evaluation phase.

For further information about the option and security bytes initialization refer to the corresponding chapters in the 78K0R/KG3 user's manual.

6.2.2 Changing the Clock source

When you want to change the clock source from the external clock to the internal high speed clock you have to change the **configCLOCK_SOURCE** in the *FreeRTOSConfig.h* from 0 to 1.

configCLOCK_SOURCE	Function
0	Use external clock source
1	Use internal high speed clock

Table 3 Clock Source settings

6.2.3 Changing the Clock settings (using the internal high speed clock)

If the internal high speed clock is selected as shown above there are a few settings set can be changed. The clock initialization can be found in the *main.c* **__low_level_init** function.
First you can activate or stop the subsystem clock. When the subsystem clock is enabled a 32.768 kHz resonator has to be connected to XT1 and XT2.

XTSTOP	Function
0	Subsystem clock enabled
1	Subsystem clock disabled

Table 4 Subsystem Clock settings

The prescaler for the internal high speed clock can be changed to decrease the clock speed by editing the value of the CKC register. For further information about system clock prescaling refer to the 78K0R/KG3 user's manual.

IMPORTANT: When changing the hardware clock settings to a certain frequency you have to adjust the **configCPU_CLOCK_HZ** value in the *FreeRTOSConfig.h* to this value too, because this value is used to calculate the Tick Timer compare value used for the RTOS Tick timer interrupt.

6.2.4 Changing the Clock settings (using the external clock)

The initialization of the external clock source is a bit more complex then the one of the internal high speed clock. This is caused by the fact that first the internal high speed clock is used to bridge over the external clock stabilization time when the system is powered on.
After the stabilization time the external clock is set as system clock and further settings can be made.
The subsystem clock can be activated or disabled the same way as described for the internal high speed clock source settings above. If the external clock shall be decreased with the prescaler edit the CKC register value.

IMPORTANT: When changing the hardware clock settings to a certain frequency you have to adjust the **configCPU_CLOCK_HZ** value in the *FreeRTOSConfig.h* to this value too, because this value is used to calculate the Tick Timer compare value used for the RTOS Tick timer interrupt.

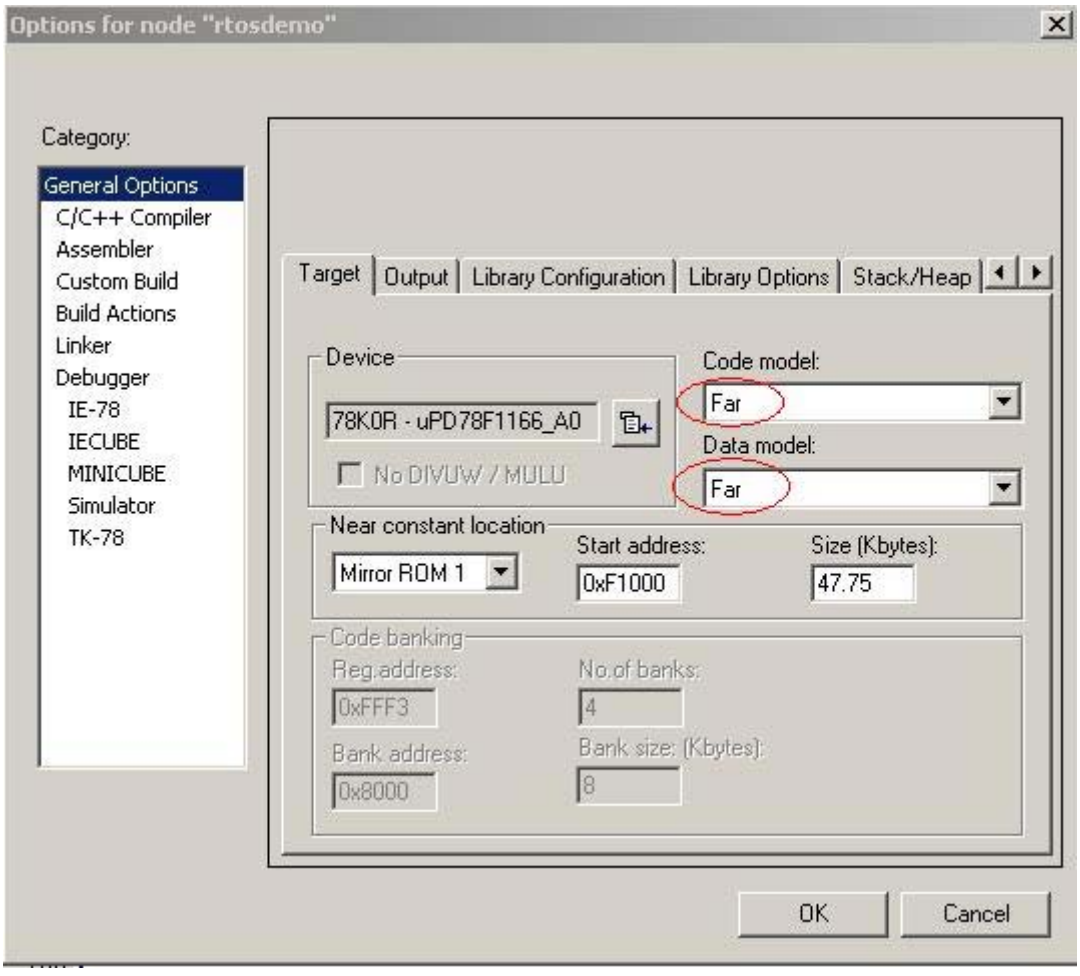
6.2.5 Changing microcontroller memory mode

The 78K0R/KG3 family provides two different memory modes. The first one is the near memory mode in which only 64kbyte code and 64kbyte data can be addressed. Because only 16bit pointers are used for addressing.

In opposition to that the far memory mode allows a addressing over the whole memory space of the microcontroller. This is possible by using a 24bit address pointer.
The function calling convention is different between these two memory modes, what leads to a different task stack layout needed by the freeRTOS.
To change the memory mode the only 2 things, that have to be changed by the user, are the **configMEMORY_MODE** has to be edited in the *FreeRTOSConfig.h* file and the memory mode must be selected in the IAR Embedded Workbench project options (Project → options… → General Options → Code model + Data model).

configMEMORY_MODEL	Function
0	Use near memory model
1	Use far memory model

Table 5 Memory mode settings



The memory mode can also be chosen by using the #pragma directives __far/__near and __far_func/__near_func. A more detailed introduction into using #pragma directives is given in the IAR Embedded Workbench Compiler manual delivered with the corresponding IAR Embedded Workbench IDE.

6.2.6 Changing the TAU channel

In case that the TAU channel 5 is needed by the user, for any reason, every other TAU channel can be used for the Tick creation of the freeRTOS. The tick timer initialization can be found in the **prvSetupTimerInterrupt** function located in the *port.c* file. For further information about the TAU initialization refer to the corresponding section in the 78K0R/KG3 user's manual.
The more complex part of changing the TAU channel is to change the ISR, because this is written in assembler to give the maximum possible performance to the RTOS. The ISR can be found in the *portmacro.s26* file. First you have to find out where to find the position of the new ISR in the interrupt vector table. Therefore you can refer to the corresponding chapter in the 78K0R/KG3 user's manual.
And then write the start address of the ISR to this location.

6.2.7 Changing the Switch Input Pin

In the Startup sample which is designed for the 78K0R/KG3 target board the Pin P120/INTP0/EXLVI is used with the INTP0 functionality, because the switch located on the target board is connected there. If this Pin shall be used in a different mode the initialization must be changed. This initialization can be found in the **prvLEDInit** function in the *LED.c* file.

6.2.8 Changing P76 and P77 Pin functionality

The Pins P76 and P77 are used as output pins in the startup sample because the two LEDs LED01 and LED02 of the target board are connected to this Pins. When another function is needed for these Pins the Initialization can be found in the **prvLEDInit** function in the *LED.c* file.

6.2.9 Changing the tick count frequency

The tick count is an essential part of the RTOS because it controls the task management. To change the tick count frequency the user only has to edit the **configTICK_RATE_HZ** value in the *FreeRTOSConfig.h* file. With the selected clock frequency the compare value for the tick timer interrupt is calculated to generate the interrupt whenever the tick time compare value is reached by the timer counter.

7. Functions of the Startup sample

As mentioned before in this document the Startup sample is written to show the user the functionality of the freeRTOS based on a 78K0R/KG3 target board in an easy way. Based on the possibilities that are given from the target board there will be some toggling LEDs and the implementation of the given switch within an ISR. Furthermore there are some tasks in the background that are used to produce some processor load and can not be seen at the target board itself. All task and queue initializations are performed using the standard freeRTOS functions **xTaskCreate** and **xQueueCreate**. To get further information about the freeRTOS standard functions refer to the freeRTOS homepage.
A further description of the tasks and their functionalities follows now.

7.1 main.c

The *main.c* file includes the task creation for all installed tasks for the freeRTOS and the RTOS scheduler is started from here by calling the **vTaskStartScheduler**. Furthermore the **vErrorChecks** task is implemented in the *main.c*.

7.1.1 vErrorChecks

The only function of this task is to check if all other tasks are still running in the expected behavior. To do this, the task is reentered every 3 seconds and then check if the task control

parameters that are written in the corresponding tasks, have the value expected. If and error occurs the error flag is set, but there is no further control sequence behind this flag that maybe sets and LED or a port pin. So this flag can only be checked with a given debugger in the Startup sample.

7.1.2 __low level init

This function implements the system clock initialization. How to change the clock behavior was already described in this document.

7.2 Int78K0R.c

This file is a customized version of the standard *integer.c* sample file given by the freeRTOS.org. It differs only in the allocated stack size, which can be reduced because there is no need of such much stack space which is given for each **IntMath** task in the standard version and the array size allocated in the functions **vCompeteingIntMathTask3** and **vCompeteingIntMathTask4** for the tasks IntMath3, IntMath4, IntMath7 and IntMath8.
The tasks only perform some mathematical operation to create some processor load and use some processing time on a low task priority. If a smaller device with less memory is chosen or some other tasks shall be added this is a good position to save data memory space by deleting the last 4 tasks (IntMath5 to IntMath8), because they have exactly the same behavior as the first 4 tasks (for example IntMath1 has the same functionality as IntMath5).

7.3 PollQ.c

The *PollQ.c* is also a freeRTOS.org sample file. It shows the functionality of queues. Therefore a producer and a consumer task are created as well as the needed queue. To create the queue the **xQueueCreate** function is used.

7.3.1 vPolledQueueProducer

The producer task writes a value to the queue whenever it is enabled. After writing this value it is incremented. If a problem occurs when trying to write the value to the queue an error flag is set and can be found by the **vErrorChecks** task in the *main.c* file, which uses the given **xArePollingQueuesStillRunning** function.

7.3.2 vPolledQueueConsumer

This task is the counter part to the producer task and reads the values out of the queue. If the read value is not the expected one, an error occurs and will also be detected by the **vErrorChecks** task using the **xArePollingQueuesStillRunning** function. The expected value is just incremented every time the task is entered. So this control sequence is very simple in this startup sample and only works because the producer and consumer task have the same predefined yield time.

7.4 semtest.c

Here two sets of tasks are created and each set of tasks share one semaphore protected variable. So when a task is entered it first tries to obtain the semaphore to be able to change this variable. When the task gets the semaphore the value of this shared variable is checked if it holds the expected value. If this is given the value is cleared to 0 and afterwards incremented by 1 every program cycle until the value reaches the pre-cleared value again. After this value is reached the tasks releases the semaphore again so that the other task of this task set can perform exactly the same thing.

In the first set of tasks the semaphore is polled and in the second set the semaphore is blocked by the corresponding block calls.
An error is flagged if at any time during the process a shared variable is found to have a value other than that expected. Such an occurrence would suggest an error in the mutual exclusion mechanism by which access to the variable is restricted. This error flag will be checked by the **xAreSemaphoreTasksStillRunning** which is called within the **vErrorChecks** task.

7.5 LED.c

The *LED.c* file includes the target board specific task to toggle the both given LEDs of the 78K0R/KG3 target board. Also the initialization of the needed peripherals (LEDs and switch) is implemented in this file. A queue to send the push counter of the switch is implemented.

7.5.1 prvLEDInit

Function to initialize the microcontroller pins connected to the LED01 and LED02 of the target board as output ports and the pin connected to SW1 as input with internal pull-up resistor.

7.5.2 vLEDToggleTask1

This task switches the LED01 of the 78K0R/KG3 target board LED01 every second and checking if this toggle has been executed.

7.5.3 vLEDToggleTask2

In this task the LED02 is toggled whenever the SW1 is pushed and an interrupt caused by this push occurs. To realize this, a switch push count variable **usClick** is incremented every time the switch is pushed within the ISR and is sent to a queue from there. It is read out by the **vLEDToggleTask2** task and whenever the new value of this counter differs from the old one in the task the LED is switched.

8. Stack Coverage within the IAR Embedded Workbench

When using the FreeRTOS within the IAR Embedded Workbench for NEC there will be some messages during debugging that the Stack is out of range. This leads from the fact that every task has its own stack in a special memory area allocated during the task initialization. In future there will be a C-SPY Plugin available which manages this Stack changing too. Until this time it is advantageous to disable the stack coverage in the IAR Embedded Workbench.

9. Summary

This described Startup gives a first overview about how the freeRTOS works on an NEC 78K0R/KG3 device and therefore with the whole 78K0R/Kx3 microcontroller family. By just using easy sample tasks it should be easy for the user to get into the topic and learn a bit more about using freeRTOS. With using the 78K0R/KG3 target board a wide range of expansions of the sample is possible and can be performed by the user.