University
of Victoria

*Department of Electrical and
Computer Engineering*

CENG 455
REAL TIME COMPUTER SYSTEMS
DESIGN PROJECTS

Introduction and Project 1

Acknowledgement

The collection of 455 Real Time Computer Systems Design Projects is the result of the hard work of numerous individuals. Without their effort, time, and ideas over the years, it would not have been possible to provide our students the best possible learning experience. The Department of Electrical and Computer Engineering would like to express its gratitude to these people who have contributed to the design projects in a major way: N. Agarwal, S. Campbell, N.J. Dimopoulos, J. Dorocicz, K. Jones, T. Kroeker, E. Laxdal, K.F. Li, L. Palmer, N. Rebenich, T. Torres-Bonet, B. Zakeri, D. Zhao, H. El Miligi, T. Antoniou and S. Khoshbakht. In addition, the Department acknowledges the generous financial support from the Wighton Engineering Product Development Fund and the Curriculum Development Fund from the UVic Design Engineering Office, hardware and software donations from Freescale Semiconductor, and MQX support from Embedded Access Inc.

1. Introduction

The CENG 455 Laboratory component consists of one simple introduction project and two medium-sized design projects. The purpose of the projects is to introduce you to designing and implementing hardware and software for applications in a real time, multitasking environment.

1.1 Marking

The mark distribution for the laboratory component is:

Project 1,	5%
Project 2,	45%
Project 3,	50%

Total 100%	

1.2 Time Table

The timeline and milestones for the projects are as follows

First scheduled lab. Project 1 must be demonstrated by 6:30 PM; no late Project 1 will be accepted. No report is required for Project 1. Project 2 will be assigned.

Fourth scheduled lab. Project 2 must be demonstrated by 6:30 PM; any Project 2 demonstrated after this time will be subjected to an initial penalty of -1 mark, and an additional -1 mark for every 24 hours. Project 3 will be announced.

72 hours after the Fourth scheduled lab. The last day for demonstrating a late Project 2. After 6:30 PM, no demonstrations of Project 2 will be accepted, and a failing grade for the COURSE will be given.

One week after Fourth scheduled lab. Reports for Project 2 must be handed in by 5:30 PM. Late reports will be penalized the same way as late demonstration.

72 hours after above deadline. The last day for submitting reports for Project 2.

Eighth scheduled lab. ‘Demo days’ for Project 3. (Late penalties as Project 2).

One week after the Demo. Reports for Project 3 due. (Late penalties as Project 2).

1.3 Notes

- All three projects must be demonstrated to the lab instructor, and for Project 3, to the course instructor as well, in order to have a mark assigned. A project report without a demonstration will result in a 0 mark, and thus a **failing grade for the COURSE**. An acceptable demonstration is one that presents to the lab/course instructors how much of your project works according to specification, and shows the project source code. You can demonstrate the project even if it is not working completely or correctly. Be prepared to answer questions.

- The stated deadlines are firm and will not be extended.

- The projects must be performed in groups of no more than three.

- The lab instructor will in general not be available outside of the lab session hours. If you want to see the lab instructor outside of the lab session you must arrange it via e-mail. The lab instructor will not be available on a drop-in basis.

2. Overview

This section gives a general overview of the hardware and software development platform used in the lab.

2.1 The Development Environment

No setup of the development environment should be required. The PCs in the lab run Windows. You login to Windows using your UVic Netlink username and password. The home directory of your Engineering account will be mounted automatically as drive M. It is a good practice to save your work to your M drive regularly. Freescale's Kinetis Design Studio is available on your workstation.

2.2 Hardware

The hardware platform is an NXP/Freescale FRDM-K64F board. Information on this board can be found on the course web site (www.ece.uvic.ca/~ceng455).

2.3 Software

The software development platform Kinetis Design Studio is used to build, run, and debug your MQX applications. Relevant manuals and references can be found on the course lab support web pages.

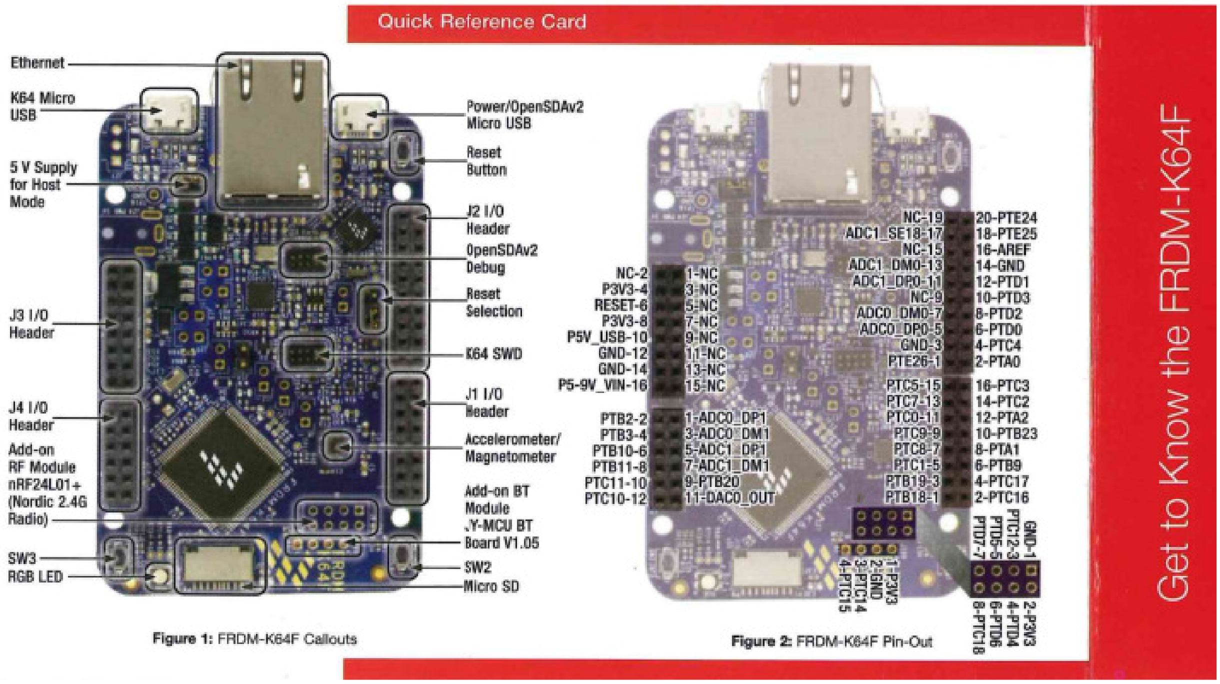


Figure 1: **FRDM-K64F board** (image courtesy of NXP/Freescale)
(Connect USB to OpenSDAv2 micro-USB connector at top-right for programming)

Project 1

Introduction to MQX

The purpose of this project is to introduce you to MQX and its related software and hardware.

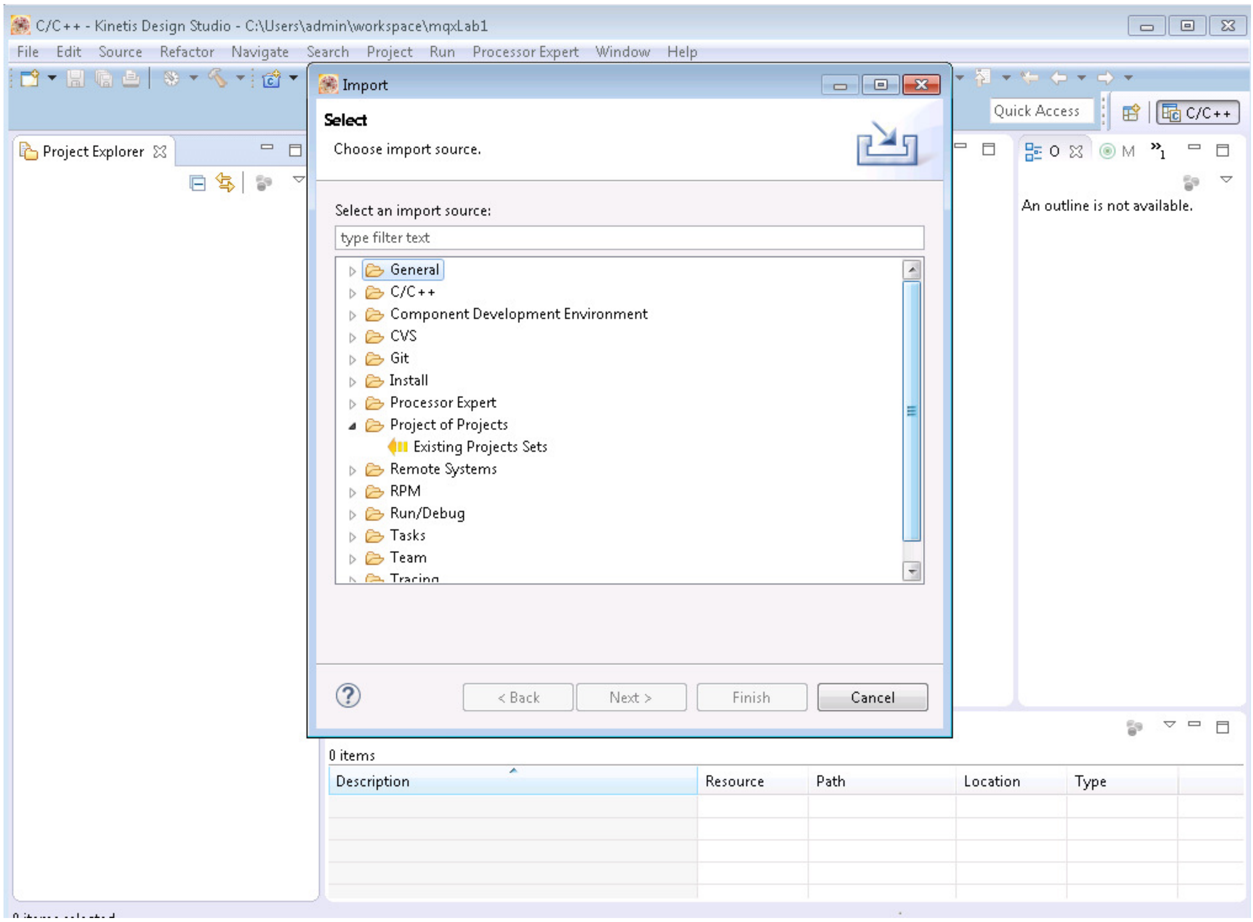
P1.1 Marking

This project is worth 5% of your Lab mark. This mark will be given if you demonstrate a successful implementation of the project before the end of the first lab session. There is no write up required for this project.

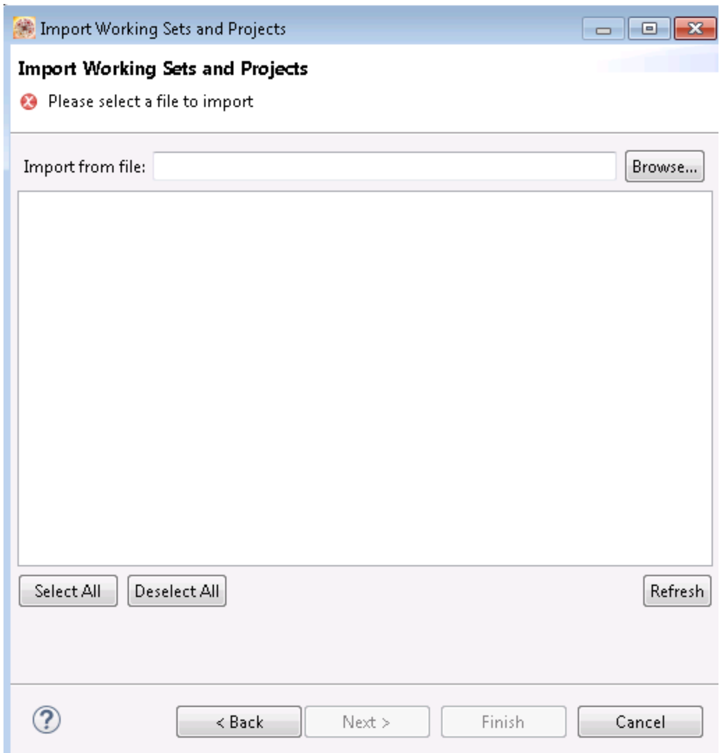
P1.2 Introduction to MQX and KDS by importing an example program

1. Use the shortcut on your desktop to run the development environment “Kinetis Design Studio”. When the “Select a Workspace” window appears, choose “C:\Users\NETLINK\workspace\mqxLab1” where NETLINK = your username, then click “OK”

2. Using the “File”, “Import”... menu in this KDS IDE, click “Project of Projects”, “Existing Projects Sets”, Next.



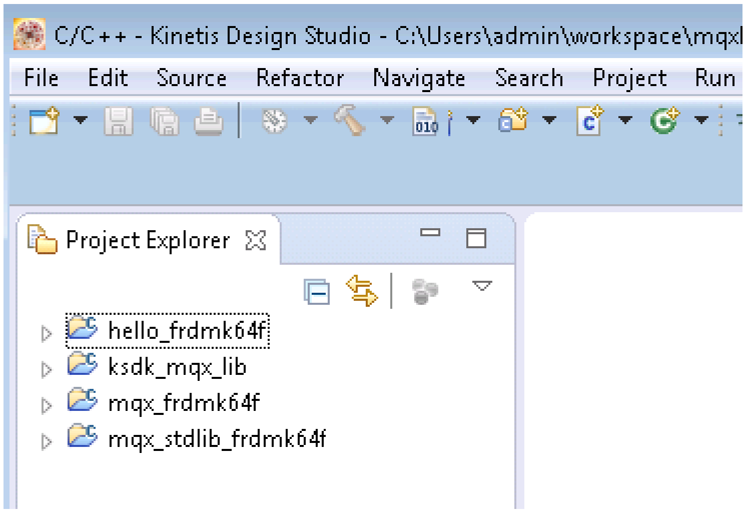
3. Click “Next” and in “Importing Working Sets and Projects” beside “Import from File”, click “Browse”.



Please select a file to import

4. Open the file called “hello_frdmk64f.wsd” at:
“C:\Freescale\KSDK_1.3.0\rtos\mqx\mqx\examples\hello\build\kds\hello_frdmk64f\”
by moving to this folder, clicking “Open” then “Finish”.

5. The Project Explorer window of KDS will now look like this:



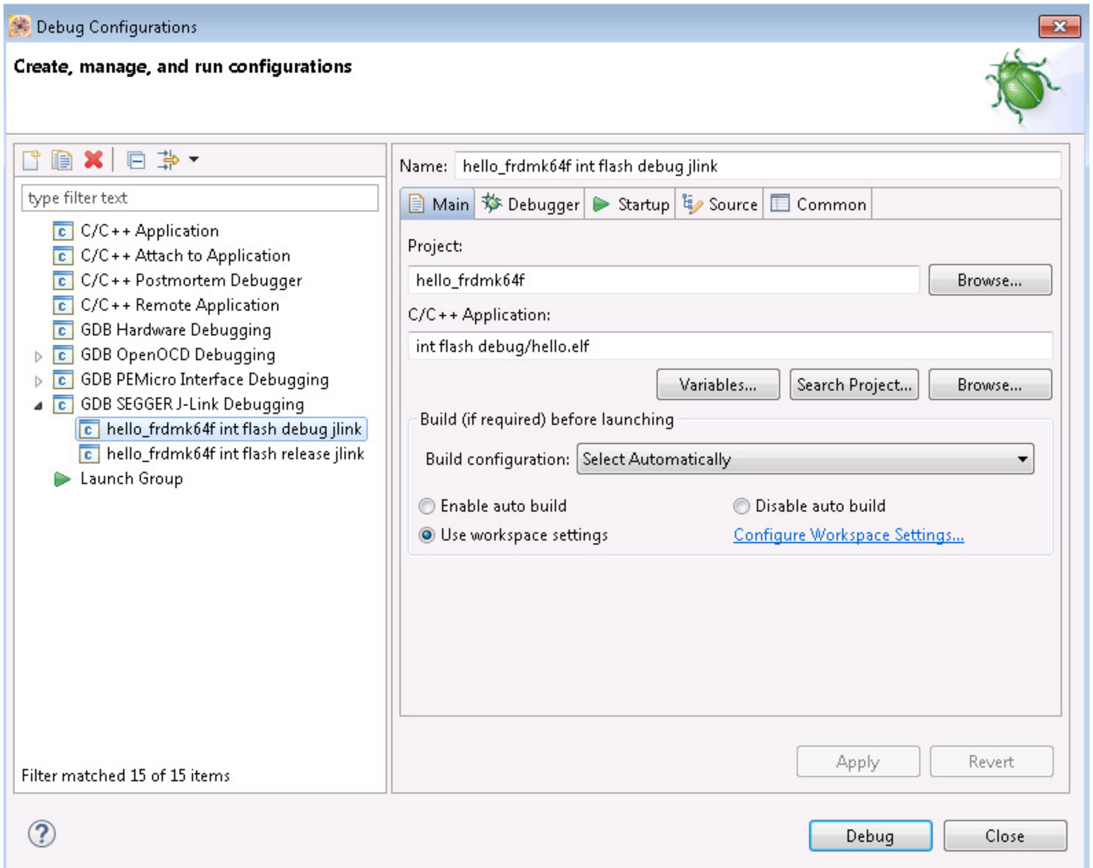
First time through the process, you'll need to build the 3 libraries under the "hello_frdmk64f" folder, then the "hello" folder itself. Simply click on each library (starting with the bottom one) and click the little "hammer" icon above the Project Explorer window. Once you've got one demo built (in this case the "hello" demo), you can import other demos and usually just compile the demo itself (much faster, as the libraries are already done).

You can keep track of what the compiler is doing by clicking on the "Console" tab at the bottom of the KDS screen. Once everything is done, you should end up with a "hello.elf" file showing in the console tab, which can be downloaded to the board in step 6.

One caveat to this…. If things ever get really messed up by a bunch of projects in this workspace, you can simply, click "File", "Switch Workspace" and set up a clean environment with just that last imported demo (or project) and continue working.

Finally, one really important note….. although building software is much faster on drive C, the local drive, it is important to copy your workspace folder to drive M at the end of your session, so that you can drop into the lab and use any workstation (plus your files get backed up regularly in case the PC disk has a problem).

6. On the Menu Bar, click "Run", "Debug Configurations" and choose "GDB SEGGER J-Link Debugging", "hello_frdmk64f int flash debug jlink". Then click "Debug" as shown.


Debug Configurations
Create, manage, and run configurations
type filter text
C/C++ Application
C/C++ Attach to Application
C/C++ Postmortem Debugger
C/C++ Remote Application
GDB Hardware Debugging
GDB OpenOCD Debugging
GDB PEMicro Interface Debugging
GDB SEGGER J-Link Debugging
hello_frdmk64f int flash debug jlink
hello_frdmk64f int flash release jlink
Launch Group
Main
Debugger
Startup
Source
Common
Browse...
Variables...
Search Project...
Browse...
Configure Workspace Settings...
Apply
Revert
Debug
Close

Name: hello_frdmk64f int flash debug jlink

Project: hello_frdmk64f

C/C++ Application: int flash debug/hello.elf

Build (if required) before launching

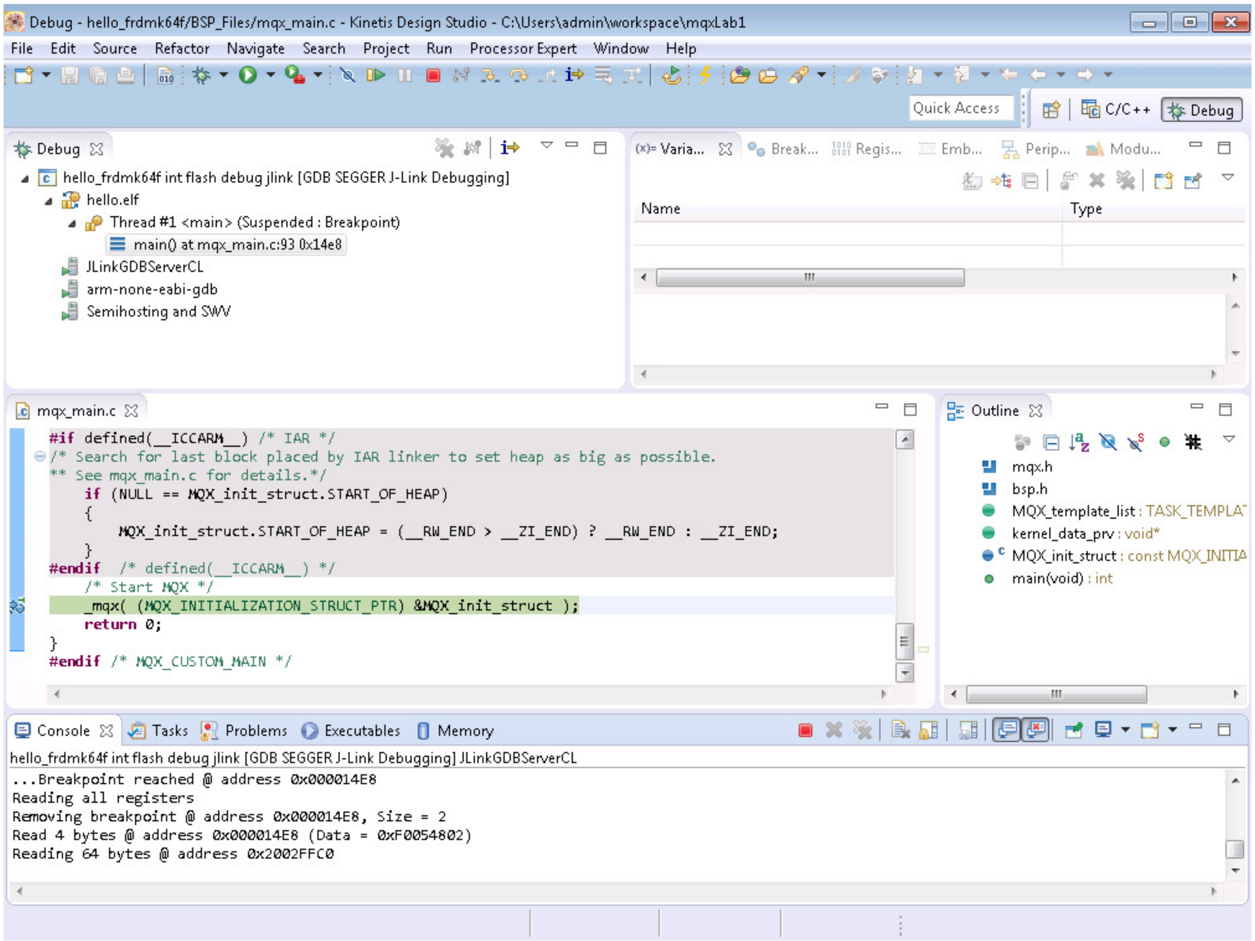
Build configuration: Select Automatically

○ Enable auto build ○ Disable auto build

● Use workspace settings

Filter matched 15 of 15 items

After pressing “Debug”, you’ll see a message to accept terms of use since this company sells programmers for production use and non-FRDM boards. Then a message about SWO, where you choose “OK”, then an “open perspective now” message and a “remember my decision” option. All this is required to first to download the code and open the debugger perspective shown below.



Debug - hello_frdmk64f/BSP_Files/mqx_main.c - Kinetis Design Studio - C:\Users\admin\workspace\mqxLab1

hello_frdmk64f int flash debug jlink [GDB SEGGER J-Link Debugging]
hello.elf
Thread #1 <main> (Suspended : Breakpoint)
main() at mqx_main.c:93 0x14e8
JLinkGDBServerCL
arm-none-eabi-gdb
Semihosting and SWV

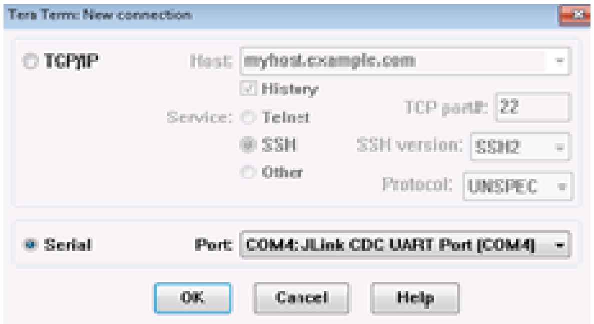
```
#if defined(__ICCARM__) /* IAR */
/* Search for last block placed by IAR linker to set heap as big as possible.
** See mqx_main.c for details.*/
    if (NULL == MQX_init_struct.START_OF_HEAP)
    {
        MQX_init_struct.START_OF_HEAP = (__RW_END > __ZI_END) ? __RW_END : __ZI_END;
    }
#endif  /* defined(__ICCARM__) */
    /* Start MQX */
    _mqx( (MQX_INITIALIZATION_STRUCT_PTR) &MQX_init_struct );
    return 0;
}
#endif /* MQX_CUSTOM_MAIN */
```

hello_frdmk64f int flash debug jlink [GDB SEGGER J-Link Debugging] JLinkGDBServerCL

```
...Breakpoint reached @ address 0x000014E8
Reading all registers
Removing breakpoint @ address 0x000014E8, Size = 2
Read 4 bytes @ address 0x000014E8 (Data = 0xF0054802)
Reading 64 bytes @ address 0x2002FFC0
```

At this point, the USB is now emulating a serial port to display output, so we need to set up a program to display the results before proceeding.

7. Using TeraTerm as a serial monitor. Minimize KDS for a moment, click “TeraTerm” on your desktop. The following window will open. Choose “serial” and the port that shows “JLink” when the FRDM-K64F board is connected, then “OK”.



Tera Term: New connection

○ TCP/IP Host: myhost.example.com

☑ History

Service: ○ Telnet

● SSH

○ Other

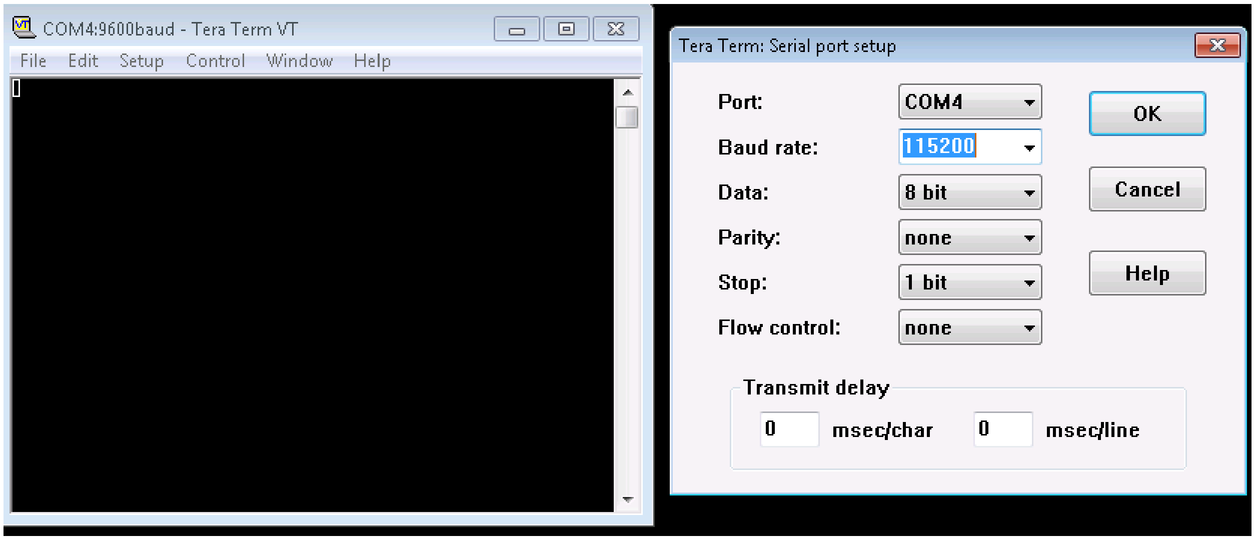
TCP port#: 22

SSH version: SSH2

Protocol: UNSPEC

● Serial Port: COM4: JLink CDC UART Port (COM4)

Then choose “Setup”, “Serial” and “115200 baud”, “OK”



COM4:9600baud - Tera Term VT

Tera Term: Serial port setup

Port: COM4

Baud rate: 115200

Data: 8 bit

Parity: none

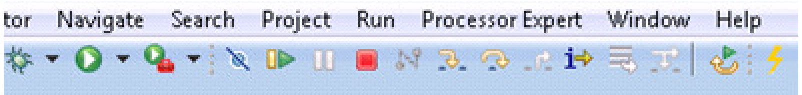
Stop: 1 bit

Flow control: none

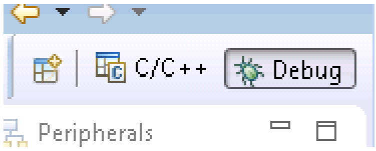
Transmit delay

0 msec/char 0 msec/line

8. Maximize KDS and use the green triangle under Project for “resume” (labels are seen when moving mouse over icons). The message “Hello world” should appear on the TeraTerm terminal. Notice that the program can be paused, restarted, single stepped, etc. When you are done, press the red “stop” button before switching perspective in the top-right corner back to “C/C++” from “Debug”. If you don’t stop the debug session, it will not be possible to start the next debug session without an error (since this debug session would still be running.



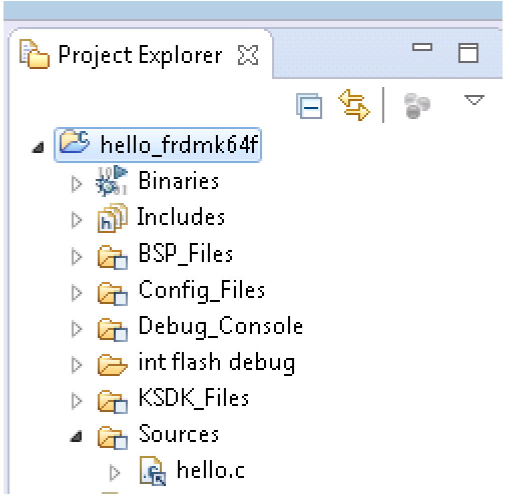
Controls for Debugging



Modes: C/C++ or Debug

9. Next steps. Source code, New Program Options.

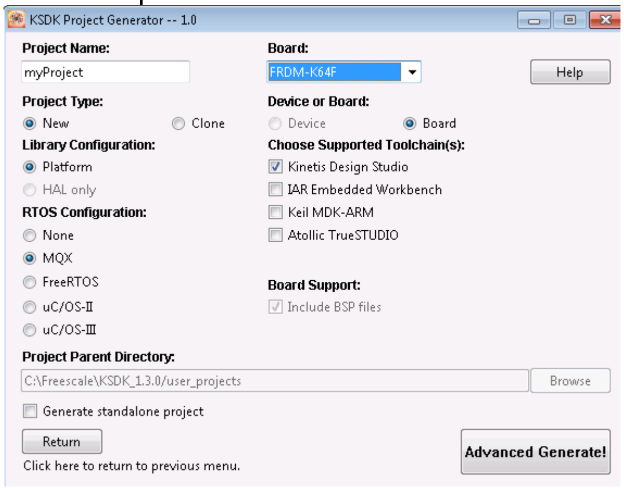
To view the C code from your imported project, expand the “Sources” folder as shown and click on “hello.c”.



There will be more information on the source code in the next section, but for now, have a look at “File”, “New”. It is apparent that there is no “New MQX Project” option, like in older Codewarrior or other environments. This may seem to be a concern at first, but there are other options available:

A. You’ve seen here that you can import an existing project. You’ll see in a moment how this project can easily be developed further.

B. You can also minimize KDS for a moment, and click Start, Computer, c:\Freescale, and see “Freescale Project Generator.zip”. Copy this ZIP file to your Desktop, right-click to “Extract All”, and open the folder that appears. Choose the Windows subfolder and then click on the “KSDK_Project_Generator.exe” file, and choose the “Advanced” menu. This program lets you easily make an MQX project that you can import or open in KDS and build upon.



Project Name: myProject

Board: FRDM-K64F

Project Type: ● New ○ Clone

Device or Board: ○ Device ● Board

Library Configuration: ● Platform ○ HAL only

Choose Supported Toolchain(s): ☑ Kinetis Design Studio ☐ IAR Embedded Workbench ☐ Keil MDK-ARM ☐ Atollic TrueSTUDIO

RTOS Configuration: ○ None ● MQX ○ FreeRTOS ○ uC/OS-II ○ uC/OS-III

Board Support: ☑ Include BSP files

Project Parent Directory: C:\Freescale\KSDK_1.3.0/user_projects

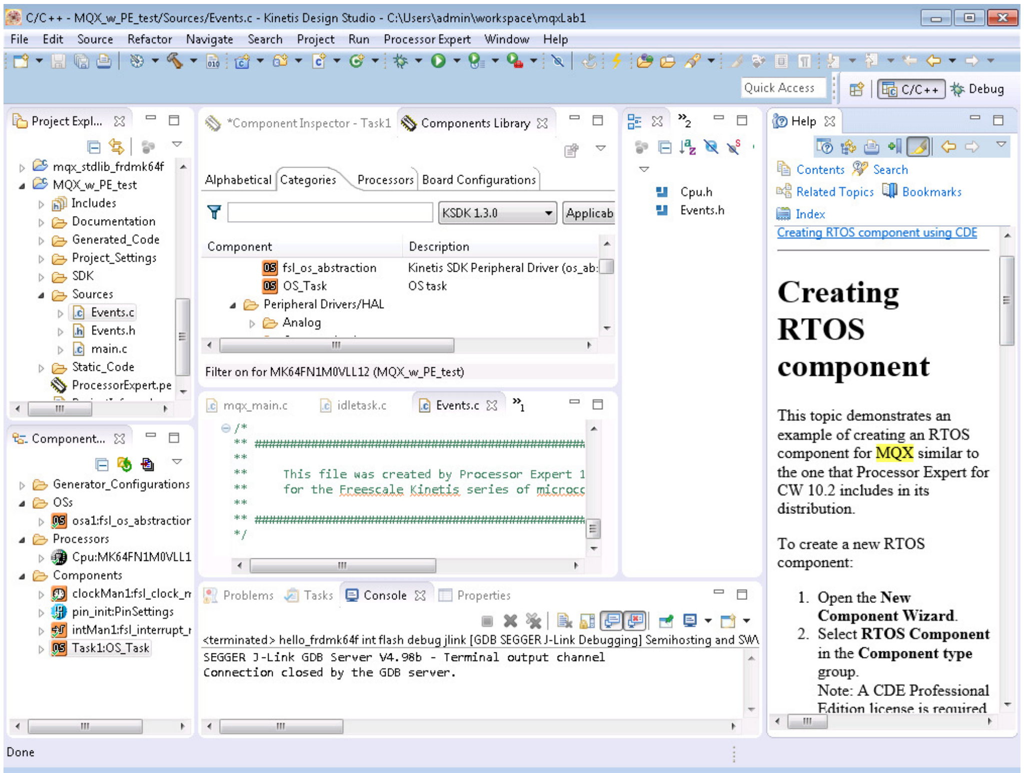
☐ Generate standalone project

Click here to return to previous menu.

One important point to note is that although the “import” method is straightforward and works well, it also contains a weakness. In the current version of KDS, the “build” is by default location specific, and it is possible to over-write the example.   If the folder is write-protected, the example doesn’t build without a lot of adjustments, so it is not straightforward for multi-user operation. Best thing is to try the examples, but leave them in tact for the next person. In fact, they will be over-written regularly to assure the examples are available.

The preferred method seems to be using the Project Generator, and specifying the “standalone option” so that you can put the project and libraries all within your “c:\Users\Netlink\ceng455” folder. The initial build is a case of “import c:\Users\Netlink\ceng455\kdsTestLab1\kds\kdsTestLab1.wsd” for example), building the libraries then the application (just like with the examples). And backups to drive M would be everything under the “ceng455” folder. When you move to a new PC, make a new workspace if needed and drag the folder across to its “c:\users\Netlink\ceng455” location.

C. You can choose “New”, “Processor Expert Project” and specify within this code generator that you want an MQX Project, and specify any hardware settings you want for your project. Becoming good at using this interface means you can set up complex hardware options quickly on different hardware targets, but the learning curve and details are considerable at first and may not be worth the effort for small or medium complexity “software only” projects.



```
/*
** ###################################################################
**
**     This file was created by Processor Expert 1
**     for the Freescale Kinetis series of microcc
**
** ###################################################################
*/
```

```
<terminated> hello_frdmk64f int flash debug jlink [GDB SEGGER J-Link Debugging] Semihosting and SW\
SEGGER J-Link GDB Server V4.98b - Terminal output channel
Connection closed by the GDB server.
```

Creating RTOS component using CDE

Creating RTOS component

This topic demonstrates an example of creating an RTOS component for MQX similar to the one that Processor Expert for CW 10.2 includes in its distribution.

To create a new RTOS component:

1. Open the **New Component Wizard**.
2. Select **RTOS Component** in the **Component type** group.
   Note: A CDE Professional Edition license is required

In any of these situations, using MQX or other real-time software can mean that most of the code is reusable and transportable to other processor targets. The code for each task can be very concise and easy to maintain, so further development later can be much simpler than having one big main loop with lots of function calls and in-line code going on.

P1.3 MQX Task Creation

Maximize KDS once again, and have a look at the “hello.c” source file.

Have a look at the task template. There is “world” which is an auto_start_task, and “hello” which is created by world.

```
//----------------------------------------------------------------------------------
const TASK_TEMPLATE_STRUCT   MQX_template_list[] =
{
    /* Task Index, Function, Stack, Priority, Name, Attributes, Param, Time Slice */
      { WORLD_TASK, world_task,700, 9, "world",   MQX_AUTO_START_TASK, 0, 0 },
      { HELLO_TASK, hello_task,700,8,"hello", 0, 0, 0 },
      { 0 }
};
//----------------------------------------------------------------------------------
```

Experiment with changing the priority of the “hello” task from “8” to “9”. Compile and run the program again to see what happens. What will be the result if the “hello” task’s priority is increased or decreased?

P1.4 Task Communication

Now go back to the examples, and open up the example called “msg”.   This example shows an approach to pass messages from clients to a server process and display text on the TeraTerm terminal. The intent of this is to merely get used to looking at the examples, and a brief overview of how message queues and pools are created and used.

At this point, have a look at both the MQX Users Guide and the MQX Reference Manual to see how to find out more information on various functions, such as creating the message queues. Knowing where to look later will save time throughout the term.

Finally, after looking at the message example, see if you can modify the “hello” example to have one task send a string to the other task, which prints it out. Show this to your TA. No writeup is required for this lab.   And please remember, it is a good practice to save your work to your M drive regularly.

University of Victoria

Department of Electrical and Computer Engineering

CENG 455
REAL TIME COMPUTER SYSTEMS
DESIGN PROJECTS

Hardware Configuration Tutorial and Project 2

Acknowledgement

The collection of 455 Real Time Computer Systems Design Projects is the result of the hard work of numerous individuals. Without their effort, time, and ideas over the years, it would not have been possible to provide our students the best possible learning experience. The Department of Electrical and Computer Engineering would like to express its gratitude to these people who have contributed to the design projects in a major way: N. Agarwal, S. Campbell, N.J. Dimopoulos, J. Dorocicz, K. Jones, T. Kroeker, E. Laxdal, K.F. Li, L. Palmer, N. Rebenich, T. Torres-Bonet, B. Zakeri, D. Zhao, H. El Miligi, T. Antoniou and S. Khoshbakht. In addition, the Department acknowledges the generous financial support from the Wighton Engineering Product Development Fund and the Curriculum Development Fund from the UVic Design Engineering Office, hardware and software donations from Freescale Semiconductor**,** and MQX support from Embedded Access Inc.

Part 1. Hardware Configuration Tutorial

Setting up MQX and UART in Processor Expert

By: Saman Khoshbakht (January 2016)

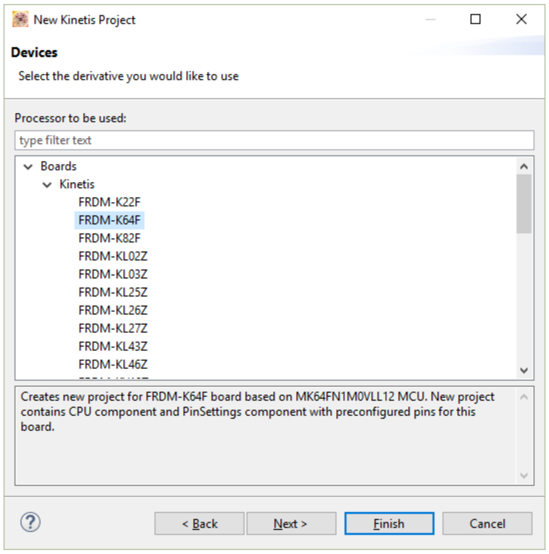
In this tutorial we create a new MQX Project in KDS, using Processor Expert tool. In addition, we will see how to change from MQX_Lite to MQX_Standard settings, create tasks, setup peripherals such as UART. We will write a short code to create a loopback for UART 3. This will echo back every character the user types in the console connected to UART 3. The following steps will guide you through this process:

Create a new workspace

Go to File > New > Processor Expert Project

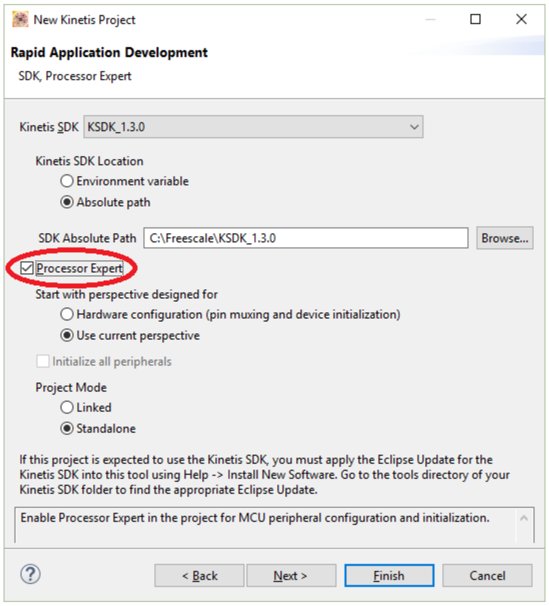
Choose the name of your project (e.g. ***"serial_echo"***), click "**Next**"

From the device selection menu, choose Board > Kinetis > **FRDM-K64F**, click "**Next**"



Creates new project for FRDM-K64F board based on MK64FN1M0VLL12 MCU. New project contains CPU component and PinSettings component with preconfigured pins for this board.

In the "Rapid Application Development" window, enable Processor Expert by checking the box as shown. Click "Finish"



New Kinetis Project

Rapid Application Development

SDK, Processor Expert

Kinetis SDK: KSDK_1.3.0

Kinetis SDK Location
- ○ Environment variable
- ● Absolute path

SDK Absolute Path: C:\Freescale\KSDK_1.3.0 [Browse...]

☑ Processor Expert

Start with perspective designed for
- ○ Hardware configuration (pin muxing and device initialization)
- ● Use current perspective

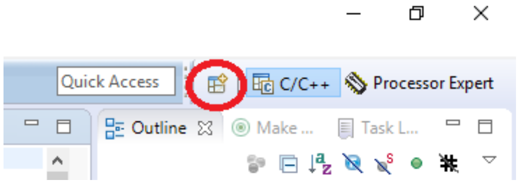
□ Initialize all peripherals

Project Mode
- ○ Linked
- ● Standalone

If this project is expected to use the Kinetis SDK, you must apply the Eclipse Update for the Kinetis SDK into this tool using Help -> Install New Software. Go to the tools directory of your Kinetis SDK folder to find the appropriate Eclipse Update.

Enable Processor Expert in the project for MCU peripheral configuration and initialization.

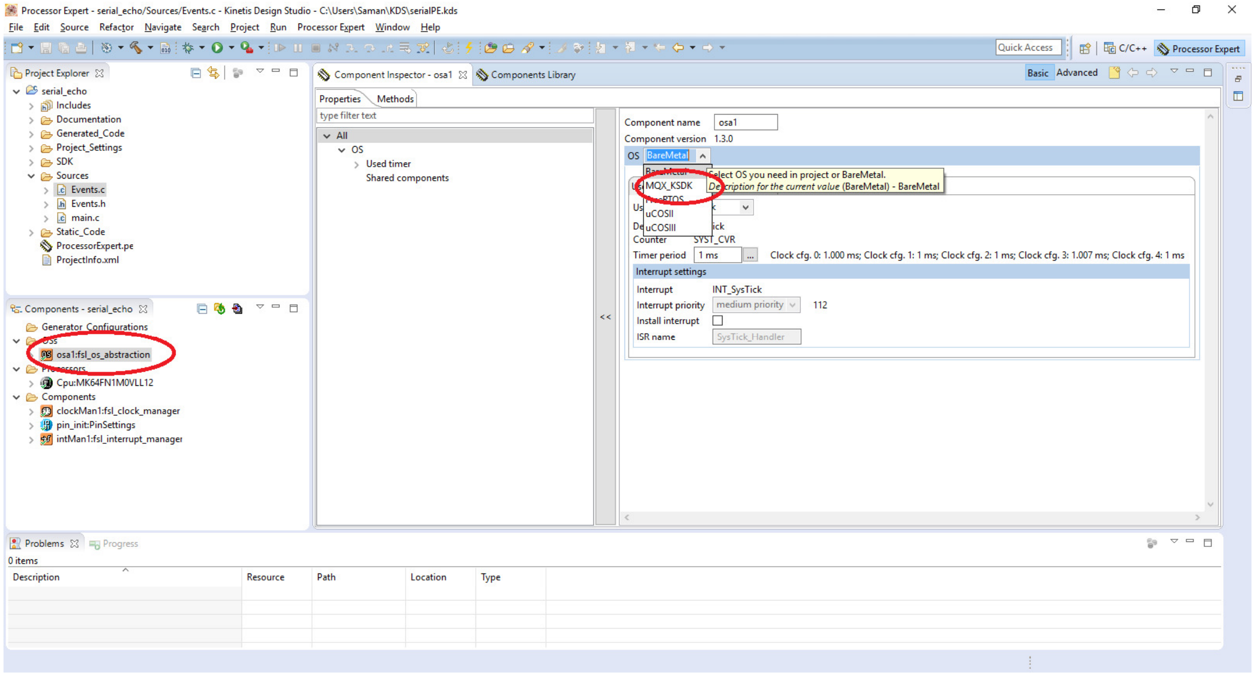
At this point it is easier to create a new perspective for Processor Expert, one separate from your "C/C++" development perspective. This will make it easier to quickly switch between code development and Processor Expert. You can do this by clicking on as shown below and choose "Processor Expert". You can now close both "Component Inspector" and "Components Library" Windows in the **"C/C++"** Perspective.



Setting up MQX Standard

Switch to "Processor Expert" Perspective.

You can keep track of all currently used components in your design (at any time during development) by looking at the "**Components - serial_echo**" window pane at the bottom left corner of KDS, as shown below. The default OS for a new project is "BareMetal". As it is shown in the figure below, open the component inspection window for your current OS Abstraction (i.e. *osa1:fsl_os_abstraction*) by double-clicking on it, then from the OS drop-down menu, choose "**MQX_KSDK**"



Component name osa1
Component version 1.3.0
OS BareMetal

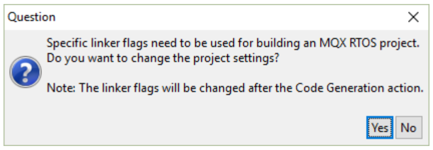
BareMetal
MQX_KSDK
FreeRTOS
uCOSII
uCOSIII

Select OS you need in project or BareMetal.
Description for the current value (BareMetal) - BareMetal

Counter SYST_CVR
Timer period 1 ms ... Clock cfg. 0: 1.000 ms; Clock cfg. 1: 1 ms; Clock cfg. 2: 1 ms; Clock cfg. 3: 1.007 ms; Clock cfg. 4: 1 ms

Interrupt settings
Interrupt INT_SysTick
Interrupt priority medium priority 112
Install interrupt □
ISR name SysTick_Handler

Answer yes to the question window that appears next.

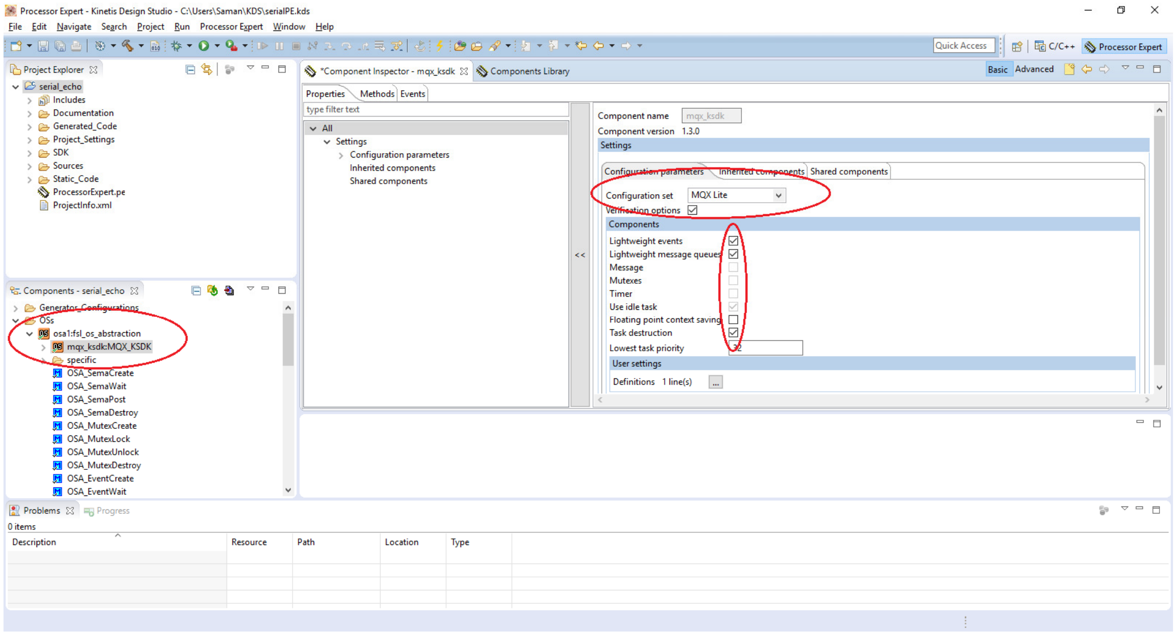


Question

Specific linker flags need to be used for building an MQX RTOS project. Do you want to change the project settings?

Note: The linker flags will be changed after the Code Generation action.

At this point what we have configured is "MQX Lite", which is the default option for PE, but as you can see in the configurations of the current *osa1* component, as shown below, some of the functionalities of MQX cannot be enabled (without manually configuring the generated code).



Component name mqx_ksdk
Component version 1.3.0

Settings

Configuration parameters | Inherited components | Shared components

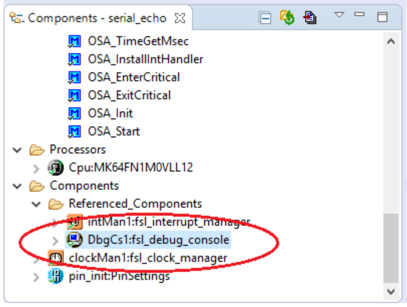
Configuration set MQX Lite
Verification options ☑

Components
Lightweight events ☑
Lightweight message queues ☑
Message □
Mutexes □
Timer □
Use idle task ☑
Floating point context saving □
Task destruction ☑
Lowest task priority 32

User settings
Definitions 1 line(s) ...

Thus, we need to enable MQX Standard by following these steps:

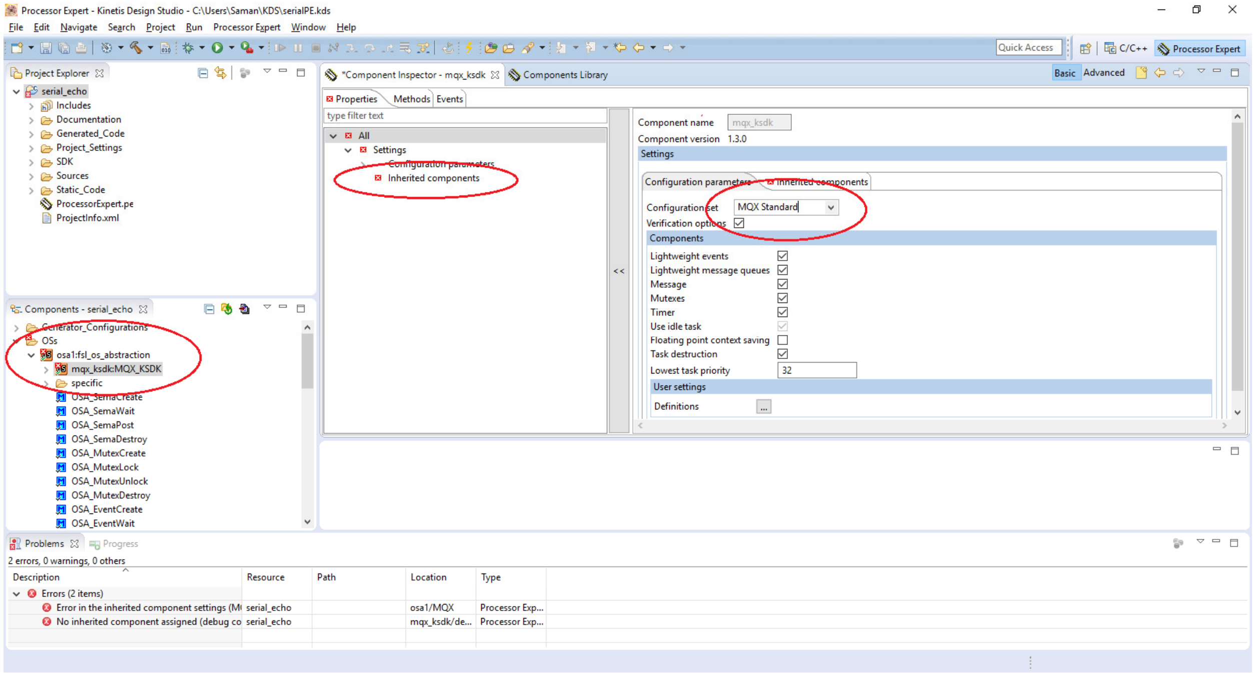
First remove ***DbgCs1*** Component. MQX Standard comes with its own debug console driver, which will conflict with this component later on if we don't do this.



This might add some errors to the project, but it is ok for now.

In *osa1->mqx_ksdk* component, as shown below, select **MQX Standard** for Configuration set. It might take a few seconds for KDS to change the configuration.

Select **Inherited components** configuration from **Properties** tab.



Component name: mqx_ksdk
Component version: 1.3.0

Configuration set: MQX Standard
Verification options: ☑

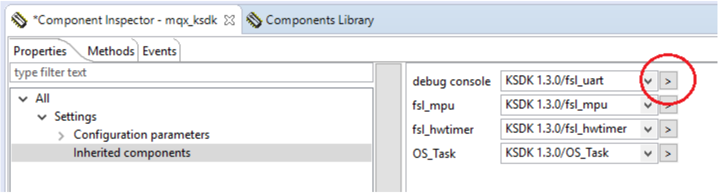
Lightweight events	☑
Lightweight message queues	☑
Message	☑
Mutexes	☑
Timer	☑
Use idle task	☑
Floating point context saving	□
Task destruction	☑
Lowest task priority	32

Definitions ...

2 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (2 items)				
Error in the inherited component settings (M	serial_echo		osa1/MQX	Processor Exp...
No inherited component assigned (debug co	serial_echo		mqx_ksdk/de...	Processor Exp...

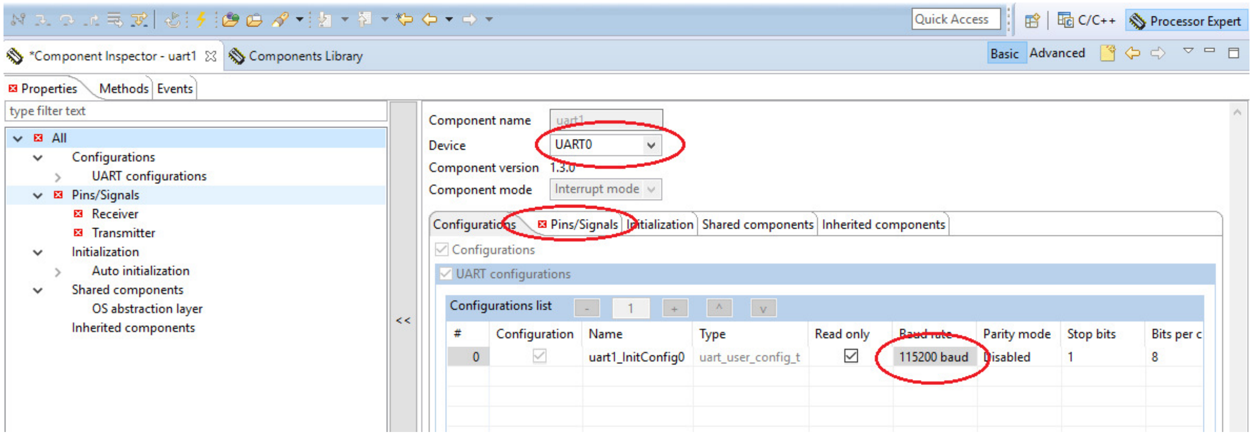
Choose *"KDS 1.3.0/fsl_uart"* as the debug console component, then click on in front of debug console to get to the configurations for this component.



debug console	KSDK 1.3.0/fsl_uart
fsl_mpu	KSDK 1.3.0/fsl_mpu
fsl_hwtimer	KSDK 1.3.0/fsl_hwtimer
OS_Task	KSDK 1.3.0/OS_Task

Here we configure the settings for the debug console, as we want to have access to debug console via the J-Link serial host (which is connected to UART 0 on the microcontroller side) to be able to use *printf* statement for debug purposes. In addition (later on in this document) we will configure **UART3** as our target serial port for the driver project, which can be accessed via on-board connectors.

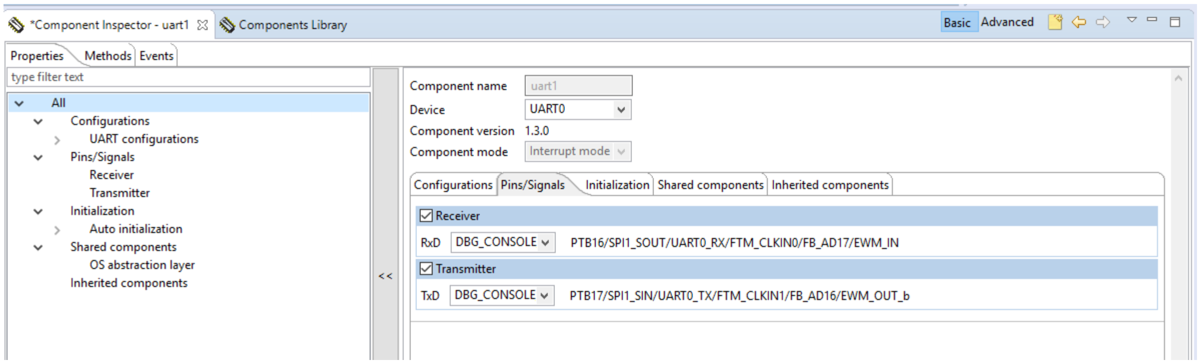
Change the configurations for Device to "**UART0**" and Choose **115200** for Baud rate, then click on the Pins/Signals Configurations.



Component name: uart1
Device: UART0
Component version: 1.3.0
Component mode: Interrupt mode

#	Configuration	Name	Type	Read only	Baud rate	Parity mode	Stop bits	Bits per c
0	☑	uart1_InitConfig0	uart_user_config_t	☑	115200 baud	Disabled	1	8

Here we route the pins for RX and TX. Choose "**DBG_CONSOLE_RX**" for **RxD**, and "**DBG_CONSOLE_TX**" for **TxD**. These pins are routed to the J-Link serial driver.



Component name: uart1
Device: UART0
Component version: 1.3.0
Component mode: Interrupt mode

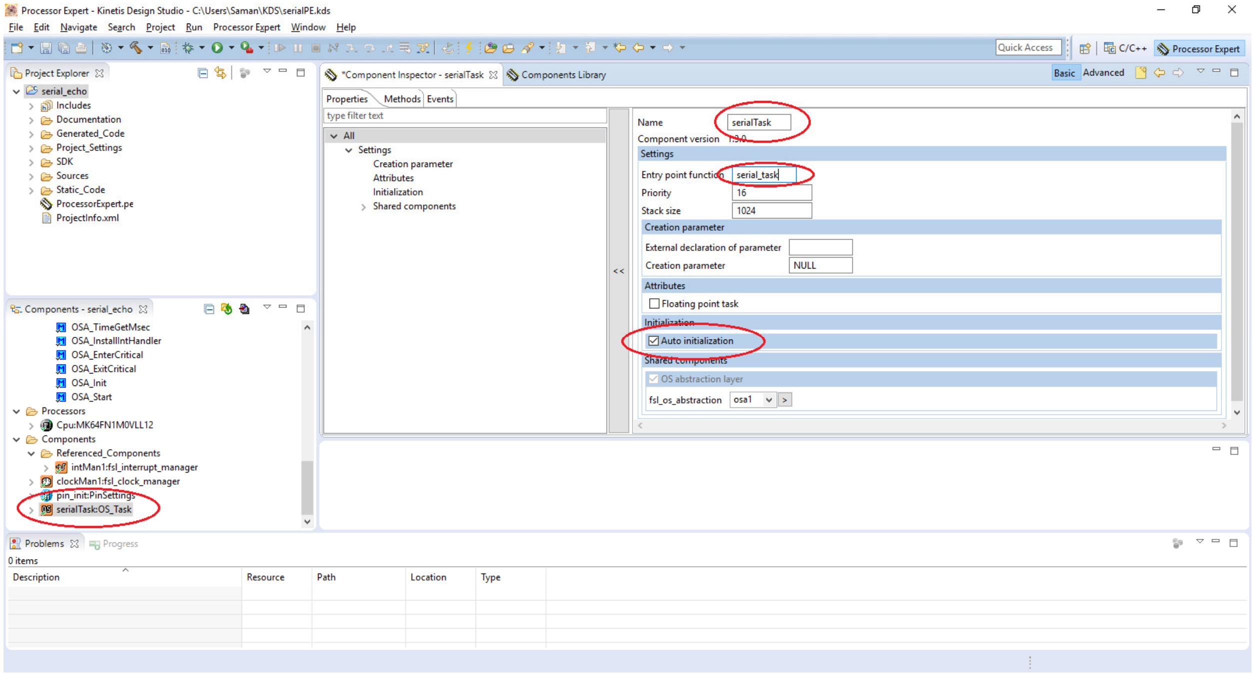
☑ Receiver
RxD: DBG_CONSOLE — PTB16/SPI1_SOUT/UART0_RX/FTM_CLKIN0/FB_AD17/EWM_IN

☑ Transmitter
TxD: DBG_CONSOLE — PTB17/SPI1_SIN/UART0_TX/FTM_CLKIN1/FB_AD16/EWM_OUT_b

We have now finished setting up MQX Standard for the project along with its debug console.

Creating Tasks

Tasks can be created using the *OSA_Task* component. Switch to "Components Library" tab and create an *OS_Task* component by double-clicking on OS OS_Task . This will create a task called "**Task 1**" in your **Component** folder. Bring up the configurations for this task and change the "**Name**" of it to "**SerialTask**" and the "**Entry point function**" to "**serial_task**". If you leave the "**Auto Initialization**" checkbox enabled, this task will be automatically created and started. This is what we want in this example but later on you might want to change this option for other tasks and create them in an on demand fashion. You can hover your pointer over "**Auto Initialization**" to get some useful information about where to find the initialization codes.



Processor Expert - Kinetis Design Studio - C:\Users\Saman\KDS\serialPE.kds

Name: serialTask
Component version: 1.3.0

Settings
Entry point function: serial_task
Priority: 16
Stack size: 1024

Creation parameter
External declaration of parameter:
Creation parameter: NULL

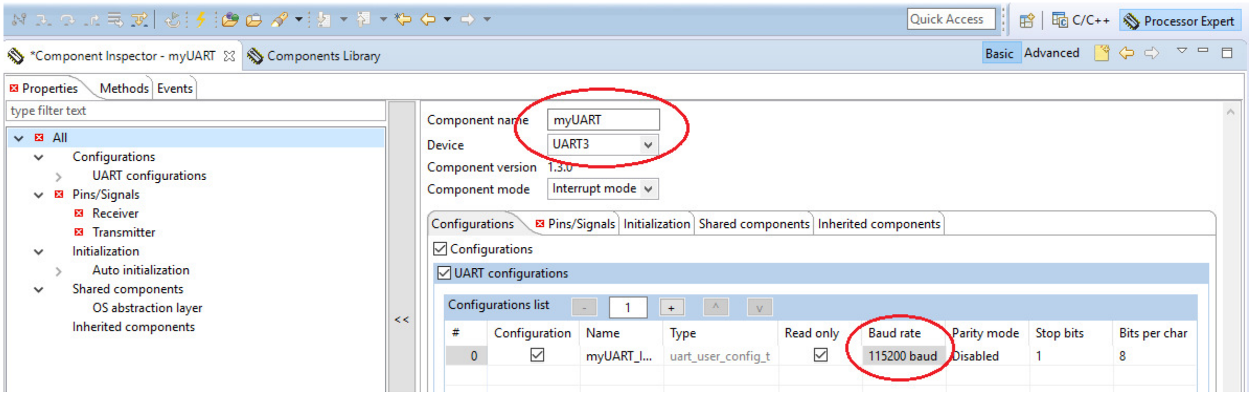
Attributes
☐ Floating point task

Initialization
☑ Auto initialization

Shared components
☑ OS abstraction layer
fsl_os_abstraction: osa1

Creating a *UART* Component for UART3

To create a UART component head to "**Component Library**" again. Create a *fls_uart* component by double-clicking on fsl_uart . This will add a new component called "**uartCom1**" to your components folder. Double-click on this component and bring up the configurations window. Change the "**Name**" to "**myUART**" and "**Device**" to "**UART3**". Choose **115200** for Baud rate, as shown below, then go to "**Pins/Signals**" tab.



Component name: myUART

Device: UART3

Component version: 1.3.0

Component mode: Interrupt mode

Configurations | Pins/Signals | Initialization | Shared components | Inherited components

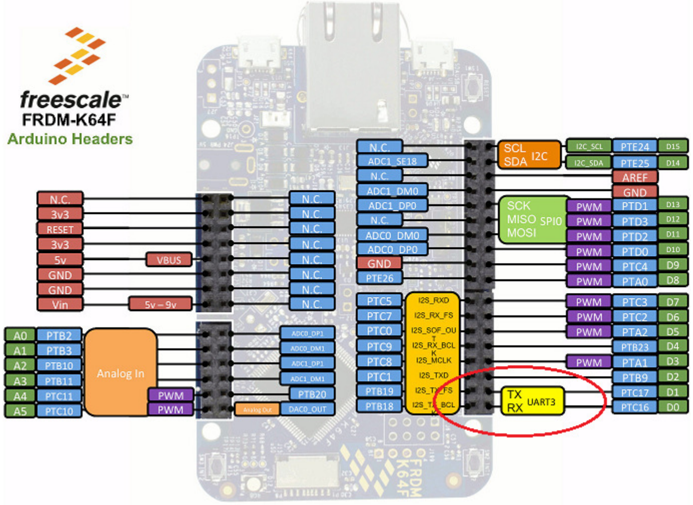
☑ Configurations

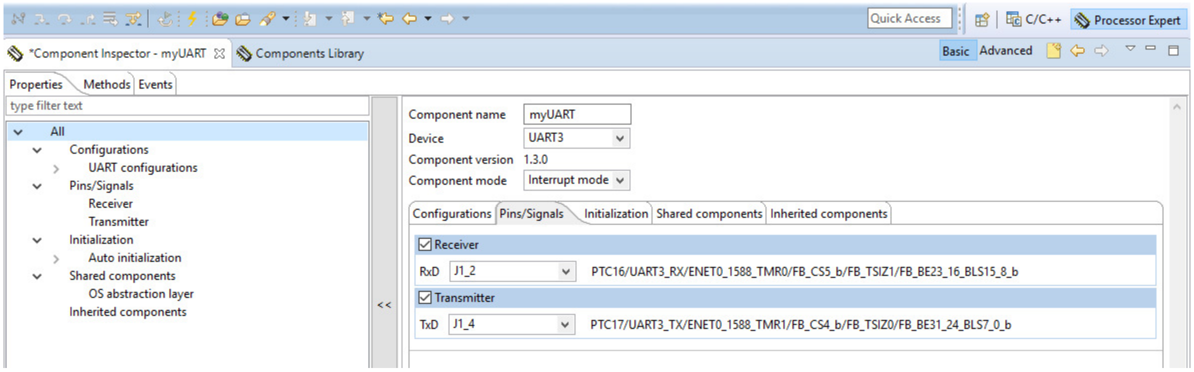
☑ UART configurations

Configurations list: 1

#	Configuration	Name	Type	Read only	Baud rate	Parity mode	Stop bits	Bits per char
0	☑	myUART_I...	uart_user_config_t	☑	115200 baud	Disabled	1	8

In "**Pins/Signals**" tab, choose "**J1_2**" as RxD pin, and "**J1_4**" as TxD. These pins are routed to the onboard connectors as shown below.





Component name: myUART

Device: UART3

Component version: 1.3.0

Component mode: Interrupt mode

Configurations | Pins/Signals | Initialization | Shared components | Inherited components

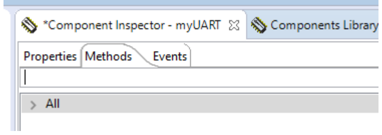
☑ Receiver

RxD: J1_2 — PTC16/UART3_RX/ENET0_1588_TMR0/FB_CS5_b/FB_TSIZ1/FB_BE23_16_BLS15_8_b

☑ Transmitter

TxD: J1_4 — PTC17/UART3_TX/ENET0_1588_TMR1/FB_CS4_b/FB_TSIZ0/FB_BE31_24_BLS7_0_b

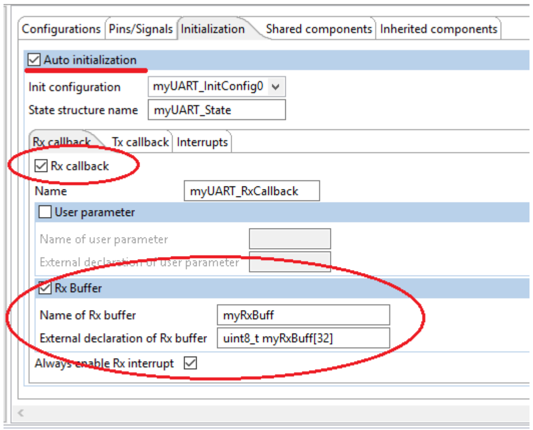
Now you can check out the methods you can use in regarding this component in the Methods menu. This will help you a lot regarding how to use each component.



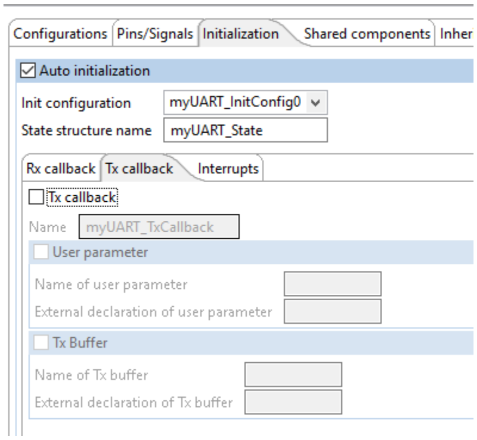
Back in the "**Properties**" tab, click on initialization tab of *myUART*, and hover your pointer over the (already checked) box called "**Auto Initialization**". This information helps you find the initialization functions for this component based on your current settings, it is useful in particular if you decide to use your code elsewhere (like in Project Generator) as in that case you will need to do the initializations yourself and this code can be a helpful start.

We need to enable the Receive interrupt and create a callback for it, this callback function will be called every time *myUART* receives a character. It is this function that we are going to modify in this example in order to echo the input stream back.

Enable "**Rx callback**" from the "**Initialization**" tab. You can change the name of your callback functions here but we will leave it as **myUART_RxCallback** for now. We can also create a buffer for *myUART* receiver here by choosing a name (e.g. "**myRxBuff"**) and a declaration (e.g. "**uint8_t myRxBuff[32]**"). Check the box called "**Always enable Rx Interrupt**". At this point we should Disable the callback function for Transmission (Tx callback). This is important because in order to use the Tx callback, some considerations need to be taken which are out of the focus of this document.

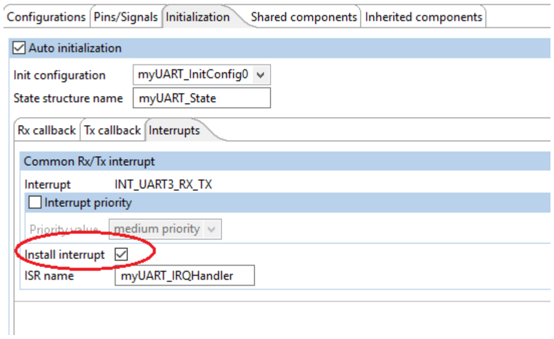


☑ Auto initialization
Init configuration myUART_InitConfig0
State structure name myUART_State
☑ Rx callback
Name myUART_RxCallback
☐ User parameter
Name of user parameter
External declaration of user parameter
☑ Rx Buffer
Name of Rx buffer myRxBuff
External declaration of Rx buffer uint8_t myRxBuff[32]
Always enable Rx interrupt ☑



☑ Auto initialization
Init configuration myUART_InitConfig0
State structure name myUART_State
☐ Tx callback
Name myUART_TxCallback
☐ User parameter
Name of user parameter
External declaration of user parameter
☐ Tx Buffer
Name of Tx buffer
External declaration of Tx buffer

Finally, in the "**Interrupts**" tab, Install a new ISR for the interrupt as shown below (It is needed by MQX).



☑ Auto initialization

Init configuration: myUART_InitConfig0

State structure name: myUART_State

Common Rx/Tx interrupt

Interrupt: INT_UART3_RX_TX

□ Interrupt priority

Priority value: medium priority

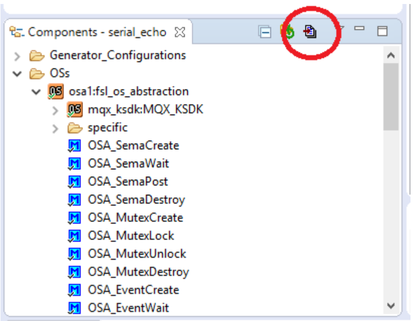
Install interrupt ☑

ISR name: myUART_IRQHandler

We have now finished setting up the components needed for our project.

Generating the code using PE

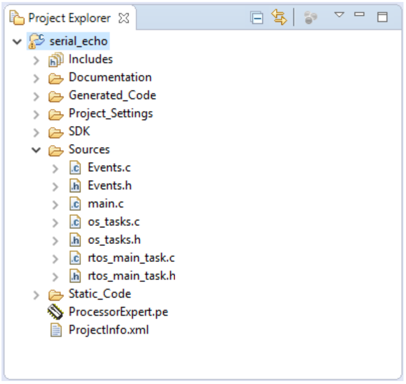
Now that we have finished setting up the components in the system, we need to generate the appropriate code using PE. You can do this either by right-clicking on *"ProcessorExper.pe"* in Project Explorer pane and choosing "Generate Processor Expert Code", or by clicking on **"Generate Processor Expert Code"** button in the lower left pane as shown below.



In this step you might be asked to approve some changes in the project files structure, which you can agree upon by clicking "**OK**".

Processor Expert code structure

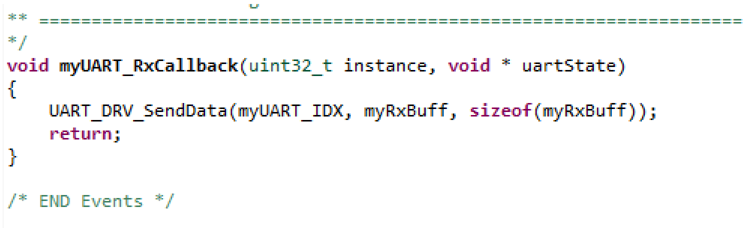
At this point, switch back to **C/C++** perspective. You can check the code structure PE creates in the Project Explorer pane. The main files you need to work with are located in **"Sources"** folder. You can access your generated task code in *os_tasks.c*. The Rx Callback function can be seen in *Events.c*. It is also a good practice at this moment to familiarize yourself with other folders, specially the **"Generated Code"** folder which includes initializations for your project components.



For this example, we are going to use the Rx callback function to return each character as it is received by UART 3. However, we have created **serialTask** for later improvement of this code. It is up to you to develop the code in order for **serialTask** to receive each character from the Rx callback via messaging and act upon it as needed. For now, just add the following lines to **serial_task()** in ***os_tasks.c.*** Be sure to add these lines before the while loop as we just want them to execute once. This is enough for now in order to test our debugging console and **UART3** transmission.

```c
void serial_task(os_task_param_t task_init_data)
{
  /* Write your local variable definition here */
  printf("serialTask Created!\n\r");

  char buf[13];
  sprintf(buf, "\n\rType here: ");
  UART_DRV_SendDataBlocking(myUART_IDX, buf, sizeof(buf), 1000);

#ifdef PEX_USE_RTOS
  while (1) {
#endif
    /* Write your code here ... */


    OSA_TimeDelay(10);                 /* Example code (for task release) */




#ifdef PEX_USE_RTOS
  }
#endif
}

/* END os_tasks */
```

Open "**Events.c**" and find **myUART_RxCallback()** function. Add the line of code as shown below.



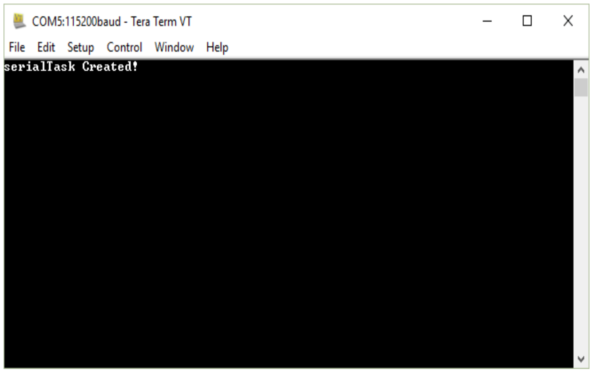
```
** ===================================================================
*/
void myUART_RxCallback(uint32_t instance, void * uartState)
{
    UART_DRV_SendData(myUART_IDX, myRxBuff, sizeof(myRxBuff));
    return;
}

/* END Events */
```

Using *UART_DRV_Senddata()* function, we echo back the buffer to **UART3** as soon as this ISR executes.

Testing the code

In order to test the code you need two separate runs of Tera term (One for debug console and one for UART 3). Use 115200 baud for both, then build and debug your project. The result should look like this:

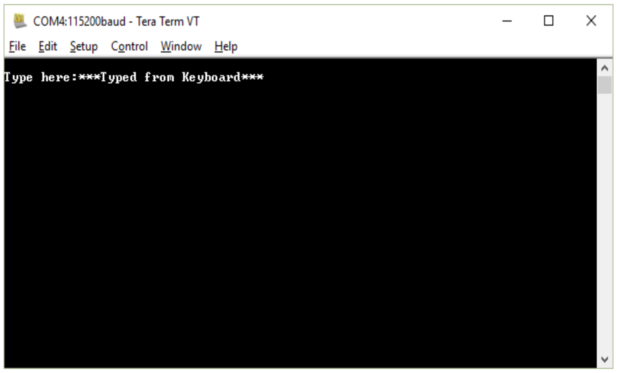
Debug console



COM5:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
serialTask Created!
```

UART3 terminal



COM4:115200baud - Tera Term VT
File Edit Setup Control Window Help

```
Type here:***Typed from Keyboard***
```

Part 2. Device Driver for Serial Channel

Serial Channels

This project introduces you to writing a device driver. You are required to develop a terminal driver capable of rudimental editing and buffering so that type-ahead can be used. This terminal driver must be able to handle data received and transmitted over a serial channel (in this case a UART). You are required to design and integrate the appropriate interrupting structures and create a Handler task to implement the functionality involved.

P2.1 Marking

This project counts for 45% of your Lab mark. Your mark for this project will be based mainly on your demo and your project documentation. The mark distribution is:

Overall Documentation: 10
Design Documents: 10
Code Design and Implementation: 10
Correctness: 15

Total: 45

P2.2 Project Report

Using your project report, a person familiar with this field should be able to understand and recreate your approach to the project. Some guidelines on what is expected in your project report are given below.

P2.2.1 Overall Documentation

The project report must be an appropriately formatted engineering report. The report should consist of an introduction, a discussion of the design (i.e., the design document), a discussion of the implementation, a brief summary, and an appendix with the source code. Note that the report will also be marked for presentation, grammar, punctuation, spelling, etc.

P2.2.2 Design Documents

Before you start any coding you must do a thorough design on paper. This should consist of diagrams indicating how the various tasks interact with each other, pseudo code, and so on.

P2.2.3 Code Design and Implementation

The code must be implemented in C with good coding style: meaningful variable names (e.g., use counter instead of ct), sufficient comments and documentation, modular design, and clean coding style (i.e., avoid gotos, etc.). Any problems encountered during implementation (i.e., bugs), testing methodology, and known bugs should also be described.

P2.2.4 Correctness

This mark will depend on how much of the project works correctly, and if there are any design faults in the implementation.

P2.3 Project Description and Guidelines

The following is a description of the functionality you are required to implement. It is assumed that there is only one serial line in your hardware, and all the functionality described refers to this one serial device. The PC acts as a dumb terminal. When a character is typed on the keyboard the character is sent via the serial channel to the MCF52259DEMOKIT board and is to be received by the terminal driver on board. The terminal driver then echoes the character to the monitor via the same serial channel. This interaction is depicted in FIGURE P2.1 and FIGURE P2.2 below.
The received character may be a special character (e.g. DELETE). The terminal driver interprets
such special characters and acts accordingly manipulating the text that is displayed on the monitor.

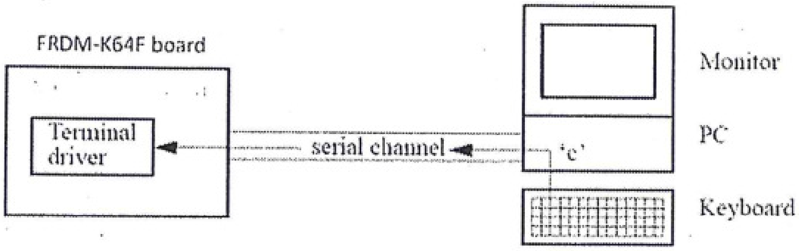


FIGURE P2.1. Example of the terminal driver receiving the character 'c'.

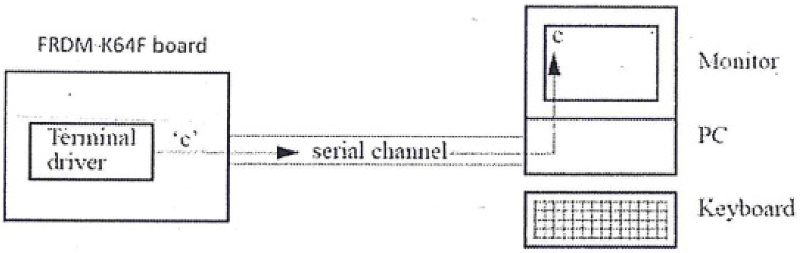


FIGURE P2.2. Example of the terminal driver echoing to the monitor the character 'c'.

The overall architecture of the terminal driver is depicted in FIGURE P2.3 and FIGURE P2.4. Note that user tasks are tasks created by the user. These tasks are not part of the terminal driver!

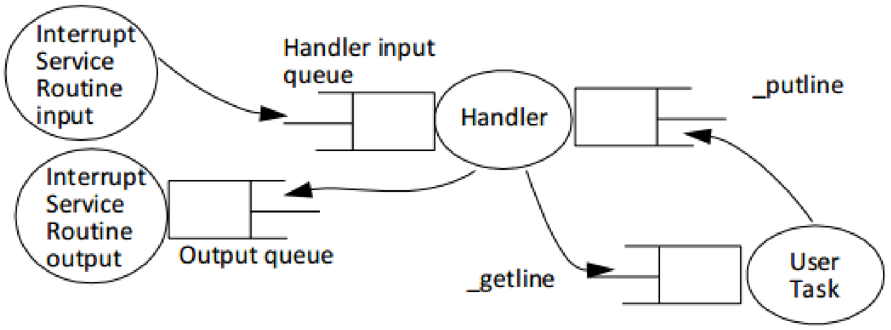

Interrupt Service Routine input
Handler input queue
Handler
_putline
Interrupt Service Routine output
Output queue
_getline
User Task

FIGURE P2.3. Overall system configuration.

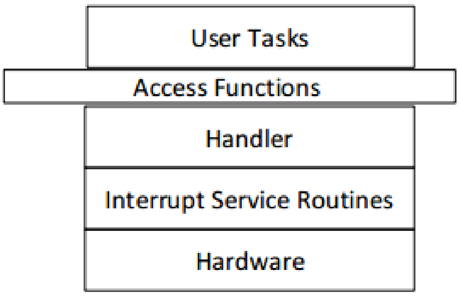


FIGURE P2.4. Overall system layers. Note that the “Access Functions” is not a layer, but an API and specifies the manner in which the user task layer communicates with the Handler task.

There are two parts to this project. The first part is writing the Interrupt Service Routines (ISRs) and the Handler task. The ISRs handle the *receive* and *transmit* interrupts caused by the receiving and transmitting of a character. The Handler is responsible for ‘dealing with’ the arrival of this character. The Handler task must always be running as it is part of the operating system. The second part of the project is writing the five access functions for the user tasks. These access functions allow the user tasks to gain read and write access to the serial channel.

P2.3.1 Handler Task and Interrupt Handlers

When a character arrives at serial port 3 (UART3) of the FRDM-K64F board an interrupt occurs. You need to write an ISR that takes the received character and passes it to the Handler. The ISR does this by placing the character on a 'handler input queue'. If no user task has opened the device for reading, characters received should be sent to the null device (i.e., discarded) and not be echoed at the terminal. If however, at least one user task has opened the device for reading then in the case of a printable character the Handler simply echoes the character to the screen (i.e., the character is placed on an 'output queue'.) An output queue is a system queue rather than a task queue.
The character must be sent back via the serial channel. A transmit interrupt occurs just after the UART finishes transmitting a character, i.e., the UART is 'telling you' (by causing an interrupt) that it is ready to transmit another character. If the output queue is not empty, then the ISR should be able to obtain a character from the output queue and transmit it. If the output queue is empty, then the ISR does nothing. A notification function attached to the output queue can be used to forward a subsequent character to the serial channel.

The Handler task should be able to perform simple line editing. That is, it should interpret special command characters appearing in the input sequence (including ^H: erase character, ^W: erase previous word, ^U: erase line) and implement these commands by deleting the appropriate entity from the input stream to be delivered to the waiting tasks, and by erasing the echoed characters on the terminal window. ASCII codes for these special characters can be found on a Unix system by issuing the command: man ascii. Additional information on terminal capabilities can be found in the file /etc/termcap.

Some pseudo code for the Handler task is shown below:

```
Initialize system

Do forever

  Check handler queue
  If a user task has opened the device for reading
      Determine what the character is
      If the character is printable put it in the notification queue
      Else if the character is a special character, handle it appropriately
      Else if the character is none of the above, ignore it
  Else discard the character
```

Once the first part of the project is completed (i.e., the Handler task and ISRs are written and working correctly) you are required to write some functions that will allow user tasks to access the serial channel.

P2.3.2 User Task Access Functions

You have to write five functions accessible by user tasks. Note that these are functions, not tasks.User tasks do not ‘know’ about the Handler task and cannot access the serial channel directly. User tasks can only access the serial channel through these access functions. The Handler task will need to be modified from the above description to accommodate OpenR and OpenW. There are two ways to implement these functions: using global data structures or message passing.

P2.3.2.1 OpenR

boolean OpenR(*uint_16 stream_no*)

Before a user task can access the serial channel for reading, the task must call OpenR. The task will open the serial device and attach to it the *stream_no* specified. The *stream_no* is the number of a queue owned by the user task at which any character received by the Handler will be sent. Many tasks can have read privileges. All user tasks with read privilege should receive identical copies of the stream of characters arriving at the device. If OpenR grants the requesting user task read privilege, the function should return TRUE, otherwise it should return FALSE. If a task already has read privilege, OpenR should return FALSE.

If you implement the access functions using global data structures, then OpenR accesses a data structure that indicates which user task(s) have read privilege. Note that the OpenR data structure is a shared resource (i.e., it is possible that more than one task could access the data structure at the same time). If you implement the access functions using message passing technique, then OpenR sends a message to the Handler task, and the Handler task maintains the data structure that indicates which user task(s) have read privilege. In this case the data structure is not a shared resource. In both cases the data structure should contain task ids and the associated queue ids of the tasks that have read privilege. Task ids can be found using _task_get_id().

P2.3.2.2 _getline

boolean _getline(pointer string)

OpenR only gives user tasks read privilege. For a user task to actually get data from the serial channel it must call the blocking _getline. When the Handler task receives a return on the serial channel (indicated by a ‘\n’) it distributes the most recent line of characters it has received to all the tasks with reading privilege. It places the received line in the character array string, normally returning TRUE. FALSE is returned on error or if the calling task does not have read privilege.

P2.3.2.3 OpenW

_queue_id qid OpenW(void)

Before a user task can access the serial channel for writing, the task must call OpenW. OpenW allows only ONE user task to have write privilege on the serial channel. If the device has already been opened for writing (by this task, or any other task) this call should return an error (i.e., queue id 0). Upon successful return, this call returns the qid of the queue attached to the device handler task.

If you implement the access functions using global data structures, then OpenW accesses a data structure that indicates which user tasks have write privilege. Note that the OpenW data structure is a shared resource (i.e., it is possible that more than one task could access the data structure at the same time). If you implement the access functions using message passing technique, then OpenW sends a message to the Handler task, and the Handler task maintains the data structure that indicates which user task has write privilege. In this case the data structure is not a shared resource.

P2.3.2.4 _putline

boolean _putline(_queue_id qid, pointer string)

A user task must call _putline after receiving write privilege to actually put a string onto the serial channel. _putline sends the string of characters in the array *string*, appended with a '\n', to the queue specified by *qid* and returns TRUE. The Handler task is responsible for further processing of the string of characters received and forwarding them to the device. FALSE is returned on error or if the calling task does not have write privilege.

P2.3.2.5 Close

boolean Close(void)

A user task calling this function will have its read and/or write privileges revoked. On successful closing of a task's read privilege, write privilege, or both, TRUE is returned. If the calling task does not have read or write privilege, FALSE is returned.

If you implement the access functions using global data structures, then Close accesses both the read privilege data structure and the write privilege data structure. If you implement the access functions using message passing technique, then Close sends a message to the Handler task, and the Handler task maintains the data structures as necessary.

P2.4 Testing

There are numerous ways to test your code. One suggested approach is to start the Handler task with a Master task. This Master task would initially request read and write privileges. The Master task would respond to typed commands from the user; one of the commands would be for the Master task to create Slave tasks. These Slave tasks would initially request read privilege. The Slave tasks would also respond to typed commands from the user.

Your test code should be capable of covering all possible combinations of user function calls. It is also important that your test code does not have internal knowledge of the inner workings of your Handler (and vice versa); testing should be done at the access function level.

P2.5 Notes

This project is extremely difficult if not impossible to ‘hack’ together. Doing your design of the interrupt handlers, Handler task, and the user task access functions on paper is a must (i.e., detailed pseudo code). You will not save yourself any time by skipping this step and in fact it will probably take you much longer to complete the project. Furthermore, you are required to hand in your design document as part of your report. As a rough estimate, you should expect to spend at least 20 hours designing, implementing, and testing the project.

Low level specifics (such as the interrupt vector and the control register addresses of the serial channel) will be given in the lab.

When doing the second part of the project, it is probably easier to implement OpenW and _putline first.

University of Victoria

Department of Electrical and Computer Engineering

CENG 455
REAL TIME COMPUTER SYSTEMS
DESIGN PROJECTS

Project 3

Project 3

This project is intended to introduce you to writing software and interfacing hardware that deal with periodic and aperiodic tasks that have hard deadlines. In the past there have been various options for this project. The results have sometimes been wonderful, but at a cost, since everybody has several courses to deal with, and sometimes unexpected issues happen which consume valuable time. It is highly recommended that the main project, the deadline driven scheduler, be built as Project 3.

If you really want more of a challenge, or you'd really like to do something else, please consider building the default Project 3 first quickly, so that you have fulfilled the target requirements, and then you will be in an excellent position to judge how much time is still available to do further work. Since the software is free and the FRDM series of microcontroller boards are both inexpensive and readily available, you can take a project further at any future time that suits you best.

Otherwise, please discuss options with both your instructor and TA before proceeding, And if doing any hardware I/O, be sure that you have a TA approved over-voltage, over-current protection setup in place, so that the board doesn't get damaged, affecting you and other users of these shared boards.

P3.1 Marking

This project accounts for 50% of your Lab mark. Your mark for this project will be based mainly on your demo and your project documentation. The mark distribution is:

1.	Based on final report:	
	Overall Documentation:	10
	Design Documents:	10
2.	Based on demo:	
	Code Design and Implementation:	10
	Correctness:	20
	Total:	50

P3.2 Project Presentation

Each group is required to make a 20-minute presentation and demonstration describing the project done, the approach taken, and the results obtained, to the instructors during the last week of classes.

P3.3 Project Report

Using your project report, a person familiar with the field should be able to understand and recreate your approach to the project. Some guidelines on what is expected in your project report are given below (as per P3.1 Marking Distribution).

CENG 455 Project Manual 3

P3.3.1 Overall Documentation

The project report must be an appropriately formatted engineering report. The report should consist of an introduction, a discussion of the design (i.e., the design document), a discussion of the implementation, a brief summary, and an appendix with the source code. Note that the report will also be marked for grammar, punctuation, spelling, etc.

P3.3.2 Design Documents

Before you start any coding you must do a thorough design on paper. This should consist of diagrams indicating how the various tasks interact with each other, pseudo code, and so on.

P3.3.3 Code Design and Implementation

The code must be implemented in C with good coding style: meaningful variable names (e.g., use counter instead of ct), sufficient comments and documentation, modular design, and clean coding style (i.e., avoid GOTOs, etc.). Any problems encountered during implementation, testing methodology, and known bugs should also be described.

P3.3.4 Correctness

This mark will depend on how much of the project works correctly, and if there are any design faults in the implementation. For all projects, you must perform testing to determine a variety of measures such as processor efficiency, upper bounds and limits of the tasks, and so on.

P3.3.5 Notes

Not every project will necessarily be offered in a given year.

CENG 455 Project Manual 4

Project 3A Deadline Driven Scheduler

This project is intended to introduce you to writing a task scheduler, using the deadline driven scheduling algorithm. You are required to test the scheduler and collect statistics with a variety of task loads.

P3A.1 Project Description and Guidelines

The following describes what is required for the deadline driven scheduler project. There are two parts to the project: the deadline driven scheduler task (DD-scheduler), and the auxiliary tasks. The auxiliary tasks are required for testing the DD-scheduler and for collecting statistics to determine processor utilization and system overhead.

P3A.1.1 DD-scheduler

You are to write a general scheduler capable of dynamically altering the priorities of the currently active tasks so that a dynamic real time environment can be controlled. You are to consider the priority-based task scheduling provided by MQX as the mechanism, while the scheduler you are to implement is the policy that will use the mechanism to achieve fine scheduling control. Because not enough 'hooks' are supplied with MQX to access and modify its native scheduler, the policy scheduler is to be implemented as a supervisory task. The policy you are to implement is the deadline driven scheduling.

The policy scheduler, to be called the DD-scheduler, is a task of the highest priority. The DD-scheduler is a task that is responsible for scheduling 'user tasks'[1] . By definition of this system, the user tasks must have fixed execution times, and hard deadlines. It is also assumed that the user tasks execute independently of each other; for example, a user task would not depend on a message from another user task in order to complete its execution. The user tasks can be periodic or aperiodic. The DD-scheduler task is normally suspended (it pends on its input queue for policy request messages to arrive). It resumes every time a scheduling request message arrives.

These messages arrive
(a) at task creation time so that the correct priority could be given to the new task
(b) at task deletion time, so that the task with the earliest deadline could be assigned the highest priority
(c) at the next deadline[2]

1 User tasks are tasks that the user has created and wants scheduled. These tasks are not part of the operating system or the DD-scheduler.

2 In the case of periodic tasks, the task deadline coincides with the subsequent request of the task. Thus in this case, an explicit invocation of the scheduler at the deadline is not necessary. On the other hand, if the task is aperiodic, then the scheduler must also be awaken at the deadline to ensure that the task has not overrun its deadline. Of course if the task has terminated (and deleted itself) prior its deadline, this must cancel the awakening of the scheduler at the task's deadline.

(d) when an information_request message arrives.

The DD-scheduler, maintains a list of active tasks with their deadlines and priorities. The DD-scheduler keeps a list of current user tasks; this list is sorted by the user tasks' deadlines. You must determine what sorting algorithm to use to sort the task list. Upon activation, the DD-scheduler obtains a message from its input queue, and acts on the type of the message. If for example the message was a task creation message, the scheduler will introduce the new task in its list of active tasks assign the correct priorities to them and finally reply to the requestor with a status message. The DD-scheduler must adjust the priorities of the user tasks to ensure that the user task with the soonest deadline is running. If the message was an information_request message, the DD-scheduler will simply post the requested information (e.g. the list of the active tasks) at the destination queue specified by the requesting message.

The DD-scheduler also deletes tasks which have exceeded their deadlines, and places their ID in the overdue task list. At the end, it pends on its input queue for the next message to arrive.

The DD-scheduler uses the two data structures given in Table 3A-1and Table 3A-2.

TABLE 3A-1 struct task_list

```
struct task_list {
        uint_32 tid;
        uint_32 deadline;
        uint_32 task_type;
        uint_32 creation_time;
        struct task_list *next_cell;
        struct task_list *previous_cell;
}
```

TABLE 3A-2 struct overdue_tasks

```
struct overdue_tasks {
        uint_32 tid;
        uint_32 deadline;
        uint_32 task_type;
        uint_32 creation_time;
        struct overdue_tasks *next_cell;
        struct overdue_tasks *previous_cell;
}
```

These data structures are internal to the DD-scheduler and cannot be accessed directly by any other tasks (i.e., any of the auxiliary tasks).

There are four functions that are part of the DD-scheduler (note that these are functions not tasks):

_task_id dd_tcreate(template_index, deadline)
uint_32 dd_delete(task_id)
uint_32 dd_return_active_list(**list)
uint_32 dd_return_overdue_list(**list)

As the DD-scheduler can be considered part of the operating system, the user tasks can only access the DD-scheduler through the above functions.

P3A.1.1.1 dd_tcreate

When a user task is created using dd_tcreate the scheduler must put the task id (and other information) into the task list and then sort the list.

_task_id dd_tcreate(uint_32 template_index, uint_32 deadline)

This primitive, creates a deadline driven scheduled task. It follows the steps outlined below
1. Opens a queue
2. Creates the task specified and assigns it the minimum priority possible
3. Composes a create_task_message and sends it to the DD-scheduler
4. Waits for a reply at the queue it created above
5. Once the reply is received, it obtains it
6. Destroys the queue
7. Returns to the invoking task

template_index is the template index of the task to be created.

deadline is the number of clock ticks to the task's deadline.

It returns the task_id of the created task, or an error. The error is either an MQX task creation error or a DD-scheduler specific error (to be determined).

P3A.1.1.2 dd_delete
uint_32 dd_delete(uint_32 task_id)

When the function dd_delete is called, the specified task is deleted.

This primitive deletes the task specified. It parallels the structure of the dd_tcreate as outlined above.

P3A.1.1.3 dd_return_active_list
uint_32 dd_return_active_list(struct task_list **list)

This primitive requests the DD-scheduler for the list of active tasks and returns this information to the requestor.

The designer must decide whether the list should be copied and sent to the requestor, or simply a pointer pointing to the start of the list could suffice. Analyze the alternatives and justify your implementation choice.

P3A.1.1.4 dd_return_overdue_list
uint_32 dd_return_overdue_list(struct overdue_tasks **list)

This primitive requests the DD-scheduler for the list of overdue tasks and returns this information to the requestor. Similar in structure as the dd_return_active_list primitive.

P3A.1.2 Auxiliary Tasks
To test the deadline driven scheduler, some auxiliary tasks are required. The auxiliary tasks consist of: a task or tasks used to generate periodic user tasks; periodic user tasks; and a monitor task. Note that the DD-scheduler is independent of the auxiliary tasks; i.e., the auxiliary tasks exist solely to test the system. The auxiliary tasks can only access the DD-scheduler using the four functions defined for the DD-scheduler; the auxiliary tasks should not have any direct access to the data structures internal to the DD-scheduler.

P3A.1.2.1 Task(s) for Generating Periodic User Tasks
To test the DD-scheduler you must generate periodic user tasks for the DD-scheduler to schedule. There are at least two different ways to generate periodic user tasks: a periodic tasks generator, or periodic task generators.

A periodic task generator is a single task responsible for creating all periodic user tasks at the beginning of their periods. In the case of the periodic task generators, each periodic user task would have a generator task whose only job would be to create the periodic user task at the beginning of its period. These the two methods are illustrated in Figure 3A-1.

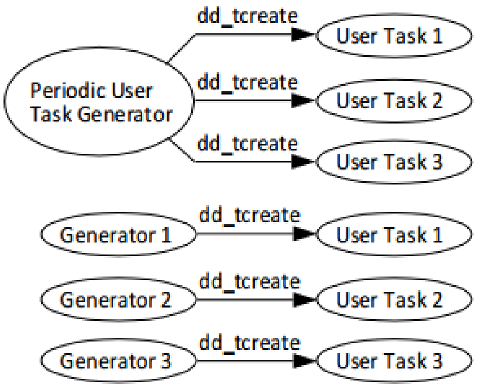


Figure 3A-1 Method for generating periodic user tasks

The periodic user tasks are described in the next section. Note that the task generator(s) do not get scheduled by the DD-scheduler.

P3A.1.2.2 Periodic User Task(s) The periodic user tasks are the tasks that the DD-scheduler will schedule. For this project, these tasks do not do anything useful3 ; i.e., a periodic user task just executes an empty loop for the duration of its execution time. A periodic user task must be created at the beginning of its period using dd_tcreate with a fixed execution time (i.e., 500 msec.) and a fixed period (i.e., 3000 msec.). A periodic user task must delete itself using dd_delete when it has finished its execution. A sample periodic user tasks test set is shown in Table 3A-3.

TABLE 3A-3 Sample Periodic User Tasks Test Set

Task	Execution time (C)	Period (T)
t_1	85	250
t_2	150	500
t_3	250	750

The scheduling diagram for approximately the first 600 time units for the task set shown in Table 3A-3, is illustrated in Figure 3A-2.

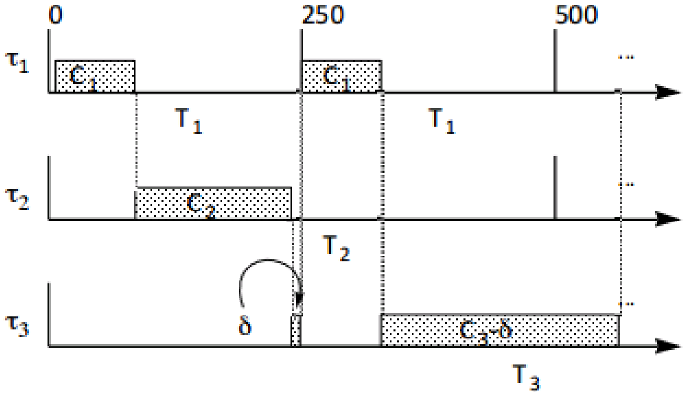


Figure 3A-2 Scheduling diagram for sample test set

Note that the diagram in Figure 3A-2 is incomplete. It is important to keep in mind that the task

[3] Obviously in a real system the user tasks would do something useful such as perform a calculation. However for this project, the user tasks exist only to test the DD-scheduler.

set given in Table 3A-3 is only an example; this task set is not necessarily the best task set for testing your DD-scheduler.

P3A.1.2.3 Monitor Task

The monitor task is a task with the lowest priority that will run when no other tasks are running. This task is used to calculate and report processor utilization and system overhead. Note that the monitor task does not get scheduled by the DD-scheduler.

P3A.2 Notes

- As with the second project, this project is extremely difficult if not impossible to 'hack' together. Again you should do your design of the DD-scheduler, and auxiliary tasks on paper before you do any coding.

- Program I/O (i.e., printfs) should be kept to a minimum as this increases processor overhead.

- Although you will test the DD-scheduler with periodic tasks, the DD-scheduler must be able to handle aperiodic tasks as well.

- You are required to measure the processor utilization and system overhead using the monitoring task. To calculate processor utilization and system overhead, you need to find an accurate way to measure time in your system.

- You should demonstrate your work and results with a simple toy application that requires real-time functionality, perhaps using the RGB LED and pushbuttons on the FRDM-K64F. Use your work from Project 2 to implement a command line tester. Bonus marks may be awarded to a novel real-time DD scheduler application.