Freescale Semiconductor
Application Note

Document Number: AN3902
Rev. 0, 08/2009

How to Develop I/O Drivers for MQX

by: Carlos Neri
Applications Engineering
Freescale Technical Support

Luis Reynoso
Field Applications Engineer
North Florida

Contents

1 Introduction

The increasing complexity of industrial applications and expanding functionality of semiconductors are driving embedded developers towards solutions that combine proven hardware and software platforms.

To help accelerate time to market and improve application development success, Freescale Semiconductor is offering the Freescale MQX Real-Time Operating System (RTOS) with TCP/IP and USB software stacks to particular ColdFire® microcontroller (MCU) families at no additional charge.

The Freescale MQX Software Solution includes a comprehensive Board Support Package (BSP) supporting common peripherals and functions. However, some applications will require customization of the available drivers or the development of new ones.

The purpose of this application note is to guide developers through the process of creating and testing I/O drivers under MQX.

Detailed information about MQX RTOS, BSP, or drivers can be found in Freescale's website and inside the MQX software available at: http://www.freescale.com/mqx.

This application note is based on the ColdFire MCF5225x processor, which is a 32-bit device based on the Version 2 ColdFire Core, offering high performance and low power consumption.

On-chip memories connected tightly to the processor core with up to 512 KB of flash memory and 64 KB of SRAM, and a rich set of peripherals including USB OTG, Fast Ethernet Controller, CAN, and encryption makes this processor an ideal option for factory automation, building control, and medical applications.

2 I/O Device Drivers Basics

I/O device drivers are dynamically installed software packages that provide a direct interface to hardware. They are commonly installed by calling:

- `io_`***`device`***`_install()` (where ***`device`*** is replaced with the name of the driver family). This function may call `_io_dev_install()` to register the device with MQX.

The I/O device driver must provide the following services:

- `_io_`***`device`***`_open` — This function is required.
- `_io_`***`device`***`_close` — This function is required.
- `_io_`***`device`***`_read` — This function is optional.
- `_io_`***`device`***`_write` — This function is optional.
- `_io_`***`device`***`_ioctl` — This function is optional.

2.1 Customizing MQX

For simplicity, this application note uses and modifies the default MQX configuration.

If you want to create a custom MQX configuration or a new BSP, please refer to Freescale document MQXUG, *Freescale MQX Real-Time Operating System User's Guide*, and follow the instructions in Chapter 4, "Rebuilding MQX," or Chapter 5, "Developing a New BSP."

2.1.1 Device Drivers in Freescale MQX BSP

The Freescale MQX software solution includes several I/O device drivers that can be used as a reference to develop your own. These drivers are located in your default installation folder (referred to in this document as "<MQX_folder>") inside the following path:

<MQX_folder>\mqx\source\io

In order to modify these drivers and re-build the BSP, open the BSP project at:

M52259EVB: <MQX_folder>\mqx\build\codewarrior\bsp_m52259evb.mcp

M52259DEMO: <MQX_folder>\mqx\build\codewarrior\bsp_m52259demo.mcp

The next sections explain how to create one of these device drivers from scratch.

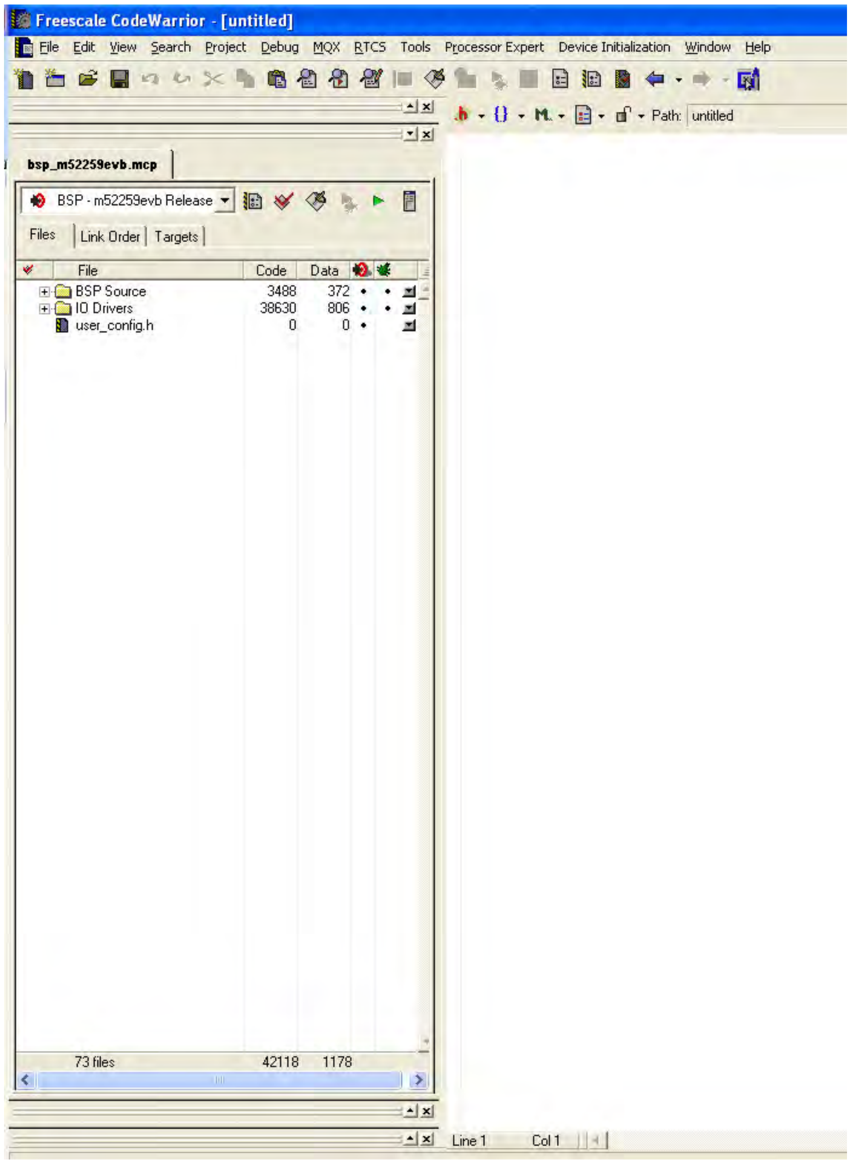
3 Null Driver

The null device driver is an I/O device that functions as a device driver but doesn’t perform any work. It can be used to understand and test basic communication with the I/O subsystem.

3.1 Create a New I/O Null Driver

This sub-section guides you through the process of creating a new I/O null driver from scratch.

1. Open the BSP project in Codewarrior located at the following path:
   M52259EVB: <MQX_folder>\mqx\build\codewarrior\bsp_m52259evb.mcp
   M52259DEMO: <MQX_folder>\mqx\build\codewarrior\bsp_m52259demo.mcp
2. Create a new text file (Menu→File→New Text File).


Freescale CodeWarrior - [untitled]
File Edit View Search Project Debug MQX RTCS Tools Processor Expert Device Initialization Window Help
Path: untitled
bsp_m52259evb.mcp
BSP - m52259evb Release
Files Link Order Targets
Line 1 Col 1

File	Code	Data
BSP Source	3488	372
IO Drivers	38630	806
user_config.h	0	0
73 files	42118	1178

3. Include the required system headers:

```
#include "mqx.h" /* Structures and constants used by MQX */
#include "fio.h" /* standard formatted I/O library */
#include "io.h"  /* I/O subsystem interface. */
#include "my_null_io.h" /*This is the header for this driver */
```

4. Add the prototypes for all your private functions:

```
_mqx_int _io_my_null_open(FILE_PTR, char_ptr, char_ptr);
_mqx_int _io_my_null_close(FILE_PTR);
_mqx_int _io_my_null_read (FILE_PTR, char_ptr, _mqx_int);
_mqx_int _io_my_null_write(FILE_PTR, char_ptr, _mqx_int);
_mqx_int _io_my_null_ioctl(FILE_PTR, _mqx_uint, pointer);
```

5. Declare the install function:

```
_mqx_uint _io_my_null_install
   (
      /* [IN] A string that identifies the device for fopen */
      char_ptr            identifier
   )
{
 _mqx_uint result;

   result = _io_dev_install(identifier,
      _io_my_null_open,
      _io_my_null_close,
      _io_my_null_read,
      _io_my_null_write,
      _io_my_null_ioctl,
      NULL);

      return result;
}
```

The parameter `identifier` is an input string identifying the device for `fopen`.

Note that the function calls `io_dev_install` which requires the pointers to all the driver functions as a parameter plus a pointer to the I/O initialization data, which in this case is not used.

6. Declare your open, close, read, write, and ioctl functions:

   a) _io_device_open

      – Purpose:

        This function is called when the user calls `fopen`. It prepares the driver for subsequent read, write, and ioctl operations.

      – Parameters:

        `FILE_DEVICE_STRUCT_PTR` *`fd_ptr`*: Pointer to a file device structure with useful elements such as error flags, file size, or a pointer to device-specific information.

        `char_ptr` *`open_name_ptr`*: Pointer to a string used to open the device. It can be used to identify a device if multiple instances are called or to pass parameters to the function.

        `char _PTR_` *`flag:`* flags passed as a parameter from `fopen`.

- Returns:

  `_mqx_int`: Integer with the error code.
- Implementation of my_null driver:

```
_mqx_int _io_my_null_open
   (
      /* [IN] the file handle for the device being opened */
      FILE_PTR   fd_ptr,

      /* [IN] the remaining portion of the name of the device */
      char_ptr   open_name_ptr,

      /* [IN] the flags to be used during operation:
      ** echo, translation, xon/xoff, encoded into a pointer.
      */
      char_ptr    flags
   )
{ /* Body */

   /* Nothing to do */
   return(MQX_OK);

} /* Endbody */
```

b) _io_device_close
- Purpose:

  This function is called when the user calls `fclose` and it closes the driver releasing all resources.
- Parameters:

  `FILE_DEVICE_STRUCT_PTR fd_ptr`: Pointer to a file device structure with useful elements such as error flags, file size, or a pointer to device-specific information.
- Returns:

  `_mqx_int`: Integer with the error code.
- Implementation of my_null driver:

```
_mqx_int _io_my_null_close
   (
      /* [IN] the file handle for the device being closed */
      FILE_aPTR   fd_ptr
   )
{ /* Body */

   /* Nothing to do */
   return(MQX_OK);

} /* Endbody */
```

c) _io_device_read
- Purpose:

  Used to retrieve data from the device and executed when the user calls `read`.

- Parameters:

  `FILE_DEVICE_STRUCT_PTR fd_ptr`: Pointer to a file device structure with useful elements such as error flags, file size, or a pointer to device-specific information.

  `char_ptr    data_ptr`: Pointer to a char array where data will be stored.

  `_mqx_int    num`: Number of bytes to read.

- Returns:

  `_mqx_int`: Number of characters read and/or error code.

- Implementation of my_null driver:

```
_mqx_int _io_my_null_read
   (
      /* [IN] the file handle for the device */
      FILE_PTR   fd_ptr,

      /* [IN] where the characters are to be stored */
      char_ptr   data_ptr,

      /* [IN] the number of characters to input */
      _mqx_int   num
   )
{ /* Body */
   return(0);
} /* Endbody */
```

d) _io_device_write

- Purpose:

  Sends data to the device when the user calls a write function.

- Parameters:

  `FILE_DEVICE_STRUCT_PTR fd_ptr`: Pointer to a file device structure with useful elements such as error flags, file size, or a pointer to device-specific information.

  `char_ptr    data_ptr`: Pointer to a char array where the data is.

  `_mqx_int    num`: Number of bytes to write.

- Returns:

  `_mqx_int`: Number of characters written and/or error code.

- Implementation of my_null driver:

```
_mqx_int _io_my_null_write
   (
      /* [IN] the file handle for the device */
      FILE_PTR   fd_ptr,

      /* [IN] where the characters are */
      char_ptr   data_ptr,

      /* [IN] the number of characters to output */
      _mqx_int   num
   )
{ /* Body */
   return(num);
} /* Endbody */
```

e) _io_device_ioctl

– Purpose:

This function offers a way to issue device-specific commands (like setting UART baud rate, which is neither read nor write). It's executed when the user calls ioctl.

– Parameters:

`FILE_DEVICE_STRUCT_PTR fd_ptr`: Pointer to a file device structure with useful elements such as error flags, file size, or a pointer to device-specific information.

`_mqx_uint cmd`: Command sent as a parameter from ioctl call.

`pointer     param_ptr`: Parameters passed from ioctl call.

– Returns:

`_mqx_int`: Result of ioctl operation (depends on the command) and/or error code.

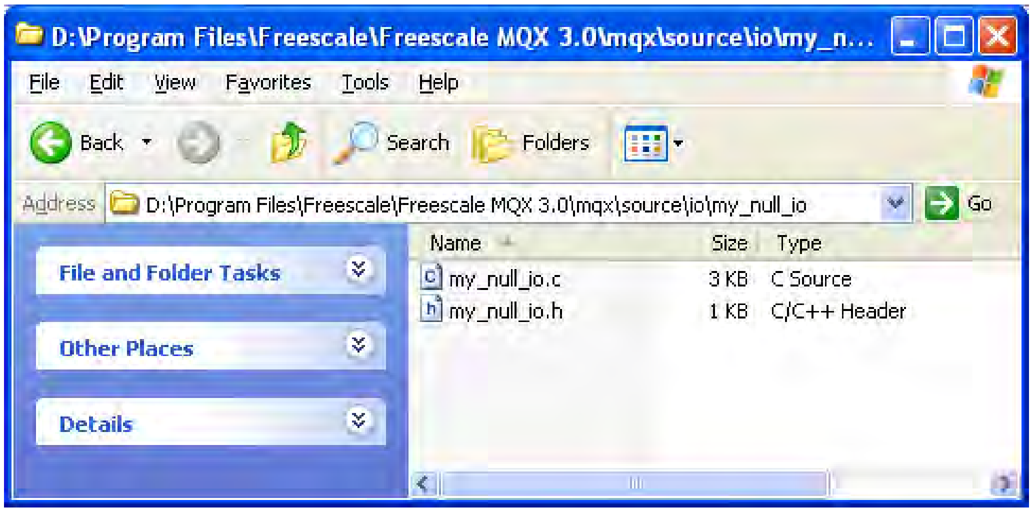
– Implementation of my_null driver:

```
_mqx_int _io_my_null_ioctl
   (
      /* [IN] the file handle for the device */
      FILE_PTR   fd_ptr,

      /* [IN] the ioctl command */
      _mqx_uint cmd,

      /* [IN] the ioctl parameters */
      pointer    param_ptr
   )
{ /* Body */
     return IO_ERROR_INVALID_IOCTL_CMD;
} /* Endbody */
```

7. Create a folder inside the BSP to contain your device driver. For this example we use:
   <MQX_folder>\mqx\source\io\my_null_io
8. Save your file as:
   <MQX_folder>\mqx\source\io\my_null_io\my_null_io.c
9. Create a new text file to be used as the header file for our driver.
10. Add the prototype for the public install function:

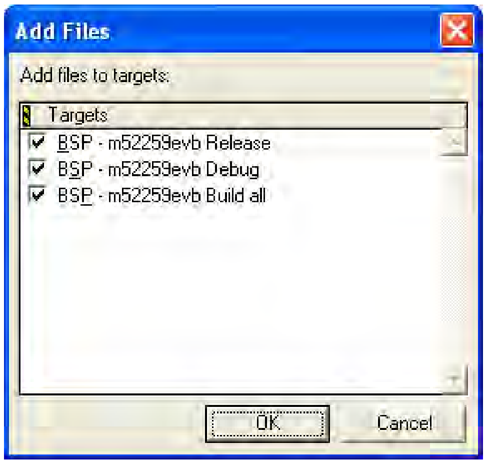
```
#ifndef __my_io_null_h__
#define __my_io_null_h__

#ifdef __cplusplus
extern "C" {
#endif

extern _mqx_uint _io_my_null_install(char_ptr);

#ifdef __cplusplus
}
#endif

#endif
```

11. Save your file as:
    <MQX_folder>\mqx\source\io\my_null_io\my_null_io.h


D:\Program Files\Freescale\Freescale MQX 3.0\mqx\source\io\my_n...
File Edit View Favorites Tools Help
Back
Search
Folders
Address D:\Program Files\Freescale\Freescale MQX 3.0\mqx\source\io\my_null_io
Go
File and Folder Tasks
Other Places
Details

Name	Size	Type
my_null_io.c	3 KB	C Source
my_null_io.h	1 KB	C/C++ Header

12. Drag-and-drop the whole my_null_io folder to your Codewarrior project inside the IO Drivers folder. (Optionally, a group can be created and files can be added manually to Codewarrior).
13. Click OK in the pop-up window to add the files to all targets.

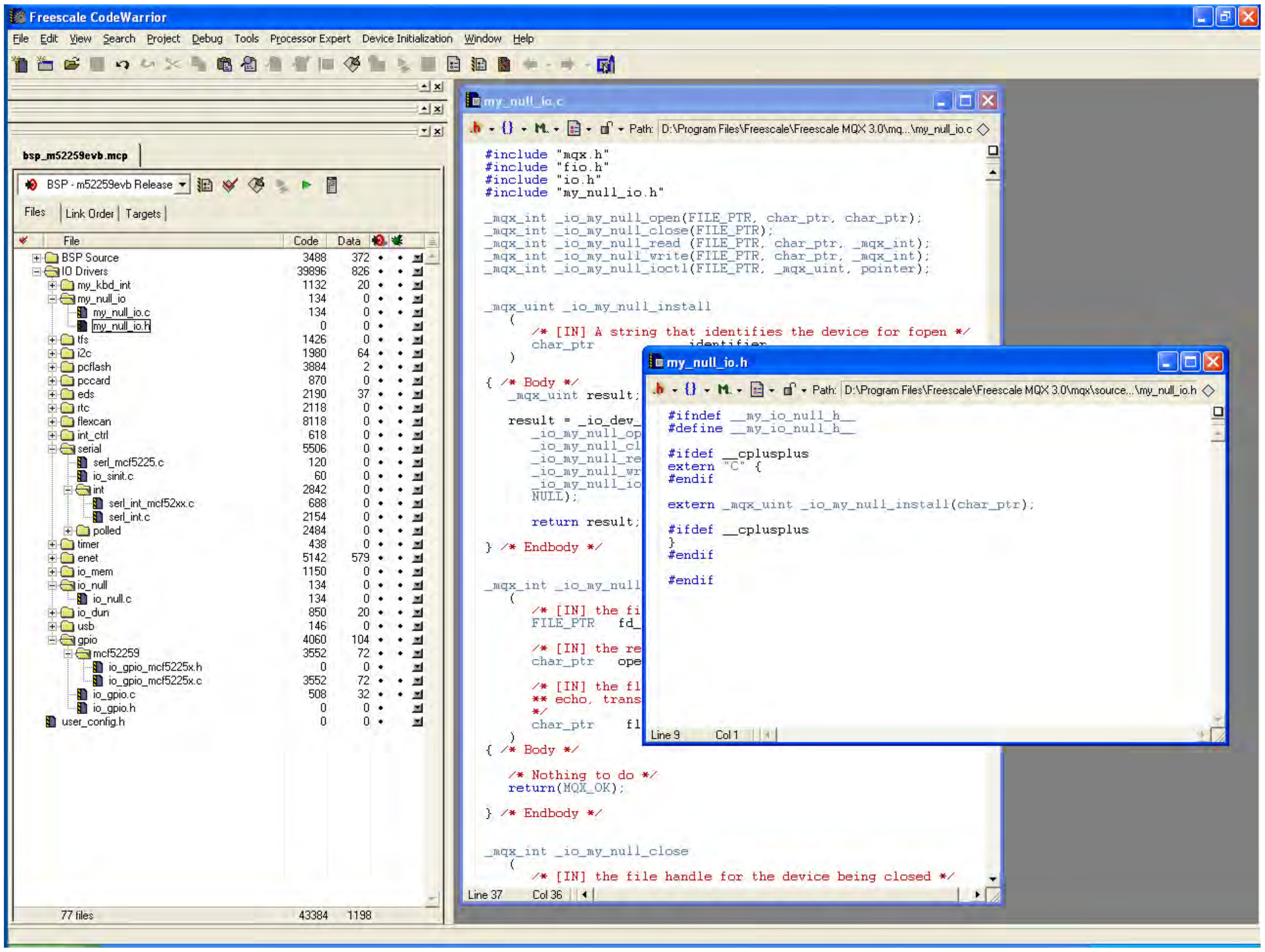


- [✓] BSP - m52259evb Release
- [✓] BSP - m52259evb Debug
- [✓] BSP - m52259evb Build all

14. The projects will execute a .bat file which among other things, copies header files to the output directory. This file is located at:
    <MQX_folder>\mqx\build\codewarrior\bsp_m52259evb.bat
    or <MQX_folder>\mqx\build\codewarrior\bsp_m52259demo.bat
    Add the following line to the appropriate file:

```
copy /Y ..\..\..\mqx\source\io\my_null_io\my_null_io.h .
```

15. Make your project.


Freescale CodeWarrior
File Edit View Search Project Debug Tools Processor Expert Device Initialization Window Help
bsp_m52259evb.mcp
BSP - m52259evb Release
Files | Link Order | Targets

File	Code	Data
BSP Source	3488	372
IO Drivers	39896	826
my_kbd_int	1132	20
my_null_io	134	0
my_null_io.c	134	0
my_null_io.h	0	0
tfs	1426	0
i2c	1980	64
pcflash	3884	2
pccard	870	0
eds	2190	37
rtc	2118	0
flexcan	8118	0
int_ctrl	618	0
serial	5506	0
serl_mcf5225.c	120	0
io_sinit.c	60	0
int	2842	0
serl_int_mcf52xx.c	688	0
serl_int.c	2154	0
polled	2484	0
timer	438	0
enet	5142	579
io_mem	1150	0
io_null	134	0
io_null.c	134	0
io_dun	850	20
usb	146	0
gpio	4060	104
mcf52259	3552	72
io_gpio_mcf5225x.h	0	0
io_gpio_mcf5225x.c	3552	72
io_gpio.c	508	32
io_gpio.h	0	0
user_config.h	0	0
77 files	43384	1198

my_null_io.c — Path: D:\Program Files\Freescale\Freescale MQX 3.0\mq...\my_null_io.c

```
#include "mqx.h"
#include "fio.h"
#include "io.h"
#include "my_null_io.h"

_mqx_int _io_my_null_open(FILE_PTR, char_ptr, char_ptr);
_mqx_int _io_my_null_close(FILE_PTR);
_mqx_int _io_my_null_read (FILE_PTR, char_ptr, _mqx_int);
_mqx_int _io_my_null_write(FILE_PTR, char_ptr, _mqx_int);
_mqx_int _io_my_null_ioctl(FILE_PTR, _mqx_uint, pointer);


_mqx_uint _io_my_null_install
   (
      /* [IN] A string that identifies the device for fopen */
      char_ptr              identifier
   )

{ /* Body */
   _mqx_uint result;

   result = _io_dev_
      _io_my_null_op
      _io_my_null_cl
      _io_my_null_re
      _io_my_null_wr
      _io_my_null_io
      NULL);

      return result;

} /* Endbody */


_mqx_int _io_my_null
   (
      /* [IN] the fi
      FILE_PTR    fd_

      /* [IN] the re
      char_ptr    ope

      /* [IN] the fl
      ** echo, trans
      */
      char_ptr     fl
   )
{ /* Body */

   /* Nothing to do */
   return(MQX_OK);

} /* Endbody */


_mqx_int _io_my_null_close
   (
      /* [IN] the file handle for the device being closed */
```

Line 37 Col 36

my_null_io.h — Path: D:\Program Files\Freescale\Freescale MQX 3.0\mqx\source...\my_null_io.h

```
#ifndef __my_io_null_h__
#define __my_io_null_h__

#ifdef __cplusplus
extern "C" {
#endif

extern _mqx_uint _io_my_null_install(char_ptr);

#ifdef __cplusplus
}
#endif

#endif
```

Line 9 Col 1

3.2 Testing the Null Driver

This device driver can be used by adding the following header to your application:

```
#include <my_null_io.h>
```

Now, you can call `_my_null_io_install` followed by fopen, fclose, write, read, and ioctl functions.

The following example installs the null driver, opens it, and attempts to write to it:

```
#include <mqx.h>
#include <bsp.h>
#include <fio.h>

#include <my_null_io.h>

#define MY_TASK 5

extern void my_task(uint_32);

TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
    {MY_TASK, my_task, 1500, 9, "null_test", MQX_AUTO_START_TASK, 0, 0},
    {0,          0,          0,    0, 0,        0,                    0, 0}
};

void my_task(uint_32 initial_data)
{
FILE_PTR null_file;
uint_8 data[10];

   if (IO_OK != _io_my_null_install("null:"))
   {
  printf("Error opening Null\n");
   }

   if (NULL == (null_file = fopen("null:", NULL )))
   {
        printf("Opening NULL device driver failed.\n");
      _mqx_exit(-1);
   }

   if (write(null_file, data, 4 ) != 4)
   {
        printf("Writing to NULL driver failed.\n");
      _mqx_exit(-1);
   }

   fclose(null_file);

 printf ("NULL driver working\n");

   _mqx_exit(0);
}
```

4 Random Number Generator Driver

At this moment we know the basics to develop an MQX I/O driver. Now we can add some functionality such as an MCU peripheral.

The next driver to develop is the random number generator, a module capable of generating 32-bit random numbers. For further information on this module, please refer to the *MCF52259 Reference Manual*, Chapter 6.

We use the same structure that was used on the null driver.

1. Add the bsp.h file, since this contains the register definitions for the MCU peripherals.

```
#include "mqx.h" /* Structures and constants used by MQX */
#include "fio.h" /* standard formatted I/O library */
#include "io.h" /* I/O subsystem interface. */
#include "bsp.h" /* Has the declaration of registers */

#include "io_rng.h"/*This driver header*/
```

2. The next step is to declare the local functions of our driver.

```
_mqx_int _io_rng_open(FILE_PTR, char_ptr, char_ptr);
_mqx_int _io_rng_close(FILE_PTR);
_mqx_int _io_rng_read (FILE_PTR, char_ptr, _mqx_int);
_mqx_int _io_rng_write(FILE_PTR, char_ptr, _mqx_int);
_mqx_int _io_rng_ioctl(FILE_PTR, _mqx_uint, pointer);
```

3. The install function has only minor differences from the null driver.

```
_mqx_int _io_rng_install
(
      /* [IN] A string that identifies the device for fopen */
      char_ptr identifier
    )
{

 _mqx_uint result;

 result = _io_dev_install(identifier,
      _io_rng_open,
      _io_rng_close,
      _io_rng_read,
      _io_rng_write,
      _io_rng_ioctl,
      NULL);

return result;
}
```

4. The open function contains the peripheral register initialization; in this we just need to set the RNG module to run mode and set the default range from 0x00000000 to 0xFFFFFFFF.

```
_mqx_int _io_rng_open
   (
      /* [IN] the file handle for the device being opened */
      FILE_PTR   fd_ptr,
      /* [IN] the remaining portion of the name of the device */
      char_ptr   open_name_ptr,

      /* [IN] the flags to be used during operation:
      ** echo, translation, xon/xoff, encoded into a pointer.
      */
      char_ptr    flags
   )
{ /* Body */

 /*Get the base address*/
VMCF5225_STRUCT_PTR reg_ptr = (VMCF5225_STRUCT_PTR)BSP_IPSBAR;
rng_range_struct.rng_max_value = 0xFFFFFFFF;
rng_range_struct.rng_min_value = 0;

/*Start the RNG*/
reg_ptr->RNG.RNGCR |= MCF5225_RNG_RNGCR_GO;

   return(IO_OK);

} /* Endbody */
```

5. The read function will receive the amount of bytes to be read and the pointer to the destination. The RNG generates 32-bit numbers, so when read is called, the amount of bytes should be a minimum of four.

```
_mqx_int _io_rng_read
   (
      /* [IN] the file handle for the device */
      FILE_PTR   fd_ptr,

      /* [IN] where the characters are to be stored */
      char_ptr   data_ptr,

      /* [IN] the number of characters to input */
      _mqx_int   num
   )
{ /* Body */

    /*Get the IPSBAR*/
  VMCF5225_STRUCT_PTR reg_ptr = (VMCF5225_STRUCT_PTR)BSP_IPSBAR;
  uint_32 status;
  uint_32 *temp_data_ptr = (unsigned long *)data_ptr;
  uint_8 total_bytes = 0;

  /*Get the RNG status*/
  status = reg_ptr->RNG.RNGSR;


  /*the output is 32-bit number (4 bytes)*/
  while(num >= MINIMUM_AMOUNT_OF_BYTES)
  {
         //Check for current data on the FIFO
         while(!(reg_ptr->RNG.RNGSR & 0x100));

         /*Place the generated number into the desire buffer*/
         *temp_data_ptr = reg_ptr->RNG.RNGOUT;
```

```
                num -= 4;
                total_bytes += 4;
                temp_data_ptr++;
        }

        /*Check that data was read*/
        if(total_bytes == 0)
        {
                return(IO_ERROR);
        }

        /*return the amount of bytes read*/
        return(total_bytes);

    } /* Endbody */
```

6. The write function will insert entropy into the module so that it can keep on generating numbers. Here the parameters are the pointer to the seed for the entropy and the number of bytes to be written.

```
_mqx_int _io_rng_write
   (
      /* [IN] the file handle for the device */
      FILE_PTR   fd_ptr,

      /* [IN] where the characters are */
      char_ptr   data_ptr,

      /* [IN] the number of characters to output */
      _mqx_int   num
   )
{ /* Body */

/*Get the IPSBAR*/
VMCF5225_STRUCT_PTR reg_ptr = (VMCF5225_STRUCT_PTR)BSP_IPSBAR;
uint_32 *temp_data_ptr = (unsigned long *)data_ptr;
uint_8 total_bytes = 0;

while(num >= MINIMUM_AMOUNT_OF_BYTES)
{

        /*Feed the entropy for the generator*/
        reg_ptr->RNG.RNGER = *temp_data_ptr;

        num -= 4;
        total_bytes += 4;
        temp_data_ptr++;
}

/*Check that data was written*/
if(total_bytes == 0)
{
        return(IO_ERROR);
}

/*return the amount of bytes written*/
return(total_bytes);
} /* Endbody */
```

7. The ioctl will be used to set the range for the numbers and to ask for a number within this range. First we create a structure with two parameters: the maximum value of the range and the minimum value.

```
typedef struct rng_struct
{
   uint_32   rng_max_value;    // The maximum value desired
   uint_32   rng_min_value;             // The minimum value desired
} RNG_STRUCT, _PTR_ RNG_STRUCT_PTR;
```

8. We need to add the ioctl type to the ioctl.h file.

```
/*
** Device types used in INCTL encoding
*/

#define IO_TYPE_MQX               0x00
#define IO_TYPE_MFS               0x01
#define IO_TYPE_FLASH           0x02
#define IO_TYPE_GPIO              0x03
#define IO_TYPE_I2C                 0x04
#define IO_TYPE_MEM             0x05
#define IO_TYPE_NVRAM         0x06
#define IO_TYPE_PCB               0x07
#define IO_TYPE_APCCARD     0x08
#define IO_TYPE_PCCARD       0x09
#define IO_TYPE_PCFLASH      0x0A
#define IO_TYPE_PIPE              0x0B
#define IO_TYPE_QSPI              0x0C
#define IO_TYPE_SERIAL          0x0D
#define IO_TYPE_SPI                 0x0E
#define IO_TYPE_USBMFS        0x0F
#define IO_TYPE_TFS                0x10
#define IO_TYPE_RNG 0x11
```

9. After adding the type, now in the io_rng.h file we will add the specific commands for the RNG driver.

```
/*
** IOCTL calls specific to RNG
*/

#define IO_IOCTL_RNG_SET_MAX_VALUE                               _IO(IO_TYPE_RNG,0x01)
#define IO_IOCTL_RNG_SET_MIN_VALUE                          _IO(IO_TYPE_RNG,0x02)
#define IO_IOCTL_RNG_NUMBER_IN_RANGE_IO(IO_TYPE_RNG,0x03)
```

10. The next step is to set the ioctl function. We need to add some validations to the commands, such as the maximum being greater than the minimum. To determine the number within the range we will interpolate the number generated by the module.

```
_mqx_int _io_rng_ioctl
   (
      /* [IN] the file handle for the device */
      FILE_PTR   fd_ptr,
      /* [IN] the ioctl command */
      _mqx_uint cmd,

      /* [IN] the ioctl parameters */
      pointer    param_ptr
   )
{ /* Body */

uint_8 status = MQX_OK;
    uint_32_ptr temp_param_ptr = (uint_32_ptr)param_ptr;
    uint_64 current_rng_number = 0;
    uint_64 range = 0;
    uint_64 temp_value;
    uint_64 number_interpolated;
VMCF5225_STRUCT_PTR reg_ptr = (VMCF5225_STRUCT_PTR)BSP_IPSBAR;

switch (cmd)
{
  case IO_IOCTL_RNG_SET_MAX_VALUE:
         /*Verify first that the new Max is greater than the current Min*/
         if(*temp_param_ptr >= rng_range_struct.rng_min_value)
         {
         rng_range_struct.rng_max_value = *temp_param_ptr;
         }
         else
         {
         status = MQX_INVALID_PARAMETER;
         }
     break;

  case IO_IOCTL_RNG_SET_MIN_VALUE:
         /*Verify first that the new Min is less than the current Max*/
         if(*temp_param_ptr <= rng_range_struct.rng_max_value)
         {
         rng_range_struct.rng_min_value = *temp_param_ptr;
         }
         else
         {
         status = MQX_INVALID_PARAMETER;
         }

     break;


  case IO_IOCTL_RNG_NUMBER_IN_RANGE:

         current_rng_number = reg_ptr->RNG.RNGOUT;

         //In order to get the random number between the range, we need to interpolate
         // d = d1 + ((g - g1)/(g2-g1))(d2-d1)
         //Where:
         // d2 = the max_value
         // d1 = the min_value
         // g  = RNG output
         // g2 = max range of the RNG which is 0xFFFFFFFF
         // g1 = min range of the RNG which is 0x00000000
         //When we use our values we get:
         // d = min_value + ((RNG output - 0x0) / (0xFFFFFFFF - 0x0))(max_value -
min_value)
         range = rng_range_struct.rng_max_value - rng_range_struct.rng_min_value;
         temp_value = current_rng_number * range;
         number_interpolated = temp_value / 0xFFFFFFFF;
         number_interpolated += rng_range_struct.rng_min_value;
         *temp_param_ptr = (unsigned long)number_interpolated;
     break;
  default:
         status = IO_ERROR_INVALID_IOCTL_CMD;
} /* Endswitch */


   return (status);
} /* Endbody */
```

11. The install function prototype is placed in the io_rng.h file.

```
#ifndef __RNG_DRIVER__
#define __RNG_DRIVER__


#include "ioctl.h"

typedef struct rng_struct
{
   uint_32   rng_max_value;                          // The maximum value desired
   uint_32   rng_min_value;         // The minimum value desired
} RNG_STRUCT, _PTR_ RNG_STRUCT_PTR;

typedef volatile struct rng_struct _PTR_ VRNG_STRUCT_PTR;

#define MINIMUM_AMOUNT_OF_BYTES(4)

/*
** IOCTL calls specific to RNG
*/

#define IO_IOCTL_RNG_SET_MAX_VALUE          _IO(IO_TYPE_RNG,0x01)
#define IO_IOCTL_RNG_SET_MIN_VALUE  _IO(IO_TYPE_RNG,0x02)
#define IO_IOCTL_RNG_SET_NUMBER_SIZE  _IO(IO_TYPE_RNG,0x03)


#ifdef __cplusplus
extern "C" {
#endif


extern _mqx_int _io_rng_install (char_ptr identifier);

#ifdef __cplusplus
}
#endif


#endif /*__RNG_DRIVER__*/
```

4.1 Testing the RNG Driver

The RNG driver can be used in the application by adding the next header file:

```
#include <io_rng.h>
```

After you install the driver, the access to it is made by the fopen, write, read, and ioctl functions.

The following example installs the io_rng driver, seeds the generator, sets the range, and reads three values from the module and one from the range.

```
void hello_task(uint_32 initial_data)
{
FILE_PTR fpmyRNG;
unsigned long random_number;
unsigned long feed_generator[3] = {198,0,125};
unsigned long max_value = 122;
unsigned long min_value = 87;
unsigned long range_random;
unsigned char error;
unsigned char u8Cycle = 8;

printf("\n\rRNG Driver Test\n\r");

/*Install the RNG driver*/
if(_io_rng_install("myrng:") != MQX_OK)
{
         printf("installing RNG driver failed");
}
/*Open the driver so we can work with it*/
fpmyRNG = fopen("myrng:", NULL);
if(fpmyRNG == NULL)
{
         printf("Opening RNG driver failed");
}
/*Write on the driver, this will insert entropy to the RNG*/
error = write(fpmyRNG, (uint_8 *)&feed_generator[0], 12);
if(error == IO_ERROR)
{
         printf("Writing to RNG driver failed");
}
/*Set Max value and bit size*/
error = ioctl(fpmyRNG, IO_IOCTL_RNG_SET_MAX_VALUE, &max_value);
if(error != MQX_OK)
{
         printf("IO Control Command write failed.");
}
error = ioctl(fpmyRNG, IO_IOCTL_RNG_SET_MIN_VALUE, &min_value);
if(error != MQX_OK)
{
         printf("IO Control Command write failed.");
}
printf("\n\rPress ANY key to get a new set of numbers");

for(;;)
{
  getchar();
  /*Read the driver, this will deliver a random number*/
```

```
        /*The byte count is 8, so we will receive 2 random generated
              numbers*/
        error = read(fpmyRNG, (uint_8 *)&random_number[0], 12);
        if(error != 12)
        {
          printf("IO Control Command write failed.");
        }
        printf("\n\rThe new number generated was %d",random_number[0]);
              printf("\n\rThe new number generated was %d",random_number[1]);
            printf("\n\rThe new number generated was %d",random_number[2]);    error =
      ioctl(fpmyRNG,
              IO_IOCTL_RNG_NUMBER_IN_RANGE,&range_random);
        if(error != MQX_OK)
        {
               printf("IO Control Command write failed.");
        }
        printf("\n\rThe new number generated was %d",range_random);
      }
        _mqx_exit(0);
      }
```

5 Keypad Driver — Use of Interrupts and Queue

Now that we have a basic understanding of an MQX I/O driver, we can add more interaction with the hardware and use interrupts.

This section will guide you through the process of creating a keypad driver using the MCF5225x EPORT module. For further information on this module, please refer to the *MCF52259 Reference Manual*, Chapter 17.

The M52259DEMO board has two buttons labeled SW1 and SW2, connected to IRQ5 and IRQ1 respectively. The following piece of the schematic shows the connection to these buttons:

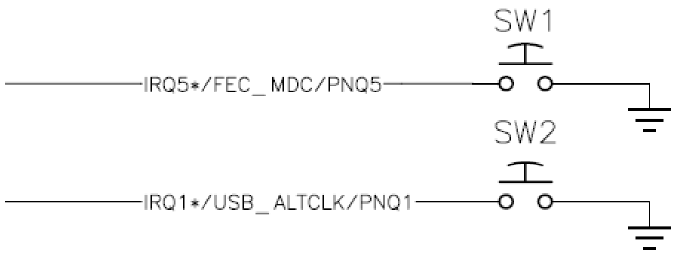


Figure 1. Part of M52259DEMO Schematic Showing Connection to SW1 and SW2

The M52259EVB has these two pins available at the MCU_PORT connector; external circuitry is needed to test this example on M52259EVB.

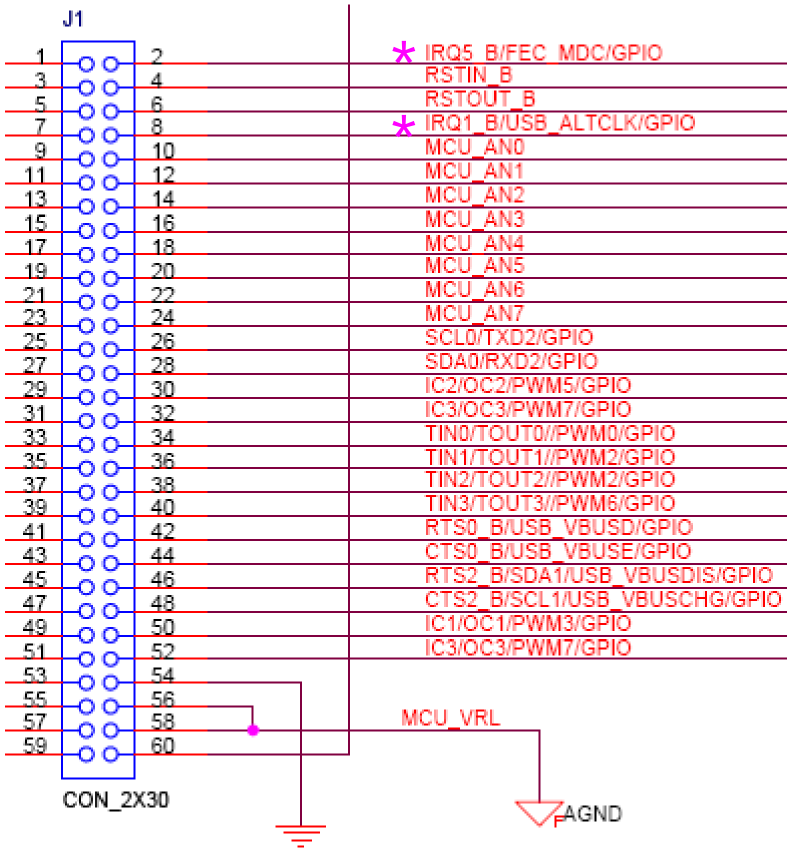


Figure 2. Part of M52259EVB Schematic Showing Location of IRQ1 and IRQ5

Follow the same steps used for the null driver example to create the basic skeleton of this driver, which we will call my_kbd_int instead of my_null_io.

1. This driver uses a character queue as defined by MQX and makes direct accesses to registers, so we need to include the following files in my_kbd_int.c.

```
#include "mqx.h"   /* Structures and constants used by MQX */
#include "fio.h" /* standard formatted I/O library */
#include "io.h" /* I/O subsystem interface. */
#include "charq.h" /* Needed for the char queue */
#include "bsp.h" /* Has the declaration of registers */
#include "my_kbd_int.h" /*This is the header for this driver */
```

2. Declare some structures and global variables needed by this driver.

```
typedef struct io_kbd_int_struct
{
   /* The queue size to use */
   _mqx_uint           QUEUE_SIZE;
   /* The input queue */
   CHARQ_STRUCT_PTR    IN_QUEUE;
   /* Count number of initializations */
   _mqx_uint                                                  COUNTER;
} IO_KBD_INT_DEVICE_STRUCT, _PTR_ IO_KBD_INT_DEVICE_STRUCT_PTR;

// Structure used to save and restore ISR pointers
typedef struct
```

```
{
pointer OLD_ISR_DATA;
void (_CODE_PTR_ OLD_ISR)(pointer);
} MY_ISR_STRUCT, _PTR_ MY_ISR_STRUCT_PTR;

// Used to save/restore ISRs for IRQ1 and IRQ5
MY_ISR_STRUCT_PTR isr_ptr[2];

// Pointers to the registers for GPIO and EPORT modules
VMCF5225_GPIO_STRUCT_PTR mcf5225_gpio_ptr;
VMCF5225_EPORT_STRUCT_PTR mcf5225_eport_ptr;
```

3. MQX defines the MCF5225x registers in the following file:

   <MQX_folder>\mqx\source\psp\coldfire\mcf5225.h

   The definitions have the following form:

```
typedef struct mcf5225_module_struct
{
   ...
   /* Definition of all registers */
   ...
} MCF5225_module_STRUCT, _PTR_ MCF5225_module_STRUCT_PTR;
typedef volatile struct mcf5225_module_struct _PTR_ VMCF5225_module_STRUCT_PTR;
```

4. Add the prototypes for all your private functions. Note the new definition of the interrupt service routine:

```
_mqx_int _io_my_kbd_int_open(FILE_PTR, char_ptr, char_ptr);
_mqx_int _io_my_kbd_int_close(FILE_PTR);
_mqx_int _io_my_kbd_int_read (FILE_PTR, char_ptr, _mqx_int);
_mqx_int _io_my_kbd_int_write(FILE_PTR, char_ptr, _mqx_int);
_mqx_int _io_my_kbd_int_ioctl(FILE_PTR, _mqx_uint, pointer);
void    _my_kbd_int_isr(pointer);
```

5. The installer of this device driver takes the `queue_size` which will be used to create the queue as an additional parameter. Note the initialization of the structure of the keyboard structure and the pointers to the module registers.

   It is also important to notice the use of the I/O initialization data which will be passed to the device functions.

```
_mqx_uint _io_my_kbd_int_install
   (
      /* [IN] A string that identifies the device for fopen */
      char_ptr  identifier,
      /* [IN] The I/O queue size to use */
      uint_32  queue_size
   )

{
    _mqx_uint result;
   VMCF5225_STRUCT_PTR  mcf5225_ptr;
  IO_KBD_INT_DEVICE_STRUCT_PTR    kbd_ptr;

     // Initialize my KBD structure
    kbd_ptr = _mem_alloc_system_zero(sizeof(IO_KBD_INT_DEVICE_STRUCT));
    if (kbd_ptr == NULL)
    {
    return MQX_OUT_OF_MEMORY;
    }
    kbd_ptr->QUEUE_SIZE = queue_size;
    kbd_ptr->COUNTER =0;

    // Get the register structure based on IPSBAR
    mcf5225_ptr = _PSP_GET_IPSBAR();

    // Get the base of GPIO and EPORT registers
    mcf5225_gpio_ptr = &mcf5225_ptr->GPIO;
    mcf5225_eport_ptr = &mcf5225_ptr->EPORT[0];

   // Install the I/O, note kbd_ptr passed as an I/O init data
   result = _io_dev_install(identifier,
      _my_kbd_int_open,
      _my_kbd_int_close,
      _my_kbd_int_read,
      _my_kbd_int_write,
      _my_kbd_int_ioctl,
      kbd_ptr);

      return result;

}
```

6. The open function initializes IRQ1 and IRQ5 to be used as EPORT, detecting a falling edge and generating an interrupt. It also starts the character queue and the interrupts. Note how the previous interrupt service routines are saved and how both the IRQ1 and IRQ5 share the same ISR.

   This open routine uses the `COUNTER` member in the keyboard structure to prevent unwanted re-initializations of the device.

```
_mqx_int _io_my_kbd_int_open ( FILE_PTR   fd_ptr, char_ptr open_name_ptr,
                                                              char_ptr    flags )
{
   _mqx_int result;
   IO_DEVICE_STRUCT_PTR               io_dev_ptr;
   IO_KBD_INT_DEVICE_STRUCT_PTR    kbd_struct;

   // Get the kbd info passed as a parameter from the system
   io_dev_ptr     = (IO_DEVICE_STRUCT_PTR)fd_ptr->DEV_PTR;
   kbd_struct = (pointer)(io_dev_ptr->DRIVER_INIT_PTR);

   //Check if the device was opened before
   if (kbd_struct->COUNTER == 0)
   {
// Initialize pins
mcf5225_gpio_ptr->PNQPAR &= ~0x0C0C; // Set IRQ1/PNQ1, IRQ5/PNQ5
mcf5225_gpio_ptr->PNQPAR |= 0x0404;  //    as EPORT

// Set IRQ1/IRQ5 to detect falling edges
mcf5225_eport_ptr->EPPAR = MCF5225_EPORT_EPPAR_EPPA1_FALLING |
                                              MCF5225_EPORT_EPPAR_EPPA5_FALLING;

// Allocate space for the queue
kbd_struct->IN_QUEUE = _mem_alloc_system(sizeof(CHARQ_STRUCT) -
                          (4 * sizeof(char)) + kbd_struct->QUEUE_SIZE);
if (kbd_struct->IN_QUEUE == NULL){
```

```
            return(MQX_OUT_OF_MEMORY);
    }

    // Initialize the queue
    _mem_set_type(kbd_struct->IN_QUEUE,MEM_TYPE_IO_SERIAL_IN_QUEUE);
          _CHARQ_INIT(kbd_struct->IN_QUEUE, kbd_struct->QUEUE_SIZE);

    // Enable IRQ1/IRQ5 Interrupts
    mcf5225_eport_ptr->EPIER = MCF5225_EPORT_EPIER_EPIE1 |
                                                          MCF5225_EPORT_EPIER_EPIE5;

    // Initialize IRQ1 ISR and backup the previous isr pointer
    isr_ptr[0] = _mem_alloc_zero(sizeof(MY_ISR_STRUCT));
    isr_ptr[0]->OLD_ISR_DATA =
        _int_get_isr_data(MCF5225_INT_EPORT0_EPF1);
    isr_ptr[0]->OLD_ISR = _int_get_isr(MCF5225_INT_EPORT0_EPF1);
    _int_install_isr(MCF5225_INT_EPORT0_EPF1, _my_kbd_int_isr,
    kbd_struct);

    // Initialize IRQ5 ISR and backup the previous isr pointer
    isr_ptr[1] = _mem_alloc_zero(sizeof(MY_ISR_STRUCT));
    isr_ptr[1]->OLD_ISR_DATA =
      _int_get_isr_data(MCF5225_INT_EPORT0_EPF5);
    isr_ptr[1]->OLD_ISR = _int_get_isr(MCF5225_INT_EPORT0_EPF5);
    _int_install_isr(MCF5225_INT_EPORT0_EPF5, _my_kbd_int_isr,
    kbd_struct);

    // Initialize both IRQ1 and IRQ5
    result = _mcf52xx_int_init(MCF5225_INT_EPORT0_EPF1, 1, 3, TRUE);
    if (result == MQX_OK)
      result = _mcf52xx_int_init(MCF5225_INT_EPORT0_EPF5 , 5, 3 , TRUE);

    kbd_ptr->COUNTER++; // to avoid re-opening the device
    return result;
       }
       else{
    return IO_DEVICE_EXISTS;
       }
    }
```

7. When using malloc and changing interrupt service routines, it’s recommended to add cases to restore the system to its previous state. The following routine can be added before the end of the function:

```
    if (result != MQX_OK)
    {
            // On error, free all resources
            _mem_free(kbd_struct->IN_QUEUE);
            _int_install_isr(MCF5225_INT_EPORT0_EPF1, isr_ptr[0]->OLD_ISR,
                                                      isr_ptr[0]->OLD_ISR_DATA);
            _int_install_isr(MCF5225_INT_EPORT0_EPF5, isr_ptr[1]->OLD_ISR,
                                                      isr_ptr[1]->OLD_ISR_DATA);
            if (isr_ptr[0] != NULL)
                                                          _mem_free(isr_ptr[0]);
            if (isr_ptr[1] != NULL)
                                                          _mem_free(isr_ptr[1]);
    }
    kbd_ptr->COUNTER++; // to avoid re-opening the device
```

```
return result;
```

8. Following that example, the close function is:

```
_mqx_int _io_my_kbd_int_close  (  FILE_PTR   fd_ptr )
{
IO_DEVICE_STRUCT_PTR               io_dev_ptr;
IO_KBD_INT_DEVICE_STRUCT_PTR    kbd_struct;

   io_dev_ptr     = (IO_DEVICE_STRUCT_PTR)fd_ptr->DEV_PTR;
   kbd_struct = (pointer)(io_dev_ptr->DRIVER_INIT_PTR);

   // Free all resources if they were initialized
 if (kbd_struct->IN_QUEUE != NULL)
    {
    _mem_free(kbd_struct->IN_QUEUE);
    }

if (isr_ptr[0] != NULL)
{
        _int_install_isr(MCF5225_INT_EPORT0_EPF1, isr_ptr[0]->OLD_ISR,
                                                      isr_ptr[0]->OLD_ISR_DATA);
        _mem_free(isr_ptr[0]);
}

if (isr_ptr[1] != NULL)
{
        _int_install_isr(MCF5225_INT_EPORT0_EPF5, isr_ptr[1]->OLD_ISR,
                                                      isr_ptr[1]->OLD_ISR_DATA);
        _mem_free(isr_ptr[1]);
}

   return(MQX_OK);

}
```

9. The interrupt service routine that is called when an IRQ falling edge is detected will store the detected key in the queue and clear the EPORT flags. Note how the keyboard structure is passed as a parameter from the system.

```
void _my_kbd_int_isr
   (
      /* [IN] Keyboard structure passed by the system */
      pointer parameter
   )
{   // Get the EPORT flags from the register
unsigned char flags = mcf5225_eport_ptr->EPFR;
IO_KBD_INT_DEVICE_STRUCT_PTR    kbd_struct;

mcf5225_eport_ptr->EPFR = flags ;// Clear the flags

kbd_struct = (pointer)(parameter);

if (_CHARQ_NOT_FULL(kbd_struct->IN_QUEUE)) {
        // Store the keys in the queue
        _CHARQ_ENQUEUE(kbd_struct->IN_QUEUE,flags);
}
}
```

10. Finally, a read routine returns the values from the queue to the application:

```
_mqx_int _io_my_kbd_int_read  ( FILE_PTR   fd_ptr, char_ptr   data_ptr, _mqx_int   num )
{
int i =0;
IO_DEVICE_STRUCT_PTR               io_dev_ptr;
    IO_KBD_INT_DEVICE_STRUCT_PTR    kbd_struct;

io_dev_ptr     = (IO_DEVICE_STRUCT_PTR)fd_ptr->DEV_PTR;
kbd_struct = (pointer)(io_dev_ptr->DRIVER_INIT_PTR);

while (!_CHARQ_EMPTY(kbd_struct->IN_QUEUE))
{
         // If there are characters in the queue return their value
_CHARQ_DEQUEUE(kbd_struct->IN_QUEUE,data_ptr[i++]);
}

// return number of characters in the queue
return i;

}
```

11. The write and ioctl are not used in this example, but they could be implemented in a similar way as in the previous examples.
12. The my_kbd_int.h header file will show the install prototype:

```
#ifndef __my_kbd_int_h__
#define __my_kbd_int_h__

#ifdef __cplusplus
extern "C" {
#endif

extern _mqx_uint _io_my_kbd_int_install(char_ptr identifier,
 uint_32  queue_size);

#ifdef __cplusplus
}
#endif

#endif
```

5.1 Testing the Keypad Driver

This device driver can be used by adding the following header to your application:

```
#include <my_kbd_int.h>
```

Now, you can call _my_kbd_int_install followed by fopen, fclose, and read functions.

The following example installs the my_kbd_int driver, opens it, and displays the pressed keys:

```
#include <mqx.h>
#include <bsp.h>
#include <fio.h>
```

```
#include <my_kbd_int.h>

#define MY_TASK 5
extern void my_task(uint_32);

TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
    {MY_TASK, my_task, 1500, 9, "kbd", MQX_AUTO_START_TASK, 0, 0},
    {0,          0,          0,    0, 0,       0,                    0, 0}
};

void my_task(uint_32 initial_data)
{
FILE_PTR kbd_file;
uint_8 data[10];
uint_8 count, i;

   printf("\n KBD with Interrupts \n");

   if (IO_OK != _io_my_kbd_int_install("kbd:", 10)){
  printf("Error opening KBD\n");
  }


   if (NULL == (kbd_file = fopen("kbd:", NULL ))){
       printf("Opening KBD device driver failed.\n");
      _mqx_exit(-1);
   }

   for (;;)
   {
         // Get number of bytes in the queue and data
         count = read(kbd_file, data, 0 );

         if ( count != 0x00)
         {
                                                          for (i=0; i<count; i++)
                                        printf ("Key(s) pressed: %x\n", data[i]);
         }
   }

   fclose(kbd_file);
   printf ("KBD driver closed \n");
   _mqx_exit(0);
}
```

6 Driver Installation

To keep things simple, the drivers included in this application note are installed in the main application task.

I/O drivers included in MQX don't call the install function explicitly in the same way, because they access drivers that were installed during BSP initialization before the first task started.

This can be easily implemented by calling the install function in init_bsp.c, available at:

M52259EVB: <MQX_folder>\mqx\source\bsp\m52259evb\init_bsp.c

M52259DEMO: <MQX_folder>\mqx\source\bsp\m52259demo\init_bsp.c

7 Conclusion

Developing and testing I/O drivers is a common and necessary task for most applications. This application note guided you through the process of creating and testing three different device drivers with increasing levels of complexity.

The Freescale MQX RTOS includes several I/O drivers available in source code that cannot only be used in your application, but can serve as a guide for more complex implementations and reduce your development time.

For the latest MQX version and documentation, please visit: http://www.freescale.com/mqx

THIS PAGE IS INTENTIONALLY BLANK

How to Reach Us:

Home Page:
www.freescale.com

Web Support:
http://www.freescale.com/support

USA/Europe or Locations Not Listed:
Freescale Semiconductor, Inc.
Technical Information Center, EL516
2100 East Elliot Road
Tempe, Arizona 85284
+1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

Europe, Middle East, and Africa:
Freescale Halbleiter Deutschland GmbH
Technical Information Center
Schatzbogen 7
81829 Muenchen, Germany
+44 1296 380 456 (English)
+46 8 52200080 (English)
+49 89 92103 559 (German)
+33 1 69 35 48 48 (French)
www.freescale.com/support

Japan:
Freescale Semiconductor Japan Ltd.
Headquarters
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
Tokyo 153-0064
Japan
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 or 303-675-2140
Fax: 303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

Document Number: AN3902
Rev. 0
08/2009