Embedded Systems Practical

Part 1

Björn Franke, Michael O'Boyle, Harry Wagstaff
University of Edinburgh – School of Informatics

Introduction

Embedded processors are extremely prevalent in the modern world. They can be found in everything from refrigerators to cars, and range in size from tiny PICs to the complex ARM processors often found in mobile phones.

Of course, these processors must be programmed, typically using specialist tools and software, and deeply embedded systems are frequently programmable only in C or assembly. The various devices (such as analogue to digital converters, serial communication devices and display controllers) must have drivers written before they are useful, and the multiple tasks running on the processor must be scheduled – this is typically done by a RTOS (Real Time Operating System).

This coursework involves programming for a Freescale Kinetis K70 microcontroller, running under the control of MQX, a RTOS developed by Freescale. The RTOS provides scheduling and device drivers, as well as a number of libraries for network communication, graphical displays etc.

The coursework is broken into 4 milestones, the last of which is to develop a web controlled security system, supporting multiple zones and wall clock scheduling for enabling/disabling each zone. The first 3 milestones act as introductions to the tools and libraries you will need, and you should aim to finish them as soon as possible. The coursework will be done in C. Each task should be a separate project in its own directory..

Although this coursework is done in the C programming language, it is not intended specifically to teach C, so you should make sure that you are familiar with at least the basics of the C programming language.

Task 0 – Setting Up Your Environment

To compile programs for the K70 Tower, and download them into the Tower's memory, you will need to use a number of special tools. In particular, you will be using the Freescale ARM toolchain for compiling and linking programs, and OpenOCD to download them to the Tower and as a debug interface.

These programs are located in a group space, at /group/teaching/espractical/Freescale, and /group/teaching/espractical/OpenOCD. The OpenOCD directory contains a number of scripts which can be used for setting up your DICE environment to use the Freescale toolchain, and to download and debug programs running on the Tower. You should 'source' the setup script every time you start working on the coursework:

```
source /group/teaching/espractical/OpenOCD/setup.sh
```

You should also ensure that you only connect the Freescale board to your DICE machine after you have logged in, to ensure that the permissions to access the device are set up correctly. If you encounter any 'Could not connect to OSBDM device' errors when attempting to use the flash or debug scripts, disconnect and reconnect the board.

A project template is also available in /group/teaching/espractical/Template. This contains a simple program which initialises the RTOS and network capabilities of the board, as well as a Makefile which can be used to compile this project using the Freescale toolchain. If you wish, you can attempt to set up a project in your IDE of choice although you will be responsible for getting this to work.

Task 1 – LED Control

The first task is to simply light an LED in response to a button press. This will introduce you to programming using the MQX RTOS, and to the IO library used to interface with the touch buttons and LEDs on the K70 board. The first part of this task is to make the LED constantly lit. This should help familiarise you with using the RTOS, and with the flash and debug scripts. Your first step should be to make a copy of the template project and place it in an 'ES' folder somewhere in your home directory.

Part 1 – LED Constantly On

You're now ready to start adding code to the project. Open up the 'main.c' file in your new copy of the template in your favourite text editor. The first thing you need to do to control the LEDs is to initialize the IO library used for controlling the touch buttons and the LEDs. The function to do this is called _bsp_btnled_init() and it returns a HMI_CLIENT_STRUCT_PTR which is the passed to the other IO functions.

When the init function is called, it will switch on all of the LEDs on the board. However, for this part we only want one lit, so we need to switch off the other LEDs. The LEDs are controlled with the btnled_set_value function, for example:

```
btnled_set_value(hmi_client, HMI_LED_1, HMI_VALUE_ON);
```

Can be used to switch on LED 1, and:

```
btnled_set_value(hmi_client, HMI_LED_3, HMI_VALUE_OFF);
```

Can be used to switch off LED 3.

Notice that this function takes several arguments: the variable containing the HMI_CLIENT_STRUCT_PTR we created earlier, the LED we are interested in, and the value we want to assign to it (on or off). Using this function, you should be able to switch off LEDs 2, 3 and 4 and complete the first part of this task.

To compile this code, open a terminal in the folder containing your project code. First of all, 'source' the 'setup.sh' script in the group space OpenOCD folder (described above):

```
source /group/teaching/espractical/OpenOCD/setup.sh
```

Now, run 'make' to compile your project. Although you will probably receive some warning messages (relating to MQX standard library functions) you should now have a file called 'main'. This is an ELF format binary file for the ARMv7-M architecture and is ready to be downloaded onto the Tower, which is done using the flash script:

```
/group/teaching/espractical/OpenOCD/flash.sh main
```

Be patient: due to the interface used to talk to the board (JTAG) the download runs at only 2-3 KB/s so it may take a couple of minutes. Once the program has downloaded, the Tower will automatically restart and begin running your program (and the LEDs should behave appropriately). Try experimenting with turning on and off various combinations of the LEDs.

Part 2 – LED On in Response to Button Press

The second part of this task is to light the LED in response to a button press. We will actually be using capacitive touch pads around the LEDs (shown by the white rectangle around the LED on the board) rather than the push buttons. The methods for handling buttons and touch pads are exactly the same when using the IO library.

Rather than have a function which detects each button press, the IO library allows us to register callback functions which will be called when a button press is detected. This means we do not have to individually check each button we are interested in continuously. We can register callbacks for two button events: pushing the button, and releasing the button.

So, the first thing we need to do for lighting the LED in response to button presses is creating our two button callbacks – one to switch on the LED when the button is pressed, and one to switch it off when the button is released:

```
void button_push(void *ptr)
{
      btnled_set_value(hmi_client, HMI_LED_1, HMI_VALUE_ON);
}

void button_release(void *ptr)
{
      btnled_set_value(hmi_client, HMI_LED_1, HMI_VALUE_OFF);
}
```

Notice that the callback functions take a single argument (void *ptr) which can be used to pass information into the callback (such as which button has been pressed), and they return void (i.e. nothing).

We now need to register the two button callbacks. This is done using the btnled_add_clb function. This function takes several arguments: the HMI_CLIENT_STRUCT_PTR from earlier, the button we are interested in, the button event we are interested in (pressing or releasing), the callback function to register, and the argument to pass to it (we pass NULL since we are not interested in passing a value):

```
void Main_task(uint_32 initial_data)
{
      btnled_add_clb(hmi_client, HMI_BUTTON_1, HMI_VALUE_PUSH, button_push, NULL);
      btnled_add_clb(hmi_client, HMI_BUTTON_1, HMI_VALUE_RELEASE, button_release,

      NULL);
}
```

We now need to poll the library. Polling involves repeatedly asking the library if it has encountered any button presses. To do this, we repeatedly call the btnled_poll() function:

```
void Main_task(uint_32 initial_data)
{
      ...
      while(1) btnled_poll(hmi_client);
}
```

Submission

Both parts of this task should be demonstrated by 4PM, Friday, 6th February. All associated code should also be submitted by this time.

Task 2 – Web Server

The K70 microcontroller board is attached to a device board containing a serial port and an ethernet port. The MQX RTOS has built-in support for using the ethernet port, both to send and receive TCP/IP traffic and as a full HTTP server. For this part of the coursework, you will be using the HTTP server capability, as well as a serial port over USB.

Part 1 – Send Static Web Page Over HTTP

For this part of the coursework we will set up an HTTP server on the board and serve a static web page. We will need to insert the web page into a file system on the device (which we will also need to initialize), then tell the RTCS library to set up and run a web server with that page as the index page. Please note: if you are combining RTCS features with the btnled library used earlier, you will need to ensure that the call to rtcs_init() is AFTER the call to _bsp_btnled_init.

The first thing to do here is to create the web page which will be hosted. We only want a simple 'hello world' style page at the moment. The web page will be stored in an in-memory file system, and we will then instruct the HTTP server on the board to use this file system as a source of pages. In order to use the TFS (Tiny File System) and HTTPD features provided by MQX and RTCS, we will need to #include two additional files: tfs.h and httpd.h.

To populate the file system, we provide it with an array of structs representing files. For example:

```
unsigned char http_refresh_text[] = "Hello World!"
const TFS_DIR_ENTRY static_data[] = {
      { "/index.html", 0, http_refresh_text, sizeof(http_refresh_text) },
      {0,0,0,0}
};
```

Represents a simple file system containing a file called 'index.html', with no flags set, using the 'http_refresh_text' string as its contents and being 'sizeof(static_page)' bytes long. The {0,0,0,0} is an empty entry used to indicate the end of the array. Note that the HTML is not being read from an external file.

Once we have our file system specified, we need to install it. We can do this using the _io_tfs_install function. This takes two arguments: where in the file system to install, and what we are trying to install. So, to install our filesystem above, we use:

```
_io_tfs_install("tfs:", static_data);
```

We can now initialize and start our http server. To initialize the http server, we use the function httpd_server_init_af, which takes several arguments: a root directory, an index page, and an IP protocol type. We need another struct to form our root directory, which simply points to the file system we set up earlier:

```
static HTTPD_ROOT_DIR_STRUCT http_root_dir[] = { {"", "tfs:"}, { 0,0 } };
```

The http server is initialised using a function called 'httpd_server_init_af', which returns a value of type 'HTTPD_STRUCT*'.

```
http_server = httpd_server_init_af(http_root_dir, "\\index.html", AF_INET);
```

Once the http server is initialized, we need to run it:

```
httpd_server_run(http_server);
```

Much like the btnled system used in Task 1, we need to poll the ethernet device to service requests. Much like we used a btnled_poll() function for this previously, we now need to use another function, ipcfg_task_poll():

```
while(1) ipcfg_task_poll();
```

One more step before you test your web page is to configure the network settings on the board. If you look in the 'main.h' file you will see 3 #define lines containing uses of the IPADDR macro. We need to change the IP addresses in the macros to configure the board. For the top one (ENET_IPADDR), enter the IP address your pair was assigned (with commas instead of full stops). For the middle one, enter 255,255,255,0. For the last one (ENET_GATEWAY) enter 192.168.105.250.

Now build and download your project.

Submission

This task should be demonstrated by 4PM, Friday, 6th February. All associated code should also be submitted by this time.

Task 3 – Real Time Clock

One of the hardware devices on the K70 board is a RTC (Real Time Clock) module. This module keeps track of the current real world time, in seconds, since it was last reset. This task involves configuring and displaying the value of the RTC using a web server running on the K70 board.

Part 1 – CGI Request

First, we will try lighting an LED when a web page is accessed. This will demonstrate how to set up the CGI system on the board. Setting up the CGI system is very similar to setting up the standard HTTP server, except that rather than assigning text strings containing HTML to page names, you assign function callbacks. In addition, the CGI system automatically appends ".cgi" to filenames, so for example, if you use the cgi table below, to access the LED toggle function you should navigate to '192.168.105.xxx/led.cgi?some_led_number'.

So, first we should create the CGI functions we plan on calling when pages are accessed. The CGI functions can view any data attached to the page URL and send back data to the browser, allowing a CGI request to both input and output data. For example:

```
_mqx_int led_callback(HTTPD_SESSION_STRUCT *session)
{
      int led = atoi(session->request.urldata);
      httpd_sendstr(session->sock, "<html><body>LED toggled</body></html");
      btnled_toogle(hmi, HMI_GET_LED_ID(led));
      return session->request.content_len;
}
```

(Notice the function btnled_toogle – this is not a typo in this document, it is a typo in the MQX API. These kinds of errors are extremely difficult to remove once this kind of code is used in the real world!)

As you can see, the httpd_sendstr function is used to send text back to the browser. Notice the function signature and return value – to ensure that the request is handled correctly, these must be correct. Once we have our CGI function written, we need to add it to a table, similarly to how we registered static web pages:

```
static HTTPD_CGI_LINK_STRUCT http_cgi_params[] = { { "led", led_callback }, {0,0}};
```

We also need to register this table with the http server:

```
HTTPD_SET_PARAM_CGI_TBL(http_server, http_cgi_params);
```

Note that in order for the light toggling to work, the _bsp_btnled_init() call must be BEFORE the rtcs_init() call.

Part 2 – Get Value Of RTC

Now you need to setup the RTC and output its value when a CGI request is received. Setting up the RTC is simple: simply call the _rtc_init() function with a certain flag:

```
_rtc_init(RTC_INIT_FLAG_ENABLE);
```

Now, in order to get the time from the rtc, call the function _rtc_get_time(). This function does not return the time directly, but rather it puts the current time into a struct which you provide the address of:

```
RTC_TIME_STRUCT curr_time;
_rtc_get_time(&curr_time);
```

In order to format the time into a string you will need to use the sprintf() function. This function is used to format data and present it as text. For example, to display the time:

```
char buffer[32];
sprintf(buffer, "%u:%u:%u\n", hours, minutes, seconds);
```

(This is actually a rather inefficient way of formatting the time since printf functions are very expensive in both time and stack space. However, we have a fairly powerful core and a lot of memory so it's not such an issue here).

You first need to define a destination array for the formatted text. You need to make sure that it is long enough to contain the full string. Often a modified version of sprintf, snprintf, is used instead, which will only write out a specified number of characters. A sprintf call to a too-short buffer can often be used as the vector for a buffer overflow attack, which can be avoided by using snprintf. Once you have the time correctly formatted, you can send it out in response to a CGI request by using the httpd_sendstr function call:

```
httpd_sendstr(session->sock, buffer);
```

Part 3 – Set Value Of RTC

You should now create and register a new CGI callback which allows the user to specify the time. The user should be able to specify the current time on a static or CGI-provided web page, and have that time written to the RTC. Writing a new value to the RTC is quite simple: You create a struct to contain the new time, set the value of it to the new time, and then pass it to the _rtc_set_time function:

```
RTC_TIME_STRUCT the_new_time;
the_new_time.seconds = 1000;
_rtc_set_time(&the_new_time);
```

Similarly to how sprintf was used to output text, you can use sscanf:

```
int hours,minutes,seconds;
sscanf(session->request.urldata, "%u:%u:%u", &hours, &minutes, &seconds);
```

Using these functions, you should be able to allow the user to set and view the time on the real time clock using a web form.

Submission

All three parts of this task should be demonstrated by 4PM, Friday 6th February. All associated code should also be submitted by this time.

Task 4 – Security System

You should now have all of the knowledge that you need to create the full security system. You should aim to use several MQX tasks for this. Although it is not strictly necessary, if you use only a single task you will end up with some rather messy code. Details on the MQX task system can be

found in the MQX documentation.

Part 1 – The Basic Security System

The basic implementation of the security system should use the four capacitive touch sensors to represent motion sensors, and the four LEDs attached to them to represent the alarms – each sensor/LED represents one room. The two push buttons should represent buttons to enable/disable the alarm, and to hush the alarm if it is set off. When the alarm is enabled, the four LEDs should be lit, and when a sensor is triggered, the LED attached to it should flash.

Part 2 – Web Control

The user should be able to enable/disable and hush the security system via a web interface using the HTTP and CGI server on the board. In addition, the user should be able to enable/disable each sensor independently – when a sensor is enabled, the LED should be on. When the sensor is disabled, the LED should be off. If a sensor is disabled, the alarm in that room cannot be triggered. The web interface does not have to be flashy or complex, simple static pages are fine.

Part 3 – Timing Control

The user should be able to specify the time at which each alarm zone is enabled/disabled independently using a web interface. The user should also be able to set and view the current time.

Submission

The entire of Task 4 (all three parts) should be demonstrated, and have code submitted, by 4PM 13th February.

References

A lot of Freescale documentation links can be found on the ES website. Documentation can also be found in the 'Documentation' directory in the espractical group space (/group/teaching/espractical/Documentation).

The TA, Harry Wagstaff, can be contacted at H.Wagstaff@sms.ed.ac.uk or during the weekly lab sessions.