Getting started with the FRDM-KL43Z Demo Board and MCUXpresso
November 3, 2018

Purpose and Background:

This document is written to help students in EE342 Microcontrollers get up and running with the NXP / Freescale KL43Z demo boards that will be used in the course from this point forward. These microcontrollers are being used since NXP has abandoned the Coldfire family and the JM variant of the S08 family microcontrollers, which had been used in EE342 up to this year. The change to the ARM processor KL43Z also makes necessary a change in the Integrated Development Environment (IDE), from Code Warrior for Microcontrollers (supported for the S08 family used in EE247 and EGR222) to MCUXpresso, which is the "preferred" IDE for the "Kinetis" family of processors and products, of which the FRDM-KL43Z is a member. The FRDM-KL43Z board is supported by resources at www.freescale.com/FRDM-KL43Z that includes "gettinb started" materials. These materials walk the user through initial demonstrations, downloading the MCUXpresso IDE and the associated "Software Development Kit" (SDK) for the KL43Z. It also includes opening a sample "Hello World" application in the IDE and compiling that application and downloading it to the KL43Z and running it.

(See material at < https://www.nxp.com/support/developer-resources/evaluation-and-development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-kl43-kl33-kl27-kl17-and-kl13-mcus:FRDM-KL43Z?tab=In-Depth_Tab> and the "FRDM-KL43Z - Quick Start Guide " at < https://www.nxp.com/support/developer-resources/evaluation-and-development-boards/freedom-development-boards/mcu-boards/freedom-development-platform-for-kinetis-kl43-kl33-kl27-kl17-and-kl13-mcus:FRDM-KL43Z?tab=Documentation_Tab>.)

These "getting started" materials are definitely very helpful, but for our class purposes in EE342 we need some supplemental materials. It is assumed that the reader already has been through the introductory material cited and has the MCUXpresso installed and operating, and has been able to download and run the "Hello World" demonstration. What we want to do is move beyond that to include use of the "terminal" from previous project that allows the microcontroller to execute commands given over the serial port, and the ability to manipulate pins (for example, to control LEDs). That's what this document addresses. Given the radical change from the SH8 environment with a focus on manipulating registers, to the function call oriented environment for the Kinetis family using MCUXpresso, this supplemental document is believed necessary for students making the transition.

Starting with "Hello World":

Usually the best way to get started is to begin by opening a demonstration program. This document assumes you've been able to open and run "Hello World" from the SDK. Figure 1 (next page) shows the "Hello World" project as it is initially opened in MCUXpress, with the main program file "hello_world.c" opened in the editor.

The logic of "Hello World" is relatively simple. The main program main() does some initializations a "PRINTF" of "hello world" is executed, then in the main loop the program repeats characters which are types in using GETCHAR and PUTCHAR, and does that as long as the board remains connected and powered.

Even as was true for the S08SH8 programs, there is a lot going on that is not apparent from the main program file. How do those initializations get done? Where is "PRINTF"?

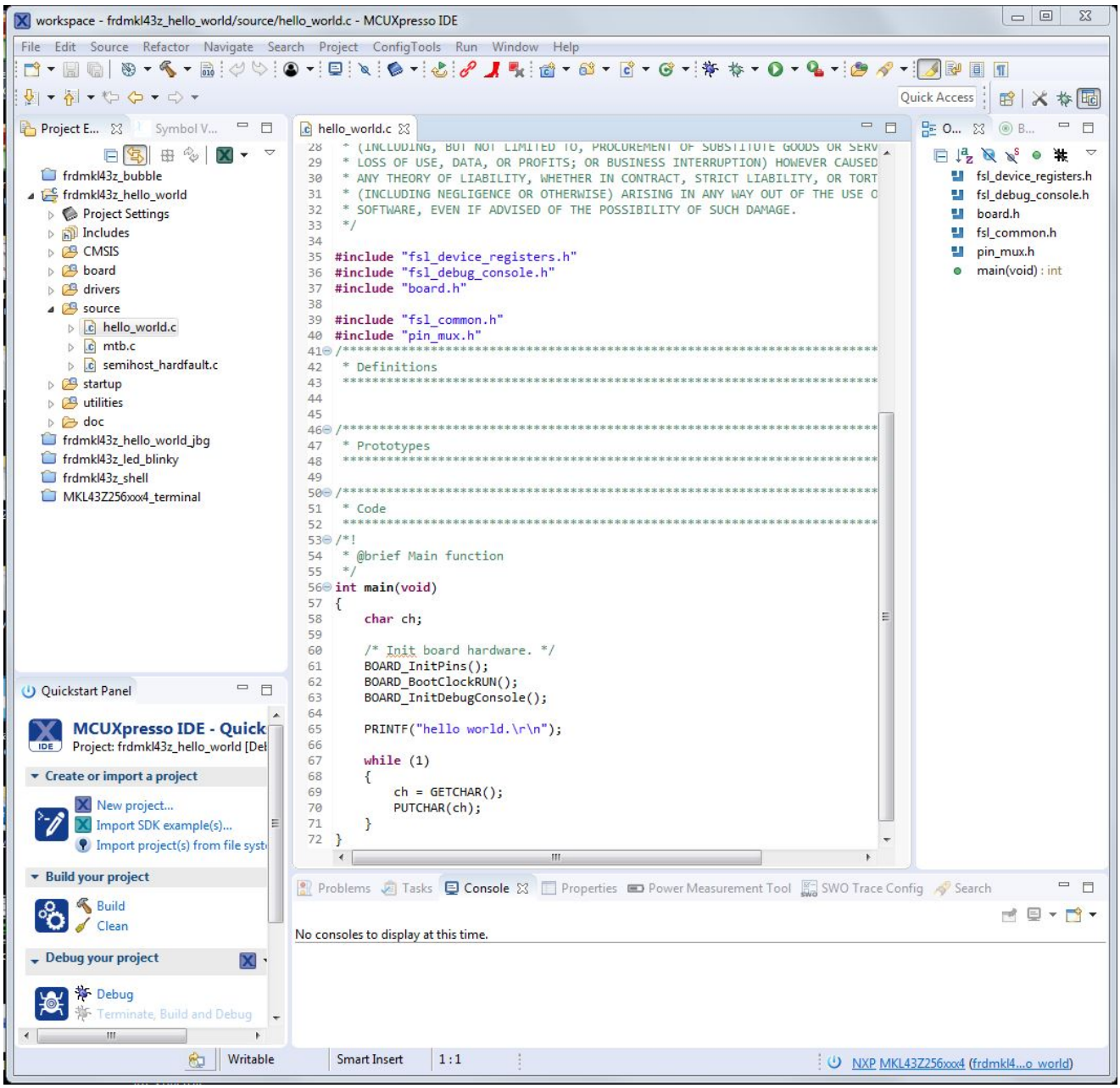

Figure 1 "Hello World" Demonstration open in MCUXpresso

Just as with the SH8 programs, there are lots of other files that contribute important code. Understanding how "Hello world" actually works, and how to modify it will require understanding how some of those things actually work. Notice that the project includes lots of "folders" in the project view. This project view is NOT the same as the file organization viewed under Windows. (To see where files actually are, open a window and find your workspace, then the project folder.) Several files contain additional source code that is of interest:

1) "board" includes initialization and function code to manipulate stuff on the KL43Z board.
2) "drivers" is the code for various capabilities of the KL43Z, for example the RTC timer
3) "utilities" contains code for utilities like the debug interface (including PRINTF)
4) "startup" includes code that does processor initialization: stuff before main() is called

Figure 2 shows some of these folders opened up to show the files within. (The drivers folder contains lots of files as well, more than could be included in the figure.) Some of these are particularly important to how "Hello world" does its business.

Notice in the main program that there are #include" statements that reference some of these files. These are:

#include "fsl_device_registers.h"
#include "fsl_debug_console.h"
#include "board.h"
#include "fsl_common.h"
#include "pin_mux.h"

Of these, "fsl_device_registers.h" is the file that (using includes itself) delivers the memory map, locations of registers, and structures used to access the device hardware, to the compiler. It is like the "…SH8.h" file under CodeWarrior that defines all the registers so you can reference them from your code. The "pinmux.h" include initialized the pins that the user will use. It doesn't need to do much for "hello_world" but it does define which UART is to be used for the debugging interface. Notice that these definitions are made in all capital letters using "#define" compiler directives. When you see something in all caps, you need to think, "This is defined somewhere; I wonder where? How is it defined?"

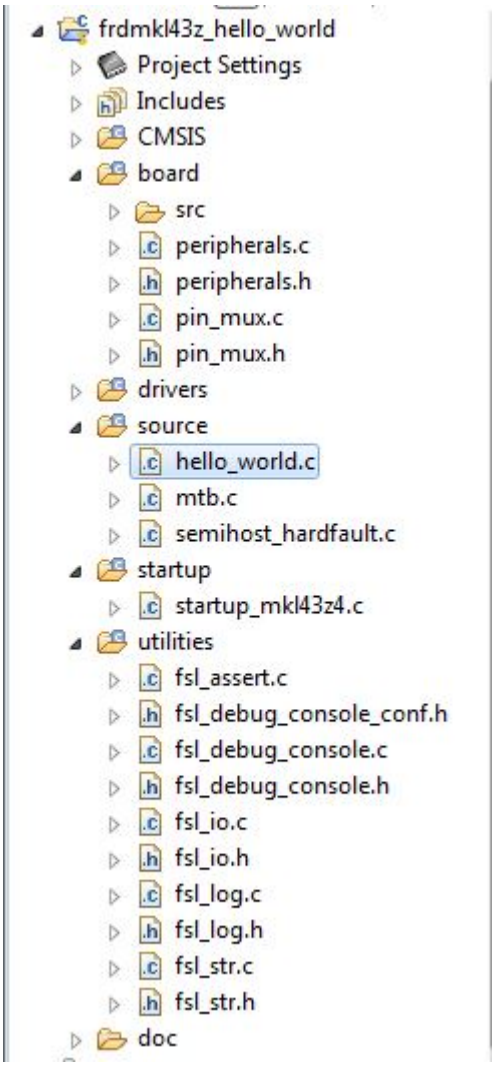

Figure 2 Files in Project View

For the "Hello world" demo, the "fsl_debug_console.h" file is particularly important. If you run "hello_world" you will see why. When I first ran this demonstration, I did not think it was working. I had found the appropriate COM port (COM7 on my computer) using the Windows system/device manager control panel. I had started the PuTTY terminal emulator configured for 115200bps (as found in the hello_world documentation file "readme.txt" in the "doc" folder). But, when (in the debugger) I started the program, nothing seemed to happen. PuTTY did not show the "hello world" message and would not repeat characters. (It had worked fine in the KDS IDE version of the "Hello world" demonstration.) What was wrong?

The answer is, it was working fine. Instead of being connected to PuTTY, though, the COM7 port was connected to the MCUXpress IDE, and was displayed in a panel in the IDE. I just had not noticed it! Figure 3 shows the "P&E Semihosting Console" panel in MCUXpresso. Sure enough, "hello world" appears there when the KL43Z code is started in the debug window. You can even type in characters, and when you hit return, those characters are repeated, and appear on the screen. However, if MCUXpress is not open and operating, PuTTY doesn't take over. You can plug the board in and reset it and run, and "hello world" never appears. It is only ever visible from within MCUXpresso.

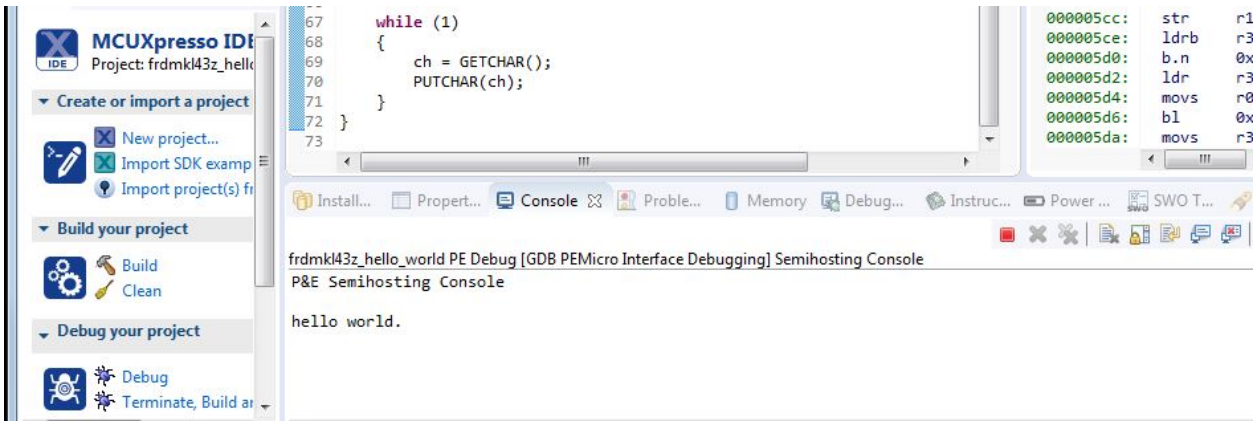

Figure 3 P&E Semihosting Console pane within MCUXpresso

Connecting to PuTTY:

So, the first modification of interest we want to make to the "Hello world" demonstration program is to connect it to PuTTY instead of the P&E Semihosting Console. The secret is inside that "fsl_debug_console.h" file. Open it, and you will find the fragment of code shown in Figure 4 below.

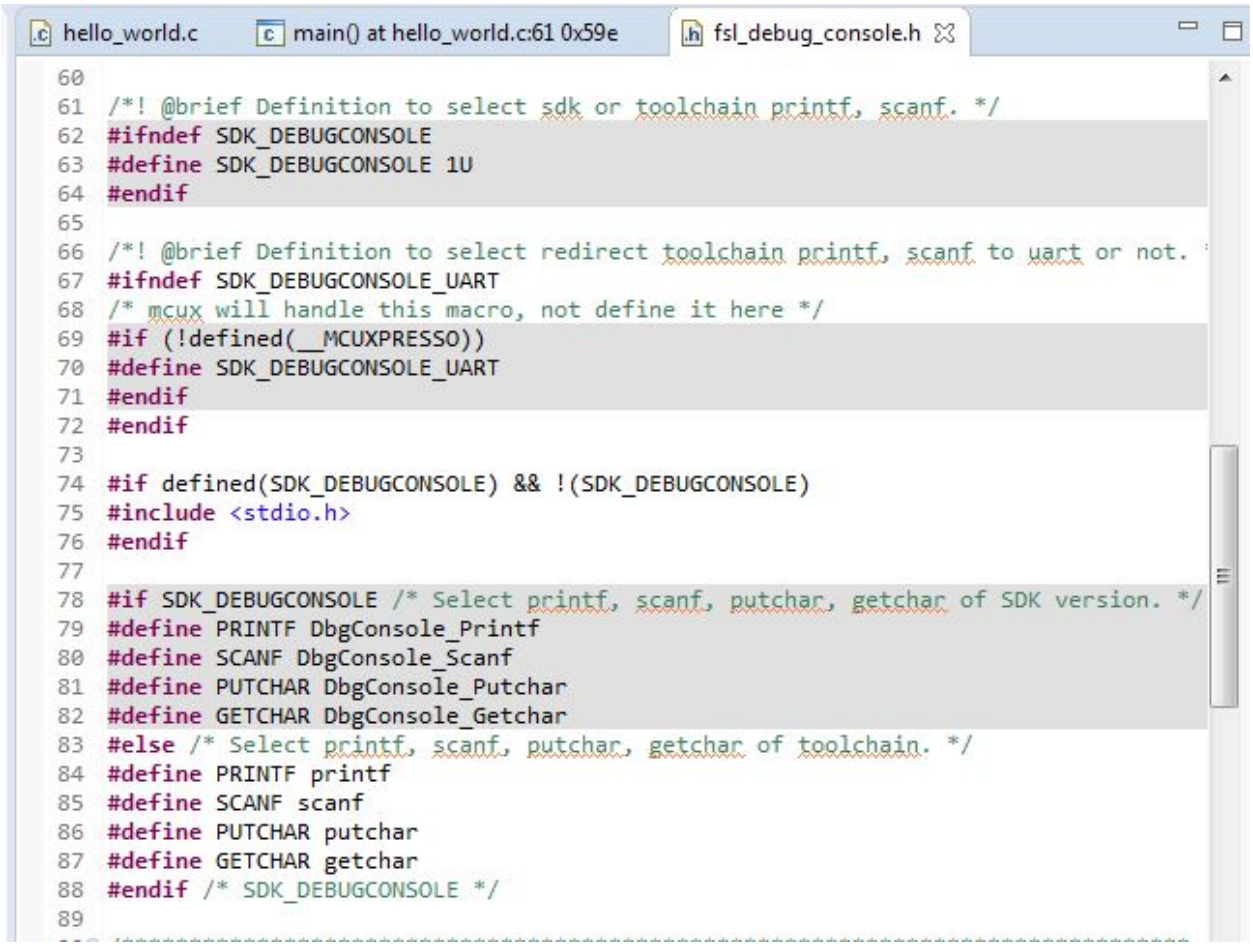

Figure 4 File "fsl_debug_console.h" fragment controlling debug interface

The line in “fsl_debug_console.h” that are greyed-out are code that is not operative due to compiler directives that depend on “#if”, “#else” directives. Note at the beginning of this fragment the code fragment:

```
/*! @brief Definition to select sdk or toolchain printf, scanf. */
#ifndef SDK_DEBUGCONSOLE
#define SDK_DEBUGCONSOLE 1U
#endif
```

What this does is check the definition of “SDK_DEBUGCONSOLE”. Apparently it has been defined somewhere, because this code is greyed out. (I have not yet figured out exactly where.) But if it is NOT defined, the compiler is directed to set it to the value of 1U (1 unsigned). Now, look further down. This definition of SDK_DEBUGCONSOLE controls what functions are used for PRINTF, CETCHAR, and PUTCHAR. If SDK_DEBUGCONSOLE is “true” (which it would be if given the value of 1U) then the functions used are those with the “DbgConsole_” prefix. If SDK_DEBUGCONSOLE is not “true” (that is, it has the value 0) then the PRINTF and other functions are simply “printf”, “scanf” etc.

```
#if SDK_DEBUGCONSOLE /* Select printf, scanf, putchar, getchar of SDK version. */
#define PRINTF DbgConsole_Printf
#define SCANF DbgConsole_Scanf
#define PUTCHAR DbgConsole_Putchar
#define GETCHAR DbgConsole_Getchar
#else /* Select printf, scanf, putchar, getchar of toolchain. */
#define PRINTF printf
#define SCANF scanf
#define PUTCHAR putchar
#define GETCHAR getchar
#endif /* SDK_DEBUGCONSOLE */
```

We can see that the ones selected here (not greyed out) are the second set. So, where “PRINTF” occurs in “hello_world.c”, the function “printf” will be substituted. This is apparently the version of printf that displays in the MCUXpresso P&E Semihosting Console. And, similarly for the other functions. Maybe we want the other set instead. In fact, that’s exactly what we need. (It turns out that you can select which way you want these to be directed when a project is created. Choose “uart” when creating a new project if you want to use PuTTY.)

To fix the problem, I edited “fsl_debug_console.h” as follows to define SDK_DEBUGCONSOLE to be true by commenting out the #if and #endif compiler directives at lines 62 and 64:

```
//#ifndef SDK_DEBUGCONSOLE commented out JBG-11-3-18
#define SDK_DEBUGCONSOLE 1U
//#endif commented out JBG-11-3-18
```

Now the other set of PRINTF etc. functions is to be used. As it turns out, this redirects the debug I/O so that it is accessible to PuTTY. Recompiling and starting a new debug run, not "hello world" appears in PuTTY as expected, and PuTTY repeats subsequent characters you may be pleased to type in, as seen in Figure 5.

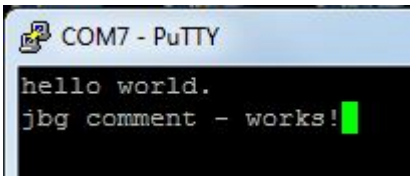

Figure 5 PuTTY works!

This issue and its resolution are a good example of how compiler directives are used to control the configuration of a particular build. Generic code can be used and adapted to different circumstances by defining or not defining constants in compiler directives, then using #if and associated compiler directives to control what is included from each file. We will find that quite often large portions of files are inapplicable to our projects.

Another point is that we are usually NOT supposed to go in and directly modify these utility and driver files. Well, I did it anyway. But it is not good practice. Instead, you should find where the applicable constant, in this case SDK_DEBUGCONSOLE, was first defined, and fix it there. Somewhere in the project definition files this constant is defined as 0, but I couldn't find it. So, I did what was expedient. Working beats not working.

Some files, however, we are expected to go in and modify for the particular project. Those are the ones in the "board" folder, and of course those in the "source" folder.

Adding the terminal:

Now that PuTTY is working, we'd like to have the "terminal;" program as well. There is a demo, "shell", that includes a capability very similar to the terminal program that we are used to. It is much more complex and capable, but also very much more difficult to fully understand. In the interest of simplicity and familiarity, I believe we are better off using the "terminal" from the S08 demonstrations. The issue is how to add it in to this "Hello world" project.

The first issue is how to add the files to the project. I have had problems when trying to move files from outside into an MCUXpresso project. I'm sure there's a correct way to do it, but I'm not yet confident in doing so. What I resorted to was creating new files and just copying the contents in. The three files we need are:

hcc_types.h
hcc_terminal.h
hcc_terminal.c

New files can be created for the project under the Filemenu, selecting "New" and then selecting "other". What you want is a new C header or source file. You can pick a template but it won't matter, since you want to replace all of the contents of the file with content from your original file of the same name. Do that for all three of these files. We can use "hcc_types.h" and "hcc_terminal.h" just as they are, but some minor modifications are needed for "hcc_terminal.c". After creating these files and copying and pasting, they should all appear in the "sources" folder of the project view.

The needed modifications to hcc_terminal.c are due to differences between the putc and gets functions. Notice that in “hello_world” and “fsl_debug_console.h there is no function “kbhit(void)”. In the SH8 terminal program, the kbhit() function is used to tell whether a character has come in. If no character has come in, getc() returns a null to say that no character was received. That’s a “non-blocking” version of getc, in that the code continues to execute regardless of whether getc actually has a character that came in.

By default, the “getc” function for the KL43Z is “blocking”. There is nothing that corresponds to kbhit. When getc is called, execution does not continue; it waits until a character is received. So, your code “hangs” and does not continue. You can see that behavior in the original “hello world” program. In the main loop, execution stops in GETCHAR() inside a busy-wait loop a couple of layers of function calls down. If you put a breakpoint at GETCHAR, you will encounter it only once per character that comes in, so that means a full circuit of the main loop only occurs once per character, because of the “blocking” operation of GETCHAR().

PUTCHAR() is also a little different. For the SH8 code used for the UART, putchar returns the character that was sent if the send was successful. For the KL43Z version we are using, PUTCHAR returns a 1 (true). The terminal_process function will hang because it will keep attempting to send out the character until it sees that same character, so it will hang in the while loop forever.

The following fragment of terminal_process corrects both of these problems. The call to kbhit is replaced by an if(1) which means that the call to terminal_process (from within the main loop) will always go to the c=(char)(*getch)(); call. And, execution will hang here until a character comes in, since the getch() function passed in terminal_init is the “blocking” version of GETCHAR() passed in from main. We will only expect terminal_process to pick up one character at a time. When we do get in a character, we need to echo it back out. So the while to keep trying until it goes out is replaced with a simple call to putch which places the returned value in a new variable cc, that we will ignore. We need to do the same thing for receiving a ‘\r’ character, which is when terminal_process() dispatches a call to the command callback function in the code that follows this fragment.

```
void terminal_process(void)
{
  char c,cc;    /* moved up one line */
  //while(c=(*kbhit)())
  if(1)
  {
    c=(char)(*getch)();
    /*while(c!=(char)(*putch)(c));*/
    /* Replace with just a simple call to putch()*/
    cc=(char)(*putch)(c);
    if (c=='\r')
    {
      cc=(char)(*putch)('\n');
    }
```

Now, the terminal should be ready to use. The remaining code modifications are made to “hello_world.c” file where the main program main() is found.

First, #include statements are needed to get access to the terminal functions:

```
#include “hcc_types.h”
#include "hcc_terminal.h"
```

Inside the main program, a call to init_terminal() is needed to initialize the terminal. Now, this presents a problem. Init_terminal() is expecting to pass pointers to the functions that (inside hcc_terminal.c) will be known as getch, putch, and kbhit. But, as explained earlier, there is no “kbhit” function in the KL43Z environment. What do we do? As modified, “hcc_terminal” no longer uses kbhit(); it was used in terminal_process() to see if a character came in, but we’ve converted terminal_process so it no longer needs kbhit(). So, we could change terminal_init() so that we pass onbly getch and putchc, omitting kbhit(). That involves making further changes to hcc_terminal, and every time we mess with it there’s a chance of breaking something. So, what I have done instead is make up a simple function for kbhit that does nothing except return true, and pass a pointer to that instead. That makes terminal_init() happy, and the vacuous kbhit() function never gets used for anything. So, in “hello_world.c” we find (before main()):

```
//New JBG dummy kbhit to initiate call to blocking PUTCH
int kbhit(void){ return 1;}
```

And inside main() after other initializations we have:

```
  //terminal_init(TERMIO_PutChar, TERMIO_GetChar, TERMIO_kbhit); original
   terminal_init(PUTCHAR, GETCHAR, kbhit);
```

The commented-out original form used for the SH8 is left in the code for reference. We are passing in the PUTCHAR and GETCHAR that we saw defined in “fsl_debug_console.h” earlier, so that means the terminal should be communicating through PuTTY (rather than the P&E Semihosting Console in MCUXpresso).

We ought to add some command (beyond the help command), so we’ll start out with a “yes” command that simply types out “Yes” whenever we give the command. (If it will help your ego, you could have it type out, “Yes, O Master!” instead. You could also have the error messasge type out, “Please repeat, for my limited understanding did not comprehend you order, O Master.” This will chew up FLASH, but the KL43Z is big.)

```
static void cmd_sayyes(char *param)
{
  param++;
  print("Yes\n\r");
}
static const command_t sayyes_cmd = {
  "yes", cmd_sayyes, "Says Yes"};
```

In main(), after terminal_init, you need to install the command (after terminal initialization), as follows:

```
(void)terminal_add_cmd((command_t*)&sayyes_cmd);
```

Finally, the main loop needs to be modified to call terminal_process() instead of GETCHAR() and PUTCHAR():

```
while (1)
{
      terminal_process();
  //ch = GETCHAR();
  //PUTCHAR(ch);
}
```

After making the project and running the debugger, it should operate. Furthermore, it should operate whether or not MCUXpresso is running or not. Just plugging in the KL43Z board and starting up PuTTY should give you the functioning system. Here's a sample transcript. Notice that the original "hello world" message comes out before terminal initialization.

```
hello world.
This is the simple terminal version 1.0

>help
I understand the following commands:
  help: Prints help about commands.
  yes:  Says Yes


>yes
Yes

>
```

So, now you can use the terminal to add commands to do things. Notice that you have available not just the print() function from hcc_terminal.c (which is what I used to say "Yes") but you also can use fprintf and fscanf as well, both very useful. (I expect that in "Hello world" floating point has not been installed, so you are probably limited to integers for the moment.)

Access to LED's (and other things):

You may be wondering why a "Yes" command was added instead of something to blink LED's. In fact, if you look at the "shell" demo, it does just what the initial CodeWarrior terminal demo did; it blinks the red LED on and off. Why didn't we do that? The answer is, it was not from lack of trying. Using the port access functions properly turns out to be much more complicated than on the SH8. So, for this "getting started" document that's been reserved for a separate step, which is what we come to now in this section of the document.

The basic problem is that when the "Hello world" project was created, it was not expected to use the "general purpose I/O" (parallel ports) features of the microcontroller or the board. So, the supporting code to do that was not included. If we want to blink an LED (or two) on and off, that code needs to be added.

If you look in the board folder of the "Hello world" project, there are four files: "peripherals.h" and "peripherals.c", which don't do much of anything. There are also files "pin_mux.h" and "pin_mux.c", which includes a function BOARD_InitPins() that sets up things for the UART so that the debug writes and reads can get to PuTTy. There is nothing that sets up ports, for example configuring the pins to the two LED's to be outputs and giving them an initial value. So, what we need to add is code to set up the LED pins for manipulation by an led command.

The first thing that is needed is to properly initialize the pins to be used. That may be done in the function BOARD_InitPins(). Instead, we will create a new function BOARD_InitLEDs() within the "pinmux.c" file, so we don't disturb the existing code for setting up the UART. Reference to the KL43Z schematic will reveal that the green LED is at pin Port D, pin 5 and the red LED at Port E pin 31. (No, Port E of the KL43Z does not have 32 pins, but maybe on some members of the Kinetis family PortE does have that many. There are many missing numbers.)

So, first we put some definitions into "pin_mux.h" and define our new initialization function "BOARD_Init_LEDs()". The definitions (in all caps) will be used for calls to initialize things in "pin_mux.c".

```
/*! @name PORTE31 (number 19), LED2
  @{ */
#define BOARD_LED2_GPIO GPIOE /*!<@brief GPIO device name: GPIOE */
#define BOARD_LED2_PORT PORTE /*!<@brief PORT device name: PORTE */
#define BOARD_LED2_PIN 31U    /*!<@brief PORTE pin index: 31 */
                      /* @} */
/*! @name PORTD5 (number 62), J2[12]/D13/SPI1_SCK/LED1/LCD_P45
  @{ */
#define BOARD_LED1_GPIO GPIOD /*!<@brief GPIO device name: GPIOD */
#define BOARD_LED1_PORT PORTD /*!<@brief PORT device name: PORTD */
#define BOARD_LED1_PIN 5U     /*!<@brief PORTD pin index: 5 */
                      /* @} */
void BOARD_Init_LEDs(void);
```

Now, in "pin_mux.c" we are now using "general purpose I/O" stuff, so we need a #include "gpio.h" to reference appropriate functions and structures found there. Then we need the code for initializing the LED's. Here's what we add to "pinmux.c":

(near the top of the file):

```
#include "fsl_gpio.h"
```

```
(later after the BOARD_InitPins function):
void BOARD_InitLEDs(void)
{
   /* Port D Clock Gate Control: Clock enabled */
   CLOCK_EnableClock(kCLOCK_PortD);
   /* Port E Clock Gate Control: Clock enabled */
   CLOCK_EnableClock(kCLOCK_PortE);

   gpio_pin_config_t LED1_config = {
      .pinDirection = kGPIO_DigitalOutput,
      .outputLogic = 1U};
   /* Initialize GPIO functionality on pin PTD5 (pin 62)  */
   GPIO_PinInit(BOARD_LED1_GPIO, BOARD_LED1_PIN, &LED1_config);

   gpio_pin_config_t LED2_config = {
      .pinDirection = kGPIO_DigitalOutput,
      .outputLogic = 1U};
   /* Initialize GPIO functionality on pin PTE31 (pin 19)  */
   GPIO_PinInit(BOARD_LED2_GPIO, BOARD_LED2_PIN, &LED2_config);

   const port_pin_config_t LED1 = {/* Internal pull-up/down resistor is disabled */
                           kPORT_PullDisable,
                           /* Slow slew rate is configured */
                           kPORT_SlowSlewRate,
                           /* Passive filter is disabled */
                           kPORT_PassiveFilterDisable,
                           /* Low drive strength is configured */
                           kPORT_LowDriveStrength,
                           /* Pin is configured as PTD5 */
                           kPORT_MuxAsGpio};
   /* PORTD5 (pin 62) is configured as PTD5 */
   PORT_SetPinConfig(BOARD_LED1_PORT, BOARD_LED1_PIN, &LED1);

   const port_pin_config_t LED2 = {/* Internal pull-up/down resistor is disabled */
                           kPORT_PullDisable,
                           /* Slow slew rate is configured */
                           kPORT_SlowSlewRate,
                           /* Passive filter is disabled */
                           kPORT_PassiveFilterDisable,
                           /* Low drive strength is configured */
                           kPORT_LowDriveStrength,
                           /* Pin is configured as PTE31 */
                           kPORT_MuxAsGpio};
   /* PORTE31 (pin 19) is configured as PTE31 */
   PORT_SetPinConfig(BOARD_LED2_PORT, BOARD_LED2_PIN, &LED2);
}
```

Notice that to initialize the LED's we have to turn on a clock for erach of the ports (D and E), then set the two pins to be outputs (using a structure "gpio_pin_config_t" for each and a calls to the function GPIO_PinInit(). Then we define "port_pin_config_t" structures for each LED that sets the port characteristics (things like pull-ups and drive strength) and call the function "PORT_SetPinConfig" for each pin to set those properties. For our SH8 things were much simpler; we would set the respective bits directly in registers without bothering with defining all these structures. But that requires direct access to the hardware. Here, all access to hardware is mediated by function calls. We could figure out a way to do it similar to how the SH8 programming has been done, but standards for software engineering are going the other way. Still, the SH8 experience is helpful because now you know what kinds of things these function calls are doing.

So, that takes care if the initialization code. Now the "hello_world.c" main program file needs to be modified to include the led command and a call to initialize the LEDs. Here's what we add:

```
static void cmd_led(char *param)
{
        static char ledr=0,ledg=0;
        if(param[0]=='r'||param[0]=='R'){
                if(ledr==1){
                        GPIO_WritePinOutput (GPIOE, 31, 1);
                        //LED_RED_OFF();
                        ledr=0;}
                else{
                        GPIO_WritePinOutput(GPIOE, 31, 0);
                        //LED_RED_ON();
                        ledr=1;}
        }
        else if(param[0]=='g'||param[0]=='G'){
                if(ledg==1){
                        //GPIO_WritePinOutput (GPIOD, 5, 1);
                        LED_GREEN_OFF();
                        ledg=0;}
                else{
                        //GPIO_WritePinOutput(GPIOD, 5, 0);
                        LED_GREEN_ON();
                        ledg=1;}
        }
        else print("Invalid LED\r\n");
}

/* the command to modify the LED state */
static const command_t led_cmd = {
  "led", cmd_led, "Toggle led R or G"};
```

Later in main() after the other initializations we need:

```
BOARD_InitLEDs();
```

And after the “yes” command install we add:

```
(void)terminal_add_cmd((command_t*)&led_cmd);
```

Now after downloading the code with the debugger and running it, we see a typical PuTTY session that actually does flip the LED’s on and off:

```
hello world.
This is the simple terminal version 1.0

>help
I understand the following commands:
  help: Prints help about commands.
  yes:  Says Yes
  led:  Toggle led R or G

>led
Invalid LED

>led r

>led g

>led g

>
```

New Projects:

This should be enough to get you started. Adding more commands, and using additional pins to control things, is similar. But, rather than modifying the existing Hello world program, wouldn’t it be better to start a new project from scratch? Well, maybe not completely from scratch, but with a little help from the MCUXpresso SDK. That’s what’s next.

You can create a new project starting with New from the File menu. If you do that, I suggest you start with the “New C/C++ Project” under “MCUXpresso IDE” rather than “C/C++ and “C project”. If you start with the latter you need to go through the process of picking the microcontroller, and you end up with an empty project that does not even include main(). If you start with New MCUXpresso IDE (or from the bottom left panel New Project instead of from the file menu) you can create a project that includes stuff designed to support projects on the KL43Z board, rather than having to start from scratch. So, that’s what I’m assuming you do here.

So, as you go through the steps that create the new project, a window pops up that shows a picture of the KL43Z board (you are in the SDK), and by picking it you are saying you are building code to run on that demo board. (If you are building your own board, you wouldn't do this.) Choose "Next".

This brings you to a pop-up window "Configure Your Project". This is the key step. You select what features you want already supported with appropriate drivers and option selections. The window is shown as Figure 6. There are lots of things to pick. Take the default processor (MKL43Z256VLP4), it's what is on your board. Select default board files, and make it a C project. I have not yet tried C++ under MCUXpresso. (Under CodeWarrior (free version) the C++ option is limited to 1K of flash, too little to do much of anything.)

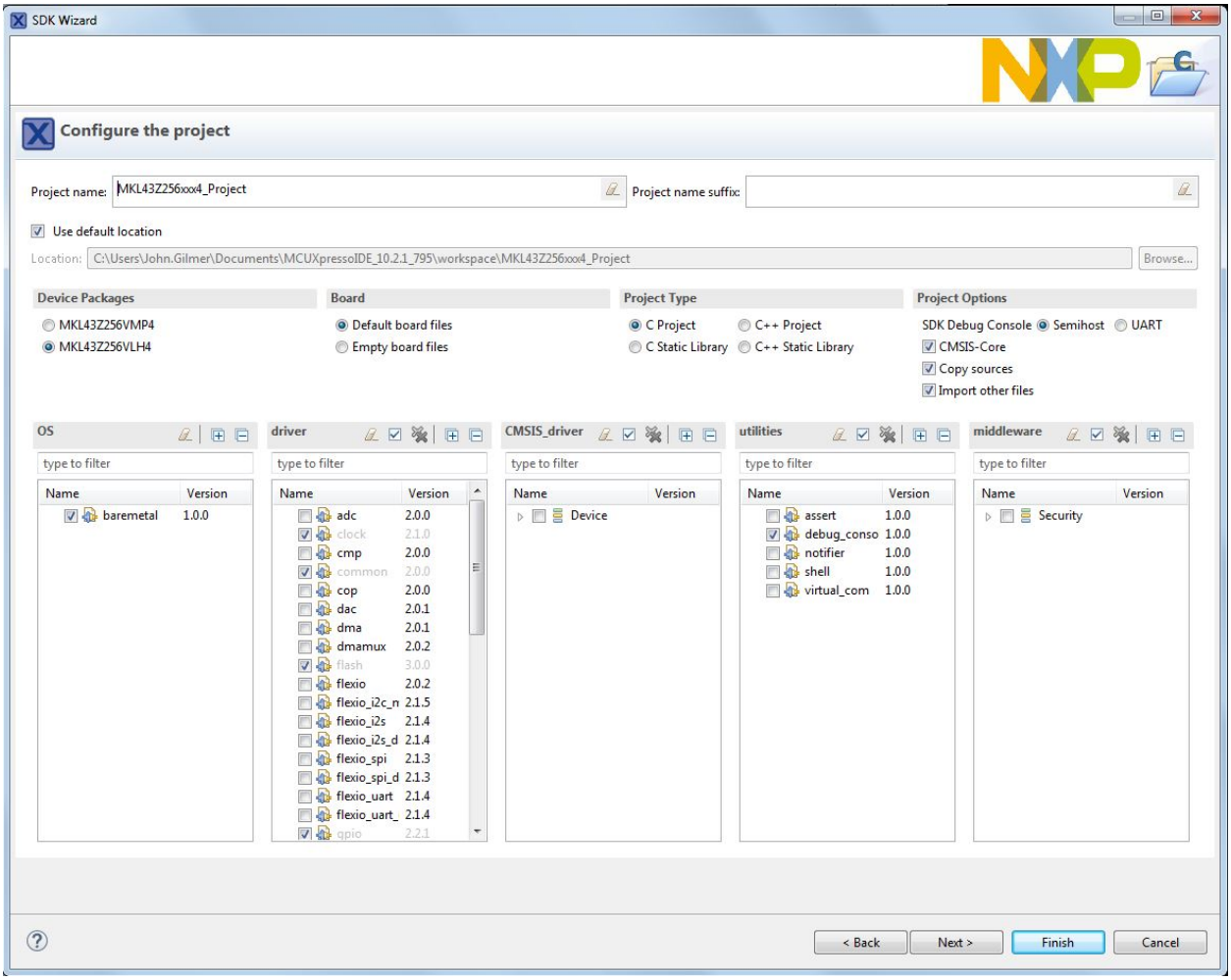

Figure 6 New Project Configuration

Toward the right is a column of "Project options". One is how you want the debug uart routed. This is where you decide whether to use the "Semihost" console within MCUXpresso or the external COM port via PuTTY. (I'm picking PuTTY, at least until I get a second UART working.) In this column for the other items, I'm not quite sure what we are choosing. I left the defaults ensabled, for CMSIS core (what's that?), copy sources, and import other files. I believe

that “copy sources” means that local copies of source files are made, rather than just referring to copies elsewhere (possibly in other projects). That’s the safe option. If you mess something up, the scope of the disaster is limited to this one project.

The “OS” (Operating System) that we are using is “baremetal”, meaning, there is none. If we wanted an operating system, a RTOS is available and can be downloaded. Life is complicated enough without getting into that now. “Baremetal” means there is no OS. That’s what we are used to from the SH8.

The “driver” column is very important. We want to pick a driver for every resource that we want to use on the board. The ones you particularly need to include: adc, dac, and maybe cop (the watchdog – did you notice that we have not been worrying about the dog?). Let’s skip DMA stuff (that’s a performance enhancing feature for handling lots of data). I’m also skipping the flexio stuff (have not had time to read up on that). The i2c communications may be needed for one of the on-board devices, but let’s skip it for now. I don’t know what ILWU is, skip it. The Low Power Timer might be usful; let’s choose it. “port” was automatically selected, but I skipped pit, pmc, and rmc. The rtc should be similar to what we are used to from the SH8; pick it. I’m skipping sai and sim. We DO want to use the LCD driver, so choose “slcd”. Skip spi, but let’s grab “tpm”; that ought to be similar to the one on the SH8 that we’ve used. The “uart” option is automatic. I chose “vref” (Voltage reference) in case I want it for something.

There are columns for CMSIS driver (what’s that?), utilities (we definitely want the debug console) and security. Who cares about security? Well, you will eventually, but for now we don’t. If you choose “shell” you are asking to include the much more capable command environment like our faithful “terminal” program. It’s much more complicated, but allows the user to pass in parameters to called command functions readily. I’d rather just use our terminal program for now, but you can switch to this sometime in the future if you want to invest the time in learning how to use it. I’m unfamiliar with the details of the other options. (I never did see an option of whether to include floating point or not. Maybe it’s automatically available.)

So, once you have finished, you get a main program that initializes things, includes a print of “hello world”, and then enters a main loop where all it does is count up an integer i. If you look in “pin_mux.h” and “pin_mux.c” you find a lot more stuff than in the “Hello world” program. You have initialization functions for all of those different things you chose in the drivers list. But, you need to call them. You will find the same BOARD_Init_LEDs() that we copied into the Hello world copy of “pin_mux.c”. (Now you know where I got that!). Likewise, “pin_mux.h” is expanded to cover all the other stuff. But to call the needed functions out of the man program, you need to include the “#include <driver>.h” file appropriate to the unit you are using, just as we needed “gpio.h” for the LEDs.

Starting with a new project, “pin_mux.h” and “pin_mux.c” already include all the stuff we had to add to the Hello world code, as well as support for the LCD display and other stuff we have not gotten to yet. So, that’s really the better way to start a new project. Still, starting with Hello world was useful because it was a working project from the start. Now that we know how to make a project work, it’s time to do it right starting from the beginning.

Conclusion: What we want to do is do-able! Let’s get to it.