Structure of a Program

1 Introduction

In this document we will summarize the structure of a 'C' program written for embedded systems. Although we concentrate on the NXP LPC824, the structure is the same for most cases. For simplicity and clarity, we will not be using any functions from Processor Support Library. The code explained here is completely self contained in the current directory.

2 How to compile

A 'C' program is typically made of the following components (in no particular order):

1. Initialization
2. Application code (what you write)
3. Library functions
4. A description of the memory map: Linker file.

When writing for a PC, you are only concerned with item 2; the source code that you write. However, the other items are already on your computer. For a microcomputer, you are under full control so that you can modify all parts if you wish.

To compile the example program, the following steps are needed.

1. Compile the source file:
   `arm-none-eabi-gcc -c -mcpu=cortex-m0 -mthumb -g  blink.c -o blink.o`
   This step is self explanatory. The source code `blink.c` is compiled to obtain the object file `blink.o`. We also tell the compiler that it should compile for a cortex M0 processor `-mcpu=cortex-m0` and use the ARM Thumb insruction set `-mthumb`.

2. Next, the initialization file `init.c` is compiled:
   `arm-none-eabi-gcc -c -mcpu=cortex-m0 -mthumb -g  init.c -o init.o`
   The same compiler settings are obtained as above.

3. We now have the source code compiled into intermediate **object files**, (.o). The next step is to link them. This step will produce the ELF file, which contains all of the information to construct the final file that will be programmed into the microcontroller:

```
arm-none-eabi-ld blink.o init.o -T lpc824_linker_script.ld
      --cref -Map blink.map -o blink.elf
```

The linking is done by using the linker command `arm-none-eabi-ld`. It is doing the following:

- Use the source files `blink.o init.o`
- Use the linker script file `lpc824_linker_script.ld`
- Generate a **map file** which tells the developer where within the memory the compiler placed each function and each variable etc. The file is `blink.map`
- Sets the filename of the output of this step as `blink.elf`. ELF i sa common format, also used in personal computers, is **Executable Linkable Format**. Essentially it contains all the information of the actual file that will be programmed into the processor.

4. Using the `.elf` file, we produce the `.hex` file, which literally lists which address of the memory will be programmed with which value. This is the final outcome of the compilation process:
   `arm-none-eabi-objcopy -O ihex  blink.elf blink.hex`
   This is essentially a simple step which translates the `.elf` file, into the `.hex` file.

5. The only remaining step is to program the microcontroller using the produced `.hex` file. The next step depends on the programmer software on your PC. You can use the following command, or use FlashMagic or a similar program: `lpc21isp -control blink.hex /dev/ttyUSB0`

The code should now be running on your Alakart with the Blue LED blinking.

3 GNU Make Program

The previous description was to show what is going on behind the scenes when you compile a program. But this is a lot of work to repeat for each compilation. There should be a simpler, automated way to do this. GNU Make is the standard method of compiling programs automatically. We provide it with a receipe of how to compile our code so that it can compile our code and program our microcontroller.

It has the following features:

- Compile each source code using a given set of rules.
- When we re-compile, take the minimum number of steps (i.e., do not re-compile the code if the source file has not been modified).

- Check dependencies. This means, if source code file `A.c` depends on a function in another file `B.c` and file `B.c` has been modified, then file `A.c` will also be compiled.
- Search and find library files within the file system.
- Also do other tasks besides compiling, such as programming the final `.hex` file into the microcontroller.

It has many more features, but for our course, we will be using the minimal set described above.

To compile a program, first we must prepare a file named `Makefile` in our current directory. All of the example code in the course has a Makefile, so that you can examine and start writing your own. It specifies the receipe for compiling your program. Once the `Makefile` is ready, compiling and burning your code is very simple. Just write the following in the terminal:

```
make
```

And GNU make will compile your code.

4 Components of a C Program

We will shortly look into the components of a 'C' program in the sequence as given in the Introduction.

4.1 Initialization

The initialization code is in the file `init.c` it does the following things:

1. Set the interrupt vectors
2. Initialize the clock
3. Plase the initial values in the variables (such as `int x=5;`)
4. Place zero as the initial value of uninitialized variables (such as `int y;`)
5. Call `main()` function.

4.1.1 The Vector Table

The vector table is in a specific location of the memory and is initialized by the following part of `init.c`. Please check the name of the second entry in the table: `init`.

```
const void * Vectors[] __attribute__((section(".vectors"))) ={
  (void *)(RAM_START+RAM_SIZE),   /* Top of stack */
  init,          /* Reset Handler */
  Default_Handler,     /* NMI */
  Default_Handler,     /* Hard Fault */
  0,                     /* Reserved */
  0,                   /* Reserved */
        ...
```

The original is quite long but what should appear where is stated in the interrupt vector table.

4.1.2 Clock initialization

This section writes the necessary registers to initialize the processor clock sourced from the internal 12MHz RC oscillator and boosts it to 30MHz. The details can be found in the 5.3.1 and 5.3.2 of the User Manual.

```
void clock_init() {
  // This function sets the main clock to the PLL
  // The PLL input is the built in 12MHz RC oscillator
  // This is multiplied up to 60MHz for the main clock
  // MSEL=4 (i.e.M=5), PSEL = 0 (P=1) see Sec. 5.3.1 of Ref Manual.

  SYSCON_PDRUNCFG &= ~(1 << 7); // Power up the PLL.
  SYSCON_SYSPLLCLKSEL = 0; // select internal RC oscillator
  ...
```

4.1.3 The main initialization function:

This is the first function that is called in the code. It calls the clock initialization (as can be seen in the listing below) and does other things that will be discussed further). But, **how does `init()` itself get called?** Check the vector table in Sec. 4.1.1. Init is defined as the Reset Vector, which means its address is written in the location the microcontroller checks right after power on.

```
void init() {
  // 1.Initialize the processor clock,
  // 2. Perform global/static data initialization
  // 3. Call "main()"

```

```
  unsigned char *src;
  unsigned char *dest;
  unsigned len;

  clock_init(); // boost speed to 30MHz
```

Next `init()` initializes the initialized variables (such as `int x=5;`) and puts zero as the value of the uninitialized variables (such as `int y;`).

```
  // Initialize variable values:
  src= &INIT_DATA_VALUES;
  dest= &INIT_DATA_START;
  len= &INIT_DATA_END-&INIT_DATA_START;
  while (len--)
    *dest++ = *src++;

  // zero out the uninitialized global/static variables
  dest = &BSS_START;
  len = &BSS_END - &BSS_START;
  while (len--)
    *dest++=0;
```

Finally `init()` calls the function `main()`:

```

  // Processor initialization is finished. Call the main() function:
  main();
}
```

Now you know that `main()` is not a magic word at all. Actually, you can modify the `init()` function to change the name of `main()` to anything you like...

At this point, `init()` is finished. Let us look at the main code, `blink.c`.

4.2 The Main User Code

Now the processor has been initialized, the variables have been prepared and it is ready to start executing from the `main()` function. We already know this fairly well, so we will go over soe interesting parts of `blink.c` only.

The first interesting part is how to set and clear specific bits in peripheral registers.

```

int main(void) {

  SYSCON_SYSAHBCLKCTRL |= 0x400C0; // Enable clocks for IOCON, SWM &
    GPIO.

```

```
6  SYSCON_PRESETCTRL &= ~(0x400);  // Assert  Reset of GPIO peripheral.
7  SYSCON_PRESETCTRL |=   0x400;   // Release Reset of GPIO peripheral.
8  // See 5.6.2 Peripheral reset control register in User Manual.
9  ...
```

In this listing the lines which set registers looks interesting:
`SYSCON_SYSAHBCLKCTRL |= 0x40040;`
It sets bits 6 and 18 of the register to '1'. This may be confusing at first, but if we write the hexadecimal number 0x40040 in binary, we will see:
`0x40040 = 0b0000 0000 0000 0100 0000 0000 0100 0000.`
The numbers are separated for easy reading (similar to the comma in the thousand's position in long numbers). You can count from the right at bit '0', and wil see that indeed bits 6 and 18 are set to 1. We can check from Table 35 in Sec. 5.6.14 "System clock control register" of the User Manual that these bits correspond to GPIO and IOCON peripherals, and by writing a '1' to those bits, those devices are turned on within the microcontroller.

The next interesting idea of the code line is the form:
`SYSCON_SYSAHBCLKCTRL |=...`
This means that we **logical OR the contents of the register** with the given number. How it works is as follows:

1. It reads the value stored in `SYSCON_SYSAHBCLKCTRL`
2. It then calculates the **logical OR of the value** with `0x40040`
   This operation sets bits 6 and 18 to '1' as desired.
3. Finally it stores the new value back in `SYSCON_SYSAHBCLKCTRL` register.

As you see, it can be done in one 'C' language operation. This is called a read-modify-store operation. Therefore this single line of code can be used to **set appropriate bits in registers**.

But how about clearing bits rather than setting them? This is in the next few lines of code:
`SYSCON_PRESETCTRL &= ~(0x400);`
Again let's write 0x400 in binary, but remembering that we use 32 bits:
`0x400 = 0b0000 0000 0000 0000 0000 0100 0000 0000.`
So bit 10 is a '1'. But since we use the **logical NOT operator** `~(0x400)`, every bit is inverted and it becomes: `~(0x400) = 0b1111 1111 1111 1111 1111 1011 1111 1111`
and since we use the **logical AND operator '&'**, this particular line of code means that we will be clearing bit 10 of the register PRESETCTRL, which means we take GPIO peripheral out of reset.

The next interesting bit is this:

```
1  //Make Pin PIO0_16 an output. On Alakart, PIO0_16 is the blue LED:
2  GPIO_DIR0 |= (1<<16);
```

We **logical OR** the content of `GPIO_DIR0` with some strange value. What does it mean? Actually this is a simple way of specifying which bit to set. In this case, we wish to set bit 16. Here '<<' is the shift operator. We take a '1' and shift it right 16 times to form:
`(1<<16) = 0b0000 0000 0000 0001 0000 0000 0000 0000`
Which is the number we wanted to create. So if we want to set or clear one bit only, we can use a simple shift operator and do not need to calculate values in hexadecimal.

We can use the same idea to clear bits:
`GPIO_DIR0 &= ~(1<<16);`
would clear bit 16 of the register.

2023