UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Lesson 5 OK05

The OK05 lesson builds on OK04 using it to flash the SOS morse code pattern (...---...). It is assumed you have the code for the Lesson 4: OK04 operating system as a basis.

Contents

1 Data

So far, all we've had to do with our operating system is provide instructions to be followed. Sometimes however, instructions are only half the story. Our operating systems may need data.

In general data is just values that are important. You are probably trained to think of data as being of a specific type, e.g. a text file contains text, an image file contains an image, etc. This is, in truth, just an idea. All data on a computer is just binary numbers, how we choose to interpret them is what counts. In this example we're going to store a light flashing sequence as data.

Some early Operating Systems did only allow certain types of data in certain files, but this was generally found to be too restrictive. The modern way does make programs a lot more complicated however.

At the end of 'main.s' copy the following code:

```
.section .data
.align 2
pattern:
.int 0b11111111101010100010001000101010
```

To differentiate between data and code, we put all the data in the .data. I've included this on the operating system memory layout diagram here. I've just chosen to put the data after the end of the code. The reason for keeping our data and instructions separate is so that if we eventually implement some security on our operating system, we need to know what parts of the code can be executed, and what parts can't.

`.align num` ensures the address of the next line is a multiple of 2^{num}.

I've used two new commands here. **`.align`** and **`.int`**. **`.align`** ensures alignment of the following data to a specified power of 2. In this case I've used **`.align 2`** which means that this data will definitely be placed at an address which is a multiple of $2^2 = 4$. It is really important to do this, because the **`ldr`** instruction we used to read memory only works at addresses that are multiples of 4.

`.int val` outputs the number **`val`**.

The **`.int`** command copies the constant after it into the output directly. That means that $11111111101010100010001000101010_2$ will be placed into the output, and so the label pattern actually labels this piece of data as pattern.

As I mentioned, data can mean whatever you want. In this case I've encoded the morse code SOS sequence, which is ...---... for those unfamiliar. I've used a 0 to represent a unit of time with the LED off, and a 1 to represent a unit of time with the LED on. That way, we can write some code which just displays a sequence in data like this one, and then all we have to do to make it display a different sequence is change the data. This is a very simple example of what operating systems must do all the time; interpret and display data.

One challenge with data is finding an efficient and useful representation. This method of storing the sequence as on and off units of time is easy to

Copy the following lines before the **`loop$`** label in 'main.s'.

```
ptrn .req r4
ldr ptrn,=pattern
ldr ptrn,[ptrn]
seq .req r5
mov seq,#0
```

This code loads the pattern into `r4`, and loads 0 into `r5`. `r5` will be our sequence position, so we can keep track of how much of the pattern we have displayed.

The following code puts a non-zero into `r1` if and only if there is a 1 in the current part of the pattern.

```
mov r1,#1
lsl r1,seq
and r1,ptrn
```

This code is useful for your calls to SetGpio, which must have a non-zero value to turn the LED off, and a value of zero to turn the LED on.

Now modify all of your code in 'main.s' so that each loop the code sets the LED based on the current sequence number, waits for 250000 micro seconds (or any other appropriate delay), and then increments the sequence number. When the sequence number reaches 32, it needs to go back to 0. See if you can implement this, and for an extra challenge, try to do it using only 1 instruction (solution in the download).

run, but would be difficult to edit, as the concept of a morse - or . is lost.

2 Time Flies When You're Having Fun...

You're now ready to test this on the Rapsberry Pi. It should flash out a sequence of 3 short pulses, 3 long pulses and then 3 more short pulses. After a delay, the pattern should repeat. If it doesn't work please see our troubleshooting page.

Once it works, congratulations you have reached the end of the OK series of tutorials.

In this series we've learnt about assembly code, the GPIO controller and the System Timer. We've learnt about functions and the ABI, as well as several basic Operating System concepts, and also about data.

You're now ready to move onto one of the more advanced series.

- The Screen series is next and teaches you how to use the screen with assembly code.
- The Input series teaches you how to use the keyboard and mouse.

By now you already have enough information to make Operating Systems that interact with the GPIO in other ways. If you have any robot kits, you may want to try writing a robot operating system controlled with the GPIO pins!
