UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Lesson 8 Screen03

The Screen03 lesson builds on Screen02, by teaching how to draw text, and also a small feature on the command line arguments ofthe operating system. It is assumed you have the code for the Lesson 7: Screen02 operating system as a basis.

Contents

1 String Theory

So, our task for this operating system is to draw text. We have several problems to address, the most pressing of which is probably about storing text. Unbelievably text has been one of the biggest flaws on computers to date. What should have been a straightforward type of data has brought down operating systems, crippled otherwise wonderful encryption, and caused many problems for users of different alphabets. Nevertheless, it is an incredibly important type of data, as it is an excellent link between the computer and the user. Text can be sufficiently structured that the operating system understands it, as well as sufficiently readable that humans can use it.

So how exactly is text stored? Simply enough, we have some system by which we give each letter a unique number, and then store a sequence of such numbers. Sounds easy. The problem is that the number of numbers is not fixed. Some pieces of text are longer than others. With storing ordinary numbers, we have some fixed limit, e.g. 32 bits, and then we can't go beyond that, we write methods that use numbers of that length, etc. In terms of text, or strings as we often call it, we want to write functions that work on variable length strings, otherwise we would need a lot of functions! This is not a problem for numbers normally, as there are only a few common number formats (byte, word, half, double).

Variable data types such as text require much more complex handling.

So, how do we determine how long the string is? I think the obvious answer is just to store how long the string is, and then to store the characters that make it up. This is called length prefixing, as the length comes before the string. Unfortunately, the pioneers of computer science did not agree. They felt it made more sense to have a special character called the null terminator (denoted \0) which represents when a string ends. This does indeed simplify many string algorithms, as you just keep working until the null terminator. Unfortunately this is the source of many security issues. What if a malicious user gives you a very long string? What if you didn't have enough space to store it. You might run a string copying function that copies until the null terminator, but because the string is so long, it overwrites your program. This may sound far fetched, but nevertheless, such buffer overun attacks are incredibly common. Length prefixing mitigates this problem as it is easy to deduce the size of the buffer required to store the string. As an operating system developer, I leave it to you to decide how best to store text.

Buffer overrun attacks have plauged computers for years. Recently, the Wii, Xbox and Playstation 2 all suffered buffer overrun attacks, as well as large systems like Microsoft's Web and Database servers.

The next thing we need to establish is how best to map characters to numbers. Fortunately, this is reasonably well standardised, so you have two major choices, Unicode and ASCII. Unicode maps almost every single useful symbol that can be written to a number, in exchange for having a lot more numbers, and a more complicated encoding system. ASCII uses one byte per character, and so only stores the latin

alphabet, numbers, a few symbols and a few special characters. Thus, ASCII is very easy to implement, compared to unicode, in which not every character takes the same space, making string algorithms tricky. Normally operating systems use ASCII for strings which will not be displayed to end users (but perhaps to developers or experts), and unicode for displaying messages to users, as unicode can support things like Japanese characters, and so could be localised.

Fortunately for us, this decision is irrelevant at the moment, as the first 128 characters of both are exactly the same, and are encoded exactly the same.

Table 1.1 ASCII/Unicode symbols 0-127

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
00	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
10	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
20		!	"	#	$	%	&	.	(	)	*	+	,	-	.	/
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

The table shows the first 128 symbols. The hexadecimal representation of the number for a symbol is the row value added to the column value, for example A is 41_{16}. What you may find surprising is the first two rows, and the very last value. These 33 special characters are not printed at all. In fact, these days, many are ignored. They exist because ASCII was originally intended as a system for typewriters to talk to hole card punches, and so a lot more information than just the symbols had to be sent. The key special symbols that you should learn are NUL, the null terminator character I mentioned before, HT, horizontal tab is what we normally refer to as a tab and LF, the line feed character is used to make a new line. You may wish to research and use the other characters for special meanings in your operating system.

2 Characters

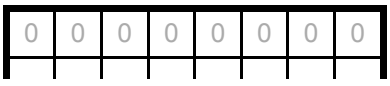
So, now that we know a bit about strings, we can start to think about how they're displayed. The fundamental thing we need to do in order to be able to display a string is to be able to display a character. Our first task will be making a DrawCharacter function which takes in a character to draw and a location, and then draws the character.

Naturally, this leads to a discussion about fonts. We already know there are many ways to display any given letter in accordance with font choice. So how does a font work? In the very early days of computer science, a font was just a series of little pictures of all the letters, called a bitmap font, and all the draw character method would do is copy one of the pictures to the screen. The trouble with this is when people want to resize the text. Sometimes we need big letters, and sometimes small. Although we could keep drawing new pictures for every font at every size with every character, this would get tedious. Thus, vector fonts were invented. in vector fonts, rather than containing an image of the font, the font file contains a description of how to draw it, e.g. an 'o' could be circle with radius half that of the maximum letter height. Modern operating systems use such fonts almost exclusively, as they are perfect at any resolution.

The true type font format used in many Operating Systems is so powerful, it has its own assembly langauge built in to ensure letters look correct at any resolution.

Unfortunately, much though I would love to include an implementation of one of the vector font formats, it would take up the remainder of this website. Thus, we will implement a bitmap font, though if you wish to make a decent graphical operating system, a vector font would be useful.

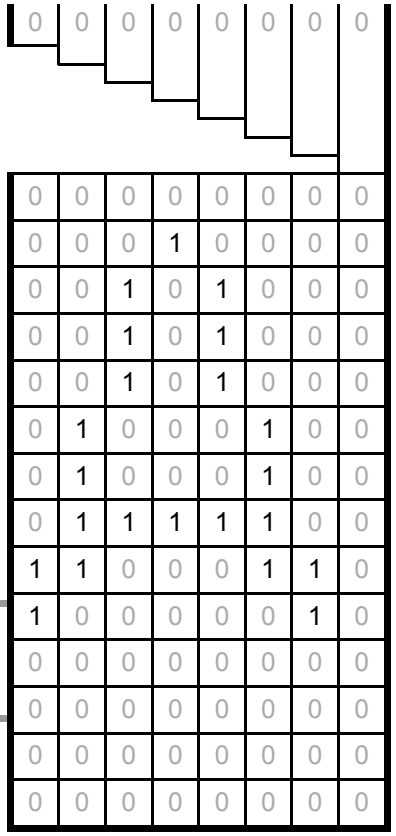
On the downloads page, I have included several '.bin' files in the font section. These are just raw binary data files for a few fonts. For this tutorial, pick your favourite from the monospace,



monochrome, 8x16 section. Download it and store it in the 'source' directory as 'font.bin'. These files are just monochrome images of each of the letters in turn, with each letter being exactly 8 by 16 pixels. Thus, each takes 16 bytes, the first byte being the top row, the second the next, etc.

The diagram shows the 'A' chracter in the monospace, monochrome, 8x16 font Bitstream Vera Sans Mono. In the file, we would find this starting at the $41_{16} \times 10_{16} = 410_{16}$th byte as the following sequence in hexadecimal:

00, 00, 00, 10, 28, 28, 28, 44, 44, 7C, C6, 82, 00, 00, 00, 00



We're going to use a monospace font here, because in a monospace font every character is the same size. Unfortunately, yet another complication with most fonts is that the character's widths vary, leading to more complex display code. I've included a few other fonts on the downloads page, as well as an explanation of the format I've stored them all in.

So let's get down to business. Copy the following to 'drawing.s' after the `.int 0` of graphicsAddress.

```
.align 4
font:
.incbin "font.bin"
```

This code copies the font data from the file to the address labeled font. We've used an `.align 4` here to ensure each character starts on a multiple of 16 bytes, which can be used for a speed trick later.

`.incbin "file"` inserts the binary data from the file `file`.

Now we want to write the draw character method. I'll give the pseudo code for this, so you can try to implement it yourself if you want to. Conventionally << means logical shift left.

```
function drawCharacter(r0 is character, r1 is x, r2 is y)
    if character > 127 then exit
    set charAddress to font + character × 16
    for row = 0 to 15
        set bits to readByte(charAddress + row)
        for bit = 0 to 7
            if test(bits << bit, 0x80)
            then setPixel(x + bit, y + row)
        next
    next
    return r0 = 8, r1 = 16
end function
```

If implemented directly, this is deliberately not very efficient. With things like drawing characters, efficiency is a top priority, as we will do it a lot. Let's explore some improvements that bring this closer to optimal assembly code. Firstly, we have a × 16, which by now you should spot is the same as a logical shift left by 4 places. Next we have a varaiable row, which is only ever added to charAddress and to y. Thus, we can eliminate it by increasing these variables instead. The only issue now is how to tell when we've finished. This is where the `.align 4` comes in handy. We know that charAddress will start with the low nibble containing 0. This means we can see how far into the character data we are by checking that low nibble.

Though we can eliminate the need for bits, we must introduce a new variable to do so, so it is best left in. The only other improvement that can be made is to remove the nested `bits << bit`.

```
function drawCharacter(r0 is character, r1 is x, r2 is y)
    if character > 127 then exit
    set charAddress to font + character << 4
    loop
        set bits to readByte(charAddress)
        set bit to 8
        loop
            set bits to bits << 1
```

```
            set bit to bit - 1
            if test(bits, 0x100)
            then setPixel(x + bit, y)
        until bit = 0
        set y to y + 1
        set chadAddress to chadAddress + 1
    until charAddress AND 0b1111 = 0
    return r0 = 8, r1 = 16
end function
```

Now we've got code that is much closer to assembly code, and is near optimal. Below is the assembly code version of the above.

```
.globl DrawCharacter
DrawCharacter:
cmp r0,#127
movhi r0,#0
movhi r1,#0
movhi pc,lr

push {r4,r5,r6,r7,r8,lr}
x .req r4
y .req r5
charAddr .req r6
mov x,r1
mov y,r2
ldr charAddr,=font
add charAddr, r0,lsl #4

lineLoop$:
    bits .req r7
    bit .req r8
    ldrb bits,[charAddr]
    mov bit,#8

    charPixelLoop$:
        subs bit,#1
        blt charPixelLoopEnd$
        lsl bits,#1
        tst bits,#0x100
        beq charPixelLoop$

        add r0,x,bit
        mov r1,y
        bl DrawPixel

        teq bit,#0
        bne charPixelLoop$
    charPixelLoopEnd$:
    .unreq bit
    .unreq bits
    add y,#1
    add charAddr,#1
    tst charAddr,#0b1111
    bne lineLoop$
.unreq x
.unreq y
.unreq charAddr

width .req r0
height .req r1
mov width,#8
mov height,#16

pop {r4,r5,r6,r7,r8,pc}
.unreq width
.unreq height
```

3 Strings

Now that we can draw characters, we can draw text. We need to make a method that, for a given string, draws each character in turn, at incrementing positions. To be nice, we shall also implement new lines and tabs. It's decision time as far as null terminators are concerned, and if you want to make your operating system use them, feel free by changing the code below. To avoid the issue, I will have the length of the string passed as an argument to the DrawString function, along with the address of the string, and the x and y coordinates.

```
function drawString(r0 is string, r1 is length, r2 is x, r3 is y)
    set x0 to x
    for pos = 0 to length - 1
        set char to loadByte(string + pos)
        set (cwidth, cheight) to DrawCharacter(char, x, y)
        if char = '\n' then /* \n is short hand for the character LF */
            set x to x0
            set y to y + cheight
        otherwise if char = '\t' then /* \t is short hand for the character HT */
            set x1 to x
            until x1 > x0
                set x1 to x1 + 5 × cwidth
            loop
            set x to x1
        otherwise
            set x to x + cwidth
        end if
    next
end function
```

Once again, this function isn't that close to assembly code. Feel free to try to implement it either directly or by simplifying it. I will give the simplification and then the assembly code below.

Clearly the person who wrote this function wasn't being very efficient (me in case you were wondering). Once again we have a pos variable that just increments and is added to something else, which is completely unnecessary. We can remove it, and instead simultaneously decrement length until it is 0, saving the need for one register. The rest of the function is probably fine, except for that annoying mutliplication by five. A key thing to do here would be to move the multiplication outside the loop; multiplication is slow even with bit shifts, and since we're always adding the same constant multiplied by 5, there is no need to recompute this. It can in fact be implemented in one operation using the argument shifting in assembly code, so I shall rephrase it like that.

```
function drawString(r0 is string, r1 is length, r2 is x, r3 is y)
    set x0 to x
    until length = 0
        set length to length - 1
        set char to loadByte(string)
        set (cwidth, cheight) to DrawCharacter(char, x, y)
        if char = '\n' then
            set x to x0
            set y to y + cheight
        otherwise if char = '\t' then
            set x1 to x
            set cwidth to cwidth + cwidth << 2
            until x1 > x0
                set x1 to x1 + cwidth
            loop
            set x to x1
        otherwise
            set x to x + cwidth
        end if
        set string to string + 1
    loop
```

```
end function
```

In assembly code:

```
.globl DrawString
DrawString:
x .req r4
y .req r5
x0 .req r6
string .req r7
length .req r8
char .req r9
push {r4,r5,r6,r7,r8,r9,lr}

mov string,r0
mov x,r2
mov x0,x
mov y,r3
mov length,r1

stringLoop$:
    subs length,#1
    blt stringLoopEnd$

    ldrb char,[string]
    add string,#1

    mov r0,char
    mov r1,x
    mov r2,y
    bl DrawCharacter
    cwidth .req r0
    cheight .req r1

    teq char,#'\n'
    moveq x,x0
    addeq y,cheight
    beq stringLoop$

    teq char,#'\t'
    addne x,cwidth
    bne stringLoop$

    add cwidth, cwidth,lsl #2
    x1 .req r1
    mov x1,x0

    stringLoopTab$:
        add x1,cwidth
        cmp x,x1
        bge stringLoopTab$
    mov x,x1
    .unreq x1
    b stringLoop$
stringLoopEnd$:
.unreq cwidth
.unreq cheight

pop {r4,r5,r6,r7,r8,r9,pc}
.unreq x
.unreq y
.unreq x0
.unreq string
.unreq length
```

This code makes clever use of a new operation, **subs** which subtracts one number from another, stores the result and then compares it with 0. In truth, all comparisons are implemented as a subtraction and then comparison with 0, but the result is normally discarded. This means that this operation is as fast as **cmp**.

subs reg,#val subtracts **val** from the

register **reg** and compares the result with 0.

4 Your Wish is My Command Line

Now that we can print strings, the challenge is to find an interesting one to draw. Normally in tutorials such as this, people just draw "Hello World!", but after all we've done so far, I feel that is a little patronising (feel free to do so if it helps). Instead we're going to draw our command line.

A convention has been made for computers running ARM. When they boot, it is important they are given certain information about what they have available to them. Most all processors have some way of acertaining this information, and on ARM this is by data left at the address 100_{16}. The format of the data is as follows:

1. The data is broken down into a series of 'tags'.
2. There are nine types of tag: 'core', 'mem', 'videotext', 'ramdisk', 'initrd2', 'serial' 'revision', 'videolfb', 'cmdline'.
3. Each can only appear once, but all but the 'core' tag don't have to appear.
4. The tags are placed from 0x100 in order one after the other.
5. The end of the list of tags always contains 2 words which are 0.
6. Every tag's size in bytes is a mutliple of 4.
7. Each tag starts with the size of the tag in words in the tag, including this number.
8. This is followed by a half word containing the tag's number. These are numbered from 1 in the order above ('core' is 1, 'cmdline' is 9).
9. This is followed by a half word containing 5441_{16}.
10. After this comes the data of the tag, which varies depending on the tag. The size of the data in words + 2 is always the same as the length mentioned above.
11. A 'core' tag is either 2 or 5 words in length. If it is 2, there is no data, if it is 5, it has 3 words.
12. A 'mem' tag is always 4 words in length. The data is the first address in a block of memory, and the length of that block.
13. A 'cmdline' tag contains a null terminated string which is the parameters of the kernel.

On the current version of the Raspberry Pi, only the 'core', 'mem' and 'cmdline' tags are present. You may find these useful later, and a more complete reference for these is on our Raspberry Pi reference page. The one we're interested in at the moment is the 'cmdline' tag, because it contains a string. We're going to write some code to search for the command line tag, and, if found, to print it out with each item on a new line. The command line is just a list of things that either the graphics processor or the user thought it might be nice for the Operating System to know. On the Raspberry Pi, this includes the MAC Address, serial number and screen resolution. The string itself is just a list of expressions such as 'key.subkey=value' separated by spaces.

Almost all Operating Systems support the notion of programs having a 'command line'. The idea is to provide a common mechanism for choosing the desired behaviour of the program.

Let's start by finding the 'cmdline' tag. In a new file called 'tags.s' copy the following code.

```
.section .data
tag_core: .int 0
tag_mem: .int 0
tag_videotext: .int 0
tag_ramdisk: .int 0
tag_initrd2: .int 0
tag_serial: .int 0
tag_revision: .int 0
tag_videolfb: .int 0
tag_cmdline: .int 0
```

Looking through the list of tags will be a slow operation, as it involves a lot of memory access. Therefore, we only want to have to do it once. This code creates some data which will store the memory address of the first tag of each of the types. Then, to find a tag the following pseudocode will suffice.

```
function FindTag(r0 is tag)
```

```
    if tag > 9 or tag = 0 then return 0
    set tagAddr to loadWord(tag_core + (tag - 1) × 4)
    if not tagAddr = 0 then return tagAddr
    if readWord(tag_core) = 0 then return 0
    set tagAddr to 0x100
    loop forever
        set tagIndex to readHalfWord(tagAddr + 4)
        if tagIndex = 0 then return FindTag(tag)
        if readWord(tag_core+(tagIndex-1)×4) = 0
        then storeWord(tagAddr, tag_core+(tagIndex-1)×4)
        set tagAddr to tagAddr + loadWord(tagAddr) × 4
    end loop
end function
```

This code is already quite well optimised and close to assembly. It is optimisitc in that the first thing it tries is loading the tag directly, as all but the first time this should be the case. If that fails, it checks if the core tag has an address. Since there must always be a core tag, the only reason that it would not have an address is if it doesn't exist. If it does have an address, the tag we were looking for didn't. If it doesn't we need to find the addresses of all the tags. It does this by reading the number of the tag. If it is zero, that must mean we are at the end of the list. This means we've now filled in all the tags in our directory. Therefore if we run our function again, it will now be able to produce an answer. If the tag number is not zero, we check to see if this tag type already has an address. If not, we store the address of this tag in our directory. We then add the length of this tag in bytes to the tag address to find the next tag.

Have a go at implementing this code in assembly. You will need to simplify it. If you get stuck, my answer is below. Don't forget the **`.section .text`**!

```
.section .text
.globl FindTag
FindTag:
tag .req r0
tagList .req r1
tagAddr .req r2

sub tag,#1
cmp tag,#8
movhi tag,#0
movhi pc,lr

ldr tagList,=tag_core
tagReturn$:
add tagAddr,tagList, tag,lsl #2
ldr tagAddr,[tagAddr]

teq tagAddr,#0
movne r0,tagAddr
movne pc,lr

ldr tagAddr,[tagList]
teq tagAddr,#0
movne r0,#0
movne pc,lr

mov tagAddr,#0x100
push {r4}
tagIndex .req r3
oldAddr .req r4
tagLoop$:
ldrh tagIndex,[tagAddr,#4]
subs tagIndex,#1
poplt {r4}
blt tagReturn$

add tagIndex,tagList, tagIndex,lsl #2
ldr oldAddr,[tagIndex]
```

```
teq oldAddr,#0
.unreq oldAddr
streq tagAddr,[tagIndex]

ldr tagIndex,[tagAddr]
add tagAddr, tagIndex,lsl #2
b tagLoop$

.unreq tag
.unreq tagList
.unreq tagAddr
.unreq tagIndex
```

5 Hello World

Now that we have everything we need, we can draw our first string. In 'main.s' delete everything after **`bl SetGraphicsAddress`**, and replace it with the following:

```
mov r0,#9
bl FindTag
ldr r1,[r0]
lsl r1,#2
sub r1,#8
add r0,#8
mov r2,#0
mov r3,#0
bl DrawString
loop$:
b loop$
```

This code simply uses our FindTag method to find the 9th tag (cmdline) and then calculates its length and passes the command and the length to the DrawString method, and tells it to draw the string at 0,0. Now test this on the Raspberry Pi. You should see a line of text on the screen. If not please see our troubleshooting page.

Once it works, congratulations you've now got the ability to draw text. But there is still room for improvement. What if we wanted to write out a number, or a section of the memory or manipulate our command line? In Lesson 9: Screen04, we will look at manipulating text and displaying useful numbers and information.
