TECHNICAL QUERIES

This month Alan Wrigley answers your questions on processor contents in Assembler, using the FontMax settings and reading the

Q Dear Sir

I have just started writing Wimp programs in Assembler and am a little confused about one point. Since other multi-tasking applications may be using the processor s registers between my calls to Wimp_Poll, does this mean I cannot rely on the contents being the same on return as they were when the call was made?

John Winwick

A Although the ARM processor only has one set of user-mode registers, which obviously must be used by all the applications currently running, the call to Wimp_Poll adheres to the same rules that apply when making any SWI call - in other words, unless the PRM states otherwise, the register contents and processor flags are preserved across the call. In the case of Wimp_Poll, only R0 is altered, since this must return the reason code. The Wimp stores the contents of all the other registers when you call Wimp_Poll, and restores them when it returns with a reason code. You may therefore keep information in other registers quite safely while your application is running.

While on the subject, it is worth repeating the warning we have given before that for SWI calls which require entry parameters in R1 and above, but not in R0, you should

Q ays assume that R0 may be corrupted by the call.

Dear Sir

A Please could you explain in plain English how to use the FontMax configuration settings.

Andrew Dalby

There are seven configuration settings for the font manager - FontMax, FontMax1 - FontMax5 and FontSize. FontMax4 and FontMax5 have a very specialised use which we will not consider here, while FontMax1 is largely irrelevant - for most purposes you can safely leave these configured to zero. FontSize and FontMax between them decide the size of the font cache (the area of memory set aside for the font manager to store information on the fonts currently in use), and were described in Into the Arc (RISC User 6:4).

This leaves FontMax2 and FontMax3, both of which have a direct bearing on the way in which fonts are used and displayed. When a font is used, the font manager normally creates from the outlines in the file a temporary bitmap for the point size in question and stores this in the cache. This is done to speed up the display of the font on the screen. However, each point size used takes up additional space in the cache. If anti-aliased bitmaps are stored, this requires much more space than simple monochrome bitmaps, and so FontMax2 is used to tell the font manager the largest point size that you want to be stored as an anti-aliased bitmap.

You can see how this works for yourself. Set FontMax2 to a value such as 14 points. Now open a document and type a line of text at 14 points, followed by another line at 15 points. You can see quite clearly that the characters in the second line look much thinner - this is because the 14-point characters have been fleshed out by anti-aliasing. A

sensible minimum value for FontMax2 is therefore the point size you would use for most normal text (usually 12-14 points).

FontMax3 determines whether outlines are converted to bitmaps and stored in the cache, or used directly. For large point sizes which are used infrequently, it makes little sense to fill up the cache, and so this configuration specifies the largest point size that will be converted to a bitmap. A sensible value is between 24 and 32 points, depending on the kind of work you are doing. You can test this out as follows: Display a paragraph of text in a point size at or just below the FontMax3 setting. Once the characters have been drawn for the first time, scrolling the text in and out of the window should be fairly smooth (assuming you have a large enough cache in the first place), since the bitmaps are all cached. Now change the point size to something larger than the configured setting and repeat the process. You will find that scrolling is no longer as smooth as it was, since the font manager now has to draw the outlines directly each time the characters reappear in the window.

Q Because the characters are drawn directly from the outlines in this case, they are not anti-aliased. This means that if you want to display anti-aliased text in any particular point size, the values of

A both FontMax2 and FontMax3 must be equal to or greater than the point size you are using.

Dear Sir

Could you please give me some information on how to detect movement of a joystick from within a Basic program.

Keith Lowe

The Joystick module which is part of RISC OS 3 allows you to read the state of either an analogue or a digital joystick. The module will only initialise if it detects built-in joystick hardware (as for example in the A3010). It provides just one SWI call, Joystick_Read (&43F40). On entry to this call, R0 should contain the joystick number, while on exit R0 holds a value which describes the current joystick state. For an analogue joystick this is as follows:

Byte 0: signed Y value: -127 (down) to 127 (up)

Byte 1: signed X value: -127 (left) to 127 (right)

Byte 2: fire buttons, starting at bit 0

For a digital joystick, byte 0 holds -64, 0 or 64 for down, centre and up respectively, while byte 1 holds -64, 0 or 64 for left, centre and right respectively.

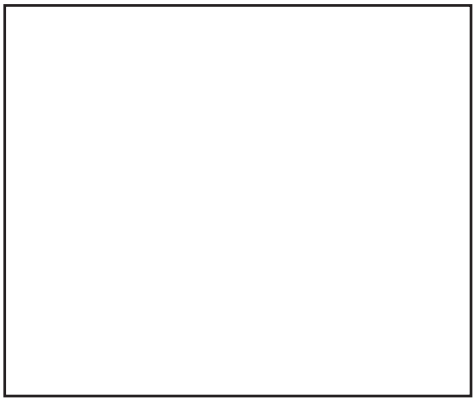
If you are only interested in the absolute state of a joystick (i.e. whether it is up, down, left, right or at rest), you should allow a margin of error for the at rest state, since analogue joysticks are not guaranteed to produce a value of zero. Acorn suggests a middle range of -32 to 32. So to read joystick 0 from a Basic program, the code would look something like this (assuming only one fire button):

```
SYS "Joystick_Read",0 TO state%
y%=FNcalc(state%):x%=FNcalc(state%>> 8)
button%=state% AND &10000
IF button% PROCfire
IF x%<0 PROCleft ELSE IF x%>0 PROCri ght
IF y%<0 PROCdown ELSE IF y%>0 PROCup
.....
DEF FNcalc(a%)
b%=(a% AND 255)-2*(a% AND 128)
=b% DIV 32
```

Bytes 0 and 1 are passed successively to FNcalc, which converts them into single-byte signed values, divides by 32 to allow for the middle range as suggested above, and passes the results back to y% and x% respectively. In the example given here, button% will be either &10000 or zero; since we only need a true or false result for one button, this is sufficient. With more than one button, however, you will need to do a little more decoding. The value in byte 2 can be obtained as follows:

```
(state%>>16) AND 255
```

caption



caption