Wimp Topics - Useful SWI Calls

by Alan Wrigley

This month s Wimp Topics is in a slightly different format from usual. Normally we discuss a single topic in detail, but this month I want to look at a selection of lesser-known Wimp SWI calls which can be useful for Wimp programmers, but which do not warrant a whole article in themselves.

WHICH ICON?

There is a very under-used SWI call, namely Wimp_WhichIcon (&400D6), which can be very useful in certain circumstances. What this call does is to tell you which icons, in a specified selection, match given criteria. The matching is done by comparing the icon flags word for each icon in the window. The entry parameters for the call, which will be explained in a moment, are as follows:

R0 = window handle (or -1 for icon bar)
R1 = pointer to block
R2 = bit mask
R3 = bit settings

On exit, the block pointed to by R1 is filled with the handles of matching icons.

The bit mask (R2) indicates which icon flags to consider when selecting the icons to compare, while the bit settings (R3) indicate which settings to compare those icons with. In other words, if (icon flags AND bit mask)=(bit mask AND bit settings) then that icon is included in the list returned by the call. This sounds a little complicated at first, but is quite straightforward when considered in the light of an example.

A very obvious use for this call, and the one which is most likely to be considered, is to detect which one of a group of radio icons is currently selected. Using the call can often simplify a program. The conventional way to handle radio icon groups is to maintain a variable to indicate the current setting, and alter that variable when mouse clicks are detected over one of the icons. However, in many cases you could dispense with the click handling procedure and simply read the currently selected icon when you need to act on it.

For example, suppose that you want to find out which icon with an ESG number of 2 is selected. The ESG number of an icon is held in bits 16-20 and has a maximum value of 31, while the selected bit is 21. So the bit mask would contain (31<<16)+(1<<21), i.e. &3F0000. This tells the Wimp that the ESG and the selected bit are the attributes you are interested in. The bit settings will tell the Wimp that the icons to match are any with the selected bit set plus an ESG of 2. This value is therefore set to (2<<16)+(1<<21), i.e. &220000.

The example would therefore be implemented as follows:

```
    SYS "Wimp_WhichIcon",whandle%,block%,&3F000
0,&220000
```

block% will now contain a list of the handles of all icons that match (4 bytes each), terminated by -1. In the example given here, there will be only one handle in the list, since only one radio icon can be selected at any one time.

BLOCK TRANSFER

If you are faced with the task of transferring the contents of a block of memory to another location, you might immediately think that a simple loop using indirection operators is the only logical way of doing it, as in the following example:

```
    FOR i%=0 TO &400 STEP 4
    destination%!i%=source%!i%
    NEXT
```

For transfers involving small amounts of memory, this method is quite acceptable. However, for larger blocks there is a much better way, which is to make use of the SWI Wimp_TransferBlock (&400F1). This is described by the PRM as a call to transfer a block of memory from one task s address space to another s, and several RISC User applications have made use of this facility to read data from another application (for example Desktop Auto-Save, RISC User 4:8). The entry parameters for the call are as follows:

```
      R0 = handle of source task
      R1 = pointer to source buffer
      R2 = handle of destination task
      R3 = pointer to destination buffer
R4 = buffer length
```

What the PRM does not tell you, however, is that the call still works if the source and destination task handles are the same; in other words, the transfer is made between two locations within the same task s memory.

This has a number of advantages over the indirection loop method. Firstly, it is very much faster. In tests that I carried out I found there could be as much as a ten-fold increase in speed over the alternative method. Secondly, it can simplify the code greatly. Although undocumented as such, the call appears to work correctly when the source and destination blocks overlap, and also the start address of the block does not have to lie on a word boundary. These two features taken together make the call extremely versatile, and enable it to be used, for example, for shunting text around in a document as it is edited.

To take a simple example, suppose that you have a text buffer 256 bytes long starting at buffer%. If you want to insert a character of ASCII value char% at position pos% in the buffer, all you need do is the following:

```
      SYS "Wimp_TransferBlock",ourtask%,buffer%+
pos%,ourtask%,buffer%+pos%+1,256-
pos%buffer%?pos%=ch ar%
```

and to delete the character at the same position:

```
      SYS "Wimp_TransferBlock",ourtask%,buffer%+
pos%+1,ourtask%,buffer%+pos%,256-pos%
```

You can also use the call to transfer a section of memory from your application workspace to a block of memory claimed from the RMA. In this case, you should quote the address of the RMA block as the destination address and use your own task handle for the destination as well as the source.

MENU SELECTION

The most widely-used method of acting on menu selections is to respond to poll code 9 (Menu_Selection), and to use the numeric data returned in the block to decide which menu item has been chosen. The following section of code is typical:

```
     DEF PROCmenuselect
     CASE !block% OF
      WHEN 0:CASE block%!4 OF
       WHEN 0:...
       WHEN 1:...
       WHEN 2:...
       ENDCASE
      WHEN 1:...
      WHEN 2:CASE block%!4 OF
       WHEN 0:...
```

and so on. This is all very incomprehensible to anyone except the programmer, and also means that if the menu structure is altered in any way, the values in the CASE statements will no longer correspond to the correct selection without alteration.

However, the Wimp provides a clearer way to do this. The SWI call Wimp_DecodeMenu can be used to convert a menu selection into a text string representing the item chosen. Not only will this make your program easier to read, but it will also mean that the menu selection procedure will need to be re-written less often - for example, as long as there is a menu item entitled Quit in the menu, it will be acted on correctly regardless of its position in the menu.

The call takes the following parameters:

```
     R1 = pointer to menu structure
     R2 = pointer to list of menu selections
     R3 = pointer to buffer
```

On exit, the buffer will contain a string representing the chosen item, with each element separated by a full stop. Examples might be Display.Font.Trinity , or Edit.Undo . The list pointed to by R2 can be taken directly from the block returned by poll code 9, so a menu selection procedure for a menu structure stored at menu% might now look