Using ANSI C

Part 11: Memory Allocation with Flex

by Lee Calcraft

ast month we took a brief look at memory allocation, concentrating on the standard ANSI function *malloc()* and its close relatives. This month it is the turn of *flex*. In contrast to *malloc()*, this powerful set of functions is primarily intended for allocating and maintaining large blocks of memory. It is ideal for use in word processors, databases or any other application where you are dealing with large amounts of data held in RAM, the size of which can vary during use.

The *flex* suite of functions does not form part of the ANSI standard, but is supplied with RISC_OSLib. The header for the suite is *flex.h*, and there are just eight functions (6 if you have not upgraded to C Release 4):

```
flex_init()
flex_alloc()
flex_size()
flex_free()
flex_extend()
flex_midextend()
flex_budge()
flex_dont_budge()
```

flex_init() must be called (with a void argument) just once in a program, at some point before any other *flex* functions are called. *flex_alloc()* allocates the required amount of memory, while *flex_size()* can be used to return the amount allocated, and *flex_free()* to release it.

The two *extend* functions provide alternative ways to increase or decrease the size of the allocated block, while the last two functions (only available on Release 4) can be used to determine whether the *malloc()* heap can expand beyond the original wimpslot boundary.

USING flex

The single feature of the *flex* suite of functions which distinguishes it from its *malloc()* relatives is that it takes its memory from the Wimp pool, extending or contracting the task's wimpslot as necessary. In Release 4 with the new extendibility of *malloc()* this distinction is diminished, but *malloc()* is still unable to contract the wimpslot when an allocated area is freed.

Because of its extreme flexibility, you must take great care when using *flex* that your pointers remain valid. Whenever you call *flex_alloc()*, *flex_extend()*, *flex_midextend()* or *flex_free()* the software which maintains the *flex* allocations may move your currently allocated blocks around without warning.

It is for this reason that the so-called *flex* anchor (a parameter used to identify a particular *flex* block) is not simply a pointer as it is with *malloc()*, but the address of a pointer (i.e. a pointer to a pointer). In this way *flex* can keep the pointer updated so that it always points to the start of a particular block, no matter where it has moved that block to.

In his turn the programmer must take great care always to reference the block using the anchor supplied together with his own integral offsets. The moment that he uses a copy of a dereferenced anchor to access his flex block (e.g. by passing the pointer to a function) he is vulnerable to any movement of the block.

A PRACTICAL EXAMPLE

An example should throw some light on all this. The prototype for *flex_alloc()* takes the following form:

```
int flex_alloc(flex_ptr anchor, int n);
```

where *n* is the required size of the block. The function returns 1 if the allocation was successful, or 0 otherwise.

The code below will allocate a 16K block of memory, store the word "Memory" in it, print the size of the allocated block, and print out the word as proof, then free the block.

```
char *ptr;
flex_init();
if (flex_alloc((flex_ptr) &ptr,16*1024))
{
 strcpy(ptr,"Memory");
 bbc_vdu(4);
 printf("size%d\n",\
 flex_size((flex_ptr) &ptr));
 printf(ptr);
 bbc_vdu(5);
 flex_free((flex_ptr) &ptr);
}
```

Note in this program segment the use of the cast to *flex_ptr type*. Another useful point is that *flex_free()* sets **ptr* to zero. This is very handy since it can be used as a flag to check whether a particular block has been released or not. In the above example, **ptr* will be zero after *flex_free()*. By initialising it to zero on declaration, we can ensure its validity as a flag at all times.

At the risk of stating the obvious, *ptr* must be defined so as to remain valid for the duration of the life of the flex block. In many cases this would mean that it is defined as a static variable. In such a case, the allocated block can be used long after the function in which the allocation was made has terminated.

Although it will not be obvious from running this example, the first *flex* allocation that a program makes will grab at least one page of memory from the Wimp pool. This is because *flex* obtains extra memory by extending an application's wimpslot, and this may only be achieved in page sized chunks. So to obtain just a single byte from *flex* will cost you up to 32K of memory - assuming that this is a first call to *flex*, or that previous calls have allocated memory up to a page boundary.

EXTENDING OR CONTRACTING A BLOCK

You can extend or contract an allocated *flex* block by using one of the two *extend* functions:

```
int flex_extend(flex_ptr,int newsize)
int flex_midextend(flex_ptr,int at,\
                                    int by)
```

Both return zero for failure, or 1 for success. The first simply extends the block, moving it and its contents as appropriate, while the second adds a new 'slice' of memory at the position *at* and of size *by* within the original block (negative values cause a slice to be removed). Again the block will be moved as appropriate, but this time the contents of the block from *at* to the end will be shuffled up to insert the slice.

As a test, if you insert the following lines immediately after *bbc_vdu(4)* in the example above, the word "Memorially" will appear in place of the word "Memory". This is because *flex_midextend()* has been used to insert 4 bytes of extra allocation between the "r" and the "y" of the original allocation, and we have then set the extra slice to read "iall". Note in this context that the allocation does not need to be an integral number of words, or to be on a word boundary.

```
if (flex_midextend((flex_ptr)&ptr,5,4))
{
 ptr[5]=i;
 ptr[6]=a;
 ptr[7]=l;
 ptr[8]=l;
}
```

THE RELATIONSHIP WITH THE MALLOC HEAP

As mentioned last month, in Release 4 the *malloc* heap is permitted to automatically extend by increasing the wimpslot. However, this would have a knock-on effect on all allocated *flex* blocks, since each time that *malloc()* grabs another page of contiguous memory, all *flex* blocks must be shuffled up to make room.

Because this would cause applications to crash which were written for Release 3 and which relied on *flex* changes occurring only when *flex* itself was called, Release 4 normally inhibits CLib from extending the wimpslot once *flex_init()* has been called.

Although this is the default on Release 4, two functions are provided to control this state of affairs. The function *flex_budge()*, which cannot be called directly, but is registered with CLib as follows:

```
    _kernel_register_slotextend(flex_budge)
```

causes the flex store to be moved up if CLib needs to extend the heap. This can be cancelled at any time by registering *flex_dont_budge()* in the same way:

```
    _kernel_register_slotextend\
                              (flex_dont_budge)
```

For further details, see the Release 4 manual, chapter 16. Next month we will take a look at aspects of the compiler and linker, including the use of libraries.