

Desktop C - the Data Transfer Protocol (Part 2)

After a somewhat lengthy gap, David Spencer concludes his look at file transfers in the

If you cast your mind back to the first part of this article (before the Risc PC launch threw a spanner in the scheduling), you will remember that we covered the Data Transfer Protocol for loading and saving data, and transferring data between applications. However, we left a couple of loose ends and these form the topic for this month.

DIRECT RAM TRANSFERS

The method we have explained previously for moving data between applications involves the sender saving the data into a temporary file (usually within !Scrap), and the recipient then loading this file and deleting it. Whilst this does work, it has drawbacks, such as the time taken to save and load the data, and the possibility of there not being enough spare disc space. Indeed, when you consider that the data is most probably being saved from RAM only to be loaded straight back into a different area of RAM, the whole thing seems unnecessarily complex, and indeed it is.

The alternative method that RISC OS supports goes something like this: an application receives a message indicating that the user has dragged a file to it; the application responds by replying that it would like to transfer the file directly through memory; assuming the sender agrees, the application moves the file from the sender's memory to its own in one or more blocks. Obviously this removes all the overheads of using a temporary file.

Before describing the details of this, one subtlety that must be addressed is what happens if either party in the exchange can't handle a direct RAM transfer. This is really quite simple; if the receiving application can't cope, then it simply opts for the temporary file method, and the sender has to go along with it. If on the other hand, the sending application can't handle the transfer, then all it has to do is to ignore the message from the receiver requesting the use of a direct transfer. When the receiver fails to get an

acknowledgement to its request, it must default to using the temporary file method.

So much for the background, but how does your poor C code handle all this. Well again, everything is thankfully made much easier by the provision of some handy functions in RISC_OSLib. From the sending application's point of view, remember back to last time when the calls `xfersend()`, `xfersend_pipe()` and `savesas()` were described. Each one of these calls allows a pointer to a function of type `xfer_sendproc` to be passed. At that time we simply set the pointer to zero to disable direct RAM transfers. However, all that is needed to enable them is to set this parameter to point to a function of the following type:

```
BOOL xfersend_sendproc(void *handle, int
*maxbuf)
```

Now, if the receiver can handle direct transfers then RISC_OSLib calls this function which must in turn call the function:

```
BOOL xfersend_sendbuf(char *buffer, int size)
which will send a block of data to the receiver of
length size from address buffer. The maximum length
that can be sent is the value maxbuf passed to
xfersend_sendproc(), although of course less can be
sent if the data is exhausted. In the case where there
is more data to send than can be sent, the sender
must update its pointer to the next chunk to send, and
RISC_OSLib will subsequently call
xfersend_sendproc() again. If at any point,
xfersend_sendbuf() returns FALSE, then the
xfersend_sendproc() function that called it must
return FALSE at once. This will abort the transfer and
return an error to the sending application.
```

At the receiver's end, the situation is slightly more complex, but not much. Now, when a DataSave message is received (which indicates a transfer from another application), the receiver should call:

```
int xferrecv_checkimport(int *estsize)
which checks if a RAM transfer is possible, instead
of calling xferrecv_checkinsert() as before. If this call
returns -1 then it means that no direct transfer is
possible, and the receiver should proceed to load the
temporary file by calling xferrecv_checkinsert() as
explained last time. However, if a direct RAM
transfer is possible then the filetype is returned, and
the integer pointed to by estsize is set to the
estimated size as given by the sender. In this case
the receiver should call:
```

```
int xferrecy_doimport(char *buf, int size,
xferrecy_buffer_processor)
which signifies that the sender should transfer size
bytes of data to the buffer pointed to by buf. If, once
the buffer is filled, the sender still has more data to
send, then RISC_OSLib will call the function passed to
the call, which takes the form:
```

```
BOOL xferrecy_buffer_processor(char **buf,
int *size)
```

At this point, the receiver can either empty the buffer and return TRUE, or can allocate a new buffer, and return its details by updating buf and size. If neither course of action is possible then it should return FALSE which will result in an error.


When xferrecy_doimport() finally returns, it returns either -1 for a failure of the transfer, or the number of actual bytes transferred.

DOUBLE-CLICKING ON FILES

The other area of concern is how to load a file when the user double-clicks on it, rather than dragging it to your application. There are in fact two quite distinct cases here. Firstly, when your application is already running, and secondly when your application must be started up first. We will only deal with the former

and has been dealt with in RISC User 3:10.

Quite simply, in response to the user double-clicking on a file, the Wimp broadcasts a message of the type DataOpen. Although the parameter block returned with this message is slightly different to that from the DataLoad message described last time, they both contain the salient information, namely the file name and type, in the same place, and both slot into the Data Transfer Protocol at the same point. Therefore the application can load the file, provided that it wants it, simply by following the procedure described last time of calling xferrecy_checkinsert(), loading the file, and finally calling xferrecy_insertfileok().

Incidentally, an application should only load a file in this way if it is the principal owner of the filetype in question. It is wrong for an application to load a file simply because it can cope with it. For example, if you double-click on a sprite file then Paint will load it because it can be considered the principal owner of the sprite filetype. However, Draw can also load sprite files, but they must be dragged in - double-clicking will not make Draw load them. 

After a longer than expected wait, the third part developers who last year purchased Medusa computers - the Risc PC prototype - have finally received ARM 700 cards. We thought we would take this opportunity to look at its performance.

ARM 700 VERSUS 600

The ARM 700 family offers three improvements over the ARM 600 to boost performance. Firstly, the cache size is increased from 4K to 8K, bringing it in line with the Intel 486. This means that fewer external memory accesses are needed. Secondly, the write buffer (see RISC User 5:4) now allows writes to four non-contiguous addresses instead of two, although the size is unchanged at eight words. This should result in less slowing down of the processor during memory writes. Finally, the ARM 700 can be clocked faster. The prototype card runs at 33MHz, as compared to 30MHz for production Risc PC 600s. In the future, faster versions of both processors will appear, though it is likely that the ARM 700 will always have the edge on clock speed.

SO HOW FAST?

The ultimate question then is just how much faster is the ARM 700 over the ARM 600? Well, the following table shows industry standard Dhrystone results for the two, as well as a 25MHz A5000. The results are a direct measure of general integer performance, with the higher the better.

We Test the ARM700

David Spencer tries out the next

From this, the ARM 700 delivers some 12.4% more power than the ARM 600. This seems a little disappointing, but remember that this is only one speed test (albeit the most widely used). Real applications that put more demand on cache performance are likely to show a markedly better improvement.

Processor	Dhrystones per second
ARM 3 25MHz	16697
ARM 610 30MHz	32144
ARM 700 33MHz	36140

MORE WAITING I M AFRAID

It must be stressed that the cards shipped are only prototypes. For those users waiting 