Implementing Link Paste in OPL

Notes prepared by Tom Dolbilin March 26, 1994.
Revision May 11, 1994.

Link Paste method is used for bringing data from one application to another. During this process the application that serves the data acts as a Link Paste Server (LPS) and the application that receives the data acts as a Link Paste Client (LPC). In general, applications may support only one of these modes, but the support for both is more common (Data, Word, Sheet and Agenda are the examples).

Notes on Link Paste Server example (LPS.OPL)

There can be only one active LPS on the system at any one time. In order to become one, the application has to register with the Window Server. Normally this is done only if the application has some of its information highlighted at the time when it is switched to background. As soon as it registers as the LPS, it should be ready to receive a message from a LPC that wants to copy over the data.

Initialization

One of the first things to do is to initialize messaging with the `MessInit` call:

```
ret%=call($83,$404,0,0,0,0)
```

Here the second parameter, `$404`, is interpreted as `4*$100+4` where the first 4 is number of bytes to allocate for the message arguments (per message). In the example these are two words `arg1%` and `arg2%`. The second `4` is the number of message slots to allocate for queuing.

Next we need the Window Server process ID. Note that in this particular case there is no need to add the terminating zero to `name$`, because it has just been declared and initialized:

```
name$="SYS$WSRV.*"
wsrvPid%=call($188,addr(name$)+1)
```

Now it is time to queue the console event and message reception. Both are asynchronous with status words `kStat%` and `mStat%` respectively:

```
ioa(-2,14,kStat%,event%(1),#0)
call($183,addr(pM%),0,0,0,addr(mStat%)) rem MessReceiveAsynchronous
```

Main event loop

In the main loop we wait for an event to happen. When it happens, we can find out what it was by checking the values of the status words. If `mStat%<>-46` then we have received a message. If `kStat%<>-46` then either a key has been pressed or a system event has occurred.

We are only interested in two types of messages:

`$21  TY_LINKSV_STEP` sets the beginning of a stage in the linking process

$22	TY_LINKSV_DEATH	sent to us if our LPC terminates

When a message is received, its type and the two arguments are read from the message structure located at pM%:

```
mtype%=peekw(pM%+4)
arg1%=peekw(pM%+8)
arg2%=peekw(pM%+10)
```

The meaning of the arguments is message-dependent. In the case of a TY_LINKSV_STEP message, it is also stage-dependent. The very first stage in the linking process is the initial request of a LPC. This is when we register the LPC as our client and find out which formats it supports. The rest of the stages are technically identical and the number of them depends of the amount of information to be transferred and the size of the temporary buffer it is going through.

The first stage in linking

We know if this is the first request because clntPid% is NULL. At this time clntPid% is set to the process ID of our LPC, which can be read from the message structure:

```
clntPid%=peekw(pM%+6)
```

However, we only do this if we support the format requested by the LPC. The format is a long integer that is stored across two words—the message arguments arg1% and arg2%:

```
fmt&=peekl(addr(arg1%))
```

fmt& can be a combination of any of the possible format values listed below:

$00	DF_LINK_NATIVE	application-dependent native format
$02	DF_LINK_TEXT	NULL paragraph delimiters are not transmitted, but different text buffers should be treated as separate paragraphs
$04	DF_LINK_TABTEXT	same as DF_LINK_TEXT, but can include tab characters
$10	DF_LINK_PARAS	the server can include NULL paragraph delimiters and the transmitted buffers of text are not treated as separate paragraphs

The subsequent stage(s) in linking

Non-zero clntPid% means that this is a subsequent stage when a chunk of data should be transferred from the server to the client. The two message arguments have the following meaning:

arg1%	the address in the client segment where the data buffer should be copied to. NULL signifies the end of the transfer.
arg2%	the size of the buffer to be copied. Normally, it should not exceed 256 bytes unless a native link-paste format is used.

If the amount of data to be sent must not exceed the size of the buffer. If it does, then it will have to be broken up in chunks and transferred in more than one step. After the length of the chunk is calculated, the buffer is copied to the client process segment by using the `ProcCopyToById` call:

```
ret%=min(len(buf$),arg2%)
call($92,clntPid%,ret%,0,addr(buf$)+1,arg1%)
```

The `ret%` value is important because it will have to be included in the `MessFree` call:

```
call($783,pM%,ret%)
```

The example only shows how to transfer a single buffer of data. However, it can be easily modified to allow for transfer of multiple buffers. Just keep the `state%` variable set to NULL until the last buffer has been transferred.

Becoming the server

None of the above will happen until we register ourselves as the LPS. We do this in response to the system event `$402` (send to background) by informing the Window Server with a `MessSendReceiveWithWait` call:

```
fmt&=$16 rem Prepared to render in any format
ret%=call($683,wsrvPid%,3,0,addr(fmt&))
```

The third parameter `3` is the `SY_LINK_SERVER` constant, the type of message to send.

Notes on Link Paste Client example (LPC.OPL)

Initialization

As in the case with LPS, we need to get the process ID of the Window Server:

```
name$="SYS$WSRV.*"
wsrvPid%=call($188,addr(name$)+1)
```

We also need to get the handle of our I/O semaphore to be able to release the signals we are not interested in. The semaphore handle is stored in the `E_PROC` structure associated with our process. This structure is found in the system kernel's segment at address calculated from our own process ID, `(ourPid% and $fff)`. Furthermore, the semaphore handle is stored in bytes 34 and 35 of the structure. We can use the `GenGetOsData` call to retrieve these 2 bytes:

```
call($78b,0,2,0,(call($88) and $fff)+33,addr(ioSem%))
```

Next, we use the `MessSendReceiveWithWait` call to ask the Window Server to give us the process ID of the LPS, if there is one. At the same time we pass the pointer to the address of `fmt&` so we can find out which formats are supported by the LPS:

```
pfmt%=addr(fmt&)
srvPid%=call($683,wsrvPid%,4,0,addr(pfmt%))
```

Getting the name of the LPS is unnecessary and the code does this only to display it in the title of the dialog, which lists all formats supported by the LPS and lets the user to select which of them to use. Again, this is done for demonstration purposes. In practice, your application should pick the best format automatically. Note, that since the native format `DF_LINK_NATIVE` is `0`, it is impossible to tell if it's available or even supported by the link-paste server. Use it only if you are sure that the LPS supports it. For example, the text editor mode of Word will crash if you request `DF_LINK_NATIVE` from it.

When the dialog returns, the selected format should be sent to the LPS as a message argument. Technically it should be 4 bytes long and stored across both message arguments, but because in this example the format values are small, it is sufficient to assign it only to the first message argument `w%(1)`.

Procedure `talk%:(arg1%,arg2%)`

This procedure sends the `TY_LINKSV_STEP ($21)` message to the LPS and waits for the reply. The procedure parameters are the message arguments. The return value is the error. The chunks of data are being received one by one until an error is returned. EOF is returned when all the data has been transferred.

One piece of code has been omitted from the LPC example for simplicity. It's supposed to make sure that the server doesn't die during the conversation. Without it, should the server die, the client will get stuck in an infinite loop. LPS.OPL shows how to set up such checking.

END OF DOCUMENT