Writing a UserShell

by Randell Jesup

One of the features of Release 2.0 is that the OS allows the user to change the system default shell, or the *UserShell*. Any time the user opens a shell with *NewShell*, executes a script, RUNs a command, or indirectly calls System() with `SYS_UserShell`, the OS will call the UserShell instead of the BootShell (by default the system sets up the BootShell as the UserShell).

Creating UserShells is not easy, and requires doing a fairly large number of things for no apparent reason (the reasons are there, they're just not obvious to the outsider). This article will attempt to give you the information you need in order to create a usable, system-friendly UserShell.

Initialization

The entity that starts the shell calls the shell code in C style (RTS to exit). This entity also sends a startup packet to your process port. You must retrieve this packet before doing any DOS I/O (much like WBMessages). You can use WaitPkt() for this. The entity will take care of attaching a CommandLineInterface structure to your process, which will be freed on exit from the UserShell by the system.

In your process structure, check the SegArray pointed to by pr_Seglist (note that it's a BPTR). If SegArray[4] is `NULL`, you must move the value from SegArray[3] to SegArray[4], and `NULL` out SegArray[3]. This is because SegArray[3] will be used to store the seglist pointer for each program you run.

The startup packet contains some information that tells the UserShell what kind of shell to be. At present, the two sets of sources can launch the UserShell:

> The Run command, Execute(), or System()
> The *NewShell* or *NewCLI* resident commands

The size of the stack that the system gives the UserShell depends on how the user started the shell. If it was started from Execute() or System(), the stack is 3200 bytes. If the UserShell was started from *Run*, *NewShell*, or *NewCLI*, the stack is 4000.

The type of shell required is specified by the combination of the packet's dp_Res1 and dp_Res2 fields. Here's a piece of code for turning them into a value from 0 to 3:

```
init_type = (parm_pkt->dp_Res1 == 0 ? 0:2)|(parm_pkt->dp_Res2 == 0 ? 0:1);
```

Currently, only types 0 and 2 are implemented. For 1 and 3 you should exit with an error (returning the packet). Type 0 is for Run, Execute() and System(), type 2 is for *NewShell* and *NewCLI*. After setting up your SegArray as above, for type 0 call CliInitRun(pkt), and for type 2 call CliInitNewcli(pkt). These both return a value we'll call ''fn''. Keep fn around, it has useful state information that you'll need later. Note that these CliInitXxxx functions don't follow the normal DOS convention of Dn for arguments (they use A0 for pkt!).

The CliInitXxxx functions do many magic things to get all the streams and structures properly set up, etc. You shouldn't need to know anything about this or what the values in the packet are, other than dp_Res1 and dp_Res2 (see the Appendix for more information on these functions).

Definitions for the values of fn:

Bit 31	Set to indicate flags are valid
Bit 3	Set to indicate an asynchronous System() call
Bit 2	Set if this is a System() call
Bit 1	Set if user provided input stream
Bit 0	Set if RUN provided output stream

If fn bit 31 is 0 (fn >= 0), then you must check IoErr() to find out what to do. If IoErr() is a pointer to your process, there has been an error in the initialization of the CLI structure and processing the packet. In this case you should clean up and exit. You don't have to return the packet because the CliInitXxxx functions take care of this for you if there is an error. If IoErr() isn't a pointer to your process, then if this is a *NewCLI* or *NewShell* command (init_type of 2), reply the packet immediately.

If the init_type is 0, you have to look at fn to determine when to send back the startup packet. If the shell was called from an asynchronous System() function ((fn & 0x8000000C) == 0x8000000C), return the packet immediately. If this is a synchronous System() call ((fn & 0x8000000C) == 0x80000004) or the fn flags are valid but this is not a System() call ((fn & 0x8000000C) == 0x80000000) (Execute() does this), you return the packet just before exiting from your shell (see the Cleanup section below). If the fn flags are invalid (bit 31 == 0), but there is something other than your task pointer in IoErr(), then this shell was called by the *Run* command. Here you can either return the packet immediately, or return it after having loaded the first command (or failed to find/load it). This delay in reply helps avoid the disk thrashing caused by two commands loading at the same instant.

When you do a ReplyPkt(), use ReplyPkt(pkt, pkt->dp_Res1, pkt->dp_Res2) to avoid losing error codes set up by CliInitXxxx.

Initialize pr_HomeDir to `NULL`, set up any local shell variables, etc.

We're all set up now, so you can now enter your main loop and start taking commands.

A Word About the Shell's I/O Handles

There are three pairs of I/O handles in a shell process. The shell's Process structure contains the pr_CIS (current input stream) and pr_COS (current output stream) file handles. That Process's CommandLineInterface structure contains the other two pairs of I/O handles: cli_StandardInput/cli_StandardOutput and cli_CurrentInput/cli_CurrentOutput. Each has different uses within a normal shell.

Routines that operate on Input() or Output(), such as ReadArgs() or ReadItem(), use the pr_CIS and pr_COS I/O handles (which they acquire by calling the *dos.library* routines Input() and Output(), not by directly looking at the Process structure). Shell-launched application programs the run on the shell's process also use these I/O handles as their normal input and output channels. This is where functions like scanf() and printf() get and send their input and output. The shell changes these file handles (using SelectInput()/SelectOutput()) according to the shell defaults and according to any I/O redirection.

The cli_StandardInput and cli_StandardOutput I/O handles are the default input and output channels for the shell. They usually refer to the user's console window and will not change while the shell is running. The shell should use these values as the default values for pr_CIS and pr_COS (via SelectInput() and SelectOutput()) when it runs a command from a command line.

The cli_CurrentInput handle is the current source of command lines. This normally is the same as cli_StandardInput. The cli_CurrentInput handle will differ from cli_StandardInput when the shell is executing a script or when handling an Execute() or System() call. In these cases, it points to a file handle from which the shell is reading commands. This handle refers to one of three files: the script file you called with the execute command, a temporary file created by the *execute* command, or a pseudo file created by Execute() or System().

When a shell runs the *execute* command, If cli_CurrentInput differs from cli_StandardInput, The *execute* command will close cli_CurrentInput and replace it with a new one, so don't cache the value of cli_CurrentInput as it will be invalid. In this case, cli_CurrentInput must *not* be the same as pr_CIS when you call RunCommand() if the executable could possible be the *execute* commands (or anything else that tries to close cli_CurrentInput).

The cli_CurrentOutput file handle is currently unused by the system. It's initialized to the same as cli_StandardOutput.

The Main Shell Loop

Note: some things in this section assume your UserShell will act similarly to the Boot Shell in 2.0. If not, modify to see fit, but pay close attention to things external programs will notice (such as the setup of the process and CLI structures). In particular, the article assumes that you handle scripts by redirecting cli_CurrentInput to a file with the script in it, as the *execute* command does. Note that the *execute* command will attempt to do this if you run it--be careful.

Before reading a command line, you need to SelectInput() on the I/O handle in the current cli_CurrentInput, and SelectOutput() on cli_StandardOutput. This makes sure the shell is reading from its command line source and writing to the the default output handle.

If this shell is executing a script, you should check if the user hit the break key for scripts (Ctrl-D is what the BootShell uses). If you do detect a script break, you can print an error message to the current output stream by calling PrintFault(304, "<your shell name>"). 304 is the error number (`ERROR_BREAK`) and the string gets prepended to the error message (which is currently " :***Break"). This allows the OS to print the error message using the standard error message which can be internationalized in future versions of the OS.

Next, determine if you should print a prompt. The nasty statement below sets up the Interactive flag for you, by setting it if the following are true:

This shell is not a background shell
input has not been redirected to an Execute/script file
this is not a System() call

You don't have to handle it precisely this way, but this works (Note: 0x80000004 is a test for whether this is a System() call, see the ‘‘fn’’ bit definitions above).

```
#define SYSTEM          ((((LONG)fn) & 0x80000004) == 0x80000004)
#define NOTSCRIPT       (clip->cli_CurrentInput == clip->cli_StandardInput)

        clip->cli_Interactive = (!clip->cli_Background && NOTSCRIPT && !SYSTEM) ?
                                DOSTRUE : FALSE;
```

The BootShell prints a prompt if cli_Interactive is `DOSTRUE`.

Do all your mucking with the input line, alias and local variable expansion, etc.

Finding a Program

There are several possible places a shell can look for commands passed to it. The resident list is an important place to look as it contains many commands that the user finds important enough to keep loaded into memory at all times. Some shells have commands built directly into them. Of course, if the shell cannot find a command in the resident list or in its collection of internal commands, the shell has to scan the path looking for the command. If a shell supports the script bit, when it finds a command on disk with the script bit set, it should read commands from the script file.

Here's how you deal with commands on the resident list: After finding the command (under Forbid()!), if the Segment structure's seg_UC is >= 0, increment it; if less than 0, don't modify it. If seg_UC is less than `CMD_DISABLED`, the corresponding resident command is currently disabled and you should not execute it. The same is true if seg_UC is equal to `CMD_SYSTEM`. After incrementing seg_UC, you can Permit(). After using a resident command, decrement the seg_UC count if it's greater than 0 (under Forbid() again).

When identifying scripts, I advise that you use something unique to identify your scripts, and pass all other scripts to the Boot Shell via System() for execution. A good method (which was worked out on BIX long ago) is to include within the first 256 characters or so, the string "#!<your shell name, ala csh>!#". BootShells could, for example, start with "; #!c:execute!#. The idea is the string inside the #!...!# should be the interpreter to run on the script. If none is specified, give it to the BootShell. If you want, you could extend this to include handling of the sequence for all interpreters. The programs should be invoked as "<interpreter> <filename> <args>" as if the user had typed that.

Don't forget to set pr_HomeDir for programs loaded from disk. The Lock in pr_HomeDir should be a DupLock() of the directory the program was loaded from. For programs from the resident list, leave it `NULL`.

Please support multi-assigned C: directories. The important thing here is to not lock C:. Instead, prepend ''C:'' onto the filename you wish to Lock()/LoadSeg(). Also, if a command is loaded from C:, get its pr_HomeDir by Lock()ing the file (with C: prepended), and then using ParentDir() to get its home directory.

The Path is attached to cli_CommandDir. It is a BPTR to a `NULL` terminated, singly-linked list (connected via BPTRs) of directory Locks:

```
    struct pathBPTRlistentry {
        BPTR   pathBPTRlistentry *next;
        struct Lock              directorylock
    }
```

Please don't modify the list; use the *Path* command instead. This will make it far easier for us to improve this in the future.

Make sure you clear the `SIGBREAK_CTRL_x` signals before starting a program. In order to prevent the user from hitting a break key somewhere between where you check for the break and where you clear the signals (thus losing the break signal), you may wish check for a break and clear the signals at the same time. The safest way is to use:

```
oldsigs = SetSignal(0L, SIGBREAK_CTRL_C |
                        SIGBREAK_CTRL_D |
                        SIGBREAK_CTRL_E |
                        SIGBREAK_CTRL_F);
```

Then you can check oldsigs for any signals that you care about.

Running a Program

To actually invoke a program on your process, use RunCommand()--it does special compatibility magic that keeps certain third-party applications working properly. If RunCommand() fails due to lack of memory, it returns -1 (normally success!). In this case, check IoErr(). If it is equal to `ERROR_NO_FREE_STORE`, then RunCommand() really ran out of memory. Note that RunCommand() stuffs a copy of your arguments into the buffer of the input handle for use by ReadArgs(), and un-stuffs them when the program exits. Also note that RunCommand() takes stack size in bytes, and cli_DefaultStack is the size of the stack in LONGs.

After the program has completed, free the Lock in pr_HomeDir and set it to `NULL`. Re-setup your I/O handles with SelectInput() on cli_CurrentInput and SelectOutput() on cli_StandardOutput. It's a good idea to `NULL` cli_Module here as well, as it can make your exit/cleanup logic easier.

You must eat any unused buffered input. Here's some tricky code that does that (read the Autodocs to figure it out if you wish):

```
        ch = UnGetC(Input(),-1) ? 0 : '\n';
        while ((ch != '\n') && (ch != END_STREAM_CH)) ch = FGetC(Input());
```

Note: `ENDSTREAMCH` is `EOF` (-1). Newer include files #define this in *<dos/stdio.h>* and *<dos/stdio.i>*.

To finish the main input loop, use the code below or something like it. This keeps compatibility with certain hacks people had figured out about 1.3 (See `SYSTEM` and `NOTSCRIPT` #defines above).

```
            /* for compatibility */
            /* system exit special case - taken only if they have played */
            /* around with the input stream */
            if (SYSTEM && NOTSCRIPT) break;
        } while (ch != ENDSTREAMCH);
```

EndCLI sets fh_End = 0, which causes FGetC() to call replenish(), which returns `ENDSTREAMCH` on fh_End == 0. *EndCLI also* sets cli_Background! This neatly avoids a prompt after *EndCLI*.

After you've gotten an `EOF` (falling out of the while(ch != `ENDSTREAMCH`) statement above), you need to check if the shell was executing a script file. For Execute-type scripts, if (cli_CurrentInput != cli_StandardInput) is true, the shell was executing a script. If this is the case, you'll need to Close() the cli_CurrentInput, and DeleteFile() the temporary file cli_CommandFile, if there is a file name there. Next, set cli_CurrentInput to cli_StandardInput, and restore the fail level**.** Then you go back to your normal input loop and accept commands again. Note: this is based on handling scripts in the same manner as the BootShell--you may handle them in different ways.

On `EOF`, you also need to check if this is a System() call. The check for a System() call is ((fn & 0x80000004) == 0x80000004). If you had been handling a System() call, or if the shell was not executing a script, you should go to your cleanup code to exit.

Cleanup

If you're exiting, use fn to tell you what to close, etc. First check if fn contains valid flags ((fn & 0x80000000) != 0)). If it *does not* have valid flags, Flush() and Close() cli_StandardOutput (if non-NULL), and Close() cli_StandardInput (if non-NULL). If fn does contain valid flags, Flush(Output()), then check the other flags in fn. If (fn&2 == 0) (if the user *didn't* provide an input stream), Close() cli_StandardInput. If (fn&1 == 1) (if *Run* provided an output stream), Close() cli_StandardOutput (note, this is opposite the previous flag!) If (fn&8 == 0) (if this is *not* an asynchronous System() call), you still have to ReplyPkt() the initial packet. Before sending back the packet put cli_ReturnCode in the packet's result1 and cli_Result2 in the packet's result2 (i.e. return the result of the last program run if this was a synchronous System() or Execute() call).

In cleanup, unlock pr_CurrentDir and set it to NULL, free up anything you allocated, and exit! The system will take care of your CommandLineInterface structure, and anything else it allocated.

Installing the New User Shell

After you have compiled your creation, you need to put its seglist on the resident list under the name ''shell''. Adding it to the resident list is a simple:

    Resident shell <shell-file> SYSTEM

Now anything that calls the user shell (like *NewShell*, Execute(), and System()) will call your shell. Note that under 2.04, the *Shell* icon actually runs *sys:System/CLI*, which calls the BootShell (the default UserShell) and not the current UserShell.

If you need to restore the BootShell as the UserShell, compile and run the program *RestoreShell.c* at the end of this article

Credits

I thank greatly the input and bug reports I got from Bill Hawes during the development of the 2.0 DOS and it's shell interface. It's still extremely ungainly, but it is now usable.

I also thank Michael B. Smith and Jesper Steen Moller for taking the initial confusing Usenet article I posted and making working shells from that information, and John Orr for making me write this article and producing the example (which helped make me clean up confusing points in it).