Appendix

```
dos.library/CliInitNewcli                          dos.library/CliInitNewcli

   NAME
        CliInitNewcli -- Set up a process to be a shell from initial packet

   SYNOPSIS
        flags = CliInitNewcli( packet )
        D0                     A0

        LONG CliInitNewcli( struct DosPacket * )

   FUNCTION
        This function initializes a process and CLI structure for a new
        shell, from parameters in an initial packet passed by the system
        (NewShell or NewCLI, etc).  The format of the data in the packet
        is purposely not defined.  The setup includes all the normal fields
        in the structures that are required for proper operation (current
        directory, paths, input streams, etc).

        It returns a set of flags containing information about what type
        of shell invocation this is.

        Definitions for the values of fn:
                Bit 31     Set to indicate flags are valid
                Bit  3     Set to indicate asynch 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 Bit 31 is 0, then you must check IoErr() to determine if an error
        occurred.  If IoErr() returns a pointer to your process, there has
        been an error, and you should clean up and exit.  The packet will
        have already been returned by CliInitNewcli().  If it isn't a pointer
        to your process and Bit 31 is 0, reply the packet immediately.
        (Note: this is different from what you do for CliInitRun().)

        This function is very similar to CliInitRun().

   INPUTS
        packet - the initial packet sent to your process MsgPort

   RESULT
        fn - flags or a pointer

   SEE ALSO
        CliInitRun(), ReplyPkt(), WaitPkt(), IoErr()
```

```
dos.library/CliInitRun                                  dos.library/CliInitRun

   NAME
        CliInitRun -- Set up a process to be a shell from initial packet

   SYNOPSIS
        flags = CliInitRun( packet )
        D0                  A0

        LONG CliInitRun( struct DosPacket * )

   FUNCTION
        This function initializes a process and CLI structure for a new
        shell, from parameters in an initial packet passed by the system
        (Run, System(), Execute()).  The format of the data in the packet
        is purposely not defined.  The setup includes all the normal fields
        in the structures that are required for proper operation (current
        directory, paths, input streams, etc).

        It returns a set of flags containing information about what type
        of shell invocation this is.

        Definitions for the values of fn:
                Bit 31     Set to indicate flags are valid
                Bit  3     Set to indicate asynch 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 Bit 31 is 0, then you must check IoErr() to determine if an error
        occurred.  If IoErr() returns a pointer to your process, there has
        been an error, and you should clean up and exit.  The packet will
        have already been returned by CliInitNewcli().  If it isn't a pointer
        to your process and Bit 31 is 0, you should wait before replying
        the packet until after you've loaded the first command (or when you
        exit).  This helps avoid disk "gronking" with the Run command.
        (Note: this is different from what you do for CliInitNewcli().)

        If Bit 31 is 1, then if Bit 3 is one, ReplyPkt() the packet
        immediately (Asynch System()), otherwise wait until your shell exits
        (Sync System(), Execute()).
        (Note: this is different from what you do for CliInitNewcli().)

        This function is very similar to CliInitNewcli().

   INPUTS
        packet - the initial packet sent to your process MsgPort

   RESULT
        fn - flags or a pointer

   SEE ALSO
        CliInitNewcli(), ReplyPkt(), WaitPkt(), System(), Execute(), IoErr()
```

```
;/* myshell.c - Execute me to compile me with Lattice 5.10b
LC -b0 -cfist -v -d0 -j73 myshell.c
Blink FROM myshell.o TO myshell LIBRARY lib:amiga.lib lib:debug.lib smallcode smalldata
quit

Where:
    -b1         means   Locate data as a 16 bit offset from A4.
    -cfist      means   Compatibility options:
                            f - Forces compiler to complain
                                  if a function has no prototype.
                            i - Ignores extra #includes of
                                  an already #included file.
                            s - Tells LC to generate a single copy
                                  of all identical string constants
                                  into a program's code section.
                            t - Tells LC to warn you if it encounters
                                  a structure or union tag that has
                                  not been defined.
    -v          means   Disable the stack checking LC normally
                            puts at the beginning of each function.
    -d0         means   Do not put any debugging info into the object file.
    -y          means   Load the base address of the data section
                            into A4.  This is used in conjunction
                            with -b1.
    -j73        means   Ignore warning #73.  This basically makes LC
                            ignore the script comment delimiter (the
                            semicolon) at the front of this file.
*/

/* The following command lines will compile myshell.c using pragmas
LC -b0 -cfist -v -r1 -rr -d0 -j73 -dDOPRAGMAS myshell.c
Blink FROM myshell.o TO myshell smallcode smalldata

Where:
    -r1         means   Make all subroutines "near".  The compiler will
                            use a 16-bit PC relative address to locate
                            the subroutine.
    -rr         means   Use registerized parameters for subroutines (no
                            amiga.lib stubs!).

The rest of the parameters were described above.
*/

/*
 * This is a basically a skeleton of a UserShell.  A UserShell is a special
 * shell that can replace the default system shell.  It has to meets some
 * system requirements to function as a system shell.  This example takes care
 * of all of those requirements. To make this shell the system shell, use the
 * resident command:
 *
 * resident shell MyShell SYSTEM
 *
 * Because this shell only serves as an minimal example of a UserShell and does
 * not do many of the standard functions a shell normally performs.  It is
 * limited to the commands from the resident list (which would make it a bad
 * idea to add this shell to the resident list if you need a useable default
 * shell!)
 */

#include <exec/types.h>
#include <exec/memory.h>
#include <dos/dosextens.h>
#include <dos/stdio.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>
#include <clib/alib_stdio_protos.h>


#ifdef DOPRAGMAS
#include <pragmas/exec_pragmas.h>
#include <pragmas/dos_pragmas.h>
#endif

long            main(void);

#define COMMANDBUFLENGTH    64
#define COMMANDLINELENGTH   512
#define PROMPTLENGTH        256

/* True if this is a System() Instigated shell */
#define SYSTEM ((ml->fn & 0x80000004) == 0x80000004)

/* true if this shell is executing a script */
#define ISSCRIPT (ml->mycli->cli_CurrentInput != ml->mycli->cli_StandardInput)

/* true if this shell is *not* executing a script */
#define NOTSCRIPT (ml->mycli->cli_CurrentInput == ml->mycli->cli_StandardInput)

struct mylocals
{
  struct Library    *sysBase;
  struct Library    *dosBase;
  struct Process    *myprocess;
  struct DosPacket  *mypacket;
  long              fn;        /* notice that fn is signed.  Some conditionals
                                  in this code rely on this value being signed.
                               */
  struct CommandLineInterface *mycli;
};

/*
 * define the library base labels (SysBase and DOSBase) that amiga.lib looks
 * for so we don't have to declare them as a global.  Can't have global data in
 * resident code.
 */
#define SysBase (ml->sysBase)
#define DOSBase (ml->dosBase)

long            mainshellloop(struct mylocals *);
long            strlen(UBYTE *);

long
main(void)
{
  struct mylocals globals, *ml = &globals;
  BPTR            *segment;
  long             shelltype, error;

  /*
   * Poof, this shell has winked into existence.  It could have come from the
   * user executing the "newshell" code via the newshell program, or it could
   * have come from some other application using one of the DOS calls that use
   * a shell, like System().  In any case, whatever caused this shell to wink
   * into existence will also cause a special startup packet to appear in this
   * process' message port.  This packet is very personal and private to DOS
   * and probably will change with future versions of the OS, so don't worry
   * about what's in it.  That would be bad.
   */
  error = RETURN_OK;
  /* Open libraries */
  SysBase = *((struct Library **) 4L);
  if (DOSBase = OpenLibrary("dos.library", 37))
  {
    /* First, get the packet that the newshell segment sends. */
    globals.mypacket = WaitPkt();
    globals.myprocess = (struct Process *) FindTask(NULL);

    /*
     * Some arcane magic here for the UserShell.  We have to look at this
     * process' array of Segment pointers.  If entry 4 in the array is NULL, we
     * have to move entry 3 to entry 4 and NULL entry 4.  This is because entry
     * 3 will be used to store the seglist pointer for each program this shell
     * runs.
     */
    segment = (BPTR *) BADDR(globals.myprocess->pr_SegList);
    if (!segment[4])
    {
      segment[4] = segment[3];
      segment[3] = NULL;
    }
```

```
    /*
     * The packet that newshell sends tells us how the shell was invoked. The
     * dp_Res1 and dp_Res2 fields of the packet structure represent,
     * respectively, the high order bit and low order bit of a two-bit
     * bitfield.  The following line of code will turn these values into a
     * value from 0 to 3:
     */
    shelltype = (globals.mypacket->dp_Res1 == 0 ? 0 : 2) |
      (globals.mypacket->dp_Res2 == 0 ? 0 : 1);

    /*
     * at the moment, only the values 0 and 2 are defined.  Type 0 is for Run,
     * Execute(), and System().  Type 2 is for NewShell and NewCli.
     */
    if ((shelltype == 2) || (shelltype == 0))
    {

      /*
       * These two functions CliInitNewcli() and CliInitRun() take care setting
       * up the shell's CommandLineInterface structure (current directories,
       * paths, input streams...) using the secret startup packet we got
       * earlier.  They differ slightly in their setup based on the shell type.
       * The exact workings of these functions is private and personal to DOS,
       * and is subject to change. If you are wondering what exactly these
       * functions do, don't worry about it.  That would also be bad.
       */
      if (shelltype == 0)
        globals.fn = CliInitRun(globals.mypacket);
      else

        /*
         * CliInitNewCli() handles the shell startup file (default is
         * s:Shell-startup) and stuffs a filehandle to it into
         * globals.mycli->cli_CurrentInput.
         */
        globals.fn = CliInitNewcli(globals.mypacket);

      /*
       * Definitions for the values of globals.fn:
       *     Bit 31     Set to indicate flags are valid
       *     Bit  3     Set to indicate asynch 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 the high bit of globals.fn is clear, check IoErr() to see if it
       * points to this process.  If it does, there was an error with the
       * CliInitXxx... function.  On an error, clean up and exit.  You won't
       * have to return the packet if there was an error because the
       * CliInitXxxx function will take care of that.
       */
      if ((globals.fn & 0x80000000) == 0)       /* Is high bit clear? */
        if ((struct Process *) IoErr() == globals.myprocess)   /* is there an error? */
          error = RETURN_FAIL;
        else if (shelltype == 0)
        {
          ReplyPkt(globals.mypacket,
                   globals.mypacket->dp_Res1,
                   globals.mypacket->dp_Res2);
          globals.mypacket = NULL;
        }
      if (error != RETURN_FAIL)
      {

        /*
         * OK, no error.  If this shell was invoked via NewShell or NewCLI
         * (shelltype == 2), or if this is an asynchronous System() initiated
         * shell, return the startup message.   Although this example doesn't
         * do it, if shelltype == 0, you can wait to reply the packet until you
         * try to LoadSeg() your first command (to avoid disk gronking). When
         * you use ReplyPkt() to reply the packet, use it like it appears below
         * to avoid losing error codes set up by CliInitXxx.
         */
        if (((globals.fn & 0x8000000C) == 0x8000000C) || (shelltype == 2))
        {
          ReplyPkt(globals.mypacket,
                   globals.mypacket->dp_Res1,
                   globals.mypacket->dp_Res2);
          globals.mypacket = NULL;
        }

        if (globals.mycli = Cli())
        {
          /* Set up local shell variables and any custom set up here */
          globals.mycli->cli_ReturnCode = 0;
          globals.mycli->cli_Result2 = 0;
          globals.myprocess->pr_HomeDir = NULL;

          /* Ready to start processing commands */
          error = mainshellloop(ml);
          if (globals.fn < 0)          /* if we got valid flags from
                                        * CliInitXxxx (High bit of fn is set). */
          {
            Flush(Output());
            /* if user DID NOT provide input stream, close standardinput */
            if ((globals.fn & 2) == 0)
              Close(globals.mycli->cli_StandardInput);

            /* if RUN provided output stream, close it */
            if ((globals.fn & 1) == 1)
            {
              Flush(globals.mycli->cli_StandardOutput);
              Close(globals.mycli->cli_StandardOutput);
            }

            /* If we didn't send the packet back yet, send it back */
            if (globals.mypacket)
              ReplyPkt(globals.mypacket, error, globals.mypacket->dp_Res2);
          }
          else
            /*
             * the flags weren't valid so close the Standard I/O handles if
             * they still exist.
             */
          {
            if (globals.mycli->cli_StandardOutput)
            {
              Flush(globals.mycli->cli_StandardOutput);
              Close(globals.mycli->cli_StandardOutput);
            }
            if (globals.mycli->cli_StandardInput)
            {
              Flush(globals.mycli->cli_StandardInput);
              Close(globals.mycli->cli_StandardInput);
            }
          }
          /* release the process' lock on the current directory */
          UnLock(globals.myprocess->pr_CurrentDir);
        }
        else
          error = RETURN_FAIL;         /* I have a NULL CLI! */
      }
    }
    else
      /* shelltype != 0 or 2 */
    {
      error = RETURN_FAIL;
      ReplyPkt(globals.mypacket,
               globals.mypacket->dp_Res1,
               globals.mypacket->dp_Res2);
    }
    CloseLibrary(DOSBase);
  }
  else
    error = RETURN_FAIL;

  return error;
}
```

```
long mainshellloop(struct mylocals * ml)
{
  BOOL            done = FALSE;
  unsigned char   ch, *prompt, *command, *commandname, *cmd, *cmdname;
  struct Segment *cmdseg;
  long            result;
  WORD            x;

  ml->mycli->cli_FailLevel = RETURN_FAIL;

  if (command = (char *) AllocVec(COMMANDLINELENGTH + COMMANDBUFLENGTH +
                                  PROMPTLENGTH, MEMF_CLEAR))
  {
    commandname = &(command[COMMANDLINELENGTH]);
    prompt = &(command[COMMANDLINELENGTH + COMMANDBUFLENGTH]);
    do
    {
      /* Make sure the shell looks to cli_CurrentInput for its command lines */

      SelectInput(ml->mycli->cli_CurrentInput);
      /* is this an interactive shell? */
      ml->mycli->cli_Interactive =
      /* if this is not a backround CLI, and */
        ((!(ml->mycli->cli_Background)) &&
      /* input has not been redirected to an script file, and */
         NOTSCRIPT &&
      /* this shell was not started from System() */
         (!SYSTEM)) ? DOSTRUE : DOSFALSE;

      /* if this is a script and the user hit CTRL-D, break out of the script */
      if (!((SetSignal(0L, SIGBREAKF_CTRL_C |
                       SIGBREAKF_CTRL_D |
                       SIGBREAKF_CTRL_E |
                       SIGBREAKF_CTRL_F) & SIGBREAKF_CTRL_D) &&
            (!SYSTEM) && (ISSCRIPT)))
      {
        /* if this shell is interactive and there is a prompt, print it */
        /* (unless, of course, this was created by Run, etc) */
        if (ml->mycli->cli_Interactive == DOSTRUE && !(ml->mycli->cli_Background))
        {

          /*
           * If this wasn't an example, I would probably change the prompt
           * here, probably to reflect the name of the current directory.
           */
          /* print the prompt */
          if (GetPrompt(prompt, 256))
          {
            FPuts(Output(), prompt);
            /* Make sure the prompt gets printed */
            Flush(Output());
          }
        }
        /* Get Command */
        if (FGets(ml->mycli->cli_CurrentInput, command, COMMANDLINELENGTH))
        {
          cmd = command;
          /* skip leading spaces in command line */
          while (*cmd == ' ')
            cmd++;

          /*
           * If I was bothering to deal with aliases, I would probably resolve
           * them here.
           */

          cmdname = commandname;
          x = 0;
          /* copy the actual command from the cmd buffer */
          while ((*cmd >= '0') && (*cmd <= 'z') && (x < (COMMANDBUFLENGTH - 1)))
          {
            *cmdname++ = *cmd++;
            x++;
          }
          *cmdname = '\0';
```

```
          /*
           * OK, now we have the actual command in commandname. Using it we can
           * find the actual executeable code.  The command could come from
           * several sources:
           *
           * The resident list
           * The shell (an internal command)
           * disk (from either an absolute or relative path)
           *
           * This example only looks through the resident list for commands. A
           * real shell would also try to load a command from disk if the
           * command is not present in the resident list (or the command is not
           * internal to the shell.
           */

          /* Search resident list for the command */
          Forbid();
          if (!(cmdseg = FindSegment(commandname, NULL, FALSE)))
            cmdseg = FindSegment(commandname, NULL, TRUE);
          if (cmdseg)
          {
            if ((cmdseg->seg_UC < CMD_DISABLED) ||
                (cmdseg->seg_UC == CMD_SYSTEM))
              cmdseg = NULL;
            else if (cmdseg->seg_UC >= 0)
              cmdseg->seg_UC++;
          }
          Permit();

          /*
           * if !cmdseg, the command was not in the resident list.  If I were
           * bothering to look for commands on disk, I would try to load the
           * command here.  If I has successfully loaded a command and was
           * going to execute it, I would have to set ml->myprocess->pr_HomeDir
           * to be a DupLock() of the directory I loaded the command from.  I
           * don't do this for commands from the resident list because they
           * have no home directory.
           */

          /* If we did find a command, run it */
          if (cmdseg)
          {
            /* Clear the error field before executing the command */
            SetIoErr(0);

            SetProgramName(commandname);
            ml->mycli->cli_Module = cmdseg->seg_Seg;

            /*
             * Set the I/O streams to their defaults. NOTE: StandardInput, NOT
             * CurrentInput!  The Execute command will cause nasty things to
             * happen if you use CurrentInput, since it must close that in
             * order to change the input stream to the next file. Obviously,
             * this only applies if you're using the normal AmigaDOS Execute
             * command for scripts.
             */
            SelectInput(ml->mycli->cli_StandardInput);
            SelectOutput(ml->mycli->cli_StandardOutput);

            /*
             * If I were doing redirection, the I/O handles above would be the
             * redirection handles.
             */

            /* Run the command */
            result = RunCommand(ml->mycli->cli_Module,
                                (ml->mycli->cli_DefaultStack * 4),
                                cmd,
                                strlen(cmd));
            /*
             * OK, we returned from the command.  Fill in any error codes in
             * the appropriate CLI fields.
             */
            ml->mycli->cli_ReturnCode = result;
            ml->mycli->cli_Result2 = IoErr();
```

```
        /* If I had bothered to load code from an executable file on disk,
         * I would have to unload it now.  Since I didn't, all I have to do
         * is NULL cli_Module.
         */
        ml->mycli->cli_Module = NULL;

        SetProgramName("");
        Forbid();
        if (cmdseg->seg_UC > 0)
          cmdseg->seg_UC--;
        Permit();
        cmdseg = NULL;
      }
      else
      {
        /* we couldn't find the command.  Print an error message unless the
         * command starts with a non-alphanumeric character (like a
         * carriage return) or the first character is a comment character.
         */
        if ((commandname[0] >= '0') &&
            (commandname[0] <= 'z') &&
            (commandname[0] != ';'))
        {
          PutStr(commandname);
          PutStr(": Command not found\n");
          Flush(Output());
        }
      }

      /* if you set up redirection I/O handles for the command don't forget
       * to flush and close them.
       */

      /* Make sure the proper I/O handles are in place. */
      SelectInput(ml->mycli->cli_CurrentInput);
      SelectOutput(ml->mycli->cli_StandardOutput);

      /* Get rid of any unused data left in the buffer */
      ch = UnGetC(Input(), -1) ? '\0' : '\n';
      while ((ch != '\n') && (ch != ENDSTREAMCH))
        ch = FGetC(Input());
      if (ch == ENDSTREAMCH)
        done = TRUE;
    }
    else
      done = TRUE;                  /* We got an EOF when reading in a
                                     * command */
    if (done)
    {
      if (ISSCRIPT)
      {
        done = FALSE;               /* this is a script (which could be
                                     * s:shell-startup), so don't quit, just
                                     * exit the script and set up IO
                                     * handles. */
        /* Close the script file */
        Close(ml->mycli->cli_CurrentInput);
        /* Reset the input to what we started with */
        SelectInput(ml->mycli->cli_StandardInput);
        ml->mycli->cli_CurrentInput = ml->mycli->cli_StandardInput;

        /* Restore Fail Level after executing a script */
        ml->mycli->cli_FailLevel = RETURN_ERROR;

        /* if the script created a file, delete it */
        if (((char *) BADDR(ml->mycli->cli_CommandFile))[0])
        {
          cmd = (char *) BADDR(ml->mycli->cli_CommandFile);
          CopyMem(&(cmd[1]), command, (LONG) cmd[0]);
          command[cmd[0]] = '\0';
          DeleteFile(command);
          cmd[0] = '\0';
        }
      }
    }
      }
      else
        /* Somebody hit CTRL_D in a script */
      {
        /* print the string associated with error #304 */
        PrintFault(304, "MyShell");
        /* Close the script file */
        Close(ml->mycli->cli_CurrentInput);
        /* Reset the input to what we started with */
        SelectInput(ml->mycli->cli_StandardInput);
        ml->mycli->cli_CurrentInput = ml->mycli->cli_StandardInput;

        cmd = (char *) BADDR(ml->mycli->cli_CommandFile);
        cmd[0] = '\0';
      }
      /* this takes care of some problems certain programs caused */
      if (SYSTEM && NOTSCRIPT)
        done = TRUE;

    } while (!done);
    FreeVec((void *) command);
  }
  return result;
}

long strlen(UBYTE * string)
{
  long            x = 0L;

  while (string[x]) x++;
  return x;
}
```

```
;/* RestoreShell.c - Execute me to compile me with Lattice 5.10b
LC -b1 -cfist -v -d0 -j73 RestoreShell.c
Blink FROM lib:c.o RestoreShell.o TO RestoreShell LIBRARY lib:lc.lib lib:amiga.lib
smallcode smalldata
quit
*/
/* restore the BootShell as the UserShell.  Note that this
   only switches back the BootShell, it does not unload the
   current user shell ("shell" on the resident list) as it
   is possible that some instance of it can still be running.
*/
#include <exec/types.h>
#include <dos/dosextens.h>
#include <clib/exec_protos.h>
#include <clib/dos_protos.h>

#ifdef LATTICE
int CXBRK(void) { return(0); }  /* Disable Lattice CTRL/C handling */
int chkabort(void) { return(0); }
#endif

UBYTE *vers = "\0$VER: RestoreShell 1.0";

void main(void)
{
    struct Segment
        *bootshell_seg,
        *shell_seg;

    Forbid();
    shell_seg = FindSegment("shell", NULL, CMD_SYSTEM);
    bootshell_seg = FindSegment("bootshell", NULL, CMD_SYSTEM);
    if (bootshell_seg)
              shell_seg->seg_Seg = bootshell_seg->seg_Seg;
    Permit();
}
```