CFHT Director/Agent Model
Revision 3

by Sidik Isani

2001 May 24th
Last revised: 2001 June 25th

The information here is available as HTML. At CFHT locally, the URL is: http://software.cfht.hawaii.edu/director/

Abstract

Science and engineering computer control interfaces have unique requirements. A visually appealing interface is good; a simple one is cruicial. Integration is also good, but not if traded for modularity or adaptability. Since science requirements are constantly changing, control systems must be able to keep up and be changed easily. When there are distinct hardware subsystems, it is a natural approach to develop separate *engineering* control interfaces for each, typically menu or command driven. If these programs are written as command line interfaces (CLIs), and follow a few guidelines set forth in this document, then they can be re-used completely as part of the final interface. (We refer to a conforming command line utility as an "Agent".) This can be a great savings in development costs.

With this model, knowledge of the system as a whole exists only in high level scripts which tie everything together. There is no mechanism (provided by Director, at least) allowing Agents to communicate with each other. The "Director" program functions as a command "shell", sending commands to the appropriate agent, and returning completion results (only a simple PASS or FAIL is supported) to the initiator (e.g., the script). The initiator can also be a graphical user interface, which is otherwise isolated from the control modules. Scripts can also be nested, and a graphical interface can call the high level scripts, the low level Agent commands or both. (In fact, it doesn't really know which commands are implement as scripts. It only has to pass the command line to Director.) Any interface which is capable of generating commands when buttons and menus are manipulated can be layered on top of Director. The RPM Web server (http://software.cfht.hawaii.edu/RPM/) is an example. This document focuses on creating the command-line level of the user interface.

History:
2001 June 4 - Document released.
2001 May 24 - Adapted from Director Version 2 documentation. Added information to reflect multi-agent and other new capabilities of Director 3.x. See the old document for some more history on Director and how it compares with CFHT's "Handler and Server" model in the Pegasus environment.

Contents

List of Figures

1 Introduction

How to use this manual

1. This section, and the *Abstract*, explain what the Director program is all about.
2. The *Users' Guide* contains mostly the same information you can get from Director by invoking it with the "-h" command line option. There is also some information here on how to make sure you are running the right version of the program.
3. The *Programmers' Guide* contains information on creating an "Agent" command line program.
4. The *Director* section contains a summary of new features that Director brings to the CFHT observing software
5. Any additional reference material, including where to get and how to build and install Director, is in the *Appendix*.

1.1 What's New in 3.x

The documentation for Director 2.x mostly described features of that version of Director. The current version of this document now covers the Director 3.x. Updated sections include the *Abstract*, the section on the big picture, expanded options for the `''agent restart/start''` builtin,

New sections describing libcli functions useful for writing agents, and also libcli cli_cmd for clients and descriptions of runon, clicmd, clicap, and clidup have been added. A new timing diagram illustrates how overlapping commands using the intent/go/wait model can be used to control slow moving devices efficiently.

1.2 Unix Shell Comparison

Director is essentially just an interactive shell. Like other interactive unix shells (csh, bash, etc.):

- it displays a prompt where you can type commands
- it has some commands that are built-in (like `''cd''` to change working directories.)
- it can "source" scripts or batch files of commands
- it expects a terminal capable of a few basic functions. Cursor positioning is a requirement. Color makes things nicer but is not required.
- it waits on commands and gets basic exit/completion status.

Director differs from other unix shells in the following ways:

- It does not support job control in the same way that a traditional unix shell does. Instead of every command you type invoking a new program, commands can end up being sent to already running programs, called *Agents*.
- Director has additional features which are useful to developers and support staff, including a *cloning* capability, which can be used to get an exact copy of the user's window to appear at a remote location. The *ncurses* library takes care of efficiently updating the screen, so it even works well over a relatively slow modem connection. Once a clone window is displayed, additional debugging messages (which were always in memory, just not displayed on the user's screen) can be "unhidden" in the clone's view of the window, "chat" messages can be sent to the user, and a support person can even take over session to send diagnostic commands, or control things from the remote location without having the user exit their copy of the software.
- The *Input Multiplexer* built into Director allows the Agent program to pretend it only has one source for command input (standard input) but commands may in fact be redirected from a user interface or a script by Director. Figure 1 illustrates this concept.
- An *Output Multiplexer* is most important new feature introduced in Director version 3.x. It allows multiple agents to run within the same session. By merging the command sets and feedback of multiple programs, Director can give the *appearance* of a unified command interpreter to the end user. Figure 2 illustrates this concept.

1.3 The Big Picture

Since we are using Director for image acquisition systems at CFHT, it might help to look at the full observing environment. There are three major types of data or messages passed in our system:

1. Command / Response (Director)
2. Image Transfer of Pixel Data

Figure 1: Director as an input multiplexer

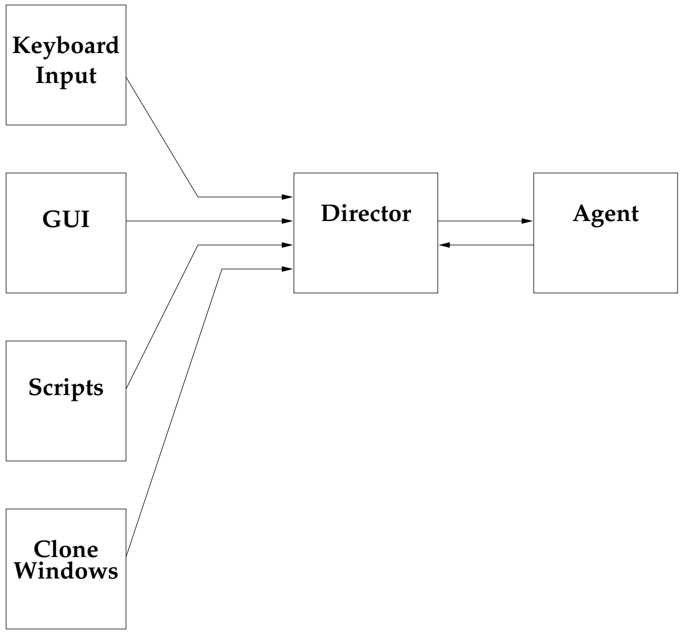
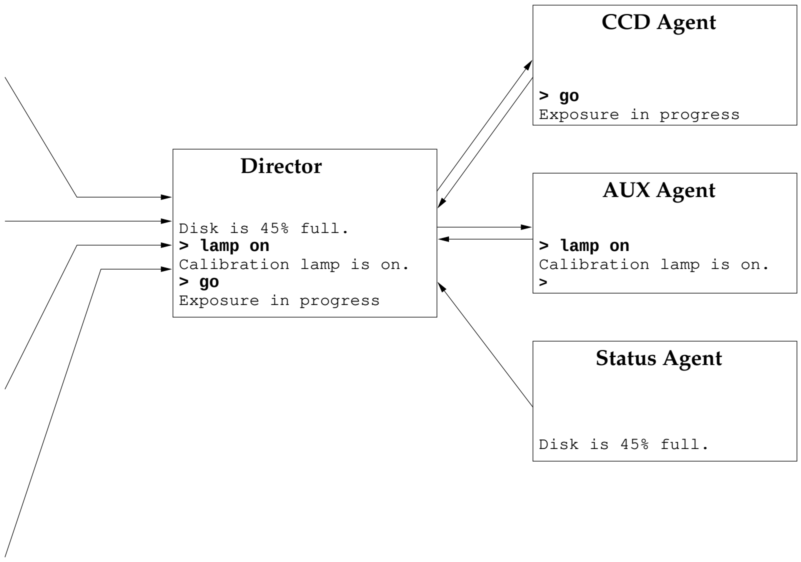


Figure 2: Director as an output multiplexer



3. State Information

Image Transfer is currently handled by writing directly from the Detector Agent (DetCom) to a file on disk (or network disk). Other than triggering the commands, *Director has no connection with Image Transfer*.

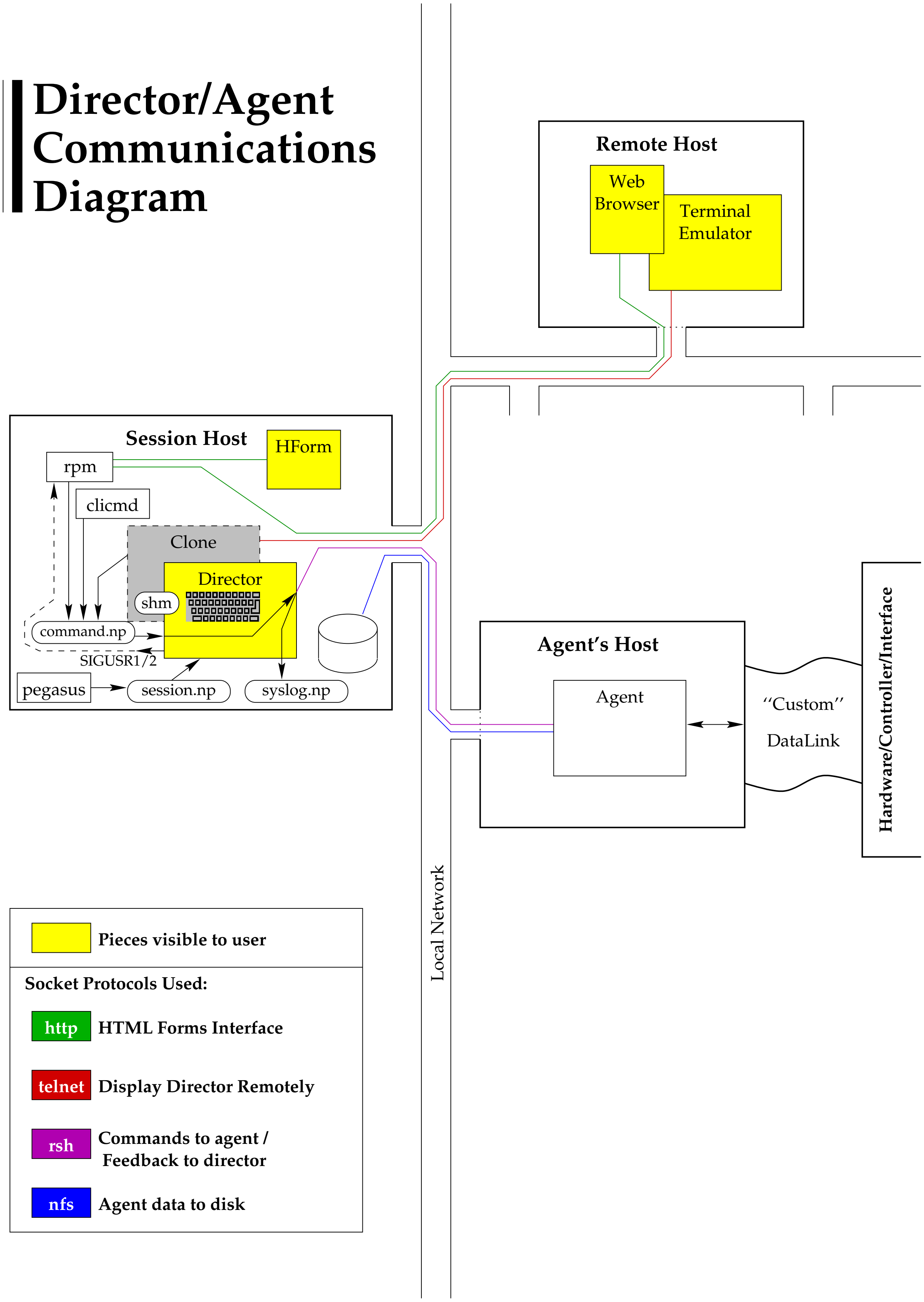
State Information will some day be handled by a Status Server, but currently we store values in "par files" on disk. This serves as a (sometimes troublesome) distributed status database. *Director has no connection with a Status Server or par files*.

In the absence of a Status Server, there are a few clunky ways to get by. One of these actually does involve Director, and is described in the section on the clicap utility.

Director only addresses the first aspect of our acquisition system: passing commands and getting a response. Warnings and errors (which Agents transmit to Director using calls like `printf()`) are merely presented to the operator highlighted or in color to draw attention, but are never *interpreted* by Director. Director also does not use any kind of error return codes. The only two possibilities for a command are for it to complete with PASS, or complete with FAIL (which halts any scripts or other commands that were waiting to be processed.)

For a detailed diagram showing how commands and responses flow in and out of Director, see figure 3. NOTE: This diagram shows some other Pegasus components which are not described in this document. A full understanding of this diagram is not necessary to use Director. See the section on *Director's features* for a summary of the practical advantages Director brings to an acquisition environment.

Figure 3: Command and Response Flow



2 Users' Guide

The information here is for those setting up environments that use Director. The actual commands available inside Director are mostly those provided by the Agents, and so are application specific. For those, the command ``help'', typed at the Director prompt, usually gives a list of available commands from Agents and scripts.

For information on invoking Director, typing "director -h" from the unix shell lists most of the information in the sections below.

2.1 Invoking Director

2.1.1 Cloning Pegasus Sessions

In an observing session where Director has been automatically started from login scripts, as part of the window session, it will appear in an rxvt terminal window in the lower right corner of the screen.

To bring up a similar window of an observing session in another session, I've created a script called `/usr/local/bin/clone` which will invoke the most appropriate version of director for you (possibly in a color terminal, if you have DISPLAY set correctly). So, for example, if you want to view the feedback from the "aobir" session, or send a message to the users of that session, type:

```
clone aobir
```

Here is the help screen, which you can get on-line on a CFHT machine by just typing `clone` by itself.

```
usage: clone <session>
 Where <session> is the name of a Pegasus session, e.g., aobir
 From another Xsession or X-Terminal:
  - Make sure your DISPLAY variable is set first.
  - A new X window will pop up on DISPLAY with a director clone

 From home, or from a telnet session not on an X-display:
  - If DISPLAY is _not_ set, clone will start in the current window.
  - Be sure that TERM is set correctly.  When logged in using
    MS Windows telnet, for example, a setting of TERM=vt100 should
    work as long as the window is big enough to not have scroll bars.

 Unless you deliberately enter the Pegasus account password in your
 director clone window, it will not be possible for you to send
 any commands to the session except for the 'say' command, which
 just broadcasts a message to all other director windows.
```

2.1.2 Running Director in your own account

To use Director to run or test your own Agent programs, make sure your PATH is set up so you get the right version. You should typically have `HOME/bin` and `/usr/local/bin` *before* `/cfht/bin` in your PATH variable if you are CFHT. Locally, I make sure that `/usr/local/bin/director` exists, which will always be a link to the most recent stable version If you need to select a specific version that you know works with your setup, then you should do as the Pegasus accounts do and create a symbolic link in `HOME/bin` pointing to the appropriate version. For example, the current stable version is 3.31, so you'd run the command:
`ln -fs /cfht/bin/director3-3.31 HOME/bin/director` just once to set up selection of this version. After that, you can start up Director in one of the following ways:

```
usage: director [director options] [ -- [rhost:]agent [agent options] ]
or     director [director options] -- [{9600,N81,see_manual...}]serialdevice
or     director            (no startup and no args, just tails log pipe)
or     director -k <user> (attach to another user's session without su'ing)
```

```
realprogram options are passed without change to the actual cli program

director options are:
    -i <n>    displays <n> info lines at bottom of screen (default is 0)
    -j        enables jumpscroll mode for slow terminals
    -v        turns on verbose mode which shows logonly and debug
    -d        turns on debug messages (also requires -v to see on screen)
    -m        used by sessions to ensure they get a master, not a clone
    -c        sends a CR character after each command instead of a newline
    -C        turn off the titlebar clock for slow connections
    -t <type> override terminal auto-sensing and TERM var (e.g., -t vt100)
    -p <str>  sets the prompt string to expect after each command completes
    -h        displays this help information
    -hf       displays summary of director-related files and resources
    -hs       displays string parsing rules used on output from realprogram
    -he       displays summary of how the environment affects director
              (hint: -he can be combined with -k<user> for testing)  (*) See appendix

All options above can also be controlled by director builtin commands.
Type 'builtin' at the director command prompt for more help on these.
```

2.1.3 Special options for serial devices

When opening a serial port, it is possible to specify optional parameters enclosed in braces , immediately before the name of the serial devices. *There must be no spaces anywhere in this string.* Multiple options can be separated by commas. Currently, the following option names are recognized. They are case sensitive:

For example, to connect to a typical 9600 baud RS232 device that has hardware flow control but is not a modem, the invocation might look something like this:

```
director -- '{9600,N81,-LOCAL,HARDFLOW}/dev/tty02'
```

For serial devices that are really terminal server "rtelnet" processes you may not need any extra options at all since the settings in the terminal server itself are the ones that matter. You may, however, need a -p option to specify what the prompt looks like, and the -c option if the program expects a CR instead of a LF as the newline character. For example, to connect to a Prolog system:

```
director -c -p '->' -- /dev/termsrv4/tty16
```

For devices which have no prompt at all, specifying an empty string for the prompt, as in:

```
director -p '' -- /dev/termsrv4/tty16
```

will cause director to assume that the device is ready for new commands after all output has been flushed and a certain timeout has elapsed. The -T option (also an option to the builtin, "agent start" in Director 3.x) can be used to tune the timeouts in this case.

2.2 Director Window Screen Layout

Depending on the type of terminal you use to run Director, you should get most of the features illustrated in figure 4.

2.3 Files used by Director

```
~/.,session.np (see also: director -he)
  Opened for reading session feedback messages from other sources.
```

Figure 4: Director Window

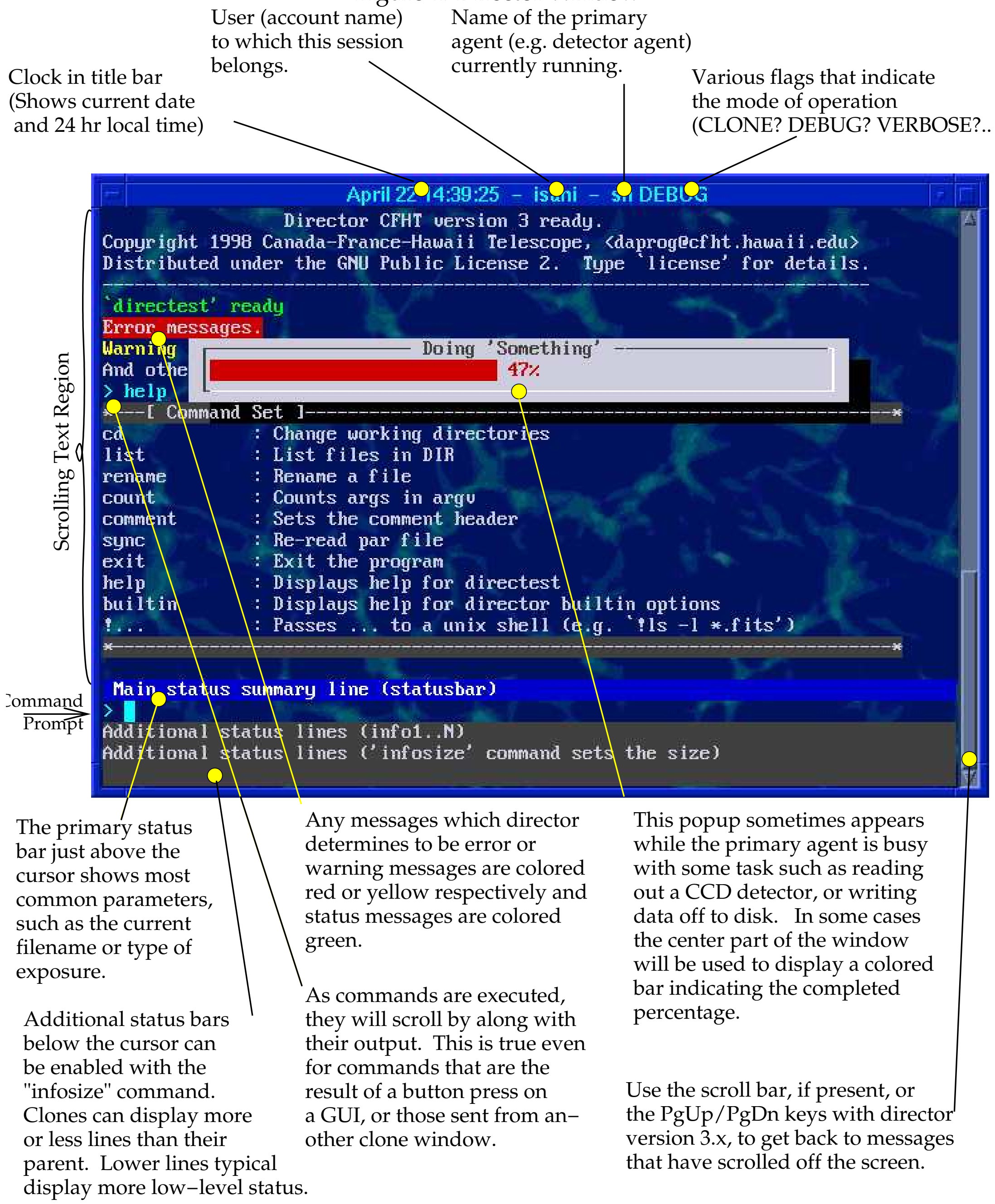


Table 1: Supported Serial Port Options

300 1200 2400 9600 19200 38400	Choose one of these to select a baud rate when the port is opened. Some systems may support other baud rates in addition to the ones listed here.
N81 N72 O71 E71	Choose one of these to set parity, data bits, and stop bits. Default is N81.
LOCAL/-LOCAL	Use this if you have/don't have modem control lines (DCD, etc.) connected. If you don't give either, the default setting of the port is kept.
SOFTFLOW/HARDFLOW	Select between software/hardware flow control. Default is both OFF. When SOFTFLOW is selected (same as XON,XOFF) the stop/start characters are set to control-S/control-Q. It is currently not possible to select any other stop/start characters in director. When HARDFLOW is selected, it is assumed that CTS and RTS lines are hardwired. Note: If you need to distinguish between output flow control and input flow control, use combinations of settings below instead.
XON/CTS_OFLOW (or use XANY instead of XON)	Select between software/hardware *output* flow control. Default is both OFF (So there is no need to give -XON or -CTS_OFLOW). On systems that have it, XANY also allows any character received to restart output (instead of just control-Q). Turning on XANY implies XON. Note: CTSXON and RTSCTS are synonyms for CTS_OFLOW.
XOFF/RTS_IFLOW	Select between software/hardware *input* flow control. Default is both OFF. Note: RTSXOFF is a synonym for RTS_IFLOW.

```
~/.director/bin/
  When Director receives a command, this subdirectory is searched first.
  A non-executable file with a matching name is sourced like a batch file.
  An executable (or link to executable, or link to clicmd) is executed
  with the same arguments given on the command line.

~/.director/sessionhost
  If this file exists as a symlink, Director will first attempt to
  'remsh' to the machine name given by the contents of the link.

~/.director/startup
  This file is automatically sourced at startup (non-clones only).
  Providing an agent on the command line (after --) suppresses this.

~/.director/history
  Command history is saved here between sessions.  The '~' is always
  replaced by the home directory of the original user starting Director.

~/.director/command.np
  Non-clones create (and remove at exit) this FIFO and read from it.
  Clones (and other external programs) must only write commands to it.
  If '-k' is present, clones use the '~' for that account.  '~' expansion
  at the command prompt also follows the '-k' user after in that case.
```

2.4 Help Screen: From Example Agent "Directest"

When you type 'help' in the Director window, you'll see something like this. The exact commands available will depend on the agent and its current mode of operation.

```
*---[ Command Set ]--------------------------------------------------------*
cd            : Change working directories
list          : List files in DIR
rename        : Rename a file
count         : Counts args in argv
comment       : Sets the comment header
sync          : Re-read par file
exit          : Exit the program
help          : Displays help for directest
builtin       : Displays help for director builtin options
!...          : Passes ... to a unix shell (e.g. '!ls -l *.fits')
*--------------------------------------------------------------------------*
```

2.5 Help Screen: Builtin Director Options

Here are some help screens about features that are handled by Director or the rxvt terminal window. These commands are reserved words and the agents themselves must never implement them.

```
*---[ Builtin Director Commands ]------------------------------------------*
termhelp      : display help screen for operating this terminal window
debug on|off  : toggles debug output (also requires verbose for displaying)
verbose on|off: toggles displaying of verbose messages
clock on|off  : controls display of titlebar clock
infosize N    : displays N lines of the info panel (N=0..9)
cancel (or ^G): cancels queued but not yet executing commands
abort  (or ^C): sends SIGINT to the agent
stop   (or ^Z): sends SIGTSTP to the agent
quit   (or ^|): sends SIGQUIT to the agent
sigsend [NAME]: sends SIG[NAME] to the agent
ready         : silently returns pass or fail (for clicmd)
say <message> : use to send messages to other users with director clones
agent ...     : starts, restarts, or stops an agent running in director
autorestart on|off: tells director whether or not to restart agents that die
source <file> : reads batch commands from <file>
*--------------------------------------------------------------------------*
```

Want more lines or bigger fonts in the text window? Here's how...

```
*---[ Terminal Window Tips ]-----------------------------------------------*
 o Resizing window will cause feedback area to fill available space
 o Shift-numpad-plus and Shift-numpad-minus change font size (rxvt only)
 o Shift-page-up and Shift-page-down scroll back the window (rxvt only)
 o Up-arrow and Down-arrow allow editing/repeating previous commands
 o Other cursor keys (home, end, left, right) do their thing
 o Most commonly used emacs bindings work also (but not ^Y!)
 o Command/file completion (usually bound to esc/tab) NOT implemented
 o Pressing Control-L will clear and redraw the entire window
*--------------------------------------------------------------------------*
```

2.5.1 Builtin `agent restart`

With Director 2.x, the `agent` command was mostly used in just one way, with `restart` as the argument. At the Director window prompt:

```
> agent restart
Restarting primary agent ...
Agent ready.
> _
```

This was mostly intended for debugging, recovering from crashes, or switching to a new version of an Agent without restarting Director. With Director 3.x, `agent restart` still works, but there are many other options. First, `agent restart` now takes an (optional) extra parameter telling it which Agent should be restarted. If none is given, the default Agent is restart. The default Agent is the determined by the following rules:

1. the most recent one started with ``agent start -D ...'', or
2. the earliest agent started, or
3. the only agent left running.

This is mainly to provide backward compatibility with Director 2.x, where the ``agent restart'' command was never ambiguous since there was only one agent. With Director 3.x, it is best to give the handle of the agent to be restarted.

2.5.2 Builtin `agent start`

The command ``agent start [options] ...'' is the recommended way to add an Agent to a Director session. The command can be placed in `.director/startup` so that `director`, typed by itself from a unix prompt, will begin a Director session without any other required parameters or aliases. The syntax is as follows:

```
> agent start [options] [hostname:]agentname
```

If the optional `hostname:` is omitted, the agent is started on the same host as Director. After this command completes, commands can be issued to this agent by using its "handle" (which, by default, is the agentname, with any version numbers removed.)

```
> agentname.somecommand someagrs ...
```

The same string could be passed to `clicmd`, described below.

The handle can be omitted if this Agent is the only one running, or if `-D` is given in the `[options]` to `agent start`. The handle name can also be omitted if a script by the name of `somecommand` is placed in `HOME/.director/bin/` (create the directory if it does not exist). Write the script so it simply calls

```
exec agentname.somecommand "$@"
```

In the future, we may create a mechanism for agents to register their command set with Director so it is this necessary, but for now, this gives the same appearance to the user.

`agent start` supports several other options:

`-D` makes this agent the (new) default Agent. There can only be one default agent at a time.

`-R` turns on auto-restart for this Agent. (A new one will be started if it exits or crashes, but not if it hangs indefinitely.)

`-H handlename` specifies the handle name which can always be used to refer to this agent specifically, by sending a command as `handlename.command ...`.

`-P promptlist` tells Director how this Agent's prompt is going to look. Use this for Agents that do not use libcli. The first character of the promptlist string is an arbitrary delimeter (choose any character *not* appearing in your prompt.) The string must also end with this character. This option is not needed for Agents that use `cli_getline()`.

`-T initial:sustained` is a hack for Agents which have no prompt at all. After a command is sent, a prompt will be "assumed" if no bytes at all are returned within `initial` milliseconds, or if bytes begin flowing but then stop for longer than `sustained` milliseconds. Use this option with caution, and only in cases where it is impossible to get an "Agent" to display any recognizable prompt. This option can be used in conjunction with `-P` for unreliable (serial, for example) connections where characters may occasionally be dropped sometimes causing the prompt to never reach Director. A long timeout can be set to prevent Director from waiting forever when this happens.

`-M debug|logonly|status|warning|error` sets the default message type Director should assume for a line with no `''type:''` prefix. The option `''-M logonly''` can be useful for agents which cannot control *all* of the messages they generate on `stdout` (some may come from a library... Channel Access is a good example.) If those extra messages should normally not be visible to the user, `''-M logonly''` hides them, but permits them to be viewed later using the `''verbose on''` command.

2.5.3 Builtin `agent stop`

```
> agent stop handlename
```

Use this to cause an agent to exit and be removed from the list in Director. If this agent needs be started again, a complete `''agent start ...''` command must be issued. Use `''agent restart''` instead for starting a new version of the Agent.

2.5.4 Builtin `agent list`

This command will print a table of all Agents in this session, along with the unix process IDs, handles, and current status.

2.5.5 Builtin `clidump`

This is another useful debugging command. It displays the internal queue of commands waiting to be processed. When scripts call other commands, this nesting is shown as a tree structure by `clidump`.

2.6 Utility Programs

As of version 3.x, Director itself no longer needs setuid-root privileges. This was required to allow Director to open `rcmd()` connections when an agent needs to run on a different host from the one running director. This is now handled with a separate utility, `runon`. There are also a set of three utilities which are useful for scripting, called `clicmd` and related programs `clicap` and `clidup`.

2.6.1 runon

Runon is based on `rsh` source code. It would be possible to have director use `rsh` (`remsh` on some systems) but not all features with signal handling and exit status may work correctly. Instead, it is recommended that you install `runon` if Agents will need to run on various hosts. This utility needs setuid-root privileges.

A man page for runon is available on the Web. Director automatically invokes runon with the correct options when an agent is preceded with `''hostname:''`.

2.6.2 clicmd

Agents are a type of *server*. The `clicmd` utility can be thought of as a very simple, non-specific *client*, because it allows any script to invoke commands as if they were separate executables. The PASS/FAIL return of the invoked command becomes the exit status of `clicmd`. It is typically used like this:

```
clicmd command [arguments ...]
```

This is equivalent to typing ``command [arguments ...]'' in the Director window at the prompt. To make it even easier to access agent commands from a shell, clicmd can also be used transparently, by creating symlinks for each command which should appear as a unix shell command. By this simple mechanism, commands to a persistent server (the agent) can appear as unix shell commands (without having to add the clicmd prefix). To configure this, follow these steps:

1. Make sure HOME/bin exists and is first in the PATH of the session. (HOME should be replaced by the home directory of the account.)

2. Make HOME/bin/clicmd a symbolic link to the clicmd program. For example, ln -s /usr/local/bin/clicmd HOME/bin/clicmd.

3. For each agent command which should appear as a unix shell command, set up a symbolic link to clicmd: ln -s clicmd HOME/bin/command.

If runon is also available on all hosts involved, the clicmd utility will even work when it is run from another host. It is most efficient, however, to run all sequencing and script on the same host as Director. Running Agents on different hosts does impose an efficiency penalty however.

2.6.3 clicap

This utility is the same as clicmd (see above), except that it captures the output of the command and sends it to the terminal where clicap was invoked, rather than sending feedback to the Director window. This can be confusing (users watching the Director window or clone windows will have no idea that a command was executed). Without a status server, clicap has been used as a temporary way of retrieving status from an agent. To do this, first create a command in the agent which prints a status value (and does nothing else.) The command with no arguments can be made to do this, or a separate command can be created. The following example demonstrates how this can be used to capture a value into a shell script environment variable.

```
#!/bin/sh
OLD_FILTER=`clicap filter`
#
#  The dialog with the Agent for the above command looks like this:
#  > filter
#  R
#  > _
#
#  OLD_FILTER now contains "R"
#
echo The old filter was \`$OLD_FILTER\'.  Moving it to \`V\'.
exec clicmd filter V
```

Note the use of exec as the last statement in the shell script. It is appropriate to exec clicmd if the clicmd is the final step in the script, and the return code of the script should indicate the success or failure of this command. For clicmd (or anything else that can fail) appearing in the middle of a script, always check for failure so that scripts exit if something goes wrong.

```
clicmd filter V || exit 1
```

2.6.4 clidup

This command is identical to clicap, except that output goes to both stdout of clicap and the director window. If a script uses clidup and this script is called from the Director window, output will be seen twice! The clidup command is only useful in some rare cases.

2.6.5 cli_cmd() in `libcli`

Most of the functions in `libcli` are for building Agents themselves. There is also a function in `libcli` which can be used to pass a command from another C program to an agent. This has the same effect as `system("clicmd ...");` but is more efficient. It is used as follows:

```
. . .
#include "cli/cli.h"

int
main(int argc, const char* argv[])
{
   /* NOTE: Do _not_ call cli_init() unless this program is also an Agent. */
   . . .
   if (cli_cmd("filter V") != PASS)
      exit(EXIT_FAILURE);
   . . .
   exit(SUCCESS);
}
```

3 The Agent (Programmers' Guide)

The following section is intended for anyone writing an agent, or wanting to adapt a command driven program to run under Director.

3.1 Functions for Agents in `libcli`

The Director package includes a C library which makes writing Agents easier. This library can manage a command table inside the Agent, handle help menus, and display the right prompt strings for Director. As such, it implements most of the protocol between the Agent and Director.

Programs which use libcli as part of Pegasus can include the header file like this:

```
#include "cli/cli.h"
```

If you are *not* building with Make.Common, create a symbolic link in your program's source directory called `cli` which points to the libcli source code and the above should work too. Linking with the library is accomplished by adding -lcli to the $(EXECNAME) line of a Make.Common Makefile:

```
$(EXECNAME): $(OBJS) -lcli
```

The following sections describe the functions provided to Agent programs by libcli.

3.1.1 Macros and Type Definitions

If not already provided by libcfht, `cli.h` will define:

`PASSFAIL` should be used as the return type for all command functions (this is a type, which can either be `PASS` or `FAIL`.)

`BOOLEAN` a type which can be either `TRUE` (1) or `FALSE` (0).

`TEMPFAILURERETRY` a macro found in the GNU C library which can be placed around all system calls to make sure they restart if interrupted by a signal.

`RCSID` a macro which is useful to include once near the top of each .c file for version tracking purposes. For example:

```
#include "cli/cli.h"

RCSID("$Id$");

. . .
```

The RCS system will insert a version number here which will find its way into the compiled binary. If RCS is not used, this macro does no harm.

`Command` a structure containing a `name`, a `function` pointer, and a one line `help` string:

```
typedef struct {
      char* name;
      Function* func;
      char* help;
} Command;
```

This structure is from the GNU readline code. Agents should contain a table of these structures, where `name` is the command name the user must type to cause `function` to execute. The table will look like this:

```
static Command comlist[] = {
   { "cd",   cli_cd,   "Change working directories" },
   { "exit", com_exit, "Exit the program" },
   { "help", cli_help, "Displays help for directest" },
   { "?",    cli_help, "Synonym for help" },
   { 0, 0, 0 } /* This terminates the command list */
};
```

Almost all agents will want to implement the commands above, along with their application specific commands. As a convention, it is suggested that internally defined command functions begin with `com_`. Functions defined in `libcli` begin with `cli_`, and for certain basic command functions, the Agent may be able to call those directly, as with `cli_cd` and `cli_help` above.

The following sections describe the functions provided by `libcli`. See also the C header file for libcli, `libcli/cli.h`, for the complete list of functions in this library.

3.1.2 cli_init()

```
PASSFAIL cli_init(const char* name, Command* comlist, int director);
```

`'name'` should be the name of your program (for example, argv[0] or the basename), but it is currently not used for anything in Director.

`'comlist'` is the command-function list, described in the previous section.

`'director'` should be set to 3 for this version of Director. The Agent should check for a FAIL from `cli_init()`, as this indicates that the required version of Director was not found.

The `cli_init` function must be the first thing inside `main()` of the Agent program, because it has to set up line buffering mode on standard output (using a `setvbuf` call) before any other code tries to print anything. Here's a typical example:

```
#include <stdlib.h>
#include "cli/cli.h"

. . .

int
main(int argc, const char* argv[])
{
   int i;

   if (cli_init(argv[0], comlist, 3)==FAIL) return EXIT_FAILURE;

   . . .
}
```

The library does not define a default minimum director version and leaves it entirely up to the agent, because some simpler agents do not use any features of Director 3. It is up to the programmer of the agent to indicate compatibility with Director 2 by calling `cli_init(..., ..., 2)`. For most cases, requiring Director 3 is not a problem though.

The `cli_init()` function may be called multiple times by the agent in order to change the available set of commands. This is done by defining multiple `comlist` tables, each of them containing a `''mode''` function. The `''mode''` function (`com_mode()`) must call `cli_init()` with the appropriate comlist, based on its parameters. For observing sessions, we have used this to implement `''mode observing''` and `''mode engineering''` commands to unmask/mask certain commands which are useful to engineering staff but dangerous if accidentally used by observers.

Functions below are only valid after `cli_init()` has been called at least once.

3.1.3 cli_help()

```
PASSFAIL cli_help(const char* arg);
```

This is another function borrowed directly from GNU readline.
If `arg` is not NULL or empty, then the one-line help strings of all matching commands are printed.
With no arguments, the full command table with help strings is displayed.

3.1.4 cli_getline()

```
char* cli_getline(void);
```

Displays a prompt string on stdout (”ok¿ ” or ”failed¿ ”, depending on result of the last command) and reads a line from stdin.
Possible returns:

- A pointer to a null terminated character string read from the user.
- NULL and errno==0 indicates that the end of file was reached.
- NULL and errno==EINTR indicates a non-blocked signal was trapped.

If a string is returned, it should be passed to `cli_execute()` (see below) and eventually `free()`'d.

3.1.5 cli_execute()

```
PASSFAIL cli_execute(const char* line);
```

Given a string obtained by `cli_getline()`, this does the work of separating the command from its arguments, looking up the command in your table and running the corresponding `com_...()` function from the table. The arguments found in `line` (if any exist, after the command name) are passed to the `com_...()` function, and the return of the `com_...()` function becomes the return of `cli_execute()`.

3.1.6 cli_arg1()

Returns the same string in args, but possibly with ”” or ’’ removed from the whole string. At the very minimum, this function strdup's the string for you, so you should always free the memory it returns when done.

3.1.7 cli_argv()

```
char** cli_argv(int* argc, const char* args);
```

`argc`, if not `NULL`, will be used to store the number of arguments found in `args`.
`args` must be a string of characters or a zero length string (but cannot be `NULL`).
The function returns an vector of arguments with the last element set to `NULL`. All of the arguments are allocated in a single string and the returned array just contains pointers into that string. Even when there are no arguments, you should always free the memory allocated to `argv` itself, and also to `argv[0]` whenever there were arguments. Example:

```
PASSFAIL
com_something(const char* args)
{
   static char** cargv = NULL;
   static int cargc = 0;
   . . .

   if (cargv) { if (cargv[0]) free(cargv[0]); free(cargv); }
   cargv = cli_argv(&cargc, args);
   . . .
}
```

3.1.8 cli_bool()

```
BOOLEAN cli_bool(const char* arg);
```

This call is derived from libcfht's `cfht_tob()` function. It takes a string and returns the string's boolean value. In addition to the usual `TRUE/FALSE` and `YES/NO` pairs, many other possibilities are handled.

The following strings return TRUE:

```
"True"        "Yes"        "Up"        "Automatic"
"Enable"      "In"         "1".."9"    "ON"
"OPen"        "COlumn"
```

Case is not significant in the tests. In the above strings, uppercase is used to denote the characters used in determining if a string is `TRUE`. Any string whose first characters do not match any of the above is considered `FALSE`.

`cli_bool()` is usually called on an argument in the array returned by cli_argv() or the return of cli_arg1().

3.1.9 cli_strdup()

```
char* cli_strdup(const char* orig);
```

See man page for strdup. VxWorks 5.2 doesn't have a strdup, so use this call instead for code that needs to be portable.

3.1.10 cli_system()

```
PASSFAIL cli_system(const char* command, const char* args);
```

Exec's a command with `/bin/sh` and passes it `'args'` as arguments. NOTE: This is not very secure at the moment. A ';' in the args causes the shell to execute the rest of the line as a new command. The two strings `'command'` and `'args'` are just concatenated with a space in between, so you can leave either of them empty or NULL if you don't need both.

3.1.11 cli_getcwd()

```
const char* cli_getcwd(void);
```

Returns a string with the current working directory.

3.1.12 cli_gethome()

```
const char* cli_gethome(void);
```

Returns the current effective user's home directory

3.1.13 cli_cd()

```
PASSFAIL cli_cd(const char* args);
```

Use this to implement your "cd" command to ensure synchronization with director's concept of the current working directory. Tilde expansion will already have been done on args when running with director.

3.1.14 cli_signal()

```
void    cli_signal(int signo, PFV handler);
void    cli_signal_block(int signo);
void    cli_signal_unblock(int signo);
BOOLEAN cli_signal_isblocked(int signo);
BOOLEAN cli_signal_event(int signo);
BOOLEAN cli_signal_peek(int signo);
```

These calls are identical to `cfht_signal_...()` when libcfht is not present. Refer to the on-line man page for cfht_signal() for more information on these functions.

3.2 Stand-alone Testing

An important requirement of the design is that the agent program MUST be able to function without the Director. Not only is this important for a fall-back mode, but it also makes for easy testing of components. When an agent is executed by hand on a terminal, and not inside Director, the output will not appear exactly as the user was meant to see it (no colors and statusbar messages scroll instead of going to the right place, etc.) Similarly the keyboard input may not be handled as gracefully (line editing functions and history will be lacking) but the program should still operate. Be sure to test this!

3.3 Restrictions on Agent

You MUST follow these requirements in order for your Agent program to function correctly when running under Director. To help comply with these points, a library has been assembled called *libcli* which is distributed with the Director source code. See the file `libcli/cli.h` for interface documentation. You do not *have* to use this library in order to be compliant.

Also included with the Director source code is a skeleton Agent program called `directest.c`. It demonstrates how to use most of the functions provided by libcli, including a method for installing a timeout while at the command prompt (lines of code relating to this are commented with /* LPS */). When this timeout expires, your agent could go off and do some "Low Priority Stuff". If you don't need this, it's better just to leave out all the LPS lines!

- Runs on machine with most direct access to hardware that it controls.
- If it needs it, must be able to access filesystem with the data (through NFS probably). This is handled as a completely separately issue.
- Must be able to run on a "dumb" terminal (do not print any terminal-specific escape codes for bold, underline, etc.) Legal control characters are listed in the next section.
- Must not change working directories unless user issues a "`cd`" command. It's best to use the `cli_cd()` call in libcli. (The current working directory issue may become important if Director is expanded to handle multiple agents at once.)
- Must operate stdout in line-buffered mode. This is taken care of by calling libcli's cli_init() call, or steal code that calls setvbuf() from there. Unbuffered modes (typically used on stderr) will work also, but will be less efficient.
- Must echo "> " ("ok> " or "failed> " when it has status of a previous command to report) as prompt character and flush this to stdout before reading a new line from stdin. (Taken care of by just using libcli's cli_getline()/cli_execute() calls.) Avoid having regular output of the program echo these prompt strings because it will confuse Director into thinking that the command has completed. Note: If it is possible for your program to be interrupted by a signal while waiting for a command, and if in that situation a new prompt gets displayed, be sure to strip the "ok" or "failed" from the prompt when it is re-displayed. Again, libcli's cli_getline() takes care of this for you.

- No subprompts. There cannot be any submenus, including (Y/N) confirmations or prompts asking for a filename or missing parameter. Director assumes that the agent is always waiting ready at the top level command prompt (or busy executing a command.) So, for example, if you had a command called `download` that required a filename argument, and someone just typed `download`< *enter* >, the following would NOT work:

```
   ok> download
Enter filename> _
```

  Instead print a usage message and force the user to type the whole command again (or press the up-arrow and edit their old command)

```
ok> download
Syntax: download <filename.lod>
failed> download full
. . .
ok>
```

- No toggled arguments. Try to avoid commands that toggle a current setting, especially if it is desirable to later have that command used in a script or by a GUI. The script does not know the previous state and should just be able to say things like "`bit on`" and "`bit off`".

3.3.1 Note about Unix Signals

Director builtin commands like `stop`, `break`, and `quit` will send signals to the Agent (so it need not be at a command prompt to receive these commands). If your program has forked any child processes, they will receive the signal also, because Director sends the signal to the entire process *group*. It will often be undesirable to have child processes receiving these signals (for example, if the child has exec()'d another program, it will dump core on a SIGQUIT unless it has a special signal handler installed.) The recommended way to keep child processes from seeing these signals is to place the child in its own process group immediately after the fork(). The POSIX call setsid() will do this for you.

The cli_system() call in libcli uses this method to prevent commands like "ls" that your program may want to run from seeing these signals. See libcli/cli.c for an example.

3.3.2 Note about child processes

There may be other incompatibilities if you mix various ways of fork()'ing and exec()'ing processes with other methods like popen() or system(). These are not really related to Director, but can occur in any Unix application that mixes these types of calls. The best advice to avoid these kind of problems might be to see if there is a way to keep your Agent program single-threaded and as simple as possible.

3.3.3 Note about stderr and stdout

Director reads both stderr and stdout from the agent. If none of the special strings (see *appendix*) are found at the beginning of the line, then lines read from stdout will be displayed in the normal color, and lines read from stderr will be displayed and logged as warnings (in yellow). Generally you should print everything to stdout, but it is ok to use C library calls like perror() which automatically print on stderr.

3.3.4 Note about line buffered mode

Director reads from the agent in line buffered mode. This means nothing will be displayed until a newline is received, so it is important to remember to put \n's at the end of all your printf()'s. Even if you are printing on stderr, you must assume line buffered behavior.

3.4 Legal Control Characters

```
\t   Moves to the next tab stop on 8 column boundaries
\a   Beep. (Bell also rings when an error message is displayed.)
\r\n Treated as a "\n" (i.e. Director handles DOS newlines properly)
\r   Treated as a "\n" if after text, ignored if after empty line
```

All other control characters are changed to the '*' character by Director, to avoid interfering with the ncurses library, which has control of the screen (and therefore should be the only one sending escape sequences to the terminal.)

3.5 Requests versus Status

Since agents are persistent (like servers, in the client-and-server model used in CFHT's Pegasus), it is possible to keep state in internal variables. State variables describing things like the current exposure type or filename are straightforward. A variable is created in the program which is changed when the corresponding command is received. Other states involving things like positions of mechanical devices can be more tricky.

There are many cases where the requested state and the actual state (sensed from encoders, for example) are not exactly the same. Normal reasons for this could be that the device is still on its way to a newly requested position (this only happens when movements are allowed in parallel; see the next section) or the device has reached a slightly different value than the target, within an acceptable tolerance. Abnormal causes for a descrepancy could include things like hardware failures. In summary, if the answer to *any* of the following conditions apply:

- Does the device or setting take time to move/change, and is there a separate command from the one which accepts the value which waits on motion to complete?
- Does the device not always end up *exactly* at the value commanded (for example, because of encoder precision)?
- Can there be a sensed value which could disagree with the requested value for any other reason (hardware failure? moved by other means?)

Then it is best to keep track of the last requested `VAL_REQUEST` and the sensed value `VAL_SENSE` separately. Additionally, there are cases where motion should not begin until another operation has completed. In this case, it may also help to keep track of whether `VAL_REQUEST` has been sent on the hardware or not, but copying it to `VAL_SENT` (or setting and clearing a flag). This is case of separated intent/go commands, described in more detail below.

3.6 Simple Move Commands

There are cases where a single straightforward command to move a device is a appropriate. In this case, there is no need to keep track of `VAL_REQUEST`, `VAL_SENT`, and `VAL_SENSE` separately. However, the following conditions should be met:

- The device moves relatively quickly, or it is acceptable for the entire *system* (Director and all agents) to wait while the device moves.
- The only chance for failure is during the move.
- The only thing that can move the device is a command to the agent (or there is no way to sense the current position anyway.)

For an example, lets consider a mirror which can be moved in and out of the beam. A dialog with the agent would look like this for the case where the mirror was currently in the beam, and requested to go out of the beam:

```
(... mirror is currently in beam)
ok> mirror out
progress: Please wait ... moving mirror out of beam.
status: Mirror is out of the beam.
ok> _
```

The final OK signifies both that the requested position was valid, and that it has been reached. If a second request for "mirror out" followed this one, the agent should return ok (PASS) as quickly as possible.

```
ok> mirror out
logonly: Mirror is out of the beam.
ok> _
```

Note the choice of "logonly:" for redundant messages, versus "status:" when a true new position has been reached.

Error conditions which could occur for a simple move command include invalid syntax and hardware failure. The return of the command is FAIL for both, so the user will rely on error messages printed by the agent to distinguish. Here are a couple of example dialogs with the agent with an error condition:

```
ok> mirror otu
error: 'otu' is not a valid mirror position.  Choose from 'in' or 'out'.
failed> _
```

Or if the hardware fails:

```
ok> mirror out
(... gears grinding ... timeout ... etc.)
error: Servo motion timeout/limit switch hit/etc.
failed> _
```

3.7 Intent/Go/Wait Move Commands

For many cases, the model above is not sufficient. One reason is a limitation/feature of Director: while one agent is processing a command, no other agent can be busy at the same time. ("Busy" only means busy from Director's point of view. If motion is in progress, but the agent is displaying a prompt ready for a command, then Director will begin the queued next command.) A single agent which has control of hardware that can do multiple things in parallel also benefits from a different model. One way to split things up is to provide commands for three different types of events, which may occur at three different points in time:

1. A series of "intent" commands configures the user's setup for the upcoming observation or experiment. If no new intent command is issued, a previous value is assumed to be "sticky" and remains in effect until a new intent command changes it.

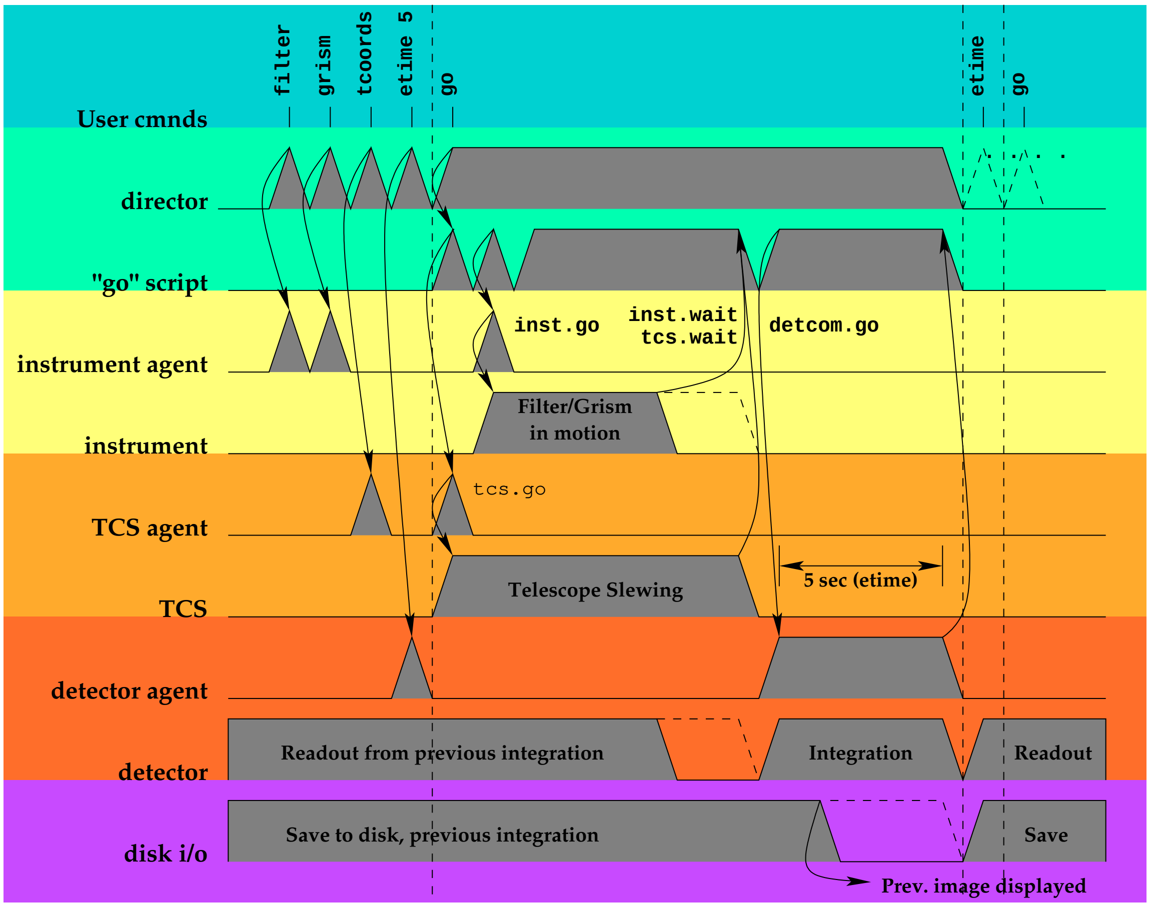
2. At a possibly later point in time, a single "go" command (per agent) is issued when it is safe to begin motion(s).

3. A final "wait" command (per agent) issued at the point in time when motions must be complete. (For example, before opening a detector shutter to take a picture.) The "wait" is the only command that is "blocking". "Intent" and "go" commands return immediately.

Figure 5 shows how a sequencing script (visible as a Director command to the user) can allow actions of various agents to proceed in progress. The example shows an observation taken with an instrument with a grism and filter, selection of telescope coordinates, and the integration and readout phases. A readout from a previous integration is still completing as the current integration is being configured.

The following sections describe in more detail what an agent must do when it receives each type of command, intent, go, and wait.

Figure 5: Timing Diagram of Overlapping Go/Wait Commands



3.7.1 Intent Commands

There should typically be a separate intent command for each possible setting or axis of movement. Combining parameters into one command is acceptable when they are always given together, from the user's point of view. Coordinate pairs is a common example. If there is any case where a user might want to issue a value separately, don't require them to issue it as part of a command that changes other values.

1. The first thing the agent should do when it receives an intent command is compare the "new" value with the previous `VAL_REQUEST`. If they match, simply return PASS.
2. In case there is already a previous motion in progress (a go has been issued, but no corresponding wait) the agent must either wait on the previous move first, or adjust the new target value to the one it just received, depending on what the hardware allows. NOTE: Having an extra "intent" command between a "go" and "wait" is not the normal flow of events, but the condition should be detected and handled, since it may sometimes occur when sequences are aborted in the middle and restarted. If the "wait" command is written as described below, this entire step could be taken care of by simply calling "wait" inside the intent command.
3. Next, if the value exceeds hardware limits or contains syntax errors, this should cause immediate failure of the intent command, if possible. It is also acceptable to trap limit errors, and even syntax errors during the "go" or the "wait", but it can make debugging more difficult.
4. Finally, the intent command must store the new value in `VAL_REQUEST` and return PASS.

Note that "intent" commands only manipulate program variables and do not communicate with the device.

3.7.2 Go Command

After a series of "intent" commands, an agent should expect a single "go" which puts all of them into effect. Generally, the "go" command compares each last known `VAL_REQUEST` with the last known `VAL_SENT`. For any that do not match, it does the following:

1. Transmit `VAL_REQUEST` to the hardware.
2. Enable servo motion, if applicable.
3. If the above two steps are successful, copy `VAL_REQUEST` into `VAL_SENT`, otherwise return FAIL.

If the agent sees two "go" commands in a row, the second one has no effect, because all the `VAL_REQUEST`s == `VAL_SENT`s.

3.7.3 Wait Command

The purpose of the "wait" command is to ensure that the user's requested values (`VAL_REQUEST`) match those of the actual hardware, `VAL_SENSE`.

1. Inside "wait", make a call to "go". Even if a "go" had already been issued by a script, if it is implemented as described above, it only has to go through its table of items to quickly determine that all `VAL_SENT` are already equal to `VAL_REQUEST`.
2. Next, the "wait" should do whatever is needed to ensure that the table of `VAL_SENSE` values are up to date. In the case of a callback system, there is nothing to do here. In a polling model, a request will need to be made to the hardware to read current status and store it in `VAL_SENSE`.
3. For each device in the agent's list, do the following:
4. If device is still busy, set a "$busy_flag$" and continue through the list.

5. If device is not busy, check if `VAL_SENSE` and `VAL_REQUEST` are equivalent (or within tolerance.) If it is within tolerance, check if `VAL_SENSE` is a new value by comparing it to the last value reported to the user (`VAL_REPORTED`). If they differ (by a tolerance also, if you don't want to report every slight drift in position), set `VAL_REPORTED` to `VAL_SENSE` and print a status: message something like this:

   ```
   status: Device has reached VAL_SENSE
   ```

   If there is a descrepancy, print an error message and possibly suggest a new intent command which could get around the condition:

   ```
   error: Device requested to go to VAL_REQUEST is reporting VAL_SENSE.
   error: Use the command ``device FAKE'' to ignore this problem.
   ```

   And then return FAIL.

6. Next if $busy_flag$ is not set, it means all device have reached there requested positions. Wait should now return PASS.

7. If $busy_flag$ *is* set, but if wait is called as "wait -poll", a FAIL should now be returned. This gives a non-blocking way to test if things are stable. An example of how this is useful is given in the main command loop of the example agent below.

   A normal "wait" should now delay (wait for an event or sleep for some polling interval) and then repeat step 2.)

3.8 Relative Moves

Scripts which call "go" and "wait" multiple times can cause potential problems for relative offsets. To avoid this, always convert relative to absolute in the intent step inside the Agent. Script should be careful not to re-send relative offsets. If agent converts it right away, it doesn't need to be concerned with this.

3.9 CLI Style Guide (under construction)

3.9.1 Log Message Types

The use of error/warning/logonly/debug/status messages should follow the guidelines set forth in the cfht_log man page. Error messages are used slightly differently, however. In handlers they are typically used when the handler is about to give up and exit. Instead error messages should be used as a final message that an individual *command* failed, in the case of a cli. A typical failure sequence would then have a bunch of warnings (and logonlys) followed by a single error message.

debug: Not usually generated except in "debug on" mode. Useful to track problems with the software itself.

logonly: Not usually displayed except in "verbose on" mode. Mostly useful in tracking down problems with the instrument or elsewhere.

warning: Displayed in yellow. Something is not 100% normal and may cause bad results down the line. (But request itself hasn't failed yet.)

error: Displayed in red (and terminal beeps/de-iconifies). Top level command request has failed. Read warnings above for details.

status: Displayed in green. Usually a confirmation that a command succeeded.

For a complete list of message types, see the appendix *or type 'director -hs'.*

3.9.2 String formatting

When echoing strings or values of parameters back to the user, *opposing* single quotes are recommended. For example:

```
error: The file ‘foobar’ does not exist.
```

4 The Director

4.1 Overview of Benefits and Features

NEW FEEDBACK WINDOW Replaces cattail (tail -f) and attaches to CFHTLOGU for log messages. Color handled by ncurses. Director on non-color terminals uses bold and reverse video to highlight things.

VIEW IN MULTIPLE PLACES AT ONCE Keeps track of recent messages in shared memory so other Director processes can listen in and provide another view, even on a different terminal-type, or with a different subset of messages! (i.e. one window shows debug messages, the other has them suppressed.)

NON-SCROLLING TEXT REGIONS The terminal's window manager title bar, a status bar above the command prompt, and additional "debugging" info bars below the command prompt are easily configurable by echoing simple text messages to the Director window (through cfhtlog, or directly from an agent's standard output).

CLOCK, DISK FREE, DETECTOR TEMPERATURE All of these displays are now available in the Director feedback window rather than as separate windows. Reduces clutter on screen and simplifies xtemp/xdisk applications.

CAN BE "PARENT" TO MOST COMMAND DRIVEN PROGRAMS The Director itself plays the roll of OCS for the agent that runs inside of it. The agent can be (1) a local process on the same machine, (2) a process on a remote machine, (3) a process talking through an RS232 port or terminal server port. (UPS, Prolog, ... doesn't have to be Unix.)

TEACHES SYNTAX FOR SCRIPTING The flow of information in Director/Agent provides a tool for training users on the command syntax as they use the system, even if they are only using the GUI! This is because every GUI action translates to some cli command which they will see echoed in the feedback window.

SAFE COMMAND INTERFACE FOR THE USER At the prompt window, users will find a well behaved area into which to type their commands, easy line editing key bindings and command history, a backspace key that works, etc.

INTERRUPTS CAN BE DELIVERED USING A COMMAND The prompt area of the Director window is always "live", even if the system is busy. This allows things like sending a signal with a command (e.g., user types "abort(enter)" and SIGINT gets sent.) A GUI could have an "Abort" button without any special logic behind it to send the command. (Echoed to Director as if it were any other command.)

TWO OPTIONS FOR SCRIPTING The easiest way to write a script for Director is to type the commands into a kind of "batch file" and then "source" it at the Director prompt. The commands will be processed one-by-one until one fails or the end of file is reached.

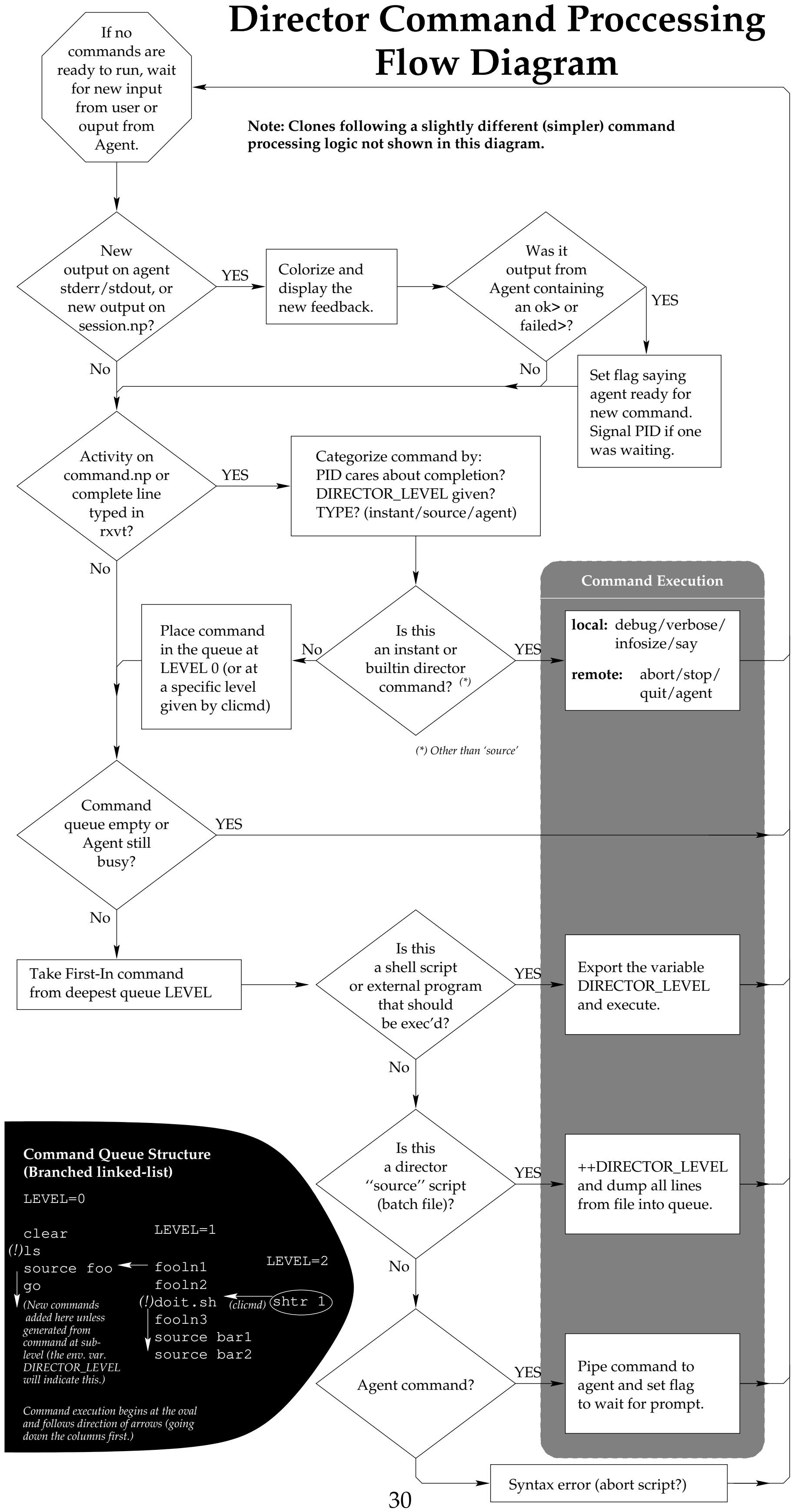
The second way to write a script assumes the user is familiar with at least some kind of unix shell or script language. Using the utility "clicmd", commands can be passed to the Director. The exit code of clicmd reports on the success or failure of the command. This method provides the same flexibility that users have always had by being able to call Pegasus handlers, but rarely used (perhaps we are guilty of not providing adequate examples?) In fact, users can write a script that combines calls to handlers and clicmd. Use the second option if you need things like loops and conditionals.

CCD HOST LOGS MOVED TO SESSION HOST Easy way of getting ccd logs to session host's roll file. All important messages now in one location.

4.2 Program Flow Chart

Figure 6 is a flow chart outline the logic implemented in the C++ code of the Director program.

Figure 6: Director Flow



5 Appendix

5.1 Terminology

Server Constantly running process that talks to some hardware or provides some other “service” for client programs (which come and go in the Pegasus model). There are usually always servers running for telescope control, CCD control, filter wheels, etc.

Handler Our name for the clients. A custom handler exists for each Pegasus server. All communication with the server, and also filling in of fits cards is done using the handlers.

Traffic/OCS The (custom) communications scheme used by Pegasus handlers to send commands and get responses back from servers.

Agent Really the same thing as a server. Just made up “agent” to have a way to distinguish from Pegasus servers. The difference is, there is no device specific “handler” per se.

Director “Parent” of the agent program. Takes care of accepting command requests on behalf of the agent from various sources and forwards them on. Displays colorized response messages in “Pegasus feedback window.”

Clicmd Plays role of handler for the Agent, but is general to all agents (it only knows how to send a command and get back PASS/FAIL.)

Pipes, Named Pipes, and Sockets A Pipe feeds the output of one program into the input of another. A Named Pipe is an empty file in the filesystem that is opened for reading by one process and written to by other process(es) to establish a pipe between programs that are started up independently. Sockets are required to establish pipe connections between different machines on the network.

remsh Remote shell protocol allows a process on one machine to start up something on another and gives it input and output pipes to communicate with the remote process.

Rxvt A terminal emulator window (like XTerm) that supports VT100 escape sequences and ANSI color escape sequences.

5.2 Environment and Director

```
DIRECTOR (this is version '2')
  DO NOT SET this externally!  Director sets it to an appropriate value
  so agents know what version of director they are running under.

SESSIONHOST
  If unset, director will set this to the current hostname.  If it is set
  but it doesn't agree with the current hostname, director will issue a warning
  and keep the value you have set. The symbolic link file ~/.director/sessionhost
  overrides this.

DISPLAY
  If unset, director sets this to SESSIONHOST:0.0.

USER and HOME
  After a userid for a director session has been determined (either that
  passed with '-k' or getuid() if no '-k') director overrides these two
  variables to agree with information from the passwd table. The original
  settings of these variables have no effect on director or its agents.

PATH
  User's current path is sent to agent as-is.
```

```
LINES and COLUMNS
  Ignored (set to 0 to force ncurses to determine window size on its own.)

CFHTCONFPATH
  Defaults to /cfht/conf.

CFHTLOGS
  Log messages from agents running within director are also written here
  using cfhtlog() calls.  The message type is determined from the first
  few characters in a line printed by the agent.  For example, a string
  starting with ‘error:’ is written as CFHT_ERROR, etc.  See also ‘-hs’.
  Defaults to /tmp/pipes/syslog.np.

CFHTLOGU
  If this file (fifo) does not exist, or if no other process is currently
  reading from it, it is opened.  Messages written to this node by other
  programs are echoed in the director feedback area.

FFTEMPLATE
  Passed on to the agent.  If not found, defaults to ~/.,template

When the agent runs on another host, these are the ONLY variables it
should expect to have defined.
```

5.3 String Parsing Rules

```
String parsing rules: ('^'->beginning of a line)

OUTPUT FROM AGENT:
^status:     displays green,  logs as CFHT_STATUS
^logonly:    displays brown,  logs as CFHT_LOGONLY
^debug:      displays brown,  logs as CFHT_DEBUG
^warning:    displays yellow, logs as CFHT_WARN
^error:      displays red,    logs as CFHT_ERROR
Any line on stderr is at least a CFHT_WARN, so C agents can use perror().
All other lines are not logged, but display in various areas of the screen:
^statusbar:  creates status bar (if needed); displays text in reverse video
^titlebar:   displays text in window title (if TERM=xterm or rxvt only)
^infoN:      (where N is 1..9) displays on info panel if infosize >= N
^instant:    displays text in user prompt area until command completes
^*           displays with highlighted background (including the *)
^ok>         command completed ok *handled internally by cli_getline()
^failed>     last command failed  *handled internally by cli_getline()
\a(bell)     rings the terminal bell immediately (and eats the character)
...          all other lines scroll in white in the feedback area
             long lines are word-wrapped to fit the current window size

INPUT AT PROMPT, COMMAND PROMPT, OR FROM SCRIPT:
 ~ or ~user  basic shell-style tilde-substitution is done by director
builtin      (where ‘builtin’ is a director builtin command) runs immediately
other        (where ‘other’ is any other command) put in queue for agent
@cmd         runs cmd as above but does not echo the command itself to window
-cmd         runs cmd as above but does not abort a script if this command fails
{PID}cmd     runs cmd as above; when it returns director signals PID with:
               SIGUSR1 if command returned PASS (prompt was ^ok>)
               SIGUSR2 if command returned FAIL (prompt was ^failed>)
               *handled internally by cli_cmd() or the shell utility ‘clicmd’
```

5.4 Licensing

Director is free software. It is distributed under the terms of the GNU General Public License version 2. If you modify it, please make sure this is clearly advertised when the program starts up.

5.5 Requirements, Downloading, Building, and Installing Director

5.5.1 Requirements under Linux

Most Linux systems already contain the necessary prerequisites for building and installing Director.

5.5.2 Requirements under other unix platforms

Director has been tested on HP-UX 9 and 10, SunOS 4.1, Solaris 2.x and Solaris 7, but only after installing a bunch of GNU software to make these systems look more like a Linux system. Most importantly, GNU make, the GNU C compiler, and the ncurses library (defaults on Linux, but not many other unix platforms) are *required* to build Director. If any of these are installed in a place other than `/usr/local`, be sure to edit `Make.Common` so Director's Makefile system can find them.

You'll want to install these items in the order given below. All GNU packages are available from ftp://ftp.gnu.org/pub/gnu/. The versions indicated are the versions that we are using. Most likely later versions (and possibly some older versions) will work as well.

GNU make Install GNU make first. Some of the Makefiles may give errors versions of GNU make older than about 3.74, so upgrade to 3.79.1 or later if you receive errors related to a ';' in the wrong place. Make sure GNU make is in your path *before* other make's like /bin/make.

GNU fileutils This is only needed for the BSD compatible **install** program. If you already have a BSD compatible install program, edit Make.Common (in the CFHT stuff) to make sure it can find it.

GNU C Compiler All of the software below has been tested to work correctly when compiled with gcc-2.7.2. Director and DetCom also compile correctly with gcc-3.0, and some versions of GCC in between.

ncurses Some of these patches may be needed to get ncurses to work correctly with rxvt. In our setup, I simply replaced the entry for rxvt with `vt100` plus `emca+color`. I commented out the old rxvt entry in ncurses-4.1/misc/terminfo.src and replaced it with:

```
rxvt|xterm-color|reduced xterm terminal (X Window System),
        kf1=\E[11~,kf2=\E[12~,kf3=\E[13~,kf4=\E[14~,khome=\E[7~,kend=\E[8~,
        use=vt100, use=ecma+color,
```

After that you have to do a "make install" in the misc subdirectory to install the new terminfo. One way to be sure things will work correctly would be to use our version of rxvt and our version of ncurses, as they are known to work together. The tar files can be downloaded from the http://software.cfht.hawaii.edu/megacam/ page. Then you won't have to worry about patching anything at all.

rxvt-2.19 or a modern xterm Once you have installed Director, you may want to experiment a bit to get the best color support, because it makes the screen a lot more read-able. Here are some things to try:

- See if there is an existing version of 'rxvt' on your system, or build CFHT's patched version (which has some nice features for dealing with colormapped displays where applications like Netscape or SAOImage have taken all the colors.) We use Director in an rxvt, and Director has special code to detect rxvt's and work correctly in this type of terminal. Our version is based on rxvt-2.19. Later versions exist but may have more bugs than our patched 2.19 version.
- See if there is an 'xterm-color' or 'color-xterm' terminal installed.

- Modern X11R6 installations usually have ANSI color support in the normal 'xterm' binary, but you might have to tell director to check for color support by choosing the right terminal type. Depending on your ncurses version, try any of these:

```
director3 -t xterm-color ...
director3 -t color-xterm ...
director3 -t rxvt ...
```

If one of these terminal types turns on color support for you, you could also make that the default by putting it in the COLORTERM environment variable.

5.5.3 Requirements under Windows NT

I have also managed to compile rxvt, ncurses, and Director under Windows NT using the cygwin32 GCC from Cynus Solutions (now RedHat). With some tweaking, Director should run properly under Windows NT (and possibly Windows 95). Without named pipes, however, its functionality will be somewhat limited on that platform.

5.5.4 Downloading

director-and-detcom-3.31.tgz [0.60MB] contains full source to both Director and DetCom (if you don't need DetCom, comment "detcom" and "fh" out of the top level Makefile.) Source for supporting libraries and Makefiles for GNU make and GCC are also included.

rxvt-2.19-cfht.tgz [0.35 MB] is a patched version of the rxvt color text terminal. (Our patches include gracefully dealing with a full colormap.) Current releases or X11R6 have an `xterm` which supports ANSI color, so you may not need this program.

A mirror of the man pages for rxvt are available on this site, and are also included with the tar file, along with a "vga.pcf" font. Again, all of this may already exist on other Linux installations, and under modern Solaris, HPUX, and others, xterm will probably work fine. If it comes up without color, try `''director -t xterm-color ...''`.

ncurses-4.1-cfht.tgz [1.35 MB] If there is already a pre-existing "ncurses" library installed on your system, use that one instead. Linux distributions should have this already. If there is no ncurses installed, you might try the latest version (ncurses-5.2) rather than this patched one.

Links to some documentation on ncurses and its libraries are included in the tar file, and at on this site.

5.5.5 Building and Installing

1. Choose the directory tree where you want to install everything. This directory will be referred to as `PREFIX` from now on. Some examples are:

   ```
   /usr/local
   /cfht
   /opt/cfht
   /home/bob/cfht
   ```

   WARNING: At the moment, /usr/local/cfht/ and /local/cfht/ in particular do NOT work as PREFIX, unless you create a symbolic link in root (/cfht) pointing to this directory.

2. Create directory `PREFIX` (`/opt/cfht` in this example) and untar `director-and-detcom-X.YY.tgz` there. This will create `PREFIX/src` with directories for projects `cli`, `fh`, and `detcom`. Example:

   ```
   cd /opt/cfht ; gzip -dc ~/director-and-detcom-3.31.tgz | tar xvf -
   ```

When you are finished, you should something similar to this:

```
/opt/cfht/src/Make.Common      (Shared Makefile stuff)
/opt/cfht/src/ThisIsTopLevel   (Marker used by Make.Common)
/opt/cfht/src/cli ->           (Symlink to the current version)
/opt/cfht/src/cli-3.31/...     (Complete source tree and libraries)
/opt/cfht/src/fh ->            (FITS Header library used by DetCom)
/opt/cfht/src/detcom ->        (Symlink to the current version)
/opt/cfht/src/detcom-3.31/...  (DetCom source code.)
```

Note: after building and installing, files will appear in:

```
/opt/cfht/obs/PROJECT/...      (Unstripped programs and object files)
/opt/cfht/lib/libNAME.a        (Archive libraries)
/opt/cfht/include/NAME/*.h     (Include files for libNAME.a)
/opt/cfht/bin/...              (Stripped program executables)
```

Assuming `PREFIX=/opt/cfht`, so choose `PREFIX` accordingly. `PREFIX=/usr/local` will cause things to go into /usr/local/bin/, so you typically won't have to adjust your path or make symlinks. It's a matter of taste really.

3. If necessary, edit `Make.Common` and make sure that the PATH in this file includes the proper version of GNU make before any other make's you might have on your system. (One alternative would be to install GNU make as "gmake" and run "gmake" instead of just "make".)

4. To verify the installation locations before doing anything, change to the `src/` directory and type "make". The "make" command will then give further instructions on how to proceed. DetCom may not build on some platforms. To avoid building DetCom, change to the `src/cli` directory and run "make world" there (which builds and installs everything from there down, skipping `src/fh` and `src/detcom`.

5. During the make process, you will be prompted for the root password once. This is used to make the "setuidinst" program setuid root. It, in turn, is run when you install "runon" (or any other program that it has been authorized to operate on) to make "runon" setuid root so you don't have to remember to do it manually. If you consider this a security risk, you'll have to manually make "runon" setuid root to be able to run Agents on remote hosts.

If the programs compile successfully, but there were warnings, read them carefully to be sure that they are not going to cause problems later on. Warnings about SIG_IGN or SIG_DFL not being proper prototypes and warnings about format arguments to printf's can be ignored. If the warnings are due to other missing prototypes on your system, you should check the man page for a prototype and add it to "missing_protos.h". This file does not appear in the dependencies for any targets, so you'll have to do a "make clean world" to rebuild everything after editing this file. (The alternative is to add "-Werror" to the gcc options to prevent anything from building unless it is completely warning-free.)

Just a tip when looking around in the directories: If you find a file called "**Index**" it will typically tell you what the files and subdirectories in the current directory are for.