The Inferno Environment

3

Introduction

This section provides an overview of working with Inferno in a hosted environment. Inferno can be set to run three distinct environments:

- The standard Inferno shell, which is always present when the emulator is running, provides a basic command interpreter similar to the various Unix shells or a DOS window. When Inferno is first invoked, the shell is presented as a console window.
- The *mux* application provides a demonstration environment that is meant to simulate services that might be provided using Inferno as the operating system for a set-top box.
- The *wm* application provides a demonstration environment similar to a traditional window manager.

In addition, the Limbo compiler, the Limbo debugger and the process of developing Limbo applications are described. For more information about the language, see *The Limbo Programming Language*. For information about the compiler, including command line options, see the *limbo*(1) man page.

Running the Inferno Emulator

The Inferno emulator, an application named *emu*, is invoked to establish the Inferno environment as a process under the host operating system. After start-up, this environment is identical to the Inferno native environment.

The procedures in this section assume that Inferno has been installed following the procedures described in *Installation and Setup*.

The Inferno File Name Space

The Inferno file system conforms to a set of conventions that promote the flexibility of name space manipulation. These conventions should be adhered to for the system to behave normally. After installation the Inferno file system under *<inferno_root>* is arranged as follows:

/	The Inferno root directory.
appl	Top level directory containing Limbo source code for various demonstration applications.
chan	Location for writing channel control files (empty).
dev	Location for writing device control files. Contains a file, NVRAM, used in set-top box simulations.
dis	Location of Dis programs. Subdirectories group specific application executables: for example, *dis/mux* contains all programs used by the *mux* application; *dis/lib* contains standard library executables such as *srv*. The *dis* directory is on the Inferno search path by default.
fonts	Location of font files.
httpcache	Location of temporary storage for browser files.
icons	Location of raster graphics files.
keydb	Location of signer-related files.
lib	Location of collections of data generally not part of programs.
locale	Location of time zone localization files.
man	Location of documentation PDF and HTML formats.
module	Location of module files for Limbo programs.
movies	Location of files used in mux application.
n	Location of network-related directories and files.
net	Location of network-related directories and files for all available network devices.
***<os_path>*/bin**	Platform-specific directory containing system executables. For example, under Windows, <os_path> is *<inferno_root>*\Nt\386.
prog	Location of thread-related control files.

services	Location of service related directories and files.
usr	Location of user home directories.

Starting the Inferno shell

The Inferno shell is the starting point for all operations involving Inferno in a hosted environment. The shell is always present when the emulator is running and provides a basic command interpreter similar. When Inferno is first invoked under Windows, the shell is presented as a separate console window. Under Unix, the shell is a child process in the terminal window in which the emulator was invoked.

To start the Inferno shell

Requirement: On a Unix server, the emulator must be run as root; on a Unix client, the emulator must be run as a process owned by user inferno. In the Windows NT environment, the Inferno file system must be open. That is, *Full Control* must be granted to *Everyone*.

1. The location of the emulator executable is referred to as *<emu_bin>.* Some default values are shown below. These can be added to your *$PATH* environment variable.

System	***Default <emu_bin>***
Solaris	*<inferno_root>/Solaris/sparc/bin*
Irix	*<inferno_root>/Irix/mips/bin*
Windows	*<inferno_root>\Nt\386\bin*

2. Under Unix, open a terminal window and enter the following command:

```
<emu_bin>/emu
```

3. Under Windows NT, double click on the emulator icon in the Inferno program group. Under Windows 95, select **Start>>Programs>>Lucent Inferno>>Emulator**.

Response: An Inferno console window is displayed running the Inferno command interpreter.

Comment: A variety of options are available for the *emu* command that allow you to customize the appearance of the emu window and control compilation options, paths and the like. These options are described in the *emu*(1) man page. If you start the shell using a Windows icon or desktop shortcut, you can add *emu* options to the icon properties.

NOTE:
In the Inferno console window, a path should use the forward slash as in Unix. Do not use the backslash as in the DOS/Windows environment.

Starting the Inferno window manager

Starting the Inferno window manager does not require a server and does not establish an authenticated connection. At the conclusion of this procedure, you can use the window manager in isolation. You cannot connect to remote servers and you cannot simulate authenticated connections. However, you can write and compile Limbo programs and use many of the demonstration programs that are provided with the Inferno distribution.

To log on the Inferno window manager

1. Start the Inferno shell.

2. In the Inferno console window, establish an initial environment by invoking the following commands:

```
bind '#I' /net
lib/cs
wm/logon
```

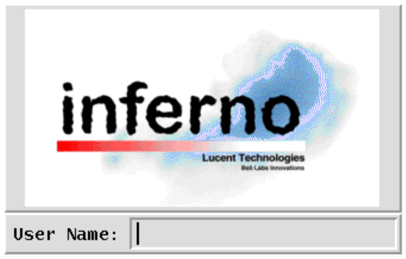
Requirement: The single quotes surrounding the `#I` device in the *bind* command are required. If they are omitted, the shell considers all text after the pound sign to be a comment.

NOTE:
Under Windows the Inferno window manager is initially iconified. To view the window manager, click on the Inferno icon in the Windows 95 task bar or on the Windows NT desktop.

Comment: The *bind* command sets up a local network environment in preparation for connection to a server.
The *cs* application provides connection services.
The *logon* application starts the window manager.

Response: The first time a user logs on, a license form is displayed. Accepting the license agreement prevents the form from being displayed again. The Logon dialog box is then displayed.

Figure 3-1. Logon Dialog Box



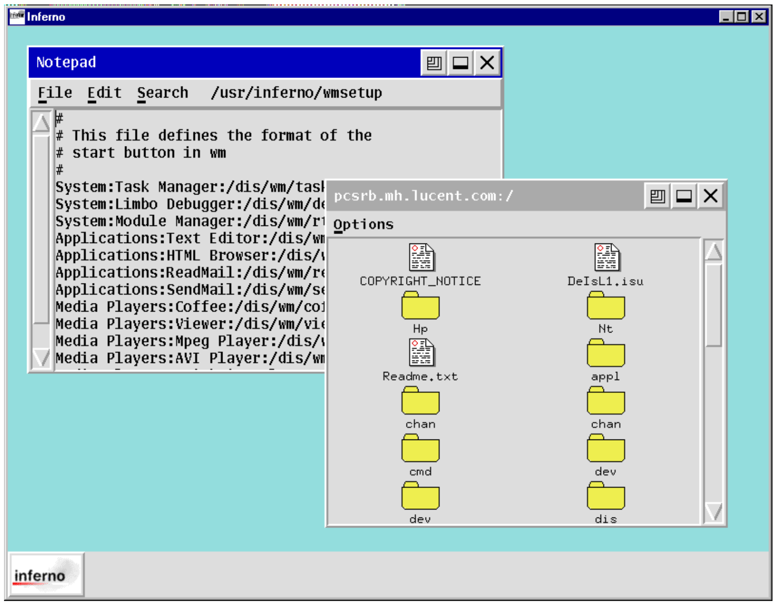
3. Enter a user name in the *User Name:* field.

Requirement: The user name entered must correspond to a valid user and a directory name in *<inferno_root>/usr* located on the server and the local machine.

Comment: The window manager reads configuration information from two files in the user directory. The content and format of these files, *namespace* and *wmsetup*, are described in *Setting Up a Custom User Environment*.

When logon is complete, the window manager appears as a clear screen with the Inferno icon displayed at the lower left of the main window.

Figure 3-2. The Inferno Window Manager in the Microsoft Windows Environment With Multiple Windows Displayed



```
#
# This file defines the format of the
# start button in wm
#
System:Task Manager:/dis/wm/tas
System:Limbo Debugger:/dis/wm/de
System:Module Manager:/dis/wm/r
Applications:Text Editor:/dis/wm
Applications:HTML Browser:/dis/
Applications:ReadMail:/dis/wm/r
Applications:SendMail:/dis/wm/s
Media Players:Coffee:/dis/wm/co
Media Players:Viewer:/dis/wm/vi
Media Players:Mpeg Player:/dis/
Media Players:AVI Player:/dis/wm
```

The Inferno window manager is now running. For more information about the *wm* user interface, see *The Inferno Window Manager*.

Connecting to the network

The procedures in this section assume that a server running the *srv* application is available to client machines attempting to use the network.

NOTE:
If you wish to run Inferno using authenticated connections on a single machine in loopback fashion, you can use the procedures in this section on one local machine running two instances of Inferno. It is necessary, however, that the single machine be initialized as a signer prior to attempting to establish authenticated connections.

Inferno is meant to be used as part of a network. The machine designated as signer, which acts as the connection authentication server, must be initialized before any authentications can be established. The initialization consists of ensuring that the signer is running the *srv* application.

To initialize the signer

NOTE:
Before any user can establish an authenticated connection, a user account exist on the signer machine. See the procedure *To create an Inferno account on a signer machine*.

1. Start the Inferno shell on the signer machine.

2. In the Inferno console window on the signer machine, run the *srv* application:

```
lib/srv
```

The signer is now ready to authenticate connections between clients and servers in the network.

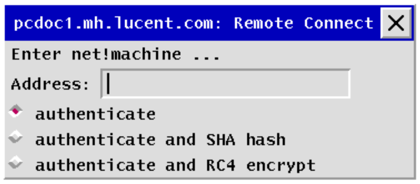
To establish an authenticated connection

1. Start the Inferno shell on the client machine.

2. Log on to the Inferno window manager.

3. In the Inferno window manager, follow the menu path **Inferno>>remote** to display the Remote Connect dialog box.

Figure 3-3. Remote Connect Dialog Box


pcdoc1.mh.lucent.com: Remote Connect

Enter net!machine ...

Address:

[✓ authenticate]

authenticate and SHA hash

authenticate and RC4 encrypt

4. Enter a machine name to connect to in the form *<net>*!*<machine>*.

NOTE:
In the current release, TCP is the only network protocol supported. That is, *<net>* is always `tcp` in this release. The *<machine>* value should be the node name of the machine to connect to, not its IP address.

5. Select the level of security that is desired for the connection and press the Enter key.

   Comment: The *authenticate* selection validates each connection in the network.
   The *authenticate and SHA hash* selection validates each connection and affixes a digital signature to each message.
   The *authenticate and RC4 encrypt* selection validates each connection and encrypts each message.

Exiting from Inferno

Response: In the Unix environment, the Inferno shell is invoked and the $ prompt is displayed. In the Windows NT environment, an Inferno console window is displayed. Under Windows 95, an inferno console window is displayed and the window manager is started as an icon on the Windows task bar.

To exit from Inferno

1. Press Ctrl+C in the Inferno console window.

   Response: The emulator is terminated.

NOTE:
Exiting from Inferno while the window manager or mux applications are running will kill those applications.

The Inferno Shell

The Inferno shell provides a basic set of commands that can be used directly in the Inferno console window. Most commands are implemented as independent Limbo programs; the source code for these programs resides in the *<inferno_root>/appl/cmd* directory. The corresponding executable *.dis* files are located in *<inferno_root>/dis*. The behavior of these commands can be modified by changing the Limbo source and recompiling the programs.

The Limbo shell is a “blank slate” environment in which you can experiment with the construction of name spaces using *mount* and *bind* commands. For example, issuing the typical *bind* command

```
bind '#I' /net
```

prepends the local IP device to the devices directory. This ensures that the local protocol stack and network interfaces are used to establish new connections instead of those of the server.

NOTE:
In the Inferno console window, a path should use the forward slash as in Unix. Do not use the backslash as in the DOS/Windows environment.

The Inferno shell can also be accessed from within the window manager. See *The Inferno Window Manager*.

Shell Commands Quick Reference

The following table lists the basic commands that are supported by the Inferno shell. These commands generally mimic the well-known Unix variants. More complete information is presented in the man pages, included in *The Manual Pages*.

bind [-abcr] <old> <new>	Modifies the local name space, creating another name *<new>* for an existing file *<old>* . For directories, the options are: `-a` to place *<new>* after *<old>* `-b` to place *<new>* before *<old>* `-c` to create a union of *<old>* and *<new>* `-r` to replace *<old>* with *<new>*
cat [fn ...]	Output the contents of a file or files to the standard output: `cat fn` lists *fn* to *stdout* `cat fn > fn2` lists *fn* to *fn2* `cat fn fn2 > fn3` concatenates *fn* and *fn2* into *fn3*
cd <directory>	Changes the current working directory to *<directory>*
chmod [augo][+-=][rwx] **chmod 777 file**	Changes Inferno file mode (access permissions). Read *<r>*, write *<w>*, or execution *<x>* permissions can be added *<+>*, removed *<->*, or assigned *<=>* for the owner *<u>*, the owner's group *<g>*, all others *<o>*, or all *<a>*. Octal values are also allowed.

cmp [-lsL] <f1> <f2> [off1] [off2]	Compare files *<f1>* and *<f2>* starting at offsets *off1* and *off2*, respectively (default 0). No output if files are identical. The options: `-s` silent mode, no output messages `-L` suppress line number message `-l` further suppress offset message
cp <old> <new>	Copies file <old> to file <new>
date	Prints the date
du [-anst] [file ...]	Print disk usage in specified files and directories. The options are: `-a` show count per file `-n` use netlib format `-s` print sum per directory `-t` use terse report format
echo <text>	Prints *<text>* to *stdout*
grep [-lnv] <pattern> [file...]	Output lines of file(s) that match *<pattern>*. The options are: `-l` to just print file names, not lines. `-n` to include line numbers `-v` to output lines that don't match *<pattern>*
kill [-g] <pid|module>	Aborts the process identified as *<pid>* or all processes running *<module>.* The option, `-g` terminates entire *process group*.
ls [–lpqdtusr] [file|directory]	Lists the files in a directory. Options are: `-l` verbose output `-p` list only final element of path names `-q` list qid with file `-d` list only directories `-t` sort by time last modified `-u` sort by last access `-s` sort by size `-r` sort in reverse order
mathcalc	Makes the calculator facilities of TKlib available from the command line.
mkdir <directory ...>	Creates a directory

mount [-abcrA] <old> <new>	Modifies the name space by attaching resource from *<old>* to directory *<new>*. The options are: `-a` to place *<new>* after *<old>* `-b` to place *<new>* before *<old>* `-c` to create a union of *<old>* and *<new>* `-r` to replace *<old>* with *<new>* `-A` to authenticate the connection
netstat	Prints the status of TCP and UDP network connections
nsbuild <nsfile>	Builds name space per <nsfile> contents.
os <command [args...]>	Executes <command> in the host operating system
ps	Lists information about currently running processes
pwd	Prints the current working directory
rm [file|directory...]	Remove named file(s) or (empty) directories.
sh [file]	Invokes the shell as a child of the current shell. Commands taken from file, if specified; else, from *standard input*.
sleep [duration]	Suspends execution for the number of seconds specified in *<duration>*
stack [-v] <pid>	Prints the contents of the stack for the named *<pid>* to *stdout*. The `-v` option prints a verbose version.
unmount <old> **unmount <new> <old>**	Undoes the effect of a mount or bind. If *<new>* is unspecified, everything bound to *<old>* is unmounted.
wc [-lwrbc] [file ...]	Counts items in the specified UTF-text files, if any, else from *stdin*. The options are; `-l` count lines `-w` count words `-r` count runes `-b` count bad runes encountered `-c` count characters (bytes)
wish [file ...]	Invokes a Tcl-like command shell. Directives in specified files, if any, are executed, then commands taken from *standard input*.

The Mux Demonstration Application

The Inferno operating system can be used to provide an environment for a variety of hardware. The mux application simulates the implementation of a range of services on a set-top box controlled by an infrared device. In the demo application, the keyboard is used instead of an IR device.

NOTE:
Individual application programs in the *<inferno_root>/mux* directory should not be executed except from within the mux demonstration application.

To start the mux application

1. Start the Inferno emulator.

NOTE:
On a Unix server, the emulator must be run as root; on a Unix client, the emulator must be run as a process owned by user inferno. In the Windows environment, admin access is necessary.

NOTE:
In the Unix environment, the location of the emulator executable is referred to as *<emu_bin>*. Under Solaris, *<emu_bin>* is by default *<inferno_root>/Solaris/sparc/bin*; under Irix, *<emu_bin>* is by default *<inferno_root>/Irix/mips/bin*. The complete path may be added to your *PATH* environment variable.

2. From an Inferno console window, change directory to *<inferno_root>* and issue the following the command:

```
mux/mux
```

Response: The mux application window is displayed

NOTE:
The window is initially iconified. To view the mux window, click on the Inferno icon in the Windows 95 task bar or on the Windows NT desktop.

Navigating the mux application

The keys that the mux application uses to simulate the IR device are:

m or down arrow	Move down the menu
i or up arrow	Move up the menu
Enter	Select a menu item
x	Bring the menu to the front
Spacebar	Kill the current menu selection

The mux application demonstrates the following services:

- Financial Reports: adds a stock quote ticker to the bottom of the mux window
- Movies: allows the selection of a specific movie from a list and provides information about the selection
- Today's Newspaper: displays a list of simulated newspapers in multiple languages and allows reading of specific articles
- Grit Bath Comics: displays some comic strips
- TV Information: provides a schedule of television programs
- Order Pizza: allows selection of food and processes orders
- Internet Mail: simulates an email facility
- Internet Web Browser: simulates a menu-driven web browser
- Register: simulates a set-top registration process
- Audio Control: allows audio control of the set-top box

The Inferno Window Manager

Inferno provides a basic window manager environment that demonstrates many of the graphic capabilities of Inferno applications developed using the Limbo programming language. Included by default as menu options are:

- About: a standard about box
- Media Players: various programs to display multimedia files, including Coffee (a simple animation), Viewer (a bitmap viewer), and players for MPEG, AVI and QuickTime files
- Applications: including a fully functional text editor, an HTML browser, and applications to send and read email

- System: several applications designed to help in Limbo program development, including a system task manager, the Limbo debugger and a Limbo module manager
- Local: a file manager application for the local machine
- Remote: a file manager application for remote connections
- Tasks: a system task manager
- Notepad: a fully functional text editor
- Shell: a command line interpreter window

Setting Up a Custom User Environment

When the Inferno emulator window manager is invoked, the *namespace* and *wmsetup* files are read to provide a configuration for the machine similar to a *.profile* configuration file in the Unix K shell or an *autoexec.bat* file in Windows. These are simple ASCII text files that can be freely edited to provide a customized environment for each user.

The Name Space File

The *namespace* file contains a set of mount and bind commands that provide a specific custom view of network resources. Edit this file to provide transparent access to any available resources in the network. Minimally, this file must contain the following line:

```
bind -ia #C /
```

NOTE:
Single quotes surrounding the device identifier in the bind command are not required because the namespace file is not run by the shell.

The wmsetup File

The *wmsetup* file is read during logon in order to configure the window manager main menu. The default main menu can be augmented to list any Inferno application by editing the *wmsetup* file in a user's home directory. Menu items can be grouped as sets of related items on a submenu. The default main menu can be modified by changing the wm.b file and recompiling it.

wmsetup File Format

The format for the wmsetup file is:

```
<main_menu>:<menu_text>:<command>
```

The *<main_menu>* column specifies the word(s) that appears on the main menu. The *<menu_text>* column specifies the word(s) that appears on the cascaded submenu. The *<command>* specified the application that is to be invoked. Spaces are allowed between colons, for example:

```
Applications:Text Editor:/appl/wm/wmedit.dis
```

The Limbo Development Process

The development of Limbo applications proceeds in a fashion similar to other development environments. A brief Limbo tutorial written by B. Kernighan that is designed to supplement this section is available for downloading from the Inferno web site at:

http://www.lucent.com/inferno/

Source Code Compilation

The Limbo compiler is a machine dependent program that is run under the native operating system. The compiler executable is located in *<inferno_bin>*: under windows it is named *limbo.exe*, under Unix it is named *limbo*. The compiler translates Limbo source code files into binary files, conventionally given the extension *.dis*.

The object files that are output from the Limbo compiler are machine independent programs that run under Dis. Depending on the options selected, invoking the compiler with the command

```
limbo [option ...] [file ...]
```

produces either output files or information to the standard output device. The compiler options control the form and type of output. Conventional files and their extensions include the following:

*filename***.b**	Limbo source code file
*filename***.sbl**	Debugging information
*filename***.m**	Limbo source file for module declarations and *include* statements
*filename***.dis**	Byte code file executed by the interpreter
*filename***.s**	Assembly code

These conventions are not enforced by the compiler. The compiler options and their definitions are described in *The Manual Pages*.

The Limbo Debugger

Before you can begin debugging there are several concerns to be addressed.

- To operate on an existing program the debugger needs information that is not provided by the usual compilation of source code. You must compile the program with a special option.
- You must be able to locate and start the debugger.
- You must be able to specify the program of interest to the debugger. That might be either a currently running program or one that will be explicitly started by the debugger.

Compilation for Debugging

To debug a program under development, invoke the Limbo compiler with the command line option `-g`. This option directs the compiler to generate additional information needed by the debugger.

The following files are needed by the debugger to act on a program.

filename.**b**	Limbo source code
filename.**dis**	Compiled Dis code
filename.**sbl**	Additional (line number) information.

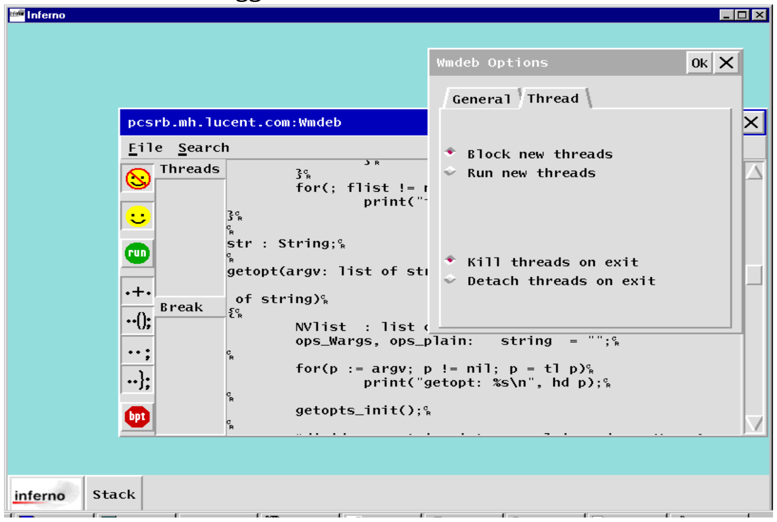
For example, in a Windows cross development environment, the steps might be:

```
C:>dir getopt.* /w
getopt.b
C:>limbo -g getopt.b
C:>dir getopt.* /w
getopt.b     getopt.dis   getopt.sbl
C:>
```

Starting the Debugger

The debugger is a graphical application that is run from the Inferno window manager.

Figure 4. The Limbo Debugger Windows



Wmdeb Options — Thread:
- [✓ Block new threads] Run new threads
- [✓ Kill threads on exit] Detach threads on exit

```
        }
        for(; flist != n
                print("
}
str : String;
getopt(argv: list of str
 of string)
{
        NVlist  : list o
        ops_Wargs, ops_plain:   string  = "";

        for(p := argv; p != nil; p = tl p)
                print("getopt: %s\n", hd p);

        getopts_init();
```

To start the Limbo debugger

1. From the Inferno icon, follow the menu path **Inferno>>System>>Limbo Debugger**.

2. When the debugger starts two windows are created, one for program control, and the other for the display of stack information (Figure 4).

The Debugger Control Window

You can control the debugger by clicking on the various icons found on the display. At any time, you can determine which actions are valid by passing the cursor over them. The valid buttons are highlighted. Initially, only the **File** button (to choose a program for debugging) is valid.

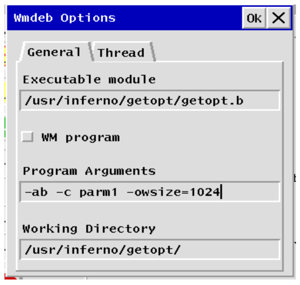
The File Menu Item

The **File** menu item is used to select the program for debugging. That program can be:

- A currently running thread selected via **File>>Threads**. A thread can also be selected via **Inferno>>Shell>>Task Manager**.

- A newly created thread running the program selected via **File>>Open**.

Figure 5. Sample File>>Options Usage



Executable module
/usr/inferno/getopt/getopt.b

☐ WM program

Program Arguments
-ab -c parm1 -owsize=1024

Working Directory
/usr/inferno/getopt/

The File Menu Options

The following options are available on the **File** menu:

Open... A file-tree browser to locate program source for debugging.

Threads... A list of currently running threads with columns for process (thread) id, process group id, process state, and the name of the currently running module

	Button: **Add Thread**	Start debugger for the selected thread.
	Button: **Add Group**	Start debugger for the selected thread (process) group.
Options...	**General Options**	Input Field: **`executable module`**
		Radio Button: **`wm program`**
		input Field: **`Program arguments`**
		Input Field: **`Working Directory`**
	Thread Options	Radio Buttons: **`Block new threads (default)`** **`Run new threads`**
		Radio Buttons **`Kill threads on exit (default)`** **`Detach threads on exit`**

The Select Menu Item

The **Select** option provides navigation through the source file opened via the **File** option. The Select option provides two actions

Look	Find the next occurrence in the file of the *current search string.*
Search for	Prompt for a search string (via a dialogue box) and find the next occurrence. That string is then displayed in the **Select** menu as the *current search string* for **Look** operations.

Debugger Control Icons

Cancel Icon: Terminate thread

Cancel Icon: Terminate thread.

Smile Icon: Detach thread.

Green Light Icon: Run thread.

Step by expression.

Step by statement.

Step past function calls (do not step into function).

Step to end of function. This is especially useful if you unintentionally step the debugger into a function.

Stop Sign Icon: Set/Unset break point

Running the Debugger

After choosing a program to debug via the **File** menu, clicking on the *Green Light* icon makes the debugger run the previously selected program. By default, the program breaks at the first executable statement.

NOTE:
This default behavior can be suppressed by engaging the radio button **Run new threads** under **File>>Options...>>Thread**.

You can advance the execution by one expression at a time, by one statement at a time, or by other options pictured in *Debugger Control Icons*. Alternatively, you can set break points in the code and run the program until a breakpoint is encountered in the execution. In each case, the code that has caused the debugger to stop program execution appear highlighted in the code display panel.

Stepping through Function Evaluations

When stepping through a program one expression at a time, there are several interesting stages when a function call is encountered.

- First, the entire function statement is highlighted. This represents the allocation of the stack fame for the function.
- Next, the debugger stops and highlights each of the function arguments as each is evaluated.
- Finally, the debugger highlights the entire function statement again to represent the actual calling of the function with the computed arguments.

BreakPoints

To set a breakpoint

1. Click on the *Stop Sign* icon.
2. Select the code that should cause the break.
3. Click on the *Stop Sign* icon.

   Response: The code of interest will be displayed in red thereafter. Also, the number of the newly created breakpoint is entered in the panel labelled **Breakpoints**.

To unset a breakpoint

1. Select the code of interest.

   Response: The code section is highlighted.

2. Click on the *Stop Sign*.

Response: The code resumes its normal display color and the associated breakpoint number is removed from the **Breakpoints** panel.

Terminating a Program

There is an implicit breakpoint at the end of every program (as there was at the first executable statement). This allows you an opportunity to examine the final state of the computation.

To proceed further, the program must be terminated by pressing the *Cancel* Icon. Afterwards, you can (if appropriate) restart the program by clicking on the *Green Light* icon.

By default, all other threads created by the debugged program will be terminated upon exit. This default behavior can be suppressed by setting the switch for **Detach thread on exit** under **File>>Options>>Thread Options**. If set, those threads continue execution.

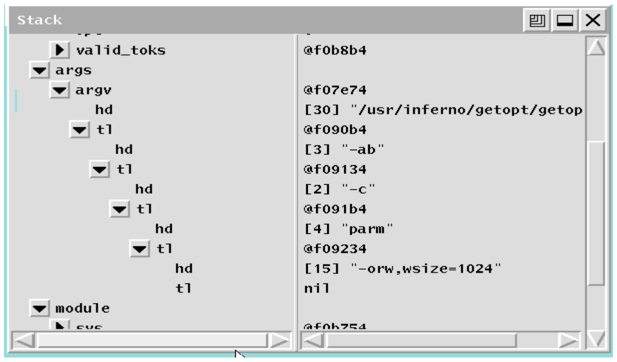
The Debugger Stack Window

The Stack window is divided into two panels. The left panel is used to select which information to display and the right displays the contents.

▶ Initially, the various parts of the name space (locals, modules, etc.) are represented by icons in the “closed” state.

▼ When clicked the icons will be expanded into further details (e.g., various local variable names) and their current values will be displayed on the corresponding line in the right panel.

Figure 6. Sample View of the Stack Window



```
  ▶ valid_toks              @f0b8b4
▼ args
  ▼ argv                    @f07e74
      hd                    [30] "/usr/inferno/getopt/getop
    ▼ tl                    @f090b4
        hd                  [3] "-ab"
      ▼ tl                  @f09134
          hd                [2] "-c"
        ▼ tl                @f091b4
            hd              [4] "parm"
          ▼ tl              @f09234
              hd            [15] "-orw,wsize=1024"
              tl            nil
▼ module
```