Ski IA-64 Simulator Reference Manual

Rev. 1.0L (9 Oct 07)

Copyright © 2000 Hewlett-Packard Co.

Notice

The information in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with furnishing, performance, or use of this material.

This document contains information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced, or translated to another language without the prior written consent of Hewlett-Packard.

Copyright © 2000-2007 by Hewlett-Packard Development Company, L.P.

Trademarks

Linux is a registered trademark of Linus Torvalds.

MS-DOS and *Windows* are registered trademarks of Microsoft Corporation. *UNIX* is a trademark or registered trademark of the Santa Cruz Operation.

Intel is a registered trademark of the Intel Corporation.

Preface

This document is the Ski IA-64 Simulator Reference Manual. The goal of this document is to provide a description of the features, commands, and simulation environment provided by the Ski IA-64 simulator. The version of the simulator described here is Version 0.873l.

How to Use This Manual

The first chapter of this manual is a quick-start tutorial. Using only the first chapter, you can learn enough about Ski to do useful work. If you are using Ski to simulate an IA-64 application program and are familiar with debuggers such as HP's xdb, the first chapter and Appendix A, Command Reference may be all you need to read.

The remaining chapters provide information about Ski in depth. Use these chapters to learn about commands not covered in the tutorial and to learn more about how Ski operates.

Use "Command Reference" and the on-line **`help`** command to find a list of all Ski commands and a brief description of each command.

Use "Simulator Status and Error Messages" to understand the causes and possible solutions for each of Ski's error messages.

Font Conventions

In this manual, fonts are used as described below. Depending on how you are viewing this document (paper, a web page, a PDF file, etc.), some distinctions may not be visible.

italic

is used for optional text including operand fields such as *count*, and for the names of bitfields such as *psr.be*.

`light italic`

is used for graphical button names such as *Run*.

`fixed-width bold`

is used for literal text including commands such as **`dbndl`**, and for examples such as **`bski -icnt foo <bar >baz`**.

SMALL UPPERCASE

is used for processor instructions such as BREAK.

`fixed-width regular`

is used for directories and filenames such as `hello`, and for web URL's such as `http://www.hp.com`.

Syntax Conventions

In this manual, symbols are used as described below.

[*italic***]**

Square brackets surrounding optional argument(s) indicate that the argument(s) can be omitted, as in the "Command Reference" description of the **`dj`** command: **`dj`** [*address*].

italic+

A plus sign applied to an argument indicates that the argument must be supplied one or more times, as in the "Command Reference" description of the **`eval`** command: **`eval`** *expression_without_spaces*+.

[*italic***]+**

A plus sign applied to optional argument(s) in square brackets indicates that the argument(s) can be supplied zero or more times, as in the "Command Reference" description of the **`load`** command: **`load`** *filename* [*args*]+.

Table of Contents

Illustration Index

Index of Tables

1 Getting Started - A Ski Tutorial

In this chapter, you learn how to use Ski by executing a brief tutorial. At the end of the tutorial, you will learn where to look in this manual for detailed descriptions of Ski's operation and commands. Introductory information on Ski is presented in the "Overview" on page 30.

The Ski Simulator

Ski simulates the IA-64 architecture and also has limited support for simulating IA-32 programs. Ski runs on IA-32 Linux host systems. You can use Ski for many purposes, as described in the "Introduction" on page 30. One of the most common uses of Ski is to test an IA-64 program in a Linux environment, and in this chapter, you will learn how to use ***xski***, the X Window System version of Ski, by "walking through" a sample session, in about ten minutes. Ok, twenty minutes.

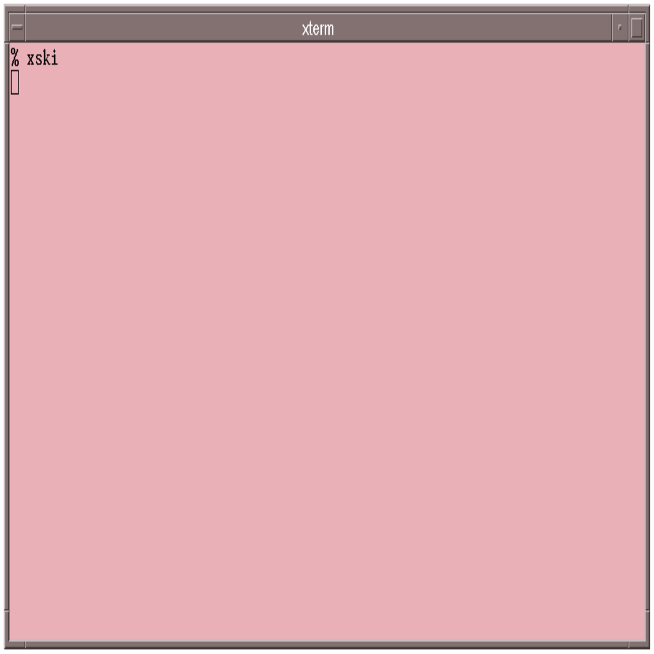
You should already be familiar with the IA-64 architecture and the C programming language, have ***xski*** installed on your Linux system, and have the `XSki` file in your home directory or in your X Window System app-defaults directory, typically `/usr/lib/X11/app-defaults`. You will also need to have an executable Linux IA-64 program such as the classic "`hello world`" program.

How to Run an IA-64 Application Program

Ski provides a Linux application environment in which an IA-64 program you provide can be simulated. The release notes provide the most up-to-date information on Ski's support for the Linux Application Binary Interface (ABI). The following sections provide a short tutorial which leads you through an IA-64 program session with ***xski***. You will learn how to use the most common Ski commands.

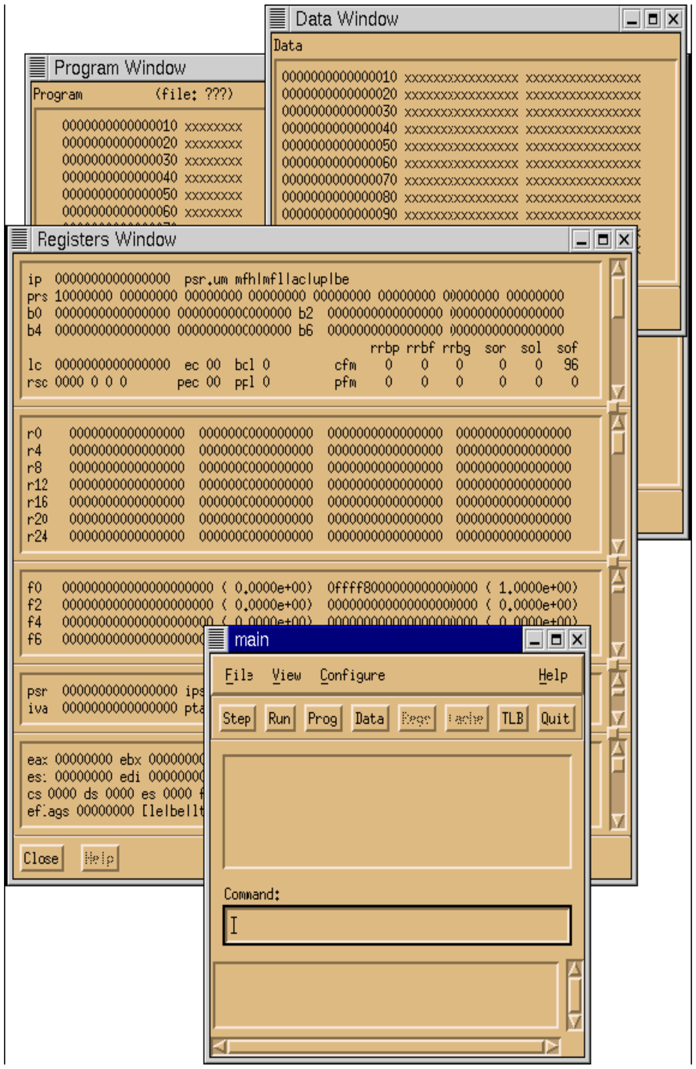
Starting *xski*

As shown in Illustration 1: Starting xski From the Command Line1, start ***xski*** by typing its name to the Linux shell, just like any other Linux program. When running inside the IA-64 Linux Native User Environment (NUE), make sure that the environment variable DISPLAY is set to a string of the form *`hostname`*`:`*`display`* (e.g., **`myhost:0`**", values such as **`unix:0`**" or **`:0`**" won't work) before invoking ***xski***. If you have never run the simulator before, it will first prompt you to read and accept the software license it is distributed under. After accepting the license, the four primary ***xski*** windows will be displayed on your screen, as shown in Illustration 2: The Four Primary xski Windows. No IA-64 program is loaded yet, so the Program Window and Data Window are empty. Scroll the various panes of the Register Window and note that with few exceptions, the registers are set to zero.



```
% xski
```

Illustration 1: Starting xski From the Command Line



```
Program          (file: ???)

    0000000000000010 xxxxxxxx
    0000000000000020 xxxxxxxx
    0000000000000030 xxxxxxxx
    0000000000000040 xxxxxxxx
    0000000000000050 xxxxxxxx
    0000000000000060 xxxxxxxx
```

```
Data

0000000000000010 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000020 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000030 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000040 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000050 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000060 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000070 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000080 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000090 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
```

```
ip  0000000000000000  psr.um mfh|mfl|ac|up|be
prs 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
b0  0000000000000000 000000000C000000 b2  0000000000000000 0000000000000000
b4  0000000000000000 000000000C000000 b6  0000000000000000 0000000000000000
                                             rrbp rrbf rrbg  sor  sol  sof
lc  0000000000000000  ec 00  bcl 0        cfm   0    0    0    0    0   96
rsc 0000 0 0 0       pec 00  ppl 0        pfm   0    0    0    0    0    0

r0    0000000000000000  000000C000000000  0000000000000000  0000000000000000
r4    0000000000000000  000000C000000000  0000000000000000  0000000000000000
r8    0000000000000000  000000C000000000  0000000000000000  0000000000000000
r12   0000000000000000  000000C000000000  0000000000000000  0000000000000000
r16   0000000000000000  000000C000000000  0000000000000000  0000000000000000
r20   0000000000000000  000000C000000000  0000000000000000  0000000000000000
r24   0000000000000000  000000C000000000  0000000000000000  0000000000000000

f0   00000000000000000000 ( 0.0000e+00)  0ffff8000000000000000 ( 1.0000e+00)
f2   00000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f4   00000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f6   00000000000000000000

psr  0000000000000000 ips
iva  0000000000000000 pta

eax 00000000 ebx 0000000
esi 00000000 edi 0000000
cs 0000 ds 0000 es 0000 f
eflags 00000000 [le|be|lt
```

Illustration 2: The Four Primary xski Windows

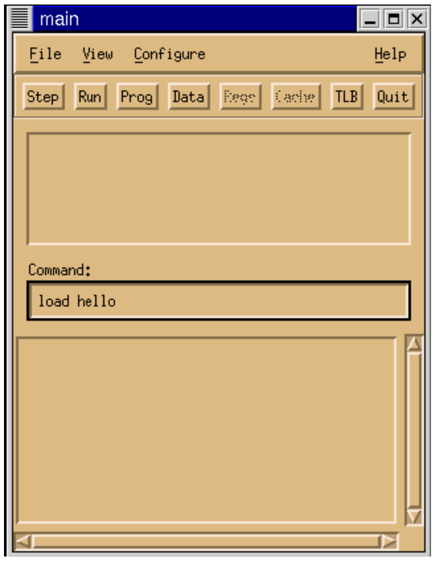
Exiting Ski

You can quit ***xski*** and this tutorial with the *`Quit`* button, with the File->Quit menu selection, or with the "**`quit`**" command. All are in the Main Window. (Don't quit now; you are just beginning!)

Loading Your Program

Use the "Command" area of the "main" Window to load your program. For example, let's say your program is the famous "Hello, world" program, the executable file is named "`hello`", and the source code file is named "`hello.c`". Type "**`load hello`**" in the Command area to load it into Ski, as you see in Illustration 3: Loading the "hello" Program. After a moment, the other three windows will change appropriately: the Program Window will show the program code in assembly language form as shown in Illustration 4: The xski Program Window, the Data Window will show global and static data as shown in

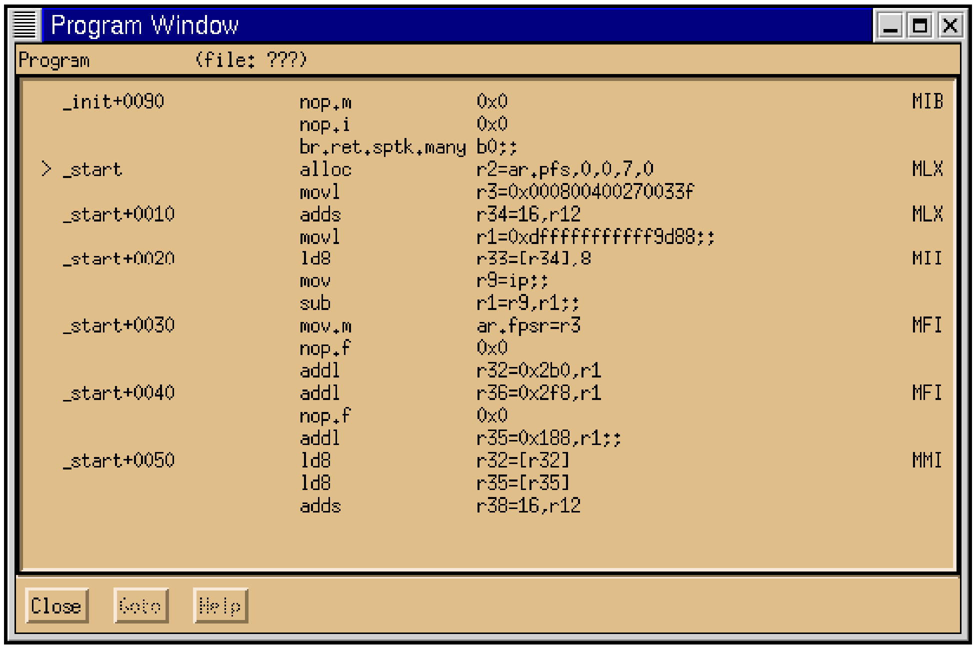
Illustration 5: The xski Data Window, and the Register Window will show, in **`r12`** the value of the stack pointer, as shown in Illustration 6: The xski Register Window. (You may need to use the scrollbar in the general registers pane of the Register Window to see these registers.)



Command:

```
load hello
```

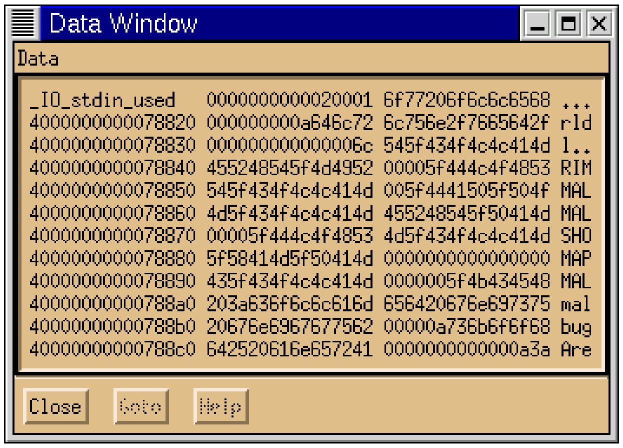
Illustration 3: Loading the "hello" Program



```
Program              (file: ???)

    _init+0090             nop.m            0x0                                MIB
                           nop.i            0x0
                           br.ret.sptk.many b0;;
  > _start                 alloc            r2=ar.pfs,0,0,7,0                  MLX
                           movl             r3=0x000800400270033f
    _start+0010            adds             r34=16,r12                         MLX
                           movl             r1=0xdfffffffffff9d88;;
    _start+0020            ld8              r33=[r34],8                        MII
                           mov              r9=ip;;
                           sub              r1=r9,r1;;
    _start+0030            mov.m            ar.fpsr=r3                         MFI
                           nop.f            0x0
                           addl             r32=0x2b0,r1
    _start+0040            addl             r36=0x2f8,r1                       MFI
                           nop.f            0x0
                           addl             r35=0x188,r1;;
    _start+0050            ld8              r32=[r32]                          MMI
                           ld8              r35=[r35]
                           adds             r38=16,r12
```

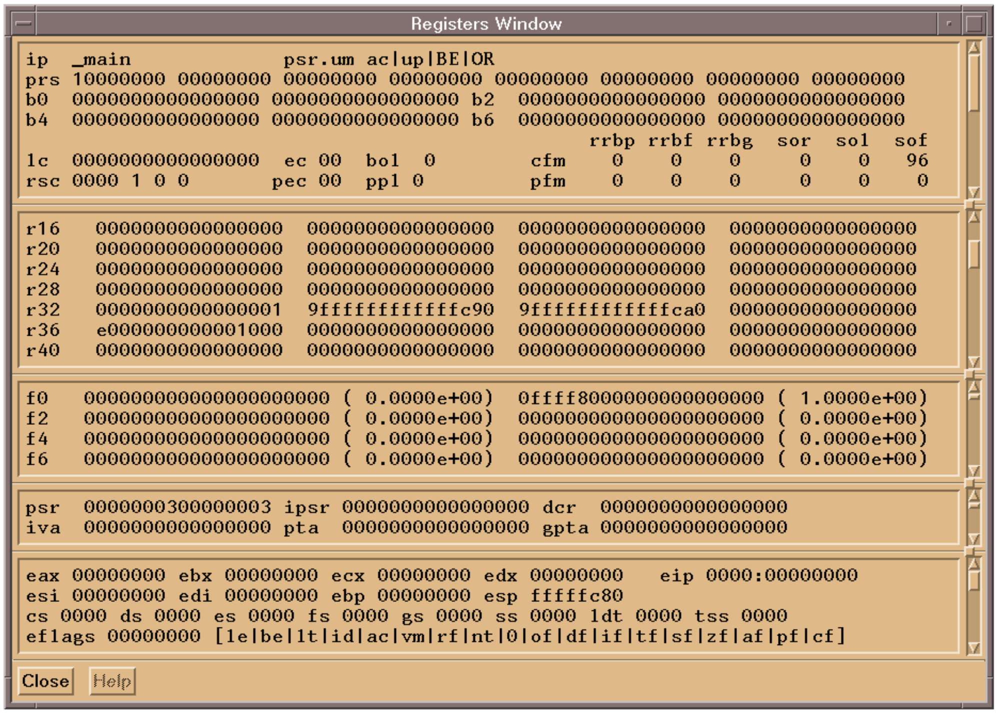
Illustration 4: The xski Program Window

"



```
Data

_IO_stdin_used   0000000000020001 6f77206f6c6c6568 ...
4000000000078820 000000000a646c72 6c756e2f7665642f rld
4000000000078830 000000000000006c 545f434f4c4c414d l..
4000000000078840 455248545f4d4952 00005f444c4f4853 RIM
4000000000078850 545f434f4c4c414d 005f4441505f504f MAL
4000000000078860 4d5f434f4c4c414d 455248545f50414d MAL
4000000000078870 00005f444c4f4853 4d5f434f4c4c414d SHO
4000000000078880 5f58414d5f50414d 0000000000000000 MAP
4000000000078890 435f434f4c4c414d 0000005f4b434548 MAL
40000000000788a0 203a636f6c6c616d 656420676e697375 mal
40000000000788b0 20676e6967677562 00000a736b6f6f68 bug
40000000000788c0 642520616e657241 0000000000000a3a Are
```

Illustration 5: The xski Data Window

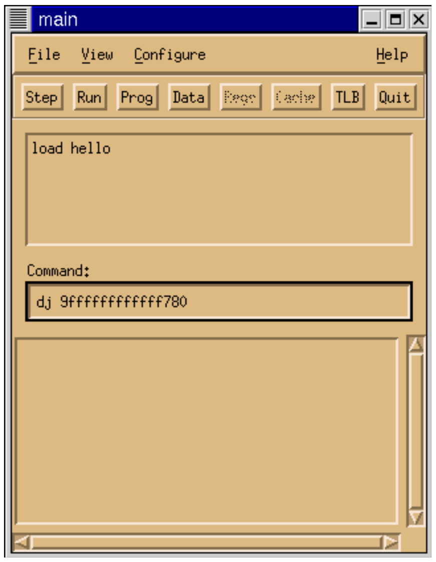


```
ip   _main                psr.um ac|up|BE|OR
prs 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
b0  0000000000000000 0000000000000000 b2  0000000000000000 0000000000000000
b4  0000000000000000 0000000000000000 b6  0000000000000000 0000000000000000
                                                rrbp rrbf rrbg  sor  sol  sof
lc  0000000000000000  ec 00  bol  0        cfm    0    0    0    0    0   96
rsc 0000 1 0 0       pec 00  ppl 0         pfm    0    0    0    0    0    0

r16   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r20   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r24   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r28   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r32   0000000000000001  9ffffffffffffc90  9ffffffffffffca0  0000000000000000
r36   e000000000001000  0000000000000000  0000000000000000  0000000000000000
r40   0000000000000000  0000000000000000  0000000000000000  0000000000000000

f0   000000000000000000000 ( 0.0000e+00)  0ffff8000000000000000 ( 1.0000e+00)
f2   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f4   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f6   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)

psr  0000000300000003 ipsr 0000000000000000 dcr  0000000000000000
iva  0000000000000000 pta  0000000000000000 gpta 0000000000000000

eax 00000000 ebx 00000000 ecx 00000000 edx 00000000   eip 0000:00000000
esi 00000000 edi 00000000 ebp 00000000 esp fffffc80
cs 0000 ds 0000 es 0000 fs 0000 gs 0000 ss 0000 ldt 0000 tss 0000
eflags 00000000 [le|be|lt|id|ac|vm|rf|nt|0|of|df|if|tf|sf|zf|af|pf|cf]
```

Illustration 6: The xski Register Window

Inspecting Data

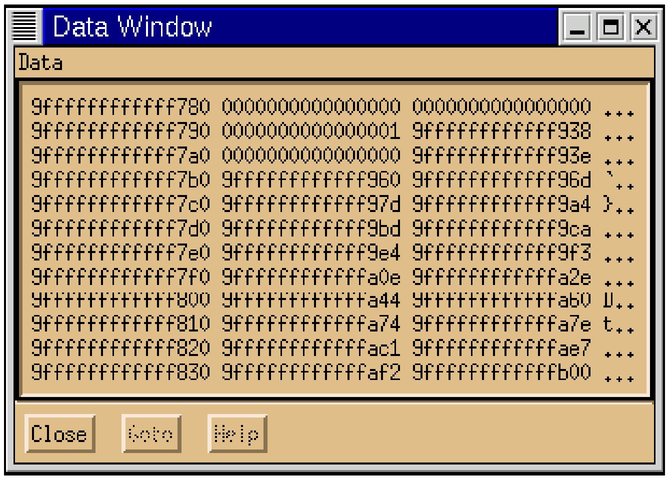
To look at the **`argv`** and **`envp`** strings, you need to use the Data Window. Linux passes **`argc`**, **`argv`**, and **`envp`** on the memory stack (**`r12`**). To look at this memory area, use the "**`dj`**" command ("**`d`**ata **`j`**ump") in "Command" area of the Main Window. Supply, as an operand, the address of the memory stack. For example, if **`r12`** is set to **`9ffffffffff780`**, you can type "**`dj r12`**" or "**`dj 9ffffffffff780`**", as shown in Illustration 7: Changing the Data Window Display and the Data Window changes to display the hexadecimal data stored at the location, as shown in Illustration 8: The Data Window Showing the argv and envp Vectors. Find the value of **`r12`** in your program and use "**`dj`**" now. (You might wonder why "**`dj`**" exists, instead of a simple scroll bar. Imagine scrolling through the entire IA-64 address space it would take a long, long time!)



```
load hello
```

```
dj 9ffffffffffff780
```

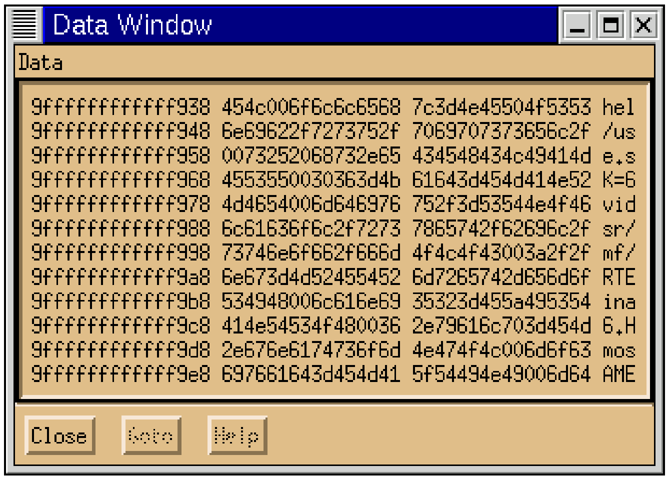
Illustration 7: Changing the Data Window Display



```
9ffffffffffff780 0000000000000000 0000000000000000 ...
9ffffffffffff790 0000000000000001 9ffffffffffff938 ...
9ffffffffffff7a0 0000000000000000 9ffffffffffff93e ...
9ffffffffffff7b0 9ffffffffffff960 9ffffffffffff96d `..
9ffffffffffff7c0 9ffffffffffff97d 9ffffffffffff9a4 }..
9ffffffffffff7d0 9ffffffffffff9bd 9ffffffffffff9ca ...
9ffffffffffff7e0 9ffffffffffff9e4 9ffffffffffff9f3 ...
9ffffffffffff7f0 9ffffffffffffa0e 9ffffffffffffa2e ...
9ffffffffffff800 9ffffffffffffa44 9ffffffffffffab0 D..
9ffffffffffff810 9ffffffffffffa74 9ffffffffffffa7e t..
9ffffffffffff820 9ffffffffffffac1 9ffffffffffffae7 ...
9ffffffffffff830 9ffffffffffffaf2 9ffffffffffffb00 ...
```

Illustration 8: The Data Window Showing the argv and envp Vectors

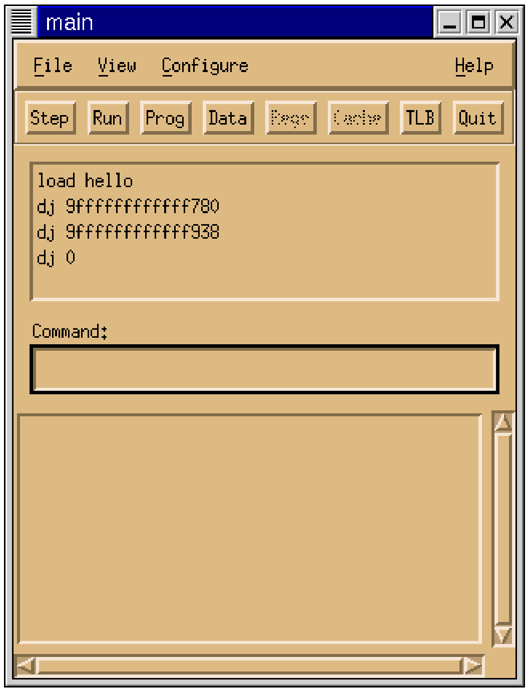
Looking at the Data Window, you can see that the first 16 bytes of the stack are all zeros. This is a scratch storage area. The next 8-byte word contains **`argc`**, the argument count. It has a value of 1 as the only argument passed to the program is the program name itself. The **`argc`** count is then followed by the **`argv`** and **`envp`** vectors. All C programs receive the same kind of data structure for **`argv`**: a variable-length vector of **`char *`** pointers whose end is marked with a NULL pointer. In Illustration 8: The Data Window Showing the argv and envp Vectors, the first of the **`char *`** pointers is **`9ffffffffffff938`**. (The first **`char *`** pointer may be in a different place on your system. Adjust the following instructions accordingly.) Jump the Data Window there using the command "**`dj 9ffffffffffff938`**" (12 f's) and you will see Illustration 9: The Data Window Showing argv and envp Strings in Hexadecimal, showing the hexadecimal codes for the null-terminated ASCII character strings of **`argv`** and **`envp`**. (In a moment, you'll learn how to see data in ASCII translation.)



```
9ffffffffffff938 454c006f6c6c6568 7c3d4e45504f5353 hel
9ffffffffffff948 6e69622f7273752f 706970737365 6c2f /us
9ffffffffffff958 0073252068732e65 434548434c49414d e.s
9ffffffffffff968 4553550030363d4b 61643d454d414e52 K=6
9ffffffffffff978 4d4654006d646976 752f3d53544e4f46 vid
9ffffffffffff988 6c61636f6c2f7273 7865742f62696c2f sr/
9ffffffffffff998 73746e6f662f666d 4f4c4f43003a2f2f mf/
9ffffffffffff9a8 6e673d4d52455452 6d7265742d656d6f RTE
9ffffffffffff9b8 534948006c616e69 35323d455a495354 ina
9ffffffffffff9c8 414e54534f480036 2e79616c703d454d 6.H
9ffffffffffff9d8 2e676e6174736f6d 4e474f4c006d6f63 mos
9ffffffffffff9e8 697661643d454d41 5f54494e49006d64 AME
```

Illustration 9: The Data Window Showing argv and envp Strings in Hexadecimal

Typing hexadecimal numbers is error-prone, and Ski provides several shortcuts to avoid it. The first is ***xski***'s Command History, an unlabeled window pane just above the "Command" area in the Main Window. As you execute commands, they move up to the Command History. Later, you can bring them back into the Command area. A single click brings a command back for you to edit. A double click brings the command back and re-executes it immediately. Try the Command History by doing this: Type "**`dj 0`**" to jump the Data Window to location 0. The Main Window should look like Illustration 10: The Main Window Showing Commands in the Command History. Then click on the "**`dj 9ffffffffffff938`**" command in the Command History. Hit the enter/return key to execute it.



```
load hello
dj 9ffffffffffff780
dj 9ffffffffffff938
dj 0
```

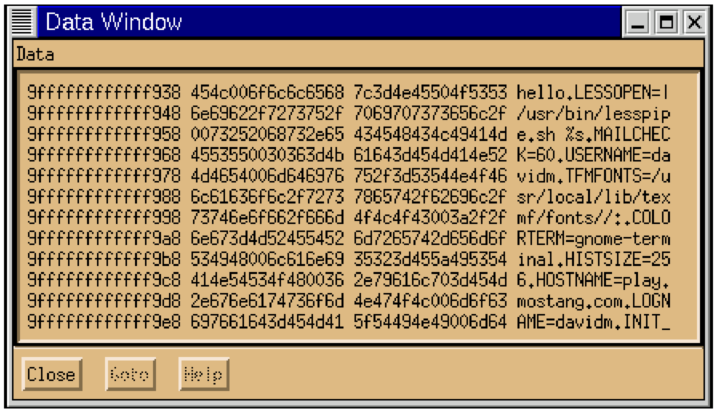
Illustration 10: The Main Window Showing Commands in the Command History

Another shortcut is the **`*`** pointer-dereference operator for indirect addressing. Type "**`dj 0`**" to jump the Data Window to location 0. Then type "**`dj *(r12+18)`**". Ski will take the contents of **`r12`** (**`9ffffffffff780`**, remember?), add **`18`** (hex) and use that as the address of the operand. The **`*`** operator fetches the contents of **`*(r12+18)`** and uses that value, **`9ffffffffff938`**, as the address to jump to. Compare the Data Window display resulting from "**`dj r12+18`**" with the display resulting from "**`dj *(r12+18)`**".

You will use the **`*`** operator a lot in debugging C programs because it performs the same function as C's **`*`** operator: it dereferences pointers. Unlike C's **`*`**, however, Ski's **`*`** operator is not type-specific: you can use it in any context where any kind of address is needed and you can use it to dereference registers like **`r12`**, memory locations, or anything that has a value. (This doesn't always make sense, of course. For example, dereferencing a floating-point register is rarely useful because floating-point registers don't hold pointers.)

Viewing Data in ASCII

Hexadecimal is no fun. To expose the ASCII translation, use your window manager's standard mechanism to make the Data Window wider. (How you do this depends on the window manager you're using, but generally this can be accomplished by grabbing the edge of the Data Window with your mouse cursor and dragging it to the right.) You should see approximately Illustration 11: The Data Window Showing argv and envp Strings in ASCII. Now click on the Main Window, to make it the active window again. Try the "**`df`**" ("**`d`**ata **`f`**orwards") and "**`db`**" ("**`d`**ata **`b`**ackwards") commands without operands to move forwards and backwards in the Data Window, one screenful each time.



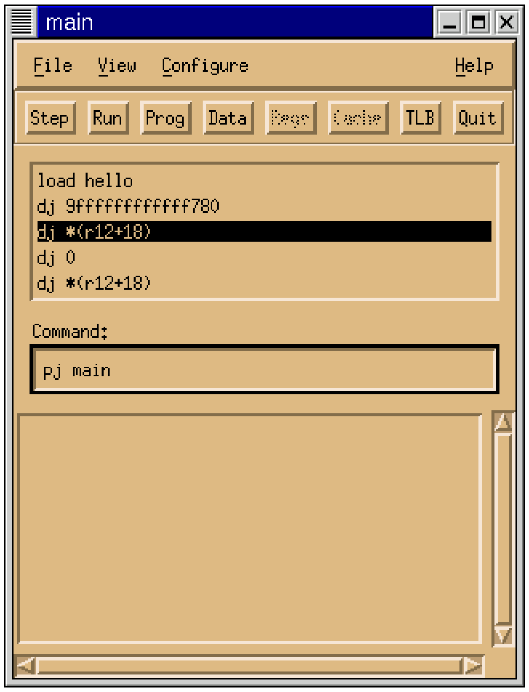
```
9fffffffffff938 454c006f6c6c6568 7c3d4e45504f5353 hello.LESSOPEN=|
9fffffffffff948 6e69622f7273752f 7069707373656c2f /usr/bin/lesspip
9fffffffffff958 0073252068732e65 434548434c49414d e.sh %s.MAILCHEC
9fffffffffff968 4553550030363d4b 61643d454d414e52 K=60.USERNAME=da
9fffffffffff978 4d4654006d646976 752f3d53544e4f46 vidm.TFMFONTS=/u
9fffffffffff988 6c61636f6c2f7273 7865742f62696c2f sr/local/lib/tex
9fffffffffff998 73746e6f662f666d 4f4c4f43003a2f2f mf/fonts//:.COLO
9fffffffffff9a8 6e673d4d52455452 6d7265742d656d6f RTERM=gnome-term
9fffffffffff9b8 534948006c616e69 35323d455a495354 inal.HISTSIZE=25
9fffffffffff9c8 414e54534f480036 2e79616c703d454d 6.HOSTNAME=play.
9fffffffffff9d8 2e676e6174736f6d 4e474f4c006d6f63 mostang.com.LOGN
9fffffffffff9e8 697661643d454d41 5f54494e49006d64 AME=davidm.INIT_
```

Illustration 11: The Data Window Showing argv and envp Strings in ASCII

Looking at Code

Initially, the Program Window shows the beginning of the program. For C programs, this isn't the first line of user code, it's the start-up routine from `crt1.o` that provides an interface between the operating system environment and the ANSI C environment. This routine is named "**`_start`**" and the ELF header in `hello` names it as the start of the program. That's what Ski shows in the Program Window by default: the start of the program according to ELF.

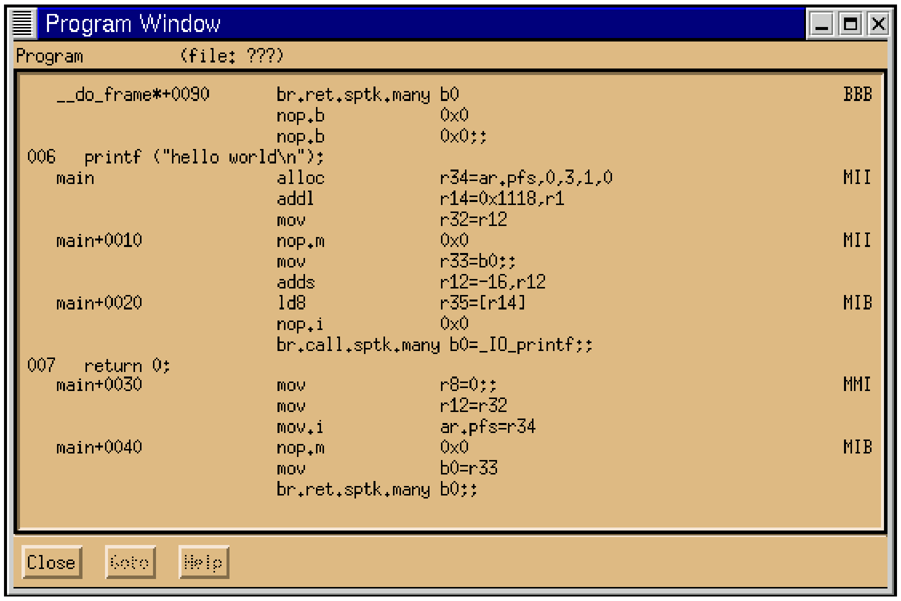
You use the "**`pj`**" command ("**`p`**rogram **`j`**ump") to jump the program window elsewhere. For example, jump it to the first instruction in the user's main(), as shown in Illustration 12: Jumping the Program Window to the Beginning of main(). The Program Window now looks like Illustration 13: The Program Window Showing Code at the Beginning of main(). You can move the Program Window forwards and backwards through program code with the "**`pf`**" ("**`p`**rogram **`f`**orwards") and "**`pb`**" ("**`p`**rogram **`b`**ackwards") commands, respectively. Try these commands, and then try using "**`pj`**" without an operand: note how it jumps you back and forth between the previous and current locations. The "**`dj`**" command does the same thing in the Data Window. Handy, eh?



```
load hello
dj 9ffffffffffff780
dj *(r12+18)
dj 0
dj *(r12+18)
```

```
pj main
```

Illustration 12: Jumping the Program Window to the Beginning of main()



```
  __do_frame*+0090      br.ret.sptk.many b0                              BBB
                        nop.b            0x0
                        nop.b            0x0;;
006   printf ("hello world\n");
   main                 alloc            r34=ar.pfs,0,3,1,0              MII
                        addl             r14=0x1118,r1
                        mov              r32=r12
   main+0010            nop.m            0x0                             MII
                        mov              r33=b0;;
                        adds             r12=-16,r12
   main+0020            ld8              r35=[r14]                       MIB
                        nop.i            0x0
                        br.call.sptk.many b0=_IO_printf;;
007   return 0;
   main+0030            mov              r8=0;;                          MMI
                        mov              r12=r32
                        mov.i            ar.pfs=r34
   main+0040            nop.m            0x0                             MIB
                        mov              b0=r33
                        br.ret.sptk.many b0;;
```

Illustration 13: The Program Window Showing Code at the Beginning of main()

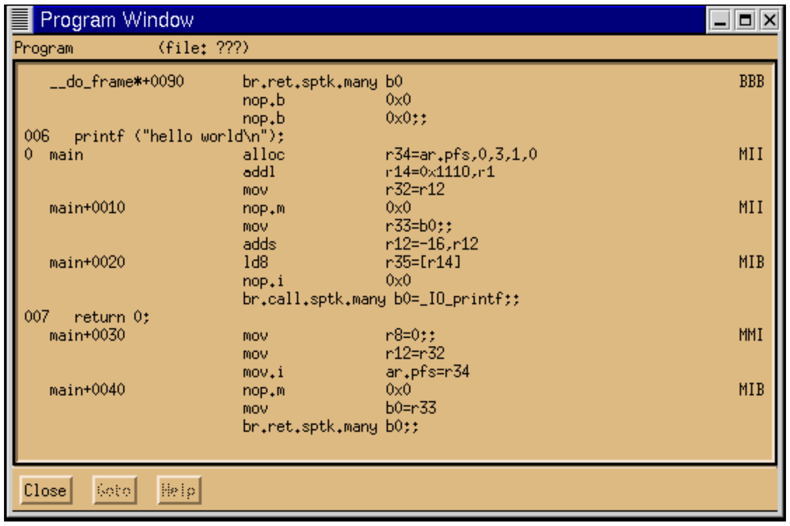
Viewing Source Code Mixed In with Assembly Code

The Program Window shows the C source code intermixed with the IA-64 assembly code. You can turn the source code display off or on using the **`pa`** ("**p**rogram **a**ssembly") and **`pm`** ("**p**rogram **m**ixed") commands, respectively. Mixed code display only works if you have the source code to the program available to Ski; the source code isn't embedded in the ELF file. Also, you must compile your code with the appropriate compiler flags, for example, with the **`-g`** flag used by many C compilers to generate debug line record information. If your program is composed of multiple object files, for example "**`cc -o test foo.o bar.o baz.o`**", Ski can only show source code from the files compiled with the **`-g`** flag. Make sure the Program Window is in mixed mode for now.

Controlling Breakpoints

You can think of Ski as a debugger that happens to work on a simulated processor rather than a real processor. Like any good debugger, Ski provides breakpoints. To set a breakpoint in an IA-64 program, use the "**`bs`**" command ("**b**reakpoint **s**et"). In the example that follows, you will want to have the Program Window display the area of code near main(). Use the command "**`pj main`**", as you learned above.

To set a breakpoint at the beginning of main(), type "**`bs main`**" in the Main Window. The Program Window shows a "**`0`**" in the first column of the window at the breakpoint location (the alloc'instruction), because you just used breakpoint #0, as Illustration 14: The Program Window Showing a Breakpoint at main() shows. (The first three columns are also used for line numbers.) Set a breakpoint at **`main+10`** and another at **`main+20`**. Ski lets you set up to ten breakpoints.

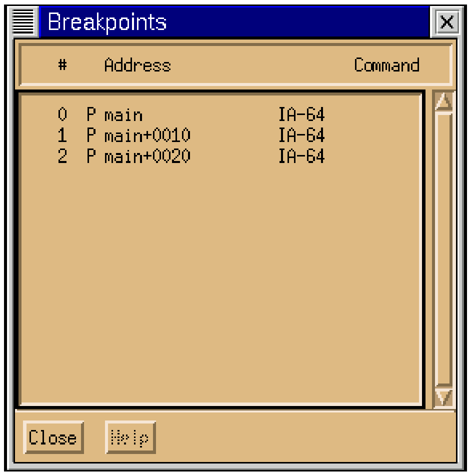


```
Program          (file: ???)
   __do_frame*+0090       br.ret.sptk.many b0                     BBB
                          nop.b            0x0
                          nop.b            0x0;;
006   printf ("hello world\n");
0  main                   alloc            r34=ar.pfs,0,3,1,0     MII
                          addl             r14=0x1110,r1
                          mov              r32=r12
   main+0010              nop.m            0x0                    MII
                          mov              r33=b0;;
                          adds             r12=-16,r12
   main+0020              ld8              r35=[r14]              MIB
                          nop.i            0x0
                          br.call.sptk.many b0=_IO_printf;;
007   return 0;
   main+0030              mov              r8=0;;                 MMI
                          mov              r12=r32
                          mov.i            ar.pfs=r34
   main+0040              nop.m            0x0                    MIB
                          mov              b0=r33
                          br.ret.sptk.many b0;;
```

Illustration 14: The Program Window Showing a Breakpoint at main()

Use the "**`bl`**" command ("**b**reakpoint **l**ist") to see a list of the breakpoints, as shown in Illustration 15: The Breakpoint List Window. If you prefer using a mouse, use the "Breakpoints" item on the View menu instead of the "**`bl`**" command. When you are finished viewing the breakpoint list, click its *`Close`* button to dismiss the window.

To delete breakpoints individually, use the "**`bd`**" command ("**b**reakpoint **d**elete"). Use the "**`bD`**" command ("**b**reakpoint **D**elete all") to delete all breakpoints at once. Delete all your breakpoints before continuing this tutorial.



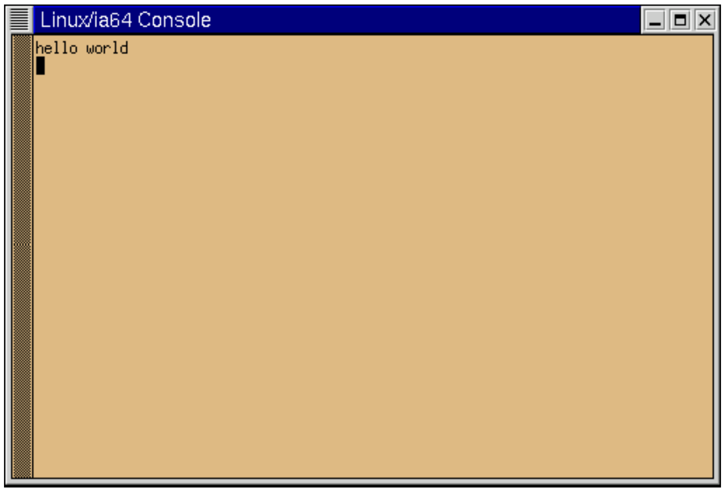
```
 #   Address                    Command

 0  P main                IA-64
 1  P main+0010           IA-64
 2  P main+0020           IA-64
```

Illustration 15: The Breakpoint List Window

Running a Program

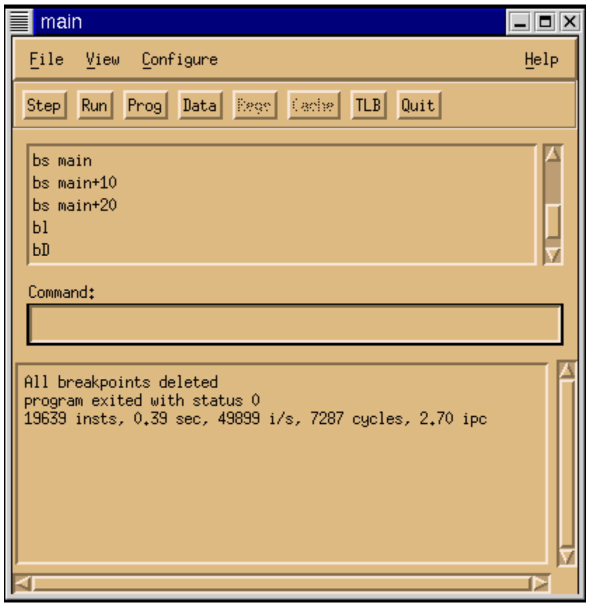
To run your program, type the "**`run`**" command or click the *`Run`* button in the Main Window. Ski will start the simulation and connect the program's standard I/O ports (stdin, stdout, and stderr) to Ski's standard ports. For example, assuming there are no breakpoints still set in `hello`, you will see "hello world" printed out when you run it, as Illustration 16: The Terminal Window After the "hello" Program is Run shows, and run statistics will appear in the Main Window, as Illustration 17: The xski Main Window after the "hello" Program is Run shows. The statistics tell you how many instructions were simulated and how much time it took, the instructions-per-second rate, the number of IA-64 processor cycles that were consumed on the simulated CPU, and the average number of instructions per cycle, which provides an indication of the best-case effective parallelism of the program. (Ski simulates all the instructions in an instruction group in one cycle; a hardware implementation may not be as capable.)

Ski will stop the simulation for three reasons: if a breakpoint is reached, if the IA-64 program attempts to access privileged resources or non-existent memory, or if the program ends normally by calling exit() or similar functions. If simulation stops due to a breakpoint, you can continue simulation with the "**`cont`**" command ("**`cont`**inue") or you can step through the simulation with the "**`step`**" command or *`Step`* button. You cannot re-run a program, nor can you re-load it and start over. You must exit and re-enter ***xski*** and then reload your program.



```
hello world
```

Illustration 16: The Terminal Window After the "hello" Program is Run



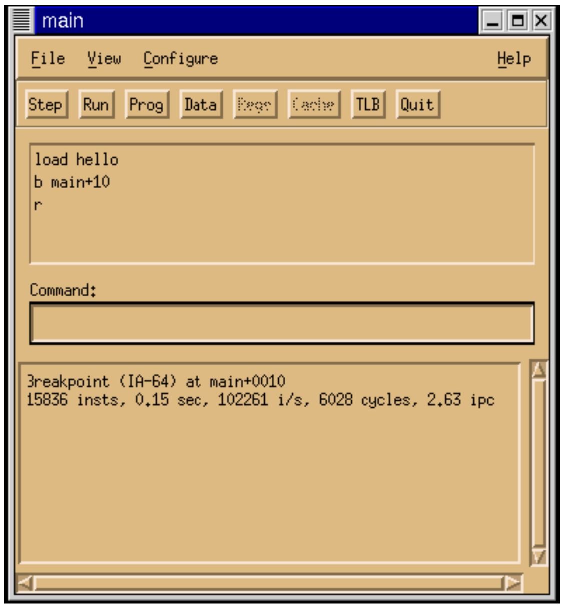
```
bs main
bs main+10
bs main+20
bl
bD
```

```
All breakpoints deleted
program exited with status 0
19639 insts, 0.39 sec, 49899 i/s, 7287 cycles, 2.70 ipc
```

Illustration 17: The xski Main Window after the "hello" Program is Run

Single-stepping a Program

To try single-stepping (and no, this is not a kind of ethnic dance), set a breakpoint at **`main+10`**. Then use the "**`run`**" command or *Run* button to simulate the program up to the breakpoint. (If you receive the error message "**`Nothing to run`**", stop and reread the last sentence in the previous paragraph.) Ski stops at the breakpoint and notifies you with a message in the Main Window. Ski tells you why it stopped and gives you statistics about program execution up to this point, as you can see in Illustration 18: The Main Window After Reaching the Breakpoint at main+10. The Program Window marks the next instruction to be fetched with a greater-than symbol in the second column. If the instruction is predicated off, Ski uses an asterisk instead of a greater-than symbol, and shows the predication register in parentheses.



```
load hello
b main+10
r
```

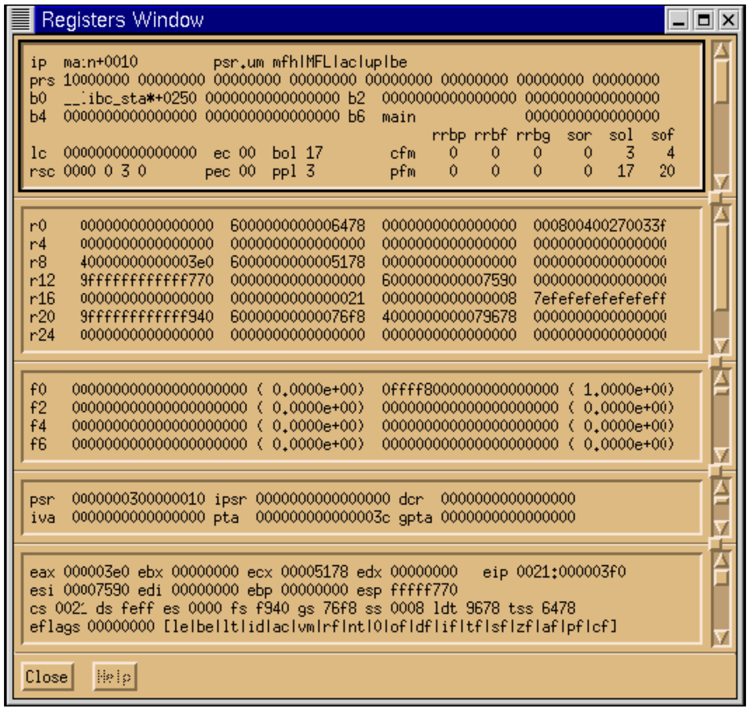
```
Breakpoint (IA-64) at main+0010
15836 insts, 0.15 sec, 102261 i/s, 6028 cycles, 2.63 ipc
```

Illustration 18: The Main Window After Reaching the Breakpoint at main+10

Move and resize your windows so the Main Window and Program Window don't overlap. Now use the "**`step`**" command or *`Step`* button to execute one instruction. Note that the greater-than symbol moves down one line: Ski keeps track of IA-64 bundles and groups but it simulates individual instructions. You can follow the "**`step`**" command with a (decimal) number to specify how many steps Ski should take, for example, "**`step 10`**" to execute ten instructions. As a shortcut, shift-clicking on the *`Step`* button causes Ski to take ten steps. Most Ski commands can be abbreviated, as described in "Command Reference" on page 87. The **`step`** command can be abbreviated as "**`s`**".

Changing Registers and Memory

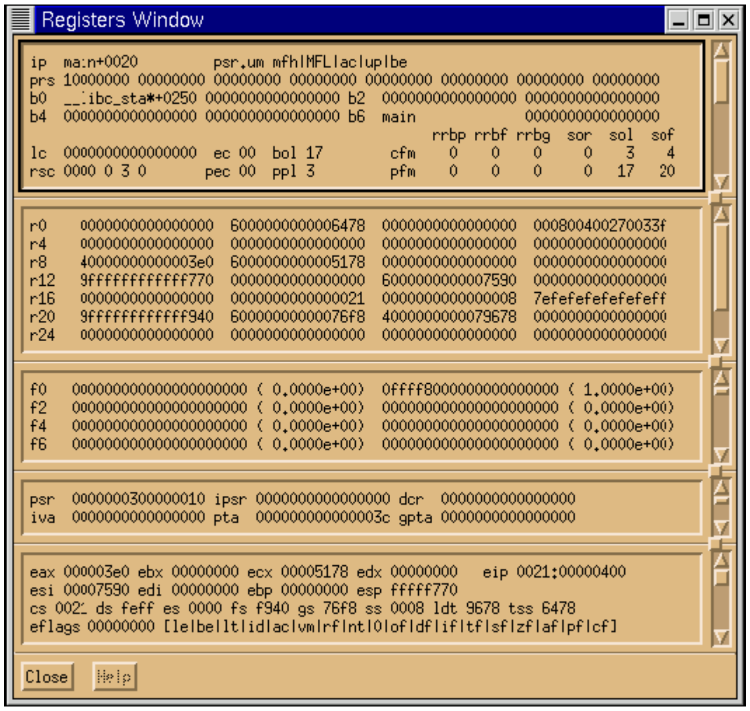
To debug a program, you usually need to inspect and alter registers and memory. The first three panes in the Register Window shows the registers of most concern to application programmers: user registers in the first pane, general registers in the second pane, and floating point registers in the third pane, as you can see in Illustration 19: The xski Register Window After Stopping at a Breakpoint at main+10.



```
ip  ma:n+0010          psr.um mfh|MFL|ac|up|be
prs 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
b0  __:ibc_sta*+0250 0000000000000000 b2  0000000000000000 0000000000000000
b4  0000000000000000 0000000000000000 b6  main             0000000000000000
                                             rrbp rrbf rrbg  sor  sol  sof
lc  0000000000000000  ec 00  bol 17       cfm    0    0    0    0    3    4
rsc 0000 0 3 0       pec 00  ppl 3        pfm    0    0    0    0   17   20

r0    0000000000000000  6000000000006478  0000000000000000  000800400270033f
r4    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r8    40000000000003e0  6000000000005178  0000000000000000  0000000000000000
r12   9ffffffffffff770  0000000000000000  6000000000007590  0000000000000000
r16   0000000000000000  0000000000000021  0000000000000008  7efefefefefefeff
r20   9ffffffffffff940  60000000000076f8  4000000000079678  0000000000000000
r24   0000000000000000  0000000000000000  0000000000000000  0000000000000000

f0   000000000000000000000 ( 0.0000e+00)  0ffff8000000000000000 ( 1.0000e+00)
f2   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f4   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f6   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)

psr  0000000300000010 ipsr 0000000000000000 dcr  0000000000000000
iva  0000000000000000 pta  000000000000003c gpta 0000000000000000

eax 000003e0 ebx 00000000 ecx 00005178 edx 00000000   eip 0021:000003f0
esi 00007590 edi 00000000 ebp 00000000 esp fffff770
cs 002: ds feff es 0000 fs f940 gs 76f8 ss 0008 ldt 9678 tss 6478
eflags 0000000 [le|be|lt|id|ac|vm|rf|nt|0|of|df|if|tf|sf|zf|af|pf|cf]
```

Illustration 19: The xski Register Window After Stopping at a Breakpoint at main+10

By changing the value of the **`ip`** register, you can change where in the program Ski will resume simulation. Enter the command "**`= ip main+20`**" in the Main Window and observe the first line of the first pane in the Register Window: notice that the **`ip`** register changes to reflect your command, as Illustration 20: The xski Register Window After Changing the ip Register shows. (You may need to left-click in the Main Window to make it active.) You can make similar changes to all of the architecturally-visible, non-hardwired IA-64 registers, which helps you debug your program. You can test your program's behavior in exceptional cases, such as handling unusual errors.



```
ip   ma:n+0020           psr.um mfh|MFL|ac|up|be
prs 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
b0   __:ibc_sta*+0250 0000000000000000 b2  0000000000000000 0000000000000000
b4   0000000000000000 0000000000000000 b6  main             0000000000000000
                                              rrbp rrbf rrbg  sor  sol  sof
lc   0000000000000000  ec 00  bol 17        cfm   0    0    0    0    3    4
rsc 0000 0 3 0        pec 00  ppl 3         pfm   0    0    0    0   17   20
```

```
r0    0000000000000000  6000000000006478  0000000000000000  000800400270033f
r4    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r8    40000000000003e0  6000000000005178  0000000000000000  0000000000000000
r12   9ffffffffffff770  0000000000000000  6000000000007590  0000000000000000
r16   0000000000000000  0000000000000021  0000000000000008  7efefefefefefeff
r20   9ffffffffffff940  60000000000076f8  4000000000079678  0000000000000000
r24   0000000000000000  0000000000000000  0000000000000000  0000000000000000
```

```
f0   00000000000000000000 ( 0.0000e+00)  0ffff8000000000000000 ( 1.0000e+00)
f2   00000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f4   00000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f6   00000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
```

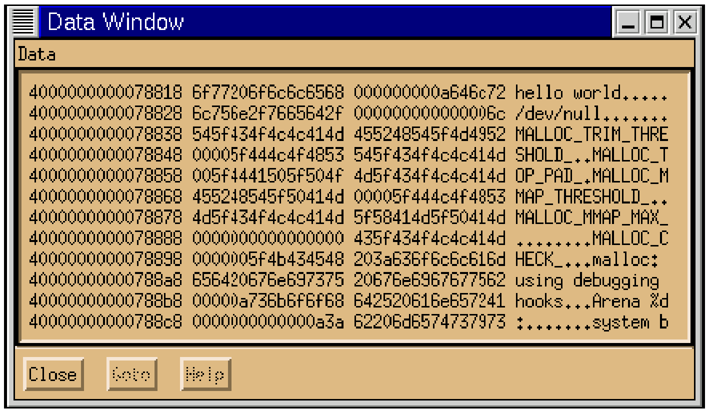
```
psr  0000000300000010 ipsr 0000000000000000 dcr  0000000000000000
iva  0000000000000000 pta  000000000000003c gpta 0000000000000000
```

```
eax 000003e0 ebx 00000000 ecx 00005178 edx 00000000   eip 0021:00000400
esi 00007590 edi 00000000 ebp 00000000 esp fffff770
cs 0021 ds feff es 0000 fs f940 gs 76f8 ss 0008 ldt 9678 tss 6478
eflags 00000000 [le|be|lt|id|ac|vm|rf|nt|0|of|df|if|tf|sf|zf|af|pf|cf]
```

Illustration 20: The xski Register Window After Changing the ip Register

Changing registers isn't enough to debug most programs, however. Often, you need to change values in memory as well. Ski provides several commands for this, differing in whether they modify one-byte chunks, two-byte chunks, four-byte chunks, eight-byte chunks, or variable-length C-language text strings. For example, instead of "hello world", you can have the program output "Ski!Ski!Ski!". You can do this by using the "**`=s`**" command ("**`= s`**tring") to modify the data stored at the address "**`_IO_stdin_used+8`**". (The string may be stored at a different address in your program. If so, use the Data Window to locate the string and then use the corresponding address instead.) Here's what to do:

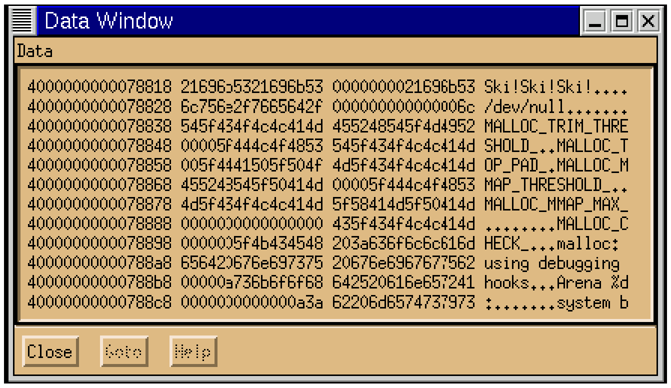
First, make sure the Data Window is wide enough to show ASCII translations along with hexadecimal, as in Illustration 21: The xski Data Window Widened to Show ASCII. To avoid confusion, make sure the Data Window doesn't overlap the Main Window.



```
4000000000078818 6f77206f6c6c6568 000000000a646c72 hello world.....
4000000000078828 6c756e2f7665642f 000000000000006c /dev/null.......
4000000000078838 545f434f4c4c414d 455248545f4d4952 MALLOC_TRIM_THRE
4000000000078848 00005f444c4f4853 545f434f4c4c414d SHOLD_..MALLOC_T
4000000000078858 005f441505f504f 4d5f434f4c4c414d OP_PAD_.MALLOC_M
4000000000078868 455248545f50414d 00005f444c4f4853 MAP_THRESHOLD_..
4000000000078878 4d5f434f4c4c414d 5f58414d5f50414d MALLOC_MMAP_MAX_
4000000000078888 0000000000000000 435f434f4c4c414d ........MALLOC_C
4000000000078898 0000005f4b434548 203a636f6c6c616d HECK_...malloc:
40000000000788a8 656420676e697375 20676e6967677562 using debugging
40000000000788b8 00000a736b6f6f68 642520616e657241 hooks...Arena %d
40000000000788c8 0000000000000a3a 62206d6574737973 :.......system b
```

Illustration 21: The xski Data Window Widened to Show ASCII

Next, issue the command "**`=s _IO_stdin_used+8 Ski!Ski!Ski!`**" in the Main Window. (You may need to left-click in the Main Window to make it active.) Observe how the Data Window changes: the hexadecimal values at, and after, **`_IO_stdin_used+8`** have changed, as have their corresponding ASCII translations, and a null byte (the value zero) has been added to the end of your string to make it a valid C-language string. Compare Illustration 21: The xski Data Window Widened to Show ASCII and Illustration 22: The xski Data Window After Changing the "Hello, world" String.

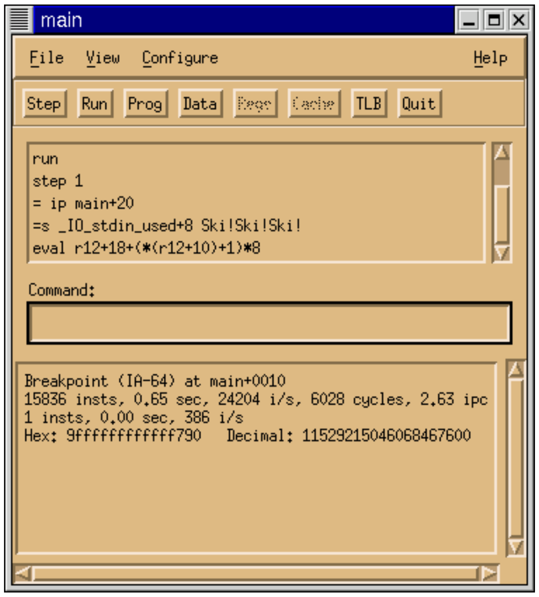


```
4000000000078818 21696b5321696b53 0000000021696b53 Ski!Ski!Ski!....
4000000000078828 6c756e2f7665642f 000000000000006c /dev/null.......
4000000000078838 545f434f4c4c414d 455248545f4d4952 MALLOC_TRIM_THRE
4000000000078848 00005f444c4f4853 545f434f4c4c414d SHOLD_..MALLOC_T
4000000000078858 005f4441505f504f 4d5f434f4c4c414d OP_PAD_.MALLOC_M
4000000000078868 455243545f50414d 00005f444c4f4853 MAP_THRESHOLD_..
4000000000078878 4d5f434f4c4c414d 5f58414d5f50414d MALLOC_MMAP_MAX_
4000000000078888 0000000000000000 435f434f4c4c414d ........MALLOC_C
4000000000078898 0000005f4b434548 203a636f6c6c616d HECK_...malloc: 
40000000000788a8 6564206676e697375 20676e6967677562 using debugging
40000000000788b8 00000a736b6f6f68 642520616e657241 hooks...Arena %d
40000000000788c8 0000000000000a3a 62206d6574737973 :.......system b
```

Illustration 22: The xski Data Window After Changing the "Hello, world" String

The commands to change one, two, four, and eight byte quantities are **=1**, **=2**, **=4**, and **=8**, respectively. They are described in detail in "Changing Registers and Memory with Assignment Commands" on page 75 and in "Command Reference on page 87.

Often, you will need to evaluate formulas. For example, to find the address of the first **envp** string, you would need to compute the sum of the contents of **r12** and **18** (hex) and then add the length of the **argv** vector (**argc+1**) multiplied by eight (the size of a **char *** on IA-64). To do this, you use the "**eval**" command in the Main Window, as shown in Illustration 23: The xski Main Window Showing an eval Command and Its Result. (The use of the "*****" operator was discussed in "Inspecting Data" on page 15.) As you see, the result is shown in decimal and hexadecimal.



```
run
step 1
= ip main+20
=s _IO_stdin_used+8 Ski!Ski!Ski!
eval r12+18+(*(r12+10)+1)*8
```

```
Breakpoint (IA-64) at main+0010
15836 insts, 0.65 sec, 24204 i/s, 6028 cycles, 2.63 ipc
1 insts, 0.00 sec, 386 i/s
Hex: 9ffffffffffff790   Decimal: 11529215046068467600
```

Illustration 23: The xski Main Window Showing an eval Command and Its Result

Getting Help

To see what commands are available, type "**`help`**" in the Main Window or use the Help->Commands menu selection. To see the syntax of a specific command, type "**`help`**" followed by the command name, as in "**`help eval`**".

Next Steps

Congratulations! You now know how to use ***xski*** to test an IA-64 program. In the rest of this manual, you'll find out how to use ***ski*** and ***bski*** and the many additional commands and facilities not covered in this brief tutorial.

- The "Overview" chapter, presents the capabilities of Ski, how to start it and stop it, and a brief discussion of installation issues. The chapter also shows how to use ***bski*** for batch simulation.
- The "Screen Presentation" chapter, discusses the various screen displays of ***xski*** and ***ski*** in depth.
- The "Command Language" chapter, defines the syntax of the language you use to control Ski's operation.
- The "Screen Manipulation Commands" chapter, presents the Ski commands for controlling Ski's screen displays.
- The "Program Simulation" chapter, introduces the concepts of Ski program simulation, shows you how to load programs, and presents the Ski commands for simulating a program. Much of the information needed to use Ski for firmware development and operating system simulation is in this chapter.
- The "Linux and MS-DOS ABI Emulation" chapter, discusses the Ski mechanisms and support for simulating application programs. If you are using Ski for to develop system software, such as bootstrap firmware or operating systems, you can skip this chapter.
- The "Debugging" chapter, presents Ski commands and facilities that are useful in debugging and tuning programs.
- The "Command Files" chapter, introduces command files, a mechanism that lets you extend Ski to meet your particular needs.
- The remaining chapters contain summaries of the Ski command set, a list of the registers and internal variables Ski recognizes, a description of the Ski error and status messages, their causes, and any possible solutions., and any applicable licenses.

2 Overview

Introduction

The Ski simulator is a software package designed to functionally simulate the IA-64 processor architecture at the instruction level. Ski offers an informative, screen-oriented machine state display and a friendly, powerful command interface. Programs may be loaded from disk in executable format; they may be run from start to finish, single-stepped, and breakpointed. Translation lookaside buffers may be simulated. Certain Linux and MS-DOS operating system functions (system calls) are provided for simulation of application programs. These capabilities are complemented by screen-oriented symbolic debugging to provide a view into the simulated IA-64 processor.

Ski's Strengths

Ski is particularly well-suited for:

- IA-64 application development:

  Ski can simulate IA-64 programs in a Linux environment and IA-32 programs in an MS-DOS environment. Ski provides a user interface that looks very much like a typical debugger but the processor you are debugging on is virtual, simulated by Ski. Ski has successfully executed the SPEC-92 and SPEC-95 benchmark suites.

- IA-64 compiler tuning:

  Ski provides performance statistics that can help you tune IA-64 compiler code generators. Ski can help you improve your compiler's use of IA-64 architectural enhancements for parallelism.

- IA-64 operating system and firmware development:

  Ski can simulate a "raw" IA-64 processor, with no operating system provided. Because of this, you can use Ski to simulate an IA-64 operating system running IA-64 and IA-32 programs. For example, Ski has been used successfully to develop the IA-64 version of the Linux kernel.

- IA-64 processor functional hardware verification:

  Ski is a true implementation of the IA-64 architecture. You can compare the behavior of code simulated with Ski to the same code running on other IA-64 implementations. This helps you verify the correctness of those implementations.

Ski's Scope

Many different kinds of simulators can be created: device simulators that function at the semiconductor quantum physics level, circuit simulators that model the behavior of small numbers of transistors and other circuit elements, gate simulators that model digital circuits at the boolean logic level, and so on. Ski is an instruction simulator, which makes it very fast. Ski doesn't model any particular physical IA-64 implementation. Instead, it models an architecturally-compliant IA-64 processor with extensive compute resources.

What You Need to Know to Use This Manual

This manual describes the user interface of Ski in detail. In reading this manual, you will learn how to use Ski to simulate your IA-64 and IA-32 programs. To understand this manual, you should already be familiar with the IA-64 architecture. IA-64 abbreviations such as **`ip`**, **`psr`**, and **`eax`** are used without explanation.

Defects and Defect Reporting

Ski is provided "as is", without any guarantees or warranties. However, a mailing list has been created for reporting Ski defects and for general Ski discussions. See the release notes for details on the mailing list address and how to subscribe.

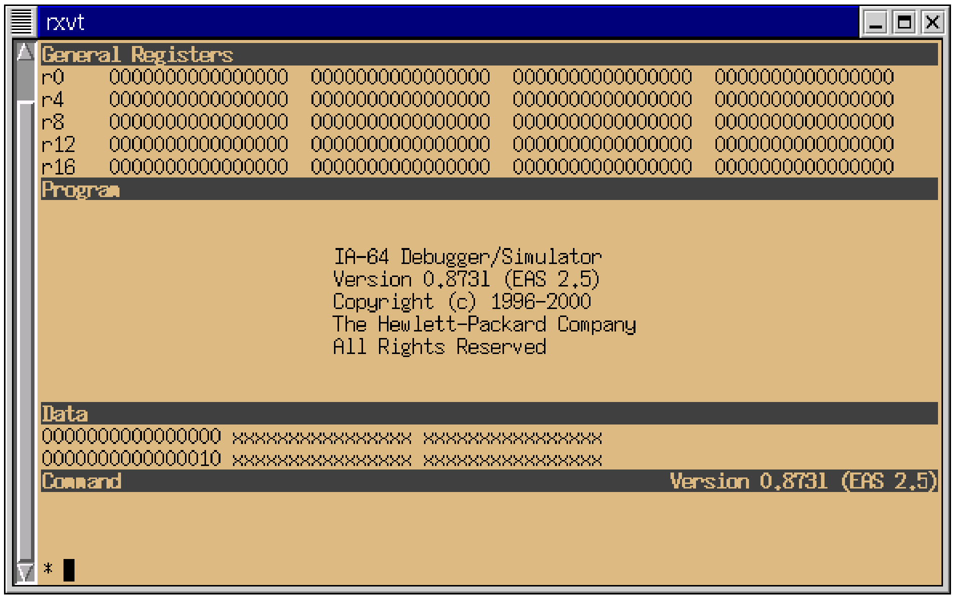
Ski Variations

The simulator is available in three varieties, distinguished by their user interfaces: ***ski***, ***xski***, and ***bski***. The underlying simulation engine is identical across all three varieties. The figures below show how each variety looks when first started. Illustration 24: The Curses-based ski Interface shows ***ski***, which uses a terminal-oriented, curses-based, character user interface. Illustration 25: The X Window System, Motif-based xski Interface shows ***xski***, using an X Window System, Motif-based, graphical user interface. Illustration 26: The Command-Line bski Interface shows ***bski***, which provides a batch-oriented, command-line-driven environment and no user interface. Ski command line flags, some of which are shown in Illustration 26: The Command-Line bski Interface are described in "Command Line Flags".

The three varieties understand the same command language. There are a few, unavoidable differences and they are pointed out where appropriate in this manual. Most examples and sample screen displays are taken from ***xski*** sessions. All examples have been verified in actual use.

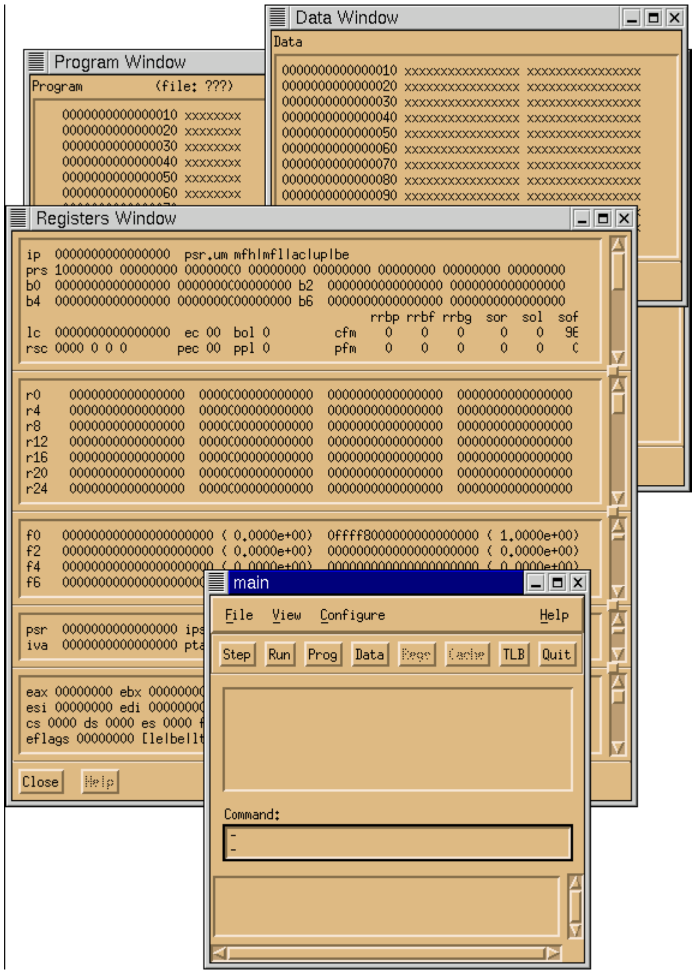
Using *bski* for Batch Simulations

Because ***bski*** has no user interface, you typically control it using a command file (see "Command Files" on page 84) and the **`-i`** command line flag (see "Command Line Flags"). ***ski*** and ***xski*** are intended for you to use interactively, while ***bski*** excels at batch simulations that might run for a long time as background jobs on your workstation or on a higher-powered remote simulation server. The `cron` and `make` programs work well with ***bski***. With `cron`, you can schedule simulations to run at night and on remote servers. With `make`, you can execute complex networks of tests quickly, letting `make` keep track of the dependencies between the tests. These programs are documented in man pages.



```
General Registers
r0   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r4   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r8   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r12  0000000000000000  0000000000000000  0000000000000000  0000000000000000
r16  0000000000000000  0000000000000000  0000000000000000  0000000000000000
Program


                          IA-64 Debugger/Simulator
                          Version 0.8731 (EAS 2.5)
                          Copyright (c) 1996-2000
                          The Hewlett-Packard Company
                          All Rights Reserved


Data
0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000010 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
Command                                          Version 0.8731 (EAS 2.5)


*
```

Illustration 24: The Curses-based ski Interface

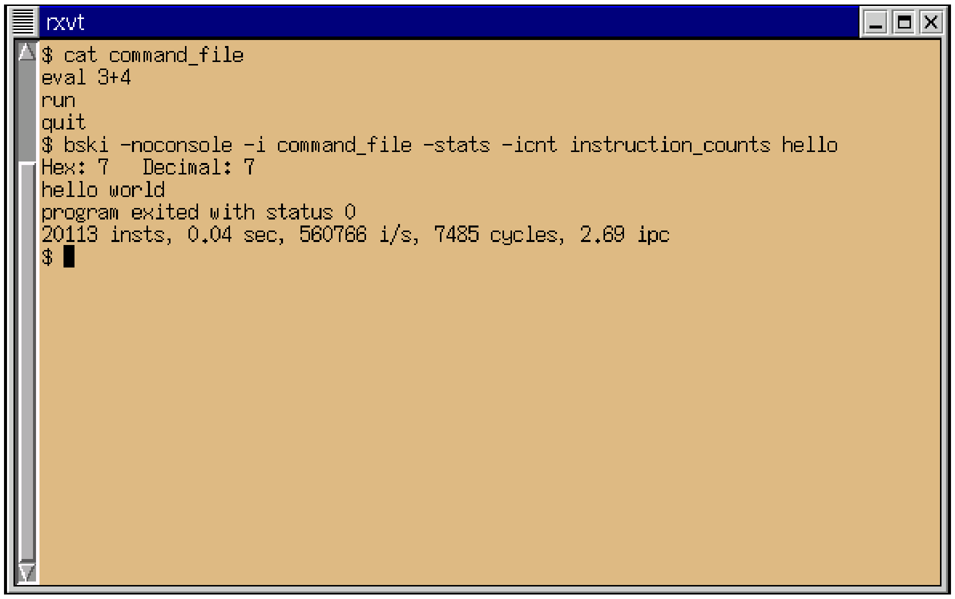


```
Data

0000000000000010 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000020 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000030 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000040 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000050 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000060 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000070 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000080 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000090 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
```

```
Program          (file: ???)

    0000000000000010 xxxxxxxx
    0000000000000020 xxxxxxxx
    0000000000000030 xxxxxxxx
    0000000000000040 xxxxxxxx
    0000000000000050 xxxxxxxx
    0000000000000060 xxxxxxxx
```

```
ip  0000000000000000  psr.um mfh|mfl|ac|up|be
prs 10000000 00000000 000000C0 00000000 00000000 00000000 00000000 00000000
b0  0000000000000000 0000000C00000000 b2  0000000000000000 0000000000000000
b4  0000000000000000 0000000C00000000 b6  0000000000000000 0000000000000000
                                            rrbp rrbf rrbg  sor  sol  sof
lc  0000000000000000  ec 00  bol 0        cfm   0    0    0    0    0   9E
rsc 0000 0 0 0       pec 00  ppl 0        pfm   0    0    0    0    0    C

r0    0000000000000000  0000C00000000000  0000000000000000  0000000000000000
r4    0000000000000000  0000C00000000000  0000000000000000  0000000000000000
r8    0000000000000000  0000C00000000000  0000000000000000  0000000000000000
r12   0000000000000000  0000C00000000000  0000000000000000  0000000000000000
r16   0000000000000000  0000C00000000000  0000000000000000  0000000000000000
r20   0000000000000000  0000C00000000000  0000000000000000  0000000000000000
r24   0000000000000000  0000C00000000000  0000000000000000  0000000000000000

f0   000000000000000000000 ( 0.0000e+00)  0ffff8000000000000000 ( 1.0000e+00)
f2   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f4   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f6   0000000000000000000

psr  0000000000000000 ips
iva  0000000000000000 pta

eax 00000000 ebx 0000000
esi 00000000 edi 0000000
cs 0000 ds 0000 es 0000 f
eflags 00000000 [le|be|lt
```

Illustration 25: *The X Window System, Motif-based xski Interface*



```
$ cat command_file
eval 3+4
run
quit
$ bski -noconsole -i command_file -stats -icnt instruction_counts hello
Hex: 7   Decimal: 7
hello world
program exited with status 0
20113 insts, 0.04 sec, 560766 i/s, 7485 cycles, 2.69 ipc
$ 
```

Illustration 26: The Command-Line bski Interface

Starting Ski

To start the Ski simulator, type its name (***ski***, ***xski***, or ***bski***) and any necessary command line options and file redirections, just as you would start any other Linux program. (Command line options are described in "Command Line Flags".) The simplest invocation of the simulator is:

`ski`

> This starts the (curses-based) ***ski*** version of the simulator with no program loaded: a "bare" IA-64 emulation is ready for you to use.

A more sophisticated invocation would be:

`xski my_program`

> This starts the (X/Motif-based) ***xski*** version of the simulator and loads the IA-64 executable file `my_program`, ready to run. The program will not receive any command line arguments (via the argc/argv mechanism) when you run it.

To run the simulator as a batch job in the background on an all-night run, you might execute this command line:

`bski -noconsole -stats -i my_commands my_program foo bar <test_data >out_stuff 2>bad_news &`

> This invokes the (batch) ***bski*** version of the simulator and loads the IA-64 executable file `my_program`, ready to run. The **`-noconsole`** flag tells ***bski*** not to create a separate console window for the program's standard I/O. The program will receive the command line arguments **`foo`** and **`bar`** via the argc/argv mechanism when ***bski*** runs it. Both the simulator and the program being simulated will have standard in, standard out, and standard err redirected from/to `test_data`, `out_stuff`, and `bad_news`, respectively, and the simulator will execute the commands in `my_commands`. (Ski never reads from standard in, so there is no possibility of confusion.) The **`-stats`** flag specifies that at the end of the run, collected statistics will be output to standard out (which is redirected). The ampersand ("**`&`**") runs the job in the background.

Command Line Flags

The simulator accepts certain flags on the command line when you start it up. The flags are passed on the command line in standard Linux fashion. The Ski command line syntax is shown below. The **`-i`**, **`-rest`**, **`-icnt`**, and **`-stats`** flags can appear in any order.

`ski` [**`-help`**] [**`-i`** *filename*] [**`-rest`** *filename*] [*program_filename* [*args*]+]

`xski` [**`-help`**] [**`-noconsole`**] [**`-i`** *filename*] [**`-rest`** *filename*] [*program_filename* [*args*]+]

`bski` [**`-help`**] [**`-noconsole`**] [**`-i`** *filename*] [**`-rest`** *filename*] [**`-icnt`** *filename*] [**`-stats`**] [*program_filename* [*args*] +]

Summary of Flags

`-help`

A list of flags accepted by this variety of Ski (***ski***, ***xski***, or ***bski***) is printed out. No other processing is done and Ski terminates.

`-i` *filename*

The specified file is run as a command file before the first prompt to the user. If an *program_filename* is provided on the same command line, the *program_filename* is loaded before the command file is run. This provides a convenient way to load a program, initialize other machine state, and then turn control over to the user.

`-icnt` *filename*

For ***bski*** only: This flag specifies instruction counts should be saved in the specified file. For each kind of instruction executed during the simulation, the instruction count file shows five fields of information:

- The instruction mnemonic
- The total number of times the instruction was executed
- The number of executions that were predicated on
- The number of executions that were predicated off
- The number of executions that were predicated on predicate register 0, which is "hardwired" on

The value in the second field equals the sum of the values in the last three fields.

`-noconsole`

For ***xski*** and ***bski*** only: This flag tells Ski not to create a separate console window for the simulated program's standard I/O. Instead, Ski will use the existing console window's for standard I/O purposes in the simulated program.

`-rest` *filename*

Restore the simulator run saved in *filename*. See "Saving and Restoring the Simulator State" on page 82. This flag cannot be combined with an *program_filename*. If combined with a **`-i`** flag, the **`-i`** flag is accepted and the **`-rest`** flag is silently ignored.

`-stats`

For ***bski*** only: specifies execution run-time and instruction rate information should be send to standard out (stdout) at the end of the run. This information is normally displayed in the Main/Command Window of ***xski*** and ***ski***. The **`-stats`** flag allows users of ***bski*** to get the same information.

The `XSki` File

xski's screen presentation is substantially controlled by the contents of the `XSki` file, which uses the X Window System's resource mechanism to provide information to ***xski***. You can edit this file to change ***xski***'s use of graphic buttons, described in "The xski Main Window" on page 48. The `XSki` file is part of the standard Ski distribution and you should put this file in your X Window System's `app-defaults` directory or in your home directory. If there is no valid `XSki` file, the simulator will not be usable. You can find more information on installing ***xski*** in the release notes that come with each Ski distribution.

Quitting Ski

The **`quit`** command causes the simulator to exit. If a numeric operand or expression is supplied, the value is returned to the shell as Ski's exit status. This can be particularly useful with ***bski*** and command files (see "Command Files" on page 84), for automated testing and regression testing. The exit status from Ski becomes the new value of your shell's **`$?`** variable (for most shells) and can also be retrieved automatically by the make program, if you use makefiles to control batch runs.

Summary of the Quit Command

`quit` [*expression*]

Terminates the simulator and returns control to the system, setting the exit status to *expression* (default is 0).

3 Screen Presentation

Ski's Use of Windows

xski and ***ski*** generally divide the screen into four windows. (***bski*** doesn't create any windows because it has no user interface, only a command line interface.) ***xski*** uses Motif windows which you can move and resize using the mechanisms provided by your window manager (WindowMaker, Englightenment, fvwm, twm, etc.) ***xski*** creates additional windows as necessary.

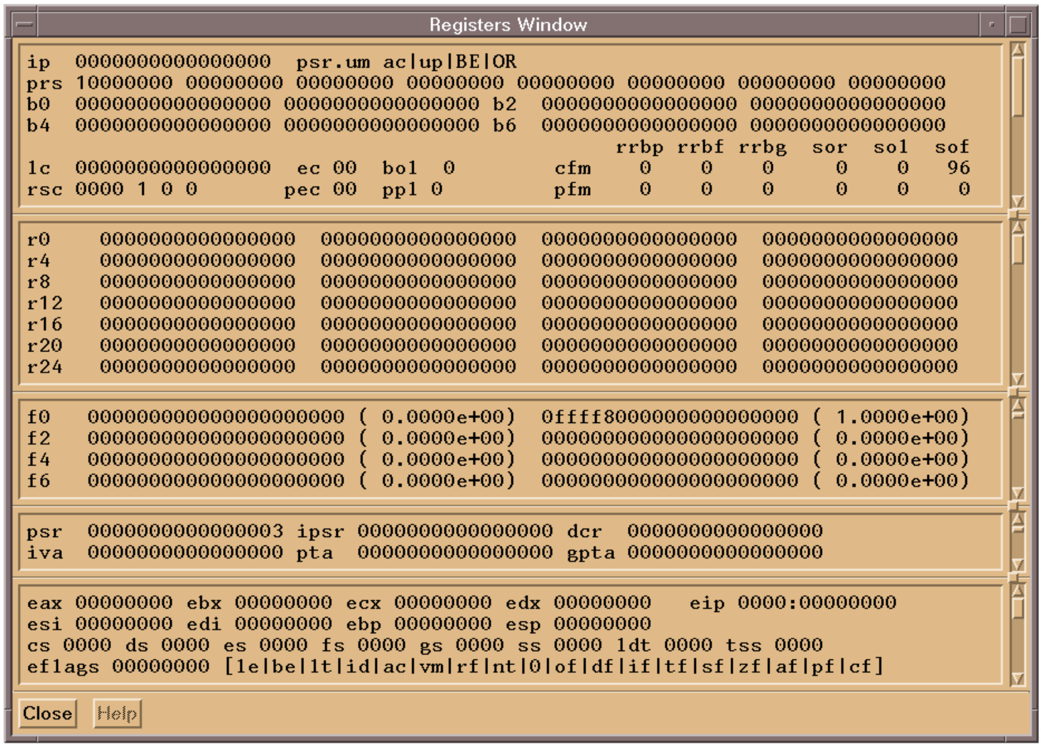
ski uses the curses package to create four windows on the terminal screen. Because ***ski*** uses curses, it runs on nearly any terminal or terminal emulator, including xterm. When ***ski*** needs to show data that isn't appropriate for one of its four windows, it uses a pager such as "**`more`**" or "**`less`**" instead and restores the curses windows when the pager completes.

Ski uses three of the windows to display information to you. The fourth window is shared between you and Ski: You enter commands that control Ski and Ski reports errors and other immediate information to you. You control the windows using Ski commands (see "Screen Manipulation Commands" on page "59") and the simulator updates the windows whenever necessary to maintain consistency with the internal state of the simulator engine. The four windows are described in more detail below.

The Register Window

Ski divides the IA-64 processor registers into five sets. In ***xski***, all five sets are displayed in one window, the Register Window, with each set in its own subwindow or "pane". The panes show user registers, general registers, floating point registers, system registers, and IA-32 registers respectively, as shown in Illustration 27:The Register Window in xski The five panes share screen space and, unless you have a very large screen, it's not possible to see all five panes at full size simultaneously. ***xski*** shows portions of all five panes by default, but you can toggle any panes off with commands described in "Screen Manipulation Commands" on page 59).

xski understands the Page Up and Page Down keys and the up-arrow and down-arrow keys found on most keyboards. These keys operate on the current pane, which is usually highlighted with a bright border. When the Register Window has the X Window System focus, the Page Up and Page Down keys scroll the current pane one "pane-full" less one line of overlap. The up-arrow and down-arrow keys scroll the current pane one line. The Tab and Shift+tab keys change the current pane highlight to the next or previous pane, respectively, "wrapping around" the top and bottom of the Register Window.



```
ip  0000000000000000  psr.um ac|up|BE|OR
prs 10000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
b0  0000000000000000 0000000000000000 b2  0000000000000000 0000000000000000
b4  0000000000000000 0000000000000000 b6  0000000000000000 0000000000000000
                                                rrbp rrbf rrbg  sor  sol  sof
lc  0000000000000000  ec 00  bol  0        cfm    0    0    0     0    0   96
rsc 0000 1 0 0       pec 00  ppl 0         pfm    0    0    0     0    0    0

r0    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r4    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r8    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r12   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r16   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r20   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r24   0000000000000000  0000000000000000  0000000000000000  0000000000000000

f0   000000000000000000000 ( 0.0000e+00)  0ffff8000000000000000 ( 1.0000e+00)
f2   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f4   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)
f6   000000000000000000000 ( 0.0000e+00)  000000000000000000000 ( 0.0000e+00)

psr  0000000000000003 ipsr 0000000000000000 dcr  0000000000000000
iva  0000000000000000 pta  0000000000000000 gpta 0000000000000000

eax 00000000 ebx 00000000 ecx 00000000 edx 00000000   eip 0000:00000000
esi 00000000 edi 00000000 ebp 00000000 esp 00000000
cs 0000 ds 0000 es 0000 fs 0000 gs 0000 ss 0000 ldt 0000 tss 0000
eflags 00000000 [le|be|lt|id|ac|vm|rf|nt|0|of|df|if|tf|sf|zf|af|pf|cf]
```

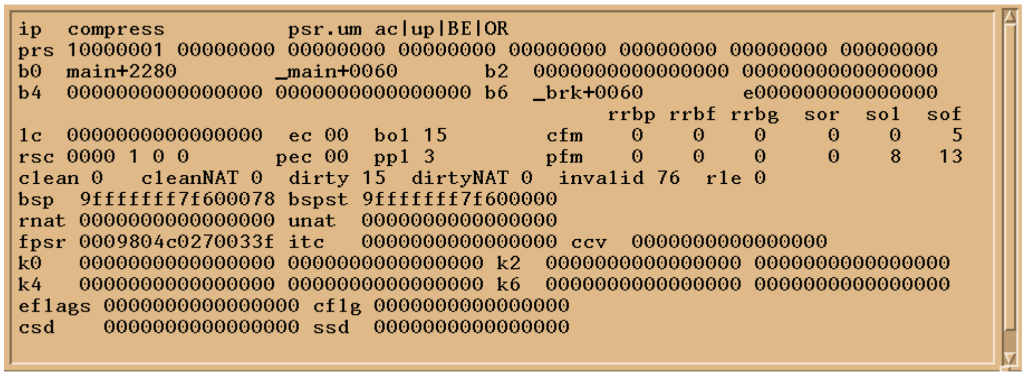
Illustration 27:The Register Window in xski

ski shows only a portion of a register set at a time and you use the commands described in "Register Window Commands" on page 59 to select which portion of which set to see. The sets are described below in the order they appear in the Register Window. Their ***xski*** realizations are shown as well.

The User Registers Pane

The user registers pane (see Illustration 28: The xski User Registers Pane) displays the Predicate Registers (**`prs`**) in binary, the Application Registers in hexadecimal, and the Branch Registers (**`b0`**-**`b7`**) and the Instruction Pointer (**`ip`**) symbolically if possible, otherwise in hexadecimal. Symbolic displays are limited to sixteen characters; when more than sixteen characters are needed, the first fifteen are displayed and an asterisk ("**`*`**") is added to indicate that the symbolic display has been abbreviated. The fields of the Current Frame Marker (**`cfm`**) register and subfields of the Previous Frame Marker field (**`pfm`**) are displayed in decimal. For bit-encoded registers, some bits are displayed individually using their IA-64 mnemonics. If a bit name is displayed in uppercase, the bit is currently set, and if the name is displayed in lowercase, the bit is currently clear. For example, the *psr.be* bit is shown as "**`BE`**" in Illustration 28: The xski User Registers Pane, indicating that the bit is set. The User Mask bitfield (*psr.um*) from the Processor Status Register (**`psr`**) is displayed in this pane; the entire **`psr`** is shown in the System Registers pane, described in "The System Registers Pane". Predicate Registers **`pr16`**-**`pr63`** are displayed in their rotated form, as indicated by the **`rrbp`** field of the Current Frame Marker (**`cfm`**) register.

At the middle of the pane, the line starting "**`clean`**" shows, in decimal, the values in the internal registers that control the Register Save Engine (**`rse`**). The IA-64 architecture requires that these registers exist but provides no program-visible access to them.

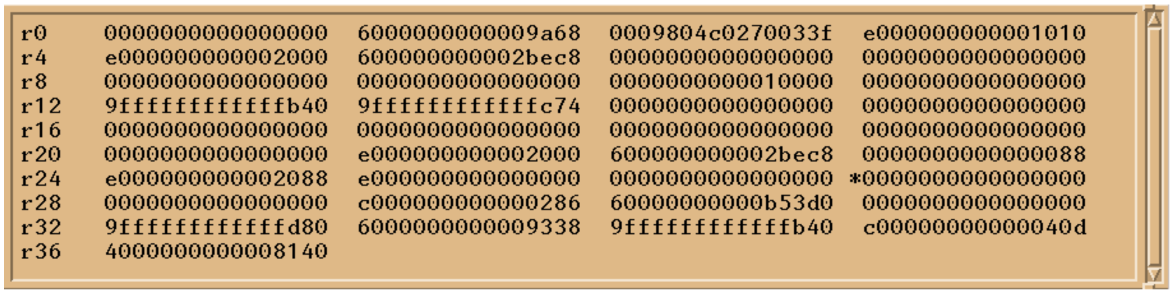


```
ip   compress           psr.um ac|up|BE|OR
prs 10000001 00000000 00000000 00000000 00000000 00000000 00000000 00000000
b0  main+2280        _main+0060       b2  0000000000000000 0000000000000000
b4  0000000000000000 0000000000000000 b6  _brk+0060        e000000000000000
                                                 rrbp rrbf rrbg  sor  sol  sof
lc  0000000000000000  ec 00  bol 15        cfm    0    0    0     0    0    5
rsc 0000 1 0 0       pec 00  ppl 3         pfm    0    0    0     0    8   13
clean 0   cleanNAT 0  dirty 15  dirtyNAT 0  invalid 76  rle 0
bsp  9fffffff7f600078 bspst 9fffffff7f600000
rnat 0000000000000000 unat  0000000000000000
fpsr 0009804c0270033f itc   0000000000000000 ccv  0000000000000000
k0   0000000000000000 0000000000000000 k2  0000000000000000 0000000000000000
k4   0000000000000000 0000000000000000 k6  0000000000000000 0000000000000000
eflags 0000000000000000 cflg 0000000000000000
csd    0000000000000000 ssd  0000000000000000
```

Illustration 28: The xski User Registers Pane

The General Registers Pane

The general registers pane shows the current values of the 64-bit general (integer) data registers, four to a line, in hexadecimal. Registers whose corresponding NaT bits are set are displayed with a leading asterisk ("`*`") to indicate this. The display reflects IA-64 register stacking and rotation: only the 32 static registers and the stacked registers allocated to a function are displayed. The allocated rotating registers are displayed in their rotated form, as indicated by the **`rrbg`** field of the **`cfm`** register, displayed in the user registers pane. The general registers pane is shown in Illustration 29: The xski General Registers Pane



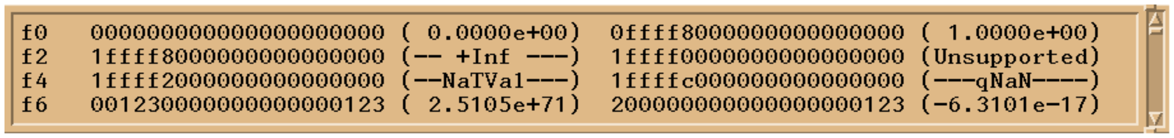
```
r0    0000000000000000  6000000000009a68  0009804c0270033f  e000000000001010
r4    e000000000002000  600000000002bec8  0000000000000000  0000000000000000
r8    0000000000000000  0000000000000000  0000000000010000  0000000000000000
r12   9ffffffffffffb40  9ffffffffffffc74  0000000000000000  0000000000000000
r16   0000000000000000  0000000000000000  0000000000000000  0000000000000000
r20   0000000000000000  e000000000002000  600000000002bec8  0000000000000088
r24   e000000000002088  e000000000000000  0000000000000000 *0000000000000000
r28   0000000000000000  c000000000000286  60000000000b53d0  0000000000000000
r32   9ffffffffffffd80  6000000000009338  9ffffffffffffb40  c00000000000040d
r36   4000000000008140
```

Illustration 29: The xski General Registers Pane

The Floating Point Registers Pane

The floating point registers pane shows the current values of the 82-bit floating point data registers, two to a line displayed in hex and scientific decimal notation. Floating point registers **`f32-f127`** are displayed in their rotated form, as indicated by the **`rrbf`** field of the **`cfm`** register, displayed in the user registers pane. The floating point registers pane is shown in Illustration 30: The xski Floating Point Registers Pane with various values in the registers.

Due to the nature of floating point arithmetic on the host computer, the scientific decimal displays may be inaccurate for very large and very small numbers, positive and negative. The hexadecimal display is always correct, as are all calculations done by the simulated program.

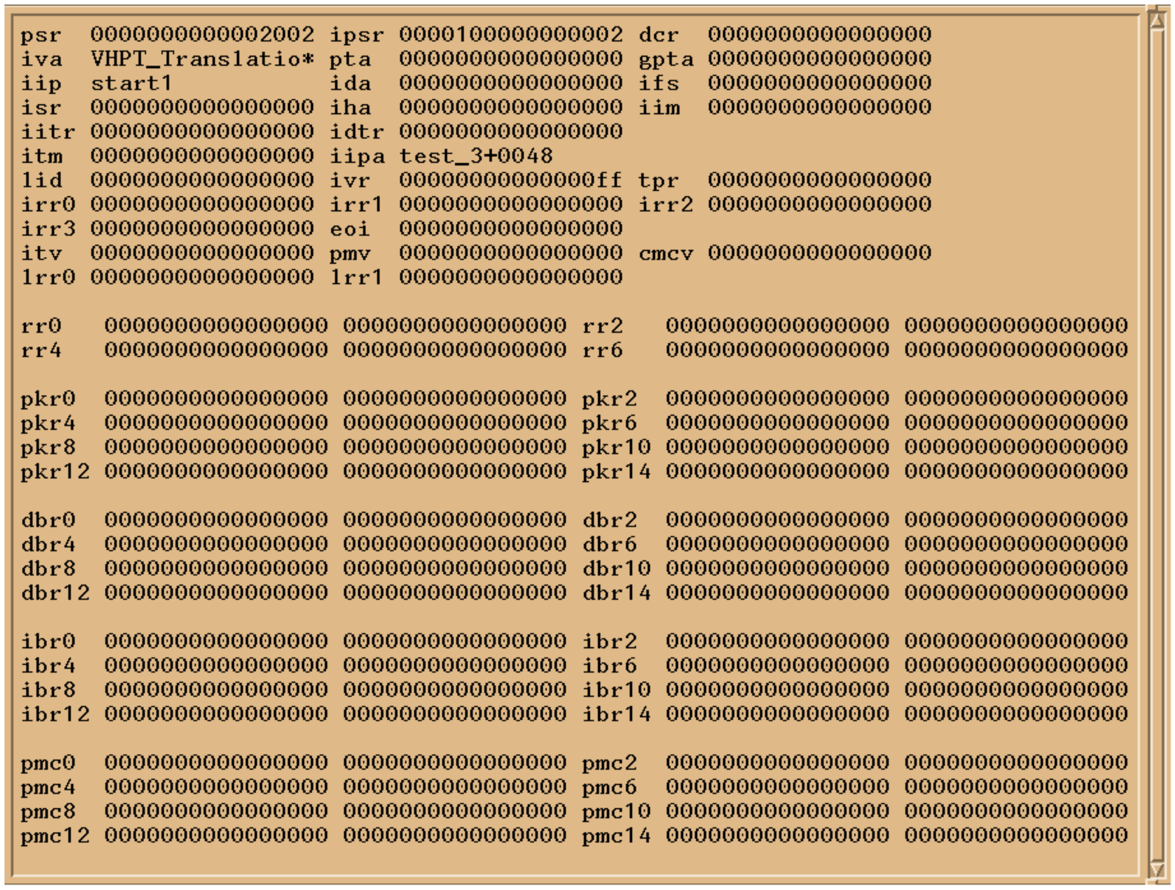


```
f0   000000000000000000000 ( 0.0000e+00)  0ffff8000000000000000 ( 1.0000e+00)
f2   1ffff8000000000000000 (-- +Inf ---)  1ffff0000000000000000 (Unsupported)
f4   1ffff2000000000000000 (--NaTVal---)  1ffffc000000000000000 (---qNaN----)
f6   001230000000000000123 ( 2.5105e+71)  200000000000000000123 (-6.3101e-17)
```

Illustration 30: The xski Floating Point Registers Pane

The System Registers Pane

The system registers pane shows the Processor Status Register (**`psr`**), Control Registers, Region Registers (**`rr0`**-**`rr7`**), Protection Key Registers (**`pkr0`**-**`pkr15`**), Data Breakpoint Registers (**`dbr0`**-**`dbr15`**), Instruction Breakpoint Registers (**`ibr0`**-**`ibr15`**), and Performance Monitor Configuration Registers (**`pmc0`**-**`pmc15`**), in hexadecimal. Application programs have limited access to these registers. Addresses are displayed symbolically when possible. Symbolic displays are limited to sixteen characters; when more than sixteen characters are needed, the first fifteen are displayed and an asterisk ("**`*`**") is added to indicate that the symbolic display has been abbreviated. The **`iva`** register shown on the second text line in Illustration 31: The xski System Registers Pane is an example of this.

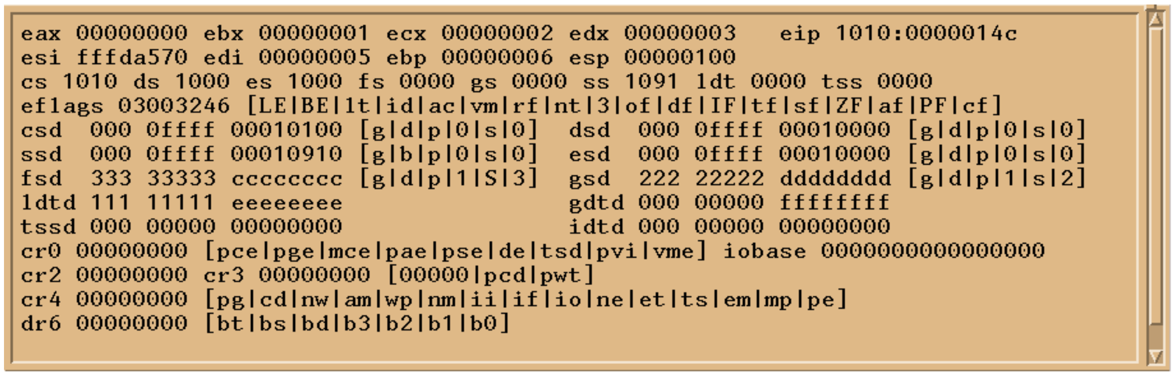


```
psr  0000000000002002 ipsr 0000100000000002 dcr  0000000000000000
iva  VHPT_Translatio* pta  0000000000000000 gpta 0000000000000000
iip  start1           ida  0000000000000000 ifs  0000000000000000
isr  0000000000000000 iha  0000000000000000 iim  0000000000000000
iitr 0000000000000000 idtr 0000000000000000
itm  0000000000000000 iipa test_3+0048
lid  0000000000000000 ivr  00000000000000ff tpr  0000000000000000
irr0 0000000000000000 irr1 0000000000000000 irr2 0000000000000000
irr3 0000000000000000 eoi  0000000000000000
itv  0000000000000000 pmv  0000000000000000 cmcv 0000000000000000
lrr0 0000000000000000 lrr1 0000000000000000

rr0   0000000000000000 0000000000000000 rr2   0000000000000000 0000000000000000
rr4   0000000000000000 0000000000000000 rr6   0000000000000000 0000000000000000

pkr0  0000000000000000 0000000000000000 pkr2  0000000000000000 0000000000000000
pkr4  0000000000000000 0000000000000000 pkr6  0000000000000000 0000000000000000
pkr8  0000000000000000 0000000000000000 pkr10 0000000000000000 0000000000000000
pkr12 0000000000000000 0000000000000000 pkr14 0000000000000000 0000000000000000

dbr0  0000000000000000 0000000000000000 dbr2  0000000000000000 0000000000000000
dbr4  0000000000000000 0000000000000000 dbr6  0000000000000000 0000000000000000
dbr8  0000000000000000 0000000000000000 dbr10 0000000000000000 0000000000000000
dbr12 0000000000000000 0000000000000000 dbr14 0000000000000000 0000000000000000

ibr0  0000000000000000 0000000000000000 ibr2  0000000000000000 0000000000000000
ibr4  0000000000000000 0000000000000000 ibr6  0000000000000000 0000000000000000
ibr8  0000000000000000 0000000000000000 ibr10 0000000000000000 0000000000000000
ibr12 0000000000000000 0000000000000000 ibr14 0000000000000000 0000000000000000

pmc0  0000000000000000 0000000000000000 pmc2  0000000000000000 0000000000000000
pmc4  0000000000000000 0000000000000000 pmc6  0000000000000000 0000000000000000
pmc8  0000000000000000 0000000000000000 pmc10 0000000000000000 0000000000000000
pmc12 0000000000000000 0000000000000000 pmc14 0000000000000000 0000000000000000
```

Illustration 31: The xski System Registers Pane

The IA-32 Registers Pane

The IA-32 registers pane shows IA-32 registers in hexadecimal. For bit-encoded registers, the bits are named individually using their IA-32 mnemonics. If a name is displayed in uppercase, the corresponding bit is currently set, and if the name is displayed in lowercase, the bit is currently clear, as shown in Illustration 32: The xski IA-32 Registers Pane.

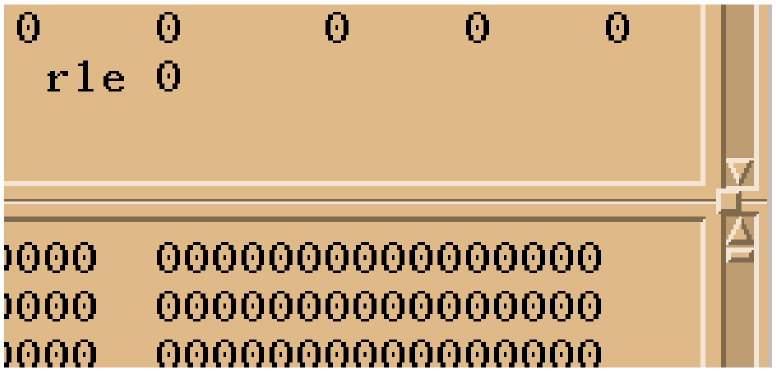


```
eax 00000000 ebx 00000001 ecx 00000002 edx 00000003   eip 1010:0000014c
esi fffda570 edi 00000005 ebp 00000006 esp 00000100
cs 1010 ds 1000 es 1000 fs 0000 gs 0000 ss 1091 ldt 0000 tss 0000
eflags 03003246 [LE|BE|1t|id|ac|vm|rf|nt|3|of|df|IF|tf|sf|ZF|af|PF|cf]
csd  000 0ffff 00010100 [g|d|p|0|s|0]  dsd  000 0ffff 00010000 [g|d|p|0|s|0]
ssd  000 0ffff 00010910 [g|b|p|0|s|0]  esd  000 0ffff 00010000 [g|d|p|0|s|0]
fsd  333 33333 cccccccc [g|d|p|1|S|3]  gsd  222 22222 dddddddd [g|d|p|1|s|2]
ldtd 111 11111 eeeeeeee                gdtd 000 00000 ffffffff
tssd 000 00000 00000000                idtd 000 00000 00000000
cr0 00000000 [pce|pge|mce|pae|pse|de|tsd|pvi|vme] iobase 0000000000000000
cr2 00000000 cr3 00000000 [00000|pcd|pwt]
cr4 00000000 [pg|cd|nw|am|wp|nm|ii|if|io|ne|et|ts|em|mp|pe]
dr6 00000000 [bt|bs|bd|b3|b2|b1|b0]
```

Illustration 32: The xski IA-32 Registers Pane

Resizing Register Window Panes with xski

As mentioned above, even a large X Window System screen is too small to display all the registers simultaneously, so you may have to scroll a pane to see the registers you want, or resize the pane by dragging Pane Resizer, the small resize square on the right side of the dividing line between each pair of panes, as shown in Illustration 33: An xski Pane Resizer: The Small Box Between the Scroll bars.



```
0     0      0     0     0
 rle 0

000   0000000000000000
000   0000000000000000
000   0000000000000000
```

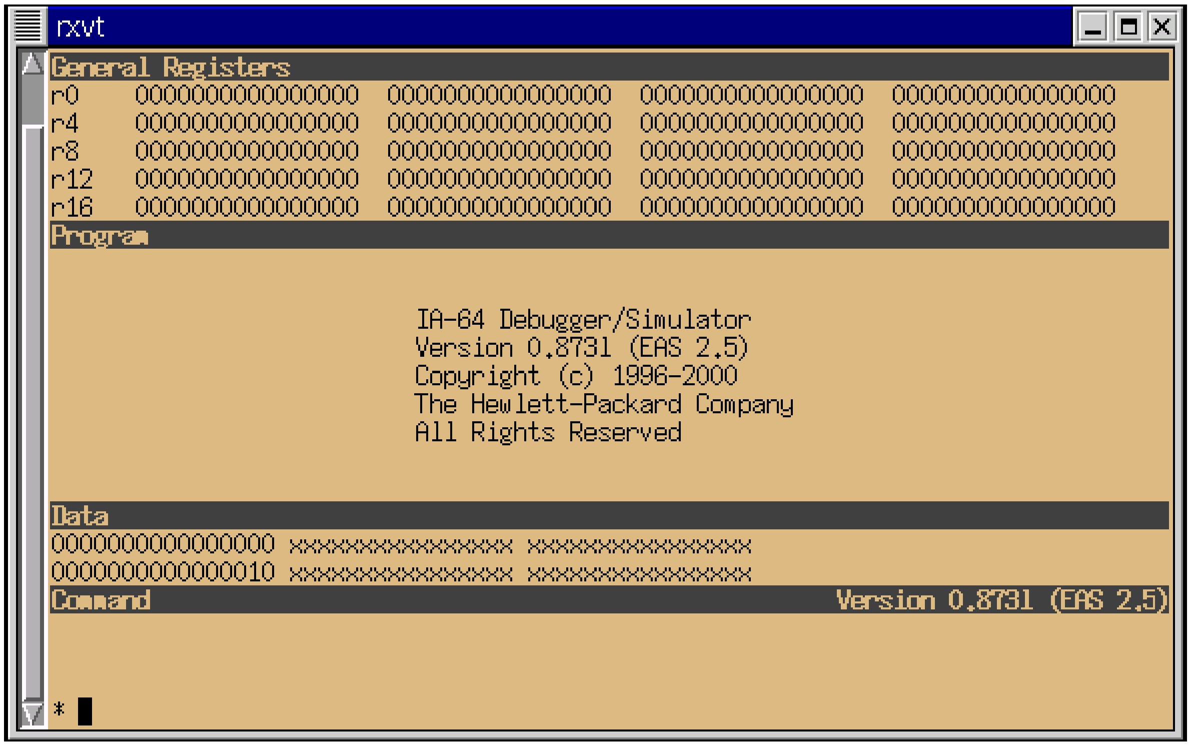
Illustration 33: An xski Pane Resizer: The Small Box Between the Scroll bars

The Register Window and ski

The ***ski*** simulator, as noted above, uses curses to display multiple windows on non-graphic (text) terminals and terminal emulators. These windows are fixed in size and are not big enough to display all the data at the same time. On a conventional, twenty four line screen, ***ski*** uses five lines for the Register Window, as shown in Illustration 34: The ski

. Because of this lack of space, the Register Window shows only one of the five sets of registers at a time: user, integer, floating point, system, or IA-32, and then only a portion of each set. If your screen is larger than twenty four lines when you start ***ski***, ***ski*** will make use of the extra space. (You can resize terminal emulators using command-line arguments or by using your window manager's standard mechanisms for window resizing.)

You use the **`ur`**, **`gr`**, **`fr`**, **`sr`**, and **`iar`** commands to tell ***ski*** which set of registers to display. To see the various registers in a set, you use the **`rf`** and **`rb`** commands to scroll the Register Window forwards and backwards, respectively. These commands are described in "Summary of Register Window Commands" on page 59.



```
rxvt
General Registers
r0      0000000000000000  0000000000000000  0000000000000000  0000000000000000
r4      0000000000000000  0000000000000000  0000000000000000  0000000000000000
r8      0000000000000000  0000000000000000  0000000000000000  0000000000000000
r12     0000000000000000  0000000000000000  0000000000000000  0000000000000000
r16     0000000000000000  0000000000000000  0000000000000000  0000000000000000
Program


                           IA-64 Debugger/Simulator
                           Version 0.8731 (EAS 2.5)
                           Copyright (c) 1996-2000
                           The Hewlett-Packard Company
                           All Rights Reserved


Data
0000000000000000 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
0000000000000010 xxxxxxxxxxxxxxxx xxxxxxxxxxxxxxxx
Command                                                   Version 0.8731 (EAS 2.5)


* 
```

Illustration 34: The ski Register Window (at Top)

The Program Window

The Program Window provides a view into the program space. Whether you load a program into the simulated processor's address space via the command line or using Ski's **`load`**, **`iaload`**, or **`romload`** commands, the program is displayed in a format resembling a compiler's assembler listing file. For IA-64 programs compiled from a high-level language such as C' and linked with the appropriate options, the source code is displayed with line numbers, mixed in with the generated assembly language as shown in Illustration 35: xski's Program Window Showing Part of an IA-64 "hello world" Program. As an example, to compile the "hello world" program with the IA-64 compiler used in testing Ski, the command line is:

```
cc -o hello -g hello.c
```

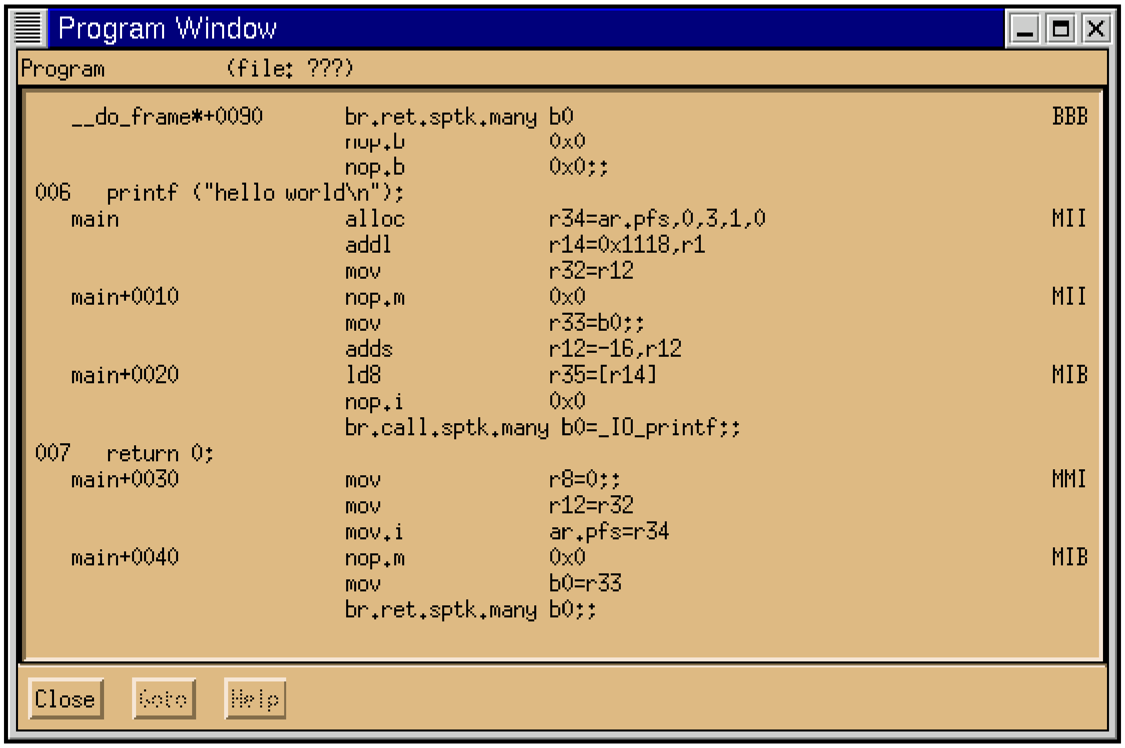
Note that the **`-O`** (capital-O) "optimization" flag was not specified. Optimization, by definition, rearranges the object code. If you turn on optimization, the correspondence between source code and object code will be obscured and you may find the resulting display difficult to interpret.

IA-64 assembly code is displayed through disassembly; the original assembler source code is not displayed. Source code for IA-32 programs, high-level and assembly, is not displayed.

Ski chooses whether to interpret the instructions as IA-64 or IA-32 encodings based on the setting of the *psr.is* bit. If your program has a mix of IA-64 and IA-32 code, you may need to manually set or clear this bit when trying to view a part of the program that is in a different encoding from the encoding at the current **`ip`** location. You can set the bit with the Ski command "**`= psr.is 1`**" and you can clear the bit with "**`= psr.is 0`**". If the bit is set incorrectly, Ski will use the wrong instruction decoder and will show IA-64 code disassembled as if it was IA-32 code or vice-versa! Remember to set the bit back before resuming simulation.

IA-64 Instruction Display

Each IA-64 instruction bundle is labelled on the left with an hexadecimal byte-addressed offset from the nearest, preceding symbol up to 0xffff bytes away. If the symbol name and offset are longer than sixteen characters, the first fifteen are displayed and an asterisk ("**`*`**") is added to indicate that the symbolic display has been abbreviated. For each 128 bit bundle, the two or three instructions are displayed in the center of the window with operands to their immediate right. The template for the bundle is shown as a triplet of capital letters, such as "**`MII`**," to the right of the last operand of the first instruction in the bundle. The end of each instruction group (a unit of potentially parallel execution) is marked with a pair of semicolons ("**`;;`**") after the last operand of the last instruction in the group.

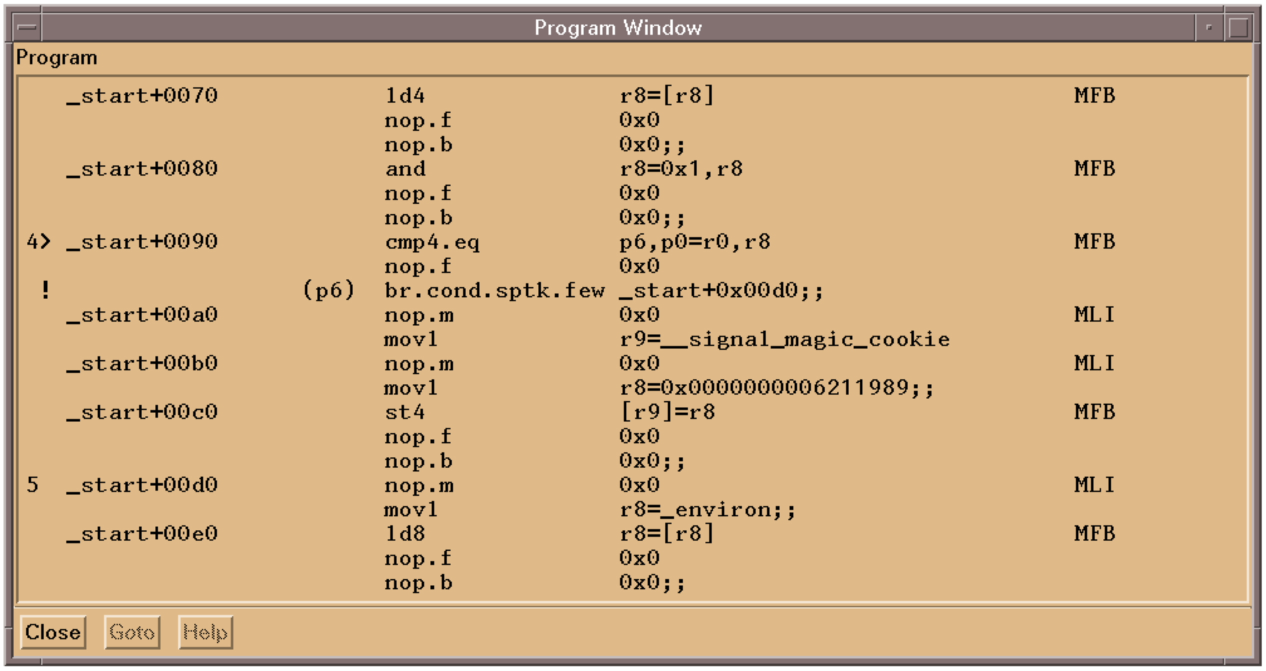


```
Program          (file: ???)

   __do_frame*+0090       br.ret.sptk.many b0                   BBB
                          nop.b            0x0
                          nop.b            0x0;;
006   printf ("hello world\n");
   main                   alloc            r34=ar.pfs,0,3,1,0   MII
                          addl             r14=0x1118,r1
                          mov              r32=r12
   main+0010              nop.m            0x0                  MII
                          mov              r33=b0;;
                          adds             r12=-16,r12
   main+0020              ld8              r35=[r14]            MIB
                          nop.i            0x0
                          br.call.sptk.many b0=_IO_printf;;
007   return 0;
   main+0030              mov              r8=0;;               MMI
                          mov              r12=r32
                          mov.i            ar.pfs=r34
   main+0040              nop.m            0x0                  MIB
                          mov              b0=r33
                          br.ret.sptk.many b0;;
```

Illustration 35: xski's Program Window Showing Part of an IA-64 "hello world" Program

Ski uses the first few columns for source code line numbers. Ski also uses the first column to show breakpoint locations for IA-64 assembly language instructions, numbering the breakpoints "**`0`**" through "**`9`**." IA-64 breakpoint commands include **`bs`**, **`bD`**, **`bd`**, and **`bl`**, and are described in "Program Breakpoints" on page 78. For the purpose of setting breakpoint addresses, Ski "pretends" that the slot 0 instruction in a bundle is located at the first byte of the bundle, the slot 1 instruction is located at the fourth byte, and the slot 2 instruction is located at the eighth byte. See "How Ski Implements Breakpoints" on page 80.

Predication is an IA-64 feature that increases the usable parallelism of user programs and allows better utilization of functional units. Ski shows predication information in the second column of the Program Window, as shown in Illustration 36: xski's Program Window Showing IA-64 Predication and Breakpoints. If the second column of a given instruction line contains an exclamation mark ("**!**"), the instruction is predicated on a predicate register that is currently 0: the instruction is "predicated off". The predicate register is displayed in parenthesis immediately to the left of the instruction mnemonic. Ski uses a different encoding for the instruction pointed to by the **ip** register: an asterisk ("*****") indicates that the instruction is predicated off and a greater-than symbol ("**>**") indicates that the instruction is predicated on. (That is, the "**>**" symbol means "This is the next instruction to be simulated.")



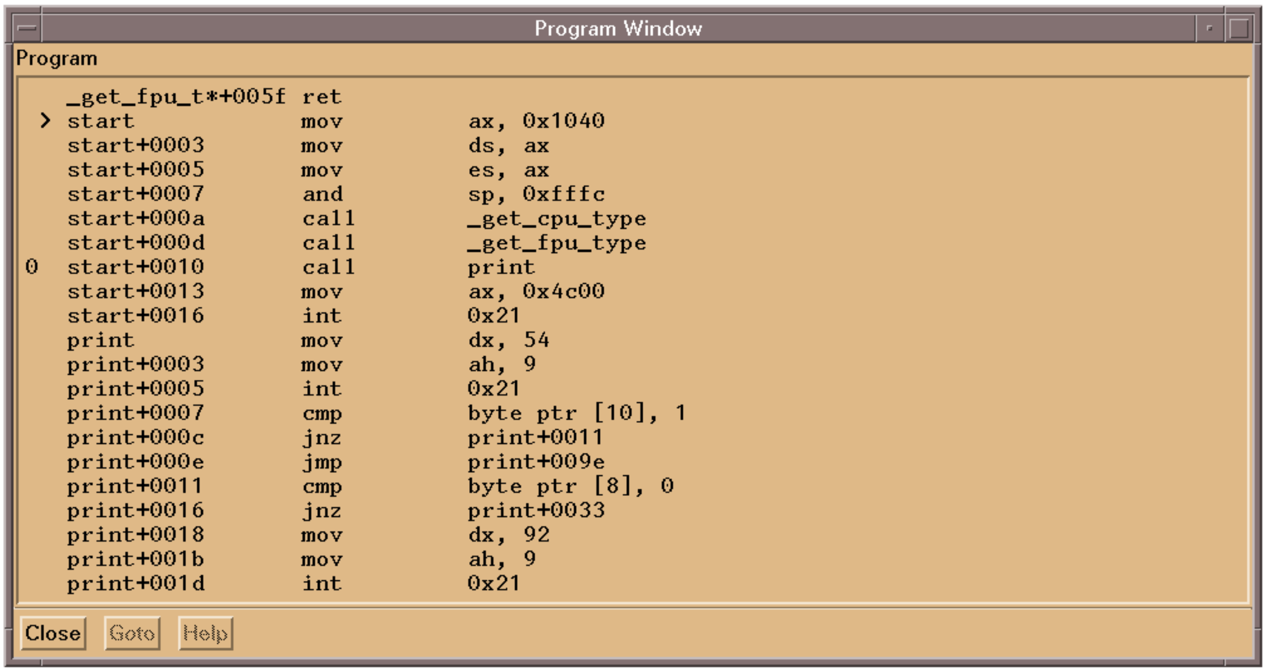
```
   _start+0070          ld4                r8=[r8]                        MFB
                        nop.f              0x0
                        nop.b              0x0;;
   _start+0080          and                r8=0x1,r8                      MFB
                        nop.f              0x0
                        nop.b              0x0;;
4> _start+0090          cmp4.eq            p6,p0=r0,r8                    MFB
                        nop.f              0x0
 !                (p6)  br.cond.sptk.few   _start+0x00d0;;
   _start+00a0          nop.m              0x0                            MLI
                        movl               r9=__signal_magic_cookie
   _start+00b0          nop.m              0x0                            MLI
                        movl               r8=0x0000000006211989;;
   _start+00c0          st4                [r9]=r8                        MFB
                        nop.f              0x0
                        nop.b              0x0;;
5  _start+00d0          nop.m              0x0                            MLI
                        movl               r8=_environ;;
   _start+00e0          ld8                r8=[r8]                        MFB
                        nop.f              0x0
                        nop.b              0x0;;
```

Illustration 36: xski's Program Window Showing IA-64 Predication and Breakpoints

IA-32 Instruction Display

IA-32 instructions are displayed as shown in Illustration 37: xski's Program Window Showing IA-32 Code, the Instruction Pointer, and a Breakpoint, according to the conventions for Intel assembly code. As with IA-64 instruction display, Ski uses the first column of each assembly language instruction line to show breakpoint locations, numbering them "**0**" through "**9**." Except for the use of **iabs** rather than **bs**, IA-32 breakpoint commands are the same as IA-64 breakpoint commands and include **iabs**, **bD**, **bd**, and **bl**,as described in "Program Breakpoints" on page 78. In the second column, Ski puts a greater-than symbol ("**>**") to point to the next instruction to be executed, i.e., the location pointed to by the **ip** register.

Because IA-32 instructions are variable in length, it is possible to set the **ip** to point into the middle of an instruction. This can happen, for example, when an instruction with prefix bytes is needed at the top of the first pass through a loop, and the same instruction without the prefix bytes is needed at the top of subsequent passes. When this happens, Ski uses a plus-sign ("**+**") in column two, rather than a greater-than symbol, to warn you that **ip** points somewhere in the middle of the line of code displayed on the screen. To update the display, use the command "**pj ip**". This will cause Ski to reinterpret the instruction stream and to display the variable length instructions with the new interpretation.



```
    _get_fpu_t*+005f ret
  > start            mov        ax, 0x1040
    start+0003       mov        ds, ax
    start+0005       mov        es, ax
    start+0007       and        sp, 0xfffc
    start+000a       call       _get_cpu_type
    start+000d       call       _get_fpu_type
0   start+0010       call       print
    start+0013       mov        ax, 0x4c00
    start+0016       int        0x21
    print            mov        dx, 54
    print+0003       mov        ah, 9
    print+0005       int        0x21
    print+0007       cmp        byte ptr [10], 1
    print+000c       jnz        print+0011
    print+000e       jmp        print+009e
    print+0011       cmp        byte ptr [8], 0
    print+0016       jnz        print+0033
    print+0018       mov        dx, 92
    print+001b       mov        ah, 9
    print+001d       int        0x21
```

Illustration 37: xski's Program Window Showing IA-32 Code, the Instruction Pointer, and a Breakpoint

Changing the Range of Locations Shown in the Program Window

xski doesn't place a scroll bar in the Program Window. Instead, like ***ski***, ***xski*** provides the **`pf`** and **`pb`** commands, described in "Program Window Commands" on page 60. You use these commands to scroll the Program Window forwards and backwards, respectively, through the assembly language program display. Ski also provides the **`pj`** command which lets you "jump" the Program Window to any location in the address space. In addition, ***xski*** understands the Page Up and Page Down keys and the arrow keys. When the Program Window has the X Window System focus, the Page Up, Page Down, up-arrow, and down-arrow keys emit the "**`pb`**", "**`pf`**", "**`pb 1`**", and "**`pf 1`**" commands, respectively.

You can control the size of ***xski***'s Program Window using your window manager's standard mechanisms. If you are using ***ski***, the window is fixed in size; on a twenty four line terminal, the window will be nine lines tall.

Invalid Code and the Program Window

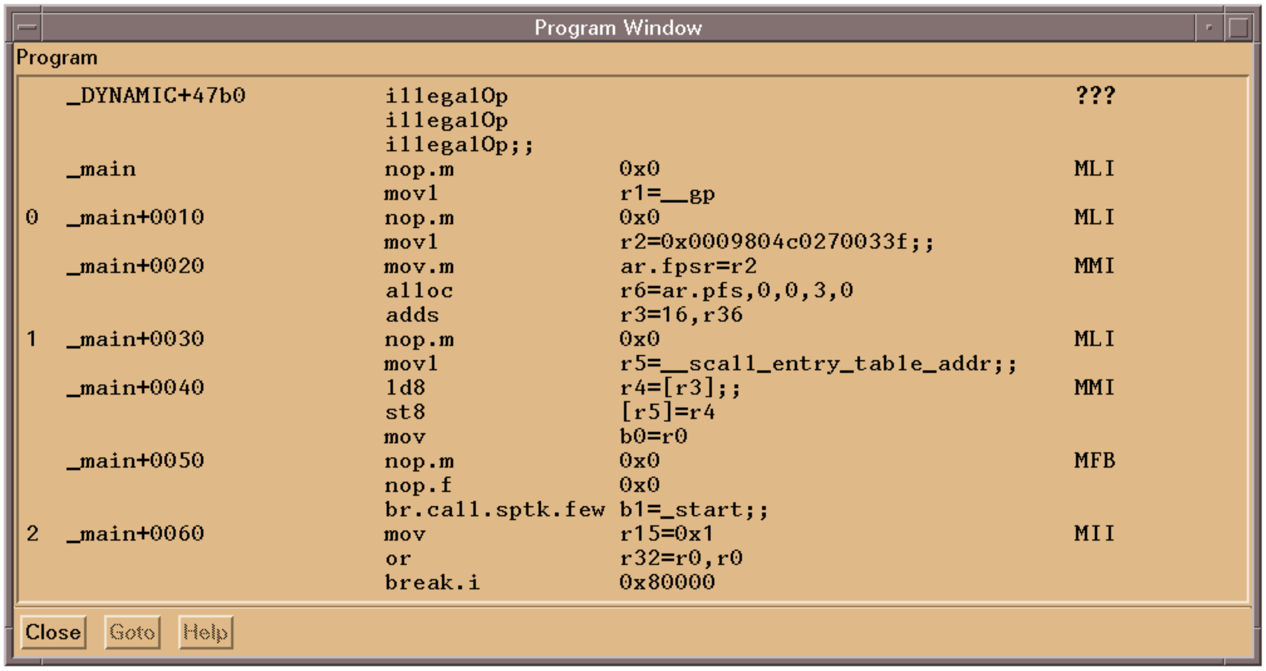
Ski will disassemble the area of memory it is displaying in the program window, regardless of whether the area contains program code or data. If you tell Ski to display non-program memory, Ski attempts to display the (non-existent) instructions. When Ski finds bit encodings that don't represent valid instructions, it displays the word "**`illegalOp`**" instead, as shown in Illustration 38: xski's Program Window Showing Illegal Instructions. Sometimes, Ski may display **`x`**'s, indicating that you asked Ski to show a page of memory that doesn't exist, as shown in Illustration 39: xski's Program Window Showing Unallocated Space or No Translation. There are three cases to consider:

- In application-mode, **`x`**'s indicate a page of memory that hasn't been accessed by the program and therefore hasn't been allocated by Ski.
- In system-mode with instruction address translation enabled (the *psr.it* bit is on), **`x`**'s indicate a page of memory for which no entry exists in the Translation Lookaside Buffer (TLB) or in the Virtual Hash Page Table

(VHPT).

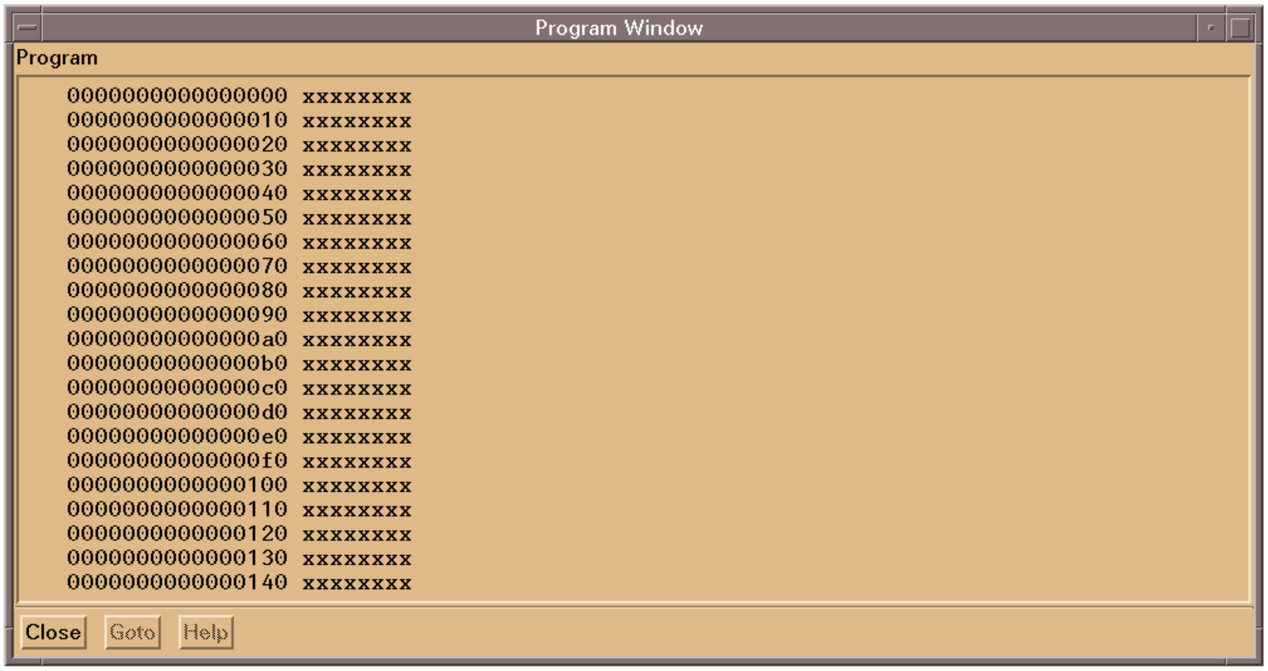
- In system-mode with instruction address translation disabled (the *psr.it* bit is off), **x**'s indicate a page of memory that has not yet been accessed by the program.

Application-mode and system-mode programming are discussed in more detail in Program Simulation on page 66.



```
   _DYNAMIC+47b0     illegalOp                                       ???
                     illegalOp
                     illegalOp;;
   _main             nop.m            0x0                            MLI
                     movl             r1=__gp
0  _main+0010        nop.m            0x0                            MLI
                     movl             r2=0x0009804c0270033f;;
   _main+0020        mov.m            ar.fpsr=r2                     MMI
                     alloc            r6=ar.pfs,0,0,3,0
                     adds             r3=16,r36
1  _main+0030        nop.m            0x0                            MLI
                     movl             r5=__scall_entry_table_addr;;
   _main+0040        ld8              r4=[r3];;                      MMI
                     st8              [r5]=r4
                     mov              b0=r0
   _main+0050        nop.m            0x0                            MFB
                     nop.f            0x0
                     br.call.sptk.few b1=_start;;
2  _main+0060        mov              r15=0x1                        MII
                     or               r32=r0,r0
                     break.i          0x80000
```

Illustration 38: xski's Program Window Showing Illegal Instructions



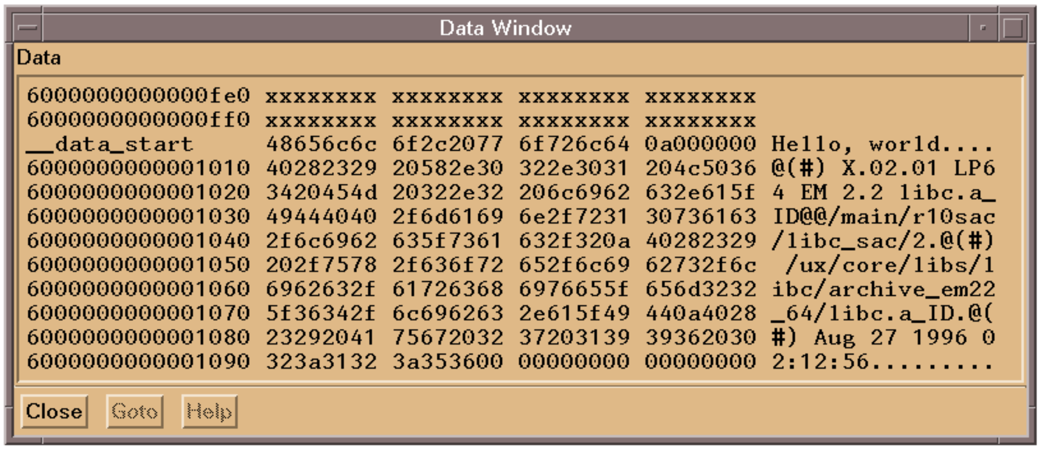
```
0000000000000000 xxxxxxxx
0000000000000010 xxxxxxxx
0000000000000020 xxxxxxxx
0000000000000030 xxxxxxxx
0000000000000040 xxxxxxxx
0000000000000050 xxxxxxxx
0000000000000060 xxxxxxxx
0000000000000070 xxxxxxxx
0000000000000080 xxxxxxxx
0000000000000090 xxxxxxxx
00000000000000a0 xxxxxxxx
00000000000000b0 xxxxxxxx
00000000000000c0 xxxxxxxx
00000000000000d0 xxxxxxxx
00000000000000e0 xxxxxxxx
00000000000000f0 xxxxxxxx
0000000000000100 xxxxxxxx
0000000000000110 xxxxxxxx
0000000000000120 xxxxxxxx
0000000000000130 xxxxxxxx
0000000000000140 xxxxxxxx
```

Illustration 39: xski's Program Window Showing Unallocated Space or No Translation

The Data Window

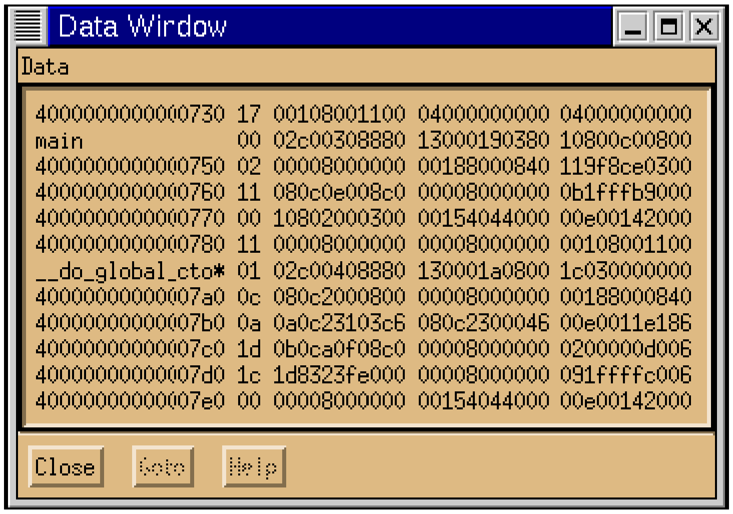
In the Data Window, ***xski*** and ***ski*** present data in hexadecimal format, sixteen bytes to a line, as shown in Illustration 40: xski's Data Window Showing Unallocated Space Followed by Data. The data are displayed as four groups of eight hexadecimal digits each, with an ASCII character translation on the right and the data address on the left. (The endianness of the displayed bytes is determined by the current value of the *psr.be* bit which may change by the time the simulated IA-64 processor actually loads the bytes.) The address is expressed as a symbol from the executable file's symbol table or as a sixteen digit hexadecimal number.

With the **`dbndl`** command, Ski can display data formatted as IA-64 instruction bundles in hexadecimal, as shown in Illustration 41: xski's Data Window Showing Data Interpreted as Instruction Bundles. (The figure was generated by loading a program and then issuing the command "**`dj main-10`**" followed by the **`dbndl`** command.) This is useful when you need to see the raw hexadecimal instruction encodings. The first column displays the address of each bundle. The second column displays the template field. The remaining three columns are the 41-bit instructions from slots 0, 1, and 2. Note: for the purpose of setting breakpoint addresses, Ski "pretends" that the slot 0 instruction is located at the first byte of the bundle, the slot 1 instruction is located at the fourth byte, and the slot 2 instruction is located at the eighth byte. See "How Ski Implements Breakpoints" on page 80 for more information.



```
6000000000000fe0 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
6000000000000ff0 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
__data_start     48656c6c 6f2c2077 6f726c64 0a000000 Hello, world....
6000000000001010 40282329 20582e30 322e3031 204c5036 @(#) X.02.01 LP6
6000000000001020 3420454d 20322e32 206c6962 632e615f 4 EM 2.2 libc.a_
6000000000001030 49444040 2f6d6169 6e2f7231 30736163 ID@@/main/r10sac
6000000000001040 2f6c6962 635f7361 632f320a 40282329 /libc_sac/2.@(#)
6000000000001050 202f7578 2f636f72 652f6c69 62732f6c  /ux/core/libs/l
6000000000001060 6962632f 61726368 6976655f 656d3232 ibc/archive_em22
6000000000001070 5f36342f 6c696263 2e615f49 440a4028 _64/libc.a_ID.@(
6000000000001080 23292041 75672032 37203139 39362030 #) Aug 27 1996 0
6000000000001090 323a3132 3a353600 00000000 00000000 2:12:56.........
```

Illustration 40: xski's Data Window Showing Unallocated Space Followed by Data



```
4000000000000730 17 00108001100 04000000000 04000000000
main             00 02c00308880 13000190380 10800c00800
4000000000000750 02 00008000000 00188000840 119f8ce0300
4000000000000760 11 080c0e008c0 00008000000 0b1fffb9000
4000000000000770 00 10802000300 00154044000 00e00142000
4000000000000780 11 00008000000 00008000000 00108001100
__do_global_cto* 01 02c00408880 130001a0800 1c030000000
40000000000007a0 0c 080c2000800 00008000000 00188000840
40000000000007b0 0a 0a0c23103c6 080c2300046 00e0011e186
40000000000007c0 1d 0b0ca0f08c0 00008000000 0200000d006
40000000000007d0 1c 1d8323fe000 00008000000 091ffffc006
40000000000007e0 00 00008000000 00154044000 00e00142000
```

Illustration 41: xski's Data Window Showing Data Interpreted as Instruction Bundles

Changing the Range of Locations Shown in the Data Window

As with the Program Window, ***xski*** doesn't place a scroll bar in the Data Window. Instead, like ***ski***, ***xski*** provides the **`df`**, **`db`**, and **`dj`** commands, described in "Data Window Commands on page 63. Use these commands to scroll the Data Window forwards and backwards and to "jump" the Data Window. In addition, ***xski*** understands the Page Up and Page Down keys and the arrow keys. When the Data Window has the X Window System focus, the Page Up, Page Down, up-arrow, and down-arrow keys emit the "**`db`**", "**`df`**", "**`db 1`**", and "**`df 1`**" commands, respectively.

You can control the size of ***xski***'s Data Window with your window manager's standard mechanisms. If you are using ***ski***, the window is fixed in size; on a twenty four line terminal, the window will be two lines tall.

Invalid Code and the Data Window

If you tell Ski to display non-existent memory, Ski will display `x`'s instead, as shown in Illustration 40: xski's Data Window Showing Unallocated Space Followed by Data. Non-existent memory is defined for the Data Window similarly to its definition for the Program Window, described in the "Invalid Code and the Program Window" section, except that the relevant bit for system-mode programs is *psr.dt*.

The Command/Main Window

xski and ***ski*** are command-driven simulators. Most of your interaction with them is done by typing commands. Your commands are typed in a window titled "**`main`**" in ***xski*** (see Illustration 42: xski's Main (Command) Window and "**`Command`**" in ***ski*** (see Illustration 43: ski's Command Window (at Bottom)).

The *xski* Main Window

xski divides the Main Window into five areas:

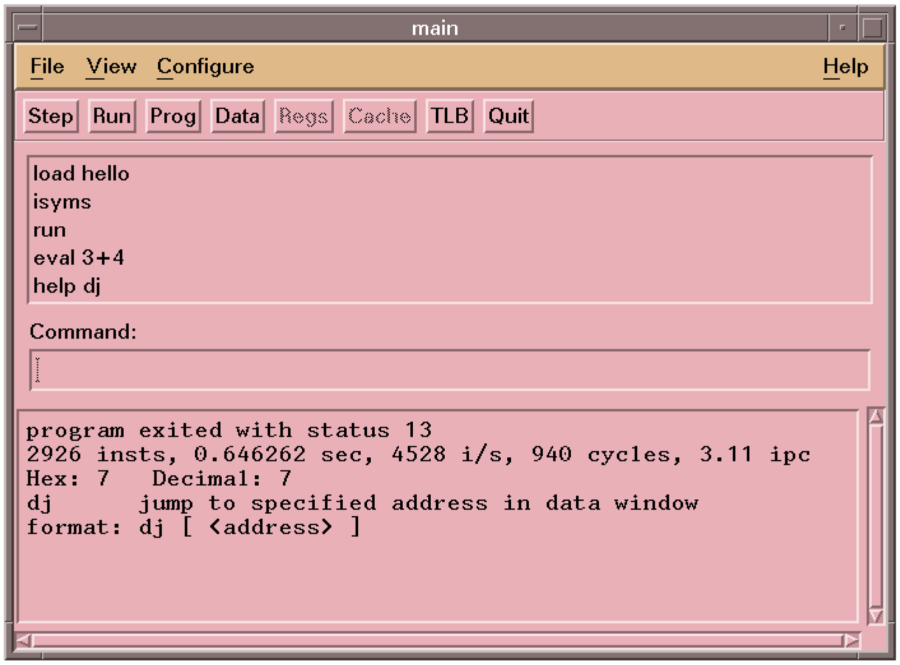
- Menus: File, View, Configure, and Help. The File menu provides a "Quit" selection for you to exit the program. The View menu lets you choose which windows to see. The Configure menu is currently non-functional. The Help menu provides a "Commands" selection that displays the commands Ski recognizes and a "Product Information" selection that displays information about ***xski***.
- Buttons: *`Step`*, *`Run`*, *`Prog`*, *`Data`*, *`Regs`*, *`Cache`*, *`TLB`*, and *`Quit`*. Clicking on the *`Step`* button executes the command "**`step 1`**", single-stepping the simulated program. Shift-clicking the button executes the command "**`step 10`**", stepping the simulated program through ten instructions. The *`Run`*, *`Prog`*, *`Data`*, and *`TLB`* buttons execute the **`run`**, **`pj`**, **`dj`**, and **`sdt`** commands respectively. If the Program Window has been closed (removed from the screen, not merely minimized to an icon), the *`Prog`* button recreates it. The *`Data`* button operates similarly with respect to the Data Window. The *`Regs`* and *`Cache`* buttons are currently non-functional.

  xski's buttons are configurable. Using the X Window System resource mechanism, you can change the number of buttons, the button labels, and the commands the buttons emit. The easiest way to do this is to edit the `XSki` file, described in "The XSki File" on page 34. Much of ***xski***'s user interface behavior is controlled by this file but you should be careful in making changes to any elements other than button descriptions; ***xski*** may change in the future in ways that are not backwards-compatible with changes you make.
- Command History: commands you've already entered.
- Command: where you type commands to ***xski***.
- Responses: responses and error messages from ***xski***.

The Menu, Button, and Command History areas provide shortcuts for typing commands. The *`Step`* button is particularly useful: when you are single-stepping through a program, you can click on the *`Step`* button instead of repeatedly typing the "**`step`**" command. The Command History area provides another way to avoid typing: you can double-click on a command in the Command History to run the command again, or single-click on the command to move it to the Command area where you can edit and then re-run it. The Command area is where you type commands to the simulator, but, as mentioned above, you can use the menus, buttons, and Command History as shortcuts. Two useful commands to know are "**`help`**", which causes a window listing all the commands to be displayed, and "**`help`** *command*" which causes information about the *command* to be shown in the Responses area. The Responses area is also used by the simulator to give you feedback when it can't execute one of your commands.

xski understands the Prev and Next keys and the arrow keys found on many HP keyboards. When the Main Window has the X Window System focus, the current area is highlighted, usually with a bright outline. You can make a different area current with Tab and Shift-tab. The Prev, Next, up-arrow, and down-arrow keys scroll through the current area, allowing you to easily edit and re-run previous commands from the Command History and review previous messages in the Response area.

In addition, you can use the Alternate key ("alt") like a Shift key, along with the underlined letter in each menu name as a shortcut to access the menu, rather than using the mouse. For example, Alt+F brings up the File menu. This lets you spend less time shuttling between the keyboard and mouse, and more time doing productive work.



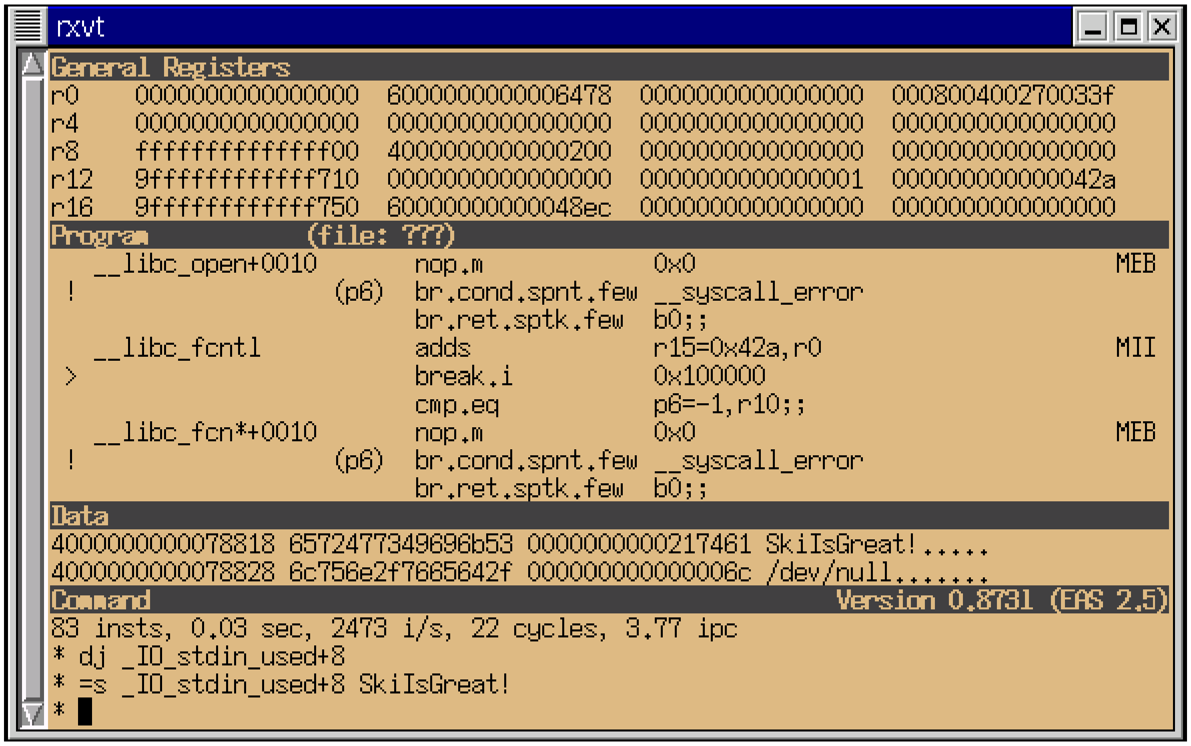
```
load hello
isyms
run
eval 3+4
help dj
```

```
program exited with status 13
2926 insts, 0.646262 sec, 4528 i/s, 940 cycles, 3.11 ipc
Hex: 7   Decimal: 7
dj      jump to specified address in data window
format: dj [ <address> ]
```

Illustration 42: xski's Main (Command) Window

The *ski* Command Window

ski's Command Window is simpler, as shown in Illustration 43: ski's Command Window (at Bottom). There are no menus, buttons, or Command History. Instead, you enter commands when you see a `*` prompt in the 4-line Command Window at the bottom of the screen. ***ski*** displays its responses in this window as well. The window scrolls so that information lost off the top of the window may be recovered using the up and down arrows on your keyboard (for Emacs fans, Ctrl-P and Ctrl-N serve the same function). As a typing shortcut, if you hit the enter/return key, ***ski*** will repeat the last command you entered.



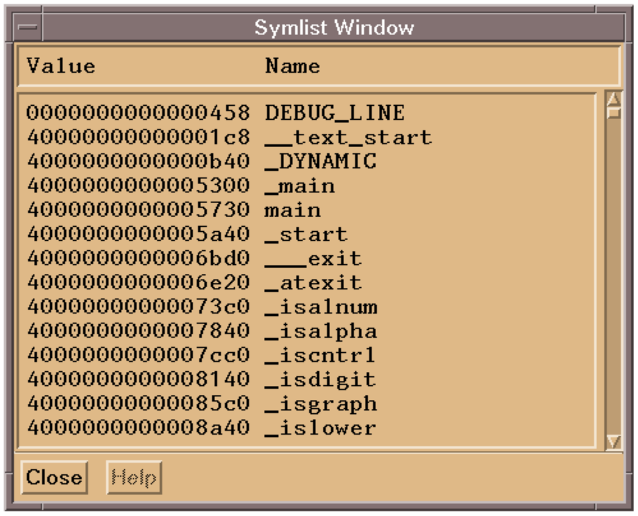
```
General Registers
r0     0000000000000000  6000000000006478  0000000000000000  000800400270033f
r4     0000000000000000  0000000000000000  0000000000000000  0000000000000000
r8     ffffffffffffff00  4000000000000200  0000000000000000  0000000000000000
r12    9ffffffffffff710  0000000000000000  0000000000000001  000000000000042a
r16    9ffffffffffff750  60000000000048ec  0000000000000000  0000000000000000
Program              (file: ???)
   __libc_open+0010       nop.m            0x0                                MEB
 !                  (p6)  br.cond.spnt.few __syscall_error
                          br.ret.sptk.few  b0;;
   __libc_fcntl           adds             r15=0x42a,r0                       MII
 >                        break.i          0x100000
                          cmp.eq           p6=-1,r10;;
   __libc_fcn*+0010       nop.m            0x0                                MEB
 !                  (p6)  br.cond.spnt.few __syscall_error
                          br.ret.sptk.few  b0;;
Data
4000000000078818 6572477349696b53 0000000000217461 SkiIsGreat!.....
4000000000078828 6c756e2f7665642f 000000000000006c /dev/null.......
Command                                              Version 0.8731 (EAS 2.5)
83 insts, 0.03 sec, 2473 i/s, 22 cycles, 3.77 ipc
* dj _IO_stdin_used+8
* =s _IO_stdin_used+8 SkiIsGreat!
*
```

Illustration 43: ski's Command Window (at Bottom)

Other Windows

Some commands, such as **help**, **isyms**, and **symlist**, cause ***xski*** and ***ski*** to create additional windows. When ***xski*** creates an additional window, it adds scroll bars if there is more information than will fit. As an example, the output window created by ***xski*** for the **symlist** command is shown in Illustration 44: xski's Symbol List Window. ***xski*** understands the Page Up and Page Down keys and the arrow keys. The Page Up and Page Down keys scroll through the window a windowful at a time, with one line of overlap. The up-arrow and down-arrow keys scroll through the window a line at a time.

When ***ski*** needs to display additional information, it does so by overwriting the four standard windows. ***ski*** sends the information through a pager, using `less` by default. When the pager finishes, ***ski*** refreshes the screen with the standard ***ski*** windows. If you prefer to use a different pager, for example `more` or `page`, set the PAGER environment variable accordingly, before starting the simulator.



```
Value            Name
0000000000000458 DEBUG_LINE
40000000000001c8 __text_start
4000000000000b40 _DYNAMIC
4000000000005300 _main
4000000000005730 main
4000000000005a40 _start
4000000000006bd0 ___exit
4000000000006e20 _atexit
40000000000073c0 _isalnum
4000000000007840 _isalpha
4000000000007cc0 _iscntrl
4000000000008140 _isdigit
40000000000085c0 _isgraph
4000000000008a40 _islower
```

Illustration 44: xski's Symbol List Window

4 Command Language

The Ski command language is simple, efficient, and easy to learn. It consists of commands you can invoke from the keyboard or from a command file (see "Command Files" on page 84). Each command is given with an appropriate set of arguments (some optional) to further qualify the command. Commonly-used commands may be abbreviated as described in "Command Reference" on page 87 and commands may be repeated easily. A limited on-line help facility (the **`help`** command) is provided for quick reference. This chapter presents the syntax of the command language. Information about specific commands (command semantics) is in later chapters and in "Command Reference" on page 87.

Command Entry

xski and ***ski*** provide similar mechanisms for controlling the simulator. Both provide for direct keyboard entry of commands. In addition, ***xski*** offers buttons, menus, and the Command History to minimize typing, as described in "The xski Main Window" on page 48, and ***ski*** provides the command repetition mechanism for the same purpose, as described in "The ski Command Window" on page 49. You give a command to Ski by typing the command name at the keyboard followed by operands and the enter/return key. (Use the **`help`** command to see a menu of available commandsor **`help`** followed by the command name to see the command syntax.) ***xski*** displays the command you typed in the Command area of the Main Window. ***ski*** displays the command in the Command Window at the bottom of the screen following the **`*`** prompt. Commands are case sensitive. When you hit the enter/return key, Ski acts on your command and updates the screen to reflect any changes caused by the command. For example, the command

```
db
```

causes the Data Window to show the contents of lower addresses in memory.

Command Arguments

Some commands, such as **`save`**, require additional information. If you don't provide the information, Ski displays an error message. Some commands have optional arguments. As described in "Syntax Conventions" on page 3, command summaries in this manual show optional arguments surrounded by square brackets [*like this*]. If you don't specify an optional argument, Ski uses a suitable default value. For example,

```
    pf 3
```

causes the Program Window to advance three bundles after the last bundle in the Program Window, while

```
    pf
```

alone moves the Program Window ahead one windowful. Some arguments can be supplied in a list, one or more times; these are shown by putting a plus sign ("+") after the argument name *like this*+. For example, the syntax description for the **`=1`** command is:

`=1` *address_or_symbol value*+

which suggests that the command

```
    =1 __data_start 12 56 90 cd
```

assigns the hexadecimal values 12, 56, 90, and cd to the four bytes starting at the location specified by the symbol **`__data_start`**. Brackets and plus signs can be combined, [*like this*]+, to signify optional arguments that can be supplied zero or more times.

Command Sequences, Repetition, and Abbreviation

You can type multiple commands on a single command line by separating the individual commands with semicolons ("**`;`**").

This is called a "command sequence". Command sequences make re-executing a series of commands easy, using the Command History mechanism of ***xski*** (see "The xski Main Window" page 48) or the command repetition mechanism of ***ski*** (see "The ski Command Window" on page 49). For example, you might want to repeatedly execute the commands "**`step 100`**" and "**`eval my_buffer`**". This pair of commands would execute one hundred instructions and then print the value of (your) variable named "**`my_buffer`**". By combining these two commands into one command sequence, i.e., "**`step 100 ; eval my_buffer`**", you can use the Command History or command repetition mechanism to run these commands over and over. (The spaces around the semicolon are optional but improve readability.)

There is no grouping construct in Ski. This can be important when you write command files: when you want to execute commands conditionally using the **`if`** command, you cannot use the semicolon to group several commands into the "then" or "else" clauses. Instead, you must use labels and the **`goto`** command. The "Command Files" on page 84 discusses command files in depth.

Most commands may be abbreviated, some to a single letter. A command may be abbreviated to the shortest prefix which is not also a prefix of a command which precedes it in the command menu. (See "Command Reference" on page 87.)

Argument Specification

The arguments which are given with commands are, in general, obvious and natural. The description which follows should clarify those cases which are not. The terms defined here are used in the command summaries throughout the remainder of this manual.

Numeric Arguments

Many commands accept numeric arguments. The argument may be an address, a value, an execution count, or some other variable which is best expressed numerically.

Numbers and Counts

Some commands take arguments that are naturally expressed in hexadecimal: addresses, for example. Other commands take arguments that are naturally expressed in decimal, such as the number of instructions to simulate with the **`step`** command. To make using Ski easier, some Ski commands default to interpreting their arguments as (hexadecimal) *numbers* and some default to interpreting their arguments as (decimal) *counts*. You can always override the default interpretation by specifying a radix override, as described below.

Hexadecimal digits may be upper or lower case. The default radix may be overridden by preceding the *number* or *count* with **`0D`** or **`0d`** for decimal, **`0X`** or **`0x`** for hexadecimal, **`0O`** or **`0o`** (zero-oh) for octal, and **`0B`** or **`0b`** for binary. Since both the decimal and binary prefixes look like hexadecimal, hexadecimal values such as **`0d600000`** and **`0b100000`** must be specified either with an explicit hexadecimal prefix, as in **`0x0d600000`** and **`0x0b100000`**, or without the leading **`0`**, as in **`d600000`** and **`b100000`**.

Expressions

Wherever a *number* or *count* is needed, you can use a numeric expression instead, with parenthesis as needed for grouping. No spaces are allowed in an expression. In an expression whose result will be used as a *number*, numbers not preceded by a radix override are assumed to be hexadecimal. If the result will be used as a *count*, numbers not preceded by a radix override are assumed to be decimal. For example, the **`step`** command expects a *count* operand, so the command

```
step r0+10
```

steps (decimal) ten instructions. On the other hand, the **`pj`** command expects an address operand, which is a *number*, so the command

```
pj r0+10
```

displays (hexadecimal) address 0x10 in the Program Window. (**`r0`** is hardwired to always return a zero when read.)

The available operators are shown in order from higher to lower precedence in Table 1: Ski Simulator Arithmetic and Logic Operators. Operator precedence rules follow the C language rules.

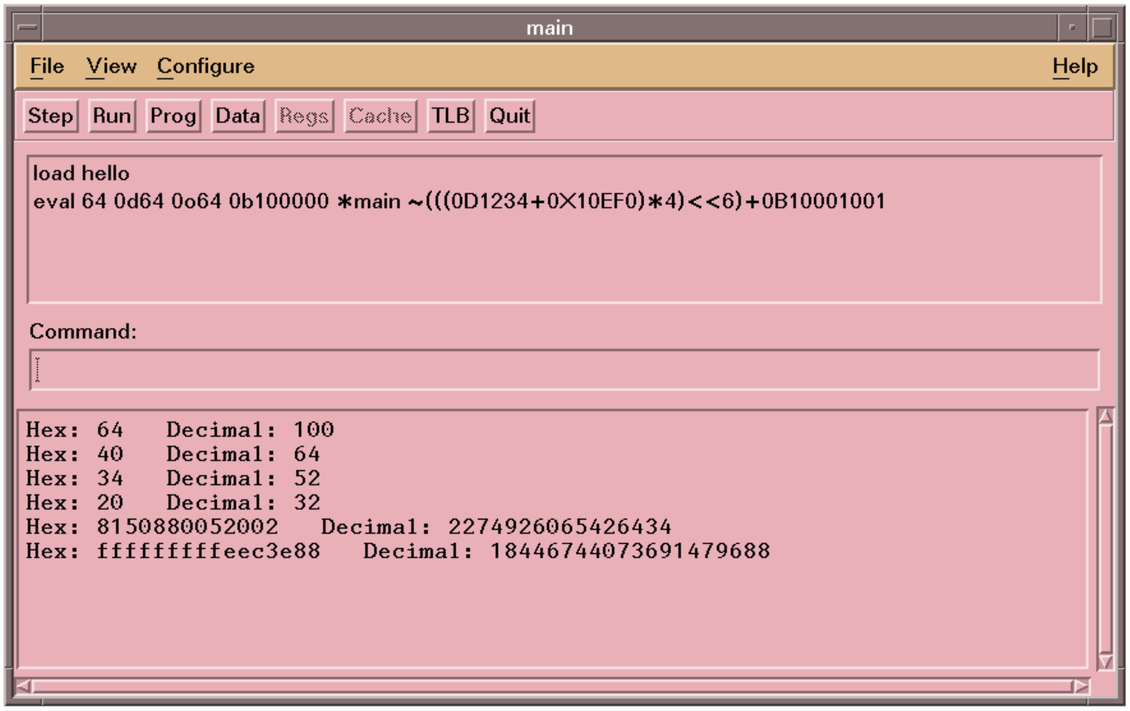
Operator	Description
`( )`	group operators with operands
`! ~ + - *`	opposite truth value, logical one's complement, unary plus, unary minus, dereference: treat as an address and read eight bytes
`* /`	multiply, divide
`+ -`	add, subtract
`<< >>`	logical left shift, logical right shift
`< <= > >=`	less than, less than or equal to, greater than, greater than or equal to
`== !=`	equal to, not equal to
`&`	bitwise and
`^`	bitwise exclusive or
`|`	bitwise or
`&&`	logical and
`||`	logical or

Table 1: Ski Simulator Arithmetic and Logic Operators

As an example, in ***xski***,

```
    eval 64 0d64 0o64 0b100000 *main ~(((0D1234+0X10EF0)*4)<<6)+0B10001001
```

prints the values of the six expressions in the Main Window, as shown in Illustration 45: xski Evaluating Expressions. The first expression is taken as a hexadecimal number, the second as a decimal number, the third as an octal number, and the fourth as a binary number. The fifth expression is the value at the location specified by the symbol "**`main`**" (the first 64 bits of the code bundle at that location), and the sixth expression is the result of some arithmetic.



```
load hello
eval 64 0d64 0o64 0b100000 *main ~(((0D1234+0X10EF0)*4)<<6)+0B10001001
```

```
Hex: 64   Decimal: 100
Hex: 40   Decimal: 64
Hex: 34   Decimal: 52
Hex: 20   Decimal: 32
Hex: 8150880052002   Decimal: 2274926065426434
Hex: ffffffffeec3e88   Decimal: 18446744073691479688
```

Illustration 45: xski Evaluating Expressions

Addresses

An address is specified by a 64 bit hexadecimal number. For example, the command

```
pj 1000
```

repositions ("jumps") the Program Window to address 0x1000. As discussed in "Application-Mode and System-Mode Simulation" on page 66, Ski supports generic addresses in application-mode programs (that is, the concept of "virtual memory" doesn't apply to application mode programs), and physical and virtual addresses in system-mode programs. For system-mode programs, the *psr.dt* and *psr.it* bits control whether Ski interprets addresses as physical or virtual. In some cases, you may need to change the value of one or both of these bits temporarily, so that Ski will interpret addresses the way you want. You should restore the bit values before resuming simulation, of course. You can set the *psr.dt* bit with the Ski command "**`= psr.dt 1`**" and clear the bit with "**`= psr.dt 0`**". The corresponding commands for the *psr.it* bit are "**`= psr.it 1`**" and "**`= psr.it 0`**", respectively.

Addresses may be computed using expressions. For example, the command

```
dj 1000+0d50
```

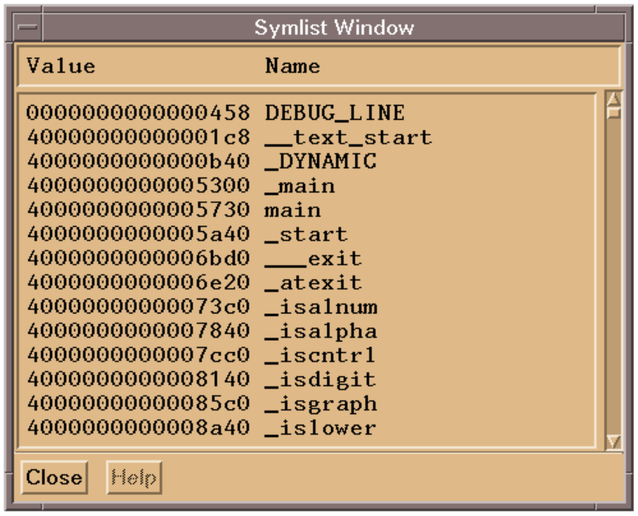
repositions ("jumps") the Program Window to address 1032, because 1000 (hexadecimal) added to 50 (decimal) is 1032 (hexadecimal). Address expressions are particularly useful in symbolic constructs, as described below.

Symbolic Arguments

A symbol is a sequence of characters (a "name"). Examples of symbols are program-defined symbols, registers, internal variables, labels, and filenames. Arguments may (and sometimes must) be expressed symbolically.

Program-Defined Symbols

A program-defined symbol is an identifier which can be used as a mnemonic for a memory location. Program-defined symbol names are defined in the executable file for the program being simulated. Some symbols are common, well-known names (e.g. **`printf`**, **`main`**), and others are defined by the programmer (e.g. **`foo`**, **`bar`**). The **`symlist`** command shows you the symbols sorted by address, as Illustration 46: xski's Symbol List Window shows.



```
Value             Name
0000000000000458 DEBUG_LINE
40000000000001c8 __text_start
4000000000000b40 _DYNAMIC
4000000000005300 _main
4000000000005730 main
4000000000005a40 _start
4000000000006bd0 ___exit
4000000000006e20 _atexit
40000000000073c0 _isalnum
4000000000007840 _isalpha
4000000000007cc0 _iscntrl
4000000000008140 _isdigit
40000000000085c0 _isgraph
4000000000008a40 _islower
```

Illustration 46: xski's Symbol List Window

Registers

A register name is a predefined mnemonic for a processor register. The general registers, for example, are referred to as **`r0`**, **`r1`**, ..., **`r127`**. (The register names Ski recognizes are listed in "Register Names" on page 93.) For example, the command

```
    = r31 ip
```

assigns the value contained in the **`ip`** register to general register 31. (For a description of the **`=`** command, see "Changing Registers and Memory with Assignment Commands" on page 75.) Wherever the simulator expects you to supply a numeric argument, you can use a register instead. You may only refer to currently-visible registers, according to the stacking and rotation mechanisms of the IA-64 architecture.

Internal Variables

The simulator provides internal variables for you to use in command files (see "Command Files" on page 84). These internal variables are read-only; you cannot change their values. You can refer to an internal variable in any context where you could refer to an IA-64 register. Ski has four internal variables:

`$cycles$`

The total number of "virtual cycles" simulated. A virtual cycle is a cycle on a machine with an very large number of execution units and very fast memory; a real IA-64 processor may take more cycles. In a command file, you might use this variable to gather statistics about the efficiency of a particular compiler optimization algorithm. The value of

`$cycles$` is always equal to the value of **`$insts$`** for IA-32 programs.

`$exited$`

The value 0 until the simulated program exits. Then the variable takes the value 1. In a command file, you would use **`$exited$`** to detect a program termination. Program termination is defined for IA-64 application-mode programs as a call to the **`exit()`** function or the receipt of an unhandled signal. For IA-64 system-mode programs, normal termination is defined to be a call to the Simulator System Call exit function or execution of BREAK 0 instruction. This variable is not supported for IA-32 programs in application-mode or system-mode. (See "Application-Mode and System-Mode Simulation" on page 66 for details on these modes.)

`$heap$`

This variable has meaning only for IA-64 programs running in application-mode, as described in "Application-Mode and System-Mode Simulation" on page 66. **`$heap$`** marks the address past the "far end" of the simulated heap, that is, the end farthest from the end of the data section. The heap starts at the first sixteen-byte-aligned address after the data section. Ski updates the **`$heap$`** variable as the program being simulated malloc's memory (for programs written in C; adapt accordingly for other programming languages). You can use the **`$heap$`** variable to debug wild pointer problems: if your program has a pointer that allegedly points to a malloc'ed data structure, but the pointer value exceeds **`$heap$`**, the pointer is invalid. For system-mode programs and IA-32 programs, this variable is meaningless, as there is no malloc support.

`$insts$`

The number of instructions that have been simulated so far (including any faulting instructions, for programs running in system-mode, described in "Application-Mode and System-Mode Simulation" on page 66 . In a command file, you might use this variable to stop simulation after a certain number of instructions. The value of **`$insts$`** is always equal to the value of **`$cycles$`** for IA-32 programs.

Labels

Labels (see "Labels and Control Flow in Command Files" on page 84) are names which consist of an alpha (upper or lower case alphabetic, **`$`**, or _), followed by a sequence of alphas or digits (e.g., **`abc123`**, **`$foo_bar`**, etc.) and ending with a colon ("**`:`**"). They may be up to 132 characters long. Labels are used in command files as targets of the **`goto`** command.

Filenames

Filenames are subject to the restrictions of the underlying Linux operating system. Ski performs tilde ("**`~`**") expansion: if you provide a pathname whose first word starts with a tilde, Ski assumes the word is a username and tries to replace it (and the tilde) with the user's home directory. For example, "`~david/hello`" might be expanded to "`/home/david/hello`".

Resolving Ambiguous Symbols and Numbers

Some character sequences can be interpreted in more than one way. For example, the character sequence "b3" can be interpreted as a branch register, a program-defined symbol, or a hexadecimal number. To resolve the ambiguity, Ski looks first in its symbol tables for program-defined symbols and internal variables (which includes register names). If a match is found, the matching value is used, otherwise the character sequence is taken as a number. You can force the numeric interpretation by putting a "**`0x`**" or "**`0X`**" prefix in front of the number, such as "**`0xb3`**". It is undefined whether Ski searches the symbol table for program-defined symbols before or after the internal variable symbol table. Because of this, it is wise to avoid naming global variables and functions with names duplicating any of Ski's internal variables. In practice, this means you should avoid using register names as names of variables and functions in your programs.

5 Screen Manipulation Commands

Ski provides several commands to manipulate windows. These commands let you make major changes of context or fine adjustments. ***xski*** provides more flexibility: you can change the location and size of ***xski*** windows using the mechanisms provided by your window manager, and ***xski*** provides scrollbars in some windows, for minor adjustments.

Register Window Commands

As described in "The Register Window" on page 36, ***xski*** shows all five sets of registers in the Register Window, with scroll bars and pane resizers so you can select what registers to see within each set and how much screen space should be devoted to each set. The **`fr`**, **`gr`**, **`iar`**, **`sr`**, and **`ur`** commands allow you to toggle display of individual sets on and off. Illustration 47: xski's Program Window Showing IA-64 Assembly Language Code on page 61 shows the ***xski*** Register Window.

ski has much less screen space available and therefore shows only one set and only a part of it at a time. The **`fr`**, **`gr`**, **`iar`**, **`sr`**, and **`ur`** commands allow you to choose which register set to see. The **`rf`** and **`rb`** commands let you choose what part of the chosen register set to see Illustration 47: xski's Program Window Showing IA-64 Assembly Language Code in "The Register Window" on page 36, which shows the ***ski*** Register Window.

Summary of Register Window Commands

`rd` [*filename*]

Dump the Register Window to the screen in a new window (***xski***) or using a pager (***ski***), or, if *filename* is provided, to the file given by *filename*. The mnemonic stands for "register dump".

xski Register Window Commands

`fr`

Toggles display of the floating point registers (**`fr`**) pane in the Register Window. See Illustration 50: xski Showing Data as Instruction Bundles on page 64.

`gr`

Toggles display of the general registers (**`gr`**) pane in the Register Window. See Illustration 49: xski's Assembly Language Dump Window on page 63.

`iar`

Toggles display of the IA-32 registers (**`eax`**, **`ebx`**, **`esp`**, etc.) pane in the Register Window. See Illustration 52: xski's Hexadecimal Dump Window on page 65.

`sr`

Toggles display of the system registers (**`cr`**, **`rr`**, **`pkr`**, **`dbr`**, **`ibr`**, **`pmc`**, and **`pmd`**) pane in the Register Window. See Illustration 51: xski Showing Data in Raw Hexadecimal and ASCII on page 64.

`ur`

Toggles display of the user registers (**`pr`**, **`br`**, **`ar`**, **`ip`**, *psr.um*) pane in the Register Window. See Illustration 48: xski's Program Window Showing Intermixed C and IA-64 Assembly Code on page 62

ski Register Window Commands

`fr`

Displays the floating point registers (**`fr`**) in the Register Window.

`gr`

Displays the general registers (**`gr`**) in the Register Window.

`iar`

Displays the IA-32 (**`eax`**, **`ebx`**, **`esp`**, etc.) registers in the Register Window.

`sr`

Displays the system registers (**`cr`**, **`rr`**, **`pkr`**, **`dbr`**, **`ibr`**, **`pmc`**, and **`pmd`**) in the Register Window.

`ur`

Displays the user registers (**`pr`**, **`br`**, **`ar`**, **`ip`**, *psr.um*) in the Register Window.

`rf` [*count*]

Moves the Register Window "forward" (scrolls down) through the currently-displayed register set. The Register Window is scrolled *count* lines. If *count* is omitted, the Register Window scrolls down one windowful less one line, i.e. the last line of the old window is displayed as the first line of the new window.

`rb` [*count*]

Moves the Register Window "backward" (scrolls up) through the currently-displayed register set. The Register Window is scrolled *count* lines. If *count* is omitted, the Register Window scrolls up one windowful less one line, i.e. the first line of the old window is displayed as the last line of the new window.

Program Window Commands

The Program Window displays disassembled instructions, one instruction per line. (See "The Program Window" on page 41.)

`pj` [*address*]

If *address* is specified, repositions ("jumps") the Program Window so that the IA-64 bundle or IA-32 instruction containing the specified address is second in the window. If no *address* is given, jumps to the previous location. The mnemonic stands for "program jump".

`pf` [*count*]

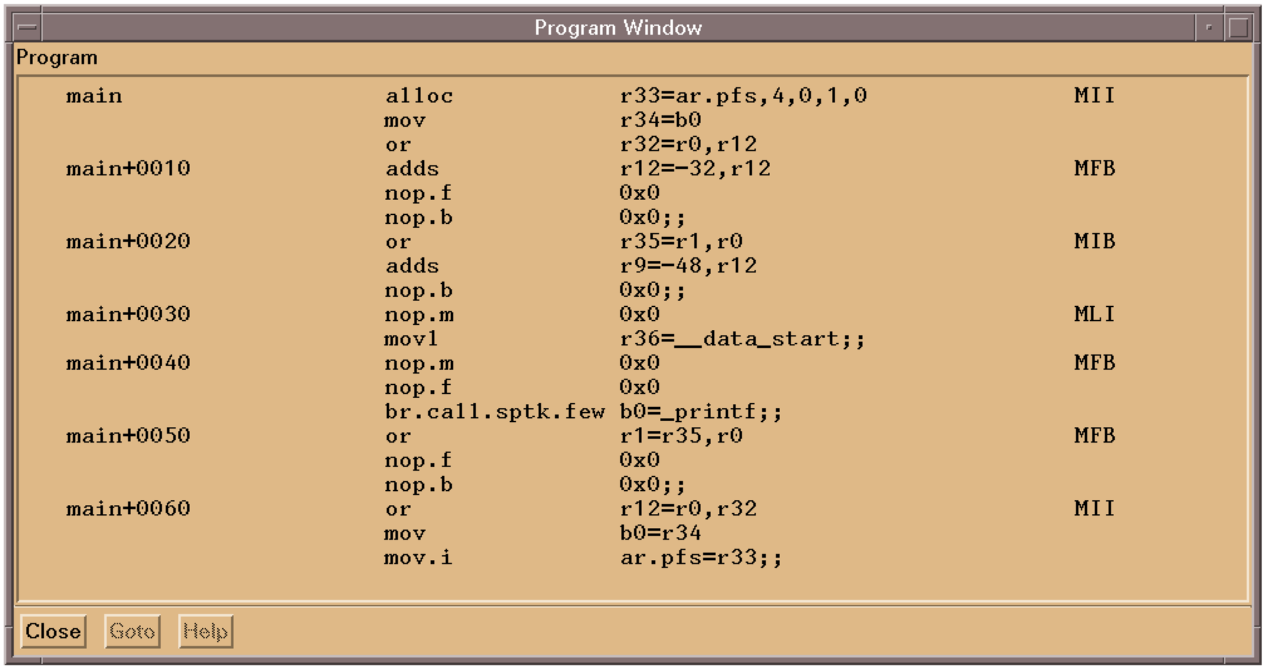
Moves the Program Window forward *count* IA-64 bundles or IA-32 instructions. If *count* is not specified, moves the Program Window forward one windowful less one bundle or instruction. The mnemonic stands for "program forward".

`pb` [*count*]

Moves the Program Window backward *count* IA-64 bundles or IA-32 instructions. If *count* is not specified, moves the Program Window backward one windowful less one bundle or instruction. The mnemonic stands for "program backward".

`pa`

Display the program being simulated in assembly language only, as shown in Illustration 47: xski's Program Window Showing IA-64 Assembly Language Code. This command is valid for IA-64 code only. The mnemonic stands for "program display assembly".

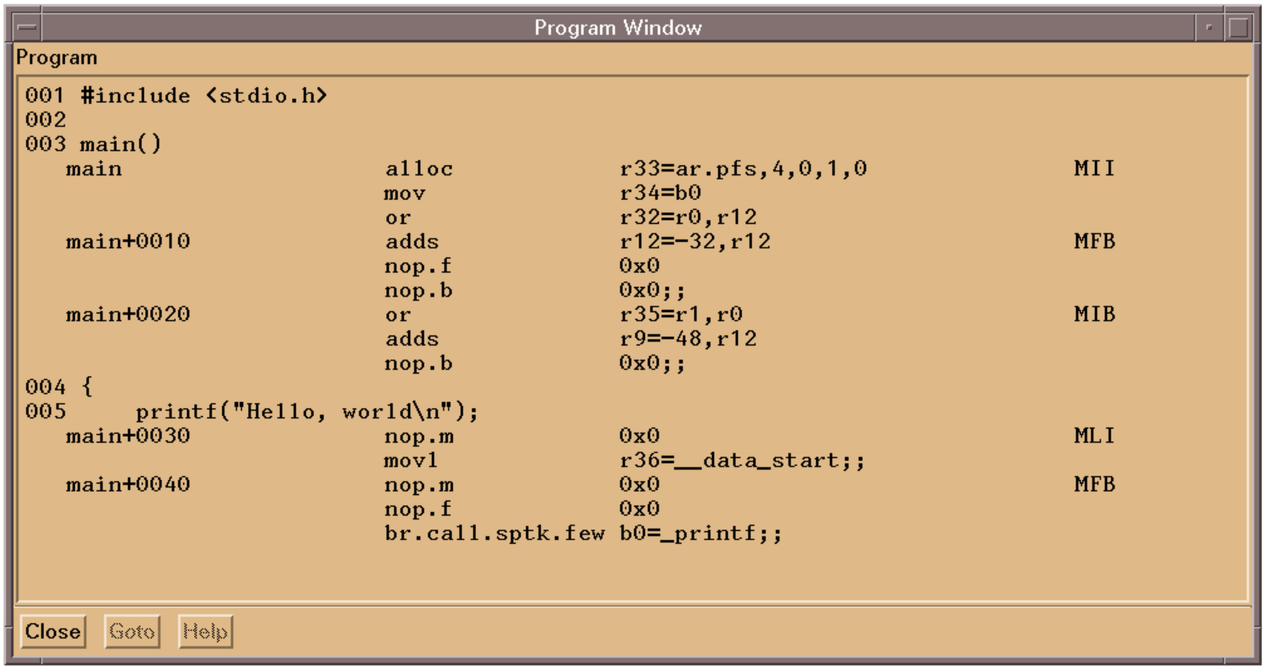


```
main            alloc               r33=ar.pfs,4,0,1,0    MII
                mov                 r34=b0
                or                  r32=r0,r12
main+0010       adds                r12=-32,r12           MFB
                nop.f               0x0
                nop.b               0x0;;
main+0020       or                  r35=r1,r0             MIB
                adds                r9=-48,r12
                nop.b               0x0;;
main+0030       nop.m               0x0                   MLI
                movl                r36=__data_start;;
main+0040       nop.m               0x0                   MFB
                nop.f               0x0
                br.call.sptk.few    b0=_printf;;
main+0050       or                  r1=r35,r0             MFB
                nop.f               0x0
                nop.b               0x0;;
main+0060       or                  r12=r0,r32            MII
                mov                 b0=r34
                mov.i               ar.pfs=r33;;
```

Illustration 47: xski's Program Window Showing IA-64 Assembly Language Code

pm

Display the program being simulated in its source code form with the assembly language translation mixed in, as shown in Illustration 48: xski's Program Window Showing Intermixed C and IA-64 Assembly Code . The source code display is for your convenience only; you cannot interact with the source code, e.g., modify the source code, click on a variable name to see its value in the Data Window, and so on. The source code is not embedded in the executable file. Instead, the compiler and linker place into the executable file a record of the location and filename of the source code. The source code file must be available to Ski in the location recorded in the executable file. In practice, this means you will want to run ***xski*** or ***ski*** from the directory where the program was compiled. (See "The Program Window" on page 41 for more information on source code compilation.) This command is valid for IA-64 code only. The mnemonic stands for "program display mixed".

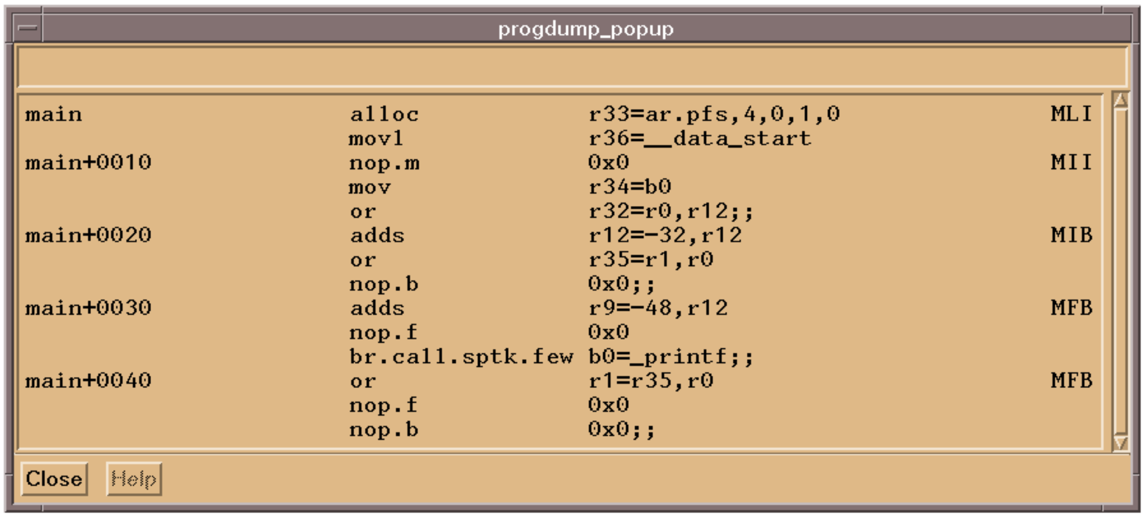


```
001 #include <stdio.h>
002
003 main()
   main                    alloc            r33=ar.pfs,4,0,1,0            MII
                           mov              r34=b0
                           or               r32=r0,r12
   main+0010               adds             r12=-32,r12                   MFB
                           nop.f            0x0
                           nop.b            0x0;;
   main+0020               or               r35=r1,r0                     MIB
                           adds             r9=-48,r12
                           nop.b            0x0;;
004 {
005     printf("Hello, world\n");
   main+0030               nop.m            0x0                           MLI
                           movl             r36=__data_start;;
   main+0040               nop.m            0x0                           MFB
                           nop.f            0x0
                           br.call.sptk.few b0=_printf;;
```

Illustration 48: xski's Program Window Showing Intermixed C and IA-64 Assembly Code

pd *starting_address ending_address* [*filename*]

Dump the assembly language translation of the program in the area between the two addresses (inclusive) to the screen (***ski***) or to a window (***xski***) if no *filename* is given, or to the specified file if one is. Source code will not be dumped along with the assembly language, even if the **pm** command is given. Illustration 49: xski's Assembly Language Dump Window shows an example of an assembly language dump of the program in Illustration 47: xski's Program Window Showing IA-64 Assembly Language Code and Illustration 48: xski's Program Window Showing Intermixed C and IA-64 Assembly Code . The mnemonic stands for "program dump".



```
main            alloc               r33=ar.pfs,4,0,1,0          MLI
                movl                r36=__data_start
main+0010       nop.m               0x0                         MII
                mov                 r34=b0
                or                  r32=r0,r12;;
main+0020       adds                r12=-32,r12                 MIB
                or                  r35=r1,r0
                nop.b               0x0;;
main+0030       adds                r9=-48,r12                  MFB
                nop.f               0x0
                br.call.sptk.few    b0=_printf;;
main+0040       or                  r1=r35,r0                   MFB
                nop.f               0x0
                nop.b               0x0;;
```

Illustration 49: xski's Assembly Language Dump Window

Data Window Commands

The Data Window displays an area of memory in hexadecimal format and, if the window is wide enough, an ASCII translation. (See "The Data Window" on page 46.) The commands to adjust the Data Window are similar to those for the Program Window and are described below.

Summary of Data Window Commands

`dj` [*address*]

If *address* is specified, repositions ("jumps") the Data Window so that the bytes containing the specified address are first in the window. If no *address* is given, jumps to the previous location. The mnemonic stands for "data jump".

`df` [*count*]

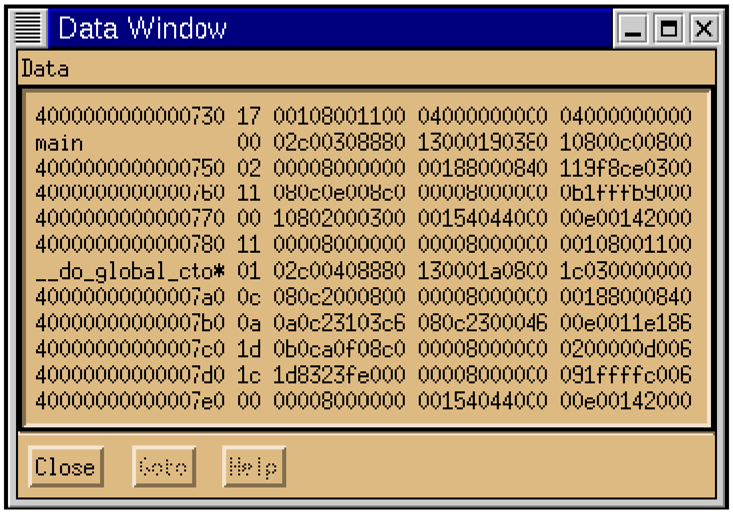
Moves the Data Window forward *count* display lines or one windowful if *count* is not specified. The mnemonic stands for "data forward".

db [*count*]

Moves the Data Window backward *count* display lines or one windowful if *count* is not specified. The mnemonic stands for "data backward".

`dbndl`

Displays the data as hexadecimal instruction bundles, as shown in Illustration 50: xski Showing Data as Instruction Bundles and in on page . It is your responsibility to ensure that the Data Window is actually positioned on instructions; if not, Ski will dutifully display nonsense. The first column displays the address. The second column displays the template field. The remaining three columns display the 41-bit instructions from slots 0, 1, and 2, with the least-significant bit to the right. The mnemonic stands for "data window bundle".

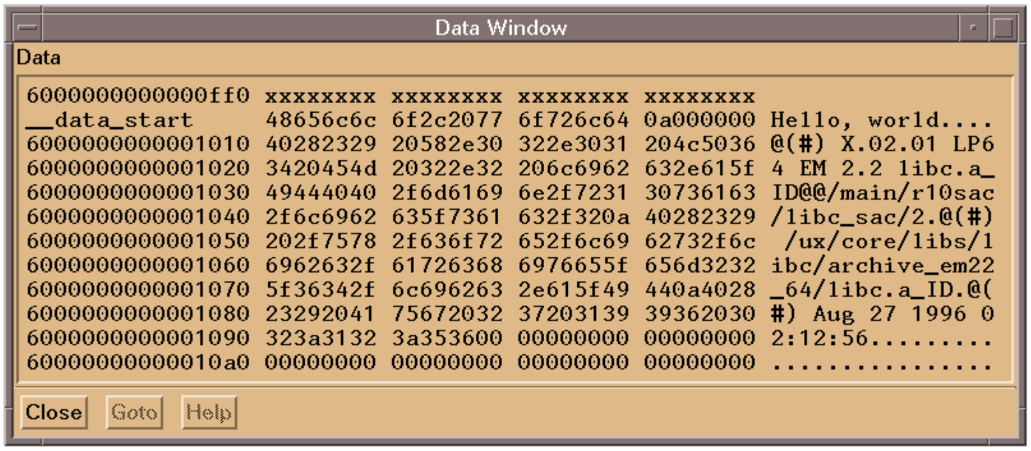


```
4000000000000730 17 00108001100 04000000000 04000000000
main             00 02c00308880 130001903E0 10800c00800
4000000000000750 02 00008000000 00188000840 119f8ce0300
4000000000000760 11 080c0e008c0 000080000C0 0b1fffb9000
4000000000000770 00 10802000300 001540440C0 00e00142000
4000000000000780 11 00008000000 000080000C0 00108001100
__do_global_cto* 01 02c00408880 130001a08C0 1c030000000
40000000000007a0 0c 080c2000800 000080000C0 00188000840
40000000000007b0 0a 0a0c23103c6 080c2300046 00e0011e186
40000000000007c0 1d 0b0ca0f08c0 000080000C0 0200000d006
40000000000007d0 1c 1d8323fe000 000080000C0 091ffffc006
40000000000007e0 00 00008000000 001540440C0 00e00142000
```

Illustration 50: xski Showing Data as Instruction Bundles

dh

Displays the data as raw hexadecimal with an ASCII translation, as shown in Illustration 51: xski Showing Data in Raw Hexadecimal and ASCII. The mnemonic stands for "data window hexadecimal".

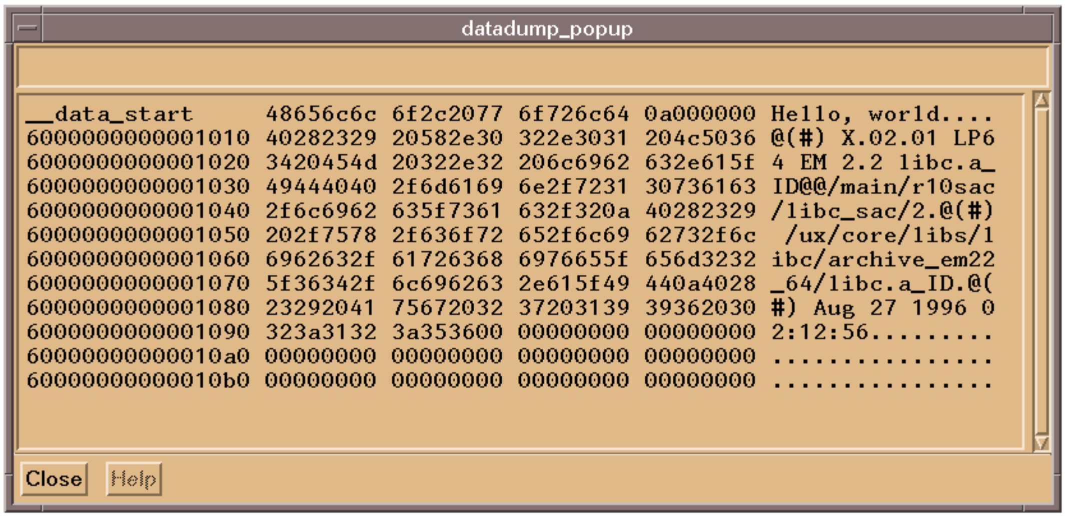


```
6000000000000ff0 xxxxxxxx xxxxxxxx xxxxxxxx xxxxxxxx
__data_start     48656c6c 6f2c2077 6f726c64 0a000000 Hello, world....
6000000000001010 40282329 20582e30 322e3031 204c5036 @(#) X.02.01 LP6
6000000000001020 3420454d 20322e32 206c6962 632e615f 4 EM 2.2 libc.a_
6000000000001030 49444040 2f6d6169 6e2f7231 30736163 ID@@/main/r10sac
6000000000001040 2f6c6962 635f7361 632f320a 40282329 /libc_sac/2.@(#)
6000000000001050 202f7578 2f636f72 652f6c69 62732f6c  /ux/core/libs/l
6000000000001060 6962632f 61726368 6976655f 656d3232 ibc/archive_em22
6000000000001070 5f36342f 6c696263 2e615f49 440a4028 _64/libc.a_ID.@(
6000000000001080 23292041 75672032 37203139 39362030 #) Aug 27 1996 0
6000000000001090 323a3132 3a353600 00000000 00000000 2:12:56.........
60000000000010a0 00000000 00000000 00000000 00000000 ................
```

Illustration 51: xski Showing Data in Raw Hexadecimal and ASCII

dd *starting_address ending_address* [*filename*]

Dump the memory area between the two addresses (inclusive) to the screen (***ski***) or window (***xski***) if no *filename* is given or to the specified file if one is. The dump will be in the format selected by the most recent **`dbndl`** or **`dh`** command. An example of a hexadecimal dump is shown in Illustration 52: xski's Hexadecimal Dump Window. The mnemonic stands for "data dump".



```
__data_start     48656c6c 6f2c2077 6f726c64 0a000000 Hello, world....
6000000000001010 40282329 20582e30 322e3031 204c5036 @(#) X.02.01 LP6
6000000000001020 3420454d 20322e32 206c6962 632e615f 4 EM 2.2 libc.a_
6000000000001030 49444040 2f6d6169 6e2f7231 30736163 ID@@/main/r10sac
6000000000001040 2f6c6962 635f7361 632f320a 40282329 /libc_sac/2.@(#)
6000000000001050 202f7578 2f636f72 652f6c69 62732f6c  /ux/core/libs/l
6000000000001060 6962632f 61726368 6976655f 656d3232 ibc/archive_em22
6000000000001070 5f36342f 6c696263 2e615f49 440a4028 _64/libc.a_ID.@(
6000000000001080 23292041 75672032 37203139 39362030 #) Aug 27 1996 0
6000000000001090 323a3132 3a353600 00000000 00000000 2:12:56.........
60000000000010a0 00000000 00000000 00000000 00000000 ................
60000000000010b0 00000000 00000000 00000000 00000000 ................
```

Illustration 52: xski's Hexadecimal Dump Window

6 Program Simulation

Ski's main responsibility is to simulate IA-64 instructions and programs built from these instructions. Many commands and features are supplied to provide you with a great deal of flexibility in using Ski.

Application-Mode and System-Mode Simulation

Ski supports two instruction sets and two modes of simulation. The two instruction sets supported by Ski are the IA-64 instruction set and a subset of the traditional IA-32 instruction set, often called the "Intel x86" instruction set.

Ski's two simulation modes let you simulate an application program ("application-mode") or an operating system or firmware ("system-mode"). For IA-64 programs, Ski determines the mode based on the presence or absence of the ***`_atexit`*** symbol. (If you strip symbols from your IA-64 program, Ski will not find ***`_atexit`*** and will assume your program is a system-mode program.) For IA-32 programs, you select the mode, using the ***`iaload`*** command for application-mode simulation and the ***`romload`*** command for system-mode simulation. Program loading is discussed in "Program Loading".

Ski Support for Application-Mode Programs

To support application-mode programs, Ski emulates a Linux operating system (for IA-64 programs) or an MS-DOS operating system (for IA-32 programs).

Application-Mode IA-64 Programs

For IA-64 programs, Ski provides (simulated) memory for the text and data portions of the program's address space. Ski also manages a growable heap for the C language's malloc() function, a growable Register Save Engine area, and a growable stack. As your program runs, Ski tracks the memory references emitted by the program. Ski tries to distinguish between reasonable references and ridiculous references indicative of wild pointers. To track stack-based data structures, Ski adds stack pages when it notices a reference to a location just past the end of the stack. To track heap-based data structures, Ski provides an implementation of the malloc() family of functions. ("Linux and MS-DOS ABI Emulation" on page 71 discusses Ski's pseudo-operating system in detail.) Ski tracks pages used by the Register Save Engine as well.

Application program calls to Linux system functions are emulated by the simulator or passed to the host Linux operating system; unsupported calls cause simulation to stop. Registers are initialized according to Linux calling conventions. Application mode programs can't access (simulated) I/O devices or privileged registers. Application mode programs can't execute privileged instructions or receive interrupts; any interruptions cause Ski to stop simulation and generate an error message. Application-mode programs never see virtual memory page faults or TLB faults and therefore the ***`sit`*** and ***`sdt`*** simulator commands (see "System-Mode TLB Simulation") are disabled when simulating application-mode programs.

Application-Mode IA-32 Programs

For IA-32 programs, Ski's support is more limited. Ski provides a subset of MS-DOS " ***`int 21`*** " functions. Ski does not simulate Microsoft Windows. Loadable libraries (DLL's), ***`config.sys`*** , and ***`autoexec.bat`*** are not supported. Environment variables are not available to MS-DOS programs. Registers and memory are initialized according to MS-DOS conventions.

Ski Support for System-Mode Programs

A system-mode program is, as far as Ski is concerned, running on a "bare" IA-64 processor. No operating system emulation is provided and the system-mode program has complete access to the simulated IA-64 processor.

System-Mode IA-64 Programs

A system-mode IA-64 program "sees" a more complete simulated environment: writeable registers are initialized to zero, page and TLB faults are simulated and cause a transfer to the interruption vector table (IVT), privileged instructions can be executed, privileged registers can be accessed, and so on. A tricky issue for system-mode simulation is handling I/O because there are no real I/O devices to simulate! Instead, Ski provides a special interface using *BREAK* instructions to implement Simulator SystemCalls (SSC's), which provide access to the console, keyboard, SCSI disk and Ethernet devices. A system-mode IA-64 program can't access the underlying operating system; it "thinks" it's running on a real IA-64 computer.

A system-mode IA-64 program must provide interruption handlers. The program must create a valid Interruption Vector Table (IVT) and set the Interruption Vector Address (IVA) accordingly. You can test your interruption code by creating code that generates conditions corresponding to internal faults, traps, and interrupts, such as divide-by-zero and page-not-present. To test code for external interrupts, use the inter-processor interruption mechanism, as defined by the IA-64 architecture manual. Example assembly code for this is shown in Table 2: Example Code to Simulate an External Interrupt. Timer interruptions can be simulated using the Simulator System Call mechanism.

```
ssm 0x6000           // Set psr.i and psr.ic to 1
mov cr.lid=r0        // For processor 0
movl r4=0xfee00000   // Interrupt block base for proc 0
mov r5=0x10          // Interrupt vector 16
st8 [r4]=r5          // Code branches to iva+0x3000 (the
external
                     // interrupt handler). irr0{16} is set
to 1,
                     // ivr = 0x10
```

Table 2: Example Code to Simulate an External Interrupt

System-Mode IA-32 Programs

Ski does not support IA-32 programs running in system-mode.

System-Mode TLB Simulation

The simulator provides facilities for modeling the TLB's (Translation Lookaside Buffers) for system-mode programs.

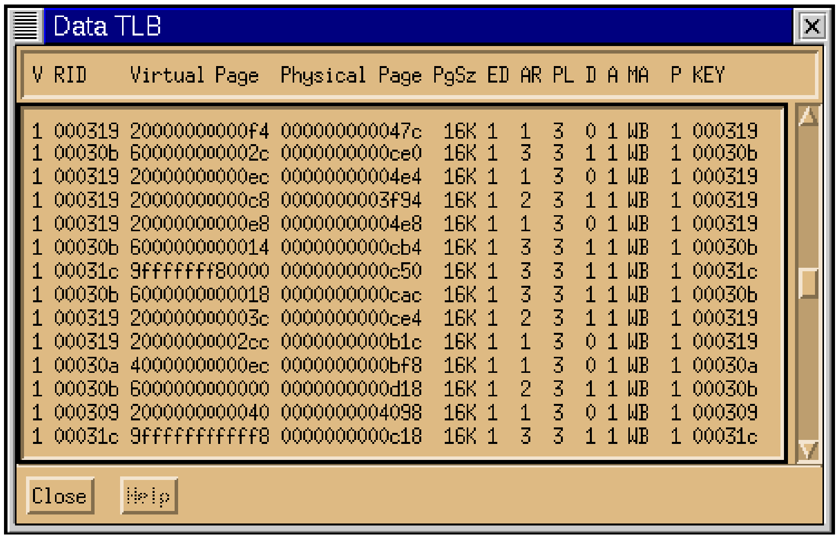
Summary of TLB Display Commands

sit

sdt

When a system-mode IA-64 program is loaded, these commands display information from the Instruction Translation Lookaside Buffer (ITLB) and Data Translation Lookaside Buffer (DTLB), respectively. The simulator displays the entire selected TLB (Translation Registers and the Translation Cache) on the screen, as shown in Illustration 53: sdt Command Output in xski

The " ***V*** " and " ***RID*** " columns represent the V (valid) bit and Region Identifier, respectively, for each TLB entry. The " ***Virtual Page*** " and " ***Physical Page*** " columns show the actual address translation handled by each TLB entry. The " ***PgSz*** ", " ***ED*** ", " ***AR*** ", " ***PL*** ", " ***D*** ", " ***A*** ", " ***MA*** ", and " ***P*** " columns represent the Page Size, Exception Deferral, Access Rights, Privilege Level, Dirty Bit, Accessed Bit, Memory Attribute, and Present fields, respectively, for each TLB entry. Finally, the " ***KEY*** " column represents the Protection Key for each TLB entry. A blank line separates the Translation Registers (TR's) from the Translation Cache (TC). The number of TR's and the size of the TC is implementation-dependent. Current versions of Ski provide 16 TR's and 128 entries for the TC but this may change. If the precise value is important, check the release notes.



```
V RID    Virtual Page  Physical Page PgSz ED AR PL D A MA  P KEY
1 000319 20000000000f4 00000000047c  16K 1  1  3  0 1 WB  1 000319
1 00030b 600000000002c 0000000000ce0 16K 1  3  3  1 1 WB  1 00030b
1 000319 20000000000ec 00000000004e4 16K 1  1  3  0 1 WB  1 000319
1 000319 20000000000c8 0000000003f94 16K 1  2  3  1 1 WB  1 000319
1 000319 20000000000e8 00000000004e8 16K 1  1  3  0 1 WB  1 000319
1 00030b 6000000000014 0000000000cb4 16K 1  3  3  1 1 WB  1 00030b
1 00031c 9fffffff80000 0000000000c50 16K 1  3  3  1 1 WB  1 00031c
1 00030b 6000000000018 0000000000cac 16K 1  3  3  1 1 WB  1 00030b
1 000319 200000000003c 0000000000ce4 16K 1  2  3  1 1 WB  1 000319
1 000319 20000000002cc 0000000000b1c 16K 1  1  3  0 1 WB  1 000319
1 00030a 40000000000ec 0000000000bf8 16K 1  1  3  0 1 WB  1 00030a
1 00030b 6000000000000 0000000000d18 16K 1  2  3  1 1 WB  1 00030b
1 000309 2000000000040 0000000004098 16K 1  1  3  0 1 WB  1 000309
1 00031c 9fffffffffff8 0000000000c18 16K 1  3  3  1 1 WB  1 00031c
```

Illustration 53: sdt Command Output in xski

Misaligned Data Access Trap

If the *psr.ac* bit is set, the IA-64 architecture requires alignment checks on memory accesses; i.e., when data accesses are made to items larger than a byte, the appropriate number of low-order address bits must be zero. If the bit is clear, the IA-64 implementation may choose whether or not to make such checks; Ski chooses to make the checks for references from IA-64 code. When an IA-64 program attempts an misaligned access, the behavior of the simulator depends on whether it is running in application-mode or system-mode. In application-mode, the simulator stops the program and displays an error message. In system-mode, the simulator traps to the unaligned access vector.

Program Loading

The Ski simulator supports loading IA-64 programs in the standard IA-64 ELF executable format and in MS-DOS `.com` and `.exe` formats. ELF files contain enough information to allow the simulator not only to load the program and its data, but also to build a symbol table, properly structure virtual memory, and initialize the screen and ***`ip`*** with the proper values. For IA-64 Linux programs, the *psr.be* bit is always initialized to zero, indicating that the program will run with little-endian byte-order.

The MS-DOS formats do not include symbol table information. Instead, you must supply the information in the form of a mapfile compatible with those created by Microsoft's "ML" linker. If you don't provide Ski with a mapfile, no program-defined symbols will be available. The MS-DOS formats do not specify where to place the program in memory. You must provide this information to Ski yourself. The `.com` format is very basic and is supported with the ***`iaload`*** and ***`romload`*** commands, described in "Summary of Program Loading Commands" The `.exe` format contains header information that is used by the ***`iaload`*** command and ignored by the ***`romload`*** command. For this reason, `.exe` files are not useful in system-mode simulation. For IA-32 programs, only IA-32 (little-endian) byte ordering is supported.

How to Load a Program

There are two ways to load a file. The first way is to run the simulator with a IA-64 (not IA-32) executable program filename as an argument. The file will be loaded immediately after the simulator initializes itself and before any command file specified with the ***-i*** flag is executed. (see "Command Files" on page 84 and "Command Line Flags" on page 33.) An example is " ***xski my_program*** ". The second way is to use the ***load*** , ***iaload*** , or ***romload*** commands, which take the filename as the first argument, for example, " ***load my_program*** ".

Summary of Program Loading Commands

load *filename* [*args*] **+**

Prepare for IA-64 application-mode simulation: Load the file specified by *filename* and prepares to pass the program *args* encoded using the C-language argc/argv mechanism. The file must be an IA-64 ELF file.

iaload *filename address* [*mapfile* [*args*]+]

Prepare for IA-32 application-mode simulation: Load the IA-32 executable file specified by *filename* , which must be an MS-DOS *.com* or *.exe* file and prepare to pass the program *args* encoded using MS-DOS command line argument conventions. The *address* specifies where Ski should load the program. This should be a physical address; virtual addressing is only supported for system-mode programs. The value you provide is used, along with information from the *.exe* file or MS-DOS defaults for a *.com* file, to setup the IA-32 execution environment, such as segment descriptors, the stack pointer, etc. The *mapfile* is an ASCII text file providing the mappings between symbols and addresses; it must be compatible in format with the mapfile produced by the Microsoft "ML" linker. The *psr.is* bit is set.

romload *filename address* [*mapfile*]

Prepare for IA-64, IA-32, or mixed system-mode simulation: Load the MS-DOS *.com* -format file specified by *filename* . (The MS-DOS *.com* format is essentially raw binary.) *Address* and *mapfile* are as described for the ***iaload*** command above. The *address* can be physical or virtual, depending on the setting of the *psr.it* bit, as described in "Addresses" on page 55.

Notes about Program Loading

Adding Information after Loading

Sometimes, the load file doesn't contain enough information. In this case, you can use a command file (see "Command Files" on page 84) to add more information. You execute the command file at the appropriate time, generally after loading the program. For example, perhaps you want to test how an application program handles error conditions that are hard to create in a "real" hardware environment. You could load the program and use a command file to create the error condition. Then you would run the program and test its behavior.

As another example, perhaps you want to simulate the transfer of control from a bootstrap program, an interrupt, or an application program to the operating system. You could load the operating system as a system-mode program and use a command file to set up memory and registers to their appropriate state at the instant of the control transfer.

Creating the argc, argv, and envp Parameters

The first time an application-mode simulated program starts, it receives command line parameters and environment variables using the C language argc/argv/envp mechanism. (IA-32 application-mode programs do not receive environment variables.) By default, the program receives the same command line parameters you gave to Ski when you started it. For example, if you invoked Ski as " ***xski my_program foo bar*** ", Ski would start up using the X Window System interface, load the executable IA-64 program *my_program* , and use " ***foo*** ", " ***bar*** ", and environment variables to initialize the argc, argv and envp parameters passed on the memory stack. The environment variables are a copy of the variables Ski received from the shell when it started.

Instead of specifying the executable program on Ski's invocation line as in the example above, you can use the ***`load`*** or ***`iaload`*** commands to load the executable program. You can add extra arguments to ***`load`*** and ***`iaload`*** . Later, when you invoke the ***`run`*** command, Ski will pass the extra arguments to the simulated program as command line parameters. For example, you could issue the command " ***`load my_program foo bar`*** ". When you ***`run`*** the program, Ski would pass " ***`foo`*** " and " ***`bar`*** " to the program as command line parameters using the argc/argv/envp mechanism. Note that IA-32 application-mode programs must be loaded with the ***`iaload`*** command; they cannot be loaded from the Ski invocation line.

Program Execution

Programs may be run in their entirety without interruption, they may be stopped at appropriate places (see "Program Breakpoints" on page 78) and continued, or they may be single-stepped for debugging purposes. The different program execution choices are described below.

You can stop a running simulation in ***ski*** at any time with your interrupt character (usually ^C). The interrupt will be honored at the beginning of simulation of the next instruction. ***xski*** and ***bski*** do not have interrupt handlers; if you use your interrupt character while they are running, they will be terminated by the operating system.

Summary of Program Execution Commands

`run`

Starts / restarts execution of a program at the current ***`ip`*** value. Generally used after a breakpoint is encountered.

`cont`

Same function as the ***`run`*** command. The mnemonic stands for "continue".

`step` [*count*]

With no argument, executes a single instruction. If a *count* is specified, executes *count* instructions.

`step until` *expression*

7 Linux and MS-DOS ABI Emulation

As discussed in "Application-Mode and System-Mode Simulation" on page 66, Ski can provide application programs with a Linux-compatible or MS-DOS-compatible environment. The environments aren't full-blown operating system emulations, however. The most common OS functions are provided, as described below.

Interruptions

The IA-64 architecture defines a large set of interruption types, including faults, traps, and interrupts. Interruptions may happen asynchronously, during an instruction, or between instructions. Like application programs running on a "real" Linux machine, IA-64 application-mode programs in Ski never see interruptions. Instead, Ski translates interruptions into the signal that a real IA-64 Linux kernel would generate. For example, a memory access violation gets translated into the SIGSEGV signal. Similarly, if Ski receives a keyboard signal such as the SIGINT generated (usually) by control-C, it passes this signal on to the IA-64 application. Ski does not accurately simulate the *`siginfo`* and *`sigcontext`* structures that a real IA-64 Linux kernel would pass to a signal handler. Thus, applications relying on either of these parameters cannot be simulated in Ski application mode.

Linux Application Environment

Ski provides a commonly-used subset of the Linux environment to IA-64 application-mode programs. Both statically linked and dynamically linked programs are supported. The argc, argv, and envp parameters are created on the stack as described in "Creating the argc, argv, and envp Parameters" on page 69. Ski initializes the IA-64 registers like this:

`sp` points to the top of the stack.

`bsp`, and **`bspstore`** are initialized in the same way the IA-64 version of Linux is likely to do.

rsc.pl is initialized to 3.

rsc.be and *psr.be* are cleared.

Ski supports the Linux system calls shown in Table 3: Linux System Calls Supported by Ski. This list is subject to change; consult the release notes for the latest information. The data passed between the application program and the simulated Linux environment is interpreted as 64 bit (LP64) quantities.

accept	access	acct	adjtimex
bind	brk	chdir	chmod
chown	chroot	clone (fork & vfork)	close
connect	dup	dup2	execve (IA-32 & IA-64)
exit	fchdir	fchmod	fchown
fcntl	fdatasync	flock	fstat
fstatfs	fsync	ftruncate	getcwd
getdents	getegid	geteuid	getgid
getgroups	getitimer	getpagesize (4KB)	getpeername
getpgid	getpid	getppid	getpriority
getresgid	getresuid	getrlimit	getrusage
getsid	getsockname	getsockopt	gettimeofday
getuid	ioctl	ioperm	kill
lchown	link	listen	lseek
lstat	mkdir	mknod	mmap
mmap2	mount	mprotect	mremap
msgget	msgrcv	msgsnd	msync
nanosleep	open	personality	pipe
poll	pread (not atomic)	pwrite (not atomic)	read
readlink	readv (not atomic)	reboot	recv
recvfrom	recvmsg	rename	rmdir
rt_sigaction	rt_sigpending	rt_sigprocmask	rt_sigsuspend
sched_get_priority_max	sched_get_priority_min	sched_getparam	sched_getscheduler
sched_rr_get_interval	sched_setparam	sched_setscheduler	sched_yield
select	semget	semop	send
sendmsg	sendto	setdomainname	setfsgid
setfsuid	setgid	setgroups	sethostname
setitimer	setpgid	setpriority	setregid
setresgid	setresuid	setreuid	setrlimit
setsid	setsockopt	settimeofday	setuid
shmat	shmdt	shmget	shutdown
sigalstack	socket	socketpair	stat
statfs	swapoff	swapon	symlink
sync	syslog	times	truncate
umask	umount	uname	unlink
ustat	utimes	vhangup	wait4
write	writev (not atomic)		

Table 3: Linux System Calls Supported by Ski

Ski accepts but ignores the system calls shown in Table 4: Linux System Calls Accepted but Ignored by Ski. For those that return an error indication, the errno code is shown in parentheses. All other ignored system calls return with a success indication, having done nothing.

_sysctl (**ENOSYS**)	bdflush (**ENOSYS**)	capget	capset
create_module (**ENOSYS**)	delete_module (**ENOSYS**)	get_kernel_syms (**ENOSYS**)	getpmsg
init_module (**ENOSYS**)	msgctl (**ENOSYS**)	munlockall	nfsservctl
prctl	ptrace (**EOPNOTSUPP**)	putpmsg	query_module (**ENOSYS**)
quotactl (**ENOSYS**)	rt_sigqueueinfo	rt_sigtimedwait	semctl (**ENOSYS**)
sendfile	shmctl (**ENOSYS**)	sysfs (**ENOSYS**)	sysinfo (**ENOSYS**)

Table 4: Linux System Calls Accepted but Ignored by Ski

All other system calls are unsupported. When an IA-64 application-mode program makes an unsupported system call, the simulator stops the simulation and displays an error message.

MS-DOS Application Environment

IA-32 application-mode programs "see" a limited MS-DOS environment. The MS-DOS environment is emulated by creating and initializing an MS-DOS Program Segment Prefix (PSP) and by setting up the stack pointer (**iasp**) and segmentation registers. The arguments you gave with the **iaload** command, such as "**iaload my_program foo bar baz**", are placed in the PSP as if they were command line parameters.

Ski supports the MS-DOS "**INT 20**" call to terminate the simulated program and the "**INT 21**" system calls shown in Table 5: MS-DOS System Calls (in Hexadecimal) Supported by Ski. When an IA-32 program makes an **INT 21** call that's not supported, the simulator stops the simulation and displays an error message.

00: terminate program	02: display character	08: read keyboard without echo
09: display string	2a: get date	2c: get time
30: get version number	3c: create file with handle	3d: open file with handle
3e: close file with handle	3f: read file or device	40: write file or device
44: device status control	44, sub-function 0: get device data	4c: end program
51: get PSP address	62: get PSP address (same as 51)	

Table 5: MS-DOS System Calls (in Hexadecimal) Supported by Ski

Program I/O

Your program may need to read from standard in (stdin: file descriptor 0) and write to standard out (stdout: file descriptor 1) and standard err (stderr: file descriptor 2). As with all Linux programs, these file descriptors are connected, by default, to your keyboard and screen. You can redirect them in the usual way: when you invoke Ski, use the < and > operators recognized by most Linux shells. For example, "**bski -noconsole my_program foo bar baz < test_input >simulated_output**" runs ***bski***, loading the IA-64 program file `my_program` and passing it the arguments **foo** , **bar** and **baz** via the argc/argv mechanism. Because no command file was provided via the **-i** flag (described in "Command Line Flags" on page 33), ***bski*** internally generates a **run** command followed by a **quit** command. The (simulated) program reads on standard in from the file `test_input` and writes on standard out to the file `simulated_output`. Having not been redirected, writes to standard err go to the default place, normally the terminal screen.

8 Debugging

The simulator provides many facilities to help you debug your programs. You can modify the current state of the simulated processor, set program breakpoints, trace program execution, and dump a memory image into a file.

Changing Registers and Memory with Assignment Commands

Use the `=` command to assign a value to a register. The `=` command takes two arguments: the first is the name of a register and the second is the value to be assigned.

To change the contents of memory, you use one of five different commands, depending on whether you want to set a byte, two bytes, four bytes, eight bytes, or a C-language string (a sequence of bytes terminated by a byte with the value zero, the "null" byte). The commands are **`=1`**, **`=2`**, **`=4`**, **`=8`**, and **`=s`** respectively. Each command takes at least two arguments (some take more): an address or symbol or expression resolving to an address, and the new value you want placed there.

Summary of Assignment Commands

`=` *register_name value*

The *value* is assigned to the register specified by *register_name*. The old value is lost. Unless a modifying prefix such as **`0d`**, **`0b`**, or **`0o`** is used, *value* will be treated as a hexadecimal number. Floating point registers must be set piecewise, using the register name (**`f2`** through **`f127`**) followed by a **`.s`** to set the sign, **`.m`** to set the mantissa, or **`.e`** to set the exponent. The first general register, **`r0`**, is "hardwired" to 0 and any attempt to assign to it will be rejected. Similarly, floating registers **`f0`** and **`f1`** are "hardwired" to be 0.0 and 1.0, respectively, and predicate register **`p0`** is "hardwired" to 1 and they too cannot be changed. Some IA-64 registers are read-only according to the IA-64 architecture specification, but all non-hardwired registers are writable with Ski's **`=`** command to assist your debugging.

`=1` *address value+*

`=2` *address value+*

`=4` *address value+*

`=8` *address value+*

The *value* is assigned to the specified location in memory. The old value at the location is lost. The location may be on any allocated page, including instruction pages, as discussed in "Page Allocation". Multiple values, separated by spaces, may be supplied; if so, they will be assigned to sequential memory addresses. Unless a modifying prefix such as **`0d`**, **`0b`**, or **`0o`** is used, *value* will be treated as a hexadecimal number.

The **`=1`** command truncates any extra high-order bytes of the *value* to make a single byte. The **`=2`** command truncates or pads (with zero) the high order bytes of the *value* as necessary to make a two-byte quantity. Similarly, the **`=4`** and **`=8`** commands truncate or pad high order bytes to make four- and eight-byte quantities, respectively.

The **`=2`**, **`=4`**, and **`=8`** commands respect the current value of the *psr.be* bit, which controls whether multi-byte data memory references are big-endian (if the bit is set) or little-endian (if the bit is clear). The bit also controls the format of data display in the Data Window (see "The Data Window" on page 46). You can set the *psr.be* bit with the command "**`= psr.be 1`**" and you can clear it with "**`= psr.be 0`**".

Ski supports physical and virtual addressing. For more information, see "Addresses" on page 55.

`=s` *address string_without_spaces+*

The *string_without_spaces* is assigned to memory locations starting at the location specified by *address*. A null byte is added to the end of the string automatically. The old value at the location is lost. The location may be on any allocated page, including instruction pages, as discussed in "Page Allocation". Multiple values may be supplied, separated by a space. The strings may not contain spaces and quoting it is not a workaround.

Examples of Assignment Commands

`= r1 1234`

The hexadecimal value 0x1234 is assigned to general register 1. The six upper (more significant) bytes are padded with zeroes.

`= r1 ip+10`

The value in **`ip`** added to 0x10 is assigned to general register 1.

`= f2.m 1234 ; = f2.s 1 ; = f2.e 10033`

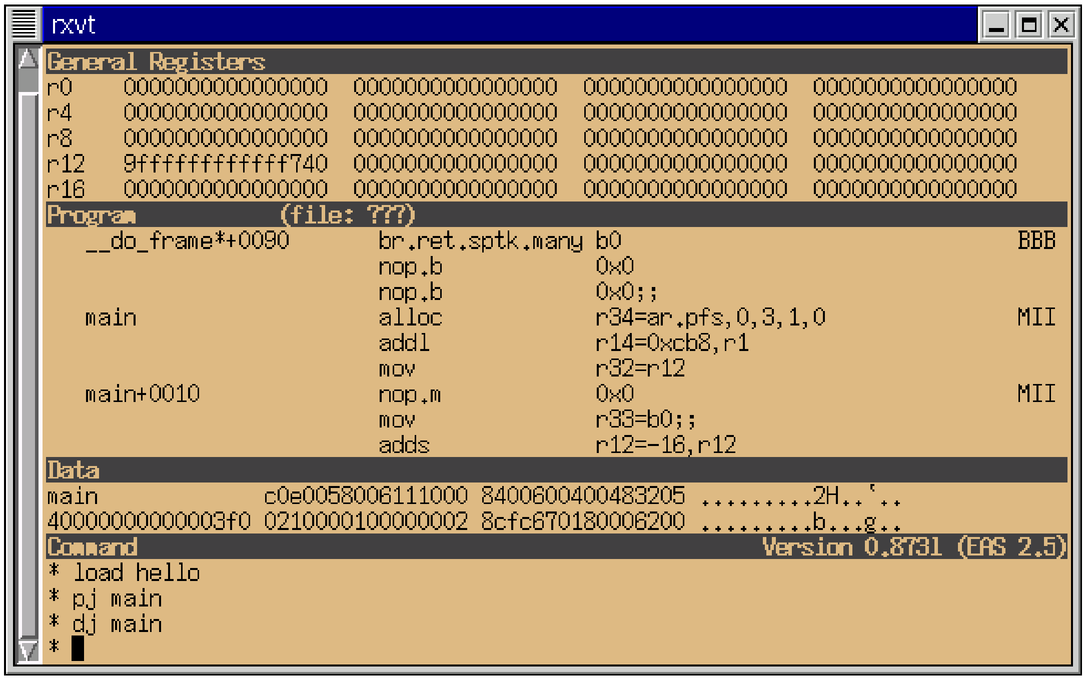
The hexadecimal value 0x30033000000000001234 is assigned to floating register 2. The register now encodes the decimal value of -2.2754, approximately. The "**`= f2.m 1234`**" part sets the mantissa (the 64 low-order bits). The "**`= f2.s 1`**" part encodes the mantissa sign (the most significant of the 82 bits). The "**`= f2.e 10033`**" encodes the 17 exponent bits (which fit between the sign bit and mantissa bits), using a bias of 65,535 (0xffff).

`=4 __data_start+30 0d10 13feffff b3`

The decimal value 10 is assigned to the four bytes starting 48 bytes past the location of the symbol "**`__data_start`**". Because the value 10 occupies only one byte, three high-order zero bytes will be padded in, so the actual value assigned will be 0x0000000a. The value 13feffff is assigned to the four bytes starting 52 bytes past the location of **`__data_start`**. The lower four bytes of branch register 3 will be copied into the four bytes starting 56 bytes past the location of **`__data_start`**. (To assign the value 0xb3, use the **`0x`** prefix.)

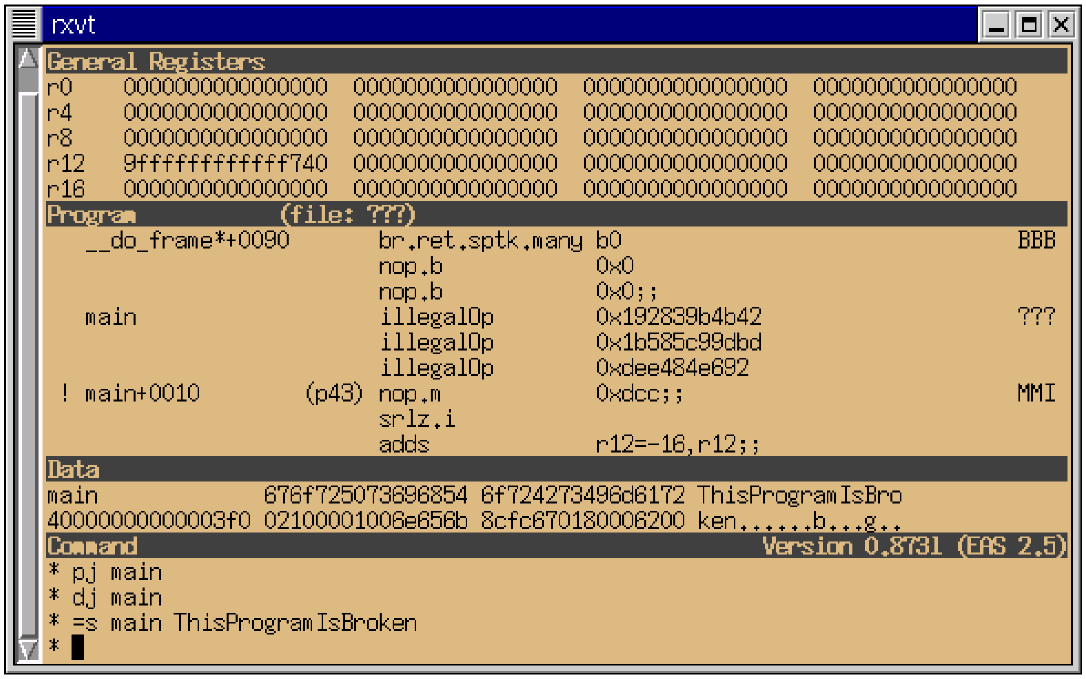
`=s main ThisProgramIsBroken`

The string "**`ThisProgramIsBroken`**" with a null byte appended is placed in memory overwriting the instructions at the start of the program, as shown in the "before" and "after" views of Illustration 54: The Original Program Loaded in ski and Illustration 55: The Program After Assigning a String in ski. (The symbol "**`main`**" traditionally marks the first instruction of a user program written in the C language.) The instructions previously at that location are lost. If you attempt to run the program, it will almost certainly fail! Note that the string is not quoted and has no whitespace.



```
rxvt
General Registers
r0    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r4    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r8    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r12   9ffffffffffff740  0000000000000000  0000000000000000  0000000000000000
r16   0000000000000000  0000000000000000  0000000000000000  0000000000000000
Program             (file: ???)
   __do_frame*+0090          br.ret.sptk.many b0                         BBB
                             nop.b            0x0
                             nop.b            0x0;;
   main                      alloc            r34=ar.pfs,0,3,1,0         MII
                             addl             r14=0xcb8,r1
                             mov              r32=r12
   main+0010                 nop.m            0x0                        MII
                             mov              r33=b0;;
                             adds             r12=-16,r12
Data
main              c0e0058006111000 8400600400483205 .........2H..`..
40000000000003f0  0210000100000002 8cfc670180006200 .........b...g..
Command                                                 Version 0.8731 (EAS 2.5)
* load hello
* pj main
* dj main
*
```

Illustration 54: The Original Program Loaded in ski



```
rxvt
General Registers
r0    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r4    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r8    0000000000000000  0000000000000000  0000000000000000  0000000000000000
r12   9ffffffffffff740  0000000000000000  0000000000000000  0000000000000000
r16   0000000000000000  0000000000000000  0000000000000000  0000000000000000
Program             (file: ???)
   __do_frame*+0090          br.ret.sptk.many b0                         BBB
                             nop.b            0x0
                             nop.b            0x0;;
   main                      illegalOp        0x192839b4b42              ???
                             illegalOp        0x1b585c99dbd
                             illegalOp        0xdee484e692
 ! main+0010           (p43) nop.m            0xdcc;;                    MMI
                             srlz.i
                             adds             r12=-16,r12;;
Data
main              676f725073696854 6f724273496d6172 ThisProgramIsBro
40000000000003f0  02100001006e656b 8cfc670180006200 ken......b...g..
Command                                                 Version 0.8731 (EAS 2.5)
* pj main
* dj main
* =s main ThisProgramIsBroken
*
```

Illustration 55: The Program After Assigning a String in ski

Notes on Assignment

Address Alignment

Ski aligns addresses on natural boundaries: two-byte quantities are aligned on addresses divisible by two, four-byte quantities are aligned on addresses divisible by four, and eight-byte quantities are aligned on addresses divisible by eight. For example, the command

```
=4 __data_start+1 0x12345678
```

results in the message

```
Non word-aligned address. Aligned to 0x6000000000001000
```

and the value is assigned starting one byte before the requested address. ("**`__data_start`**" is a program-defined symbol for **`0x6000000000001000`**.)

Bit-encoded Registers

Many registers are bit-encoded. You can assign to individual bits or to entire registers. For example, you can set the *psr.it* bit with this:

```
    = psr.it 1
```

and you can set the entire Processor Status Register (**`psr`**) with this:

```
    = psr 1234567890abcdef
```

A complete list of the registers and bits Ski recognizes is in "Register Names" on page 93.

Page Allocation

Virtual memory is simulated only for system-mode programs. In system-mode, your program is responsible for page allocation. In application-mode, Ski handles page allocation for you. Either way, if you try to assign data to a non-existent page using the assignment commands, Ski will refuse, with an error message. The assignment commands never cause a TLB miss or replacement.

Evaluating Formulas and Formatting Data

The **`eval`** command evaluates one or more expressions and prints the result(s) in decimal, and hexadecimal. An example of the **`eval`** command and a more complete discussion are in "Expressions" on page 53.

Summary of The `eval` Command

`eval` *expression***+**

Evaluate the *expression*(s) and print the result(s) on the screen. If the *expression* is simply a register name, the value is display in the appropriate format: decimal, hexadecimal, or symbolically, depending on the kind of register. If the expression has any operators, the result is displayed in decimal and hexadecimal. For example, "**`eval ip`**" causes the current value of the **`ip`** register to be displayed symbolically or in hexadecimal. But "**`eval +ip`**" causes the value to be printed out in hexadecimal and decimal.

Program Breakpoints

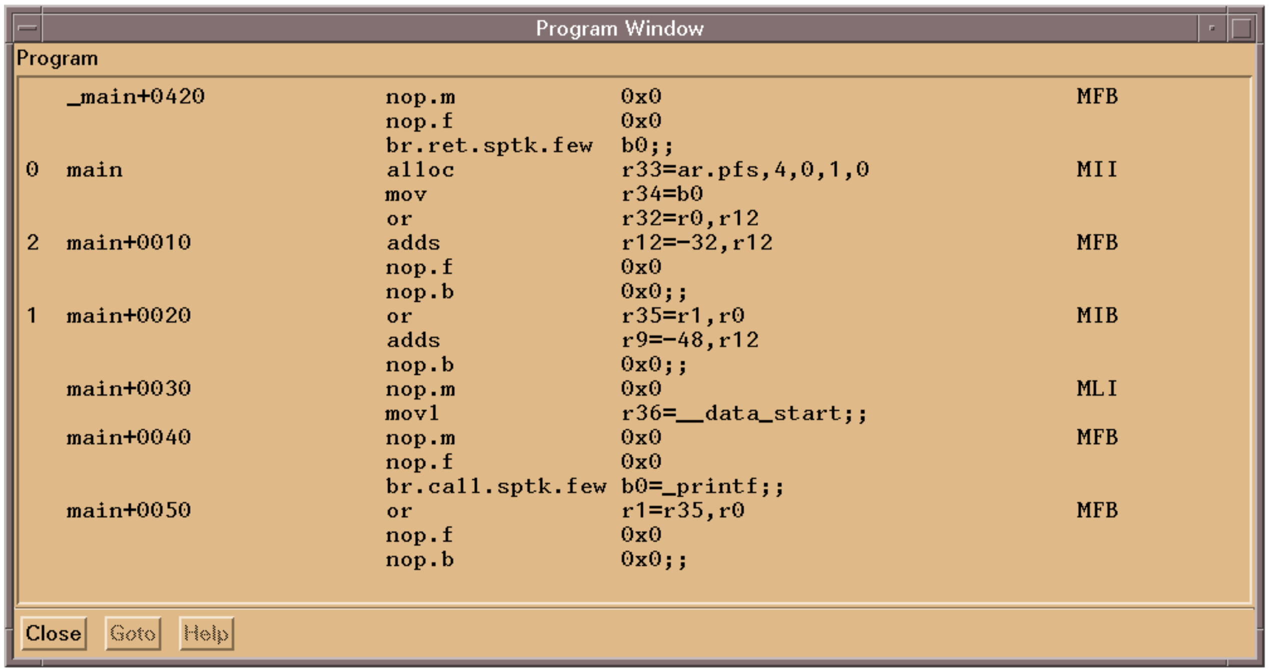
Program breakpoints are "marks" within the executable code of a program that cause simulation to halt when they are encountered in the normal flow of a running program. When simulation stops because of a breakpoint, the instruction pointer (**`ip`**) is pointing to the instruction at which the breakpoint is set (before the instruction is executed) and control is returned to you.

The simulator provides several commands to let you manipulate program breakpoints. These commands are explained in detail below.

Setting Program Breakpoints

To set a breakpoint in IA-64 code, use the **`bs`** command. For IA-32 code, use the **`iabs`** command. If given with no arguments, these commands set a breakpoint at the instruction pointed to by the **`ip`** register. If an address is given following the command, the breakpoint is set at that address. The address must be valid when Ski resumes simulation; Ski will refuse to simulate code if any breakpoints are set at non-existent addresses. You can set breakpoints in system-mode programs using physical or virtual addresses. See "Application-Mode and System-Mode Simulation" on page 66 for information about system-mode programming and "Addresses" on page 55 for information on physical vs. virtual addressing.

Up to ten breakpoints may be set at any one time. They are indicated by the digits "**`0`**" through "**`9`**" in the first column of the program window, as the example in Illustration 56: Three Breakpoints, 0, 2, and 1, Visible in xski's Program Window shows.



```
   _main+0420          nop.m           0x0                       MFB
                       nop.f           0x0
                       br.ret.sptk.few b0;;
0  main                alloc           r33=ar.pfs,4,0,1,0        MII
                       mov             r34=b0
                       or              r32=r0,r12
2  main+0010           adds            r12=-32,r12               MFB
                       nop.f           0x0
                       nop.b           0x0;;
1  main+0020           or              r35=r1,r0                 MIB
                       adds            r9=-48,r12
                       nop.b           0x0;;
   main+0030           nop.m           0x0                       MLI
                       movl            r36=__data_start;;
   main+0040           nop.m           0x0                       MFB
                       nop.f           0x0
                       br.call.sptk.few b0=_printf;;
   main+0050           or              r1=r35,r0                 MFB
                       nop.f           0x0
                       nop.b           0x0;;
```

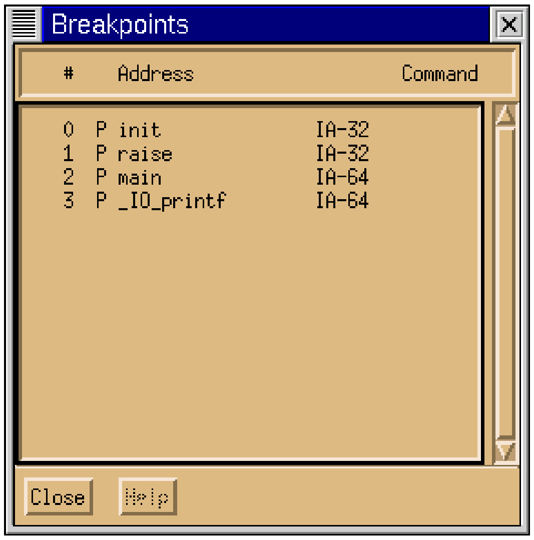
Illustration 56: Three Breakpoints, 0, 2, and 1, Visible in xski's Program Window

Deleting Program Breakpoints

Two commands delete program breakpoints. The **`bd`** command deletes a specified breakpoint. The **`bD`** command deletes all breakpoints currently set.

Listing Program Breakpoints

The **`bl`** command causes a list of currently set program breakpoints to be displayed on the screen, symbolically if possible, as shown in Illustration 57: xski's Breakpoint List Window Showing IA-64 and IA-32 Breakpoints. The first column of the display shows the breakpoint number, for use with the **`bd`** command. The second column displays a "**`P`**" for physically-addressed breakpoints and "**`V`**" for virtually-addressed breakpoints. The column labelled "Address" is, of course, the breakpoint address. In the next column, "**`IA-64`**" indicates a breakpoint in IA-64 code and "**`IA-32`**" indicates a breakpoint in IA-32 code. The "Command" column is currently unused.



```
 #   Address                Command

 0  P init            IA-32
 1  P raise           IA-32
 2  P main            IA-64
 3  P _IO_printf      IA-64
```

Illustration 57: xski's Breakpoint List Window Showing IA-64 and IA-32 Breakpoints

Notes on Program Breakpoints

How Ski Implements Breakpoints

Program breakpoints are implemented by replacing the instruction at the address of each breakpoint with an IA-64 BREAK instruction or an IA-32 INT3 instruction. The replacement is done at the time the program is started or restarted (e.g., with **`cont`**) and the original instructions are replaced when the program halts. Thus, if your program reads the location where a breakpoint is set, it will retrieve the BREAK or INT3 instruction instead. Ski detects if your program attempts to write new data into the breakpoint location and automatically reinstalls the breakpoint after such an update.

You need to tell Ski where to set your IA-64 breakpoints but the IA-64 architecture doesn't provide for addressability of individual instructions. Instead, instructions are bundled. To work around this, Ski "pretends" that the slot 0 instruction of a bundle is in the first four bytes of the bundle's location, the slot 1 instruction is in the second four bytes of the bundle, and the slot 2 instruction is in the third four bytes of the bundle. You can only set breakpoints at these "pretend" locations. For example, setting a breakpoint at "**`main`**", "**`main+1`**", "**`main+2`**", and "**`main+3`**" all result in the breakpoint being set on the first instruction in the bundle at "**`main`**". Similarly, "**`main+5`**", "**`main+6`**", and "**`main+7`**" all correspond to "**`main+4`**", and "**`main+9`**", "**`main+a`**", and "**`main+b`**" all correspond to "**`main+8`**", If you try to set a breakpoint at the remaining bytes in the bundle ("**`main+c`**", "**`main+d`**", "**`main+e`**", and "**`main+f`**" in this example), Ski will generate the error message "**`Illegal slot field in breakpoint address`**". Ski can place IA-32 breakpoints at any byte address. If the breakpoint address doesn't correspond to the beginning of an IA-32 instruction, Ski's behavior is undefined.

Unexpected Breakpoints

The IA-64 breakpoint mechanism uses BREAK.M 0, BREAK.I 0, BREAK.B 0, and BREAK.F 0, and BREAK.X 0 instructions.

These are special cases and executing these instructions will not cause "BREAK instruction trap" interrupts for system-mode programs. The same is true for INT3 instructions in IA-32 code. However, if Ski finds BREAK or INT3 instruction at a location which doesn't correspond to a breakpoint, Ski's behavior depends on whether the program is simulating in application-mode or system-mode. Application-mode programs should never generate, or expect to receive, interrupts. If Ski reaches a BREAK or INT3 instruction in an application-mode program at a location which doesn't correspond to a breakpoint, simulation halts and Ski displays an error message. System-mode IA-64 programs will receive the BREAK interrupt.

Summary of Program Breakpoint Commands

`bs` [*address*]

Sets an IA-64 breakpoint at the specified *address* or, if no *address* is given, at the location pointed to by **`ip`**.

`iabs` [*address*]

Sets an IA-32 breakpoint at the specified *address* or, if no *address* is given, at the location pointed to by **`ip`**.

`bd` *breakpoint_number*

Deletes the breakpoint numbered by *breakpoint_number.*

`bD`

Deletes all breakpoints.

`bl`

Displays a list of currently set breakpoints.

Data Breakpoints

Data breakpoints can be viewed as temporary access restrictions on an area of data. Access of a datum within the specified area causes a running program to halt at the instruction which attempted the access. Control is then returned to the user at command level.

The simulator allows up to ten areas to be specified within which data breakpoints may be set. They may vary in size from one byte to an entire region. Further, the area may be specified to cause a break either only on reads, writes, or on both reads and writes. Several commands apply to the manipulation of these data breakpoints.

Setting Data Breakpoints

The **`dbs`** command sets data breakpoints. The command requires two arguments and accepts an optional third argument. The first argument is the starting address of the area which is associated with the break. The second argument specifies the length of the area (in bytes). The third argument, if present, is the string **`rw`** (default), which indicates that the break is to occur on both reads or writes, **`r`**, which indicates that only reads cause breaks, or **`w`**, which indicates that only writes cause breaks.

Deleting Data Breakpoints

Two commands delete data breakpoints. The **`dbD`** command deletes all data breakpoints currently set. It takes no arguments and requires no verification from the user. The **`dbd`** command deletes the data breakpoint with the number specified by the argument.

Listing Data Breakpoints

The **`dbl`** command causes a list of currently set data breakpoints to be displayed on the screen, symbolically if possible.

Summary of Data Breakpoint Commands

`dbs` *address length* [*type*]

Sets a data breakpoint at the specified *address*. The length of the area (in bytes) is set to *length*. Type is the string **`rw`** (default) specifying breaks on reads or writes, **`r`**, specifying breaks on reads only, or **`w`**, specifying breaks on writes only.

`dbd` *number*

Deletes the data breakpoint numbered by *number.*

`dbD`

Deletes all data breakpoints.

`dbl`

Displays on the screen a list of currently set data breakpoints.

Dumping Registers and Memory to a File

You can dump the registers to a file with the "**`rd`**" command, described in "Register Window Commands" on page 59. You can dump a block of memory into a file in two forms: in hexadecimal or in symbolic disassembled form, corresponding (roughly) to the formats in the Data Window and the Program Window, respectively. The commands to do this are "**`dd`**" and "**`pd`**" and are described in "Data Window Commands" on page 63 and "Program Window Commands" on page 60, respectively.

Saving and Restoring the Simulator State

You may need to interrupt a simulation session and continue it later. For example, you might be tracking down a difficult bug and want to save the state of the simulator just before the bug occurs so you can replay the problem and try different strategies. The **`save`** command saves the state of the currently executing program to a named disk file. Later, you restore the saved file with the **`rest`** command or the **`-rest`** command line flag (described in "Command Line Flags" on page 33).

The **`save`** command saves the state of the simulated IA-64 processor, including the overlaid IA-32 registers, the symbol table for program-defined symbols, and memory. Certain simulator state information, in particular the values of internal variables and window-related information, is not saved. Linux and MS-DOS state information such as open file handles and **`fseek`** pointers is not currently saved; this will probably change, so you should check the release notes.

Summary of Save and Restore Commands

`save` *filename*

Saves an image of the machine state (IA-64 and IA-32) in the specified file.

`rest` *filename*

Restores an image of the machine state (IA-64 and IA-32) from the specified file.

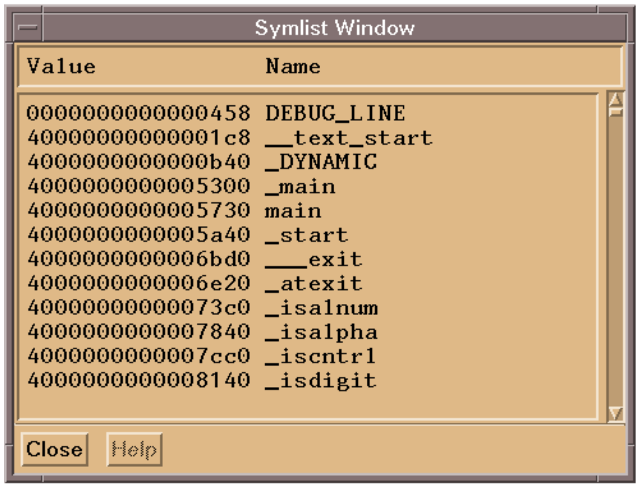
Symbol Table Commands

Ski supports two kinds of symbols: program-defined symbols, which are identifiers provided by a compiler, linker, or human programmer (see "Program-Defined Symbols" on page 56), and internal symbols, which include register names and internal variables (see "Registers" on page 56 and "Internal Variables" on page 56). Ski places program-defined symbols in one symbol table; you can see the contents with the **`symlist`** command. For IA-64 programs, the ELF executable file always contains symbols, regardless of whether you used your compiler's debug symbols flag (typically **`-g`**), unless you stripped the symbols. Internal symbols are stored in a second symbol table along with the register names Ski recognizes, listed in "Register Names" on page 93. The **`isyms`** command displays the contents of this table.

Summary of Symbol Commands

`symlist` [*filename*]

Shows the list of program-defined symbols sorted by ascending address, as seen in Illustration 58: The symlist Output from xski. If *filename* is given, the list is written to the named file, otherwise the list is written to the screen.



```
Value              Name

0000000000000458  DEBUG_LINE
40000000000001c8  __text_start
4000000000000b40  _DYNAMIC
4000000000005300  _main
4000000000005730  main
4000000000005a40  _start
4000000000006bd0  ___exit
4000000000006e20  _atexit
40000000000073c0  _isalnum
4000000000007840  _isalpha
4000000000007cc0  _iscntrl
4000000000008140  _isdigit
```

Illustration 58: The symlist Output from xski

`isyms` [*filename*]

Writes the list of internal variables to *filename* if given, otherwise to the screen.

9 Command Files

The dot (" `.` ") command temporarily redirects command input to the simulator so that input is taken from the file provided as an argument to the command. Into this file (a "command file"), you put commands as if you had typed them from the keyboard. Several commands are specifically applicable to command files and are described below. Command files may be nested; i.e., one command file may invoke another. The maximum nesting depth is operating-system-dependent.

Some syntax rules that apply to keyboard input don't make sense or would be cumbersome in command files. Most notably, in ***ski*** , a shortcut for re-executing the previous command is to hit the enter/return key on an empty line. This rule is removed in command files, so you are free to put in blank lines for readability. You can also indent lines as necessary.

The ability to assign values to registers and memory and the flow control features provide the simulator with a powerful Church-Turing-complete command language; i.e., tasks which can be accomplished in any programming language, subject to memory constraints, can be accomplished in the command language of the simulator. Command files are particularly appropriate for initializing the state of the simulator and for implementing complex facilities on top of Ski's native commands. For example, you can write command files to setup the machine state just before an I/O interrupt, to create sophisticated breakpointing, and to take complex performance measurements.

Initialization File

If you start Ski with a ***-i*** option followed by a filename, the named file will be executed as a command file before the first prompt (see "Command Line Flags" on page 33). This feature is particularly important for ***bski*** , because without a command file to guide it, ***bski*** will only ***run*** your program and then ***quit*** . If you want to do anything else, you need a command file. When you combine the ***-i*** option with Ski's ability to load a program on the command line, you can create a powerful debugging environment. For example, this command line:

```
bski -i test.init -stats -icnt instruction_counts
```

combined with this `test.init` command file:

load ***ia_test 0x26c50***

romload test.com etext test.map

uses the command file `test.init` to load an IA-64 Platform Support File named `ia_test` (filling in Ski's symbol table for program-defined symbols), and then loads the IA-32 system-mode program `test.com` , putting it at the location corresponding to the symbol " ***etext*** " in `ia_test` . The command file finishes and ***bski*** automatically executes a ***run*** command followed by a ***quit*** command. To start the run, the `ia_test` program receives 0x26c50 as its argv[1] value. This corresponds to the value of the symbol " ***etext*** " and tells `ia_test` where `test.com` was loaded. The IA-64 program completes its initialization and transfers control to the IA-32 program, setting the *psr.is* bit appropriately. When the IA-32 program completes, ***bski*** prints out end-of-run performance statistics and writes an instruction frequency count to the file `instruction_counts` .

Labels and Control Flow in Command Files

Command files are useful as macro sequences of simple commands and, more interestingly, to create small programs that do useful things for you: create formatted displays of data structures, create complete breakpoints, and gather run-time statistics, for example. Two commands provide the ability to change the flow of control in a command file: ***goto*** and ***if*** .

The *goto* Command and Labels

A label identifies a particular line in a command file. Labels are defined in "Labels" on page 57. No other text can appear on a label line.

The ***goto*** command takes a label as an argument and searches the command file for a line with that label. Execution resumes at the first command after the label. There is no good reason to have a label appear more than once in a particular command file; if this condition occurs, only the first occurrence of the label will be noticed and all subsequent occurrences will be ignored. The ***goto*** command can only be executed in a command file. A ***goto*** may go forward or backward. An example of using ***goto*** and a label is:

```
loop:
    ... other commands ...
    goto loop
```

The *if* Command

The ***if*** command allows for conditional execution. If the expression following the command evaluates to nonzero, the remainder of the line is executed; otherwise it is ignored. (No spaces are allowed in the expression.) For example, this command file steps through a IA-64 application-mode simulation 600 instructions at a time until the program finishes, printing the contents of general register 32 after each step:

```
loop:
    step 600
    eval r32
    if !$exited$ goto loop
quit
```

If a colon surrounded by spaces is present on the line, the remainder of the line is taken to be an "else" clause. That is, if the ***if*** expression evaluates to nonzero, the remainder of the line up to but not including the colon is executed; if zero, that part of the line is ignored and execution continues immediately following the colon. For example, the following command file line sets the contents of general register 4 to zero or one depending on whether the sum of the contents of general integer 1 and 2 are equal to the contents of the location pointed to by general register 13.

```
if (r1+r2)==*r13 = r4 0 : = r4 1
```

Comments in Command Files

To document command files, you can add comments any characters following an octothorpe (also called a "pound sign" or "sharp sign" and shown, typically, as " *#* ") are ignored by the command interpreter. Examples of comments are in Table 6: An Example Command File to Compute Fibonacci Numbers

An Example Command File

Command files are easy to write. The command file in Table 6: An Example Command File to Compute Fibonacci Numbers for computing Fibonacci numbers was written in less than five minutes and most of that time was spent making the comments correct.

```
# Compute and print Fibonacci numbers from 1 to 50.
# Initialize variables
= r10 1 # Hold n-2  th value
= r11 1 # Hold n-1  th value
= r12 0 # Temporary holding place for n-1  th value
= r13 0 # Loop counter
# Print out first two Fibonacci numbers (initial values of r10 & r11)
eval r10
eval r11
# Calculate and print the rest of the numbers. The last line has the
# stopping value of the loop index. (This is a simple counting loop.)
loop:
     eval +r11 #   +   makes an expression: decimal and hex printing
     = r12 r11 # Compute n  th Fibonacci term
     = r11 r11+r10
     = r10 r12
     = r13 r13+1 # Increment loop counter
     if r13<0d50 goto loop # Loop again?
```

Table 6: An Example Command File to Compute Fibonacci Numbers

Summary of Command File Commands

. *filename*

Executes commands in the given command file. The file is opened and its contents are executed as if they were entered from the keyboard. When the contents of a non-nested command file are exhausted, ***xski*** and ***ski*** resume keyboard input and ***bski*** executes a ***run*** command followed by a ***quit*** command. When a nested command file is exhausted, control returns to the next-higher-level command file.

if *expression-without-spaces true-command*

if *expression-without-spaces true-command* **:** *false-command*

In the first form, causes the rest of the line to be ignored if *expression-without-spaces* evaluates to zero. Otherwise, *true-command* is executed. In the second form, if *expression-without-spaces* evaluates to nonzero, the *true-command* is executed. Otherwise, the *false-command* is executed.

The ***if*** command may be executed from the keyboard. In combination with ***xski*** 's Command History (see "The xski Main Window" on page 48) or ***ski*** 's command repetition mechanism (see "The ski Command Window" on page 49), this can be quite powerful.

goto *label*

In a command file (only), causes execution to continue following the first line in the file which contains the *label* . Goto's may be forward or backward.

# comment

The " ***#*** " and all characters following it until the next newline are ignored.

label **:**

The colon (" ***:*** ") command marks a ***goto*** label. All characters following the " ***:*** " and preceding the next newline are ignored.

10 Command Reference

In the command descriptions that follow, **`this face`** indicates literal text you should type, *this face* indicates operand text you should modify, [bracketed text] indicates text you may choose to omit (never type the brackets), and the + symbol indicates items you may repeat. The syntax of the command language is described in "Command Language" on page 52.

The order in which commands appear here is the order in which they may be abbreviated: any command may be abbreviated to as few letters as are needed to distinguish it from all commands preceding it in the list below. For example, the "**`step`**" command may be spelled out in full or abbreviated as "**`ste`**", "**`st`**", or "**`s`**". The "**`save`**" command can be spelled out in full or abbreviated as "**`sav`**" or "**`sa`**". It can't be abbreviated as "**`s`**" because it follows "**`step`**" in the list below.

`.` *filename*

Execute commands from the command file specified by *filename*. The file is opened and its contents are executed as if they were entered from the keyboard. When the contents are exhausted, ***ski*** and ***xski*** resume reading commands from the keyboard. ***bski***, on the other hand, executes a **`run`** command and then a **`quit`** command (unless, of course, the command file already executed a **`quit`** command). Command files can be nested to a reasonable level. See "Command Files" on page 84.

`#` *comment*

Comments may be used to help document the design and implementation of command files. A comment is any part of a line following an octothorpe ("**`#`**"). The octothorpe and everything following it on the line are ignored. See "Comments in Command Files" on page 85.

label:

Labels are targets for **`goto`** commands and are valid only in command files. See "Labels and Control Flow in Command Files" on page 84

`=` *register_name value*

Assign *value* to the register specified by *register_name*. Unless a modifying prefix such as **`0d`**, **`0o`**, or **`0b`** is used, *value* will be treated as a hexadecimal number. See "Changing Registers and Memory with Assignment Commands" on page 75. The register names recognized by Ski are listed in "Register Names" on page 93.

`=1` *address value+*

`=2` *address value+*

`=4` *address value+*

`=8` *address value+*

The *value* is assigned to the specified location in memory. The old value at the location is lost. The location may be on any allocated page, including instruction pages. Multiple values separated by whitespace may be supplied; if so, they will be assigned to sequential memory addresses. Unless a modifying prefix such as **`0d`**, **`0o`**, or **`0b`** is used, *value* will be treated as a hexadecimal number. See "Changing Registers and Memory with Assignment Commands" on page 75.

The **`=1`** command truncates any extra high-order bytes of the *value* to make a single byte. The **`=2`** command truncates or pads (with zero) the high order bytes of the *value* as necessary to make a two-byte quantity. Similarly, the **`=4`** and **`=8`** commands truncate or pad high order bytes to make four- and eight-byte quantities, respectively. The *psr.be* bit controls whether the data is stored in big-endian or little-endian format.

`=s` *address string_without_spaces*

The *string_without_spaces* is assigned to memory locations starting at the location specified by *address*. A null byte is added to the end of the string automatically. The old value at the location is lost. The location may be on any allocated page, including instruction pages. Multiple values may not be supplied. The string may not contain spaces and quoting it is not a workaround. See "Changing Registers and Memory with Assignment Commands" on page 75.

`bs` **[***address***]**

Set breakpoint at the location specified by the current value of **`ip`** or at the specified *address*. (IA-64 code only). See "Setting Program Breakpoints" on page 79.

`bD`

Delete all breakpoints. See "Deleting Program Breakpoints" on page 79.

`bd` *breakpoint_number*

Delete breakpoint *breakpoint_number*. Use the **`bl`** command to get a list of all breakpoints and their corresponding numbers. See "Deleting Program Breakpoints" on page 79.

`bl`

Display a list of current breakpoints. See "Listing Program Breakpoints" on page 79.

`cont`

Continue simulating the program from the current **`ip`** value. Most commonly used after the simulator stops at a breakpoint. See "Program Execution" on page 70.

`dj` **[***address***]**

Jump the Data Window display to the specified *address*. If no *address* is given, the window display changes to the previous location, providing a handy way to swap the display between two different parts of memory. See "Data Window Commands" on page 63.

`db` **[***count***]**

Move the Data Window backward *count* lines or one windowful if no *count* is given. See "Data Window Commands" on page 63.

`dbndl`

Display the Data Window contents as instruction bundles. See "Data Window Commands" on page 63.

`dbs` *address length* **`[r|w|rw]`**

Set data breakpoint covering the memory area of *length* bytes starting at *address*. See "Setting Program Breakpoints" on page 79

dbD

Delete all data breakpoints. See "Deleting Program Breakpoints" on page 79.

dbd *breakpoint_number*

Delete data breakpoint *breakpoint_number*. Use the **`dbl`** command to get a list of all breakpoints and their corresponding numbers. See "Deleting Program Breakpoints" on page 79.

dbl

Display a list of current data breakpoints. See "Listing Program Breakpoints" on page 79

`dd` *starting_address ending_address* **[***filename***]**

Dump memory contents to the screen or to the file given by *filename*. The range dumped is between *starting_address* and *ending_address* inclusive. The dump is formatted as hexadecimal. See "Data Window Commands" on page 63.

`df` **[***count***]**

Move the Data Window forward *count* lines or one windowful if no *count* is given. See "Data Window Commands" on page 63.

`dh`

Display Data Window contents in hexadecimal format. See "Data Window Commands" on page 63.

`eval` *expression_without_spaces+*

Evaluate one or more *expression_without_spaces* and print the result in an appropriate format, typically hexadecimal, and/or decimal, or symbolically. An *expression_without_spaces* can include numbers, registers, internal variables, program-defined symbols, operators, and parentheses for grouping. See "Evaluating Formulas and Formatting Data" on page 78.

`fr`

ski: Show the floating point registers in the Register Window. See "Register Window Commands" on page 59.

xski: Toggle the display of the floating point registers pane. See "Register Window Commands" on page 59.

`goto` *label*

Causes execution to continue following the first line in the file which contains the *label*. Goto's may be forward or backward. Goto's are valid only in command files. See "The goto Command and Labels" on page 84.

`gr`

ski: Show the general registers in the Register Window. See "Register Window Commands" on page 59.

xski: Toggle the display of the general registers pane. See "Register Window Commands" on page 59.

`help` **[***command_name***]**

Display a list of the commands Ski recognizes, or, if a *command_name* is specified, a syntax description for that command. See "Command Entry" on page 52.

`iar`

ski: Show the IA-32 registers in the Register Window. See "Register Window Commands" on page 59.

xski: Toggle the display of the IA-32 registers pane. See "Register Window Commands" on page 59.

`iabs` **[***address***]**

Set IA-32 breakpoint at *address* or at the current value of `ip` if *address* is omitted. (IA-32 code only) See "Setting Program Breakpoints" on page 79.

`iaload` *filename address* **[***mapfile* **[***args***]**+**]**

Prepare for IA-32 application-mode simulation: Load an IA-32 executable file (`.com` or `.exe`) and prepare to pass the program *args* using the MS-DOS command line parameter mechanism. *address* specifies where to load the program. *mapfile* provides Ski with the mapping between program-defined symbols and their addresses and must specify an ASCII text file exactly compatible with mapfiles produced by the Microsoft "ML" linker. See "How to Load a Program" on page 69.

`if` *expression_without_spaces true_command* **[`:`** *false_command***]**

Execute *true_command* if the *expression_without_spaces* evaluates to a non-zero value, *false_command* if it evaluates to zero. See "The if Command" on page 85.

`isyms` **[***filename***]**

Write internal symbols to the screen or to the file given by *filename*. See "Symbol Table Commands" on page 82.

`load` *filename* **[***args***]**+

Prepare for IA-64 application-mode simulation: Load the IA-64 ELF executable program file given by *filename* and prepare to pass the program *args* using the C language argc/argv parameter mechanism. See "How to Load a Program" on page 69.

`pj` **[***address***]**

Jump the Program Window display to the specified *address*. If no *address* is given, the window display changes to the previous location, providing a handy way to swap the display between two different parts of the program. See "Summary of Program Loading Commands" on page 69.

`pa`

Display the program in assembly language format only. (IA-64 only) See "Summary of Program Loading Commands" on page 69.

`pb` **[***count***]**

Move the Program Window backward *count* IA-64 bundles or IA-32 instructions, or one windowful less one bundle or instruction if no *count* is given. See "Summary of Program Loading Commands" on page 69.

`pd` *starting_address ending_address* **[***filename***]**

Dump memory to the screen or to the file given by *filename*. The range dumped is between *starting_address* and *ending_address* inclusive. The dump is formatted as disassembled instructions, without source code. See "Summary of Program Loading Commands" on page 69..

`pf` **[***count***]**

Move the Program Window forward *count* IA-64 bundles or IA-32 instructions, or one windowful less one bundle or instruction if no *count* is given. See "Summary of Program Loading Commands" on page 69.

`pm`

Display an IA-64 program in both source and assembly form. The source code file must be available to the simulator in the location recorded in the executable file when this command is issued. The source code is displayed for convenience; it cannot be modified or interacted with. Mixed display may not be useful if a high degree of optimization was applied during compilation. (IA-64 only) See "Summary of Program Loading Commands" on page 69.

`quit` **[***return_value_for_shell***]**

Quit the simulator. If no *return_value_for_shell* is given, a zero value is returned to the shell. Return values are useful in shell script programming. See "Quitting Ski" on page 35.

`run`

Simulate the program. Using the C language argc/argv mechanism, Ski will pass the program a copy of the command line parameters Ski received on its command line, or, if specified, the command line parameters provided with the **`load`** and **`iaload`** commands. See "Program Execution" on page 70.

`rest` *filename*

Restore the state of a simulated processor from the specified file and prepare to resume a suspended simulation. Only the registers and memory of the simulated processor are restored; state information private to the simulator such as cycle counts is not restored. See "Saving and Restoring the Simulator State" on page 82.

`rf` **[***count***]**

Moves the Register Window "forward" (scroll down) through the currently-displayed register set. The Register Window is scrolled *count* lines. If *count* is omitted, the Register Window scrolls down one windowful less one line, i.e. the last line of the old window is displayed as the first line of the new window. (***ski*** only) See "ski Register Window Commands" on page 59.

`rb` **[***count***]**

Moves the Register Window "backward" (scroll up) through the currently-displayed register set. The Register Window is scrolled *count* lines. If *count* is omitted, the Register Window scrolls up one windowful less one line, i.e. the first line of the old window is displayed as the last line of the new window. (***ski*** only) See "ski Register Window Commands" on page 59.

`rd` **[***filename***]**

Dump the Register Window to the screen or to the file given by *filename*. See "Register Window Commands" on page 59.

`romload` *filename address* **[***mapfile***]**

Load an MS-DOS `.com`-format file for IA-64, IA-32, or mixed system-mode simulation. *address* specifies where to load the program. *mapfile* provides Ski with the mapping between program-defined symbols and their addresses and must specify an ASCII text file exactly compatible with mapfiles produced by the Microsoft "ML" linker. See "How to Load a Program" on page 69.

`step` **[***count*

Execute *count* instructions or, if no *count* is specified, one instruction. See "Program Execution" on page 70.

`step until` *expression_without_spaces*

Execute instructions until the *expression_without_spaces* has a non-zero value. See "Program Execution" on page 70.

`save` *filename*

Save the state of a simulated processor in the file given by *filename*. Only the registers and memory of the simulated processor are saved; state information private to the simulator such as cycle counts is not saved. See "Saving and Restoring the Simulator State" on page 82.

`sdt`

Show the Data Translation Lookaside Buffer (DTLB) (system-mode only). See "Symbol Table Commands" on page 82.

`sit`

Show Instruction Translation Lookaside Buffer (ITLB) (system-mode only). See "Symbol Table Commands" on page 82.

`sr`

ski: Show the system registers (Control Registers, Region Registers, Debug Registers, Protection Key Registers, Data Breakpoint Registers, Instruction Breakpoint Registers, Performance Monitor Configuration Registers, Performance Monitor Data Registers) in the Register Window. See "Register Window Commands" on page 59.

xski: Toggle the display of the system registers pane. See "Register Window Commands" on page 59.

`symlist` **[***filename***]**

Write program-defined symbols to the screen or to the file given by *filename*. See "Symbol Table Commands" on page 82.

`ur`

ski: Show the user registers (Predicate Registers, Branch Registers, Application Registers, Instruction Pointer, User Mask) in the Register Window. See "Register Window Commands" on page 59.

xski: Toggle the display of the user registers pane. See "Register Window Commands" on page 59.

11 Register Names

IA-64 registers are fully described in other documents. This chapter provides a list for convenience only. The register names are documented here as recognized by Ski and, in a few cases, don't exactly match the names in other documents due to program limitations. For example, the floating point registers must be accessed in three pieces: the mantissa part, the sign part, and the (biased) exponent part. Similarly, the "Not a Thing" bits of the various registers are separate entities for Ski. Individual bits of complex registers such as the **`psr`** are documented here as well, corresponding to the names by which Ski recognizes them.

IA-64 Registers

al, ah, ax, eax	IA-32 Registers: al and ah are byte-wide, ax is al and ah taken together as two bytes, eax is four bytes wide with ax as the two least significant bytes.
ar0 - ar127	IA-64 Application Registers
b0 - b7	IA-64 Branch Registers
bl, bh, bx, ebx	IA-32 Registers: bl and bh are byte-wide, bx is bl and bh taken together as two bytes, ebx is four bytes wide with bx as the two least significant bytes.
bp, ebp	IA-32 Base Pointers: bp is two bytes wide, ebp is four bytes wide with bp as the two least significant bytes.
bsp	IA-64 Register Save Engine (RSE) Backing Store Pointer Register
bspst	IA-64 Register Save Engine (RSE) Backing Store Pointer Register for memory stores
ccv	IA-64 Compare and Exchange Value Register
cl, ch, cx, ecx	IA-32 Registers: cl and ch are byte-wide, cx is cl and ch taken together as two bytes, ecx is four bytes wide with cx as the two least significant bytes.
cmcv	IA-64 Corrected Machine Check Vector Register
cr0 - cr127	IA-64 Control Registers
cs	IA-32 Code Segment Register
csd	IA-32 Code Segment Register Descriptor
dbr0 - dbr15	IA-64 Data Breakpoint Registers
dcr	IA-64 Default Control Register
dl, dh, dx, edx	IA-32 Registers: dl and dh are byte-wide, dx is dl and dh taken together as two bytes, edx is four bytes wide with dx as the two least significant bytes.
di, edi	IA-32 Arithmetic Registers: di is two bytes wide, edi is four bytes wide with di as the two least significant

	bytes.
ds	IA-32 Data Segment Register
dsd	IA-32 Data Segment Register Descriptor
ec	IA-64 Epilog Count Register
eflags	IA-32 Flags Register
eflags.ac	IA-32 Alignment Check bit
eflags.af	IA-32 Auxiliary Carry Flag bit, also called the IA-32 Adjust Flag bit
eflags.be	IA-32 Below Equal Flag bit
eflags.cf	IA-32 Carry Flag bit
eflags.df	IA-32 Direction Flag bit
eflags.id	IA-32 ID Flag bit
eflags.if	IA-32 Interruption Flag bit
eflags.iopl	IA-32 I/O Privilege Level bit
eflags.le	IA-32 Less Equal Flag bit
eflags.lt	IA-32 Less Than Flag bit
eflags.nt	IA-32 Nested Task bit
eflags.of	IA-32 Overflow Flag bit
eflags.pf	IA-32 Parity Flag bit
eflags.rf	IA-32 Resume Flag bit
eflags.sf	IA-32 Sign Flag bit
eflags.tf	IA-32 Trap Flag bit
eflags.vm	IA-32 Virtual 8086 Mode bit
eflags.zf	IA-32 Zero Flag bit
eoi	IA-64 End of Interrupt
es	IA-32 "Extra" Segment Register
esd	IA-32 "Extra" Segment Register Descriptor

esp	IA-32 four byte Stack Pointer; see "iasp" below
f0.e, f1.e, & f127.e	IA-64 Floating-point Register exponent parts
f0.m, f1.m, & f127.m	IA-64 Floating-point Register mantissa parts
f0.s, f1.s, & f127.s	IA-64 Floating-point Register sign bits
fpsr	IA-64 Floating-point Status Register
fpsr.traps	IA-64 FPSR Trap Bits
fpsr.sf0	IA-64 FPSR Status Field 0
fpsr.sf0.ftz	IA-64 FPSR Status Field 0, Flush-to-Zero mode bit.
fpsr.sf0.wre	IA-64 FPSR Status Field 0, Widest range exponent mode bit
fpsr.sf0.pc	IA-64 FPSR Status Field 0, Precision control bits
fpsr.sf0.rc	IA-64 FPSR Status Field 0, Rounding control bits
fpsr.sf0.v	IA-64 FPSR Status Field 0, IEEE Invalid Operation status bit
fpsr.sf0.d	IA-64 FPSR Status Field 0, Denormal/Unnormal Operand status bit
fpsr.sf0.z	IA-64 FPSR Status Field 0, IEEE Zero Divide status bit
fpsr.sf0.o	IA-64 FPSR Status Field 0, IEEE Overflow status bit
fpsr.sf0.u	IA-64 FPSR Status Field 0, IEEE Underflow status bit
fpsr.sf0.i	IA-64 FPSR Status Field 0, IEEE Inexact status bit
fpsr.sf1	IA-64 FPSR Status Field 1
fpsr.sf2	IA-64 FPSR Status Field 2
fpsr.sf2.pc	IA-64 FPSR Status Field 2, Precision control bits
fpsr.sf2.rc	IA-64 FPSR Status Field 2, Rounding control bits
fpsr.sf2.v	IA-64 FPSR Status Field 2, IEEE Invalid Operation status bit
fpsr.sf2.d	IA-64 FPSR Status Field 2, Denormal/Unnormal Operand status bit
fpsr.sf2.z	IA-64 FPSR Status Field 2, IEEE Zero Divide status bit

fpsr.sf2.o	IA-64 FPSR Status Field 2, IEEE Overflow status bit
fpsr.sf2.u	IA-64 FPSR Status Field 2, IEEE Underflow status bit
fpsr.sf2.i	IA-64 FPSR Status Field 2, IEEE Inexact status bit
fpsr.sf3	IA-64 FPSR Status Field 3
fs	IA-32 additional extra Segment Register
fsd	IA-32 additional extra Segment Register Descriptor
gdtd	IA-32 Global Descriptor Table Descriptor
gp	IA-64 Global Pointer, a synonym for r1
gp.nat	IA-64 Global Pointer Not-a-Thing bit, a synonym for r1.nat
gs	IA-32 additional extra Segment Register
gsd	IA-32 additional extra Segment Register Descriptor
iasp, esp	IA-32 Stack Pointer: iasp is two bytes wide, esp is four bytes wide with iasp as the two least significant bytes. (The x86 mnemonic for the iasp register is "sp" but that conflicts with the IA-64 Stack Pointer of the same name, hence the name change for IA-32.)
ibr0 - ibr15	IA-64 Instruction Breakpoint Registers
ifa	IA-64 Interruption Faulting Address Register
ifs	IA-64 Interruption Function State
iha	IA-64 Interruption Hash Address
iim	IA-64 Interruption Immediate Register
iip	IA-64 Interruption Instruction Bundle Pointer
iipa	IA-64 Interruption Instruction Previous Address
ip	IA-64 Instruction Pointer
ipsr	IA-64 Interruption Processor Status Register
irr0-irr3	IA-64 Interrupt Request Registers
isr	IA-64 Interruption Status Register
itc	IA-64 Interval Time Counter
itir	IA-64 Interruption TLB Insertion Register

itm	IA-64 Interval Timer Match Register
itv	IA-64 Interval Timer Vector
iva	IA-64 Interrupt Vector Address
ivr	IA-64 Interrupt Vector Register
k0 - k7	IA-64 Kernel Registers
lc	IA-64 Loop Count Register
ldt	IA-32 Local Descriptor Table
ldtd	IA-32 Local Descriptor Table Descriptor
lid	IA-64 Local Interrupt ID
lrr0-lrr1	IA-64 Local Redirection Registers
p0 - p63	IA-64 Predicate Registers
pfs	IA-64 Previous Function State
pkr0 - pkr15	IA-64 Protection Key Registers
pmc0 - pmc15	IA-64 Performance Monitor Configuration Registers
pmd0 - pmd15	IA-64 Performance Monitor Data Registers
pmv	IA-64 Performance Monitoring Vector
psr	IA-64 Processor Status Register
psr.ac	IA-64 PSR Alignment Check bit
psr.be	IA-64 PSR Big-Endian bit
psr.bn	IA-64 PSR Register Bank bit
psr.cpl	IA-64 PSR Current Privilege Level
psr.da	IA-64 PSR Disable Access and Dirty-bit faults bit
psr.db	IA-64 PSR Debug Breakpoint fault bit
psr.dd	IA-64 PSR Data Debug fault disable bit
psr.dfh	IA-64 PSR Disabled Floating-point High bit
psr.dfl	IA-64 PSR Disabled Floating-point Low bit

psr.di	IA-64 PSR Disable Instruction set transition bit
psr.dt	IA-64 PSR Data address Translation bit
psr.ed	IA-64 PSR Exception Deferral bit
psr.i	IA-64 PSR Interrupt unmask bit
psr.ic	IA-64 PSR Interrupt Collection bit
psr.id	IA-64 PSR Instruction Debug fault disable bit
psr.is	IA-64 PSR Instruction Set bit
psr.it	IA-64 PSR Instruction address Translation bit
psr.lp	IA-64 PSR Lower Privilege transfer trap bit
psr.mfh	IA-64 PSR Floating-point High modified bit
psr.mfl	IA-64 PSR Floating-point Low modified bit
psr.mc	IA-64 PSR Machine Check abort mask bit
psr.pk	IA-64 PSR Protection Key enable bit
psr.pp	IA-64 PSR Privileged Performance monitor enable bit
psr.ri	IA-64 PSR Restart Instruction slot number
psr.rt	IA-64 PSR Register stack Translation bit
psr.si	IA-64 PSR Secure Interval timer bit
psr.sp	IA-64 PSR Secure Performance monitors bit
psr.ss	IA-64 PSR Single Step enable bit
psr.tb	IA-64 PSR Taken Branch trap bit
psr.um	IA-64 PSR User Mask bits
psr.up	IA-64 PSR User Performance monitor enable bit
pta	IA-64 Page Table Address
r0, r1, & r127	IA-64 General Registers
r0.nat, & r127.nat	IA-64 General Register Not-a-Thing bits

rnat	IA-64 Register Save Engine (RSE) Not-a-Thing (NaT) Collection Register
rp	IA-64 Return Pointer, a synonym for b0
rr0 - rr7	IA-64 Region Registers
rrbf	IA-64 CFM Register Rename Base for floating-point registers
rrbg	IA-64 CFM Register Rename Base for general registers
rrbp	IA-64 CFM Register Rename Base for predicate registers
rsc	IA-64 Register Stack Configuration Register
si, esi	IA-32 Arithmetic Registers: si is two bytes wide, esi is four bytes wide with si as the two least significant bytes.
sof	IA-64 CFM Size of Stack frame
sol	IA-64 CFM Size of Locals Portion of Stack frame
sor	IA-64 CFM Size of Rotating Portion of Stack frame
sp	IA-64 Stack Pointer, a synonym for r12. For the IA-32 equivalent of the x86 "sp" register, see the description of "iasp" above.
sp.nat	IA-64 Stack Pointer Not-a-Thing bit, a synonym for r12.nat.
ss	IA-32 Stack Segment Register
ssd	IA-32 Stack Segment Register Descriptor
tpr	IA-64 Task Priority Register
unat	IA-64 User Not-a-Thing (NaT) Collection Register

12 Internal Variable Names

Ski has one combined symbol table for registers and internal variables. (See "Registers" on page 56 and "Internal Variables" on page 56.) A separate symbol table describes program-defined symbols.

Internal Variables

$cycles$	Number of "virtual cycles" simulated.
$exited$	The value 0 until the simulated program exits. Then the variable takes the value 1.
$heap$	The address of the bottom of the simulated heap.
$insts$	The number of instructions simulated so far.

13 Simulator Status and Error Messages

The following is a description of some of the status and error messages which can be produced by the simulator. "Fault" and "Trap" messages are usually the result of a program trying to do something that, under Linux, would cause a signal to be generated.

The "%" constructs are printf() substitutions. Where "%s" appears, a string will be substituted in the error message at runtime. Where "%llx" appears, a 64-bit hexadecimal integer will be substituted in the error message at runtime. See the printf() man page for more information on % substitutions.

All breakpoints deleted

You executed the **`bD`** command. Ski is confirming that it has deleted all the breakpoints. This is a status message, not an error message. See See "Deleting Program Breakpoints" on page 79.

All breakpoints in use

You tried to set a breakpoint but all ten are in use. Use the **`bl`** command to list them and then the **`bd`** or **`bD`** commands to free up some for you to use. See "Setting Program Breakpoints" on page 79.

Assignment failed

You tried to use the **`=1`**, **`=2`**, **`=4`**, **`=8`**, or **`=s`** commands to write data to an invalid location. Ski creates new pages of memory when the simulated program needs them; Ski will not create new pages in response to the assignment commands. See "Changing Registers and Memory with Assignment Commands" on page 75.

Bad breakpoint number. (Use 0-9)

You tried to specify a breakpoint but used an invalid specifier. There are ten breakpoints, numbered 0 through 9. See "Deleting Program Breakpoints" on page 79.

Break instruction fault

A non-Ski-breakpoint BREAK instruction was executed. One possible cause is a wild branch to page with all zeroes. This can only happen for application-mode programs; system-mode programs handle this fault through the interruption mechanism. See "How Ski Implements Breakpoints" on page 80 and "Interruptions" on page 71.

Breakpoint already set at that location

You tried to set a breakpoint at an address where there already is a breakpoint. Your request is ignored; Ski will not set two breakpoints at one address. See "Setting Program Breakpoints" on page 79.

Breakpoint #%d at %s (%s) deleted

You used the **`bd`** command to delete a specific breakpoint. Ski is confirming that it has deleted the breakpoint. This is a status message, not an error message. See "Deleting Program Breakpoints" on page 79.

Breakpoint (IA-64) at %s

An IA-64 breakpoint has been reached. This is a status message, not an error message. See "Program Breakpoints" on page 78.

Breakpoint (IA-32) at %s

An IA-32 breakpoint has been reached. This is a status message, not an error message. See "Setting Program

Breakpoints" on page 79.

Breakpoint #%d wasn't set

You used the **`bd`** command to delete a specific breakpoint but that breakpoint doesn't exist. Did you specify the right breakpoint? Use the **`bl`** command to list the breakpoints. See "Deleting Program Breakpoints" on page 79 and perhaps "Listing Program Breakpoints" on page 79.

Cannot access registers outside current frame

You tried to use the **`=`** command to assign a new value to a register that isn't in the set of registers currently visible to your program. The only registers for which this can occur are the General Registers (**`gr`**) and their NaT bits. Ski faithfully implements IA-64 register stacking and rotation. Look at the most recent ALLOC instruction.

Cannot open file %s (%s) for %s

This generic error message indicates that Ski tried to open a file and failed. The first %s field is replaced with the filename you provided, the second %s field is replaced with the filename Ski tried to use after tilde expansion, and the third %s field is replaced with the mode Ski tried to use, either "**`reading`**", "**`writing`**", or "**`appending`**". Check that you typed the filename correctly and that the directories you specified are accessible. Is there a permissions problem or a network failure, perhaps? See "Filenames" on page 57.

Construct DWARF image: can't find .debug_info section

You told Ski to load a program. Ski couldn't find the part of the executable file containing source code line number information. As a result, Ski won't be able to show source code in the Program Window. See "Program Window Commands" on page 60.

Could not open %s for reading

You told Ski to load a program but Ski couldn't open the file you specified. Perhaps you specified a file that is doesn't exist or a pathname that includes non-existent or inaccessible directories? See "Program Loading" on page 68.

couldn't find label %s

A command file tried to use the **`goto`** command but Ski can't find the label to which the **`goto`** refers. The %s field is replaced with the label. Perhaps the label is spelled incorrectly? See "The goto Command and Labels" on page 84.

Couldn't open file %s'. Was ski started in the right directory?

Ski loaded a program to simulate, per your request, and tried to access source code pointed to by that program. But, for some reason, Ski couldn't open the specified file. This can happen, for example, if files have been moved after compilation. See "Program Window Commands" on page 60.

Couldn't open instruction count file

You started ***bski*** with the **`-icnt`** option but ***bski*** couldn't open the file you specified. Perhaps you specified a file that is write-protected or a pathname that includes non-existent or inaccessible directories? See "Using bski for Batch Simulations" on page 31 and "Command Line Flags" on page 33.

Data larger than a %s. Truncated to 0x%llx

You used the **`=`**, **`=1`**, **`=2`**, **`=4`**, or **`=8`** commands to write data to a register or to memory. You provided more data than would fit, so Ski truncated the excess most significant part away and used the least significant part. The %s field on the left is how many bytes Ski needed. The %llx field on the right is the value after truncation. See "Changing Registers and Memory with Assignment Commands" on page 75.

Error reading %s' line: %d

Ski tried to display the source code corresponding to an IA-64 program you loaded. For some reason, it failed to read a line from the file represented by the %s field, at the line number represented by the %d field. Perhaps the file permissions are wrong or a remote file has suddenly become inaccessible? See "The Program Window" on page 41 and the discussion of the **`pm`** command in "Program Window Commands" on page 60.

Error: unrecognized restore file tag: %s

You are trying to restore a saved simulator state and either the save file is corrupt or Ski is broken. See "Saving and Restoring the Simulator State" on page 82.

Expression aligned to (mod %lld) boundary

You tried to assign an address to a register that requires an address on a specific boundary, but the address you specified isn't on that boundary. Ski has adjusted the address for you, but you should check to make sure the adjustment matches your intent. See "Changing Registers and Memory with Assignment Commands" on page 75.

Expression > 47

You tried to assign a value greater than 47 to the **`rrbp`** register.

Expression > 95

You tried to assign a value greater than 95 to the **`rrbf`** or **`rrbf`** register.

File size > Memory size

You tried to load an IA-64 program but the library Ski uses to parse ELF files can't make sense of the file. Are you sure it's an IA-64 program and not an IA-32 program, an object file, or something completely different? See "Program Loading" on page 68.

Following values could not be assigned:

You supplied multiple values in an **`=1`**, **`=2`**, **`=4`**, or **`=8`** command. Some of the values overflowed on to the next page of memory but that page hasn't been allocated. Ski creates new pages of memory when the simulated program needs them; Ski will not create new pages in response to assignment commands. See "Changing Registers and Memory with Assignment Commands" on page 75.

FP exception fault

An IA-64 application-mode program attempted to execute a floating point operation that doesn't make sense, such as divide by zero or square root of a negative number. This can only happen for IA-64 application-mode programs; IA-64 system-mode programs handle this fault through the interruption mechanism. See "Program Simulation" on page 66 and "Interruptions" on page 71.

FP exception trap

An IA-64 application-mode program caused a floating-point trap. This trap, like all traps, stops simulation of application-mode programs. A trap is different from a fault: faults are detected before the machine state is changed, for example when an attempt is made to divide by zero. Traps are detected after the machine state is changed, for example, when numeric overflow occurs. This can only happen for application-mode programs; system-mode programs handle this trap through the interruption mechanism. See "Program Simulation" on page 66 and "Interruptions" on page 71.

goto only allowed inside a command file

You tried to execute the **`goto`** command from the keyboard. The command is only legal within command files. See "The goto Command and Labels" on page 84.

Halting Simulation

Your IA-64 system-mode program executed a BREAK 0 instruction at a place where there is no Ski breakpoint. See "How Ski Implements Breakpoints" on page 80 and "System-Mode IA-64 Programs" on page 67.

help: Unknown command: %s

You asked Ski to tell you about a particular command but the command you asked for doesn't exist. Try the **`help`** command alone to get a list of all of the commands Ski understands. See "Command Entry" on page 52.

IA-32 program terminated

An IA-32 application-mode program finished executing and invoked an MS-DOS system function to terminate itself. The function it used doesn't provide a way for the program to return a completion status. See "Application-Mode IA-32 Programs" on page 66.

IA-32 program terminated with status %d

Your IA-32 application-mode program finished execution in the normal fashion and invoked an MS-DOS system function to terminate itself and indicate a completion status. See "Application-Mode IA-32 Programs" on page 66.

Ignored attempt to write a Read-Only symbol

Some registers and symbols recognized by Ski are read-only. You tried to modify one of them. See "Symbolic Arguments" on page 55 and "Changing Registers and Memory with Assignment Commands" on page 75.

Illegal expression: %s

You used an expression that can't be parsed. Check parentheses, variable names, and the matching of operands and operators. See "Expressions" on page 53.

%s: Illegal number of arguments < %d >:

You passed too few or too many operands with a Ski command. The command appears in the %s field on the left and the number of operands you passed appears in the %d field on the right. Use the **`help`** command for information about the command of interest or see "Command Reference" on page 87.

Illegal operation fault

An attempt was made to execute an invalid instruction; probably a wild pointer in a jump table caused a wild branch. This can only happen for application-mode programs; system-mode programs handle this fault through the interruption mechanism. See "Program Simulation" on page 66.

Illegal slot field in breakpoint address

You used the **`bs`** command to set an IA-64 breakpoint, but you specified an address in the last four bytes of a bundle. Because the IA-64 architecture provides for bundle-level, but not instruction-level, addressing, Ski "pretends" that the first instruction of the bundle is in the first four bytes, the second instruction is in the second four bytes, and the third instruction is in the third four bytes. You specified a location in the fourth four bytes of a bundle and that isn't allowed by Ski. See "Setting Program Breakpoints" on page 79 and "How Ski Implements Breakpoints" on page 80.

Interrupting simulation

Ski received a SIGINT signal while simulating, probably because you hit control-C (or whatever key you have configured to interrupt a running program.) This is a status message, not an error message. See "Interruptions" on page 71 and the first few paragraphs of "Command Files" on page 84.

missing command

You used the "`if` *expression true_command* `:` *false_command*" command. Either you left the *true_command* blank and the *expression* evaluated to a non-zero value, or you left the *false_command* blank and the *expression* evaluated to zero. See "The if Command" on page 85.

Missing ELF header

See "File size > Memory size".

Missing file version number

You are trying to restore a saved simulator state and the first non-blank, non-comment line of the file doesn't begin with "`file_ver`", the file version string. Is the file a Ski simulator state save file? See "Saving and Restoring the Simulator State" on page 82.

missing value for option %s

You specified a command line option that requires an argument. See "Command Line Flags" on page 33.

More than %d characters in expression: %s

You gave Ski an expression that is too long for it to parse. Try a shorter expression. See "Expressions" on page 53.

Nesting overflow

You invoked a command file from within another command file, and another command file from within there, and again and again... and you did it too much. Do you have an recursive loop, where a command file invokes itself? See "Command Files" on page 84.

No breakpoints set

You tried to list all breakpoints with the `bl` command but there aren't any. See "Listing Program Breakpoints" on page 79.

No breakpoints to delete

You tried to delete all breakpoints with the `bD` command but there aren't any. See "Deleting Program Breakpoints" on page 79.

No previous command

You tried to re-run the previous command in ***ski*** but you haven't executed any commands yet there is nothing to re-run. See "The ski Command Window" on page 49.

No such command

You typed a command to Ski that Ski doesn't understand. Either you mis-typed, or Ski is broken, or the rules that underpin the basic functioning of our universe have ceased to operate properly. In the first case, try typing your command correctly; use the "`help`" command or see "Command Reference" on page 87 to find out what the commands are. In the third case, you're on your own; bring film.

No such user %s

You specified a filename with a leading tilde ("**~**"), causing Ski to try to expand the first word into the home directory for the corresponding user. Ski wasn't able to the find the user. Perhaps you mis-typed the filename or specified a user that doesn't exist? See "Filenames" on page 57.

Non %s-aligned address. Aligned to 0x%llx

You used the **`=2`**, **`=4`**, or **`=8`** commands to write data to memory but you specified an improperly-aligned address. The %s field on the left tells what kind of alignment was needed and the %llx field on the right is the address that Ski used. This may not be the address you want! See "Changing Registers and Memory with Assignment Commands" on page 75.

Not an ELF file

Not an IA-64 file

See "File size > Memory size".

Nothing to run

No program has been loaded. Use the **`load`**, **`iaload`**, or **`romload`** command, depending on what kind of program you want to simulate or load an IA-64 program by naming it on Ski's command line. See "Program Loading" on page 68.

Out of memory

Ski needed to get more memory to run but couldn't get it. You need more virtual memory swap space or you've found a Ski defect. See your local Linux specialist.

Page not allocated

When Ski loads an IA-64 application-mode program, Ski allocates pages for the fixed-size parts of the program and allocates a small stack. As the program runs, Ski allows the stack to grow. If the program tries to access a page which isn't in one of those areas, Ski detects the error and prints the message. The most likely cause is a wild pointer. See "Application-Mode IA-64 Programs" on page 66.

Pager %s not found

You executed a ***`ski`*** command that sends output through a pager and there was a problem. Did you set the PAGER environment variable to point to a program that's not reachable through your PATH shell variable? Did you set the PAGER variable to point to a non-executable program? If your pager is on a remote file system, is there a problem with accessing that system? Did your pager program return a failure status for some reason? If none of these reasons is applicable, you may have found a Ski defect. See "Other Windows" on page 50.

popen failed

A call to the Linux system routine popen() failed, that is, a -1 was returned from the call. This is unusual and, while it doesn't indicate an internal Ski error, it may suggest that your Linux operating system is corrupt, perhaps due to some other program. ***`ski`*** uses popen() when it needs to invoke a pager to display a large amount of text to you, for example, when you use the **`help`** and **`symlist`** commands. The popen() function might fail if you have the maximum allowed number of processes running on your computer or if you have run out of swap space.

Privileged operation fault

Your IA-64 application-mode program tried to execute a privileged instruction. This can only happen for application-mode programs; system-mode programs handle this fault through the interruption mechanism. See "Program Simulation" on page 66 and "Interruptions" on page 71.

Privileged register fault

Your IA-64 application-mode program tried to access a privileged register. This can only happen for application-mode programs; system-mode programs handle this fault through the interruption mechanism. See "Program Simulation" on page 66 and "Interruptions" on page 71.

program exited with status %d

Your IA-64 program finished execution in the normal fashion. This is a status message, not an error message.

Register NaT Consumption fault

Your IA-64 application-mode program tried to reference the contents of a register that didn't contain a valid value. This can only happen for application-mode programs; system-mode programs handle this fault through the interruption mechanism. See "Program Simulation" on page 66 and "Interruptions" on page 71.

Reserved register/field fault

Your IA-64 application-mode program tried to access a reserved register or portion of a register. This can only happen for application-mode programs; system-mode programs handle this fault through the interruption mechanism. See "Program Simulation" on page 66 and "Interruptions" on page 71.

screen size is %dx%d -- minimum is %dx%d

ski uses the curses package to create a multi-window interface on a terminal. Curses requires a terminal of the specified minimum size but your terminal is smaller than that. See "Ski Variations" on page 31.

Starting address > ending address

You used the **dd** or **pd** command to dump data or program code to a file but the starting address you passed is greater than the ending address. Perhaps you have them reversed? Are you are using symbolic addresses that don't bind to the locations you think they bind to? See "Program Window Commands" on page 60 and "Data Window Commands" on page 63.

Stopping at %s due to IA-32 halt instruction

An IA-32 HALT instruction was reached; simulation has stopped. This is a status message, not an error message. See "Application-Mode IA-32 Programs" on page 66 and "System-Mode IA-32 Programs" on page 67.

Stopping at %s due to reserved IA-32 instruction

An attempt was made to execute an IA-32 instruction whose encoding has been reserved by Intel. Ski recognizes the encoding but doesn't know what to do with it. See "Application-Mode IA-32 Programs" on page 66 and "System-Mode IA-32 Programs" on page 67.

Stopping at %s due to unimplemented IA-32 instruction

An attempt was made to execute an IA-32 instruction that isn't implemented by Ski. See "Application-Mode IA-32 Programs" on page 66 and "System-Mode IA-32 Programs" on page 67.

Stopping at %s due to unimplemented instruction

Your program tried to execute an IA-64 instruction that isn't implemented by Ski.

Symbol %s' not found

You referred to a symbol that Ski doesn't know about. Did you spell the symbol correctly, with leading underscores as needed? Is the symbol a C++ mangled name? Have you loaded the right program? See the section "Argument Specification" on page 53, particularly "Symbolic Arguments" on page 55.

%s: Too many arguments (> %d)

You passed too many operands with a Ski command. Ski's internal parser can handle a maximum number of arguments (currently 64) and you tried to pass more than that number. This could happen with the **`=1`**, **`=2`**, **`=4`**, and **`=8`** assignment commands, the **`eval`** and **`if`** commands, and the **`load`** and **`iaload`** program loading commands. See "Changing Registers and Memory with Assignment Commands" on page 75, "Evaluating Formulas and Formatting Data" on page 78, "The if Command" on page 85, and the section "Program Loading" on page 68.

Too many commands in a line (> %d)

You can type multiple commands on a line by separating them with semicolons. But there's a limit, as shown, to the number of commands you can do this to... and you exceeded it. See "Command Sequences, Repetition, and Abbreviation" on page 52.

Unable to open console window

Your system-mode program tried to open a console with the appropriate Simulator System Call but Ski wasn't able to spawn the corresponding xterm program. First, verify that environment variable DISPLAY is set to the proper *hostname*:*displaynumber* string. If this does not help, perhaps there is no xterm available via your PATH environment variable? Perhaps you have hit the process limit or used all the pseudo-tty devices on your Linux system? See "System-Mode IA-64 Programs" on page 67.

Unaligned Data fault

An attempt was made to access data on an unnatural boundary. Two-byte quantities must be on addresses evenly divisible by two; four-byte quantities must be on addresses evenly divisible by four, and so on. See "Misaligned Data Access Trap" on page 68 and "Interruptions" on page 71.

Unexpected end of file

You are trying to restore a saved simulator state and either the save file is corrupt or Ski is broken. See "Saving and Restoring the Simulator State" on page 82.

unrecognized option %s

You specified a command line option that Ski doesn't understand. Different varieties of Ski (***xski***, ***ski***, and ***bski***) understand different flags. See "Command Line Flags" on page 33.

Unrecognized symbol name: %s

You tried to refer to a symbol in an expression but Ski doesn't know about that symbol. Perhaps you mis-typed it? Or perhaps it is a program-defined symbol in a file that wasn't compiled with debugging symbol generation enabled (the **`-g`** flag on many compilers)? Or perhaps you referred to an IA-64 register using a mnemonic that Ski doesn't recognize? See "Symbolic Arguments" on page 55, "Symbol Table Commands" on page 82, and "Register Names" on page 93.

unsupported DOS int 21 function %02x%02x

Your IA-32 application-mode program tried to invoke an MS-DOS function that Ski doesn't emulate. The first hexadecimal number is the MS-DOS function code and the second number is the sub-function code. See "Application-Mode IA-32 Programs" on page 66 and "MS-DOS Application Environment" on page 73.

Unsupported SSC: %d

Your IA-64 system-mode program invoked a Simulator System Call that Ski doesn't support. Either your program has a bug or Ski is broken. See "System-Mode IA-64 Programs" on page 67.

unsupported system call %d

Your IA-64 application-mode program tried to invoke an Linux system call that Ski doesn't emulate. See "Linux Application Environment" on page 71 and "Application-Mode IA-64 Programs" on page 66.

Usage: %s [options] [file [args]]

Ski's generic command line help message.

14 Licenses

This chapter lists the applicable licenses for Ski.

Creative Commons Public License

CREATIVE COMMONS CORPORATION IS NOT A LAW FIRM AND DOES NOT PROVIDE LEGAL SERVICES. DISTRIBUTION OF THIS LICENSE DOES NOT CREATE AN ATTORNEY-CLIENT RELATIONSHIP. CREATIVE COMMONS PROVIDES THIS INFORMATION ON AN "AS-IS" BASIS. CREATIVE COMMONS MAKES NO WARRANTIES REGARDING THE INFORMATION PROVIDED, AND DISCLAIMS LIABILITY FOR DAMAGES RESULTING FROM ITS USE.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

"Collective Work" means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with one or more other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

"Derivative Work" means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

"Licensor" means the individual, individuals, entity or entities that offers the Work under the terms of this License.

"Original Author" means the individual, individuals, entity or entities who created the Work.

"Work" means the copyrightable work of authorship offered under the terms of this License.

"You" means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

to create and reproduce Derivative Works provided that any such Derivative Work, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";;

to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e. For the avoidance of doubt, where the Work is a musical composition:

Performance Royalties Under Blanket Licenses. Licensor waives the exclusive right to collect, whether individually or, in the event that Licensor is a member of a performance rights society (e.g. ASCAP, BMI, SESAC), via that society, royalties for the public performance or public digital performance (e.g. webcast) of the Work.

Mechanical Rights and Statutory Royalties. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

Webcasting Rights and Statutory Royalties. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of a recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. When You distribute, publicly display, publicly perform, or publicly digitally perform the Work, You may not impose any technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by Section 4(b), as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any credit as required by Section 4(b), as requested.

If You distribute, publicly display, publicly perform, or publicly digitally perform the Work (as defined in Section 1 above) or any Derivative Works (as defined in Section 1 above) or Collective Works (as defined in Section 1 above), You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, consistent with Section 3(b) in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(b) may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear, if a credit for all contributing authors of the Derivative Work or Collective Work appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND ONLY TO THE EXTENT OF ANY RIGHTS HELD IN THE LICENSED WORK BY THE LICENSOR. THE LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MARKETABILITY, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works (as defined in Section 1 above) or Collective Works (as defined in Section 1 above) from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

Each time You distribute or publicly digitally perform the Work (as defined in Section 1 above) or a Collective Work (as defined in Section 1 above), the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt, this trademark restriction does not form part of the License.

Creative Commons may be contacted at http://creativecommons.org/.