Interprocess Communication SKILL Functions Reference

Product Version 06.10
April 2003

Cadence Design Systems, Inc., 555 River Oaks Parkway, San Jose, CA 95134, USA

Contents

Before You Start

Overview information:

About This Manual

This manual is for the following users.

- Programmers beginning to program in SKILL™
- CAD developers who have experience programming in SKILL, both Cadence internal users and Cadence customers
- CAD integrators

About the SKILL Language

The SKILL programming language lets you customize and extend your design environment. SKILL provides a safe, high-level programming environment that automatically handles many traditional system programming operations, such as memory management. SKILL programs can be immediately executed in the Cadence environment.

SKILL is ideal for rapid prototyping. You can incrementally validate the steps of your algorithm before incorporating them in a larger program.

Storage management errors are persistently the most common reason cited for schedule delays in traditional software development. SKILL's automatic storage management relieves your program of the burden of explicit storage management. You gain control of your software development schedule.

SKILL also controls notoriously error-prone system programming tasks like list management and complex exception handling, allowing you to focus on the relevant details of your algorithm or user interface design. Your programs will be more maintainable because they will be more concise.

The Cadence environment allows SKILL program development such as user interface customization. The SKILL Development Environment contains powerful tracing, debugging, and profiling tools for more ambitious projects.

SKILL leverages your investment in Cadence technology because you can combine existing functionality and add new capabilities.

SKILL allows you to access and control all the components of your tool environment: the User Interface Management System, the Design Database, and the commands of any integrated design tool. You can even loosely couple proprietary design tools as separate processes with SKILL's interprocess communication facilities.

Other Sources of Information

For more information about SKILL and other related products, you can consult the sources listed below.

Product Installation

The *Cadence Installation Guide* tells you how to install the product.

Other SKILL Development Documentation

The following are SKILL development-related documents. You can access this information directly using the CDSDoc SKILL menu.

SKILL Development Help
SKILL Development Functions Reference
SKILL Language User Guide
SKILL Language Functions Reference
SKILL+ Object System Functions Reference

Related SKILL API Documentation

Cadence tools have their own application procedural interface functions. You can access the API manuals directly using the CDSDoc SKILL menu.

Design Framework II SKILL Functions contains APIs for the graphics editor, database access, design management, technology file administration, online environment, design flow, user entry, display lists, component description format, and graph browser.

User Interface SKILL Functions contains APIs for management of windows and forms.

Software Installation and License Management Reference in the *Cadence Configuration Guide* contains SKILL licensing functions.

Document Conventions

The conventions used in this document are explained in the following sections. This includes the subsections used in the definition of each function and the font and style of the syntax conventions.

Section Names and Meaning

Each function can have up to seven sections. Not every section is required for every function description.

- Syntax

The syntax requirements for this function.

- Prerequisites

Steps required before calling this function.

- Description

A brief phrase identifying the purpose of the function.

A text description of the operation performed by the function.

- Arguments

An explanation of the arguments input to the function.

- Return Value

An explanation of the value returned by the function.

- Example

Actual SKILL code using this function.

- References

Other functions that are relevant to the operation of this function: ones with partial or similar functionality or which could be called by or could call this function. Sections in this manual which explain how to use this function.

Syntax Conventions

This list describes the syntax conventions used in this document.

`literal (LITERAL)`

Nonitalic (UPPERCASE) words indicate keywords that you must enter literally. These keywords represent command (function, routine) or option names.

argument (z_argument)
: Words in italics indicate user-defined arguments for which you must substitute a name or a value. (The characters before the underscore (_) in the word indicate the data types that this argument can take. Names are case sensitive. Do not type the underscore (*z_*) before your arguments.)

|
: Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other character.

[ ]
: Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.

{ }
: Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.

...
: Three dots (`...`) indicate that you can repeat the previous argument. If you use them with brackets, you can specify zero or more arguments. If they are used without brackets, you must specify at least one argument, but you can specify more.

```
argument...   ;specify at least one,
              ;but more are possible

[argument]... ;specify zero or more
```

,...
: A comma and three dots together indicate that if you specify more than one argument, you must separate those arguments by commas.

=>
: A right arrow points to the return values of the function. Variable values returned by the software are shown in italics. Returned literals, such as `t` and `nil`, are in plain text. The right arrow is also used in code examples in SKILL manuals.

/
: A slash separates the possible values that can be returned by a SKILL function.

Note: The language requires any characters not included in the list above. You must enter required characters literally.

SKILL Syntax Examples

The following examples show typical syntax characters used in SKILL.

Example 1

```
list( g_arg1 [g_arg2] ...) => l_result
```

This example illustrates the following syntax characters.

`list`	Plain type indicates words that you must enter literally.
`g_arg1`	Words in italics indicate arguments for which you must substitute a name or a value.
`( )`	Parentheses separate names of functions from their arguments.
`_`	An underscore separates an argument type (left) from an argument name (right).
`[ ]`	Brackets indicate that the enclosed argument is optional.
`...`	Three dots indicate that the preceding item can appear any number of times.
`=>`	A right arrow points to the description of the return value of the function. Also used in code examples in SKILL manuals.
`l_result`	All SKILL functions compute a data value known as the return value of the function.

Example 2

```
needNCells( s_cellType | st_userType x_cellCount) => t / nil
```

This example illustrates two additional syntax characters.

`|`	Vertical bars separate a choice of required options.
`/`	Slashes separate possible return values.

1

Interprocess Communication SKILL Functions

Overview information:

Overview

The Interprocess Communication (IPC) SKILL functions allow you to create and communicate with child processes. This mechanism allows SKILL-based programs access to IPC and process control functionality that would normally require system level programming.

Using this mechanism you can:

- Create encapsulation tools or utility programs.
- Communicate with encapsulated programs using standard IO channels.
- Control the encapsulated programs by sending signals like kill, interrupt, stop, and continue.
- Allow encapsulated programs to execute SKILL commands in the parent process.
- Run child processes on remote hosts.

The ability to run child processes, establish communication channels and control the processes through a SKILL procedural interface is a powerful utility. Programmers are advised to familiarize themselves with the basic principles of network and distributed programming.

Communicating With Child Processes

A child process can be a program that executes normally under the given operating system. Design Framework II runs non-Cadence software as a child process. A child process can be as simple as execution of a UNIX utility, such as, `mail`, `wc`, `cat`, `ls`, stand-alone simulator, a batch program, and so forth. Basically any process can be a child process, and run in parallel with the parent process that created it.

The parent process communicates with a child process by writing to the child process's `stdin` channel and reading from its `stdout` and `stderr` channels. Communication can be carried out in one of two modes: synchronous or asynchronous.

Handling Child Process Output

When using SKILL interprocess communication, you should be aware of two possible modes of dealing with output from a child process. You can synchronize the flow of a program with child process output by performing blocking read operations. A blocking read operation will

wait until data arrives from the child process thereby guaranteeing sequential flow of your program.

Alternatively, you can choose to deal with output from a child process by registering a call-back function (referred to in this document as `outputHandler`). This function will be called asynchronously whenever data is received from a child process and the event manager in the parent program is ready to handle the data.

There is only one mode of operation for the write function. Write always returns with a success/failure status. When a call to write returns, it does not always mean that the child process received the data. It just means that the data was dispatched successfully.

Blocking Reads and the SKILL Evaluation Process

A blocking read overrides the outputHandler and data entered using one of the methods is never available again for the other method to retrieve.

You should determine in advance whether the use for SKILL IPC requires synchronous or asynchronous input and output handling, in which case either blocking reads or handlers should be the mode of operation. Synchronous and asynchronous output handling should not be mixed. An `errHandler`, once defined for a process, always receives the error messages despite a blocking read.

Remember when writing asynchronous data handling code that the SKILL evaluation process blocks out any incoming messages. These messages cannot be gathered until the evaluator winds down and control returns to the top level.

It is sometimes necessary to open gaps in the evaluator to collect incoming messages. These gaps can be opened using one of the following methods:

- Blocking read with a time-out greater than 0 (`dataHandlers` will not be called during a blocking read)
- `ipcSleep`, `ipcWait`, `ipcWaitForProcess` (`dataHandlers` will be called during these calls)

Tuning the Handlers to Avoid Freezing Graphics

The data handlers are routines invoked by the SKILL interpreter in a non-deterministic fashion. You must tune their performance with respect to the frequency of incoming data because their activation can disrupt the responsiveness of the user interface graphics. Remember, it can be annoying to a user when the system feels unresponsive during the time data handlers are executing.

Waiting for the Child to Become Active

The `ipcBeginProcess` and `ipcSkillProcess` function calls initiate a child process and return without waiting for that child to become active.

To synchronize the activity of the parent process with that of the child process spawning and being ready for communication, use the `ipcWaitForProcess` function to force the parent process to wait until the child process is ready to communicate.

Data Buffers

The input and output performed by child processes must take into account buffer limitations. The standard IO channels have a 4096 byte buffer. For example, a child process's output may not always get flushed immediately after the child writes to `stdout`. A child process may have to flush data at appropriate points so the parent process can receive the data.

Buffer limits do not apply to the SKILL-based parent process. For example, a child process's data is buffered in the parent process using memory pools limited only by the availability of runtime memory.

Data written to a child process's `stdin` channel should be read by the child process frequently. If the `stdin` channel buffer fills up then the parent process discards data to prevent blocking on write.

Child Process Handles

A child process handle returned from a call to `ipcBeginProcess`, `ipcSkillProcess`, or `ipcBatchProcess` is an opaque data structure.

Child Process Read-Only Properties

A child process handle has the following read-only properties that can be accessed programmatically using the `->` syntax.

Property	Meaning
command	Name of the command
host	Name of the host machine running the process
processPid	Process id of the child process on host
exitStatus	Exit status of the child process
priority	Priority given to the child process
type	Begin, SKILL, or Batch process
state	Active, Dead, or Stopped

Some of these properties are only meaningful if the child process is active. Once the child process expires, only `state` and `exitStatus` are guaranteed to have meaningful results.

Formatting Child to Parent SKILL Communication

Processes invoked using `ipcSkillProcess` send SKILL commands back to the parent for execution. Each command sent by the child must be formatted in the following way to ensure error-free execution.

Surround Each Command with Parentheses

For example, to send two `println` commands, format the string this way:

```
(println x) (println y)
```

When the child performs multiple print statements in sequence, the parentheses are needed:

```
..printf("(println x) ");printf("(println x) ");
```

Insert Spaces at the End of Each Command

Alternatively, use the SKILL `prog` construct to send compound statements to SKILL. SKILL commands sent by a child process can become packed together in one string and sent to SKILL to evaluate. Therefore, exercise care in using the correct syntax as in the example above.

This is similar to typing more than one command per line at the Command Interpreter Window. In fact, the CIW is a good place to experiment with formats of compound statements.

Detecting Child Process Termination

There are two ways of detecting child process termination:

- The synchronous method using `ipcIsAliveProcess` or `ipcWait`.
- The asynchronous method using `postFunc` at initiation time.

Behavior is undefined if you mix the use of synchronous and asynchronous child process exit detection.

What's New

The SKILL IPC mechanism was originally implemented as part of the Human Interface layer (*hi*). This layer was represented by the `hiBeginProcess` family of functions. This layer used X as the communication medium.

New Functions

The following functions have been added or changed in recent releases.

- ipcWait has been modified to include the *`x_interval`* argument

New Layer

The hi-based layer is being replaced by a newly rewritten layer represented by the `ipcBeginProcess` family of functions. Although the function names are slightly different, their functionality is identical.

The aim of the new layer is to enhance performance and remove the dependency on X as a communication medium. The new layer uses *sockets* for interprocess communications.

New Function Names and Handle Type

Users are encouraged to migrate their code from using hi-based communication functions to using the ipc* functions.

The main difference between the two procedural interfaces is that the child process handle returned from the function spawning the process (such as, `hiBeginProcess`) is no longer of type `integer`. The new implementation returns an opaque handle, that is, a C structure wrapped in SKILL. Only the type of the handle is different; its use remains the same.

Although the function names are slightly different, their functionality is identical.

Function Name Changes

New Name	Function Being Discontinued
ipcActivateBatch	hiActivateBatch
ipcActivateMessages	hiActivateMessages
ipcBatchProcess	hiBatchProcess
ipcBeginProcess	hiBeginProcess
ipcCloseProcess	hiCloseChild
ipcContProcess	hiContChild
ipcGetExitStatus	hiGetExitStatus
ipcGetPid	hiGetPid
ipcGetPriority	hiGetPriority
ipcIsActiveProcess	hiIsActiveChild
ipcIsAliveProcess	hiIsAliveChild
ipcKillAllProcesses	hiKillAllProcs
ipcKillProcess	hiKillChild
ipcReadProcess	hiReadChild
ipcSetPriority	hiSetPriority
ipcSkillProcess	hiSkillProcess
ipcSleep	hiSleep
ipcSoftInterrupt	hiSoftInterrupt
ipcStopProcess	hiStopChild
ipcWait	hiWait
ipcWaitForProcess	hiWaitForChild

New Name	Function Being Discontinued
ipcWriteProcess	hiWriteChild

New Installation Requirement

To make IPC work, an installation in addition to the ones needed for normal SKILL operations is required. The `bin` package `cdsServIpc` must be present in the search path. This replaces the `serv` daemon used with the hi-based implementation. If you are running a child process remotely, then `cdsServIpc` must be present locally on the remote host.

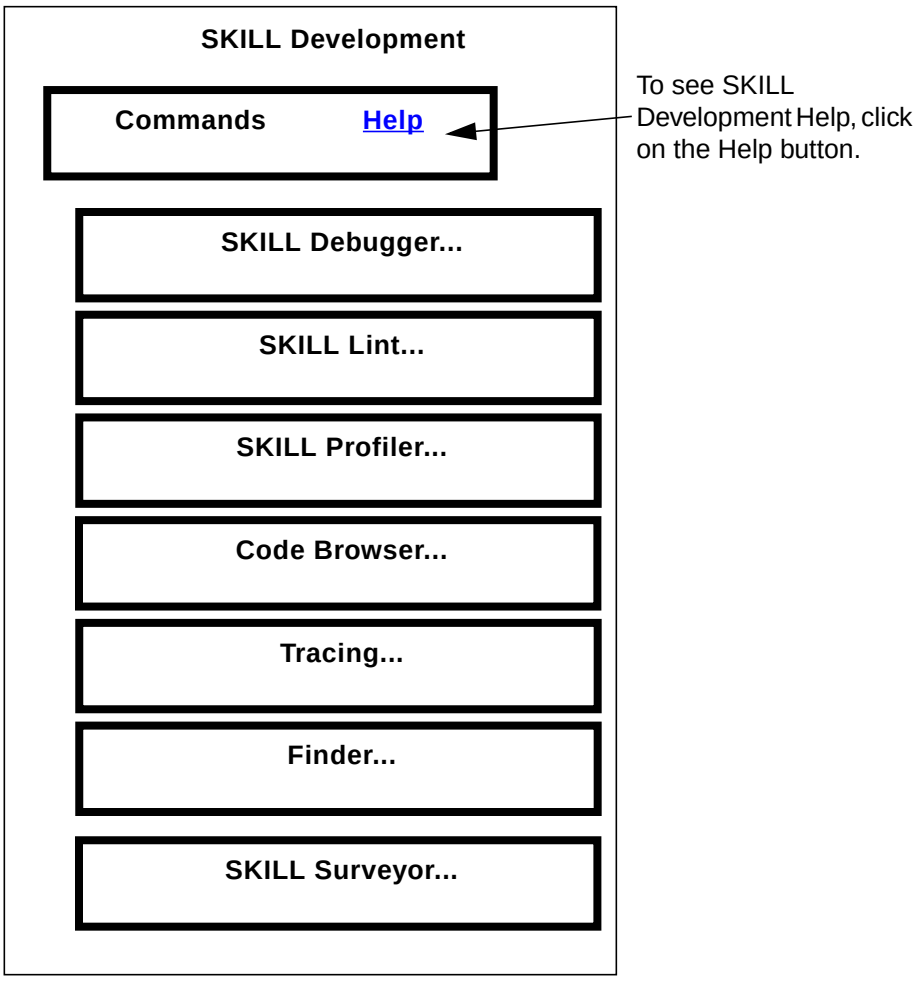
Copying and Pasting Code Examples

You can copy examples from CDSDoc windows and paste the code directly into the CIW or use the code in nongraphics SKILL mode.

To select text,

- Press `Control-drag left mouse` to select a text segment of any size.
- Press `Control-double click left mouse` to select a word.
- Press `Control-triple click left mouse` to select an entire section.

SKILL Development Help



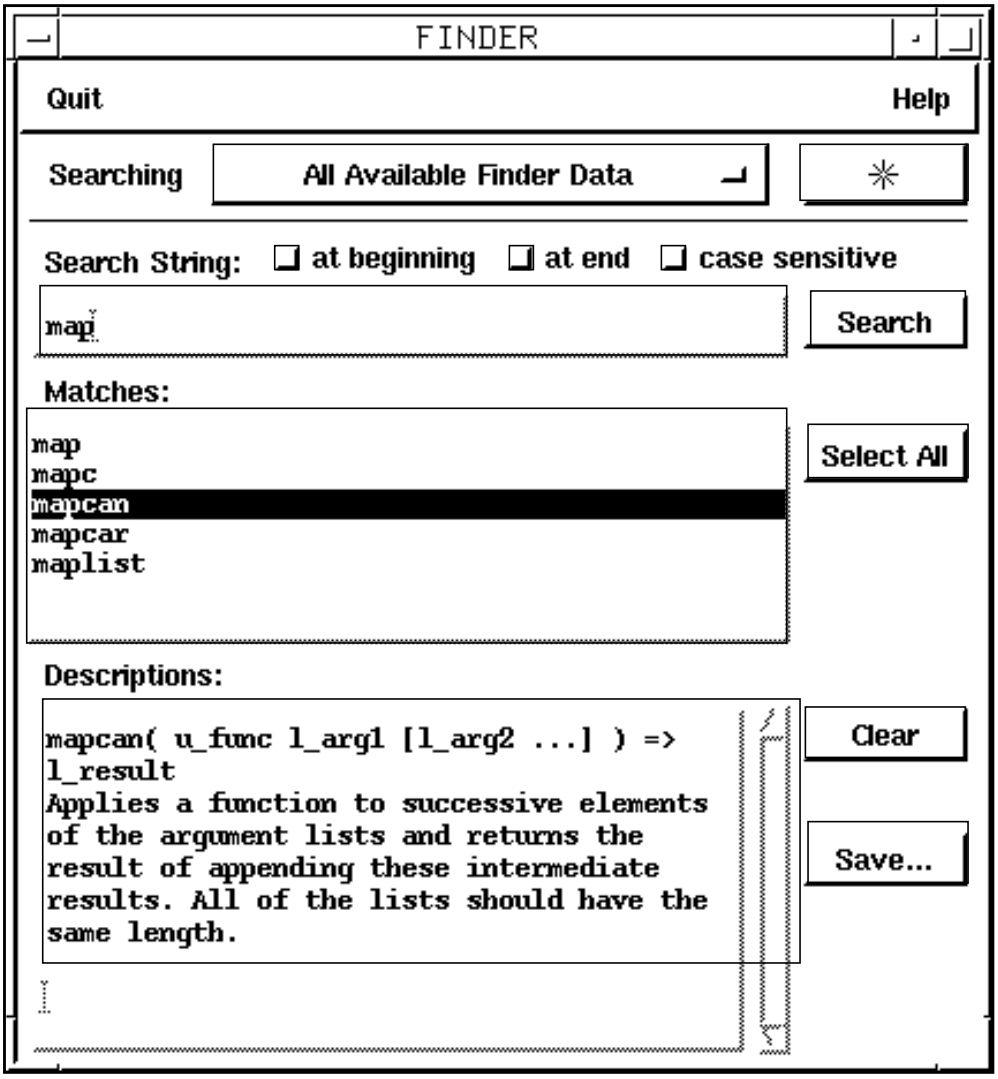
Information about the SKILL Development Toolbox is available in SKILL Development Help, which you access by clicking the *Help* button on the toolbox. Use this source for toolbox command reference information.

The Walkthrough topic in this help system identifies and explains the tasks you perform when you develop SKILL programs using the SKILL Development Toolbox. Using a demonstration program, it explains the various tools available to help you measure the performance of your code and also look for possible errors and inefficiencies in your code. It includes a section on working in the non-graphical environment.

For a list of SKILL lint messages, and message groups, refer to the *SKILL Development Help*.

Quick Reference Tool - Finder

Quick reference information for syntax and abstract statements for SKILL language functions and application procedural interfaces (APIs) is available using the Finder, a new tool accessible from the SKILL Development Toolbox or from UNIX.



For more information, refer to Chapter 7, “Finder,” in *SKILL Development Help*.

SKILL Functions

ipcActivateBatch

```
ipcActivateBatch(
     o_childId
     )
     => t / nil
```

Description

Switches a child process to batch mode.

This means that output from the child is written only to the log file given when the child was created.

Prerequisites

The child process must have started its life through either `ipcBeginProcess` or `ipcSkillProcess` and a log file must have been given. An error could result if these conditions are not met.

Arguments

o_childId	Child process handle.

Value Returned

`nil`	If the child process has already expired.
`t`	Otherwise.

Example

```
      cid = ipcBeginProcess("ls -lR /" "" nil nil nil
                            "/tmp/ls.log")
ipc:0
      ipcActivateBatch(cid)
                  ; Write output to log file /tmp/ls.log only
t
      ipcActivateMessages(cid)      ; Output is written to the
                              ; log file and passed
```

```
                                    ; to parent process
t
```

Reference

ipcActivateMessages, ipcBeginProcess, ipcSkillProcess

ipcActivateMessages

```
ipcActivateMessages(
     o_childId
     )
     => t / nil
```

Description

Switches a child process into interactive mode. In interactive mode, output from the child is written to a log file and is passed on to the parent process.

Prerequisites

The child process must have started its life through either `ipcBeginProcess` or `ipcSkillProcess` and a `logFile` must have been given. An error could result if these conditions are not met.

Arguments

o_childId	Child process handle.

Value Returned

`nil`	If the child process has already expired.
`t`	Otherwise.

Example

```
      cid = ipcBeginProcess("ls -lR /" "" nil nil nil
                            "/tmp/ls.log")
ipc:0
      ipcActivateBatch(cid)
                     ; Write output to log file /tmp/ls.log only
t
      ipcActivateMessages(cid)          ; Output is written to the
                                  ; log file and passed
                                  ; to parent process
t
```

Reference

ipcActivateBatch, ipcBeginProcess, ipcSkillProcess

ipcBatchProcess

```
ipcBatchProcess(
     t_command
     t_hostName
     t_logFile
     )
     => o_childId
```

Description

Invokes a UNIX process to execute batch commands. The child process in this case is a batch process that does not communicate with the parent process.

This child process is locked in the batch mode and cannot be switched into the active data passing mode.

Arguments

t_command	Command to be executed locally or on a network node.
t_hostName	Network node. A null *hostName* means the process is run locally.
t_logFile	Data written to the child's `stdout` and `stderr` is written into this *logFile*. The *logFile* is closed when the child terminates and can be read subsequently using file input and output functions.

Value Returned

o_childId	Batch process that does not communicate with the parent process.

Example

```
cid = ipcBatchProcess("ls /tmp" "" "/tmp/ls.log")
ipc:0
```

Then, `/tmp/ls.log` has the file listing of `/tmp`.

ipcBeginProcess

```
ipcBeginProcess(
     t_command
     [ t_hostName ]
     [ tsu_dataHandler ]
     [ tsu_errHandler ]
     [ tsu_postFunc ]
     [ t_logFile ]
     )
     => o_childId
```

Description

Invokes a UNIX process to execute a command or sequence of commands specified.

The commands are executed locally or on a network node as specified by the argument *hostName*. The newly initiated child process communicates with its parent process using the standard descriptors, `stdin`, `stdout` and `stderr`, as the main input and output channels. Data written by the child into `stdout` and `stderr` is received by the parent, and data sent by the parent is written into the child's `stdin`.

With the exception of the command string, the parameters passed to `ipcBeginProcess` are optional.

The call back arguments (data handlers and post function) can given as symbols, strings or function objects.

- A "" *hostName* means the process is run locally.
- If a handler is `nil`, the data received from the child is buffered for a `ipcReadProcess` call.
- If *postFunc* is `nil`, the child process's state and exit status must be checked using the `ipcIsAliveProcess` or `ipcWait` and `ipcGetExitStatus` functions (or use the *state* and *exitStatus* handle properties).
- If *logFile* is null, the child process cannot be switched to batch mode and its output is always sent to the parent.

Arguments

t_command	Command to be executed locally or on a network node.

t_hostName Specifies the network node. A null *hostName* means the process is run locally.

tsu_dataHandler, *tsu_errHandler*, *tsu_postFunc*
These call back functions can be given as strings, symbols or function objects. Handlers are called whenever the parent process receives data from the child process. Activation of handler calls occurs at the top level of SKILL; that is, it does not interrupt the current evaluation. Define handlers to accept two parameters: *o_childId* and *t_data*. Handlers are called with the *childId* of the child that sent the data and the data itself is packed into a SKILL string.

If *tsu_dataHandler* is `nil`, the parent must use *ipcReadProcess* to read the data.

tsu_dataHandler, *tsu_errHandler*
Correspond to a child's `stdout` and `stderr` respectively.

The *tsu_postFunc* function is called when a child terminates. It must be defined to accept two parameters: *o_childId* and *x_exitStatus*, where *exitStatus* is the value returned by the child process on exit. If *tsu_postFunc* is `nil`, the child's health and exit status must be checked using the `ipcIsAliveProcess` and `ipcGetExitStatus` functions.

t_logFile File that can be used to log all output from a child process.

A child invoked with the *t_logFile* present starts its life duplicating its output to the log file and sending the data to the parent. If at any point the child is to be put in batch mode and its communications with the parent silenced, use `ipcActivateBatch`. Once in batch mode, the output of a child process is written to the *logFile* only. Subsequently, the messages to the parent can be turned back on using `ipcActivateMessages`. Using these two functions, a child process can be made to switch between the batch and active data passing states.

Value Returned

o_childId Your handle on the child process. All operations performed on a child process need *o_childId*. The value of *o_childId* is

not meaningful to the underlying operating system. System calls, therefore, cannot use this value.

The UNIX commands executed by the child process do not require special modification to be invoked under SKILL. Their input and output streams function the same way as they do in the UNIX operating system. For example, if the child process tries to read from its `stdin` and there is no data currently available, the read operation blocks until data becomes available.

Example

```
cid = ipcBeginProcess("date")
ipc:0
      ipcReadProcess(cid)
"Tue Aug 1 14:23:07 PDT 1995\n"

      handler = (lambda (cid data) printf("\n Date:%s\n" data))
funobj:0x2848e8
      cid = ipcBeginProcess("date" "" handler)
ipc:0
Date: Tue Aug 1 14:29:17 PDT 1995
```

Note: Single quotation marks are used inside of double quotation marks for exact words.

```
ipcBeginProcess("grep '> is a greater' /tmp/temp_ipcfile")
```

Do not use single quotation marks for grouping commands.

```
cid = ipcBeginProcess("'cd /tmp; ls -l temp_ipcfile'")
ipc:0
```

Use double quotation marks for spawning multiple commands.

```
cid = ipcBeginProcess("cd /tmp; ls -l temp_ipcfile")
ipc:0
```

Reference

ipcBatchProcess, ipcSkillProcess

ipcCloseProcess

```
ipcCloseProcess(
     o_childId
     )
     => t / nil
```

Description

Closes the input channel of the child process.

This is the equivalent of a `Control-d` sent down the input channel of the child process. Some commands in UNIX will wait for a `Control-d` before processing their input, so this function allows for that to happen programmatically.

Arguments

o_childId	Child process handle.

Value Returned

`t`	If the child process is alive.
`nil`	If the child process is expired.

Example

```
      cid = ipcBeginProcess("wc")
ipc:0
      ipcWriteProcess(cid "line 1\n")
t
      ipcWriteProcess(cid "line 2\n")
t
      ipcCloseProcess(cid)
t
      ipcReadProcess(cid)
"   2    4    14\n"
```

ipcContProcess

```
ipcContProcess(
     o_childId
     )
     => t / nil
```

Description

Causes a suspended child process to resume executing. Equivalent to sending a UNIX `CONT` signal.

Arguments

o_childId	Child process handle.

Value Returned

`nil`	If the child has already expired, `nil` is the result.
`t`	Otherwise.

Example

```
      cid = ipcBeginProcess("ls -lR /")
ipc:0
      ipcIsActiveProcess(cid)
t
      ipcStopProcess(cid)                ; Stop the execution
t
      ipcIsActiveProcess(cid)
nil
      ipcContProcess(cid)                ; Resume the execution
t
      ipcIsActiveProcess(cid)
t
```

Reference

ipcStopProcess

ipcGetExitStatus

```
ipcGetExitStatus(
     o_childId
     )
     => x_status
```

Description

Returns the exit value of the child process.

If *postFunc* is used in the initiation of a process, this call is not necessary.

Arguments

o_childId	Child process handle.

Value Returned

x_status	Exit value of the child process.

Example

```
      cid = ipcBeginProcess("ls")
ipc:0
      ipcGetExitStatus(cid)
0
      cid = ipcBeginProcess("bad_command")
                           ; The command will cause an error
ipc:0
      ipcGetExitStatus(cid)
1
```

Reference

ipcBeginProcess

ipcGetPid

```
ipcGetPid(
     )
     => x_pid
```

Description

Returns the runtime process identification number of the process executing this function.

Arguments

None.

Value Returned

x_pid	Runtime process identification number.

Example

```
      ipcGetPid
885               ; Runtime process identification number
```

ipcGetPriority

```
ipcGetPriority(
     [ o_childId ]
     )
     => x_priority
```

Description

Gets the current default priority. If a child process handle is given, `ipcGetPriority` returns the priority under which the relevant child process was invoked.

Arguments

o_childId	Child process handle returned from `ipcBatchProcess`, `ipcBeginProcess`, or `ipcSkillProcess`. This is an optional argument.

Value Returned

x_priority	Current default priority or the priority under which a child process that associates with the given *o_childId* was invoked.

Example

```
      ipcGetPriority()              ; Default priority
15
      ipcSetPriority(5)
t
      ipcGetPriority()              ; New default priority
5
cid0 = ipcBeginProcess("pwd")
ipc:0
      ipcGetPriority(cid0)          ; Priority of the child
5                                   ; process associates with
                                    ; 'cid0'


      ipcSetPriority(10)
t
      ipcGetPriority()
10
      cid1 = ipcBeginProcess("ls")
ipc:1
      ipcGetPriority(cid1)          ; The child process associates
10                                  ; with 'cid1' runs at the new
                                    ; default priority
```

```
      ipcGetPriority(cid0)           ; Priority of the child
5                                    ; process associates with
                                     ; only tighted to the
                                     ; priority under which it
                                     ; was invoked.
```

Reference

ipcSetPriority

ipcIsActiveProcess

```
ipcIsActiveProcess(
     o_childId
     )
     => t / nil
```

Description

Determines if a child process is alive; that is, not stopped.

Arguments

o_childId	Child process handle.

Value Returned

nil	If the child process is stopped or expired.
t	If the child is alive.

Example

```
      cid = ipcBeginProcess("ls -lR /")
ipc:0
      ipcIsActiveProcess(cid)
t
      ipcStopProcess(cid)                  ; Stop the execution
t
      ipcIsActiveProcess(cid)
nil
      ipcContProcess(cid)                  ; Resume the execution
t
      ipcIsActiveProcess(cid)
t
```

Reference

ipcContProcess, ipcKillProcess, ipcStopProcess

ipcIsAliveProcess

```
ipcIsAliveProcess(
     o_childId
     )
     => t / nil
```

Description

Checks if a child process is still alive.

In real time, notice of a child process's expiration can never be made available immediately after it happens. It is subject to the operating system's underlying process communication delays and to network delays if the child is executing remotely. You need to make allowances for such delays.

Arguments

o_childId	Child process handle.

Value Returned

t	Child process is still alive.
nil	Child process is not alive.

Example

```
      cid = ipcBeginProcess("sleep 15")
ipc:0
      ipcIsAliveProcess(cid)
t
      cid->state
Active
;; wait 15 seconds

      ipcIsAliveProcess(cid)
nil
      cid->state
Dead
```

Reference

ipcBeginProcess, ipcKillProcess

ipcKillAllProcesses

```
ipcKillAllProcesses(
     )
     => t
```

Description

Kills every process initiated by the parent through one of the `ipcBeginProcess` class of functions.

Note: This call will terminate all processes initiated by other applications active in the same parent process.

Arguments

None.

Value Returned

`t`	Always returns `t` so it can be used to clean up without failing.

Example

```
      c1 = ipcBeginProcess("sleep 100")
ipc:0
      c2 = ipcBeginProcess("sleep 100")
ipc:1
      c1
ipc:0
      c2
ipc:1
      ipcKillAllProcesses()
t
      c1
ipc:-1
      c2
ipc:-1
```

Reference

`ipcKillProcess`

ipcKillProcess

```
ipcKillProcess(
     o_childId
     )
     => t / nil
```

Description

Kills the UNIX process identified by *o_childId*. This call results in a UNIX SIGKILL signal being sent to the child process.

Arguments

o_childId	Child process handle.

Value Returned

nil	If the child has already expired.
t	Otherwise.

Example

```
      cid = ipcBeginProcess("sleep 15")
ipc:0
      ipcKillProcess(cid)
t
      ipcKillProcess(cid) ; The child process has expired already
nil
```

Reference

ipcKillAllProcesses

ipcReadProcess

```
ipcReadProcess(
     o_childId
     [ x_timeOut ]
     )
     => t_data / nil
```

Description

Reads data from the child process's `stdout` channel. Permits developer to specify a time, in seconds, beyond which the read operation must not block.

This function takes the child process's handle *`o_childId`* and an integer value *`x_timeOut`* denoting a permitted time, in seconds, beyond which the read operation must not block. Zero is an acceptable value and is a request for a non-blocking read where only buffered data is returned. If data is not available during the allowed time, `nil` is returned.

In the ensuing block caused by a read, incoming data from other child processes is buffered and, once the blocking read releases, all buffers are scanned and data is dealt with accordingly.

Note: A blocking read freezes the parent process's user interface graphics.

The `ipcReadProcess` function takes a finite number of seconds to time-out a block, therefore, deadlocks cannot occur. A *deadlock* occurs when two or more processes block indefinitely while waiting for each other to release a needed resource. The data retrieved by `ipcReadProcess` is not labeled as to its originating port, such as, `stderr` or `stdout`. You can either parse the data to determine the origin or use *`errHandler`* to always trap the errors.

When a blocking read is in progress, the user interface graphics become inactive. Child processes, however, can continue to communicate during the ensuing block, and send SKILL commands (if the child process is invoked by `ipcBatchProcess`) that are executed and their results returned. If an error handler is defined, error messages are buffered rather than given to the blocking read. The activation of the error handler occurs immediately after the read releases. Termination messages are received and any post functions defined are called. This allows a blocking read to release if the corresponding child terminates. Data from other child processes is buffered and dealt with after `ipcReadProcess`.

Arguments

`o_childId`	Child process handle.

x_timeOut	Integer value denoting a permitted time, in seconds, beyond which the read operation must not block. Zero is an acceptable value and is a request for a non-blocking read where only buffered data is returned.

Value Returned

t_data	Data made available during the allowed time.
`nil`	If data is not made available during the allowed time, `nil` is returned.

Example

```
cid = ipcBeginProcess("date")
ipc:0
      ipcReadProcess(cid)
"Tue Aug 1 14:23:07 PDT 1995\n"
```

Reference

ipcBeginProcess, ipcWriteProcess

ipcSetPriority

```
ipcSetPriority(
     x_priorityChange
     )
     => t / nil
```

Description

Sets the priority value for child processes. All processes spawned after this call will run at the priority offset to *x_priorityChange*.

Arguments

x_priorityChange The default value, if this function is not called, tends to be lower than the default operating system priority. The higher the value you give to *x_priorityChange*, the lower the child's scheduling priority. The child process's priority set at the beginning of its life cannot be changed thereafter. The acceptable range of values that *x_priorityChange* can take is 0 to 20 with 15 as the default priority.

Typically, a batch process is run with a low priority. Interactive processes run under normal priority settings. The `ipcSetPriority` function lets you lower priorities more easily than raise them. Some increase is permitted but even the lowest value given to *x_priorityChange* increases actual priority from the norm by little.

Note that *x_priorityChange* is not the absolute priority value that will be used to set the scheduling priority of a process. An actual value of priority change will be derived from the value given to *x_priorityChange*. For example, a child process invoked with the default priority value of 15 will be running at the UNIX OS nice value of 30 (assume the invoking process that calls ipcBeginProcess to spawn the child process is running at the default UNIX OS nice value of 20 and the range of nice values imposed by an UNIX system is 0/40).

Processes with super-user privileges can spawn child processes with nice values lower than the default UNIX OS nice value (thus, raise the scheduling priority) by giving to *x_priorityChange* the range of priority values 0,1,2,3,...9, which maps to the ranges

of UNIX OS nice values 0,2,4,6,...18, respectively (assume that the default UNIX OS nice value is 20).

For non-super-user processes, the range of priority values can be given to *x_priorityChange* is 10,11,12,13,...20, which maps to the ranges of UNIX OS nice values 20,22,24,26,...40 (or 39 because a nice value of 40 is treated as 39 by OS), respectively. The range of priority values 0-9 given to x_priorityChange for non-super-user processes will not lower the UNIX OS nice value further from the default UNIX OS nice value (that is, the lowest value can be given to *x_priorityChange* by non-super-user processes is 10, which maps to the default UNIX OS nice value; typically 20).

Value Returned

`t`	Always returns `t`. Signals an error if the given priority is out of range.

Example

```
      ipcGetPriority()              ; Default priority
15
      ipcSetPriority(10)
t
      ipcGetPriority()
10
      ipcSetPriority(21)            ; Priority out of range
*Error* ipcSetPriority: priority value must be in the range 0-20 - 21
```

Reference

ipcGetPriority

ipcSkillProcess

```
ipcSkillProcess(
     t_command
     [ t_hostName ]
     [ tsu_dataHandler ]
     [ tsu_errHandler ]
     [ tsu_postFunc ]
     [ t_logFile ]
     [ x_cmdDesc ]
     [ x_resDesc ]
     )
     => o_childId
```

Description

Invokes a UNIX process capable of executing SKILL functions in the parent process. Opens two additional channels to the child process that let the child send and receive the results of SKILL commands.

Sending Channel

The SKILL command channel is by default bound to file descriptor number 3 in the child process. In addition to whatever input and output the child process may perform, it can write SKILL executable commands on this descriptor that are in turn sent to the parent to be executed. The parent executes these commands during the next cycle of SKILL's top level without interrupting the current evaluation. The result of this execution is sent back to the child over the SKILL result channel, which is by default bound to file descriptor number 4 in the child process.

The defaults can be over-ridden by supplying the descriptors in the call to `ipcSkillProcess`. These descriptors must be declared and used by the child process, that is, the parent process cannot force the child process to use a particular pair of channels.

SKILL functions written into the SKILL command channel should have sound syntactic structures. For example,

- Use parentheses when writing function calls, even for infix functions.
- Ensure that all command expressions are separated by at least a single space character.

Command expressions with missing parentheses or incomplete strings can cause syntax errors in the SKILL interpreter, thereby causing other functions in the pipeline to fail also.

Result Channel

The results of executing SKILL functions are sent back on the result channel (descriptor 4 by default). It is up to the child process to read from the result channel.

Because of limited buffer sizes, if the child process fails to read accumulated data from the result channel there is a chance that results will be discarded if the buffer fills up.

The buffer for the result channel is separate from all other buffers so the process does not have to empty the buffer if the results are not needed.

Arguments

t_command — Command to be executed locally or on a network node.

t_hostName — Specifies the network node. A null *hostName* means the process is run locally.

tsu_dataHandler, *tsu_errHandler*, *tsu_postFunc*
These call back functions can be given as strings, symbols or function objects. Handlers are called whenever the parent process receives data from the child process. Activation of handler calls occurs at the top level of SKILL; that is, it does not interrupt the current evaluation. Define handlers to accept two parameters: *o_childId* and *t_data*. Handlers are called with the *childId* of the child that sent the data and the data itself is packed into a SKILL string.

If *tsu_dataHandler* is `nil`, the parent must use *ipcReadProcess* to read the data.

tsu_dataHandler, *tsu_errHandler*
Correspond to a child's `stdout` and `stderr` respectively.

The *tsu_postFunc* function is called when a child terminates. It must be defined to accept two parameters: *o_childId* and *x_exitStatus*, where *exitStatus* is the value returned by the child process on exit. If *tsu_postFunc* is `nil`, the child's health and exit status must be checked using the `ipcIsAliveProcess` and `ipcGetExitStatus` functions.

t_logFile — File that can be used to log all output from a child process.

A child invoked with the *t_logFile* present starts its life duplicating its output to the log file and sending the data to the parent. If at any point the child is to be put in batch mode and its communications with the parent silenced, use `ipcActivateBatch`. Once in batch mode, the output of a child process is written to the *logFile* only. Subsequently, the messages to the parent can be turned back on using `ipcActivateMessages`. Using these two functions, a child process can be made to switch between the batch and active data passing states.

x_cmdDesc — SKILL command sending channel.

x_resDesc — SKILL result receiving channel.

Example

Suppose we have a C program, `sample.c`:

```
/*****************************************
* Sample process for executing SKILL commands
* in parent process.
*****************************************/
#include "stdio.h"
#define skill_cmd 3
#define skill_result 4

main(int argc, char **argv)
{
    int status;
    char s[100];
    sprintf(s, "%s", "(let () (println \"Hello world \") (1 + 1))");
    printf("Executing %s", s);
    fflush(stdout);
    status = write(skill_cmd, &s[0], strlen(s));
    status = read(skill_result, &s[0], 100);
    s[status] = '\0';
    printf("Result = %s",  s);
    fflush(stdout);
```

```
    exit(0);
}
```

Compile this into an executable named `sample.exe`. Then in SKILL:

```
      cid = ipcSkillProcess("sample.exe")
ipc:0
"Hello world"
      ipcReadProcess(cid)
"Executing (let () (println \"Hello world\") (1 + 1))"
      ipcReadProcess(cid)
"Result = (2)"
      cid->exitStatus
0
```

Reference

ipcBatchProcess, ipcBeginProcess

ipcSleep

```
ipcSleep(
     x_time
     )
     => t
```

Description

Causes the parent to sleep for the given number of seconds.

While the sleep is in progress, incoming data from child processes is buffered. If handlers are defined, they are called and, if there are SKILL commands among the data, they are executed and their results sent back to the child process.

The `ipcSleep` function gives the programmer a way to break the sequence of evaluations and allow incoming data to take effect without having to return to the SKILL top level.

Arguments

x_time	Number of seconds for the parent to sleep.

Example

```
      handler = (lambda (cid data)
                   when(index(data "cshrc")
                              path = data))
;; Look for the first occurrence of file .cshrc.
;; Do not spend more than n seconds looking
procedure( look_for_cshrc(n)
    path = nil
    n = n/2
    cid = ipcBeginProcess("cd $HOME ; find . -name '.cshrc' -print" "" handler)
    while(and(!path !zerop(n)) ipcSleep(2) n--)
    ipcKillProcess(cid)
    path
)
      look_for_cshrc(150)
"./.cshrc\n"
```

Reference

ipcWait, ipcWaitForProcess

ipcSoftInterrupt

```
ipcSoftInterrupt(
     o_childId
     )
     => t / nil
```

Description

Equivalent to executing the UNIX `kill -2` command. If the child process is active, it is sent a soft interrupt. The child is responsible for catching the signal.

Arguments

o_childId	Child process handle.

Value Returned

`t`	If the child process is active.
`nil`	Otherwise.

Example

```
      cid = ipcBeginProcess("sleep 100")
ipc:0
      ipcSoftInterrupt(cid)
t
      cid
ipc:-1
```

Reference

ipcKillProcess, ipcKillAllProcesses

ipcStopProcess

```
ipcStopProcess(
     o_childId
     )
     => t / nil
```

Description

Causes the child process to suspend its execution. Is equivalent to sending a STOP signal through the UNIX `kill` command.

Arguments

o_childId	Child process handle.

Value Returned

`nil`	If the child has already expired, `nil` is the result.
`t`	Otherwise.

Example

```
      cid = ipcBeginProcess("ls -lR /")
ipc:0
      ipcIsActiveProcess(cid)
t
      ipcStopProcess(cid)                  ; Stop the execution
t
      ipcIsActiveProcess(cid)
nil
      ipcContProcess(cid)                  ; Resume the execution
t
      ipcIsActiveProcess(cid)
t
```

Reference

ipcContProcess

ipcWait

```
ipcWait(
     o_childId
     [ x_interval ]
     [ x_timeOut ]
     )
     => t
```

Description

Causes the parent process to suspend until the child terminates.

This function is like the sleep function in that it allows incoming messages to take effect while waiting.

Arguments

o_childId	Child process handle.
x_interval	The interval at which “Waiting for ... to terminate” message is printed. Default is 30 seconds.
x_timeOut	Time beyond which this call should not block. Defaults to 1000000 seconds.

Value Returned

t	Always returns t.

Example

```
      cid = ipcBeginProcess("sleep 30")
ipc:0
      ipcWait(cid)
            ; Suspends here until the child process terminates
t
```

Reference

ipcSleep, ipcWaitForProcess

ipcWaitForProcess

```
ipcWaitForProcess(
     o_childId
     [ x_timeOut ]
     )
     => t
```

Description

Causes the parent process to suspend until the child process is alive and ready for communication.

Prerequisites

This function is normally used in conjunction with one of the `ipcBeginProcess` class of functions.

Arguments

o_childId	Child process handle.
x_timeOut	Time beyond which this call should not block.

Value Returned

`nil`	If the child has already expired.
`t`	Otherwise.

Example

```
      cid = ipcBeginProcess("date")
ipc:0
      ipcWaitForProcess(cid)
                        ; Wait for the child process coming up
                        ; to guarantee a safe read
t
      ipcReadProcess(cid)
"Thu Aug 24 19:03:14 PDT 1995\n"
```

Reference

ipcBeginProcess, ipcSleep, ipcStopProcess, ipcWait

ipcWriteProcess

```
ipcWriteProcess(
     o_childId
     t_data
     )
     => t / nil
```

Description

Writes data to the child's `stdin` port.

This function takes a *`o_childId`* and a SKILL string containing the data destined for the child process. This function does not block and always returns `t`. However, if the destination child process expires before `ipcWriteProcess` is performed, `nil` is returned.

The data sent through `ipcWriteProcess` is written into the child's `stdin` port. You must ensure that the data sent is appropriately packaged for the child to read in. For example, if the child performs a string read operation such as `gets`, the string given to `ipcWriteProcess` must terminate with a line feed character; otherwise `gets` continues blocking.

Reference

`o_childId`	Child process handle.
`t_data`	SKILL string containing the data destined for the child process. For a child process to read the input, this string must be terminated by a `\n` character.

Value Returned

`t`	If write is successful.
`nil`	If the destination child process expires before `ipcWriteProcess` is performed.

Example

```
;; substitute your login name for user
      cid = ipcBeginProcess("mail user") ;user is your login name
ipc:0
      ipcWriteProcess(cid "Hello from SKILL IPC\n")
```

```
t
      ipcCloseProcess(cid)

Check your email. You should have a message from yourself containing "Hello from
SKILL IPC".
```

Note the \n character at the end of the *t_data* string.

Reference

ipcBeginProcess, ipcReadProcess

Programming Examples

The following programming examples deal with synchronous and asynchronous input and output.

Synchronous Input/Output

The following example is a C program called `X` that reads from its `stdin`, converts every character in the buffer to uppercase, and writes the result back to `stdout`. SKILL puts this program to use by sending to it a string for conversion to uppercase. Copy this program into a file and compile it into a program called `upper.exe`.

```
#include <stdio.h>
#define bufflen 4096
main()
{
    char buff[bufflen];

    while (1) {
        gets(buff);
        {   int i;
            for(i=0; i < strlen(buff); i++)
                buff[i] = toupper(buff[i]);
        }
        printf(buff);
        fflush(stdio);
    }
}
```

The SKILL program to use the previous program is as follows:

```
cid = ipcBeginProcess( "upper.exe" )
ipcWriteProcess( cid "hello\n" )
x = ipcReadProcess( cid 20 )
when(x printf(" New string : %s", x ))
ipcKillProcess( cid ) ;; Kill Or send another string
```

Asynchronous Input/Output

The example is that of a tool such as a simulator being invoked from SKILL and the results of the simulation displayed in the SKILL environment.

```
;SimCid

procedure( dataH(cid data)
    (unless (displaySimResults data)
            error("Display failed \n"))
)

procedure( simErr(cid err)
    print("Error %L Msg: %s\n" cid err)
```

```
    ipcKillProcess(cid) /*
)

procedure (simTerm(cid exit)
    print("Simulator expired with exit status = \n" exit)
)

procedure( initSym(symCommand networkNode)
    ipcBeginProcess(symCommand networkNode
        "dataH" "simErr" "simTerm")
)
```

Assume that a function called `displaySimResults` takes a string of simulation results and displays it as appropriate output. Also, `simErr` and `simTerm` are functions that handle simulator errors and simulator termination condition.

Once the above program, `SimCid`, is loaded into SKILL, the user can run the Verilog® simulator on a powerful computer called `super` available on the network, as follows:

```
SimCid = initSym("verilog" "super")
```

Afterwards the user can continue working with SKILL without having to wait for the simulator. The results of simulation are displayed automatically whenever they become available and the evaluator is free to call the `dataH` function. In this case the simulator must write its output on `stdout` so results can get to the parent SKILL program.

Multiple UNIX Commands

Multiple UNIX commands can be invoked from within a SKILL program by using the `ipcBeginProcess` function, the `ipcBatchProcess` function, or the `ipcSkillProcess` function. For example, the following functions invoke UNIX commands to get a listing of the `tmp` directory. To signal to the operating system that another command follows, separate multiple UNIX commands with either two ampersands (&&) or a single semicolon (;).

```
ipcBeginProcess( "cd /tmp && ls . ")
ipcSkillProcess( "cd /tmp; ls . ")
```