cadence®

# Cadence Interprocess Communication SKILL Reference

**Product Version ICADVM20.1**
**October 2020**

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

# Contents

# Preface

This manual describes how to use SKILL-based programs to access Interprocess Communication (IPC) functions.

It is intended for the following users.

- Programmers beginning to program in SKILL language
- CAD developers (internal users and customers) who have experience in SKILL programming
- CAD integrators

This preface contains the following topics:



## Scope

Unless otherwise noted, the functionality described in this guide can be used in both mature node (for example, IC6.1.8) and advanced node and methodologies (for example, ICADVM20.1) releases.

| Label | Meaning |
|---|---|

| | |
|---|---|
| **(ICADVM20.1 Only)** | Features supported only in the ICADVM20.1 advanced nodes and advanced methodologies releases. |
| **(IC6.1.8 Only)** | Features supported only in mature node releases. |
| **(ICADVM20.1 Photonics Only)** | Features supported only in the ICADVM20.1 release and which require the Virtuoso Photonics Option license (95550). |

# Licensing Requirements

SKILL uses **Cadence Design Framework II** license (License Number 111), which is checked out at the launch of the `skill` executable or the workbench.

For information on licensing in the Cadence SKILL Language, see the *Virtuoso Software Licensing and Configuration User Guide*.

# Related Documentation

## What's New

- *Cadence SKILL Language What's New*

## Installation, Environment, and Infrastructure

- *Cadence Installation Guide*
- *Virtuoso Design Environment SKILL Reference*
- *Cadence Application Infrastructure User Guide*
- *Virtuoso Software Licensing and Configuration Guide*

## Other SKILL Books

- *Cadence SKILL IDE User Guide*
- *Cadence SKILL Development Reference*

- *Cadence SKILL Language User Guide*
- *Cadence SKILL++ Object System Reference*

# Additional Learning Resources

## Video Library

The Video Library on the Cadence Online Support website provides a comprehensive list of videos on various Cadence products.

To view a list of videos related to a specific product, you can use the *Filter Results* feature available in the pane on the left. For example, click the *Virtuoso Layout Suite* product link to view a list of videos available for the product.

You can also save your product preferences in the Product Selection form, which opens when you click the *Edit* icon located next to *My Products*.

## Virtuoso Videos Book

You can access certain videos directly from Cadence Help. To learn more about the related features and to access the list of available videos, see Virtuoso Videos.

## Rapid Adoption Kits

Cadence provides a number of Rapid Adoption Kits that demonstrate how to use Virtuoso applications in your design flows. These kits contain design databases and instructions on how to run the design flow.

In addition, Cadence offers the following training courses on the SKILL programming language:

- SKILL Language Programming Introduction
- SKILL Language Programming
- Advanced SKILL Language Programming

To explore the full range of training courses provided by Cadence in your region, visit Cadence Training or write to training_enroll@cadence.com.

**Note:** The links in this section open in a separate web browser window when clicked in Cadence Help.

### Help and Support Facilities

Virtuoso offers several built-in features to let you access help and support directly from the software.

- The Virtuoso *Help* menu provides consistent help system access across Virtuoso tools and applications. The standard Virtuoso *Help* menu lets you access the most useful help and support resources from the Cadence support and corporate websites directly from the CIW or any Virtuoso application.
- The Virtuoso Welcome Page is a self-help launch pad offering access to a host of useful knowledge resources, including quick links to content available within the Virtuoso installation as well as to other popular online content.

  The Welcome Page is displayed by default when you open Cadence Help in standalone mode from a Virtuoso installation. You can also access it at any time by selecting *Help – Virtuoso Documentation Library* from any application window, or by clicking the *Home* button on the Cadence Help toolbar (provided you have not set a custom home page).

For more information, see Getting Help in *Virtuoso Design Environment User Guide.*

## Customer Support

For assistance with Cadence products:

- Contact Cadence Customer Support

  Cadence is committed to keeping your design teams productive by providing answers to technical questions and to any queries about the latest software updates and training needs. For more information, visit https://www.cadence.com/support.
- Log on to Cadence Online Support

  Customers with a maintenance contract with Cadence can obtain the latest information about various tools at https://support.cadence.com.

## Feedback about Documentation

You can contact Cadence Customer Support to open a service request if you:

- Find erroneous information in a product manual
- Cannot find in a product manual the information you are looking for
- Face an issue while accessing documentation by using Cadence Help

You can also submit feedback by using the following methods:

- In the Cadence Help window, click the *Feedback* button and follow instructions.
- On the Cadence Online Support Product Manuals page, select the required product and submit your feedback by using the *Provide Feedback* box.

# Understanding Cadence SKILL

Cadence SKILL is a high-level, interactive programming language based on the popular artificial intelligence language, Lisp. It lets you customize and extend your design environment. Using SKILL you can validate the steps of your algorithm incrementally before incorporating them into a larger program.

For more information about the SKILL language, see Getting Started in the *SKILL Language User Guide*.

## Using SKILL Code Examples

The SKILL APIs in this user manual are explained with illustrative code examples.

You can copy these examples from the manual and paste them directly into the Command Interpreter Window (CIW) or use the code in non-graphical SKILL mode.

## Sample SKILL Code

The following code sample shows the syntax of a SKILL API that accepts three arguments.

**axlGetRunStatus**

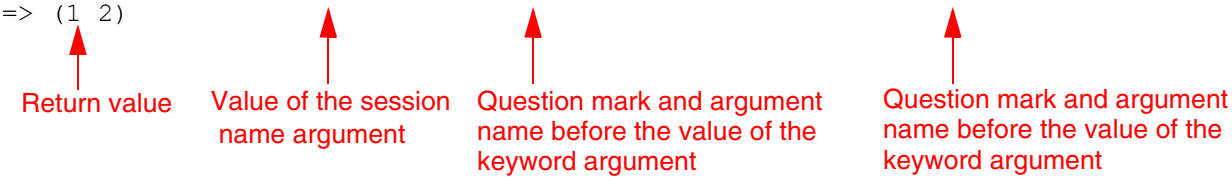
```
axlGetRunStatus(
    t_sessionName                          <-- Required argument
    [ ?optionName t_optionName ]           <-- Optional keyword argument
    [ ?historyName t_historyName ]         <-- Optional keyword argument
    )
    => l_statusValues                      <-- Return value
```

The first argument `t_sessionName` is a required argument, where `t` signifies the data type of the argument. The second and third arguments `?optionName t_optionName` and `?historyName t_historyName` are optional keyword arguments (identified by a question mark), which are specified in name-value pairs and can be placed in any order during the function call.

The return value is the value that the SKILL API returns after evaluating the expression. In this case, it is a list of status values, *l_statusValues*.

**Example**

```
axlSession=axlGetWindowSession( hiGetCurrentWindow())
=> "session0"
axlGetRunStatus("session0" ?historyName "Interactive.10" ?optionName "tests")
=> (1 2)
```




## Accessing API Help

Quick reference information for SKILL APIs is available from the CIW and the SKILL API Finder. To access the reference information for a particular SKILL API, do one of the following:

- Type `help <function_name>` in the CIW.
- Type `startFinder ( [ ?funcName t_functionName ] )` in the CIW.
- Start the SKILL API Finder from the CIW by choosing *Tools – Finder* or type `cdsFinder` on the UNIX command line.

  In the *Search in* field of the displayed Cadence SKILL API Finder window, type the SKILL API name for which you want to display the help information and click *Go*.

  The matches for the searched SKILL API appear in the *Results* area.

  To view the complete documentation of the searched SKILL API, select the API name in the *Results* area and click the *More Info* button. The complete documentation of the selected SKILL API appears in a new Cadence Help window.

# Typographic and Syntax Conventions

The following typographic and syntax conventions are used in this manual.

| | |
|---|---|
| *text* | Indicates names of manuals, menu commands, buttons, and fields. |
| `text` | Indicates text that you must type exactly as presented. Typically used to denote command, function, routine, or argument names that must be typed literally. |
| *`z_argument`* | Indicates text that you must replace with an appropriate argument value. The prefix (in this example, *`z_`*) indicates the data type the argument can accept and must not be typed. |
| \| | Separates a choice of options. |
| `{ }` | Encloses a list of choices, separated by vertical bars, from which you **must** choose one. |
| `[ ]` | Encloses an optional argument or a list of choices separated by vertical bars, from which you **may** choose one. |
| `[ ?argName` *`t_arg`* `]` | Denotes a *key argument*. The question mark and argument name must be typed as they appear in the syntax and must be followed by the required value for that argument. |
| `...` | Indicates that you can repeat the previous argument.<br>Used with brackets to indicate that you can specify zero or more arguments.<br>Used without brackets to indicate that you must specify at least one argument. |
| `,...` | Indicates that multiple arguments must be separated by commas. |
| `=>` | Indicates the values returned by a Cadence® SKILL® language function. |
| `/` | Separates the values that can be returned by a Cadence SKILL language function. |

If a command-line or SKILL expression is too long to fit within the paragraph margins of this document, the remainder of the expression is moved to the next line and indented. In code excerpts, a backslash ( \ ) indicates that the current line continues on to the next line.

# Identifiers Used to Denote Data Types

Data type identifiers are used to indicate the type of value required by an API argument. These data types are denoted by a single letter that is prefixed to the argument label and is separated from the argument by an underscore; for example, *t* is the data type in *t_viewName*. Data types and underscores are used only as identifiers; they must not be typed when specifying the argument in a function.

| Prefix | Internal Name | Data Type |
|---|---|---|
| *a* | array | array |
| *A* | amsobject | AMS object |
| *b* | ddUserType | DDPI object |
| *B* | ddCatUserType | DDPI category object |
| *C* | opfcontext | OPF context |
| *d* | dbobject | Cadence database object (CDBA) |
| *e* | envobj | environment |
| *f* | flonum | floating-point number |
| *F* | opffile | OPF file ID |
| *g* | general | any data type |
| *G* | gdmSpecIlUserType | generic design management (GDM) spec object |
| *h* | hdbobject | hierarchical database configuration object |
| *I* | dbgenobject | CDB generator object |
| *K* | mapiobject | MAPI object |
| *l* | list | linked list |
| *L* | tc | Technology file time stamp |
| *m* | nmpIlUserType | nmpIl user type |
| *M* | cdsEvalObject | cdsEvalObject |
| *n* | number | integer or floating-point number |
| *o* | userType | user-defined type (other) |
| *p* | port | I/O port |
| *q* | gdmspecListIlUserType | gdm spec list |

| Prefix | Internal Name | Data Type |
|---|---|---|
| *r* | defstruct | defstruct |
| *R* | rodObj | relative object design (ROD) object |
| *s* | symbol | symbol |
| *S* | stringSymbol | symbol or character string |
| *t* | string | character string (text) |
| *T* | txobject | transient object |
| *u* | function | function object, either the name of a function (symbol) or a lambda function body (list) |
| *U* | funobj | function object |
| *v* | hdbpath | hdbpath |
| *w* | wtype | window type |
| *sw* | swtype | subtype session window |
| *dw* | dwtype | subtype dockable window |
| *x* | integer | integer number |
| *y* | binary | binary function |
| *&* | pointer | pointer type |

For more information, see *Cadence SKILL Language User Guide*.

# 1

# 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.

## Installation

For SKILL-IPC to start a remote process, it must be able to locate the `cdsServIpc` program on the remote host. This is typically done by using the same filepath to the Cadence installation hierarchy on both the local and remote machines.

## cdsServIpc

`cdsServIpc` is a program that is started by the `ipcBeginProcess` function.

`cdsServIpc` uses `setpgid` to create a new process group but it remains in the same session as the parent process (the software from which it was started).

Therefore, the system resources used by the child processes will be included in the accounting for the parent process session.

**Note:** In releases prior to 06.05 (IC 6.1.0), `cdsServIpc` used `setsid` to create both a new session group and a new process group.

## cdsRemote

`cdsRemote` is used to start a program on a remote host.

`cdsRemote` Usage:

```
cdsRemote shell [-n] hostname command [args ...]
cdsRemote copy srcFilename hostname:destFilename
cdsRemote copy -r srcPath [...] hostname:destDirectory
```

By default, the `cdsRemote` program works in the `rsh` or `rcp` environments. However, it can be modified to use `ssh` by copying `<inst>/share/cdssetup/cdsRemsh/cdsRemote.scr` to a directory with a higher search precedence (such as, `$HOME/cdssetup/cdsRemsh/cdsRemote.scr` or `$CDS_SITE/cdssetup/cdsRemsh/cdsRemote.scr`) and then modifying the copy. For more information on the Cadence search mechanism, see Search Mechanism in the *Cadence Application Infrastructure User Guide*

If you only have the `ssh` setup, modify the `cdsRemote.scr` file and uncomment the `#remoteSystem=ssh` entry.

To use `ssh` with the DFII Distributed Processing feature, modify the `cdsRemote.scr` file and add the following in your `.cdsinit` file:

```
envSetVal("asimenv.distributed" "remoteShell" 'string "ssh")
```

This chapter covers the following sections:

# 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 an Operating System 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. When using the Windows Operating System, 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.

# 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.

# Cadence SKILL Development Tools

Information about the SKILL development tools is available in *Cadence SKILL IDE User Guide*.

The Walkthrough topic in this help system identifies and explains the tasks you perform when you develop SKILL programs using the SKILL development tools. 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 *Cadence SKILL IDE User Guide*.

# 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, which is accessible from the SKILL IDE window, CIW or from the UNIX command line.


For more information, see the *Cadence SKILL IDE User Guide*.

# 2

# Interprocess Communication 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:3
      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:4
      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 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:4
```

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 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.

**Note:** The maximum number of child processes is limited by the system resources and a warning message displays when the `fileDescriptor` limit is exceeded.

### 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.<br><br>If *tsu_dataHandler* is `nil`, the parent must use *ipcReadProcess* to read the data.<br><br>*tsu_dataHandler*, *tsu_errHandler* correspond to a child's `stdout` and `stderr` respectively.<br><br>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.<br><br>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 shell 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 when invoked from a shell. 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("hostname")
ipc:5
      ipcReadProcess(cid)
"foghorn\n"

      handler = (lambda (cid data) printf("\n Hostname:%s\n" data))
funobj:0x2848e8
      cid = ipcBeginProcess("hostname" "" handler)
ipc:6
Hostname: foghorn
```

**Note:** Single quotation marks can be used to enclose a group of characters which should be treated as a single word without shell interpretation of special characters.

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

Do not use single quotation marks for grouping commands that need to be run locally.

```
cid = ipcBeginProcess("cd /tmp; ls -l temp_ipcfile")
ipc:7
cid = ipcBeginProcess("'cd /tmp; ls -l temp_ipcfile'" "someHost")
ipc:8
```

### 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 Unix child process. Some commands will wait for the input channel to be closed before they complete, so this function allows for that to happen programmatically.

### Arguments

| | |
|---|---|
| *o_childId* | Child process handle. |

### Value Returned

### Example

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

```
      cid = ipcBeginProcess("wc")
ipc:3
      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

## Example

| | |
|---|---|
| `nil` | If the child has already expired, `nil` is printed. |
| `t` | Otherwise. |

```
      cid = ipcBeginProcess("ls -lR /")
ipc:4
      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:3
      ipcGetExitStatus(cid)
0
      cid = ipcBeginProcess("bad_command")
                           ; The command will cause an error
ipc:4
      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:7
      ipcGetPriority(cid0)           ; Priority of the child
5                                    ; process associates with
                                     ; 'cid0'


      ipcSetPriority(10)
t
      ipcGetPriority()
10
      cid1 = ipcBeginProcess("ls")
ipc:8
      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

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

## Example

```
      cid = ipcBeginProcess("ls -lR /")
ipc:3
      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:10
      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:11
      c2 = ipcBeginProcess("sleep 100")
ipc:12
      c1
ipc:11
      c2
ipc:12
      ipcKillAllProcesses()
t
      c1
ipc:11
      c2
ipc:12
```

## Reference

ipcKillProcess

# ipcKillProcess

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

## Description

Kills the 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

| | |
|---|---|
| t | Returns t if the child process is successfully killed. |
| nil | If the child has already expired. |

## Example

```
      cid = ipcBeginProcess("sleep 15")
ipc:3
      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("hostname")
ipc:3
      ipcReadProcess(cid)
"foghorn\n"
```

### Reference

ipcBeginProcess, ipcWriteProcess

## ipcSetPriority

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

### 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 readily than raise them. Some increase is permitted but even the lowest value given to *x_priorityChange* increases the priority from the norm by little.

The *x_priorityChange* value is not the absolute priority value that will be used to set the scheduling priority of a process. A 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

## ipcSignalProcess

```
ipcSkillProcess(
     o_childId
     s_signal
     )
     => t / nil
```

### Description

Sends the specified POSIX signal to the specified UNIX/Linux child process.

### Arguments

| | |
|---|---|
| *o_childId* | Child process handle obtained when the child process is launched. |
| *s_signal* | Symbol identifying the desired signal. It can have the following values:<br>`INT:` The interruption signal is sent to a process for requesting its interruption. Although the default POSIX behavior is for the process to be terminated, application-defined behavior may include discontinuing the current task and proceeding to the next task.<br>`TERM:` The termination signal is sent to a process for requesting its termination. Unlike the `KILL` signal, it can be caught and interpreted or ignored by the process. This allows the process to perform termination by releasing resources and saving the state, if appropriate.<br>`QUIT:` The quit signal is sent to a process for requesting its termination after performing a core dump. The core dump can be used in conjunction with a debugger to understand the state of the process when the `QUIT` signal was delivered.<br>`KILL:` The kill signal is sent when immediate process termination is required. Unlike `TERM` and `INT`, this signal cannot be caught or ignored, and the receiving process cannot perform any clean-up upon receiving this signal. For this reason, `TERM` is preferred. |

### Value Returned

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

### Example

```
Send SIGINT to a child process and observe the exitStatus

cid = ipcBeginProcess( "sleep 60" )
=>ipc:1

ipcSignalProcess( cid 'INT )
=>t

cid->exitStatus
```

```
130   ; subtract 128 to obtain the signal number (2 is a typical value for SIGINT)
```

## Reference

ipcSoftInterrupt, ipcCloseProcess, ipcKillProcess

## ipcSkillProcess

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

### Description

Invokes an Operating System 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.

**Note:** The maximum number of child processes is limited by the system resources and a warning message displays when the `fileDescriptor` limit is exceeded.

### 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.


***When using the Windows Operating System, 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.<br><br>If *tsu_dataHandler* is `nil`, the parent must use *ipcReadProcess* to read the data.<br><br>*tsu_dataHandler*, *tsu_errHandler* correspond to a child's stdout and stderr respectively.<br><br>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.<br><br>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 1

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:5
"Hello world"
      ipcReadProcess(cid)
"Executing (let () (println \"Hello world\") (1 + 1))"
      ipcReadProcess(cid)
"Result = (2)"
      cid->exitStatus
0
```

## Example 2

```
/*********************************************
* Example of ipcSkillProcess using a Perl script.
*********************************************/
=== Perl script ===

#!/usr/bin/perl
use IO::Handle;
use Fcntl;

# open descriptor 3 and ensure it flushes automatically
open(outPort, ">&3");
outPort->autoflush(1);
print outPort "(myTest)\n";

# open descriptor 4 and ensure it's non-blocking
open(inPort, "<&4");
fcntl(inPort,F_GETFL,$flags);
```

```
$flags|=O_NONBLOCK;
fcntl(inPort,F_SETFL,$flags);

# wait a bit and then read
sleep(2);
$inLine=<inPort>;
print "From Perl: $inLine\n";


=== SKILL script ===


procedure( testIpc()
   let( (child)
     printf("Executing ipc: %s\n" getCurrentTime())
     child=ipcSkillProcess("./test.perl")
     ipcWaitForProcess(child)
     printf("%s\n" ipcReadProcess(child 10))
   ) ;let
) ;procedure


procedure( myTest()
   prog( ()
      printf("Executed by Perl\n")
      return("123")
   ) ;prog
) ;procedure
```

## Reference

ipcBatchProcess, ipcBeginProcess

# ipcSleep

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

### Description

Causes the parent process to be delayed 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. |

### Value Returned

| | |
|---|---|
| `t` | Always returns `t`. |

### 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

ipcSleepMilli, ipcWait, ipcWaitForProcess

# ipcSleepMilli

```
ipcSleepMilli(
     x_time
     )
     => t
```

## Description

Causes the parent process to be delayed for the given number of milliseconds. Use the otherwise identical ipcSleep function to specify the delay in seconds.

## Arguments

| | |
|---|---|
| *x_time* | Number of milliseconds for the parent to sleep. |

## Value Returned

| | |
|---|---|
| t | Always returns t. |

## Example

The following example illustrates the performance characteristics for more, shorter delays as compared to fewer, longer delays.

```
> procedure(SleepPerformance(x_numIters x_milliPerIter)
    let( (measured)
        measured = measureTime(
            for(i 1 x_numIters ipcSleepMilli(x_milliPerIter)) )
        printf("for %d iterations of %dms, (userTime sysTime wallclock)
                is (%f %f %f)\n"
            x_numIters x_milliPerIter
            nth(0 measured) nth(1 measured) nth(2 measured) )
        ))

> (SleepPerformance 2000 50)
for 2000 iterations of 50ms, (userTime sysTime wallclock) is (0.018998 0.016997
100.167697)
t

> (SleepPerformance 50 2000)
for 50 iterations of 2000ms, (userTime sysTime wallclock) is (0.000999 0.001000
100.083947)
t
```

## Reference

ipcSleep, 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:15
      ipcSoftInterrupt(cid)
t
      cid
ipc:15
```

### 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:4
      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. The default timeout value is 1000000 seconds and the maximum is 2592000 seconds (= one month). |

## Value Returned

| | |
|---|---|
| `t` | Always returns `t`. |

## Example

```
      cid = ipcBeginProcess("sleep 30")
ipc:4
      ipcWait(cid)
            ; Suspends here until the child process terminates
t
```

## Reference

ipcSleep, ipcSleepMilli, 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("hostname")
ipc:6
      ipcWaitForProcess(cid)
                        ; Wait for the child process coming up
                        ; to guarantee a safe read
t
      ipcReadProcess(cid)
"foghorn\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:7
      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

# 3

# Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

This chapter describes the `phoIPC` functions, which are intended to help formalize the interface between an optical shape generator and Virtuoso. The shape generator is a standalone, exterior processor communicating through the IPC channel with Virtuoso. The `phoIPC` functions provide a set of Cadence-supported utilities to simplify some of the shape-generator integration tasks. In addition, these functions can help a PDK developer build some uniformity when using various third-party shape generators.

The functions in this chapter do not replace the basic IPC SKILL interface. These functions are also not intended to remove the need for the integrator of a shape generator to create integration code.

**Note:** To use the `phoIPC` functions, you need the `Virtuoso_Photonics_Option` license. For information about obtaining the required license, contact your local Cadence representative.

Only the functions documented in this chapter are supported for public use. Any other functions, regardless of their name or prefix, and undocumented aspects of the functions described below, are private and subject to change at any time.

## List of Photonic Interprocess SKILL Functions

# Server Registration and Check Functions

Use the functions defined in this section to define the name of the tool to be used by the `phoIPC` interface. The tool name is a string, and most `phoIPC` functions require it as an argument.

This section also describes the functions that you can register and use to check if the IPC channel is open and connected to the child process.

Basic messaging for the supported server registration and check functions is handled by the `phoIPC` infrastructure.

## List of Server Registration and Check Functions

phoIPCGetServerCheck

phoIPCRegisterServerCheck

phoIPCServerCheck

# phoIPCGetServerCheck

```
phoIPCGetServerCheck(
     t_toolName
     )
     => s_funcName / nil
```

### Description

Gets the name of the registered function that can be called to check if the server associated with the specified tool is operating correctly.

### Arguments

| | |
|---|---|
| *t_toolName* | Name of the tool. |

### Value Returned

| | |
|---|---|
| *s_funcName* | Name of the registered server check function. |
| nil | The command failed. |

### Example

```
checkFn = phoIPCGetServerCheck("myIPCTool")
```

### *Related Topics*

Server Registration and Check Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

## phoIPCRegisterServerCheck

```
phoIPCRegisterServerCheck(
     t_toolName
     s_serverCheckFunc
     )
     => t / nil
```

### Description

Registers a function that can be used to check if the server associated with the specified tool is operating correctly. The registered server check function is expected to have no argument. If the function is not specified, or it has an argument, registration fails. If the function is registered, it checks that the IPC channel is open and connected to the child process. Note that this function is intended to perform a quick check and is likely to be used inside a Pcell.

### Arguments

| | |
|---|---|
| *t_toolName* | Name of the tool. |
| *s_serverCheckFunc* | Name of the function to call to check if the server is operating correctly. |

### Value Returned

| | |
|---|---|
| t | The server check function is registered. |
| nil | The command failed. |

### Example

```
phoIPCRegisterServerCheck("phoIPCSample" 'myCheckFn)
```

### *Related Topics*

Server Registration and Check Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

## phoIPCServerCheck

```
phoIPCServerCheck(
     t_toolName
     [ ?cellViewID d_cellViewID ]
     )
     => t / nil
```

### Description

Calls the registered server check function for the specified tool, and issues a warning in the CIW if the server is not running. In addition, when a valid cellview is specified, the function creates a label at the origin that displays the same message in the layout canvas.

### Arguments

| | |
|---|---|
| *t_toolName* | Name of the tool. |
| ?cellViewID *d_cellViewID* | Name of the layout cellview ID in which a label is created on the canvas that displays the warning message. |

### Value Returned

| | |
|---|---|
| t | The registered server check function is called. |
| nil | The command failed. |

### Example

```
when(phoIPCServerCheck("phoIPCSample" pcCellView)
```

Checks that the processor IPC channel is open and that the server is working and ready to process any commands.

### *Related Topics*

Server Registration and Check Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

# Message Processor Functions

Use the functions defined in this section to process a server message that is returned in the *standard* `phoIPC` format.

A standard `phoIPC` message is a string containing a list of six sublists, as defined in Standard phoIPC Message Format. Each sublist in the message represents a different part of the element being built.

**Note:** The message returned by the shape processor must exist in the standard `phoIPC` format for it to be used with the `phoIPC` shape-processing infrastructure.

### *Related Topics*

Standard phoIPC Message Format

Generic Message Processor Example

## List of Message Processor Functions

phoIPCGetMessageProcessor

phoIPCProcessMarkers

phoIPCProcessPorts

phoIPCProcessServerMessage

phoIPCProcessShapes

phoIPCRegisterMessageProcessor

## phoIPCGetMessageProcessor

```
phoIPCGetMessageProcessor(
     t_toolName
     )
     => s_msgProcessor / nil
```

### Description

Gets the name of the message processor associated with the specified tool.

### Arguments

| | |
|---|---|
| *t_toolName* | Name of the tool. |

### Value Returned

| | |
|---|---|
| *s_msgProcessor* | Name of the message processor function associated with the specified tool. |
| nil | The command failed. |

### Example

```
msgProcessor = phoIPCGetMessageProcessor("myIPCTool")
```

### *Related Topics*

Message Processor Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

# phoIPCProcessMarkers

```
phoIPCProcessMarkers(
     d_cellviewID
     l_markerList
     s_displayToolName
     )
     => t / nil
```

## Description

Specifies the default marker processor for the specified tool and generates markers in the Annotation Browser assistant, categorizing the markers based on the specified markerList.

### Arguments

*d_cellviewID*

Name of the cellview.

*l_markerList*

List of sublists, where each sublist represents a marker object that will be created in the layout.

For more information, see Additional Information.

*s_displayToolName*

Specifies the tool name to use for tagging the markers and categorizing them in the Annotation Browser assistant.

The display tool name does not need to be the same as the tool name used internally for `phoIPC` registration functions.

### Value Returned

| | |
|---|---|
| `t` | The markers for the specified tool were generated in the Annotation Browser assistant. |
| `nil` | The command failed. |

### Additional Information

**`markerList`**

This is a list of sublists, where each sublist represents a marker object that is created in the layout Annotation Browser assistant. `markerList` is intended to convey visual messages to indicate issues, such as DRC violations, that are created by the shape generator as a result of the requested parameters for generation.

The syntax of a marker processor function is:

```
(
( s_severity s_msgText l_pointList [layerList] )
...
)
```

where:

- `severity` is categorized as: `error`, `info`, or `warning`.

  Any other values are defined as `error`.
- `msgText` is a string that describes the issue and is displayed as the description for the marker generated in the layout canvas.
- `pointList` is used to generate the shape for the marker object in the canvas.
- `layerList` (*optional*) is a list of layers associated with the marker. `layerList` is used to provide additional information in the Annotation Browser assistant.

### Example

```
phoIPCProcessMarkers(cv markers "My IPC Tool")
```

### *Related Topics*

Message Processor Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

## phoIPCProcessPorts

```
phoIPCProcessPorts(
     d_cellviewID
     l_opticalPortList
     l_portMap
     [ ?lppMap l_lppMap ]
     )
     => t / nil
```

### Description

Creates optical ports in the specified cellview using the specfied `portList`. This is the default optical port processor. For more information about the required format of the optical port list and the port map, which can be used to override the direction of a port, see Additional Information.

### Arguments

| | |
|---|---|
| *d_cellviewID* | Name of the cellview. |
| *l_opticalPortList* | List of sublists, where each sublist represents an optical port that will be created for the waveguide in the specified cellview.<br>By default, `portName` is used to determine the name for the optical net or terminal to be created. The port direction is determined from the argument, `l_portMap`.<br>For more information, see Additional Information. |
| *l_portMap* | List of sublists, where each sublist is used to specify the optical or electrical port name-to-direction mapping. For more information, see Additional Information. |
| *l_lppMap* | `lppMap` is an optional argument that is used to specify the layer, purpose, or layer-purpose pair mapping to be used for generating optical ports.<br>For more information, see Additional Information. |

### Value Returned

| | |
|---|---|
| `t` | Optical ports are created in the specified cellview. |
| `nil` | The command failed. |

### Additional Information

- **`opticalPortList`**

  `opticalPortList` is a list of sublists, where each sublist coincides with an optical port, and is represented in the following format:

```
(
( (n_width n_angle n_radius t_name) g_pointList t_layerName)
…
)
```

where:

- `width` is a float value that represents the waveguide width in user-defined units.
- `angle` is a float value that represents the waveguide facet angle in degree.
- `radius` is a float value that represents the bend radius for the waveguide at the port.
- `pointList` is the coordinate that represents the center of the port measured in user-defined units.
- `layerName` is the name of the waveguide layer.

- **portMap**

  `portMap` is an optional argument that is used to specify the optical or electrical port name-to-direction mapping. A `portMap` can be useful to determine the new direction to use for a port when integrating with a PDK to match the simulator. A `portMap` is a list of sublists, represented in the following format:

```
(
( t_serverPortName (t_newPortName [t_newPortDirection])
…
)
```

  where:

  - `serverPortName` is the name for the port as returned by the server in the original message.
  - `newPortName` is the new name for the port.
  - `newPortDirection` is a valid port direction. If a port direction is not specified, the default direction used is `inputOutput`.

- **lppMap**

  `lppMap` is a list of layers, purposes, or a sublist that establishes the associations between the supported layer-purpose pairs (LPP). The second value in the association list must be of the same type as the key.

  Mapping by layer-purpose pairs is given priority over mapping by layer names or purpose names.

  An `lppMap` can be represented in the following format:

```
list(
list( list("src_layer" "src_purpose") list("dst_layer" "dst_purpose") ) ;LPP
list("src_layer" "dst_layer") ;layer
list("src_purpose" "dst_purpose") ;purpose
```

```
);list
```

where:

- ("src_layer" "src_purpose") and ("dst_layer" "dst_purpose") represent the supported source and destination layer-purpose pair associations.
- "src_layer" and "dst_layer" represent the source and destination layer names.
- "src_purpose" and "dst_purpose" represent the source and destination purpose names.

### Example

```
phoIPCProcessPorts(cv1 portList1 portMap2 lppMap2)
```

Creates new ports in the cellview, cv1, using portList1 and portMap2.

### *Related Topics*

Message Processor Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

## phoIPCProcessServerMessage

```
phoIPCProcessServerMessage(
     d_cellviewID
     t_waveguideLayer
     t_serverMessage
     [ ?dropPorts { t | nil } ]
     [ ?portMap l_portMap ]
     [ ?lppMap l_lppMap ]
     )
```

### Description

Issues messages from the IPC processor, which can further be used in Pcells to create geometries.

### Arguments

*d_cellviewID*

Name of the cellview where the object needs to be created.

*t_waveguideLayer*

Default name for the waveguide layer.

*t_serverMessage*

Message returned by the IPC server.

*?dropPorts b_dropPorts*

Specifies whether or not the specified electrical or optical ports must be retained.

*?portMap l_portMap*

List of sublists, where each sublist is used to specify the optical or electrical port name-to-direction mapping.

For more information, see Additional Information.

*l_lppMap*

`lppMap` is an optional argument that is used to specify the layer, purpose, or layer-purpose pair mapping to be used for generating shapes and electrical pins.

For more information, see Additional Information.

### Value Returned

None

### Additional Information

- **portMap**

  `portMap` is an optional argument that is used to specify the optical or electrical port name-to-direction mapping. A `portMap` can be useful to determine the new direction to use for a port when integrating with a PDK to match the simulator. A portMap is a list of sublists, represented in the following format:

```
(
( t_serverPortName (t_newPortName [t_newPortDirection])
```

```
...
)
```

where:

- `serverPortName` is the name for the port as returned by the server in the original message.
- `newPortName` is the desired new name for the port.
- `newPortDirection` is a valid port direction specified as a string value. If a port direction is not specified, the default direction used is `inputOutput`.

■ **`lppMap`**

`lppMap` is a list of layers, purposes, or a sublist that establishes the associations between the supported layer-purpose pairs (LPP). The second value in the association list must be of the same type as the key.

Mapping by layer-purpose pairs is given priority over mapping by layer names or purpose names.

An `lppMap` can be represented in the following format:

```
list(
list( list("src_layer" "src_purpose") list("dst_layer" "dst_purpose") ) ;LPP
list("src_layer" "dst_layer") ;layer
list("src_purpose" "dst_purpose") ;purpose
);list
```

where:

- ("`src_layer`" "`src_purpose`") and ("`dst_layer`" "`dst_purpose`") represent the supported source and destination layer-purpose pair associations.
- `"src_layer"` and `"dst_layer"` represent the source and destination layer names.
- `"src_purpose"` and `"dst_purpose"` represent the source and destination purpose names.

### Example

```
phoIPCProcessServerMessage(pcCellView "waveguide" msg nil portMap lppMap2)
```

Creates the objects in the submaster. For a more detailed example illustrating the use of `phoIPCProcessServerMessage` and other related functions, see the generic message processor example.

***Related Topics***

Message Processor Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

## phoIPCProcessShapes

```
phoIPCProcessShapes(
     d_cellviewID
     l_shapeList
     l_portMap
     [ ?lppMap l_lppMap ]
     )
     => t / nil
```

### Description

Uses `shapeList` to create shapes and electrical pins in the specified cellview. This is the default shape processor. For more information, see Additional Information.

### Arguments

*d_cellviewID*

Name of the cellview in which the object needs to be created.

*l_shapeList*

List of sublists, where each sublist defines a point list that is used to create a polygon on the specified layer using the purpose `drawing`.

If the electrical port name is specified, it is used to create a pin with the corresponding polygon as the shape.

For the format of the `shapeList`, see Additional Information.

*l_portMap*

`portMap` is used to specify the optical or electrical port name-to-direction mapping.

For more information, see Additional Information.

*l_lppMap*

`lppMap` is an optional argument that is used to specify the layer, purpose, or layer-purpose pair mapping to be used for generating shapes and electrical pins.

For more information, see Additional Information.

### Value Returned

None

### Additional Information

- **shapeList**

  The required format for a `shapeList` is given below:

```
(
(  t_layerName l_pointList [t_portName]] )
…
)
```

where:

- `layerName` is the name of the waveguide layer.
- `pointList` is the list of points to be used for creating a polygon.
- `portName` is the name of the port as returned by the server in the original message.

- **portMap**

`portMap` is used to specify the optical or electrical port name-to-direction mapping. A `portMap` can be useful to determine the new direction to use for a port when integrating with a PDK to match the simulator. A portMap is a list of sublists, represented in the following format:

```
(
( t_serverPortName (t_newPortName [t_newPortDirection])
…
)
```

where:

- `serverPortName` is the name for the port as returned by the server in the original message.
- `newPortName` is the desired new name for the port.
- `newPortDirection` is a valid port direction specified as a string value. If a port direction is not specified, the default direction used is `inputOutput`.

- **lppMap**

`lppMap` is a list of layers, purposes, or a sublist that establishes the associations between the supported layer-purpose pairs (LPP). The second value in the association list must be of the same type as the key.

Mapping by layer-purpose pairs is given priority over mapping by layer names or purpose names.

An `lppMap` can be represented in the following format:

```
list(
list( list("src_layer" "src_purpose") list("dst_layer" "dst_purpose") ) ;LPP
list("src_layer" "dst_layer") ;layer
list("src_purpose" "dst_purpose") ;purpose
);list
```

where:

- ("src_layer" "src_purpose") and ("dst_layer" "dst_purpose") represent the supported source and destination layer-purpose pair associations.
- "src_layer" and "dst_layer" represent the source and destination layer names.
- "src_purpose" and "dst_purpose" represent the source and destination purpose names.

### Example

```
phoIPCProcessShapes(cv polygons portMap lppMap2)
```

Creates a polygon in the specified cellview.

### *Related Topics*

Message Processor Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

## phoIPCRegisterMessageProcessor

```
phoIPCRegisterMessageProcessor(
     t_toolName
     s_msgProcessor
     )
     => t / nil
```

### Description

Registers the message processor corresponding to the specified tool. In addition, the function ensures that the message processor function is defined, and has the right number of arguments.

### Arguments

*t_toolName*

Name of the tool.

*s_msgProcessor*

Name of the message processor function.

### Value Returned

| | |
|---|---|
| `t` | The message processor for the specified tool was registered. |
| `nil` | The command failed. |

### Example

```
phoIPCRegisterMessageProcessor("phoIPCSample" 'myMsgProc)
```

### *Related Topics*

Message Processor Functions

Photonic Interprocess Communication Functions (ICADVM20.1 Photonics Only)

# Standard phoIPC Message Format

The standard `phoIPC` message format is given below:

```
(
t_toolName
l_shapeList
l_opticalPortList
l_markerList
l_attributeList
l_netList
)
```

where:

- **`toolName`**

  `toolName` is a string. `phoIPC` parses the `toolName` string and retains the first element as the `toolName` used by the `phoIPC` functions.

- **`shapeList`**

  `shapeList` is a list of sublists. Each sublist in this list has the following format:

  ```
  (
  (  t_layerName l_pointList [t_portName]])
  …
  )
  ```

  The `shapeList` is used by the phoIPCProcessShapes function.

- **`opticalPortList`**

  An `opticalPortList` is a list of sublists, where each sublist coincides with an optical port, and is represented in the following format:

  ```
  (
  ( (n_width n_angle n_radius t_name) g_pointList t_layerName )
  …
  )
  ```

  where:

  - `width` is a float value that represents the waveguide width in user-defined units.
  - `angle` is a float value that represents the waveguide facet angle in degree.
  - `radius` is a float value that represents the bend radius for the waveguide at the port.
  - `pointList` is the coordinate that represents the center of the port measured in user-defined units.

- `layerName` is the name of the `waveguide` layer.

- **`markerList`**

  This is a list of sublists, where each sublist represents a marker object that is created in the layout Annotation Browser assistant. `markerList` is intended to convey visual messages to indicate issues, such as DRC violations, that are created by the shape generator as a result of the requested parameters for generation.

  The syntax of a marker processor function is:

```
(
( s_severity s_msgText l_pointList [layerList])
...
)
```

  where:

  - `severity` is categorized as: `error`, `info`, or `warning`.

    Any other values are defined as `error`.

  - `msgText` is a string that describes the issue and is displayed as the description for the marker generated in the layout canvas.

  - `pointList` is used to generate the shape for the marker object in the canvas.

  - `layerList` (*optional*) is a list of layers associated with the marker. `layerList` is used to provide additional information in the Annotation Browser assistant.

- **`attributeList`** and **`netlist`**

  Please contact your Cadence customer support representative, if you would like to use either of these two elements. Else, set them to `nil`.

**Note:** For each sublist described above, the `phoIPC` infrastructure provides a default handling function.

# Generic Message Processor Example

Here is a simple IPC message processor, built using the `phoIPC` sublist processing functions. The message processor requires the following five arguments:

- `cv`: Cellview in which the objects specified in the message need to be created.
- `msg`: Message containing the sublists. It must be formatted as specified in the Standard phoIPC Message Format.
- `designIntentLayer`: Default waveguide layer.
- `dropPort`: Boolean value, if set to `nil`, the electrical or optical port is not created. `dropPort` can be used when creating a hierarchical Pcell, where the lower-level ports are not used by the tool, and therefore, can be skipped (or *dropped*) for improved performance.
- `portMap`: List representing the port mapping information.

```
procedure( myMsgProc( cv msg designIntentLayer dropPorts portMap)
let( (polygons ports markers)
; get the relevant parts from the server message
polygons = car(msg)
ports = cadr(msg)
markers = caddr(msg)
; send each one to the appropriate default processor
phoIPCProcessShapes( cv polygons portMap)
phoIPCProcessPorts( cv ports portMap)
phoIPCProcessMarkers(cv markers "My IPC Tool")
)
)
```

### Example

Let us consider the following example that demonstrates the use of some functions described in this chapter. The code snippet displayed below is part of a Pcell code block and it illustrates at a high-level how the various `phoIPC` functions can be used.

```
; make sure the communication is open for the "phoIPCSample" processor
when( phoIPCServerCheck("phoIPCSample" pcCellView)
```

The server is started and working.

- Modify the parameters, as required.
- Prepare the call to the IPC server.

```
msg =
```

Calls the IPC server and gets the message back.

```
portMap =
```

Creates a `portMap`, if required.

```
phoIPCProcessServerMessage(pcCellView "waveguide" msg nil portMap)
) ; done
```

Creates the objects in the submaster.

# 4

# 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
int main(int argc, char* argv[])
{
    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)
    printf("Error %L Msg: %s\n" cid err)
    ipcKillProcess(cid) /*
)

procedure (simTerm(cid exit)
    printf("Simulator expired with exit status = %d\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 . ")
```