Carnegie Mellon University

School of Computer Science / Robotics Institute

Inter-Process Communication

A Reference Manual

Reid Simmons • Dale James

Manual Version: August 2011 (updated April 2022)
for IPC Version 3.9.1

Abstract

This manual is a programmer's guide to using the Inter-Process Communication (IPC) library, a platform-independent package for distributed network-based message passing. IPC provides facilities for both publish/subscribe and client/server type communications. It can efficiently pass complicated data structures between different machines, and even between different languages (currently, C/C++, Java, Python and LISP). IPC can run in either centralized-routed mode or direct point-to-point mode. With centralized routing, message traffic can be logged automatically, and there are tools available for visualizing and analyzing the message traffic.

Credits

The IPC was designed by Reid Simmons, a Research Professor in the School of Computer Science at Carnegie Mellon University. It is based on the communications infrastructure used by the Task Control Architecture (TCA), with changes needed to support the NASA New Millennium Program. The primary implementers of TCA were Christopher Fedor, Reid Simmons, and Richard Goodwin, although contributions were made by other members of Reid Simmons' research group. Trey Smith designed and implemented the `xdrgen` facility (see Appendix B).

Contacts

Questions about IPC, and suggestions for future releases may be addressed to:

Reid Simmons (reids@cs.cmu.edu)
Robotics Institute
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh PA 15213-3891

If you send a suggestion for the manual, or a correction to it, please be sure to specify the manual version, which is printed on the cover page. If you have a question about IPC, be sure to include the version number, which is printed when the central server is started.

The IPC mailing list is no longer functional (too much spam). To request to be informed when new IPC versions are created, please send email to reids@cs.cmu.edu..

Obtaining IPC Code

IPC is available via the IPC web page: http://www.cs.cmu.edu/~IPC. The "alpha" release is typically a stable release with the latest features and bug fixes, so feel assured using that version (although back versions are also available). The download contains the latest version of this manual, the IPC source code, installation instructions for most supported operating systems, the Comview visualization tool (has not been updated in many years), and the `xdrgen` tool for generating IPC format strings. In addition, the web site contains background information on IPC.

You can also retrieve IPC via anonymous ftp. Login to ftp.cs.cmu.edu as "anonymous" and use your email address as the password. "cd" to "project/TCA". That directory contains tarred and compressed copies of the latest IPC releases.

Contents

IPC Reference Manual

1 INTRODUCTION

The IPC (Inter-Process Communication) software package is designed to facilitate communication between heterogeneous control processes in a large engineered system. An important design principle for the IPC package was that it should provide sufficient functionality and flexibility to meet the needs of real-time autonomous systems, being robust and reliable without weighing the IPC implementation down with unnecessary "bells and whistles." IPC can be used by C, C++, Java, Python and LISP (currently Allegro and Lispworks) processes. It is supported on a number of different machine types (including Sun, SGI, x86, PPC, Rad6000, M68K) and operating systems (SunOS, Solaris, VxWorks, Linux, IRIX, Windows, MacOS).

An IPC-based system consists of an application-independent central server and any number of application-specific processes (see Figure 1). The central server is a repository for system-wide information (such as defined message names), and routes messages and logs message traffic. IPC also supports direct point-to-point communications between processes. The application-specific processes interface with the central server, and with each other, using a linkable library. The interface is the main subject of this manual.

The basic IPC communication package is quite simple: It is essentially a publish/subscribe model, where tasks/processes indicate their interest in receiving messages of a certain type, and when other tasks/processes publish messages, the subscribers all receive a copy of the message. Since message reception is asynchronous, each subscriber provides a callback function (a "handler") that is invoked for each instance of the message type. Tasks/processes can connect to the IPC network, define messages, publish messages, and listen for (and process) instances of messages to which they subscribe.

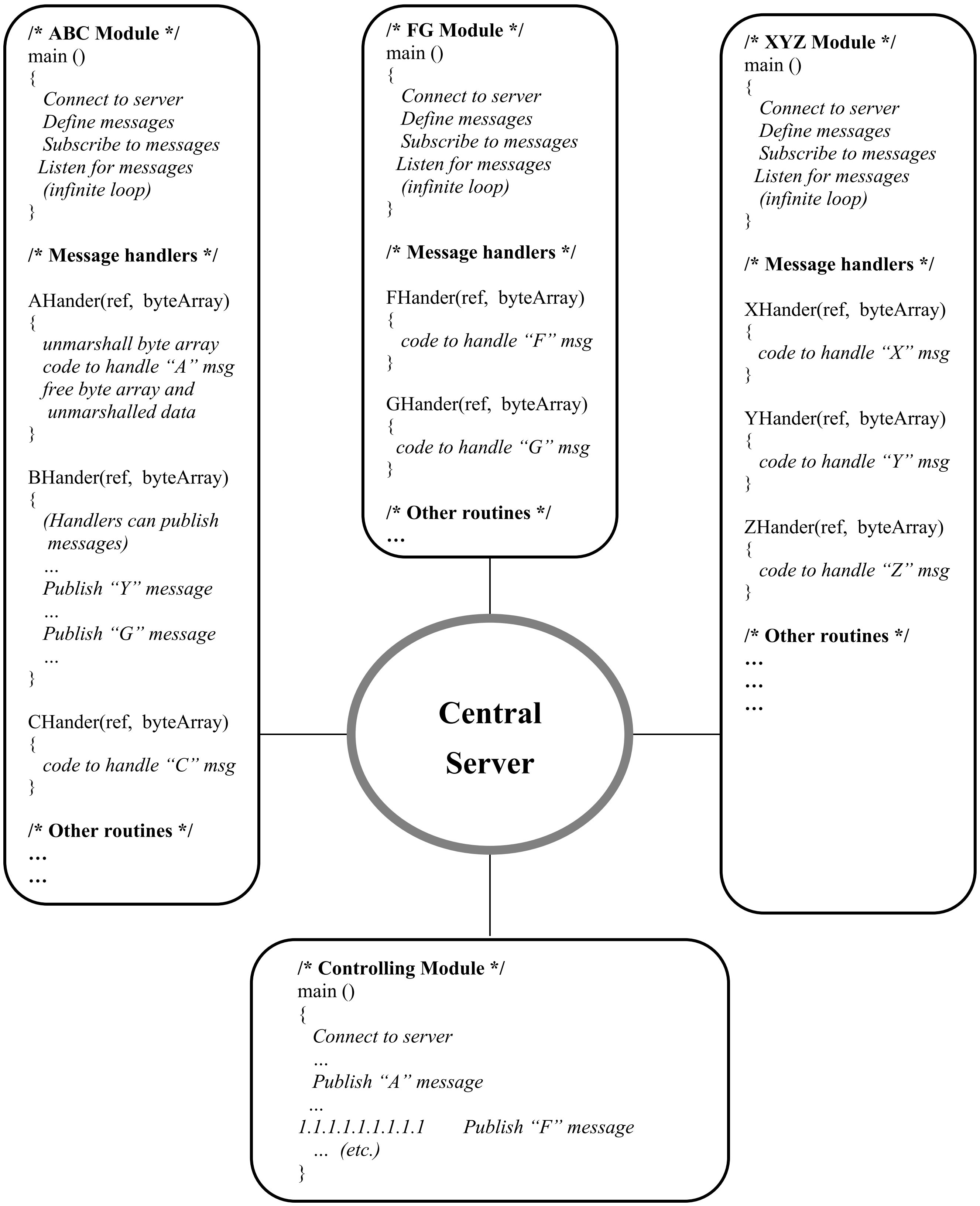
In addition to IPC message events, tasks/processes can indicate their interest in responding to other events (X window events, keyboard inputs, etc.), where such events can be characterized by input on a C-language "file descriptor" (fd). This provides needed functionality to implement more sophisticated event loops.

IPC also supports a version of the client/server paradigm: sending a directed response to a "query". Both blocking and non-blocking versions of this facility are provided. This facility should be used with caution, as query/response is typically not as safe a way of programming as pure publish/subscribe.

To facilitate passing messages containing complex data structures, IPC provides utilities for marshalling (serializing) a data structure (in any of the supported languages) into a byte stream, suitable for publication as a message, and for unmarshalling a byte stream back into a data structure in the appropriate language by the subscribing handlers. These facilities enable programs to transparently send a wide variety of data formats, including structures that include pointers (strings, variable length arrays, linked lists, etc.) to machines with possibly different byte orderings and packing schemes and to programs running different languages. It is recommended that, rather than sending byte-streams directly, these marshalling/unmarshalling functions be used as they may improve safety of the overall system (by dealing with byte ordering, packing, and non-flat data structures) with only a small penalty in added computation time and memory.

IPC can also be used to invoke a user-specified function at a specific time, or with a specified frequency. These "timer" capabilities enable a module to perform time-critical actions, or to dispatch events at specific times.

The IPC package supports message logging and message data logging. The *Comview* tool (see the Comview Reference Manual) can be used to visualize and analyze patterns of communication. However, the software has not been updated for a number of years, and may no longer work with current compilers/OS.



```
/* ABC Module */
main ()
{
   Connect to server
   Define messages
   Subscribe to messages
  Listen for messages
   (infinite loop)
}

/* Message handlers */

AHander(ref,  byteArray)
{
   unmarshall byte array
   code to handle "A" msg
   free byte array and
    unmarshalled data
}

BHander(ref,  byteArray)
{
   (Handlers can publish
    messages)
   ...
   Publish "Y" message
   ...
   Publish "G" message
   ...
}

CHander(ref,  byteArray)
{
   code to handle "C" msg
}

/* Other routines */
...
...
```

```
/* FG Module */
main ()
{
   Connect to server
   Define messages
   Subscribe to messages
  Listen for messages
   (infinite loop)
}

/* Message handlers */

FHander(ref,  byteArray)
{
   code to handle "F" msg
}

GHander(ref,  byteArray)
{
  code to handle "G" msg
}

/* Other routines */
...
```

```
/* XYZ Module */
main ()
{
   Connect to server
   Define messages
   Subscribe to messages
  Listen for messages
   (infinite loop)
}

/* Message handlers */

XHander(ref,  byteArray)
{
   code to handle "X" msg
}

YHander(ref,  byteArray)
{
   code to handle "Y" msg
}

ZHander(ref,  byteArray)
{
   code to handle "Z" msg
}

/* Other routines */
...
...
...
```

```
/* Controlling Module */
main ()
{
   Connect to server
   ...
   Publish "A" message
  ...
1.1.1.1.1.1.1.1.1       Publish "F" message
   ...  (etc.)
}
```

Figure 1. Sample Layout of an IPC-Based Distributed System

Installing IPC

For Linux, doing `make install` in the src directory to create `central` (installed in `bin/<OS>`, where `<OS>` is an identifier for your operating system) and `libipc.a` (installed in `lib/<OS>`). The Java interface to IPC can be created by doing `make install` in the Java directory, and similarly for Python in the python directory (you will need `SWIG` in order to make the Python version). To create libraries for LISP, do `make USE_LISP=1 install` in the src directory. In addition, for the Java interface, one needs to add the `USE_JAVA=1` directive to the `make` command. In addition, the `MAKE_SHARED_LIBS=1` directive makes shareable libraries (.so and .sa), the `THREADED=1` directive makes a thread-safe version of IPC, and the `DEBUG=NONE` directive makes a version of IPC without debugging symbols.

To run the Python version of IPC, set `PYTHONPATH` to include both the IPC `python` and `lib/<OS>` directories. To run the Java version, set `CLASSPATH` to include the `java/build` directory and set LD_LIBRARY_PATH to include the `lib/<OS>` directory.

For Windows, the `src/Windows` directory has Visual Studio projects for both central and the IPC library. IPC for Windows has been tested on XP, Vista, and Windows 7.

Finally, the test directory contains a number of test programs (both for Linux – look at the `GNUmakefile` – and for Windows – look at the `test/Windows` directory). One suggestion is to start with `sizesTest`, which tries to ensure that the marshalling functions understand your OS/compiler correctly. If it runs without printing out any error information, you should be able to send any data structure around without problems.

This manual serves as a central information resource for programmers building complex systems. Section 2 describes the central server and the process to start it. Section 3 explains how to describe basic data structures for message passing. Section 4 is a directory of the basic IPC interface function. Section 5 describes query/response functions that IPC provides. Section 6 details the IPC data-marshalling functions. Sections 0 and 8 describe IPC contexts and timers, respectively. Example programs are provided in the appendix.

2 THE CENTRAL SERVER

IPC uses an application-independent central server module to maintain system-wide information and to route and log message traffic. Before starting any modules, a program named "central" must first be started (Figure 2 lists command line options). The most basic service that the central server provides is message passing. A message sent from any module connected to the server will be forwarded by the server to the module containing the handler for the message (optionally, messages my be sent directly between application modules; see below).

More than one server can run on the same machine, using separate communication ports for each server. Having multiple servers is especially useful for software development – if independent developers must run their IPC servers on the same machine, there is a way to distinguish them.

On machines other than those running VxWorks, one can give commands to the central server while it is running, to display status or change some options.

Modules must first connect to the IPC central server using `IPC_connect`. They then can define messages, together with a description of their data formats, using `IPC_defineMsg`. Modules that want to handle messages must indicate their interest using `IPC_subscribeData` or `IPC_subscribe`. Definitions and subscriptions can be done in any order – including subscribing to a message before it is defined. When all messages have been thus registered, modules can publish messages using `IPC_publishData` or `IPC_publish`, and the appropriate message handlers will be invoked. Finally, before exiting a module, `IPC_disconnect` should be called, to cleanly disconnect from the network.

The environment variable *`CENTRALHOST`* must be set before starting the module. Specify the machine on which the central server is running:

```
setenv CENTRALHOST lung.learning.cs.cmu.edu
```

The default TCP socket port is 1381. If the desired server uses a different socket port (i.e., the **-p** option to **central** was used to start the server), the port number must be provided:

```
setenv CENTRALHOST lung.learning.cs.cmu.edu:1621
```

Starting a module after making this definition attempts to connect to TCP port 1621 of the host "lung....".

Access Control

[LINUX VERSION ONLY]

You may have an application where it would be beneficial to have the IPC central server be running all the time, but you are hesitant to do so because of possible security holes (e.g., someone from the outside could connect to the IPC port and potentially wreak havoc). To handle this, starting in version 3.7, the IPC central server provides an (optional) capability for access control.

Access control is a layer of network security that automatically denies connections from untrusted hosts, as described by `hosts_access` (5). In particular, portscanners are denied connections on access-controlled open sockets, and therefore cannot exploit potential bugs in network-level code, either at the user level or the system level. Connections are allowed by consulting the `/etc/hosts.allow` and `/etc/hosts.deny` files, where clients listed in `hosts.deny` but not in `hosts.allow` are automatically denied access.

Access control can be enabled in IPC by compiling with the `ACCESS_CONTROL` flag set:

```
% gmake ACCESS_CONTROL=1 install
```

Note that this only affects the central server.

The access control language is specified fully in the man page `hosts_access` (5), but in brief, the two files contain lines of the form

```
daemon : hostname
```

where `hostname` can be an IP address or a domain name. The daemon name used by the IPC central server is "`central`".

For instance, a typical `/etc/hosts.deny` control file might look like simply:

```
ALL : ALL
```

The following are central commands:

- **help**: print this message
- **display**: display the active and pending messages
- **status**: display the known modules and their status
- **memory**: display total memory usage
- **close** <module>: close a connection to a module
- **unlock** <resource>: unlock a locked resource

The following command line options can also be used as commands:

- **-v**: display server version information
- **-l**<option>: logging onto terminal. Options are:
  - **m** (message traffic)
  - **s** (status of IPC)
  - **t** (time messages are received by central)
  - **d** (data associated with messages)
  - **i** (ignore logging certain internal messages)
  - **h** (handle time summary of incoming messages)
  - **r** (log the reference ID as well as the message name)
  - **p** (log the reference ID of the message's parent)
  - **x** (no logging)

  -l (no options) is equivalent to **-lmstdh**; the default is **-lmsi**
- **-L**<option>: logging into file.
  Options are the same as above, with the addition of
  - **F** (don't flush file after each line)
  - **n** (don't prompt user for comments)

  The default is **-Lx**
- **-f**<filename>: filename to use for logging; If not specified, name is automatically generated.
- **-p**<port>: connect to central server on this port number.
- **-c**: Use direct (not via central) connections when possible
- **-I**<msgName>: Ignore logging this message (can occur multiple times).
- **-I**<filename>: File with names of messages to ignore logging
- **-s**: silent running; don't print anything to terminal.
- **-u**: don't run the user (tty) interface.
- **-r**: try resending non-completed messages when modules crash and then reconnect.

Figure 2. Starting the Central Server

and `/etc/hosts.`allow might look like:

```
central  : localhost
sshd     : .cs.cmu.edu
sshd-x11 : .cs.cmu.edu
```

This example denies access to all connections outside the `.cs.cmu.edu` domain, allows `ssh` connections inside `.cs.cmu.edu`, and allows IPC connections only on the local host. For message passing between computers, a second, comma-separated, hostname could be added to the entry for central:

```
central  : localhost, foo.cs.cmu.edu
```

Remember that the only affected program is `central`, and so access control only works when IPC connections are central mode, not in direct mode (direct connections are used by starting central with the **-c** option). However, since *all* IPC modules start by connecting with the central server, even peer-to-peer mode, coupled with access control, is pretty safe.

3 DEFINING MESSAGE DATA FORMATS

This section explains how to describe data structures that will be passed in messages. IPC can send raw byte arrays between processes, but it also provides a powerful data-marshalling facility that enables it to pass data *transparently* between processes, even if the hosts have different byte order or different alignment. To use the facility, one must specify the data formats used by each message. A programmer provides such a structural specification (called a "format string") in parameters to message definition routines. Once this is done, IPC can know how to convert the data structure to a byte stream and how to reconstruct it in the receiving module.

Suppose that the programmer needs to define a message called "SendData." It passes a single integer of data. He would use the following call to define it:

```
IPC_defineMsg("SendData",
              IPC_VARIABLE_LENGTH, "int");
```

Generally, however, one needs to pass more complicated data structures. Suppose that the message must pass a data structure containing an integer, a character string, and another integer. This call would register the message:

```
 IPC_defineMsg("SendData",
               IPC_VARIABLE_LENGTH,
               "{int, string, int}")
```

Rather than specifying data formats directly in message registration calls, we recommend first defining a data type, defining a format specifier for that type, and then using the format specifier in the message definition call:

```
 typedef struct {
   int x;
   char *y;
   int z;
 } DATA1_TYPE;
 #define DATA1_FORM "{int, string, int}"
 #define SEND_DATA_QUERY "SendData"
 IPC_defineMsg(SEND_DATA_QUERY,
               IPC_VARIABLE_LENGTH,
               DATA1_FORM);
```

Keeping the type definitions close to the format specifiers ensures that if one needs to make changes to a type, the corresponding definitions can be located and changed quickly. We also recommend that all message names be defined as macros to avoid the possibility of misspelling message names.

As in standard programming languages, IPC format strings are composed of primitive data type specifiers and composite specifiers that enable users to define more complex data types. The following sections describe both of these types of specifiers. Appendix B describes a program called `xdrgen` that automatically constructs IPC format strings from XDR type definitions.

IPC formats for Primitive Data Types

The previous example used the form "string" to stand for a list of characters. IPC also provides names for other data primitives; some of these names coincide with standard C language types, while others do not. Here is a complete list of primitives:

- char: an 8-bit piece of information, probably an ASCII code; this can be either *signed* or *unsigned*.
- byte: any 8-bit piece of information (signed or unsigned);
- short: any 16-bit piece of information (signed or unsigned);
- long: any 32-bit piece of information (signed or unsigned);
- int: 32 bits of information (signed or unsigned);
- float: 32 bits of information;
- double: 64 bits of information;
- Boolean: information that takes on one of two values: `TRUE` or `FALSE`. In C, 1 is `TRUE` and 0 is `FALSE`; Java uses `true` and `false`; Python uses `True` and `False`; LISP uses `T` and `NIL`. In the rest of the manual, we use Boolean (`TRUE` and `FALSE`) as shorthands for the above language-specific values.
- string: A list of characters–in C, this list is terminated by `NULL` ('\0');

A table of the various IPC formats, their equivalent LISP and C types, and (for C) size in bytes is given

Format Name	LISP Type	C Type	Bytes
“ubyte”	(unsigned-byte 8)	unsigned char	1
“byte”	(signed-byte 8)	signed char	1
“ushort”	(unsigned-byte 16)	unsigned short	2
“short”	(signed-byte 16)	signed short	2
“uint”	(unsigned-byte 32)	unsigned int	4
“int”	(signed-byte 32)	signed int	4
“ulong”	(unsigned-byte 32)	unsigned long	4
“long”	(signed-byte 32)	signed long	4
“float”	single-float	float	4
“double”	double-float	double	8
“boolean”		int	4

Figure 3. Primitive Data Types: Names and Lengths

in Figure 3. While some computers/compilers use different sizes for primitives (e.g., 64 bit longs), IPC currently supports only those formats listed. Generalizing IPC to handle different sized primitives is being contemplated.

IPC Formats for Composite Data Types

Composite data formats are aggregates of other data types. The supported composites include:

Structures

To describe a C language “struct” to IPC, surround the component data type names with braces, and place commas between them.

```
typedef struct {
  int x;
  char *str;
  int y;
} DATA_TYPE;
#define DATA_FORM "{int, string, int}"
#define USE_DATA_COMMAND "UseData"
IPC_defineMsg(USE_DATA_COMMAND,
              IPC_VARIABLE_LENGTH,
              DATA_FORM);
```

Structures can be nested. For example, a pair of `DATA_TYPE` components could be specified as follows:

```
"{{int, string, int},{int, string, int}}"
```

Unlike the other supported languages, Python does not support structures that have a fixed sequence of attributes. Thus, without further information, the marshalling/unmarshalling functions will not know which components correspond with which portions of the format string. To rectify this, one can define Python classes that have the `_fields` component, which is a tuple of the names (not the types) of each component, in order. In addition, for nested structures or embedded arrays, one can supply a tuple of `(<name>, <type>)` to provide additional information to help the marshalling functions allocate the correct instances. Subclassing off of IPCdata provides a print function that uses the `_fields` information to format the data structure nicely. For instance:

```
class DATA_TYPE(IPCdata) :
  _fields = ('x', 'str', ('y', int))
```

If the `_fields` information is not provided, IPC names the attributes `_f0`, `_f1`, etc.

Fixed-length and Variable-length Arrays

To describe a fixed-length array, use the following form:

```
"[ data_type: n ]"
```

`data_type` is the base type of the array, and `n` is the array dimension. If the array is multi-dimensional, separate the dimension numbers by commas, as in the following example:

```
typedef int array[17][42] DATA_TYPE;
#define DATA_FORM "[int:17, 42]"
```

Variable-length arrays are specified as part of a larger structure, which must contain "int" elements specifying dimensions. The notation

```
"<char: 1,2,3>"
```

indicates that a three dimensional array is contained in a larger data structure whose 1st, 2nd, and 3rd elements specify the dimensions of the array.

For example, a two-dimensional variable array of integers can be specified by placing it inside of the following structure:

```
typedef struct {
  int dimension1;
  int dimension2;
  int **variableArray;
} VARIABLE_ARRAY_TYPE;
```

The appropriate IPC format string would be:

```
#define VARIABLE_ARRAY_FORM
                "{int, int, <int: 1, 2>}"
```

Pointers, Linked Lists, Recursive Data Structures

Pointers are denoted by an asterisk followed by a data format name. If the pointer value is `NULL` (or `NIL` in LISP, or None in Python) no data is sent. Otherwise the data is sent and the receiving end creates a pointer to the data. Note that only the data is passed, not the actual pointers, so that structures that share structure or point to themselves (cyclic or doubly linked lists) will not be correctly reconstructed.

```
typedef struct {
  int x, *pointerToX;
} POINTER_EXAMPLE_TYPE;
#define POINTER_EXAMPLE_FORM
                "{int, *int}"
```

The "self pointer" notation, !*, is used in defining linked (or recursive) data formats. IPC will translate linked data structures into a linear form before sending and then recreate the linked form in the receiving module. IPC routines assume that the end of any linked list is designated by a `NULL` (or `None` or `NIL`) pointer value. Therefore it is important that all linked data structures be `NULL` terminated so that the data translation routines work correctly.

```
typedef struct _LIST {
  int x;
  struct _LIST *next;
} LIST_TYPE;
#define LIST_TYPE_FORMAT"{int,!*}"
```

While only C/C++ differentiates between objects and pointers to objects, the pointer format is still useful in Java, Python and LISP as a way of indicating whether data actually exists to be transmitted.

Enumerated Types

There are two forms for specifying an enumerated type. The basic format is "`{enum : <maxVal>}`", which indicates that the format is an enumerated type whose last element has the value `maxVal`. For example, the format string for "`typedef enum {A, B, C, D} ENUM_TYPE`" would be "`{enum : 3}`", since 3 is the implicit value of `D`. Similarly, the format for:

```
typedef enum{E=1,F=2,G=4,H=8} ENUM1_TYPE
```

would be "`{enum : 8}`". Note that this cannot be used for enumerated types that have negative values – for those types, you need to represent them using "`int`".

The alternate form for specifying an enumerated type includes the actual values themselves: "`{enum <enumVal0>, <enumVal1>, <enumVal2>, …, <enumValN>}`". For example, the format for `ENUM_TYPE` given above would be "`{enum A, B, C, D}`". There are two advantages of this form of specification: (1) the logs produced by the central server, and the output of `IPC_printData`, will contain the symbolic values of the enumeration, rather than just the integer values; (2) The LISP version will automatically convert the symbolic value to the associated integer (for C), and vice versa. The symbolic value is the upper-case version of `<enumVali>`, interned into the `:KEYWORD` package. For instance, a LISP module could send a message containing the atom `:B`, and a C-language module would receive the enumerated value “B” (which would have the integer value 1, given the example above). Note that you cannot use the alternate form if the type declaration explicitly sets the enumerated values (e.g., “{enum E, F, G, H}” will not correctly represent ENUM1_TYPE, given above). Python uses integers as enumerated values.

Of course, as with all of the other format specifiers, enumerated formats can be embedded in more complex format specifications:

```
“{int, {enum A, B, C, D},
       [double : 3], {enum : 10}}”
```

Another caveat: The colon (:) is a reserved symbol in the format specification language. You cannot use a colon in any of the enumerated values (same for braces, brackets, commas, and periods).

4 BASIC IPC INTERFACE FUNCTIONS

Types and function prototypes are defined in `ipc.h` for C users, IPC.java for Java users, IPC.py for Python users, and in `ipc.lisp` for LISP users. Unless otherwise indicated, all functions are available in all supported languages. The Python and LISP functions are all in the IPC package, and have identical names, arguments, and return types as their C equivalents, except where indicated. The Java functions are also in the IPC package but, for historical reasons, they currently do not have the "IPC_" prefix (it is anticipated that this will change in the near future).

4.1 Return Type

```
typedef enum {
   IPC_Error, IPC_OK, IPC_Timeout
} IPC_RETURN_TYPE
```

Return type for most IPC functions. If the return type is `IPC_Error`, the cause of the error will be indicated by the variable `IPC_errno` (4.11).

4.2 Describe Detectable Errors

```
typedef enum {
   IPC_No_Error,
   IPC_Not_Connected,
   IPC_Not_Initialized,
   IPC_Message_Not_Defined,
   IPC_Not_Fixed_Length,
   IPC_Message_Lengths_Differ,
   IPC_Argument_Out_Of_Range,
   IPC_Null_Argument,
   IPC_Illegal_Formatter,
   IPC_Mismatched_Formatter,
   IPC_Wrong_Buffer_Length,
   IPC_Communication_Error
} IPC_ERROR_TYPE
```

Type for describing the different types of errors that IPC can detect.

4.3 Define a Byte Array

```
typedef void *BYTE_ARRAY
```

An array of bytes (chars) that is passed around by the IPC communication functions. Not often needed, but for Python, can be created using `createByteArray(numBytes)`. Not currently implemented for Java or LISP.

4.4 Variable Length Byte Array

```
typedef struct {
   unsigned int length;
   BYTE_ARRAY content;
}IPC_VARCONTENT_TYPE,*IPC_VARCONTENT_PTR
```

Type used to represent a variable length array of bytes. Used to facilitate interfacing between the publish/subscribe functions and the marshalling/unmarshalling functions (Section 6).

For Python, can be created using `IPC_VARCONTENT_TYPE()` and accessed using `vc.length` and `vc.content`. Not currently implemented for Java or LISP.

4.5 Message Handler Type

```
typedef void (*HANDLER_TYPE)
                (MSG_INSTANCE msgInstance,
                 BYTE_ARRAY callData,
                 void *clientData)
```

The type of message handlers. MSG_INSTANCE is a predefined type that is not accessible to the user (although attributes of it can be accessed – see Section 4.37 and Section 6.5). `callData` is the content of the message, as sent via a "publish" invocation. `clientData` is a pointer to any user-defined data, and is associated with the message handler in the "subscribe" call (Section 4.26). Java version currently does not support client data.

4.6 Message Handler Type for Automatic Unmarshalling

```
typedef void (*HANDLER_DATA_TYPE)
                (MSG_INSTANCE msgInstance,
                 void *callData,
                 void *clientData)
```

The type of message handlers used with `IPC_subscribeData` (Section 4.27). Similar to HANDLER_TYPE (Section 4.5), except that the second argument is a pointer to the *unmarshalled* content of the message. Java version currently does not support client data.

4.7 Handler Type for Non-Message Events

```
typedef void (*FD_HANDLER_TYPE)
             (int fd, void *clientData)
```

The type of handlers for non-message events (e.g., X window events, keyboard input). `fd` is a C-language file descriptor. `clientData` is a pointer to any user-defined data, and is associated with the message handler in the "subscribe" call. Note that it is the responsibility of these types of event handlers to actually read the input that is on the `fd` file. Java version currently does not support client data. For Python, the `fileno()` method can be used to get the file descriptor from a Python file object.

4.8 Handler Type for Connection Notifications

```
typedef void (*CONNECT_HANDLE_TYPE)
             (const char *moduleName,
              void *clientData)
```

The type of handlers for notification of new modules connecting or disconnecting to the IPC server. `moduleName` is the name of the module that just connected/disconnected. `clientData` is a pointer to any user-defined data, and is associated with the handler in the "subscribe" call (see 4.42 and 4.43). Java version currently does not support client data.

4.9 Handler Type Subscription Notifications

```
typedef void (*CHANGE_HANDLE_TYPE)
             (const char *msgName,
              int numHandlers,
              void *clientData)
```

The type of handlers for notification of new subscriptions to messages. `msgName` is the name of the message that just had a subscriber added to, or removed from, it. `numHandlers` is the total number of handlers currently subscribed to that message. `clientData` is a pointer to any user-defined data, and is associated with the handler in the "subscribe" call (see 4.47). Java version currently does not support client data.

4.10 Describe Level of "Verbosity" for IPC Output

```
typedef enum {
   IPC_Silent,
   IPC_Print_Warnings,
   IPC_Print_Errors,
   IPC_Exit_On_Errors
} IPC_VERBOSITY_TYPE
```

Type for describing the different levels of "verbosity" that IPC supports. `IPC_Silent` produces no output; `IPC_Print_Warnings` prints only warning (but not error) messages; `IPC_Print_Errors` prints both warning and error messages; `IPC_Exit_On_Errors` prints warnings, and if an error occurs, prints the error message (using `IPC_perror`, see 4.12) and exits. The default verbosity is `IPC_Exit_On_Errors`. The verbosity level can be changed with `IPC_setVerbosity` (see 4.39).

4.11 Set Variable to Last Detected Error

```
IPC_ERROR_TYPE IPC_errno;
```

Variable set to the last error detected by an IPC function. Possible error values are given in Section 4.2. Initially set to `IPC_NO_ERROR`. Not currently available through the Java and Python interfaces.

4.12 Print Error Message

```
void IPC_perror (cont char *msg)
```

This function prints out the message `msg` to `stderr`, followed by a textual description of the current error (see also Sections 4.1 and 4.2).

4.13 Initialize IPC Data Structures

```
IPC_RETURN_TYPE IPC_initialize (void)
```

After initialization, one can parse format strings, and marshall and unmarshall data, but not define, subscribe or publish messages (which can only be done after connecting to the network, see 4.14 and 4.15). It is not necessary to call `IPC_initialize` before calling `IPC_connect` (4.15), but it is not an error to do so. This function always returns `IPC_OK`.

4.14 Connect to IPC Communication Network on a Given Host Machine

```
IPC_RETURN_TYPE IPC_connectModule
                    (const char *taskName,
                     const char *serverName)
```

Connect the task/process to the IPC communication network. `taskName` is used only for message logging purposes, and needs not be unique (although it is preferable to give each task a unique name). Connects to the central server running on the machine named `serverName` (see Section 2).

If `serverName` is `NULL`, use the machine defined by the environment variable `CENTRALHOST`. If `CENTRALHOST` is not set, tries to connect to the local machine.

`IPC_connectModule` returns `IPC_OK` if a connection is made or the task is already connected. If a connection cannot be made (e.g., if the central server task that manages communications is not responding), `IPC_Error` is returned and `IPC_errno` is set to `IPC_Not_Connected`.

4.15 Connect to IPC Communication Network

```
IPC_RETURN_TYPE IPC_connect
                    (const char *taskName)
```

Connect the task/process to the IPC communication network. Equivalent to:

`IPC_connectModule(taskName, NULL)`

(see Section 4.14).

4.16 Connect to IPC Communication Network without Listening

```
IPC_RETURN_TYPE
IPC_connectModuleNoListen
                    (const char *taskName,
                     const char *serverName)
IPC_RETURN_TYPE IPC_connectNoListen
                    (const char *taskName)
```

Most IPC modules listen periodically for message traffic (using IPC_dispatch, Section 4.36, or any of the IPC_listen variants, Sections 4.32-4.35). Some modules, though, merely publish and do not listen. In those cases, it is best to use one of the variants of IPC_connect above, to tell IPC that the module will not be listening periodically for incoming messages. This will prevent the *central* process from sending internal messages to the module. It may affect the functionality of point-to-point (direct) message communication, so this should be used only when absolutely necessary.

A warning is printed if a module subscribes to a message (Section 4.26) or file descriptor (Section 4.29) when in the “no listen” condition, since it is likely that the module will not receive incoming communications on a timely basis.

Not currently implemented in Java or LISP.

4.17 Disconnect from IPC Communication Network

```
IPC_RETURN_TYPE IPC_disconnect (void)
```

Disconnect the task/process from the IPC communication network. Any messages that the task/process subscribes to are unsubscribed, and the task can no longer listen for messages or events. This function provides tasks with a clean way of shutting down. Always returns `IPC_OK` (even if the task is not currently connected).

4.18 Is the IPC Network Connected

```
Boolean IPC_isConnected(void)
```

Determine if the task/process is currently connected to the IPC network (i.e., to the central server).

Returns a Boolean value, which depends on the language being used (see Section 3).

4.19 Is the Named Module Connected

```
Boolean IPC_isModuleConnected
```

```
                    (const char *moduleName)
```

Determine if the named module is currently connected to the IPC network (i.e., to the central server).

Returns a Boolean value, which depends on the language being used (see Section 3). In addition, the C version returns -1 on error (which can occur if the module invoking the function is not itself currently connected to the IPC server).

The LISP version is currently not implemented.

4.20 Register Message with IPC Network

```
IPC_RETURN_TYPE IPC_defineMsg
                   (const char *msgName,
                    unsigned int length,
                    const char *formatString)
```

Register a message with the IPC network. The message is referred to by its `msgName`. The `msgName` can be any valid string, although it is preferable (but not required) that it consist of alphanumeric characters, plus "-", "_" and "*". Message instances pass arrays of `length` bytes. `length` may be the constant `IPC_VARIABLE_LENGTH`, in which case each message instance can have a variable length byte array – which is published using `IPC_publish`, `IPC_publishVC`, or `IPC_publishData` (Section 4.22, Section 4.23, Section 6.12). `formatString` is used to provide information for use by the marshalling/unmarshalling functions (Section 6).

A message needs to be defined only once, in just one task/process. Its definition is propagated to all publishing/subscribing tasks (in particular, a module gets message information from the central server the first time it is published or subscribed, and then caches the definition). It is an error to define a message if it already exists *and* if `length` and `formatString` are different from the existing definition.

Typically a message is defined in the task that publishes the message. An exception is for messages that essentially are the "request" portion of a query/response pair of messages (e.g., there may be a pair of messages "`NAV_request_emphemeris`" and "`NAV_emphemeris`"). In such cases, the subscriber to the message (who is also the publisher of the response) typically defines both messages.

The function returns `IPC_Error` if the task is not currently connected to the IPC network (setting `IPC_errno` to `IPC_Not_Connected`).

The Java version does not take the `length` argument (it is always `IPC_VARIABLE_LENGTH`).

4.21 Is the Message Defined

```
Boolean IPC_isMsgDefined
                    (const char *msgName)
```

Determine whether some task/process has registered a message with the name `msgName`. Returns a Boolean value, which depends on the language being used (see Section 3).

Note that `FALSE` is also returned if an error occurs. In C, this can be distinguished from "not defined" by checking the value of `IPC_errno`: It is set to `IPC_Not_Connected` if the process is not connected to the central server, and is set to `IPC_Null_Argument` if `msgName` is `NULL`.

4.22 Publish a Message

```
IPC_RETURN_TYPE IPC_publish
                   (const char *msgName,
                    unsigned int length,
                    BYTE_ARRAY content)
```

Publish (broadcast) an instance of the message msgName, sending a copy of the `content` byte array to all subscribers of that message. `length` is the number of bytes of the array pointed to by `content`. This function can be used to publish fixed length messages: either by passing the constant `IPC_FIXED_LENGTH` as the `length` argument, or by passing the length provided when the message was defined.

The function returns `IPC_Error` if the task is not currently connected to the IPC network (setting `IPC_errno` to `IPC_Not_Connected`). It also returns `IPC_Error` if the message has not been defined (IPC_Message_Not_Defined), if the message is fixed length and the lengths are not equal (`IPC_Message_Lengths_Differ`), or if the message is variable length and the length argument is `IPC_FIXED_LENGTH` (`IPC_Not_Fixed_Length`).

There is no way for IPC to determine if `length` matches the actual number of bytes in the byte array.

Not implemented in Java or LISP – use IPC_publishData, instead; implemented in Python, but not really useful.

4.23 Publish a Variable Length Message

```
IPC_RETURN_TYPE IPC_publishVC
        (const char *msgName,
         IPC_VARCONTENT_PTR varcontent)
```

Equivalent to:

```
IPC_publish(msgName, varcontent->length,
            varcontent->content),
```

but, in addition, it returns `IPC_Error` if `varcontent` is `NULL` (setting `IPC_errno` to `IPC_Null_Argument`). Not implemented in Java – use `IPC_publishData`, instead.

4.24 Publish a Fixed-Length Message

```
IPC_RETURN_TYPE IPC_publishFixed
              (const char *msgName,
               BYTE_ARRAY content)
```

Equivalent to:

```
IPC_publish(msgName, IPC_FIXED_LENGTH,
            content)
```

Not implemented in Java or LISP; Implemented in Python, but not really useful.

4.25 Return Message Name

```
const char *IPC_msgInstanceName
             (MSG_INSTANCE msgInstance)
```

Return the message name of the given instance of the message.

4.26 Subscribe to Specific Message Type

```
IPC_RETURN_TYPE IPC_subscribe
               (const char *msgName,
                HANDLER_TYPE handler,
                void *clientData)
```

Indicate interest in receiving messages of type `msgName`. When a message instance of that type is received (Sections 4.22, 4.23, 4.24), the handler function is invoked and passed three arguments: an identifier of the specific message instance, the message content sent in the publish call, and the `clientData`, which is a pointer to any user-defined data (and which may be `NULL`). Java version currently does not support client data.

A given task may subscribe to a message before it is defined, and it may subscribe to the same message type multiple times: if the handler is the same as in a previous subscription, the new `clientData` replaces the old (in which case, a warning message is issued); if the `handler` differs from all other handlers for that message, it is added as an additional handler. This enables tasks to subscribe to a message for a specific purpose, and then unsubscribe (Section 4.28) after some period of time, without impacting the rest of the task.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC communication network (`IPC_Not_Connected`)

Not implemented in Java – use `IPC_subscribeData`, instead.

4.27 Subscribe to Specific Message Type with Automatic Unmarshalling

```
IPC_RETURN_TYPE IPC_subscribeData
             (const char *msgName,
              HANDLER_DATA_TYPE handler,
              void *clientData)
```

Indicate interest in receiving messages of type `msgName`. When a message instance of that type is received(Sections 4.22, 4.23, 4.24), the `handler` function is invoked and passed three arguments: an identifier of the specific message instance, the unmarshalled message content sent in the publish call, and the clientData, which is a pointer to any user-defined data (and which may be NULL). Java version currently does not support client data.

IPC_subscribeData function behaves identically to IPC_subscribe, except that, when invoked, the handler is passed *unmarshalled* data, rather than a raw byte array. It is still the user's responsibility to free the data (preferably using IPC_freeData, Section 6.10).

The Java and Python interfaces enable one to specify the class type of the data that will be received. For Java, this is the required third argument (no clientData argument); For Python, it is an optional fourth argument – if the class is not specified (or None), a structure of type IPCdata will be created.

The function returns IPC_Error if the task/process is not currently connected to the IPC communication network (IPC_Not_Connected)

4.28 Unsubscribe to Specific Message Type

```
IPC_RETURN_TYPE IPC_unsubscribe
                (const char *msgName,
                 HANDLER_TYPE handler)
```

Indicate that the task/process is no longer interested in having the handler invoked on messages of the given type. Note that if a task/process subscribes to multiple handlers for that message, only the specified handler is removed. If handler is NULL, then all handlers for that message type, subscribed to by that task, are removed (thus, that task will no longer receive any messages of that type). It is not an error to unsubscribe a handler that is not currently subscribed. Message instances that have been published, but not received, when the handler is unsubscribed will not be processed by that handler.

The function returns IPC_Error if the task/process is not currently connected to the IPC communication network (IPC_Not_Connected)

4.29 Integrate Non-Message Event Handling with Message Event Handling

```
IPC_RETURN_TYPE IPC_subscribeFD
                (int fd,
                 FD_HANDLER_TYPE handler,
                 void *clientData)
```

Some tasks/processes need to handle events other than IPC message traffic (e.g., X events, keyboard input, RS232 input from hardware devices). IPC_subscribeFD is used to integrate such additional event handling with the IPC message event handling. fd is a C-language file descriptor that can be used in a select system call. clientData is the same as in IPC_subscribe (Section 4.26, Java version currently does not support client data). The handler for an fd-event is invoked with the file descriptor that raised the event and the clientData. Note that the data which is on the file descriptor is *not* read by the IPC – it is up to the handler to read that data, or to handle it in the appropriate manner (e.g., read and parse tty input, invoke a standard X event handler function).

This function always returns IPC_OK. Since it is up to the handler function to determine how to handle input on the file descriptor, it does not make sense for multiple handlers to subscribe to the same fd input. Note that this is contrary to the behavior of IPC_subscribe: IPC_subscribeFD will never have more than one handler per file descriptor. If an additional handler is subscribed for an fd, it will replace the old handler (and old client data).

4.30 Unsubscribe to File-Descriptor Type

```
IPC_RETURN_TYPE IPC_unsubscribeFD
                (int fd,
                 FD_HANDLER_TYPE handler)
```

Similar to IPC_unsubscribe (Section 4.28), except that the handlers are associated with file descriptor (fd) events, rather than with message types.

The function always returns IPC_OK.

4.31 Get the Open IPC Sockets

```
fd_set IPC_getConnections(void)
```

Returns the set of file descriptors that represent all the communication sockets currently open within IPC. This set may change over time as new modules connect to the system or as messages are sent to, or received from, other modules (especially when messages are sent peer-to-peer).

The function returns the empty set (all zeroes) if IPC is not connected.

4.32 Listen for Subscribed Events

```
IPC_RETURN_TYPE IPC_listen
                (unsigned int timeoutMSecs)
```

Listen for events (messages or other fd events) that have been subscribed to. The appropriate handlers are invoked for each message instance, or other event, received. The function returns IPC_Error if the task/process is not currently connected (IPC_Not_Connected). It returns with IPC_Timeout if timeoutMSecs pass without the task having received an event. The function returns, with IPC_OK, immediately after handling an event. Actually, if several events arrive simultaneously, or several events are waiting when IPC_listen is invoked, then they will *all* be handled before the function returns – but events that arrive after event handling begins will *not* be handled within that invocation of IPC_listen. The predefined constant IPC_WAIT_FOREVER indicates that the listen call should never time out.

4.33 Listen for Queued Subscribed Events

```
IPC_RETURN_TYPE IPC_listenClear
                (unsigned int timeoutMSecs)
```

A message can still be waiting when IPC_listen returns if it arrives while the IPC_listen is handling another message. To ensure that no messages are in the queue, use IPC_listenClear, which is roughly equivalent to:

```
if (IPC_listen(timeoutMSecs) !=
    IPC_Timeout)
  while (IPC_listen(0) != IPC_Timeout)
```

If the first call to IPC_listen does not time out, the function will continue listening for messages until there are none (note that a timeout of zero milliseconds means to return immediately, unless an event is already waiting). The function returns IPC_Error if the task/process is not currently connected (IPC_Not_Connected).

4.34 Listen for Given Amount of Time

```
IPC_RETURN_TYPE IPC_listenWait
                (unsigned int timeoutMSecs)
```

This function handles messages until at least timeoutMSecs have passed. It is a bit like the UNIX "sleep" function, except that it will handle messages while it waits. It differs from IPC_listenClear in that it will continue to wait the requested time, even if there are no more messages to process.

Note that the function may take longer than timeoutMSecs to return if it is in the middle of processing a message. The function returns IPC_Error if the task/process is not currently connected (IPC_Not_Connected).

4.35 Handle One IPC Event

```
IPC_RETURN_TYPE IPC_handleMessage
                (unsigned int timeoutMSecs)
```

Handle a single IPC message or external event. Return after either (a) the message/event was handled or (b) timeoutMSecs have passed.

IPC_Error is returned if the task/process is not currently connected (IPC_Not_Connected). The function returns with IPC_Timeout if timeoutMSecs pass without the task having received an event.

4.36 Enter Infinite Dispatch Loop

```
IPC_RETURN_TYPE IPC_dispatch (void)
```

IPC_dispatch is essentially equivalent to:

```
while (IPC_listen(IPC_WAIT_FOREVER) !=
IPC_Error)
```

It returns (with `IPC_Error`) only if the task/process is not connected to the IPC network (`IPC_Not_Connected`).

4.37 Return Message Size

```
unsigned int IPC_dataLength
                (MSG_INSTANCE msgInstance)
```

A message handler receives three arguments: A pointer to the message instance, a byte array of message data, and client data. This function takes as its argument the message instance and returns the number of bytes in the message data (which may be zero or greater). This function may be useful when the message is a variable length message, but the user does not want to (or cannot) unmarshall it into a known data structure.

4.38 Enable Receiving Multiple Messages

```
IPC_RETURN_TYPE IPC_setCapacity
                    (int capacity)
```

When used in the mode in which a central server routes messages, the server by default sends a module only one message at a time. There are situations in which this may produce undesired latencies. `IPC_setCapacity` can be used to change the default behavior, causing the central server to send up to `capacity` messages at a time to the module (where they will be queued on the socket until handled).

Warning: `capacity` should not be set too high, especially if large messages are being sent, as the central server could possibly become blocked if the socket/pipeline to the module becomes full. Typically, a capacity of 2-4 is sufficient for all purposes.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`), or if capacity is less than 1 (`IPC_Argument_Out_Of_Range`).

4.39 Select Level of “Verbosity” for Module Output

```
IPC_RETURN_TYPE IPC_setVerbosity
             (IPC_VERBOSITY_TYPE verbosity)
```

Set the IPC current “verbosity” level of the module. Affects if, and how, warning and errors are reported. The function returns `IPC_Error` (with `IPC_errno` set to the value `IPC_Argument_Out_Of_Range` if verbosity is not a legal value of `IPC_VERBOSITY_TYPE` (Section 4.10).

4.40 Set Priority for Message Instances

```
IPC_RETURN_TYPE IPC_setMsgPriority
                    (char *msgName,
                     int priority)
```

This function sets the priority (an integer value) for all instances of the given message name. The messages are queued, and dispatched, according to priority value. All messages with the same priority value are queued in order of receipt. Messages that have not been explicitly assigned a priority value are assumed to be at the lowest priority.

As of IPC 3.2, this function works both for messages that are queued within IPC central, as well as for messages that are sent directly, with direct module-to-module communications. This function returns an error if priority is less than zero (`IPC_Argument_Out_Of_Range`) or if IPC is not connected (`IPC_Not_Connected`).

4.41 Set Message Queue Length

```
IPC_RETURN_TYPE IPC_setMsgQueueLength
                    (char *msgName,
                     int length)
```

This function sets the maximum queue length (an integer value) for instances of the given message name. If `length` messages of the given type are already queued for a module, and a new message of that type arrives, then the oldest message is discarded in order to maintain the maximum queue length.

This function works for both centrally queued messages and point-to-point messages. Currently, there appears to be a bug if `length` is zero. This function returns an error if priority is less than one (`IPC_Argument_Out_Of_Range`) or if IPC is not connected (`IPC_Not_Connected`).

4.42 Notify of New Connections

```
IPC_RETURN_TYPE IPC_subscribeConnect
                (CONNECT_HANDLE_TYPE handler,
                 void *clientData)
```

Invoke `handler` whenever a new module connects to the IPC server. The `handler` is invoked with the name of the connecting module and the `clientData` (see 4.8). Note that the handler is invoked only for modules that connect *after* the subscription – if a module is already connected, no notification is given (you can use `IPC_isModuleConnected`, Section 4.19, for that purpose). If the function is called with same handler, the old client data is replaced with the new `clientData`, but the handler is invoked only once per connection. Java version currently does not support client data.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`).

4.43 Notify of New Disconnections

```
IPC_RETURN_TYPE IPC_subscribeDisconnect
                (CONNECT_HANDLE_TYPE handler,
                 void *clientData)
```

Invoke `handler` whenever a module disconnects from the IPC server (either because the module exited or because it explicitly called `IPC_disconnect`). The `handler` is invoked with the name of the connecting module and the `clientData` (see 4.8). If the function is called with same handler, the old client data is replaced with the new `clientData`, but the handler is invoked only once per disconnection. Java version currently does not support client data.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`).

4.44 Unsubscribe to Connection Notifications

```
IPC_RETURN_TYPE IPC_unsubscribeConnect
                (CONNECT_HANDLE_TYPE handler)
```

Tells IPC to no longer invoke `handler` when a new module connects to the IPC server. Note: Does not free the `clientData` associated with the handler (see 4.42) – that is up to the user.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`).

4.45 Unsubscribe to Disconnection Notifications

```
IPC_RETURN_TYPE IPC_unsubscribeDisconnect
                (CONNECT_HANDLE_TYPE handler)
```

Tells IPC to no longer invoke `handler` when a module disconnects from the IPC server. Note: Does not free the `clientData` associated with the handler (see 4.43) – that is up to the user.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`).

4.46 Number of Current Subscribers

```
IPC_RETURN_TYPE IPC_numHandlers
                (const char *msgName)
```

Returns the number of handlers currently subscribed to message `msgName`. The function returns zero (0) if the message is not currently defined.

The function returns -1 on error. The error conditions include if the module is not currently connected to the IPC network (`IPC_Not_Connected`) or if `msgName` is null (`IPC_Null_Argument`).

4.47 Notify of New Subscribers

```
IPC_RETURN_TYPE
IPC_subscribeHandlerChange
            (const char *msgName,
             CHANGE_HANDLE_TYPE handler,
             void *clientData)
```

Tells IPC to invoke `handler` whenever the subscription information changes for message `msgName`, that is, whenever some module either subscribes to receive instances of the message, or unsubscribes to the message. The handler is invoked with the name of the message, the total number of handlers currently subscribed for that message, and the user-define `clientData` (see 4.9, Java version currently does not support client data). Note that the handler is not invoked for any current subscriptions – only for those that are added or removed *after* this function is invoked. To determine the number of handlers currently subscribed, you can use `IPC_numHandlers` (see 4.46). If the function is called with same handler, the old client data is replaced with the new `clientData`, but the handler is invoked only once per change in subscription status.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`), if `msgName` is null (`IPC_Null_Argument`), or if the message is not currently defined (`IPC_Message_Not_Defined`). Note that, in particular, it is not currently possible to use this function on messages that have not been defined (via `IPC_defineMsg`). This is a limitation that may be lifted in the future, especially if any IPC user feels a need for it.

4.48 Unsubscribe to Subscription Notifications

```
IPC_RETURN_TYPE
IPC_unsubscribeHandlerChange
             (const char *msgName,
              CHANGE_HANDLE_TYPE handler)
```

Tells IPC to no longer invoke `handler` when the subscription information changes for message `msgName`. Note: Does not free the `clientData` associated with the handler (see 4.47) – that is up to the user.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`), if `msgName` is null (`IPC_Null_Argument`), or if the message is not currently defined (`IPC_Message_Not_Defined`).

4.49 Shut Down Central Server

```
void killCentral (void)
```

[VXWORKS VERSION ONLY]

This function, which is meant to be invoked from the VxWorks shell, cleanly shuts down the central server, closing all sockets and file descriptors. This enables the central server to be restarted, without having to reboot the real-time board.

Implementationally, the task id of the central server is saved at startup, and `killCentral` sends a `SIGTERM` to that task. The same functionality can be had by using "`i`" to print a list of active tasks, then doing "`kill 0x<taskid>,15`" (15 is the value of `SIGTERM`).

4.50 Shut Down Specific Task

```
void killModule (char *taskName)
```

[VXWORKS VERSION ONLY]

This function, which is meant to be invoked from the VxWorks shell, cleanly shuts down the named task, closing all sockets and file descriptors. This tells the central server that the task has disconnected, and enables the task to be restarted without having to reboot the real-time board.

Implementationally, the task id is looked up from the task name, and `killModule` sends a `SIGTERM` to that task. The same functionality can be had by using "`i`" to print a list of active tasks, then doing "`kill 0x<taskid>,15`" (15 is the value of `SIGTERM`).

5 QUERY/RESPONSE

While there is evidence that pure event-driven (publish/subscribe) systems are more reliable and maintainable than those that include query/response (client/server), it is also difficult to restructure existing code to fit this paradigm. Thus, the IPC contains functions for query/response, but it is recommended that they be used with caution (in particular `IPC_queryResponse`, the blocking form of query/response).

5.1 Reply to a Query Message

```
IPC_RETURN_TYPE IPC_respond
                (MSG_INSTANCE msgInstance,
                 const char *msgName,
                 unsigned int length,
                 BYTE_ARRAY content)
```

Similar to `IPC_publish`, except that it sends the message `msgName` *directly* to the task that sent the message represented by `msgInstance` (where `msgInstance` is the first argument of a message handler). The receiving task should be expecting a response by having invoked `IPC_queryNotify` or `IPC_queryResponse`. Note that IPC_respond will not trigger any other handlers that subscribe to that message type, either in the same task or different tasks.

For example, suppose task "`A`" includes the following code:

```
IPC_subscribe("foo", fooHandler, NULL);
IPC_queryNotify("bar", length, content,
                fooResponse, NULL);
```

and suppose task "`B`" includes subscribes the following handler to receive message "`bar`":

```
void barHandler
          (MSG_INSTANCE msgInstance,
           BYTE_ARRAY callData,
           void *clientData)
{
  IPC_respond(msgInstance, "foo",
              length, bar1(callData));
  free(callData);
}
```

When task "`A`" publishes "`bar`" (via `IPC_queryNotify`), `barHandler` will be invoked in task "`B`" (via `IPC_dispatch` or `IPC_listen`). Task "`B`" responds to the request by computing some result (function `bar1`), and sending a directed response back to task "`A`". In task "`A`", only the `fooResponse` handler will be invoked – the `fooHandler` function will *not* be invoked in this situation, even though it subscribes to "`foo`" messages, in general.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`). It returns `IPC_Error` if the message has not been defined (`IPC_Message_Not_Defined`), if the message is fixed length and the lengths are not equal (`IPC_Message_Lengths_Differ`), or if the message is variable length and the length argument is `IPC_FIXED_LENGTH` (`IPC_Not_Fixed_Length`).

Not implemented in Java – use `IPC_respondData`, instead.

5.2 Enable Replies Outside a Handler

```
IPC_RETURN_TYPE IPC_delayResponse
                (MSG_INSTANCE msgInstance)
```

Typically, IPC considers a message has been completed when a handler returns. In particular, the message instance passed to that handler is reclaimed. Occasionally, one needs to respond to a query message outside of the handler – for instance, a query handler might set up something to monitor a piece of hardware and return a result when it becomes available.

To prevent IPC from assuming that the message is completed, one should invoke this function before exiting the handler. In this way, when `IPC_respond` (Section 5.1) is invoked with that message instance, IPC will consider the message completed and reclaim the message instance.

The function returns `IPC_Error` if the message instance is `NULL` or invalid (`IPC_Null_Argument`).

5.3 Await Response to a Query

```
IPC_RETURN_TYPE IPC_queryNotify
                 (const char *msgName,
                  unsigned int length,
                  BYTE_ARRAY content
                  HANDLER_TYPE handler,
                  void *clientData)
```

Set up the handler to await the response to the (query) message `msgName`. Assumes that the receiver of `msgName` will use a call to `IPC_respond` to direct the response. The handler is invoked exactly as any other message handler – with the message instance identifier, call data, and client data. Java version currently does not support client data.

This function is *non-blocking*. The handler is invoked asynchronously, from within an `IPC_dispatch` or `IPC_listen` invocation.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`). It returns `IPC_Error` if the message has not been defined (`IPC_Message_Not_Defined`), if the message is fixed length and the lengths are not equal (`IPC_Message_Lengths_Differ`), or if the message is variable length and the length argument is `IPC_FIXED_LENGTH` (`IPC_Not_Fixed_Length`).

Not implemented in Java – use `IPC_queryNotifyData`, instead.

5.4 Send a Query and Block Waiting

```
IPC_RETURN_TYPE IPC_queryResponse
              (const char *msgName,
               unsigned int length,
               BYTE_ARRAY content,
               BYTE_ARRAY *replyHandle,
               unsigned int timeoutMSecs)
```

Sends the (query) message `msgName`, and blocks waiting for a response (sent via `IPC_respond`) to that particular message instance. When the response is received, sets the `replyHandle` to point to the data contained within the response. Returns `IPC_Timeout` if the reply has not been received within the specified interval.

Note that although this function is *blocking*, the calling task can still process other messages while it is awaiting the response. Thus, the state of the task/process may change during the time the function is invoked! For this reason, this function should be used with extreme caution – either check the local state when the function returns, or somehow guarantee that the local state you depend on will not be altered by any other message that you could receive during that time.

The function returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`), if the message has not been defined (`IPC_Message_Not_Defined`), if the message is fixed length and the lengths are not equal (`IPC_Message_Lengths_Differ`), or if the message is variable length and the `length` argument is `IPC_FIXED_LENGTH` (`IPC_Not_Fixed_Length`).

Not implemented in Java – use `IPC_queryResponseData`, instead. For Python, the fifth argument (`replyHandle`) is not used; instead, the function returns a tuple of two values – the reply byte-array and the `IPC_RETURN_TYPE` value.

5.5 Respond with a Variable Length Message

```
IPC_RETURN_TYPE IPC_respondVC
               (MSG_INSTANCE msgInstance,
                const char *msgName,
                IPC_VARCONTENT_PTR varcontent)
```

Equivalent to `IPC_respond` (Section 5.1), except that it uses a pointer to a structure that includes both the length and content of the message data. Designed to facilitate interfacing with the marshalling/unmarshalling functions. In addition to the return values of `IPC_respond`, it returns `IPC_Error` if varcontent is `NULL` (`IPC_Null_Argument`).

5.6 Await a Response with a Variable Length Message

```
IPC_RETURN_TYPE IPC_queryNotifyVC
          (const char *msgName,
           IPC_VARCONTENT_PTR varcontent
           HANDLER_TYPE handler,
           void *clientData)
```

Equivalent to `IPC_queryNotify` (Section 5.3), except that it uses a pointer to a structure that includes both the length and content of the message data. Designed to facilitate interfacing with the marshalling/unmarshalling functions. In addition to the return values of `IPC_queryNotify`, it returns `IPC_Error` if `varcontent` is `NULL` (`IPC_Null_Argument`). Java version currently does not support client data.

5.7 Send a Variable Length Query and Block Waiting

```
IPC_RETURN_TYPE IPC_queryResponseVC
          (const char *msgName,
           IPC_VARCONTENT_PTR varcontent
           BYTE_ARRAY *replyHandle,
           unsigned int timeoutMSecs)
```

Equivalent to `IPC_queryResponse` (Section 5.4), except that it uses a pointer to a structure that includes both the length and content of the message data. Designed to facilitate interfacing with the marshalling/unmarshalling functions. In addition to the return values of `IPC_queryResponse`, it returns `IPC_Error` if varcontent is `NULL` (`IPC_Null_Argument`).

6 MARSHALLING DATA

Strictly speaking, these functions are not needed to run the basic IPC. They are included in the IPC API because they provide a powerful interface between the low-level IPC protocols (which deal in byte streams) and higher-level functions (which deal in C, Java, Python, and LISP structures).

It is suggested, for reasons of both safety and ease of use, that these functions be utilized for all messages, as they can correctly deal with byte ordering and packing between machines. In particular, code that uses them does not need to be changed (except for specifying the format string) if the format of the data structure changes. The overhead for using these functions is small, both in time and memory used.

IMPORTANT: In order to deal with inter-machine differences, it is imperative that messages sent using `IPC_marshall` be handled by calling `IPC_unmarshall`. Your code should **NOT** depend in any way on assumptions about the way the marshalling functions transform data structures.

6.1 Compile a Format String

```
FORMATTER_PTR IPC_parseFormat
                (const char *formatString)
```

Returns a pointer to a data structure that encodes the format represented textually by `formatString`, where `formatString` adheres to the syntax described in Section 3). Returns `NULL` if the `formatString` argument is `NULL`. Sets `IPC_errno` to `IPC_Illegal_Formatter` if the syntax of `formatString` is illegal (and exits if the verbosity is `IPC_Exit_On_Errors`, see Section 4.10), and sets it to `IPC_Not_Initialized` if the IPC has not been initialized (see Section 4.13).

6.2 Define a New Format

```
IPC_RETURN_TYPE IPC_defineFormat
                (const char *formatName,
                 const char *formatString)
```

Enable users to associate names with format strings, and use the names in other format strings. For example, one could write:

```
IPC_defineFormat
    ("point","{double, double, double}");
IPC_defineFormat
    ("point-array", "[point:5]");
IPC_defineFormat
    ("two-point-arrays",
     "{point-array, point-array}");
```

Without named formatters, the latter would have to be written:

```
"{[{double, double, double}:5],
  [{double, double, double}:5]}"
```

The use of named formatters reduces the chances of mistyping format strings, promotes modularity (if a type definition changes, the format string need be changed in only one place), and promotes understandability, by enabling one to define names for semantic types (e.g.,

`IPC_defineFormat("radians", "double"))`.

It is suggested that you actually use `#define`'s and `defconstant`'s, rather than explicit strings in the calls to `IPC_defineFormat`:

```
#define RADIANS_NAME "radians"
#define RADIANS_FORMAT "double"
IPC_defineFormat(RADIANS_NAME,
                 RADIANS_FORMAT);
```

One thing to note: You must be connected to the IPC server before calling `IPC_defineFormat`. You do not have to define a named format before you use it in another `IPC_defineFormat` or an `IPC_defineMsg` call, but it must be defined before it is used (either explicitly or implicitly) in a marshalling or unmarshalling call.

Defined formats propagate among modules, so only one module need define each format (although it is not an error for multiple modules to define a format - if the definitions are inconsistent, the last definition will take precedence).

`IPC_Error` is returned if the task/process is not currently connected to the IPC network (with `IPC_errno` being set to `IPC_Not_Connected`) or if `formatName` is `NULL` (`IPC_Null_Argument`); otherwise `IPC_OK` is returned.

6.3 Is Format Consistent

```
IPC_RETURN_TYPE IPC_checkMsgFormats
                (const char *msgName,
                 const char *formatString)
```

Check whether `formatString` is the same as the format associated with the message `msgName`. Checks for semantic equality, not just syntactic equality (that is, the format strings don't have to be exactly the same - the question is whether they parse to the same formatter).

Returns `IPC_OK` if the `formatString` is the same as the format associated with `msgName`. `IPC_Error` is returned if the formats differ (`IPC_errno` is set to `IPC_Mismatched_Formatter`). The function also returns `IPC_Error` if the task/process is not currently connected to the IPC network (`IPC_Not_Connected`), if `msgName` is `NULL` (`IPC_Null_Argument`), or if the message has not been defined (`IPC_Message_Not_Defined`).

6.4 Format Associated with a Message Name

```
FORMATTER_PTR IPC_msgFormatter
                (const char *msgName)
```

Return a pointer to a "formatter" that encodes the format string associated with `msgName` in the `IPC_defineMsg` call. Returns `NULL` if the message has not been defined, if the format string associated with the message is `NULL`, or if the format string does not adhere to the format string syntax (Section 3). The way to differentiate these situations is that in the latter case, `IPC_errno` will be set to `IPC_Illegal_Formatter`). Also returns `NULL` and sets `IPC_errno` to `IPC_Not_Initialized` if IPC has not been initialized (see Section 4.13). In addition, will exit if the verbosity is `IPC_Exit_On_Errors` (see Section 4.10).

The message formatter is cached so that, except for the first call, it is very efficient to retrieve.

6.5 Format Associated with a Message Instance

```
FORMATTER_PTR IPC_msgInstanceFormatter
                (MSG_INSTANCE msgInstance)
```

Equivalent to (but more efficient than) `IPC_msgFormatter(IPC_msgInstanceName(msgInstance))`. Included in the IPC API because it is useful in unmarshalling data within a message handler.

6.6 Converting Data Structures to Byte Arrays

```
IPC_RETURN_TYPE IPC_marshall
        (FORMATTER_PTR formatter,
         void *dataptr,
         IPC_VARCONTENT_PTR varcontent)
```

"Marshalling" a data structure means converting it to a format (byte array) that is suitable for transmission by the IPC. Based on the `formatter` data structure, `IPC_marshall` sets `varcontent->content` to the marshalled byte array that represents the data structure pointed to by `dataptr`, and sets `varcontent->length` to the length of that array. The result can then be used to publish the message.

For example (blithely ignoring errors):

```
{ IPC_VARCONTENT_TYPE varcontent;

  IPC_marshall(IPC_msgFormatter(msgName),
               &datastruct, &varcontent);
  IPC_publishVC(msgName, &varcontent);
  IPC_freeByteArray(varcontent.content);
}
```

The implementation of `IPC_marshall` and `IPC_unmarshall` uses the data formatter facilities (refer to Section 3), which can transform a large variety of structures, in all the supported languages (C, Java, Python, LISP), including structures with pointers (strings, variable length arrays, matrices, linked lists, etc.), taking into account differences in byte ordering and packing between machine types. For example, the format string for the data structure:

```
struct _matrixList {
   float matrix[2][2];
```

```
    char *matrixName;
    int count;
    struct _matrixList *next;
 }
```

is: "{[float:2,2], string, int, *!}".

This function returns IPC_Error if IPC has not been initialized (IPC_Not_Initialized), if the formatter is invalid (IPC_Illegal_Formatter) or if var-content is NULL (IPC_Null_Argument). Otherwise returns IPC_OK.

Note that, in general, there is no way to determine whether the byteArray actually matches the format of the formatter. There may be some specific error conditions that can be detected, and if so, IPC_Error will be returned.

6.7 Converting Byte Arrays to Data Structures

```
IPC_RETURN_TYPE IPC_unmarshall
               (FORMATTER_PTR formatter,
                BYTE_ARRAY byteArray,
                void **dataHandle)
```

Allocates and fills in a data structure based on the formatter (Section 3) and the byteArray. Sets the dataHandle to the newly created structure (note that the third argument is not simply a pointer, it is a *handle* – a pointer to a pointer). For example, a handler may be written:

```
void fooMsgHandler
        (MSG_INSTANCE msgInstance,
         BYTE_ARRAY callData,
         void *clientData)
{ FOO_MSG_PTR fooDataPtr;
  FORMATTER_PTR formatter;

  formatter = IPC_msgInstanceFormatter
                      (msgInstance);
  IPC_unmarshall(formatter, callData,
                 (void **)&fooDataPtr);
  IPC_freeByteArray(callData);
  fooFn(fooDataPtr->foo,
        fooDataPtr->bar);
  IPC_freeData(formatter,
               (void *)fooDataPtr);
}
```

The intent is that the result of unmarshalling a byte array produced by the IPC_marshall function should return an identical data structure, up to pointer equality.

For Java and Python, the dataHandle argument is not used; instead, the Java function (called unmarshallMsgData) returns the newly created structure and the Python function returns a tuple of the structure and the IPC_RETURN_TYPE. In addition, the Java interface enables one to specify the class type of the data that will be created as the third argument to the function. For Python, use the function IPC_msgClass to associate a class type with a message (if the class is not specified, a structure of type IPCdata will be created).

This function returns IPC_Error if IPC has not been initialized (IPC_Not_Initialized), or if the formatter is invalid (IPC_Illegal_Formatter). Otherwise, the function returns IPC_OK.

6.8 Unmarshalling a Pre-Allocated Data Structure

```
IPC_RETURN_TYPE IPC_unmarshallData
               (FORMATTER_PTR formatter,
                BYTE_ARRAY byteArray,
                void *dataptr,
                int dataSize)
```

This function is similar to IPC_unmarshall, except that it does not allocate new space for the unmarshalled data, but instead fills in the dataptr pointer. The function assumes that dataptr points to an already allocated data structure (either on the stack or the heap), that is described by the formatter and is dataSize bytes long. In general, it is a bit more efficient than IPC_unmarshall, in that it does less memory allocation and byte copying.

For example, one could write:

```
static void fooMsgHandler
              (MSG_INSTANCE msgInstance,
               BYTE_ARRAY callData,
               void *clientData)
{ FOO_MSG_TYPE fooData;
  FORMATTER_PTR formatter;

  formatter = IPC_msgInstanceFormatter
                        (msgInstance);
```

```
    IPC_unmarshallData(formatter,
                       callData, &fooData,
                       sizeof(fooData));
    IPC_printData(formatter, stdout,
                  &fooData);
    IPC_freeDataElements(formatter,
                         &fooData)
    IPC_freeByteArray(callData);
  }
```

Note the call to `IPC_freeDataElements` in the above example. While IPC does not allocate new space for the top-level data structure (pointed to by `dataptr`), it *will* allocate space for substructures that may occur (for instance, if the structure contains pointers or strings). To perform proper memory management, you should be sure to free such allocated memory using `IPC_freeDataElements` (Section 6.11). While you do not need to do this if you are sure the structure is fixed-size, it does not hurt to always call `IPC_freeDataElements`.

Not implemented in Java. For Python, the `dataSize` argument is not used and the function takes an optional class type argument (in case the `dataPtr` argument is `None`, in which case a new structure is created). The Python function returns a tuple of the new structure and the `IPC_RETURN_TYPE`.

The function returns `IPC_Error` if IPC has not been initialized (`IPC_Not_Initialized`), if the `formatter` is invalid (`IPC_Illegal_Formatter`), if `dataPtr` is `NULL` but the formatter is not (`IPC_Null_Argument`), or if `dataSize` does not match the size as dictated by the formatter (`IPC_Wrong_Buffer_Length`). Note that, in general, there is no way to determine whether the `byteArray` actually matches the format of the `formatter`. There may be some specific error conditions that can be detected, and if so, `IPC_Error` will be returned.

6.9 Free up a Byte Array

```
void IPC_freeByteArray
            (BYTE_ARRAY byteArray)
```

The basic IPC protocols pass around C byte arrays. This function is used to perform memory management by freeing up those byte arrays. This is done automatically in the `IPC_xxxData` functions. This function should be used in C programs instead of `free`, since that enables the memory to be reclaimed by IPC and reused.

6.10 Free the Data Pointer

```
IPC_RETURN_TYPE IPC_freeData
              (FORMATTER_PTR formatter,
               void *dataptr)
```

Frees the `dataptr`, and any substructures it may have, according to the given format. For example, if `dataptr` were of type "`struct _matrixList *`" (see Section 6.6), `IPC_freeData` would free the top-level structure pointed to by `dataptr`, the `matrixName` string, and would recursively free each element of the list.

This function is not available in Java, Python, or LISP (it is not needed).

Returns `IPC_Error` if IPC is not initialized (`IPC_Not_Initialized`), if the `formatter` is invalid (`IPC_Illegal_Formatter`), or if `dataptr` is `NULL` but `formatter` is not (`IPC_Null_Argument`). Otherwise returns `IPC_OK`.

6.11 Free the Elements of the Structure

```
IPC_RETURN_TYPE IPC_freeDataElements
              (FORMATTER_PTR formatter,
               void *dataptr)
```

Frees any substructure the `dataptr` may have, according to the given format. For example, if the `dataptr` were of type "`struct _matrixList *`" (see Section 6.6), `IPC_freeDataElements` would free the `matrixName` string, and would recursively free each element of the list, but would *not* free the top-level structure pointed to by `dataptr`. This function may be useful in conjunction with `IPC_unmarshallData` (Section 6.8).

This function is not available in Java, Python, or LISP (it is not needed).

Returns `IPC_Error` if IPC is not initialized (`IPC_Not_Initialized`), if the `formatter` is invalid (`IPC_Illegal_Formatter`), or if `dataptr` is `NULL` but formatter is not (`IPC_Null_Argument`). Otherwise returns `IPC_OK`.

6.12 Marshall a Structure and Publish a Message

```
IPC_RETURN_TYPE IPC_publishData
                    (const char *msgName,
                     void *dataptr)
```

Use the formatter associated with the `msgName` to marshall the structure pointed to by `dataptr` into a byte array, and publish the message. Combines the marshalling and publish functionality. Roughly equivalent to:

```
{IPC_VARCONTENT_TYPE varcontent;

 IPC_marshall(IPC_msgFormatter(msgName),
              dataptr, &varcontent);
 IPC_publishVC(msgName, &varcontent);
 IPC_freeByteArray(varcontent->content);
}
```

One can subscribe to messages using either `IPC_subscribeData` or `IPC_subscribe`. With `IPC_subscribeData` the data is automatically unmarshalled, so you never need to unmarshall or deal with byte arrays. With `IPC_subscribe`, you should put the following at the beginning of your handler functions:

```
IPC_unmarshall
  (IPC_msgInstanceFormatter(msgInstance)
   byteArray, (void **)&dataPtr);
IPC_freeByteArray(byteArray);
```

(in the LISP version, there is another alternative – using the macro `IPC_defun_handler` (Section 6.18).

Returns `IPC_Error` under all the situations that `IPC_marshall` (Section 6.6) and `IPC_publish` (Section 4.22) would return `IPC_Error`.

6.13 Combine Marshalling and Response

```
IPC_RETURN_TYPE IPC_respondData
                  (MSG_INSTANCE msgInstance,
                   const char *msgName,
                   void *dataptr)
```

Use the formatter associated with the `msgName` to marshall the structure pointed to by `dataptr` into a byte array, and respond to the message instance. Combines the marshalling and query/response functionality. Roughly equivalent to:

```
{ IPC_VARCONTENT_TYPE varcontent;

  IPC_marshall(IPC_msgFormatter(msgName),
               dataptr, &varcontent);
  IPC_respond(msgInstance, msgName,
              varcontent.length,
              varcontent.content);
  IPC_freeByteArray(varcontent.content);
}
```

Returns `IPC_Error` under all the situations that `IPC_marshall` (Section 6.6) and `IPC_respond` (Section 5.1) would return `IPC_Error`.

6.14 Combine Marshall and Query

```
IPC_RETURN_TYPE IPC_queryNotifyData
                     (const char *msgName,
                      void *dataptr,
                      HANDLER_TYPE handler,
                      void *clientData)
```

Use the formatter associated with the `msgName` to marshall the structure pointed to by `dataptr` into a byte array, and send a query. Combines the marshalling and query/response functionality. Roughly equivalent to:

```
{ IPC_VARCONTENT_TYPE varcontent;
  IPC_marshall(IPC_msgFormatter(msgName)
               dataptr, &varcontent);
  IPC_queryNotifyVC(msgName, varcontent,
                    handler, clientData);
  IPC_freeByteArray(varcontent.content);
}
```

Java version currently does not support client data. The Java version takes, instead, a fourth argument that is the class type of the data to be passed to the handler. The Python version has an optional fifth argument that likewise indicates the class type.

Returns `IPC_Error` under all the situations that `IPC_marshall` (Section 6.6) and `IPC_queryNotify` (Section 5.3) would return `IPC_Error`.

6.15 Combine Marshall, Query, and Response

```
IPC_RETURN_TYPE IPC_queryResponseData
                (const char *msgName,
                 void *dataptr,
                 void **replyData,
                 unsigned int timeoutMSecs)
```

Use the formatter associated with the `msgName` to marshall the structure pointed to by `dataptr` into a byte array, send a query, and wait for the response. Unmarshall the response into a data structure, and sets `replyData` to that value. Combines the marshalling and query/response functionality. Roughly equivalent to:

```
{ IPC_VARCONTENT_TYPE varcontent;
  FORMATTER_PTR formatter;
  BYTE_ARRAY reply;

  formatter = IPC_msgFormatter(msgName);
  IPC_marshall(formatter, dataptr,
               &varcontent);
  IPC_queryResponseVC(msgName,
                      varcontent,
                      &reply,
                      timeoutMSecs);
  IPC_unmarshall(formatter, reply,
                 replyData);
  IPC_freeByteArray(varcontent.content);
  IPC_freeByteArray(reply);
}
```

Neither the Java nor Python versions use the third argument (`replyData`); instead, both return the data (with the Python version returning a tuple of the reply data and the `IPC_RETURN_TYPE` value). Instead, the Java version has a required third argument that is the class type of the data to be returned; the Python version has an optional fourth argument (after `timeoutMSecs`) that is the class type.

Returns `IPC_Error` under all the situations that `IPC_marshall` (Section 6.6) and `IPC_queryResponse` (Section 5.4) would return `IPC_Error`.

6.16 Write a Textual Representation of The Data

```
IPC_RETURN_TYPE IPC_printData
                (FORMATTER_PTR formatter,
                 FILE *stream,
                 void *dataptr)
```

Write on the given `stream` a (human-readable) textual representation of the `dataptr`, including any substructures it may have, according to the given `formatter`. The `stream` can be an open file or the terminal (`stdout` or `stderr`).

This function is included mainly for debugging purposes.

Returns `IPC_Error` if IPC is not initialized (`IPC_Not_Initialized`), if the `stream` is not open for writing, the `formatter` is invalid (`IPC_Illegal_Formatter`), or if `dataptr` is `NULL` but `formatter` is not (`IPC_Null_Argument`). Otherwise returns `IPC_OK`.

6.17 Force Data Structure to be an Array

```
(IPC_defstruct (name) &rest slots)
```

[LISP ONLY]

Has the same syntax as the LISP `defstruct` construct, but forces the structure to be an array, so that the marshalling/unmarshalling functions can access and set slots of the structure, without having to know the names of its accessory functions. For example:

```
(IPC:IPC_defstruct (sample)
  (i1 0 :type integer)
  (str1 "" :type string)
  (d1 0.0 :type float))
```

6.18 Automatic Data Unmarshalling

```
(IPC_defun_handler name
    (msg-instance lisp-data client-data)
  &rest body)
```

[LISP ONLY]

Has the same syntax as the LISP `defun` construct, but produces an IPC handler function that automatically unmarshalls the data and creates a LISP data structure for use by the handler. The following are roughly equivalent:

```
(IPC:IPC_defun_handler barHnd
    (msg-ref msg-data client-data)
  (declare (ignore client-data))
  (format T "~a~%" msg-data)
  (IPC_publishData "fooMsg" msg-data))


(defun barHnd
    (msg-ref byte-array client-data)
  (declare (ignore client-data))
  (let (msg-data)
    (IPC_unmarshall
      (IPC_msgInstanceFormatter msg-ref)
      byte-array msg-data)
    (format T "~a~%" msg-data)
    (IPC_publishData "fooMsg" msg-data)
    (IPC_freeByteArray byte-array)))
```

This facility is now largely superceded by use of `IPC_subscribeData` (Section 4.27).

6.19 Associating a Class with Message

```
IPC_msgClass(msg-name, class)
```

[PYTHON ONLY]

If you associate a Python class with a message name, then whenever a message of that name gets unmarshalled, IPC will use that class to fill in the slots. That is, the resulting data structure will be an instance of that class.

7 CONTEXTS

There are occasions when a module needs to connect to more than one central server. For instance, if you have two robots with a relatively slow radio link connecting them, it may be desirable for reasons of bandwidth and latency to have a central server residing on each robot. However, if one robot wants to send a message to the other robot, it needs to (temporarily) access the other robot's IPC subnetwork.

The following functions can be used to achieve this. A module/task can call `IPC_connectModule` multiple times, giving different `serverName`'s each time (Section 4.14). Each call to `IPC_connectModule` (or `IPC_connect`) sets up a different *context*, which is essentially a connection to a particular central server, along with all the messages defined by the modules connected to that server.

To use the context mechanism, it is advisable to store all the contexts in global variables. That is, call `IPC_getContext` immediately after a call to `IPC_connectModule`, and store the return value. Then, one can call `IPC_setContext` with the stored context value before sending a message to a module on that context's subnetwork.

For instance, to implement a *bridge* program (one that passes messages from one subnetwork to another), one could use this fragment of code:

```
static IPC_CONTEXT_PTR central1;
static IPC_CONTEXT_PTR central2
int main (void)
{
  ...
  IPC_connectModule("foo", HOST1);
  central1 = IPC_getContext();
  IPC_subscribe(MSG1, msg1Handler, NULL)
  IPC_connectModule("foo", HOST2);
  central2 = IPC_getContext();
  IPC_defineMsg(MSG1,
                IPC_VARIABLE_LENGTH,
                MSG1_FMT);
  ...
}

void msg1Handler (...)
{
  ...
  if (IPC_getContext() != central1)
    printf("Something screwy going on");
  IPC_unmarshall(..., &data);
  IPC_setContext(central2);
  IPC_publishData(MSG1, data);
  IPC_freeData(..., data);
  ...
}
```

Note that this example illustrates that when a message handler is invoked, IPC automatically sets the current context to be that of the subnetwork that sent the message.

7.1 Get the Current Context

```
IPC_CONTEXT_PTR IPC_getContext (void)
```

Get the current IPC context, where *context* is a connection to a given central server. Returns `NULL` if there is no current IPC connection (i.e., either `IPC_connectModule` or `IPC_connect` have not been called, or `IPC_disconnect` has been called).

7.2 Set the Current Context

```
IPC_RETURN_TYPE IPC_setContext
                (IPC_CONTEXT_PTR context)
```

Set the current IPC context to be `context`, where *context* is a connection to a given central server. `context` should be the return value of a previous `IPC_getContext` call.

Returns `IPC_Error` (`IPC_Null_Argument`) if `context` is `NULL`. Otherwise, returns `IPC_OK`.

8 TIMERS

There are occasions when a module needs to perform some action at a particular time. IPC provides several functions that enable user-specified functions to be invoked at a given point in time, or periodically over a given interval.

While these functions can be used for time-dependent operations, note that they are not truly interrupt driven - they will be invoked only when the module is within some IPC function that is listening for messages (`IPC_dispatch`, `IPC_listen`, `IPC_listenClear`, `IPC_queryResponse`, `IPC_queryResponseVC`, `IPC_queryResponseData`). If the specified time passes while the module is doing some other computation (or is swapped out), the timer function will be invoked at the next available opportunity. Thus, you should not rely on these functions to provide guaranteed real-time response. This also implies that the timer functions are in effect only while the task/process is connected to the IPC network (i.e., all timer functions are "disabled" before calling `IPC_connect` and after calling `IPC_disconnect`).

These functions are not currently available for LISP (please contact us if you need this functionality).

8.1 Timer Callback Type

```
typedef void (*TIMER_HANDLER_TYPE)
             (void *clientData,
              unsigned long currentTime,
              unsigned long scheduledTime);
```

The type of timer callback handlers. `clientData` is a pointer to any user-defined data, and is associated with the timer in the "add" call (Sections 8.2, 8.3, and 8.4). Note that Java version currently does not support client data. `currentTime` is the time at which the handler function is invoked; `scheduledTime` is the time when it was supposed to be invoked (as indicated by the "add" call). `scheduledTime` may be later than `currentTime` because timers are invoked only from within IPC functions that are listening for messages (see above).

8.2 Add a Timer

```
IPC_RETURN_TYPE IPC_addTimer
                (unsigned long tdelay,
                 long count,
                 TIMER_HANDLER_TYPE handler,
                 void *clientData)
```

Add a timer, which will periodically invoke the `handler` function while IPC is running. `clientData` is passed to the handler routine when it is invoked (Section 8.1). Java version currently does not support client data.

`tdelay` is the number of milliseconds to wait for, or between, the timer events. The first invocation of the `handler` is `tdelay` milliseconds after the timer is "added". Each additional invocation occurs `tdelay` milliseconds after the previous invocation was begun.

`count` is the number of invocations before the timer is automatically removed. If `count` is `TRIGGER_FOREVER`, then the timer continues indefinitely, or until explicitly removed by the user (Section 8.5).

If a timer already exists that invokes `handler`, then the new definition replaces the old one (even if the old definition had been running for a while).

Returns `IPC_OK` if the timer was successfully added. Returns `IPC_Error` (and sets `IPC_errno` appropriately) if the `handler` is `NULL` (`IPC_Null_Argument`), if `tdelay` is zero (`IPC_Argument_Out_Of_Range`), or if `count` is negative (`IPC_Argument_Out_Of_Range`).

8.3 Add Timer Invoked Once

```
IPC_RETURN_TYPE IPC_addOneShotTimer
                (unsigned long tdelay,
                 TIMER_HANDLER_TYPE handler,
                 void *clientData)
```

Shorthand for setting up a timer that triggers just once. Equivalent to:

```
IPC_addTimer(tdelay, 1,
             handler, clientData)
```

Java version currently does not support client data.

8.4 Add Timer Invoked Periodically

```
IPC_RETURN_TYPE IPC_addPeriodicTimer
              (unsigned long tdelay,
               TIMER_HANDLER_TYPE handler,
               void *clientData)
```

Shorthand for setting up a timer that triggers forever. Equivalent to:

```
IPC_addTimer(tdelay, TRIGGER_FOREVER,
             handler, clientData)
```

Java version currently does not support client data.

8.5 Remove a Timer

```
IPC_RETURN_TYPE IPC_removeTimer
              (TIMER_HANDLER_TYPE handler)
```

Remove a timer whose handler function matches `handler` (there will be at most one such timer existing at any given time).

Returns `IPC_Error` if the `handler` is `NULL` (setting `IPC_errno` to `IPC_Null_Argument`). Otherwise, returns `IPC_OK` (if a timer with the given handler does not currently exist, a warning is issued, but the function still returns `IPC_OK`).

8.6 Add a Timer by Reference

```
IPC_RETURN_TYPE IPC_addTimerGetRef
              (unsigned long tdelay,
               long count,
               TIMER_HANDLER_TYPE handler,
               void *clientData,
               TIMER_REF *timerRef)
```

Same functionality as `IPC_addTimer` (Section 8.2), except that this function never replaces any current timer, even if it has the same `handler` and `clientData`. Instead, the `timerRef` pointer is set to a reference to the instance of the timer (so that it can be used to remove the timer, see Section 8.7). Note that `TIMER_REF` is an internal IPC data type, and its elements cannot be accessed by user application code. Java version currently does not support client data.

If `timerRef` is `NULL`, then this function works exactly the same as `IPC_addTimer` (Section 8.2).

Returns the same values as `IPC_addTimer`, under the same error conditions.

8.7 Remove a Timer by Reference

```
IPC_RETURN_TYPE IPC_removeTimerByRef
                                (TIMER_REF timerRef)
```

Remove a timer whose reference matches `timerRef`. `timerRef` is a reference to a timer gotten from a call to `IPC_addTimerByRef` (Section 8.6).

Returns the same values as `IPC_removeTimer` (Section 8.5), under the same error conditions.

Appendix A Example Programs

File “**module.h**” is a header file that defines various data structures and format strings for message passing between modules.

```
*****************************************************************
typedef enum { WaitVal, SendVal, ReceiveVal, ListenVal } STATUS_ENUM;
typedef struct { int i1;
                STATUS_ENUM status;
                double matrix[2][3];
                double d1;
} T1_TYPE, *T1_PTR;
#define T1_NAME "T1"
/* First form of "enum". 3 is the maximum value-i.e., the value of WaitVal */
#define T1_FORMAT "{int, {enum : 3}, [double:2,3], double}"

typedef struct { char *str1;
                int count;
                T1_TYPE *t1; /* Variable length array of type T1_TYPE */
                STATUS_ENUM status;
} T2_TYPE, *T2_PTR;

#define T2_NAME "T2"
/* Alternate form of "enum". */
#define T2_FORMAT
      "{string, int, <T1:2>, {enum WaitVal, SendVal, ReceiveVal, ListenVal}}"

typedef int MSG1_TYPE, *MSG1_PTR;
#define MSG1 "message1"
#define MSG1_FORMAT "int"

typedef char *MSG2_TYPE, **MSG2_PTR;
#define MSG2 "message2"
#define MSG2_FORMAT "string"

typedef T1_TYPE QUERY1_TYPE, *QUERY1_PTR;
#define QUERY1 "query1"
#define QUERY1_FORMAT T1_NAME

typedef T2_TYPE RESPONSE1_TYPE, *RESPONSE1_PTR;
#define RESPONSE1 "response1"
#define RESPONSE1_FORMAT T2_NAME

#define MODULE1_NAME "module1"
#define MODULE2_NAME "module2"
#define MODULE3_NAME "module3"
================================================================
```

File “**module1.c**” defines a single message handler to print out its data, and a single terminal interface to send out messages and quit the program. It is a test program for IPC that publishes MSG1 and QUERY1, and subscribes to MSG2. It sends MSG1 whenever an “m” is typed at the terminal; sends a QUERY1 whenever an “r” is typed, and quits the program when a “q” is typed. It should be run in conjunction with module2.

```
******************************************************************************
#include <stdio.h>
#include <math.h>
#ifndef M_PI
#define M_PI 3.14159
#endif

#include "ipc/ipc.h"
#include "module.h"

static void msg2Handler (MSG_INSTANCE msgRef, BYTE_ARRAY callData,
                         void *clientData)
{
  MSG2_TYPE str1;
  IPC_unmarshallData(IPC_msgInstanceFormatter(msgRef), callData,
                     &str1, sizeof(str1));
  printf("msg2Handler: Receiving %s (%s) [%s]\n",
  IPC_msgInstanceName(msgRef), str1, (char *)clientData);
  IPC_freeByteArray(callData);
}

#ifndef VXWORKS
static void stdinHnd (int fd, void *clientData)
{
  char inputLine[81];
  fgets(inputLine, 80, stdin);
  switch (inputLine[0]) {
    case 'q': case 'Q':
    IPC_disconnect();
    exit(0);
  case 'm': case 'M':
    { MSG1_TYPE i1 = 42;
      printf("\n IPC_publishData(%s, &i1) [%d]\n", MSG1, i1);
      IPC_publishData(MSG1, &i1);
      break;
    }
  case 'r': case 'R':
    { QUERY1_TYPE t1 = {666, SendVal,
                        {{0.0, 1.0, 2.0}, {1.0, 2.0, 3.0}}, M_PI};
      RESPONSE1_PTR r1Ptr;
      printf("\n IPC_queryResponseData(%s, &t1, &r1Ptr, IPC_WAIT_FOREVER)\n",
             QUERY1);
      IPC_queryResponseData(QUERY1, &t1, (void **)&r1Ptr, IPC_WAIT_FOREVER);
      printf("\n Received response:\n");
      IPC_printData(IPC_msgFormatter(RESPONSE1), stdout, r1Ptr);
      IPC_freeData(IPC_msgFormatter(RESPONSE1), r1Ptr);
      break;
    }
  default:
    printf("stdinHnd [%s]: Received %s", (char *)clientData, inputLine);
    fflush(stdout);
  }
}
#endif
```

```
#if defined(VXWORKS)
#include <sys/times.h>

void module1(void)
#else
void main (void)
#endif
{
  /* Connect to the central server */
  printf("\nIPC_connect(%s)\n", MODULE1_NAME);
  IPC_connect(MODULE1_NAME);

  /* Define the named formats that the modules need */
  printf("\nIPC_defineFormat(%s, %s)\n", T1_NAME, T1_FORMAT);
  IPC_defineFormat(T1_NAME, T1_FORMAT);
  printf("\nIPC_defineFormat(%s, %s)\n", T2_NAME, T2_FORMAT);
  IPC_defineFormat(T2_NAME, T2_FORMAT);

  /* Define the messages that this module publishes */
  printf("\nIPC_defineMsg(%s, IPC_VARIABLE_LENGTH, %s)\n", MSG1, MSG1_FORMAT);
  IPC_defineMsg(MSG1, IPC_VARIABLE_LENGTH, MSG1_FORMAT);
  printf("\nIPC_defineMsg(%s, IPC_VARIABLE_LENGTH, %s)\n",
         QUERY1, QUERY1_FORMAT);
  IPC_defineMsg(QUERY1, IPC_VARIABLE_LENGTH, QUERY1_FORMAT);

  /* Subscribe to the messages that this module listens to.
   * NOTE: No need to subscribe to the RESPONSE1 message, since it is a
   * response to a query, not a regular subscription! */
  printf("\nIPC_subscribe(%s, msg2Handler, %s)\n", MSG2, MODULE1_NAME);
  IPC_subscribe(MSG2, msg2Handler, MODULE1_NAME);

  #ifndef VXWORKS /* Since vxworks does not handle stdin from the terminal,
                     this does not make sense. Instead, send off messages
                     periodically */
  /* Subscribe a handler for tty input.
     Typing "q" will quit the program; Typing "m" will send MSG1;
     Typing "r" will send QUERY1 ("r" for response) */
  printf("\nIPC_subscribeFD(%d, stdinHnd, %s)\n", fileno(stdin),
         MODULE1_NAME);
  IPC_subscribeFD(fileno(stdin), stdinHnd, MODULE1_NAME);
  printf("\nType 'm' to send %s; Type 'r' to send %s; Type 'q' to quit\n",
         MSG1, QUERY1);
  IPC_dispatch();
#else
#define NUM_MSGS (10)
#define INTERVAL (5)
  {
    int i;
    printf("\nWill send a message every %d seconds for %d seconds\n",
           INTERVAL, NUM_MSGS);
    for (i=1; i<NUM_MSGS; i++) {
      /* Alternate */
      if (i & 1) {
         MSG1_TYPE i1 = 42;
        printf("\n IPC_publishData(%s, &i1) [%d]\n", MSG1, i1);
        IPC_publishData(MSG1, &i1);
       } else {
        QUERY1_TYPE t1 = {666, SendVal,
                          {{0.0, 1.0, 2.0}, {1.0, 2.0, 3.0}}, M_PI};
        RESPONSE1_PTR r1Ptr;
        printf("\n IPC_queryResponseData(%s, &t1, &r1Ptr, IPC_WAIT_FOREVER)\n",
               QUERY1);
        IPC_queryResponseData(QUERY1, &t1, (void **)&r1Ptr, IPC_WAIT_FOREVER);
        printf("\n Received response:\n");
        IPC_printData(IPC_msgFormatter(RESPONSE1), stdout, r1Ptr);
        IPC_freeData(IPC_msgFormatter(RESPONSE1), r1Ptr);
      }
      /* This works instead of sleep */
      { struct timeval sleep = {INTERVAL, 0};
        select(FD_SETSIZE, NULL, NULL, NULL, &sleep);
      }
     }
  }
#endif
  IPC_disconnect();
}
================================================================
```

File "**module2.c**" provides examples of both a publish/subscribe message handler and a query/response message handler. It receives the messages sent by module1 and responds when appropriate.

This test program for IPC publishes MSG2 and subscribes to MSG1 and QUERY1. It listens for MSG1 and prints out message data. When QUERY1 is received, it publishes MSG2 and responds to the query with RESPONSE1. It exits when "q" is typed at the terminal, and should be run in conjunction with module1.

```
*****************************************************************************
#include <stdio.h>

#include "ipc/ipc.h"
#include "module.h"

static void msg1Handler (MSG_INSTANCE msgRef, BYTE_ARRAY callData,
                         void *clientData)
{
  MSG1_TYPE i1;

  IPC_unmarshallData(IPC_msgInstanceFormatter(msgRef), callData,
                     &i1, sizeof(i1));

  printf("msg1Handler: Receiving %s (%d) [%s]\n",
         IPC_msgInstanceName(msgRef), i1, (char *)clientData);

  IPC_freeByteArray(callData);
}

static void queryHandler (MSG_INSTANCE msgRef,
                          BYTE_ARRAY callData, void *clientData)
{
  QUERY1_TYPE t1;
  MSG2_TYPE str1 = "Hello, world";
  RESPONSE1_TYPE t2;

  printf("queryHandler: Receiving %s [%s]\n",
         IPC_msgInstanceName(msgRef), (char *)clientData);

  /* NOTE: Have to pass a pointer to t1Ptr! */
  IPC_unmarshallData(IPC_msgInstanceFormatter(msgRef), callData,
                     &t1, sizeof(t1));
  IPC_printData(IPC_msgInstanceFormatter(msgRef), stdout, &t1);

  /* Publish this message -- all subscribers get it */
  /* NOTE: You need to pass a *pointer* to the string,
     not just the string itself! */
  printf("\n IPC_publishData(%s, &str1) [%s]\n", MSG2, str1);
         IPC_publishData(MSG2, &str1);

  t2.str1 = str1;
  /* Variable length array of one element */
  t2.t1 = &t1;
  t2.count = 1;
  t2.status = ReceiveVal;

  /* Respond with this message -- only the query handler gets it */
  printf("\n IPC_respondData(%#X, %s, &t2)\n", (int)msgRef, RESPONSE1);
  IPC_respondData(msgRef, RESPONSE1, &t2);

  IPC_freeByteArray(callData);
}

static void stdinHnd (int fd, void *clientData)
{
  char inputLine[81];

  fgets(inputLine, 80, stdin);

  switch (inputLine[0]) {
    case 'q': case 'Q':
      IPC_disconnect();
      exit(0);
    default:
      printf("stdinHnd [%s]: Received %s", (char *)clientData, inputLine);
      fflush(stdout);
  }
}

#if defined(VXWORKS)
void module2(void)
#else
void main (void)
#endif
{
  /* Connect to the central server */
  printf("\nIPC_connect(%s)\n", MODULE2_NAME);
  IPC_connect(MODULE2_NAME);

  /* Define the messages that this module publishes */
  printf("\nIPC_defineMsg(%s, IPC_VARIABLE_LENGTH, %s)\n", MSG2, MSG2_FORMAT);
  IPC_defineMsg(MSG2, IPC_VARIABLE_LENGTH, MSG2_FORMAT);

  printf("\nIPC_defineMsg(%s, IPC_VARIABLE_LENGTH, %s)\n",
         RESPONSE1, RESPONSE1_FORMAT);
  IPC_defineMsg(RESPONSE1, IPC_VARIABLE_LENGTH, RESPONSE1_FORMAT);

  /* Subscribe to the messages that this module listens to. */
  printf("\nIPC_subscribe(%s, msg1Handler, %s)\n", MSG1, MODULE2_NAME);
  IPC_subscribe(MSG1, msg1Handler, MODULE2_NAME);

  printf("\nIPC_subscribe(%s, queryHandler, %s)\n", QUERY1, MODULE2_NAME);
  IPC_subscribe(QUERY1, queryHandler, MODULE2_NAME);

#ifndef VXWORKS /* Since vxworks does not handle stdin from the terminal,
                   this does not make sense. */
  /* Subscribe a handler for tty input. Typing "q" will quit the program. */
  printf("\nIPC_subscribeFD(%d, stdinHnd, %s)\n", fileno(stdin),
         MODULE2_NAME);
  IPC_subscribeFD(fileno(stdin), stdinHnd, MODULE2_NAME);

  printf("\nType 'q' to quit\n");
#endif

  IPC_dispatch();
  IPC_disconnect();
}
================================================================
```

File "**module.lisp**" is the LISP equivalent of "**module.h**".

```
**************************************************************************
;;; typedef enum { WaitVal, SendVal, ReceiveVal, ListenVal } STATUS_ENUM;
(defconstant STATUS_ENUM '(:WaitVal :SendVal :ReceiveVal :ListenVal))

(IPC:IPC_defstruct (T1)
 (i1 0 :type integer)
 (status 0 :type (or integer symbol))
 (matrix NIL :type array)
 (d1 0.0 :type double))

(defconstant T1_NAME "T1")
;;; First form of "enum". 3 is the maximum value -- i.e., the value of WaitVal
(defconstant T1_FORMAT "{int, {enum : 3}, [double:2,3], double}")

(IPC:IPC_defstruct (T2)
 (str1 "" :type string)
 (count 0 :type integer)
 (t1 NIL :type array)
 (status :ReceiveVal :type (or integer symbol)))
```

```
(defconstant T2_NAME "T2")
;;; Alternate form of "enum".
(defconstant T2_FORMAT
"{string, int, <T1:2>, {enum WaitVal, SendVal, ReceiveVal, ListenVal}}")
;;; typedef int MSG1_TYPE, *MSG1_PTR
(defconstant MSG1 "message1")
(defconstant MSG1_FORMAT "int")

;;; typedef char *MSG2_TYPE, **MSG2_PTR;
(defconstant MSG2 "message2")
(defconstant MSG2_FORMAT "string")

;;; typedef T1_TYPE QUERY1_TYPE, *QUERY1_PTR;
(defconstant QUERY1 "query1")
(defconstant QUERY1_FORMAT T1_NAME)

;;; typedef T2_TYPE RESPONSE1_TYPE, *RESPONSE1_PTR;
(defconstant RESPONSE1 "response1")
(defconstant RESPONSE1_FORMAT T2_NAME)

(defconstant MODULE1_NAME "module1")
(defconstant MODULE2_NAME "module2")
(defconstant MODULE3_NAME "module3")
=================================================================
```

File "**module1.lisp**" is the LISP equivalent of "**module1.c**".

It publishes MSG1 and QUERY1 and subscribes to MSG2. It sends MSG1 whenever a "m" is typed at the terminal, send a QUERY1 whenever an "r" is typed, and quits the program when a "q" is typed. It should be run in conjunction with module2.

```
**************************************************************************
;;; Load the common file with all the type and name definitions
(load (make-pathname :DIRECTORY (pathname-directory *LOAD-TRUENAME*)
                     :NAME "module.lisp"))
(IPC:IPC_defun_handler msg2Handler (msgRef lispData clientData)
  (format T "msg2Handler: Receiving ~s (~s) [~s]~%"
          (IPC:IPC_msgInstanceName msgRef) lispData clientData))

(defun stdinHnd (fd clientData)
  (declare (ignore fd))
  (let ((inputLine (read-line)))
    (case (aref inputLine 0)
      ((#\q #\Q)
       (IPC:IPC_disconnect)
       #+ALLEGRO (top-level:do-command "reset") #+LISPWORKS (abort)
       )
      ((#\m #\M)
       (format T "~% (IPC_publishData ~s ~d)~%" MSG1 42)
       (IPC:IPC_publishData MSG1 42))
      ((#\r #\R)
       (let ((t1 (make-T1 :i1 666
                          ;; T1 does not support symbolic enums, so have to
                          ;; use the corresponding integer value
                          :status (position :SendVal STATUS_ENUM)
```

```
                        :matrix (make-array '(2 3)
                                            :element-type 'double-float
                                            :initial-contents
                                            '((0.0d0 1.0d0 2.0d0)
                                              (1.0d0 2.0d0 3.0d0)))
                                            :d1 pi))
            r1)
         (format T "~% (IPC_queryResponseData ~s ~a r1 IPC_WAIT_FOREVER)~%"
                 QUERY1 t1)
         (IPC:IPC_queryResponseData QUERY1 t1 r1 IPC:IPC_WAIT_FOREVER)
         (format T "~% Received response ~a~%" r1)
         ;; (IPC:IPC_printData (IPC:IPC_msgFormatter RESPONSE1) T r1Ptr)
         ))
      (T (format T "stdinHnd [~s]: Received ~s" clientData inputLine)))))

(defun module1 ()

  ;; Connect to the central server
  (format T "~%(IPC_connect ~s)~%" MODULE1_NAME)
  (IPC:IPC_connect MODULE1_NAME)

  ;; Define the named formats that the modules need
  (format T "~%(IPC_defineFormat ~s ~s)~%" T1_NAME T1_FORMAT)
  (IPC:IPC_defineFormat T1_NAME T1_FORMAT)
  (format T "~%(IPC_defineFormat ~s ~s)~%" T2_NAME T2_FORMAT)
  (IPC:IPC_defineFormat T2_NAME T2_FORMAT)

  ;; Define the messages that this module publishes
  (format T "~%(IPC_defineMsg ~s IPC_VARIABLE_LENGTH ~s)~%" MSG1 MSG1_FORMAT)
  (IPC:IPC_defineMsg MSG1 IPC:IPC_VARIABLE_LENGTH MSG1_FORMAT)

  (format T "~%(IPC_defineMsg ~s IPC_VARIABLE_LENGTH ~s)~%"
          QUERY1 QUERY1_FORMAT)
  (IPC:IPC_defineMsg QUERY1 IPC:IPC_VARIABLE_LENGTH QUERY1_FORMAT)

  ;; Subscribe to the messages that this module listens to.
  ;; NOTE: No need to subscribe to the RESPONSE1 message since it is a
  ;; response to a query not a regular subscription!
  (format T "~%(IPC_subscribe ~s 'msg2Handler ~s)~%" MSG2 MODULE1_NAME)
  (IPC:IPC_subscribe MSG2 'msg2Handler MODULE1_NAME)

  ;; Subscribe a handler for tty input.
  ;; Typing "q" will quit the program; Typing "m" will send MSG1;
  ;; Typing "r" will send QUERY1 ("r" for response)
  ;; NOTE: 0 is the file descriptor number of stdin (the terminal)
  (format T "~%(IPC_subscribeFD ~d 'stdinHnd ~s)~%" 0 MODULE1_NAME)
  (IPC:IPC_subscribeFD 0 'stdinHnd MODULE1_NAME)
  (format T "~%Type 'm' to send ~s; Type 'r' to send ~s; Type 'q' to quit~%"
          MSG1 QUERY1)

  (IPC:IPC_dispatch)
)
================================================================
```

The file “**module2.lisp**” is the LISP equivalent of **module2.c**.

It is a test program for IPC that publishes MSG2, and subscribes to MSG1 and QUERY. It listens for MSG1 and prints out message data. When QUERY1 is received, it publishes MSG1 and responds to the query with RESPONSE1. It exits when 'q' is typed at terminal. module2 should be run in conjunction with module1.

```
******************************************************************************
;;; Load the common file with all the type and name definitions
(load (make-pathname :DIRECTORY (pathname-directory *LOAD-TRUENAME*)
                     :NAME “module.lisp”))

(IPC:IPC_defun_handler msg1Handler (msgRef msg1Data clientData)
  (format T “msg1Handler: Receiving ~s (~d) [~s]~%”
         (IPC:IPC_msgInstanceName msgRef) msg1Data clientData))

(IPC:IPC_defun_handler queryHandler (msgRef queryData clientData)
  (declare (ignore clientData))
  (let ((str1 “Hello, world”)
         t2)

    (format T “queryHandler: Receiving ~s [~a]~%”
    (IPC:IPC_msgInstanceName msgRef) queryData)

    ;; Publish this message -- all subscribers get it
    (format T “~% (IPC_publishData ~s, ~s)~%” MSG2 str1)
    (IPC:IPC_publishData MSG2 str1)

    (setq t2 (make-T2 :str1 str1
                      ;; Variable length array of one element
                      :t1 (make-array '(1) :initial-contents (list queryData))
                      :count 1
                    ;; T2 supports symbolic enums, so can use keyword directly
                      :status :ReceiveVal))

    ;; Respond with this message -- only the query handler gets it
    (format T “~% (IPC_respondData ~d ~s ~a)~%” msgRef RESPONSE1 t2)
    (IPC:IPC_respondData msgRef RESPONSE1 t2)
))

(defun stdinHnd (fd clientData)
  (declare (ignore fd))
  (let ((inputLine (read-line)))
    (case (aref inputLine 0)
      ((#\q #\Q)
       (IPC:IPC_disconnect)
       #+ALLEGRO (top-level:do-command “reset”) #+LISPWORKS (abort)
      )
      (T (format T “stdinHnd [~s]: Received ~s” clientData inputLine)))))

(defun module2 ()

  ;; Connect to the central server
  (format T “~%(IPC_connect ~s)~%” MODULE2_NAME)
  (IPC:IPC_connect MODULE2_NAME)
  ;; Define the messages that this module publishes
  (format T "~%(IPC_defineMsg ~s IPC_VARIABLE_LENGTH ~s)~%" MSG2 MSG2_FORMAT)
  (IPC:IPC_defineMsg MSG2 IPC:IPC_VARIABLE_LENGTH MSG2_FORMAT)

  (format T "~%(IPC_defineMsg ~s IPC_VARIABLE_LENGTH ~s)~%"
          RESPONSE1 RESPONSE1_FORMAT)
  (IPC:IPC_defineMsg RESPONSE1 IPC:IPC_VARIABLE_LENGTH RESPONSE1_FORMAT)

  ;; Subscribe to the messages that this module listens to.
  (format T "~%(IPC_subscribe ~s 'msg1Handler ~s)~%" MSG1 MODULE2_NAME)
  (IPC:IPC_subscribe MSG1 'msg1Handler MODULE2_NAME)

  (format T "~%(IPC_subscribe ~s 'queryHandler ~s)~%" QUERY1 MODULE2_NAME)
  (IPC:IPC_subscribe QUERY1 'queryHandler MODULE2_NAME)

  ;; Subscribe a handler for tty input. Typing "q" will quit the program
  (format T "~%(IPC_subscribeFD ~d 'stdinHnd ~s)~%" 0 MODULE2_NAME)
  (IPC:IPC_subscribeFD 0 'stdinHnd MODULE2_NAME)

  (format T "~%Type 'q' to quit~%")

  (IPC:IPC_dispatch)
)
```

File “**module.java**” is the Java equivalent of “**module.h**”.

```
*****************************************************************
public class module {
  /* STATUS_ENUM */
  protected static final int WaitVal    = 0;
  protected static final int SendVal    = 1;
  protected static final int ReceiveVal = 2;
  protected static final int ListenVal  = 3;

  protected static class T1 {
      public int i1;
      public int status; /* STATUS_ENUM */
      public double matrix[/*2*/][/*3*/];
      public double d1;
      public String toString () {
      String str = "{" + i1 +", "+ Integer.toString(status) +", [";
      for (int i=0; i<matrix.length; i++) {
          str += "[";
          for (int j=0; j<matrix[i].length; j++) {
            str += matrix[i][j];
            if (j != matrix[i].length-1) str += ", ";
          }
          str += "]";
          if (i != matrix.length-1) str += ", ";
      }
      str += "]";
      return str +", "+ d1 +"}";
      }
  }

  protected static final String T1_NAME  = "T1";
  // First form of "enum". 3 is the maximum value - i.e., the value of WaitVal
  protected static final String  T1_FORMAT =
      "{int, {enum : 3}, [double:2,3], double}";

  protected static class T2 {
    public String str1;
    public int count;
    public T1 t1[]; /* Variable length array of type T1_TYPE */
    public int status; /* STATUS_ENUM */

    public String toString () {
      String str = "{\"" + str1 +"\", "+ count +", ";
      str += "<";
      for (int i = 0; i<count; i++) str += t1[i].toString();
      str += ">, ";
      str += (status == WaitVal ? "WaitVal"
           : status == SendVal ? "SendVal"
           : status == ReceiveVal ? "ReceiveVal"
           : status == ListenVal ? "ListenVal" : Integer.toString(status));
      return str +"]";
    }
  }

  protected static final String  T2_NAME = "T2";
  // Alternate form of "enum".
  protected static final String  T2_FORMAT =
      "{string, int, <T1:2>, {enum WaitVal, SendVal, ReceiveVal, ListenVal}}";

  protected static final String  MSG1        = "message1";
  protected static final String  MSG1_FORMAT = "int";

  protected static final String  MSG2        = "message2";
  protected static final String  MSG2_FORMAT = "string";

  protected static final String  QUERY1        = "query1";
  protected static final String  QUERY1_FORMAT = T1_NAME;

  protected static final String  RESPONSE1        = "response1";
  protected static final String  RESPONSE1_FORMAT = T2_NAME;

  protected static final String  MODULE1_NAME = "module1";
  protected static final String  MODULE2_NAME = "module2";
  protected static final String  MODULE3_NAME = "module3";
}

================================================================
```

File “**module1.java**” is the Java equivalent of “**module1.c**”.

It publishes MSG1 and QUERY1 and subscribes to MSG2. It sends MSG1 whenever a “m” is typed at the terminal, send a QUERY1 whenever an “r” is typed, and quits the program when a “q” is typed. It should be run in conjunction with module2.

```
*************************************************************************
import ipc.java.*;

public class module1 extends module {
  private static class msg2Handler implements IPC.HANDLER_TYPE {
    msg2Handler(String theClientData) { clientData = theClientData; }
    public void handle (IPC.MSG_INSTANCE msgRef, Object callData) {
      System.out.println("msg2Handler: Receiving "+
                   IPC.msgInstanceName(msgRef) +" (\""+ callData
                   +"\") ["+ clientData +"]");
    }
    String clientData;
  }

  private static class stdinHnd implements IPC.FD_HANDLER_TYPE {
    stdinHnd(String theClientData) { clientData = theClientData; }
    public void handle (int fd) {
      try {
      int in = System.in.read();

      if (in == 'q' || in == 'Q') {
        IPC.disconnect();
        System.exit(-1);
      } else if (in == 'm' || in == 'M') {
```

```
      int i1 = 42;
      System.out.println("\n  IPC.publishData(\""+ MSG1 +"\", "+ i1 +")");
      IPC.publishData(MSG1, i1);
    } else if (in == 'r' || in == 'R') {
      T1 t1 = new T1();
      t1.i1 = 666;
      t1.status = SendVal;
      t1.matrix = new double[][] {{0.0, 1.0, 2.0}, {1.0, 2.0, 3.0}};
      t1.d1 = java.lang.Math.PI;

      System.out.println("\n  r1 = IPC.queryResponseData(\""+ QUERY1
                    +"\", "+ t1 +", T2.class, IPC.IPC_WAIT_FOREVER)");
      T2 r1 = (T2)IPC.queryResponseData(QUERY1, t1, T2.class,
                                 IPC.IPC_WAIT_FOREVER);
      System.out.println("\n  Received response: "+ r1.toString());
    } else {
      System.out.println("stdinHnd ["+ clientData +"]: Received "+(char)in);
    }
    // Read in any extra bytes
    while (System.in.available() > 0) System.in.read();
    } catch (Exception e) { e.printStackTrace(); }
  }
  String clientData;
}

private static class handlerChangeHnd implements IPC.CHANGE_HANDLE_TYPE {
    public void handle (String msgName, int num) {
      System.err.println("HANDLER CHANGE: "+ msgName +": "+ num);
    }
}

private static class handlerChangeHnd2 implements IPC.CHANGE_HANDLE_TYPE {
    public void handle (String msgName, int num) {
      System.err.println("HANDLER CHANGE2: "+ msgName +": "+ num);
    }
}

private static class connect1Hnd implements IPC.CONNECT_HANDLE_TYPE {
  public void handle (String moduleName) {
    System.err.println("CONNECT1: Connection from "+ moduleName);
    System.err.println("          Confirming connection ("+
                IPC.isModuleConnected(moduleName) +")");
  }
}

private static class connect2Hnd implements IPC.CONNECT_HANDLE_TYPE {
  public void handle (String moduleName) {
    System.err.println("CONNECT2: Connection from "+ moduleName);
    System.err.println("          Number of handlers: "+
                IPC.numHandlers(MSG1));
  }
}

private static class disconnect1Hnd implements IPC.CONNECT_HANDLE_TYPE {
  static boolean first = true;
  public void handle (String moduleName) {
    System.err.println("DISCONNECT: "+ moduleName);
```

```
      if (first) IPC.unsubscribeConnect(connect1Hnd.class);
      else IPC.unsubscribeConnect(connect2Hnd.class);
      if (first) IPC.unsubscribeHandlerChange(MSG1, handlerChangeHnd2.class);
      else IPC.unsubscribeHandlerChange(MSG1, handlerChangeHnd.class);
      first = false;
    }
  }

  public static void main (String args[]) throws Exception {
    // Connect to the central server
    System.out.println("\nIPC.connect(\""+ MODULE1_NAME +"\")");
    IPC.connect(MODULE1_NAME);

    IPC.subscribeConnect(new connect1Hnd());
    IPC.subscribeConnect(new connect2Hnd());
    IPC.subscribeDisconnect(new disconnect1Hnd());

    // Define the named formats that the modules need
    System.out.println("\nIPC.defineFormat(\""+ T1_NAME +"\", \""+
                   T1_FORMAT +"\")");
    IPC.defineFormat(T1_NAME, T1_FORMAT);
    System.out.println("\nIPC.defineFormat(\""+ T2_NAME +"\", \""+
                   T2_FORMAT +"\")");
    IPC.defineFormat(T2_NAME, T2_FORMAT);

    // Define the messages that this module publishes
    System.out.println("\nIPC.defineMsg(\""+ MSG1 +"\", \""+
                   MSG1_FORMAT +"\")");
    IPC.defineMsg(MSG1, MSG1_FORMAT);

    IPC.subscribeHandlerChange(MSG1, new handlerChangeHnd());
    IPC.subscribeHandlerChange(MSG1, new handlerChangeHnd2());

    System.out.println("\nIPC.defineMsg(\""+ QUERY1 +"\", \""+
                   QUERY1_FORMAT +"\")");
    IPC.defineMsg(QUERY1, QUERY1_FORMAT);
    IPC.subscribeHandlerChange(QUERY1, new handlerChangeHnd());

    // Subscribe to the messages that this module listens to.
    // NOTE: No need to subscribe to the RESPONSE1 message, since it is a
    //       response to a query, not a regular subscription!
    System.out.println("\nIPC.subscribeData(\""+ MSG2 +"\", new msg2Handler(\""+
                   MODULE1_NAME +"\"), String.class)");
    IPC.subscribeData(MSG2, new msg2Handler(MODULE1_NAME), String.class);

    // Subscribe a handler for tty input.
    //   Typing "q" will quit the program; Typing "m" will send MSG1;
    //   Typing "r" will send QUERY1 ("r" for response)
    System.out.println("\nIPC_subscribeFD(0, new stdinHnd(\""+
                   MODULE1_NAME +"\"))");
    IPC.subscribeFD(0, new stdinHnd(MODULE1_NAME));

    System.out.println("\nType 'm' to send "+ MSG1 +"; Type 'r' to send "+
                   QUERY1 +"; Type 'q' to quit");
    IPC.dispatch();

    IPC.disconnect();
```

```
}
```

```
======================================================================
```

The file “**module2.java**” is the Java equivalent of **module2.c**.

It is a test program for IPC that publishes MSG2, and subscribes to MSG1 and QUERY. It listens for MSG1 and prints out message data. When QUERY1 is received, it publishes MSG1 and responds to the query with RESPONSE1. It exits when 'q' is typed at terminal. module2 should be run in conjunction with module1.

```
************************************************************************
import ipc.java.*;

public class module2 extends module {
  private static class msg1Handler implements IPC.HANDLER_TYPE {
    msg1Handler(String theClientData) { clientData = theClientData; }
    public void handle (IPC.MSG_INSTANCE msgRef, Object callData) {
      System.out.println("msg1Handler: Receiving "+
                  IPC.msgInstanceName(msgRef) +" ("+ callData
                  +") ["+ clientData +"]");
    }
    String clientData;
  }

  private static class queryHandler implements IPC.HANDLER_TYPE {
    queryHandler(String theClientData) { clientData = theClientData; }
    public void handle (IPC.MSG_INSTANCE msgRef, Object callData) {
      System.out.println("queryHandler: Receiving "+
                  IPC.msgInstanceName(msgRef) +" ["+ clientData +"]");
      System.out.println(callData.toString());

      /* Publish this message -- all subscribers get it */
      String str1 = "Hello, world";
      System.out.println("\n  IPC.publishData(\""+ MSG2 +"\", \""+ str1 +"\")");
      IPC.publishData(MSG2, str1);

      T2 t2 = new T2();
      t2.str1 = str1;
      /* Variable length array of one element */
      t2.t1 = new T1[1];
      t2.t1[0] = (T1)callData;
      t2.count = 1;
      t2.status = ReceiveVal;

      /* Respond with this message -- only the query handler gets it */
      System.out.println("\n  IPC.respondData("+ msgRef +", \""+
                  RESPONSE1 +"\", "+ t2 +")");
      IPC.respondData(msgRef, RESPONSE1, t2);
    }
    String clientData;
  }

  private static class stdinHnd implements IPC.FD_HANDLER_TYPE {
    stdinHnd(String theClientData) { clientData = theClientData; }
    public void handle (int fd) {
      try {
        int in = System.in.read();

        if (in == 'q' || in == 'Q') {
          IPC.disconnect();
          System.exit(-1);
        } else {
          System.out.println("stdinHnd ["+ clientData +"]: Received "+
                        (char)in);
        }
        // Read in any extra bytes
      while (System.in.available() > 0) System.in.read();
      } catch (Exception e) { e.printStackTrace(); }
    }
    String clientData;
  }

  public static void main (String args[]) throws Exception {
    /* Connect to the central server */
    System.out.println("\nIPC.connect(\""+ MODULE2_NAME +"\")");
    IPC.connect(MODULE2_NAME);

    /* Define the messages that this module publishes */
    System.out.println("\nIPC.defineMsg(\""+ MSG2 +"\", \""+
                  MSG2_FORMAT +"\")");
    IPC.defineMsg(MSG2, MSG2_FORMAT);

    System.out.println("\nIPC.defineMsg(\""+ RESPONSE1 +"\", \""+
                  RESPONSE1_FORMAT +"\")");
    IPC.defineMsg(RESPONSE1, RESPONSE1_FORMAT);

    /* Subscribe to the messages that this module listens to. */
    System.out.println("\nIPC.subscribeData(\""+ MSG1 +"\", new msg1Handler(\""+
                  MODULE2_NAME +"\"), int.class)");
    IPC.subscribeData(MSG1, new msg1Handler(MODULE2_NAME), int.class);

    System.out.println("\nIPC.subscribeData(\""+ QUERY1
                  +"\", new queryHandler(\""+
                  MODULE2_NAME +"\"), T1.class)");
    IPC.subscribeData(QUERY1, new queryHandler(MODULE2_NAME), T1.class);

    /* Subscribe a handler for tty input. Typing "q" will quit the program. */
    System.out.println("\nIPC_subscribeFD(0, new stdinHnd(\""+
                  MODULE1_NAME +"\"))");
    IPC.subscribeFD(0, new stdinHnd(MODULE1_NAME));

    System.out.println("\nType 'q' to quit");
    IPC.dispatch();

    IPC.disconnect();
  }
}
```

File “**module.java**” is the Java equivalent of “**module.h**”.

```
**********************************************************************
import IPC

WaitVal    = 0
SendVal    = 1
ReceiveVal = 2
ListenVal  = 3

class T1(IPC.IPCdata) :
  _fields = ('i1', 'status', 'matrix', 'd1')

class T2(IPC.IPCdata) :
  _fields = ('str1', 'count', ('t1', T1), 'status')

T1_NAME  = "T1"
# First form of "enum". 3 is the maximum value -- i.e., the value of WaitVal
T1_FORMAT = "{int, {enum : 3}, [double:2,3], double}";

T2_NAME = "T2"
# Alternate form of "enum".
T2_FORMAT = \
      "{string, int, <T1:2>, {enum WaitVal, SendVal, ReceiveVal, ListenVal}}"

MSG1        = "message1"
MSG1_FORMAT = "int"

MSG2        = "message2"
MSG2_FORMAT = "string"

QUERY1        = "query1"
QUERY1_FORMAT = T1_NAME

RESPONSE1        = "response1"
RESPONSE1_FORMAT = T2_NAME

MODULE1_NAME = "module1"
MODULE2_NAME = "module2"
MODULE3_NAME = "module3"
==================================================================
```

File “**module1.py**” is the Python equivalent of “**module1.c**”.

It publishes MSG1 and QUERY1 and subscribes to MSG2. It sends MSG1 whenever a “m” is typed at the terminal, send a QUERY1 whenever an “r” is typed, and quits the program when a “q” is typed. It should be run in conjunction with module2.

```
********************************************************************************
import sys
import IPC
from module import *

def msg2Handler (msgRef, callData, clientData) :
  print "msg2Handler: Receiving %s (%s) [%s] " % \
      (IPC.IPC_msgInstanceName(msgRef), callData, clientData)
```

```
done = False

def stdinHnd (fd, clientData) :
  global done
  input = sys.stdin.readline()

  if (input[0] == 'q' or input[0] == 'Q') :
    IPC.IPC_disconnect()
    done = True
  elif (input[0] == 'm' or input[0] == 'M') :
    i1 = 42
    print "\n  IPC_publishData(%s, %d)" % (MSG1, i1)
    IPC.IPC_publishData(MSG1, i1)
  elif (input[0] == 'r' or input[0] == 'R') :
    t1 = T1()
    t1.i1 = 666
    t1.status = SendVal
    t1.matrix = ((0.0, 1.0, 2.0), (1.0, 2.0, 3.0))
    t1.d1 = 3.14159
    print "\n  IPC_queryResponseData(%s, %s, IPC_WAIT_FOREVER, %s)" % \
          (QUERY1, t1, T1.__name__)
    (r1, ret) = IPC.IPC_queryResponseData(QUERY1, t1, IPC.IPC_WAIT_FOREVER, T1)
    print "\n  Received response"
    IPC.IPC_printData(IPC.IPC_msgFormatter(RESPONSE1), sys.stdout, r1)
  else :
    print "stdinHnd [%s]: Received %s" % (clientData, input),

def handlerChangeHnd (msgName, num, clientData) :
  print "HANDLER CHANGE: %s: %d" % (msgName, num)

def handlerChangeHnd2 (msgName, num, clientData) :
  print "HANDLER CHANGE2: %s: %d" % (msgName, num)

def connect1Hnd (moduleName, clientData) :
  print "CONNECT1: Connection from %s" % moduleName
  print "          Confirming connection (%d)" % \
        IPC.IPC_isModuleConnected(moduleName)

def connect2Hnd (moduleName, clientData) :
  print "CONNECT2: Connection from %s" % moduleName
  print "          Number of handlers: %d" % IPC.IPC_numHandlers(MSG1)

first = True

def disconnect1Hnd (moduleName, clientData) :
  global first
  print "DISCONNECT:", moduleName
  if (first) : IPC.IPC_unsubscribeConnect(connect1Hnd)
  else : IPC.IPC_unsubscribeConnect(connect2Hnd)
  if (first) : IPC.IPC_unsubscribeHandlerChange(MSG1, handlerChangeHnd2)
  else : IPC.IPC_unsubscribeHandlerChange(MSG1, handlerChangeHnd)
  first = False

def main () :
  global done, first
  done = False; first = True
```

```
# Connect to the central server
print "\nIPC.IPC_connect(%s)" % MODULE1_NAME
print IPC.IPC_connect, sys.stdin, sys.stdin.fileno()
IPC.IPC_connect(MODULE1_NAME)
print "HERE1"

IPC.IPC_subscribeConnect(connect1Hnd, None)
IPC.IPC_subscribeConnect(connect2Hnd, None)
IPC.IPC_subscribeDisconnect(disconnect1Hnd, None)

# Define the named formats that the modules need
print "\nIPC.IPC_defineFormat(%s, %s)" % (T1_NAME, T1_FORMAT)
IPC.IPC_defineFormat(T1_NAME, T1_FORMAT)
print "\nIPC.IPC_defineFormat(%s, %s)" % (T2_NAME, T2_FORMAT)
IPC.IPC_defineFormat(T2_NAME, T2_FORMAT)

# Define the messages that this module publishes
print "\nIPC.IPC_defineMsg(%s, IPC_VARIABLE_LENGTH, %s)" %(MSG1, MSG1_FORMAT)
IPC.IPC_defineMsg(MSG1, IPC.IPC_VARIABLE_LENGTH, MSG1_FORMAT)

IPC.IPC_subscribeHandlerChange(MSG1, handlerChangeHnd, None)
IPC.IPC_subscribeHandlerChange(MSG1, handlerChangeHnd2, None)

print "\nIPC.IPC_defineMsg(%s, IPC_VARIABLE_LENGTH, %s)" % \
      (QUERY1, QUERY1_FORMAT)
IPC.IPC_defineMsg(QUERY1, IPC.IPC_VARIABLE_LENGTH, QUERY1_FORMAT)
IPC.IPC_subscribeHandlerChange(QUERY1, handlerChangeHnd, None)

# Subscribe to the messages that this module listens to.
# NOTE: No need to subscribe to the RESPONSE1 message, since it is a
#       response to a query, not a regular subscription!
print "\nIPC.IPC_subscribeData(%s, msg2Handler, %s)" % (MSG2, MODULE1_NAME)
IPC.IPC_subscribe(MSG2, msg2Handler, MODULE1_NAME)

# Subscribe a handler for tty input.
#  Typing "q" will quit the program; Typing "m" will send MSG1;
#  Typing "r" will send QUERY1 ("r" for response)
print "\nIPC.IPC_subscribeFD(%d, stdinHnd, %s)" % \
      (sys.stdin.fileno(), MODULE1_NAME)
IPC.IPC_subscribeFD(sys.stdin.fileno(), stdinHnd, MODULE1_NAME)

print "\nType 'm' to send %s; Type 'r' to send %s; Type 'q' to quit" % \
      (MSG1, QUERY1)

while (not done) : IPC.IPC_listen(250)

IPC.IPC_disconnect()
```

==

The file “**module2.py**” is the Python equivalent of **module2.c**.

It is a test program for IPC that publishes MSG2, and subscribes to MSG1 and QUERY. It listens for MSG1 and prints out message data. When QUERY1 is received, it publishes MSG1 and responds to the query with RESPONSE1. It exits when 'q' is typed at terminal. module2 should be run in conjunction with module1.

```
**************************************************************************
import sys
import IPC
from module import *

def msg1Handler (msgRef, callData, clientData) :
  print "msg1Handler: Receiving %s (%d) [%s] " % \
      (IPC.IPC_msgInstanceName(msgRef), callData, clientData)

def queryHandler (msgRef, t1, clientData) :
  print "queryHandler: Receiving %s [%s]",  \
      (IPC.IPC_msgInstanceName(msgRef), clientData)
  IPC.IPC_printData(IPC.IPC_msgInstanceFormatter(msgRef), sys.stdout, t1)

  # Publish this message -- all subscribers get it
  str1 = "Hello, world"
  print '\n  IPC.IPC_publishData(%s, "%s")' % (MSG2, str1)
  IPC.IPC_publishData(MSG2, str1)

  t2 =  T2()
  t2.str1 = str1
  # Variable length array of one element
  t2.t1 = [T1()]
  t2.t1[0] = t1
  t2.count = 1
  t2.status = ReceiveVal

  # Respond with this message -- only the query handler gets it
  print "\n  IPC.IPC_respondData(%s, %s, %s)" % (msgRef, RESPONSE1, t2)
  IPC.IPC_respondData(msgRef, RESPONSE1, t2)

done = False

def stdinHnd (fd, clientData) :
  global done
  input = sys.stdin.readline()

  if (input[0] == 'q' or input[0] == 'Q') :
    IPC.IPC_disconnect()
    done = True
  else :
    print "stdinHnd [%s]: Received %s" % (clientData, input)

def main () :
  global done
  done = False
  # Connect to the central server
  print "\nIPC.IPC_connect(%s)" % MODULE2_NAME
```

```
IPC.IPC_connect(MODULE2_NAME)

# Define the messages that this module publishes
print "\nIPC.IPC_defineMsg(%s, IPC_VARIABLE_LENGTH, %s)" % \
      (MSG2, MSG2_FORMAT)
IPC.IPC_defineMsg(MSG2, IPC.IPC_VARIABLE_LENGTH, MSG2_FORMAT)

print "\nIPC.IPC_defineMsg(%s, IPC_VARIABLE_LENGTH, %s)" % \
      (RESPONSE1, RESPONSE1_FORMAT)
IPC.IPC_defineMsg(RESPONSE1, IPC.IPC_VARIABLE_LENGTH, RESPONSE1_FORMAT)

# Subscribe to the messages that this module listens to
print "\nIPC.IPC_subscribeData(%s,%s, %s)" % \
      (MSG1, msg1Handler.__name__, MODULE2_NAME)
IPC.IPC_subscribeData(MSG1, msg1Handler, MODULE2_NAME)

print "\nIPC.IPC_subscribeData(%s, %s, %s, %s)" % \
      (QUERY1 , queryHandler.__name__, MODULE2_NAME, T1.__name__)
IPC.IPC_subscribeData(QUERY1, queryHandler, MODULE2_NAME, T1)

# Subscribe a handler for tty input. Typing "q" will quit the program.
print "\nIPC_subscribeFD(%d, stdinHnd, %s)" % \
      (sys.stdin.fileno(), MODULE2_NAME)
IPC.IPC_subscribeFD(sys.stdin.fileno(), stdinHnd, MODULE2_NAME)

print "\nType 'q' to quit"
while (not done) : IPC.IPC_listen(250)

IPC.IPC_disconnect()
```

Appendix B xdrgen

Purpose of `xdrgen`

Defining an IPC message requires a format string, which corresponds to the data structure of the message. Typically, the designer of the message has defined this format string by hand, as a macro in the same header file which defines the data structure. The `xdrgen` parser automates this process. It parses an XDR data structure specification (similar to a list of `C` type definitions) and generates a `C` header, which includes both type definitions and macros defining the IPC format strings.

Automating this process helps to avoid inconsistencies between the `C` data structure and the IPC format string. In our experience, inconsistencies are often introduced when the data structure is changed, but the person modifying the code is not aware that the format string must also be changed. These inconsistencies can lead to garbled binary messages, which are sometimes very difficult to track down.

Running `xdrgen` for the first time

Installing IPC will place an `xdrgen` binary in the same location as the central server. To see an example run of `xdrgen`, run the following command in the `xdrgen` directory of the IPC distribution:

```
> xdrgen example.xdr example.xdr.h
```

This will output `example.xdr.h`, a C header file based on the XDR specification in `example.xdr`. You can compare the two to get a quick idea of the relationship between XDR and C.

`xdrgen` command-line options

```
usage: xdrgen OPTIONS <xdrFile>
            [outputHeaderFile]
-h or --help   Print this help
--lang=[c,c++] Change language for
             header output(default: C++)
```

When `xdrgen` is run, it will parse `xdrFile` and output the resulting C header to `outputHeaderFile` (or to `stdout` if `outputHeaderFile` isn't specified). Specifying `C++` for the header output language (the default) will cause the header to use some `C++` language features that are not supported by `C`. The differences will be discussed below.

Basic `xdrgen` type declarations

We now move on to the types of declarations that `xdrgen` can parse. There are two basic kinds of declarations. The first is a `typedef`. The declaration:

```
typedef int foo;
```

generates a `typedef` and a macro in the output header file:

```
typedef int foo;
#define foo_IPC_FORMAT "int"
```

The second kind of declaration is a `struct`. The declaration:

```
struct Zoo {
  int foo;
  int goo;
};
```

generates a `struct` and a macro in the output header file:

```
typedef struct _Zoo {
  int foo;
  int goo;
} Zoo;
#define Zoo_IPC_FORMAT "{int, int}"
```

If the header language is `C++`, the generated code is slightly different:

```
struct Zoo {
  int foo;
  int goo;
  #define Zoo_IPC_FORMAT "{int, int}"
  static const char *getIPCFormat(void)
    { return Zoo_IPC_FORMAT; }
};
```

If you are using a `C++` compiler, the `C++` output has the following advantages:

- As discussed below, `xdrgen` allows arbitrary code to be inserted at the end of a struct definition. In C++, this can be used to define member functions. Defining the struct starting with "struct Zoo" instead of "typedef struct _Zoo " allows one to define constructors for Zoo in the arbitrary code section.

- Code, which requires the IPC format, can access either the macro Zoo_IPC_FORMAT or the member function Zoo::getIPCFormat(), which may enable you to write cleaner code.
- You may nest struct declarations to arbitrary depths, and use previously defined types in declarations of new types, as in the following:

```
struct MyIncludableStruct {
  int foo;
  struct { int a; } goo;
};
struct MyNestedStruct {
  MyIncludableStruct b;
  struct {
    char a;
    MyIncludableStruct b2;
  } roo;
};
```

You may *not* declare multiple fields of the same type in one line, so the C construction “int a, b;” must be replaced with “int a; int b;”.

Primitive types

The following struct definition has fields with all of the supported primitive types:

```
struct PrimitiveTypes {
  string a<>;
  unsigned char b;
  char c;
  unsigned int d;
  int e;
  bool f;
  float g;
  double h;
};
```

Some notes about how these types are used:

- The string field is followed by <> because strings are always variable-length arrays in XDR. This will be discussed more later.
- The bool type is not defined by default in C. Therefore, whenever xdrgen creates C-language output, it includes a definition for bool as an enumerated type compatible with the built-in C++ definition. In C++, a bool is a 4-byte data structure that takes on the values false=0 or true=1. In the IPC format string, the bool field is represented as “int”.
- For the other types, there is a straightforward mapping both to C data types and to IPC format strings (see Section 3).

Fixed-length arrays

Fixed length arrays in the XDR file are mapped directly to fixed-length arrays in C. For the XDR declarations:

```
typedef unsigned char ImagePixel[3];
struct Transform {
  double mat[4][4];
};
```

we get the following header output (abbreviated for clarity):

```
typedef unsigned char ImagePixel[3];
#define ImagePixel_IPC_FORMAT
        "[uchar:3]"
struct Transform {
  double mat[4][4];
};
#define Transform_IPC_FORMAT
        "{[double:4,4]}"
```

Variable-length arrays

Variable-length arrays are specified in XDR using angle brackets <>. For the XDR declaration:

```
struct Image {
int rows;
int cols;
unsigned char data<><>;
};
```

we get the following header output (abbreviated for clarity):

```
struct Image {
  int rows;
  int cols;
  unsigned char *data;
};
#define Image_IPC_FORMAT
        "{int, int,<uchar:1,2>}"
```

Variable-length array fields in IPC must be inside a `struct`, and the length in each dimension of the array must correspond to an `int` or `unsigned int` field of the `struct`. In IPC, which fields of the `struct` are used for each dimension is controlled by the format string. `xdrgen` has the following stricter requirements:

- If the variable-length array has `n` dimensions, the struct must have exactly `n+1` fields.
- The first `n` fields of the `struct` must have type `int` or `unsigned int`.
- The `int` fields of the `struct` correspond to dimensions of the array in order from left to right (most significant dimension to least significant).

`xdrgen` is designed this way to make it clear where the size of each dimension of the array is coming from, and to remove the need for extra language features to specify how `int` fields correspond to array dimensions. This simplicity comes at the cost of discarding some of IPC's flexibility.

Fixed and variable-length array dimensions cannot be mixed (except for strings – see below), so the following declaration is illegal:

```
struct Alpha {
  int size;
  char beta[5]<>;
}
```

However, a similar effect could be achieved with the following declaration:

```
struct Alpha {
  int size;
  struct { char data[5]; } beta<>;
};
```

For consistency with XDR, `xdrgen` allows a maximum possible length to be declared for a variable-length array, as in the following definition:

```
struct Gamma {
  int delta;
  float epsilon<20>;
};
```

However, there is no notion of a maximum length for a variable-length array in IPC, so the length does not currently appear in the header output.

Enumerated types

Enumerated types in XDR are mapped directly to enumerated types in `C`. From the declaration

```
enum Color {
  RED, ORANGE, YELLOW
};
```

we get the following header output:

```
enum Color {
  RED,
  ORANGE,
  YELLOW
};
#define Color_IPC_FORMAT
        "{enum RED,ORANGE,YELLOW}"
```

Values for the named options of an enumerated type can also be specified, but IPC is only flexible enough to handle a consecutive set of options, so if any values are specified, `xdrgen` represents the field as "`int`" to IPC (with the disadvantage that IPC cannot expand values to option names during data logging). For the declaration:

```
enum Mixed {
  TWO = 2, FOUR, SIX = 6
};
```

we get the following header output:

```
enum Mixed {
  TWO = 2,
  FOUR,
  SIX = 6
};
#define Mixed_IPC_FORMAT "int"
```

More about the string type

Strings are a special case. A string is a null-terminated array of `char`. IPC does not need to have an integer dimension for the size of the array because it can detect the size from the null termination. Therefore the last variable-length dimension of a string:

- is required to be present (as specified in RFC 1014),
- is ignored in the IPC format generated by `xdrgen`,
- does not need to appear in a struct with a corresponding int field.

Also, fixed- and variable-length arrays of strings are allowed, as in the following definition:

```
struct ExecCall {
  struct {
    int argc;
    string argv<>;
  } args;
  string envVars[20]<>;
};
```

which produces the header output (abbreviated for clarity):

```
struct ExecCall {
  struct {
    int argc;
    char **argv;
  } args;
  char *envVars[20];
};
#define ExecCall_IPC_FORMAT
        "{{int,<string:1},[string:20]}"
```

Arbitrary code sections

Your XDR file can include arbitrary code sections wrapped in the delimiters `%{` and `%}`. Arbitrary code can be placed at the beginning or end of the file, between declarations, or at the end of a struct declaration, just before the closing `}` character. The arbitrary code will be copied into the generated header at the corresponding point in the C code.

For example, the XDR file text:

```
%{
#include "my_arbitrary_code.h"
#define N 3
extern int foo;
%}
typedef double Meters;
%{
extern Meters length;
%}
struct Roo {
  int a;
  char b;
%{
  Roo(int _a, char _b) { a=_a; b=_b; }
%}
};
```

generates the header output (abbreviated for clarity):

```
#include "my_arbitrary_code.h"
#define N 3
extern int foo;
typedef double Meters;
#define Meters_IPC_FORMAT "double"
extern Meters length;
struct Roo {
  int a;
  char b;
  Roo(int _a, char _b) { a=_a; b=_b; }
};
#define Roo_IPC_FORMAT "{int, char}"
```

External format definitions

You can use external format definitions if the `structs` you define using XDR contain other data types that are not defined using XDR. This could happen if you want to manually define the IPC format for a data type using an IPC feature not supported by `xdrgen`, or if the type definition comes from a standard system include file. For instance, the declaration:

```
ipc_type ExternalStruct1;
```

tells `xdrgen` to expect that the type `ExternalStruct1` has an IPC format defined in the macro `ExternalStruct1_IPC_FORMAT`. The macro definition must appear before the generated code for any `struct` that includes `ExternalStruct1`, which means that it should appear either in an arbitrary code section of the XDR file or it should be included via an `#include` directive in an arbitrary code section.

Once ExternalStruct1 is declared using ipc_type, it can be included in subsequent struct declarations without causing a warning. For example, the declaration:

```
struct IncExternalStruct {
  int a;
  ExternalStruct1 s1;
}
```

generates the header output (abbreviated for clarity):

```
struct IncExternalStruct {
  int a;
  ExternalStruct1 s1;
}
#define IncExternalStruct_IPC_FORMAT
  "{int," ExternalStruct1_IPC_FORMAT "}"
```

You can also manually define the IPC format for a type using ipc_type. Thus, the declaration:

```
ipc_type ExternalStruct2 =
         "{char, double}";
```

tells xdrgen to use the given format string instead of trying to refer to the macro ExternalStruct2_IPC_FORMAT.

Formal XDR language definition

The xdrgen parser aims to parse the XDR language (as specified by Sun Microsystems in RFC 1014) wherever this makes sense. However, there are both unsupported features and extensions in the xdrgen input language:

- RFC 1014 specifies a mapping between an XDR data type specification and the binary format of the corresponding network message. Because XDR syntax is so similar to C syntax, there is also an implicit mapping between the XDR data type and the corresponding C data type. It is this second mapping that xdrgen tries to capture. The IPC format string for a data type is generated from the C data type according to the rules in Section 3. There is no reason to expect that IPC network messages will follow the XDR binary packing rules.
- Unsupported feature: union types. There are no corresponding types in IPC.
- Unsupported feature: hyper (8-byte integer) types. There are no corresponding types in IPC.
- Unsupported feature: optional data (the XDR * syntax). Supported in IPC, but not implemented by xdrgen.
- Extension: opaque replaced with char and unsigned char. A char type need not be in an array.
- Extension: arbitrary code sections.
- Extension: multi-dimensional arrays.

Those interested in a full grammar for the xdrgen input language can look at bison input file src/XDRGen/XDR.y in the IPC distribution.

xdrgen was developed and documented by Trey Smith, February 2001.

Index