

The Mjølner System

Process Library

Reference Manual

Mjølner Informatics Report

MIA 94-29 (1.2)

October 1997

Copyright © 1990-97 Mjølner Informatics ApS.

All rights reserved.

No part of this document may be copied or distributed
without the prior written permission of Mjølner Informatics

Contents

1	INTRODUCTION.....	3
2	MANIPULATING PROCESSES	4
2.1	CHILD PROCESSES	4
2.2	THIS PROCESS AND ITS ENVIRONMENT.....	5
3	COMMUNICATING WITH OTHER PROCESSES.....	6
3.1	COMMUNICATION CONCEPTS	6
3.2	SCHEDULING.....	7
3.3	THE TWO FAMILIES OF SOCKETS.....	8
3.4	THE FRAGMENT BASICSOCKET.....	8
3.4.1	<i>The Patterns of basicsocket.....</i>	<i>8</i>
3.5	THE FRAGMENT BINARYSOCKET.....	10
3.5.1	<i>The Patterns of binarysocket.....</i>	<i>10</i>
3.6	THE FRAGMENT STREAMSOCKET.....	10
3.7	THE FRAGMENT COMMPIPE	11
3.8	SOCKETGENERATORS	11
3.8.1	<i>The patterns of socketgenerator.....</i>	<i>11</i>
3.8.2	<i>The patterns of streamgenerator and binarygenerator.....</i>	<i>11</i>
3.9	ERROR HANDLING.....	12
3.9.1	<i>Error Callbacks.....</i>	<i>12</i>
3.9.2	<i>Error Propagation.....</i>	<i>12</i>
3.9.3	<i>Categories of Errors.....</i>	<i>13</i>
3.10	TIMEOUT MANAGEMENT.....	13
4	ADDRESSES.....	15
4.1	SPECIFICATION OF CONNECTION REQUIREMENTS	15
4.2	THE ABSTRACT LEVEL.....	16
4.3	THE CONCRETE LEVEL.....	16
5	MANAGING A POOL OF CONNECTIONS	18
6	THE DEMO FILES	20
7.1	PIPELINE, CONSUMER AND PRODUCER.....	20
7.2	FIRSTPROGRAM AND OTHERPROGRAM.....	20
7.3	STREAMCOUNTERSERVER AND STREAMCOUNTERCLIENT	20
7.4	BINARYCOUNTERSERVER AND BINARYCOUNTERCLIENT	21
7.5	XPILOTGAMES	21
7.6	REPCHATCLIENT AND REPCHATSERVER	21
8	KNOWN BUGS AND INCONVENIENCES.....	22
8.1	GENERAL	22
8.2	WINDOWS	23
8.3	MACINTOSH.....	23
9	INTERFACE DESCRIPTION	24
9.1	COMMADDRESS	24
9.2	ERRORCALLBACK.....	30
9.3	BASICSOCKET	31
9.4	BINARYSOCKET	33
9.5	STREAMSOCKET	35
9.6	SOCKETGENERATOR.....	37

9.7	BINARYGENERATOR.....	40
9.8	STREAMGENERATOR.....	40
9.9	COMMPOOL.....	40
9.10	SYSTEMCOMM.....	43
9.11	PROCESSMANAGER.....	43
9.12	COMMPIPE.....	45
	REFERENCES.....	47
	INDEX.....	49

1 Introduction

This document describes the version 1.6 of the process library in the Mjølnir System. This library implements support for manipulating operating system processes and for communicating with them. All fragments in the process library demand that the program uses the BETA simulated concurrency, i.e. the slot `program:descriptor` must be a specialization of `systemenv`. In return, one does not have to explicitly transfer the thread of control by `suspending` when an operation is about to block - the `systemenv` scheduler and the process library cooperate to make it look like implicit scheduling. This ensures that co-routines which can proceed with their work will never be prevented from this because of a blocking communication operation in some other co-routine.

The fragment dealing with the manipulation of processes is `processmanager`. `Processmanager` supports starting a child process, stopping it, and similar things.

Processmanager

The fragments dealing with communication between processes are `basicsocket`, `streamsocket`, `socketgenerator` and a few variations hereof.

Communication

Some aspects of support for the communication between processes have been separated into the fragments `commaddress` and `errorcallback`. `commaddress` defines a hierarchy of patterns, which model addresses (destinations for communications) in a platform independent way. `errorcallback` defines a few patterns used for error handling in this library.

On top of the support for single communication connections, `commpool` implements support for holding a set of connections, and providing concurrency-secure access to these connections by means of platform independent addresses, i.e. instances of patterns in `commaddress`. This abstracts away the need to open and close these connections: if connections to the required destination is available, one of them will be used, otherwise a new connection will automatically be opened. If the process hits a maximum limit for the number of open connections, a least recently used (and currently unused) connection will be closed.

Connections

2 Manipulating Processes

First, a bit of terminology. A binary file is a diskfile, from which the operating system is able to create a process, which is then called an instance of the binary. A process is a dynamic entity within a computer which has an internal state and may interact with other processes. So there may be more than one process which is instantiated from any given binary file, and these processes are by no means the same thing. Here, each BETA object which is an instance of the pattern `process`, models one process. If you want to manipulate more than one instantiation of a given binary, use more than one `process` object.

2.1 Child Processes

The fragment `processmanager` is concerned with child processes. An instance of the `process` pattern in this fragment is attached to a binary file by initializing it with a file specification, like

```
'/bin/someApplication' -> aProcess.init;
```

Process

In the following, `aProcess` denotes an instance of the pattern `process`, which has been attached to a binary file.

Arguments and instantiating

One has the option to set up arguments for an instantiation of the binary, using `aProcess.argument.append`, once for each argument. Afterwards, the process can be instantiated with `aProcess.start`. In the following, this instantiation is referred to as the child process. When it has been started, it is possible to change its life cycle and to adjust to it: `aProcess.stop` causes the child process to be killed, `aProcess.awaitStopped` causes this process to sleep until the child process terminates, and `aProcess.stillRunning` is a predicate which returns true if the child process has not yet terminated.

The `onStart` virtual is a hook, into which one can put code to be executed immediately after the child process has been started, and the `onStop` virtual is a hook which is executed when `stop` has stopped the process. Please notice that `onStop` will NOT be executed in the (typical) case when the child process terminates for any other reason, e.g. when it terminates normally.

Inter-process communication

The remaining pattern attributes of `process` are concerned with inter-process communication. The network of inter-process communication must be defined before the child processes are started. `ConnectToProcess` and `connectInPipe` enter a reference to another `process` object and connect the referred child processes in a pipeline. `redirectFromFile` arranges for the child process to take standard input from the specified file, and `redirectToFile` makes it redirect standard output to the given file.

Finally, `redirectFromChannel` enters the `writeEnd` of a pipe and makes the child process accept standard input from that pipe, and `redirectToChannel` enters the

`readEnd` of a `pipe` and makes the child process send standard output to it. The entered parameter is declared to be a (specialization of a) stream. The reason for this is that a future release may accept a broader range of types of objects entered; it should, for instance, be possible to use sockets.

2.2 This Process and its Environment

`commaddress` defines `thisHost`, that returns the name and IP address on the internet.

Scanning of the command line and other functions that used to be in the process library are now supported in `betaenv`.

3 Communicating with other Processes

3.1 Communication Concepts

Inter-process communication is usually described as “message based” or as “connection based”. In both cases, any primitive communication act has a number of participants, playing roles as the receiving or the transmitting end. In this context, there will always be exactly one transmitting party and one receiving party. There is support for specifying a group address, but there is not currently any ready-made implementation of a group communication protocol.

For a message based communication, each message is sent to an explicitly specified receiver. For a connection based communication, at first a connection between two parties is established. From that point, messages can be transmitted via this connection without any explicit reference to their destination. Here, the model of communication is connection oriented.

Pipes For operating systems that support a notion of standard channels for receiving input and delivering output and possibly other things, it is possible for the communicating processes to be unaware (i.e. independent) of the fact that standard input comes from another process or that standard output goes to another process: It all looks the same as if the data came from a keyboard and went to a display or whatever. On the other hand, this level of abstraction implies that the connection lifetime will be the lifetime of the process and that there cannot be more connections than standard channels. Like standard output and standard input, each connection only supports sending data in one direction. Pipes establish this kind of connections. Use the pattern `pipe`.

Sockets To implement more elaborate patterns of communication, one must be able to create and destroy connections during the execution of a process, and to explicitly choose with whom to communicate. Sockets are used for this, and with sockets, every connection is two-way. Sockets come in two main variants: passive and active. A passive socket is used to define a name, which may be used by active sockets when establishing an actual connection. The interplay is like:


```

Passive: "Here I am! My name is Bob"
...
Active-1: "I want to speak with Bob"
Passive(Bob): "OK, here's a connection"
...
Active-2: "I want to speak with Bob"
Passive(Bob): "OK, here's a connection"
...
Active-3: "I want to speak with Cindy"
(Error: Here's no such thing as "Cindy")
...

```

I.e. active sockets connect by name, and more than one connection may be established by means of one passive socket. The “name” is actually a pair whose first part is an identification of the host (its IP address) and whose second part is an integer (the port number). This pair is unique for each passive socket, at least from the time where the operating system accepts registration of the name until the passive socket is closed. After that, the pair may be reused, that is: the port number may be reused on the given host, if the operating system wishes to do so.

In this library, sockets are also divided along another axis, namely into stream sockets and binary sockets. Stream sockets are specializations of the basic `stream` pattern, and support textual communication. Binary sockets support transfers of blocks of data with a well-known size.

The patterns related to these concepts are: `StreamSocket`, `BinarySocket` and `SocketGenerator`. `SocketGenerators` are used to accept incoming connection requests. When a request arrives, a new socket of the specified type is created and connected to the requesting party.

3.2 Scheduling

Any program using the process library must be a `systemenv` program, because the process library depends heavily on cooperation with the scheduler present in `systemEnv` programs.

Instances of the patterns of these fragments are expected to be executed from BETA co-routines, and such co-routines must tolerate being suspended (de-scheduled) and later re-scheduled as part of the execution of possibly lengthy operations. This means that concurrency control by means of `semaphores`, `monitors`, and the like must be established almost as rigourously as had the co-routines been fully concurrent threads of execution.

BETA co-routines

In return for this increase in complexity, a usually very important reduction in complexity arises from having implicit instead of explicit scheduling. Especially when fitting a new piece into an existing framework it is a great asset to be able to simply “spawn” the new piece as part of an initialization phase and then have it running along with the rest of the program without changing any of the other parts not directly interacting with this new piece.

In more concrete terms, it works like this: Whenever an operation is about to block, the current component will be suspended. It will be resumed some time later, when the requested IO is available. In the meantime, some other component which has requested IO available or is not waiting for IO will be resumed. In this way the follow-

ing liveness property of the program is ensured: it will never be the case that a communication operation by blocking delays the continuation of the execution of all of those components which are either (1) not executing a communication operation or (2) executing a communication operation, but has IO of the requested kind available. Of course, any component can still block the whole system by, for example, entering an infinite loop that does nothing.

There are some operations, that may block the entire process for a while. These include `gethostbyname`, `starting a process`, and waiting for a `process` to stop

3.3 The Two Families of Sockets

Basically, the process library supports two families of sockets: stream sockets and binary sockets. Both are implemented using `basicsocket`.

Stream socket

A stream socket is suitable for transferring data which is readable for human beings, such as the data transferred in a UNIX “talk” session, or the more formal communication between a mail program and an SMTP mail server. A `streamSocket` is a `stream`, so you may “put”, “get” etc. However, do not rely on this kind of socket to transfer data which contains zero-valued bytes, as arbitrary binary data may very well do.

Binary socket

A binary socket is guaranteed to transfer any given block of arbitrary bytes unmodified, but you must always specify the length of the data block when sending. To enable cross-platform communication, the headers of the datablocks are modified internally. The current implementation make little-endian machines (e.g. machines running Linux or Windows NT) transmit their package headers in the format used by big-endian machines. It is your responsibility that the contents of the datablocks are in a format understood by the receiver.

In general, you must have a way of choosing either a binary or a stream variant of a connection to be established, because it is not possible to change a `streamSocket` into a `binarySocket` on the same connection, or vice versa. And each socket object models one connection, so it is not possible to use the same socket object for several different connections - use a fresh object each time instead. For `socketGenerators`, of course, this one-shot-restriction does not apply. See below.

3.4 The Fragment `basicsocket`

The following section describes the top level patterns of `basicsocket`. After that, there is a section with a general discussion of error handling. Finally another section discusses the treatment of timeout.

3.4.1 The Patterns of `basicsocket`

`waitForever` is a constant used to specify an infinite timeout.

`AssignGuard` is used to detect wrong usage of other patterns, and `localhost_IP_number` is the number used by convention to indicate the ‘this host’. None of them are important for the understanding of the fragment.

`sameConnection` answers the question of whether `this` and `other` wraps the same OS level connection. `getPortableAddress` returns a portable address for the connection.

BasicSocket

Use `connect` to connect to a passive socket, like those generated by `socketgenerator`. `connect` establishes a connection to a `(host, port)` pair given either as arguments, or stored in the attributes `port`, `host` or `inetAddr`. You should set only one of `host` and `inetAddr`, as `inetAddr` is set to the internet address of `host` if `inetAddr` is not set. `host` is ignored when `inetAddr` is set.

connect

`Host` must be given in a format like “quercus.daimi.aau.dk” or “130.225.16.15”. Depending on the network topologi and the whereabouts of this process, some prefixes of the first format may also suffice, notably a format like “quercus”. The port must be an integer. By convention, port numbers below 5000 are reserved for system administration purposes and for special, well-known services like e-mail and ftp. On the other hand, do not expect to be able to use more than a 16-bit unsigned value (0 through 65535). The value to use when assigning `inetAddr` must be the four-byte internet address, given as an integer value. E.g. the absolute address “130.225.16.15” is given as the integer 2195787791. The integer must be in the normal byte-order of the platform running the program.

`forceTimeout` is used to provoke the same response within an ongoing operation as would have been the result of a timeout. This makes it possible to exercise timeout control over an operation from within a co-routine different from the one executing that operation. Moreover, it makes it possible to define a timeout limit for the execution of a number of operations, instead of setting timeouts for each of them. `UsageTimeStamp` returns an integer value which indicates when this socket was last used. The value makes sense only when compared to usage time stamps of other sockets in this same process. The purpose is to enable a user of many sockets to close the least recently used connection or similarly when and if the process runs out of system resources (e.g. it experiences a “to many open files” error).

`close` must be called when done with the socket. Every local `idle` executes the `idle` on `basicsocket`. The global error is called whenever a operation-level error is called and did not handle the error. `nonBlockingScope` is explained below.

The `nonBlockingScope` pattern is used for specifying non-blocking communication. This means that operations which cannot begin right away are discontinued. An example is: We try to read from a socket, but no data at all is available to read. If, on the other hand, any irreversible actions have been taken in an operation (e.g. reading a few bytes), it will not be interrupted by the `nonBlockingScope` mechanism. This means it is always safe to interrupt an operation by enclosing it in a `nonBlockingScope`, and then later to retry it. It also means that the granularity of scheduling by means of `nonBlockingScope` is one communication operation; e.g. if the communication partner sends half a block and then takes a break, this process can only execute an `idle` in the mean time, it cannot switch forth and back between several such ongoing transfers. With each `Idle` pattern comes a `Blocking` virtual. This is executed if the current operation is blocking, i.e. if nothing can be done right away and nothing has been done yet. You may extend this virtual to take some action in response to the operation being blocked. If the operation is enclosed in a `nonBlockingScope`, `Blocking` gets executed immediately before the operation is interrupted. If you do not want to interrupt the operation, execute `continue` in a extending of `Blocking`. (If you are not using a `nonBlockingScope`, the operation wil automatically continue when possible)

**nonBlocking-
Scope**

`withPE` and `withIdle` are auxiliary patterns used in implementing the scheduling system.

3.5 The Fragment `binarysocket`

3.5.1 The Patterns of `binarysocket`

The only pattern defined is `BinarySocket`. `BinarySocket` inherits from `BasicSocket`.

`endOfData`, `putInt` and `getInt`

`endOfData` returns true if no data is immediately available for reading. `putInt` and `getInt` are used to transmit a single integer. The integer is transmitted in big-endian format. This makes communication across little- and big-endian machines of integers easy.

`putRep` and `getRep`

`putRep` and `getRep` sends and receives instances of `ExtendedRepstream`. This is a generic container for arbitrary blocks of data, in particular it is possible to put texts and integers into it and read them out again. When receiving data into an `ExtendedRepstream` with `getRep`, the `ExtendedRepstream` will automatically be extended in case the received amount of data exceeds its current capacity.

```

len      header  data
|-----|-----|-----|

```

`putRepObj` and `getRepObj`

`putRepObj` and `getRepObj` are used to send and receive instances of the pattern `RepetitionObject`. The protocol for transmitting `RepetitionObject`s is a little different from the one used with `ExtendedRepstream` objects: there is no header field, and the length field is the first element in the repetition from the `repetitionObject`, i.e. `repetitionObject`s have their length “built-in”.

```

len      data
|-----|-----|

```

Otherwise, it is like the protocol for `ExtendedRepstream` objects.

3.6 The Fragment `streamsocket`

The following describes the operations of `StreamSocket` in order of appearance. `StreamSocket` inherits from `Stream`. `theSocket` is the `BasicSocket` used to transfer the data.

`timeoutValue` is the timeout used on all operation that do not enter a timeout. This includes all the patterns inherited from `stream`.

`sameConnection` checks if the OS level connection wrapped in this `StreamSocket` is the same as the one wrapped in `other`. A `StreamSocket` connection may be closed by `close`. After this point, the `StreamSocket` cannot be used for communication, so you can discard (i.e. forget) it. `StreamSockets` should be closed after use to free up

system resources. `Flush` ensures that all data in internal buffers of the `StreamSocket` actually gets sent. `close` does an automatic flush. `Put`, `get` and `peek` work as with other streams.

`PutText`, `getLine` and `getAtom` work like in other streams. `Eos` returns true if no data can possibly be read from this connection now or ever. On the other hand, it may still happen that the communication partner holds the connection alive but will not write any more data to it. In this case, this process has no chance of guessing that no more data will actually arrive, so `eos` will “spontaneously” change from false to true when the other process actually closes the connection.

`Init`, `forceTimeout`, `usageTimestamp`, `NonBlockingScope`, `leaveNBScope`, `connect`, `error`, `host`, `port`, `inetAddr`, `idle` uses the implementation in `BasicSocket` directly and are described in 3.4.1.

3.7 The Fragment `commpipe`

A pipe must be initialized with `init` before usage. Then giving a reference to its `readEnd` (`writeEnd`) as `enter` parameter to `redirectFromChannel` (`redirectToChannel`) of a not yet started `process` object will attach this pipe to another (not yet created) process. If only one end of the pipe is attached to another process, the current process may read from (write to) the other end of the pipe, when the other process has been created.

3.8 SocketGenerators

3.8.1 The patterns of `socketgenerator`

A `socketGenerator` is a factory from which instances of `streamSocket` and of `binarySocket` can be obtained, in response to active sockets connecting to the `socketGenerator`'s port.

`Bind` must be executed to establish the given port number as an address, to which active sockets may connect. Executing `bind` with `port=0` establishes a randomly chosen port number as an address. The actual port number used may be read from `port`. None of the other operations make sense on an unbound generator.

As usual, when you are done, execute `close` on the `socketGenerator`.

`getPortableAddress` exits a `portableCommunicationAddress` which describes the network identity of this `socketGenerator`. `ForceTimeout` and `usageTimeStamps` work as with the other socket variants, and the considerations concerning `nonBlockingScope` and `leaveNBScope` are as usual.

3.8.2 The patterns of `streamgenerator` and `binarygenerator`

To obtain a `streamSocket` on the next connection requested, execute `getStreamConnection`, and to obtain a `binarySocket`, execute `getBinaryConnection`. Remember to enter a timeout value. When you are done with the created socket, execute `close` on it.

3.9 Error Handling

Throughout `process`, the facilities from the fragment `errorCallback` are used in the handling of errors.

3.9.1 Error Callbacks

An error callback is a virtual pattern which is invoked in response to the occurrence of some error. Whenever an error condition is detected on a socket, a corresponding virtual pattern is instantiated and executed. These patterns are specializations of `errCB`, as declared in `errorCallback`. Such virtual patterns are hereafter denoted error callback patterns. To catch and treat an error, extend the corresponding error callback.

If an error callback is not extended and the corresponding error occurs, an exception is executed and the program terminates. If the error callback is extended, the following holds:

- if `abort` is executed in the extending `dopart`, the operation (but not the program) is aborted. You may execute `leave` within a specialization of `abort`. Do not `leave` an error callback from any other point, as this may put the object or the process into an unstable state. If you `abort` but do not `leave`, the operation aborts, but control flow is like when the operation succeeds; in this case, any exited values are dummy values, reflecting that the operation failed. Do not use them! Actually, do not `abort` without `leave`!
- if `continue` is executed in the extending `dopart`, there will be an attempt to recover and finish the operation after the execution of the error callback terminates. For many types of errors, no general recovery is possible at the operation level. But you could close a couple of files in response to a `resourceError` and then execute `continue`. In case of timeout, you can always choose to take another turn with `continue`.
- if `fatal` is executed in the extending `dopart`, an exception will be executed and the program will be terminated. So the execution of the error callback will not return. This is also the default, but with hierarchical error callbacks, you may need `fatal` to undo a `continue` at a higher level.

In case it happens more than once that an operation from the set `{abort,continue,fatal}` is executed, the one executed as the last takes precedence.

3.9.2 Error Propagation

As mentioned, the error callback patterns are present at three different levels: Concrete error callbacks, operation level error callbacks, and socket level error callbacks.

The concrete error callbacks provide the greatest level of detail: their names indicate the kind of error condition detected. This makes it possible to treat different errors differently.

The operation level error callback is executed whenever an error condition is detected during the execution of that operation. In a extending of this kind of error callback, you can adjust the default action for all the concrete error callbacks in this operation. The single socket level error callback is executed whenever any operation detects any error condition. In a extending of this error callback, you can adjust the default action for all concrete and operation level error callbacks.

The means for adjusting the behaviour is in all cases to execute `abort` (probably `abort(# leave L #)`), `continue`, or `fatal`, and the semantics of these imperatives are the semantics of the concrete error callbacks described in section 3.9.1.

Error callback extendings take precedence like this, in ascending order: concrete level, operation level, socket level. This means that the higher level specifies a default, and the more concrete level may override this default by executing `continue`, `abort`, or `fatal`.

3.9.3 Categories of Errors

At the concrete level of error callbacks, errors are categorized according to classes of operating system level error messages.

The list of names used for concrete error callbacks and a short description of the corresponding class of operating system level error is as follows:

Error callback name	Meaning
<code>accessError</code>	insufficient access rights
<code>addressError</code>	address (i.e. (host,port)) in use or invalid
<code>badMsgError</code>	(EBADMSG, hardly documented in man page)
<code>connBrokenError</code>	connection has become unusable
<code>eosError</code>	unexpected end-of-stream
<code>getHostError</code>	error when getting hostname
<code>internalError</code>	should not happen; please report if it does!
<code>intrError</code>	operation interrupted by signal
<code>refusedError</code>	connection refused by peer
<code>resourceError</code>	too few file descriptors/buffers etc.
<code>timedOut</code>	specified timeout period has expired
<code>timedOutInTransfer</code>	timed out, and some data have been
<code>transferred</code>	
<code>unknownError</code>	OS reports unknown errno (new OS?)
<code>usageError</code>	e.g. you must initialize port before connecting

3.10 Timeout Management

Because most operations may provoke the suspension (de-scheduling) of the current co-routine, any such operation may implicitly prevent this co-routine from making any progress for an indefinite period of time. To give the co-routine the power to do something about this, each of these operations takes a specification of an upper limit (in seconds) to the time elapsed during the execution of that operation.

When such a timeout has been specified for some operation, the scheduler will resume the execution of that operation if it gets the control and the timeout period has expired. This means that lots of activity in the system as a whole may postpone the detection of a timeout somewhat, and - as usual - an infinite loop somewhere could stop everything.

In practical terms, the operation is resumed when and if the timeout period expires, and of course it resumes by executing an error callback. Two different error callbacks may be used to indicate the problem. If no irreversible actions have been taken, the `timedOut` error callback is used. If some irreversible actions have been taken, such as receiving or sending part of a message, the `timedOutInTransfer` error callback is used. This last situation is considerably more grave than the first: Aborting an operation "in-transfer" means breaking the protocol, which again means that any subse-

quent messages received on the same connection will be garbled. Resynchronization is hardly possible unless the data transferred are lines of text or some other format with built-in structural markers. So in this situation, give it another chance, or close the connection.

For `streamSocket` the socket level attribute `timeoutValue` decides the timeout for all operations inherited from `stream`. For `binarySocket` each operation which has timeout control takes the timeout value as its first enter parameter. Likewise with `socketGenerator`. If you forget to specify such a timeout value, the operation will always terminate at once with a timeout error.

4 Addresses

The fragment `commaddress` supports representing addresses of communication ports with which one might like to establish connections. In this setting, more different operating systems and kinds of communication ports are covered than what is actually supported in `BasicSocket` yet. Accordingly, TCP/IP sockets are just one example of a kind of communication port, though a very important one.

Instances of any of these patterns are values, and under normal circumstances their identity will make no difference. This ensures that it makes sense to translate them from BETA objects into simple strings of text and back again, and this eases the migration of such values across networks and other media.

At the most abstract level, `portableCommAddress` models a portable communication address. This specifies the address of a single destination or the address(es) of a group of destinations.

The patterns `portableMultiAddress` and `portablePortAddress` specialize `portableCommAddress` into concrete patterns for the multiple-destination case and one-destination case, respectively.

The pattern `concretePortAddress` and its specializations represent non-portable, protocol specific communication port addresses. Of course, any `concretePortAddress` is portable, being a normal BETA object; but only on some platforms will it be possible to have such a communication port as is specified by the `concretePortAddress`.

`ConcretePortAddresses` are kept in `portableCommAddresses` and selected according to protocol specifications, given as `protocolSpec` objects.

4.1 Specification of Connection Requirements

The pattern `protocolSpec` is used to package a specification of requirements to a communication transfer. This package is given to a `portablePortAddress`, which will then use it to choose an appropriate channel. A specification is built with an instance of `protocolSpec` by setting its `cType` and `rType` attributes. For these, choose from the constant values given in the fragment `commError`.

The `cType` value can be any of the constants `commProtocol_...` and specifies that the chosen channel must be a TCP/UDP/etc. connection or that any kind of connection will do (`commProtocol_dontcare`).

The value of `rType` is any of the constants `commRely_dontcare` (no requirements), `commRely_unreliable` (allow all the below mentioned kinds of malfunction) or `commRely_reliable` (prevent all those malfunctions). Or it is a sum of some of the

constants `commRely_loss` (prevent packet lossage), `commRely_dup` (prevent packet duplication), `commRely_order` (prevent packets from arriving out of order), `commRely_contents` (prevent packets from having corrupt data).

In reality, the last guarantee is enforced by means of checksums or something similar, so it is only very unlikely that a packet with corrupt data will pass unnoticed, not impossible. Moreover, all the other guarantees depend on having packets with trustworthy (header) contents, so not all combinations make sense.

4.2 The Abstract Level

The abstract pattern `portableCommAddress` is used to specify the identity of an abstract communication address. The patterns `portableMultiAddress` and `portablePortAddress` are its non-abstract specializations.

Before usage, initialize any specialization of `portableCommAddress` with `init`.

Any `portableCommAddress` is able to express its value in textual form, by the operation `asText`. This enables simple and safe migration of an instance of any specialization of `portableCommAddress`: Translate it into text, send it across the network, write it into a disk file, or whatever, and then reconstruct it as a BETA object from its text value.

Tell a `portableCommAddress` what properties are required of the communications associated with it by entering a `protocolSpec` object reference. This affects its choice of concrete communication port(s) in subsequent communications.

To reconstruct a `portableCommAddress` from its text representation, give it as `enter` parameter to `portableCommAddressFromText`, and a corresponding object will be exited. The text is expected to have been produced by some instance of a specialization of `portableCommAddress` using its `asText`.

Problems in this process are reported by invoking `parseError`. This terminates the application, unless you extend `parseError` to handle it.

4.3 The Concrete Level

A `portableMultiAddress` specifies a group of communication ports. Start or enhance the group by `inserting` members. Reduce it by `deleting` members.

A `portablePortAddress` specifies the identity of one logical communication destination. A logical destination corresponds to a number of concrete communication ports, represented by instances of specializations of `concretePortAddress`. It is up to the user of these patterns to ensure that the contained set of concrete ports actually “logically belong to the same destination”.

The idea is that if “I” can talk on a channel of type “{A,B}” and “you” can talk on a channel of type “{B,C,D}”, it is up to the underlying framework to discover that in order to establish a connection, “we” must use type “B”.

A `portablePortAddress` can be built by inserting specializations of `concretePortAddress`. Only one concrete address is allowed for each known type - inserting a sec-

ond instance overrides the previously inserted one. With `delete`, any concrete port can be removed again. To retrieve a concrete port (without removing it), use one of the `Get...Port` operations. If this `portablePortAddress` does not contain any concrete port of the requested variety, `NONE` is exited.

`ConcretePortAddress` is an abstract superpattern for specifying the address of a concrete communication port, such as a UNIX stream socket, a Macintosh PPC ToolBox session, a shared memory buffer etc.

Like a `portableCommAddress`, each concrete specialization is able to express its value textually with the operation `asText`, and it is able to characterize its communication protocol with the operation `protocol`. The operation `protName` exits a text which is a short, descriptive name for that protocol, and `conformsTo` answers true/false to the question, whether this kind of connection conforms to the protocol associated with an entered `commProtocol_...` constant.

The pattern `unixAbstractPortAddress` captures similarities between TCP and UDP ports, represented by `tcpPortAddress` and `udpPortAddress`. The `tcpPortAddress` also fits a MacTCP port. The pattern `unixPortAddress` represents an AF_UNIX address family socket, i.e. it appears as a name in some directory, just like a file; `ppcPortAddress` represents a Macintosh PPC ToolBox session; `memPortAddress` corresponds to a shared memory implementation of inter-process communication.

5 Managing a Pool of Connections

A connection pool manages a number of client side communication interfaces (e.g. active sockets), and allows choosing which one of them to use for a communication transfer by means of a `portableCommAddress`. This abstracts away the need to establish connections: whenever a connection as specified is available in the pool, we use it. Otherwise, such a connection will implicitly be established and added to the pool. If this process runs out of resources associated with these connections (e.g. file handles), it is possible to ask the pool to close the least recently used connection.

Concurrency control

The connections are subject to concurrency control, so they must be used in a “take-it, use-it, give-it-back” fashion. This is achieved by the pattern `communication`. The concurrency control is necessary to prevent the situation where two users of the pool both transmit messages to some other party on one given connection, and randomly divide the incoming messages on that connection between them, both believing to have the other party for themselves. Using the pattern `communication`, at most one user of the pool communicates on any given connection at any given point of time.

Binary socket connections

By now, the only variant of connection pool implemented is the `binaryConnectionPool`. Instances of `binaryConnectionPool` are used for managing a number of binary socket connections. Before usage, initialize it. The user of a `binaryConnectionPool` gives a specification of the receiver, the type of connection, the quality of service etc. in a `portableCommAddress` to a (specialization of) the control pattern `communication`. This is used as follows (where `bcPool` is an instance of `binaryConnectionPool`):

```
addr[] -> bcPool.communication
(# (* Extend error callbacks here *)
do
  (* Within this dopart: use 'sock' to communicate *)
  (* Do not bring references to sock outside *)
#);
```

If you want to leave the `dopart` of a specialization of a `communication`, use a construction like `leaving(# do leave L #)` in stead of `leave L`. Otherwise some resources may be rendered inaccessible.

Whenever the pool establishes a new connection, the hook `onNewConnection` of `communication` is executed. In an extending of this hook, a reference to the newly established connection is available, and by assigning a co-routine to `actor`, the connection gets associated with this co-routine. This is used to handle incoming messages to connections in the pool, which are not the immediate response to an outgoing message transmitted in a usage of `communication`: have the co-routine sit around waiting for the incoming messages. To support such things, one must specialize `binaryConnectionPool`.

If the connection delivered as `sock` within a specialization of `communication` is to be taken away from the pool and used outside, execute `removeSock` and bring out a reference to `sock`. If it is known that the connection will not be useful anymore, execute `removeSock` and `sock.close`.

On exceptions, see the description in section 3.9.

The operation `markAsDead` is used to tell the pool that it certainly cannot have a connection like the one entered. If a communication partner closes a connection (or perhaps terminates unexpectedly), and the other end of that connection is in a connection pool, it could happen that this connection is not chosen in any `communication` for some time. If a new connection is created, the operating system may then reuse the local connection identifier (file handle, in case of UNIX sockets), giving a totally different connection, which is then administrated by some new BETA socket object. Now two BETA socket objects will talk to the same OS level connection (file handle), but this means that the first object (in the pool) has silently been “redirected” to a new communication partner. Of course, this leads to strange errors.

So, whenever creating a BETA socket object OUTSIDE a connection pool, please tell it by means of `markAsDead`, that any connections in the pool with the same OS level identifier must have died silently and thus should be removed from the pool. Internally, the connection pool handles this automatically.

Please note that this problem is not specific for connection pools, for the `process` library, or even for BETA programs, for that matter. But it occurs mainly in the presence of complicated and very dynamic communication topologies, which are more likely to appear with connection pools. It would actually be best to carry out similar checks (using `sameConnection`) also when using only simple socket objects in an application.

`removeSomeConnection` will seek through all unused connections in the pool. An unused connection is a connection such that no instance of `communication` in any routine of this process currently refers to it with its `sock` attribute. From this set of unused connections, it chooses the least recently used (as reported by its `usageTimestamp`), closes it, and removes it from the pool. If all connections are currently in use, application specific actions must be taken to free some of them. The callback `noConnectionsRemovable` is executed in this situation. It does not terminate the application by default, so beware of the possible infinite retry loop if `removeSomeConnection` is used in response to `resourceError`, and no connections could actually be removed.

When done with a `connectionPool`, close it to close all of the connections contained within it.

6 The Demo Files

A number of demonstration files are provided in the subdirectory `demo`. They show simple and typical ways to use the process library.

Because of the “process” aspect, and because of the nature of inter-process communication, the demo files come in small groups. For some groups, one program will manipulate others. For other groups, one may start a “server” and some “clients” and then interact with the clients to initiate communication. In the following, the groups are presented one by one.

7.1 pipeline, consumer and producer

Execute `pipeline`, which will then start `producer` and `consumer` in such a way that standard output from `producer` is piped into standard input of `consumer`. The file `items` is read in by `producer` and written to its standard output. `consumer` reads its standard input and writes it to its standard output. The result is, that `items` is written to standard output.

7.2 firstProgram and otherProgram

When executed, `firstProgram` will start `otherProgram` and accept a `StreamSocket` connection from `otherProgram`. Then they exchange a couple of words, and both terminate.

7.3 streamcounterserver and streamcounterclient

Start an instance of `streamcounterserver`. Then start a number of instances of `streamcounterclient`.

7.4 binarycounterserver and binarycounterclient

Start an instance of `binarycounterserver`. Then start a number of instances of `binarycounterclient`.

7.5 xpilotgames

`xpilotgames` demonstrates how to use the `StreamSocket` pattern to connect to the xpilot meta server and send a query about ongoing games.

7.6 repChatClient and repChatServer

This group is used interactively. Start `repChatServer` and then a number of instances of `repChatClient`. Each client will connect to the server, resulting in a star-shaped connection topology. One may interact with each of the clients, and the clients in turn interact with the server.

The fragment `commandCategory` is used to distinguish different types of commands. The command language is very simple: anything starting with the letter “q” is a Quit command, anything starting with an “a” is an Answer command, and anything starting with an “A” is an AnswerWait command. Anything else is a Default command. Enter commands as any piece of text at the prompt, ending with `RETURN`. Please note that leading whitespace is significant.

All commands are immediately forwarded to the server. Then, if the command was a Quit command, the client closes down the connection and terminates. If it was an Answer command, the client notifies the user of that fact by printing a message containing the sequence number of this Answer command. Some time later, the server will return an answer, and the sequence number of the answer makes it possible to match up outgoing requests with incoming answers. In case of an AnswerWait command, the client blocks until the answer from the server arrives. For Default commands, the contents are just echoed at the server.

For each command received, the server echoes the identification number of the client which sent that command and the contents of the command. You may wish to examine the source code in `repChatServer` to see how `nonblockingScope` enables the server to (semi-)simultaneously receive incoming messages, accept connections from new clients, and do other work.

8 Known Bugs and Inconveniences

8.1 General

For `StreamSockets`, reading a line of text with the operation `getLine` or a word with `getAtom` only works correctly when the line/word becomes available to read as a whole. If a non-empty part of the line/word but not all of it can be read, the operation incorrectly detects an error. A possible workaround is to use `get` and collect characters in a normal BETA text object, on which `getLine` and `getAtom` can be used.

If the transmitting side always sends lines/words in one go, the problem is unlikely to show up. In this case, if the purpose is non-critical of course, you could try to ignore the problem.

Outputting operations in `streamSocket`, such as `put`, `flush` and `putLine`, will not detect a buffer full condition before attempting to transmit data. This means that they may block until the operating system has relieved the full buffer of some of its contents. This usually happens quickly, though.

`Eos` on pipes seems to fail on some systems.

Certain operations take as `enter` parameter a timeout value, which does not affect the execution of the operation, because timing out makes no sense - the operation is not “possibly lengthy”. An example is `close` of a `Socket`.

In `portableMultiAddress`, members are deleted by identity, i.e. entering a reference to some `portablePortAddress` in an invocation of the `delete` operation will delete that exact instance, if present. It would make more sense to delete every `portablePortAddress` contained by this `portableMultiAddress`, which specifies the same communication port as the one entered. That is, it would be better if members were deleted by value equality.

`portableMultiAddress` ought to have means for iterating through all its members, such as a `scan` operation. There should also be a way to test for equality and for subset-relations between `portablePortAddresses`, and between `portableMultiAddresses`.

In the fragment `commpool`, in the pattern `communication` in `binaryConnectionPool`, the operation `removeSock` does not remove the connection denoted by `sock` as it should. Workaround: Use `sock[]->markAsDead` wherever `removeSock` should have been used.

The proxy demo is undocumented and probably not quite working

8.2 Windows

Redirecting output through `redirectFromFile` has not yet been implemented on systems running Windows 95/NT. The same limitation exists for reading and writing to a pipe. Using a pipe to connect to external programs has been implemented, though.

`UsageTimeStamp` has not yet been implemented. `Commpool` therefore selects a random socket when choosing a connection to break, not the least recently used.

8.3 Macintosh

Processmanager has no implementation on Macintosh.

`UsageTimeStamp` has not yet been implemented.

9 Interface Description

9.1 commaddress

```

ORIGIN '~beta/basiclib/v1.6/betaenv';

(*
 * COPYRIGHT
 *      Copyright (C) Mjolner Informatics 1994-97
 *      All rights reserved.
 *)

BODY 'private/commaddressbody';

(* CONTENTS
 * =====
 *
 * Defines patterns for representing communication addresses.
 *
 * The most abstract pattern, portableCommAddress, models a
 * portable communication address. This specifies the address
 * of a single destination or the address(es) of a group of
 * destinations.
 *
 * The patterns portableMultiAddress and portablePortAddress
 * specialize portableCommAddress into concrete patterns for
 * the multiple-destination case and one-destination case,
 * respectively.
 *
 * The pattern concretePortAddress and its specializations
 * represent non-portable, protocol specific communication
 * port addresses. These are kept in portableCommAddresses
 * and selected according to protocol specifications, given
 * as protocolSpec objects.
 *
 * As a best-fit addition, there are also some patterns
 * to aid the process of looking up TCP/IP hosts, getting the
 * hostname of this machine, etc.
 *)

--- lib:attributes ---
(* Reliability
 * =====
 *
 * Used to specify the reliability properties
 * required for a transfer (in a protocolSpec).
 * The properties are additive.

```

```

*)

commRely_dontcare:    (# exit 0 #);
commRely_loss:      (# exit 2 #); (* packets are not lost *)
commRely_dup:       (# exit 4 #); (* packets are not duplicated *)
commRely_order:    (# exit 8 #); (* packets arrive
                               * in correct order *)
commRely_contents: (# exit 16 #); (* corrupt data unlikely
                               * (e.g. checksum) *)

commRely_unreliable: (# exit 1 #); (* ensures none of the above *)
commRely_reliable:  (# exit 31 #); (* ensure loss, dup,
                               * order & contents *)

(* Type of connection protocol
 * =====
 *
 * OS level category of connection. An implementation
 * level description of an individual connection
 * managed by a connectionPool. Weird numbers chosen
 * to make data containing these constants recognizable
 * in a raw communication dump.
 *)

commProtocol_dontcare:  (# exit 0 #);
commProtocol_tcp:      (# exit 72301 #); (* TCP/IP *)
commProtocol_udp:     (# exit 72302 #); (* UDP/IP *)
commProtocol_unix:    (# exit 72303 #); (* UNIX domain
                               * (socket as file) *)
commProtocol_ppc:     (# exit 72304 #); (* Mac PPC ToolBox *)
commProtocol_mem:     (# exit 72305 #); (* Shared memory buffer *)

(* Mnemonic names of the protocols *)
commProtName_tcp:      (# exit 'TCP' #);
commProtName_udp:     (# exit 'UDP' #);
commProtName_unix:    (# exit 'UNIX' #);
commProtName_ppc:     (# exit 'PPC' #);
commProtName_mem:     (# exit 'MEM' #);

(* Specification of connection requirements
 * =====
 *
 * Used to package spec. of requirements to a communication
 * transfer, and then given to a portablePortAddress, which
 * will use it when choosing an appropriate channel.
 *)
protocolSpec:
  (#
    cType: @integer; (* one of 'commProtocol_.*'
                      * dontcare is default *)
    rType: @integer; (* one of 'commRely_.*'
                      * dontcare is default *)
    (* bandwidth/r-rr-rra/etc *)
    enter (cType, rType)
    exit cType
  #);

(* Portable communication address
 * =====
 *
 * Specifies identity of an abstract communication address.
 * This pattern is abstract, and no instances of it are

```

```

* expected to exist. The patterns portableMultiAddress and
* portablePortAddress are non-abstract specializations.
*
* Any portableCommAddress is able to express its value
* in textual form, by 'asText'.
*
* Tell a portableCommAddress what properties are required
* of the communications associated with it by entering
* a protocolSpec object. This affects its choice of
* concrete communication port(s) in subsequent
* communications.
*)
portableCommAddress:
  (#
    init:< Object;
    asText: @asTextPattern;

    (* private *)
    asTextPattern:< (# t: ^text do INNER exit t[] #);
    enterSpec: @...;
    private: @...;
    enter enterSpec
  #);

(* Portable communication address constructor
* =====
*
* Function. Takes a text value, which is expected to have
* been produced by some instance X of a specialization of
* portableCommAddress using its 'asText'. Returns an object
* with the same value as X.
*
* Problems are reported by invoking 'parseError'. The
* application will then terminate with an exception,
* unless you furtherbind parseError to leave it.
*)
portableCommAddressFromText:
  (#
    parseError:<
      (# msg: ^text;
      enter msg[]
      ...
      #);
    txt: ^text;
    addr: ^portableCommAddress;
    <<SLOT portableCommAddressFromTextLib:attributes>>;
    enter txt[]
    ...
    exit addr[]
  #);

(* Portable multicast address
* =====
*
* Specifies identities of the members of a group of
* communication destinations.
*
* The group can be built from scratch or enhanced
* by 'insert'ing members. It can be reduced by
* 'delete'ing members.
*)
portableMultiAddress: portableCommAddress

```

```

(#
  init::< (# ... #);

  insert:
    (# addr: ^portablePortAddress;
     enter addr[]
     ...
     #);

  delete:
    (# addr: ^portablePortAddress;
     enter addr[]
     ...
     #);

  (* private *)
  asTextPattern::< (# ... #);
  private2: @...;
#);

(* Portable communication port address
 * =====
 *
 * Specifies identity of one logical communication destination.
 * A logical destination corresponds to a number of concrete
 * communication ports, represented by instances of
 * specializations of concretePortAddress.
 *
 * A portablePortAddress can be built from scratch by
 * by 'insert'ing such instances. Only one concrete address
 * is allowed for each known type - inserting a second instance
 * overrides the previously inserted one.
 *)
portablePortAddress: portableCommAddress
  (#
    insert:
      (# addr: ^concretePortAddress;
       addrHasUnknownType:< exception;
       enter addr[]
       ...
       #);
    delete:
      (# prot: @integer; (* one of 'commProtocol_.*' *)
       addrHasUnknownType:< exception;
       enter prot
       ...
       #);
    getTcpPort:
      (# addr: ^tcpPortAddress;
       ...
       exit addr[] (* NONE if not present *)
       #);
    getUdpPort:
      (# addr: ^udpPortAddress;
       ...
       exit addr[] (* NONE if not present *)
       #);
    getUnixPort:
      (# addr: ^unixPortAddress;
       ...
       exit addr[] (* NONE if not present *)
       #);
  )

```

```

    getPpcPort:
      (# addr: ^ppcPortAddress;
      ...
      exit addr[] (* NONE if not present *)
      #);
    getMemPort:
      (# addr: ^memPortAddress;
      ...
      exit addr[] (* NONE if not present *)
      #);

    (* private *)
    asTextPattern::< (# ... #);
    private2: @...;
  #);

(* Concrete communication port address
* =====
*
* Abstract superpattern for specifying the address
* of a concrete communication port, such as a UN*X
* stream socket, a Mac PPC ToolBox session, a shared
* memory buffer etc.
*
* Is able to express its value textually with 'asText',
* and to characterize its communication protocol
* with 'commType'.
*)
concretePortAddress:
  (#
    asText: @asTextPattern;
    asTextPattern:< (# t: ^text do INNER exit t[] #);

    protocol:< integerValue; (* one of 'commProtocol_.*' *)
    protName:< (# t: ^text do &text[] -> t[]; INNER exit t[] #);
    conformsTo: BooleanValue
      (# p: @integer;
      enter p
      ...
      #);
    private: @...;
  #);

(* Unix communication port address types
* =====
*
* The pattern unixAbstractPortAddress captures similarities
* between TCP and UDP ports, represented by
* tcpPortAddress and udpPortAddress.
*
* The pattern unixPortAddress represents an AF_UNIX address
* family socket, i.e. it appears as a name in some directory,
* just like a file.
*
* NB: The tcpPortAddress also fits a MacTCP port.
*)
unixAbstractPortAddress: concretePortAddress
  (#
    inetAddr: @integer;
    portNo: @integer;
    asTextPattern::< (# ... #);
  #);

```

```

tcpPortAddress: unixAbstractPortAddress
  (#
    protocol::< (# do commProtocol_tcp -> value #);
    protName::< (# do commProtName_tcp -> t #);
  #);

udpPortAddress: unixAbstractPortAddress
  (#
    protocol::< (# do commProtocol_udp -> value #);
    protName::< (# do commProtName_udp -> t #);
  #);

unixPortAddress: concretePortAddress
  (#
    asTextPattern::< (# ... #);
    pathName: @text;
    protocol::< (# do commProtocol_unix -> value #);
    protName::< (# do commProtName_unix -> t #);
  #);

(* Mac communication port address
 * =====
 *
 * Represents a PPC ToolBox session.
 *)
ppcPortAddress: concretePortAddress
  (#
    host: @text;
    portNo: @integer;
    sessionId: @integer;
    asTextPattern::< (# ... #);
    protocol::< (# do commProtocol_ppc -> value #);
    protName::< (# do commProtName_ppc -> t #);
  #);

(* Shared memory buffer port address
 * =====
 *
 * Corresponding communication support NOT IMPLEMENTED.
 * Could be very fast, perhaps for communicating within
 * one process, using the same source code as for remote
 * communication.
 *)
memPortAddress: concretePortAddress
  (#
    bufferID: @integer; (* !!! This may have to change *)
    asTextPattern::< (# ... #);
    protocol::< (# do commProtocol_mem -> value #);
    protName::< (# do commProtName_mem -> t #);
  #);

(* IPv4 Miscellaneous address conversions *)

(* Look up the IPv4 address of a given host. *)
gethostbyname:
  (#
    notfound:< Exception;
    name: ^Text;
    inadr: @Integer;
  enter name[]
  ...

```

```

    exit inadr
  #);

(* Find the name and IPv4 address of this host. *)
thisHost:
  (# name: ^Text;
   inadr: @Integer;
   err: @Integer; (* Private *)
   ...
  exit (name[], inadr)
  #);

```

9.2 errorcallback

```

ORIGIN '~beta/basiclib/v1.6/betaenv';

(*
 * COPYRIGHT
 *      Copyright (C) Mjolner Informatics 1995-97
 *      All rights reserved.
 *)

BODY 'private/errorcallbackbody';

--- lib:attributes ---

errCB_initialValue: (# exit -1 #);
errCB_abortProgram: (# exit 0 #);
errCB_abortOperation: (# exit 1 #);
errCB_continueOperation: (# exit 2 #);

errCB: IntegerValue
  (# abort: (# ... #);
   continue: (# ... #);
   fatal: (# ... #);
   addMsg: (# t: ^text enter t[] ... #);
   exceptionType:< exception;
   cleanup: ^object;
   private: @...;
  enter cleanup[]
  ...
  #);

hiErrCB: IntegerObject
  (# abort: (# ... #);
   continue: (# ... #);
   fatal: (# ... #);
   cleanup: ^object;
  enter cleanup[]
  do INNER
  #);

```


9.3 basicsocket

```

ORIGIN '~beta/basiclib/v1.6/basicssystemenv';

(*
 * COPYRIGHT
 * Copyright (C) Mjolner Informatics 1995-97
 * All rights reserved.
 *)

INCLUDE 'errorcallback';
INCLUDE 'commaddress';

--- systemlib:attributes ---
(* Used for timeouts *)
waitForever: (# exit -1 #);

(* Used to make it checkable whether something is uninitialized *)
assignGuard: (# assigned: @Boolean do true -> assigned #);

(* The number 127.0.0.1 by convention is 'this host' *)
localHost_IP_number: (# exit 2130706433 #);

BasicSocket:
  (# <<SLOT socketlib:attributes>>;

  (* OPERATIONS
   * =====
   *)

  (* do 'this' and 'other' wrap the same OS level connection? *)
  sameConnection: booleanValue
    (# other: ^basicSocket;
     enter other[]
     ...
    #);

  (* construct portable address for this connection *)
  getPortableAddress:
    (# addr: ^portablePortAddress;
     ...
     exit addr[]
    #);

  (* Initiator of socket communication.
   * Pass 'host' and 'port' to 'connect' to connect
   * to a passive socket to establish communication.
   *)
  connect: open
    (# accessError:< loErrCB(# do INNER #);
     resourceError:< loErrCB(# do INNER #);
     addressError:< loErrCB(# do INNER #);
     refusedError:< loErrCB(# do INNER #);
     intrError:< loErrCB(# do INNER #);
     getHostError:< loErrCB(# do INNER #);
     aHost: ^Text;
     aPort: @Integer;
     enter (aHost[],aPort)
     ...
    #);

```

```

(* provoke a timeout error in the current operation *)
forceTimeout:< (# ... #);

(* return timestamp of latest operation on this socket *)
usageTimestamp:< integerValue
  (# ... #);

(* return true iff no data is
 * immediately available for reading
 *)
endOfDataPattern:
  (# error:< hiErrCB (* operation level error callback *)
    (#
      do INNER;
      (if value=errCB_initialValue then
        (value,cleanup[])->this(basicSocket).error->value;
        if);
      #);
    loErrCB: errCB (* superpattern for
      * concrete error callbacks *)
    (#
      do INNER;
      (if value=errCB_initialValue then
        (value,cleanup[])->error->value;
        if);
      #);
    connBrokenError:< loErrCB(# do INNER #);
    internalError:< loErrCB(# do INNER #);
    unknownError:< loErrCB(# do INNER #);
    value: @boolean;
    ...
    exit value
    #);

close:< withIdle(# ... #);

(* CALLBACKS
 * =====
 *)

(* every local 'idle' executes this global one *)
idle:< Object;

(* socket level error callback *)
error:< hiErrCB(# do INNER #);

(* EXPLICIT SCHEDULING
 * =====
 *)

(* NB: don't 'leave' a 'nonBlockingScope'. Use 'leaveNBscope'. *)
nonBlockingScope: (# ... #);
leaveNBscope: (# ... #);

(* ATTRIBUTES
 * =====
 *)

host: @assignGuard(# t: @text; enter t exit t #);
port: @assignGuard(# rep: @integer enter rep exit rep #);
inetAddr: @assignGuard(# rep: @integer enter rep exit rep #);

```

```

(* AUXILIARY PATTERNS
 * =====
 *)

withPE:
  (# error:< hiErrCB (* operation level error callback *)
    (#
      do INNER;
      (if value=errCB_initialValue then
        (value,cleanup[])->this(basicSocket).error->value;
      if);
    #);
  loErrCB: errCB (* superpattern for
                  * concrete error callbacks *)
    (#
      do INNER;
      (if value=errCB_initialValue then
        (value,cleanup[])->error->value;
      if);
    #);
  timedOut:< loErrCB(# do INNER #);
  timedOutInTransfer:< loErrCB(# do INNER #);
  internalError:< loErrCB(# do INNER #);
  connBrokenError:< loErrCB(# do INNER #);
  usageError:< loErrCB(# do INNER #);
  unknownError:< loErrCB(# do INNER #);
  resourceError:< loErrCB(# do INNER #);
  badMsgError:< loErrCB(# do INNER #);
  timeout: @integer;
  enter timeout
  do INNER
  #);

withIdle: withPE
  (# idle:< (# do INNER; this(basicSocket).idle #);
    blocking:<(# continue: (# do true->doContinue #);
      doContinue: @boolean;
    do INNER;
      (if not doContinue then leaveNBScope if);
    idle;
    #);
  do INNER
  #);

open: withIdle(# ... #);
init:< (# ... #);

private: @...;
#);

```

9.4 binarysocket

```
ORIGIN 'basicsocket';
```

```

(*
 * COPYRIGHT
 *      Copyright (C) Mjolner Informatics 1995-97

```

```

*      All rights reserved.
*)

BODY 'private/binarysocketbody';
INCLUDE '~beta/sysutils/v1.6/RepetitionObject';
INCLUDE 'repstream/extendedRepstream';

--- systemlib: attributes ---
BinarySocket: basicSocket
  (# <SLOT binarysocketlib: attributes>>;

  (* send an integer *)
putIntPattern: withIdle
  (# i: @integer;
   enter i
   ...
  #);

  (* receive an integer *)
getIntPattern: withIdle
  (# i: @integer;
   ...
   exit i
  #);

  (* send contents of an ExtendedRepstream *)
putRepPattern: repIO
  (#
   enter header
   ...
  #);

  (* receive contents to an ExtendedRepstream *)
getRepPattern: repIO
  (#
   ...
   exit header
  #);

  (* send contents from RepetitionObject *)
putRepObjPattern: repObjIO
  (#
   ...
  #);

  (* receive contents to RepetitionObject.
   * Furtherbinding maxLongs can be used to limit the size of
   * packets being received, which is useful in particular when
   * the sender cannot be trusted.
   *)
getRepObjPattern: repObjIO
  (# maxLongs:< IntegerValue
   (# do MaxInt div 4 -> value; INNER #);
   MaxlongsExceeded:< Object;
   ...
  #);

repIO: withIdle
  (* Read/write a block to/from 'rep',
   * returning/using 'header'. The length of the block is
   * stored in/retrived from 'rep.end'.
   *)

```

```

    (# rep: ^ExtendedRepstream;
      header: @integer;
      enter rep[]
      do INNER
      #);

repObjIO: withIdle
  (#
    (* Read/write a block to/from 'rep'.
     * The length of the block is stored
     * in/retrived from 'rep.end'.
     *)
    rep: ^RepetitionObject;
    enter rep[]
    do INNER
    #);

endOfData: @endOfDataPattern;
putInt: @putIntPattern;
getInt: @getIntPattern;
putRep: @putRepPattern;
getRep: @getRepPattern;
putRepObj: @putRepObjPattern;
getRepObj: @getRepObjPattern;

binpriv: @...;
#)

```

9.5 streamsocket

```

ORIGIN '~beta/basiclib/v1.6/basicssystemenv';

(*
 * COPYRIGHT
 * Copyright (C) Mjolner Informatics 1995-97
 * All rights reserved.
 *)

INCLUDE 'basicsocket';
BODY 'private/streamsocketbody';

--- systemlib:attributes ---
StreamSocket: Stream
  (# <<SLOT streamsocketlib:attributes>>;

   theSocket: @basicSocket; (* The socket
                             * communication goes through
                             *)

   (* basics *)
   timeoutValue:<
     (* Length in seconds.
      * All operations that do not enter a timeout
      * themselves uses this timeout.
      *)
     integerValue(# do waitForever->value; INNER #);

   (* operations *)

```

```

sameConnection: booleanValue
  (* do 'this' and 'other' wrap
   * the same OS level connection?
   *)
  (# other: ^StreamSocket;
   enter other[]
   ...
  #);

close:< theSocket.withPE
  (#
   ...
  #);

flush: theSocket.withIdle
  (#
   ...
  #);

put::
  (# Idle:< (# do INNER #);
   Blocking:< (# do INNER #);
   ...
  #);

puttext::
  (# Idle:< (# do INNER #);
   Blocking:< (# do INNER #);
   ...
  #);

get::
  (# theIdle: @theSocket.withIdle
   (# connBrokenError::
    (# do errCB_abortOperation -> value;
     this(Stream).EOSerror;
    #)
   #);
   Idle:< (# do INNER #);
   Blocking:< (# do INNER #);
   ...
  #);

peek::
  (# theIdle: @theSocket.withIdle
   (# connBrokenError::
    (# do errCB_abortOperation -> value;
     this(Stream).EOSerror;
    #)
   #);
   Idle:< (# do INNER #);
   Blocking:< (# do INNER #);
   ...
  #);

getline::
  (# priv: @...;
   Idle:< (# do INNER #);
   Blocking:< (# do INNER #);
   timedOut: @Boolean;
   do priv;
  #);

```

```

getAtom::
  (# ch: @Char;
   Idle:< (# do INNER #);
   Blocking:< (# do INNER #);
   ...
  #);

eos::
  (# priv: @...
   do priv;
  #);

getPos::
  (#
   do -1 -> value;
  #);

setPos::
  (#
   do this(Stream).otherError;
  #);

init:< (# do theSocket.init; INNER #);
forceTimeout:< (# do theSocket.forceTimeout #);
usageTimestamp:< integerValue
  (# do theSocket.usageTimestamp -> value #);

(* nonBlockingScope support *)
(* Note: don't 'leave' a 'nonBlockingScope'.
 * Use 'leaveNBScope'
 *)
nonBlockingScope: theSocket.nonBlockingScope(# do INNER #);
leaveNBScope: theSocket.nonBlockingScope(# do INNER #);

connect: theSocket.connect(# do INNER #);

Idle:< Object; (* every local 'Idle' executes this global one *)

(* socket level error callback *)
error:< hiErrCB(# do INNER #);

(* attributes *)
host: (# enter theSocket.host exit theSocket.host #);
port: (# enter theSocket.port exit theSocket.port #);
inetAddr: (# enter theSocket.inetAddr exit theSocket.inetAddr #);

(* private *)
private: @...;
#); (* StreamSocket *)

```

9.6 socketgenerator

```

ORIGIN '~beta/basiclib/v1.6/basicssystemenv';
INCLUDE 'basicsocket';

```

```
(*
```

```

* COPYRIGHT
*      Copyright (C) Mjolner Informatics 1995-97
*      All rights reserved.
*)

BODY 'private/socketgenbody';

--- systemlib:attributes ---

SocketGenerator:
  (# <<SLOT socketgeneratorlib:attributes>>;

  (* OPERATIONS
  * =====
  *)

  (* Setting 'port'=0 and executing 'bind' gives you
  * a SocketGenerator that accepts connections on a
  * randomly chosen portnumber, which may be found in 'port'.
  *)
  bind: withIdleAndPE
    (#
    enter port
    ...
    #);

  (* construct portable address for this generator *)
  getPortableAddress:
    (# addr: ^portablePortAddress;
    ...
    exit addr[]
    #);

  (* De-register the bind *)
  close: withIdleAndPE
    (#
    ...
    #);

  (* provoke a timeout error in the current operation *)
  forceTimeout: @
    (#
    ...
    #);

  (* return timestamp of latest operation on this generator *)
  usageTimestamp: @integerValue
    (#
    ...
    #);

  (* CALLBACKS
  * =====
  *)

  (* every local 'idle' executes this global one *)
  idle:< object;

  (* socket level error callback *)
  error:< hiErrCB(# do INNER #);

  (* EXPLICIT SCHEDULING

```



```

* =====
*)

(* NB: don't 'leave' a 'nonBlockingScope'.
* Use 'leaveNBScope'.
*)
nonBlockingScope: (# ... #);
leaveNBScope: (# ... #);

(* ATTRIBUTES
* =====
*)

port: @assignGuard(# rep: @integer enter rep exit rep #);

(* AUXILIARY PATTERNS
* =====
*)

withIdleAndPE:
  (# error:< hiErrCB (* operation level error callback *)
    (#
      do INNER;
      (if errCB_initialValue=value then
        (value,cleanup[])->
          this(socketGenerator).error->value;
      if);
    #);
  loErrCB: errCB (* superpattern for
    * concrete error callbacks
    *)

    (#
      do INNER;
      (if errCB_initialValue=value then
        (value,cleanup[])->error->value;
      if);
    #);
  usageError:< loErrCB(# do INNER #);
  resourceError:< loErrCB(# do INNER #);
  accessError:< loErrCB(# do INNER #);
  addressError:< loErrCB(# do INNER #);
  connBrokenError:< loErrCB(# do INNER #);
  intrError:< loErrCB(# do INNER #);
  internalError:< loErrCB(# do INNER #);
  unknownError:< loErrCB(# do INNER #);
  timedOut:< loErrCB(# do INNER #);
  Idle:< (# do INNER; this(socketGenerator).Idle #);
  Blocking:<(# continue: (# do true->doContinue #);
    doContinue: @boolean;
  do INNER;
    (if not doContinue then leaveNBScope if);
  Idle;
  #);
  do INNER
  #);

(* PRIVATE
* =====
*)

private: @...;
#);

```

9.7 binarygenerator

```

ORIGIN 'socketgenerator';
INCLUDE 'binarysocket';
BODY 'private/binarygenbody';

-- socketgeneratorlib:attributes --
(* accept a connection and return a binarySocket on it *)
getBinaryConnection: withIdleAndPE
  (# sockType:< BinarySocket;
   sock: ^sockType;
   timeout: @integer;
   enter timeout
   ...
   exit sock[]
  #);

```

9.8 streamgenerator

```

ORIGIN 'socketgenerator';
INCLUDE 'streamsocket';
BODY 'private/streamgenbody';

-- socketgeneratorlib:attributes --
(* accept a connection and return a streamSocket on it *)
getStreamConnection: withIdleAndPE
  (# sockType:< StreamSocket;
   sock: ^sockType;
   timeout: @integer;
   enter timeout
   ...
   exit sock[]
  #);

```

9.9 commpool

```

ORIGIN 'binarysocket';

(*
 * COPYRIGHT
 *      Copyright (C) Mjolner Informatics 1995-97
 *      All rights reserved.
 *)

BODY 'private/commpoolbody';
INCLUDE 'commaddress';
INCLUDE '~/beta/containers/v1.6/list';

--- systemlib:attributes ---

BinaryConnectionPool:

```

```

(#
  <<SLOT binaryconnectionpoollib:attributes>>;

  (* TYPES
   * =====
   *)

  socketType:< BinarySocket;

  (* OPERATIONS
   * =====
   *)

  init:< (# ... #);

  communication:
    (#
      (* OPERATIONS
       * =====
       *)

      (* remove sock from this pool *)
      removeSock:
        (#
          ...
          #);

      (* NB: always wrap leave/restart out of
       * this(communication) in a specialization
       * of 'leaving' *)
      leaving: (# ... #);

      (* CALLBACKS
       * =====
       *)

      onNewConnection:<
        (* executed when a new connection has been created *)
        (# sock: ^socketType; (* The new connection *)
          context: ^object; (* NB: Should've been private *)
          actor: ^|system; (* process to associate with sock *)
          enter (sock[],context[])
          do INNER
          exit actor[]
          #);

      error:< hiErrCB (* operation level error callback *)
        (#
          do INNER;
          (if errCB_initialValue=value then
            (value,cleanup[])->
              this(BinaryConnectionPool).error->value;
            if);
          #);

      concrErrCB: hiErrCB (* superpattern for
                           * concrete error callbacks *)
        (#
          do INNER;
          (if errCB_initialValue=value then
            (value,cleanup[])->error->value;
            if);
        #);
    #);

```

```

    #);

    addrHasUnknownType:< exception; (* Considered fatal,
                                     * for now *)
    internalError:< concrErrCB(# do INNER #);
    unknownError:< concrErrCB(# do INNER #);
    accessError:< concrErrCB(# do INNER #);
    resourceError:< concrErrCB(# do INNER #);
    addressError:< concrErrCB(# do INNER #);
    refusedError:< concrErrCB(# do INNER #);
    intrError:< concrErrCB(# do INNER #);
    getHostError:< concrErrCB(# do INNER #);

    (* ATTRIBUTES
     * =====
     *)

    addr: ^portableCommAddress;
    sock: ^socketType;

    (* PRIVATE
     * =====
     *)

    priv: @...;

    enter addr[]
    ...
    #);

markAsDead:
    (# sock: ^socketType;
    enter sock[]
    ...
    #);

removeSomeConnection:
    (* Removes least recently used currently unused connection *)
    (# noConnectionsRemovable:< object;
    ...
    #);

close:< (# ... #);

    (* CALLBACKS
     * =====
     *)

    error:< hiErrCB(# do INNER #);

    (* PRIVATE
     * =====
     *)

    private: @...;
#);

```

9.10 systemcomm

```

ORIGIN '~beta/basiclib/v1.6/basicssystemenv';

(*
 * COPYRIGHT
 *   Copyright (C) Mjolner Informatics 1994-97
 *   All rights reserved.
 *)

INCLUDE 'streamgenerator';
INCLUDE 'binarygenerator';

```

9.11 processmanager

```

ORIGIN '~beta/basiclib/v1.6/basicssystemenv';

(*
 * COPYRIGHT
 *   Copyright (C) Mjolner Informatics, 1992-97
 *   All rights reserved.
 *)

BODY 'private/processmanagerbody';
INCLUDE '~beta/basiclib/v1.6/file';

--- systemlib:attributes ---
BetaEnvStop: (# T: ^Text; I: @Integer;
               enter (I,T[])
               do (I,T[]) -> Stop;
               #);

Process:
  (* Notice, this(Process) can only be executed once.
   *
   * Two program executions of the same Process,
   * can be executed by instantiating and executing two different BETA
   * objects from the same Process.
   *)
  (#
    <<SLLOT ProcessLib:attributes>>;

    name: ^Text;
    init:< (# enter name[] ... #);

    argType:
      (# argument: @Text;
        putArg:
          (# t: ^Text;
            enter t[]
            ...
            #);
        append: @putArg;
        scanArguments: (* calls INNER for each argument *)
          (# current: @Text;
            ...

```

```

        #);
    #);
argument: @argType; (* arguments to this(Process) *)

(* operations *)

start: (* starts this(Process)'s program execution *)
    (# error:< ProcessManagerException;
      twoCurrent:< ProcessManagerException;
      ...
    #);

stop: (* stops this(Process)'s program execution *)
    (# error:< ProcessManagerException;
      ...
    #);

awaitStopped: (* Returns when THIS(Process) stops *)
    (# error:< ProcessManagerException;
      ...
    #);

stillRunning: (* Returns true if
                  * THIS(Process) is still running
                  *)
    (# error:< ProcessManagerException;
      value: @Boolean;
      ...
      exit value
    #);

(* input/output redirection *)

connectToProcess: (* connect output of this(process)
                    * to toProcess's input
                    * In Unix shell terms:
                    *   this(Process) | toProcess
                    *)
    (# error:< ProcessManagerException;
      toProcess: ^Process;
      enter toProcess[]
      ...
    #);

connectInPipe: (* connect output of fromProcess
                  * to input of this(process)
                  * In Unix shell terms:
                  *   fromProcess | this(Process)
                  *)
    (# error:< ProcessManagerException;
      fromProcess: ^Process;
      enter fromProcess[]
      ...
    #);

redirectFromFile: (* redirect input to this(process)
                    * from inputFile
                    * In Unix shell terms:
                    *   this(Process) < inputFile
                    *)
    (# error:< ProcessManagerException;
      inputFile: ^File;

```

```

    enter inputFile[]
    ...
    #);

redirectToFile: (* redirect output of this(process)
                  * to outputFile
                  * In Unix shell terms:
                  *   this(Process) > outputFile
                  *)
    (# error:< ProcessManagerException;
     outputFile: ^File;
     enter outputFile[]
     ...
     #);

redirectFromChannel: (* redirect input to this(process)
                       * from inputChannel
                       *)
    (# error:< ProcessManagerException;
     inputChannel: ^Stream;
     enter inputChannel[]
     ...
     #);

redirectToChannel: (* redirect output of this(process)
                      * to outputChannel
                      *)
    (# error:< ProcessManagerException;
     outputChannel: ^Stream;
     enter outputChannel[]
     ...
     #);

(* Callbacks: called when the proper action has occurred *)

onStart:< (# do INNER #);
onStop:< (# do INNER #);

doDebug: @Boolean;
private: @...;
#);

```

```

ProcessManagerException: Exception
    (# message: ^Text;
     enter message[]
     ...
     #);

```

9.12 commpipe

```
ORIGIN '~beta/basiclib/v1.6/basicssystemenv';
```

```

(*
 * COPYRIGHT
 *   Copyright (C) Mjolner Informatics 1995-97
 *   All rights reserved.
 *)

```

```

MDBODY default 'private/commpipe_unix'
      ppcmac   'private/commpipe_mac'
      nti      'private/commpipe_nt';

INCLUDE 'private/sysFdStream';

--- systemlib:attributes ---
propagateException: (# msg: ^Text enter msg[] do INNER #);

pipe:
  (# <<SLOT pipelib:attributes>>;

    (* OPERATIONS *)
    init:<
      (# error:< propagateException
        (# do INNER; msg -> pipeError #);
      ...
      #);

    close:< (# ... #);

    pipeException: Exception
      (#
        enter msg
        do (if not msg.empty then msg.newline if);
          INNER;
      #);

    pipeError:< PipeException;

    (* ATTRIBUTES *)
    readEnd: ^fdStream;
    writeEnd: ^fdStream;

    private: @...;
  #);

```


References

- [Knudsen 94] J. L. Knudsen, M. Löfgren, O. L. Madsen, B. Magnusson (eds.): *Object-Oriented Environments – The Mjølner Approach*, Prentice Hall, 1994, ISBN 0-13-009291-6.
- [Madsen 93] O. L. Madsen, B. Møller-Pedersen, K. Nygaard: *Object-Oriented Programming in the BETA Programming Language*, Addison-Wesley, 1993, ISBN 0-201-62430-3
- [MIA 90-1] Mjølner Informatics: *The Mjølner System: – Overview*, Mjølner Informatics Report MIA 90-1.
- [MIA 90-2] Mjølner Informatics: *The Mjølner System: BETA Compiler Reference Manual* Mjølner Informatics Report MIA 90-2.
- [MIA 91-20] Mjølner Informatics: *The Mjølner System – Persistent Store*, Mjølner_Informatics Report MIA 91-20.
- [MIA 94-24] Mjølner Informatics: *The Mjølner System – The Mjølner System Tutorial*, Mjølner_Informatics Report MIA 94-24.
- [MIA 93-25] Mjølner Informatics: *The Mjølner System – Distribution* Mjølner_Informatics Report MIA 94-25.
- [MIA 94-26] Mjølner Informatics: *The Mjølner System – BETA Language Introduction* Mjølner_Informatics Report MIA 94-26.

Index

The entries in the index with *italic* pagenumbers are the identifiers defined in the public interface of the libraries:

The minor level entries refer to identifiers defined local to the identifier of the major level entry. For those index entries referring to patterns with super- or subpatterns within the library, these patterns are specified in special sections of the minor level index for that identifier.

Entries with plain pagenumbers refer to the text of this manual.

A

abort · 29, 30
activeBinarySocket · 7
activeStreamSocket · 7
addMsg · 29
addr · 26
addresses · 15
argType · 41
argument · 41
assigned · 30
assignGuard · 30
 assigned · 30
AssignGuard · 8
asText · 26, 28
asTextPattern · 26, 27, 28, 29
awaitStopped · 42
awaitStopped · 4

B

basicSocket
 subpatterns:
 BinarySocket · 33
BasicSocket · 30
 booleanValue
 subpatterns:
 sameConnection · 30
close · 31
 superpattern:
 withIdle · 44
connect · 30
 superpattern:
 open · 44
endOfDataPattern · 31
error · 31
 superpattern:
 hiErrCB · 44
forceTimeout · 31
getPortableAddress · 30
hiErrCB
 subpatterns:

 error · 31
host · 31
idle · 31
inetAddr · 32
init · 32
integerValue
 subpatterns:
 usageTimestamp · 31
leaveNBSScope · 31
nonBlockingScope · 31
open · 32
 subpatterns:
 connect · 30
 superpattern:
 withIdle · 44
port · 31
private · 32
sameConnection · 30
 superpattern:
 booleanValue · 44
usageTimestamp · 31
 superpattern:
 integerValue · 44
withIdle · 32
 subpatterns:
 close · 31
 open · 32
 superpattern:
 withPE · 44
withPE · 32
 subpatterns:
 withIdle · 32
BetaEnvStop · 41
 I · 41
 T · 41
Binary socket · 8
binaryConnectionPool · 18
BinaryConnectionPool · 39
 close · 40
 communication · 39
 error · 40
 superpattern:
 hiErrCB · 44
hiErrCB
 subpatterns:
 error · 40

- init · 39
- markAsDead · 40
- private · 40
- removeSomeConnection · 40
- socketType · 39
- BinarySocket · 33
 - binpriv · 34
 - endOfData · 34
 - getInt · 34
 - getIntPattern · 33
 - superpattern:*
 - withIdle · 44
 - getRep · 34
 - getRepObj · 34
 - getRepObjPattern · 33
 - superpattern:*
 - repObjIO · 44
 - getRepPattern · 33
 - superpattern:*
 - repIO · 44
 - putInt · 34
 - putIntPattern · 33
 - superpattern:*
 - withIdle · 44
 - putRep · 34
 - putRepObj · 34
 - putRepObjPattern · 33
 - superpattern:*
 - repObjIO · 44
 - putRepPattern · 33
 - superpattern:*
 - repIO · 44
 - repIO · 33
 - subpatterns:*
 - getRepPattern · 33
 - putRepPattern · 33
 - superpattern:*
 - withIdle · 44
 - repObjIO · 33
 - subpatterns:*
 - getRepObjPattern · 33
 - putRepObjPattern · 33
 - superpattern:*
 - withIdle · 44
 - superpattern:*
 - basicSocket · 33
- withIdle
 - subpatterns:*
 - getIntPattern · 33
 - putIntPattern · 33
 - repIO · 33
 - repObjIO · 33

- bind · 36
- binpriv · 34
- bufferID · 29

C

- Categories of Errors · 13
- cleanup · 29, 30
- close · 31, 34, 36, 40, 43
- commAddress · 15
- commProtName_mem · 25
- commProtName_ppc · 25
- commProtName_tcp · 25
- commProtName_udp · 25
- commProtName_unix · 25
- commProtocol_dontcare · 25
- commProtocol_mem · 25

- commProtocol_ppc · 25
- commProtocol_tcp · 25
- commProtocol_udp · 25
- commProtocol_unix · 25
- commRely_contents · 25
- commRely_dontcare · 24
- commRely_dup · 24
- commRely_loss · 24
- commRely_order · 24
- commRely_reliable · 25
- commRely_unreliable · 25
- communication · 39
- concretePortAddress · 27
 - asText · 28
 - asTextPattern · 28
 - BooleanValue
 - subpatterns:*
 - conformsTo · 28
 - conformsTo · 28
 - superpattern:*
 - BooleanValue · 44
 - private · 28
 - protName · 28
 - protocol · 28
 - subpatterns:*
 - memPortAddress · 29
 - ppcPortAddress · 28
 - unixAbstractPortAddress · 28
 - unixPortAddress · 28
- ConcretePortAddress · 17
- concurrency control · 18
- conformsTo · 28
- connect · 30, 36
- connectInPipe · 42
- connectToProcess · 42
- continue · 29, 30
- co-routines · 7
- cType · 25

D

- delete · 26, 27
- demo · 20
- doDebug · 43

E

- endOfData · 34
- endOfDataPattern · 31
- enterSpec · 26
- eos · 35
- err · 29
- errCB · 29
 - abort · 29
 - addMsg · 29
 - cleanup · 29
 - continue · 29
 - exceptionType · 29
 - fatal · 29
 - private · 29
 - superpattern:*
 - IntegerValue · 29
- errCB_abortOperation · 29
- errCB_abortProgram · 29
- errCB_continueOperation · 29
- errCB_initialValue · 29
- error · 31, 36, 37, 40
- error · 12

errorCallback · 12
 Exception
 subpatterns:
 ProcessManagerException · 43
 exceptionType · 29

F

fatal · 29, 30
 flush · 34
 forceTimeout · 31, 35, 37

G

get · 35
 getAtom · 35
 getAtom · 11
 getBinaryConnection · 38
 sock · 38
 sockType · 38
 superpattern:
 withIdleAndPE · 38
 timeout · 38
 gethostbyname · 29
 inadr · 29
 name · 29
 notfound · 29
 getInt · 34
 getIntPattern · 33
 getline · 35
 getLine · 11
 getMemPort · 27
 getPortableAddress · 30, 36
 getPos · 35
 getPpcPort · 27
 getRep · 34
 getRepObj · 34
 getRepObjPattern · 33
 getRepPattern · 33
 getStreamConnection · 38
 sock · 38
 sockType · 38
 superpattern:
 withIdleAndPE · 38
 timeout · 38
 getTcpPort · 27
 getUdpPort · 27
 getUnixPort · 27

H

hiErrCB · 30
 abort · 30
 cleanup · 30
 continue · 30
 fatal · 30
 superpattern:
 IntegerObject · 30
 host · 28, 31, 36
 hostMachine · 5

I

I · 41

idle · 31, 37
 Idle · 36
 inadr · 29
 inetAddr · 28, 32, 36
 init · 26, 32, 35, 39, 41, 43
 insert · 26, 27
 IntegerObject
 subpatterns:
 hiErrCB · 30
 IntegerValue
 subpatterns:
 errCB · 29
 Inter-process communication · 6

L

leaveNBSCOPE · 31, 36, 37
 leaveNBSCOPE · 11
 localhost_IP_number · 30

M

MacTCP · 17
 markAsDead · 40
 markAsDead · 19
 memPortAddress · 29
 asTextPattern · 29
 bufferID · 29
 protName · 29
 protocol · 29
 superpattern:
 concretePortAddress · 29
 message · 43
 monitor · 7
 msg · 43

N

name · 29, 41
 nonBlockingSCOPE · 31, 36, 37
 notfound · 29

O

onStart · 43
 onStart · 4
 onStop · 43
 open · 32

P

parseError · 26
 pathName · 28
 peek · 35
 pipe · 43
 close · 43
 Exception
 subpatterns:
 pipeException · 44
 init · 43
 pipeError · 44
 pipeException · 44
 superpattern:

- Exception · 44
- private · 44
- readEnd · 44
- writeEnd · 44
- pipe · 11
- pipeError · 44
- pipeException · 44
- Pipes · 6
- port · 31, 36, 37
- portableCommAddress · 25
 - asText · 26
 - asTextPattern · 26
 - enterSpec · 26
 - init · 26
 - private · 26
 - subpatterns:
 - portableMultiAddress · 26
 - portablePortAddress · 27
- portableCommAddress · 16
- portableCommAddressFromText · 26
 - addr · 26
 - parseError · 26
 - txt · 26
- portableCommunicationAddress · 11
- portableMultiAddress · 26
 - asTextPattern · 26
 - delete · 26
 - init · 26
 - insert · 26
 - private2 · 26
 - superpattern:
 - portableCommAddress · 26
- portableMultiAddress · 16
- portablePortAddress · 27
 - asTextPattern · 27
 - delete · 27
 - getMemPort · 27
 - getPpcPort · 27
 - getTcpPort · 27
 - getUdpPort · 27
 - getUnixPort · 27
 - insert · 27
 - private2 · 27
 - superpattern:
 - portableCommAddress · 27
- portablePortAddress · 16
- portNo · 28
- ppcPortAddress · 28
 - asTextPattern · 28
 - host · 28
 - portNo · 28
 - protName · 28
 - protocol · 28
 - sessionId · 28
 - superpattern:
 - concretePortAddress · 28
- private · 26, 28, 29, 32, 36, 38, 40, 43, 44
- private2 · 26, 27
- process · 4
- Process · 41
 - argType · 41
 - argument · 41
 - awaitStopped · 42
 - connectInPipe · 42
 - connectToProcess · 42
 - doDebug · 43
 - init · 41
 - name · 41
 - onStart · 43
 - onStop · 43
 - private · 43

- redirectFromChannel · 43
- redirectFromFile · 42
- redirectToChannel · 43
- redirectToFile · 42
 - start · 41
 - stillRunning · 42
 - stop · 42
- ProcessManagerException · 43
 - message · 43
 - superpattern:
 - Exception · 43
- propagateException · 43
 - msg · 43
- protName · 28, 29
- protocol · 28, 29
- protocolSpec · 25
 - cType · 25
 - rType · 25
- protocolSpec · 15
- put · 34
- putInt · 34
- putIntPattern · 33
- putRep · 34
- putRepObj · 34
- putRepObjPattern · 33
- putRepPattern · 33
- puttext · 35
- PutText · 11

R

- readEnd · 44
- redirectFromChannel · 43
- redirectFromChannel · 4
- redirectFromFile · 42
- redirectFromFile · 4
- redirectToChannel · 43
- redirectToChannel · 4
- redirectToFile · 42
- redirectToFile · 4
- removeSomeConnection · 40
- removeSomeConnection · 19
- repIO · 33
- repObjIO · 33
- rType · 25

S

- sameConnection · 30, 34
- semaphore · 7
- sessionId · 28
- setPos · 35
- sock · 38
- socketGenerator · 7
- SocketGenerator · 36
 - bind · 36
 - superpattern:
 - withIdleAndPE · 44
 - close · 36
 - superpattern:
 - withIdleAndPE · 44
 - error · 37
 - superpattern:
 - hiErrCB · 44
 - forceTimeout · 37
 - getPortableAddress · 36
 - hiErrCB

- subpatterns:*
 - error · 37
- idle · 37
- leaveNBSCOPE · 37
- nonBlockingSCOPE · 37
- port · 37
- private · 38
- usageTimestamp · 37
- withIdleAndPE · 37
 - subpatterns:*
 - bind · 36
 - close · 36
- Sockets · 6
- socketType · 39
- sockType · 38
- start · 41
- start · 4
- stillRunning · 42
- stillRunning · 4
- stop · 42
- Stream
 - subpatterns:*
 - StreamSocket · 34
- Stream socket · 8
- StreamSocket · 34
 - booleanValue
 - subpatterns:*
 - sameConnection · 34
 - close · 34
 - connect · 36
 - eos · 35
 - error · 36
 - superpattern:*
 - hiErrCB · 44
 - flush · 34
 - forceTimeout · 35
 - get · 35
 - getAtom · 35
 - getline · 35
 - getPos · 35
 - hiErrCB
 - subpatterns:*
 - error · 36
 - host · 36
 - Idle · 36
 - inetAddr · 36
 - init · 35
 - integerValue
 - subpatterns:*
 - timeoutValue · 34
 - usageTimestamp · 35
 - leaveNBSCOPE · 36
 - nonBlockingSCOPE · 36
 - peek · 35
 - port · 36
 - private · 36
 - put · 34
 - puttext · 35
 - sameConnection · 34
 - superpattern:*
 - booleanValue · 44
 - setPos · 35
 - superpattern:*
 - Stream · 34
 - theSocket · 34
 - timeoutValue · 34
 - superpattern:*
 - integerValue · 44
 - usageTimestamp · 35
 - superpattern:*
 - integerValue · 44

systemEnv · 7

T

T · 41

TCP · 17

TCP/IP · 15

tcpPortAddress · 28

- protName · 28
- protocol · 28
- superpattern:*
 - unixAbstractPortAddress · 28

tcpPortAddress · 17

theSocket · 34

thisHost · 29

- err · 29
- inadr · 29
- name · 29

timeout · 38

timeout · 13

timeoutValue · 34

txt · 26

U

UDP · 17

udpPortAddress · 28

- protName · 28
- protocol · 28
- superpattern:*
 - unixAbstractPortAddress · 28

udpPortAddress · 17

unixAbstractPortAddress · 28

- asTextPattern · 28
- inetAddr · 28
- portNo · 28
- subpatterns:*
 - tcpPortAddress · 28
 - udpPortAddress · 28
- superpattern:*
 - concretePortAddress · 28

unixAbstractPortAddress · 17

unixPortAddress · 28

- asTextPattern · 28
- pathName · 28
- protName · 28
- protocol · 28
- superpattern:*
 - concretePortAddress · 28

usageTimestamp · 31, 35, 37

usageTimeStamP · 11

W

waitForever · 30

WaitForever · 8

withIdle · 32

withIdleAndPE · 37

- subpatterns:*
 - getBinaryConnection · 38
 - getStreamConnection · 38

withPE · 32

writeEnd · 44