Process Libraries – Reference Manual

Mjølner Informatics Report
MIA 94–29
February 2002

Table of Contents

Table of Contents

1 Introduction

 This document describes the version 1.6 of the process library in the Mjølner 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.

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

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.

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;
```

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

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.

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.

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.

ConnectErrToProcess, redirectErrToFile and redirectErrToChannel work like connectToProcess, redirectToFile and redirectToChannel with the exception that they use the standard error from the process, rather than the standard output. It is possible to connect both the standard output and the standard error to the same stream.

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.

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.

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.

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 initalization 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 following 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.

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, it is also possible to use this kind of socket to transfer arbitrary binary data, as no conversion or translation is performed. This may be used to connect to existing services with a known binary protocol.

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 responsebility 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.

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.

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)

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

3.5 The Fragment binarysocket

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

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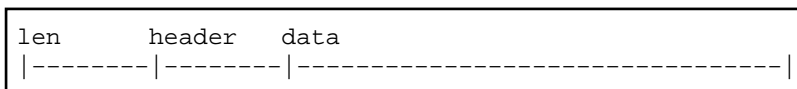
putRepObj and getRepObj are used to send and receive instances of the pattern RepetitionObject. The protocol for transmitting RepetitionObjects is a little different from the one used with ExtendedRepstream objects: there is no header field, and the length field is the first element in the repetition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

```
len       data
|--------|------------------------------|
```
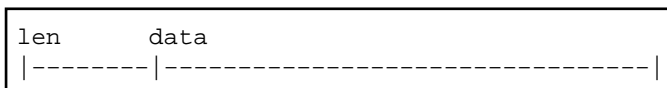
Otherwise, it is like the protocol for ExtendedRepstream objects.

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.

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

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

3.8.1 The patterns of socketgenerator

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 usageTimeStamp 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.

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

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.

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 a 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.

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 co–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.

6.1 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 it standard input and writes it to its standard output. the result is, that items is written to standard output.

6.2 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.

6.3 streamcounterserver and streamcounterclient

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

6.4 binarycounterserver and binarycounterclient

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

6.5 xpilotgames

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

6.6 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.

7 Known Bugs and Inconveniences

7.1 General

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 lenghty**. 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 whereever removeSock should have been used.

The proxy demo is undocumented and probably not quite working

7.2 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.

7.3 Macintosh

Processmanager has no implementation on Macintosh.

UsageTimeStamp has not yet been implemented.

8.1 Basicsocket Interface

```
ORIGIN '~beta/basiclib/basicsystemenv';
LIB_DEF 'processbasic' '../lib';

(*
 * 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.
      * If you need to control the local port number,
      * use firstLocalPort and lastLocalPort. These are then
      * tried one at a time starting with first and ending
      * with last. None of them can be zero.
      *)
     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 #);
          firstLocalPort:<IntegerValue;
          lastLocalPort:<IntegerValue;
          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 socket completely. Any further operations are
 * disallowed and the other end gets EOS if it tries *)
close: withIdle(# ... #);

(* Close socket partially. closeRead makes further reads
 * at this end of the socket and further writes at
 * the other end fail with EOS. *)
closeRead: (# ... #);

(* Close socket partially. closeWrite makes further writes
 * at this end of the socket and further reads at
 * the other end fail with EOS. *)
closeWrite: (# ... #);

(* 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: @...;
  #)
```

8.2 Binarygenerator Interface

```
ORIGIN  'socketgenerator';
LIB_DEF 'processbinarygen' '../lib';

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[]
  #)
```

8.3 Binarysocket Interface

```
ORIGIN 'basicsocket';
LIB_DEF 'processbinary' '../lib';

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

BODY 'private/binarysocketbody';
INCLUDE '~beta/sysutils/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: @...;
#)
```

8.4 Commaddress Interface

```
ORIGIN '~beta/basiclib/betaenv';
LIB_DEF 'processaddress' '../lib';

(*
 * 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 proporties
 * required for a transfer (in a protocolSpec).
 * The proporties 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 proporties 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
  #);

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

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

8.5 Commpipe Interface

```
ORIGIN '~beta/basiclib/basicsystemenv';
LIB_DEF 'processpipe' '../lib';

(*
 * 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: @...;
#)
```

8.6 Commpool Interface

```
ORIGIN 'binarysocket';
LIB_DEF 'processpool' '../lib';

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

BODY 'private/commpoolbody';
INCLUDE 'commaddress';
INCLUDE '~beta/containers/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: @...;
  #)
```

8.7 Errorcallback Interface

```
ORIGIN '~beta/basiclib/betaenv';
LIB_DEF 'processerrcb' '../lib/';

(*
 * 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
   #)
```

8.8 Processmanager Interface

```
ORIGIN '~beta/basiclib/basicsystemenv';
LIB_DEF 'processmanager' '../lib';

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

BODY 'private/processmanagerbody';
INCLUDE '~beta/basiclib/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.
   *)
  (#
     <<SLOT 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[]
    ...
    #);

  connectErrToProcess:  (* connect stdout of this(process)
                         * to toProcess's input
                         * In Unix shell terms:
                         *    this(Process) |2 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[]
```

```
      ...
      #);

  redirectErrToFile:  (* redirect stderr output of this(process)
                       * to outputFile
                       * In Unix shell terms:
                       *    this(Process) >2 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[]
     ...
     #);

  redirectErrToChannel:  (* redirect stderr 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[]
   ...
   #)
```

8.9 Socketgenerator Interface

```
ORIGIN '~beta/basiclib/basicsystemenv';
LIB_DEF 'processsockgen' '../lib';

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

INCLUDE 'basicsocket';
BODY 'private/socketgenbody';

--- systemlib:attributes ---

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

      (* OPERATIONS
       * ==========
       *)

      (* Setting 'port'=0 and executing 'bind' gives you
       * a SocketGeneratorthat accepts connections on a
       * randomly chosen portnumer, 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: @...;
  #)
```

8.10 Streamgenerator Interface

```
ORIGIN  'socketgenerator';
LIB_DEF 'processstreamgen' '../lib';

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[]
  #)
```

8.11 Streamsocket Interface

```
ORIGIN '~beta/basiclib/basicsystemenv';
LIB_DEF 'processstream' '../lib';

(*
 * 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 *)
       (# error::
              (#
              do (if value=errCB_initialValue then
                     (* Error not handled yet *)
                     (value,cleanup[])->this(StreamSocket).error->value;
                     (if value=errCB_initialValue then
                         this(StreamSocket).otherError;
                         (* If otherError did not terminate the
                          * program, let it continue here as well
                          *)
                         errCB_continueOperation->value;
                     if)
                  if)
              #)
        #);

     (* 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[]
       ...
       #);

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

     close: theSocket.withPE
       (* Close socket completely. Any further operations are
        * disallowed and the other end gets EOS if it tries *)
       (# ... #);
```

```
closeRead: theSocket.closeRead
(* Close socket partially. closeRead makes further reads
 * at this end of the socket and further writes at
 * the other end fail with EOS. *)
  (# #);

closeWrite: theSocket.closeWrite
(* Close socket partially. closeWrite makes further writes
 * at this end of the socket and further reads at
 * the other end fail with EOS. *)
  (# #);

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: @...;
#)
```

8.12 Systemcomm Interface

```
ORIGIN '~beta/basiclib/basicsystemenv';

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

INCLUDE 'streamgenerator';
INCLUDE 'binarygenerator';
```

Index

The entries in the alphabetic index consists of selected words and symbols from the body files of this manual – these are in **bold** font – as well as the identifiers defined in the public interfaces of the libraries – set in regular font.
In the manual, the entries, which can be found in the index are typeset like this. This can help localizing the identifier, when the link from the index if followed – especially in the case where the browser does not scroll the line to the top, e.g. because there is less than a page of text left.
In the small table of letters and symbols below, each entry links directly to the section of the index containing entries starting with the corresponding letter or symbol.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

A

abort [2]
Abstract Level
addMsg
addr
Addresses

addresses
argType
argument
assigned
AssignGuard

assignGuard
assigned
asText [2]
asTextPattern [2] [3] [4] [5] [6] [7] [8]
awaitStopped [2]

B

BasicSocket
    close
    closeRead
    closeWrite
    connect
    endOfDataPattern
    error
    forceTimeout
    getPortableAddress
    host
    idle
    inetAddr
    init
    leaveNBScope
    nonBlockingScope
    open
    port
    private
    sameConnection

    usageTimestamp
    withIdle
    withPE
BetaEnvStop
    I
    T
binaryConnectionPool
BinaryConnectionPool
    close
    communication
    error
    init
    markAsDead
    private
    removeSomeConnection
    socketType
binarycounterserver and binarycounterclient
BinarySocket [2]
    binpriv

    endOfData
    getInt
    getIntPattern
    getRep
    getRepObj
    getRepObjPattern
    getRepPattern
    putInt
    putIntPattern
    putRep
    putRepObj
    putRepObjPattern
    putRepPattern
    repIO
    repObjIO
bind
binpriv
bufferID

C

Child Processes
cleanup [2]
close [2] [3] [4] [5]
closeRead [2]

commProtocol_udp
commProtocol_unix
commRely_contents
commRely_dontcare

asTextPattern
conformsTo
private
protName

T

U

W