



Summary of Foresight's Intrinsic Functions

Intrinsic routines are built-in functions and procedures that may be called within Mini-Spec language statements. The Mini-Spec language is Foresight's VHDL-like text based programming language. This means such functions are available to Mini-Spec process blocks in Data Flow Diagrams (DFDs) as well as the statements of State Transition Diagrams (STDs) such as actions on transitions and actions on entering or exiting a state.

The information contained in this summary is taken from the Mini-Spec Reference Manual delivered with Foresight and is intended to provide the essence of the kind of supporting library routines provided. Please refer to that document for the full definition of the routines including details such as:

- Data types supported
- Restrictions on usage

For purposes of this summary, simply be aware that:

- Some intrinsics are overloaded. That is, they support multiple data types for call arguments and return values with appropriate data type matching being enforced. An example of overloading is the `put()` function which outputs (prints) the current value of a variable. It supports virtually all the built-in scalar types, such as integer, real, complex, and boolean.
- Some intrinsics support polymorphism in that they support varying numbers of arguments. Again using the `put()` function as an example, it will write to standard output by default or to a specific file if a file type argument is supplied in addition to a variable containing a data value.
- Some parameters have default values. That is, if not specified explicitly in the Mini-Spec statement, a value is provided for the call. An example is a timeout argument, where if not supplied, then the default is that there is no timeout.
- The names of arguments are included here to provide an idea of the meaning of the parameterization of the calls. The names are descriptive of the functionality.
- Optional arguments are indicated by surroundings `<>`, such as `<name>`, where *name* is the name of the argument.
- Some of the functionality documented here are statements rather than function or procedure calls. Those without parentheses are Mini-Spec statements.
- Access to most of these routines is also provided graphically via the predefined blocks of the Foresight library.
- All times values are in seconds and of the real data type.



1 Two Kinds of Mini-Spec Modules

Regular or plain Mini-Specs are process blocks in a Data Flow Diagram whose functionality is specified with the Mini-Spec language. Mini-Spec blocks are free to use all the Mini-Spec language routines to affect the timing of the process block they implement.

Callable Mini-Specs are root level specification-items that serve as utility routines for other spec-items. That is, they are spec-items in the top level of the table of contents of a subsystem. They are not used as blocks within a diagram. A Callable Mini-Spec has the same structure as a regular Mini-Spec except that it cannot have an initialization section. The statements in a Callable Mini-Spec are restricted to not affect the timing or resource usage and as such may not call some of the intrinsic routines described here. The basic purpose of a Callable Mini-Spec is to capture an algorithm for use by process blocks.

2 Intrinsic Summary

The routines have been categorized into functional groups to organize and help remember the type of support provided.

2.1 Process Control

Many of the calls here control the signals output on process block ports (flows), which in turn may be connected to other process blocks. Such flows are used to control the execution status of the downstream blocks. Basically they are changed between being eligible for execution or not. Such execution flows are known as *prompts* in Foresight. The difference between several prompts is whether the *initialization* section of the spec-item is executed or not when the module is again eligible for execution.

2.1.1 *disable*(outflow <, after>)

Activates a disable prompt at the current time, or *after* the specified time elapses.

2.1.2 *disconnect outflow* <after t>

Computes effective value of a flow with multiple sources and possibly conflicting values. Modeled after VHDL behavior to emulate tri-state behavior.

2.1.3 *enable*(outflow <, after>)

Activates an enable prompt at the current time, or *after* the specified time elapses.

2.1.4 *getTime*()

Returns the current simulation time.

2.1.5 *hasChanged*(inputFlow)

Returns whether the *inputFlow* value changed from the previous execution. This is equivalent to the following Co-Design syntax:



inputFlow'Event

2.1.6 *resume*(outflow <, after>)

Activates a resume prompt at the current time, or *after* the specified time elapses.

2.1.7 *signal*(outflow <, after>)

Activates a signal event at the current time, or *after* the specified time elapses.

2.1.8 *suspend*(outflow <, after>)

Activates a suspend prompt at the current time, or *after* the specified time elapses.

2.1.9 *trigger*(outflow <, after >)

Activates a trigger prompt at the current time, or *after* the specified time elapses.

2.1.10 *wait statement*

Supports unconditional and conditional wait in the following forms:

wait for time -- Unconditional wait

wait on inputFlow <for time>-- Unconditional wait on *inputFlow*

wait on inputFlow <until BoolExpression> --Conditional wait on *inputFlow*

Suspends execution of the spec-item for the given amount of time or until the next event on the *inputFlow*.

2.2 *Resource Manipulation*

Basic definitions of resources are made graphically via the blocks of the Foresight Library. However, one has control to change the attributes of a resource during simulation in addition to the basic allocating and releasing actions for a resource. In the calls described, the phrase *proc* is included in the name when a *process* resource is the target; otherwise the calls refer to a basic resource. Resources are referred to by name, which is a string.

2.2.1 *increaseRes*(name, amount)

Increases the total capacity of a resource by the specified *amount*. Once the resource has been increased, any processes currently waiting for the resource with a request that can be satisfied by the increased capacity are given the requested resource.

2.2.2 *isProcAvailable*(name, minunits, priority)

Queries a process resource to discover whether the corresponding requestProc would obtain the requested resource, or block the calling process.



2.2.3 *reduceProcRes*(name,amount,preempt)

Reduces the number of pool units of a process resource. If the boolean preempt flag is false, this call will succeed only if the resource is currently available.

2.2.4 *reduceRes*(restoken)

Reduces the total capacity of a resource by the amount that was acquired with the token. Note that a process must request and be granted a resource before it can reduce the capacity of the resource.

2.2.5 *releaseRes*(restoken)

Releases a resource designated by *restoken* for a previous request call.

2.2.6 *requestProc* (name, amount <, minunits, maxunits, priority, rtnOnPreempt, robinTime>)

Requests the named process resource for the specified *amount* of effort. The arguments are:

name = string; --name of resource

amount = real; --amount requested

minunits = real; --minimum number of units requested

maxunits = real; --maximum number of units requested

priority = integer; --request priority

rtnOnPreempt = boolean; --return flag

robinTime = real; --timeslice; return flag

2.2.7 *requestRes*(name, amount <,timeout>)

Requests a particular *amount* of the named resource with an optional *timeout* parameter, which defaults to no timeout. A token is returned for subsequent call use.

2.2.8 *requestUDR*(name,x)

Requests a named User Defined Resource (UDR) with the user defined parameter structure *x*. UDRs are Data Flow Diagrams with a single entry point and possibly multiple exit points. The blocks in between define the resource usage accounting and policy to be applied for the resource type.

2.3 Type Conversions

These functions provide conversions between basic data types.

2.3.1 *float*(integer)

Converts an integer to a real number.



2.3.2 *imag*(complex)

Returns the imaginary part of *complex* as a real number.

2.3.3 *int*(real)

Converts a real number to an integer by truncating the fractional part.

2.3.4 *integer_to_signed*(integer_to_convert, NumOfBits)

The integer is converted to 2's *complement*.

2.3.5 *integer_to_unsigned*(integer_to_convert, size)

Converts integer to binary (size, bits)

2.3.6 *real*(complex)

Returns the real part of *complex* as a real number.

2.3.7 *signed_to_integer*(bitvector_to_convert)

Converts 2's complement bit vector to integer.

2.3.8 *unsigned_to_integer*(bitvector_to_convert)

Converts 2's complement bit vector to integer.

2.4 Basic Mathematics (e.g. Trigonometric, Logarithm, Square Root)

In this group, the routines are not overloaded. Rather, the leading character of the name indicates the data type of the argument and return values. 'C' is for complex data, while 'R' is for real data. Real trigonometric function arguments are expressed in radians.

2.4.1 *ccos*(complex)

Returns the trigonometric function *cosine* for the *complex* argument.

2.4.2 *cexp*(complex)

Returns the exponential function for the *complex* argument.

2.4.3 *cln*(complex)

Returns the natural logarithm for the *complex* argument.

2.4.4 *clog*(complex)

Returns the base 10 logarithm for the *complex* argument.



2.4.5 csin(complex)

Returns the trigonometric function *sine* for the *complex* argument.

2.4.6 csqrt(complex)

Returns the square root for the *complex* argument.

2.4.7 ctan(complex)

Returns the trigonometric function *tangent* for the *complex* argument.

2.4.8 ratan(real)

Returns the trigonometric function *arc tangent* for the *real* argument.

2.4.9 rcos(real)

Returns the trigonometric function *cosine* for the *real* argument.

2.4.10 rexp(real)

Returns the exponential for the *real* argument.

2.4.11 rln(real)

Returns the natural logarithm exponential for the *real* argument.

2.4.12 rlog(real)

Returns the base 10 logarithm for the *real* argument.

2.4.13 rsin(real)

Returns the trigonometric function *sine* for the *real* argument.

2.4.14 rsqrt(real)

Returns the square root for the *real* argument.

2.4.15 rtan(real)

Returns the trigonometric function *tangent* for the *real* argument.

2.5 Data Manipulation (Shift, Rotate)

These operations are available as functions (as shown here) or as infix operators, e.g. using the ROL operator

Result := bitvector ROL numOfBits;



2.5.1 ROL(bitvector, numOfBits)

Rotates bit vector left by the specified number of bits, wrapping to the end of the list.

2.5.2 ROR(bitvector, numOfBits);

Rotates bit vector right by the specified number of bits, wrapping to the beginning of the list.

2.5.3 SLA(bitVector, numOfBits)

Shift Left Algebraic applies to signed data where the first bit is held and the bits are shifted the designated number of bits. Values are not wrapped and empty bits are sign extended.

2.5.4 SLL(bitVector, numOfBits)

Shift Left Logical applies to unsigned shifts where all bits are shifted the number of bits designated. Values are not wrapped and empty bits are padded with 0's.

2.5.5 SRA(bitVector, numOfBits)

Shift Right Algebraic applies to signed data where the first bit is held and the bits are shifted the designated number of bits. Values are not wrapped and empty bits are sign extended.

2.5.6 SRL(bitVector, numOfBits)

Shift Right Logical applies to unsigned data where all values are shifted the number of bits designated. Values are not wrapped and empty bits are padded with 0's.

2.6 Iteration Manipulation

Iterations are variable length lists within Foresight, which may have zero to N elements on them during a simulation. Once populated, elements are accessed by array indexing notation with indices of 1 to the current length.

2.6.1 deleteNth(iter,n)

Deletes the *n*th element from the iteration and repacks the iteration to be of length *n*-1.

2.6.2 insertAfter(iter,v,n)

Inserts the element *v* into the iteration after the *n*th position.

2.6.3 sizeof(iter)

Returns the current length of the iteration.

2.7 File Manipulation

These routines provide for file operations such as creation, reading, and writing to both files as well as standard input and output. On Windows based platforms standard input and output are directed to the starting DOS window. As in 'C', a handle is returned when a file is



opened or created and represented here by the *filevar* argument. Most calls return a status variable that may be checked to see if the operation succeeded.

File formats supported include text and keyed. Keyed files are like one-dimensional arrays where a string key is used to index into the array elements. The data elements within a keyed file may be of different data types.

2.7.1 File Handling

2.7.1.1 open(filevar,name)

Opens an existing file and updates the *filevar* argument.

2.7.1.2 create(filevar,name)

Creates a new file, opens it, and updates the *filevar* argument.

2.7.1.3 rename(oldname,newname)

Changes the name of a file.

2.7.1.4 close(filevar)

Flushes and closes an open file.

2.7.1.5 closeAll()

Flushes and closes all open files.

2.7.1.6 delete(name)

Delete a file.

2.7.2 File Reading

These routines are overloaded in that they support multiple data types for the *value* argument.

2.7.2.1 get(<filevar,> value<, width>)

Reads an element of the *value* data type. The optional *width* argument controls the number of columns to read.

2.7.2.2 getline(filevar,str)

Reads an entire line into a string, positioning the file pointer at the beginning of the next line.

2.7.2.3 getKey(filevar,keyStr,value)

Reads from *filevar* into the *value* using the *keyStr* as the index into the keyed file.



2.7.2.4 getKeyLine(filevar,keyStr,value)

Synonym for getKey().

2.7.3 File Writing

These routines are overloaded in that they support multiple data types for the *value* argument.

2.7.3.1 newline(<filevar>)

Writes a new line character to the specified file.

2.7.3.2 put(<filevar,>value<,width>)

Writes the value to the specified file.

2.7.3.3 putline(<filevar,>formatStr<,value>)

Similar to 'C' printf() where the *formatStr* contains placeholders for variable substitution (as %v) and a variable number of *values* may be included.

2.7.3.4 putKey(filevar,keyStr,value)

Writes to *filevar* from the *value* using the *keyStr* as the index into the keyed file.

2.7.3.5 putKeyLine

Synonym for putKey().

2.7.4 File Positioning

2.7.4.1 reset(filevar)

Resets the file position for the *filevar* to the beginning of the file.

2.7.4.2 skipline(filevar)

For reading from a file, this skips to the next line as defined by a newline character.

2.7.4.3 eol(filevar)

Returns whether the input stream is at the end of a line.

2.7.4.4 eof(filevar)

Returns whether the input stream is at the end of file.

2.8 Random Number Generators

These generators return real numbers. Each generator call returns a single value.



2.8.1 exponential(mean)

Generates random numbers exponentially distributed with the given *mean*.

2.8.2 gaussian(mean,stdDev)

Generates random numbers normally distributed with the given mean and standard deviation.

2.8.3 setRandSeq(seqNum)

Sets the seed of the random number generators. By default a seed of zero is used.

2.8.4 uniform(low,high)

Generates random numbers uniformly distributed between the *low* and *high* limits.

2.9 Semaphore Handling

Semaphores come in three flavors in Foresight: Binary, Counting, and Mutual Exclusion. Semaphores are given names (a string) and creation calls return an ID by which the entity is referenced in later calls.

2.9.1 createBsem(name, initiallyFull)

Creates a binary semaphore with the given initial state of full or empty per the boolean *initiallyFull* argument.

2.9.2 createCSem(name, initialValue)

Creates a counting semaphore with the given *initialValue*.

2.9.3 createMSem(name)

Creates a mutual exclusion semaphore, which are always created with a status of full.

2.9.4 deleteSem(semid)

Deletes the specified semaphore.

2.9.5 getSem(name)

Gets the *semId* associated with the named semaphore. This may block the execution of the caller if the semaphore is not available.

2.9.6 semGive(semid)

Gives up the semaphore. If another process has already taken the semaphore (and the process is now blocked, waiting for the semaphore), that process will become ready to run. If the semaphore is already full, that is, it has been given but not taken, this call has no effect.



2.9.7 *semIsEmpty*(semid <, waitflag>)

Determines whether the semaphore is empty. If it is called with boolean *waitflag* true, it will block until the semaphore is empty to permit mutual synchronization. If it is called with *waitflag* false, it returns immediately with the status of the semaphore.

2.9.8 *semTake*(semid <, timeout>)

Attempts to take the semaphore until the optional *timeout* expires. An unspecified timeout is defaulted to no timeout and waits indefinitely.

2.9.9 *semValue*(semid)

Returns the current value of the semaphore. The main purpose of this call is to count semaphores and return the current count. This allows one process to determine whether it is “falling behind” another.

2.10 *Deprecated Routines*

Foresight supports two Mini-Spec syntaxes known by various names in the documentation:

- *Foresight* or *V4* syntax is the original Mini-Spec language syntax.
- *VHDL* or *V5* syntax is the syntax introduced with Foresight Co-Design and is the current default syntax for new models.

The routines presented here are from V4. Their functionality has been replaced by new syntax in Foresight Co-Design. Users are strongly encouraged to use V5 syntax as it is the actively supported syntax.

2.10.1 *delay*(time_delay)

Suspends operation of the calling spec-item for the given time interval. Upon expiration of the time interval, the process will read any continuous input flows and resume operation at the point after the call. This syntax has been replaced by the **wait for** statements.

2.10.2 *writeAllOutputs*(<after>)

Schedules any output data flows which have been assigned, but which have not been written using *writeOutput* (see below). The outputs are scheduled at the current time, or *after* the specified elapsed time. This syntax has been replaced by the <= syntax that also allows an *after* parameter.

2.10.3 *writeOutput*(outflow <, after >)

Schedules the designated output flow at the current time or *after* the specified elapsed time. This syntax has been replaced by the <= syntax that also allows an *after* parameter.

2.10.4 *rotate_left*(bitvector,numOfBits)

Deprecated form of ROL.



2.10.5 rotate_right(bitvector,numOfBits)

Deprecated form of ROR.

2.10.6 signed_shift_left(bitvector,numOfBits);

Deprecated form of SLA.

2.10.7 unsigned_shift_left(bitvector,numOfBits)

Deprecated form of SLL.

2.10.8 signed_shift_right (bitvector,numOfBits)

Deprecated form of SRA.

2.10.9 unsigned_shift_right(bitvector,numOfBits)

Deprecated form of SRL.