Chapter

63 *The ADT Library*

This chapter provides information about the library of Abstract Data Types (ADT) that comes with the SDL suite. The data types provide services that are often needed in SDL systems.

The ADT library is mainly intended for usage together with the Cadvanced/Cbasic SDL to C Compiler and the ordinary simulation, validation, and applications kernels. Some of the ADTs are, however, also possible to use together with OS integrations (Cadvanced) and with Cmicro. If this is the case it is indicated in the description of the ADT.

General

The ADT library currently contains the following:

- A package, `ctypes`, that contains a number of sorts and generators to simplify an integration with C data types
- A data type that provides handling of text files and I/O operations as SDL operator calls
- A data type to generate random numbers from a number of distributions
- A data type that implements linked lists
- Data types for byte, unsigned, long int and so on (provided only for backward compatibility; use package `ctypes` instead)
- A data type that makes it possible to define PId literals for static process instances as synonyms
- A data type that provides a number of general purpose operators that may be used to reduce the complexity of an SDL system

These data types are delivered in source code. Feel free to change and adapt these data types for your own needs.

Important!

There is no commitment from Telelogic to support the ADTs described in this chapter. Telelogic has used the ADTs in internal projects with successful results.

The files that are contained in the ADT library are located in the subdirectory `<installation directory>/include/ADT`. (**In Windows**, replace `/` in the path above with `\`)

Note: Conformance with earlier releases

The ADTs in Telelogic Tau 4.5 are backward compatible with the ADTs in earlier releases, in the sense that you only need to include the new 4.5 versions of the ADTs to obtain the same behavior.

However, it is important to remember that the ADTs and the code generators you use, must be from the same version of Telelogic Tau.

Integration with C Data Types

The package `ctypes` presented below contains a number of types and generators that is intended to directly support C data types in SDL. The package `ctypes` can also be used in OS integrations and with Cmicro. However, usage of C pointers (generator Ref) might cause problems, due to potential memory leaks and potential memory access protection between OS tasks.

The file `ctypes.sdl` is a SDL/PR version of this package suitable to use in an include statement in an SDL/PR system, while `ctypes.sun` is a SDL/GR version of the package.

In an SDL/GR system it is only necessary to insert a use clause, i.e.

```
use ctypes;
```

at a proper place. The Organizer will then by itself include the `ctypes` package, for example when the system is to be analyzed. To use the `ctypes` package in an SDL/PR system the following structure should be used.

```
/*#include 'ctypes.sdl'*/

use ctypes;
system example;
  ...
endsystem;
```

The `ctypes` package consists of the following newtypes, syntypes, and generators:

SDL	**C**
`syntype ShortInt`	`short int, short`
`syntype LongInt`	`long int, long`
`syntype UnsignedShortInt`	`unsigned short int, unsigned short`
`syntype UnsignedInt`	`unsigned int, unsigned`
`syntype UnsignedLongInt`	`unsigned long int, unsigned long`
`syntype Float`	`float`
`newtype Charstar`	`char *`

SDL	C
`newtype Voidstar`	`void *`
`newtype Voidstarstar`	`void **`
`generator Carray`	`array type`
`generator Ref`	`pointer type`

All the newtypes and syntypes introduce type names for “standard types” in C.

Some of the types and generators are briefly described below.

Charstar

In Charstar there is a literal and some operators included:

```
LITERALS
  Null;
OPERATORS
  cstar2cstring : Charstar   -> Charstring;
  cstring2cstar : Charstring -> Charstar;
  cstar2vstar   : Charstar   -> Voidstar;
  vstar2cstar   : Voidstar   -> Charstar;
  cstar2vstarstar : Charstar -> Voidstarstar;
```

Note that the operators `cstar2cstring` and `cstring2cstar` are not available when using Cmicro.

The operators are all conversion routines to convert a value from one type to another. Note that Charstar and Charstring are **not** the same types even if they both corresponds to char * in C. Note also that freeing allocated memory for Charstar is the responsibility of the user, as there is not enough information to handle this automatically (as for Charstring). For more information about how to free memory, see the Ref generator below.

Voidstarstar

The Voidstarstar type has all the properties of the Ref generator (see below). This means that `*>`, `&`, `+`, and `-` can be used and that the following literal and operators are defined:

```
LITERALS
  Null,
  Alloc;
OPERATORS
  vstarstar2vstar : Voidstarstar -> Voidstar;
  vstar2vstarstar : Voidstar -> Voidstarstar;
```

Carray

The generator Carray has the following parameters:

```
GENERATOR Carray (CONSTANT Length, TYPE Itemsort)
```

where Length is an integer giving the number of elements of the array (index from 0 to Length-1), and Itemsort gives the type of each element in the array. A Carray in SDL is translated to an array in C. Indexing a Carray variable in SDL follows the same rules as for ordinary SDL Arrays.

Ref

The generator Ref has the following definition:

```
GENERATOR Ref (TYPE Itemsort)
  LITERALS
    Null,
    Alloc;
  OPERATORS
    "*>"       : Ref, Itemsort -> Ref;
    "*>"       : Ref -> Itemsort;
    "&"        : Itemsort -> Ref;
    make!      : Itemsort -> Ref;
    free       : in/out Ref;
    "+"        : Ref, Integer -> Ref;
    "-"        : Ref, Integer -> Ref;
    Ref2VStar  : Ref -> Voidstar;
    VStar2Ref  : Voidstar -> Ref;
    Ref2VStarStar : Ref /*#REF*/ -> Voidstarstar;
  DEFAULT Null;
ENDGENERATOR Ref;

procedure Free; fpar p Voidstarstar;
external;
```

Instantiating the Ref generator creates a pointer type on the type given as generator parameter. The literals and operators have the following behavior:

- **Null**: The NULL value (= 0) for the pointer type. This is also the default value for a pointer variable.

- **Alloc**: An operator without parameters that returns a new allocated data area with the size of the Itemsort.
- ***>**: This is the extract! and modify! operator and can be used to reference the value referenced by a pointer. If p is a pointer type, `p*>` is the value the pointer refers to. Comparing with C, `p*>` is the same as `*p`. If p is a pointer to a struct, then `p*>!a` is the same as `(*p).a`.
- **&**: The & operator corresponds to the C operator with the same name. It can be used to take the address of a variable. Comparing with C, &p and &(p) in SDL is the same as &p in C.
- **make!**: The make! operator, which as usual in SDL has the syntax (. .), is a short hand for creating memory and initializing it to a given value. The statement:

  ```
  a := (. 2 .);
  ```

  has the same meaning as

  ```
  a := Alloc, a*> := 2;
  ```

- **free**: The Free operator is used to deallocate memory referenced by a Ref pointer. If the component type contains automatically handled pointers (Charstring, Octet_string, Bit_string, Bags, Own pointers, and so on) the memory for these components is also deallocated.
- **+, -**: These operator have the meaning of pointer arithmetics in exactly the same way as in C. For example, p+1 (if p is of a pointer type) will add the Itemsort size to the value p. The + and - operators are mostly used to step through an array in C.
- **ref2vstar**, **vstar2ref**, **ref2vstarstar**: These operators are conversion operators, that can be used to cast between pointers and void * and void **.
- procedure **Free**: NOTE: Old feature provided for backward compatibility. Use operator free above instead.

  This external procedure is closely connected to the Ref generator. It should be used to deallocate memory allocated by the Alloc operator. Free should be passed the pointer variable that references the data area to be released. The variable should be casted to Voidstarstar. After calling Free the pointer variable will have the value Null.

  Example: `Free(Ref2VStarStar(variable_name))`

  Apart from the difficult syntax for calling the Free procedure it has another problem, it does not free components inside the referenced data area as the free operator above does.

The SDL Analyzer can allow implicit type conversion of pointer data types created by the Ref generator; see "Implicit Type Conversions" on page 134 in chapter 3, *Using SDL Extensions, in the SDL Suite Methodology Guidelines*.

Abstract Data Type for File Manipulations and I/O

The ADT TextFile

In this section an SDL abstract data type, `TextFile`, is discussed where file manipulations and I/O operations are implemented as operations on the abstract data type. This ADT can be used also in OS integrations and in Cmicro if the target system has support for files in C.

This data type, which you may include in any SDL system, makes it possible to access, at the SDL level, a subset of the file and I/O operations provided by C.

The implementation of the operators are harmonized with the I/O in the monitor, including the Simulator Graphical User interface. All terminal I/O, for example, will be logged on the interaction log file if the monitor command Log-On is given.

The data type defines a "file" type and contains three groups of operations:

1. Operations to open and close files
2. Operations to write information onto a file
3. Operations to read information from a file.

The operations may handle I/O operations both on files and on the terminal (file `stdin` and `stdout` in C).

Note:

This data type is not intended to be used in the SDL Validator!

Purpose

The `TextFile` data type supplies basic file and I/O operations as abstract data type operations in SDL, whereby I/O may be performed within the SDL language. The operations may handle I/O both on the

terminal and on files and are harmonized with the I/O from the monitor, from the trace functions, and from the functions handling dynamic errors.

To make the data type available you include the file containing the definition with an analyzer include in an appropriate text symbol:

Example 514: Including an ADT File ––––––––––––––––––––––––––

```
/*#include 'file.pr' */
```

––

Remember that all file systems are operating system specific. Any rules in your file system apply.

Summary of Operators

The following literals are available in the data type FileName:

```
SYNTYPE FileName = Charstring
ENDSYNTYPE;

SYNONYM NULL    FileName = 'NULL';
SYNONYM stdin   FileName = 'stdin';
SYNONYM stdout  FileName = 'stdout';
SYNONYM stderr  FileName = 'stderr';
```

The following literals and operators are available in the data type `TextFile`:

```
NEWTYPE TextFile
  LITERALS
    NULL, stdin, stdout, stderr;

  OPERATORS
    GetAndOpenR   : FileName -> TextFile;
    GetAndOpenW   : FileName -> TextFile;
    OpenR         : FileName -> TextFile;
    OpenW         : FileName -> TextFile;
    OpenA         : FileName -> TextFile;
    Close         : TextFile -> TextFile;
    Flush         : TextFile -> TextFile;
    IsOpened      : TextFile -> Boolean;
    AtEOF         : TextFile -> Boolean;
    AtLastChar    : TextFile -> Boolean;

    PutReal       : TextFile, Real -> TextFile;
    PutTime       : TextFile, Time -> TextFile;
    PutDuration   : TextFile, Duration -> TextFile;
    PutPId        : TextFile, PId -> TextFile;
    PutInteger    : TextFile, Integer -> TextFile;
    PutBoolean    : TextFile, Boolean -> TextFile;
```

```
    PutCharacter  : TextFile, Character -> TextFile;
    PutCharstring : TextFile, Charstring -> TextFile;
    PutString     : TextFile, Charstring -> TextFile;
    PutLine       : TextFile, Charstring -> TextFile;
    PutNewLine    : TextFile -> TextFile;
    "//"          : TextFile, Real -> TextFile;
    "//"          : TextFile, Time -> TextFile;
    "//"          : TextFile, Duration -> TextFile;
    "//"          : TextFile, Integer -> TextFile;
    "//"          : TextFile, Charstring -> TextFile;
    "//"          : TextFile, Boolean -> TextFile;
    "//"          : TextFile, PId -> TextFile;
    "+"           : TextFile, Character -> TextFile;

    GetReal       : TextFile, Charstring -> Real;
    GetTime       : TextFile, Charstring -> Time;
    GetDuration   : TextFile, Charstring -> Duration;
    GetPId        : TextFile, Charstring -> PId;
    GetInteger    : TextFile, Charstring -> Integer;
    GetBoolean    : TextFile, Charstring -> Boolean;
    GetCharacter  : TextFile, Charstring -> Character;
    GetCharstring : TextFile, Charstring -> Charstring;
    GetString     : TextFile, Charstring -> Charstring;
    GetLine       : TextFile, Charstring -> Charstring;
    GetSeed       : TextFile, Charstring -> Integer;

    GetReal       : TextFile -> Real;
    GetTime       : TextFile -> Time;
    GetDuration   : TextFile -> Duration;
    GetPId        : TextFile -> PId;
    GetInteger    : TextFile -> Integer;
    GetBoolean    : TextFile -> Boolean;
    GetCharacter  : TextFile -> Character;
    GetCharstring : TextFile -> Charstring;
    GetString     : TextFile -> Charstring;
    GetLine       : TextFile -> Charstring;
    GetSeed       : TextFile -> Integer;
ENDNEWTYPE TextFile;
```

The operators may be divided into three groups with different purpose:

1. Operators that, together with the literals, are used for handling files.
2. Operators suited for writing information to files.
3. Operators intended for reading information from files.

The next three subsections provide the necessary information for using these operators. The data type itself will be discussed together with the operators for handling files.

File Handling Operators

First in this subsection each operator and literal will be discussed in detail and then some typical applications of the operators will be presented.

Caution!

The operators GetAndOpenR and GetAndOpenW **do not work** with the Application library. The operators GetPId and PutPId (and the // operator to write PIds) can be used with the Application library, but they will use a different output format.

Operator Behavior

The type `TextFile` is implemented using the ordinary C file type `FILE`. A `TextFile` is a pointer to a `FILE`.

```
typedef FILE * TextFile;
```

The literal `NULL` represents a null value for files. This literal is translated to `TextFileNull()` in the generated C code by an appropriate `#NAME` directive and is then implemented using the macro:

```
#define TextFileNull() (TextFile)0
```

All variables of the type `TextFile` will have this value as default value.

The literals `stdin` and `stdout` represent the standard files `stdin` and `stdout` in C, which are the files used in C for I/O to the terminal. The file `stdin` is used for reading information from the keyboard, while `stdout` is used for writing information on the screen.

The standard operators assignment and test for equality is implemented in such a way that `A:=B` means that now A refers to the same file as B, while `A=B` tests if A and B refer to the same file.

FileName

The data type `FileName` is used to represent file names in the operators `GetAndOpenR`, `GetAndOpenW`, `OpenR`, `OpenW`, and `OpenA`. It has all Charstring literals and the special synonyms `NULL`, `stdin` (input from the keyboard), `stdout` (output to the screen), and `stderr` (output to the screen from which the SDL suite was started). As `FileName` is a syn-

type of Charstring, the usual Charstring operators are defined for this type.

Caution!

The synonyms `stdin`, `stdout`, `stderr` in some circumstances hide the literals with the same names according to SDL scope rules. If that is the case, please insert a qualifier `<<type textfile>>` before the literal name.

GetAndOpenR – GetAndOpenW

The operators `GetAndOpenR` and `GetAndOpenW` are used to open a file with a name prompted for on the terminal. `GetAndOpenR` opens the file for read, while `GetAndOpenW` opens the file for write. The operators take the prompt as parameter (type charstring), print the prompt on the screen (on `stdout`), and read a file name from the keyboard (from `stdin`). An attempt is then made to open a file with that name. If the open operation was successful, a reference to the file is returned by the `GetAndOpenR` or `GetAndOpenW` operator, otherwise `NULL` is returned. After a successful open operation you may use the file for reading or writing.

If you type `<Return>`, `-` or the file name `stdin` at the prompt in `GetAndOpenR` a reference to `stdin` is returned by the operator. `GetAndOpenW` will, in the same way, return a reference to `stdout` if the prompt is answered by `<Return>`, `-` or the file name `stdout`.

Note:

To work properly in the *Simulator Graphical User Interface*, the prompt string should be terminated with: ": ", i.e. colon space.

OpenR – OpenW – OpenA

The operators `OpenR`, `OpenW`, and `OpenA` are used to open a file with a file name passed as parameter. `OpenR` opens the file for read, while `OpenW` opens the file for write and `OpenA` opens the file for append. An attempt is made to open a file with the name given as a parameter. If the open operation was successful, a reference to the file is returned by the `OpenR`, `OpenW`, or `OpenA` operator, otherwise `NULL` is returned. After a successful open operation you may read, write or append on the file.

Close

The operator `Close` is used to close the file passed as parameter. `Close` always returns the value `NULL`. This operator should be used on each file opened for write after all information is written to the file to ensure that any possibly buffered data is flushed.

> **Note:**
>
> Always close a file variable before assigning it to a new file, otherwise data may be lost.

Flush

Output to files is usually buffered, and is therefore not immediately written on the physical output device. The operator `Flush` forces the output buffer of the file that is passed as parameter to be written on the physical output device. It is equivalent to C function `fflush`.

IsOpened

The operator `IsOpened` may be used to determine if a `TextFile` is open or not. It may, for example, be used to test the result of the `Open` operation discussed above. The test `IsOpened(F)` is equivalent to `F /= NULL`.

AtEOF

The operator `AtEof` may be used to determine if a `TextFile` has reached the end of file or not. This operator could be used in order to determine when to stop reading input from a file. The test `AtEof(F)` is equivalent to `feof(F)`.

> **Note:**
>
> `AtEof` first becomes true when attempts are made to read behind the end-of-file. Operator `AtLastChar` becomes true when the last character of the file has been read, and is usually more useful than `AtEof`.

AtLastChar

The operator `AtLastChar` may be used to determine if a `TextFile` has reached the end of file or not. This operator is useful in order to determine when to stop reading input from a file. The test `AtLastChar(F)` returns `true` if there are no more characters to be read from the file.

Examples of Use

Three typical situations when you want to write information are easily identified:

1. The information is to be printed on the screen.
2. The information is to be printed on a file with a given name.
3. You want to determine at run-time where the information is to be printed.

Example 515: ADT for File I/O, Print to Screen ––––––––––––––––––

If the information is to be **printed on the screen**, you may use the following structure:

```
DCL F TextFile;
TASK F := stdout // 'example';
```

Declare a variable of type `TextFile` and assign it the value `stdout`. You may then use it in the write operators discussed under "Write Operators" on page 3151.

––

Example 516: ADT for File I/O, Print to File ––––––––––––––––––––

If the information is to be **printed on a file** with a given name, you may use the following structure:

```
DCL F TextFile;
TASK F := OpenW('filename');
TASK F := F // 'example';
```

The difference from the above is that the operator `OpenW` is used to open a file with the specified name. This outline may be complemented with a test if the `OpenW` operation was successful or not.

––

Example 517: ADT for File I/O, Accessing Text File ––––––––––––––––

If you **want to be able to determine at run-time where the information should be printed**, you should define a `TextFile` as in the examples above, and then use the following structure.

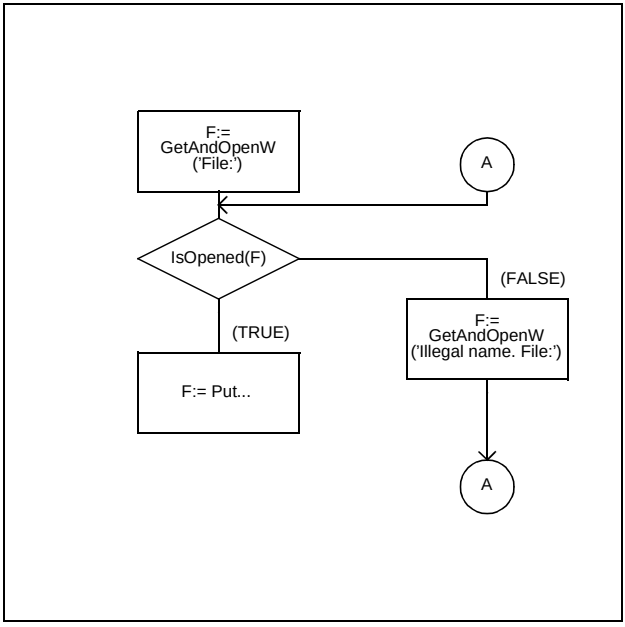

Figure 553: Accessing a TextFile

If you answer the question by hitting `<Return>` or by typing `stdout`, the information will be printed on screen (`stdout`). If you type the name of a file, the information will be printed on that file.

––

If you want to open the file for read instead of write, you may use almost identical structures.

Write Operators

Operator Behavior

The write operators `PutReal`, `PutTime`, `PutDuration`, `PutInteger`, `PutBoolean`, `PutCharacter`, and `PutCharstring` all take a `TextFile` and a value of the appropriate type as parameters. The operators print the value passed as parameter on the file referenced by the `TextFile` parameter and then return the `TextFile`. The *Put** operators will print the values in the same format as the monitor uses for the command Examine-Variable, and will append a space after each printed value.

The operator `PutString` takes a `TextFile` and a `Charstring` parameter and prints the string on the `TextFile`. `PutString` prints the string as a C string, not using the format for SDL Charstring. This means that no ' is printed. `PutString` returns the `TextFile` given as parameter as result.

The infix write operator `"//"` takes as parameters a `TextFile` and a value of type `Boolean, Charstring, Integer, PId, Real, Time,` or `Duration`. `TextF // Val` prints the value `'Val'` to the `TextFile` referenced by `'TextF'`, and returns value `'TextF'`. `Character` strings are printed without enclosing `''`. All // operators except the one for `Charstring` append a space to the file, after the value is written.

The infix write operator `"+"` takes as parameters a `TextFile` and a `Character`. `"+"` behaves just as `"//"`, but it has its special name in order to avoid type conflicts with `Charstring`.

The operator `PutNewLine` takes a `TextFile` as parameter, prints a carriage return (actually a `"\n"`) on this file, and returns the `TextFile` as operator result.

The different `Put` operators are equivalent to the `//` operators, and they are mainly present for backward compatibility reasons.

There is a function named `xPutValue` in the implementation of the data type `TextFile`. This function may print a value of any type that may be handled by the monitor system, but may only be accessed from in-line C code and not from SDL. A detailed description of the `xPutValue` function may be found under "Accessing the Operators from C" on page 3153.

Example 518: ADT for File I/O, Print to File ––––––––––––––––––––

To print a line according to the following example, where 137 is the value of the variable `NoOfJobs`:

```
Number of jobs: 137 Current time: 137.0000
```

You could use the following statements, assuming that the `TextFile F` is already opened:

```
TASK
  F := F // 'Number of jobs: ' // NoOfJobs;
TASK
  F := F // 'current time: ' // Now;
TASK
  F := PutNewLine(F);
```

––

Read Operators

Operator Behavior

The read operators `GetReal`, `GetTime`, `GetDuration`, `GetInteger`, `GetBoolean`, `GetCharacter`, `GetCharstring`, and `GetSeed` are used to read values of the various sorts.

The operator `GetSeed` is used to read appropriate values to initialize random number generators (odd integers in the range 1 to 32767).

There are two versions of each `Get` operator: one that only takes as parameters a `TextFile`, and the other that takes as parameters a `TextFile` and a `Charstring` which is used as prompt. All `Get` operators behave differently depending on if the value should be read from the terminal (`stdin`) or from a file.

If the value should be **read from the terminal**, the `Get` operators with the prompt parameter may be used. This prompt is printed on the screen, and then an attempt to read a value of the current type is made. If a `Get` operator with only the `TextFile` parameter is used, a default prompt is used, that depends on the type that is to be input. If the operation is successful, the read value is returned as a result, otherwise the message `"Illegal value"` is printed and the user is given a new chance to type a value.

Note:

To work properly in the *Simulator Graphical User Interface*, the prompt string should be terminated with: ": ", i.e. colon space.

If the value should be **read from a file**, it is recommended to use the `Get` operators without the prompt parameter, as it is not used. It is assumed that a value of the correct type will be found.

There are several `Get` operators for reading character strings:

`GetString` reads a sequence of characters until the first white space character, and is equivalent to `fscanf (f, "%s")`.

`GetLine` reads a sequence of characters until the end of line is reached. It is equivalent to `fgets`, but the end-of-line character will not be part of the string.

`GetCharstring` reads a sequence of characters on a single line that is enclosed by single quotes `(')`. This operator is mainly present for backward compatibility reasons.

There is a function named `xGetValue` in the implementation of the data type `TextFile`, which may read a value of any type that may be handled by the monitor system. This function can only be accessed from in-line C code and not from SDL. A detailed description of the `xGetValue` function may be found under "Accessing the Operators from C" on page 3153.

Example 519: ADT for File I/O, Read from File

```
TASK
  Mean := GetReal (F),
  A(1) := GetReal (F),
  A(2) := GetReal (F),
  A(3) := GetReal (F);
```

Accessing the Operators from C

In some circumstances it may be easier to use C code (in `#CODE` directives) rather than SDL to implement an algorithm. SDL implementations for linear algorithms sometimes become unnecessarily large and complex, as SDL for example lacks a loop concept. Consider the SDL

graph in Figure 553 on page 3150. This graph could be replaced by a TASK with the following contents:

```
'Open file F' /*#CODE
#(F) = GetAndOpenW("LFile : ");
while ( ! IsOpened(#(F)) )
  #(F) = GetAndOpenW("LIllegal name. File : "); */
```

which is more compact and gives a better overview at the SDL level.

#(F) is an SDL directive telling the SDL to C compiler to translate the SDL variable F to the name it will receive in the generated C code.

To simplify the use of in-line C code, #NAME directives are introduced on all identifiers defined in this data type. The same names are used in C as in SDL.

Note:

Upper and lower case letters are significant in C (but not in SDL).

Note also the additional L in the Charstring literals, for example "LIllegal name. File : ". This first character is used in the implementation of the SDL sort Charstring and should **always** be L in a charstring literal.

From in-line C, two functions xGetValue and xPutValue are also available to read and write values of any type. These functions have the following prototypes:

```
extern void xGetValue(
  TextFile       F,
  SDL_Charstring Prompt,
  xSortIdNode    SortId,
  void         * Result,
  char         * FunctionName );

extern void xPutValue(
  TextFile    F,
  xSortIdNode SortId,
  void      * Value,
  char      * FunctionName );
```

Parameter	Interpretation
TextFile F	The file to read from or to print to.
In xGetValue: SDL_Charstring Prompt	Is used as prompt in exactly the same way as for the ordinary Get operators.

Parameter	Interpretation
`xSortIdNode SortId`	A reference to the `xIdNode` that represents the SDL sort to be read or printed For the **predefined SDL sorts** you may use variables named `xSrtN_SDL_Real`, `xSrtN_SDL_Integer`, and so on, as parameter. For **user-defined sorts**, you may use similar variables named `ySrtN_#(SortName)`, where `SortName` should be replaced by the sort name.
`void * Result` `void * Value`	The address to the variable where the result should be stored, or the address to the variable that should be printed.
`char * FunctionName`	A string specifying the name of an appropriate function. This name will be given if an error is detected during reading or printing.

Note:

xGetValue and xPutValue **will not work** together with the Application library.

To handle, for example, I/O of an SDL struct, the ideas presented below may be used.

Example 520: ADT for File I/O of an SDL Struct

```
NEWTYPE SName STRUCT
  a, b Integer;
ENDNEWTYPE;

DCL
  FIn, FOut TextFile,
  SVar SName;

TASK 'Put SVar on FOut' /*#CODE
  xPutValue( #(FOut), ySrtN_#(SName),
  &(#(SVar)), "PutSName" ); */;

TASK 'Get SVar from FIn' /*#CODE
  xGetValue( #(FIn), "Value : ",
  ySrtN_#(SName),&(#(SVar)), "LGetSName"); */;
```

Abstract Data Type for Random Numbers

One important feature, especially in performance simulations, is the possibility to generate random numbers according to a number of distributions, like for example the negative exponential distribution and the Erlang distribution. It is also important that the random number sequences are reproducible, to be able to run a slightly modified version of a simulation with the same sequence of random numbers.

In this section an SDL abstract data type according to the previous discussion is presented. This data type may be included in any SDL system. This ADT can also be used in OS integrations and in Cmicro. It is, however, necessary to check that the typedef for the RandomControl, see below, refers to an unsigned 32-bit type.

Purpose

The SDL `RandomControl` data type allows you to generate pseudo-random numbers. A number of distributions are supported, including the negative exponential distribution and the Erlang distribution. In performance simulations, which is the main application area for this data type, the most important need for random numbers is in connection with `Time` and `Duration` values. It is, for example, interesting to draw inter- arrival times in job generators, and service lengths in servers. Distributions returning positive real numbers are thus most meaningful.

The basic mechanism behind pseudo-random number generation is as follows. A sequence of bit-patterns is defined using a formula of type:

$$Seq_{n+1} = f(Seq_n)$$

The function f should be such that the sequence of elements seen as numbers should be “random”, and the number of element in the sequence, before it starts to repeat itself, should be as large as possible.

To obtain a new random number is thus a two step process:

1. Compute and store a new bit-pattern from the old bit-pattern
2. Interpret the new bit-pattern as a number, which is returned as the new random number.

In this data type, 32 bit patterns, implemented in C using the type unsigned long, are used together with the formula:

$$Seq_{n+1} = ((2^{16} + 3) \cdot Seq_n) \bmod 2^{32}$$

The result returned by the operator `Random`, which is the basic random number generator, is this bit-pattern seen as a number between 0 and 1, and expressed as a float.

The data type `RandomControl` may be included in any SDL system using an analyzer include statement, where the file containing the definition of the data type is included. Example:

```
/*#include 'random.pr' */
```

As the C standard functions `log` and `exp` are used in the `random.pr` file, it is necessary to link the application together with the library for math functions, i.e. to have `-lm` in the link operation in the makefile. See "Makefile Options" on page 122 in chapter 2, *The Organizer*. To use the entry `-lm` in the link list seems to be a fairly standard way to find the library for math functions. If this does not work, or you want more details, please see the documentation for your C compiler.

Available Operators

The type `RandomControl`, introduced by this data type, is in C implemented as the address of an unsigned long.

```
typedef unsigned long * RandomControl;
```

Note that you have check that `unsigned long` is a 32 bit type, otherwise you have to change the typedef.

The reason for passing the address to the bit-pattern (that is to the unsigned long), is that this bit-pattern has to be updated by the random functions.

Below the operators provided in this data type are listed. There are, for many of the operators, several versions with different sets of parameters and/or result types to support different usage of the operator.

```
Random : RandomControl -> Real;
Random : RandomControl -> Duration;
Random : RandomControl -> Time;

Erlang : Real, Integer, RandomControl -> Real;
Erlang : Real, Integer, RandomControl -> Duration;
Erlang : Real, Integer, RandomControl -> Time;
Erlang : Duration, Integer, RandomControl -> Real;
Erlang : Duration, Integer, RandomControl -> Duration;
Erlang : Duration, Integer, RandomControl -> Time;
Erlang : Time, Integer, RandomControl -> Real;
Erlang : Time, Integer, RandomControl -> Duration;
Erlang : Time, Integer, RandomControl -> Time;
```

```
HyperExp2 :
  Real, Real, Real, RandomControl -> Real;
HyperExp2 :
  Real, Real, Real, RandomControl -> Duration;
HyperExp2 :
  Real, Real, Real, RandomControl -> Time;
HyperExp2 :
  Duration, Duration, Real, RandomControl -> Real;
HyperExp2 :
  Duration, Duration, Real, RandomControl -> Duration;
HyperExp2 :
  Duration, Duration, Real, RandomControl -> Time;
HyperExp2 :
  Time, Time, Real, RandomControl -> Real;
HyperExp2 :
  Time, Time, Real, RandomControl -> Duration;
HyperExp2 :
  Time, Time, Real, RandomControl -> Time;

NegExp : Real, RandomControl -> Real;
NegExp : Real, RandomControl -> Duration;
NegExp : Real, RandomControl -> Time;
NegExp : Duration, RandomControl -> Real;
NegExp : Duration, Rando mControl -> Duration;
NegExp : Duration, RandomControl -> Time;
NegExp : Time, RandomControl -> Real;
NegExp : Time, RandomControl -> Duration;
NegExp : Time, RandomControl -> Time;

Uniform : Real, Real, RandomControl -> Real;
Uniform : Real, Real, RandomControl -> Duration;
Uniform : Real, Real, RandomControl -> Time;
Uniform : Duration, Duration, RandomControl -> Real;
Uniform :
  Duration, Duration, RandomControl -> Duration;
Uniform : Duration, Duration, RandomControl -> Time;
Uniform : Time, Time, RandomControl -> Real;
Uniform : Time, Time, RandomControl -> Duration;
Uniform : Time, Time, RandomControl -> Time;

Draw    : Real, RandomControl -> Boolean;

Geometric : Real, RandomControl -> Integer;
Geometric : Real, RandomControl -> Duration;
Geometric : Real, RandomControl -> Time;
Geometric : Duration, RandomControl -> Integer;
Geometric : Duration, RandomControl -> Duration;
Geometric : Duration, RandomControl -> Time;
Geometric : Time, RandomControl -> Integer;
Geometric : Time, RandomControl -> Duration;
Geometric : Time, RandomControl -> Time;

Poisson : Real, RandomControl -> Integer;
Poisson : Real, RandomControl -> Duration;
```

```
Poisson : Real, RandomControl -> Time;
Poisson : Duration, RandomControl -> Integer;
Poisson : Duration, RandomControl -> Duration;
Poisson : Duration, RandomControl -> Time;
Poisson : Time, RandomControl -> Integer;
Poisson : Time, RandomControl -> Duration;
Poisson : Time, RandomControl -> Time;

RandInt : Integer, Integer, RandomControl -> Integer;
RandInt : Integer, Integer, RandomControl -> Duration;
RandInt : Integer, Integer, RandomControl -> Time;

DefineSeed : Integer -> RandomControl;
GetSeed    : Charstring -> Integer;
Seed       : RandomControl -> Integer;
```

Random (RandomControl)

The operator `Random` is the basic random generator and is called by all the other operators. `Random` uses the formula

$$Seq_{n+1} = ((2^{16} + 3) \cdot Seq_n) \bmod 2^{32}$$

to compute the next value stored in the parameter of type `RandomControl`. The result from `Random` is a real random number in the interval 0.0 < Value < 1.0.

Erlang (Mean, N, RandomControl)

The operator Erlang provides random numbers from the Erlang-N distribution with mean `Mean`. The first parameter Mean should be > 0.0, and the second parameter N should be > 0.

HyperExp2 (Mean1, Mean2, Alpha, RandomControl)

The `HyperExp2` operator provides random numbers from the hyperexponential distribution. With probability `Alpha` it return a random number from the negative exponential distribution with mean `Mean1`, and with the probability `1-Alpha` it returns a random number from the negative exponential distribution with mean `Mean2`. `Mean1` and `Mean2` should be > 0.0, and `Alpha` should be in the range 0.0 <= `Alpha` <= 1.0.

NegExp (Mean, RandomControl)

The operator `NegExp` provides random numbers from the negative exponential distribution with mean `Mean`. `Mean` should be > 0.0.

Uniform (Low, High, RandomControl)

The operator `Uniform` is given a range `Low` to `High` and returns a uniformly distributed random number in this range.
`Low` should be <= `High`.

Draw (Alpha, RandomControl)

The `Draw` operator returns true with the probability `Alpha` and false with the probability `1-Alpha`. `Alpha` should be in the range 0.0 <= `Alpha` <= 1.0.

Geometric (p, RandomControl)

The operator `Geometric` returns an integer random number according to the geometric distribution with the mean p/(1-p). The parameter `p` should be 0.0 <= `p` < 1.0.

Caution!

Since the range of feasible samples from the distribution is infinite and the result type is integer, integer overflow may occur.

Poisson (m, RandomControl)

The operator `Poisson` returns an integer random number according to the Poisson distribution with mean `m`. The parameter `m` should be >= 0.0.

Caution!

Since the range of feasible samples from the distribution is infinite and the result type is integer, integer overflow may occur.

RandInt (Low, High, RandomControl)

This operator `RandInt` returns one of the values `Low`, `Low+1`, . . . , `High-1`, `High`, with equal probability. `Low` should be <= `High`.

DefineSeed (Integer) -> RandomControl

Each `RandomControl` variable, which is used as a control variable for a random generator, has to be initialized correctly so the first bit-pattern used by the basic random function is a legal pattern. This `DefineSeed`

operator takes an integer parameter, which should be an odd value in the range 1 to 32767, and creates a legal bit-pattern. This first value is usually referred to as the seed for the random generator. Using the same seed value, the same random number sequence is generated, which means that the random number sequences are reproducible.

Seed (RandomControl) -> Integer

The `Seed` operator returns random numbers that are acceptable as parameters to the operator `DefineSeed`. If many `RandomControl` variables are to be initialized, the `Seed` operator may be useful.

GetSeed (Prompt) -> Integer

The `GetSeed` operator, which is implemented in the data type `TextFile` (see "The ADT TextFile" on page 3143), may be used to read an integer value that is acceptable as parameter to the `DefineSeed` operator.

Using the Data Type

To use the abstract data type for random number generation you must:

- Include the definition of the data type using an analyzer include. Usually it is appropriate to include the data type in a text symbol in the system diagram.
- Define a suitable number of `RandomControl` variables, one for each random number sequence that is to be used.
- Initialize the `RandomControl` variables, either in the variable declaration or in a `TASK` often placed in the start transition of the process. The operator `DefineSeed` should be used to initialize a `RandomControl` variable.
- Use the `RandomControl` variables in appropriate random number operators.

Note:

SDL variables can only be declared in processes and will be local to the process instances.

To have global RandomControl variables you may, however, define synonyms of type RandomControl and use them in random generator operators.

Example 521: Using RandomControl, DefineSeed ––––––––––––––––

```
SYNONYM Seed1 RandomControl =
  DefineSeed(GetSeed(stdin, 'Seed1 : '));

TASK Delay := NegExp(Mean1, Seed1);
```

––

This is correct according to SDL as operators only have IN parameters and therefore expressions are allowed as actual parameters. In C it is also an IN parameter and cannot be changed. But as a RandomControl value is an address it is possible to change the contents in that address.

The SDL to C Compiler will, for synonyms that cannot be computed at generation time, allocate a variable and initialize it according to the synonym definition at start-up time. Note that this will be performed before any transitions have been executed.

A typical application of RandomControl synonyms are together with the Seed operator. The Seed operator is used to generate values suitable to initialize RandomControl variables with.

Example 522: Using RandomControl, Seed ––––––––––––––––––––

```
SYNONYM BasicSeed RandomControl =
  DefineSeed(GetSeed(stdin, 'Seed : '));

DCL S1 RandomControl :=
  DefineSeed(Seed(BasicSeed));
DCL S2 RandomControl :=
  DefineSeed(Seed(BasicSeed));
```

––

The variety of operators with the same name makes it possible to directly use operators in many more situations. This is called overloading of operators. If, for example, there were only the NegExp version:

```
NegExp : Real, RandomControl -> Real;
```

then explicit conversion operators would have been necessary to draw, for example, a Duration value from the negative exponential distribution. The code to draw a Duration value would then be something like:

```
RealToDuration(NegExp(Mean, Seq))
```

We have instead introduced several operators with the same name and purpose, but with different combinations of parameter types and result type. So for the `NegExp` operator discussed above, there is also a version:

```
NegExp : Real, RandomControl -> Duration;
```

which is exactly what we wanted.

There is, however, a price to be paid for having overloaded operators. It must be possible for the SDL Analyzer to tell which operator that is used in a particular situation. It then uses all available information about the parameters and what the result is used for. Consider Example 523 below.

Example 523: Overloaded Operator ––––––––––––––––––––––––––

```
TIMER T;
DCL
  Mean, Rand Real,
  D Duration,
  Seq RandomControl :=
    DefineSeed(GetSeed('Seed : '));

TASK Rand := NegExp(Mean, Seq);
TASK D := NegExp(Mean, Seq);
TASK D := NegExp(TYPE Real 1.5, Seq);

DECISION NegExp(Mean, Seq) >
         TYPE Duration 10.0;
  (true) : ....
  ELSE :
ENDDECISION;
SET (Now + NegExp(Mean, Seq), T);
```

- The first two applications of `NegExp` are no problem, as the parameter type is given by the type of the `Mean` variable, and the result type is given by the variable that result is assigned to.
- In the third `NegExp` call, the value 1.5 has to be given a qualifier, that is, `TYPE Real`, as the literal 1.5 may be of type `Real`, `Duration`, or `Time`.
- In the fourth example it is the result type that cannot be determined if the literal 10.0 was not given with a qualifier.
- In the fifth example the only + operator that takes `Time` as left parameter and returns `Time` (`SET` should have a `Time` value as first parameter) is:

```
"+" : Time, Duration -> Time;
```

defined in the sort `Time`. So, both the type for the parameter and the result are possible to determine for the `NegExp` operator in this example.

Most of these problems can be avoided by using `SYNONYMS` or variables instead of literal values. This is in most cases a better solution than to introduce qualifiers.

Example 524: Using SYNONYMS

If, for example, the synonyms:

```
SYNONYM MeanValue Real = 1.5;
SYNONYM Limit Duration = 10.0;
```

were defined, the third and fourth `NegExp` call would cause no problem:

```
TASK D := NegExp(MeanValue, Seq);
DECISION NegExp(Mean, Seq) > Limit;
  (true) : ....
  ELSE :
ENDDECISION;
```

Trace Printouts

Trace printouts are available for the functions in this abstract data type. By assigning a trace value greater or equal to nine (9) using the monitor command Set-Trace, each call to an operator in this data type causes a printout of the name of the operator.

> **Note:**
>
> Each operator returning a random number will call the basic operator `Random` at least once.

Accessing the Operators from C

The operator for random number generation may be used directly in C by using the name given in the appropriate `#NAME` directive. Please look at the `random.pr` file for the `#NAME` directives.

Abstract Data Types for List Processing

The abstract data types defined in this “package” are intended for processing of linked lists. Linked lists are commonly appearing in applications and are one of the basic data structures in computer science. With these data types, you can concentrate on using the lists and do not have to worry about the implementation details, as all list manipulations are hidden in operators in the data types.

> **Note:**
>
> This data type is **not** implemented in a way that makes it possible to be used in the SDL Validator. It can be used in OS integrations and with Cmicro, but it is **not** recommended, due to the risk for memory leaks.

Purpose

Definitions

A *queue* is a list in which the members are ordered. The ordering is entirely performed by the user. The available operations make it possible to access members of the queue and insert members into or remove members from any position. Furthermore, the operators suppress the implementation aspects. That is, the fact that the queue is implemented as a doubly linked list with a queue head. The operators also prevent the unwary user from trying to access, for instance, the successor of the last member or the predecessor of the first member.

The entities which may be members of a queue are called *object instances*. An object instance is a passive entity containing user defined information. This information is described in the *object description.*

In SDL these definitions are implemented using sorts called `Queue`, `ObjectInstance`, and `ObjectDescr`, where `ObjectDescr` should be defined by the user. `ObjectDescr` should have the structure given in the example below (Example 525).

The data types for list processing may be included in any SDL system using Analyzer `#include` statements, where the files containing the definitions of the data types are included. The definitions should be placed in the order given in the example below:

Example 525: Including ADT for List Processing

```
/*#include 'list1.pr'*/
NEWTYPE ObjectDescr /*#NAME 'ObjectDescr'*/
  STRUCT
    SysVar SysTypeObject;
    /* other user defined components */
ENDNEWTYPE;
/*#include 'list2.pr'*/
```

The file `list1.pr` contains the definition of the sort `Queue` (and the help sorts `ObjectType` and `SysTypeObject`), while the file `list2.pr` contains the definition of the type `ObjectInstance`.

Available Sorts

When the data types for list processing are included, two new sorts, `Queue` and `ObjectInstance`, are mainly defined, together with the type `ObjectDescr` defined by the user. The user can declare variables of type `Queue` and type `ObjectInstance`, but should never declare a variable of type `ObjectDescr`.

Variables of the sorts `Queue` and `ObjectInstance` are references (pointers) to the representation of the queue or the object instance. In both sorts there is a null value, the literal `NULL`, which indicates that a variable refers to no queue or no object instance. The default value for `Queue` and `ObjectInstance` variables is `NULL`.

A variable of sort `ObjectInstance` can refer to a data area containing the components defined in the struct `ObjectDescr`. The example below shows how to manipulate these components.

Example 526: ADT for List Processing, Struct ObjectDescr

```
/*#include 'list1.pr'*/
NEWTYPE ObjectDescr /*#NAME 'ObjectDescr'*/
  STRUCT
    SysVar SysTypeObject;
    Component1  Integer;
    Component2  Boolean;
ENDNEWTYPE;
/*#include 'list2.pr'*/

DCL O1 ObjectInstance;

TASK
  O1 := NewObject;  /* see next section */
```

```
TASK
  O1!ref!Component1 := 23,
  O1!ref!Component2 := false;

TASK
  IntVar := O1!ref!Component1,
  BoolVar := O1!ref!Component2;
```

A component is thus referenced by the syntax:

```
ObjectInstanceVariable ! ref ! ComponentName
```

Caution!

You should never directly manipulate the component `SysVar` in the struct `ObjectDescr`. It contains information about if and how the object instance is inserted into a queue and should only be used by the queue handling operators.

Assignments and test for equality may be performed for queues and for object instances. The assignments:

```
Q1 := Q2;  O1 := O2;
```

mean that `Q1` now refers to the same queue as `Q2` and that `O1` now refers to the same object instance as `O2`. Assignment is thus implemented as copying of the reference to the queue (and not as copying of the contents of the queue). The same is true for object instances.

The test for equality is in the same way implemented as a test if the left and right hand expression reference the same queue or the same object instance (and not if two queue or object instances have the same contents).

Due to the order in which the sorts are defined, a component of sort `Queue` can be a part of the `ObjectDescr` struct, while components of type `ObjectInstance` cannot be part of `ObjectDescr`.

If you want several different types of objects in a queue, with different contents, the `#UNION` directive (see *"Union" on page 2598 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler*) may be used according to the following example:

Example 527: Unions and Queues ––––––––––––––––––––––––––––

```
NEWTYPE Ob1 STRUCT
  Comp1Ob1 integer;
  Comp2Ob1 boolean;
ENDNEWTYPE;

NEWTYPE Ob2 STRUCT
  Comp1Ob2 character;
  Comp2Ob2 charstring;
ENDNEWTYPE;

NEWTYPE Ob /*#UNION*/ STRUCT
  Tag integer;
  C1  Ob1;
  C2  Ob2;
ENDNEWTYPE;

NEWTYPE ObjectDescr /*#NAME 'ObjectDescr'*/
  STRUCT
    SysVar SysTypeObject;
    U      Ob;
/*#ADT (X)*/
ENDNEWTYPE;
```

The components may now be reached using:

```
O1 ! ref ! U ! Tag
O1 ! ref ! U ! C1 ! Comp1Ob1
O1 ! ref ! U ! C2 ! Comp2Ob1
```

––

Available Operators

Operators in the Sort Queue

In the sort `Queue`, the following literals and operators are available:

```
null
NewQueue

Cardinal     : Queue -> Integer;
DisposeQueue : Queue -> Queue;
Empty        : Queue -> Boolean;
FirstInQueue : Queue -> ObjectInstance;
Follow       :
     Queue, ObjectInstance, ObjectInstance -> Queue;
IntoAsFirst  : Queue, ObjectInstance -> Queue;
IntoAsLast   : Queue, ObjectInstance -> Queue;
LastInQueue  : Queue -> ObjectInstance;
Member       : Queue, ObjectInstance ->  Boolean;
```

```
Precede      :
     Queue, ObjectInstance, ObjectInstance -> Queue;
Predecessor  : ObjectInstance -> ObjectInstance;
Remove       : ObjectInstance -> ObjectInstance;
Successor    : ObjectInstance -> ObjectInstance;
```

Operators in the Sort ObjectInstance

In the sort `ObjectInstance`, the following literals and operators are available:

```
null
NewObject

DisposeObject: ObjectInstance -> ObjectInstance;
```

The operators defined in the sorts `Queue` and `ObjectInstance` have the behavior described below. All operators will check the consistency of the parameters. Each queue and object instance parameter should, for example, be `/= null`. If an error is detected the operator will cause an SDL dynamic error that will be treated as any other dynamic error found in an SDL system.

NewQueue: -> Queue

The literal `NewQueue` is used as an operator with no parameters and returns a reference to a new empty queue. The data area used to represent the queue is taken from an avail stack maintained by the list processing sorts. Only if the avail stack is empty new dynamic memory is allocated.

Cardinal: Queue -> Integer

This operator takes a reference to a queue as parameter and returns the number of components in the queue.

DisposeQueue: Queue -> Queue

This operator take a reference to a queue as parameter and returns all object instances and the data area used to represent the queue to the avail stack mentioned in the presentation of `NewQueue`. `DisposeQueue` always returns the value `null`.

> **Note:**
>
> Any references to an object instance or to a queue that is returned to the avail stack is now invalid and any use of such a reference is erroneous and has an unpredictable result.

Empty: Queue -> Boolean

This operator takes a reference to a queue as parameter and returns `false` if the queue contains any object instances. Otherwise the operator returns `true`.

FirstInQueue: Queue -> ObjectInstance

This operator takes a reference to a queue as parameter and returns a reference to the first object instance in the queue. If the queue is empty, `null` is returned.

Follow: Queue, ObjectInstance, ObjectInstance -> Queue

Follow takes a reference to a queue and to two object instances and inserts the first object instance directly after the second object instance. It is assumed and checked that the second object instance is a member of the queue given as parameter, and that the first object instance is not a member of any queue prior to the call.

Note:

The operator `Member` is used to check that the second object instance is member of the queue.

IntoAsFirst: Queue, ObjectInstance -> Queue

This operator takes a reference to a queue and to an object instance and inserts the object instance as the first object in the queue. The queue given as parameter is returned as result from the operator. It is assumed and checked that the object instance is not a member of any queue prior to the call.

IntoAsLast: Queue, ObjectInstance -> Queue

This operator takes a reference to a queue and to an object instance and inserts the object instance as last object in the queue. The queue given as parameter is returned as result from the operator. It is assumed and checked that the object instance is not a member of any queue prior to the call.

LastInQueue: Queue -> ObjectInstance

This operator takes a reference to a queue as parameter and returns a reference to the last object instance in the queue. If the queue is empty, `null` is returned.

Member: Queue, ObjectInstance -> Boolean

This operator takes a reference to a queue and to an object instance and returns `true` if the object instance is member of the queue, otherwise it returns `false`.

Precede: Queue, ObjectInstance, ObjectInstance-> Queue

Precede takes a reference to a queue and to two object instances and inserts the first object instance directly before the second object instance. It is assumed and checked that the second object instance is a member of the queue given as parameter, and that the first object instance is not a member of any queue prior to the call.

Note:

The operator `Member` is used to check that the second object instance is member of the queue.

Predecessor: ObjectInstance -> ObjectInstance

This operator takes a reference to an object instance and returns a reference to the object instance immediately before the current object instance. If the object instance given as parameter is the first object in the queue, `null` is returned. It is assumed and checked that the object instance given as parameter is a member of a queue.

Remove: ObjectInstance -> ObjectInstance

Remove takes a reference to an object instance and removes it from the queue it is currently a member of. A reference to the object instance is returned as result from the operator. It is assumed and checked that the object instance given as parameter is a member of a queue.

Successor: ObjectInstance -> ObjectInstance

This operator takes a reference to an object instance and returns a reference to the object instance immediately after the current object instance. If the object instance given as parameter is the last object in the queue,

`null` is returned. It is assumed and checked that the object instance given as parameter is a member of a queue.

NewObject: -> ObjectInstance

The literal `NewObject` is used as an operator with no parameters and returns a reference to a new object instance, which is not member of any queue. The data area used to represent the object instance is taken from an avail stack maintained by the list processing sorts. Only if the avail stack is empty new dynamic memory is allocated.

DisposeObject: ObjectInstance -> ObjectInstance

This operator take a reference to an object instance as parameter and returns it to the avail stack mentioned above. `DisposeObject` always returns the value `null`.

> **Note:**
>
> Any references to an object instance that is returned to the available stack are now invalid and any use of such a reference is erroneous and has an unpredictable result.

Examples of Use

In this section a number of examples will be given to give some indications of how to use the list processing "package". The following sort definitions are assumed to be included in the system diagram:

```
/*#include 'list1.pr' */

NEWTYPE ObjectDescr /*#NAME 'ObjectDescr'*/
  STRUCT
    SysVar SysTypeObject;
    Number Integer;
    Name   Charstring;
ENDNEWTYPE;

/*#include 'list2.pr' */
```

Example 528: Creating a Queue ––––––––––––––––––––––––––––––

To create a new queue and insert two objects in the queue, so that the first object has `Number = 23` and `Name = 'xyz'` and the second object has `Number = 139` and `Name = 'Telelogic'`, you could use the following code (assuming appropriate variable declarations):

```
TASK
  Q := NewQueue,
  O1 := NewObject,
  O1!ref!Number := 23,
  O1!ref!Name := 'xyz',
  Q := IntoAsFirst(Q, O1),
  O1 := NewObject,
  O1!ref!Number := 139,
  O1!ref!Name := 'Telelogic',
  Q := IntoAsLast(Q, O1);
```

Example 529: Removing from Queue

To remove the last object instance from a queue, assuming the queue is not empty, you could use the following code:

```
TASK
  O1 := Remove(LastInQueue(Q));
```

Example 530: Looking in Queue

You may look at the component `Name` in the first object instance in the queue in the following way:

```
TASK
  O1 := FirstInQueue(Q),
  StringVar := O1!ref!Name;
```

or if the reference to `O1` is not going to be used any further

```
TASK
  StringVar := FirstInQueue(Q)!ref!Name;
```

Example 531: Searching in Queue ––––––––––––––––––––––––––––

The result of the following algorithm is that `O1` will be a reference to the first object instance that has the value `IntVar` in the component `Number`. If no such object is found `O1` is assigned the value `null`.

```
TASK O1 := FirstInQueue(Q);
NextObject:
DECISION O1 /= null;
  (true) :
    DECISION O1!ref!Number /= IntVar;
      (true):
        TASK O1 := Successor(O1);
        JOIN NextObject;
      (false):
    ENDDECISION;
  (false):
ENDDECISION;
```

––

Example 532: Removing Duplicates from Queue ––––––––––––––––

The algorithm below removes all duplicates from a queue (and returns them to the avail stack). A duplicate is here defined as an object instance with the same `Number` as a previous object in the queue.

```
TASK O1 := FirstInQueue(Q);
NextObject:
DECISION O1 /= null;
  (true) :
    TASK O2 := Successor(O1);
    NextTry:
    DECISION O2 /= null;
      (true):
        DECISION O1!ref!Number = O2!ref!Number;
          (true):
            TASK Temp := O2,
                 O2 := Successor(O2),
                 Temp := DisposeObject (
                           Remove(Temp));
          (false):
            TASK O2 := Successor(O2);
        ENDDECISION;
        JOIN NextTry;
      (false):
        TASK O1 := Successor(O1);
        JOIN NextObject;
    ENDDECISION;
  (false) :
ENDDECISION;
```

––

Connection to the Monitor

Trace printouts are available for the operators in this abstract data type. By assigning a trace value greater or equal to eight (8) using the monitor command Set-Trace, each call to an operator in this data type causes a printout of the name of the current operator. Note that some of the operators may call some other operator to perform its task.

You may use the monitor command Examine-Variable to examine the values stored in a variable of type `ObjectInstance`. By typing an additional index number after the variable `Queue` the value of the `ObjectInstance` at that position of the queue is printed.

Accessing List Operators from C

The sorts `Queue`, `ObjectInstance`, and `ObjectDescr`, and all the operators and the literals `NewQueue` and `NewObject` have the same name in C as in SDL, as `#NAME` directives are used. The literal `null` is the sort `Queue` and is translated to `QueueNull()`, while the literal `null` in sort `ObjectInstance` is translated to `ObjectInstanceNull()`.

In C you access a component in an `ObjectInstance` using the -> operator:

```
OI_Var -> Component
```

As an example of an algorithm in C, consider the algorithm in Example 531 on page 3174. A reference to the first object instance that has the value `IntVar` in the component `Number` is computed:

```
#(O1) = FirstInQueue(#(Q));
while ( #(O1) != ObjectInstanceNull () ) {
  if ( #(O1)->Number == #(IntVar) ) break;
  #(O1) = Successor(#(O1));
}
```

Abstract Data Type for Byte

In this section an abstract data type for byte, i.e. `unsigned char` in C, is presented. This ADT can be used also in OS integrations and with Cmicro. However, please see the note below.

> **Note:**
> This ADT is only provided for backward compatibility, as the new predefined data type Octet should be used instead of Byte.

Purpose

The purpose of this data type is of course to have the type byte and the byte operations available directly in SDL.

The data type becomes available by including the file containing the definition with an analyzer included in an appropriate text symbol.

Example 533: ––

```
/*#include 'byte.pr' */
```

––

Available Operators

The following operators are available in this data type:

BAND: byte, byte -> byte

Bitwise and. Corresponds to C operator &

BOR: byte, byte -> byte

Bitwise or. Corresponds to C operator |

BXOR: byte, byte -> byte

Bitwise exclusive or. Corresponds to C operator ^

BNOT: byte -> byte

Unary not. Corresponds to C operator ~

BSHL: byte, integer -> byte

Left shift of the byte parameter the number of steps specified by the integer parameter. Corresponds to C operator <<

Implementation:

```
(byte)( (b << i) & 0xFF )
```

BSHR: byte, integer -> byte

Right shift of the byte parameter the number of steps specified by the integer parameter. Corresponds to C operator >>

```
Implementation: ( b >> i)
```

BPLUS: byte, byte -> byte

Byte plus (modulus 0xFF). Corresponds to C operator +

BSUB: byte, byte -> byte

Byte minus (modulus 0xFF). Corresponds to C operator -

BMUL: byte, byte -> byte

Byte multiplication (modulus 0xFF). Corresponds to C operator *

BDIV: byte, byte -> byte

Byte division. Corresponds to C operator /

BMOD: byte, byte -> byte

Byte modulus. Corresponds to C operator %

BHEX: charstring -> byte

This operator transforms a charstring (’00’ - ’ff’ or ’FF’) into a byte. The string may be prefixed with an optional ’0x’.

I2B: integer -> byte

I2B transforms an integer in range 0 - 255 into a byte.

B2I: byte -> integer

B2I transforms a byte into an integer.

Unsigned (and Similar) Types

There are three files called:

```
unsigned.pr
unsigned_long.pr
longint.pr
```

where three SDL sorts implemented in C as unsigned, unsigned long, and long int may be found. All these types are in SDL implemented as syntypes of integer. For more information please see the definitions of the data types.

Note:

These ADTs are only provided for backward compatibility, as is recommended to use the types in the package `ctypes` instead. The package `ctypes` is discussed first in this chapter.

How to Obtain PId Literals

This section describes a way to obtain `PId` literals for static process instances. `PId` literals will make it possible to simplify the start-up phase of an SDL system, as direct communication (*OUTPUT TO*) may be used from the very beginning. It is otherwise necessary to start sending signals without *TO*, as the only `PId` values known at the beginning are the *Parent* - *Offspring* relations.

Note:

This ADT **cannot** be used in OS integrations or with Cmicro. There are, however, a special version for OS integrations that can be found in the directory for the OS integration, and a special version for Cmicro that can be found in the Cmicro installation directory.

Note:

`PId` literals cannot be created for processes within block types or system types.

Purpose

In SDL the only way to obtain a `PId` value is to use one of the basic functions *Self*, *Parent*, *Offspring*, or *Sender*. Such values may then, of course, be passed as parameters in signals, in procedure calls and in create operations.

During system start-up there is no way to obtain the `PId` value for a static process instance at the output that starts a communication session. The receiver of the first signal must therefore be implicit, by using an output without *TO*.

To be able to handle outputs without TO, in SDL-92 types and in separate generated units, complete knowledge about the structure of channels and signal routes must be known at run-time. The same knowledge is also necessary if we want to check that there is a path from the sender to the receiver in an output with TO. As the information needed about channels and signal routes requires substantial amounts of memory, it would be nice, in applications with severe memory requirements, to be able to optimize this.

To remove all information about channels and signal routes from a generated application means two things:

1. Output without TO cannot be used in SDL-92 types or in separate generated units.
2. It is not possible to check that there is a path between the sender and the receiver at an output with TO.

The second limitation is no problem as this is the way we probably want it in a running application (during debugging the test ought to be used, but not in the application).

The first limitation, that output without TO cannot be used, is however more difficult. In an SDL system not using the OO concepts (block type, process type, and so on) and not using separate generation there are no problems, but otherwise such outputs are necessary at the system start-up phase to establish communication between processes in different blocks. The purpose of this abstract data type is to provide a way to establish `PId` literals and thereby to be able to avoid outputs without TO.

The Data Type PIdLit

Caution!

The PIdLit data type should only be used in the way described here to introduce synonyms referring to static process instances. Other usage may not work!

If you are using this data type in a system that is to be validated using the SDL Validator there are two additional requirements:

- Only process types with the number of instances equal to (N,N) for N>0, may be referenced in PId_Lit operators.
- No process type with the number of instances equal to (N,N) for N>0, may contain a Stop symbol, independently if a PId_Lit operator is used for the process type or not.

The data type PIdLit contains the following operators:

```
PId_Lit : xPrsIdNode -> PId;
PId_Lit : xPrsIdNode -> PIdList;
PId_Lit : xPrsIdNode, Integer -> PId;
```

In the file containing the data type (`pidlist.pr`) there is also a synonym that you may use to access the environment:

```
SYNONYM EnvPId PId = ...;
```

The type `xPrsIdNode` corresponds to the C type `xPrsIdNode`, which is used to refer to the process nodes in the symbol table tree built up by a generated application.

Use the **first** version of `PId_Lit` to obtain a synonym referring to the process instance of a process instance set with one initial instance.

Use the **second** version of `PId_Lit` to obtain a synonym of array type referring to the process instances of a process instance set with several initial instances.

Use the **third** version of `PId_Lit` to obtain a synonym referring to one of the process instances of a process instance set with several initial instances.

To introduce PId literals implemented as SDL synonyms, follow the steps below:

1. Include the file pidlist.pr, which contains the implementation of the PIdList type, among the declarations in the system:

   ```
   /*#include 'pidlist.pr' */
   ```

2. Identify which process instance sets that should have PId literals.

3. Introduce #NAME directives for these process instance sets.

4. Insert a #CODE directive among the declarations in the system. If. however, separate generation is not used, this #CODE directive need **not** be included.

   ```
   /*#CODE
   #HEADING
   extern XCONST struct xPrsIdStruct
          yPrsR_ProcessName1;
   extern XCONST struct xPrsIdStruct
          yPrsR_ProcessName2;
   extern XCONST struct xPrsIdStruct
          yPrsR_ProcessName3;
   */
   ```

   There should be an external definition for each process instance set identified in step 2. ProcessNameX should be replaced by the name introduced in the #NAME directives for the processes.

5. For each process instance set that should have PId literals, introduce the following synonym definition in the system diagram.

   If the process type has **one initial instance**:

   ```
   SYNONYM Name1 PId =
     PId_Lit(#CODE('&yPrsR_ProcessName1'));
   ```

   If the process type has **several initial instances**:

   ```
   SYNONYM Name2 PIdList =
     PId_Lit(#CODE('&yPrsR_ProcessName2'));
   ```

   If the process type has **several initial instances, but only one of them should be possible to refer to by a synonym**:

   ```
   SYNONYM Name3 PId =
     PId_Lit(#CODE('&yPrsR_ProcessName3'), No);
   ```

where `No` should be the instance number, that is, if `No` is 2, then the synonym `Name3` should refer to the second instance of the process type.

Of course, you may choose the names of the synonyms, but the string in the `#CODE` directive should be the `xPrsIdNode` variables in the `extern` definitions discussed in step 4 above.

6. You may now use the synonyms of type `PId` that you defined in step 5 in expressions of `PId` type, for example as a receiver in the `TO` clause in an output. The synonym `EnvPId`, which refers to an environment process instance, can be used in the same way.

   Synonyms of type `PIdList` may be indexed (as an array) by an integer expression to obtain a `PId` value and may then be used in the same way as the synonyms of type `PId`. Indexes should be in the range 1 to the number of initial instances.

Example 534: PIdList Data Type ----------

```
OUTPUT Sig1 TO Name1;
OUTPUT Sig2 TO Name2(2);
OUTPUT Sig3 TO Name2(InstNo);
OUTPUT Sig4 TO EnvPId;
DECISION (Name3 = Sender);
TASK PId_Variable := Name2(1);
```

where `InstNo` is an integer variable or synonym and `PId_Variable` is a variable of type `PId`.

Note:

Note that no index check will be performed when indexing a PIdList synonym.

General Purpose Operators

Introduction

The abstract data type `IdNode` described in this section introduces a number of operators that may be used to simplify an SDL system. The simplifications will give both reduced code size and higher speed of execution for your application, as well as make debugging easier. This ADT **cannot** be used in OS integrations or with Cmicro.

The operators may be grouped into two groups:

- "Almost SDL operations", that is, operators that are easy to understand in an SDL context, but which are not available in SDL. Examples are the possibility to enumerate all active instances of a certain process instance set, or to count the number of signals in an input port.
- Operators that handle implementation aspects. An example is an operator to reuse memory in avail lists.

Caution!

Be very careful using these operators, as you will then not be designing true SDL systems.

If the SDL description is a goal in itself you should not use the operators. If the SDL system is just a means to obtain something else, an application for example, the operators may be very useful.

Type IdNode

This abstract data type becomes available by inserting the analyzer include:

```
/*#include 'idnode.pr'*/
```

This abstract data type file introduces three SDL sorts called `PrsIdNode, PrdIdNode,` and `SignalIdNode` in SDL. These sort correspond to the types `xPrsIdNode, xPrdIdNode,` and `xSignalIdNode` in C, which are used to represent the symbol table in the generated application. The symbol table, which is a tree, will contain the static information about the SDL system during the execution of the generated program.

Is is possible to refer to processes, procedures, and signals (among others) using the following the names:

```
yPrsN_ProcessName     or &yPrsR_ProcessName
yPrdN_ProcedureName   or &yPrdR_ProcedureName
ySigN_SignalName      or &ySigR_SignalName
```

where *ProcessName*, *ProcedureName*, and *SignalName* should be replaced by the name of the process, procedure, or signal with prefix, or by the name given to the unit in a `#NAME` directive. To obtain a name of a unit with prefix the directive `#SDL` may be used:

```
yPrsN_#(ProcessName)   or &yPrsR_#(ProcessName)
```

To avoid problems when separate generation is to be used, the `&yPrsR_...` syntax is recommended.

The `#SDL` directive is not always possible to use. It will look for an entity with the specified name in the current scope unit (where the `#SDL` directive is used) and outwards in the scope hierarchy. So, for example, if the reference for a process is to be used in a process defined in another block, a `#SDL` directive cannot be used for the referenced process. The name of the referenced process ought then to be given in a `#NAME` directive.

If separate generation is used there may be more problems to access these references. The variables will be defined in the compilation unit where the entity they represent is defined.

- The `xPrsIdNode` for a process will be defined in the file containing the code for the block enclosing the process.
- The `xPrdIdNode` for a procedure will be defined in the file containing code for the enclosing unit.
- The `xSignalIdNode` for a signal will be defined in the file containing code for the enclosing system, block, or process.

A reference is visible in the compilation unit (file) where it is defined and in all subunits to the unit, as a compilation unit will include the `.h` file of all its parent units.

Problems occur when we want to use a reference in a place where it is not visible, for example using an `xPrsIdNode` for a process defined in a separate block, in a process in another block. All references are, however, extern, which makes it possible for a user to introduce an appro-

priate `extern` definition (in a `#CODE` directive) himself in the compilation units where it is needed.

Example 535 ––

```
/*#CODE
#HEADING
extern XCONST struct xPrsIdStruct yPrsR_ProcessName;
*/
```

––

To know the name of the referenced process, a `#NAME` directive ought to be used.

> **Note:**
> Such `extern` definitions introduce dependencies between otherwise independent compilation units. It is your responsibility completely to maintain these dependencies.

Available Operators

GetIdNode: PId -> PrsIdNode;

This operator takes a `PId` value and returns a reference to the `PrsIdNode` that represents the process type. `PrsIdNode` values are not useful for anything except as parameters to the operators discussed here.

Kill: PId -> PId;

The `Kill` operator can be used to stop another process instance. In SDL a process instance may only stop itself. This operator has exactly the same effect as if the process instance given as parameter executed a stop operation. The `Kill` operator always returns the value `null`.

KillAll: PrsIdNode -> Integer;

This operator takes a reference to an `PrsIdNode` representing a process type and will kill all the instances of the specified process type. The effect is the same as if all the instances executed stop operations. The operator returns the number of “killed” process instances.

FirstPId: PrsdNode -> PId;

See “SucPId: PId -> PId;” on page 3186 (next).

SucPId: PId -> PId;

This operator, together with FirstPId, are intended to be used to enumerate all process instances of the process type referenced by the `PrsIdNode` given as parameter to `FirstPId`. `FirstPId` should be given a reference to an `PrsIdNode` for a process type and returns the first (last created) process instance. `SucPId` should be given a `PId` value and will return the next `PId` for the given process type.

> **Note:**
>
> During the enumeration of the process instances, no action that stops any instance of the enumerated process type may be executed.
>
> This means, for example, that the complete enumeration should take place in one transition and that `Kill` operations should not be used in the enumeration.

InputPortLength: PId -> Integer;

This operator returns the number of signals in the input port of the given process instance.

InputPortLength: PId, SignalIdNode -> Integer;

This operator returns the number of signals, of the signal type given as `IdNode` parameter, that are present in the input port of the given process instance. The `SignalIdNode` parameter should refer to a `SignalIdNode` that represents a signal or a timer.

NoOfProcesses: PrsIdNode -> Integer;

This operator should be given a reference to an `PrsIdNode` representing a process instance set and will return the number of active instances of this instance set.

IsStopped: PId -> Boolean;

The operator may be used to determine if a `PId` value refers to a process instance that is active or has executed a stop operation.

Broadcast: PrsIdNode, SignalIdNode, PId -> Integer;

This operator may be used to send one signal (without parameters) to each active process instance of a specified process instance set. The value of the third parameter, of type `PId`, will be used as sender in the signals. The result of the operator is the number of signals that are sent dur-

ing this operation, i.e. the number of active process instances of the specified type.

Note:

When you use this operator you hide signal sending in an expression in, for example, a task. This will decrease the readability of your SDL description, and should be well documented, at least with a comment.

FreeAvailList: PrsIdNode -> Integer and PrdIdNode -> Integer and SignalIdNode -> Integer

Note:

The FreeAvailList operator has no meaning in the SDL Validator. It can be used but will in the Validator be a null action.

The operator takes a reference to an `IdNode` (of one of the three type above) that represents a process, a procedure, or a signal and returns the memory in the avail list for the specified `IdNode` to the free memory by calling the `sctOS` function `xFree`. The function `xFree` uses the C standard function 'free' to release the memory. The `FreeAvailList` operator requires thus that free really releases the memory in such a way that it can be reused in subsequent memory allocations. Otherwise the operator is meaningless.

`FreeAvailList` is intended to be applied for reusing memory allocated for processes, procedures, and signals used only during a start-up phase. If the system, for example, contains a process used only during start-up, that is, all instances of this process perform stop actions early during the execution and no more processes will be created later, then the memory for these instances can be reused.

Caution!

This operator should only be used as one of the last resorts in the process of minimizing the memory requirements of an application.

Connection to Monitor

In the trace output, operators like `Kill` and `Broadcast` will produce trace messages exactly in the same way as the equivalent `Stop` operation and the sequence of `Output` operations.

Summary of Restrictions

The table below summarizes the restrictions concerning the usability of the various Abstract Data Types that are delivered with the SDL suite.

	Sim.	Real-Time Sim.	Perf. Sim.	Sim. with env.	Appl. with Cadv.	Valid.
`list1, list2`	✓	✓	✓	✓	✓	✝
`file`	✓	✓	✓	✓	☛	☛
`random`	✓	✓	✓	✓	✓	☛
`pidlist`	✓	✓	✓	✓	✓	☛
`idnode`	✓	✓	✓	✓	✓	✓
`byte`	✓	✓	✓	✓	✓	✓
`longint`	✓	✓	✓	✓	✓	✓
`unsigned`	✓	✓	✓	✓	✓	✓
`unsigned_long`	✓	✓	✓	✓	✓	✓

Table Legend:

✓ Compatible

✝ Incompatible

☛ Meaningless combination, or restrictions. See the respective section for more information.