PRAGMADEV
real time development tools

Real Time Developer Studio V4.3

Reference manual

Contents

Technologies overview - 9

Abstract Syntax Notation 1 (ASN.1) - 11

Presentation 11
ASN.1 module example 12
ASN.1 types 12
Base and string types 12
SEQUENCE types 13
SET types 14
CHOICE types 14
SEQUENCE OF types 14
SET OF types 14
Embedded type definitions 15
Type extensibility 15
Type constraints 16
Mapping from ASN.1 to C / SDL-RT, SDL & TTCN 16
Including ASN.1 definitions 16
Naming conventions 17
Type mapping 17

SDL reference guide - 25

Introduction 25
Benefits 25
Relation between SDL and other languages 25
SDL Components 26
Structure 26
System 26
Block(s) 26
Process(es) 27
Communication 28
Signal 29
Signal list definition 29
Channel 29
Behavior 30
Text 30
Procedure 31
Macro 36
Composite state 37
Start 40
Stop 41
State 42
Input 45
Priority input 47
Save 48
Continuous signal 49
Output 51
Priority output 52

Task 54
Process creation 55
Timer 56
Decision 58
If statement 60
Loop statement 60
Conditional expression 64
Transition option 65
Connectors 66
Text extension 67
Comment 68
Data Types 68
Basic types 68
Constants: SYNONYM 79
Renaming or constraining existing types: SYNTYPE 79
Complex types 79
Object Orientation 87
Block class 87
Process class 88
Class diagram 89

SDL support in RTDS - 92
Architecture and communication 92
Behavior 92
SDL Abstract Data Types 94
Macros diagrams 94
Composite state support 96

Mapping of SDL to IF concepts - 99
Scope 99
Translation table 99
Detailed translation rules 102
Architecture 102
Communication 103
Behavior 103
IF observers 107

SDL to C translation rules - 110
Conversion guidelines for declarations 110
SYNTYPE declaration 110
NEWTYPE declarations 111
SYNONYM declaration 115
Variable declarations 116
FPAR and RETURNS declarations 117
Other declarations 118
Conversion guidelines for statements and expressions 118
Assignment statements 118
Booleans operations 119
Numeric operations 119
Character string operations 119

String(...) types operations 120
Comparison operations 120
Conditional operator 120
Field extraction 121
Array indexing 121
Procedure and operator calls 121
Inline values for structures or arrays 121
Nested scopes management 122
Problem 122
C implementation 123

SDL to SDL-RT conversion - 129
Project tree 129
Files 130
Diagrams 130
SDL diagrams 130
UML class diagrams 131
Other diagrams 131

SDL and SDL-RT code generation - 132
Basic principles 132
C code generation with a RTOS 132
Principles 132
Generated files 138
Structure of a RTOS integration 139
Types used in the generated code 142
Generated constants and prototypes (`RTDS_gen.h`) 147
Additional generated types & macros for message handling 147
C translation for symbols 149
Memory allocation 171
Build process 172
C++ code generation for passive classes (UML) 182
C++ code generation with or without a RTOS 183
Objectives 183
Principles 183
Generated code 185
Whole system scheduling with no RTOS 189
C code generation with RTDS C scheduler 192
Process instance context handling 192
General architecture 192
Whole system scheduling with no RTOS 193
Limitations 193
C code generation with external C scheduler (SDL only) 194
Integration with external C code 198
Function call 198
Message exchange 198
RTOS integrations 201
Common features 201
Creating a new RTOS integration for RTDS 205

VxWorks integration 206
Posix integration 211
Windows integration 213
CMX RTX integration 214
uITRON 3.0 integration 222
uITRON 4.0 integration 223
Nucleus integration 225
OSE Delta integration 227
OSE Epsilon integration 230
ThreadX integration 233

TTCN-3 reference guide - 235
Acronyms 235
TTCN-3 architecture 235
Port type 235
Component type 235
Test system interface 236
Test system 236
Communication 237
Starting PTC behaviour 238
TTCN-3 test system anatomy 239
TTCN-3 Control Interface 240
TTCN-3 Executable 243
TTCN-3 Runtime Interface 244

TTCN-3 concepts support in simulation and generation - - - - - - - - - - - 246
Types and values 246
Operators 247
Modular 248
Template 249
Template matching mechanisms 249
Tests configuration 250
Functions and altsteps 252
Statements 253
Operations 254
Attributes 256

Mapping of SDL data types to TTCN data types - - - - - - - - - - - - - - - - 257

TTCN-3 Code generation- 260
Basic principles 260
Generated Files 263
TTCN Control Interface 263
Automatic main function generation (`RTDS_TTCN_main.c`) 264
Adaptation to a target 264
Generated data types 264
Requirements 264
Communication from TSI to SUT 265
Communication from SUT to TSI 266
Naming convention 266

Types used in TTCN-3 generated code - RTDS_TTCN.h 267
Generated TTCN-3 constants and prototypes - RTDS_TTCN_gen.h 270

Debugger integrations - 271
Tasking Cross View Pro debugger integration 271
Version 271
Interface 271
Make utility 271
Restrictions 271
gdb debugger integration 272
Version 272
Interface 272
Remote debugging 272
MinGW debugger integration 274
Version 274
Library 274
Interface 274
Console 274
Restriction 274
XRAY debugger integration 276
Version 276
Interface 276
Restrictions 277
Multi 2000 debugger integration 278
Version 278
Interface 278
Target connexion 278
Restrictions 278
RTDS footprints 279
Static memory footprint 279
Dynamic memory allocation 281

RTDS commands - 282
rtds: main application 282
Usage 282
Environment variables 283
rtdsPrintAssoc: display association information 283
rtdsGenerateCode: code generation 284
rtdsImportPR: PR/CIF file import 284
rtdsExportPR: PR file export 285
rtdsGenerateXmlRpcWrappers: XML-RPC wrapper generation 287
rtdsShell: RTDS command line interface 289
rtdsSimulate 290
rtdsObjectServer: RTDS API server 291
rtdsSearch: Low-level search utility 291
rtdsDiagramDiff: RTDS diagram diff utility 291
rtdsAutoMerge: RTDS diagram merge utility 292

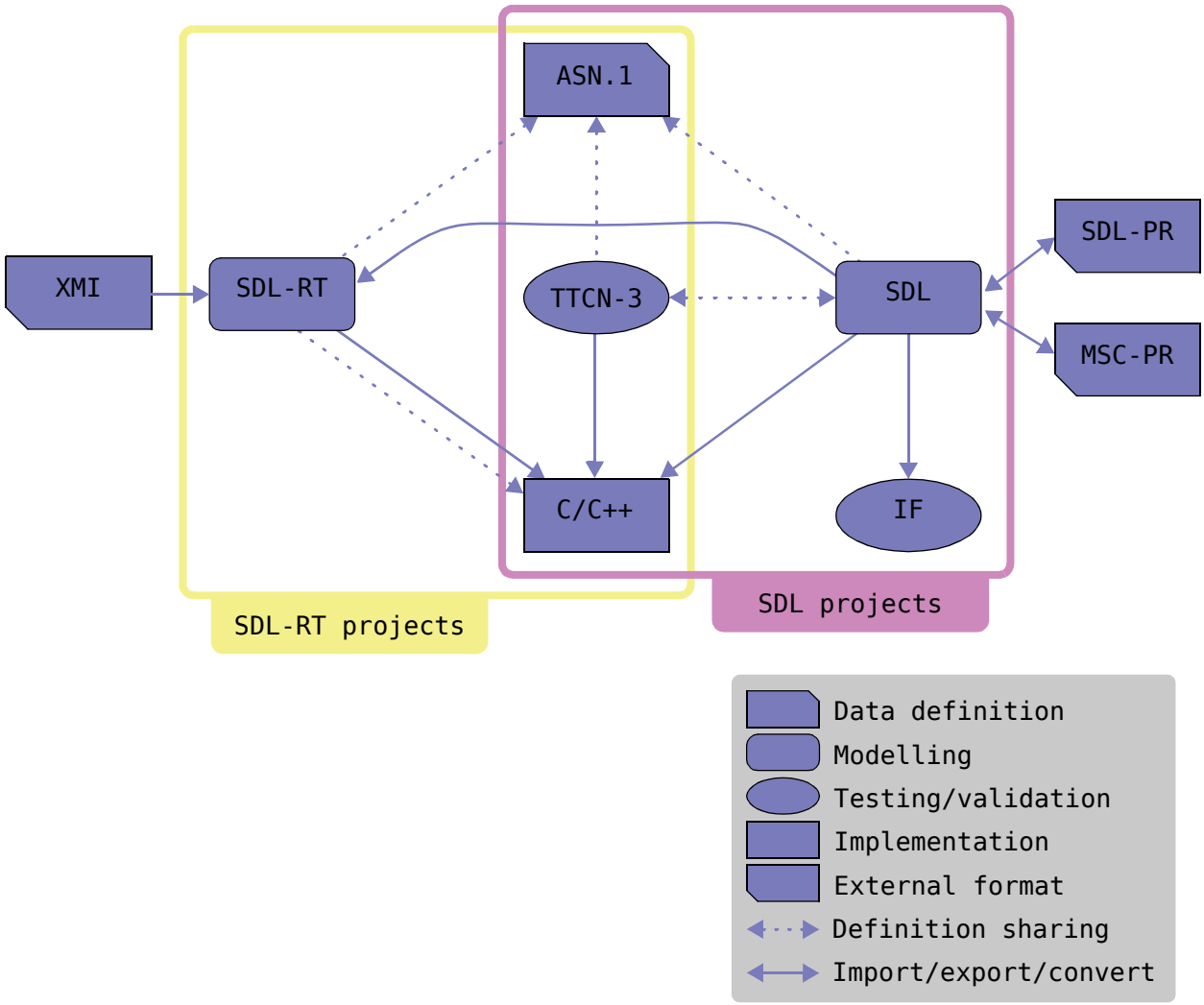
Syntax & semantics check - 293

**XMI Import - - -

Usage: cc1 input [switches] 339
Language specific options: 342
Target specific options: 349
Usage: gcc [options] file... 350
ld 351
Options 351
emulation specific options 355
gdb commands 356
Aliases 356
Breakpoints 356
Examining data 360
Files 364
Internals 365
Obscure features 366
Running the program 368
Examining the stack 372
Status inquiries 373
Support facilities 377
User-defined commands 380
**ASCII table - - -

1 - Technologies overview

The following diagram shows an overview of all technologies available in RTDS and how they interact with each other:



Details on the support of all these technologies in RTDS can be found in the following sections:

- ASN.1 is described in “Abstract Syntax Notation 1 (ASN.1)” on page 11, as well as its mapping to all other languages;
- SDL is described in “SDL reference guide” on page 25;
- Conversion from SDL to SDL-RT is described in “SDL to SDL-RT conversion” on page 129;
- Conversion from SDL to IF is described in “Mapping of SDL to IF concepts” on page 99;
- Conversion from SDL to C or C++ is described in “SDL to C translation rules” on page 110 and “SDL and SDL-RT code generation” on page 132;
- Code generation from SDL-RT is described in “SDL and SDL-RT code generation” on page 132;
- TTCN-3 is described in “TTCN-3 reference guide” on page 235;

- Shared definitions between TTCN-3 and SDL is described in “Mapping of SDL data types to TTCN data types” on page 257;
- Conversion from TTCN-3 to C or C++ is described in “TTCN-3 Code generation” on page 260;
- Importing an XMI file as a SDL-RT project is described in “XMI Import” on page 298.

2 - Abstract Syntax Notation 1 (ASN.1)

2.1 - Presentation

The *Abstract Syntax Notation* 1 or *ASN.1* is a formal language allowing to specify any kind of information exchanged within a software system in a machine- and programming language-independent way.

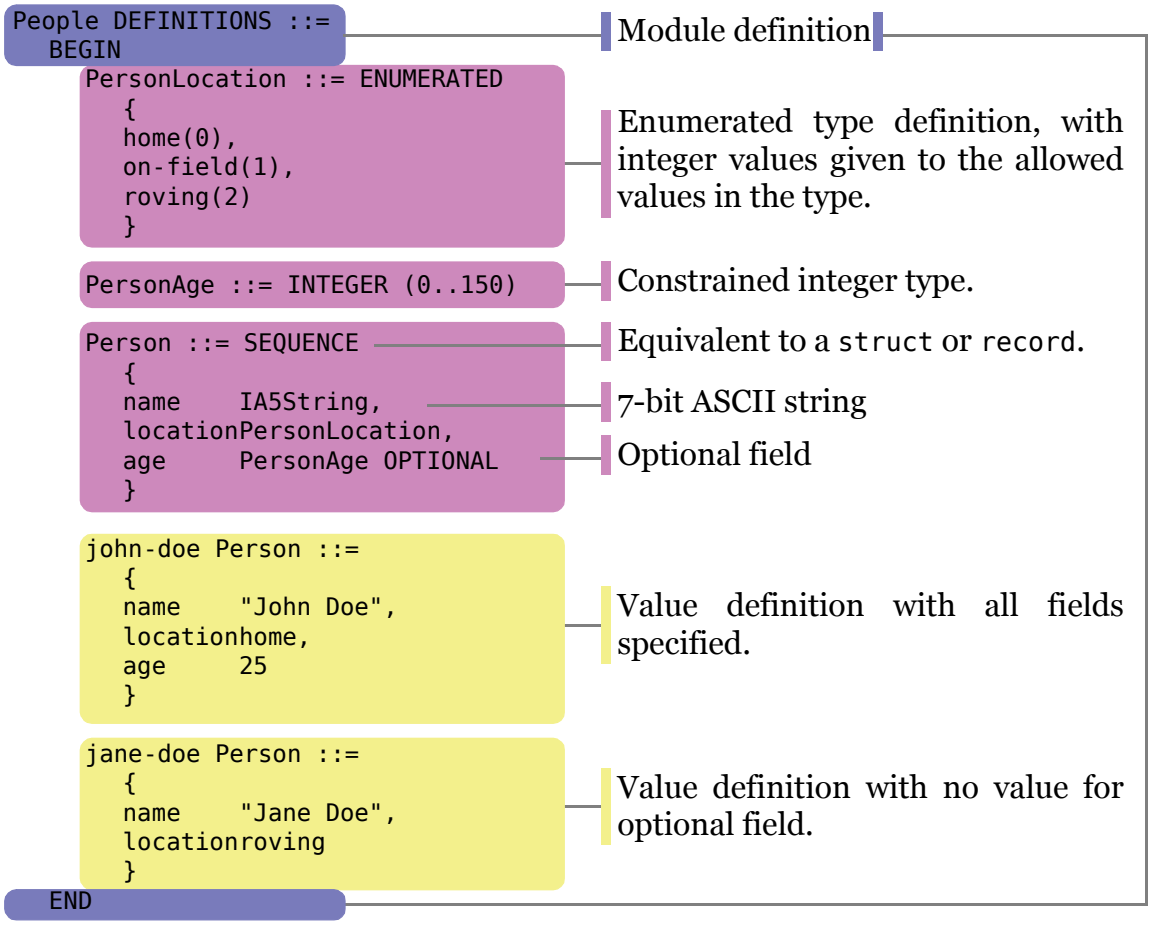
The main elements of ASN.1 are data types and values based on these types. All these definitions are grouped in what is called an ASN.1 *module*.

ASN.1 also offers a set of encoding rules to transport any kind of data from any process on any architecture to any other process on any other architecture in a deterministic and non-ambiguous way. This allows to represent any kind of data independently of the machine it is on and of the programming language used to handle it.

Note that in RTDS, ASN.1 data is only used internally, so the various encoding rules are not supported. RTDS also supports only a subset of all ASN.1 types and values that are described in the following paragraphs.

2.2 - ASN.1 module example

Here is a simple ASN.1 module example, showing some type and value definitions in a module:



2.3 - ASN.1 types

The next paragraphs give a brief description of the main ASN.1 types and indicate if and how these types are supported in RTDS.

2.3.1 Base and string types

The following table gives all ASN.1 base and string types and wether they are supported or not in RTDS:

ASN.1 type	Support	Comments
BOOLEAN	☑	
NULL	☒	

ASN.1 type	Support	Comments
INTEGER	☑	Constraints specified as list of values or ranges are supported, but not named numbers (e.g my-value(42)).
ENUMERATED	☑	
REAL	☑	The usual dotted notation (e.g 3.1416) is supported, but not the sequence like one with the mantissa, the base and the exponent.
BIT STRING	☑	Size constraints are supported.
OCTET STRING	☑	Size constraints are supported.
OBJECT IDENTIFIER	☒	
RELATIVE-OID	☒	
IA5String	☑	Used as a general string. No check is made on the actual characters in the string, which can therefore have values greater than 127.
(Other string types)	☒	

2.3.2 SEQUENCE types

In ASN.1, SEQUENCE types are a set of named fields, equivalent to struct or record types found in other languages. ASN.1 specificities include the possibility to mark a field as optional, and to give a default value to a given field.

For example:

```
Person ::= SEQUENCE
  {
  first-name  IA5String,
  last-name   IA5String,
  age         INTEGER OPTIONAL,
  registered  BOOLEAN DEFAULT TRUE
  }
```

Values for SEQUENCE types are specified as field values separated with commas and enclosed between curly brackets, with each field value preceded by the field name. For example:

```
john-doe Person ::= { first-name "John", last-name "Doe", age 42 }
```

Values for optional fields or fields with a default value can be specified, but do not need to be. All other fields must have a value. The order of the fields is not significant in values, since the field name is specified.

SEQUENCE types and values are supported in RTDS, including OPTIONAL fields, but not DEFAULT values on fields.

2.3.3 SET types

ASN.1 SET types are equivalent to SEQUENCE types, but the fields are unordered. This only has an impact on encoding, which RTDS does not support. So in RTDS, SET is just a synonym for SEQUENCE.

2.3.4 CHOICE types

ASN.1 CHOICE types allow to select exactly one out of several fields, similarly to union types in other languages. For example:

```
MealEndingType ::= CHOICE
  {
  dessert   DessertType,
  cheese    CheeseType
  }
```

Values for CHOICE types are specified with the name for the selected alternative followed by its value after a colon. For example, supposing the DessertType type is an ENUMERATED containing a value named chocolate-cake:

```
my-meal-ending MealEndingType ::= dessert:chocolate-cake
```

CHOICE types and values are fully supported in RTDS.

2.3.5 SEQUENCE OF types

ASN.1 SEQUENCE OF types are ordered lists of elements with the same type with an arbitrary length, which can be constrained. For example:

```
PolylineType ::= SEQUENCE SIZE(2..MAX) OF PointType
```

means a polyline is an ordered list of points with at least 2 points in it (MAX denotes the maximum possible value of an INTEGER).

Values for SEQUENCE OF types are the values for all elements in order, enclosed with curly brackets and separated with commas. For example, supposing PointType is a SEQUENCE type with the fields x and y of type INTEGER:

```
my-polyline PolylineType ::= {{x 0, y 0}, {x 1, y 0}, {x 2, y 3}}
```

SEQUENCE OF types and values are fully supported in RTDS.

2.3.6 SET OF types

ASN.1 SET OF types are unordered lists of elements with the same type. It is equivalent to a mathematical set, but a given element can appear several times in it. For example:

```
PrimeFactorsSet ::= SET OF INTEGER
```

As for SEQUENCE OF types, the capacity of the set can be constrained via a SIZE constraint.

Values for SET OF types are written like values for SEQUENCE OF types:

```
twelve-prime-factors PrimeFactorsSet ::= { 2, 2, 3 }
```

but the order is not significant. So:

```
twelve-prime-factors PrimeFactorsSet ::= { 2, 3, 2 }
```

or:

```
twelve-prime-factors PrimeFactorsSet ::= { 3, 2, 2 }
```

represent the same value as the first one.

`SET OF` types and values are fully supported in RTDS.

2.3.7 Embedded type definitions

In ASN.1, types for type components such as `SEQUENCE` fields, `CHOICE` alternatives or `SEQUENCE OF` elements do not have to be specified as a type name; Their definition can be embedded in the parent type. For example, the following definition is valid:

```
Player ::= SEQUENCE
  {
  full-name   SEQUENCE {first-name IA5String, last-name IA5String},
  age         INTEGER (0..150) OPTIONAL,
  rounds      SEQUENCE OF
                SEQUENCE
                  {
                  start-time SEQUENCE
                       {
                       hour INTEGER (0..23),
                       minute INTEGER (0..59),
                       second INTEGER (0..59),
                       },
                  won-points INTEGER
                  }
  }
```

ASN.1 also allows:

- To extract the definition of a type embedded in another one via the `field < TypeName` construct. For example, in the type above, the notation `age < Player` would represent the type `INTEGER (0..150)`.
- To include the complete specification of a `SEQUENCE` type within another one using the `COMPONENTS OF` construct.
- To create a subtype of a complex type such as the one above and adding constraints on specific fields using the `WITH COMPONENT(S)` construct.

Embedding type definitions is supported in RTDS, but none of the constructs above are.

2.3.8 Type extensibility

ASN.1 allows to define types that can be extended in future versions. For example:

```
Person ::= SEQUENCE
  {
  first-name  IA5String,
  last-name   IA5String,
  ...
  }
```

means the `Person` type contains at this point only the fields `first-name` and `last-name`, but can be extended in the future. An example of a redefinition of this type for a future version could be:

```
Person ::= SEQUENCE
  {
  first-name  IA5String,
  last-name   IA5String,
  ...
```

```
  age         INTEGER
  }
```

Here again, the only impact is on the encoding for the type `Person`, as all fields have to be specified completely in each version. So the syntax for extensibility is supported in RTDS, but has no effect.

2.3.9 Type constraints

Single and multiple value constraints such as in `INTEGER (0|1|4|256)` are fully supported.

Simple range constraints such as in `INTEGER (-4..4)` are supported, but:

- Exclusive ranges as in `INTEGER (-4<..<4)` are not supported;
- The special values `MIN` and `MAX` respectively representing the minimum and maximum values for an `INTEGER` are not recognized.

Type inclusion constraints are not supported. So for example:

```
Day ::= ENUMERATED {monday, tuesday, wednesday, thursday,
  friday, saturday, sunday }
WeekEnd ::= Day(saturday |sunday)
```

works, but:

```
LongWeekEnd ::= Day(WeekEnd | monday)
```

is not supported.

Since only the type `IA5String` is supported, constraints on the alphabet used in strings are not supported.

The `PATTERN` keyword allowing to constrain a string via a regular expression is not recognized.

Additional constraints put on `SEQUENCE` / `SET` / `CHOICE` fields or `SEQUENCE OF` / `SET OF` elements via the `WITH COMPONENT(S)` directive are not supported.

2.4 - Mapping from ASN.1 to C / SDL-RT, SDL & TTCN

ASN.1 type specifications can be included in any project, allowing to share data types and values across the different languages RTDS supports.

2.4.1 Including ASN.1 definitions

Including type and values definitions written is ASN.1 is made differently depending on the language:

- For SDL-RT or SDL systems:
  - An ASN.1 module included in the RTDS project at the same package level as the system will be automatically included in it. Folders can be used to separate the ASN.1 part from the SDL-RT / SDL part.
  - If an ASN.1 module is in a package, a SDL-RT or SDL system containing a `USE` clause on this package will import all definitions it contains.
- For TTCN-3 modules, including ASN.1 definitions is made via the standard import mechanism, for example:
  ```
  import from ASN1Definitions language "ASN.1:2002" all
  ```

This will import all definitions stored in the ASN.1 module called `ASN1Definitions`, which must be in a file called `ASN1Definitions.asn1` or `ASN1Definitions.asn` at the same package level as the TTCN-3 module. Here again, folders can be used to separate the different parts.
Note the year specified in the language clause for the import is actually ignored.
There is today no possibility to include a set of ASN.1 definitions from a module in another package than the parent one for the TTCN-3 module.

2.4.2 Naming conventions

Generally, the names found in the ASN.1 definitions are reused in all mappings for all languages. The only exception is the dash character ("-"), valid in ASN.1 identifiers but neither in C, SDL-RT, SDL or TTCN-3 ones, which is transformed to an underscore ("_"). This cannot generate problems, as the underscore is forbidden in ASN.1 identifiers.

There are also a few cases where the ASN.1 identifier can be transformed; These are described in the next paragraph.

2.4.3 Type mapping

The following table shows the mapping from the supported ASN.1 types to their equivalent in C / SDL-RT, SDL and TTCN.

BOOLEAN	
SDL RT	An ASN.1 `BOOLEAN` is mapped to a `RTDS_BOOLEAN`, which is defined as: `enum { FALSE = 0, TRUE = 1 }` in the file `RTDS_CommonTypes.h` in `$RTDS_HOME/share/ccg/common`. This file is always automatically included in the generated code for all projects.
sdl	An ASN.1 `BOOLEAN` is mapped to a SDL `Boolean`.
TTCN-3	An ASN.1 `BOOLEAN` is mapped to a TTCN-3 `boolean`.
INTEGER	
SDL RT	An ASN.1 `INTEGER` is mapped to a C `int`. Note that constraints set on the ASN.1 type are not considered in the generated C code, except if the type is used as an index for an array, for example. In this case, the constraints are used to figure out the minimum and maximum value for the type to set the number of elements for the generated array.
sdl	An ASN.1 `INTEGER` is mapped to a SDL `Integer`. If constraints are set on the ASN.1 type, they are recreated via a SDL `SYNTYPE` based on an `Integer`.

TTCN-3	An ASN.1 `INTEGER` is mapped to a TTCN-3 `integer`. If constraints are set on the ASN.1 type, they are recreated via a TTCN-3 sub-type of `integer`.
ENUMERATED	
SDL RT	An ASN.1 `ENUMERATED` type is mapped to a C `enum`. The constants defined by the type have the same name in C as in ASN.1, optionally prefixed with the prefix for constants and/or the type name, depending on the generation options. The numerical values set for the values are also used in the generated C type.
sdl	An ASN.1 `ENUMERATED` type is mapped to a SDL `NEWTYPE` defining a set of `LITERALS`, having the same name as the ASN.1 `ENUMERATED` type values. Note that the numerical values set for the values will not be accessible through SDL, as SDL does not support them on `LITERALS`. They will however be remembered and used in any code generation from the SDL system.
TTCN-3	An ASN.1 `ENUMERATED` type is mapped to a TTCN-3 `enumerated` type defining the same constants. The same note as for SDL applies: The numerical values set in the ASN.1 type are not accessible via TTCN-3, but are remembered and used in generated code.
REAL	
SDL RT	An ASN.1 `REAL` type is mapped to a C `double`. Constraints set on the type are ignored.
sdl	An ASN.1 `REAL` is mapped to a SDL `Real`. If constraints are set on the ASN.1 type, they are recreated in SDL via a `SYNTYPE` based on `Real`.
TTCN-3	An ASN.1 `REAL` is mapped to a TTCN-3 `float`. If constraints are set on the ASN.1 type, they are recreated via a TTCN-3 sub-type of `float`.

BIT STRING	
	An ASN.1 `BIT STRING` is mapped to a `RTDS_BitString`, which is defined as a `struct` with the following fields: • `__string` is an array of `unsigned char` containing the actual bits in the string. • `__length` is an `unsigned int` giving the length of the bit string in bits. The definition of `RTDS_BitString` is in the file `RTDS_String.h`, automatically included in the generated code for all projects. If the ASN.1 type defines any size constraint for the type, it is actually ignored in the generated code. The capacity for the `__string` array will be a constant (today, 64 bytes, i.e 512 bits). Convenience macros are provided to handle bit string comparisons and concatenation: • `RTDS_BIN_OCTET_STRING_CMP` allows to compare two bit strings. Parameters should be the first string, the second string and the constant 8. It returns an integer having the same meaning as the standard C functions `memcmp` or `strcmp`. • `RTDS_BIN_STRING_CAT` allows to concatenate two bit strings into a third one. Its parameters are the bit string where the result must be put, then the first and second bit strings to concatenate.
	An ASN.1 `BIT STRING` type is mapped to a SDL `BITSTRING` type. If the ASN.1 type has constraints, they are recreated in SDL via a `SYNTYPE` based on a `BITSTRING`.
	An ASN.1 `BIT STRING` type is mapped to a TTCN-3 `bitstring` type. If the ASN.1 type has constraints, they are recreated via a TTCN-3 subtype of `bitstring`.

OCTET STRING	
	An ASN.1 `OCTET STRING` is mapped to a `RTDS_OctetString`, which is defined as a `struct` with the following fields: • `__string` is an array of `unsigned char` containing the actual bytes in the string. • `__length` is an `unsigned int` giving the length of the octet string in bytes. The definition of `RTDS_OctetString` is in the file `RTDS_String.h`, automatically included in the generated code for all projects. If the ASN.1 type defines any size constraint for the type, it is actually ignored in the generated code. The capacity for the `__string` array will be a constant (today, 256 bytes). Convenience macros are provided to handle octet string comparisons and concatenation: • `RTDS_BIN_OCTET_STRING_CMP` allows to compare two octet strings. Parameters should be the first string, the second string and the constant 1. It returns an integer having the same meaning as the standard C functions `memcmp` or `strcmp`. • `RTDS_OCTET_STRING_CAT` allows to concatenate two octet strings into a third one. Its parameters are the octet string where the result must be put, then the first and second octet strings to concatenate.
	An ASN.1 `OCTET STRING` type is mapped to a SDL `OCTETSTRING` type. If the ASN.1 type has constraints, they are recreated in SDL via a `SYNTYPE` based on an `OCTETSTRING`.
	An ASN.1 `OCTET STRING` type is mapped to a TTCN-3 `octetstring` type. If the ASN.1 type has constraints, they are recreated via a TTCN-3 subtype of `octetstring`.
IA5String	
	An ASN.1 `IA5String` is mapped to a `RTDS_String`, which is an array of characters. Today, the size constraints set on the ASN.1 type are not taken into account and the capacity of the array is a constant (256 characters in the current version). The type `RTDS_String` is defined in the file `RTDS_String.h`, along with functions allowing to manipulate strings: • `RTDS_StringAssign(s1, s2)` copies `s2` into `s1` in a safe way; • `RTDS_StringCat(s1, s2, s3)` concatenates `s2` and `s3` and puts the result in `s1`, taking care of buffer overflows; • `RTDS_SubString(s1, s2, i, len)` extracts the substring of `s2` starting at index `i` and with the length `len` and puts it in `s1`; • `RTDS_StringReplace(s1, s2, i, len, s3)` replaces the portion of `s2` starting at index `i` and with the length `len` with `s3`, and puts the results in `s1`.

	An ASN.1 `IA5String` is mapped to a SDL `CharString`. If there are constraints on the ASN.1 type, they are recreated via a SDL `SYNTYPE` based on a `CharString`.
	An ASN.1 `IA5String` is mapped to a TTCN-3 `charstring`. If there are constraints on the ASN.1 type, they are recreated via a TTCN-3 subtype `charstring`.
SEQUENCE / SET	
	An ASN.1 `SEQUENCE` or `SET` type is mapped to a C `struct` type with the same fields.
	An ASN.1 `SEQUENCE` or `SET` type is mapped to a SDL `NEWTYPE STRUCT` with the same fields.
	An ASN.1 `SEQUENCE` or `SET` type is mapped to a TTCN-3 `record` type with the same fields.
CHOICE	
	An ASN.1 `CHOICE` type is mapped to a set of 3 C types: • An `enum` type defining the constants corresponding to the field names in the `CHOICE`. This type is named `t_<type name>`. • A `union` type for the alternative, with the same fields as the `CHOICE` type. This type is named `_<type name>_choice`. • An overall `struct` type, having the same name as the ASN.1 type, and containing the fields `present`, with the `enum` type `t_<type name>`, and `__value`, with the `union` type `_<type name>_choice`. This definition is derived from the SDL notion of `CHOICE` type, where the selected alternative is known via a pseudo-field named `present`, containing the field name for the chosen alternative.
	An ASN.1 `CHOICE` type is mapped to a SDL `CHOICE` type with the same fields.
	An ASN.1 `CHOICE` type is mapped to a TTCN-3 `union` type with the same fields.

SEQUENCE OF	
	An ASN.1 `SEQUENCE OF` type is mapped to a C `struct` type with the following fields: • `elements`, which is an array of values with the type of an element in the `SEQUENCE OF`; • `length`, which is an `unsigned int` giving the actual number of elements in the array. If a constraint is defined on the size of the `SEQUENCE OF`, it is used to define the size of the `elements` array. If there isn't any constraint, the constant `RTDS_MAX_STRING` is used, which is usually 256 unless explicitly defined with another value. A convenience macro is defined in `RTDS_String.h`, allowing to concatenate two sequences with any types and copy the result to a third one. It is called `RTDS_SEQUENCE_CONCAT` and takes as parameters the sequence that will receive the result, then the 2 sequences to concatenate. The file `RTDS_String.h` is automatically included in the generated code for all projects.
	An ASN.1 `SEQUENCE OF` type is mapped to a SDL `NEWTYPE` using the `String(…)` generator. If the ASN.1 type has a size constraint, it is translated to the same constraint on the SDL type (`CONSTANTS` clause with a `SIZE(…)` constraint).
	An ASN.1 `SEQUENCE OF` type is mapped to a TTCN-3 `record of` type. If the ASN.1 type has a size constraint, it is translated to the same constraint on the TTCN-3 type (`length(…)` constraint).

SET OF

An ASN.1 `SET OF` type is translated to a C type that is not intended to be used directly. To manipulate a `SET OF` with type *T* in C code, the following macros and functions should be used:

- `RTDS_SET_OF_INIT(s)`: Initializes a set. This macro *must* be called before the set variable is used, or the behavior of the other macros is undefined.
- `RTDS_SET_OF_T_COPY(s1, s2)`: Copies the set `s2` into the set `s1`.
- `RTDS_SET_OF_T_INCL(s1, s2, elt)`: Creates a copy of the set `s2`, adds the element `elt` to it and puts the result in `s1`.
- `RTDS_SET_OF_T_DEL(s1, s2, elt)`: Creates a copy of the set `s2`, removes the element `elt` from it and puts the result in `s1`. If `elt` is not in `s2`, `s2` is just copied to `s1`.
- `RTDS_SET_OF_LENGTH(s)`: Returns the number of elements in the set `s` as an `unsigned int`.
- `RTDS_SET_OF_T_TAKE(s)`: Returns a random element from the set `s`. The set is not modified.
- `RTDS_setOf_T_cmp(op, s1, s2)`: Compares two sets for equality and/or inclusion. The operator `op` has the type `RTDS_setOfCompareOperator`, defined in `RTDS_Set.h`, which can be `RTDS_SET_OF_CMP_` followed by:
  - `EQ` / `NE` to test for set equality / inequality;
  - `LT` / `LE` to test if `s1` is a strict / non-strict subset of `s2`;
  - `GT` / `GE` to test if `s1` is a strict / non-strict superset of `s2`.
- `RTDS_setOf_T_in(elt, s)`: Tests for the inclusion of `elt` in the set `s`.
- `RTDS_SET_OF_T_FIRST(s, elt)`: Allows to initialize a iteration on the elements of the set `s`. After the call, `elt` will be set to one of the element in the set. To iterate over the other ones, use `RTDS_SET_OF_T_NEXT`. Returns a boolean value indicating if `elt` was actually set. If it weren't, the set is empty.
- `RTDS_SET_OF_T_NEXT(s, elt)`: Allows to iterate over all the elements in the set `s`. Must be called only after a call to `RTDS_SET_OF_T_FIRST`. Sets the `elt` to the next element in the set if possible, and returns a boolean value indicating if `elt` was actually set. If not, it means there are no more elements to iterate over. Note that the macros `RTDS_SET_OF_T_FIRST` and `RTDS_SET_OF_T_NEXT` are not thread-safe and shouldn't be used on the same set in parallel.

In all macros above, `s1` and `s2` must be sets and have the exact same type, and `elt` must have the type of an element in the set(s), or the behavior is undefined.

These macros and functions are defined either in `RTDS_Set.h` or in the generated file containing the type itself. The functions are implemented in the generated file `RTDS_comp_functions.c`.

	An ASN.1 `SET OF` type is mapped to a SDL `NEWTYPE` using the `Bag(…)` generator. If the ASN.1 type has a size constraint, it is translated to the same constraint on the SDL type (`CONSTANTS` clause with a `SIZE(…)` constraint).
	An ASN.1 `SET OF` type is mapped to a TTCN-3 `set of` type. If the ASN.1 type has a size constraint, it is translated to the same constraint on the TTCN-3 type (`length(…)` constraint).

3 - SDL reference guide

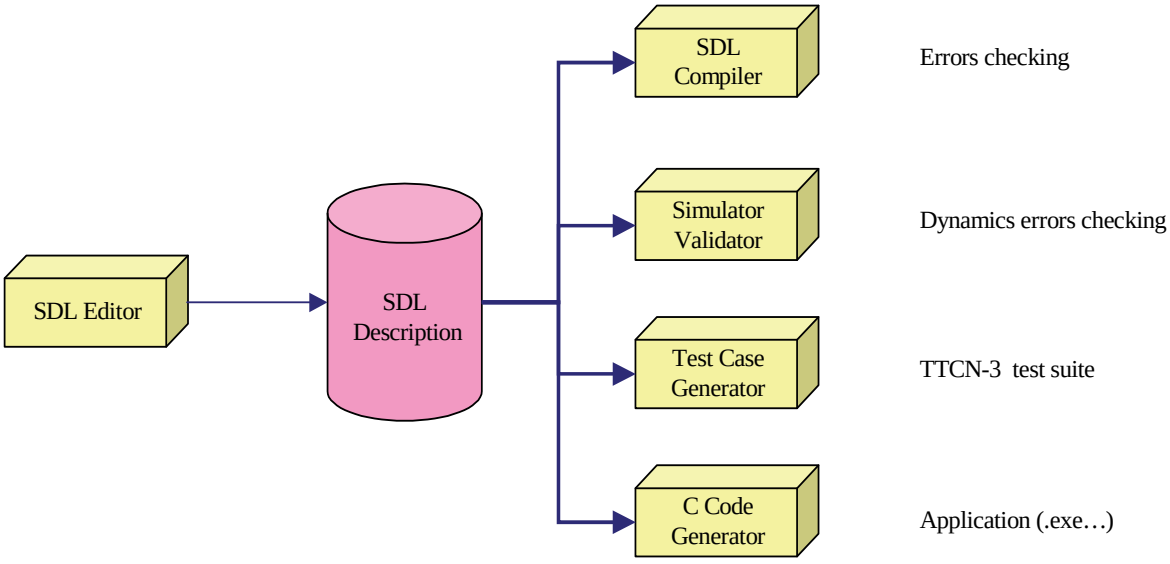
3.1 - Introduction

Specification and Description Language (SDL) is a formal language standardized by the International Telecommunication Union (ITU) as the Z.100 Recommendation. It is an object-oriented language created to describe complex, event-driven, and interactive applications involving many concurrent activities, communicating together through signals.

SDL is initially introduced to provide a precise specification and description of the telecommunications systems' behavior so that it is possible to analyze and interpret them without ambiguity. The objectives of SDL include providing a language that is easy to learn, use and interpret, while implementing a precise system specification.

3.1.1 Benefits

The advantage of SDL language is the ability to produce a complete system specification. It describes the structure, behavior, and data of distributed communication systems accurately, while eliminating ambiguities and thus, guarantees system integrity. With the presence of a complete semantics, the SDL description can be rapidly checked and debugged using compilers, as well as simulators during the modeling process (or even upon its completion). This enables a very fast correction model. In addition, SDL gives users a choice of two different syntactic forms to use when representing a system; graphical and textual phrase representation, which makes it an advantageous language.



SDL Description

3.1.2 Relation between SDL and other languages

SDL has the ability to interface with other languages, as well as other high-level notations frequently used for system analysis. This includes Unified Modeling Language (UML) object models and Message Sequence Chart (MSC) use-cases, as well as Abstract Syntax

Notation One (ASN.1) for data-type definitions. Besides that, tests validations can also be performed by using Testing and Test Control Notation version 3 (TTCN-3).

3.1.3 SDL Components

A set of extended Finite State Machines (FSM), running in parallel, represents the fundamental theoretical model of an SDL system. Various entities are independent of each other and they communicate with predefined signals.

The following components are provided to form an SDL description:

- **Structure**: Agents and procedure hierarchy
- **Communication**: Channels, connection, signals and signal list
- **Behavior**: Processes
- **Data**: Abstract Data Types (ADT)
- **Inheritance**: Relations and specialization description

3.2 - Structure

A structure is composed of agents. The term agent is used to denote a **system**, **block** or **process** that contains one or more extended finite state machines.

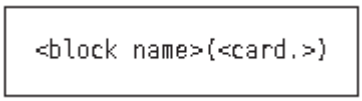
3.2.1 System

The overall design of an SDL architecture is call a system. Everything that is outside the system is called the **environment**. An SDL description must contain at least a system. There is no specific graphical representation for a system in SDL. However, a block representation can be used if needed since a system is the outermost block.

3.2.2 Block(s)

A block is a structuring element that does not imply any physical implementation on the target. A system must have at least a block. A block can contain other blocks and processes, which allows large systems definition.

A block is represented by a solid rectangle:



Block symbol

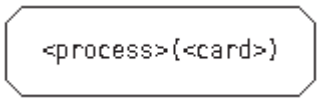
Example of a block declaration

Syntax: `<block name>(<card,>)`

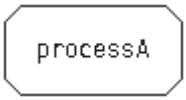
- `<block name>`: name of the block
- `<card>`: cardinality

3.2.3 Process(es)

To describe a block's functionality, processes are inserted. A process contains the code to be executed in the form of an FSM. It may contain other processes. A process is represented by a rectangle with cut corners:



Process symbol



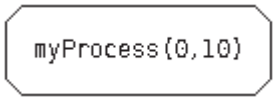
Example of a process declaration

Syntax: `<process>(<card>)`

- `<process>`: process name
- `<card>`: cardinality

It is possible to have more than an instance of a particular process running in parallel independently. The number of instances presents during the start-up and the maximum number of instances running are declared between parenthesis after the name of the process. If these values are not defined, the number of instances at start-up is 1 by default and infinite for the maximum number of instances.

Below is an example of a process that has no instance at start-up and has a maximum of 10 instances:



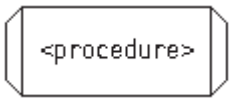
Example of a process definition

Every process contains four implicit variables, which are used to output a signal to a particular process. They are:

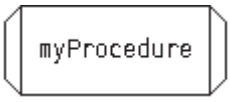
- `SELF`: Pid of the current instance
- `SENDER`: Pid of the instance that sent the last input signal
- `PARENT`: Pid of the instance that created the current instance
- `OFFSPRING`: Pid of the last instance created

3.2.3.1 Procedures

The emplacement of an SDL procedure may be anywhere in a diagram: system, block or process. It is usually not connected to the architecture. However, since it can output messages, a **channel** can be connected to a procedure for informational purpose. A procedure may return a value, and it may also contain nested procedure declarations. Moreover, procedures may also contain an FSM and procedures calls may be recursive. Below is the procedure declaration symbol in SDL:



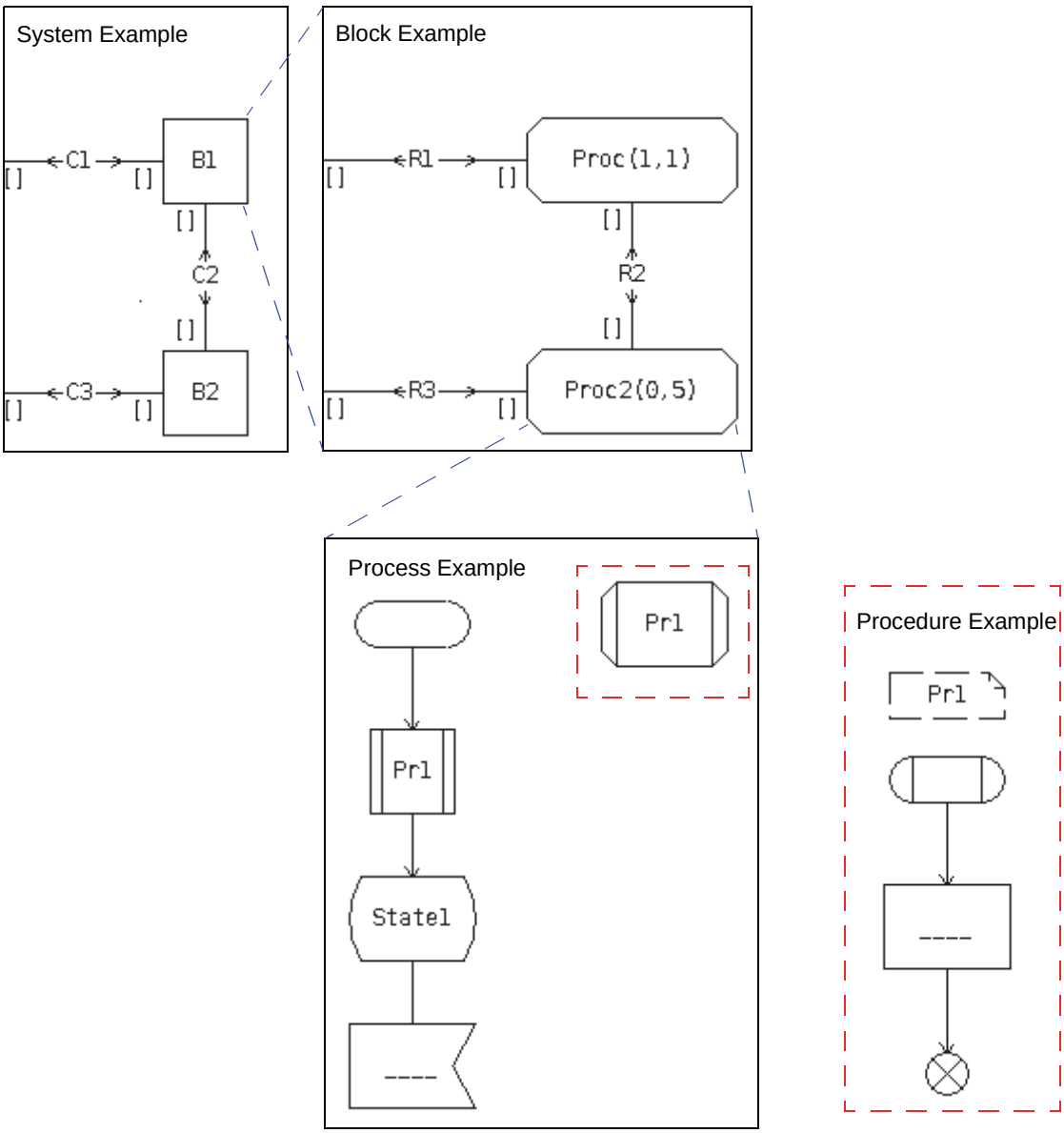
Procedure declaration symbol



Example of a process declaration

Syntax: `<procedure>`, procedure name

Relation between agents and procedures in a system:



SDL Structure

It is possible to call a procedure implemented in a different language such as C. To do so the procedure must be declared as external in a declaration symbol.

```
PROCEDURE myExternalProcedure(IN x Integer, IN y Integer)
-> MyStructType EXTERNAL;
```

3.3 - Communication

The communication in SDL is based on:

- Signal
- Signal list definition

- Channel

3.3.1 Signal

A signal instance is a flow of information between agents. It is an instantiation of a signal type defined by a signal definition. A signal instance can be sent by either the environment or an agent and is always directed to either an agent or the environment. A signal has a name and a parameter that is basically a pointer to some data. Below is an example of a signals definition:

```
SIGNAL  signal1, signal2, signal3;
SIGNAL  signal4(parameter);
```

Signals definition

3.3.2 Signal list definition

A signal list is used to gather all possible signals that run through a channel. In order to indicate a signal list in a list of signals that go through a channel, the signal list is surrounded by parenthesis. Possible signals that can be received by/transmitted to a process/environment are listed between brackets, separated by commas. Below is an example of a signal list:

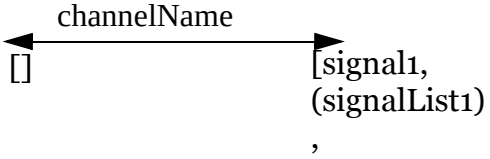
[signal1,(signalList1),signal2]

Signals definition with signal list

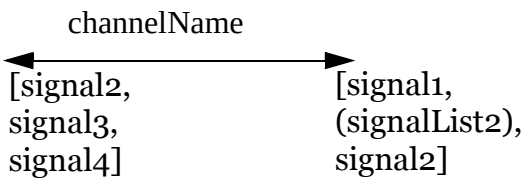
3.3.3 Channel

SDL is event-driven, which mean that communication is based on signals transmission/ reception. A channel represents a transportation path for signals. Signals are present at the destination endpoint of a channel and they pass through it, which connect agents and end up in the processes' implicit queues. Represented by one or two-way arrow(s), a channel can be considered as one or two independent unidirectional channel paths, between two agents or between an agent and its environment. A channel name is written next to the arrow without any specific parameter:

<u>*One independent channel path:*</u>

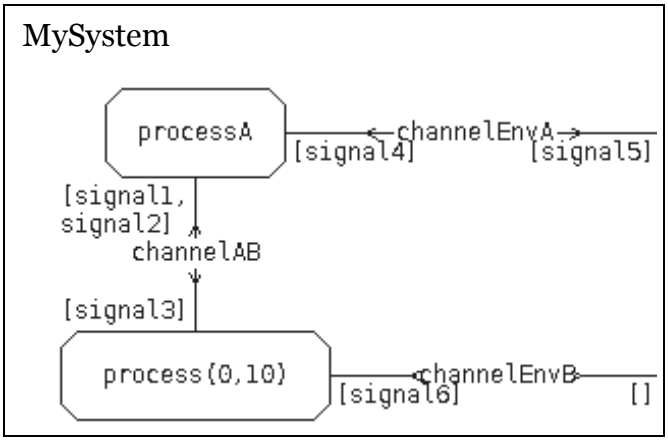


<u>*Two independent channel paths:*</u>



Channel symbol

An example of channels representation between two processes and the environment in a system:



Example of channels representation

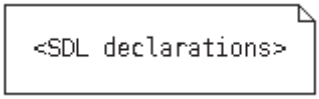
Note:

Same signal can be listed in both directions. If in a direction, there are no signals sent/ received, the signal list definition is left blank.

3.4 - Behavior

An SDL system/block hierarchy is unfortunately only a static description of the system structure. In order to control a system, the dynamic behavior of the system is described in the processes through extended FSMs, communicating with signals. A process state determines what behavior the process will have upon reception of a specific action. A **transition** is the code between two states. The SDL symbols, as well as SDL expressions that are frequently used to describe a system's behavior are listed in this section.

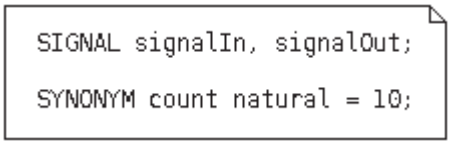
3.4.1 Text

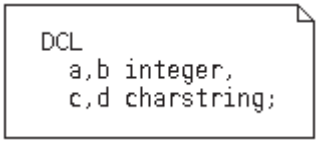
SDL Symbol	Description
	A text is used in any diagram. Its content is usually SDL declarations.

Syntax: `<SDL declarations>`, SDL declarations.

Note that a comma (,) is used to separate between SDL declarations of the same type, whereas a semicolon (;) is used to separate between different types of SDL declarations.

Example:





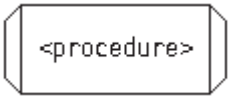
Example on text

In the first text symbol, two different types of SDL declarations are defined: `SIGNAL` and `SYNONYM`. These two types are separated by a semicolon. However, different signals `signalIn` and `signalOut` are separated by a comma.

The second example shows different elements of different data types definition. Different elements of a data type (for example integers `a` and `b`) are separated by a comma. Different data types (`integer` and `charstring`) declared in the text are also separated by a comma.

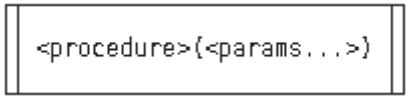
3.4.2 Procedure

3.4.2.1 Procedure declaration

SDL Symbol	Description
	A procedure can be defined in any diagram: system, block or process. The procedure's name is precised in the block. A procedure is usually not connected to the architecture. However, since it can output signals, a channel can be connected to it for informational purpose. A procedure is invoked by calling the procedure definition.

Syntax: `<procedure>`, procedure name

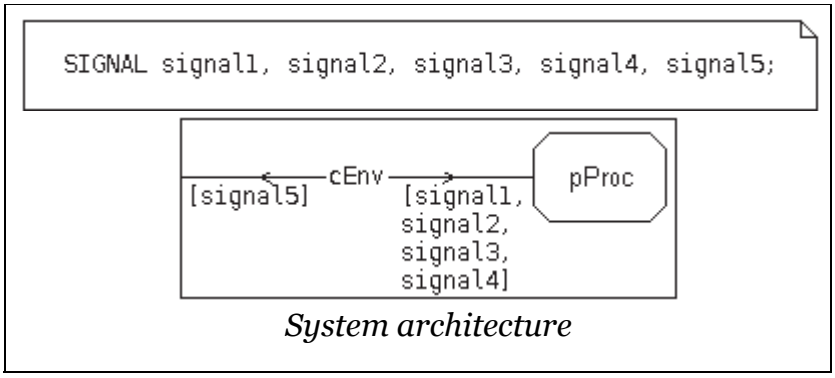
3.4.2.2 Procedure call

SDL Symbol	Description
	Used to call an SDL procedure.

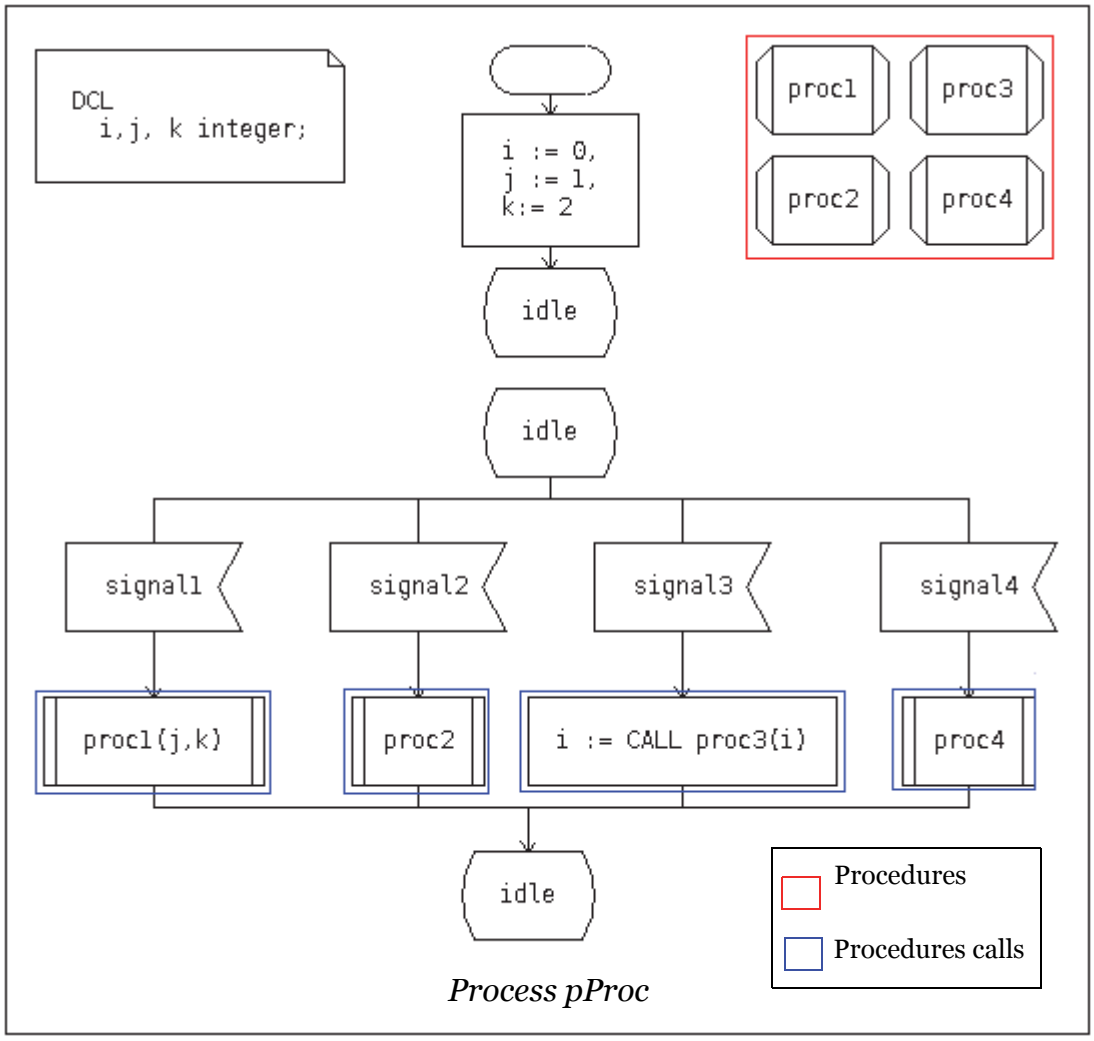
Syntax: `<return variable> = <procedure name>({<parameters>}*)`

- `<return variable>`: value returned upon procedure call
- `<procedure name>`: procedure name
- `<parameters>`: parameters

Example:



System architecture



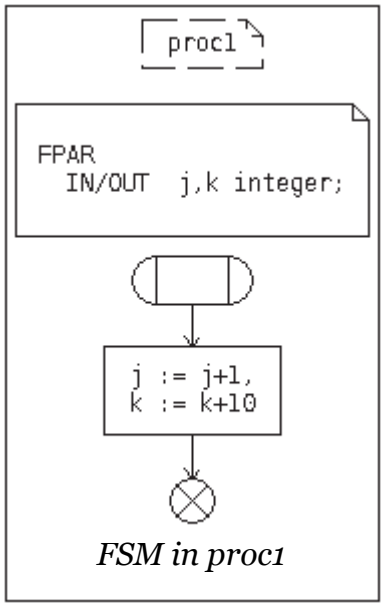
Process pProc

Example of procedures declarations and calls

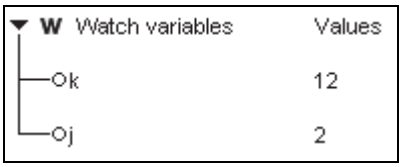
Example above shows four different ways of a procedure call in SDL:

Procedure, `proc1`

`proc1` shows the utilization of SDL syntax `IN/OUT` to modify the variables values defined in the process `pProc` directly in the procedure. In this example, a procedure call symbol is used to call `proc1` upon reception of `signal1`. It takes two parameters: `j` and `k` of type `integer`, which are the variables to be modified.



FSM in proc1



Observation in SDL

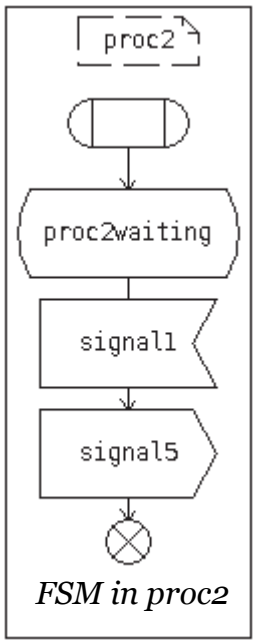
```
FPAR
IN  input parameters
IN/OUT  input/output parameters
```

The procedure proc1

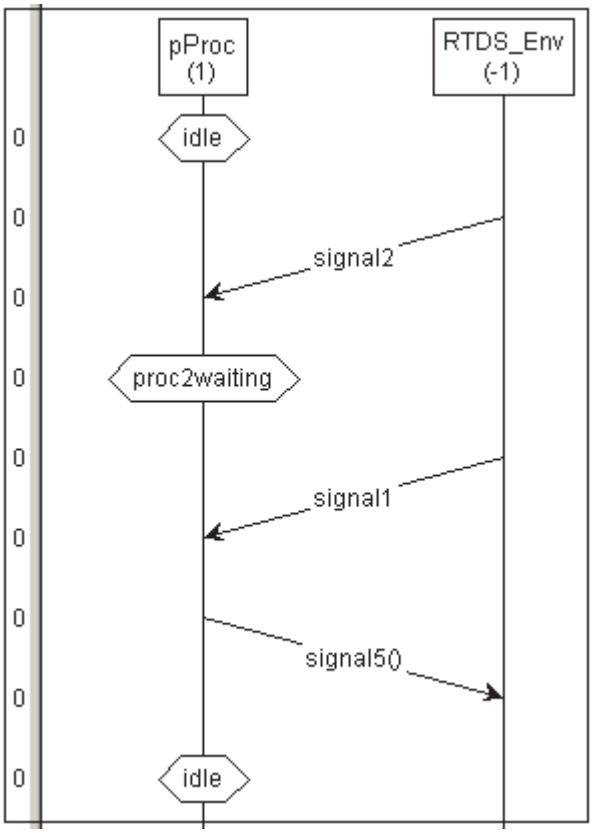
`proc1` takes two formal parameters (`FPAR`) of type `IN/OUT` which are `integers j` and `k`. Upon termination of `proc1`, `j` and `k` values are updated.

Procedure, `proc2`

`proc2` is an example of a procedure containing states.



FSM in proc2

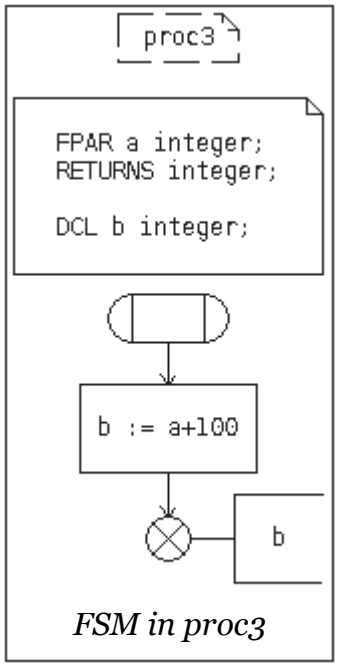


MSC Tracer

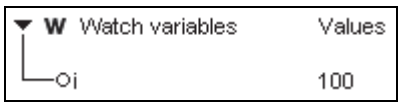
The procedure proc2

Procedure, `proc3`

`proc3` is an example of a procedure call using `CALL` keyword from a task block. The procedure returns a value in the return symbol. Before entering `proc3`, `i` equals to `0` and it is updated to `100` upon `proc3` termination.



FSM in proc3

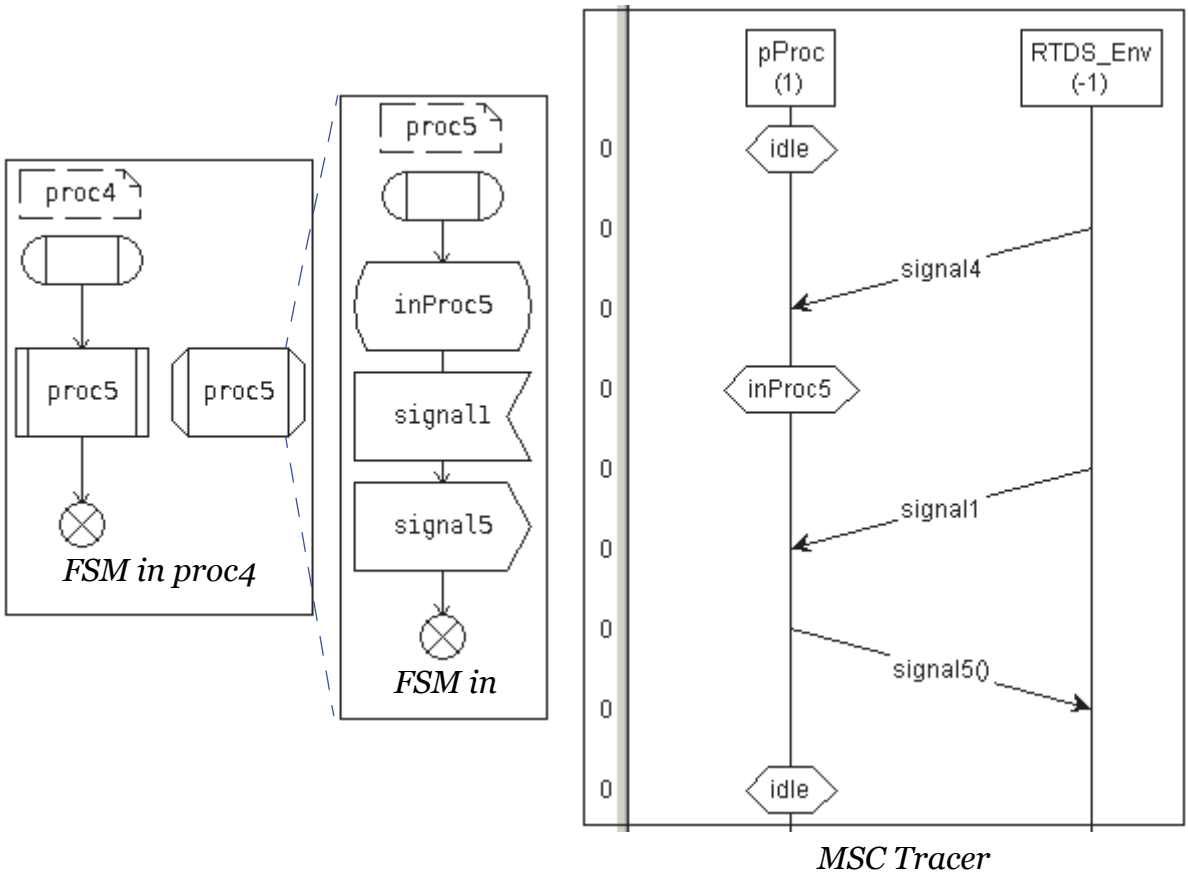


Observation in SDL Simulator, i is the value returned by b

The procedure proc3

Procedure, `proc4`

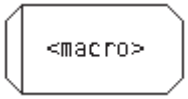
Another procedure, `proc5` is nested in `proc4`.



The procedure proc4 and nested procedure, proc5

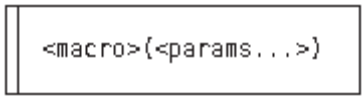
3.4.3 Macro

3.4.3.1 Macro declaration

SDL Symbol	Description
	Each macro call in an SDL specification must be preceded by the macro definition before the specification can be analyzed. A macro call is replaced by the contents of the corresponding macro during the compilation.

Syntax: `<macro>`, macro name

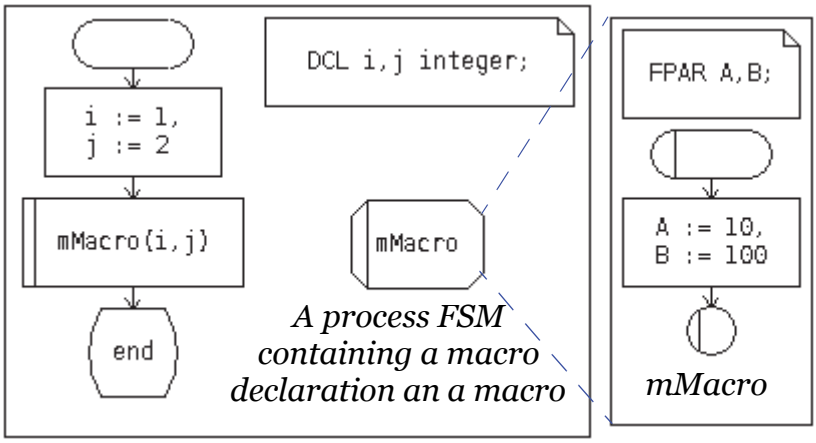
3.4.3.2 Macro call

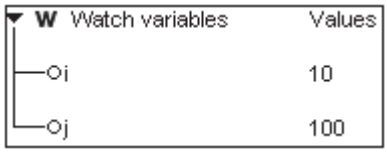
SDL Symbol	Description
	Used to call a macro by specifying its name.

Syntax: `<macro name>({<parameters>}*)`

- `<macro name>`: macro name, defined in the macro declaration
- `<parameters>`: parameters names

Example:



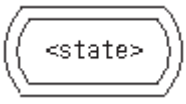


Observation in SDL

Example of a macro

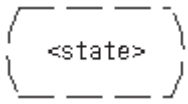
3.4.4 Composite state

3.4.4.1 Composite state definition

SDL Symbol	Description
	A composite state definition indicates that the state machine has a composite state. It is a state composed of concurrent states machines, also known as services.

Syntax: `<state>`, composite state name

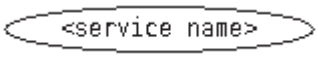
3.4.4.2 Composite state

SDL Symbol	Description
	A dashed state symbol is used to indicate that the FSM is getting into a composite state.

Syntax: `<state>`, composite state name, refers to the composite state machine definition name.

Once the FSM is in a composite state, signals are routed towards the corresponding service. In a situation where a signal can be received by both the super-FSM and the service, the super-FSM has the priority.

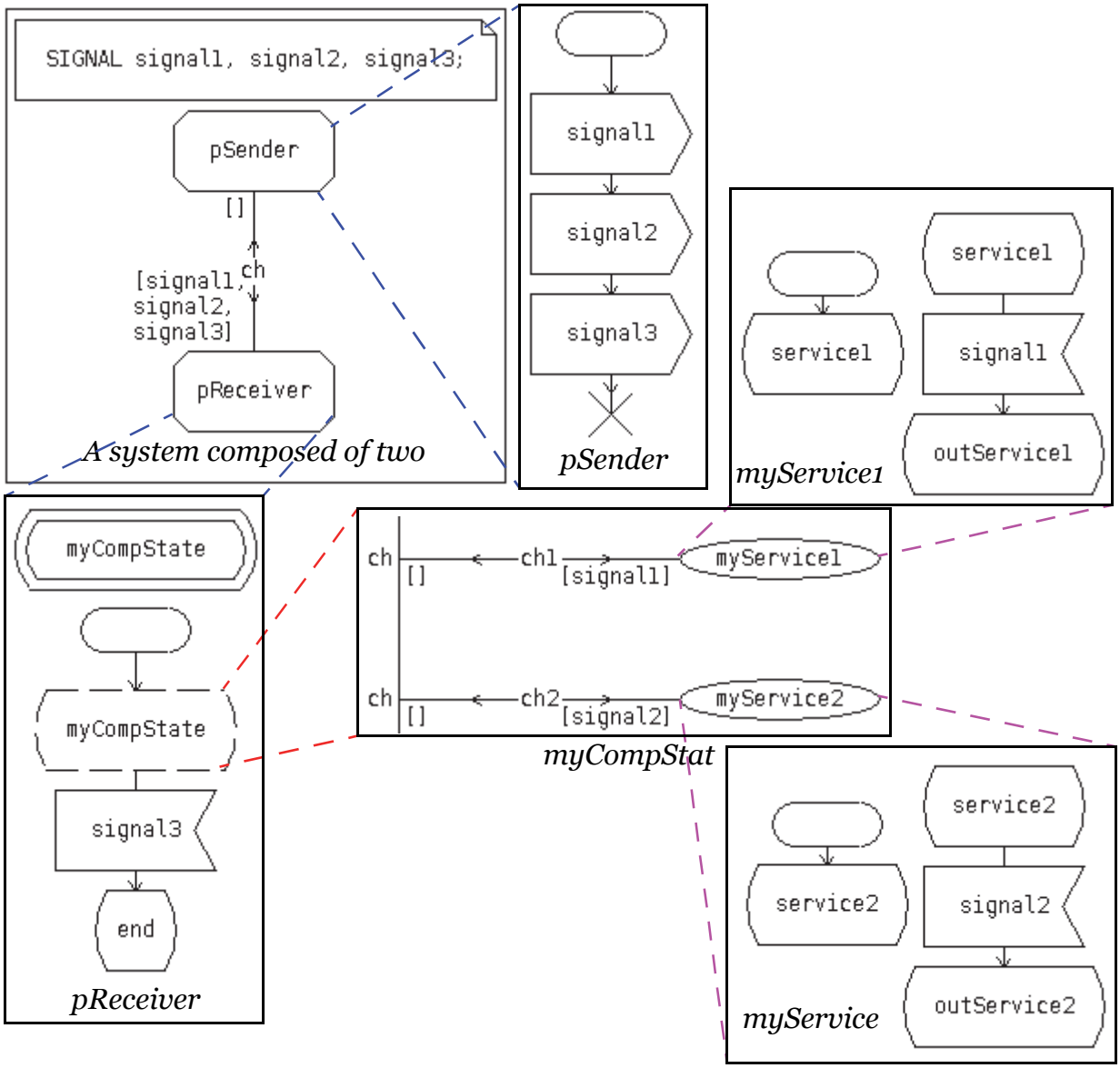
3.4.4.3 Service

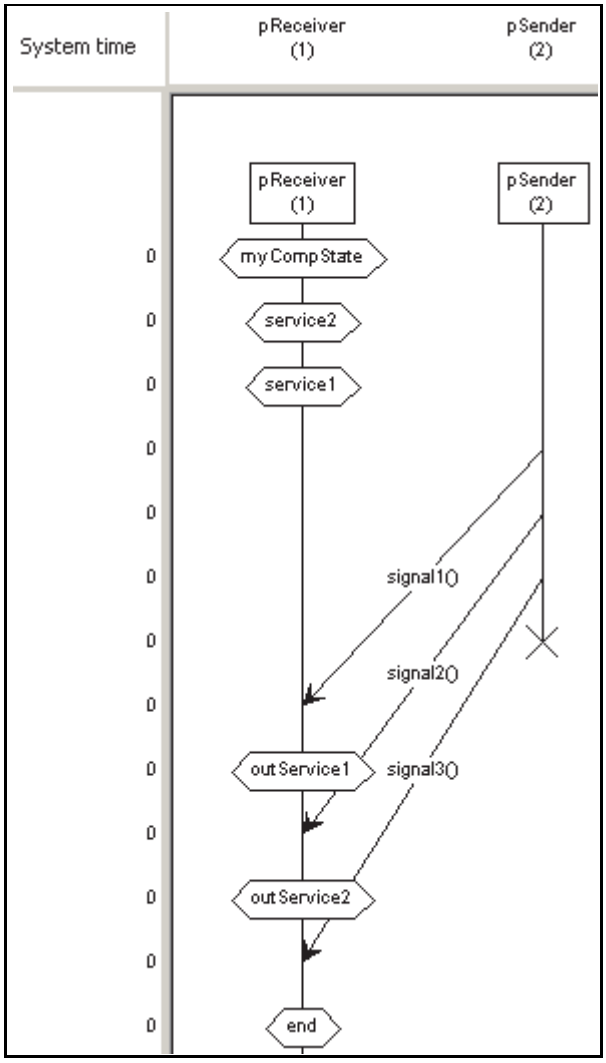
SDL Symbol	Description
	Concurrent state machine/service symbol, it is connected to channels. Each signal is routed to a specific service; the same signal can not be received by two different services.

Syntax: `<service name>`, service name

Each service handles a different subset of signals. The super-FSM also handles its own inputs. When a signal is for one of the services, the super-state does not change. However, when a signal is for the super-FSM, all services are terminated.

Example:


SIGNAL signal1, signal2, signal3;
pSender
[]
ch
[signal1,
signal2,
signal3]
pReceiver
A system composed of two
signal1
signal2
signal3
pSender
service1
service1
signal1
outService1
myService1
myCompState
myCompState
signal3
end
pReceiver
ch
[]
ch1
[signal1]
myService1
ch
[]
ch2
[signal2]
myService2
myCompStat
service2
service2
signal2
outService2
myService



MSC Tracer

Example of composite state definition and services

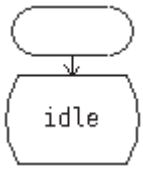
3.4.5 Start

SDL Symbol	Description
	Initial state of an FSM

This symbol marks the starting point for the execution of a process. The transition between the Start symbol and the first state of the process is called the start transition. This transition is the first thing the process will do when started. During this initialization phase, the process can not receive signals. All other symbols are allowed.

Syntax: none

Example:



Start example

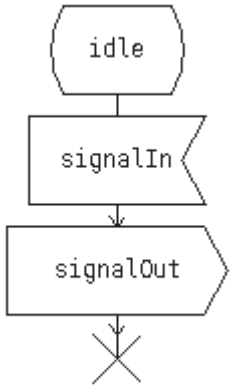
3.4.6 Stop

SDL Symbol	Description
	Process termination

A process can terminate itself with this symbol. Note that a process can only terminate itself; it can not kill another process.

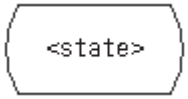
Syntax: none

Example:



Stop example

3.4.7 State

SDL Symbol	Description
	The state symbol means that the process is waiting for some input to go on.

The symbols that are allowed to be followed after a state are:

- **Input**

  The signal could be coming from an external channel, or it could be a timer signal started by the process itself.
- **Continuous signal**
  When reaching a state with continuous signals, the expressions in the continuous signals are evaluated, following the defined priorities. All continuous signals are evaluated after input signals.
- **Save**
  In a situation where the incoming signal can not be treated in the current process state, it is therefore saved until the process state changes. When the process state has changed, the saved messages are treated first (before any other signals in the queue and continuous signals).

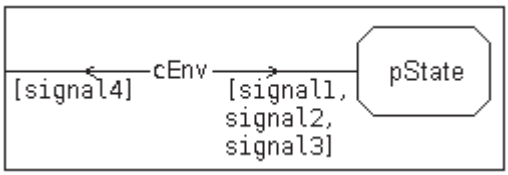
Syntax: `<state>`, state name, where any names can be defined.

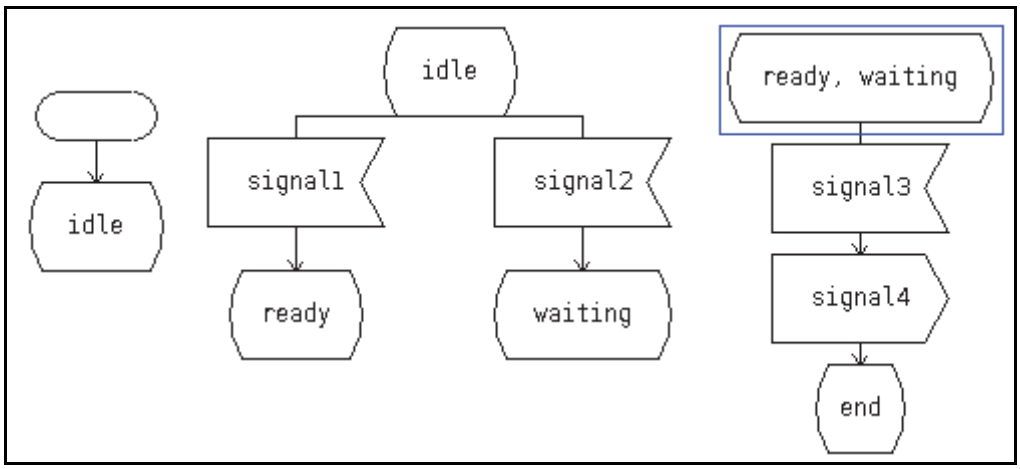
In addition, specific symbols can be introduced, which are:

Symbol	Description
(,)	Used to specify several states in a state symbol when several states have identical transitions
*	Any state in the process, this notation is used in the state symbol for adding the same transition to all the states in a process
*(,)	All the states in a process except for the states mentioned in the brackets
-	The state remains unchanged. It means that after executing the transition, the state will remain unchanged.

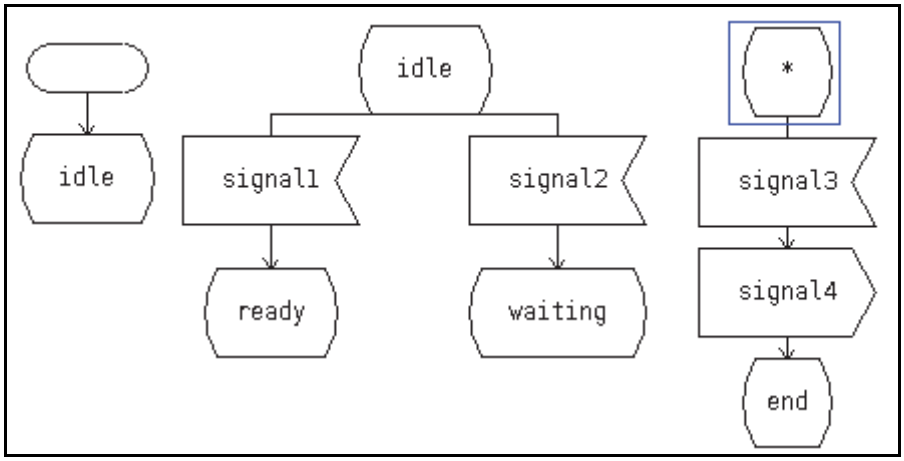
Examples:

```
SIGNAL signal1, signal2, signal3, signal4;
```

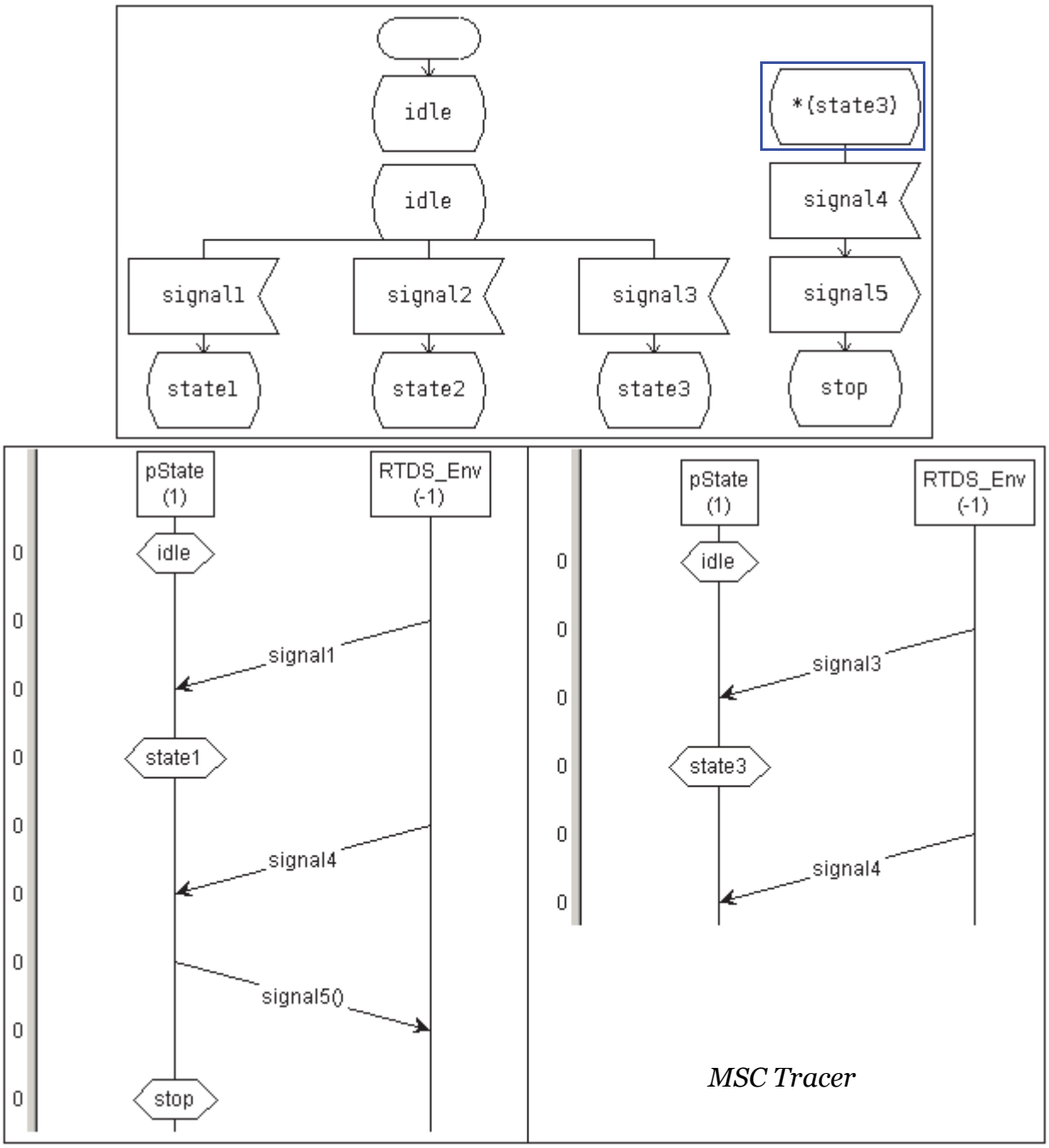




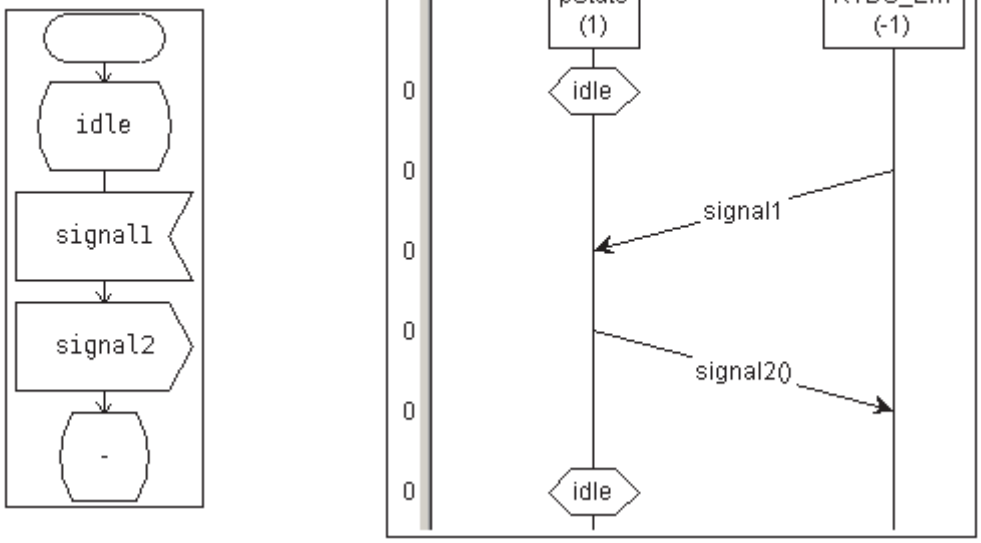
Several states in a state symbol



Any state in a process



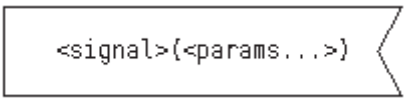
Any state except states listed in the bracket (in this example, all states except state3)



MSC Tracer

The state idle remains unchanged

3.4.8 Input

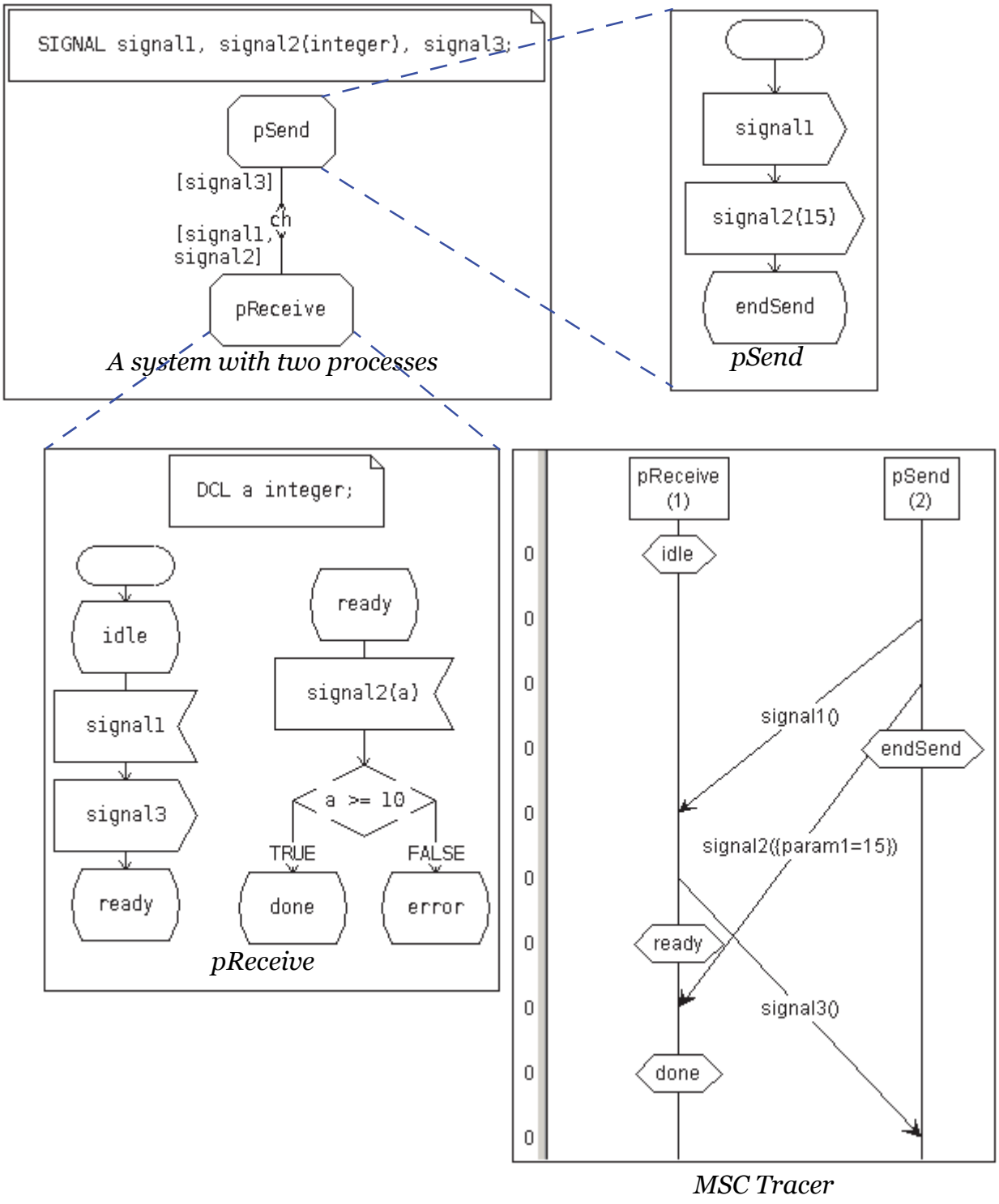
SDL Symbol	Description
	The signal input symbol in SDL, it is always followed by an SDL state.

Syntax: `<signal>(params...>]`

- `<signal>`: signal name
- `<params...>`: parameters names

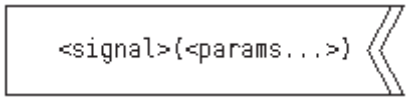
In order to receive the parameters, it is necessary to declare the variables that will be assigned to the parameters values in accordance with the signal definition.

Example:



Input signals example

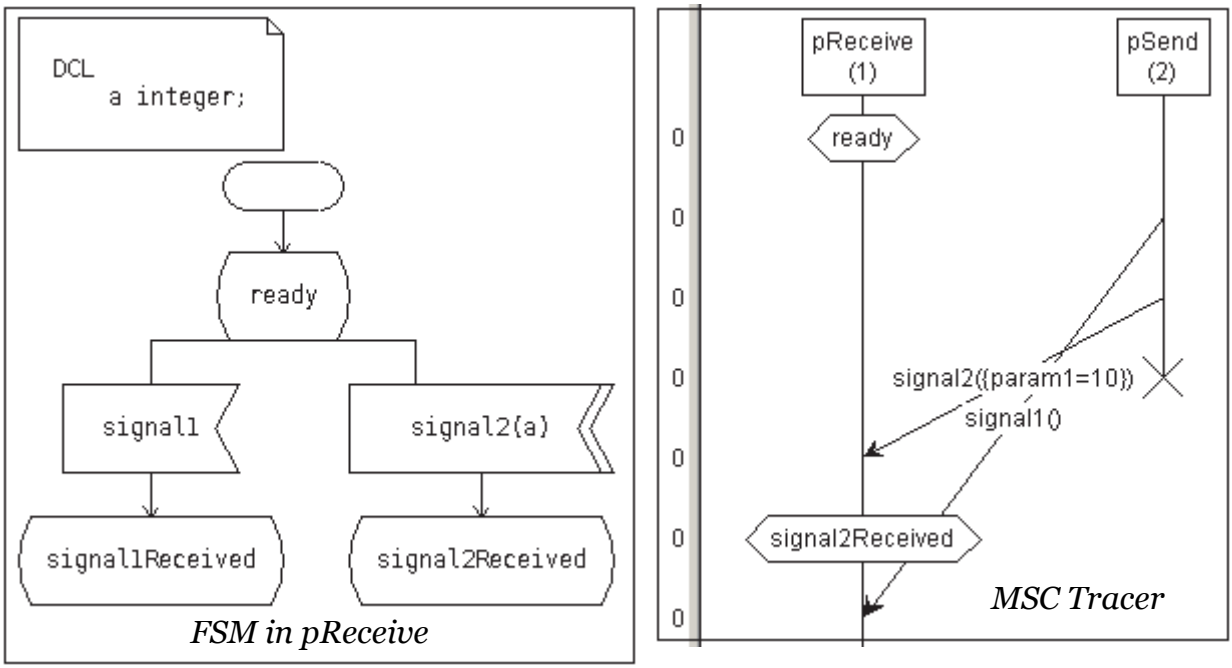
3.4.9 Priority input

SDL Symbol	Description
	In some cases, user can indicate in the FSM that reception of a signal takes priority over reception of other signals using the signal priority input symbol.

Syntax: `<signal>(<params...>)`

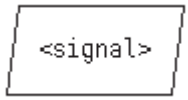
- `<signal>`: signal name
- `<params>`: parameters names

Example:



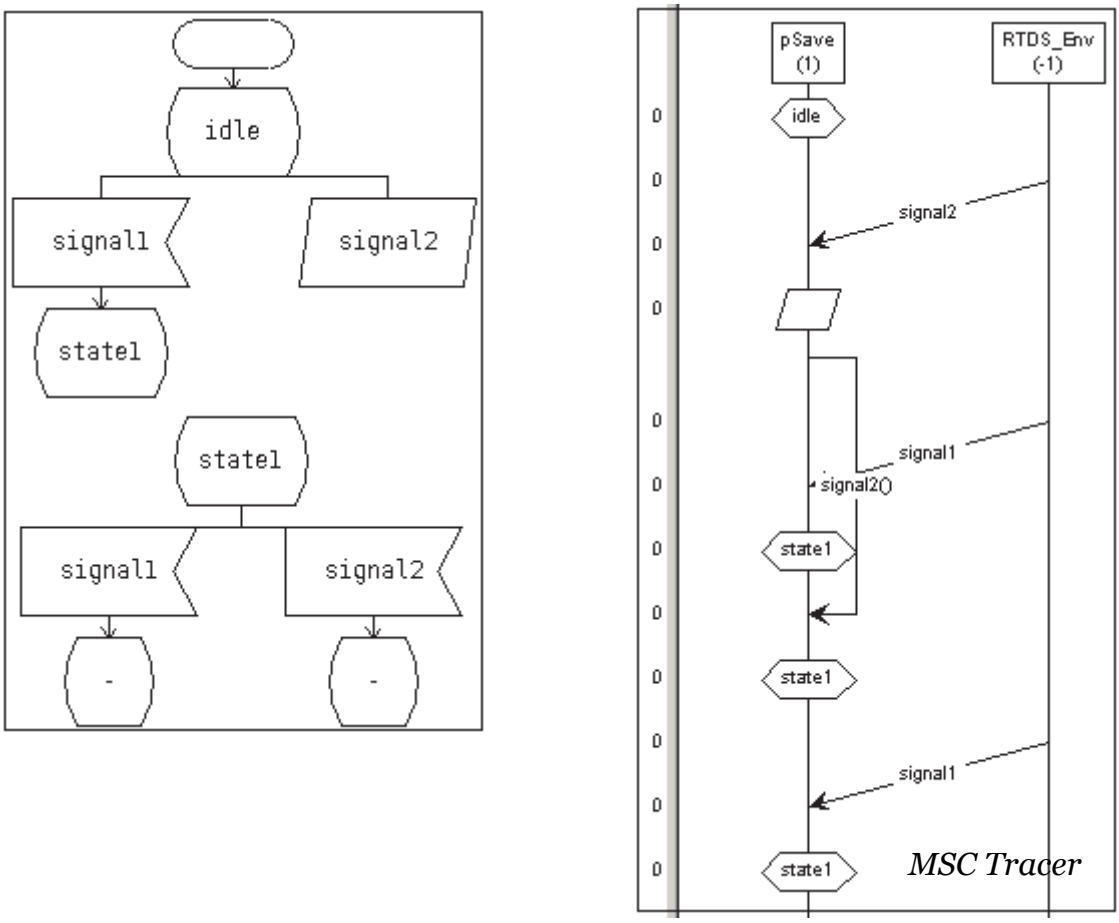
Priority input example

3.4.10 Save

SDL Symbol	Description
	In an FSM, a process may have intermediate states that can not deal with a new request until the on-going job is done. These new requests should not be lost. It must be kept until the process reaches a stable state. Therefore, save concept has been introduced to hold the input signal until it can be treated. No symbol is introduced after this symbol in an FSM.

Syntax: `<signal>`, input signal name

Example:



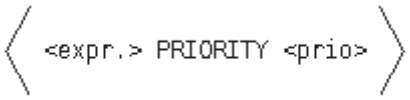
Save example

In the example above, `idle` can receive two signals: `signal1` and `signal2`:

- when `signal2` is received, it will be saved
- upon reception of `signal1`, go to `state1`

Once in `state1`, since `signal2` has been saved earlier, it has the priority over `signal1`. Thus, it will first be treated before the `signal1`.

3.4.11 Continuous signal

SDL Symbol	Description
	Used to verify the value of a variable (an if statement)

A continuous signal is an expression that is evaluated right after a process reaches a new state. A continuous signal symbol can be followed by any other symbol except:

- another continuous signal
- an input signal

Note that input signals have always priority over a continuous signal.

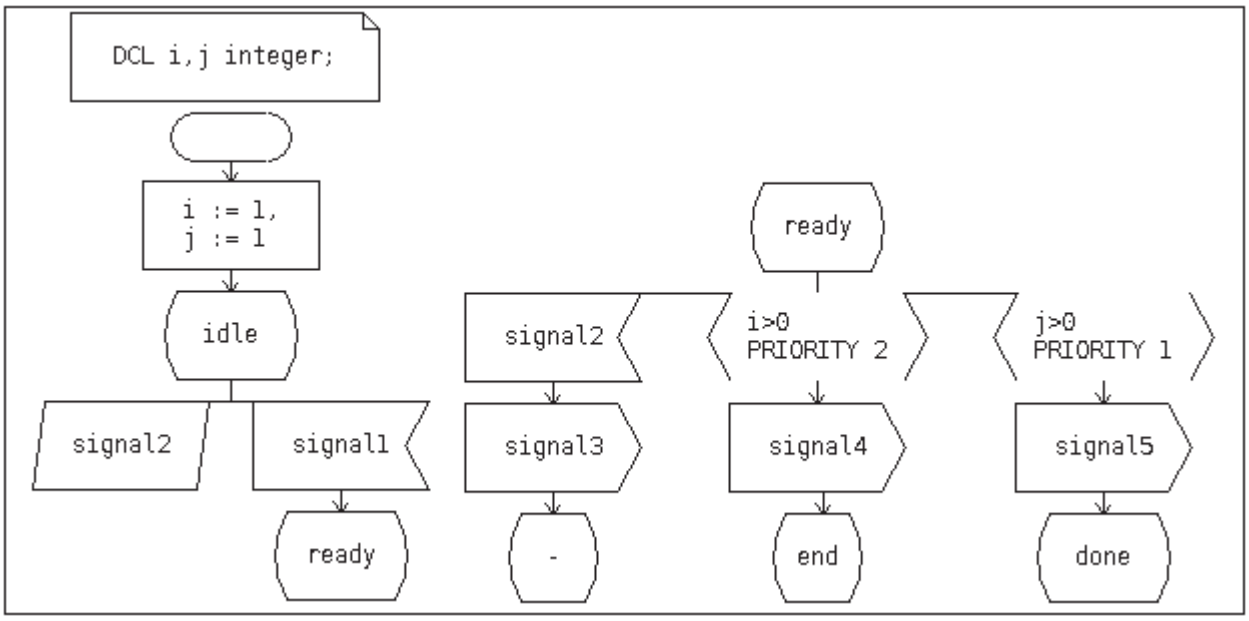
Syntax: `<expr.> PRIORITY <prio>`

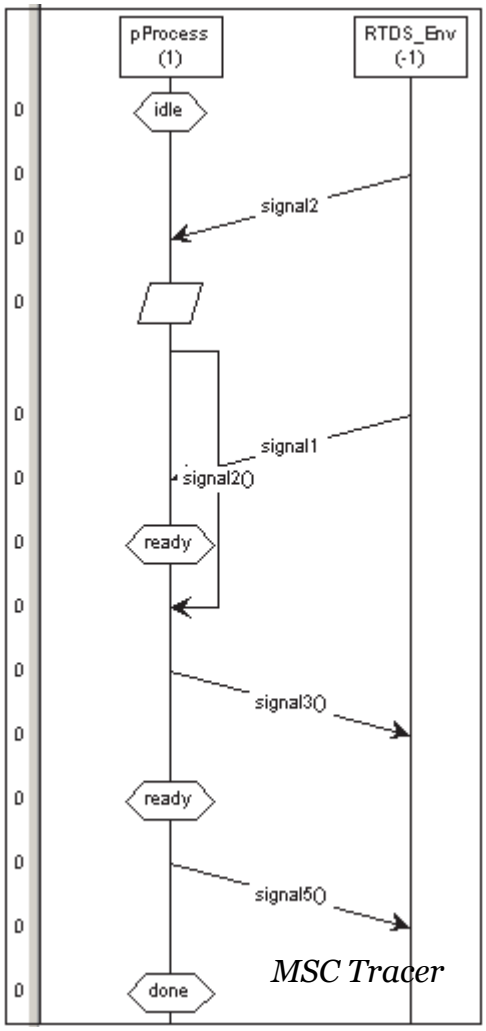
- `<expr.>`: any condition that returns a `TRUE/FALSE` expression
- `PRIORITY`: an SDL state may contain several continuous signals. Therefore, a priority level needs to be defined with this keyword.
- `<prio>`: the priority level, lower value corresponds to higher priority

Below is an example of a continuous signal. In this example, when the process is in `ready`:

- the process will firstly execute the `signal2` since it has been saved earlier.
- the expression `j>0` is evaluated since it has the highest priority. If the expression returns false, then:

- the expression `i>0` is evaluated. Else, this expression will never be evaluated.

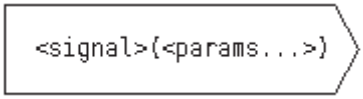




An example on continuous signal

If several continuous signals have the same priority, a non-deterministic choice is made between them.

3.4.12 Output

SDL Symbol	Description
	Used for information exchange, it puts data in the receiver's queue in an asynchronous way.

When an output signal has parameters, user defined local variables are used to assign the parameters.

Syntax: `<signal>[<params...>] TO_`

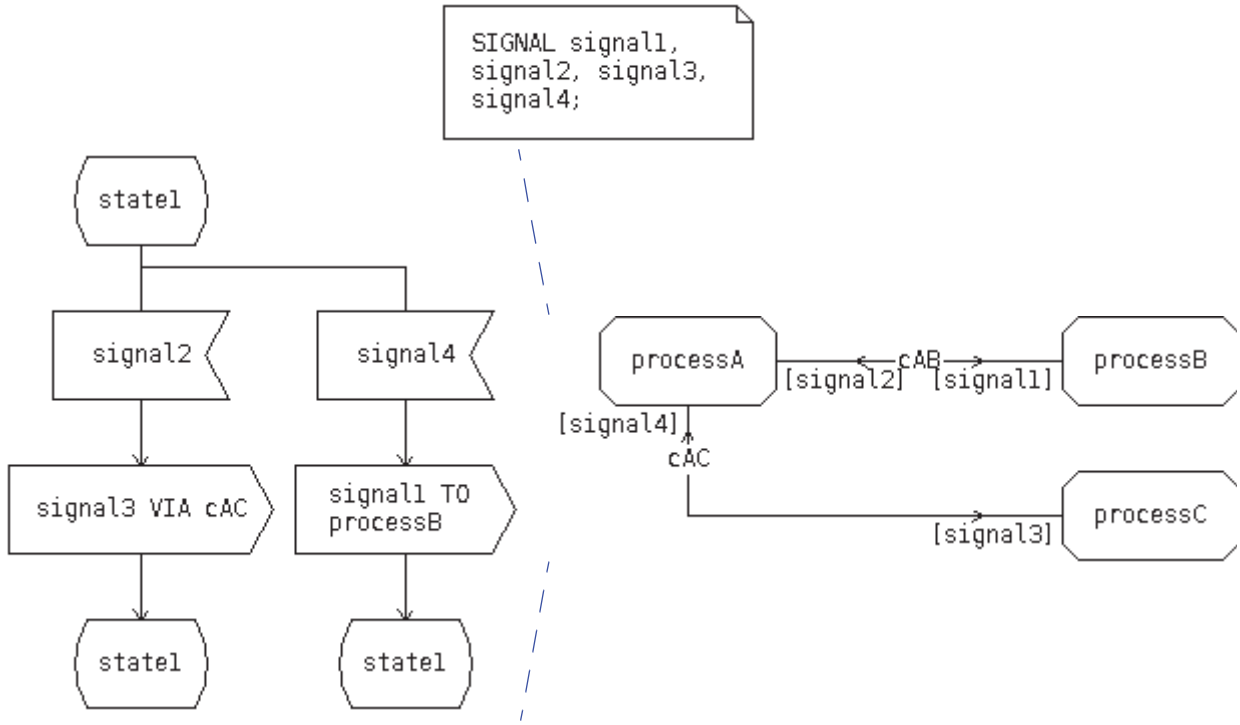
- `<signal>`: output signal name
- `<params...>`: parameters' values
- `TO_`: receiver, choices between:
  simple, where the receiver is not precised
  `VIA channel name/gate name`, the path used for signal transmission is precised
  `TO processName`, the process name where the signal is to be sent is precised
  `TO pid_variable`, the Pid of the process is precised

The various output configuration for the receiver are summarized in the table below. Note that OK means that only one process instance receives the signal output, which is generally the desired behavior in a system:

Situation	Output	Result
one process one instance	simple	OK
several processes mono-instance	simple	non-deterministic choice of one process
several processes mono-instance	VIA path	OK
several processes mono-instance	TO processName	OK
one process, several instances	simple	non-deterministic choice of one instance
one process, several instances	TO pid_variable	OK
several processes with several instances	simple	non-deterministic choice of one process, then non-deterministic choice of one instance

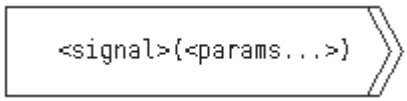
Situation	Output	Result
several processes with several instances	VIA path	non-deterministic choice of one instance
several processes with several instances	TO processName	non-deterministic choice of one instance
several processes with several instances	TO pid_variable	OK

Example:



Example of output signals for processA

3.4.13 Priority output

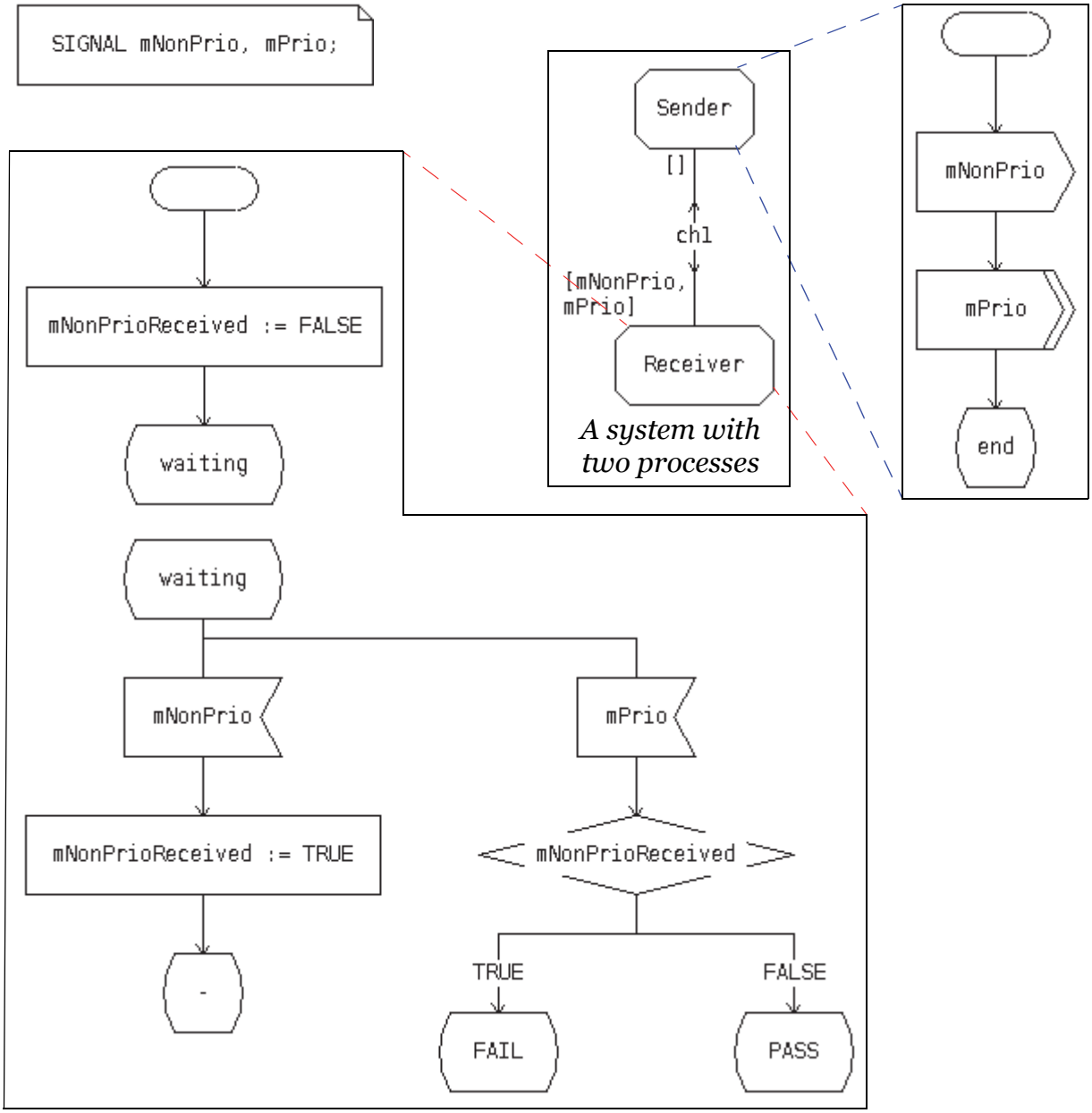
SDL Symbol	Description
	When priority output is inserted, the system prioritizes this signal to be sent to the receiver before any other output signals, even though it is sent after other signals in an FSM.

Syntax: `<signal>(<params...>)`

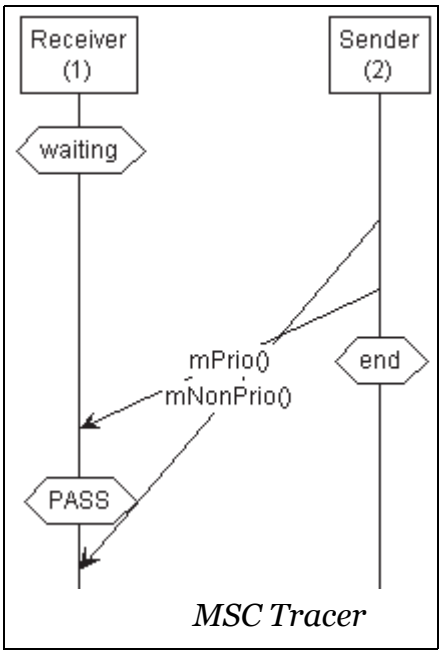
- `<signal>`: signal output name
- `<params...>`: parameters values

Example:

In the example below, the process `Sender` sends an output signal, before sending a priority output signal to the process `Receiver`. In the process `Receiver`, an FSM that validates that reception of the two signals from `Sender` is inserted.



MSC Tracer below is the result of the simulation. Observation shows that the signal `mPrio` is first received by the `Receiver`, followed by the signal `mNonPrio`, which validates the usage of priority output signal:



Example of priority output

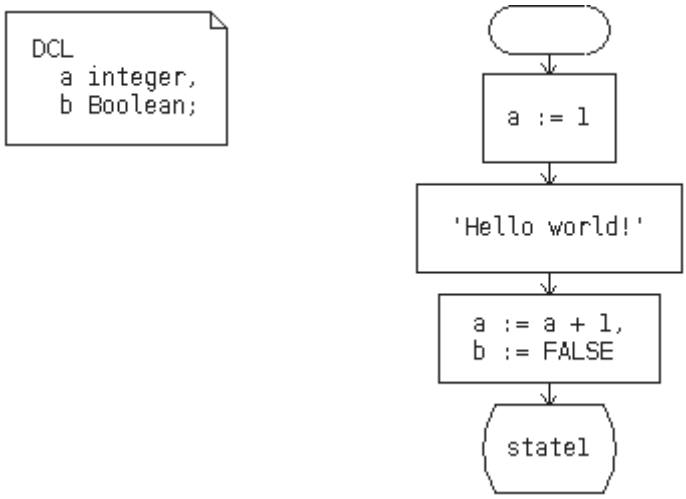
3.4.14 Task

SDL Symbol	Description
<actions...>	Also known as action symbol

Syntax: `<actions...>`, tasks to be performed written in SDL language

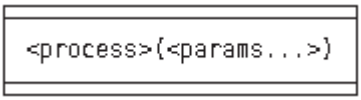
Example:

Figure below shows three tasks example. The first one performs a mathematical, the second one contains informal text (sometimes called informal task), and the third one shows how to insert several statements in a task symbol.



Tasks example

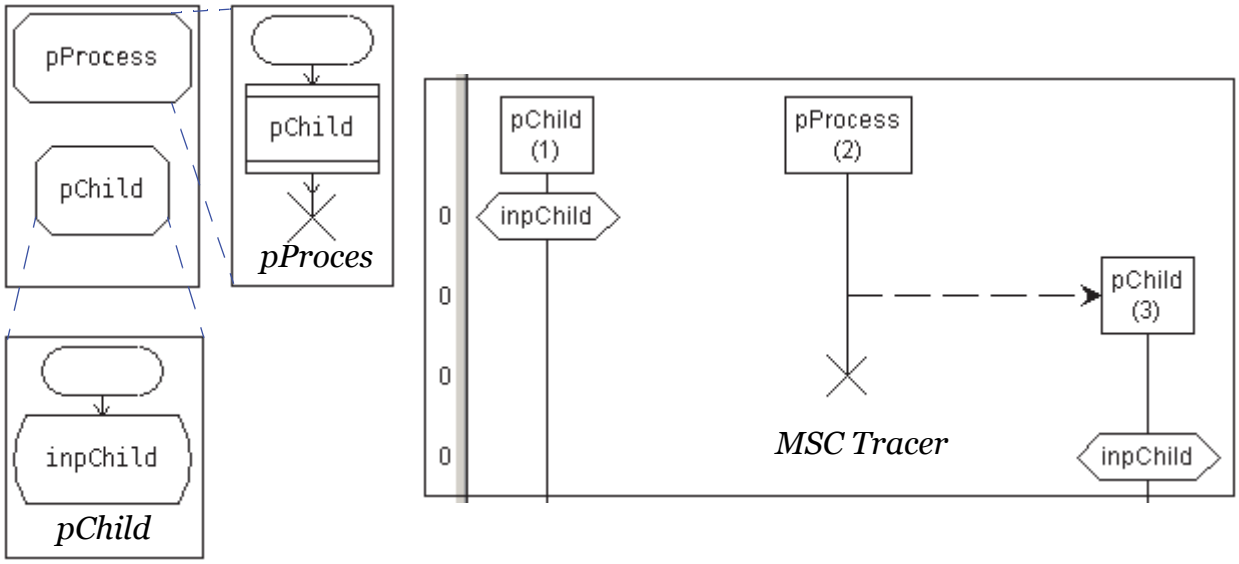
3.4.15 Process creation

SDL Symbol	Description
	SDL allows dynamic creation of process instances.

Syntax: `<process>(<params...>)`

- `<process>`: process name
- `<params...>`: parameters

Example:



Example of process creation

3.4.16 Timer

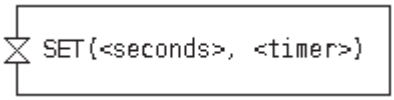
A timer in SDL is like a signal, and not a watchdog. It can be activated/deactivated using Init timer and Reset timer symbols. The syntax of a timer is:

Syntax: `TIMER <timer name>;`

SDL keywords that are used commonly with timers are:

SDL keyword	Description
`NOW`	It represents the current value of the system clock
`ACTIVE`	It is used to test whether a timer is active.

3.4.16.1 Init timer

SDL Symbol	Description
	It represents a timer instance that can be activated/deactivated.

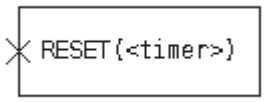
When a timer is set, a Time value is associated with the timer. It is active from the moment of setting up to the moment of consumption of the timer signal. If there is no reset or other setting of this timer before the system time reaches this Time value, a signal with the same name as the timer is put in the input port of the process.

Remark: When a timer is already activated, the action of setting the timer is equivalent to resetting the timer, immediately followed by setting it.

Syntax: `SET(<seconds>,<timer>)`

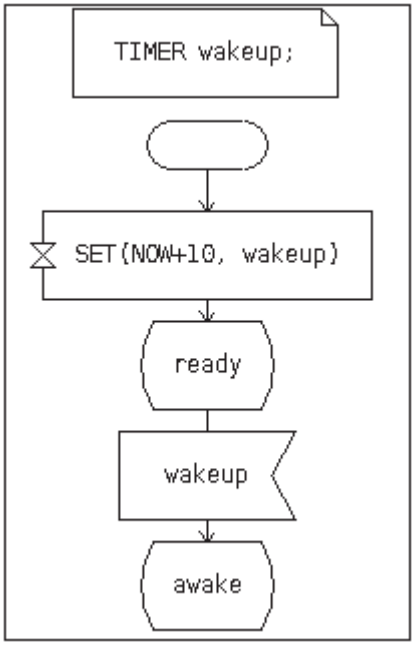
- <seconds>: timer value in seconds for simulation
- <timer>: time name

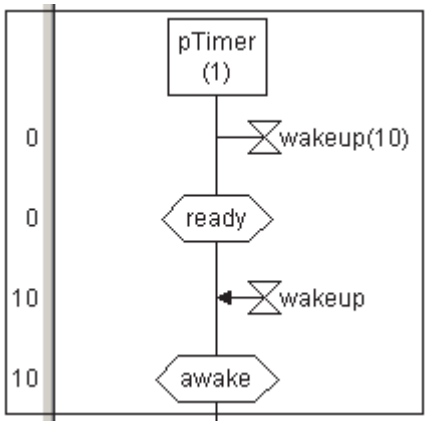
3.4.16.2 Reset timer

SDL Symbol	Description
	Used to cancel a timer. When an active timer is reset, the association with the Time value is lost. If there is a corresponding retained timer signal in the input port, this signal is removed and the timer becomes inactive.

Syntax: `RESET<timer name>`, timer name to be resetted

Example:

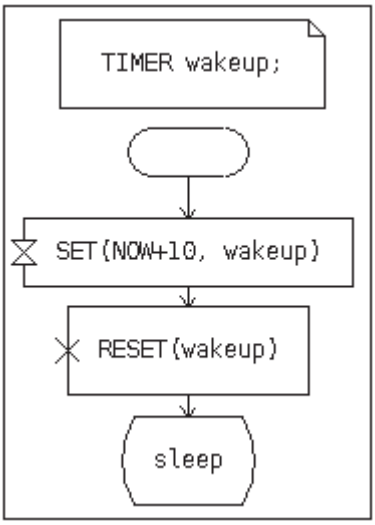


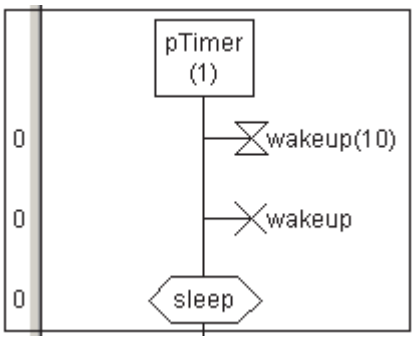


MSC Tracer

Init timer example

Example: The timer `wakeup` is resetted just after it is activated





MSC Tracer

Reset timer example

3.4.17 Decision

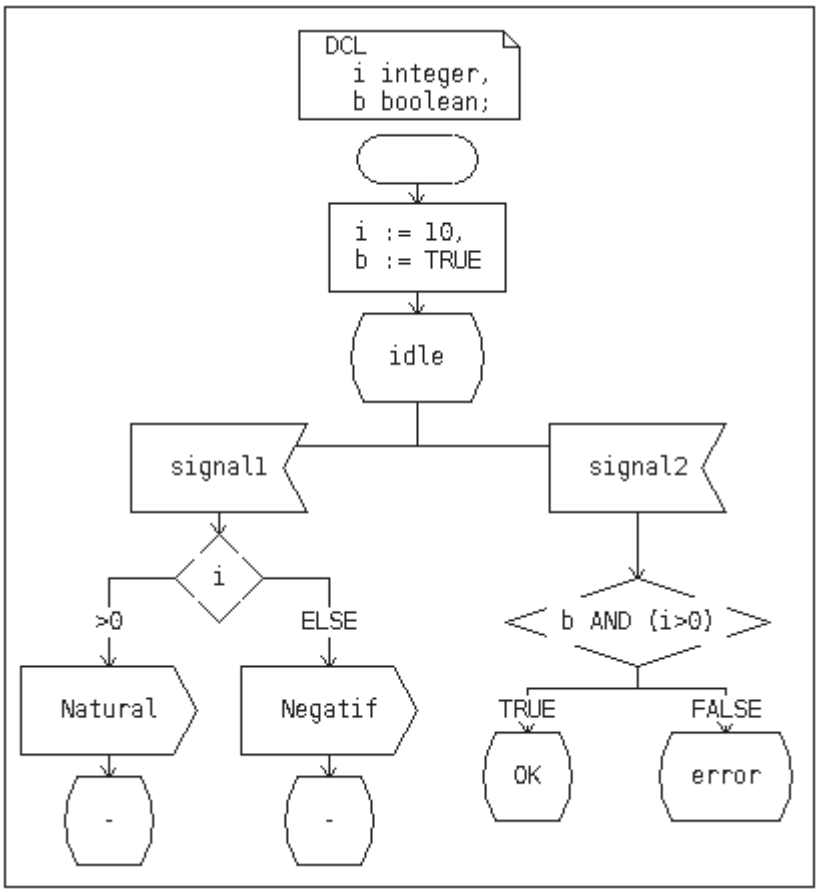
SDL Symbol	Description
	Condition - Decision, which is used to branch according to the value of an expression.

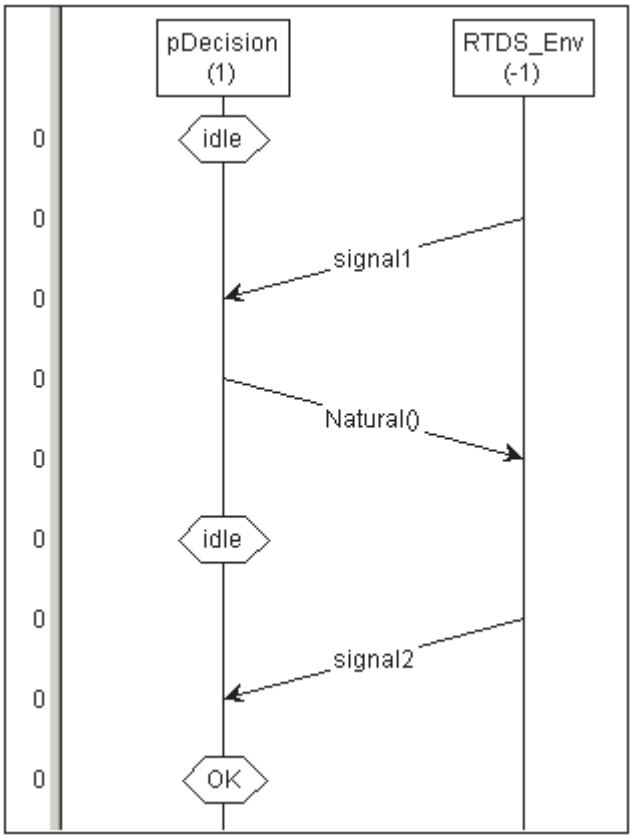
Syntax: `<expr.>`, the condition to be evaluated. There are two ways to evaluate an expression:

- A variable is evaluated by expressing its condition in one of the branches: `<expression>, ELSE`
- A condition expression that returns `TRUE/FALSE` is written in the decision symbol.

Example:

- When receiving `signal1`, `i` is evaluated. If `i>0`, then the left branch is executed. Else, the right branch is executed.
- Upon reception of `signal2`, the expression `b AND (i>0)` is evaluated. If the expression returns `TRUE`, the left branch is evaluated. Else, the right branch is executed.


DCL
i integer,
b boolean;
i := 10,
b := TRUE
idle
signal1
signal2
i
>0
ELSE
b AND (i>0)
Natural
Negatif
TRUE
FALSE
OK
error
-
-


pDecision
(1)
RTDS_Env
(-1)
idle
signal1
Natural()
idle
signal2
OK
0
0
0
0
0
0
0
0
0

Decision example

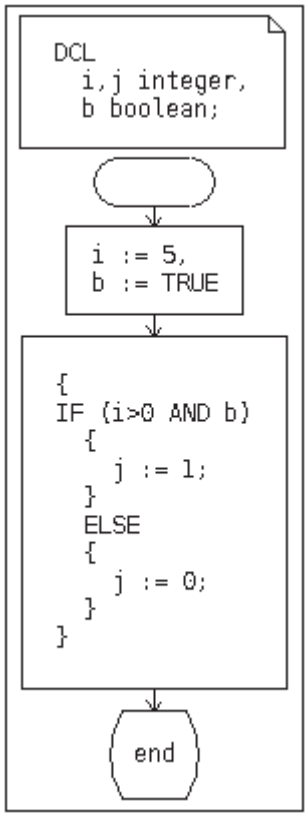
3.4.18 If statement

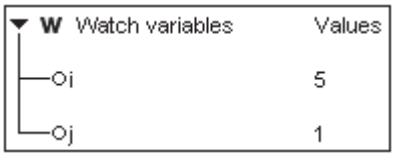
Syntax:

```
{
IF <boolean expression>
  {
    <consequence expression>
  }
  ELSE
  {
    <alternative expression>
  }
}
```

Note that `ELSE` part is optional. Do not confuse the If statement with the conditional expression.

Example:





Observation in SDL Simulator

If statement example

3.4.19 Loop statement

A loop statement is introduced in a task block to execute an iteration statement. It may be used to replace a decision when a more compact notation is preferred. The two optional keywords `BREAK` and `CONTINUE` can be used in a loop statement.

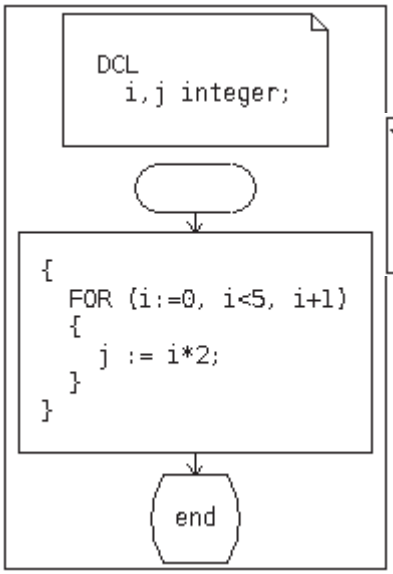
Loop statement

Syntax:

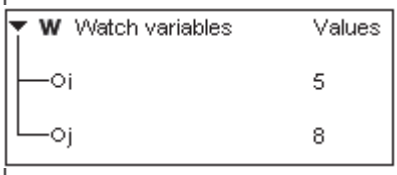
```
{
FOR <statement>
  {
    <expression>;
  }
}
```

Example:

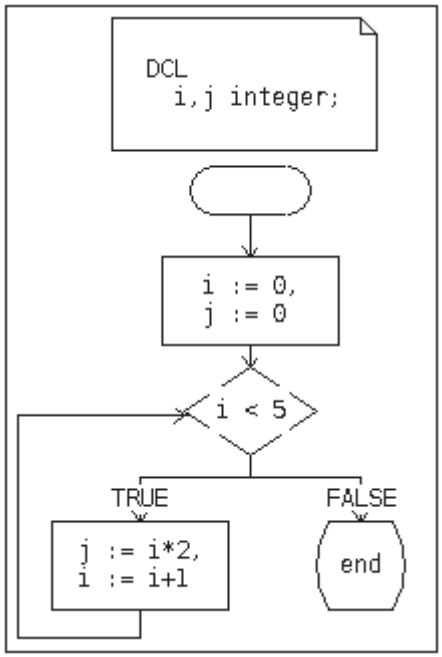
The example below shows the loop statement in textual representation and in graphical representation.



Loop statement



Observation in SDL Simulator in the end of



Graphical representation of

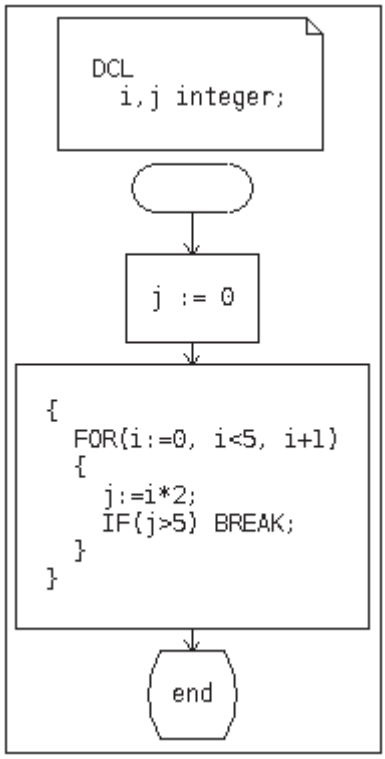
Example of for loop

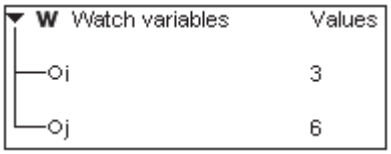
Terminate a FOR loop using a BREAK

Syntax:

```
{
  FOR(statement)
  {
    <expression>;
    IF(condition) BREAK;
  }
}
```

Example:





Observation in SDL Simulator

Loop example using a BREAK

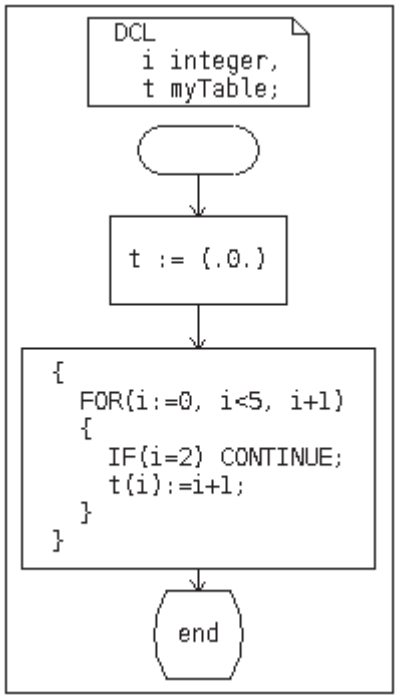
Loop statement with CONTINUE

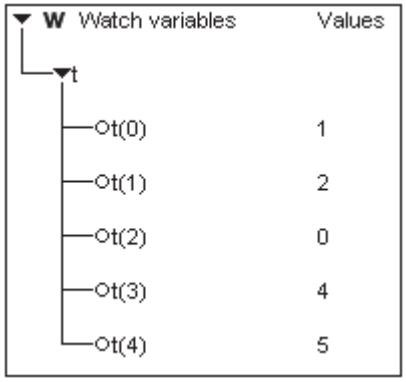
CONTINUE may be used to jump to the next iteration.

Syntax:

```
{
  FOR(statement)
  {
    <expression>;
    IF(condition) CONTINUE;
    <expression>;
  }
}
```

Example:





Observation in SDL Simulator

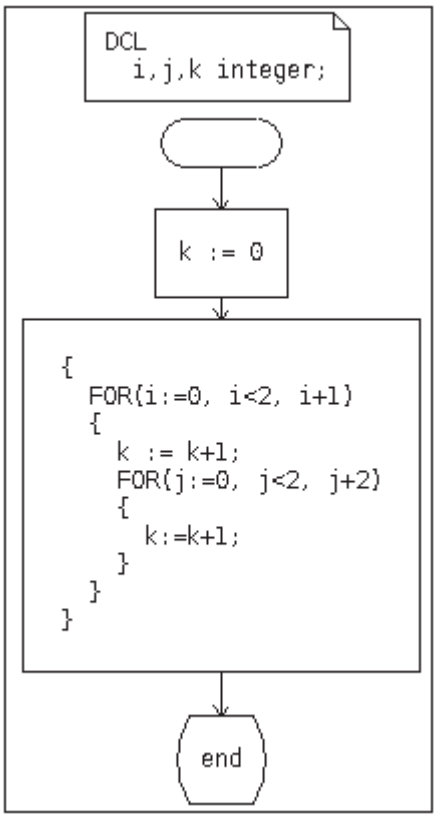
Loop example using a CONTINUE

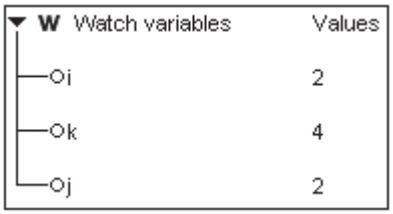
Loop statement in another loop statement

Syntax:

```
{
  FOR(statement)
  {
    <expression>;
    FOR(statement)
    {
      <expression>;
    }
  }
}
```

Example:





Observation in SDL Simulator

Example of a loop statement in a loop statement

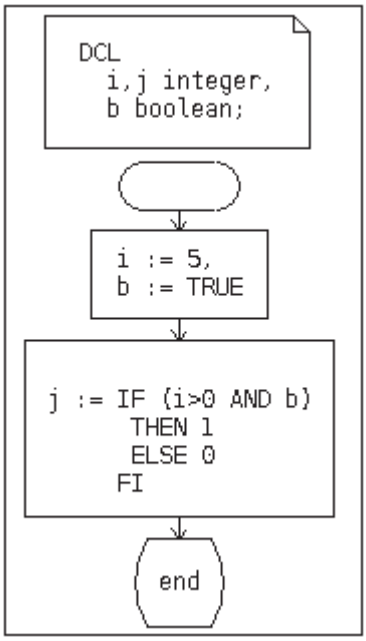
3.4.20 Conditional expression

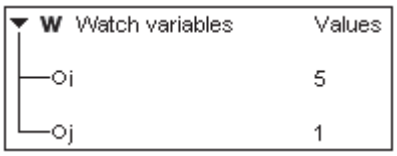
A conditional expression may be used where a return value is expected. Note that the `ELSE` branch cannot be omitted.

Syntax;

```
IF <boolean expression>
  THEN <consequence expression>
  ELSE <alternative expression>
FI
```

Example:

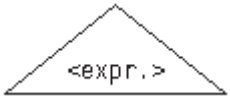




Observation in SDL Simulator

Conditional expression example

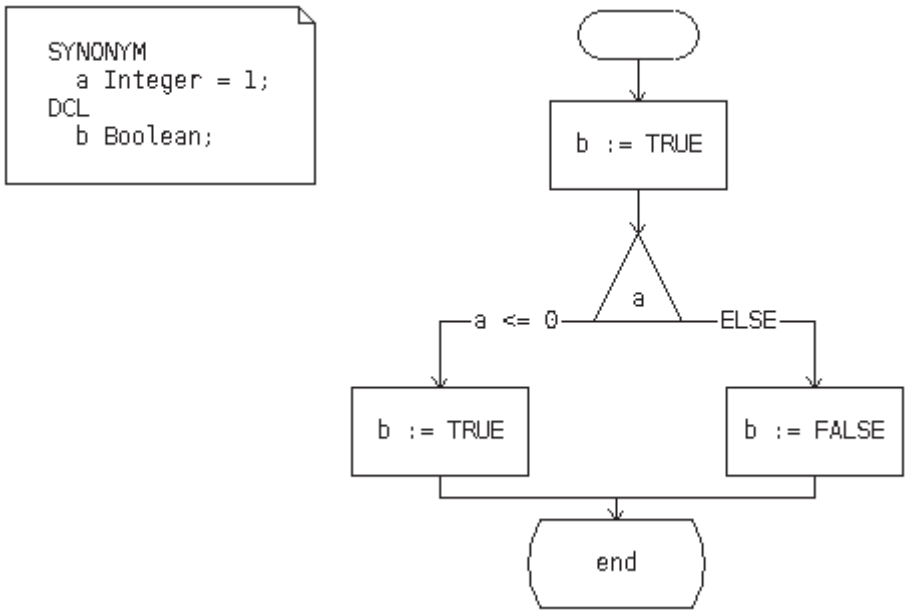
3.4.21 Transition option

SDL Symbol	Description
	Transition options are similar to C pre-processor directive `#if... #endif`. The expression defined must only contain `SYNONYM` types.

The branches of the symbol have values `true` or `false`. The `true` branch is defined when the
expression is defined so the equivalent C code is: `#if <expression>`
The branches can stay separated until the end of the transition or they can meet again and close the option as would do an `#endif`.

Syntax: `<expr.>`, simple expressions (must contain only literals, operators, and methods defined within the package `Predefined` defined as a `SYNONYM`) or informal text.

Example:



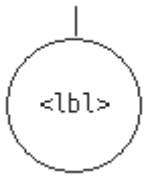
Transition option example

3.4.22 Connectors

Connector are used to:

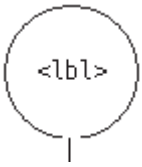
- split a transition into several pieces so that the diagram stays legible and printable
- gather different branches to a same point

3.4.22.1 Connector out/join

SDL Symbol	Description
	It modifies the flow in a FSM by expressing that the next action to be interpreted should be the one that contains the same name for a connector in. The flow of execution goes from the connector out to the connector in symbol.

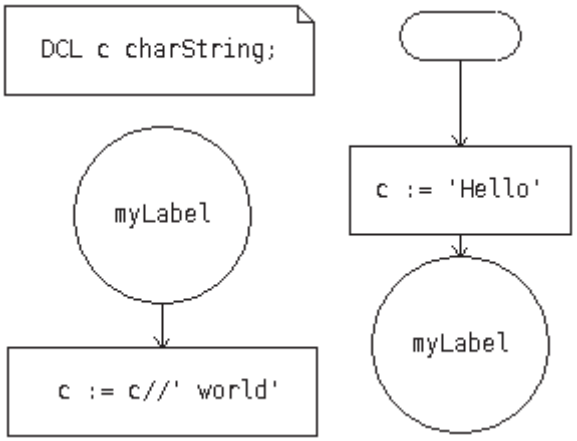
Syntax: `<lbl>`, connector name

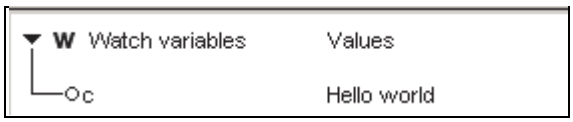
3.4.22.2 Connector in/label

SDL Symbol	Description
	It is the entry point of a transfer control from corresponding joins.

Syntax: `<lbl>`, connector name

Example:

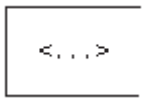




Observation in SDL Simulator

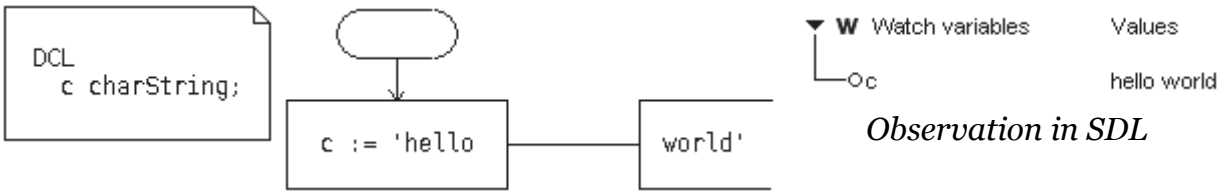
Connectors example

3.4.23 Text extension

SDL Symbol	Description
	It can be connected to any graphical symbol containing text. Its contents is a continuation of the text within the graphical symbol

Syntax: `<. . .>`, any expressions of the same type as the graphical symbol that precede the text extension.

Example:



Observation in SDL

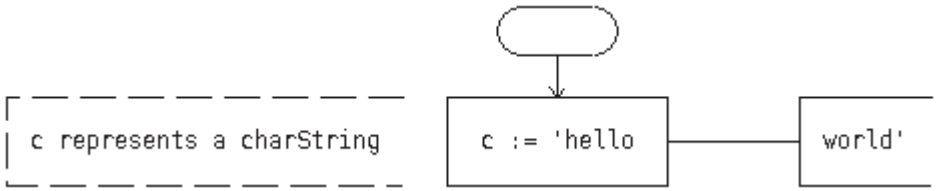
Text extension example

3.4.24 Comment

SDL Symbol	Description
<comment>	A notation to represent comments associated with symbols or text.

Syntax: none

Example:



Comment example

3.5 - Data Types

SDL predefined data in SDL is based on abstract data types (ADT). They are defined in terms of their properties, which means that as long as the same behavior is preserved, SDL does not precise implementation dependent features. For instance the number of bits to store in an `Integer` is not specified. Thus, the SDL descriptions are more general and portable to be used. Furthermore, pointers and memory allocation are also not part of SDL definition.

An ADT is declared in SDL using the `NEWTYPE` construct. Similar to a class, it provides a data structure in addition to other operations to manipulate the structure. The data structure can be in various forms: literals (enumerated values), a struct, an array, etc.

3.5.1 Basic types

Basic data types in SDL are defined in a package called `Predefined`, which is used implicitly by any SDL description.

3.5.1.1 Boolean

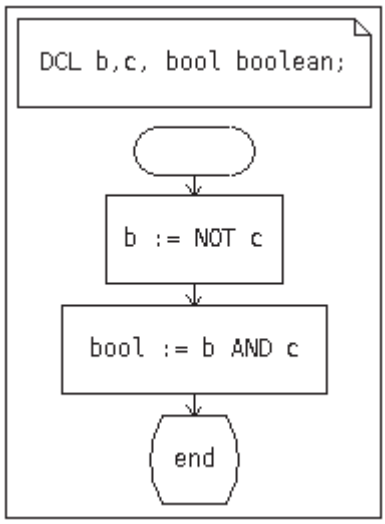
A boolean represents a logical quantity with two possible literals values: `true` or `false`.

Default value: `FALSE`

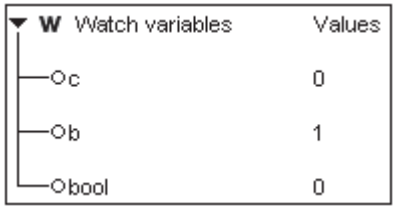
Operators:

Boolean operator	Description
`NOT`	Negation, it is an operation on propositions.
`=`	Equivalence of two operands
`/=`	Non-equivalence of two operands
`AND`	Logical conjunction, it returns `true` if both operands are `true`, otherwise it returns `false`.
`OR`	Inclusive disjunction, it results in `true` whenever one or more of its operands are `true`.
`XOR`	Exclusive disjunction, the result of two operands is `true` if exactly one of the operands has a value `true` (one or the other, but not both).
`=>`	Implication operator
`ANY(boolean)`	Returns true or false randomly.

Example:



Boolean example



Observation in SDL Simulator

3.5.1.2 Character

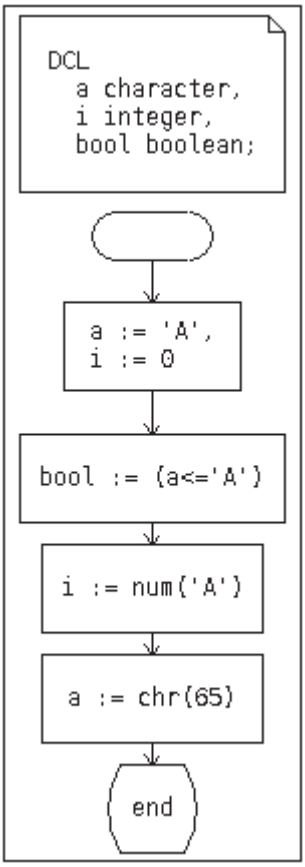
A character represents an ASCII character with the ASCII character codes varies from 0 to 127.

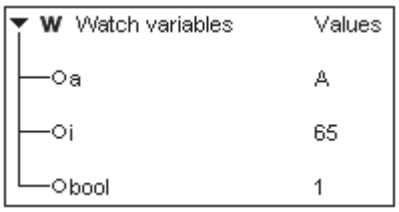
Default value: `''`

Operators:

Character operator	Description
`"="`	Equality of two characters.
`"/="`	Non-equality of two characters.
`"<"`	Less than; the characters are compared by their ASCII code.
`"<="`	Less than or equal.
`">"`	Greater than.
`">="`	Greater than or equal.
`num('MyCharacter')`	Returns the ASCII code of a character.
`chr(MyInteger)`	Returns the character having the given ASCII code.

Example:



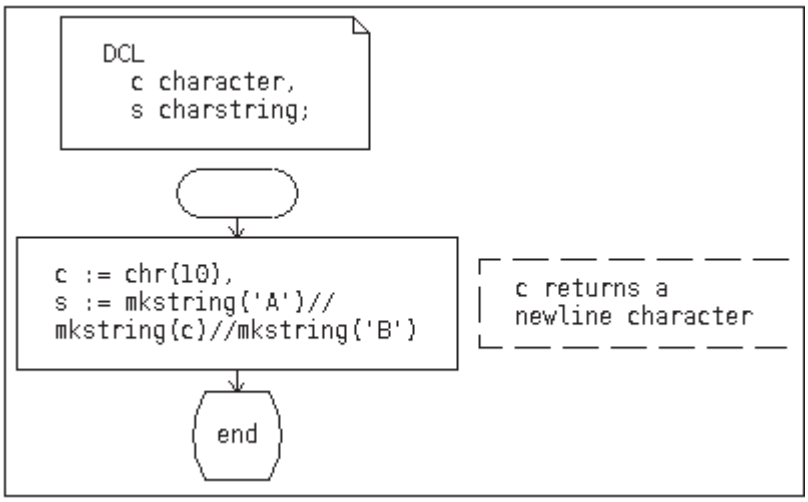


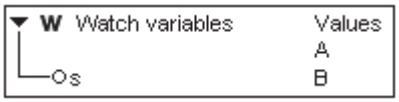
Observation in SDL Simulator

Example of a character

Note that literal names for control characters (ASCII characters from 0 to 32, and 127) such as `NUL`, `STX` or `DEL` are not supported. These characters must be created via the `chr` standard operator with the corresponding ASCII code.

Example: The example below shows the insertion of a new line character between two characters.





Observation in SDL

Example on control character insertion

3.5.1.3 Charstring

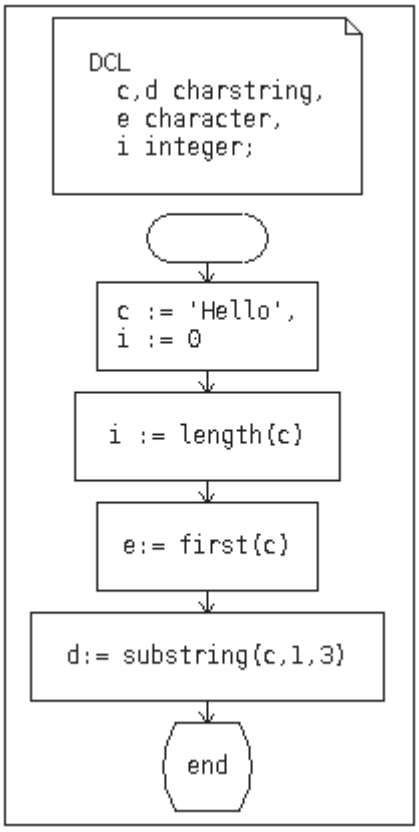
A charstring defines strings of characters. A charstring literal can contain printing characters and spaces. Indices for charstrings start at 1.

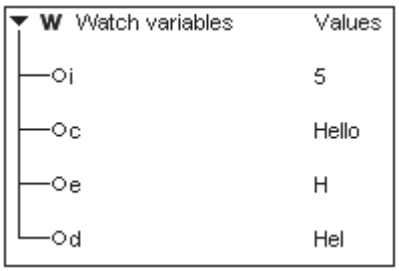
Default value: `''`

Operators:

Charstring operator	Description
`mkstring(character)`	Returns the charstring with length 1 containing the given character.
`length(charstring)`	Returns the length of the charstring.
`first(charstring)`	Returns the first character of the charstring.
`last(charstring)`	Returns the last character of the charstring.
`"//"`	Concatenates two charstrings.
`substring(charstring, integer, integer)`	Returns the substring of the charstring starting at the index given by the second parameter and having the length specified by the third parameter.

Example:





Observation in SDL Simulator

Example of a Charstring

3.5.1.4 Integer

A integer type represents mathematical integers with decimal notation.

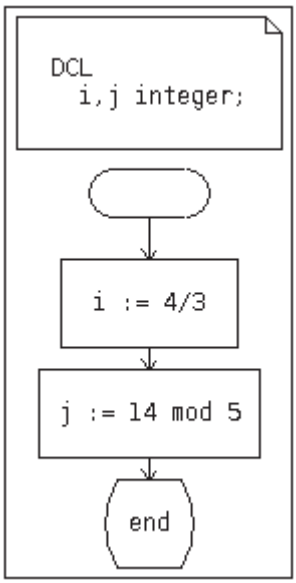
Default value: `0`

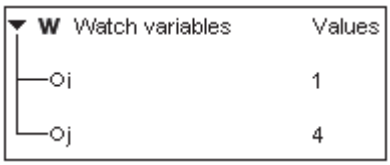
Operators:

Operator	Description
`-`	(Unary) Negative natural numbers
`+`	Addition of two integers
`-`	Subtraction of two integers
`*`	Multiplication of two integers
`/`	Division of two integers
`mod`	Modulo operation of two integers
`rem`	Remainder of the division of two integers
`=`	Equality

Operator	Description
`/=`	Non-equality
`<`	Less than
`>`	Greater than
`<=`	Less than or equals to
`>=`	Greater than or equals to
`ANY(MyIntegerSyntype)`	Returns a random value of the type. This operator is only available on a SYNTYPE of INTEGER with an upper and a lower bound.

Example:





Observation in SDL Simulator

Example of Integers

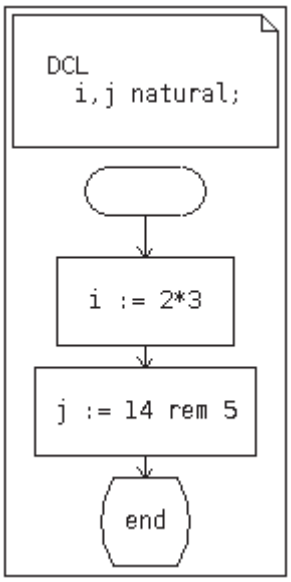
3.5.1.5 Natural

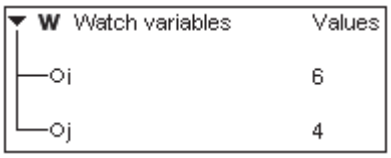
A natural is a zero or a positive Integer.

Default value: `0`

Operators: same as **Integer**

Example:





Observation in SDL Simulator

Example of Naturals

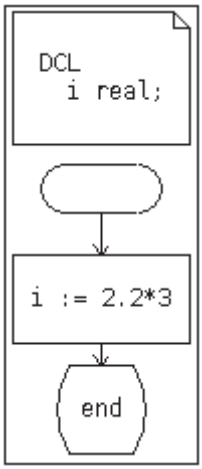
3.5.1.6 Real

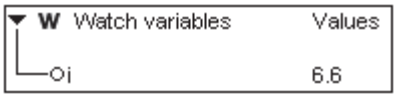
Real represents real numbers. All operators defined to be used for Integer can also be used for Real, except `mod`, `rem`, `Float` and `Fix`.

Default value: `0.0`

Operators: All **Integer** operators can be used except **`"mod"`** and **`"rem"`**

Example:





Observation in SDL

Example of a Real

3.5.1.7 Pid

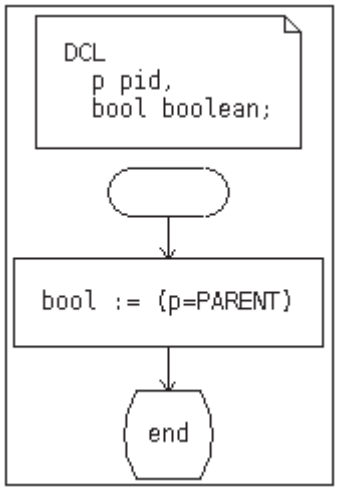
Pid, or process id, is used to identify the process instances.

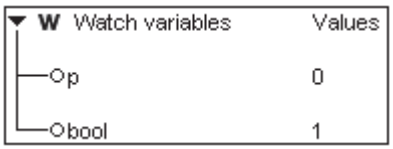
Default value: `NULL`

Operators:

Operator	Description
`=`	Equality
`/=`	Non-equality

Example:





Observation in SDL Simulator

Example of a Pid

3.5.1.8 Duration

The duration sort is used for the values to be added to the current time to set timers. The literals of the sort duration are the same as the literals for the `Real` sort. The meaning of one unit of duration will depend on the system being defined.

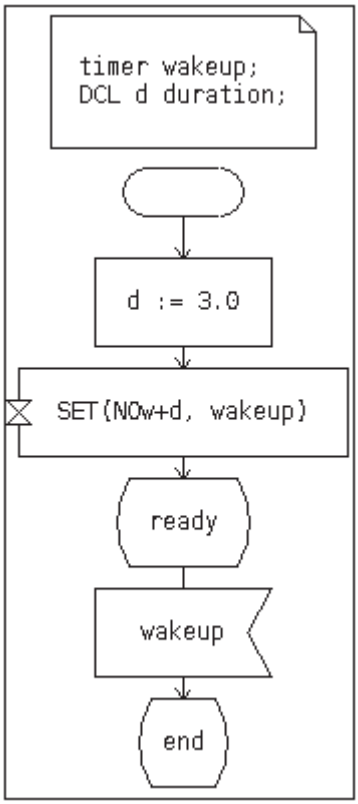
Default value: `0`

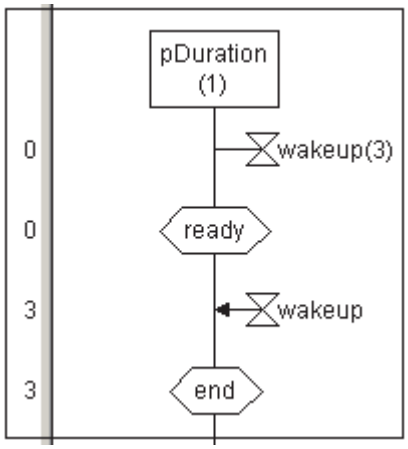
Operators:

Operator	Description
`+`	Addition between two duration
`-`	Negative duration
`-`	Substraction between two durations
`>`	A comparison (strictly greater than) between two durations, returns a boolean
`<`	A comparison (strictly less than) between two durations, returns a boolean

Operator	Description
>=	A comparison (greater than or equals) between two durations, returns a boolean
<=	A comparison (less than or equals) between two durations, returns a boolean
*	Multiplication between a duration and a real number, returns a duration
/	Division between a duration and a real number, returns a duration

Example:





MSC Tracer

Example of a duration

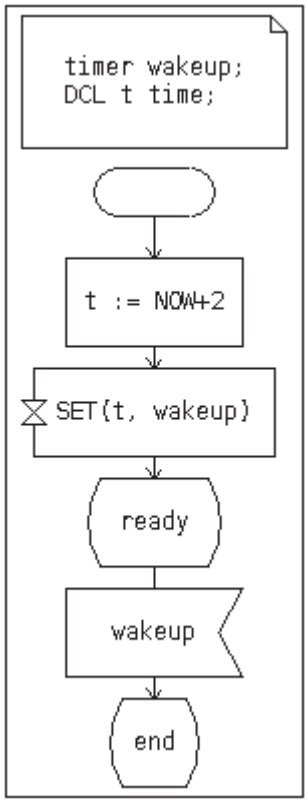
3.5.1.9 Time

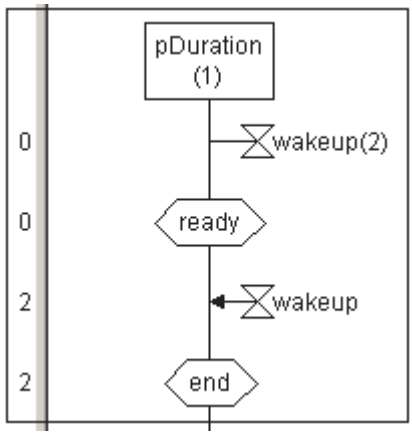
Time values are used to set the expiration time of timers. The origin of time is system dependent. A time unit is the amount of time represented by adding one duration unit to a time. A `NOW` expression has the Time sort.

Operators:

Operator	Description
<	A comparison (strictly less than) between two times, returns a boolean
<=	A comparison (less than or equals) between two times, returns a boolean
>	A comparison (strictly greater than) between two times, returns a boolean
>=	A comparison (greater than or equals) between two times, returns a boolean
+	Addition between a time and a duration, returns time
-	Substraction of (time - duration), returns a time
-	Substraction of (duration - time), returns a duration

Example:





MSC Tracer

Example on time

3.5.2 Constants: SYNONYM

A synonym is used to define constants in SDL.

Syntax: `SYNONYM <synonym name> <synonym type> = <constant expression>;`

Example:
```
SYNONYM count Natural = 100;
SYNONYM Yes Boolean = True;
SYNONYM No Boolean = False;
```

3.5.3 Renaming or constraining existing types: SYNTYPE

A `SYNTYPE` defines an alternative name for an existing type, optionally adding constraints to the base type. A syntype based on a given type will always be compatible with it, meaning that values for the base type can be assigned to the syntype. This might however trigger a runtime error if the value doesn't match the constraints defined in the syntype.

Syntax:
```
SYNTYPE <syntype name> = <syntype base type>
  DEFAULT <default value>
  CONSTANTS <value constraint>
ENDSYNTYPE;
```

Example:
```
SYNONYM NUM_PHONE Integer = 5;
SYNTYPE PhoneNumberType = Integer
  DEFAULT 1;
  CONSTANTS 1..NUM_PHONE
ENDSYNTYPE;
SYNTYPE MyIndexType = Natural
  CONSTANTS < 16
ENDSYNTYPE;
```

All values of the type `PhoneNumberType` will be between 1 and 5 (inclusive); all values of the type `MyIndexType` will be between 0 and 15 (inclusive).

3.5.4 Complex types

Complex user-defined types are created by using the SDL keyword `NEWTYPE`. There are several kinds of complex types, described in the following sections.

Note that the `NEWTYPE` keyword actually defines a *new* type, which won't be compatible with any other one, even if both type have exactly the same definition. Defining a type compatible with another one can only be done via the `SYNTYPE` keyword (see "Renaming or constraining existing types: SYNTYPE" on page 79).

There are some common features for all types defined via the `NEWTYPE` keyword:

- Special constant values of the type can be defined via a `LITERALS` clause within the `NEWTYPE`. For example, the default value `NULL` for the `PID` type is a literal for

the PID type. These literals have no explicit value. They also can be used to define the equivalent of an enumerated type; the NEWTYPE will then contain only a LITERALS clause.

- A default value for the type can be given via the DEFAULTS clause within the NEWTYPE. See the sections on the actual type kinds below for the possible syntax for these values.
- Types can define operators acting on the values of this type. Operators are mostly equivalent to functions, except they depend on the actual type of their parameters and of their return value. For example, two different types can define the same operator op, as long as the types of the parameters or of the return value for both op operators are different in the two types. Operators are defined via the OPERATORS clause.
  For example:

```
NEWTYPE MyType1
   ...
   OPERATORS
     op1: MyType1 -> Integer;
     op2: Integer -> MyType1;
ENDNEWTYPE;
NEWTYPE MyType2
   ...
   OPERATORS
     op2: Integer -> MyType2;
     op3: MyType2, Integer -> Integer;
ENDNEWTYPE;
```

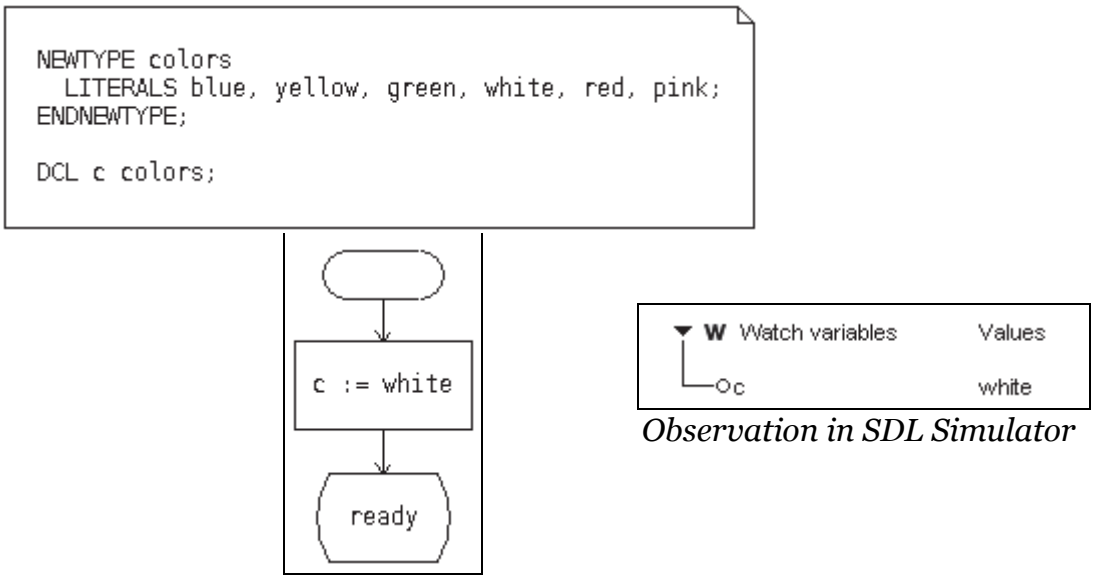
  Operators op2 in MyType1 and op2 in MyType2 are different and not ambiguous, since their return type is different. When writing "var := op2(1)", the type of var will determine the actual operator to call.
  Note that only the signature of the operators are defined in the SDL specification. Their implementation is supposed to be external. Note also that defining an operator with no parameters or with no return type is illegal in SDL, even if it is often allowed in tools supporting the language.

Note that SDL has a semantics based on values. This means that assigning to a variable, or passing a value to a parameter, or to a message, and so on, will *always* copy the value from the source to the destination, even for complex types. The only exception to this rule are parameters declared as IN/OUT to a procedure. So an operator will never modify any value passed to it.

3.5.4.1 Enumerated type: LITERALS

The equivalent of an enumerated type can be created in SDL by using a NEWTYPE with no definition, but having a LITERALS clause containing the names for the constants in the type.

Example:



Observation in SDL Simulator

Note the ANY operator will randomly return one of the LITERALS value. Usage is: `ANY(<type name>)`.

3.5.4.2 Structure type: STRUCT

A `STRUCT` is a collection of typed information, defined as fields in the `STRUCT`. The syntax for defining a `STRUCT` is the following:

```
NEWTYPE MyStructType
  STRUCT
    fieldName1  FieldType1;
    fieldName2  FieldType2;
    ...
ENDNEWTYPE;
```

Values for `STRUCT` types can be specified as a whole using the syntax `(. <value for first field>, <value for second field>, ... .)`. This can typically be used in default values for the type itself:

```
NEWTYPE PointType
  STRUCT
    x Integer;
    y Integer;
  DEFAULT (. 0, 0 .);
ENDNEWTYPE;
```

The syntax to access a field in a `STRUCT` is `<STRUCT variable name>!<field name>`. So for example, if there is a variable `p` with the type `PointType`, changing its `x` field will be done via: `p!x := 2`

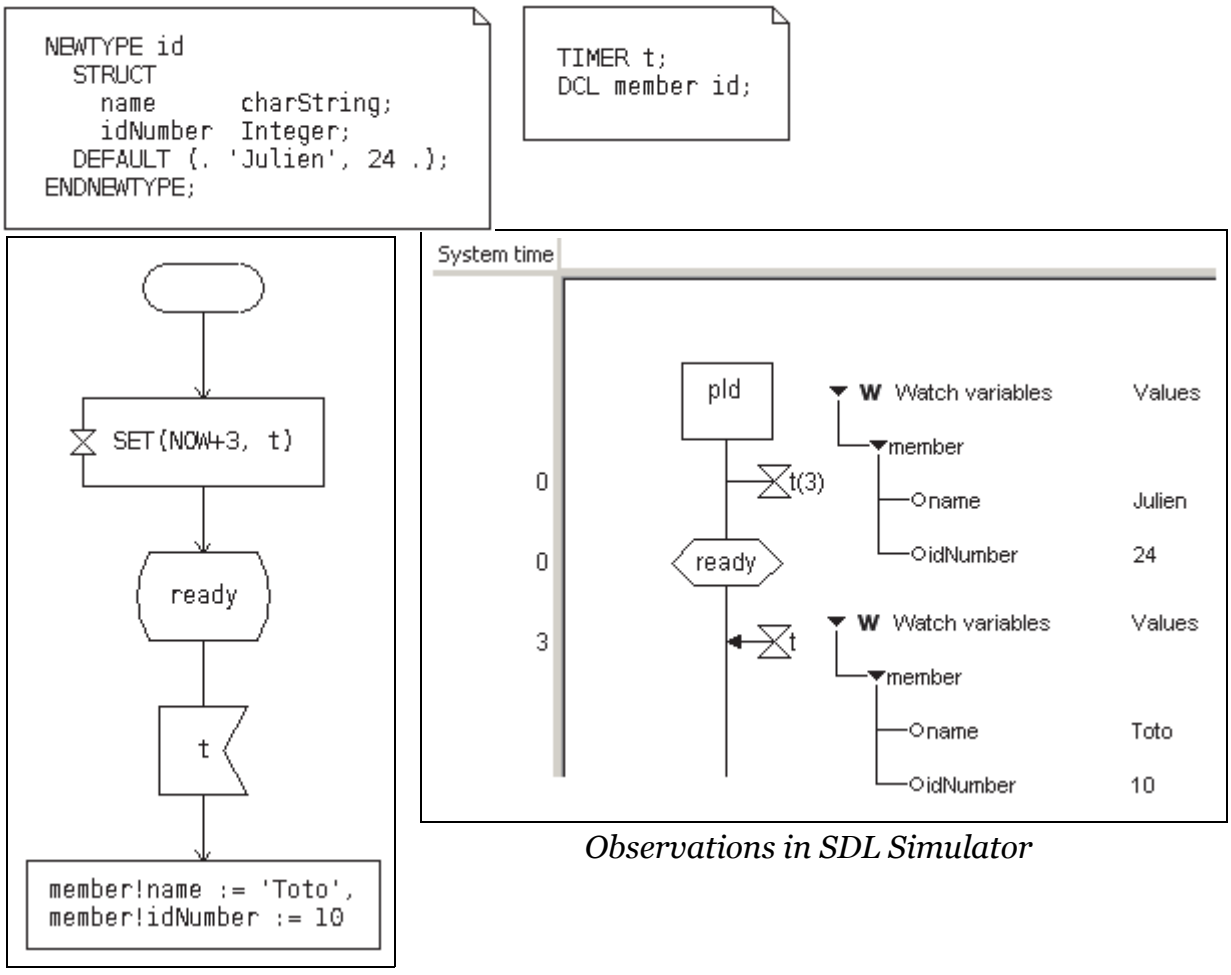
A field in a `STRUCT` can be marked as `OPTIONAL`:

```
NEWTYPE TaggedPointType
   STRUCT
      x    Integer;
      y    Integer;
      tag  CharString OPTIONAL;
ENDNEWTYPE;
```

In this case, an implicit field having the name of the optional field followed by "Present" will be added to the STRUCT, with the type Boolean, and will be automatically set to True if the field has a value, and to False if it hasn't. For example:

```
DCL p TaggedPointType;
p!x := 3,                    -> p!tagPresent is False
p!y := 5,                    -> p!tagPresent is False
p!tag := 'Ending point'      -> p!tagPresent is True
```

Example:



Observations in SDL Simulator

3.5.4.3 Choice/union types: CHOICE

A CHOICE is an alternative of different typed informations, defined as fields in the CHOICE. The syntax to define a CHOICE is the following:

```
NEWTYPE MyChoiceType
  CHOICE
    field1 FieldType1;
    field2 FieldType2;
    ...
ENDNEWTYPE;
```

The syntax to access fields in the CHOICE is the same as for STRUCT types: <CHOICE variable name>!<field name>. An implicit field named present is always added to a CHOICE, having a type defining only the literals with the name of the CHOICE fields (field1, field2, etc... in the example). Any assignment of one of the fields in the CHOICE will set the value for the present field to the name of the assigned field. For example:

```
NEWTYPE MyChoiceType
  CHOICE
    i Integer;
    s CharString;
ENDNEWTYPE;
DCL c MyChoiceType;
c!i := 1;      -> value for c!present is i
c!s := 'foo'; -> value for c!present is s
```

3.5.4.4 Associative arrays: Array generator

In SDL, an Array is a mapping from one typed information to another. It does not match what is usually called an array in programming languages (see “Ordered lists: String generator” on page 84 for that). The Array generator has two type parameters, named the index type and the element type, which defines a mapping from the index to the element.

Syntax:

```
NEWTYPE <Array type name>
  Array(<index type>, <element type>)
ENDNEWTYPE;
```

Default value: all elements in the array are set to the default value for its element type.

Operators:

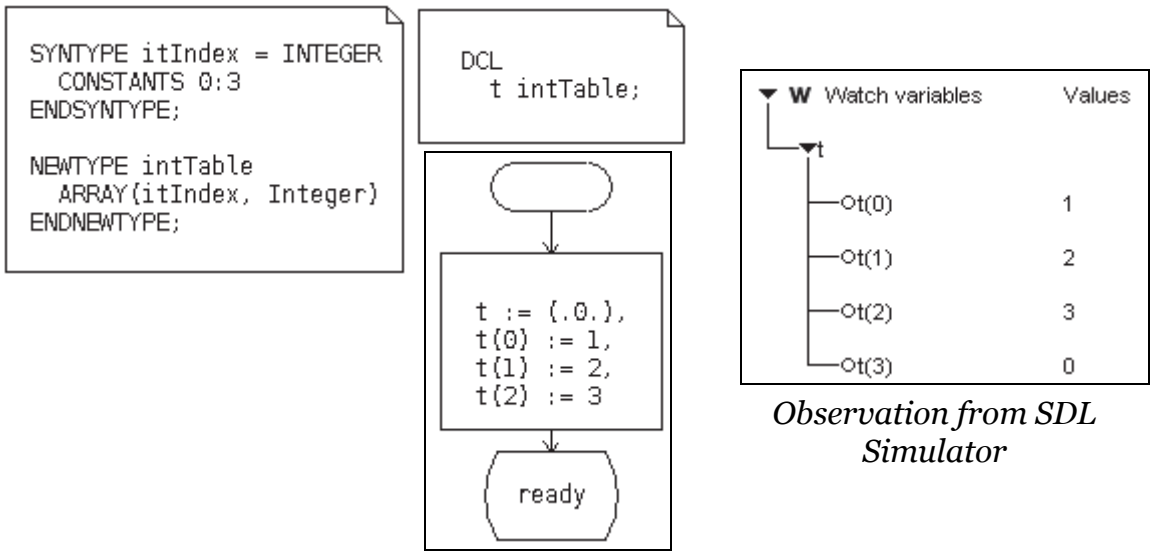
The syntax to access the element in an array a for the index i is a(i). So for example:

```
NEWTYPE MyArrayType
  Array(CharString, Integer)
ENDNEWTYPE;
DCL a MyArrayType;
a('foo') := 42
```

will set the value in the array for the character string index 'foo' to the integer 42.

Values for arrays can be specified as a whole using the syntax `(. <element value> .)`. Note that only a single `<element value>` is allowed, and that it will be set for *all* indices in the array.

Example: In the example below, the new data type `intTable` is an Integer array, indexed by the type `itIndex`.



Observation from SDL Simulator

3.5.4.5 Ordered lists: String generator

A `String` type is an ordered list of elements of any other type. Elements are accessed using their integer index, which starts at `1`. The syntax to access the element in the `String s` at index `i` is `s(i)`.

Syntax:

```
NEWTYPE <String type name>
  String(<element type name>)
ENDNEWTYPE;
```

Default value: the empty string

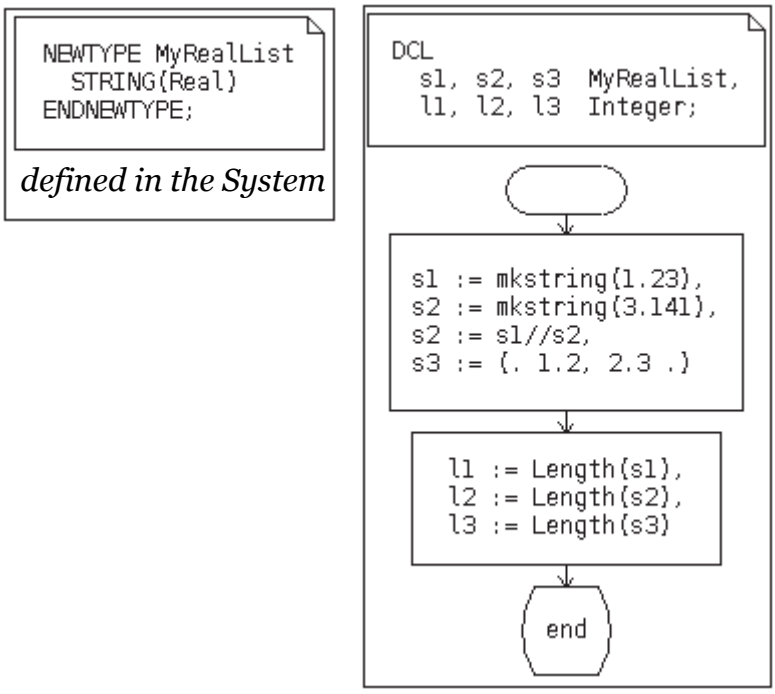
Operators:

Charstring operator	Description
`mkstring(element)`	Returns the string with length 1 containing the given element.
`length(string)`	Returns the length of the string.
`first(string)`	Returns the first character of the string.
`last(string)`	Returns the last character of the string.
`"//"`	Concatenates two strings.

Charstring operator	Description
`substring(string, integer, integer)`	Returns the substring of the string starting at the index given by the second parameter and having the length specified by the third parameter.

Values for strings can also be specified as a whole via the syntax `(. <element value 1>, <element value 2>, ... .)`. This will define a value for the string with as many elements as there are within the `(. .)`, with the given values.

Example:



3.5.4.6 Multi-sets: Bag generator

A `Bag` type is a collection of items with no specific order. It is similar to the mathematical notion of a set, except that an element can appear several times in a `Bag`.

Syntax:

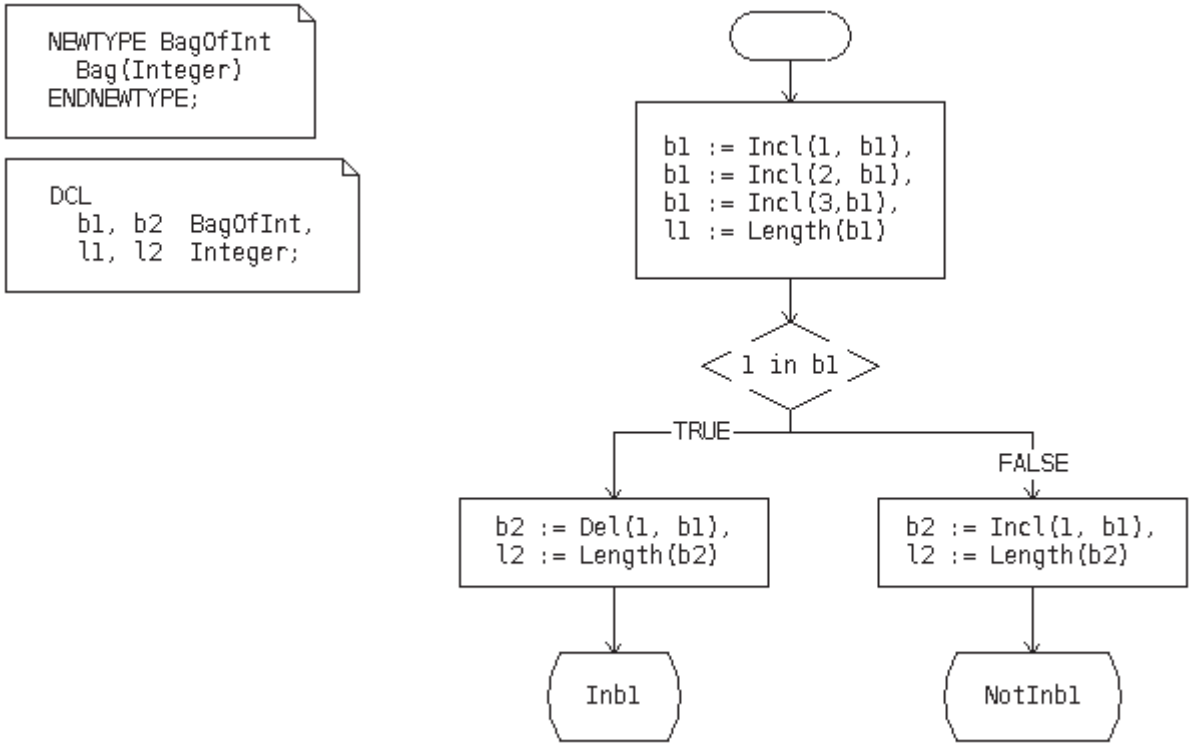
```
NEWTYPE <Bag type name>
  Bag(<element type name>)
ENDNEWTYPE;
```

Operators:

Operator	Description
`empty`	Returns an empty bag.

Operator	Description
`"in"`	Checks if a value appears as an element in a bag, returns a boolean.
`incl(element, bag)`	Returns the bag having the same contents as `bag`, with `element` added to it.
`del(element, bag)`	Returns the bag having the same contents as `bag`, with `element` removed from it. If `element` is not in `bag`, returns a copy of `bag`.
`"<"`	Tests if a bag is a strict subbag of another bag.
`">"`	Tests if a bag is a strict superbag of another bag.
`"<="`	Tests if bag is a subbag of another bag, or is equal to it.
`">="`	Tests if a bag is a superbag of another bag, or is equal to it.
`"and"`	Returns the intersection of two bags.
`"or"`	Returns the union of two bags.
`length(bag)`	Returns the number of elements in a bag, counting duplicates.
`take(bag)`	Returns a random element in bag.

Example:

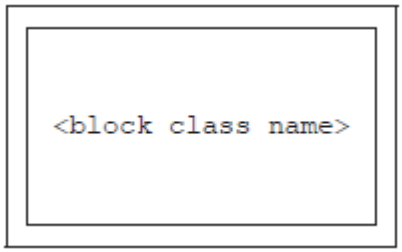


3.6 - Object Orientation

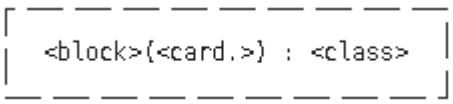
In order to reuse structural entities, SDL offers to users object-oriented features.

3.6.1 Block class

By defining a block class, all blocks of the same class have the same properties as defined for the specific block class. A block class is represented by a block symbol with a double frame with no channels connected to it:



A block class can be instantiated in a system or a block by inserting the block class instance represented below:

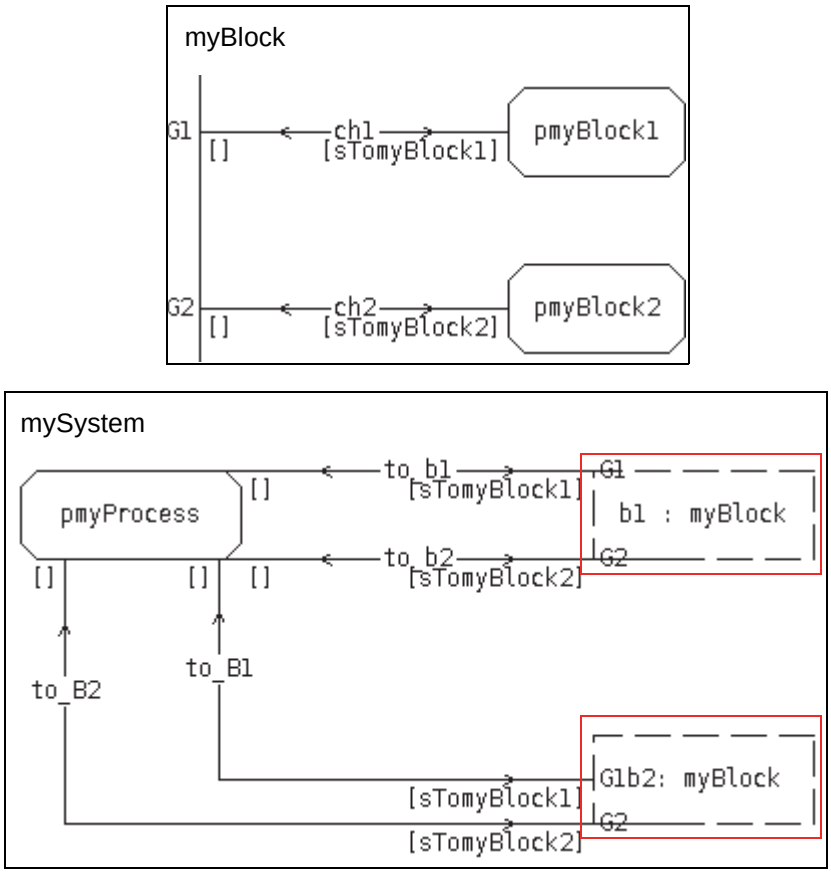


Syntax: `<block>(<card.>) : <class>`

- `<block>`: Block name
- `<card.>`: Cardinality
- `<class>`: Class name

In a block class instance, gates are defined and represent connection points for channels. They are used to connect the block class instances to the system. The signals listed in the gates must be consistent with the signals listed in the connected channels.

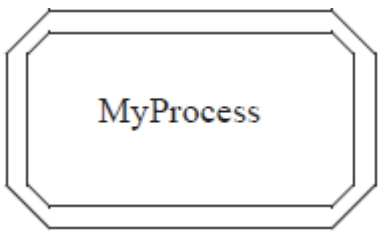
Example: In the example below, `b1` and `b2` are block class instances of `myBlock`.



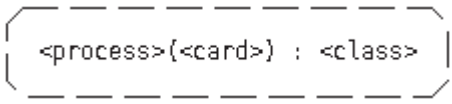
Example of block class

3.6.2 Process class

A process class allows all process instances of the same class to have the same properties as defined for the process class, besides having the possibility to inherit from a process super-class.A process class is represented by a process symbol with a double frame with no channels connected to it:



A process class can be instantiated in a system or a block by inserting the process class instance represented below:

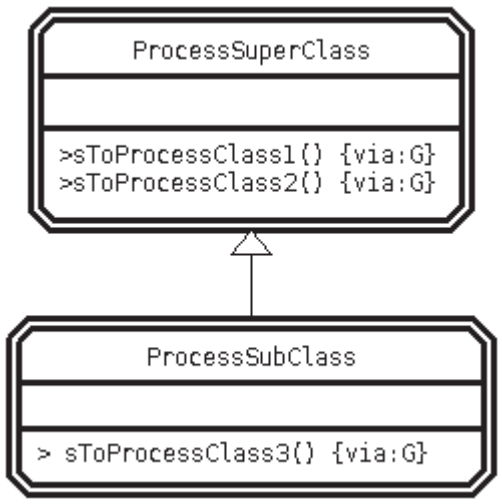


Syntax: `<process>(<card>) : <class>`

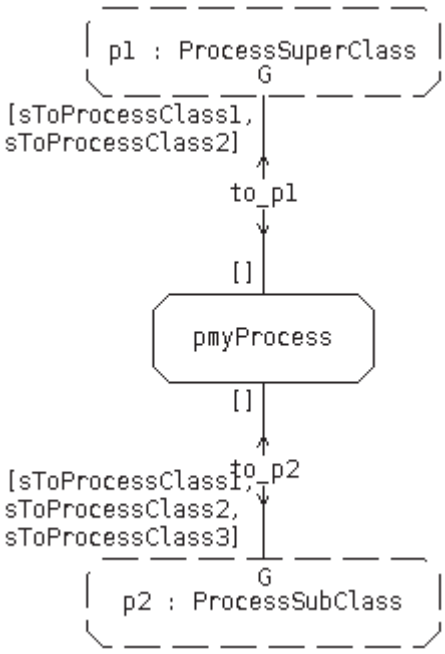
- `<process>`: Process name
- `<card>`: Cardinality
- `<Class>`: Class name

Since a class is not supposed to know the surrounding architecture, signal outputs should not use the `TO_NAME` concept. Instead `VIA` should be used.

Example:



Relation between sub-class and super-class



Example of process class

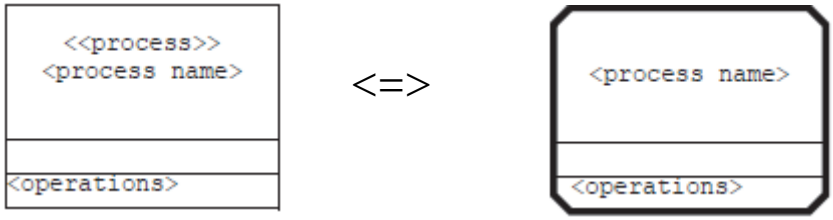
3.6.3 Class diagram

The SDL class diagram is conform to UML class diagram.

3.6.3.1 Class

A class is the descriptor for a set of objects with similar structure, behavior and relationships. In order to create specific types of classes, a **stereotype**, which is an extension of the UML vocabulary is used. When the stereotype is present, it is placed above the class

name within quotation mark (guillemets). However, besides using this purely textual notation, special symbols may also be used in place of the class symbol:



Different notations for a class stereotyped as a process

There are two types of classes:

- **active classes**: an instance of an active class owns a thread of control and may initiate control activity
- **passive classes**: an instance of a passive class holds data, but does not initiate control.

In a class diagram, agents are represented by active classes. Agent type is defined by the class stereotype. Known stereotypes are: `block class` and `process class`. Active classes do not have any attribute. Operations defined for an active class are incoming or outgoing signals.

Syntax: `<signal direction> <signal name> [(<parameter type>)] [{via <gate name>}]`

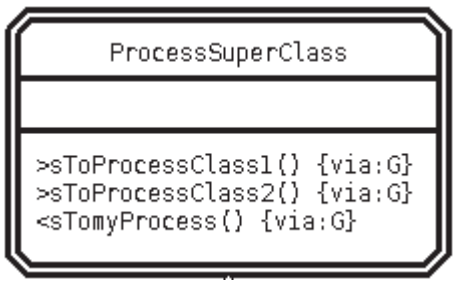
- `<signal direction>`: The signal direction can be
  `>`: for incoming signals
  `<`: for outgoing signals
- `<signal name>`: Signal name
- `<parameter type>`: Parameters associated to the signal
- `{via <gate name>}`: The gate name where signals pass through

Example:

The process class `ProcessSuperClass` can:

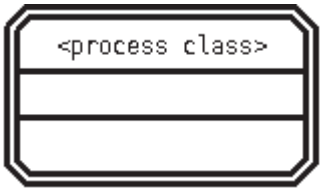
- receive signals `sToProcessClass1()` and `sToProcessClass2()`
- transmit signal `sTomyProcess()`

through gate `G`.



Pre-defined graphical symbols for stereotyped classes in SDL are:

Block class

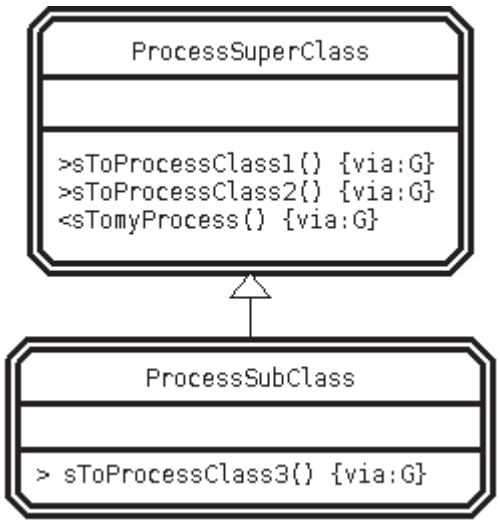


Process class

3.6.3.2 Specialization

Specialization defines the relationship 'is a kind of' between two classes. It is normally used to describe the relation between a super-class and a sub-class. The most general class is called the super-class and the specialized class is the sub-class. The relationship from the sub-class to the super-class is called **generalization**.

Example: In the example below, `ProcessSubClass` is a kind of `ProcessSuperClass`.



4 - SDL support in RTDS

4.1 - Architecture and communication

Features	SDL Simulator	SDL-RT	C Code Generation	C++ Code Generation
Block	YES	YES	YES	YES
Process	YES	YES	YES	YES
Procedure	YES	YES	YES	YES
Package	YES	YES	YES	YES
Block class	PARTIAL	YES	YES	YES
Process class	PARTIAL	PARTIAL	YES	YES
Channel with delay	NO	NO	NO	NO
Class diagram	NO	YES	NO	YES
HISTORY nextstate(_*)	NO	N/A	NO	NO
ERROR keyword	NO	N/A	NO	NO
ANY keyword	NO	N/A	NO	NO
Delays in channel	NO	N/A	NO	NO
Class diagram	NO	YES	NO	NO
Package diagram	NO	N/A	NO	NO

4.2 - Behavior

Features	SDL Simulator	SDL-RT	C Code Generation	C++ Code Generation
Remote procedure	NO	N/A	NO	NO
Macro	YES	NO	NO	NO
Composite state	YES	NO	NO	NO
Start	YES	YES	YES	YES
Stop	YES	YES	YES	YES
State	YES	YES	YES	YES

Features	SDL Simulator	SDL-RT	C Code Generation	C++ Code Generation
Input	YES	YES	YES	YES
Priority input	YES	NO	NO	NO
Save	YES	YES	YES	YES
Continuous signal	YES	YES	YES	YES
Enabling condition	NO	NO	NO	NO
Output	YES	YES	YES	YES
Priority output	YES	NO	NO	NO
Task	YES	YES	YES	YES
Process creation	YES	YES	YES	YES
Timer	YES	YES	YES	YES
Decision	YES	YES	YES	YES
Transition option	YES	YES	YES	YES
Connectors	YES	YES	YES	YES
Text extension	YES	YES	YES	YES
Comment	YES	YES	YES	YES
`HISTORY` nextstate (_*) in composite states	NO	N/A	NO	NO
`ENTRY` and `EXIT` procedure in composite states	NO	N/A	NO	NO
`ANY` operator	YES[1]	N/A	NO	NO
Interface definition	NO	N/A	NO	NO
Generator bag	NO	N/A	NO	NO
INHERIT in NEWTYPE	NO	N/A	NO	NO

1. Partial: Only for booleans, literal types and types based on integers from which a lower bound and upper bound can be easily extracted.

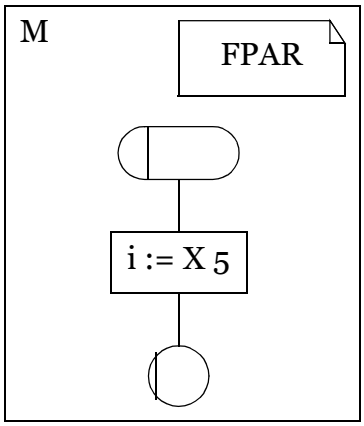
4.3 - SDL Abstract Data Types

Features	SDL Simulator	C Code Generation	C++ Code Generation
Boolean	YES	YES	YES
Character	YES	YES	YES
Charstring	YES	YES	YES
Integer	YES	YES	YES
Natural	YES	YES	YES
Real	YES	YES	YES
Pid	YES	YES	YES
Duration	YES	YES	YES
Time	YES	YES	YES
String	YES	YES	YES
Array	YES	YES	YES
Bag	YES	YES	YES
Powerset	NO	NO	NO
Synonym	YES	YES	YES
Syntype	YES	YES	YES
Newtype	YES	YES	YES
Remote variables	YES	NO	NO
If statement	YES	YES	YES
Loop statement	YES	YES	YES
Conditional expression	YES	YES	YES

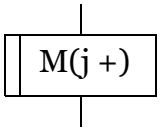
4.4 - Macros diagrams

Since RTDS does not rely on a textual representation such as the PR format for the SDL diagrams, a few differences had to be introduced between the semantics defined by the Z100 specification and the one supported by RTDS:

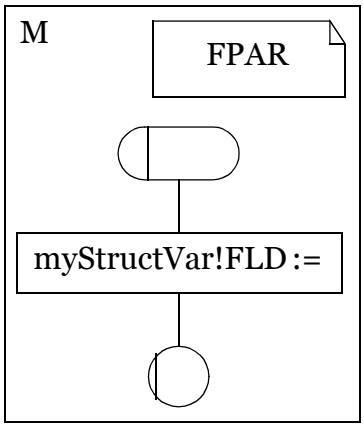
- Parameters passed to macros must be valid expressions. So, for example, it is impossible to do:



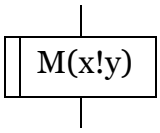
INVALID!



- If a parameter is used in a macro where only a name is valid, the actual parameter must be a valid name. So for example, it is impossible to do:

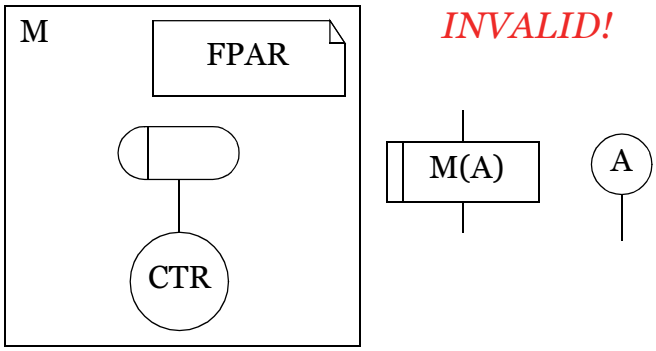


INVALID!

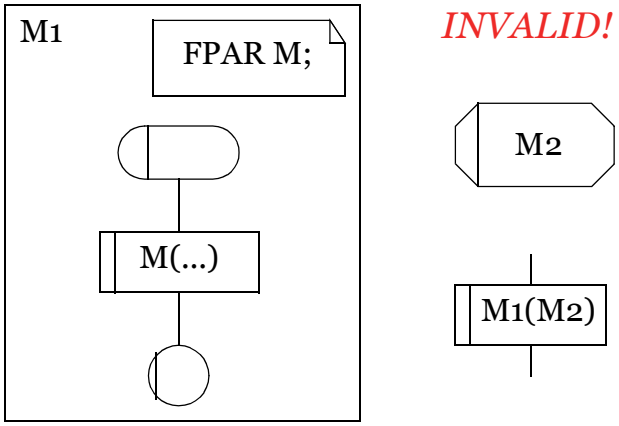


- If macro parameters are expressions, they will be evaluated on their own, and not in the context where they are used. For example, in a macro with a parameter `X`, if the actual value passed for X is `1 + 3`, the value for `X * 5` in the macro will be `(1 + 3) * 5`, and not `1 + 3 * 5` as expected.

- Some names cannot be specified as macro parameters:
  - Connector names, either in or out. So doing this will not work:



  - Names of macros called in a macro. So doing this will not work:



4.5 - Composite state support

RTDS supports composite states, but not exactly as described in the SDL 2000 version of the Z.100 specification.The concepts are however basically the same:

- There are special states that can have one or several sub-state machines.
- The sub-state machine(s) associated to such a state begin their execution when the state is entered.When several sub-state machines are present, they execute in parallel.
- Transitions can be defined on such a state as a whole. If such a transition ends with a next state, all sub-state machines are ended. There is a special pseudo-state named "history state" allowing to continue the execution of all sub-state machines where they were when the transition was entered.
- The concept of sub-state machine entry points allowing to start a sub-state machine in different contexts is currently not supported by RTDS.

RTDS however introduces differences in terminology and graphical representations. The reasons for these differences are:

- *Terminology consistency problems between SDL 2000 and former SDL versions*:
  In SDL 2000, a state with exactly one sub-state machine is named a state sub-structure. In former SDL versions, the term "sub-structure" was used for block diagrams containing other blocks. This additional level has been dropped in SDL

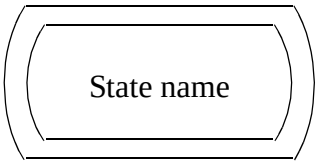
2000 and blocks may now contain other blocks without an intermediate sub-structure level. This makes the concept of sub-structure quite unclear, especially for users of former SDL versions.

- *Terminology consistency problems between SDL and the UML*:
  In SDL 2000, a composite state is a state containing other states each having a sub-structure, i.e. a sub-state machine. In the UML, the meaning is reversed: a composite state is a state having one or several sub-state machines. When several sub-state machines are present, the UML uses the name "state aggregation". This is especially confusing since the concepts in SDL 2000 are clearly inspired from those found in the UML.
- *Confusing graphical representations in SDL 2000*:
  In SDL 2000, the same state symbol is used with no less than 3 different meanings:
  - for "normal" states;
  - for states with a sub-structure, i.e. with an associated sub-state machine;
  - for composite states, containing several states with sub-structures.

  This makes the reading of state machines quite difficult.
- *Graphical consistency problems between SDL 2000 and former SDL versions*:
  Former SDL versions already had a notation for a sub-state machine with services. Even if this concept cannot be reused as it was defined, re-using the symbol itself to represent a sub-state machine can improve the overall readability, especially for users of former SDL versions.
- *Loss of information from former SDL versions*:
  Former SDL versions supported channels connecting the services to the parent process environment and between each other. This allows to specify the messages exchanged between services and to the other processes. These messages can no more be specified in SDL 2000.

So here are the terminology and graphical representation used in RTDS:

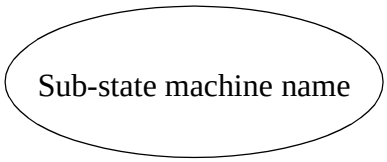
- A *composite state* is a state with one or several sub-state machines. This is consistent with the UML terminology.
- A composite state must be declared in its parent process with a *composite state declaration symbol*. Since such a symbol does not exist in the Z.100 specification, the symbol used is the one for a composite state type definition:



  This allows to quickly spot which states are composite and which aren't.
- Each time a composite state is used, it should be specified with a *composite state symbol*. Since this symbol does not exist in the Z.100 specification, the symbol used is the one for a composite state type instanciation:

NB: the use of this symbol is only recommended, since transitions can be defined for several states, some of them "normal" and some of them composite. If a normal state symbol contains only composite states, a warning will however be generated during the semantics check.

- The definition of a composite state in terms of sub-state machines is contained in a *composite state diagram*, represented in the project tree and stored in its own file. This diagram is composed of:
  - A set of *sub-state machine symbols*, also named *service symbols* since their graphical representation is exactly the one for services found in former SDL versions:



  - A set of *channels*, linking the services between each other or to the diagram's external frame. These channels have the same meaning as the channels between services in former SDL versions. If connection names are given for channels connecting to the diagram's external frame, they refer to channels connecting to the parent process or service.
- To each sub-state machine or service is attached a *sub-state machine diagram* or *service diagram*, which is normal process state machine. The start and end symbols are the same as in processes. These diagrams can define and use other composite states, allowing composite state nesting up to any level.

5 - Mapping of SDL to IF concepts

5.1 - Scope

The IF language, specified by VERIMAG laboratory, is a representation of automaton systems. Based on that representation toolkits offer model checking and test generation. This chapter describes the translation rules implemented by RTDS to export an SDL system to an IF system. As not all SDL concepts can be translated, restrictions or possible ways around will be described.

5.2 - Translation table

The symbols used in this table are:

- ✓ if the feature is fully supported;
- ✓ if the feature is partially supported;
- ✗ if the feature is not supported at all.

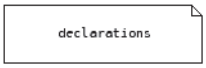
SDL Category	SDL concept	Translated
	BOOLEAN	✓
	INTEGER	✓
	REAL	✓
	CHARACTER	✓
	CHARSTRING	✓
	PID	✓
	SIGNAL	✓
	SIGNALLIST	✓
	SYNONYM	✓
	SYNTYPE	✓
	NEWTYPE LITERALS	✓

Table 1: SDL2IF translation table

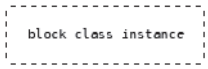
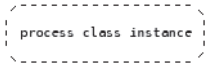
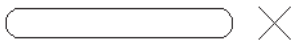
SDL Category	SDL concept	Translated
	NEWTYPE STRUCT	✓
	NEWTYPE STRUCT SELECT	✗
	NEWTYPE CHOICE	✗
	NEWTYPE ARRAY	✓
	OPERATORS	✗
	DCL	✓
	FPAR	✓
	RETURNS	✓
	TIMER	✓
Architecture		✓
		✗
		✓
		✗
		✓
		✗
		✓

Table 1: SDL2IF translation table

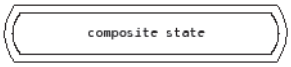
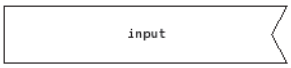
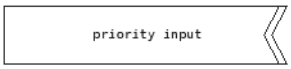
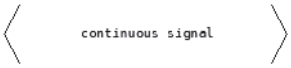
SDL Category	SDL concept	Translated
		✓
		✗
		✓
		✗
		✓
		✓
		✓
		✗
	var:=val	✓
	for, break, continue	✓
	if (statement)	✓
	val bin-op val	✓
	un-op val	✓
	struct!field	✓
	array(index)	✓

Table 1: SDL2IF translation table

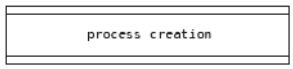
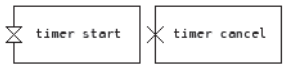
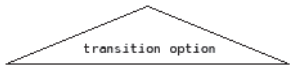
SDL Category	SDL concept	Translated
	operator(params)	✗
	if...then...else...fi	✓
Finite state machine		✓
		✓
		✓
		✗
		✓
		✓
		✓

Table 1: SDL2IF translation table

5.3 - Detailed translation rules

5.3.1 Architecture

SDL supports a hierarchical structure with blocks and processes connected with channels; IF is based on a flat architecture made of processes connected by signalroutes. In order to be exported, the SDL system is "flattened" with one signalroute per message exchanged.

In SDL, the description of the blocks and processes come after the endsystem keyword in the SDL-PR file. In IF, the endsystem is at the end of the file.

It is possible to indicate the initial number of instances of an IF process, but not the maximum number. The maximum number of instances in the SDL system is not translated.

SDL	IF
SYSTEM system_name**;**	**system** system_name;
BLOCK block_name;	No IF equivalent. The SDL system is flattened.
PROCESS process_name(initial_nb, max_nb)	**process** process_name**(**initial_nb**);**

Table 2: Structural mapping between SDL and IF

5.3.2 Communication

Signals, processes, and their associated signal queues are similar concepts in IF and SDL. The SENDER keyword does not exist in IF, so the pid of the instance that sends the signal is explicitly put among the signal parameters. IF and SDL both support the ENV keyword.

SDL	IF
SIGNAL signal_name(param_type);	**signal** sig_name**(**param_type**);**
SIGNALLIST list_name(...);	Not needed; all signal lists are replaced by their contents wherever they appear.
CHANNEL ... & **SIGNALROUTE** ...	**signalroute**, created between the IF processes by flattening SDL channels & signalroutes.
CONNECT ...;	(not needed in IF)

Table 3: Mapping of communication principles from SDL to IF

5.3.3 Behavior

5.3.3.1 States

All SDL states are transformed in stable states in the IF language. The start transition is replaced by a stable state in the IF language with the option `#start` and an automatic transition ("`provided true`"). As in SDL, it is possible to save a message with the save instruction. However, in IF, if a signal is received in a state where it's not expected, the process is blocked; in SDL, the message is discarded. To bypass this problem with the Verimag toolset, the `-discard` option must be used during execution of the system.

Only simple states are supported in IF, so any state symbol with multiple states, *, or *(...) states will make the conversion fail.

A label will be replaced by an unstable state whereas a **JOIN** will be a **nextstate** to one of these states.

SDL	IF
START;	**state** RTDS_Start **#start;** **provided** true; ...;
STATE state_name;	**state** state_name;
SAVE signal_name;	**save** signal_name; NB: saved signals must always appear just after the **state** declaration and before any **input** or **provided**.
label_name: ...	**state** label_name **#unstable ;** ... **endstate**;
JOIN label_name;	**nextstate** label_name;

Table 4: Behavioral mapping between SDL and IF

5.3.3.2 Transitions

As in SDL, IF transitions take no time. The SDL input are translated to IF input with the pid of the sender in parameter in order to support the sender SDL keyword.

IF does not support priority inputs or outputs.

Continuous signals are supported in IF but without any priority.

SDL	IF
INPUT signal_name1; **INPUT** signal_name2(params);	**input** signal_name1(); **input** signal_name2(params);
PROVIDED condition;	**provided** condition**;**

Table 5: Mapping between SDL and IF for transitions

5.3.3.3 Actions

SDL **decisions** are translated to an **unstable states** in IF with a guard (**provided**) in accordance to each branch of the **decision**. The SDL **join** is replaced by a ne**xtstate** to the **unstable** state in accordance to the **label**.

SDL	IF
TASK statement; /* Formal task */	**task** statement;
TASK 'string'; /* Informal task */	**informal** "string";
OUTPUT signal_name1(params); **OUTPUT** signal_name2 **VIA** signal_route; **OUTPUT** signal_name3 **TO** process_name; **OUTPUT** signal_name4 **TO** process_id;	**output** signal_name1(params) **via** signal_route; **output** signal_name2() **via** signal_route; **output** signal_name3() **via** signal_route; **output** signal_name4() **to** process_id; NB: no possibility for undeterministic signal sending in IF; every signal must have one and only one possible receiver. A signal route is created for each signal, so a "**TO** process_name", not available in IF, can sent **via** the signal_route for the signal.
CREATE process_name;	**fork** process_name**();**
STOP;	**stop**;
DECISION cond; (false): /* action 1 */ (true): /* action 2 */ **ENDDECISION**;	**state** etat_decision **#unstable ;** **provided not** cond**;** /* action 1 */ **provided** cond**;** /* action 2 */ **endstate;**

Table 6: Mapping between SDL and IF for actions

5.3.3.4 Timer

Timers are handled differently in SDL and IF. A time out value is set in SDL when a timer is started, and a signal is received when the timer goes off. In IF, a clock is set to 0 and the clock value is checked until it reaches the timeout value. To be able to handle timers started with different values depending on conditions, the IF clock is initialized to a negative value so that the time-out occurs when the clock reaches 0.

SDL	IF
SET(NOW+15,myclock) ... **INPUT** myclock	**set** myclock **:=** -15**;** ... **when** myclock=0**;**
RESET myclock	**reset** myclock**;**

Table 7: Mapping of timers between SDL and IF

A reset instruction is automatically added after each when instruction in IF to avoid combinatorial explosion.

5.3.3.5 Data types

IF defines 5 predefined types : integer, real, boolean, pid and clock. Character and charstring are replaced by string of integers which holds the ASCII value of each character. Constants are not typed. Constants for complex values cannot be translated.

SDL syntypes based on anything else than an integer or a natural cannot translated. If no lower or upper bound can be extracted, the translation also fails.

In SDL, the index type for an ARRAY can be any type. In IF, only indices between 0 and an upper bound are supported. So, only SDL arrays based on an integer index type with 0 or a positive value as lower bound can be translated. All others make the translation fail.

SDL	IF
SYNONYM maxCount integer = 3;	**const** maxCount = 3;
DCL v integer, b boolean, c charstring;	**var** v integer**;** **var** b boolean**;** **var** c RTDS_charstring;
NEWTYPE nouveau **LITERALS** carotte, asperge, haricot; **ENDNEWTYPE**;	**type** nouveau = **enum** carotte,asperge,haricot **endenum**;

Table 8: Mapping of datas types between SDL and IF

SDL	IF
NEWTYPE nouveau **STRUCT** field1 integer; field2 boolean; **ENDNEWTYPE**;	**type** nouveau = **record** field1 integer; field2 boolean; **endrecord**;
SYNTYPE interv = Integer **CONSTANT** 0:4 **ENDSYNTYPE**;	**type** interv = **range** 0 **..** 4**;**
NEWTYPE intTable ARRAY(interv, integer) **ENDNEWTYPE**;	**type** intTable = **array[**4**] of** integer**;**

Table 8: Mapping of datas types between SDL and IF

Variables **PARENT**, **OFFSPRING** and **SENDER** predefined in SDL do not exist in IF. However, it is possible to create these variables:

- **PARENT**: the pid of the parent process is given as a parameter to the **fork**:
  ```
  fork child_process(self, ...); // in the parent process
  ```
  The child process will include a definition for this additional parameter:
  ```
  fpar PARENT pid;
  ```
- **OFFSPRING** : use the **fork** return value:
  ```
  var OFFSPRING pid;
  OFFSPRING := fork child_process(self, ...);
  ```
- **SENDER**: the pid of the sender is added to the signal parameters and retrieved when the signal is received:
  ```
  var SENDER pid;
  input signal1(SENDER, ...);
  ```
  The output for the message will then be:
  ```
  output signal1(self, ...);
  ```

5.3.4 IF observers

IF observers are used to observe the system behavior and verify it is correct.

An observer can be :

- **pure** : only checks the system.
- **cut** : checks the system and can stop it with the action **cut.**
- **intrusive** : checks and can modify the variables, sends signals to other processes.

As a normal process, it has an initial state **#start**, but also has specific state types for observers:

- **#success** indicates a success state, meaning all conditions have been verified;

- **#error** indicates an error state, where one of the conditions is not met.

In the Verimag IFx toolset, the following execution options can be specified :

- `-cs/-ce` to stop the system when it reaches a success or error state (resp.); this allows to avoid explicit **cut** instructions in the observer.
- `-ms/-me` make the executable display a message in its standard output when a success or error state is reached (resp.).

transition:

Three possible events can trigger a state: *match, provided, when*.

- **match** : observe the events:
  ```
  match input signal(pidSender,params) in pidLocal;
  match output signal(pidSender,params) from pidCentral;
  match fork(pidLocal) in pidCreator;
  match kill(pidLocal) in pidCreator;
  ```
  pidSender, pidLocal, pidCentral and pidCreator have previously been declared like pid. For match input and output signal, it is required to declare a variable for each parameter, even if it is not used thereafter.

- **provided** : check the value of variables or the state of a process:
  ```
  provided (process_id).var_name = expression;
  provided (process_id) instate state_name;
  ```
  process_id is written with the process name and its instance number: ({pLocal}0) for example.
  It is possible to gather several of these conditions to make only one transition, for instance:
  ```
  var i integer;
  provided i = 2 and (process_id) instate Idle;
  ```

- **when** : check when a timer reaches the timeout value:
  ```
  when myclock=0;
  ```
  Observers can not check timer from system, it is required to declare a timer in the observer.

action:

- **cut**: stop the system execution;
- **flush**: erase the set of event kept in memory by the observer
  Example :
  ```
  pure observer obs;
    var i integer := 0 ;
    state start #start ;
      match fork(pidLocal) in pidCreator;;
        flush;
        task i := i+1;
        nextstate -;
    endstate;
  endobserver;
  ```
  - without the flush, `i` will be 3 at the end of the execution,

  - with the flush, `i` will be 1.
- **get_queue_length**: return an integer with the queue length for the choosen process
  Example :
  ```
  var queue_Length integer;
  task queue_Length := obs_queue_length(({pLocal}0));
  ```
- **output**: an observer can send a signal to a process of the system, but can not receive any signal.

The observer only activates itself when a process changes of state. For example:

- Process :
  ```
  state start #start ;
    task i := i+1;
    task i := i+1;
    task i := i+1;
    task i := i+1;
    nextstate state2;
  endstate;
  ```
- Observer :
  ```
  state start_obs #start ;
    provided i = 3;
    nextstate obs_state2;
  endstate;
  ```

The observer will never execute the transition since it will only be activated when the process reaches `state2`, where `i` is 4 and not 3.

6 - SDL to C translation rules

This section describes how SDL code is translated to C. This translation happens when producing an executable for a SDL system (see “SDL and SDL-RT code generation” on page 132) and when converting a SDL project to a SDL-RT one.

This section focuses on the translation of the actual code, i.e the declarations, statements and expressions found in the symbols in the diagrams. For the parts specific to the code generation or the SDL to SDL-RT conversion, refer to the corresponding sections.

6.1 - Conversion guidelines for declarations

6.1.1 SYNTYPE declaration

SYNTYPE declarations are converted to:

- Two constants definitions, one for the type’s lower bound, the other for the type’s upper bound;
- A `typedef` for the type itself.

For example:

```
SYNTYPE MyIndexType = INTEGER
  CONSTANTS 0:7
ENDSYNTYPE;
```

will be converted to:

```
#define RTDS_MYINDEXTYPE_MIN 0
#define RTDS_MYINDEXTYPE_MAX 7 + 1
typedef int MyIndexType;
```

The constants are used when the `SYNTYPE` is used as an index type for an `ARRAY`; see “NEWTYPE ... Array(...) declaration” on page 112.

Notes:

- The `SYNTYPE` declarations are supposed to be there for array indices. So a `SYNTYPE` based on anything else than `INTEGER` or `NATURAL` is not likely to produce any usable result.
- The range for the `SYNTYPE` should be closed and continuous. For example, a `SYNTYPE` defined with "`CONSTANTS > -4, /= 0, < 4`" will ignore the "`/= 0`" constraint and be used exactly like a `SYNTYPE` with "`CONSTANTS -4:4`". If no lower or upper bound is specified, as in "`CONSTANTS < 10`" or "`CONSTANTS >= -2`", they will be forced respectively to 0 and 1.
- `SYNTYPE` definitions based on another `SYNTYPE` are handled:
  `SYNTYPE t1 = t2 CONSTANTS < 3 ENDSYNTYPE;`
  will be translated to:
  ```
  #define RTDS_T1_MIN RTDS_T2_MIN
  #define RTDS_T1_MAX 3
  typedef t2 t1;
  ```

6.1.2 NEWTYPE declarations

The actual conversion depends on the actual type kind defined. The exact conversion is described in the following paragraphs. The conversion performed for the operators defined for the type is described in “OPERATORS conversion” on page 114.

6.1.2.1 NEWTYPE ... STRUCT declaration

SDL `STRUCT` types are converted to C `struct` types. For example:

```
NEWTYPE MyStructType
STRUCT
  i INTEGER;
  c CHARACTER;
  x MyOtherType;
ENDNEWTYPE;
```

will be converted to:

```
typedef struct _MyStructType
  {
  int i;
  char c;
  MyOtherType x;
  } MyStructType;
```

Notes:

- Pointers are never used: in the example above, the conversion would be the same if `MyOtherType` was a complex type such as a struct, a choice or an array. This prevents from having to manage dynamic allocation for variables.
- The correct order is ensured by the conversion process: In the example above, the struct for `MyStructType` will always be created after the declaration of `MyOtherType`, even if the SDL declarations were in the reverse order.
- If an optional field `x` is present in the `STRUCT`, an additional field `xPresent` with the type `RTDS_BOOLEAN` is automatically added just after it. The generated code then handles this field to reflect the field presence.
- This is the same C representation as for ASN.1 `SEQUENCE` types; See “Type mapping” on page 17.

6.1.2.2 NEWTYPE ... CHOICE declaration

SDL `CHOICE` types are converted to two C types:

- An `enum` type for each of the fields in the `CHOICE`;
- A `struct` type containing:
  - A `present` field with the `enum` type defined above;
  - A `__value` field with a `union` type containing all the fields in the `CHOICE`.

For example:

```
NEWTYPE MyChoiceType
CHOICE
  i INTEGER;
  c CHARACTER;
  x MyOtherType;
ENDNEWTYPE;
```

will be converted to:

```
typedef enum _t_MyChoiceType
  {
  MyChoiceType_i,
  MyChoiceType_c,
  MyChoiceType_x
  } t_MyChoiceType;
typedef struct _MyChoiceType
  {
  t_MyChoiceType present;
  union _MyChoiceType_choice
     {
     int i;
     char c;
     MyOtherType x;
     } __value;
  } MyChoiceType;
```

Notes:

- The first 3 notes in “NEWTYPE ... STRUCT declaration” on page 111 also apply.
- This is the same C representation as for ASN.1 `CHOICE` types; See “Type mapping” on page 17.

6.1.2.3 NEWTYPE ... Array(...) declaration

The `ARRAY` generator has two parameters: an index type and an element type. The index type must be a `SYNTYPE` based on `INTEGER` or `NATURAL`. The declaration is then converted to a C array `typedef`, using the constants defined for the index type’s lower and upper bounds. For example:

```
SYNTYPE IndexType = INTEGER
  CONSTANTS 0:7
ENDSYNTYPE;
NEWTYPE ArrayType
  ARRAY(IndexType, NATURAL)
ENDNEWTYPE;
```

will be converted to:

```
#define RTDS_INDEXTYPE_MIN 0
#define RTDS_INDEXTYPE_MAX 7 + 1
typedef int IndexType;
typedef unsigned int ArrayType[RTDS_INDEXTYPE_MAX - RTDS_INDEXTYPE_MIN];
```

This ensures that enough entries in the array are created, even if the lower bound for its index type is negative. The offset in indices in this case is handled when a variable with this type is used; see “Variable declarations” on page 116.

Notes:

- Pointers are never used, even if the array element is a complex type such as a struct, choice or another array.
- If the array in the SDL declarations is declared before its index or element type, C declarations are re-ordered so that it appears after both of them.

6.1.2.4 NEWTYPE ... Bag(...) declaration

The BAG generator has a single parameter which is the type for the bag element. The declaration is converted to a C typedef with the same name as the SDL type, which is not intended to be directly manipulated in the code. All manipulations on a BAG type *T* are done via the following macros and functions:

- RTDS_SET_OF_INIT(<*bag variable*>) initializes the variable to an empty bag;
- RTDS_SET_OF_*T*_COPY(<*bag var. 1*>, <*bag var. 2*>) copies the first bag variable to the second one;
- RTDS_SET_OF_*T*_INCL(<*bag var. 1*>, <*bag var. 2*>, <*element var.*>) adds the element to the second bag variable and puts the result in the first bag variable;
- RTDS_SET_OF_*T*_DEL(<*bag var. 1*>, <*bag var. 2*>, <*element var.*>) removes the element from the second bag variable and puts the result in the first bag variable;
- RTDS_SET_OF_*T*_TAKE(<*bag variable*>) returns a random element from the bag variable;
- RTDS_SET_OF_LENGTH(<*bag variable*>) returns the length of the bag;
- RTDS_setOf_*T*_cmp(<operator>, <bag var. 1>, <bag var. 2>) compares the two bags for equality and/or inclusion; <*operator*> is defined by the enum type RTDS_setOfCompareOperator and contains constants for the traditionnal comparison operators (==, !=, <, <=, >, >=);
- RTDS_setOf_*T*_in(<*element variable*>, <*bag variable*>) tests if the element is in the bag and returns a boolean;
- RTDS_SET_OF_*T*_FIRST(<*bag variable*>, <*element variable*>) initializes a loop over the bag, puts its first element in <*element variable*> if any, and returns true if the bag is not empty or false if it is;
- RTDS_SET_OF_*T*_NEXT(<*bag variable*>, <*element variable*>) continues a loop over the bag initialized with RTDS_SET_OF_*T*_FIRST; It puts the next bag element in <*element variable*> if any and returns true if there actually was a next element, or false if there are no more element to iterate over.

Notes:

- Bags are never modified 'in place': The result of an operation on a bag is always put in another bag.
- Pointers are never used: If an element is added to a bag, a modification on the variable used in the addition will not be reflected on the bag element.
- This is the same C representation as for ASN.1 SET OF types; See "Type mapping" on page 17.

6.1.2.5 NEWTYPE ... String(...) declaration

The STRING generator has a single parameter which is the type for the string element. A declaration for a type StringType with elements of type ElementType is converted to the following C typedef:

```
typedef struct _StringType
  {
  ElementType   elements[<size>];
  unsigned int  length;
  } StringType;
```

If a size constraints is specified in the NEWTYPE, <*size*> is set according to it. If no size constraint is specified, <*size*> is set to RTDS_MAX_STRING.

Note: This is the same C representation as for ASN.1 SEQUENCE OF types; See "Type mapping" on page 17.

6.1.2.6 NEWTYPE ... LITERALS declaration

SDL LITERALS type are converted to their equivalent enum types in C. For example:

```
NEWTYPE MyLiteralsType
  LITERALS literal1, literal2, literal3;
ENDNEWTYPE;
```

will be converted to:

```
typedef enum _MyLiteralsType
  {
  <prefix>literal1,
  <prefix>literal2,
  <prefix>literal3
  } MyLiteralsType;
```

where <*prefix*> can be a standard prefix for all constants and/or the name of the type, depending on the generation options.

Notes:

- The LITERALS clause is not allowed in STRUCT, CHOICE or generator-based types: It must be alone in the NEWTYPE declaration.
- This is the same C representation as for ASN.1 ENUMERATED types; See "Type mapping" on page 17.

6.1.2.7 OPERATORS conversion

An operator defined in a SDL type is translated to a function with the same name in C. Each parameter for the operator is mapped to a parameter with the same name in C; the parameter C type is the translation of the parameter's SDL type, except for structures and choices, where it becomes a pointer on this type.

For the return value, an additional parameter is added as the last one in the C function which is always a pointer on the operator's return type, even for simple types. The function also returns a pointer on the return value, which may or may not be the pointer passed as last parameter. At each call of the function, a pointer on a pre-allocated area for the return value will be passed. The function can use this area if it needs to. The actual return value will be the one returned by the function, and not the contents of the pre-allocated area. This allows to return a pointer on a static or global variable for example.

The C function is always declared as extern and no implementation is generated.

For example:

```
NEWTYPE Point
STRUCT
  x, y REAL;
OPERATORS
  newPoint: REAL, REAL -> Point;
  movePoint: Point, REAL, REAL -> Point;
  dist0: Point -> REAL;
ENDNEWTYPE;
```

will be converted to:

```
typedef struct _Point
  {
  float x, y;
  } Point;
extern Point * newPoint(float, float, Point*);
extern Point * movePoint(Point*, float, float, Point*);
extern float * dist0(Point*, float*);
```

Note: If an operator uses a type, its C translation is always generated after the used type's one, even if the SDL declarations were the other way. This may lead to defining the operators quite far from the type defining them in the SDL declarations.

6.1.3 SYNONYM declaration

Synonyms are usually converted to `#define` directives in the SDL-RT project. For example:

```
SYNONYM MyConstant INTEGER = 5;
```

will be translated to:

```
#define MyConstant 5
```

Please note the constant name is not upper cased, and that the constant type is not used.

Synonyms are however converted differently when their type is a complex type such as a structure or an array. In this case, they will be converted to a `#define` directive, plus a variable definition. For example:

```
SYNONYM MyStructuredConstant MyStructType = (. 2, 'foo' .);
```

will be translated to:

```
#define _MyStructuredConstant_ { 2, "foo" }
MyStructType MyStructuredConstant = _MyStructuredConstant_;
```

The generated variable is automatically declared `static` if in the context of a header file. An option allows to also declare it `const` if the compiler allows it. This translation allows to use the synonym in all C contexts, including variable initializers and as values in expressions. For example, with the declaration above:

```
DCL st MyStructType := MyStructuredConstant;
```

will be converted to:

```
MyStructType st = _MyStructuredConstant_;
```

but:

```
x := MyStructuredConstant!field1
```

will be converted to:

```
x = MyStructuredConstant.field1;
```

The same conversion is performed for arrays:

```
SYNONYM IndexType = INTEGER
  CONSTANTS 0:7
ENDSYNONYM;
NEWTYPE ArrayType
  ARRAY(IndexType, REAL)
ENDNEWTYPE;
SYNONYM ArrayConstant ArrayType = (. 3.14 .);
```

will be converted to:

```
#define RTDS_INDEXTYPE_MIN 0
#define RTDS_INDEXTYPE_MAX 7 + 1
typedef int IndexType;
typedef float ArrayType[RTDS_INDEXTYPE_MAX - RTDS_INDEXTYPE_MIN];
#define _ArrayConstant_ { 3.14 }
ArrayType ArrayConstant = _ArrayConstant_;
```

Note that the semantics of the C type is different from the SDL one: initializing a SDL array initializes all its elements with the given value; the initialization for the C array will only set the array's first element. However, whenever possible, a dynamic initialisation will be performed for the array setting all its elements to the given value. This initialisation may be performed:

- At the beginning of the process's initial transition if the array variable is in a process;
- In the middle of the code if an array is set to a `(. … .)` constant in a task block;
- During system start for a `SYNONYM`. This will mainly happen for SDL to C/C++ code generation, not for SDL to SDL-RT conversion.

6.1.4 Variable declarations

A variable declaration in SDL is usually converted to its direct equivalent in C. For example:

```
DCL i INTEGER;
```

will be converted to:

```
int i;
```

The built-in conversion for the base SDL types are:

SDL base type	C type	Comment
`INTEGER`	`int`	
`NATURAL`	`unsigned int`	
`REAL`	`double`	
`BOOLEAN`	`RTDS_BOOLEAN`, defined as an `enum`	The enum type defines the constants `TRUE` and `FALSE` to 1 and 0 respectively. Its definition is in the file `RTDS_CommonTypes.h`, in `$RTDS_HOME/share/ccg/common`.
`CHARACTER`	`char`	
`CHARSTRING`	`RTDS_String`	Custom type used to represent a string. Functions handling this type are declared in `RTDS_String.h` and `RTDS_String.c`, automatically included in every converted project.
`PID`	`RTDS_PID`	Type for process identifiers in all SDL-RT profiles.

SDL base type	C type	Comment
TIME	long	
DURATION	long	

If an initializer is present, it is copied for each declared variable. For example:

```
DCL x, y REAL := 0.0;
```

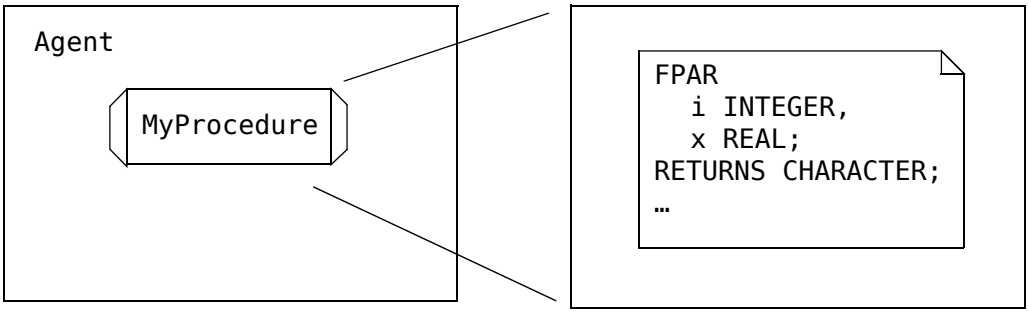
will be converted to:

```
float x = 0.0, y = 0.0;
```

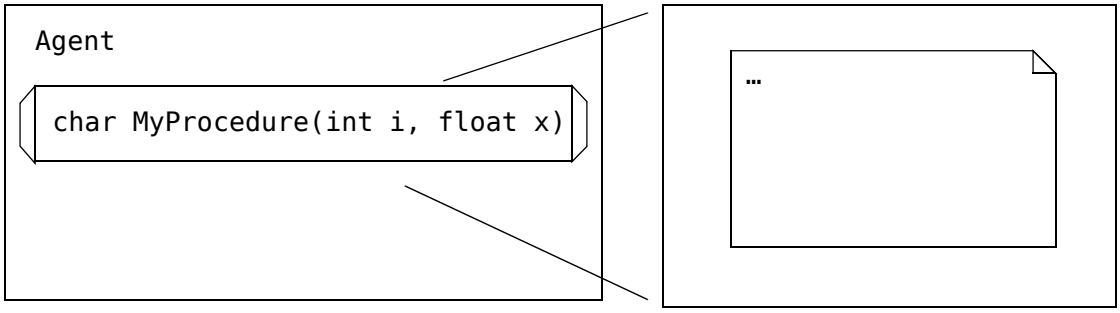
6.1.5 FPAR and RETURNS declarations

In processes, the SDL FPAR declaration is dropped since process parameters are not supported in SDL-RT.

In procedures, SDL FPAR and RETURNS declarations are inserted in the procedure declaration symbol declaration in the procedure's parent, and removed from the procedure diagram. For example:



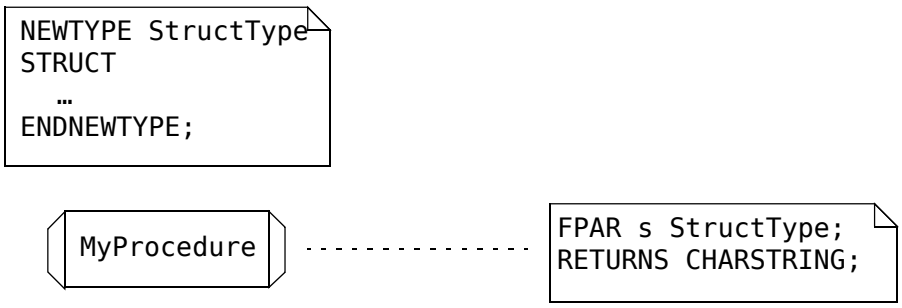
will be translated to:



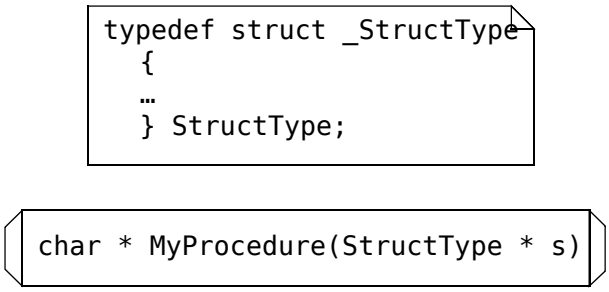
If the SDL procedure takes as parameter or returns a structure or a choice, its SDL-RT counterpart will respectively take as parameter or return a pointer on this structure or choice.

If the SDL procedure returns a CHARSTRING or an array, is SDL-RT counterpart will respectively return a char* or a pointer on the array element.

For example:



will be translated to:



All these type changes are considered in the procedure or caller body. In the example, getting field `s!x` in the `MyProcedure`'s body will be translated to `(*s).x`.

6.1.6 Other declarations

The following declarations: `INHERITS`, `USE`, `SIGNAL`, and `SIGNALLIST` do not have any direct translation to C code.

SDL `TIMER` declarations are ignored. If a value is specified in the declaration, it is repeated in each timer set symbol that does not include an explicit time-out in the generated C code.

External procedure declaration is translated as the prototype of the C function implementing the procedure. The code is generated as for the operators (See “OPERATORS conversion” on page 114) except that IN/OUT and OUT parameters are always generated as pointers.

6.2 - Conversion guidelines for statements and expressions

6.2.1 Assignment statements

These are the only "true" statements that can happen in SDL task blocks. They are usually converted to a C assignment, except in the following cases:

- For strings, the function `RTDS_StringAssign` is used: `s1 := s2` will be converted to `RTDS_StringAssign(s1, s2);`. This function is defined in the files `RTDS_String.h` and `RTDS_String.c`.

- For structures, choices or arrays, the standard C function `memcpy` is used. So if `x` and `y` are structures, choices or arrays with type `T`, `x := y` will be translated to: `memcpy(&(x), &(y), sizeof(T));`

Note: One or both `&` may not appear if the corresponding variable is already an address (array, `IN/OUT` parameter for a procedure, nested scopes, ...).

6.2.2 Booleans operations

The following conversions are made for boolean operations:

SDL expression	C expression	Comment
`b1 AND b2`	`b1 && b2`	
`b1 OR b2`	`b1 || b2`	
`b1 XOR b2`	`b1 != b2`	False only if `b1` and `b2` are both false or both true
`b1 => b2`	`!(b1) || b2`	False only if `b1` is true and `b2` is false

6.2.3 Numeric operations

The conversion for numeric operations is trivial for operators `+`, `-` and `*`. The `/` operation is converted to the C `/`, which alters its semantics when both operands are integers, with the first positive and the second negative: the result in SDL is then positive where the result in C is negative.

Both SDL operators `mod` and `rem` are also translated to the C operator `%`. This is actually only valid for the `rem` operator, so the semantics for `mod` is altered.

6.2.4 Character string operations

The SDL character string concatenation operator `//` is translated to:

- The declaration of a temporary C character string used to store the results of the concatenation;
- The call to the function `RTDS_StringCat` with 3 parameters: the temporary string defined above, then the two strings to concatenate.

For example, the statement:

```
s := 'foo' // 'bar'
```

will be converted to:

```
{
RTDS_String RTDS_tempString1;
RTDS_StringAssign(s, RTDS_StringCat(RTDS_tempString1, "foo", "bar"));
}
```

The `RTDS_StringCat` function is defined in the files `RTDS_String.h` and `RTDS_String.c`.

Note: This means using the `//` operator in a symbol that can only contain an expression will not work. For example, testing `length(strVar // 'foo') >= 12` in a decision will be translated to a C code that is not an expression.

6.2.5 String(...) types operations

The operator `//` can also be used to concatenate any `String(…)` type. The translation is then similar to the one for the `//` for character strings described above, but uses the macro `RTDS_SEQUENCE_CONCAT` instead of the function `RTDS_StringCat`.

For example:

```
DCL a, b, c MyStringType;
…
a := b // c;
```

is converted to:

```
MyStringType a, b, c;
…
MyStringType tmp_string;
RTDS_SEQUENCE_CONCAT(&tmp_string, &b, &c);
memcpy(&a, &tmp_string);
```

6.2.6 Comparison operations

When both operands for comparison operators are booleans, integers, reals, times or durations, the SDL comparison operators are directly translated to their C equivalent.

For comparison operations on character strings, the C function `strcmp` is used: `s1` *op* `s2` is simply translated to `strcmp(s1, s2)` *op′* `0`, *op′* being the C translation of *op*.

For comparison operations on structures, choices, arrays or strings, a C function named `RTDS_<`*type name*`>_cmp` is generated, returning an integer having the same meaning as the standard C function `memcmp`: If variables `a` and `b` have the type `T`, `a` *op* `b` is translated to `RTDS_T_cmp(&a, &b)` *op′* `0`, *op′* being the C translation of *op*. In this case, *op* can only be `=` or `/=` (translated respectively to `==` and `!=`). All comparison functions are declared just after the type itself, and all are implemented in a single generated file called `RTDS_comp_functions.c`.

For comparisons on bag types, the function is actually named `RTDS_setOf_<`*type name*`>_cmp` since it has a different signature and can test for inclusions too. It is described in paragraph “NEWTYPE ... Bag(...) declaration” on page 113.

6.2.7 Conditional operator

The SDL conditional operator `IF cond THEN expr1 ELSE expr2 FI` is converted to its C equivalent `((cond') ? (expr1') : (expr2'))`, where `cond'`, `expr1'` and `expr2'` are the C translations for `cond`, `expr1` and `expr2` respectively. Note that types for the expressions is not considered, so using structures, choices or arrays in `expr1` or `expr2` will generate invalid C code.

6.2.8 Field extraction

A field extraction `s!x` is usually converted to its C equivalent `s.x`, except in the following cases:

- If `s` is a variable passed as an `in/out` parameter to a procedure, it is passed as a pointer. So the field extraction becomes `(*s).x`;
- If `s` is a variable inherited from an enclosing scope, it is also accessed via a pointer (see “Nested scopes management” on page 122), so the field extraction also becomes `(*s).x`;
- If `s` is a choice:
  - If `x` is `present`, the conversion is normal: `s.present`;
  - If `x` is not `present`, the `__value` sub-structure is added in the middle and the field extraction becomes `s.__value.x`. See “NEWTYPE ... CHOICE declaration” on page 111.

Note that a field extraction via `s(x)` instead of `s!x` is not recognized.

6.2.9 Array indexing

The conversion for an array indexing `a(i)` depends on the index type for the array:

- If the index type’s lower bound is 0, the translated code is `a[i]`.
- If the index type’s lower bound is not 0, the translated code is:
  `a[i - RTDS_`*`IndexType`*`_MIN]`
  where *`IndexType`* is the name of the array index type.

Note that this will happen for all expressions with the format `x(y)` if `x` is not a declared operator.

6.2.10 Procedure and operator calls

Procedure and operator calls both generate a call to a C function with the same name as the procedure or operator, passing structure or choice parameters by address. In addition, for procedure calls:

- Parameters defined as `IN/OUT` are always passed by address.
- The additional parameter `RTDS_localsStack` is always passed to the procedure except for external ones (see “Nested scopes management” on page 122).
- When in task blocks:
  - The `CALL` keyword is removed;
  - The parameter `RTDS_currentContext` is automatically added, as required by the SDL-RT semantics expect for external ones.

6.2.11 Inline values for structures or arrays

Inline values for structures are `(. field1, field2, … .)`, and `(. element .)` for arrays. These values can only be converted when in the context of a variable assignment. In this case, the translation consists in the declaration of a temporary variable with the structure or array type at the beginning of the parent task block, initialized with the given value. The value itself is then replaced by this temporary variable. For example:

```
myStruct := (. 1, 'foo' .)
```

where the type for `myStruct` is `MyStruct_t` is translated to:

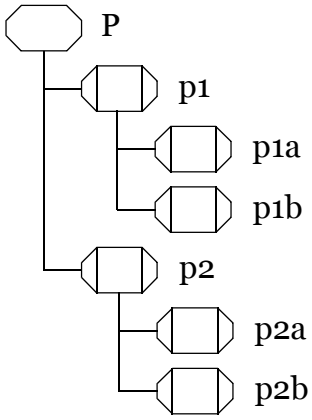
```
{
MyStruct_t RTDS_MyStruct_t_CONSTANT1 = { 1, "foo" };
memcpy(&(myStruct), &(RTDS_MyStruct_t_CONSTANT1), sizeof(MyStruct_t));
}
```

Note: this means that expressions in an inline structure or array initializer must always be constants: if a variable is used, its value may change between the beginning of the task block and where the initializer is actually used. This is consistent with the semantics used for the SDL simulator.

6.3 - Nested scopes management

6.3.1 Problem

In SDL, if a procedure is defined in a process or another procedure, it may see or modify all variables defined in its parent, recursively. For example, with the following architecture:



each procedure may modify the following variables:

Procedure:	may access variables in:
p1	P, p1
p1a	P, p1, p1a
p1b	P, p1, p1b
p2	P, p2
p2a	P, p2, p2a
p2b	P, p2, p2b

Since C function declarations cannot be nested, this feature has no equivalent in C, so it must be implemented explicitly. Moreover, one has to consider that procedures may call

any other procedure if the latter has been defined in the caller procedure or any of its ancestors. In the example:

Procedure:	may call:
p1	p1a, p1b, p1, p2
p1a	p1a, p1b, p1, p2
p1b	p1b, p1a, p1, p2
p2	p2a, p2b, p2, p1
p2a	p2a, p2b, p2, p1
p2b	p2b, p2a, p2, p1

So if the process calls p1, which in turn calls p1a, which calls p2, which calls p2b, p2b should still be able to access any variable in p2 and P, even if a variable with the same name exists in p1 or p1a.

6.3.2 C implementation

6.3.2.1 General case

The implementation chosen in C to allow this behavior is to manage explicitly a secondary call stack containing pointers on process or procedure's local variables:

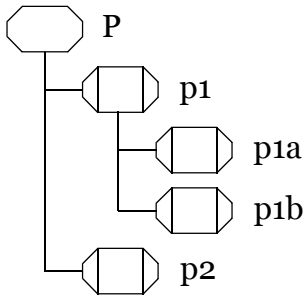
- Each process or procedure stores the addresses of all its local variables in a C array named `RTDS_<proc. name>_myLocals`, declared as an array of `void*`. The variables are sorted by name to ensure a consistent order across scopes (there is an exception to this rule for specialized process classes; see "Specialized process classes" on page 127).
  For example, if a process or procedure declares the local variables:
  ```
  int i, j;
  MyStructType st;
  RTDS_String msg;
  ```
  the following additional declarations and statements will be generated:
  ```
  void * RTDS_myLocals[4];
  RTDS_myLocals[0] = (void*)&i;
  RTDS_myLocals[1] = (void*)&j;
  RTDS_myLocals[2] = (void*)&msg;
  RTDS_myLocals[3] = (void*)&st;
  ```
- All these arrays are organized into a stack that is filled from the top-level process or procedure. This stack is stored in an C array called `RTDS_localsStack` and declared as an array of `void**`. This array is then passed to every procedure as the last parameter under the name `RTDS_inhLocalsStack`.
  For example, for a procedure `p1` with no parameters or return value in the process `P`, `p1` is actually defined as:
  ```
  void p1(void *** RTDS_inhLocalsStack)
  ```
  and the procedure declares:
  ```
  void ** RTDS_localsStack[2];
  ```
  which is initialized by:

```
RTDS_localsStack[0] = RTDS_inhLocalsStack[0];
RTDS_localsStack[1] = RTDS_myLocals;
```

where `RTDS_myLocals` is the array referencing the local variables in `p1`.
When `p1` is called from the process, the array passed in `RTDS_inhLocalsStack` only contains the element for `P`'s local variables, enabling the procedure to access them.
When `p1` calls any other procedure, it will pass in its `RTDS_inhLocalsStack` the array containing `P`'s variables and the array containing its own variables. So if the called procedure is defined in `p1`, it will be able to access `p1`'s and `P`'s variables. If the called procedure is defined in `P`, it will be able to access `P`'s variables.

- All inherited variables in procedure are declared as pointers, initialized with the corresponding element in the corresponding stack entry. If the variable is shadowed by another variable with the same name in an enclosed scope, it is simply not defined.

Here is a detailed example, with the following architecture:



The variables declared in the process and procedures are:

Process / procedure	Variables
P	`int i, j;`
p1	`int n;`
p1a	`int j, k;`
p1b	`int m;`
p2	`float x, y;`

The additional declarations and statements in the process and procedures are:

- In process `P`:

```
void * RTDS_myLocals[2];
void ** RTDS_localsStack[1];
RTDS_myLocals[0] = (void*)&i;
RTDS_myLocals[1] = (void*)&j;
RTDS_localsStack[0] = RTDS_myLocals;
```

- In procedure `p1` (declared with additional parameter `void *** RTDS_inhLocalsStack`):

```
void * RTDS_myLocals[1];
void ** RTDS_localsStack[2];
int * i;
int * j;
```

```
RTDS_myLocals[0] = (void*)&n;
RTDS_localsStack[0] = RTDS_inhLocalsStack[0];
RTDS_localsStack[1] = RTDS_myLocals;
i = (int*)(RTDS_inhLocalsStack[0][0]);
j = (int*)(RTDS_inhLocalsStack[0][1]);
```

When `p1` is called, `RTDS_inhLocalsStack[0]` is always the `RTDS_myLocals` array declared by process `P`. So `P`'s variables `i` and `j` may be accessed in `p1` via `*i` and `*j`.

- In procedure `p1a` (declared with additional parameter `void *** RTDS_inhLocalsStack`):

```
void * RTDS_myLocals[2];
void ** RTDS_localsStack[3];
int * i;
int * n;
RTDS_myLocals[0] = (void*)&j;
RTDS_myLocals[1] = (void*)&k;
RTDS_localsStack[0] = RTDS_inhLocalsStack[0];
RTDS_localsStack[1] = RTDS_inhLocalsStack[1];
RTDS_localsStack[2] = RTDS_myLocals;
i = (int*)(RTDS_inhLocalsStack[0][0]);
n = (int*)(RTDS_inhLocalsStack[1][0]);
```

`j` is not declared as in `p1` since it is shadowed by variable `j` in `p1a`. Again, `RTDS_inhLocalsStack[0]` is always `P`'s `RTDS_myLocals`, and `RTDS_inhLocalsStack[1]` is always `p1`'s `RTDS_myLocals`, allowing access to their respective local variables.

- In procedure `p1b` (declared with additional parameter `void *** RTDS_inhLocalsStack`):

```
void * RTDS_myLocals[1];
void ** RTDS_localsStack[3];
int * i;
int * j;
int * n;
RTDS_myLocals[0] = (void*)&m;
RTDS_localsStack[0] = RTDS_inhLocalsStack[0];
RTDS_localsStack[1] = RTDS_inhLocalsStack[1];
RTDS_localsStack[2] = RTDS_myLocals;
i = (int*)(RTDS_inhLocalsStack[0][0]);
j = (int*)(RTDS_inhLocalsStack[0][1]);
n = (int*)(RTDS_inhLocalsStack[1][0]);
```

Same thing as in `p1a`, but `j` is not shadowed by a local variable, so it is extracted from the first entry in the stack, corresponding to process `P`.

- In procedure `p2` (declared with additional parameter `void *** RTDS_inhLocalsStack`):

```
void * RTDS_myLocals[2];
void ** RTDS_localsStack[2];
int * i;
int * j;
RTDS_myLocals[0] = (void*)&x;
RTDS_myLocals[1] = (void*)&y;
RTDS_localsStack[0] = RTDS_inhLocalsStack[0];
RTDS_localsStack[1] = RTDS_myLocals;
i = (int*)(RTDS_inhLocalsStack[0][0]);
j = (int*)(RTDS_inhLocalsStack[0][1]);
```

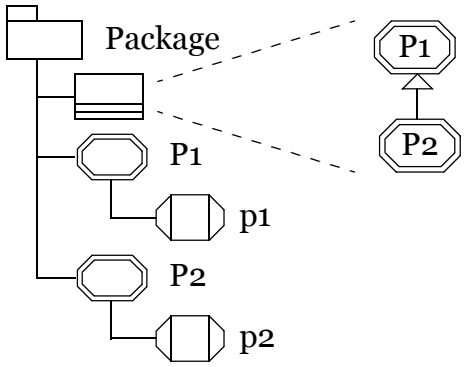
Here is the contents of the RTDS_inhLocalsStack additional parameter passed to each procedure when called by the process or another procedure:

Caller	Called	Variables in RTDS_inhLocalsStack		
		Index 0	Index 1	Index 2
P	p1	P	////	////
P	p2	P	////	////
p1	p1	P	p1	////
p1	p2	P	p1	////
p1	p1a	P	p1	////
p1	p1b	P	p1	////
p1a	p1	P	p1	p1a
p1a	p2	P	p1	p1a
p1a	p1a	P	p1	p1a
p1a	p1b	P	p1	p1a
p1b	p1	P	p1	p1b
p1b	p2	P	p1	p1b
p1b	p1a	P	p1	p1b
p1b	p1b	P	p1	p1b
p2	p1	P	p2	////
p2	p2	P	p2	////

//// denotes that there is no element in RTDS_inhLocalsStack at this index. The name in the "Index" columns is the name of the process or procedure whose variables are stored in the specified element. The greyed cells with bold text emphasize the entries that are required in a given context. For example, when p1a is called, it must be able to access variables from two levels above, i.e. P's and p1's variables. The table shows that all required entries are always correctly set.

6.3.2.2 Specialized process classes

An additional problem appears when a process class is specialized, since a procedure called in the sub-class can access variables defined in the super-class. Here is an example architecture:



In this case:

- Procedure `p2` has access to all variables defined for process class `P2`, including those inherited from `P1`;
- Process class `P2` may call procedure `p1`, which has access to variables inherited from `P1`, but not to `P2`'s own local variables.

To ensure correct behavior, the variables in the `P1`'s and `P2`'s arrays `RTDS_myLocals` are ordered first by inheritance level, then by variable name. This ensures that procedures that may access only variables defined in the super-class will always find them at the same index in their `RTDS_inhLocalsStack` parameter.

In the example, if the variables declared in the processes are:

Process	Variables
P1	`int i, j;`
P2	`short b;`

the definitions and initialisations for the arrays for nested scope management will be:

Process	Array declarations

P1	`void * RTDS_myLocals[2];`
	`void ** RTDS_localsStack[1];`
	`RTDS_myLocals[0] = (void*)&i;`
	`RTDS_myLocals[1] = (void*)&j;`
	`RTDS_localsStack[0] = RTDS_myLocals;`

P2	`void * RTDS_myLocals[2];`
	`void ** RTDS_localsStack[1];`
	`RTDS_myLocals[0] = (void*)&i;`
	`RTDS_myLocals[1] = (void*)&j;`
	`RTDS_myLocals[2] = (void*)&b;`
	`RTDS_localsStack[0] = RTDS_myLocals;`

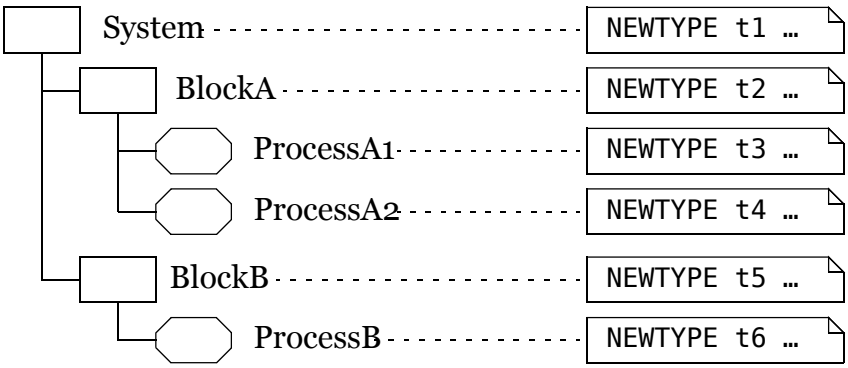
Note that the address for variable `b`, defined in `P2`, is inserted after the addresses for the variables `i` and `j` inherited from `P1`. By doing so, we ensure that addresses for variables `i` and `j` are always found at indices `0` and `1` in the array, allowing procedures `p1` and `p2` to access them in a consistent way in their `RTDS_inhLocalsStack` parameter.

7 - SDL to SDL-RT conversion

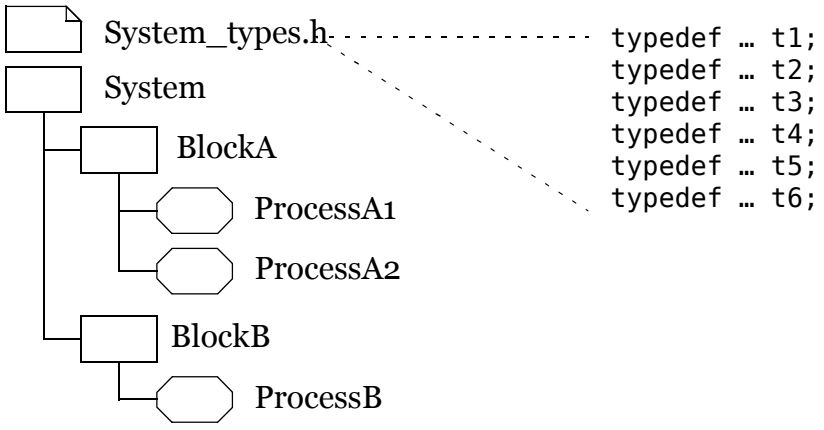
7.1 - Project tree

The conversion of project nodes in the project tree is generally one to one: A package node will be converted to a package node, a diagram node to a diagram node, and so on... The following exceptions apply:

- Two special files named `RTDS_String.h` and `RTDS_String.c` are always added to the converted project. These files define and implement the functions required to handle strings in converted diagrams.
- If an agent hierarchy contains type, syntype or synonym declarations, these declarations are not inserted in the diagrams, but in a special C header file at the same level as the top-level agent in the hierarchy. All type declarations are gathered in this file, wherever they appear. For example:



will be converted to:



The corresponding type definitions will be removed from the corresponding symbols and a new text symbol will be inserted in the top-level diagram containing a `#include` for the created header file (see “Diagrams” on page 130).
This mechanism ensures that all types defined at all levels in the hierarchy will be seen in all diagrams. This also allows to re-order the declarations to ensure that types are always declared after the types they depend on, as required by the C semantics. For more details, see “Conversion guidelines for declarations” on page 110.

- A SDL declarations file node may be split into two file nodes: A C header file and a SDL-RT declarations file. For further details, see “Files” on page 130.

7.2 - Files

Declaration files are translated as explained in “Conversion guidelines for declarations” on page 110. The conversion result is:

- A C header file containing the standard import of the `RTDS_String.h` file and the translation for all type, syntype and synonym declarations;
- Optionally a SDL-RT declarations file containing the `MESSAGE` and `MESSAGE_LIST` declarations corresponding to the `SIGNAL` and `SIGNALLIST` declarations in the original file.

If any C source, C header or external file is present in the SDL project, it is copied to the translated SDL-RT project without any conversion.

7.3 - Diagrams

The general principles for diagram conversion are:

- Partitions are converted one to one: Each partition in the diagram will have an equivalent in the converted one;
- Symbols in partitions are also converted one to one whenever possible.

There are exceptions to these rules, depending on the original diagram type. They are detailed in the following paragraphs, along with the actual conversions performed.

7.3.1 SDL diagrams

The following exceptions apply for SDL diagrams:

- The conversions performed for declarations and statements are respectively described in “Conversion guidelines for declarations” on page 110 and “Conversion guidelines for statements and expressions” on page 118. The latter paragraph also describes how are converted the expressions that may appear in other symbols (e.g. message parameters in message outputs, procedure parameters in procedure calls, etc...). Apart from declarations, statements and expressions, the conversion is usually quite straightforward, except in the following cases:
  - In architecture diagrams, cardinalities for blocks are ignored.
  - In behavioral diagrams, virtualities (`VIRTUAL`, `REDEFINED`, `FINALIZED`) is start, input, save or continuous signal symbols are ignored.
  - In behavioral diagrams, message outputs `TO <receiver id>` are converted either to a `TO_NAME` or a `TO_ID` depending on the existence of a variable named `<receiver id>`.
  - Parameters for processes are ignored: the `FPAR` declaration will disappear from the process diagram and the parameters will be removed from the process creation symbols.
- `NEWTYPE`, `SYNTYPE` and `SYNONYM` declarations in text symbols are copied to a global C header file as explained in “Project tree” on page 129. If no other declaration is present in the symbol, it will not appear in the converted diagram.

- An additional text symbol will appear in all top-level SDL diagrams (system, block class and process class). It will contain the `#include` directive for the global C header file for types.
- Some declarations in SDL text symbol will be translated to SDL-RT declarations (in dashed text symbols), and some to C declarations (in regular text symbols). If a symbol contains both, it will be split in two. For example:

```
INHERITS SuperClass;
DCL i INTEGER;
```

will be translated to:

```
INHERITS SuperClass;
```

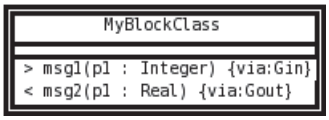
```
int i;
```

- An additional partition containing an additional text symbol may appear in process and procedure diagrams. This symbol is used to manage SDL nested scopes, i.e. the ability for procedures to access their parent process's variables. For more details, see "Nested scopes management" on page 122.

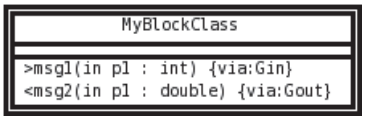
7.3.2 UML class diagrams

The only conversion involved in UML class diagrams is the conversion of the message parameter types to their C equivalent.

For example, if the UML class diagram in the SDL project contains:



it will be converted in the SDL-RT project to:



7.3.3 Other diagrams

The following diagram types are copied to the SDL-RT project without any conversion:

- MSC and HMSC diagrams,
- UML use case and deployment diagrams.

8 - SDL and SDL-RT code generation

8.1 - Basic principles

In a generated executable, SDL-RT or SDL process instances must execute in parallel. To handle this, two solutions are available:

- The generated code can rely on a RTOS to actually execute the instances in parallel using tasks or threads;
- The generated code can use a scheduler to handle the parallelism by executing instances transition by transition, based on the messages they send to each other.

RTDS offers both ways of generating code, and even ways to mix the two. For example, the generated code can use tasks or threads for each block, and schedule the process instances within the blocks. In addition, it is possible to generate code in C or C++, each language offering different possibilities.

Here is a summary of the different code generation types that are available in RTDS:

- **C code generation with a RTOS**: This is the basic code generation feature in RTDS. It requires an RTOS for which the code will be generated. Each process instance will be mapped to one RTOS task. This type of code generation is available for SDL-RT and SDL, and is described in paragraph “C code generation with a RTOS” on page 132.
- **C++ code generation with or without a RTOS**: This is the most advanced code generation feature. It allows to set up precisely how process instances will be scheduled and arranged into tasks, based on the system architecture. This type of code generation is available for SDL-RT and SDL, and is described in paragraph “C++ code generation with or without a RTOS” on page 183.
- **C code generation with the built-in scheduler**: This type of code generation allows only a full system scheduling. In this specific case, it can be used instead of the more advanced C++ code generation if C++ cannot be used for any reason. It is described in paragraph “C code generation with RTDS C scheduler” on page 192.
- **C code generation with an external scheduler**: This type of code generation is based on the previous one, but does not include the built-in C scheduler. It can be used if the generated code must be integrated in an external scheduler with code coming from other tools. It also has some limitations, since RTDS cannot know how the external scheduler is working. This type of code generation is described in paragraph “C code generation with external C scheduler (SDL only)” on page 194.

8.2 - C code generation with a RTOS

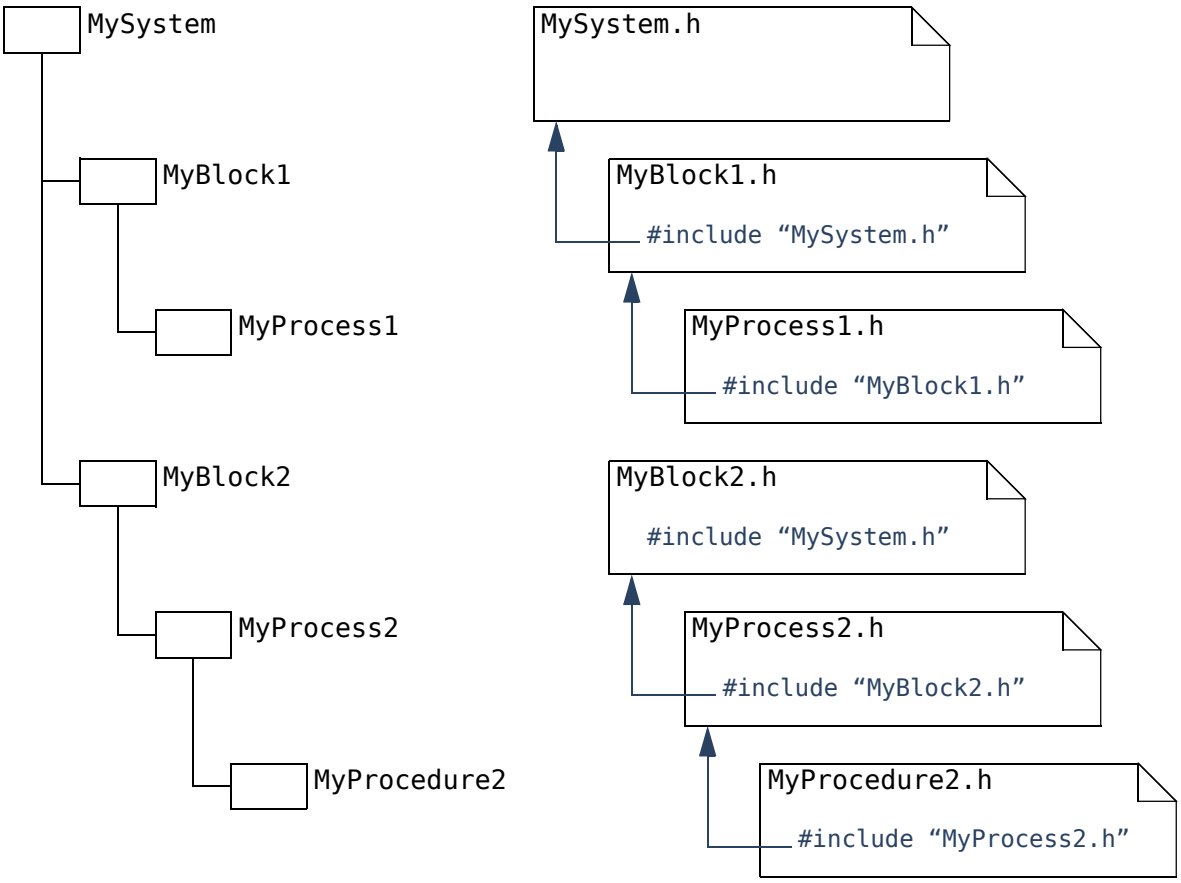
8.2.1 Principles

As described above, this code generation requires a RTOS or OS to work. It maps each process instance to exactly one task or thread and relies on the RTOS to perform the

scheduling. The basic services such as instance creation, message sending, semaphore handling, and so on..., use the RTOS or OS API. So the generated code is interfaced with the underlying RTOS using a RTOS *integration*, as described below.

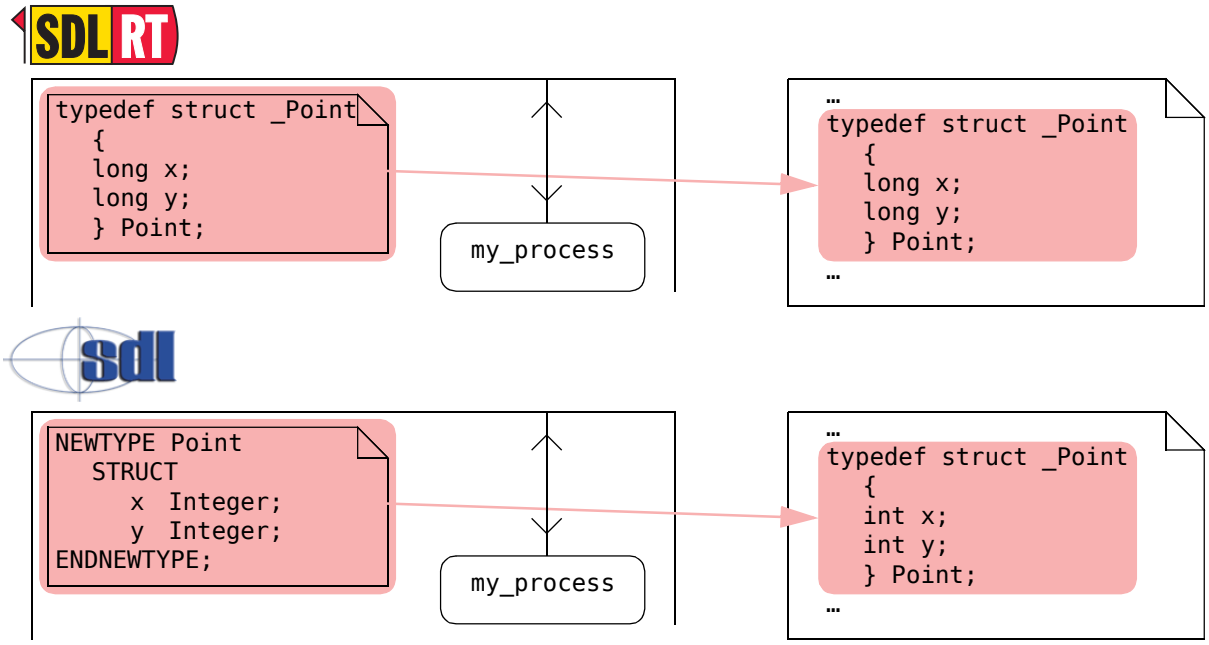
Code generation is run on a diagram in a project with a set of generation options. For each diagram is generated a C header file, and a C source file for behavioral elements (processes and procedures).

To ensure the visibility of declarations made in all ancestors of a diagram in any element, each header file always includes the header file generated for its parent element:



The name for the generated C header or source file is always the name of the agent with the proper extension, except when there are several agents with the same name. In this case, the files for the first encountered one are named `<agent name>__1`, then `<agent name>__2` for the second one, and so on...

For all architecture diagrams, all declarations are copied without modification from the diagram to the generated C header file in SDL-RT, or translated according to the rules described in “SDL to C translation rules” on page 110:

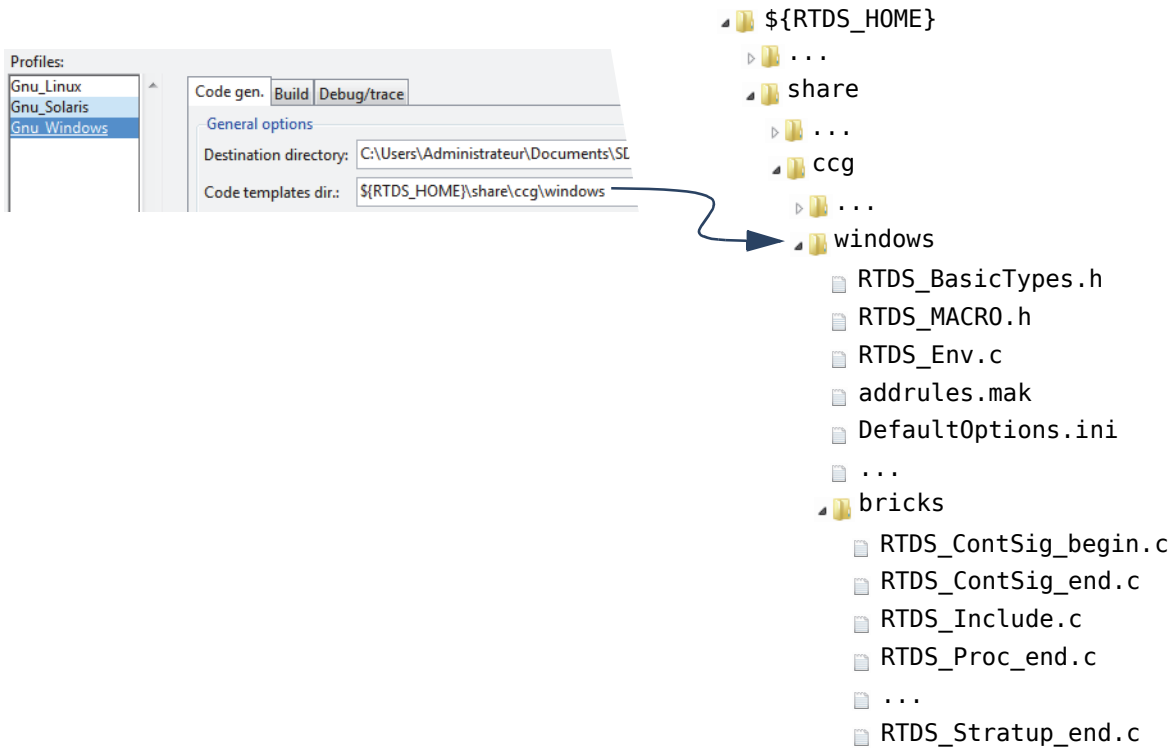


If several declaration symbols are present in the same partition of the diagram:

- In SDL-RT projects, their order in the generated header file is random. However, the order of the partitions is kept: declarations in the first partition will all appear before the declarations in the second, and so on...
- In SDL, the declarations and their dependencies are analysed and they are put in the proper order in the generated file. This means that the sequence of declarations in the diagram is never preserved in the generated file. Please also note that recursive types are not supported and will make the code generation fail.

For behavioral elements, the generated C header file won’t contain the declarations made in declaration symbols, since those can be declarations for variables that must go in the process body. Only the other declarations are inserted in the header file, such as those for procedures or messages declared in the process.

The C source file for a process actually contains very few code generated directly by RTDS. Most of the code is taken from the RTOS integration, which is described in the directory entered in the field “Code templates dir.” in the generation options:

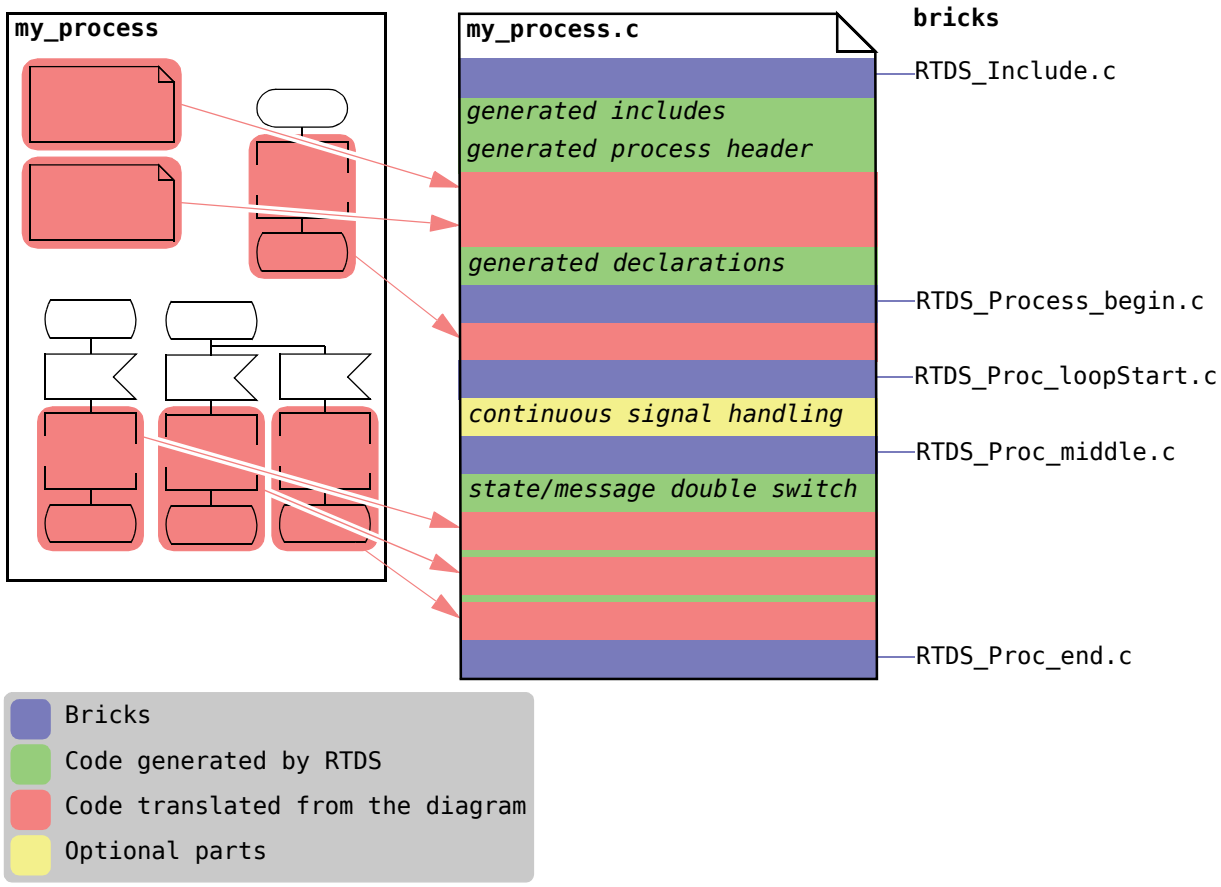


RTOS integrations delivered with RTDS can be found in RTDS’s installation directory, subdirectory `share`, then `ccg`. Each directory having the name of a RTOS is a RTOS integration.

Each integration contains the following:

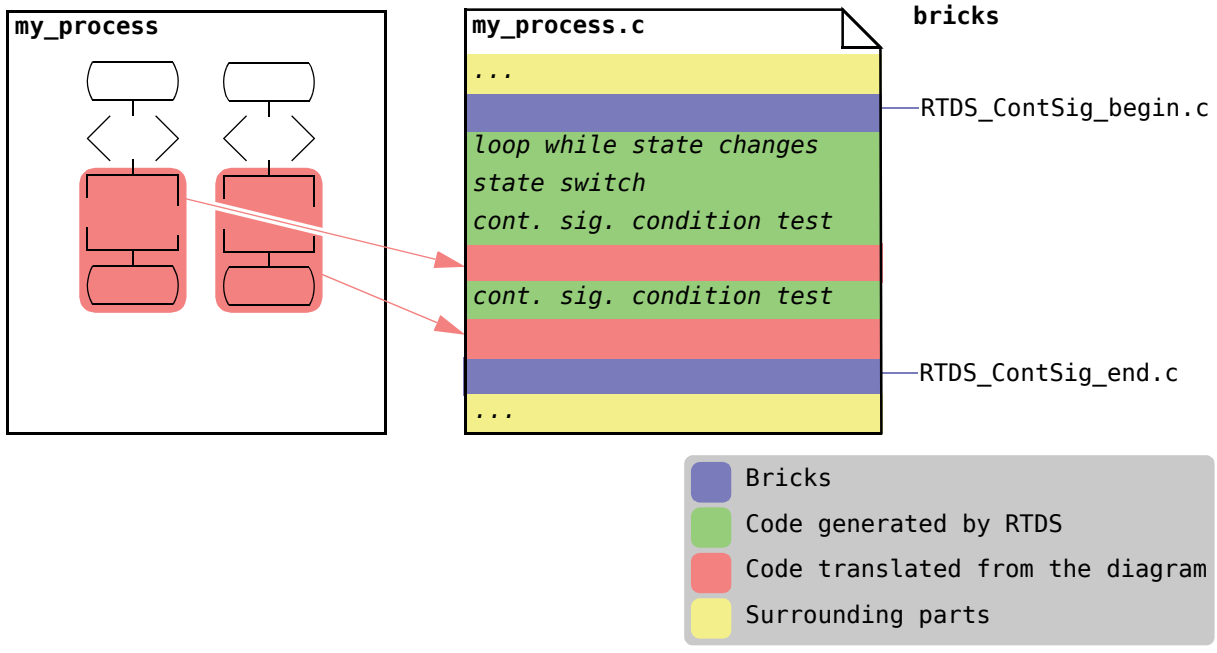
- A set of mandatory files, that will actually be used during the code generation process or within the generated code. These files are described below.
- A set of optional files, that are specific to the integration, and that will be integrated in the built executable via a special file named `addrules.mak`, described below.
- A set of partial C source files called “bricks”, that must be in the subdirectory named `bricks`. These files will be used to generate the code for processes and procedures.

For a process, here is how the corresponding code is generated:

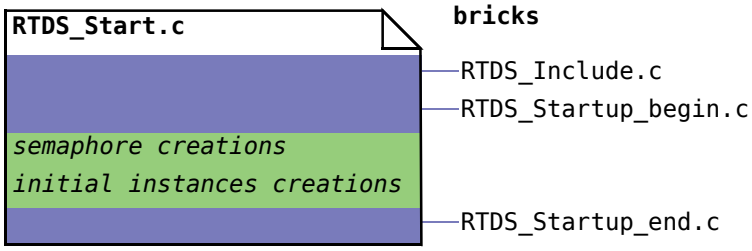


The code for a procedure is generated the same way, except the bricks `RTDS_Procedure_begin.c` and `RTDS_Procedure_loopStart.c` are used instead of `RTDS_Process_begin.c` and `RTDS_Proc_loopStart.c`.

The part for continuous signal handling is optional and only appears if a state in the process or procedure has continuous signals. In this case, this part is generated as follows:

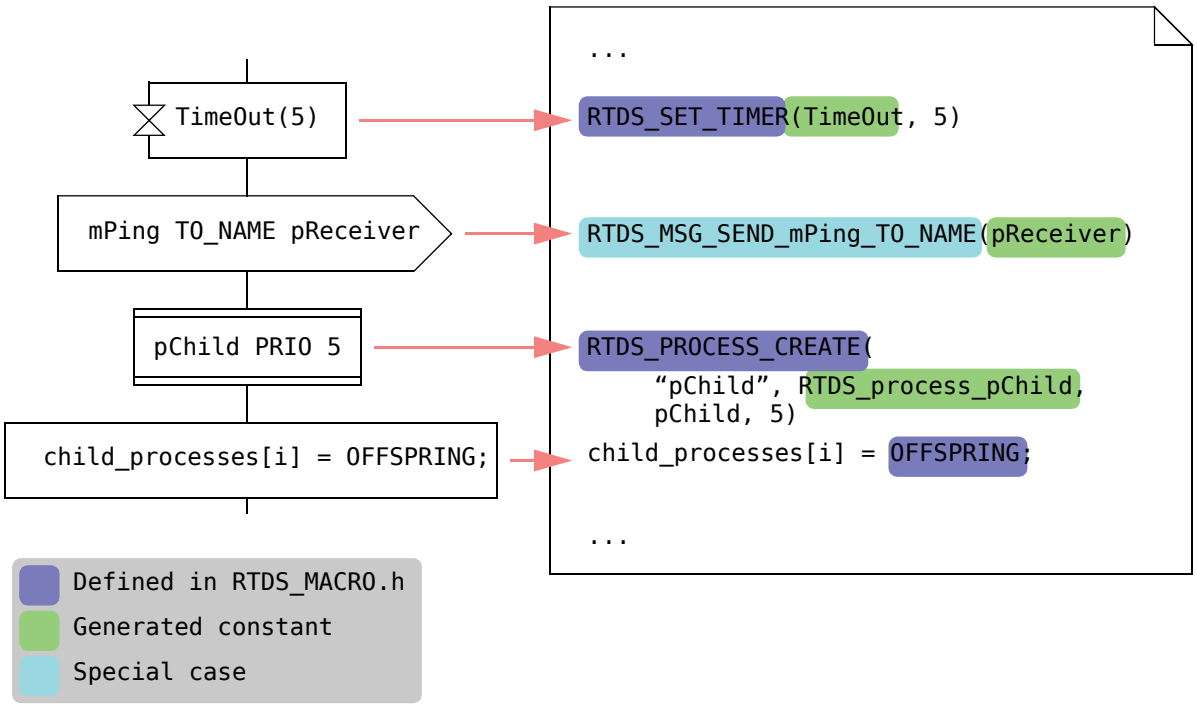


The remaining bricks are used for the generation of the startup task, which is run automatically when the system starts and creates all global data and all initial instances:



Concerning the code in the transitions, all C code is just copied from the symbols to the generated source file, and all SDL code is translated according to the rules described in “SDL to C translation rules” on page 110. For symbols representing a call to an RTOS service, the generated code uses a macro that must be defined in the file `RTDS_MACRO.h` in the RTOS integration. For a given symbol type, the generated macro is always the same, so the code actually generated by RTDS (red and green parts in the diagrams above) is always the same.

Here are some examples of generated code for a few symbol types:



There are a few special cases, especially for message handling where RTDS generates itself some macros to handle message parameters more easily. However, the macro generated by RTDS always ends up calling a macro defined in `RTDS_MACRO.h`. In the example, the generated macro `RTDS_MSG_SEND_mPing_TO_NAME` will only transform its parameters and then call the macro `RTDS_MSG_QUEUE_SEND_TO_NAME`, defined in `RTDS_MACRO.h`.

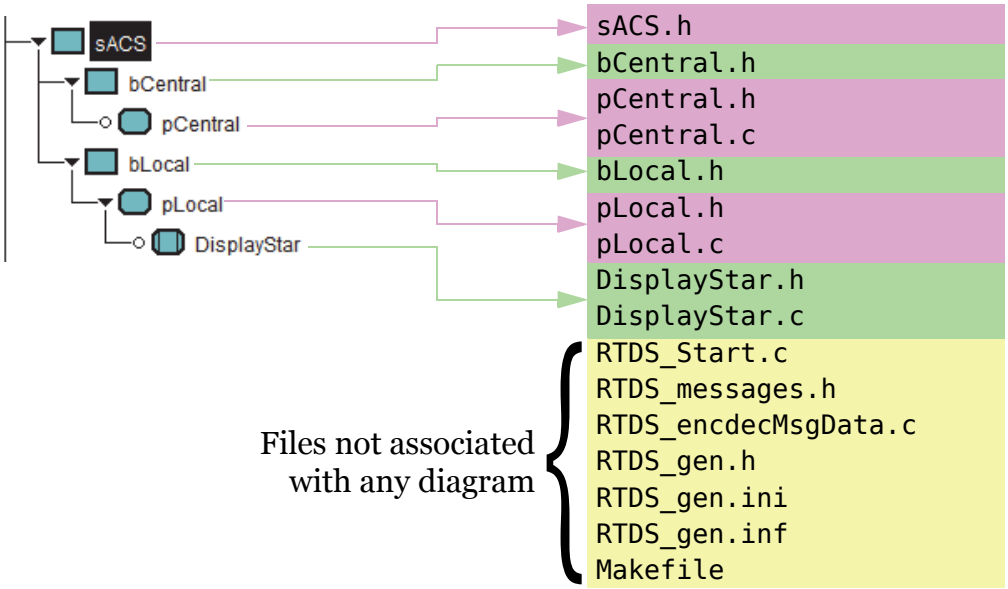
The next paragraphs get into more details concerning the code generation:

- The files actually generated (page 138);
- The structure of a RTOS integration (page 139);
- The various types used in the generated code (page 142);

- The various generated constants in the code (page 147);
- The additional macros generated by RTDS for some specific services such as message handling (page 147);
- The translation performed for all symbols for code generation (page 149);
- The build process, including the generation of the makefile (page 172).

8.2.2 Generated files

The generated files for a typical system are shown on the figure below:



As explained above, a C header file is generated for each diagram, and a C source file is generated for each behavioral diagram. The header files contain the declarations found in the corresponding diagram; they all include the one generated for the parent diagram to ensure the correct scope for declarations.

The other generated files are global to the whole system and are not associated to any diagram. They will always have the same name for all systems and for all RTOS integrations:

- `RTDS_Start.c` contains the startup task for the whole system. This may or may not include a `main` function, depending on the RTOS. The startup task contains all the necessary global initializations for the system, including the creation of semaphores, then creates all initial process instances and runs them.
- `RTDS_messages.h` contains all the generated macros used to handle message inputs and outputs. These macros are described in detail in paragraph “Additional generated types & macros for message handling” on page 147.
- `RTDS_encdecMsgData.c` is generated only if the debugging level in the generation options is not `None`. It contains the utility functions used to encode message parameters to the form used in the RTDS debugger or the MSC tracer (“`|{param1|=…|,param2|=…|,…|}`”) and to decode such a form to the actual message parameters. These functions are built from the analysis of the parameter types for all messages. This file is generated in a second phase during the build, as described in “Pre-build action: Message encoders & decoders generation” on page 180.

- `RTDS_gen.h` contains all generated constants used in the generated code. These include the numerical identifiers for all messages, timers, states, processes, and semaphores in SDL-RT. This file also contains all prototype declarations for the functions implementing the processes. For more details, see paragraph "Generated constants and prototypes (`RTDS_gen.h`)" on page 147.
- `RTDS_gen.ini` contains the same constants as `RTDS_gen.h` in the form of a Windows INI file. The section are `[States]`, `[Processes]`, `[Semaphores]` and `[Messages]`, the option names are the names for the states, processes, semaphores and messages and their value is the numerical value as defined in `RTDS_gen.h`. There are 2 additional sections in the file:
  - `[CodeGen]` contains a single option called `profileName`, containing the name of the code generation profile used for the generation.
  - `[MessageParams]` contains the type for the transport structure for parameters for all messages, the option name being the message name, and its value the name of the structure type.

  This file is used by the RTDS debugger to map numerical identifiers to the corresponding names. It can also be used by external tools.
- `RTDS_gen.inf` contains some detailed information about the running system that can be used by some RTOS integrations, especially to create static structures when the underlying RTOS doesn't allow dynamic creations. It includes:
  - The name and numerical identifier for all declared semaphores;
  - The name, numerical identifier, function name, minimum and maximum number of instances and priority for all processes;
  - The name, numerical identifier and parameter transport structure type for all messages.
- `Makefile` is the makefile for the whole system. It is described in detail in paragraph "Build process" on page 172.

8.2.3 Structure of a RTOS integration

As described in "Principles" on page 132, a RTOS integration is a directory containing 3 sets of files:

- A set of mandatory files, that have to be present in all integrations, since they are explicitely used in the generated code.
- A set of optional files, that will be included in the final build. These files are typically utility functions above the RTOS API.
- A set of files called bricks, used to generate the code for the processes, procedures and the startup task.

These files are described in the following paragraphs, as well as the naming conventions used in all integrations.

8.2.3.1 Naming conventions

All file names in all RTOS integrations are prefixed with "`RTDS_`" to avoid name clashes with generated files.

In the code for the integration as well as in the generated code where it applies, all variables and types used internally are prefixed with "`RTDS_`", and global variables used by RTDS to gather information about the running system are prefixed with "`RTDS_global`".

8.2.3.2 Mandatory files

The files that have to be present in every RTOS integration are the following ones:

- `RTDS_BasicTypes.h` defines the names used by RTDS for the basic RTOS concepts such as message queues, tasks, semaphores, and so on. These types are usually not those used directly in the generated code, as RTDS adds a layer above these to add specific information. All these types are described in detail in paragraph “Types used in the generated code” on page 142.
- `RTDS_MACRO.h` defines all macros used to translate the symbols found in the diagrams to C code, as explained in “Principles” on page 132. The exact translation performed for all symbols and the macros that should be in this file are described in detail in paragraph “C translation for symbols” on page 149.
- `RTDS_Env.c` is not always used, so it is not exactly mandatory. However, it has to be present if the option “Generate environment process” is checked in the “Code gen.” tab of the generation options in the project. It contains the default implementation for the environment process (`RTDS_Env`).
- `addrules.mak` references the optional files that must be included in the final built and gives the rule that has to be included in the generated makefile for them. It can use the macros defined in the generated makefile, described in paragraph “Build process” on page 172.
  For example, the default OSE integration contains 2 optional files, named `RTDS_OS.c` and `RTDS_Trace.c`. To include them in the final build, the file `addrules.mak` must contain the following lines:
  ```
  RTDS_OS.o: $(RTDS_TEMPLATES_DIR)/RTDS_OS.c
  RTDS_Trace.o: $(RTDS_TEMPLATES_DIR)/RTDS_Trace.c
  ```
  The macro `RTDS_TEMPLATES_DIR` is always defined in the makefile and points to the directory for the RTOS integration.
- `DefaultOptions.ini` contains a set of options for the integration and the build process. It is a regular Windows INI file, containing the following sections:
  - `[general]` contains the options for the integration itself. The options it contains are described below.
  - `[common]` contains the options for the build process that should apply to all cases: build only, with MSC Tracer support and for RTDS debugger. The options it can contain are described in paragraph “Build process” on page 172.
  - `[tracer]` contains the additional options that should apply when MSC Tracer support is turned on in the generation options. The possible options are the same as in the `[common]` section.
  - `[debug]` contains the additional options that should apply when RTDS debugger support is turned on in the generation options. The possible options are the same as in the `[common]` section.

  The available options in the `[general]` section are:
  - `rtos`: The name for the RTOS for the integration. This information is used internally by RTDS to access to the debugging information.
  - `socketAvailable`: 1 if a connection by socket to the debugged program is available, 0 if it’s not; Default is 0.
  - `malloc`: Specifies whether dynamic memory allocation is supported by the integration. Possible values are `forbidden` and `allowed`; Default is `allowed`.

- `requiredEnvVariables`: Environment variables that have to be defined to be able to use the integration. The value is a semicolon separated list of strings; Default is empty, meaning no environments variables are required.
- `scheduling`: Specifies whether scheduling is supported by the integration. Possible values are `required`, `supported` and `unsupported`; Default is `unsupported`. The value `required` means that this integration does not support the notion of thread or task and that it has to be used with a scheduler (built-in or external).

NB: In some integrations, there might be some other sections such as `[options]` or `[makefile]` containing various options. These sections are deprecated and should not be used anymore.

8.2.3.3 Optional files

Any integration can contain any number of additional source files implementing functions that are used directly or indirectly by the macros in `RTDS_MACRO.h`. To include them in the final build, they have to be referenced in the `addrules.mak` file described above. If an additional header file is required, it does not have to be declared anywhere, since the integration directory is always inserted in the include path, so a simple `#include` directive will find it.

Integrations provided with RTDS usually contain a file named `RTDS_OS.c`, containing various functions calling the RTOS API after adapting the data found in the common types used in all integrations (see “Types used in the generated code” on page 142). They also usually contain a file named `RTDS_Error.h`, gathering all the error codes that might be returned by all operations.

8.2.3.4 Bricks

Bricks are used to generate the code for processes, procedures and the startup task in `RTDS_Start.c`, as explained in “Principles” on page 132:

- `RTDS_Include.c` contains the `#include` directives that should be inserted at the beginning of all source files: processes, procedures and startup task.
- `RTDS_Process_begin.c` & `RTDS_Procedure_begin.c` contain the definitions of the local variables for each process or procedure (resp.). This brick is inserted just after the declaration of the function implementing the process or procedure.
- `RTDS_Proc_loopStart.c` & `RTDS_Procedure_loopStart.c` contain the statement opening the infinite loop for the process or procedure (resp.). Today, for all integrations delivered with RTDS, this brick contains “`for ( ; ; ) {`”, but it can be customized in a user-defined integration.
- `RTDS_ConSig_begin.c` & `RTDS_ConSig_end.c` are only used if the process or procedure contains states handling continuous signals. These bricks are inserted before and after the code block handling these continuous signals. See paragraph “Continuous signal” on page 158.
- `RTDS_Proc_middle.c` is inserted just after the declarations in the process or procedure, or after the block handling continuous signals if any, and before the generated double-switch/case for states and messages. It contains the acquision of the next message to consider, either in the save queue for the process (see “Message save” on page 160) or in the process’s own message queue (see “Message input” on page 156). It is common to processes and procedures.

- `RTDS_Proc_end.c` is inserted after the generated double-switch/case for state and message. It usually contains clean-up code that should be executed before checking continuous signals after a state change, or before acquiring the next message to consider. It is common to processes and procedures.
- `RTDS_Startup_begin.c` contains the beginning of the file `RTDS_Start.c` containing the startup task. It is inserted after the code from `RTDS_Include.c` and the inclusion of the necessary generated header files. It is followed by the initial creations for the generated system: semaphores, then initial process instances.
- `RTDS_Startup_end.c` is the end of the startup task. It is inserted in `RTDS_Start.c` after the initial creations for the generated system.

8.2.4 Types used in the generated code

RTDS defines its own types for basic SDL-RT or SDL concepts such as process instances or messages. These types are the same ones for all integrations, as RTDS needs to find the same information in the same structures, typically for the RTDS debugger. For this reason, these types are not defined in a header file in the integration itself, but in a file named `RTDS_Common.h`, located in `$RTDS_HOME/share/ccg/common`.

However, some information in these types are actually dependent on the RTOS. For example, the message queue for a process instance will be a message queue created by the RTOS. So its type is RTOS dependent. The mechanism chosen to handle this problem is the following:

- The generated files and the files in the RTOS integration always include the shared file `RTDS_Common.h`.
- `RTDS_Common.h` itself includes another file, called `RTDS_BasicTypes.h`, which is not common, but defined in the RTOS integration. This file contains the type definitions for all basic concepts that have a RTOS dependent type.

This allows to have common top-level types that can have RTOS-specific parts, that the RTOS integration will be able to use, but that RTDS itself doesn't need to know.

The following paragraphs describe the common and RTOS-specific types.

8.2.4.1 Common types - `RTDS_Common.h`

The common types are described in the following table:

Table 9: Common types

`RTDS_SdlInstanceId`		Identifier for a process instance in a running system. Each instance will have its own unique one.
`queueId`	*`RTDS_RtosQueueId`*	Identifier for the message queue for the instance. If the instance is a task, it has its own message queue; If it is scheduled, this will be the message queue for its scheduler. The type `RTDS_RtosQueueId` must be defined in `RTDS_BasicTypes.h`.

Table 9: Common types

instanceNumber	*int*	Order number for instance if it is run within a scheduler. If the instance runs in its own task, the value of this field is not significant.
RTDS_MessageHeader		Descriptor for a message sent or received by a process instance.
RTDS_MESSAGE_HEADER_ADDITIONNAL_FIELDS		Macro that has to be defined in RTDS_BasicTypes.h, allowing to add fields at the beginning of the message descriptors. Some RTOSes require having some specific information at the beginning of the descriptor.
messageNumber	*long*	Numerical identifier for the message type. This is the message identifier generated in RTDS_gen.h (see “Generated files” on page 138).
timerUniqueId	*long*	Unique numerical identifier for a timer “instance”. Each running timer will have its own, even if it has the same timer name as another one. Used to identify a timer when it times out or when it has to be cancelled. If this field is 0, the message is not for a timer.
messageUniqueId	*unsigned long*	Unique identifier for the message instance. Each sent message will have its own, even if it has the same message name as another one. Used to match the message reception with its sending, especially in the trace. Only present if the code is generated for the RTDS debugger.
sender	*RTDS_SdlInstanceId**	Points to the descriptor of the message’s sender instance. It is typically used to implement the SENDER variable in the processes.

Table 9: Common types

receiver	*RTDS_SdlInstanceId**	Points to the descriptor for the message's receiver instance. This field is set when the message is sent. It is necessary in the context of scheduled instances, to figure out if the message should be put in another message queue, or if it should be handled internally by the scheduler when both the sender and receiver instances are run within the same scheduler.
dataLength	*long*	Length for the data associated with the message, i.e its parameters.
pData	*unsigned char**	Pointer on the data associated with the message, i.e its parameters. The contents of this field varies depending on the message. See "Additional generated types & macros for message handling" on page 147.
next	*RTDS_MessageHeader**	Pointer on the next message if this one is in a list or queue.
RTDS_GlobalProcessInfo		Descriptor for a running process instance.
myRtosTaskId	*RTDS_RtosTaskId*	Identifier for the task for the instance. This is the identifier at the RTOS level. The type RTDS_RtosTaskId has to be defined in RTDS_BasicTypes.h.
sdlProcessNumber	*int*	Numerical identifier for the process. This is the process identifier as generated in RTDS_gen.h (see "Generated files" on page 138).
mySdlInstanceId	*RTDS_SdlInstanceId**	Identifier for this instance.
parentSdlInstanceId	*RTDS_SdlInstanceId**	Identifier for this instance's parent if any.
offspringSdlInstanceId	*RTDS_SdlInstanceId**	Identifier for the last process instance created by the current one if any.

Table 9: Common types

`sdlState`	*`int`*	Current state for this instance. The value is a state identifier generated in `RTDS_gen.h` (see “Generated files” on page 138).
`currentMessage`	*`RTDS_MessageHeader*`*	Last message read for the instance.
`timerList`	*`RTDS_TimerState*`*	List of currently active timers in the instance. The type `RTDS_TimerState` has to be defined in `RTDS_BasicTypes.h`.
`readSaveQueue`	*`RTDS_MessageHeader*`*	Save queue used when reading saved messages. See how saved messages are handled in paragraph “Message save” on page 160.
`writeSaveQueue`	*`RTDS_MessageHeader*`*	Save queue used when actually saving messages. See how saved messages are handled in paragraph “Message save” on page 160.
`next`	*`RTDS_GlobalProcessInfo*`*	Pointer to the next descriptor when it is in a list.
`RTDS_GLOBAL_PROCESS_INFO_ADDITIONNAL_FIELDS`		Macro defined in `RTDS_BasicTypes.h` to add RTOS-specific fields to the structure when needed.
`RTDS_EventType`		Enumerated type for types of events that can be reported to the RTDS debugger.
`RTDS_GlobalTraceInfo`		Descriptor for an event that has to be traced by the debugger. Defined only when the debugger or tracer support is on.
`event`	*`RTDS_EventType`*	Type for the event.

Table 9: Common types

`eventParameter1`	*`void*`*	First parameter for the event. The actual type for this parameter depends on the event type. For example, for a message send, this parameter will be a pointer on the corresponding `RTDS_MessageHeader`, or for an instance creation, it will be the `RTDS_GlobalProcessInfo` for the created instance.
`eventParameter2`	*`long`*	Second parameter for the event. For example, it will be the timer delay for a timer start, or the state identifier for a state change.
`currentContext`	*`RTDS_GlobalProcessInfo*`*	Descriptor for the instance within which the event happened.

NB: In addition to these types, `RTDS_Common.h` also defines type named `RTDS_QueueId`, which is an alias for `RTDS_SdlInstanceId*`. This is for backwards compatibility with previous versions of RTDS that identified a process instance with its message queue. This type should no more be used.

8.2.4.2 RTOS-specific types - `RTDS_BasicTypes.h`

The types that must be defined in `RTDS_BasicTypes.h` are the following ones:

- `RTDS_RtosTaskId` is the identifier for a task in the underlying RTOS. For example, it is an integer in the VxWorks integration, a `PROCESS` in the OSE 5.2 integration, and a `pthread_t` in the POSIX integration.
- `RTDS_RtosQueueId` is the identifier for a message queue in the underlying RTOS. For example, it is a `MSG_Q_ID` in the VxWorks integration, and a pointer on a custom type named `RTDS_QCB` in the POSIX integration.
- `RTDS_TimerState` is a descriptor for a running timer. It is usually a custom `struct` type, including RTOS-specific fields such as the `WDOG_ID` in the VxWorks integration.

In addition to these types, fields can be added to the types defined in RTDS_Common.h by defining the following macros:

- `RTDS_GLOBAL_PROCESS_INFO_ADDITIONNAL_FIELDS` can be used to add fields at the end of the `RTDS_GlobalProcessInfo` type. This is typically used to add RTOS-specific information such as the instance's priority, or its stack, or its message queue.
- `RTDS_MESSAGE_HEADER_ADDITIONNAL_FIELDS` can be used to add fields at the beginning of the `RTDS_MessageHeader` type. It is very often unused, except when the RTOS requires a specific field at the beginning of the message descriptor, e.g OSE which requires its own signal identifier to be there (type `SIGSELECT`).

The file RTDS_BasicTypes.h can of course also include other type or macro definitions if they have to be known in the whole generated code and/or integration files.

8.2.5 Generated constants and prototypes (RTDS_gen.h)

This paragraph describes in detail the contents of the generated file RTDS_gen.h. As said above, this file defines constants for all processes, messages, states and semaphores in the SDL-RT or SDL system. It also defines the prototypes for the functions implementing the processes.

More precisely:

- A #define'd constant is generated for each process in the system. The constant has the name of the process prefixed with RTDS_process_. If there are several processes with the same name in the system, the first encountered one will have a suffix __1, the second one a suffix __2, and so on...
- A #define'd constant is generated for each state name in all processes. The constant has the name of the state. If a state with the same name is used in two or more processes, a single constant is generated, so the numerical value for a given state name is always the same, whichever process it appears in.
- A #define'd constant is generated for each message. The constant has the name of the message. All messages are considered global: If a message is declared in a block, its numerical identifier will be declared globally in RTDS_gen.h anyway. This means that hiding a message declared at system level with one declared in a block cannot work.
- A prototype for each function implementing a process. The prototype is not written directly, but via a macro defined in RTDS_MACRO.h. The declaration for a process foo will be:
  RTDS_TASK_ENTRY_POINT_PROTO(foo);
  This is necessary since some RTOSes use themselves macros to declare the function that can be used as tasks.

8.2.6 Additional generated types & macros for message handling

As described in paragraph "Common types - RTDS_Common.h" on page 142, the descriptor for message includes the message parameters as a simple couple length + pointer (field dataLength and pData), the pointer having the generic type unsighed char*, allowing to point to any type of memory buffer.

In the SDL-RT or SDL system, the message is however described in a structured way: For any message can be given one or several parameter types. When sending the message, the vaules for each individual parameter is specified, and when receiving it, each parameter is assigned to a given variable.

To be able to pass all parameters in a single buffer, RTDS generates a *transport structure* for the message parameters, with a field for each one with the type specified in the message declaration. For example, for a message declared as:

```
MESSAGE CTIdentify_req(t_callIdentity, t_number, t_vendor_extension);
```

will generate the transport structure:

```
typedef struct RTDS_CTIdentify_req_data
  {
  t_callIdentity       param1;
  t_number             param2;
  t_vendor_extension   param3;
  } RTDS_CTIdentify_req_data;
```

This way, if a process sends a message of type `CTIdentify_req`, it just has to allocate a variable data with the type `RTDS_CTIdentify_req_data`, fill its fields, then specify for the fields `dataLength` and `pData` in the `RTDS_MessageHeader` for the sent message the values `sizeof(RTDS_CTIdentify_req_data)` and `(unsigned char*)(&data)` respectively. Similarly, if a process receives a message of type `CTIdentify_req`, specifying the variables `ci`, `n` and `ve` for its parameters, it just has to get the message parameters in the field `pData` in the received `RTDS_MessageHeader`, cast it to the type `RTDS_CTIdentify_req_data*` and extract the fields `param1`, `param2` and `param3` to assign them to the variables `ci`, `n` and `ve` respectively.

Things are actually a bit more complicated than this, since there are two cases for each parameter that forces to do things differently:

- If a parameter has a base type, assigning to it, or assigning its value to a variable can be done with a simple assignment. This applies for parameters declared directly with a base type in the message declaration, but also to synonyms for these ones. If the type `t_number` in the example above was declared as:
  `typedef long t_number;`
  it should be considered as a base type.
- If a parameter has a complex type, such as a struct, union or array, assigning to it, or assigning its value to a variable *cannot* be done with a simple assignment. Doing so requires to use the standard function `memcpy` to copy the whole variable.

Note also that using `memcpy` sometimes doesn't work in the first case: For example, if a parameter is an `int` and the message sending directly specifies the value `0` for the parameter, one of the parameters to `memcpy` would be `&(0)`, which isn't valid C.

To avoid having to handle all these specific cases each time a message is sent or received, RTDS actually generates a set of message-specific macros allowing to directly send them, or decode a received `RTDS_MessageHeader` without having to write everything each time. These macros are:

- `RTDS_MSG_RECEIVE_<message>(RTDS_PARAM1, RTDS_PARAM2, …)`
  This macro decodes the current message, which must be a message with the name `<message>`, and assigns all its parameters to the variable passed in `RTDS_PARAM1`, `RTDS_PARAM2`, and so on...
  In the example:
  `RTDS_MSG_RECEIVE_CTIdentify_req(ci, n, ve)`
  will assign the parameters for the current message, which must be a `CTIdentify_req`, to the variables `ci` (a `t_callIdentity`), `n` (a `t_number`) and `ve` (a `t_vendor_extension`).
- `RTDS_MSG_SEND_<message>_TO_ID(RECEIVER, RTDS_PARAM1, RTDS_PARAM2, …)`
  This macro sends a message with name `<message>` to the process identified by the `RTDS_SdlInstanceId RECEIVER`, with the parameter values specified in `RTDS_PARAM1`, `RTDS_PARAM2`, and so on... The macro allocates itself a variable with the type `RTDS_<message>_data`, fills its fields, then sends the message using the

standard message sending macro RTDS_MSG_QUEUE_SEND_TO_ID, defined in the file RTDS_MACRO.h of the RTOS integration.

- Other variants for sending a message are implemented using the following macros:
  - RTDS_MSG_SEND_mCardAndCode_TO_NAME (RECEIVER, RECEIVER_NUMBER, RTDS_PARAM1, RTDS_PARAM2, …)
    Sends the message to a named process. RECEIVER is the receiver name as a string; RECEIVER_NUMBER is its numerical identifier as found in RTDS_gen.h. Eventually calls the macro RTDS_MSG_QUEUE_SEND_TO_NAME found in RTDS_MACRO.h.
  - RTDS_MSG_SEND_mCardAndCode_TO_ENV(RTDS_PARAM1, RTDS_PARAM2, …)
    Sends the message to the environment. Eventually calls the macro RTDS_MSG_QUEUE_SEND_TO_ENV found in RTDS_MACRO.h.
  - RTDS_MSG_SEND_mCardAndCode_TO_ENV_W_MACRO (MACRO_NAME, RTDS_PARAM1, RTDS_PARAM2, …)
    Sends the message to the environment using a custom macro. No macro in RTDS_MACRO.h is called, but the macro named MACRO_NAME is, passing as parameters the numerical identifier for the sent message as found in RTDS_gen.h, the size of the transport structure and a pointer on it.

All these macros are generated in a file called RTDS_messages.h. All of them are protected against redefinition using #ifndef directives, so it is possible to override RTDS's default message handling with a custom one. The file RTDS_messages.h also defines a macro called RTDS_MSG_DATA_DECL that contains the definition for the pointer variable used to allocate the transport structure for all messages. A call to this macro is inserted in the generated code after all declarations in each process.

8.2.7 C translation for symbols

8.2.7.1 SDL-RT declaration symbol

The SDL-RT declaration symbol is the dashed text symbol that contains every declaration which is not a C declaration. This symbol can only be found in SDL-RT projects. Here are its translations in the different cases:

Table 10: SDL-RT declaration symbol translations

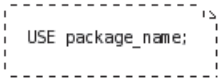
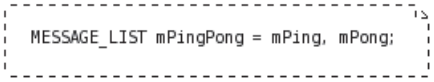
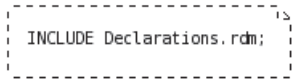
Symbol contents	Translation
	These symbols are analysed before the code generation for their parent diagram starts and the code for the used package will be generated before the one for the diagram. The translation for this symbol will just be a set of #include directives for the header file(s) produced by the package code generation.

Table 10: SDL-RT declaration symbol translations

Symbol contents	Translation
	This kind of declaration only generates the declaration of a constant for `mPing` in `RTDS_gen.h` (see "Generated files" on page 138) and the macros for handling it in `RTDS_messages.h` (see "Additional generated types & macros for message handling" on page 147).
	This kind of declaration is only used during the code generation and doesn't have any translation in the generated code.
	This kind of declaration is only used during the code generation and doesn't have any translation in the generated code.
	This declaration will analyse translate the code found in `Declarations.rdm` and insert its translation at this point in the generated code.

8.2.7.2 Plain declaration symbol

This symbol is the plain text symbol. In SDL-RT, it contains C declarations; In SDL, it contains declarations for signals, types, variables, and so on... Here are its translations in the different cases:

Table 11: Plain declaration symbol translations

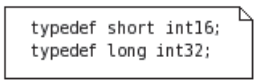
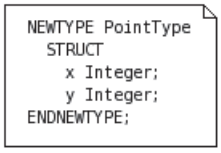
Language	Symbol contents	Translation
SDL RT		The text for the symbol is simply copied in the generated code. For architecture diagrams, it is copied in the diagram's header file; For behavioral diagrams, it is copied at the top of the function, in the part for declarations. No code analysis is performed and if several symbols are present in the diagram, they are inserted in a random order in the generated code.

Table 11: Plain declaration symbol translations

Language	Symbol contents	Translation
		For declarations equivalent to those appearing in a SDL-RT declaration symbol, the translation is the one described in paragraph “SDL-RT declaration symbol” on page 149. For all others, the declaration is translated to C code according to the rules described in “SDL to C translation rules” on page 110 and the result is inserted in the generated code. A dependency analysis is performed before the actual translation and the declarations are put in the order expected in C code.

8.2.7.3 Semaphore declaration symbol

This symbol can only appear in SDL-RT projects. Here are its translations in the different cases, which always appear in `RTDS_Start.c` (see “Generated files” on page 138):

Table 12: Semaphore declaration symbol translations

Symbol contents	Translation

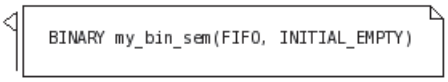
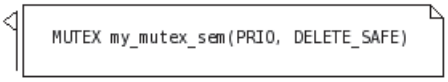
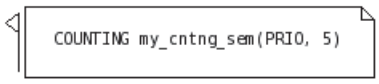
	This generates the macro call for the creation of the binary semaphore:
	`RTDS_BINARY_SEMAPHORE_CREATE(`
	`“my_bin_sem”, my_bin_sem,`
	`RTDS_SEMAPHORE_OPTION_FIFO,`
	`RTDS_BINARY_SEM_INITIAL_EMPTY)`
	Likewise, option `PRIO` would be translated to `RTDS_SEMAPHORE_OPTION_PRIO` and option `INITIAL_FULL` to `RTDS_BINARY_SEM_INITIAL_FULL`.
	The constant `my_bin_sem` is the one generated for the semaphore in `RTDS_gen.h` (see “Generated files” on page 138).

Table 12: Semaphore declaration symbol translations

Symbol contents	Translation

	This generates the macro call for the creation of the mutex semaphore:
	`RTDS_MUTEX_SEMAPHORE_CREATE(`
	`  "my_mutex_sem", my_mutex_sem,`
	`  RTDS_SEMAPHORE_OPTION_PRIO |`
	`  RTDS_MUTEX_SEM_DELETE_SAFE)`
	The options are combined through a binary 'or'; The common options (`PRIO`, `FIFO`) are prefixed with `RTDS_SEMAPHORE_OPTION_` and the mutex-specific options (`DELETE_SAFE`, `INVERSION_SAFE`) with `RTDS_MUTEX_SEM_`.
	The constant `my_mutex_sem` is the one generated for the semaphore in `RTDS_gen.h` (see "Generated files" on page 138).

	This generates the macro call for the creation of the counting semaphore:
	`RTDS_COUNTING_SEMAPHORE_CREATE(`
	`  "my_cntng_sem", my_cntng_sem,`
	`  RTDS_SEMAPHORE_OPTION_PRIO, 5)`
	The `FIFO` option would be similarly translated to `RTDS_SEMAPHORE_OPTION_FIFO`.
	The constant `my_cntng_sem` is the one generated for the semaphore in `RTDS_gen.h` (see "Generated files" on page 138).

8.2.7.4 Block / block class instance declaration symbol

There is no direct generated code for a block or block class instance symbol. It only triggers the code generation for the corresponding block or block class diagram.

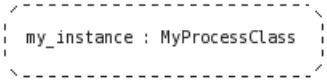
Please note that for a block class instance symbol, the C header file for the corresponding block class will not include the header file generated for the class instance symbol: If a system includes an instance of a block class, the block class has no visibility on the declarations made in the system.

8.2.7.5 Process / process class instance declaration symbol

The only generates code for a process or process class instance symbol is:

- The constant for the process or process class generated in `RTDS_gen.h`;
- The prototype for the function implementing the process or process class, also in `RTDS_gen.h`.
- The creation of the initial instance(s) if any in `RTDS_Start.c`.

Please refer to paragraph “Generated files” on page 138 for more details. Note also that for a process class instance, the numerical identifier is associated to the process class, and not the instance. For example, for a such a symbol:



the generated constant will be called `RTDS_process_MyProcessClass`; There will be no constant named `RTDS_process_my_instance`.

All initial instances are created using a macro defined in `RTDS_MACRO.h`:

```
RTDS_STARTUP_PROCESS_CREATE(
  “my_process”, RTDS_process_my_process, my_process, <priority>);
```

The name `my_process` refers to the function implementing the process. The priority will be the priority specified in the diagram if any (SDL-RT only). If no priority is specified, the priority passed to the macro is `RTDS_DEFAULT_PROCESS_PRIORITY`.

The same call will be made as many times as there are initial instances for the process.

NB: The macro is different from the one used for dynamic process creation (see “Dynamic process instance creation” on page 163). This is due to the fact that the actual start for initial instances usually must be synchronized to avoid an instance to send a message to a not-yet created instance, for example.

8.2.7.6 Procedure declaration symbol

Here are the translations for a procedure symbol in all cases:

Table 13: Procedure declaration symbol translations

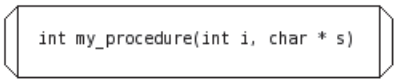
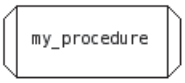
Language	Symbol contents	Translation
SDL RT		The text for the procedure declaration is simply copied to the C header file generated for the parent diagram. An `extern` specifier is added in front of it and an additional parameter with type `RTDS_GlobalProcessInfo` is added.

Table 13: Procedure declaration symbol translations

Language	Symbol contents	Translation
		The full prototype for the procedure is inserted in the C header file generated for the parent diagram. Since the parameters and return type are not present in the symbol, but required in the prototype, the declaration symbols in the diagram for the procedure are parsed to find them in the `FPAR` and `RETURNS` declarations. As in SDL-RT, the prototype is prefixed with `extern` and an additional parameter with type `RTDS_GlobalProcessInfo` is added.

In both cases, the additional parameter with the type `RTDS_GlobalProcessInfo` is added to allow to pass to the procedure the context for the caller process. This allows for example to use the predefined variable `SELF` in procedures, or to identify the sender of a message sent in the procedure as the caller process.

8.2.7.7 Macro definition symbol

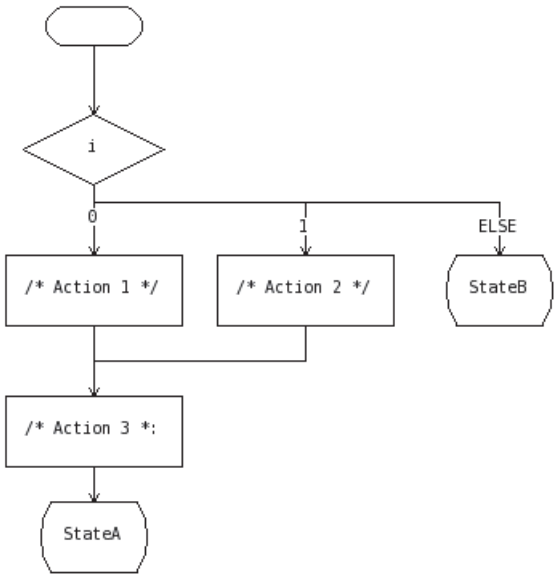
Macros are not supported in code generation. If a macro definition symbol is used in a diagram, code generation will fail.

8.2.7.8 Start symbol

In all cases, no code is generated for a start symbol. It is only used to figure out the beginning of the start transition.

There is however a specific piece of code generated in SDL projects for processes with parameters: These parameters are actually passed to the process using a pseudo-message, named `RTDS_<process name>_start_message`, which accepts as parameters the process parameters. When the instance starts, it goes into a specific internal state and waits for this message before actually executing its start transition. However, if such an instance is created at system stratup, this message is not sent to the instance, since no parameters can be specified in this case. A piece of code is then generated figuring out if the instance was created at startup or dynamically created by testing the value of its `PARENT` variable, and if it was created at startup, send the start pseudo-message with default values for its parameters to itself before changing to the state where it waits for this message.

In all cases, a specific construct is also generated around the start transition itself to be able to handle the following case:



In this case, there is no easy way to generate the C code: The common part after the decision ("`Action 3` and next state to `StateA`) has to be put after the `if` testing the variable `i`, but the `ELSE` branch must not execute it. To avoid generating a `goto`, the following code is generated:

```
do/* Dummy do/while(0) to be able to do 'break's */
  {
  if ( i == 0 )
    {
    /* Action 1 */
    }
  else if ( i == 1 )
    {
    /* Action 2 */
    }
  else
    {
    RTDS_SDL_STATE_SET(StateB);
    break;
    }
  /* Action 3 */
  RTDS_SDL_STATE_SET(StateA);
  break;
  } while (0);
```

The `do/while(0)` block allows to insert a `break` statement after all state changes, which will go out of the block without the need for a label and a `goto`. This way, the generated code has exactly the behavior described in the diagram.

8.2.7.9 State symbol

When they are not next-state symbols, state symbols do not directly generate any code. They will be translated as cases in a `switch` statement in the part handling continuous signals, the part handling message input, or both. See the corresponding paragraphs "Message input" on page 156 and "Continuous signal" on page 158.

8.2.7.10 Composite state

Composite states are not supported in code generation. If a composite state declaration or definition symbol is present in a diagram, the code generation will fail.

8.2.7.11 Message input

As explained in paragraph “Bricks” on page 141, the actual acquisition of the next message to consider in a given process or procedure is done in the code of a brick (`RTDS_Proc_middle.c`). So the code generated for a message input does not actually get the message, but only tests its type and gets in parameters into the variables specified in the symbol.

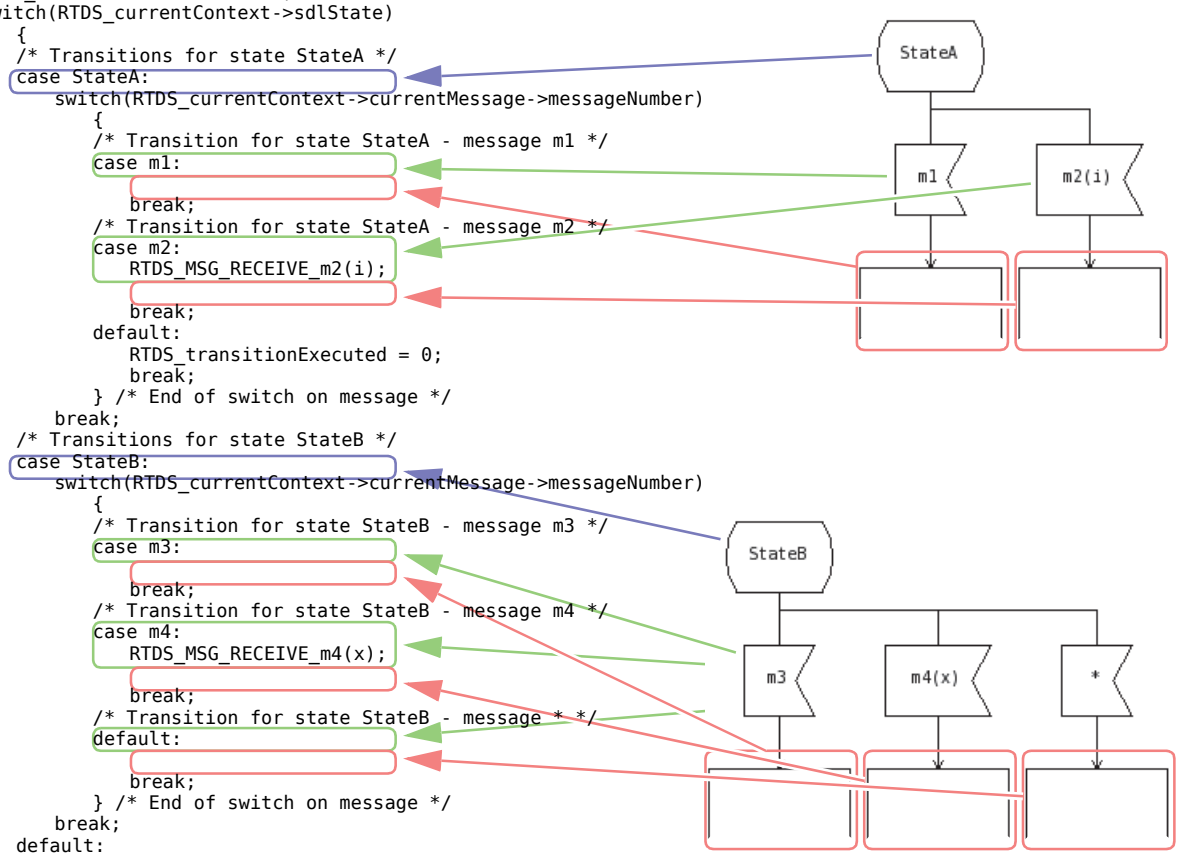
More precisely, the generated code for message inputs includes two parts:

- A `case` entry in the inner level of a double `switch` statement, the outer level being the test of the current state;
- An optional call to the macro decoding the received message parameters in the specified variables. This macro is called `RTDS_MSG_RECEIVE_<message name>` and is described in paragraph “Additional generated types & macros for message handling” on page 147. This call is inserted only if the message has parameters.

There is a special case when the message input symbols contains the text “`*`”: The generated entry in the `switch` is then the `default` case, and parameter decoding is never performed.

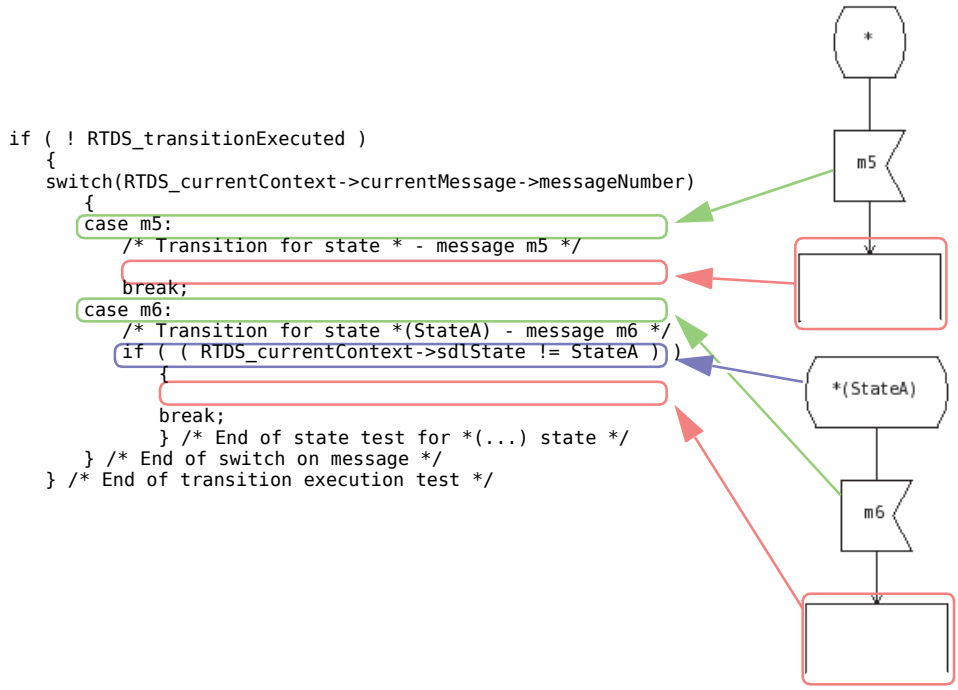
Here is an example of the generated double-switch in a simple case:

```
/* Double switch state / signal */
RTDS_transitionExecuted = 1;
switch(RTDS_currentContext->sdlState)
   {
   /* Transitions for state StateA */
   case StateA:
      switch(RTDS_currentContext->currentMessage->messageNumber)
         {
         /* Transition for state StateA - message m1 */
         case m1:

            break;
         /* Transition for state StateA - message m2 */
         case m2:
            RTDS_MSG_RECEIVE_m2(i);

            break;
         default:
            RTDS_transitionExecuted = 0;
            break;
         } /* End of switch on message */
      break;
   /* Transitions for state StateB */
   case StateB:
      switch(RTDS_currentContext->currentMessage->messageNumber)
         {
         /* Transition for state StateB - message m3 */
         case m3:

            break;
         /* Transition for state StateB - message m4 */
         case m4:
            RTDS_MSG_RECEIVE_m4(x);

            break;
         /* Transition for state StateB - message * */
         default:

            break;
         } /* End of switch on message */
      break;
   default:
      RTDS_transitionExecuted = 0;
      break;
   } /* End of switch(RTDS_currentContext->sdlState) */
```



A special handling has to be done in the case where message inputs are present on a state symbol containing “*” or “*(<state 1>, <state 2>, …)”. For this case, a special indicator named RTDS_transitionExecuted remembers if a “normal” transition has been exe-

cuted during the double-switch. If that isn't the case, the inputs on "*(…)" states are considered via a secondary `switch` on the received message like follows:



8.2.7.12 Message priority input

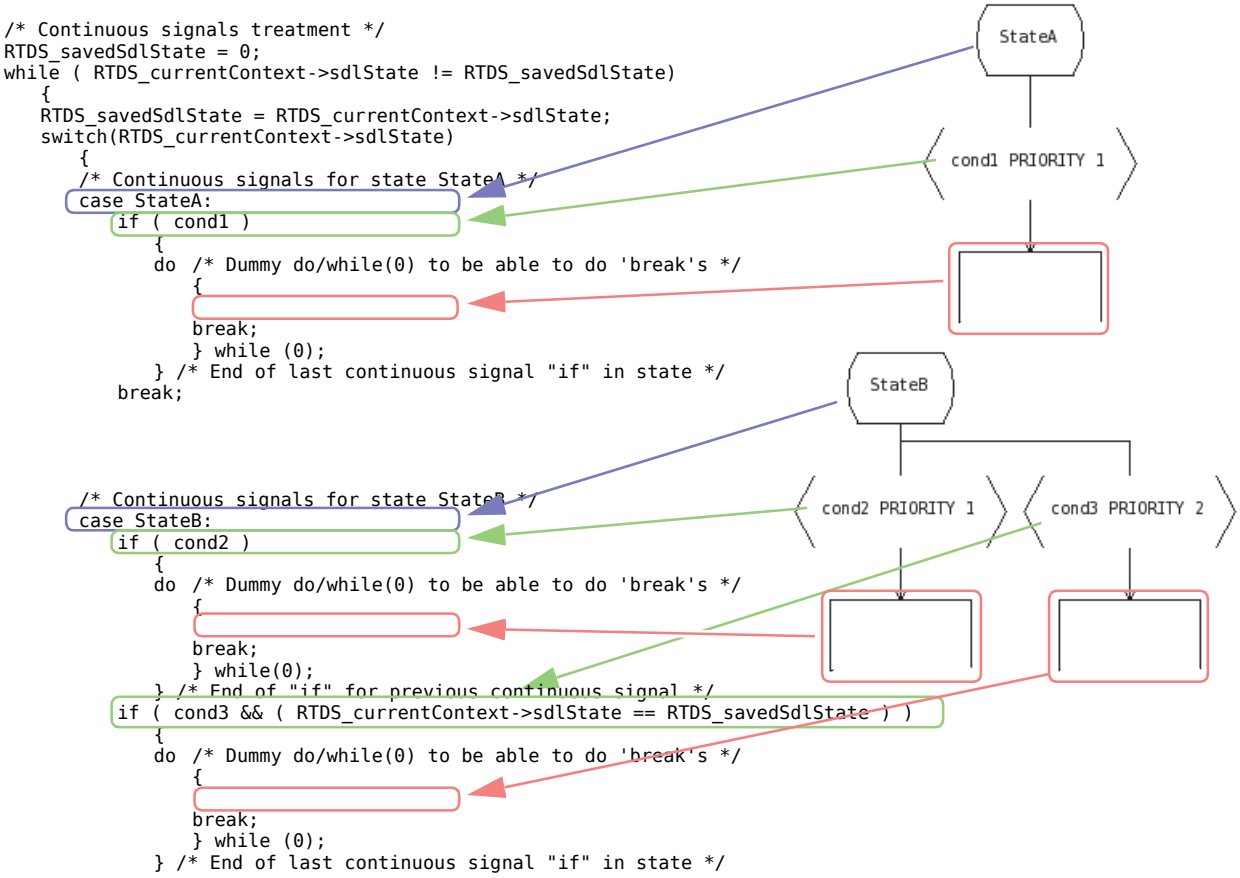
Priority inputs are not supported in code generation. If such a symbol is present in a diagram, the code generation will fail.

8.2.7.13 Continuous signal

The code block handling continuous signals is inserted before getting the next message to handle, as explained in paragraph "Bricks" on page 141. This means that for SDL, the semantics of continuous signals is different from the one described in the Z.100 standard: The standardized behavior is to consider continuous signals only if there are no pending messages for the instance. Since RTOSes usually do not offer a way of checking if there is any pending message in a message queue, the behavior had to be changed to consider continuous signals before any message reception, whether there is a pending message or not. Note this is the standard way to handle continuous signals in SDL-RT.

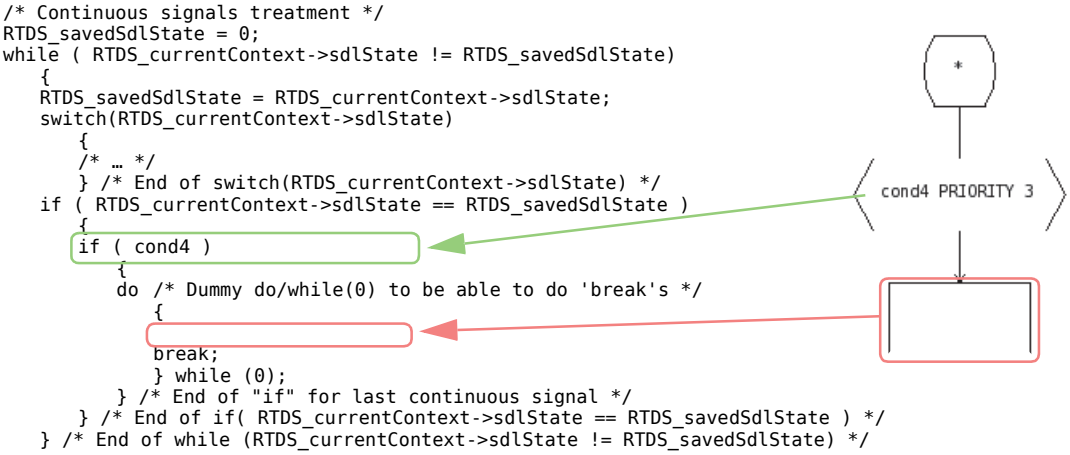
Note however that all continuous signals will be considered at most once for each state: If the condition for a continuous signal is still true after the corresponding transition has been executed, and if the instance stayed in the same state, the condition will *not* be reevaluated again, and the continuous signal transition will not be executed again. This mechanism has been introduced to avoid getting into an infinite loop. Note that this behavior is not the standard one in SDL, where the continuous signal transition would be executed again and again, until a message is received by the instance. Note also that if the instance changes its state in a continuous signal transition, the continuous signals for the new state *will* be taken into account.

The following diagram shows the generated code for a simple example:



The outer `while` loop ensures that if the instance changes its state, the continuous signals for the new state will be considered. In each state `case`, all conditions for all continuous signals are tested successively, in the order of their priority. For the less prioritary continuous signals, a test is also made to make sure the previous transitions didn't change the instance's state.

Continuous signals for a "*" or "*(...)" state are handled simply by testing the condition for the continuous signal after all others have been considered, only if the state hasn't changed in the previously executed transitions:

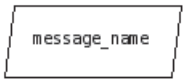


Note also that the transitions for continuous signals use the same C construct as the start transition to be able to get out of the transition with a `break` statement after each state change. See paragraph “Start symbol” on page 154 for more details.

8.2.7.14 Message save

Here is the translation for a save symbol:

Table 14: Save symbol translation

Symbol contents	Translation
	`RTDS_MSG_SAVE(RTDS_currentContext->currentMessage)` This line is inserted in place of the message parameter decoding line using the `RTDS_MSG_RECEIVE_message_name` macro, as no parameter decoding for a message is needed to be able to save it. This line is always the only one in the `case` for the message in the state/message double-switch for the process.

8.2.7.15 Task block

The contents for a task block in a SDL-RT project is simply copied to the generated code. In SDL projects, it is translated according to the rules described in “SDL to C translation rules” on page 110.

8.2.7.16 Message output

Here are the translations for a message output symbol in all cases:

Table 15: Message output symbol translations

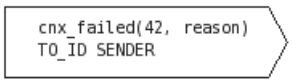
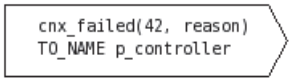
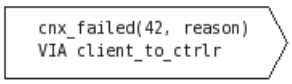
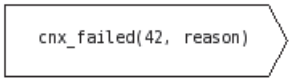
Language	Symbol contents	Translation
SDL RT		`RTDS_MSG_SEND_cnx_failed_TO_ID( SENDER, 42, reason)` This macro is generated in the file `RTDS_messages.h` (see “Additional generated types & macros for message handling” on page 147) and ends up calling the generic macro `RTDS_MSG_QUEUE_SEND_TO_ID` defined in `RTDS_MACRO.h`.

Table 15: Message output symbol translations

Language	Symbol contents	Translation

		`RTDS_MSG_SEND_cnx_failed_TO_NAME(`
		`  “p_controller”,`
		`  RTDS_process_p_controller,`
		`  42, reason)`
		This macro is generated in the file `RTDS_messages.h` (see “Additional generated types & macros for message handling” on page 147) and ends up calling the generic macro `RTDS_MSG_QUEUE_SEND_TO_NAME` defined in `RTDS_MACRO.h`.

		This form of the message output is actually handled within the code generator: The possible receivers for the message are figured out statically, and if it resolves to a unique process name, the translation is the same as for the `TO_NAME` form with the found process name. In all other cases, the code generation will fail. The code generation has this militation since no information about the system architecture is present in the generated code. So the instances have no information about their parent block, or the channel connected to it and their connections. So everything has to be resolved statically.
		Same case as the output `VIA` a channel: All possible receivers are figured out and the output is generated as a `TO_NAME` if possible.

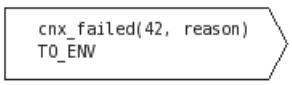
		`RTDS_MSG_SEND_cnx_failed_TO_ENV(`
		`  42, reason)`
		This macro is generated in the file `RTDS_messages.h` (see “Additional generated types & macros for message handling” on page 147) and ends up calling the generic macro `RTDS_MSG_QUEUE_SEND_TO_ENV` defined in `RTDS_MACRO.h`.
		Note the build will fail if no process named `RTDS_Env` is present in the system. This process can be explicitely defined, or generated automatically by checking the “Generate environment process” checkbox in the the “Code gen.” tab of the generation options.

Table 15: Message output symbol translations

Language	Symbol contents	Translation
	cnx_failed(42, reason) TO_ENV SEND_TO_ENV	This form of output is generated differently depending on whether the checkbox “Communicate with environment via macros” is checked or not in the “Code gen.” tab of the generation options. If it isn’t, the code is generated like for a plain TO_ENV with no macro name. If it is, the generated code is: `RTDS_MSG_SEND_cnx_failed_TO_ENV_W_MACRO( SEND_TO_ENV, 42, reason)` which will actually just pack the parameters in a transport structure and call the macro SEND_TO_ENV (see “Additional generated types & macros for message handling” on page 147).
	cnx_failed(42, reason) TO <something>	Generated differently depending of <something>: • If <something> is ENV, the generated code is the same as for a TO_ENV in SDL-RT (see above). • If <something> is a single name which isn’t a variable of type PID in the current context, the generated code is the same as for a TO_NAME in SDL-RT (see above). • In all other cases, the generated code is the same as for a TO_ID in SDL-RT (see above).
	cnx_failed(42, reason) VIA client_to_ctrlr	Generated exactly the same way as in SDL-RT, with the same limitations (see above).
	cnx_failed(42, reason)	Generated exactly the same way as in SDL-RT, with the same limitations (see above).

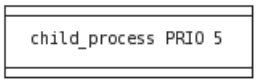
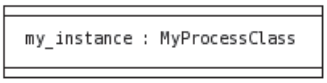
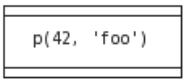
8.2.7.17 Message priority output

Priority outputs are not supported in code generation. If such a symbol is used in a diagram, code generation will fail.

8.2.7.18 Dynamic process instance creation

Here are the translations for a process instance creation symbol in all cases:

Table 16: Process instance creation symbol translations

Language	Symbol contents	Translation
SDL RT		`RTDS_PROCESS_CREATE( “child_process”, RTDS_process_child_process, child_process, 5)` `RTDS_PROCESS_CREATE` is defined in `RTDS_MACRO.h`. `RTDS_process_child_process` is the constant for the process defined in `RTDS_gen.h` and `child_process` is the function implementing it, prototyped in `RTDS_gen.h` (see “Generated constants and prototypes (RTDS_gen.h)” on page 147).
		`RTDS_PROCESS_CREATE( “my_instance”, RTDS_process_MyProcessClass, MyProcessClass, RTDS_DEFAULT_PROCESS_PRIORITY)` For the dynamic creation of a process class instance, the instantiated process is the process class. If no priority is specified, the generated code uses the constant `RTDS_DEFAULT_PROCESS_PRIORITY`, defined in `RTDS_MACRO.h`.
sdl		`RTDS_PROCESS_CREATE( “p”, RTDS_process_p, p, RTDS_DEFAULT_PROCESS_PRIORITY)` then: `RTDS_MSG_SEND_RTDS_p_start_message_TO_ID (OFFSPRING, 42, “foo”)` The pseudo-message `RTDS_p_start_message` is automatically created when seeing the process `p` has parameters. It is used to transfer the parameters to the newly created instance. If an instance of this process is started at system startup, it will automatically send its start message with default parameter values to itself (see “Start symbol” on page 154). Since there is no process priority in SDL, the one passed to the macro `RTDS_PROCESS_CREATE` will always be `RTDS_DEFAULT_PROCESS_PRIORITY`.

8.2.7.19 Procedure call

A procedure call is just transformed into the call to the function implementing the procedure, with an additional parameter which is the context of the caller (pointer on the local variable `RTDS_currentContext`, which is the `RTDS_GlobalProcessInfo` for the current process or procedure; See “Procedure declaration symbol” on page 153).

This translation is also performed for SDL procedure calls in expressions using the `CALL` keyword.

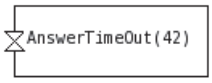
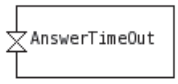
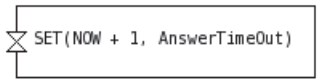
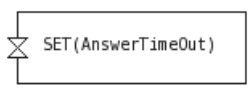
8.2.7.20 Macro call

Macro calls are not supported in code generation. If such a symbol is used in a diagram, code generation will fail.

8.2.7.21 Timer set

Here is the translation for a timer set in all cases:

Table 17: Timer set symbol translations

Language	Symbol contents	Translation
SDL RT		`RTDS_SET_TIMER(AnswerTimeOut, 42)` The macro `RTDS_SET_TIMER` in defined in `RTDS_MACRO.h`. The constant `AnswerTimeOut` is the one defined for the timer in `RTDS_gen.h` (see “Generated files” on page 138).
		Same translation as above, using the delay specified in the timer’s declaration. If no timer declaration exists for `AnswerTimeOut`, the code generation fails.
sdl		`RTDS_SET_TIMER(AnswerTimeOut, 1)` See explanations for SDL-RT equivalent. Please note that the SDL form uses the absolute time for the timer end. The macro `RTDS_SET_TIMER` uses the delay. So the expression for the timer time-out is actually decoded and the “`NOW +`” part removed. This means that an expression that is not an addition between `NOW` and a duration cannot be translated.
		Same translation as above, using the delay specified in the timer’s declaration. If no timer declaration exists for `AnswerTimeOut`, the code generation fails. Note that the declaration actually specifies a delay, so no translation has to be performed in this case.

8.2.7.22 Timer reset

Here is the translation for a timer reset in all cases:

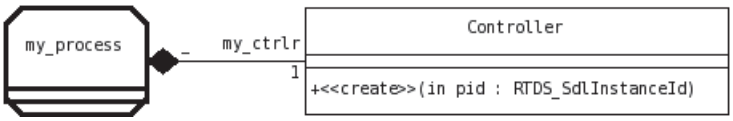
Table 18: Timer reset symbol translations

Language	Symbol contents	Translation
SDL RT	AnswerTimeOut	`RTDS_RESET_TIMER(AnswerTimeOut)` The macro `RTDS_RESET_TIMER` is defined in `RTDS_MACRO.h`. The constant `AnswerTimeOut` is the one defined for the timer in `RTDS_gen.h` (see “Generated files” on page 138).
sdl	RESET(AnswerTimeOut)	`RTDS_RESET_TIMER(<timer name>)` See explanations for SDL-RT equivalent.

8.2.7.23 Object creation

Here is the translation for the SDL-RT specific object creation symbol in all cases:

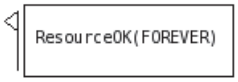
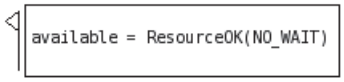
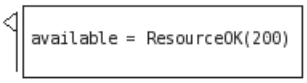
Table 19: Semaphore take symbol translation

Symbol contents	Translation
my_ctrlr : Controller(SELF)	`my_ctrlr = new Controller(SELF);` No macro in `RTDS_MACRO.h` is called. This supposes that the class `Controller` has been linked with the current process in a class diagram, for example like follows:  For more details on passive classes and how the code is generated for them, see paragraph “C++ code generation for passive classes (UML)” on page 182.

8.2.7.24 Semaphore take

Here are the translations for the SDL-RT specific semaphore take symbol in all cases:

Table 20: Semaphore take symbol translation

Symbol contents	Translation
	`RTDS_SEMAPHORE_NAME_TAKE( “ResourceOK”, ResourceOK, RTDS_SEMAPHORE_TIME_OUT_FOREVER)` The macros `RTDS_SEMAPHORE_NAME_TAKE` and `RTDS_SEMAPHORE_TIME_OUT_FOREVER` are defined in `RTDS_MACRO.h`. The constant `ResourceOK` is the one defined for the semaphore in `RTDS_gen.h` (see “Generated files” on page 138).
	`available = RTDS_SEMAPHORE_NAME_TAKE( “ResourceOK”, ResourceOK, RTDS_SEMAPHORE_TIME_OUT_NO_WAIT)` The macro `RTDS_SEMAPHORE_TIME_OUT_NO_WAIT` is defined in `RTDS_MACRO.h`.
	`available = RTDS_SEMAPHORE_NAME_TAKE( “ResourceOK”, ResourceOK, 200)`

8.2.7.25 Semaphore give

Here is the translation of the SDL-RT specific semaphore give symbol in all cases:

Table 21: Semaphore give symbol translation

Symbol contents	Translation
	`RTDS_SEMAPHORE_NAME_GIVE(“ResourceOK”, ResourceOK)` The macro `RTDS_SEMAPHORE_NAME_GIVE` is defined in `RTDS_MACRO.h`. The constant `ResourceOK` is the one defined for the semaphore in `RTDS_gen.h` (see “Generated files” on page 138).

8.2.7.26 Dynamic semaphores

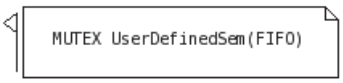
The semaphore handling symbols described in paragraphs “Semaphore declaration symbol” on page 151, “Semaphore take” on page 166 and “Semaphore give” on page 166 only allow static semaphores: A semaphore has to be defined by its name in a semaphore declaration symbol, then this name is used in semaphore take or give symbols. This cannot be used when dynamically defined semaphores have to be used, or to create arrays of semaphores, for example.

RTDS allows such dynamic semaphores by allowing the semaphore definition macros to be called as many times as needed with the same parameters. To do so, it is necessary to use a user-defined semaphore number, making sure that it will not be reused by RTDS

for a static semaphore. Then, to defined for example an array of semaphores, the following code can be written in a task-block:

```
RTDS_SemaphoreId my_semaphores[MAX];
...
for ( i = 0; i < MAX; i++ )
  {
  my_semaphores[i] = RTDS_MUTEX_SEMAPHORE_CREATE(
                        “UserDefinedSem”, 100, RTDS_SEMAPHORE_OPTION_FIFO);
  }
```

This performs the equivalent of the following semaphore declaration symbol:



on all semaphores in the array `my_semaphores`, assigning the semaphore name “`UserDefinedSem`” and the number 100 to all of them.

After that, all semaphores in the array will have the same name and number, so the manipulation macros described above cannot be used, since they only take a semaphore name and number as identifiers. So two additional macros are defined in `RTDS_MACRO.h` to take or give semaphores dynamically defined:

- `RTDS_SEMAPHORE_ID_TAKE(semaphore_id, time_out)`
  This macro takes the semaphore with the identifier `semaphore_id`, which is a `RTDS_SemaphoreId`, with the time-out specified in `time_out`, allowing the same special values as in “Semaphore take” on page 166. In the example, `semaphore_id` would be an element in the array `my_semaphores`.
- `RTDS_SEMAPHORE_ID_GIVE(semaphore_id)`
  This macro gives the semaphore with the identifier `semaphore_id`, which is a `RTDS_SemaphoreId`. In the example, it would be an element in the array `my_semaphores`.

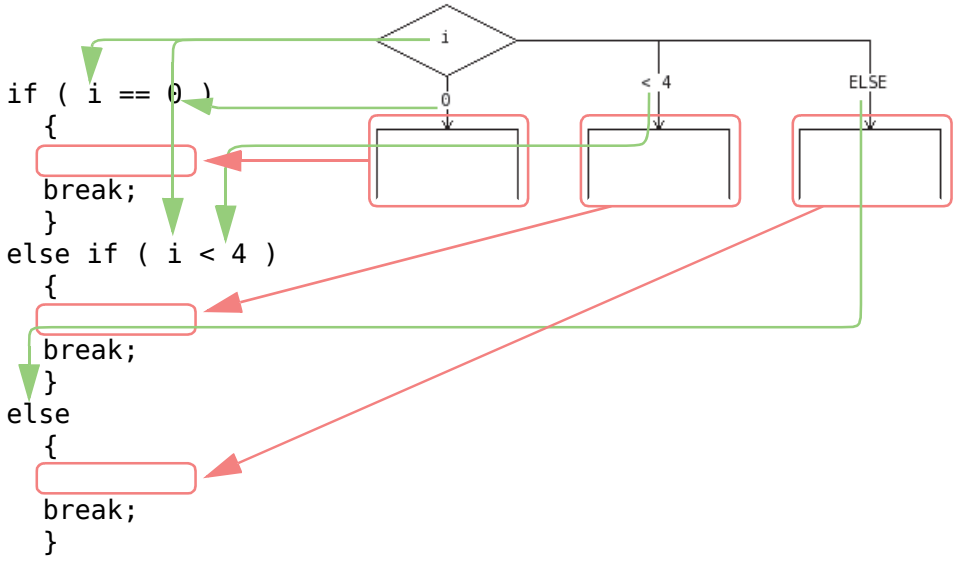
If a dynamically defined semaphore has to be deleted, RTDS also provides a macro to do so, also defined in `RTDS_MACRO.h`:

```
RTDS_SEMAPHORE_DELETE(semaphore_id)
```

deletes the semaphore with the identifier `semaphore_id`, which is a `RTDS_SemaphoreId` (element of the array `my_semaphores` in the example).

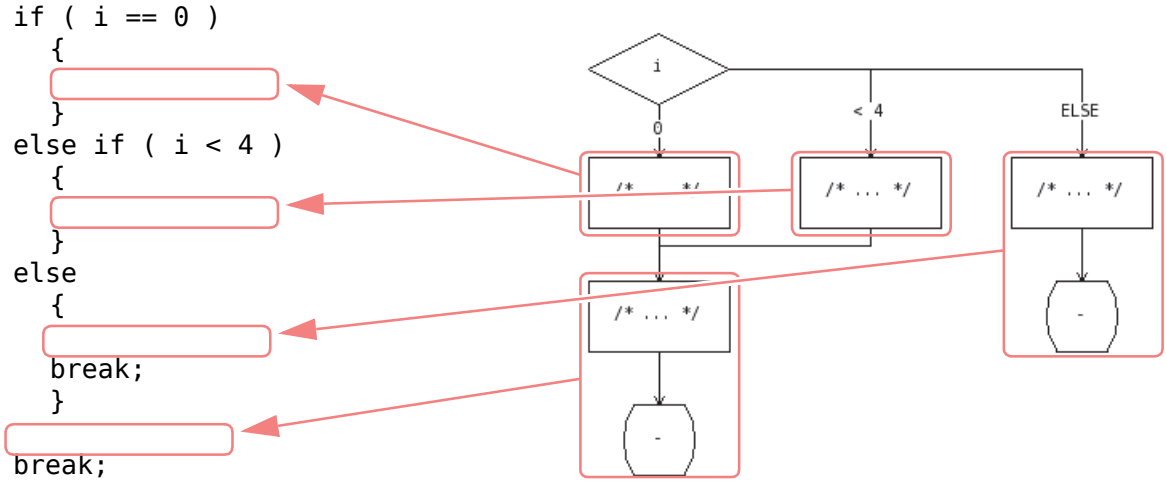
8.2.7.27 Decision

A decision is translated to a simple sequence of `if` / `else if` statements, as shown in the following example:

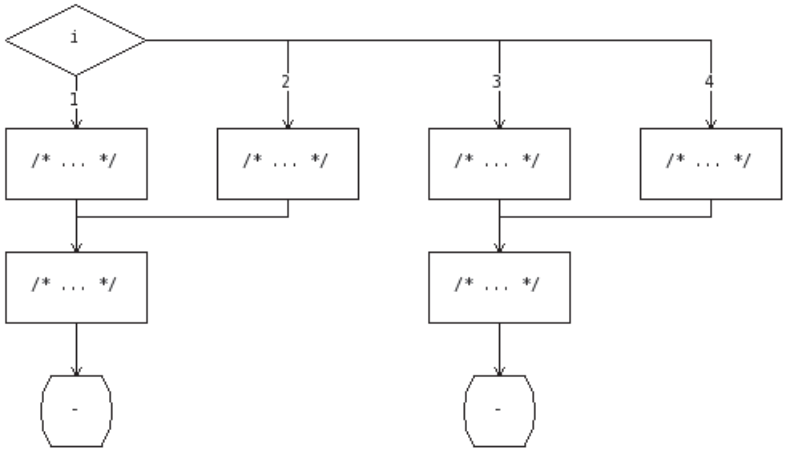


The order of the `if` statements are always the graphical order of the decision branches, so in the example above, the `if ( i < 4 )` will never be generated before the `if ( i == 0 )`.

If branches in the decision rejoin, the generated code will avoid the use of `goto` statements as much as possible, as in the following example:



The code for the common part for the branches `i == 0` and `i < 4` is generated after the `if` / `else if` block to avoid a `goto`. Some cases will however generate a `goto` anyway, when there is no way to avoid it, like in the following example:



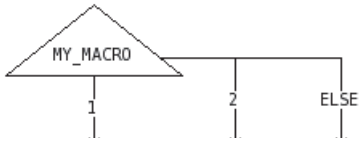
In this case, one of the common parts for either `i == 1` and `i == 2`, or `i == 3` and `i == 4` has to generate a `goto` to be handled.

8.2.7.28 Transition option

Here are the translations for a transition option in all cases:

Table 22: Timer reset symbol translations

Language	Symbol contents	Translation

SDL RT		`#if MY_MACRO == 1`
		`...`
		`#else`
		`#if MY_MACRO == 2`
		`...`
		`#else`
		`...`
		`#endif`
		`#endif`
		In the general case, the value of the expression in the transition option symbol is tested via `#if` directives.

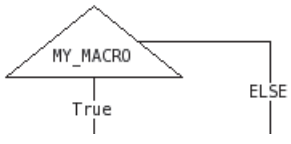
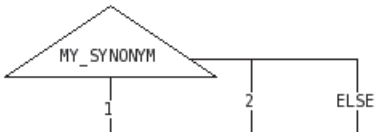
		`#ifdef MY_MACRO`
		`...`
		`#else`
		`...`
		`#endif`
		The case where the values on the branches are booleans is a special case in SDL-RT: The generated code will then only test the macro's existence; and not its value.

Table 22: Timer reset symbol translations

Language	Symbol contents	Translation

		`#if MY_SYNONYM == 1`
		`...`
		`#else`
		`#if MY_SYNONYM == 2`
		`...`
		`#else`
		`...`
		`#endif`
		`#endif`
		In SDL, the value for the synonym is always tested and there is no special case for boolean values.

8.2.7.29 Connector out (JOIN)

Here is the translation for a connector out symbol:

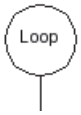
Table 23: Connector out symbol translation

Symbol contents	Translation
	`goto Loop;` The `Loop` label is defined when the corresponding connector in is encountered.

8.2.7.30 Connector in (label)

Here is the translation for a connector in symbol:

Table 24: Connector in symbol translation

Symbol contents	Translation
	`Loop:`

8.2.7.31 Nextstate

Here is the translation for a nextstate symbol:

Table 25: Nextstate symbol translation

Symbol contents	Translation
	`RTDS_SDL_STATE_SET(StateX)` The macro `RTDS_SDL_STATE_SET` is defined in `RTDS_MACRO.h`; The constant `StateX` is the one generated for the state in `RTDS_gen.h` (see “Generated constants and prototypes (RTDS_gen.h)” on page 147).

8.2.7.32 Process kill

Here is the translation for a process kill symbol:

Table 26: Process kill symbol translation

Symbol contents	Translation
	`RTDS_PROCESS_KILL` The macro `RTDS_PROCESS_KILL` is defined in `RTDS_MACRO.h`.

8.2.8 Memory allocation

The code directly generated by RTDS contains no dynamic memory allocation, except in the macros sending a message with parameters: Since a transport structure is needed to contain the parameters, this one is dynamically allocated before sending the message. The allocated buffer is freed by the receiver after the message has been treated.

If dynamic memory allocation is not desirable, some RTOS integrations offer the possibility to avoid it. Please note that this is indeed a feature of the integration itself: If the code in the integration uses dynamic memory allocation, it is useless to generate a code that doesn't.

Integrations supporting this feature will have the option `malloc` set to `forbidden` in their `DefaultOptions.ini` file (see “Mandatory files” on page 140). In this case, the code generation will do the following:

- In addition to the transport structures generated for the messages, a type called `RTDS_MessageData` will be generated, which is a union of all transport structures generated for all messages.
- The type for the buffer for message parameters in the macro `RTDS_MSG_DATA_DECL`, which is normally a simple `unsigned char*`, will be changed to `RTDS_MessageData`.
- The macros for sending messages will be changed to use this statically allocated `RTDS_MessageData` for the message parameters instead of performing a dynamic memory allocation.

This way, neither the integration nor the generated code will contain any dynamic memory allocation. So if the code in the diagram doesn't contain any either, there won't be a single one in the generated executable.

In Real Time Developer Studio V4.3, the only integration supporting this feature is the integration `rtosless`, used to generate fully scheduled systems without a RTOS.

8.2.9 Build process

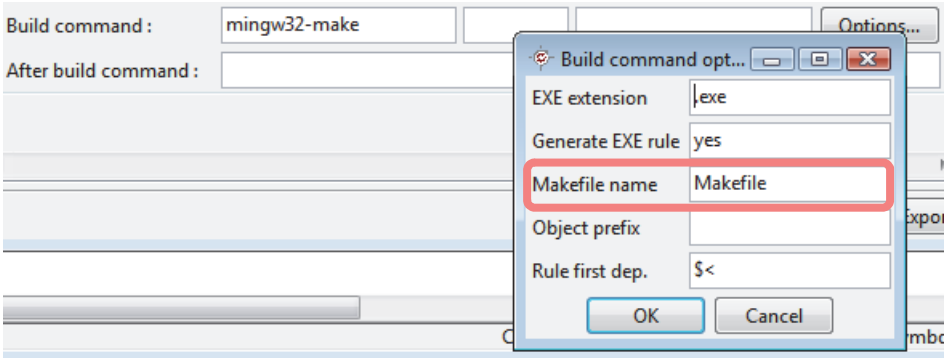
The build process is usually done in two steps:

- If needed, a first partial makefile is generated allowing to produce a file used by RTDS to generate the code for message encoders and decoders. These are used only when the support for the MSC Tracer or the RTDS debugger is turned on. As this step uses the same mechanisms as the actual build, it is described in the last paragraph of this section ("Pre-build action: Message encoders & decoders generation" on page 180).
- The actual makefile is generated.
- The build process is launched using the generated makefile.

8.2.9.1 Makefile generation principles

The process for the generation of the makefile is the following:

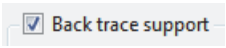
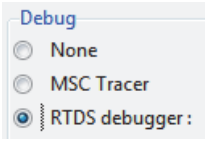
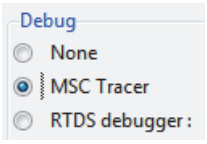
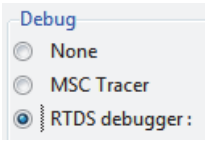
- The name for the makefile itself is taken from the build command options:



- Makefile variables are created to hold the options found in the generation options, such as the code generation directory, the RTOS integration integration directory, the preprocessor, compiler and linker commands, and so on...
- RTDS figures out the options to use for each stage: preprocessing, compiling, linking. It does so by identifying which services are required depending on the options activated in the "Debug/trace" tab of the generation options. To each of

these services corresponds a directory in `$RTDS_HOME/share/ccg`. Here are the available services:

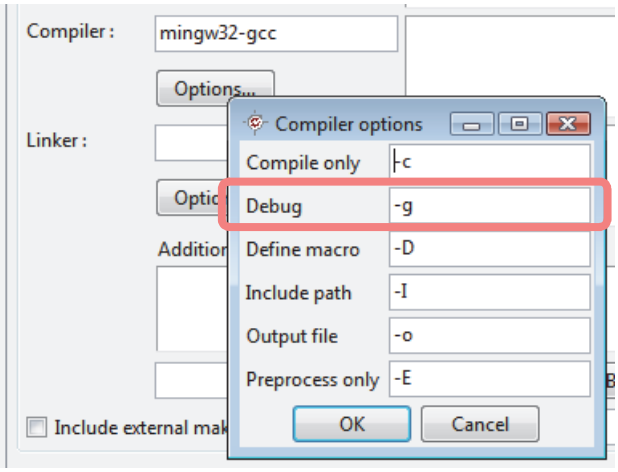
Table 27: Code generation services

Service	Directory	Description	Activated when:
Backtrace	`…/backtrace`	Allows to record the backtrace of a given number of events in a buffer within the debugged program.	
Command interpreter	`…/commandinterpreter`	Allows to interpret the commands that can be sent by the RTDS debugger to the running system.	
Trace format	`…/formattrace`	Allows to format the events that happen in the system in a format suitable to send to an external tracer, like the RTDS debugger or the MSC Tracer. Requires a socket connection with the debugged program to send the trace. Not required when the backtrace is active, since the corresponding buffer is then directly read by RTDS.	 or 

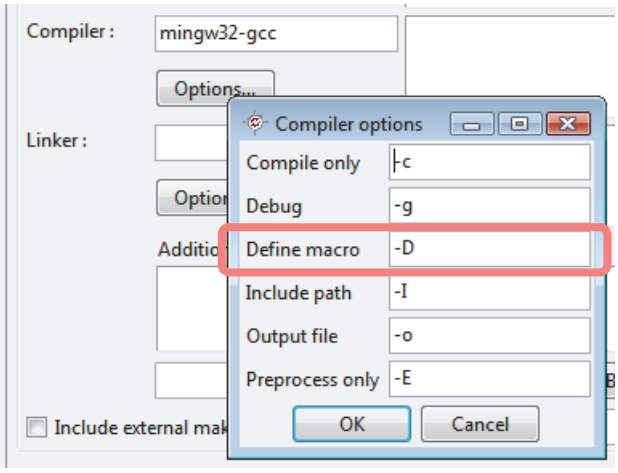
- When the required services are identified, RTDS reads all the files `DefaultOptions.ini` and `addrules.mak` in the directories for all services, the RTOS integration directory and the common directory (`$RTDS_HOME/share/ccg/common`). These files are those described in section “Mandatory files” on page 140.
- In all identified `DefaultOptions.ini` files, RTDS looks for the ones containing the following sections:
  - `[common]` in all cases;
  - `[tracer]` if the “Debug” option is set to “MSC Tracer”;
  - `[debug]` if the “Debug” options is set to “RTDS debugger”.

  These sections are used to produce the preprocessor and/or compiler options, as described below.
- If any of these sections contains an option named debug set to 1, the compiler option for producing a debug build is used. This options can be found in the gen-

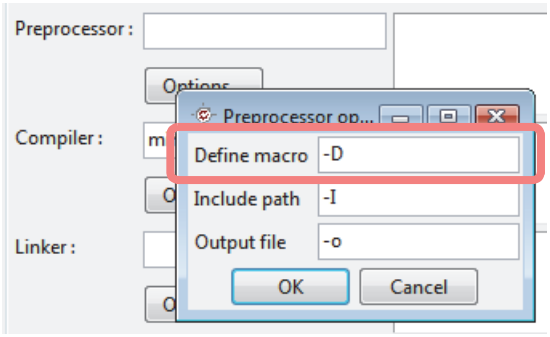
eration options dialog by using the “Options...” button under the compiler name in the “Build” tab:



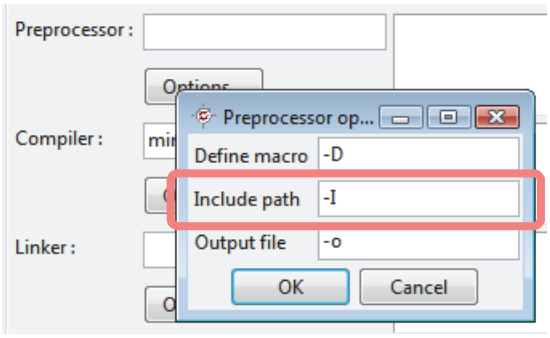
- The options `defines` in all considered sections in all `DefaultOptions.ini` files are parsed and compiler options defining all macros in all options are generated, using the “Define macro” option in the compiler options:

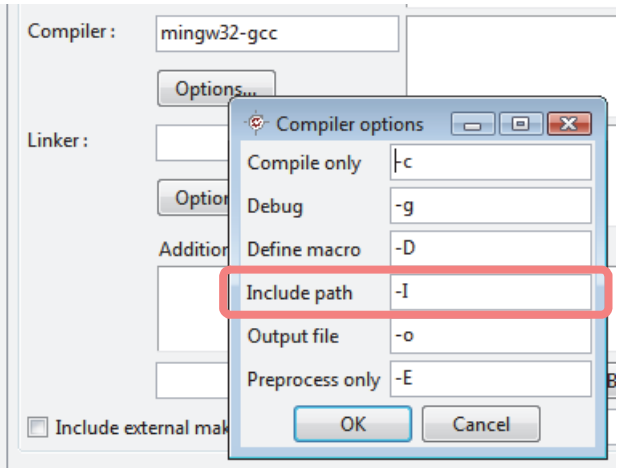


If a preprocessor command is defined in the generation options, the same option is added to the preprocessor options in the makfile, using this time the “Define macro” option from the preprocessor options in the generation options:



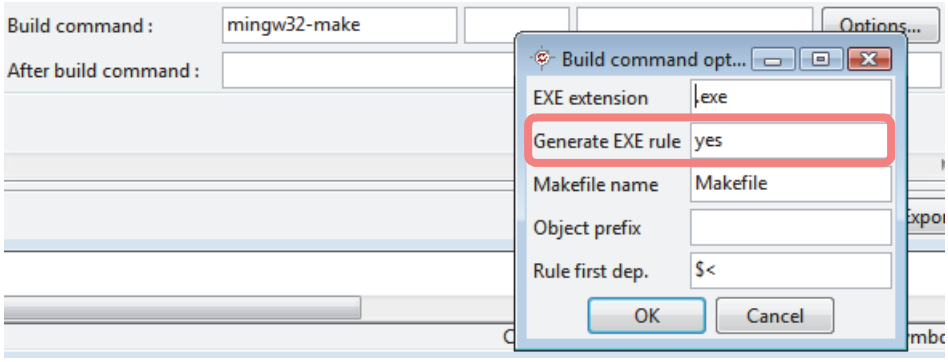
- The same is done for all options `includes` found in all considered sections in all `DefaultOptions.ini` files, using this time the option for the include path in the preprocessor and/or compiler options:





- All the object files that will have to be linked together to produce the final executables are figured out. These files are those for all generated source files, and all those listed in the files `addrules.mak` for the RTOS integration and all identified services.
  An additional file is added if no process named `RTDS_Env` is generated, and if the option “Generate environment process” is checked in the “Code gen.” tab of the generation options. In this case, the file `RTDS_Env.c` from the RTOS integration is integrated in the build.
- The list of additional files to link are copied from the generation options to the makefile if necessary.
- The external makefile set in the generation options is included in the makefile if necessary.
- The rule for the final link is written to the makefile, using the information described above. This rule is only written if the option “Generate EXE rule” in

the build command options in the generation options are set to “yes” (the default):



- All rules to compile individual source files are then inserted, using the dependencies figured out by the code generation, or found in the `addrules.mak` files. The compilation commands themselves also use the options found in the compiler options (options “Compile only” and “Output file”) and the option “Rule first dep.” found in the build command options.
- Rules to clean the generation directories are inserted.

8.2.9.2 Generated makefile example

Here is a summarized example of a generated makefile, showing where the different parts come from. This makefile has been generated with RTDS debugger support (option “Debug” set to “RTDS debugger” in the “Debug/trace” tab in the generation options), therefore selecting the `formattrace` and `commandinterpreter` services:

```
RTDS_GEN_DIR=c:\users\administrateur\documents\sdl-rt-symbols\ccg[1]
RTDS_CLASSES_DIR=c:\users\administrateur\documents\sdl-rt-symbols\ccg
RTDS_TEMPLATES_DIR=z:\project\workspace\rtds_rel\share\ccg\windows[2]

RTDS_CC=mingw32-gcc[3]
RTDS_LNK=$(RTDS_CC)[4]
RTDS_CC_INCLUDES=-I"." -I"${RTDS_TEMPLATES_DIR}" -I"${RTDS_HOME}\share\ccg\trace\formattrace" \
                 -I"${RTDS_HOME}\share\ccg\common" -I"${RTDS_HOME}\share\ccg\trace\commandinterpreter" \
                 -I"$(RTDS_CLASSES_DIR)"
RTDS_CC_OPTIONS=-D[5]RTDS_FORMAT_TRACE[1] -I[5]"."[2] -I[5]"${RTDS_TEMPLATES_DIR}"[2] \
                -I[5]"${RTDS_HOME}\share\ccg\trace\formattrace"[1] -I[5]"${RTDS_HOME}\share\ccg\common"[3] \
                -g[5][2] -D[5]RTDS_SIMULATOR[2] -D[5]RTDS_CMD_INTERPRETER[4] \
                -I[5]"${RTDS_HOME}\share\ccg\trace\commandinterpreter"[4] -I[5]"$(RTDS_CLASSES_DIR)" \
                -D[5]RTDS_SOCKET_IP_ADDRESS=10.211.55.5[6] -D[5]RTDS_SOCKET_PORT=49250[7]
RTDS_LNK_OPTIONS=

RTDS_RM=cmd /c del

RTDS_OBJECTS = \
    rtds_env.o[1] \
    rtds_start.o[2] \
    pstatesinputscontsigs.o[3] \
    rtds_encdecmsgdata.o[2] \
    RTDS_String.o[4] \
    RTDS_Set.o[4] \
    rtds_os.o[1] \
    rtds_tcp_client.o[1] \
    rtds_cmdinterpreter.o[2] \
    rtds_formattrace.o[3]

RTDS_ADDL_OBJECTS = \
    Z:/Project/WORKSPACE/rtds_rel/share/3rdparty/MinGW/lib/libws2_32.a[8]
```

```
RTDS_EXT_OBJECTS =

RTDS_TARGET_BASE_NAME=StatesInputsContsigs❸
RTDS_TARGET_EXTENSION=.exe❿

all: $(RTDS_TARGET_BASE_NAME)$(RTDS_TARGET_EXTENSION)

$(RTDS_TARGET_BASE_NAME)$(RTDS_TARGET_EXTENSION): $(RTDS_OBJECTS) $(RTDS_ADDL_OBJECTS) $(RTDS_EXT_OBJECTS)
   $(RTDS_LNK) $(RTDS_LNK_OPTIONS) -o❾ "$@" $(RTDS_OBJECTS) $(RTDS_ADDL_OBJECTS) $(RTDS_EXT_OBJECTS)

rtds_env.o: $(RTDS_TEMPLATES_DIR)\rtds_env.c❶ Makefile
   $(RTDS_CC) $(RTDS_CC_OPTIONS) -c -o❺ "$@" "$<"❿

rtds_start.o: $(RTDS_GEN_DIR)\rtds_start.c❷ Makefile
   $(RTDS_CC) $(RTDS_CC_OPTIONS) -c -o❺ "$@" "$<"❿

pstatesinputscontsigs.o:$(RTDS_GEN_DIR)\pstatesinputscontsigs.c❸ $(RTDS_GEN_DIR)\rtds_gen.h❷ \
                        $(RTDS_GEN_DIR)\pstatesinputscontsigs.h❸ $(RTDS_GEN_DIR)\rtds_messages.h❷ \
                        $(RTDS_GEN_DIR)\statesinputscontsigs.h❸ $(RTDS_GEN_DIR)\rtds_string.h❹ \
                        $(RTDS_GEN_DIR)\rtds_set.h❹ Makefile
   $(RTDS_CC) $(RTDS_CC_OPTIONS) -c -o❺ "$@" "$<"❿

(...)

rtds_os.o: $(RTDS_TEMPLATES_DIR)\rtds_os.c❶ Makefile
   $(RTDS_CC) $(RTDS_CC_OPTIONS) -c -o❺ "$@" "$<"❿

rtds_tcp_client.o: $(RTDS_TEMPLATES_DIR)\rtds_tcp_client.c❶ Makefile
   $(RTDS_CC) $(RTDS_CC_OPTIONS) -c -o❺ "$@" "$<"❿

rtds_cmdinterpreter.o: $(RTDS_HOME)\share\ccg\trace\commandinterpreter\rtds_cmdinterpreter.c❷ Makefile
   $(RTDS_CC) $(RTDS_CC_OPTIONS) -c -o❺ "$@" "$<"❿

rtds_formattrace.o: $(RTDS_HOME)\share\ccg\trace\formattrace\rtds_formattrace.c❸ Makefile
   $(RTDS_CC) $(RTDS_CC_OPTIONS) -c -o❺ "$@" "$<"❿
```

Table 28: Origin of makefile parts

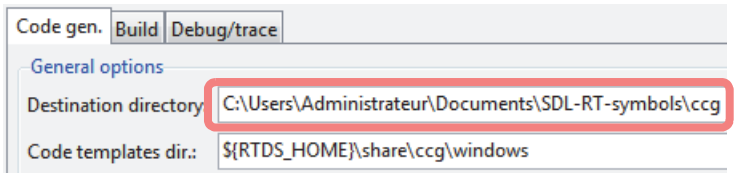
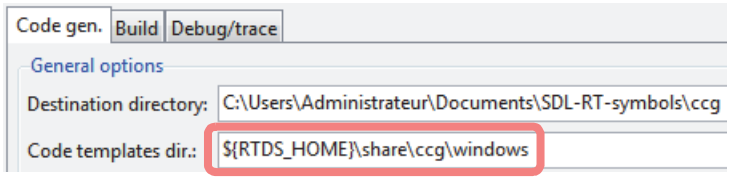
Main origin		Origin details
Generation options	❶	
	❷	
	❸	
	❹	Set to the same thing as the compiler since the field “Linker” in the “Build” tab is empty.

Table 28: Origin of makefile parts

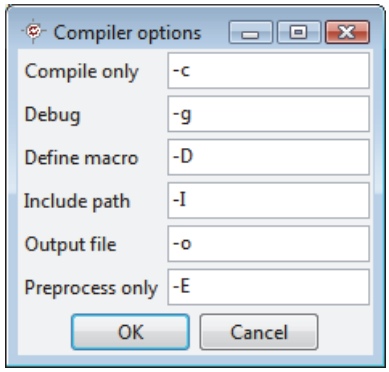
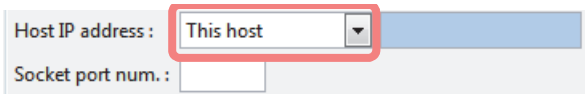
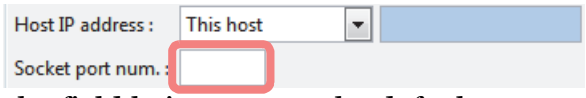
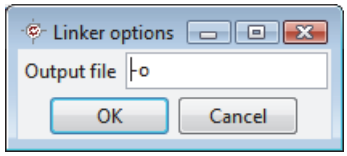
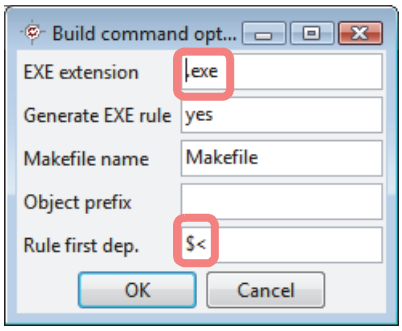
Main origin		Origin details
	❺	All options taken from compiler options in the “Buidl” tab: 
	❻	In the “Debug/Trace” tab:  The IP adress for the current host is figured out and put in the macro `RTDS_SOCKET_IP_ADDRESS`.
	❼	In the “Debug/Trace” tab:  The field being empty, the default port number is used for `RTDS_SOCKET_PORT`.
	❽	In the “Build” tab:
	❾	Linker options in the “Build” tab: 

Table 28: Origin of makefile parts

Main origin		Origin details
	⑩	Build command options in the "Build" tab: 

`GenerationOptions.ini` files	❶	Options taken from the contents of the file in `$RTDS_HOME/share/ccg/trace/formattrace`:
		`[common]`
		`includes=${RTDS_HOME}/share/ccg/trace/format-trace`
		`defines=RTDS_FORMAT_TRACE`

	❷	Options taken from the contents of the file in the RTOS integration:
		`[common]`
		`includes=.;${RTDS_TEMPLATES_DIR}`
		`[debug]`
		`debug=1`
		`defines=RTDS_SIMULATOR`

	❸	Options taken from the contents of the file in `$RTDS_HOME/share/ccg/common`:
		`[common]`
		`includes=${RTDS_HOME}/share/ccg/common`

	❹	Options taken from the contents of the file in `$RTDS_HOME/share/ccg/trace/commandinterpreter`:
		`[debug]`
		`includes=${RTDS_HOME}/share/ccg/trace/commandin-terpreter`
		`defines=RTDS_CMD_INTERPRETER`

`addrules.mak` files	❶	Options taken from the contents of the file in RTOS integration:
		`RTDS_OS.o: $(RTDS_TEMPLATES_DIR)/RTDS_OS.c`
		`RTDS_TCP_Client.o: $(RTDS_TEMPLATES_DIR)/RTDS_TCP_Client.c`

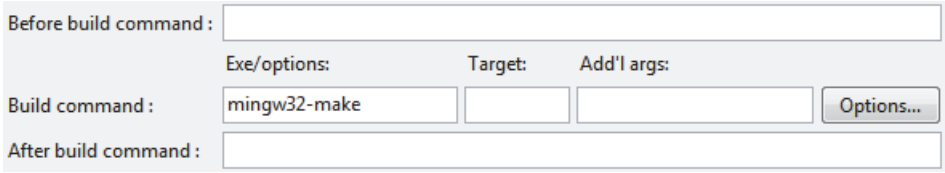
	❷	Options taken from the contents of the file in `$RTDS_HOME/share/ccg/trace/commandinterpreter`: `RTDS_CmdInterpreter.o: $(RTDS_HOME)/share/ccg/trace/commandinterpreter/RTDS_CmdInterpreter.c`

Table 28: Origin of makefile parts

Main origin		Origin details
	❸	Options taken from the contents of the file in `$RTDS_HOME/share/ccg/trace/formattrace`: `RTDS_FormatTrace.o: $(RTDS_HOME)/share/ccg/trace/formattrace/RTDS_FormatTrace.c`
Generated code	❶	Generated environment process, actually taken from the RTOS integration.
	❷	Common generated files, not attached to a diagram.
	❸	Code generated from diagrams.
	❹	Code used to translate SDL concepts, copied from `$RTDS_HOME/share/sdlconv`.

8.2.9.3 Actual build

The build is run by using the options found in the "Build" tab of the generation options:



- If anything is present in the "Before build command" field, it is run first through a shell.
- The actual build command is created by concatenating the values in the fields "Exe/options", "Target" and "Add'l args", separated by spaces. This command is then executed through a shell.
- If anything is present in the "After build command" field, it is run through a shell.

If any of the commands return an error, the build process is stopped. If after the build command, the expected generated executable is not found in the generation directory, the build is stopped too and an error is reported.

8.2.9.4 Pre-build action: Message encoders & decoders generation

To send or receive the message parameters to or from the MSC Tracer or the RTDS debugger, the generated code uses an encoded form. For example, if a message has 2 parameters, an int, and a struct containing the fields x and y, both of type double, the encoding could be:

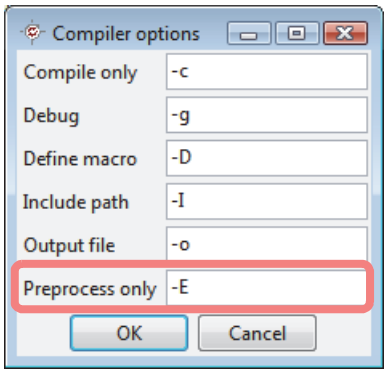
```
|{param1|=3|,param2|=|{x|=1.0|,y|=0.0|}|}
```

When this form is sent to the tracer or debugger, the encoding is actually produced in the debugged program. When it is received from the debugger, the encoded form is received and has to be decoded in the debugged program.

To do so, a set of encoding and decoding functions is generated in the file `RTDS_encDecMessageData.c` and linked with the final executable.

The principles to generate this file is the following:

- A “dummy” C source file named `RTDS_includes4StructMsg.c` is generated in the generation directory. This source file only contains the global includes for the project, and the includes of the header files generated for all agents defining a message.
- A temporary makefile is produced in the generation directory. This makefile has the same name as the final one and the contantes is the same, except it contains a single rule which runs the C preprocessor on `RTDS_includes4StructMsg.c`. If no preprocessor command is set, it uses the compiler command with the “Preprocess only” option:



- A pre-build is launched, using the options set in the “Build command” options, except the target is set to the results of the preprocessing (via a dummy target named `RTDS_STRUCT_MSG`).
- The result of the preprocessing, named `RTDS_includes4StructMsg.i`, is analysed by RTDS to get all types for the message paramaters and all types they use, and so on, recursively.
- RTDS uses the analysis of these types to produce the encoder and decoder functions for all message parameters and write these functions to `RTDS_encDecMessageData.c`.
  The encoder function for the type `T` has the name `RTDS_typeDataToString_T` and produces the string corresponding to a variable of this type in the form described above.
  The decoder function for the type `T` has the name `RTDS_typeStringToData_T` and fills a variable with type `T` with the data found in a string in the format described above.
  Two top-level functions, named `RTDS_messageDataToString` and `RTDS_stringToMessageData`, perform the decoding of any set of message parameters depending on the numerical identifier generated for the message in `RTDS_gen.h` (see “Generated files” on page 138).
- This file `RTDS_encDecMessageData.c` is then added as an additional file to compile in the general build, and the actual build process is run.

8.3 - C++ code generation for passive classes (UML)

The generated code for passive classes described in UML class diagrams is described in RTDS User Manual. Note that the whole code is generated directly by RTDS; No bricks are involved, and no part of any RTOS integration is used. The only external files that are required are those located in `$RTDS_HOME/share/ccg/cpptemplates`.

There is no specific build process for C++ files generated for passive classes: They are integrated in the makefile used for the SDL-RT / SDL diagrams. This means that the compiler specified in the generation options has to be able to compile C++:

- Either it should be a C++ compiler; The code generated by RTDS is guaranteed to compile with a C++ compiler. Note however that some RTOS integrations do not support C++ (e.g CMX).
- Or the compiler should recognize the source file language from its extension. All C files are generated with an extension `.c`, and all C++ files with an extension `.cpp`.

Note that in all cases, all generated C functions are surrounded by an "`extern "C"`" block, so compiling them with a C++ compiler will keep them callable from C.

8.4 - C++ code generation with or without a RTOS

8.4.1 Objectives

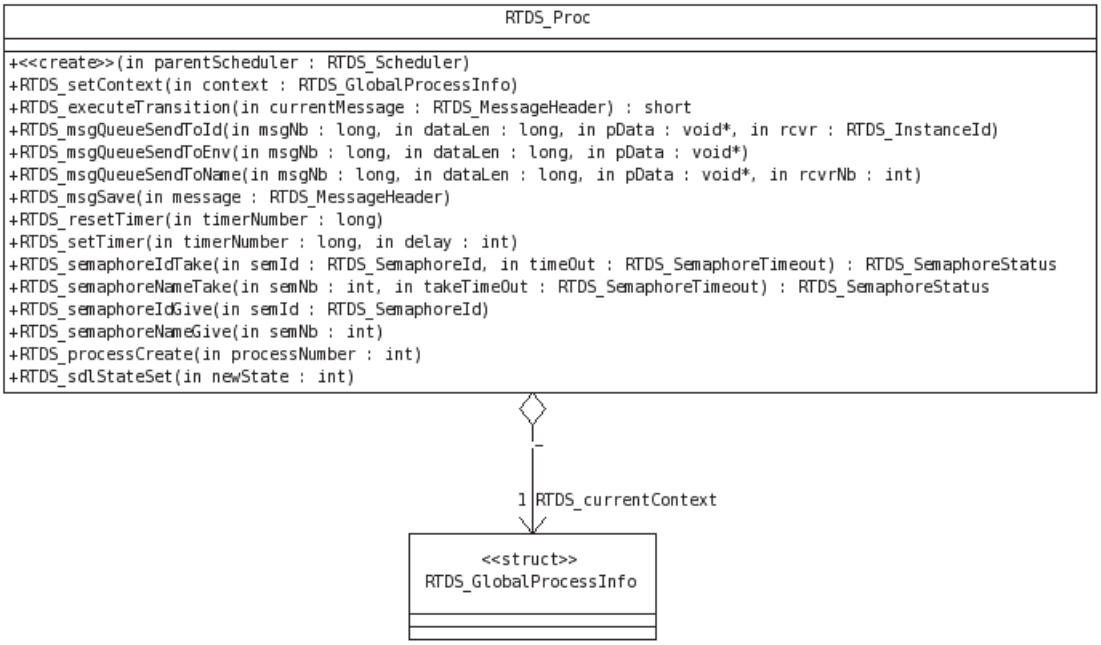
The C++ code generation has been introduced for SDL-RT and SDL projects to be able to put several process instances in a single thread. The instances will then be scheduled in the thread.

The C++ language has been chosen to allow easier handling of process instance context, mainly for its local variables. By generating process local variables as attributes in a C++ class, code in SDL-RT task blocks do not need to be analysed or modified: A local variable `i` in a function implementing a process in “normal” code generation will be handled exactly the same way as an attribute named `i` in a class implementing it in C++.

8.4.2 Principles

In C++ code generation, processes are generated as classes which are all subclasses of a class named `RTDS_Proc`, defined in files `RTDS_Proc.h` & `RTDS_Proc.c`, in `$RTDS_HOME/share/ccg/cppscheduler`. This means that the code generation do not use the bricks defined in the RTOS integration: The code generated for the process will only contain the parts that are specific to the process, and all common parts are handled in the super-class `RTDS_Proc`. Some bricks are still used, but only for the common includes (`RTDS_Include.c`) and to generate the startup task (`RTDS_Startup_*.c`).

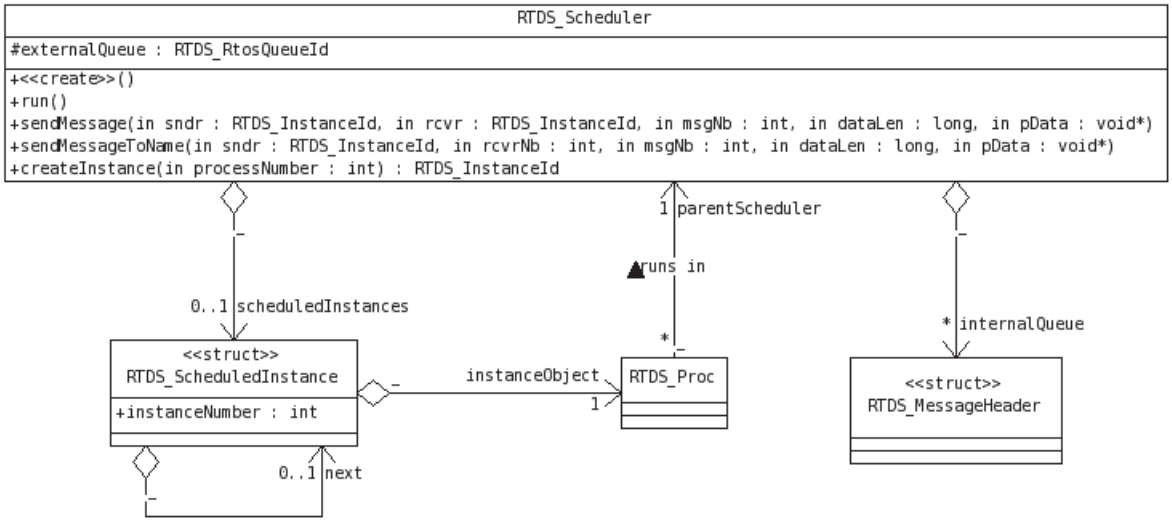
This class is defined as follows:



The class has an operation for each of the RTOS services. For C++ code generation, each call to such a service will actually call this operation, and not the corresponding macro. The operation may itself call the macro when the service actually has to be handled by the RTOS, but may also handle the service locally, typically when the impacted instances are executed within the same scheduler.

To be consistent with the code generation in C, a `RTDS_Proc` instance will always have an attribute named `RTDS_currentContext` which is the same as the local variable with the same name in C code generation, giving the context for the current instance.

Each `RTDS_Proc` instance is executed within a scheduler; Even if the instance has its own thread, it will actually be scheduled alone in this thread. A scheduler is implemented as an instance of a class named `RTDS_Scheduler`, defined in the files `RTDS_Scheduler.h` & `RTDS_Scheduler.c`, in `$RTDS_HOME/share/ccg/cppscheduler`:

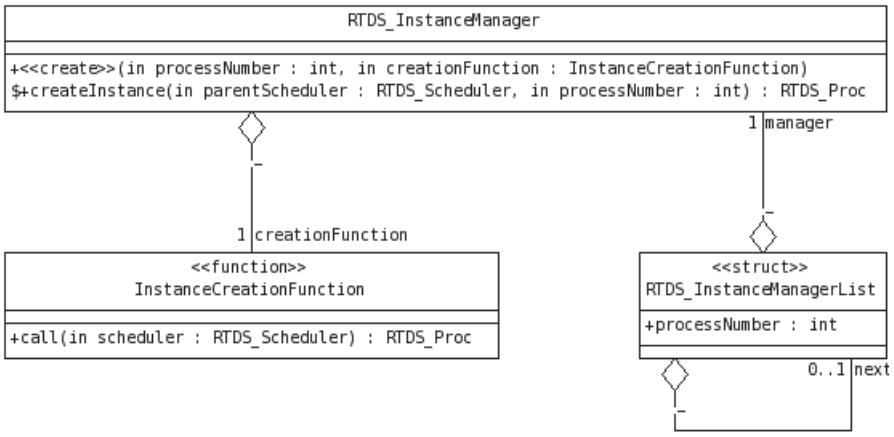


Running the scheduler is done via its `run` operation. This operation will contain the infinite loop getting all the pending messages, either from its `internalQueue` (for messages sent from other instances in the scheduler) or from its `externalQueue` (for messages sent from instances outside the scheduler or from the environment). An incoming message will then trigger the corresponding transition in its receiver instance. A scheduler knows the instances it schedules via a chained list of type `RTDS_ScheduledInstances` and identifies them by an instance number.

The scheduler also implements the operation for sending a message and creating a new process instance:

- To send a message, the scheduler actually figures out wether it schedules the receiver instance or not. If it is, it just puts the message in its internal queue. If it isn't, it uses the regular macros to send the message.
- The creation of a process instance is limited to the processes run in the current scheduler. The process to instantiate is identified by its process number, as the one handled in the C code generation (see "Generated files" on page 138). This

instantiation is a bit tricky, since the class to instantiate has to be dynamically figured out. To do this, a specific class named `RTDS_InstanceManager` is used:



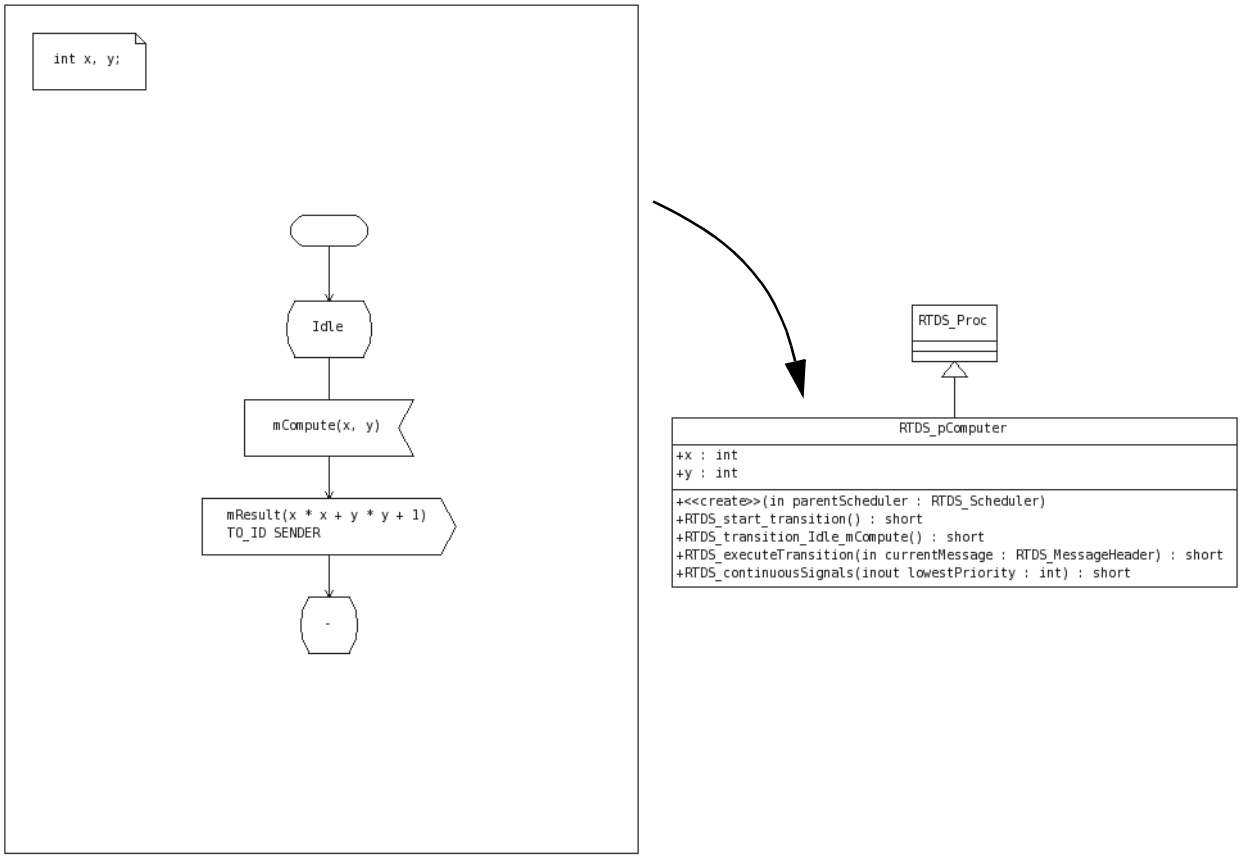
For each generated process, a function is created instantiating the class for the process and return the instance. This function is then recorded in a statically created `RTDS_InstanceManager` instance, along with the process number. The class `RTDS_InstanceManager` records all its instances in the chained list `RTDS_InstanceManagerList`.
Whenever the scheduler has to create an instance of the process identified by processNumber, it calls the class operation `createInstance` on `RTDS_InstanceManager`, which finds the corresponding `RTDS_InstanceManager` instance recorded in the chained list, and calls the corresponding creation function, returning a new instance of the appropriate class.

8.4.3 Generated code

8.4.3.1 Processes

As explained above, all processes are generated as a subclass of `RTDS_Proc`. In addition to those inherited from `RTDS_Proc`, the attributes for this class are the process's local variables, and one operation is generated for each transition. A common entry point is also

generated for all transitions, named `RTDS_executeTransition` and another for all continuous signals, named `RTDS_continuousSignals`:



The operation for each transition contains the same code as the corresponding `case` in the double-`switch` generated in C (see “C translation for symbols” on page 149). There are only a few things handled differently:

- Procedure calls can interrupt the transition and are handled in a special way. See “Procedures” on page 187 for more details.
- In SDL-RT, semaphore takes can also interrupt the transition are may be handled in a special way. See “Semaphore handling (SDL-RT)” on page 189 for more details.
- Process kills cannot be handled as in threaded C code and are indicated via the return value of the operation: If this value is true, the instance has killed itself. The actuall killing is handled by the calling scheduler.

The entry point for all transitions `RTDS_executeTransition` takes the received message as parameter. It records it in the instance context (`RTDS_currentContext`) and calls the appropriate operation for the transition, depending on the received message and on the instance state. It returns the value returned by the transition operation.

The entry point for all continuous signals `RTDS_continuousSignals` takes as parameter the lowest priority for the executed continuous signals in the current state. If there is any continuous signal with a lowest priority left to execute, it executes its code and sets back the lowest priority to this signal’s priority. This operation is called repeatedly by the scheduler until the lowest priority is the same after the call as before it. The return value for the operation also indicates if the instance has killed itself.

Note that contrarily to the C code generation, bricks are almost not used in C++ code generation, as the common part for all processes is very small. The only bricks actually used are:

- `RTDS_Include.c` to ensure that the correct includes are always done for the current integration;
- `RTDS_Startup_begin.c`, `RTDS_Startup_begin_cpp.c` and `RTDS_Startup_end.c` to generate the startup task. `RTDS_Startup_begin_cpp.c` is specific to the C++ code generation and contains code starting a scheduler.

A special header file named `RTDS_ADDL_MACRO.h` must also be present in the integration, defining a few macros that are not needed in C code generation, but are used in a sceduled context:

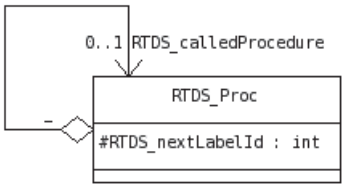
- `RTDS_CREATE_TASK` just creates a task in the RTOS without registering it as a SDL process. This is needed to create tasks for schedulers.
- `RTDS_FORGET_INSTANCE_INFO` removes the descriptor for the current instance from `RTDS_globalProcessInfo` without trying to delete its task. This is needed to forget process instances that are actually run in a scheduler and not in their own task.
- `RTDS_NEW_MESSAGE_QUEUE` creates a new message queue and returns it. This is needed to create the external message queue for a scheduler.
- `RTDS_READ_MESSAGE_QUEUE` reads the next message from a given queue. This macro is used to read the next message on a scheduler's external queue. It is needed because the starndard `RTDS_MSG_QUEUE_READ` macro only works in the context of a process instance.
- `RTDS_SETUP_CURRENT_CONTEXT` gets the current context in a task created for a single instance. In C code generation, this is done in the brick `RTDS_Process_begin.c`, which cannot be used in scheduled tasks.
- `RTDS_GET_MESSAGE_UNIQUE_ID` and `RTDS_RELEASE_MESSAGE_UNIQUE_ID` are used to compute a new unique identifier for a message sent in a scheduler's internal or external queue. These macros are needed because the unique identifier for a message is computed within the standard macros for sending a message in C code generation, and these macros only work in the context of a process instance in a single task.

8.4.3.2 Procedures

Since procedures can have states and handle incoming messages in transitions, they are also managed as a subclass of `RTDS_Proc` and their code is generated the same way as processes.

However, there is a specific problem with procedure calls, as they can do a state change, which will interrupt the transition of the calling process and should give control back to the scheduler. So the caller's transition will be interrupted in the middle of an instruction, and all messages it receives should then be forwarded to the procedure it has called.

To handle this, specific attributes are added to the class `RTDS_Proc`:



- Whenever it calls a procedure, a process or procedures instantiates the generated class for the called procedure and records it in its `RTDS_calledProcedure` attribute.
- The calling transition then calls the procedure's initial transition, which returns either 1 if the procedure did a `RETURN`, or 0 if it went into a state and is now expecting messages.
  In the first case, the procedure's return value if any can be retrieved via its public attribute `RTDS_return_value`, which is generated since its type depends on the procedure.
  In the last case, the calling transition returns control back to the scheduler.
- When the operation `RTDS_executeTransition` is called with an incoming message, the first thing it does it to figure out if a procedure was called by testing the `RTDS_calledProcedure` attribute. If there is one, the message is just forwarded to the procedure by calling its own `RTDS_executeTransition` operation.
- As for processes, the `RTDS_executeTransition` operation then returns 0 if the procedure just changed its state, or 1 if it did a `RETURN`. In this last case, the calling transition in the caller must then be resumed just after the procedure call. To do so, the generated code includes a special mechanism:
  - Whenever a procedure is called, a specific label identifier is stored in an attribute called `RTDS_nextLabelId`;
  - Each transition calling a procedure always starts with the following piece of code:

```
switch(this->RTDS_nextLabelId)
   {
   case 1: goto RTDS_procedure_return_1;
   case 2: goto RTDS_procedure_return_2;
   …
   }
```

  - The generated code for the procedure call itself looks like:

```
RTDS_calledProcedure =
     new RTDS_<procedure>_proc(RTDS_parentScheduler, <parameters>);
RTDS_calledProcedure->RTDS_currentContext =
     this->RTDS_currentContext;
if ( ! RTDS_calledProcedure->RTDS_executeTransition(NULL) )
   {
   RTDS_nextLabelId = <i>;
   return 0;
   }
RTDS_procedure_return_<i>:
<code after procedure call>
```

  The first line creates the instance of the subclass of `RTDS_Proc` implementing the procedure. The constructor parameters are the parent scheduler for the caller and the procedure's own parameters if any.

The second line makes the procedure inherit its context from its caller. This allows to share process-based information such as the process id.
The third line executes the procedure's start transition. If it doesn't do a `RETURN`, the `RTDS_nextLabelId` attribute is initialized to the label id for this procedure call and the caller transition returns.
Whenever the procedure does a `RETURN`, the operation for this transition will be called again, and the `switch/case` described above will do a `goto` to `RTDS_procedure_return_<i>`. The execution will then resume just after the procedure call.
NB: Since all variables are stored in attributes, the caller context is preserved. The only thing that may have changed is the message triggering the transition. This one is actually saved when each transition is executed and restored by the `RTDS_executeTransition` operation if the currently called procedures does a `RETURN`.

8.4.3.3 Semaphore handling (SDL-RT)

Like procedure calls, taking semaphores may interrupt a transition in the middle of its execution. The problem is actually far more complex than for procedures, as the process instance holding the semaphore when a take happens may or may not be in the same RTOS task as the instance attempting the take. And of course, several takes can be attempted by different instances, either in the same RTOS task or not.

Since the only way to handle semaphores properly when instances are not in the same RTOS task is by using the semaphores provided by the RTOS, the choice has been made in RTDS to always use them, even in partially or fully scheduled systems. This may lead to problems as taking a semaphore may block an entire scheduler and not a single process instance. This also can cause problems when using mutex semaphores, as several instances may be seen as a single task. So care should be taken when the system is designed if it mixes semaphores and scheduling.

The only case when semaphores are handled in a specific way is when no RTOS is available. This case is described in details in paragraph "Whole system scheduling with no RTOS" on page 189.

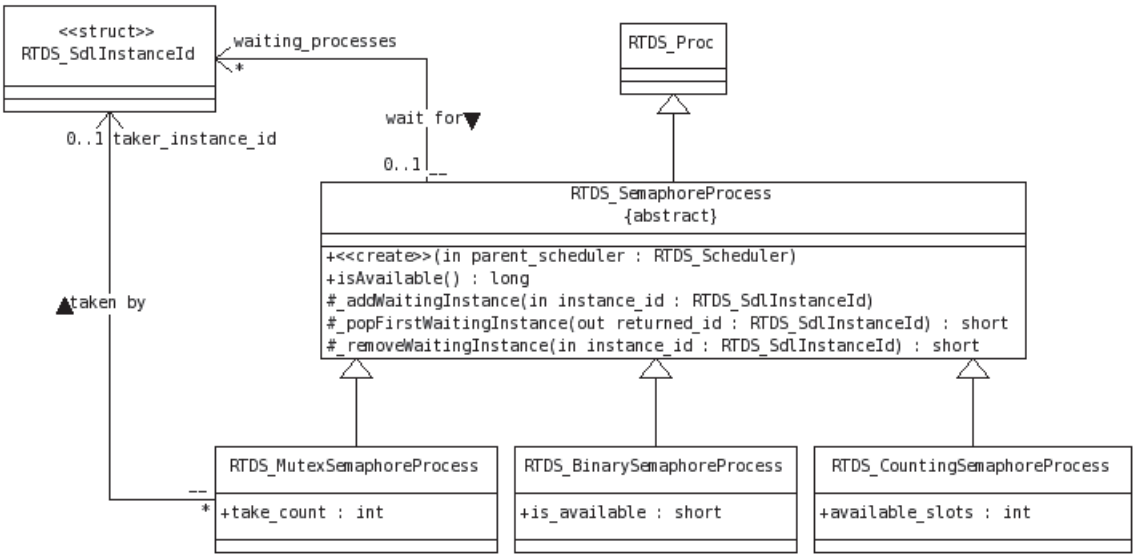
8.4.4 Whole system scheduling with no RTOS

The scheduling of a whole system with no RTOS is handled in a specific integration located in RTDS installation directory, subdirectory `share/ccg/rtosless`. In this integration:

- This integration only supports C++ code generation in fully scheduled mode.
- Only `malloc`, `free` and `memcpy` system calls are used. No other service is required. In the context of debugging, the generated code may however need socket management services (`socket`, `send`, `recv`, ...).
- Message queues are handled as chained lists of messages.
- Message saving is handled by the scheduler, with save queues stored in instance contexts.
- Timers are handled via two specific functions named `RTDS_SystemTick` and `RTDS_incomingSdlEvent`:

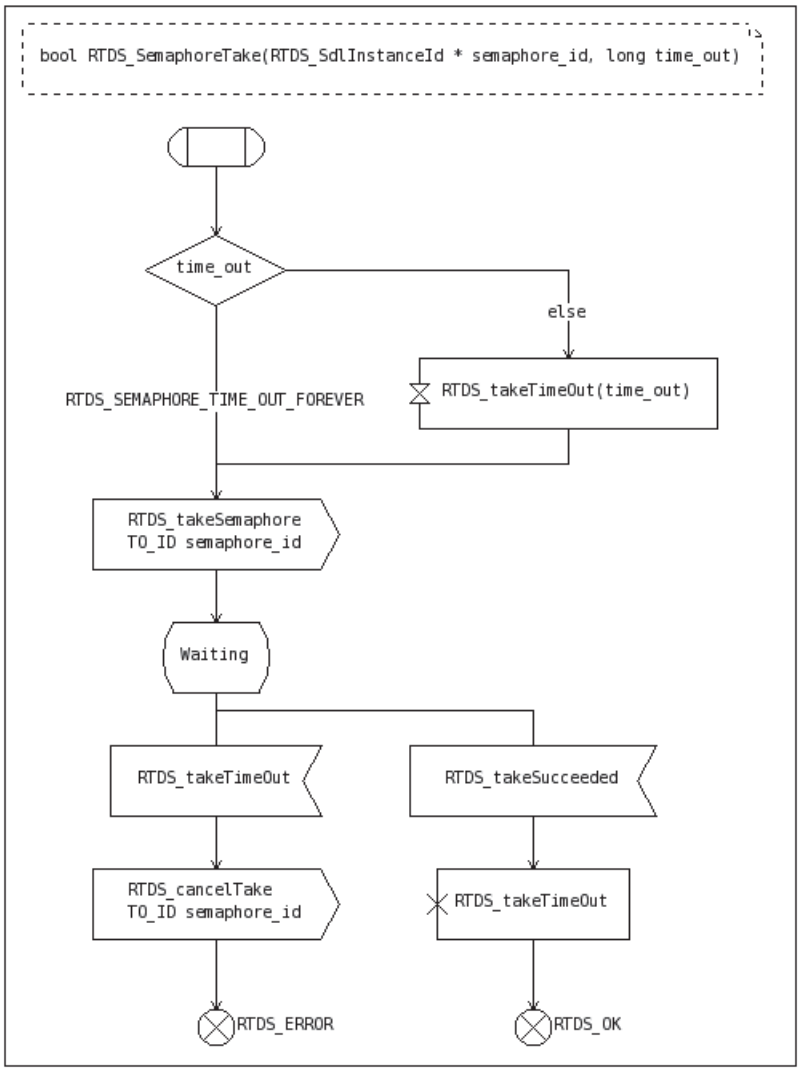
- `RTDS_SystemTick` should be called regularly to make the system time increase. It will typically be called by an interrupt.
- `RTDS_incomingSdlEvent` must be written by the user. It takes as argument a pointer to a preallocated `RTDS_MessageHeader` variable and a time left before the next timer should time-out. The function should acquire external events if any before the time-out time expires. It should then return true if an external message has arrived, or false if the acquisition timed-out.
  A default implementation is provided by RTDS in debug mode, which just calls `RTDS_SystemTick` to make time increase. This implementation should not be used in production code.

- Semaphores are actually handled like pseudo-processes:



Taking and giving semaphores then consist in sending messages to the corresponding pseudo-process: `RTDS_takeSemaphore` to take it, `RTDS_cancelTake` to cancel the take if a time-out occurs, `RTDS_giveSemaphore` to give it back. The answer to a take is also handled via a message, named `RTDS_takeSucceeded`. If the take fails, the semaphore process just doesn't answer.

Taking a semaphore can then be handled via a regular SDL-RT procedure:



This procedure is implemented in the `rtosless` integration in the files `RTDS_SemaphoreTakeProcedure.h` and `RTDS_SemaphoreTakeProcedure.cpp`. It is called in the generated code just like any other procedure (see “Procedures” on page 187).

Giving back a semaphore is implemented by just sending the `RTDS_giveSemaphore` message to the process implementing it.

8.5 - C code generation with RTDS C scheduler

This feature allows to generate a partially or fully scheduled code for a system just as the feature described in “C++ code generation with or without a RTOS” on page 183, except the generated code is in C instead of C++.

8.5.1 Process instance context handling

C++ allows to handle the contexts of the instances transparently, since a variable v can be translated to a class attribute v in a C++ class. This cannot be done in C, so explicit management of process instance contexts is required. This is done in different ways, depending on the language used in the project:

- For SDL projects, a full code analysis is performed and allows to figure out when local variables are manipulated. The local variables are not represented as local variables in the code, but as fields in an explicitely managed process context, passed to each function implementing a transition.
- For SDL-RT projects, no code analysis is performed, so the instance context is handled another way: Each function implementing a transition redefines the local variables and copies them from the instanc context before executing the transition code. When the transition ends, the local variables are copied back to the instance context.
  Please note this has an impact on the memory consumption of the generated code, since each local variable will require twice the space it should consume: One in the instance context, and one as an actual local variable in the currently executing transition. This also forbids some use of local variables, like storing their address anywhere: This would store the address of the duplicate local variable, and not the address of the variable in the instance context.

8.5.2 General architecture

The general architecture of the generated code and built-in scheduler is the same as the one for C++ code generation, replacing the C++ classes by a C `struct` type containing its attributes and a set of C functions replacing the class’s operations. For example, the C++ class for the scheduler becomes a C `struct` type with the same name and the `void run()` operation becomes a function:

```
void RTDS_Scheduler_run(RTDS_Scheduler * scheduler);
```

The C functions are prefixed with the C++ class name to avoid name clashes, and always take an additional first parameter with the corresponding `struct` type, corresponding to the implicit `this` in C++.

Like the C++ code generation, the C code generation for scheduling uses very few bricks, and needs the same additional macros defined in `RTDS_ADDL_MACRO.h` (see “Processes” on page 185). For the startup task, the brick `RTDS_Startup_begin_cpp.c` is replaced by the brick `RTDS_Startup_begin_c.c`.

There are a few differences in the way some features are implemented, due to the lack of some possibilities in C:

- Classes constructor are translated to specific functions: `RTDS_Scheduler_init` for `RTDS_Scheduler` and `RTDS_Proc_createInstance` for `RTDS_Proc`. These construc-

tors accept as first parameter a pointer on a `RTDS_Scheduler` or `RTDS_Proc` (respectively) which should reference a preallocated variable.

- Process instance creation are handled differently than in C++: Since it is impossible to use the same mechanism of a class registering its statically created instances, creating an instance of a process `p` is always explicitely done via a function named `RTDS_Proc_p_createInstance`. Therefore, there is no function `RTDS_Scheduler_createInstance` corresponding to the `createInstance` operation in the `RTDS_Scheduler` class: the function `RTDS_Proc_p_createInstance` calls `RTDS_Proc_createInstance` to initialize fields defined in the `RTDS_Proc struct` type, then calls a specific function called `RTDS_Scheduler_registerInstance` in its parent scheduler, which only registers the already existing instance as scheduled by this scheduler.
- Since inheritance is not available in C, another mechanism is used to specialize `RTDS_Proc` into the instances of the different generated processes: the `RTDS_Proc` type contains an additional field called `myLocals` with the type `RTDS_ProcessLocals`, which is a generated type. `RTDS_ProcessLocals` is a union between types generated for each process, named `RTDS_Proc_<process name>_Locals`, containing one field for each local variable in the process. The name for the option in the union is the name of the process itself. So a local variable `v` in a process `p` can be adressed in the corresponding `RTDS_Proc proc` by `proc.myLocals.p.v`. The type `RTDS_ProcessLocals` is generated in a special header file, called `RTDS_all_processes.h`, only present in C code generation for scheduling.

8.5.3 Whole system scheduling with no RTOS

As for C++, a specific integration named `crtosless` is provided with RTDS allowing to generate code from a SDL or SDL-RT project without any RTOS. This integration is based on exactly the same principles as the one available for C++, described in “Whole system scheduling with no RTOS” on page 189, except the code is written in C and not in C++.

8.5.4 Limitations

The specialization of process classes that was naturally handled by the C++ code generation does not work in C. So specialized process classes do not work in C code generation for scheduling.

8.6 - C code generation with external C scheduler (SDL only)

This feature allows to generate C code for a single process from RTDS so that this code can be easily integrated within an existing external scheduler. This is an option in the C code generation for scheduler, so it only works for SDL projects, not SDL-RT ones.

The typical setup for this feature is a set of RTDS projects, each containing a single process. Data and signals are usually shared via a package exported as a subtree (see RTDS User Manual). Each project has its own generation options with the 'Partial code generation' checkbox turned on. It is better to have each project generate into its own code generation directory and gather generated files from each one into a single directory later (e.g via a global makefile).

With this feature turned on in the generation options, RTDS avoids as much as possible to generate any global information and the generated files are almost only the ones for the process itself. However, there are still a few global definitions that are required, and a few files are generated differently than in the "normal" code generation:

- RTDS still requires global identifiers for the messages sent and received by the processes, as well as for the processes themselves. It also needs to generate a correct definition for the type `RTDS_ProcessLocals` used to store instance local variables.
  Due to this requirements, the list of all message and process names used by RTDS that will end up in the final system must be specified in the generation options. For this, two options are available, named respectively 'File containing all process names' and 'File containing all message names', that should be set to the full name of a file containing all process and message names respectively, one name per line, in any order. The names appearing must be only the ones visible to RTDS: Processes defined in and generated by RTDS, and messages sent or received by these processes. If a name is missing, or if a name of a process or message not generated by RTDS appears in the files, the generated code may fail to compile.
  Note that using these files is not mandatory: If the generated code for each RTDS project is compiled separately, no global identifier for processes or messages will be needed. Note however that in this case, the generated identifiers will be different for each project, and that they cannot be used to identify the process or message globally.
- Some global files such as `RTDS_gen.h` will still be generated in all projects. If the lists of all process and message names are used, these will actually always be the same in each code generation.
- Some information will be generated in different places in full and partial code generation. This is typically the case for state numerical identifiers: They are generated globally in `RTDS_gen.h` in full code generation, but are generated directly in the C source file for the process using them in partial code generation.
- Some global files in full code generation cannot be global anymore in partial code generation and therefore have their name changed. This is the case for the file `RTDS_messages.h`, containing the macros used to send and receive messages with structured parameters. This file is renamed `RTDS_<project name>_messages.h` in partial code generation.

- Since the whole architecture is not known by RTDS when it generates the code, a message output should always be specified with no receiver specification (no `TO`, no `VIA`). They will always be translated to the call of a specific macro:
  `RTDS_MSG_SEND_<message name>(<parameter1>, <parameter2>, …)`
  These macros have to be provided externally, as the mechanism for sending messages should be the one used in the external scheduler and not RTDS's.

The table below shows a summary of the files generated in full and partial code generation. A file generated and needed is marked with a ✓; a file generated, but not needed in the final build is marked with a ✓, and a file not generated at all is marked with a ✗:

Table 29: Generated files in full and partial code generation

File name	Code generation		Comment
	Full	Partial	
`<system>.h`	✓	✓	Definitions in the project, mainly data types shared by all processes. In partial code generation, all these files are used, so system names must be different for each project.
`<process>.h`	✓	✓	Definitions in the process itself.
`<process>_decl.h`	✓	✓	Declaration for the `struct` type implementing the process.
`<process>_locals.h`	✓	✓	Definition of the type `RTDS_Proc_<process>_locals`.
`<process>_tmpvars.h`	✓	✓	Temporary variables used by the process. Included in the process body.
`<process>.c`	✓	✓	Implementation for the process itself. Contains numerical identifiers for states in partial code generation, but not in full one.
`RTDS_all_processes.h`	✓	✓	In partial code generation, based on the list of all process names specified in the generation options.
`RTDS_gen.h`	✓	✓	Based on the list of all process and message names specified in the generation options. Contains numerical identifier for states in full code generation, but not in partial one.

Table 29: Generated files in full and partial code generation

File name	Code generation		Comment
	Full	Partial	
`RTDS_messages.h`	✓	✗	In full code generation, contains types and macros to handle sending and receiving messages with structured parameters.
`RTDS_<project>_messages.h`	✗	✓	Equivalent of `RTDS_messages.h` in partial code generation, but only contains the messages used in the current project.
`RTDS_Start.c`	✓	✗	Startup task; not generated in partial code generation.
`Makefile`	✓	✓	Not used for build in partial code generation, but needs to be generated to build the file `RTDS_<project>_messages.h`.
`RTDS_gen.ini`	✓	✓	Same comment as for `RTDS_gen.h`; not used in partial code generation.
`RTDS_gen.inf`	✓	✓	Same comment as for `RTDS_gen.h`; not used in partial code generation.

8.7 - Integration with external C code

8.7.1 Function call

In SDL-RT projects, calling an external C function is quite straightforward: If you include the C header and source files for the function, you can just include the header in any diagram and call the function directly, since the source file will be automatically compiled and linked by the build process. Calling a C function in an external library is simple too: It just requires the header file to be accessible for include, either by putting it in the project, or via a compiler option, and the library must be added as an additional file to link.

In SDL projects, C functions cannot be called directly. To be able to call external C code, an operator in a SDL `NEWTYPE` must be defined and implemented in C. This operator's signature will be translated to C according to the rules described in "OPERATORS conversion" on page 114. Its implementation should be put in an external object file or library, and added in the generation options in the additional files to link.

In both cases, if a RTOS integration is used, please note that the function will be executed in the calling task's context, so the stack size must be sufficient to handle all local variables.

8.7.2 Message exchange

Communication with SDL-RT active objects described through finite state machines is done through messages. Both communication ways are possible: From the SDL-RT / SDL system to the outside, or from the outside to the system.

8.7.2.1 Sending messages from the SDL-RT / SDL system

The outside of the system is represented in RTDS by the environment process, called `RTDS_Env`. This environment process can be either generated automatically if the option "Generate environment process" is checked in the generation options, or implemented explicitely by creating a process named `RTDS_Env` at system level. In this case, there should be no channels connecting to the system diagram's frame; All should go to the process `RTDS_Env`. This way, if a message is sent `TO_ENV` / `TO ENV` in the system, it will automatically be sent to the process named `RTDS_Env`. This process should then handle the communication with the outside of the system:

- In SDL-RT, it can do so by calling C functions.
- In SDL, it can do so by calling a procedure declared as external with its implementation written in C.

In SDL-RT, there is also an easier way to communicate with the environment: If a message is sent `TO_ENV` with a macro name behind the `TO_ENV` clause, the actual communication can be handled via this macro. This feature is also designed to work both on target and in debugging environment:

- On target, the generation profile should have its option "Communicate with environment via macros" checked. This way, whenever a message is sent `TO_ENV` with a macro name, the macro will actually be called.
- In debugging environment, another generation profile should be used, with the option "Communicate with environment via macros" unchecked. This way, when

a message is sent TO_ENV with a macro name, the macro is actually ignored and the message is sent to the process RTDS_Env.

8.7.2.2 Sending messages to the SDL-RT / SDL system

8.7.2.2.1 Identifying the receiver by its name

The best way to send message to an instance of a process with a given name is via the macros generated for message output, described in “Additional generated types & macros for message handling” on page 147. So for example, for a message declared like follows:

```
MESSAGE m_query(int, MyStructType)
```

in SDL-RT, or:

```
SIGNAL m_query(Integer, MyStructType)
```

in SDL, sending a message to a process named my_process with the parameters i and str should be done via:

```
RTDS_MSG_SEND_m_query_TO_NAME(“my_process”, RTDS_process_my_process, i, str);
```

This macro is defined in the generated file RTDS_messages.h, so it must be included in the caller.

However, this macro is supposed to be called in the context of a RTDS process instance, so it requires some variable to be defined:

- The macro uses the context for the sender to identify it. This context is always present in RTDS process instances in a variable named RTDS_currentContext, with the type RTDS_GlobalProcessInfo*. So such a variable has to exist, or the macro call won’t compile.
  In addition, it is safer to initialize the whole RTDS_currentContext variable with zeros before sending any message. This way, if the message receiver ever attempts sending a message back to the sender, the generated code will handle the error cleanly, and not crash with a segmentation fault or send the message to a random instance. This initialization can be done safely via a memset, or by using calloc instead of malloc if the variable is dynamically allocated.
- The macro also uses a buffer to contain message parameters that has to be declared in the caller. This declaration is done through the macro RTDS_MSG_DATA_DECL, which has to be called before the message output macro can be used.

Note that for messages that do not accept any parameter, no macro RTDS_MSG_SEND_… is generated, and that the generic macro found in RTDS_MACRO.h should be used. For example, for a message m_ping with no parameters:

```
RTDS_MSG_QUEUE_SEND_TO_NAME(
  m_ping, 0, NULL, “my_process”, RTDS_process_my_process
);
```

8.7.2.2.2 Other ways to identify message receivers

There might be cases where identifying the reciver of a message by its name is not convenient or impossible:

- If the message is sent to a specific instance of a process that has several ones, the macros shown above will take a random instance of this process, which might not be the one that is supposed to be the receiver.

- In most RTOS integrations, finding an instance of a process with a given name requires access to a global list of running instances, which is protected by a semaphore. This means that the call can block on this semaphore. This might not be possible in some contexts, for example if the message is sent by an ISR.

In the first case, there must be a way for the instance that should receive the message to transmit its `RTDS_SdlInstanceId` to the external sender. It can do so by sending a message to the environment: The field sender in the received `RTDS_MessageHeader` will automatically contain the `RTDS_SdlInstanceId` for the sender instance. Then the external message can be sent via the `..._TO_ID` macros, similar to the ones described above. In other very specific cases, the instance might have to publish its `RTDS_SdlInstanceId` in a global variable. This is not the recommended way, but it is sometimes the only solution.

The second case is trickier, since almost all macros used for message sending have to manipulate global variables, which are always protected by semaphores. This is done for example to check if the receiver instance still exists. In this case, the only way to send a message is to avoid using the high-level macros and to rely on the low-level mechanisms in the RTOS integration. This requires to analyse how the macros and the functions they call are implemented and do the same thing in the external code, removing all critical sections. This means that the external code will be RTOS-specific.

8.8 - RTOS integrations

8.8.1 Common features

8.8.1.1 Process information handling

The relevant information on a process is stored in a structure that is placed in a global chained list. When a process is created, a pointer to its structure is passed. The structure of an element in the chained list is `RTDS_GlobalProcessInfo` described in “Common types - RTDS_Common.h” on page 142. The RTDS debugger uses this chained list to display internal information regarding a process.

8.8.1.2 Semaphore information handling

Semaphores are defined and manipulated using their names in the SDL description. Since most of the time the operating system does not identify semaphore by names but by addresses, the generated code updates a list of all the semaphore ids and their corresponding names. The structure of an element is `RTDS_GlobalSemaphoreInfo` and is usually defined in the integration's `RTDS_BasicTypes.h` file (see “RTOS-specific types - RTDS_BasicTypes.h” on page 146). It contains the semaphore id and the related semaphore name.

When a semaphore is created, a new element is added in the chained list. When a semaphore is deleted the corresponding element is removed. When a semaphore is taken or given away the chained list is read to determine the semaphore address.

8.8.1.3 Saved messages handling

The handling of saved messages is very complex for the following reasons:

- A saved message should only be considered when the SDL state changes otherwise it would be saved again;
- The order in which the messages arrived must be respected. In fact the SDL concept is more or less like leaving the message in the queue if it is saved. So when the SDL state changes, the saved messages must be considered first.

To do so the RTDS kernel manipulates 2 save queues:

- One queue to write messages to save (field `writeSaveQueue` in type `RTDS_GlobalProcessInfo`);
- One queue to read messages that have been saved (field `readSaveQueue` in type `RTDS_GlobalProcessInfo`).

Both are chained list of messages, with the type `RTDS_MessageHeader` (see “Common types - RTDS_Common.h” on page 142 for more details).

The basic algorithm is the following:

- When executing, the automaton will first check if there is any message in the `readSaveQueue`. If not it will read from the RTOS message queue.
- When a message needs to be saved, it is put in the `writeSaveQueue`.
- When the SDL state changes:
  - The `readSaveQueue` is put at the end of the `writeSaveQueue` to keep the messages order;

- The `readSaveQueue` takes the value of the `writeSaveQueue` that now contains all saved messages;
- The `writeSaveQueue` is emptied.

8.8.1.4 Timers information handling

Handling timers with the semantics defined in SDL-RT or SDL is tricky to do with an RTOS for the following reasons:

- When a SDL-RT or SDL timer goes off, it generates a message sent to the task that started the timer. This is not how most of operating systems work. For example, in VxWorks, timers are implemented using watchdogs that call a function when it goes off.
- When a SDL-RT or SDL timer is cancelled, it is cancelled even if the timer message is already in the task's queue. That makes implementation quite tricky to be sure the timer is properly cancelled.

The RTDS timer implementation supports SDL timer concepts and works the following way:

- Each task has its own timer information chained list (field `timerList` in `RTDS_GlobalProcessInfo` - see "Common types - RTDS_Common.h" on page 142);
- Each element of the chained list has the type `RTDS_TimerState`, defined in the integration's `RTDS_BasicTypes.h` file, and which usually contains at least the following information:
  - The timer id, which is the message number of the timer;
  - A unique id so that if the same timer is started 20 times, it is possible to distinguish the timer instances. This identifier is actually unique within a task, since it is not possible to start the same timer several times within a task;
  - The SDL instance id of the instance that started the timer;
  - The state of the timer (is it valid or has it been cancelled);
  - The watchdog id of the timer instance.

When a timer is started:

- All other instances of the same timer are cancelled;
- Aunique timer id is calculated;
- A watchdog is created;
- A new element of the chained is allocated and inserted in the list;
- The watchdog is started with the new element as a parameter of the timeout function.

When a timer is cancelled:

- The corresponding information is retrieved from the chained list;
- The watchdog is stopped if possible.
  - If it couldn't be stopped because it already was, this means the watchdog already went off. The corresponding element in the chained list is then marked as cancelled;
  - If it could be stopped, the element is removed from the chained list.

When the timer message is read from the task message queue:

- The corresponding information is read from the chained list;
- The corresponding element is removed from the chained list and freed;

- If the timer is marked as cancelled the corresponding message is discarded;
- If the timer is valid it is processed.

8.8.1.5 Automaton structure

The implementation of SDL automatons is made of an initialization part including the SDL start transition followed by an infinite loop that is the core of the task. The infinite loop usually does roughly the following:

```
for (;;)
  {
  /* Continuous signals treatment - see “Continuous signal” on page 158 */

  if ( messages to read in the save queue)
    retrieve the next message in the save queue
  else
    read the message queue
  if (message  is a timer)
    if (timer is cancelled)
      discard it (message = NULL)

  /* Message input treatment - see “Message input” on page 156 */

  free message memory space
  if (SDL state has changed)
    reorganize save queue
  }
```

8.8.1.6 Critical sections

As described above the generated processes manipulate global variables to handle process and semaphore information. Since all processes are manipulating the same chained lists an RTDS system critical section has been introduced. It uses a binary semaphore so that only one process at a time manipulates one of the chained list. It is made of 6 parts:

- `RTDS_CRITICAL_SECTION_DECL`
  Declaration of the system semaphore id global variable: `RTDS_globalSystemSemId.`
- `RTDS_CRITICAL_SECTION_PROTO`
  Prototype of the declaration of the system semaphore id global variable.
- `RTDS_CRITICAL_SECTION_INIT`
  Creation and initialisation of the system semaphore made by the start up process.
- `RTDS_CRITICAL_SECTION_START`
  A system critical section starts, a blocking attempt to take the semaphore is done.
- `RTDS_CRITICAL_SECTION_STOP`
  A system critical section ends, the semaphore is given back.
- `RTDS_CRITICAL_SECTION_POSTAMBLE`
  Deletes the system semaphore. Used when the system terminates.

8.8.1.7 Startup synchronization

An SDL process can send a message to another process in the start transition. This will fail if the receiver instance is not yet created. Therefore, at startup time, a synchronization semaphore is created and all created processes are put on wait. When initialisation is done, the semaphore is flushed and all tasks start running.

This handling is based on 6 macros:

- `RTDS_START_SYNCHRO_DECL`
  Declaration of the synchronization semaphore id global variable : `RTDS_globalStartSynchro`.
- `RTDS_START_SYNCHRO_PROTO`
  Prototype of the declaration of the synchronization semaphore id global variable.
- `RTDS_START_SYNCHRO_INIT`
  Creation and initialisation of the system semaphore made by the start up process.
- `RTDS_START_SYNCHRO_WAIT`
  A newly created task waits for the semaphore to be freed.
- `RTDS_START_SYNCHRO_GO`
  The startup task flushes the semaphore.
- `RTDS_START_SYNCHRO_POSTAMBLE`
  Deletes the synchronization semaphore. Used when the system terminates.

The same mechanism is used when tracing execution. The same semaphore is used to hold the newly created process until the process create trace is updated. This is to handle the case where the created task priority is higher that its parent. For this purpose 2 extra macros have been created:

- `RTDS_START_SYNCHRO_HOLD`
  The semaphore is taken so that the newly created task is put on hold.
- `RTDS_START_SYNCHRO_UNHOLD`
  The semaphore is released so that the newly created task can run.

Note this is only done when the RTDS debugger support is activated in the generation options. For a “plain” code generation, it is not needed.

8.8.1.8 Environment task

When debugging, the environment surrounding the SDL-RT system is represented by an RTOS task: `RTDS_Env`. Messages going out or coming in the system are actually exchanged with `RTDS_Env`. Since `RTDS_Env` receives the messages coming from the system, it might be wise to have it to free the parameters of the messages. Defining `RTDS_ENV_FREE_PARAMETER` in the compiler options will free parameter pointers in automatically generated `RTDS_Env`.

`RTDS_Env` can also be a user-defined process. In that case freeing memory is up to the user in the design.

8.8.1.9 Error handling

If an error is detected in the *Real Time Developer Studio* kernel the `RTDS_SYSTEM_ERROR` C macro is called with an error code. The error codes are listed and explained in the `RTDS_Error.h` file.

The `RTDS_SYSTEM_ERROR` C macro needs to be defined by the user to react properly to system errors. By default it is defined in `RTDS_MACRO.h`.

8.8.2 Creating a new RTOS integration for RTDS

Creating a new integration basically consists in creating all required files for RTDS in a directory so that it can be used as the code templates directory in a project's generation options. Required files are described in "Mandatory files" on page 140.

The best way to do so is to actually take an existing integration and adapt it to the new RTOS. A good candidate for this is the VxWorks integration, since it almost supports all the features implemented in RTDS. The POSIX or Windows integrations are however quite bad candidates for this, because since these are not "real" RTOSes, a lot of features are actually missing from their API and have been implemented directly in the integration.

We strongly recommend to avoid modifying the existing integrations directly: Always create a separate directory for each new integration. This will avoid losing the original integration if you ever need it, and allows to keep a clean base for other customizations as well.

8.8.3 VxWorks integration

8.8.3.1 Version

All VxWorks function and macro definitions are available in the RTOS profile directory:

`$(RTDS_HOME)/share/ccg/vxworks/`

The VxWorks integration is based on version 5.4.2 of VxWorks (Tornado V2.0.2). It has been developed and tested with VxSim on Windows 2000 and Solaris 7.

8.8.3.2 Timers

SDL-RT timers are implemented using VxWorks watchdogs. When the timer goes off, the watchdog function `RTDS_WatchDogFunction` will receive as a parameter the address of the `RTDS_TimerState` structure related to the timer. This structure is in the `RTDS_TimerState` chained list of the task that has started the timer.

It is possible to set the system time with the `set time` shell command. It does it in two ways:

- set the `vxTicks` global variable value,
- call VxWorks `tickSet()` function.
  It is important to know that changing the system value will not make the timer go off before their dead line. This is due to the fact that the code generator uses watch dogs; when a watch dog is set it decreases its time out value without referring to the system time value.

8.8.3.3 Make process

VxWorks profile requires a specific make process to be able to generate an executable whatever the target CPU is. To do so the `WindMake.inc` makefile needs to be included at the end of the generated makefile. The make utility should be the one from Wind River and specify the target CPU. For example when using VxSim on PC host, the make command should be:

```
make CPU=SIMNT
```

A list of the available CPU can be found in:

`$(WIND_BASE)/host/include/cputypes.h`

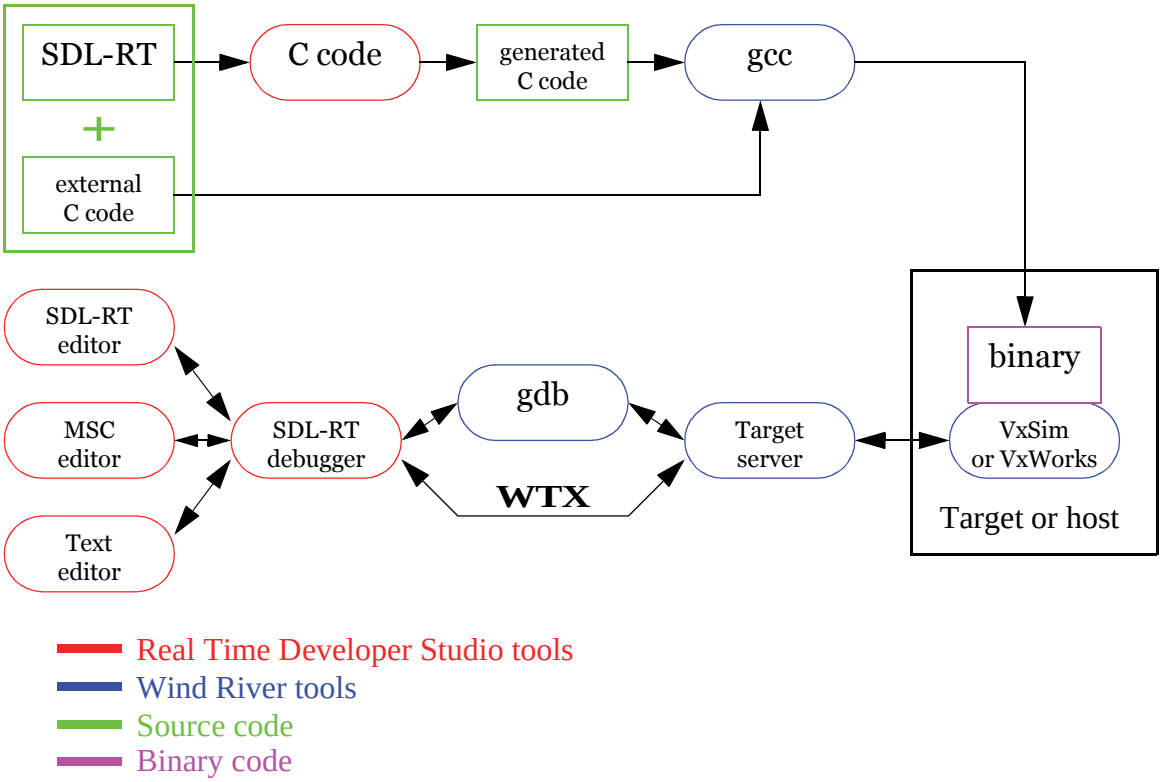
8.8.3.4 Tornado integration

8.8.3.4.1 Architecture

The *SDL-RT debugger* allows you to execute your SDL system and the associated C code. To do so *Real Time Developer Studio* generates the code necessary to execute the SDL processes and uses *Tornado's* compiler, debugger and RTOS (*VxWorks*) or RTOS simulator (*VxSim*).

Tornado architecture is done so that the tools on host interface with the *Target server*. The *Target server* is a unique interface whatever the target is including *VxSim*. Wind River has an API to connect to its *Target server*: WTX (Wind River Tool eXchange proto-

col). But not all information can be retrieved from WTX so a modified version of gdb is also running at the same time. This modified version is different depending on the target architecture. For example gdb to debug an application running on *VxSim* under *Windows* is `gdbsimnt`.

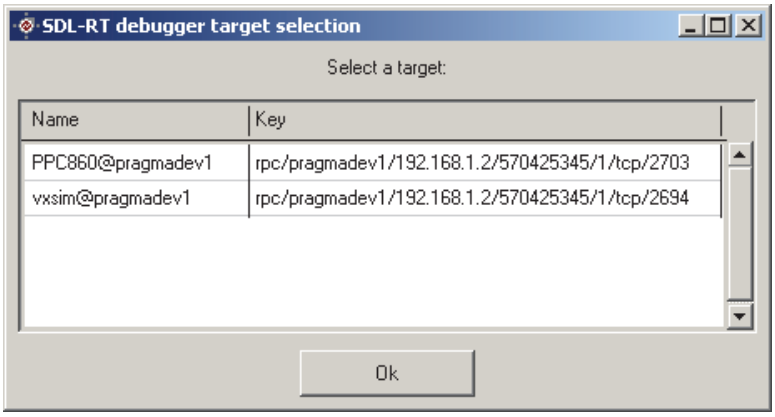


8.8.3.4.2 Launching process and target connection

When debugging with a Tornado debug environment selected, after compilation is done, RTDS will automatically:

- Check if `WIND_BASE` environment variable is properly defined
- Check if *Tornado Registery* is started. If not the SDL-RT debugger will try to start one
- If more than one *Target servers* are found

- A pop up window will list the existing target server and ask the user to select one



  - Connects to the selected *Target server*
  - Restarts the target
- If only one *Target server* is found
  - Connects to the existing *Target server*
  - Restarts the target
- If no target server can be found
  - Starts *VxSim*
  - Starts a *Target server*
  - Connects to the *Target server*
- Download the executable
- Starts *Wind River gdb* and *WTX* protocol
- Starts the executable with a breakpoint on it so it will stop at the entry point of the executable

8.8.3.4.3 Configuring VxWorks image to debug on target

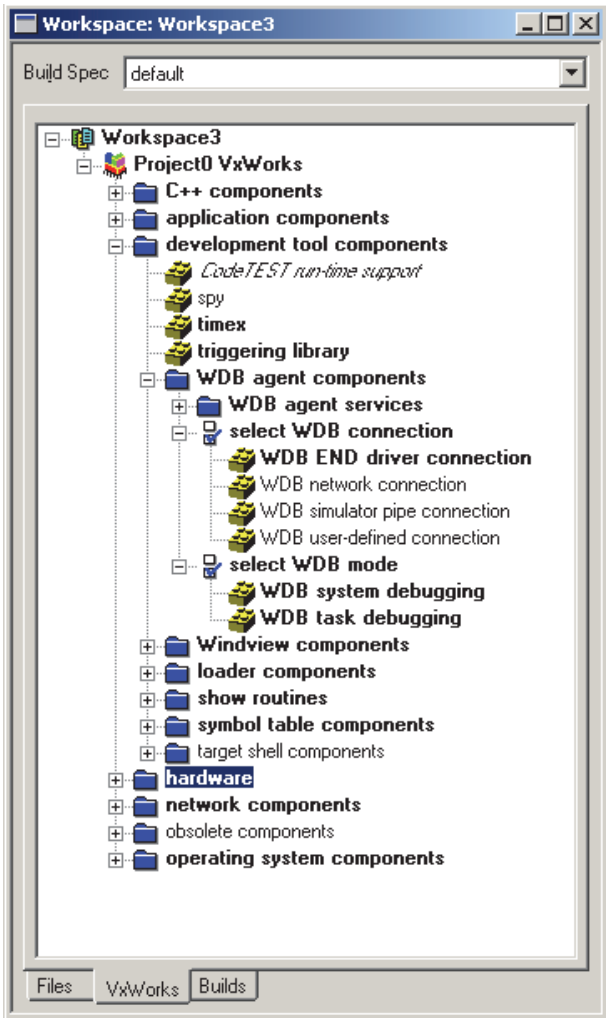
Wind Debug agent

The Tornado integration uses *System mode debugging* facility. To be able to debug on target with *RTDS SDL-RT debugger* the VxWorks image must be configured to support that mode.

- In the *development tools component / WDB agent components / Select WDB connection*:
  - *WDB END driver connection* should be selected
- In the *development tools component / WDB agent components / Select WDB mode*:
  - *WDB system debugging*

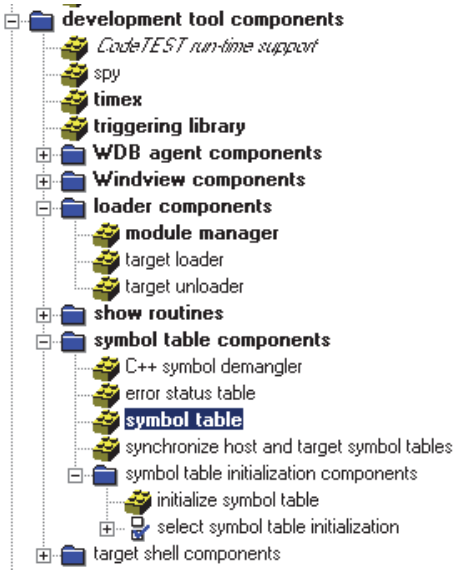
  and
  - *WDB task debugging*

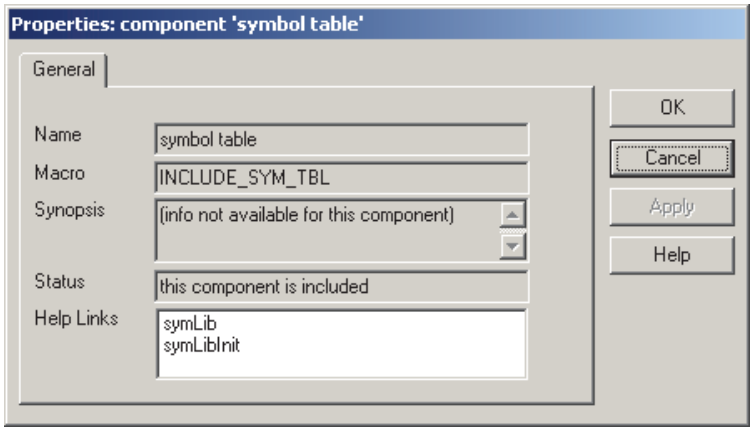
should both be selected.



Symbol table

The symbol table should be built-in the executable:





8.8.4 Posix integration

All Posix function and macro definitions are available in the RTOS profile directory:

`$(RTDS_HOME)/share/ccg/posix/`

Posix integration has been tested with the following compilers and debuggers:

- Solaris 7
  - gcc v2.95
  - gdb v5.2
- Linux Mandrake 8.0 and Red Hat 7.2
  - gcc v2.96
  - gdb v5.2

8.8.4.1 SDL-RT task

SDL-RT tasks are mapped to Posix threads.

8.8.4.2 Message queue

SDL-RT messages are not implemented using the Posix `mqueue` library. The message queue is defined as a chained list protected by 2 semaphores; the first one to avoid conflicting access and the second one to allow to wait for messages. In order to exchange messages between threads the following structures and functions have been defined

8.8.4.2.1 Internal structures:

- Queue Control Block: `RTDS_Profile_QCB`
  This structure contains:
  - pointer on the chained list semaphore (Mutex protecting chained list)
  - pointer on blocking semaphore (Binary allowing to wait for a message)
  - pointer on the first message
- Chained List Messages: `RTDS_Chained_Message`
  This structure contains:
  - pointer on message data (*message)
  - size of data
  - pointer on the next chained message

8.8.4.2.2 Functions:

- Message queue create: `RTDS_Profile_msgQCreate`
- Message queue delete: `RTDS_Profile_msgQDelete`
- Message queue send: `RTDS_Profile_msgQSend`
- Message queue read: `RTDS_Profile_msgQRead`

8.8.4.3 Semaphore

The SDL-RT mutex semaphore is mapped to `pthread` mutex semaphores. Binary and counting semaphores are mapped to standard Posix semaphores (`semaphore.h`).

8.8.4.4 Timer

RTDS Posix profile implements its own timer mechanism. Each SDL-RT timer is a `nanosleep` function call in a separate thread. Timer's thread priority is 59 for Solaris and 99 for Linux.

- The timer information structure `RTDS_Profile_TimerInfo` contains:
  - timer delay
  - timer thread ID
  - pointer on the SDL-RT send message function
  - pointer on the SDL-RT send message function's parameter
- Timer function:
  Create timer: `RTDS_Profile_TimerCreate`
  Delete timer: `RTDS_Profile_TimerDelete`
  Start timer: `RTDS_Profile_TimerStart`
  Timer thread function: `RTDS_Profile_TimerThread`

8.8.5 Windows integration

RTDS *win32* integration files are in the directory below:

`$(RTDS_HOME)/share/ccg/windows/`

Windows integration has been tested with cygwin version 2.249.2.5 and the following compilers and debuggers:

- gcc 2.95.3-5
- gdb 2002-04-11-cvs (cygwin-special)

8.8.5.1 SDL-RT task

SDL-RT tasks are mapped to *Windows* threads.

8.8.5.2 Message queues

Windows API does not provide any messaging functionnality for threads. A simple message queue has been implemented in the RTDS profile files for communication among the generated threads.

A message queue is basically a chained list protected by 2 semaphores; the first one to avoid conflicting access to the list and the second one to wait for messages. An RTDS message queue is a pointer to a Queue Control Block structure `(RTDS_QCB)` that contains the following fields:

- pointer on the mutex semaphore protecting the chained list access,
- pointer on the blocking binary semaphore to wait for messages,
- pointer on the first message.

8.8.5.3 Semaphores

SDL-RT semaphores are mapped to *Windows* semaphores. An internal structure `RTDS_SemaphoreInfo` used to identify the semaphores contains the following fields:

- *Windows* handle on the semaphore,
- the semaphore type.

8.8.5.4 Timers

RTDS *Windows* profile implements its own timer mechanism. Each SDL-RT timer is a `Sleep` function call in a separate thread. Timer's thread priority is `THREAD_PRIORITY_TIME_CRITICAL.`

8.8.6 CMX RTX integration

8.8.6.1 Version

All CMX RTX function and macro definitions are available in the RTOS profile directory:

`$(RTDS_HOME)/share/ccg/cmx/`

The CMX RTX integration is based on version 5.3 of the RTOS. It has been developed and tested with Tasking cross compiler and debugger with the 167 instruction set simulator (C167CS) on Windows 2000 SP2 host.

8.8.6.2 File organization

8.8.6.2.1 RTDS_Cmx.c

This file contains some very specific functions needed to support CMX RTX. It defines:

- `TickTimerInit`
  Initialize timer with predefined values. This function can be removed by the user if timer functions are already existing on its system.
- `TickTimerInt`
  Timer 2 interrupt service routine. This function can be removed by the user if timer functions are allready existing on its system.
- `RTDS_globalSemaphoreId`
  `RTDS_globalSemaphoreId` is an array of short used to keep track of available semaphores in the system. If element n of the array is 1 the semaphore n is not available, if it is 0 it is available.
- `RTDS_initResourceId`
  This function initialize the `RTDS_globalSemaphoreId` array.
- `RTDS_semaphoreIdGet`
  This function retrieves an available semaphore id from the `RTDS_globalSemaphoreId` array.
- `RTDS_semaphoreIdFree`
  This function frees a semaphore id and make it available in the `RTDS_globalSemaphoreId` array.

8.8.6.2.2 CMX files

Since CMX is delivered as source code, the following files should be compiled and linked with the generated code in order to get a working executable (small memory model) :

- `cmx_init.c`
- `cxskv5s.src`
- `liba66s.lib`

The examples included in the distribution show how to include these files in the make process using an external makefile.

8.8.6.2.3 Defines

The generated C macros definition file (`RTDS_MACRO.h`) includes `Cxconfig.h` and `Cxfuncs.h` files. Please note a basic stack size verification is done so that the sum of all tasks stack is not superior to the total available stack.

8.8.6.2.4 main function

CMX expects the application to have a `main()` function. It has been introduced in `RTDS_Startup_begin.c` and replaces the `RTDS_Start()` function.

The main difference with a classical RTDS_Start() are:

- `K_OS_Init()` is called at the beginning of the main function (`RTDS_Startup_begin.c`) to initialize CMX,
- `TickTimerInit()` is called to initialize system tick count interrupt function defined in `RTDS_Cmx.c`,
- `K_OS_Start()` is called at the very end of the main function to start the RTOS.

8.8.6.3 Task slot and mailbox numbering

CMX RTOS handles static mailboxes. Since no dynamic creation is possible, the task slot id given back by CMX RTX is also used as the mailbox id. That means the generated code considers the corresponding mailbox id is available. So if the application running on CMX handles mailboxes from external C code, the external modules should use higher mailbox id than the maximum number of SDL-RT tasks.

8.8.6.4 Task context

In the generated code, when a task is started, it needs to know about its context. That is the address of its `RTDS_GlobalProcessInfo` structure in the `RTDS_globalProcessInfo` chained list.

Since CMX does not allow to give parameters when creating a task, the following is done at the beginning of the task (`bricks/RTDS_Proc_begin.c`):

- the newly created task reads its own task slot id with `K_OS_Task_Slot_Get()` function,
- the task reads the `RTDS_globalProcessInfo` chained list until the `processId` field of the `RTDS_GlobalProcessInfo` structure correspond to its slot id. Then the task knows it is its own context structure.

8.8.6.5 Semaphores

8.8.6.5.1 Types

CMX supports only counting semaphores with a FIFO queueing mechanism. SDL-RT mutex and binary semaphores have been mapped to counting semaphore with an initial value of:

- 0 when the semaphore is created not available
- 1 when the semaphore is created available

The SDL-RT debugger Semaphore list window will display the RTOS type of semaphore. That means that whatever type of semaphore you have created in SDL-RT, they will be displayed as counting semaphores with FIFO queue type in the Semaphore list window. That ensures the user does not have a distorted view of the generated code.

8.8.6.5.2 Identification

Semaphore are handled statically in CMX; there is no dynamic semaphore creation. Since the generated code is based on a dynamic creation mechanism, functions have

been implemented to handle dynamically allocated semaphore ids in `$(RTDS_HOME)/share/ccg/cmxRTDS_Cmx.c` as described in paragraph “RTDS_Cmx.c” on page 214.

8.8.6.6 Timers

CMX cyclic timers are used to implement SDL-RT timers. To understand this integration it is important to clearly separate the different identifiers manipulated:

- SDL-RT timer id
  That is the SDL-RT message number. It is defined in RTDS_gen.h file.
- CMX cyclic timer id
  This the global timer id for the whole application. The application can handle up to `C_MAX_CYCLIC_TIMERS` separate cyclic timers.
- CMX event number related to the timer
  When the timer goes off, it is identified by the event number received by the task. There can be 16 different events per task. But event number 1 is reserved for normal messages, so event number 2 to 15 can be used for timers. That means an SDL-RT system can only handle 15 simultaneous timers.

To handle timers in this integration, an global array of void pointers has been added:

```
void *RTDS_globalTimerEvent[C_MAX_CYCLIC_TIMERS];
```

That array is used to relate the CMX cyclic timer id to the corresponding CMX event when the timer goes off and to the SDL-RT timer id. The main steps are:

- get a CMX timer id
  A free position in the array is searched. The resulting position is used as the cyclic timer id. The address of the corresponding `RTDS_TimerState` is stored in the array at this position.
- create the timer
  A timer is created using the previously defined id. The id is also used to define the corresponding event left shifting of `0x0002`. For example, that means if id is 2, the event id is `0x0002<<2=8`. If id is 0, the event id is `0x0002<<0=2`.
- start the timer
  The CMX cyclic timer is started.

When an event is received, if not a normal message, the main steps are:

- retrieve CMX timer id
  The event id is used to find back the CMX timer id. The number of right shift to get to `0x0002` is CMX timer id.
- get the timer information
  Since the CMX timer is the position in the array the necessary information is at hand to inform the SDL-RT system.
- Send a timer message to the SDL-RT system

8.8.6.7 Event handling

Because cyclic timers only manipulate events, tasks are waiting for events, not directly for messages on their respective mailboxes. When a task is created, the parent task uses the `K_Mbox_Event_Set()` function so that the task receives an event when a message

is in the mailbox. The event id used for messages is `0x0001` (`RTDS_MBOX_EVENT` in `RTDS_OS.h`).

8.8.6.8 Examples

8.8.6.8.1 Installation

In the examples provided in RTDS distribution it is assumed:

- CMX has been installed in `c:\cmx` directory
- Tasking environment has been installed in `c:\c166` directory

If CMX or Tasking have been installed in other directories the Tasking generation profile and the `extra.mak` makefile should be modified to fit the environment.

8.8.6.8.2 Utilities

The Tasking *ieee166* utility is used to create a downloadable executable in the `after compil command` field of the Generation profile. If the name of the SDL-RT system changes, the command needs to be modified.

8.8.6.8.3 Restrictions

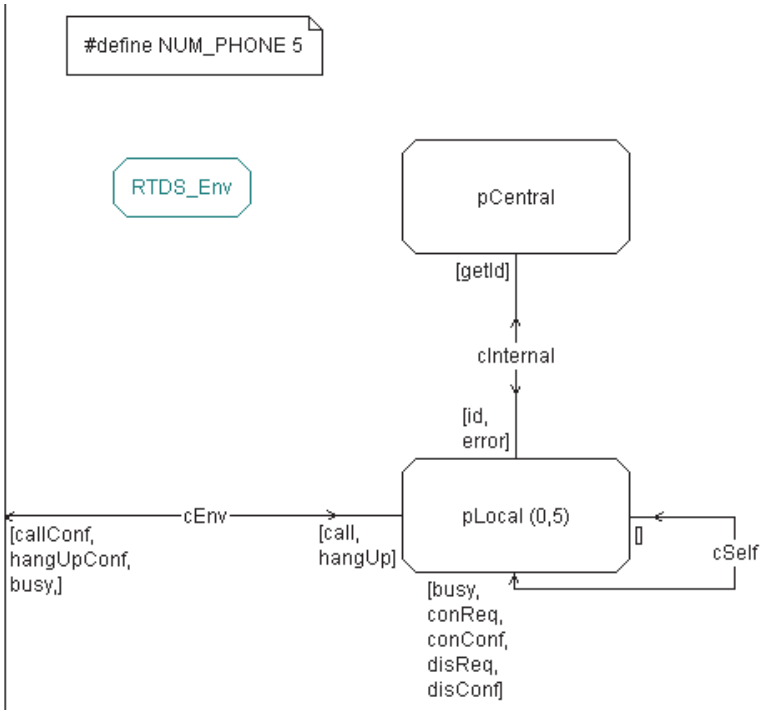
Some of the examples can not run as delivered because it is not possible to send messages to the system from the debugger.

8.8.6.8.4 Tutorial

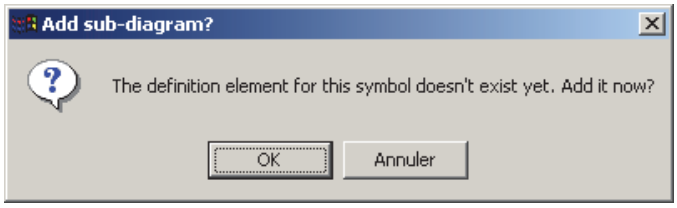
Since it is not possible to send messages to the system, a test scenario needs to be pre-defined inside the system. To do so it is possible to create a process that will replace the environment. This process needs to be named RTDS_Env. Such a process already exist in the example but is not defined in the SDL-RT system. To add it to replace the environment

- open the system definition,

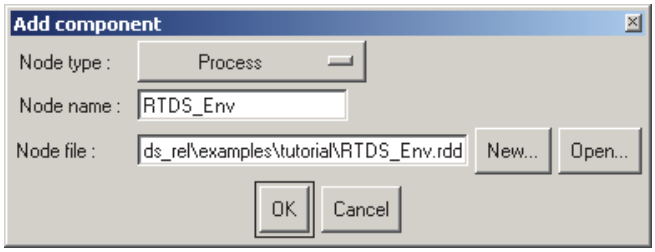
- add a process named RTDS_Env,



- open the process definition, a window will ask to add the definition element:

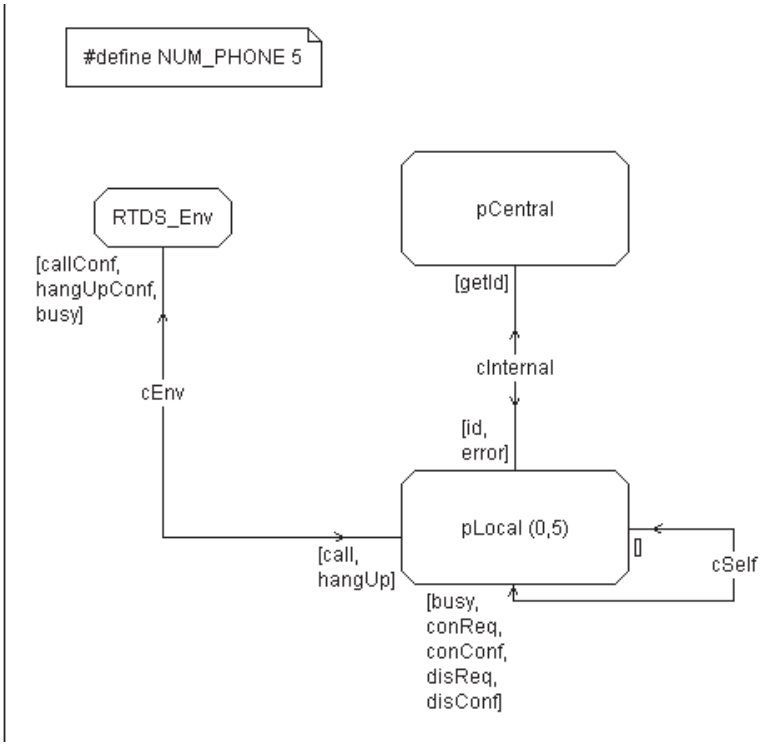


- Click *OK*, the *Add component* window will open:



- Click on *Open* and select the existing file:
  `$(RTDS_HOME)/examples/tutorial/RTDS_Env.rdd`
  The element will be added in the *Project manager*.

- The new process should now be connected to the existing architecture the following way:



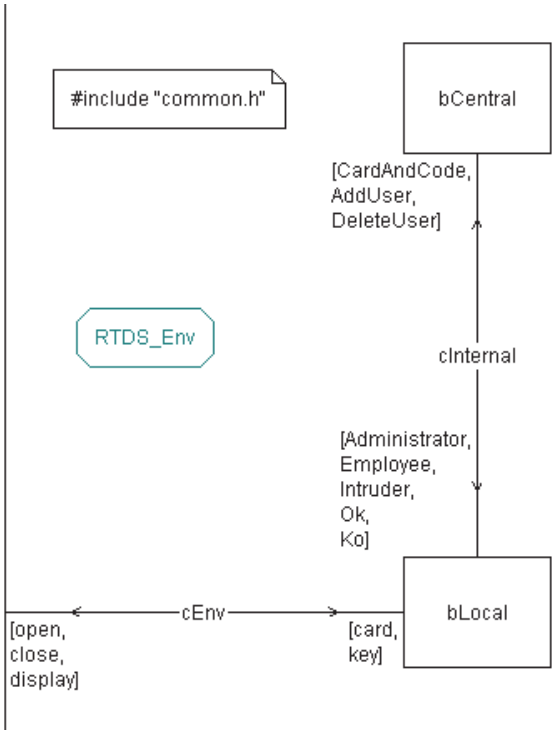
When debugging the system the `RTDS_Env` process will run a basic scenario automatically.

8.8.6.8.5 Access Control System

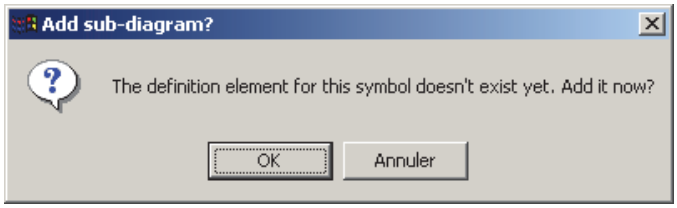
Since it is not possible to send messages to the system, a test scenario needs to be predefined inside the system. To do so it is possible to create a process that will replace the environment. This process needs to be named RTDS_Env. Such a process already exist in the example but is not defined in the SDL-RT system. To add it to replace the environment

- open the system definition,

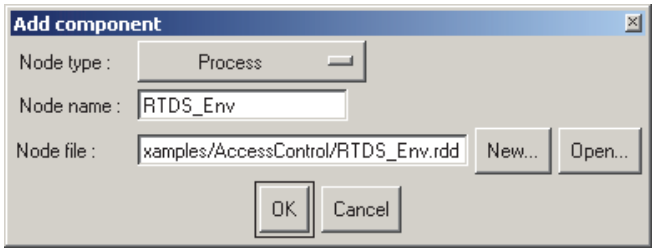
- add a process named RTDS_Env,



- open the process definition, a window will ask to add the definition element:

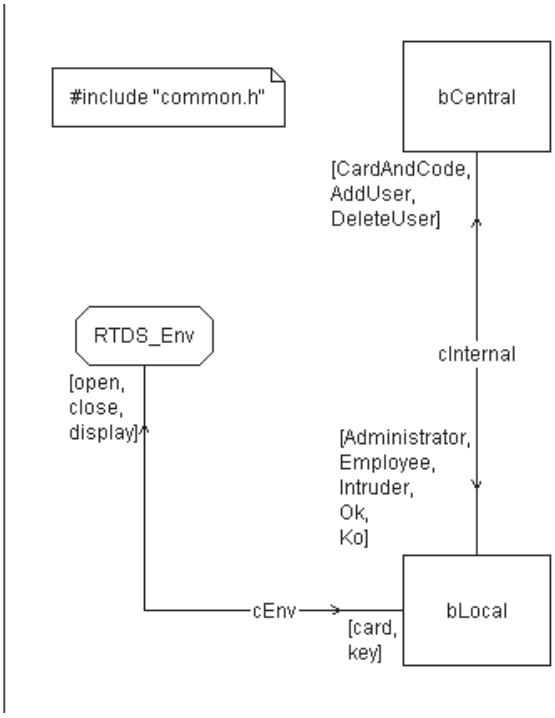


- Click *OK*, the *Add component* window will open:



- Click on *Open* and select the existing file:
  `$(RTDS_HOME)/examples/AccessControl/RTDS_Env.rdd`
  The element will be added in the *Project manager*.

- The new process should now be connected to the existing architecture the following way:



When debugging the system the `RTDS_Env` process will run an endless scenario automatically:

- Enter supervisor card
- Enter supervisor code
- Register new user card
- Register new user code
- Wait 5 seconds
- Use new card and code
- Go back to wait 5 seconds and so on...

8.8.7 uITRON 3.0 integration

All profile functions and macro definitions are available in the RTOS profile directory:

`$(RTDS_HOME)/share/ccg/uitron3_0/`

uITRON integration has been tested with the following compilers and debuggers:

- Solaris 7
  - arm-elf-gcc PragmaDev build
  - XRAY v4.4dgd and the Armulator
- Windows 2000
  - arm-elf-gcc PragmaDev build
  - XRAY v4.6ap and the Armulator

8.8.7.1 ID affectation

The maximum number of task, semaphore, message queue and timer are statically defined therefore the uITRON profile implement functions to get and release IDs.

8.8.7.2 SDL-RT task

SDL-RT tasks are mapped to uITRON task and handled using `cre_tsk, sta_tsk` and `exd_tsk`

8.8.7.3 Message queue

SDL-RT messages are mapped to uITRON mailbox.

8.8.7.4 Semaphore

All semaphore are based on uITRON counting semaphore and handled using `cre_sem`, `sig_sem`, `twai_sem`, `preq_sem` functions.

8.8.7.5 Timer

SDL-RT timers use uiTRON alarm mechanism. When an eCos alarm goes off, a pre-defined function is called and a specific procedure is used to read parameters. According to the uITRON specification the alarm should be defined using `DALM` structure but instead of using uITRON `almatr` field eCos use the RTOS specific `extinf` field to send parameters.

8.8.7.6 Using uITRON from eCos

Building tools

- All required information can be found in the following web pages:
  - if using Windows: http://sources.redhat.com/ecos/install-windows.html
  - if using Linux: http://sources.redhat.com/ecos/install-linux.html
- Building compiler: depending on the microprocessor what will be used the compiler should be re-compiled according to the building procedure available on the eCos website
- Building eCos library: this build should be made without the -g option

Configuration to use eCos :

- Install "uITRON compatibility" package in order to use uITRON interface

The eCos uITRON configuration can be made using the eCos configuration tool or changing the header file.

Task configuration:

- declare maximum number of task (CYGNUM_UITRON_TASK)
- set CYGNUM_UITRON_TASK to the start task ID
- declare task prototype for each task

for example if an SDL-RT system is using 4 tasks:

```
CYGDAT_UITRON_TASK_EXTERN = \
extern "C" void RTDS_Start( unsigned int ); \
 static char stack1[ CYGNUM_UITRON_STACK_SIZE ], \
 stack2[ CYGNUM_UITRON_STACK_SIZE ], \
 stack3[ CYGNUM_UITRON_STACK_SIZE ], \
 stack9[ CYGNUM_UITRON_STACK_SIZE ],\
 stack10[ CYGNUM_UITRON_STACK_SIZE ];
and
CYGDAT_UITRON_TASK_INITIALIZER = \
CYG_UIT_TASK( "t1", 1, RTDS_Start, &stack1, CYGNUM_UITRON_STACK_SIZE ), \
CYG_UIT_TASK_NOEXS( "t2", &stack2, CYGNUM_UITRON_STACK_SIZE ), \
CYG_UIT_TASK_NOEXS( "t3", &stack3, CYGNUM_UITRON_STACK_SIZE ), \
CYG_UIT_TASK_NOEXS( "t4", &stack4, CYGNUM_UITRON_STACK_SIZE ), \
CYG_UIT_TASK_NOEXS( "t5", &stack5, CYGNUM_UITRON_STACK_SIZE ), \
```

Semaphore configuration:

- declare maximum number of semaphores (CYGNUM_UITRON_SEMAPHORE)

Timer configuration:

- declare maximum number of timers (CYGNUM_UITRON_TIMER)

Message queue configuration (CYGNUM_UITRON_MBOXES)

- declare maximum number of message queues

8.8.8 uITRON 4.0 integration

All profileì functions and macro definitions are available in the RTOS profile directory:

`$(RTDS_HOME)/share/ccg/uitron4_0/`

uITRON integration has been tested with the following compilers and debuggers:

- Windows 2000
  - MinGW gdb V6.5.50

8.8.8.1 ID affectation

The maximum number of task, semaphore, message queue and timer are statically defined but ID affectation is dynamic.

8.8.8.2 SDL-RT task

SDL-RT tasks are mapped to uITRON task and handled using `acre_tsk` and `ext_tsk`

8.8.8.3 Message queue

SDL-RT messages queue are mapped to uITRON mailbox and handled using `acre_mbx, isnd_mbx, snd_mbx, rcv_mbx, del_mbx` functions.

8.8.8.4 Semaphore

Binary and counting semaphores are based on uITRON counting semaphore and handled using `acre_sem, sig_sem, del_sem, ref_sem, wai_sem, twai_sem` functions. But mutexes are not supported.

8.8.8.5 Timer

SDL-RT timers use uiTRON alarm mechanism. They are handled using `acre_alm, sta_alm, del_alm` functions.

8.8.8.6 Using uITRON 4 from NUCLEUS

The uITRON 4.0 integration has been tested with NUCLEUS uITRON 4 API with the following utilities:

- C compiler :
  `mingw32-gcc`
- Compiler options :
  `-mno-cygwin -I"$(SIMTEST_ROOT)\..\nucleus"`
  `-I"${RTDS_HOME}\share\ccg\uitron4_0\Miplus"`
- Linker :
  `-L"$(SIMTEST_ROOT)\..\nucleus\lib\mingw" -lsim -lplus -lvt -lwinmm`
  `-luiplus -mno-cygwin`
- Make utility :
  `mingw32-make`
- Make options
  `mingw32-make` utility should be run with `all` as a target and with a path to the Makefile.mk file:
  `mingw32-make -f "%RTDS_HOME%/share/ccg/uitron4_0/MIPlus/Makefile.mk"`

8.8.9 Nucleus integration

8.8.9.1 Version

All Nucleus function and macro definitions are available in the RTOS profile directory: `$(RTDS_HOME)/share/ccg/nucleus/`

The Nucleus integration is based on version 1.3.2 of the EDGE Development Environment. It has been developed and tested with SimTest V1.3.2 on Windows 2000 and MinGW gdb V6.5.50.

8.8.9.2 Task context

In the generated code, when a task is started, it needs to know about its context. That is the address of its `RTDS_GlobalProcessInfo` structure in the `RTDS_globalProcessInfo` chained list.

Nucleus task creation allows to give parameters when creating a task, the following is done at the beginning of the task (`bricks/RTDS_Proc_begin.c`):

8.8.9.3 Semaphores

Nucleus only supports counting semaphores. Thus SDT-RT mutex semaphores are not supported . However SDL-RT binary semaphores have been mapped to counting semaphore with an initial value of:

- 0 when the semaphore is created not available
- 1 when the semaphore is created available

8.8.9.4 SDL-RT system start

Nucleus application entry point is `Application_Initialize` function which is defined in the `RTDS_OS.c` file. `Application_Initialize` creates the `RTDS_Start` task that will creates and starts all SDL-RT processes present at startup.

8.8.9.5 Message queue

Message queue in Nucleus is a memory area identified by an id. All created processes have a message queue in RTDS. The creation of the message queue is located in the `RTDS_ProcessCreate` function located in the `RTDS_OS.c` file.

8.8.9.6 Memory management

The memory management in Nucleus relies on the functions `NU_Allocate_Memory` et `NU_Deallocate_Memory`. These fonctions allocate and deallocate memory from a memory pool created by the SDL-RT system. The creation of this memory pool is located in the Application_Initialize function in the `RTDS_OS.c` file. The size of the memory pool is defined by the `SYSTEM_MEMORY_SIZE` macro defined in the `RTDS_OS.h` file.

8.8.9.7 Testing environment

The Nucleus integration has been tested with the following utilities:

- C compiler :
  `mingw32-gcc`
- Compiler options :

```
-mno-cygwin -I"$(SIMTEST_ROOT)\..\nucleus" -DNU_SIMULATION
```

- Linker :
  ```
  -L"$(SIMTEST_ROOT)\..\nucleus\lib\mingw"
  -lsim -lplus -lvt -lwinmm -mno-cygwin
  ```
- Make utility :
  `mingw32-make`
- Make options
  `mingw32-make` utility should be run with `all` as a target and with a path to the Makefile.mk file:
  ```
  mingw32-make -f "%RTDS_HOME%/share/ccg/nucleus/make/Make-
  file.mk"
  ```

8.8.10 OSE Delta integration

8.8.10.1 OSE 4.5.1

8.8.10.1.1 Version

All OSE function and macro definitions are available in the RTOS profile directory:
`$(RTDS_HOME)/share/ccg/ose/`

The OSE integration is based on version 4.5.1. It has been developed and tested with OSE soft kernel on Solaris and gdb V5.2.

8.8.10.1.2 Socket support

Socket communication with the target is supported. It allows for example to debug a soft kernel application on host easily and to display an MSC Trace of the target execution.

When debugging with socket support a RTDS_SocketProcess task is created to handle commands coming from the SDL-RT debugger going to the target.

8.8.10.1.3 printf support

Please note it is not possible to use dbgprintf with gdb because the application print out will get mixed up with gdb control information.

8.8.10.1.4 Timers

SDL-RT timers are based on OSE Time-Out Server (TOSV). As OSE kernel is pretty close to SDL concept of timer, when the timer goes off it is already translated as a signal (or message) in the process queue. Still a timer list is kept for each process in case cancellation of the OSE timer does not succeed. That would mean the timer is already in the queue; RTDS implementation will take care of cancelling it anyway.

8.8.10.1.5 Make process

The OSE build process is based on `dmake` utility and `makefile.mk` and `userconf.mk` makefiles. When using OSE code generation, RTDS generates `pragmadev.mk` makefile in the code generation directory. The `makefile.mk` and `userconf.mk` should be put somewhere else because they are not generated file and `makefile.mk` must include the generated `pragmadev.mk` file. For example:

```
include .$/pragmadev.mk
```

In the generation profile:

- The compiler should be defined as:
  `$(CC)`
- The compiler options should be set to:
  `$(CFLAGS) $(DEFINES) $(STDINCLUDE) $(INCLUDE) $(CCOUT)`
  plus any extra user options.
- External specific OSE makefile should be included:
  `${RTDS_HOME}\share\ccg\ose\make\OseMake.inc`
- dmake utility should be run with `RTDS_ALL` as a target and with a path to the makefile.mk file:
  `dmake -f ../makefile.mk RTDS_ALL RTDS_HOME=%RTDS_HOME%`

8.8.10.1.6 Error handling

OSE kernel handles errors implicitly. Therefore there is no need to check return values of each system call in the integration files. When an RTDS integration error occurs the OSE `error2` primitive is called so that all errors are raised the same way. Explanations for the system error code are in `RTDS_Error.h`.

8.8.10.1.7 Task context

In the generated code, when a task is started, it needs to know about its context. That is the address of its `RTDS_GlobalProcessInfo` structure in the `RTDS_globalProcessInfo` chained list.

Since OSE does not allow to give parameters when creating a task, the following is done at the beginning of the task (`bricks/RTDS_Proc_begin.c`):

- the newly created task reads its own task slot id with `current_process()` function,
- the task reads the `RTDS_globalProcessInfo` chained list until the `processId` field of the `RTDS_GlobalProcessInfo` structure correspond to its own process id. Then the task knows it is its context structure.

8.8.10.1.8 Semaphores

OSE only supports counting semaphores. SDL-RT mutex and binary semaphores have been mapped to counting semaphore with an initial value of:

- 0 when the semaphore is created not available
- 1 when the semaphore is created available

8.8.10.1.9 SDL-RT system start

SDL-RT system entry point is RTDS_Main function defined in generated RTDS_Start.c. The OseMake.inc defines RTDS_Main as the only OSE static process. RTDS_Main calls RTDS_Start that will create all SDL-RT static processes (processes present at startup). Once they are all created, RTDS_Start will start them to guarantee synchronization. Then RTDS_Start stops because OSE does not allow OSE static processes to die.

8.8.10.1.10 Signal queue

Signal queues in OSE are implicit. The signal queue id in RTDS process context will be the same as the process id.

8.8.10.1.11 Signal output

When using SDL-RT `OUTPUT TO_NAME` the current OSE integration uses its internal information to find the receiver. It would be possible to use the OSE hunt system call to find a receiver.

8.8.10.2 OSE 5.2

8.8.10.2.1 Version

All OSE function and macro definitions are available in the RTOS profile directory: `$(RTDS_HOME)/share/ccg/ose52/`

The OSE integration is based on version 5.2. It has been developed and tested with OSE soft kernel on Windows 2000 and gdb V6.3. There are few differences between OSE 5.2 and OSE 4.5.1 integration, the following chapters will discuss these differences.

8.8.10.2.2 Socket support

Socket communication with the target is supported. It allows for example to debug a soft kernel application on host easily and to display an MSC Trace of the target execution.

When debugging with socket support a RTDS_SocketProcess task is created to handle commands coming from the SDL-RT debugger going to the target. OSE 5.2 and OSE 4.5.1 integrations use different primitives for socket communication: `inet_send()` replaces `write()` function and `recv()` replace `read()` function. In OSE 4.5.1 the socket descriptor is a static variable defined in `RTDS_Trace.c` file and it is shared by all processes. This is not possible with OSE 5.2; thus to share a socket descriptor in this integration, `efs_clone()` function is called in `RTDS_ProcessCreate` function. This function copies all the file descriptors for a target process.

8.8.10.2.3 Make process

The OSE build process is based on `make` utility and `main.mk, RTDS.mk` and `Makefile` makefiles. When using OSE code generation, RTDS generates `RTDS.mk` makefile in the code generation directory. This file contains all the rules to build the object files and the final library which is linked with the OSE soft kernel. After having generated `RTDS.mk` and before calling the build command, the code generation directory is exported to `CURDIR` environment variable with the following command : `sh -c "CURDIR=`pwd`"`.

Then, the `RTDS_ALL` rule is called on the `main.mk` makefile. This rule generates

- the `osemain.con` file which defines RTDS_Main as the only OSE static process,
- the `Makefile` file which contains the dependancy to the `RTDS.mk` makefile,
- a makefile that is named like the code generation directory with `.mk` extension; in which the `CURDIR` variable is used to override `OSEMAICON` and `LIBS` variables.

The SDL-RT system is built as a library via the command `make all FLAVOR=debug XMOD=$(CURDIR) USE_CPLUSPLUS=$(USE_CPLUSPLUS)` after entering in `<OSE_ROOT>/refsys/rtose/sfk-win32` directory (for Win32 integration). When using a C/C++ compiler the USE_CPLUSPLUS must be set to `yes` in OSE52 profile. Then, after linking the library to the OSE soft-kernel, the final executable can then be debugged with SDL-RT debugger.

On Linux and Solaris2 integration, environment variables `OSE_ROOT` and `REFSYS_ROOT` must be set in file `.bashrc`. Moreover, to avoid potential conflict on Solaris2 and Linux, the ports indicated in the file `rtose5.conf` can be changed. For instance this line `surfer=port:80` can be changed to `surfer=port:1205`. On Solaris2, ethernet device is named `hme0`, so all references of this device in the file `rtose5.conf` have to be replaced by `hme0`.

8.8.11 OSE Epsilon integration

8.8.11.1 Version

All OSE function and macro definitions are available in the RTOS profile directory:

`$(RTDS_HOME)/share/ccg/oseepsilon/`

The OSE integration is based on version 3.0 for C166. It has been developed and tested with Tasking Cross View Pro C166 instruction set simulator on Windows 2000.

8.8.11.2 Timers

SDL-RT timers are based on OSE Time-Out Server (TOSV). As OSE kernel is pretty close to SDL concept of timer, when the timer goes off it is already translated as a signal (or message) in the process queue. OSE Epsilon does not allow to pass parameters to the timer so a timer list is kept for each process and when the timer goes off the list is used to fill in the corresponding RTDS message header. Cancelling a timer that is already in a message queue is not possible in this integration unlike what is done with OSE Delta.

8.8.11.3 Dynamic process creation

OSE Epsilon does not support dynamic process creation. All processes are created at startup by OSE.

8.8.11.4 Make process

The make process is a bit tricky because the OSE Epsilon kernel is static and needs to be re-configured every time a build is done. To do so, it is recommanded to use an *awk* script to parse the `RTDS_gen.inf` generated file in order to extract the necessary information and generate an `os166.con` file (in the case of a 166 target). An example script is provided in `$(RTDS_HOME)/share/ccg/oseepsilon/make` directory. The script requires *Cygwin* installation to be run and foes through the followin steps:

- It first copies a pre-defined `os166.con` preamble that contains RTDS_Start task:
  ```
  %PRI_PROC RTDS_Start,C,1024,40,1
  ```
- adds the list of static processes with the `.con` syntax out of the `RTDS_gen.inf`,
- copies the `os166.con` postamble at the end of the file.

This file is used by `conf166.exe` called from `TaskingOseMake.inc` that should be inlcuded in the generation profile.

In the example generation profile based on Tasking, the Cygwin make is used instead of Tasking make in order to be as generic as possible:

- The compiler should be defined as:
  `cc166`
- The compiler options should be set to:
  `$(C_FLAGS)`
  plus any extra user options.
- The linker options should be set to:
  `$(D_FLAGS)`

plus any extra user options.
- External specific OSE makefile should be included:
  `${RTDS_HOME}\share\ccg\oseepsilon\make\TaskingOseMake.inc`
- make utility should be run with `all` as a target:
  `make all`

8.8.11.5 Error handling

OSE kernel handles errors implicitly. Therefore there is no need to check return values of each system call in the integration files. When an RTDS integration error occurs the `RTDS_ErrorHandler` function is called and the OSE `error` primitive is called so that all errors are raised the same way. Explanations for the system error code are in `RTDS_Error.h`. For OSE errors the RTDS error code is combined with the OSE error code. For example:

```
Error. System error on target no: 0x3004, in task: 0x0, at: 0x6f
ticks
```

with:

```
#define RTDS_ERROR_OSE_ERROR_HANDLER 0x3000 /*The first byte is the OSE error number (ERR_MSG[0]) */
```

Means OSE error number 0x04.

8.8.11.6 Task context

In the generated code, when a task is started, it needs to know about its context. That is the address of its `RTDS_GlobalProcessInfo` structure in the `RTDS_globalProcessInfo` chained list.

Since all processes are started statically by the system, each process creates its own context and stores it in the global chained list.

8.8.11.7 Semaphores

OSE only supports counting semaphores. SDL-RT mutex and binary semaphores have been mapped to counting semaphore with an initial value of:
- 0 when the semaphore is created not available
- 1 when the semaphore is created available

8.8.11.8 SDL-RT system start

SDL-RT system entry point is usually RTDS_Start that dynamically creates all semaphores and static processes. With OSE Epsilon, processes are all created by the kernel but not the semaphores. So RTDS_Start is defined as a static process with a very high priority to create the semaphores.

Also, since the processes generate their own context, a small delay has been introduced at process startup in order to give time to all processes to allocate their context.

8.8.11.9 Signal queue

Signal queues in OSE are implicit. The signal queue id in RTDS process context will be the same as the process id.

8.8.11.10 Priorities

OSE Epsilon highest priority level is 0 and lowest priority level is 31.

8.8.12 ThreadX integration

8.8.12.1 Version

All ThreadX function and macro definitions are available in the RTOS profile directory:

`$(RTDS_HOME)/share/ccg/threadx/`

The ThreadX integration is based on version G4.0a.4.0a for MIPS R3000 processors. It has been developed and tested with Green Hills Multi 2000 V 3.5 for MIPS with the included instruction set simulator on Windows 2000.

8.8.12.2 General considerations

The ThreadX philosophy is to let the user controls as much implementation details as possible in order to maximize performance and memory footprint. Therefore, in comparison with other RTOS integrations a lot of things need to be done manually. For example:

- when creating a thread
  - queue control block memory allocation
  - message queue memory allocation
  - thread control block memory allocation
  - thread stack memory allocation
- when deleting a thread
  - thread stack memory liberation
  - thread control block memory liberation
  - message queue memory liberation
  - queue control block memory liberation

8.8.12.3 Timers

SDL-RT timers use ThreadX timers:

- `tx_timer_create`
- `tx_timer_activate`
- `tx_timer_delete`
- `tx_timer_deactivate`

When the timer goes off the `RTDS_WatchDogFunction` is called to translate the timer in an SDL-RT message. Note the message header has already been allocated when the timer was started so the `RTDS_WatchDogFunction` is basically a `tx_queue_send` followed by a `tx_timer_deactivate`.

8.8.12.4 Make process

The ThreadX build process includes specific kernel and processor files. In RTDS distribution the ThreadX integration supports MIPS. The following file:

`$(RTDS_HOME)/share/ccg/threadx/make/ThreadX.inc`

is included in the make process to compile `reset.mip` and `tx_ill.mip` that are necessary to build a ThreadX application with Green Hills compiler.

8.8.12.5 Memory management

By default RTDS ThreadX integration uses a unique ThreadX memory byte pool -similar to a standard C heap-. The pool is created in `tx_application_define` function situated in `RTDS_startup_begin.c` (building brick for `RTDS_Start.c`). Since the `tx_byte_allocate` function does not return a pointer value but uses a pointer address to return the allocated value; the memory allocation macro calls a C function defined in `RTDS_Utilities.c`.

The ThreadX memory management functions used are:

- `tx_byte_pool_create`
- `tx_byte_allocate`
- `tx_byte_release`
- `tx_byte_pool_delete` (commented out in the `main` function after `tx_kernel_enter`)

8.8.12.6 Synchronization

The synchronization semaphore `RTDS_START_SYNCHRO` has been removed from the ThreadX integration because:

- during dynamic creation threads are created suspended,
- and because at startup we know `tx_application_define` function will be fully executed before all tasks start.

So synchronization of dynamic process creation is done by the process creator (PARENT).

8.8.12.7 Thread management

8.8.12.7.1 Priority

Threads default priority is 15 and `preempt_threshold` has the same value as priority.

8.8.12.7.2 Thread deletion

A `pStack` and a `pQueue` fields have been added to `RTDS_GlobalProcessInfo` struct in order to free the stack and the queue when the thread dies. So when a thread dies it frees its stack and queue and terminates itself; but since a thread can not delete itself, the thread deletion is done in the next thread creation. Therefore threads about to be deleted are still listed in the *Process information* window of the SDL-RT debugger with an `N/A` state.

8.8.12.7.3 Thread creation

In the thread creation function `RTDS_ProcessCreate()` in `RTDS_OS.c` threads to be deleted are detected because their `pStack` field in the `RTDS_globalProcessInfo` list has been set to `NULL`. The RTDS process context (`RTDS_GlobalProcessInfo` structure in the `RTDS_globalProcessInfo` list) is then freed and the thread is deleted at ThreadX level.

9 - TTCN-3 reference guide

9.1 - Acronyms

TC	Test Component
MTC	Main Test Component
PTC	Parallel Test Component
TSI	Test System Interface
TCI	TTCN-3 Control Interface
TMC	Test Managment and Control
TRI	TTCN-3 Runtime Interface
TE	TTCN-3 Executable
SUT	System Under Test
SA	SUT Adaptor
TM	Test Management
CH	Component Handling

9.2 - TTCN-3 architecture

9.2.1 Port type

Ports are points where communications take place.

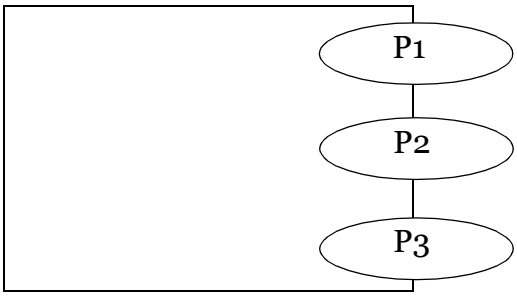
```
type port Myporttype{
  inout Message1
}
```

9.2.2 Component type

Component type defines which ports are associated with a component.

```
type component cmp1{
```

```
  port Myporttype P1;
  port Myporttype P2;
  port Myporttype P3;
}
```



component type cmp1

It is also possible to declare variables, constants and timers in a component type. All these declarations would be visible in testcases, functions and altsteps runing on an instance of this component type.

```
type component cmp1{
  port Myporttype Myport;
  var integer Myinteger;
  timer Mytimer;
}
```

9.2.3 Test system interface

A test system interface is declared like a component. The TSI ports connect the test system to the SUT. Every variable, constant and timer declarations in TSI will have no effect.The TSI has no name and can be called by the special operation **system**. The component type for the TSI is references by the keyword **system** in testcase declaration:

```
testcase testcase_name() runs on MTC_cmptype system TSI_cmptype
{}
```

9.2.4 Test system

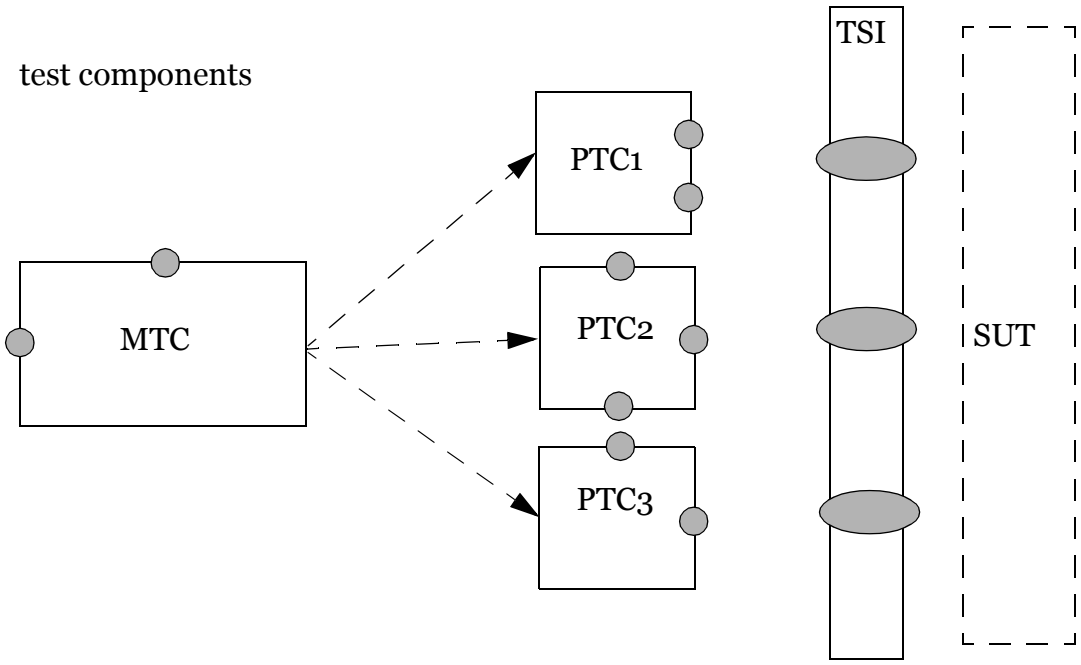
A test system is always composed of a main test component (MTC), and many possible parallel test components (PTC).The component type for the MTC is references by the keyword **runs on** in testcase declaration:

```
testcase testcase_name() runs on MTC_cmptype {}
```

MTC is automatically created when testcase is executed. The MTC has no name and can be called by the special operation **mtc**. PTC are created by the MTC inside testcase:

```
testcase testcase_name() runs on MTC_cmptype system TSI_cmptype {
  var cmptype PTC1 := cmptype.create;
  var cmptype PTC2 := cmptype.create;
  var cmptype PTC3 := cmptype.create;
}
```

`cmptype` could be any defined component type, including `MTC_cmptype` and `system_cmptype`. The special operation **self** return the component reference of the component in which this operation is called.



9.2.5 Communication

For communication between SDL and TTCN, a SDL message with x parameters, have to be present in TTCN has a record with x fields in the same order as the SDL parameters. The record name is the same as the message name. The name of TSI ports have to be the name of the SDL channels which communicate with environment. The name of the component type used as system interface in TTCN has to be the same as the system name in SDL.

9.2.5.1 Connection

Connections between components and test system interface are dynamically configured.

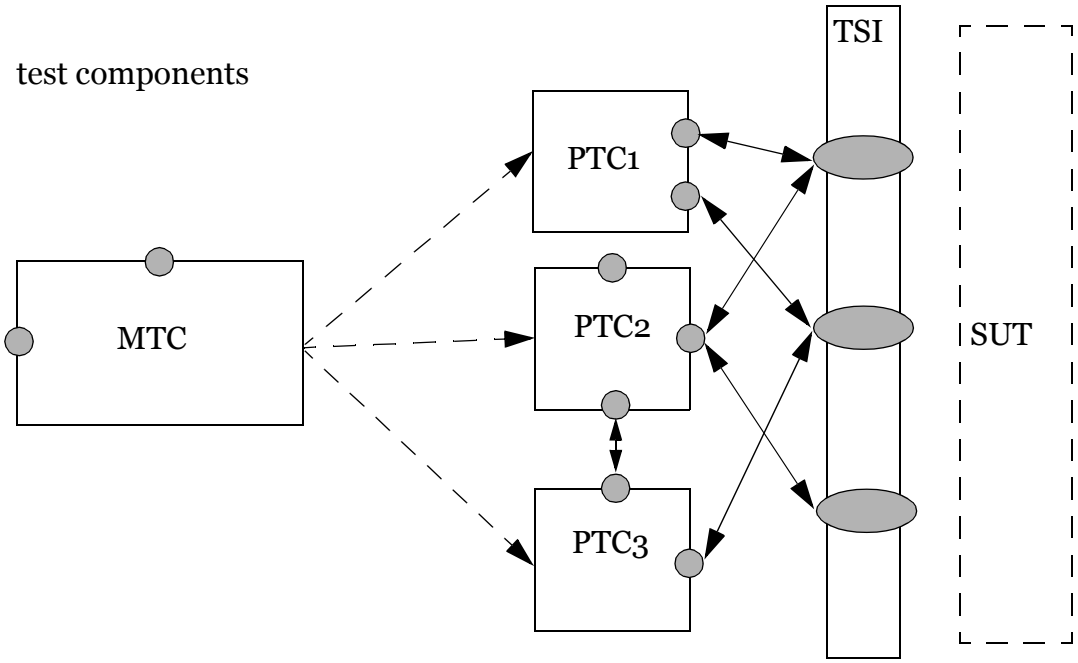
To map or unmap any test component ports to system ports:

```
map (TC_name:port_name, system:port_name)
unmap (TC_name:port_name, system:port_name)
```

To connect or disconnect test component among themselves:

```
connect (TC_name:port_name, TC_name:port_name)
disconnect (TC_name:port_name, TC_name:port_name)
```

TC_name could be predefined operator **mtc, system** or **self**.



9.2.6 Starting PTC behaviour

PTC behaviour are described into functions. A specific clause `runs on` is required during the function declaration to signal the component type:

```
function f_name(parameters) runs on cmptype
```

To start execution of the behaviour described in this function, the `start` operation is called, with the function as parameter:

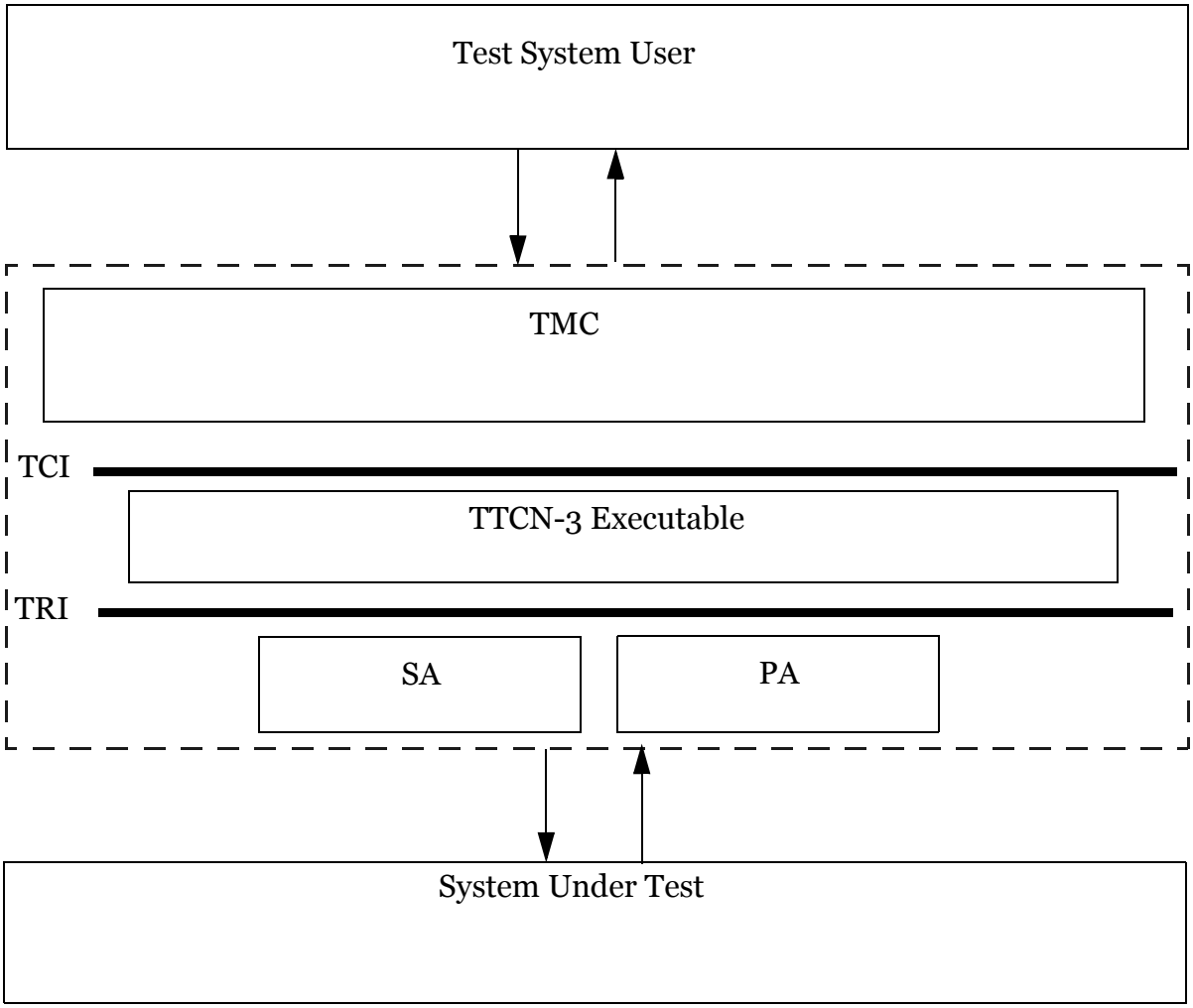
```
PTC_name.start(f_name(parameters))
```

PTC_name must be of the same type than the component type of the `runs on` clause.

9.3 - TTCN-3 test system anatomy

A TTCN-3 test system is composed of three mains entities:

- TTCN-3 Executable (TE): executes TTCN-3 module.
- TTCN-3 Control Interface (TCI): defines the interactions between the TE and the Test Sytem User.
- TTCN-3 Runtime Interface (TRI): defines the interactions between the TE and the System Under Test.



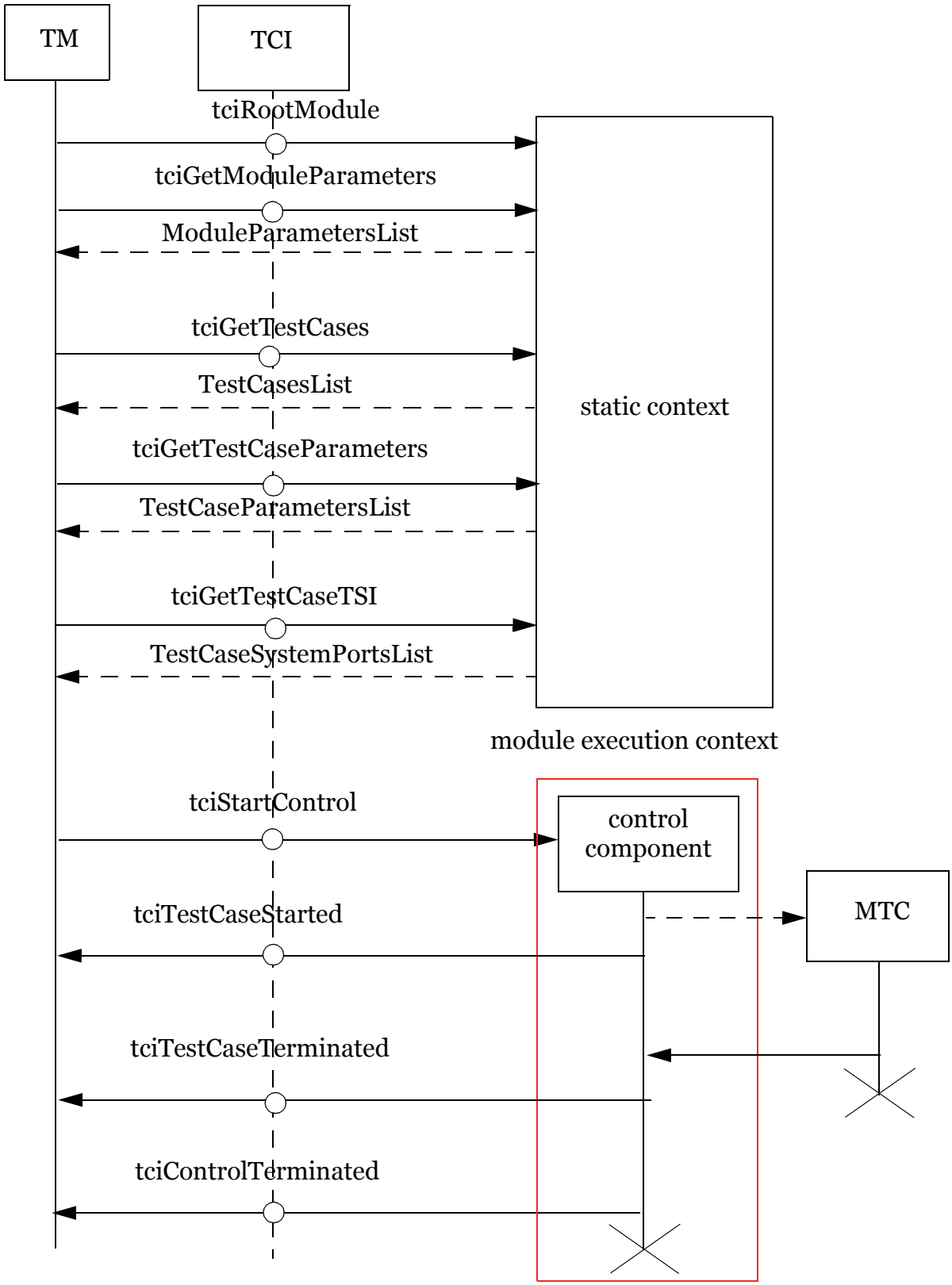
Structure of a TTCN-3 test system

9.3.1 TTCN-3 Control Interface

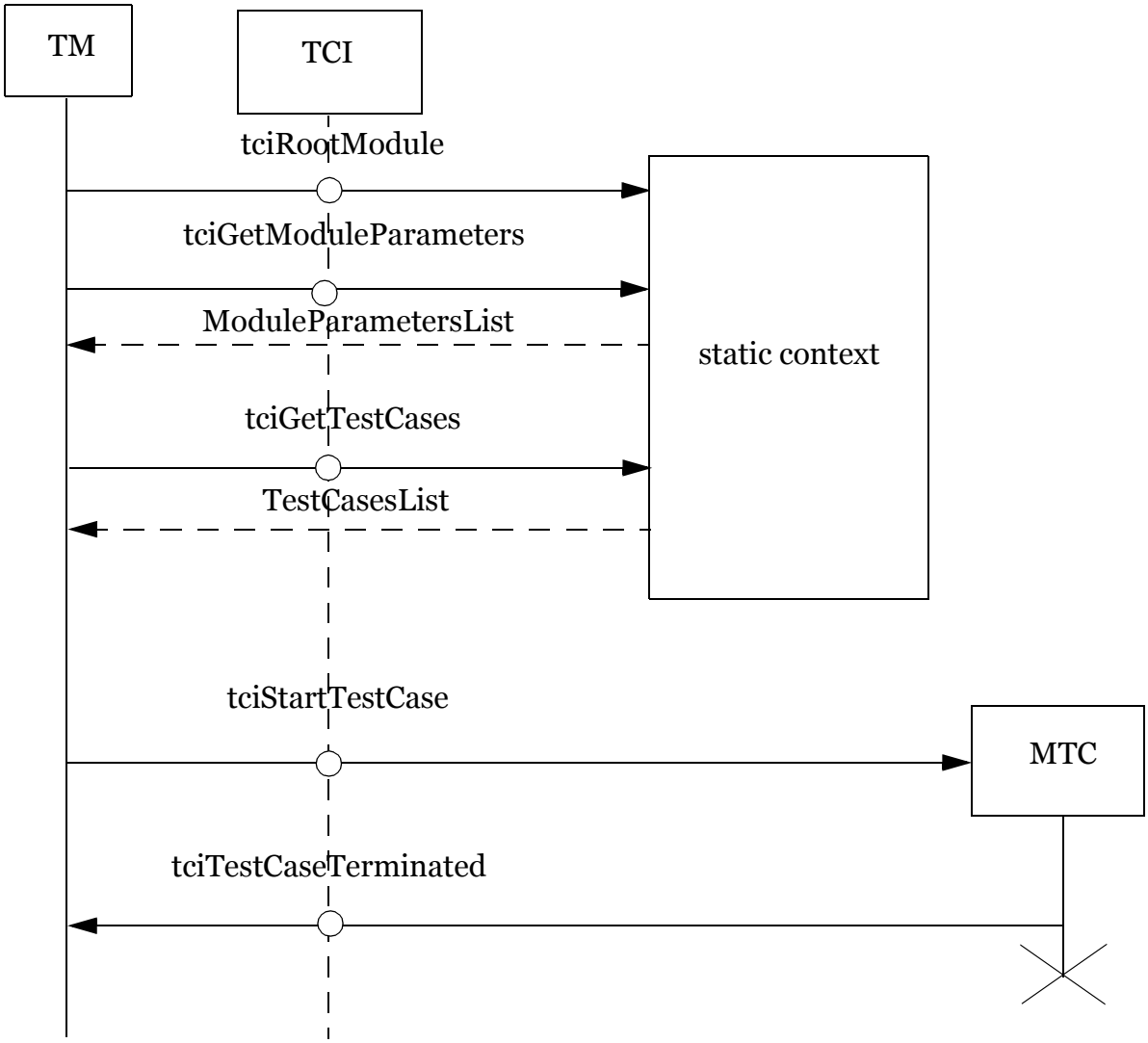
There are two types of operation that could be called inside TCI:

- Synchronous operations will be used by the TM to collect some information. these operation will immediatly return wanted data (getImportedModule, tciGetModuleParameters, tciGetTestCases, ...)
- Asynchronous operations to notify TM about testcase and control part behaviour (TestCaseStarted, TestCaseTerminated, ...)

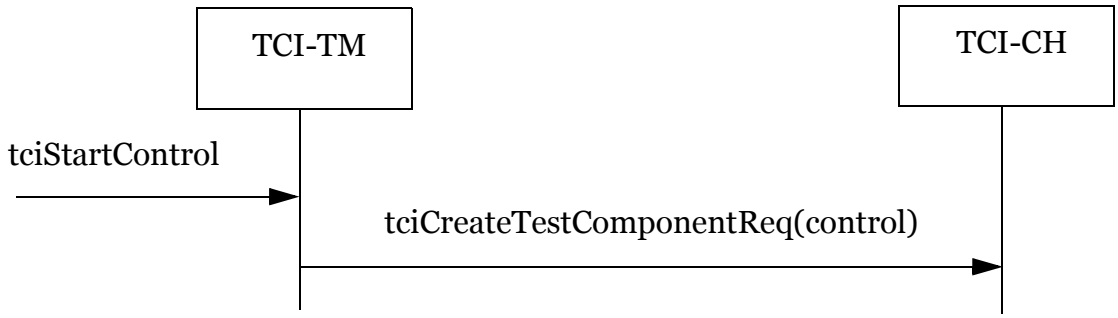
If a control part is present inside the test module, the tciStartControl will be called to create a thread inside module execution context, the control component. This is this component which will create and ask for start MTC.


TM
TCI
tciRootModule
tciGetModuleParameters
ModuleParametersList
tciGetTestCases
TestCasesList
tciGetTestCaseParameters
TestCaseParametersList
tciGetTestCaseTSI
TestCaseSystemPortsList
static context
module execution context
tciStartControl
control component
tciTestCaseStarted
MTC
tciTestCaseTerminated
tciControlTerminated

In case there is no control part, the TCI-TM will directly call the method `tciStart-TestCase` to create MTC for each testcase.:



When the operation tciStartControl is called in TCI-TM, a request tciCreateTestComponentReq is called in the TCI-CH to ask creation of a component. the first parameter of tciCreateTestComponentReq notices the type of component to create (CONTROL, MTC or PTC).



Then TCI-CH will call tciCreateTestComponent to create wanted component. The component identifier will be then returned to the TCI-TM. Then for creation of MTC and all

PTC, the TCI-CH will call tciCreateTestComponent for each component, and the identifier will be return to the control component, not to the TCI-TM. TCI-TM will just be informed of starting and ending of testcases.

9.3.2 TTCN-3 Executable

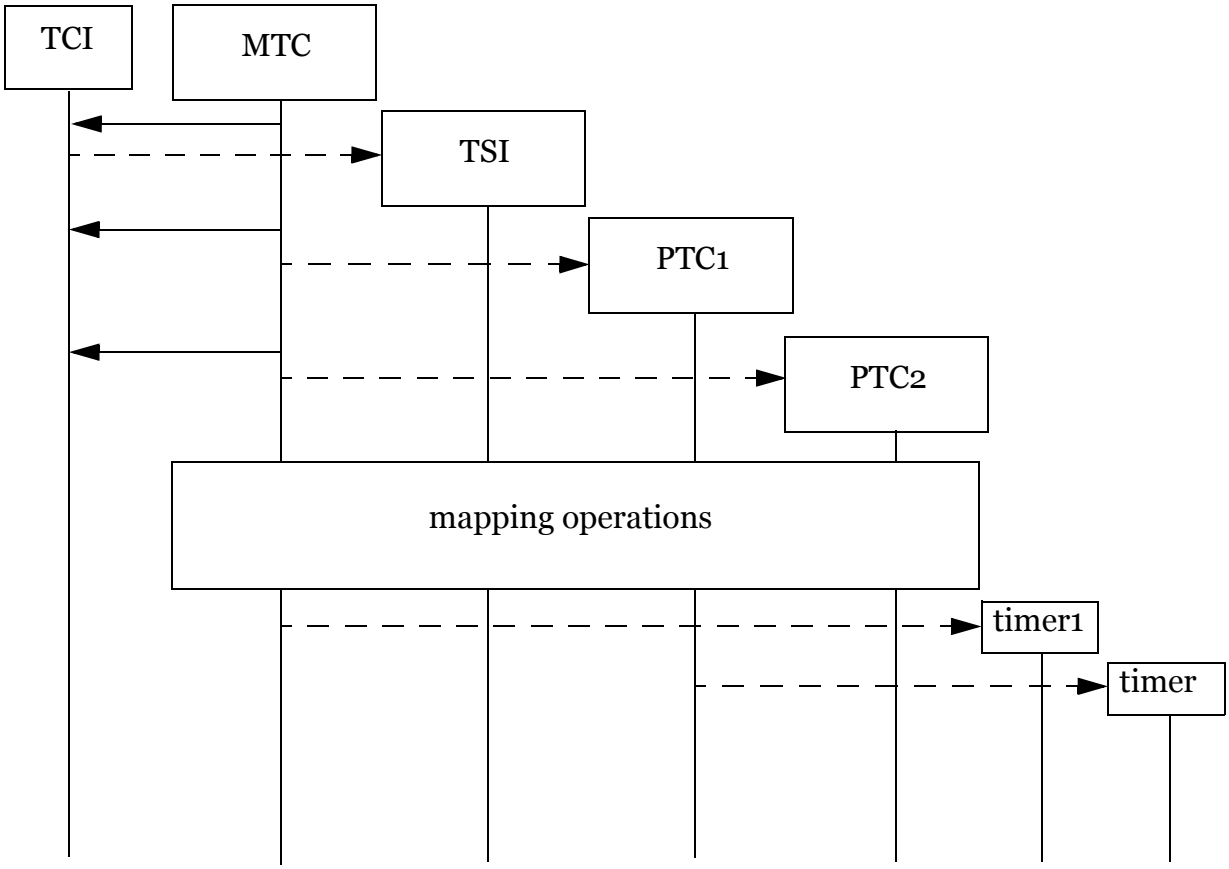
During its creation, a PTC can be declared normal or alive:

- normal type: PTC can execute only one behaviour function and then is killed,
- alive: many behaviour functions can be execute on the PTC, which must be explicitly killed with the **kill** operation.

In fact, all test component could act as alive component, but for normal component, an implicit **kill** command will be send after the first and unique start of function declared as behaviour.

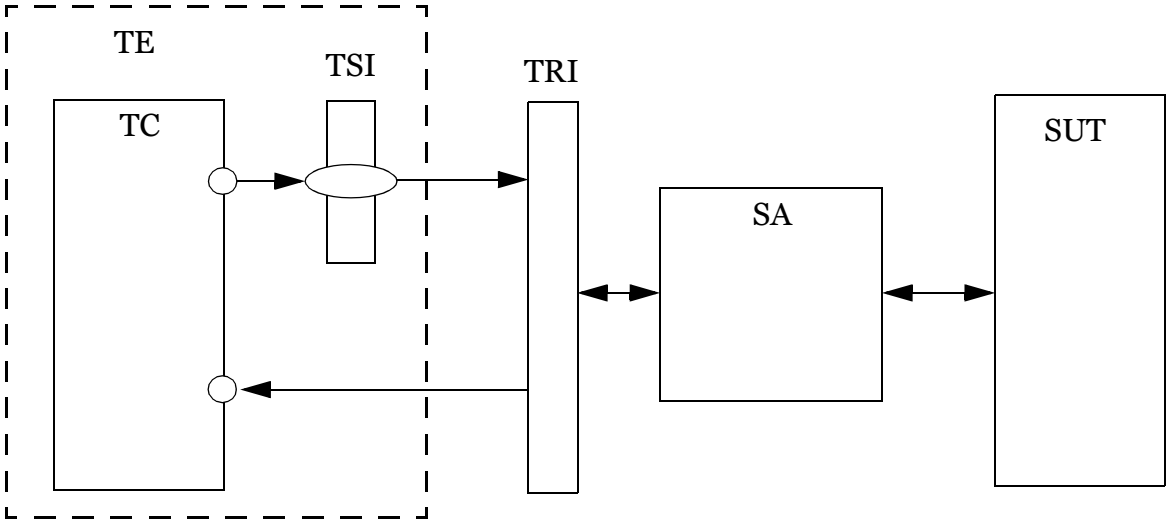
The MTC will be created by the TCI or a control part inside execution module context. If a system clause has been specified in testcase, the MTC will request to the TCI the creation of the TSI, otherwise a TSI of the same type than the MTC will be implicitly created. Then MTC will ask to TCI the creation of all needed PTC.

.



9.3.3 TTCN-3 Runtime Interface

The communication between the TE and the SUT takes place via the TRI.

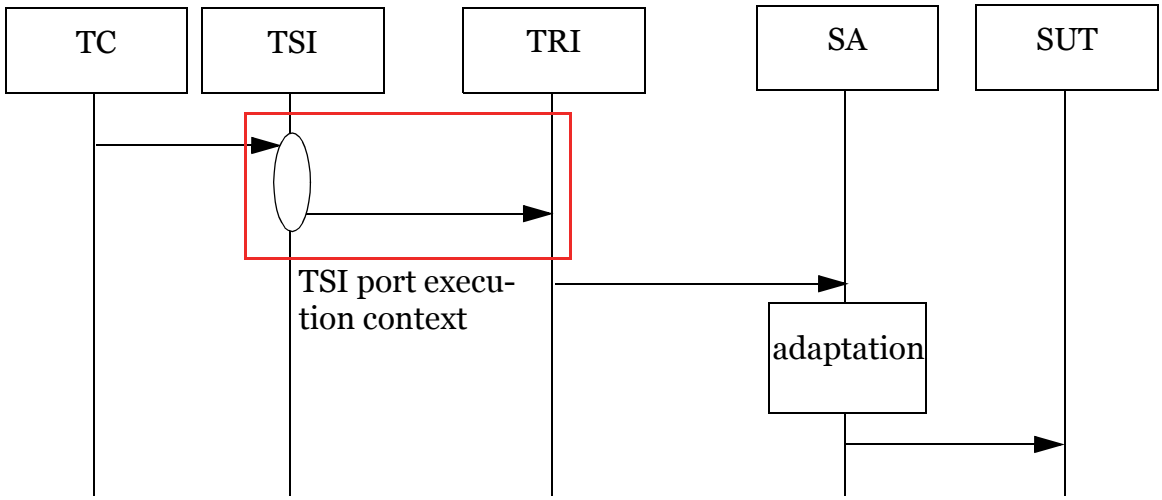


Communication links between TE and SUT

The SUT adaptor (SA) adapts message exchange betwen the TE and the SUT and is aware of the real mapping between TE and SUT. In the case of procedure-based communication,every operations with SUT are implemented into the SA.

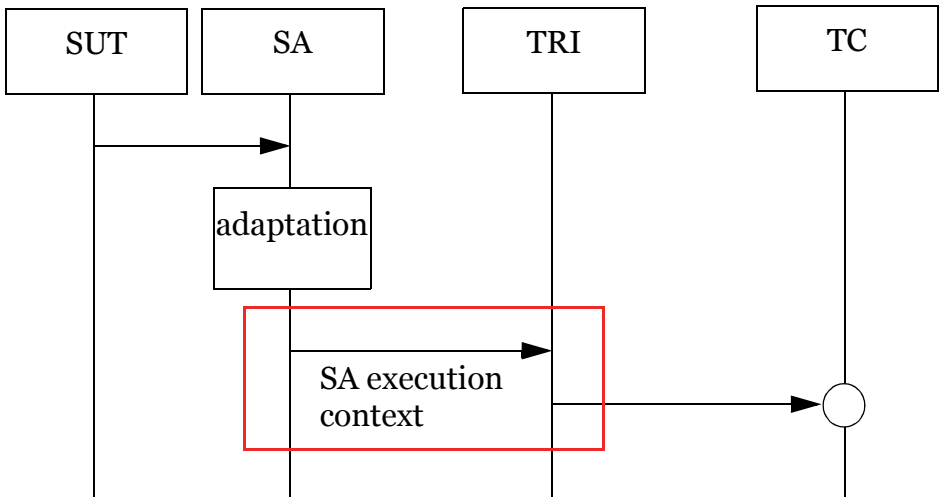
There are two ways of communication:

From the TE to the SUT: a message send from a test component to the SUT will first be send to a port of the TSI, and this is in this port execution context than TSI will call operations of TRI. TRI will then use SA to communicate with SUT.



From the SUT to the TE: every message send from the SUT will be adapted by the SA, then, into SA execution context, operations will be called to send and enqueue message

directly in the targeted port in test component. The TSI is not involved in this mechanism.



10 - TTCN-3 concepts support in simulation and generation

10.1 - Types and values

TTCN-3 concepts	Simulation	Generation
Basic types		
integer	OK	OK
float	OK	OK
boolean	OK	OK
objid	NOK	NOK
verdicttype	OK	OK
bitstring	OK	OK
hexstring	OK	NOK
octetstring	OK	OK
charstring	OK	OK
universal charstring	OK	NOK
constant type	OK	OK
Structured types		
record	OK	OK
record of	NOK	NOK
set	OK	OK
set of	NOK	NOK
optional field	NOK	OK
enumerated	OK	OK
union	OK	OK
Special types and values		
anytype	NOK	NOK
address	NOK	NOK

Table 30: types and values

TTCN-3 concepts	Simulation	Generation
port	OK	OK
component	OK	OK
default	NOK	NOK
recursive types	NOK	NOK
infinity as value	NOK	NOK
infinity as constraint	OK	OK
omit	NOK	NOK
Type compatibility		
non-structured types	OK	OK
structured types	OK	OK
component types	OK	OK
comunication operations	OK	OK

Table 30: types and values

10.2 - Operators

TTCN-3 concepts	Simulation	Generation
Arithmetic operators		
addition	OK	OK
substraction	OK	OK
multiplication	OK	OK
division	OK	OK
modulo	OK	OK
remainder	OK	OK
String operators		
concatenation	OK	NOK
Relational operators		
equal	OK	OK

Table 31: operators

TTCN-3 concepts	Simulation	Generation
less than	OK	OK
greater than	OK	OK
not equal	OK	OK
greater than or equal	OK	OK
less than or equal	OK	OK
Logical operators		
logical not	OK	OK
logical and	OK	OK
logical or	OK	OK
logical xor	OK	OK
Bitwise operators		
bitwise not	NOK	NOK
bitwise and	NOK	NOK
bitwise or	NOK	NOK
bitwise xor	NOK	NOK
Shift operators		
shift left	NOK	NOK
shift right	NOK	NOK
Rotate operators		
rotate left	NOK	NOK
rotate right	NOK	NOK

Table 31: operators

10.3 - Modular

TTCN-3 concepts	Simulation	Generation
language clause	NOK	NOK
module parameters	NOK	NOK

Table 32: modular

TTCN-3 concepts	Simulation	Generation
groups of definitions	OK	
Importing from module		
single definition	NOK	NOK
all definition	OK	OK
groups	NOK	NOK
definitions of the same kind	NOK	NOK
non-TTCN-3 modules	NOK	NOK
language clause	NOK	NOK

Table 32: modular

10.4 - Template

TTCN-3 concepts	Simulation	Generation
complex type as parameter	NOK	NOK
global template	OK	OK
local template	OK	OK
in-line template	NOK	NOK
modified template	OK	NOK
template as parameters	NOK	OK
Template operations		
match	OK	OK
valueof	OK	OK

Table 33: template

10.5 - Template matching mechanisms

TTCN-3 concepts	Simulation	Generation
Value		

Table 34: template matching mechanisms

TTCN-3 concepts	Simulation	Generation
specific value	OK	OK
omit value	NOK	OK
Instead of values		
complement list	NOK	NOK
value list on basic type	OK	OK
value list on complex type	NOK	NOK
any value (?)	OK	OK
any value or none (*)	OK	OK
range	OK	NOK
superset	NOK	NOK
subset	NOK	NOK
pattern	OK	NOK
Inside values		
any element (?)	NOK	OK
any element or none (*)	NOK	OK
permutation	NOK	NOK
Attributes		
length restriction	NOK	OK
ifpresent	OK	OK

Table 34: template matching mechanisms

10.6 - Tests configuration

TTCN-3 concepts	Simulation	Generation
non-concurent tests	OK	OK
concurent tests	NOK	OK
test-case parameters	OK	OK
runs on clause	OK	OK

Table 35: tests configuration

TTCN-3 concepts	Simulation	Generation
with clause	NOK	NOK
Port type		
message	OK	OK
signature	OK	NOK
mixed	OK	NOK
Procedure signatures		
in	OK	NOK
out	NOK	NOK
inout	NOK	NOK
non-blocking procedure	NOK	NOK
nowait call	NOK	NOK
call	NOK	NOK
getreply	NOK	NOK
exceptions	NOK	NOK
timeout exception	NOK	NOK
Component type		
multiple component	NOK	NOK
extends	OK	NOK
Component references		
mtc	NOK	OK
system	NOK	OK
self	NOK	OK
sender	NOK	NOK
to	NOK	NOK
from	NOK	NOK
value	OK	NOK

Table 35: tests configuration

10.7 - Functions and altsteps

TTCN-3 concepts	Simulation	Generation
functions	OK	OK
altsteps	NOK	NOK
Predefined conversion functions		
int2char	OK	OK
int2unichar	OK	NOK
int2bit	OK	NOK
int2hex	OK	NOK
int2oct	OK	NOK
int2str	OK	NOK
int2float	OK	OK
float2int	OK	OK
char2int	OK	OK
char2oct	OK	NOK
unichar2int	OK	NOK
bit2int	OK	NOK
bit2hex	OK	NOK
bit2oct	OK	NOK
bit2str	OK	NOK
hex2int	OK	NOK
hex2bit	OK	NOK
hex2oct	OK	NOK
hex2str	OK	NOK
oct2int	OK	NOK
oct2bit	OK	NOK
oct2hex	OK	NOK
oct2str	OK	NOK

Table 36: functions and alsteps

TTCN-3 concepts	Simulation	Generation
oct2char	OK	NOK
str2int	OK	NOK
str2oct	OK	NOK
str2float	OK	NOK
predefined size functions		
lengthof	OK	NOK
sizzeof	OK	NOK
sizeoftype	OK	NOK
predefined presence functions		
ispresent	OK	OK
ischosen	OK	NOK
predefined string handling functions		
regexp	NOK	NOK
substr	OK	OK
replace	OK	OK
Other predefined function		
rnd	NOK	NOK

Table 36: functions and alsteps

10.8 - Statements

TTCN-3 concepts	Simulation	Generation
Basic statements		
assignment	OK	OK
if-else	OK	OK
select case	OK	OK
for loop	OK	OK
while loop	OK	OK

Table 37: statements

TTCN-3 concepts	Simulation	Generation
do while loop	OK	OK
label and goto	OK	NOK
stop execution	NOK	NOK
returning control	OK	NOK
logging	OK	NOK
Statements and operations for alternative behaviours		
alternative bahaviour	OK	OK
re-evaluation of alternative	NOK	NOK
interleaved behaviour	NOK	NOK
activate a default	NOK	NOK
deactivate a default	NOK	NOK
guard conditions	OK	OK

Table 37: statements

10.9 - Operations

TTCN-3 concepts	Simulation	Generation
Connection operations		
connect	NOK	OK
disconnect	NOK	OK
map	NOK	OK
unmap	NOK	OK
Test component operations		
create	NOK	OK
create alive	NOK	NOK
start	NOK	OK
stop	NOK	NOK
kill	NOK	NOK

Table 38: operations

TTCN-3 concepts	Simulation	Generation
alive	NOK	OK
running	NOK	OK
done	NOK	OK
killed	NOK	OK
Communication operations		
send	OK	OK
receive	OK	OK
any port	NOK	OK
all port	NOK	OK
trigger	NOK	NOK
call	NOK	NOK
getcall	OK	NOK
reply	OK	NOK
getreply	NOK	NOK
raise	NOK	NOK
catch	NOK	NOK
check	NOK	OK
clear	NOK	OK
start	NOK	OK
stop	NOK	OK
halt	NOK	OK
Timer operations		
start	OK	OK
stop	OK	OK
read	NOK	NOK
running	OK	OK
timeout	OK	OK
Test verdict operation		

Table 38: operations

TTCN-3 concepts	Simulation	Generation
setverdict	OK	OK
getverdict	OK	OK

Table 38: operations

10.10 - Attributes

TTCN-3 concepts	Simulation	Generation
display	NOK	NOK
encode	NOK	NOK
variant	NOK	NOK
extension	NOK	NOK

Table 39: attributes

11 - Mapping of SDL data types to TTCN data types

SDL	TTCN-3
Predefined types:	
BOOLEAN	boolean
CHARACTER and CHARSTRING	charstring
INTEGER	integer
NATURAL	Not existing. But it can be create as sub-type of integer.
REAL	float
PID	Not existing.
DURATION	Not existing.
TIME	Not existing.
Data types:	

Syntype definition:
`SYNTYPE syntypeName = typeName`
`  CONSTANTS value1, value2, value3`
`ENDSYNTYPE;`
	`type typeName syntypeName`
	`  (value1, value2, value3);`

Array definition:
`SYNTYPE IndexSort = Integer`
`  CONSTANTS indexMin : indexMax`
`ENDSYNTYPE;`
`NEWTYPE ArrayType`
`  ARRAY (IndexSort, DataType1)`
`ENDNEWTYPE;`
	`type DataType1 ArrayType [indexMax];`
	TTCN forces index to be an integer.

Struct definition:
`NEWTYPE structName`
`STRUCT`
`  nameField1 field1Type;`
`  nameField2 field2Type;`
`  ...`
`ENDNEWTYPE;`
	`type record structName`
	`  {`
	`  field1Type nameField1,`
	`  field2Type nameField2,`
	`  ...`
	`  }`

Table 40: SDL data types to TTCN-3 data types mapping

SDL	TTCN-3

Choice definition:
`NEWTYPE choiceName`
`CHOICE`
`  nameField1 field1Type;`
`  nameField2 field2Type;`
`  ...`
`ENDNEWTYPE;`
	`type union choiceName`
	`  {`
	`  field1Type nameField1,`
	`  field2Type nameField2,`
	`  ...`
	`  }`

Literals definition:
`NEWTYPE literalsName`
`LITERALS`
`  literal1, literal2, literal3`
`ENDNEWTYPE;`
	`type enumerated literalsName`
	`  { literal1, literal2, literal3 }`

Synonym definition:
`SYNONYM synonymName typeName :=`
`value;`
	`const typeName synonymName := value;`

Communication	
`SYSTEM systemName`	`type component systemName {...}`

Channel ch with signal messOut coming out from the system sys and messIn coming in the system sys	A type port should be defined like this:
	`type port ch_type message`
	`  {out messIn; in messOut;};`
	And a port should be defined in correspondant component sys:
	`type component sys`
	`  { port ch_type ch; };`

`SIGNAL signalName(typeName);`	`type record signalName` `  { typeName paramName }`
Message message1 received by system on the channel ch	`ch.send(message1);`
Message message2 sent by system on the channel ch	`ch.receive(message2);`

Table 40: SDL data types to TTCN-3 data types mapping

12 - TTCN-3 Code generation

12.1 - Basic principles

The code generation feature for TTCN has several aspects in common with the one for SDL, as described in “Basic principles” on page 132:

- Everything in a test suite runs on a component, which is very similar to a SDL process, as all components execute in parallel;
- Test components communicate with each other and with the SUT via messages;
- Both languages support timers.

However, there are also significant differences between the two languages in the way they handle these concepts:

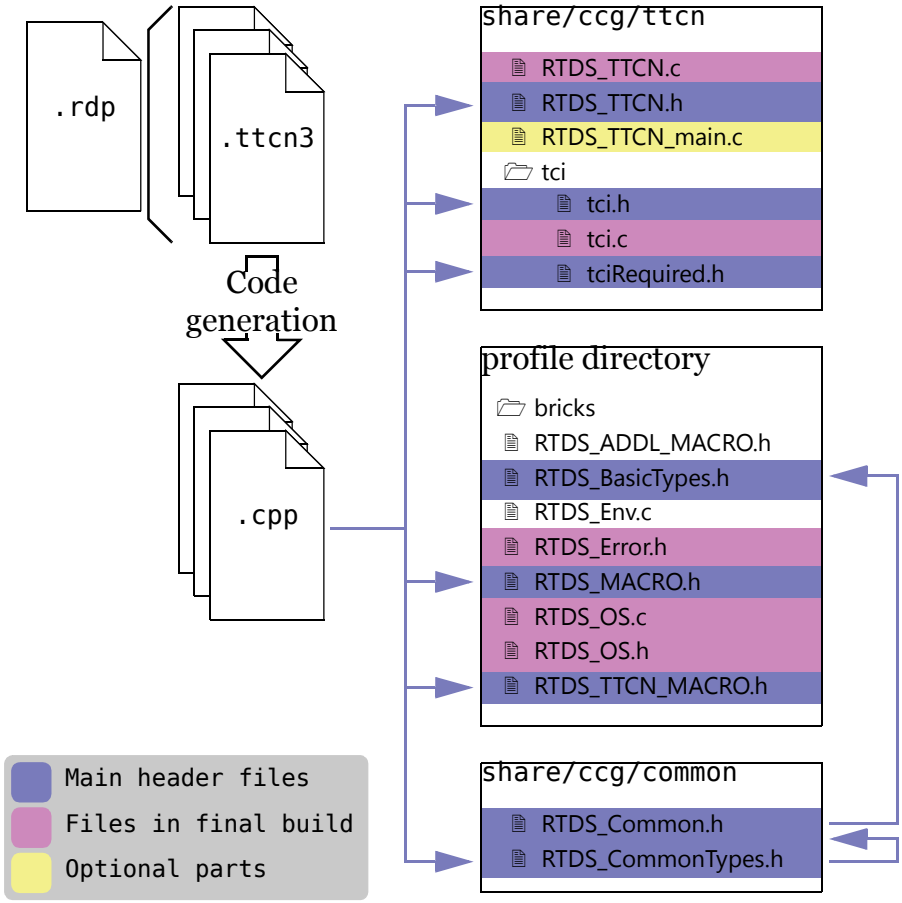
- TTCN testcases are not organized in transitions. There is no concept of testcase state in TTCN where a given set of messages can be expected. A testcase is just a continuous suite of statements that can pause its execution at any time to receive any message.
- In SDL, the message queues are implicit and a single one is attached to each process instance. In TTCN, the message queues are explicitely managed via ports, and a component can have any number of ports, accepting different sets of incoming and outgoing messages.
- The mechanism to handle message reception is very different in SDL and TTCN:
  - In SDL, if a process instance is waiting for a message, it is blocked on the reading of its single incoming queue. When an actual message is received, the transition corresponding to it will execute, and then the process instance will wait for the next message to be received.
  - In TTCN, a message is usually received within an `alt` statement, which allows to wait for messages on different ports, and/or to specify conditions for message receptions, or to test if other kinds of events have happened. So the mechanism is based on what is called a *snapshot*: A partial view of the testcase is "frozen", including the state of message queues associated to ports, its variables used in boolean conditions, and so on... Then all conditions and events specified in the `alt` statement are tested, and the execution continues with the branch for the first one that is satisfied.
    The concept of template also allows to test the contents of a message, and the TTCN semantics specifies that the message should be kept in the port’s message queue if the condition is not satisfied.
- Timers are also handled very differently in SDL and in TTCN: In SDL, only the process instance that started the timer can test for its time-out, and the corresponding event is a message. In TTCN, any component can test for any timer time-out, and there is a specific event for this, which is not based on a message.

Considering all of the above, the TTCN code generation is done as follows:

- Because of the common concepts between TTCN and SDL, the generated code uses the same RTOS integration, as described in “C code generation with a RTOS” on page 132. However, only the RTOS services implemented in the macros in `RTDS_MACRO.h` are used; The generated code is not based on the integration bricks.

- The snapshot mechanism requires to have a more important level of control on message queues that what is needed in SDL: In SDL, only a blocking wait on a message queue is required, which is possible in every RTOS. In TTCN, the snapshot and template mechanism requires to be able to test if a message is present on a given queue, and to read the message without actually removing it from the queue. So only the RTOS allowing to do that can be supported for TTCN code generation. Therefore, today, only the POSIX and Win32 integrations are supported.
- Since TTCN testcases are not transition-based, scheduling is not available. Each component instance is mapped to a thread in the generated executable.
- TTCN requires some mechanisms to be implemented which were not needed in SDL:
  - Some of these are specific to the RTOS, for example, the creation of a message queue that is not associated to a process instance. For these, an additional file named `RTDS_TTCN_MACRO.h` has been added in the integration itself.
  - Some of these do not depend on the RTOS, or can use the basic mechanisms provided in the integration. For example, the handling of the information associated to a TTCN component, or the snapshot mechanism itself. For these, a set of files have been created in `$RTDS_HOME/share/ccg/ttcn`, which will be included in all generated executables for TTCN. This directory also includes a partial implementation of the standard TTCN Control Interface (TCI) in the subdirectory tci. For more details, see “TTCN Control Interface” on page 263.
- Some concepts such as templates are much more easily expressed using object-orientation. So the code generated for TTCN is always C++, regardless of the language chosen in the generation options.

A summary of the main organization of the various files used for TTCN code generation is shown on the following diagram:

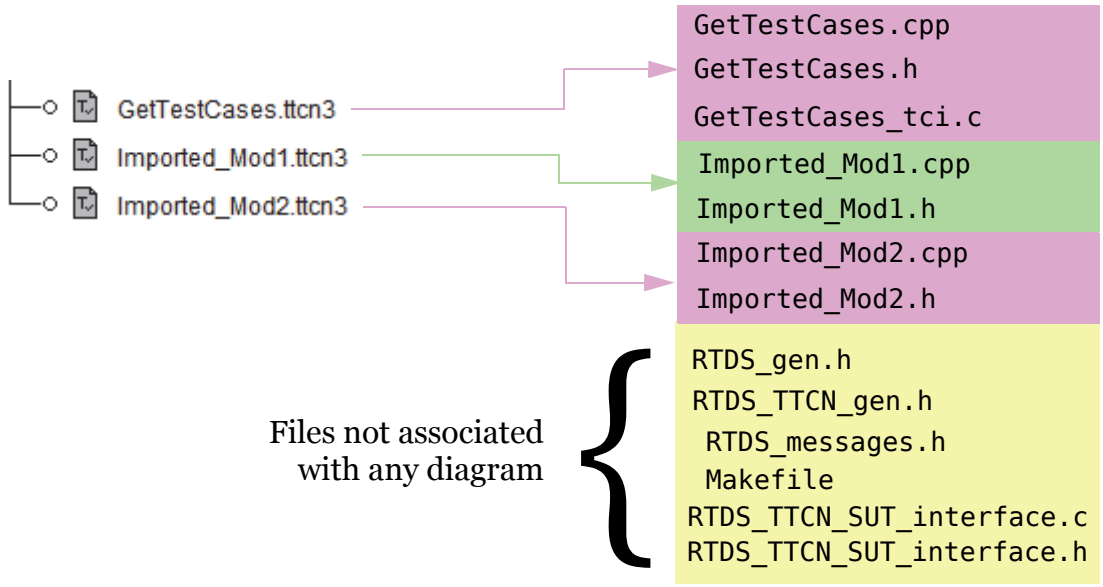


For TTCN code generation, the following options are also available:

- If the project also contains a SDL system, the code for the TTCN and the system can be generated together. In this case, an implementation of the Test System Interface is automatically provided, making the testcases and the SDL processes communicate with each other.
- A main function can be generated with the TTCN code. If this option is not selected, no main function is generated and the generated code can only be used as a library. The entry points for this library are the functions from the TCI implemented by RTDS, and optionally some other RTDS-specific functions. More details on these functions in “TTCN Control Interface” on page 263.
  If a main function is generated, the file `RTDS_TTCN_main.c` is included in the final build, and provides a command line interface allowing to run either the control parts found in the TTCN modules, or any test in any module. If used with the preceding option (TTCN + SDL co-generation), the generated executable includes all tests with the tested system, allowing to run all tests on the system in any order.
- As for SDL, it is also possible to debug a TTCN test suite, either with or without the tested SDL system. Note that in this case, the previous option is not significant: A non-interactive main function is always generated, running the main control part and the tested system if any.

12.2 - Generated Files

The generated files for a typical TTCN test suite are shown on the figure below. In this example, `GetTestCases` imports `Imported_Mod1` and `Imported_Mod1` imports `Imported_Mod2`:



For each module a C++ source file and a header file are generated.

12.3 - TTCN Control Interface

Prototypes of available TCI functions can be found in the `tci.h` file in `$RTDS_HOME/share/ccg/ttcn` directory. These functions are implemented in `tci.c`:

- `tciRootModule`: set root module.
- `tciGetTestCases`: returns testcases list of current root module.
- `tciStartControl`: starts control part.

These functions use information from the generated code provided in a specific generated file called `<module name>_tci.c`. Additional functions are available to browse the generated information:

- `RTDS_TTCN_GetAllRootModules` returns the list of all available modules.
- `RTDS_TTCN_GetModuleTestcases` returns the list of all testcases available in the specified module.
- `RTDS_TTCN_GetProcessNumber` returns the identifier of the control part. Returns 0 if no control part.
- `RTDS_TTCN_GetProcessFunction` returns the function to launch on the control component.
- `RTDS_TTCN_ExecuteTestcase` starts a specific testcase.

12.4 - Automatic main function generation (`RTDS_TTCN_main.c`)

RTDS allows to automatically generate main function in `RTDS_TTCN_main.c`. If using this main function, generated code can be interactivly executed.

Usage of the generated executable with automatic main function:

- With no parameters, will go in ineractive mode.
- 'start' will start the control part for the default root module.
- 'start <testcase name>' will start the testcase with this name in the default module.

Some functions required by the TCI are also defined in this file.

- `tciError`: Called when an error occurs in the test.
- `tciControlTerminated`: Called when the execution of a control part ends.
- `tciTestCaseStarted`: Called when the execution of a testcase starts.
- `tciTestCaseTerminated`: Called when the execution of a testcase ends.

12.5 - Adaptation to a target

RTDS allows to generate TTCN alone (without SDL system). In this case, it is required to adapt to the SUT to make possible communication between the testsuite and the system under test.

12.5.1 Generated data types

As there is no specific message type in TTCN-3, any data type can be sent or received in a test case via a template. The generated code for basic data types and for exchanged messages is the same. For example, a TTCN record is generated as a C structure with a field for each field of the record :

```
type record mRequest { integer param1, integer param2};
```

will be generated as the following structure in C++ code:

```
typedef struct _mRequest
  {
  int param1;
  int param2;
  } mRequest;
```

12.5.2 Requirements

To allow communication between the TTCN and an external SUT, an adaptation layer is required. The interface between the TTCN and the SUT has to be done in RTDS_TTCN_SUT_interface.h file. To add this file during compilation, it is necessary to add -DRTDS_TTCN_SUT_INTERFACE in the compilation option.

If the SUT is linked with the TTCN, this is the automatically generated main which will start the SUT. The macro RTDS_TTCN_SUT_INIT specifies the start function of the SUT. This macro has to be defined is the compiler option of the generation profil (e.g.: -DRTDS_TTCN_SUT_INIT = SUT_init).

12.5.3 Communication from TSI to SUT

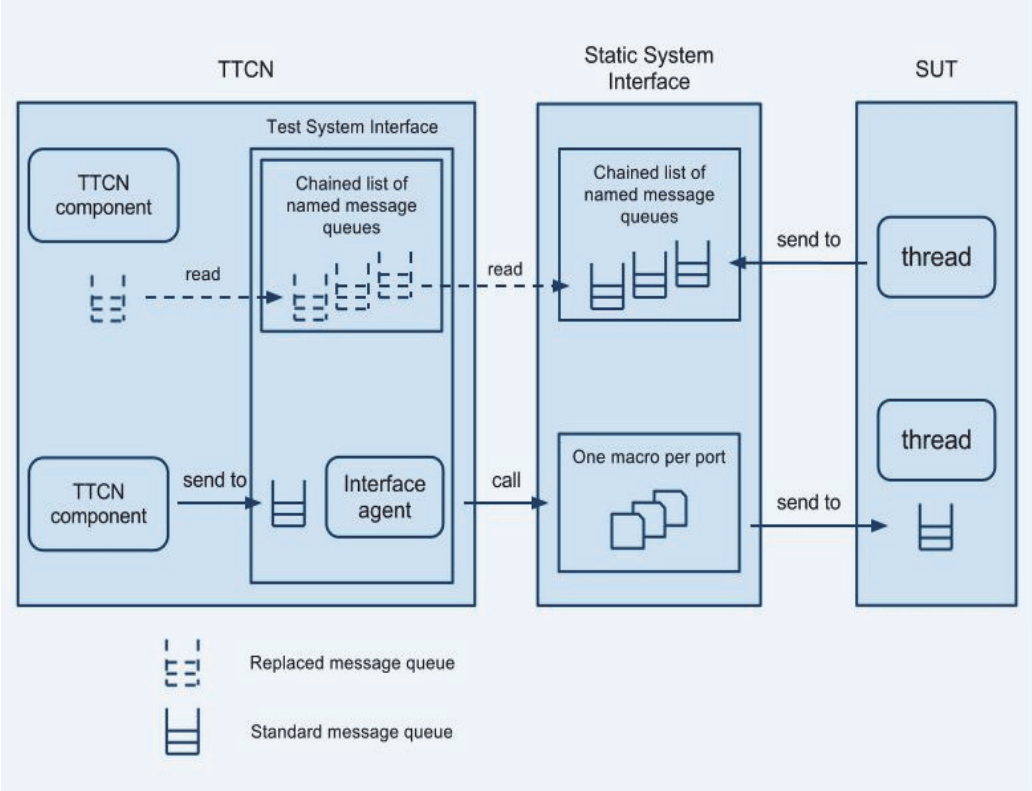
Communication in TTCN is done via a template.

```
template mRequest request := { param1 := 42, param2 := 13};
```

TTCN-3 templates are generated as C++ class. These classes have two methods:

- `match`: used to match a received message with a specified template.
- `valueof`: returns a pointer on the record value of the template.

For sending messages from TTCN-3 test suite to SUT, a macro for each port of the TSI is called : RTDS_TTCN_SEND_MESSAGE_VIA_TSI_PORT_*portName*. Definitions of this macro have to be done in `RTDS_TTCN_SUT_interface.h` file.



The following send command in TTCN:

```
sut_port.send(request);
```

will be generated as:

```
RTDS_TTCN_SEND_MESSAGE_VIA_TSI_PORT_sut_port(message);
```

Parameter `message` is of type `RTDS_MessageHeader*`. `RTDS_MessageHeader` is a transport structure used in SDL and TTCN generated code to communicate. The messageNumber field is the message identifier. The generated values are in RTDS_gen.h. A value is generated for each type sent through the port prefixed with `RTDS_message`. In our example it would be `RTDS_message_mRequest`. The `pData` field of `RTDS_MessageHeader` is a pointer on the data associated to the message. In our case, `pData` is the return value of the method `valueof` of `request` template.

For example, if port sut_port of the TSI is used to send mRequest_1 and mRequest_2, the messageNumber field is tested to identify which message has been sent. Then a specific function is called in the SUT to deal with each message:

```
#define RTDS_TTCN_SEND_MESSAGE_VIA_TSI_PORT_sut_port(MESSAGE_HEADER) \
  switch (MESSAGE_HEADER->messageNumber) {\
    case RTDS_message_mRequest_1: \
       handleRequestMessage_1(((mRequest_1*)(MESSAGE_HEADER->pData))-
>param1); \
      break; \
    case RTDS_message_mRequest_2: \
       handleRequestMessage_2(((mRequest_2*)(MESSAGE_HEADER->pData))-
>param1,((mRequest_2*)(MESSAGE_HEADER->pData))->param2); \
      break; \
    }
```

12.5.4 Communication from SUT to TSI

In generated code for TTCN, a global chained list of first element RTDS_TTCN_systemInterfacePorts of type RTDS_TTCN_SystemInterfacePort* is declared to store all the system interface ports.

The list of ports must be defined in the function defined by RTDS_TTCN_SUT_INIT. For each incoming port of the TSI, a new element of type RTDS_TTCN_SystemInterfacePort* must be inserted in the list.

For example, to add port sut_port to this list:

```
SUT_output_queue.portName = "sut_port";
SUT_output_queue.portMessageQueue = RTDS_NEW_MESSAGE_QUEUE;
SUT_output_queue.next = NULL;
RTDS_TTCN_systemInterfacePorts = &SUT_output_queue;
```

To send the message to the test case, use the RTDS_MSG_QUEUE_SEND_TO_QUEUE_ID_FROM_SUT macro defined in RTDS_TTCN_MACRO.h.

This macro has five parameters:

- MESSAGE_NUMBER: Numerical identifier of the message.
- LENGTH_DATA: Length of the data associated with the message.
- P_DATA: Pointer on the data associated with the message.
- RECEIVER: Pointer to the message receiver instance.
- PORTID: Pointer to the queue where the message is sent.

To send a message of type AnswerMessageType to sut_port:

```
AnswerMessageType * message;
RTDS_MSG_QUEUE_SEND_TO_QUEUE_ID_FROM_SUT(
  RTDS_message_mAnswer, sizeof(AnswerMessageType), message,
  NULL, SUT_output_queue.portMessageQueue);
```

12.6 - Naming convention

All file names used by the generated C code are prefixed with RTDS_TTCN_ to avoid name clash with generated files. In these files all TTCN function names, TTCN variables and TTCN types used internally are prefixed with RTDS_TTCN_.

12.7 - Types used in TTCN-3 generated code - RTDS_TTCN.h

Table 41: TTCN-3 types

RTDS_TTCN_PortIdList		List of every ports of a component.
portId	TriPortId*	Descriptor for a port instance needed in TRI and TCI.
next	RTDS_TTCN_PortIdList*	Pointer on the next timer Id.
RTDS_TTCN_TimerInfo		Descriptor for a timer.
timerID	long	Identifier for the timer.
timerNumber	int	Numerical identifier for the timer. This is the timer identifier in RTDS_TTCN_gen.h
timerUniqueID	unsigned long	Unique identifier for the timer. Each timer will have its own, even if it has the same name as another one. Used only for trace.
duration	TriTimerDuration	Duration of the timer if a default value has been specified during timer declaration.
timeoutValue	TriTimerDuration	System tick counter value when timer will go off
timerStatus	TimerStatusType	Status of the timer (inactive, running or expired).
RTDS_TTCN_TimerInfoList		List of timers of current component.
timerInfo	RTDS_TTCN_TimerInfo*	Descriptor for a timer.
next	RTDS_TTCN_TimerInfoList*	Pointer on next timer of the list.

Table 41: TTCN-3 types

RTDS_TTCN_PortMappingInfo		Descriptor for ports instance.
queueControlBlock	RTDS_RtosQueueId	Identifier for the message queue for the instance. The type RTDS_RtosQueueId must be defined in RTDS_BasicTypes.h.
currentMessage	RTDS_MessageHeader*	Current message of the queue.
connectionType	RTDS_TTCN_ConnectionType	Type of connection of the port (map or connect).
portStatus	PortStatusType	Status of the port (started, halted, stopped or error).
portId	TriPortId*	Descriptor for a port instance needed in TRI and TCI.
mappedPort	RTDS_TTCN_PortIdList*	List of all ports connected to the current port.
next	RTDS_TTCN_PortMappingInfo*	Pointer to the next descriptor.
RTDS_TTCN_Template_Pool		Pool of the templates defined in current component.
templateID	RTDS_TTCN_Template*	Identifier of the template. RTDS_TTCN_Template class has to be defined in RTDS_TTCN.h.
next	RTDS_TTCN_Template_Pool*	pointer to next template if any.
RTDS_TTCN_GlobalComponentInfo		Descriptor for a component instance.
componentID	TriComponentId	Identifier for the component. The type TriComponentId has to be defined in tci.h.

Table 41: TTCN-3 types

componentKindType	TciTestComponentKindType	Type of the component (control, system, MTC, PTC or alive PTC).
componentStatus	ComponentStatusType	Status of the component (inactive, running, killed or error).
componentVerdict	verdicttype	Local verdict of current component.
RTDS_currentContext	RTDS_GlobalProcessInfo*	Descriptor for the instance within which the event happened.
portMappingInfo	RTDS_TTCN_PortMappingInfo*	Descriptor for ports instance.
templatePool	RTDS_TTCN_Template_Pool*	Pool of the templates defined in current component.
timerInfoList	RTDS_TTCN_TimerInfoList*	Descriptor for timers of the component.
cmpVariables	void*	Descriptor for all variables, timers and constant of current component.
next	RTDS_TTCN_GlobalComponentInfo*	Pointer to the next descriptor.
RTDS_TTCN_SystemInterfacePort		Descriptor for a port of the system interface.
portName	char*	Name of the port.
portMessageQueue	RTDS_RtosQueueId	Identifier for the message queue for the instance. The type RTDS_RtosQueueId must be defined in RTDS_BasicTypes.h.
next	RTDS_TTCN_SystemInterfacePort*	pointer to the next port if any.

12.8 - Generated TTCN-3 constants and prototypes - RTDS_TTCN_gen.h

This file defines constants for components, testcases, timers and internal messages used in TTCN-3 system. It also defines protoypes for the functions implementing the components.

More precisely:

- A `#define`'d constant is generated for each TTCN-3 component type. This constant has the name of the component type prefixed with `RTDS_process_RTDS_Component_`.
- A `#define`'d constant is generated for each testcase. This constant has the name of the testcase prefixed with `RTDS_process_RTDS_TestCase_`.
- A `#define`'d constant is generated for the control part (if any) of each module. This constant has the name of the module prefixed with `RTDS_process_RTDS_Control_`.
- A unique `#define`'d constant is generated for the system component with the name `RTDS_process_System`.
- A `#define`'d constant is generated for each timer. This constant has the name os the timer prefixed with `RTDS_TTCN_`.
- `#define`'d constants are also generated for internal system messages.
- A prototype for each function implementing a component or a testcase. The prototype is not written directly, but via a macro defined in `RTDS_MACRO.h`.

13 - Debugger integrations

13.1 - Tasking Cross View Pro debugger integration

13.1.1 Version

PragmaDev SDL-RT debugger is interfaced with Tasking Cross View Pro C166/ST10 debugger version 7.5 rev. 2. It has been developed and tested with Tasking 167 instruction set simulator (C167CS) on Windows 2000 SP2 host. This integration is only available on Windows platform.

13.1.2 Interface

CrossView Pro provides a COM object interface on MS-Windows RTDS connects to. When starting up RTDS SDL-RT debugger does the following:

- `xfw166 -RegServerS` command line
  Activates the COM object
- Calls the Init() method on the COM object to pass the debugger options
  The debugger options should contain the debug configuration file such as in the example provided: `-tcfg C:\c166\ETC\SIM167CS.CFG`
- Load the executable file in Tasking debugger with the `abs` extension.
- If unsuccessful, tries to load the executable with the default extension (`exe`).
- Step once to get into the main function.

13.1.3 Make utility

The generated makefile is not supported by Tasking make utilities. The gnu make should be used instead such as the one distributed in our distribution in Cygwin environment.

13.1.4 Restrictions

Because Tasking Cross View Pro debugger does not allow to call functions on target, it is not possible to send an SDL-RT message to the system running on the target. An SDL-RT test process can be used to do so. In that case the test process should be called RTDS_Env so that it is not generated for a non SDL-RT debug profile.

13.2 - gdb debugger integration

13.2.1 Version

PragmaDev SDL-RT debugger is interfaced with gdb debugger version 2002-04-11-cvs on Windows and version 5.2 on Unix. It has been tested on Windows 2000 SP4 host, Solaris 7, Solaris 8, Linux Mandrake 8.0 and Linux Red Hat V7.2.

13.2.2 Interface

RTDS is interfaced with gdb through pipes. The annotated gdb mode is used to parse the information. When starting up RTDS, SDL-RT debugger does the following:

- `<gdb> -quiet -annotate=2`
- `set height 0`
- `set width  0`

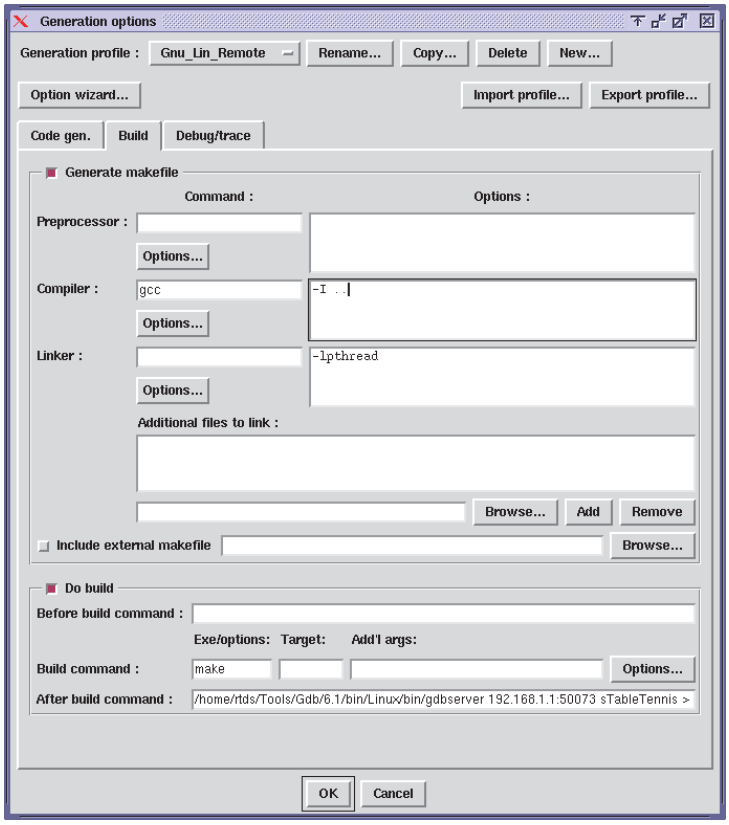
13.2.3 Remote debugging

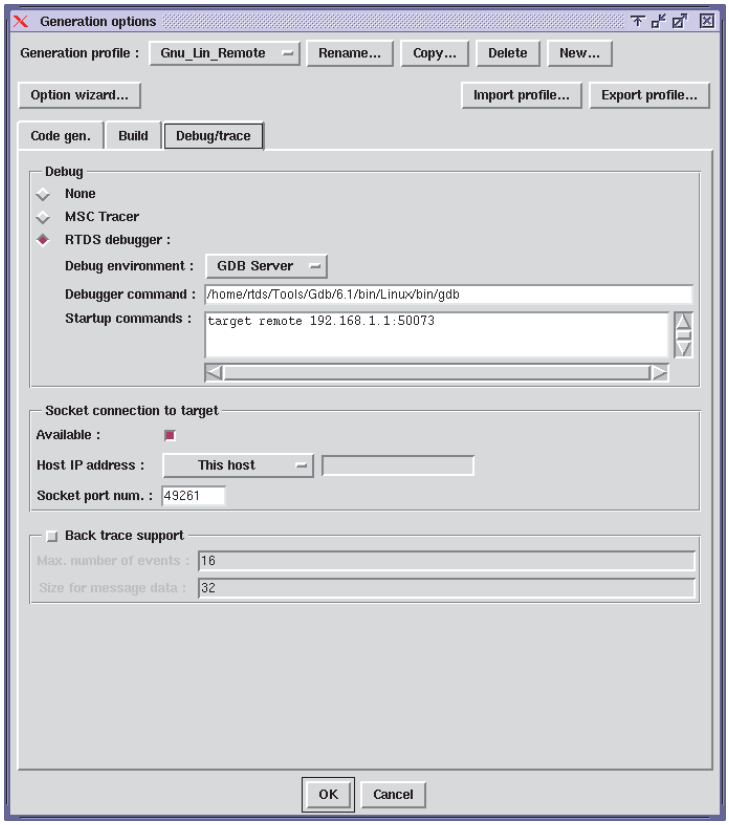
SDL-RT debugger also supports the remote gdb debugging facility based on gdbserver. This integration has been tested with gdb v6.1 on Linux. When it comes to debug on target, one of the key aspect is to download the executable on the target. In the example profile below, for the sake of clarity, the target is the host. So launching gdbserver on the target is simply:

```
gdbserver <executable name>
```

On the client side, standard gdb is started and remote connection commands are provided to connect to the target:

```
target remote <IP address>:<port number>
```


Generation options
Generation profile :
Gnu_Lin_Remote
Rename...
Copy...
Delete
New...
Option wizard...
Import profile...
Export profile...
Code gen.
Build
Debug/trace
Generate makefile
Command :
Options :
Preprocessor :
Options...
Compiler :
gcc
-I ..
Options...
Linker :
-lpthread
Options...
Additional files to link :
Browse...
Add
Remove
Include external makefile
Browse...
Do build
Before build command :
Exe/options:
Target:
Add'l args:
Build command :
make
Options...
After build command :
/home/rtds/Tools/Gdb/6.1/bin/Linux/bin/gdbserver 192.168.1.1:50073 sTableTennis >
OK
Cancel


Generation options
Generation profile :
Gnu_Lin_Remote
Rename...
Copy...
Delete
New...
Option wizard...
Import profile...
Export profile...
Code gen.
Build
Debug/trace
Debug
None
MSC Tracer
RTDS debugger :
Debug environment :
GDB Server
Debugger command :
/home/rtds/Tools/Gdb/6.1/bin/Linux/bin/gdb
Startup commands :
target remote 192.168.1.1:50073
Socket connection to target
Available :
Host IP address :
This host
Socket port num. :
49261
Back trace support
Max. number of events :
16
Size for message data :
32
OK
Cancel

13.3 - MinGW debugger integration

13.3.1 Version

PragmaDev SDL-RT debugger is interfaced with MinGW gdb debugger version 5.2.1. This integration has been tested with MinGW gcc version 3.4.5.

13.3.2 Library

Socket communication requires to link with an external MinGW library: `libws2_32.a`.

13.3.3 Interface

RTDS is interfaced with gdb through pipes. The annotated gdb mode is used to parse the information. When starting up RTDS, SDL-RT debugger does the following:

- `<gdb> -quiet -annotate=2`
- `set height 0`
- `set width  0`

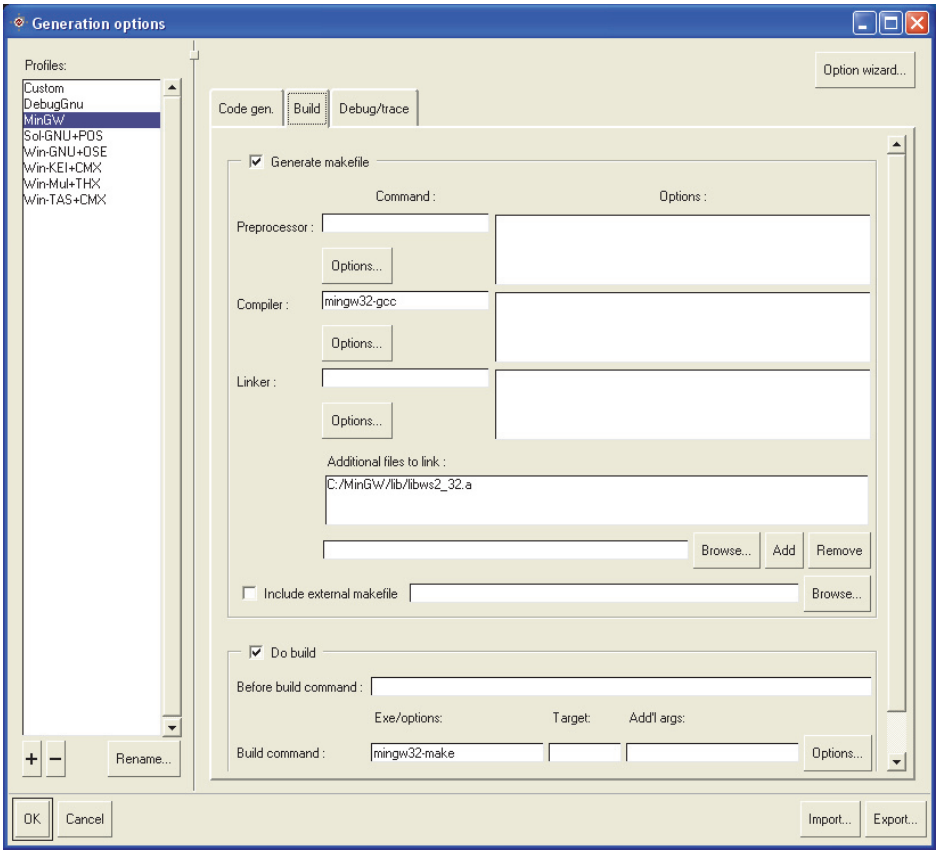
13.3.4 Console

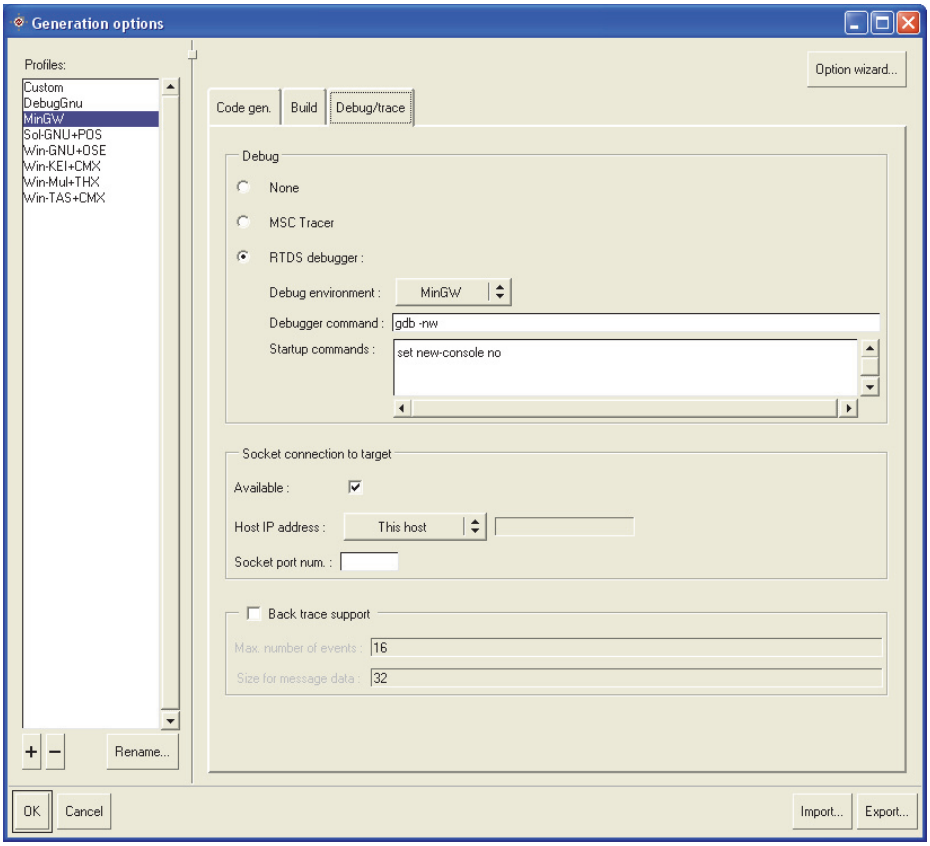
gdb should not start a new console. The following command should be run when starting the debugger:

```
set new-console no
```

13.3.5 Restriction

MinGW gdb only works when socket communication is available, otherwise RTDS will not be able to interrupt the debugger.


Generation options
Profiles:
Custom
DebugGnu
MinGW
Sol-GNU+POS
Win-GNU+OSE
Win-KEI+CMX
Win-Mul+THX
Win-TAS+CMX
Option wizard...
Code gen.
Build
Debug/trace
Generate makefile
Command :
Options :
Preprocessor :
Options...
Compiler :
mingw32-gcc
Options...
Linker :
Options...
Additional files to link :
C:/MinGW/lib/libws2_32.a
Browse...
Add
Remove
Include external makefile
Browse...
Do build
Before build command :
Exe/options:
Target:
Add'l args:
Build command :
mingw32-make
Options...
Rename...
OK
Cancel
Import...
Export...


Generation options
Profiles:
Custom
DebugGnu
MinGW
Sol-GNU+POS
Win-GNU+OSE
Win-KEI+CMX
Win-Mul+THX
Win-TAS+CMX
Option wizard...
Code gen.
Build
Debug/trace
Debug
None
MSC Tracer
RTDS debugger :
Debug environment :
MinGW
Debugger command :
gdb -nw
Startup commands :
set new-console no
Socket connection to target
Available :
Host IP address :
This host
Socket port num. :
Back trace support
Max. number of events :
16
Size for message data :
32
Rename...
OK
Cancel
Import...
Export...

13.4 - XRAY debugger integration

13.4.1 Version

PragmaDev *SDL-RT debugger* is interfaced with XRAY debugger version 4.6ap. It has been developed and tested with MiPlus (uITRON for Nucleus) and the ARMulator on Windows 2000 SP2 host. This integration is only available on Windows platform using XRAY in graphical mode (without `-cmd` command line option).

Several version of XRAY have been tested. The table below summarizes the result of these tests.

Table 42: Tested versions of XRAY and gdb with Arm target and OSE

Debugger	Type	Host	Target	Mode	Result	Comment
XRAY	**RDI**	**Win**	**Armulator**	**Run mode**	**KO**	
XRAY	RDI	Win	Armulator	Freeze mode	OK	Time does not increase.
XRAY	RDI	Solaris	Armulator	Run mode	KO	
XRAY	RDI	Solaris	Armulator	Freeze mode	KO	
XRAY	OSE soft kernel	Win	host		KO	Connects automatically but a load problem has been detected. gcc for OSE can not compile RTDS codec for printable traces.
gdb	OSE soft kernel	Win	host		OK	Requires OSE version of gcc. The system can not be stopped on the fly because the gdb does not receive the Ctrl-C.
XRAY	OSE soft kernel	Solaris	host		KO	Compiles but can not debug because it does not connect neither to the soft kernel nor to the Illuminator.
gdb	OSE soft kernel	Solaris	host		OK	
XRAY	RDI	Win	OSE evka7	Run mode	KO	Load should be done through the Illuminator. Freeze mode is not supported by the board. Run mode is way too different.
XRAY	RDI	Sol	OSE evka7	Run mode	N/A	Has not been tested.

13.4.2 Interface

Communication between RTDS *SDL-RT debugger* and XRAY is done through telnet. When starting up, RTDS *SDL-RT debugger* does the following:

- `xray.inc` configuration file generation that contain `tcpopen <tcp_port>`, where `<tcp_port>` is the socket port specified in the generation options or default `49250` if not defined. This file is used by XRAY to setup the telnet connection and is generated in the code generation directory.
- XRAY is start with `xray.inc` as a parameter: `xray -inc xray.inc`
- All available targets are listed in a choice window.

- The executable file is loaded in XRAY debugger.
- A single step is done once, to get into the main function.

13.4.3 Restrictions

13.4.3.1 Using Xray debugger

Using the `-cmd` option when starting the XRAY debugger, change its behavior. At the present time the only way to use the XRAY interface is to start debugger with its graphical interface.

It's also strongly recommended to only use the RTDS debugger and keep Xray interface mimimized. If the XRAY interface is used to drive debug, some errors could occur due to the fact that XRAY debugger does not send back all informations about operations that have been done and its state.

For example:

- a breakpoint set from the XRAY debugger must be removed from the xray interface.
- after a break when running program from Xray debugger : RTDS does not get the information and remains in the stopped state,

13.4.3.2 Sending messages

Because XRAY debugger does not allow to call several parameters functions on target, it is not possible to send an SDL-RT message to the system running on the target. An SDL-RT test process can be used to do so. In that case the test process should be called RTDS_Env so that it is not generated for a non SDL-RT debug profile.

13.4.3.3 Stopping timers

For the same reason that explained on "Sending messages" on page 277, timers can't be stopped from the RTDS debugger.

13.4.3.4 Message queue

The number of messages displayed in the process information area will not be updated and remains to zero.

13.5 - Multi 2000 debugger integration

13.5.1 Version

PragmaDev SDL-RT debugger is interfaced with Multi 2000 debugger version 4.01. It has been developed and tested with MIPS R3000 instruction set simulator on Windows 2000 SP4 host.

13.5.2 Interface

RTDS is connected with Multi through socket connexion (port 49248) and Multi is launched with the -nodisplay option. The launch command is:

```
<multi> -nodisplay -socket 49248 <executable name>
```

In case of problem, the SDL-RT debugger will retry to connect to Multi every second up to a maximum of ten times.

13.5.3 Target connexion

The target connexion is defined through commands defined in the generation profile to be executed by the debugger. For example, to connect to the MIPS instruction set simulator with a timer tick every 999 instructions the following commands should be set:

```
# Connects to the r3000 MIPS simulator
connect simmips -cpu=r3000 -rom
# Sets a simulated timer every 1000 instructions
target timer 999 1
```

13.5.4 Restrictions

Because Multi debugger does not allow to call functions on target, it is not possible to send an SDL-RT message to the system running on the target. An SDL-RT test process can be used to do so. In that case the test process should be called RTDS_Env so that it is not generated for a non SDL-RT debug profile.

13.6 - RTDS footprints

13.6.1 Static memory footprint

The following figures have been measured with gcc V 2.95.3-5 on a Pentium computer under Windows 2000 with an empty process (start transition only) called pEmpty.

```
.text           0x00401000      0x1400
.text           0x00401000      0x230 pEmpty.o
                0x00401000                pEmpty
.text           0x00401230      0x180 RTDS_Start.o
                0x0040127c                RTDS_Start
.text           0x004013b0      0xd4  RTDS_Env.o
                0x004013b0                RTDS_Env
.text           0x00401484      0xc4  RTDS_Utilities.o
                0x004014a8                RTDS_StringLength
                0x004014dc                RTDS_StringCompare
                0x00401484                RTDS_MemAlloc
.text           0x00401548      0xce0 RTDS_OS.o
                0x00401eec                RTDS_Sem_Info_Insert
                0x00402164                RTDS_GetSemaphoreId
                0x00401584                RTDS_GetTimerUniqueId
                0x00401660                RTDS_WatchDogFunction
                0x00401a58                RTDS_ProcessCreate
                0x004016e4                RTDS_StartTimer
                0x00401904
RTDS_GetProcessQueueId
                0x00401868                RTDS_StopTimer
                0x00402130                RTDS_SemaphoreIdTake
                0x00402084                RTDS_Sem_Delete
                0x00401cf8                RTDS_ProcessKill
                0x004019c4                RTDS_MsgQueueSend
                0x00401a34                RTDS_MsgQueueReceive
.data           0x00403000      0x8   RTDS_Start.o
                0x00403004
RTDS_globalSemaphoreInfo
                0x00403000
RTDS_globalProcessInfo
.bss            0x00404000      0x20
                0x00404000                __bss_start__=.
*(.bss)
*(COMMON)
COMMON          0x00404000      0x20  RTDS_Start.o
                                0x0   (size before relaxing)
                0x00404000
RTDS_globalSystemSemId
                0x00404010
RTDS_globalStartSynchro
```

```
                0x00404020                __bss_end__=.
```

To be summarized as:

- 4 436 bytes in ROM (note RTDS_Env is not included in this figure),
- 40 bytes in RAM

	VxWorks	uItron3.0	uItron4.0	Windows	CMX
compiler	gcc	armelf-gcc	armcc	gcc	cc166
target	SimSparcSol	ARM7TDMI	ARM7TDMI	Win32	Sim167CS
ROM pEmpty RTDS_Start RTDS_Utilities RTDS_OS **TOTAL**	 696 264 424 4528 **5912**	 768 224 316 6212 **7520**	 308 160 112 3140 **3720**	 540 304 196 4816 **5856**	 346 108 56 2142 **2652**
RAM	**8**	**14**	**C**	**10**	**10**

13.6.2 Dynamic memory allocation

Allocated when	Size	Freed when
Sending an SDL message	0	-
Receiving an SDL message	sizeof(RTDS_MessageHeader)	SDL transition is executed Note: not freed if signal is saved.
Starting a timer	sizeof(RTDS_TimerState)	When cancelled if watchdog has been fully deleted. Anyway when timer message is executed.

Table 43: Size of allocated memory by the RTDS kernel

	32 bits CPU	16 bits CPU
SDL message sizeof(RTDS_MessageHeader)	28	22
SDL process info sizeof(RTDS_GlobalProcessInfo)	40	16
SDL timer sizeof(RTDS_TimerState)	28	20

14 - RTDS commands

14.1 - rtds: main application

14.1.1 Usage

The `rtds` command actually runs RTDS. Its syntax is the following:

```
rtds \
   [ --no-server ] \
   [ -assoc-file=<file> --assoc-line=<line> ] \
   [ --diagram-file=<file> [ --symbol-id=<id> [ --symbol-line=<line> ] ] ] \
   [ <project file> ]
```

The meaning of the options are:

- `--no-server`:
  Prevents RTDS from opening a server connection. By default, RTDS runs in server mode, which means that trying to run another RTDS instance when one is already running will connect to the running one and open the requested element in the running instance. If the first instance was run with the `--no-server` option, the following one will not connect to it and run as if it were alone.
  Note that connecting to an already running instance is done only if an element is actually opened by the command, either by passing a project name or via command line options. If RTDS is run without options or arguments, it will run on its own. Note also that only the first RTDS instance run without the `--no-server` option can open the server connection. If another instance is run afterwards with no options or arguments, it will not be able to open the connection since it is already opened.
- `--assoc-file=<file> --assoc-line=<line>`:
  The `<file>` specified in the `--assoc-file` option must be a file generated from a diagram, i.e. either an exported PR file or a generated C code file. The `<line>` must be a valid line in the this file. Specifying these options will figure out the project, diagram and symbol from which the line in the file have been generated, open this project, open the diagram and highlight the symbol.
  If these options are used, no project file name must be passed as argument to the command.
  If a project is already opened in RTDS, it will be replaced by the project from which the file was generated (a confirmation is asked).
- `--diagram-file=<file> [ --symbol-id=<id> [ --symbol-line=<line> ] ]`
  The `<file>` must be a valid RTDS diagram file. If the `--symbol-id` option is used, the `<id>` must be a valid symbol internal identifier in the diagram. If the `--symbol-line` option is used, the `<line>` must be a valid line in the symbol identified by `<id>`. Specifying these options will open the requested diagram, highlighting the symbol line if needed.
  If no project is opened, the diagram will be opened alone and the project manager will iconify itself. If the opened project contains the requested diagram, the project will not be closed and the diagram will be opened inside the project. If the

opened project does not contain the diagram, a dialog box will pop up, asking if the diagram should be opened alone or added to the current project.
If these options are used, a project file name may be passed as argument to the command; this project must contain the specified diagram.

The only valid argument to the command is a project file name, specifying a project to open. As states above, it cannot be specified with the options `--assoc-file` / `--assoc-line`.

14.1.2 Environment variables

The following environment variables may be used by RTDS:

- `RTDS_HOME`: This variable *must* be set to be able to run RTDS. It must reference RTDS's installation directory (the one containing the `bin`, `doc`, `share` and `examples` sub-directories).
- `RTDS_DEFAULT_PREF_FILE`: This variable only has an effect the first time RTDS is launched. It identifies the file which will be copied to the current user's preference file. If this variable is not set, the preferences file is copied from:
  `$RTDS_HOME/share/conf/rtds.ini`
  If this file does not exist, RTDS will created a default preference file itself.
- `RTDS_FAST_SEARCH`: If set, this variable must reference the utility named `rtdsSearch` or `rtdsSearch.exe`, delivered with RTDS (in `$RTDS_HOME/bin`). It allows to perform searches on diagram files faster than the built-in search operation. It may be used for large projects with many diagrams.

14.2 - rtdsPrintAssoc: display association information

The `rtdsPrintAssoc` command is used to display the project, diagram, symbol identifier and symbol line corresponding to a line in a file generated from RTDS (exported PR file or generated C code file). Its syntax is the following:

```
rtdsPrintAssoc [-o] <file> <line>
```

where `<file>` is the name of the file generated from RTDS and `<line>` is a valid line in this file.

The command will then print:

- the name of the project file from which the given file was generated,
- the file name for the parent diagram for the symbol corresponding to the given line in the generated file,
- the internal identifier for this symbol,
- the line number in the symbol corresponding to the given line.

If no option is specified, the command will output a single line containg all the information above, in this order, separated by tabs.

If the `-o` option is specified, the command will output a set of options that can be passed to the `rtds` command, i.e.:

```
--diagram-file=<...>  --symbol-id=<...>  --symbol-line=<...>  <project file>
```

14.3 - rtdsGenerateCode: code generation

This command generates and optionnaly compiles the code for a given diagram in a given project. Its syntax is the following:

```
rtdsGenerateCode [-c] <project file> <diagram name> <profile name>
```

`<diagram name>` is the name that appears for the diagram in the project tree; `<profile name>` is the name of the generation profile as it appears in the generation options dialog. A full syntax and semantics check is made before the generation, except if the `-c` option is specified; in this case, the check is made in "critical only" mode.

If the specified generation profile specifies that a compilation must be made, it will automatically be run by the command.

14.4 - rtdsImportPR: PR/CIF file import

The `rtdsImportPR` command converts a PR/CIF file to a RTDS project. Its syntax is:

```
rtdsImportPR <options…> <pr file> <project file>
```

The `<pr file>` will be converted to a RTDS project, which will be saved to `<project file>`. All diagrams generated by the conversion are put in the same directory as the project.

The available options are:

- `-T <int> / --license-time-out=<int>`
  Time-out for license in seconds. If not set, command fails immediatly if license cannot be checked out.
- `-c / --case-insensitive`
  Make parser case insensitive for keywords. Default is to recognize keywords only if all lowercase or all uppercase.
- `-I / --ignore-cif`
  Ignore all CIF comments in imported file
- `-v / --verbose`
  Verbose mode. Default is to print only warnings and errors
- `-x / --allow-link-crossing`
  Allow link crossing in converted block and system diagrams
- `-a <type> / --auto-size-texts=<type>`
  Forces auto-sizing for diagrams. `<type>` may be `all`, `architecture` or `behavior`
- `-1 / --single-line`
  Put text for most symbols on a single line
- `-m / --split-on-commas`
  Split the text for symbols on commas.
- `-s <int> / --shortcut-text-threshold=<int>`
  Automatically create a shortcut text for symbols having more than the specified number of lines. Default is to never create shortcut texts.
- `-r <mode> / --external-references=<mode>`
  Mode for handling external references and Geode CIF includes. `<mode>` may be:
  - `k` to always keep references as in the PR file
  - `rk` to resolve references if possible and keep them if not [default]
  - `rd` to resolve references if possible and discard them if not

- `-i / --no-invisible-names`
  Remove names marked invisible in CIF. Ignored is CIF comments are ignored.
- `-z <real> / --zoom=<real>`
  Zoom factor for all coordinates and dimensions in CIF. Ignored if CIF comments are ignored.
- `-h / --shift-connectors`
  Correct positions given by Geode for "in" connectors. Ignored if CIF comments are ignored.
- `-p / --one-partition-per-state`
  When CIF comments are ignored: create one partition for each state in behavioral diagrams.
- `--partition-diagram-type=<diagram type>`
  Use only when importing a partition alone to specify the type for the partition's parent diagram. `<diagram type>` may be:
  - `sys` for a system diagram;
  - `blk` for a block diagram;
  - `blktype` for a block class diagram;
  - `prcs` for a process diagram;
  - `prcstype` for a process class diagram;
  - `prcd` for a procedure diagram;
  - `macro` for a macro diagram;
  - `compstate` for a composite state diagram;
  - `service` for a service diagram.
- `--partition-file-naming=<none|p|dp>`
  Allows to save partitions in separate files and setup naming scheme for these files:
  - `none` keeps the partitions in the diagram file [default]
  - `p` stores the partition in a file named after partition alone
  - `dp` stores the partition in a file named after diagram & partition

14.5 - rtdsExportPR: PR file export

This command exports the PR file corresponding to a given project. Its syntax is:

```
rtdsExportPR [-T <seconds>] [-h] [-f] [-s] [-c] <project file> <output file>
```

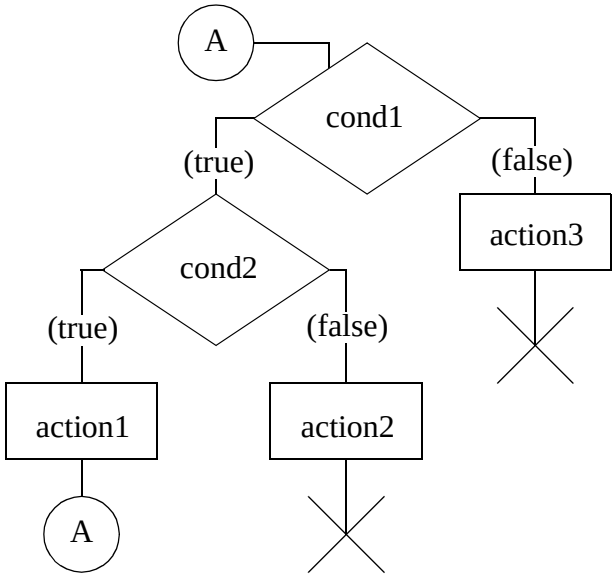
which exports the given project to the given `<output file>`.

The options are:

- `-T <seconds>`: specifies that the command should wait for the license for the given number of seconds. The default is 0, so if the option is not specified, the command fails immediatly if the license cannot be checked out.
- `-h`: generate a hierarchical PR file. This means that definitions for agents will be generated in their parent's code. The default is to generate a flat PR file, where agents are just defined as `REFERENCED` in their parent's code, and the code for the agent is elsewhere in the generated PR file. To avoid ambiguity when several agents have the same name, agent names are always fully qualified in flat exported PR files.
- `-f`: when used with an SDL-RT project, forces a full export. This means that the exported PR file *will not be a valid PR file*. Without this option, only a skeleton

of the project is exported, containing mainly the architecture and the message exchanges. This option has no effect on SDL projects.

- `-s`: prevents generated `SUBSTRUCTURE`s in blocks to be named. By default, sub-structures are named with the name of the block. This option should only be used with hierarchical PR files (`-h` option), since qualifiers cannot include unnamed sub-structures.
- `-c`: expands code for labels in behavioral diagrams at the first `JOIN` on the label. The default is to put the code for the label in a separated `CONNECTION` block.
- `-j`: tries to put `JOIN`s to labels referencing a decision in one of this decision's branches. This typically has an impact in the following situation:



The PR code generated by default for this diagram would be:

```
A:
DECISION cond1;
   (true):
     DECISION cond2;
        (true):
          TASK action1;
          JOIN A;
        (false):
          TASK action2;
          STOP;
     ENDDECISION;
   (false):
     TASK action3;
     STOP;
ENDDECISION;
```

With this option, the code will be:

```
A:
DECISION cond1;
   (true):
     DECISION cond2;
        (true):
          TASK action1;
        (false):
```

```
            TASK action2;
            STOP;
        ENDDECISION;
        JOIN A;
    (false):
        TASK action3;
        STOP;
ENDDECISION;
```

The `JOIN A` is then generated in a branch of the decision referenced by the label `A`, which is `DECISION cond1`.

- `-o`: deterministic order for export. The PR text is exported as far as possible in an order corresponding to the display order in diagrams.
- `-w`: displays a warning message if a complex DECISION cannot be exported without generating an additional JOIN not in the exported diagram.

14.6 - rtdsGenerateXmlRpcWrappers: XML-RPC wrapper generation

This command generates the XML-RPC wrappers for all external operators in a given RTDS SDL project. The exact syntax is:

```
rtdsGenerateXmlRpcWrappers <project file> <output header file> <output C file>
```

The generated files are:

- A C header file (`.h`) containing the translation to C for all SDL types used by operators. These types are figured out automatically and do not require any markup. The translation rules are as follows:

SDL type		C type	Comment
Base types	BOOLEAN	BOOLEAN	`#define`'d to an `int`. The two constants `FALSE` & `TRUE` are also defined.
	NATURAL	unsigned int	
	INTEGER	int	
	REAL	float	
	CHARACTER	char	
	TIME	long	
	DURATION	long	
	PID	unsigned long	
	CHARSTRING	RTDS_String	`RTDS_String` is a 2048 character array.

SDL type		C type	Comment
SYN-TYPE	on CHARSTRING	char[...]	The actual size for the string is extracted from the SIZE constraints if any.
	on other types	typedef	The constraints are ignored.
STRUCT types		typedef struct	The type for the fields are the C translations of the filed types in the SDL STRUCT. No pointers are used, even if a field has a complex type.
CHOICE types		typedef enum for the present field type + typedef struct with present field & union for the actual CHOICE.	The values in the enum type for the present field are the names of the field in the CHOICE prefixed with the CHOICE type name. The union field in the struct generated for the CHOICE itself is called __value.
STRUCT SELECT types		typedef struct with discriminating field and union.	The union field is called __value as for CHOICEs.
ARRAY types		typedef with C array	Array index must be an integer type. It is always supposed to start at 0. The constraints for its maximum value are taken into account if possible.
Enumerated types (LITERALS)		typedef enum	

- A C source file containing:
  - The encoding and decoding functions to and from XML-RPC for all types;
  - Wrappers for all operators defined in the project, decoding the values for their parameters, calling their C implementation and encoding their return values to XML-RPC;
  - A main function running the XML-RPC server with the declared operators.

  The library used to implement the XML-RPC server is the xmlrpc-c library, available on http://xmlrpc-c.sourceforge.net.
  The following rules are used for the C implementation of the operators:
  - All base types and enumerated (LITERALS) types are passed by value;
  - All complex types are passed by pointer;
  - An additional parameter is passed containing a pointer on the return value. The C function for the operator must also return this pointer.

So for example, the following SDL type:

```
NEWTYPE Point
```

```
STRUCT
  x, y INTEGER;
OPERATORS
  isOrigin: Point -> BOOLEAN;
  movePoint: Point, INTEGER, INTEGER -> Point;
ENDNEWTYPE;
```

will be translated to:

```
typedef struct
{
int x;
int y;
} Point;
BOOLEAN * isOrigin(Point *, BOOLEAN *);
Point * movePoint(Point *, int, int, Point *);
```

An implementation for these operators can be:

```
BOOLEAN * isOrigin(Point * p, BOOLEAN * result)
{
result = (p->x == 0 && p->y == 0);
return result;
}
Point * movePoint(Point * p, int dx, int dy, Point * result)
{
result->x = p->x + dx;
result->y = p->y + dy;
return result;
}
```

14.7 - rtdsShell: RTDS command line interface

This command allows a minimal set of operations on a RTDS project:

- Navigating in the project tree;
- Renaming nodes in the project tree and moving their files;
- Renaming partitions in diagrams, store them in external files or in the diagram file.

The syntax for the command is:

```
rtdsShell [ -i <command file> | -c <command(s)> ] <project file>
```

By default, commands are read from the calling terminal. With the `-i` option, commands are read from the specified file; with the `-c` option, the commands are read from the command line.

The available commands are:

- `lsnode`
  Lists the available nodes in the current context. For each node is displayed its name, its type ([D]iagram, [P]ackage, [S]ource file, [E]xternal file, [C]ode coverage results) and its file name if any.
- `pwnode`
  Print current (working) node. The whole path to the node is displayed, with node names separated with '/'.

- `chnode <node name>`
  Changes the current node to the one with the given name, which must exist in the current context. There is no way to specify a full node path in this command.
- `mvnode [ -f <new node file> ] <node name> [ <new node name> ]`
  Renames the node with the given name and/or moves its file. If `<new node name>` is specified, the node is renamed; if `-f <new node file>` is specified, the node file is moved to this file.
- `lspart`
  Lists partitions in the current node, which must be a diagram. For each partition is displayed its index, its name and its file if any.
- `mvpart [ -f <new part. file> | -d ] <part. index> [ <new part. name> ]`
  Renames the partition at the specified partition index and/or moves it to a separate file or stores it in the diagram file. If `<new part. name>` is specified, the partition is renamed; if `-f <new part.file>` is specified, the partition is stored in this file; if `-d` is specified, the partition is stored in the diagram file.
- `help`
  Displays a short help on all commands.
- `exit`
  Exit from the shell. The shell will also exit immediatly if its input stream is closed.

Several commands can be entered on the same line separated with ';'.

Examples:

- To rename interactively the (process) node `proc` in system `sys` to `p1`, also moving its file (user input is shown in blue):

```
$ rtdsShell prj.rdp
[RTDS v3.1 shell - type 'help' for help]
> chnode sys
> lsnode
proc [D] /path/to/proc.rdd
> mvnode -f /path/to/p1.rdd proc p1
> lsnode
p1 [D] /path/to/p1.rdd
```

- To rename the first partition in the system `sys` at project top-level via the command line:

```
rtdsShell -c 'chnode sys; mvpart 1 "First partition"' prj.rdp
```

14.8 - rtdsSimulate

This command runs a script on the SDL Z.100 simulator or the SDL-RT debugger. The syntaxe is:

```
rtdsSimulate.py [-c] [-q] [-l <log file>] [-g <profile name>] [-t <default target>]
<project file> <diagram node name> <script file>

  -q: quiet, no print on stdout but errors

  -c: ignore non-critical syntax/semantics errors before generation
```

<profile name> is required only for SDL-RT projects

This command will load the <project file> project, select the <diagram node name> diagram, start the Simulator if Z.100 or the SDL-RT debugger if SDL-RT and run the <script file> scenario. The <script file> is a simple text file containing Simulator or SDL-RT debugger shell commands.

Command example:

```
rtdsSimulate.py H:\Z.100\PingPong.rdp mySystem H\Z.100\myScript.sce
```

Script file example:

```
startMscTrace
step
step
keySdlStep
keySdlStep
keySdlStep
keySdlStep
saveMscTrace c:\tmp\MyMscTrace.rdd
stopMscTrace
```

14.9 - rtdsObjectServer: RTDS API server

This command runs the CORBA server allowing to use the RTDS standard API. For more details, see paragraph “Model browsing API” on page 316.

14.10 - rtdsSearch: Low-level search utility

This command is not intended to be used by end-users, but may be called from RTDS. See “Environment variables” on page 283 for its usage. Issuing the command in a terminal will not produce any user-readable results.

14.11 - rtdsDiagramDiff: RTDS diagram diff utility

This utility allows to compute the differences between two diagrams on the command line. The syntax for the command is:

```
rtdsDiagramDiff [-q|--quiet] [--grahical] <diagram file 1> <diagram file 2>
```

The options are:

- `-q` or `--quiet`: No output is produced. Typically used to know if two diagrams are logically equivalent or not via the command return code.
- `--graphical`: By default, only the logical differences between the two diagrams are analysed and/or reported. Specifying this option will analyse and report graphical differences as well.

The return code for the command is zero if no difference could be found. A non-zero return code indicates that the diagrams are different, or that an error occured.

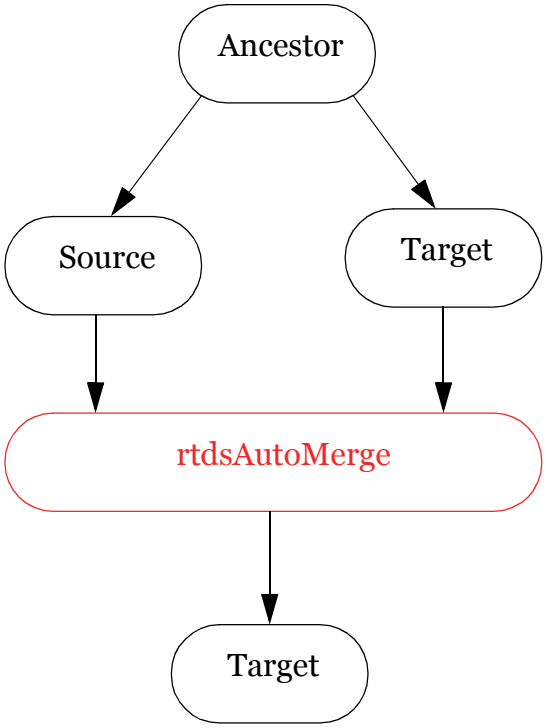
14.12 - rtdsAutoMerge: RTDS diagram merge utility

This utility is meant to be used with configuration management systems. This utility merges automatically the modifications made in the source version and in the target version in reference to their common ancestor diagram. The resulting diagram is saved in the target diagram.

The syntax for the command is:

```
rtdsAutoMerge <common ancestor diagram> <source diagram> <target diagram>
```

If successful the return code is zero. If a conflict is detected, a non zero error code is returned and a description of the conflict is printed on stderr.



15 - Syntax & semantics check

The following table is a list of identifiers for warnings that can be returned by RTDS syntax and semantics checks. These identifiers may be used to ignore some warning types via the preferences, "Diagrams" tab. A warning identifier always consists in the fiollowing parts:

- A single letter identifying the object type for the warning. This letter may be 'C' for a link to symbol connector, 'D' for a diagram, 'E' for any element, 'L' for a link and 'S' for a symbol.
- Three letters identifying the actual type for the object. For example, 'LMTH' is a method link in a MSC diagram; 'SBCI' is a block class instance symbol.
- Three digits identify the warning itself. This is just a sequence number and has no meaning.

Please note that warning can only be ignored if they are actually warnings. Some of these warnings are actually reported as errors when the checking mode is not "critical only". In such a case, the messages cannot be ignored.

Identifier	Meaning
CASS001	Syntax warning for roles in a UML association connector.
CCHN001	Undeclared signal in a channel connector.
CCHN002	Undeclared signal list in a channel connector.
CCHN003	No gate name for a channel connector on an agent class instance.
CCHN004	Unknown gate name for a channel connector on an agent class instance.
CCHN005	Unknown channel name for a connection name in a channel connector.
CCHN006	Signal sent from a channel connector, but not from the attached agent.
CCHN007	Signal received by a channel connector, but not by the attached agent.
DCST001	Signal received by a composite state, but not by the parent process.
DCST002	Signal sent by a composite state, but not by the parent process.
DPRO001	Dead state in a process, procedure or service diagram.
DPRO002	Invalid symbol ending a transition in a process, procedure or service diagram.
DPRO003	Missing object initializer in a process diagram.
ESEQ001	Symbol sequence problem in a diagram.
LMSG001	Syntax error in a message link in a MSC diagram.
LMSG002	Message link to a semaphore in a MSC diagram.
LMSG003	Incorrect received message name for a lifeline in a MSC diagram.

Identifier	Meaning
LMTH001	Syntax error in a method call link in a MSC diagram.
LMTH002	Semaphore calling a method in a MSC diagram.
LMTH003	Incorrect method call on a semaphore in a MSC diagram.
LMTH004	Incorrect method call on an object lifeline in a MSC diagram.
LSTX001	Syntax warning on a link.
SBCI001	Missing gate name on channel connected to a block class instance symbol.
SBCI002	Unknown gate name on channel connected to a block class instance symbol.
SBCI003	Incorrect sent or received message for gate on channel connected to a block class instance symbol.
SCLA001	General purpose warning for attributes in a class symbol in a class diagram.
SCLS001	Unknown stereotype in a class symbol.
SCLS002	Unknown property in a class symbol.
SCLS003	Consistency problem between diagrams for an active or passive class.
SCMP001	Stereotype specified for a component symbol in a deployment diagram.
SCMP002	Properties specified for a component symbol in a deployment diagram.
SCMP003	Operations specified for a component symbol in a deployment diagram.
SDCL001	Signal declared, but never used.
SDCL002	Signal declared more than once.
SFIL001	Special characters used for file name in a file symbol in a deployment diagram.
SINP001	Timer received, but never started in current context.
SINP002	Undeclared signal in input symbol.
SINP003	Signal in an input symbol not in any incoming channel for agent, but possibly received from self.
SINP004	Signal in an input symbol not in any incoming channel for agent that cannot be received from self.
SINP005	Unsupported use of VIRTUAL, REDEFINED or FINALIZED in an input symbol.
SLBL001	Label defined, but never used.

Identifier	Meaning
SLLI001	Lifeline for non-existent semaphore in a MSC diagram.
SLLI002	Lifeline for non-existent agent or object in a MSC diagram.
SNOD001	Stereotype specified for a node symbol in a deployment diagram.
SNOD002	Properties specified for a node symbol in a deployment diagram.
SNOD003	Package specified for a node symbol in a deployment diagram.
SNOD004	Operations specified for a node symbol in a deployment diagram.
SOUT001	Explicit sending of a timer signal.
SOUT002	Undeclared signal sent.
SOUT003	Signal in an output symbol not in any outgoing channel for agent, but possibly sent to self.
SOUT004	Signal in an output symbol not in any outgoing channel for agent that cannot be sent to self.
SOUT005	Unknown process name after `TO_NAME` in output.
SPCI001	Missing gate name on channel connected to a process class instance symbol.
SPCI002	Unknown gate name on channel connected to a process class instance symbol.
SPCI003	Incorrect sent or received message for gate on channel connected to a process class instance symbol.
SPRC001	Unknown procedure in a procedure call symbol.
SPRC002	C procedure called via a procedure call symbol.
SSAV001	Timer saved, but never started in current context.
SSAV002	Undeclared signal saved.
SSAV003	Signal in a save symbol not in any incoming channel for agent, but possibly received from self.
SSAV004	Signal in a save symbol not in any incoming channel for agent that cannot be received from self.
SSMD001	Semaphore declared, but never used.
SSMG001	Undeclared semaphore given.
SSMT001	Undeclared semaphore taken.
SSTA001	State used as `NEXTSTATE`, but with no transition defined.
SSTA002	Transitions defined for state, but no never used as a `NEXTSTATE`.

Identifier	Meaning
SSTA003	Inconsistency between state type ("normal" / composite) and state symbol type.
SSTR001	Unsupported use of VIRTUAL, REDEFINED or FINALIZED in a start symbol.
SSTX001	Syntax warning in a symbol. Used only for Geode-specific syntax.
STIC001	Non-timer signal used in a timer cancel symbol.
STIC002	Timer cancelled, but never started in current context.
STIC003	Timer cancelled, but never received anywhere in the system.
STIC004	Timer cancelled, but never received in the current context.
STIS001	Non-timer signal used in a timer start symbol.
STIS002	Timer started, but never received anywhere in the system.
STIS003	Timer started, but never received in the current context.

16 - XMI Import

When importing an XMI file, a default package is always created.

16.1 - Diagrams supported

UML diagrams	Concept supported
Class	Inheritance, Association, Composition, Aggregation, Package, Attribute, Operation
Structural	Port, Message, Link, Contract, Package
Use case	Actor, Scenario
Sequence Diagram	All concepts
State chart	All concepts with some transformation

Table 44: UML diagrams supported

16.2 - XMI version

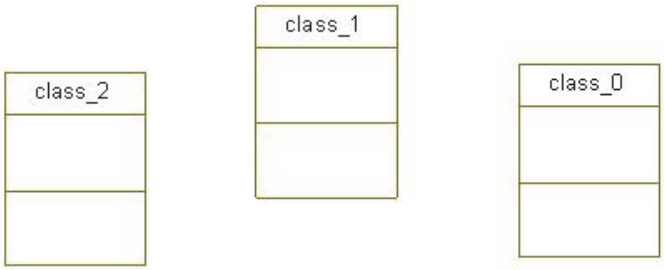
The XMI import is based on version 2.1 of the XMI standard. It has been tested with files exported from different tools.

16.3 - Class diagram

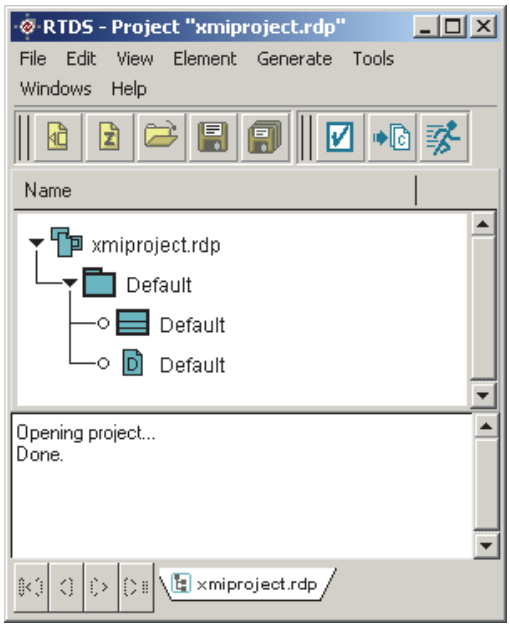
An imported Class diagram in RTDS is represented with a symbol in the tree of the project manager. This symbol represents the class diagram and contains all classes of this view.

16.3.1 Structure of a Class diagram

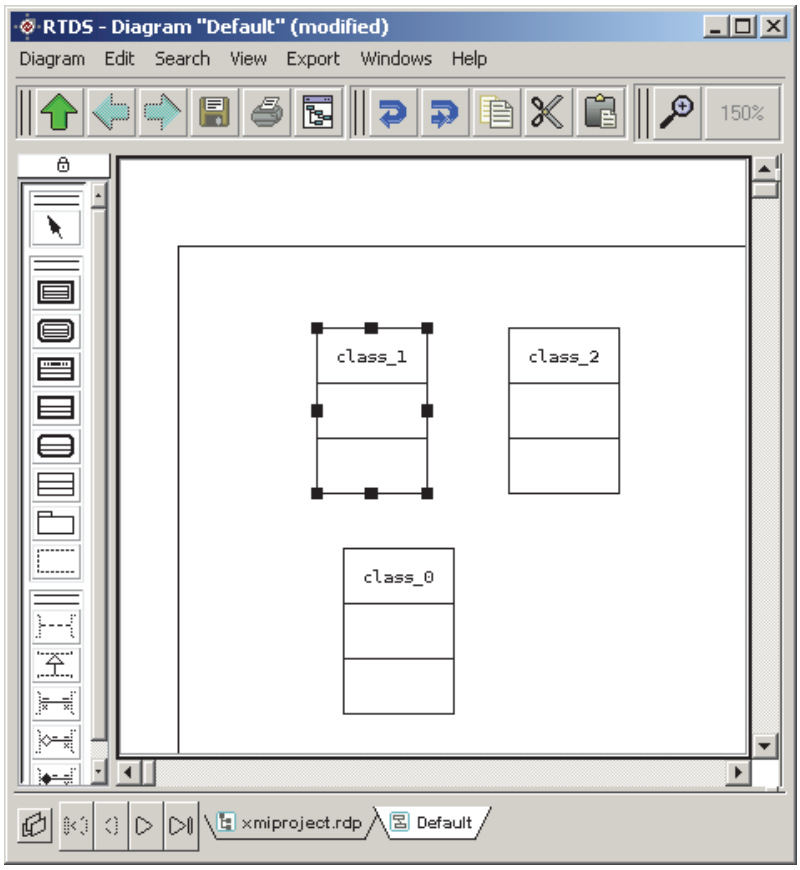
An imported class diagram is always in a package. If a class has no parent package, the class is imported in a default package named "Default". The following class diagram :



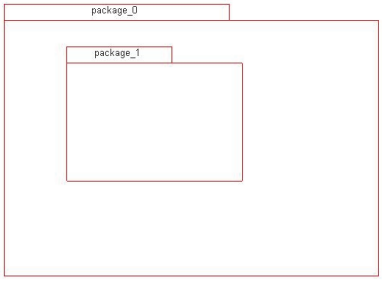
will be imported in RTDS as follow :



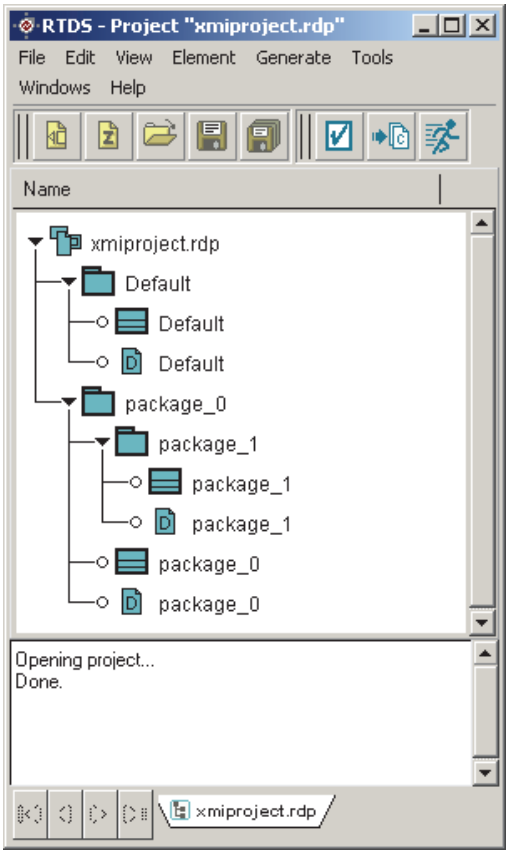
And the "Default" class diagram will contain the classes "class_0", "class_1" and "class_2" :



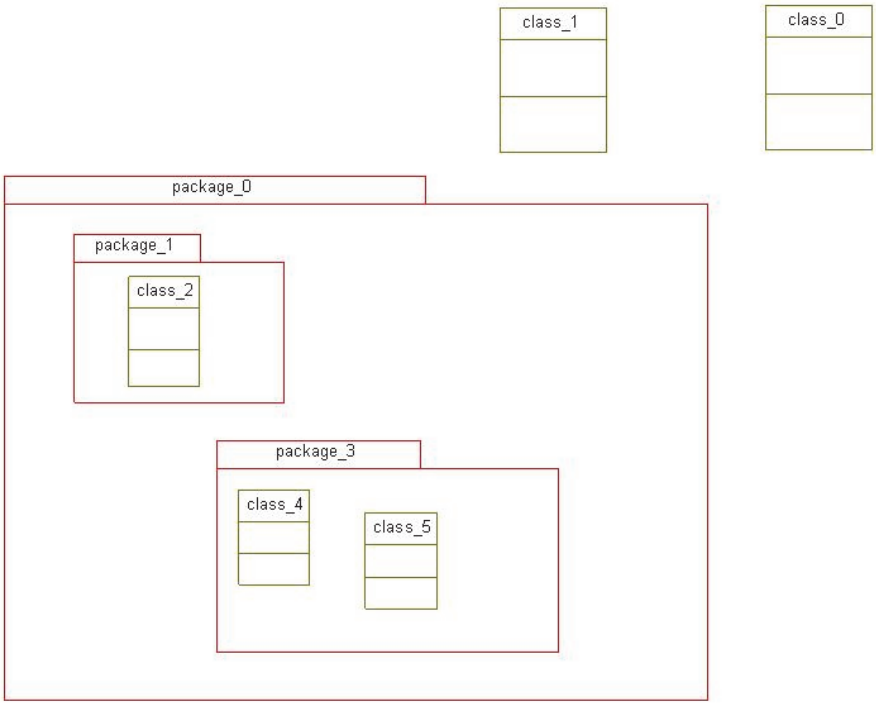
If the class diagram is contained in a package. The package will be represented in the tree of the project manager of RTDS and the class diagram will have the same name as the one of the package. For example, the following class diagram which contains a package :



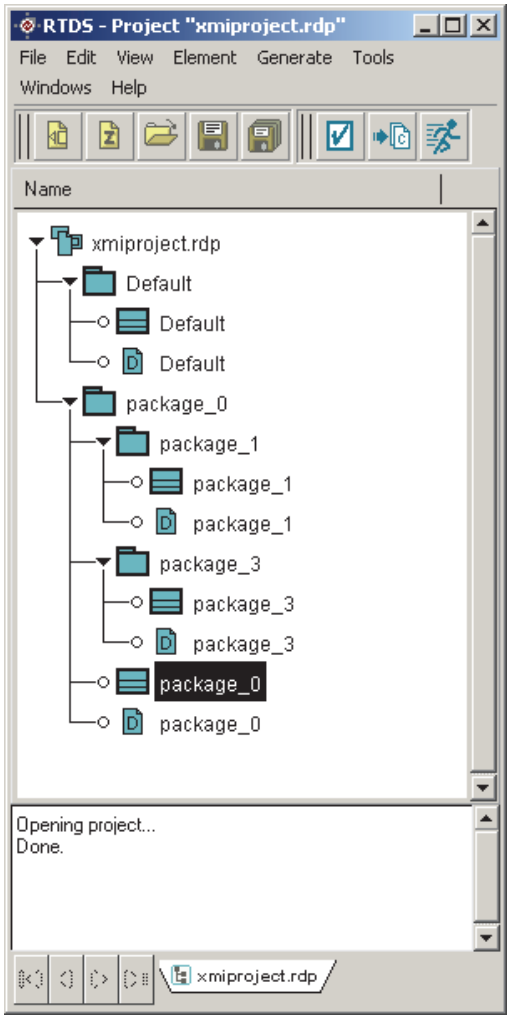
will be imported in RTDS as follow :



No classes are defined in the XMI packages. Thus, the imported class diagrams will be empty. Note that when a class diagram is imported in RTDS; the packages will only contain the classes they define. This is illustrated in the following example:



After importing this class diagram in RTDS, the project manager will show the following tree :



The class diagram "package_0" will be empty because it does not contain any classes. The classes "class_0" and "class_1" will be in class diagram "Default", "class_2" in "package_1", "class_3" and class_5" in "package_3".

16.3.2 Association and direct association

All types of association are supported.

16.3.3 Aggregation and Composition

All types of aggregation or composition are supported.

16.3.4 Inheritance

All types of an inheritance are supported.

16.3.5 Generalization and Realization

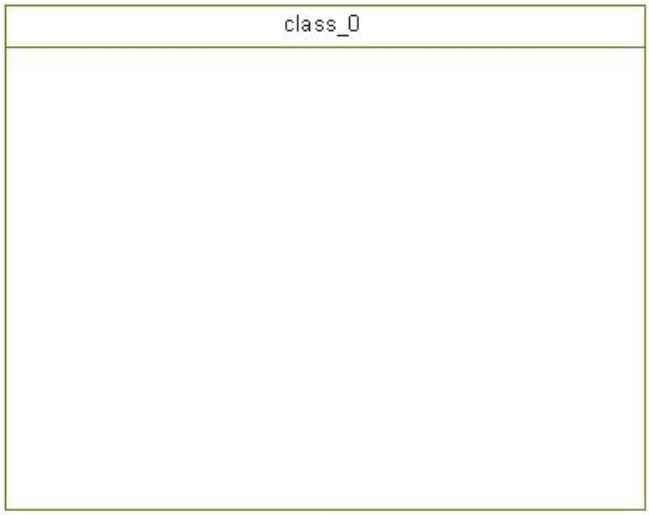
Generalization and realization are mapped into RTDS on the specialization concept.

16.4 - Structural diagram

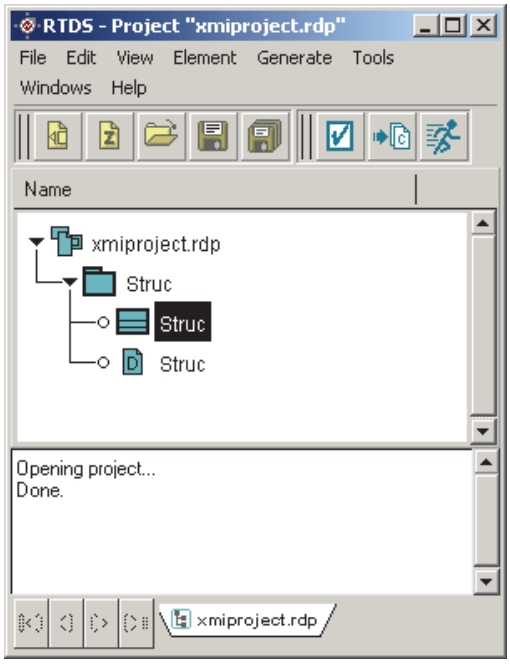
Objects and classes of a structural diagram are imported as SDL blocks and processes.

16.4.1 Structure of a Structural diagram

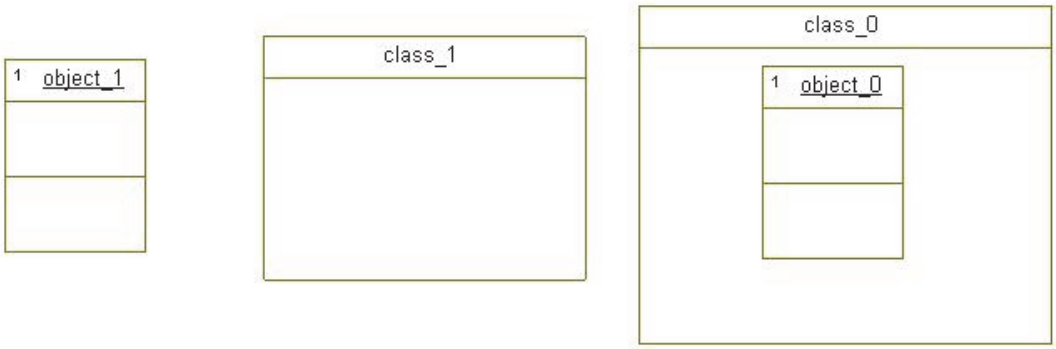
The structural diagram is always imported in a package. If a class (or an object) of a structural diagram does not contain an object, this class (or object) will be imported as a class of the default class diagram. For instance, when importing this structural diagram :



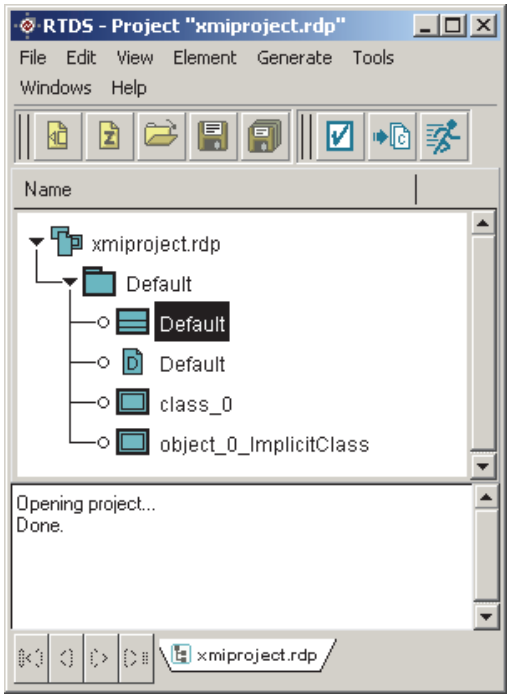
The project manager will import the structural diagram as a class diagram with the following structure:



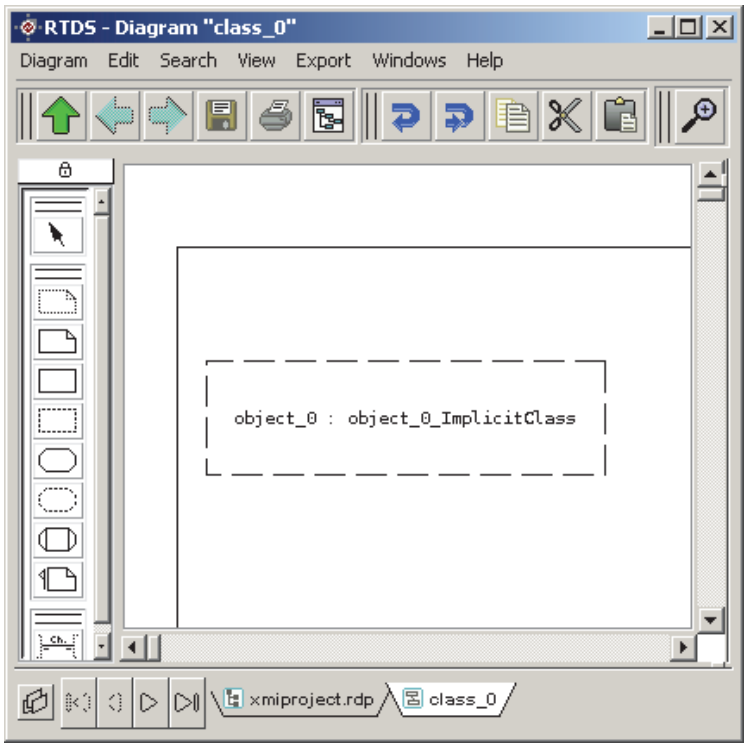
and the class diagram "Struc" will contain the class "class_0". But if the following structural diagram is imported in RTDS :



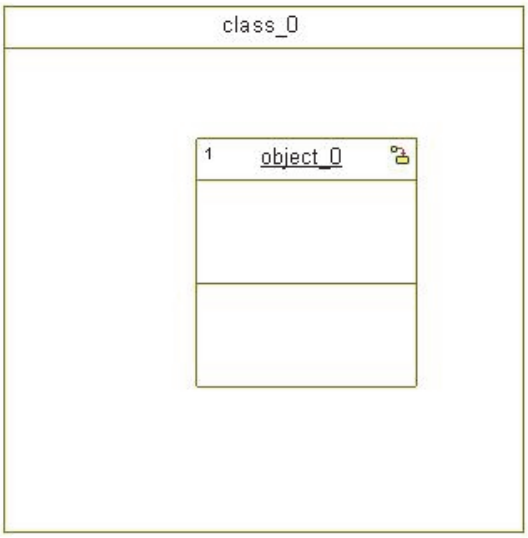
The project manager will create the following SDL project :



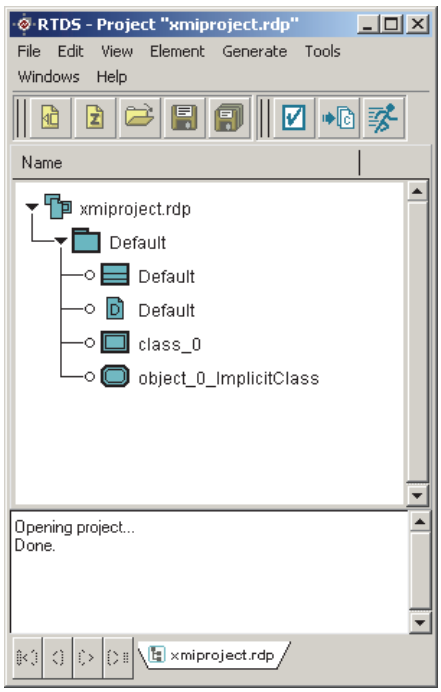
Because "object_1" and class_1" are not in an class and are not composed of objects or classes; RTDS has imported them like classes of a class diagram. But the "class_0" is composed of an instance of the class "object_0_ImplicitClass" named "object_0". RTDS has created two classes of block named "class_0" and "object_0_ImplicitClass". There is the corresponding diagram of the block "class_0" in RTDS :



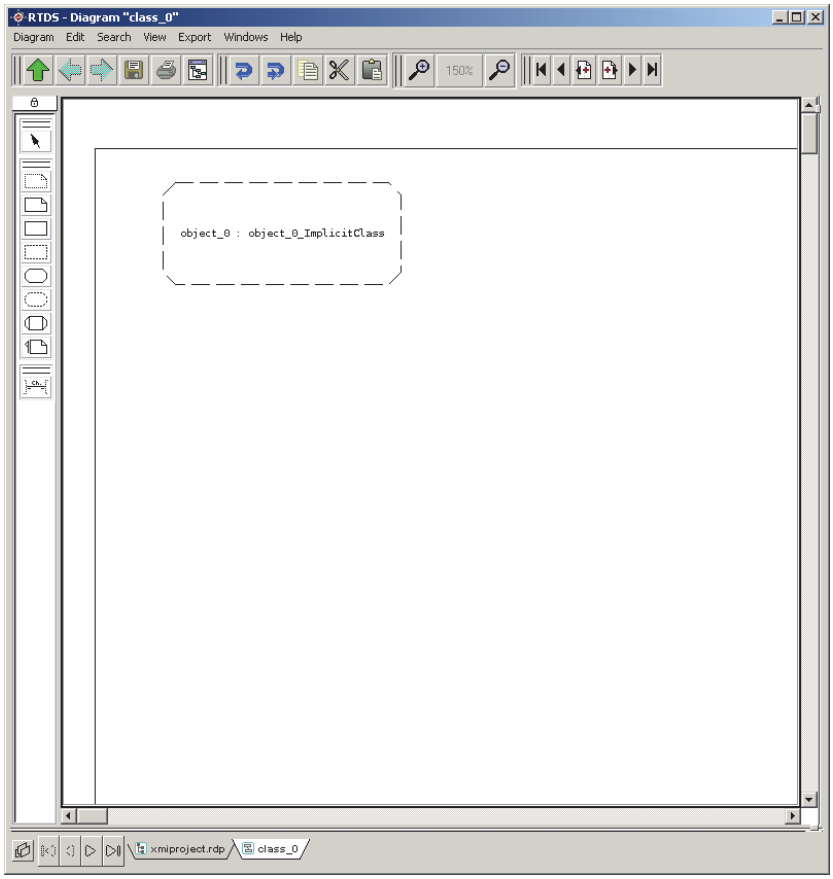
When importing a structural diagram (a class which contains a component), a class of block is always the element for mapping a passive class. When a state machine is defined for a object, RTDS will create a class of process with is corresponding state machine. This is illustrated in the following example:



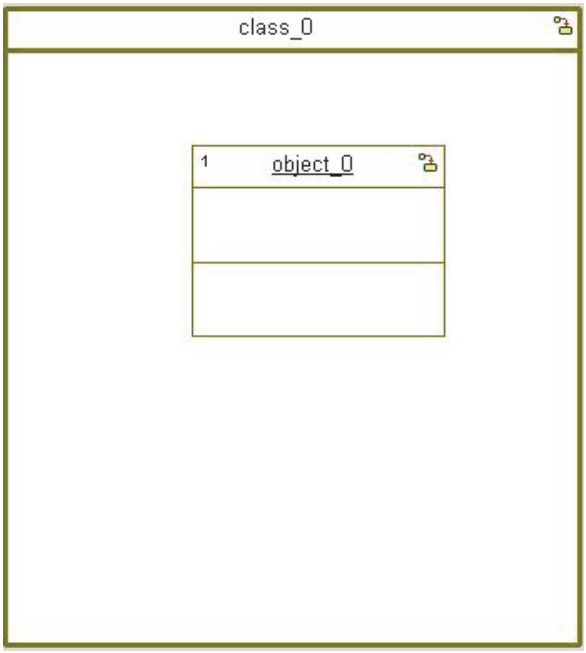
When importing the corresponding XMI file of this model, the project manager creates classes of block and process :



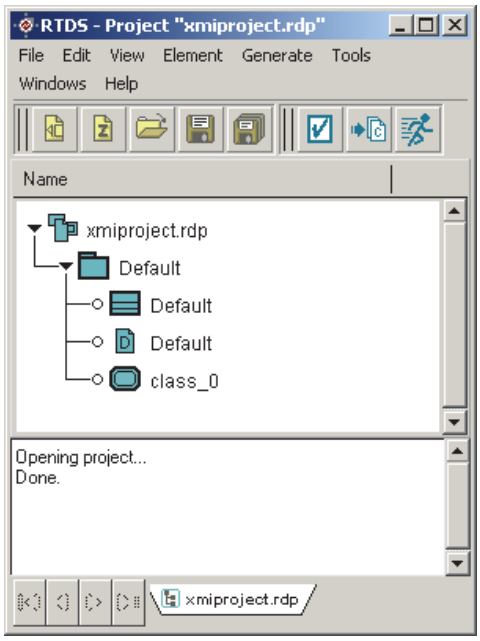
The diagram for "class_0" is:



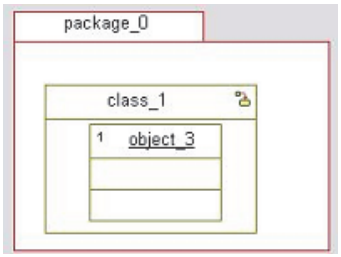
Note that it is not possible into RTDS, to define a state machine for a block. Thus if RTDS tries to import this structural model :



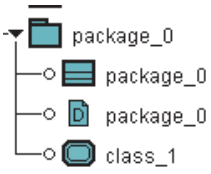
RTDS will not import the "object_0" component. The resulting project will be :



And finally, because a package can also contain a structural diagram if the following diagram is imported in RTDS :



The resulting project will be :



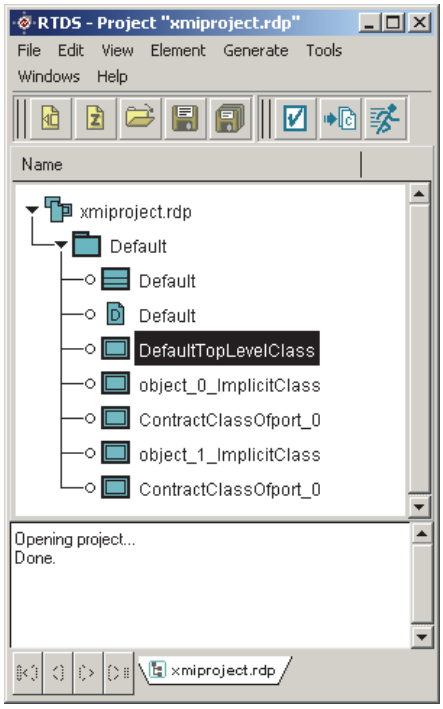
16.5 - Communication

The components of a structural diagram can communicate through ports, interfaces and links. Ports are mapped on SDL gates. Interfaces are mapped on classes of block (process) with their associated operations for sending and receiving messages. And links are mapped on SDL channels.

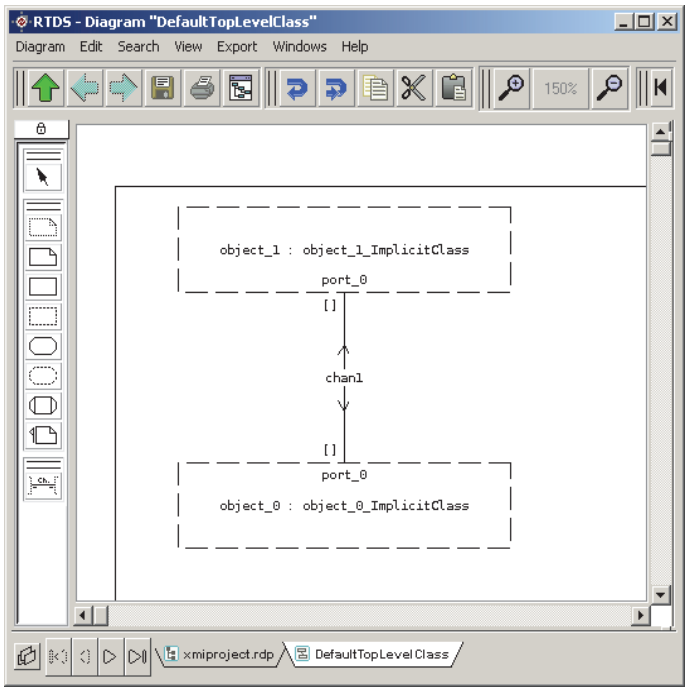
16.5.1 Links and ports

Components of a structural diagram send and receive messages through ports which are connected via links. For instance, if the following structural diagram is imported into RTDS :

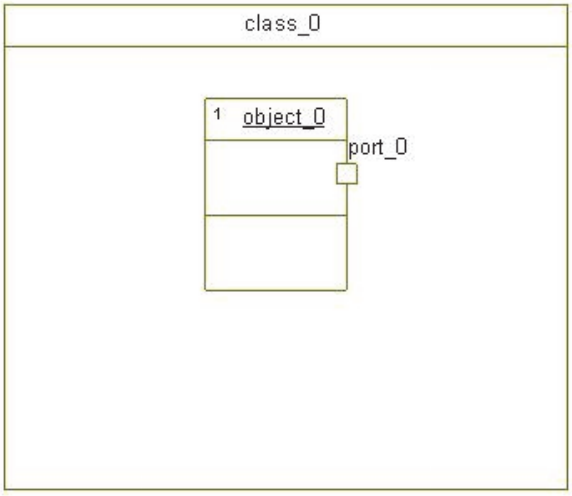
The corresponding RTDS SDL-RT project will be :



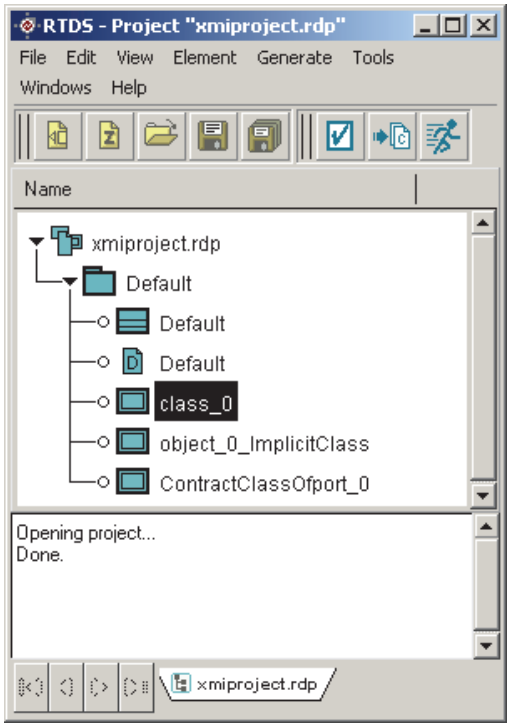
The block "DefaultTopLevelClass" is composed of two class instances. The first "object_1" is an instance of the class of block "object_1_ImplicitClasse" and the second "object_0" is an instance of the class of block "object_0_ImplicitClass" :



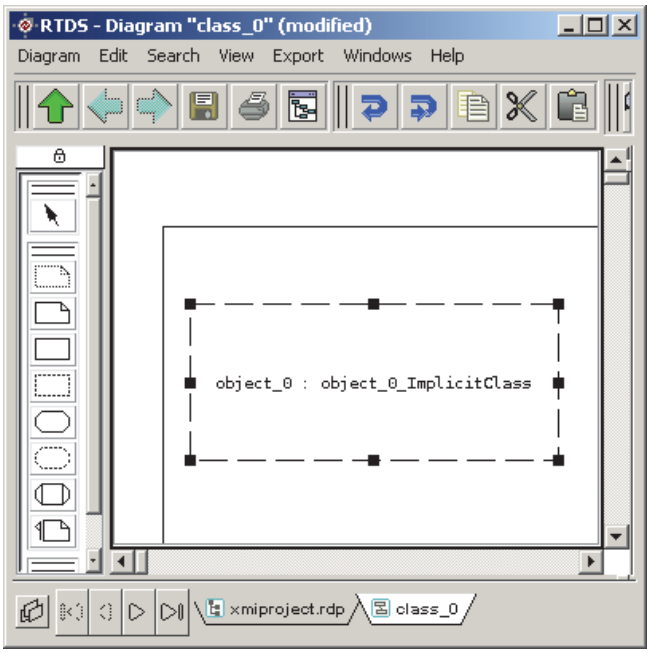
The ports of each component are mapped on SDL gates. The link "chan1" between the components is mapped on an SDL channel. Note that it is not possible to define a gate without a channel in RTDS. Thus if the following structural diagram is imported :



The port "port_0" which is not linked to another port will not be imported and the corresponding RTDS SDL-RT project will be :



And the diagram of "class_0" does not contain any reference to port "port_0" :

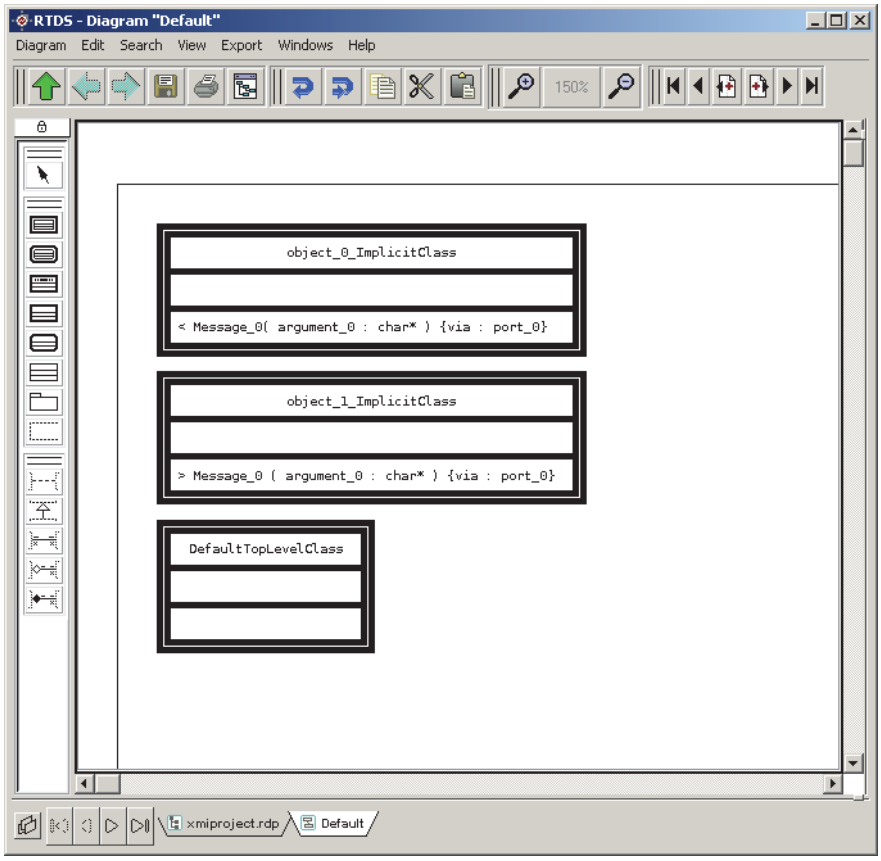


16.5.2 Interfaces and messages

The messages received and send by a component can be declared by defining the contract of this component. If a contract is defined for a component, RTDS will generated a inter-

face for it. This interface is defined in a class diagram and defines all messages received and sent by a specific gate. For example :

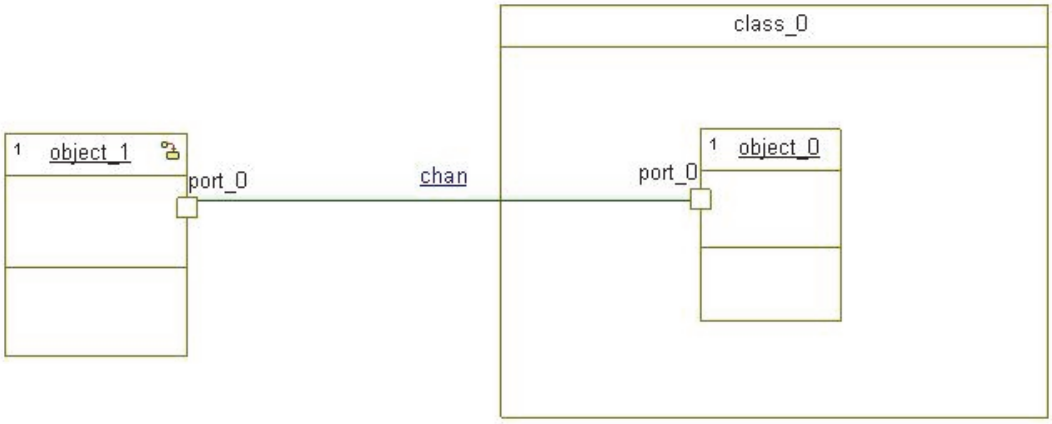
After importing the XMI file into RTDS, the default class diagram will be :



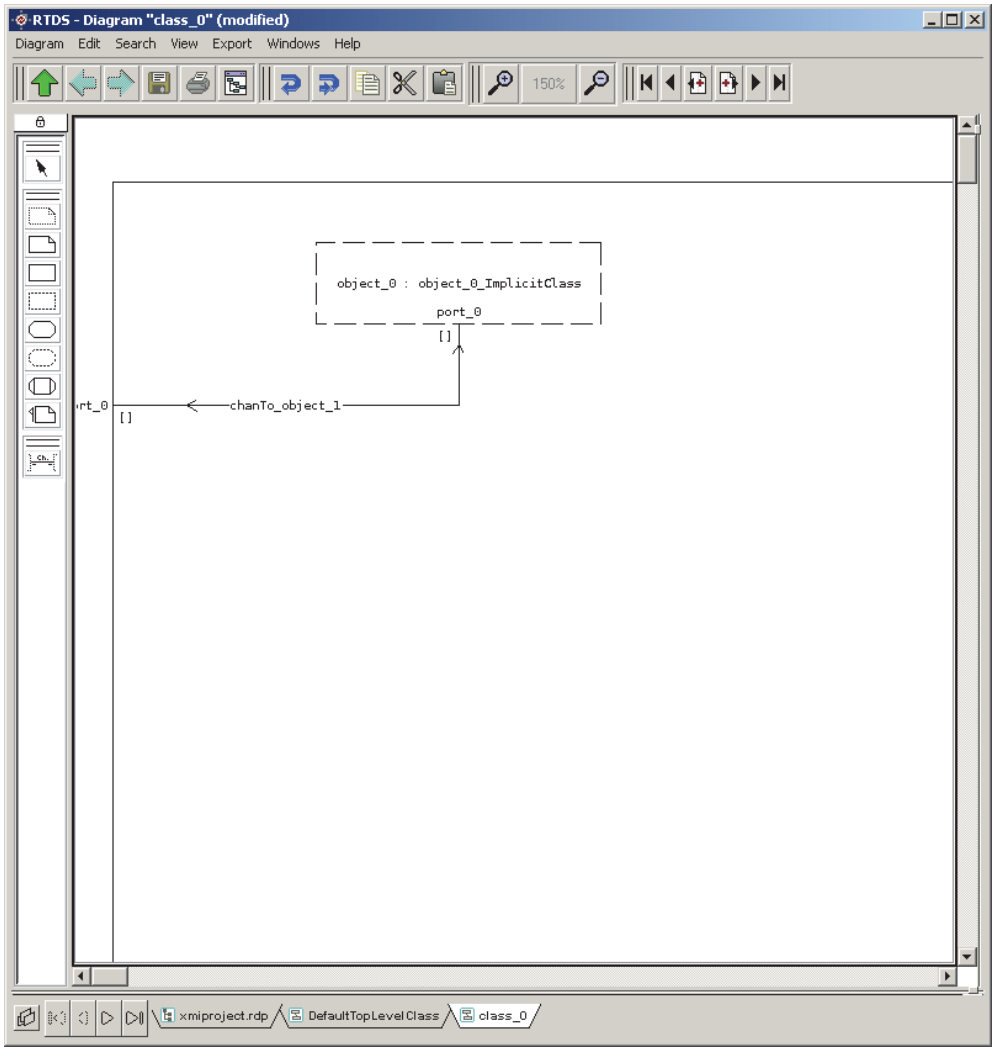
The concept of a UML message is "Primitive operation" when defining the contract of the component.

16.5.3 Channel

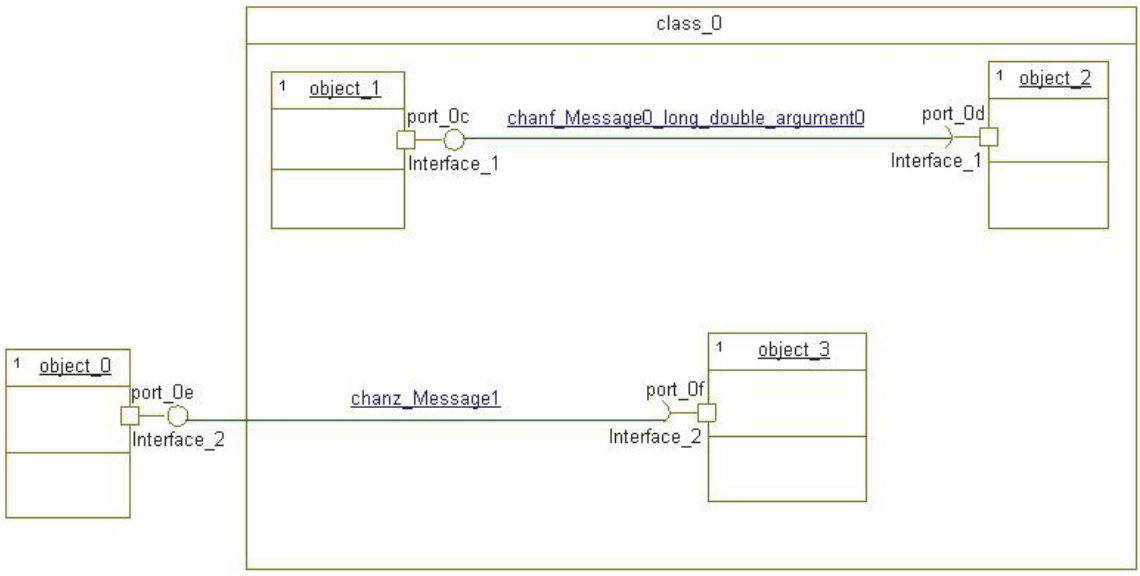
RTDS supports links between ports. A channel is created for each link of communication. If a component in a class communicates with a external object, intermediate ports are created. For example:



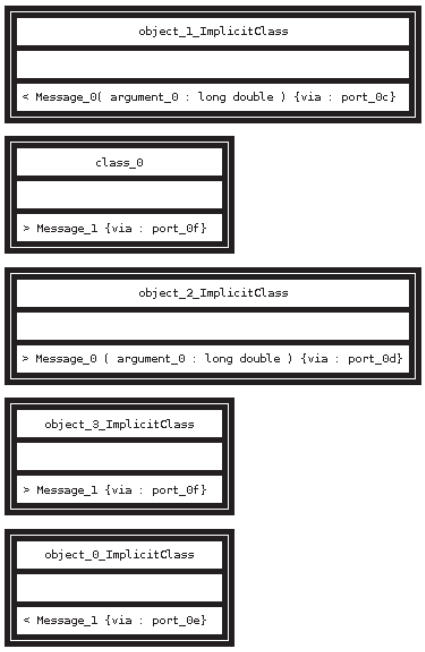
Since the "class_0" does not have a port. RTDS will create a port named "port_0" on it and a channel of communication between "object_0" and "class_0":



Moreover, if a UML component of a structural diagram has a contract and communicates with an external object:



RTDS will create an intermediate link between "object_3" and "class_0" but it will also add in the interface of "class_0" a contract which is represented by the message sent and received :



17 - Model browsing API

RTDS model browsing API (a.k.a RTDS "object server") allows to browse all information contained in a RTDS project in a structured way. More precisely, this API allows to:

- Open a project file;
- Browse all agents, classes, ... defined in the project;
- Decode all declarations made in these objects;
- Get a description of all transitions appearing in diagrams.

17.1 - General principles

17.1.1 Architecture

RTDS object server is defined as a CORBA server. This allows to write clients in any language on any platform. The interface itself is described in the IDL file `rtdsObjectServer.idl`, located in the directory `$RTDS_HOME/share/object_server`. The main class for the API is `ObjectServer` (described last in the IDL file). An instance of this class is automatically created by the server when it starts, and a textual form of its object identifier is stored in a file. The client should get this identifier from the file and ask the ORB to connect to the corresponding object. The `ObjectServer` class is described in detail in paragraph “Class ObjectServer” on page 327.

The file to which the `ObjectServer` identifier is written must be passed to the server via the `-f` option when it’s started, as in:

```
rtdsObjectServer -f /path/to/rtdsObjectServer.ref
```

A typical client connecting to this server - written in C and using the ORBit CORBA ORB - would be:

```
#include <orb/orbit.h>

#include "rtdsObjectServer.h"

#define ORB_INIT_ERROR  1
#define ORB_SHUTDOWN_ERROR  2
#define NO_SERVER_REF_FILE  3
#define NO_SERVER           4
#define INVALID_SERVER_REF  5

int main(int argc, char * argv[])
{
  CORBA_ORB                       orb;
  CORBA_Environment               ev[1];
  FILE                          * serverRefFile;
  char                            serverRef[1024];
  rtdsObjectServer_ObjectServer   server;

  /* Initialize the CORBA environment */
  CORBA_exception_init(ev);
```

```
  orb = CORBA_ORB_init(&argc, argv, "orbit-local-orb", ev);
  if ( ev->_major != CORBA_NO_EXCEPTION) return ORB_INIT_ERROR;

  /* Get textual form for server reference from file created by server */
  serverRefFile = fopen("/path/to/rtdsObjectServer.ref", "r");
  if ( serverRefFile == NULL ) return NO_SERVER_REF_FILE;
  fgets(serverRef, 1024, serverRefFile);
  fclose(serverRefFile);

  /* Actually connect to server */
  server = (rtdsObjectServer_ObjectServer)CORBA_ORB_string_to_object(
              orb, serverRef, ev);
  if ( ev->_major != CORBA_NO_EXCEPTION) return INVALID_SERVER_REF;
  if (server == NULL) return NO_SERVER;

  /* Test that we've got is actually a RTDS ObjectServer
   * with the correct version
   */
  if ( ! CORBA_Object_is_a(server,
            "IDL:pragmadev.com/rtdsObjectServer/ObjectServer:1.0") )
    return INVALID_SERVER_REF;

  /* Client specific code ... */

  /* Disconnect from server */
  CORBA_Object_release(server, ev);
  if ( ev->_major != CORBA_NO_EXCEPTION) return ORB_SHUTDOWN_ERROR;
  CORBA_ORB_shutdown(orb, CORBA_TRUE, ev);
  if ( ev->_major != CORBA_NO_EXCEPTION) return ORB_SHUTDOWN_ERROR;
  return 0;
}
```

Notes:

- The `rtdsObjectServer` executable is installed in `$RTDS_HOME/bin`, so it should be directly available on the command line if all instructions in RTDS installation manual have been followed.
- All clients must test if the server is the correct one in the correct version via the instructions:
  ```
  if ( ! CORBA_Object_is_a(server,
           "IDL:pragmadev.com/rtdsObjectServer/Object-
  Server:1.0") )
      /* ... */;
  ```
  Not testing this condition may cause crashes or produce unpredictable results.

17.1.2 Organization

In addition to the `ObjectServer` class described above, the main classes for the model browsing API are the following :

- `Project`: as in RTDS, every piece of information accessible via the API must live in a project. Instances of this class represent projects as they are defined and used in RTDS.

- `Item`: this class is the super-class for almost every object accessible via the API (except for a few; see below). An item is an element is the logical architecture described in a project. It can be:
  - an agent (system, block, process, block class, process class);
  - a package;
  - a procedure;
  - a macro;
  - a state, which can be composite;
  - a concurrent state machine, a.k.a service;
  - a type, defined via `NEWTYPE` or `SYNTYPE`;
  - a variable, including process, procedure or macro parameters and synonyms;
  - a connector in a state machine (label);
  - a static class defined in a UML class diagram;
  - a timer;
  - a semaphore.

  Items are organized into a tree, and provide an access for most information attached to the corresponding objects:
  - Channels, connections and gates in systems, blocks and composite states;
  - Incoming and outgoing signals for agents, procedures and services;
  - Transitions for processes, procedures, services and macros.
- `Element`: as items define the logical architecture of a project, elements define the physical architecture of the project. An element can be:
  - a diagram;
  - a symbol in a diagram;
  - a source file.

  Several elements may be attached to any item:
  - The element describing the item, e.g the process diagram for a process item;
  - The elements defining the item, e.g the process symbol in the process's parent block for a block item;
  - The elements using the item, e.g a process dynamic creation symbol in a transition for a process item.

  If an element is a symbol or a source file, all parts of its text are accessible through the element's syntax tree, allowing to browse the whole contents of the model. For a symbol describing an action in a transition, it is also possible to get the symbol(s) following it.

Several other classes are used in the API:

- `Signal` and `SignalList` are used to describe signals and signal lists defined in the project.
- `GlobalDataManager` is used to access all objects that are not managed in the item tree, but globally in the project. This includes signals, signal lists and semaphores.
- `Channel` is used to describe channels in the architecture.
- `SignalWindow` represent a connection point between a channel and an agent. It is also used to represent gates, either in architecture diagrams or defined for agent classes in UML class diagrams.
- For static or active class items, the classes `Attribute`, `Operation` and `Role` are used to describe all their declared attributes, operations and roles played in

associations respectively. A class named `Association` describes the associations themselves.

All these classes are described in detail in the next section.

17.2 - Interface detailed description

The following paragraphs are an overview of all features on the classes that can be accessed through RTDS objet server. For a really detailled description, please refer to the IDL file `rtdsObjectServer.idl`, located in the directory `$RTDS_HOME/share/object_server`.

17.2.1 Class Agent

This class is a sub-class of `Class` (cf. “Class Class” on page 322). Its instances describe agents in the project, i.e. the system, the blocks and the processes. So items having the types `SYSTEM_TYPE`, `BLOCK_TYPE` and `PROCESS_TYPE` are actually instances of `Agent`.

Among the features inherited from `Item`:

- Operations dealing with signal windows always work;
- Operations dealing with contained channels only work for systems and blocks;
- Operations dealing with contained transitions only work for processes.

Among the features inherited from `Class`, roles will be defined for all associations from this agent to static classes in UML class diagrams.

In addition to the features inherited from `Item` and `Class`, agents know:

- The minimum (initial) and maximum number of instances declared for the agent. If not specified or not applicable, the minimum number is set to the string `"1"` and the maximum is empty.
- The class for the agent if the agent is an instance of an agent class. For example:

```
b : MyBlockClass
```

  declares the (block) `Agent` `b` as being an instance of the (block class) `AgentClass` `MyBlockClass`. So there’s two items referenced by this symbol:
  - The `AgentClass` `MyBlockClass` is instantiated by the symbol and will have it in its `usingSymbols`;
  - The `Agent` `b` is defined by the symbol and will have it in its `definingSymbols`.

Note: it may be surprising that `Agent` inherits from `Class`, as agents directly in the system architecture are usually said to be instances in SDL systems, with the term class representing only block or process classes defined in packages. As we use them, the terms "class", "instance" and "object" must actually be taken as the common terms used in the object-oriented paradigm: an object is what physically appears in a running system, with associated memory used to contain the object’s attribute values. A class is a template used to build objects, also named instances of the class. As a consequence of these definitions, models never describe objects or instances, as it would be pointless to describe what actually appears in the memory associated to such an object. What a model describe is always the "template" used to create such objects in the running system, i.e the class.

17.2.2 Class AgentClass

This class is a sub-class of `Class` (cf. “Class Class” on page 322). Its instances describe agent classes in the project, i.e. block classes and process classes. So items having the types `BLOCK_CLASS_TYPE` and `PROCESS_CLASS_TYPE` are actually instances of `AgentClass`.

Among the features inherited from `Item`:

- Operations dealing with signal windows always work;
- Operations dealing with contained channels only work for block classes;
- Operations dealing with contained transitions only work for process classes.

Among the features inherited from `Class`:

- The operations will contain all signals declared as sent or received by the agent class in UML class diagrams, with the visibilities `SIGNAL_OUT_VISIBILITY` and `SIGNAL_IN_VISIBILITY` respectively;
- Roles will be defined by all associations from this agent class to static classes in UML class diagrams.

In addition to features inherited from `Item` and `Class`, agent classes record their declared instances as a list of `Agent` items.

17.2.3 Class Association

Instances of this class represent associations in UML class diagrams. An association knows:

- Its type ("regular" association, aggregation, composition);
- Its name;
- The role where it starts and the role where it ends (instances of `Role`; cf. “Class Role” on page 329);
- Whether the association name should be read from the declared starting role to the declared ending role or the reverse.

Note: specialization links in class diagrams are not represented as instances of `Association`. To access super-classes for a given class, use the attribute `superClasses` on the instance of `Class` describing it (cf. “Class Class” on page 322).

17.2.4 Class Attribute

An instance of this class describes an attribute declared in a class symbol in a UML class diagram. Its attributes are all information that may be extracted from the UML attribute declaration:

- Visibility (public, protected, private);
- Name;
- Multiplicity (as `12` in "`myIntArray[12] : int`");
- Type;
- Default value (as `0.0` in "`myFloat : float = 0.0`");
- Properties (as `read-only` in "`myAttr : int {read-only}`").

17.2.5 Class Channel

An instance of this class represent a channel in an architecture diagram. A channel knows:

- Its name;
- The signal window where it starts and the signal window where it ends (instances of `SignalWindow`; cf. “Class SignalWindow” on page 330).

17.2.6 Class Class

This class is a sub-class of `Item` (cf. “Class Item” on page 324). Its instances are all objects in the current project that are considered as classes, i.e static classes, systems, blocks, processes, block classes and process classes (cf. note in paragraph “Class Agent” on page 320).

Among the features inherited from `Item`:

- Operations dealing with signal windows only work for active classes;
- Operations dealing with contained channels only work for systems, blocks and block classes;
- Operations dealing with contained transitions only work for process and process classes.

In addition to these features, an instance of `Class` knows:

- Its attributes as a list of instances of `Attribute` (cf. paragraph “Class Attribute” on page 321);
- Its operations as a list of instances of `Operation` (cf. paragraph “Class Operation” on page 328);
- Its direct super-classes as a list of instances of `Class`;
- The roles it plays in associations as a list of instances of `Role` (cf. paragraph “Class Role” on page 329);
- A list of descriptors for the attributes that would be created for each role. These descriptors give the attribute name, type and multiplicity. They are not created if the association is not navigable;
- If the class is a static class, it also knows the C++ source file in the project that was generated for it if any.

Note: the role and the role attributes are not considered on the same side of the associations. For example, in:

the class `C1` plays the role `r1` in association `assoc`, so `r1` will appears in the roles played by `C1`. But the attribute created for `assoc` will be for role `r2` (and will reference instances of `C2`), so the role attributes for `C1` will include an attribute named `r2`.

17.2.7 Class Element

Elements define the physical architecture of the project. An element can be a diagram, a symbol or a source file.

Features for elements are:

- The name of the file for the element. This is the diagram file name for diagrams and symbols, and the source file name for source files;
- A description of the textual contents of the element if any as a syntax tree (see below);
- A list of declarations symbols contained in the element if applicable. This is available only for diagram elements.

A syntax tree consists in the following fields:

- A type, indicating what kind of information the element contains;
- A set of attributes, depending on the type.

For example:

- For a process symbol in a block diagram, the syntax tree type will be `PROCESS_DECLARATION` and its attributes will be the process name, its minimum and maximum number of instances as written in the symbol and the process priority (for SDL-RT).
- For a timer start in a transition, the syntax tree type will be `TIMER_START` and its attributes will be the timer name and a descriptor of the expression for the timer time-out if any.
- For a declaration symbol (SDL, SDL-RT or C declarations), the syntax tree type will be `DECLARATIONS` and its attributes will be a list of descriptors for the declarations in the symbol.
- For a task block, the syntax tree type will be `TASK` and its attributes will include a list of statement descriptors for all statements in the task block.

Notes:

- In SDL, expressions and statements must be described recursively. For example, a binary operation expression may be described as an expression for the left operand, followed by a binary operator, followed by an expression for the right operand. For statements, an `IF` statement may be described by a descriptor for the tested condition, followed by the list of statements for the `THEN` part, followed by the list of statements for the `ELSE` part if any. This is a problem to build the descriptors, as recursive data structures are not supported by all versions of CORBA, and cannot be represented in all IDL language mappings.
  The chosen solution has been to store all descriptors for expressions and statements in lists attached to the syntax tree, then to reference them in the syntax tree itself by their index in these lists. So the syntax tree actually has the following fields:
  - A list of expression descriptors, called `expressionNodes`, containing all expression descriptors used in the syntax tree;
  - A list of statement descriptors, called `statementNodes`, containing all statement descriptors used in the syntax tree;
  - A syntax tree body, called `body`, which is the actual syntax tree with its type and attributes. Whenever this syntax tree must reference an expression descriptor or a statement descriptor, it uses an `ExpressionNodeId` or a `StatementNodeId` (resp.), that are simply indices in the `expressionNodes` or `statementNodes` lists (resp.).

  So, in the examples above:

- In the attributes for a timer start syntax tree, the expression for the time-out value is actually stored as an `ExpressionNodeId`, index in the syntax tree's `expressionNodes` list for the actual expression descriptor. This expression descriptor may in turn reference other expressions by their `ExpressionNodeId`.
- In the attributes for a task block syntax tree, the list of statements is actually a list of `StatementNodeId`, indices for the actual statement descriptors in the syntax tree's `statementNodes` list. These statement descriptors may in turn reference other statements by their `StatementNodeId`.

- In SDL-RT projects, expressions, statements and declarations are not parsed, and their exact description is not known by RTDS. So special types have been introduced, named `RAW_EXPRESSION`, `RAW_STATEMENT` and `RAW_DECLARATION` respectively, for which only the expression, statement or declaration text is known.

17.2.8 Class GlobalDataManager

Instances of this class gather all the objects managed at project level. These objects are:

- The signals defined in the whole project as instances of `Signal` (cf. paragraph "Class Signal" on page 329);
- The signal lists defined in the whole project as instances of `SignalList` (cf. paragraphe "Class SignalList" on page 329);
- For SDL-RT projects, the semaphores defined in the whole project as instances of `Item` (cf. paragraph "Class Item" on page 324).

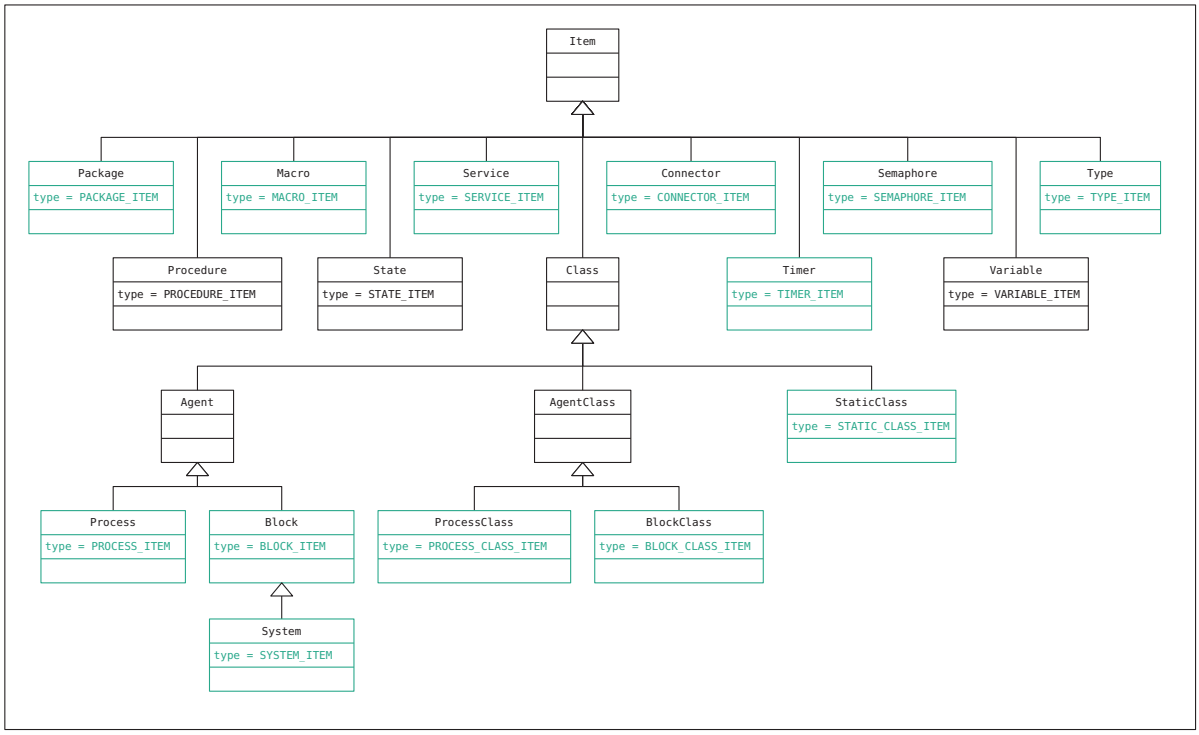
An instance of `GlobalDataManager` is accessible for each project via its attribute `gdm`.

17.2.9 Class Item

Items define the logical architecture of a project. An item may be:

- an agent (system, block or process),
- an agent class (block class or process class),
- a package,
- a procedure,
- a macro (SDL projects only),
- a service (SDL projects only),
- a state (which may be composite in SDL projects),
- a type (defined via `NEWTYPE` and `SYNTYPE` in SDL projects),
- a variable (including synonyms; only in SDL projects),
- a connector in a behavioral diagram,
- a timer,
- a static (passive) class (only in SDL-RT projects),
- a semaphore (only in SDL-RT projects).

The exact type for the item is remembered in its attribute `type`. The type hierarchy for items is the following:



Note: in this diagram, only the classes in black are actually declared in the IDL. The other ones have nothing to add to their super-class and their declarations would be empty.

In addition to their type, items know:

- Their name;
- Their full name, which generally is their name prefixed with the full name for their parent package separated by ’::’;
- Their children, parent and parent package (instances of `Item`);
- The element describing them as an instance of `Element` (cf. paragraph “Class Element” on page 322). This element is for example the process diagram for a process. For a procedure in a SDL-RT project, it can also be a C source file if the procedure is implemented in C;
- The list of symbols defining the item and the list of symbols using it, as instances of `Symbol` (cf. “Class Symbol” on page 331). The meaning of these lists depend on the item type:

Item type	Defining symbols	Using symbols([a])
`PACKAGE_ITEM`	(none)	(none)
`STATIC_CLASS_ITEM`	(none)	Class symbols for this class in UML class diagram
`SYSTEM_ITEM`	(none)	(none)

Item type	Defining symbols	Using symbols([a])
BLOCK_ITEM	Block symbol in block's parent	(Same as defining symbols)
BLOCK_CLASS_ITEM	(none)	Block class symbols in UML class diagrams + block class instance symbols for this class
PROCESS_ITEM	Process symbol in process's parent	Process creation symbols for this process + defining symbol(s)
PROCESS_CLASS_ITEM	(none)	Process class symbols in UML class diagrams + process class instance symbols for this class
PROCEDURE_ITEM	Procedure symbol in procedure's parent	(none)
SERVICE_ITEM	Service symbol in service's parent	(none)
MACRO_ITEM	Macro declaration symbol in macro parent diagram if any	(none)
STATE_ITEM	State declaration symbol for composite state + state symbols starting transitions for this state	State symbols ending transitions for this state
CONNECTOR_ITEM	"In" connector symbols (labels)	"Out" connector symbols (JOINs)
TIMER_ITEM	Symbols starting the timer	Symbols cancelling the timer
SEMAPHORE_ITEM	Declaration symbol for the semaphore	Take and give symbols for the semaphore.
TYPE_ITEM	Declaration symbol for the type	(none)
VARIABLE_ITEM	Declaration symbol for the variable	(none)

a. Always includes an additional symbol for items having diagrams, representing the diagram frame.

- The list of source files defining the item as instances of `Element` (cf. paragraph "Class Element" on page 322). This may happen for types and variables, that can be defined in declaration files in packages.
  Note: in a valid project, a given `Item` may only have exactly one defining symbol or exactly one defining source file.
- The package declared as `USE`d in the item.

In addition to these, an item also manages:

- The signal windows attached to it if applicable. Signal windows are connection points between channels and the item and are represented as instances of `Sig-`

nalWindow (cf. paragraph “Class SignalWindow” on page 330). For agent classes, signal windows also include the gates declared in the class diagrams.
- The channels declared in the item’s diagram if applicable. These channels are represented as instances of Channel (cf. paragraph “Class Channel” on page 322).
- The transitions defined in the item’s diagram if applicable. The fields in a descriptor for a transition are:
  - A boolean statesExcluded indicating if the transition is defined for a given set of states, or for all states except a given set. For example, a transition starting from a state symbol containing "S1, S2" is defined for the states S1 and S2, so its statesExcluded indicator will be false. But a transition starting from a state symbol containing "*(S1, S2)" is defined for all states but S1 and S2, so its statesExcluded indicator will be true.
  - The list of states appearing in the state symbol starting the transition. Depending on the statesExcluded indicator, it can be the list of states for which the transition is defined (if indicator is false), or the states for which the transition is not defined (indicator is true). The states are represented by instances of State (cf. paragraph “Class State” on page 331).
    If a transition has a statesExcluded indicator set to true and an empty list of states, it means that it’s defined for all states (the state symbol contains "*"). If a transition has a statesExcluded indicator set to false and an empty list of states, the transition is the start transition for the item.
  - The symbol starting the transition. This symbol may be an input symbol, a save symbol, a continuous signal symbol or a start symbol. Its type and contents may be decoded via its syntax tree.

Signal windows, channels and transitions are managed via operations which are defined in the Item class. However, for items not supporting the operation, it will raise an exception DictionaryItemOperationError. This is the case for example when trying to get transitions from a block or a connector, or when trying to get channels from a procedure or a timer.

17.2.10 Class ObjectServer

This class is the entry point for the server. When started, the server creates an ObjectServer instance and writes a textual representation of its identifier to the file specified in its -f option. So a client will always get an instance of this class first.

The only operations on this class are:
- loadProject, loading a project in a given file and returning the instance of Project describing it (cf. “Class Project” on page 328);
- quit, making the server quit.

Notes:
- The loadProject operation is executed by the server, so if the server and the client are on different machines, the project file must be seen by the server, and specified as it is seen by the server. For example, if the server is executed on a Unix machine, and the client on a Windows one, the server won’t be able to understand paths like H:\MyFiles\MyProject.rdp, even if H: is a shared

drive, as such a path is only known to Windows. The specified path for `loadProject` must then be something like:
`/path-to-shared-drive/MyFiles/MyProject.rdp`
To avoid this kind of problem, running the client and the server on different machines is strongly discouraged.

- There is no checking at all done by the `quit` operation. So any other client is connected to the same server, its connection will be immediatly closed without warning. So this operation must be used only if the client knows that it is the only one connected to the server, e.g if the client started the server itself for its own use.

17.2.11 Class Operation

An instance of this class describes an operation declared in a class symbol in a UML class diagram. Its attributes are all information that may be extracted from the UML operation declaration:

- Visibility (public, protected, private);
- Name;
- Parameters with for each:
  - its direction (in, out or in/out),
  - its name,
  - its type,
  - its default value (if any);
- Return type;
- Properties (as `query` in "`myOp() : int {query}`").

17.2.12 Class Procedure

This class is a sub-class or `Item` (cf. “Class Item” on page 324).

Among the features inherited from `Item`:

- Operations dealing with signal windows do not work;
- Operations dealing with contained channels do not work;
- Operations dealing with contained transitions work.

In addition to these features, the procedure knows its signature as declared in the procedure declaration symbol (NB: this means that in SDL, the signature will actually only be the procedure name).

17.2.13 Class Project

Instances of this class represent a project as loaded from a project file via the operation `loadProject` on class `ObjectServer`.

A project only knows:

- Its associated `GlobalDataManager` instance (cf. paragraph “Class GlobalDataManager” on page 324);
- An item representing the top-level pseudo-package for the project itself. This pseudo-package contains all top-level items in the project (usually a system and

a set of packages). It is represented as an instance of `Item` with the type `PACKAGE_ITEM` (cf. paragraph “Class Item” on page 324).

17.2.14 Class Role

Instances of this class represent roles played by classes in associations in UML class diagrams. Attributes for a role are:

- The association for which it is defined as an instance of `Association` (cf. paragraph “Class Association” on page 321);
- Its name;
- Its cardinality;
- Whether it is navigable or not.

17.2.15 Class Signal

Instances of this class represent signals in the project. Signals are always managed globally via an instance of `GlobalDataManager` (cf. paragraph “Class GlobalDataManager” on page 324).

The attributes for a signal are:

- Its name;
- Whether signal is a timer or not. Note that signals for timers are referenced by the instance of `GlobalDataManager` for a project, even if they are local to a process or procedure;
- The type of its parameters. These types are given as strings, not as type items;
- The signal windows sending and receiving this signal (cf. paragraph “Class SignalWindow” on page 330);
- The elements where the signal is declared. These elements may be declaration symbols or declaration files. In a correct project, there should be exactly one declaration element for each signal;
- The input and output symbols for this signal.

17.2.16 Class SignalList

Instances of this class represent signal lists in the project. Signal lists are always managed globally via an instance of `GlobalDataManager` (cf. paragraph “Class GlobalDataManager” on page 324).

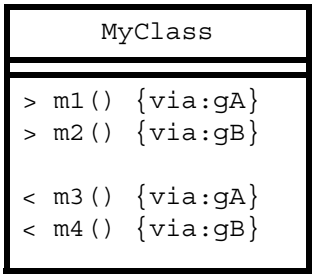
The attributes for a signal list are:

- Its name;
- The signals and signal lists it contains as instances of `Signal` and `SignalList` respectively;
- The signal windows sending and receiving this signal list (cf. paragraph “Class SignalWindow” on page 330);
- The elements where this signal list is declared, which can be declaration symbols or declaration files. In a correct project, there should be exactly one declaration element for each signal list.

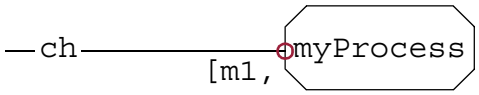
17.2.17 Class SignalWindow

Instances of this class represent connection points between channel and agents. There are three main types of signal windows:

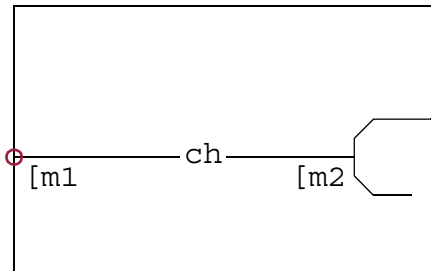
- Gates as defined in class diagrams for agent classes:



  Here, two signal windows are defined: the one for gate `gA` (receiving `m1` and sending `m3`) and the one for gate `gB` (receiving `m2` and sending `m4`).
- "Outside" signal windows, connecting a channel to the outside border of an agent:



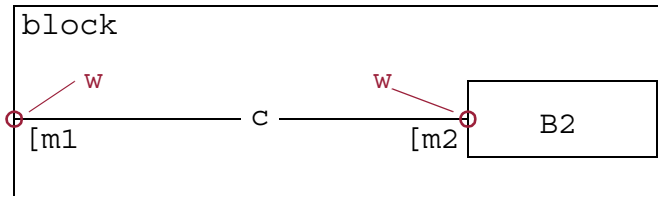
- "Inside" signal windows, connecting a channel to the inner border of an agent:



The attributes for a signal window are:

- The item to which it is attached (cf. paragraph “Class Item” on page 324);
- A boolean indicating if the signal window is inside or outside. Gates defined in class diagrams are considered to be outside the parent item;
- The gate name for the signal window if any. If the signal window is not a gate, the gate name is empty;
- The channel attached to the signal window if any, as an instance of `Channel` (cf. “Class Channel” on page 322). If the signal window is a gate defined in a class diagram, this attribute is set to an empty reference;
- The signal window facing this one on its attached channel. If the signal window is a gate defined in a class diagram, this attribute is set to an empty reference;
- The list of connections for this signal window as a list of strings. Since connections are only meaningful for "inside" signal windows, this list will always be empty for an "outside" signal window or a gate defined in a class diagram;

- The signals and signal lists that the window sends and receives. Note that these lists are defined from the signal window point of view. So for example, in:



  - Signal window `w1` for block `B1` sends `m2` and receives `m1`;
  - Signal window `w2` for block `B2` sends `m1` and receives `m2`.
- A boolean indicating if the signal window is actually a defined one, i.e if it appears in an agent class symbol in a class diagram (NB: this feature is actually implemented via the operation `isDefined()` instead of an attribute).

17.2.18 Class State

This class is a sub-class of `Item` (cf. “Class Item” on page 324).

Among the features inherited from `Item`:

- Operations dealing with signal windows work, but no signal window will be defined if the state is not composite;
- Operations dealing with contained channels work, but no channel will be defined if state is not composite;
- Operations dealing with contained transitions do not work.

In addition to these features, a state knows whether it is composite or not.

17.2.19 Class Symbol

This class is a sub-class of `Element` (cf. “Class Element” on page 322). Its instances represent symbols in a diagram. The `nodeFileName` attribute for symbols is the file name for their container diagram.

In addition to the features inherited from `Element`, a symbol knows:

- Its internal identifier. This is the string "`SYMBnnn`" used to identify the symbol inside its diagram.
- The symbols following it if the symbol is part of a transition. For each following symbol, the syntax tree for the link going from the current symbol to the following one is also available. This syntax tree is only used for decision and transition option branches, where the link text contains the conditions on the tested value.

17.2.20 Class Variable

This class is a sub-class of `Item` (cf. “Class Item” on page 324).

Among the features inherited from `Item`:

- Operations dealing with signal windows do not work;
- Operations dealing with contained channels do not work;
- Operations dealing with contained transitions do not work.

In addition to these features, a variable knows:

- The name for its type;

- Its type itself as an instance of `Item` (cf. paragraph “Class Item” on page 324). This instance will only be available if the type definition can be found in any ancestor for the variable. If it can’t, this attribute will contain an empty object reference.
  Note: this will be the case for all predefined types, as they are not declared in the project.

18 - SGML export for RTDS documents

18.1 - Principles

In addition to be able to export documents in formats usable in word processors, RTDS also provides a means to entirely generate a full document by using the SGML format. This is a somewhat advanced feature and should only be used by experienced users.

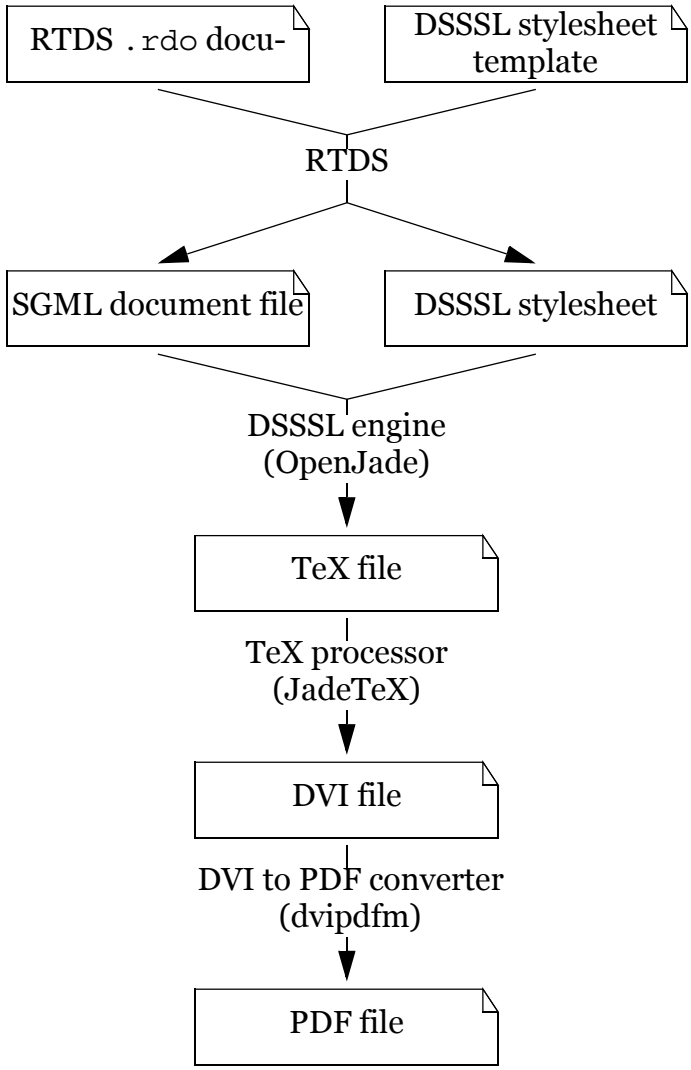
The formats and tools used during this process are the following:

- *SGML* (Standard Generalized Markup Language) is a "meta-language" allowing to define custom document types. It is an ISO standard (ISO 8879:1986). SGML documents are usually based on structure rather than presentation: documents will be split in logical units and a presentation will be applied on each logical unit afterwards, considering the unit type and its context.
  SGML is very closely related to XML, as XML is a simplified version of SGML. The main difference is that SGML documents have to be defined via a *DTD* (Document Type Definition) file. The DTD used by RTDS SGML documents is specific and is provided in RTDS distribution under `$RTDS_HOME/share/sgml/rtds-doc.dtd`.
  For more information on SGML, see `http://xml.coverpages.org/sgml.html`.
- *DSSSL* (Document Style and Semantics Specification Language) is a way of defining the presentation for a SGML document. DSSSL files are usually called stylesheets, but they are far more powerful than this name implies, and rather look like formatting programs for SGML documents than simple stylesheets. DSSSL is also an ISO standard (ISO 10179:1996).
  Because of its complexity, DSSSL is not widely used today, and simpler, but less powerful formats such as XML/XSL are usually preferred.
  For more information on DSSSL, including the full text of the standard, see `http://xml.coverpages.org/dsssl.html`.
- *OpenJade* is the most advanced - if not the only - free DSSSL engine available today. A DSSSL engine takes a SGML document, its DTD and a DSSSL stylesheet and produces a document that can be directly printed or opened in a word processor. Output formats supported by OpenJade include RTF (Rich Text Format, supported by almost all major word processors), MIF (Maker Interchange format, supported by Adobe FrameMaker) and TeX through a specific macro package called JadeTeX.
  OpenJade's home page is `http://openjade.sourceforge.net`; the latest version and the recommended one for RTDS is 1.3.2.
  JadeTeX home page is `http://jadetex.sourceforge.net`; the latest version and the one recommended for RTDS is 3.13.
- TeX is the well-known typesetting system commonly used on Unix. More information and downloads can be found on the TeX Users Group website at `http://www.tug.org`.
- `dvipdfm` converts DVI files as output by TeX to very clean PDF files including hyperlinks. `dvipdfm` is available for download at `http://gaspra.kettering.edu/dvipdfm`.

Warning: The main difficulty in producing a document using these formats and tools is the design of the DSSSL "stylesheet". As said above, this "stylesheet" looks much more like a formatting program for a SGML document, written in a Lisp / Scheme-like functional language. This kind of language is considered as difficult to handle even among experienced programmers; so producing documents through SGML is obviously not for the average user.

18.2 - Documentation generation process

The preferred process for documentation generation via SGML is the following:



- Starting from a document file (`.rdo`) and from a DSSSL stylesheet template, RTDS generates a SGML document file, conforming to a provided DTD, and a full DSSSL stylesheet. How the final DSSSL stylesheet is produced from the template is described in the paragraph “DSSSL stylesheet production” on page 335.
- Using OpenJade as the DSSSL engine, a TeX file is produced, using macros defined in the JadeTeX package.
- Processing the TeX file via `jadetex` produces a DVI file.
- Converting the DVI file with `dvipdfm` produces the final PDF document.

This process has the advantage of not requiring any word processor to produce a final document. Except for the first step (SGML export of the RTDS document), the production can even be fully automated via a makefile.

It is obviously possible to produce documents in other formats such as RTF or MIF (using OpenJade) or other ones if another DSSSL engine is used.

18.3 - DSSSL stylesheet production

RTDS is able to produce a full DSSSL stylesheet by using a stylesheet template and the style descriptions set up in the document export formats for SGML. The stylesheet template may contain the following marker lines:

- `;%HEADING_STYLES%:`
  This line will be replaced by the definition of a variable name `*heading-styles*` describing the paragraph styles for section headings.
- `;%PARAGRAPH_STYLES%:`
  This line will be replaced by the definition of a variable named `*paragraph-styles*` describing the "normal" paragraph styles.
- `;%CHARACTER_STYLES%:`
  This line will be replaced by the definition of a variable named `*character-styles*` describing all character styles.

These variables are written as association lists, usable in the DSSSL stylesheet via the `assoc` standard function. For character and "normal" paragraph styles, the key in the association list is the style name. For section heading paragraph styles, the key is the section level, starting at 1.

The description for a paragraph style - "normal" and section heading ones - is a list containing:

- The font for the paragraph as a list (font family name, font size, bold indicator, italic indicator, underlined indicator). The indicators are boolean values.
- The color for the text, as a list of RGB coordinates, which are 3 real numbers between `0.0` and `1.0`.
- The alignment for the paragraph as a valid value for the `quadding` option for the `make paragraph` DSSSL construct.
- The line spacing for the paragraph.
- The left margin for the paragraph, as a distance from the left display area border.
- The right margin for the paragraph, as a distance from the right display area border.
- The indent for the paragraph first line, as a distance from its left margin.
- The width for the paragraph heading if any.
- The additional space above the paragraph.
- The additional space below the paragraph.
- A boolean indicating if a page break should be forced before the paragraph.
- A boolean indicating if the paragraph should be kept on the same page as the one following it.
- A boolean indicating if the paragraph should be kept on the same page as the one preceding it.
- The widow / orphan line count for the paragraph.

The description of a character styles is a list containing:

- The font for the characters as a list (font family name, font size, bold indicator, italic indicator, underlined indicator). The indicators are boolean values.
- The color for the text, as a list of RGB coordinates, which are 3 real numbers between `0.0` and `1.0`.

A full example of a DSSSL stylesheet template with a RTDS document using it is in the RTDS distribution in `$RTDS_HOME/share/sgml/example`.

19 - GNU distribution

19.1 - gcc options

This chapter gathers all gcc options as printed by the help command.

19.1.1 Usage: cpp [switches] input output

Switches:

-include <file>	Include the contents of <file> before other files
-imacros <file>	Accept definition of marcos in <file>
-iprefix <path>	Specify <path> as a prefix for next two options
-iwithprefix <dir>	Add <dir> to the end of the system include paths
-iwithprefixbefore <dir>	Add <dir> to the end of the main include paths
-isystem <dir>	Add <dir> to the start of the system include paths
-idirafter <dir>	Add <dir> to the end of the system include paths
-I <dir>	Add <dir> to the end of the main include paths
-nostdinc	Do not search the system include directories
-nostdinc++	Do not search the system include directories for C++
-o <file>	Put output into <file>
-pedantic	Issue all warnings demanded by strict ANSI C
-traditional	Follow K&R pre-processor behaviour
-trigraphs	Support ANSI C trigraphs
-lang-c	Assume that the input sources are in C
-lang-c89	Assume that the input is C89; depricated
-lang-c++	Assume that the input sources are in C++
-lang-objc	Assume that the input sources are in ObjectiveC
-lang-objc++	Assume that the input sources are in ObjectiveC++
-lang-asm	Assume that the input sources are in assembler
-lang-chill	Assume that the input sources are in Chill
-std=<std name>	Specify the conformance standard; one of: gnu89, gnu9x, c89, c9x, iso9899:1990, iso9899:199409, iso9899:199x
-+	Allow parsing of C++ style features
-w	Inhibit warning messages
-Wtrigraphs	Warn if trigraphs are encountered
-Wno-trigraphs	Do not warn about trigraphs
-Wcomment{s}	Warn if one comment starts inside another
-Wno-comment{s}	Do not warn about comments
-Wtraditional	Warn if a macro argument is/would be turned into a string if -traditional is specified
-Wno-traditional	Do not warn about stringification
-Wundef	Warn if an undefined macro is used by #if

Switches:

-Wno-undef	Do not warn about testing undefined macros
-Wimport	Warn about the use of the #import directive
-Wno-import	Do not warn about the use of #import
-Werror	Treat all warnings as errors
-Wno-error	Do not treat warnings as errors
-Wall	Enable all preprocessor warnings
-M	Generate make dependencies
-MM	As -M, but ignore system header files
-MD	As -M, but put output in a .d file
-MMD	As -MD, but ignore system header files
-MG	Treat missing header file as generated files
-g	Include #define and #undef directives in the output
-D<macro>	Define a <macro> with string '1' as its value
-D<macro>=<val>	Define a <macro> with <val> as its value
-A<question> (<answer>)	Assert the <answer> to <question>
-U<macro>	Undefine <macro>
-u or -undef	Do not predefine any macros
-v	Display the version number
-H	Print the name of header files as they are used
-C	Do not discard comments
-dM	Display a list of macro definitions active at end
-dD	Preserve macro definitions in output
-dN	As -dD except that only the names are preserved
-dI	Include #include directives in the output
-ifoutput	Describe skipped code blocks in output
-P	Do not generate #line directives
-$	Do not include '$' in identifiers
-remap	Remap file names when including files.
-h or --help	Display this information

19.1.2 Usage: cc1 input [switches]

Switches:

-ffixed-<register>	Mark <register> as being unavailable to the compiler
-fcall-used-<register>	Mark <register> as being corrupted by function calls
-fcall-saved-<register>	Mark <register> as being preserved across functions
-finline-limit-<number>	Limits the size of inlined functions to <number>
-fident	Process #ident directives
-fleading-underscore	External symbols have a leading underscore
-finstrument-functions	Instrument function entry/exit with profiling calls
-fdump-unnumbered	Suppress output of instruction numbers and line number notes in debugging dumps
-fprefix-function-name	Add a prefix to all function names
-fcheck-memory-usage	Generate code to check every memory access
-fstrict-aliasing	Assume strict aliasing rules apply
-fargument-noalias-global	Assume arguments do not alias each other or globals
-fargument-noalias	Assume arguments may alias globals but not each other
-fargument-alias	Specify that arguments may alias each other & globals
-fstack-check	Insert stack checking code into the program
-fpack-struct	Pack structure members together without holes
-foptimize-register-move	Do the full regmove optimization pass
-fregmove	Enables a register move optimisation
-fgnu-linker	Output GNU ld formatted global initialisers
-fverbose-asm	Add extra commentry to assembler output
-fdata-sections	place data items into their own section
-ffunction-sections	place each function into its own section
-finhibit-size-directive	Do not generate .size directives
-fcommon	Do not put unitialised globals in the common section
-ffast-math	Improve FP speed by violating ANSI & IEEE rules
-fbranch-probabilities	Use profiling information for branch probabilities
-ftest-coverage	Create data files needed by gcov
-fprofile-arcs	Insert arc based program profiling code
-fasynchronous-exceptions	Support asynchronous exceptions
-fsjlj-exceptions	Use setjmp/longjmp to handle exceptions
-fnew-exceptions	Use the new model for exception handling
-fexceptions	Enable exception handling
-fpic	Generate position independent code, if possible
-fbranch-count-reg	Replace add,compare,branch with branch on count reg
-fsched-spec-load-dangerous	Allow speculative motion of more loads
-fsched-spec-load	Allow speculative motion of some loads
-fsched-spec	Allow speculative motion of non-loads

Switches:

-fsched-interblock	Enable scheduling across basic blocks
-fschedule-insns2	Run two passes of the instruction scheduler
-fschedule-insns	Reschedule instructions to avoid pipeline stalls
-fpretend-float	Pretend that host and target use the same FP format
-frerun-loop-opt	Run the loop optimiser twice
-frerun-cse-after-loop	Run CSE pass after loop optimisations
-fgcse	Perform the global common subexpression elimination
-fdelayed-branch	Attempt to fill delay slots of branch instructions
-freg-struct-return	Return 'short' aggregates in registers
-fpcc-struct-return	Return 'short' aggregates in memory, not registers
-fcaller-saves	Enable saving registers around function calls
-fshared-data	Mark data as shared rather than private
-fsyntax-only	Check for syntax errors, then stop
-fkeep-static-consts	Emit static const variables even if they are not used
-finline	Pay attention to the 'inline' keyword
-fkeep-inline-functions	Generate code for funcs even if they are fully inlined
-finline-functions	Integrate simple functions into their callers
-ffunction-cse	Allow function addresses to be held in registers
-fforce-addr	Copy memory address constants into regs before using
-fforce-mem	Copy memory operands into registers before using
-fpeephole	Enable machine specific peephole optimisations
-fwritable-strings	Store strings in writable data section
-freduce-all-givs	Strength reduce all loop general induction variables
-fmove-all-movables	Force all loop invariant computations out of loops
-funroll-all-loops	Perform loop unrolling for all loops
-funroll-loops	Perform loop unrolling when iteration count is known
-fstrength-reduce	Perform strength reduction optimisations
-fthread-jumps	Perform jump threading optimisations
-fexpensive-optimizations	Perform a number of minor, expensive optimisations
-fcse-skip-blocks	When running CSE, follow conditional jumps
-fcse-follow-jumps	When running CSE, follow jumps to their targets
-fomit-frame-pointer	When possible do not generate stack frames
-fdefer-pop	Defer popping functions args from stack until later
-fvolatile-static	Consider all mem refs to static data to be volatile
-fvolatile-global	Consider all mem refs to global data to be volatile
-fvolatile	Consider all mem refs through pointers as volatile
-ffloat-store	Do not store floats in registers
-O[number]	Set optimisation level to [number]
-Os	Optimise for space rather than speed
-pedantic	Issue warnings needed by strict compliance to ANSI C

Switches:

-pedantic-errors	Like -pedantic except that errors are produced
-w	Suppress warnings
-W	Enable extra warnings
-Winline	Warn when an inlined function cannot be inlined
-Wuninitialized	Warn about unitialized automatic variables
-Wcast-align	Warn about pointer casts which increase alignment
-Waggregate-return	Warn about returning structures, unions or arrays
-Wswitch	Warn about enumerated switches missing a specific case
-Wshadow	Warn when one local variable shadows another
-Wunused	Warn when a variable is unused
-Wid-clash-<num>	Warn if 2 identifiers have the same first <num> chars
-Wlarger-than-<number>	Warn if an object is larger than <number> bytes
-p	Enable function profiling
-a	Enable block profiling
-ax	Enable jump profiling
-o <file>	Place output into <file>
-G <number>	Put global and static data smaller than <number> bytes into a special section (on some targets)
-gdwarf-2	Enable DWARF-2 debug output
-gdwarf+	Generated extended DWARF-1 format debug output
-gdwarf	Generate DWARF-1 format debug output
-gstabs+	Generate extended STABS format debug output
-gstabs	Generate STABS format debug output
-ggdb	Generate default extended debug format output
-g	Generate default debug format output
-aux-info <file>	Emit declaration info into <file>.X
-quiet	Do not display functions compiled or elapsed time
-version	Display the compiler's version
-d[letters]	Enable dumps from specific passes of the compiler
-dumpbase <file>	Base name to be used for dumps from specific passes
-sched-verbose-<number>	Set the verbosity level of the scheduler
--help	Display this information

19.1.3 Language specific options:

Language specific options

-ansi	Compile just for ANSI C
-fallow-single-precisio	Do not promote floats to double if using -traditional
-std=	Determine language standard
-funsigned-bitfields	Make bitfields by unsigned by default
-fsigned-char	Make 'char' be signed by default
-funsigned-char	Make 'char' be unsigned by default
-traditional	Attempt to support traditional K&R style C
-fno-asm	Do not recognise the 'asm' keyword
-fno-builtin	Do not recognise any built in functions
-fhosted	Assume normal C execution environment
-ffreestanding	Assume that standard libraries & main might not exist
-fcond-mismatch	Allow different types as args of ? operator
-fdollars-in-identifier	Allow the use of $ inside identifiers
-fshort-double	Use the same size for double as for float
-fshort-enums	Use the smallest fitting integer to hold enums
-Wall	Enable most warning messages
-Wbad-function-cast	Warn about casting functions to incompatible types
-Wmissing-noreturn	Warn about functions which might be candidates for attribute noreturn
-Wcast-qual	Warn about casts which discard qualifiers
-Wchar-subscripts	Warn about subscripts whose type is 'char'
-Wcomment	Warn if nested comments are detected
-Wcomments	Warn if nested comments are detected
-Wconversion	Warn about possibly confusing type conversions
-Wformat	Warn about printf format anomalies
-Wimplicit-function-dec	Warn about implicit function declarations
-Wimplicit-int	Warn when a declaration does not specify a type
-Wimport	Warn about the use of the #import directive
-Wno-long-long	Do not warn about using 'long long' when -pedantic
-Wmain	Warn about suspicious declarations of main
-Wmissing-braces	Warn about possibly missing braces around initialisers
-Wmissing-declarations	Warn about global funcs without previous declarations
-Wmissing-prototypes	Warn about global funcs without prototypes
-Wmultichar	Warn about use of multicharacter literals
-Wnested-externs	Warn about externs not at file scope level
-Wparentheses	Warn about possible missing parentheses
-Wpointer-arith	Warn about function pointer arithmetic
-Wredundant-decls	Warn about multiple declarations of the same object
-Wsign-compare	Warn about signed/unsigned comparisons
-Wunknown-pragmas	Warn about unrecognised pragmas

Language specific options

-Wstrict-prototypes	Warn about non-prototyped function decls
-Wtraditional	Warn about constructs whose meaning change in ANSI C
-Wtrigraphs	Warn when trigraphs are encountered
-Wwrite-strings	Mark strings as ’const char *’

19.1.3.1 Options for Objective C:

-gen-decls	Dump decls to a .decl file
-fgnu-runtime	Generate code for GNU runtime environment
-fnext-runtime	Generate code for NeXT runtime environment
-Wselector	Warn if a selector has multiple methods
-Wno-protocol	Do not warn if inherited methods are unimplemented
-print-objc-runtime-inf	Generate C header of platform specific features

19.1.3.2 Options for Chill:

-fno-local-loop-counter	Do not make seperate scopes for every 'for' loop
-fgrant-only	Stop after successfully generating a grant file
-fold-strings	Implement the 1984 Chill string semantics
-fignore-case	convert all idenitifers to lower case
-fpack	Pack structures into available space
-fspecial_UC	Make special words be in uppercase
-fno-runtime-checking	Disable runtime checking of parameters

19.1.3.3 Options for C++:

-fno-access-control	Do not obey access control semantics
-fall-virtual	Make all member functions virtual
-falt-external-template	Change when template instances are emitted
-fcheck-new	Check the return value of new
-fconserve-space	Reduce size of object files
-fno-const-strings	Make string literals 'char[]' instead of 'const char[]'
-fno-default-inline	Do not inline member functions by default
-fno-rtti	Do not generate run time type descriptor information
-fno-for-scope	Scope of for-init-statement vars extends outside
-fguiding-decls	Implement guiding declarations
-fno-gnu-keywords	Do not recognise GNU defined keywords
-fhandle-signatures	Handle signature language constructs
-fhonor-std	Treat the namespace 'std' as a normal namespace
-fhuge-objects	Enable support for huge objects
-fno-implement-inlines	Export functions even if they can be inlined
-fno-implicit-templates	Only emit explicit template instatiations
-fno-implicit-inline-te	Only emit explicit instatiations of inline templates
-finit-priority	Handle the init_priority attribute
-flabels-ok	Labels can be used as first class objects
-fnew-abi	Enable experimental ABI changes

-fno-nonnull-objects	Do not assume that a reference is always valid
-foperator-names	Recognise and/bitand/bitor/compl/not/or/xor
-fno-optional-diags	Disable optional diagnostics
-fpermissive	Downgrade conformance errors to warnings
-frepo	Enable automatic template instantiation
-fsquangle	Enable squashed name mangling
-fstats	Display statistics accumulated during compilation
-fno-strict-prototype	Do not assume that empty prototype means no args
-ftemplate-depth-	Specify maximum template instantiation depth
-fthis-is-variable	Make 'this' not be type '* const'
-fvtable-gc	Discard unused virtual functions
-fvtable-thunks	Implement vtables using thunks
-fweak	Emit common-like symbols as weak symbols
-fxref	Emit cross referencing information
-Wreturn-type	Warn about inconsistent return types
-Woverloaded-virtual	Warn about overloaded virtual function names
-Wno-ctor-dtor-privacy	Don't warn when all ctors/dtors are private
-Wnon-virtual-dtor	Warn about non virtual destructors
-Wextern-inline	Warn when a function is declared extern, then inline
-Wreorder	Warn when the compiler reorders code
-Wsynth	Warn when synthesis behaviour differs from Cfront
-Wno-pmf-conversions	Don't warn when type converting pointers to member functions
-Weffc++	Warn about violations of Effective C++ style rules
-Wsign-promo	Warn when overload promotes from unsigned to signed
-Wold-style-cast	Warn if a C style cast is used in a program
-Wno-non-template-frien	Don't warn when non-templatized friend functions are declared within a template
-Wno-deprecated	Don't announce deprecation of compiler features

19.1.3.4 Options for Fortran:

Options for Fortran

-fversion	Print g77-specific compiler version info, run internal tests
-ff66	Program is written in typical FORTRAN 66 dialect
-ff77	Program is written in typical Unix f77 dialect
-fno-f77	Program does not use Unix-f77 dialectal features
-ff90	Program is written in Fortran-90-ish dialect
-fno-automatic	Treat local vars and COMMON blocks as if they were named in SAVE statements
-fdollar-ok	Allow $ in symbol names
-fno-f2c	f2c-compatible code need not be generated
-fno-f2c-library	Unsupported; do not generate libf2c-calling code
-fflatten-arrays	Unsupported; affects code-generation of arrays
-ffree-form	Program is written in Fortran-90-ish free form
-fpedantic	Warn about use of (only a few for now) Fortran extensions
-fvxt	Program is written in VXT (Digital-like) FORTRAN
-fno-ugly	Disallow all ugly features
-fno-ugly-args	Hollerith and typeless constants not passed as arguments
-fugly-assign	Allow ordinary copying of ASSIGN'ed vars
-fugly-assumed	Dummy array dimensioned to (1) is assumed-size
-fugly-comma	Trailing comma in procedure call denotes null argument
-fugly-complex	Allow REAL(Z) and AIMAG(Z) given DOUBLE COMPLEX Z
-fno-ugly-init	Initialization via DATA and PARAMETER is type-compatible
-fugly-logint	Allow INTEGER and LOGICAL interchangeability
-fxyzzy	Print internal debugging-related info
-finit-local-zero	Initialize local vars and arrays to zero
-fno-backslash	Backslashes in character/hollerith constants not special (C-style)
-femulate-complex	Have front end emulate COMPLEX arithmetic to avoid bugs
-fno-underscoring	Disable the appending of underscores to externals
-fno-second-underscore	Never append a second underscore to externals
-fintrin-case-initcap	Intrinsics spelled as e.g. SqRt
-fintrin-case-upper	Intrinsics in uppercase
-fintrin-case-any	Intrinsics letters in arbitrary cases
-fmatch-case-initcap	Language keywords spelled as e.g. IOStat
-fmatch-case-upper	Language keywords in uppercase
-fmatch-case-any	Language keyword letters in arbitrary cases
-fsource-case-upper	Internally convert most source to uppercase
-fsource-case-preserve	Internally preserve source case
-fsymbol-case-initcap	Symbol names spelled in mixed case
-fsymbol-case-upper	Symbol names in uppercase

Options for Fortran

-fsymbol-case-lower	Symbol names in lowercase
-fcase-strict-upper	Program written in uppercase
-fcase-strict-lower	Program written in lowercase
-fcase-initcap	Program written in strict mixed-case
-fcase-upper	Compile as if program written in uppercase
-fcase-lower	Compile as if program written in lowercase
-fcase-preserve	Preserve all spelling (case) used in program
-fbadu77-intrinsics-del	Delete libU77 intrinsics with bad interfaces
-fbadu77-intrinsics-dis	Disable libU77 intrinsics with bad interfaces
-fbadu77-intrinsics-hid	Hide libU77 intrinsics with bad interfaces
-ff2c-intrinsics-delete	Delete non-FORTRAN-77 intrinsics f2c supports
-ff2c-intrinsics-disabl	Disable non-FORTRAN-77 intrinsics f2c supports
-ff2c-intrinsics-hide	Hide non-FORTRAN-77 intrinsics f2c supports
-ff90-intrinsics-delete	Delete non-FORTRAN-77 intrinsics F90 supports
-ff90-intrinsics-disabl	Disable non-FORTRAN-77 intrinsics F90 supports
-ff90-intrinsics-hide	Hide non-FORTRAN-77 intrinsics F90 supports
-fgnu-intrinsics-delete	Delete non-FORTRAN-77 intrinsics g77 supports
-fgnu-intrinsics-disabl	Disable non-FORTRAN 77 intrinsics F90 supports
-fgnu-intrinsics-hide	Hide non-FORTRAN 77 intrinsics F90 supports
-fmil-intrinsics-delete	Delete MIL-STD 1753 intrinsics
-fmil-intrinsics-disabl	Disable MIL-STD 1753 intrinsics
-fmil-intrinsics-hide	Hide MIL-STD 1753 intrinsics
-funix-intrinsics-delet	Delete libU77 intrinsics
-funix-intrinsics-disab	Disable libU77 intrinsics
-funix-intrinsics-hide	Hide libU77 intrinsics
-fvxt-intrinsics-delete	Delete non-FORTRAN-77 intrinsics VXT FORTRAN supports
-fvxt-intrinsics-disabl	Disable non-FORTRAN-77 intrinsics VXT FORTRAN supports
-fvxt-intrinsics-hide	Hide non-FORTRAN-77 intrinsics VXT FORTRAN supports
-fzeros	Treat initial values of 0 like non-zero values
-fdebug-kludge	Emit special debugging information for COMMON and EQUIVALENCE
-fonetrip	Take at least one trip through each iterative DO loop
-fno-silent	Print names of program units as they are compiled
-fno-globals	Disable fatal diagnostics about inter-procedural problems
-ftypeless-boz	Make prefix-radix non-decimal constants be typeless
-fbounds-check	Generate code to check subscript and substring bounds
-ffortran-bounds-check	Fortran-specific form of -fbounds-check
-Wno-globals	Disable warnings about inter-procedural problems
-Wsurprising	Warn about constructs with surprising meanings

Options for Fortran

-I	Add a directory for INCLUDE searching
-ffixed-line-length-	Set the maximum line length

19.1.3.5 Options for Java:

-fno-bounds-check	Disable automatic array bounds checking
-fassume-compiled	Make is_compiled_class return 1
-femit-class-files	Dump class files to <name>.class
-MD	Print dependencies to FILE.d
-MMD	Print dependencies to FILE.d
-M	Print dependencies to stdout
-MM	Print dependencies to stdout
-fclasspath	Set class path and suppress system path
-fCLASSPATH	Set class path
-I	Add directory to class path
-foutput-class-dir	Directory where class files should be written
-Wredundant-modifiers	Warn if modifiers are specified when not necessary
-Wunsupported-jdk11	Warn if ‘final’ local variables are specified

19.1.4 Target specific options:

Target specific options

-mno-stack-bias	Do not use stack bias
-mstack-bias	Use stack bias
-m64	Use 64-bit ABI
-m32	Use 32-bit ABI
-mptr32	Pointers are 32-bit
-mptr64	Pointers are 64-bit
-msupersparc	Optimize for SuperSparc processors
-mv8	Use V8 Sparc ISA
-mf934	Optimize for F934 processors
-mf930	Optimize for F930 processors
-msparclite	Optimize for SparcLite processors
-mcypress	Optimize for Cypress processors
-mno-vis	Do not utilize Visual Instruction Set
-mvis	Utilize Visual Instruction Set
-mno-v8plus	Do not compile for v8plus ABI
-mv8plus	Compile for v8plus ABI
-msoft-quad-float	Do not use hardware quad fp instructions
-mhard-quad-float	Use hardware quad fp instructions
-mno-app-regs	Do not use ABI reserved registers
-mapp-regs	Use ABI reserved registers
-mno-flat	Do not use flat register window model
-mflat	Use flat register window model
-mno-impure-text	Do not pass -assert pure-text to linker
-mimpure-text	Pass -assert pure-text to linker
-mno-unaligned-doubles	Assume all doubles are aligned
-munaligned-doubles	Assume possible double misalignment
-mno-epilogue	Do not use FUNCTION_EPILOGUE
-mepilogue	Use FUNCTION_EPILOGUE
-msoft-float	Do not use hardware fp
-msoft-float	Do not use hardware fp
-mhard-float	Use hardware fp
-mno-fpu	Do not use hardware fp
-mno-fpu	Do not use hardware fp
-mfpu	Use hardware fp
-malign-functions=	Function starts are aligned to this power of 2
-malign-jumps=	Jump targets are aligned to this power of 2
-malign-loops=	Loop code aligned to this power of 2
-mcmodel=	Use given Sparc code model
-mtune=	Schedule code for given CPU
-mcpu=	Use features of and schedule code for given CPU

19.1.5 Usage: gcc [options] file...

Options:

--help	Display this information
-dumpspecs	Display all of the built in spec strings
-dumpversion	Display the version of the compiler
-dumpmachine	Display the compiler's target processor
-print-search-dirs	Display the directories in the compiler's search path
-print-libgcc-file-name	Display the name of the compiler's companion library
-print-file-name=<lib>	Display the full path to library <lib>
-print-prog-name= <prog>	Display the full path to compiler component <prog>
-print-multi-directory	Display the root directory for versions of libgcc
-print-multi-lib	Display the mapping between command line options and multiple library search directories
-Wa,<options>	Pass comma-separated <options> on to the assembler
-Wp,<options>	Pass comma-separated <options> on to the preprocessor
-Wl,<options>	Pass comma-separated <options> on to the linker
-Xlinker <arg>	Pass <arg> on to the linker
-save-temps	Do not delete intermediate files
-pipe	Use pipes rather than intermediate files
-specs=<file>	Override builtin specs with the contents of <file>
-std=<standard>	Assume that the input sources are for <standard>
-B <directory>	Add <directory> to the compiler's search paths
-b <machine>	Run gcc for target <machine>, if installed
-V <version>	Run gcc version number <version>, if installed
-v	Display the programs invoked by the compiler
-E	Preprocess only; do not compile, assemble or link
-S	Compile only; do not assemble or link
-c	Compile and assemble, but do not link
-o <file>	Place the output into <file>
-x <language>	Specify the language of the following input files Permissable languages include: c c++ assembler none 'none' means revert to the default behaviour of guessing the language based on the file's extension

Options starting with -g, -f, -m, -O or -W are automatically passed on to the various sub-processes invoked by gcc. In order to pass other options on to these processes the -W<letter> options must be used.

19.2 - ld

Usage: ld [options] file...

19.2.1 Options

Options:

-a KEYWORD	Shared library control for HP/UX compatibility
-A ARCH, --architecture ARCH	Set architecture
-b TARGET, --format TARGET	Specify target for following input files
-c FILE, --mri-script FILE	Read MRI format linker script
-d, -dc, -dp	Force common symbols to be defined
-e ADDRESS, --entry ADDRESS	Set start address
-E, --export-dynamic	Export all dynamic symbols
-EB	Link big-endian objects
-EL	Link little-endian objects
-f SHLIB, --auxiliary SHLIB	Auxiliary filter for shared object symbol table
-F SHLIB, --filter SHLIB	Filter for shared object symbol table
-g	Ignored
-G SIZE, --gpsize SIZE	Small data size (if no size, same as --shared)
-h FILENAME, -soname FILENAME	Set internal name of shared library
-l LIBNAME, --library LIBNAME	Search for library LIBNAME
-L DIRECTORY, --library-path DIRECTORY	Add DIRECTORY to library search path
-m EMULATION	Set emulation
-M, --print-map	Print map file on standard output
-n, --nmagic	Do not page align data
-N, --omagic	Do not page align data, do not make text readonly
-o FILE, --output FILE	Set output file name

Options:

-O	Optimize output file
-Qy	Ignored for SVR4 compatibility
-q, --emit-relocs	Generate relocations in final output
-r, -i, --relocateable	Generate relocateable output
-R FILE, --just-symbols FILE	Just link symbols (if directory, same as --rpath)
-s, --strip-all	Strip all symbols
-S, --strip-debug	Strip debugging symbols
-t, --trace	Trace file opens
-T FILE, --script FILE	Read linker script
-u SYMBOL, --undefined SYMBOL	Start with undefined reference to SYMBOL
--unique [=SECTION]	Don't merge input [SECTION | orphan] sections
-Ur	Build global constructor/destructor tables
-v, --version	Print version information
-V	Print version and emulation information
-x, --discard-all	Discard all local symbols
-X, --discard-locals	Discard temporary local symbols (default)
--discard-none	Don't discard any local symbols
-y SYMBOL, --trace-symbol SYMBOL	Trace mentions of SYMBOL
-Y PATH	Default search path for Solaris compatibility
-(, --start-group	Start a group
-), --end-group	End a group
-assert KEYWORD	Ignored for SunOS compatibility
-Bdynamic, -dy, -call_shared	Link against shared libraries
-Bstatic, -dn, -non_shared, -static	Do not link against shared libraries
-Bsymbolic	Bind global references locally
--check-sections	Check section addresses for overlaps (default)
--no-check-sections	Do not check section addresses for overlaps

Options:

--cref	Output cross reference table
--defsym SYMBOL=EXPRESSION	Define a symbol
--demangle [=STYLE]	Demangle symbol names [using STYLE]
--dynamic-linker PROGRAM	Set the dynamic linker to use
--embedded-relocs	Generate embedded relocs
-fini SYMBOL	Call SYMBOL at unload-time
--force-exe-suffix	Force generation of file with .exe suffix
--gc-sections	Remove unused sections (on some targets)
--no-gc-sections	Don't remove unused sections (default)
--help	Print option help
-init SYMBOL	Call SYMBOL at load-time
-Map FILE	Write a map file
--no-demangle	Do not demangle symbol names
--no-keep-memory	Use less memory and more disk I/O
--no-undefined	Allow no undefined symbols
--allow-shlib-undefined	Allow undefined symbols in shared objects
--no-warn-mismatch	Don't warn about mismatched input files
--no-whole-archive	Turn off --whole-archive
--noinhibit-exec	Create an output file even if errors occur
--oformat TARGET	Specify target of output file
-qmagic	Ignored for Linux compatibility
--relax	Relax branches on certain targets
--retain-symbols-file FILE	Keep only symbols listed in FILE
-rpath PATH	Set runtime shared library search path
-rpath-link PATH	Set link time shared library search path
-shared, -Bshareable	Create a shared library
--sort-common	Sort common symbols by size
--split-by-file [=SIZE]	Split output sections every SIZE octets
--split-by-reloc [=COUNT]	Split output sections every COUNT relocs
--stats	Print memory usage statistics
--target-help	Display target specific options
--task-link SYMBOL	Do task level linking
--traditional-format	Use same format as native linker
--section-start SECTION=ADDRESS	Set address of named section
-Tbss ADDRESS	Set address of .bss section
-Tdata ADDRESS	Set address of .data section
-Ttext ADDRESS	Set address of .text section
--verbose	Output lots of information during link
--version-script FILE	Read version information script
--version-exports-section SYMBOL	Take export symbols list from .exports, using SYMBOL as the version.

Options:

--warn-common	Warn about duplicate common symbols
--warn-constructors	Warn if global constructors/destructors are seen
--warn-multiple-gp	Warn if the multiple GP values are used
--warn-once	Warn only once per undefined symbol
--warn-section-align	Warn if start of section changes due to alignment
--fatal-warnings	Treat warnings as errors
--whole-archive	Include all objects from following archives
--wrap SYMBOL	Use wrapper functions for SYMBOL
--mpc860c0 [=WORDS]	Modify problematic branches in last WORDS (1-10, default 5) words of a page

ld: supported targets: pe-i386 pei-i386 elf32-i386 elf32-little elf32-big srec symbolsrec tekhex binary ihex
ld: supported emulations: i386pe

19.2.2 emulation specific options

i386pe:

--base_file <basefile>	Generate a base file for relocatable DLLs
--dll	Set image base to the default for DLLs
--file-alignment <size>	Set file alignment
--heap <size>	Set initial size of the heap
--image-base <address>	Set start address of the executable
--major-image-version <number>	Set version number of the executable
--major-os-version <number>	Set minimum required OS version
--major-subsystem-version <number>	Set minimum required OS subsystem version
--minor-image-version <number>	Set revision number of the executable
--minor-os-version <number>	Set minimum required OS revision
--minor-subsystem-version <number>	Set minimum required OS subsystem revision
--section-alignment <size>	Set section alignment
--stack <size>	Set size of the initial stack
--subsystem <name>[:<version>]	Set required OS subsystem [& version]
--support-old-code	Support interworking with old code
--thumb-entry=<symbol>	Set the entry point to be Thumb <symbol>
--add-stdcall-alias	Export symbols with and without @nn
--disable-stdcall-fixup	Don't link _sym to _sym@nn
--enable-stdcall-fixup	Link _sym to _sym@nn without warnings
--exclude-symbols sym,sym,...	Exclude symbols from automatic export
--export-all-symbols	Automatically export all globals to DLL
--kill-at	Remove @nn from exported symbols
--out-implib <file>	Generate import library
--output-def <file>	Generate a .DEF file for the built DLL
--warn-duplicate-exports	Warn about duplicate exports.
--compat-implib	Create backward compatible import libs; create __imp_<SYMBOL> as well.
--enable-auto-image-base	Automatically choose image base for DLLs unless user specifies one
--disable-auto-image-base	Do not auto-choose image base. (default)
--dll-search-prefix=<string>	When linking dynamically to a dll witout an importlib, use <string><basename>.dll in preference to lib<basename>.dll

19.3 - gdb commands

This chapter is a gathering of all the explanations you can get from the help command in gdb.

19.3.1 Aliases

Aliases of other commands.

19.3.1.1 delete breakpoints

Delete some breakpoints or auto-display expressions.

Arguments are breakpoint numbers with spaces in between.

To delete all breakpoints, give no argument.

This command may be abbreviated "delete".

19.3.1.2 disable breakpoints

Disable some breakpoints.

Arguments are breakpoint numbers with spaces in between.

To disable all breakpoints, give no argument.

A disabled breakpoint is not forgotten, but has no effect until reenabled.

This command may be abbreviated "disable".

19.3.1.3 ni

Step one instruction, but proceed through subroutine calls.

Argument N means do this N times (or till program stops for another reason).

19.3.1.4 si

Step one instruction exactly.

Argument N means do this N times (or till program stops for another reason).

19.3.1.5 where

Print backtrace of all stack frames.

Step one instruction exactly.

Argument N means do this N times (or till program stops for another reason).

19.3.2 Breakpoints

Making program stop at certain points.

19.3.2.1 awatch

Set a watchpoint for an expression.

A watchpoint stops execution of your program whenever the value of

an expression is either read or written.

19.3.2.2 break

Set breakpoint at specified line or function.

Argument may be line number, function name, or "*" and an address. If line number is specified, break at start of code for that line. If function is specified, break at start of code for that function. If an address is specified, break at that exact address. With no arg, uses current execution address of selected stack frame. This is useful for breaking on return to a stack frame. Multiple breakpoints at one place are permitted, and useful if conditional.

19.3.2.3 catch

Set breakpoints to catch exceptions that are raised. Argument may be a single exception to catch, multiple exceptions to catch, or the default exception "default". If no arguments are given, breakpoints are set at all exception handlers catch clauses within the current scope.

A condition specified for the catch applies to all breakpoints set with this command

19.3.2.4 clear

Clear breakpoint at specified line or function.

Argument may be line number, function name, or "*" and an address. If line number is specified, all breakpoints in that line are cleared. If function is specified, breakpoints at beginning of function are cleared. If an address is specified, breakpoints at that address are cleared.

With no argument, clears all breakpoints in the line that the selected frame

is executing in.

See also the "delete" command which clears breakpoints by number.

19.3.2.5 commands

Set commands to be executed when a breakpoint is hit. Give breakpoint number as argument after "commands". With no argument, the targeted breakpoint is the last one set. The commands themselves follow starting on the next line. Type a line containing "end" to indicate the end of them. Give "silent" as the first line to make the breakpoint silent; then no output is printed when it is hit, except what the commands print.

19.3.2.6 condition

Specify breakpoint number N to break only if COND is true. Usage is `condition N COND', where N is an integer and COND is an expression to be evaluated whenever breakpoint N is reached.

19.3.2.7 delete

Delete some breakpoints or auto-display expressions. Arguments are breakpoint numbers with spaces in between. To delete all breakpoints, give no argument. Also a prefix

command for deletion of other GDB objects. The "unset" command is also an alias for "delete".

List of delete subcommands:

delete breakpoints -- Delete some breakpoints or auto-display expressions

Arguments are breakpoint numbers with spaces in between. To delete all breakpoints, give no argument. This command may be abbreviated "delete".

delete display -- Cancel some expressions to be displayed when program stops Cancel some expressions to be displayed when program stops. Arguments are the code numbers of the expressions to stop displaying. No argument means cancel all automatic-display expressions. Do "info display" to see current list of code numbers.

19.3.2.8 disable

Disable some breakpoints.

Arguments are breakpoint numbers with spaces in between. To disable all breakpoints, give no argument. A disabled breakpoint is not forgotten, but has no effect until reenabled.

List of disable subcommands:

disable breakpoints -- Disable some breakpoints

Arguments are breakpoint numbers with spaces in between. To disable all breakpoints, give no argument. A disabled breakpoint is not forgotten, but has no effect until reenabled. This command may be abbreviated "disable".

disable display -- Disable some expressions to be displayed when program stops

Arguments are the code numbers of the expressions to stop displaying. No argument means disable all automatic-display expressions. Do "info display" to see current list of code numbers.

19.3.2.9 enable

Enable some breakpoints.

Give breakpoint numbers (separated by spaces) as arguments. With no subcommand, breakpoints are enabled until you command otherwise. This is used to cancel the effect of the "disable" command. With a subcommand you can enable temporarily.

List of enable subcommands:

enable delete -- Enable breakpoints and delete when hit

Enable breakpoints and delete when hit. Give breakpoint numbers. If a breakpoint is hit while enabled in this fashion, it is deleted.

enable display -- Enable some expressions to be displayed when program stops

Enable some expressions to be displayed when program stops. Arguments are the code numbers of the expressions to resume displaying. No argument means enable all automatic-display expressions. Do "info display" to see current list of code numbers.

enable keep -- Enable breakpoints for normal operation

Enable breakpoints for normal operation. Give breakpoint numbers. This cancels the effect of "enable once" or "enable delete".

enable once -- Enable breakpoints for one hit

Enable breakpoints for one hit. Give breakpoint numbers. If a breakpoint is hit while enabled in this fashion, it becomes disabled.

19.3.2.10 gbreak

Set a global breakpoint. Args like "break" command.

Like "break" except the breakpoint applies to all tasks.

19.3.2.11 hbreak

Set a hardware assisted breakpoint. Args like "break" command.

Like "break" except the breakpoint requires hardware support,

some target hardware may not have this support.

19.3.2.12 ignore

Set ignore-count of breakpoint number N to COUNT.

Usage is `ignore N COUNT'.

19.3.2.13 obreak

Set a one-time breakpoint. Args like "break" command. Like "break" except the breakpoint will be disabled when hit. Equivalent to "break" followed by using "enable once" on the breakpoint number.

19.3.2.14 ohbreak

Set a one-time hardware assisted breakpoint. Args like "break" command. Like "hbreak" except the breakpoint will be disabled when hit. Equivalent to "hbreak" followed by using "enable once" on the breakpoint number.

19.3.2.15 rbreak

Set a breakpoint for all functions matching REGEXP.

19.3.2.16 rwatch

Set a read watchpoint for an expression. A watchpoint stops execution of your program whenever the value of an expression is read.

19.3.2.17 tbreak

Set a temporary breakpoint. Args like "break" command. Like "break" except the breakpoint is only temporary, so it will be deleted when hit. Equivalent to "break" followed by using "enable delete" on the breakpoint number.

19.3.2.18 thbreak

Set a temporary hardware assisted breakpoint. Args like "break" command. Like "hbreak" except the breakpoint is only temporary, so it will be deleted when hit.

19.3.2.19 watch

Set a watchpoint for an expression. A watchpoint stops execution of your program whenever the value of an expression changes.

19.3.3 Examining data

19.3.3.1 call

Call a function in the program.

The argument is the function name and arguments, in the notation of the current working language. The result is printed and saved in the value history, if it is not void.

19.3.3.2 delete display

Cancel some expressions to be displayed when program stops. Arguments are the code numbers of the expressions to stop displaying. No argument means cancel all automatic-display expressions. Do "info display" to see current list of code numbers.

19.3.3.3 disable display

Disable some expressions to be displayed when program stops. Arguments are the code numbers of the expressions to stop displaying. No argument means disable all automatic-display expressions. Do "info display" to see current list of code numbers.

19.3.3.4 disassemble

Disassemble a specified section of memory. Default is the function surrounding the pc of the selected frame. With a single argument, the function surrounding that address is dumped. Two arguments are taken as a range of memory to dump.

19.3.3.5 display

Print value of expression EXP each time the program stops.

/FMT may be used before EXP as in the "print" command.

/FMT "i" or "s" or including a size-letter is allowed,

as in the "x" command, and then EXP is used to get the address to examine and examining is done as in the "x" command.

With no argument, display all currently requested auto-display expressions. Use "undisplay" to cancel display requests previously made.

19.3.3.6 enable display

Enable some expressions to be displayed when program stops. Arguments are the code numbers of the expressions to resume displaying. No argument means enable all automatic-display expressions. Do "info display" to see current list of code numbers.

19.3.3.7 inspect

Same as "print" command, except that if you are running in the epoch environment, the value is printed in its own window.

19.3.3.8 output

Like "print" but don't put in value history and don't print newline

Like "print" but don't put in value history and don't print newline. This is useful in user-defined commands.

19.3.3.9 print

Print value of expression EXP

Variables accessible are those of the lexical environment of the selected stack frame, plus all those whose scope is global or an entire file.

$NUM gets previous value number NUM. $ and $$ are the last two values. $$NUM refers to NUM'th value back from the last one. Names starting with $ refer to registers (with the values they would have if the program were to return to the stack frame now selected, restoring all registers saved by frames farther in) or else to debugger "convenience" variables (any such name not a known register). Use assignment expressions to give values to convenience variables.

{TYPE}ADREXP refers to a datum of data type TYPE, located at address ADREXP.

@ is a binary operator for treating consecutive data objects anywhere in memory as an array.

FOO@NUM gives an array whose first element is FOO, whose second element is stored in the space following where FOO is stored, etc. FOO must be an expression whose value resides in memory.

EXP may be preceded with /FMT, where FMT is a format letter but no count or size letter (see "x" command).

19.3.3.10 printf

printf "printf format string", arg1, arg2, arg3, ..., argn

This is useful for formatted output in user-defined commands.

19.3.3.11 ptype

Print definition of type TYPE.

Argument may be a type name defined by typedef, or "struct STRUCT-TAG" or "class CLASS-NAME" or "union UNION-TAG" or "enum ENUM-TAG".

The selected stack frame's lexical context is used to look up the name.

19.3.3.12 reformat

Change /FMT of expression CODENUM in the "display" list.

This is a shortcut to avoid using a new code number for the same expression.

19.3.3.13 set

Evaluate expression EXP and assign result to variable VAR, using assignment syntax appropriate for the current language (VAR = EXP or VAR := EXP for example). VAR may be a debugger "convenience" variable (names starting with $), a register (a few standard names starting with $), or an actual variable in the program being debugged. EXP is any valid expression.

Use "set variable" for variables with names identical to set subcommands.

With a subcommand, this command modifies parts of the gdb environment.

You can see these environment settings with the "show" command.

List of set subcommands:

set annotate -- Set annotation_level

0 == normal

1 == fullname (for use when running under emacs)

2 == output annotated suitably for use by programs that control GDB.

set architecture -- Set architecture of target

set args -- Set argument list to give program being debugged when it is started

Follow this command with any number of args, to be passed to the program.

set assembly-language -- Set x86 instruction set to use for disassembly

This value can be set to either i386 or i8086 to change how instructions are disassembled.

set check -- Set the status of the type/range checker

List of set check subcommands:

set check range -- Set range checking (on/warn/off/auto)

set check type -- Set type checking (on/warn/off/auto)

set complaints -- Set max number of complaints about incorrect symbols

set demangle-style -- Set the current C++ demangling style

set editing -- Set editing of command lines as they are typed

set environment -- Set environment variable value to give the program

set gnutarget -- Set the current BFD target

set height -- Set number of lines gdb thinks are in a page

set history -- Generic command for setting command history parameters

set inhibit-gdbinit -- Set whether gdb reads the gdbinit files

set input-radix -- Set default input radix for entering numbers

set language -- Set the current source language

set listsize -- Set number of source lines gdb will list by default

set longjmp-breakpoint-enable -- Set internal breakpoints to handle longjmp() properly

set output-radix -- Set default output radix for printing of values

set prefetch-mem-enable -- Set whether to enable target memory prefetching

set print -- Generic command for setting how things print

set prompt -- Set gdb's prompt

set radix -- Set default input and output number radices

set remotebaud -- Set baud rate for remote serial I/O

set remotedebug -- Set debugging of remote protocol

set remotelogbase -- Set

set remotelogfile -- Set filename for remote session recording

set remotetimeout -- Set timeout limit to wait for target to respond

set symbol-readnow -- Set immediate reading of full symbol table data

set targetdebug -- Set target debugging

set unsettable-breakpoint-autodisable -- Set automatic disabling of unsettable breakpoints

set variable -- Evaluate expression EXP and assign result to variable VAR

set verbose -- Set verbosity

set watchdog -- Set watchdog timer

set width -- Set number of characters gdb thinks are in a line

set write -- Set writing into executable and core files

set wrs-detach-behavior -- Set whether to quietly detach tasks in target/attach/quit

set wtx-event-debug-print -- Set whether to print debug messages in the new event handling code

set wtx-gui-bp2-message -- Set whether to send bp2 messages to the Tornado GUI

set wtx-ignore-exit-status -- Set whether the exit status of the debugged task is ignored (assumed zero)

set wtx-load-flags -- Set load flags used when loading a new object module

set wtx-load-path-qualify -- Set passing of full object path to target

set wtx-load-timeout -- Set timeout in seconds used when loading new objects on target

set wtx-new-target-message -- Set whether to use the new target message format

set wtx-order-debug-print -- Set whether to print orders issued to the inferior

set wtx-override-configuration-check -- Set override checking Gdb configuration

set wtx-task-priority -- Set priority of tasks created using "run" command

set wtx-task-stack-size -- Set stack size (in bytes) of tasks created using "run" command

set wtx-tool-name -- Set tool name used by debugger when connecting to the target server

set xfer-mem-debug-print -- Set whether to enable target_xfer_memory debug print

19.3.3.14 tclprint

Same as "print" command, except that results are formatted appropriately for use as a Tcl_DString, and nothing is added to the value history.

19.3.3.15 undisplay

Cancel some expressions to be displayed when program stops.

Arguments are the code numbers of the expressions to stop displaying.

No argument means cancel all automatic-display expressions.

"delete display" has the same effect as this command.

Do "info display" to see current list of code numbers.

19.3.3.16 whatis

Print data type of expression EXP.

19.3.3.17 x

Examine memory: x/FMT ADDRESS.

ADDRESS is an expression for the memory address to examine.

FMT is a repeat count followed by a format letter and a size letter.

Format letters are o(octal), x(hex), d(decimal), u(unsigned decimal), t(binary), f(float), a(address), i(instruction), c(char) and s(string). Size letters are b(byte), h(halfword), w(word), g(giant, 8 bytes). The specified number of objects of the specified size are printed according to the format.

Defaults for format and size letters are those previously used.

Default count is 1. Default address is following last thing printed with this command or "print".

19.3.4 Files

Specifying and examining files

add-shared-symbol-files -- Load the symbols from shared objects in the dynamic linker's link map

add-symbol-file -- Load the symbols from FILE

cd -- Set working directory to DIR for debugger and program being debugged

core-file -- Use FILE as core dump for examining memory and registers

directory -- Add directory DIR to beginning of search path for source files

exec-file -- Use FILE as program for getting contents of pure memory

file -- Use FILE as program to be debugged

forward-search -- Search for regular expression (see regex(3)) from last line listed

list -- List specified function or line

load -- Dynamically load FILE into the running program

path -- Add directory DIR(s) to beginning of search path for object files

pwd -- Print working directory

reverse-search -- Search backward for regular expression (see regex(3)) from last line listed

search -- Search for regular expression (see regex(3)) from last line listed

section -- Change the base address of section SECTION of the exec file to ADDR

set gnutarget -- Set the current BFD target

show gnutarget -- Show the current BFD target

symbol-file -- Load symbol table from executable file FILE

unload -- Unload and close an object module that is currently being debugged

19.3.5 Internals

Maintenance commands

Some gdb commands are provided just for use by gdb maintainers.
These commands are subject to frequent change, and may not be as
well documented as user commands.

List of commands:

maintenance -- Commands for use by GDB maintainers

maintenance check-blockranges -- Check consistency of block address ranges within each blockvector

maintenance check-symbols -- Check consistency of addresses between minsyms and symbols

maintenance check-symtabs -- Check consistency of psymtabs and symtabs

maintenance demangle -- Demangle a C++ mangled name

maintenance dump-eventpoints -- Dump internal list of WTX eventpoints

maintenance dump-sections -- Dump interesting info about the sections of OBJFILE

maintenance info -- Commands for showing internal info about the program being debugged

maintenance info breakpoints -- Status of all breakpoints

maintenance info gbreakpoints -- Like "maintenance info breakpoints" but with F/G flags

maintenance info sections -- List the BFD sections of the exec and core files

maintenance print -- Maintenance command for printing GDB internal state

maintenance print msymbols -- Print dump of current minimal symbol definitions

maintenance print objfiles -- Print dump of current object file definitions

maintenance print psymbols -- Print dump of current partial symbol definitions

maintenance print statistics -- Print statistics about internal gdb state

maintenance print symbols -- Print dump of current symbol definitions

maintenance print type -- Print a type chain for a given symbol

maintenance space -- Set the display of space usage

maintenance time -- Set the display of time usage

maintenance translate-address -- Translate a section name and address to a symbol

set targetdebug -- Set target debugging

set watchdog -- Set watchdog timer

show targetdebug -- Show target debugging

show watchdog -- Show watchdog timer

19.3.6 Obscure features

List of commands:

complete -- List the completions for the rest of the line as a command

divert -- Run the supplied command

printdiversion -- Print the contents of the diversion buffer

set annotate -- Set annotation_level

set assembly-language -- Set x86 instruction set to use for disassembly

set longjmp-breakpoint-enable -- Set internal breakpoints to handle longjmp() properly

set prefetch-mem-enable -- Set whether to enable target memory prefetching

set unsettable-breakpoint-autodisable -- Set automatic disabling of unsettable breakpoints

set wtx-event-debug-print -- Set whether to print debug messages in the new event handling code

set wtx-gui-bp2-message -- Set whether to send bp2 messages to the Tornado GUI

set wtx-ignore-exit-status -- Set whether the exit status of the debugged task is ignored (assumed zero)

set wtx-load-flags -- Set load flags used when loading a new object module

set wtx-load-path-qualify -- Set passing of full object path to target

set wtx-load-timeout -- Set timeout in seconds used when loading new objects on target

set wtx-new-target-message -- Set whether to use the new target message format

set wtx-order-debug-print -- Set whether to print orders issued to the inferior

set wtx-override-configuration-check -- Set override checking Gdb configuration

set wtx-task-priority -- Set priority of tasks created using "run" command

set wtx-task-stack-size -- Set stack size (in bytes) of tasks created using "run" command

set wtx-tool-name -- Set tool name used by debugger when connecting to the target server

set xfer-mem-debug-print -- Set whether to enable target_xfer_memory debug printfs

show annotate -- Show annotation_level

show assembly-language -- Show x86 instruction set to use for disassembly

show longjmp-breakpoint-enable -- Show internal breakpoints to handle longjmp() properly

show prefetch-mem-enable -- Show whether to enable target memory prefetching

show unsettable-breakpoint-autodisable -- Show automatic disabling of unsettable breakpoints

show wtx-event-debug-print -- Show whether to print debug messages in the new event handling code

show wtx-gui-bp2-message -- Show whether to send bp2 messages to the Tornado GUI

show wtx-ignore-exit-status -- Show whether the exit status of the debugged task is ignored (assumed zero)

show wtx-load-flags -- Show load flags used when loading a new object module

show wtx-load-path-qualify -- Show passing of full object path to target

show wtx-load-timeout -- Show timeout in seconds used when loading new objects on target

show wtx-new-target-message -- Show whether to use the new target message format

show wtx-order-debug-print -- Show whether to print orders issued to the inferior

show wtx-override-configuration-check -- Show override checking Gdb configuration

show wtx-task-priority -- Show priority of tasks created using "run" command

show wtx-task-stack-size -- Show stack size (in bytes) of tasks created using "run" command

show wtx-tool-name -- Show tool name used by debugger when connecting to the target server

show xfer-mem-debug-print -- Show whether to enable target_xfer_memory debug printfs

stop -- There is no `stop' command

tcl -- Pass the argument string to the Tcl interpreter

tcldebug -- Toggle printing of Tcl requests to GDB

tclerror -- Toggle printing of verbose Tcl error information

tclproc -- Attach the name of a Tcl procedure to the name of a GDB command

19.3.7 Running the program

19.3.7.1 attach

Attach to a process or file outside of GDB.

This command attaches to another target, of the same type as your last `target' command (`info files' will show your target stack). The command may take as argument a process id or a device file. For a process id, you must have permission to send the process a signal, and it must have the same effective uid as the debugger. When using "attach", you should use the "file" command to specify the program running in the process, and to load its symbol table.

19.3.7.2 continue

Continue program being debugged.

Continue program being debugged, after signal or breakpoint. If proceeding from breakpoint, a number N may be used as an argument, which means to set the ignore count of that breakpoint to N - 1 (so that the breakpoint won't break until the Nth time it is reached).

19.3.7.3 detach

Detach a process or file previously attached.

If a process, it is no longer traced, and it continues its execution. If you were debugging a file, the file is closed and gdb no longer accesses it.

19.3.7.4 finish

Execute until selected stack frame returns.

Upon return, the value returned is printed and put in the value history.

19.3.7.5 halt

Halt program being debugged.

19.3.7.6 handle

Specify how to handle a signal.

Args are signals and actions to apply to those signals. Symbolic signals (e.g. SIGSEGV) are recommended but numeric signals from 1-15 are allowed for compatibility with old versions of GDB. Numeric ranges may be specified with the form LOW-HIGH (e.g. 1-5). The special arg "all" is recognized to mean all signals except those used by the debugger, typically SIGTRAP and SIGINT. Recognized actions include "stop", "nostop", "print", "noprint", "pass", "nopass", "ignore", or "noignore". Stop means reenter debugger if this signal happens (implies print). Print means print a message if this signal happens. Pass means let program see this signal; otherwise program doesn't know. Ignore is a synonym for nopass and noignore is a synonym for pass. Pass and Stop may be combined.

19.3.7.7 idle

Perform one iteration of the debugger's idle processing. Normally invoked internally, but may be used manually for testing.

19.3.7.8 info handle

What debugger does when program gets various signals.

Specify a signal as argument to print info on that signal only.

19.3.7.9 jump

Continue program being debugged at specified line or address. Give as argument either LINENUM or *ADDR, where ADDR is an expression for an address to start at.

19.3.7.10 kill

Kill execution of program being debugged.

19.3.7.11 next

Step program, proceeding through subroutine calls. Like the "step" command as long as subroutine calls do not happen; when they do, the call is treated as one instruction. Argument N means do this N times (or till program stops for another reason).

19.3.7.12 nexti

Step one instruction, but proceed through subroutine calls. Argument N means do this N times (or till program stops for another reason).

19.3.7.13 run

Start debugged program.

Start debugged program. You may specify arguments to give it. Args may include "*", or "[...]"; they are expanded using "sh". Input and output redirection with ">", "<", or ">>" are also allowed.

With no arguments, uses arguments last specified (with "run" or "set args"). To cancel previous arguments and run with no arguments, use "set args" without arguments.

19.3.7.14 sattach

Leave current task suspended, and attach to another. Effect is that of "sdetach" followed by "attach".

19.3.7.15 sdetach

Detach current task, leaving it suspended. Effect is that of "detach 1".

19.3.7.16 set args

Set argument list to give program being debugged when it is started.

Follow this command with any number of args, to be passed to the program.

19.3.7.17 set environment

Set environment variable value to give the program.

Arguments are VAR VALUE where VAR is variable name and VALUE is value. VALUES of environment variables are uninterpreted strings. This does not affect the program until the next "run" command.

19.3.7.18 show args

Show argument list to give program being debugged when it is started.

Follow this command with any number of args, to be passed to the program.

19.3.7.19 signal

Continue program giving it signal specified by the argument.

An argument of "0" means continue program without giving it a signal.

19.3.7.20 step

Step program until it reaches a different source line.

Argument N means do this N times (or till program stops for another reason).

19.3.7.21 stepi

Step one instruction exactly.

Argument N means do this N times (or till program stops for another reason).

19.3.7.22 target

Connect to a target machine or process.

The first argument is the type or protocol of the target machine. Remaining arguments are interpreted by the target protocol. For more information on the arguments for a particular protocol, type `help target ' followed by the protocol name.

List of target subcommands:

target wtx -- Remote target connected via the Wind River Tool eXchange (WTX) protocol

Remote target connected via the Wind River Tool eXchange (WTX) protocol. Specify the name of the WTX target server connected to the target as the argument. Once connected to the target server you can access target system memory and download object files, as well as use the "run" command to spawn a task you wish to debug or "attach" command to debug an already existing task.

target wtxsystem -- System running on a remote target connected to the host

System running on a remote target connected to the host via the Wind River Tool eXchange (WTX) protocol. To connect to the system first establish a connection to the target by giving the command "target wtx" specifying as an argument the WTX target server name. Then use the "attach system" command to enter system debugging mode.

target wtxtask -- Task running on a remote target system connected to the host

Task running on a remote target system connected to the host via the Wind River Tool eXchange (WTX) protocol. To spawn a task, first establish a connection to the target by giving the command "target wtx" specifying as an argument the WTX target server name. Then use the "run" command to spawn the task, supplying the name of the task entry point and arguments to the task (if any).

19.3.7.23 thread

Use this command to switch between threads.

The new thread ID must be currently known.

19.3.7.24 thread apply

Apply a command to a list of threads.

19.3.7.25 apply all

Apply a command to all threads.

19.3.7.26 tty

Set terminal for future runs of program being debugged.

19.3.7.27 unset environment

Cancel environment variable VAR for the program.

This does not affect the program until the next "run" command.

19.3.7.28 untarget

Undo the effect of previous "target" commands.

19.3.7.29 until

Execute until the program reaches a source line greater than the current or a specified line or address or function (same args as break command). Execution will also stop upon exit from the current stack frame.

19.3.8 Examining the stack

The stack is made up of stack frames. Gdb assigns numbers to stack frames counting from zero for the innermost (currently executing) frame.

At any time gdb identifies one frame as the "selected" frame. Variable lookups are done with respect to the selected frame. When the program being debugged stops, gdb selects the innermost frame. The commands below can be used to select other frames by number or address.

19.3.8.1 backtrace

Print backtrace of all stack frames, or innermost COUNT frames. With a negative argument, print outermost -COUNT frames.

19.3.8.2 bt

Print backtrace of all stack frames, or innermost COUNT frames. With a negative argument, print outermost -COUNT frames.

19.3.8.3 down

Select and print stack frame called by this one.

An argument says how many frames down to go.

19.3.8.4 frame

Select and print a stack frame.

With no argument, print the selected stack frame. (See also "info frame"). An argument specifies the frame to select. It can be a stack frame number or the address of the frame. With argument, nothing is printed if input is coming from a command file or a user-defined command.

19.3.8.5 pptype

Exercise the GUI protocol type formatter.

19.3.8.6 ppval

Exercise the GUI protocol value formatter.

19.3.8.7 return

Make selected stack frame return to its caller.

Control remains in the debugger, but when you continue execution will resume in the frame above the one now selected. If an argument is given, it is an expression for the value to return.

19.3.8.8 select-frame

Select a stack frame without printing anything.

An argument specifies the frame to select. It can be a stack frame number or the address of the frame.

19.3.8.9 up

Select and print stack frame that called this one.

An argument says how many frames up to go.

19.3.9 Status inquiries

List of commands:

info -- Generic command for showing things about the program being debugged

info address -- Describe where symbol SYM is stored

info all-registers -- List of all registers and their contents

info architecture -- List supported target architectures

info args -- Argument variables of current stack frame

info breakpoints -- Status of user-settable breakpoints

Status of user-settable breakpoints, or breakpoint number NUMBER.

The "Type" column indicates one of:

breakpoint - normal breakpoint

watchpoint - watchpoint

The "Disp" column contains one of "keep", "del", or "dis" to indicate the disposition of the breakpoint after it gets hit. "dis" means that the breakpoint will be disabled. The "Address" and "What" columns indicate the address and file/line number respectively.

Convenience variable "$_" and default examine address for "x" are set to the address of the last breakpoint listed.

Convenience variable "$bpnum" contains the number of the last breakpoint set.

info catch -- Exceptions that can be caught in the current stack frame

info common -- Print out the values contained in a Fortran COMMON block

info copying -- Conditions for redistributing copies of GDB

info display -- Expressions to display when program stops

info files -- Names of targets and files being debugged

info float -- Print the status of the floating point unit

info frame -- All about selected stack frame
All about selected stack frame, or frame at ADDR.

info functions -- All function names
info gbreakpoints -- Like "info breakpoints" but with F/G flags
info handle -- What debugger does when program gets various signals

info line -- Core addresses of the code for a source line
Line can be specified as
  LINENUM, to list around that line in current file,
  FILE:LINENUM, to list around that line in that file,
  FUNCTION, to list around beginning of that function,
  FILE:FUNCTION, to distinguish among like-named static functions.
Default is to describe the last source line that was listed. This sets the default address for "x" to the line's first instruction so that "x/i" suffices to start examining the machine code. The address is also stored as the value of "$_".

info locals -- Local variables of current stack frame

info prefetch-registers -- List of integer registers and their contents
info program -- Execution status of the program

info prototype -- Prototype of function containing the specified source line
Any argument that works with "info line" may be used.

info registers -- List of integer registers and their contents
info set -- Show all GDB settings
info signals -- What debugger does when program gets various signals
info source -- Information about the current source file

info sources -- Source files in the program

info stack -- Backtrace of the stack
Backtrace of the stack, or innermost COUNT frames.

info symbol -- Describe what symbol is at location ADDR
Only for symbols with fixed locations (global or static scope).

info tags -- List various sets of names relevant to the current stack frame
Names are printed one per line, with no annotation, for machine-readability.
 func-args - arguments
 func-autos - automatic variables on the stack
 func-locals - union of func-args, func-autos, func-regs, and func-statics
 func-regs - automatic variables in registers
 func-statics - static variables local to the current function
 file-statics - static variables local to the current file
 file-globals - static variables global to the entire program

info target -- Names of targets and files being debugged
info taskname -- Display the name associated with a given task ID

info terminal -- Print inferior's saved terminal status

info threads -- IDs of currently known threads

info types -- All type names

info variables -- All global and static variable names
All global and static variable names, or those matching REGEXP.

info warranty -- Various kinds of warranty you do not have
info watchpoints -- Synonym for ``info breakpoints''

show -- Generic command for showing things about the debugger
show annotate -- Show annotation_level
show architecture -- Show architecture of target

show args -- Show argument list to give program being debugged when it is started
show assembly-language -- Show x86 instruction set to use for disassembly
show check -- Show the status of the type/range checker
show commands -- Show the history of commands you typed
show complaints -- Show max number of complaints about incorrect symbols
show convenience -- Debugger convenience ("$foo") variables
show copying -- Conditions for redistributing copies of GDB
show demangle-style -- Show the current C++ demangling style
show directories -- Current search path for finding source files
show editing -- Show editing of command lines as they are typed
show environment -- The environment to give the program
show gnutarget -- Show the current BFD target
show height -- Show number of lines gdb thinks are in a page
show history -- Generic command for showing command history parameters
show inhibit-gdbinit -- Show whether gdb reads the gdbinit files
show input-radix -- Show default input radix for entering numbers
show language -- Show the current source language
show listsize -- Show number of source lines gdb will list by default
show longjmp-breakpoint-enable -- Show internal breakpoints to handle longjmp() properly
show output-radix -- Show default output radix for printing of values
show paths -- Current search path for finding object files
show prefetch-mem-enable -- Show whether to enable target memory prefetching
show print -- Generic command for showing print settings
show prompt -- Show gdb's prompt
show radix -- Show the default input and output number radices
show remotebaud -- Show baud rate for remote serial I/O
show remotedebug -- Show debugging of remote protocol
show remotelogbase -- Show
show remotelogfile -- Show filename for remote session recording
show remotetimeout -- Show timeout limit to wait for target to respond
show symbol-readnow -- Show immediate reading of full symbol table data
show targetdebug -- Show target debugging

show unsettable-breakpoint-autodisable -- Show automatic disabling of unsettable breakpoints

show user -- Show definitions of user defined commands

show values -- Elements of value history around item number IDX (or last ten)

show verbose -- Show verbosity

show version -- Show what version of GDB this is

show warranty -- Various kinds of warranty you do not have

show watchdog -- Show watchdog timer

show width -- Show number of characters gdb thinks are in a line

show write -- Show writing into executable and core files

show wrs-detach-behavior -- Show whether to quietly detach tasks in target/attach/quit

show wtx-event-debug-print -- Show whether to print debug messages in the new event handling code

show wtx-gui-bp2-message -- Show whether to send bp2 messages to the Tornado GUI

show wtx-ignore-exit-status -- Show whether the exit status of the debugged task is ignored (assumed zero)

show wtx-load-flags -- Show load flags used when loading a new object module

show wtx-load-path-qualify -- Show passing of full object path to target

show wtx-load-timeout -- Show timeout in seconds used when loading new objects on target

show wtx-new-target-message -- Show whether to use the new target message format

show wtx-order-debug-print -- Show whether to print orders issued to the inferior

show wtx-override-configuration-check -- Show override checking Gdb configuration

show wtx-task-priority -- Show priority of tasks created using "run" command

show wtx-task-stack-size -- Show stack size (in bytes) of tasks created using "run" command

show wtx-tool-name -- Show tool name used by debugger when connecting to the target server

show xfer-mem-debug-print -- Show whether to enable target_xfer_memory debug printfs

19.3.10 Support facilities

List of commands:

define -- Define a new command name

document -- Document a user-defined command

dont-repeat -- Don't repeat this command
down-silently -- Same as the `down' command
echo -- Print a constant string
help -- Print list of commands
if -- Execute nested commands once IF the conditional expression is non zero
info architecture -- List supported target architectures
make -- Run the ``make'' program using the rest of the line as arguments
overlay -- Commands for debugging overlays
overlay auto -- Enable automatic overlay debugging
overlay list-overlays -- List mappings of overlay sections
overlay load-target -- Read the overlay mapping state from the target
overlay manual -- Enable overlay debugging
overlay map-overlay -- Assert that an overlay section is mapped
overlay off -- Disable overlay debugging
overlay unmap-overlay -- Assert that an overlay section is unmapped
quit -- Exit gdb
set architecture -- Set architecture of target
set check range -- Set range checking
set check type -- Set type checking
set complaints -- Set max number of complaints about incorrect symbols
set demangle-style -- Set the current C++ demangling style
set editing -- Set editing of command lines as they are typed
set height -- Set number of lines gdb thinks are in a page
set history -- Generic command for setting command history parameters
set input-radix -- Set default input radix for entering numbers
set language -- Set the current source language
set listsize -- Set number of source lines gdb will list by default
set output-radix -- Set default output radix for printing of values
set print address -- Set printing of addresses
set print array -- Set prettyprinting of arrays
set print asm-demangle -- Set demangling of C++ names in disassembly listings
set print demangle -- Set demangling of encoded C++ names when displaying symbols
set print double-format -- Set 'printf' format for double-precision floating point values
set print float-format -- Set 'printf' format for single-precision floating point values

set print object -- Set printing of object's derived type based on vtable info

set print pretty -- Set prettyprinting of structures

set print sevenbit-strings -- Set printing of 8-bit characters in strings as \nnn

set print static-members -- Set printing of C++ static members

set print union -- Set printing of unions interior to structures

set print vtbl -- Set printing of C++ virtual function tables

set prompt -- Set gdb's prompt

set radix -- Set default input and output number radices

set symbol-readnow -- Set immediate reading of full symbol table data

set verbose -- Set verbosity

set width -- Set number of characters gdb thinks are in a line

set write -- Set writing into executable and core files

set wrs-detach-behavior -- Set whether to quietly detach tasks in target/attach/quit

shell -- Execute the rest of the line as a shell command

show architecture -- Show architecture of target

show check range -- Show range checking

show check type -- Show type checking

show complaints -- Show max number of complaints about incorrect symbols

show demangle-style -- Show the current C++ demangling style

show editing -- Show editing of command lines as they are typed

show height -- Show number of lines gdb thinks are in a page

show history -- Generic command for showing command history parameters

show input-radix -- Show default input radix for entering numbers

show language -- Show the current source language

show listsize -- Show number of source lines gdb will list by default

show output-radix -- Show default output radix for printing of values

show print address -- Show printing of addresses

show print array -- Show prettyprinting of arrays

show print asm-demangle -- Show demangling of C++ names in disassembly listings

show print demangle -- Show demangling of encoded C++ names when displaying symbols

show print double-format -- Show 'printf' format for double-precision floating point values

show print float-format -- Show 'printf' format for single-precision floating point values

show print object -- Show printing of object's derived type based on vtable info

show print pretty -- Show prettyprinting of structures

show print sevenbit-strings -- Show printing of 8-bit characters in strings as \nnn

show print static-members -- Show printing of C++ static members

show print union -- Show printing of unions interior to structures

show print vtbl -- Show printing of C++ virtual function tables

show prompt -- Show gdb's prompt

show radix -- Show the default input and output number radices

show symbol-readnow -- Show immediate reading of full symbol table data

show verbose -- Show verbosity

show width -- Show number of characters gdb thinks are in a line

show write -- Show writing into executable and core files

show wrs-detach-behavior -- Show whether to quietly detach tasks in target/attach/quit

source -- Read commands from a file named FILE

up-silently -- Same as the `up' command

while -- Execute nested commands WHILE the conditional expression is non zero

19.3.11 User-defined commands

The commands in this class are those defined by the user. Use the "define" command to define a command.

20 - ASCII table

Table 45: ASCII table

Decimal	Octal	Hex	Binary	Value	Comment
000	000	000	00000000	NUL	(Null char.)
001	001	001	00000001	SOH	(Start of Header)
002	002	002	00000010	STX	(Start of Text)
003	003	003	00000011	ETX	(End of Text)
004	004	004	00000100	EOT	(End of Transmission)
005	005	005	00000101	ENQ	(Enquiry)
006	006	006	00000110	ACK	(Acknowledgment)
007	007	007	00000111	BEL	(Bell)
008	010	008	00001000	BS	(Backspace)
009	011	009	00001001	HT	(Horizontal Tab)
010	012	00A	00001010	LF	(Line Feed)
011	013	00B	00001011	VT	(Vertical Tab)
012	014	00C	00001100	FF	(Form Feed)
013	015	00D	00001101	CR	(Carriage Return)
014	016	00E	00001110	SO	(Shift Out)
015	017	00F	00001111	SI	(Shift In)
016	020	010	00010000	DLE	(Data Link Escape)
017	021	011	00010001	DC1	(XON) (Device Control 1)
018	022	012	00010010	DC2	(Device Control 2)
019	023	013	00010011	DC3	(XOFF)(Device Control 3)
020	024	014	00010100	DC4	(Device Control 4)
021	025	015	00010101	NAK	(Negative Acknowledgement)
022	026	016	00010110	SYN	(Synchronous Idle)
023	027	017	00010111	ETB	(End of Trans. Block)
024	030	018	00011000	CAN	(Cancel)

Table 45: ASCII table

Decimal	Octal	Hex	Binary	Value	Comment
025	031	019	00011001	EM	(End of Medium)
026	032	01A	00011010	SUB	(Substitute)
027	033	01B	00011011	ESC	(Escape)
028	034	01C	00011100	FS	(File Separator)
029	035	01D	00011101	GS	(Group Separator)
030	036	01E	00011110	RS	(Request to Send) (Record Separator)
031	037	01F	00011111	US	(Unit Separator)
032	040	020	00100000	SP	(Space)
033	041	021	00100001	!	
034	042	022	00100010	"	
035	043	023	00100011	#	
036	044	024	00100100	$	
037	045	025	00100101	%	
038	046	026	00100110	&	
039	047	027	00100111	'	
040	050	028	00101000	(	
041	051	029	00101001	)	
042	052	02A	00101010	*	
043	053	02B	00101011	+	
044	054	02C	00101100	,	
045	055	02D	00101101	-	
046	056	02E	00101110	.	
047	057	02F	00101111	/	
048	060	030	00110000	0	
049	061	031	00110001	1	
050	062	032	00110010	2	
051	063	033	00110011	3	
052	064	034	00110100	4	

Table 45: ASCII table

Decimal	Octal	Hex	Binary	Value	Comment
053	065	035	00110101	5	
054	066	036	00110110	6	
055	067	037	00110111	7	
056	070	038	00111000	8	
057	071	039	00111001	9	
058	072	03A	00111010	:	
059	073	03B	00111011	;	
060	074	03C	00111100	<	
061	075	03D	00111101	=	
062	076	03E	00111110	>	
063	077	03F	00111111	?	
064	100	040	01000000	@	
065	101	041	01000001	A	
066	102	042	01000010	B	
067	103	043	01000011	C	
068	104	044	01000100	D	
069	105	045	01000101	E	
070	106	046	01000110	F	
071	107	047	01000111	G	
072	110	048	01001000	H	
073	111	049	01001001	I	
074	112	04A	01001010	J	
075	113	04B	01001011	K	
076	114	04C	01001100	L	
077	115	04D	01001101	M	
078	116	04E	01001110	N	
079	117	04F	01001111	O	
080	120	050	01010000	P	

Table 45: ASCII table

Decimal	Octal	Hex	Binary	Value	Comment
081	121	051	01010001	Q	
082	122	052	01010010	R	
083	123	053	01010011	S	
084	124	054	01010100	T	
085	125	055	01010101	U	
086	126	056	01010110	V	
087	127	057	01010111	W	
088	130	058	01011000	X	
089	131	059	01011001	Y	
090	132	05A	01011010	Z	
091	133	05B	01011011	[	
092	134	05C	01011100	\	
093	135	05D	01011101	]	
094	136	05E	01011110	^	
095	137	05F	01011111	_	
096	140	060	01100000	`	
097	141	061	01100001	a	
098	142	062	01100010	b	
099	143	063	01100011	c	
100	144	064	01100100	d	
101	145	065	01100101	e	
102	146	066	01100110	f	
103	147	067	01100111	g	
104	150	068	01101000	h	
105	151	069	01101001	i	
106	152	06A	01101010	j	
107	153	06B	01101011	k	
108	154	06C	01101100	l	

Table 45: ASCII table

Decimal	Octal	Hex	Binary	Value	Comment
109	155	06D	01101101	m	
110	156	06E	01101110	n	
111	157	06F	01101111	o	
112	160	070	01110000	p	
113	161	071	01110001	q	
114	162	072	01110010	r	
115	163	073	01110011	s	
116	164	074	01110100	t	
117	165	075	01110101	u	
118	166	076	01110110	v	
119	167	077	01110111	w	
120	170	078	01111000	x	
121	171	079	01111001	y	
122	172	07A	01111010	z	
123	173	07B	01111011	{	
124	174	07C	01111100	|	
125	175	07D	01111101	}	
126	176	07E	01111110	~	
127	177	07F	01111111	DEL	

21 - About the reference manual

21.1 - Lexical rules

That is the boring part... let's make it short !

A subset of the BNF (Backus-Naur Form) will be used in the following pages :

<traditional English expression>	as it says...
[<stuff>]	stuff is optional
{<stuff>}+	stuff is present at least one or more times
{<stuff>}*	stuff is present 0 or more times

22 - Index

A

API
Model browsing 316

C

CMX
Integration 214
Semaphores 215
Timers 216
Code generation
automaton structure 203
C++ from SDL and SDL-RT 183
Command line 284
error handling 204
Memory footprint 279
Partial C code generation 182, 194
Command line 282
Code generation 284
Merge 292
PR export 285
PR/CIF import 284
Print 283
RTDS API server 291
Search utility 291
Shell 289
Simulation 290
XML-RPC wrapper generation 287
Critical sections 203

D

Debugger integration
gdb 272
MinGW 274
Multi 278
Tasking 271
XRAY 276
Documentation
SGML export 333
DSSSL 335

G

gcc
Commands 337
gdb
Commands 356
Integration 272
gdbserver 272

I

IF
Export mapping 99
Observers 107

L

ld 351
Lexical rules 386

M

Memory
footprint 279
Merge
Command line 292
utility 292
MinGW
Integration 274
Model checking 99
Multi
Integration 278

N

Nucleus
Semaphores 225
Nucleus integration 225

O

OSE Delta
Integration 227
Semaphores 228
OSE Epsilon
Integration 230
Semaphores 231

P

Posix
Integration 211
PR export
Command line 285
PR/CIF import
Command line 284
Print
Command line 283
Process
code generation
handling 201

R

RTDS API
Command line 291
RTDS_Env 204
RTDS_ENV_FREE_PARAMETER 204
RTOS integration
CMX RTX 214
Nucleus 225
OSE Delta 227
OSE Epsilon 230
Posix 211
RTOS less 189
ThreadX 233
uITRON 3.0 222
uITRON 4.0 223
VxWorks 206
Win32 213

S

Save
code generation
handling 201
SDL
C++ code generation 183
Conversion to IF 99
Export command line 285
Import command line 284
Model browsing API 316
Support 92
SDL-RT
C++ code generation 183
Search
Command line 291
Semantic check 293
Semaphore
code generation
code 201
handling 201
SGML
Export 333
Shell
Command line 289
Simulation
Command line 290
Startup synchronisation 203
Syntax check 293

T

Tasking
Integration 271
ThreadX
Integration 233
Timer
code generation
handling 202

Tornado integration 206
target connection 207
VxWorks image configuration 208
TTCN-3
Code generation 260
Data type mapping from SDL 257
Language support 246
Reference guide 235

U

uITRON 3.0
Integration 222
uITRON 4.0
Integration 223
UML
Import 298

V

VxWorks
Integration 206

W

Win32
Integration 213

X

XMI import 298
XML-RPC
Command line 287
XRAY
Integration 276