Graphical language to specify and design real-time and embedded software.

- Object: Graphical language to specify and design real-time and embedded software
- Date: December 16th 2002
- Version: 1.2
- Reference: http://www.sdl-rt.org
Introduction ................................................................. 7

Architecture ................................................................. 8
  System ........................................................................... 8
  Agents .......................................................................... 8

Communication .............................................................. 10

Behavior ................................................................. 13
  Start ............................................................................ 13
  State ............................................................................ 13
  Stop .......................................................................... 14
  Message input .......................................................... 15
    To a queue Id 16
    To a process name 17
    To the environment 18
    Via a channel or a gate 18
  Message output ....................................................... 16
    To a queue Id 16
    To a process name 17
    To the environment 18
    Via a channel or a gate 18
  Message save ........................................................... 21
  Continuous signal ....................................................... 21
  Action ........................................................................ 22
  Decision ...................................................................... 22
  Semaphore take .......................................................... 23
  Semaphore give .......................................................... 24
  Timer start .................................................................... 24
  Timer stop .................................................................... 24
  Task creation .............................................................. 25
  Procedure call ............................................................ 25
  Connectors .................................................................. 26
  Transition option ........................................................ 26
  Comment ...................................................................... 27
  Extension ..................................................................... 28
  Procedure start ........................................................... 29
  Procedure return ........................................................ 29
  Text symbol .................................................................. 29
  Additional heading symbol ......................................... 30
  Symbols ordering ......................................................... 30

Declarations ................................................................. 32
  Process ........................................................................ 32
  Procedure declaration ................................................ 33
    SDL-RT defined procedure 33
    C defined procedure 34
  Messages .................................................................... 34
  Timers ........................................................................ 35
  Semaphores .................................................................. 35

MSC ................................................................. 36
  Agent instance ........................................................... 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semaphore representation</td>
<td>37</td>
</tr>
<tr>
<td>Semaphore manipulations</td>
<td>37</td>
</tr>
<tr>
<td>Message exchange</td>
<td>39</td>
</tr>
<tr>
<td>Synchronous calls</td>
<td>41</td>
</tr>
<tr>
<td>State</td>
<td>42</td>
</tr>
<tr>
<td>Timers</td>
<td>44</td>
</tr>
<tr>
<td>Time interval</td>
<td>46</td>
</tr>
<tr>
<td>Coregion</td>
<td>48</td>
</tr>
<tr>
<td>MSC reference</td>
<td>49</td>
</tr>
<tr>
<td>Text symbol</td>
<td>51</td>
</tr>
<tr>
<td>Comment</td>
<td>51</td>
</tr>
<tr>
<td>Action</td>
<td>51</td>
</tr>
<tr>
<td>High-level MSC (HMSC)</td>
<td>52</td>
</tr>
<tr>
<td>Data types</td>
<td>54</td>
</tr>
<tr>
<td>Type definitions and headers</td>
<td>54</td>
</tr>
<tr>
<td>Variables</td>
<td>54</td>
</tr>
<tr>
<td>C functions</td>
<td>54</td>
</tr>
<tr>
<td>External functions</td>
<td>54</td>
</tr>
<tr>
<td>Memory management</td>
<td>55</td>
</tr>
<tr>
<td>Global variables</td>
<td>55</td>
</tr>
<tr>
<td>Message parameters</td>
<td>55</td>
</tr>
<tr>
<td>Syntax</td>
<td>56</td>
</tr>
<tr>
<td>Semantic</td>
<td>57</td>
</tr>
<tr>
<td>Object orientation</td>
<td>58</td>
</tr>
<tr>
<td>Package</td>
<td>58</td>
</tr>
<tr>
<td>Block class</td>
<td>58</td>
</tr>
<tr>
<td>Process class</td>
<td>60</td>
</tr>
<tr>
<td>Symbols contained in diagrams</td>
<td>68</td>
</tr>
<tr>
<td>Textual representation</td>
<td>69</td>
</tr>
<tr>
<td>Example systems</td>
<td>73</td>
</tr>
<tr>
<td>Ping Pong</td>
<td>73</td>
</tr>
<tr>
<td>A global variable manipulation</td>
<td>77</td>
</tr>
<tr>
<td>Keywords</td>
<td>81</td>
</tr>
<tr>
<td>Differences with classical SDL</td>
<td>82</td>
</tr>
<tr>
<td>Data types</td>
<td>82</td>
</tr>
<tr>
<td>Semaphores</td>
<td>82</td>
</tr>
<tr>
<td>Inputs</td>
<td>82</td>
</tr>
<tr>
<td>Names</td>
<td>82</td>
</tr>
<tr>
<td>Object orientation</td>
<td>82</td>
</tr>
<tr>
<td>Modifications from previous releases</td>
<td>84</td>
</tr>
<tr>
<td>Semaphore manipulation</td>
<td>84</td>
</tr>
</tbody>
</table>
1 - Introduction

As its name states, SDL-RT is based on SDL standard from ITU extended with real time concepts.

SDL has been developed in the first place to specify telecommunication protocols but experience showed some of its basic principles could be used in a wide variety of real time and embedded systems. Its main benefits are:

- architecture definition,
- graphical finite state machine,
- object orientation.

But SDL was not meant to design real time systems and some major drawbacks prevented it to be widely used in the industry:

- obsolete data types,
- old fashioned syntax,
- no pointer concept,
- no semaphore concept.

SDL being a graphical language it is obviously not suited for any type of coding. Some parts of the application still need to be written in C or assembly language. Furthermore legacy code or off the shelf libraries such as RTOS, protocol stacks, drivers have C APIs. Last but not least there is no SDL compilers so SDL need to be translated into C code to get down to target. So all SDL benefits are lost when it comes to real coding and integration with real hardware and software.

Considering the above considerations a real time extension to SDL needed to be defined that would keep the benefits of SDL and solve its weaknesses. The simpler the better! SDL-RT was born based on 2 basic principles:

- Replace SDL data types by C,
- Add semaphore support in the behavior diagrams.

The result, SDL-RT, is a:

- simpler,
- object oriented,
- graphical language,
- supporting all basic real time concepts,
- based on standard languages.
2 - Architecture

2.1 - System

The overall design is called the system and everything that is outside the system is called the environment. There is no specific graphical representation for the system but the block representation can be used if needed.

2.2 - Agents

The system can be decomposed in functional blocks. A block does not imply any physical implementation on the target, it is a structuring element. A block can be further decomposed in blocks and so on allowing to handle large systems. A block symbol is a solid rectangle with its name in it:

```
MyBlock
```

A simple block example.

When the SDL-RT system is decomposed down to the simplest block, the way the block fulfils its functionality is described with processes. A lowest level block can be composed of one or several processes. To avoid having blocks with only one process it is allowed to mix together blocks and processes at the same level e.g. in the same block.

A process symbol is a rectangle with cut corners with its name in it:

```
MyProcess
```

A simple process example.

A process is basically the code that will be executed. It is a finite state machine based task (Cf. “Behavior” on page 13) and has an implicit message queue to receive messages. It is possible to have several instances of the same process running independently. The number of instances present when the system starts and the maximum number of instances are declared between parenthesis after the name of the process. The full syntax in the process symbol is:

```
<process name>[<number of instances at startup>, <maximum number of instances>]
```

If omitted default values are 1 for the number of instances at startup and infinite for the maximum number of instances.
An example process that has no instance at startup and a maximum of 10 instances.

The overall architecture can be seen as a tree where the leaves are the processes.

A view of the architecture tree

When viewing a block, depending on the size of the system, it is more comfortable to only represent the current block level without the lower agents.
3 - Communication

SDL-RT is event driven, meaning communication is based on message exchanges. A message has a name and a parameter that is basically a pointer to some data. Messages go through channels that connect agents and end up in the processes implicit queues. Channels have names and are represented by a one-way or two-ways arrows. A channel name is written next to the arrow without any specific delimiter. The list of messages going in a specific way are listed next to the arrow between brackets and separated by commas. Messages can be gathered in message lists, to indicate a message list in the list of messages going through a channel the message list is surrounded by parenthesis. Note the same message can be listed in both directions.

```
channelName
[message1, (messageList1), message2]

channelName
[message4, message5, message2] [message1, message2, (messageList1)]
```

Channels end points can be connected to: the environment, another channel or a process. Graphically a channel can be connected to a block but it is actually connected to another channel inside the block. To represent the outside channels connected to the block at the upper architecture level, a block view is surrounded by a frame representing the edge of the block. The upper level channels connected to the block are then represented outside the frame and channels inside the block can be connected to these upper level channels. Note a channel can be connected to several channels. In any case consistency is kept between levels e.g. all messages in a channel are listed in the upper or lower level channels connected to it.
Example:
Let us consider an SDL-RT system made of two blocks: blockA and blockB.

![An example system view](image)

The channels chEnvA and chEnvB are connected to the surrounding frame of the system mySystem. They define communication with the environment, e.g. the interface of the system. chEnvA and chAB are connected to blockA and define the messages coming in or going out of the block.

![An inner block view](image)

The inner view of block blockA shows it is made of the blocks blockC and blockD and of the process processE. chEnvAC is connected to the upper level channel chEnvA and chABD is connected
to the upper channel \textit{chAB}. The flow of messages is consistent between levels since for example the messages coming in block \textit{blockA} through \textit{chEnvA} (\textit{message1, message2, message3}) are also listed in \textit{chEnvAC}. 
4 - Behavior

First of all a process has an implicit message queue to receive the messages listed in the channels. A process description is based on an extended finite state machine. A process state determines which behavior the process will have when receiving a specific stimulation. A transition is the code between two states. The process can be hanging on its message queue or a semaphore or running e.g. executing code.
SDL-RT processes run concurrently; depending on the underlying RTOS and sometimes on the target hardware the behavior might be slightly different. But messages and semaphores are there to handle process synchronization so the final behavior should be independent of the RTOS and of the hardware. Since SDL-RT is open to any C code it is up to the designer to make sure this statement stays true!
Note that in a state diagram the previous statement is always connected to the symbol upper frame and the next statement is connected to the lower frame or on the side.

4.1 - Start

The start symbol represent the starting point for the execution of the process:

\[
\text{Start symbol}
\]

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 messages. All other symbols are allowed.

4.2 - State

The name of the process state is written in the state symbol:

\[
\text{<state name>}
\]

\[
\text{State symbol}
\]

The state symbol means the process is waiting for some input to go on, the allowed symbols to follow a state symbol are:

- message input
  the message could be coming from an external channel, or it could be a timer message 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 signal expressions are evaluated before the message input!

- save
  the incoming message can not be treated in the current process state. It is saved until the process state changes. When the process state has changed the saved messages are treated first (before any other messages in the queue but after continuous signals).

Some transitions can be valid for several states, the different state names are then listed separated by a comma. A star (’*’) means all states.

**Examples:**

```
idle
```

In state `idle` message `msg1` can be received and `msg2` is saved.

```
*               idle, maintenance
```

Message `msg1` can be received in any state. In states `idle` and `maintenance` the expression `a>0` is first evaluated.

A process in a specific state can receive several types of messages or treat several continuous signals. To represent such a situation it is possible to have several message inputs connected to the state or to split the state in several symbols with the same name.

**Examples:**

```
idle              idle
```

Two ways of writing in state `idle`, `sig1` or `sig2` can be received.

**4.3 - Stop**

A process can terminate itself with the stop symbol.
Stop symbol

Note a process can not kill another process, it can only kill itself. There is no syntax for that symbol.

4.4 - Message input

The message input symbol represent the type of message that is expected in an SDL-RT state. It always follows an SDL-RT state symbol and if received the symbols following the input are executed.

An input has a name and can come with parameters. To receive the parameters it is necessary to declare at least one variable that will be assigned to point on the parameters. If the parameter length is unknown, because the parameters are unstructured data, it is also possible to get the parameter length assigned to a pre-declared variable.

The syntax in the message input symbol is the following:

`<Message name> [([<data length>,] <pointer on data>)]`

`<data length>` is a variable that needs to be declared.
`<pointer on data>` is a variable that needs to be declared as an int.

Examples:

```
MESSAGE
    ConReq(unsigned char *),
    ConConf,
    DisReq(myStruct *);

long myDataLength;
unsigned char *myData;
myStruct *pData;
```

```
ConReq (myDataLen, myData)   ConConf    DisReq (pData)
```
4.5 - Message output

A message output is used to exchange information. It puts data in the receiver’s message queue in an asynchronous way.

When a message has parameters, a pointer to the parameters is given. If the parameter is structured, its length does not need to be specified since it is basically a sizeof of the parameter type. Otherwise its length is given as a first parameter of the output symbol.

The syntax in the message output symbol can be written in several ways depending if the queue Id or the name of the receiver is known or not. A message can be sent to a queue Id or to a process name or via a channel or a gate. When communicating with the environment, a special syntax is provided.

4.5.1 To a queue Id

The symbol syntax is:

<message name>[[<data length>,] <pointer on data>] TO_ID <receiver queue id>

It can take the value given by the SDL-RT keywords:

- PARENT: The queue id of the parent process.
- SELF: The queue id of the current process.
- OFFSPRING: The queue id of the last created process if any or NULL if none.
- SENDER: The queue id of the sender of the last received message.
Examples:

```c
long myDataLength;
unsigned char *myData;
myStruct *pData;

MESSAGE
ConReq(unsigned char *),
ConConf,
DisReq(myStruct *);
```

4.5.2 To a process name

The syntax is:

```
<message name>[(<data length>,) <pointer on data>] TO_NAME <receiver name>
```

**Message output to a process name**

If several instances have the same process name (several instances of the same process for example), the 'TO_NAME' will send the message to the first created process with the corresponding name. Therefore this method should not be used when the process name is not unique within the system.
4.5.3 To the environment

**Message output to environment**

The symbol syntax is:

```
<message name>[([<data length>],) <pointer on data>] TO_ENV <C macro name>
```

*C macro name* is the name of the macro that will be called when this SDL output symbol is hit. If no macro is declared the message will be sent to the environment.

Example:

```
ConReq (0xFF, myData) TO_ENV
```

In this second example the generated code will be:

```
MESSAGE_TO_HDLC(ConReq,myDataLength,myData)
```

**Note:**

When sending data pointed by <pointer on data>, the corresponding memory should be allocated by the sender and should be freed by the receiving process. This is because this memory area is not copied to the receiver; only the pointer value is transmitted. So after being sent the sender should not use it any more.

4.5.4 Via a channel or a gate

A message can be sent via a channel in the case of a process or via a gate in the case of a process class.

**Message output via a channel or a gate**

The symbol syntax is:

```
<message name>[([<data length>],) <pointer on data>] VIA <channel or gate name>
```

*<channel or gate name>* is the name of the channel or gate the message will go through. This concept is especially useful when using object orientation since classes are not supposed to know their environment; so messages are sent via the gates that will be connected to the surrounding environment when instantiated.
Examples:

With the architecture defined above, both outputs are equivalent.
aProcess sends msg2 to myProcess without knowing its name nor its PID
4.6 - Message save

A process may have intermediate states that can not deal with new request until the on-going job is done. These new requests should not be lost but kept until the process reaches a stable state. Save concept has been made for that matter, it basically holds the message until it can be treated.

![Save symbol]

The Save symbol is followed by no symbol. When the process changes to a new state the saved messages will be the first to be treated (after continuous signals if any).

The symbol syntax is:

\[ \text{<message name>} \]

Even if the message has parameters.

Example:

Let’s consider the above process in state `inter` to receive the following messages:

- `msg3` will be saved,
- `msg2` will make the process go to state `stable`.
- Since `msg3` has been saved it will first be treated and finally `msg1`.

4.7 - Continuous signal

A continuous signal is an expression that is evaluated right after a process reaches a new state. It is evaluated before any message input or saved messages.

![Continuous signal symbol]

The continuous signal expression to evaluate can contain any standard C expression that returns a C true/false expression. Since an SDL state can contain several continuous signal a priority level
needs to be defined with the \texttt{PRIO} keyword. Lower values correspond to higher priorities. A continuous signal symbol can be followed by any other symbol except another continuous signal or a message input. The syntax is:
\[ <C \text{ condition expression}> \]
\[ \text{PRIO } <\text{priority level}> \]

Example:

In the above example, when the process gets in state idle it will first evaluate expression \( (b<10) \text{ || } (c!=0) \). If the expression is not true or if the process stayed in the same state it will evaluate expression \( a>5 \). If the expression is not true or if the process stayed in the same state it will execute \texttt{msg1} transition.

4.8 - Action
An action symbol contains a set of instructions in C code. The syntax is the one of C language.

Example:

```c
/* Say hi to your friend */
printf("Hello world !\n");
for (i=0;i<MAX;i++)
{
    newString[i] = oldString[i];
}
```

4.9 - Decision
A decision symbol can be seen as a C switch / case.

\textit{Decision symbols}
Since it is graphical and therefore uses quite some space on the diagram it is recommended to use it when its result modifies the resulting process state. The decision symbol is a diamond with branches. Since a diamond is one of the worst shape to put text in it, it can be a "diamonded" rectangle. Each branch can be seen as a case of the switch.

The expression to evaluate in the symbol can contain:
- any standard C expression that returns a C true/false expression,
- an expression that will be evaluated against the values in the decision branches.

The values of the branches have keyword expressions such as:
- $\text{>, <, >=, <=, !=, ==}$
- true, false, else

The else branch contains the default branch if no other branch made it.

Examples:

4.10 - Semaphore take

The Semaphore take symbol is used when the process attempts to take a semaphore.

![Semaphore take symbol]

Semaphore take symbol

To take a semaphore, the syntax in the ‘semaphore take SDL-RT graphical symbol’ is:

$\text{[<status> = ] <semaphore name>(<timeout option>)}$

where $\text{<timeout option>}$ is:
- FOREVER
  Hangs on the semaphore forever if not available.
- NO_WAIT
  Does not hang on the semaphore at all if not available.
• <number of ticks to wait for>
  Hangs on the semaphore the specified number of ticks if not available.
and <status> is:
  • OK
    If the semaphore has been successfully taken
  • ERROR
    If the semaphore was not found or if the take attempt timed out.

4.11 - Semaphore give

Semaphore give symbol

To give a semaphore, the syntax in the ‘semaphore give SDL-RT graphical symbol’ is:
<semaphore name>

4.12 - Timer start

Timer start symbol

To start a timer the syntax in the ‘start timer SDL-RT graphical symbol’ is:
<timer name>(<time value in tick counts>)
<time value in tick counts> is usually an ‘int’ but is RTOS and target dependant.

4.13 - Timer stop

Timer stop symbol

To cancel a timer the syntax in the ‘cancel timer SDL-RT graphical symbol’ is:
<timer name>
4.14 - Task creation

Task creation symbol
To create a process the syntax in the create process symbol is:
<process name>[:<process class>] [PRIO <priority>]
to create one instance of <process class> named <process name> with priority <priority>.

Examples:

```
myProcess
anotherProcess: aClassOfProcess
myProcess PRIO 80
```

4.15 - Procedure call

Procedure call symbol
The procedure call symbol is used to call an SDL-RT procedure (Cf. “Procedure declaration” on page 33). Since it is possible to call any C function in an SDL-RT action symbol it is important to note SDL-RT procedures are different because they know the calling process context, e.g. SDL-RT keywords such as SENDER, OFFSPRING, PARENT are the ones of the calling process.
The syntax in the procedure call SDL graphical symbol is the standard C syntax:
[<return variable> =] <procedure name>(({<parameters>})*);

Examples:

```
myResult = myProcedure (myParameter);
anotherProcedure();
```
4.16 - Connectors

Connectors are used to:
- split a transition into several pieces so that the diagram stays legible and printable,
- to gather different branches to a same point.

A connector-out symbol has a name that relates to a connector-in. The flow of execution goes from the connector out to the connector in symbol.

A connector contains a name that has to be unique in the process. The syntax is:

\[ <\text{connector name}> \]

Examples:

\[
\text{printf("Hello ");}
\]

\[
\text{printf("world !\n");}
\]

4.17 - Transition option

Transition options are similar to C \#ifdef.

\[
\text{Transition option symbol}
\]

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:
#ifdef <expression>
The branches can stay separated to the end of the transition or they can meet again and close the option as would do an #endif.

Examples:

4.18 - Comment
The comment symbol allows to write any type of informal text and connect it to the desired symbol. If needed the comment symbol can be left unconnected.
4.19 - Extension

The extension symbol is used to complete an expression in a symbol. The expression in the extension symbol is considered part of the expression in the connected symbol. Therefore the syntax is the one of the connected symbol.

Example:

```
idle

msg1

msg1 indicates the system is ready.
```

Comment symbol
Example:

4.20 - Procedure start
This symbol is specific to a procedure diagram. It indicates the procedure entry point.

\[\text{Procedure start symbol}\]

There is no syntax associated with this symbol.

4.21 - Procedure return
This symbol is specific to a procedure diagram. It indicates the end of the procedure.

\[\text{Procedure return symbol}\]

This symbol is specific to a procedure diagram. It indicates the end of the procedure. If the procedure has a return value it should be placed by the symbol.

4.22 - Text symbol
This symbol is used to declare C types variables.
The syntax is C language syntax.

4.23 - Additional heading symbol

This symbol is used to declare SDL-RT specific headings.

```
<SDL-RT contextual declaration >
```

Additional heading symbol

It has a specific syntax depending in which diagram it is used.

4.24 - Symbols ordering

The following table shows which symbols can be connected to a specific symbol.

| The symbol in this column can be followed by the ticked symbols in its row. | start | state | stop | input | output | save | continuous signal | action | decision | semaphore take | semaphore give | timer start | timer stop | task creation | procedure call | connector in | connector out | transition option | procedure start | procedure return |
| start | - | X | X | - | X | - | - | X | X | X | X | X | - | X | X | X | X | X | X | - | - |
| state | - | - | - | X | - | X | X | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| stop | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| input | - | X | X | - | X | - | - | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| output | - | X | X | - | X | - | - | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| save | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - | - |
| continuous | - | X | X | - | X | - | - | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| action | - | X | X | - | X | - | - | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| semaphore take | - | X | X | - | X | - | - | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| semaphore give | - | X | X | - | X | - | - | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| timer start | - | X | X | - | X | - | - | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
| timer stop | - | X | X | - | X | - | - | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X | X |
The table above should be read row by row. The symbol in the left column can be followed by the ticked symbols on its row. For example the stop symbol cannot be followed by any other symbol. The start symbol can be followed by input, save, or continuous signal symbols.

<table>
<thead>
<tr>
<th>Task Creation</th>
<th>Start</th>
<th>State</th>
<th>Stop</th>
<th>Input</th>
<th>Output</th>
<th>Save</th>
<th>Continuous Signal</th>
<th>Action</th>
<th>Decision</th>
<th>Semaphore Take</th>
<th>Semaphore Give</th>
<th>Timer Start</th>
<th>Timer Stop</th>
<th>Task Creation</th>
<th>Procedure Call</th>
<th>Connector In</th>
<th>Connector Out</th>
<th>Transition Option</th>
<th>Procedure Start</th>
<th>Procedure Return</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task Creation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Procedure Call</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Connector Out</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Connector In</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Transition Option</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Procedure Start</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Procedure Return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
5 - Declarations

5.1 - Process

A process is implicitly declared in the architecture of the system (Cf. “Architecture” on page 8) since the communication channels need to be connected.

Process symbol

A process has an initial number of instances at startup and a maximum number of instances. A process can also be an instance of a process class (Cf. “Object orientation” on page 58), in that case the name of the class follows the name of the instance after a colon.

The general syntax is:

\(<process\ \text{instance\ name}>[:<process\ \text{class}>][(<\text{initial\ number\ of\ instances}>, \ <\text{maximum\ number\ of\ instances}>)]\)

When a process is an instance of a process class the gates of the process class need to be connected in the architecture diagram. The names of the gates appear in the process symbol with a black circle representing the connection point.

Process class instance

The messages defined in the package going through the gates must be consistent with the messages listed in the architecture diagram where the process instance is defined.
Example:

![Procedure declaration symbol]

### 5.2 - Procedure declaration

An SDL-RT procedure can be defined in any diagram: system, block, or process. It is usually not connected to the architecture but since it can output messages a channel can be connected to it for informational purpose.

The declaration syntax is the same as a C function. A procedure definition can be done graphically with SDL-RT or textually in a standard C file.

### 5.2.1 SDL-RT defined procedure

If defined with SDL-RT the calling process context is implicitly given to the procedure. So if a message output is done, the message will be output from the process calling the procedure. That is why the message should be defined in one of the channels connected to the process instead of a channel connected to a procedure. To call such a procedure the procedure call symbol should be used.
5.2.2 C defined procedure
If defined in C language the process context is not present. To call such a procedure a standard C statement should be used in a action symbol.

Example:

![Diagram](image)

5.3 - Messages
Messages are declared at any architecture level in the additional heading symbol. A message declaration may include the parameter type in C. The syntax is:

```plaintext
MESSAGE <message name> [(<parameter type>)] {,<message name> [(<parameter type>)]};
```

It is also possible to declare message lists to make the architecture view more synthetic. Such a declaration can be made at any architecture level in the additional heading symbol. The syntax is:

```plaintext
MESSAGE_LIST <message list name> = <message name> <message name> (*);
```
A message list can contain a message list, the included message list name is surrounded by parenthesis.
Example:

```
5.4 - Timers

There is no need to declare timers. They are self declared when used in a diagram.

5.5 - Semaphores

Semaphores can be declared at any architecture level. Since each RTOS has its own type of semaphores with specific options there will be no detailed description of the syntax. The general syntax in the declaration symbol is:

```
<semaphore type>
<semaphore name>({<list of options>}[,,])*;
```

**Semaphore declaration**

It is important to note the semaphore is identified by its name.
6 - MSC

SDL-RT integrates the Message Sequence Chart dynamic view. On such a diagram, time flows from top to bottom. Lifelines represent SDL-RT agents or semaphores and key SDL-RT events are represented. The diagram put up front the sequence in which the events occur. In the case of embedded C++ it is possible to use a lifeline to represent an object. In that case the type is object and the name should be <object name>:<class name>

6.1 - Agent instance

An agent instance starts with an agent instance head followed by an instance axis and ends with an instance tail or an instance stop as shown in the diagrams below.

The type of the agent can be specified on top of the head symbol and the name of the agent is written in the instance head symbol. The instance tail symbol means the agent lives after the diagram. The instance stop symbol means the agent no longer exist after the symbol.

When an agent creates another agent a dashed arrow goes from the parent agent to the child agent.
Example:

6.2 - Semaphore representation
A semaphore representation is made of a semaphore head, a lifeline, and a semaphore end or tail. The symbols are the same as for a process except for the head of the semaphore.

6.3 - Semaphore manipulations
Several cases are to be considered with semaphore manipulations. A process makes an attempt to take a semaphore, its attempt can be successful or unsuccessful, if successful the semaphore might still be available (counting semaphore) or become unavailable. During the time the semaphore is unavailable, its lifeline gets thicker until it is released.
The manipulation symbols are the following:

- **Semaphore creation from a known process.**
  - take
  - succeeded
  - timed out
  - give

- **Semaphore creation from an unknown process.**
  - take
  - succeeded
  - give
Example:

Process `myProc1` first creates semaphore `mySem`, then takes it successfully. Process `myProc2` makes an attempt to take semaphore `mySem` but gets blocked on it. Process `myProc1` releases the semaphore so `myProc2` successfully gets the semaphore. Process `myProc2` gives it back, and kills it.

### 6.4 - Message exchange

A message symbol is a simple arrow with its name and optional parameters next to it. The arrow can be horizontal meaning the message arrived instantly to the receiver or the arrow can go down to show the message arrived after a certain time or after another event. A message can not go up! When the sender and the receiver are represented on the diagram the arrow is connected to their instances. If the sender is missing it is replaced by a white circle, if the receiver is missing it is replaced by a black circle. The name of the sender or the receiver can optionally be written next to the circle.
An external agent called keypad sends run message to process sender. Process sender sends initMsg that is considered to be received immediately to block receiver. Block receiver replies readyMsg, process sender sends startMsg, and block receiver sends run to an external agent.

A message is considered received by an agent when it is read from the agent’s message queue; not when it arrives in the message queue!
6.5 - Synchronous calls

This representation is used when using embedded C++ to show method calls on an object. Object can be represented by lifelines. Synchronous calls are shown with an arrow to the instance representing the object. While the object has the focus its lifeline becomes a black rectangle and the agent lifeline becomes a white rectangle. That means the execution flow has been transferred to the object. When the method returns a dashed arrow return to the method caller.
6.6 - State

A lifeline represents a process and depending on its internal state a process reacts differently to the same message. It is interesting to represent a process state on its lifeline. It is also interesting to represent a global state for information. In that case the state symbol covers the concerned instances. In both cases the same symbol is used.

Process keyboard calls method set_URL from myPhoneBook object that is an instance of PhoneBook class.
Example:

Process server goes to idle state. Process caller in its start transition sends a conReq to server and goes to state idle. Process server returns an conConf message and goes to connected state. When conConf message is received by process caller it also moves to connected state.
6.7 - Timers

Two symbols are available for each timer action depending if the beginning and the end of the timer are connected or not. The timer name is by the cross and timeout value is optional. When specified the timeout value unit is not specified; it is usually RTOS tick counts.

- Timer start connected
- Timer stop unconnected
- Timeout unconnected
- Timer start unconnected
- Timer stop connected
- Timeout connected
- Timer restart connected
Examples:

Process **caller** tries to initiate connection with **conReq** message. At the same time it starts timer **tConReq** so that if no answer is received it will retry connecting. If an answer is received the timer is cancelled and process **caller** goes to state **connected**.
6.8 - Time interval

To specify a time interval between two events the following symbol is used.

Process caller tries to initiate connection with `conReq` message. Since it receives no answer after two tries it gives up and goes to unconnected state.

Time constraint syntax is the following:
- absolute time is expressed with an @ up front the time value,
• relative time is expressed with nothing up front its value,
• time interval is expressed between square brackets,
• time unit is RTOS specific -usually tick counts- unless specified (s, ms, µs).

Note it is possible to use time constraint on a single MSC reference.

Absolute time can also be specified with the following symbol:

\[ <\text{absolute time value}> \]

**Examples:**

**Table 1: Examples of time constraint expressions**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.3ms</td>
<td>takes 1.3 ms to do</td>
</tr>
<tr>
<td>[1,3]</td>
<td>takes a minimum of 1 to a maximum of 3 time units</td>
</tr>
<tr>
<td>@[12.4s,14.7s]</td>
<td>should not occur before absolute time 12.4 s and should not finish after absolute time 14.7 s.</td>
</tr>
<tr>
<td>&lt;5</td>
<td>takes less than 5 time units</td>
</tr>
</tbody>
</table>

Process server reaches state idle at absolute time 34 Sec.
Process client request process server to compute some work in less than 0x02FF time units.
6.9 - Coregion

Coregion is used whenever the sequence of events does not matter. Events in a coregion can happen in any order. The coregion symbol replaces the lifeline instance.

Example:

Process controller

sends `stopEngine` and `displayInfo` or
sends `displayInfo` and `stopEngine`.

Connecting MSC should take less than 200mS.
6.10 - MSC reference

MSC reference allows to refer to another MSC. The resulting MSC is smaller and more legible.

A reference concerns the connected instances. An instance is connected if its lifeline disappears in the symbol. An instance is not connected if it goes over the reference symbol.
The **DataTransfer** MSC starts with a reference to **Connecting** MSC. That means the scenario described in **Connecting** MSC is to be done before the rest of the **DataTransfer** MSC occur.
6.11 - Text symbol

The text symbol contains data or variable declarations if needed in the MSC.

\[
\text{Text symbol}
\]

6.12 - Comment

As its name states...

\[
\text{Comment symbol}
\]

6.13 - Action

An action symbol contains a set of instructions in C code. The syntax is the one of C language.

Examples:

\[
\text{/* Say hi to your friend */}
\text{printf("Hello world !\n");}
\text{for \(i=0; i<\text{MAX}; i++\)}
\text{\{}
\text{\hspace{1em}newString[i] = oldString[i];}
\text{\}}
\]
6.14 - High-level MSC (HMSC)

High level MSC diagram is a synthetic view of how MSCs relate to each other. It is only a few symbols: start, stop, alternative, parallel, state or condition, and MSC reference.

The action symbol contains standard C instructions related to data declarations.

unsigned char *fullData, *data;
int length;

memcpy(fullData, data, length); fullData pointer points at the end of buffer.
The SDL-RT HMSC starts with the start symbol and ends with the stop symbol. The parallel symbol means the following connected path will be executed in parallel. The Alternative symbol means one and only one of the connected path is executed. Whenever two paths meet again the path separator symbol is to be repeated. That means if a parallel symbol is used that creates two different paths, the parallel symbol should be used when the path merge back.

Symbols are connected with lines or arrows if clearer. A symbol is entered by its upper level edge and leaved by any other edge.

Example:

The system starts in disconnected state. Connection attempts are made, either the conFailed scenario or the conSucceeded scenario is executed. If conSucceeded is executed supervising and dataTransfer are executing in parallel. They merge back to disconnect and end the HMSC scenario.
7 - Data types

The data types, the syntax and semantic are the ones of ANSI C language. There is no SDL-RT predefined data types at all but just some keywords that should not be used in the C code. Considering the SDL-RT architecture and concepts surrounding the C code some important aspects need to be described.

7.1 - Type definitions and headers

Types are declared in the text symbol:

```
<Any C type declaration >
```

Types declared in an agent are only visible in the architecture below the agent.

7.2 - Variables

Variables are declared after the type definitions in the same text symbol.

```
<Any C type definition >
<Any C global variable definition >
```

Variables declared in an agent are only visible in the architecture below the agent. For example global variables are to be declared at system level. A variable declared in a block level is not seen by an upper level block. Variables declared in an SDL-RT process in a text symbol are local to the process. They can not be seen or manipulated by any other process.

7.3 - C functions

SDL-RT internal C functions are to be defined through the SDL-RT procedure symbol. An SDL-RT procedure can be defined graphically in SDL-RT or textually in C. When defined in C the procedure call symbol should not be used. A standard C statement in an action symbol should be used.

7.4 - External functions

External C functions can be called from the SDL-RT system. These should be prototyped in the system or in an external C header. It is up to an SDL-RT tool to gather the right files when compiling and linking.
8 - Memory management

Real time systems need to exchange information. The best way to do so is to have a reserved chunk of shared memory that several tasks can access. SDL-RT implicitly runs on such an underlying architecture since it supports global variables and exchanges message parameters through pointers. That raises memory management rules to follow to ensure a proper design.

8.1 - Global variables

SDL-RT processes can share global variables. This is very powerful but also very dangerous since the data can be corrupted if manipulated without caution. It is strongly recommended to use semaphores to access global variables to be sure data is consistent. An example of such a design is given later in this document.

8.2 - Message parameters

Parameters of a message are passed through a pointer. This implies the data pointed by the sending process will be accessible by the receiving process. Therefore a good design should meet the following rules:

- Sending processes allocate specific memory areas to store parameters,
- Once the message is sent the parameter memory area should never be manipulated again by the sending process,
- Receiver processes are responsible for freeing memory containing message parameters.
9 - Syntax

All SDL-RT names must be a combination of alphabetical characters, numerical characters, and underscores. No other symbols are allowed.

Examples:

myProcessName
my_procedure_name
block_1
_semaphoreName
10 - Semantic

Since some SDL-RT concepts can be reached through their names (processes, semaphores) each name in the system must be unique. This will make the design more legible and ease the support of SDL-RT in a tool.

It is suggested to use the following convention for names:

- block names should start with 'b',
- process names should start with 'p',
- timer names should start with 't',
- semaphore names should start with 's',
- global variables should start with 'g'.

11 - Object orientation

11.1 - Package

Object orientation in SDL-RT allows to define classes of processes and blocks. Classes definitions are gathered in an SDL-RT package. To be able to use classes defined in a package, an SDL-RT system should explicitly import the package with USE keyword in an additional heading symbol at system level.

```
USE <package name>
```

A package is a separated entity that contains agents or classes of agents. It is referenced by its name.

```
<package name>
```

It can contain:
- blocks,
- classes of blocks,
- processes,
- classes of processes,
- procedures,
- data definitions.

11.2 - Block class

Defining a block class allows to use the same block several times in the SDL-RT system. The SDL-RT block does not support any other object oriented features. A block class symbol is a block symbol with a double frame. It has no channels connected to it.

```
[block class name]
```

A block class can be instantiated in a block or system. The syntax in the block symbol is:

```
<block instance name>:<block class name>
```
Messages come in and go out of a block class through gates. In the block class diagram gates are represented out of the block class frame. When a block class is instantiated the gates are connected to the surrounding SDL-RT architecture. The messages listed in the gates are to be consistent with the messages listed in the connected channels.

```
<block instance name>: 
<block class name>

<gate name>
```
Example:

Definition diagram of myBlockClass block class

blockA is an instance of myBlockClass

11.3 - Process class

Defining a process class allows to:

• have several instances of the same process in different places of the SDL-RT architecture,
• inherit from a process super-class,
• specialize transitions and states.
A process class symbol is a process symbol with a double frame. It is has no channels connected to it.

A process class can be instantiated in a block or a system. The syntax in the process symbol is:

```
<process instance name>:<process class name>
```

Messages come in and go out of a process class through gates. In the process class diagram, gates are represented out of the process class frame. When a process class is instantiated the gates are connected to the surrounding SDL-RT architecture. The messages listed in the gates are to be consistent with the messages listed in the connected channels. The names of the gates appear in the process symbol with a black circle representing the connection point.

Since a class is not supposed to know the surrounding architecture, message outputs should not use the TO_NAME concept. Instead TO_ID, VIA, or TO_ENV should be used.
Example:

SDL-RT transitions, gates and data are the elements that can be redefined when specializing. In the sub class the super class to inherit from is defined with the `INHERITS` keyword in an additional heading symbol. There are several ways to specialize a process class depending on what is defined in the super class.
• If the element is new in the sub class, it is simply added to the super class definition.

```
INHERITS MySuperClass;
```

```
MySuperClass

stable

msg3

idle

MyClass

stable

msg1

unstable

An instance of MyClass
```
• If the element exists in the super class, the new element definition overwrites the one of the super class,

```
int myVar;

stable

msg3

myVar = 2;

idle

INHERITS MySuperClass;

stable

msg3

myVar = 3;

unstable
```

```
INHERITS MySuperClass;

INHERITS MySuperClass;
```

```
myGate1 [msg3]

myGate1 [msg3]
```

MySuperClass

MyClass

An instance of MyClass

• A class can be defined as abstract with the `ABSTRACT` keyword. It means the class cannot be instantiated as is; it needs to be specialized. A class can define abstract transitions or
abstract gates. It means the abstract transition or gate exists but that it is not defined. Such a class is obviously abstract and needs to be defined as such.

MyAbstractSuperClass

MyClass

An instance of MyClass
Here comes an example mixing all object oriented concepts and the resulting object:

```plaintext
int myVar;

MyAbstractSuperClass

INHERITS MyAbstractSuperClass;

char myOtherVar;

MyClass
```
```c
int myVar;
char myOtherVar;

An instance of MyClass
```
## 12 - Symbols contained in diagrams

The table below shows what symbols can be contained in a specific diagram type.

<table>
<thead>
<tr>
<th>In the diagrams listed in this column the ticked symbols on the right can be used.</th>
<th>package</th>
<th>block class</th>
<th>process class</th>
<th>block</th>
<th>process</th>
<th>procedure declaration</th>
<th>semaphore declaration</th>
<th>channel</th>
<th>additional heading</th>
<th>text</th>
<th>gate definition</th>
<th>gate usage</th>
<th>behavior symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>package</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>block class</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process class</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>block</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>process</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>procedure</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A diagram listed in the first column can contain the ticked symbols in the other columns. For example the process symbol can contain the additional heading symbol, the text symbol and all the behavior symbols. The behavior symbols are all symbols described in “Behavior” on page 13.
13 - Textual representation

Storage format follows XML (eXtensible Markup Language standard from W3C available at http://www.w3c.org) standard with the following DTD (Document Type Definition):

```xml
<!-- Entity for booleans -->
<!-- ---------------------------------- -->
<!ENTITY % boolean "(TRUE|FALSE)"/>

<!-- Entities for symbol types -->
<!-- ---------------------------------- -->
<!ENTITY % sdlSymbolTypes1
  "sdlSysDgmFrm|sdlSysTypeDgmFrm|sdlBlkDgmFrm|sdlBlkTypeDgmFrm|sdlBlkType|sdlBlk|sdlBlkTypeInst|sdlPrcsType|sdlPrcs|sdlPrcsTypeInst">
<!ENTITY % sdlSymbolTypes2
  "sdlInherits|sdlPrcsTypeDgmFrm|sdlPrcsDgmFrm|sdlPrcdDgmFrm|sdlStart|sdlState|sdlInputSig|sdlSendSig|sdlSaveSig|sdlContSig">
<!ENTITY % sdlSymbolTypes3
  "sdlTask|sdlDecision|sdlTransOpt|sdlJoin|sdlText|sdlComment|sdlTextExt|sdlCnctrIn|sdlPrcsCreation|sdlStop">
<!ENTITY % sdlSymbolTypes4
  "sdlInitTimer|sdlResetTimer|sdlSemDecl|sdlSemTake|sdlSemGive|sdlPrcdProto|sdlPrcdDecl|sdlPrcdCall|sdlPrcdStart|sdlPrcdReturn">
<!ENTITY % mscSymbolTypes
  "mscExternalFrm|mscInlineExpr|mscLifeline|mscSemaphore|mscLostMsg|mscFoundMsg|mscComment|mscGenNameArea|mscText|mscAbsTimeConstr|mscCondition|mscMscRef|mscInlineExprZone">
<!ENTITY % hmscSymbolTypes
  "hmscDgmFrm|hmscParallel|hmscStart|hmscEnd|hmscCondition|hmscMscRef|hmscAlternativePoint">
<!ENTITY % mscdocSymbolTypes
  "mscdocDgmFrm|mscdocMscRef|mscdocHeader">
<!ENTITY % SymbolType
  "(%sdlSymbolTypes1;|%sdlSymbolTypes2;|%sdlSymbolTypes3;|%sdlSymbolTypes4;|%mscSymbolTypes;|%hmscSymbolTypes;|%mscdocSymbolTypes;)">

<!-- Entity for lifeline component type -->
<!-- ---------------------------------- -->
<!ENTITY % LifelineComponentType "(norm|susp|meth|coreg|act)"/>

<!-- Entity for time interval type -->
<!-- ---------------------------------- -->
<!ENTITY % TimeIntervalType "(start|end|timeout|constraint)"/>

<!-- Entity for connector types -->
<!-- ---------------------------------- -->
<!ENTITY % ConnectorType
  "(void|chnl|chnlgate|sdlarrow|mscvoid|mscgate|mscarrowgate|hmscarrow)"/>
```
<!-- Entity for side for connectors -->
<!-- --------------------------------------------- -->

<!ENTITY % Side "(n|s|w|e|x|y)">

<!-- Entity for end types for connectors -->
<!-- --------------------------------------------- -->

<!ENTITY % ConnectorEndType "(voidend|arrow|midarrow)">

<!-- Entity for link segment orientation -->
<!-- --------------------------------------------- -->

<!ENTITY % Orientation "(h|v)">

<!-- Entity for link types -->
<!-- --------------------------------------------- -->

<!ENTITY % LinkType "(sbvoid|dbvoid|ssvoid|dsvoid|chnl|dec|transopt|msg|rtn|instcre)">

<!-- Entity for diagram types -->
<!-- --------------------------------------------- -->

<!ENTITY % DiagramType "(sys|systype|blk|blktype|prcs|prcstype|prcd|msc|hmsc|mscdoc|class|usec|act|state|comp |seq|coll|depl)">

<!-- Element for text in symbols/links/... -->
<!-- --------------------------------------------- -->

<!ELEMENT Text (#PCDATA)>
<!ATTLIST Text
  id CDATA "0"
>

<!-- Element for lifeline symbol components (MSC specific) -->
<!-- --------------------------------------------- -->

<!ELEMENT LifelineComponent (Text?)>
<!ATTLIST LifelineComponent
  type %LifelineComponentType; #REQUIRED
  height CDATA #REQUIRED
  color CDATA "$000000"
  width CDATA "-1"
>

<!-- Element for lifeline symbol time intervals (MSC specific) -->
<!-- --------------------------------------------- -->
<!ELEMENT TimeInterval (Text)>  
<!ATTLIST TimeInterval  
  type %TimeIntervalType; #REQUIRED  
  startpos CDATA #REQUIRED  
  endpos CDATA "-1"  
  offset CDATA #REQUIRED  
  color CDATA "#000000"  
>  
<!-- Element for spanned lifelines for spanning symbols (MSC specific) -->  
<!-- ================================================================= -->  
<!ELEMENT SpannedLifeline EMPTY>  
<!ATTLIST SpannedLifeline  
  lifelineId IDREF #REQUIRED  
>  
<!-- Element for inline expression zones (MSC specific) -->  
<!-- ============================================================== -->  
<!ELEMENT Zone EMPTY>  
<!ATTLIST Zone  
  zoneSymbolId IDREF #REQUIRED  
>  
<!-- Element for symbols -->  
<!-- ============================================================== -->  
<!-- The "LifelineComponent" and "TimeInterval" components and the "dies" attribute are only for lifelines symbols -->  
<!-- The "Zone" component is only for inline expression symbols -->  
<!-- The "SpannedLifeline" component is only for spanning symbols in MSC diagrams -->  
<!ELEMENT Symbol (Text, (((LifelineComponent*), (TimeInterval*)) | ((SpannedLifeline*), (Zone*))) | (Symbol*))>  
<!ATTLIST Symbol  
  symbolId ID #REQUIRED  
  type %SymbolType; #REQUIRED  
  xCenter CDATA #REQUIRED  
  yCenter CDATA #REQUIRED  
  fixedDimensions %boolean; "FALSE"  
  width CDATA "10"  
  height CDATA "10"  
  dies %boolean; "FALSE"  
  color CDATA "#000000"  
>  
<!-- Element for connectors -->  
<!-- ============================================================== -->  
<!ELEMENT Connector (Text, Text)>  
<!ATTLIST Connector  
  attachedSymbolId IDREF #REQUIRED  
  type %ConnectorType; #REQUIRED  
  isOutside %boolean; #REQUIRED  
  side %Side; #REQUIRED  
>
<!DOCTYPE SDL-RT SYSTEM "SDL-RT.dtd">

<ELEMENT LinkSegment EMPTY>
<!ATTLIST LinkSegment
  orientation %Orientation; #REQUIRED
  length CDATA #REQUIRED
>
</-- Element for link segments -->
</-- ------------------------ -->

<ELEMENT Link (Text, Connector, Connector, LinkSegment*)>
<!ATTLIST Link
  type %LinkType; #REQUIRED
  textSegmentNum CDATA #REQUIRED
  color CDATA "#000000"
>
</-- Element for links -->
</-- ---------------- -->

<ELEMENT PageSpecification EMPTY>
<!ATTLIST PageSpecification
  pageWidth CDATA "21"
  pageHeight CDATA "29.7"
  topMargin CDATA "1.5"
  bottomMargin CDATA "1.5"
  leftMargin CDATA "1.5"
  rightMargin CDATA "1.5"
  pageFooter %boolean; "TRUE"
>
</-- Element PageSpecification -->
</-- ---------------- -------- -->

<!ELEMENT Diagram (PageSpecification?, Symbol, Link*)>
<!ATTLIST Diagram
  type %DiagramType; #REQUIRED
  pageWidth CDATA "21"
  pageHeight CDATA "29.7"
  nbPagesH CDATA "1"
  nbPagesV CDATA "1"
  cellWidthMm CDATA "5"
>
</-- Element for diagrams -->
</-- ---------------- -------- -->
14 - Example systems

14.1 - Ping Pong

---

This example system is a basic send and receive test. First two processes ping and pPong are created. pPing receives start message from the environment and the game starts. To slow it down a bit a timer has been introduced.

---

**Ping pong system view**
Ping process
Pong process
MSC trace of the ping pong system
14.2 - A global variable manipulation

This example shows how to handle a global variable. Both processes try to modify a global variable in their start transition. To do so they use the semaphore dedicated to this global variable. When they are done they give back the semaphore so that another process can access the variable. In this example we have put timers in each process so that they keep the semaphore long enough to have a conflict while accessing the global variable. Note the global variable is defined in an external C file and resolved at link time.

```
extern int myGlobalVariable;

GLOBAL mySemaphore
(PROD, INITIAL, FULL)
```

Global variable manipulation example system
Process A
Process $B$
MSC trace of the global variable manipulation
15 - Keywords

The following keywords have a meaning at in some specific SDL-RT symbols listed below:

<table>
<thead>
<tr>
<th>keywords</th>
<th>concerned symbols</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRIO</td>
<td>Task definition</td>
</tr>
<tr>
<td></td>
<td>Task creation</td>
</tr>
<tr>
<td></td>
<td>Continuous signal</td>
</tr>
<tr>
<td>TO_NAME, TO_ID, TO_ENV, VIA</td>
<td>Message output</td>
</tr>
<tr>
<td>FOREVER, NO_WAIT</td>
<td>semaphore manipulation</td>
</tr>
<tr>
<td>&gt;, &lt;, &gt;=, &lt;=, !=, ==, true, false, else</td>
<td>decision branches</td>
</tr>
<tr>
<td>USE SDL_MESSAGE_LIST</td>
<td>text symbol</td>
</tr>
</tbody>
</table>

Table 2: Keywords in symbols
16 - Differences with classical SDL

It is difficult to list all the differences between SDL-RT and SDL even though an SDL developer would understand SDL-RT and vice versa. Still to be able to clearly state the differences between these languages we will pinpoint the main differences in the paragraphs below.

16.1 - Data types

This is the most significant difference between SDL and SDL-RT. Classical SDL has its own data types and syntax where SDL-RT basically uses ANSI C language. Some symbols have a specific syntax with SDL-RT since there is no C equivalent instruction such as output, input, save, or semaphore manipulations.

The advantages are obvious:
- the syntax is known by all real time developers,
- it implicitly introduces the concept of pointers that does not exist in SDL,
- it eases integration of legacy code where it is quite tricky to do from classical SDL,
- and last but not least it makes code generation out of SDL-RT quite straightforward.

16.2 - Semaphores

Semaphore is a key concept in real time systems that classical SDL misses. Specific semaphore symbols have been introduced in SDL-RT to answer the real time developer needs.

16.3 - Inputs

Classical SDL has nice concepts when it comes to dealing with message exchanges. But these concepts are not so interesting in real time development and are quite tricky to implement on a real target or operating system. That is why SDL-RT has removed the following concepts: enabling conditions when receiving a message, internal messages, two levels priority messages.

16.4 - Names

Classical SDL uses exotic names for some well known concepts such as "signal" where it is basically related to a "message". Since "message" is the usual name in Real Time Operating Systems SDL-RT uses the same term.

When it comes to object orientation classical SDL talks about "type" instead of the usual "class" term. SDL-RT uses the common developer word "class".

16.5 - Object orientation

Classical SDL uses "virtual", "redefined", and "finalized" when it comes to object oriented concepts. For example a super class should specify a transition is "virtual" so that the sub class is
allowed "redefine" or "finalize" it. This is C++ like but actually quite painful when it comes to write and does not make things any clearer. SDL-RT takes the Java notation instead where there is no need to specify anything to be able to redefine it in a sub class.
17 - Modifications from previous releases

17.1 - Semaphore manipulation

17.1.1 V1.0 to V1.1
The semaphore take now returns a status that indicates if the take attempt timed out or was successful. The semaphore lifeline gets grayed when the semaphore is unavailable.

17.2 - Object orientation

17.2.1 V1.1 to V1.2
There has been an error in the object orientation chapter: it is not possible to declare a process class or a block class in a block class definition diagram.

17.3 - Messages

17.3.1 V1.1 to V1.2
- Messages now needs to be declared.
- Message parameters are now typed with C types.
- Parameter length can be omitted if the parameter is structured. Then the length is implicitly the sizeof the parameter type.
- The VIA concept has been introduced.

17.4 - MSC

17.4.1 V1.1 to V1.2
- Saved messages representation introduced.
18 - Lexical rules

A subset of the BNF (Backus-Naur Form) is used in these 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
## 19 - Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANSI</td>
<td>American National Standards Institute</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
</tr>
<tr>
<td>MSC</td>
<td>Message Sequence Chart</td>
</tr>
<tr>
<td>RTOS</td>
<td>Real Time Operating System</td>
</tr>
<tr>
<td>SDL</td>
<td>Specification and Description Language</td>
</tr>
<tr>
<td>SDL-RT</td>
<td>Specification and Description Language - Real Time</td>
</tr>
</tbody>
</table>
20 - Index

A

Action
  symbol 22
Action symbol
  MSC symbol 51
Additional heading symbol 30
Agents 8

B

Block
  class 58

C

call
  procedure 25
channels 10
Class
  block 58
  process 60
Comment 27
  MSC symbol 51
Connectors 26
Continuous signal 21
Coregion 48
creation
  task 25

D

Data type
  difference with classical SDL 82
Data types 54
Decision 22

Declaration
  message 34
  procedure 33
  process 32
  semaphore 35
  timer 35
  variables 29

Diagram
  contained symbols 68

E

else
  decision 23
  keyword 81

Environment
  definition 8
  message output 18

Extension 28

F

false
  decision 23
  keyword 81
  transition option 26

FOREVER
  keyword 81

G

give
  semaphore 24
H
HMSC 52

I
if 22
ifdef 26
Input
difference with classical SDL 82
message 15
instance
MSC 36

K
Keywords 81

L
Lexical rules 85

M
Memory
management 55
Message
communication principles 10
declaration 34
input 15
memory management 55
MSC 39
output 16
parameters 55
save 21
MSC 36
action 51
agent instance 36
comment 51
reference 49
semaphore 37
text symbol 51

N
Naming
convention 57
difference with classical SDL 82
syntax 56
NO_WAIT
keyword 81

O
Object
concept 58
difference with classical SDL 82
OFFSPRING
procedure 25
output 16

P
Package 58
PARENT
procedure 25
PRIO
continuous signal 22
keyword 81
Procedure
call 25
declaration 33
return 29
start 29
Process
behavior 13
class 60
declaration 32
R

reference
    MSC 49
return
    procedure 29

S

save 21
SDL_MESSAGE_LIST
    keyword 81
SDL-RT
    Lexical rules 85
Semaphore
    declaration 35
difference with classical SDL 82
give 24
global variable 55
MSC 37
take 23
SENDER
    procedure 25
Start
    procedure 29
    symbol 13
timer 24
State 13
MSC 42
Stop
    symbol 14
timer 24
Storage format 69
Symbol
    additional heading 30
    in diagram 68
    ordering 30
text 29
Synchronous calls
    MSC 41
System 8

T

take
    semaphore 23
Task
    creation symbol 25
Text
    MSC symbol 51
    symbol 29
Time interval
    MSC 46
Timer
    declaration 35
    MSC 44
    start 24
    stop 24
TO_ENV 18
    keyword 81
TO_ID 16
    keyword 81
TO_NAME 17
    keyword 81
Transition option 26
true
    decision 23
    keyword 81
    transition option 26

U

USE
    keyword 81

V

VIA 18
    keyword 81
XML  
  data storage 69