Chapter 62 The Master Library

This chapter covers the following topics:

- **The structure of the source code of the SDL suite Master Library**
- **The runtime model used for the programs generated by the SDL to C compiler. The chapter also describes the data structures for representing the various SDL objects.**
- **The memory requirements for applications**
- **How to make a customized Master Library**
- **The compilation switches which affect the properties of the Master Library**

Note that the Master Library is only used together with the Cadvanced/Cbasic SDL to C Compiler. It cannot be used together with the Cmicro SDL to C Compiler.

Chapter 62 The Master Library

Introduction

This chapter describes the source code of the runtime library for applications generated by the Cadvanced/Cbasic SDL to C Compiler. Applications generated by the Cmicro SDL to C Compiler are **not** covered.

The chapter covers basically two topics:

1. The sections "File Structure" on page 2951, "The Symbol Table" on page 2954, and "The SDL Model" on page 2992 describe the runtime model for programs generated by the SDL Cadvanced/Cbasic SDL to C Compiler.

   Mainly it is the data structure used to represent various SDL objects that is discussed, both from the static point of view (the type definitions), and from the dynamic point of view (what information it represents and how it is initialized, changed, and used).

   The full runtime model that is used during simulations (with the monitor) is described. From this model, an optimization is made to obtain an application (not using the monitor). The optimization is discussed under "Compilation Switches" on page 3041.

2. In the sections "Compilation Switches" on page 3041, "Creating a New Library" on page 3061, and "Adaptation to Compilers" on page 3069, different aspects on how to make new versions of the runtime library are discussed.

   The compilation switches treated in the section "Compilation Switches" on page 3041 are used to determine the properties of the runtime library and the generated C code, while section "Creating a New Library" on page 3061 shows how to make new versions of the runtime library using for example new combinations of compilation switches.

   In the section "Adaptation to Compilers" on page 3069, porting issues are discussed.

File Structure

The runtime library is structured into a number of files. These are:

- `scttypes.h`
- `sctlocal.h`
- `sctpred.h`
- `sctsdl.c`
- `sctpred.c`
- `sctutil.c`
- `sctmon.c`
- `sctpost.c`
- `sctos.c`
- `post.h, sdt.h, itex.h, dll.h`

On UNIX:

- `post.o`

In Windows:

- `libpost.lib` (the statically linked library)
- `post.lib` (for dynamically linking)
- `post.dll` (the dynamically linked library)

On UNIX, all files can be found in the directory `$telelogic/sdt/sdtdir/`*`<machine dependent dir>`*`/INCLUDE` where *<machine dependent dir>* is for example `sunos5sdtdir` on SunOS 5, and `hppasdtdir` on HP.

In Windows, all files can be found in the directory *`<installation directory>`*`\sdt\sdtdir\wini386\include`. There are different kernel directories for the Borland and the Microsoft compiler. The Borland directories have the prefix `scta` while the Microsoft compiler directories have the prefix `sctam`.

Description of Files

scttypes.h

This file contains type definitions and extern declarations of variables and functions. The file is included by `sctsdl.c, sctpred.c, sctutil.c, sctmon.c, sctpost.c, sctos.c`, and by each generated C file.

sctlocal.h

This file contains type definitions and extern declarations of variables and functions that are used only in the kernel. This file is not included in generated code.

sctpred.h

This file contains type definitions and extern declarations handling the predefined data types in SDL (except PId, which is in `scttypes.h`). This file is included in generated code via `scttypes.h`.

sctsdt.c

In this file the implementation of the SDL operations can be found, together with the functions used for scheduling. In more detail, this file contains groups of functions for:

- Handling and reporting SDL dynamic errors
- SDL operations, such as Output, Create, Stop, Nextstate, Set, Reset, together with help functions for these activities
- Initialization and the main loop (the scheduler).

sctpred.c

The functions implementing the operations defined in the SDL predefined data types can be fund in this file. Operators for PId is implemented in `sctsdl.c`.

sctutil.c

This file contains basic read and write functions together with functions to handle reading and writing of values of abstract data types, including the predefined data types. It also contains the functions for MSC trace.

sctmon.c

The `sctmon.c` file contains the functions that implement the monitor interface, that is, interpreting and executing monitor commands.

sctpost.c

This file contains all the basic functions that are used to connect a simulator with the other parts of the SDL suite.

sctos.c

In this file, some functions that represent the dependencies of hardware, operating system and compiler are placed.

The basic functions necessary for an application are a function to read the clock and a function to allocate memory.

To move a generated C program plus the runtime library to a new platform (including a new compiler), the major changes are to be made in this file, together with writing a new section in `scttypes.h` to describe the properties of the new compiler.

post.h and sdt.h

These files are included in `sctpost.c` if the communication mechanism with other the SDL suite applications should be part of the actual object code version of the library. The file `post.h` contains the function interface, while `sdt.h` contains message definitions.

Caution!

Windows only: When linking with the PostMaster's dynamically linked libraries (`post.lib` and `post.dll`), the environment variable `USING_DLL` must be defined before including `post.h`. Example:

```
#define USING_DLL
#include "post.h"
#undef USING_DLL
```

post.o (post.lib in Windows)

This file contains the implementation of functions needed to send messages, via the Postmaster, to other tools in the SDL suite.

The Symbol Table

The symbol table is used for storing information mainly about the static properties of the SDL system, such as the block structure, connections of channels and the valid input signal set for processes. Some dynamic properties are also placed in the symbol table; for example the list of all active process instances of a process instance set. This is part of the node representing the process instance set.

The nodes in the symbol table are structs with components initialized in the declaration. During the initialization of the application, in the `yInit` function in generated code, a tree is built up from these nodes.

Symbol Table Tree Structure

The symbol table is created in two steps:

1. First, symbol table nodes are declared as structs with components initialized in the declaration (in generated code).
2. Then, the `yInit` function (in generated code) updates some components in the nodes and builds a tree from the nodes. This operation is not needed in an application!

The following names can be used to refer to some of the nodes that are always present. These names are defined in `scttypes.h`.

```
xSymbolTableRoot
xEnvId
xSrtN_SDL_Bit
xSrtN_SDL_Bit_String
xSrtN_SDL_Boolean
xSrtN_SDL_Character
xSrtN_SDL_Charstring
xSrtN_SDL_Duration
xSrtN_SDL_IA5String
xSrtN_SDL_Integer
xSrtN_SDL_Natural
xSrtN_SDL_Null
xSrtN_SDL_NumericString
xSrtN_SDL_Object_Identifier
xSrtN_SDL_Octet
xSrtN_SDL_Octet_String
xSrtN_SDL_PId
xSrtN_SDL_PrintableString
xSrtN_SDL_Real
xSrtN_SDL_Time
xSrtN_SDL_VisibleString
```

`xSymbolTableRoot` is the root node in the symbol table tree. Below this node the system node is inserted. After the system node, there is a node representing the environment of the system (`xEnvId`). Then there is one node for each package referenced from the SDL system. This is true also for the package predefined containing the predefined data types. The nodes for the predefined data types, that are sons to the node for the package predefined, can be directly referenced by the names `xSrtN_SDL_xxx`, according to the list above.

Nodes in the symbol table are placed in the tree exactly according its place of declaration in SDL. A node that represent an item declared in a block is placed as a child to that block node, and so on. The hierarchy in the symbol table tree will directly reflect the block structure and declarations within the blocks and processes.

A small example can be found in Figure 550 on page 3029. The following node types will be present in the tree:

Node Type	**Description**
`xSystemEC`	Represent the system or the system instance.
`xSystemTypeEC`	Represents a system type.
`xPackageEC`	Represents a package.
`xBlockEC`	Represent blocks and block instances.
`xBlockTypeEC`	Represents a block type.
`xBlockSubstEC`	Represents a block substructure and can be found as a child of a block node.
`xProcessEC`	Represent processes and process instances. The environment process node is placed after the system node and is used to represent the environment of the system.
`xProcessTypeEC`	Represents a process type.
`xServiceEC`	Represents a service or service instance.
`xServiceTypeEC`	Represents a service type.
`xProcedureEC`	Represents a procedure.

Node Type	Description
`xOperatorEC`	Represents an operator diagram, i.e. an ADT operator with an implementation in SDL.
`xCompoundStmtEC`	Represents a compound statement containing variable declarations.
`xSignalEC` `xTimerEC`	Represents a signal or timer type.
`xRPCSignalEC`	Represents the implicit signals (`pCALL`, `pREPLY`) used to implement RPCs.
`xSignalParEC`	There will be one signal parameter node, as a child to a signal, timer, and RPC signal node, for each signal or timer parameter.
`xStartUpSignalEC`	Represents a start-up signal, that is, the signal sent to a newly created process containing the actual FPAR parameters. An `xStartUpSignalEC` node is always placed directly after the node for its process.
`xSortEC` `xSyntypeEC`	Represents a newtype or a syntype.
`Struct Component Node (xVariableEC)`	A sort node representing a struct has one struct component node as child for each struct component in the sort definition.
`xLiteralEC`	A sort node similar to an `enum` type has one literal node as child for each literal in literal list.
`xStateEC`	Represents a state and can be found as a child to process and procedure nodes.
`xVariableEC` `xFormalParEC`	Represents a variable (DCL) or a formal parameter (FPAR) and can be found as children of process and procedure nodes.

`xChannelEC`
`xSignalRouteEC`
`xGate`
	Represents a channel, a signal route, or a gate.

`xRemoteVarEC`	Represents a remote variable definition.
`xRemotePrdEC`	Represents a remote procedure definition.

Node Type	Description
`xSyntVariableEC`	Represents implicit variables or components introduced by the Cadvanced/Cbasic SDL to C Compiler. Used only by the Validator.
`xSynonymEC`	Represent synonyms. Used only by the Validator.

The nodes (the struct variables) will in generated code be given names according to the following table:

```
ySysR_SystemName
        (system, system type, system instance)
yPacR_PackageName
yBloR_BlockName
        (block, block type, block instance)
yBSuR_SubstructureName
yPrsR_ProcessName
        (process, process type, process instance)
yPrdR_ProcedureName  (procedure, operator)
ySigR_SignalName
        (signal, timer, startup signal, RPC signal)
yChaR_ChannelName    (channel, signal route, gate)
yStaR_StateName
ySrtR_NewtypeName    (newtype, syntype)
yLitR_LiteralName
yVarR_VariableName
       (variable, formal parameter, signal
        parameter, struct component, synt.variable)
yReVR_RemoteVariable
yRePR_RemoteProcedure
```

In most cases it is of interest to refer to a symbol table node via a pointer. By taking the address of a variable according to the table above, i.e.

```
& yPrsR_Process1
```

such a reference is obtained. For backward compatibility, macros according to the following example is also generated for several of the entity classes:

```
#define yPrsN_ProcessName  (&yPrsR_ProcessName)
```

Types Representing the Symbol Table Nodes

The following type definitions, from the file `scttypes.h`, are used in connection with the symbol table:

```
typedef enum {
  xRemoteVarEC,
  xRemotePrdEC,
  xSignalrouteEC,
  xStateEC,
  xTimerEC,
  xFormalParEC,
  xLiteralEC,
  xVariableEC,
  xBlocksubstEC,
  xPackageEC,
  xProcedureEC,
  xOperatorEC,
  xProcessEC,
  xProcessTypeEC,
  xGateEC,
  xSignalEC,
  xSignalParEC,
  xStartUpSignalEC,
  xRPCSignalEC,
  xSortEC,
  xSyntypeEC,
  xSystemEC,
  xSystemTypeEC,
  xBlockEC,
  xBlockTypeEC,
  xChannelEC,
  xServiceEC,
  xServiceTypeEC,
  xCompoundStmtEC,
  xSyntVariableEC
  xMonitorCommandEC
}   xEntityClassType;


typedef enum {
  xPredef, xUserdef, xEnum,
  xStruct, xArray, xGArray, xCArray,
  xOwn, xORef, xRef, xString,
  xPowerSet, xGPowerSet, xBag, xInherits, xSyntype,
  xUnion, xUnionC, xChoice
}   xTypeOfSort;

typedef char  *xNameType;
typedef struct xIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
}  xIdRec;


                            /*BLOCKSUBSTRUCTURE*/
typedef struct xBlockSubstIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
}  xBlockSubstIdRec;

                                  /*LITERAL*/
typedef struct xLiteralIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   int               LiteralValue;
}  xLiteralIdRec;

                                  /*PACKAGE*/
typedef struct xPackageIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
#ifdef XIDNAMES
   xNameType         ModuleName;
#endif
}  xPackageIdRec;
```

```
                                    /*SYSTEM*/
typedef struct xSystemIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xIdNode          *Contents;
   xPrdIdNode       *VirtPrdList;
   xSystemIdNode     Super;
#ifdef XTRACE
   int               Trace_Default;
#endif
#ifdef XGRTRACE
   int               GRTrace;
#endif
#ifdef XMSCE
   int               MSCETrace;
#endif
}  xSystemIdRec;


                       /*CHANNEL,SIGNALROUTE,GATE*/
#ifndef XOPTCHAN
typedef struct xChannelIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xSignalIdNode    *SignalSet; /*Array*/
   xIdNode          *ToId;      /*Array*/
   xChannelIdNode    Reverse;
}  xChannelIdRec;   /* And xSignalRouteEC.*/
#endif


                                    /*BLOCK*/
typedef struct xBlockIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
```

```
   xIdNode            Parent;
#ifdef XIDNAMES
   xNameType          Name;
#endif
   xBlockIdNode       Super;
   xIdNode           *Contents;
   xPrdIdNode        *VirtPrdList;
   xViewListRec      *ViewList;
   int                NumberOfInst;
#ifdef XTRACE
   int                Trace_Default;
#endif
#ifdef XGRTRACE
   int                GRTrace;
#endif
#ifdef XMSCE
   int                MSCETrace;
   int                GlobalInstanceId;
#endif
}  xBlockIdRec;


                                  /*PROCESS*/
typedef struct xPrsIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode            First;
   xIdNode            Suc;
#endif
   xIdNode            Parent;
#ifdef XIDNAMES
   xNameType          Name;
#endif
   xStateIdNode      *StateList;
   xSignalIdNode     *SignalSet;
#ifndef XNOUSEOFSERVICE
   xIdNode           *Contents;
#endif
#ifndef XOPTCHAN
   xIdNode           *ToId; /*Array*/
#endif
   int                MaxNoOfInst;
#ifdef XNRINST
   int                NextNr;
   int                NoOfStaticInst;
#endif
   xPrsNode          *ActivePrsList;
   xptrint            VarSize;
#if defined(XPRSPRIO) || defined(XSIGPRSPRIO) || \
    defined(XPRSSIGPRIO)
   int                Prio;
#endif
   xPrsNode          *AvailPrsList;
#ifdef XTRACE
   int                Trace_Default;
```

```
#endif
#ifdef XGRTRACE
   int                GRTrace;
#endif
#ifdef XBREAKBEFORE
   char   *(*GRrefFunc) (int, xSymbolType *);
   int                MaxSymbolNumber;
   int                SignalSetLength;
#endif
#ifdef XMSCE
   int                MSCETrace;
#endif
#ifdef XCOVERAGE
   long int          *CoverageArray;
   long int           NoOfStartTransitions;
   long int           MaxQueueLength;
#endif
   void              (*PAD_Function) (xPrsNode);
   void              (*Free_Vars) (void *);
   xPrsIdNode         Super;
   xPrdIdNode        *VirtPrdList;
   xBlockIdNode       InBlockInst;
#ifdef XBREAKBEFORE
   char              *RefToDefinition;
#endif
}  xPrsIdRec;


#ifndef XNOUSEOFSERVICE
                                  /*SERVICE*/
typedef struct xSrvIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode            First;
   xIdNode            Suc;
#endif
   xIdNode            Parent;
#ifdef XIDNAMES
   xNameType          Name;
#endif
   xStateIdNode      *StateList;
   xSignalIdNode     *SignalSet;
#ifndef XOPTCHAN
   xIdNode           *ToId;
#endif
   xptrint            VarSize;
#ifdef XBREAKBEFORE
   char   *(*GRrefFunc) (int, xSymbolType *);
   int                MaxSymbolNumber;
   int                SignalSetLength;
#endif
#ifdef XCOVERAGE
   long int          *CoverageArray;
   long int           NoOfStartTransitions;
#endif
```

```
   xSrvNode          *AvailSrvList;
   void             (*PAD_Function) (xPrsNode);
   void             (*Free_Vars) (void *);
   xSrvIdNode         Super;
   xPrdIdNode        *VirtPrdList;
}  xSrvIdRec;
#endif


                                  /*PROCEDURE*/
typedef struct xPrdIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode            First;
   xIdNode            Suc;
#endif
   xIdNode            Parent;
#ifdef XIDNAMES
   xNameType          Name;
#endif
   xStateIdNode      *StateList;
   xSignalIdNode     *SignalSet;
   xbool            (*Assoc_Function) (xPrsNode);
   void             (*Free_Vars) (void *);
   xptrint            VarSize;
   xPrdNode          *AvailPrdList;
#ifdef XBREAKBEFORE
   char            *(*GRrefFunc) (int, xSymbolType*);
   int                MaxSymbolNumber;
   int                SignalSetLength;
#endif
#ifdef XCOVERAGE
   long int          *CoverageArray;
#endif
   xPrdIdNode         Super;
   xPrdIdNode        *VirtPrdList;
}  xPrdIdRec;


typedef struct xRemotePrdIdStruct {
   xEntityClassType    EC;
#ifdef XSYMBTLINK
   xIdNode             First;
   xIdNode             Suc;
#endif
   xIdNode             Parent;
#ifdef XIDNAMES
   xNameType           Name;
#endif
   xRemotePrdListNode  RemoteList;
}  xRemotePrdIdRec;


                                /* SIGNAL, TIMER */
typedef struct xSignalIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xptrint           VarSize;
   xSignalNode      *AvailSignalList;
   xbool           (*Equal_Timer) (void *, void *);
#ifdef XFREESIGNALFUNCS
   void            (*Free_Signal) (void *);
#endif
#ifdef XBREAKBEFORE
   char             *RefToDefinition;
#endif
#if defined(XSIGPRIO) || defined(XSIGPRSPRIO) ||
defined(XPRSSIGPRIO)
   int               Prio;
#endif
}  xSignalIdRec;  /* and xTimerEC, xStartUpSignalEC,
                     and xRPCSignalEC.*/


                                  /*STATE*/
typedef struct xStateIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   int               StateNumber;
   xInputAction     *SignalHandlArray;
   int              *InputRef;
   xInputAction    (*EnablCond_Function)
                          (XSIGTYPE, void *);
   void            (*ContSig_Function)
                  (void *, int *, xIdNode *, int *);
   int               StateProperties;
#ifdef XCOVERAGE
   long int         *CoverageArray;
#endif
   xStateIdNode      Super;
#ifdef XBREAKBEFORE
   char             *RefToDefinition;
#endif
}  xStateIdRec;
```

```
                                           /*SORT*/
typedef struct xSortIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
#ifdef XFREEFUNCS
   void             (*Free_Function) (void **);
#endif
#ifdef XTESTF
   xbool            (*Test_Function) (void *);
#endif
   xptrint           SortSize;
   xTypeOfSort       SortType;
   xSortIdNode       CompOrFatherSort;
   xSortIdNode       IndexSort;
   long int          LowestValue;
   long int          HighestValue;
   long int          yrecIndexOffset;
   long int          typeDataOffset;
}  xSortIdRec;

                                           /*VARIABLE,...*/
typedef struct xVarIdStruct {
   xEntityClassType  EC;
#ifdef XSYMBTLINK
   xIdNode           First;
   xIdNode           Suc;
#endif
   xIdNode           Parent;
#ifdef XIDNAMES
   xNameType         Name;
#endif
   xSortIdNode       SortNode;
   xptrint           Offset;
   xptrint           Offset2;
   int               IsAddress;
}  xVarIdRec;    /* And xFormalParEC and
                    xSignalParEC.*/


typedef struct xRemoteVarIdStruct {
   xEntityClassType     EC;
#ifdef XSYMBTLINK
   xIdNode              First;
   xIdNode              Suc;
#endif
   xIdNode              Parent;
#ifdef XIDNAMES
   xNameType            Name;
#endif
   xptrint                 SortSize;
   xRemoteVarListNode      RemoteList;
}  xRemoteVarIdRec;
```

There are also pointer types defined for each of the `x`*`EC`*`IdStruct` according to the following example:

```
typedef XCONST struct xIdStruct  *xIdNode;
```

The type definitions above define the contents in the symbol table nodes. Each `x`*`EC`*`IdStruct`, where EC should be replaced by an appropriate string, have the first five components in common. These components are used to build the symbol table tree. To access these components, a pointer to a symbol table node can be type cast to any of the `xId`*`EC`*`Node` types. The type `xIdNode` is used as such general type, for example when traversing the tree.

The five components present in all `xIdNode` are:

- **`EC`** of type `xEntityClassType`. This component is used to determine what sort of SDL object the node represents. `xEntityClassType` is an enum type containing elements for all entity classes in SDL.
- **`First`**, **`Suc`**, and **`Parent`** of type `xIdNode`. These components are used to build the symbol table tree. `First` refers to the first child of the current node. `Suc` refers to the next brother, while `Parent` refers to the father node. Only `Parent` is needed in an application.
- **`Name`** of type `xNameType`, which is defined as `char *`. This component is used to represent the name of the current SDL object as a character string. Not needed in an application.

Next there are components depending on what entity class that is to be represented. Below we discuss the non-common elements in the other `x`*`EC`*`IdStruct`.

Package

- **`ModuleName`** of type `xNameType`. If the package is generated from ASN.1, this component holds the name of the ASN.1 module as a `char *`.

System, System Type

- **`Content`** of type `xIdNode *`. This component contains a list of all channels at the system level.
- **`VirtPrdList`** of type `xPrdIdNode *`. This is a list of all virtual procedures in this system instance.
- **`Super`** of type `xSystemIdNode`. This is a reference to the inherited system type. In a system this component in null. In a system instance it is a reference to the instantiated system type.
- **`Trace_Default`** of type `int`. This component contains the current trace value defined for the system.
- **`GRTrace`** of type `int`. This component contains the current GR (graphical) trace value defined for the system.
- **`MSCETrace`** of type `int`. This component contains the current MSCE (Message Sequence Chart Editor) trace value defined for the system.

Channel, Signal route, Gate

For channels, signal routes, and gates there are always two consecutive `xChannelIdNodes` in the symbol table, representing the two possible directions for a channel, signal route, or gate. The components are:

- **`SignalSet`** of type `xIdNode *`. This component represents the signal set of the channel in the current direction (a unidirectional channel has an empty signal set in the opposite direction).

  `SignalSet` is an array with components referring to the `xSignalIdNodes` that represent the signals which are members of the signal set. The last component in the array is always a `NULL` pointer (the value `(xSignalIdNode)0`).
- **`ToId`** of type `xIdNode *`. This is an array of `xIdNode`s, where each array component is a pointer to a symbol table node representing an SDL object, which this Channel/Signal route/Gate is connected to (connected to in the sense: to the SDL objects that signals are sent forward to).

  The SDL objects that may be referenced in `ToId` are channels, signal routes, gates, processes, and services. The last component in the array is always a `NULL` pointer (the value `(xIdNode)0`). See also "Channels and Signal Routes" on page 3028.

- **`Reverse`** of type `xChannelIdNode`. This is a reference to the symbol table node that represents the other direction of the same channel, signal route, or gate.

Block, Block Type, Block Instance

- **`Super`** of type `xBlockIdNode`. In a block, this component is `NULL`. In a block type this component is a reference to the block that this block inherits from (`NULL` if no inheritance). In a block instance, this is a reference to the block type that is instantiated.
- **`Contents`** of type `xIdNode *`. In a block instance, these components contains list of:
  - The process instantiations in the block
  - The signal routes in the block
  - The outgoing gates from the block
  - The processes in the block
  - The gates defined in process instantiations in the block.
- **`VirtPrdList`** of type `xPrdIdNode *`. This is a list of all virtual procedures in this block instance.
- **`ViewList`** of type `xViewListRec *`. This is a list of all revealed variables in the block or block instance.
- **`NumberOfInst`** of type `int`. This is the number of block instances in a block instance set. The component is thus only relevant for a block instance.
- **`Trace_Default`** of type `int`. This component contains the current value of the trace defined for the block.
- **`GRTrace`** of type `int`. This component contains the current value of the GR trace defined for the block.
- **`MSCETrace`** of type `int`. This component contains the current MSCE (Message Sequence Chart Editor) trace value defined for the block.
- **`GlobalInstanceId`** of type `int`. This component is used to store a unique id needed when performing MSCE trace.

Process, Process Type, Process Instance

- **`StateList`** of type `xStateIdNode *`. This is a list of references to the `xStateIdNodes` for this process or process type. Using the state value of an executing process, this list can be used to find the corresponding `xStateIdNode`.

- **`SignalSet`** of type `xIdNode *`. This represents the valid input signal set of the process or process type.

  `SignalSet` is an array with components that refer to `xSignalIdNodes` that represent the signals and timers which are part of the signal set. The last component in the array is always a `NULL` pointer (the value `(xSignalIdNode)0`).

- **`Contents`** of type `xIdNode *`. This is an array containing references to the `xSrvIdNodes` of the services and service instances in this process.

- **`ToId`** of type `xIdNode *`. This is an array of `xIdNode`, where each array component is a pointer to an IdNode representing an SDL object that this process or process instance is connected to (connected to in the sense: to the SDL objects that signals are sent forward to).

  The SDL objects that may be referenced in `ToId` are channels, signal routes, gates, processes, and services. The last component in the array is always a `NULL` pointer (the value `(xIdNode)0`). See also section “Channels and Signal Routes” on page 3028.

- **`MaxNoOfInst`** of type `int`. This represents the maximum number of concurrent processes that may exist according to the specification for the current process or process instance. An infinite number of concurrent processes is represented by -1.

- **`NextNo`** of type `int`. This is the instance number that will be assigned to the next instance that is created of this process instance set.

- **`NoOfStaticInst`** of type `int`.This component contains the number of static instance of this process instance set that should be present at start up. Used for process and process instance.

- **`ActivePrsList`** of type `xPrsNode *`. This is the address of a pointer to the “first” in the (single linked) list of active process instances of the current process or process instantiation.

The list is continued using the `NextPrs` component in the `xPrsRec` struct that is used to represent a process instance. The order in the list is such that the first created of the active process instances is last, and the latest created is first.

- **`VarSize`** of type `xptrint`. The size, in bytes, of the data area used to represent the process (the struct: `yVDef_`*`ProcessName`*).
- **`Prio`** of type `int`. This represents the process priority.
- **`AvailPrsList`** of type `xPrsNode`. This is the address to the avail list pointer for process instances that have stopped. The data area can later be reused in subsequent Create actions on this process or process instantiation.
- **`Trace_Default`** of type `int`. This component contains the current value of the trace defined for the process.
- **`GRTrace`** of type `int`. This component contains the current value of the GR trace defined for the process.
- **`GRrefFunc`**, which is a pointer to a function that, given a symbol number (number assigned to a process symbol), will return a string containing the SDT reference to that symbol.
- **`MaxSymbolNumber`** of type `int`. This component is the number of symbols contained in the current process or process type.
- **`SignalSetLength`** of type `int`. This component is the number of signals contained in the signal set of the current process or process type.
- **`MSCETrace`** of type `int`. This component contains the current MSCE (Message Sequence Chart Editor) trace value defined for the process.
- **`CoverageArray`** of type `long int *`. This component is used as an array over all symbols in the process. Each time a symbol is executed the corresponding array component is increased by 1.
- **`NoOfStartTransitions`** of type `long int`. This component is used to count the number of times the start transition of the current process is executed. This information is presented in the coverage tables.

- **MaxQueueLength** of type `long int`. This component is used to register the maximum input port length for any instance of the current process. The information is presented in the coverage tables.
- **PAD_Function**, which is a pointer to a function. This pointer refers to the `yPAD_`*`ProcessName`* function for the current process. This function is called when a process instance of this type is to execute a transition. The `PAD_Functions` will of course be part of generated code, as they contain the action defined in the process graphs.
- **Free_Vars**, which is a pointer to a function. This pointer refers to the `yFree_`*`ProcessName`* function for the current process. This function is called when the process performs a stop action to deallocate memory used by the local variables in the process.
- **Super** of type `xPrsIdNode`. In a process this component is `NULL`. In a process type this component is a reference to the process type that this process type inherits from (`NULL` if no inheritance). In a process instance set, this is a reference to the process type that is instantiated.
- **VirtPrdList** of type `xPrdIdNode *`. This is a list of all virtual procedures in this process instantiation.
- **InBlockInst** of type `xBlockIdNode`. This component is a reference to the block instance set (if any) that this process or process instantiation is part of.
- **RefToDefinition** of type `char *`. This is the SDT reference to this process.

Service, Service Type, Service Instance

- **StateList** of type `xStateIdNode *`. This is a list of the references to the `xStateIdNodes` for this service or service type. Using the state value of an executing service, this list can be used to find the corresponding `xStateIdNode`.
- **SignalSet** of type `xIdNode *`. This represents the valid input signal set of the service or service type.

  `SignalSet` is an array with components that refer to `xSignalIdNodes` that represent the signals and timers which are part of the signal set. The last component in the array is always a `NULL` pointer (the value `(xSignalIdNode)0)`.

- **ToId** of type `xIdNode *`. This is an array of `xIdNode`, where each array component is a pointer to an IdNode representing an SDL object that this service or service instance is connected to (connected to in the sense: to the SDL objects that signals are sent forward to).

  The SDL objects that may be referenced in `ToId` are channels, signal routes, gates, processes, and service. The last component in the array is always a `NULL` pointer (the value `(xIdNode)0`). See also section "Channels and Signal Routes" on page 3028.

- **VarSize** of type `xptrint`. The size, in bytes, of the data area used to represent the service (the struct: `yVDef_`*`ServiceName`*).

- **GRrefFunc**, which is a pointer to a function that, given a symbol number (number assigned to a service symbol), will return a string containing the SDT reference to that symbol.

- **MaxSymbolNumber** of type `int`. This component is the number of symbols contained in the current service or service type.

- **SignalSetLength** of type `int`. This component is the number of signals contained in the signal set of the current service or service type.

- **CoverageArray** of type `long int`. This component is used as an array over all symbols in the service. Each time a symbol is executed the corresponding array component is increased by 1.

- **NoOfStartTransitions** of type `long int`. This component is used to count the number of times the start transition of the current service is executed. This information is presented in the coverage tables.

- **AvailSrvList** of type `xSrvNode`. This is the address to the avail list pointer for service instances that have stopped. The data area can later be reused.

- **PAD_Function**, which is a pointer to a function. This pointer refers to the `yPAD_`*`ServiceName`* function for the current service. This function is called when a service instance of this type is to execute a transition. The `PAD_Functions` will of course be part of generated code, as they contain the action defined in the service graphs.

- **Free_Vars**, which is a pointer to a function. This pointer refers to the `yFree_`*`SeriveName`* function for the current service. This function is called when the service performs a stop action to deallocate memory used by the local variables in the service.

- **`Super`** of type `xSrvIdNode`. In a service this component is `NULL`. In a service type this component is a reference to the service type that this service type inherits from (`NULL` if no inheritance). In a service instantiation this is a reference to the service type that is instantiated.
- **`VirtPrdList`** of type `xPrdIdNode *`. This is a list of all virtual procedures in this service instantiation.

Procedure, Operator Diagram, Compound Statement

Note that operator diagrams and compound statements containing variable declarations are treated as procedures. However, such objects can, for example, not contain states.

- **`StateList`** of type `xStateIdNode *`. This is a list of references to the `xStateIdNodes` for this process or process type. Using the state value of an executing process, this list can be used to find the corresponding `xStateIdNode`.
- **`SignalSet`** of type `xIdNode *`. This represents the valid input signal set of the process or process type.

  `SignalSet` is an array with components that refer to `xSignalIdNodes` that represent the signals and timers which are part of the signal set. The last component in the array is always a `NULL` pointer (the value `(xSignalIdNode)0)`.
- **`Assoc_Function`**, which is a pointer to a function. This pointer refers to the `y`*`ProcedureName`* function for the current procedure. This function is called when the SDL procedure is called and will execute the appropriate actions. The `y`*`ProcedureName`* functions will, of course, be part of generated code as they contain the action defined in the procedure graphs.
- **Free_Vars**, which is a pointer to a function. This pointer refers to the `yFree_`*`ProcedureName`* function for the current procedure. This function is called when the procedure performs a return action to deallocate memory used by the local variables in the procedure.
- **`VarSize`** of type `xptrint`. The size, in bytes, of the data area used to represent the procedure (struct `yVDef_`*`ProcedureName`*).

- **`AvailPrdList`** of type `xPrdNode *`. This is the address of the avail list pointer for the data areas used to represent procedure instances. At a return action the data area is placed in the avail list and can later be reused in subsequent Calls of this procedure type.
- **`GRrefFunc`**, which is a pointer to a function that given a symbol number (number assigned to a procedure symbol) will return a string containing the SDT reference to that symbol.
- **`MaxSymbolNumber`** of type `int`. This component is the number of symbols contained in the current procedure.
- **`SignalSetLength`** of type `int`. This component is the number of signals contained in the signal set of the current procedure.
- **`CoverageArray`** of type `long int`. This component is used as an array over all symbols in the procedure. Each time a symbol is executed the corresponding array component is increased by 1.
- **`Super`** of type `xPrdIdNode`. This component is a reference to the procedure that this procedure inherits from (`NULL` if no inheritance).
- **`VirtPrdList`** of type `xPrdIdNode *`. This is a list of all virtual procedures in this procedure.

Remote Procedure

- **`RemoteList`** of type `xRemotePrdListNode`. This component is the start of a list of all processes that exports this procedure. This list is a linked list of `xRemotePrdListStructs`, where each node contains a reference to the exporting process.

Signal, Timer, StartUpSignal, and RPC Signals

- **`VarSize`** of type `xptrint`. The size, in bytes, of the data area used to represent the signal (the struct: `yPDef_`*`SignalName`*).
- **`AvailSignalList`** of type `xSignalNode *`. This is the address to the avail list pointer for signal instances of this signal type.
- **`Equal_Timer`**, which is a pointer to a function. This pointer only refers to a function when this node is used to represent a timer with parameters.

  In this case the referenced function can be used to investigate if the parameters of two timers are equal or not, which is necessary at reset actions. The `Equal_Timer` functions will be part of generated code. These functions are called from the functions `xRemoveTimer` and `xRemoveTimerSignal`, both defined in `sctsdl.c`

- **`Free_Signal`**, which is a function. This function takes a signal reference and returns any dynamic data referenced from the signal parameters to the pool of available memory.
- **`RefToDefinition`** of type `char *`. The SDT reference to the definition of the signal or timer.
- **Prio**, of type int. The priority of the signal.

State

- **`StateNumber`** of type `int`. The int value used to represent this state.
- **`SignalHandlArray`** of type `xInputAction` *. This component refers to an array of `xInputAction`, where `xInputAction` is an enum type with the possible values `xDiscard`, `xInput`, `xSave`, `xEnablCond`, `xPrioInput`.

  The array will have the same number of components as the `SignalSet` array in the node representing the process in which this state is contained. Each position in the `SignalHandlArray` represents the way the signal in the corresponding position in the `SignalSet` array in the process should be treated in this state.

  The last component in the `SignalHandlArray` is equal to `xDiscard`, which corresponds to the `0` value last in the `SignalSet`.

  If the `SignalHandlArray` contains the value `xInput`, `xSave`, or `xDiscard` at a given index, the way to handle the signal is obvious. If the `SignalHandlArray` contains the value `xEnablCond`, it is, however, necessary to calculate the enabling condition expression to know if the signal should cause an input or should be saved. This calculation is exactly the purpose of the `EnablCond_Function` described below.
- **`InputRef`** of type `int *`. This component is an array. If the `SignalHandlArray` contains `xInput`, `xPrioInput`, or `xEnablCond` at a certain index, this `InputRef` contains the symbol number for the corresponding input symbol in the graph.

- **`EnablCond_Function`**, which is a function that returns `xInputAction`. If the state contains any enabling conditions, this pointer will refer to a function. Otherwise it refers to `0`. An `EnablCond_Function` takes a reference to an `xSignalIdNode` (referring to a signal) and a reference to a process instance and calculates the enabling condition for the input of the current signal in the current state of the given process instance.

  The function returns either of the values `xInput` or `xSave`. The `EnablCond_Functions` will of course be part of generated code, as they contain enabling condition expressions. These functions are called from the function `xFindInputAction` in the file `sctsdl.c`. `xFindInputAction` is used by the `SDL_Output` and `SDL_Nextstate` functions.

- **`ContSig_Function`**, which is a function returning `int`. If the state contains any continuous signals, this pointer will refer to a function. Otherwise it refers to `0`.

- **`StateProperties`** of type `int`. In this component the three least significant bits are used to indicate:

  – If any enabling condition or continuous signal expression in the state contains a reference to an object that might change its value even though the process does not execute any actions.
  – If there are any priority inputs in the state.
  – If there are any virtual priority inputs in the state.

  Objects according to the first item in the list are: `Now`, `Active` (timer is active), `Import`, `View`, and `Sender`. `StateProperties` is used in the function `SDL_Nextstate` to take appropriate actions when a process enters a state.

- **`CoverageArray`** of type `long int`. This component is used as an array over the signalset (+1) of the process. Each time an input operation is performed, the corresponding array component is increased by 1. The last component, at index equal to the length of the signalset, is used to record the number of continuous signals "received" in the state. The information stored in this component is presented in the coverage table.

- **`Super`** of type `xPrdIdNode`. This component is a reference to the procedure that this procedure inherits from (`NULL` if no inheritance).

- **`RefToDefinition`** of type `char *`. The SDT reference to the definition of the state (one of the symbols where this state is defined).

Sort and Syntype

- **`Free_Function`**, which is a function. This function pointer is non-`0` for types represented using dynamic memory (Charstring, Octet_string, Strings, Bags, for example). The `Free_Functions` are used to return dynamic memory to the pool of dynamic memory.
- **`Test_Function`**, which is a function returning `xbool`. This function is non-0 for all types containing range conditions. The function pointers are used by the monitor system to check the validity of a value when assigning it to a variable.
- **`SortSize`** of type `xptrint`. This component represents the size, in bytes, of a variable of the current sort.
- **`SortType`** of type `xTypeOfSort`. This component indicates the type of sort. Possible values are: `xPredef`, `xUserdef`, `xEnum`, `xStruct`, `xArray`, `xGArray`, `xCArray`, `xRef`, `xString`, `xPowerSet`, `xBag`, `xGPowerSet`, `xInherits`, `xSyntype`, `xUnion`, `xUnionC`, and `xChoice`.

SortType is **`xArray, xGArray, xCArray`**

- **`CompOrFatherSort`** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the component sort.
- **`IndexSort`** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the index sort. In a xCArray the index sort in always Integer.
- In xGArray, **`LowestValue`** is used as the offset of `Data` in the `xxx_ystruct`.
  In xArray and xCArray it is 0.
- In xGArray, **`HighestValue`** is used as the size of the `xxx_ystruct`.
  In xArray it is 0.
  In xCArray it is the highest index, i.e. the Length - 1.
- In xGArray, **yrecIndexOffset** is used as the offset of `Index` in the `xxx_ystruct`.
  In xArray and xCArray it is 0.

- In xGArray, **yrecDataOffset** is used as the offset of `Data` in the type (i.e. the value representing the default value).
  In xArray and xCArray it is 0.

SortType is `xString, xGPowerSet, xBag`

- **`CompOrFatherSort`** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the component sort.
- **`LowestValue`** is used as the offset of `Data` in the `xxx_ystruct`.
- **`HighestValue`** is used as the size of the `xxx_ystruct`.

SortType is `xPowerSet, xRef, xOwn, xORef`

- **`CompOrFatherSort`** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the component sort.

SortType is `xInherits`

- **`CompOrFatherSort`**, of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the inherited sort.

SortType is `xSyntype`

- **`CompOrFatherSort`,** of type `xSortIdNode`. This is a pointer to the `SortIdNode` that represents the father sort (the newtype from which the syntype originates, even if it is a syntype of a syntype).
- **`IndexSort`**, of type `xSortIdNode`. This is a pointer to the SortIdNode that represents the represents the father sort (the newtype or syntype from which the syntype originates).
- **`LowestValue`**, of type `long int`. If the syntype can be used as an index in an array (translated to a C array) then this value is the lowest value in the syntype range, otherwise it is 0.
- **`HighestValue`**, of type `long int`. If the syntype can be used as an index in an array (translated to a C array) then this value is the highest value in the syntype range, otherwise it is 0. The `LowestValue` and `HighestValue` are used by the monitor when it handles arrays with this type as index type.

Variable, FormalPar, SignalPar, and Struct Components

- **`SortNode`** of type `xSortIdNode`. This component is a pointer to the SortIdNode that represents the sort of this variable or parameter.
- **`Offset`** of type `xptrint`. This component represents the offset, in bytes, within the struct that represents the process or procedure variables, the signal parameter, or the SDL struct. In other words, this is the relative place of this component within the struct.
- **`Offset2`** of type `xptrint`. For a formal parameter in a process this component represents the offset, in bytes, of a formal parameter in the `StartUpSignal`. For an exported variable in a process this component represents the offset, in bytes, of the exported value for this variable.
- **`IsAddress`** of type `int`. This component is only used for procedure and operator formal parameters and is then used to indicate if the parameter in IN or IN/OUT or a result variable.

Remote Variable

- **`SortSize`** of type `xptrint`. This component is the size of the type of the exported variables.
- **`RemoteList`** of type `xRemoteVarListNode`. This component is the start of a list of all processes that exports this variable. This list is a linked list of `xRemoteVarListStruct`s, where each node contains a reference to the exporting process and the `Offset` where to find the exported value.

Type Info Nodes

This section describes the most important implementation details regarding the type info node. Type info nodes are data structures that are used during run-time by the functions providing generic implementations of SDL operators. As the type info nodes contain essentially the same information as the xSortIdNodes, the type info nodes are used in more and more places in the code where xSortIdNode previously were used. In the longer perspective the xSortIdNode will be removed completely.

The type definitions that describe the type info nodes are listed in the `sctpred.h` file.

Each type info node is a struct that consists of:

- general components that are available for all type info nodes
- type-specific components that describe each specific type.

The following utility macros can be used to configure the type info nodes:

```
#ifndef T_CONST
#define T_CONST const
#endif

#ifndef T_SDL_EXTRA_COMP
#define T_SDL_EXTRA_COMP
#define T_SDL_EXTRA_VALUE
#endif

#ifndef T_SDL_USERDEF_COMP
#define T_SDL_USERDEF_COMP
#endif

#if defined(XREADANDWRITEF) && !defined(T_SDL_NAMES)
#define T_SDL_NAMES
#endif

#ifdef T_SDL_NAMES
#define T_SDL_Names(P) , P
#else
#define T_SDL_Names(P)
#endif

#ifdef T_SIGNAL_SDL_NAMES
#define T_Signal_SDL_Names(P) , P
#else
#define T_Signal_SDL_Names(P)
#endif

#ifdef T_SDL_INFO
#define T_SDL_Info(P) , P
#else
#define T_SDL_Info(P)
#endif

#ifndef XNOUSE_OPFUNCS
#define T_SDL_OPFUNCS(P) , P
#else
#define T_SDL_OPFUNCS(P)
#endif

struct tSDLFuncInfo;
```

General Components

The following components are available for all type info nodes. The definition of the components is only listed in this section, but it is valid for each type info node listed in the next section.

```
/* --- General type information for SDL types --- */

typedef T_CONST struct tSDLTypeInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
} tSDLTypeInfo;
```

- **TypeClass**: This component defines which type the info node describes. A list of available types and their corresponding values can be found in the enum type definition below:

```
typedef enum
{
  /*SDL - standard types*/
  type_SDL_Integer=128,
  type_SDL_Real=129,
  type_SDL_Natural=130,
  type_SDL_Boolean=131,
  type_SDL_Character=132,
  type_SDL_Time=133,
  type_SDL_Duration=134,
  type_SDL_Pid=135,
  type_SDL_Charstring=136,
  type_SDL_Bit=137,
  type_SDL_Bit_string=138,
  type_SDL_Octet=139,
  type_SDL_Octet_string=140,
  type_SDL_IA5String=141,
  type_SDL_NumericString=142,
  type_SDL_PrintableString=143,
  type_SDL_VisibleString=144,
  type_SDL_NULL=145,
  type_SDL_Object_identifier=146,

  /* SDL - standard ctypes */
  type_SDL_ShortInt=150,
  type_SDL_LongInt=151,
  type_SDL_UnsignedShortInt=152,
  type_SDL_UnsignedInt=153,
  type_SDL_UnsignedLongInt=154,
  type_SDL_Float=155,
  type_SDL_Charstar=156,
  type_SDL_Voidstar=157,
  type_SDL_Voidstarstar=158,

  /* SDL - user defined types */
  type_SDL_Syntype=170,
  type_SDL_Inherits=171,
  type_SDL_Enum=172,
  type_SDL_Struct=173,
  type_SDL_Union=174,
  type_SDL_UnionC=175,
  type_SDL_Choice=176,
  type_SDL_ChoicePresent=177,
  type_SDL_Powerset=178,
  type_SDL_GPowerset=179,
  type_SDL_Bag=180,
  type_SDL_String=181,
  type_SDL_LString=182,
  type_SDL_Array=183,
  type_SDL_Carray=184,
  type_SDL_GArray=185,
  type_SDL_Own=186,
  type_SDL_Oref=187,
  type_SDL_Ref=188,
  type_SDL_Userdef=189,
  type_SDL_EmptyType=190,

  /* SDL - signals */
  type_SDL_Signal=200,
  type_SDL_SignalId=201

} tSDLTypeClass;
```

- **OpNeeds**: This component contains four bits that give the properties of the type regarding assignment, equal test, free function, and initialization.
  - The first bit indicates if the type is a pointer that needs to be automatically freed, or if it contains a pointer that needs to be au-

  tomatically freed. If the first bit is set, it is necessary to look for memory to be freed inside of a value of this type.
- The second bit indicates if memcmp can be used to test if two values of this type are equal or not. If the bit is set, special treatment is needed.
- The third bit indicates if memcpy can be used to perform assign of this type. If the bit is set, special treatment is needed.
- The fourth bit indicates if this type needs to be initialized to anything else than 0.

The following macros can be used to test these properties:

```
#define NEEDSFREE(P) \
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)1)
#define NEEDSEQUAL(P) \
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)2)
#define NEEDSASSIGN(P) \
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)4)
#define NEEDSINIT(P) \
  (((tSDLTypeInfo *)(P))->OpNeeds & (unsigned char)8)
```

- **SortSize**: This component defines the size of the type.
- **OpFuncs**: This is a pointer to a struct containing references to specific assign, equal, free, read, and write functions. This component is only used in special cases. If assign, equal, free, read or write functions have been implemented using #ADT directives, information about this is stored in the OpFuncs field. The default value of the OpFuncs field is 0, but if you have provided any of these functions, the field will be a pointer to a tSDLFuncInfo struct. This struct will in turn refer to the provided functions.

```
typedef struct tSDLFuncInfo {
  void *(*AssFunc) (void *, void *, int);
  SDL_Boolean (*EqFunc) (void *, void *);
  void (*FreeFunc) (void **);
#ifdef XREADANDWRITEF
  char *(*WriteFunc) (void *);
  int (*ReadFunc) (void *);
#endif
} tSDLFuncInfo;
```

- **Name**: This is the name of the type as a string literal.
- **FatherScope**: This is a pointer to the IdNode for the scope that the type is defined in.
- **SortIdNode**: This is a pointer to the xSortIdNode that describes the same type. This field will in a longer perspective be removed.

Type-Specific Components

The following section lists the components that defines the type info nodes. Only the type-specific components are explained. The general components are listed and explained in the section above.

Enumeration types

```
/* ------------- Enumeration type --------------- */
typedef T_CONST struct {
  int              LiteralValue;
  char            *LiteralName;
} tSDLEnumLiteralInfo;

typedef T_CONST struct tSDLEnumInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char            *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
#ifdef XREADANDWRITEF
  int             NoOfLiterals;
  tSDLEnumLiteralInfo *LiteralList;
#endif
} tSDLEnumInfo;
```

- **NoOfLiterals**: The number of literals in the enum type.
- **LiteralList**: a pointer to an array of `tSDLEnumLiteralInfo` elements. This list implements a translation table between enum values and literal names as strings

Syntypes, types with inheritance, and Own, Ref, Oref instantiations

```
/* ------ Syntype, Inherits, Own, Oref, Ref ----- */
typedef T_CONST struct tSDLGenInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char            *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode          FatherScope;
  xSortIdNode      SortIdNode;
#endif
  tSDLTypeInfo    *CompOrFatherSort;
} tSDLGenInfo;
```

- **CompOrFatherSort**: Reference to the type info node of the father sort (syntype, inherits) or component sort (Own, Ref, Oref).

Powersets (implemented as unsigned in [ ])

```
/* ------------------ Powerset ------------------ */
typedef T_CONST struct tSDLPowersetInfoS {
  tSDLTypeClass    TypeClass;
  unsigned char    OpNeeds;
  xptrint          SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char            *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode          FatherScope;
  xSortIdNode      SortIdNode;
#endif
  tSDLTypeInfo    *CompSort;
  int              Length;
  int              LowestValue;
} tSDLPowersetInfo;
```

- **CompSort**: Reference to the type info node of the component sort.
- **Length**: The number of possible values in the component sort.
- **LowestValue**: The value of the lowest value in the component sort.

Structs

```
/* ------------------- Struct ------------------- */
typedef int (*tGetFunc) (void *);
typedef void (*tAssFunc) (void *, int);


typedef T_CONST struct {
  xptrint     OffsetPresent; /* 0 if not optional */
  void       *DefaultValue;
} tSDLFieldOptInfo;


typedef T_CONST struct {
  tGetFunc        GetTag;
  tAssFunc        AssTag;
} tSDLFieldBitFInfo;


typedef T_CONST struct {
  tSDLTypeInfo   *CompSort;
#ifdef T_SDL_NAMES
  char           *Name;
#endif
  xptrint         Offset;     /* ~0 for bitfield */
  tSDLFieldOptInfo *ExtraInfo;
} tSDLFieldInfo;


typedef T_CONST struct tSDLStructInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLFieldInfo  *Components;
  int             NumOfComponents;
} tSDLStructInfo;
```

- **Components**: An array of tSDLFieldInfo; one component in the array for each field of the struct.
- **NumOfComponents**: The number of fields in the struct.
- **CompSort in tSDLFieldInfo**: The reference to the type info node of the field sort.
- **Name in tSDLFieldInfo**: The name of the field as a string.
- **Offset in tSDLFieldInfo**: The offset of the field in the C struct that represents the SDL struct. This component is ~0 for bitfield in SDL (offsets cannot be calculated for bitfields).

- **ExtraInfo in tSDLFieldInfo**: The interpretation of this component depends on the properties in the SDL field.
  - if Offset is ~0, the field is a bitfield and ExtraInfo is a pointer to a tSDLFieldBitFInfo struct containing two functions to set and get the value of the bitfield.
  - if Offset is not ~0 and ExtraInfo != 0, the SDL field is either optional or has a default value. ExtraInfo is a pointer to a tSDLFieldOptInfo struct containing the offset for the Present flag (0 if not optional) and a pointer to the default value (0 if no default value).

Choice and #union

```
/* --------------- Choice, Union -------------- */

typedef T_CONST struct {
  tSDLTypeInfo        *CompSort;
#ifdef T_SDL_NAMES
  char                *Name;
#endif
} tSDLChoiceFieldInfo;

typedef T_CONST struct tSDLChoiceInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLChoiceFieldInfo *Components;
  int                  NumOfComponents;
  xptrint              OffsetToUnion;
  xptrint              TagSortSize;
#ifdef XREADANDWRITEF
  tSDLTypeInfo        *TagSort;
#endif
} tSDLChoiceInfo;
```

- **Components**: An array of tSDLChoiceFieldInfo; one component in the array for each field in the choice/#union.
- **NumOfComponents**: The number of fields in the choice/#union.

- **OffsetToUnion**: The offset to where the union, within the representation of the choice/#union, starts.
- **TagSortSize**: The size of the tag type.
- **TagSort**: A reference to the type info node of the tag sort.

Array and Carray

```
/* --------------- Array, CArray ---------------- */
typedef T_CONST struct tSDLArrayInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  int             Length;
#ifdef XREADANDWRITEF
  tSDLTypeInfo   *IndexSort;
  int             LowestValue;
#endif
} tSDLArrayInfo;
```

- **CompSort**: The reference to the type info node of the component sort.
- **Length**: The number of components in the array.
- **IndexSort**: The reference to the type info node of the index sort.
- **LowestValue**: The start value of the index range (as an int).

General arrays

A general array is an array that is represented as a linked list in C.

```
/* ------------------- GArray ------------------- */
typedef T_CONST struct tSDLGArrayInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *IndexSort;
  tSDLTypeInfo   *CompSort;
  xptrint         yrecSize;
  xptrint         yrecIndexOffset;
  xptrint         yrecDataOffset;
  xptrint         arrayDataOffset;
} tSDLGArrayInfo;
```

- **IndexSort**: The reference to the type info node of the index sort.
- **CompSort**: The reference to the type info node of the component sort.
- **yrecSize**: The size of the type SDLType_yrec.
- **yrecIndexOffset**: The offset of Index in type SDLType_yrec.
- **yrecDataOffset**: The offset of Data in type SDLType_yrec.
- **arrayDataOffset**: The offset of Data in type SDLType, where SDLType is the name in C of the translated array type.

General powersets, Bags, Strings and Object_identifier

```
/* -- GPowerset, Bag, String, Object_Identifier - */
typedef T_CONST struct tSDLGenListInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  xptrint         yrecSize;
  xptrint         yrecDataOffset;
} tSDLGenListInfo;
```

- **CompSort**: The reference to the type info node of the component sort.
- **yrecSize**: The size of the type SDLType_yrec
- **yrecDataOffset**: The offset of Data in type SDLType_yrec

Limited strings

A limited string is a string that is implemented as an array in C.

```
/* ------------------ LString ------------------- */
typedef T_CONST struct tSDLLStringInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  tSDLTypeInfo   *CompSort;
  int             MaxLength;
  xptrint         DataOffset;
} tSDLLStringInfo;
```

- **CompSort**: The reference to the type info node of the component sort.
- **MaxLength**: The maximum length of the string.
- **DataOffset**: The offset of Data in type SDLType, where SDLType is the name in C of the translated string type.

SDL type (C representation decided with a #ADT directive)

```
/* ------------------ Userdef ------------------- */
/* used for user defined types #ADT(T(h)) */
typedef T_CONST struct tSDLUserdefInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SDL_NAMES
  char           *Name;
#endif
#ifdef XREADANDWRITEF
  xIdNode         FatherScope;
  xSortIdNode     SortIdNode;
#endif
  T_SDL_USERDEF_COMP
} tSDLUserdefInfo;
```

SDL signal

A signal is treated in the same way as a struct.

```
/* ------------------- Signal ------------------- */
typedef T_CONST struct {
  tSDLTypeInfo       *ParaSort;
  xptrint             Offset;
} tSDLSignalParaInfo;

typedef T_CONST struct tSDLSignalInfoS {
  tSDLTypeClass   TypeClass;
  unsigned char   OpNeeds;
  xptrint         SortSize;
  struct tSDLFuncInfo *OpFuncs;
  T_SDL_EXTRA_COMP
#ifdef T_SIGNAL_SDL_NAMES
  char           *Name;
#endif
  tSDLSignalParaInfo *Param;
  int             NoOfPara;
} tSDLSignalInfo;
```

- **Param**: An array with a component of the type tSDLSignalParaInfo for each signal parameter type. For each parameter, the parameter sort is given as a reference to the type info node and as the offset for the parameter value within the struct representing the signal.
- **NoOfPara**: The number of parameters in the signal.

The SDL Model

Signals and Timers

Data Structure Representing Signals and Timers

A signal is represented by a struct type. The `xSignalRec` struct, defined in `scttypes.h`, is a struct containing general information about a signal except from the signal parameters. In `scttypes.h` the following information about signals can be found:

```
#ifdef XMSCE
#define GLOBALINSTID  int GlobalInstanceId;
#else
#define GLOBALINSTID
#endif

#if defined(XSIGPATH) && defined(XMSCE)
#define ENVCHANNEL  xChannelIdNode EnvChannel;
  /* Used if env split into channels in MSC trace */
#else
#define ENVCHANNEL
#endif

#ifdef XENV_CONFORM_2_3
#define XSIGNAL_VARP  void * VarP;
#else
#define XSIGNAL_VARP
#endif

define SIGNAL_VARS \
   xSignalNode   Pre; \
   xSignalNode   Suc; \
   int           Prio; \
   SDL_PId       Receiver; \
   SDL_PId       Sender; \
   xSignalIdNode NameNode; \
   GLOBALINSTID \
   ENVCHANNEL \
   XSIGNAL_VARP

typedef struct xSignalStruct  *xSignalNode;
typedef struct xSignalStruct {
   SIGNAL_VARS
}  xSignalRec;
```

The `xSignalNode` type is thus a pointer type which is used to refer to allocated data areas of type `xSignalRec`. The components in the `xSignalRec` struct are used as follows:

- **`Pre`** and **`Suc`**. These pointers are used to link a signal into the input port of the receiving process instance.

The input port is a doubly linked list of signals. `Suc` is also used to link a signal into the avail lists for the current signal type. This list can be found in the `SignalIdNode` that represents this signal type. If the signal is in the avail list `Pre` is `0`.

- **`Prio`** is used to represent the priority of the signal instance. Signal priorities are used by continuous signals and by ordinary signals if signal priorities are defined (signal priority is a possible extension provided in the product).
- **`Receiver`** is used to reference the receiver of the signal. It is either set in the output statement (OUTPUT TO), or calculated (OUTPUT without TO).
- **`Sender`** is the `PId` value of the sending process instance. This value is necessary to provide the SDL function SENDER.
- **`NameNode`** is a reference to the `xSignalIdNode` representing the signal type and thus defines the signal type of this signal instance.
- **`VarP`** is a pointer introduced via the macro `XSIGNAL_VARP` to make signal compatible with SDT 2.3. Normally this components is **not** present.
- **`EnvChannel`** is used to identify the outgoing channel in MSCE trace.
- **`GlobalInstanceId`** is used in the MSCE trace as a unique identification of the signal instance.

A signal without parameters are represented by a `xSignalStruct`, while for signals with parameters a struct type named `yPDef_`*`SignalName`* and a pointer type referencing this struct type (`yPDP_`*`SignalName`*) are defined in generated code. The struct type will start with the `SIGNAL_VARS` macro and then have one component for each signal parameter, in the same order as the signal parameters are defined. The components will be named Param1, Param2, and so on.

Example 491 ––

```
typedef struct {
    SIGNAL_VARS
    SDL_Integer  Param1;
    SDL_Boolean  Param2;
} yPDef_sig;
typedef yPDef_sig  *yPDP_sig;
```

These types would represent a signal sig(Integer, Boolean).

As all signals starts with the components defined in `SIGNAL_VARS` it is possible to type cast a pointer to a signal, to the `xSignalNode` type, if only the components in `SIGNAL_VARS` is to be accessed.

Allocation of Data Areas for Signals

In `sctos.c` there are two functions, `xGetSignal` and `xReleaseSignal`, where data areas for signal are handled:

```
xSignalNode xGetSignal(
  xSignalIdNode  SType,
  SDL_PId        Receiver,
  SDL_PId        Sender )

void xReleaseSignal( xSignalNode  *S )
```

`xGetSignal` takes a reference to the `SignalIdNode` identifying the signal type and two PId values (sending and receiving process instance) and returns a signal instance. `xGetSignal` first looks in the avail list for the signal type (the component `AvailSignalList` in the `SignalIdNode` for the signal type) and reuses any available signal there. Only if the avail list is empty new memory is allocated. The component `VarSize` in the `SignalIdNode` for the signal type provides the size information needed to correctly allocate the `yPDef_`*`SignalName`* even though the type is unknown for the `xGetSignal` function.

The function `xReleaseSignal` takes the address of an `xSignalNode` pointer and returns the referenced signal to the avail list for the signal type. The `xSignalNode` pointer is then set to `0`.

The function `xGetSignal` is used:

- In generated code (output, set, reset)
- In a number of places in the library:
  SDL_Create
  SDL_SimpleReset
  SDL_Nextstate (to handle continuous signals)
- In the postmaster communication section and in the monitor to obtain signal instances.

The function `xReleaseSignal` is used by:

- SDL_Nextstate
- SDL_Stop, in both cases to release the signal that initiated the transition.

Overview of Output and Input of Signals

In this subsection the signal handling operation is only outlined. More details will be given in the section treating processes. See "Output and Input of Signals" on page 3010.

Signal instances are sent using the function SDL_Output. That function takes a signal instance and inserts it into the input port of the receiving process instance.

If the receiver is not already in the ready queue (the queue containing the processes that can perform a transition, but which have not yet been scheduled to do so) and the current signal may cause an immediate transition, the process instance is inserted into the ready queue.

If the receiver is already in the ready queue or in a state where the current signal should be saved, the signal instance is just inserted into the input port.

If the signal instance can neither cause a transition nor should be saved, it is immediately discarded (the data area for the signal instance is returned to the avail list).

The input port is scanned during nextstate operations, according the rules of SDL, to find the next signal in the input port that can cause a transition. Signal instances may then be saved or discarded.

There is no specific input function, instead this behavior is distributed both in the runtime library and in the generated code. The signal instance that should cause the next transition to be executed is removed from the input port in the main loop (the scheduler), immediately before the `PAD` function for the current process is called. The `PAD` function is the function where the behavior of the process is implemented and is part of the generated code. The assignment of the signal parameters to local SDL variables is one of the first actions performed by the `PAD` function.

The signal instance that caused a transition is released and returned to the avail list in the nextstate or stop action that ends the current transition.

Timers and Operations on Timers

A timer with parameters is represented by a type definition, where the timer parameters are defined, in exactly the same way as for a signal definition, see “Data Structure Representing Signals and Timers” on page 2992. At runtime, all timers that are set and where the timer time has not expired, are represented by a `xTimerRec` struct and a signal instance:

```
#define TIMER_VARS \
   xSignalNode   Pre; \
   xSignalNode   Suc; \
   int           Prio; \
   SDL_PId       Receiver; \
   SDL_PId       Sender; \
   xSignalIdNode NameNode; \
   GLOBALINSTID \
   ENVCHANNEL \
   SDL_Time      TimerTime;

typedef xTimerRec  *xTimerNode;

typedef struct xTimerStruct {
   TIMER_VARS
}  xTimerRec;
```

The `TIMER_VARS` is and must be identical to the `SIGNAL_VARS` macro, except for the `TimerTime` component last in the macro. A timer with parameters have `yPDef_timername` and `yPDP_timername` types in generated code exactly as a signal (see previous section), except that `SIGNAL_VARS` is replaced by `TIMER_VARS`.

During its life-time a timer have two different appearances. First it is a timer waiting for the timer time to expire. In that phase the timer is inserted in the `xTimerQueue`. When the timer time expires the timer becomes a signal and is inserted in the input port of the receiver just like any other signal. Due to the identical typedefs for `xSignalRec` and `xTimerRec`, there are no problems with type casting between `xTimerNode` and `xSignalNode` types.

When a timer is treated as a signal the components in the `xTimerRec` are used in the same ways as for a `xSignalRec`. While the timer is in the timer queue, the components are used as follows:

- **`Pre`** and **`Suc`** are pointers used to link the `xTimerRec` into the timer queue (the queue of active timers, see below).
- **`TimerTime`** is the time given in the `Set` operation.

The queue mentioned above, the timer queue for active timers is represented by the component xTimerQueue in the variable `xSysD`:

```
    xTimerNode xTimerQueue;
```

The variable is initialized in the function `xInitKernel` in `sctsdl.c`. `xTimerQueue` is initialized it refers to the queue head of the timer queue.

The queue head is an extra element in the timer queue that does not represent a timer, but is introduced as it simplifies the algorithms for queue handling. The `TimerTime` component in the queue head is set to a very large time value (`xSysD.xMaxTime`).

The timer queue is thus a doubly linked list with a list head and it is sorted according to the timer times, so that the timer with lowest time is at the first position.

The `xTimerRec` structs are allocated and reused in the same way as signal.

From the SDL point of view, timers are handled in:

- Timer definitions
- Set and reset operations
- Timer outputs.

The timer output is the event when the timer time has expired and the timer signal is sent. After that, a timer signal is treated as an ordinary signal. These operations are implemented as follows:

```
void SDL_Set(
  SDL_Time     T,
  xSignalNode  S )
```

This function, which represents the `Set` operation, takes the timer time and a signal instance as parameters. It first uses the signal instance to make an implicit reset (see reset operation below) It then updates the TimerTime component in S and inserts S into the timer queue at the correct position.

The `SDL_Set` operation is used in generated code, together with `xGetSignal`, in much the same way as `SDL_Output`. First a signal instance is created (by `xGetSignal`), then timer parameters are assigned their values, and finally the `Set` operation is performed (by `SDL_Set`).

```
void SDL_Reset( xSignalNode  *TimerS )

void SDL_SimpleReset(
  xPrsNode       P,
  xSignalIdNode  TimerId )
```

Two functions are used to represent the SDL action reset. `SDL_SimpleReset` is used for timers without parameters and `SDL_Reset` for timers with parameters.

`SDL_Reset` uses the two functions `xRemoveTimer` and `xRemoveTimerSignal` to remove a timer in the timer queue and to remove a signal instance in the input port of the process. It then releases the signal instance given as parameter. This signal is only used to carry the parameter values given in the reset action.

The function `SDL_SimpleReset` is implemented in the same way as `SDL_Reset`, except that it creates its own signal instance (without parameters).

At a reset action the possibly found timer is removed from the timer queue and returned to the avail list. A found signal instance (in the input port) is removed from the input port and returned to the avail list for the current signal type.

```
static void SDL_OutputTimerSignal( xTimerNode  T )
```

The SDL_OutputTimerSignal is called from the main loop (the scheduler) when the timer time has expired for the timer first in the timer queue. The corresponding signal instance is then sent.

SDL_OutputTimerSignal takes a pointer to an xTimerRec as parameter, removes it from the timer queue and sends as an ordinary output using the function SDL_Output.

It can be checked if timer is active by using a call to the function SDL_Active. This function is used in generated code to represent the SDL operator active.

```
SDL_Boolean SDL_Active (
  xSignalIdNode TimerId,
  xPrsNode      P )
```

Note:

Only timers without parameters can be tested. This is a restriction in the Cadvanced/Cbasic SDL to C Compiler.

There is one more place where timers are handled. When a process instance performs a stop action all timers in the timer queue connected to this process instance are removed. This is performed by calling the function xRemoveTimer with the first parameter equal to 0.

Processes

Data Structure Representing Processes

A process instance is represented by two structs, an `xLocalPIdRec` and a struct containing both the general process data and the local variables and formal parameters of the process (`yVDef_`*`ProcessName`*), see also Figure 544. The reason for having both the `xLocalPIdRec` and the `yVDef_`*`ProcessName`* will be discussed under "Create and Stop Operations" on page 3007.

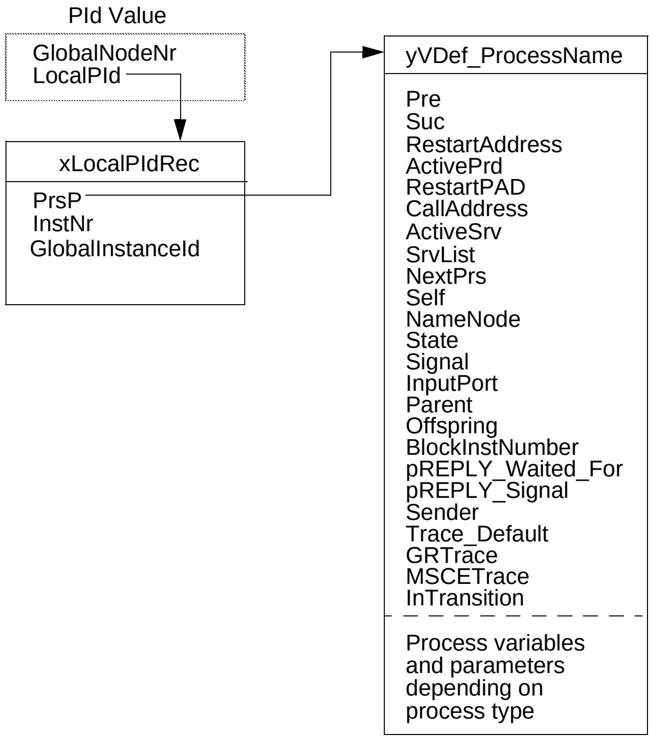

Figure 544: Representation of a process instance

The corresponding type definitions, which can be found in `scttypes.h`, are:

```
#ifdef XPRSSENDER
#define XPRSSENDERCOMP   SDL_PId  Sender;
#else
#define XPRSSENDERCOMP
#endif

#ifdef XTRACE
#define XTRACEDEFAULTCOMP   int Trace_Default;
#else
#define XTRACEDEFAULTCOMP
#endif

#ifdef XGRTRACE
#define XGRTRACECOMP   int GRTrace;
#else
#define XGRTRACECOMP
#endif

#ifdef XMSCE
#define XMSCETRACECOMP   int  MSCETrace;
#else
#define XMSCETRACECOMP
#endif

#if defined(XMONITOR) || defined(XTRACE)
#define XINTRANSCOMP   xbool InTransition;
#else
#define XINTRANSCOMP
#endif

#ifdef XMONITOR
#define XCALL_ADDR   int  CallAddress;
#else
#define XCALL_ADDR
#endif

#ifndef XNOUSEOFSERVICE
#define XSERVICE_COMP \
        xSrvNode ActiveSrv; xSrvNode SrvList;
#else
#define XSERVICE_COMP
#endif



#define PROCESS_VARS \
   xPrsNode       Pre; \
   xPrsNode       Suc; \
   int            RestartAddress; \
   xPrdNode       ActivePrd; \
   void (*RestartPAD) (xPrsNode  VarP); \
   XCALL_ADDR \
   XSERVICE_COMP \
   xPrsNode       NextPrs; \
   SDL_PId        Self; \
   xPrsIdNode     NameNode; \
   int            State; \
   xSignalNode    Signal; \
   xInputPortRec  InputPort; \
   SDL_PId        Parent; \
   SDL_PId        Offspring; \
   int            BlockInstNumber; \
   XSIGTYPE       pREPLY_Waited_For; \
   xSignalNode    pREPLY_Signal; \
   XPRSSENDERCOMP \
   XTRACEDEFAULTCOMP \
   XGRTRACECOMP \
   XMSCETRACECOMP \
   XINTRANSCOMP

typedef struct {
   xPrsNode      PrsP;
   int           InstNr;
   int           GlobalInstanceId;
}  xLocalPIdRec;

typedef xLocalPIdRec  *xLocalPIdNode;

typedef struct {
   int            GlobalNodeNr;
   xLocalPIdNode  LocalPId;
}  SDL_PId;

typedef struct xPrsStruct  *xPrsNode;

typedef struct xPrsStruct {
   PROCESS_VARS
} xPrsRec;
```

A `PId` value is thus a struct containing two components:

- The global node number
- A pointer to a `xLocalPIdRec` struct.

The use of the global node number is discussed in the chapter 58, *Building an Application*.

A **`xLocalPIdRec`** contains the following three components:

- **`PrsP`** of type `xPrsNode`. This component is a pointer to the `xPrsRec` struct that is part of the representation of the process instance.
- **`InstNr`** of type `int`. This is the instance number of the current process instance, which is used in the communication with the user in the monitor and in dynamic error messages.
- **`GlobalInstanceId`** is used in MSCE traces to have a unique identification of the process instance.

A **`xPrsRec`** struct contains the following components described below. As each `yVDef_ProcessName` struct contains the `PROCESS_VARS` macro as first item, it is possible to cast pointer values between a pointer to `xPrsRec` and a pointer to a `yVDef_ProcessName` struct.

- **`Pre`** and **`Suc`** of type `xPrsNode`. These components are used to link the process instance in the ready queue (see below).
- **`RestartAddress`** of type `int`. This component is used to find the appropriate SDL symbol to continue execute from.
- **`ActivePrd`** of type `xPrdNode`. This is a pointer to the `xPrdRec` that represents the currently executing procedure called from this process instance. The pointer is 0 if no procedure is currently called.
- **`RestartPAD`**, which is a pointer to a `PAD` function. This component refers to the `PAD` function where to execute the sequence of SDL symbols. `RestartPAD` is used to handle inheritance between process types.
- **`CallAddress`** of type `int`. This component contains the symbol number of the procedure call currently executed by this process.
- **`ActiveSrv`** of type `xSrvNode`. This component contains a reference to the currently active service (or latest active service) in this process.

- **`SrvList`** of type `xSrvNode`. This component contains a reference to the first service contained in this process. The component `NextSrv` in the struct representing a service can be used to find next active service in the process.
- **`NextPrs`** of type `xPrsNode`. This component is used to link the process instance either in the active list or in the avail list for this process type. The start of these two lists are the components `ActivePrsList` and `AvailPrsList` in the `IdNode` representing the current process type.
- **`Self`** of type `SDL_PId`. This is the `PId` value of the current process instance.
- **`NameNode`** of type `xPrsIdNode`. This is a pointer to the `PrsIdNode` representing the current process or process instantiation.
- **`State`** of type `int`. This component contains the int value used to representing the current state of the process instance.
- **`Signal`** of type `xSignalNode`. This is a pointer to a signal instance. The referenced signal is the signal that will cause the next transition by the current process instance, or that caused the transition that is currently executed by the process instance.
- **`InputPort`** of type `xInputPortRec`. This is the queue head in the doubly linked list that represents the input port of the process instance. The signals are linked in this list using the `Pre` and `Suc` components in the `xSignalRec` struct.
- **`Parent`** of type `SDL_PId`. This is the `PId` value of the parent process (according to the rules of SDL). A static process instance has parent equal to `NULL`.
- **`Offspring`** of type `SDL_PId`. This is the `PId` value of the latest created process instance (according to the rules of SDL). A process instance that has not created any processes has offspring equal to `NULL`.
- **`BlockInstNumber`** of type `int`. If the process is part of a block instance set, this component indicates which of the blocks that the process belongs to.

- **`pREPLY_Waited_For`** of type `xSignalIdNode`. When a process is waiting in the implicit state for the `pREPLY` signal in a RPC call, this components is used to store the `IdNode` for the expected `pREPLY` signal.
- **`pREPLY_Signal`** of type `xSignalNode`. When a process receives a `pCALL` signal, i.e. accepts a RPC, it immediately creates the return signal, the `pREPLY` signal. This component is used to refer to this `pREPLY` signal until it is sent.
- **`Sender`** of type `SDL_PId`. This component represents the SDL concept Sender.
- **`Trace_Default`** of type `int`. This component contains the current value of the trace defined for the process instance.
- **`GRTrace`** of type `int`. This component contains the current value of the GR trace defined for the process instance.
- **`MSCETrace`** of type `int`. This component contains the current MSCE trace value for the process instance.
- **`InTransition`** of type `xbool`. This component is true while the process is executing a transition and it is false while the process is waiting in a state. The monitor system needs this information to be able to print out relevant information.

The Ready Queue, Scheduling

The ready queue is a doubly linked list with a head. It contains the process instances that can execute an immediate transition, but which has not been allowed to complete that transition. Process instances are inserted into the ready queue during output operations and nextstate operations and are removed from the ready queue when they execute the nextstate or stop operation that ends the current transition. The head in the ready queue, which is an object in the queue that does not represent any process but is inserted only to simplify the queue operations, is referenced by the `xSysD` component:

```
xPrsNode   xReadyQueue;
```

This component is initiated in the function `xInitKernel` and used throughout the runtime library to reference the ready queue.

Scheduling of events is performed by the function `xMainLoop`, which is called from the `main` function after the initialization is performed.

```
void xMainLoop()
```

The strategy to have all interesting queues (the ready queue, the timer queue, and the input ports) sorted in the correct order is used in the library. Sorting is thus performed when an object is inserted into a queue, which means that scheduling is a simple task: select the first object in the timer queue or in the ready queue and submit it for execution.

There are several versions of the body of the endless loop in the function `xMainLoop`, which are used for different combinations of compilation switches. When it comes to scheduling of transitions and timer outputs they all have the following outline:

```
while (1) {
  if ( xTimerQueue->Suc->TimerTime <= SDL_Now() )
    SDL_OutputTimerSignal( xTimerQueue->Suc );
  else if ( xReadyQueue->Suc != xReadyQueue) {
    xRemoveFromInputPort(xReadyQueue->Suc->Signal);
    xReadyQueue->Suc->Sender =
       xReadyQueue->Suc->Signal->Sender;
    (*xReadyQueue->Suc->RestartPAD)(xReadyQueue->Suc);
  }
}
```

or, in descriptive terms:

```
while (1) {
  if ( there is a timer that has expired )
    send the corresponding timer signal;
  else if ( there is a process that can execute
              a transition ) {
    remove the signal causing the transition
      from input port;
    set up Sender in the process to Sender of
      the signal;
    execute the PAD function for the process;
  }
}
```

The different versions of the main loop handle different combinations of compilation switches. Other actions necessary in the main loop are dependent of the compilation switches. Example of such actions are:

- Handling of the monitor
- Calling the `xInEnv` function

- Handling real time or simulated time
- Delay execution up to the next scheduled event
- Handling enabling conditions and continuous signals that need to be recalculated.

Create and Stop Operations

A process instance is, while it is active, represented by the two structs:

- `xLocalPIdRec`
- The `yVDef_ProcessName` struct.

These two structs are dynamically allocated. A `PId` value is also a struct (not allocated) containing two components, `GlobalNodeNr` and `LocalPId`, where `LocalPId` is a pointer to the `xLocalPIdRec`. Figure 545 shows how the `xLocalPIdRec` and the `yVDef_ProcessName` structs representing a process instance are connected.

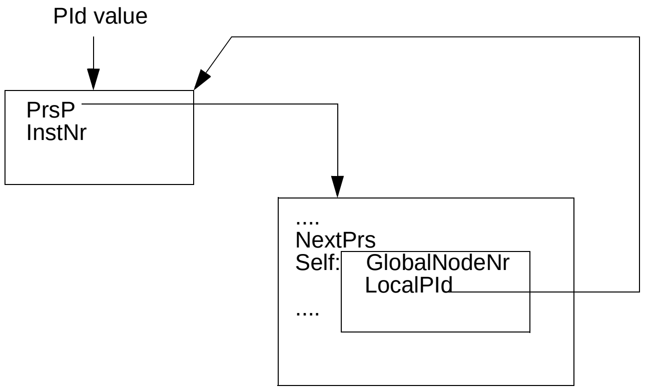

Figure 545: A xLocalPIdRec and a yVDef_ProcessName representing a Process instance

When a process instance performs a stop action, the memory used for the process instance should be reclaimed and it should be possible to re-use in subsequent create actions. After the stop action, old (invalid) `PId` values might however be stored in variables in other process instances.

If a signal is sent to such an old `PId` value, that is, to a stopped process instance, it should be possible to find and perform appropriate actions.

If the complete representation of a process instance is reused then this will not be possible. There must therefore remain some little piece of information and thus some memory for each process instance that has ever existed. This is the purpose of the `xLocalPIdRec`. These structs will never be reused. Instead the following (see Figure 546) will happen when the process instance in Figure 545 performs a stop action.

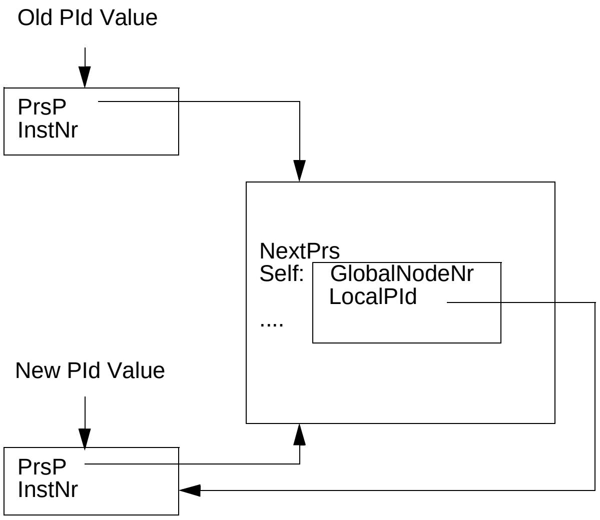

Figure 546: The memory structure after the process in Figure 545 has performed a stop action

A new `xLocalPIdRec` is allocated and its `PrsP` references the `yVDef_ProcessName` (`InstNr` is 0). The `Self` component in the `yVDef_ProcessName` is changed to reference this new `xLocalPIdRec`. The old `xLocalPIdRec` still references the `yVDef_ProcessName`. The `yVDef_ProcessName` is entered into the avail list for this process type.

To reuse the data area for a process instance at a create operation it is only necessary to remove the `yVDef_ProcessName` from the avail list and update the `InstNr` component in the `xLocalPIdRec` referenced by `Self`.

Using this somewhat complicated structure to represent process instances allows a simple test to see if a `PId` value refers to an active or a stopped instance:

If P is a PId variable then the following expression:

```
P.LocalPId == P.LocalPId->PrsP->Self.LocalPId
```

is true if the process instance is active and false if it is stopped.

The basic behavior of the create and stop operations is performed by the functions SDL_Create and SDL_Stop.

```
void SDL_Create(
  xSignalNode  StartUpSig,
  xPrsIdNode   PrsId )

void SDL_Stop( xPrsNode  PrsP )
```

To create a process instance takes three steps performed in generated code:

1. Call xGetSignal to obtain the start-up signal.
2. Assign the actual process parameters to the start up signal parameters.
3. Call SDL_Create with the start-up signal as parameter, together with the PrsIdNode representing the process to be created.

In xGetProcess the process instance is removed from the avail list of the process instance set (the component AvailPrsList in the PrsIdNode representing the process instance set), or if the avail list is empty new memory is allocated.

The process instance is linked into the list of active process instances (the component ActivePrsList in the PrsIdNode representing the process instance set). Both the avail list and the active list are single linked lists (without a head) using the component NextPrs in the yVDef_ProcessName struct as link.

To have an equal treatment of the initial transition and other transitions, the start state is implemented as an ordinary state with the name “start state” It is represented by 0. To execute the initial transition a “startup” signal is sent to the process. The start state can thus be seen as a state with one input of the startup signal and with save for all other signals. This implementation is completely transparent in the monitor, where startup signals are never shown in any way.

Note:

The actual values for FPARs are passed in the startup signal.

Two `IdNodes` that are not part of the symbol table tree are created to represent a start state and a startup signal.

```
xStateIdNode   xStartStateId;
xSignalIdNode  xStartUpSignalId;
```

These `xSysD` components are initialized in the function `xInitSymbolTable`, which is part of `sctsdl.c`.

At a stop operation the function `SDL_Stop` is called. This function will release the signal that caused the current transition and all other signals in the input port. It will also remove all timers in the timer queue that are connected to this process instance by calling `xRemoveTimer` with the first parameter equal to `0`. It then removes the process executing the stop operation from the ready queue and from the active list of the process type and returns the memory to the avail list of the current process instance set.

Output and Input of Signals

There are three actions performed in generated code to send a signal. First `xGetSignal` is called to obtain a data area that represents the signal instance, then the signal parameters are assigned their values and finally the function `SDL_Output` is called to actually send the signal. First in the `SDL_Output` function there are a number of dynamic tests (check if receiver in TO-clause is not `NULL` and not stopped, check if there is a path to the receiver). If the output does not contain any TO-clause and the Cadvanced/Cbasic SDL to C Compiler has not been able to calculate the receiver, the `xFindReceiver` function is called to calculate the receiver according to the rules of SDL.

Next, in `SDL_Output` signals to the environment are handled. Three cases can be identified here:

1. The environment function `xOutEnv` is called.
2. The corresponding function that sends signals via the SDL suite communication mechanism (`xOutPM`) is called.
3. The signal is inserted into the input port of the process representing the environment (`xEnv`).

Finally, internal signals in the SDL system are treated. Here also three cases can be identified (how this is evaluated is described last in this subsection):

1. The signal can cause an immediate transition by the receiver.
2. The signal should be saved.
3. The signal should be immediately discarded.

If the signal can cause an immediate transition, the signal is inserted into the input port of the receiver, and the receiving process instance is inserted into the ready queue.

If the signal should be saved, the signal is just inserted into the input port of the receiver.

If the signal should be discarded, the function `xReleaseSignal` is called to reused the data area for the signal.

When a signal is identified to be the signal that should cause the next transition by the current process instance (at an Output or Nextstate operation), the component `Signal` in the `yVDef_ProcessName` for the process is set to refer to the signal. The signal is still part of the input port list.

When the transition is to be executed, the signal is removed from the input port in the main loop (see "The Ready Queue, Scheduling" on page 3005) immediately before the `PAD` function for the process is called.

First in the `PAD` function, the parameters of the signal are copied to the local variables according to the input statement. In the ending Nextstate or Stop operation of the transition the signal instance is returned to the avail list.

Evaluating How To Handle a Received Signal

There are two places in the run-time kernel where it is necessary to evaluate how to handle signals (input, save, discard,...):

- At an Output operation to a currently idle process.
- At a Nextstate operation, when the process have signals in the input port.

This calculation is implemented in the run-time kernel function `xFindInputAction`.

```
typedef unsigned char xInputAction;
#define xDiscard        (xInputAction)0
#define xInput          (xInputAction)1
#define xSave           (xInputAction)2
#define xEnablCond      (xInputAction)3
#define xPrioInput      (xInputAction)4

static xInputAction xFindInputAction(
  xSignalNode  SignalId,
  xPrsNode     VarP,
  xbool        CheckPrioInput )
```

The parameters of this function is:

- `SignalId`, which is a pointer to a signal.
- `VarP`, which is a pointer to a process instance.
- `CheckPrioInput`, which is a boolean value indicating is the function should check only for priority inputs or for ordinary inputs.

As a result the function should return:

- The action that should be performed for this signal (input, save,...), taking all information about this process into account, like inheritance between processes, virtual - redefined transitions and so on.
- If the function result is `xInput` or `xPrioInput`, then the `RestartPAD` and `RestartAddr` components in the `VarP` struct should be updated with information about where this input can be found.

After this last update the correct transition can be started by the scheduler by just calling the function referenced by `RestartPAD`, which the as first action performs switch `RestartAddr` and starts execute the input symbol.

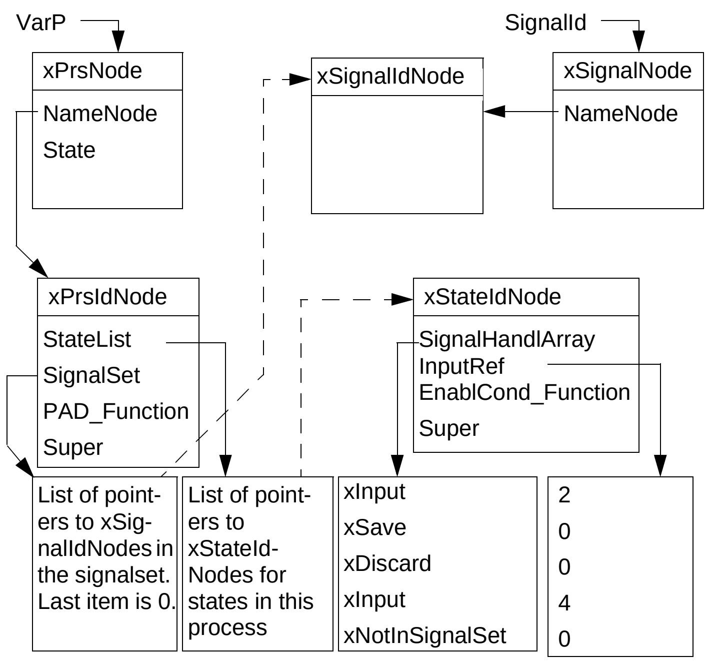

Figure 547: Data structure used to evaluate the xFindInputAction

The algorithm to find the `InputAction`, the `RestartAddr`, and the `RestartPAD` is as follows:

1. Let `ProcessId` become `yVarP->NameNode` and let `StateId` become `ProcessId->StateList[yVarP->State]` .
2. In `ProcessId->SignalSet` find the index (`Index`) where `SignalId->NameNode` is found. If the signal is not found, this signal is not in the signal set of the process, and the algorithm terminates returning the result `xDiscard`.
3. `StateId->SignalHandlArray[Index]` now gives the action to be performed. If this value is `xEnablCond`, then the function `StateId->EnablCond_Function` is called. This function returns either `xInput` or `xSave`.

4. If the result from step 3 is `xInput`, the algorithm terminates returning this value. `yVarP->RestartAddr` is also updated to `StateId->InputRef[Index]`, while `yVarP->RestartPAD` is updated to `ProcessId->PAD_Function`.

   If the result from step 3 is `xSave`, the algorithm terminates returning this value.

   If the result from step 3 is `xDiscard` and `ProcessId->Super` equal to `NULL`, then the algorithm terminates returning this value.

   If the result from step 3 is `xDiscard` and `ProcessId->Super` not equal to `NULL`, then we are in a process type that inherits from another process type. We then have to perform step 2 - 4 again, with `ProcessId` assigned the value `ProcessId->Super` and `StateId` assigned the value `StateId->Super`.

Nextstate Operations

The nextstate operation is implemented by the `SDL_Nextstate` function, where the following actions are performed:

1. The signal that caused the current transition (component `Signal` in the `yVDef_ProcessName`) is released and the state variable (component `State` in the `yVDef_ProcessName`) is updated to the new state.
2. Then the input port of the process is scanned for a signal that can cause a transition. During the scan signals might be saved or discarded until a signal specified in an input is found. Priority inputs are treated according to the rules of SDL.
3. If no signal that can cause a transition is found, a check is made if any continuous signal can cause a transition (see “Enabling Conditions and Continuous Signals” on page 3015). The process is thereafter removed from the ready queue.
4. If any signal (or continuous signal) can cause a transition then the process is re-inserted into the ready queue again at a position determined by its priority, else if the new state contains any continuous signal or enabling condition with an expression that might change its value during the time the process is in the state (view, import...), the process is inserted into the check list (see also “Enabling Conditions and Continuous Signals” on page 3015).

Decision and Task Operations

Decision and Task operations are implemented in generated code, except for the Trace-functions implemented in the `sctutil.c` and `sctmon.c` files and for informal and any decisions that uses some support functions in `sctmon.c`. A Decision is implemented as a C if-statement, while the assignments in a Task are implemented as assignments or function calls in C.

Compound Statements

A compound statement without variable declarations is translated just to the sequence of action it contains, while a compound statement with variable declarations is translated in the same way as an SDL procedure (without parameters). Statements within a compound statement are translated according to the normal rules. The new statement types in compound statements are translated as:

- **if** in SDL is translated to **if** in C
- **decision** in compound statements is translated as ordinary decisions.
- **for** loops, **continue**, and **break** are all translated using **goto** in C.

Enabling Conditions and Continuous Signals

The expressions involved in continuous signals and enabling conditions are implemented in generated code in functions called `yCont_`*`StateName`* and `yEnab_`*`StateName`*. These functions are generated for each state containing continuous signals respectively enabling conditions. The functions are referenced through the components `ContSig_Function` and `EnablCond_Function` in the `StateIdNode` for the state. These components are `0` if no corresponding functions are generated.

The `EnablCond_Functions` are called from the function `xFindInputAction`, which is called from `SDL_Output` and `SDL_Nextstate`. If the enabling condition expression for the current signal is true then `xInput` is returned else `xSave` is returned. This information is then used to determine how to handle the signal in this state.

The `ContSig_Functions` are called from `SDL_Nextstate`, if the component `ContSig_Function` is not `0` and no signal that can cause an immediate transition is found during the input port scan. A `ContSig_Function` has the following prototype:

```
void ContSig_Function_Name (
  void *, int *, xIdNode *, int *);
```

where the first parameter is the pointer to the yVDef_ProcessName. The remaining parameters are all out parameters; the second contains the priority of the continuous signal with highest priority (=lowest value) that has an expression with the value true. Otherwise <0 is returned here. The third and fourth is only defined the second parameter >=0; the third is the IdNode for the process/procedure where the actual continuous signal can be found and the fourth is the RestartAddress connected to this continuous signal.

If a continuous signal expression with value true is found, a signal instance representing the continuous signal is created and inserted in the input port, and is thereafter treated as an ordinary signal. The signal type is continuous signal and is represented by an SignalIdNode (referenced by the variable xContSigId).

The check list is a list that contains the processes that wait in a state where enabling conditions or continuous signals need to be repeatedly recalculated.

A process is inserted into the check list if:

1. It enters a state containing enabling conditions and/or continuous signals and
2. No signal or continuous signal can cause an immediate transition and
3. One or several of the expressions in the enabling conditions or continuous signals can change its value while the process is in the state (view, import, now, ...)

The component StateProperties in the StateIdNode reflects if any such expression is present in the state.

The check list is represented by the xSysD component:

```
xPrsNode  xCheckList;
```

The behavior of enabling conditions and continuous signals is in SDL modeled by letting the process repeatedly send signals to itself, thereby to repeatedly entering the current state. In the implementation chosen here, nextstate operations are performed “behind the scene” for all processes in the check list directly after a call to a PAD function is completed, that is directly after a transition is ended and directly after a timer output. This is performed by calling the function xCheckCheckList in the main loop of the program.

View and Reveal

A view expression is part of an expression in generated code and implemented by calling the function SDL_View.

```
void * SDL_View (
  xViewListRec *VList,
  SDL_PId       P,
  xbool         IsDefP,
  xPrsNode      ViewingPrs,
  char *        Reveal_Var
  int           SortSize);
```

- **VList** is a list of all revealed variables in this block.
- **P** is the PId expression given in the view statement.
- **IsDefP** is 1 is the view expression contained a PId value, 0 otherwise.
- **ViewingPrs** is the process instance performing the view operation.
- **Reveal_Var** is the name of the revealed variable as a string. The Reveal_Var parameter is only used in error messages and is remove under certain conditions.
- **SortSize** is the size of the data type of the viewed variable.

The SDL_View function performs a test that the view expression is not NULL, refers to a process in the environment, or to a stopped process instance. If no errors are found the address of the revealed variable is returned as result from the SDL_View function. Otherwise the address of a variable containing only zeros is returned.

Import, Export, and Remote Variables

For an exported variable there are two components in the yVDef_ProcessName struct. One for the current value of the variable and one for the currently exported value of the variable. For each exported variable there will also be a struct that can be linked into a list in the corresponding `RemoteVarIdNode`. This list is then used to find a suitable exporter of a variable in an import action.

An export action is a simple operation. The current value of the variable is copied to the component representing the exported value. This is performed in generated code.

An import action is more complicated. It involves mainly a call of the function `xGetExportAddr`:

```
void * xGetExportAddr (
  xRemoteVarIdNode RemoteVarNode,
  SDL_PId          P,
  xbool            IsDefP,
  xPrsNode         Importer )
```

`RemoteVarNode` is a reference to the `RemoteVarIdNode` representing the remote variable (implicit or explicit), `P` is the PId expression given in the import action and `IsDef` is 0 or 1 depending on if any `PId` expression was given in the import action or not, Importer is the importing process instance. The `xGetExportAddr` will check the legality of the import action and will, if no `PId` expression is given, calculate which process it should be imported from.

If no errors are found the function will return the address where the exported value can be found. This address is then casted to the correct type (in generated code) and the value is obtained. If no process possible to import from is found, the address of a variable containing only zeros is returned by the `xGetExportAddr` function.

Note:

The strategy for import actions is in one sense not equal to the model for import given in the SDL recommendation. An import action is in the recommendation modeled as a signal sent from the importing process to the exporting process asking for the exported value, and a signal with this value sent back again. The synchronization effects by this signal communication is lost in the implementation model we have chosen. Instead our model is much easier and faster and the primary part of the import action, to obtain the exported value, is the same.

Services

Data Structure Representing Services

A service is represented by a struct type. The `xSrvRec` struct defined in `scttypes.h`, is, just like `xPrsRec` for processes, a struct containing general information about a service, while the parameters and variables of the service are defined in generated code in the same way as for processes.

In `scttypes.h` the following types concerning procedures can be found:

```
#ifdef XMONITOR
#define XCALL_ADDR   int  CallAddress;
#else
#define XCALL_ADDR
#endif

#define SERVICE_VARS \
   xSrvNode       NextSrv; \
   xPrsNode       ContainerPrs; \
   int            RestartAddress; \
   xPrdNode       ActivePrd; \
   void (*RestartPAD) (xPrsNode  VarP); \
   XCALL_ADDR \
   xSrvIdNode     NameNode; \
   int            State; \
   XSIGTYPE       pREPLY_Waited_For; \
   xSignalNode    pREPLY_Signal; \
   XINTRANSCOMP
#ifndef XNOUSEOFSERVICE
typedef struct xSrvStruct *xSrvNode;
#endif

#ifndef XNOUSEOFSERVICE
typedef struct xSrvStruct {
   SERVICE_VARS
}  xSrvRec;
#endif
```

In generated code yVDef_ProcedureName structs are defined according to the following:

```
typedef struct {
   SERVICE_VARS
   components for FPAR and DCL
}  yVDef_ServiceName;
```

The components in the `xSrvRec` are used as follows:

- **`NextSrv`** of type `xSrvNode`. Reference to next service contained in this process.
- **`ContainerPrs`** of type `xPrsNode`. Reference to the process instance containing this service.
- **`RestartAddress`** of type `int`. This component is used to find the appropriate SDL symbol to continue execution from.
- **`ActivePrd`** of type `xPrdNode`. This is a pointer to the xPrdRec that represents the currently executing procedure called from this service instance. The pointer is 0 if no procedure is currently called.
- **`RestartPAD`**, which is a pointer to a PAD function. This component refers to the PAD function where to execute the sequence of SDL symbols. RestartPAD is used to handle inheritance between service types.
- **`CallAddress`** of type `int`. This component contains the symbol number of the procedure call performed from this procedure (if any).
- **`NameNode`** of type `xSrvIdNode`. This is a pointer to the IdNode representing the service or service instantiation.
- **`State`** of type `int`. This component contains the int value used to represent the current state of the service instance.
- **`pREPLY_Waited_For`** of type `xSignalIdNode`. When a service is waiting in the implicit state for the pREPLY signal in a RPC call, this components is used to store the IdNode for the expected pREPLY signal.

- **`pREPLY_Signal`** of type `xSignalNode`. When a service receives a pCALL signal, i.e. accepts a RPC, it immediately creates the return signal, the pREPLY signal. This component is used to refer to this pREPLY signal until it is sent.
- **`InTransition`** of type `xbool`. This component is true while the service is executing a transition and it is false while the service is waiting in a state. The monitor system needs this information to be able to print out relevant information.

Executing Transitions in Services

From the scheduler's point view, it is not of interest if a process contains services or not. It is still the process instance that is scheduled in the ready queue and the PAD function of the process that is to be called to execute a transition. The PAD function for a process containing services performs three different actions:

- Assign default value to variables declared at the process level
- Create one service instance for each service or service instantiation in the process.
- Calls the proper PAD function for a service to execute transitions.

The structure for a PAD function for a process with services are as follows:

```
YPAD_FUNCTION(yPAD_z00_P1)
{
  YPAD_YSVARP
  YPAD_YVARP(yVDef_z00_P1)
  YPRSNAME_VAR("P1")
  LOOP_LABEL_SERVICEDECOMP
  CALL_SERVICE

/*-----
* Initialization (no START symbol)
------*/
  BEGIN_START_TRANSITION(yPDef_z00_P1)
  yAssF_SDL_Integer(yVarP->z002_Global,
     SDL_INTEGER_LIT(10), XASS);
  START_SERVICES
}
```

where `LOOP_LABEL_SERVICEDECOMP` and `BEGIN_START_TARNSITION` are empty macros, i.e. expanded to no code. The `yAss_SDL_Integer` statement in an assignment of a default value to a process variable.

The macro CALL_SERVICE is expanded to:

```
if (yVarP->ActiveSrv != (xSrvNode)0) {
  (*yVarP->ActiveSrv->RestartPAD)(VarP);
  return; \
}
```

that is to a call of the PAD function of service reference by ActiveSrv.

The macro START_SERVICE is expanded to a call to the function `xStart_Services`, which can be found in `sctsdl.c`.The function creates the service instances, sets up the ActiveSrv pointer for the process to the first service, and then schedules the process for a new transition. This means that the next action performed by the system will be the start transition by the first service instance. When the first service executes a nextstate or stop action in the end of its start transition, the process will be scheduled again to execute the start transition of the second service, and so on until all services in the process has executed its start transitions.

For ordinary transitions, i.e. reception of a signal, it is obvious from the code above that the ActiveSrv pointer is essential. It should refer to the service instance that is to be executed. When a signal is to be received by a process, it is the function xFindInputAction (in `sctsdl.c`) that determines how to handle the signal and if it is to be received, where is the code for that transition. This function now also determines and sets up the ActiveSrv pointer.

Procedures

Data Structure Representing Procedures

A procedure is represented by a struct type. The `xPrdRec` struct defined in `scttypes.h`, is, just like `xPrsRec` for processes, a struct containing general information about a procedure, while the parameters and variables of the procedure are defined in generated code in the same way as for processes.

In `scttypes.h` the following types concerning procedures can be found:

```
#define PROCEDURE_VARS \
   xPrdIdNode   NameNode; \
   xPrdNode     StaticFather; \
   xPrdNode     DynamicFather; \
   int          RestartAddress; \
   XCALL_ADDR \
   void (*RestartPAD) (xPrsNode  VarP); \
   xSignalNode  pREPLY_Signal; \
   int          State;

typedef struct xPrdStruct  *xPrdNode;

typedef struct xPrdStruct {
   PROCEDURE_VARS
}  xPrdRec;
```

In generated code yVDef_ProcedureName structs are defined according to the following:

```
typedef struct {
   PROCEDURE_VARS
   components for FPAR and DCL
}  yVDef_ProcedureName;
```

The components in the `xPrdRec` are used as follows:

- **`NameNode`** of type `xPrdIdNode`. This is a pointer to the `IdNode` representing the procedure type.
- **`StaticFather`** of type `xPrdNode`. This is a pointer that represents the scope hierarchy of procedures (and the process at the top), which is used when a procedure instance refers to non-local variables. An example is shown in Figure 548 on page 3025. `StaticFather == 0` means that the static father is the process.
- **`DynamicFather`** of type `xPrdNode`. This is a pointer that represents that this procedure is called by the referenced procedure. `DynamicFather == 0` means that this procedure was called from the process. This component is also used to link the `xPrdRec` in the avail list for the procedure type.
- **`RestartAddress`** of type `int`. This component is used to find the appropriate SDL symbol to continue execution from.
- **`CallAddress`** of type `int`. This component contains the symbol number of the procedure call performed from this procedure (if any).
- **`RestartPRD`** is a pointer to a procedure function. This component refers to the `PRD` function where to execute the next sequence of SDL symbols. `RestartPRD` is used to handle inheritance between procedures.
- **`pREPLY_Signal`** of type `xSignalNode`. When a process receives a pCALL signal, i.e. accepts a RPC, it immediately creates the return signal, the `pREPLY` signal. This component is used to refer to this `pREPLY` signal until it is sent.
- **`State`** of type `int`. This is the value representing the current state of the procedure instance.

In Figure 548 on page 3025 an example of the structure of `yVDef_ProcedureName` after four nested procedure calls are presented. Note that procedure Q is declared in the process, procedure R and S in Q and T in S.

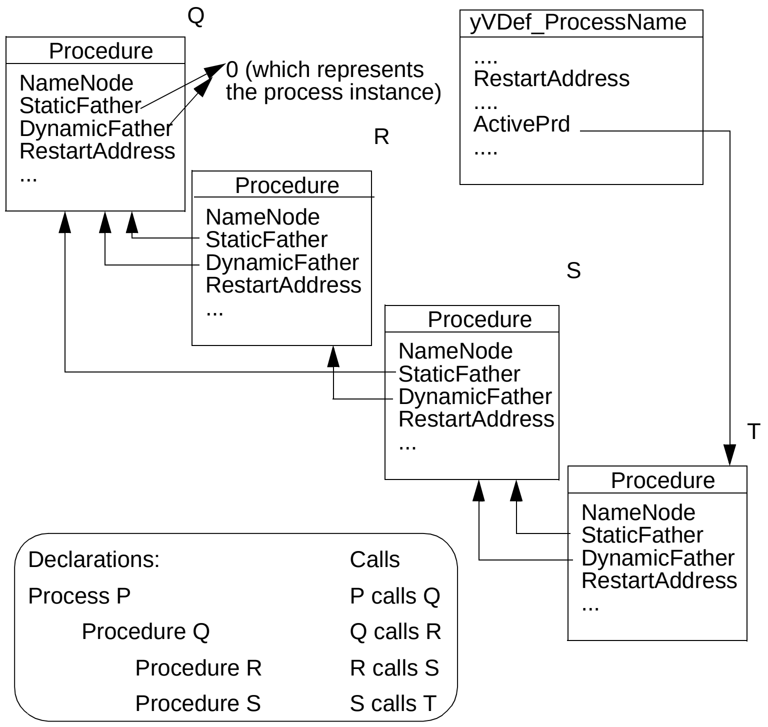

Figure 548: Structure of `yVDef_ProcedureName` *after four nested procedure calls*

The SDL procedures are partly implemented using C functions and partly using the structure shown above. Each SDL procedure is represented by a C function, which is called to execute actions defined in the procedure. This function corresponds to the `PAD` function for processes. The formal parameters and the variables are however implemented using a struct defined in generated code. The procedure stack for nested procedure calls is implemented using the components `StaticFather` and `DynamicFather`, and does not use the C function stack.

Calling and Returning from Procedures

Procedure calls and procedure returns are handled by three functions, one handling allocation of the data areas for procedures:

```
xPrdNode  xGetPrd( xPrdIdNode  PrdId )
```

and two functions called from generated code at a procedure call and a procedure return:

```
void xAddPrdCall(
  xPrdNode  R,
  xPrsNode  VarP,
  int       StaticFatherLevel,
  int       RestartAddress )

void xReleasePrd (xPrsNode  VarP)
```

A procedure call in SDL is in C represented by the following steps:

1. Calling `xGetPrd` to obtain a data area for the procedure.
2. Assigning procedure parameters to the data area.
3. Calling `xAddPrdCall` to link the procedure into the static and dynamic chains.
4. Calling the C function modeling the SDL procedure, i.e. the `yProcedureName` function.

The parameters to `xAddPrdCall` are as follows:

- **`R`**. A reference to the `xPrdNode` obtained from the call of `xGetPrd`.
- **`VarP`**. A reference to the yVDef_ProcessName, i.e. the data area for variables and parameters of the process (even if it is a procedure that performed the procedure call).
- **`StaticFatherLevel`**. This is the difference in declaration levels between the caller and the called procedure. This information is used to set up the `StaticFather` component correctly.
- **`RestartAddress`**. This is the symbol number of the SDL symbol directly after the procedure call. The symbol number is the switch case label generated for all symbols.

The `xGetPrd` returns a pointer to an `xPrdRec`, which can then be used to assign the parameter values directly to the components in the data area representing the formal parameters and variables of the procedure. Note that IN/OUT parameters are represented as addresses in this struct.

A procedure return is in generated code represented by calling the `xReleasePrd` followed by return 0, whereby the function representing the behavior of the SDL procedure is left.

The function representing the behavior of the SDL procedure is returned in two main situations:

- When an SDL `Return` is reached (the function returns 0)
- When a `Nextstate` is reached (the function returns 1).

If 0 is returned then the execution should continue with the next SDL symbol after the procedure call, while if 1 is returned the execution of the process instance should be terminated and the scheduler (main loop) should take control. This could mean that a number of nested SDL procedure calls should be terminated.

To continue to execute at the correct symbol when a procedure should be resumed after a nextstate operation, the following code is introduced in the `PAD` function for processes containing procedure calls:

```
while ( yVarP->ActivePrd != (xPrdNode)0 )
  if ((*yVarP->ActivePrd->RestartPRD)(VarP))
    return;
```

This means that uncompleted procedures are resumed one after one from the bottom of the procedure stack, until all procedures are completed or until one of them returns 1, i.e. executes a nextstate operation, at which the process is left for the scheduler again.

Channels and Signal Routes

The `ChannelIdNodes` for channels, signal routes, and gates are used in the functions `xFindReceiver` and `xIsPath`, which are both called from `SDL_Output`, to find the receiving process when there is no TO clause in the Output statement, respectively to check that there is a path to the receiver in the case of a TO clause in the Output statement. In both cases the paths built up using the `ToId` components in the `IdNodes` for processes, channels and signal routes are followed. To show the structure of these paths we use the small SDL system given in Figure 549.

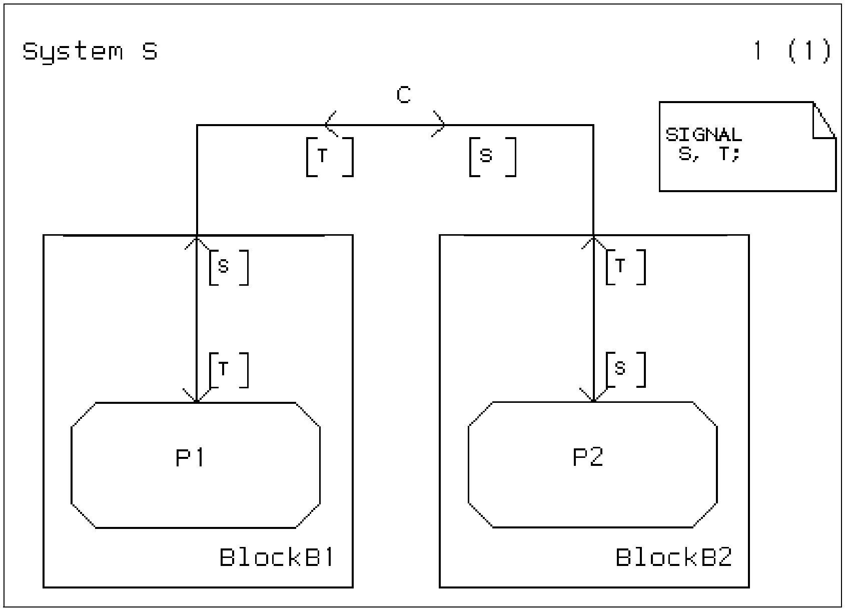

Figure 549: A small SDL system

During the initialization of the system, the symbol table is built up. The part of the symbol table starting with the system will then have the structure outlined in Figure 550. As we can see in this example the declarations in the SDL system are directly reflected by `IdNodes`.

Note:

Each channel and signal route is represented by two `IdNodes`, one for each direction. This is also true for an unidirectional channel or signal route. In this case the signal set will be empty for the unused direction.

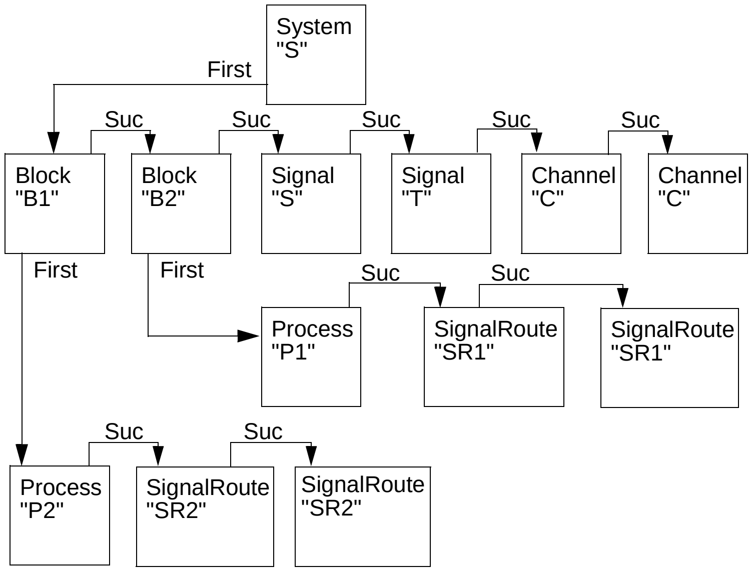

Figure 550: The symbol table tree for the system in Figure 549

Each IdNode representing a process, a signal route, or a channel will have a component `ToId`. A `ToId` component is an address to an array of references to `IdNodes`. The size of this array is dependent on the number of items this object is connected to. A process that has three outgoing signal routes will have a `ToId` array which can represent three pointers plus an ending `0` pointer.

In the example in Figure 549 and Figure 550 there is no branching, so all `ToId` arrays will be of the size necessary for two pointers. Figure 551 shows how the `IdNodes` for the processes, signal routes and channels are connected to form paths, using the components `ToId`. In this case only simple paths are found (one from P1, via SR1, C, SR2, to P2, and one in the reverse direction). The generalization of this structure to handle branches is straightforward and discussed in the previous paragraph.

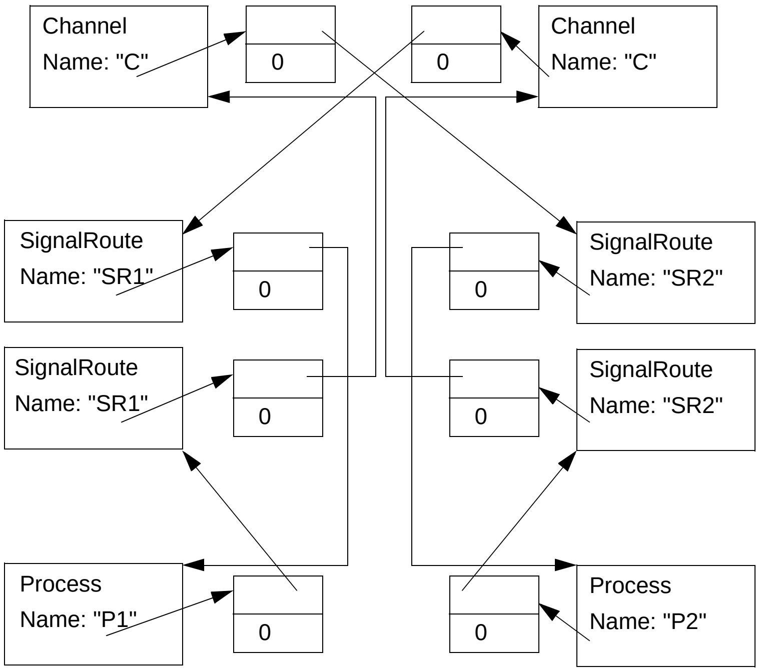

Figure 551: The connection of `ToId` *for the system in Figure 549 and Figure 550*

The Type Concept in SDL-92

The probably most important new feature in SDL-92 is the introduction of the object oriented features, such as TYPE, INHERITS, VIRTUAL, and REDEFINED. Here we start by discussing process types.

For each process type the Cadvanced/Cbasic SDL to C Compiler will generate:

- a `PrsIdNode`
- a `PAD` function
- a `yVDef_ProcessName` struct.

In the `PrsIdNode` there is one component (Super) that will refer to the `PrsIdNode` for the process type inherited by this process type. As sons to a `PrsIdNode`, `IdNodes` for declaration that are common for all instantiation of the process type can be found. Examples of such `IdNodes` are: nodes for variables, formal parameters, signals, timers, procedures, states, newtypes, and syntypes. Any typedefs or help functions for such units are also treated in the process type.

The `PAD` function will be independent of the `PAD` function for a inherited type, each `PAD` function just implementing the action described in its process type.

A `yVDef_ProcessName` struct will on the other hand include all variables and formal parameters from the top of the inheritance chain and downwards. Example:

```
process type P1;
fpar f1 integer;
dcl  d1 integer;
...
endprocess;

process type P2 inherits P1;
fpar f2 integer;
dcl  d2 integer;
...
endprocess;
```

This will generate the following principle yVDef_... structs:

```
typedef struct {
  PROCESS_VARS
  SDL_Integer f1;
  SDL_Integer d1;
} yVDef_P1;
```

```
typedef struct {
  PROCESS_VARS
  SDL_Integer f1;
  SDL_Integer d1;
  SDL_Integer f2;
  SDL_Integer d2;
} yVDef_P2;
```

A pointer to `yVDef_P2` can thus be casted to a pointer to `yVDef_P1`, if only the common component (in `PROCESS_VARS`) or the variables in P1 is to be accessed. This possibility is used every time the `PAD` function for an inherited process type is called.

Each process instantiation will all be implemented as a xPrsIdNode. The `Super` component in such an object refers to the process type that is instantiated. No `PAD` function or yVDef_... struct will be generated. As sons to the `PrsIdNode` for a process instantiation, only such object are inserted that are different in different instantiations. For a process instantiation this is the gates. For other types of information the process instantiation uses the information given for its process type.

A very similar structure when it comes to `IdNodes` generated for block types and block instantiations are used by the code generator. There will be a `BlockIdNode` for both a block type and for a block instantiation. As sons to a block type, nodes that are the same in each block instantiation can be found (example: signal, newtype, syntype, block type, process type, procedure). As sons to a block instantiation, nodes that are needs to be represented in each block instantiation can be found (example: block instantiation, process instantiation, channel, signal route, gate, remote definitions).

Note:

A block or process (according to SDL-88), that is contained in a block type or a system type, is translated as if it was a type and instantiation at the same place.

A way to look at the structure of `IdNodes` in a particular system is to use the command Symboltable in the monitor system. This command prints the `IdNode` structure as an indented list of objects.

Allocating Dynamic Memory

Introduction

This section deals with the allocation and deallocation of dynamic memory.

> **Note:**
> This information is only valid when the Master Library is used. The OS integrations might have different strategies for memory allocation.

Information is provided about the following topics:

- Explanation about how dynamic memory is allocated and reused (deallocation and avail lists)
- How to estimate the total need of dynamic memory for an application.

Dynamic memory is used for a number of objects in a run-time model for applications generated by the Cadvanced/Cbasic SDL to C Compiler. These objects are:

- Process instances
- Signal and timer instances
- Procedure instances
- Charstring, Octet_string, Bit_string, and Object_identifer variables and variables of String, Bag, general Array, and general Powerset types.
- Variables of other user-defined data types, where the user has decided to use dynamic memory.

To help to estimate the need for memory for an application we will give information about the size of these objects and about how many of the objects are created. The size information given is true for generated applications, that is, ones that do not, for example, contain the monitor. The type definitions given are stripped of components that will not be part of an application. The full definitions may be found in the file `scttypes.h`.

Processes

Each process instance is represented by two structs that will be allocated on the heap. In `scttypes.h` the type `xLocalPIdRec` is defined and in generated code `yVDef_ProcessName` structs are defined:

```
typedef struct {
   xPrsNode      PrsP;
}  xLocalPIdRec;

typedef struct {
   xPrsNode       Pre;
   xPrsNode       Suc;
   int            RestartAddress;
   xPrdNode       ActivePrd;
   void (*RestartPAD) (xPrsNode  VarP);
#ifndef XNOUSEOFSERVICE
   xSrvNode       ActiveSrv;
   xSrvNode       SrvList;
#endif
   xPrsNode       NextPrs;
   SDL_PId        Self;
   xPrsIdNode     NameNode;
   int            State;
   xSignalNode    Signal;
   xInputPortRec  InputPort;
   SDL_PId        Parent;
   SDL_PId        Offspring;
   int            BlockInstNumber;
   xSignalIdNode  pREPLY_Waited_For;
   xSignalNode    pREPLY_Signal;

   /* variables and formal parameters in the
      process */
} yVDef_ProcessName;
```

To calculate the size of the structs above it is necessary to know more about the components in the structs. The types `xPrsNode`, `xPrdNode`, `xSignalNode`, `xPrsIdNode`, `xStateIdNode`, and `xSignalIdNode` are all pointers, while `SDL_PId` is a struct containing an int and a pointer. The `xInputPortRec` is a struct with two pointers and one int.

This means that it is possible to calculate the size of the `xLocalPIdRec` and the `xPrsRec` struct using the following formulas, if the compiler does not use any strange alignment rules:

$$\mathrm{Size}_{\mathtt{xLocalPIdRec}} = \mathrm{Size}_{\mathrm{address}}$$

$$\mathrm{Size}_{\mathtt{xPrsRec}} = 16 \cdot \mathrm{Size}_{\mathrm{address}} + 7 \cdot \mathrm{Size}_{\mathtt{int}}$$

The size of xPrsRec can be reduced by 2 x sizeof (address) if the code is compiled with the XNOUSEOFSERVICE flag. Then, of course, the SDL concept service cannot be used. The size of `yVDef_ProcessName` is the size of the `xPrsRec` plus the size of the variables and parameters in the process. Any overhead introduced by the C system should also be added. The size of the formal parameter and variables is of course dependent on the declarations in the process. The translation rules for SDL types, both predefined and user defined, can be found in *chapter 57, The Cadvanced/Cbasic SDL to C Compiler*.

For each process instance set in the system the following number of structs of a different kind will be allocated:

- There will be one `xLocalPIdRec` for each process instance created. These structs will not be reused, as they serve as identification of process instances that have existed (see also optimizations below).
- There will be as many `yVDef_ProcessName` structs as the maximum concurrently executing process instances of the process instance set (maximum during the complete execution of the program).

The `yVDef_ProcessName` structs are reused by having an avail list where this struct is placed when the process instance it represents perform a stop action. There is one avail list for each process type. When a process instance should be created, the runtime library first looks at the avail list and reuses an item from the list. Only if the avail list is empty new memory is allocated.

Compilation switch XPRSOPT

If the compilation switch `XPRSOPT` is defined then:

- `xLocalPIdRec`s are reused together with the `xPrsRec`s.
- `xLocalPIdRec`s contain an additional int component.

Services

Services are handled very similar to processes. The following struct type are allocated for each service instance.

```
typedef struct xSrvStruct {
   xSrvNode       NextSrv;
   xPrsNode       ContainerPrs;
   int            RestartAddress;
   xPrdNode       ActivePrd;
   void (*RestartPAD) (xPrsNode  VarP);
   xSrvIdNode     NameNode;
   int            State;
   XSIGTYPE       pREPLY_Waited_For;
   xSignalNode    pREPLY_Signal;
}  xSrvRec;
```

This means that:

$$\text{Size}_{\texttt{xSrvRec}} = 7 \cdot \text{Size}_{\text{address}} + 2 \cdot \text{Size}_{\texttt{int}}$$

The size of `yVDef_ServiceName` is the size of the xSrvRec plus the size of the variables in the service. `yVDef_ServiceName` struct are re-used in the same way as for processes (see previous section).

Signals

Signals are handled in much the same way as processes. A signal instance is represented by one struct (in generated code generated).

```
typedef struct {
   xSignalNode   Pre;
   xSignalNode   Suc;
   int           Prio;
   SDL_PId       Receiver;
   SDL_PId       Sender;
   xIdNode       NameNode;

   /* Signal parameters */
} yPDef_SignalName;
```

This struct type contains one component for each signal parameter. The component types will be the translated version of the SDL types of the parameters.

This means that it is possible can calculate the size of a `xSignalRec`, which is the same as a struct for a signal without parameters, using the following formula:

$$\text{Size}_{\texttt{xSignalRec}} = 5 \cdot \text{Size}_{\text{address}} + 3 \cdot \text{Size}_{\texttt{int}}$$

The size of a `yPDef_SignalName` struct is thus equal to the size of the `xSignalRec` plus the size of the parameters. The translation rules for SDL types, both the predefined and user defined, can be found in *chapter 57, The Cadvanced/Cbasic SDL to C Compiler*.

For each signal type in the system the following number of data areas will be allocated:

- There will be as many `yPDef_SignalName` struct as the maximum number of signals (during the complete execution of the program) of the signal type that are sent but not yet received in an input operation.

The `yPDef_SignalName` struct is reused by having an *avail* list, where the struct is placed when the signal instance they represent is received. The exact point where the signal instance is returned to the avail list is when the transition caused by the signal instance is ended by a nextstate or stop action. There is one avail list for each signal type. When a signal instance should be created, for example during an output operation, the runtime library first looks at the avail list and reuses an item from this list. Only if the avail list is empty new memory is allocated.

Note:

There is one common avail list for all signals without parameters.

Timers

The memory needed for timers can be calculated in the same way as for signals with one exception, each timer contains an extra `SDL_Time` component, i.e. two extra 32-bit integers.

Procedures

Procedures and processes have much in common in terms of memory allocation. A procedure is, during the time it exists from call to return, represented by a struct; the `yVDef_ProcedureName`.

```
typedef struct {
   xPrdIdNode  NameNode;
   xPrdNode    StaticFather;
   xPrdNode    DynamicFather;
   int         RestartAddress;
   int (*RestartPRD) (xPrsNode  VarP);
   xSignalNode pREPLY_Signal;
   int         State;

   /* Formal parameters and variables */
} yVDef_ProcedureName;
```

The struct type contains one component for each formal parameter or variable. The component types will be the translated version of the SDL types of the parameters, except for an IN/OUT parameter which is represented as an address.

The size of the `xPrdRec` struct (which is the same as a procedure without variables and formal parameters) can be calculated using the following formula:

$$\mathrm{Size}_{\mathtt{xPrdRec}} = 5 \cdot \mathrm{Size}_{\mathrm{address}} + 2 \cdot \mathrm{Size}_{\mathtt{int}}$$

The size of a `yVDef_ProcedureName` struct is the size of the `xPrdRec` plus the size of the formal parameter and variables defined in the procedure. The translation rules for SDL types, both the predefined and user defined can be found in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*.

For each type of procedure in the system the following number of data areas will be allocated:

- There will be as many `yVDef_ProcedureName` structs as the maximum number of concurrent calls (during the complete execution of the program) of the procedure. Concurrent calls occur both when a procedure calls itself recursively within one process instance, and when several process instances of the same process type calls the same procedure during overlapping times.

The `yVDef_ProcedureName` struct is reused by having an avail list, where this two struct is placed when the procedure instance executes a return action. There is one avail list for each procedure type. When a procedure instance should be created, that is, at a call operation, the runtime library first looks at the avail list and reuses an item in the list. Only if the avail list is empty new memory is allocated.

Data types

The predefined SDL type charstring is implemented as `char *` in C and thus requires dynamic memory allocation. The predefined data types Bit_string, Octet_string, and Object_identifier are also implemented using dynamic memory.

The implementation of the SDL sorts Charstring, Bit_string, Octet_string, and Object_identifier is both flexible in length and all memory can be reused.

The mechanism used to release unused memory is to call the `xFree` function in the file `sctos.c`, which uses the standard function free to release the memory.

Charstrings, Bit_strings, Octet_strings, and Object_identifiers are also handled correctly if they are part of structs or arrays. When, for example, a new value is given to a struct having a charstring component, the old charstring value will be released. For all structured types containing any of these types there will also be a `Free` function that is utilized to release all dynamic memory in the structured variable.

Functions for Allocation and Deallocation

The allocation and deallocation of memory is handled by the functions `xAlloc` and `xFree` in the file `sctos.c`. The functions in this file are used for the adoption of the generated applications to the operating system or hardware. The `sctos.c` file is described in detail in “The sctos.c File” on page 3071.

In generated code and in the run-time library the functions `xAlloc` and `xFree` are used in each situation where memory is needed or can be released. `xAlloc` receives as parameter a requested size in bytes and returns the address to a data area of the requested size. All bytes in the data area are set to zero. `xFree` takes the address of a pointer and returns the data area referenced by the pointer to the pool of free memory. It also sets the pointer to 0.

The `xAlloc` and `xFree` functions are usually implemented using some version of the C standard functions for allocation (`malloc`, `calloc`) and deallocation (`free`). Other implementations are of course possible as long as the interface described in the previous section is fulfilled. In a micro controller, for example, it is probably necessary to handle allocation and deallocation directly towards the physical memory.

To prevent memory fragmentation we have used our own avail lists in almost all circumstances. Memory fragmentation is phenomena occurring when a program allocates and de-allocates data areas (of different sizes) in some “random” order. Then small pieces of memory here and there are lost, since their sizes are to small to fit an allocation request. This can lead to a slowly increasing demand for memory for the application.

Note that deallocation of memory is only used for data types. More specific it is used for variables of type:

- Charstring
- Octet_string
- Bit_string
- Object_identifier
- Types created by String (not #STRING) and Bag generator
- Types created by Array generator, if the index type is such that an array in C cannot be used. (General array)
- Types created by Powerset generator, if the component type has the has property as for the index type in general arrays.

This means that if variables of the above mentioned types are not used and the user has not introduced the need for deallocation of memory himself, no memory deallocation will occur. In this case it is of course unnecessary to implement the `xFree` function.

It is easy to trace the need for dynamic memory. As all memory allocation is carried out through the `xAlloc` function and this function is available in source code (in `sctos.c`), it is only necessary to introduce whatever count statements or printout statements that are appropriate.

Compilation Switches

The compilation switches are used to decide the properties of the Master Library and the generated C code. Both in the library and in generated code `#ifdefs` are used to include or exclude parts of the code.

The switches that are used can be divided into four groups.

1. Switches defining properties of the compiler.
2. Switches defining a library version.
3. Switches defining a property of a library version.
4. Switches defining the implementation of a property.

The first group will be discussed in "Adaptation to Compilers" on page 3069.

The following switches define the library version:

Switch	Corresponds to Library
`SCTDEBCOM`	*Simulation*
`SCTDEBCLCOM`	*RealTimeSimulation*
`SCTAPPLCLENV`	*Application*
`SCTDEBCLENVCOM`	*ApplicationDebug* (Simulation with environment)
`SCTPERFSIM`	*PerformanceSimulation* (Library with simulated time, no environment functions, no monitor.)

The definition of the properties of these libraries can be found in `scttypes.h` and will be discussed below. Each library version is specified by the switches in the group property switches that it defines.

New library versions, containing other combinations of property switches, can easily be defined by introducing new library definitions in the `scttypes.h` file.

The property switches discussed below can be used to form library versions. If not stated otherwise for a certain property, all code, variables, struct components, and so on, are either included or excluded using conditional compiling (`#ifdef`), depending on whether the property is used or not.

This means, for example, that all code for the monitor interface will be removed in an application not using the monitor, which makes the application both smaller and faster.

Description of Compilation Switches

XCLOCK

If this compilation switch is not defined then simulated time is used, otherwise the system time is connected to a real clock, via the `sctos.c` function `SDL_Clock`.

XCALENDARCLOCK

This is the same as `XCLOCK` (it will actually define `XCLOCK`), except that if `XCLOCK` is used, time will be zero at system start up, while if `XCALENDARCLOCK` is used, time will be whatever the clock returns at system start up.

XPMCOMM

Define this compilation switch if the application should be able to communicate with signals via the SDL suite communication mechanism. This facility is used to accomplish communicating simulations and simulations communicating with, for example, user interfaces.

XITEXCOMM

This switch should be defined if a generated simulator should be able to communicate with a TTCN simulator.

XENV

If this compilation switch is defined the environment functions `xInitEnv`, `xCloseEnv`, `xInEnv`, and `xOutEnv` will be called at appropriate places.

XTENV

This is the same as `XENV` (it will actually define `XENV`), except that xInEnv should return a time value which is the next time it should be called (a value of type `SDL_Time`). The main loop will call `xInEnv` at the first possible occasion after the specified time has expired, or when the SDL system becomes idle.

XENV_CONFORM_2_3

This switch make signals using a compatible data structure as in SDT 2.3. This means that an extra and unnecessary component `yVarP` is inserted in each signal.

XSIGLOG

This facility makes it possible for a user to implement his own log of the major events in the system. This compilation switch is normally not defined. By defining this switch, each output of a signal, i.e. each call of the function `SDL_Output`, will result in a call of the function `xSignalLog`. Each time a transition is started, the function `xProcessLog` will be called.

These functions have the following prototypes:

```
extern void xSignalLog
  (xSignalNode  Signal,
   int          NrOfReceivers,
   xIdNode    * Path,
   int          PathLength);

extern void xProcessLog
  (xPrsNode P);
```

which are included in `scttypes.h` if `XSIGLOG` is defined.

`Signal` will be a pointer to the data area representing the signal instance.

`NrOfReceivers` will indicate the success of the output according to the following table:

NrOfReceivers	Output Statement Contents
-1:	A TO clause, but no path of channels and signal routes were found between the sender and the receiver.
0:	No TO clause, and no possible receivers were found in the search for receivers.

NrOfReceivers	Output Statement Contents
1:	If the output statement contains a TO clause, a path of channels and signal routes was found between the sender and the receiver. If the output statement contains no TO clause, exactly one possible receiver was found in the search for receivers. The output was thus successful. The only error situation that still might be present is if an output with a TO clause is directed to a process instance that is stopped.

The third parameter, `Path`, is an array of pointer to `IdNodes`, where `Path[0]` refers to the `IdNode` for the sending process, `Path[1]` refers to the first signal route (or channel) in the path between the sender and the receiver, and so on, until `Path[PathLength]` which refers to the `IdNode` for the receiving process.

The parameter `P` in the `xProcessLog` function will refer to the process just about to start executing.

The fourth parameter, `PathLength`, represents thus the number of components in the `Path` array that are used to represent the path for the signal sent in the output. If the signal is sent to or from the environment, either `Path[0]` or `Path[PathLength]` will refer to `xEnvId`, that is to the `IdNode` for the environment process.

In the implementation of the `xSignalLog` and `xProcessLog` functions which should be provided by the user, the user has full freedom to use the information provided by the parameters in any suitable way, except that it is not possible to change the contents of the signal instance. The functions are provided to make it possible for a user to implement a simple log facility in environments where standard IO is not provided, or where the monitor system is too slow or too large to fit. A suitable implementation can be found in the file `sctenv.c`

XTRACE

If this compilation switch is defined, traces of the execution can be printed.

This facility is normally used together with the monitor, but can also be used without the monitor. The file `stdout` must of course be available for printing.

Setting trace values must, without the monitor, be performed in included C code, as the monitor interface is excluded. The trace components are called `Trace_Default` and can be found in `IdNodes` representing system, blocks, and processes, and in the struct `xPrsRec` used to represent a process instance. The values stored in these components are the values given in the Set-Trace command in the monitor. The value undefined is represented by -1.

When the monitor is excluded all trace values will be undefined at startup, except for the system which has trace value 0. This means that no trace is active at start up.

Example 492 –––

Suitable statements to set trace values in C code:

```
xSystemId->Trace_Default = value;
   /* System trace */
xPrsN_ProcessName->Trace_Default = value;
   /* Process type trace */
PId_Var.LocalPId->PrsP->NameNode->Trace_Default =
  value
   /* Process type trace */
PId_Var.LocalPId->PrsP->Trace_Default = value;
   /* Process instance trace */
```

`PId_Var` is assumed to be a variable of type `PId`.

Note:

Note that the variable `xPrsN_ProcessName` is declared, and therefore only available, in the file containing the block where the process is defined (and in files representing processes contained in the block).

–––

XGRTRACE

If this compilation switch is defined it is possible for a simulation to communicate with the Organizer and the SDL Editor to highlight SDL symbols in the graphical representation.

This feature is used together with the monitor to implement graphical trace and commands like Show-Next-Symbol and Show-Previous-Symbol. It is possible to use graphical trace without the monitor in the same way as the ordinary trace (substitute `Trace_Default` with `GRTrace` in the description above). However the graphical trace is synchronized which means that the speed of the application is dramatically reduced.

XCTRACE

Defining this compilation switch makes information available to the monitor about where in the source C code the execution is currently suspended. This facility, which is used together with the monitor, makes it possible to implement the monitor command Show-C-Line-Number.

XMONITOR

If this compilation switch is defined, the monitor system is included in the generated application.

XCOVERAGE

This compilation switch makes it possible to generate coverage tables. It should be used together with `XMONITOR`.

MAX_READ_LENGTH

This macro controls the length of the `char *` buffers used to read values of SDL sorts. A typical usage is when the monitor commands Assign-Value is entered. If large data types are used, it is possible to redefine the sizes of the buffers from their default size (10000 bytes) to something more appropriate.

XSIMULATORUI

This compilation switch should be defined if the generated simulator is to be executed from the Graphical User Interface to the simulator monitor.

XMSCE

This compilation switch should be defined if the generated simulator should be able to generate Message Sequence Charts.

XSDLENVUI

This compilation switch should be defined if it should be possible to start and communicate with a user interface (or another application) from the simulation. This feature should be used together with the monitor and will define the switch `XPMCOMM` (see also this switch).

XNOMAIN

When this compilation switch is defined the functions main and `xMainLoop` are removed using conditional compiling. This feature is intended to be used when a generated SDL application should be part of an already existing application, that is when the SDL system implements a new function in an existing environment. The following functions are available for the user to implement scheduling of SDL actions:

```
extern void xMainInit(
  void (*Init_System) (void)
#ifdef XCONNECTPM
 ,int argc,
  char *argv[]
#endif
  );

#ifdef XNOMAIN
extern void SDL_Execute (void);

extern int SDL_Transition_Prio (void);

extern void SDL_OutputTimer (void);

extern int SDL_Timer_Prio (void);

extern SDL_Time SDL_Timer_Time (void);
#endif
```

The behavior of these functions are as follows:

`xMainInit`: This function should be called to initialize the SDL system before any other function in the runtime library is called. An appropriate way to call `xMainInit` is:

```
#ifdef XCONNECTPM
xMainInit(yInit, argc, argv);
#else
xMainInit(yInit);
#endif
```

The compilation switch XCONNECTPM will be defined if the any switch that requires communication via the SDL suite communication mechanism is defined (XPMCOMM or XGRTRACE).

SDL_Execute: This function will execute one transition by the process instance first in the ready queue.

Before calling this function it must be checked that there really is at least one process instance in the ready queue. This test can be performed using the function SDL_Transition_Prio discussed below.

SDL_Transition_Prio: This function returns the priority of the process first in the ready queue (if signal priorities are used it is the priority of the signal that has caused the transition by the actual process instance). If the ready queue is empty, -1 is returned.

SDL_OutputTimer: This function will execute one timer output and may only be called if there is a timer ready to perform a timer output. This test can be performed with either SDL_Timer_Prio or SDL_Timer_Time described below.

SDL_Timer_Prio: This function returns the priority of the timer first in the timer queue if the timer time has expired for this timer. That is, if Now is greater than or equal to the time given in the Set statement for the timer.

If the timer queue is empty or the timer time for the first timer has not expired, -1 will be returned.

If signal priorities are used, the priority returned is the priority assigned to the timer type (in the timer definition) or the default timer priority; while if process priorities are used the priority returned is the priority of the process that has set the timer.

SDL_Timer_Time: This function returns the time given in the set statement for the first timer in the timer queue. If the timer queue is empty, the largest possible time value (xSysD.xMaxTime) is returned.

Depending on how the SDL system is integrated in an existing environment it might be possible to also use the monitor system. In that case the function xCheckMonitors should be called to execute monitor commands.

```
extern void xCheckMonitors (void);
```

To give some idea of how to use the functions discussed above, an example reflecting the way the internal scheduler in the runtime library works is given below:

Example 493 ––––––––––––––––––––––––––––––––––––––

```
     while (1) {
   #ifdef XMONITOR
       xCheckMonitors();
   #endif
       if ( SDL_Timer_Prio() >= 0 )
         SDL_OutputTimer();
       else if ( SDL_Transition_Prio() >= 0 )
         SDL_Execute();
     }
```

–––

XMAIN_NAME

Sometimes when integrating generated application or simulations in larger environments the main function can be useful but cannot have the name `main`. This name can be changed to something else by defining the macro `XMAIN_NAME`. The main function came be found in the file `sctsdl.c`.

XSIGPRIO

The `XSIGPRIO` compilation switch defines that priorities on signals (set in Output statements) should be used. This switch and the three other switches for priorities given below are, of course, mutually exclusive.

A signal priority is specified with a priority directive (see “Assigning Priorities – Directive #PRIO” on page 2667 in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*, that is by a comment with the following outline:

```
    /*#PRIO 5 */.
```

A priority can be assigned to a signal instance in an output statement by putting a #PRIO directive **last in the output symbol**. In SDL/PR it is possible to put the #PRIO directive both immediately before and immediately after the semicolon ending the output statement. The Cadvanced/Cbasic SDL to C Compiler will first look for #PRIO directives in the output statement. If no directive is found there it will look in the signal definition for the signal for a priority directive. A #PRIO directive should be placed directly **before** the comma or semicolon ending the definition of the signal.

Example 494 ––

```
   SIGNAL
     S1 /*#PRIO 3 */,
     S2 (Integer) /*#PRIO 5 */;
```

––

If no priority directive is found in the output symbol or in the definition of the signal, the default value for signal priority is used. This value is 100. Timers can be assigned priorities in timer definitions in the same way as signals in signal definitions.

The signal priorities will be used to sort the input port of process instances in priority order, so that the signal with highest priority (lowest priority value) is at the first position. Two signals with same priority are placed in the order they arrive. The priority of the signal that can cause the next transition by a process instance is used to sort the ready queue in priority order, so that the process with a signal of highest priority is first. With equal priority, the processes are placed in the order they are inserted into the ready queue. If a continuous signal caused a processes to be inserted into the ready queue, it is the priority of the continuous signal that will be used as signal priority for this “signal”.

Note that a start transition also have a “signal priority”. This is by default also 100 and is set by the macro `xDefaultPrioCreate` described below.

Caution!

Signal priority is not included in SDL according to ITU Recommendation Z.100, and that sorting the signals in the input port of a process instance according to priorities is a direct violation of the SDL standard. This feature is however included for users that need such a behavior to implement their applications.

XPRSPRIO

This compilation switch defines that process priorities should be used. For more information see chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*, section Assigning Priorities - Directive #PRIO.

XSIGPRSPRIO

This compilation switch defines that priorities on signals should be used as first key for sorting in priority order, and process priorities should be used as second key.

XPRSSIGPRIO

This compilation switch defines that process priorities should be used as first key for sorting in priority order, and priorities on signals should be used as second key.

xDefaultPrio...

It is possible to redefine the default priorities for processes, signals, timer signals, continuous signals and start-up signals by defining the symbols below to appropriate values. The default value for these defaults are 100.

```
xDefaultPrioProcess
xDefaultPrioSignal
xDefaultPrioTimerSignal
xDefaultPrioContSignal
xDefaultPrioCreate
```

XOPT

This compilation switch will turn on full optimization (except `XOPTCHAN`), that is, it will define the following switches:

XOPTSIGPARA	XOPTDCL
XOPTFPAR	XOPTSTRUCT
XOPTLIT	XOPTSORT

For more information, see these switches below. The `XOPT` switches should not be used together with the monitor.

XOPTSIGPARA

In the symbol table tree (see section "Symbol Table Tree Structure" on page 2954) there will be one node for each parameter to a signal. These nodes are not necessary in an application and can be removed by defining the compilation switch `XOPTSIGPARA`.

XOPTDCL

There will be a `VarIdNode` in the symbol table tree for each variable declared in processes, procedures, or operator diagram. These nodes are not used in an application (without the monitor) and can be removed by defining the compilation switch `XOPTDCL`.

XOPTFPAR

There will be a `VarIdNode` in the symbol table tree for each formal parameter in a processes, procedures, or operator diagram. These node are not used in an application and may be removed by defining the compilation switch `XOPTFPAR`.

XOPTSTRUCT

For each component in an SDL struct there will be one `VarIdNode` defining the properties of this component. These `VarIdNodes` are not used in an application and can be removed by defining the compilation switch `XOPTSTRUCT`.

XOPTLIT

For each literal in a newtype that will be translated to an enum type, there will be an `LitIdNode` representing the literal. These nodes will not be used in an application and can be removed by defining the compilation switch `XOPTLIT`.

XOPTSORT

Each newtype and syntype, including the SDL standard types, will be represented by an `SortIdNode`. These nodes are not used in an application if all the other `XOPT...` mentioned above are defined.

XNOUSEOFREAL

Defining this compilation switch will remove all occurrences of C `float` and `double` types, and means for example that the SDL type Real is no longer available.

This switch is intended to be used in situations when it is important to save space, to see to that the library functions for floating type operations are not necessary to load. It cannot handle situations when the user includes floating type operations in C code, for example #CODE directives. Another consideration is if BasicCTypes.pr, or other ADTs, are included in the system. If so, it is required that types dependent on SDL Real be removed from these packages.

XNOUSEOFOBJECTIDENTIFER

Defining this switch will remove all code for the SDL predefined sort Object_identifier.

XNOUSEOFOCTETBITSTRING

Defining this switch will remove all code for the SDL predefined sorts Bit_string, Octet, and Octet_string.

Special consideration needs to be taken if BasicCTypes.pr, or other ADTs, are included in the system. If so, it is required that types dependent on these types be removed from these packages.

XNOUSEOFEXPORT

By defining this switch the user states that he is not going to use the export - import concept in SDL.

> **Caution!**
>
> An attempt to perform an import operation when XNOUSEOFEXPORT is defined will result in a compilation error, as the function `xGetExportAddr` is not defined.

XNOUSEOFSERVICE

This compilation switch can be defined to save space, both in data and in the size of the kernel, if the SDL concept service is not used. If services are used and this switch is defined, there will be compilation errors (probably many!), when the generated code is compiled.

XPRSOPT

Section “Create and Stop Operations” on page 3007 describes how `xLocalPIdRec` structs are allocated for each created process instance, and how these structs are used to represent process instances even after they have performed stop actions. This method for handling `xLocalPIdRecs` is required to be able to detect when a signal is sent to a process instance that has performed a stop operation.

In an application that is going to run for a “long” period of time and that uses dynamic processes instances, this way of handling `xLocalPIdRecs` will eventually lead to no memory being available.

By defining the compilation switch `XPRSOPT`, the memory for the `xLocalPIdRecs` will be reused together the `yVDef_ProcessName` structs. This has two consequences:

1. The need for memory will not increase due to the use of dynamic processes (the memory need depends on the maximum number of concurrent instances).
2. It will no longer be possible to always find the situation when a signal is sent to a process instance that has performed a stop action.

More precisely, if we have a PId variable that refers to a process instance which performs a stop operation and after that a create operation (on the same process instance set) is performed where the same data area is reused, then the PId variable will now refer to the new process instance.

This means, for example, that signals intended for the old instance will be sent to the new instance. Note that it is still possible to detect signal sending to processes in the avail list even if `XPRSOPT` is defined.

XOPTCHAN

This switch can be used to remove all information about the paths of channels and signal routes in the system. The following memory optimization will take place:

- The two `ChannelIdNodes` for each channel, signal route, and gate are removed.
- The `ToId` component in the `xPrsIdNodes` representing processes is removed.
- A number of functions in the library (`sctsdl.c`) are no longer needed and are removed.

When the information about channels, signal routes, and gates is not present two types of calculations can no longer be performed:

1. To check if there is a path of channels and signal routes between the sender and the receiver in an OUTPUT statement with a TO clause.

This is no problem as this is just an error test that we probably do not want to be performed in an application.

2. To calculate the receiver in an OUTPUT without TO clause, if the Cadvanced/Cbasic SDL to C Compiler has not performed this calculation at generate time (see *"Calculation of Receiver in Outputs" on page 2585 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler*). This is more serious, as it means that **OUTPUT without TO cannot always be used. The restrictions are:**
   - No outputs without to in process types, or in process in block or system types.
   - No outputs without to, designated to a process in a SEPARATE unit.

Caution!

If the `XOPTCHAN` switch is defined and still OUTPUT without TO clause are used (which the Cadvanced/Cbasic SDL to C Compiler cannot optimize), there will be a C compilation error saying that the name `xNotDefPId` is not defined.

In an ordinary SDL system OUTPUTs without TO must be used to start up the communication between different parts of the system, as there is no other way in SDL to distribute the PId values needed for OUTPUTs with TO.

This problem is solved if the Cadvanced/Cbasic SDL to C Compiler can calculate the receiver. Otherwise the data type `PIdList` in the library of abstract data types is intended to solve this problem. It is described in chapter 63, *The ADT Library*. When this data type is used, global PId literals my be introduced, implemented as SDL synonyms. These literals can then be used to utilize OUTPUT statements with TO clauses from the very beginning.

X_LONG_INT

The SDL sort Integer is translated to int in C. To translate the Integer sort to long int instead, just define the compilation switch `X_LONG_INT`.

XENVSIGNALLIMIT

If this switch is defined, only a limited number of signals will be stored in the input port of the Env function. The limit is equal to the value defined for `XENVSIGNALLIMIT` and is normally set to 20.

XEALL

This switch will define all error handling switches (XE...) and `XASSERT` given below.

XECREATE

This switch will report if the initial number of instances of a process type is greater than the maximum number.

XECSOP

This switch will report error situations in ADT operator.

XEDECISION

This switch will report if no path out from a Decision is found.

XEEXPORT

This switch will report errors during Import actions.

XEFIXOF

This switch will report overflow when an SDL Real value is converted to an SDL Integer value using the operator Fix.

XEINDEX

This switch will report value out of range for array index.

XEINTDIV

This switch will report division by zero in an integer division.

XEOUTPUT

This switch will report errors during Output operations.

XERANGE

This switch will report range errors when a value is assigned to a variable of a sort containing range conditions.

XEREALDIV

This switch will report division by zero in a real division.

XEVIEW

This switch will report errors in View operations

XECHOICE

This switch will turn on error reports when accessing non-active choice components.

XEOPTIONAL

This switch will turn on error reports when accessing non-present optional struct components.

XEUNION

This switch will turn on error reports when accessing non-active union components.

XEREF, XEOWN

These switches turn on error checking on pointers (generator Ref and Own).

XASSERT

By defining this switch the possibility to define user assertions which is described in “Assertions” on page 2127 in chapter 50, *The SDL Simulator*.

XTRACHANNELSTOENV

When using partitioning of a system a problem during the redirection of channels is that the number of channels going to the environment is not known at code generation time, which means that the size of the data area used for the connections is not known. This problem is solved in two ways.

Either the function handling redirections allocates more memory, which is the default, or the user specifies how many channels that will be redirected (which could be difficult to compute, but will lead to less need of memory).

In the first case (allocation of more memory) the macros:

```
#define XTRACHANNELSTOENV  0
#define XTRACHANNELLIST
```

should be defined like above. This is the standard in scttypes.h. If the user wants to specify the number of channels himself then

```
#define XTRACHANNELSTOENV  10
#define XTRACHANNELLIST    ,0,0,0,0,0,0,0,0,0,0
```

i.e. `XTRACHANNELSTOENV` should be the number of channels, while `XTRACHANNELLIST` should be a list of that many zeros.

XDEBUG_LABEL

It is for debugging purposes sometimes of interest to introduce extra labels. The macro `XDEBUG_LABEL` is inserted in the code for each input symbol. As macro parameter it has a name which is the name of the state concatenated with an underscore concatenated with the signal name.

Example 495 ––

```
state State1; input Sig1;
state State2; input *;
state *; input Sig2;
```

In the generated code for these input statements the following macros will be found:

```
XDEBUG_LABEL(State1_Sig1)
XDEBUG_LABEL(State2_ASTERISK)
XDEBUG_LABEL(ASTERISK_Sig2)
```

A suitable macro definition to introduce label would be:

```
#define XDEBUG_LABEL(L)  L: ;
```

To use these label the usage of SDL must be restricted in one area. The same state may not receive two different signals with the same name! This is allowed and handled by the SDL suite. The signal have to be defined at different block or system level and the outermost signal must be referenced with a qualifier.

–––

XCONST, XCONST_COMP

Using these compilation switches most of the memory used for the `IdStructs` can be moved from RAM to ROM. This depends of course on the compiler and what properties it has.

The following macro definitions can be inserted:

```
#define XCONST const
#define XCONST_COMP const
```

This will introduce **const** in the declaration of most of the `IdStructs`. It is then up to the compiler to handle const.

The XCONST_COMP macro is used to introduce const on components within a struct definition. This is necessary for some compilers to accept const on the struct as such.

If `const` is successfully introduced, there is a lot of RAM memory that will be saved, as probably 90% of the data area for `IdStructs` can be made `const`.

Compilation Switches – Summary

The property switches are in principle independent, except for the relations given in the descriptions above, and it should always be possible to any combination.

The number of combinations is, however, so huge that it is impossible for us to even compile all combinations. If you happen to form a combination that does not work, please let us know, so that we either can correct the code, or, if that is not possible, publish a warning against that combination.

The switches defining a standard library version will define the following property switches:

SCTDEBCOM XPRSPRIO XPARTITION XEALL XMONITOR XTRACE XCTRACE XMSCE XCOVERAGE XGRTRACE XPMCOMM XSDLENVUI XITEXCOMM XSIMULATORUI	**SCTDEBCLCOM** XCLOCK XPRSPRIO XPARTITION XEALL XMONITOR XTRACE XCTRACE XMSCE XCOVERAGE XGRTRACE XPMCOMM XSDLENVUI XSIMULATORUI
SCTAPPLCLENV XCALENDARCLOCK XENV XPRSPRIO XOPT XPRSOPT	**SCTDEBCLENVCOM** XCALENDARCLOCK XPRSPRIO XPARTITION XENV XPRSOPT XEALL XMONITOR XTRACE XCTRACE XMSCE XCOVERAGE XGRTRACE XPMCOMM XSDLENVUI XSIMULATORUI
SCTPERFSIM XEALL XPRSPRIO	

The lowest layer of switches (that handle the implementation details) are set up using the three layers above. These switches will not be discussed here. Please refer to the source code files `scttypes.h` and `sctsdl.c` for more details.

Creating a New Library

Caution!

If you create new versions of the library, make sure that the library and the generated code are compiled with the same compilation switches. If not, you might experience any type of strange behavior in the generated application!

This section describes how to generate a new library. The following topics are covered:

- The **directory structure** for source and object code.
- The **`sdtsct.knl`** file, which determines what libraries the Analyzer knows about, that is, what libraries that will be shown when Generate-Options is selected.
- The **`comp.opt`** file and the **`makeoptions`** (**`make.opt`** **in Windows**) file, which determines the properties of an object code library.
- The make file **`Makefile`**, which includes the `makeoptions` `(make.opt)` file and generates a new object code library with the properties given by the included `makeoptions (make.opt)` file.
- The relations with the generated make files for SDL systems will also be discussed.

Directory Structure

The structure of files and directories used for the Cadvanced/Cbasic SDL to C Compiler libraries is shown in Figure 552 The directory `sdtdir` is in the installation:

```
<installation directory>/sdt/sdtdir/<machine
dependent dir>
```

where <*machine dependent dir*> is for example `sunos5sdtdir` on SunOS 5, `hppasdtdir` on HP, and `wini386` in Windows. (**In Windows**, `/` should be replaced by `\` in the path above.)

This directory is here called sdtdir and is in UNIX normally referred to by the environment variable `sdtdir`.

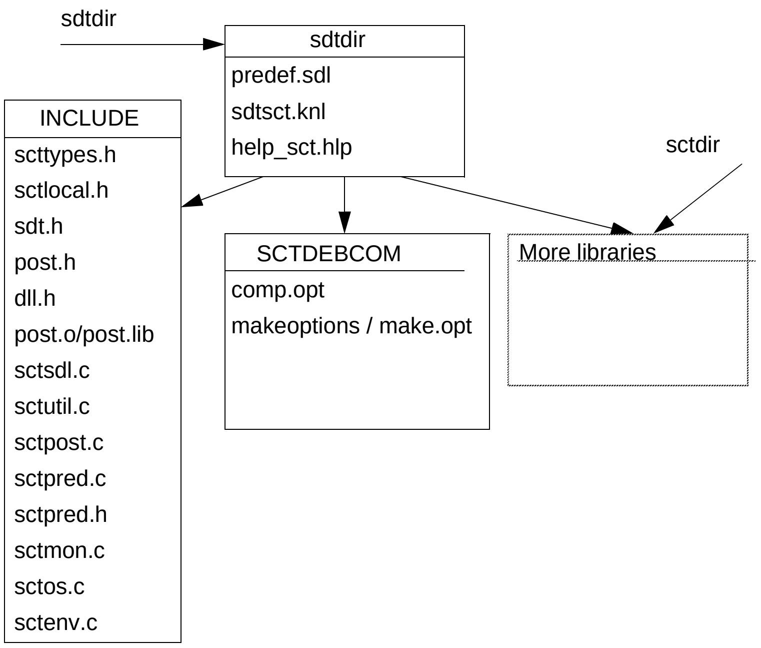

Figure 552: Directory structure

"sctdir" is a reference to an object library and is usually setup as a parameter in the call to make. It can also be an environment variable.

In the `sdtdir` directory three important files are found:

3. `predef.sdl` contains the definition of the predefined sorts in SDL.
4. `sdtsct.knl` contains a list of the available libraries that can be used together with code generated by the Cadvanced/Cbasic SDL to C Compiler.
5. `help_sct.hlp` contains the help information that can be obtained using the monitor command help.

The file `predef.sdl` is read by the SDL Analyzer during analysis, while the file `sdtsct.knl` is used to present the available libraries in the Make dialog in the Organizer (see "Make" on page 119 in chapter 2, *The Organizer*).

In the `INCLUDE` directory, there are two important groups of files:

1. The source code files for the runtime library:
   `scttypes.h`, `sctlocal.h`, `sctsdl.c`, `sctutil.c`, `sctpost.c`, `sctpred.h`, `sctpred.c`, `sctmon.c`, `sctos.c` and `sctenv.c`
2. The files necessary to include communication with other SCT applications: `post.h`, `post.o (post.lib` **in Windows**`)`, `sdt.h`, `itex.h`.

In parallel with the `INCLUDE` directory there are a number of directories for libraries in object form. The `SCTDEBCOM` directory in Figure 552 is an example of such a directory. Each of these directories will contain three files: `comp.opt`, `makeoptions` (`make.opt` **in Windows**).

The `comp.opt` file determines the contents of the generated makefile and how make is called. For more details see below.

The `makeoptions (make.opt)` file describes the properties of the library, such as the compiler used, compiler options, linker options, and so on.

To guarantee the consistency of, for example, compilation flags between the SDL system and the kernel, the `makeoptions (make.opt)` file is used both by the make file compiling the library (`Makefile`) and by the generated make files used to compile the generated SDL system. Non-consistency in this sense between the library and the SDL system will make the result unpredictable.

File sdtsct.knl

The `sdtsct.knl` file describes which libraries that are available. This is presented by the Organizer in the *Make* dialog see “Make” on page 119 in chapter 2, *The Organizer*). The `sdtsct.knl` file has the following structure. Each available library is described on a line of its own. Such a line should first contain the name of the library (the name presented in the dialog), then the path to the directory containing the library, and last a comment up to end of line.

The path to the library can either be the complete path or a relative path. A relative path is relative to the environment variable `sdtdir` **(on UNIX)** or `SDTDIR` **(in Windows)**, if that variable is defined. Otherwise it is relative to the SDL suite installation.

Example 496 ---

```
Simulation           SCTDEBCOM
RealTimeSimulation   SCTDEBCLCOM
Application          SCTAPPLCLENV
ApplicationDebug     /util/sct/SCTDEBCLENVCOM

ApplicationDebug     x:\sdt\sdtdir\dbclecom
MyTestLibrary        ..\testlib\dbcom
```

Not that the two last lines is examples for Windows, while the fourth line is for UNIX.

The Organizer will look for an `sdtsct.knl` file first in the directory the SDL suite is started from, then in the home directory for the user, and then in the directory referenced by the environment variable `sdtdir` (`SDTDIR`) if it is defined, and in the directory where the SDL suite was installed.

File Makefile

In each directory that contains a library version there is a `Makefile` that can “make” the library. To create a new library after an update of the source code, change directory to the directory for the library and execute the `Makefile`. The `Makefile` uses the `makeoptions (make.opt)` file in the directory to get the correct compilation switches and other relevant information.

Caution!

Do not generate and test libraries in the installation directory structure. Create an appropriate copy.

Note:

The environment variables (if used) `sdtdir (SDTDIR)` and `sctdir (SCTDIR)` need not necessarily refer to directories in the installation directory. Any directory containing the relevant files may be used.

File comp.opt

This file determines the details of the generated make files, and the command issued to execute the makefile. A comp.opt file contains zero, one or more initial lines starting with a #. These lines are treated as comments. After that it contains five lines of essential data.

- Line 1: How to include the makeoptions (make.opt) file
- Line 2: Compile script
- Line 3: Link script
- Line 4: Command to run make
- Line 5: How to build a library (archive). Used for coders/decoders.

On each of these lines % codes can be used to insert specific information.

On all five lines:

```
%n : newline
%t : tab
%d : target directory
%s : source directory
%k : kernel directory
%f : base name of generated executable (no path, no
     file extension). NOT on line 2 or 5.
```

On line 2, the compile script:

```
%c : c file in compile script
%C : c file in compile script, without extension
%o : resulting object file in compile script
```

On line 3, the link script:

```
%o : list of all object files in link script
%O : list of all object files in link script, with
     \ followed by newline between files
%e : executable file in link script
```

On line 4, the make command:

```
%m : name of generated makefile
```

On line 5, the archive command:

```
%o : list of object files, i.e. $(sctCODER_OBJS).
%a : the archive file, i.e.
     libstcoder$(sctLIBEXTENSION)
```

Example 497: comp.opt file for UNIX ----------------------------

```
# makefile for unix make
include $(sctdir)/makeoptions
%t$(sctCC) $(sctCPPFLAGS) $(sctCCFLAGS) $(sctIFDEF) %c -o %o
%t$(sctLD) $(sctLDFLAGS) %o -o %e
make -f %m sctdir=%k
%t$(sctAR) $(sctARFLAGS) %a %o
```

File makeoptions / make.opt

This file has the following structure:

Example **on UNIX:**

```
# #
sctLIBNAME      = Simulation
sctIFDEF        = -DSCTDEBCOM
sctEXTENSION    = _smd.sct
sctOEXTENSION   = _smd.o
sctLIBEXTENSION=   _smd.a
sctKERNEL       = $(sctdir)/../INCLUDE
sctCODERDIR     = $(sctdir)/../coder

#Compiling, linking
sctCC           = cc
sctCODERFLAGS   = -I$(sctCODERDIR)
sctCPPFLAGS     = -I. -I$(sctKERNEL) $(sctCODERFLAGS)
 $(sctCOMPFLAGS) $(sctUSERDEFS)
sctCCFLAGS      = -c -Xc
sctLD           = cc
sctLDFLAGS      =
sctAR           = ar
sctARFLAGS      = rcu

all : default

# below this point there are a large number of
# compilation rules for compiling the Master Library
# and the Coder library (used for encoding/decoding)
# The following name of any importance are defined:

sctLINKKERNEL  =
sctLINKKERNELDEP =
sctLINKCODERLIB =
sctLINKCODERLIBDEP =
```

The information to the right of the equal signs should be seen as an example. These environment variables set in the `makeoptions` (`make.opt`) file should specify:

- **`sctLIBNAME`**. This is only used by the `Makefile` to report what it is doing.
- **`sctIFDEF`**. This variable should specify what compilation switches, among those defined by the Cadvanced/Cbasic SDL to C Compiler system, that should be used. Usually there is one switch defining the library version.

- **`sctEXTENSION`**. This is used to determine the file extension of the executable files.
- **`sctOEXTENSION`**. This is used to determine the file extension of the object files.
- **`sctLIBEXTENSION`**. The extension of the archive/library
- **`sctKERNEL`**. Directory of Master Library source code.
- **`sctCODERDIR`**. The directory for the source code of the coders/decoders.
- **`sctCC`**. This defines the compiler to be used.
- **`sctCODERFLAGS`**. Compilation options needed to compile the coder/decoder files
- **`sctCPPFLAGS`**. This variable should give the compilation flag necessary to specify where the C preprocessor can find the include files `scttypes.h`, `sctlocal.h`, `sctpred.h`, `sdt.h`, and `post.h`.
- **`sctCCFLAGS`**. This should specify other compiler flags that should be used, as for example -g (Sun cc) or -v (Borland bcc32) for debug information, -O for optimization.
- **`sctLD`**. This defines the linker to be used.
- **`sctLDFLAGS`**. This should specify other flags that should be used in the link operation.
- **`sctAR`**. The archive application
- **`sctARFLAGS`**. Flags to sctAR.
- **`sctLINKKERNEL`**. This variable should specify the `.o` files for the Master Library source files. It will be used in the link command in the generated make file.
- **`sctLINKKERNELDEP`**. Used to implement the dependencies to recompile the kernel when it is needed.
- **`sctLINKCODERLIB`**. This variable should specify the `.o` files for the Coder Library source files. It will be used in the link command in the generated make file.

- **`sctLINKCODERLIBDEP`**. Used to implement the dependencies to re-compile the Coder Library when it is needed.

Generated Make Files

The generated make files for an SDL system will as first action include the `makeoptions` (`make.opt`) file in the directory referenced by the environment variable `sctdir`. It will then use the variables `sctIFDEF`, `sctLINKKERNEL`, `sctCC`, `sctCPPFLAGS`, `sctCCFLAGS`, `sctLD`, and `sctLDFLAGS` to compile and link the SDL system with the selected library.

The make file is generated and executed by the Cadvanced/Cbasic SDL to C Compiler.

Example 498 ––

Below, a UNIX make file generated for the SDL system *example* is shown.

```
# makefile for System: example

sctAUTOCFGDEP =
sctCOMPFLAGS = -DXUSE_GENERIC_FUNC

include $(sctdir)/makeoptions

default: example$(sctEXTENSION)

example$(sctEXTENSION): \
  example$(sctOEXTENSION) \
  $(sctLINKKERNELDEP)
        $(sctLD) $(sctLDFLAGS) \
  example$(sctOEXTENSION) $(sctLINKKERNEL) \
  -o example$(sctEXTENSION)

example$(sctOEXTENSION): \
  example.c
        $(sctCC) $(sctCPPFLAGS) $(sctCCFLAGS) \
  $(sctIFDEF) example.c -o example$(sctOEXTENSION)
```
––

Adaptation to Compilers

In this section the necessary changes to the source code to adapt it to a new environment are discussed. Adapting to a new environment could mean moving the code to new hardware or using a new compiler.

There are two parts of the source code that might need changes:

1. In `scttypes.h` there is a section defining the properties of different compilers, where a new compiler can be added.
2. In `sctos.c` the functions that depend on the operating system or hardware are collected. These might need to be changed due to a new compiler, a new OS, or a new hardware.

In “Compiler Definition Section in scttypes.h” on page 3069 the compiler definition section in `scttypes.h` is discussed in detail, while `sctos.c` is treated in “The sctos.c File” on page 3071.

Compiler Definition Section in scttypes.h

Caution!

Do not to use the compiler `/usr/ucb/cc`. Our experience is that the bundled compiler is subject to generating compilation errors.

Instead, we recommend to run the unbundled compiler `/opt/SUNWSpro/bin/cc` or the GNU C compiler.

In `scttypes.h` the properties of the compiler is recognized by the compiler/computer dependent switches set by the compiler:

```
#if defined(__linux)
#define SCT_POSIX

#elif defined(__sun)
#define SCT_POSIX

#elif defined(__hpux)
#define SCT_POSIX

#elif defined(__CYGWIN__)
#define SCT_POSIX

#elif defined(QNX4_CC)
#define SCT_POSIX
```

```
#elif defined(__BORLANDC__)
#define SCT_WINDOWS

#elif defined(_MSC_VER)
#define SCT_WINDOWS

#else
#include "user_cc.h"

#endif
```

Basically this section distinguishes between Unix-like/POSIX compilers and Windows compilers. In the case the compiler is not in the list above, the user must configure it himself by writing a file `user_cc.h`, which is best placed in the target directory.

The compilers above are:

- __linux : gcc on linux
- __sun : different compilers on SUN
- __hpux : different compilers on HP
- __CYGWIN__ : gcc on windows, for more information please see http://sources.redhat.com/cygwin/
- QNX4_CC : QNX
- __BORLANDC__ : Borland compiler on Windows
- _MSC_VER : Microsoft compiler on Windows

After this compiler configuration section a general configuration section follows:

```
#if defined(SCT_POSIX) || defined(SCT_WINDOWS)
#define XMULTIBYTE_SUPPORT
#endif

#include <string.h>
#include <stdlib.h>
#include <limits.h>
#include <stdarg.h>
#ifdef XREADANDWRITEF
#include <stdio.h>
#ifdef XMULTIBYTE_SUPPORT
#include <locale.h>
#endif
#endif

#ifndef GETINTRAND
#define GETINTRAND      rand()
#endif
#ifndef GETINTRAND_MAX
#define GETINTRAND_MAX  RAND_MAX
#endif
```

```
#ifndef xptrint
#if (UINT_MAX < 4294967295)
#define xptrint         unsigned long
#define X_XPTRINT_LONG
#else
#define xptrint         unsigned
#endif
#endif

#ifndef xint32
#if (INT_MAX >= 2147483647)
#define xint32          int
#define X_XINT32_INT
#else
#define xint32          long int
#endif
#endif
```

First, the presence of multi-byte character support is set up. Then a number of standard include files are included, followed by setting up properties for random number generation. Last the two types, `xptrint`, which defines an `unsigned int` type with the same size as an address, and `xint32`, which defines a 32-bits int type, is configured.

The last three parts in this section handle the utility functions needed by `sctos.c` to implement some of the operating system dependent functions. Please see below where `sctos.c` is discussed in detail.

The sctos.c File

The following important functions are defined in `sctos.c`

```
extern void * xAlloc (xptrint Size);

extern void xFree (void **P);

extern void xHalt (void);

#ifdef XCLOCK
extern SDL_Time SDL_Clock (void);
#endif

#if defined(XCLOCK) && !defined(XENV)
extern void xSleepUntil (SDL_Time WakeUpTime);
#endif

#if defined(XPMCOMM) && !defined(XENV)
extern int xGlobalNodeNumber (void);
#endif
```

```
#if defined(XMONITOR) && !defined(XNOSELECT)
extern xbool xCheckForKeyboardInput (
  long xKeyboardTimeout);
#endif
```

Several of these functions have three different implementations, one for SCT_POSIX, one for SCT_WINDOWS and one for other cases. The other cases solution is “an empty implementation” that does not do anything. If the standard solutions in sctos.c do not fit the needs of a certain application, any of the functions above can be supplied by the user instead. By defining some of the switches:

- XUSER_ALLOC_FUNC
- XUSER_FREE_FUNC
- XUSER_HALT_FUNC
- XUSER_CLOCK_FUNC
- XUSER_SLEEP_FUNC
- XUSER_KEYBOARD_FUNC

the corresponding function or functions are removed from sctos.o and have to be supplied by the user instead.

xAlloc

The function `xAlloc` is used to allocate dynamic memory and is used throughout the runtime library and in generated code. The function is given a size in bytes and should return a pointer to a data area of the requested size. All bytes in this data area are set to zero. The standard implementation of this function uses the C function `calloc`.

A user who wants to estimate the need for dynamic memory can introduce statements in `xAlloc` to record the number of calls of `xAlloc` and the total requested size of dynamic memory. Please note two things. A program using the monitor requires more dynamic memory than a program not using the monitor, so estimates should be made with the appropriate compilation switches. A call of `calloc` will actually allocate more memory than is requested to make it possible for the C runtime system to deallocate and reuse memory. The size of this additional memory is compiler-dependent.

A user who wants to handle the case when no more memory is available at an allocation request can implement that in `xAlloc`. In the standard implementation for `xAlloc` a test if calloc returns 0 can be introduced, at which the program can be terminated with an appropriate message.

xFree

The function xFree is used to return memory to the list of free memory so it can be reused by subsequent calls of xAlloc. The standard implementation of this function uses the C function free. In very simple cases, no data types using dynamic memory are used and no other introduction of dynamic data by the user, this function will not be used.

The parameter of the xFree function, is the address of the pointer to the allocated memory.

Example 499 Using the xFree function –––––––––––––––––––––––––

```
unsigned char *ptr;
ptr = xAlloc(100);
xFree (&ptr);       /* NOTE: Not xFree(ptr); */
```

––

xHalt

The function xHalt is used to exit from a program and is in the standard implementation using the C function exit to perform its task.

SDL_Clock

The function SDL_Clock should return the current time, read from a clock somewhere in the OS or hardware. The return value is of type SDL_Time, that is a struct with two 32-bits integer components, representing seconds and nanoseconds in the time value.

```
typedef struct {
  xint32  s;        /* for seconds */
  xint32  ns;       /* for nanoseconds */
} SDL_Time;
```

The standard implementation of SDL_Clock uses the C function **time**, which returns the number of seconds since some defined date.

Note:

Note that the C function time only handles full seconds.

In an embedded system or any other application that requires better time resolution, or when the C function time is not available, SDL_Clock should be implemented by the user.

Note:

If an application does not require a connection with real time (for example if it is not using timers and should run as fast as possible), there is no need for a clock function. In such a case it is probably suitable to use simulated time by not defining the compilation switch `XCLOCK`, whereby `SDL_Clock` is never called and does not need to be implemented. An alternative is to let `SDL_Clock` always return the time value 0.

A typical implementation in an embedded system is to have hardware generating interrupts at a predefined rate. At each such interrupt a variable containing the current time is updated. This variable can then be read by `SDL_Clock` to return the current time.

Caution!

The variable must be protected from updates during the period of time that `the SDL_Clock` reads the clock variable.

Calling the interrupt routine while the `SDL_Clock` reads the clock variable would cause a system disaster.

xSleep_Until

The function `xSleep_Until` is given a time value, as a value of type `SDL_Time` (see above) and should suspend the executing until this time is reached, when it should return.

This function is used only when real time is used (the switch `XCLOCK` is defined) and when there is no environment functions (`XENV` is not defined). The `xSleep_Until` function is used to wait until the next event is scheduled when there is no environment that can generate events.

xGlobalNodeNumber

The function `xGlobalNodeNumber` is used to assign unique numbers to each SDL system which is part of an application.

If environment functions are used for an SDL system this function should be implemented there. If, however, we have communicating simulations, there are no `env` functions and the `xGlobalNodeNumber` function is defined in `sctos.c` instead.

So the `xGlobalNodeNumber` function is only used if `XPMCOMM` is defined and `XENV` is not defined. As this function is only used in the case of a communicating simulation, it is only necessary to implement it for computers/compilers that communicate with the SDL suite, which means that it is not interesting for a user to change the standard implementation of this function. The implementation calls the function `getpid`, and uses thus the OS process number as global node number.

xCheckForKeyboardInput

The function `xCheckForKeyboardInput` is used to determine if there is a line typed on the keyboard (`stdin`) or not. If this is difficult to implement it can instead determine if there are any characters typed on the keyboard or not. This function is only used by the monitor system, (when `XMONITOR` is defined).

The `xCheckForKeyboardInput` function is used to implement the possibility to interrupt the execution of SDL transitions by typing `<Return>` and to handle polling of the environment (`xInEnv` or its equivalent when communicating simulations is used) when the program is waiting at the “`Command` :” prompt in the monitor.

List of All Compilation Switches

Introduction

This section is a reference to the macros that are used together with the generated C code from the Cadvanced/Cbasic SDL to C Compiler. Here the macros are just enumerated and explained. The section is divided in a number of subsection, each treating one major aspect of the code. Within the subsections the macros are enumerated in alphabetic order.

Information about some of the macros (Library Version Macros, Compiler Definition Section Macros, and General Properties) can also be found in the section “Compilation Switches” on page 3041.

To fully understand the descriptions of the macros in this section it is also necessary to know the basic data structures used, especially for the static structures, i.e. the xIdNodes. This information can be found in the section “The Symbol Table” on page 2954.

The information about the data types used for the dynamic structure of the system, i.e. about process instances, signal, timers, and so on, are also of interest. This can be found in “The SDL Model” on page 2992.

Library Version Macros

SCTAPPLCLENV

Application.

SCTAPPLENV

Application without clock.

SCTDEB

Stand-alone simulator for any environment. Should be executed from OS.

SCTDEBCL

Stand-alone simulator with real time for any environment. Should be executed from OS.

SCTDEBCLCOM

Simulator with real time for host. Can be executed from the SDL suite or from OS.

SCTDEBCLENV

Stand-alone simulator, with real time and env functions, for any environment. Should be executed from OS.

SCTDEBCLENVCOM

Simulator, with real time and env functions, for any environment. May be executed from OS or from simulator GUI.

SCTDEBCOM

Simulator for host. Can be executed from the SDL suite or from OS.

SCTOPT1APPLCLENV

Application with minimal memory requirements. Real cannot be used. No channel information

SCTOPT2APPLCLENV

Application with minimal memory requirements. Real cannot be used. Const for all channel information.

SCTPERFSIM

Suitable for execution of performance simulations.

Compiler Definition Section Macros

SCT_POSIX

Set up for UNIX/POSIX like compilers/systems.

SCT_WINDOWS

Set up for compilers on Windows

Some Configuration Macros

COMMENT(P)

Should be defined as:

```
    #define COMMENT(P)
```

The macro is used to insert comments in included C code. See Example 360 on page 2613.

GETINTRAND

A random generation function. Usually `rand()` or `random()`.

GETINTRAND_MAX

The max int value generated by function mentioned in GETINTRAND. Usually RAND_MAX or 2147483647 (32-bit integers).

SCT_VERSION_4_5

Defined in generated code if the Cadvanced/Cbasic SDL to C Compiler version 4.5 was used.

XCAT(P1,P2)

Should concatenate token P1 and P2. Possibilities:

```
   #define XCAT(P1,P2) P1##P2
```

or

```
   #define XCAT(P1,P2) P1/**/P2
```

or

```
   #define XCAT(P1,P2) XCAT2(P1)P2
   #define XCAT2(P2) P2
```

XMULTIBYTE_SUPPORT

Should be set if the compiler supports multi byte characters.

XNOSELECT

Should be defined if there is no support for the select function found in UNIX operating systems. This is used to implement “user defined interrupt” by typing the return key while simulating.

XNO_VERSION_CHECK

If this macro is defined there will be no version check between the generated code and the `scttypes.h` file.

XSCT_CBASIC

Defined in generated code if Cbasic was used.

XSCT_CADVANCED

Defined in generated code if Cadvanced was used.

X_SCTTYPES_H

Defined in `scttypes.h` in a way that it possible to include the `scttypes.h` file several times without any problems.

X_XINT32_INT

Should be defined if `xint32` is `int`.

X_XPTRINT_LONG

Should be defined if `xptrint` is `unsigned long`.

General Properties

TARGETSIM

Can be used to connect an application with a monitor on a target system with the SDL suite running on a host computer.

XASSERT

Detect and report user defined assertions that are not valid.

XCALENDARCLOCK

Use the clock function in `sctos.c` (not simulated time). Time is whatever the clock function returns.

XCLOCK

Use the clock function in `sctos.c` (not simulated time). Time is zero at system start up.

XCOVERAGE

Compile with code to store information about the current coverage of the SDL system. This information can also be printed in the monitor.

XCTRACE

Compile preserving the possibility to report the current C line number during simulations.

XEALL

Defines XEOUTPUT, XEINTDIV, XEREALDIV, XECSOP, XEFIXOF, XERANGE, XEINDEX, XECREATE, XEDECISION, XEEXPORT, XEVIEW, XEERROR, XEUNION, XECHOICE, XEOPTIONAL, XEREF, XEOWN, and XASSERT.

For more information, see these macros.

XECHOICE

Detect and report attempts to access non-active components in Choice variables.

XECREATE

Detect and report if more static instances are created at start up, than the maximum number of concurrent instances.

XECSOP

Detect and report errors in ADT operators.

XEDECISION

Detect and report when there is no possible path out from a decision.

XEERROR

Detect and report the usage of the error term in an SDL expression.

XEEXPORT

Detect and report errors in import actions.

XEFIXOF

Detect and report integer overflow in the operator fix.

XEINDEX

Detect and report index out of bounds in arrays.

XEINTDIV

Detect and report integer division with 0.

XENV

Call the env functions.

XENV_CONFORM_2_3

Insert the `VarP` pointer in the `xSignalNode` so that signals conform with their implementation in SDT 2.3.

XEOPTIONAL

Detect and report attempts to access optional struct components that are not present.

XEOUTPUT

Detect and report warnings in outputs (mainly outputs where signal is immediately discarded).

XEOWN

Detect and report illegal usage of Own and ORef pointers.

XERANGE

Detect and report subrange errors.

XEREALDIV

Detect and report real division with 0.0.

XEREF

Detect and report attempts to dereference null pointer.

XEUNION

Detect and report attempts to access non-active components in a #UNION.

XEVIEW

Detect and report errors in view actions.

XGRTRACE

Compile with the trace in source SDL graphs enabled.

XITEXCOMM

Enable the possibility for an executable to communicate with the TTCN suite via the Postmaster.

XMAIN_NAME

If this macro is defined the main function in `sctsdl.c` will be renamed to the name given by the macro.

XMONITOR

Compile with the monitor system. This macro will implicitly set up a number of other macros as well.

XMSCE

Compile with the MSC trace enabled.

XNOMAIN

If this macro is defined the main function in `sctsdl.c` will be removed.

XPMCOMM

Enable the possibility for an executable to communicate via the Postmaster.

XPRSPRIO

Use priorities on process instance sets.

XPRSSIGPRIO

Use first priorities on process instance sets and then priorities on signal instances.

XSDLENVUI

Enable the possibility to communicate with a user-defined UI.

XSIGLOG

Call the `xSignalLog` and `xProcessLog` functions.

XSIGPRIO

Use priorities on signal instances.

XSIGPRSPRIO

Use first priorities on signal instances and then priorities on process instance sets.

XSIMULATORUI

Enable the possibility to communicate with the simulator UI.

XTENV

As XENV but call `xInEnv` at specified times (next event time is out parameter from function `xInEnv`).

XTRACE

Compile with the textual trace enabled.

Code Optimization

XCONST

The majority of the `xIdNode` structs can be made const by defining `CONST` as `const`. This is only possible in applications (not simulations).

XCONST_COMP

This should normally be defined as const if XCONST is const. It is used to introduce const in the component declarations within the `xIdNode` structs.

XNOCONTSIGFUNC

Do not include functions to calculate the expressions in continuous signals. This saves also one function pointer in the `xIdNode` for the states. If this switch is defined, continuous signals cannot be used.

XNOENABCONDFUNC

Do not include functions to calculate the expressions in enabling conditions. This saves also one function pointer in the `xIdNode` for the states. If this switch is defined, enabling conditions cannot be used.

XNOEQTIMERFUNC

Do not include function to compare the parameters of two timers. This saves also one function pointer in the `xIdNode` for the signals. If this switch is defined, timers with parameters cannot be used.

XNOREMOTEVARIDNODE

Do not include xIdNodes for remote variable definitions.

XNOSIGNALIDNODE

Do not include `xSignalIdNodes` for signals and timers.

XNOSTARTUPIDNODE

Do not include `xSignalIdNodes` for start up signals.

XNOUSEOFOBJECTIDENTIFIER

The type Object_identifier and all operations on that type are removed.

XNOUSEOFOCTETBITSTRING

The types Bit_string, Octet, Octet_string and all operations on these types are removed.

XNOUSEOFSERVICE

All data and code needed to handle services are removed.

XNOUSEOFREAL

The type real and all operations on real are removed.

XOPT

Defines XOPTSIGPARA, XOPTDCL, XOPTFPAR, XOPTSTRUCT, XOPTLIT, and XOPTSORT.

For more information see these macros.

XOPTCHAN

Do not include xIdNodes for channels, signal routes, and gates. Information in services and processes about connections to signal routes and gates are also removed.

Note:

If this compilation switch is defined all outputs must either be sent TO a process or the receiver must be possible to calculate during code generation.

XOPTDCL

Do not include xIdNodes for variables.

XOPTFPAR

Do not include xIdNodes for formal parameters.

XOPTLIT

Do not include xIdNodes for literals.

XOPTSIGPARA

Do not include xIdNodes for signal parameters.

XOPTSORT

Do not include xIdNodes for newtypes and syntypes.

XOPTSTRUCT

Do not include xIdNodes for struct components.

XPRSOPT

Optimize memory for process instances. All memory for a process instance can be reused, but signal sending to a stopped process, who's memory has been reused by a new process, cannot be detected. The new process will in this case receive the signal.

XSYNTVAR

If this compilation switch is defined, xVarIdNodes are inserted for the Present components for optional struct components. This feature is only needed by the Validator and by LINK. It should not be defined otherwise.

Definitions of Minor Features

XBREAKBEFORE

Should be used mainly if the MONITOR or GRTRACE switches are defined. It will make the functions and struct components for SDT references available and is also used to expand the macros XAT_FIRST_SYMBOL, XBETWEEN_SYMBOLS, XBETWEEN_SYMBOLS_PRD, XBETWEEN_STMTS, XBETWEEN_STMTS_PRD, XAFTER_VALUE_RET_PRDCALL, and XAT_LAST_SYMBOL to suitable function calls. These functions are used to interrupt a transition between symbols during simulation.

XCASEAFTERPRDLABELS

See XCASELABELS below. The SDL symbols just after an SDL procedure call have to be treated specially, as the symbol number (=case label) for these symbols are used as the restart address for the calling graph. Normally this macro should be defined. If SDL procedure calls are transformed to proper C function calls, and SDL return is translated to a C return, and nextstate in a procedure is NOT translated to a C return (i.e. the process will be hanging in the C function representing the SDL procedure) then it is not necessary to define XCASEAFTERPRDLABELS.

XCASELABELS

The function implementing the behavior of a process, procedure, or service contains one large switch statement with a case label for each SDL symbol in the graph. This switch is used to be able to restart the execution of a process, procedure, or service at any symbol. In an application most of these label can be removed (all except for those symbols that start a transition, i.e. start, input, continuous signal). The macro XCASELABELS should be defined to introduce the case labels for all SDL symbol. This means that XCASELABELS should be defined in a simulation but not in an application.

XCONNECTPM

If XCONNECTPM is defined the SDL simulation will try to connect itself to the postmaster. This is necessary if GR trace (XGRTRACE), communicating simulations (XPMCOMM), or communication with the

TTCN suite (XITEXCOMM) is to be used. The XCONNECTPM feature is normally only used in simulations.

XCOUNTRESETS

Count the number of timers that are removed at a reset operation. This information is used by the textual trace system (XTRACE) to present this information. The information is really only of interest at a stop action when more then one timer might be (implicitly) reset. XCOUNTRESETS should not be defined in an application.

XENVSIGNALLIMIT

This macro is used to determine the number of signals sent to the environment that, during simulation, should be saved in the input port of the env process instance. Such signals can be inspected with the normal monitor commands for viewing of signals. This macro is only of interest in a simulation and has the default value 20.

XERRORSTATE

Insert the data structure to represent an "error" state that can be used if no path is found out from a decision. This should normally be defined if XEDECISION is defined.

XFREESIGNALFUNCS

Insert free functions for each signal, timer, or startup signal that contains a parameter of a type having a free function. These signal free functions can the be used to free allocated data within a signal. This macro should be defined if Master Library is used.

XFREEVARS

Insert free function calls for all variables of a type with free function, just before stop or return actions. This means that free actions are performed on allocated data referred to from variables is before the object ceases to exist. This macro should be defined.

XIDNAMES

This macro is used to determine if the SDL name of an SDL object should be stored in the `xIdNode` for the object. This character string is used for communication with the user in for example the monitor. Normally this macro should not be used in an application. Sometime it might be useful for target debugging to define XIDNAMES, as it is then fairly easy to identify objects by just printing the SDL name from a debugger. On average this seems to cost approximately 5% more memory.

XNRINST

This macro should be defined if process instance numbers are to be maintained. The instance number is the number in the monitor printout `Test:2`, identifying the individual instances of the process instance set Test in this case. XNRINST is normally only used in a simulation.

XOPERRORF

Include the function `xSDLOpError` in `sctsdl.c`. This function is used to print run-time errors in ADT operators.

XPRSSENDER

Store the value of sender also in the `xPrsNode`. The normal place is in the latest received signal. This is only needed in a simulation as sender might be accessed from the monitor system after the transition is completed and the signal has been returned to the pool of available memory.

XREADANDWRITEF

Include the functions for basic Read and Write. This is needed mainly in simulations.

XREMOVETIMERSIG

Allow the removal of timer signals for not-executing PIds. This is needed only in simulations to implement the monitor commands set-timer and reset-timer.

XSIGPATH

If this macro is defined then the functions `xIsPath` and `xFindReceiver` will return the path of signal routes, channels, and gates from the sender to the receiver, as out parameters. This information can then be used in the monitor system, for example, to produce signal logs. This macro should normally not be defined in an application.

XSYMBTLINK

The XSYMBTLINK macro is used to determine if a complete tree should be built from the xIdNodes of the system. If XSYMBTLINK is defined then all xIdNodes contains a `Parent`, a `Suc`, and a `First` pointer. The value of the `Parent` pointer is generated directly into the xIdNodes. `Suc` and `First`, however, are calculated in the `yInit` function by calling the `xInsertIdNode` function. The `Suc` and `First` pointers are needed by the monitor system, but not in an application, i.e. XSYMBTLINK should be defined in a simulation but not in an application.

XTESTF

This macro is used to include or remove test functions for syntype (or newtypes) with range conditions. The `yTest` function is used by the monitor system and by the functions to test index out of bounds in arrays and to test subranges. This means that XTESTF should be defined if the monitor is used or if XERANGE or XEINDEX is defined.

XTRACHANNELSTOENV

When using partitioning of a system a problem during the redirection of channels is that the number of channels going to the environment is not known at code generation time, which means that the size of the data area used for the connections is not known. This problem is solved in two ways.

Either the function handling redirections allocates more memory, which is the default, or the user specifies how many channels that will be redirected (which could be difficult to compute, but will lead to less need of memory).

In the first case (allocation of more memory) the macros:

```
#define XTRACHANNELSTOENV  0
#define XTRACHANNELLIST
```

should be defined like above. This is the standard in scttypes.h. If the user wants to specify the number of channels himself then

```
#define XTRACHANNELSTOENV  10
#define XTRACHANNELLIST    ,0,0,0,0,0,0,0,0,0,0
```

i.e. `XTRACHANNELSTOENV` should be the number of channels, while `XTRACHANNELLIST` should be a list of that many zeros.

XTRACHANNELLIST

See `XTRACHANNELSTOENV` just above.

Static Data, Mainly xIdNodes

XBLO_EXTRAS

All generated struct values for block, block type, and block instance structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xBlockIdStruct` must be updated as well. Normally this macro should be empty.

Example 500 ––

```
#define XBLO_EXTRAS  ,0
```

––

XBLS_EXTRAS

All generated struct values for block substructure structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xBlockSubstIdStruct` must be updated as well. Normally this macro should be empty.

Example 501 ––

```
#define XBLS_EXTRAS  ,0
```

––

XCOMMON_EXTRAS

All generated struct values for `xIdNode` structs contain this macro after the common components. This means that it is possible to insert new components in all xIdNodes by defining this macro. Normally this macro should be empty.

Example 502 ––

To insert a new int component with value 0 the following definition can be used:

```
#define XCOMMON_EXTRAS  ,0
```

––

XLIT_EXTRAS

All generated struct values for literal structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xLiteralIdStruct` must be updated as well. Normally this macro should be empty.

Example 503 ––

```
#define XLIT_EXTRAS  ,0
```

––

XPAC_EXTRAS

All generated struct values for package structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xPackageIdStruct` must be updated as well. Normally this macro should be empty.

Example 504 ––

```
#define XSYS_EXTRAS  ,0
```

––

XPRD_EXTRAS

All generated struct values for procedure structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xPrdIdStruct` must be updated as well. Normally this macro should be empty.

Example 505 ––

```
#define XSYS_EXTRAS  ,0
```

––

XPRS_EXTRAS

```
(PREFIX_PROC_NAME)
```

All generated struct values for process, process type, and process instance structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xPrsIdStruct` must be updated as well.

Example 506 ––

```
#define XPRS_EXTRAS(PREFIX_PROC_NAME) \
   ,XCAT(PREFIX_PROC_NAME,_STACKSIZE)
```

––

XSIG_EXTRAS

All generated struct values for signal, timer, RPC_signal, startup signal structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xSignalIdStruct` must be updated as well. Normally this macro should be empty.

Example 507 ––

```
#define XSIG_EXTRAS  ,0
```

––

XSPA_EXTRAS

All generated struct values for signal parameter structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xVarIdStruct` must be updated as well (Note that variables, formal parameters, signal parameters, and struct components are all handled in `xVarIdStruct`.) Normally this macro should be empty.

Example 508 ––

```
#define XSPA_EXTRAS  ,0
```

––

XSRT_EXTRAS

All generated struct values for newtype and syntype structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xSortIdStruct` must be updated as well. Normally this macro should be empty.

Example 509 ––

```
#define XSRT_EXTRAS  ,0
```

––

XSRV_EXTRAS

All generated struct values for service, service type, and service instance structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xSrvIdStruct` must be updated as well. Normally this macro should be empty.

Example 510

```
#define XSRV_EXTRAS  ,0
```

XSTA_EXTRAS

All generated struct values for state structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xStateIdStruct` must be updated as well. Normally this macro should be empty.

Example 511

```
#define XSTA_EXTRAS  ,0
```

XSYS_EXTRAS

All generated struct values for system, system type, and system instance structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xSystemIdStruct` must be updated as well. Normally this macro should be empty.

Example 512

```
#define XSYS_EXTRAS  ,0
```

XSYSTEMVARS

This macro gives the possibility to introduce global variables declared in the beginning of the C file containing the implementation of the SDL system unit.

XSYSTEMVARS_H

If extern definitions are needed for the data declared in XSYSTEMVARS, this is the place to introduce it. These definitions will be present in the `.h` file for the system unit (if separate generation is used).

XVAR_EXTRAS

All generated struct values for variables, formal parameters, and struct components structs contain this macro last in the struct. By defining this macro new components can be inserted. Note that the type `xVarIdStruct` must be updated as well (Note that signal parameters also uses the type `xVarIdStruct`). Normally this macro should be empty.

Example 513

```
   #define XVAR_EXTRAS  ,0
```

Data in Processes, Procedures and Services

PROCEDURE_VARS

The struct components that are needed for each procedure instance. Example: state.

PROCESS_VARS

The struct components that are needed for each process instance. Example: state, parent, offspring, self, sender, inputport.

SERVICE_VARS

The struct components that are needed for each service instance. Example: state

YGLOBALPRD_YVARP

This macro is used to declare the `yVarP` pointer (which is a pointer to the `yVDef` struct for the process) in a procedure defined outside of a process. As a global procedure never can access process local data, it is suitable to let `yVarP` be a pointer to a struct only containing the components defined in the macro PROCESS_VARS.

YGLOBALSRV_YVARP

This macro is used to declare the `yVarP` pointer (which is a pointer to the `yVDef` struct for the process) in a service type defined outside of a process. As a global service type never can access process local data, it is suitable to let `yVarP` be a pointer to a struct only containing the components defined in the macro PROCESS_VARS.

YPAD_TEMP_VARS

Local variables in the PAD function for a process or service. Example: temporary variables needed for outputs, create actions.

YPAD_YSVARP

Declaration of the `ySVarP` pointer used to refer to the received signal. Normally `ySVarP` is `void *`.

YPAD_YVARP

```
    (VDEF_TYPE)
```

This macro is used within a process and in a service defined within a process. It should be expanded to a declaration of `yVarP`, which is the pointer that is used to access SDL variables in the process. `yVarP` should be of type `VDEF_TYPE *`, where `VDEF_TYPE` is the type of the `yVDef` struct for the process. If the pointer to the `yVDef` struct is passed as parameter to the PAD function, `yVarP` can be assigned its correct value already in the declaration.

YPRD_TEMP_VARS

Local variables in the function implementing the behavior of an SDL procedure.

YPRD_YVARP

```
    (VDEF_TYPE)
```

This macro is used within a procedure defined in a process. It should be expanded to a declaration of `yVarP`, which is the pointer that is used to access SDL variables in the process. `yVarP` should be of type `VDEF_TYPE *`, where `VDEF_TYPE` is the type of the `yVDef` struct for the process. If the pointer to the `yVDef` struct is passed as parameter to the procedure function, `yVarP` can be assigned its correct value already in the declaration.

Some Macro Used Within PAD Functions

BEGIN_PAD

```
(VDEF_TYPE)
```

BEGIN_PAD is a macro that can be used to insert code that is executed in the beginning of the PAD functions. `VDEF_TYPE` is the yVDef `type` for the process.

BEGIN_START_TRANSITION

```
(STARTUP_PAR_TYPE)
```

This macro can be used to introduce code that is executed at the beginning of the start transition. `STARTUP_PAR_TYPE` is the `yPDef` struct for the startup signal for this process.

CALL_SERVICE

This macro is used in the PAD function of a process that contains services. It should be expanded to a call to PAD function for the service that should execute the next transition (`ActiveSrv`).

CALL_SUPER_PAD_START

```
(PAD)
```

During the start transition of a process all inherited PAD functions up to and including the PAD function containing the START symbol have to be called. The reason is to initialize all variables defined in the process. This macro is used to perform a call to the inherited PAD function (the macro parameter PAD). Usually this macro is expanded to something like:

```
yVarP->RestartPAD = PAD; PAD(VarP);
```

followed by either a `return` or a `goto NewTransition` depending on execution model.

CALL_SUPER_PRD_START

```
(PRD, THISPRD)
```

This macro is used in the same way as CALL_SUPER_PAD_START (see above) but for the start transition in a procedure. `THISPRD` is the executing procedure function, while `PRD` is the inherited procedure function.

CALL_SUPER_SRV_START

`(PAD)`

This macro is used in the same way as CALL_SUPER_PAD_START (see above) but for the start transition in a service. `PAD` is the inherited PAD function.

LOOP_LABEL

The LOOP_LABEL macro should be used to form the loop from a next-state operation to the next input operation necessary in the OS where OS tasks does not perform return at end of transition (most commercial OS). This macro is also suitable to handle free on received signals and the treatment of the save queue. In an OS where SDL nextstate is implemented using a C `return` (the Master Library for example) the LOOP_LABEL macro is usually empty.

LOOP_LABEL_PRD

Similar to LOOP_LABEL but used in procedures with states.

LOOP_LABEL_PRD_NOSTATE

Similar to LOOP_LABEL but used in procedures without states. This macro is in many circumstances expanded to nothing.

LOOP_LABEL_SERVICEDECOMP

Similar to LOOP_LABEL but used in the PAD function for a process containing services.

SDL_OFFSPRING

Should return the value of offspring.

SDL_PARENT

Should return the value of parent.

SDL_SELF

Should return the value of self.

SDL_SENDER

Should return the value of sender.

START_SERVICES

This macro is used in the PAD function of a process that contains services. It should be expanded in such a way that the start transitions for all of the services are executed.

XEND_PRD

This is a macro generated at the end of a function that represents the behavior of a procedure. It needs not to be expanded to anything. To define it as

```
return (xbool)0;
```

might remove a compiler warning that the end of a value returning function might be reached.

XPRSNODE

Should usually be expanded to the type `xPrsNode`.

XNAMENODE

How to reach the `xPrsIdNode` from a PAD function. Normally this is `yVarP->NameNode`.

XNAMENODE_PRD

How to reach the `xPrdIdNode` from a PRD function. Normally this is `yPrdVarP->NameNode`.

XNAMENODE_SRV

How to reach the `xSrvIdNode` from a PAD function. Normally this is `ySrvVarP->NameNode`.

YPAD_FUNCTION

```
(PAD)
```

The function heading of the PAD function given as parameter.

YPAD_PROTOTYPE

```
(PAD)
```

The function prototype of the PAD function given as parameter.

YPRD_FUNCTION

`(PRD)`

The function heading of the PRD function given as parameter.

YPRD_PROTOTYPE

`(PRD)`

The function prototype of the PRD function given as parameter.

yInit Function

BEGIN_YINIT

This macro is placed in the beginning of the `yInit` function in the file containing code for the system. It can be expanded to variable declarations and initialization code.

XPROCESSDEF_C

```
(PROC_NAME, PROC_NAME_STRING, PREFIX_PROC_NAME,
PAD_FUNCTION, VDEF_TYPE)
```

This macro can be used to introduce code for each process instance set in the system.

Parameters:

- `PROC_NAME`
  the name of the process without prefix.
- `PROC_NAME_STRING`
  the name of the process as a character string.
- `PREFIX_PROC_NAME`
  the name of the process with prefix.
- `PAD_FUNCTION`
  the PAD function for this process instance set.
- `VDEF_TYPE`
  the `yVDef` struct for this process.

XPROCESSDEF_H

```
(PROC_NAME, PROC_NAME_STRING, PREFIX_PROC_NAME,
PAD_FUNCTION, VDEF_TYPE)
```

This macro can be used to introduce extern declaration (placed in the proper `.h` file) for each process instance set in the system.

Parameters:

- `PROC_NAME`
  the name of the process without prefix.
- `PROC_NAME_STRING`
  the name of the process as a character string.
- `PREFIX_PROC_NAME`
  the name of the process with prefix.
- `PAD_FUNCTION`
  the PAD function for this process instance set.
- `VDEF_TYPE`
  the `yVDef` struct for this process.

xInsertIdNode

In the `yInit` function the function `xInsertIdNode` is called for each `IdNode`. In an application this is not necessary, and `xInsertIdNode` can be defined as

```
#define xInsertIdNode(Node)
```

The function `xInsertIdNode` is needed if XSYMBTLINK, XCOVERAGE, or XMONITOR is defined.

YINIT_TEMP_VARS

This macro is placed in all yInit functions and can be expanded to local variables needed within the `yInit` function.

Implementation of Signals and Output

ALLOC_SIGNAL

ALLOC_SIGNAL_PAR

```
(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)
```

These macros are used to allocate a data area for a signal to be sent. ALLOC_SIGNAL is used if the signal has no parameters, while ALLOC_SIGNAL_PAR is used if the signal has parameters. The resulting data area should be reference by the variable mentioned by the macro OUTSIGNAL_DATA_PTR (see below).

Parameters:

- `SIG_NAME`
  the name of the signal without prefix.
- `SIG_IDNODE`
  the `xSignalIdNode` of the signal.
- `RECEIVER`
  the receiver given in the TO clause, or calculated. In a NO_TO output, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_TYPE`
  the yPDef `type` of the signal. If the signal has no parameters this macro parameter is XSIGNALHEADERTYPE (see below).

INSIGNAL_NAME

This macro should be expanded to the identification of the currently received signal. It is used to distinguish between signals when several signal is enumerated in the same input symbol.

OUTSIGNAL_DATA_PTR

This should be the pointer referring to be signal data area while building the signal during an output. It should be assigned its value in ALLOC_SIGNAL or ALLOC_SIGNAL_PAR, and will then be used during assignment of signal parameters and in the SDL_2OUTPUT macro just below.

SDL_2OUTPUT

SDL_2OUTPUT_NO_TO

SDL_2OUTPUT_COMPUTED_TO

SDL_ALT2OUTPUT

SDL_ALT2OUTPUT_NO_TO

SDL_ALT2OUTPUT_COMPUTED_TO

```
(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER,
SIG_PAR_SIZE, SIG_NAME_STRING)
```

These six macros are used to send the signal created in ALLOC_SIGNAL or ALLOC_SIGNAL_PAR. The SDL_ALT versions of the macros are used if the directive `/*#ALT*/` has been given in the output. The version without suffix is used for an output TO, while the suffix _COMPUTED_TO is used for an output without to but it was possible to compute the receiver during code generation time. The suffix _NO_TO indicates an output without to, where the receiver cannot be calculated during code generation time.

Parameters:

- `PRIO`
  the priority of the signal specified in a #PRIO directive.
- `VIA`
  the via list given in the output.
- `SIG_NAME`
  the name of the signal without prefix
- `SIG_IDNODE`
  the `xSignalIdNode` for the signal.
- `RECEIVER`
  the receiver given in the TO clause, or calculated. In a NO_TO output, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_SIZE`
  the size of the `yPDef` struct of the signal. If signal without parameters `SIG_PAR_SIZE` is 0.
- `SIG_NAME_STRING`
  the name of the signal as a character string.

SDL_THIS

In an output TO THIS in SDL, the RECEIVER parameter in the ALLOC_SIGNAL and SDL_2OUTPUT macros discussed above will become SDL_THIS.

SIGCODE

```
(P)
```

This macro makes it possible to store a signal code (signal number) in the `xSignalIdNode` for a signal. The macro parameter `P` is the signal name without prefix.

SIGNAL_ALLOC_ERROR

This macro is inserted after the ALLOC_SIGNAL macro and the assignment of parameter values to the signal. It can be used to test if the alloc was successful or not.

SIGNAL_ALLOC_ERROR_END

This macro is inserted after the SDL_2OUTPUT macro.

SIGNAL_NAME

```
(SIG_NAME, SIG_IDNODE)
```

This macro should be expanded to an identification of the signal given as parameter. Normally the identification is either the `xSignalIdNode` for the signal or an `int` value. If the id is an `int` value it is suitable to insert defines of type #define signal_name number. A file containing such defines can be generated using the Generate Signal Numbers feature in Cadvanced/Cbasic.

Parameters:

- `SIG_NAME`
  the name of the signal without parameters
- `SIG_IDNODE`
  the `xSignalIdNode` for the signal.

SIGNAL_VARS

The struct components that are needed for each signal instance. Example: sender, receiver, signal type.

TO_PROCESS

```
(PROC_NAME, PROC_IDNODE)
```

This macro is used as RECEIVER in the ALLOC_SIGNAL and SDL_2OUTPUT macros if the signal is sent to a process instance set in SDL.

Parameters:

- `PROC_NAME`
  the name of the receiving process without prefix.
- `PROC_IDNODE`
  the `xPrsIdNode` of the receiving process.

TRANSFER_SIGNAL

TRANSFER_SIGNAL_PAR

```
(SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)
```

These macros are used as alternative for the ALLOC_SIGNAL macros (see these macros above) if the directive `#TRANSFER` if given in the output.

- `SIG_NAME`
  the name of the signal without prefix.
- `SIG_IDNODE`
  the `xSignalIdNode` of the signal.
- `RECEIVER`
  the receiver given in the TO clause, or calculated. In a NO_TO output, `RECEIVER` is `xNotDefPId`.
- `SIG_PAR_TYPE`
  the `yPDef` type of the signal. If the signal has no parameters this macro parameter is XSIGNALHEADERTYPE (see below).

XNONE_SIGNAL

The representation for a none signal.

XSIGNALHEADERTYPE

This macro is used to indicate a `yPDef` struct for a signal without parameters. Such a signal has no generated `yPDef` struct. It is suitable to let

XSIGNALHEADERTYPE be the name of a struct just containing the components in SIGNAL_VARS.

XSIGTYPE

Depending on the representation of the signal type that is used (`xSignalIdNode` or `int`) this macro should either be `xSignalIdNode` or `int`.

Implementation of RPC

ALLOC_REPLY_SIGNAL

ALLOC_REPLY_SIGNAL_PAR

ALLOC_REPLY_SIGNAL_PRD

ALLOC_REPLY_SIGNAL_PRD_PAR

```
   (SIG_NAME, SIG_IDNODE, RECEIVER, SIG_PAR_TYPE)
```

These macros are used to allocate the Reply signal in the signal exchange in an RPC. The suffix _PAR is used if the reply signal contains parameters. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

Parameters:

- `SIG_NAME`
  the reply signal name without prefix.
- `SIG_IDNODE`
  the `xSignalIdNode` for the reply signal.
- `RECEIVER`
  the receiver of the reply signal. The macro XRPC_SENDER_IN_ALLOC or XRPC_SENDER_IN_ALLOC_PRD are used as actual parameter. The suffix _PRD is used if the implicit RPC transition is part of a procedure.
- `SIG_PAR_TYPE`
  the `yPDef` type for the reply signal. If the reply signal does not contain any parameters the macro name XSIGNALHEADERTYPE is generated as actual parameter.

REPLYSIGNAL_DATA_PTR

REPLYSIGNAL_DATA_PTR_PRD

This should be a reference to the data area for the reply signal that is allocated in the ALLOC_REPLY_SIGNAL macro. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

SDL_RPCWAIT_NEXTSTATE

SDL_RPCWAIT_NEXTSTATE_PRD

```
(PREPLY_IDNODE, PREPLY_NAME, RESTARTADDR)
```

These macros are used to implement the implicit nextstate operation in the caller of an RPC. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

Parameters:

- `PREPLY_IDNODE`
  the `xSignalIdNode` for the reply signal.
- `PREPLY_NAME`
  the name without prefix for the reply signal.
- `RESTARTADDR`
  the restart address (symbol number) for the implicit input of the reply signal.

SDL_2OUTPUT_RPC_CALL

```
(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER,
SIG_PAR_SIZE, SIG_NAME_STRING)
```

Send the call signal of an RPC.

Parameters:

- `PRIO`
  priority of signal.
- `VIA`
  the via list, which in this case always is `(xIdNode *)0`, i.e. no via list.
- `SIG_NAME`
  the RPC call signal name without prefix.
- `SIG_IDNODE`
  the `xSignalIdNode` for the RPC call signal.

- `RECEIVER`
  the receiver of the call signal. This is either expressed as an ordinary TO-expression or using the macro XGETEXPORTINGPRS (see below) in case of no explicit receiver specified in the call.

- `SIG_PAR_SIZE`
  the size of the `yPDef` struct for the call signal. If the call signal has no parameters this parameter will be 0.

- `SIG_NAME_STRING`
  the name of the RPC call signal as a character string.

SDL_2OUTPUT_RPC_REPLY

SDL_2OUTPUT_RPC_REPLY_PRD

```
(PRIO, VIA, SIG_NAME, SIG_IDNODE, RECEIVER,
SIG_PAR_SIZE, SIG_NAME_STRING)
```

These macros are used to send the RPC reply signal. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

Parameters:

- `PRIO`
  priority of signal.

- `VIA`
  the via list, which in this case always is `(xIdNode *)0`, i.e. no via list.

- `SIG_NAME`
  the RPC reply signal name without prefix.

- `SIG_IDNODE`
  the `xSignalIdNode` for the RPC reply signal.

- `RECEIVER`
  the receiver of the reply signal. This is expressed using the macro XRPC_SENDER_IN_OUTPUT or XRPC_SENDER_IN_OUTPUT_PRD.

- `SIG_PAR_SIZE`
  the size of the yPDef struct for the reply signal. If the reply signal has no parameters this parameter will be 0.

- `SIG_NAME_STRING`
  the name of the RPC reply signal as a character string.

XGETEXPORTINGPRS

```
(REMOTENODE)
```

This macro should be expanded to an expression that given the remote procedure given as actual macro parameter (more exactly the `IdNode` for the remote procedure), returns one possible exporter of this remote procedure. Usually this macro is expanded to a call of the library function `xGetExportingPrs`.

XRPC_REPLY_INPUT

XRPC_REPLY_INPUT_PRD

Macros that can be used for special processing needed to receive an RPC reply signal. The macros are usually expanded to nothing.

XRPC_SAVE_SENDER

XRPC_SAVE_SENDER_PRD

These macros can be used to save the sender of a received RPC call signal, for further use when the reply signal is to be sent. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

XRPC_SENDER_IN_ALLOC

XRPC_SENDER_IN_ALLOC_PRD

These macros are used to obtain the receiver of the reply signal (from the sender of the call signal) in the ALLOC_REPLY_SIGNAL macros. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

XRPC_SENDER_IN_OUTPUT

XRPC_SENDER_IN_OUTPUT_PRD

These macros are used to obtain the receiver of the reply signal (from the sender of the call signal) in the SDL_2OUTPUT_RPC_REPLY macros. The suffix _PRD is used if the implicit RPC transition is part of a procedure.

XRPC_WAIT_STATE

The state number used for a RPC wait state. XRPC_WAIT_STATE is usually defined as -3.

Implementation of View and Import

XGETEXPORTADDR

`(REMOTENODE, EXPORTER, IS_DEF_EXPORTER)`

This macro should be expanded to an expression that returns the address of the exported variable. Usually the function `xGetExportAddr` is called.

Parameters:

- `REMOTENODE`
  the IdNode for the remote variable.

- `EXPORTER`
  the value of the optional PId expression. If no PId expression is given this parameter is SDL_NULL.

- `IS_DEF_EXPORTER`
  has the value `(xbool)1` if a PId expression was found in the import statement, otherwise it is `(xbool)0`.

SDL_VIEW

```
(PID_EXPR, HAS_EXPR, VAR_NAME_STRING, REVEALED_LIST,
SORT_SIZE)
```

This macro should be expanded to an expression that returns the address of the viewed variable. Usually the function SDL_View is called.

Parameters:

- `PID_EXPR`
  the value of the optional PId expression. If no PId expression is given this parameter is SDL_NULL.

- `HAS_EXPR`
  has the value `(xbool)1` if a PId expression was found in the view statement, otherwise it is `(xbool)0`.

- `VAR_NAME_STRING`
  the name of the viewed variable as a character string.

- `REVEALED_LIST`
  the list of the revealed variables.

- `SORT_SIZE`
  the size of the sort of the revealed variable.

Implementation of Static and Dynamic Create and Stop

ALLOC_STARTUP

ALLOC_STARTUP_PAR

```
(PROC_NAME, STARTUP_IDNODE, STARTUP_PAR_TYPE)
```

Allocate the data area for a startup signal and let the pointer mentioned in the macro STARTUP_DATA_PTR refer to this data area. The suffix _PAR is used if the startup signal contains parameters.

Parameters:

- `PROC_NAME`
  the name without prefix for the created process.
- `STARTUP_IDNODE`
  the `xSignalIdNode` for the startup signal of the created process.
- `STARTUP_PAR_TYPE`
  the `yPDef` for the startup signal of the created process.

ALLOC_STARTUP_THIS

Allocate the data area for a startup signal and let the pointer mentioned in the macro STARTUP_DATA_PTR refer to this data area. This macro is used in a create THIS operation.

INIT_PROCESS_TYPE

```
(PROC_NAME, PREFIX_PROC_NAME, PROC_IDNODE,
PROC_NAME_STRING, MAX_NO_OF_INST, STATIC_INST,
VDEF_TYPE, PRIO, PAD_FUNCTION)
```

This macro will be call once for each process instance set in the yInit function. It should be used to initiated common features for all instances of a process instance set.

Parameters:

- `PROC_NAME`
  the name without prefix for the process instance set.
- `PREFIX_PROC_NAME`
  the name with prefix for the process instance set.
- `PROC_IDNODE`
  the `xPrsIdNode` for the process instance set.

- `PROC_NAME_STRING`
  the name as character string for the process instance set.
- `MAX_NO_OF_INST`
  the maximum number of instances of this process instance set.
- `STATIC_INST`
  the number of static instances of this process instance set.
- `VDEF_TYPE`
  the `yVDef` type for this process instance set.
- `PRIO`
  the priority for process instance set.
- `PAD_FUNCTION`
  the PAD for this process instance set.

SDL_CREATE

`(PROC_NAME, PROC_IDNODE, PROC_NAME_STRING)`

This macro is used to create (a create action) a process instance.

Parameters:

- `PROC_NAME`
  the name without prefix for the process instance set.
- `PROC_IDNODE`
  the `xPrsIdNode` for the process instance set.
- `PROC_NAME_STRING`
  he name as character string for the process instance set.

SDL_CREATE_THIS

This macro is used to implement create this.

SDL_STATIC_CREATE

```
(PROC_NAME, PREFIX_PROC_NAME, PROC_IDNODE,
PROC_NAME_STRING, STARTUP_IDNODE, STARTUP_PAR_TYPE,
VDEF_TYPE, PRIO, PAD_FUNCTION, BLOCK_INST_NUMBER)
```

This macro is called in the yInit function once for each static process instances that should be created of a process instance set.

Parameters:

- `PROC_NAME`
  the name without prefix for the process instance set.
- `PREFIX_PROC_NAME`
  the name with prefix for the process instance set.
- `PROC_IDNODE`
  the `xPrsIdNode` for the process instance set.
- `PROC_NAME_STRING`
  the name as character string for the process instance set.
- `STARTUP_IDNODE`
  the `xSignalIdNode` for the startup signal for the process instance set.
- `STARTUP_PAR_TYPE`
  the `yPDef` type for the startup signal for the process instance set.
- `VDEF_TYPE`
  the `yVDef` type for the process instance set.
- `PRIO`
  the priority for the process instance set.
- `PAD_FUNCTION`
  the PAD function for the process instance set.
- `BLOCK_INST_NUMBER`
  if this process instance set is part if a block instance set then this macro is the block instance number for the block instance set that this process belongs to. Otherwise this macro parameter is 1.

SDL_STOP

This macro is used to implement the SDL operation stop (both in processes and in services).

STARTUP_ALLOC_ERROR

This macro is inserted after the ALLOC_STARTUP macro and the assignment of parameter values to the signal. It can be used to test if the alloc was successful or not.

STARTUP_ALLOC_ERROR_END

This macro is inserted after the SDL_CREATE macro.

STARTUP_DATA_PTR

This macro should be expanded to a temporary variable used to store a reference to the startup signal data area. It should be assigned in the ALLOC_STARTUP macro and will be used to assign the actual signal parameters (the fpar values) to the startup signal.

STARTUP_VARS

This macro can be used to insert additional general components in the startup signals. In all startup signal yPDef structs SIGNAL_VARS will be followed by STARTUP_VARS.

Implementation of Timers, Timer Operations and Now

ALLOC_TIMER_SIGNAL_PAR

```
(TIMER_NAME, TIMER_IDNODE, TIMER_PAR_TYPE)
```

Allocate a data area for the timer signal with parameters.

Parameters:

- `TIMER_NAME`
  the name without prefix of the timer.
- `TIMER_IDNODE`
  the `xSignalIdNode` for the timer.
- `TIMER_PAR_TYPE`
  the `yPDef` for the timer.

DEF_TIMER_VAR

DEF_TIMER_VAR_PARA

```
(TIMER_VAR)
```

There will be one application of this macro in the `yVDef` type for the process for each timer declaration the process contains. These declarations can be used to introduce components (timer variables) in the `yVDef` struct to track timers. The parameter `TIMER_VAR` is a suitable name for such a variable. The suffix _PARA is used if the timer has parameters.

INIT_TIMER_VAR

INIT_TIMER_VAR_PARA

    (TIMER_VAR)

These macros will be inserted in start transitions, during initialization of process variables. This makes it possible to initialize the timer variables that might be inserted in the DEF_TIMER_VAR macro. The parameter `TIMER_VAR` is the name for such a variable. The suffix _PARA is used if the timer has parameters.

INPUT_TIMER_VAR

INPUT_TIMER_VAR_PARA

    (TIMER_VAR)

These macros will be inserted at an input operation on a timer signal. This makes it possible to update the timer variables that might be inserted in the DEF_TIMER_VAR macro. The parameter `TIMER_VAR` is the name for such a variable. The suffix _PARA is used if the timer has parameters. Note that if a timer signal is received in an input * statement, no INPUT_TIMER_VAR will be present in this case.

RELEASE_TIMER_VAR

RELEASE_TIMER_VAR_PARA

    (TIMER_VAR)

These macros will be inserted at a stop. This makes it possible to perform cleaning up of the timer variables that might be inserted in the DEF_TIMER_VAR macro. The parameter `TIMER_VAR` is the name for such a variable. The suffix _PARA is used if the timer has parameters.

SDL_ACTIVE

    (TIMER_NAME, TIMER_IDNODE, TIMER_VAR)

This macro is used to implement the SDL operation active on a timer. Note that active on timers with parameters is not implemented in the Cadvanced/Cbasic SDL to C Compiler.

Parameters:

- `TIMER_NAME`
  the name without prefix of the timer.
- `TIMER_IDNODE`
  the xSignalIdNode for the timer.
- `TIMER_VAR`
  the timer variable that might be inserted in the macro DEF_TIMER_VAR.

SDL_NOW

This is the implementation of now in SDL.

SDL_RESET

```
(TIMER_NAME, TIMER_IDNODE, TIMER_VAR,
TIMER_NAME_STRING)
```

This macro is used to implement the SDL operation reset on a timer without parameters.

Parameters:

- `TIMER_NAME`
  the name without prefix of the timer.
- `TIMER_IDNODE`
  the `xSignalIdNode` for the timer.
- `TIMER_VAR`
  the timer variable that might be inserted in the macro DEF_TIMER_VAR.
- `TIMER_NAME_STRING`
  the name of the timer as a character string.

SDL_RESET_WITH_PARA

```
(EQ_FUNC, TIMER_VAR, TIMER_NAME_STRING)
```

This macro is used to implement the SDL operation reset on a timer with parameters. Before this macro a timer signal with the timer parameters in the reset operation is created.

Parameters:

- `EQ_FUNC`
  the name of the generated equal function that can test if two timer instance are equal or not.
- `TIMER_VAR`
  the timer variable that might be inserted in the macro DEF_TIMER_VAR.
- `TIMER_NAME_STRING`
  the name of the timer as a character string.

SDL_SET

```
(TIME_EXPR, TIMER_NAME, TIMER_IDNODE, TIMER_VAR,
TIMER_NAME_STRING)
```

SDL_SET_WITH_PARA

```
(TIME_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR,
TIMER_NAME_STRING)
```

SDL_SET_DUR

```
(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_VAR, TIMER_NAME_STRING)
```

SDL_SET_DUR_WITH_PARA

```
(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR,
TIMER_NAME_STRING)
```

SDL_SET_TICKS

```
(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_VAR, TIMER_NAME_STRING)
```

SDL_SET_TICKS_WITH_PARA

```
(TIME_EXPR, DUR_EXPR, TIMER_NAME, TIMER_IDNODE,
TIMER_PAR_TYPE, EQ_FUNC, TIMER_VAR,
TIMER_NAME_STRING)
```

These six SDL_SET macros are used to implement the SDL operation set on a timer. The suffix _WITH_PARA indicates the set of a timer with parameters. In this case the SDL_SET macro is preceded by an ALLOC_TIMER_SIGNAL_PAR macro call, plus the assignment of the timer parameters. The suffix _DUR is used if the time value in the set operation is expressed as:

```
now + expression
```

In this case both the time value and the duration value (the expression above) is available as macro parameter. The suffix _TICKS is used if the time value in the set operation is expressed as:

```
now + TICKS(...)
```

where TICKS is an operator returning a duration value. In this case both the time value and the duration value (the TICKS expression above) is available as macro parameter.

Parameters:

- `TIME_EXPR`
  the time expression.
- `DUR_EXPR`
  the duration expression (only in _DUR and _TICKS).
- `TIMER_NAME`
  the timer name without prefix.
- `TIMER_IDNODE`
  the `xSignalIdNode` for the timer.
- `TIMER_PAR_TYPE`
  the `yPDef` struct for the timer (only in _WITH_PARA)
- `EQ_FUNC`
  the function that can be used to test if two timers have the same parameter values (only in _WITH_PARA).
- `TIMER_VAR`
  the name of the timer variable that might be introduced in the macro DEF_TIMER_VAR.
- `TIMER_NAME_STRING`
  the name of the timer as a character string.

TIMER_DATA_PTR

This should be the pointer referring to be timer data area while building the timer. It should be assigned its value in ALLOC_TIMER_SIGNAL_PAR, and will then be used during assignment of signal parameters and in the SDL_SET macro

TIMER_SIGNAL_ALLOC_ERROR

This macro is inserted after the ALLOC_TIMER_SIGNAL_PAR macro and the assignment of parameter values to the timer. It can be used to test if the alloc was successful or not.

TIMER_SIGNAL_ALLOC_ERROR_END

This macro is inserted after the SDL_SET macro.

TIMER_VARS

The struct components that are needed for each timer instance. Example: sender, receiver, timer type.

As timers are signals as well, after the timer signal has been sent, TIMER_VARS has to be identical to SIGNAL_VARS, except that new component may be add last in TIMER_VARS.

XTIMERHEADERTYPE

This macro is used to indicate a `yPDef` struct for a timer without parameters. Such a timer has no generated `yPDef` struct. It is suitable to let XTIMERHEADERTYPE be the name of a struct just containing the components in TIMER_VARS.

Implementation of Call and Return

ALLOC_PROCEDURE

`(PROC_NAME, PROC_IDNODE, VAR_SIZE)`

Allocate a data area (`yVDef`) for the called procedure.

Parameters:

- `PROC_NAME`
  the name of procedure with prefix.
- `PROC_IDNODE`
  the `xPrdIdNode` of the called procedure
- `VAR_SIZE`
  the size of the `yVDef` struct for the procedure.

ALLOC_THIS_PROCEDURE

Allocate a data area (`yVDef`) for a procedure when call THIS is used.

ALLOC_VIRT_PROCEDURE

```
(PROC_IDNODE)
```

Allocate a data area (`yVDef`) for the called procedure when calling a virtual procedure. The `PROC_IDNODE` parameter is the `xPrdIdNode` for the call procedure.

CALL_PROCEDURE

CALL_PROCEDURE_IN_PRD

```
(PROC_NAME, PROC_IDNODE, LEVELS, RESTARTADDR)
```

These macros are used to implement a call operation in SDL. The `yVDef` struct has been allocated earlier (in ALLOC_PROCEDURE) and the actual parameters have been assigned to components in this struct. The suffix _IN_PRD indicates that the procedure call is made in a procedure.

Parameters:

- `PROC_NAME`
  the name of procedure with prefix, which is the same as the name of the C function representing the behavior of the procedure.
- `PROC_IDNODE`
  the `xPrdIdNode` of the called procedure.
- `LEVELS`
  the scope level between the caller and the called procedure.
- `RESTARTADDR`
  the restart address the symbol number for the symbol after the procedure call.

CALL_PROCEDURE_STARTUP

CALL_PROCEDURE_STARTUP_SRV

These two macros are only of interest if the PAD functions are left via a return at the end of transitions. In that case any outstanding procedure must be restarted when the process becomes active again.

CALL_THIS_PROCEDURE

`(RESTARTADDR)`

This macro is used to implement a call THIS operation in SDL. `RESTARTADDR` is the restart address the symbol number for the symbol after the procedure call.

CALL_VIRT_PROCEDURE

CALL_VIRT_PROCEDURE_IN_PRD

`(PROC_IDNODE, LEVELS, RESTARTADDR)`

These macros are used to implement a call operation on a virtual procedure in SDL. The `yVDef` struct has been allocated earlier (in ALLOC_VIRT_PROCEDURE) and the actual parameters have been assigned to components in this struct. The suffix _IN_PRD indicates that the procedure call is made in a procedure.

Parameters:

- `PROC_IDNODE`
  the xPrdIdNode of the called procedure.
- `LEVELS`
  the scope level between the caller and the called procedure.
- `RESTARTADDR`
  the restart address the symbol number for the symbol after the procedure call.

PROCEDURE_ALLOC_ERROR

This macro is inserted after the ALLOC_PROCEDURE macro and the assignment of parameter values to the procedure parameters. It can be used to test if the alloc was successful or not.

PROCEDURE_ALLOC_ERROR_END

This macro is inserted after the CALL_PROCEDURE macro.

PROC_DATA_PTR

This macro should be expanded to a temporary variable used to store a reference to the procedure data area. It should be assigned in the ALLOC_PROCEDURE macro and will be used to assign the actual procedure parameters (the fpar values).

SDL_RETURN

The implementation of return in SDL.

XNOPROCATSTARTUP

If this macro is defined then all the code discussed for the macro CALL_PROCEDURE_STARTUP (just above) is removed.

Implementation of Join

Joins in SDL are normally implemented as goto:s in C, but in one case a more complex implementation is needed. This is when the label, mentioned in the join, is in a super type.

XJOIN_SUPER_PRS

`(RESTARTADDR,RESTARTPAD)`

XJOIN_SUPER_PRD

`(RESTARTADDR,RESTARTPRD)`

XJOIN_SUPER_SRV

`(RESTARTADDR,RESTARTSRV)`

These macros represent join to super type in processes, procedures, and services, in that order.

Parameters:

- `RESTARTADDR`
  The restart address in the super type.
- `RESTARTPAD, RESTARTPRD, RESTARTSRV`
  The PAD function for the super type.

Implementation of State and Nextstate

Note:

Implicit nextstate operations in RPC calls are treated in the RPC section.

ASTERISK_STATE

The state number for an asterisk state. ASTERISK_STATE is usually defined as -1.

ERROR_STATE

The state number used for the error state. ERROR_STATE is usually defined as -2.

START_STATE

The state number for the start state. START_STATE should be defined as 0.

START_STATE_PRD

The state number for the start state in a procedure. START_STATE_PRD should be defined as 0.

SDL_NEXTSTATE

```
    (STATE_NAME, PREFIX_STATE_NAME, STATE_NAME_STRING)
```

Nextstate operation (in process or service) of the given state.

Parameters:

- `STATE_NAME`
  the name without prefix of the state.
- `PREFIX_STATE_NAME`
  the name with prefix for the state. This identifier is defined as a suitable state number in generated code and is usually used as the representation of the state.
- `STATE_NAME_STRING`
  the name of the state as a character string.

SDL_DASH_NEXTSTATE

Dash nextstate operation in a process.

SDL_DASH_NEXTSTATE_SRV

Dash nextstate operation in a service.

SDL_NEXTSTATE_PRD

```
    (STATE_NAME, PREFIX_STATE_NAME, STATE_NAME_STRING)
```

Nextstate operation (in procedure) of the given state.

Parameters:

- `STATE_NAME`
  the representation of the state.
- `PREFIX_STATE_NAME`
  the name with prefix for the state. This identifier is defined as a suitable state number in generated code and is usually used as the representation of the state.
- `STATE_NAME_STRING`
  the name of the state as a character string.

SDL_DASH_NEXTSTATE_PRD

Dash nextstate operation in a procedure.

Implementation of Any Decisions

An any decision with two paths are generated according to the following structure:

```
BEGIN_ANY_DECISION(2)
DEF_ANY_PATH(1, 2)
DEF_ANY_PATH(2, 0)
END_DEFS_ANY_PATH(2)
BEGIN_FIRST_ANY_PATH(1)
  statements
END_ANY_PATH
BEGIN_ANY_PATH(2)
  statements
END_ANY_PATH
END_ANY_DECISION
```

BEGIN_ANY_DECISION

```
(NO_OF_PATHS)
```

Start of the any decision. `NO_OF_PATHS` is the number of paths in the decision.

BEGIN_ANY_PATH

```
(PATH_NO)
```

A path (not the first) in implementation part of the any decision. `PATH_NO` is the path number.

BEGIN_FIRST_ANY_PATH

```
(PATH_NO)
```

The first possible path in implementation part of the any decision. `PATH_NO` is the path number.

DEF_ANY_PATH

`(PATH_NO, SYMBOLNUMBER)`

Definition of a path in the decision.

Parameters:

- `PATH_NO`
  the path number.
- `SYMBOLNUMBER`
  the symbol number for the first symbol in this path.

END_ANY_DECISION

The end of the any decision.

END_ANY_PATH

End of one of the paths in the implementation section.

END_DEFS_ANY_PATH

`(NO_OF_PATHS)`

End of the definition part of the any decision. `NO_OF_PATHS` is the number of paths in the decision.

Implementation of Informal Decisions

The implementation of informal decisions are similar to any decisions.

BEGIN_FIRST_INFORMAL_PATH

`(PATH_NO)`

The first possible path in implementation part of the informal decision. `PATH_NO` is the path number.

BEGIN_INFORMAL_DECISION

`(NO_OF_PATHS, QUESTION)`

Start of the any decision.

Parameters:

- `NO_OF_PATHS`
  the number of paths in the decision.

- `QUESTION`
  the question charstring.

BEGIN_INFORMAL_ELSE_PATH

`(PATH_NO)`

The else path in implementation part of the any decision. `PATH_NO` is the path number.

BEGIN_INFORMAL_PATH

`(PATH_NO)`

A path in implementation part of the any decision. `PATH_NO` is the path number.

DEF_INFORMAL_PATH

`(PATH_NO, ANSWER, SYMBOLNUMBER)`

Definition of a path in the decision.

Parameters:

- `PATH_NO`
  the path number.

- `ANSWER`
  the answer string.

- `SYMBOLNUMBER`
  the symbol number for the first symbol in this path.

DEF_INFORMAL_ELSE_PATH

`(PATH_NO, SYMBOLNUMBER)`

Definition of the else path in the decision.

Parameters:

- `PATH_NO`
  the path number.

- `SYMBOLNUMBER`
  the symbol number for the first symbol in this path.

END_DEFS_INFORMAL_PATH

`(NO_OF_PATHS)`

End of the definition part of the informal decision. `NO_OF_PATHS` is the number of paths in the decision.

END_INFORMAL_ELSE_PATH

End of the else paths in the implementation section.

END_INFORMAL_DECISION

The end of the informal decision.

END_INFORMAL_PATH

End of one of the paths in the implementation section.

Macros for Component Selection Tests

The macros in this section handles testing the validity of for example a component selection of a choice or #UNION variable. Also tests for optional components in structs and for de-referencing of pointers is treated here.

XCHECK_CHOICE_USAGE

```
(TAG,VALUE,NEQTAG,COMPNAME,CURR_VALUE,TYPEINFO)
```

XSET_CHOICE_TAG

```
(TAG,VALUE,ASSTAG,NEQTAG,COMPNAME,CURR_VALUE,
 TYPEINFO)
```

XSET_CHOICE_TAG_FREE

```
(TAG,VALUE,ASSTAG,NEQTAG,FREEFUNC,COMPNAME,
 CURR_VALUE,TYPEINFO)
```

The CHOICE macros are used to test and to set the implicit tag in a choice variable. The XSET_CHOICE_TAG and XSET_CHOICE_TAG_FREE set the tag when some component of the choice is assigned a value. The FREE version of the macro is used if the choice contains some component that has a Free function. The XCHECK_CHOICE_USAGE is used to test if an accessed component is active or not.

Parameters:

- TAG
  The implicit tag component
- VALUE
  The new or expected tag value
- ASSTAG
  The assignment function for the tag type
- NEQTAG
  The equal test function for the tag type
- FREEFUNC
  The Free function for the Choice type
- COMPNAME
  The name of the selected component as a char string
- CURR_VALUE
  The current value of the tag type
- TYPEINFO
  The type info node for the tag type.

XCHECK_OPTIONAL_USAGE

```
(PRESENT_VAR,COMPNAME)
```

This macro is used to check that a selected optional component is present. The PRESENT_VAR parameter is the present variable for this component, while COMPNAME is the selected components name as a char string.

XCHECK_UNION_TAG_USAGE

```
(TAG,VALUE,NEQTAG,COMPNAME,CURR_VALUE,TYPEINFO)
```

XCHECK_UNION_TAG

```
(TAG,VALUE,ASSTAG,NEQTAG,COMPNAME,CURR_VALUE,
 TYPEINFO)
```

XCHECK_UNION_TAG_FREE

```
(TAG,VALUE,ASSTAG,NEQTAG,FREEFUNC,COMPNAME,
 CURR_VALUE,TYPEINFO)
```

The UNION macros are used to test tag in a union variable. The XCHECK_UNION_TAG and XCHECK_UNION_TAG_FREE check the tag when some component of the union is assigned a value. The

FREE version of the macro is used if the union contains some component that has a Free function. The XCHECK_UNION_USAGE is used to test if an accessed component is active or not.

Parameters:

- TAG
  The tag component
- VALUE
  The expected tag value
- ASSTAG
  The assignment function for the tag type
- NEQTAG
  The equal test function for the tag type
- FREEFUNC
  The Free function for the UNION type
- COMPNAME
  The name of the selected component as a char string
- CURR_VALUE
  The current value of the tag type
- TYPEINFO
  The type info node for the tag type.

XCHECK_REF

XCHECK_OWN

XCHECK_OREF

```
    (VALUE,REF_TYPEINFO,REF_SORT)
```

These macros are used to implement a test that Null pointers (using the Ref, Own, or ORef generator) are not de-referenced. These macros are inserted before each statement containing a Ref/Own/ORef pointer de-referencing. In case of an ORef pointer it is also checked that the ORef is valid, i.e. that it refers to an object owned by the current process.

Parameters:

- VALUE
  This is the value of the pointer.

- REF_TYPEINFO
  The typeinfo node for the Ref sort.
- REF_SORT
  The C type that corresponds to the Ref instantiation newtype.

XCHECK_OREF2

`(VALUE)`

Checks that a ORef pointer is a valid pointer, i.e. NULL, or that it refers to an object owned by the current process.

Debug and Simulation Macros

XAFTER_VALUE_RET_PRDCALL

`(SYMB_NO)`

A macro generated between the implementation of a value returning procedure call (implicit call symbol) and the symbol containing the value returning procedure call. `SYMB_NO` is the symbol number of the symbol containing the value returning procedure call.

XAT_FIRST_SYMBOL

`(SYMB_NO)`

A macro generated between an input or start symbol and the first symbol in the transition. `SYMB_NO` is the symbol number of the first symbol in the transition.

XAT_LAST_SYMBOL

A macro generated immediately before a nextstate or stop operation.

XBETWEEN_STMTS

XBETWEEN_STMTS_PRD

`(SYMB_NO, C_LINE_NO)`

A macro generated between statements in a task. The suffix _PRD indicates that these statements are part of a procedure.

Parameters:

- `SYMB_NO`
  the symbol number of the next statement.

- `C_LINE_NO`
  line number in C of this statement.

XBETWEEN_SYMBOLS

XBETWEEN_SYMBOLS_PRD

`(SYMB_NO, C_LINE_NO)`

A macro generated between symbols in a transition. The suffix _PRD indicates that these symbols are part of a procedure.

Parameters:

- `SYMB_NO`
  the symbol number of the next symbol.
- `C_LINE_NO`
  line number in C of this statement.

XDEBUG_LABEL

`(LABEL_NAME)`

This macro gives the possibility to insert label at the beginning of transitions. Such labels can be useful during debugging. The `LABEL_NAME` parameter is a concatenation of state name and the signal name. The * in state *; and input *; will cause the name ASTERISK to appear.

XOS_TRACE_INPUT

`(SIG_NAME_STRING)`

This macro is generated at input statements and can, for example, be used to generated trace information about inputs. The `SIG_NAME_STRING` parameter is the name of the signal in the input.

YPRSNAME_VAR

`(PRS_NAME_STRING)`

This macro is generated among the declarations of variables in the PAD function for a process. It can, for example, be used to declare a `char *` variable containing the name of the process. Such a variable can be useful during debugging. The `PRS_NAME_STRING` parameter is the name of the process as a character string.

YPRDNAME_VAR

`(PRD_NAME_STRING)`

This macro is generated among the declarations of variables in the PRD function for a procedure. It can, for example, be used to declare a `char*` variable containing the name of the procedure. Such a variable can be useful during debugging. The `PRD_NAME_STRING` parameter is the name of the procedure as a character string.

Utility Macros to Be Inserted

The following sequence of macros should be inserted. Most of them concern removal of struct components (in IdNodes) that are not used due to the combination of other switches used.

```
#define NIL 0
#define XXFREE xFree
#define XSYSD xSysD.

#if defined(XPRSPRIO) || defined(XSIGPRSPRIO) ||
    defined(XPRSSIGPRIO)
#define xPrsPrioPar(p)   , p
#else
#define xPrsPrioPar(p)
#endif

#if defined(XSIGPRIO) || defined(XSIGPRSPRIO) ||
    defined(XPRSSIGPRIO)
#define xSigPrioPar(p)   , p
#define xSigPrioParS(p)  p;
#else
#define xSigPrioPar(p)
#define xSigPrioParS(p)
#endif

#ifdef XTESTF
#define xTestF(p)   , p
#else
#define xTestF(p)
#endif

#ifdef XREADANDWRITEF
#define xRaWF(p)   , p
#else
#define xRaWF(p)
#endif

#ifdef XFREEFUNCS
#define xFreF(p)   , p
#else
#define xFreF(p)
#endif

#ifdef XFREESIGNALFUNCS
#define xFreS(p)   , p
```

```
#else
#define xFreS(p)
#endif

#define xAssF(p)
#define xEqF(p)



#ifdef XIDNAMES
#define xIdNames(p)   , p
#else
#define xIdNames(p)
#endif

#ifndef XOPTCHAN
#define xOptChan(p)   , p
#else
#define xOptChan(p)
#endif

#ifdef XBREAKBEFORE
#define xBreakB(p)   , p
#else
#define xBreakB(p)
#endif

#ifdef XGRTRACE
#define xGRTrace(p)   , p
#else
#define xGRTrace(p)
#endif

#ifdef XMSCE
#define xMSCETrace(p)   , p
#else
#define xMSCETrace(p)
#endif

#ifdef XTRACE
#define xTrace(p)   , p
#else
#define xTrace(p)
#endif

#ifdef XCOVERAGE
#define xCoverage(p)   , p
#else
#define xCoverage(p)
#endif

#ifdef XNRINST
#define xNrInst(p)   , p
#else
#define xNrInst(p)
```

```
#endif

#ifdef XSYMBTLINK
#define xSymbTLink(p1, p2) , p1, p2
#else
#define xSymbTLink(p1, p2)
#endif

#ifdef XEVIEW
#define xeView(p)  p,
#define xeViewS(p)  p;
#else
#define xeView(p)
#define xeViewS(p)
#endif

#ifdef XCTRACE
#define xCTrace(p)  p,
#define xCTraceS(p)  p;
#else
#define xCTrace(p)
#define xCTraceS(p)
#endif

#ifndef XNOUSEOFSERVICE
#define xService(p)  , p
#else
#define xService(p)
#endif

#if !defined(XPMCOMM) && !defined(XENV)
#define xGlobalNodeNumber() 1
#endif

#define xSizeOfPathStack 50

#ifndef xOffsetOf
#define xOffsetOf(type, field) \
        ((xptrint) &((type *) 0)->field)
#endif
#define xToLower(C) \
        ((C >= 'A' && C <= 'Z') ? \
        (char)((int)C - (int)'A' + (int)'a') : C)

#ifndef xDefaultPrioProcess
#define xDefaultPrioProcess      100
#endif

#ifndef xDefaultPrioSignal
#define xDefaultPrioSignal       100
#endif

#ifndef xDefaultPrioTimerSignal
#define xDefaultPrioTimerSignal  100
#endif
```

```
#ifndef xDefaultPrioContSignal
#define xDefaultPrioContSignal   100
#endif

#ifndef xDefaultPrioCreate
#define xDefaultPrioCreate       100
#endif

#define xbool int

#ifndef MAX_READ_LENGTH
#define MAX_READ_LENGTH 5000
                      /* max length of input line */
#endif
```

The xDefaultPrio macros above should, of course, be defined to the suitable default values.

Other macros that should be defined are.

SDL_NULL

a null value for the type PId.

xNotDefPId

which is used as RECEIVER parameter in the SDL_2OUTPUT macros. Please see also the section were signals are treated.