Chapter

58 *Building an Application*

This chapter describes how you can use the Cadvanced SDL to C Compiler to generate applications and especially how to design the environment functions. These functions allows you connect the SDL system with the environment of the system.

You should read *chapter 57, The Cadvanced/Cbasic SDL to C Compiler* before reading this chapter to understand the general behavior of the Cadvanced SDL to C Compiler. Much of what you need to know to generate an application may be found there, and that information is not repeated here.

Chapter 58 Building an Application

Introduction

The Basic Idea

An application generated with the Cadvanced SDL to C Compiler can be viewed as having three parts:

- The SDL system
- The physical environment of the system
- The environment functions, where you connect the SDL system with the environment of the system

In the SDL system process transitions are executed in priority order, signals are sent from one process to another initiating new transitions, timer signals are sent, and so on. These are examples of internal actions that only affect the execution of the SDL system. An SDL system communicates with its environment by sending signals to the environment and by receiving signals from the environment.

The physical environment of an application consists of an operating system, a file system, the hardware, a network of computers, and so on. In this world other actions than just signal sending are required. Examples of actions that an application wants to perform are:

- To read or to write on a file
- To send or receive messages over a network
- To respond on interrupts
- To read and to write information on hardware ports or on sockets

The environment functions are the place where the two worlds, the SDL system and the physical environment, meet. Here signals sent from the SDL system to the environment can induce all kinds of events in the physical environment, and events in the environment might cause signals to be sent into the SDL system. You have to provide the environment functions, as the Cadvanced SDL to C Compiler has no knowledge of the actions that should be performed.

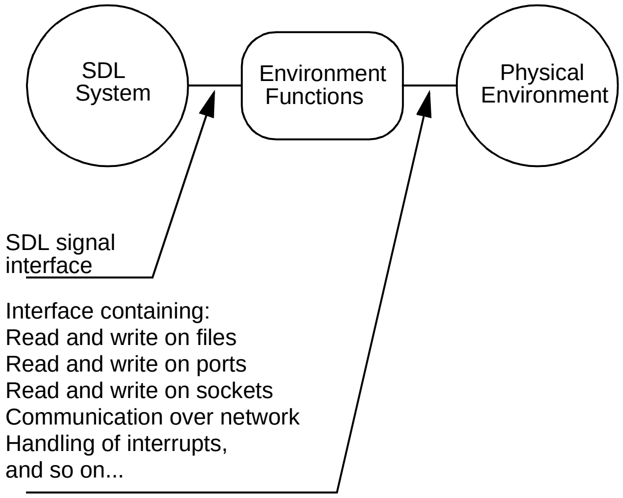

Figure 490: Structure of an application

In a distributed system an application might consist of several communicating SDL systems. Each SDL system will become one executable program. It might execute either as an operating system process, communicating with other operating system processes, or it might execute in a processor of its own, communicating over a network with other processors. There may, of course, also be combinations of these cases. Let us for the sake of simplicity call the operating system processes or processors for nodes communicating over a network. In the case of communicating operating system (OS) processes, the network will be the facilities for process communication provided by the OS.

There are no problems in building an application consisting of several nodes communicating over a network using the Cadvanced SDL to C Compiler. However, you have to implement the communication between the nodes in the environment functions.

Note:

All nodes in a network do not need to be programs generated by the *Cadvanced SDL to C Compiler* from SDL systems. As long as a node can communicate with other nodes, it might be implemented using any technique.

The PId values (references to process instances), are a problem in a distributed world containing several communicating SDL systems. We still want, for example, “Output To Sender” to work, even if Sender refers to a process instance in another SDL system. To cope with this kind of problem, a global node number has been introduced as a component in a PId value. The global node number, which is a unique integer value assigned to each node, identifies the node where the current process instance resides, while the other component in the PId value is a local identification of the process instance within the node (SDL system).

The partitioning of an application into an SDL system and the environment functions has additional advantages. It separates **external actions** into the **logical decision** to perform the action (the decision to send a signal to the environment) and the **implementation details** of the action (treating the signal in the environment functions). This kind of separation reduces the complexity of the problem and allows separate testing. It also allows parallel development of the logic (the SDL system) and the interface towards the environment (the environment functions). When the signal interface between the SDL system and its environment is settled, it is possible to continue both the activities in parallel.

Libraries

Two libraries, *Application* and *ApplicationDebug*, are provided to generate applications. Both use real time (see *“Time” on page 2576 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler* and perform calls to environment functions (see section “The Environment Functions” on page 2702). The difference is that *ApplicationDebug* includes the monitor system while *Application* does not include the monitor system.

When an application is developed, it is usually appropriate to first simulate and debug the SDL system or systems without its environment. One of the libraries *Simulation* or *RealTimeSimulation* may then be used. You first simulate each SDL system on its own and can then simulate the systems together (if you have communicating systems) using the facility of communicating simulations. After that you probably want to debug the application with the environment functions. This may be performed with the library *ApplicationDebug*. You may then generate the application with the library *Application*.

The library *Validation* allows you to build validators from the code generated by the Cadvanced SDL to C Compiler. A Validator has a user interface and executing principles that are similar to a Simulator. The technical approach is however different; a Validator is based on a technique called *state space exploration* and operates on structures called *behavior trees*. Its purpose is to validate an SDL system in order to find errors and to verify its consistency against Message Sequence Charts.

Reference Section

Representation of Signals and Processes

In this first section, the representation of signals and processes is presented. The symbol table, which is a representation of the static structure of the system, will also be discussed. The information given here will be used in the next part of this section where the environment functions, which should be provided by the user, are described.

Types Representing Signals

A signal is represented by **a C struct** containing **general information** about the signal followed by the **parameters** carried by the signal.

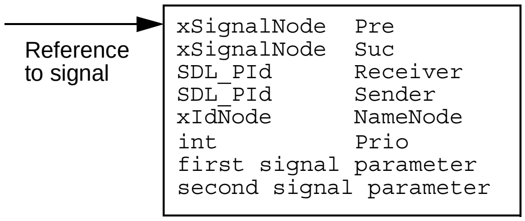

Figure 491: Data structure representing a signal

A general typedef `xSignalRec` for a signal without parameters and for a pointer to such a signal, `xSignalNode`, are given below. These types may be found in the file `scttypes.h`. These types may be used for type casting of any signal to access the general components.

```
typedef struct xSignalRec *xSignalNode;
typedef struct xSignalRec {
  xSignalNode Pre;
  xSignalNode Suc;
  SDL_PId     Receiver;
  SDL_PId     Sender;
  xIdNode     NameNode;
```

```
    int          Prio;
  } xSignalRec;
```

A `xSignalRec` contains the following components:

- `Pre` and `Suc`. These components are used to link the signal in the input port list of the receiving process instance. The input port is implemented as a double linked list. When a signal has been consumed and the information contained in the signal is no longer needed, the signal will be returned to an avail list to be re-used in future outputs. The component `Suc` is used to link the signal into the avail list, while `Pre` will be `(xSignalNode)0` as long as the signal is in the avail list.
- `Receiver`. The receiving process instance.
- `Sender`. The sending process instance.
- `NameNode`. This component is a pointer to the node in the symbol table that represents the signal type. The symbol table is a tree with information about the SDL system and contains, among other things, one node for each signal type that is defined within the SDL system.
- `Prio`. This component represents the priority of the signal and is used in connection with continuous signals.

In the generated code there will be types to represent the parameters of the signals according to the following example:

Example 410: Generated C Code for Signal Definition –––––––––––––

Assume the following signal definitions in SDL:

```
SIGNAL
 S1(Integer),
 S2,
 S3(Integer, Boolean, OwnType);
```

then the C code below will be generated:

```
typedef struct {
  SIGNAL_VARS
  SDL_Integer Param1;
} yPDef_z0f_S1;
typedef yPDef_z0f_S1 *yPDP_z0f_S1;

typedef struct {
  SIGNAL_VARS
  SDL_Integer Param1;
  SDL_Boolean Param2;
```

```
    z09_OwnType Param3;
  } yPDef_z0h_S3;
  typedef yPDef_z0h_S3 *yPDP_z0h_S3;
```

where `SIGNAL_VARS` is a macro defined in `scttypes.h` that is expanded to the common components in a signal struct.

———————————————————————————————

For each signal with parameters there are two generated types, a struct type and a pointer type. The struct type contains one component for each parameter in the signal definition and the components will be named `Param1`, `Param2` and so on. The components will be placed in the same order in the struct as the parameters are placed in the signal definition.

Note:

There are no generated types for a signal without parameters.

Types Representing Processes

A PId value is a struct consisting of two components, a **global node number**, which is an integer (see also "Function xGlobalNodeNumber" on page 2722 and "The Basic Idea" on page 2696) and a **local PId value**, which is a pointer.

```
typedef xLocalPIdRec *xLocalPIdNode;

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

The global node number identifies the SDL system that the process instance belongs to, while the local PId value identifies the process instance within the system. The local PId pointer value should not be referenced outside the SDL system where it is defined.

By introducing a global node number in the PId values, these values are possible to interpret throughout an application consisting of several SDL systems. You can also define your own PId values in non-SDL defined parts of the application and still use communication with signals.

The variable `SDL_NULL`, which represents a null value for PIds and which is defined in the runtime library and available through the file `scttypes.h`, contains zero in both the global node number and the local PId component. Note that the global node number should be greater than zero in all PId values except SDL_NULL.

The Symbol Table

The symbol table is a tree built up during the initialization phase in the execution of the generated program and contains information about the **static structure of the SDL system**. The symbol table contains, for example, nodes which represent signal types, blocks, channels, process types, and procedures. The C type that are used to represent for example signals in the symbol table is given below.

```
typedef struct xSignalIdStruct *xSignalIdNode;
typedef struct xSignalIdStruct {
 /* components */
} xSignalIdRec;
```

It is the nodes that represent the signal types, for signals sent to and from the environment of the SDL system, that are of major interest in connection with the environment functions. For each signal type there will be a symbol table node. That node may be referenced using the name ySigN_ followed by the signal name with prefix. Such references may be used in, for example, xOutEnv to find the signal type of the signal passed as parameter.

In some cases the symbol table nodes for channels from the environment to a block in the system are of interest to refer to. In a similar way as for signals such nodes may be referenced using the name yChaN_ followed by the channel name with prefix.

The Environment Functions

An SDL system communicates with its environment by sending signals to the environment and by receiving signals from the environment. As no information about the environment is given in the SDL system, the Cadvanced SDL to C Compiler cannot generate the actions that should be performed when, for instance, a signal is sent to the environment. Instead you have to provide a function that performs this mapping between a signal sent to the environment and the actions that then should be performed. Examples of such actions are writing a bit pattern on a port, sending information over a network to another computer and sending information to another OS process using some OS primitive.

You should provide the following functions to represent the environment of the SDL system:

- `xInitEnv` and `xCloseEnv`, which are called during initialization and termination of the application
- `xOutEnv` which should treat signals sent to the environment
- `xInEnv` which should treat signals sent into the SDL system from the environment

There are two ways to get a skeleton for the env functions:

- You can copy the file `sctenv.c` from the directory
  `<`*`installation directory`*`>/sdt/sdtdir/<`*`machine dependent dir`*`>/INCLUDE`
  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 file also contains some trace mechanisms that may be used to trace the execution in a target computer. This trace can, however, only be used if you have the source code for the run-time library (included in the Cadvanced SDL to C Compiler) and can produce a new object library with the appropriate switches.
- You can generate a skeleton by using the *Generate environment functions* option in the Make dialog in the Organizer. In very simple cases you might obtain executable env functions by just tuning the macros in this generated file, but in the general case you must use it as a skeleton and edit it. Remember then to copy the file so that it is not overwritten when code is generated the next time.

An advantage with the generated env functions is that the SDL to C Compiler knows about the signal interface to be implemented in the env functions, and can therefore insert code or macros for all signals in the interface. To calculate this information is not that easy, especially if partitioning (generating code for a part of a system) is used.

Note:

A make template file is generated every time you generate an environment file. This file contains make information for the environment file and possibly for data encoding and decoding files. If you need to change this skeleton file, then remember to copy it so it is not overwritten next time an environment file is generated. The file can be used as make template in the Organizer's generate options. **Note that you may have to generate the file first, before you can select it in the Organizer.**

The env functions are thoroughly discussed below, but first we will introduce the *system interface header file* which considerably simplifies writing the environment functions.

System Interface Header File

The system interface header file contains code for objects that are defined in the system diagram. Included are all type definitions and other external definitions that are needed in order to implement external C code. These object definitions simplify the implementation of the environment functions. Therefore the system interface header file is also known the environment header file. This file is generated if:

- *Code* is generated for the complete system.
- The *Generate environment header file* option is selected in the *Make* dialog in the Organizer (see "Code Generation Options" on page 120 in chapter 2, *The Organizer*).

The default name of the generated interface header file is `<system_file_name>.ifc`.

The system interface header file, has the following structure:

- Macros for all synonyms that are translated to macros.
- All type definitions generated from newtypes and syntypes. This includes `#TYPE` and `#HEADING` sections in `#ADT` directives and in `#CODE` directives.
- External definitions of variables for all synonyms that are translated to variables.

- For each signal defined in the system diagram there will be an extern definition for the `xSignalIdRec` variable representing the signal.
- For each signal with parameters defined in the system diagram, there will be definitions of the types `yPDef_SignalName` and `yPDP_SignalName`, i.e. of the types used to represent a signal.
- For each remote procedure (that can be sent to or from the environment), code will be generated exactly as for two signals named `pCALL_procedurename` and `pREPLY_procedurename`.
- For each channel defined in the system diagram there will be extern definitions for the `xChannelIdRec` representing the channel.

Together with these definitions, macros that simplify the translation of SDL names to C names are also be generated.

Names of SDL Objects in C

Due to differences in naming rules in SDL and C, prefixing is used to make C identifiers unique (see section *"Names and Prefixes in Generated Code" on page 2663 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler)*. These prefixes, however, may change when you update your SDL diagrams and cannot be predicted. Therefore you should not use the prefixed object names in the environment functions. Instead macros, generated in the system interface header file, assist you by mapping static names to the prefixed names. This means that you must regenerate the system interface header file each time you regenerate code for the system. The good part is that you do not have to make any changes in the environment functions, as the interface names are static.

Example 411: Macro in the system interface header file –––––––––––

If an SDL signal called Sig1 is defined in the system, the following macro is created:

```
extern XCONST struct xSignalIdStruct ySigR_z5_Sig1;
#ifndef Sig1
#define Sig1 (&ySigR_z5_Sig1)
#endif
```

This macro allows you to refer to the `xIdNode` by using the static name `Sig1` rather than the prefixed name, `ySigR_z5_Sig1`.

–––

Macros generate static names for the following SDL types:

- Synonyms (both translated to macros and variables).
- Newtypes and Syntypes. If the newtype is translated to an enumeration type, all the literals are available directly in C using their SDL names.
- `xSignalIdNode` representing signals. (No `ySigN_` prefix).
- `xChannelIdNode` representing channels. (Use prefix `xIN_` or `xOUT_` to access the incoming or outgoing direction of the channel).
- The `yPDP_SignalName` pointer type. This type may be referred to using the name `yPDP_SignalName`, where `SignalName` is the SDL name.

Note:

You must always generate the system interface header file before editing or generating the environment functions.

Avoiding name clashes

In SDL it is allowed to give the same name to different objects. This is not allowed in C. For instance, in SDL you can give a signal and a Newtype the same name. In order to distinguish between the names in the system interface header file, you must define static unambiguous names. Using the *Env. Header File Generation* tab in the Targeting Expert, you can do this by using available general identifiers. The identifiers are:

- `%n` - This identifier is the SDL name
- `%s` - This identifier is the SDL name of the scope.
- `sdlobject` - In order to identify the type of object, you can type the object name as a prefix, e.g. signal, literal, etc.

Any combination of the identifiers can be used and they are all optional. However, in order to create a useful system interface header file, it is recommended that the `%n` identifier is always included. Leaving the field empty means that no objects of that type is included in the system interface header file at all.

Note:

If you select to include all objects and use the `%n` identifier only, the system interface header file will become compatible with earlier versions.

Example 412: Name Mapping in an system interface header file ––––

If the signal `Signal1` is in the system `System2` you should type the following in the Signal field in the TAEX.

```
sig_%n_%s
```

The result in the system interface header file will be:

```
sig_Signal1_System2
```

––

This approach helps you to avoid name clashes in the ifc file. Literals, for example, are often given the same name when defined in different types. The following example shows how this can be solved.

Example 413: Avoiding Name Clashes –––––––––––––––––––––––––

In an SDL system the following two newtypes are defined:

```
NewType s
   literals red, green

NewType t
   literals red, yellow
```

As the literal `red` appears in both newtypes, the C code cannot distinguish between them. However, by using the identifiers in the literals field,

```
lit_%n_%s
```

the literals are given the following names:

```
lit_red_s
lit_red_t
```

Thus we will avoid a possible name clash.

––

Structure of File for Environment Functions

The file containing the environment functions should have the following structure:

```
#include "scttypes.h"
#include "file with macros for external synonyms"
#include "systemfilename.ifc"

void xInitEnv XPP((void))
{
}

void xCloseEnv XPP((void))
{
}

#ifndef XNOPROTO
void xOutEnv (xSignalNode *S)
#else
void xOutEnv (S)
  xSignalNode *S;
#endif
{
}

#ifndef XNOPROTO
void xInEnv (SDL_Time Time_for_next_event)
#else
void xInEnv (Time_for_next_event)
  SDL_Time Time_for_next_event;
#endif
{
}

int xGlobalNodeNumber XPP((void))
{
}
```

The last function, `xGlobalNodeNumber`, will be discussed later, see "Function xGlobalNodeNumber" on page 2722. The usage of the macros `XPP` and `XNOPROTO` makes the code possible to compile both with compilers that can handle prototypes and with compilers that cannot. If you do not need this portability, you can reduce the complexity of the function headings somewhat. In the minor examples in the remaining part of this section, only versions with prototypes are shown.

Functions xInitEnv and xCloseEnv

There are two functions among the environment functions that handle initialization and termination of the environment. These functions, as well as the other environment functions, should be provided by the user.

```
void xInitEnv ( void );

void xCloseEnv ( void );
```

In the implementation of these functions you can place the appropriate code needed to initialize and terminate the software and the hardware. The function `xInitEnv` will be called during the start up of the program as first action, while the `xCloseEnv` will be called in the function `SDL_Halt`. Calling `SDL_Halt` is the appropriate way to terminate the program. The easiest way to call `SDL_Halt` is to include the call in a `#CODE` directive in a TASK. `SDL_Halt` is part of the runtime library and has the following definition:

```
void SDL_Halt ( void );
```

Note:

`xInitEnv` will be called before the SDL system is initialized, which means that no references to the SDL system are allowed in this function. To, for example, send signals into the system during the initialization phase, the `#MAIN` directive should be used (see "Initialization – Directive #MAIN" on page 2668 in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*). This directive code can be used after the initialization of the SDL system, but before any transitions are executed.

Function xOutEnv

Each time a signal is sent from the SDL system to the environment of the system, the function `xOutEnv` will be called.

```
void xOutEnv ( xSignalNode *S );
```

The `xOutEnv` function will have the current signal as parameter, so you have all the information contained in the signal at your disposal when you implement the actions that should be performed. The signal contains the signal type, the sending and receiving process instance and the parameters of the signal. For more information about the types used to represent signals and processes, see section "Types Representing Signals" on page 2699 and "Types Representing Processes" on page 2701.

Note that the parameter of `xOutEnv` is an address to `xSignalNode`, that is, an address to a pointer to a struct representing the signal. The reason for this is that the signal that is given as parameter to `xOutEnv` should be returned to the pool of available memory before return is made from the `xOutEnv` function. This is made by calling the function `xReleaseSignal`, which takes an address to an `xSignalNode` as parameter, returns the signal to the pool of available memory, and assigns 0 to the `xSignalNode` parameter. Thus, there should be a call

```
xReleaseSignal(S);
```

before returning from `xOutEnv`. The `xReleaseSignal` function is defined as follows:

```
void xReleaseSignal ( xSignalNode *S );
```

In the function `xOutEnv` you may use the information in the signal that is passed as parameters to the function. First it is usually suitable to determine the signal type of the signal. This is best performed by `if` statements containing expressions of the following form, assuming the use of the system interface header file and that the signal has the name `Sig1` in SDL:

```
(*S)->NameNode == Sig1
```

Suitable expressions to reach the `Receiver`, the `Sender`, and the signal parameters are:

```
(*S)->Receiver
(*S)->Sender
((yPDP_Sig1)(*S)) -> Param1
((yPDP_Sig1)(*S)) -> Param2
```

(and so on)

`Sender` will always refer to the sending process instance, while `Receiver` is either a reference to a process in the environment or the value `xEnv`. `xEnv` is a PId value that refers to an environment process instance, which is used to represent the general concept of environment, without specifying an explicit process instance in the environment.

Note:

It is not possible to calculate the PId value for a process in the environment, the value has to be taken from an incoming signal (sender or signal parameter). This is the normal procedure in SDL to establish direct communication between two processes in the same SDL system.

Receiver will refer to the process `xEnv` if the PId expression in an output TO refers to `xEnv`, or if the signal was sent in an output without a TO clause and the environment was selected as receiver in the scan for receivers.

Remote procedure calls to or from the environment should in the environment functions be treated a two signals, a `pCALL_procedurename` and a `pREPLY_procedurename` signal.

Recommended Structure of the xOutEnv Function

You can, of course, write the `xOutEnv` function as you wish – the structure discussed below may be seen as an example – but also as a guideline of how to design `xOutEnv` functions.

Example 414: Structure of xOutEnv Function ––––––––––––––––––––

```
void xOutEnv ( xSignalNode *S )
{
 if ( (*S)->NameNode == Sig1 ) {
  /* perform appropriate actions */
  xReleaseSignal(S);
  return;
 }
 if ( (*S)->NameNode == Sig2 ){
  /* perform appropriate actions */
  xReleaseSignal(S);
  return;
 }
 /* and so on */
}
```

–––

Function xInEnv

To make it possible to receive signals from the environment and to send them into the SDL system, the user provided function `xInEnv` is repeatedly called during the execution of the system (see section "Program Structure" on page 2723). During such a call you should scan the environment to see if anything has occurred which should trigger a signal to be sent to a process within the SDL system.

```
void xInEnv (SDL_Time Time_for_next_event);
```

To implement the sending of a signal into the SDL system, two functions are available: `xGetSignal`, which is used to obtain a data area suitable to represent the signal, and `SDL_Output`, which sends the signal to the specified receiver according to the semantic rules of SDL. These functions will be described later in this subsection.

The parameter `Time_for_next_event` will contain the time for the next event scheduled in the SDL system. The parameter will either be 0, which indicates that there is a transition (or a timer output) that can be executed immediately, or be greater than Now, indicating that the next event is a timer output scheduled at the specified time, or be a very large number, indicating that there is no scheduled action in the system, that is, the system is waiting for an external stimuli. This large value can be found in the variable `xSysD.xMaxTime`.

You should scan the environment, perform the current outputs, and return as fast as possible if Time has past `Time_for_next_event`.

If Time has not past `Time_for_next_event`, you have a choice to either return from the `xInEnv` function at once and have repeated calls of `xInEnv`, or stay in the `xInEnv` until something triggers a signal output (a signal sent to the SDL system) or until Time has past `Time_for_next_event`.

Note:

We recommend always to return from the `xInEnv` function as fast as possible to ensure that it will work appropriately together with the monitor (during debugging). **Otherwise**, the **keyboard polling**, that is, typing `<RETURN>` in order to interrupt the execution, **will not work**.

The function `xGetSignal`, which is one of the service functions suitable to use when a signal should be sent, returns a pointer to a data area that represents a signal instance of the type specified by the first parameter.

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

The components `Receiver` and `Sender` in the signal instance will also be given the values of the corresponding parameters.

- `SType`. This parameter should be a reference to the symbol table node that represents the current signal type. Using the *system interface header file*, such a symbol table node may be referenced using the signal name directly.
- `Receiver`. This parameter should either be a PId value for a process instance within the system, or the value `xNotDefPId`. The value `xNotDefPId` is used to indicate that the signal should be sent as an output without TO clause, while if a PId value is given the output, it is treated as an output with TO clause. Note that PId values for process instances in an SDL system cannot be calculated, but have to be captured from the information (sender or parameter) carried by signals coming from the system. This is the normal procedure in SDL to establish direct communication.
- `Sender`. Sender should either be a PId value representing a process instance in the environment of the current SDL system or the value `xEnv`. `xEnv` is a PId value that refers to an environment process instance, which is used to represent the general concept of the SDL environment, without specifying an explicit process instance in the environment.

The function `SDL_Output` takes a reference to a signal instance and outputs the signal according to the rules of SDL.

```
void SDL_Output
  ( xSignalNode S,
    xIdNode     ViaList[] );
```

- `S`. This parameter should be a reference to a signal instance with all components filled in.
- `ViaList`. This parameter is used to specify if a VIA clause is or is not part of the output statement. The value `(xIdNode *)0` (a null pointer), is used to represent that no VIA clause is present. For information about how to build a via list, please see below.

We now have enough information to be able to write the code to send a signal. Suppose we want to send a signal S1, without parameters, from `xEnv` into the system without an explicit receiver (without TO). The code will then be:

Example 415: C Code to Send a Signal to the Environment

```
SDL_Output( xGetSignal(S1, xNotDefPId, xEnv),
  (xIdNode *)0 );
```

If S2, with two integer parameters, should be sent from `xEnv` to the process instance referenced by the variable P, the code will be:

```
xSignalNode OutputSignal; /* local variable */
...
OutputSignal = xGetSignal(S2, P, xEnv);
((yPDP_S2)OutputSignal)->Param1 = 1;
((yPDP_S2)OutputSignal)->Param2 = 2;
SDL_Output( OutputSignal, (xIdNode *)0 );
```

For the details of how to reference the parameters of a signal see the subsection "Types Representing Signals" on page 2699.

To introduce a via list in the output requires a variable, which should be an array of `xIdNode`, that contains references to the symbol table nodes representing the current channels (or signal routes) in the via list. In more detail, we need a variable

```
ViaList xIdNode[N];
```

where N should be replaced by the length of the longest via list we want to represent plus one. The components in the variable should then be given appropriate values, such that component 0 is a reference to the first channel (its symbol table node) in the via list, component 1 is a reference to the second channel, and so on. The last component with a reference to a channel must be followed by a component containing a null pointer (the value `(xIdNode)0`). Components after the null pointer will not be referenced. Below is an example of how to create a via list of two channels, C1 and C2.

Example 416: Via List of two Channels. ––––––––––––––––––––––––––

```
ViaList xIdNode[4];
/* longest via has length 3 */
...
/* this via has length 2 */
ViaList[0] = (xIdNode)xIN_C1;
ViaList[1] = (xIdNode)xIN_C2;
ViaList[2] = (xIdNode)0;
```

––

The variable `ViaList` may then be used as a `ViaList` parameter in a subsequent call to `SDL_Output`.

Guidelines for the xInEnv Function

It is more difficult to give a structure for the `xInEnv` function, than for the `xOutEnv` function discussed in the previous subsection. A `xInEnv` function will in principle consist of a number of `if` statements where the environment is investigated. When some information is found that means that a signal is to be sent to the SDL system, then the appropriate code to send a signal (see above) should be executed.

The structure given in the example below may serve as an idea of how to design the `xInEnv` function.

Example 417: Structure of xInEnv Function ––––––––––––––––––––––

```
void xInEnv (SDL_Time Time_for_next_event)
{
  xSignalNode S;

  if ( Sig1 should be sent to the system ) {
    SDL_Output (xGetSignal(Sig1, xNotDefPId,
      xEnv), (xIdNode *)0);
  }
  if ( Sig2 should be sent to the system ) {
    S = xGetSignal(Sig1, xNotDefPId, xEnv);
    ((xPDP_Sig2)S)->Param1 = 3;
    ((xPDP_Sig2)S)->Param2 = SDL_True;
    SDL_Output (S, (xIdNode *)0);
  }
  /* and so on */
}
```

––

This basic structure can be modified to suit your own needs. The if statements could, for example, be substituted for while statements. The signal types might be sorted in some “priority order” and a return can be introduced last in the if statements. This means that only one signal is sent during a `xInEnv` call, which reduces the latency.

Alternative to OutEnv - Directive #EXTSIG

To speed up an application it is sometimes possible to use the directive `#EXTSIG` instead of the `xOutEnv` function. The decision to use `#EXTSIG` or `xOutEnv` may be taken individually for each signal type.

The usage of the `#EXTSIG` directive is described in the section *"Modifying Outputs – Directive #EXTSIG, #ALT, #TRANSFER" on page 2668 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler*. This information is not repeated here.

By using the `#EXTSIG` directive the following overhead can be avoided:

- Calling `SDL_Output` (the library function for outputs)
- `SDL_Output` determines that the signal is to be sent to the environment
- `SDL_Output` calls `xOutEnv`
- `xOutEnv` executes nested "if" statements to determine the signal type.

Including the Environment Functions in the SDL System Design

Apart from having the environment functions on a file of their own, it is of course possible to include these function directly into the system diagram in a `#CODE` directive.

Example 418: Including Environment Functions in SDL System

```
/*#CODE
#BODY
... code for the environment functions ...
*/
```

In this case you cannot use the system interface header file, but instead you have all the necessary declarations already at your disposal, as the functions will be part of the SDL system. The only problem you will encounter is the prefixing of SDL names when they are translated to C. The `#SDL` directive should be used to handle this problem (or the `#NAME` directive), see sections "Accessing SDL Names in C Code – Directive #SDL" on page 2654 and "Specifying Names in Generated Code – Directive #NAME" on page 2667 in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*. The following table shows how to obtain C names for some SDL objects of interest:

```
#(Synonym name)
#(Newtype or syntype name)
ySigN_#(Signal name)
yPDP_#(Signal name)
yChaN_#(Channel name)
```

SDL Data Encoding and Decoding, ASCII coder

Communication between nodes often requires data encoding and decoding between node internal representations and a common format in a protocol buffer. The sending node writes information in the common format and the receiving node reads information from the common format. Data encoding is the transformation from a node internal representation into a common format and data decoding is the transformation from a common format into a node internal representation.

Supported common formats are:

- BER
- PER
- ASCII

BER (Basic Encoding Rules) is specified in ITU standard X.690 and PER (Packed Encoding Rules) is specified in ITU standard X.691. BER and PER are based on ASN.1 specifications of data types and can only be used for types specified in ASN.1 specifications. See chapter 59, *ASN.1 Encoding and De-coding in the SDL Suite* and chapter 8, *Tutorial: Using ASN.1 Data Types*.

ASCII is a format where the data is represented as ASCII characters. It is easy to read and analyze. The ASCII format is specified in appendix A.

Example 419 ASCII common format ––––––––––––––––––––––––––––

```
newtype Person struct
    nm Charstring;
    nr Integer;
    fm Boolean;
```

```
endnewtype Person;

dcl boss Person := (.'Joe',5,false.);

ASCII format:{'Joe',5,F}
```

The ASCII coder uses the same buffer management and error management as the BER and PER coders, see chapter 59, *ASN.1 Encoding and De-coding in the SDL Suite*.

Type description nodes for SDL types

SDL data encoders and data decoders need information about types and signals, information that is stored in type descriptions nodes. Type description nodes for an SDL system are generated if the Generate SDL coder option is selected in the organizer, see chapter 2, *The Organizer*, or in the targeting expert, see chapter 60, *The Targeting Expert*. Declarations to access the type nodes are in the system interface header files and in a file with the name `<system_file_name>_cod.h`. Type nodes can be found in any generated c-file from a system or a package and also in the `<system_file_name>_cod.c` file.

A type description node for SDL is implemented as a static variable with the type information. The variable can be accessed by using the name ySDL_<type_name> or ySDL_<signal_name>, where <type_name> and <signal_name> are the names used in the interface header file. See "Names of SDL Objects in C" on page 2705 for more information.

Encoding signal and signal parameters into a buffer

You can use an encode function to encode signal parameters into a buffer. There is one encoding function for each common format. For ASCII, it is accessed by using the macro ASCII_ENCODE. An encoding function has a buffer reference as the first parameter, a pointer to a type node as the second parameter and a pointer to the variable to encode as the third. The encoding function returns an integer value, which is 0 if the encoding was successful and error code if it was not. More details about encoding functions, the buffer reference, type nodes and error codes can be found in chapter 59, *ASN.1 Encoding and De-coding in the SDL Suite*.

Function declarations are in the file `"ascii/ascii.h"` (`"ascii\ascii.h" on Windows platforms)` in the coder directory.

Example 420 ASCII encoding of signal parameters ––––––––––––––––

```
In SDL:
SIGNAL Sig1(Integer,Person,Boolean);

In xOutEnv:
BufInitWriteMode(buf);
result = ASCII_ENCODE(buf,
              (tSDLTypeInfo *)&ySDL_Integer,
              (void *)&((yPDP_Sig1)(*S))->Param1));
if (result!=0) /* handle error */;
result = ASCII_ENCODE(buf,
              (tSDLTypeInfo *)&ySDL_Person
              (void *)&((yPDP_Sig1)(*S))->Param2));
if (result!=0) /* handle error */;
result = ASCII_ENCODE(buf,
              (tSDLTypeInfo *)&ySDL_Boolean
              (void *)&((yPDP_Sig1)(*S))->Param3));
if (result!=0) /* handle error */;
BufCloseWriteMode(buf);
```

Example 421 ASCII encoding of whole signal (all signal parameters)

```
In SDL:
SIGNAL Sig1(Integer,Person,Boolean);

In xOutEnv:
BufInitWriteMode(buf);
result = ASCII_ENCODE(buf,
             (tSDLTypeInfo *)&ySDL_Sig1,
              (void *)(*S));
if (result!=0) /* handle error */;
BufCloseWriteMode(buf);
```

Note:

The names of the type nodes in the examples, the second parameter in ASCII_ENCODE, depend on the settings for generating system interface header files.

Decoding into signal parameters from a buffer

You can use a decode function to decode from a buffer into a signal parameter. There is one decoding function for each common format. For ASCII, it is accessed by using the macro ASCII_DECODE. A decode function has a buffer reference as the first parameter, a pointer to a type node as the second parameter and a pointer to the variable to decode as the third. The decoding function returns an integer value, which is 0 if the decoding was successful and an error code if it was not. More details about decoding functions, the buffer reference, type nodes and error codes can be found in chapter 59, *ASN.1 Encoding and De-coding in the SDL Suite*.

Function declarations are in the file `"ascii/ascii.h"` (`"ascii\ascii.h" on Windows platforms`) in the coder directory.

Example 422 ASCII decoding into signal parameters

```
In SDL:
SIGNAL Sig1(Integer,Person,Boolean);

In xInEnv:
BufInitReadMode(buf);
result = ASCII_DECODE(buf,
              (tSDLTypeInfo *)&ySDL_Integer,
              (void *)&((yPDP_Sig1)(S))->Param1));
if (result!=0) /* handle error */;
result = ASCII_DECODE(buf,
              (tSDLTypeInfo *)&ySDL_Person
              (void *)&((yPDP_Sig1)(S))->Param2));
if (result!=0) /* handle error */;
result = ASCII_DECODE(buf,
              (tSDLTypeInfo *)&ySDL_Boolean
              (void *)&((yPDP_Sig1)(S))->Param3));
if (result!=0) /* handle error */;
BufCloseReadMode(buf);
```

Example 423 ASCII decoding of whole signal (all signal parameters)

```
In SDL:
SIGNAL Sig1(Integer,Person,Boolean);

In xInEnv:
BufInitReadMode(buf);
result = ASCII_DECODE(buf,
              (tSDLTypeInfo *)&ySDL_Sig1,
              (void *)S);
if (result!=0) /* handle error */;
BufCloseReadMode(buf);
```

Note:

The names of the type nodes in the examples, the second parameter in ASCII_DECODE, depend on the settings for generating system interface header files.

Encoding and decoding signal identifier for ASCII encoding

When sending signals between nodes it is often important to put a signal identifier first in the buffer. The signal identifier must be a unique identifier of the signal in the distributed system. Decoding is then a two step process, first decode signal identifier and find signal information and second decode signal parameters.

You can use any representation of signal identifiers in the environment functions.

For SDL types there is a special signal id type node that supports character string signal ids. The type node can be used for ASCII encoding.

Example 424 Using signal identifier

```
In SDL:
SIGNAL Sig1(Integer);

In xOutEnv:
void xOutEnv ( xSignalNode *S )
{
 char * signalId;

 BufInitWriteMode(buf);
 if ( (*S)->NameNode == Sig1 ) {
   /* encode signal id into buffer */
   signalId=”Sig1”;
   result = ASCII_ENCODE(buf,
                         (tSDLInfo *)&ySDL_SignalId,
                         (void *)signalId);
   if (result!=0) /* handle error */;
   /* encode signal parameter */
   result =  ASCII_ENCODE(buf,
                      (tSDLTypeInfo *)&ySDL_Sig1,
                       (void *)(*S));
   if (result!=0) /* handle error */;

   /* send buffer using protocol*/

   /* release memory */

   xReleaseSignal(S);
   return;
 }
```

```
     BufCloseWriteMode(buf);
    }

    In xInEnv:
    void xInEnv (SDL_Time Time_for_next_event) {
       char SId[100];

       BufInitReadMode(buf);
       /* decode signal id */
       result = ASCII_DECODE(buf,
                        (tSDLTypeInfo *)&ySDL_SignalId,
                         SId) );
       if (result!=0) /* handle error */;
       /* signal Sig1 in buffer */
       if ( strcmp(SId,"Sig1") ) {
          S=xGetSignal(Sig1,xNotDefPId, xEnv);
          result = ASCII_DECODE(buf,
                        (tSDLTypeInfo *)&ySDL_Sig1,
                         (void *)S);
          if (result!=0) /* handle error */;
          SDL_Output(S, (xIdNode *)0 );
       }
       BufCloseReadMode(buf);
    }
```

Function xGlobalNodeNumber

You should also provide a function, `xGlobalNodeNumber`, with no parameters, which returns an integer that is unique for each executing system.

```
    int xGlobalNodeNumber ( void )
```

The returned integer should be greater than zero and should be unique among the communicating SDL systems that constitutes an application. If the application consists of only one application then this number is of minor interest (it still has to be set). The global node number is used in PId values to identify the node (OS process / processor) that the process instance belongs to. PId values are thereby universally accessible and you may, for example, in a simple way make “Output To Sender” work between processes in different SDL systems (OS processes / processors).

When an application consisting of several communicating SDL systems is designed, you have to map the global node number to the current OS process or processor, to be able to transmit signals addressed to non-local PIds to the correct OS process or processor. This will be part of the `xOutEnv` function.

Program Structure

The generated code will contain two important types of functions, the initialization functions and the PAD functions. The PAD functions implement the actions performed by processes during transitions. There will be one initialization function in each generated `.c` file. In the file that represents the system this function will have the name `yInit`. Each process in the system will be represented by a PAD function, which is called when a process instance of the current instance set is to execute a transition.

The example below shows the structure of the `main`, `MainInit`, and `MainLoop` functions.

Example 425: Start up structure ––––––––––––––––––––––––––––––––

```
void main ( void )
{
 xMainInit();
 xMainLoop();
}

void xMainInit ( void )
{
  xInitEnv();
  Init of internal data structures in the
  runtime library;
  yInit();
}

void xMainLoop ( void )
{
  while (1) {
    xInEnv(...);
    if ( Timer output is possible )
      SDL_OutputTimerSignal();
    else if ( Process transition is possible )
      Call appropriate PAD function;
  }
}
```
––

The function `xMainLoop` contains an endless loop. The appropriate way to stop the execution of the program is to call the runtime library function `SDL_Halt`. The call of this C function should normally be included in an appropriate task, using the directive `#CODE`. `SDL_Halt` which has the following structure:

```
void SDL_Halt ( void )
{
  xCloseEnv();
  exit(0);
}
```

To complete this overview, which emphasizes the usage of the environment functions, we have to deal with the `xOutEnv` function. Within PAD functions, the runtime library function `SDL_Output` is called to implement outputs of signals. When `SDL_Output` identifies the receiver of a signal to be a process instance that is not part of the current SDL system, `SDL_Output` will call the `xOutEnv` function.

Dynamic Errors

In the library for applications SDL runtime errors will not be reported. The application will just perform some appropriate actions and continue to execute. These actions are in almost all cases the same as the actions at dynamic errors described in the "Dynamic Errors" on page 2121 in chapter 50, *The SDL Simulator*.

- **Output warnings**: If a signal is sent to NULL or to a stopped process instance, or if no receiver is found in an output without a "to" clause, the signal will not be sent, that is, the output statement is a null action. If a signal is sent to a process instance and there is no path between the sender and the receiver, the signal will be sent anyway (actually, no check will be performed).
- If the error was a **decision error**, that is, no path exists for the current decision value, the execution of the program will continue in an **unpredictable way**. To avoid these kind of problems you should always have else paths in decisions (if not all values in the current data type are covered in other paths).
- If the error occurred during an **import** or **view** action, a data area of the correct size containing zero in all positions is returned.
- **No checks** of **assignment** or **index out of range** will be performed. This means that if an array index is out of bounds, then the corresponding C array will be indexed out of its bounds.
- No checks when accessing struct, #UNION, or choice components are performed. No checks are performed when de-referencing a pointer value (Ref generator). These operations will just be executed.

- If the dynamic error occurred **within an SDL expression**, the operator that found the error will return a default value and the evaluation of the expression is continued. The default values returned depend on the result type of the operator and are given in the section "Default Values" on page 2604 in chapter 57, *The Cadvanced/Cbasic SDL to C Compiler*.

Example Section

In this section a complete example of an application is presented. The application is simple but it still contains most of the problems that arise when the Cadvanced SDL to C Compiler is used to generate applications. All source code for this example, together with the running application are delivered with the runtime libraries for application generation. **Note that the example is developed for SunOS 5 and HP-UX. The example is not updated to use encoding and decoding support.**

The Example

We want to develop an application that consists of several communicating UNIX processes. Each UNIX process should also be connected to the keyboard and the screen. When a complete line is typed on the keyboard (when `<Return>` is pressed) in one of the UNIX processes, that line should be sent to and printed by all the UNIX processes, including the one where the line was entered. If a line starting with the character "." is entered in any UNIX process then all the UNIX processes should terminate immediately.

There are some observations we can make from this short description.

- Firstly, all the UNIX processes should behave exactly the same, which means that the behavior can be described in one SDL system and one application can be generated. This application should be able to communicate with other instances of itself and should simultaneously be started in as many instances as we want communicating UNIX processes.
- Secondly, each UNIX process needs access to the terminal. To simplify the connection between a UNIX process and the terminal, we assume that each instance of the application is started from its own window (its own shell tool). In this way the underlying window manager will solve the problem of directing input typed on the key-

board to the correct application, as well as making it possible for you to distinguish between output from different applications.

- Thirdly, the UNIX processes have to be connected so they can communicate with each other. We have decided to use sockets as communication media, and to let the UNIX processes form a ring. This means that when a UNIX process receives a message containing a line, it should print the line and send the message on to the next UNIX process in the ring, if it did not itself originally send the line message.

The SDL System

The SDL system with a behavior as outlined above is very simple. It contains, for example, only one process. The system can receive three types of signals, TermInput from the terminal, and Message and Terminate from the SDL system that is the previous node in the ring. The system will respond by sending Display to the terminal and Message and Terminate to the SDL system next in the ring. The signals TermInput and Display take a line (which is read from the terminal or should be printed on the terminal) as parameter. The signal Message takes a line and a PId value (the original sender in the ring) as parameter, while the signal Terminate takes a PId value (the original sender in the ring) as parameter.

The diagrams for the SDL system may be found in “Appendix C: The SDL System” on page 2747. In the section “Where to Find the Example” on page 2734, references to where to find the source code for this example are given.

Simulating the Behavior

At this stage of the development of the application, when the SDL system is completed but the environment functions are not implemented, it is time to simulate the SDL system to debug it at the SDL level. The runtime library *Simulation* is appropriate in this case for simulation.

There are six cases that should be tested:

- If TermInput (with a line not starting with a period) is sent to the system, it should respond by sending a Message signal to the environment. The first parameter in this signal should be equal to the parameter in the received TermInput signal. The second parameter should be the PId value of the sending process.

- If TermInput (with a line starting with a period) is sent to the system, it should respond by sending a Terminate signal to the environment. The parameter should be the PId value of the sending process.
- If Message (with a PId parameter not equal to the receiving process) is sent to the system, it should respond by sending a copy of the Message signal to the environment. It should also send Display to the environment with the received line as parameter.
- If Message (with a PId parameter equal to the receiving process) is sent to the system, it should respond by just sending a Display signal to the environment with the received line as parameter.
- If Terminate (with a PId parameter not equal to the receiving process) is sent to the system, it should respond by sending a copy of the Terminate signal to the environment. The execution of the program should then terminate.
- If Terminate (with a PId parameter equal to the receiving process) is sent to the system, the program should just stop executing.

Let us now verify that the SDL system behaves according to this. In the two executions of the simulation shown below, the cases described above are tested in the same order as they are listed.

Example 426: Execution Trace of Generated Application ––––––––––

Start program `Phone.sim.sct`:

```
Command : set-trace 6
Default trace set to 6

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : start state
*      Now    : 0.0000
*** NEXTSTATE  idle

Command : output-via
Signal name : TermInput
 Parameter 1 (charstring) : 'hello'
Channel name :
Signal TermInput was sent to PhonePr:1 from env:1
Process scope : PhonePr:1
```

```
Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : idle
*      Input  : TermInput
*      Sender : env:1
*      Now    : 0.0000
*      Parameter(s) : 'hello'
*   DECISION  Value: true
*   DECISION  Value: false
*   OUTPUT of Message to env:1
*      Parameter(s) : 'hello', PhonePr:1
*** NEXTSTATE  idle
Command : output-via TermInput '.' -
Signal TermInput was sent to PhonePr:1 from env:1
Process Scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : idle
*      Input  : TermInput
*      Sender : env:1
*      Now    : 0.0000
*      Parameter(s) : '.'
*   DECISION  Value: true
*   DECISION  Value: true
*   OUTPUT of Terminate to env:1
*      Parameter(s) : PhonePr:1
*** NEXTSTATE  idle

Command : output-via Message
 Parameter 1 (charstring) : 'hello'
 Parameter 2 (pid) : env
Channel name :
Signal Message was sent to PhonePr:1 from env:1
Process scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : idle
*      Input  : Message
*      Sender : env:1
*      Now    : 0.0000
*      Parameter(s) : 'hello', env:1
*   DECISION  Value: false
*   OUTPUT of Message to env:1
*      Parameter(s) : 'hello', env:1
*   OUTPUT of Display to env:1
*      Parameter(s) : 'hello'
```

```
*** NEXTSTATE  idle

Command : **output-via Message**
 Parameter 1 (charstring) : ‘hello’
 Parameter 2 (pid) : PhonePr:1
Channel name :
Signal Message was sent to PhonePr:1 from env:1
Process scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : idle
*      Input  : Message
*      Sender : env:1
*      Now    : 0.0000
*      Parameter(s) : ‘hello’, PhonePr:1
*   DECISION  Value: true
*   OUTPUT of Display to env:1
*      Parameter(s) : ‘hello’
*** NEXTSTATE  idle

Command : output-via Terminate
 Parameter 1 (pid) : env
Channel name :
Signal Terminate was sent to PhonePr:1 from env:1
Process scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : idle
*      Input  : Terminate
*      Sender : env:1
*      Now    : 0.0000
*      Parameter(s) : env:1
*   DECISION  Value: false
*   OUTPUT of Terminate to env:1
*      Parameter(s) : env:1
*   TASK  Halt
```

Example 427 ––

Start program `Phone.sim.sct`:

```
Command : set-trace 6
Default trace set to 6

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : start state
*      Now    : 0.0000
*** NEXTSTATE  idle

Command : output-via Terminate
 Parameter 1 (pid) : PhonePr:1
Channel name :
Signal Terminate was sent to PhonePr:1 from env:1
Process scope : PhonePr:1

Command : next-transition

*** TRANSITION START
*      PId    : PhonePr:1
*      State  : idle
*      Input  : Terminate
*      Sender : env:1
*      Now    : 0.0000
*      Parameter(s) : PhonePr:1
*   DECISION  Value: true
*   TASK  Halt
```

By running the system with the SDL monitor, as in the examples above, you may debug the system at the SDL level. The overall behavior of the system can thus be tested.

It is possible to start two instances of the simulation and have the simulators communicate with each other. Then Message and Terminate signals sent to the environment in one of the simulations will appear as signals coming from the environment in the other.

Note:

Do not forget the monitor command Start-SDL-Env to make the simulation programs start communicating.

The Environment

In the environment functions we use the socket facility in UNIX to implement the communication between the executing programs. **In the current example, the implementation is developed for SunOS 5 and HP-UX.**

To simplify the example we assume that each instance of the application is started in a window of its own (a shell tool window under for instance X Windows, where UNIX commands can be entered). This means that we will have no problems with the interpretation of `stdin` and `stdout` in the programs.

The name of the socket for incoming messages for a certain instance of the application will be the string "Phone" concatenated with the UNIX process number for the current program. The socket will be created in the directory `/tmp`. Each application instance will print this number during the initialization and will then ask for the process number of the application instance where it should send its messages. You have to enter these numbers in such a way as to form a ring among the applications.

The Environment Functions

The environment functions, which may be found in the file `PhoneEnv.c`, are shown in section "Appendix D: The Environment Functions" on page 2750. The file is developed according to the structure discussed in the previous part of this chapter and uses the system interface header file generated from the SDL system.

As the `PhoneEnv.c` file includes `scttypes.h` and uses some C macros, it should be compiled using the same compiler options as the C file for the SDL system. For information about how to extend the generated make file to handle also non-generated files, please see "Makefile Options" on page 122 in chapter 2, *The Organizer*.

In the code for the environment functions a number of UNIX functions are used. Their basic behavior is described below. For any details please see the UNIX manuals available from Sun Microsystems.

Function name	Functionality
`getpid`	Returns the UNIX process number for the current program.
`socket`	Returns a new, unnamed socket.
`bind`	Binds a socket to a name in the file system.
`listen`	Starts listen for other programs trying to connect to this socket.
`connect`	Should be called by other programs that want to establish a connection to the current socket.
`accept`	Accepts a connection request.

Function name	Functionality
`select`	Returns 1 if anything readable can be found in any of the specified file descriptors, where a file descriptor can represent a file, a socket, and the terminal (`stdin` and `stdout`).
`read, write`	Reads or writes on a file (a file descriptor).
`close`	Closes a file.
`unlink`	Removes a file.

If we now look at the code for the environment functions (see “Appendix D: The Environment Functions” on page 2750), we see that `xInitEnv` mainly performs the following actions:

- It creates two unnamed sockets, one for message in (`Connection_Socket`) and one for messages out (`Out_Socket`).
- It binds `Connection_Socket` to the file system (in the `/tmp` directory) and starts listen for connections.
- It prints its UNIX process number.
- It reads the UNIX process number of the process where to send messages.
- It attempts to connect to the socket of the process where to send messages.
- It accepts the connection from the process that will send messages here.

In `xCloseEnv` the created sockets are closed and removed.

The `xInEnv` and `xOutEnv` functions follow the guidelines for these functions given in the reference section. In `xInEnv` the select function is used to determine if any messages are ready to be received from the terminal (`stdin`) or from the incoming socket. An available message is then read and the information is converted to an SDL signal, which is sent to the SDL system using the `SDL_Output` function. In `xOutEnv` a test on the NameNode in the signal is used to determine the signal type. Depending on the signal type the appropriate information is written either on the outgoing socket or on the terminal (`stdout`).

Debugging

The first part of the debugging activity is, of course, when the SDL system is simulated and examined through the monitor system. Now we also want to include the environment functions during debugging. The intention of the library *ApplicationDebug* is to use the monitor and the environment functions together.

When the environment functions (`xInEnv`) read information from the keyboard there is, however, a problem in using `xInEnv` together with the monitor. In our system, for instance, a line typed on the keyboard may either be a monitor command or a line typed to the SDL system. As both the monitor and `xInEnv` are polling for lines from `stdin`, the interpretation of a typed line depends on which one first finds the line.

A better way is to eliminate this indeterministic behavior by not polling for typed lines in `xInEnv`. Instead, you may use the monitor command:

```
Output-Via TermInput 'the line'
```

to simulate a line typed on the keyboard. In this way all the other parts of the environment functions can be tested under the monitor. If you enclose the sections in `xInEnv` handling keyboard polling between `#ifndef XMONITOR` and `#endif` this code is removed when the monitor is used; that is if the library *ApplicationDebug* is used (see the code for `xInEnv` in "Appendix D: The Environment Functions" on page 2750).

A C source code debugger is of course also useful when debugging the environment functions. The initialization phase, `xInitEnv`, is probably the most difficult part to get working correctly in our system. All the source code for this function is available, and a C debugger can be used.

While debugging generated code from SDL at the C level, it is always easy to find the currently executing SDL symbol, by using the SDT references (see "Syntax" on page 911 in chapter 19, *SDT References*) in the C code and the *Go To Source* menu choice in the *Edit* menu in the Organizer. For more details please see "Go To Source" on page 99 in chapter 2, *The Organizer*.

Running the Application

To have an application of the Phone system you now only need to make a new executing program with the library *Application*.

When you run the Phone system, start the program from two (or more) shell tools **(on UNIX)**. Each instance of the program will then print:

```
My Pid: 2311
Connect me to:
```

You should answer these questions in such a way that a ring is formed by the programs. When the initialization is completed for a program it prints:

```
******** Welcome to SDT Phone System ********
phone ->
```

The program is now ready to receive lines printed on the keyboard and messages sent from other programs. A Display signal received from another program is printed as follows:

```
display -> the line received in Display signal
```

Where to Find the Example

All files concerning this example may be found in the directory:

`<`*`installation directory`*`>/sdt/examples/phone`

Use these files if you only want to look at the source files and if you are using a Sun workstation you could try the executing versions of the program. Otherwise you should copy the files to one of your own directories. Please be sure not to change the original files.

In the directory you will find the following files:

File name	**Purpose**
`Phone.sdt`	The system file
`Phone.ssy`	Represents the SDL system
`PhoneBl.sbk`	Represents the SDL block
`PhonePr.spr`	Represents the SDL process
`phone.pr`	The generated PR file after GR to PR
`phone.c`	The generated C file after C code generation
`phone.ifc`	The generated `.ifc` file

File name	Purpose
`PhoneEnv.c`	Contains the environment functions
`Phone.m`	The make file for HP-UX
`Phone.solaris.m`	The make file for SunOS 5

Appendix A: Formats for ASCII

The ASCII encoding function, ASCII_ENCODE, encodes the SDL signal parameters and variables as ASCII characters before adding into the buffer. The output into the buffer is illustrated with a number of examples.

Braces { } are used for most data types and show where the data types start and stop. Signal parameters start and stop with braces.

Comma is used to delimit.

For more information about data types see "Using SDL Data Types" on page 42 in chapter 2, *Data Types*.

Array, CArray

`Array` is used to define a fixed number of elements.

`Array` takes two generator parameters, an index sort and a component sort.

Example 428: Using Array ––

```
newtype A1 Array(b, Integer) endnewtype;

dcl Var_Array A1;

task Var_Array := (. 3 .);
```

Output to the buffer: {3,3,3}

Bag

`Bag` is almost the same as `Powerset`. The only difference is that `Bag` contains the same value several times. `Bag` can be used as an abstraction of other data types.

`Bag` takes one generator parameter, the item sort.

Example 429: Using Bag ––

```
newtype B1 Bag(Integer) endnewtype;

dcl Var_Bag B1;
```

```
task Var_Bag := (. 7, 4, 7 .);
```

Output to the buffer: {2:7,1:4}

Example 430: Using Bag (old-style SDL operator code generation)

```
newtype B1 Bag(Integer) endnewtype;

dcl Var_Bag B1;

task Var_Bag := Incl(7, Incl(4, Incl(7, Empty)));
```

Output to the buffer: {2:7,1:4}

Bit

`Bit` can only take two values, 0 and 1.

Example 431: Using Bit

```
dcl Var_Bit Bit;

task Var_Bit := 1;
```

Output to the buffer: 1

Bit_String

`Bit_String` is used to represent a sequence of bits.

Example 432: Using Bit_String

```
dcl Var_Bit Bit_String;

task Var_Bit := Mkstring(I20(1101));
```

Output to the buffer: ‘1101’

Boolean

`Boolean` can only take two values, False and True.

Example 433: Using Boolean

```
dcl Var_Boolean Boolean;
```

```
task Var_Boolean := TRUE;
```

Output to the buffer: T

If `Var_Boolean` is set to FALSE, the output to the buffer will be F.

Character

`Character` is used to represent the ASCII characters.

Example 434: Using Character

```
dcl Var_Character Character;

task Var_Character := ‘M’
```

Output to the buffer: M

CharStar, PId, UnionC, Userdef, VoidStar, VoidStarStar

The user has to define encoding/decoding procedures for these types, see “Appendix B: User defined ASCII encoding and decoding” on page 2745. If no encoding/decoding procedure is defined, then there will be no output to buffer.

For threaded integrations, the PId value will be encoded/decoded by coding the memory address as an integer.

CharString, IA5String, NumericString, PrintableString, VisibleString

These string types are used to represent sequences of characters.

Example 435: Using Charstring

```
dcl Var_Charstring Charstring;

task Var_Charstring := ‘Hello world’
```

Output to the buffer: ‘Hello world’

Choice, Union

`Choice` represents the ASN.1 concept CHOICE and can also be seen as a C *union* with an implicit tag field.

Example 436: Using Choice

```
newtype C Choice
  c1 Character;
  c2 Boolean;
endnewtype;

dcl Var_Choice C;

task Var_Choice := c2:true;
```

Output to the buffer: {1,T}, where 1 is an implicit tag.

Duration, Time

`Time` is used to denote ‘a point in time’ and `Duration` is used to denote ‘a time interval’.

A value of sort `Time` represents a point of time in the real world. The `Time` unit is usually 1 second.

Example 437: Using Time

```
dcl Var_Time Time;

task Var_Time := 1;
```

Output to the buffer: {1,0}. The first field is seconds and the second field is nano-seconds.

Enum

`Enum` contains only the values enumerated in a sort.

Example 438: Using Enum

```
newtype E /*Enum*/
  literals e1, e2, e3;
endnewtype;

dcl Var_Enum E;

task Var_Enum := e2;
```

Output to the buffer: 1

Float

`Float` is equivalent to *float* in C.

Example 439: Using Float ––––––––––––––––––––––––––––––––––––

```
dcl Var_Float Float;

task Var_Float := 3.14159;
```
–––

Output to the buffer: 3.1415901e0 (always 7 decimals)

GPowerset, Object_Identifier, String

`GPowerset` is used as an abstraction of other data types. `GPowerset` takes one generator parameter, the item sort.

`Object_Identifier` is a sequence of `Natural` values.

`String` can be used to build lists of items of the same type. `String` has two generator parameters, the component sort and the name of an empty string.

Example 440: Using GPowerset –––––––––––––––––––––––––––––––––

```
newtype GP Powerset(Integer) endnewtype;

dcl Var_GPowerset GP;

task Var_GPowerset := Incl(7, Incl(4, Empty));
```
–––

Output to the buffer: {3,4}

`Object_Identifier` and `String` are used in the same way as `GPowerset.`

Inherits, Syntype

`Syntype` and `Inherits` create a new type, that has the same properties as an existing type, by inheriting the type or make a syntype of the type.

Example 441: Using Syntype –––––––––––––––––––––––––––––––––––

```
syntype newinteger = Integer endsyntype;

dcl Var_Newinteger newinteger;
```

```
   task Var_Newinteger := 2
```

Output to the buffer: 2

Integer, LongInt, ShortInt, UnsignedInt, UnsignedLongInt, UnsignedShortInt, Natural

`Integer` is used to represent mathematical integers.

`Natural` is a syntype of `Integer`.

Example 442: Using Integer

```
   dcl Var_Integer Integer;

   task Var_Integer := 1
```

Output to the buffer: 1

Null

The sort `Null` only contains one value, Null.

Example 443: Using Null

```
   dcl Var_Null Null;

   task Var_Null := Null;
```

Output to the buffer: 0

Octet

`Octet` is used to represent eight-bit values.

Example 444: Using Octet

```
   dcl Var_Octet Octet;

   task Var_Octet := I2O(12);
```

Output to the buffer: 0c

Octet_String

`Octet_String` represents a sequence of `Octet` values.

`Octet_String` always contains an equal number of characters, since every octet takes two characters.

Example 445: Using Octet_String ——————————————————

```
dcl Var_OctetString Octet_String;

task Var_OctetString := Mkstring(I20(12));
```

Output to the buffer: ‘0c’

ORef, Own, Ref

`ORef, Own` and `Ref` are used to define pointer types.

Example 446: Using Ref ——————————————————

```
newtype R Ref(r1) endnewtype;

dcl Var_Ref R;

task Var_Ref := (. (. 1, 2, ‘Telelogic’ .) .);
```

Output to the buffer: {{1,2,’Telelogic’}}

Powerset

`Powerset` takes one generator parameter, the item sort, and implements a powerset over that sort. `Powerset` can be used as an abstraction of other data types.

Example 447: Using Powerset ——————————————————

```
newtype P Powerset(p1) endnewtype;

dcl Var_Powerset P;

task Var_Powerset := (. 4, 3 .);
```

Output to the buffer: ‘00110000000000000000000000000000’

The bits are an equal multiple of `sizeof(unsigned long)`.

Example 448: Using Powerset (old-style SDL operator code generation) ——————————————————

```
newtype P Powerset(p1) endnewtype;
```

```
dcl Var_Powerset P;

task Var_Powerset := Incl(4, Incl(3, Empty));
```
––

Output to the buffer: ‘00110000000000000000000000000000’

The bits are an equal multiple of `sizeof(unsigned long)`.

Real

`Real` is used to represent the mathematical real values.

Example 449: Using Real ––––––––––––––––––––––––––––––––

```
dcl Var_Real Real;

task Var_Real := 1.0;
```
––

Output to the buffer: 1.0000000000000e0 (always 13 decimals)

SignalId

`SignalId` is used to describe the signal ID.

`SignalId` is a sequence with characters.

Example 450: Using SignalId ––––––––––––––––––––––––––––––

```
dcl Var_SignalId SignalId;

task Var_SignalId := ‘Sig1’;
```
––

Output to the buffer: ‘Sig1’

Struct

`Struct` can be used to make an aggregate of data that belong together.

Example 451: Using Struct –––––––––––––––––––––––––––––––

```
newtype S struct
  s1 integer;
  s2 integer;
  s3 charstring;
endnewtype;
```

```
  dcl Var_Struct S;

  task Var_Struct := (. 1, 2, ‘Telelogic’ .);
```

Output to the buffer: {1,2,’Telelogic’}

Appendix B: User defined ASCII encoding and decoding

The types CharStar, PId, UnionC, Userdef, VoidStar, VoidStarStar do not have a natural transformation to ASCII format. The desired encoding probably differs from application to application. The ASCII encoding procedure and the ASCII decoding procedures do not encode or decode these types, but they can invoke user written procedures for encoding and decoding them.

For threaded integrations, the PId value will be encoded/decoded by coding the memory address as an integer.

If you want to add encoding for these types, then do the following steps:

- Implement a C-function for encoding with input and output parameters compatible with tEncodeFunc, which is declared in file “coderucf.h”. The encode functions in “coderascii.c” can be used as an example.
- Set static variable AsciiUserEncode to your encode function in xInitEnv.

If you want to add decoding for these types, then do the following steps:

- Implement a C-function for decoding with input and output parameters compatible with tDecodeFunc, which is declared in file “coderucf.h”. The decode functions in “coderascii.c” can be used as an example.
- Set static variable AsciiUserDecode to your decode function in xInitEnv.

Example 452 User defined ASCII encoding ––––––––––––––––––––––

```
int MyAsciiEncoder( tBuffer Buf,
                    tSDLTypeInfo* TypeNode,
                    void* Value )
{
  /* my error handling code for one or more of
     CharStar, PId, UnionC, Userdef, VoidStar, VoidStarStar */
  /* return 1 if it was succesful,
     return 0 if it failed */
}

In xInitEnv:
  AsciiUserEncode = MyAsciiEncoder;
```

Appendix C: The SDL System

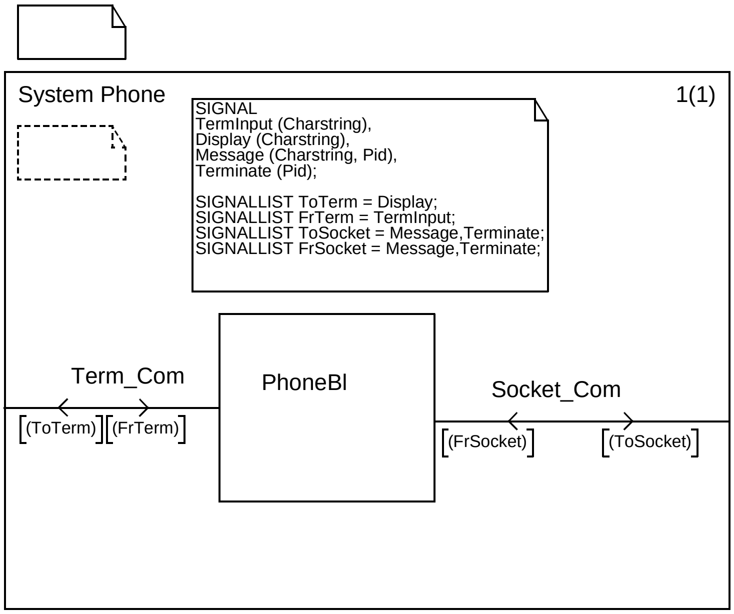
System Phone
1(1)
SIGNAL
TermInput (Charstring),
Display (Charstring),
Message (Charstring, Pid),
Terminate (Pid);
SIGNALLIST ToTerm = Display;
SIGNALLIST FrTerm = TermInput;
SIGNALLIST ToSocket = Message,Terminate;
SIGNALLIST FrSocket = Message,Terminate;
Term_Com
[(ToTerm)]
[(FrTerm)]
PhoneBl
Socket_Com
[(FrSocket)]
[(ToSocket)]

Figure 492: The system Phone

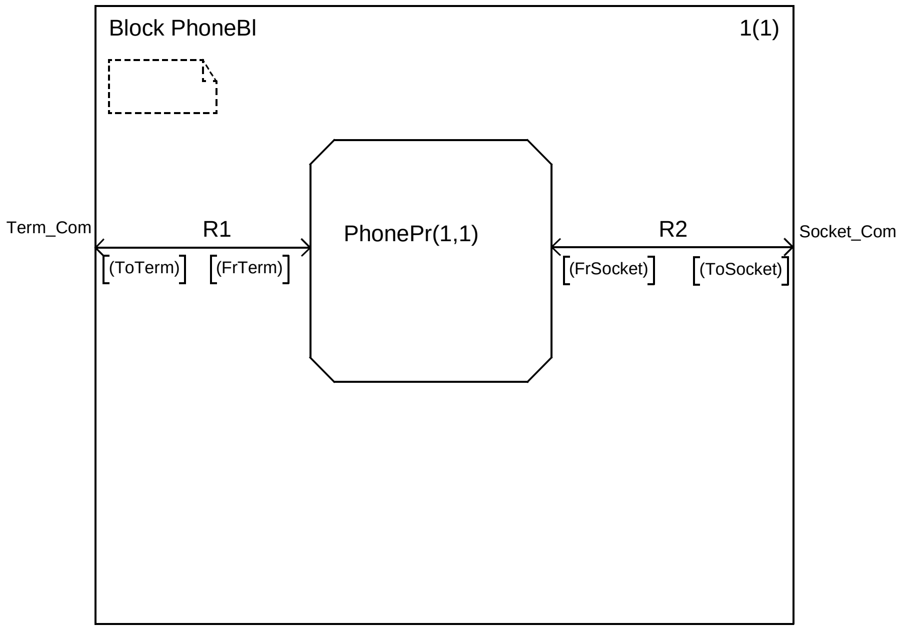

Figure 493: The block PhoneBl

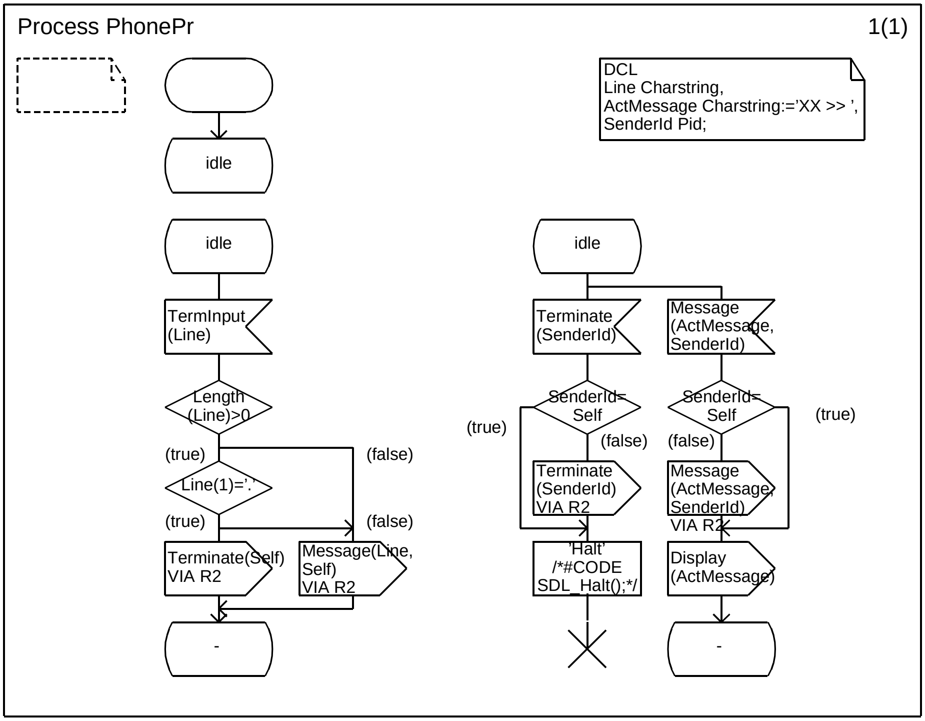

Figure 494: The process PhonePr

Appendix D: The Environment Functions

This section contains the environment functions included in the example. Note that this example is not updated to use the ASCII encoder.

```
/****+**********************************************************
00   sctEnv.c for SimplePhoneSys
*************************************************************/
#include "scttypes.h"
#include <stdio.h>

#include "phone.ifc"

#include <sys/types.h>
#include <sys/time.h>
#include <sys/socket.h>
#ifdef AIXV3CC
#include <sys/select.h>
#endif
#include <sys/un.h>

#include <unistd.h>
#define getdtablesize() ( (int) sysconf(_SC_OPEN_MAX) )

int Out_Socket, In_Socket;
struct sockaddr_un  Connection_Socket_Addr;
struct sockaddr_un  Connected_Socket_Addr;

#ifdef ULTRIXCC
#define PRINTF(s) \
  printf(s); \
  /* flush output to get a prompt */ \
  fflush( stdout)
#else
#define PRINTF(s) printf(s)
#endif

#ifdef XENV

/*#if !defined(XPMCOMM) && defined(XENV)*/
/*---+---------------------------------------------------------
     xGlobalNodeNumber  extern
-------------------------------------------------------------*/
#ifndef XNOPROTO
 int
xGlobalNodeNumber( void )
#else
 int
xGlobalNodeNumber()
#endif
{
  static int ProcId = -1;

  if (ProcId < 0)
    ProcId = getpid();
  return (ProcId);
}
/*#endif*/
```

```
/*---+---------------------------------------------------------
     xInitEnv  extern
-----------------------------------------------------------*/
#ifndef XNOPROTO
 void
xInitEnv( void )
#else
 void
xInitEnv()
#endif
{
  fd_set  readfds;
  int  addr_size;
  int  Connection_Socket;
  char  TmpStr[132];
  struct timeval  t;

  t.tv_sec = 60;
  t.tv_usec = 0;

  if ( (Connection_Socket = socket(PF_UNIX,SOCK_STREAM,0)) < 0
) {
    PRINTF("\nError: No Connection_Socket available!\n");
    SDL_Halt();
  }
  if ( (Out_Socket = socket(PF_UNIX,SOCK_STREAM,0)) < 0 ) {
    PRINTF("\nError: No Out_Socket available!\n");
    SDL_Halt();
  }

  sprintf(Connection_Socket_Addr.sun_path,
          "/tmp/Phone%d", xGlobalNodeNumber());
  Connection_Socket_Addr.sun_family = PF_UNIX;
  if ( 0 > bind(Connection_Socket, &Connection_Socket_Addr,
           strlen(Connection_Socket_Addr.sun_path)+2) ) {
    PRINTF("\nError: Bind did not succeed!\n");
    SDL_Halt();
  }
  listen(Connection_Socket, 3);

  sprintf(TmpStr, "\nMy Pid: %d\n", xGlobalNodeNumber());
  PRINTF(TmpStr);
  PRINTF("\nConnect me to: ");

  FD_ZERO(&readfds);
  FD_SET(1,&readfds);
  FD_SET(Connection_Socket,&readfds);
  if ( 0 < select(getdtablesize(), &readfds,
                  (fd_set*)0, (fd_set*)0, &t) ) {
    if ( FD_ISSET(1, &readfds) ) {
      (void)gets(TmpStr);
      sscanf(TmpStr, "%s", TmpStr);
      sprintf(Connected_Socket_Addr.sun_path, "/tmp/Phone%s",
              TmpStr);
      Connected_Socket_Addr.sun_family = PF_UNIX;
      if (connect(Out_Socket, (struct sockaddr
*)&Connected_Socket_Addr,
              strlen(Connected_Socket_Addr.sun_path)+2) < 0) {
        PRINTF("Error from connect\n");
        SDL_Halt();
      }
      FD_ZERO(&readfds);
      FD_SET(Connection_Socket,&readfds);
      if ( 0 < select(getdtablesize(), &readfds, (fd_set*)0,
                      (fd_set*)0, &t) ) {
        if ( FD_ISSET(Connection_Socket, &readfds) ) {
          addr_size =
strlen(Connection_Socket_Addr.sun_path)+2;
          In_Socket = accept(Connection_Socket,
&Connection_Socket_Addr,
                             &addr_size);
        }
      } else {
        PRINTF("\nError: Timed out\n");
        SDL_Halt();
      }
    }
    else if ( FD_ISSET(Connection_Socket, &readfds) ) {
      addr_size = strlen(Connection_Socket_Addr.sun_path)+2;
      In_Socket = accept(Connection_Socket,
&Connection_Socket_Addr,
                         &addr_size);
      FD_ZERO(&readfds);
      FD_SET(1,&readfds);
      if ( 0 < select(getdtablesize(), &readfds,
                      (fd_set*)0, (fd_set*)0, &t) ) {
        if ( FD_ISSET(1, &readfds) ) {
          (void)gets(TmpStr);
          sscanf(TmpStr, "%s", TmpStr);
          sprintf(Connected_Socket_Addr.sun_path,
"/tmp/Phone%s",
                  TmpStr);
          Connected_Socket_Addr.sun_family = PF_UNIX;
          if (connect(Out_Socket, (struct sockaddr
*)&Connected_Socket_Addr,
                  strlen(Connected_Socket_Addr.sun_path)+2) <
0) {
            PRINTF("Error from connect\n");
            SDL_Halt();
          }
        }
      } else {
        PRINTF("\nError: Timed out\n");
        SDL_Halt();
      }
    }
  } else {
    PRINTF("\nError: Timed out\n");
    SDL_Halt();
  }

  PRINTF("\n\n************ Welcome to SDT Phone System
************\n");
  PRINTF("\nphone -> ");
}

/*---+---------------------------------------------------------
     xCloseEnv  extern
-------------------------------------------------------------*/
#ifndef XNOPROTO
 void
xCloseEnv( void )
#else
 void
xCloseEnv()
#endif
{
  close(Out_Socket);
  close(In_Socket);
  unlink(Connected_Socket_Addr.sun_path);
  unlink(Connection_Socket_Addr.sun_path);
  PRINTF("\nClosing this session.\n");
}
```

```
/*---+---------------------------------------------------------
     xInEnv  extern
-------------------------------------------------------------*/
#ifndef XNOPROTO
 void
xInEnv( SDL_Time  Time_for_next_event )
#else
 void
xInEnv( Time_for_next_event )
  SDL_Time  Time_for_next_event;
#endif
{
  struct timeval t;
  fd_set         readfds;
  char          *Instr;
  int            NrOfReadChars;
  char           SignalName = '\0';
  xSignalNode    yOutputSignal;
  int            i = 0;
  char           chr = '\0';

  t.tv_sec = 0;
  t.tv_usec = 1000;
  FD_ZERO(&readfds);
#ifndef XMONITOR
  FD_SET(1,&readfds);
#endif
  FD_SET(In_Socket,&readfds);
  if ( select(getdtablesize(),&readfds,0,0,&t) > 0 ) {
#ifndef XMONITOR
    /*SDL-signal TermInput */
    if FD_ISSET(1, &readfds) {
      Instr = (char *)xAlloc(132);
      Instr[0]='L';
      Instr++;
      (void)gets(Instr);
      yOutputSignal = xGetSignal(TermInput, xNotDefPId, xEnv);
      xAss_SDL_Charstring(
        &((yPDP_TermInput)(OUTSIGNAL_DATA_PTR))->Param1, --
Instr,XASS);
      SDL_Output(yOutputSignal, (xIdNode *)NIL);
      xFree((void**)&Instr);
    }
#endif
    if FD_ISSET(In_Socket, &readfds) {
      Instr = (char *)xAlloc(151);
      do {
        read(In_Socket, &chr, 1);
        Instr[i++] = chr;
      } while ( chr!='\0' );
      sscanf(Instr, "%c", &SignalName);

      if ( SignalName == 'M' ) {
        /* SDL-signal Message */
        yOutputSignal = xGetSignal(Message, xNotDefPId, xEnv);
        sscanf(
          Instr+1,
          "%d %x%n",
          &(((yPDP_Message)(OUTSIGNAL_DATA_PTR))-
>Param2.GlobalNodeNr),
          &(((yPDP_Message)(OUTSIGNAL_DATA_PTR))-
>Param2.LocalPId),
          &NrOfReadChars);
        xAss_SDL_Charstring(
          &((yPDP_Message)(OUTSIGNAL_DATA_PTR))->Param1,
          (Instr+NrOfReadChars+2),XASS);
        SDL_Output(yOutputSignal, (xIdNode *)NIL);
      }
      else if ( SignalName == 'T' ) {
        /* SDL-signal Terminate */
        yOutputSignal = xGetSignal(Terminate, xNotDefPId,
xEnv);
        sscanf(
          Instr+1,
          "%d %x",
          &(((yPDP_Terminate)(OUTSIGNAL_DATA_PTR))-
>Param1.GlobalNodeNr),
          &(((yPDP_Terminate)(OUTSIGNAL_DATA_PTR))-
>Param1.LocalPId));
        SDL_Output(yOutputSignal, (xIdNode*)0);
      }
      xFree((void**)&Instr);
    }
  }
}

/*---+---------------------------------------------------------------
     xOutEnv  extern
-----------------------------------------------------------------*/
#ifndef XNOPROTO
 void
xOutEnv( xSignalNode  *S )
#else
 void
xOutEnv( S )
  xSignalNode  *S;
#endif
{
  char   Outstr[150];

  /* SDL-signal Message */
  if ( (*S)->NameNode == Message ) {
    sprintf(Outstr,
            "M %d %x %.*s",
            ((yPDP_Message)((*S)))->Param2.GlobalNodeNr,
            ((yPDP_Message)((*S)))->Param2.LocalPId,
            strlen(((yPDP_Message)((*S)))->Param1),
            ((yPDP_Message)((*S)))->Param1);
    write(Out_Socket, Outstr, strlen(Outstr)+1);
    xReleaseSignal(S);
    return;
  }
  /* SDL-signal Terminate */
  if ( (*S)->NameNode == Terminate) {
    sprintf(Outstr,
            "T %d %x",
            ((yPDP_Terminate)((*S)))->Param1.GlobalNodeNr,
            ((yPDP_Terminate)((*S)))->Param1.LocalPId);
    write(Out_Socket, Outstr, strlen(Outstr)+1);
    xReleaseSignal(S);
    return;
  }
  /* SDL-signal Display */
  if ( (*S)->NameNode == Display ) {
    sprintf(Outstr, "\ndisplay ->%.*s",
           strlen(((yPDP_Display)((*S)))->Param1),
           ((yPDP_Display)((*S)))->Param1+1);
    PRINTF(Outstr);
    PRINTF("\nphone -> ");
    xReleaseSignal(S);
    return;
  }
}
#endif
```