Chapter

65

Integration with Operating Systems

This chapter describes how to integrate code generated by the Cadvanced SDL to C Compiler with operating systems.

The different integration models (Light Integration, Threaded Integration and Tight Integration) are explained in the introduction.

There is a separate annex for each supported RTOS where the OS specific parts are described.

Important!

You are free to reuse the integrations supplied by Telelogic or modify them to your needs. The OS integrations are tested by Telelogic but Telelogic does not guarantee that they will perform in your specific target environment (hardware, CPU, RTOS version etc.). Please refer to "RTOS integrations" on page 9 in chapter 1, *Platforms and Products, in the Installation Guide* for information about host and target environments where the integration types have been developed and tested.

Light and Threaded integrations

The Light and Threaded integrations are included in the delivery.

The standard product support and maintenance agreement **only** includes support for the Light and Threaded integrations available from Telelogic if **no changes** have been made to the integrations.

Tight Integration

The Tight integration is meant to serve as a **template**, to be adapted to your needs. It is available for you as a free download from the Telelogic Support web site.

The standard product support and maintenance agreement **does not** assist in adapting to your target environment.

Customer specific OS integrations

All OS integration models can be supported, enhanced and customized by using Telelogic's Professional Services.

Other integrations

Telelogic has developed a large number of integrations based on the company's vast experience of integrating with all operating systems on the market.

Introduction

The code that is generated by the Cadvanced SDL to C compiler is designed to run on different platforms. This is done by letting all platform dependent concepts be represented by C macros that can be expanded appropriately for each environment. There are also types used in the generated code that have to be defined. Integration, as referred to in this chapter, is the process of adapting the generated code to a certain platform.

This chapter describes the different models that are supported in Telelogic Tau.

Note:

Throughout this chapter, annexes excluded, VxWorks terminology has been used whenever there are differences between operating systems. Particularly, this means that the term *task* has been used on several occasions. The corresponding term would be *thread* for Win32 and Solaris, *process* for OSE Delta, and *task* for Nucleus.

Different Integration Models

With Cadvanced, there are three different run-time models, called Light Integration, Threaded Integration and Tight Integration. Tight integrations are then divided into two submodels, the Standard model and the Instance-Set model. All models use the same generated code.

You will find descriptions of the different models below, as well as guidelines for choosing between them.

Light integration

The simplest case is called a Light integration because only a minimum of interaction with the operating system is required; a Light integration could even run on a target system without any OS at all.

The complete SDL system runs as a single OS task. Scheduling of SDL processes is handled by the standard kernel, running a complete state transition at a time (no preemption). Worst case scheduling latency is thus the duration of the longest transition.

Communication between SDL processes is handled by the kernel; communication between the SDL system and the environment is handled in the two user supplied functions `xInEnv`() and `xOutEnv`(). These functions are platform dependent. See Figure 558.

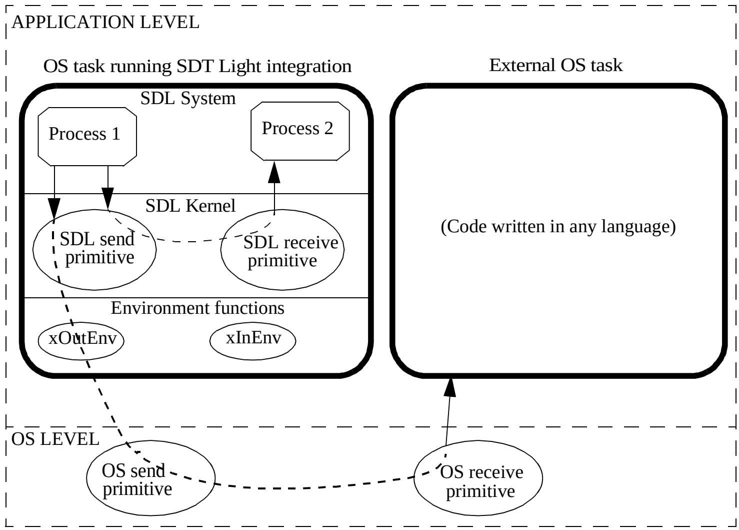


Figure 558: Signalling in the Light Integration Model.

An SDL process sends one internal and one external signal. `xOutEnv`*() must be written to call the OS send primitive. Thick borders denote OS tasks.*

Other properties of a Light Integration:

- The scheduling order can be controlled by process priorities and/or signal priorities as set by the `#PRIO` directive.
- An SDL system can be partitioned, i.e. split into several executables. Each partition has its own kernel/scheduler and set of environment functions. Partitioning is explained in *“Partitioning” on page 2571 in chapter 57, The Cadvanced/Cbasic SDL to C Compiler*.
- It is easy to “go to target”: Write the environment functions and recompile the standard kernel with a cross compiler.

Threaded Integration

The main difference between a Light integration and a Threaded integration is that any part of the SDL system can execute in its own thread in a Threaded integration. A thread in a Threaded integration can execute one or several SDL processes or even blocks. How the different SDL objects should be mapped to threads is specified in the Deployment Editor. The user can specify thread-specific parameters like: STACKSIZE, PRIORITY, MAXMESSAGEQUEUESIZE and MAXMESSAGESIZE in the Deployment Editor. The Deployment Editor works together with the Targeting Expert to generate a Threaded integration.

Note:

A Threaded integration can only be generated using the Deployment Editor and the Targeting Expert, i.e. the make feature in the Organizer CANNOT be used.

Communication and execution control in one thread is handled by the SDL kernel and not the OS kernel. See Figure 559. This model shows the default Threaded integration where OS semaphores are the only OS primitives used in the signal sending. The `xMainLoop()` function is the entry point for each thread.

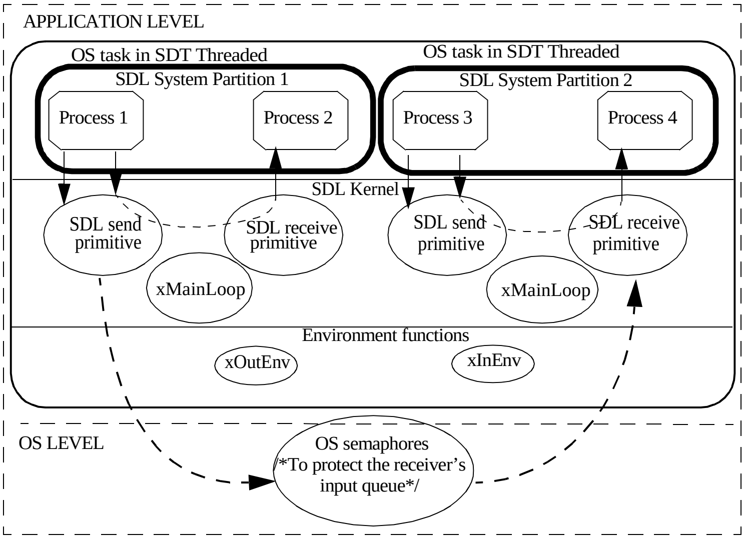


Figure 559: Signalling in Threaded Integration.

SDL Kernel functions are used for sending signals within or between threads. Thick borders denote threads.

Semaphores are used frequently in Threaded to protect globally accessible data like input queues and export queues. Semaphores are also used in the start-up to synchronize the execution of newly created threads. We must ensure that no thread is allowed to start executing until ALL threads have been created in the start-up phase.

Communication with external threads (Threads not made from SDL specifications) is handled by the environment functions in the same way as for a Light integration. There is though one major difference and that is that it is possible in a Threaded to send signals directly to an SDL process with the `SDL_Output()` function. The `SDL_Output()` function is "thread-safe" because of the use of OS semaphores.

Variations of Threaded Integration

There are two different implementation of signal sending between threads in Threaded. The default model send signals in the same way as in a Light integration but the input queues are protected by OS semaphores. In the alternative model, the signals are sent/received by OS_Send and OS_Received. In the alternative model there is no use for OS semaphores. Which model to use can be decided at compilation time by setting the compilation switch `THREADED_ALTERNATIVE_SIGNAL_SENDING`.

Threaded Default

The sender of an SDL signal "takes" a semaphore before accessing the receivers input queue. The signal is then linked into the input queue and the semaphore is "given" back.

The receiver is normally "waiting" for a semaphore. When a signal is sent, the semaphore is "released" by the sender. To send a signal, two semaphores are normally needed. One semaphore is protecting the input queue and the other is used for synchronizing the sender and the receiver. In Solaris, where we use POSIX threads and semaphores they are called: xInputPortMutex and xInputPortCond.

One OS feature that is needed in Threaded is a "Conditional Wait". We are only allowed to wait for a signal until the first internal timer expires. In POSIX, there is a primitive called `pthread_cond_wait()` and in Windows there is a similar concept called `WaitForSingleObject()` where a time-out can be specified. In VxWorks and OSE, the only concept where you can specify a time-out is OS_Receive. For synchronization between the sender and receiver we are sending a small OS_Message (the character ‘c’) in these two RTOS.

Threaded using OS Send and OS Receive

In this alternative implementation we are sending the signals using OS_Send and OS_Receive. Signals are now sent to OS_Message queues. The sender sends the pointer to the SDL signal in an OS_Message to the receiver’s OS_Queue. When the signal is received, the SDL signal is unpacked and linked in to the receivers input queue. The advantage with this implementation is that there is now synchronization between the sender and the receiver. Any number of threads can send signals at the same time to a specific receiver without having to wait for a semaphore.

Tight integration

We also provide an alternate run-time model, which is called a Tight integration because the generated code interacts directly with the underlying operating system when creating processes, sending signals, etc.

The SDL processes run as separate OS tasks as explained below. Scheduling is handled by the OS and is normally time-sliced, priority based and preemptive.

Communication takes place using the inter-process communications mechanisms offered by the OS, normally message queues. This applies to signals sent between SDL processes as well as signals sent to or received from the environment. There are no environment functions, as illustrated in Figure 560.

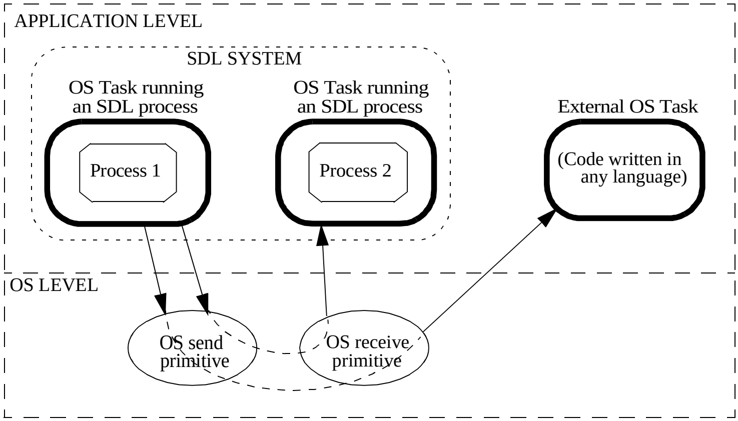


Figure 560: Signalling in the Tight Integration Model.

An SDL process sends one internal and one external signal. OS primitives are always used. The SDL system is not an entity of its own. Thick borders denote OS tasks.

Other properties of a Tight Integration:

- There is a single timer task which handles all SDL timers.
- SDL Simulators cannot be tightly integrated with an RTOS.
- Execution trace is available in textual or MSC format.[1]

Variations on Tight integration

Tight integrations come in two varieties, the Standard model and the Instance-Set model. Consider an SDL system with processes as outlined in Figure 561.

1. The MSC trace will be printed on standard output. To see the trace in an MSC diagram, copy the text into a file and read it into the MSC Editor.

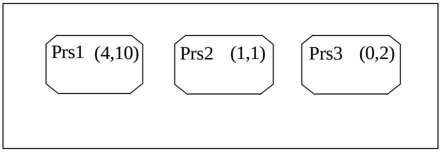


Figure 561: A partial SDL system

This system will be used for explaining the different models of Tight Integration.

Tight Standard

In the standard model of a Tight integration every SDL process instance is mapped to its own OS task. Tasks are created and destroyed dynamically as SDL process instances are created and terminated.

The example system will initially run as five OS tasks (four for Prs1, and one for Prs2). Up to 13 tasks may be created as needed.

Tight Instance-Set

In the Instance-Set model of a Tight integration every SDL process instance set is mapped to its own OS task. Tasks are created statically when the SDL system is initialized and are never destroyed. Thus there may be OS tasks that have no SDL process instances to run.

The example system will run as three OS tasks regardless of the number of process instances (one task each for Prs1, Prs2 and Prs3).

Choosing between Light, Threaded and Tight Integration

Choosing between the three integration models depends on many factors. Some of them are related to properties of the SDL system, others relate to the target environment. Important factors to consider are:

- The trade-off between performance and scheduling latency.
- How complex interaction the system has with the environment.
- Whether an operating system will be used or not.
- Memory management

Performance vs. scheduling latency

Generally, the Light integration model provides better performance than a Tight integration, but worst-case scheduling latency is the duration of the longest state transition.

A Tight integration is normally preferable when low scheduling latency is important. On the other hand, performance suffers from the overhead associated with OS scheduling and inter-process communications.

Another consideration is that blocking calls (either inlined in SDL code or as part of environment functions) will completely stop execution of a Light integration for the duration of the call. Making blocking calls in a Tight integration will only stop the thread making the call.

The Threaded integration is in a sense a combination of the other two integration models. An application part that makes blocking calls can be mapped to a separate OS thread. Other parts, where fast inter-process communication is important, can be mapped to another OS thread. Internally in this thread, signal exchange will be handled in the same way as in a Light integration.

Environment interaction

If interaction with the environment is simple then a Light integration is the best choice, especially if the system sends many signals and receives few. You can generate templates for the environment functions from the SDL suite and just add code for converting between the signal representation and the actual environment hardware or software.

If interaction with the environment is complex then a Tight integration is probably the easiest to use, since you only need to interface to the operating system queues. This is the case if you need process behavior in the interaction (for example, to establish a communications session before sending the signal).

Other cases where a Tight integration might be the best choice are

- when the environment consists of many OS tasks written in other languages than SDL
- when external processes send signals directly to SDL processes rather than to the SDL system in general
- when there are many signals passing to and from the environment

In a Threaded integration, the integration with the environment should normally be handled in the environment functions in the same way as for a Light integration. It is possible though in Threaded to send signals directly to an SDL process with the `SDL_Output()` function without involving the `xInEnv()` function.

Operating system issues

If you will not use an OS at all then you have to select a Light integration. In addition, you will have to provide some simple functions for getting system time and handling memory allocation (depending on compiler and libraries, standard C library functions can often be used).

If you use an OS that takes care of load balancing between CPUs then you should select a Tight integration, because load balancing normally uses threads as the load unit to distribute.

If you want to distribute your SDL System over several nodes (CPUs) you should use the Threaded integration together with the TCP/IP feature.

Considerations when choosing between Tight and Threaded

Overall, Telelogic recommends using the Threaded integration mode, unless there are specific technical reasons to prefer one of the other integration models.

The two most important reasons for choosing the Threaded model are:

1. The Threaded integration out-perform a Tight Integration in most situations, e g implementation of Timers and creation of tasks is about 100% faster in Threaded.
2. In Threaded you are not limited to two partitioning models. Any mapping between SDL objects and threads can be specified in the Deployment editor.

For a more comprehensive list of the differences between Threaded and Tight see the following sections.

Advantages of Threaded

- The default model is very simple to implement for a new OS. A working prototype should normally take 2-3 days.

- The Threaded model can be used together with the TCP/IP feature.
- The Threaded model is supporting all partitioning models (1 process instance/1 or many process instance-set/1 or many blocks/1 entire system mapped to one Thread.
- The user can easily specify Thread specific parameters like: stack size, Thread priority, Thread message queue size and maximum message size for each Thread.
- The Thread specific code for all supported OS is placed in one file.

Advantages of Tight

- Simple mapping od SDL concepts to OS primitives.
- Simple interactions with environment (at least in the default model), by direct call to the OS message passing primitives.
- Support textual and MSC-pr textual trace in console window when executing.

Disadvantages of Threaded

- Slightly more difficult to interact with the environment. An external function/thread must use the SDL_Output function when sending signals to an SDL process.

Disadvantages of Tight

- Very difficult and time consuming to support a new OS.
- Can only support two partitioning models.
- Very inefficient Timer model.

Common Features

This section describes the parts of the integration that are common to the different models, but also some important differences.

Note:

Many of the data structures and macros described in this section are described in more detail in chapter 62, *The Master Library*. That chapter is, however, focused on a Light Integration. Some things, especially the listings of data structures, are not correct in every detail for Tight Integrations but should still be very useful.

The Use of Macros

A source file generated by an SDL to C compiler is independent from the choice of integration model and operating system. Instead of system calls, it uses macros that have to be defined elsewhere when the system is built. Each SDL concept is represented by one or more C macros. These macros will have different definitions for different integration models and operating systems. Telelogic Tau provides a number of integration packages for this purpose. In a Light Integration, many macros are expanded into functions of the standard kernel. A Tight Integration has lower level macros that are defined in separate files for each operating system, and finally expanded into OS primitives or certain OS dependent constructions.

Below is an example of generated code for signal sending. Code in capital letters are the SDL suite macros. The bracketed numbers indicate corresponding lines of code. For a description of the different macros, see chapter 62, *The Master Library*.

SDL symbol: Output

Generated code before macro expansion:

```
[1]    ALLOC_SIGNAL(sig2, ySigN_z3_sig2,
         TO_PROCESS(Env, &yEnvR_env),
         XSIGNALHEADERTYPE)
       SIGNAL_ALLOC_ERROR
[2]    SDL_2OUTPUT_COMPUTED_TO(
         xDefaultPrioSignal,
         (xIdNode *)0, sig2,
         ySigN_z3_sig2,
         TO_PROCESS(Env, &yEnvR_env),
         0, "Sig2")
       SIGNAL_ALLOC_ERROR_END
```

Generated code after macro expansion for a Light Integration:

```
[1]    yOutputSignal = xGetSignal
         ((&ySigR_z3_sig2),
         (*(&yEnvR_env)->
         ActivePrsList !=
         (xPrsNode) 0 ?
         (*(&yEnvR_env)->
         ActivePrsList)->Self :
         xSysD.SDL_NULL_Var),
         yVarP->Self);
[2]    SDL_Output (yOutputSignal,
         (xIdNode *) 0);
```

File Structure

Note:

The source file and examples for Tight Integration are not included in the standard delivery. They are available as free downloads from Telelogic Support web site.

The Generated Files

An integration uses one or more of four files that can be generated by the Code Generator. All integrations require the C source file `<systemname>.c`, whereas the interface file `<systemname>.ifc` is optional. For a Tight Integration the signal number file `<systemname>.hs` may be used, and for a Light Integration the file `sctenv.c`, representing the environment, can be used.

The source file uses the highest level set of macros, that are defined in the different integration packages.

The Integration Packages

The files containing the necessary macro definitions and support functions are organized as shown in Figure 562. For each operating system there is one package for the Light Integration, and one for the Tight Integration. Although the files in different packages may have the same name they do not necessarily contain the same code. The principles for Tight Integration packages are described more thoroughly in "Tight Integration" on page 3249. Details about each operating system can be found in the annexes.

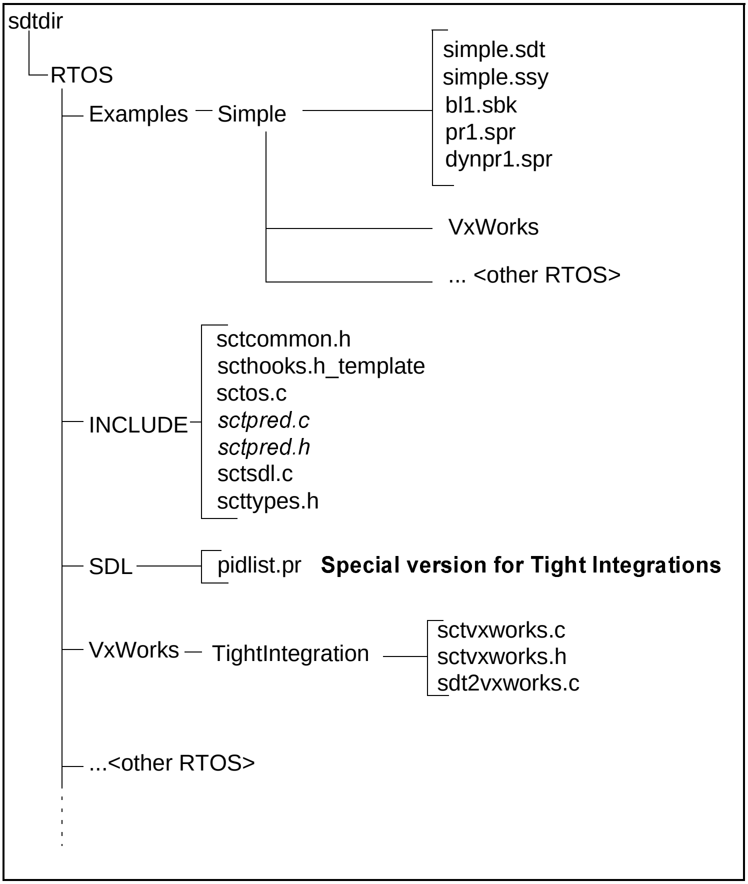


Figure 562: File structure for the RTOS integrations

There is also an `INCLUDE` directory containing source files common to all supported operating systems. These files are described in "File Structure" on page 2951 in chapter 62, *The Master Library*. Below are some additional comments on these files:

- `scthooks.h`:

  This file contains macro hooks into the system. These hooks are used for customizing a Tight Integration, for example by adding external tasks. See the examples to find out how this is done.

  All the hook macros are initially empty.

- `sctcommon.h`:

  This file contains general macros for all Tight Integrations

- `sctos.c`:

  This file contains functions that are operating system and/or compiler dependent, like allocation and free of dynamic memory.

- `sctpred.h` and `sctpred.c`:

  This is the header and source file where all the SDL predefined datatypes are implemented.

- `sctsdl.c`:

  This file contains SDL support functions, the timer task and the timer support functions. It is non-OS-specific and calls many second level OS-specific macros defined in `sct<RTOS>.h`.

- `scttypes.h`:

  This file contains the general datatype definitions for signals, `IdNodes`, etc. It also contains the macro definitions found in generated code. Note that this file is non-OS-specific. This means that if a call to an OS-specific primitive is needed, then a second level of macro is defined, according to the following model.

  In the generated code:

  ```
  ALLOC_SIGNAL_PAR(ok, ySigN_z3_ok, TO_PROCESS(Env,
  &yEnvR_env), yPDef_z3_ok)
  ```

  In `scttypes.h`:

```
#define ALLOC_SIGNAL_PAR(SIG_NAME, SIG_IDNODE, \
   RECEIVER, SIG_PAR_TYPE) \
   RTOSALLOC_SIGNAL_PAR(SIG_NAME, SIG_IDNODE, \
   RECEIVER, SIG_PAR_TYPE)
```

In `sct<RTOS>.h`:

```
#define RTOSALLOC_SIGNAL_PAR(SIG_NAME,SIG_IDNODE,\
   RECEIVER, SIG_PAR_TYPE) \
   yOutputSignalPtr = \
    xAlloc(sizeof(xSignalHeaderRec)+ \
    sizeof(SIG_PAR_TYPE)); \
   yOutputSignalPtr->SigP = yOutputSignalPtr+1; \
   yOutputSignalPtr->SignalCode = SIG_NAME; \
   yOutputSignalPtr->Sender = SDL_SELF;
```

The `Examples` directory contains a simple SDL system that also uses an external process (a separate OS task). For each supported operating system there is an implementation of this, demonstrating how to hook into the integration package. Further description of the example can be found in "A Simple Example" on page 3264.

Naming Conventions

Names of variables, datatypes and support functions in generated code and package files often start with one of the letters x, y and z.

The general rules (there are some exceptions) are:

- Names and objects starting with an 'x' represent general datatypes and support functions in the kernel.

  Examples: `extern XCONST struct xVarIdStruct`, `xInputSignal`, `xFindReceiver`

- Names starting with a 'y' are names of IdNodes representing SDL variables, process states, channels, blocks, datatypes for signals, PAD functions, etc. in generated code.

  Examples: `extern XCONST struct xVarIdStruct yVarR_z012_okmess`, `ySigR_z3_ok`, `yPAD_z01_pr1`

- Names and objects starting with a 'z' are SDL variables, SDL names, process state names, etc. in generated code.

  Examples: `#define z010_idle 1`, `z012_okmess`

The Symbol Table

All signals, blocks, processes, channels, etc. in an SDL system have a corresponding representation in the symbol table described in chapter 62, *The Master Library*. This symbol table consists of nodes (`IdNodes`) each representing one entity of the system. The `IdNodes` are pointers to structs. See the following example:

```
extern XCONST struct xPrsIdStruct yPrsR_z02_dynpr1;
#define yPrsN_z02_dynpr1  (&yPrsR_z02_dynpr1)
```

The `N_` and the `R_` just before `z02_dynpr1` indicate if it is a node (`N_`) or a record (`R_`).

Memory Allocation

The `xAlloc` function is always used when allocating dynamic memory. The function is placed in the `sctos.c` file, but in the Tight Integrations the body of the function is found in the `sct<RTOS>.h` file.

Start-up

The `yInit` function is called during start-up of the SDL system. It is responsible for creating all static processes and for initializing SDL synonyms.

Implementation of SDL Concepts

SDL Processes

An SDL process consists of three parts in generated code: Instance set common data, instance specific data and dynamic behavior.

Instance set common data

Variables and structures that are common to all instances of a process are stored in a record of the type `xPrsIdStruct`, defined in `scttypes.h`. This record is referenced by a node in the symbol table.

Instance specific data

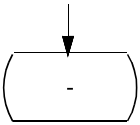
Variables and structures of the process instance are declared via the macro `PROCESS_VARS`. This macro is defined in different ways in the Light and the two models of Tight Integration. It contains state information, local variables, pointers to parent and offspring, etc. One important entry is the `RestartAddress`, pointing out which transition to execute when the PAD function runs (see below).

Dynamic behavior

The dynamic behavior of an SDL process is implemented in a PAD function (Process Activity Definition). The PAD function is used somewhat differently in the different integration models:

- In the Light Integration the scheduler calls the PAD function of the process. The PAD function then returns to the scheduler when the transition is finished.
- In the Standard Model Tight Integration the PAD function is called when the process is started and does not return until process termination. It then contains a main loop where one iteration corresponds to one transition.
- The Instance Set Model Tight Integration contains a mix of the two.

Below is the code for ending a transition, before and after macro expansion for a Light Integration.



SDL symbol: Nextstate

Generated code before macro expansion:

```
      /*-----
      * NEXTSTATE -
      * #SDTREF(SDL,/ti/RTOS/MANUAL/SDL/simple.spr(1),
                143(55,100),1)
      ------*/
      #ifdef XCASELABELS
[1]     case 6:
      #endif
[2]     XAT_LAST_SYMBOL
[3]     SDL_DASH_NEXTSTATE
```

Generated code after macro expansion for a Light Integration:

```
      /*-----
      * NEXTSTATE -
      * #SDTREF(SDL,/ti/RTOS/MANUAL/SDL/simple.spr(1),
                143(55,100),1)
      ------*/

[1]     case 6:

[2]       xGRSetSymbol (-1);
[3]       SDL_NextState (VarP, yVarP->State);
          return;
```

Signal Queues

Basically, every process has at least one signal queue.

- In a Light Integration each process instance has an input queue.
- In the Standard Model Tight Integration there is an input queue and a save queue for each instance.
- In the Instance Set Model Tight Integration there is an input queue and a save queue for every instance set. These two queues are shared by all the instances.

All signals that are sent to a process arrive in the input queue. The save queue is used to keep signals that cannot be handled in the current state but should be saved for future use. This is tightly connected to the SDL Save concept, but is also used in the implementation of timers. For a Light Integration there are no save queues. Instead, all signals that should be saved will remain in the input queue until they can be received.

Process Priorities

The use of process priorities requires some caution. Priorities can be set with the `#PRIO` directive in the SDL suite, but there is no mapping of priorities for different platforms. The generated code will use exactly the values specified in the SDL system. This will not be a problem in a Light Integration, but for Tight Integrations the result may not be the expected.

Example 536: Process priority problems ––––––––––––––––––––––––

Assume that Process1 has its priority set to 100 using the `#PRIO 100` directive and Process2 has `#PRIO 50`. These priority values will be used as-is by the underlying scheduler.

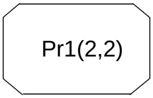
In the SDL Suite simulators and Light integrations, the highest priority is 0. In VxWorks, 0 is the highest priority, whereas in pSOS the highest priority is 255.

Thus the simulated system, a Light integration for any operating system and a Tight integration for VxWorks will all run Process2 at a higher priority than Process1. In contrast, a Tight integration for pSOS will run Process1 at a higher priority than Process2!

––

Process Creation

Regardless of the integration model there are a number of things that have to be done when an SDL process instance is created. The structs that represent the instance have to be created. It needs a representation in the symbol tree and a signal queue, except in the Instance Set Model of Tight Integration where the signal queue belongs to the instance set. A start-up signal is also always allocated and sent to the process.



SDL symbol: SDL Static Create

Generated code before macro expansion:

```
      /*********************************************************
                    SECTION      Initialization
      *********************************************************/

      extern void yInit XPP((void))
      {
        int  Temp;
      ........
        INIT_PROCESS_TYPE(pr1,z01_pr1,yPrsN_z01_pr1,"z01_pr1",
                          SDL_INTEGER_LIT(2),SDL_INTEGER_LIT(2),
                          yVDef_z01_pr1, xDefaultPrioProcess,
                          yPAD_z01_pr1)
      #ifdef SDL_STATIC_CREATE
[1]     for (Temp=1; Temp<=SDL_INTEGER_LIT(2); Temp++) {
[2]        SDL_STATIC_CREATE(pr1,z01_pr1,yPrsN_z01_pr1,
                             "pr1",ySigN_z01_pr1,
                             yPDef_z01_pr1,yVDef_z01_pr1,
                             xDefaultPrioProcess,yPAD_z01_pr1,1)
        }
      #endif
```

Generated Code after macro expansion for a Light Integration:

```
      /*********************************************************
                      SECTION      Initialization
      *********************************************************/
      extern void
      yInit ()
      {
        int  Temp;
      ..........
[1]     for (Temp = 1; Temp <= 2; Temp++) {

[2]       SDL_Create (xGetSignal ((&ySigR_z01_pr1),
                      xSysD.SDL_NULL_Var, xSysD.SDL_NULL_Var),
                     (&yPrsR_z01_pr1), 1);
        }
```

SDL Signals

Signal Sending

In generated code a signal sending is handled by two macros: ALLOC_SIGNAL (or ALLOC_SIGNAL_PAR for a signal with parameters) and SDL_2OUTPUT_xxxx (there are different macros depending on how the SDL output was defined, e.g. with or without an explicit TO).

SDL symbol: Output
Generated code before macro expansion:

```
[1]    ALLOC_SIGNAL(go,ySigN_z03_go,TO_PROCESS(p,yPrsN_z09_p),
                    XSIGNALHEADERTYPE)
       SIGNAL_ALLOC_ERROR
[2]    SDL_2OUTPUT_COMPUTED_TO(xDefaultPrioSignal,(xIdNode*)0, go,
                  ySigN_z03_go,TO_PROCESS(p, yPrsN_z09_p), 0, "Go")
       SIGNAL_ALLOC_ERROR_END
       XBETWEEN_SYMBOLS(4, 579)
```

Generated code after macro expansion for a VxWorks Tight Integration:

```
[1]    yOutputSignalPtr = xAlloc(sizeof(xSignalHeaderRec));
[1]    yOutputSignalPtr->SignalCode = 2;
[1]    yOutputSignalPtr->Sender = yVarP->Self;
[2]    Err = msgQSend (xTo_Process ((&yPrsR_z09_p)),
[2]                    (char*)*&yOutputSignalPtr,
[2]                    sizeof(xSignalHeaderRec) + 0, 0, 0);
[2]    xFree ((void **) &yOutputSignalPtr);
[2]    if (Err == (-1)) {
[2]      taskLock ();
[2]      printf ("Error during %s found in VXWORKS
[2]      function %s. Error code %s\n", "OUTPUT",
[2]      "msgQSend", strerror ((*__errno ())));
[2]      taskUnlock ();
[2]    }
```

In this example the signal is called go and has no parameters. The SDL_2OUTPUT_COMPUTED_TO macro indicates that it was sent without an explicit TO.

SDL Procedures

An SDL procedure is represented by a function similar to a PAD function. Before a procedure is called there are two support functions that need to be called: xGetPrd and xAddPrdCall.

The xGetPrd function allocates an xPrdStruct for the called procedure and returns an xPrdNode pointing to the struct.

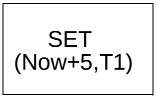
The `xAddPrdCall` function adds the new procedure call in the calling process' `ActivePrd` list (an element in the `xPrsStruct`).

The procedure is called with a pointer to the instance data of the calling SDL process. This is because the procedure must be able to use internal variables in the calling process.

Before a procedure returns to the caller it performs an `xReleasePrd` call. This function removes the call from the `ActivePrd` list.

SDL Timers

SDL timers are represented by signals. All active timer signals are kept in a sorted list, either within the single task of a Light Integration or in a certain timer task in a Tight Integration. When a timer expires, the signal representing it is sent to the SDL process that set it.



SDL symbol: SET Timer

Generated code before macro expansion:

```
      /*-----
      * SET T1
      * #SDTREF(SDL,/ti/RTOS/MANUAL/SDL/dynpr1.spr(1),
                119(55,25),1)
      ------*/
      #ifdef XCASELABELS
[1]     case 2:
      #endif
[2]       SDL_SET_DUR(xPlus_SDL_Time(SDL_NOW,
[3]        SDL_DURATION_LIT(5.0, 5, 0)),
[4]        SDL_DURATION_LIT(5.0, 5, 0), t1, ySigN_z021_t1,
             yTim_t1, "T1")
```

Generated code after macro expansion for a Light Integration:

```
      /*-----
      * SET T1
      * #SDTREF(SDL,/ti/RTOS/MANUAL/SDL/dynpr1.spr(1),
                119(55,25),1)
      ------*/

[1]     case 2:

[2]       SDL_Set (xPlus_SDL_Duration (SDL_Now (),
[3]         SDL_Duration_Lit (5, 0)), xGetSignal
[4]         ((&ySigR_z021_t1), yVarP->Self,
            yVarP->Self));
```

Light Integration

A Light Integration is a stand-alone executable which can be generated with or without a simulator. An executable that should run under UNIX can use the precompiled kernels. Only the environment functions need to be written by the user.

PAD Functions

The PAD function is called by the scheduler when its process is in turn to execute a transition. The scheduler calls the PAD function with a symbol table node of the type `xPrsNode`, pointing to the instance specific data of the instance that is scheduled.

Start-Up

A Light Integration starts when the generated main function is called. The start-up phase works like this (pseudocode shown in *italics*):

```
void main( void )
{
   xMainInit();
   Code from #MAIN
   xMainLoop();
}


void xMainInit( void )
{
   xInitEnv();
   Init of internal structures
}
```

You must supply the `xInitEnv()` function to initialize external code and hardware, etc. (this is of course application dependent). This function is placed in the same program module (environment module) as the `xInEnv()` and `xOutEnv()` functions. The `xMainLoop()` function contains an eternal loop, which constitutes the scheduler itself. See below:

```
void xMainLoop (void)
{
   while (1)
   {
    xInEnv(...)
    if (a timer has expired)
       send the corresponding timer signal
    else if (a process can execute a transition)
     {
       remove the signal from the input port
       set up Sender in the process to Sender of the signal
       call the PAD function for the process
     }
   }
}
```

Connection to the Environment

Signals going in and out of the SDL system are handled in the two user written functions `xInEnv` and `xOutEnv`. There is a template file for writing these two function in the standard distribution. This file can be found at `<installation directory>/sdt/sdtdir/<your platform os version>sdtdir/INCLUDE/sctenv.c`.

Running a Light Integration under an External RTOS

Since there are some fundamental differences between different RTOS we can only give a general idea of how to generate a Light Integration under an external RTOS here. Typical things that may be different in different RTOS:

- If you are allowed to have a main function in your application.
- If your start-up function must be specified in a configuration file.
- If the cross compiler requires additional OS-specific header files to be included.
- If it is possible to run the application in a simulated target environment.
- Syntax for the makefile.

General Steps

The normal steps to create a Light Integration under an external RTOS can be summarized as follows:

1. Copy the source and header files for an application kernel from the installation of the SDL suite. The files are residing in the following directory:

   `<installation directory>/sdt/`
   `sdtdir/<your host and os version>sdtdir/INCLUDE`.

2. Generate an `<application>.c` file with the SDL to C Compiler.

> **Note:**
>
> The code generator option *Cadvanced* **must** be used.

3. Generate an environment header file (an option in the Organizer *Make* dialog).

4. Edit the `sctenv.c` template file to handle your in and out environment signals. Include the generated `<application>.ifc` file (the environment header file).

5. Create a makefile or edit the generated makefile. Write entries for the kernel source files, the environment file and the application file.

6. Set the appropriate compilation switches for your RTOS and your compiler.

7. Compile the application and the kernel file to create a relocatable object file.

8. Download.

Threaded Integration

Introduction

The first part of this section is about the Threaded integration. The second part is about the TCP/IP feature. With the Threaded and TCP/IP features you can partition an SDL system into several threads that can execute in a distributed environment.

Implementation Details for Threaded

The main areas where a Threaded differs from a Light integration are:

- Symbol table structures and global variables
- Process creation
- Signal sending

Symbol Table Structures and Global Variables

Each thread has a global variable of type `xSystemData` in Threaded. In Light, there is only one global variable of this type for the entire system. In Threaded this variable is initialized in `yInit()` and is one of the parameters in the macro for creating a new thread.

The data structure `xSystemData` has a couple of new entries:

```
...
#ifdef THREADED
   xInputPortRec  xNewSignals;
   THREADED_THREAD_VARS
#endif
...
```

The entry `xNewSignals` is used when a new signal is received. The signal is first linked into the receiver's `xNewSignals` queue before it is handled in the receiver's `xMainLoop()`.

The entry `THREADED_THREAD_VARS` is a macro that contains different thread variables like the semaphore handles `xInputPort`.

Global variables are declared in the macro `THREADED_GLOBAL_VARS:`

```
#elif THREADSOLARIS
.....
#define THREADED_GLOBAL_VARS \
  pthread_mutex_t xListMutex; \
```

```
  pthread_mutex_t xExportMutex; \
  sem_t xInitSem; \
  int xNumberOfThreads; \
  int QueueCounter; \
  int xInitPhase;  \
  pthread_attr_t Attributes ;
#endif /* THREADED_ALTERNATIVE_SIGNAL_SENDING */
#endif
```

Process Creation

There are three macros related to the creation of threads in a Threaded:

- THREADED_START_THREAD(F, SYSD, THREAD_STACKSIZE, THREAD_PRIO, THREAD_MAXQUEUESIZE, THREAD_MAXMESSIZE)

  This macro is used if there is only one thread to be created for all instances (Instance Set) of an SDL process (or another SDL object). The parameters are:

Parameters	Explanation
F	The entry point of the thread, i.e. the SDL kernel function xMainLoop()
SYSD	A variable of type xSystemData
THREAD_STACKSIZE	The stacksize for the thread is specified in the Deployment Editor
THREAD_PRIO	The thread priority specified in the Deployment Editor
THREAD_MAXQUEUESIZE	The maximum number of messages/signals in the input queue of the thread, as specified in the Deployment Editor. This parameter is only used if the alternate signal sending model is used.

Parameters	Explanation
THREAD_MAXMESSIZE	The maximum size of a message/signal. This parameter is ignored at the moment in both implementation models. The maximum size of a message/signal in the alternate signal sending model is always the size of the pointer to an SDL signal (`sizeof(xSignalNode`).

Note:

Thread parameters for individual threads can be specified in the Deployment Editor. If no values are specified, default values will be used.

- `SDL_CREATE(PROC_NAME, PROC_IDNODE, PROC_NAME_STRING)`

  This macro is used for creation of dynamic processes (processes created during run-time).

  It is defined exactly the same way in Threaded as in Light. It will call the SDL kernel function `SDL_Create()`.

  `SDL_Create`() will create a new thread if, for instance, there should be one thread for each instance of an SDL process. See the following extract from the `SDL_Create()` function:

```
...................
#ifdef THREADED
 if (PrsId->SysD == 0) {
    THREADED_START_THREAD(xMainLoop, StartUpSig-
>Receiver.LocalPId->PrsP->SysD, PrsId->ThreadParam-
>ThreadStackSize, PrsId->ThreadParam->ThreadPrio,
PrsId->ThreadParam->MaxQueueLength, PrsId-
>ThreadParam->MaxMesSize);
}
#endif
....................
```

  Please note that the thread parameters are taken from the `xPrsIdNode` for the process.

  `SDL_STATIC_CREATE(PROC_NAME, PREFIX_PROC_NAME, PROC_IDNODE, PROC_NAME_STRING, STARTUP_IDNODE,`

```
STARTUP_PAR_TYPE, PRIV_DATA_TYPE, PRIO,
PAD_FUNCTION, BLOCK_INST_NUMBER)
```

This macro is used for creating static processes (processes that are created at system start-up). It will be called from the `yInit()` function and is mapped to the SDL kernel function `SDL_Create()` in the same way as in the `SDL_CREATE` macro.

The `xMainLoop()` function is the entry point for the thread. First in this function is the macro

```
THREADED_THREAD_BEGINNING(SYSD)
```

The main purpose of this macro is to wait for the start-up semaphore `xInitSem`. No thread is allowed to start executing until ALL static processes/threads have been created.

Sending Signals

In the default model, signals are sent in the same way as for a Light integration except that they are first linked into the `xNewSignals` queue.

In the `xMainLoop()` function, the `xNewSignals` queue is checked for new entries. If a new signal is available, it is sent to the process itself with the `SDL_Output()` function. See the following extract from the `xMainLoop()` function:

```
........
THREADED_LOCK_INPUTPORT(((xSystemData *)SysD))
while (((xSystemData *)SysD)->xNewSignals.Suc != (xSig-
nalNode)&((xSystemData *)SysD)->xNewSignals)
SDL_Output(((xSystemData *)SysD)->xNewSignals.Suc
xSigPrioPar(((xSystemData *)SysD)->xNewSignals.Suc-
>Prio), 0);
THREADED_UNLOCK_INPUTPORT(((xSystemData *)SysD))
.........
```

The Alternative Signal Sending Model.

In the alternative signal sending model, an OS message/signal is sent containing a pointer to the SDL signal. For this, a message queue must be created for each thread at thread creation. How this is done differs from OS to OS. See : Signalling in Threaded Integration.

The sender sends the pointer to the SDL signal with the OS_Send primitive. The receiver receives the message, extracts the pointer to the SDL signal and links it into the xNewSignal queue. After this, the signal is handled in the same way as in the default model.

New Macros

The same source files are used in a Threaded integration as in a Light. The only SDL kernel source files that are affected by the Threaded integration are `scttypes.h`, `sctsdl.c` and `sctos.c`.

Most of the OS specific code is found in the `scttypes.h` file. The following macros are new for Threaded:

MACRO NAME	Explanation
`THREADWIN32`	Main macro for Threaded Windows integration
`THREADSOLARIS`	Main macro for Threaded Solaris integration
`THREADVXWORKS`	Main macro for Threaded VxWorks integration
`THRAEDOSE`	Main macro for Threaded OSE integration
`THREADED_ALTERNATIVE_SIGNAL_SENDING`	Main macro for using the alternative signal sending model.
`THREADED_POSIX_THREADS`	Main macro for the Threaded integration model defining the kernel specifics
`THREADED`	Internal macro used only in the kernel source files
`THREADED_TRACE`	Enabling textual execution trace
`THREADED_OSTRACE`	This macro is mapped to a printf statement.
`THREADED_ERROR`	Enabling textual error printout when calling OS primitives.
`THREADED_ERROR_VAR`	Defines a variable used in `THREADED_ERROR_RESULTS`.

MACRO NAME	Explanation
THREADED_ERROR_RESULT	Stores the return result after calling an OS primitive.
THREADED_ERROR_REPORT	Checks the value of the return variable and if ERROR makes a printout.
THREADED_ERROR_REPORT_NULL	Checks the value of the return variable and if ERROR makes a printout.
THREADED_ERROR_REPORT_NEG	Checks the value of the return variable and if ERROR makes a printout.
THREADED_ERROR_REPORT_OPEN_NEG	Checks the value of the return variable and if ERROR makes a printout.
THREADED_ERROR_REPORT_WAIT_NEG	Checks the value of the return variable and if ERROR makes a printout.
THREADED_GLOBAL_VARS	Global variable defines.
THREADED_GLOBAL_INIT	Initialization of global variables like semaphores.
THREADED_THREAD_VARS	Definitions of thread variables.
THREADED_THREAD_INIT	Initialization of thread variables.
THREADED_THREAD_BEGINNING	Wait for xInitSem to be released.
THREADED_LOCK_INPUTPORT	Protect the input queue by taking the semaphore.
THREADED_UNLOCK_INPUTPORT	Releasing the semaphore for the input queue.
THREADED_WAIT_AND_UNLOCK_INPUTPORT	Wait for next message/signal to arrive or the next internal timer to expire.
THREADED_SIGNAL_AND_UNLOCK_INPUTPORT	Send a signal and release the semaphore for the input queue.

MACRO NAME	Explanation
`THREADED_LISTREAD_START`	Protect global active and available lists with a semaphore before reading it.
`THREADED_LISTWRITE_START`	Protect global active and available lists with a semaphore before writing to it.
`THREADED_LISTACCESS_END`	Release the semaphore protecting a global active or available list.
`THREADED_EXPORT_START`	Protect export and import actions by taking a semaphore
`THREADED_EXPORT_END`	Release the semaphore after export and import actions.
`THREADED_START_THREAD`	Start a new thread.
`THREADED_STOP_THREAD`	Terminate a thread.
`THREADED_AFTER_THREAD_START`	Synchronize the start-up of newly created threads.
`THREADED_SEND_OUTPUT`	Send messages/signals. Used in both models.

Textual and MSC Trace in Threaded

Textual SDL Trace, similar to the Textual Trace in Telelogic Tau Simulator, can be turned on by selecting the flag *SDL trace* (will set the flag `THREADED_XTRACE`) in the Target Library/Kernel window in Targeting Expert.

On-line MSC trace is possible when running an application under a soft-kernel on Windows or UNIX. Select the flag *MSC trace* (will set the flag `THREADED_MSCTRACE`) in the Target Library/Kernel window in Targeting Expert.

API for interfacing a Threaded Integration

An API is available with a number of useful functions/MACROS that will facilitate the work of sending/receiving signals between external processes/threads and an SDL Threaded Application.

The following functions and MACROS are available:

SDL_PId xThreadedRegExtTask()

This function will return an SDL PId representing the calling External process/thread. It works in the following way:

1. Get a ThreadId for the calling process/thread.
2. Allocate and assign an SDL struct (`xPrsIdRec`) representing the external process/thread.
3. Call the SDL kernel function `xGetPId()` to get an `SDL_PId`.
4. If MSC trace is on it will generate an entry for the process/thread in the MSC diagram.
5. Allocate and initialize an SDL “system data record” for the external process/thread.
6. Create a queue for the calling process/thread.
7. Assign the “system data record” entry in the `SDL_PId`.
8. Return the `SDL_PId`.

xSignalNode xThreadedReceiveSDLSig(SDL_PId)

This function will wait for an SDL signal indefinitely. When a signal arrives the signal will be taken out of the queue and return the signal. If MSC trace is on the signal will also be traced in the MSC diagram.

xSignalNode xThreadedReceiveSDLSig_WithTimeOut(SDL_PId, SDL_Duration)

This function will work in the same way as `xThreadedReceiveSDLSig()` except that it will only wait for the specified time.

THREADED_ASSIGN_SDL_SIG_PARAMS(sigptr,signame,paramno,param)

This macro will use the macro token operator ## to concatenate the sigptr, signame, paramno into a simple assignment of the signal parameter with the specified number.

> **Note:**
> This macro should only be used with simple datatypes where assignments can be done using the simple assignment operator "="

THREADED_GET_SDL_SIG_PARAM(sigptr,signame,paramno,param)

This macro will assign the user's param the value of the signal parameter with the specified number.

> **Note:**
> This macro should only be used with simple datatypes where assignments can be done using the simple assignment operator "="

SDL_Output()

This is the standard SDL kernel function that should be used when sending signal into a Threaded Application. The following must be done before a signal can be sent from an external process/thread into a Threaded Application.

1. Use the `xThreadedRegExtTask/xThreadedRegExtTask_WithQueue` function to get an `SDL_PId`.
2. Call the SDL kernel function `xGetSignal` with the following parameters: `signalId`, `Receiver`, `Sender`.
3. Assign signal parameters using the API macro `THREADED_ASSIGN_SDL_SIG_PARAMS()`
4. Call the SDL_Output function with the SDL signal.

For an example see Annex 6: Building a Threaded Integration

THREADED_START_EXTTASK

This macro is normally empty. It is called after all static SDL threads are created in the `THREADED_AFTER_THREAD_START` macro (last in generated c file for application).

By defining this macro to user can make the application start external tasks.

THREADED_SIMPLE_EXAMPLE

If this flag is defined the external tasks in the simple example will be started.

Implementation Details for Different RTOS

VxWorks

The Threaded integration for VxWorks is developed and tested using a Solaris Softkernel in Tornado 2.

The following vxWorks header files are used:

```
#ifdef THREADVXWORKS

#include "errno.h"
#include "vxWorks.h"
#include "semLib.h"
#include "msgQLib.h"        /* msgQCreate msgQDelete ms-
gQSend msgQReceive *//* msgQNumMsgs */
#include "taskLib.h"        /* taskSpawn */
#include "semaphore.h"      /* POSIX semaphores */
#endif
```

The following VxWorks primitives have been used:

VxWorks primitives	Explanation
sem_init(..), sem_wait(..), sem_post(..), sem_destroy(..),	POSIX semaphores are frequently used, e.g. to protect the input queue, to synchronize start-up...
msgQCreate(..), msgQReceive(..), msgQSend(..), msgQDelete(..)	Message queues are used in both models in VxWorks. The message queue is created in the THREADED_START_THREAD MACRO.

VxWorks primitives	Explanation
`taskSpawn(..)`, `taskDelete(..)`, `taskSuspend(..)`	taskSpawn is used for creating a thread. The name entry is not used. taskSuspend is only used by the "Main" thread. It is called from the macro `THREADED_AFTER_THREAD_START`.
`taskIdSelf()`	Used in taskDelete() and taskSuspend()

Thread Parameters	Default values
`DEFAULT_STACKSIZE`	800
`DEFAULT_PRIO`	100
`DEFAULT_MAXQUEUESIZE`	250
`DEFAULT_MAXMESSIZE`	sizeof(xSignalNode)

Solaris

The Threaded integration for Solaris is developed and tested in the following environment:

Sun Solaris 2.6 with Sun WorkShop compiler cc version 5.0.

Included header files for Solaris:

```
#elif THREADSOLARIS
#include <pthread.h>
#include <semaphore.h>
#ifdef THREADED_ALTERNATIVE_SIGNAL_SENDING
#include <mqueue.h>
#include <signal.h>
#include <time.h>
#endif /* THREADED_ALTERNATIVE_SIGNAL_SENDING */
#endif
```

The following Solaris primitives have been used:

Solaris primitives	Explanation

`sem_init(..),`
`sem_wait(..),`
`sem_post(..),`
 	
`pthread_mutex_init(..),`
`pthread_mutex_lock(..),`
`pthread_mutex_unlock(..),`
`pthread_mutex_destroy(..)`
 	
`pthread_cond_init(..),`
`pthread_cond_wait(..),`
`pthread_cond_timedwait(..),`
`pthread_cond_signal(..),`
`pthread_cond_destroy(..)`
	The sem_.. primitives are only used in the start-up to synchronize static processes/threads.
	
	The pthread_mutex_.. primitives are used to protect the input queues and other queues.
	
	The pthread_cond_... primitives are used to synchronize sender and receiver in the default model.

`pthread_attr_init(..),`
`pthread_attr_setstacksize(..),`
`pthread_attr_setschedpolicy(..),`
`thread_attr_setdetachstate(..),`
`pthread_attr_setscope(..)`
	The pthread_attr_.. primitives are used to set the attributes of a thread before it is created.

`mq_open(..),`
`mq_close(..),`
`mq_receive(..),`
`mq_unlink(..),`
`mq_send(..)`
	These primitive are only used in the alternative signal sending model.

`pthread_create(..),` `pthread_exit(..),`	These primitives are used when a thread is created and when it terminates.
`timer_settime(..),` `timer_delete(..)`	These primitives are used for setting a timer before the mq_receive() is called in the alternative signal sending model. The time-out for the timer is the duration for the next internal SDL timer to expire. When the timer expires the signal_handler function sets the error message to EINTR.

Thread Parameters	Default values
DEFAULT_STACKSIZE	15000
DEFAULT_PRIO	10
DEFAULT_MAXQUEUESIZE	128
DEFAULT_MAXMESSIZE	sizeof(xSignalNode)

OSE

The Threaded integration for OSE have been developed and tested with an OSE Softkernel (version 4.3) on Solaris 2.6 using the gcc compiler (version 2.95).

The following OSE header file is include:

```
#ifdef THREADOSE
#include "ose.h"
#endif /* THREADOSE */
```

The following OSE primitives have been used:

OSE Primitives	Explanation
create_sem(..), wait_sem(..), signal_sem(..),	These primitives are used for protecting input queues, other queues and for synchronizing start-up
receive(..), receive_w_tmo(..), send(..)	These primitive are used for receiving/sending signals in both models. Receive/send is also used to pass the start-up parameter xSysD to a new thread, since OSE does not support start-up parameters in the create_process primitive.
alloc(..), free_buf(..)	These primitives are used for allocating and returning OSE signals.

OSE Primitives	Explanation
`start(..)`	This primitive is used for starting a newly created thread.
`kill_proc(..)`	This primitive terminates a thread
`stop(..)`	This primitive stop the execution of a thread. Used in the `THREADED_AFTER_THREAD_START` macro.
`current_process()`	Used in the kill_proc() and stop() calls.
`create_process(..)`	This primitive is used for creating Threads.

Thread Parameters	Default values
`DEFAULT_STACKSIZE`	1024
`DEFAULT_PRIO`	8
`DEFAULT_MAXQUEUESIZE`	1024
`DEFAULT_MAXMESSIZE`	sizeof(xSignalNode)

Declaration of xMainLoop().

The declaration and definition of the thread's entry point (xMainLoop()) is special in OSE:

Declaration in scttypes.h;

```
.......
#elif THREADOSE
extern OSENTRYPOINT xMainLoop;
#else
extern void xMainLoop XPP((xSystemData *));
#endif
```

Definition in sctsdl.c:

```
......
#elif THREADOSE
```

```
OS_PROCESS(xMainLoop)
#else
#ifndef XNOPROTO
void xMainLoop (xSystemData * SysD)
#else
void xMainLoop (SysD)
                              xSystemData * SysD;
#endif
#endif
..............
```

Definition of NIL.

NIL is defined in the OSE kernel and must not be redefined in the SDL kernel.

```
#ifndef THREADOSE
#define NIL 0
#endif /* THREADOSE */
```

Forward declaration of xSignalNode.

The xSignalNode must have forward declaration very early in the scttypes.h file since it is used in the union SIGNAL definition.

```
.............
typedef struct xSignalStruct *xSignalNode;
union SIGNAL
{
  SIGSELECT      sigNo;
  xSignalNode    SDLSig;
  xSystemDataPtr SysD;
};

.....
```

Further down in the scttypes.h file.

```
.....
#ifndef THREADOSE
typedef struct xSignalStruct       *xSignalNode;
#endif /* THREADOSE */
.....
```

Windows

The Threaded integration for Windows is developed and tested on Windows 2000 Professional with the Borland C++ compiler, version 5.02.

The following header files for Windows are included:

```
#ifdef THREADWIN32
#include "limits.h"
#include "windows.h"
#include "dos.h"
......
```

The following Windows primitives are used in the Threaded integration:

Windows primitives	Explanation

`CreateSemaphore(..),`
`ReleaseSemaphore(..),`
`WaitForSingleObject(..),`
`CloseHandle(..)`
	These primitives are used for protecting input queues, other queues and synchronization in start-up.
	One extra semaphore, xInitQueue, is used when a newly created thread is creating his own input queue.

`PeekMessage(..)`	This primitive is used in the `THREADED_THREAD_BEGINNING` macro in the alternative signal sending model. This primitive force the thread to create a message queue. It is used together with the xInitQueue semaphore.
`GetMessage(..),` `PostThreadMessage(..),`	These primitives are used for receiving/sending messages in the alternative signal sending model.
`SetTimer(..),` `KillTimer(..)`	These primitives are used to set an OS timer that will signal the thread when it expires. The timeout of this timer is the duration until the next internal SDL timer expires for this thread. The window message will be set to `WM_TIMER` if the OS timer expires.

Windows primitives	Explanation
`CreateThread(..)`, `ExitThread(..)`	These primitives are used for creating and terminating threads.
`SetThreadPriority(..)`	This primitive is used for setting the priority of the thread.
`SuspendThread(..)`	Called by the "main" thread in the macro `THREADED_AFTER_THREAD_START`
`GetCurrentThread()`	Used in the SuspendThread macro.

Thread Parameters	Default values
`DEFAULT_STACKSIZE`	0 (Automatically resized by OS)
`DEFAULT_PRIO`	`THREAD_PRIORITY_NORMAL`
`DEFAULT_MAXQUEUESIZE`	1024
`DEFAULT_MAXMESSIZE`	sizeof(xSignalNode)

Signal Sending over TCP/IP

Introduction

For applications using the Threaded integration model, a plug-in module for TCP/IP communication is available. The module supports signal sending between distributed SDL applications via a TCP/IP connection. ASCII Encoding/Decoding is used for the conversion between signal and transport format. The module is delivered as C source code which is integrated and built together with code generated by the CAdvanced SDL to C Compiler.

The TCP/IP adapter supports the four operating systems for which Threaded integrations are available. These are Windows, Solaris, VxWorks and OSE.

Architecture

The TCP/IP functions are called from the environment functions `xInitEnv` and `xOutEnv`. The functions are included by setting the `XENV_INC` flag to `"tcpipcomm.h"`. The `tcpipcomm.h` file contains `#define` directives that translates macros in the environment file to function calls in the TCP/IP adapter.

When a signal is sent to the environment using the `xOutEnv` function, `xSendSignal` is called. A connection is set up with a remote server. The signal destination is specified using a user-implemented function. When the connection is accepted, the signal is encoded into ASCII format and sent via a TCP/IP socket. The session is then closed.

From `xInitEnv`, a thread is started which polls a socket for incoming connections. When a connection from a remote client is accepted, a new thread is started, which receives and decodes data for one signal. When the signal is decoded, it is inserted in the signal queue of the SDL system. The thread finishes its execution after the connection is closed.

An executing SDL system thus acts as a server when signals are received from the environment and as a client when a signal should be sent to the environment.

The environment file of an SDL application may not be modified if the TCP/IP adapter is to be used. Making modifications may override the TCP/IP signal sending functionality. If you want to use the TCP/IP adapter together with other external code from an environment file, please consult "Configuration" on page 3246.

File Structure

The TCP/IP adapter consists of four files, located in `$sdtdir/tcpip/`.

- `tcpipcomm.c` should be compiled and linked with the generated C code
- `tcpipcomm.h` is a C header file which is included from the generated environment file
- `tcpipthr.h` and `tcpipsock.h` are header files that are included from `tcpipcomm.c` and `tcpipcomm.h`.

The `tcpip` directory is referenced relative to the `$sdtdir` variable.

Hint: Using TCP/IP with a new $sdtdir

If you use an `$sdtdir` that is different from the default `$sdtdir` (in the SDL suite installation directory), be sure to copy the **`tcpip`** directory to the new location. Otherwise, you may encounter problems in finding the TCP/IP files at compilation.

Note: Pointers as Signal Parameters

Distributed components execute in separate memory spaces. Care must be taken so that pointers are not sent as signal parameters over TCP/IP and used in a remote component.

Routing of Signals

For each signal that is sent from your SDL application, you must specify a destination in the form of an IP address (or host name) and a TCP port number. This information should be accessed from a routing function which is called when a signal is sent from the SDL application.

The routing function is made accessible from the TCP/IP adapter by setting the flag `XROUTING_INC` to the name of a routing header file. This is a plain C header file where the macro `XFINDDEST(OUTSIG, SIGNAME, IP, PORT)` is defined as a function. `OUTSIG` and `SIGNAME` are **in** parameters. `IP` and `PORT` are **out** parameters.

`OUTSIG` should be declared as `xSignalNode*`. From this parameter, signal data such as parameters can be accessed. `SIGNAME` holds the name of the signal and is declared as char*. In many cases, the signal name is sufficient routing information. The `IP` and `PORT` variables should be set to the host address and port number. `IP` should be declared as `char*` and `PORT` as `int*`.

The routing function should be implemented in a C file which is compiled and linked together with the application. The server IP address should be given as `char*`, e.g. `"255.255.255.255"` (dotted decimal notation) or `"server.the_company.com"` (hostname notation). The server port number should be given as `int`, e.g. `8888`.

On the server side, the IP address is inherent to the host that the application resides on. The port number on which a server should listen for incoming connections should be specified using the flag `XSERVPORT`.

Note: Usage of Port Numbers

Generally, TCP port numbers below 1024 are reserved by operating system services or internet applications. For instance, the port numbers 21 and 23 are used by FTP and port number 80 is used by HTTP. If a port number is occupied, you will get an error message at start-up and the server thread will not start. It is recommended that you select port numbers larger than 1024 for your SDL application.

Example 537: TCP/IP Adapter Routing Settings ––––––––––––––––––

This example shows a situation where different signals should be directed to different recipients. An SDL system is partitioned into three components (i.e. executable files) using the Threaded integration model. From component 1, two different signals can be sent. Sig1 should be sent to component 2 and Sig2 should be sent to component 3.

Component 2 resides on a host called `“host2”`. It uses port number `7001` for listening for incoming connections, which means that `XSERVPORT` is set to `7001 (XSERVPORT=7001)`. Component 3 resides on `“host3”` and listens on port number `7001 (XSERVPORT=7001)`. Please note that the ports are not in conflict since the hosts are different.

For component 1, a routing function is implemented. The flag `XROUTING_INC` is set to `“router.h”`. The routing header file has the following contents:

```
#define XFINDDEST(OUTSIG, SIGNAME, IP, PORT)\
xGetDestination(OUTSIG, SIGNAME, IP, PORT)
```

The routing C file, `router.c`, contains the implemented function:

```
#include “stdlib.h”

void xGetDestination(xSignalNode *sig, char
*sigName, char *IPAddr, int *Port)
{
  if (strcmp(sigName, “Sig1”) == 0)
  {
    strcpy(IPAddr, "host2\0");
    (*Port)= 7001; /* Dereferencing */
  }
  else if (strcmp(sigName, “Sig2”) == 0)
  {
    strcpy(IPAddr, "host3\0");
```

```
        (*Port)= 7001;
      }

      return;
    }
```

`router.c` is then compiled and linked together with the generated code, the coder library and the TCP/IP adapter. This example shows a fairly trivial routing scenario. The routing is based only on the name of the signal. The xSignalNode pointer is not used.

Error Handling

The TCP/IP adapter contains basic error handling. Error checks are performed when encoding/decoding, socket and thread functions are invoked. If `THREADED_ERROR` is defined, an error message is printed on stdout with the name of the function where the error occurred. A platform-dependent error code is included in the error message. For a description of the error code, consult the User's Manual of your target operating system.

An error implies that the function where the error occurred exits. Clean-up is performed, which means that the application can continue its execution.

A special case to consider is when an error occurs in the server thread function, which runs statically during the execution. The thread exits and must be restarted manually.

If `THREADED_TRACE` is defined, the execution of the TCP/IP adapter is logged onto stdout when signals are encoded, sent, received and decoded.

Example 538: TCP/IP Adapter Error Message

An error occurs when a signal should be sent via the TCP/IP adapter. The following is logged on stdout:

```
ERROR xSendSignal/SCM_CONNECT: 146
```

The target platform is Solaris. The error code indicates that the connection was refused, probably because a server can not be found at the specified address. `xSendSignal` exits and the application continues its execution.

Configuration

The TCP/IP adapter is configured using compilation flags. The basic configuration can be done using the TCP/IP Connection Wizard in the Targeting Expert.

Including the TCP/IP adapter

The TCP/IP adapter requires environment files, environment header files and ASCII encoding/decoding for correct operation. These options are activated when the TCP/IP Signal Sending check box is enabled in the Targeting Expert's TCP/IP Connection Wizard.

Server Port Number

For a component that receives signals, a TCP port number must be specified. The server thread uses this port number to listen for incoming connections.

The port number is set in a text box in the TCP/IP Connection Wizard. You can also set the port manually by setting the flag `XSERVPORT` to the desired value. If no port number is specified, the port number is set to 5000 by default.

Routing

You must manually implement a routing function, so that a destination is specified for every SDL signal that is sent to the environment. The function must be declared in a C header file. The file is specified in a text box in the TCP/IP Connection Wizard or by setting the flag `XROUTING_INC` to the header file name.

The routing function implementation should be placed in a C file that is compiled and linked with the other code. It is specified using either the text box in the TCP/IP Connection Wizard or by including it in the compiler settings manually. See Example 537 for an example of how a routing function is implemented.

Using the TCP/IP Adapter with Other Environment Functionality

The TCP/IP adapter header file (`tcpipcomm.h`) is included from the environment file of a component. The file `tcpipcomm.h` contains `#define` statements for macros in the environment functions that invoke TCP/IP functions.

If you want to use other functionality in parallel with the TCP/IP adapter, these macros can be defined outside `tcpipcomm.h`. By setting the flag `XEXTENV_INC` to the header file you wish to use, your header file is included from `tcpipcomm.h`. This enables you to insert your code without modifying `tcpipcomm.h`. Please note that the `XEXTENV` flag can not be set in the TCP/IP Connection Wizard. It must be set manually.

Before using the `XEXTENV_INC` flag, look carefully at the #define statements in `tcpipcomm.h`. These must be valid for proper operation of the TCP/IP adapter.

Hint: Using External env Code with the TCP/IP Adapter

When combining the TCP/IP adapter with other environment functions, always preprocess the environment file to verify that the code is expanded as expected.

Example 539: External Code in Combination with the TCP/IP Adapter

A file called `mycomm.h` contains declarations of functions that should be used in parallel with the TCP/IP adapter. From the xInitEnv function, an initialization function should be called (`InitComm()`). From the `xInEnv` function, a function for polling communication (`ReadComm()`) should be invoked.

In `mycomm.h`, the following is inserted:

```
#define XENV_INIT Initcomm();\
                  xInitSignalSender();\
                  xInitSignalReceiver();
#define XENV_IN_START ReadComm();
```

`xInitsignalSender` and `xInitSignalReceiver` are taken from `tcpipcomm.h`, which contains the following:

```
#ifndef XENV_INIT
#define XENV_INIT xInitSignalSender();\
                  xInitSignalReceiver();
#endif
#ifndef XENV_IN_START
#define XENV_IN_START
#endif
```

The #defines in `tcpipcomm.h` are overridden. Still, the original calls are invoked, which ensures that the TCP/IP adapter executes properly.

Using Thread Parameters

Some thread parameters in the TCP/IP adapter can be set to fine-tune performance of your Threaded SDL application. The TCP/IP adapter threads do not use OS queues. Two parameters can be set: Thread Priority and Thread Stack Size. These are set manually using flags. They can not be set using the TCP/IP Communication Wizard.

The server wait thread uses the following flags:

```
XSERVTHRPRIO
XSERVTHRSTACK
```

The signal receiver threads use the following flags:

```
XRECVTHRPRIO
XRECVTHRSTACK
```

Set the flags to values that are specific to the target platform used.

Tight Integration

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from Telelogic Support web site.

Note:

This presentation is focused on the general principles and models used in a Tight Integration. When specific RTOS primitives are needed in the presentation, examples from the VxWorks implementation are used. The implementation and RTOS calls used in other integrations are covered in separate annexes to this chapter, one for each supported RTOS.

There are two models of Tight Integration. In the Standard Model one SDL process instance is mapped to one OS task. In the Instance Set Model an entire instance set (all instances of a process) is mapped to one OS task. Scheduling between OS tasks is managed by the RTOS scheduler; this means that preemption is normally used, though only on an instance set level in the Instance Set Model. SDL semantics are preserved in a Tight Integration, for example setting a timer implies an automatic reset first.

The start-up of a system, i.e. creation of static processes, initialization of synonyms and creation of an environment task and a timer task, is handled by a generated initialization function called `yInit`. Normally this function is called from another initialization function, where some additional initializations take place before the `yInit` function is called.

Timers in the system are handled by one central timer task. This task receives messages[1], each containing a request to set a timer, and will send messages back as the timers expire.

Common Features

File Structure

The files related to the tight integration concept are placed in the following directory in the installation: `<installation directo-`

1. SDL signals will be implemented as messages in VxWorks.

`ry>/sdt/sdtdir/RTOS/<operating system>/TightIntegra-tion/`. The same files are used for both the Standard Model and the Instance Set Model.

Each RTOS directory contains the following files:

- `sct<RTOS>.h`:

  This file contains the second level of macros (see the comments for `scttypes.h` in "The Integration Packages" on page 3213). All macros are using OS-specific calls or types.

- `sct<RTOS>.c`:

  This file contains OS-specific support functions.

- `sdt2<RTOS>.c`:

  Most RTOS require that signals/messages are represented with an integer value. This is the source file for a utility program for generating signal identities. Each signal will be assigned an integer value. The output will be a file with the suffix `.hs`. This file is automatically included in the application.

  In the SDL suite, the `.hs` file can also be generated by the SDL to C Compiler by turning on the option *Generate signal number file* in the *Make* dialog. The `.hs` file is included in the application if the compilation switch `XINCLUDE_HS_FILE` is set.

The SDL_PId Type

The `SDL_PId` (SDL Process ID) type has different meanings in the Standard and the Instance Set Models. In the Standard Model it represents the message queue, while it represents the process instance in the Instance Set Model. This is because the entire instance set will have the same message queue in the last case.

```
#ifdef X_ONE_TASK_PER_INSTANCE_SET
typedef xEPrsNode SDL_PId;
#else
typedef MSG_Q_ID  SDL_PId;
#endif
```

Signals

The signal header consists of a struct with information needed to handle the signal inside an SDL system. The signal header struct is defined in the RTOS-specific file `sct<RTOS>.h`.

```
typedef struct xSignalHeaderStruct *xSignalHeader;
typedef struct xSignalHeaderStruct {
  int            SignalCode;
  xSignalHeader  Pre, Suc;
  SDL_PId        Sender;
  void          *SigP;
  #ifdef X_ONE_TASK_PER_INSTANCE_SET
    SDL_PId        Receiver;
  #endif
  #ifdef XMSC_TRACE
    int            SignalId;
    int            IsTimer;
  #endif
} xSignalHeaderRec;
```

The signal header stores `SignalCode`, in this case an integer, two pointers `Pre` and `Suc` used when saving the signal in the save queue, and `Sender`, holding the `SDL_PId` of the sending SDL process. In the Instance Set Model there is an extra parameter `Receiver`, necessary to make a distinction between the SDL processes in an instance set task.

If the signal contains parameters they are allocated in the same function call. Example:

```
OutputSignalPtr = xAlloc (sizeof (xSignalHeaderRec)
+ sizeof(yPDef_z05_s2));
```

The second parameter to the `xAlloc` function is a struct representing the signal parameters of the signal. In this case, with one integer, it is defined in the following way:

```
typedef struct {
  SIGNAL_VARS
  SDL_Integer     Param1;
} yPDef_z05_s2;
```

The macro `SIGNAL_VARS` is in most RTOS empty.

There is an extra element in the `SignalHeader` defined as a `void` pointer. This pointer `SigP` is set to point to the parameter area.

```
 OutputSignalPtr->SigP = OutputSignalPtr+1;
```

This pointer is used in the Signal-Free-Function to address the parameter-part of either a signal-structure as also a timer-signal-structure.

Note:

The SDL signal parameters are always named `Param1`, `Param2`, etc.

Assignment of the signal parameter is done in generated code and not in a macro. Example:

```
SIGNAL_ALLOC_ERROR
yAssF_SDL_Integer(((yPDef_z05_s2*)
  OUTSIGNAL_DATA_PTR)->Param1 ,yVarP->z023_param1,
  XASS);
```

The macro `OUTSIGNAL_DATA_PTR` macro is defined:

```
#define OUTSIGNAL_DATA_PTR (yOutputSignalPtr->SigP)
```

After expansion of the whole expression the code will be:

```
((yPDef_z05_s2 *) ((xSignalHeader) yOutputSignalPtr
+ 1))->Param1 = yVarP->z023_param1;
```

Signal reception

The support function `xInputSignal` is used for receiving signals in both models of Tight Integration. The implementation and the parameters are different though.

Timer Signals

A timer signal is defined similarly to an ordinary signal but will contain some additional elements representing time-out time, etc. The timer header struct looks like this:

```
typedef struct xTimerHeaderStruct *xTimerHeader;
typedef struct xTimerHeaderStruct {
      int             SignalCode;
      xTimerHeader    Pre, Suc;
      SDL_PId         Sender;
      void           *SigP;
#ifdef X_ONE_TASK_PER_INSTANCE_SET
      SDL_PId         Receiver;
#endif
#ifdef XMSC_TRACE
      int             SignalId;
      int             IsTimer;
#endif
      SDL_Time        TimerTime;
```

```
        int             TimerToSetOrReset;
        xbool           (* yEq)();
        xbool           TestParams;
        xTimerHeader    Param;
    } xTimerHeaderRec;
```

> **Note:**
>
> An ordinary signal is identical to the first part of a timer signal. This makes it possible to type-cast between the two types as long as only elements in the common part of the headers are used.

Time

When the System time is required, for example when using NOW, the macro `SDL_NOW` is used. The macro is in turn mapped to the function `SDL_Clock()` (in `sctos.c`). This function is implemented differently depending on the RTOS representation of time. In VxWorks it returns the result of calling the RTOS function `tickGet`. `SDL_Time` is normally implemented as `int` or `unsigned long int`.

Mapping Between SDL Time and RTOS Time

The macro `SDL_DURATION_LIT` specifies the mapping between the SDL time in seconds and the local RTOS representation of time. In VxWorks the system time is given in ticks and the translation is defined as follows:

```
#define SDL_DURATION_LIT(R,I,D) \
  ((I)*1000 + (D)/1000000)
```

`R` is the real type representation of the time in seconds. `I` and `D` are the integer and decimal parts of an integer type representation of the time. `I` is in seconds and `D` in nanoseconds. The code generator will generate all three numbers but either `R`, or `I` and `D` will be used depending on the RTOS.

Timers

All timer activity in the SDL system is handled by a dedicated timer task. The timer task accepts requests in the form of messages (in VxWorks). It then keeps the requests for setting a timer sorted in a timer queue and uses some OS mechanism to wait for the first request to time out. The mechanism used can be either an OS timer, or a timeout in the waiting for new requests. When a request times out, the timer task sends a signal back to the task that first sent the request. The function calls and

OS signaling involved in setting and waiting for an SDL timer can be viewed in Figure 563.

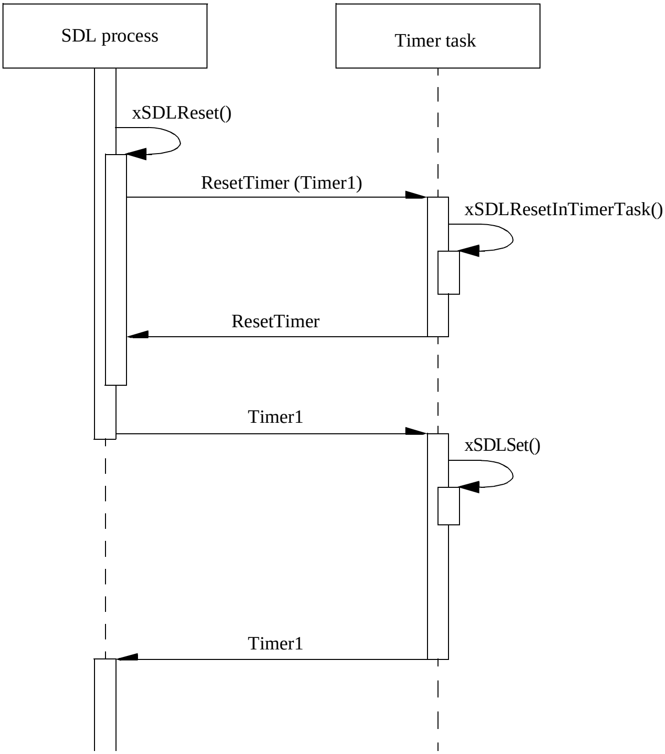


Figure 563: Function calls and OS signaling when setting and waiting for a timer.

UML notation is being used, thus the full arrowheads represent function calls and the half arrowheads represent messages. Parameters have been left out, except for the name of the timer in the ResetTimer request. Note that a reset is performed first, as required in the SDL specification.

To be able to implement the full semantics of SDL timers a number of support functions have been implemented:

- `xSDLActive`

  Checks whether an SDL timer is active and returns true or false. The function passes the question on to the timer task in the form of a request.

- `xSDLActiveInTimerTask`

  Called by the timer task upon request. Checks if an SDL timer is active and returns true or false.

- `xSDLReset`

  Resets an SDL timer by sending a request to the timer task. While waiting for a reply all new signals to the calling task are saved in the save queue. In the case of an Instance Set Model Tight Integration this means that no instance of the process can execute a transition until a reply is received. If the reply states that the timer couldn't be found it might be in the save queue or the input queue of the task because it has recently expired. If so, it is simply removed. SDL semantics require that a reset is always performed implicitly prior to setting a timer.

- `xSDLResetInTimerTask`

  Called by the timer task when a request has been made for resetting a timer. Checks the timer queue to see if the timer to reset is there. If the timer is found, it is removed and the data area it holds is freed. A message is sent back to the task that made the request, telling whether the timer was found or not.

- `xSDLSet`

  Called by the timer task when a request has been made for setting a timer. This function sorts the request into the timer queue.

Addressing SDL Processes

There are two ways to address SDL processes from an external task. Either the `xFindReceiver` function can be called to find an arbitrary receiver, or the file `pidlist.pr` can be used to provide a list of the SDL processes and then address the receiver explicitly via the input queue ID of its OS task.

The xFindReceiver Function

When sending a signal into the SDL system where the receiver is not known a support function called `xFindReceiver` can be used. This function takes the following parameters:

- The ID of the signal

- The sender ID (in this case an SDL_PId representing the environment)
- An optional VIA-list.

The following files are needed to get access to SDL types, signal numbers and signal parameter types: `scttypes.h`, `<system_name>.hs` and `<system_name>.ifc`.

Example of how to use the `xFindReceiver` function:

```
#include "scttypes.h"
#include "<system_name>.ifc"
#include "<system_name>.hs"

void MyExtTask(void) {

   xSignalHeader yOutputSignalPtr;
   int Err;

   /*Allocate signal header and signal parameter
     buffer */
   yOutputSignalPtr =
     (xSignalHeader)xAlloc(sizeof(xSignalHeaderRec)
      + sizeof(yPDef_go);

   /*Setup signal header */
   yOutputSignalPtr->SignalCode = go;
   yOutputSignalPtr->Sender = xEnvPId;

   /*Give value 100 to integer parameter */
   ((yPDef_go *)(yOutputSignalPtr+1))->Param1 = 100;

   /*Send signal from environment */
   Err = msgQSend(xFindReceiver(go, xEnvPrs, 0),
           (char*) yOutputSignalPtr,
           sizeof(xSignalHederRec)+sizeof(yPDef_go),
           0 ,0);
}
```

The following types, signal definitions and global variables are used in the example:

- `xEnvPId`: An SDL PId representing the environment, from `scttypes.h`
- `xEnvPrs`: A PrsNode representing the environment, from `scttypes.h`
- `xSignalHeader`: A datatype representing an SDL signal, from `scttypes.h`

- `yPDef_go`: A datatype representing the signal parameter types, from `<system_name>.ifc`
- `go`: An SDL signal, from `<system_name>.ifc`

The File pidlist.pr

An alternative way to get the PId for the Receiver is to use an ADT defined in the ADT library called `pidlist.pr`. This file defines an ADT called PidList and an operator called PId_Lit. With this ADT it is possible to directly address any static process instance in the system, both from internal SDL processes and from external OS-tasks. You can find more information about this feature in "How to Obtain PId Literals" on page 3178 in chapter 63, *The ADT Library*.

> **Note:**
> If you need the `pidlist.pr` ADT in a Tight Integration then you must use the version in the
> `<installation directory>/sdt/sdtdir/RTOS/SDL/`
> directory.

The Standard Model

In the Standard Model of the Tight Integration each SDL process is implemented as an OS task. Preemption and the use of process priorities is only limited to what the OS supports.

Processes

Process Creation

An SDL process is created in the following way (in the VxWorks integration):

1. A start-up signal is allocated.
2. A message queue is created. Some operating systems create the message queue automatically when the task is created. This is explained for each operating system in the annexes to this chapter.
3. The task is created with the message queue ID as a start-up parameter. In the case of VxWorks, the task will have a name starting with `VXWORKSPAD_`. This is a function which will first initialize some internal variables and then call the PAD function.

4. A function (`xAllocPrs`) is called to create a representation of the new instance in the global symbol tree.
5. The start-up signal is sent. When this signal is received in the task the start transition of the process is executed.

Process Termination

The following actions are carried out when a process terminates:

1. The save queue and the message queue are emptied.
2. The save queue is deleted.
3. A message is sent to the `xTimerTask` with a request to remove all active timers of the process.
4. `xFreePrs` is called to free the `PrsNode`.
5. The message queue is deleted. In some operating systems this is done automatically when the task is deleted.
6. The task is deleted.

PAD functions

Each PAD (Process Activity Definition) function will contain an eternal loop with an OS receive statement. When a process instance is created it is the PAD function that is called in the OS Create primitive.

The start-up and execution of a PAD function works like this:

1. The support function `xInputSignal` is called. This function will wait for the start-up signal, that is always received first, and then return to the PAD function.
2. The PAD function goes to the label `Label_Execute_Transition`. This label is the start of a code block containing a switch statement that evaluates the process variable `RestartAddress`. The code under each different case then represents a transition. At the end of this block the process variable `State` is updated and execution continues at `Label_New_Transition`.
3. In `Label_New_Transition` a new call is made to `xInputSignal` and execution then continues at `Label_Execute_Transition`.

The structure of a PAD function is described below (with pseudo-code shown in *italics*):

```
void yPAD_z01_pr1 (void *VarP)
{
  Variable declarations
  xInputSignal is called to receive the start-up signal
  ...........
  goto Label_Execute_Transition;
  ...........
  Label_New_Transition:
  xInputSignal is called to receive a signal

  Label_Execute_Transition:
  Local declarations
  switch (yVarP->RestartAddress) {
  case 0:
    Execute the start transition
    Update the process state variable
    goto Label_New_Transition;
  case 1:
    Execute the transition
    Update the process state variable
    goto Label_New_Transition;
  ...........
  }
  ...........
}
```

Scheduling

Since each SDL process is implemented as an OS task, scheduling between processes will be handled completely by the OS.

Start-up

Start-up of a Standard Model Tight Integration can be described as follows (pseudocode is shown in *italics*):

```
MyMain() {
/* initialization of semaphores etc */
 yInit();
 Give startup semaphores
 taskSuspend(Mymain);
}

yInit() {
 Create the timer task
 Create an environment task or only an environment queue
 for(i=1;i<=NoOfStaticProcessTypes;i++){
  for(j=1;j<=NoOfStaticInstancesOfEachProcesstype;
      j++){
    Allocate a startup signal
    Create a message queue
```

```
        Create a task
        Call xAllocPrs
        Send the startup signal
      }
    }
  Assign SDL synonyms
}
```

The semaphore is used for synchronizing start-up of static processes. No static process is allowed to execute its start transition before all static processes are created, because a start transition can have signal sending to other static instances.

The `MyMain` function is placed among other support functions in the file `sctsdl.c`.

The `yInit` function is generated by the code generator and placed last in the generated file for the system.

The Instance Set Model

The Instance Set Model is based on the same principles as the Standard Model with the difference that the instance set is the basic unit rather than the process instance.

Processes

Both the instances and the instance sets are represented in the symbol table. In addition to the three parts that always make up an SDL process there is also an extra struct for the instance set, defining for example the input queue which is common to all the instances. Further, there is a PAD function for each instance, but also for the instance set.

Process Representation

An SDL_PId is represented by an `xEPrsNode`, pointing to an `xEPrsStruct`. An `xEPrsNode` also represents a process instance in the symbol table both in the Standard Model and the Instance Set Model.

```
typedef struct xEPrsStruct {
  xEPrsNode       NextPrs;
  SDL_PId         Self;
  xPrsIdNode      NameNode;
  int             BlockInstNumber;
  xPrsNode        VarP;
} xEPrsRec;
```

Instance Set Data

The datatype `xPrsInstanceSetVars` is only used in the Instance Set Model. It defines common data for all instances of the set, like the save queue and the size of the instance data.

```
typedef struct {
  xSignalHeader   SaveQ;
  xSignalHeader   CurrentInSaveQ;
  xSignalHeader   yInSignalPtr;
  char            name[100];
  unsigned        PrsDataSize;
} xPrsInstanceSetVars.
```

PAD functions

In the Instance Set Model there is a PAD function for each process instance but also for each instance set. The instance set PAD functions will be called at system start-up and contain an eternal loop in the same fashion as PAD functions in the Standard Model. Instance PAD functions are only called to execute transitions.

Process Creation

All instance sets, even for dynamic processes, are created at system start-up. Since the OS task and the signal queues are created with the instance set, the creation of an instance requires less labor than in the Standard Model. For the instance set creation the macro `INIT_PROCESS_TYPE` is used.

Process Termination

The instance set task is never terminated. Termination of a process instance will not remove the save queue, the input queue and the task. This is done at system termination. All queues, including the active timer queue, are emptied of messages to the terminated process though.

Signal queues

The message queue id of the receiver's instance set is accessed through `NameNode` in the receiver's `xEPrsStruct` and the variable `PROCID`.

Example from `xSDLResetInTimerProcess`:

```
Err=msgQSend ((MSG_Q_ID) (yInSignalPointer->Sender)
  ->NameNode->PROCID, (char *) yInSignalPointer,
  sizeof (xTimerHeaderRec), 0, 0);
```

In this case the receiver is the same as the original sender.

Signal sending

A support function `xHandle_Sig` is used when sending signals, instead of the macro `RTOSSEND` as in the Standard Model. This difference is shown in **bold** in the code below:

```
#ifdef X_ONE_TASK_PER_INSTANCE_SET
#define SDL_2OUTPUT(PRIO, VIA, SIG_NAME, SIG_IDNODE,\
   RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)\
   XOS_TRACE_OUTPUT(SIG_NAME_STRING) \
   XMSC_TRACE_OUTPUT(RECEIVER, yOutputSignalPtr, \
     SIG_NAME_STRING) \
   xHandle_Sig(yOutputSignalPtr,SDL_SELF,SIG_PAR_SIZE,\
    RECEIVER,(RTOSTASK_TYPE)RECEIVER-> \
    NameNode->PROCID RTOSHANDLESIG_PAR);
#else
#define SDL_2OUTPUT(PRIO, VIA, SIG_NAME, SIG_IDNODE, \
   RECEIVER, SIG_PAR_SIZE, SIG_NAME_STRING)\
   XOS_TRACE_OUTPUT(SIG_NAME_STRING) \
   XMSC_TRACE_OUTPUT(RECEIVER, yOutputSignalPtr,\
     SIG_NAME_STRING) \
   RTOSSEND(&yOutputSignalPtr, RECEIVER,SIG_PAR_SIZE)
#endif
```

Scheduling

Scheduling between instance sets is handled by the operating system. Within the instance sets, however, scheduling is based on the signal queue. When the instance set PAD function is executing, it takes the first signal in the input queue and calls the PAD function of the addressed SDL process. The instance PAD function then executes one transition and returns control to the scheduling loop of the instance set PAD function.

Integrating with external code

You can easily integrate the SDL system with external code, for example written in C. Just use the hooks described below for inserting C statements in the `main()` function of the SDL system.

The hooks are in the form of `#define` macros located in a file called `scthooks.h`. A file called `scthooks.h_template` with empty macros can be found in the `INCLUDE` directory. Use this file as a template for your own application. You will find usage examples in the `Examples` directory.

`HOOK_GLOBAL_DECLARATIONS`

This hook lets you declare function prototypes etc. at file scope.

HOOK_MAIN_DECLARATIONS

This hook lets you declare variables for use in the `main()` function.

HOOK_MAIN_START_OF_CODE

Any code inserted here will execute first in the `main()` function.

HOOK_MAIN_AFTER_PROCESS_RELEASE

Any code inserted here will execute as soon as all static processes have been created and are allowed to run.

HOOK_MAIN_AFTER_SIGNAL_RECEPTION

The `main()` function of the SDL system enters an infinite loop after having created all static processes. This loop is used to receive signals sent to the environment queue.

Use the `HOOK_MAIN_AFTER_SIGNAL_RECEPTION` to insert code for processing these signals.

Limitations for Integrations

In general, the same restrictions as for the SDL to C Compiler apply, but Tight integrations have some further restrictions. The detailed limitations for Light and Tight integrations are listed in the Release Guide.

A Simple Example

This section describes an example system named Simple. The annexes show how to integrate the example with different operating systems.

The example demonstrates the following techniques:

- How to integrate an SDL system with an operating system
- How to make the environment communicate with the SDL system in a Light integration
- How to make an external process written in C communicate with the SDL system in a Tight integration
- How to use the special Tight Integration version of the ADT `pidlist.pr`.

The Simple System

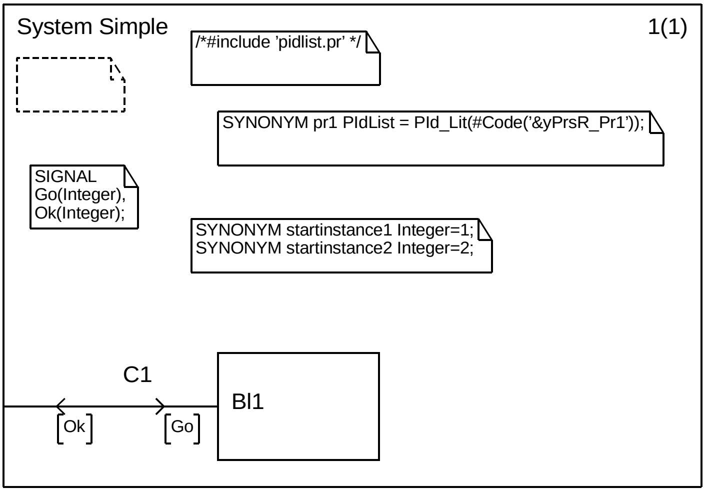


Figure 564: System Simple

The SDL representation of Simple consists of a single block Bl1. Seen from the outside, the system accepts the signal Go and responds after about five seconds by sending the signal Ok. The signal Go may be sent twice to the system.

Block Bl1

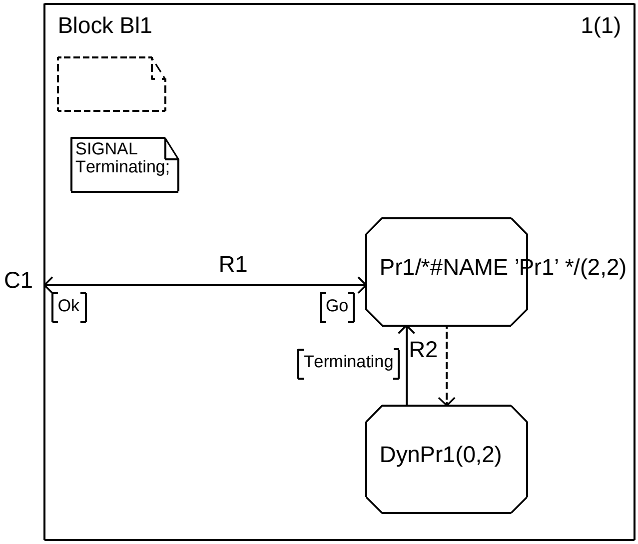


Figure 565: Block Bl1

Block Bl1 has two processes. The static process Pr1 and the dynamic process DynPr1. DynPr1 is created by Pr1 and can send the signal Terminating back to its parent. Pr1 handles all the interaction with the environment, through the signals Go and Ok.

Process Pr1

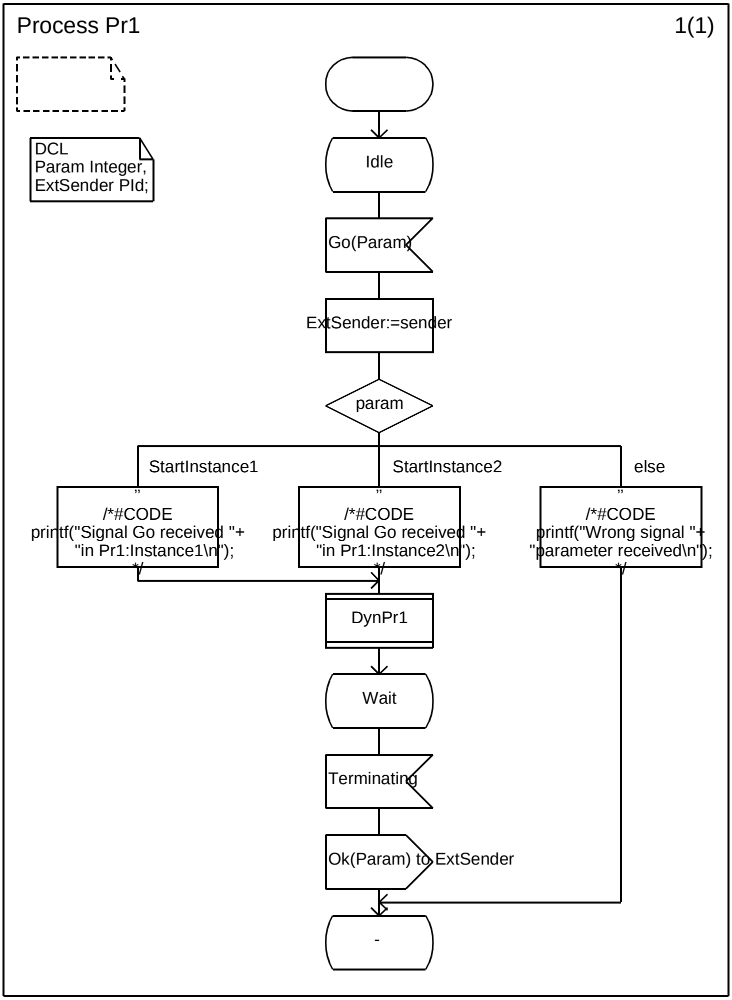


Figure 566: Process Pr1

Process Pr1 is a static process with two instances. It has two states, Idle and Wait. In the Idle state the process waits for the signal Go with an integer parameter representing the instance number of this instance. It then prints the instance number to the standard output, creates one instance of DynPr1 and enters the Wait state. In the Wait state it waits for the signal Terminating from the created instance of DynPr1, sends Ok back to the environment and goes back into the Wait state. Since the Terminating signal will only be received once, the process is going to remain in the Wait state forever.

Process DynPr1

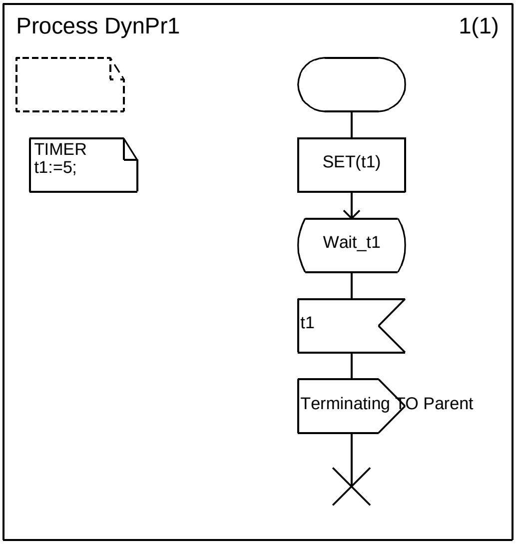


Figure 567: Process DynPr1

The dynamic process DynPr1 has no instances at system start and a maximum of two instances. Each instance of Pr1 creates one instance of DynPr1. DynPr1 sets the timer t1 to five seconds, waits for a timeout, sends the signal Terminating to its creator and finally terminates.

Connection to the Environment

The environment is handled in different ways depending on the integration model. See below for details.

Building and Running a Light Integration

This section will take you through the general steps required to build a Light Integration for the Simple example. The procedure works the same for most operating systems. Please also check the annexes for important information about your operating system.

General Steps for a Light Integration

1. Create a working directory and an INCLUDE directory below it.

> **Note:**
>
> `$(sdtdir) = <installation directory>/sdt/sdtdir/<machine dependent dir>`
>
> where `<machine dependent dir>` is `sunos5sdtdir` on SunOS 5, `hppasdtdir` on HP, and `wini386` in Windows.

2. Copy all files from `$(sdtdir)/RTOS/Examples/Simple/<selected RTOS>/LightIntegration` to the working directory.
3. Copy the following files to the `INCLUDE` directory:

```
$(sdtdir)/INCLUDE/sctlocal.h
$(sdtdir)/INCLUDE/sctpred.c
$(sdtdir)/INCLUDE/sctsdl.c
$(sdtdir)/INCLUDE/sctos.c
$(sdtdir)/INCLUDE/sctpred.h
$(sdtdir)/INCLUDE/scttypes.h
```

4. Open the system file `$(sdtdir)/RTOS/Examples/Simple/simple.sdt`
5. Change the destination directory to your working directory
6. Set the options *Lower Case* and *Generate environment header file*.
7. Select the Cadvanced SDL to C Compiler and generate an application.
8. Edit the `makefile` supplied with the example to fit your environment. Normally you will only have to point out the directory where the RTOS is installed.
9. Use the makefile to build an executable.
10. Download the executable to target or run it under a kernel simulator (“soft kernel”).

Result From Running the System

The output when running the example should be:

```
Signal Go received in Pr1:Instance1
Signal Go received in Pr1:Instance2
Signal Ok received with the following parameter:1
Signal Ok received with the following parameter:2
```

The xInEnv Function

This is where the start-up signal Go is sent. In a real system `xInEnv` may be used for polling hardware devices for data. The code looks like this:

```
xSignalNode S;
static int SendGo = 0;

if(SendGo<=1){
  if(SendGo==0) {
    S = xGetSignal(go,  pr1[1],   xEnv );
    ((yPDef_go *)(S))->Param1 = startinstance1;
  }
  else {
    S = xGetSignal(go,  pr1[2],   xEnv );
    ((yPDef_go *)(S))->Param1 = startinstance2;
  }
  SDL_Output( S, xSigPrioPar(xDefaultPrioSignal)
              (xIdNode *)0 );
  SendGo++;
```

The signal Go is sent the first and the second time `xInEnv` is called. The parameters `startinstance1 and startinstance2` are integer constants defined in the SDL system, as integer `SYNONYM`'s. They are made available by generating and including the file `simple.ifc`.

The xOutEnv Function

The code in `xOutEnv` for receiving the signal Ok looks like this:

```
if ( ((*S)->NameNode) == ok ) {
  printf("Signal Ok received with the following
          parameter:%lu\n",
          ((yPDef_ok *)(*S))->Param1);
  xReleaseSignal( S );
  return;
```

The signal Ok also has an integer parameter, the value of this should be 1 if it is sent by Pr1 instance one and 2 if it is sent by instance two.

Building and Running a Tight Integration

This section will take you through the general steps required to build a Tight Integration for the Simple example. The procedure works the same for most operating systems. Please also check the annexes for important information about your operating system.

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from Telelogic Support web site.

General Steps for a Tight Integration

1. Create a working directory and an INCLUDE directory below it.

Note:

`$(sdtdir) = <installation directory>/sdt/sdtdir/<machine dependent dir>`

where `<machine dependent dir>` is `sunos5sdtdir` on SunOS 5, `hppasdtdir` on HP, and `wini386` in Windows.

2. Copy all files from `$(sdtdir)/RTOS/Examples/Simple/<selected RTOS>/TightIntegration` to the working directory.
3. Copy all files from `$(sdtdir)/RTOS/Examples/Simple/<selected RTOS>/TightIntegration`/INCLUDE to the INCLUDE directory.
4. Open the system file `$(sdtdir)/RTOS/Examples/Simple/simle.sdt`
5. Change the destination directory to your working directory.
6. Set the options *Lower Case*, *Generate environment header file* and *Generate signal number file*.
7. Select the Cadvanced SDL to C Compiler and generate an application.
8. Edit the `makefile` supplied with the example to fit your environment. Normally you will only have to point out the directory where the RTOS is installed.

9. Use the makefile to build an executable.
10. Download the executable to target or run it under a kernel simulator ("soft kernel").

Result From Running the System

The output when running the example depends on what kind of trace has been enabled. If you set `XMSC_TRACE` it should be similar to the following:

```
System_Init_Proc: instancehead process Environment;
msc RTOS_Trace;
Pr11: instancehead process Pr1;
Pr12: instancehead process Pr1;
Pr11: condition Idle;
Pr12: condition Idle;
Pr11: in Go,0 from MyExtTask3;
Signal Go received in Pr1:Instance1
dynpr14: instancehead process dynpr1;
Pr11 : create dynpr14;
Pr11: condition Wait;
Pr12: in Go,1 from MyExtTask3;
Signal Go received in Pr1:Instance2
dynpr15: instancehead process dynpr1;
Pr12 : create dynpr15;
Pr12: condition Wait;
dynpr14: set T1,2 (5000); /* #SDTNOW(269) */
dynpr14: condition wait;
dynpr15: set T1,3 (5000); /* #SDTNOW(276) */
dynpr15: condition wait;
dynpr14: timeout  T1,2; /* #SDTNOW(5284) */
dynpr14: out Terminating,4 to Pr11;
dynpr14: endinstance;
Pr11: in Terminating,4 from dynpr14;
Pr11: out Ok,5 to MyExtTask3;
Ok received in MyExtTask with paramer = 1
dynpr15: timeout  T1,3; /* #SDTNOW(5463) */
dynpr15: out Terminating,6 to Pr12;
dynpr15: endinstance;
Pr12: in Terminating,6 from dynpr15;
Pr12: out Ok,7 to MyExtTask3;
Ok received in MyExtTask with paramer = 2
```

Setting the `XOS_TRACE` flag should result in the following output:

```
** Process Pr1:9901455 created **

** Process Pr1:9901456 created **

** Process instance 9901455 **
Pr1: nextstate Idle
```

```
** Process instance 9901456 **
Pr1: nextstate Idle

** Process instance 9901455 **
Pr1: input signal Go
Signal Go received in Pr1:Instance1
Pr1: process dynpr1:9901458 created
Pr1: nextstate Wait

** Process instance 9901456 **
Pr1: input signal Go
Signal Go received in Pr1:Instance2

** Process instance 9901458 **

DynPr1: Set timer T1
Process instance 9901459
DynPr1: nextstate wait

** Process instance 9901458 **
DynPr1: input signal T1
DynPr1: signal Terminating sent
DynPr1: stopped

** Process instance 9901455 **
Pr1: input signal Terminating
Pr1: signal Ok sent
Pr1: dash nextstate
Ok received in MyExtTask with parameter = 1

** Process instance 99014516 **
DynPr1: input signal T1
DynPr1: signal Terminating sent
DynPr1: stopped

** Process instance 99014513 **
Pr1: input signal Terminating
Pr1: signal Ok sent
Pr1: dash nextstate
Ok received in MyExtTask with parameter = 2
```

Standard Model

In the standard model each instance of the Pr1 and the DynPr1 processes will be represented by an OS task (in all four tasks). The environment is represented by a task called `MyExtTask`. This task is external to the SDL system.

Instance Set Model

In the instance set model there will be two OS tasks, one each for Pr1 and DynPr1. The environment is represented by a task called `MyExTask`, just as in the standard model. This task is external to the SDL system.

How Signals are Sent to and from the Environment

There is an external task called `MyExtTask` which is written in C. It sends the signal Go into the SDL system and receives the signal Ok back by using services in the operating system.

The `HOOK_MAIN_AFTER_PROCESS_RELEASE` macro in `scttypes.h` is used to create `MyExtTask` as soon as the SDL system allowed to run.

The source code for the external task is placed in the file `MyExtTask.c`. This file is specific to the selected operating system because it calls the operating system directly.

Tight Integration Code Reference

This section explains data types, procedures and macros used in a Tight Integration (Light Integrations are explained in the Master Library).

General Macros

XPP(x)

The macro `XPP` is used in function declarations to specify the function parameters. It is defined like this:

```
#define XPP(x)  x
```

if function prototypes according to ANSI C can be used.

xptrint

The following type is also always defined:

```
#define xptrint unsigned
```

where `xptrint` should be an `int` type with the same size as a pointer.

xPrsNode and xPrdNode

```
typedef struct xPrsStruct  *xPrsNode;
typedef struct xPrdStruct  *xPrdNode;
```

`xPrsNode` and `xPrdNode` are pointers to structs holding instance data for an instance of a process or a procedure. Note that some parts of the structs are OS-dependent.

xInputAction, xNotInSignalSet ...

These defines specify the different ways of handling a signal.

Macros to Exclude Unnecessary Code

The following macros are defined to exclude unnecessary code for `IdNode` variables etc.:

```
#define XNOSTARTUPIDNODE
#define XOPTSIGPARA
#define XOPTDC
#define XOPTFPAR
#define XOPTSTRUCT
#define XOPTLIT
```

```
#define XOPTSORT
#define XNOUSEOFSERVICE
......
```

Macros to activate Signal-Free-Functions

The following macro must be defined to activate the Signal-Free-Functions. This is necessary if signals and timers with string parameter (dynamic allocated) are used - to avoid memory leaks.

```
#define XFREESIGNALFUNCS
```

If strings not are used as parameters in signals this flag should not be set cause it does lead to some performance deterioration. (Normally this define is set in the make-file).

For timer string parameters the define

```
#define XTIMERSWITHSTRINGPARAMS
```

must be set.

The following macro must be defined to activate the Signal-Free-Functions. This is necessary if signals and timers with string parameter (dynamic allocated) are used - to avoid memory leaks.

```
#define XFREESIGNALFUNCS
```

Default Priorities

One group of macros defines default priorities for processes and signals:

```
#ifndef xDefaultPrioProcess
#define xDefaultPrioProcess      RTOSPRIODEFAULT
#endif

#ifndef xDefaultPrioSignal
#define xDefaultPrioSignal       RTOSPRIODEFAULT
#endif
...
```

Macros to Implement SDL

First in this section is a macro defining the symbol table root:

```
xIdNode         xSymbolTableRoot;
```

XPROCESSDEF_C and XPROCESSDEF_H

These macros define the start-up function for a PAD function. It is this function that is called in the task creation. As mentioned before this function will after some variable initializations call the PAD function. The definition of the start-up function varies in different RTOS, that is why there is a second OS-specific macro here.

STARTUPSIGNAL, ALLOCPRSSIGNAL, etc.

Each signal in an application is assigned a unique integer value. The values 31992 through 32000 are reserved for internal signals like the start-up signal.

Variables in the PAD Function

PROCESS_VARS, PROCEDURE_VARS

These macros define the elements in the `xPrsStruct` and `xPrdStruct` respectively.

YPAD_YSVARP

This macro defines a variable representing a pointer to a signal's parameter area.

YPAD_YVARP

This macro defines the variable `yVarP` which represents the process instance data.

LOOP_LABEL, LOOP_LABEL_PRD, LOOP_LABEL_PRD_NOSTATE

These three macros define the eternal loop inside processes, procedures and procedures without a state.

START_STATE

Each state in a process and procedure is represented as an integer. The Start state will always have the value 0.

Using OSE Trace Features

Note:

Telelogic has noticed that the OSE-trace feature can make the application crash in some situations. This seems to happen when a SDL process (OSE-task) sends a signal immediately before terminating. If you come across this problem, first check if the application works correctly when generated without OSE-trace.

Annex 1: Integration for OSE Delta

Introduction

This annex briefly describes the OSE Delta models and primitives used in the SDL suite OSE tight integration. The presentation is focused on the differences from the OSE classic model described in the previous annex.

One section describes how to set up and run a simple test example in both a light and tight integration.

> **Note:**
> Third-party products referred to in this manual may have limitations that have impact on the usability of Telelogic Tau. Please consult the supplier's support organization or the third-party product's technical reference documentation for up-to-date information about such limitations.

Principles

This integration is developed with OSE Delta Soft Kernel 3.2 and tested on a Sun workstation with SunOS Release 5.6.

The main differences between the OSE Delta and the OSE Classic model are:

- The OSE Delta model uses three semaphores to avoid synchronization problems in SDL start transitions.
- The timer is implemented in `systimer.c`, which is supplied by ENEA. This is not accurate and is only for demonstration purposes. You will have to supply a suitable timer implementation for the target environment.

Running the Test Example: Simple

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from Telelogic Support web site.

Prerequisites

This test example is developed as an OSE Delta application on a Sun workstation. The makefile and compilation switches are set up for the application to run under an OSE Simulator for OS68. If you are using another configuration of OSE you probably need to edit the provided makefile.

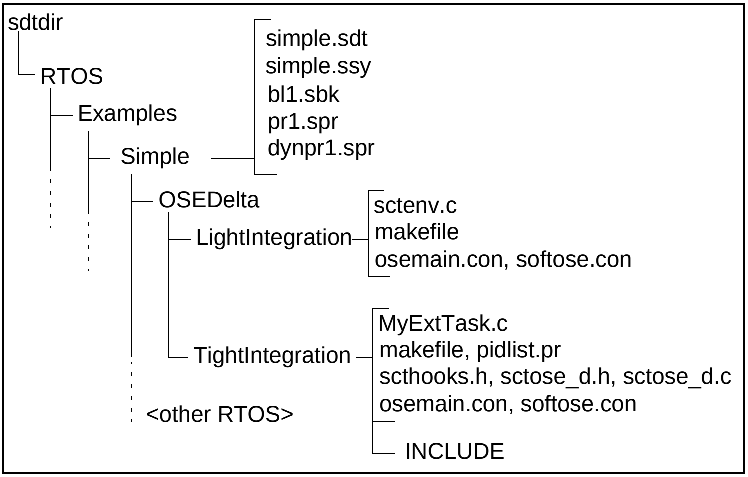


Figure 568: File structure for the Simple example

Light Integration

Limitations for the Light Integration

Please see the Release Guide.

Building a Light Integration

Please see the "Building and Running a Light Integration" on page 3268 for instructions.

Tight Integration

Limitations for the Tight Integration

Please read the Release Guide for details about limitations that apply to all systems using Tight Integration.

Building a Tight Integration

Please see the "Building and Running a Tight Integration" on page 3270 for instructions.

How Signals are Sent to and from the Environment.

The signal Go is sent from an external task `MyExtTask`. The code for this task is placed in the program file `MyExtTask.c`. This is the same as used for OSE Classic.

Annex 2: Integration for VxWorks

Introduction

This annex describes briefly the VxWorks models and primitives used in the SDL Suite VxWorks tight integration. The presentation is focused on the differences from the general model described earlier in this chapter.

One section describes how to set up and run a simple test example in both a light and tight integration.

> **Note:**
>
> Third-party products referred to in this manual may have limitations that have impact on the usability of Telelogic Tau. Please consult the supplier's support organization or the third-party product's technical reference documentation for up-to-date information about such limitations.

Principles

This integration is developed with VxWorks Tornado 1.0 version and tested under VxSim version 5.3 on a Sun workstation with SunOS 4.1.4.

The main differences between VxWorks and the general model are:

- The VxWorks `msgQReceive` copies the Signal into a buffer when it is received. The sender makes free of the signal immediately after it has been sent and the receiver allocates a buffer (signal) before a receive statement.
- An extra optimization flag `XOPTSIGNALALLOC` has been introduced for the VxWorks tight integration. When freeing a signal’s memory, we place it in an availlist so that subsequent signal memory allocation calls can check to see if suitable sized memory already exists which may be reused. Otherwise memory is allocated as normal.

Running the Test Example: Simple

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from Telelogic Support web site.

Prerequisites

This test example is developed as a VxWorks Tornado application on a Sun workstation. The makefile and compilation switches are set up for the application to run under an VxSim target simulator. If you are using another configuration of VxWorks you probably need to edit the provided makefile.

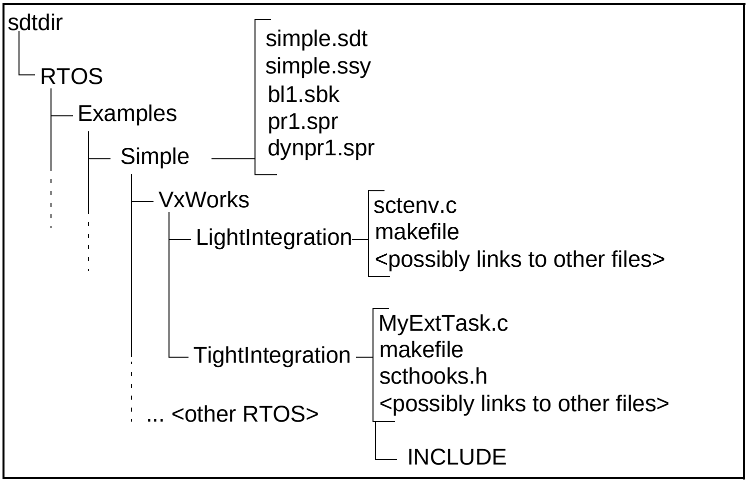


Figure 569: File structure for the Simple example

Note:

A VxWorks application is not allowed to contain a main function. The name of the generated main is changed to "root" with the compilation switch `-DXMAIN_NAME=root`.

Light Integration

Limitations for the Light Integration

Please see the Release Guide.

Building a Light Integration

Please see the “Building and Running a Light Integration” on page 3268 for instructions.

Tight Integration

Limitations for the Tight Integration

Please read the Release Guide for details about limitations that apply to all systems using Tight Integration.

Building a Tight Integration

Please see the “Building and Running a Tight Integration” on page 3270 for instructions.

Annex 3: Integration for Win32

This annex briefly describes integration with Win32. The presentation is focused on the differences from the general model described earlier in this chapter.

Note:

Third-party products referred to in this manual may have limitations that have impact on the usability of Telelogic Tau. Please consult the supplier's support organization or the third-party product's technical reference documentation for up-to-date information about such limitations.

Principles

This integration is developed using the Microsoft 32-bit C/C++ Compiler Version 11.00.7022 and tested on NT 4.0, NT 3.51 and Windows 95 platforms. The integration is also compiled with Borland C++ 5.2 for Win32 and tested on NT 4.0, NT 3.51 and Windows 95 platforms.

The main differences between integration with Win32 and the general model are:

- Threads are created with the Win32 primitive CreateThread(). The thread is then automatically given an input queue the first time it calls a USER or GDI function
- xAlloc is implemented with Win32 function `HeapAlloc()`.
- xFree is implemented with Win32 function `HeapFree()`.
- The timer implementation uses the Win32 `GetTickCount()` function.

Running the Test Example: Simple

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from Telelogic Support web site.

Prerequisites

This test example is developed as a Win32 console application on a PC. The makefiles and compilation switches are set up for the application to compile using either the Borland or the Microsoft compiler listed above. There is a separate makefile for each compiler.

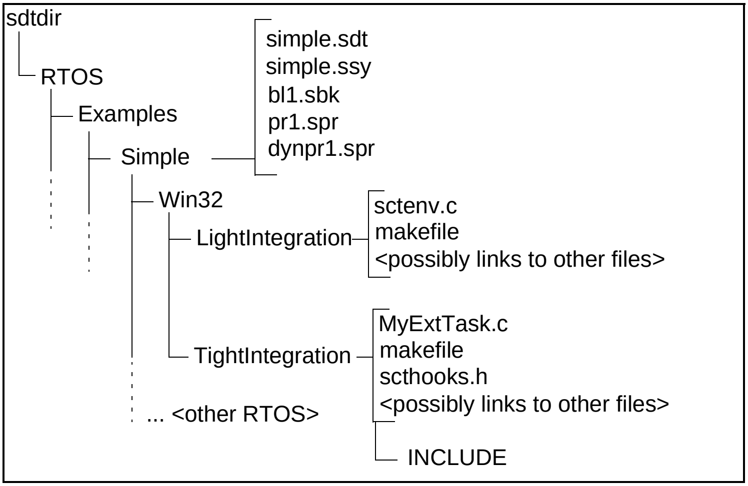


Figure 570: File structure for the Simple example

Light Integration

Limitations for the Light Integration

Please see the Release Guide.

Building a Light Integration

Please see the "Building and Running a Light Integration" on page 3268 for instructions.

There are different makefiles provided for Borland compilers and Microsoft compilers.

Tight Integration

Limitations for the Tight Integration

Please read the Release Guide for details about limitations that apply to all systems using Tight Integration.

Building a Tight Integration

Please see the "Building and Running a Tight Integration" on page 3270 for instructions.

There are different makefiles provided for Borland compilers and Microsoft compilers.

> **Note:**
>
> The command line length limitation for the Borland compiler can sometimes be exceeded. If this happens, you should define the `DEFINE MACROS` at the beginning of the `sctwin32.h` file.

Compiler Flags

The following defines (`#ifdef`) are used in this integration:

- `WIN32_INTEGRATION`: Ensures that the `sctwin32.h` file is included in each C file. Must be set in all cases.
- `XOS_TRACE`: Gives a textual trace for most of the SDL events by using printf to some device. This flag should not be used together with `XMSC_TRACE`.
- `XMSC_TRACE`: Will give a textual trace in the format of MSC/PR Z.120 by using printf. This trace is possible to view in the MSC Editor included in Telelogic Tau. This flag should not be used together with `XOS_TRACE`.
- `XMSC_EDITOR`: Used together with the `XMSC_TRACE` flag, the MSC trace is automatically displayed in the MSC Editor. Note that you must have the Organizer open on your machine.
- `X_ONE_TASK_PER_INSTANCE_SET`: States that the Instance Set Model is used. The Standard Model is otherwise chosen by default.
- `XERR`: When this flag is defined, the return status of all Win32 function calls will be printed.

- `XINCLUDE_HS_FILE`: Includes the system signal header file which is required for tight integrations. This file maps signal names to integers.
- `XRTOSTIME`: Should always be set for all tight integrations.
- `XUSING_SCCD`: This should be set when using the preprocessor SCCD to ensure that the windows header files are not included on the preprocessor pass. The files are included though on the compiler pass and this ensures that the preprocessed C files only contain the expanded the SDL suite macros. It also helps greatly to speed up the process. Note that this flag only works with the Microsoft compiler and should not be used with any other compiler.
- `XWINCE`: This flag allows you to compile the integration for Microsoft WinCE target systems. This flag should not be used together with the MSC trace flags.

Annex 4: Integration for Solaris 2.6

Introduction

This annex describes briefly the Solaris 2.6 model and primitives used in the SDL Suite Solaris 2.6 tight integration. The presentation is focused on the differences from the general model described earlier in this chapter.

One section describes how to set up and run a simple test example for a tight integration.

Note:

The Solaris 2.6 tight integration is fully POSIX compliant. For this reason it will not work with earlier versions of Solaris.

Note:

Third-party products referred to in this manual may have limitations that have impact on the usability of Telelogic Tau. Please consult the supplier's support organization or the third-party product's technical reference documentation for up-to-date information about such limitations.

Principles

This integration is developed using cc:WorkShop Compilers 4.2 with Solaris 2.6 running on a workstation.

The main differences between the Solaris 2.6 integration and the general model are:

- In the file `sctsolaris.h`, the macros which contain the Solaris 2.6 specific function calls are implemented. The file `sctsolaris.c` contains the Solaris 2.6 integration specific functions.
- The Solaris 2.6 integration is fully POSIX compliant. SDL processes are mapped to POSIX threads using the `pthread_create()` function and POSIX queues are created for each thread using `mq_open()`. The threads are suspended when its corresponding queue is empty.

Running the Test Example: Simple

Note:

The source file and examples for Tight Integrations are not included in the standard delivery. They are available as free downloads from Telelogic Support web site.

Prerequisites

This test example is developed as a Solaris 2.6 application on a workstation. The makefile and compilation switches are set up for the application to run under Solaris 2.6 using the cc:WorkShop Compilers 4.2 compiler.

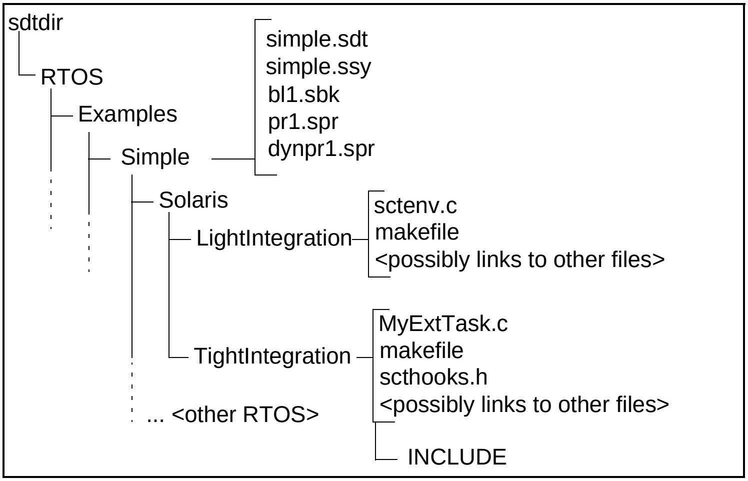


Figure 571: File Structure for the Simple example.

Light Integration

Limitations for the Light Integration

Please see the Release Guide.

Building a Light Integration

Please see the "Building and Running a Light Integration" on page 3268 for instructions.

Tight Integration

Limitations for the Tight Integration

Please read the Release Guide for details about limitations that apply to all systems using Tight Integration.

Building a Tight Integration

Please see the “Building and Running a Tight Integration” on page 3270 for instructions.

Compiler Flags

The following defines (`#ifdef`) are used in this integration:

- `SOLARIS_INTEGRATION`: Ensures that the `sctsolaris.h` file is included in each C file. Must be set in all cases.
- `XOS_TRACE`: Gives a textual trace for most of the SDL events by using printf to some device. This flag should not be used together with `XMSC_TRACE`.
- `XMSC_TRACE`: Will give a textual trace in the format of MSC/PR Z.120 by using printf. This trace is possible to view in the MSC Editor included in Telelogic Tau. This flag should not be used together with `XOS_TRACE`.
- `XMSC_EDITOR`: Used together with the `XMSC_TRACE` flag, the MSC trace is automatically displayed in the MSC Editor. Note that you must have the Organizer open on your machine.
- `X_ONE_TASK_PER_INSTANCE_SET`: Should be defined when the alternative runtime model is to be used.
- `XINCLUDE_HS_FILE`: Includes the system signal header file which is required for tight integrations. This file maps signal names to integers.
- `XRTOSTIME`: Should always be set for all tight integrations.

Annex 5: Generic POSIX Tight Integration

Introduction.

Note:

The Generic POSIX integration is based on the Solaris integration. For a description and instructions on how to generate and run the example see “Annex 4: Integration for Solaris 2.6” on page 3288.

Annex 6: Building a Threaded Integration

Introduction

This Tutorial, on how to create a Threaded integration, is developed on a Windows machine and is intended to be run under a Windows OS. If you want to use this example on another machine and for another OS, please remember to choose the appropriate integration and compiler for your OS in the Targeting Expert.

Preparations

The same SDL source files for the example Simple, that is used in the Light integration example will be used in this tutorial.

Copy the Source files for the Example: Simple

1. Create your own test directory and enter it.
2. Copy the SDL source files for the example Simple:

   ```
   cp <installation>/sdt/sdtdir/RTOS/Example/Sim-
   ple/*.s*
   ```

3. Copy the environment file from the Win32/ThreadedIntegration directory:

   ```
   cp <your Installation>/sdt/sdtdir/RTOS/Examples/Sim-
   ple/Win32/ThreadedIntegration/MyExtTask.c
   ```

4. Start the SDL suite and open the system file for the Simple example.

Partition the system using the Deployment Editor

1. Create a new deployment diagram and call it Simple.
2. Edit the deployment diagram according to the figure below, see : Signalling in Threaded Integration.

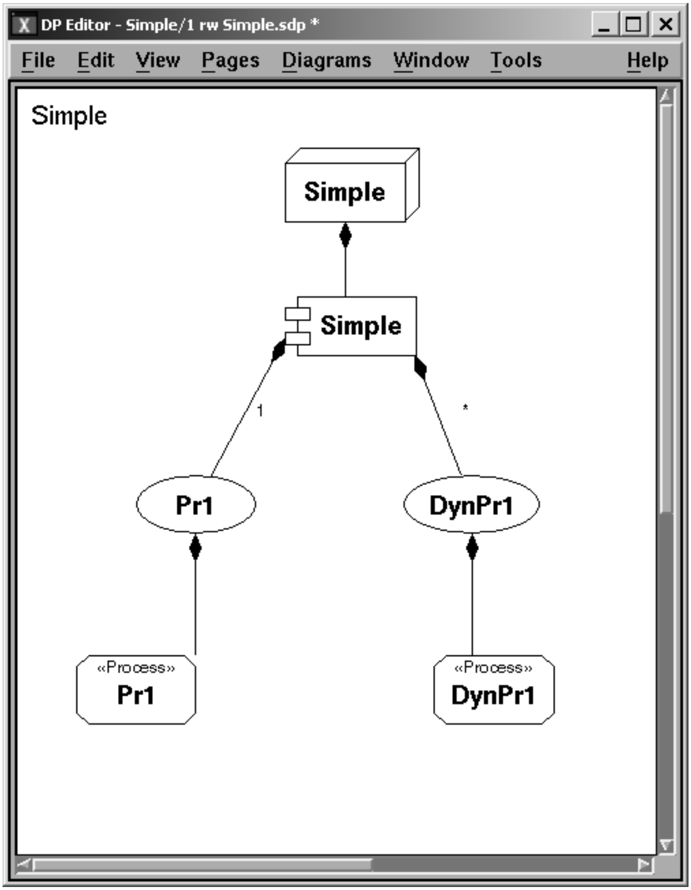


Figure 572 Deployment Diagram for the example Simple

3. Make sure that you got the multiplicity right on the aggregation line from the component to thread.

   To check this you can double click on the line. The right value should be:

   – 1 - On aggregation line from component to thread Pr1.

   – * - On aggregation line from component to thread DynPr1.

The multiplicity on the aggregation lines specifies how the component should be mapped to threads.

– A ‘*’ means that each instance of the component should be mapped to an individual thread.

– A name means that the entire component should be mapped to a thread.

In our example this means that there should be one thread for **all** instances of Pr1 and one thread for **each** instance of DynPr1.

4. Double-click the component symbol. In the Symbol Details window specify that the integration model should be Threaded, see : Signalling in Threaded Integration.

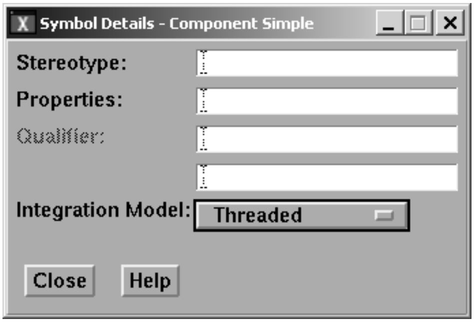


Figure 573 Symbol Details for the Component Simple

5. Double-click the thread symbol for Pr1. Specify the following Thread Parameters for the Thread P1:

   Thread Stack Size = 2048
   Thread Priority = 8
   Queue Size = 128
   Max Signal Size = 1024

6. Double-click the object symbol and specify that the stereotype should be Process and that the qualifier (for Pr1) should be:

   – Simple/Bl1/Pr1.

   Make the appropriate specifications for the object Symbol for DynPr1(Qualifier = Simple/Bl1/DynPr1).

7. Save the deployment diagram.
8. Select the deployment diagram in the Organizer and open the Targeting Expert from the Generate menu.
9. Choose the integration: Threaded Integrations->Win32 threaded.
10. You will be prompted if you want to generate the sdl_cfg.h file. Select No!
11. Disable the Generation of Environment Function by deselecting Environment Functions in the Environment section of the window.
12. Click on the Compiler/Linker/Make line in the Partitioning Diagram Model. You should now see the following in the Targeting Expert window, see:: The Compile/Linker/Make Window in Targeting Expert.

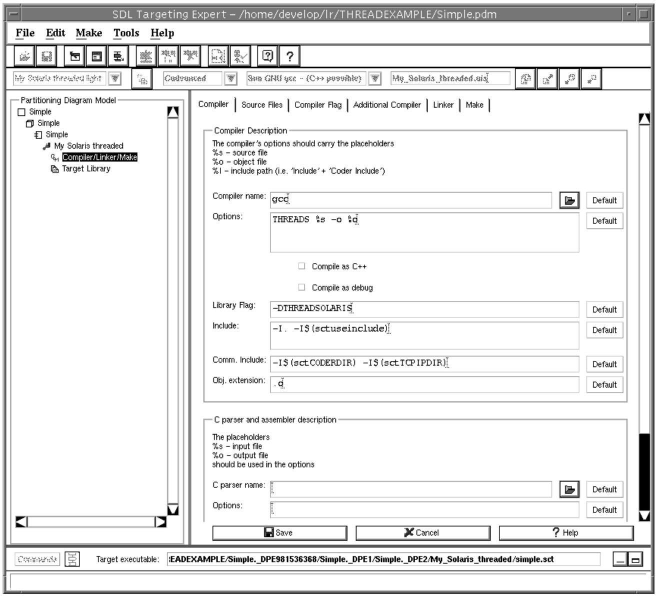


Figure 574: The Compile/Linker/Make Window in Targeting Expert.

13. Define the following flag: THREADED_SIMPLE_EXAMPLE in the Compiler description/Options window. This flag will start the External threads in the simple example.

14. Define the following Compilation flags:

    – THREADED_XTRACE,

    – THREADED_MSCTRACE

    by selecting the flags: *SDL trace and MSC* trace in the *Target library/Kernel window*

15. Add the `MyExtTask.c` file as a new source file.

    Click on the Source Files entry in the window and add the `MyExtTask.c` file in the source file list.

    Save the settings. You are now prompted again to generate configuration file, this time select Yes.

16. You should now be back in the Analyze/Generate code window and be ready to generate the application.

    Do a full Make and if you have followed the instructions the Targeting Expert will now analyze, generate code, generate makefile, compile and link the application.

17. Run the application simple.sct. Please note that you have to traverse down in the generated directory structure to find the application.

    You will find the application in a subdirectory similar to this path:

    – `<your test directory>/Simple._DPE981536368/Simple._DPE1/Simple._DPE2/Win32_threaded/....`

The output you should see when you run the application should be as follows:

```
Connected with the Postmaster.
*   OUTPUT of go to Pr1:1
*      Parameter(s) : 1
*** NEXTSTATE  Idle
*   OUTPUT of go to Pr1:2
*      Parameter(s) : 2
*** NEXTSTATE  Idle
Signal Go received in Pr1:Instance1
*   CREATE DynPr1:1
*** NEXTSTATE  Wait
Signal Go received in Pr1:Instance2
*   CREATE DynPr1:2
*   SET on timer t1 at 17.5580
*** NEXTSTATE  Wait_t1
*** NEXTSTATE  Wait
*   SET on timer t1 at 17.5880
*** NEXTSTATE  Wait_t1

*** TIMER signal was sent
*   Timer    : t1
*   Receiver : DynPr1:1
*** Now      : 17.5650
*   OUTPUT of t1 to DynPr1:1
*   PROCEDURE START Prd
Not doing much
*   PROCEDURE RETURN Prd
*   OUTPUT of terminating to Pr1:1

*** TIMER signal was sent
*   Timer    : t1
*   Receiver : DynPr1:2
*** Now      : 17.6050
*   OUTPUT of t1 to DynPr1:2
*** STOP   (no signals were discarded)
*   PROCEDURE START Prd
*   OUTPUT of ok to env:1
*      Parameter(s) : 1
Signal Ok received with the following parameter:1
*** NEXTSTATE  Wait
Not doing much
*   PROCEDURE RETURN Prd
*   OUTPUT of terminating to Pr1:2
*** STOP   (no signals were discarded)
*   OUTPUT of ok to env:1
*      Parameter(s) : 2
Signal Ok received with the following parameter:2
*** NEXTSTATE  Wait
```

Figure 575 Textual SDL trace for Simple example.

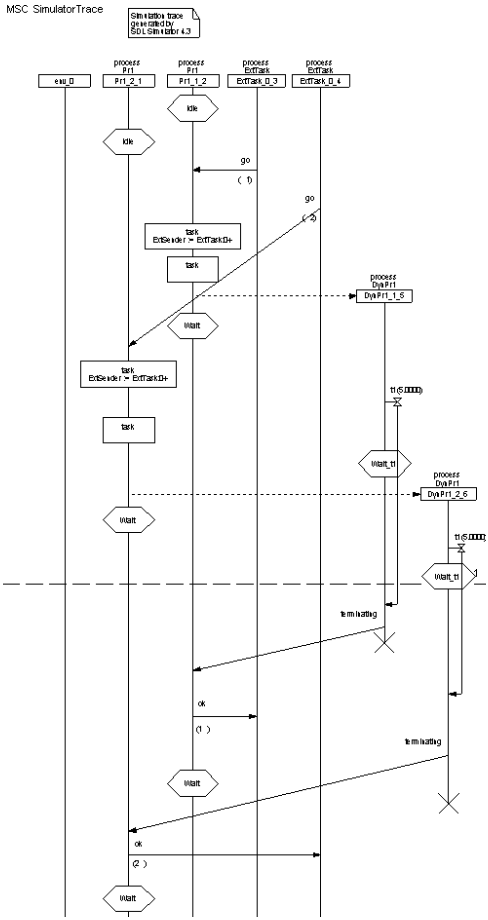


Figure 576 MSC trace for Simple Example