High Performance
Real-Time Operating Systems

Modeling

SCIOPTA Systems

Copyright

Copyright (C) 2023 by SCIOPTA Systems GmbH.. All rights reserved. No part of this publication may be reproduced, transmitted, stored in a retrieval system, or translated into any language or computer language, in any form or by any means, electronic, mechanical, optical, chemical or otherwise, without the prior written permission of SCIOPTA Systems GmbH. The Software described in this document is licensed under a soft-ware license agreement and maybe used only in accordance with the terms of this agreement.

Disclaimer

SCIOPTA Systems GmbH, makes no representations or warranties with respect to the contents hereof and spe-cifically disclaims any implied warranties of merchantability of fitness for any particular purpose. Further, SCIOPTA Systems GmbH, reserves the right to revise this publication and to make changes from time to time in the contents hereof without obligation to SCIOPTA Systems GmbH to notify any person of such revision or changes.

Trademark

SCIOPTA is a registered trademark of SCIOPTA Systems GmbH.

Contact

Corporate Headquarters
SCIOPTA System GmbH
Hauptstrasse 293
79576 Weil am Rhein
Germany
Tel. +49 7621 940 919 0
Fax +49 7621 940 919 19
email: sales@sciopta.com
www.sciopta.com

Document No. S09135RL1

Table of Contents

1. Introduction 1-1

1.1 SCIOPTA Metamodel 1-1
1.2 About SCIOPTA 1-1
1.3 Standard Modeling Languages 1-2
1.3.1 The UML Unified Modeling Language 1-2
1.3.2 The SDL and SDL-RT 1-2

2. Class Diagram 2-1

2.1 Overview 2-1
2.2 SCIOPTA System Class Diagram 2-1
2.3 SCIOPTA Application Class Diagram 2-2
2.4 SCIOPTA Application Object Diagram 2-3
2.5 Module Classes 2-4
2.5.1 Module Class Symbols 2-4
2.6 Process Classes 2-4
2.6.1 Process Class Symbols 2-4
2.6.2 Dynamic Process Classes 2-5
2.7 Standard Classes 2-5
2.8 Message Classes 2-6
2.8.1 Message Class Symbol 2-6
2.8.2 Example 2-7

3. SCIOPTA State Diagram 3-1

3.1 Introduction 3-1
3.2 Overview 3-1
3.3 Process Start 3-4
3.4 Process Kill (Return) 3-4
3.5 State 3-5
3.6 Time-out 3-5
3.7 Message Input 3-6
3.8 State Examples 3-7
3.8.1 Blocking State Example 3-7
3.8.2 Blocking State with Time-out Example 3-9
3.8.3 Non-Blocking State Example 3-11
3.8.4 State Example Including Sender Process 3-13
3.9 Message Object Output 3-15
3.9.1 Message Object Output Example 3-15
3.10 Message Object Creation 3-17
3.10.1 Message Object Creation Example 3-17
3.11 Message Object Free 3-19
3.12 Action 3-19
3.13 Decision 3-20
3.13.1 “Decision” Example 3-20
3.14 Switch-Case Decision 3-22
3.14.1 “Switch-Case Decision” Example 3-22
3.15 Timer Start 3-23
3.15.1 “Timer Start” Example 3-23
3.16 Timer Remove 3-25
3.16.1 “Timer Remove” Example 3-25

Modeling SCIOPTA Systems

3.17 Process Object Creation 3-26
3.18 Operation Call 3-27
3.19 Operation Start 3-27
3.20 Operation Return 3-27
3.21 Transition Option 3-28
3.21.1 "Transition Option" Example 3-28
3.22 Connectors 3-30
3.23 Comments 3-30

4. SCIOPTA Sequence Diagram 4-1

4.1 Introduction 4-1
4.2 SCIOPTA Scheduling 4-1
4.3 Lifelines 4-2
4.4 State 4-2
4.5 Activation Box 4-3
4.6 Asynchronous Messages 4-3
4.7 Synchronous Messages 4-4
4.8 Timer Start 4-5
4.9 Timer Time-out 4-5
4.10 Timer Remove 4-6

5. Document Revisions 5-1

5.1 Document Version 1.0 5-1
5.2 Document Version 1.0 5-1

6. Index 6-1

Modeling SCIOPTA Systems

1 Introduction

This document presents the preferred way for embedded software engineers to communicate and organize their SCIOPTA designs, using present-day modeling techniques. In other words, a SCIOPTA-oriented domain-specific language (DSL) is informally defined in the form of a metamodel and explained herein. Industry experts both familiar with the philosophy of SCIOPTA and leading modeling tools contributed to this document.

1.1 SCIOPTA Metamodel

The SCIOPTA metamodel presented here is based on the UML metamodel. The SCIOPTA process classes can be modeled by using specific state diagrams using a subset of the standard UML state diagram syntax.

Standard modeling tools are often used to model a wide variety of systems including embedded real-time systems. As these tools do not support all features of a possible underlaying real-time operating systems, these features are often underused. This metamodel directly supports all features available in SCIOPTA.

SCIOPTA systems using this metamodel can easily be designed by using standard flowchart design tools such as Microsoft's® Vision® or even OpenOffice Draw.

There is no automatic code generation yet available for the SCOPTA model. But it should be very easy to transform the graphical notation into code including the SCIOPTA system calls.

If advanced features such as model integrity checking or code generation are needed PragmaDev®'s Real Time Developer Studio or IBM's® Telelogic® Rhapsody® may be used.

1.2 About SCIOPTA

SCIOPTA is a fully pre-emptive, priority controlled, direct message passing real-time operating system with exceptional features and a very high performance. Direct message passing means that interprocess communication (IPC) and synchronisation is done exclusively by messages. SCIOPTA messages have an ID and a data field of variable size which is accessible by the user and a header which cannot be accessed directly. The header includes system information such as owner, sender and addressee process and message size. SCIOPTA system calls are using this header information extensively to realize very efficient functionality. Messages can be seen as objects where the message ID is a message type or class. The SCIOPTA processes are working with pointers to the message objects. Only the message owner (by allocating or receiving messages) have direct access to the message data and therefore there is no need to protect data by using semaphores or other mutual access protection methods found in old traditional real-time kernels.

Please consult the SCIOPTA kernel manuals for a detailed description of the SCIOPTA methods.

1.3 Standard Modeling Languages

A modeling language is textual or graphical language that can be used to express systems and information in a structure that is defined by a consistent set of rules. The rules are used for interpretation of the meaning of components in the structure.

1.3.1 The UML Unified Modeling Language

The Unified Modeling Language (UML) is a standardized modeling language which is mainly used in the field of software engineering. The UML includes a set of graphical notation techniques to create abstract models of specific systems.

For more information: http://www.uml.org/

UML Tools

IBM's® Telelogic® Rhapsody® is a UML/SysML-based model-driven development for real-time or embedded systems.

For more information: http://www.ibm.com/

There is a SCIOPTA integration for Rhapsody® available.

1.3.2 The SDL and SDL-RT

The SDL (Specification and Description Language) is a specification language to be used at the unambiguous specification and description of the behaviour of reactive and distributed systems. It is defined by the ITU-T (Recommendation Z.100.) Originally focused on telecommunication systems, its current areas of application include process control and real-time applications in general.

SDL-RT is based on the SDL standard from ITU extended with real time concepts. V2.0 has introduced support of the UML from OMG in order to extend SDL-RT usage to static part of the embedded software and distributed systems.

For more information: http://www.sdl-rt.org/

SDL/SDL-RT Tools

PragmaDev's® Real Time Developer Studio includes a set of graphical software modelling tools based on SDL, SDL-RT and UML.

For more information: http://www.pragmadev.com/

There is a SCIOPTA integration for PragmaDev's® Real Time Developer Studio available.

2 Class Diagram

2.1 Overview

Standard UML Class Diagram can be used to show the classes and their relations of a SCIOPTA system. Except the operations no dynamic behaviour is shown.

This is not an introduction into UML Class Diagram. Please consult the specific UML 2.0 literature for more information.

2.2 SCIOPTA System Class Diagram

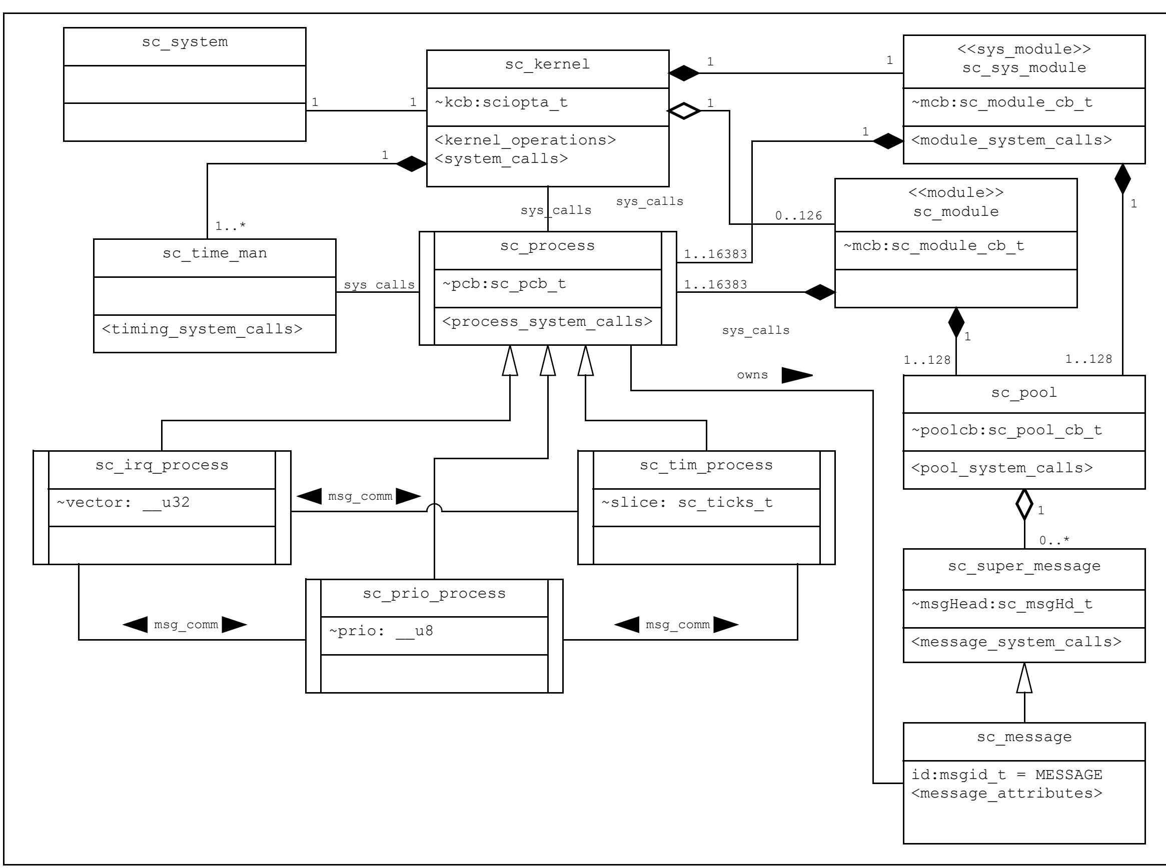


Figure 2-1: Simplified Abstract Class Diagram of a SCIOPTA System

Figure 2-1: shows the upper level kernel classes of a SCIOPTA system. These are abstract classes and will not be directly instantiated. The attributes contain the kernel internal (private) data, such as kernel-, module-, pool- and process-control-blocks. The operations include beside private kernel functions the public kernel system calls.

A SCIOPTA System has one SCIOPTA kernel represented by the class **sc_kernel**. The main attribute of the kernel class is the kernel control block (kcb) which is not accessible by the user. SCIOPTA kernel class operations contain all public system calls which are not included in the other system classes.

A SCIOPTA kernel has always one system module represented by the class **sc_sys_module** and the stereotype <<sys_module>>. The main attribute of the system module class is the module control block (mcb) which is not accessible by the user. The system module class operations include the public module system calls. Additionally a SCIOPTA kernel can have up to 126 modules represented by the class **sc_module** and the stereotype <<module>>.

A SCIOPTA module can have up to 16383 processes represented by the class **sc_process**. The main attribute of the process class is the process control block (pcb) which is not accessible by the user. The process class operations include the public process system calls. There are three specific process classes which are derived from the class **sc_process**: the interrupt process class (**sc_irq_process**), the prioritized process class (**sc_prio_process**) and the timer process class (**sc_tim_process**). The binary associations **sys_calls** show system call relations between the process class and the kernel system classes. Binary associations named **msg_comm** showing message passing channels between processes.

A SCIOPTA module can have up to 128 pools represented by the class **sc_pool**, which holds the SCIOPTA messages. The main attribute of the message pool class is the pool control block (pool cb) which is not accessible by the user. The pool class operations include the public pool system calls.

A SCIOPTA message pool can have any number of SCIOPTA messages represented by the class **sc_message**. The number of messages is only limited by the memory size of the pool. There is a super class for the SCIOPTA message (**sc_super_message**) containing the message header attribute which is not accessible by the user. The message supper class operations include the public message system calls. The **sc_message** class is derived from the message super class and contains the user accessible message class identifier (message ID) and the message user attributes (message data). The binary association **owns** shows that a message always is owned by a process.

2.3 SCIOPTA Application Class Diagram

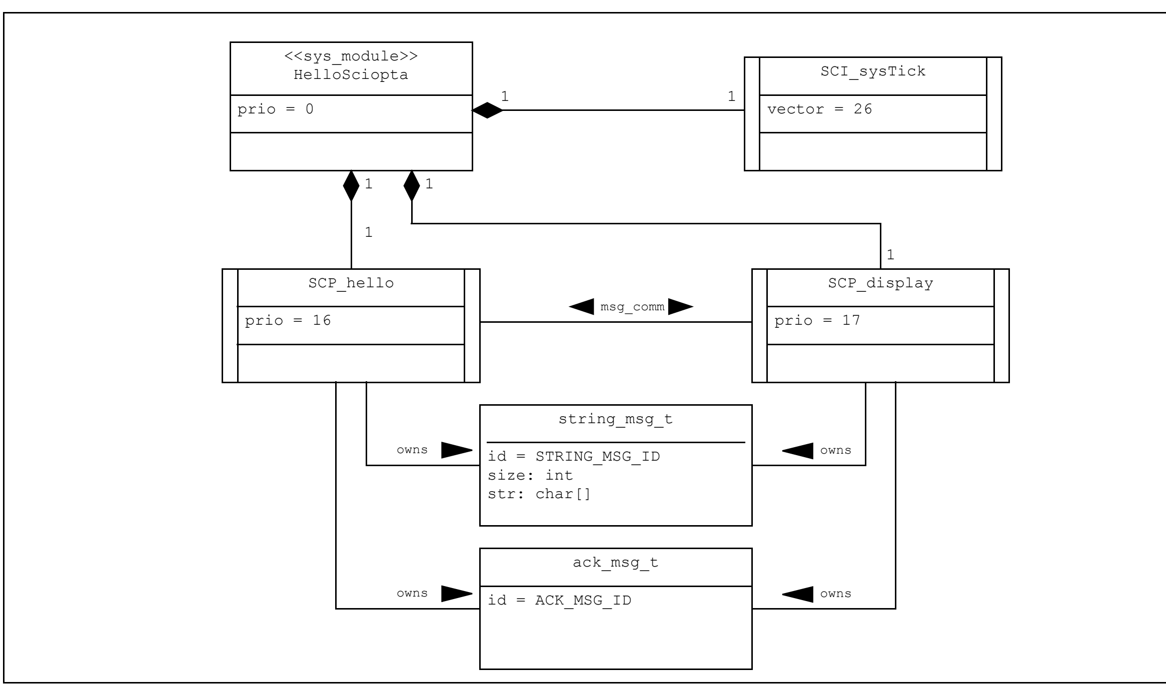


Figure 2-2: Class Diagram of a Simple Hello World Example

Figure 2-2: shows the user class diagram of a very simple SCIOPTA “Hello World” application. Please note that the super classes from where above user sub-classes are derived are not shown in an application class diagram, as the user has no access to these classes.

This simple system has only one module, the system module class “HelloSciopta”. A SCIOPTA system must have at least the system module.

There is an interrupt process class “SCI_sysTick” which implements the operating system tick timer functionality. The process classes “SCP_hello” and “SCP_display” communicate by exchanging message objects instantiated from message classes “string_msg_t” and “ack_msg_t”.

2.4 SCIOPTA Application Object Diagram

UML object diagrams use a notation similar to class diagrams and are used to show instances of classes at **a particular point in time**.

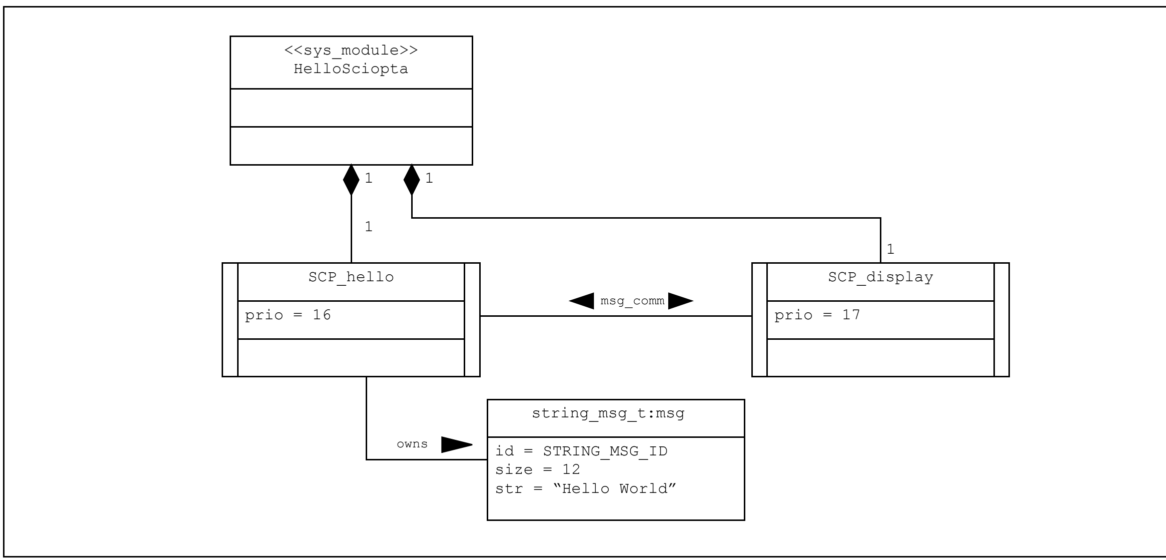


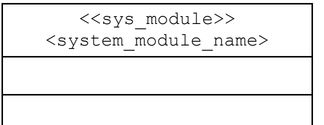
Figure 2-3: Simple Hello World Example Object Diagram

Figure 2-3: shows the object diagram of the system in **Figure 2-2:** after setup and before process object “SCP_hello” sends the message object “msg” to process object “SCP_display”. The message object “msg” is an instance of message class “string_msg” and was instantiated by process “SCP_hello”.

2.5 Module Classes

In SCIOPTA processes can be grouped into modules to improve system structure. There is always at least one system module in a SCIOPTA system. This module is called system module (sometimes also named module 0). Please consult the SCIOPTA - Kernel, User's Guides for more information about modules.

2.5.1 Module Class Symbols



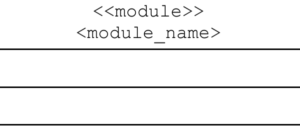


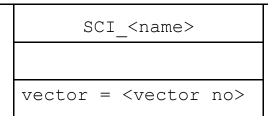
Figure 2-4: Module Classes

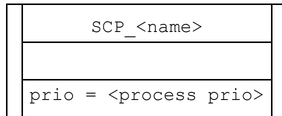
Stereotypes are used to differentiate between the SCIOPTA system module and the other modules. Specific private attributes such as module priority can be given. They are initialized when the module will be created.

2.6 Process Classes

An independent instance of a program running under the control of SCIOPTA is called process. SCIOPTA is assigning CPU time by the use of processes and guarantees that at every instant of time, the most important process ready to run is executing. Please consult the SCIOPTA - Kernel, User's Guides for more information about processes.

2.6.1 Process Class Symbols





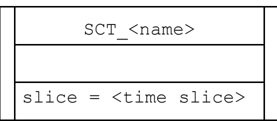


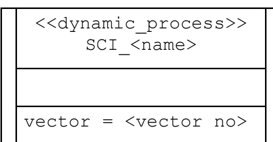
Figure 2-5: Process Classes

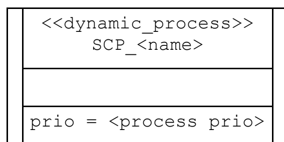
Process Class	Class Name	Typical Private Attributes
Interrupt Process	SCI_<name>	vector = <irq vector number>
Prioritized Process	SCP_<name>	prio = <process priority>
Timer Process	SCT_<name>	slice = <time slice>

The specific private attributes are initialized when the process is created. Class attributes and operations are declared as described in the UML norm.

If process classes have no stereotype they are considered to be **Static Processes Classes**. The kernel will instantiate static process classes automatically at system start from the static process classes.

2.6.2 Dynamic Process Classes





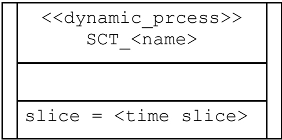


Figure 2-6: Dynamic Process Classes

Dynamic Process Classes are declared with the stereotype <<dynamic_process>>. Dynamic process objects will be instantiated in the code from the dynamic process classes. See also chapter 3.17 “Process Object Creation” on page 3-26.

2.7 Standard Classes

UML Symbols and declarations can be used for standard classes.

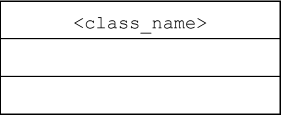


Figure 2-7: Standard Classes

Please note that public operations of standard classes are running in the context of the calling process.

2.8 Message Classes

SCIOPTA is a message based real-time operating system. Interprocess communication and co-ordination is done by messages. Message passing is a very fast, secure, easy to use method.

The SCIOPTA message class attributes include a public message identity (ID) which identifies the message class. The message class contains also private attributes (message header) which are internally used by the kernel and not directly accessible by the programmer.

Please consult the SCIOPTA - Kernel, User's Guides for more information about messages and direct message passing.

2.8.1 Message Class Symbol

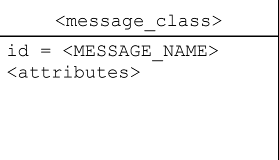


Figure 2-8: Message Class Symbol

The message attribute "msgid" identifies the class to be a message class.

C Code

```
// SCIOPTA style message declaration

#define <MESSAGE_NAME> (<number>)         // Message class ID (in upper case) which
                                          // defines the message class
typedef struct <message_name>_s {         // Struct name (lower case)
  sc_msgid_t id;                          // Stored message class ID
  member_type1 member1;                   // Attributes
  member_type1 member1;
  member_type1 member1;
            .
            .
            .
} <message_name>_t;                       // Message class (type)
```

2.8.2 Example

```
mymsg_t
-------------------
id = MYMSG
cval: __u32
cnum: __s16
time: sc_ticks_t
-------------------
```

Figure 2-9: Message Class Example

C Code

```
#define MYMSG (0x1000)
typedef struct mymsg_s{
  sc_msgid_t id;
  __u32      cval;
  __s16      cnum;
  sc_ticks_t time;
} mymsg_t;
```

Modeling SCIOPTA Systems

3 SCIOPTA State Diagram

3.1 Introduction

Process classes are modeled using a specific state chart. A combination of UML state diagram and SDL finite state diagram with some extensions is used as base for the SCIOPTA State Diagram. This process class model supports all features of the SCIOPTA real-time kernel such as direct message passing, selective receive system call, state time-outs, message queue management and others.

Please consult the SCIOPTA - Kernel, User's Guides for more information about messages and direct message passing.

In the code examples the message types are always casted. This avoids message union declarations as proposed in the kernel manuals and some examples in the SCIOPTA deliveries. Message type unions might not be accepted in some projects.

3.2 Overview

This is a list of all presented elements in alphabetical order.

Graphical Presentation	Element Name	Page
i = msg1->val_1;	Action	3.12
Comment text . .	Comments	3.23
A A	Connectors	3.22
val_1 == 5	Decision	3.13
mymsg_t msg txproc_pid	Message Input	3.7

Graphical Presentation	Element Name	Page
msg1_t msg1 plid = SC_DEFAULT_POOL tmo = SC_FATAL_IF_TMO val_1 = 5	Message Object Creation	3.10
msg1	Message Object Free	3.11
msg2 SCI_uart_pid	Message Object Output	3.9
set_off (lamp);	Operation Call	3.18
result	Operation Return	3.20
	Operation Start	3.19
	Process Kill	3.4
sc_pid_t sc_procPrioCreate (const char *name, void (*entry) (void), sc_bufsize_t stacksize, sc_ticks_t slice, sc_prio_t prio, int state, sc_poolid_t plid);	Process Object Creation	3.17
	Process Start	3.3
IDLE	State	3.5

Graphical Presentation	Element Name	Page
src	Switch-Case Decision	3.14
tim	Timer Remove	3.16
tim tmomsg onmsg->ton	Timer Start	3.15
tmo = 200	Time-out	3.6
DEBUG	Transition Option	3.21

3.3 Process Start

Figure 3-1: “Process Start” Element

This element represents the start of a process. It is the entry point into the process. Process initialization can be done from process start until the first state of the process.

C code:

```
SC_PROCESS (<process_name>)
{
        .
        .
        .
```

3.4 Process Kill (Return)

Figure 3-2: “Process Kill” Element

In SCIOPTA prioritized process will never return. They must be implemented as endless loops. Use the system call sc_procKill inside an actions element (see chapter 3.12 “Action” on page 3-19) to kill prioritized processes.

The “process Kill” element is used to show the return point of SCIOPTA interrupt (and timer) processes.

Modeling SCIOPTA Systems

3.5 State

<state_name>

Figure 3-3: “State” Element

The “State” element has name and means the process is waiting for some message objects.

The “State” element, the following “Message Input” and “Time-out” elements form a unified whole called “Process State”. All together are directly represented by SCIOPTA receive (sc_msgRx) system calls.

Please Note:

State names that appear in different parts of a diagram with the same name represent the same state.

3.6 Time-out

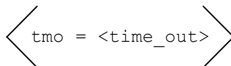


Figure 3-4: “Time-out” Element

The “Time-out” element always follows a “State” element. The elements following the “Time-out” element will be executed if the state time-out has expired.

There are three cases depending on the “Time-out” element which define the process state type and behaviour:

1. If there is no “Time-out” element defined after the “State” element, the “State” element blocks until one of the defined message objects are received. The elements after the “Message Input” element which has received the message object will be executed.
2. If there is a “Time-out” element defined with a time-out value (e.g. tmo = 100), the state blocks during the time-out period until one of the defined message objects are received. The elements after the “Message Input” element which has received the message object will be executed. If the time-out has expired and no message object has been received during that time, the elements after the “Time-out” element will be executed.
3. If there is a “Time-out” element defined with no time-out (tmo = 0) the state does not block at all and program flow continues. If there are message objects in the input queue, the elements after the “Message Input” element which has received the message object will be executed. If there is no message object in the input queue, the elements after the “Time-out” element will be executed.

3.7 Message Input

<message_class> <message_object>
<process_name>

Figure 3-5: “Message Input” Element

The “Message Input” element represents the message object of a message class which is received and/or a sender process from where the message object is expected in a state. It always follows a “State” element and if received the elements following the “Message Input” element are executed.

<message_class> is a defined message class in the style: <message_name>_t

<message_object> is the received object of the class <message_class>. The code must declare this object. It is usually the pointer to the received object.

If <message_class> is defined as an asterix (*) the next message object in the queue of any message class will be received.

If <message_class> is negated (strikethrough: <~~message_class~~>) the next message object in the queue of any message class except the given negated message class will be received.

<process_name> is the process (process ID) from where a message object will be received.

If <process_name> is defined as an asterix (*) the message objects from any process will be received.

3.8 State Examples

3.8.1 Blocking State Example

Figure 3-6: Blocking State Example

There is no “Time-out” element after state IDLE present. Therefore IDLE state will block until either the message object of message class sc_msg1_t or the message object of message class sc_msg2_t is received from any process. If a message object is received, the element after the corresponding “Message Input” element will be executed and msg1 or msg2 contains the pointers to the received message object.

C Code

```
/* Message definitions */

#define MSG1 (0x1)
typedef struct msg1_s{
   sc_msgid_t   id;
   int          val_1;
} *msg1_t;

#define MSG2 (0x2)
typedef struct msg2_s{
   sc_msgid_t   id;
   int          val_2;
} *msg2_t;

union sc_msg {
   sc_msgid_t   id;
};

/* Prioritized Process */

SC_PROCESS(SCP_example)
{
   static const sc_msgid_t select[3] = {
      MSG1, MSG2, 0
   };

   int i,k;

   sc_msg_t msg;
   msg1_t    msg1;
   msg2_t    msg2;

   for(;;){
      /* IDLE state */
      msg = sc_msgRx (SC_ENDLESS_TMO, select, SC_MSGRX_MSGID);

      switch( msg->id ){
         case MSG1:
            msg1 = (msg1_t)msg;
            i = msg1->val_1;
            sc_msgFree((sc_msgptr_t)&msg1);
         break;

         case MSG2:
            msg2 = (msg2_t)msg;
            k = msg2->val_2;
            sc_msgTx((sc_msgptr_t)&msg2, SCI_uart_pid, 0);
         break;

         default:
            /* Error handling */
         break;
      }
}
```

3.8.2 Blocking State with Time-out Example

Message Classes

msg1_t
id = MSG1
val_1: int

msg2_t
id = MSG2
val_2: int

IDLE
msg1_t msg1 *
msg2_t msg2 *
tmo = 500
i = msg1->val_1;
k = msg2->val_2;
i = 0;
msg1
msg2 SCI_uart_pid
IDLE
IDLE
IDLE

Figure 3-7: Blocking State with Time-out Example

IDLE state will block for 500 ticks until either the message object of message class sc_msg1_t or the message object of message class sc_msg2_t is received from any process. If a message object is received during the 500 ticks time-out, msg1 or msg2 contains the pointer to the received message object and the elements after the corresponding "Message Input" element will be executed.

If the time-out expires (no message object has been received during this time) the elements after the "Time-out" element will be executed.

C Code

```
/* Message definitions */

#define MSG1 (0x1)
typedef struct msg1_s{
    sc_msgid_t    id;
    int           val_1;
} *msg1_t;

#define MSG2 (0x2)
typedef struct msg2_s{
    sc_msgid_t    id;
    int           val_2;
} *msg2_t;

union sc_msg {
    sc_msgid_t    id;
};

/* Prioritized Process */

SC_PROCESS(SCP_example)
{
    static const sc_msgid_t select[3] = {
        MSG1, MSG2, 0
    };

    int i,k;

    sc_msg_t msg;
    msg1_t    msg1;
    msg2_t    msg2;

    for(;;){

        /* IDLE state */

        msg = sc_msgRx (500, select, SC_MSGRX_MSGID);
        if (msg) {
            switch( msg->id ){
                case MSG1:
                    msg1 = (msg1_t)msg;
                    i = msg1->val_1;
                    sc_msgFree((sc_msgptr_t)&msg1);
                break;

                case MSG2:
                    msg2 = (msg2_t)msg;
                    k = msg2->val_2;
                    sc_msgTx((sc_msgptr_t)&msg2, SCI_uart_pid, 0);
                break;

                default:
                    /* Error handling */
                break;
            }
        } else {
            i = 0;
        }
}
```

3.8.3 Non-Blocking State Example

Message Classes

msg1_t	msg2_t
id = MSG1 val_1: int	id = MSG2 val_2: int

IDLE

msg1_t msg1 *

msg2_t msg2 *

tmo = 0

i = msg1->val_1;

k = msg2->val_2;

i = 0;

msg1

msg2 SCI_uart_pid

Figure 3-8: Non-Blocking State Example

IDLE state will **not** block at all and program flow continues. If the message object of message class sc_msg1_t or the message object of message class sc_msg2_t is received from any process msg1 or msg2 contains the pointer to the received message object and the elements after the corresponding “Message Input” element will be executed.

If no message object has been received (is in the message queue) the elements after the “Time-out” element will be executed.

Non-blocking state must be used in interrupt and timer processes.

C Code

```
/* Message definitions */

#define MSG1 (0x1)
typedef struct msg1_s{
    sc_msgid_t    id;
    int           val_1;
} *msg1_t;

#define MSG2 (0x2)
typedef struct msg2_s{
    sc_msgid_t    id;
    int           val_2;
} *msg2_t;

union sc_msg {
    sc_msgid_t    id;
};

/* Code fragment of an interupt process */

    int i,k;

            .
            .
            .
    static const sc_msgid_t select[3] = {
        MSG1, MSG2, 0
    };

    sc_msg_t msg;
    msg1_t    msg1;
    msg2_t    msg2;

        /* IDLE non-blocking state */

    msg = sc_msgRx (SC_NO_TMO, select, SC_MSGRX_MSGID);
    if (msg) {
        switch( msg->id ){
            case MSG1:
                msg1 = (msg1_t)msg;
                i = msg1->val_1;
                sc_msgFree((sc_msgptr_t)&msg1);
            break;

            case MSG2:
                msg2 = (msg2_t)msg;
                k = msg2->val_2;
                sc_msgTx((sc_msgptr_t)&msg2, SCI_uart_pid, 0);
            break;

            default:
                /* Error handling */
            break;
        }
    } else {
        i = 0;
    }
```

Modeling SCIOPTA Systems

3.8.4 State Example Including Sender Process

Message Classes

msg1_t	msg2_t
id = MSG1 val_1: int	id = MSG2 val_2: int

IDLE

msg1_t msg1
SCP_procA_pid

msg2_t msg2
SCP_procB_pid

i = msg1->val_1;

k = msg2->val_2;

msg1

msg2
SCI_uart_pid

IDLE

IDLE

Figure 3-9: State Example Including Sender Process

There is no “Time-out” element after state IDLE present. Therefore IDLE state will block until either the message object of message class sc_msg1_t is received from process SCP_procA or the message object of message class sc_msg2_t is received from process SCP_procB. If a message object is received, the element after the corresponding “Message Input” element will be executed and msg1 or msg2 contains the pointers to the received message object.

C Code

```
/* Message definitions */

#define MSG1 (0x1)
typedef struct msg1_s{
   sc_msgid_t    id;
   int           val_1;
} *msg1_t;

#define MSG2 (0x2)
typedef struct msg2_s{
   sc_msgid_t    id;
   int           val_2;
} *msg2_t;

union sc_msg {
   sc_msgid_t    id;
};

/* pids of static processes are global */
extern sc_pid_t SCP_procA_pid;
extern sc_pid_t SCP_procB_pid;

/* Prioritized Process */

SC_PROCESS(SCP_example)
{
   static const sc_msg_rx_t select[3];
   select[0].msgid =    MSG1;
   select[0].pid =      SCP_procA_pid;
   select[1].msgid =    MSG2;
   select[1].pid =      SCP_procB_pid;
   select[2].msgid =    0;
   select[2].pid =      0;

   int i,k;

   sc_msg_t msg;
   msg1_t    msg1;
   msg2_t    msg2;

   for(;;){
      /* IDLE state, please note the flag value */
      msg = sc_msgRx (SC_ENDLESS_TMO, select, SC_MSGRX_PID|SC_MSGRX_MSGID);

      switch( msg->id ){
         case MSG1:
            msg1 = (msg1_t)msg;
            i = msg1->val_1;
            sc_msgFree((sc_msgptr_t)&msg1);
         break;

         case MSG2:
            msg2 = (msg2_t)msg;
            k = msg2->val_2;
            sc_msgTx((sc_msgptr_t)&msg2, SCI_uart_pid, 0);
         break;

         default:
            /* Error handling */
         break;
      }
}
```

Modeling SCIOPTA Systems

3.9 Message Object Output

<message_object>
<process_ID>

Figure 3-10: “Message Object Output” element

The “Message Object Output” element is used to send a message object to a process. It puts the message in the receiver’s message queue in an asynchronous way.

<message_object> is the message to send and contains the message class identifier (message ID) and the message data. The message object was allocated or received before. It is the pointer to the pointer of message object to send.

<process_name> is the process (process ID) to where the message object will be sent.

As the process ID can be just a number (of type sc_pid_t) which is defined somewhere in the code, the receiver process name is not always visible here. It is therefore recommended to include the receiver process name as a comment connected to the “Message Output” element.

3.9.1 Message Object Output Example

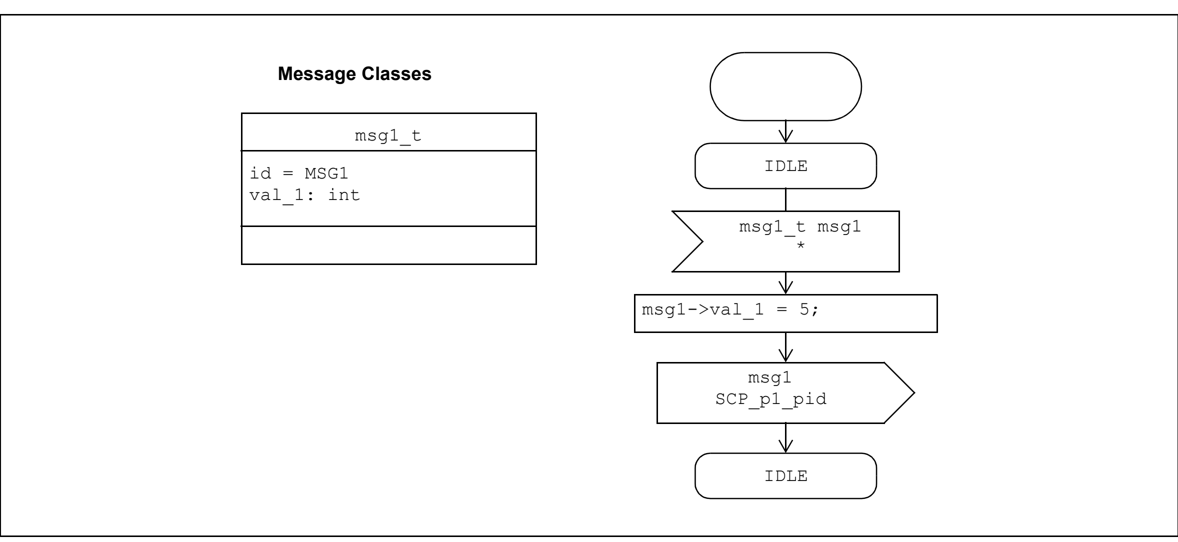


Figure 3-11: Message Object Output Example

IDLE state will block until the message object of message class sc_msg1_t is received from any process. If the message object is received, attribute val_1 will be set to 5 and the same message object sent to process SCP_p1.

C Code

```
/* Message definitions */

#define MSG1 (0x1)
typedef struct msg1_s{
    sc_msgid_t    id;
    int           val_1;
} *msg1_t;

union sc_msg {
    sc_msgid_t    id;
};

/* Prioritized Process */

SC_PROCESS(SCP_example)
{
    static const sc_msgid_t select[2] = {
        MSG1, 0
    };

    sc_msg_t msg;
    msg1_t    msg1;

    for(;;){
        /* IDLE state */
        msg = sc_msgRx (SC_ENDLESS_TMO, select, SC_MSGRX_MSGID);

        if( msg->id == MSG1 ){
            msg1 = (msg1_t)msg;
            msg1->val_1 = 5;
            sc_msgTx((sc_msgptr_t)&msg1, SCP_p1_pid, 0);
        } else {
            /* Error handling */
        }
}
```

Modeling SCIOPTA Systems

3.10 Message Object Creation

```
<message_class> <message_object>

plid:sc_poolid_t = <pool>
tmo:sc_ticks_t = <time_out>

[<attribute initialization>]
```

Figure 3-12: “Message Object Creation” element

The “Message Object Creation” element will create a message object <message_object> of class <message_class>.

<pool> defines the message pool name (pool ID) from where the message object will be allocated.

The message object creation can have a <time-out> which defines a waiting time for the message object creation from a full message pool. Please consult the **sc_msgAlloc** system call chapter of the SCIOPTA - Kernel, User’s Guides for more information about the possible time-out value.

3.10.1 Message Object Creation Example

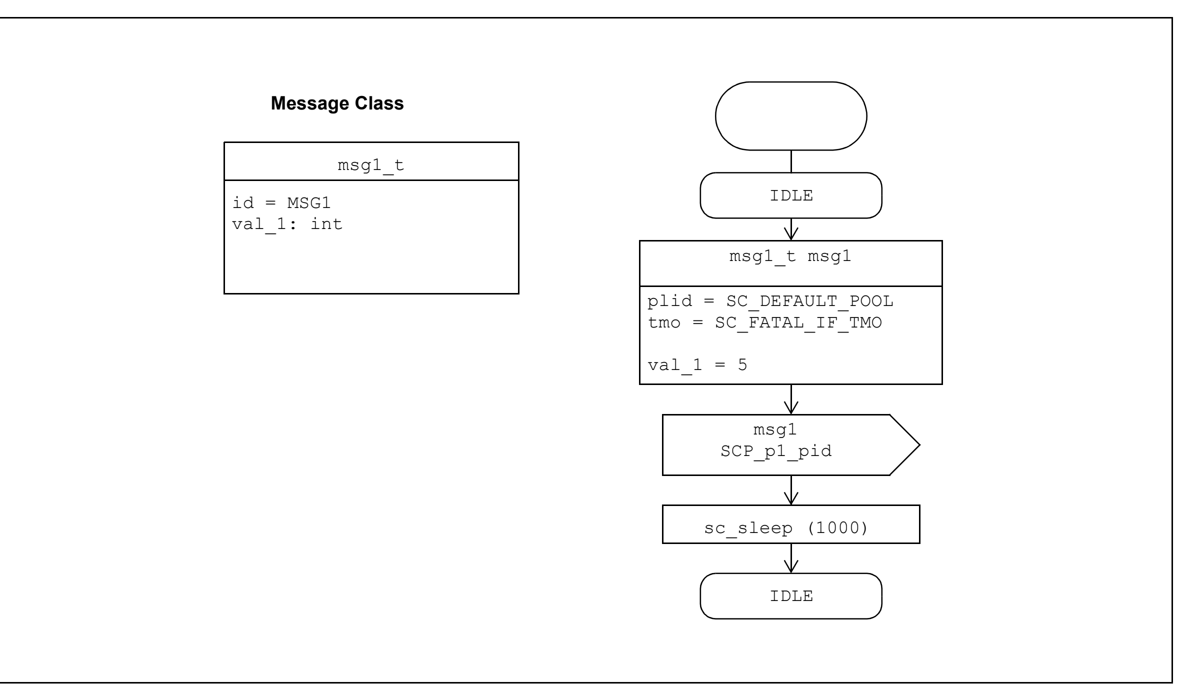


Figure 3-13: Message Object Creation Example

A message object (msg1) of message class msg1_t will be instantiated. The value of 5 will be written to attribute val_1 and the message object sent to static process SCP_p1. This loop will be repeated every 1000 ticks.

C Code

```
/* Message definitions */

#define MSG1 (0x1)
typedef struct msg1_s{
    sc_msgid_t    id;
    int           val_1;
} *msg1_t;

union sc_msg {
    sc_msgid_t    id;
};

/* Prioritized Process */

SC_PROCESS(SCP_example)
{
    msg1_t    msg1;

    for(;;){
        msg1 = (msg1_t)sc_msgAlloc(sizeof(msg1_t), MSG1, SC_DEFAULT_POOL, SC_FATAL_IF_TMO);

        msg1->val_1 = 5;

        sc_msgTx((sc_msgptr_t)&msg1, SCP_p1_pid, 0);

        sc_sleep(1000);
        }
}
```

3.11 Message Object Free

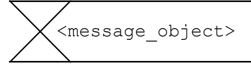


Figure 3-14: “Message Object Free” element

The “Message Object Free” element will return the memory buffer of a message object (<message_object>) to the message pool.

See chapter 3.8.2 “Blocking State with Time-out Example” on page 3-9 for a “Message Object Free” example.

3.12 Action

```
// Fill message

msg5->error = 0;
msg5->handle = NULL;
msg5->open.flags = 0;
```

Figure 3-15: “Action” element Example

An “Action” element contains a set of programming language statements.

The “Action” element should also be used for SCIOPTA system calls which do not have their own notation.

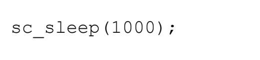


Figure 3-16: “Action” Element Example with System Call

3.13 Decision

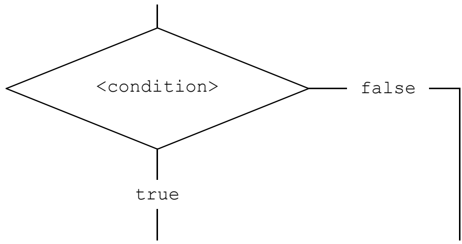


Figure 3-17: "Decision" elements

The "Decision" element is used to show if-then-else constructs.

3.13.1 "Decision" Example

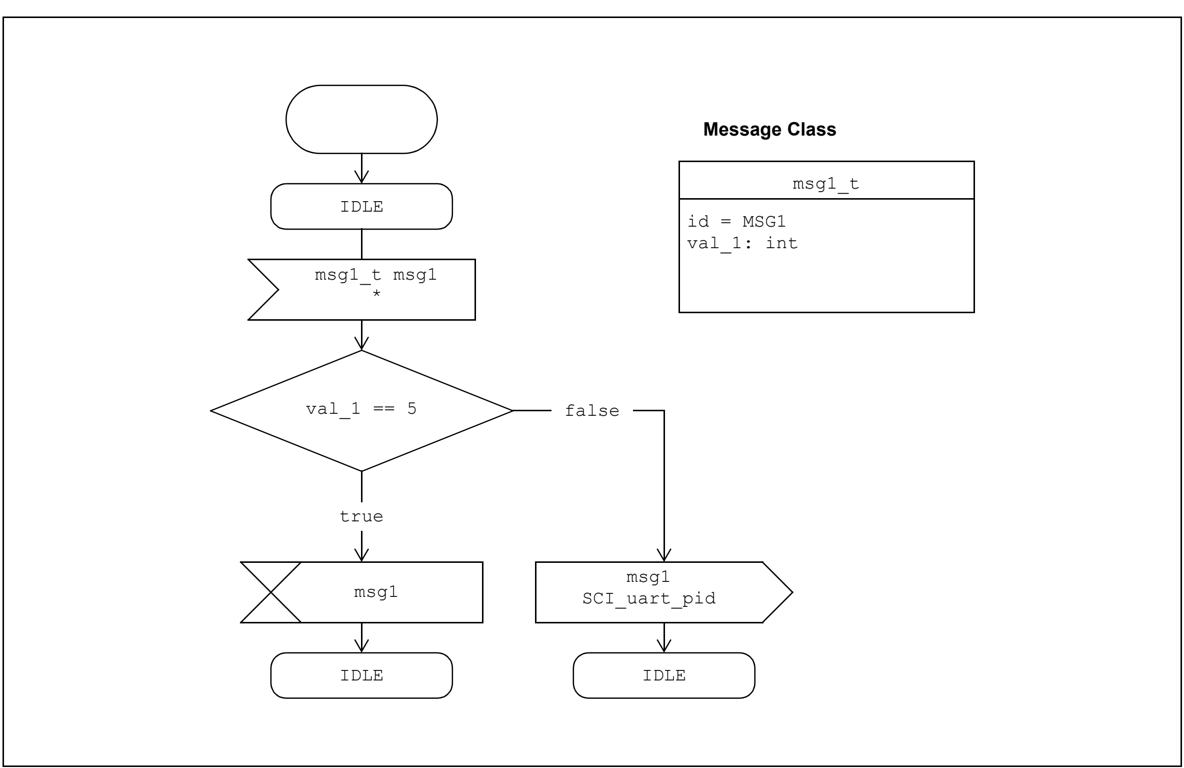


Figure 3-18: "Decision" Example

Modeling SCIOPTA Systems

C Code

```
/* Message definitions */

#define MSG1 (0x1)
typedef struct msg1_s{
    sc_msgid_t    id;
    int           val_1;
} *msg1_t;

union sc_msg {
    sc_msgid_t    id;
};

/* Prioritized Process */

SC_PROCESS(SCP_example)
{
    static const sc_msgid_t select[2] = {
        MSG1, 0
    };

    sc_msg_t msg;
    msg1_t    msg1;

    for(;;){
        /* IDLE state */
        msg = sc_msgRx (SC_ENDLESS_TMO, select, SC_MSGRX_MSGID);

        switch( msg->id ){
            case MSG1:
                msg1 = (msg1_t)msg;
                if (msg1->val_1 == 5) {
                    sc_msgFree((sc_msgptr_t)&msg1)
                }
                else {
                    sc_msgTx((sc_msgptr_t)&msg1, SCI_uart_pid, 0)
                }
            break;
            default:
                /* Error handling */
            break;
        }
    }
}
```

3.14 Switch-Case Decision

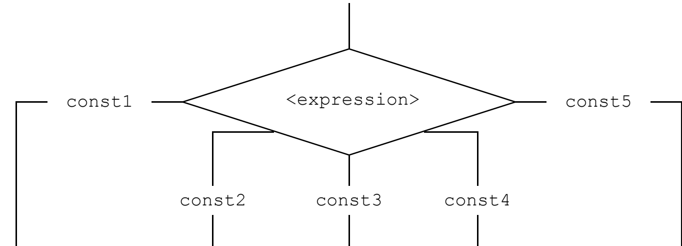


Figure 3-19: "Switch-Case Decision" elements

The "Switch-Case Decision" element is used to show switch-case constructs.

3.14.1 "Switch-Case Decision" Example

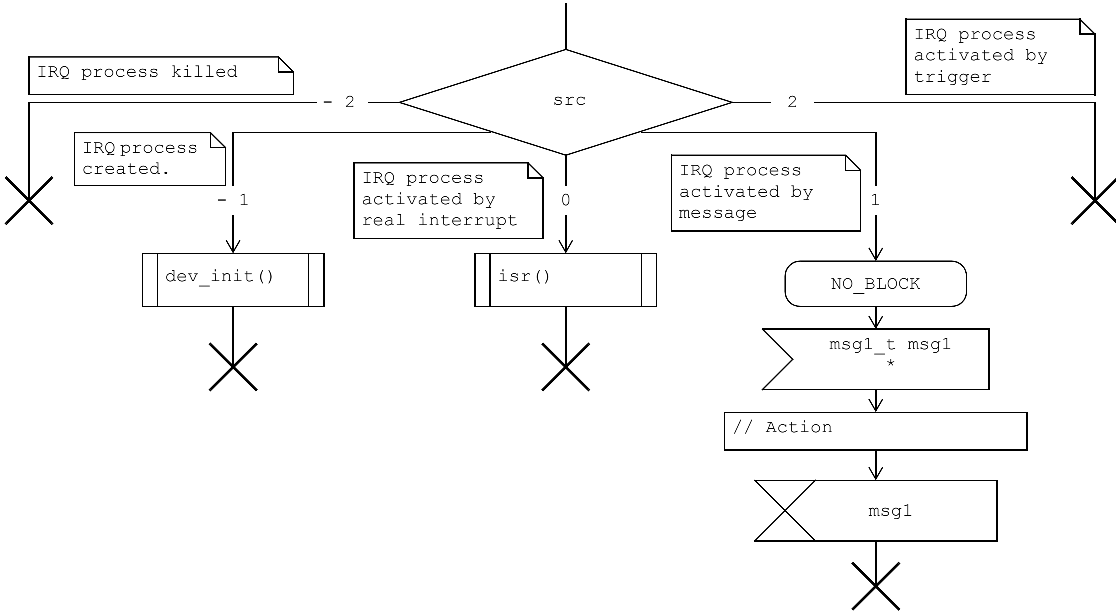


Figure 3-20: "Switch-Case Decision" Example

Figure 3-20: shows an extract of an interrupt state diagram. The decision is evaluating the interrupt process parameter "src" which defines from what source the interrupt was activated.

Please consult SCIOPTA interrupt process examples for code.

3.15 Timer Start

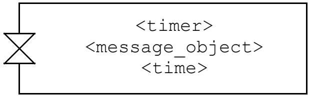


Figure 3-21: “Timer Start” element

The “Timer Start” element will request a timer message from the kernel after a defined time. It is represented by the SCIOPTA **sc_tmoAdd** system call.

<timer> is the timer name. A variable of type sc_tmoid_t, where a timer ID will be returned by the kernel after the “Timer Start” element has been executed.

<message_object> is the message object which will be sent back to the process as the time-out message. This must be an instance of a message class (see chapter 3.10 “Message Object Creation” on page 3-17)

<time> is the number of system tick after which the message will be sent back by the kernel.

3.15.1 “Timer Start” Example

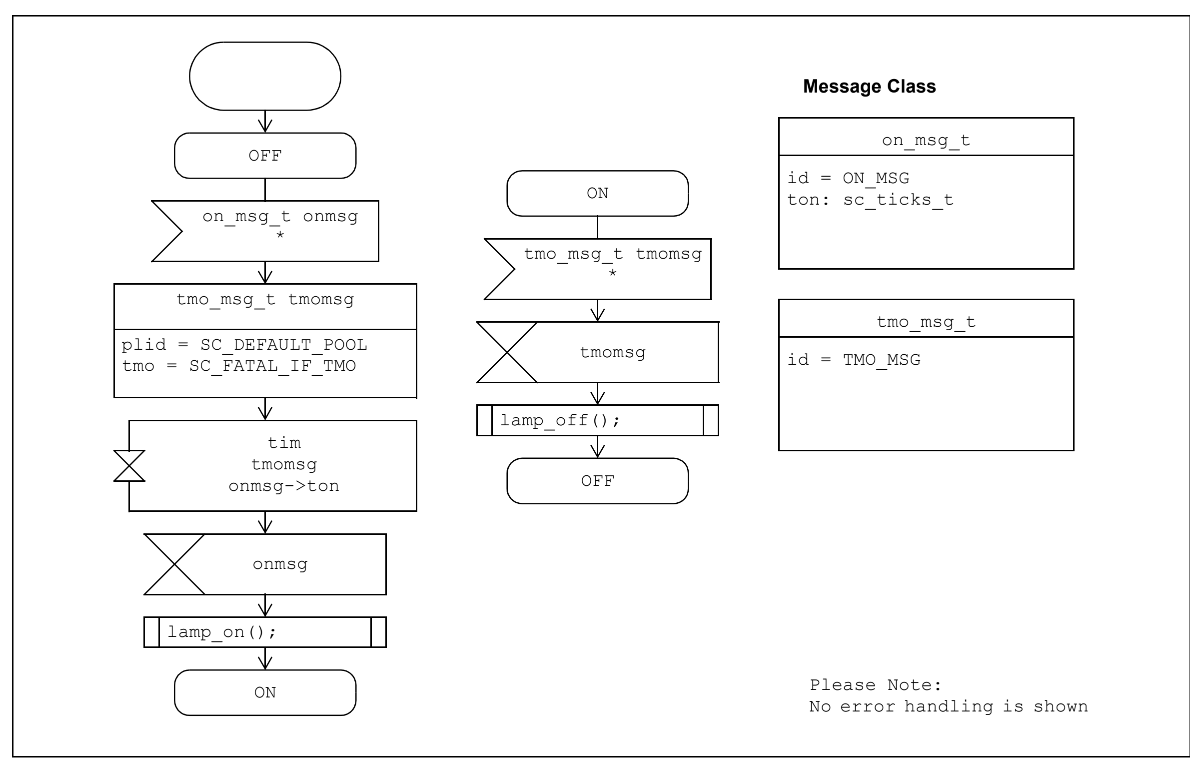


Figure 3-22: “Timer Start” Example

C Code

```
/* Message definitions */

#define ON_MSG (0x1)
typedef struct on_msg_s{
    sc_msgid_t    id;
    sc_ticks_t    ton;
} *on_msg_t;

#define TMO_MSG (0x2)
typedef struct tmo_msg_s{
    sc_msgid_t    id;
} *tmo_msg_t;

union sc_msg {
    sc_msgid_t    id;
};

/* States */
#define OFF (1)
#define ON  (2)

/* Prioritized Process */

SC_PROCESS(SCP_example)
{
    static const sc_msgid_t selon[2] = {
        ON_MSG, 0
    };
    static const sc_msgid_t seltmo[2] = {
        TMO_MSG, 0
    };

    sc_msg_t      msg;
    on_msg_t      onmsg;
    tmo_msg_t     tmomsg;

    static int state = OFF;
    static int lamp = OFF;

    for(;;){
        if (state == OFF){
            msg = sc_msgRx (SC_ENDLESS_TMO, selon, SC_MSGRX_MSGID);
            if (msg->id == ON_MSG) {
                onmsg = (on_msg_t)msg;
                tmomsg = (tmo_msg_t)sc_msgAlloc(sizeof(tmo_msg_t), TMO_MSG, SC_DEFAULT_POOL, SC_FATAL_IF_TMO);
                sc_tmoAdd (onmsg->ton, (sc_msgptr_t)&tmomsg);
                sc_msgFree((sc_msgptr_t)&onmsg);
                lamp_on();
                state = ON;
            }
        }
        else {
            /* State: ON */
            msg = sc_msgRx (SC_ENDLESS_TMO, seltmo, SC_MSGRX_MSGID);
            if (msg->id == TMO_MSG) {
                tmomsg = (on_msg_t)msg;
                sc_msgFree((sc_msgptr_t)&tmomsg);
                lamp_off();
                state = OFF;
            }
        }


    }
}
```

3.16 Timer Remove

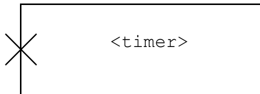


Figure 3-23: "Timer Remove" Element

The "Timer Remove" element will remove a timer before it is expired.

<timer> is the timer name. A variable of type sc_tmoid_t, where the timer ID which was returned when the timer was started, is stored. This variable contains zero after execution.

3.16.1 "Timer Remove" Example

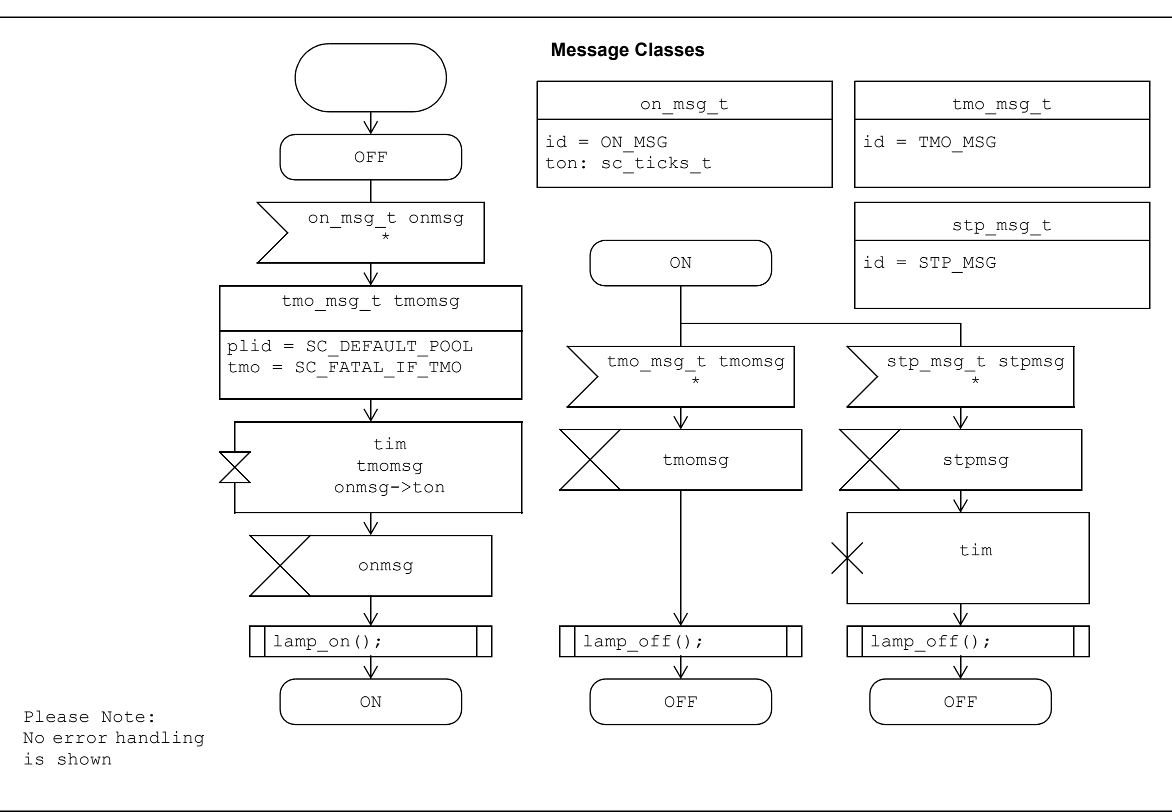


Figure 3-24: "Timer Remove" Example

3.17 Process Object Creation

```
sc_pid_t sc_procPrioCreate (
   const char      *name,
   void (*entry)   (void),
   sc_bufsize_t    stacksize,
   sc_ticks_t      slice,
   sc_prio_t       prio,
   int             state,
   sc_poolid_t     plid
);
```

Prioritized Process Creation

```
sc_pid_t sc_procIntCreate (
   const char      *name,
   void (*entry)   (void),
   sc_bufsize_t    stacksize,
   int             vector,
   sc_prio_t       prio,
   int             state,
   sc_poolid_t     plid
);
```

Interrup Process Creation

```
sc_pid_t sc_procTimCreate (
   const char      *name,
   void (*entry)   (void),
   sc_bufsize_t    stacksize,
   sc_ticks_t      period,
   sc_ticks_t      initdelay,
   int             state,
   sc_poolid_t     plid
);
```

Timer Process Creation

Figure 3-25: “Process Object Creation” Element

The process creation element will create a SCIOPTA process object. The element contains the original process creation system call which is different for prioritized, interrupt and timer processes.

Please consult the SCIOPTA - Kernel, User’s Guides for more information about processes and process creation system calls.

3.18 Operation Call

```
[<return_variable> = ]
  <operation_name>
  ({<parameters>}*);
```

Figure 3-26: "Operation Call" Element

The "Operation Call" is used to call an operation of a class. If implemented in C it corresponds to a C function and the syntax is the standard C function call syntax. If implemented in C++ it corresponds to a C++ method and the syntax is the standard C++ method call syntax.

3.19 Operation Start

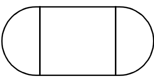


Figure 3-27: "Operation Start" Element

This element represents the start of a operation. It is the entry point into the operation. This is used if you are using SCIOPTA Extended State Diagrams to model operations. Please note that operations are always executed in the context of the process object who called the operation.

3.20 Operation Return

Figure 3-28: "Operation Return" Element

This element is used if you are using SCIOPTA Extended State Diagrams to model operations. It indicates the end of the procedure. If the procedure has a return value it should be placed by the element.

3.21 Transition Option

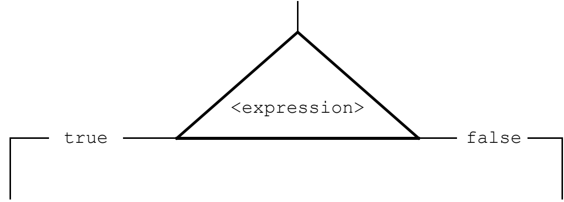


Figure 3-29: "Transition Option" Elements

The "Transition Option" element is used to show conditional flow depending on standard C #ifdef expression.

3.21.1 "Transition Option" Example

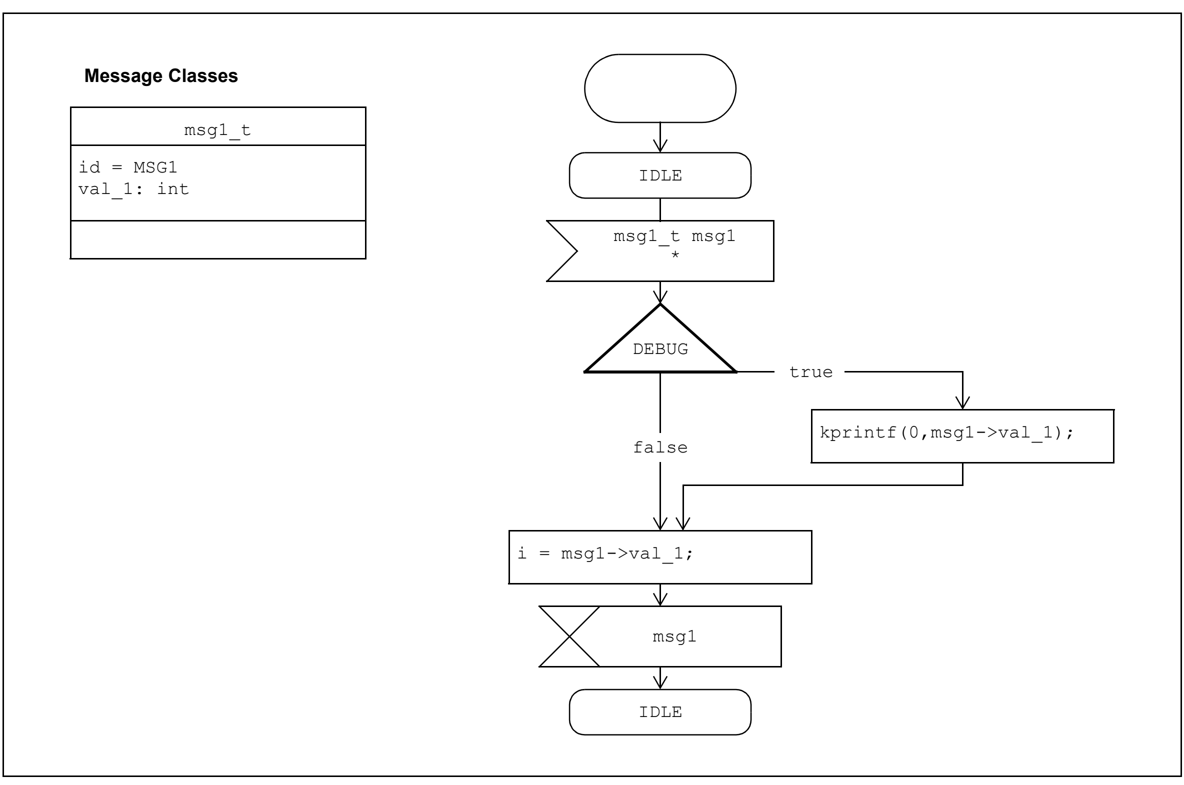


Figure 3-30: "Transition Option" Example

C Code

```
/* Message definitions */

#define MSG1 (0x1)
typedef struct msg1_s{
    sc_msgid_t    id;
    int           val_1;
} *msg1_t;

union sc_msg {
    sc_msgid_t    id;
};

/* Prioritized Process */

SC_PROCESS(SCP_example)
{
    static const sc_msgid_t select[2] = {
        MSG1, 0
    };

    int i;

    sc_msg_t  msg;
    msg1_t    msg1;

    for(;;){
        /* IDLE state */
        msg = sc_msgRx (SC_ENDLESS_TMO, select, SC_MSGRX_MSGID);

        switch( msg->id ){
            case MSG1:
                msg1 = (msg1_t)msg;
#ifdef DEBUG
                kprintf (0, msg1->val_1);
#endif
                i = msg1->val_1;
                sc_msgFree((sc_msgptr_t)&msg1);
            break;

            default:
                /* Error handling */
            break;
        }
}
```

3.22 Connectors

Figure 3-31: "Connector" Elements

Connectors are used to split a transition into several parts so that the diagram stays legible and printable. It can also be used to gather different branches to a same point.

A "Connector-Out" element has a name that relates to a "Connector-In" element. The flow of execution goes from the "Connector-Out" to the "Connector-In" element. A connector contains a name that has to be unique in the process.

3.23 Comments

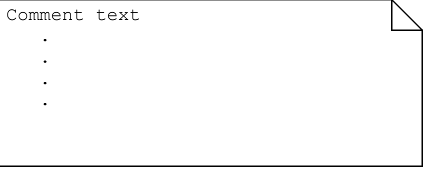


Figure 3-32: "Comment" Element Example

"Comment" elements can be placed anywhere in the document to include additional information and comments.

4 SCIOPTA Sequence Diagram

4.1 Introduction

SCIOPTA Sequence Diagrams define the asynchronous message passing between process objects and synchronous operation calls between objects. The interactions between the objects are shown in the sequential order that those interactions occur. The diagram conveys this information along the horizontal and vertical dimensions: the vertical dimension shows, top down, the time sequence of messages/calls as they occur, and the horizontal dimension shows, left to right, the object instances that the messages are sent to.

A sequence diagram describes the sequence of actions that need to perform to complete a task or scenario.

This chapter does not include information about **Interaction Frames** and **Combined Fragments**. And it is not a general introduction into UML Sequence Diagram. Please consult the specific UML 2.0 literature for more information.

4.2 SCIOPTA Scheduling

To design sequence diagrams which are executed correctly in a SCIOPTA system the designer should have a good knowledge about the SCIOPTA scheduling.

SCIOPTA uses the pre-emptive prioritized scheduling for all prioritized process types. Timer process are scheduled on a cyclic base at well defined time intervals.

The prioritized process with the highest priority is running (owning the CPU). SCIOPTA is maintaining a list of all prioritized processes which are ready. If the running process becomes not ready (i.e. waiting at a message receive which has not yet arrived) SCIOPTA will activate the next prioritized process with the highest priority. If there are more than one processes on the same priority ready SCIOPTA will activate the process which became ready in a first-in-first-out methodology.

Interrupt and timer process will always pre-empt prioritized processes. The intercepted prioritized process will be swapped in again when the interrupting system on the higher priority has terminated.

Timer processes run on the tick-level of the operating system.

The SCIOPTA kernel will do a re-scheduling at every, receive call, transmit call, process yield call, trigger wait call, sleep call and all system time-out which have elapsed.

4.3 Lifelines

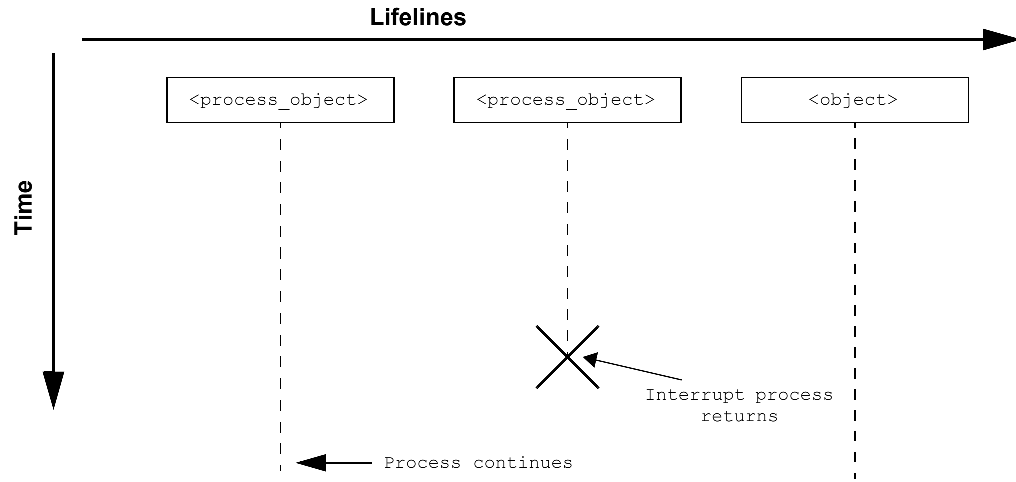


Figure 4-1: Lifelines

A lifeline represents exactly one individual object or role in an interaction. They are placed across the top of the diagram and are drawn as a box with a dashed line descending from the box. If the dashed line of a lifeline just ends with no symbol the object or role lives after the diagram. The “Process Kill” symbol at the end of the dashed line is only used in interrupt and timer processes showing the return of interrupt.

4.4 State

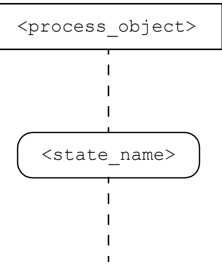


Figure 4-2: State

A lifeline usually represents a process and depending on its internal state a process might react differently to the same message object. Therefore it can be important to show a process state on its lifeline.

4.5 Activation Box

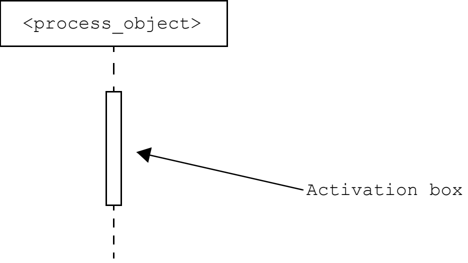


Figure 4-3: Activation Box

The dashed line in the lifeline represents the passive lifetime of a lifeline while the activation box represents its active lifetime.

4.6 Asynchronous Messages

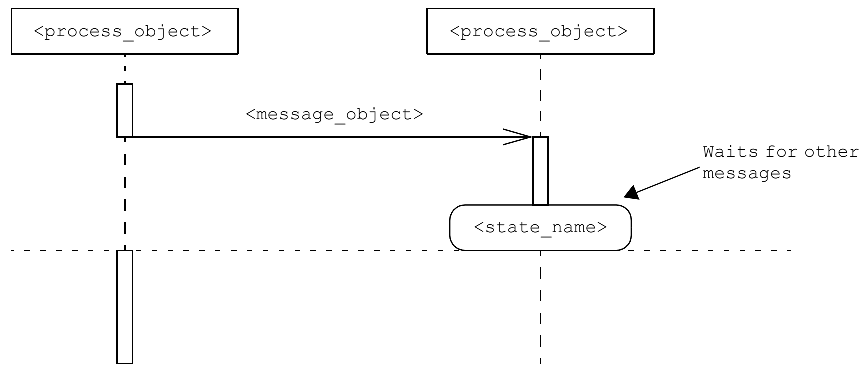


Figure 4-4: Asynchronous Messages

Asynchronous messages are showing the communication between process objects with SCIOPTA message objects. It is represented by a line from the sending object to the receiving object with a stick arrowhead.

The process activities are controlled by the scheduling rules of the SCIOPTA RTOS (see chapter 4.2 “SCIOPTA Scheduling” on page 4-1).

Figure 4-4: shows a scenario where the receiving process has a higher priority than the sending process. It runs until it will enter a state to wait for a message. After this the sender process will be swapped-in and continues. Scheduling and process swapping can be shown by horizontal dashed lines between the swapped processes.

Modeling SCIOPTA Systems

Modeling SCIOPTA Systems

4.7 Synchronous Messages

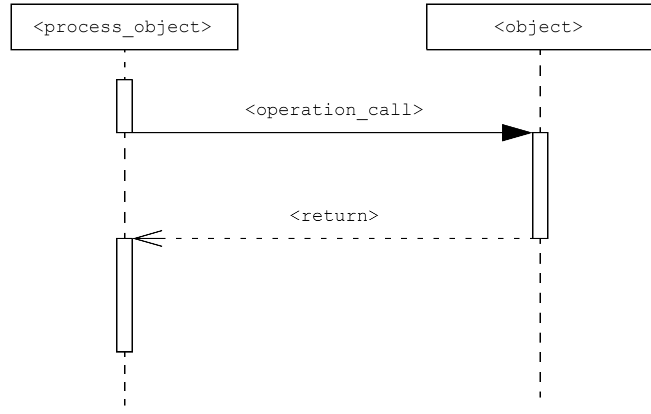


Figure 4-5: Synchronous Messages

Synchronous messages are showing calls to operation of objects. It is represented by a line from the sending object to the receiving object with a solid arrowhead. Optionally a return message can be shown which is represented by a dotted line and an open arrowhead. Above the dotted line the return value of the operation can be placed.

In a SCIOPTA system synchronous messages (operation or function calls) can only be used from a process object to a standard object. Synchronous messages between two process objects must not been used. Interprocess communication and co-ordination in a direct message passing kernel must be done by asynchronous messages.

The operation which is called from the process object runs in the context (priority and stack) of that process.

4.8 Timer Start

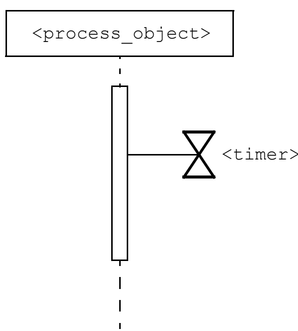


Figure 4-6: Timer Start

Starts a timer (see also chapter 3.15 “Timer Start” on page 3-23).

4.9 Timer Time-out

Figure 4-7: Timer Time-out

Shows time-out position of a timer in the lifeline.

4.10 Timer Remove

Figure 4-8: Timer Remove

Removes a timer (see also chapter 3.16 “Timer Remove” on page 3-25).

Modeling SCIOPTA Systems

5 Document Revisions

5.1 Document Version 1.0

- Initial version

5.2 Document Version 1.1

- Contact update

Modeling SCIOPTA Systems

Modeling SCIOPTA Systems

6 Index

Symbols

#ifdef expression 3-28

A

Action 3-19
Activation Box 4-3
Asynchronous messages 4-3

B

Binary association 2-2
Blocking State Example 3-7
Blocking State with Time-out 3-9

C

C function 3-27
C++ method call 3-27
Class Diagram 2-1
class sc_sys_module 2-2
Comments 3-30
Connector 3-30
Connector-In 3-30
Connector-Out 3-30
Context 2-5

D

Decision 3-20
direct message passing real-time operating system 1-1
Dynamic Process Class 2-5

E

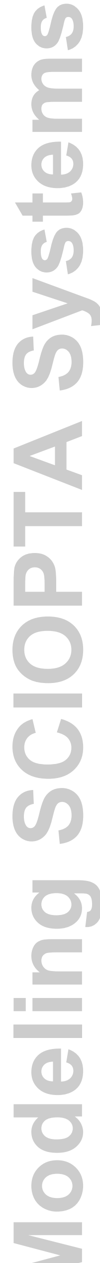
Entry point 3-4

F

Flowchart 1-1
Function Call 3-27

I

IBM's® 1-2
if-then-else constructs 3-20
Installation 2-1
Interaction 4-2
Interprocess Communication 2-6
Interrup process 3-4
Interrup Process Creation 3-26
Interrupt process 4-2
Interrupt process class 2-2
ITU-T 1-2



K

Kernel control block 2-1

L

Lifelines 4-2

M

Message Based RTOS 2-6
Message Class 2-6
Message class 3-6
Message Class Symbol 2-6
Message data 2-2
Message header 2-6
Message ID 2-2
Message identity (ID) 2-6
Message Input 3-6
Message object 3-6
Message Object Creation 3-17
Message Object Free 3-19
Message Object Output 3-15
Message owner 1-1
Message passing channels 2-2
Message pool 2-2
Message supper class 2-2
Modeling language 1-2
Module class 2-4
Module Class Symbol 2-4
Module control block (mcb) 2-2

N

Negated message 3-6
Non-Blocking State 3-11

O

Object diagram 2-3
OMG 1-2
Operation Call 3-27
Operation Return 3-27
Operation Start 3-27

P

Pool 2-2
Pool control block (pool cb) 2-2
PragmaDev 1-2
Prioritized process 4-1
Prioritized process class 2-2
Prioritized Process Creation 3-26
Process Class 2-4
Process Class Symbol 2-4
Process control block (pcb) 2-2

Process ID 3-6
Process Kill 3-4, 4-2
Process object 3-26
Process Object Creation 3-26
Process priority 2-4
Process return 3-4
Process Start 3-4
Process State 3-5
process yield call 4-1

R

Real Time Developer Studio 1-2
receive call 4-1
Recommendation Z.100 1-2
Return value 3-27
Rhapsody® 1-2
Role 4-2

S

sc_msgAlloc 3-17
sc_msgRx 3-5
sc_tmoAdd 3-23
SCIOPTA Application Class Diagram 2-2
SCIOPTA Application Object Diagram 2-3
SCIOPTA kernel class 2-1
SCIOPTA message 2-2
SCIOPTA Metamodel 1-1
SCIOPTA process 1-1
SCIOPTA Scheduling 4-1
SCIOPTA Sequence Diagram 4-1
SCIOPTA State Diagram 3-1
SDL 1-2
SDL finite state diagram 3-1
SDL/SDL-RT Tools 1-2
SDL-RT 1-2
Sequence Diagram 4-1
sleep call 4-1
Specification and Description Language 1-2
Standard Class 2-5
State 3-5, 4-2
State Diagram 3-1
State Example Including Sender Process 3-13
Static Processes Class 2-4
stereotype > 2-5
switch-case constructs 3-22
Switch-Case Decision 3-22
Synchronous Message 4-4
Synchronous operation calls 4-1
SysML 1-2
System Calls List 3-1
System module 2-2, 2-3, 2-4

Modeling SCIOPTA Systems

System Structure 2-4

T

Telelogic® 1-2
Time slice 2-4
Time-out 3-5
Time-out message 3-23
Timer ID 3-23, 3-25
Timer name 3-23, 3-25
Timer process 4-2
Timer process class 2-2
Timer Process Creation 3-26
Timer Remove 3-25, 4-6
Timer Start 3-23, 4-5
Timer Time-out 4-5
Transition Option 3-28
transmit call 4-1
trigger wait 4-1

U

UML Class Diagram 2-1
UML state diagram 3-1
UML Tools 1-2
UML Unified Modeling Language 1-2

V

Vector 2-4

Modeling SCIOPTA Systems