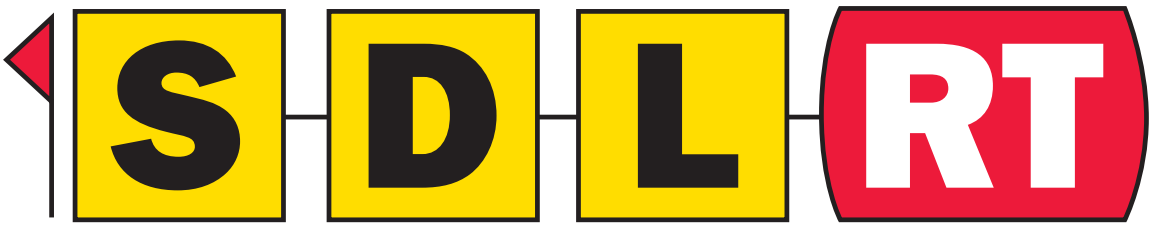
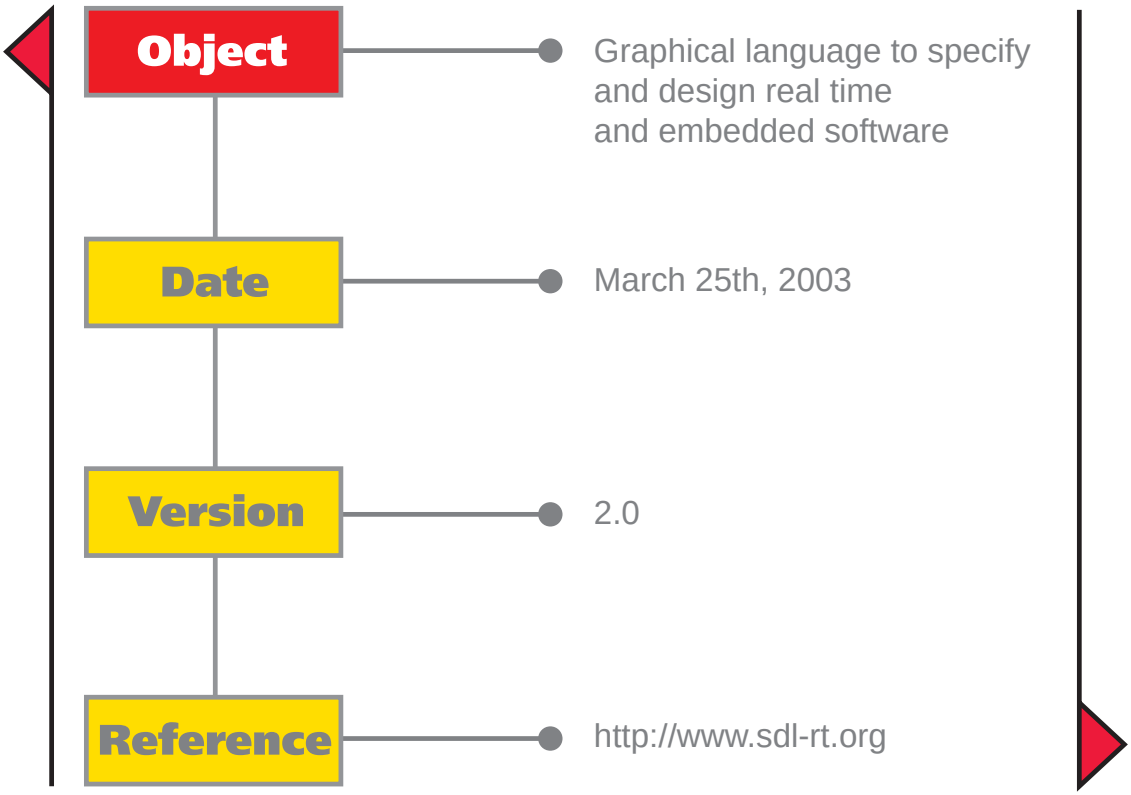


specification & description language - real time



1. 引言

顾名思义，SDL-RT 是基于 ITU 标准 SDL 对实时概念的扩展。V2.0 引入对 OMG 的 UML 的支持，以便于 SDL-RT 扩展到嵌入式软件和分布式系统的静态部分。

SDL 最初用来说明电信协议，然而，经验证明它的某些基本概念可以广泛用于实时和嵌入式系统，其主要优点是：

- 体系结构定义
- 图形化有限状态机
- 面向对象

但是，SDL 并不意味着设计实时系统，并且许多主要的缺点妨碍了在工业界的广泛使用：

- 过时的数据类型
- 陈旧的语法
- 没有指针的概念
- 没有信号灯的概念

SDL 是图形化的语言，明显不适用于所有类型的编程。应用系统的某些部分的仍然需要用大量 C 或汇编来写。然而，有效的代码或现成的库，如 RTOS、协议栈、驱动器等，都有 C 的 API。最后也是最起码的，缺乏 SDL 到机器代码的编译器，因此，SDL 必须被转换为 C 代码下载到目标机上。由此，在实际编码和与真实的硬件和软件集成时，所有 SDL 的优点都丢失了。

考虑到上面的论述，SDL 的实时扩展必须保留 SDL 的优点并且解决其缺点。越简单越好！SDL-RT 的诞生主要基于两个原则：

- SDL 的数据类型用 C 的代替，
- 在行为图中，增加信号灯支持。

SDL-RT2.0 中增加 UML 框图，扩展 SDL-RT 的应用领域：

- 需要面向对象时，UML 类图具有完美的类组织机构和关系的图形表示。动态类表示 SDL 代理（agent）和静态类表示 C++类。
- 对于分布式系统，UML 部署图提供物理体系结构的图形表示，以及节点之间的如何相互通信。

最终，SDL-RT 是：

- 更简单，
- 面向对象，
- 图形化语言，
- 结合动态和静态表示，

- 支持传统的实时概念，
- 扩展到分布式系统，
- 以标准语言为基础。

2．体系结构

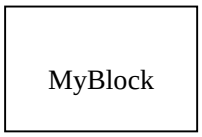
2．1 - 系统

整个设计被称为**系统**（System），系统之外的一切称为**环境**（environment）。没有特定的符号表示系统，但是需要时可以用**块**（block）表示。

2．2 - 代理

一个**代理**（**agent**）是系统结构中的一个元素。共有两种代理：**块**（block）和**进程**（process）。系统是整个块。

一个**块**（block）是一个并不暗示任何在目标系统上的物理实现的结构化元素。一个块可以进一步被一直分解下去，由此，允许处理大系统。块的符号是实线长方形，中间是其名称：



一个简单的块示例

当 SDL-RT 系统被分解成最简单的块时，用进程实现块的功能。最底层的块可以被分解成一个或数个进程。为避免块中只有一个进程，允许块和进程在同一层次上，即，在同一个块中，混合使用。

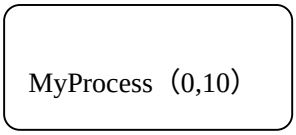
进程的符号是削掉角的长方形，中间是其名称：

一个简单的进程示例

一个**进程**（process）基本上是被执行的代码。它是基于有限状态机的任务（参考第 14 页的“**行为**（Behavior）”），并且具有一个隐含的消息队列。同一进程可以有几个独立运行的实例。在声明时，将系统启动时的实例个数和实例的最大个数放在进程名后面的括号之间。进程符号的完整语法是：

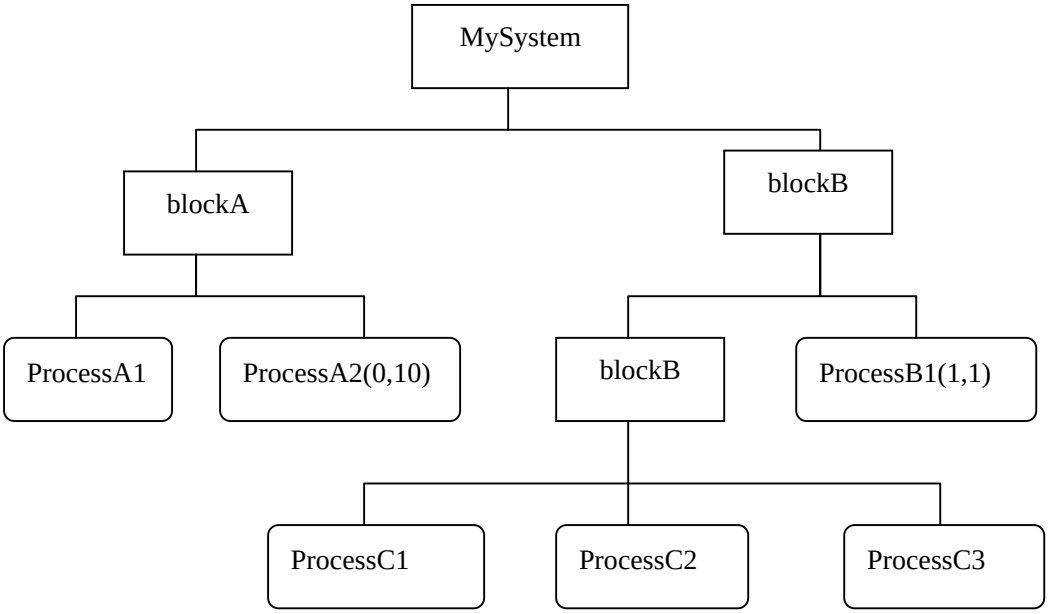
<进程名> [（<启动时的实例个数>，<实例的最大个数>）]

如果忽略，启动时的省缺值是 1，实例的最大个数是无穷大。



一个进程示例，启动时没有实例，最大的实例数是 10。

整体结构可以看作为一棵数，其叶节点是进程。



体系结构视图

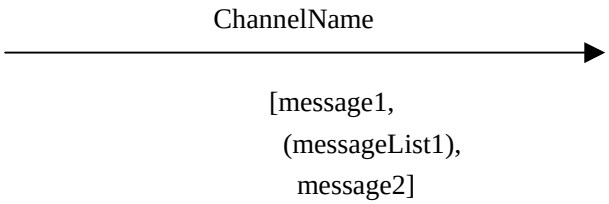
当浏览一个块时，依据系统的大小，只表示当前块层次而不带更低层代理时要方便的多。

3．通信

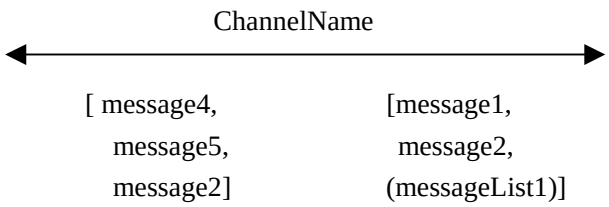
SDL-RT 是事件驱动的，这意味着通信是基于消息交换机制的。一个**消息**（message）具有名字和参数，它基本上是一个指向某些数据的指针。消息通过连接代理的**信道**（Channel），结束在隐含队列的进程中。

信道具有名称，用单向或双向箭头表示。信道名称近邻箭头，并没有特定的指示符。消息列表按指定的方式紧邻箭头，在中括号之内，并用逗号分开。消息可以收集在消息列表中，为表示消息队列中的消息列表通过信道，消息列表用括号括住。注意，同一个消息可以列在两个方向中。

一个单向信道的例子：



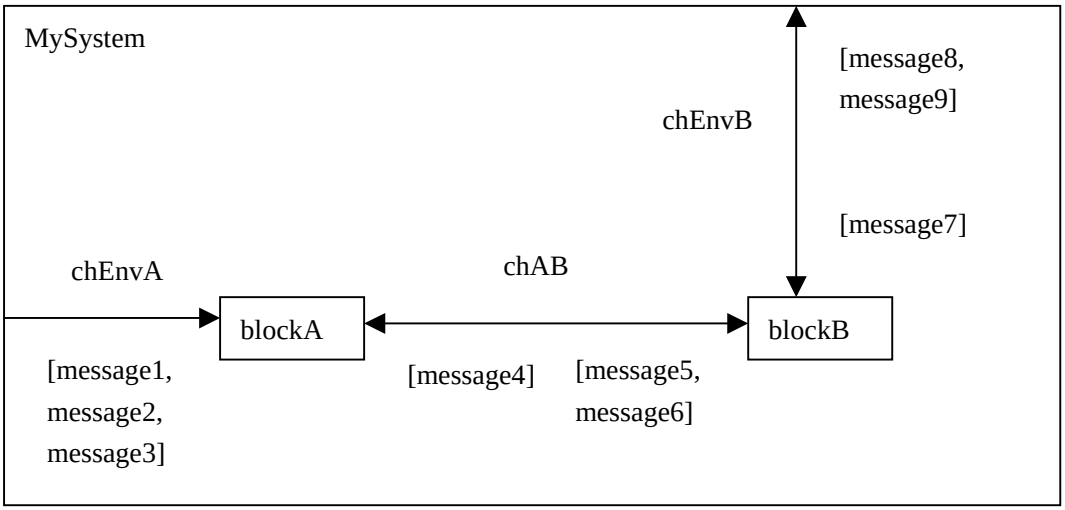
一个双向信道的例子：



信道结束点可以连接到：环境、其它信道、或一个进程。从图形上看，信道可以连接到一个块，但是，实际上它是连接到块中的其它信道。在较高的体系结构层上，为表示连接到块的外部信道，一个块视图用表示块的边界的框架围住。然后，连接到块的高层的信道表示在框架的外面，块里面的信道可以连接到这些高层的信道。注意，一个信道可以连接到几个信道。在任何情况下，保持层之间的一致性，例如，一个信道中的所有消息列在它所连接的高层或低层信道上。

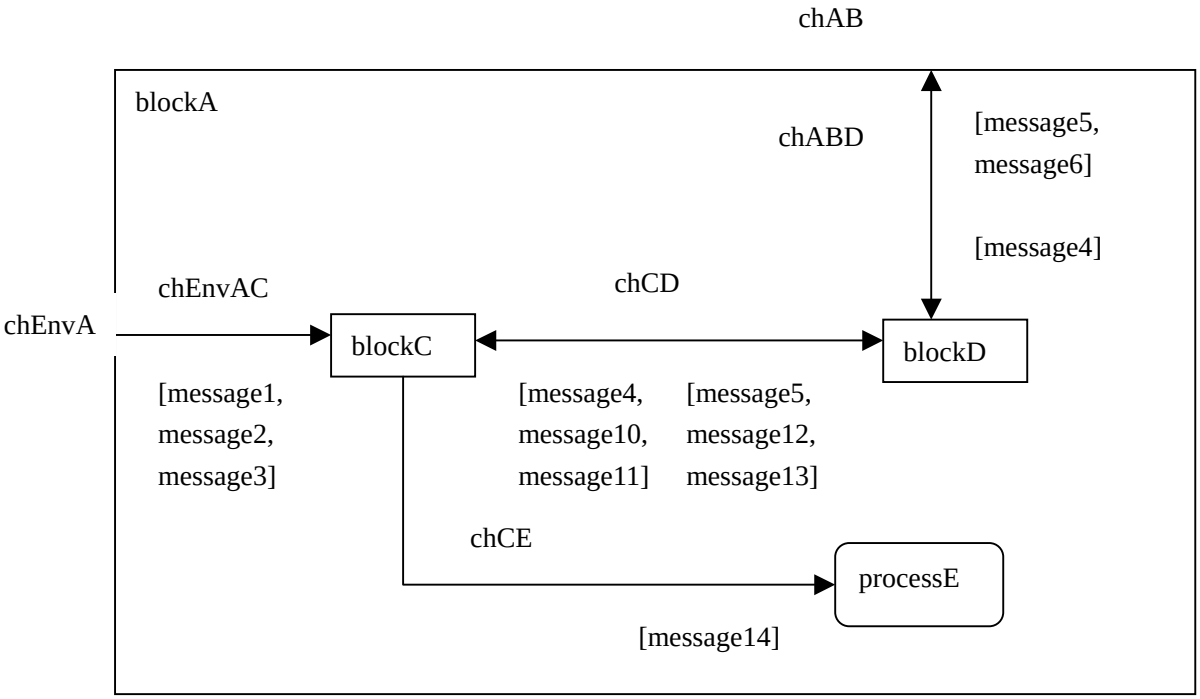
例如：

考虑一个 SDL-RT 系统，其有两个块组成：*blockA* 和 *blockB*。



一个系统视图的例子

信道 chEnvA 和 chEnvB 连接到系统 mySystem 的外围框架上。它们定义与环境的通信，例如，系统的接口。信道 chEnvA 和 chAB 连接到 blockA，并且定义进出块的消息。



一个内部块视图

块 *blockA* 的内部视图表明它有块 *blockC* 和 *blockD* 及其进程 *processE* 组成。*ChEnvAC* 连接到高层信道 *chEnvA* 和 *chABD* 连接到高层信道 *chAB*。层之间的消息流程是一致的，因为，在例

子中，从块 *blockA* 出来通过信道 *chEnvA(message1,message2,message3)*也列在 *chEnvAC* 中。

4．行为

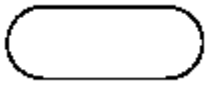
首先，一个进程具有一个隐含的消息队列接收信道中列出的消息。进程描述是基于扩展的有限状态自动机的。进程状态决定其在收到激励时应有的行为。转移是两个状态之间的代码。进程可以挂在其消息队列或一个信号灯（semaphore）上，或运行，即，执行代码。

SDL-RT进程运行是并发的；依据基本的RTOS和目标机硬件的行为而略有不同。但是，消息和信号灯在此处理进程的同步，因此，最后的行为不依赖于RTOS和硬件。由于SDL-RT对任意的C代码是开放的，设计者可以确信此陈述是千真万确的！

注意，状态图中，前一个语句总是连接到符号的更上层的框架，下一个语句连接到较低层的框架上。

4.1 – 开始

开始符号表示进程执行的开始点：

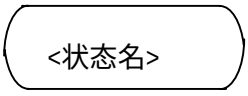


开始符号

开始符号和进程的第一个状态之间的转移称为开始转移。此转移是进程开始时做的第一件事。在此初始化阶段，进程不可能接收消息。允许连接所有其它符号。

4.2 – 状态

进程状态的名称写在进程符号中：



状态符号

状态符号是指进程要等待某些输入才能前进，允许的符号是：

- 消息输入

消息可以来源于外部信道，或是进程自身启动的计时器消息。

- 连续信号

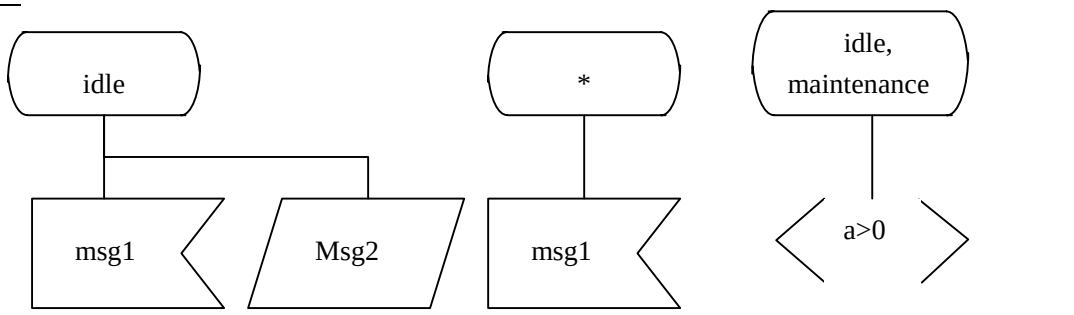
当用连续信号到达一个状态时，连续信号中的表达式按下面定义的优先权进行计算。所有连续信号表达式在消息输入之前计算！

- 存储

进来的消息不能在当前进程状态中处理。直到状态改变时才存储。进程状态改变时，存储的消息首先被处理(在队列的任何其它消息之前，但是，在连续信号之后)。

某些转移对几个状态都是合法的。不同的状态名称列成表并用逗号分开。一个星号“ * ”指所有的状态。

例如：



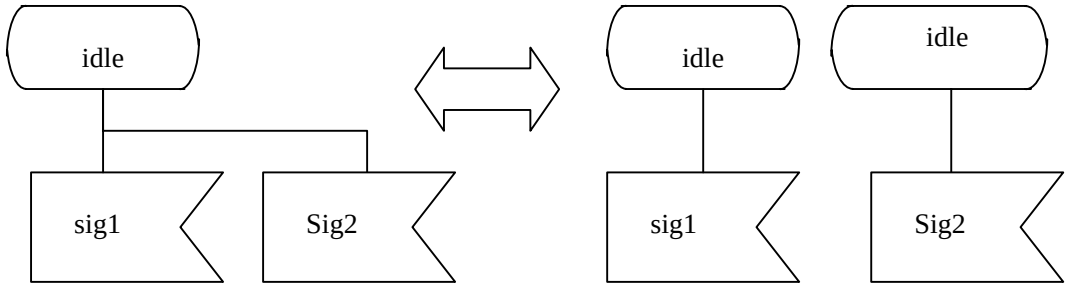
在状态 idle, msg1 可以被接收，msg2 被存储。

消息 msg1 可以在任何状态被接收。

在状态 idle 和 maintenance, 表达式 a>0 首先被计算。

特定状态的一个进程可以接收几种类型的消息或处理几种连续信号。为表示这样的情况，可以有几个消息连接到状态或将状态（名字要一样）分到几个符号中。

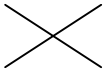
例如：



两种方法表示，在状态 idle 中，可以接收 sig1 或 sig2。

4.3 – 停止

进程可以用停止（stop）符号终止自己。

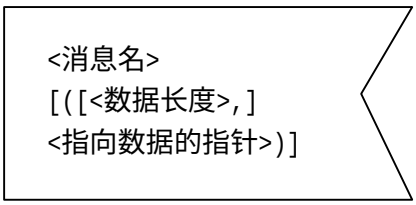


停止符号

注意，进程不能杀掉其它进程，只能杀掉自己。
此符号没有语法。

4.4 – 消息输入

消息输入符号表示SDL-RT状态中期望的消息类型。它总是跟一个SDL-RT状态符号；如果收到此符号，跟随的输入被执行。



消息输入符号

一个输入有一个名字并可以带参数。为接收此参数，必须至少声明一个变量，该变量指向参数。如果参数的长度是不知道的，因为参数是非结构化的数据，也可以通过一个预先定义的变量得到参数长度。

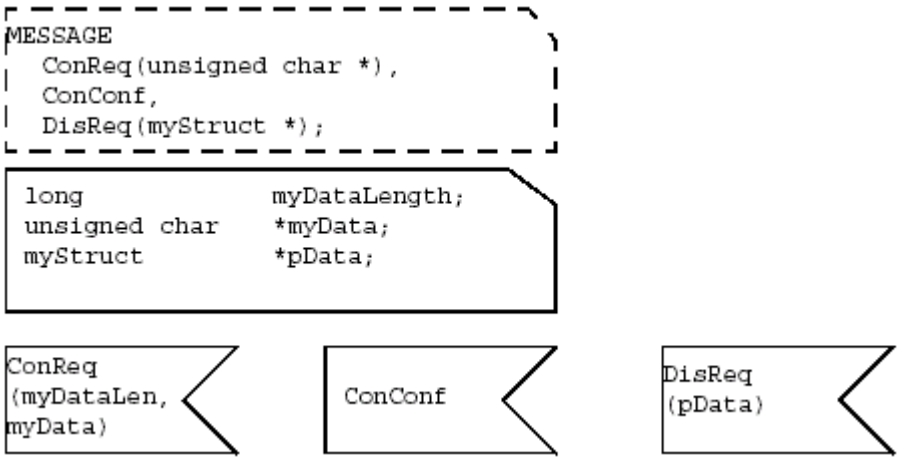
消息输入符号的语法如下：

<消息名> [([<数据长度>,] <指向数据的指针>)]

<数据长度> 是必须声明的变量。

<指向数据的指针> 是一个变量，必须声明为int。

例如：



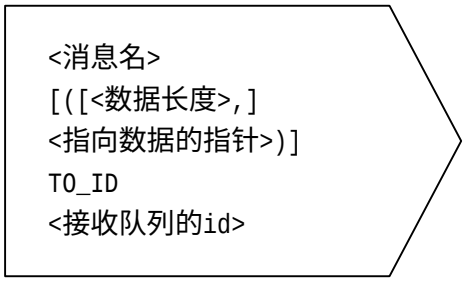
4.5 – 消息输出

消息输出用来交换信息。它按异步方式将数据放到接收者的消息队列中。

消息输出符号

当消息有参数时，给定指向参数的指针。如果参数是结构型的，不用指定长度，因为基本上是参数类型的sizeof。否则，其长度作为输出符号的第一个参数。消息输出符号的语法可以用多种形式书写，具体要看队列Id或接收者的名字是否知道。通过信道或门（gate），一个消息可以被送给一个队列的Id、或进程名。与环境通信时，需要特定的语法。

4.5.1 送给队列 Id



消息输出到队列 id

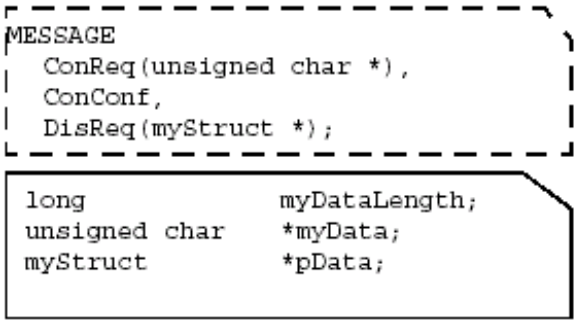
符号语法是：

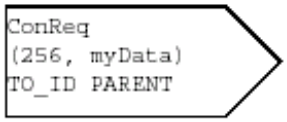
<消息名> [([<数据长度>,] <指向数据的指针>)] TO_ID <接收队列的id>

它可以得到由SDL-RT关键字给定的值。

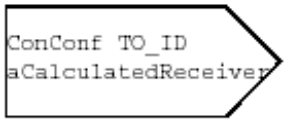
PARENT	父进程的队列id.
SELF	当前进程的队列id。
OFFSPRING	最新创立的进程的队列id，如果任意；或NULL,如果没有。
SENDER	最新接收消息的发送者的进程的队列id。

例如：

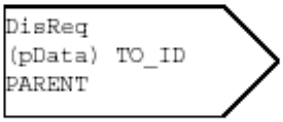




ConReq 不是结构参数，因此必须指定长度

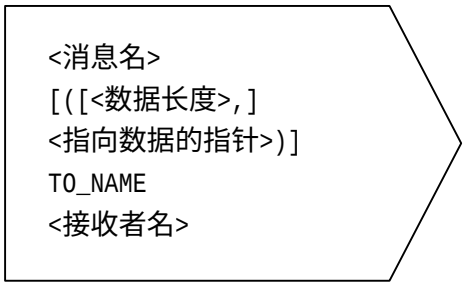


ConConf没有参数



参数长度不用指定：隐含的是 sizeof(mystruct)。

4.5.2 送给进程名



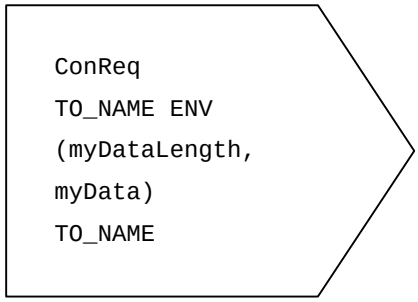
消息输出到队列进程名

语法是：

<消息名> [([<数据长度>,] <指向数据的指针>)] TO_NAME <接收者名>

<接收者名>是进程名字，如果是唯一的；仿真时，可以是 ENV，消息被送到 SDL 系统之外。

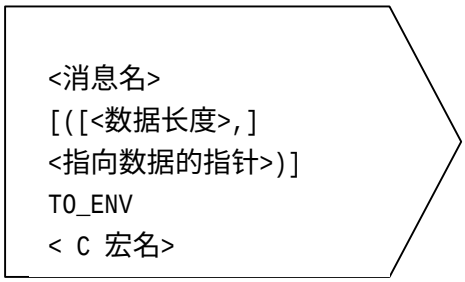
例如：



注释：

如果几个实例有同样的进程名字（例如，同一个进程的几个实例），“ TO_NAME ”将消息送到相关的名字第一个创立的进程。因此，当系统中进程名不是唯一的时候，不能用此方法。

4.5.3 送给环境

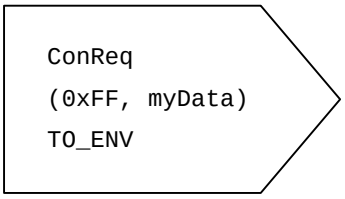


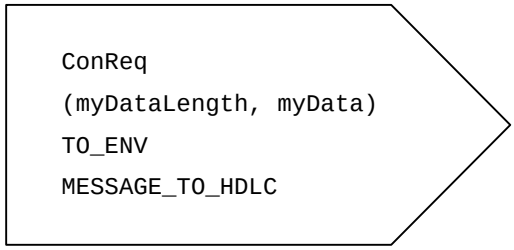
消息输出到环境

语法是：

<消息名> [([<数据长度>,] <指向数据的指针>)] TO_ENV < C 宏名>
< C 宏名>是被调用的宏的名字。如果没有声明宏，消息被送到环境。

例如：





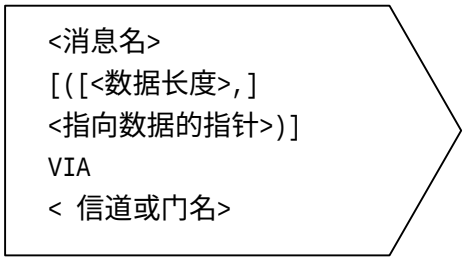
在第二个例子中，生成的代码是：

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

注释：

当传送的数据由<指向数据的指针>指出时，相应的内存应当有发送者分配，并由接收的进程释放。这是因为此内存区域没有复制给接收者，只传输指针值。因此，在发送之后，发送者不能再用。

4.5.4 通过信道或门



消息经过信道或门输出

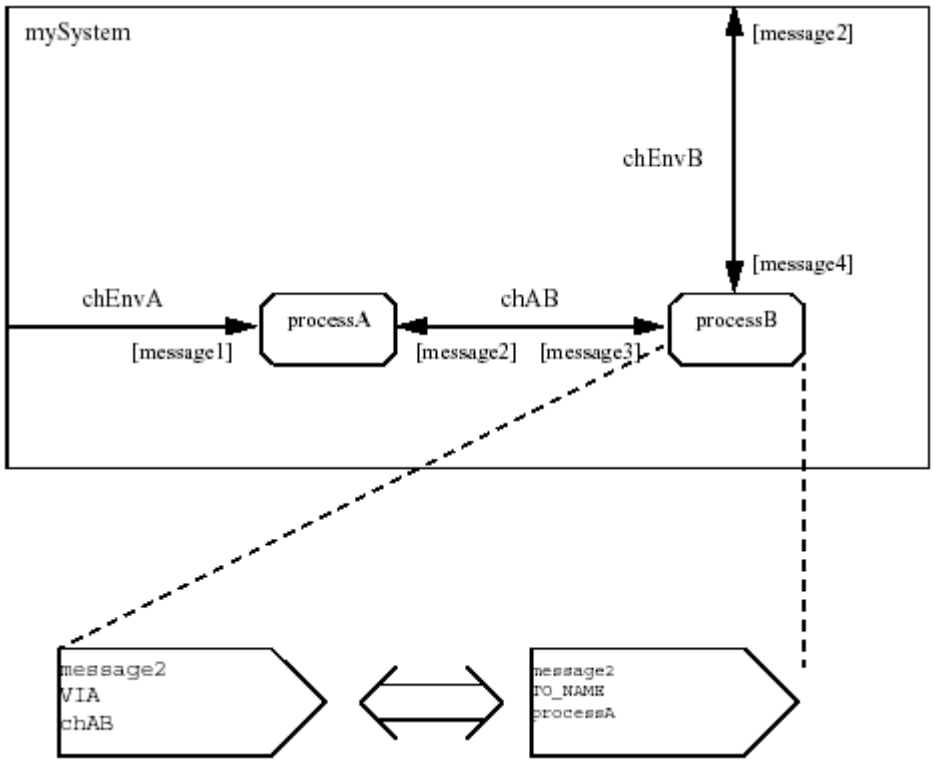
语法是：

<消息名> [([<数据长度>,] <指向数据的指针>)] VIA <信道或门名>。

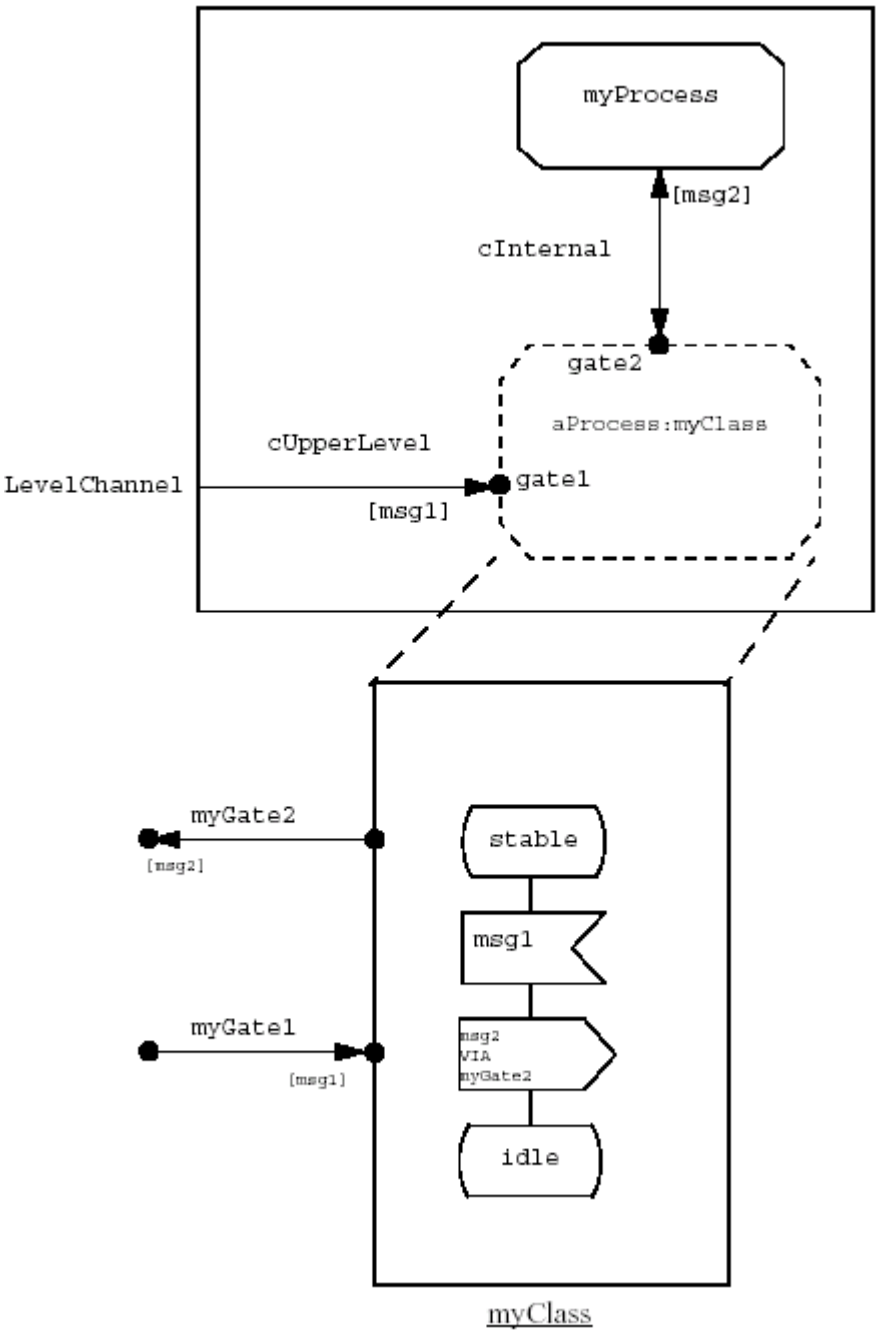
<信道或门名> 是消息将通过的信道或门的名字。

当使用面向对象时，此概念特别有用。因为不能假定类知道它们的环境；因此，消息通过门发送，该门在实例化时被连接到周围的环境。

例如：



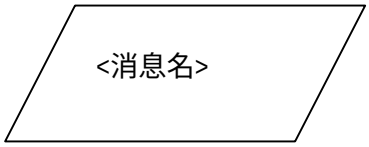
用上面定义的结构，两个输出是等价的。



aProcess 发送msg2 给myProcess，并不知道其名字，也不知道其PID.

4.6 – 消息存储

进程可以有中间状态，它不处理新的请求，直到手头的工作做完。这些新的请求不丢失而是保持，直到进程到达一个稳定的状态。存储概念就是为此而设计的，它基本上是保持消息，直到其被处理。



存储符号

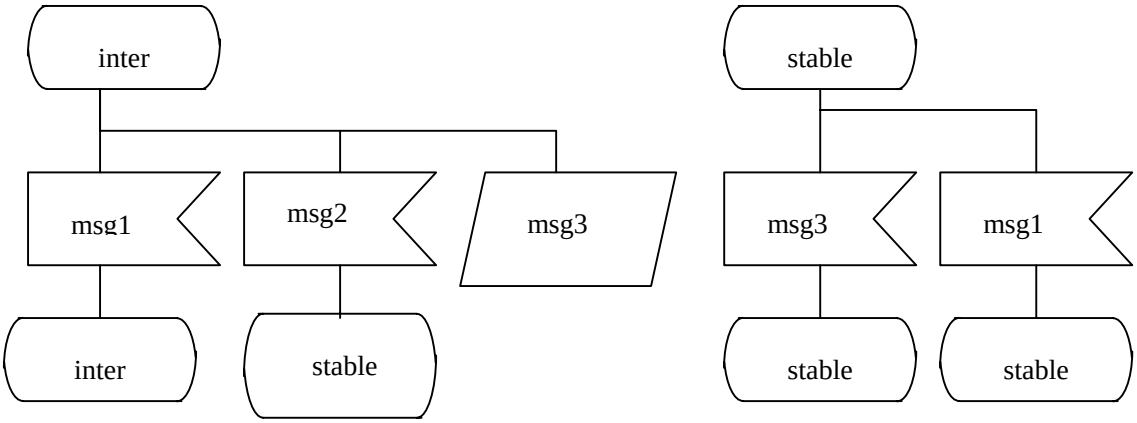
存储符号下面不跟随其它符号。当进程更改到新的状态时，已存储的消息将首先被处理（在连续信号之后，如果有的话）。

符号语法是：

<消息名字>

不管消息有没有参数。

例如，



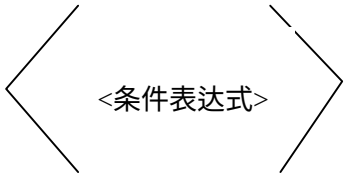
考虑上面的进程，状态 inter 接收下列消息：msg3, msg2, msg1。

msg3 被存储。msg2使进程进入状态stable。

因为，msg3 已被存储，首先被处理，最后是msg1。

4.7 – 连续信号

连续信号是一个表达式，当进程到一个新状态时，计算它。它在任何消息输入或存储消息之前计算。

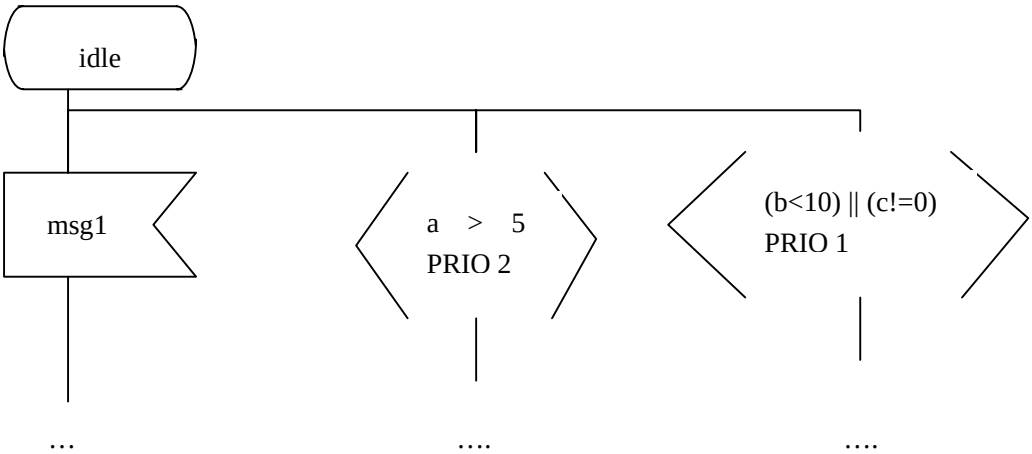


连续信号符号

连续信号表达式可以是任何标准C表达式，该表达式返回C的true/false。由于SDL状态可以包含多个连续信号，需要用关键字PRIO定义优先权等级。一个连续信号符号可以跟随任何其它的符号，连续信号或消息输入除外。语法是：

```
< C 条件表达式 >
PRIO <优先权等级>
```

例如：



上例中，当进程到达状态idle时，首先计算表达式(b<10)||(c!=0)。如果表达式不为真，或表达式仍在同一个状态，计算表达式a>5。如果表达式不为真，或表达式仍在同一个状态，执行msg1转移。

4.8 – 动作

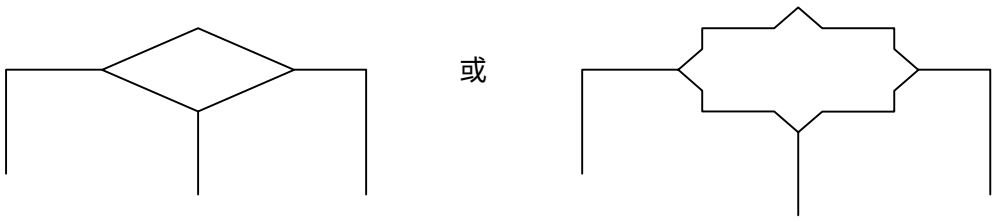
一个动作符号中包含一组C代码的指令集合。语法同C语言。

例如：

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

4.9 – 判断

判断符号可以看作为 C 的switch / case 语句。.



判断符号

因为是图形化的，在图上要占相当大的空间，建议在修改进程状态时再使用。判断符号是一个带分支的“凌形”。由于在此形状中放置文本是最糟的情况，建议用“凌形化”的长方形表示。每个分支是switch的一个case。

符号中的表达式可以是：

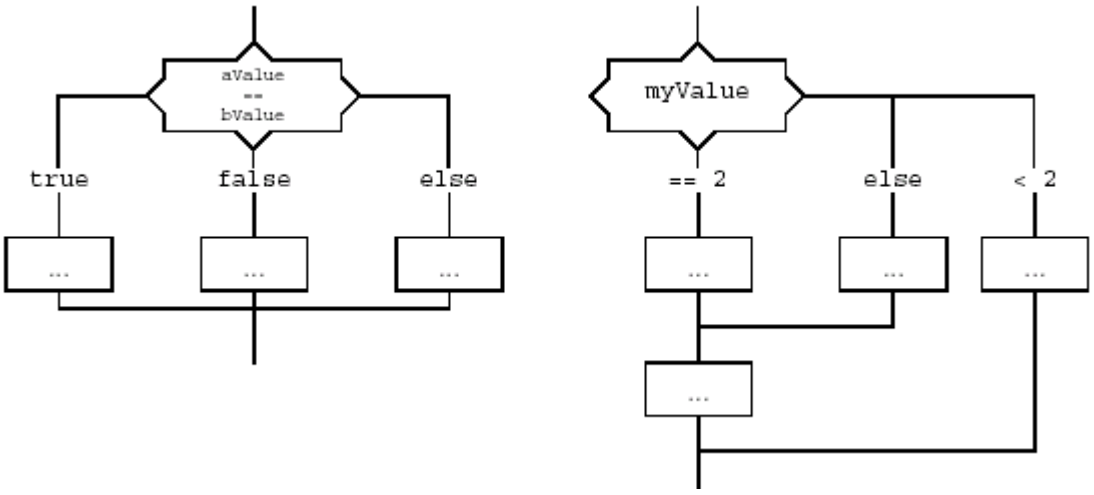
- 任何标准C表达式，该表达式返回C的true/false。
- 可以针对判断分支数进行计算的表达式。

分支的值具有的关键字表达式，例如：

- >, <, >=, <=, !=, = =
- true, false, else

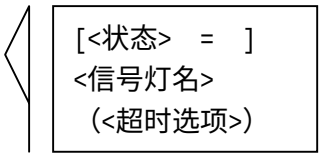
else分支包含省缺的分支，如果没有其它分支的话。

例如：



4.10 – 信号灯开启

当进程企图去开启信号灯时，使用信号灯（Semaphore）开启符号。



信号灯开启符号

为开启信号灯，“ 信号灯开启SDL-RT图形符号 ”的语法为：

[<状态> =] <信号灯名字> （<超时选项>）

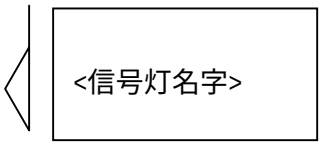
这里，<超时选项> 是：

- FOREVER
  永远悬挂信号灯，如果不使用。
- NO_WAIT
  不悬挂信号灯，如果不使用。
- <等待时钟滴答的次数>
  悬挂信号灯，按指定的滴答次数，如果不使用

<状态> 是

- OK
  如果信号灯成功的开启。
- ERROR
  如果信号灯没发现，或者，开始开启时超时。

4.11 – 信号灯取消

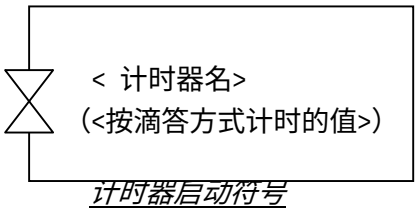


信号灯取消符号

为取消信号灯，“ 信号灯取消的SDL-RT图形符号 ”的语法为：

<信号灯名字>。

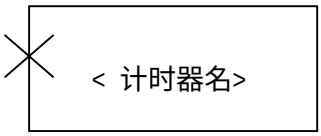
4.12 – 计时器启动



计时器启动符号

为启动计时器，“ 计时器启动的SDL-RT图形符号 ”的语法为：
< 计时器名> （<按滴答方式计时的值>） 。
<按滴答方式计时的值> 通常是整数（‘int’），但是，这依赖于RTOS和目标机。

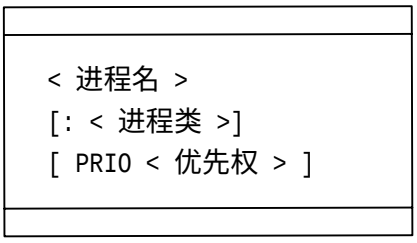
4.13 – 计时器停止



计时器停止符号

为取消计时器，“ 计时器停止的SDL-RT图形符号 ”的语法为：
< 计时器名> 。

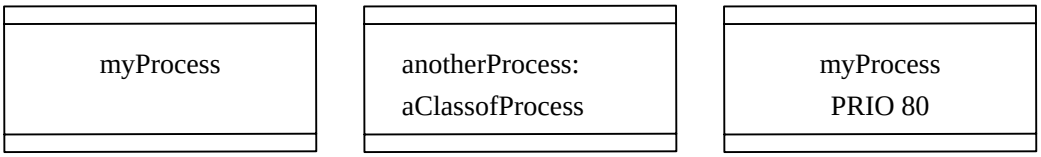
4.14 – 任务创立



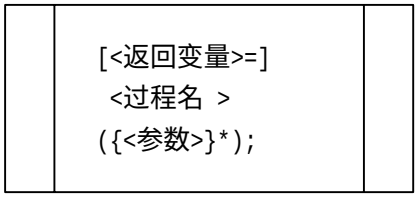
任务创立符号

为创立进程，创立进程符号的语法为：
< 进程名 > [: < 进程类 >] [PRIO < 优先权 >]
创立< 进程类 >的一个实例，其名字为<进程名>，优先权为 <优先权>。

例如：



4.15 – 过程调用

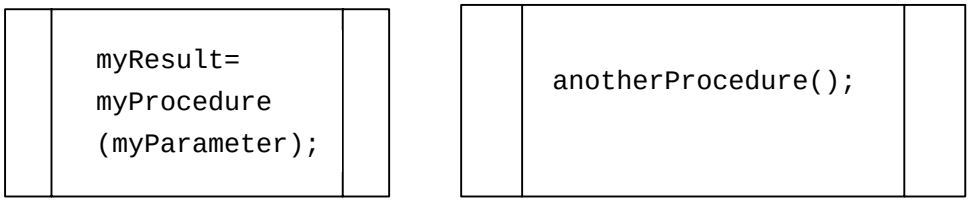


过程调用符号

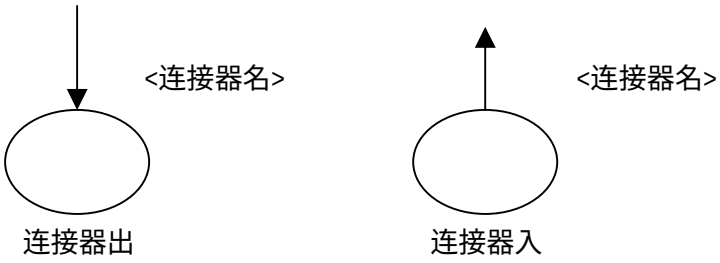
过程调用符号用来调用SDL-RT过程（参看第35页“ 过程声明 ”）。由于可能调用SDL-RT动作符号中的任何C函数，需要注意SDL-RT过程的不同，因为它们知道进程的上下文，例如，SDL-RT关键字，如，SENDER、OFFSPRING、PARENT是调用过程其中之一。过程调用的SDL图形语法是标准的C语法：

[<返回变量>=] <过程名 > ({<参数>}*)。

例如：



4.16 – 连接器



连接器用来：

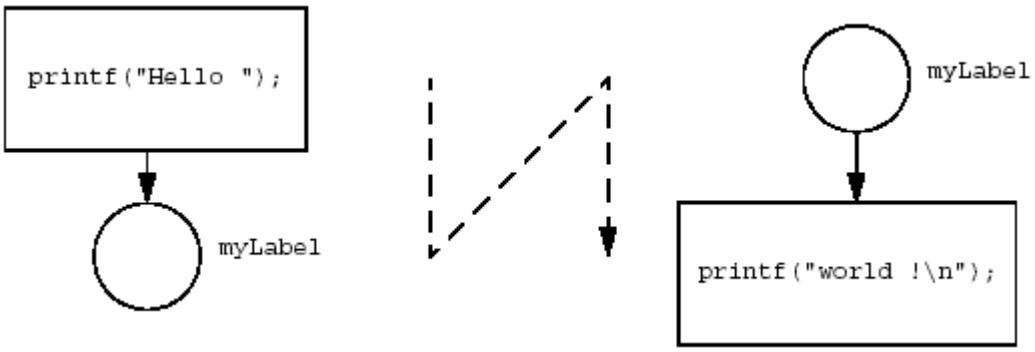
- 将转移分解成几小片，使框图易读并且好打印，
- 将不同的分支聚到同一个点。

连接器出的符号名字与连接器入的名字直接关联。执行的流程从连接器出到连接器入符号。

连接器的名字在进程中必须是唯一的。语法为：

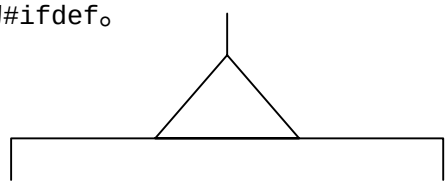
<连接器名>

例如：



4.17 – 转移选项

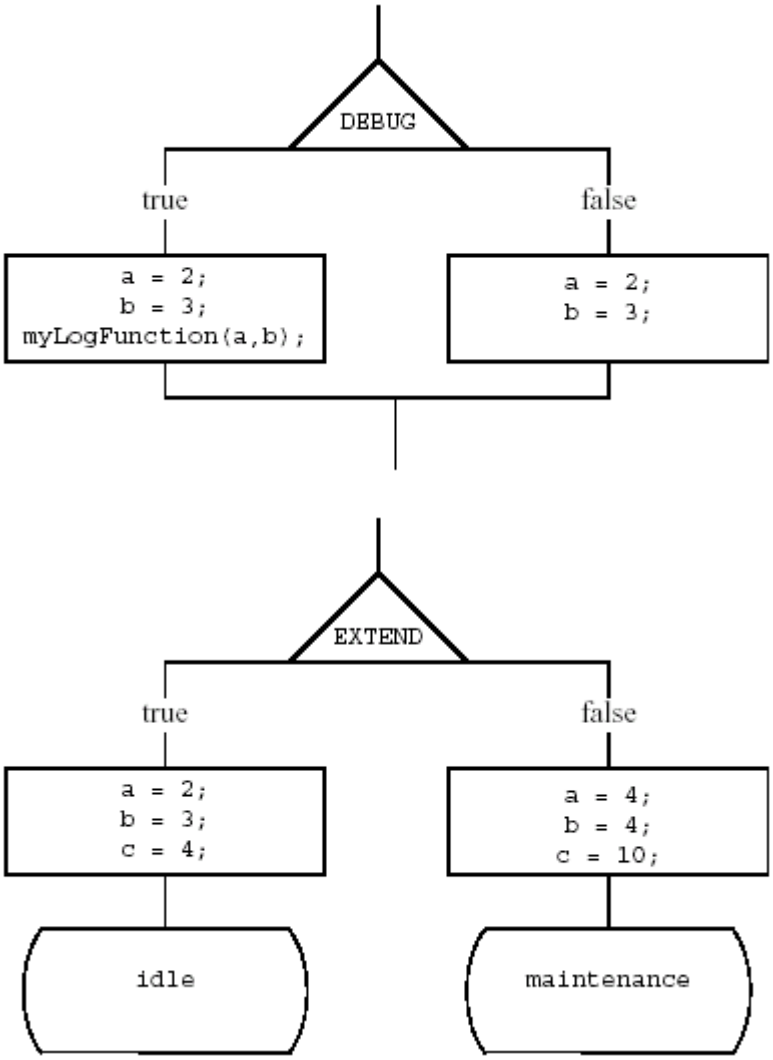
转移选项类似于C语言的#ifdef。



符号的分支具有值true 或 false。当表达式定义时，True分支被定义，等价的C代码为：#ifdef <表达式>。

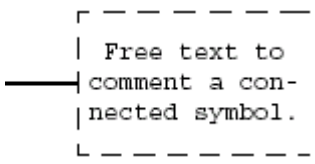
分支可以保持分割到转移的结束，或再聚到一起并且关闭选项，就象#endif一样。

例如：



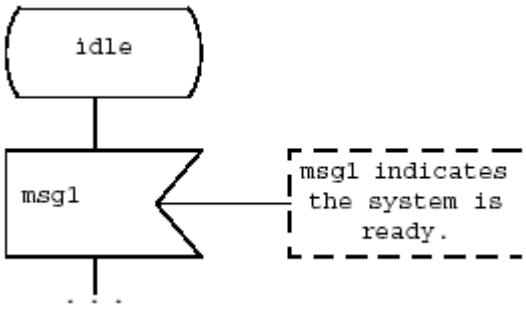
4.18 – 注释

注释符号允许写任何类型的非形式化文本，并且与要求的符号连接。如果需要，注释符号可留者不连接。



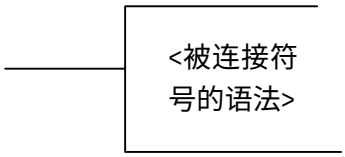
注释符号

例如：



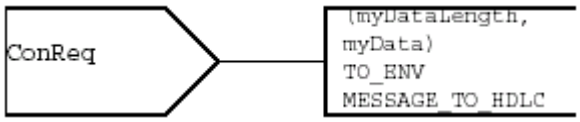
4.19- 扩展

扩展符号用来完成符号中的表达式。扩展符号中的表达式作为其所连接的符号的整个表达式的一部分。因此，其语法是连接符号之一。

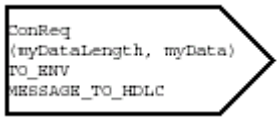


扩展符号

例如：

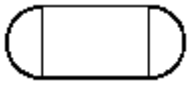


等价于:



4.20 – 过程开始

此符号是为过程框图特定的。它指明过程的入口点。



过程开始符号

此符号没有相应的语法。

4.21 – 过程返回

此符号是为过程框图特定的。它指明过程的结束。

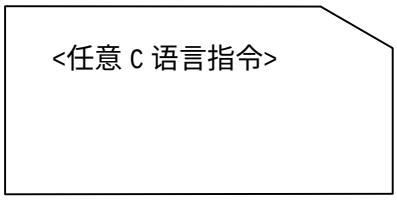
过程返回符号

[<返回值>]

如果过程有返回值，将被符号代替。

4.22 – 文本符号

此符号用来声明C类型的变量。



文本符号

语法是C语言的语法。

4.23 – 附加头部符号

此符号用来声明SDL-RT特殊的头部。

附加的头部符号

它具有特定的语法，要看在哪个图中使用。

- 块头部

  允许声明消息和消息列表:

```
MESSAGE <消息名> [(<参数类型>)] {,<消息名> [(<参数类型>)]};
MESSAGE_LIST <消息列表名> = <消息名> {,<消息名>}*;
```

- 进程类头部

  指定从其继承的超类：

```
INHERITS <超类名>;
```

- 系统，块，块类的头部

  指定使用的包。

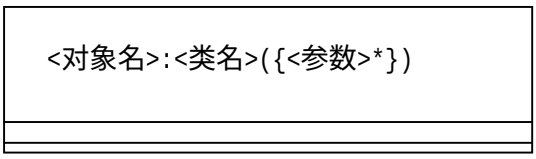
```
USE   <包名>
```

- 进程、进程类头部

  定义栈的大小：

```
STACK   <栈大小的值>
```

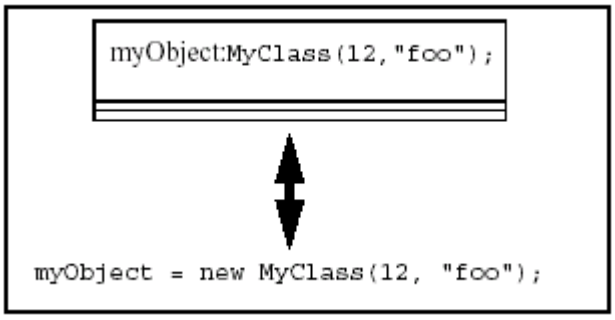
4.24 – 对象创立符号

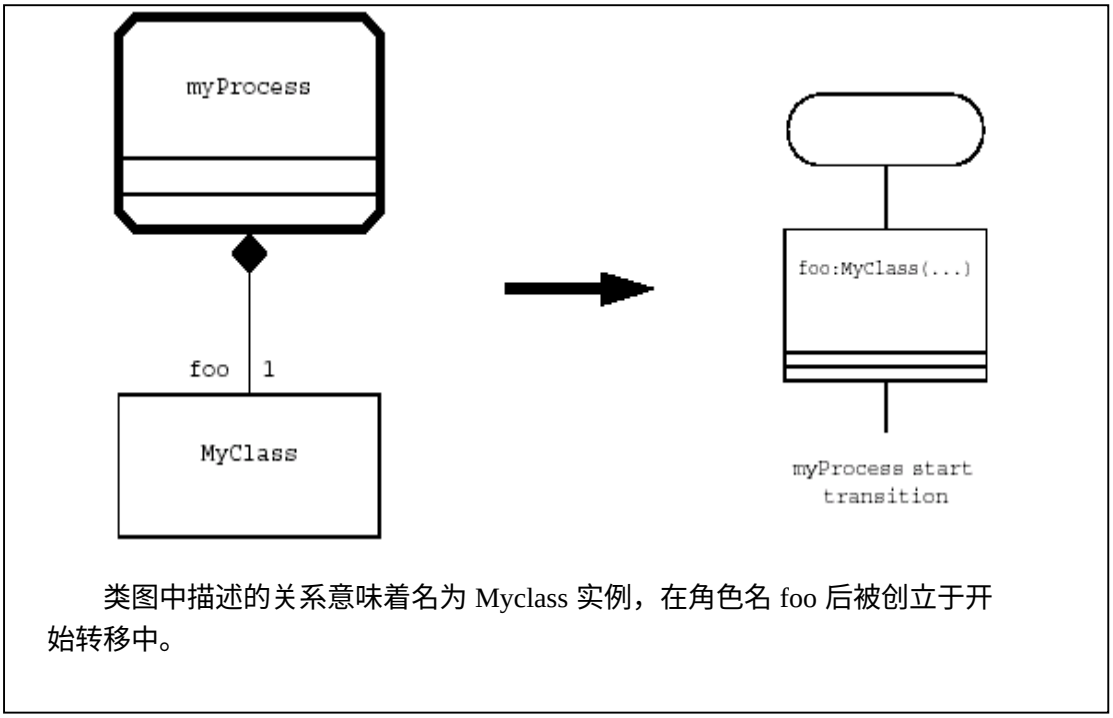


等价于建立名为<类名>类的实例，其名字为<对象名>。此符号可以被工具使用来检查动态

SDL视图和静态UML视图的一致性。

例如：





类图中描述的关系意味着名为 Myclass 实例，在角色名 foo 后被创立于开始转移中。

4.25 – 符号顺序

下表表明哪些符号可以连接到特定的符号中。

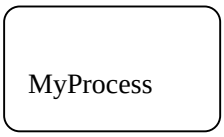
此列中的符号可以跟随此行的符号	开始	状态	停止	输入	输出	存储	连续信号	动作	判断	信号灯开启	信号灯取消	计时器启动	计时器停止	任务创立	过程调用	连接器入	连接器出	转移选项	过程开始	过程返回	对象创立
开始	-	x	x	-	x	-	-	x	x	x	x	x	-	x	x	x	x	x	-	-	x
状态	-	-	-	x	-	x	x	-	-	-	-	-	-	-	-	-	-	-	-	-	-
停止	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
输入	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
输出	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
存储	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
连续	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
动作	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
信号灯开启	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
信号灯取消	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
计时器启动	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
计时器停止	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
任务创立	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
过程调用	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	x	x	x	-	x	x
连接器出	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
连接器入	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	-	-	x	-	x	x
转移选项	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	-	x	x	-	x	x
过程开始	-	x	x	-	x	-	-	x	x	x	x	x	x	x	x	-	x	x	-	x	x
过程返回	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

上面的表要按行读。列可以跟随该行上的符号。例如，停止符号不能跟任何符号。状态符号可以跟输入、存储、或连续信号符号。

5. 声明

5.1 – 进程

进程隐含的在系统的体系结构中声明（参见，第9页的“ 体系结构 ”），因为必须建立通信信道。



进程符号

一个进程具有启动时的初始实例个数和允许的最大实例个数。一个进程也可以是进程类的一个实例（参见第37页“ 面向对象 ”），此时，类名后面跟一个冒号和实例名。

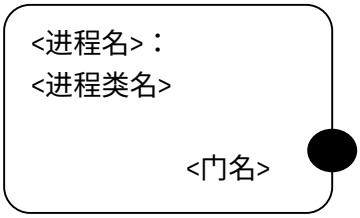
通用的语法是：

<进程实例名> [:<进程类>][(<初始的实例个数>, <最大实例个数>)] [PRIO <优先权>]

优先权是目标RTOS中的参数之一。

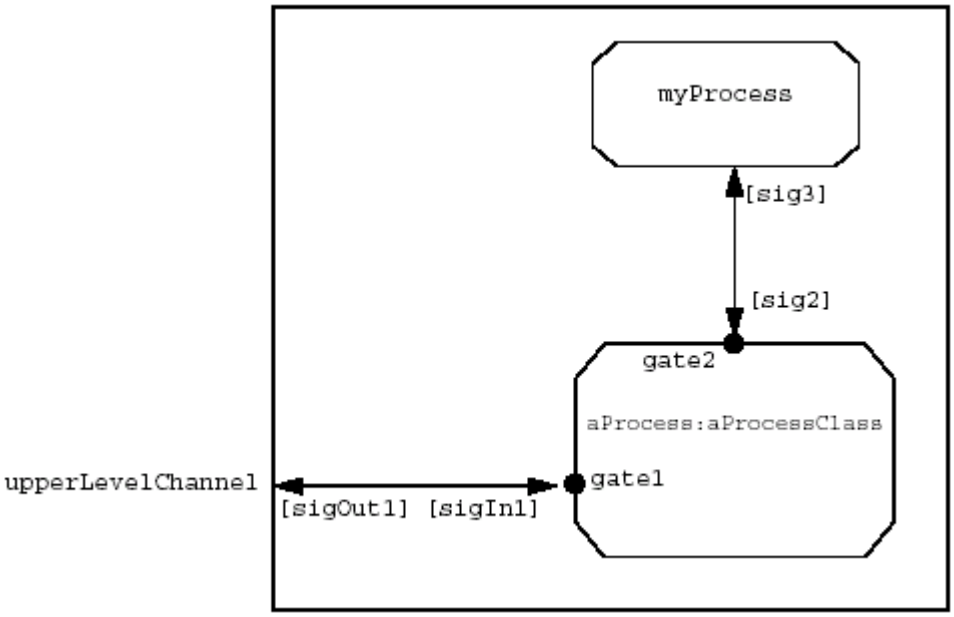
请注意，栈大小可以在进程或进程类附加头部符号中定义，如同第31页中“ 附加头部符号 ”中描述的一样。

当进程是进程类的实例时，在体系结构的框图中，进程类的门必须连接起来。门的名字出现在进程符号中，带一个黑圈表示连接点。



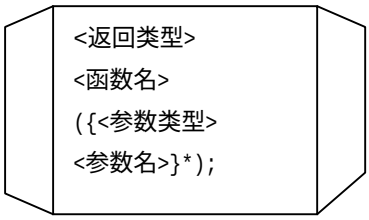
进程类实例

包中定义的、经过门的消息必须与定义进程实例的体系结构图中所列的消息一致。



5.2 – 过程声明

SDL-RT过程可以在任意的框图：系统、块、或进程中定义。它通常不能与体系结构连接，但是，由于它可以输出消息，可以用一个信道连接用于信息交换。



声明的语法与C函数的相同。过程定义可以用SDL-RT图形化表示，也可以用标准的C文本文件表示。

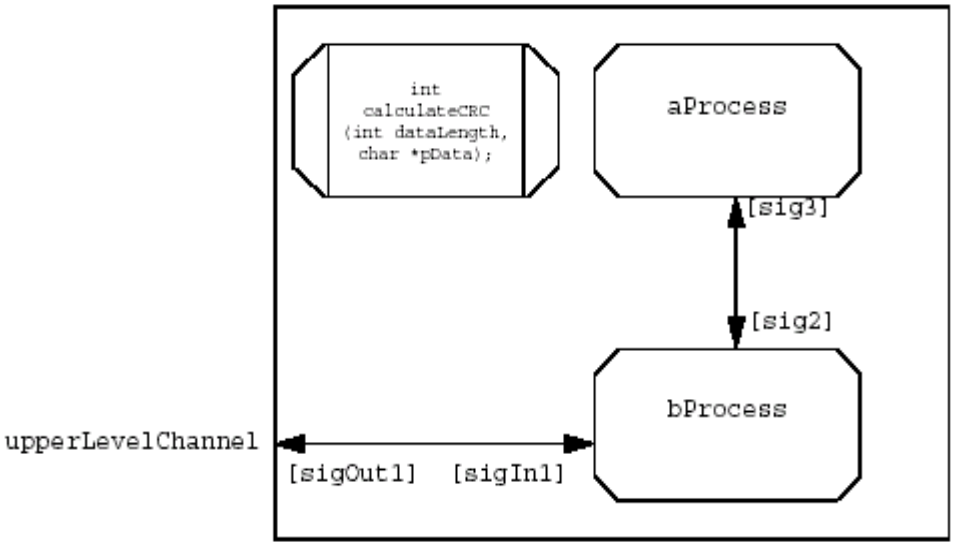
5.2.1 用 SDL-RT 定义过程

如果用SDL-RT定义调用进程，上下文隐含在过程中。因此，如果有消息输出，消息将从调用过程的进程中输出。这就是为何消息应当定义在一个连接进程的信道中，而不是连接过程的信道。要调用这样的过程，需要使用过程调用符号。

5.2.2 用 C 定义的过程

如果在C语言中定义，进程的上下文中没必要出现。为调用这样的过程，在动作符号中要使用标准的C语句。

例如：



5.3 – 消息

消息可以在任意的体系结构层的附加头部符号中声明。一个消息的声明必须包含C的参数类型。语法如下：

```
MESSAGE <消息名> [(<参数类型>)] {,<消息名> [(<参数类型>)]};
```

也可以声明消息列表，使体系结构显得更清晰。可以在任意的体系结构层的附加符号中声明。语法是：

```
MESSAGE_LIST <消息列表名> = <消息名> {,<消息名>}*;
```

一个消息列表可以包含一个消息列表，被包含的消息列表用括号括住。

例如：

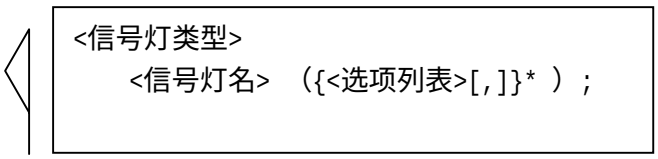
```
MESSAGE
msg1(myStruct *),
msg2(void),
msg3(void *),
msg4(int *),
msg5;
MESSAGE_LIST
myMessageList = msg1, msg2;
MESSAGE_LIST
anotherMessageList = (myMessageList), msg3;
```

5.4 – 计时器

不必声明计时器。在使用的框图中自身就声明了。

5.5 – 信号灯

信号灯可以在任意的体系结构层声明。由于每个RTOS具有自己的信号灯类型和特定的选项，因此，不必要描述详细的语法。声明符号的一般语法如下：



信号灯声明

重要的是要注意信号灯由名字标识。

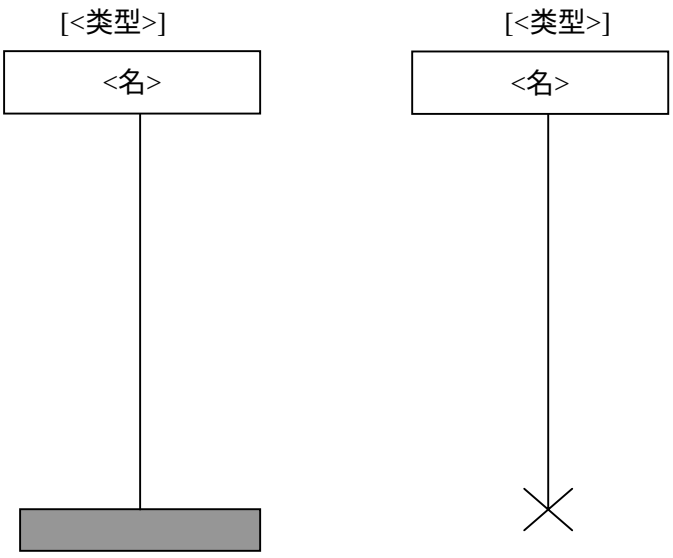
6. MSC

SDL-RT 集成了消息顺序图（Message Sequence Chart）作为动态视图，时间从上向下流失。生命线表示SDL-RT代理或信号灯和关键SDL-RT事件。放在上面的序列是先发生的。

在嵌入式C++中，可能用生命线表示一个对象。在那里，类型是对象（object），其名字应当是<对象名>:<类名>。

6.1 – 代理实例

一个代理实例开始是一个代理实例的头部，接着是实例轴，结束是实例尾或实例停止，图形如下：

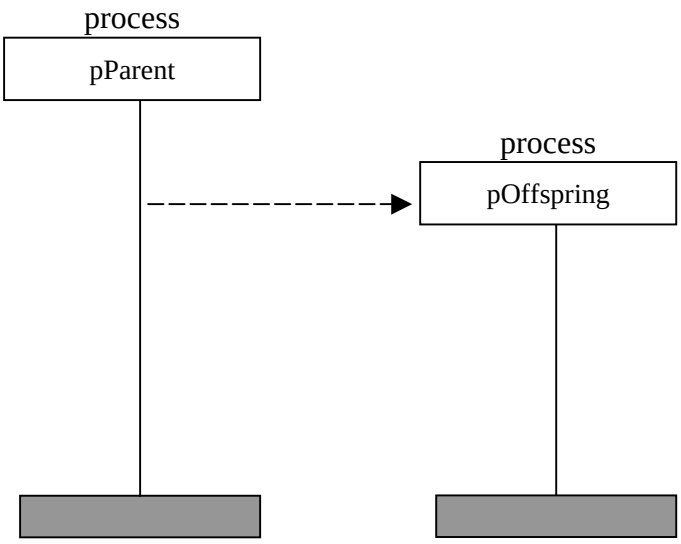


代理的类型可以在头部符号中指定，代理的名字写在实例头部符号中。实例尾符号表明在此图之后代理仍活着。

实例停止符号表明此符号之后实例不存在了。

当一个代理创立另一个代理时，虚线箭头从父代理指向子代理。

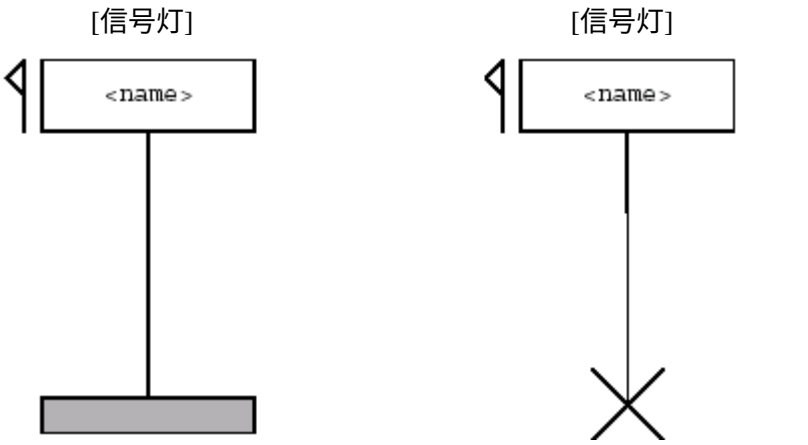
例如：



进程pParent创立子进程pOffspring.

6.2 – 信号灯表示

信号灯由信号灯头部、生命线、和信号灯结束或尾部组成。
除头部外，其它符号与进程的相同。



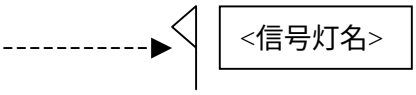
带实例尾部的信号灯　　带实例停止符号的信号灯

6.3 – 信号灯操作

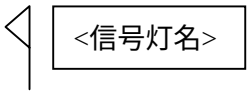
几种情况下，可以考虑用信号灯操作。一个进程企图开启信号灯，该企图可以成功或失败，

如果成功，信号灯仍然有效（计数信号灯）或无效。在信号灯无效其间，其生命线是更粗的实线，直到它被释放。

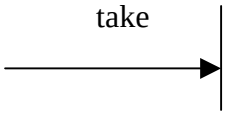
操作符号如下：



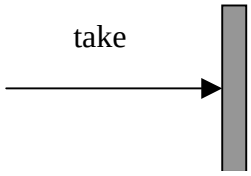
信号灯从已知的进程中创立



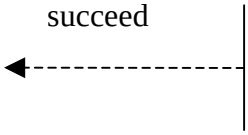
信号灯从未知的进程中创立



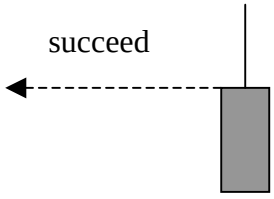
信号灯企图开启



信号灯企图开启一个琐住的信号灯

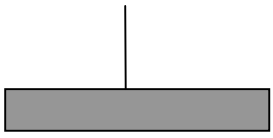


信号灯开启成功，但信号灯仍可使用

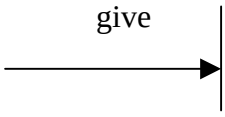


信号灯开启成功，并且信号灯不可再使用

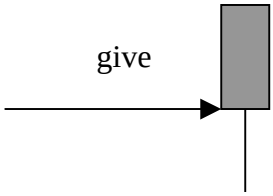
信号灯开启超时



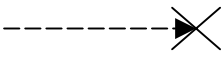
信号灯继续



信号灯取消，但在取消之前
信号灯仍可使用



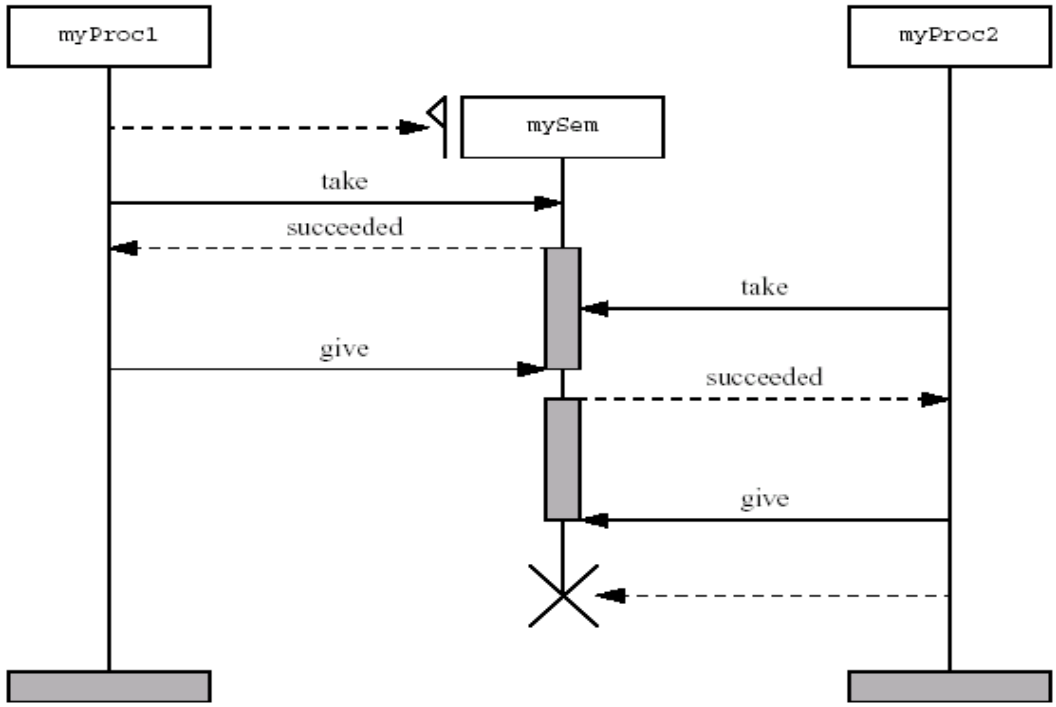
信号灯取消，但在取消之前
信号灯不能使用



信号灯被一个已知
的进程取消

信号灯被一个未知
的进程取消

例如：

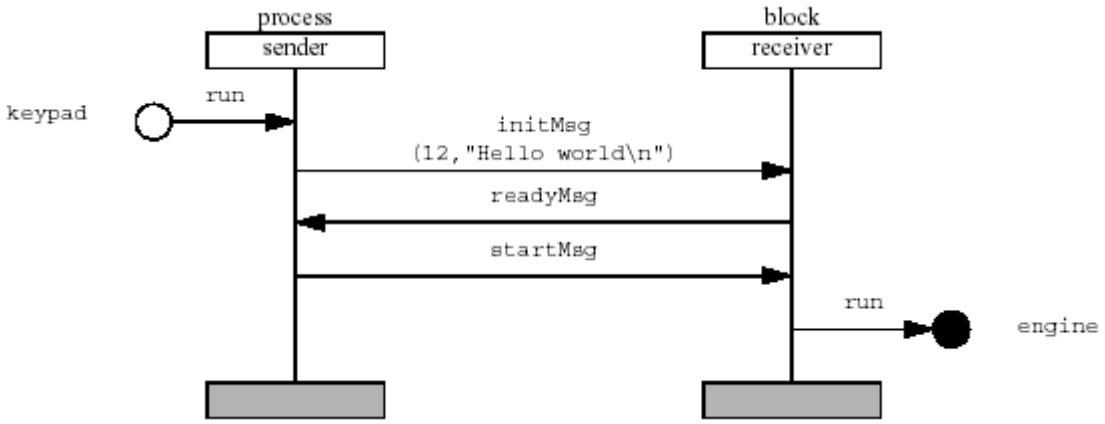


进程 myProc1 首先创立信号灯mySem，然后成功开启。进程myProc2 企图开启信号灯光mySem，但它是琐上的。进程 myProc1 释放信号灯，因此，myProc2 成功得到信号灯。进程 myProc2 给回信号灯，并且取消它。

6.4 – 消息交换

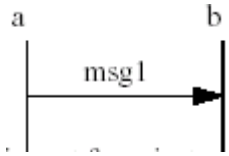
消息符号是简单的箭头，带有名字和可选择的参数。水平箭头说明到达的消息立即发送给接收者，向下的箭头表示在确定时间之后或另一事件后到达的消息。不能出现向上的消息！

当发送者和接收者表示在图上时，箭头连接到它们的实例上。如果发送者忽略它，用白圈代替；如果接收者忽略它，用黑圈代替。可以紧邻着圈写出发送者或接收者。

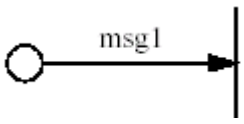


一个称作keypad的外部代理发送 run 消息给进程 sender。
进程 sender 发送initMsg，认为它被块 receiver 立即接收。块 receiver 回复 readyMsg，进程 sender 发送 startMsg，块 receiver 发送 run 到一个外部代理。.

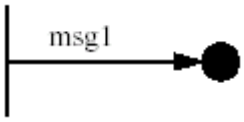
当一个消息从代理的消息队列读出时，就认为被一个代理接收，而不是到达消息队列时！



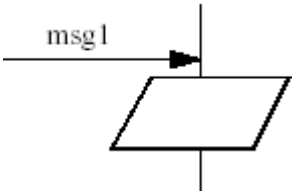
msg1 从实例 a 送到 实例 b



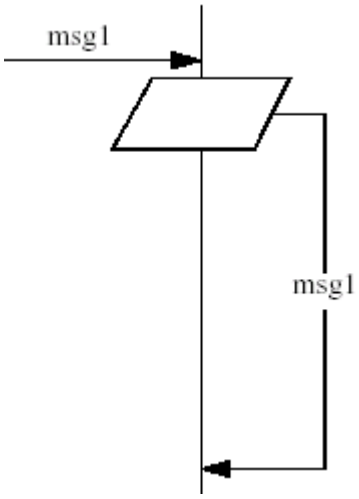
msg1 从一个不知道的发送者处来



msg1 被送到一个不知道的接收者



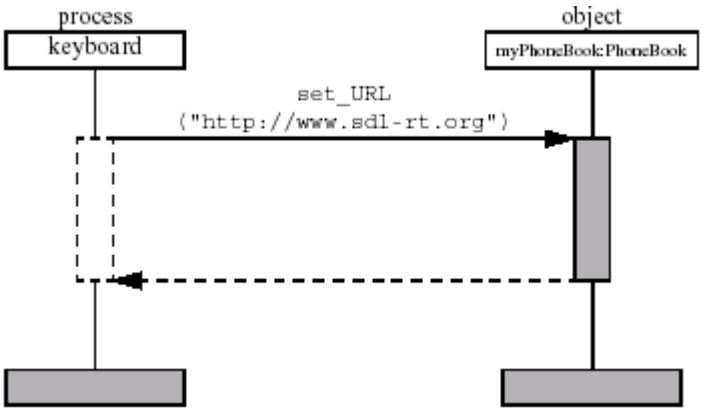
msg1 被存储并且仍在存储队列中



存储的msg1现在被消费掉

6.5 – 同步调用

此种表示用于嵌入式C++对一个对象上的方法调用。对象可以用生命线表示。同步调用用一个指向表示实例的箭头表示。对象起作用时，其生命线为黑色长方形，而代理的生命线为白色长方形。这意味着执行流程已经转移到对象。当方法返回时，虚线箭头返回到方法调用者。



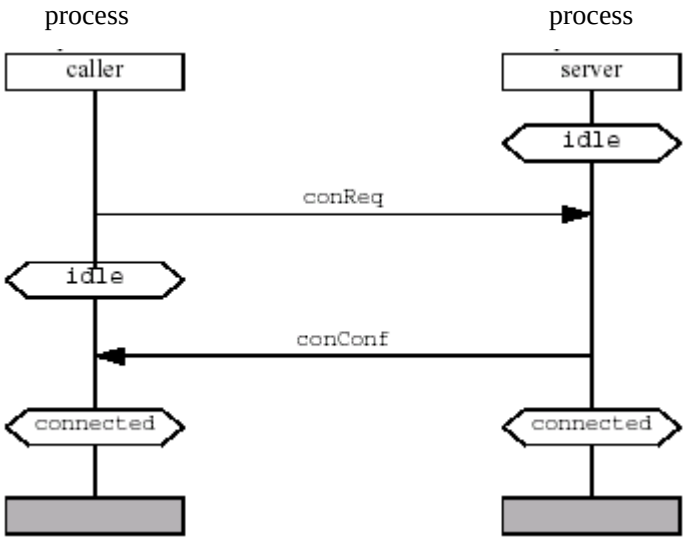
进程 keyboard 调用来源于对象myPhoneBook的方法 set_URL，它是PhoneBook类的一个实例。

6.6 – 状态

生命线表示一个进程，并且根据其内部状态，一个进程对同样的消息反应是不一样的。在生命线上表示一个进程的状态是非常有趣的，也能表示全局状态的信息。那种情况下，状态符号覆盖所涉及的实例。两种情况下，用的符号是一样的。

状态符号

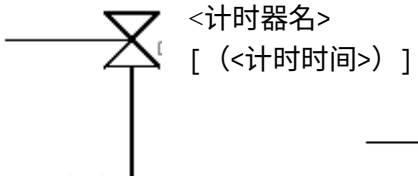
例如：



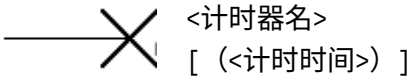
进程 server 走到状态 idle。进程 caller 在开始转移发送 conReq 给出server并走到状态idle。进程server返回 一个消息conConf并走到connected状态。当消息conConf被进程caller收到后，它也走到connected状态。

6.7 – 计时器

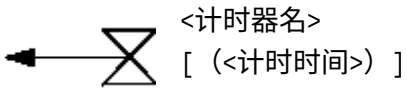
有两个符号用作计时器的动作，要看计时器在开始和结束时是否被连接。计时器的名字是必须的，时间是可选的。未指定时间单位时，通常是RTOS的滴答计数值。



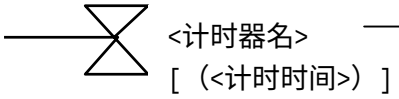
计时器启动，
已连接



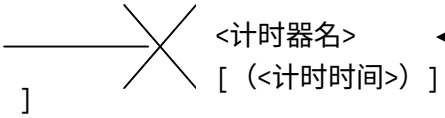
计时器停止，
未连接



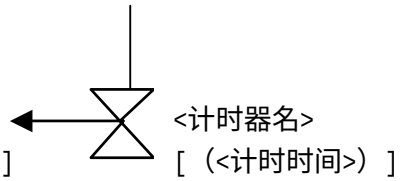
计时器到时，
未连接



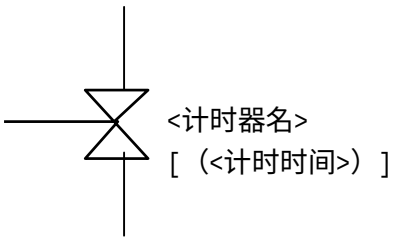
计时器启动，
未连接



计时器停止，
已连接

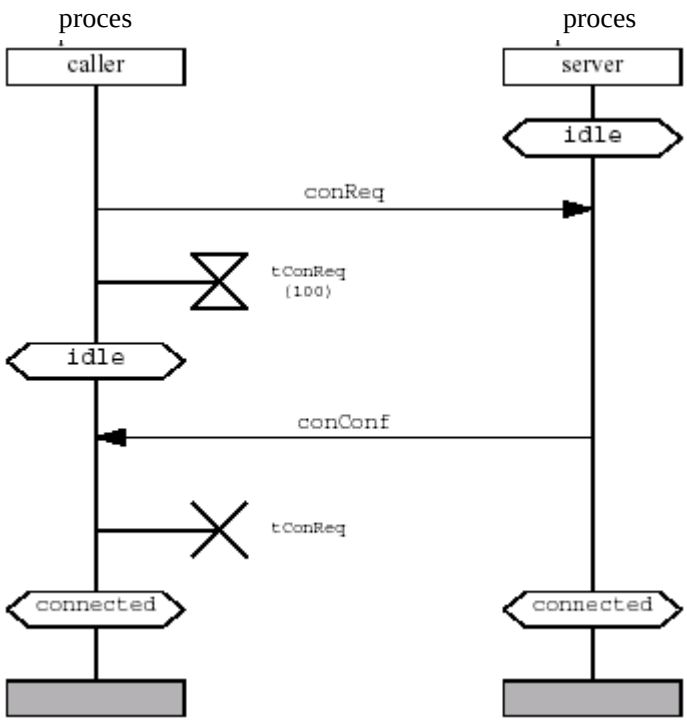


时间到，
已连接



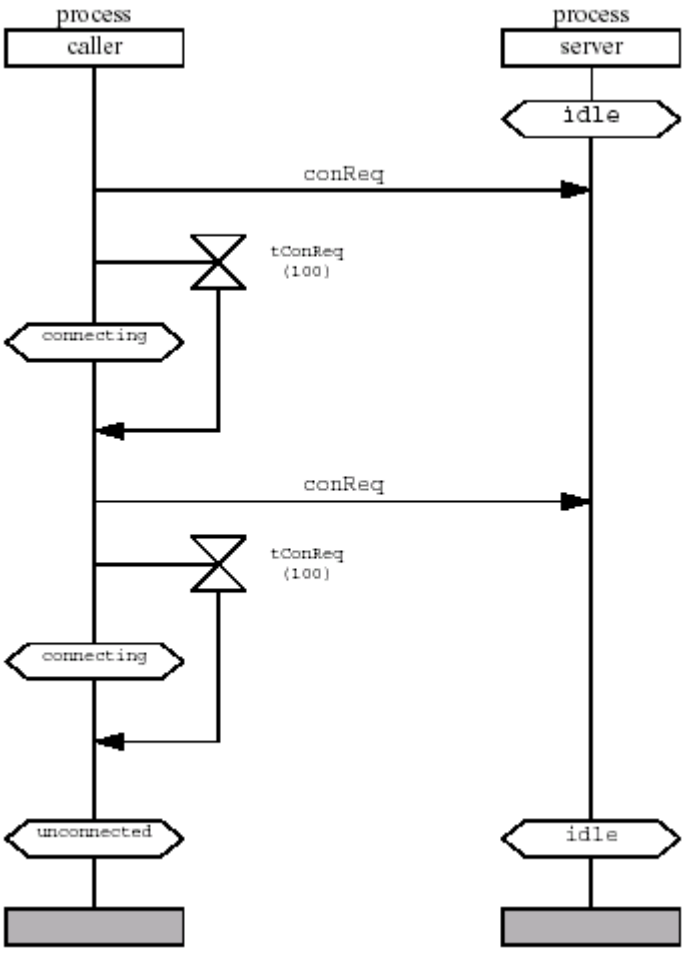
计时器重新启动，
已连接

例如：



进程caller 企图用conReq 消息初始化连接。同时，启动计时器tConReq，因此，如果没有收到回答，它将试图再连接。如果收到回答，计时器取消，并且进程caller走到状态connected。

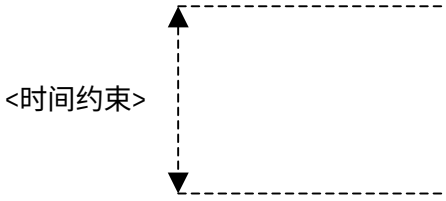
例如：



进程caller企图用conReq初始化连接。由于两次试探后，均未收到回答，它放弃并走到状态unconnected 。

6.8 – 时间间隔

要指定两个事件之间的时间间隔，用下面的符号。

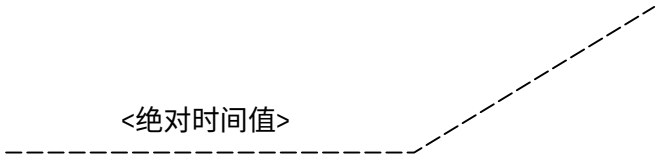


时间约束语法如下：

- 绝对时间值前面加@。

- 相对时间前不加任何标记，
- 时间间隔用方括号括住，
- 时间单位是RTOS特定的--通常是滴答值-除非特别指明（s, ms, μs）。

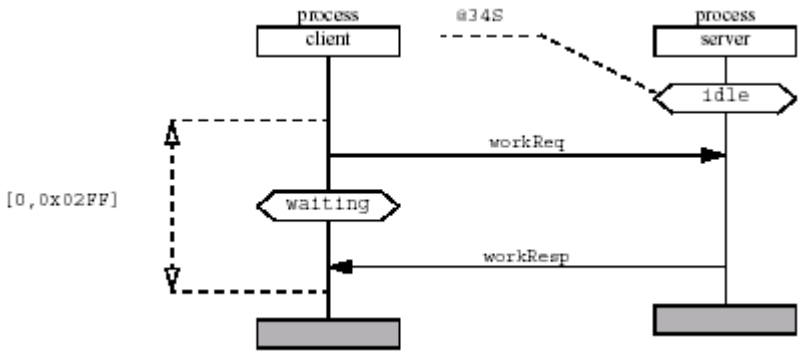
注意，也可以在单个的MSC上用时间约束。
绝对时间也可以用下面的符号指定：



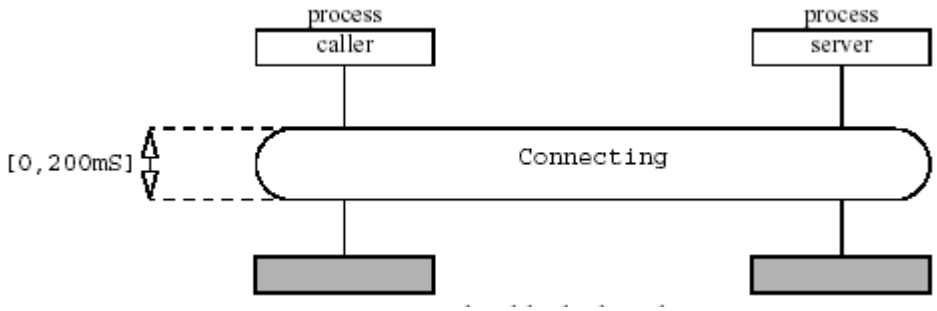
例如：

表1：时间约束表达式的例子

表达式	含义
1.3ms	用1.3ms
[1,3]	用最少1个，最多3个时间单元
@[12.4s,14.7s]	在绝对时间12.4s前不发生，应在绝对时间14.7s之前完成
< 5	用小于5个时间单位。



进程server在绝对时间34秒到达状态idle。
进程client要求进程server 计算某些工作，在小于0x02FF 时间单位内。



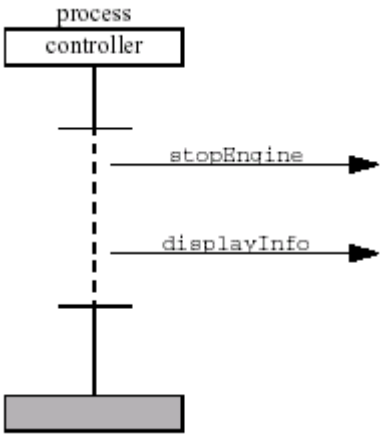
Connecting MSC 应当在200mS内完成

6.9 – 公用区域

不关心事件发生的顺序时，可以使用公用区域（coregion）。公用区域中事件的发生顺序是任意的。公用区域符号代替生命线实例。

公用区域符号

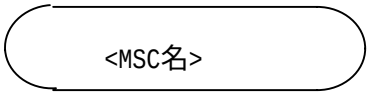
例如：



进程 controller 发送 stopEngine 和 displayInfo，或发送 displayInfo 和 stopEngine。

6.10 - MSC 引用

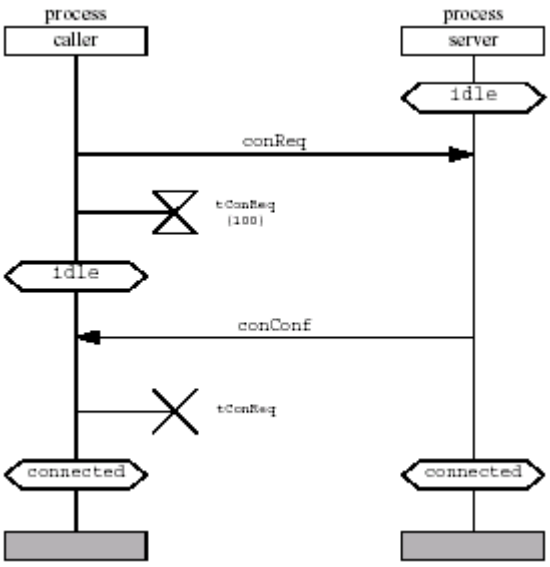
MSC 可以引用另一个MSC。这样使MSC更小和更易读。



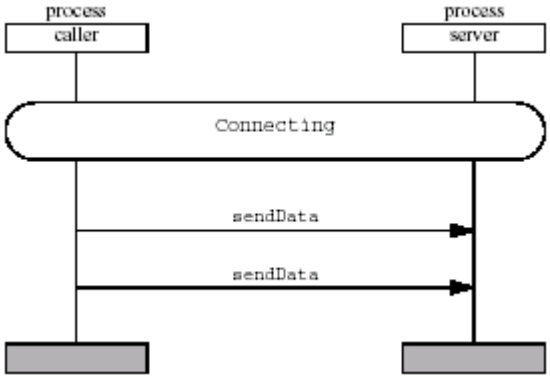
MSC引用符号

一个引用涉及到被连接的实例。一个实例被连接，如果其生命线不出现在符号中。一个实例未被连接，如果其生命线跨越引用符号。

例如：



Connecting MSC

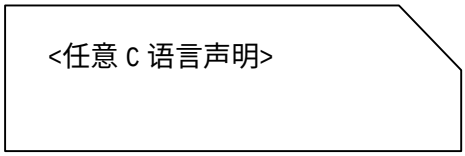


DataTransfer MSC

DataTransfer MSC 开始时就引用Connecting MSC。即，Connecting MSC描述的案况在DataTransfer MSC的其余部分发生前完成。

6.11 – 文本符号

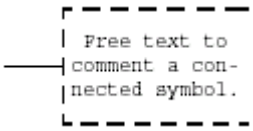
文本符号含数据或变量声明，如果MSC需要的话。



文本符号

6.12 – 注释

顾名思义.....。

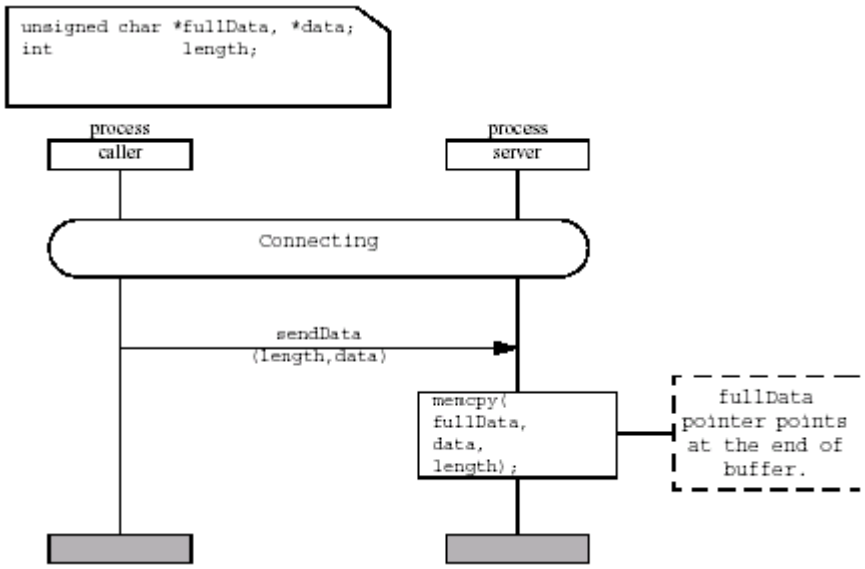


注释符号

6.13 – 动作

一个动作符号包含一组C代码指令集。语法为C语言。

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



DataTransfer MSC

动作符号包含与数据声明相关的标准C指令。

6.14 – 高级 MSC (HMSC)

高级MSC（High level MSC）是MSC如何相互连接的综合视图。它只有几个符号: 开始、停止、选择、并发、状态或条件，以及MSC引用。

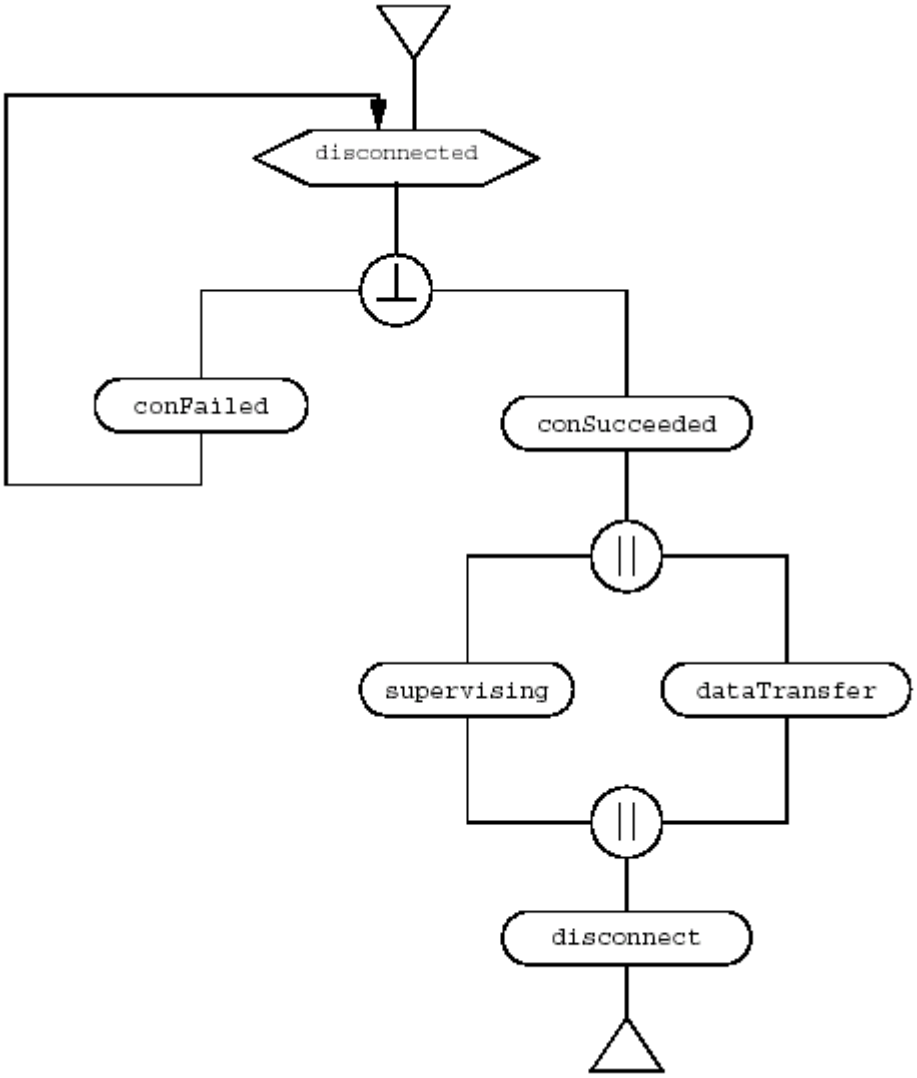
开始　　停止　　并发　　选择

MSC 引用

状态或条件

SDL-RT HMSC 用开始符号开始，停止符号结束。并发符号指下面连接的路径并发执行。选择符号指有一个且只有一个连接路径执行。无论何时两个路径再次相遇时，路径分隔符被重复。这意味着，如果并发符号用来创立两个不同路径，当路径再次合并回来时要用并发符号。

符号用线或带箭头的线连接。符号从上边进入，从其它边离开。



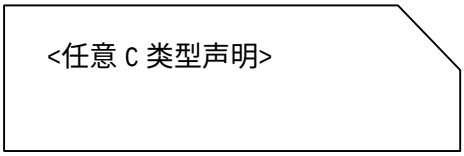
系统开始于disconnected 状态。企图连接或执行案况ConFailed，或conSucceeded。如果conSucceeded 被执行，supervising 和 dataTransfer并发执行。它们合并后返回disconnect，并且结束 HMSC 案况。

7. 数据类型

数据类型，是ANSI C 和C++其中之一。为文档更易读，后面的‘ C代码 ’指‘ ANSI C ’和C++代码。SDL-RT没有预定义的数据类型，但是，有些关键字不能在C代码中使用。考虑 SDL-RT体系结构和C代码的一些概念，需要对某些方面进行论述。

7.1 – 类型定义和头部

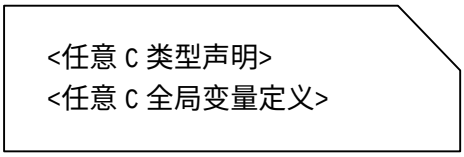
类型在文本符号中声明：



一个代理中声明的类型只在该代理的体系结构中可见。

7.2 – 变量

变量声明在类型声明之后，在同一个文本符号中。



一个代理中声明的变量只在该代理的体系结构中可见。例如，全局变量必须在系统层声明。块中声明的变量对于更高层的块是不可见的。SDL-RT进程中声明的变量只局限于此进程，其它进程是不可见的，也不能操作。

7.3 - C 函数

SDL-RT 内部 C 函数用SDL-RT过程符号定义。SDL-RT的过程可以用SDL-RT图形化定义或用C文本定义。当用C定义时，不能用过程调用符号。动作符号中可以使用标准C语句。

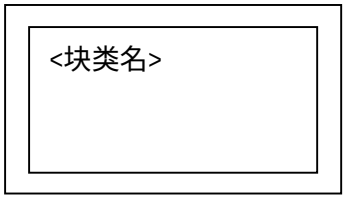
7.4 – 外部函数

外部C函数可以从SDL-RT系统中调用。可以在系统或外部C头文件中声明原型。SDL-RT工具在编译和连接时收集正确的文件。

8. 面向对象

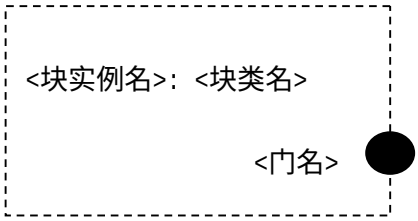
8.1 – 块类

定义一个块类，可以在SDL-RT系统中多次使用。SDL-RT 块不支持任何其它面向对象的特征。一个块类符号是带双线的块符号。没有信道与其连接。

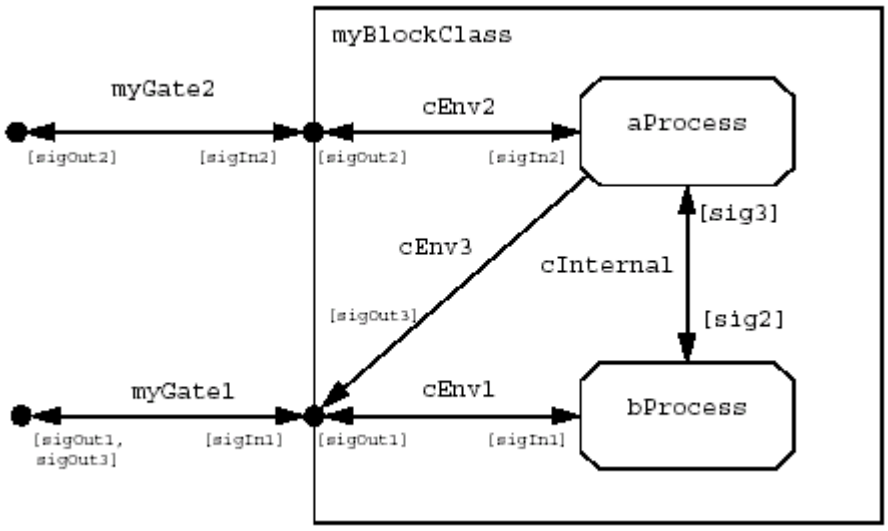


一个块类可以在块或者系统中实例化。块符号的语法如下：
<块实例名>:<块类名>

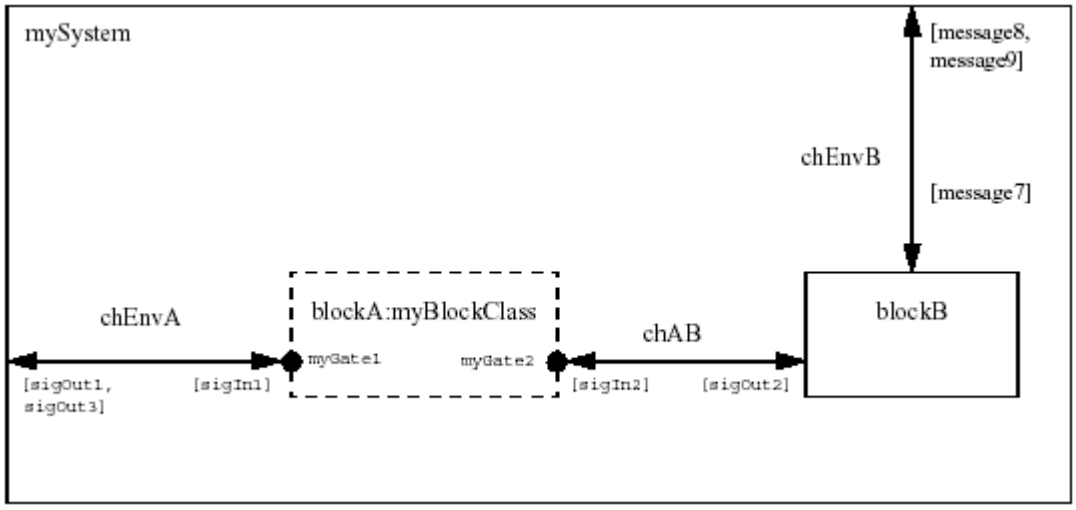
消息进或出块类需要通过门。在块类的框图中，门代表块类框图的出。当块类实例化时，门被连接到周围的SDL-RT体系结构。列在门中的消息与列在连接的信道上的消息是一致的。



例如：



myBlockClass 块类的定义框图



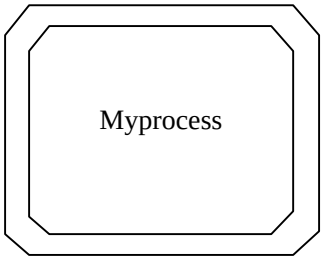
blockA 是 myBlockClass 的一个实例

8.2 – 进程类

定义进程类可以：

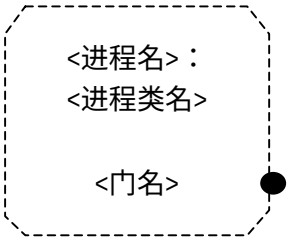
- 在SDL-RT体系结构中不同的地方，同一个进程可以有几个实例，
- 从一个进程的超类继承过来，
- 将转移和状态特殊处理。

进程类符号是一个带双框的进程符号。没有与其连接的信道。



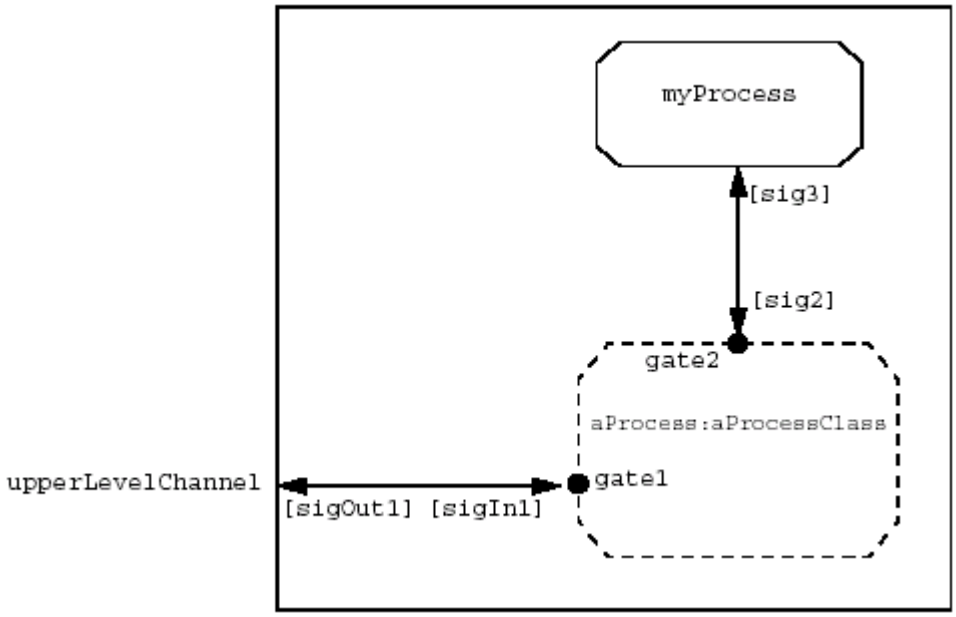
一个进程类可以避免在块或系统中实例化。进程符号中的语法是：
<进程实例名>：<进程类名>

消息通过门进入和走出进程。在进程类框图中，门的表示在进程类框架之外。当进程类被初始化时，门连接到周围的SDL-RT体系结构。门中列写的消息与连接的信道上列写的消息一致。门的名字出现在进程符号中，用黑圈表示连接点。



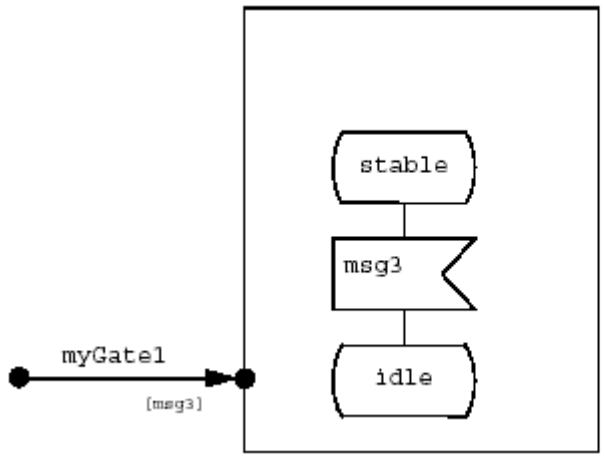
由于不假设类知道周围的体系结构，消息输出不能用TO_NAME的概念。而是用TO_ID、VIA、或TO_ENV。

例如：

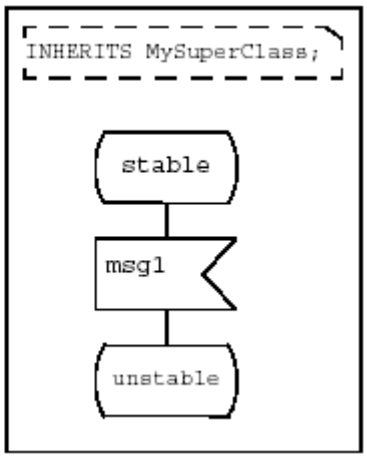


SDL-RT的转移、门和数据在特殊化时，是可以被重新定义的元素。在子类中，从继承的超类中用**附加头部符号**中的关键字INHERITS定义。依据超类中的定义，有几种方法对进程类进行特殊化。

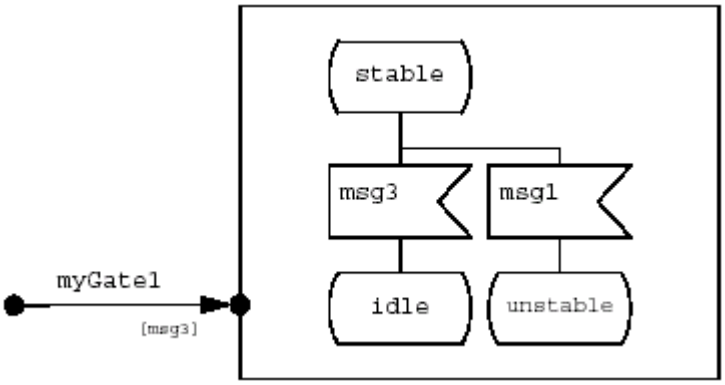
- 如果子类中的元素是新的，就简单的在超类的定义上添加。



MySuperClass

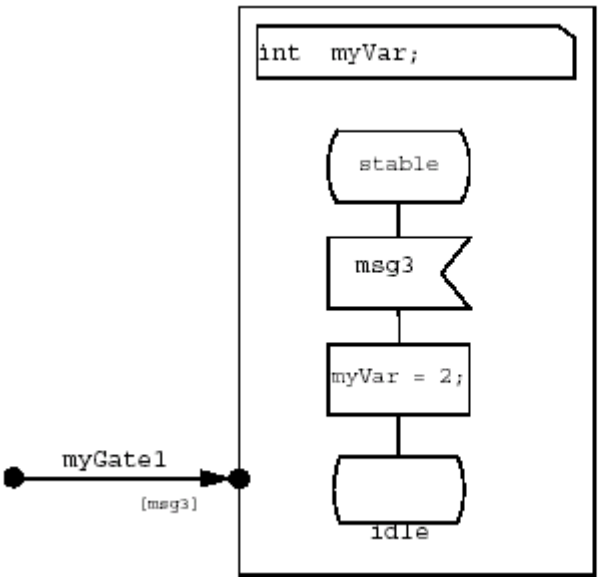


MyClass

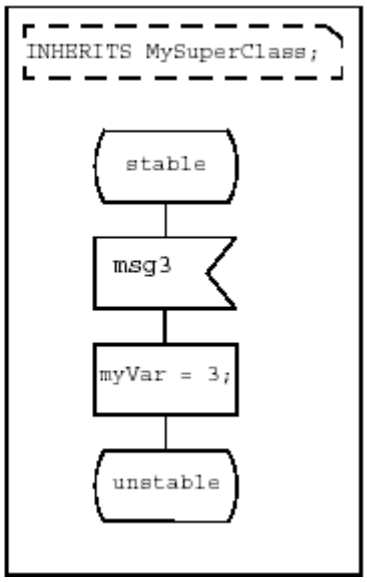


MyClass的一个实例

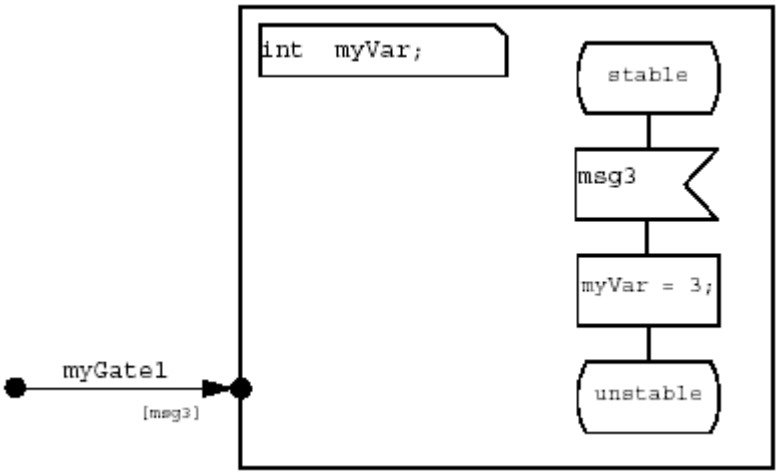
- 如果元素在超类中存在，新的元素定义覆盖超类中的定义。



MySuperClass

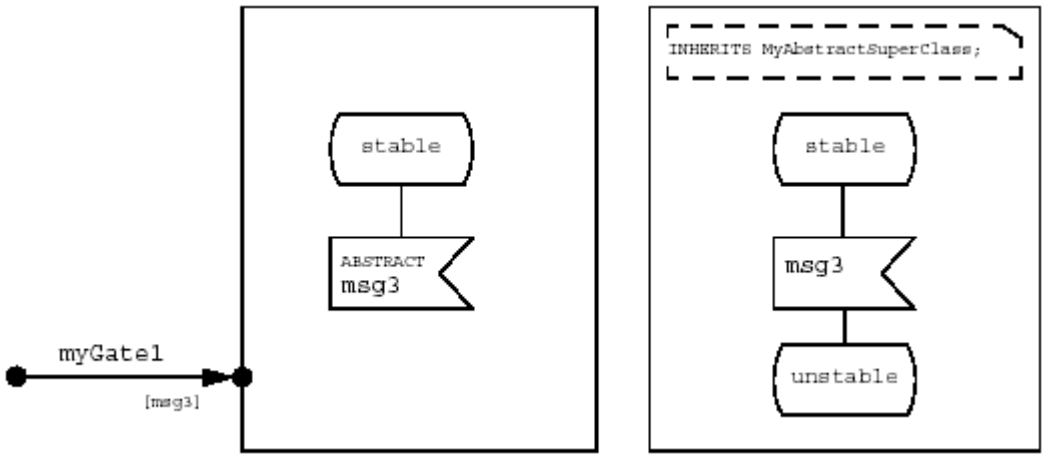


MyClass

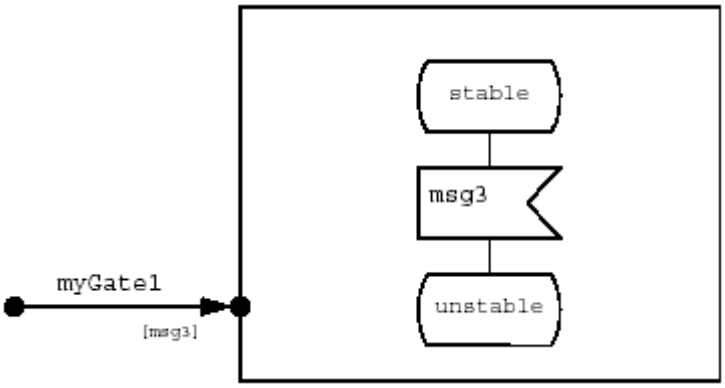


MyClass的一个实例

- 一个类可以用关键字ABSTRACT定义为抽象类。它表示此类不能实例化，它必须被特殊化。一个类可以定义抽象转移或抽象门，这表示抽象类转移或门的存在，但是不能被定义。这样，一个类可以明显抽象，并且必须这样定义。

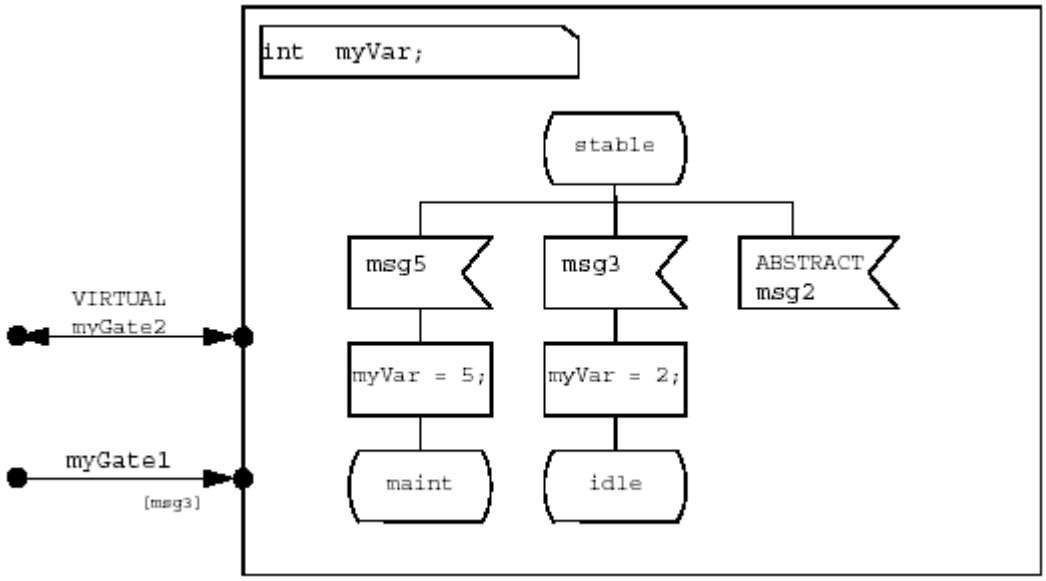


MyAbstractSuperClass MyClass

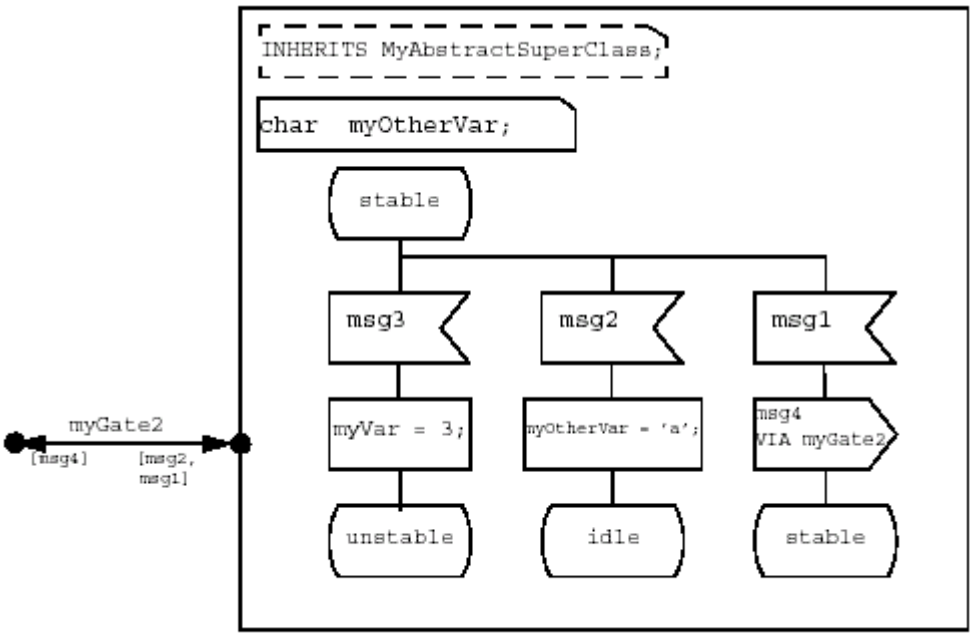


MyClass的一个实例

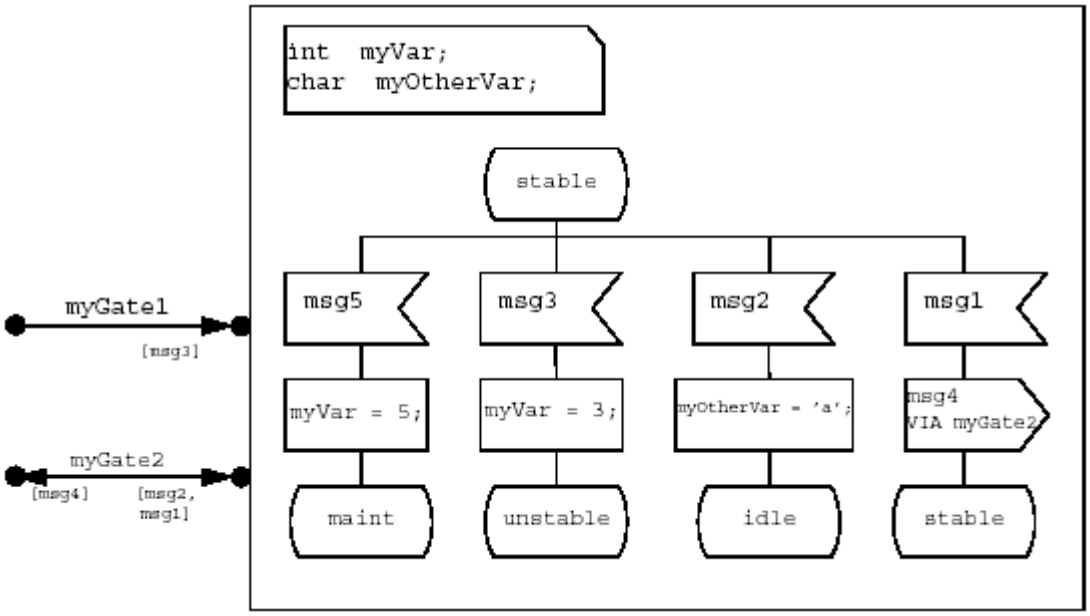
这里用一个综合例子表明面向对象的概念和最终的对象。



MyAbstractSuperClass



MyClass



MyClass 的一个实例

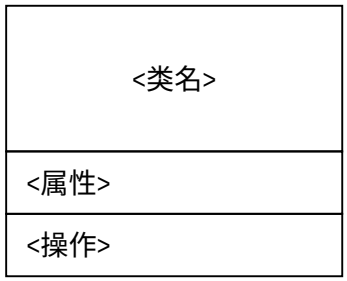
8.3 – 类图

SDL-RT 的类图与UML 1.3 类图一样。定义带特殊图形符号的规范化衍型（stereotype）与SDL图形表示连接。下面的段落简要解释所有的符号。更详细的信息请参看OMG UML v1.3 的规格说明。

8.3.1 类

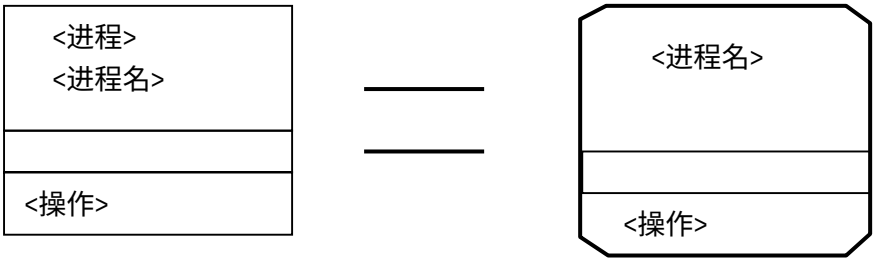
一个类（**class**）是对一组具有相类似的结构、行为和关系的对象的描述符号。

不带详细信息的类符号



类符号的全部表示

一个衍型（**stereotype**）是UML词汇的扩展，可以创立特殊类型的类。如果出现，衍型放在类名字的上面。除这种纯文本表示之外，可以用特殊的符号替换类符号。



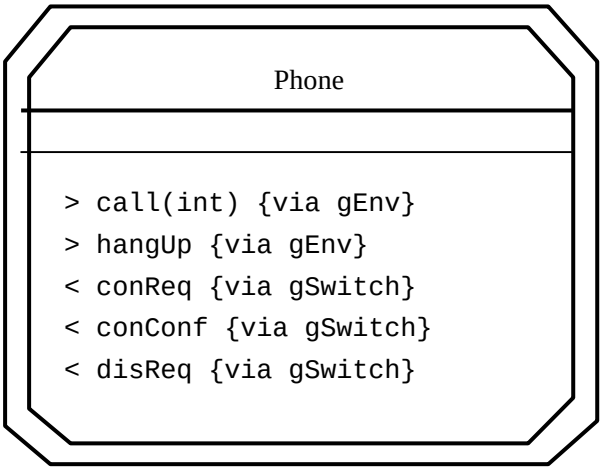
类衍型作为一个过程　　类衍型作为一个过程

类分为**主动类**（**active class**）和**被动类**（**passive class**）。主动类实例具有一个控制线程并可以初始化控制活动。被动类保持数据，但是，不初始化控制。在类图中，代理用主动类表示。代理类型用类的衍型定义。已知的衍型是：系统、块、块类、进程和进程类。活动类没有任何属性。对活动类定义的操作是进入或出去的异步消息。语法为：

<消息方向> <消息名> [(<参数类型>)] [{via <门名>}]

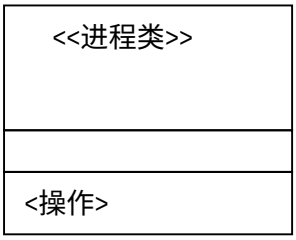
<消息方向> 可以是下面的特性之一：

- '>' 用于进入的消息，
- '<' 用于出去的消息。

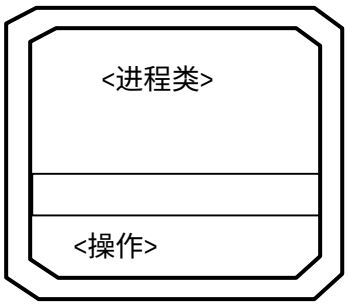


进程类pPhone可以通过门gEnv接收消息call 和hangU。通过门gSwitch 发送消息conReq, conConf, disReq, disConf。

对有衍型的类，预先定义的图形符号如下：

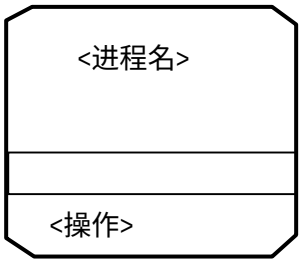


衍型的类作为一个进程类

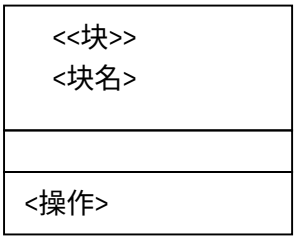


衍型的类作为一个进程类

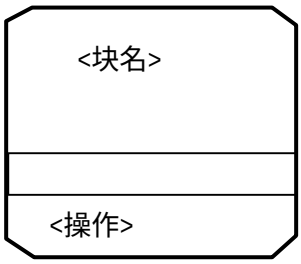
衍型的类作为一个进程



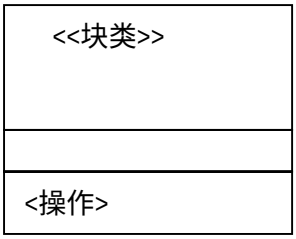
衍型的类作为一个进程



衍型的类作为一个块

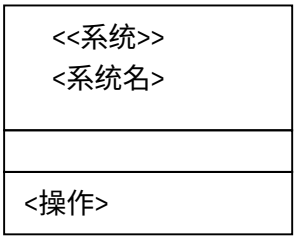


衍型的类作为一个块

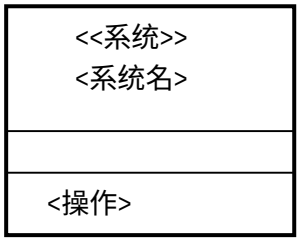


衍型的类作为一个块类

衍型的类作为一个块类



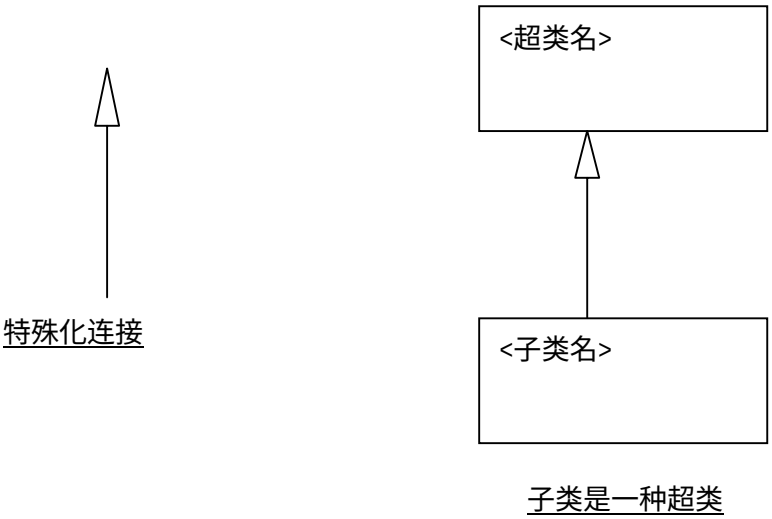
衍型的类作为一个系统



衍型的类作为一个系统

8.3.2 特殊化

特殊化（Specialisation）定义为两个类之间的“是….的一种（is a kind of）”关系。最通用的类称为超类（superclass），特殊化的类称为子类（subclass）。

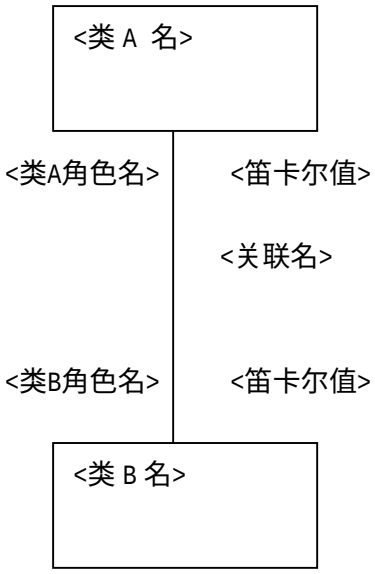


子类是一种超类

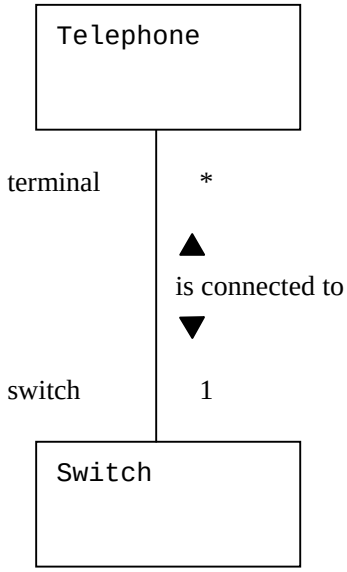
从子类到超类的关系称为**泛化（generalisation）**。

8.3.3 关联

一个关联（**association** ）是两个类之间的关系。它使对象之间相互通信。关联的含义由其名字或相关联的角色名定义。笛卡尔值（**Cardinality** ）表示在关联的端头连接多少个对象。



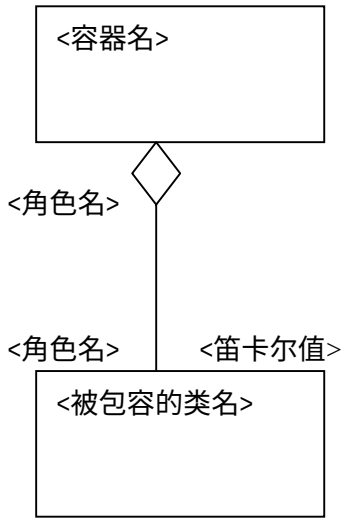
每个Telephone连接到一个Switch。一个Switch连接多个Telephone。
一个 Telephone对于Switch来讲是一个终端。



一个类的实例用关联类通过其角色名标识出来。
上面的例子中，交换机（Switch）的一个实例标识出Telephone的实例，通过名terminal连接。

8.3.4 聚合

聚合（Aggregation）定义两个类‘ 是…..的一部分（is a part of） ’的关系。

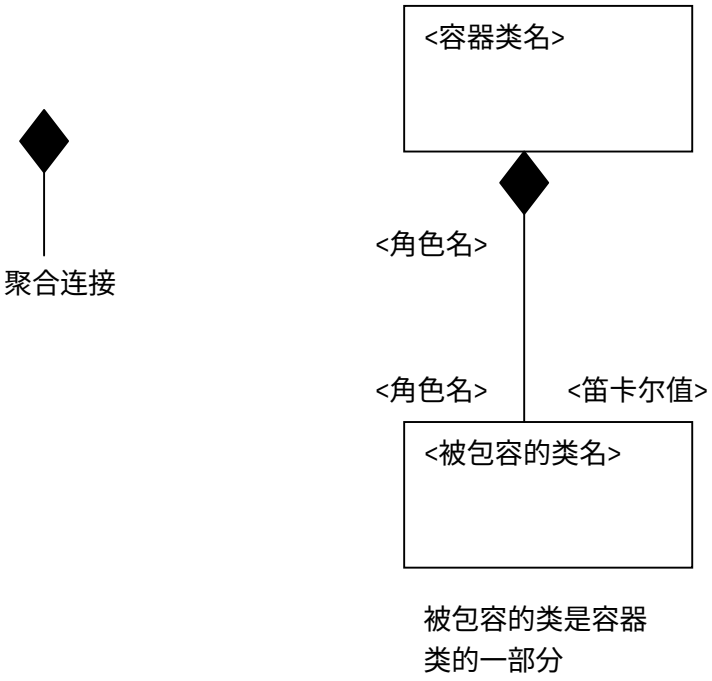


被包容的类是容器类的一部分

对象相互标识为有规则的关联（参见第68页“关联”）。

8.3.5 组合

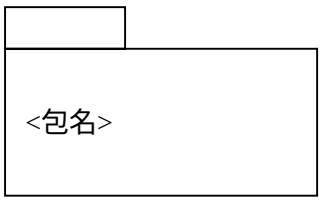
组合(composition)是聚合的严格表示形式，表明那部分的存在取决于容器。



对象相互标识为有规则的关联（参见第68页“关联”）。

8.4 – 包

包（**package** ）是一个分离的实体，其中包含类、代理或代理的类。可通过其名字引用。



它可以含：

- 类
- 系统
- 块
- 多个块的类
- 进程
- 多个进程的类
- 过程
- 数据定义.

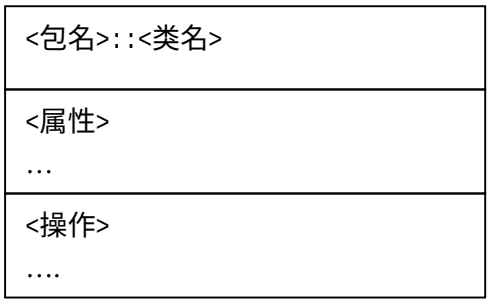
8.4.1 在代理中的用法

代理类定义可以集中在一个包里。为了能够使用包中定义的类，SDL-RT系统可以明显的装入包，在系统层的附加头部符号中用关键字USE。

8.4.2 在类图中的用法

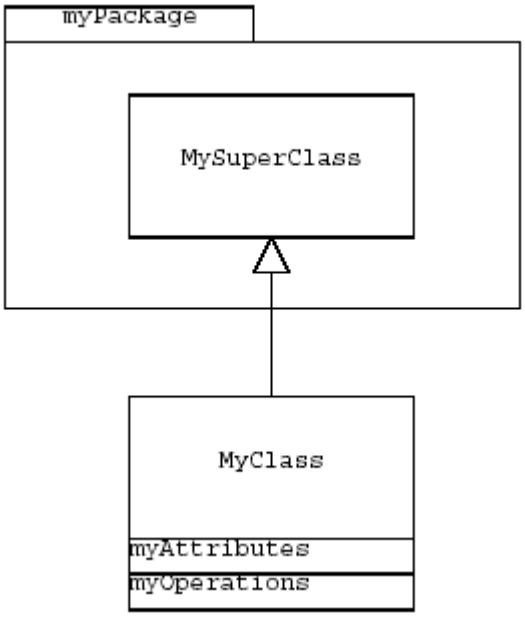
在包中定义的类有两种引用方式：

•用包名作为类名的前缀



类<类名> 在包<包名>中定义

• 把包图形符号作为类符号的一个容器



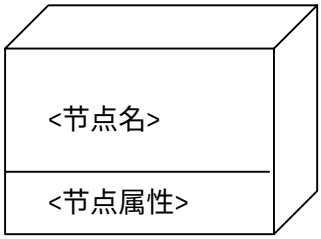
MyClass 将myPackage 中定义的MySuperClass特殊化

9. 部署图

部署图表示在一个分布式系统中，运行时所处理的元素的物理配置情况。

9.1 – 节点

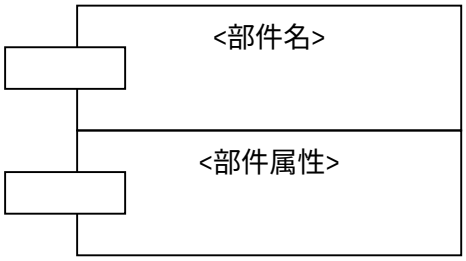
一个节点（**node**） 表示处理资源的一个物理对象。



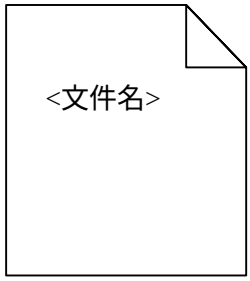
9.2 – 部件

一个部件（**component**）表示系统中可实现的分布片。有两种类型的部件：

- 可执行部件

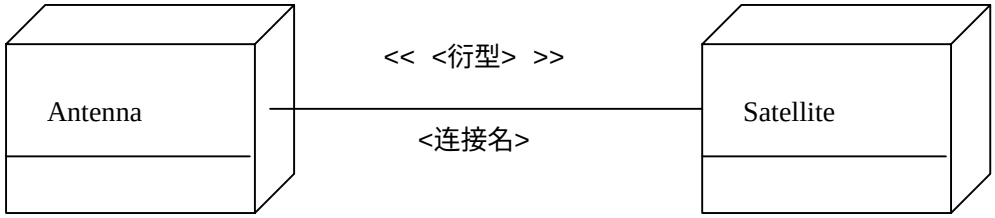


- 文件部件



9.3 – 连接

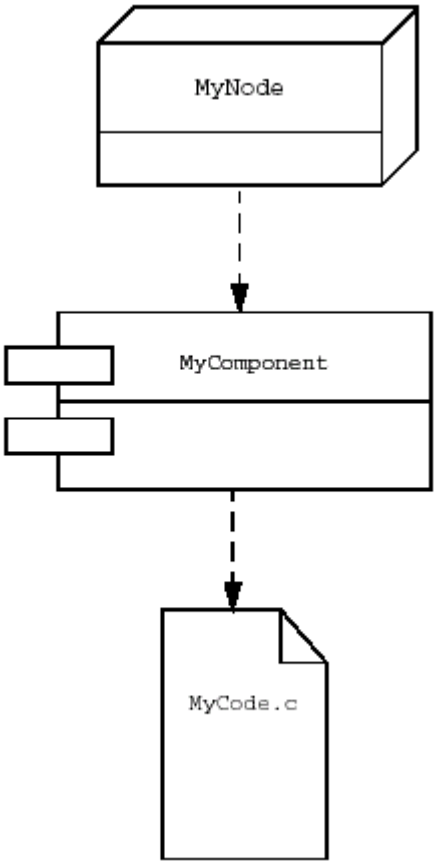
一个连接（**connection**）是两个节点或两个可执行部件之间的物理链。用名字和衍型（stereotype）来定义。



9.4 – 依赖性

两个元素之间的**依赖性（Dependency）**可以用图形表示。

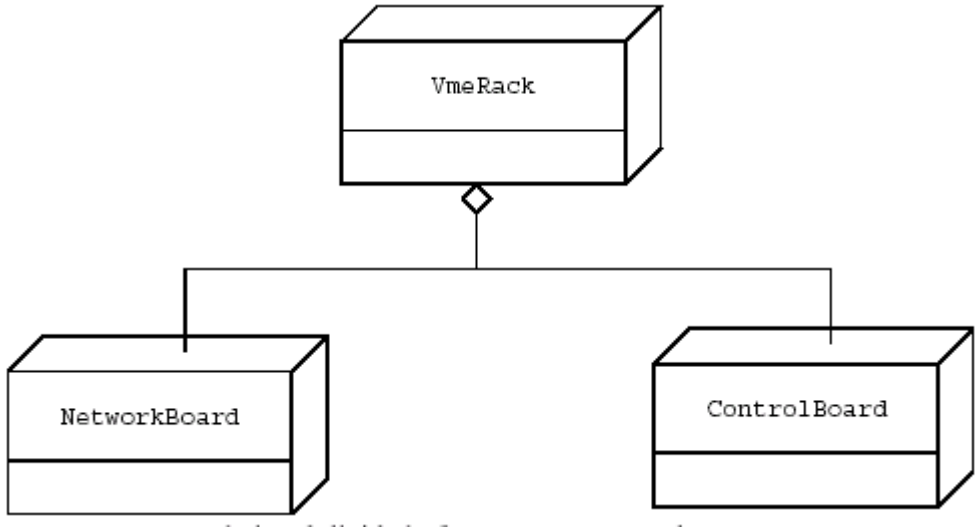
- 从节点到一个可执行部件的依赖性，意味着在节点上执行。
- 从一个部件到文件部件的依赖性，意味着部件构造需要此文件。
- 从一个节点到文件的依赖性，意味着节点构造需要此文件。



MyComponent 在MyNode运行，并需要 MyCode.c 文件建造。

9.5 – 聚合

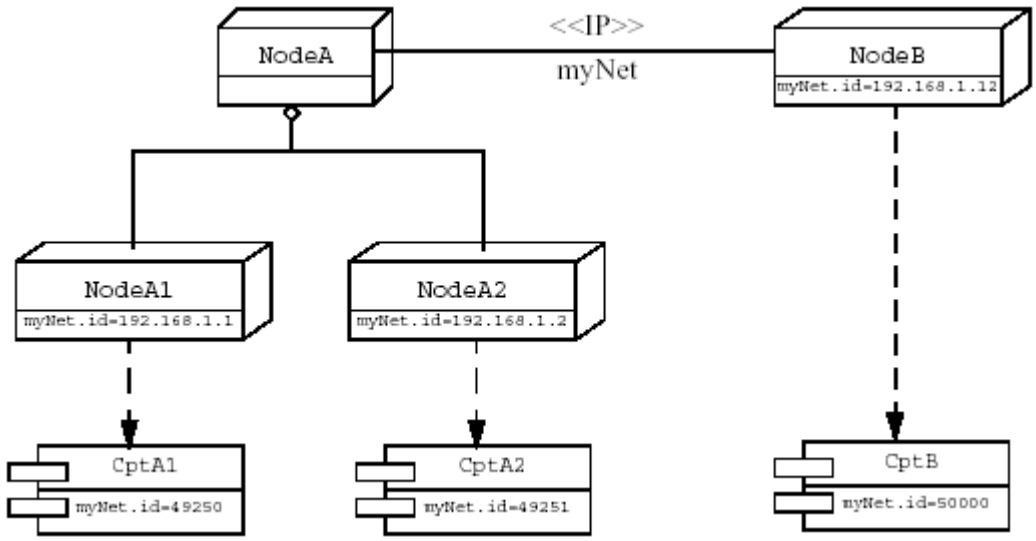
一个节点可以再分解成多个节点。



VmeRack 节点进一步分解成NewtorkBoard和ControlBoard。

9.6 – 节点和部件标识符

属性被所连接的节点或部件用来相互标识。



CptB 可以通过myNet 连接到CptA1，使用NodeA1 myNet.id 属性和CptA1 myNet.id 属性。

不需要时，节点属性可以忽略。

10. 框图中包含的符号

下表表示那些符号可以包含在特殊的框图类型中。

此列中的图可以使用右面所列的符号。	包	块类	进程类	块	进程	过程声明	信号灯声明	信道	附加头部	文本	门定义	门使用	行为符号	类	关联 组合	特殊化	节点 部件 连接 依赖性	聚合
包	X	X	X	X	X	X	X	X	X	X	X	-	-	X	X	X	-	X
类图	X	X	X	X	X	-	-	-	-	-	-	-	-	X	X	X	-	X
块类	-	-	-	X	X	X	X	X	X	X	X	X	-	-	-	-	-	-
进程类	-	-	-	-	-	-	-	-	X	X	X	-	X	-	-	-	-	-
块	-	-	-	X	X	X	X	X	X	X	-	X	-	-	-	-	-	-
进程	-	-	-	-	-	-	-	-	X	X	-	-	X	-	-	-	-	-
过程	-	-	-	-	-	-	-	-	-	X	-	-	X	-	-	-	-	-
部署																	X	X

图中第一列可以含其它列的图。例如，进程符号可以含附加头部符号、文本符号和所有的行为符号。行为符号是第14页“行为”中所描述的所有符号。

11. 文本表示

存储格式遵循XML(源于W3C的eXtensible Markup Language 标准，见http://www.w3.org)标准所使用的DTD(Document Type Definition)：

```
<!-- Entity for booleans -->
<!-- ================== -->
<!ENTITY % boolean "(TRUE|FALSE)">
<!-- Entities for symbol types -->
<!-- ======================== -->
<!ENTITY % sdlSymbolTypes1 "sdlSysDgmFrm|sdlSysTypeDgmFrm|sdlBlkDgmFrm|sdlBlkTypeDgmFrm|
sdlBlkType|sdlBlk|sdlBlkTypeInst|sdlPrcsType|sdlPrcs|sdlPrcsTypeInst">
<!ENTITY % sdlSymbolTypes2 "sdlInherits|sdlPrcsTypeDgmFrm|sdlPrcsDgmFrm|sdlPrcdDgmFrm|
sdlStart|sdlState|sdlInputSig|sdlSendSig|sdlSaveSig|sdlContSig">
<!ENTITY % sdlSymbolTypes3 "sdlTask|sdlDecision|sdlTransOpt|sdlJoin|sdlText|sdlComment|
sdlTextExt|sdlCnctrOut|sdlCnctrIn|sdlPrcsCreation|sdlStop">
<!ENTITY % sdlSymbolTypes4 "sdlInitTimer|sdlResetTimer|sdlSemDecl|sdlSemTake|sdlSemGive|
sdlPrcdProto|sdlPrcdDecl|sdlPrcdCall|sdlPrcdStart|sdlPrcdReturn">
<!ENTITY % sdlSymbolTypes "%sdlSymbolTypes1;|%sdlSymbolTypes2;|%sdlSymbolTypes3;|
%sdlSymbolTypes4;">
<!ENTITY % mscSymbolTypes1
"mscExternalFrm|mscInlineExpr|mscLifeline|mscSemaphore|mscLostMsg|
mscFoundMsg|mscComment">
<!ENTITY % mscSymbolTypes2 "mscGenNameArea|mscText|mscAbsTimeConstr|mscCondition|mscMscRef|
mscInlineExprZone|mscSave">
<!ENTITY % mscSymbolTypes "%mscSymbolTypes1;|%mscSymbolTypes2;">
<!ENTITY % hmscSymbolTypes "hmscDgmFrm|hmscParallel|hmscStart|hmscEnd|hmscCondition|
hmscMscRef|hmscAlternativePoint">
<!ENTITY % mscdocSymbolTypes "mscdocDgmFrm|mscdocMscRef|mscdocHeader">
<!ENTITY % umlClassSymbolTypes
"umlClassDgmFrm|umlPckg|umlClass|umlComment|umlSys|umlBlkCls|
umlBlk|umlPrcsCls|umlPrcs">
<!ENTITY % umlDeplSymbolTypes "umlDeplDgmFrm|umlNode|umlComp|umlFile">
<!ENTITY % umlUCSymbolTypes "umlUCDgmFrm|umlUseCase|umlActor">
<!ENTITY % SymbolType
"(%sdlSymbolTypes;|%mscSymbolTypes;|%hmscSymbolTypes;|%mscdocSymbolTypes;|
%umlClassSymbolTypes;|%umlDeplSymbolTypes;|%umlUCSymbolTypes;)">
<!-- Entity for lifeline component type -->
<!-- ================================= -->
<!ENTITY % LifelineComponentType "(norm|susp|meth|coreg|act)">
<!-- Entity for time interval type -->
<!-- ============================ -->
<!ENTITY % TimeIntervalType "(start|end|timeout|constraint)">
```

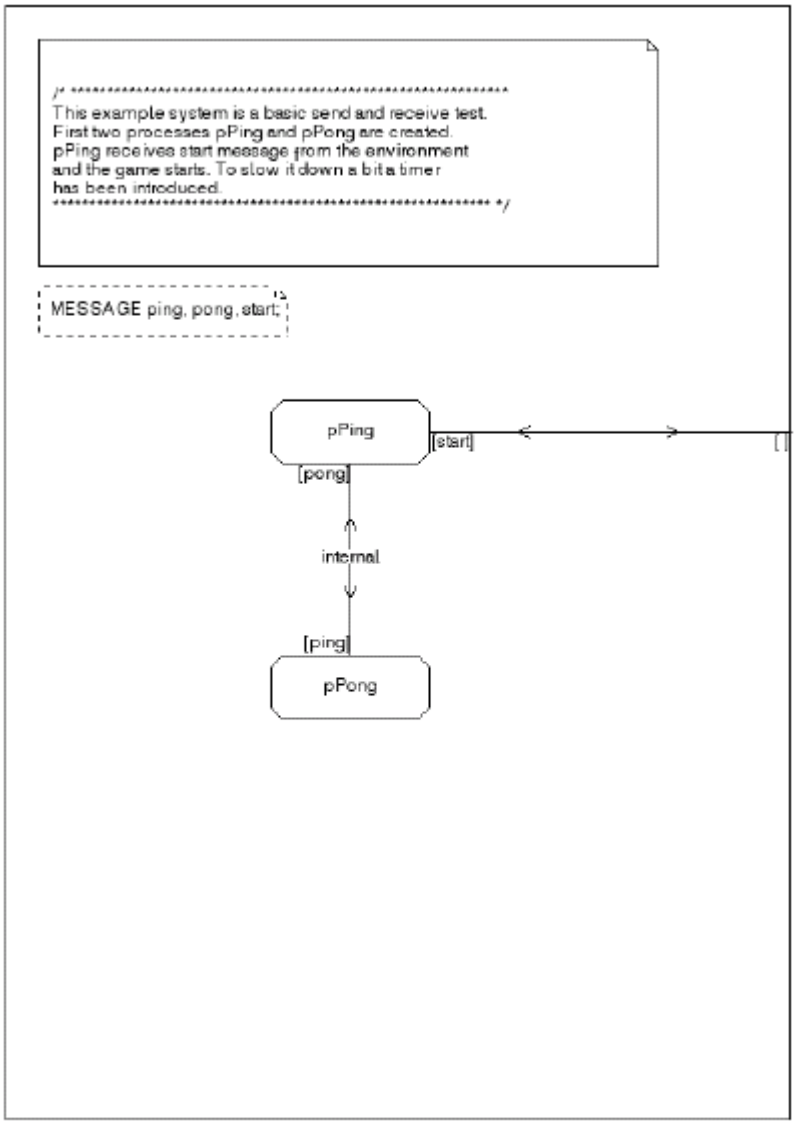
```
<!-- Entity for connector types -->
<!-- ========================== -->
<!ENTITY % ConnectorType
"(void|chnl|chnlgate|sdlarrow|mscvoid|mscgate|mscarrowgate|hmscarrow|
umlcvoid|umlassoc|umlrole|umldvoid)">
<!-- Entity for side for connectors -->
<!-- ============================== -->
<!ENTITY % Side "(n|s|w|e|x|y)">
<!-- Entity for end types for connectors -->
<!-- ================================== -->
<!ENTITY % ConnectorEndType "(voidend|arrow|midarrow|outltri|outldiam|filldiam)">
<!-- Entity for link segment orientation -->
<!-- ================================== -->
<!ENTITY % Orientation "(h|v)">
<!-- Entity for link types -->
<!-- ===================== -->
<!ENTITY % LinkType
"(sbvoid|dbvoid|ssvoid|dsvoid|chnl|dec|transopt|msg|rtn|instcre|assoc|spec|aggr|comp|cnx|
dep)">
<!-- Entity for diagram types -->
<!-- ======================== -->
<!ENTITY % DiagramType
"(sys|systype|blk|blktype|prcs|prcstype|prcd|msc|hmsc|mscdoc|class|usec|
depl)">
<!-- Element for text in symbols/links/... -->
<!-- ===================================== -->
<!ELEMENT Text (#PCDATA)>
<!ATTLIST Text
id CDATA "0"
>
<!-- Element for lifeline symbol components (MSC specific) -->
<!-- ====================================================== -->
<!-- The "Text" component and "width" attribute are only for action symbols -->
<!ELEMENT LifelineComponent (Text?)>
<!ATTLIST LifelineComponent
type %LifelineComponentType; #REQUIRED
height CDATA #REQUIRED
color CDATA "#000000"
width CDATA "-1"
>
<!-- Element for lifeline symbol time intervals (MSC specific) -->
<!-- ========================================================= -->
<!ELEMENT TimeInterval (Text)>
<!ATTLIST TimeInterval
```

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

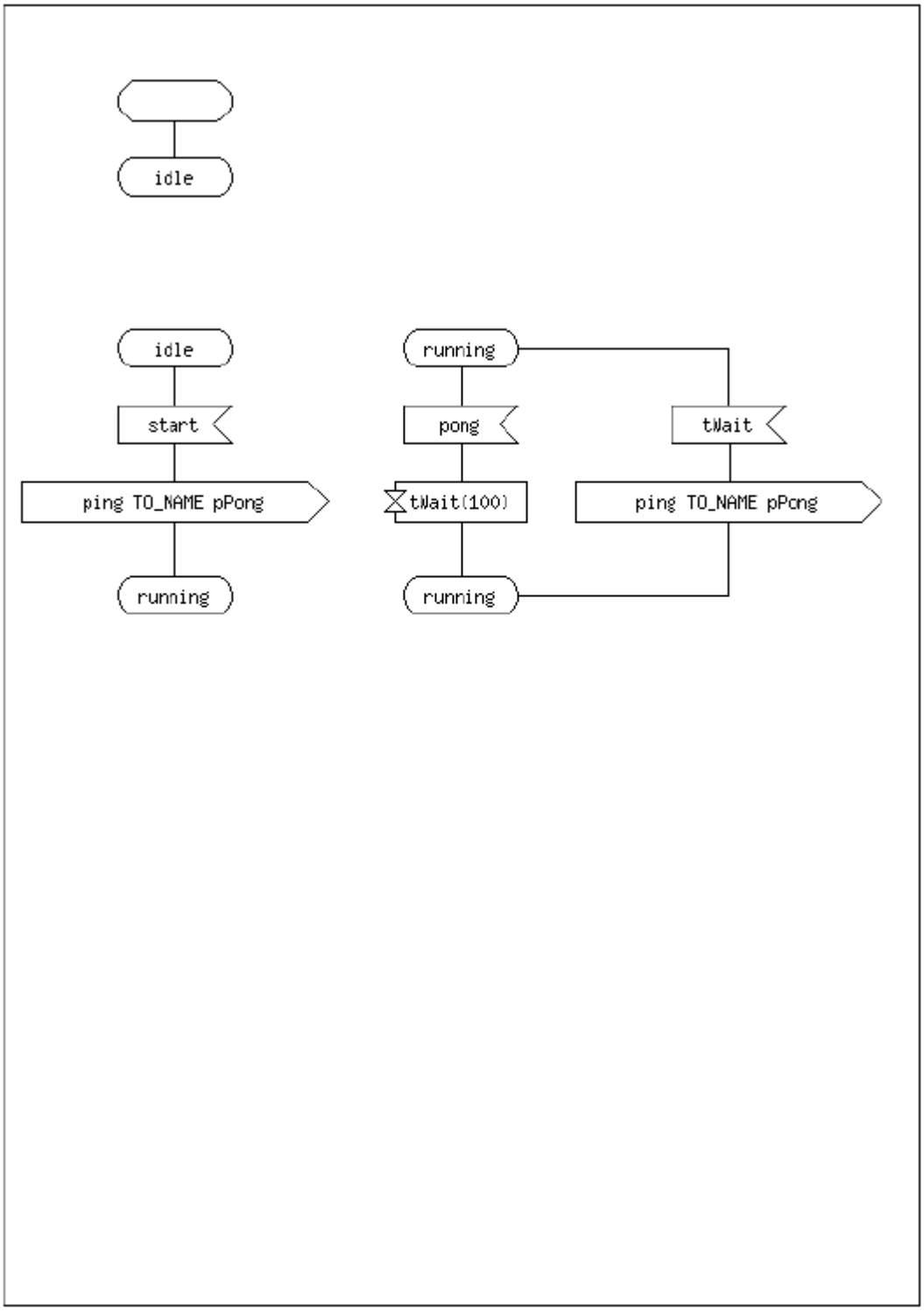
```
type %ConnectorType; #REQUIRED
isOutside %boolean; #REQUIRED
side %Side; #REQUIRED
position CDATA #REQUIRED
endType %ConnectorEndType; #REQUIRED
>
<!-- Element for link segments -->
<!-- ========================= -->
<!ELEMENT LinkSegment EMPTY>
<!ATTLIST LinkSegment
orientation %Orientation; #REQUIRED
length CDATA #REQUIRED
>
<!-- Element for links -->
<!-- ================= -->
<!ELEMENT Link (Text, Connector, Connector, LinkSegment*)>
<!ATTLIST Link
type %LinkType; #REQUIRED
textSegmentNum CDATA #REQUIRED
color CDATA "#000000"
>
<!-- Element PageSpecification -->
<!-- ========================= -->
<!-- Attributes for diagram pages; all dimensions are centimetres -->
<!ELEMENT PageSpecification EMPTY>
<!ATTLIST PageSpecification
pageWidth CDATA "21"
pageHeight CDATA "29.7"
topMargin CDATA "1.5"
bottomMargin CDATA "1.5"
leftMargin CDATA "1.5"
rightMargin CDATA "1.5"
pageFooter %boolean; "TRUE"
>
<!-- Element for diagrams -->
<!-- ==================== -->
<!ELEMENT Diagram (PageSpecification, Symbol, Link*)>
<!ATTLIST Diagram
type %DiagramType; #REQUIRED
nbPagesH CDATA "1"
nbPagesV CDATA "1"
cellWidthMm CDATA "5"
```

12. 例子

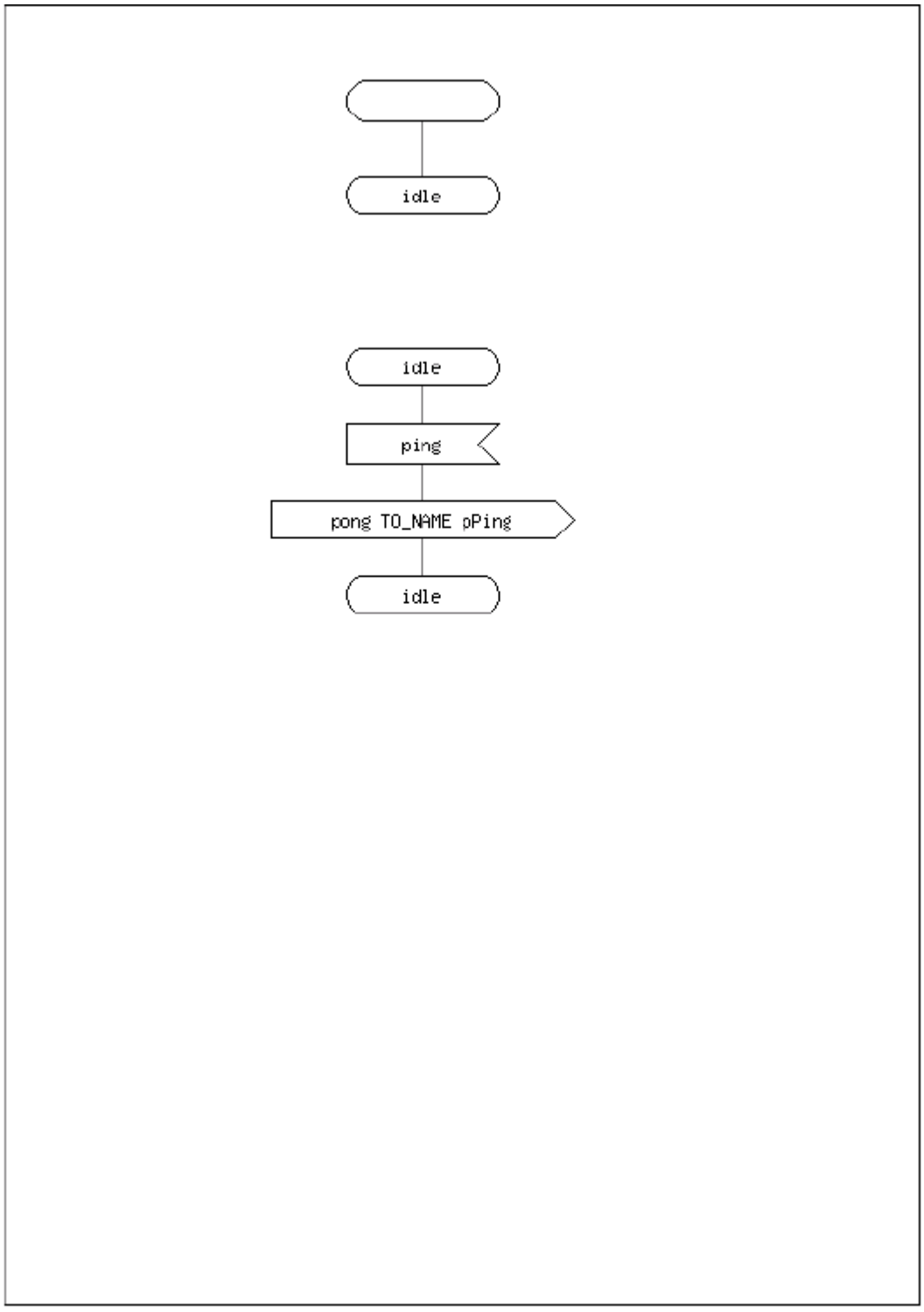
12.1 - Ping Pong



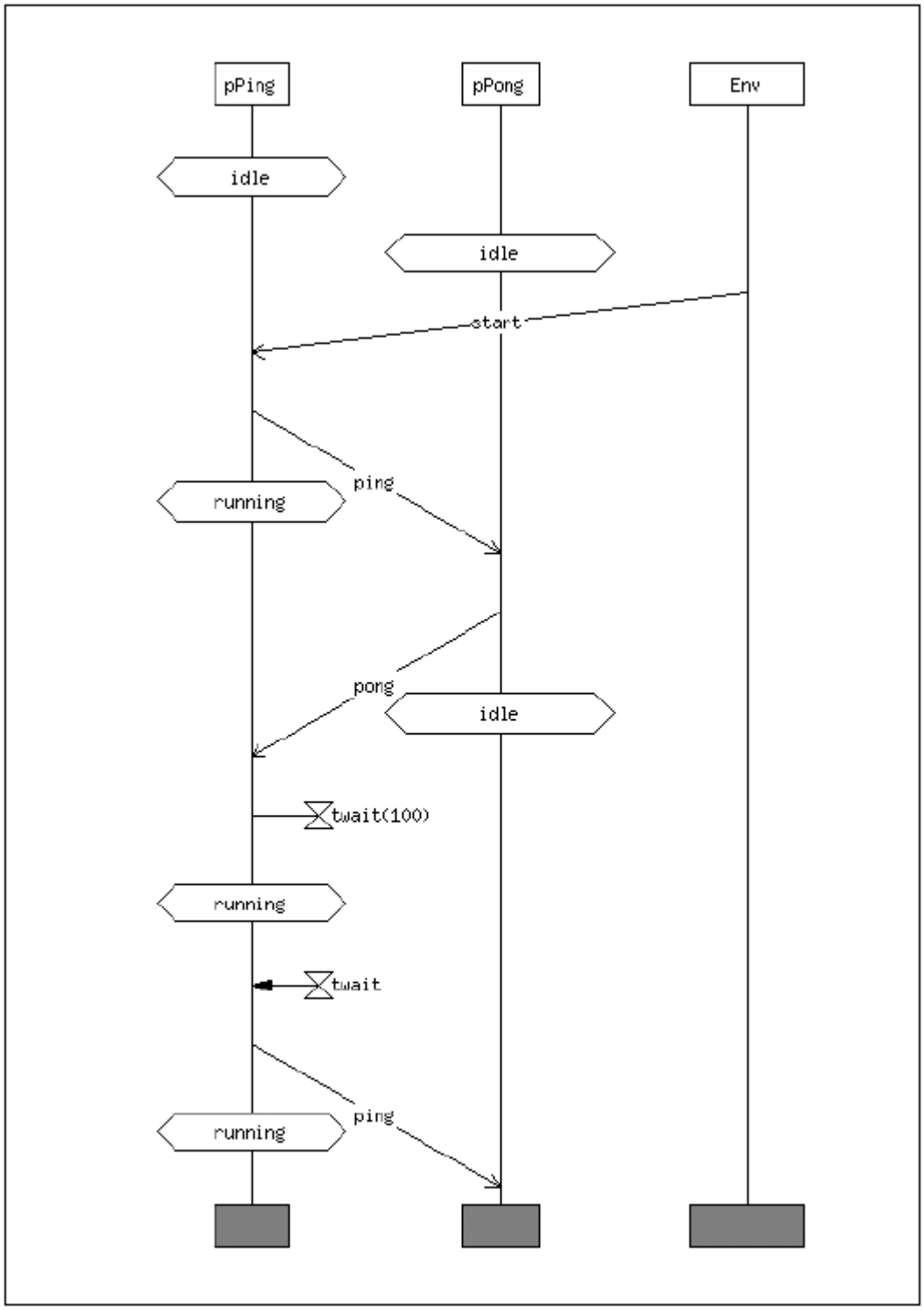
Ping pong system view



Ping process

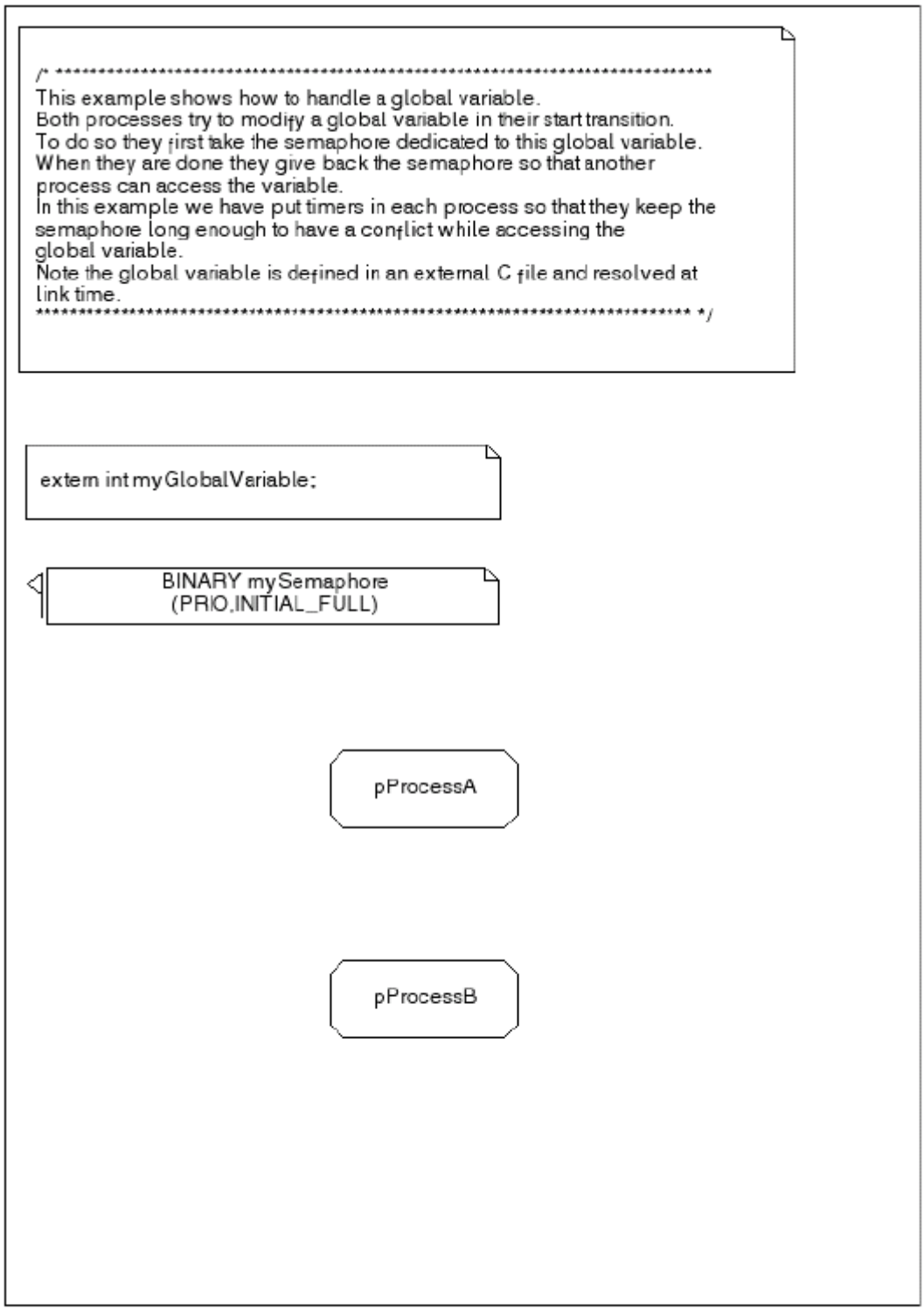


Pong process

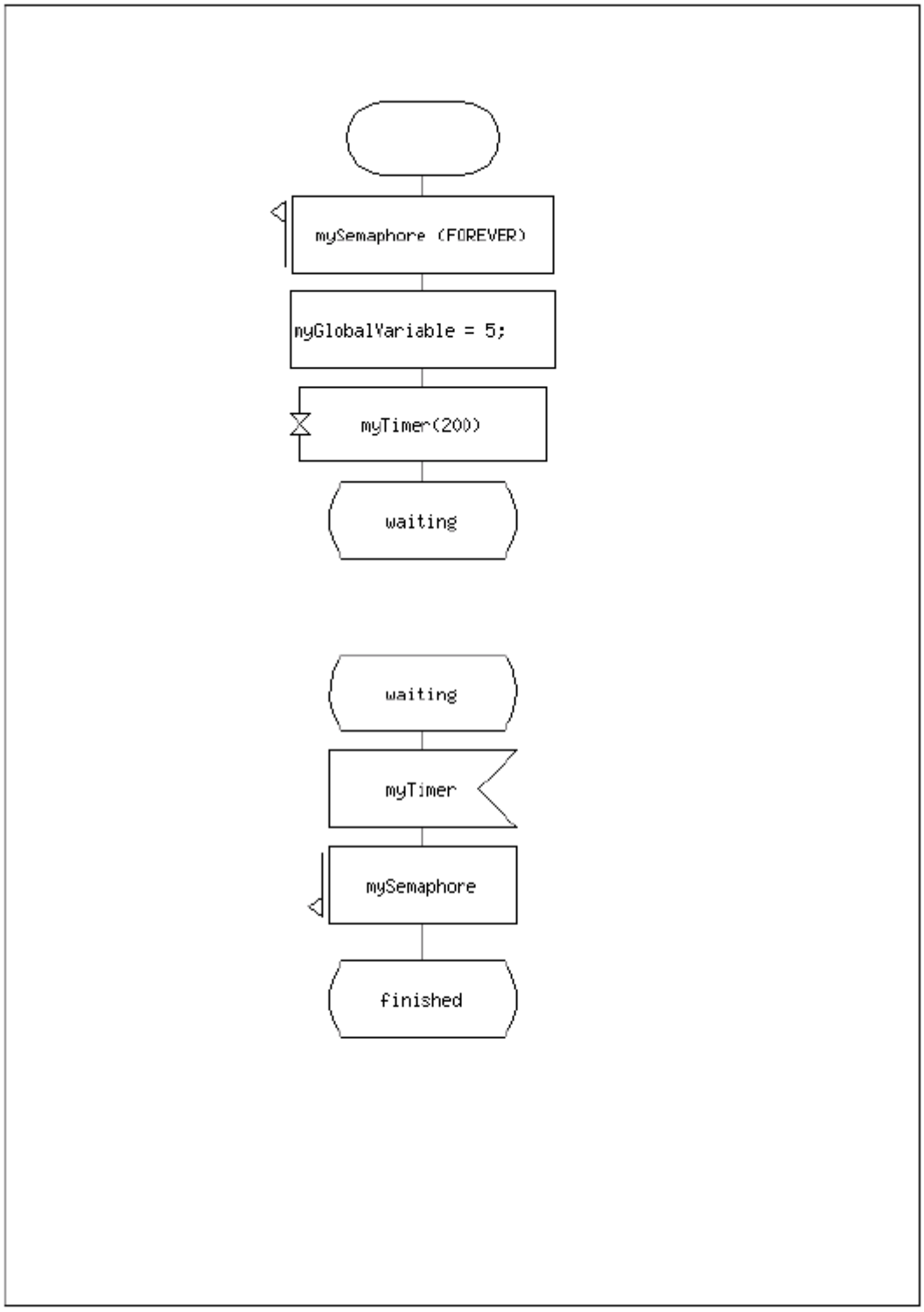


MSC trace of the ping pong system

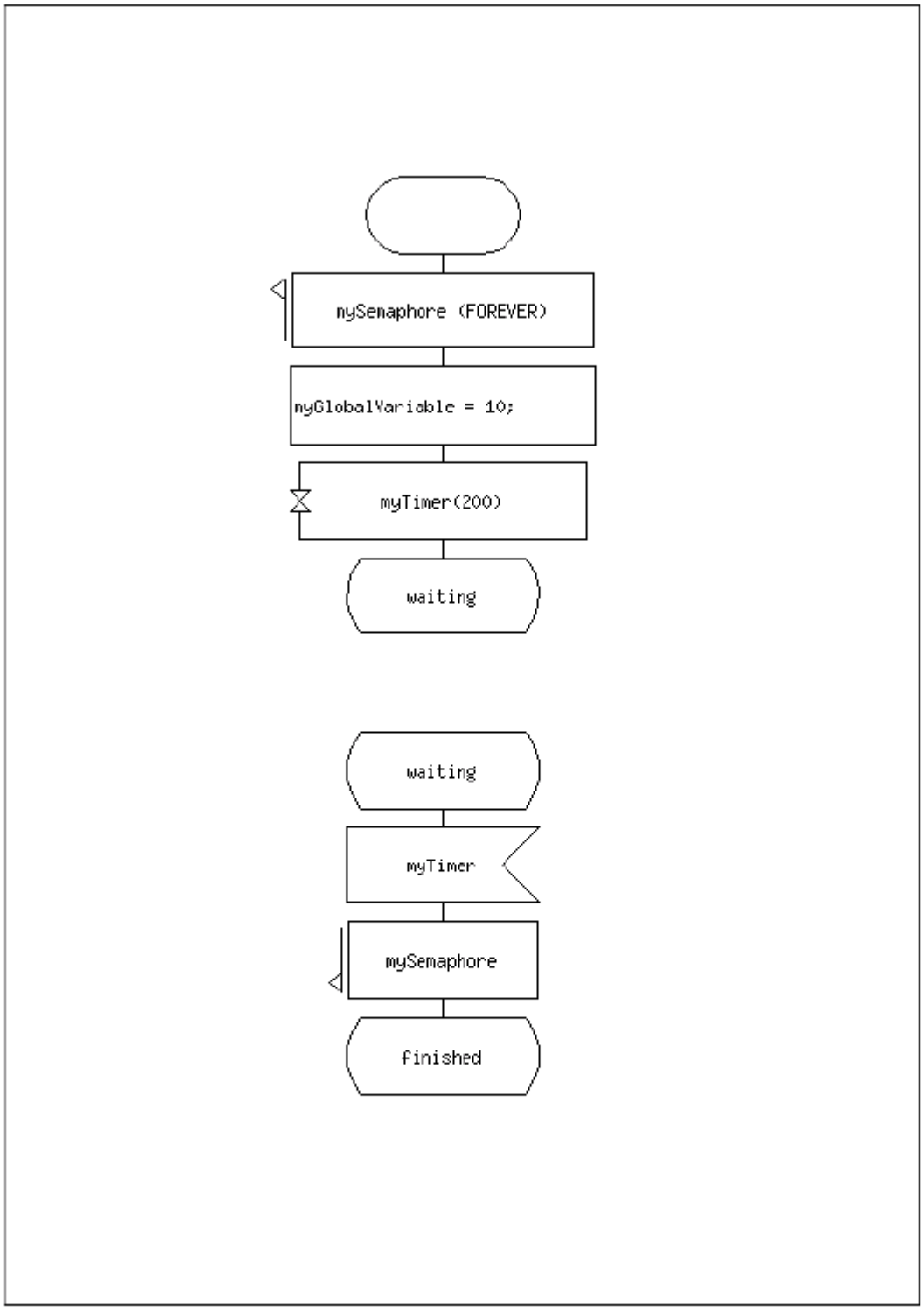
12.2 全局变量操作



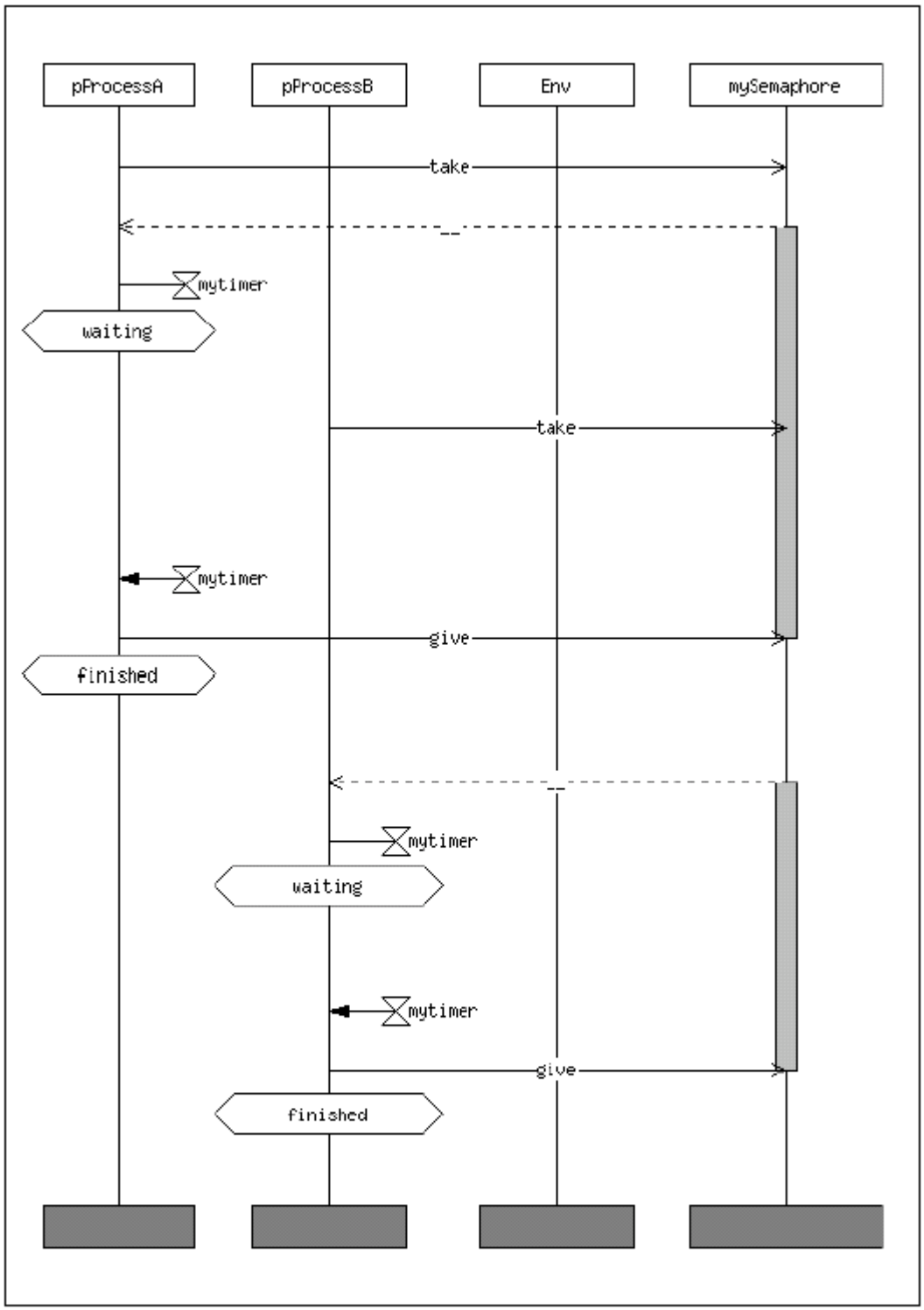
Global variable manipulation example system


mySemaphore (FOREVER)
myGlobalVariable = 5;
myTimer(200)
waiting
waiting
myTimer
mySemaphore
finished

Process A


mySemaphore (FOREVER)
myGlobalVariable = 10;
myTimer(200)
waiting
waiting
myTimer
mySemaphore
finished

Process B

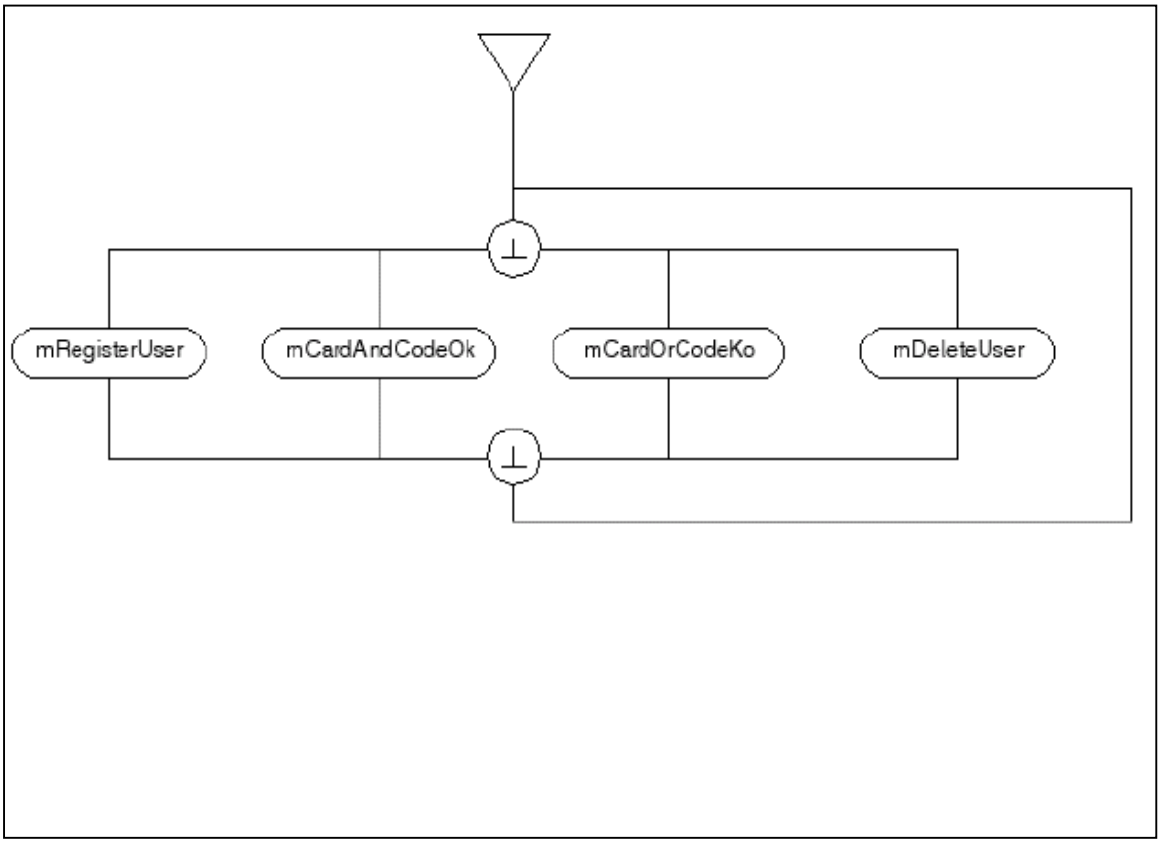


MSC trace of the global variable manipulation

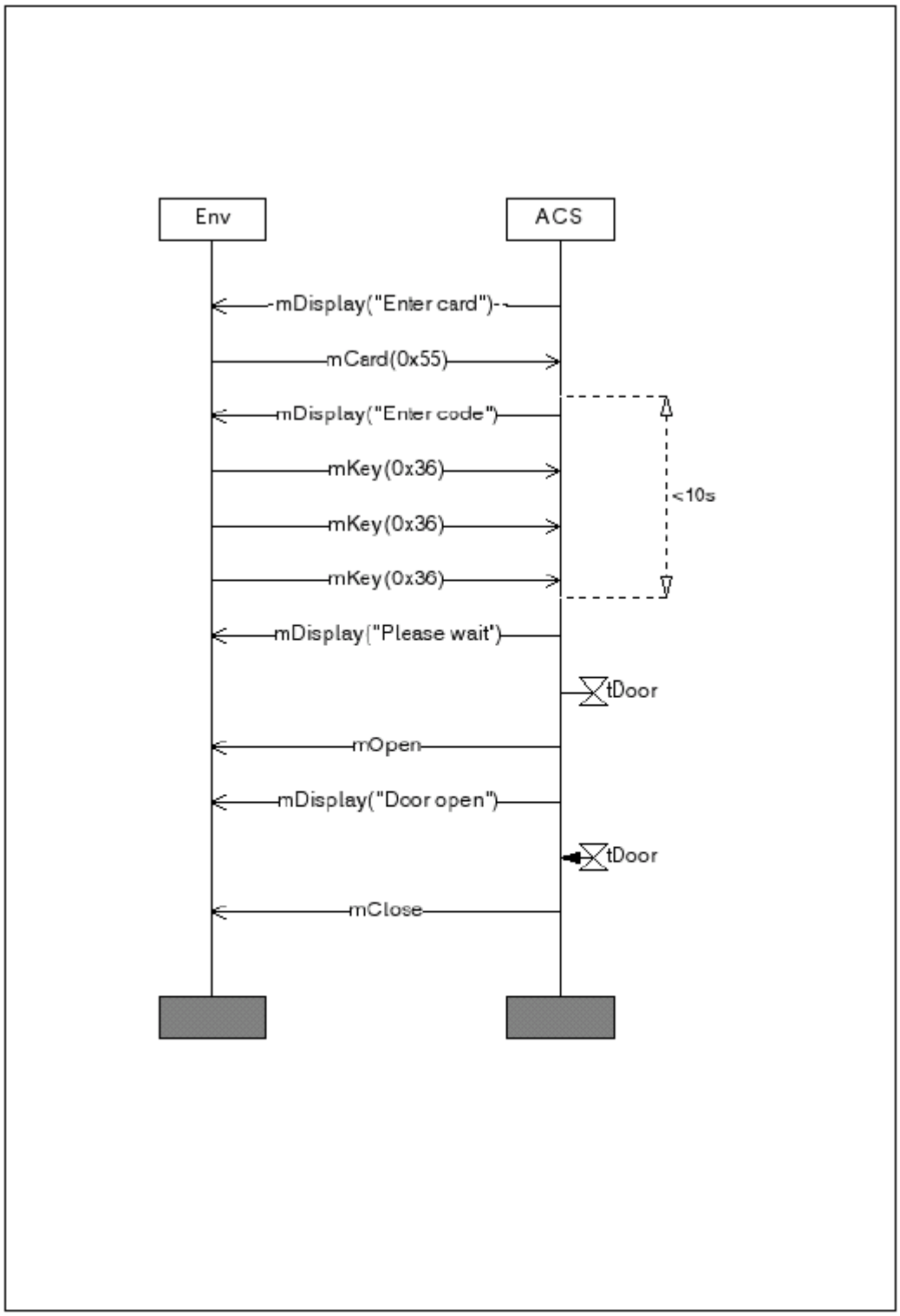
12.3 – 访问控制系统

本系统用来访问一个楼宇。进入此楼宇时，必须插入磁卡，并输入密码。central块中是数据库。系统启动时，没有用户注册，因此，第一个进入者必须是管理员。

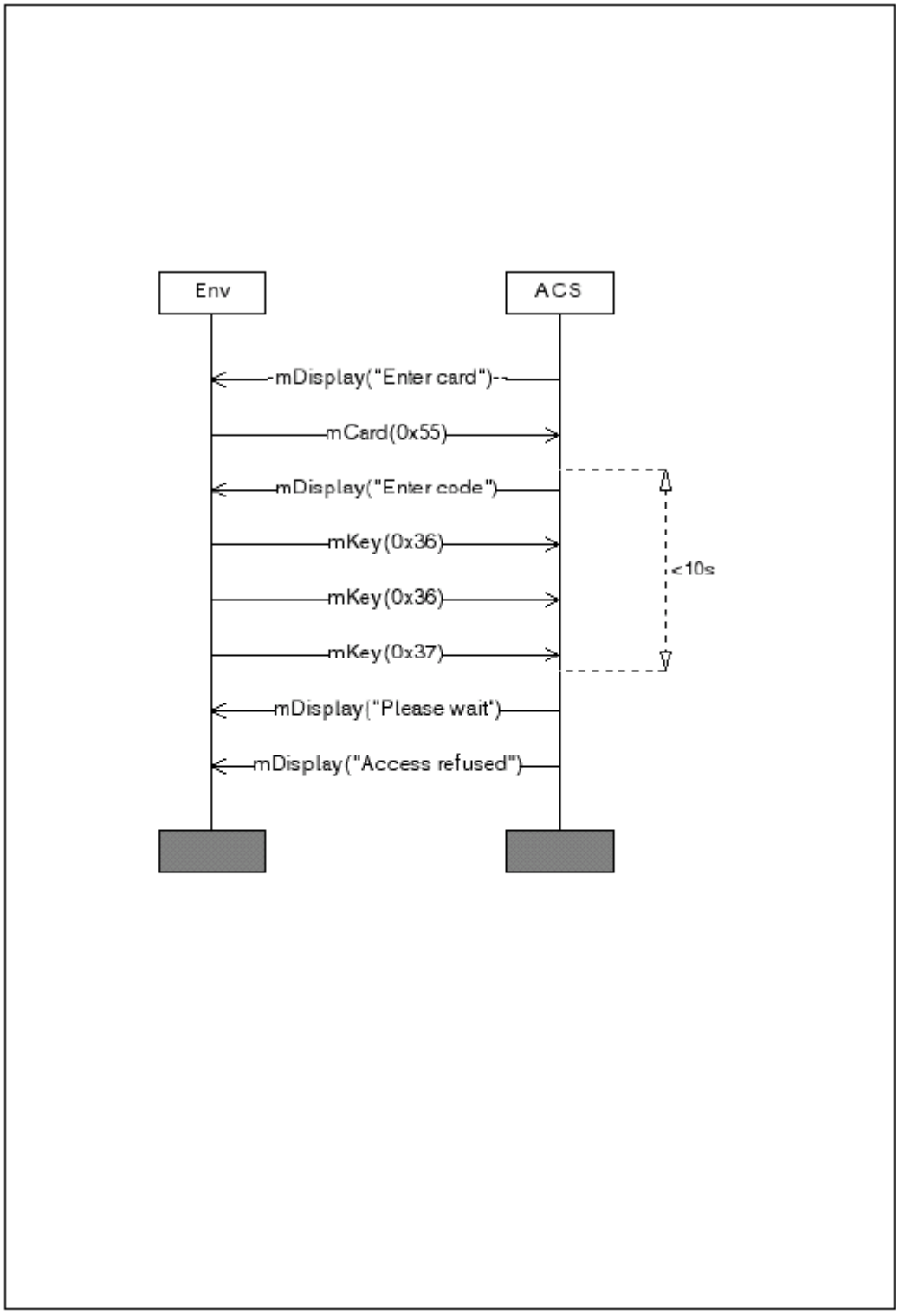
12.3.1 需求



Either one of the MSCs can be executed indefinitely

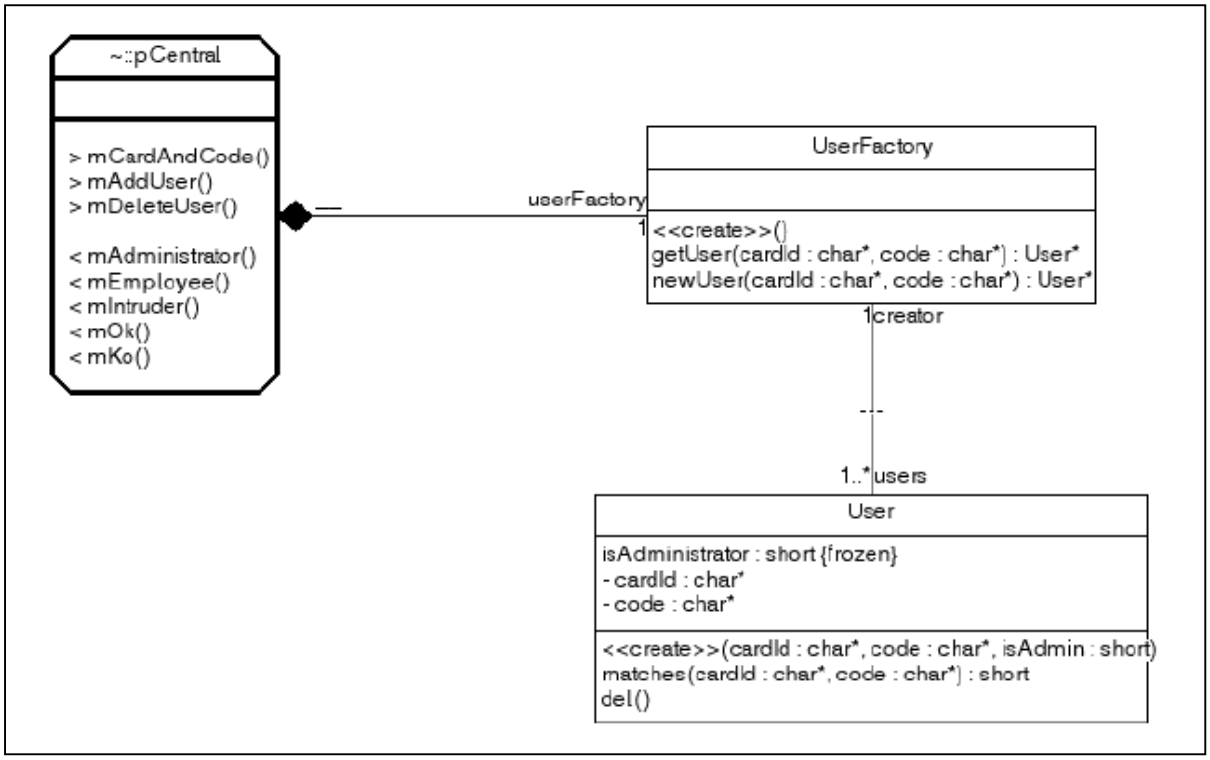


Standard scenario



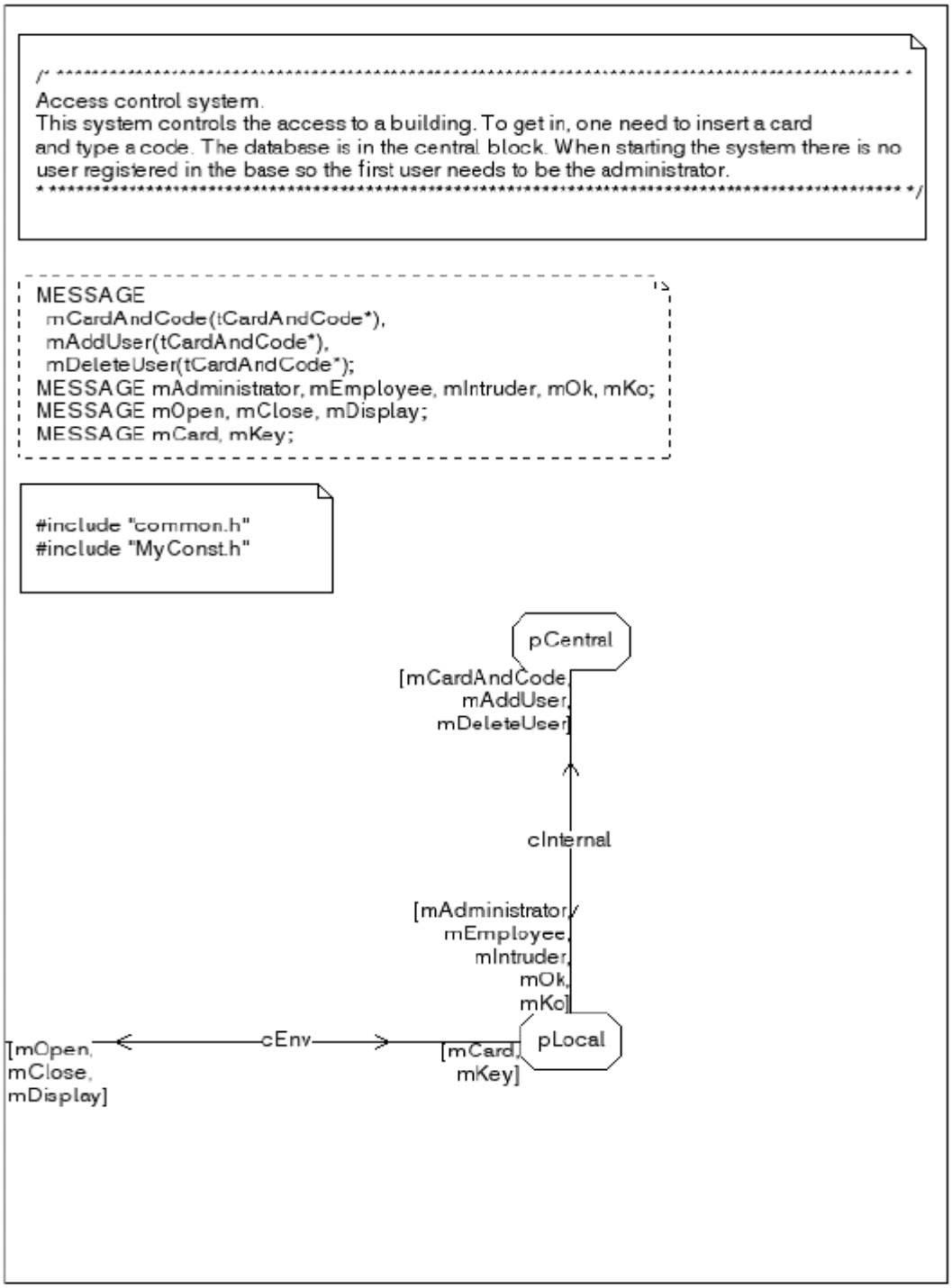
Standard refusal scenario

12.3.2 分析



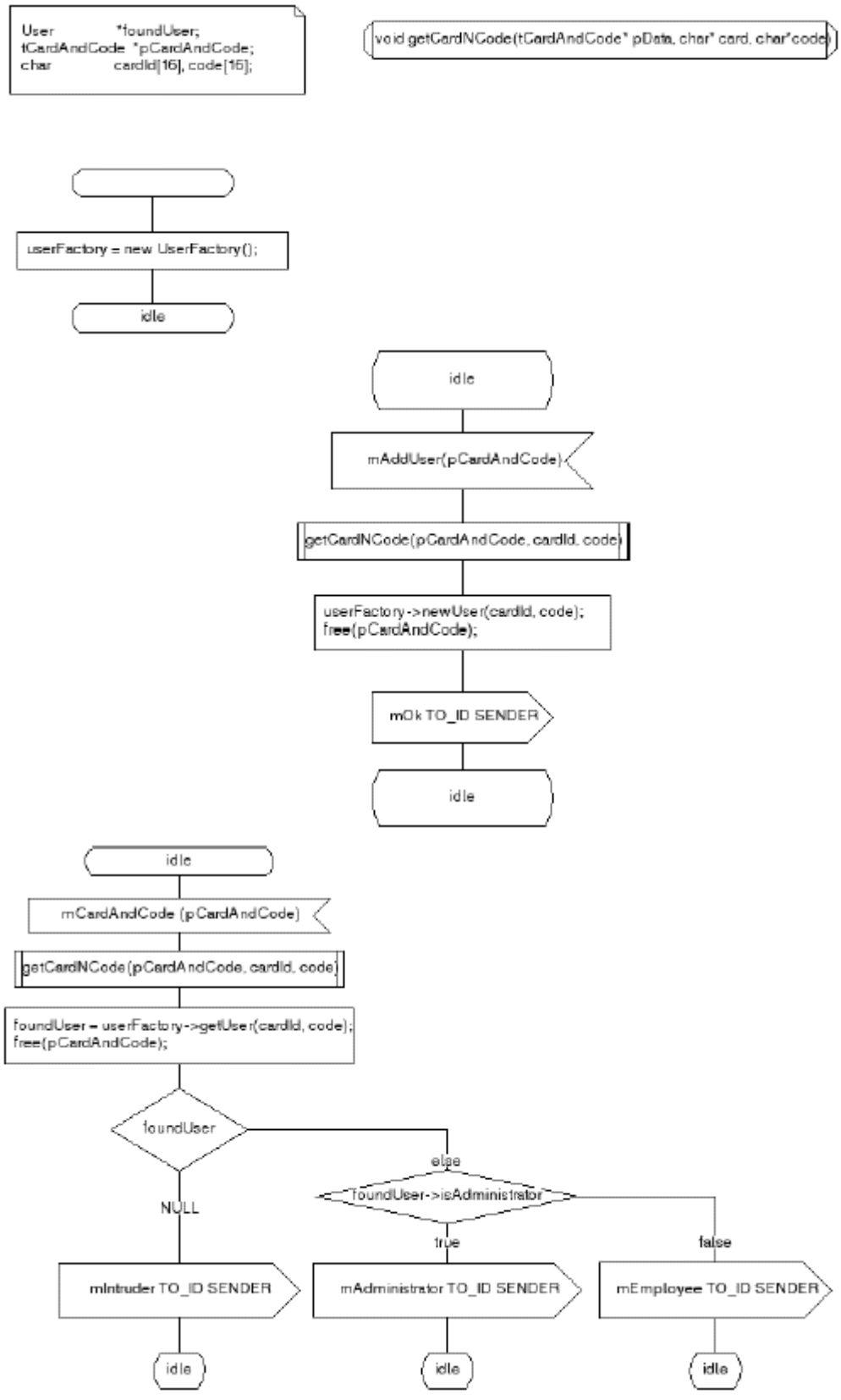
The class diagram shows the relation between pCentral (task) active class and UserFactory and User passive classes (C++)

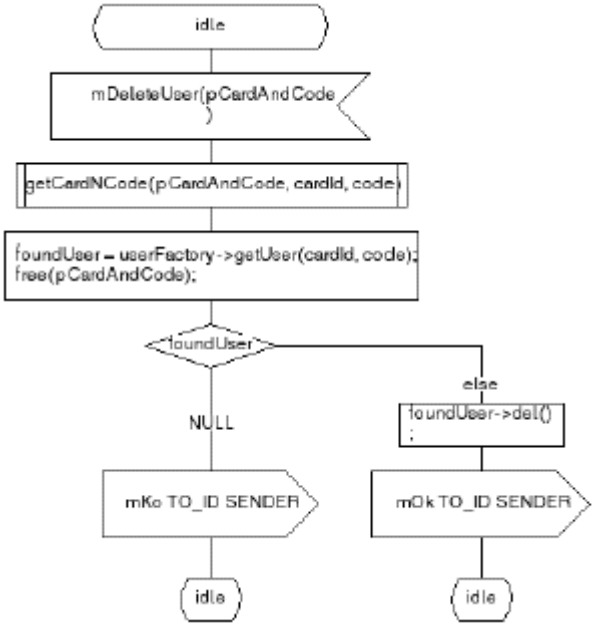
12.3.3 体系结构



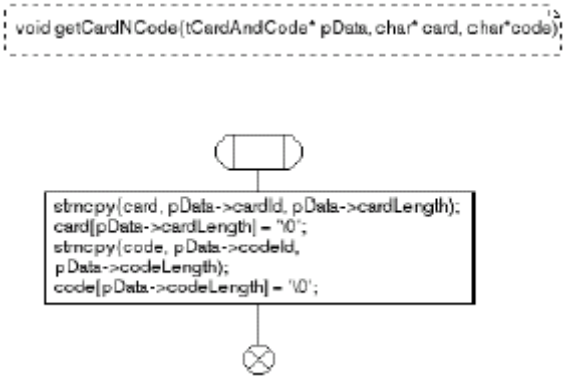
The system is made of two tasks: pCentral and pLocal

12.3.4 pCentral process

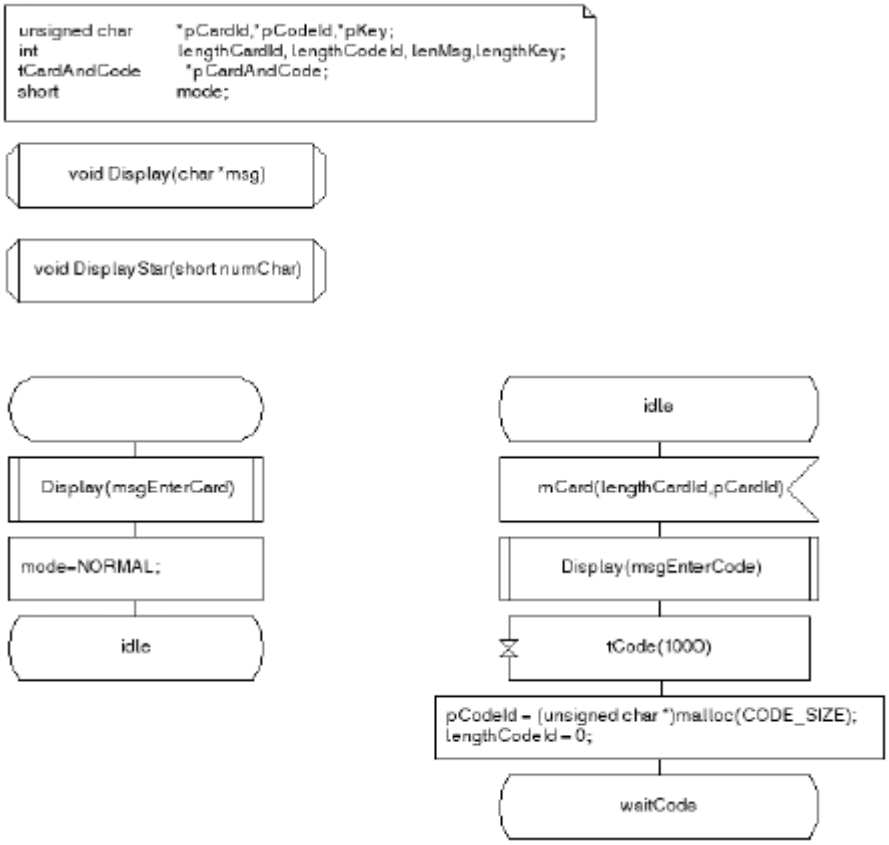


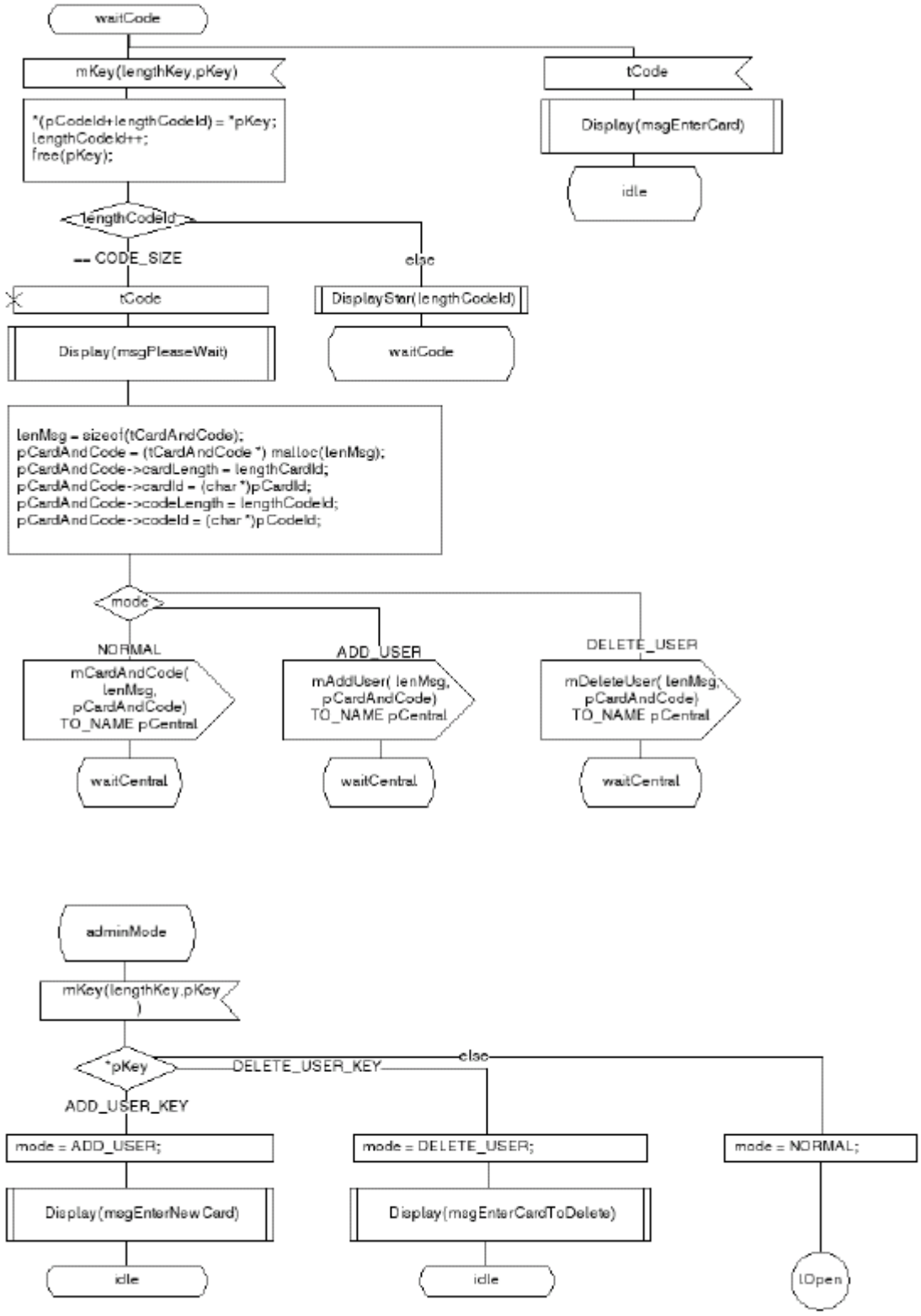


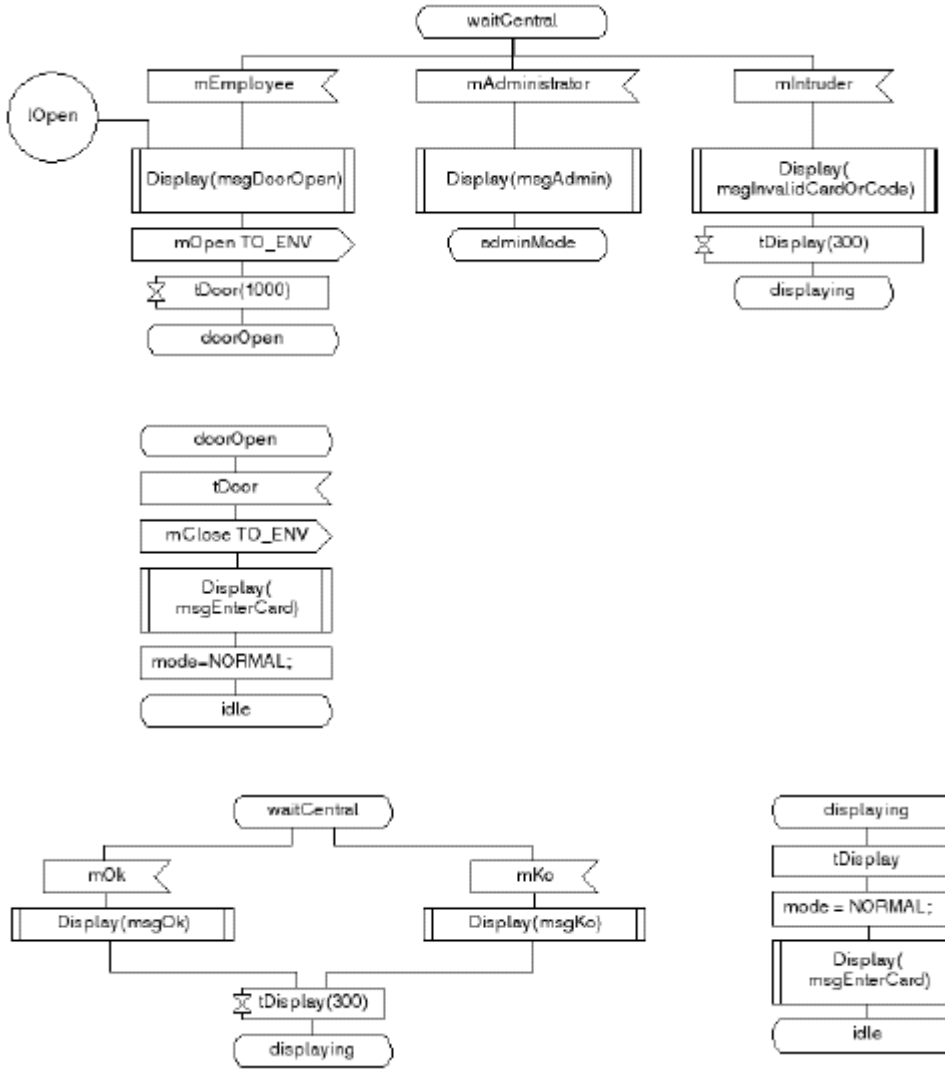
12.3.5 getCardNCode procedure



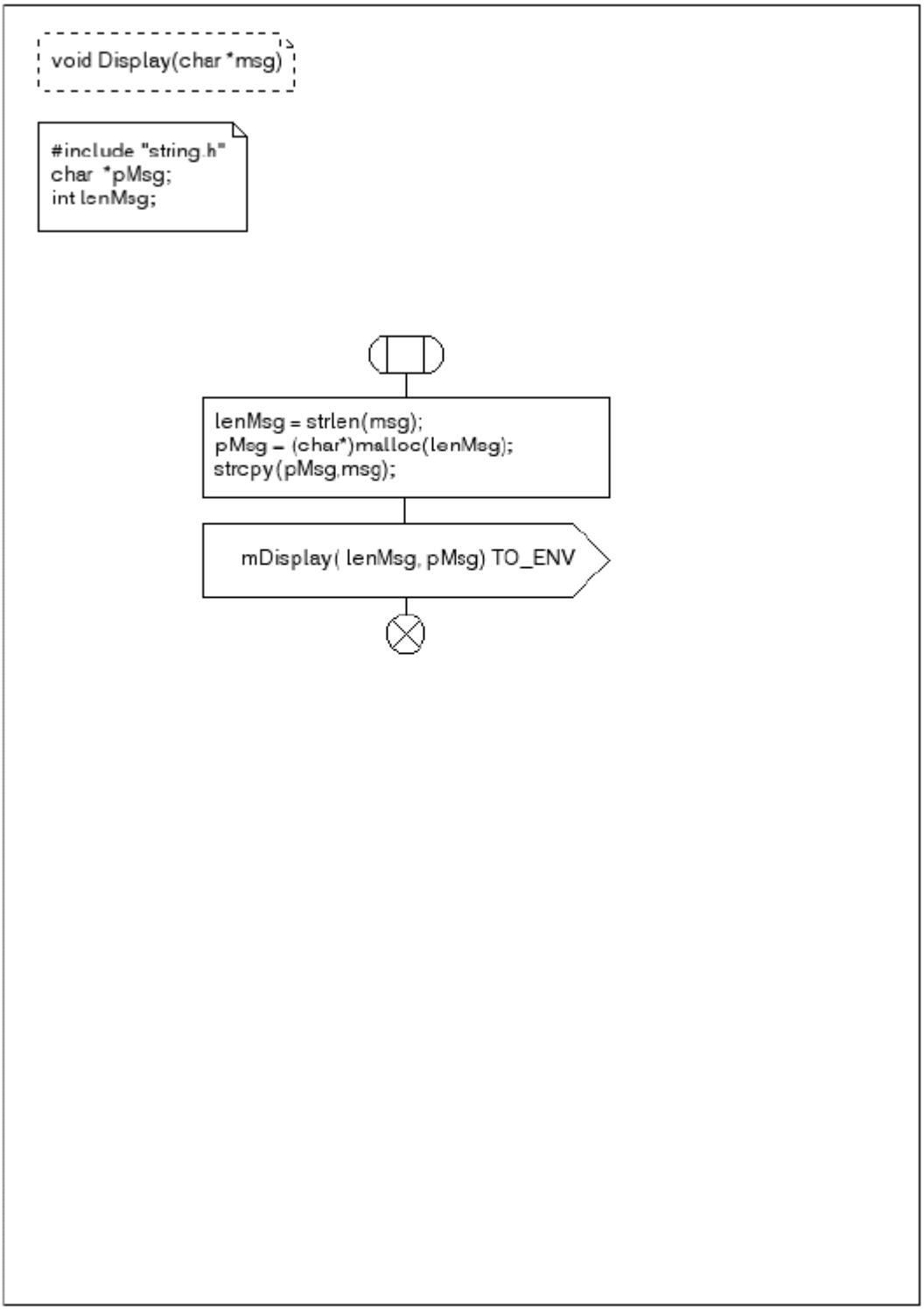
12.3.6 pLocal process



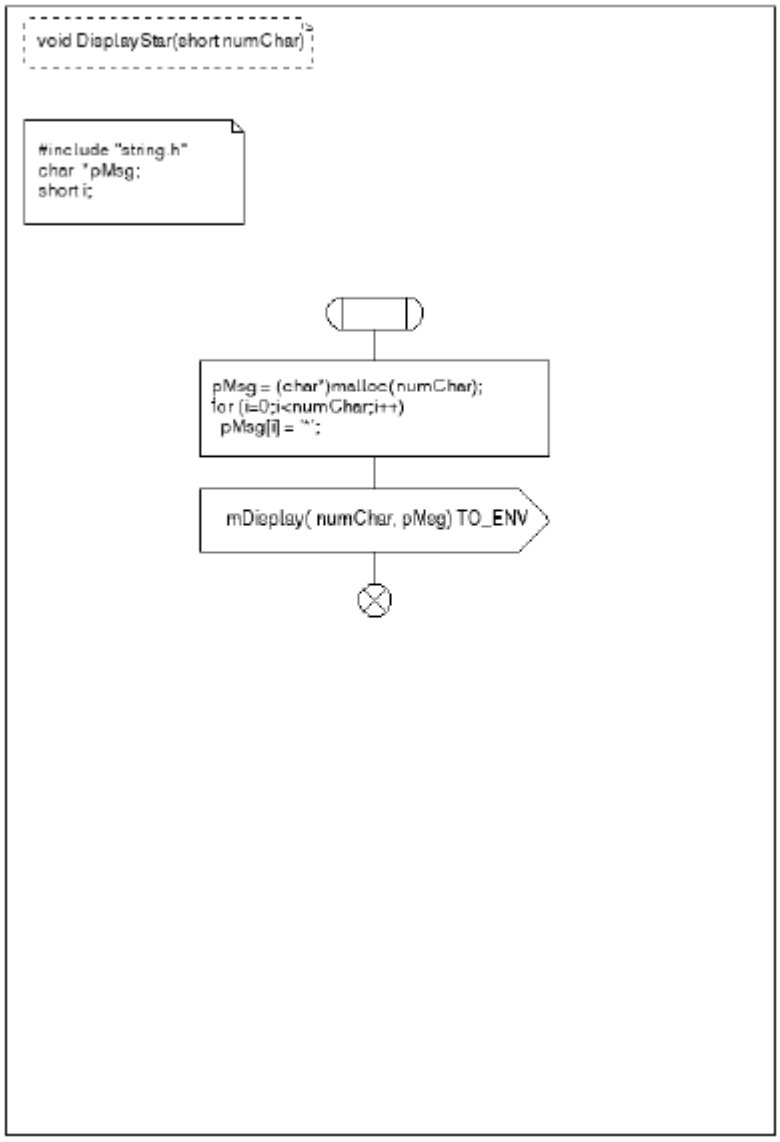

waitCode
mKey(lengthKey,pKey)
tCode
*(pCodeId+lengthCodeId) = *pKey;
lengthCodeId++;
free(pKey);
Display(msgEnterCard)
idle
lengthCodeId
== CODE_SIZE
else
tCode
DisplayStar(lengthCodeId)
Display(msgPleaseWait)
waitCode
lenMsg = sizeof(tCardAndCode);
pCardAndCode = (tCardAndCode *) malloc(lenMsg);
pCardAndCode->cardLength = lengthCardId;
pCardAndCode->cardId = (char *)pCardId;
pCardAndCode->codeLength = lengthCodeId;
pCardAndCode->codeId = (char *)pCodeId;
mode
NORMAL
ADD_USER
DELETE_USER
mCardAndCode(lenMsg, pCardAndCode) TO_NAME pCentral
mAddUser(lenMsg, pCardAndCode) TO_NAME pCentral
mDeleteUser(lenMsg, pCardAndCode) TO_NAME pCentral
waitCentral
waitCentral
waitCentral
adminMode
mKey(lengthKey,pKey)
*pKey
DELETE_USER_KEY
else
ADD_USER_KEY
mode = ADD_USER;
mode = DELETE_USER;
mode = NORMAL;
Display(msgEnterNewCard)
Display(msgEnterCardToDelete)
idle
idle
tOpen


waitCentral
mEmployee
mAdministrator
mIntruder
IOpen
Display(msgDoorOpen)
Display(msgAdmin)
Display(
msgInvalidCardOrCode)
mOpen TO_ENV
adminMode
tDisplay(300)
tDoor(1000)
displaying
doorOpen
doorOpen
tDoor
mClose TO_ENV
Display(
msgEnterCard)
mode=NORMAL;
idle
waitCentral
mOk
mKo
Display(msgOk)
Display(msgKo)
tDisplay(300)
displaying
displaying
tDisplay
mode = NORMAL;
Display(
msgEnterCard)
idle

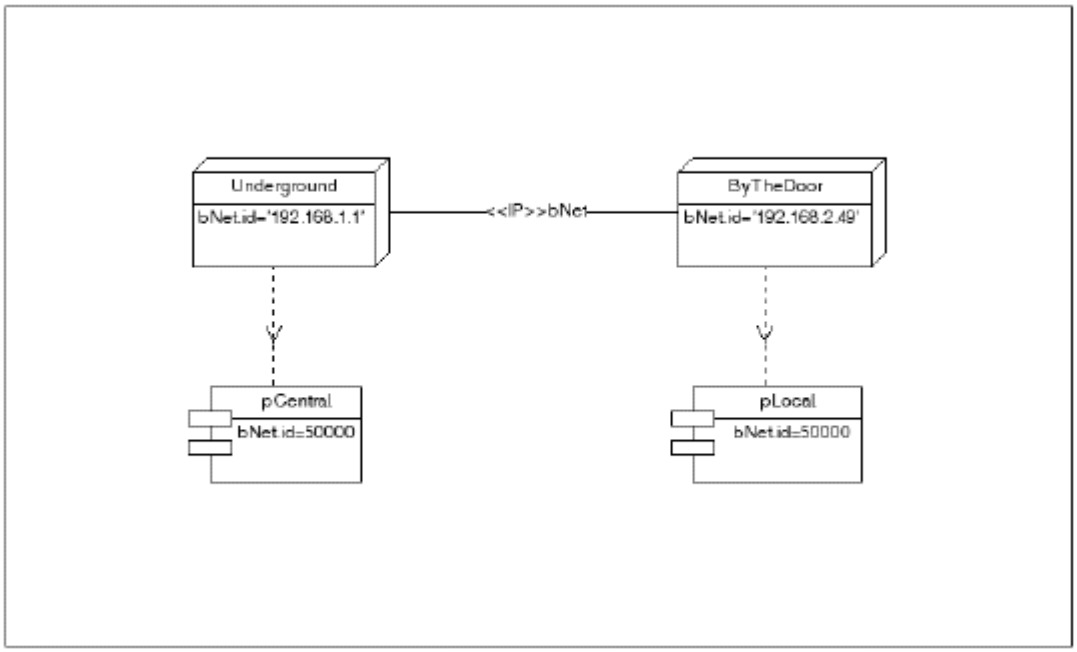
12.3.7 Display procedure



12.3.8 DisplayStar procedure



12.3.9 部署



The components communicate through IP

13. 与传统 SDL 的差异

很难列出SD L-RT和SDL之间的所有差异，即使SDL开发者能很好的理解SDL-RT，反之依然。但是，仍然需要清晰的阐述其差异，下面的段落指出主要的差异。

13.1 – 数据类型

这是SDL和SDL-RT主要的差别。传统SDL具有自己的数据类型和语法，SDL-RT基本上用ANSI C语言。某些符号有特殊的语法，因为C没有相等价的指令，如，输出、输入、存储、或信号灯操作。

其先进性是显而易见的：

- 这些语法对所有的实时开发者来讲众所周知的，
- 隐含的引入了指针的概念，这在SDL中是没有的，
- 非常容易与已有的代码集成，这在传统的SDL中是非常琐碎的，
- 最后，也是最起码的，SDL-RT使代码生成非常直接。

13.2 – 信号灯

信号灯在实时系统中是一个重要的概念，传统SDL忽略了此概念。信号灯符号已经引入了SDL-RT中，回答了实时开发者必须面对的问题。

13.3 – 输入

传统的SDL处理消息交换时，有很好的概念。但是，这些概念对于实时开发者并没有好处，而且在实际的目标机或操作系统上实现时，相当琐碎。这就是为何SDL-RT删除了下面的概念：

当接收一个消息、内部消息、两层优先权消息时，需要使能(enable)条件。

13.4 – 命名

传统的SDL对某些众所周知的概念，如，“ 信号（signal） ”，使用外来的名字，它基本上与“ 消息（message） ”是相关的。由于“ 消息 ”是实时操作系统中普遍使用的，因此 ，SDL-RT也用同一个术语。

谈到面向对象时，传统的SDL讲“ 类型（type） ”，而不是通常的术语“ 类（class） ”。SDL-RT用开发者普遍知道的词“ 类（Class） ”。

13.5 – 面向对象

当涉及到面向对象的概念时，传统的SDL用 "virtual"、"redefined"、和"finalized"。例如，一个超类应当指定的转移是"virtual"，子类允许"redefine" 或 "finalize"。这是C++的风格，

但是，当仅仅写出而没有起到更清晰的目的时，实际上会很厌烦的。当不需要在子类中重新定义任何事情时，SDL-RT用Java的表示方法。

14. 内存管理

实时系统必须交换信息。最好的方法是有一个保留的共享内存，几个任务都可以访问。SDL-RT隐含的运行在这样的基本体系结构上，因为，它支持全局变量和通过指针的交换消息参数。这样引发需要遵循的内存管理的规则，以便保证合理的设计。

14.1 – 全局变量

SDL-RT进程可以共享全局变量。这是非常有用的，但也非常危险，因为，如果没有注意操作，数据会崩溃。强烈的建议用信号灯访问全局变量，保证数据的一致性。在本文档有这种设计的一个例子。

14.2 – 消息参数

消息的参数通过指针传递。这意味着发送进程指向的数据被接收进程访问。良好的设计要遵循下面的规则：

- 发送进程分配特定的内存区域来存储参数，
- 一旦消息被发送，参数内存区域再也不能由发送进程操作，
- 接收进程负责释放包含消息参数的内存。

15. 关键字

下面的关键字在某些SDL-RT符号中具有意义：

关键字	涉及到的符号
PRIO	任务定义 任务创立 连续信号
TO_NAME TO_ID TO_ENV VIA	消息输出
FOREVER NO_WAIT	信号灯操作
>, <, >=, <=, !=, == true, false, else	判断分支
USE MESSAGE MESSAGE_LIST STACK	附加头部符号

表2：符号中的关键字

16. 语法

所有SDL-RT名字必须是字母字符、数字字符、和下划线的组合。不允许其它符号出现。例如：

```
myProcessName
my_procedure_name
block_1
_semaphoreName
```

17. 命名约定

由于某些SDL-RT概念可以顾名思义（进程（process）、信号灯（semaphore）），每个名字在系统中必须是唯一的。这样使得设计更加合理，也容易使工具对SDL-RT进行支持。

建议使用下面约定名：

- 块（block）名用’b’开头，
- 进程（process）名用’p’开头，
- 计时器（timer）名用’t’开头，
- 信号灯（semaphore）名用’s’开头，
- 全局变量名用’g’开头。

18. 词法规则

本文档中使用BNF（巴克斯范式）子集：
<传统英语表达>，…（翻译稿中用汉语—译者）
<stuff>] stuff 是可选项
{<stuff>}+ stuff 至少使用一次或多次使用
{<stuff>}* stuff 使用0次或多次。

19. 词汇

ANSI	American National Standards Institute	美国国家标准化所
BNF	Backus-Naur Form	巴克斯范式
ITU	International Telecommunication Union	国际电信联盟
MSC	Message Sequence Chart	消息顺序图
OMG	Object Management Group	对象管理组
RTOS	Real Time Operating System	实时操作系统
SDL	Specification and Description Language	规格说明和描述语言
SDL-RT	Specification and Description Language - Real Time	规格说明和描述语言—实时
UML	Unified Modeling Language	统一模型语言
XML	eXtensible Markup Language	扩展标记语言

20. 对先前释放的修改

20.1 – 信号灯操作

20.1.1 V1.0 到 V1.1

信号灯返回一个值，指明时间超时或成功。
当信号灯不能用时，信号灯生命线变为灰色。

20.2 – 面向对象

20.2.1 V1.1 到 V1.2

面向对象一章中有错，不可能在块类的定义框图中声明进程类或块类。

20.2.2 V1.2 到 V2.0

- 引入 UML
- 引入 UML 部署图
- 面向对象符号引入到行为图中。

20.3 – 消息

20.3.1 V1.1 到 V1.2

- 消息必须被声明。
- 消息参数可以是C类型。
- 参数长度可以忽略，如果参数是结构型的。其长度隐含的是参数类型的sizeof。
- VIA 概念已经引入。

20.4 - MSC

20.4.1 V1.1 到 V1.2

- 引入存储消息的表示。

20.5 – 任务

20.5.1 V1.2 到 V2.0

在创立任务时，增加STACK 参数作为一个参数。

20.6 – 组织

20.6.1 V1.2 到 V2.0

章节被重新组织。

21. 索引

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X