XMOS 装置上之 XC 程序设计语言

Douglas Watt

XMOS®

XMOS 装置上之 XC 程序设计语言

Douglas Watt 着

林琬珍 译

XMOS®

本书的编排，作者是使用 LaTeX 以及手动编排的方式完成排版，其中使用的字体为 HeiS ASC、Song ASC、Minion Pro、Latin Modern 以及 Computer Modern。

XMOS 也以电子格式出版其书籍，一些印制版的内容并没有在电子书中出现。

关于 XMOS 产品的资讯，浏览我们的网页：www.xmos.com。

因为网路的动态本质，任何此书中包含的网址或是连结可能于出版后变更，或者不再有效。

CPI Antony Rowe，United Kingdom 印刷装订

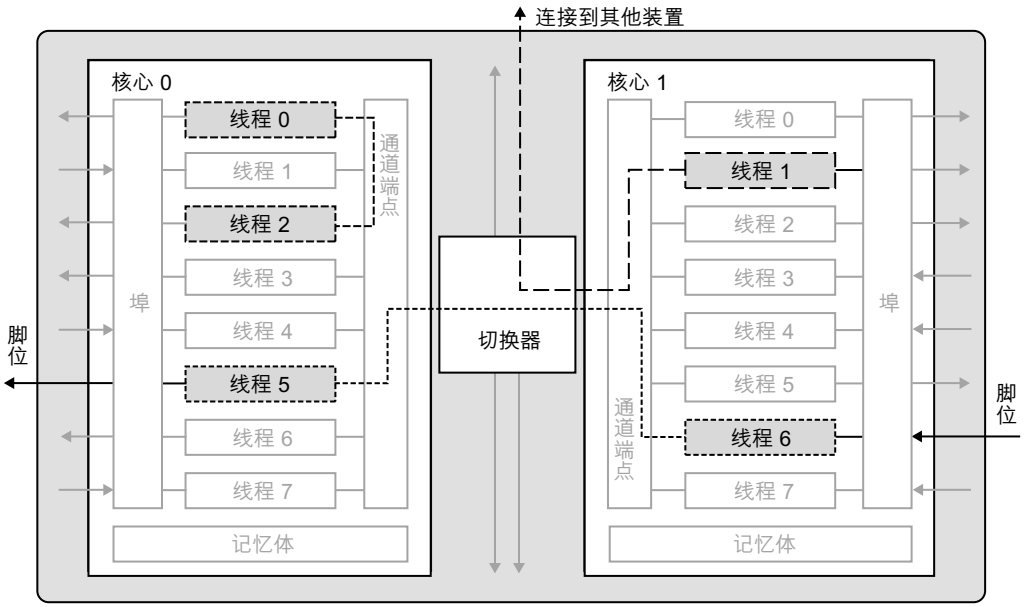
ISBN: 978-1-907361-32-6

ISBN: 978-1-907361-03-6 (原文版)

XMOS Limited 出版

XMOS 前言

XMOS 晶片架构能使介面、数位信号处理以及控制函数的组合操作得以软件方式实现，一个 XMOS 装置是由一个或是多个 XCore 组成，每一个皆包括事件驱动多线程处理器，并且有高度整合输入/输出以及晶片内存记忆体来达到实时效能为目的的架构，每个处理器都由硬件支援并行运行多个线程的功能，并且有特定的指令控制输入和输出。



这是决定性的架构，每个线程皆被保证可以获得一部分的处理效能；线程可以用来运行计算，处理实时的输入/输出操作以及多个事件的回应；利用单一的指令，可以使这些输入/输出脚位进行取样或是驱动，而数据传送速度则可以用计时器或是时序控制。高

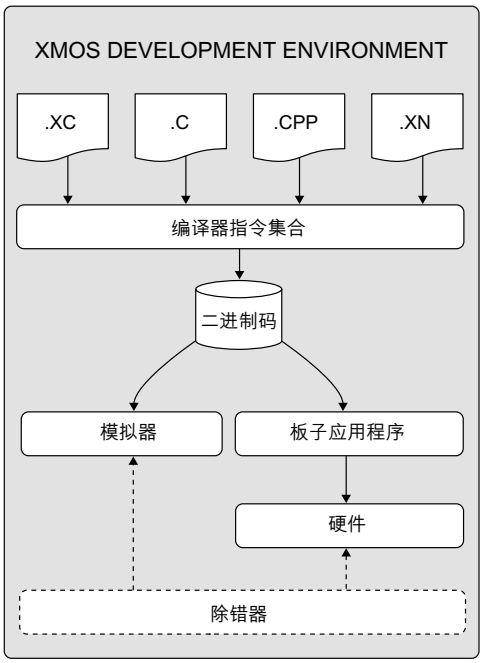
效能的切换器使处理器间的沟通以及使多个装置的系统建构变得简单；在同一个处理器上的线程间沟通是没有时延，而处理器之间，时延对已知的通讯模式是明确地呈现，硬件完整的描述在另外文件[1]中说明。

程序设计模式

程序可以使用 XC、C 以及 C++ 合并编写，XC 具有 C 延伸的特性，可以简化并发运行、输入/输出以及时间的控制，这些延伸出的特性直接对应到 XCore 硬件资源，例如线程、通道以及埠，这可避免大量调用库函数。XC 的构造是有**效率**的 — 编译成简短指令序列，也是**安全**的 — 不受到可能的死锁、竞速现象以及记忆体侵犯这些问题的威胁，这使得程序容易撰写和理解以及除错，本书对于 XC 程序语言有详细完整的介绍。

工具架构

XMOS 工具是基于标准的嵌入式开发流程以及业界标准平台上发展的，工具的使用因此变得更直觉简单。



在开发的所有阶段，工具都支援语言本身以及架构上并发和实时的能力:

- 边译器会静态地分析所产生的二进制文件，以提供在源码中所给定的时间资讯，这使得程序在不同时间特性的多个装置上，更具可携性。
- 编译器工具会产生单一的二进制的文件，文件中包含所有装置的指令以及数据区段，模拟器和板子工具可以对这个文件进行操作，对程序设计人员而言，则省去复杂性。
- 目的平台的资讯置放在 XN 文件中，其中包含 XMOS 装置的网路、SPI 闪存记忆体、振荡器和 JTAG，XN 文件使工具能够完全地自动化系统启动和设定。
- 板子工具可以将系统设定由特定方式启动，由开发主机电脑启动、板子上闪存记忆体或是晶片内 OTP 记忆体启动。
- 除错器是开发主机与目的平台上所有处理器的介面，对程序设计人员而言，它呈现一组线程，并且可以一起检阅，使除错 XMOS 装置和除错一般处理器的方式相同。

工具的使用说明在另外的使用手册[2]中。

如何阅读

这本书的主要章节便是一份如何在 XMOS 架构上，以 XC 程序设计语言的使用手册，附录部分有完整的 XC 规格以及在 XMOS 第一代装置 XS1 上 XC 的实作细节，使这份文件更趋完美；这份使用手册假设读者有一些程序设计经验。

第 1 章 概述 XC 支援的计算，包含算术表达式、控制流程建构以及函数。

第 2 章 解释如何在埠上输入和输出数据以及和外部元件连接的介面，这个章节也说明如何使用计时器控制输入/输出数据传输率及使用 `select` 语句与多个元件衔接。

第 3 章 显示如何运行并发的多任务，也就是分别的线程，并说明如何使用通道在不同的线程间通讯。

第 4 章 描述如何同步输入/输出操作与时序，以及如何纪录与控制在哪个边缘输入和输出操作会有作用。

第 5 章 示范使用埠的缓冲区将运算需要的输入/输出操作去藕，以改善效能。

第 6 章 解释如何将数据串行至输入/输出脚位，并且说明如何解译，以及如何使用埠内建的功能产生信号撷取信号。

附录 A 阐释标准正式的 XC 程序语言规格。

附录 B 提供正式的 XC 埠的输入/输出语义。

附录 C 说明使用 XC 语言的 XS1 实作，包含不同的输入/输出操作、埠和脚位的对应以及 XC 数据类型的大小以及对齐。

XC 的运算框架和 C 语言相似，特别是其类型系统以及控制流程的构成；因此，有经验的 C 语言程序设计人员可能希望先阅读章节 2 — 6，其中涵盖输入/输出和并发性的建构；这些章节的编排，前面章节介绍的概念会用于后面的章节，因此，应该依照其编排顺序阅读。

XMOS 架构的发展以及 XC 程序语言会持续演进，目前的实作于附录 C 中讨论，也视为将来发展方向以及技术的标准化。

致谢

关于埠的使用范例，多数原本是由 Henk Muller 为 XS1 上埠的使用手册[3]所撰写，LCD 驱动程序的范例研讨以及以太网控制器则是由他和 Larry Snizek 共同开发；Richard Osborne 则是 XS1 标准程序库文件的作者，另外，Huw Geddes 费心地建立所有用到的波形图以及图表。

Ali Dixon、Peter Hedinger、Russell Gallop 和许多其他的公司成员对这份文件原稿提供最仔细的校阅，他们的意见，校正以及建议使最后的文字获得重要的改善。

目录

第 1 章

基本运算

XC 是以 C 语言为基础，并含有运算框架的指令式程序语言，XC 程序是由包含语句的函数组成，语句会依据储存在变量上的数值而动作；控制流程语句表达决定，而循环语句表示反覆运行。

下面的章节中，XC 新诠释的结构或是和 C 语言不同点会在页面边界注明。

XC 新增

1.1 Hello, World!

学习一个新的程序语言通常第一个任务是打印出“Hello, world!”字符串，相对的 XC 程序如下：

```
#include <stdio.h>

main(void) {
  printf("Hello, world!\n");
}
```

程序的第一行会告诉编译器从头文件 stdio.h 引入需要的资讯，该头文件包含 printf 函数的声明，这个函数会输出一个字符串到标准输出，如开发系统中的终端机视窗便是个标准输出装置。

每个程序必须包含一个 main 函数，这个函数就是程序开始运行的地方；在此例中，main 被定义为没有引数的函数，以关键字 void 表示。

函数的主体是由花括号 { } 所包住的，并且它包含所要进行操作的语句，在这个例子中， main 含了单一个函数，也就是调用 printf 函数的语句，而字符串文字为调用时的引数，跳脱字符 \n 表示一个换行字符。

1.2 变量、常量和表达式

变量表示数据在记忆体上储存的位置，所有的变量必须在使用和给值之前声明，最常见的算术类型为 char 和 int，char 是一个八个位元长的位元组整数，而 int 为 32 位元的整数型态。

声明:

```
char c;
```

c 被声明为一个 8 位元的带号字符，可以表示 -128 到 127 之间的整数值。

限定符 signed 或是 unsigned 用来陈述类型的带号性，声明如下所示:

```
unsigned char c;
```

声明 c 为一个八位元的无号字符，可以表示 0 到 255 之间的整数值。

变量可以给定初始值，声明方式如下所示:

```
int i = 0, j = 1;
```

i 和 j 被声明为整数类型，并且分别被初始化为 0 和 1。

限定符 const 可以用在任何变量声明，预防变量的值在初始化之后改变，声明如下所示:

```
const int MHz = 1000000;
```

声明 MHz 是值为 1000000 的整数常量，在这之后任何企图去修改的操作皆无效。

一个或多个相同类型的变量可以结合起来形成一个**数组**，声明方式如下所示:

```
int data[3] = {1, 2, 3};
```

以上的代码将 data 声明为含有三个整数的数组，并且初始化为 1 、 2 和 3 ；数组的下标由 0 开始，因此数组的元素为 data[0]、data[1] 和 data[2]，下标可以是任何表示数组中有效元素位置的整数。

数组也可以由一个或多个的数组组成一个**多维数组**，声明方式如下:

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

matrix 被声明为二维数组，第一维度表示列，第二维度表示行，形成矩阵 matrix ，如下所示:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

下标顺序是由最高的维度先开始，例如： matrix[0][1] 的值为 2。

1.2.1 常量

常量是一个文字形式的数值且有其数据类型，在下表栏位中的例子皆为常量:

文字表示	类型	数值
123	int	123
123u	unsigned int	123
0b10000	int	16
020	int	16
0x10	int	16
0xAu	unsigned int	10
'x'	char	120
'0'	char	48
'\n'	char	10 (换行字符)
'\\'	char	92 (反斜线)
'\0'	char	0 (字符结束字符)
"str"	char 数组	's', 't', 'r', '\0'

数字序列默认是整数 (int)，无号的常量是由字尾 u 指定，二进制表示的整数常量是用 0b 为字首，八进制是用 0，而十六进制则是 0x。

字符常量通常写在两个单引号中，其值便是字符的数值，一些无法表示的字符是用反斜线表示的**跳脱字符**。

文字串是写在两个双引号间，由零个或是多个字符组成的序列，文字串内部的表示包含一个空字符字尾 \0，这个字符允许程序能找到字符串的结尾，也使字符串储存空间的需求增加一个位元组；文字串是用来对字符数组初始化:

```
char msg[] = "Hello, world!\n";
```

这个例子声明 msg 为一个 15 个字符的数组，包含结束的空字符，如果数组的大小在声明的时候已经知道，则数组大小必须至少和字符串大小一样大。

1.2.2 表达式

表达式是运算符结合变量和常量产生一个数值，下表中的栏位皆为表达式的例子:

代数表达式	XC 表达式
$a*b-c$	`a * b - c`
$(a+b)(c+d)$	`(a + b) * (c + d)`
$a/b+c$	`(a/b) + c`

没有小括号的表达式通常是利用运算符的优先顺序规则由左到右来评值，规则规范 * 运算符比 + 运算符有更高的优先顺序，也就是说在表中的第二个表达式需要小括号在两个加号附近，使加法运算可以先被运行。

表格 I 总结 XC 所支援的运算符，列在较上面的运算符有较高的优先顺序，在相同区块中的运算符有相同的优先顺序，运算符定义与 C 语言中的定义相同，完整的细节请参照 §A.6。

以分号结束的表达叙述为语句，大部分的语句为赋值式，如:

```
x = a * b;
```

或是函数的调用，如下:

```
printf("Hello, world!\n");
```

XC 新增

表达式值必须是非歧义的表示，当表达式的值会根据其运算元评值的顺序而有不同，则产生歧义性，例如:

```
i = i++;  /* 无效 */
```

在这个例子中，`i` 的值是依据所运行的加号运算符顺序而来，一般而言，如果子表达式包含变量 V 的更改，没有其他的子表达式允许使用 V，这个规则递归地用于表达式中所有会去读或写全局变量的函数。

1.2.3 类型转换

如果运算符有不同类型的运算元，运算元会被转换成一个共同的类型；通常在运算之前，类型较“低”的会被**提升**为较“高”的类型，结果会是较高等级的类型，例如在表达式中:

```
'c' + 1
```

二元运算符 + 包含 `char` 和 `int` 运算元，`char` 运算元被转换成 `int`，而表达式的结果也是 `int`。

表格 I
XC 运算符

运算符	说明	种类	结合性
`++ --`	后置递增/递减	一元运算符	由左到右

`++ --`
`+ -`
`!`
`~`
（**类型**）
`sizeof`
`isnull`
	前置递增/递减
	一元加/减
	逻辑否定
	位元逐位补数
	明确强制类型转换
	位元组大小
	判断是否为空的参考
		一元运算符	由右到左

`* / %`	乘法/除法/模数运算	二元运算符	由左到右
`+ -`	加法/减法	二元运算符	由左到右
`<< >>`	位元左/右移位	二元运算符	由左到右

`<`
`<=`
`>`
`>=`
	小于关系
	小于或等于关系
	大于关系
	大于或等于关系
		二元运算符	由左到右

`== !=`	关系等于/不等于	二元运算符	由左到右
`&`	位元 AND	二元运算符	由左到右
`^`	位元互斥 OR	二元运算符	由左到右
`|`	位元内含 OR	二元运算符	由左到右
`&&`	逻辑 AND	二元运算符	由左到右
`||`	逻辑 OR	二元运算符	由左到右
`c?`t`:`f	三元条件	三元运算符	由右到左

`=`
`+= -= *= /=`
`%= &= ^= |=`
`<<= >>=`
	赋值
	计算赋值
	计算赋值
	计算赋值
		二进制	由右到左

一般性提升规则以及算术转换在 §A.1.1 有规范，对于 XS1 装置则总结如下：

- 如果 int 可以表示所有原本类型的值，则将 char 和 short 转换成 int，否则转换成 unsigned int。
- 如果任一个运算元是 unsigned int，则转换其他的成 unsigned int。

明确的类型转换可以在表达式中使用一元类型转换运算符强制运行，如：

```
(char)('a' + i); // 强制转型 32 位元整数成 8 位元字符
```

强制类型转换时常被用于输出的语句，目的是指定要通讯的数据量 (参见 §3.3)，类型转换的意义有如将表达式的结果以指定的类型赋值到变量；而类型转换不能用于数组以及表达式。

1.3 控制流程

控制流程语句表达裁定语句运行顺序的决定，语句串列在由 { } 组成的**区块**中依序被运行，如下所示：

```
main(void) {
  int x = 2, y = 3;
  int z = x * y;
  z++;
}
```

在一个区块中，所有声明必须在所有语句的最上面；一个区块是语法上等价于单一个语句，可以置放于任何语句需要的位置。

1.3.1 if-else

if-else 结构最多选择一个语句运行，如下所示：

```
if (n > 0)
  printf("Positive");
else if (n < 0) {
  printf("Negative");
  x = 0;
}
else
  printf("Zero");
```

对每一个以小括号包围的表达式，皆监督可能会运行的语句或是代码区块，而 else-if 以及 else 语句不是必要的，这些表达式会依照他们出现的顺序被检验，并且第一个非零的表达式对应的语句将被运行，然后整个结构便结束。

else 语句总是和前一个没有 else 的 if 呼应，如果有多重 if 语句的巢状结构，这是很重要的一点；花括号可以用来强制不同的相关性。

1.3.2 switch

switch 语句测试一个表达式是否符合某个常量整数值，并且分岔至相对应 case 的代码主体中，如下所示:

```
switch(state) {
  case READY :
    state = SET;
    if(x < 0)
      state = FAIL;
    break;
  case SET :
     state = GO;
     if(y > 0)
       state = FAIL;
     break;
  case GO:
    printf("Go!\n");
    break;
  case FAIL :
  default :
    /* 错误 */
}
```

如果有 default 标签出现在 switch 语句中，并且没有任何常量值符合的情况下，default 下面的代码会被运行。

每段 case 代码主体必须以 break 或是 return 结束，避免运行接在其后面的代码区段。

XC 新增

1.3.3 循环

只要表达式中的值仍是非零，while 循环会重复运行语句，如下所示:

```
int i = 0;
while (i<n) {
  a[i] = b[i] * c[i];
  i++;
}
```

这是许多程序中会见到的典型例子，可以在数组中前 n 个原素上反覆地运行，另一个形式是使用 for 循环，在最上面的地方，提供一个结合初始化、条件式测试以及加总的方式，上面这个例子也可以写成:

```
for (int i=0; i<n; i++)
  a[i] = b[i] * c[i];
```

第一个和第三个表达式通常是赋值式，并且第二个是关系表达式，第一个赋值式可以含变量声明，该变量声明仅对循环中的程序区段有效，任何这三部分的其中之一皆可以省略，但是分号必须保留。

do-while 循环于运行相对应的代码后会检测条件是否符合，保证至少运行一次代码主体，其形式如下：

```
do {
  代码主体
} while (表达式);
```

1.3.4 break 和 continue

break 语句立即跳出一个循环，如下所示：

```
while (1) {
  //...资料输入及处理...
  if (错误)
    break;
}
```

在这例子里面，当错误出现的时候，break 语句由 while 循环退出；一般说来，break 会跳出最靠近的循环或是 switch 语句。

continue 语句和 break 相似，除了它会造成包含的循环开始下一次反覆运行，如下所示：

```
for (int i=0; i<n; i++) {
  if (a[i] == 0)
    continue;
  //...处理非零元素...
}
```

在 for 循环中，continue 后立即运行的语句是循环的递增，在 while 和 do 循环当中，下一个运行的语句是条件的检测。

continue 语句经常使用于后面代码较复杂的情况下，若将检测条件反向以及缩排至另一层次，会造成程序巢状结构过深而不易理解。

1.4 函数

函数命名是一个语句的区块，提供将运算封装以及参数化的方式；下面这个程序定义并使用计算阶乘的函数 fact：

```
int fact(int);

/* 测试 fact 函数 */
int main(void) {
  for (int i=0; i<10; i++) {
    int f = fact(i);
    // ... 打印数值 f ...
  }
}

int fact(int n) {
  for (int i=n-1; i>1; i--)
    n = n * i;
  return n;
}
```

main 之后的 fact 声明:

```
int func(int n)
```

声明 fact 为函数，带有类型 int 名称为 n 的参数，而且回传一个整数类型的数值，当 main 调用 fact 时，i 的值将被复制给 fact 里的新变量 n，且变量 n 对 fact 是局部私有的，其他函数可以使用这名称不会产生冲突。

在 fact 函数声明之后，{ } 括号聚集语句形成区块，使得该区块为此函数之**定义**，每个函数中至多允许一个定义。

fact 代码主体中的 return 语句传回一个计算而得的阶乘数值给 main；若函数无回传值，其类型则指定为 void。

在 main 之前的第一个 fact 声明:

```
int fact(int);
```

是**函数原型**，声明 fact 类型但没有给予其定义，函数的原型或是其定义必须在被使用之前必须于代码中出现，函数原型必须符合其定义以及使用方式，原型中的参数名称可省略。

1.4.1 函数引数

函数引数通常是以传值的方式传递，在这情况下，值会被复制到一个属于该函数私有的新变量。

引数也可能由传参考的方式来传递，使得任何对区域变量的更改也会使调用函数中的引数一并改变，下面的程序交换了由传参考来的引数数值。

XC 新增

```
void swap(int &x, int &y) {
  int tmp = x;
  x = y;
  y = tmp;
}

int main(void) {
  int a = 1;
  int b = 2;
  swap(a, b);
}
```

这个声明:

```
void swap(int &x, int &y)
```

将 swap 声明成接受两个传参考变量的函数，参考参数是用 & 为前缀表示。

XC 新增

建立超过一个以上的参考到相同的对象是无效的，在上面的例子，使用相同变量两次去调用 swap 是无效的。

数组是隐含地由参考传入的，也就是说一个数组不能被传到一个函数的两个参数，但是，例如：传一个二维数组的两个不同列到一个函数是允许的。

在函数声明中，可以省略数组参数的最大维度，这允许函数可以运算任意大小的数组，举例来说:

```
int strcount(char str[], int len);
```

如果数组参数的大小有被指定，传一个较大数组是允许的，但是其最高的元素是不能访问的，而传入较小的数组是无效的。

1.4.2 选择性引数

XC 新增

传参考的参数可被指定为**可为空**，也就是说，它可以包含有效的参考或是特别的 null 参考，下面的程序检查两个数组中，有多少对应的元素有相同的数值，并且当调用者有提供第三个数组时，将 0 或是 1 赋值给第三个数组中的元素。

```
int compare(int x[], int y[], int ?matches[], unsigned size) {
  int n = 0;
  for (int i=0; i<size; i++) {
    int match = (x[i] == y[i]);
    n += match;
    if (!isnull(matches))
      matches[i] = match;
  }
  return n;
}
```

```
int main(void) {
  int x[5] = {0, 1, 2, 3, 4};
  int y[5] = {1, 1, 1, 3, 3};
  int z[5] = {1, 1, 1, 1, 1};
  int m[5] = {0};
  int n;

  (void)compare(x, y, m, 5);
  n = compare(y, z, null, 5);

  return 0;
}
```

下面代码:

```
int compare(int x[], int y[], int ?matches[], unsigned size)
```

声明 compare 为函数，该函数可以接受三个数组及一个 size 变量的参数，第三个参数 matches 被指定成可为空，在其名称之前以 ? 为前缀表示。

运算符 isnull 会产生数值 1 如果其引数是有效的参考，否则，产生数值 0；企图引用或是使用空对象是无效的。

在 main 第一个调用当中，数组 m 在第三个引数被传入，compare 给定数组中的值，而第二个调用当中，第三个传入引数是 null，compare 则不会去对数组赋值。

1.4.3 多重回传值函数

函数可以回传超过一个以上的回传值，如下所示:

XC 新增

```
{int, int} swap(int a, int b) {
  return {b, a};
}

void main(void) {
  int a  = 1;
  int b  = 2;
  {a, b} = swap(b, a);
}
```

回传类型的串列、return 后面的值串列，以及被赋值的变量串列是以花括号括住；在赋值的串列中，元素数目必须符合函数回传的数量，但可以使用 void 来忽略任何的回传值，如下所示:

```
{a, void} = f();
```

1.5 再解译

XC 新增

再解译会使变量被当成有另外不同的类型，但并无类型转换，下面的函数使用再解译来传输一个位元组数组里多个元素，当成一个 32 位元整数传输。

```
void transmitMsg(char msg[], int nwords) {
  for (int i=0; i<nwords; i++)
    transmitInt((msg, int[])[i]);
}
```

这个结构:

```
(msg, int[])
```

再次解译 msg 数组是一个整数数组，并以索引值取值:

```
(msg, int[])[i]
```

在这个例子中，整数数组的大小是动态决定的，举例来说，如果传入引数数组大小为 10 位元组，被再次解译的整数数组上限是 2，而且最上面两个字符是不能被访问；如果再次被解译的大小是有给定的，必须不能超过原来类型的大小。

再解译一个对象成为另外一个较大容量类型的动作是无效的 (参考 §C.5)，原本的声明应该要说明所有可能再解译所需要的最大的储存空间。

1.6 和 C 语言的比较

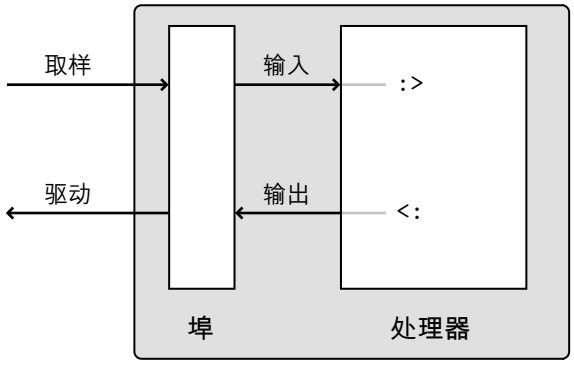
XC 有许多和 C 语言相同的特性，最主要缺少的是对指针的支援，因此，许多在 C 语言中没有定义的编程错误对 XC 是无效的，并且会造成边译的错误或是动态运行时候的异常；所有 XC 的数据型态和运算符和在 C 语言中的意义相同，而用户定义的类型包含结构、联集、枚举以及自订型态都有支援；传参考的参数以及多重回传值函数延伸自 C 语言中经常使用指针的操作，XC 的作用域以及连结规则和 C 语言相同，两个语言都使用预处理器。

XC 不支援浮点数、long long 计算、位元栏位结构以及 volatile 数据类型，也不提供 goto 语句的支援，这些限制在将来的版本中可能会有改善和其他语言之间的相容性。

第 2 章

输入输出

埠是连接处理器和一个或多个实体脚位，这样定义了处理器和环境之间的介面，埠逻辑可以驱动其脚位至高或低的准位，或是可以取样其脚位上的值，又或者可以等待特定条件的发生。埠不是记忆体映射式的，相对地，他们是透过特定的指令访问，XC 提供整合性输入输出语句，使在埠上的操作写法变得简单，下图说明这些操作:



输入/输出数据率可以使用硬件的计时器控制，硬件计时器可用来对输入和输出指令运行延缓一段定义的时间，处理器也可以等待超过一个以上埠的输入，允许多个输入/输出装置同时被接上。

2.1 输出数据

下面是切换一个脚位成高准位或是低准位的范例程序:

```
#include <xs1.h>

out port p = XS1_PORT_1A;

int main(void) {
  p <: 1;
  p <: 0;
}
```

声明:

```
out port p = XS1_PORT_1A;
```

声明一个名称为 p 的输出埠，对应到一位元的埠标识符 1A[1]。

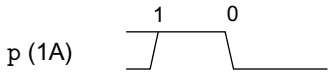
下面语句:

```
p <: 1;
```

输出数值 1 到埠 p，会使埠驱动其相对应的脚位为高准位，这个运行的动作会一直持续到下一个语句:

```
p <: 0;
```

这个语句输出 0 到这个埠，使得埠驱动其相对应的脚位为低准位，下面的图显示该程序产生的输出。



这个脚位一开始并没有被驱动，在第一次输出被运行时，脚位变为高准位，在第二次输出运行之后则被驱动至低准位，一般而言，当输出到 *n* 个位元的埠时，输出值的最低有效 *n* 个位元会在脚位上驱动，其馀的则被忽略。

所有的埠必须被声明成全局变量，并且没有任何两个埠可以初始化为相同的埠标识符，在初始化之后，埠可能不被赋值；传埠给函数是允许的，但该埠不能出现在超过一个以上的函数引数中，否则会产生非法别名。

[1] XS1_PORT_1A 的值是定义在头文件 <xs1.h> 中，大多的开发平台皆提供 XN 文件，从该文件中，头文件 <platform.h> 会由 XN 文件产生，并且定义更多直觉的埠名称，例如: PORT_UART_TX 和 PORT_LED_A；这些名称皆在相对的硬件使用手册上有说明。

2.2 输入数据

下面程序连续取样输入埠的四个脚位，当取样值超过 9 时，会驱动一个输出埠到高准位:

```
#include <xs1.h>

in  port  inP = XS1_PORT_4A;
out port outP = XS1_PORT_1A;

int main(void) {
  int x;
  while (1) {
    inP :> x;
    if (x > 9)
      outP <: 1;
    else
      outP <: 0;
  }
}
```

下面声明:

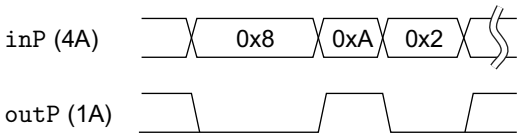
```
in port inP = XS1_PORT_4A;
```

声明一个名称 inP 为输入埠，也就是对应到 4 个位元的埠标识符 4A。

下面语句:

```
inP :> x;
```

读进被 inP 埠取样的数据到变量 x，下面的图显示这个程序的输入以及预期的输出。



这个程序连续地由 inP 埠输入：当取样到数据 0x8 时，输出被驱动至低准位，相对地，当取样到数据 0xA 时，输出会转为高准位，而当取样到 0x2 时，则又回到低准位，每个输入值会被取样许多次。

2.3 于输入脚位等待条件产生

输入的操作可以等待脚位上两个条件之一发生：等于或是不等于某个特定值；下面程序使用**条件式的输入**来计算在其输入脚位上的转换次数。

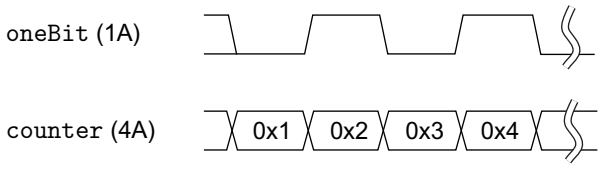
```
#include <xs1.h>

in  port oneBit  = XS1_PORT_1A;
out port counter = XS1_PORT_4A;

int main(void) {
  int x;
  int i = 0;

  oneBit :> x;
  while (1) {
    oneBit when pinsneq(x) :> x;
    counter <: ++i;
  }
}
```

下面这个语句:

```
oneBit when pinsneq(x) :> x;
```

指示埠 oneBit 等待到脚位上的值不等于原本 x 的值，将该值取样并存到 x 以提供处理器。

下面波型图显示这个程序的范例输入和预期输出。



如另外的例子所示，在 4 位元的埠上等待以太网的前序信号所需要的唯一操作是:

```
ethData when pinseq(0xD) :> void;
```

一但条件符合后，处理器必须完成从埠输入的操作，即使输入值是不需要的，这在 XC 中是以输入到 void 的类型来表示。

使用一个条件式的输入比在软件中用轮询有更低耗电使效率更高，因为它在埠仍在监控其脚位的同时，使处理器闲置而消耗较少的电量。

2.4 使用计时器控制输入/输出数据量

计时器是埠的一种特别类型，用来量测以及控制事件之间的时间；每个计时器都有一个 32 位元的计数器，连续地以 100MHz 的频率增加，其值可以在任何时间点被输

入，输入到计时器也可以延迟到将来的一个时间点。下面程序使用计时器来控制一位元埠被开关的速度:

```
#include <xs1.h>
#define DELAY 50000000

out port p = XS1_PORT_1A;

int main(void) {
  unsigned state = 1, time;
  timer t;
  t :> time;
  while (1) {
    p <: state;
    time += DELAY;
    t when timerafter(time) :> void;
    state = !state;
  }
}
```

下面声明:

```
timer t;
```

声明一个名称为 t 的计时器，由 XCore 的计时器集合而来的计时器资源。

下面的声明:

```
t :> time;
```

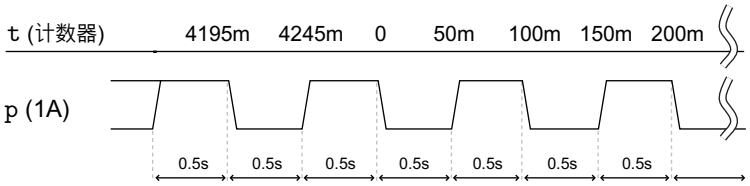
输入 t 的计数值到变量 time，这个变量接着增加 DELAY 值，此值给定计数递增值，由于计时器的周期是 10ns，因此，指定一个将来的时间是: $50,000,000 * 10ns = 0.5s$。

下面的条件式输入语句:

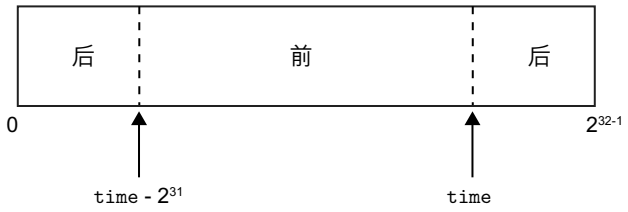
```
t when timerafter(time) :> void;
```

会等到时间到了后马上完成之后的输入。

下面波形图显示这个程序驱动的数据。



timerafter 函数将计时器的计数器视同具有两个不同的范围，如下所示:



所有在范围 (time-2^{31}..time-1) 之间的值都被当成于 time 之前到来，而范围 (time$+1$..time$+2^{32-1}$, 0..time-2^{31}) 之间的值则视为之后出现。如果两个输入的值之间间距可以放进 31 个位元，则保证 timerafter 有正确的行为，否则可能由于溢位而有不正确行为，也就是说，计时器可以被用来量测最多 $2^{31}/100,000,000 = 21s$ 的时间。

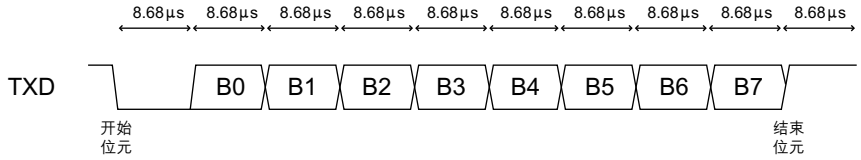
一个可能的编程错误会出现于输入新的时间而不是将其类型转换成 void 而忽略，如下所示:

```
t when timerafter(time) :> time;
```

因为处理器并非在时间到达之后马上完成输入，这个动作实际上会增加一点点额外的数量到 time，这个量在循环反覆时会复合累加，导致信号飘掉并且最后和接受器不同步。

2.5 范例研讨: UART (第一部分)

通用异步接收器/传送器 (UART) 在并行与串列形式之间转换，以在两个一位元线上以固定数据传输率通讯，每个位元数据是在数据传输率所定义的时间驱动的，并且接收器必须在这个时间内取样，下图显示以每秒 115200 位元的速度传送一位元组的数据。



导线静态时为高准位，当传输一个位元组时需先送出一个**开始位元** (0)，接着是八个数据位元，最后是一个**结束位元** (1)；每秒 115200 位元的意思是指每个位元被驱动需要的时间是: $\frac{1}{115200} = 8.68\mu s$。

UART 通常是以微控制器实作，并使用中断机制来排程记忆体映射式的输入和输出操作；由于有特定的输入/输出指令，用 XMOS 装置实作 UART 是简单的，下面的程序定义一个 UART 的传输器:

```
#include <xs1.h>

#define BIT_RATE 115200
#define BIT_TIME 100000000 / BIT_RATE

out port TXD = XS1_PORT_1A;
out port RXD = XS1_PORT_1B;

void transmitter(out port TXD) {
  unsigned byte, time;
  timer t;

  while (1) {
    /* 取得下个传送位元组 */
    byte = getByte();
    t :> time;

    /* 输出开始位元 */
    TXD <: 0;
    time += BIT_TIME;
    t when timerafter(time) :> void;

    /* 输出数据 */
    for (int i=0; i<8; i++) {
      TXD <: >> byte;
      time += BIT_TIME;
      t when timerafter(time) :> void;
    }

    /* 输出停止位元 */
    TXD <: 1;
    time += BIT_TIME;
    t when timerafter(time) :> void;
  }
}
```

传输器首先输出一个位元组的开始位元，接着是以条件式输入等待位元传输的时间经过，而数据位元以及停止位元都以相同的方式传送。

在 `for` 循环的输出语句:

```
TXD <: >> byte;
```

包括修饰符 >>，会在输出最低有效埠宽位元后，右移 `byte` 一个埠的宽度 (在此例中，为一个位元)，这个操作是整合于输出指令的，终究比传输后单独操作位移更有效率。

下面这个函数，在一位元导线上接收位元组的串流数据:

```
void receiver(in port RXD) {
  unsigned byte, time;
  timer t;

  while (1) {
    /* 等待接收开始位元 */
    RXD when pinseq(0) :> void;
    t :> time;
    time += BIT_TIME/2;

    /* 输入数据 */
    for (int i=0; i<8; i++) {
      time += BIT_TIME;
      t when timerafter(time) :> void;
      RXD :> >> byte;
    }

    /* 输入停止位元 */
    time += BIT_TIME;
    t when timerafter(time) :> void;
    RXD :> void;

    putByte(byte >> 24);
  }
}
```

这个接收器对送进来的信号取样并等待开始位元，在接收到这个位元之后，它会等待 $1\frac{1}{2}$ 个位元传输时间，之后在导线的第一个传输位元中点取样，接下来每隔 $8.68\mu s$ 进行取样一个位元。这个在 `for` 循环的输入语句:

```
RXD :> >> byte;
```

包含限定符 `>>`，用来先右移 `byte` 的值一个埠的宽度 (此例为一个位元)，然后输入下个样本到它的最高有效埠宽位元；语句里最后的表达式:

```
putByte(byte >> 24);
```

将整数 `byte` 的位元右移 24 个位元，因此输入的值最后会少于最低有效位元组。

2.6 多重输入的反应

下面程序仅使用单一线程从两个不同的埠输入两个串流数据，其中一个埠的数据是否可取得是由切换脚位上的信号来指示的，而另一个埠上数据则是以固定速度接收的。

```
#include <xs1.h>

#define DELAY_Q 10000000

in port toggleP = XS1_PORT_1A;
in port dataP   = XS1_PORT_4A;
in port dataQ   = XS1_PORT_4B;

int main(void) {
  timer t;
  unsigned time, x = 0;

  t :> time;
  time += DELAY_Q;
  while (1)
    select {
      case toggleP when pinsneq(x) :> x :
        readData(dataP);
        break;
      case t when timerafter(time) :> void :
        readData(dataQ);
        time += DELAY_Q;
        break;
    }
}
```

select 语句会在 toggleP 埠或是计时器 t 上进行输入，依照这些资源准备好输入的状态来决定先后顺序，如果两个输入在相同的时间都是准备好的状态，只有一个会被运行，另外一个在循环下次返回的时候仍旧会是准备好的状态；在运行输入操作之后，下面的代码主体会被运行；每个代码主体都必须以 break 或是 return 语句结束。

case 语句不允许含有输出操作，因为 XMOS 架构需要输出操作完成运行，但是允许输入操作等待一个相对应的输出才完成。

每个埠和计时器只能出现在其中之一的 case 语句，这是因为 XMOS 架构限制每个埠以及计时器资源在一个时间点只能等待一个条件。

在此例中，处理器有效地运行两个独立的任务，但需要足够快到可以实时处理两个串流数据，如果这是不可能的，则需要使用两个线程来处理数据 (参考章节 3)。

2.7 范例研讨: UART (第二部分)

下面的程序使用 select 语句实作以单一线程在 UART 两端传送以及接收:

```
void UART(port RX, int rxPeriod, port TX, int txPeriod) {
  int txByte, rxByte;
  int txI, rxI;
  int rxTime, txTime;
  int isTX = 0;
  int isRX = 0;
  timer tmrTX, tmrRX;
  while (1) {
    if (!isTX && isData()) {
      isTX = 1;
      txI = 0;
      txByte = getByte();
      TX  <: 0;             // 传送开始位元
      tmrTX :> txTime;      // 设定数据位元的逾时
      txTime += txPeriod;
    }
    select {
      case !isRX => RX when pinseq(0) :> void :
        isRX = 1;
        tmrRX :> rxTime;
        rxI = 0;
        rxTime += rxPeriod;
        break;
      case isRX => tmrRX when timerafter(rxTime) :> void :
        if (rxI < 8)
          RX :> >> rxByte;
        else { // 接收停止位元
          RX :> void;
          putByte(rxByte >> 24);
          isRX = 0;
        }
        rxI++;
        rxTime += rxPeriod;
        break;
      case isTX => tmrTX when timerafter(txTime) :> void :
        if (txI < 8)
          TX <: >> txByte;
        else if (txI == 8)
          TX <: 1; // 停止位元
        else
          isTX = 0;
        txI++;
        txTime += txPeriod;
        break;
} }  }
```

isTX 、txI 、isRX 和 rxI 这些变量决定 UART 的哪些部分是工作状态以及多少位元数据已经传送和接收。

while 循环首先检查传送器是否在有数据等待传送时处于不工作状态，如果是，它会输出开始位元并且设定输出第一个数据位元的逾时时间。

在 select 语句中，防卫条件:

```
case !isRX => RX when pinseq(0) :> void :
```

检查 isRX 是否等于零，也就是接收器是不工作状态，在这情况下，当在此脚位上的值为 0 时，会开始使 RX 埠上的数据输入；在运算符 => 左边的表达式是用来**开始**输入状态，这个条件的主体是对输入第一个数据位元设定逾时时间。

第二个防卫条件:

```
case isRX => tmrRX when timerafter(rxTime) :> void :
```

检查 isRX 是否为非零，也就是接收器是工作状态，如果是，会使输入由计时器 tmrRX 传入，这个条件主体会输入数据的下一个位元，并且在所有位元都输入时，数据会被储存并且再将 isRX 设定回零。

第三个防卫条件:

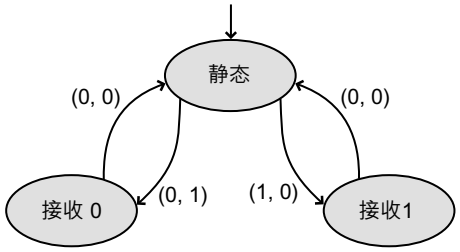
```
case isTX => tmrTX when timerafter(txTime) :> void :
```

检查 isTX 是否为非零，也就是传输器是否为工作状态，如果是，会使输入由计时器 tmrTX 传入，这个条件的主体会输出数据的下一个位元，并且在输出所有的位元后，将 isTX 设定为零。

如果这个 UART 控制器被用在有杂讯的环境下，可以透过对每个输入位元多次取样并取其平均，以改善可靠性；更强健的实作是检查接收到的停止位元是否为 1，也就是预期的值。

2.8 参数化的 select

select 语句可以实作为函数，以允许在其他地方重覆使用，一个这样的例子是参数化二位元编码线上取样数据的代码；如下所示，静态编码是以 (0, 0) 来编码，其中 0 是有转换到 (0, 1) 的意味，而 1 是指转换到 (1, 0)，任何转换的情况都会跟随着另一个转换，最后回到 (0, 0)。



下面程序利用这样的结构，以 select 函数透过两个脚位输入一位元组的数据:

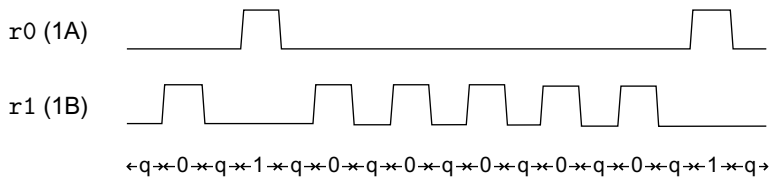
```
#include <xs1.h>

in port r0 = XS1_PORT_1A;
in port r1 = XS1_PORT_1B;

select inBit(in port r0, in port r1,
             int &x0, int &x1, char &byte) {
  case r0 when pinsneq(x0) :> x0 :
    if (x0 == 1)  /* 转换到 (1, 0) */
      byte = (byte << 1) | 1;
    break;
  case r1 when pinsneq(x1) :> x1 :
    if (x1 == 1)  /* 转换到 (0, 1) */
      byte = (byte << 1) | 0;
    break;
}

int main(void) {
  int x0 = 0, x1 = 0;
  char byte;
  for (int i=0; i<8; i++)
    inBit(r0, r1, x0, x1, byte);
}
```

下面代码:

```
select inBit(in port r0, in port r1, int &x0, int &x1, char &byte)
```

声明 inBit 为 select 函数，该函数含有五个引数，并且有一个隐含的回传值 void，代码主体有两个 case 语句。

下面波形图显示这个程序的范例输入，接收到的位元值分别是：0、1、0、0、0、0、0 和 1 (‘A’)。



相对于以固定速度传送数据的 UART，这个方式提供所连接的 XMOS 装置或是元件支援的最高可能传输率。

定义 inBit 为 select 函数的一个好处是其分别都可以用于一个大的 select 语句的一部分，如下个程序，可以解码在四个脚位上取样到的两个位元组的值。

```
#include <xs1.h>
#define NBYTES 2

in port r[NBYTES*2] = { XS1_PORT_1A, XS1_PORT_1B,
                        XS1_PORT_1C, XS1_PORT_1D };
int main(void) {
  int state[NBYTES*2] = {0, 0, 0, 0};
  char byte[NBYTES];
  for (int i=0; i<8*NBYTES; i++)
    select {
      case inBit(r[0], r[1], state[0], state[1], byte[0]);
      case inBit(r[2], r[3], state[2], state[3], byte[1]);
    }
}
```

此 select 语句于 case 语句中调用 inBit 函数，使处理器允许事件在由参数传入的埠上发生。

更简洁的方式是使用**重复器**来诠释上层 select 语句，如下:

```
select {
  case (int i=0; i<NBYTES; i++)
    inBit(r[i*2], r[i*2+1], state[i*2], state[i*2+1], byte[i]);
}
```

以下重复器:

```
(int i=0; i<2; i++)
```

会反覆两次，每次会调用 select 中的 inBit 函数，致使埠有不同对应的索引值 i；反覆的次数不需要是常量，但是反覆器不能在重复器外修改。

第 3 章

并发性

许多设计需要同时运行多个任务，其中一些任务是彼此独立不相关，而其他的会需要依靠其他的任务完成共有目的；XC 提供简单的机制建立**并发线程**，这些并发线程能独立地运行，并且在需要的时候和其他的线程沟通，线程之间使用**通道**来进行数据通讯，通道是点对点地将一对线程连接起来，可以同步地或是异步地传输数据。

3.1 建立并发线程

下面的程是建立四个并发线程，所有的线程独立地运行不同任务，其中两个线程是在 XCore 0 核心上运行，另外的分别在 XCore 1 以及 XCore 2 核心上运行。

```
#include <platform.h>

on stdcore[0] : out port tx      = XS1_PORT_1A;
on stdcore[0] : in  port rx      = XS1_PORT_1B;
on stdcore[1] : out port lcdData = XS1_PORT_32A;
on stdcore[2] : in  port keys    = XS1_PORT_8B;

int main(void) {
  par {
    on stdcore[0] : uartTX(tx);
    on stdcore[0] : uartRX(rx);
    on stdcore[1] : lcdDrive(lcdData);
    on stdcore[2] : kbListen(keys);
  }
}
```

头文件 platform.h 提供全局变量 stdcore 的声明，用来说明埠和线程的位置[1]。

下面代码:

```
on stdcore[0] : out port p = XS1_PORT_1A;
```

声明一个名称为 p 的一位元输出埠，这个埠会对应到标准核心 0 上的埠标识符 1A。

四个在 par 花括号中的语句是并发地运行，如四个使用**分叉-联结并行化**的独立线程：在开始花括号 { 时，父线程去建立其他三个线程；其中每个线程接着运行函数，而在结束花括号 } 的地方，父线程会等待所有的函数返回再继续运行。

par 语句可以用在程序的任何地方，每个 XS1 装置上，每个处理器至多有八个线程的限制，程序如果企图运行超过这个数目的线程是无效的。

on 语句是用来指定连接到埠的元件实体位置，并且在所有的 XCore 间划分可用的线程。

对单核心程序而言，埠的声明是不需要有前缀 on，该情况下的所有埠以及线程都是在 XCore 0 上面运行；而在多核心程序中，所有的埠以及线程都需要明确地将前缀字 on 放入声明。

多核心系统的 main 函数中，只允许有通道的声明和单一的 par 语句和可以省略的 return，on 语句只能在这个函数中指定线程位置。

3.2 不交叉线程规则

所有的变量皆受到使用规则的规范，避免以具有潜在危险方式在线程间共用；一般而言，每个线程，对本身私有的变量有完整的访问权限，但是和其他线程共享的变量只能有限制的访问，线程 $\mathrm{T}_0 \cdots \mathrm{T}_i$ 和一组变量 $\mathrm{V}_0 \cdots \mathrm{V}_j$ 之间不交叉的规则如下:

- 如果线程 T_x 包含任何对变量 V_y 修改，其他的线程 $(\mathrm{T}_t, t \neq x)$ 皆不可以使用 V_y，也就是不可以读或写。
- 如果线程 T_x 包含变量 V_y 的参考，则其他的线程 $(\mathrm{T}_t, t \neq x)$ 皆不可以改变 V_y。
- 如果线程 T_x 包含埠 V_p 的参考，则其他的线程皆不可以使用 V_p，也就是不可以读或写。

换句话说，一群线程可以有**共用**的唯读变量，但是只有单一个线程对变量能有**独占**的读/写权限，这些规则保证每个线程皆有明确定义，并且和在其他线程运行的指令排程是独立不影响的，使用通道进行输入和输出来明确地让线程间通讯 (参考 §3.3)。

[1]目的平台是用 XMOS 网络规格语言 XN 来诠释，大多数的板提供完整套件，含相对应的 XN 文件，该文件说明可以使用的装置以及他们的连接性。这些数据在编译对应连接阶段会用来产生一个多节点的可运行文件，可以启动并设定整个系统。

3.2.1 范例

以下的程序范例是合法的，因为 k 对线程 X 和线程 Y 是唯读的共用变量，i 在线程 X 中有修改，在线程 Y 当中并没有使用，在线程 Y 中对 j 有修改，但该变量在线程 X 中并没有使用。

```
int main(void) {
  int i = 1, j = 2, k = 3;
  par {
    i = k + 1;  // 线程 X
    j = k - 1;  // 线程 Y
  }
}
```

如果 i 或是 j 也在另外的线程被读取，则这例子便是非法，如下面程序所示:

```
int main(void) {
  int i = 1, j = 2, k;
  par {
    i = j + 1;  // 线程 X: 不合法的共用 i
    k = i - 1;  // 线程 Y: 不合法的共用 i
  }
}
```

这个程序是有歧义的，因为线程 Y 读取 i 的值会依据在线程 X 中赋值与否而不同。

下面的程序是合法的，因为在线程 X 中修改的 a[0] 并没有被线程 Y 用到，而 a[1] 在线程 Y 有修改但是在线程 X 中并没有使用该变量。

```
int main(void) {
  int a[2];
  par {
    a[0] = f(0); // 线程 X
    a[1] = f(1); // 线程 Y
  }
}
```

下面这个程序是非法的，因为线程 X 修改变量 a[1]，并且 a 中一个未知的元素也在线程 Y 中被修改。

```
int x;
int main(void) {
  int a[10];
  par {
    a[1] = f(1); // 线程 X: 不合法的共用 x[1]
    a[x] = f(x); // 线程 Y: 不合法的共用 x[1]
  }
}
```

一般来说，不是以常量值索引的数组会被视为如同所有数组中元素皆被访问。

下面的程序是非法的，因为数组 a 在线程 X 和 Y 中，都被以传址的方式传入函数 f 中，如此在两个线程中皆有被修改的可能性。

```
void f(int[]);

int main(void) {
  int a[10];
  par {
    f(a); // 线程 X: 不合法的共用 a
    f(a); // 线程 Y: 不合法的共用 a
  }
}
```

如果 f 没有修改该数组，其参数应该被声明为 const，如此会使上面的程序合法化。不交叉规则分别也用于并行的语句以及递归的语句，下面的程序范例是合法的:

```
int main(void) {
  int i = 1, j = 2, k = 3;
  par {
    i = k + 1;  // 线程 X
    j = k - 1;  // 线程 Y
  }
  i = i + 1;
  par {
    j = i - 1;  // 线程 U
    k = i + 1;  // 线程 V
  }
}
```

在这个范例中，首先 i 在主线程中声明并初始化，接着在线程 X 明确地使用该变量 (线程 Y 则不被允许访问该变量)，当线程 X 和 Y 交会联结后，i 则再次于主线程中使用，最后共用于线程 U 和 V 之间。

3.3 使用通道通讯

通道以同步及点对点的方式连接两个线程，并且可在连结上通讯数据；下面的程序使用一个通道于某处理器上生产端线程和另一处理器上用户端线程进行数据通讯:

```
#include <platform.h>

on stdcore[0] : out port tx   = XS1_PORT_1B;
on stdcore[1] : in  port keys = XS1_PORT_8B;
```

```
void uartTX(chanend dataIn, port tx) {
  char data;
  while (1) {
    dataIn :> data;
    transmitByte(tx, data);
  }
}

void kbListen(chanend c, port keys) {
  char data;
  while (1) {
    data = waitForKeyStroke(keys);
    c <: data;
  }
}

int main(void) {
  chan c;
  par {
    on stdcore[0] : uartTX(c, tx);      // 线程 X
    on stdcore[1] : kbListen(c, keys); // 线程 Y
  }
}
```

以下代码:

```
void uartTX(chanend dataIn, port tx)
```

声明 uartTX 为一个函数，函数的引数是通道和埠。

下面代码:

```
void kbListen(chanend c, port keys);
```

声明 kbListen 为一个函数，函数的引数是通道和埠。

在函数 main 中，以下代码:

```
chan c;
```

声明一个通道变量，此通道被用在 par 中的两个线程间，并且每次隐含地使用对应到通道两端点的其中一个，这个使用方式在 XCore 0 上的线程 *X* 和 XCore 1 上的线程 *Y* 之间建立连结。

线程 X 调用函数 uartTX，这个函数会接收通道上的数据，并且输出到埠；而线程 Y 会调用 kbListen，用来等候从埠进来的键盘敲进，并输出数据到通道给在线程 X 的 UART 传输器，因为通道是同步的，在 kbListen 输出数据，它会等待 uartTX 准备好接收数据的状态后再继续运行。

通道是无损的，也就是说，数据在一个线程输出，一定保证被送至另外线程的输入，因此，线程的每个输出必须和另一线程的输入对应，并且输出的数据量必须等于输入的数据量，否则程序是无效的。

3.3.1 通道不交叉性规则

线程集合 $T_0 \cdots T_i$ 和通道集合 $C_0 \cdots C_j$ 的不交叉规则如下:

- 如果线程 T_x 和 T_y (其中 $x \neq y$) 使用通道 C_y，则其他的线程 $(T_t, t \neq x, y)$ 皆不能使用 C_y。
- 如果线程 T_x 使用通道端点 C_y，则其他的线程 $(T_t, t \neq x)$ 都不被允许使用 C_y。

换句话说，每个通道最多用于两个线程，如果一个通道只用于一个线程，所有通道上的输入或是输出都会永远的堵塞住。

变量和通道的不交叉规则保证：只要是在处理器间的实体路径下，任何两个线程可以同时在任何两个处理器上运行；一个通用规则是：互动频繁的线程通常应该被派置在靠近的位置。

3.4 交易

通道上的输入和输出语句通常同步沟通数据，然而，这并不永远是值得，因为它中断程序的流程，导致线程阻塞，同步所需要的时间，包含停滞时闲置等待的时间，会使整体效能缩减。

在 XC 中，两线程可以进行通讯交易，一连串匹配的输入和输出以异步的方式在通道上通讯，而且整个交易运行的开始和结束是同步的，交易如同个别的通道通讯，数据总输出量必须等于总输入量。

下面的程序使用交易有效率地在两个线程之间进行数据包通讯:

```
#include <platform.h>

int snd[10], rcv[10];

int main(void) {
  chan c;
  par {
    on stdcore[0] : master {   // 线程 X
      for (int i=0; i<10; i++)
        c <: snd[i];
    }
    on stdcore[1] : slave {    // 线程 Y
      for (int i=0; i<10; i++)
        c :> rcv[i];
     }
  }
}
```

每个交易包含同时运行的一个**主**线程和一个**从**线程，此二线程首先于主和从区块开始的时候同步，10 个整数值接着异步地通讯：只有当数据不再发送的时候，线程 X 才会被阻塞 (因为通道缓冲已满)；而只有当数据要不到的时候，线程 Y 才会停住，最后线程于离开主和从区块时同步。

交易只被允许在单一个通道上通讯，这可避免死锁发生，例如：由于切换器已满还没准备好接收数据，输出到通道因而停滞。

下面的程序将交易主体定义为函数，被称为通讯的主元件。

```
transaction inArray(chanend c, int data[], int size) {
  for (int i=0; i<size; i++)
    c :> data[i];
}

int main(void) {
  chan c;
  int snd[3], rcv[3];
  par {
    master inArray(c, rcv, 3);
    slave {
      for (int i=0; i<10; i++)
        c :> rcv[i];
    }
  }
}
```

以下代码:

```
transaction inArray(chanend c, char data[], int size)
```

声明 inArray 为交易函数，函数的引数包含一个通道端点、整数数组和数组的大小；交易函数必须仅声明一个通道端点的参数。

在 main 里面，对 inArray 的调用以 master 为前缀，表示该函数被调用时是主线程，用来和从线程进行通讯。

从的语句可以用在 select 语句中的防卫条件语句，如下所示:

```
select {
  case slave { inArray(c1, packet, P_SIZE) } :
    process(packet);
    break;
  case slave { inArray(c2, packet, P_SIZE) } :
    process(packet);
    break;
}
```

主操作定义上是有义务去完成操作的，因此，不被允许在防卫条件的语句中出现。

3.5 串流

串流通道建立两个线程之间的永久性路径，数据在上面可以有效率的传送，不需要同步化。下面的程序含有三个线程，用来读入由一个埠进来的串流数据、过滤数据并且将数据输出到另一个埠。

```
#include <platform.h>

on stdcore[0] : port lineIn = XS1_PORT_8A;
on stdcore[1] : port spkOut = XS1_PORT_8A;

int main(void) {
  streaming chan s1, s2;
  par {
    on stdcore[0] : audioRcv(lineIn, s1);
    on stdcore[0] : BiQuadFilter(s1,s2);
    on stdcore[1] : audioSnd(spkOut, s2);
  }
}
```

下面代码:

```
streaming chan s1, s2;
```

声明 s1 和 s2 为串流通道，并且不经同步化便能传送数据，串流的路径建立是由其本身的声明而来，当超出声明作用域时则会被关闭。

串流通道提供可能的最快数据传送率，一个输出语句只需单一个指令便能完成，并且只要通道的缓冲没有满，便会立刻将数据派送出去，一个输入语句也只需单一个指令，并且只有在通道的缓冲是空的时候才会停滞；不像交易，多串流可以同时被处理，但由于建立在 XCore 之间的串流需要保留切换器里的空间，因此，可声明的串流通道总数是有限制的，通道和交易并无此限制。

3.6 并行重复

重复器提供简洁以及简单的方式实作并发程序，于一组节点上，对不同的数据组运行相同的操作，下面程序建构在四个不同线程上运行的四个节点之间的通讯网络。

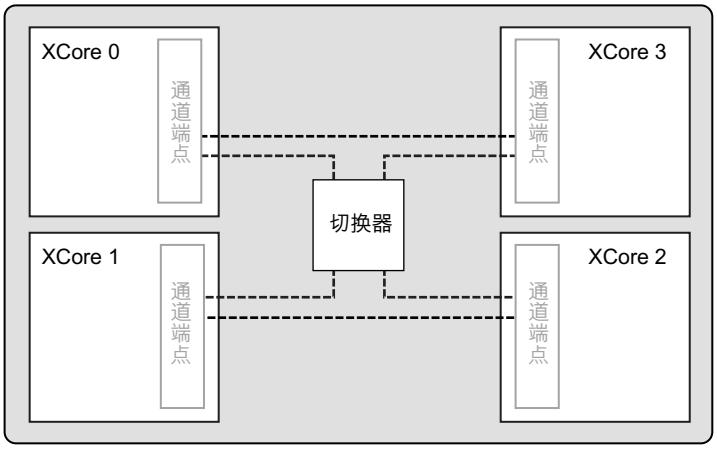
```
#include <platform.h>

port p[4] = {
  on stdcore[0] : XS1_PORT_1A,
  on stdcore[1] : XS1_PORT_1A,
  on stdcore[2] : XS1_PORT_1A,
  on stdcore[3] : XS1_PORT_1A
};
```

```
void node(chanend, chanend, port, int n);

int main(void) {
  chan c[4];
  par (int i=0; i<4; i++)
    on stdcore[i] : node(c[i], c[(i+1)%4], p[i], i);
  return 0;
}
```

下面重复器代码:

```
(int i=0; i<4; i++)
```

运行四个代码主体，每个皆包含运行在不同线程的一个函数 node 实例，重复的次数必须是常量，而且反覆器 (在此例中是 i) 不能在重复器的外面被改变，这个程序所建立的通讯网络如下:



这个程序的结构和令牌环网络相似，在通讯网络中，每个线程从邻近的线程读进令牌，运行一些动作后将令牌输出到其相邻的线程。

3.7 服务

XMOS 网络可以和任何实作支援 XMOS Link 通信协议的装置衔接，下面的程序和连接在网络上的一个 FPGA 服务设备通讯:

```
#include <platform.h>

port p = XS1_PORT_1A;

void inData(chanend c, port p) {
  // ... 从 p 输入数据并且输出到 c...
}

service fpgaIF(chanend);

int main(void) {
  chan c;
  par {
    inData(c, p);
    fpgaIF(c);
  }
}
```

下面代码:

```
service fpgaIF(chanend);
```

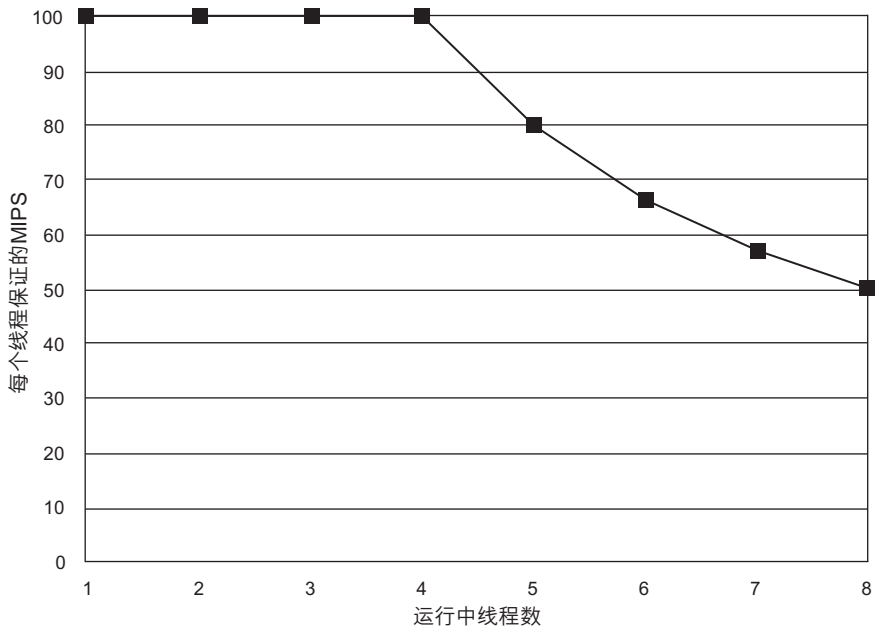
声明 fpgaIF 为网络上的服务，被声明为服务的函数只能含有通道端点的参数，并且不能给其定义，这些用来实作通道端点连接的特性定义于 XN 文件中。

另一个服务的使用是用来和由第三方事先编程在一个 XCore 的非挥发性记忆体的函数接口，典型的作法是制造商会提供一个 XN 文件，其中包含所有服务声明，这些在文件 <platform.h> 中都可以找到。

3.8 线程效能

XMOS 架构设计是可以同时进行多个实时任务，每个线程都有保证的可预期线程效能，每个处理器使用循环制的线程排程方式，这样保证如果四个以下的线程是处于工作行状态，每个线程会被配置四分之一的处理周期，如果超过四个线程在运行，则每个线程至少被配置 $\frac{1}{n}$ 周期 (对 n 个线程)，因此，一个线程的最小效能可以由计算程序中某个特定点的并发线程数量而得知。

下面的图表显示在一个 400MHz XCore 上，每个线程保证能获得的效能，也就是根据使用中的线程数量而来:



因为个别的线程可能因输入/输出而延缓，不被使用的处理器周期可以被其他的线程占有；因此，超过四个以上的线程时，每个线程的效能一般会高于上面显示的最小效能。

第 4 章

时序输入和输出

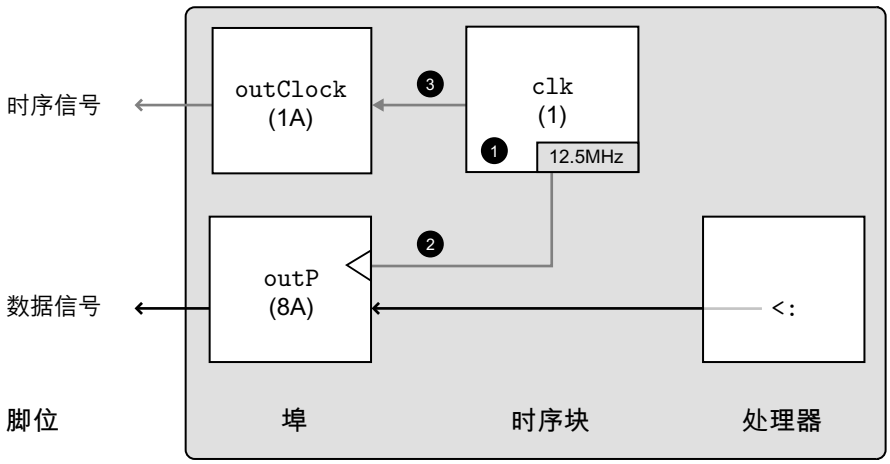
许多的通信协议规定数据必须在特定的时序边缘取样和驱动，埠可以设定使用内部产生的时序或是外部的时序，并且处理器可以记录以及控制输入和输出操作发生在哪个边缘上。在 XC 中，可以使用**时戳性**的和**时效性**运算符来于输入和输出语句中直接表示这些操作。

4.1 产生时序信号

下面程序设定一个埠以 12.5MHz 频率输出相对应的时序信号和其本身的数据:

```
#include <xs1.h>

out port outP     = XS1_PORT_8A;
out port outClock = XS1_PORT_1A;
clock    clk      = XS1_CLKBLK_1;

int main(void) {
  configure_clock_rate(clk, 100, 8);
  configure_out_port(outP, clk, 0);
  configure_port_clock_output(outClock, clk);
  start_clock(clk);

  for (int i=0; i<5; i++)
    outP <: i;
}
```

此程序将埠 outP 和埠 outClock 设定为如下图所示:



代码:

```
clock clk = XS1_CLKBLK_1;
```

声明一个名称为 clk 类型是时序的变量，该变量对应到**时序块**中的标识符 XS1_CLKBLK_1; 时序被声明为全局变量，并且每个声明皆以唯一的资源标识符初始化。

❶ 这个语句:

```
configure_clock_rate(clk, 100, 8);
```

设定时序 clk 的频率为 12.5MHz，频率是以分数 (100/8) 表示的，这是因为 XC 只支援整数算数类型。

❷ 下面的语句:

```
configure_out_port(outP, clk, 0);
```

设定输出埠 outP 是以时序 clk 为其时序，并且其脚位上的初始值是 0。

❸ 以下语句:

```
configure_port_clock_output(outClock, clk)
```

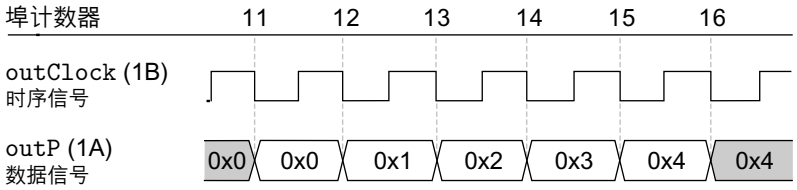
在连接到埠 outClock 的脚位上驱动时序信号 clk，使接收器可以取样被埠 outP 驱动的数据。

以下语句:

```
start_clock(clk);
```

使时序块开始产生时序边缘。

埠内部有个十六位元的计数器，在每个时序下降边缘其计数会增加，下面的波形图显示埠计数器、时序信号和被埠驱动的数据。



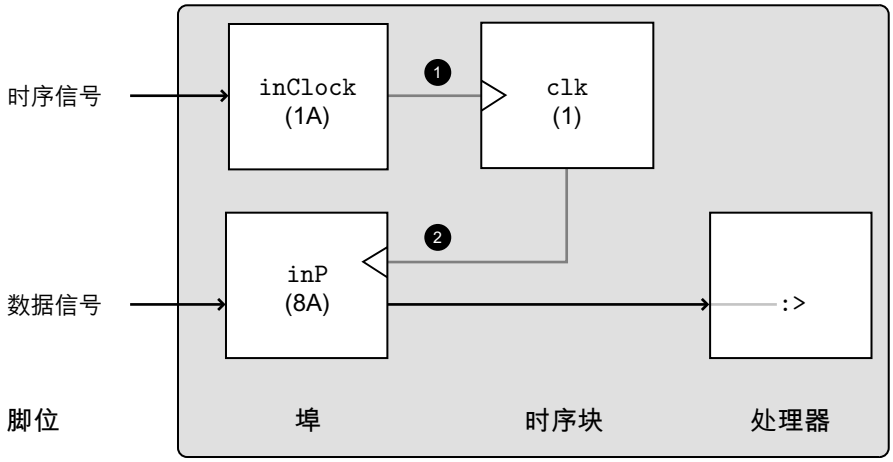
处理器的输出会造成埠在下一个时序的下降边缘驱动输出数据，数据会被埠保留着，直到另一个输出被运行。

4.2 使用外部时序信号

下面程序设定埠将取样的数据和外部时序信号同步:

```
#include <xs1.h>

in port inP     = XS1_PORT_8A;
in port inClock = XS1_PORT_1A;
clock   clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port(inP, clk);
  start_clock(clk);

  for (int i=0; i<5; i++)
    inP :> int x;
}
```

此程序将埠 inP 和 inClock 设定如下:



❶ 以下语句:

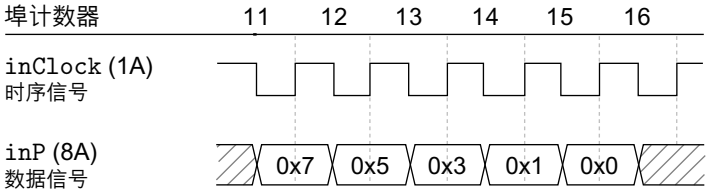
```
configure_clock_src(clk, inClock);
```

设定一位元的输入埠 inClock 提供边缘给时序 clk，当每次埠取样的值有变化时，会产生边缘。

❷ 下面语句:

```
configure_in_port(inP, clk);
```

设定 clk 为输入埠 inP 的时序，下面波形图显示埠计数器、时序信号和范例输入数据:



处理器的输入操作会造成埠在下一个时序信号的上升边缘取样数据，得到的输入值是: 0x7、0x5、0x3、0x1 和 0x0。

4.3 特定时序边缘上的输入/输出操作

于时序的特定时间点，在埠上运行输入/输出的操作是经常需要的，下面程序在第三个时序周期将脚位驱动至高准位，并且在第五个周期驱动到低准位。

```
void doToggle(out port toggle) {
  int count;
  toggle <: 0 @ count;     // 时戳性输出
  while (1) {
    count += 3;
    toggle @ count <: 1;   // 时效性输出
    count += 2;
    toggle @ count <: 0;   // 时效性输出
  }
}
```

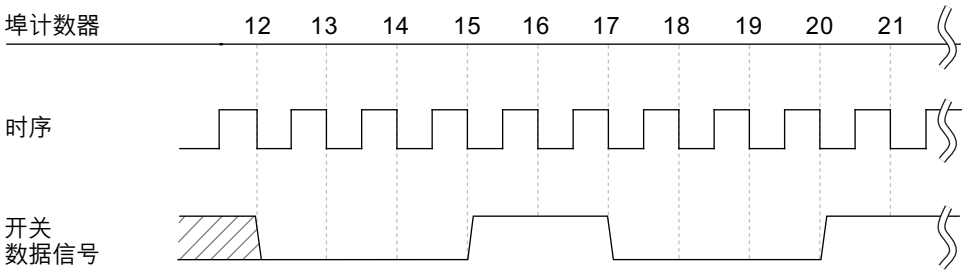
以下语句:

```
toggle <: 0 @ count;
```

运行时戳性输出操作，将 0 输出至埠 toggle，并且当输出数据在该脚位被驱动时，埠计数器的值读进到变量 count，程序接着将 count 加上 3 且运行时效性输出语句。

```
toggle @ count <: 1;
```

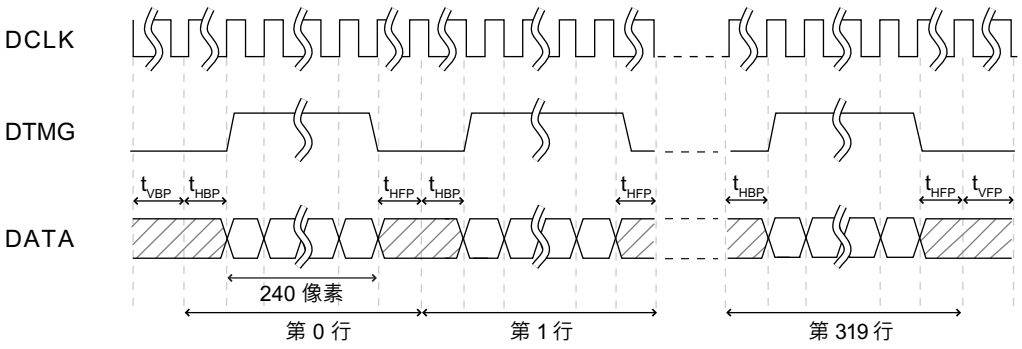
这个语句使埠等候至其计数器的值等于 count+3 (前进三个时序周期)，然后驱动其脚位至高准位；最后的两个语句延迟下一个输出两个时序周期的时间，下面波形图显示埠计数器、时序信号以及被埠驱动的数据。

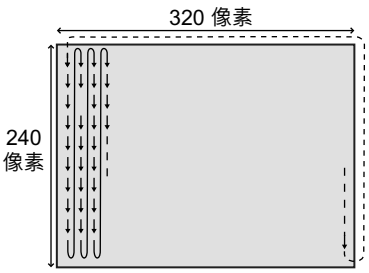


埠的计数器是在时序的下降边缘时增加，在中间边缘是没有值提供的，埠会持续在其脚位驱动前一个输出的数据。

4.4 范例研讨: LCD 屏幕驱动程序

LCD 屏用于许多嵌入系统中，驱动屏幕的细节每个屏幕皆有不同，但是主要方式是相同的。下面这个图以 Hitachi TX14 系列屏幕的操作来说明，其中包含传送单一幅影像数据所需要的波形[4]。





时序周期

t_{VBP}	2310
t_{VFP}	1925
t_{HBP}	30
t_{HFP}	25

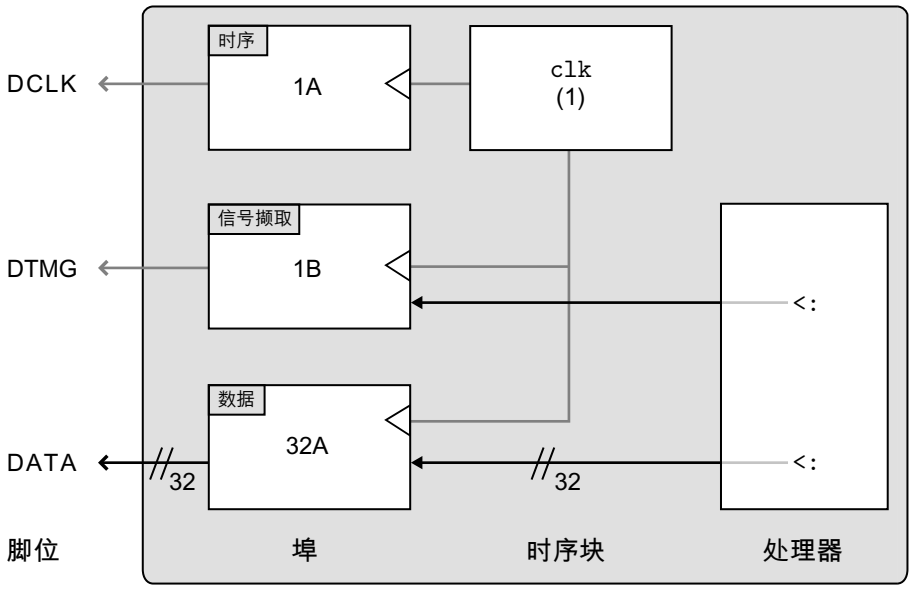
表格 1

这个屏幕解析度是 320x240 像素，它需要的像素数据以行的顺序提供，并且每个值都是在特定的时序边缘驱动，信号说明如下:

- DCLK 是由驱动程序产生的时序信号，必须设定在 4.85MHz 到 7.00MHz 的范围间，选到的值会决定屏幕刷新的速度。
- DTMG 是数据有效信号，每当数据传送的时候，该信号必须被驱动至高准位。
- DATA 会传载十八个位元的 RGB 像素数据到屏幕上。

规格规范每一行上的像素是以连续的周期驱动，并且行与行数据之间必须有 55 个周期的延迟，而每幅影像之间的延迟是 4235 个周期 (参见表 1)。

因为 LCD 屏幕本身时序的需求，它们通常是被专用的硬件元件驱动，而由于 XMOS 架构支援时序同步功能，用 XC 实作 LCD 屏幕的驱动程序是简单的，所需要的埠设定举例说明如下:



埠 DATA 和埠 DTMG 产生的时序皆来自内部，此时序也经由 DCLK 提供给外部使用，下面的程序定义一个以此方式来设定埠的函数:

```
#include <xs1.h>

out port DCLK = XS1_PORT_1A;
out port DTMG = XS1_PORT_1B;
out port DATA = XS1_PORT_32A;
clock    clk  = XS1_CLKBLK_1;

void lcdInit(void) {
  configure_clock_rate(clk, 100, 17); // 100/17 = 5.9MHz
  configure_out_port(DATA,  clk, 0);
  configure_out_port(DTMG, clk, 0);
  configure_port_clock_output(DCLK, clk);
  start_clock(clk);
}
```

程序所指定的时序频率是 5.9MHz，传送一幅影像所需要时间是：$320 * 240 + 240 * 55 + 4235 = 94235$ 时序，因此侦幅率是 $\frac{5.9}{94235} = 62$Hz；下面函数在规格定义的时序边缘依序输出像素的数据到 LCD 屏幕。

```
void lcdDrive(streaming chanend c, out port DATA,
              out port DTMG) {
  unsigned x, time;
  DTMG <:0 @ time;
  while (1) {
    time += 4235;
    for (int cols=0; cols<320; cols++) {
      time +=30;
      c :> x;
      DTMG @ time <: 1;      // 高撷取信号
      DATA @ time <: x;      // 像素 0
      for (int rows=1; rows<240; rows++) {
        c :> x;
        DATA <: x;           // 像素 1..239
      }
      DTMG @ time+240 <: 0; // 低撷取信号
      time += 25;
} } }
```

串流数据由通道端点输入，while 循环主体传送单一幅影像，并且外部的 for 循环主体传送每行数据。这个程序指示埠 DTMG 在开始输出一行数据的时候，将其本身脚位驱动至高准位，最后停止驱动。

另一个方法是设定埠 DATA 在 DTMG 产生信号撷取讯号 (参见 §6.4)，并且在代码中移除两个由处理器送到 DTMG 的输出。

4.5 时序行为摘要

以下总结有关于具时序性输入和输出埠 (未缓冲的) 的语义:

输出语句

- **输出**操作使数据在下个时序下降边缘驱动，输出会阻塞直到接下来的上升边缘。
- 当计数器等于其设定的时间，**时效性的输出**将数据由埠驱动，输出会阻塞直到这时间点之后的下一个上升边缘。
- 在没有新的输出数据的情形下，在时序边缘驱动的数据会连续地在接下来的边缘驱动。

输入语句

- **输入**使埠在埠的下一个时序上升边缘时进行数据取样，输入会堵塞直到运行完成的时间点。

- **时效性输入**使埠在其计数器等于所设定的时间进行数据取样，输入会堵塞直到运行完成的时间点。
- **条件式输入**使埠在每个时序上升边缘进行数据取样，一直持续到取样到的数据符合该条件，输入会堵塞直到运行完成的时间点，并且输入只取最新取样到的数据。

选择语句

选择语句会等候在其设置的情况之一是准备好的状态，完成相对应的输入操作，其中:

- 对于**输入**而言，埠在其时序周期的时间内至多会有一次是准备好的状态。
- 对于**时效性输入**而言，埠只有在其本身的计数器等于所指定的值时，才会是准备好的状态。
- 对于**条件式输入**而言，埠只有当取样的数据符合设定的条件时，才会是准备好的状态。
- 对**时效性的条件输入**而言，埠只有当其计数器等于或是大于所指定的时间时，才会是准备好的状态，并且是在取样的值符合条件的情况下。

对于纪录 t 的时戳性操作，线程可以输入或是输出的下一个可能时间是 $t+1$ 。

在 XS1 装置上，所有的埠都有缓冲区 (参见 §C.1)，产生的语义则是上述内容的延伸，并在下个章节中讨论。

第 5 章

埠之缓冲

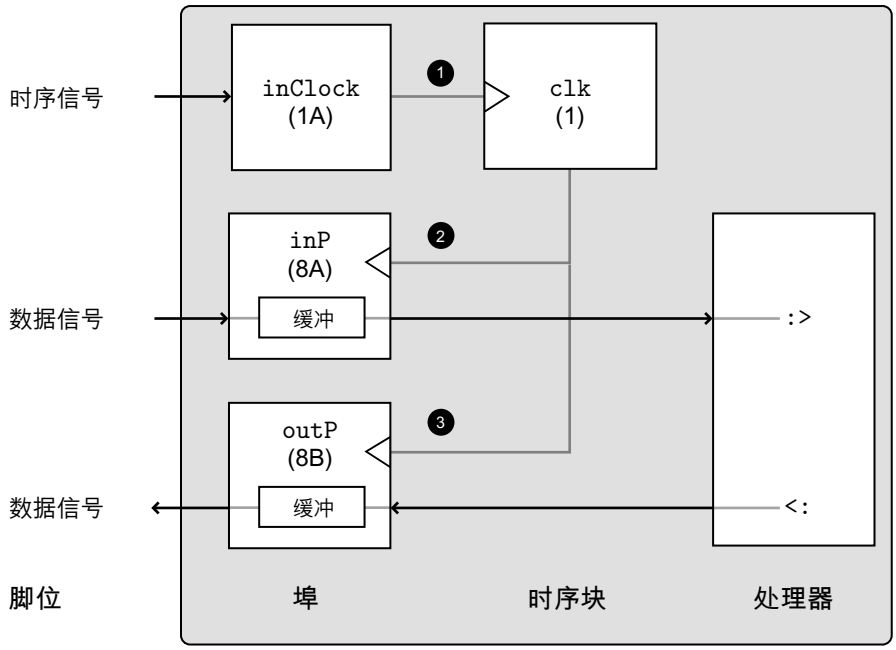
XMOS 架构中含有缓冲区，对于在有时序的埠上进行输入输出的程序，能改善其效能，缓冲区可以保留处理器欲输出的数据直到该埠的时序到达下一个下降边缘，这样允许处理器在这期间能够运行其他的指令；也可以储存埠取样到的数据直到处理器准备好输入数据；使用缓冲，线程可以在多个埠上并行地操作输入和输出。

5.1 使用有缓冲区的埠

下面的程序使用具有缓冲区的埠，来移除埠上的取样及驱动，与计算之间的耦合:

```
#include <xs1.h>

in  buffered port:8 inP     = XS1_PORT_8A;
out buffered port:8 outP    = XS1_PORT_8B;
in port             inClock = XS1_PORT_1A;
clock               clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port(inP, clk);
  configure_out_port(outP, clk, 0);
  start_clock(clk);
  while (1) {
    int x;
    inP  :> x;
    outP <: x + 1;
    f();
} }
```

此程序设定埠 inP、outP 和 inClock，如下所示:



以下代码:

```
in buffered port:8 inP = XS1_PORT_8A;
```

声明一个名称为 inP，是有缓冲的输入埠，该埠对应到八位元的埠标识符 8A。

❶ 这个语句:

```
configure_clock_src(clk, inClock);
```

设定一位元的输入埠 inClock 提供边缘给时序变量 clk。

❷ 这个语句:

```
configure_in_port(inP, clk);
```

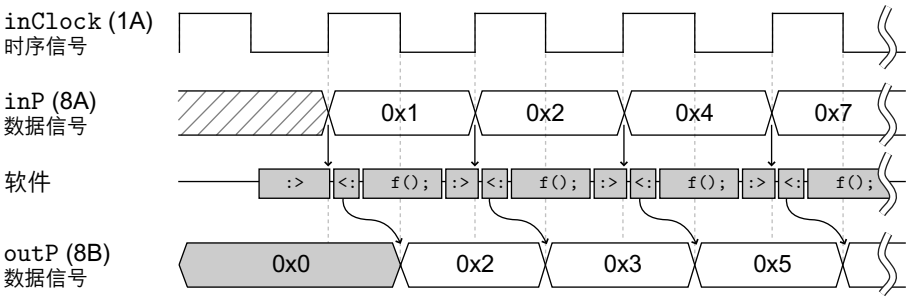
设定输入埠 inP 的时序来自时序 clk。

❸ 以下语句:

```
configure_out_port(outP, clk, 0);
```

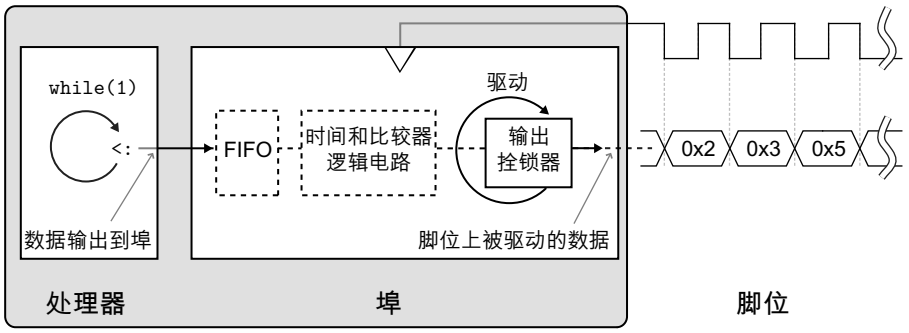
设定输出埠 outP 的时序来源是 clk，并且在其脚位上的初始值为 0。

下面的波形图说明范例输入信号以及这个程序预期的输出，也显示在 while 循环中被处理器运行的语句相对应波形。



被输入的前三个值是 0x1、0x2 和 0x4，并且回应的输出值为 0x2、0x3 和 0x5。

下面的图说明硬件的缓冲操作行为：



这个图说明处理器在运行 `while` 循环将数据输出到埠，如此，埠将数据留在缓冲区，当埠还在将前面数据送出的时候，处理器可以继续运行接下来的指令；在时序的下降边缘，埠会从其缓冲区中读入下个位元组数据，并且在其脚位上送出数据。只要循环中指令运行的时间少于埠的时序周期，则在每个时序周期会有一个新的值在脚位上驱动出。

第一个输入语句是在上升边缘之前运行，这表示输入的缓冲区还未被使用；在下个数据被取样之前，处理器会一直处于准备好输入数据的状态，这样会使处理器阻塞并等效于让本身慢下来至埠的速度；然而，如果第一个输入发生在第一个值被取样之后，输入的缓冲区会保留该数据一直到处理器准备好接受为止，每一个输出都会阻塞直到前一个输出数据被送出。

时效性操作表示将来的时间，这个时间和比较器逻辑允许时效性的输出到缓冲区，但是对于时效性输入以及条件式输入，缓冲在输入操作之前会被清空。

5.2 同步多个埠上有时序的输入/输出

藉由设定多个具缓冲的埠有相同的时序来源，可以在一个线程中，使数据并行地在这些埠上取样和驱动，下面的程序首先将自己和时序周期起点同步，如此，在下个下降缘之前能有最长的时间；接着送出一序列的八位元字节数据到被并行驱动的四位元埠。

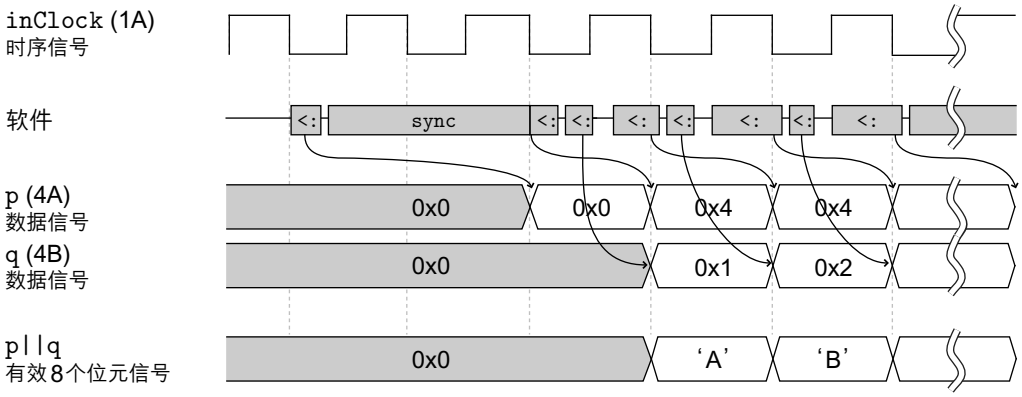
```
#include <xs1.h>

out buffered port p:4     = XS1_PORT_4A;
out buffered port q:4     = XS1_PORT_4B;
in            port inClock = XS1_PORT_1A;
clock         clk          = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_out_port(p, clk, 0);
  configure_out_port(q, clk, 0);
  start_clock(clk);
  p <: 0;  // 开始输出
  sync(p); // 与下降边缘同步
  for (char c='A'; c<='Z'; c++) {
    p <: (c & 0xF0) >> 4;
    q <: (c & 0x0F);
  }
}
```

这个代码中的语句:

```
sync(p);
```

使处理器等待下一个下降边缘，也就是缓冲区的上个数据已经在一个完整的周期中驱动，这样能确保下个指令，能在下降边缘后马上运行，使接下来循环中的两个输出语句，能在相同的时序周期运行。

下图显示处理器输出以及被埠驱动的数据:



与上升边缘同步的方式建议是使用标准库函数 clearbuf 清除缓冲区数据，然后进行一个输入操作。

5.3 缓冲行为摘要

在有缓冲和时序的埠上，输入/输出语义摘录如下:

输出语句

- 输出操作会将数据插入埠的 FIFO，只有当 FIFO 满的状态下，处理器才会进入等待状态。
- 在埠本身的一个时序周期中，最多一笔数据会从埠的 FIFO 中移除和送出。
- 当埠的计数器等于所指定的时间，有时效性的输出会将数据插入至埠的 FIFO 等待送出，只有当 FIFO 满的时候，处理器才会进入等待状态。
- 时戳性的输出会使处理器等待到输出被送出为止 (需要决定时戳的值)。
- 在边缘上驱动的数据会持续在接下来的边缘进行。

输入语句

- 在埠的一个时序周期中，最多只有一个值会被埠取样并插入到其本身的 FIFO，如果 FIFO 已经满了，最老的值会被丢弃，以让出空间给最新取样的值。

- 输入操作会从埠的 FIFO 移除数据；只有当 FIFO 是空的时候，处理器才会进入等候的状态。
- 时效性及条件式的输入会丢弃任何在 FIFO 的数据，而之后的行为和没有缓冲的情况相同。

第 6 章

串行化和信号撷取

XMOS 架构对于通信协议中常见的操作提供硬件的支援，埠可以被设定来操作**串行化**，当数据必须在只有几个位元宽的埠上通讯时是很有用的 (如 §2.5 中的例子)，也可被设定来进行**信号撷取**，当数据伴随着独立的数据有效信号时很有帮助 (如 §4.4 的例子)；将这些任务移到其他的埠能腾出处理器的时间来运行计算。

6.1 使用埠串行输出数据

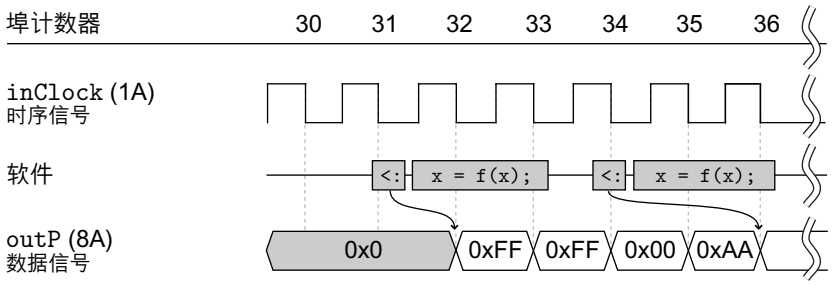
有时效性的埠可以将数据串行化，以减少操作输出所需的指令数目；下面程序输出一个 32 位元的数值到八个脚位，并使用一个时序来决定每八个位元值被驱动的时间。

```
#include <xs1.h>

out buffered port:32 outP    = XS1_PORT_8A;
in           port    inClock = XS1_PORT_1A;
clock                clk     = XS1_CLKBLK_1;

int main(void) {
  int x = 0xAA00FFFF;
  configure_clock_src(clk, inClock);
  configure_out_port(outP, clk, 0);
  start_clock(clk);

  while (1) {
    outP <: x;
    x = f(x);
  }
}
```

下面代码:

```
out buffered port:32 outP = XS1_PORT_8A;
```

声明埠 outP 从 32 位元的**位移寄存器**驱动到八个脚位，类型 port:32 指每个输出操作的传输位元数量 (**传输宽度**)，XS1_PORT_8A 这个初始化指定连接到埠的实体脚位的数量 (**埠宽**)，下面的波形图说明这个程序送出的数据:



如果将串行化移至埠，处理器只需要在每四个时序周期进行一次输出；在每个时序下降边缘，位移寄存器的最低有效八个位元会被送到脚位上，然后位移寄存器在向右移八个位元。

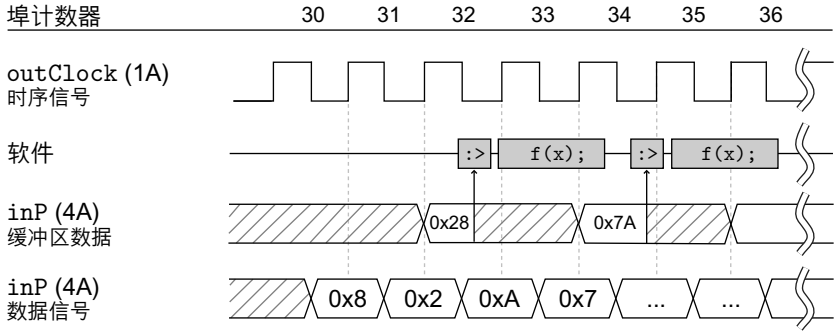
在 XS1 装置上，用来串行化的埠必须是用关键字 buffered 修饰，细节参见 §C.1 的说明。

6.2 使用埠反串行化输入数据

埠可以反串行化数据，减少输入数据所需的指令数量；下面的程序在时序为 25MHz 的输入埠上，运行四位元转八位元的操作。

```
#include <xs1.h>
in  buffered port:8 inP      = XS1_PORT_4A;
out          port   outClock = XS1_PORT_1A;
clock               clk25    = XS1_CLKBLK_1;

int main(void) {
  configure_clock_rate(clk25, 100, 4);
  configure_in_port(inP, clk25);
  configure_port_clock_output(outClock, clk25);
  start_clock(clk25);
  while (1) {
    int x;
    inP :> x;
    f(x);
} }
```

这个程序声明 inP 是个四位元宽的埠，并且其传输宽度是八位元，也就是该埠可先取样两个四位元值，然后才被处理器读取，如同输出，反串行器缩减取得数据所需要的指令数量；下面的波形图显示范例输入信号以及在埠中存于缓冲区的可用数据的期间。



数据是在时序的上升边缘取样的，并且当位移的时候，最低有效半字节会先读取，而取样到的数据会停在埠的输入缓冲区两个时序周期，前面的两个输入值是 0x28 和 0x7A。

6.3 伴随有效信号的数据

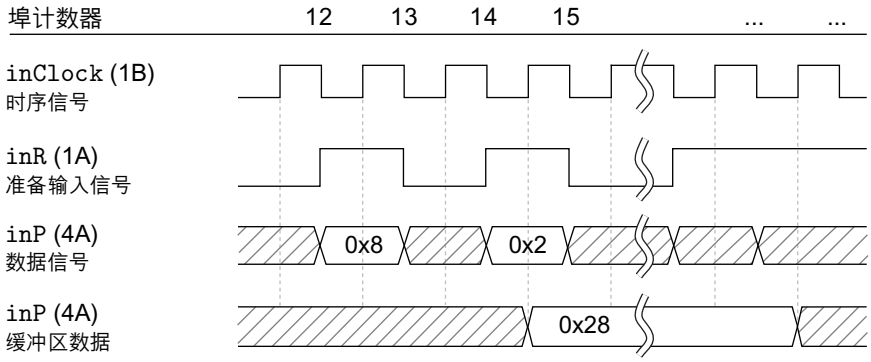
有时效性的埠能解译**准备输入**撷取信号，以决定伴随数据的有效性，下面的程序只在准备输入信号为高的时候，才由有时效性的埠输入数据:

```
#include <xs1.h>

in buffered port:8 inP     = XS1_PORT_4A;
in          port   inReady = XS1_PORT_1A;
in          port   inClock = XS1_PORT_1B;
clock              clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_in_port_strobed_slave(inP, inReady, clk);
  start_clock(clk);

  inP :> void;
}
```

下面代码:

```
configure_in_port_strobed_slave(inP, inReady, clk);
```

设定输入埠 inP 只在当埠 inReady 的值等于 1 时才取样，有准备输入信号的埠必须是一位元宽；下面波形图显示范例输入信号以及这个程序读进的数据。



每当准备输入信号为高准位时，数据会在时序上升边缘被取样，埠会取样两个四位元的值，并且将他们结合产生单一个八位元的值给处理器读取，输入的数据是 0x28。XS1 装置有个单一栏位的缓冲区，也就是在接下来的时序上升边缘，准备输入信号有两次为高准位的状态，才可以输入数据。请注意埠计数器在每个时序周期都会被加总，无论信号撷取的准位高低。

6.4 输出数据和数据有效信号

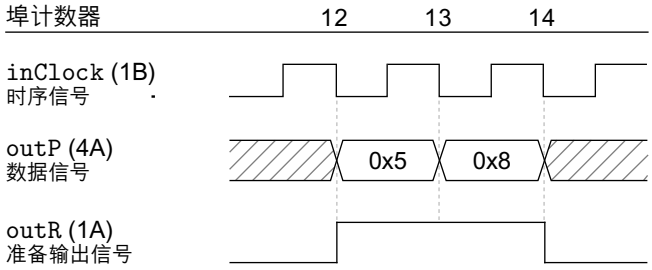
每当输出数据的时候，有时序的埠可以产生**准备输出**撷取信号，下面的程序会使数据在四位元的埠上驱动时，在输出埠上驱动数据有效信号。

```
#include <xs1.h>

out buffered port:8 outP    = XS1_PORT_4B;
out           port  outR    = XS1_PORT_1A;
in            port  inClock = XS1_PORT_1B;
clock               clk     = XS1_CLKBLK_1;

int main(void) {
  configure_clock_src(clk, inClock);
  configure_out_port_strobed_master(outP, outR, clk, 0);
  start_clock(clk);

  outP <: 0x85;
}
```

下面语句:

```
configure_out_port_strobed_master(outP, outR, clk, 0);
```

设定输出埠 outP 在输出数据时，驱动埠 outR 至高准位，这个准备输出埠必须是一位元宽；下面的波形图显示这个程序驱动的数据以及撷取信号。



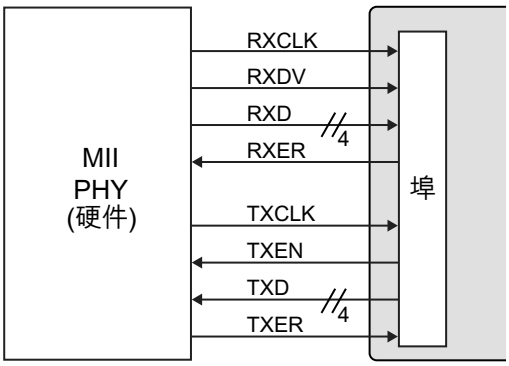
埠在两个时序周期驱动两个四位元的值，并在这期间将准备输出信号拉高。

也可以实作控制流程演算法，使用准备输入撷取信号来输出数据，以及使用准备输出撷取信号来输入数据，当设定好这两个信号，埠实作对称的信号撷取通信协议，使用时序来进行数据通讯的信号交换 (参见 §C.2.2 和 §B.2)。

在 XS1 装置上，用来撷取信号的埠必须以关键字限定符 `buffered` 修饰，参见 §C.1 中进一步的详细说明。

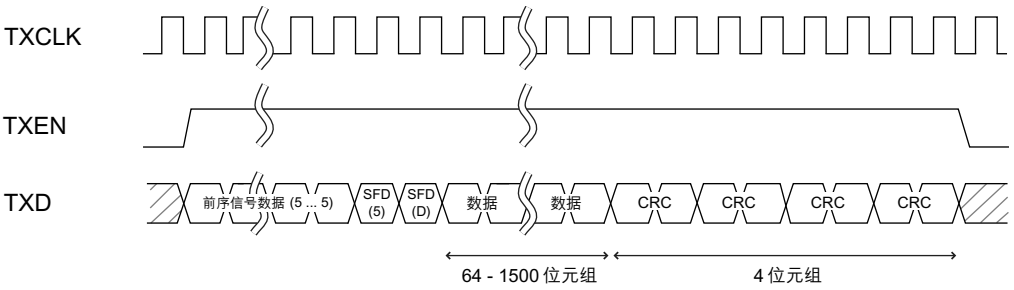
6.5 范例研讨：以太网 MII

可以用 XS1 装置上的单一个线程来实作全双工的 100Mbps 以太网媒体独立介面通信协议[5]，该通信协议实作在连接层和实体装置之间数据传送的信号，信号显示如下:



6.5.1 MII 传输

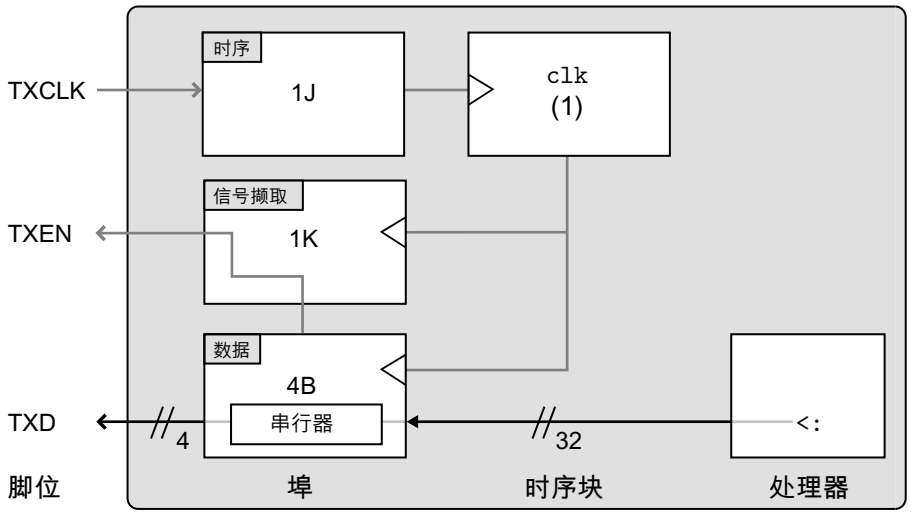
下面波形图显示单一个框讯的数据传输到 PHY，为了简化，下图省略取得错误信号 TXER。



信号如下所示:

- TXCLK 是由 PHY 产生自由振荡的 25MHz 时序。
- TXEN 是在框讯传送期间，被传送器驱动至高准位的数据有效信号。
- TXD 在每个时序周期从传送器运送半字节数据至 PHY，传送器会先传送前序信号，此信号为一串值为 0x5 的半字节，接着是两个半字节，值分别为 0x5 和 0xD; 这个数据必须是在 64 和 1500 位元组之间才能被传送，最低有效位元先送，接着是含有 CRC 的四个位元组。

下图说明串行化输出数据并产生数据有效信号所需要的埠设定:



埠 TXD 在其脚位上进行 32 转 4 位元的串行化数据操作，此操作是和一位元的埠 TXCLK 同步，并且使用一位元的埠 TXEN 当作是准备输出撷取信号，每当数据来的时候，该信号被驱动至高准位。在这个设定里，处理器只需要在每八个时序周期输出一次数据，并且不需要特别输出数据有效信号，下面这个程序以这样的方式定义埠:

```
#include <xs1.h>
out buffered port:32 TXD   = XS1_PORT_4B;
out          port    TXEN  = XS1_PORT_1K;
in           port    TXCLK = XS1_PORT_1J;
clock                clk   = XS1_CLKBLK_1;

void miiConfigTransmit(clock clk,
  buffered out port:32 TXD, out port TXEN) {

  configure_clock_src(clk, TXCLK);
  configure_out_port(TXD, clk);
  configure_out_port(TXEN, clk);
  configure_out_port_strobed_master(TXD, TXEN, clk, 0);
  start_clock(clk);
}
```

下面的函数由其他的线程读进框讯数据并且输出至 MII 埠，为了简化，以下程序忽略取得错误信号以及 CRC。

```
void miiTransmitFrame(out buffered port:32 TXD,
                      streaming chanend c) {
  int numBytes, tailBytes, tailBits, data;

  /* 输入下个封包的大小 */
  c :> numBytes;
  tailBytes = numBytes / 4;
  tailBits = tailBytes * 8;

  /* 输出一串 0x5 之后输出 0xD */
  TXD <: 0xD5555555;

  /* 串行化输出 32 个位元的位元组 */
  for (int i=0; i<numBytes-tailBytes; i+=4) {
    c   :> data;
    TXD <: data;
  }

  /* 串行化输出数据剩馀的位元 */
  if (tailBits != 0) {
    c :> data;
    partout(TXD, tailBits, data);
  }
}
```

这个程序首先从通道 c 读入框讯大小的位元组数，接着输出 32 位元的前序信号至 TXD，此信号在脚位上是在八个时序周期中以半字节的形式传送的；for 循环每个反覆会使下个 32 位元的数据被送至 TXD，以串行化到脚位上，这使处理器在下个数据区段必须被传送之前，有足够的时间处理循环。

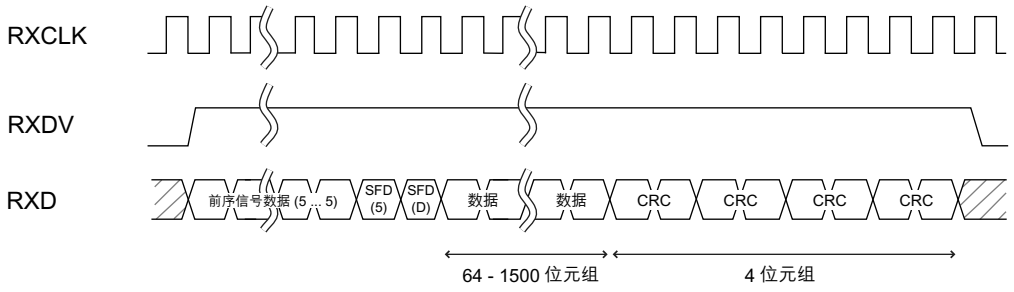
最后的语句:

```
partout(TXD, tailBits, data);
```

是个**部分输出**操作，传送的数据是 data 剩馀的有效位元。

6.5.2 MII 接收

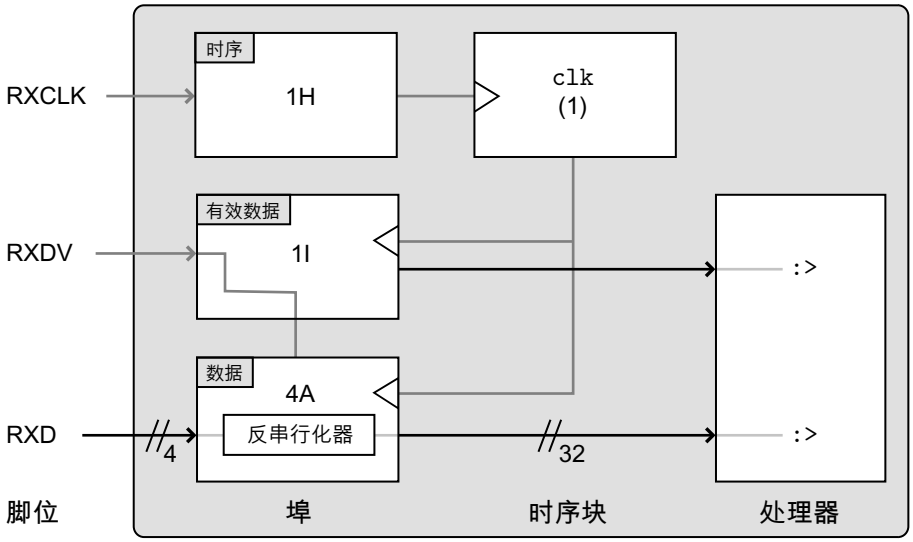
下面的波形图显示接收由 PHY 来的单一页框数据的波形，为了简化，此图省略取得错误信号的 RXER:



信号如下所示:

- RXCLK 是由 PHY 产生的自由振荡时序。
- RXDV 是在页框数据传送期间，被 PHY 驱动至高准位的数据有效信号。
- RXD 在每个时序周期从 PHY 运送半字节至接收器，接收器会等待前序信号，此信号为一串数值为 0x5 的半字节，接着是两个半字节，值分别是 0x5 和 0xD，然后才收到实际数据，数据的范围是在 64 和 1500 位元组之间，最低有效半字节先传，接着是含有 CRC 的四个位元组。

下图说明当有数据有效信号时，反串行输入数据所需的埠设定:



埠 RXD 将由其脚位而来的数据进行反串行化，将四位元转为三十二位元，此动作是和一位元宽的埠 RXCLK 同步的，并且使用一位元埠 RXDV 当成是准备输入撷取信号，该信号用在撷取信号为高准位状态时取样数据；在这个设定中，在处理器必须将数据输入之前，埠可以取样八个值，且处理器不需要等待数据有效信号，下面的程序以这样的方式设定埠:

```
#include <xs1.h>

in buffered port:32 RXD   = XS1_PORT_4A;
in          port    RXDV  = XS1_PORT_1I;
in          port    RXCLK = XS1_PORT_1H;
clock               clk   = XS1_CLKBLK_1;

void miiConfigReceive(clock clk, in port RXCLK,
  buffered in port:32 RXD, in port RXDV, in port RXER) {

  configure_clock_src(clk, RXCLK);
  configure_in_port(RXD, clk);
  configure_in_port(RXDV, clk);
  configure_in_port_strobed_slave(RXD, RXDV, clk);
  start_clock(clk);
}
```

下面函数接收单一个无错误页框数据，并且将它输出至另一个线程，其中取得错误信号以及 CRC 被省略以助于理解。

```
#define MORE 0
#define DONE 1

void miiReceiveFrame(in buffered port:32 RXD, in port RXDV,
                     streaming chanend c) {
  int notDone = 1;
  int data, tail;

  /* 等待页框的开始 */
  RXD when pinseq(0xD) :> void;

  /* 接收页框数据以及 crc */
  do {
    select {
      case RXD :> data :
        /* 输入接下来的 32 个位元数据 */
        c <: MORE;
        c <: data;
        break;
      case RXDV when pinseq(0) :> notDone :
        /* 输入剩馀在埠中的数据 */
        tail = endin(RXD);
        for (int byte=tail>>3; byte > 0; byte-=4) {
          RXD :> data;
          c <: MORE;
          c <: data;
        }
        c <: DONE;
        c <: tail >> 3;
        break;
    }
  } while (notDone);
}
```

处理器会先等候埠 RXD 取样到前序信号的最后一个半字节 (0xD)，接着在循环每次的反覆中，等待 RXD 要输入的八个半个字节数据或是数据有效信号 RXDV 变成低准位。

同时使用埠的串行化和信号撷取功能的结果是：在接收到一个完整的传输宽度的数据之前，准备输入信号可能会转为低准位。

以下语句:

```
tail = endin(RXD);
```

会使埠 RXD 回应尚未输入的剩馀位元数，它也会使埠在接下来的输入提供这数据，即使在数据有效信号仍然是低准位或是位移寄存器还没满的状况。

XS1 装置提供单一栏位的缓冲区，最大宽度为 32 位元和一个 32 位元的位移寄存器，所以，每当数据有效信号为低准位时，最多可有 64 位元的数据，须用两个输入语句来输入。

6.6 摘要

在串行埠上输入/输出的语义如下 (其中 p 是指埠宽而 w 指埠的传输宽度):

- 一个 w 位元的输出在 $\frac{w}{p}$ 个连续时序周期上传送，以最低有效位元先传送，准备输出信号在每个周期中皆被驱动至高准位。
- 对于有时效性输的出，埠会等到其计数器等于所给定的时间才**开始**将数据串行，准备输出信号在等待串行的时间不会被驱动。
- 一个 w 位元的输入值会在 $\frac{w}{p}$ 个时序周期取样，先接收到的位元是 w 的最低有效位元 (如果有使用准备输入信号，时序周期可以是不连续)。
- 对于有时效性的输入，埠在其计数器等于所给定的时间时，提供所取样到的数据的**最后** p 个位元。

如果将埠设定为使用准备输入信号:

- 当准备输入信号为高准位时，数据只在埠的时序上升缘时取样。

如果将埠设定为使用准备输出信号:

- 准备输出信号和数据会一起被驱动至高准位，并且会被保存住单一个时序周期的时间。

信号撷取信号以及串行化的语义在附录 B 有完整描述。

附录 A

XC 程序语言规格

这份附录说明 XC 版本 9.9 的规格，埠上的输入/输出操作的行为则于附录 B 中另有说明。

> 这份手册的编排和其中部分文字是基于 K&R 所着对 C 语言定义一书而来[6]，对于 XC 语言和 C 语言之间差异的说明则以缩排及较小字体标注。

A.1 词法约定

一个程序包含储存在文件中的一个或是多个**编译单元**，是由好几个阶段的翻译而来，其中的说明收录在 §A.13 单元中。第一个阶段是低阶词法转换，对于代码中以 # 为开始字符的指令进行宏定义以及展开；当预置处理完成时，程序已经缩减成标记的序列。

A.1.1 标记

标记被区分为六类：标识符、关键字、常量、文字串、运算符和分隔号；空格字符、水平或垂直制表符以及换行字符统称为**泛空白字符**，这些字符是被用来分隔标记，因此是被忽略的；部分的泛空白字符是必须存在的，用来隔开两个相邻的标识符、关键字以及常量。

A.1.2 注解

注解的方式有两种：其一是使用字符 /* 为注解的开始并以 */ 结束注解，另一种是以 // 开始，并结束于换行字符；注解不能以巢状结构形式存在，也不能在字符串或是文字串中间。

A.1.3 标识符

标识符是一个由字母、数字以及底线 (_) 组成任意长度的序列，其中第一个字符不能为数字，大写和小写的字母表示不同的标识符。

A.1.4 关键字

以下的标识符保留为关键字的使用，不能有其他用途:

```
auto        else        return      union
break       enum        short       unsigned
case        extern      signed      void
char        for         sizeof      volatile
const       if          static      while
continue    int         struct
default     long        switch
do          register    typedef
```

以下的标识符也是保留为关键字的使用，不能有其他用途:

```
buffered    inline      out         slave
chan        isnull      par         streaming
chanend     master      port        timer
core        null        select      transaction
in          on          service     when
```

埠的结构 `port:`*n* 也是标识符，其中 *n* 是由数字组成，该数字是十进制，并且解译为整数常量，下面的标识符由于相容性的关系，保留至将来使用:

```
accept      claim       float       restrict
asm         double      module
```

A.1.5 常量

常量有好几种类型，每种都有数据类型，在 §A.3.2 当中，有讨论基本类型。

```
constant  ::=  integer-constant
           |   character-constant
           |   enumeration-constant
           |   null
```

浮点常量是不支援的。

A.1.5.1 整数常量

一组数字所组成的数值如果是以 0b 或 0B 开头，表示二进制数值，如果开头是 0，则是八进制；而用 0x 或 0X 开头是十六进制。整数常量也可以用字母结尾，像是 u 或 U (unsigned)，以及 l 和 L (long)，或者两者一起使用 (unsigned long)。

整数常量的类型是依其形式、数值以及字尾而定 (参见 §A.3 中类型的讨论)，对没有字尾字母的十进制常量，它的可能类型依序是 int、long int、unsigned long int，如果是没有字尾字母的八进制或是十六进制常量，可能的类型则是 int 、unsigned int、long int、unsigned long int；无号数可能类型是 unsigned int、unsigned long int，而长整数常量则是 long int、unsigned long int。

A.1.5.2 字符常量

字符常量是由一个或是多个字符组成的，并且以单引号括住 (不含单引号以及换行字符)，字符常量的值是该字符在运行时于所使用的机器中的符号集合里所对应的值，多字符的数值则是实作定义。

宽字符常量不支援。

下列的转义字符不支援:

换行字符	NL	\n	反斜线字符	\	\\
水平制表符	HT	\t	问号	?	\?
垂直制表符	VT	\v	单引号	'	\’
退格符	BS	\b	双引号	"	\"
回车符	CR	\r	八进制数值	*ooo*	*ooo*
换页字符	FF	\f	十六进制数值	*hh*	\x*hh*
铃响字符	BEL	\a			

转义序列 *ooo* 需要使用一个、两个或是三个八进制数字来表示，而 \x*hh* 则需要一个或是多个十六进制数字；如果这些数字的值超过最大的字符值，其行为是未定义的，会根据实作定义而有不同。对于八进制或是十六进制的转义字符，如果 char 被视为带号的字符，其数值则是由正负号扩展而得，就如同强迫转型到 char 类型一样；若是上述未提到的其他字母接在 \ 之后，则其行为是未定义的。

A.1.5.3 枚举常量

标识符若被声明为枚举形式 (参见 §A.7.5)，其类型是 int 常量。

A.1.5.4 空字符常量

空字符常量的类型是 null。

A.1.6 文字串

文字串是由零个或是多个字符组成，并且以双引号括住 (不包含双引号以及换行字符)，它的类型是字符数组，而存储类是 `static` (参见 §A.3.1)，其初始化值是所给定的字符串；写法相同文字串是否被视为相同的常值是实作定义，而企图变更文字串会导致无法预期的行为。

相邻的文字串可以联结成另一个字串，联结后，一个空字符 `\0` 会被加在联结的字串之后，所有转义字符皆有支援。

A.2 语法记号

在本手册中使用的语法记号，语法类型是以 Roman 字体表示，一些单字和字符则是以打字机字样呈现，至于可省略的的终端符号和非终端符号由一个写在右下方的“*opt*”表示，例如:

$$\{\ \text{expression}_{opt}\ \}$$

就表示花括号中的表达式不是必要的，术语“没有或是多个”与“一个或是多个”则分别用尖形括号和星号 (∗) 与加号 (+) 一起表示，例如:

$$\langle \text{declaration} \rangle^{*}$$

表示零个或是多个声明的序列，而

$$\langle \text{declaration} \rangle^{+}$$

是指一个或是多个声明序列。

A.3 识别字的意义

标识符 (或是名称) 是指函数、结构的标签、联合、结构或是联合的成员以及对象的总称，对象 (或是变量) 是个储存位置，它的意义取决于两项特性，一个是其**存储类**，另一个是**类型**，存储类决定标识符的储存有效生命期，其类型决定该标识符在辨识体上所占用的数值意义；识别名称也有其作用域，在有效程序区段内，这个识别名称是已知的。识别名称尚有其连结关系，它决定相同识别名称是否指到相同的函数或是对象；关于作用域和连接的讨论收录在 §A.10 中。

A.3.1 存储类

对象的存储类可以是**自动型**或是**静态型**其中一种，自动类型的对象是局部有效的区段 (§A.8.4)，在一个区段中的声明，如果没有描述其存储类，或使用 `auto` 或 `register`，则会新建成自动类型。

静态的对象可以是局部有效或是所有外部区段皆有效，但是不管哪种情况，在下次进入时，其值皆会保有上次离开时的值。在一个区段内，静态的对象是以关键字 `static` 声明，若对象声明是在函数外部，和函数定义同一层，则属于静态的对象；使用 `static` 关键字，可以使其对某个特定的编译单元是局部有效的，也就是赋予只在该文件有效的特性 (或是内部连结用途)。如果未使用清楚的存储类或是关键字 `extern`，则对象对整个程序是全局性，也就是给予其程序整体作用域 (或是外部连结)。

函数可以被声明为服务，该性质对于函数的行为并没有任何作用；对于该类型函数的作用则是由实作定义。

A.3.2 基本类型

声明为 char 的对象，其容量是足够存放使用的机器中的字符集合中任何字符，如果将该集合中的字符存放在 char 的对象，其值是等价于该字符的正整数值，其他的数值可能存放在 char 变量中，但是数值的范围，特别是数值是否是带号数，则是实作定义。

声明为 unsigned char 的对象，占有和一般字符相同空间，但必须是非负数，以 signed char 声明的带号字符也和前两者相同，皆使用相同空间。

除了 char 类型之外，有三种不同大小的整数声明：short int、int 和 long int，一般的 int 对象，使用机器硬件架构中用来存放一个整数的容量，而长整数类型则会使用至少和短整数一样多的空间，但是实作上，一般会将整数类型视为长整数或是短整数，整数 int 类型皆表示带号数值，除非有另外指定。

无号整数是由 2^n 的算术模数规则得来，其中 n 表示这个数值需要的位元数，存放在带号对象的非负数值是所有可以存在相对的无号对象数值的子集合，而重复数值的表示方式是相同的。

所有上面的类型统称为**算术**类型，因为这些可以解读为数值或整数的类型，也因为这些表示整数值。

类型 void 指定一个空集合的数值，是被用来当成无回传值的函数类型。

> 在 C 规格中，类型 long long int 、 float 和 double 是不支援的。

类型 chan 指定一个逻辑的通讯通道，在该通道上，数值可以在两个并行运行的语句中传递通讯；而 chanend 类型指定一个通道端点。

一个 chan 意指最多两个端点，其位置 (本身则是通道端点) 是透过通道的使用时定义，一个通道最多只能用于两个并行语句中 (§A.8.8)。

通道端点是被当成输入和输出语句的运算元 (§A.8.3) 使用，通道是双向并且同步的，也就是说：在数据传输之前，输出会等待对应的输入完成，才进入准备好的状态，而串流通道是同步或者异步，则是实作定义。

类型埠 port 指定一个 p 个位元的寄存器，和一群 p 个脚位接口，用来与环境通讯，其中 p 是实作定义；而 port:n 类型指定一个 n 个位元的寄存器，和 p 个脚位接口，也用来与环境通讯 (其中 p 和 n 必须是不相等)。类型 void port 是埠的一种特别类型，不能用来输入或是输出，埠有名义的**传输**类型和**计数器**类型 (参见 §A.8.3)，这些类型皆为实作定义。

埠是被当成输入和输出语句的运算元 (§A.8.3，附录 B) 使用。

类型 timer 是输入埠的一种特殊类型，该类型回传输入来时的时间，void timer 是计时器，不能当成输入使用，计时器也有名义的**传输**类型和**计数器**类型 (参见 §A.8.3)，这些类型是实作定义。

核心类型指定一个处理器核心，让埠和并行语句摆放，core 类型的对象没有储存位置。

通道端点、埠、计时器和核心统称为**资源**类类型，除了核心没有保留其储存空间之外，其他资源类类型的对象指到储存空间的一个位置，该位置是记录该资源的标识符。

> chan、chanend、port、timer、core、buffered 和 streaming 是新增的定义。

A.3.3 派生类型

除了基本类型之外，派生类型可以用下列的方式建构:

- 给定类型的对象**数组**。
- 回传给定类型对象的**函数**。
- 给定类型对象的**位址**。
- 含有一序列不同类型的**结构**。
- 可以包含不同类型的对象的任一种**联合**类。
- 含有不同类型一序列对象的**串列**。

一般而言，这些建构对象的方法可以递归地使用。

串列类型是用在多重赋值式语句 (§A.8.2) 中；指针则以位址取代 (参见 §A.6.1、 §A.6.3.2、 §A.7.7.2)。

A.3.4 类型限定符

对象的类型通常可以再加上限定符 const，表示其值将不会变更，加上限定符之后，不会改变它的数值作用域以及算术运算的性质。

埠可以用 in 或是 out 限定，说明只会被用来当成输入或是输出运算符 (§A.8.3)。

限定符在 §A.6.3.2 和 §A.7.2 当中讨论。

in 和 out 是新增的限定符。

A.4 对象和左值

对象是命名的储存区域，而**左值**是对对象的参考，例如：如果 str 是一个类型为“一维字符数组”的标识符，则 str[0] 便是个左值，指到这个字符对象，也就是位于数组 str 中第一个索引值元素。

可修改的左值是一个可以被修改的左值，其本身不能是数组，也不能是资源类或是不完整类型或是函数，且其类型必不能有 const 限定，如果是结构或是联合，则所有成员或是递归的结构中子成员皆不能含有限定符 const。

A.5 类型转换

一些运算符会根据其运算元使运算元的值从一种类型转换至另一种类型，这个单元解释这些转换预期的结果， §A.5详细说明大多数运算符所需要的类型转换。

A.5.1 整数提升

字符或是短整数不管是带号或是无号，可以用于使用整数的表达式中；如果 int 可以表示原本类型的所有的数值，则数值则被转成 int 类型，否则则转换成 unsigned int。

A.5.2 整数类型转换

任何转换成一个给定的无号类型整数，是以该无号数类型可以表示的最大数加 1 之后为模，找到与该整数馀数相同的最小非负值，当任何整数被转成带号类型时，如果可以用新的类型表示，则其值是不改变的；若不是，则是实作定义。

A.5.3 算术转型

许多运算符会转换其运算元为一个共用的类型，也就是结果的类型，用来进行这些**一般的算术转型**方式如下:

- 首先，整数提升是在两个运算元上进行。
- 如果其中一个运算元是 unsigned long int，另一个则被转型成 unsigned long int。
- 否则，如果其中一个运算元是 long int，另一个是 unsigned int，若是 long int 可以表示所有 unsigned int 的值，则 unsigned int 运算元被转型为 unsigned long int。
- 否则，如果其中一个运算元是 long int，另外一个会被转型成 long int。
- 否则，如果其中一个运算元是 unsigned int，另一个会被转型为 unsigned int。
- 否则，两个运算元都是 int。

A.5.4 void

空的对象的 (不存在) 的值可以用于任何方式，并且可以明确地或是不明确地转型至非 void 类型，void port 或是 void timer 类型的对象可以用在输入或是输出。

A.6 表达式

表达式的运算符优先顺序和这个单元的主要子单元顺序相同，以高优先顺序的先开始，在每一个单元，运算符有相同的优先顺序，用到的运算符的左向右或是右向左结合性会在每个子单元中说明。

运算符的优先顺序及结合性有完整的规范，表达式求值的顺序则没有，会有些例外情况影响程序的行为，即使是子表达式也有边际效应，特别是一个变量在表达式的一部分被修改，除非有特别处理，否则不会出现在表达式另外的部分，这个规则递归地用在所有用于在函数中被表达式调用的变量的变更。对于溢位、除式检查及表达式求值的其他异常则是实作定义。

A.6.1 产生参考

如果表达式的类型是“T 的数组”，对一些类型 T，表达式的值则是数组的参考，并且表达式的类型变更为“T 的参考”。

A.6.2 初级表达式

主表达式包含变量参考、函数调用、常量、字串、以及在括号内的表达式:

```
primary-expression  ::=  variable-reference
                      |  function-call
                      |  constant
                      |  string
                      |  ( expression )
```

变量参考是初级表达式，具有如下说明的适当变量声明 (§A.6.3) 的标识符，标识符的类型是在其声明时指定，而表达式类型则是标识符的类型。

函数调用也是初级运算，表达式的类型依函数回传类型而定 (§A.6.3.2)。

常量也是初级运算，表达式类型是常量 (依据其型式而来，型式于 §A.1.5 单元中讨论)。

文字串是初级表达式，其类型是“字符数组”。

有括号的表达式也是初级表达式，类型和值与纯粹的表达式相同。

A.6.3 后置式表达式

后置式表达式中的运算符具有由左向右的顺序关联性。

```
postfix-expression  ::=  primary-expression
                      |  variable-reference ++
                      |  variable-reference --

variable-reference  ::=  identifier
                      |  variable-reference [ expression ]
                      |  variable-reference . identifier
                      |  ( variable-reference , type-name )

function-call       ::=  identifier ( expression-list_opt )

expression-list     ::=  expression
                      |  expression , expression-list
```

A.6.3.1 数组参考

在变量参考后接着一个中括号刮住的表达式，代表数组中某特定元素的参考，这个变量参考必须是“*n* 个 T 的数组”类型或是“*n* 个 T 的数组参考”，其中 *n* 是数组维度大小，T 是某个类型，

并且该表达式必须是整数类型；下标的变量参考类型是 T，如果值小于零或是大于等于 *n*，则表达式是无效的，进一步细节，参见 §A.7.7.1。

A.6.3.2 函数调用

函数调用是标识符后面跟着括号，括号中含有一系列以逗号分开的表达式，这些组成了函数的引数，如果标识符有“回传空的交易函数”类型，其调用必须是在交易语句 (§A.8.9) 的作用域间，否则，标识符类型必须是“回传 T 的函数”或是“回传 T 的 `select` 函数”，T 是某个类型，也就是调用的函数值的类型是 T。

函数声明具有仅对文件有效的限制 (§A.9)，不支援隐含的函数声明 (参考 K&R § A7.3.2)。

术语**引数**是指被函数调用传递的表达式，而**参数**则是指被函数定义接收的输入对象 (或是其标识符)，或是在函数声明中的语句。

如果参数的类型是“指到 T 的参考”，则其引数传递形式是传址，否则是传值。在准备让函数调用时，传值而来的每个引数皆被复制一份，函数可以对这些复制而来的参数对象进行修改，但是这些修改不会影响原本引数的值；至于由传址而来的对象，函数则可以改变这些对象参考的值，因此影响这些引数的值；因为不相交性检查的目的，用传址传递对象视为对该对象进行写入，除非这个参数的类型是有 `const` 限定符限定。

对于传值而来的引数，如果该引数提升后的类型是参数本身没有提升时的类型，引数和参数视为有相同类型；而传址形式的引数，引数和参数在忽略限定符以及数组大小后若具有相等的类型 (参见 §A.7.11)，它们的类型则是相同的，并且遵照下面的规则：

- 有限定符 `const` 声明的对象或对象数组，只能用在函数参数是有 `const` 限定的引数。
- 有限定符 `in` 声明的对象，只可以用在函数参数是有 `in` 限定或是有 `void` 说明的引数。
- 有限定符 `out` 声明的对象，只可以用在函数参数是有 `out` 限定或是有 `void` 说明的引数。
- 有说明符 `void` 声明的对象，只可以用在函数参数具有 `void` 说明符的函数引数；没有限定符 `in`、`out` 或是 `void` 的对象，可以用在函数参数是 `in`、`out` 或是 `void` 的引数。
- 以限定符 `buffered` 声明的对象，只可以用在函数参数是 `buffered` 的函数引数；没有限定符 `buffered` 的对象，可以用在函数参数没有是 `buffered` 限定的引数。
- 以限定符 `streaming` 声明的对象，只可以用在函数参数是 `streaming` 的函数引数；没有限定符 `streaming` 的对象，可以用在函数参数中没有 `streaming` 限定的引数。
- 声明为数组大小为 *n* 的对象，只可以用在函数参数是未知大小的数组或是大小为 *m* 的数组引数，其中 *m* 小于等于 *n*。
- 被传至没有 `const` 限定的参数对象必是左值。

在引数中改变的变量，不能被其他的引数使用，这个规则递归地用于被引数调用的函数中出现的所有变量。

被传值带入的引数会如同指定一样的被转型，转成函数声明 (或是函数原型) 中相对应的参数类型；引数的数量必须是和参数的数量相同，除非声明的参数列是以省略的记号 (, ...) 结束，在

这个情况下，引数的数量必须等于或是超过参数的数量，有明确类型的参数之后的这些跟随的整数引数，会进行整数的提升 (§A.5.1)。

引数求值的顺序是没有给定的，然而，在进入函数之前，引数和其相关的表达式，会先被计算，对任何函数而言，都可以递归地调用它。

建立超过一个参考指到相同基本类型的对象、联合或是数组是无效的；建立结构、联合或是数组的参考，以及递归地包含在其中的成员或是元素的参考是无效的；建立超过一个以上在联合中不同成员的对象参考是无效的。

A.6.3.3 结构参考

变量参考后跟着一个点 (.)，再接着一个成员参考标识符，变量的参考必须是结构或是联合，并且该标识符必须是结构或是联合中的成员名称，其参考的值就是该结构或是联合成员的值，类型也和该成员相同。

结构和成员在 §A.7.4 当中讨论。

A.6.3.4 再解译

在左括号后跟着一个变量参考，再接着逗号和类型名称 (§A.7.9)，最后是右括号，是再解译强制转型。

变量参考必不能是资源类类型，其类型必须是完整的或是不完整的数组，该数组的第一个维度是没给定的，若有给定，则是完整的类型。变量类型名称必不能是资源类类型，因为它必须是完整类型。

如果变量参考的类型大小是未知，因为其参考的数组参数是未知的大小的关系，下面两个规则会被用上，第一，如果类型名称的大小是编译时的常量 T，在运行时，如果变量参考的大小是小于 T，则再解译的操作是无效的；第二，如果类型名称的大小在编译时不是已知的，因为它是数组并且最大维度是没有给定的，在运行时，再次解译操作提供这个维度一个值 d，使得结果类型的大小不会大过变量参考的类型大小，但是 d+1 则会。

如果变量参考的类型大小是编译时的常量 V，则下列的两个规则会用上，第一，如果类型名称的大小是编译时的常量 T，则 T 必须不能大于 V；第二，如果类型名称大小是未知的，因为其参考一个数组而该数组的最大维度是没有给定的，则在编译时这个维度是完整的值 d，使得结果类型的大小不会大于 V，但是 d+1 则会。

不会进行任何算术转型：再解译的效应是将变量当成如其所给定的类型，一个大小为零的数组是有效的再解译，任何企图对数组的索引都是无效的。

使用再次解译对象可能是无效的如果它在储存空间中不是适当对齐的，只有在对象可以再解译成一个类型用到较少或是刚好的对齐空间的对象，才是被保证有效的。“对齐”是实作定义，但是类型为 `char` 的对象有较不严格的对齐要求。

A.6.4 一元运算符

表达式中的一元运算符具有由右向左的顺序关联性。

```
unary-expression  ::=  postfix-expression
                    |  ++ variable-reference
                    |  -- variable-reference
                    |  unary-operator cast-expression
                    |  sizeof unary-expression
                    |  sizeof ( type-name )
                    |  isnull ( unary-expression )

unary-operator  ::=  以下其中之一
                       +  -  ~  !
```

A.6.4.1 前置式递增运算符

以 ++ 或是 -- 运算符置放在前的一元运算符是一元表达式，运算元会增加 (++) 1 或是减少 (--) 1，表达式的值是增加 (减少) 后的值，运算元必须是可修改的左值，参见加法运算符的讨论 (§A.6.7) 与赋值 (§A.6.17) 的详细讨论，运算的结果不是左值。

A.6.4.2 一元加运算符

一元 + 运算符的运算元必须是算术类型，所得到的结果就是其运算元的值；整数运算元经过整数提升后，结果的类型是提升的运算元类型。

A.6.4.3 一元减运算符

一元 - 运算符必须有算术类型，运算结果是其运算元的负值，整数的运算元进行整数提升后，无号数的负值是提升后类型的最大数值减去提升的数值再加上 1；而负数零就是零，运算结果的类型就是提升的运算元类型。

A.6.4.4 1's 补数运算符

一元 ~ 运算符的运算元必须是整数类型，而运算结果是其运算元的 1's 补数；在整数提升进行后，结果是提升后的类型的最大数值减去该值，如果运算元是带号的，则结果是转换提升后的运算元至相对应的无号类型之后，再进行 ~ 运算，并且再转回带号的类型，运算结果的类型是提升运算元后的类型。

A.6.4.5 逻辑否定运算符

! 运算符的运算元必须是算术类型，并且如果其运算元的值等于零，运算结果是 1，否则就是 0，结果的类型是 int。

A.6.4.6 sizeof 运算符

sizeof 运算符会算出存放一个其对象的运算元类型所需要的位元组数，其运算元不是尚未运算求值的表达式就是由括号刮住的类型名称；当 sizeof 用在 char 类型时，结果为 1，当用在数组时，结果则是数组所占的位元组数；当用在结构或是联合，其结果就是对象所使用的位元组数，包含填充空格成数组状所使用的空间。*n* 个元素的数组大小是一个元素大小乘上 *n*。当用于参考时，结果是对象所指的物体的位元组数；运算符不能用在函数类型的运算元、资源类类型以及不完全类型，运算符也不能用于参考类型的运算元，其中参考是指到一个未知大小的数组。运算结果是实作定义的值，值是无号的整数常量，该特定的结果类型是实作定义。

A.6.4.7 isnull 运算符

isnull 运算符的运算元必须是左值，如果其运算元是 null，运算结果是 1，否则为 0；运算结果的类型是 int。

A.6.5 强制类型转换

一元表达式前用以括号刮住的类型名称会使表达式的值转成这个名称的类型。

```
cast-expression ::= unary-expression
                  | ( type-name ) cast-expression
```

这个结构称为强制转型，强制转型不能是结构、联合、数组以及资源类类型，表达式也不行，类型名称在 §A.7.9 中有描述，算术类型的转型则是在 §A.5.3；强迫转型的表达式不是左值。

A.6.6 乘法类运算符

乘法性运算符含 *、/ 和 %，具有由左向右依序计算的性质。

```
multiplicative-expression ::= cast-expression
                            | multiplicative-expression * cast-expression
                            | multiplicative-expression / cast-expression
                            | multiplicative-expression % cast-expression
```

* 和 / 的运算元必须是有算术的类型，% 则必须要有整数类型，一般的算术类型转型是对运算元进行，并有确定的结果类型。

二元运算符 * 是乘法运算。

二元运算符 / 是产生第一个运算元除以第二个运算元的商数，而 % 运算符则是产生该运算的馀数，如果第二个运算元是 0，结果是实作定义，否则，此式“(a/b)*b + a%b 等于 a”会永远成立；然而，如果两个运算元都是非负数，则馀数也是非负数并且小于除数；如果不是，只有当馀数的绝对值小于除数的绝对值才会保证成立。

A.6.7 加法类运算符

加法类运算符含 + 和 -，具有由左向右依序计算的性质。

```
additive-expression ::= multiplicative-expression
                      | additive-expression + multiplicative-expression
                      | additive-expression - multiplicative-expression
```

对这两个运算符，每个运算元都必须是算术类型，一般的算术转型是对运算元进行，并有确定的结果类型。

运算符 + 的结果是运算元的和，而运算符 - 的结果则是运算元的差。

A.6.8 位移运算符

位移运算符含 << 和 >>，具有由左向右的计算性质。对这两个运算符而言，每个运算元必须是整数，并受到整数提升的规范，结果的类型是左运算元提升后的类型，如果右边的运算元是负数或是大于等于左边表达式类型的位元数，其结果是未定义的。

```
shift-expression ::= additive-expression
                   | shift-expression << additive-expression
                   | shift-expression >> additive-expression
```

运算符 << 的运算结果是左边的运算元向左位移右边运算元所指定的位元数；运算符 >> 的运算结果是左边的运算元向右位移右边运算元所指定的位元数。

A.6.9 关系运算符

关系运算符有 < (小于)、> (大于)、<= (小于或等于) 以及 >= (大于或等于)，具有由左向右的运算性质 (但这个性质并不实用)。

```
relational-expression ::= shift-expression
                        | relational-expression < shift-expression
                        | relational-expression > shift-expression
                        | relational-expression <= shift-expression
                        | relational-expression >= shift-expression
```

对于这些运算符，每个运算元必须是算术类型，当进行一般的算术转型，结果的类型是 `int`；当给定的关系不成立时，这些运算符产生的结果为 0，否则为 1。

A.6.10 相等性运算符

```
equality-expression ::= relational-expression
                      | equality-expression == relational-expression
                      | equality-expression != relational-expression
```

相等性运算符 == (等于) 和 != (不相等) 和关系运算符相似，除了他们的优先顺序较低。

A.6.11 位元 AND 运算符

```
AND-expression ::= equality-expression
                 | AND-expression & equality-expression
```

位元 AND 运算符 & 的运算元必须是整数类型，进行一般的算术转型时，结果是运算元的每个位元进行 AND 函数运算。

A.6.12 位元互斥 OR 运算符

```
exclusive-OR-expression ::= AND-expression
                          | exclusive-OR-expression ^ AND-expression
```

位元互斥 OR 运算符 ^ 的运算元必须是整数类型，当进行一般的算术转型时，结果是每个位元进行互斥 OR 的函数运算。

A.6.13 位元内含 OR 运算符

```
inclusive-OR-expression ::= exclusive-OR-expression
                          | inclusive-OR-expression | exclusive-OR-expression
```

位元内含 OR 运算符 | 的运算元必须是整数类型，当进行一般的算术转型时，结果是每个位元进行内含 OR 的函数运算。

A.6.14 逻辑 AND 运算符

```
logical-AND-expression ::= inclusive-OR-expression
                         | logical-AND-expression && inclusive-OR-expression
```

逻辑 AND 运算符 && 是由左向右进行运算，如果两个运算元皆不等于零则回传 1，否则为 0；它保证由左向右短路的求值，也就是说，只有当左边的运算元求得值 1 的时候，右边的运算元才会被运算。运算元必须是算术类型，但可以不是相同类型，结果的类型是 `int`；被运算元修改的变量可以用在其他运算元。

A.6.15 逻辑 OR 运算符

```
logical-OR-expression ::= logical-AND-expression
                        | logical-OR-expression || logical-AND-expression
```

逻辑 OR 运算符 || 是由左向右运算，只要其运算元之一不为 0，它会回传 1，否则为 0。它保证由左向右短路的求值，也就是说，只有当左边的运算元求得值 0 的时候，右边的运算元才会被运算。运算元必须是算术类型，但可以不是相同类型，结果的类型是 int；被运算元修改的变量可以用在其他运算元。

A.6.16 条件运算符

```
conditional-expression  ::= logical-OR-expression
                          | logical-OR-expression ?
                              expression : conditional-expression
```

如果第二和第三个运算符都不是 null 类型，它们两者都必须是算术类型，或者是有等价 (参见 §A.7.11) 的类型，忽略 buffered 和 streaming 之外的所有的限定符，也略过任何数组大小，否则两者必须皆是算术类型。

第一个表达式会先进行计算，包含其所有的影响作用域，如果评值不等于 0，计算结果的值是第二个表达式的值，否则结果是第三个表达式的值。如果第二个或是第三个运算元其中之一的类型是 null，运算结果的类型是另外一个运算元的类型，否则，若第二或是第三个运算元在忽略所有的限定符以及数组大小的情况下有等价的类型，结果的类型会是相同类型并且有限定符，而限定符是以下列这些规则决定:

- 如果其中之一的运算元是有 const 限定，则运算结果也是具有限定符 const。
- 如果其中之一的运算元是有说明符 void，则运算结果也是以 void 说明。
- 如果一个运算元是以 in 限定，而其他的运算元是 out，则运算结果是具有 void 说明符；否则，如果运算元之一是以 in 或是 out 限定，则运算结果也是分别以 in 或是 out 限定；如果运算元是不同大小的数组，结果的类型是“未知大小的数组”。

如果第二和第三个运算元是算术类型，但是类型不等价，则会进行算术转型并且有明确的结果类型。

如果没有进行算术转换，并且第二和第三个运算元两者皆是 null 类型或是左值，则表达式是个左值。

A.6.17 赋值运算

赋值运算符有好几种，皆是由右向左依序计算求值。

```
assignment-expression  ::= conditional-expression
                         | variable-reference assignment-operator
                             assignment-expression

assignment-operator    ::= 以下其中之一
                           = *= /= %=  += -= <<= >>=  &= ^= |=
```

所有这类的运算符都需要可以修改的左值为其左边的运算元，以 variable-reference 命名的标识符可以使用在任何赋值式的任何部分，包括递归调用的函数，只要在这些部分中标识符名称的变量没有被修改即可。赋值运算的类型是其左运算元的类型，并且值是赋值后存放在左运算元的值。

在简单的 = 赋值的赋值式中，表达式的值会取代被左值指到的对象，下列其中之一情况必须要成立：其一，两个运算元皆是算术类型，无论哪种类型，右边的运算元会被赋值式转型成左边的类型；其二，两个运算元皆是相同类型的结构或联合。

这个型式的表达式 `V` *op*`=` `E` 和 `V = V` *op* `(E)` 等价，唯一的不同是，`V` 只会运算一次。

A.6.18 逗号运算符

逗号运算符的限定形式是在 `for` 循环 (参见 §A.8.6) 中支援。

A.6.19 常量表达式

语法上而言，常量表达式是只限用一个运算符子集合的表达式。

constant-expression **::=** conditional-expression

表达式的值在下述情况下必须是常量：在标签的语句的 `case` 之后、作为数组范围、以及预处理器的表达式。

除了 `sizeof` 运算元之外，常量表达式不能含有赋值式、递增或递减运算符以及函数的调用，如果常量表达式要是整数，其运算元必须是由整数、枚举或是字符常量组成，强制转型时必须给定整数的类型。

A.7 声明

声明解译各个标识符，有保留储存空间的声明称作**定义**，声明的语法如下：

declaration **::=** on-statement$_{opt}$ actual-declaration

actual-declaration **::=** var-declaration
| fnc-declaration **;**
| trn-declaration **;**
| sel-declaration **;**

on-statement **::=** **on** variable-reference **:**

var-declaration **::=** ⟨dec-specifier⟩* init-var-declarator-list$_{opt}$ **;**

```
fnc-declaration    ::= ⟨dec-specifier⟩* fnc-declarator
                     | { dec-specifier-list } fnc-declarator

trn-declaration    ::= ⟨dec-specifier⟩* transaction fnc-declarator

sel-declaration    ::= ⟨dec-specifier⟩* select fnc-declarator

dec-specifier-list ::= ⟨dec-specifier⟩*
                     | dec-specifier-list , ⟨dec-specifier⟩*
```

以 on 为字首的变量声明，必须是声明类型为 port 或是 port:*n* 的对象；而跟在 on 之后的变量必须指到类型是 core 的对象。

on 并不会改变其后面的声明意义。

在 init-var-declarator-list 和 fnc-declarator (参见 §A.7.6) 里面的 var-declarator 包含声明的标识符。

```
dec-specifier          ::= storage-class-specifier
                         | type-specifier
                         | type-qualifier
                         | inline

init-var-declarator-list ::= init-var-declarator
                         | init-var-declarator , init-var-declarator-list

init-var-declarator    ::= var-declarator ⟨= initialiser⟩opt
```

声明符于后面的单元 (§A.7.6) 讨论，含声明的名称。声明必须要有至少一个声明符，或是其类型说明符必须声明一个结构标签或是联合标签；空的声明是不允许的。

A.7.1 存储类说明符

储存类别说明符有:

```
storage-class-specifier ::= auto
                          | register
                          | static
                          | extern
                          | typedef
                          | service
```

存储类的意义在 §A.3 单元中介绍。

auto 以及 register 说明符给予声明的对象自动存储类，并且只能用在函数内，这些声明也当成定义用，并且使储存空间得以保留。

static 说明符给被声明的对象静态的储存空间，并且可以用在内部或是外部的函数。在一个函数内，这个说明符配置储存空间，并且也当成定义使用，对于函数外部的影响效果，请参见 §A.10.2 中的说明。

用在函数内的 extern 说明符，说明这个声明的对象储存空间是在其他地方定义，其对函数外部的影响效果，可参见 §A.10.2 中的说明。

typedef 说明符并没有保留储存空间，并且由于语法上的方便性，被称为是存储类说明符，这在 §A.7.10 单元中讨论。

在声明中，存储类说明符最多只能用一个，如果没有给，会使用以下这些规则给定：

声明在函数内的对象是 auto，声明在函数之外的对象和函数是文件有效，当成是有外部连结的 static；service 说明符只被给予外部函数声明。

在函数内使用 extern 是不支援的。

A.7.2 类型说明符

类型说明符有下列：

```
type-specifier ::= void
                 | char
                 | short
                 | int
                 | long
                 | signed
                 | unsigned
                 | chan
                 | chanend
                 | port
                 | port:n
                 | timer
                 | core
                 | struct-or-union-specifier
                 | enum-specifier
                 | typedef-name
```

最多只有一个 long 或是 short 可以和 int 一起用，其意义和没有 int 时相同；最多也只有一个 signed 或是 unsigned 可以和 int、short、long 以及 char 一起用，任何一个都可以单独使用，那情况表示 int；signed 说明符对于强制 char 对象带有正负号是很实用的，这对其他整数类型虽然是允许的，但多馀。说明符 void 可以和 port 或是 port:*n* 一起用，用来声明一个空类型的埠，也可以用于 timer，表示空计时器。

否则，最多只有一个类型说明符可以用在一个声明中，如果省略，则被当成 int。

类型也可以限定，用来指定声明的对象的特别性质。

```
type-qualifier ::= const
                 | volatile
                 | in
                 | out
                 | buffered
                 | streaming
```

const 可以和任何的类型说明符一起使用，而 const 的对象可以给初始值，但是在之后就不能再给定值；没有对象可以使用限定符 volatile，编译器需要能够辨识这个限定符，并且发出适当的错误讯息。

in 和 out 可以和 port 以及 port:*n* 一起使用，但是不能和 void 一起。有限定符 in 的对象只能用于输入操作，而有限定符 out 的对象只能用于输出 (§A.8.3) 操作。buffered 可以和说明符 port 以及 port:*n* 一起用；streaming 可以和说明符 chan 以及 chanend 一起用。

自动类型的变量不能和 port、port:*n*、chanend 以及 core 一起声明；静态的变量不能和 chan 以及 chanend 一起声明。有 void 的埠不能用在输入或是输出的操作。

A.7.3 行内说明符 inline

类型中若有 inline，表示对该函数的调用越快越好，定义为**行内定义**，如果在编译单元中，函数中所有文件有效的声明，都含有行内说明符 inline 但没有 extern。有外部连结的函数行内定义必不含有在静态储存空间的可修改对象定义，也不能含有外部连结的标识符参考。

这些说明的有效延伸是实作定义。

A.7.4 结构和联合的声明

结构是由不同类型且有名称的成员组成的对象，联合是在不同时间表示不同类型成员的对象，结构和成员有相同的形式:

struct-or-union-specifier ::= struct-or-union identifier$_{opt}$ **{** ⟨member⟩$^{+}$ **}**
| struct-or-union identifier

struct-or-union ::= **struct**
| **union**

member 是结构或是联合的成员声明。

member ::= ⟨specifier-or-qualifier⟩$^{+}$ struct-var-declarator-list **;**

specifier-or-qualifier ::= type-specifier
| type-qualifier

struct-var-declarator-list ::= var-declarator
| var-declarator **,** struct-var-declarator-list

有这个型式的类型说明符如下:

struct-or-union identifier **{** ⟨member⟩$^{+}$ **}**

声明作为结构或是联合标签的标识符，和成员串列一起说明，之后使用该标签的声明指相同的类型，标签不含成员串列:

struct-or-union identifier

当有标签的说明符出现，但是没有串列时，**不完全类型**便产生；不完全类型结构或是联合对象可用在当他们的大小并不需要的场合；在之后的说明符中，如果附有该标签，并且有声明串列时，就变成完全类型，但是在花括号 } 结束前，仍然是不完全类型。

结构或是成员不能含有不完全类型或是数据类型的成员，除非该结构含有类型是埠或是计时器的成员；如果结构声明中含有一个这些类型的成员，则这个结构的变量只可以声明为外部声明(参见 §A.9)。

一个有串列的结构或是联合的说明符，但是没有标签，则产生一个独立的类型，它只可以从被声明处引用。

成员以及标签的名称互相不溷淆，也不会与一般的变量溷淆。成员名称在相同的结构或是联合不能出现两次，但是相同的成员名称可以用在不同的结构或是成员当中。

结构成员的位址是跟着其声明的顺序递增的，结构中的每个成员都是对齐在由其类型而定的位址边界。

联合可以看成是所有成员的位移量都是从零开始，并且其容量大足够大到可以放下任何成员，任何时间点，最多只有一个成员可以储存在联合中。

如果联合包含好几个共用初始序列的结构，而且如果该联合目前包含这些结构的其中之一，则允许指到任何结构初始序列中的任一成员。

位元栏位是不支援的。

A.7.5 枚举

枚举是种独特的类型，由有名称的常量所组成的集合，称作枚举符，枚举类型的说明符和结构或是联合相似:

```
enum-specifier   ::= enum identifier_opt { enumerator-list }
                   | enum identifier_opt { enumerator-list , }
                   | enum identifier

enumerator-list  ::= enumerator
                   | enumerator-list , enumerator

enumerator       ::= identifier
                   | identifier = constant-expression
```

枚举符类型和 int 相容，能够用在任何需要这些常量的地方，如果枚举符没有 =，则相对应的常量值由 0 开始，并且随着声明由左向右增加 1，有 = 的枚举符使其表示的标识符的值就等于标出的值，接在后面的标识符的值则由该指定值继续增加。

枚举符的名称在相同的作用域内不可以与其他的枚举符以及一般变量名称重复，但其值则不受此限制。

在 enum-specifier 中的标识符，命名特定的枚举，除了不完全枚举类型不存在外，这个 enum 说明符的规则，不论是否具有标签和串列，与在结构或是联合类型说明符中类似；没有 enumerator-list 的 enum-specifier 标签必须指到内层有串列的说明符。

A.7.6 声明符

声明符语法如下:

var-declarator ::= identifier ⟨dimension-size⟩*
| & identifier
| ? indentifier ⟨dimension-size⟩*
| & ? identifier

fnc-declarator ::= identifier (parameter-type-list$_{opt}$)

dimension-size ::= [constant-expression$_{opt}$]

声明符的结构和函数以及数组表达式的结构类似，处理的次序方式也是相同的。

A.7.7 声明符的意义

一个或是多个声明符在一系列的存储类之后出现，声明符可以用 `select` 或是 `transaction` 为前缀字，不管哪种情况，只有存储类说明符是允许在声明说明符当中，并且其隐含回传类型是 `void`。每个声明符都只能声明一个唯一的标识符，存储类说明符直接用于这个标识符，但是其类型则依其型式而定。

只考虑声明说明符 (§A.7.2) 类型部分的情况下，额外的 `transaction`、`select` 和特定的声明符，声明的型式是“opt-transaction-or-select `T D`”，其中 `T` 是一种类型而 `D` 是声明符；说明标识符属性类型的不同型式声明符是使用这种记号说明。

在声明 `T D` 中，其中 `D` 纯粹是个标识符，其类型是 `T`。

埠只可以声明成外部声明 (参见 §A.9)，或是参数；通道只可以声明成局部变量，而通道端点只可以声明成参数；含有成员或是递归的成员的结构以及资源类类型的子成员，只可以声明成外部声明。

A.7.7.1 数组声明符

在无参数声明 `T D` 中，其中 `D` 的形式是:

identifier [constant-expression]

并且声明 `T` identifier 中的标识符的类型是“类型限定符 `T`”，则标识符 `D` 的类型是“*n* 个类型限定符 `T` 的数组”，其中 *n* 是常量表达式的评值结果，用来给定该数组元素的数量；这个常量表达式必须是整数类型，并且值必须大于 0。

在参数声明 `T D` 中，其中 `D` 的形式是:

identifier [constant-expression]

并且声明 `T` identifier 中的标识符的类型是“类型限定符 `T`”，则标识符 `D` 的类型是“指到 *n* 个类型限定符 `T` 的数组的参考”，其中 *n* 是常量表达式的评值结果，用来给定该数组元素的数量；这个常量表达式必须是整数类型，并且值必须大于 0。

在声明 T D 中，其中 D 的形式是:

identifier [ ]

并且声明 T identifier 中的标识符的类型是“类型限定符 T”，则标识符 D 的类型是“类型限定符 T 的不完整数组”。

数组可以从算术类型、结构、联合、埠、计时器、通道、通道端点或是另一个数组 (而产生一个多维数组) 建构，由这些类型建构而来的数组，必须是完全类型，必不能是不完整类型的数组或是结构。这表示对于多维数组，只有第一个维度可以省略，不完整类型的对象类型的完整化，是在运行时调用该程序的入口完成，完成的方式是透过另一个完整声明对该对象 (§A.9.2) 或是透过对它初始化 (§A.7.8)，或是透过函数参数，其第一维度是省略的。

如果 E1 是个数组且 E2 是整数，E1[E2] 是指 E1 的第 E2 个成员，数组是以列的顺序储存 (最后一个下标变动的最快)，因此这个声明中的第一个下标决定数组需要的储存空间，它对其馀下标计算没有关联。

A.7.7.2 参考声明符

在声明 T D 中，其中 D 的形式是:

& identifier

并且声明 T identifier 中的标识符的类型是“类型限定符 T”，则标识符 D 的类型是“类型限定符 T 的参考”。

用 & 声明的参考的类型可以是算术、结构或是联合类型，只能声明在函数参数中。

A.7.7.3 可为空声明符

在声明 T D 中，其中 D 的形式是:

? identifier

并且声明 T identifier 中的标识符的类型是“类型限定符 T”，则标识符 D 的类型是“可为空的类型限定符 T”。

以 ? 声明的可为空对象，可以是资源类或或是参考类型，并且只能声明为函数参数。

如果对象含有指到 null 的参考，对该对象的引用除以下情况之外皆是无效的：是函数中可为空的参数的引数、运算符 isnull 的运算元或是运算符 sizeof 的运算元。

A.7.7.4 函数声明符

在函数声明 T D 中，其中 D 的形式是:

D1(parameter-type-list)

并且声明 T D1 中的标识符的类型是“类型限定符 T”，则标识符 D 的类型是“回传 T 的类型限定符函数，且该函数引数是 parameter-type-list”；如果 T 的形式是 {T_1,…,T_n}，则回传值的类型是“类型为 T_1,⋯,T_n 的串列”。在函数声明 transaction void D 中，其中 D 的形式是:

D1(parameter-type-list)

并且声明 T D1 中的标识符类型是“类型限定符 T”，则标识符 D 的类型为“回传 void 的类型限

定符交易函数，且该函数引数是 parameter-type-list”。在函数声明 `select void D` 中，其中 `D` 的形式是:

```
D1(parameter-type-list)
```

并且声明 `T D1` 中的标识符类型是“类型限定符 `T`”，则标识符 `D` 的类型是“回传 `void` 的类型限定符选择函数，且该函数引数是 parameter-type-list”。

这些参数的语法是:

```
parameter-type-list      ::= parameter-list
                           | parameter-list , parameter-declaration

parameter-list           ::= parameter-declaration
                           | parameter-list , parameter-declaration

parameter-declaration    ::= ⟨dec-specifier⟩+ abstract-or-void-dec

abstract-or-void-dec     ::= var-declarator
                           | abstract-var-declarator
```

参数串列给定参数类型，一个特殊情况是没有参数的函数声明符有参数串列，该参数串列只包含一个关键字 `void`，这也可以用空的参数串列表示。如果参数串列以省略的“, ...”结尾，则函数可以接受超过明确描述的参数数量，参见 §A.6.3.2。

在参数的声明说明符当中唯一允许的存储类说明符是 `register`，除非函数声明在函数定义上方，否则该说明符会被忽略；这个说明符在函数的行为上并没有作用，使用这个说明符作会有的作用是实作定义。

相同地，如果声明符含有标识符以及函数声明符，并且没有函数定义为前缀，标识符立即超出作用域；没有提及标识符的抽象声明符则在 §A.7.9 单元中讨论。

以储存类别说明符 `service` 声明的函数只能有声明类型为 `chanend` 的变量。

旧型的 C 函数声明 (参考 K&R §A8.6.3) 是不支援的。

A.7.8 初始化

当声明对象时，其 init-var-declarator 给定声明的标识符初始值；这个初始化子是以 = 为前置，接着一个表达式或是以花括号为巢状的初始化子的串列。

```
initialiser      ::= on-statement_opt expression
                   | { initialiser-list }
                   | { initialiser-list , }

initialiser-list ::= initialiser
                   | initialiser-list
```

on 语句在 §A.8.8 单元中讨论，如果超过一个 on 语句使用相同的变量声明，则所有这些语句必须是在相同的核心上。

静态对象或是数组的初始化子的所有的表达式，必须是常量表达式，如 §A.6.19 单元中讨论的情况。而自动或是暂存类型的对象的初始化子的表达式，如果初始化子是一个花括号包住的串列，同样的也必须是常量表达式；然而，如果自动类型对象的初始化子是个单一的表达式，该表达式不须是常量表达式，但必须有适当的指定给对象的类型。

计时器、通道和核心必不能有明确初始化；没有 extern 声明的计时器在运行时被初始化，指到一个未别名的硬件计时器；没有 extern 声明的通道在运行时被初始化，指到两个未别名的硬件通道端点，这两个端点连接起来建立点对点的通讯连结；没有 extern 声明也没有明确初始化的埠，其初始值是实作定义。

不是计时器、通道或是埠的静态对象，而且该对象并没有明确初始化，则初始化成其表达式 (或是其成员) 被指定为常量 0。没有明确初始化的算术类型的自动类型对象，其初始值没有定义。

对象或是算术类型的初始化子是单一表达式，一般是在花括号中，此表达式会被指定给对象；埠的初始化子是单一的常量表达式，是在花括号中，这个表达式会被指定给对象，其解译以及有效性是实作定义。

结构的初始化子是个相同类型的表达式或是用花括号依序刮住其成员的初始化子串列，如果串列中初始化子少于结构成员，后面的成员则初始化成 0，而初始化子数量不能超过成员数量。

数组的初始化子，是以花括号刮住其成员的初始化子，如果该数组大小是未知的，则初始化子的数量决定数组的大小，而其类型变为完整类型。如果数组是固定大小，初始化子的数量不能超过数组成员的数量，如果初始化子数量较少，后面的成员则初始化为 0。

有个特殊情况是字符数组可以被字串常量初始化 (可省略花括号)，字串中连续的字符会连续地将数组中的成员初始化。如果数组大小是未知，字串中字符数量，包含结尾的空字符，决定其大小；如果其大小是固定的，字串中字符的数量不能超过数组大小，此数量不含结尾的空字符。

联合的初始化子是单一个相同类型的表达式或是花括号括住其第一个成员的初始化子。

聚集是结构或是数组，如果聚集包含成员或是聚集类型，则初始化规则会递归地使用，在初始化中的花括号是可以省略的，如下：如果聚集的成员本身是聚集的初始化子是以左括号开始，则连续的以逗号分开的初始化子的串列将子聚集的成员初始化，初始化子多过成员是错误的；然而，如果子聚集的初始化子，并非以左括号开始，则只有串列中足够的元素会被用来给子聚集的成员，任何剩下的成员则留至初始化聚集的下个成员，而子聚集为这个聚集的一部分。

A.7.9 类型名称

在一些地方 (用强制转型来做类型转换，以及再解译中函数声明符中声明参数类型)，提供数据类型的名称是需要的，这是利用**类型名称**完成的，而其语法上是那个类型的对象声明去掉对象名称。

type-name ::= ⟨specifier-or-qualifier⟩$^+$ abstract-var-declarator

abstract-var-declarator ::= ⟨dimension-size⟩*

A.7.10 Typedef

储存类别说明符是 typedef 的声明并不声明对象，相对的，他们定义命名类型的标识符 (称为自定型态名称)。

typedef-name ::= identifier

typedef 的声明和一般声明方式相同 (参见 §A.7.7)，对每个其声明符中的名称是给予类型；之后，每个这样的自定型态名称语法上是等价于这个类型的说明符的关键字，typedef 并没有产生新的类型，而是产生同义字，使类型可以用另一种方式给定，自定型态名称可以在内层重新声明，但是必须给定一个非空集合的类型说明符。

A.7.11 类型相等性

两个类型说明符串列如果包含相同集合的类型说明符，它们是相等的；包含由其说明符隐含的说明符在内 (例如，long 也代表含了 long int，register 在正式中是被省略)。结构或是联合则给定独一的类型。

如果两个类型的抽象声明符 (§A.7.9) 在消除函数参数标识符之后和在类型说明符串列的相等性相同，则这两个类型是相等的，数组的大小 (包含数组参数大小) 是重要的；对 const 且不是参考类型的参数，相比较下，其类型是去掉该限定符的类型。

A.8 语句

除非有特别说明，语句皆是依序运行的，语句的运行是为了产生其作用，并没有值的产生；语句可以区分成好几类:

```
statement          ::= simple-statement_opt ;
                     | compound-statement
                     | selection-statement
                     | iteration-statement
                     | jump-statement
                     | parallel-statement
                     | transaction-statement

simple-statement   ::= expression-statement
                     | multiple-assignment
                     | input
                     | output
```

分号本身称为是空语句，它经常使用在反覆语句中，提供一个空的主体给反覆语句。

A.8.1 运算语句

运算语句的语法是:

expression-statement **::=** expression

大多数的运算语句是赋值式或是函数的调用，运算语句不能有资源类类型，所有运算的作用都会在下一个语句运行前完成。

A.8.2 多重指定语句

多重指定语句的语法是:

multiple-assignment **::=** `{` return-list `}` arithmetic-operator function-call

return-list **::=** optional-variable
| optional-variable `,` return-list

optional-variable **::=** variable-reference
| `void`

函数的回传类型必须是“类型 $\mathtt{T}_1,\cdots,\mathtt{T}_n$ 的串列”，其中 n 和在回传串列中的可省略变量数量相同。

在回传串列中的每个变量都受到赋值式 (参见 §A.6.17) 规则规范：函数回传的第 i 个值会取代对象中第 i 个可省略变量参考指到的值；如果可省略变量参考是 `void`，回传值则会被丢弃。

在可省略变量的下标中被变更的变量，不能出现在任何可省略变量或是在函数的调用中，包含函数的引数。任何其他可省略变量或是在函数的调用中，包含该函数的引数，不能是在可省略变量的在函数调用中改变的变量，包含函数引数，不能出现在任何可省略变量中；这些规则递归地用于所有改变的变量，或出现在可省略变量所调用的函数，或调用的函数中的变量。

赋值式变更的变量不能出现在任何其他可省略变量，也不能递归地出现在其他可省略变量调用的函数中。

如果要被指定的对象是和另一个相同，则赋值式是无效的。

A.8.3 输入和输出语句

输入语句是用来接收从通道端点、埠或是计时器来的值，并且将收到的值指定给对象。

input **::=** resource time_{opt} $\text{predicate}_{\text{opt}}$ input-operator
dest $\text{input-timestamp}_{\text{opt}}$

resource **::=** variable-reference

time **::=** `@` expression

input-operator ::= `:>`
| `:> >>`

dest ::= declared-var-reference

input-timestamp ::= `@` declared-var-reference

declared-var-reference ::= ⟨declaration-specifier⟩$^{+}$ identifier$_{opt}$
| variable-reference

predicate ::= **`when`** function-call

资源必须是对通道端点、埠或是计时器命名，如果资源是对通道端点或是计时器命名，则 time 以及 input-timestamp 皆不被允许使用；如果资源是对通道端点命名，不允许有 predicate；如果资源是对埠命名，埠必须不能有 `out` 限定属性，并且结果变量必须是算术类型。

如果使用 time，则时间表达式必须是算术类型，输入则称作是有时效性的。

如果使用 input-timestamp，则 declared-var-reference 必须对算术类型的可修改左值命名，输入则称为是有时戳性的。

如果使用 predicate，命名的函数必须声明是回传 `void`，并且从它的参数串列中，必须仅有一个埠或是计时器声明，这个声明是有 `void` 限定属性，该输入被称作是**语述**输入；支援的语述是实作定义。调用函数的快速路径是：资源变量必须不能当成引数传递，它是以埠或计时器引数的型态隐含地传递。

如果有标识符被命名，declared-var-reference 必须是可变更的左值，但不能定义新类型。如果资源是对埠或是计时器命名，则左值必须不能是结构或是联合的参考，如果没有给定标识符，类型必不能是结构或是联合，但可以是 `void`；如果没有声明说明符，则变量的类型必不能是 `const`，如果是结构或联合，必不能有成员或是递归中的子成员是 `const`。如果有声明说明符，则变量参考也是声明，而说明符不能含有 `typedef` 但可以是 `const`。

在输入中变更的变量，除非有特别如下的状况，否则皆不能出现在该输入的任何其他部份：如果 declared-var-reference 是 variable-reference，只要被标识符命名的变量在这些部份都没有变更，被命名的标识符可以出现在输入的任何部分。另外，被 input-timestamp 写入的变量，不能出现在 dest，且被 dest 写入的变量，不能出现在 input-timstamp，这些规则递归地用于被输入调用的函数变更的所有变量。

在输入中的第一个变数开始一个内层作用域，该作用域在之前声明后立即开始，并且一直持续到输入的最后，如果输入出现在选择的守卫叙述中，则这个作用域持续含盖到冒号之后的最后语句。

如果资源命名的是通道端点或是计时器，或是目的标识符是省略的，则不允许使用 `:>` `>>` 运算符。

输出语句用来传送表达式的值到通道端点或是埠：

output ::= resource time$_{opt}$ output-operator
expression output-timestamp$_{opt}$

```
resource          ::= variable-reference

time              ::= @ expression

output-operator   ::= <:
                    |  <: >>

output-timestamp  ::= @ variable-reference
```

资源必须是对通道端点或是埠命名，如果资源是对通道端点命名，则 time 以及 output-timestamp 皆不允许使用；如果资源是对埠命名，埠必须不能有 `in` 限定属性，并且输出表达式必须是算术类型；否则，就输出语句必须是算术类型或必须是结构或联合。如果资源是对通道端点命名，则不允许使用 <: >> 运算符；如果有 <: >> 运算符，输出表达式必须是个可修改的左值。如果使用 time，则 output-timestamp 是不允许的；且时间表达式必须是算术类型，而输出则称作是有**时效性**的。如果使用 output-timestamp，则变量参考必须是个算术类型的可修改左值，而输出称为是有**时戳性**的。

被输出任何部分修改的变量，除非有是以下说明的情况，否则必不能出现在输出的任何部分；output-timestamp 命名的标识符，可以出现在输出的任何其他部分，只要变量名称在这些部分没更改，这些规则递归地用在被输出调用的函数中改变的所有变量。

> 输入和输出语句是新增的部份，输入/输出操作一般使用中断以及系统调用 (透过 C 库函数) 的方式进行。

A.8.3.1 通道输入和输出

在并行语句中，通道端点的输入会使处理器等待直到符合的输出完成后才接收值 (参见 §A.8.8)，如果输入变量的类型有给定但是标识符被省略，则接收到的值是被忽略的，有关通道通讯的输入，参见 §A.11。

在并行语句中，通道端点的输出会使处理器等待直到符合的输入完成后才送出值，有关通道通讯的输出，参见 §A.11。

A.8.3.2 埠的输入和输出

由埠来的输入会使该埠提供处理器一个数值，如果该埠的传输宽度是 *w* 个位元，这些 *w* 个位元会被指定到变量的最高有效位元，该变量是埠的名义传输类型 (参见 §A.3.2)，并且其馀的位元被设为零。如果有给定输入变量的类型但标识符是省略的或是 `void`，这个输入变量会被忽略。如果输入和运算符 :> >> 一起使用，目的变量则向右平移 *w* 个位元，并且于该值上进行位元内含 OR，然后将输入变量指定给目的变量，否则输入变量会被指定给目的变量。

如果满足一个 `when` 的条件，则函数和其引数会在进行输入之前提供给埠。

输出到埠会使输出表达式首先转型成埠的名义传输类型，接着提供给埠，如果输出和运算符 <: >> 一起使用，这个输出变量会向右平移 *w* 个位元。

如果输入或是输出是有时效性，在进行输入或是输出之前，被 time 指定的值被转型为埠的名义计数器类型後再提供给埠。

如果输入或是输出有时戳性，指定 *t* 个位元给变量的最低有效位元，此变量是埠的名义计数器类型 (参见 §A.3.2)，并且剩馀的位元被设成零，然后将这个变量指定给时戳变量。

关于埠的输入和输出中及埠和处理器之间的通讯相关的意义，以及埠在其脚位上的相对应操作，参阅附录 B 单元中的说明。

A.8.3.3 计时器输入

从计时器来的输入使计时器提供其计数器目前的值，这个值会指定给变量的最低有效未元，且该变量类型是计时器的名义计数器 (参阅 §A.3.2)，并且剩馀的位元被设为零。如果有给定输入变量的类型，但是标识符是省略的，或者类型是 `void`，则这个输入变量则被忽视；否则，输入变量会指定给目的变量。

A.8.4 复合语句

复合语句 (“区段”) 是为了使好几个语句可以当成一个语句使用，函数定义的主体便是个语句:

compound-statement **::= {** ⟨var-declaration⟩* ⟨statement⟩* **}**

如果在 var-declaration-list 中的标识符原本是在区段外面的作用域，则输出的声明在这个区段 (参阅 §A.10.1) 内是暂时没有作用的。标识符在同样的区段中，只可以被声明一次，这些规则用于在相同命名空间 (§A.10) 中的所有标识符，在不同命名空间的标识符是被视为不同的。

自动类型对象的初始化是在每次进入该区段的最上方时进行，并且依照声明的顺序进行；`static` 对象的初始化则是在程序开始运行时进行，并且只进行一次。

A.8.5 选择语句

选择语句会在好几个控制流程中挑选一个运行。

selection-statement **::= if (** expression **)** statement
| **if (** expression **)** statement **else** statement
| **switch (** expression **) {** ⟨labelled-statement⟩+ **}**
| **select {** ⟨guarded-statement⟩+ **}**

labelled-statement **::= case** constant-expression **:** ⟨statement⟩*
| **default :** ⟨statement⟩*

```
guarded-statement ::= case replicator_opt enable-exp_opt input : ⟨statement⟩*
                    | case replicator_opt enable-exp_opt function-call :
                          ⟨statement⟩*
                    | case replicator_opt enable-exp_opt slave-statement :
                          ⟨statement⟩*
                    | default : ⟨statement⟩*
                    | case function-call ;

replicator        ::= ( int variable = expression ; expression ; experssion )

enable-exp        ::= expression =>
```

在 `if` 语句的两种形式中，必须是算术类型的表达式会被评值，包含所有影响的运算，并且若结果不等于 0，则会运行第一个子表达式；在第二个型式中，如果表达式是 0，则第二个子语句会被运行；`else` 的歧异性处理，是透过连接 `else` 和在相同区段中同一层的最后一个没有 `else` 的 `if`。

`switch` 语句使控制会根据表达式的值传给好几个条件语句的其中一个，该表达式必须是整数类型。控制的表达式会进行整数提升 (§A.5.1)，提升后，在同一个 `switch` 当中，不能含有两个相同常量值的条件，最多也只能有一个 `default` 标签在一个 `switch` 中。

当 `switch` 语句被运行的时，其表达式会被运算求值，包括其所有的作用，并且与条件常量做比较；如果这些条件常量之一和此数值相等，流程控制将送给语句符合的 `case` 标签语句，如果没有条件常量符合表达式值，并且有 `default` 标签，则控制会传给这个 `default` 标签语句，如果没有符合的常量也没有 `default` 标签，则 `switch` 中的子语句皆不会被运行。

`select` 语句使控制被传送到好几个守卫语句中之一，一个守卫语句可以是由可省略的重复器和输入之后的一个可省略的表达式所组成，或者是由一个从交易语句组成，或是由在冒号与一个或是零个语句之后的函数调用组成。

在重复器中，第三个表达式必须是将此重复器声明的变量加上或是减去一个常量表达式，重复器是对多个情况的快速算法，并且其意义如同程序在 `for` 循环中展开，另外，如果初始化子是常量表达式并且第二个表达式是用来比较重复器声明的变量和常量表达式的关系表达式，这个重复器声明的变量会被当成是在重复器主体的常量表达式，这个变量不能在这个重复器之外的地方修改。

如果在冒号之前的语句是调用交易函数 (§A.9.1.1)，则这会被当成对从交易语句的快速算法，该从交易语句会运行这个调用。运行的表达式必须是算术类型，并且必不能修改区域变量、静态变量或是参考参数；表达式调用的任何函数必须不能递归地改变静态变量、参考参数或是进行输入或输出的操作。用于有效表达式的这些修改规则也同样地应在选择函数调用的引数，这些规则也用于输入语句中的冒号之前，除非是 (依定义) 被修改的输入左值。在被选择到之前，使埠产生可观察到行为的输入守卫是无效的；在一个选择叙述中，最多只能有一个 `default` 标签。

守卫语句也可以由选择函数 (参见 §A.9.1.2) 的调用组成，在接着分号；这些规则用在有效的表达式上，也用在对选择函数调用的引数上；每个冒号之前被命名，也对选择函数是引数的埠、计时器以及通道端点，必须是不同的。

当 `select` 语句被运行时，不含有效表达式的每个守卫则被启用；对含有有效表达式的每个

守卫，这个表达式会被运算求值，且若结果和 0 不相等，则该条件变被运行；调用选择函数的行为就如同选择函数中的情况是内含在 select 当中的情况相同。

在有效运行的序列之后，如果没有条件是有效的，则若有 default，它会被运行，若没有 default，select 当中的子语句皆不会被运行，并且 select 不会完成运行 (产生死锁)；否则，select 会等待直到有效情况的条件中的一个输入或是交易是准备好的状态并且运行相对的区段代码，如果这些输入或是交易中，有超过一个是准备好的状态，则会决定被运行的部份具不确定性；在操作输入或是交易之后，冒号之后选到的语句会被运行。

在每个 select 条件语句中冒号之后的语句，必须以 break 或是 return 结束，如此，控制流程不会由一个 case 语句跑到下一个去。

A.8.6 反覆语句

反覆语句形成循环。

```
iteration-statement ::= while ( expression ) statement
                    |   do statement while ( expression ) ;
                    |   for ( for-init_opt ; expression_opt ; simple-list_opt )
                            statement

for-init            ::= var-declaration
                    |   simple-list

simple-list         ::= simple-statement
                    |   simple-statement , simple-list
```

在 while 和 do 语句中，只要表达式的值不等于零，会重复运行子语句，而表达式必须是算术类型；在 while 当中，条件的检测包含其影响的部份，都会在运行语句之前进行，而 do 则是在每次的反覆之后进行条件检测。

for 语句可以声明变量，且该变量的作用域于声明之后立即开始，一直持续到语句结束；变量如果有初始化子，只会评值一次，或者如果有简单语句的串列，语句会被运行一次，中间的表达式必须是算术类型，它是在每次的反覆之前进行评值，当结果为 0 时，for 则结束，在分号之后的可省略的简单表达式串列是在每次反覆之后被评值；这三个部份的任何一个都可以省略，表达式的省略表示等同于检测一个非零的常量。

A.8.7 跳转语句

跳转语句无条件地将程序的控制转到其它地方。

```
jump-statement ::= continue ;
               |   break ;
               |   return expression_opt ;
               |   return { expression-list } ;
```

continue 语句只可以出现在反覆语句中，不能出现在并行语句、主语句以及从语句，除非该语句被包含在反覆语句中，它会使控制传给最小包含这样语句的循环继续。

break 语句只可以出现在反覆语句、switch 语句或是 select 语句，不能出现在并行语句、主语句以及从语句，除非该语句有反覆、switch 或是 select 语句将其包住，它会结束最小包含这样语句的运行，在结束的语句后，控制传给该语句。

return 语句是函数用来回到其调用者，return 语句不能出现在并行语句、主语句以及从语句，当 return 后面接表达式，该值会被传给该函数的调用者，该表达式会被转换成该函数回传的类型，如同被指定一般。

当 return 之后跟着一个在括号中的表达式串列，该串列的值会被传给该函数的调用者，要回传 n 个表达式的串列，函数的回传类型必须是“$T_1,\cdots,T_n$ 串列”，而所有的表达式 $(i = 1..n)$，第 i 个表达式类型会被转型成其出现的函数的第 i 个回传类型。

函数流程结束就相当于回传，而该回传没有表达式，无论哪个情况，回传值是没有定义的。

goto 语句不支援。

A.8.8 共时语句

并行语句是使好几个语句共时地运行。

parallel-statement ::= **par** replicator$_{opt}$ **{** ⟨thread⟩* **}**

replicator ::= **(** **int** variable **=** expression **;** expression **;** experssion **)**

thread ::= on-statement$_{opt}$ statement

on-statement ::= **on** variable-reference **:**

重复器在 §A.8.5单元中有讨论，另外，初始化子必须给定一个常量表达式，而第二个表达式必须是关系表达式，用来比较重复器所声明的变量和一个常量表达式，这个关系运算符不能是相等或不相等运算，且条件必须是所声明变量的一些值可以满足的 (例如，x > MAX_INT 是不允许的)。

on 语句只允许两种情况使用，其一是用于并行语句，且该并行语句是在函数中唯一的语句，其二是函数复合语句的两个语句之一，第二个是回传语句，回传一个和 0 相比的常量表达式。

on 并没有改变其所在的语句的行为。

数值可以使用通道通讯 (§A.3.2) 的输入和输出语句 (§A.8.3) 在共时语句之间传递。

用在并行语句的变量和通道是有使用上的规则，这些规则是预防它们可能以潜在危险的方式使用于语句之间并且不小心被共用，规则说明如下。

在 par 中的其中一个语句中，被赋值式或是输入改变的变量，不能出现在 par 的任何其他语句，这个规则递归地用于所有在 par 的一个语句中，被赋值式或是输入改变的变量 (也就是说，变量可以出现在 par 的任何数量的语句中，前提是只要这个变量在这些语句中并没有被指定值或是输入)。

一个通道在 par 中不能用于超过两个语句中，通道端点、埠以及计时器在 par 中则是不能用于超过一个语句中。

如果一个语句包含好几个子语句，如复合语句 (§A.8.4)，由于这个规则的目的，所有的子语句则被视为如单一个语句般。

A.8.9 交易语句

交易语句是为了使在一个通道上的数个通讯可以逻辑性地被组一起。

transaction-statement ::= slave-statement
| master-statement

slave-statement ::= **slave** statement

master-statement ::= **master** statement

在 master 或是 slave 内的所有输入和输出逻辑上是同一个交易的部份，哪些底层通讯协定为了交易通讯而被最佳化则是实作定义。

语句必须只能指一个通道端点，也就是**交易子**，如果标明交易子的变数参考包含数组下标，则下标必须是常量表达式；交易子不能是串流通道。

在交易语句中：任何除了交易子之外的通道端点上的输入和输出是禁止的，使用交易子之外的通道端点当成函数的引数是禁止的，使用交易子为不是交易函数的引数是禁止的，使用巢状的交易语句是禁止的，声明通道 (于该语句中或是递归地语句中，以及任何在这个交易中的任何函数) 是禁止的。

A.9 外部声明

给 XC 编译器的输入单位称为编译单元，它包含一系列的外部声明，外部声明不是声明就是函数定义。

translation-unit ::= ⟨external-declaration⟩$^+$

external-declaration ::= declaration
| function-definition

外部声明的作用域会持续到在它们被声明的编译单元结束。

A.9.1 函数定义

函数定义的形式如下：

function-definition ::= fnc-declaration compound-statement
| trn-declaration compound-statement
| sel-declaration { ⟨guarded-statement⟩$^+$ }

在声明说明符中，唯一允许的储存类别说明符是 extern、static 以及 inline，参见 §A.10.2 中的作用说明；在函数定义中，不允许省略运算符“, ...”。

函数可以回传算术类型、结构、联合或是 void，但不能是资源类类型、函数或是数组，或者，可以回传以算术类型、结构或是联合任意组合成的串列，函数不能回传含有资源类成员的结构或是递归的子成员结构。

除非参数是由独一的 void 组成，意指函数不带任何参数，参数串列中的每个声明符必须包含标识符，参数就在组成函数主体的复合语句之后被当成声明，因此相同的标识符不能在这里再次声明 (虽然可以在内层区段再次声明)，在调用这个函数的期间，引数在需要的情况下会被转型并且指定给参数，参见 §A.6.3.2 单元中说明。

A.9.1.1 交易函数

函数声明中有关键字 transaction 限定为交易函数 (参见 §A.7.7.4)，该函数主体是由语句的串列组成，且依定义为交易语句 (参见 §A.8.9)，在该函数的参数串列中，必须严格地声明一个通道端点，依定义为交易子。

A.9.1.2 选择函数

函数声明中若有关键字 select 限定为选择函数 (参见 §A.7.7.4)，该函数的主体是由守卫语句的串列组成，且依定义为选择语句 (参见 §A.8.5)，选择函数中的守卫条件不能含有重复器以及交易子。

A.9.2 外部声明

外部声明给定对象、函数以及其他标识符特性，“外部”这个术语是指它们的位置在函数外面，并和关键字 extern 无直接关联，外部告的对象的储存类别可以没有给定，或者可以给定为 extern 或是 static 。

在相同编译单元中的相同的标识符，如果它们的类型以及连结是相同的，且该标识符最多只有一个定义，则它们可以有好几个外部声明。

对对象或是函数的两个声明，在 §A.7.11 单元中讨论的规则中，类型被当成是相同看待，而且如果声明不同的情况，一个类型是不完整的结构或是联合，而另一个是有相同标签的相对完整类型时，则类型则是相同的；而一个类型是不完整数组 (§A.7.7.1)，且另一个是完整的数组类型，则类型被视为相同。

如果函数或是对象的第一个外部声明含有说明符 static，则标识符具有**文件有效** (**内部连结**) 的性质，否则具有**程序有效** (**外部连结**) 的性质；连结在 §A.10.2 单元中讨论。

如果对象有初始化子，则对象的外部声明是定义；外部对象的声明，如果没有初始化子也没有说明符 extern，则是**暂时性定义**。如果对象的定义出现在编译单元中，任何暂时性的定义被当成是多馀的声明，如果对象没有定义出现在编译单元，所有其暂时性定义变成单一的定义，且初始化子为 0。

每个对象必须有刚好一个定义，对一些内部连结的对象，规则是分别地用在每个编译单元；而对于外部连结的对象，则是用在整个程序。

A.10 作用域和连结

有两种作用域要考虑，第一是标识符的**词法作用域**，也就是在程序中，可以使用该识别字特性的区域；第二是对有外部连结的对象及函数的作用域，它决定在不同编译单元中之标识符的连接。

A.10.1 词法作用域

标识符会在好几种名称区域中，不同区域中的识别字不会互相干扰，相同的识别字甚至在同一作用域内，都可用作不同的目的，只要在使用时是在不同的名称区域。这些分类成：对象和函数、结构和联合的标签、结构或是联合每个的成员。

对象或是函数标识符在外部声明中的词法作用域，是从其本身的声明符结束开始，并且持续到它出现的编译单元结束为止；函数定义的参数的作用域从定义该函数的区块开始，持续于整个函数；函数声明中参数的作用域在声明符最后的地方结束；在程序区块前头声明的标识符的作用域，开始于其声明符后，并且只在该区块有效；结构或是联合的作用域是从其类型说明符出现开始，到该编译单元 (声明在外部层次) 结束或是到该区块 (声明在函数中) 结束为止。

在程序区块中，包含构成函数的区块在内，如果标识符是清楚地定义的在开始的地方，区块外任何和此标识符同名的任何声明皆被暂停使用，直到这个区块结束。

A.10.2 连接

在一个编译单元内，所有对内部连结用的同一对象或是函数标识符的声明都是指相同的物体，而且这个对象或函数对该编译单元是唯一的。所有对外部连结的同一对象或是函数标识符的声明皆是指相同的物体，这个对象或是函数是整个程序共有的。

如果使用说明符 `static` 在标识符的第一个外部声明，则该标识符是内部连结用，否则就是外部连结。

行内 (§A.7.3) 定义并不提供函数外部定义，也不禁止外部定义；行内定义提供一种对外部定义的选择，可以用它在相同的编译单元实作任何对函数的调用，然而，没有说明对函数的调用是否使用行内定义或是外部定义。

A.11 通道通讯

通道通讯发生在同一个通道上：

- 输出是和输入并行运行，或是以下其他情况。
- 主交易和从交易是并行运行。

输出和从交易并行运行是无效的，主交易和输入并行运行是无效的。

在交易外，输出 - 输入通讯中，输出的位元组数和输入的位元组数不相等是无效的；在交易内，如果所有的通讯个别皆是有效的，则交易也是有效的；另外，通讯发生当输出的位元组数和输入的位元组数是不相等的情况下，则不管交易是否有效，通讯的值是实作定义。

在交易内的无效的通讯，必不能使交易变成无效，除非从交易语句超出作用域。

输出 - 输入通讯的意义是其间输出表达式的类型 e 和输入变量的类型 v 相同，也就是和赋值式 $v = e$ 相同的意思；如果类型不同并且通讯并非无效，其意义则是 $v = (e, 类型(v))$ (参见 §A.6.3.4)。

A.12 无效操作

语法上是合法的操作，但由于某种缘故或是在某些情况下语义上是无效的，可能是下列三种情况之一：

- 出现编译错误。
- 是实作定义的行为，例如：处理器可能发出严重问题警讯，而捕捉问题信号处理机制可能结束程序运行。
- 导致无定义的行为。

如果在某个时间点 t，程序保证运行事件的一些序列，使其在将来的某个时间 $t+n$ 是无效的，则在 t 和 $t+n$ 之间的任何时间，是允许其无效性，这样允许以实作改善代码效率，例如：将安全性检查重新安排在循环外面。

A.13 预处理器

除了以下的这些例外，预处理器的规格和 C99[7, §6.10] 中的定义相同：

- 宏 `__XC__` 定义为 1。
- 宏 `__STDC__`、`__STDC_HOSTED__` 和 `__STD_VERSION__` 并没有定义。

A.14 文法

以下是在这个附录中的文法总整理，文法中有未定义的终端符号：integer-constant、character-constant、identifier、string 和 enumeration-constant，打字机字体 `typewriter` 的字或是符号皆是终端符号。

translation-unit ::= ⟨external-declaration⟩$^+$

external-declaration ::= declaration
| function-definition

function-definition ::= fnc-declaration compound-statement
| trn-declaration compound-statement
| sel-declaration { ⟨guarded-statement⟩$^+$ }

declaration ::= on-statement$_{opt}$ actual-declaration

actual-declaration ::= var-declaration
| fnc-declaration ;
| trn-declaration ;
| sel-declaration ;

var-declaration ::= ⟨dec-specifier⟩* init-var-declarator-list$_{opt}$;

fnc-declaration ::= ⟨dec-specifier⟩* fnc-declarator
| { dec-specifier-list } fnc-declarator

trn-declaration ::= ⟨dec-specifier⟩* `transaction` fnc-declarator

sel-declaration ::= ⟨dec-specifier⟩* `select` fnc-declarator

dec-specifier-list ::= ⟨dec-specifier⟩*
| dec-specifier-list , ⟨dec-specifier⟩*

dec-specifier ::= storage-class-specifier
| type-specifier
| type-qualifier
| `inline`

storage-class-specifier ::= `auto`
| `register`
| `static`
| `extern`
| `typedef`
| `service`

type-specifier ::= `void`
| `char`
| `short`
| `int`
| `long`
| `signed`
| `unsigned`
| `chan`
| `chanend`
| `port`
| `port:`*n*
| `timer`
| `core`
| struct-or-union-specifier
| enum-specifier
| typedef-name

type-qualifier ::= `const`
| `volatile`
| `in`
| `out`
| `buffered`
| `streaming`

struct-or-union-specifier ::= struct-or-union identifier$_{opt}$ `{` ⟨member⟩$^{+}$ `}`
| struct-or-union identifier

struct-or-union ::= `struct`
| `union`

init-var-declarator-list ::= init-var-declarator
| init-var-declarator-list `,` init-var-declarator

init-var-declarator ::= var-declarator ⟨`=` initialiser⟩$_{opt}$

member ::= ⟨specifier-or-qualifier⟩$^{+}$ struct-var-declarator-list `;`

specifier-or-qualifier ::= type-specifier
| type-qualifier

struct-var-declarator-list ::= struct-var-declarator
| struct-var-declarator-list `,` struct-var-declarator

struct-var-declarator ::= var-declarator
| var-declarator$_{opt}$ `:` constant-expression

enum-specifier ::= `enum` identifier$_{opt}$ `{` enumerator-list `}`
| `enum` identifier$_{opt}$ `{` enumerator-list `,` `}`
| `enum` identifier

enumerator-list ::= enumerator
| enumerator-list `,` enumerator

enumerator ::= identifier
| identifier `=` constant-expression

var-declarator ::= identifier ⟨dimension-size⟩*
| `&` identifier
| `?` indentifier ⟨dimension-size⟩*
| `&` `?` identifier

fnc-declarator ::= identifier `(` parameter-type-list$_{opt}$ `)`

dimension-size ::= `[` constant-expression$_{opt}$ `]`

parameter-type-list ::= parameter-list
| parameter-list `,` parameter-declaration

parameter-list ::= parameter-declaration
| parameter-list `,` parameter-declaration

parameter-declaration ::= ⟨dec-specifier⟩$^{+}$ abstract-or-void-dec

abstract-or-void-dec ::= var-declarator
| abstract-var-declarator

initialiser ::= on-statement$_{opt}$ expression
| `{` initialiser-list `}`
| `{` initialiser-list `,` `}`

initialiser-list ::= initialiser
| initialiser-list `,` initialiser

type-name ::= ⟨specifier-or-qualifier⟩$^{+}$ abstract-var-declarator

abstract-var-declarator ::= ⟨dimension-size⟩*

typedef-name ::= identifier

statement ::= simple-statement$_{opt}$ `;`
| compound-statement
| selection-statement
| iteration-statement
| jump-statement
| parallel-statement
| transaction-statement

simple-statement ::= expression-statement
| multiple-assignment
| input
| output

compound-statement ::= `{` ⟨var-declaration⟩* ⟨statement⟩* `}`

selection-statement ::= `if` `(` expression `)` statement
| `if` `(` expression `)` statement `else` statement
| `switch` `(` expression `)` `{` ⟨labelled-statement⟩$^{+}$ `}`
| `select` `{` ⟨guarded-statement⟩$^{+}$ `}`

labelled-statement ::= `case` constant-expression `:` ⟨statement⟩*
| `default` `:` ⟨statement⟩*

guarded-statement ::= `case` replicator$_{opt}$ enable-exp$_{opt}$ input `:` ⟨statement⟩*
| `case` replicator$_{opt}$ enable-exp$_{opt}$ function-call `:` ⟨statement⟩*
| `case` replicator$_{opt}$ enable-exp$_{opt}$ slave-statement `:` ⟨statement⟩*
| `default` `:` ⟨statement⟩*
| `case` function-call `;`

replicator ::= `(` `int` variable `=` expression `;` expression `;` experssion `)`

enable-exp ::= expression `=>`

iteration-statement ::= `while` `(` expression `)` statement
| `do` statement `while` `(` expression `)` `;`
| `for` `(` for-init$_{opt}$ `;` expression$_{opt}$ `;` simple-list$_{opt}$ `)` statement

jump-statement ::= `continue` `;`
| `break` `;`
| `return` expression$_{opt}$ `;`
| `return` `{` expression-list `}` `;`

parallel-statement ::= `par` replicator$_{opt}$ `{` ⟨thread⟩* `}`

thread ::= on-statement$_{opt}$ statement

on-statement ::= `on` variable-reference `:`

transaction-statement ::= slave-statement
| master-statement

slave-statement ::= `slave` statement

master-statement ::= `master` statement

for-init ::= var-declaration
| simple-list

simple-list ::= simple-statement
| simple-list `,` simple-statement

expression-statement ::= expression

```
expression              ::= assignment-expression

assignment-expression   ::= conditional-expression
                        |   variable-reference assignment-operator
                                assignment-expression

assignment-operator     ::= 以下其中之一
                              = *= /= %=  += -= <<= >>=  &= ^= |=

conditional-expression  ::= logical-OR-expression
                        |   logical-OR-expression ?
                              expression : conditional-expression

constant-expression     ::= conditional-expression

logical-OR-expression   ::= logical-AND-expression
                        |   logical-OR-expression || logical-AND-expression

logical-AND-expression  ::= inclusive-OR-expression
                        |   logical-AND-expression && inclusive-OR-expression

inclusive-OR-expression ::= exclusive-OR-expression
                        |   inclusive-OR-expression | exclusive-OR-expression

exclusive-OR-expression ::= AND-expression
                        |   exclusive-OR-expression ^ AND-expression

AND-expression          ::= equality-expression
                        |   AND-expression & equality-expression

equality-expression     ::= relational-expression
                        |   equality-expression == relational-expression
                        |   equality-expression != relational-expression

relational-expression   ::= shift-expression
                        |   relational-expression < shift-expression
                        |   relational-expression > shift-expression
                        |   relational-expression <= shift-expression
                        |   relational-expression >= shift-expression

shift-expression        ::= additive-expression
                        |   shift-expression << additive-expression
                        |   shift-expression >> additive-expression
```

```
additive-expression ::= multiplicative-expression
                      | additive-expression + multiplicative-expression
                      | additive-expression - multiplicative-expression

multiplicative-expression ::= cast-expression
                            | multiplicative-expression * cast-expression
                            | multiplicative-expression / cast-expression
                            | multiplicative-expression % cast-expression

cast-expression ::= unary-expression
                  | ( type-name ) cast-expression

unary-expression ::= postfix-expression
                   | ++ variable-reference
                   | -- variable-reference
                   | unary-operator cast-expression
                   | sizeof unary-expression
                   | sizeof ( type-name )
                   | isnull ( unary-expression )

unary-operator ::= 以下其中之一
                     +  -  ~  !

postfix-expression ::= primary-expression
                     | variable-reference ++
                     | variable-reference --

primary-expression ::= variable-reference
                     | function-call
                     | constant
                     | string
                     | ( expression )

multiple-assignment ::= { return-list } assignment-operator function-call

return-list ::= optional-variable
              | return-list , optional-variable

optional-variable ::= variable-reference
                    | void

input ::= resource time_opt predicate_opt input-operator
          dest timestamp_opt

resource ::= variable-reference
```

time ::= `@` expression

input-operator ::= `:>`
| `:> >>`

dest ::= declared-var-reference

input-timestamp ::= `@` declared-var-reference

output-timestamp ::= `@` variable-reference

declared-var-reference ::= ⟨declaration-specifier⟩$^{+}$ identifier$_{opt}$
| variable-reference

predicate ::= `when` function-call

output ::= resource time$_{opt}$ output-operator
expression timestamp$_{opt}$

output-operator ::= `<:`
| `<: >>`

function-call ::= identifier `(` expression-list$_{opt}$ `)`

variable-reference ::= identifier
| variable-reference `[` expression `]`
| variable-reference `.` identifier
| `(` variable-reference `,` type-name `)`

expression-list ::= expression
| expression-list `,` expression

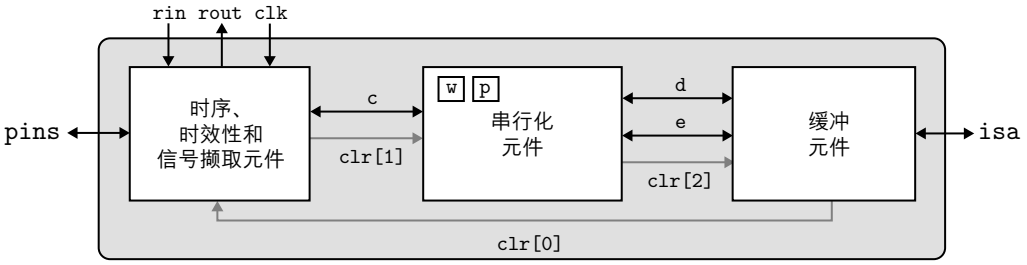
constant ::= integer-constant
| character-constant
| enumeration-constant
| `null`

附录 B

XC 输入/输出规格

附录中的这份规格说明埠上的输入/输出操作的功能性行为。

由于数据是在处理器和脚位之间传送，因此，为了说明语义，可以将埠定义成一组并时线程，对数据进行一些函数操作；XMOS 埠的构成模型如下图所示:



逻辑上而言，埠包含延伸“原始埠”的三个部分，原始埠具有时序性、时效性、信号撷取的、串行化的以及缓冲数据的功能，下面的 XC 程序为这些元件以及他们的连接性定义一个模型:

```
port pins, clk, rin, rout;
int w, p;
int direction;

void main(void) {
  chan c, d, e, isa, clr[3];
  par {
    clkTimeStrobe(pins, rin, rout, clk, c, clr[0], clr[1]);
    serialiser(w, p, c, d, e, clr[1], clr[2]);
    buffer(n, direction, d, e, isa, !isnull(rout), clr[2], clr[0]);
    processor(isa);
} }
```

透过这样的方式定义埠的功能性，可以把埠的编程看成在并时系统中和其他元件进行接口相似。

B.1 有时序的输入/输出之功能模式

对于没有使用准备输入和准备输出信号，并且有时序而没有缓冲区的埠，其输入和输出行为定义如下:

- 输出边缘发生在埠时序的下一个下降边缘。
- 输出操作在下一个输出边缘驱动数据，处理器会阻塞到下个上升边缘。
- 输入边缘发生在下一个埠时序的上升边缘。
- 输入操作使埠在下一个输入边缘取样数据，处理器会阻塞直到这个时间点。

下面的函数实作这些语义，它在和时序以及数据信号接口的两个原始埠上操作，时序被塑造成如同脚位，当切换时，如被 XCore 的时序区块切换，表示一个时序边缘 (注意这个函数并没有实作任何时效性的操作或是信号撷取，且它是直接和处理器接口)。

```
void clkTimeStrobe(port pins, port clk, chanend isa) {
  pwidth_t data   = 0;
  int      clkVal = 0;
  int      state  = QUIET;

  while (1) {
    select {
      case clk when pinsneq(clkVal) :> clkVal :
        if (!clkVal && state == OUTPUT) {
          pins <: data;
          state = PENDQUIET;
        }
        else if (clkVal && state == INPUT) {
          pins :> data;
          isa <: data;
          state = QUIET;
        }
        else if (clkVal && state == PENDQUIET) {
          isa <: 0;
          state = QUIET;
        }
        break;
      case (state == QUIET) => isa :> state :
        switch(state) {
          case OUTPUT :
            isa :> data;
            break;
          case INPUT :
            break;
        }
        break;
    }
  }
}
```

这个声明:

```
void clkTimeStrobe(port pins, port clk, chanend isa)
```

声明 clkTimeStrobe 为一个函数，输入引数包含可以取样和驱动数据的原始数据埠，以及可以用来取样时序边缘的原始埠，和用来与处理器接口的通道端点；型别 pwidth_t 和埠的宽度具有相同的大小。

该函数在原始埠进行以下的输入/输出操作:

- pins <: data;

  变量 data 里面的数据立刻在脚位上驱动。

- pins :> data;

  脚位上的值立即被读入并指定到变量 data 。

- clk when pinsneq(clkVal) :> clkVal;

  当时序脚位上的值不等于目前的值时，目前的值会被读进。

时序元件会等待两个时序边缘以及从处理器来的要求，当收到输出的要求之后，在时序的下个下降边缘，数据会被驱动且回送收到的信息给处理处；当收到一个输入的请求后，在下时序的取样边缘上，数据则被取样并且送至处理器。

下面的程序显示在处理器上执行的线程中使用有时序的埠，该线程进行和 §4.1 中例子相同的输出顺序 (注意：在 XCore 装置上，埠是直接和 ISA 介面接口的，而不是使用通道介面，所以，实际上这个程序是不会产生的)。

```
port p, c;

int main(void) {
  chan isa;
  par {
    clkTimeStrobe(p, c, isa);  // 由硬件实作
    for (int i=0; i<5; i++) {  // 由软件实作
      isa <: OUTPUT;
      isa <: i;
      isa :> int;
    }
  }
}
```

for 循环输出数据到函数 clkTimeStrobe ，接着该函数在其时序的下一个下降边缘驱动该笔数据，在 XS1 装置上，所有数据埠都是有缓冲的 (参见 §C.1)，也就是说，确认收到的动作几乎是立即发生。

注意，XC 的输入/输出语义不需要时序来提供规律间隔的边缘，甚至是完全不提供边缘，语义也不指定埠初始化的状态；输入/输出的语义因此在 for 循环中完整地描述，而与实作相关的初始化则出现在循环外部 (参见 §C.2)，输入/输出之时间特性是实作定义。

B.2 时序、时效性以及信号撷取元件

时效性的操作定义如下:

- 时效性输出使埠等待至其计数器等于所指定的时间，其之后的行为如同有时序性的输出。
- 时戳性输出使处理器等待至输出被驱动，并且记录这个时间点时埠计数器的值。
- 时效的输入使处理器等待至埠的计数器等于所指定的将来时间，其之后行为如同一个时效性输入。

如果使用准备输入撷取信号:

- 准备输入信号是在埠的时序上升边缘取样。
- 输入边缘发生在准备输入信号为高准位时埠的时序上升边缘。
- 当前一个上升边缘取样的准备输入信号为高准位时，输出边缘发生在埠时序的下降边缘。

如果使用准备输出撷取信号:

- 准备输出信号通常是驱动至低准位。
- 输出数据至少在一个埠的时序周期驱动，而准备输出信号只会在第一个周期驱动；是否继续驱动数据则是实作定义。
- 在数据被取样那个上升边缘之前的前一个下降边缘，准备输出信号会被驱动至高准位。

下面函数定义时序性/有时效性的/信号撷取元件，可以设定成使用准备输入和准备输出信号，并且可进行时效性和时戳性的操作。

```
void getInReq(chanend c, int &isTimed, counter_t &time, int &isTS) {
  /* 从通道输入'输入'请求的通信协议 */
  c :> isTimed;    // 取得时间  (控制)
  if (isTimed)
    c :> time;     // 取得时间  (数据)
  c :> isTS;       // 取得时戳  (控制)
}

void getOutReq(chanend c, pwidth_t &data, int &isTimed,
               counter_t &time, int &isTS) {
  /* 从通道输入'输出'请求的通信协议 */
  c :> data;       // 取得时间  (数据)
  c :> isTimed;    // 取得时间  (控制)
  if (isTimed)
    c :> time;     // 取得时间  (数据)
  c :> isTS;       // 取得时戳  (控制)
}
```

```
void clkTimeStrobe(port pins, int isReadyIn, port rin, int isReadyOut,
                   port rout, port clk,
                   chanend ser, chanend clrIn, chanend clrOut) {
  pwidth_t data     = 0;
  int state         = QUIET;
  int clkVal        = 0;
  int rinVal        = 0;
  int isTimed       = 0;
  counter_t counter = 0;
  counter_t time    = 0;
  int isTS          = 0;
  while (1) {
    select {
      case (state == QUIET) => ser :> state :
        switch(state) {
          case OUTPUT :
            getOutReq(ser, data, isTimed, time, isTS);
            break;
          case INPUT :
            getInReq(ser, isTimed, time, isTS);
            break;
          }
        break;
      case clk when pinsneq(clkVal) :> clkVal :
        if (!clkVal) { /* 下降边缘 */
          counter++;
          if (state == OUTPUT && (!isTimed || counter == time))
            state = R_OUT;
          else if (state == INPUT  && (!isTimed || counter == time))
            state = R_IN;
          if (state == R_OUT && (!isReadyIn || rinVal)) {
            pins <: data;
            if (isReadyOut) rout <: 1;
            state = PENDQUIET;
          }
          else if (state == R_IN && isReadyOut)
            rout <: 1;
          else if (isReadyOut)
            rout <: 0;
        }
        else { /* 上升边缘 */
          if (isReadyIn)
            rin :> rinVal;
          if (state == PENDQUIET) {
            ser <: 0; if (isTS) ser <: counter;
            state = QUIET;
          }
          else if (state == R_IN && (!isReadyIn || rinVal)) {
            pins :> data;
            ser  <: data; if (isTS) ser <: counter;
            state = QUIET;
           }
        }
        break;
      case clrIn :> int x : clrOut <: x; state = QUIET; break;
} } }
```

B.3 串行化元件

在串行化的埠上，输入和输出操作行为定义如下:

- 一个 w 个位元的值在 p 个脚位上输出，会在 w/p 个输出边缘上驱动，先送最低有效位元。
- 时效性或是时戳性输出操作所指定的时间，是从最前面的 p 个位元数据驱动算起，处理器会阻塞直到最后的 p 个位元被驱动。
- 在 p 个脚位上的 w 个位元值的输入，会在 w/p 个输入边缘上取样，较早收到的位元会被插入到 w 的最低有效位元。
- 时效性或时戳性输入操作所指定的时间，是从最前面 p 个位元数据从脚位读进来算起[1]。

下面的函数定义串行器元件:

```
void putOutReq(chanend cts, pwidth_t data, int isTimed,
               counter_t time, int isTS) {
  /* 输出‘输出’请求到通道的通信协议 */
  cts <: OUTPUT;
  cts <: data;
  if (isTimed) {
    cts <: 1;
    cts <: time;
  }
  else
    cts <: 0;
  cts <: isTS;
}

void putInReq(chanend cts, int isTimed, counter_t time, int isTS) {
  /* 输入‘输出’请求到通道的通信协议 */
  cts <: INPUT;
  if (isTimed) {
    cts <: 1;
    cts <: time;
  }
  else
    cts <: 0;
  cts <: isTS;
}
```

[1]在 XS1 装置上，时效性或是时戳性的输入操作所指定的时间，是从取样**最后**的位元时算起 (参见 §C.1.1)；这需要串行器持续维持在工作状态 (push model)，而不是被缓冲的元件启动 (pull model)，虽然预期上将来的 XMOS 架构中，会支援此语意，但并非保证。

```
void serialiser(int w, int p, chanend cts, chanend bufIn, chanend bufOut,
                chanend clrIn, chanend clrOut) {
  twidth_t data  = 0;
  counter_t time = 0;
  counter_t ts   = 0;
  int clr        = 0;
  int isTimed    = 0;
  int isTS       = 0;

  while (1) {
    select {
      case bufIn :> int op :
        switch (op) {
          case OUTPUT :
            getOutReq(bufIn, data, isTimed, time, isTS);
            for (int i=0; i<w/p; i++) {
              pwidth_t chunk = ((1 << p) - 1) & data;
              data >>= p;
              putOutReq(cts, chunk, (isTimed && i == 0), time, 1);
              cts :> int;                    // 同步
              cts :> ts;                     // 取得时戳  (数据)
            }
            bufIn <: 0;                      // 同步
            if (isTS) bufIn <: ts;           // 取得时戳  (数据和控制)
            break;
          case INPUT :
            getInReq(bufIn, isTimed, time, isTS);
            clr = 0;
            for (int i=0; i<w/p && !clr; i++) {
              if (i == 0)
                putInReq(cts, isTimed, time, isTS);
              else
                putInReq(cts, 0, time, 0);
              select {
                case cts :> pwidth_t chunk :  // 取得输入  (数据)
                  data = (data >> p) | (chunk << (w-p));
                  if (i == 0 && isTS)
                    cts :> ts;                // 取得时戳  (数据)
                  break;
                case clrIn :> int x :         // 清除      (控制)
                  clr = 1;
                  clrOut <: x;
                  break;
            } }
            if (!clr) {
              bufOut <: data & ((1<<w)-1);    // 存放输出  (数据)
              if (isTS) bufOut <: ts;         // 存放时戳  (数据)
            }
            break;
        }
        break;
      case clrIn :> int x :
        clrOut <: x;
        break;
} } }
```

B.4 缓冲元件

有缓冲但 FIFO 长度为 0 的埠，其定义如同缓冲元件是不存在的，下面的程序定义这种“穿越”式的行为，避免当缓冲不需要的时候，从中移除元件的需要。

```
void buffer(int size, int direction, chanend c, chanend, d, chanend isa,
            int isRout, chanend clrIn, chanend clrOut) {
  twidth_t  data    = 0;
  int       isTimed = 0;
  counter_t time    = 0;
  int       isTS    = 0;
  int       op      = 0;

  if (size == 0) {
    while (1) {
      isa :> int op;
      switch (op) {
        case OUTPUT :
          getOutReq(isa, data, isTimed, time, isTS);
          putOutReq(ser, data, isTimed, time, isTS);
          ser :> int;
          if (isTS) ser :> ts;
          isa <: 0;
          if (isTS) isa <: ts;
          break;
        case INPUT :
          getInReq(isa, isTimed, time, isTS);
          putInReq(ser, isTimed, time, isTS);
          ser :> data;
          isa <: data;
          if (isTS) {
            ser :> ts;
            isa <: ts;
      }  }
  } }
  else if (direction == OUT)
    portBufferOut(size, d, isa);
  else
    portBufferIn(size, d, e, isa, isRout, clrIn, clrOut);
}
```

B.4.1 FIFO 功能

以下单元中定义的缓冲元件使用标准先进先出数据结构 (FIFO)，并且用以下函数接口：

`void addTail(FIFO f, zhidth_t data, counter_t ts)`

加入一笔数据到 FIFO 尾部，如果 FIFO 已经满了，会移除最早的数据以腾出空间。

`{twidth_t, counter_t} getHead(FIFO f)`

回传在 FIFO 前面最早的数据并且从伫列中移除，如果 FIFO 是空的，回传的结果是未定义的。

int isEmpty(FIFO f)

如果 FIFO 是空的，回传非零的值，否则，回传零。

int isFull(FIFO f)

如果 FIFO 是满的，回传非零的值，否则，回传零。

B.4.2 有缓冲的输出

在有缓冲的埠上，输出行为定义如下:

- 输出操作会将数据插入至埠的 FIFO，并且在所有待完成的输出完成时进行输出，处理器会阻塞直到 FIFO 有空间接收数据为止。
- 时效性的输出操作使处理器等待至所指定的时间，然后进行输出的操作。

下面的函数定义缓冲输出元件:

```
void portBufferOut(chanend ser, chanend isa) {
  FIFO b             = EMPTY;
  twidth_t  data     = 0;
  int       isTimed  = 0;
  counter_t time     = 0;
  int       isTS     = 0;
  counter_t ts       = 0;
  int       wait     = 0;

  while (1) {
    if (!isEmpty(b) && !wait) {
      {data, time} = getHead(b);
      putOutReq(ser, data, (time != NOTIME), time, isTS);
      wait = 1;
    }
    select {
      case !isFull(b) => isa :> int op :
        switch (op) {
          case OUTPUT :
            getOutReq(isa, data, isTimed, time, isTS);
            isa <: 0;
            if (isTimed)
              addTail(b, data, time);
            else
              addTail(b, data, NOTIME);
            break;
          case INPUT :
            /* 实作定义的行为 */
            break;
         }
         break;
      case ser :> ts :
        if (isEmpty(b) && isTS)
          isa <: ts;
        wait = 0;
        break;
} } }
```

B.4.3 有缓冲区的输入

在有缓冲的埠上，输入行为定义如下:

- 在每个输入的边缘，数据被埠取样并插入至埠的 FIFO ，如果 FIFO 是满的，则最早的数据会被丢弃以腾出空间存放最新取样的值。
- 输入操作是从 FIFO 读入一笔数据，处理器会阻塞直到 FIFO 中含有数据。
- 如果有缓冲的埠被设定为有准备输出撷取信号，则当 FIFO 还没满的时候，准备输出信号在每个时序下降边缘会被驱动至高准位。
- 时效性输入的时间代表一个将来的时间，它会使处理器在进行输入之前，丢弃所有缓冲区中的数据。

下面的函数定义具缓冲的输入元件:

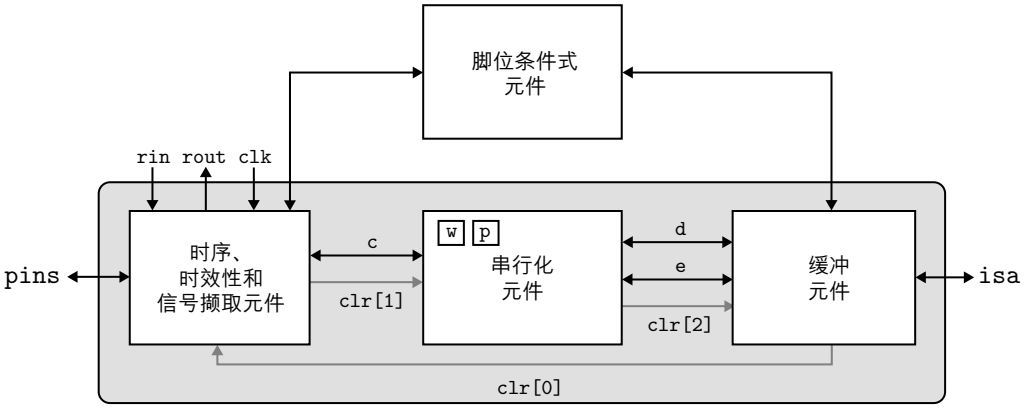
```
void portBufferIn(chanend serIn, chanend serOut, chanend isa,
                  int isReadyOut, chanend clrIn, chanend clrOut) {
  FIFO b            = EMPTY;
  twidth_t  data    = 0;
  int       isTimed = 0;
  int       isTS    = 0;
  counter_t time    = 0;
  counter_t ts      = 0;
  while (1) {
    if (!isReadyOut || !isFull(b))
      putInReq(serIn, 0, 0, 1);
    select {
      case !isEmpty(b) => isa :> int op :
        switch (op) {
          case OUTPUT : /* 实作定义的行为 */
            break;
          case INPUT :
            getInReq(isa, isTimed, time, isTS);
            if (!isTimed)
              {data, ts} = getHead(b);      // 取得缓冲输入  (数据)
            else {
              clrOut <: 1;                  // 清除任何闲置的输入
              select {
                case clrIn :> int :
                  doEmpty(b);
                  break;
                case serOut :> twidth_t :
                  clrIn :> int; doEmpty(b);
                  break;
              }
              putInReq(serIn, 1, time, isTS);
              serOut :> data;               // 取得输入  (数据)
              serOut :> ts;                 // 取得时戳  (数据)
            }
            isa <: data;                    // 存放输入  (数据)
            if (isTS) isa <: ts;            // 存放时戳  (数据)
            break;
        }
        break;
      case serOut :> data :                 // 取得输入  (数据)
        serOut :> ts;                       // 取得时戳  (数据)
        addTail(b, data, ts);               // 加到缓冲  (数据)
        break;
} } }
```

B.5 条件式输入：pinseq 和 pinsneq

在有时序的埠上的条件式输入行为是由语述函数定义的，语述函数 pinseq 和 pinsneq 定义如下:

- 函数 pinseq 的条件式输入，使埠在输入边缘取样数据，处理器直到取样到的 (埠宽) 位元的值等于所指定的参数值，处理器等待到此时间点，并取得最新取样到的数据。
- 函数 pinsnseq 的条件式输入，使埠在输入边缘取样数据，直到取样到的 (埠宽) 位元的值不等于所指定的参数值，处理器等待到此时间点，并取得最新取样到的数据。
- 时效性的条件输入，使处理器等待至埠计数器等于所指定的时间，接着其行为如同一个条件式输入。

语义上，这两个函数被定义为**脚位条件式**元件的一部分，出现在时序的/时效性/信号撷取元件以及串行器元件之间。



这个元件需要通信协议，透过用来完成条件的缓冲以及串行化元件，进行通讯输入需求，条件可以是**无**、**脚位等于**或是**脚位不等于**，下面函数定义脚位条件式元件，对这个要被整合的元件:

- 时序的/时效性的/信号撷取元件 (参见 §B.2) 被改为可以在另外的通道上接受及回覆要求。
- 缓冲输入元件 (参见 §B.4) 被改为接受条件式的输入要求；如果当有条件式的输入要求来到时，缓冲元件会清除所有悬置的输入 (如有时效性的输入一般)，接着透过另一个通道和这个元件通讯。

```
void pinsConditional(chanend cts, chanend buf) {
  pwidth_t data     = 0;
  counter_t time    = 0;
  counter_t ts      = 0;
  int isTimed       = 0;
  int isTS          = 0;
  int n             = 0;
  int cond          = 0;
  pwidth_t condData = 0;
  int matched       = 0;

  while (1) {
    buf :> int;                       // 取得输入请求 (控制)
    getInReq(buf, isTimed, time, isTS);
    buf :> cond;
    buf :> condData;                  // 取得符合情况 (数据)
    n = 0;
    matched = 0;
    while (!matched) {
      putInReq(cts, (isTimed && n == 0), time, isTS);
      cts :> data;                    // 取得输入 (数据)
      if (isTS)
        cts :> ts;                    // 取得时戳 (数据)
      switch (cond) {
        case NONE :
          matched = 1;
          break;
        case PINSEQ :
          matched = (data == condData);
          break;
        case PINSNEQ :
          matched = (data != condData);
          break;
    } }
    buf <: data;                      // 送出资料 (数据)
    if (isTS)
      buf <: ts;                      // 送出时戳 (数据)
    break;
  }
}
```

附录 C

XS1 上之 XC 实作

下面的单元说明用 XC 在 XS1 上实作，包括输入/输出规格之实作范围、标准埠函数、埠之于脚位的对应以及类型大小的对齐。

C.1 XC 埠规格的支援

在 XC 中，埠的声明如下:

```
port p;
```

此语句声明一个原始埠，在 XS1 装置中，所有用来输入和输出数据的埠都是以 100MHz 为参考时序 (参见 §C.2.1)，并且使用单一栏位的缓冲区，即使其声明中没有关键字 buffered 限定符。

下表可以被用来决定在 XS1 上埠，哪些输入/输出操作是支援的，依据相对应的 XC 声明是否有关键字 buffered 来判定。

	操作		
模式	串型化	信号撷取	@ when
无限定符	✗	✗	✗
buffered	✓	✓	✓

编译器必须要侦测出并且发出以下的错误讯息:

- 串行化: 对没有 buffered 属性的埠，声明不同于埠宽的传输宽度。
- 信号撷取: 将没有 buffered 属性的埠设定成使用准备输入或准备输出信号。
- @ when : 这两个运算符是同时用于输入语句中，且语句中含有一个不具有 buffered 限定符的埠声明。

C.1.1 串行化

当使用串行化 (参见 §B.3) 时，时效性操作的时间纪录数据的**最后**位元取样的时间；这可能会在使用串行化时造成预期外的行为，因为以下的结构：

```
par {
  p @ t <: x;
  q @ t :> y;
}
```

会使埠 p 在埠 q 输入完成的同时开始输出，为了使数据可以并发地输入和输出，输入的时间应该要以软件中补偿，补偿的量是传输宽度除以埠宽。

C.1.2 时戳印记

输入语句纪录的时戳可能会是数据取样之后的时间，这是由于 XS1 提供不同的指令达到输入数据和输入时戳，因此，这个时戳可以在下个数据取样之后才输入；这情况也对输出语句有影响，但是不影响在 `select` 语句中的守卫语句。编译器应该要在执行一个输入或是输出语句之后立即输入时戳，因此实际上，这样的行为并不常见。

C.1.3 实作定义行为

对于具有限定符 `buffered` 的埠，任何对其方向的改变，会造成无定义的行为。

C.2 XS1 埠库函数：<xs1.h>

头文件 <xs1.h> 用来声明设定埠的操作模式的函数，以及用来操作埠的函数。

C.2.1 时序设定

XS1 装置提供单一个参考时序，该时序频率源至一外部振荡器，XC 需要系统设计者确保参考时序是 100MHz，使计时器能够正确操作。

每个 XCore 皆提供一组可编程的**时序块**，用来产生埠的时序信号；一个时序块可以使用一位元的埠或是参考时序的除频。头文件 <xs1.h> 中定义资源类型 `clock`，该类型的变量必须是全域变量并且以独一的时序块资源的标识符初始化，如下所示：

```
clock c = XS1_CLKBLK_1;
```

能够使用的时序块数在装置的规格书上有说明，其名称是从 1 开始依序编号，如上声明所示。

以下的函数是用来设定时序块：

`void configure_clock_src(clock clk, void port p)`

`configure_clock_src` 设定时序以一位元的埠当成其来源，如果埠不是一位元，会产生异常。

void configure_port_clock_output(void port p, const clock clk)

configure_port_clock_output 设定一位元的埠来驱动时序信号，如果埠不是一位元，会产生异常，如果时序是内部产生的并且频率是 100MHz，则不会有输出，在设定为这样的埠上的输入或是输出会导致无定义行为；速度为 100MHz 的时序是需要的，才能有正确的操作。

configure_clock_rate(clock clk, unsigned a, unsigned b)

这个函数定义时序速度为 $\frac{a}{b}$MHz，如果给定的速度硬件 不支援，会产生异常，该硬件支援 100MHz 并且是 $(50/n)$MHz 的倍率形式，其中的 n 的范围是 1 至 255 之间，含 1 以及 255。

void configure_clock_rate_at_most(clock clk, unsigned a, unsigned b)

void configure_clock_at_least(clock clk, unsigned char divide)

这些函数设定时序这些函数设定时序使用硬件所支援的最快/最慢非零速度，也就是小于或是等于 $\frac{a}{b}$ MHz，如果没有任何符合的速度，则会有异常产生；速度为 100 MHz 的时序是需要的，才能有正确的操作。

void set_clock_fall_delay(clock clk, unsigned n)

void set_clock_rise_delay(clock clk, unsigned n)

这些函数使埠收到时序的上升/下降边缘前，延迟 n 个处理器时序周期，该暂缓数值必须在范围 0 到 512 之间，包含 0 和 512；如果时序边缘延迟超过时序周期，则所有连接到该时序的埠皆不会收到下降/上升边缘。

void start_clock(clock clk)

start_clock 使时序开始产生边缘，且将连接到该时序的所有埠计数器都重设为 0。

void stop_clock(clock clk)

stop_clock 使处理器等待直到时序为低准位，接着使其处于停止状态 (在该状态下不会产生边缘)。

C.2.2 埠的设定

下面函数用来改变埠的操作模式:

```
void configure_in_port(void port p, const clock clk)
void configure_in_port_strobed_master
       (void port p, out port readyout, const clock clk)
void configure_in_port_strobed_slave
       (void port p, in port readyin, clock clk)
```

```
void configure_in_port_handshake
       (void port p, in port readyin, out port readyout, clock clk)
```

这些函数将时序以及输入模式的准备输入信号、准备输出信号设定给埠，如果准备输入或是准备输出埠不是一位元，则会产生异常 (参见 §B.2 和 §C.1)。

```
void configure_out_port(void port p, const clock clk, unsigned initial)
void configure_out_port_strobed_master
       (void port p, out port readyout, const clock clk, unsigned initial)
void configure_out_port_strobed_slave
       (void port p, in port readyin, clock clk, unsigned initial)
void configure_out_port_handshake
  (void port p, in port readyin, out port readyout, clock clk, unsigned initial)
```

这些函数将时序以及输入模式的准备输入信号、准备输出信号设定给埠，并且使初始值立即被送出，如果准备输入或是准备输出埠不是一位元，则会产生异常 (参见 §B.2 和 §C.1)。

```
void set_pad_delay(void port p, unsigned n)
```

set_pad_delay 在连接到埠的脚位上设置延迟，且埠的脚位所取样到的输入信号会延迟 n 个处理器时序周期，才会被埠收到，硬件所支援的暂缓值是 0 至 5 之间，包含 0 和 5；如果多个使用的埠连接到相同的脚位 (参见 §C.3)，在脚位上的延迟值会被设定为最高优先埠的延迟值。

```
void set_port_inv(void port p)
void set_port_no_inv(void port p)
```

set_port_inv 设定一位元的埠，将其脚位取样和驱动的数据反向，如果该埠是时序的来源，这个设置会使时序上升边缘和下降边缘对调，set_port_no_inv 保证该埠不会使数据反向；如果埠不是一位元的，则会产生异常。

```
void set_port_sample_delay(void port p)
void set_port_no_sample_delay(void port p)
```

这些函数设置埠的取样边缘，第一个函数设置取样边缘为埠的时序下降边缘，第二个设置为上升边缘，如果延迟取样被设定为时序的下降边缘，有准备输出信号的时效性输入会使准备输出信号在所指定的时间被驱动，且输入完成会是在一个周期之后。

```
void set_port_drive(void port p)
void set_port_pull_up(void port p)
```

这些函数将埠设定为驱送或是上拉模式 (默认是驱送模式，其行为说明于附录 B 中)，在上拉模式中，当埠是用来输入，其内部的上拉电阻被开启，如果埠不是一位元宽，在这脚位上驱动出的值则无定义；对于一位元的埠，输出 0 会使其本身脚位变为低准

为，而输出 1 则不会有任何值驱动。当脚位不驱动数据时，上拉电阻确保该埠取样的值为高，这个上拉的作用并不足够保证定义的外部数值。

C.2.3 输入和输出操作

void pinseq(void port p, unsigned val)
void pinsneq(void port p, unsigned val)

这些函数需要有由埠来的条件式输入 (参见 §B.5)，这些函数必须在 when 表达式中的埠的输入中调用，并且在调用时省略埠。

void timerafter(timer t, unsigned val)

timerafter 让处理器等待，直到在一个给定的时间之后，计时器的计数器数值被解译为来到的情况，如果 ((int)(*B* - *A*) < 0) 的结果为真，则时间 *A* 被当成是在 *B* 时间点之后来到，这些函数必须在 when 表达式中的计时器上的输入调用，并且在调用时省略计时器。

void partout(void port p, unsigned bits, unsigned val)
unsigned partout_timestamped(void port p, unsigned bits, unsigned val)
unsigned partout_timed(void port p, unsigned bits, unsigned val, unsigned t)

这些函数输出 val 的最低有效 bits 个位元至所指定的埠，第二个函数将输出做一个时戳记号，第三个则是当埠计数器等于所给定的时间时，驱动输出。埠必须声明为 buffered 限定特性，位元的数量必须小于埠的传输宽度，并且大于零也必须是埠宽的倍数，否则会产生异常。

unsigned endin(void port p)

endin 使目前埠的输入终止，这个埠必须声明有 buffered 限定符，函数传回值是这个埠取样但尚未被处理器输入的位元数，包括在 FIFO 中的数据或是串行寄存器。此埠上接下来的输入使该埠提供传输宽度的位元数据，直到剩下一个小于一个传输宽度位元数据，而剩下的缓冲数据会在这个值的最高有效位元， 任何剩下数据则可在之后输入被读取。

void sync(void port p)

sync 使处理器等待直到埠将所有悬置的输出驱动，并且最后埠宽位元数据在这些脚位上会被保留一个时序周期。

void clearbuf(void port p)

clearbuf 使埠丢弃 FIFO 中的任何数据，如果埠正在其脚位上驱动数据，数据会继续驱动；如果埠是串行输出，则正在送的数据仍会送出，但剩下的数据则被丢弃。

unsigned peek(void port p)

peek 使埠对其脚位目前的值进行取样，埠会立即提供取样的埠宽位元的数据给处理器，无论其传输宽度、时序、准备信号以及缓冲的状态，这个输入对之后埠上的输入/输出操作并没有作用。

C.3 埠和脚位对应

在 XS1 装置上，脚位是透过埠和外部元件接口，以及建构和其他装置间的连接，而通道就在这连接上；埠是多路复用的，这允许脚位能被设为不同宽度的埠来使用。

表格 I 是 XS1 的埠和脚位之间的对应，其说明如下：

- 每个脚位的名称是以格式 X*n*D*pq* 命名，其中 *n* 是装置上有效的 XCore 编号，而 *pq* 则在表格中，脚位的实体位置是以封装而有不同，在规格书上有提供该资讯。
- 每个连接是由 A — D 的字母判定，而连接的线是利用下标数字 0 — 4 来判定。
- 每个埠是用其宽度 (第一个数字是 1、4、8、16 或是 32) 以及区别相同宽度的多重埠 (A — P) 的字母来判别；这些名称对应到在头文件 <xs1.h> 中的埠标识符 (举例说明：埠 1A 是对应到标识符 XS1_PORT_1A)，埠的个别位元则是利用数字下标 0 — 31 来判定。
- 这表格划分成六列 (或是**组**)，前四组是一、四以及八位元宽的埠，后面的两组则是单一 32 位元的埠，不同的封装会有不同数量的组分类，在小装置上，16 和 32 位元埠则不支援。

程式中使用的埠是用 XC 埠声明决定的，如下面的声明：

```
on stdcore[0] : in port p = XS1_PORT_1A;
```

使用在 XCore 0 上的一位元埠 1A，该埠是接到脚位 X0D00。

一般来说，设计人员应该要确保埠声明使用的脚位是没有重复的，但是优先顺序已经设计成如果该情况发生，当和宽度较窄的埠重复时，可以使用较宽的埠。表格中左边的埠比右边的埠具有较高的优先顺序，如果两个声明是使用不同脚位，则较窄的埠具有高优先顺序，例如：

```
on stdcore[2] : out port p1 = XS1_PORT_32A;
on stdcore[2] : out port p2 = XS1_PORT_8B;
on stdcore[2] : out port p3 = XS1_PORT_4C;
```

在上面例子中：

- 埠 p1 上的输入/输出使用脚位 X2D02 到 X2D09 以及 X2D49 到 X2D70。
- 埠 p2 上的输入/输出使用脚位 X2D16 到 X2D19，从 p2 输入会导致位元 0、1、6、7 产生无定义的值。
- 埠 p1 上的输入/输出使用脚位 X2D14、X2D15、X2D20 和 X2D21，从 p3 输入会导致位元 28 — 31 产生无定义的值，并且在输出时，这些位元不会被驱动。

表格 I
每个脚位可以使用的埠以及连接

脚位	连接	⇐ 最高 一位元埠	优先顺序 四位元埠	八位元埠	最低 ⇒ 十六位元埠	三十二位元埠
X*n*D00		$1A$				
X*n*D01	A^4 输入/输出	$1B$				
X*n*D02	A^3 输入/输出		$4A^0$	$8A^0$	$16A^0$	$32A^{20}$
X*n*D03	A^2 输入/输出		$4A^1$	$8A^1$	$16A^1$	$32A^{21}$
X*n*D04	A^1 输入/输出		$4B^0$	$8A^2$	$16A^2$	$32A^{22}$
X*n*D05	A^0 输入/输出		$4B^1$	$8A^3$	$16A^3$	$32A^{23}$
X*n*D06	A^0 输出/输入		$4B^2$	$8A^4$	$16A^4$	$32A^{24}$
X*n*D07	A^1 输出/输入		$4B^3$	$8A^5$	$16A^5$	$32A^{25}$
X*n*D08	A^2 输出/输入		$4A^2$	$8A^6$	$16A^6$	$32A^{26}$
X*n*D09	A^3 输出/输入		$4A^3$	$8A^7$	$16A^7$	$32A^{27}$
X*n*D10	A^4 输出/输入	$1C$				
X*n*D11		$1D$				
X*n*D12		$1E$				
X*n*D13	B^4 输入/输出	$1F$				
X*n*D14	B^3 输入/输出		$4C^0$	$8B^0$	$16A^8$	$32A^{28}$
X*n*D15	B^2 输入/输出		$4C^1$	$8B^1$	$16A^9$	$32A^{29}$
X*n*D16	B^1 输入/输出		$4D^0$	$8B^2$	$16A^{10}$	
X*n*D17	B^0 输入/输出		$4D^1$	$8B^3$	$16A^{11}$	
X*n*D18	B^0 输出/输入		$4D^2$	$8B^4$	$16A^{12}$	
X*n*D19	B^1 输出/输入		$4D^3$	$8B^5$	$16A^{13}$	
X*n*D20	B^2 输出/输入		$4C^2$	$8B^6$	$16A^{14}$	$32A^{30}$
X*n*D21	B^3 输出/输入		$4C^3$	$8B^7$	$16A^{15}$	$32A^{31}$
X*n*D22	B^4 输出/输入	$1G$				
X*n*D23		$1H$				
X*n*D24		$1I$				
X*n*D25		$1J$				
X*n*D26			$4E^0$	$8C^0$	$16B^0$	
X*n*D27			$4E^1$	$8C^1$	$16B^1$	
X*n*D28			$4F^0$	$8C^2$	$16B^2$	
X*n*D29			$4F^1$	$8C^3$	$16B^3$	
X*n*D30			$4F^2$	$8C^4$	$16B^4$	
X*n*D31			$4F^3$	$8C^5$	$16B^5$	
X*n*D32			$4E^2$	$8C^6$	$16B^6$	
X*n*D33			$4E^3$	$8C^7$	$16B^7$	
X*n*D34		$1K$				
X*n*D35		$1L$				
X*n*D36		$1M$		$8D^0$	$16B^8$	
X*n*D37		$1N$		$8D^1$	$16B^9$	
X*n*D38		$1O$		$8D^2$	$16B^{10}$	
X*n*D39		$1P$		$8D^3$	$16B^{11}$	
X*n*D40				$8D^4$	$16B^{12}$	
X*n*D41				$8D^5$	$16B^{13}$	
X*n*D42				$8D^6$	$16B^{14}$	
X*n*D43				$8D^7$	$16B^{15}$	
X*n*D49	C^4 输入/输出					$32A^0$
X*n*D50	C^3 输入/输出					$32A^1$
X*n*D51	C^2 输入/输出					$32A^2$
X*n*D52	C^1 输入/输出					$32A^3$
X*n*D53	C^0 输入/输出					$32A^4$
X*n*D54	C^0 输出/输入					$32A^5$
X*n*D55	C^1 输出/输入					$32A^6$
X*n*D56	C^2 输出/输入					$32A^7$
X*n*D57	C^3 输出/输入					$32A^8$
X*n*D58	C^4 输出/输入					$32A^9$
X*n*D61	D^4 输入/输出					$32A^{10}$
X*n*D62	D^3 输入/输出					$32A^{11}$
X*n*D63	D^2 输入/输出					$32A^{12}$
X*n*D64	D^1 输入/输出					$32A^{13}$
X*n*D65	D^0 输入/输出					$32A^{14}$
X*n*D66	D^0 输出/输入					$32A^{15}$
X*n*D67	D^1 输出/输入					$32A^{16}$
X*n*D68	D^2 输出/输入					$32A^{17}$
X*n*D69	D^3 输出/输入					$32A^{18}$
X*n*D70	D^4 输出/输入					$32A^{19}$

C.4 通道通讯方式

在 XS1 架构的一些版本中，用 select 语句中的守卫条件式，从串流通道输入少于 32 位元的数据是不能实现的。

C.5 数据类型

XC 语言本身没有给定数据类型的大小及对齐，这允许 int 大小可以被设定成目的装置本身的字元组大小，对许多运算，这样可以达到可能的最高效能；也使对齐可以被设定成足够宽使记忆体有效率的载入和储存。表格 II 说明 XMOS 应用二进制接口[8]当中所给出的数据类型的大小以及对齐，该文件提供连接 XC 和 C 编译产生的目的文件的介面，除此之外:

- 类型 char 默认是无号数 (unsigned)。
- 类型 char、short 和 int 可以被用在位元栏位的声明。
- enum 位元栏位是无号数 (unsigned)，除非是负数。
- 一个零宽度的位元栏位强制填满至下个位元平移量，并且和位元栏位的声明类型对齐。
- 埠的名义传输类型是 unsigned int (32 位元)。
- 埠的名义计数器类型是 unsigned short (16 位元)。
- 计时器的名义计数器类型是 unsigned int (32 位元)。

表格 II
在 XS1 装置上数据类型的大小以及对齐

数据类型	大小 (位元数)	对齐 (位元数)	支援 XC	支援 C	意义
char	8	8	✓	✓	字符类型
short	16	16	✓	✓	短整数
int	32	32	✓	✓	一般整数
long	32	32	✗	✓	长整数
long long	64	32	✗	✓	倍位长整数
float	32	32	✗	✓	32 IEEE 浮点数
double	64	32	✗	✓	64 位元 IEEE 浮点数
long double	64	32	✗	✓	64 位元 IEEE 浮点数
void *	32	32	✗	✓	数据指针
port	32	32	✓	✗	埠
timer	32	32	✓	✗	计时器
chanend	32	32	✓	✗	通道端点

参考文献

[1] David May. *The XMOS XS1 Architecture*. XMOS Limited, 2009.

[2] Douglas Watt and Huw Geddes. *The XMOS Tools User Guide*. XMOS Limited, 2009.

[3] Peter Hedinger and Ali Dixon and Ross Owen and Neil Richards and David May and Henk Muller. XS1 Ports: use and specification. Website, 2008. `http://www.xmos.com/published/xs1-portspec`.

[4] Hitachi TX14 Series LCD. Website, 2008. `http://www.xmos.com/references/htx14`.

[5] IEEE 802.3 Section 2. Website, 2008. `http://www.xmos.com/references/ieee802`.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Press, Upper Saddle River, NJ, USA, 1988.

[7] International Organization for Standardization. *ISO/IEC 9899:1999: Programming Languages — C*. Wiley, West Sussex, England, December 1999.

[8] Douglas Watt and Richard Osborne and Martin Young. XMOS XS1 32-Bit Application Binary Interface. Website, 2009. `http://www.xmos.com/published/abi32`.

索引

六划

七划

八划

九划

十四划

十六划

十八划

符号

A

B

C

D

E

F

G

I

L

M

N

O

P