第一章

Neuron C 简介

Neuron C 是专门为 Neuron 芯片设计的编程语言。它是从 ANSI C 中派生出来的，相对 ANSI C 而言，它进一步扩展了用以支持由 Neuron 芯片中的固件提供的各种运行特征，使之成为开发 LONWORKS 应用的强有力工具。

对 ANSI C 的扩展包括以下几点：

- 一个内部多任务调度程序

它允许程序员以自然的方式来表达逻辑并行事件驱动的任务，同时控制这些任务执行的优先级。

- 将 I/O 对象直接映射到处理器的 I/O 能力
- 网络变量对象定义

提供一种简单的实现节点之间数据共享的方法。

- when 语句

引入事件并定义这些事件的临时排序。

- 显式消息传递

用于直接对 LonTalk 协议的下层进行访问。

- 秒及毫秒软件定时器对象

可随意执行用户任务。

- 函数库

当调用时，可以执行事件检测、输入/输出管理、网上发送或接收消息以及控制各种 Neuron 芯片的功能。

Neuron C 的扩展部分与 C 语言有着非常自然的衔接，它同时提供有内嵌的类型检查为编程人员编写高效率的分布式 LONWORKS 应用程序提供了保证。

1.1 Neuron C 与 ANSI C 的差别

Neuron C 非常贴近 ANSI C 语言标准，然而 Neuron C 又不完全遵守 Standard C 的标准（X3-J11）。除了有上述的扩展外还有一些不同的地方。

- Neuron C 不支持 C 的浮点运算或浮点运算符。但是提供有浮点库来使用浮点数（标准 IEEE 754）；
- Neuron C 只定义了 8 位的短整型（int）和 16 位的长整型，默认的是短整型。对 32 位的整型数的使用只能依靠 32 位的有符号整型库；

- Neuron C 不支持寄存器变量以及寄存器的存储类别；
- Neuron C 在自动变量(auto)定义时不赋初值；
- Neuron C 中结构(struct) 或共用体(union) 不能作为过程参数或作为函数的返回值；
- 不支持指向定时器、消息标签以及 I/O 对象的指针变量；
- 网络变量指针以及 E^2PROM 变量指针同常数指针（通过指针访问的变量内容是只读内容）在特定的环境并附加某些约束，可以用指针修改存储器的内容。见 eeprom_memcpy() 函数以及 propagate() 函数；
- 宏展开后宏参数才能被扫描，这样，在嵌套的宏展开中宏运算符 ' # ' 以及 ' # # ' 不能像在 ANSI C 标准中定义的那样得出结果；
- 网络变量名以及消息标签名最多由 16 个字符组成；
- Neuron C 中只保留了很少一部分 ANSI C 的库函数，如：memcpy() 和 memset()。有字符串以及字节操作库，但也仅仅是保留了 ANSI C 的<string.h> 包含文件中定义的一部分。其它 ANSI C 库函数，如文件的输入/输出以及存储分配函数，在 Neuron C 中是找不到的；
- Neuron C 中有 3 个 ANSI C 文件，即<stddef.h>、<stdlib.h>和<limits.h>；
- Neuron C 中如果在函数定义之前调用函数必须对函数进行说明；
- Neuron C 中附加有 ANSI C 没有的保留字及语法；
- Neuron C 不仅支持十进制、十六进制还支持二进制常数。二进制常数表示方法 0b<二进制数>，如 0b1101＝13（十进制）；
- Neuron C 程序注解可使用两斜杠（ // ）开始，也可以延用 /* */。如果使用 // 开始注解，那么注解结束不需要再标识，如：

```
... C code   /* 注解 */
... C code   // 注解
```

- 不需要 main()；
- Neuron C 在单个的编译单元不支持多个源文件（不过，支持#include 指令）；
- 不支持 ANSI C 预处理指令 #if、#elif 以及 #line，但支持 #ifdef、#else 以及 #endif。

1.2 数 据 类 型

1.2.1 整型数据

1. 整型常数

负常数被看作对正常数做一元负操作，比如， －128 应是有符号长整型，而非有符号短整型。十进制的整型常数有如下默认类型：

0 .. 127　有符号短整型

128 .. 32767　有符号长整型

32768 .. 65535　无符号长整型

默认的类型可以用 u、U、l 以及 L 后缀作修饰，例如：

0L　　有符号长整型
128U　无符号短整型
128UL　无符号长整型
256U　无符号长整型

十六进制、十进制以及二进制常数有如下默认类型，也可以采用上述的后缀字母作修饰：

0x0 .. 0x7f　　有符号短整型
0x80 .. 0xff　　无符号短整型
0x100 .. 0x7fff　有符号长整型
0x8000 .. 0xffff　无符号长整型

2. 整型变量

整型变量是没有分数部分的所有的数，如：1、2、3、－5、0 等等。整型变量可以有正的或负的值。

在整型的表示符 int 前，还可以加上一个或两个以下的符号：short（可省略）、long、unsigned 以及 signed(可省略）。加上这些符号后，在整型(int）之外又构成短整型、长整型、无符号长整型以及无符号短整型，一共 4 种。

long int　　16位有符号长整型（－32768～32767）
unsigned long int　16位无符号长整型（0～65535）
int　　8位有符号短整型（－128～127）
unsigned int　8 位无符号短整型（0~255）

1.2.2 字符变量

字符变量主要有两种：一种是有符号字符变量；一种是无符号字符变量。无符号字符可用来表示一个字节或 0～255 之间的整数，有符号字符可以用来表示一个 ASCII 字符，也可表示 －128～＋127 之间的整数。字符和无符号字符在内存中占一个字节。字符变量用关键字 char 来定义，如：

signed char　　8 位数
[unsigned] char　8 位数

1.2.3 枚举类型

如果一个变量只有几种可能的值，可以将该变量定义为枚举类型。所谓“枚举”是指将变量的值一一列举出来，变量的值只限于列举出来的值的范围。

定义枚举类型用 enum 开头。如：

enum (int 型)　8位数

Neuron C 中有一个枚举类型变量 boolean，即：

```
typedef enum {FALSE, TRUE} boolean;
```

1.2.4 其它

Neuron C 中还为 I/O 对象以及 SNVTs 预定了许多变量类型（见第五章相关内容）。在

扩展了的数学函数库中还定义了 fl_type 以及 s32_type（标准 IEEE 754）(见第六章内容)。

1.3 变量的定义

、在 C 语言中，所有的变量在使用前要定义，ANSI C 和 Neuron C 支持的变量定义如下：

定义	举例
简单的数据项	int a, b, c;
数据类型	typedef unsigned long ULONG;
枚举	enum hue {RED, GREEN, BLUE};
指针	char *p;
函数	int f(int a, int b);
数组	int a[4];

结构和共用体	struct s
	{
	int field1;
	unsigned field2 : 3;
	unsigned field3 : 4;
	};

Neuron C 附加的对象定义：

定义	举例	
I/O 对象	IO_0 output oneshot relay_trigger;	//详见第五章
定时器	mtimer led_on_timer;	//详见第二章
网络变量	network input int temperature;	//详见第三章
消息标签	msg_tag command;	//详见第四章

1.4 变量的初始化

不同类别的变量，它们的初始化发生在不同的时刻。除网络变量外， const 变量必须初始化，且发生在应用映像第一次装载时。要提醒读者注意的是， const ram 变量被放置在片外的 RAM 中，片外的 RAM 必须是非易失，这样 eeprom 以及 config 变量也可以在装载时被初始化。当然，如果变量定义中使用了 uninit 关键字，eeprom 变量在程序装载时不能初始化。

全局 RAM 变量在节点复位时被初始化，初始化默认值是“ 0 ”。由于初始化“ 0 ”值是固件的作用，所以不会因此花费额外的代码空间。

I/O 对象的初始化、输入网络变量（不包括 eeprom、config 或 const 网络变量）以及定时器的初始化都发生在复位时。网络变量的默认初始值是“ 0”。

非静态局部变量不会自动初始化，当程序运行离开该局部变量定义的函数范围或 when 任务范围，其值不保留。

1.5 变量的存储类别

在文件范围（指的是不包含有函数以及任务的 Neuron C程序部分）内定义且未指定类别的数据或函数是全局变量或外部函数。全局变量（包括定义时加有关键字 static的静态全局变量）正如名字所示，程序中所有函数、任务以及事件都可以引用。当节点 Neuron 芯片加电或复位时，在 RAM 中的全局变量将赋予初值。属于 eeprom 或 config 类别的变量仅在应用映像第一次装载时被初始化。

Neuron C 支持的 ANSI C 存储类别有：

auto （自动）局部变量默认的存储类别

const（常数）应用程序不能修改的存储类别

extern （外部）在其它模块里定义的变量或函数，如：函数库或系统映像

static（静态）链接时其它模块不能引用的变量或函数。如果该变量是在某个函数中或任务中的局部变量，它的值在调用之间被保留，但编译时其它函数不能引用

除上述的存储类别，Neuron C还增加有下列类别：

config （配置）该关键字只能在输入网络变量定义时加入。配置网络变量主要用于应用配置，它相当于 const eeprom。当应用映像第一次装载时被初始化

network （网络）该类别关键字定义一个网络变量

System（系统）该类别关键字在 Neuron C中仅用于访问 Neuron 芯片的固件函数库。应用编程人员在变量定义以及函数定义时不能使用该关键字

uninit 不初始化）该类别修饰字与 eeprom 关键字结合使用，用于指定 E^2PROM 变量在程序装载时或网络上再次装载时不用初始化或修改

应用编程人员还可以使用 Neuron C 中的下列关键字将应用代码直接存放到指定的存储器中(详见第二章有关内容)。

eeprom

far

offchip(只有 3150芯片有外部存储器)

onchip

ram （外接的 RAM 区）

这些关键字对 Neuron 3150芯片特别有用，原因是 3150 芯片的地址空间很大一部分被映射到片外。

1.6 函 数 原 型

如果函数在定义之前被调用，Neuron C 需要使用函数原型。有效的函数原型如下：

```
void f(void);
```

```
int g(int a, int b);
```

下面的函数是无参函数，所以不考虑函数原型，仅仅是进一步说明。

```
void f();
g();  //默认返回值是整型
```

如果调用之前已定义该函数， Neuron C 将自动创建一个内部原型。一个给定的函数只创建一个原型。下面的例子中当函数定义时， Neuron C 将为它们创建原型。

```
void f()
 {* 函数体 */ }

g (a,b)
int a;
int b;
 {/*函数体 */}
```

1.7 编 译 指 令

编译指令#Pragma可以用来设置某一个 Neuron 芯片的系统资源以及节点参数，诸如缓存器数和大小、接收事务数等。

Neuron C 附加的#pragma 指令可以用于控制 Neuron 芯片特定的参数，这些指令可以放置在源程序的任意地方，下面是这些编译指令的定义。

1) #pragma all_bufs_offchip

该指令仅用于 LonBuilder MIP/DPS 。编译器命令固件以及链接器将所有的应用及网络缓存器放置到片外的 RAM 中。

2) #pragma codegen <option>

这个指令允许对编译器代码发生器的某些特性进行有限的控制。应用的定时及代码长短可以用这个指令来改变。选项可以是如下几项：

expand_stmts_off
expand_stmts_on
no16bitstkfn
nofastcompare
noptropt
noshiftopt
nosiofar
optimization_off
optimization_on

上面给出的有些选项目的是为了与 Neuron C 编译器的早先版本兼容，还有就是与 LonBuilder 3.0 以前的版本兼容。no16bitstkfn、nofastcompare、noptropt 以及 noshiftopt 四个选项使编译器优化功能失去作用。

因为在某些环境条件下，新的编译器的优化可能会增加代码的长度，所以有可能会出现将在 3.0 版本以前的 3120 芯片中编译和链接的程序放到版本 3.0 内编译并不合适。

在调试程序时可能要选用 noptropt 选项，原因是调试程序(debugger) 无法知道是否编译器已经排除了发生在语句边界之间的指针的过多加载。在这种情况下，如果设置断点，调试程序的指针变量修改不能修改编译器在后面的语句中可能还要使用的加载指针寄存器。如果使用 noptropt 选项的编译指令，就可以避免上述问题，但又可能在合成代码时降低性能。所以,#pragma codegen <option>只在调试程序中使用。

expand_stmts_off 以及 expand_stmts_on 选项用来控制编译器代码产生器的动作。通常该选项默认是关闭。如果要在将代码存储到可写存储器的指令中设置断点，指令代码至少要两字节长。由于优化，有些指令语句的代码长度可低于 2 字节。使用 expand_stmts_on 选项，代码产生器将被告知要确保每条指令语句至少有 2 字节的代码，如果必要的话可插入空指令 NOP。

optimization_off 以及 optimization_on 选项同样也控制编译器代码产生器的动作，通常默认的是 optimization_on。为防止编译器的代码优化器将两到多条指令语句合在一起，使得调试程序时无法设置断点。该选项可用来关闭编译器的优化器。

3) #pragma disable_mult_module_init

该指令指定编译器直接在特定的启动以及事件块内产生所需的初始化代码。该选项被选择的结果就是直接插入。直接插入的方法在存储器使用的效率方面略有改善。该编译指令只能用于 3120 芯片，如：将程序硬塞进 3120 芯片内。

4) #pragma disable_servpin_pullup

服务引脚的上拉电阻通常是使能的，使用本编译指令可以使之无效。该编译指令只有在 I/O 初始化时才有效。对 LonBuilder 的 Neuron 仿真器，不要使用该指令。

5) #pragma disable_snvt_si

该指令默认生成自确认数据。若使用该编译指令将禁止生成自确认数据（SI)，这样可以释放部分程序存储空间。在源程序中，该指令只能出现一次。

6) #pragma eeprom_locked

该指令给应用程序提供一种机制，将 E^2PROM 的校验和加锁（设置一个标志）。由于 E^2PROM 的校验和包括应用及网络映像，不包括应用程序中使用的 E^2PROM 变量，所以 E^2PROM 变量不受保护。详见 set_eeprom_lock() 函数。

7) #pragma enable_io_pullups

通常引脚 IO4-IO7 的内部上拉电阻是非使能的，使用该指令内部引脚 IO4-IO7 的上拉电阻使能。该指令在 I/O 初始化时有效。

8) #pragma enable_multiple_baud

当使用多个比特速率不等的串行 I/O 设备时，该指令必须使用。如果需要，该指令必须在调用 I/O 函数(io_in()、io_out()) 之前出现。

9) #pragma enable_sd_nv_names

使用本编译指令，编译器在自确认数据文件（SI）产生时将网络变量名插入到自编文件（SD）中。在源文件中该指令只能出现一次。

10) #pragma explicit_addressing_off

11) #pragma explicit_addressing_on

这两个指令只在使用 MIP 程序时才使用。

12) #pragma fyi_off

13) #pragma fyi_on

这两个指令控制编译器打印信息性消息。信息性消息比警告消息的严重性要低得多，它可以指示程序中的某个问题，也可以指示代码能获得改善的位置。在编译启动时信息性消息的这个指令默认是关闭的。这两个指令在程序中可以多次使用，以便按需要开启或关闭信息性消息的打印动作。

14) #pragma hidden

该指令只用于标准包含文件<echelon.h>。

15) #pragma idempotent_duplicate_off

16) #pragma idempotent_duplicate_on

这两个指令用于控制应用缓存器中的幂等请求重试位。它们仅用于 MIP 应用，且默认是关。当编译时，必须使用上述指令中的一个，同时#pragma micro_interface 指令也要使用。

17) #pragma ignore_notused <symbol>

编译器通常会打印程序中已定义但并未使用的变量、函数以及 I/O 对象等的警告消息。本指令请求编译器忽略命名符号<symbol>的“ referenced ”标志。该指令可以多次使用以便禁止打印命名符号的警告消息。

18) #pragma micro_interface

该指令用于 MIP 应用。

19) #pragma app_buf_in_count <count>

20) #pragma app_buf_in_size <size>

21) #pragma app_buf_out_count <count>

22) #pragma app_buf_out_priority_count <count>

23) #pragma app_buf_out_size <size>

24) #pragma net_buf_in_count <count>

25) #pragma net_buf_in_size <size>

26) #pragma net_buf_out_count <count>

27) #pragma net_buf_out_priority_count <count>

28) #pragma net_buf_out_size <size>

29) #pragma receive_trans_count <num>

以上指令用于指定应用缓存器、网络缓存器的数量、大小以及接收事务数。 num、size 以及 count 的取值见表 1-1。

表 1-1 缓存器的容量及数量

指 令	允 许 的 值	默 认 值
App_buf_out_size	20,21,22,24,26,30,34,42,50,66,82,114,146,210,255	A
App_buf_out_count	1,2,3,5,7,11,15,23,31,47,63,95,127,191	E

续表

指　令	允 许 的 值	默 认 值
App_buf_out_priority_count	0,1,2,3,5,7,11,15,23,31,47,63,95,127,191	E
Net_buf_out_size	20,21,22,24,26,30,34,42,50,66,82,114,146,210,255	B
Net_buf_out_count	1,2,3,5,7,11,15,23,31,47,63,95,127,191	E
Net_buf_out_priority_count	0,1,2,3,5,7,11,15,23,31,47,63,95,127,191	E
Net_buf_in_size	20,21,22,24,26,30,34,42,50,66,82,114,146,210,255	66
Net_buf_in_count	1,2,3,5,7,11,15,23,31,47,63,95,127,191	2
App_buf_in_size	20,21,22,24,26,30,34,42,50,66,82,114,146,210,255	C
App_buf_in_count	1,2,3,5,7,11,15,23,31,47,63,95,127,191	2
Receive_trans_count	1...16	D

注：

A．　app_buf_out_size **默认值**

如果使用 msg_send()**发送消息：**

　　使用显式寻址　　A = 66

　　使用隐式寻址　　A = 50

如果是网络变量：

　　使用显式寻址　　A = max(34, 19 + sizeof(largest output NV))

　　使用隐式寻址　　A = max(20, 8 + sizeof(largest output NV))

B.　net_buf_out_size　**默认值**

如果使用 msg_send()**或** resp_send()**发送消息：**

　　B = 66

如果是网络变量：

　　B = max(42, 22 + sizeof(largest NV))

C.　app_buf_in_size　**默认值**

如果使用显式消息函数或事件：

　　使用显式寻址　　C = 66

　　使用隐式寻址　　C = 50

如果是网络变量：

　　使用显式寻址　　C = max(34, 19 + sizeof(largest output NV))

　　使用隐式寻址　　C = max(22, 8 + sizeof(largest output NV))

D.　receive_trans_count **默认值**

如果应用程序接收显式消息：

　　D = max(8, min (16,　# of non-config input NVs + 2))

如果接收网络变量：

D = min(16, # of non-config input NVs + 2)

E. app_buf_out_count, app_buf_out_priority_count,
net_buf_out_count, net_buf_out_priority_count 的默认值

对 3120E2、3150芯片： E = 2

对 3120E1、3120 芯片： E = 1

如果优先级缓存器数量为 0，所有的网络变量应定义为：

non-priority（nonconfig）

如果优先级缓存器中有一个数量不是 0，那么这两个都必须不为 0 并且两个发送事务缓存器自动在 RAM 中分配得到；如果没有优先级输出缓存器，那么只有一个发送事务缓存器，容量在芯片固件 ver4.0 和 ver6.0 版本是 28字节，更早的版本是 18字节长。

30) #pragma netvar_processing_off

31) #pragma netvar_processing_on

该指令用于 MIP 应用中。

32) #pragma no_hidden

该指令仅用于标准包含文件<echelon.h>。

33) #pragma num_addr_table_entries <num>

该指令设置地址表中的最大记录数为<num>。有效值范围是 0～15。地址表默认记录数是 15。很显然该指令可以帮助用户为地址表记录分配 E^2PROM 的空间。

34) #pragma num_alias_table_entries <num>

该指令控制由编译器分配的别名表记录数。别名表记录数在编译时必须指定，运行时不能改变。<num>有效值范围是 0～62。编译器默认值是“ 0”。同样该指令可以帮助用户为别名表分配 E^2PROM 的空间。

35) #pragma num_domain_entries <num>

该指令设置域表的记录数。<num>的有效值是 1 或 2，默认值是 2。同样该指令可以帮助用户为域表分配 E^2PROM 的空间。

36) #pragma ram_test_off

该指令禁止对片外 RAM 缓存器空间的测试以加速初始化。通常 Neuron 芯片固件在复位后或者加电后要做的第一件事就是检验诸如 CPU、RAM 以及定时器/计数器的基本功能，这样要花费大量的时间，尤其在较低的时钟速率情况下耗费的时间更长。关闭 RAM 缓存器的测试，可节省部分时间。尽管如此，所有 RAM 静态变量仍然赋“ 0 ”初值。

37) #pragma read_write_protect

该指令节点程序加读写保护，这样可避免在网上被复制或修改。一旦设置保护， 节点不允许再次被装载（对 3120 芯片）。不过对 3150 芯片可以使用 EEBLANK 程序实现程序的再次装载。如果因写保护引起的装载失败，将会显示出错消息。

38) #pragma relaxed_casting_off

39) #pragma relaxed_casting_on

该指令控制编译器是否将除去常数属性这一动作看成是一个错误或是一个警告。这一动作可以是显式的也可以是隐式的（就如同赋值或函数参数传递时的自动转换）。通常编译器将除去常数属性的转换看成是一个错误动作。 relaxed_casting_on 将之看成是一个警告。

这两个指令可按照需要在程序中多次使用。

40) #pragma scheduler_reset

该指令在非优先级 when 执行周期内，每个事件处理后，调度程序复位。

41) #pragma set_id_string "ssssssss"

该指令提供一种机制用于设置 8 字节长的程序标识符 ID。在应用映像中头 8 个字节即是程序的 ID。该 ID 串在服务引脚消息发送时也被插入该消息中，作为其中的一部分。该指令还是查询 ID 这个网管消息的响应内容。 ID 串可以是一字符串常量，最多 8 个字符。该指令不能与指令#pragma set_std_prog_id 同时使用。

42) #pragma set_netvar_count <nn>

该指令仅用于 MIP 应用中。

43) #pragma set_node_sd_string <C string const>

该指令指定并控制节点应用映像中自编文件（SD）的生成。总的来说，节点有 SD 串，且 SD 串默认是空串并且串的最大长度可以达到 1023 字节，其中包括 0 终止符。该指令显式地设置该 SD 串，但不允许有串联的字符串常量。源程序中该指令只能出现一次。对 LONMARK 节点，部分 SD 串用于指定节点中对象的类型。

44) #pragma set_std_prog_id hh:hh:hh:hh:hh:hh:hh:hh

该指令提供一种机制用于设置节点 8 字节长的标识符 ID。每字节用 16 进制数表示，除第一字节的值范围是 80～FF 外，其它字节的值都是 00～FF。该指令不能与指令#pragma set_id_string 同时使用。

表 1-2 给出 8 字节长的程序 ID 串的组成成员。

表 1-2 标准程序 ID 串的组成成员

成 员	比特数（比特）	类 型	数 据 源
格式	4	无符号	LONMARK 程序
制造商标识符 ID	20	无符号	LONMARK 程序
设备级	16	无符号	LONMARK 程序
设备子级	16	无符号	LONMARK 程序
模块号	8	无符号	制造商

各成员解释如下：

- 格式——4 比特定义程序 ID 的结构，格式 8、10、11、12、13、14、15 保留供具有互操作性的 LONMARK 节点使用，格式 8 为 LONMARK 标准程序 ID 格式，格式 9 可以在开发期间供网络管理器用来测试标准 ID 的译码。
- 制造商 ID——20 比特用于唯一标识生产具有互操作性 LONMARK 节点的生产商。
- 设备级——16 比特的 ID 标识设备级。
- 模块号——8 比特 ID 用于标识指定的产品模块。模块号由产品制造商提供并且对制造商来说在设备级以及设备子级内是唯一的。同样的硬件可以使用多个模块号，这当然取决于装载进硬件的程序。在程序 ID 内的模块号不一定要与制造商的模块号相一致。

45) #pragma snvt_si_eecode

编译器迫使链接器将自确认文件以及自编文件信息放置在 EECODE 空间,通常链接器将它们放置在 E^2PROM 以及 ROM 代码空间。两文件表放在 E^2PROM 中，可以利用网络管理消息写存储器来修改，网管消息在节点安装时修改节点的自编文件。对节点来说，这一点非常有用，节点可以被连接到多个不同类型的 I/O 设备上。注意，该指令只能用于 3150 芯片。

46) #pragma snvt_si_ramcode

编译器迫使链接器将自确认文件以及自编文件信息放置在 RAMCODE 空间。两文件表放在 RAMCODE 中，可以利用网络管理消息写存储器来修改。注意： RAMCODE 空间应该是外部的存储器空间，且是非易失的。同样该指令对 3150 芯片适用。

47) #pragma transaction_by_address_off

48) #pragma transaction_by_address_on

该指令显式地控制芯片固件使用的事务 ID 配置算法的版本。芯片固件有的版本（尤其最新版本）使用新版本的事务 ID 配置算法，该算法能够极好地防止消息的重复接收。对 3150 芯片、3120E1 以及 3120E2 芯片,固件 ver6.0 或更新版本都支持新老版本事务 ID 配置算法。对 3120 芯片，固件 ver4.0 或更新版本也支持新老版本的事务 ID 配置算法。如果可选的话，默认的事务 ID 配置算法是最新版本，除非节点是基于主机节点或者节点的应用程序使用显式寻址。

49) #pragma warnings_off

50) #pragma warnings_on

控制编译器显示打印警告消息。警告消息通常指出程序的问题或建议代码放置位置。默认警告消息在编译启动时即显示。该指令在程序中可以多次使用从而按用户的愿望开启或关闭消息的显示。

第二章

Neuron C 程序设计

LON 技术中每个节点都有它自己的调度程序、定时器以及逻辑 I/O 设备。Neuron C 中提供有预定的对象供编程人员用来实现对这些设备的访问。关于网络变量、显式消息以及 I/O 对象的应用程序设计，本书将放在专门的章节讲述。本章的目的旨在帮助读者了解 Neuron C 程序的基本结构以及简单的程序设计。

2.1 调 度 程 序

Neuron 芯片的调度程序负责 Neuron 芯片的任务调度。任务调度是由事件驱动的，如：当一个给定的条件判断为“ TRUE”，与该条件有关的代码体（任务）即执行。调度程序允许编程人员定义任务，用以作为某类事件发生的结果，如：

```
    when (timer_expires(led_timer))   // when 子句
 {
    io_out (io_led,OFF); //关闭 LED 任务
}
```

上述例子中，当已定义的应用定时器 led_timer 事件溢出，事件发生为“TRUE”, when 子句下面的代码体（任务）即执行关闭指定的 I/O 对象 io led 的任务。在任务执行后，时间溢出事件被清除。当 led timer 再次溢出，when 子句中事件为“ TRUE”，任务又将执行。

可见，Neuron C 程序的基本结构就是 when() 子句。

2.1.1 when 子句

下面给出的是 when 子句的语法定义：

```
/*****************************************************************************
[priority] [preempt_safe] when (event)
{
task
}
```

说明：

priority（优先级）　　可选项。若选择该项，调度程序每次运行都必须对有此选项的 when 子句进行判断。有关优先级 when 子句后面有专门叙述：

preempt_safe	可选项。如选用，即便应用处于占先（preemption）方式，调度程序仍然执行相关的 when 任务；
event（事件）	圆括号内的事件可以是单独的预定事件也可以是有效的 Neuron C 表达式（当然预定事件可以作为表达式的一部分）；
task （任务）	任务实际是 Neuron C 语句，它由一系列 Neuron C 说明以及语句组成，用花括号括上。可以说任务等同于无返回值的函数体，使用 return 语句可以中断任务的执行。

**********/

[示例] 同一个任务与多个 when 子句关联。

```
when (reset)
when (io_changes(io_switch))
when (!timer_expires)
when (flush_completes && (y ==5))
when (x ==3)
{
        // 打开 LED 并启动定时器
        ...
}
```

注：When 子句不能嵌套，如下面的程序是错误的：

```
when (io_changes(io_switch))
{
    when (x == 3) {
    // 不能嵌套
    }
}
```

1. When 子句的事件类型

when 子句的事件类型有两种：预定事件以及用户定义事件。预定事件包括输入引脚状态变化、网络变量修改、定时器溢出以及消息接收等；用户定义事件可以是任意有效的 Neuron C 表达式。这两者并没有太大的差别。因为预定事件所需代码空间较小，所以可能的话尽量使用预定事件。

(1) 预定事件

在 Neuron C 中给出的预定义事件有：

flush_completes
offline
online
wink
timer_expires
reset

（以上见本章内容）

io_changes
io_in_ready
io_out_ready
io_update_occurs （以上见第五章内容）
msg_arrives
msg_completes
msg_fails
msg_succeeds
resp_arrives （以上见第四章内容）
nv_update_occurs
nv_update_completes
nv_update_fails
nv_update_succeeds （以上见第三章内容）

(2) 预定事件处理

与网络有关的预定事件的处理通常使用两个各自独立的队列。一个队列主要用于与输入的网络消息有关的事件服务，如：

nv_update_occurs
msg_arrives
online
offline
wink

另一个队列则用于其余与网络消息有关的事件服务，属于完成事件以及响应，如：

nv_update_completes
nv_update_succeeds
nv_update_fails
msg_completes
msg_succeeds
msg_fails
resp_arrives

除 resp_arrives ，大多数网络事件只有当 Neuron C 编译器确定应用程序中有检查这个事件的语句才会再次排队。online、offline 以及 wink 事件总是参加排队，但是一旦调度程序未找到对应这些预定事件的 when 子句，这些事件将从队列中剔除。

一旦事件位于队列之首，它将停留在队列之首，直到应用程序来处理。这样，如果应用程序不去处理该事件，该事件将阻塞队列。队列阻塞必然会使应用无法连续正常地处理事件，节点将无法响应任何后来的应用消息及网络管理消息，而实际情况是 nv_update_occurs 以及 msg_arrives 事件可以在任意时间发生，所以队列阻塞将会带来更为严重的后果。相比之下，完成事件以及响应仅仅是作为应用输出网络消息的结果而发生的。Neuron C 编译器将根据事件在程序中的存在决定应用对事件的处理。

(3) 预定事件处理方式

预定事件处理方式有两种，一种是异步事件处理方式，它是预定事件在 when()子句中检查，即预定事件构成 when 子句的表达式；一种则是直接事件处理方式，它是预定事件在任务中被检查，即预定事件构成 if、while 或 for 的表达式。

异步事件处理是典型的事件处理方法，该方法能节省程序代码。异步事件处理以及直接事件处理可以同时在一个程序中出现，不过在异步事件处理转成直接事件处理之前，程序应调用 flush_wait() 函数(见后面小节的叙述)。

[示例]

```
msg_tag motor;
#define MOTOR_ON 0
when (x==3)
{
    flush_wait();                          // 发送消息
    msg_out.tag = motor;
    msg_out.code = MOTOR_ON;
    msg_send();
        while (!msg_succeeds(motor)) {     // 检查完成状态
        post_events();
        if (msg_fails(motor))
            node_reset();
    }
}
```

2. when 子句的调度

调度程序对一组 when 子句的判断过程是一个循环的过程（round-robin）：调度程序将判断每个 when 子句，如果某个 when 子句是“TRUE”，执行相关的任务；如果该子句是“FALSE”，调度程序将挪到下一个 when 子句判断该子句的值；到最后一个 when 子句判断结束，调度程序返回到顶部再对这一组 when 子句重复刚才的过程，如：

```
when (nv_update_occurs)
//    {任务}
when (nv_update_fails)
//    {任务}
when (io_changes)
//    {任务}
when (timer_expires)
//    {任务}
```

为便于解释，下面用字母来命名各事件及相关的任务。

```
when (A)
{A}
```

when (B)

{B}

when (C)

{C}

when (D)

{D}

图 2-1 显示了该例子任务执行的顺序。

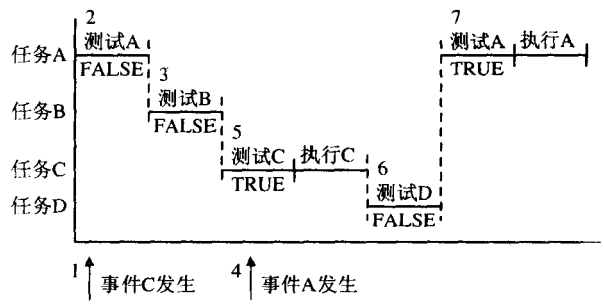

图 2-1 调度程序的调度过程

图 2-1 说明如下：

1——假使一开始只有 C 事件是 TRUE。

2——调度程序从 A 开始，由于 A 是 FALSE，任务 A 不执行。

3——调度程序移到 B。因为 B 是 FALSE，所以任务 B 不执行。

4——这个时候 A 事件变为 TRUE。

5——调度程序依然往下移至 C，因为 C 是 TRUE，任务 C 执行。

6——调度程序继续移至 D，因为 D 是 FALSE，任务 D 不执行。

7——调度程序回到 A，因为 A 是 TRUE，任务 A 执行。

3. 优先级 when 子句的调度

如果 when 子句选用 Priority（优先级）关键字，相比无优先级的 when 子句，调度程序对具有优先级的 when 子句的判断次数要频繁得多。对具有优先级的 when 子句，调度程序每次运行都将首先对之判断。如果一个优先级的 when 子句判断是 TRUE，对应的任务执行，调度程序又返回到有优先级的 when 子句。如果优先级的 when 子句判断是 FALSE，那么调度程序才会转向非优先级的 when 子句判断。对有优先级的 when 子句组的调度过程如图 2-2 所示。要提醒注意的是：要仔细考虑优先级 when 子句的使用，因为太多的优先级 when 子句可能使非优先级 when 子句无法执行。如果一个优先级 when 子句绝大多数时间都是 TRUE，那么它将独占处理器的时间。

可见，通过对 when 子句的介绍，Neuron 芯片固件调度程序的基本功能一目了然。其实调度程序概括起来可以用图 2-3 的调度环来示意。优先级 when 子句按照每次调度程序运行指定的顺序执行。如果任一个优先级 when 子句判断为 TRUE，它的任务即执行并且调度程序再开始。如果没有优先级 when 子句判断为 TRUE，那么一个非优先级 when 子句被判断，并采用循环方式（round-robin），见图 2-1。如果 when 子句判断是 TRUE，它的任务执