Neuron® C Programmer's Guide

Revision 6

日本語版

ECHELON®

Corporation

078-0002-02F

スマート・トランシーバ、Neuron チップ、Serial LonTalk®アダプタ、およびその他の OEM 製品は、人体の健康または安全に危険を及ぼすか、財産に損害を与える危険性のある機器やシステムへ使用することを目的には設計されていません。そのような用途で上記製品を使用することについて、Echelon 社は一切の責任および義務を免れるものとします。

Echelon 社以外のベンダーが製造し、本書で参照されている部品は、説明の目的のみに参照されているものであり、Echelon 社ではテストされていない場合があります。各アプリケーションにこれらの部品が適合するかどうかを判断するのはお客様の責任となります。

ECHELON 社は、商品性、品質への満足、特定の目的に対する適合性、非侵害性について、その他類似の事項に関するすべての暗黙的な保証および条件を含め、明示・暗示に関わらず、法定上、またはその他のいかなるお客様との通信においても、一切の表明、保証、および条件提示をしないものとします。。

文書番号 29300

Printed in the United States of America.

Echelon Corporation
550 Meridian Avenue
San Jose, CA. USA
95126

www.echelon.com

はじめに

本書は、Neuron® C 言語を使ったプログラム開発の手引書です。Neuron C は、Neuron チップおよびスマート・トランシーバ用に設計された、ANSI C に基づくプログラミング言語です。ANSI C の拡張機能としてネットワーク通信、入出力、およびイベント処理の機能を提供することにより、LONWORKS®アプリケーションの開発を強力にサポートしています。本書では Neuron C のキー・コンセプトをプログラム例や図表を使って説明しています。また、LONWORKS アプリケーションの設計や導入に関する一般的な方法論についても述べています。

本書を読まれる前に

『Neuron C Programmer's Guide』は、LONWORKS®アプリケーションを開発するプログラマを対象にしています。本書は読者が C 言語の経験者であることを前提にしています。

ANSI C については、下記の書籍を参照してください。

- American National Standard X3.159-1989, Programming Language C, D.F. Prosser, American National Standards Institute, 1989.
- Standard C: Programmer's Quick Reference, P.J.Plauger and Jim Brodie, Microsoft Press, 1989.
- C:　A Reference Manual, Samuel P. Harbison and Guy L. Steele, Jr., 4th edition, Prentice-Hall, Inc., 1994.
- C Programming Language, Brian W. Kernighan and Dennis M. Ritchie, 2nd edition, Prentice-Hall, Inc., 1988.

本書の内容

『Neuron C Programmer's Guide』には、次のような内容が含まれています。

- LONWORKS アプリケーション開発の一般的な手順
- Neuron C のプログラミングに関するキー・コンセプトのコード例を使った説明

関連マニュアル

『NodeBuilder® User's Guide』では、NodeBuilder 開発ツールを使用した LONWORKS アプリケーションの開発に必要なすべてのタスクについて説明しています。NodeBuilder ツールのユーザ・インターフェースと機能の詳細については、このガイドを参照してください。

『LonBuilder® User's Guide』では、LonBuilder 開発ツールを使用した LONWORKS アプリケーションの開発に必要なすべてのタスクについて説明しています。LonBuilder ツールのユーザ・インターフェースの詳細については、このガイドを参照してください。

『Neuron C Reference Guide』では、Neuron C 言語を使用してプログラムを作成するためのリファレンス情報を提供しています。

『NodeBuilder Errors Guide』では、NodeBuilder ソフトウエアに関連するすべての警告とエラー・メッセージについて説明しています。

『LonMaker® User's Guide』では、LonMaker 統合ツールを使用した、LONWORKS ネットワークのインストール、操作、および保守に必要なすべてのタスクについて説明しています。LonMaker ツールのユーザ・インターフェースと機能の詳細については、このガイドを参照してください。

『Gizmo 4 User's Guide』では、Gizmo 4 のハードウエアとソフトウエアについて説明しています。Gizmo 4 のハードウエアとソフトウエア・インターフェースの詳細については、このガイドを参照してください。

『FT 3120® and the FT 3150® Smart Transceivers Databook』では、Echelon のスマート・トランシーバのハードウエアとアーキテクチャについて説明しています（このマニュアルは本書内では『Smart Transceivers Databook』と呼ばれています）。その他の Neuron チップの情報は、各デバイスのメーカーから入手してください。

シンタックスに関する表記規則

書体	使用対象	例
太字	キーワード リテラル文字	**network** **{**
italic（斜体）	抽象的な要素	*identifier*
ブラケット（角カッコ）	省略可能フィールド	[bind-info]
縦棒	2 要素間の選択	input | output

例えばネットワーク変数宣言のシンタックスは、次のようになります。

network input | output [*netvar modifier*] [*class*] *type* [*bind-info*] *identifier*

ブラケットと縦棒以外の記号（引用符、かっこ、セミコロンなど）は表記の通りに使用する必要があります。

プログラム例は以下のように Courier フォントで表示されます。

```
#include <mem.h>

unsigned array1[40], array2[40];

// See if array1 matches array2
if (memcmp(array1, array2, 40) != 0) {
      // The contents of the two areas do not match
}
```

目次

1

概要

この章では、Neuron C バージョン 2 プログラミング言語について紹介します。ここでは言語の基本的な概念について説明し、LONWORKS プラットフォームと Neuron C プログラミング言語を使用して、相互運用が可能なデバイスやシステムを構築する方法について述べます。また、イベント駆動スケジュール、ネットワーク変数、構成プロパティ、機能ブロック（機能プロファイルの実装）といった Neuron C のキー・コンセプトも紹介します。

この章では、Neuron C の型、記憶クラス、データ・オブジェクト、Neuron C 言語と ANSI C 言語の違いなどの、基本事項を紹介します。

Neuron C とは

Neuron C バージョン 2 は、Neuron チップとスマート・トランシーバ用に設計された、ANSI C に基づくプログラミング言語です。ANSI C の拡張機能としてネットワーク通信、入出力、およびイベント処理の機能を提供することにより、LONWORKS アプリケーションの開発を強力にサポートします。Neuron C の新機能には次のものが含まれます。

- *functional block*（機能ブロック）と *network variable*（ネットワーク変数）に基づいた新しいネットワーク通信モデルによって、類似または異種のデバイス間の接続を簡素化し、データ共有を促進します。
- 機能ブロックと *configuration property*（構成プロパティ）に基づいた新しいネットワーク構成モデルによって、相互運用が可能なネットワーク構成ツールを提供します。
- 標準 *resource file*（リソース・ファイル）とユーザ定義の *resource file* に基づいた新しい型のモデルによって、複数のメーカーのデバイスを簡単に統合し、相互運用可能なデバイスの選択肢が拡大します。
- Neuron チップとスマート・トランシーバの高度な入出力能力をサポートするさまざまな *I/O object*（I/O オブジェクト）を備えています。
- 新しい **when** 文に基づいた強力な *event-driven programming*（イベント駆動プログラミング）拡張機能によって、ネットワーク、入出力、およびタイマーのイベントを簡単に処理します。

Neuron C は、分散制御アプリケーションの開発に必要な要件を満たすため、ANSI C の拡張機能を豊富に揃えています。経験豊かな C 言語プログラマにとって、Neuron C は、親しみのある ANSI C のパラダイムが自然に拡張された言語と捉えることができるはずです。また、型チェックの機能も組み込まれており、プログラマは LONWORKS 分散アプリケーション用のコードを効率良く生成することができます。

Neuron C では、自立処理系として要求されていない ANSI C の機能についてはサポートしていません。例えば、Neuron C にはいくつかの標準 C ライブラリが含まれていません。Neuron C と ANSI C のその他の相違については、本章で後述します。

Neuron C 特有の要素

Neuron C は、すべての基本的な ANSI C の型を実装しており、また必要に応じて型を変換することも可能です。Neuron C は ANSI C のデータ構造体に加え、独自のデータ要素も備えています。「ネットワーク変数」は、Neuron C と LONWORKS アプリケーションの基礎を成すものです。ネットワーク変数とは、言語とシステム・ファームウエアをサポートするデータ構造体で、C プログラムにおける変数と似たような機能を提供します。さらに、LONWORKS ネットワークと、ネットワーク上の複数のデバイス間の通信を可能にするプロパティも備えています。ネットワーク変数は、LONWORKS デバイスの「デバイス・インターフェース」の一部を構成しています。

「構成プロパティ」は、デバイス・インターフェースを構成するもう 1 つの Neuron C のデータ構造体です。構成プロパティによって、LonMaker ツールや、デバイス専用に作成するプラグインなどのネットワーク管理ツールを使用して、デバイスの動作をカスタマイズできます。

Neuron C はさらに、デバイスのネットワーク変数や構成プロパティを「機能ブロック」にまとめる手段を備えています。各機能ブロックはネットワーク変数と構成プロパティから成り、全体で 1 つのタスクを実行することができます。これらのネットワーク変数と構成プロパティは、「機能ブロック・メンバ」と呼ばれます。

ネットワーク変数、構成プロパティ、および機能ブロックは、それぞれ *resource file*「リソース・ファイル」に含まれている型定義によって定義されます。ネットワーク変数と構成プロパティは、「ネットワーク変数型（NVT）」と「構成プロパティ型（CPT）」によって定義されます。機能ブロックは、「機能プロファイル」（「機能プロファイル・テンプレート」とも呼ばれる）によって定義されます。

Neuron C のネットワーク変数、構成プロパティ、および機能ブロックは、標準の「相互運用可能な型」を使用できます。標準のデータ型を使用することで、LONWORKS ネットワーク上の異種のデバイスの相互運用がスムーズに行われるようになります。標準の構成プロパティは標準構成プロパティ型（SCPT、スキピットと発音）と呼ばれます。標準のネットワーク変数は標準ネットワーク変数型（SNVT、スニベットと発音）と呼ばれます。標準の機能ブロックは標準機能プロファイル（SFP）と呼ばれます。要件を満たす標準の型またはプロファイルが見つからない場合のために、ユーザ・ネットワーク変数型（UNVT）、ユーザ構成プロパティ型（UCPT）、およびユーザ機能プロファイル（UFP）も完全にサポートしています。

Neuron C は、Neuron システム・ファームウエアが提供する環境で実行するように設計されています。このファームウエアは、Neuron C 言語のランタイム環境として、「イベント駆動スケジュール・システム」を提供しています。

また、Neuron C ではネットワーク変数モデルの他に、より低いレベルの「メッセージ・サービス」も用意しています。ただし、ネットワーク変数モデルは情報を交換するための方法として標準化されているという利点があるのに対し、メッセージ・サービスは LONWORKS ファイル転送プロトコルによる使用以外、標準化されていません。標準型とユーザ型の両方のネットワーク変数を使用することで、複数のベンダの複数のデバイス間の相互運用性が向上します。一方、低レベルのメッセージ・サービスを使用すると、ファイル転送プロトコルに加え、独自のソリューションを使用できるようになります。

Neuron C のもう 1 つのデータ・オブジェクトに「タイマー」があります。タイマーは、変数と同じように宣言して使用することができます。タイマーの期限が切れると、システム・ファームウエアは自動的にタイマーのイベントを管理し、それらのイベントをプログラムに通知します。

Neuron C には数多くの組み込み「I/O オブジェクト」が用意されています。これらの I/O オブジェクトは、Neuron チップとスマート・トランシーバの I/O ハードウエアのために標準化された、I/O「デバイス・ドライバ」です。各 I/O オブジェクトはイベント駆動プログラミング・モデルに適合します。各 I/O オブジェクトと対話するために、関数呼び出しインターフェースが提供されています。

この章の残りの部分では、Neuron C のさまざまな要素についてもう少し詳しく説明し、次の章以降では、これらの要素をさらに詳しく、例を挙げて説明します。

Neuron C の整数定数

負の定数は正の定数の単項マイナス演算として扱われます。例えば、-128 は **signed short** ではなく、**signed long** になります。

10 進整数の定数の型は、デフォルトでは次のとおりです。

`0 .. 127`	**signed short**
`128 .. 32767`	**signed long**
`32768 .. 65535`	**unsigned long**

u、**U**、**l**、**L** を数字の後につけると、デフォルトの型を変更できます。

`0L`	**signed long**
`127U`	**unsigned short**
`127UL`	**unsigned long**
`256U`	**unsigned long**

16 進の定数の型は、デフォルトでは次のとおりです。これらも上で述べた **u**、**U**、**l**、**L** をつけてデフォルトの型を変更できます。

`0x0 .. 0x7F`	**signed short**
`0x80 .. 0xFF`	**unsigned short**
`0x100 .. 0x7FFF`	**signed long**
`0x8000 .. 0xFFFF`	**unsigned long**

8 進の定数の型は、デフォルトでは次のとおりです。これらも上で述べた **u**、**U**、**l**、**L** をつけてデフォルトの型を変更できます。

`0 .. 0177`	**signed short**
`0200 .. 0377`	**unsigned short**
`0400 .. 077777`	**signed long**
`0100000 .. 0177777`	**unsigned long**

2 進の定数の型は、デフォルトでは次のとおりです。これらも上で述べた **u**、**U**、**l**、**L** をつけてデフォルトの型を変更できます。

`0b0 .. 0b01111111`	**signed short**
`0b10000000 .. 0b11111111`	**unsigned short**
`0b0000000100000000 .. 0b0111111111111111`	**signed long**
`0b1000000000000000 .. 0b1111111111111111`	**unsigned long**

Neuron C の変数

以下では、変数宣言のさまざまな要素について簡単に説明します。データ型は変数が表すデータの性質を決定します。記憶クラスは変数が格納される場所、変更可能であるかどうか（可能である場合には変更の頻度）、およびデ

ータを変更するためのデバイス・インターフェースは存在するかどうかを決定します。

Neuron C の変数型

Neuron C は、次の C 変数型をサポートしています。ブラケット内に示されているキーワードはオプションです。省略した場合は、ANSI C 言語の規則通りに扱われます。

[signed] long int	16 ビット長
unsigned long int	16 ビット長
signed char	8 ビット長
[unsigned] char	8 ビット長
[signed] [short] int	8 ビット長
unsigned [short] int	8 ビット長
enum	8 ビット長　**(int** 型)

Neuron C では、定義済みの列挙型が用意されています。以下はその一例です。

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

Neuron C には、ANSI C 言語の変数に似たルック・アンド・フィールを持つ、定義済みオブジェクトも用意されています。Neuron C タイマーと I/O オブジェクトは、その一例です。I/O オブジェクトの詳細は本書の第 2 章に、タイマー・オブジェクトの詳細は『Neuron C Reference Guide』の「Timers」の章に記載されています。

拡張演算ライブラリでは、IEEE 754 準拠の **float_type** と、符号付き 32 ビット整数データ用の **s32_type** が定義されています。これらの型については『Neuron C Reference Guide』の「Functions」の章で詳しく説明しています。

Neuron C の記憶クラス

クラスが指定されていない宣言がファイル「スコープ」にあれば、そのデータあるいは関数はグローバルです。ファイル「スコープ」とは、関数にもタスクにも含まれていない Neuron C プログラム部分のことです。グローバル・データは、**static** キーワードで宣言されているすべてのデータを含めて、それが宣言された時点以降の、プログラムの実行全体にわたって有効です。変数の前方リファレンスを提供するには、**extern** リファレンスを使用して宣言します。関数の前方リファレンスを提供するには、関数プロトタイプを宣言する必要があります。

Neuron チップまたはスマート・トランシーバの電源を入れたときやリセットしたとき、RAM 上のグローバル・データが存在する場合にはそれが初期化され、存在しない場合はゼロになります。ただし **eeprom** または **config** クラスを使用して宣言されている変数、および **config_prop** または **cp_family** キーワードを使用して宣言されている構成プロパティは、アプリケーション・イメージが最初にロードされたときにだけ初期化されます。

Neuron C でサポートしている ANSI C の記憶クラスと型修飾子は、次のとおりです。

auto ローカル・スコープの変数を宣言します。通常、関数の中で使用します。これはローカル・スコープ内でのデフォルトの記憶クラスであり、キーワードは通常指定しません。**auto** スコープの変数で **static** でないものは、ローカル・スコープに入る時点で初期化されません。また、プログラムの実行がこのスコープを出てしまえば、変数の値も消失します。

const アプリケーション・プログラムによって変更できない値を宣言します。CP ファミリまたは構成ネットワーク変数の宣言と併用されたときは、この値が Neuron C コンパイラが生成する自己記録(SD)データに格納されます。

extern 他のモジュール、ライブラリ、またはシステム・イメージの中で定義されるデータ項目または関数を宣言します。

static リンク時に他のモジュールからは利用できないデータ項目または関数を宣言します。また、データ項目が関数や **when** 節のタスクに対してローカルであれば、データの値は呼び出しと呼び出しの間にも保存されますが、他の関数がコンパイル時にこの値を使用することはできません。

ANSI C の記憶クラスの他に、Neuron C では次の記憶クラスとクラス修飾子を用意しています。

config 入力ネットワーク変数宣言にのみ使用します。**config** が指定されたネットワーク変数は、アプリケーションの設定に使用され、**const eeprom** と同じ働きをします。このようなネットワーク変数はアプリケーション・イメージが最初にロードされたときにだけ初期化されます。**config** クラスは旧式のクラスで、古いアプリケーションをサポートするためにのみ提供されているため、Neuron C コンパイラによって **config** クラスのネットワーク変数の自己記録データが生成されることはありません。新しいアプリケーションは、本書の「構成プロパティ」の章で説明している構成ネットワーク変数のシンタックスを使用する必要があります。

network ネットワーク変数の宣言を開始します。詳細については、第 3 章「ネットワーク変数を使ったデバイス間通信」を参照してください。

system Neuron ファームウエア関数ライブラリにアクセスするためにのみ、Neuron C で使用されます。このキーワードはデータまたは関数の宣言には使用しないでください。

uninit **eeprom** キーワードと組み合わせて使用し（下記を参照）、ネットワークを介したプログラムのロード時

または再ロード時に EEPROM 変数が初期化または変更されないようにします。

次の Neuron C キーワードは、アプリケーション・コードまたはデータの一部を特定のメモリ・セクションに配置するときに使用します。

- **eeprom**
- **far**
- **offchip**　（外部メモリを持つ Neuron チップとスマート・トランシーバのみ）
- **onchip**
- **ram**　（外部メモリを持つ Neuron チップとスマート・トランシーバのみ）

Neuron 3150 チップと FT 3150 スマート・トランシーバは、アドレス領域のほとんどがオフチップにマップされているため、上記のキーワードが役立ちます。メモリとこれらのキーワードの詳しい使用方法については、第 8 章「Neuron チップ・メモリの使用」の「メモリの管理」を参照してください。

変数の初期化

変数の初期化のタイミングは、記憶クラスによって異なります。ネットワーク変数以外の **const** 変数は、必ず初期化されなければなりません。**const** 変数は、アプリケーション・イメージが Neuron チップまたはスマート・トランシーバにロードされたときに初期化されます。**const ram** 変数は、非揮発性であるオフチップ RAM に記憶されることに注意してください。このため、**eeprom** 変数や **config** 変数の定義に **uninit** クラス修飾子を使用しない限り、これらの変数もロード時に初期化されることになります。

Neuron C は自動変数の宣言と同時に初期化する機能を持たないため、自動変数を、**const** として宣言することはできません。

グローバル RAM 変数は、リセット時（デバイスをリセットしたとき、あるいは電源を入れたとき）に初期化されます。特に指定しなければ、**static** 変数も含めてすべてのグローバル RAM 変数は、このときにゼロに初期化されます。グローバル RAM 変数のゼロ初期化はファームウエア機能であるため、初期化のためのコードを書く必要がなく、コード・スペースを節約できます。

I/O オブジェクトと入力ネットワーク変数（**eeprom**、**config**、**config_prop**、**const** ネットワーク変数を除く）、およびタイマーも、リセット時に初期化されます。ネットワーク変数とタイマーのデフォルト値はゼロです。

ローカル変数（**static** 変数以外）は自動的には初期化されず、また、変数の値は、プログラムの実行がローカル・スコープを出た後は保存されません。

Neuron C の宣言

Neuron C では、次の ANSI C の宣言をサポートしています。

宣言	例
単純なデータ項目	**int a, b, c;**
データ型	**typedef unsigned long ULONG;**
列挙型	**enum hue {RED, GREEN, BLUE};**
ポインタ	**char *p;**
関数	**int f(int a, int b);**

- 配列 **int a[4];**
- 構造体と共用体

```
struct s {
    int field1;
    unsigned field2 : 3;
    unsigned field3 : 4;
};
```

これに加えて、Neuron C バージョン 2 では以下の宣言をサポートしています。

	宣言	**例**
•	I/O オブジェクト	**IO_0 output oneshot relay_trigger;** （第 2 章 参照）
•	タイマー	**mtimer led_on_timer;** （第 2 章 参照）
•	ネットワーク変数	**network input SNVT_temp temperature;** （第 3 章 参照）
•	構成プロパティ	**SCPTdefOutput cp_family defaultOut;** （第 4 章 参照）
•	機能ブロック	**fblock SFPTnodeObject { … } myNode;** （第 5 章 参照）
•	メッセージ・タグ	**msg_tag command;** （第 6 章 参照）

ネットワーク変数、SNVT および UNVT

ネットワーク変数とはデバイス上のオブジェクトのことを指し、別のデバイス上のネットワーク変数に接続することができます。ネットワークから見ると、ネットワーク変数はデバイスの入出力の定義として扱われ、分散アプリケーションでのデータ共有を可能にしています。あるプログラムが「出力」ネットワーク変数に何かを書き込むと（**polled** 修飾子を使用して宣言されている出力ネットワーク変数を除く）、ネットワークを経由して、その出力ネットワーク変数に接続している「入力」ネットワーク変数を持つ全てのデバイスにその値が伝達されます。出力ネットワーク変数が現在いずれのネットワーク変数接続のメンバにもなっていない場合は、トランザクションもエラーも発生しません。ネットワーク変数の伝達には LONWORKS メッセージを使用していますが、メッセージの送信は暗示的に行われます。アプリケーション・プログラムには、ネットワーク変数の更新を送信、受信、管理、再試行、認証、または確認するための明示的な命令は必要ありません。Neuron C アプリケーションは出力ネットワーク変数に書き込むことで最新の値を提供し、入力ネットワーク変数を読み込むことで最新のデータを取得します。

コード例

```
network input SNVT_temp nviTemperature;
network output SNVT_temp nvoTemperature;

void f(void)
{
   nvoTemperature = 2 * nviTemperature;
}
```

ネットワーク変数を使用すると、個々のデバイスを独立に定義できるため、さまざまな LONWORKS アプリケーションに簡単にデバイスを接続、再接続できます。すなわち、ネットワーク変数によって、分散システムの開発やインストール手順が大幅に簡略化されます。ネットワーク変数の詳細については、第 3 章「ネットワーク変数を使ったデバイス間通信」と『Neuron C Reference Guide』を参照してください。

ネットワーク変数は、デバイス間の通信に使用するインターフェース設計を向上させ、デバイスの相互運用性を高めます。相互運用性が高まることによって、デバイスのアプリケーションをネットワーク構成とは独立に管理することができ、別のネットワークへのインストールも簡単になります。あるデバイスをネットワークにインストールしたとき、お互いのネットワーク変数のデータ型（**int** や **long** など）さえ合っていれば、ネットワーク内の他のデバイスとの論理的接続が可能です。相互運用性をさらに向上させるため、LONWORKS プラットフォームには、デバイスの標準機能インターフェースを定義する標準機能プロファイルに加え、標準のデータ・エンコーディング、スケーリング、および摂氏、ボルト、メートルなどの単位を定義する標準ネットワーク変数型（SNVT）が用意されています。標準機能プロファイルは、さまざまな機能要件に対応しています。SNVT 定義は実質的にあらゆる物理的な数量に対応しており、特定の業界や一般的なアプリケーションにも利用できるよう、さらに抽象的な定義も用意されています。

独自のユーザ機能プロファイルやユーザ・ネットワーク変数型（UNVT）を作成することもできます。目的のデバイスを他のメーカーのデバイスと併用するには、リソース・ファイルを定義し、カスタムの型やカスタム・プロファイルを作成します。NodeBuilder ツールに含まれている NodeBuilder リソース・エディタでは、既存のリソースを表示し、独自のリソースを定義する作業を 1 つのインターフェースで行うことができます。

構成プロパティ

構成プロパティとは、ネットワーク変数と同様に、デバイスのインターフェースの一部を構成しているデータ項目で、ネットワーク管理ツールを使用して変更できます。構成プロパティは、標準化されたネットワーク変数をデバイス構成データとして使用することで、異なるデバイスのインストールと構成を支援し、相互運用性を高めます。ネットワーク変数と同様に、構成プロパティにも便利なインターフェースが用意されています。各構成プロパティ型は、型に基づいた構成プロパティのデータ・エンコーディング、スケーリング、単位、デフォルト値、範囲、および動作を指定するリソース・ファイル内で定義します。標準の構成プロパティ型（SCPT）はさまざまなものが用意されていますが、独自のユーザ構成プロパティ型（UCPT）を作成することもできます。UCPT を使用するには、NodeBuilder リソース・エディタを使用してリソース・ファイルを作成し、その中で型を定義します。

機能ブロックと機能プロファイル

LONWORKS デバイスの「デバイス・インターフェース」は、機能ブロック、ネットワーク変数、および構成プロパティで構成されています。「機能ブロック」はネットワーク変数と構成プロパティの集まりで、1 つのタスクを実行するために共に使用されます。これらのネットワーク変数と構成プロパティは、「機能ブロック・メンバ」と呼ばれます。

機能ブロックは、「機能プロファイル」によって定義されます。機能プロファイルは、共通の機能を動作の単位ごとに記述したものです。各機能プロファイルは必須のネットワーク変数とオプションのネットワーク変数、および必須の構成プロパティとオプションの構成プロパティを定義します。各機能ブロックは機能プロファイルのインスタンスを実装します。機能ブロックは、機能プロファイルが定義した必須のネットワーク変数と構成プロパティをすべて実装する必要があり、機能プロファイルが定義したオプションのネットワーク変数と構成プロパティは実装してもしなくても構いません。機能ブロックは、機能プロファイルが定義したものではないネットワーク変数と構成プロパティを実装することもできます。これらは「固有の実装」によるネットワーク変数および構成プロパティと呼ばれます。

機能プロファイルは「リソース・ファイル」で定義します。標準機能プロファイルを使用することも、NodeBuilder リソース・エディタを使用して、独自のリソース・ファイルに独自の機能プロファイルを定義することもできます。リソース・ファイルで定義される機能プロファイルは、「機能プロファイル・テンプレート（FPT）」とも呼ばれます。

LONMARK®相互運用性協会は、開発者がデバイスを認証するための手順を提供しています。LONMARK の相互運用可能なデバイスは、『LONMARK Layer 1 - 6 Interoperability Guidelines』に指定されているすべての LonTalk®プロトコル層 1～6 の要件と、『LONMARK Application Layer Interoperability Guidelines』に記述されているアプリケーション設計のすべての要素に準拠しています。

入会およびデバイスの認証の詳細については、LONMARK Interoperability Association（www.lonmark.org）にお問い合わせください。

デバイスをインストールする時にネットワーク管理ツールが使用するインターフェース情報は、デバイス・インターフェース識別データとしてデバイス内に自動的に組み込むことができます。このデータは「自己識別（SI）」データおよび「自己記録（SD）」データと呼ばれます。Neuron C コンパイラは、宣言された機能ブロック、ネットワーク変数、構成プロパティ、および提供されたリソース・ファイルに基づいてこのデータを生成します。独自の文書を SD データに追加して、デバイスとそのインターフェースの情報ををさらに記録することもできます。

#pragma enable_sd_nv_names 指令を使用すると、SD データにネットワーク変数名を含めることができます。また、各ネットワーク変数の SD データには、メッセージ数 x 10/秒単位の概算レートとメッセージ数 x 10/秒単位の最大概算レートを含めることもできます。概算レートと最大概算レートの値は **bind_info** 機能によって提供されます（この機能については、第 3 章「ネットワーク変数を使ったデバイス間通信」と『Neuron C Reference Guide』を参照してください）。

Neuron C コンパイラが作成したデバイスのアプリケーション・イメージには、**#pragma disable_snvt_si** 指令を使用していない限り、SD 情報が含まれます（詳細については、『Neuron C Reference Guide』の「Compiler Directives」の章を参照してください）。

データ駆動プロトコルとコマンド駆動プロトコル

ネットワーク変数は、デバイス間のデータおよびステート情報のやりとりに使用します。これによって、コマンド入力を主体としたシステムとは異なる通信モデルが提供されます。コマンド入力を主体としたメッセージ・システムを設計する場合、アプリケーションを管理、更新、保守するための多数の

コマンドが必要になります。また、それぞれのデバイスが個々のコマンドを知っている必要もあります。そうなると、コマンド・テーブルとアプリケーション・コードの大きさは、際限なく広がっていきます。

ネットワーク変数の場合は、コマンドやメッセージのアクション定義がメッセージの中にあるのではなく、アプリケーション・プログラムの中に存在するため、各アプリケーション・プログラムにはその機能を実行するのに必要な知識だけが必要になります。ネットワーク・インテグレータは新しい種類のデバイスを必要に応じて追加し、ネットワーク内の既存のデバイスに接続することで、デバイスの元の設計者が意図しなかった新しいアプリケーションを実行できるようになります。

イベント駆動とポーリング型スケジューリング

Neuron C 言語は主にイベント駆動型スケジューリングを自然で簡単なものにする目的で設計されていますが、集中制御可能なポーリング型アプリケーションを作成することも可能です。ポーリングの詳細については、第 3 章「ネットワーク変数を使ったデバイス間通信」を参照してください。

低レベルのメッセージ通信

機能ブロックとネットワーク変数通信モデルに加え、Neuron C はアプリケーション・メッセージもサポートしています。アプリケーション・メッセージは、ネットワーク変数の代替として使うか、または、デバイスの独自のインターフェースを実装するときに、ネットワーク変数を補う形で使用します。また、LONWORKS のファイル転送プロトコルとしても使用します。アプリケーション・メッセージについては、第 6 章「アプリケーション・メッセージを使ったデバイス間通信」を参照してください。

I/O デバイス

１つの Neuron チップまたはスマート・トランシーバには、I/O デバイスを 1 つ以上接続できます。接続できる I/O デバイスには、温度計、位置センサー、バルブ、スイッチ、LED ディスプレイなどがあります。また、Neuron チップとスマート・トランシーバに他のマイクロ・プロセッサが接続していることもあります。Neuron ファームウエアは、これらのデバイスと Neuron C アプリケーションとのインターフェースになる数多くの I/O オブジェクトを実装しています。I/O オブジェクトの詳細については第 2 章「シングル・デバイスでの機能」と『Neuron C Reference Guide』で説明されています。

Neuron C と ANSI C の相違点

Neuron C は ANSI C 言語の標準に従っていますが、「American National Standards Institute committee X3-J11」で定義されている標準 C の規格に完全に準拠しているわけではありません。

以下に Neuron C と ANSI C の相違点の概要を記述します。

- Neuron C では、C のシンタックスや演算子を用いた浮動小数点演算をサポートしていません。 ただし、IEEE 754 準拠の浮動小数点データを使用できるように、浮動小数点ライブラリが提供されています。

- ANSI C では、**short int** 型が 16 ビットまたはそれ以上、**long int** 型が 32 ビットまたはそれ以上と定義されています。Neuron C では、**short int** 型を 8 ビット、**long int** 型を 16 ビットと定義しています。また、Neuron C では **int** 型のデフォルトが **short int** 型になっています。32 ビットデータの処理には、32 ビット符号付き整数ライブラリを使用してください。
- Neuron C は、**register** および **volatile** 記憶クラスをサポートしていません。これらの記憶クラスは指定しても無視されます。
- Neuron C は、**auto** 変数の宣言文での初期化子を実装していません。
- Neuron C は、実引数パラメータまたは関数の戻り値に構造体や共用体を使用することをサポートしていません。
- ネットワーク変数構造体には、ポインタを指定できません。構成プロパティ構造体にも、ポインタは指定できません。
- タイマー、メッセージ・タグ、I/O オブジェクトへのポインタはサポートしていません。
- ネットワーク変数、構成プロパティ、および EEPROM 変数へのポインタは、定数へのポインタとして扱われます。つまり、ポインタが参照している変数の値を読むことはできますが、変更はできません。ただし、制限はありますが、ポインタを使ったメモリ内容の変更ができる場合もあります。**eeprom_memcpy()**関数の詳細については、第 8 章「メモリ管理」と『Neuron C Reference Guide』の「Functions」の章の説明を参照してください。また、『Neuron C Reference Guide』の「Compiler Directives」の章の**#pragma relaxed_casting_on** コンパイラ指令の説明も参照してください。
- マクロ引数は、そのマクロが展開された後で再スキャンされます。このため、ネストしたマクロ展開の中に#や##のマクロ演算子があると、ANSI C 標準で定義されているような展開結果とならないことがあります。
- ネットワーク変数名とメッセージ・タグ名は、最大 16 文字です。機能ブロック名は、最大 16 文字です。ただし、**external_name** 機能を使用して宣言している場合、関数ブロックの外部名は最大 16 文字で、内部名は最大 64 文字になります。
- Neuron C には、**memcpy()**や **memset()**などの ANSI C ライブラリ関数がいくつか含まれています。また文字列およびバイト演算ライブラリが提供されているので、<**string.h**>インクルード・ファイルで定義されている ANSI C のサブセットを使用できます。ファイル入出力、メモリ割り当てといった ANSI C ライブラリ関数は、Neuron C には含まれていません。ライブラリ関数の詳細については、『Neuron C Reference Guide』の関数一覧を参照してください。
- Neuron C には、<**stddef.h**>、<**stdlib.h**>、<**limits.h**>の 3 つの ANSI インクルード・ファイルがあります。
- Neuron C では、関数定義の前にその関数を呼び出す場合、関数プロトタイプ機能を使用する必要があります（第 2 章 参照）。
- Neuron C は関数プロトタイプまたは関数定義内での省略記号（...）の使用はサポートしていません。
- Neuron C では、ANSI C にはない予約語とシンタックスを追加しています。シンタックスの概要と予約語の一覧については、『Neuron C Reference Guide』を参照してください。
- Neuron C では、8 進と 16 進に加えて 2 進の定数をサポートしています。2 進定数は、0b< 2 進数> という形式で指定します。例えば、**0b1101** は 10 進での 13 を意味します。

- Neuron C では、従来の /* */ スタイルの他に C++で使用されている//スタイルのコメントをサポートしています。//スタイルでは、スラッシュ 2 つ（//）がコメントの始まりになります。このコメント・スタイルでは、区切り記号がなくても行末がコメントの終了になります。

```
C code   /* An ANSI C and NEURON C comment */

C code   // A line-style NEURON C comment
```

- **main()**構造は使用しません。その代わり、Neuron C プログラムの実行可能オブジェクトの構成要素には、関数の他にも **when** 文があります。プログラム実行のスレッドは、常に **when** 文で始まります。これについては、第 2 章「シングル・デバイスでの機能」を参照してください。
- Neuron C は、複数ソース・ファイルでの分割コンパイルをサポートしていません。ただし、**#include** 指令はサポートしています。
- ANSI C プリプロセッサ指令の**#if**、**#elif**、**#line** はサポートしていません。**#ifdef**、**#ifndef**、**#else**、**#endif** は使用できます。

Neuron C 言語の実装特性

C プログラミング言語の規格を管理している American National Standard（米国規格協会）では、セクション 3、付録 F の中で、C の各実装の動作は、「このセクションに一覧表示されている各分野において実装がどのように振舞うかを文書化しなければならない。次の言語の各要素は「実装定義」である」と述べています。

この規格では、「実装定義」という用語は、「実装の特性に依存し、各実装が文書化され、正しいプログラムと正しいデータのための動作」であることが定義されています。したがってこれらの項目は ANSI 規格ではなく、言語の定義の問題であり、すべて各実装者に任されています。これらは移植性の問題と捉えることもできます。

以下の各見出しは、ANSI C 言語規格の付録 F の条項と、その付録の該当するセクションを参照しています。それぞれの回答は、Echelon Corporation が提供する Neuron C バージョン 2 コンパイラの、本書の印刷時における最新の実装を表しています。

Translation（変換）（F.3.2）

Q: 診断はどのように識別されますか? （セクション 2.1.1.3）

A: 各 Neuron C の診断は、少なくとも 2 行で構成され、標準の出力ファイルへ出力されます。これらのキーワードは、FYI（情報）、警告、エラー、または致命的エラーのいずれかの診断を示します。1 行目の残りの部分には、診断が適用されるソースまたはインクルード・ファイルの完全なパス名に続いて、行番号、かっこ内の列番号が記載されています。

2 行目（または 2 行目以降）には、診断内容が含まれています。各診断メッセージ行はタブ 1 つ分インデントされています。

FYI と警告の診断によって、コンパイラの正常な変換が阻止されることはありません。「警告」の診断はすべて調べて修正する必要がありますが、通常

これらはプログラミングの問題または好ましくないプログラミングを示すものです。

エラーの診断では、コンパイラの正常な変換が阻止されます。これらはほかのエラーを隠してしまう可能性があるため、コンパイラによって 1 つのコンパイル・パスにあるエラーがすべて検出されるとは限りません。

致命的エラーの診断では、エラーの発生時点でコンパイラによる変換が中止されます。このような診断は、リソースの問題（メモリ不足、ディスクの領域不足など）か、コンパイラ自体の内部チェックによって発生します。***TRAP *n****の形式（*n* は 10 進数）の診断が発生した場合は、Echelon のカスタマ・サポートに報告してください。

Environment（環境）（F.3.2）

Q: **main** の引数にはどのような意味がありますか? （セクション 2.1.2.2.1）

A: Neuron C では **main** プロシージャにはどのような特別な意味も付加していません。**main** という名前は他の正当な識別子と同じように使用できます。

Q: 対話デバイスは何で構成されていますか? （セクション 2.1.2.3）

A: Neuron C では対話デバイスは定義していません。

Identifiers（識別子）(F.3.3)

Q: 外部リンケージを持たない識別子の初期有効文字数（32 以上）はいくつですか? （セクション 3.1.2）

A: 外部リンケージを持たない識別子は最高 256 文字まで拡張できます。文字はすべて有効です。

Q: 外部リンケージを持つ識別子の初期有効文字数（7 以上）はいくつですか? （セクション 3.1.2）

A: Neuron C の外部リンケージには traditional external（標準外部）リンケージと network external（ネットワーク外部）リンケージの 2 つの形式があります。標準外部リンケージは、**extern**、**static**、ファイル・スコープ変数、プロシージャ名で構成されています。これらの名前はプログラムを結合してロード・イメージを構成するときに、Neuron C のリンカによって使用されます。**extern** または **static** の記憶クラスを使用して宣言された名前、またはファイル有効範囲で宣言された名前は 63 文字に制限されます。コンパイラは文字を名前に追加して一意の名前を作成する場合があります。その場合には、外部識別子の長さがさらに制限されることがありますが、名前が 50 文字以下に制限されることはありません。名前が極端に長い場合、コンパイラは警告の診断を生成し、その名前を許容される最大の長さに切り詰めます。このため、標準外部名は 50 文字以内で指定することをお勧めします。

外部リンケージの 2 番目の形式であるネットワーク外部リンケージは、ネットワークとネットワーク管理ツールが使用する名前で構成されています。これらの名前は、ネットワーク変数名、メッセージ・タグ名、非標準型のネットワーク変数を定義するために使用する **typedef** の名前を含んでいます。ネットワーク外部名が 16 文字を超えると、コンパイラはエラーの診断を生成します。機能ブロック名は、**fblock** 宣言に **external_name** または

external_resource_name のオプションがない場合にネットワーク外部名とみなされます。オプションが提供されている場合、内部機能ブロック名は 64 文字以下で指定できます。

Q: 外部リンケージを持つ識別子では大文字と小文字が区別されますか? （セクション 3.1.2）

A: はい。外部リンケージを持つ識別子では、前述のどちらの形式の外部リンケージでも、大文字と小文字が区別されます。

Characters（文字）（F.3.4）

Q: 規格が明示的に定義しているもの以外の、ソース文字セットと実行文字セットのメンバには何がありますか? （セクション 2.2.1）

A: Neuron C 文字セットは、ソース文字セットと実行文字セットに基本的な ASCII 文字エンコーディングを使用しています。Neuron C のソース文字セットは、規格内で明示的に定義されている文字セットです。ASCII 改行文字（16 進の 0D）と ASCII バックスペース文字（16 進の 08）はどちらも空白文字として受け入れられます。行末文字は ASCII の改行（16 進の 0A）文字になります。また、Neuron コンパイラは文字定数や文字列リテラルにおいて残りの基本的な ASCII 印刷可能文字**@**（アット記号）および`（アクセント記号）を受け入れます。

Neuron C コンパイラは、ASCII EOT 文字（16 進の 04）をファイル終了マーカとして解釈します。同様に、MS-DOS のテキスト・ファイル終了文字である Ctrl-Z（16 進の 1A）は、ファイル終了マーカと解釈されます。ただし、Neuron C コンパイラによってこれらの文字が要求されることはありません。

実行文字セットには基本の ASCII（0..127 の文字値）が採用されていますが、Neuron C で作成されたプログラムでは、0..127 の範囲外の文字値の解釈を自由に使用できます。

Q: マルチバイト文字のエンコーディングに使用されるシフト状態はどれですか? （セクション 2.2.1.2）

A: Neuron C はマルチバイト文字をサポートしていません。2 文字以上を含む文字定数はエラーになります。

Q: 実行文字セット内の文字のビット数はいくつですか? ワイド文字、つまり **wchar_t** の型のサイズはいつくですか? （セクション 2.2.4.2.1）

A: 実行文字セットは 8 ビット表現を使用しています。Neuron C のコンパイラはワイド文字をサポートしていませんが、ワイド文字の型 **wchar_t** は **unsigned long** として定義されています（Neuron C は **unsigned long** を 16 ビットとして定義していることに注意してください）。

Q: ソース文字セット（文字定数と文字列リテラル内）のメンバから実行文字セットへのマップはどのように行われますか? （セクション 3.1.3.4）

A: ソース文字セットから実行文字セットのマップは恒等関係になります。

Q: 基本的な実行文字セットまたはワイド文字定数用の拡張文字セットで表現されていない文字またはエスケープ・シーケンスを含む整数文字定数の値は何ですか? （セクション 3.1.3.4）

A: 整数文字定数には基本的な実行文字セット内の文字だけを含めることができます。エスケープ・シーケンスを使用すると、文字定数は 0～255 の範囲、または **signed char** を使用する場合は、-128（¥x80）～127（¥x7F）の範囲で指定できます。

Q: 複数のマルチバイト文字を含む整数文字定数の値は何ですか? マルチバイト文字をワイド文字定数用の対応するワイド文字（コード）に変換するために使用する現在のロケールはどれですか? （セクション 3.1.3.4）

A: Neuron C コンパイラはマルチバイト文字を実装していません。

Q: 「プレーン」の **char** の値の範囲は **signed char** または **unsigned char** と同じですか? （セクション 3.2.1.1）

A: 「プレーン」の **char** は **unsigned char** と同じです。

Integers（整数）（F.3.5）

Q: 整数の表現方法と値の範囲はどのようになっていますか? マルチユニット整数表現のビットの順位はどのようになっていますか? 符号なし整数のエンコーディング方法は何ですか? 符号付き整数のエンコーディング方法は何ですか? （セクション 3.1.2.5）

A: **int** はデフォルトでは **short int** となります。これは Neuron C では 8 ビットを表します。8 ビットのバイトは Neuron チップの基本的な記憶単位です。**long int** は 16 ビット、つまり 2 バイトの整数表現です。<**limits.h**>インクルード・ファイルには、さまざまな整数型の範囲を定義しています。値は以下のとおりです。

```
  -128 ..   127      signed short
     0 ..   255      unsigned short
-32768 .. 32767      signed long
     0 .. 65535      unsigned long
```

符号なしの整数値はすべて 2 進表現を使用します。符号付き整数は 2 の補数の 2 進表現を使用します。マルチユニット表現である **long int** は、最上位のバイトがアドレスの最下位になるように格納されます。

Q: 整数をより短い符号付き整数に変換するとどうなりますか? 符号なし整数を同じ長さの符号付きの整数に変換した場合に、符号付き整数が符号なし整数の値を表現できないときはどうなりますか? （セクション 3.2.1.2）

A: **long** から **short** に変換すると、変換する値によってはデータが失われる場合があります。この変換は、**long** 整数の最上位のバイトを破棄することによって行われるためです。例えば、値が 513（16 進の 0201）の **long** 整数を **signed short** に変換する場合、**long** 整数の最上位のバイトが破棄されるため、結果の値は 1 になります。

同じ長さの **unsigned** 整数を **signed** 整数に変換すると、結果が負数になる場合があります。例えば、値が 255（16 進の FF）の **unsigned short** 整数を **signed short** 整数に変換すると、2 の補数を使用して解釈されるため、結果が-1 になります。

Neuron C コンパイラは、「暗示的」な変換操作によってデータが失われる可能性があるときに、診断メッセージを生成します。型変換操作などによる「明

示的」な変換では、診断メッセージは生成されません。以下のコード例では、x への代入によって、診断の「警告」メッセージが生成されますが、y への代入ではメッセージは生成されません。

```
int x, y;
x = 285;          // Data is lost, x is assigned 29.
                  // Warning is produced.

y = (int)285;     // Data is lost, y is assigned 29.
                  // No warning is produced.
```

Q: 符号付き整数のビット単位演算の結果はどうなりますか? （セクション 3.3）

A: 符号付き整数のビット単位演算は、オペランドの値が **unsigned** であるかのように実行され、結果は **signed** として解釈されます。したがって、(-2)|1 の結果は-1 になります。

Q: 整数の除算の余りの符号はどうなりますか? （セクション 3.3.5）

A: 整数の除算（つまり **op1 % op2**）の余りの符号は常に **op1** の符号と同じになります。

Q: 負数の値を持つ符号付き整数型の右シフトの結果はどうなりますか? （セクション 3.3.7）

A: 負数の値を持つ符号付き整数型は、右シフトされ、2 進の整数型は左からシフトインされます。したがって、**int x** と **long int x** では、**(x>>1)**は常に**(x/2)** に等しくなります。

Floating Point（浮動小数点）（F.3.6）

Neuron C は C のシンタックスまたは演算子を使用した浮動小数点演算をサポートしていません。浮動小数点ライブラリは Neuron C に含まれており、関数呼び出しとして浮動小数点演算の一部を実装しています。浮動小数点ライブラリは IEEE 754 に準拠したデータに対して使用できます。

Arrays and Pointers（配列とポインタ）（F.3.7）

Q: 配列の最大サイズを格納するために必要な整数型（**sizeof** 演算子の **size_t** 型）は何ですか? （セクション 3.3.3.4, 4.1.1）

A: 配列の最大サイズ（32,767 の要素）には **unsigned long** を使用する必要があります。

Q: ポインタを整数に型変換するか、その反対の操作を行うと、結果はどうなりますか? ある型のポインタを別の型のポインタに型変換すると、結果はどうなりますか? （セクション 3.3.4）

A: ポインタと **unsigned long** 整数の 2 進表現は同じであるため、ポインタから整数への型変換の結果は **unsigned long** から **int** への型変換の結果と同じになります。整数からポインタへの型変換では、整数から **unsigned long** への型変換と同じ結果になります。

ポインタの表現はすべて相互変換できるため、1 つの型のポインタから別の型のポインタへの型変換では変換は実行されず、予期する結果が得られるとは限りません。

Q: 同じ配列内の異なる要素への 2 つのポインタ間の差違 **ptrdiff_t** を格納するにはどの型の整数が必要ですか? （セクション 3.3.6, 4.1.1）

A: 2 つのポインタの減算結果は[**signed**] **long** になります。

Registers（レジスタ）（F.3.8）

Q: **register** 記憶クラスの指定子を使用してレジスタに実際にオブジェクトを配置するとどうなりますか? （セクション 3.5.1）

A: Neuron チップはスタック・ベースのアーキテクチャを使用しています。このアーキテクチャには汎用レジスタがないため、コンパイラは **register** 記憶クラスを無視します。また、**register** クラスが使用されると警告の診断がコンパイラによって生成されます。

Structures, Unions, Enumerations, and Bit-Fields（構造体、共用体、列挙型、およびビットフィールド）（F.3.9）

Q: 別の型のメンバを使用して共用体オブジェクトのメンバにアクセスするとどうなりますか? （セクション 3.3.2.3）

A: 異なる型のメンバは、共用体内の同一のオフセットにおいて重なり合います。このため、**long** または **unsigned long** としてポインタにアクセスするか、またはその反対の操作を行うと、その結果はメンバを型変換したときと同じになります。同様に、同じリストに含まれている別の型のメンバとして **int**、**char**、または **short** にアクセスすると、その結果はメンバを型変換したときと同じになります。**long** データ型またはポインタ・データ型を **short** としてアクセスすると、その結果は最上位のバイトの値になります。**short** データ型に **long** としてアクセスすると、未使用のバイト（**long** の最下位バイト）が読み込まれるか、変更され、**long** の最上位バイトが **short** にマップされます。

Q: 構造体のメンバのパディングまたは整列はどのように行われますか? （セクション 3.5.2.1）

A: Neuron チップは **byte** で整列されるため、Neuron C の構造体のメンバ間ではパディングは不要で、実行もされません。

Q: 「プレーン」の **int** ビット・フィールドは **signed int** ビット・フィールドまたは **unsigned int** ビット・フィールドのどちらとして扱われますか? （セクション 3.5.2.1）

A: 「プレーン」の **int** ビット・フィールドは **signed int** ビット・フィールドとして扱われます。符号が必要なとき以外は、**unsigned** ビット・フィールドの使用をお勧めします。unsigned ビット・フィールドの方がランタイム時の効率が良く、コード領域も節約できるためです。

Q: ビット・フィールドの割り当て順はどうなっていますか? （セクション 3.5.2.1）

A: ビット・フィールドはバイト内で上位ビットから下位ビットへと割り当てられていきます。

Q: ビット・フィールドが記憶単位の境界を越えることはできますか? （セクション 3.5.2.1）

A: いいえ。ビット・フィールドがバイトの境界を越えることはできません。このため、最大のビット・フィールドは 8 ビットになります。

Q: 列挙型の値を表すための整数型はどれですか? （セクション 3.5.2.2）

A: 列挙型の値を表すには整数型 **int** が使用されます。このため、列挙型の有効な値の範囲は-128 ... 127 になります。

Qualifiers（修飾子）（F.3.10）

Q: **volatile** の修飾型を持つオブジェクトにはどのようにアクセスしますか? （セクション 3.5.5.3）

A: Neuron C は volatile 修飾型をサポートしていません。**volatile** 修飾型を使用すると、コンパイラによって警告の診断が生成されます。

Declarators（宣言子）（F.3.11）

Q: 算術型、構造体型、または共用体型を修飾できる宣言子は最大いくつですか? （セクション 3.5.4）

A: 型を修飾する宣言子の最大数に上限はありません。制限は、コンパイラが使用できるヒープ・メモリとスタック領域の量に基づいてランタイム時に決定されます。

Statements（文）（F.3.12）

Q: **switch** 文における **case** 値の最大数はいくつですか? （セクション 3.6.4.2）

A: Neuron C の **switch** 文は、スイッチ値の **int** 式のみを受け入れます。**switch** 文内の **case** ラベルは重複することができないため、可能な選択肢の数は 256 になります。Neuron C は 1 つの **switch** 文において、すべて異なる 256 の **case** 値を受け入れることができます。

Preprocessing Directives（指令の事前処理）（F.3.13）

Q: 条件組み込みを制御する定数式内の 1 文字定数の値は、実行文字セット内にある同じ文字定数の値に一致しますか? この文字定数は負の値を持つことができますか? （セクション 3.8.1）

A: どちらの質問の答えも「はい」です。

Q: インクルード可能なソース・ファイルはどのように検索しますか? （セクション 3.8.2）

A: 通常のインクルード命令は、引用符で囲まれた形式になります。この「システム」インクルード・ファイルにアクセスするには、ブラケットで囲む形式を使用します。

コード例

```
#include <stddef.h>
```

引用符で囲んだ形式:

```
#include "[drive:][pathname¥]filename.ext"
```

2 番目の形式では、ファイル名が絶対名またはドライブに相対した名前である場合、コンパイラによってその名前が使用されます。それ以外の場合は、作業ディレクトリがまず検索されます（相対パス名が提供されている場合には、それが使用されます）。次に、［LonBuilder Project Configuration］ウィンドウまたは［NodeBuilder Device Templates Properties］ダイアログと［Project Properties］ダイアログの［**Include Directories**］で指定されている各ディレクトリが検索されます。

NodeBuilder プロジェクト・マネージャで作業しているとき、または NodeBuilder プロジェクト作成ユーティリティを経由してコマンド・ラインから作業しているとき、「現在の作業ディレクトリ」は Neuron C のメイン・ソース・ファイルを含むフォルダになります。

ブラケットで囲んだ形式:

```
#include <filename.ext>
```

上の形式では、標準のファイル・ディレクトリ内の **include** サブディレクトリからシステム・インクルード・ファイル（<**limits.h**>や<**stddef.h**>など）が検索されます（ブラケットで囲んだ形式で指定されたインクルード・ファイルの検索は、［LonBuilder Project Configuration］ウィンドウまたは NodeBuilder の[Project Properties]ダイアログの［**Include Directories**］に指定されているディレクトリには影響を受けません）。LonBuilder ツールでは、「標準のファイル・ディレクトリ」は、ソフトウエアがインストールされているディレクトリになります。LonBuilder ツールのデフォルトのシステム・インクルード・ディレクトリは **C:¥lb¥include** です。NodeBuilder ツールでは、標準のファイル・ディレクトリは LONWORKS NeuronC ディレクトリになります。NodeBuilder ツールのデフォルトのシステム・インクルード・ディレクトリは **C:¥LonWorks¥NeuronC¥include** です。

Q: インクルード可能なソース・ファイルの引用符で囲まれた名前は、どのようなものがサポートされていますか? （セクション 3.8.2）

A: **#include** 指令内の引用符で囲まれた名前には、Windows オペレーティング・システムで有効な任意の名前を、絶対パス名、ドライブに相対したパス名、または相対パス名を使用して（存在する場合）、指定できます。パス名は現在の作業ディレクトリに相対させるか、インクルード・ファイルの検索パスにある任意のディレクトリに相対させることができます。

NodeBuilder プロジェクト・マネージャで作業しているとき、または NodeBuilder プロジェクト作成ユーティリティを経由してコマンド・ラインから作業しているとき、現在の作業ディレクトリは Neuron C のメイン・ソース・ファイルを含むフォルダになります。

Q: **#include** 指令内ではソース・ファイルの文字シーケンスはどのようにマップされますか? （セクション 3.8.2）

A: ソース･ファイルの文字シーケンスは大文字でも小文字でも構いません。有効なファイル名の文字をどれでも使用できます、大文字と小文字は区別されません。

Q: 認識されている各**#pragma** 指令はどのように動作しますか? （セクション 3.8.6）

A: **#pragma** 指令については、『Neuron C Reference Guide』の「Compiler Directives」の章を参照してください。

Q: 変換の日付と時刻がそれぞれ使用不可の場合の**__DATE__**と**__TIME__**はどのように定義されますか? （セクション 3.8.8）

A: Neuron C では**__DATE__**マクロと**__TIME__**マクロはサポートしていません。

Library Functions（ライブラリ関数）（F.3.14）

Q: **NULL** マクロが展開される **NULL** ポインタ定数はどれですか? （セクション 4.1.5）

A: **NULL** ポインタ定数は、**<stddef.h>**ファイルでは 0 として定義されています。

通常、Neuron C は「自立処理」を行います。つまり、標準の C ライブラリはすべて実装されるわけではありません。ただし、**strcpy( )**や **memcmp( )**などの文字列関数やメモリ関数を含む一部の標準の C 関数は利用できます。サポートされている関数の詳細については、『Neuron C Reference Guide』の「Functions」の章を参照してください。

2

シングル・デバイスでの機能

本章では、Neuron C のイベント・スケジューラと I/O オブジェクトについて説明します。ここでは、*predefined event*（定義済みイベント）と *user-defined event*（ユーザ定義イベント）の概念を紹介しています。本章に出てくるコード例は、イベント、I/O オブジェクト、タイマー・オブジェクト、入出力関数の使い方を説明しています。

各 Neuron C アプリケーションに定義できるオブジェクトには、本章で説明する *timer*（タイマー）や *input/output object*（I/O オブジェクト）の他にも、第 3 章で説明する *network variable*（ネットワーク変数）、第 4 章で説明する *configuration property*（構成プロパティ）、第 5 章で説明する *functional block*（機能ブロック）、第 6 章で説明する *application message*（アプリケーション・メッセージ）があります。

はじめに

本章では、まずシングル・デバイスに注目して、Neuron チップまたはスマート・トランシーバのプログラミングについて解説します。Neuron チップとスマート・トランシーバには、「Neuron ファームウエア」と呼ばれる標準のファームウエアと、スケジューラ、タイマー、I/O デバイスのドライバやインターフェースを実装するハードウエアサポート機能があります。Neuron C 言語には、ファームウエア機能にアクセスするための定義済みオブジェクトが含まれています。これらのオブジェクトについてここで簡単に説明し、この後のセクションでより詳しく説明します。

- Neuron ファームウエアの「イベント・スケジューラ」は、アプリケーション・プログラムが実行するタスクをスケジューリングします。本章では、Neuron C 言語を使用してイベントやタスクを定義する方法と、スケジューラがどのようにそれらのイベントを評価するかについて説明します。また、優先イベントの定義方法についても説明します。
- Neuron C 言語には、ミリ秒単位と秒単位の 2 種類の「タイマー」オブジェクトがあり、これらのタイマーを使用してタスクのスケジューリングを行います。詳細については、「タイマー」のセクションで説明します。
- Neuron C では、ANSI C の拡張機能として、多くの「I/O オブジェクト」の宣言をサポートしています。これらの I/O オブジェクト、および関連する入出力関数とイベントについては、「入出力」のセクションで説明します。

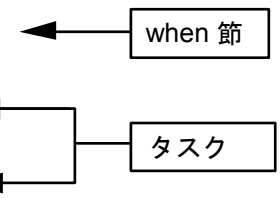
スケジューラ

アプリケーション・プログラムのスケジューリングはイベント駆動です。つまり、ある特定の条件が TRUE になったとき、その条件に結び付いているコード（「タスク」と呼びます）が実行されます。例えば、入力ピンの状態の変化、ネットワーク変数の新しい値の受信、タイマーの時間切れといった一定のイベントの結果として、特定のタスクが起動します。また、あるタスクを「優先タスク」として指定し、優先的にサービスを受けるように設定することもできます(本章の「優先 **when** 節」のセクションを参照してください)。

When 節

イベントの定義には **when** 節を使用します。**when** 節の中の式が TRUE と評価されたとき、**when** 節の後にあるコード（タスク）が実行されます。タスクの実行が終了すると、イベント処理が完了します。1 つのタスクを複数の **when** 節に結び付けることもできます。簡単な **when** 節とそれに伴うタスクを以下に示します。**when** 節とそれに関連するタスクの組み合わせは、総称して「**when** タスク」または「**when** 文」と呼ばれます。

```
when (timer_expires(led timer))
{
      // Turn off the LED
      io_out(io_led, OFF);
}
```



上の例では、**led_timer** アプリケーション・タイマー（この例では定義を省略）の時間が切れたときに、**when** 節に続く本体のコード（タスク）が実行されます。タスクは、**io_led** という名前の I/O オブジェクト（これもプログラム中のどこかで定義されている）をオフに設定して終了します。このタスクが実行されると、**timer_expires** イベントはクリアされます。そして、LED タイマーが再び時間切れになって **when** 節がもう一度 TRUE に評価されるまでは、このタスクが実行されることはありません。

次の各例は、タスクとイベントを使用するさまざまな方法を示したものです。タスクとイベントの詳細については、第 7 章「追加機能」と図 7.1 を参照してください。

```
when (reset)
when (io_changes(io_switch))
when (!timer_expires)
when (flush_completes && (y == 5))
when (x == 3)
{
      // Turn on the LED and start the timer
      . . .
}
```

when 節を入れ子にすることはできません。例えば、次のような入れ子の **when** 節は正しくありません。

```
when (io_changes(io_switch))
{
      when (x == 3) {   // Can't nest!
      ...
      }
}
```

if 文を使ってイベントをテストしても、同様の結果が得られます。

```
when (io_changes(io_switch))
{
      if (x == 3) {
      ...
      }
}
```

when 文

when 文（**when** 節あるいは **when** 節とタスク）のシンタックス：

when-clause
[when-clause ...]
task

when-clause のシンタックスは以下のとおりです。

[priority] [preempt_safe] when (*event***)**

priority スケジューラが起動するたびに、この **when** 節を強制的に評価します。本章の「優先 **when** 節」のセクションを参照してください。

preempt_safe アプリケーションが先取りモードであっても、関連する **when** タスクをスケジューラが実行できるようにします。第 6 章「アプリケーション・メッセージ

を使ったデバイス間通信」にある先取りモードの説明を参照してください。

event
この式には、定義済みイベント（次のセクションを参照してください）または正しい Neuron C の式（定義済みイベントが含まれていてもよい）を指定します。定義済みイベントも、式と同様に丸カッコで囲んで指定します。同じタスクに対して複数の **when** 節があっても構いません。

task
Neuron C の複文です。Neuron C の関数定義と同じようにブレースで囲み、その中に Neuron C の宣言や文を並べた構成にします。タスクは、**void** 型の（値を戻さない）関数の本体と同じ形式にします。タスクの実行を終了するために **return** 文を使うことはできますが、特に指定する必要はありません。

when 節に指定するイベントについて

when 節に指定するイベントには、大きく分けて「定義済みイベント」と「ユーザ定義イベント」の 2 種類があります。*predefined event*（定義済みイベント）の指定には、コンパイラに組み込まれているキーワードを使用します。定義済みイベントには、入力ピンの状態変化、ネットワーク変数の更新、タイマーの時間切れ、メッセージの受信などがあります。*user-defined event*（ユーザ定義イベント）は、Neuron C の式を使って定義します。

ユーザ定義イベントと定義済みイベントを特に区別して使用する必要はありません。ただし、定義済みイベントの方がコード量が少なくて済みますから、できるだけこちらを使うようにしてください。

when 節には、正しい C の式であればどのような式も指定できますが、１つだけ制約があります。**offline**、**online**、**wink** 定義済みイベントは、単独で使用してください。これらのイベントと他の式とを組み合わせて使用することはできません。その他の定義済みイベントは、任意の式と組み合わせることができます。これは、**when** 節だけの制限です。

コード例

```
when (msg_arrives)                      // O.K.
when (msg_arrives && flag == TRUE)      // O.K.
when (online)                           // O.K.
when (online && flag == TRUE)           // Not permitted.
```

定義済みイベント

前に紹介した **timer_expires** イベントは、定義済みイベントの 1 つです。次の表は、その他の定義済みイベントのキーワードを示したものです。

定義済みイベント	説明している章
flush_completes	第 7 章
io_changes	本章
io_in_ready	本章
io_out_ready	本章
io_update_occurs	本章
msg_arrives	第 6 章
msg_completes	第 6 章
msg_fails	第 6 章
msg_succeeds	第 6 章
nv_update_occurs	第 3 章
nv_update_completes	第 3 章
nv_update_fails	第 3 章
nv_update_succeeds	第 3 章
offline	第 7 章
online	第 7 章
reset	本章
resp_arrives	第 6 章
timer_expires	本章
wink	第 7 章

イベントの範囲を狭める修飾子が、I/O イベントやネットワーク変数イベントなどの定義済みイベントの後に続く場合があります。修飾子が任意であるか、与えられていない場合は、指定された型のすべてのイベントが受け入れられます。

定義済みイベントは、**if** 文、**while** 文、**for** 文の制御式の中に他の式やイベントと組み合わせて使用することもできます。この方式を「直接イベント処理」といいます。以下は、直接イベント処理の例です。

```
mtimer t;
when (event)
{
   . . .
   if (timer_expires(t)) {
       io_out(io_led, OFF);
   }
   . . .
}
```

定義済みイベントの組み込みキーワードもキーワードを使った式（例：**timer_expires(t)**）も他の式と同様に扱われ、標準の C で使用できる式の組み合わせはすべて Neuron C のプログラムでも使用できます。

ただし、**io_changes** イベント式には特別な使用上の注意があります。**io_changes** イベントだけに指定できる **to** と **by** という修飾子キーワードがありますが、これらのキーワードは一般的な式演算子の演算優先順位に従います。**to** と **by** 演算子は互いに同じ優先順位で（ただし互いに排他的）、比較などの関係演算子よりも優先順位が高く、シフトや数値演算子よりは優先順位が低くなります。

io_changes イベント式の分析方法の例を示します。

io_changes (*device***) by a + b**
↓
io_changes (*device***) by (a + b)**

io_changes (*device***) by a < b**
↓
(io_changes (*device***) by a) < b**

他の C の演算子を使っているときと同様、丸カッコを使って明示的に演算の順序を指定することができます。演算の順序に自信がなければ、丸カッコを使ってコードを明確にするようにしてください。余分なカッコを使用しても、コンパイル時や生成されるコードに悪影響を及ぼすことはありません。

Neuron C コンパイラは、**when** 節の中に定義済みイベントのキーワードがあると、コードを最適化するような特別な処理を行います。ただし、**when** 節の式の一部としてイベント・キーワードが使用されているときには、イベント・テーブルの最適化は行われません。次の 3 つのコード例では、最初の例だけイベント・テーブルによって最適化され、2 番目と 3 番目では最適化は行われません。

when (timer_expires) {}

when (! timer_expires) {}

if (timer_expires)

io_changes 式の定数値の指定（**by** や **to** で指定するもの）は必須ではありませんが、定数値を指定している **io_changes** 式だけは **when** 節のイベント・テーブルの中で最適化されます。

イベント処理

ネットワーク動作に関連したイベントは、2 つの待ち行列を使って処理されます。待ち行列の 1 つは、以下のような着信ネットワーク・メッセージ関連のイベントに使われます。

nv_update_occurs

msg_arrives

online

offline

wink

もう１つの待ち行列は、以下の完了イベントや応答といったネットワーク・イベントに使われます。

nv_update_completes

nv_update_succeeds

nv_update_fails

msg_completes

msg_succeeds

msg_fails

resp_arrives

resp_arrives 以外のほとんどのネットワーク・イベントは、そのイベントがアプリケーションのチェック対象になると Neuron C コンパイラが判断した場合にだけ待ち行列に登録されます。**online**、**offline**、**wink** の各イベントはいつでも待ち行列に登録されますが、該当する **when** 節がなければ、スケジューラによって行列から消去されます。

イベントが待ち行列の先頭に来ると、アプリケーションがそのイベントを処理するまでは待ち行列の先頭からなくなることはありません。そのため、アプリケーションのチェック対象となるイベントは頻繁にチェックする必要があります。そうしないと、待ち行列の先頭がブロックされた状態になってしまいます。待ち行列がブロックされていると、アプリケーションがイベントを処理できなくなり、別のアプリケーションやネットワーク管理メッセージに対してデバイスが応答できなくなります。

このような状態は、**nv_update_occurs** や **msg_arrives** のように要求とは無関係に到着するイベントにとって重大な問題になります。一方、完了イベントや応答は、アプリケーションからネットワークへの発信の結果として到着するものです。Neuron C コンパイラは、イベントが **when** 節でチェック対象となっていなくても、プログラム内にあるものはアプリケーションによって処理されるか、あるいは特別な状況でのみチェックされるかを判断します。

リセット・イベント

なんらかの理由で Neuron チップまたはスマート・トランシーバがリセットされると、その後に評価されたリセット・イベントの値が 1 度だけ TRUE になります（I/O オブジェクトとグローバル変数の初期化は、イベント処理の前に行われます）。リセット・イベントのタスクは、Neuron チップまたはスマート・トランシーバのリセット後最初に実行されるタスクということになります。

リセット・イベントのタスクはデバイスがどのような状態でも実行されるため、デバイスがオフラインになっているか、構成されていない場合でも、初期化が行われます。

リセットは、LONWORKS デバイスのコミッション・プロセスの一部です。リセット・プロセスではリセット・イベント・タスクが実行されます。コミッション・プロセスを完了するにあたってデバイスの状態が遷移しますが、状態の遷移はリセット・イベントを実行してからでないと完了できません。このため、リセット・イベントを短くして、デバイスのコミッションが最大速度で行われるようにしてください。リセット・イベントの処理時間の合計を 18 秒以下にして、コミッションに失敗しないようにする必要があります。リセット・イベントの処理時間には、Neuron ファームウエアの初期化時間が含

まれます。詳細については『Smart Transceivers Databook』を参照してください。

ユーザ定義イベント

ユーザ定義イベントには、代入と関数呼び出しを使用できます。ただし、複雑な関数を呼び出すユーザ定義イベントを使用すると、プログラム中の全イベントの応答時間に影響することがあるため、関数の呼び出しは最小限に抑えるようにしてください。また、ユーザ定義イベントの中で代入を行うときは、グローバル変数に代入しなければなりません。

さらに、次の **timer_expires(t)**のようなイベント・キーワードの評価は、式全体が TRUE であるか FALSE であるかにかかわらず、どのペンディング・イベントもクリアしてしまいます。

```
when ((timer_expires(t)) && (flag == TRUE))
```

ANSI C コンパイラの場合と同様、Neuron C コンパイラは必要がある場合にのみ論理式を評価します。例えば **if (a && b)**の式では、**b** 項は **a** が TRUE である場合にのみ評価され、**if (a || b)**の式では、**b** 項は **a** が FALSE である場合にのみ評価されます。これは *short-circuit evaluation*（短絡評価）と呼ばれ、ANSI C 言語の定義に記載されています。

前述の論理演算子を使用してユーザ定義の式と定義済みイベントを組み合わせるときは、定義済みイベントの評価が妨げられないよう、本章で前述したイベントの待ち行列のブロックを避ける必要があります。

例えば、次のユーザ定義イベントの式は問題ありません。

```
when ((timer_expires(t)) && (flag == TRUE))
```

一方、上の式の項を逆にすると、**flag** 変数が長時間 TRUE となったときにイベントの待ち行列がブロックされるおそれがあります。これは、論理和演算子の短絡によって、タイマーの時間切れイベントがチェックされなくなる場合があるためです。このため、以下のような逆の式は使用しないようにしてください。

```
when ((flag == TRUE) && (timer_expires(t)))
```

when 節のスケジューリング

スケジューラは、ラウンドロビン方式で **when** 節の評価を行います。つまり、スケジューラは各 **when** 節を評価し、それが TRUE であれば条件に結び付いているタスクを実行します。**when** 節を評価した結果が FALSE であれば、次の **when** 節の評価に移ります。最後の **when** 節を処理した後、スケジューラは最初に戻り、再び **when** 節グループの評価を始めます。例えば、**when** 節のグループは次のようになります。

```
when (nv_update_occurs) // Event A
  //  {task to execute}

when (nv_update_fails)         // Event B
  //  {task to execute}

when (io_changes)        // Event C
  //  {task to execute}

when (timer_expires)           // Event D
  //  {task to execute}
```

図 2.1 では、上記のイベント A～D を例にとって、タスクの実行順序がプログラム内の **when** 節の順番とは異なることを示しています。

この例の最初の時点では、C だけが TRUE です。

1. スケジューラは A から評価を開始します。A の評価は FALSE ですから、そのタスクは実行されません。
2. スケジューラは B の評価に移ります。B も FALSE なので、そのタスクは実行されません。
3. A が TRUE になったとします。
4. スケジューラは C の評価に移ります。C は TRUE なので、そのタスクを実行します。
5. スケジューラは D の評価に移ります。D は FALSE なので、そのタスクを無視します。
6. スケジューラは A の評価に戻ります。A の評価は TRUE なので（項目 4 で A が TRUE に変化しています）、そのタスクを実行します。

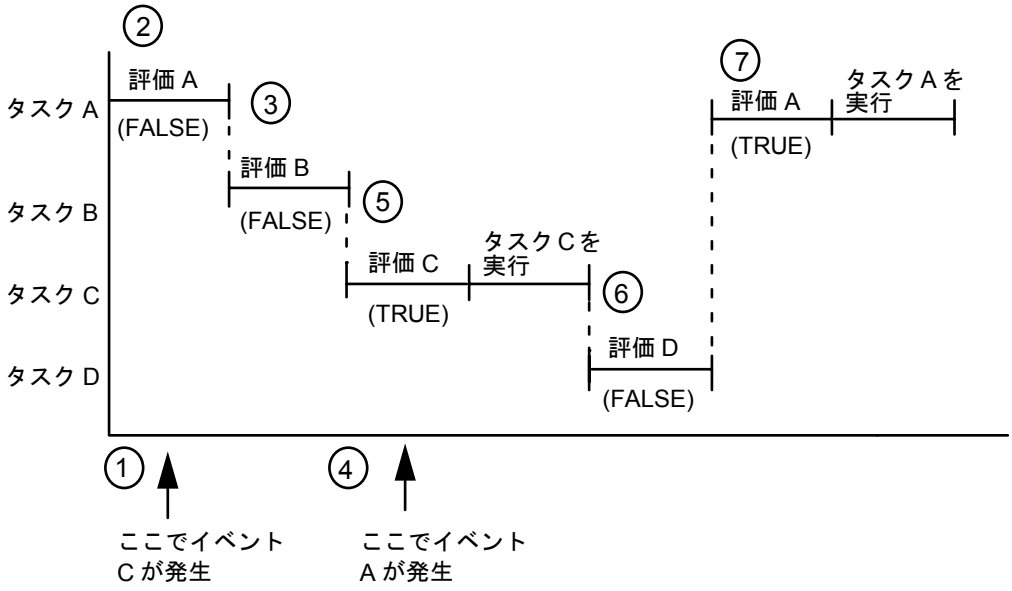


図 2.1　スケジューラによる実行順序例

優先 when 節

priority キーワードは、普通の **when** 節よりも優先的に評価したい **when** 節（優先 **when** 節）がある場合に使用します。優先 **when** 節は、スケジューラが起動するたびに指定順に評価されます。つまり、優先 **when** 節のどれかが TRUE であれば、対応するタスクが実行された後、スケジューラは優先 **when** 節の先頭から評価を再開します。

TRUE と評価される優先 **when** 節がなければ、その後に前述のラウンドロビン方式で選択した非優先 **when** 節を評価します。選択された非優先 **when** 節が TRUE であれば、そのタスクが実行され、スケジューラは再び最初の優先 **when** 節から評価を始めます。選択された **when** 節が FALSE であれば、そのタスクを無視し、スケジューラは最初の優先 **when** 節から評価を始めます。図 7.1 も参照してください。

上記のスケジューリングに関するアルゴリズムは、**scheduler_reset** プラグマを使って変更することもできます。**scheduler_reset** プラグマについては第 7 章「追加機能」で説明します。

警告： 優先 **when** 節を使いすぎると、非優先 **when** 節がなかなか実行されなくなってしまいます。いつも TRUE になるイベントを優先 **when** 節にしてしまうとプロセッサ時間が独占されてしまいます。優先 **when** 節は、まれに TRUE になるようにするか、あるいは残りのタスクが頻繁に実行されなくても構わないような設計にしてください。

関数プロトタイプ

Neuron C では、ある関数を定義する前にその関数を呼び出すには、関数プロトタイプが必要になります。プロトタイプは、次のように指定します。

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

次の例は引数のリストを持たないので、プロトタイプとは言いません。これは単なる前方宣言です。

```
void f();
g();  // defaults to 'int' return value
```

関数呼び出しの前にその関数が定義されていれば、Neuron C が自動的に内部でプロトタイプを作成します。ある関数に対して作成するプロトタイプは 1 つだけです。次の例は、形式的にはプロトタイプになってはいませんが、Neuron C はこれらに対しても関数プロトタイプを生成します。

```
void f()
{ /* body */ }

g (a,b)
int a;
int b;
{ /* body */ }
```

Neuron C はプロトタイプを生成できますが、ANSI C のプロトタイプに関する規定には準拠していません（ANSI C のプロトタイプの規定では、ある関数呼び出しに対するプロトタイプがなければ、自動的にプロトタイプを作成することになっています）。Neuron C では、その関数が定義されているときにのみ、関数プロトタイプを自動的に生成します。

タイマー

Neuron C アプリケーションでは、ミリ秒単位と秒単位の 2 種類のタイマー・オブジェクトを使用できます。ミリ秒タイマーでは 1～64,000 ミリ秒（.001～64 秒）までの時間長を扱い、秒タイマーでは 1～65,535 秒までを扱います。64 秒以下の時間長でより正確なタイミングが必要なときには、ミリ秒タイマーを使用してください。これらのタイマーは、Neuron コアに含まれている 2 つのハードウエア・タイマー/カウンタとは別のタイマーです（後述の「入力クロック周波数とタイマー精度」のセクションも参照してください）。

タイマーの宣言

１つのプログラムの中で、合計 15 個までのタイマー・オブジェクトを定義できます。タイマー・オブジェクトの宣言方法には、次の 2 種類があります。

mtimer [**repeating**] *timer-name* [=*initial-value*]**;**
stimer [**repeating**] *timer-name* [=*initial-value*]**;**

mtimer	ミリ秒タイマーの宣言です。
stimer	秒タイマーの宣言です。
repeating	タイマーが時間切れになったとき、自動的にタイマーを再起動するためのオプション指定です。このオプションを指定すれば、タイマーの時間切れイベントに対してアプリケーション・プログラムが即時に対応できなくても、正確な時間間隔を維持できます。

timer-name　タイマー名を指定します。ここに値を代入すると、指定されただけの時間長のタイマーがスタートします（時間の指定は、**stimer** の場合は秒単位、**mtimer** の場合はミリ秒単位になります）。動作中のタイマーでも、時間切れになったタイマーでも、タイマー名に対して新しい値を代入すれば、そのタイマーが再起動します。タイマーが動作しているときにタイマー・オブジェクトを評価すると、残り時間がわかります。0 に設定すると、そのタイマーは停止します。停止したタイマーには、時間切れのイベントは発生しません（『Neuron C Reference Guide』の **timer_expires** の説明を参照してください）。

initial-value　パワーアップ時もしくはリセット時にタイマーにロードする初期値を指定するオプションです。明示的に初期値が指定されない場合は、Neuron ファームウエアによってゼロがロードされます（タイマーは停止します）。

コード例

タイマー・オブジェクトを宣言して値を代入するコード例を以下に示します。

```
// start timer with value of 5 sec
stimer led_timer = 5;
```

タイマーを停止するコード例を以下に示します。

```
stimer led_timer;
when (t == 50)
{
   led_timer = 0;

}
```

動作中のタイマー値を評価するコード例を以下に示します。

```
stimer repeating led_timer;
when (nv_update_occurs)
{
   time_remaining = led_timer;
   .
   .
   .
}
```

注意： LonBuilder デバッガまたは NodeBuilder デバッガを使っているときにタイマーを設定したり評価したりすると、正確な値にならないことがあります。プログラムを停止（シングル・ステップやブレークポイントなどによる停止を含みます）してプログラム実行中に設定したタイマーの値を見ると、タイマーの値は実際の残り時間より 200 ミリ秒以上長くなります。デバッガを使用していても実行を停止することがなければ、タイマーが不正確になることはありません。

timer_expires イベント

timer_expires イベントは、タイマーの時間切れが起こると TRUE になる時間切れイベントです。このイベントのシンタックスは次のとおりです。

timer_expires [**(***timer-name***)**]

timer-name　　チェックするタイマーを指定します。

timer_name オプションを指定していない時間切れイベントを「非限定 **timer_expires** イベント」といいます。*unqualified event*（非限定イベント）の式では、イベントの適用先のオブジェクトを制限するオプションの修飾子シンタックスを省略します。

タイマー・イベントは、特定の（限定）タイマーの時間切れイベントをチェックすることによってのみクリアできるため、他のイベントとは異なることに注意してください（他のイベントは限定イベントまたは非限定イベントのどちらかをチェックすることでクリアできます）。例えば、次の **when** 節は **led_timer** の時間切れをチェックするため、そのタイマーの **timer_expires** イベントはクリアされて FALSE になります。

コード例

```
stimer led_timer;
when (timer_expires(led_timer))
{
   io_out(io_led, OFF);  // Turn off the LED
}
```

プログラム中で複数のタイマーを使っているときには、それぞれのタイマーをチェックするコードを組み込んでおき、時間切れイベントがクリアされるようにしてください。

```
mtimer x;
mtimer y;
mtimer z;
when (timer_expires(x))
{
   // task
}
when (timer_expires(y))
{
   // task
}
when (timer_expires(z))
{
   // task
}
```

特定のタイマーをチェックするには、以下のような方法もあります。この例は、イベントの式は **when** 節以外でも使えることを示しています。

```
when (timer_expires)
{
   if (timer_expires(x))
     .
     .
     .
   else if (timer_expires(y))
     .
     .
     .
   else if (timer_expires(z))
     .
     .
     .
}
```

注意： 非限定 **timer_expires** イベントを使っているとき、特定のタイマー・イベントをチェック対象とすることを忘れないでください。ペンディング・イベントごとに 1 回だけ TRUE になる他の定義済みイベントとは異なり、非限定 **timer_expires** イベントは、タイマーのどれかが時間切れになると、ずっと TRUE になったままになります。

タイマーの時間切れをチェックするために使用するスタイルは、アプリケーションの状況によって異なります。コード・スペースをなるべく節約したいのであれば、最初のタイマー・チェック方法を使ってください。実行速度、性能、応答時間などが重要なときには、2 番目の方法を使ってください。

タイマーの宣言および **timer_expires** イベントの使用方法については、本章の「サーモスタット用インターフェース」の例を参照してください。

入出力

入出力（I/O）機能を実行するため、Neuron チップおよびスマート・トランシーバにはさまざまな電気的インターフェース・オプションが組み込まれています。Neuron チップおよびスマート・トランシーバの I/O ピンは全部で 11 本あり、入出力を実行する前に、これらの I/O ピンを監視・制御する I/O オブジェクトを宣言する必要があります。I/O ピンの名前は **IO_0**、**IO_1**、…、**IO_10** です。宣言していないピンは、デフォルトでは未使用、つまりインアクティブの状態となります。インアクティブのピンは、高インピーダンス状態です。I/O オブジェクトの宣言シンタックスの詳細については、『Neuron C Reference Guide』を参照してください。

注意： 未使用の入力ピンにはプルアップ抵抗を使用してください。**IO4**～**IO7** のピンには **enable_io_pullups** プラグマを使用できます（このプラグマの詳細については、『Neuron C Reference Guide』の「Compiler Directives」の章を参照してください）。未使用のピンにプルアップ抵抗を使用しない場合は、出力として定義します。

入出力を実行するには通常、**io_in()**、**io_out()**、**io_set_direction()**、**io_select()**、**io_change_init()**、**io_set_clock()**という組み込み入出力関数を使用します。ま

た、パラレル I/O オブジェクトを使った入出力には **IO_out_request()**関数を使用します。本章では、これらの入出力関数の使用方法について説明します。

Neuron C イベントは、I/O オブジェクトにもリンクさせることができるため、入出力の変化をタスクのスケジューリングに反映させることも可能です。**when** 節の中で使用する入出力関連イベント（**io_changes** と **io_update_occurs**）については、本章の「I/O イベント」を参照してください。入出力に関する詳細情報、コード例、使用方法については、次の LONWORKS の技術資料を参照してください。

- 『Analog-to-Digital Conversion with the Neuron Chip engineering bulletin』（part no. 005-0019-01）
- 『Driving a Seven Segment Display with the Neuron Chip engineering bulletin』（part no. 005-0014-01）
- 『NEURON CHIP QUADRATURE INPUT FUNCTION INTERFACE engineering bulletin』（PART NO. 005-0003-01）
- 『Parallel I/O Interface to the Neuron Chip engineering bulletin』（part no. 005-0021-01）
- 『EIA-232C Serial Interfacing with the Neuron Chip engineering bulletin』（part no. 005-0008-01）

I/O オブジェクト型

Neuron C には、ダイレクト、タイマー/カウンタ、シリアル、パラレルといった種類の I/O オブジェクト型があります。それぞれの種類の I/O オブジェクト型について、以下に説明します。

- *Direct I/O Object*（ダイレクト I/O オブジェクト）：
  I/O ピンの論理レベルを直接扱うための I/O オブジェクトです。この I/O オブジェクトと Neuron チップまたはスマート・トランシーバのハードウエア・タイマー/カウンタとの関連はありません。同じ Neuron チップまたはスマート・トランシーバの中で複数のダイレクト I/O オブジェクトを使用できますし、それらのオブジェクトが使用するピンがオーバーラップするような組み合わせになっていても構いません。ダイレクト I/O オブジェクト型を次に示します。

入力オブジェクト型	出力オブジェクト型
bit	**bit**
bitshift	**bitshift**
byte	**byte**
nibble	**nibble**
leveldetect	**touch**
touch	

- *Timer/Counter I/O Object*（タイマー/カウンタ I/O オブジェクト）：
  Neuron チップまたはスマート・トランシーバにあるタイマー/カウンタ回路を使用する I/O オブジェクトです。Neuron チップおよび各スマート・トランシーバにはそれぞれ 2 つのタイマー/カウンタ回路があります。一方の入力は多重化でき、もう一方は専用入力回路として使用します（図 2.2 参照）。タイマー/カウンタ I/O オブジェクト型を次に示します。

入力オブジェクト型	出力オブジェクト型
dualslope	**edgedivide**
edgelog	**frequency**
ontime	**oneshot**
period	**pulsecount**
pulsecount	**pulsewidth**
quadrature	**triac**
totalcount	**triggeredcount**

- *Serial I/O Object*（シリアル I/O オブジェクト）：
  １本または複数のピンを使って、シリアル・データ転送を行うための I/O オブジェクトです。1 つの Neuron チップまたはスマート・トランシーバに対し、1 種類のシリアル I/O オブジェクトだけを定義できます。ただし、シリアル入力とシリアル出力の両方を 1 つの Neuron チップまたはスマート・トランシーバの中に共存させることは可能です。シリアル I/O オブジェクト型を次に示します。

シリアル入力オブジェクト型	出力オブジェクト型
infrared	**serial**
magcard	
magtrack1	
serial	
wiegand	

シリアル入力/出力オブジェクト型
i2c
neurowire

- *Parallel I/O Object*（パラレル I/O オブジェクト）：
  高速双方向入出力に使用する I/O オブジェクトです。このグループに属するオブジェクト型は、Neuron チップまたはスマート・トランシーバのすべての I/O ピンを使用します。パラレル I/O オブジェクト型を次に示します。

パラレル入力/出力オブジェクト型
muxbus
parallel

表 2.1（次の２ページ）では、I/O オブジェクト型、使用するピン、指定できるオプションをリストします。オブジェクト型については、『Neuron C Reference Guide』も参照してください。

表 2.1　I/O オブジェクト型（1/2）

オブジェクト型	オブジェクトの最大数	宣言に使用するピン/ １オブジェクトが使用するピンの数	オプション指定
Bit 入力	11	任意のピン/１本	--
Bit 入力	11	任意のピン/１本	initial_output_level
Bitshift 入力	5	IO_0～IO_6、IO_8、IO_9/２本	numbits, clockedge, kbaud
Bitshift 出力	5	IO_0～IO_6、IO_8、IO_9/２本	numbits, clockedge, kbaud, initial_output_level
Byte 入力	1	IO_0/8 本	--
Byte 出力	1	IO_0/8 本	initial_output_level
デュアルスロープ (Dualslope)入力	2	IO_4-～O_7。入力が IO_4（mux）または IO_5～IO_7 のとき、IO_0 を使用。入力が IO_4（ded）のとき、IO_1 を使用/2 本	invert, clock
エッジデバイド (Edgedivide)出力	2	IO_0 または IO_1。IO_0 が出力ピンのとき、IO_4～IO_7 を同期ピンとして使用可能。IO_1 が出力ピンのとき、同期ピンは IO_4/2 本	invert, clock, sync pin initial_output_level
エッジログ (Edgelog) 入力	1	IO_4/1 本	clock
フリケンシー (Frequency) 出力	2	IO_0、IO_1/1 本	invert, clock, initial_output_level
I^2C 入力/出力	1	IO_8/2 本	--
インフレアード (Infrared)入力	4	IO_4～IO_7/1 本	invert, clock
レベルデテクト (Leveldetect)入力	8	IO_0～IO_7/1 本	--
マグカード (Magcard)入力	1	IO_8（2 本使用）。タイムアウト・ピンは IO_0～IO_7（１本）/計 3 本	invert, clockedge, timeout pin
マグトラック 1 (Magtrack1)入力	1	IO_8（2 本使用）。タイムアウト・ピンは IO_0～IO_7（１本）/計 3 本	invert, clockedge, timeout pin
マックスバス (Muxbus)入力/出力	1	IO_0/11 本	--
ニューロワイヤーマスター (Neurowire master) 入力/出力	8	IO_8（3 本使用）。セレクト・ピンは IO_0～IO_7（１本）/計 4 本	select pin, kbaud

表 2.1　I/O オブジェクト型（2/2）

オブジェクト型	オブジェクトの最大数	宣言に使用するピン/１オブジェクトが使用するピンの数	オプション指定
ニューロワイヤースレーブ(Neurowire slave)入力/出力	1	IO_8（3 本使用）。タイムアウト・ピンは IO_0～IO_7（１本）/計 4 本	clockedge, timeout pin
ニブル(Nibble) 入力	2	IO_0～IO_4/4 本	--
ニブル(Nibble) 出力	2	IO_0～IO_4/4 本	initial_output_level
Oneshot 出力	2	IO_0、IO_1/1 本	invert, clock, initial_output_level
Ontime 入力	5	IO_4～IO_7/1 本	mux|ded, invert, clock
Parallel 入力/出力	1	IO_0/11 本	slave|slave_b|master
Period 入力	5	IO_4～IO_7/1 本	mux|ded, invert, clock
Pulsecount 入力	5	IO_4～IO_7/1 本	mux|ded, invert
Pulsecount 出力	2	IO_0、IO_1/1 本	invert, clock
Pulsewidth 出力	2	IO_0、IO_1/1 本	invert, clock, short, long initial_output_level
クアドラチュア(Quadrature)入力	2	IO_4、IO_6/2 本	--
Serial 入力	1	IO_8/1 本	baud
Serial 出力	1	IO_10/1 本	baud
Totalcount 入力	5	IO_4～IO_7/1 本	mux|ded, invert
Touch 入力/出力	制限なし	IO_0～IO_7/1 本	--
トライアック(Triac)出力	2	IO_0 または IO_1。IO_0 が出力ピンのとき、IO_4～IO_7 を同期ピンとして使用可能。IO_1 が出力ピンのとき、同期ピンは IO_4/2 本	sync pin, invert, clock, clockedge
Triggeredcount 出力	2	IO_0 または IO_1。IO_0 が出力ピンのとき、IO_4～IO_7 を同期ピンとして使用可能。IO_1 が出力ピンのとき、同期ピンは IO_4/2 本	sync pin, invert
ウィガンド(Wiegand)入力	4	IO_0～IO_6/2 本	timeout pin

I/O オブジェクトの宣言

アプリケーション内の I/O オブジェクトの宣言には、次の意味があります。

1 宣言では、どの種類の入出力操作をどのピン上で行うかをコンパイラに指示します。この宣言を受け、コンパイラは Neuron コア内のハードウエアを構成するための命令を作成します。ハードウエアの構成コードはデバイスのアプリケーションがリセットされるたびに実行されます。

2 宣言によって、I/O オブジェクト名とハードウエアとが関連付けられます。

このセクションでは、Neuron C 言語における I/O オブジェクト宣言の一般的なシンタックスについて説明します。各 I/O オブジェクト型のシンタックスの詳細については、『Neuron C Reference Guide』を参照してください。

pin type [options] io-object-name;

pin — Neuron C のキーワードの 1 つ。**IO_0** から **IO_10** までの 11 本の I/O ピンに名前を付けます。一般的には、同じピンが複数のオブジェクト宣言に指定されることはありません。ただし、I/O オブジェクト型が **bit**、**nibble**、**byte** のときには、同じピンが複数の宣言に出てくることもあります。また、**IO_8** を **neurowire master** の複数の宣言で使用して、それぞれ別のセレクト・ピンを指定することもできます。この場合、すべての宣言が同じ向き（入力か出力か）になっている必要はありません。「I/O オブジェクトのオーバーレイ」のセクションを参照してください。

type — I/O オブジェクト型を指定します。

options — 選択した I/O オブジェクトの型に応じたオプションの I/O パラメータです。各オブジェクト型に指定できるオプションについては、『Neuron C Reference Guide』を参照してください。特に注記がなければ、オプションの指定順序に決まりはありません。これらのオプションを省略するとデフォルト値が設定されます。

io-object-name — I/O オブジェクト名を指定します。ANSI C の変数識別子の形式で指定してください。

次の例では、デバイスの **IO3** 入力ピン（Neuron C での名前は **IO_3**）で論理レベルを計測します。このピンは、その名前が示すとおり近接検出器（proximity detector）に接続されます。

```
IO_3 input bit ioProxDetector;
```

このように宣言しておけば、作成したプログラムで **ioProxDetector** を参照することが、実際にはピン **IO3** の論理レベルを調べることになります。

I/O リソースの使用

I/O オブジェクト型の宣言に関するガイドラインを、以下のリストと表 2.2 に示します。

- 宣言できる I/O オブジェクトの最大数は 16 個です。

- タイマー/カウンタ 1 は、最大 4 個の入力オブジェクトを多重化できます。
- **neurowire**、**i2c**、**magcard**、**magtrack1**、**serial** の各 I/O オブジェクトは互いに排他的です。1 つのプログラムで宣言できるのは、どれか一種類の I/O オブジェクトです。
- **parallel**、**muxbus** の各 I/O オブジェクトは、すべての I/O ピンを使用します。そのため、このオブジェクトのいずれかが宣言されているときは、他のオブジェクト型を宣言することはできません。
- **bit**、**nibble**、**byte** などの各ダイレクト I/O オブジェクト型（本章「I/O オブジェクト型」のダイレクト I/O オブジェクトの説明を参照してください）は、どのような組み合わせでも宣言できます。次の「I/O オブジェクトのオーバーレイ」のセクションを参照してください。タイマー/カウンタ、**serial**、**neurowire** I/O オブジェクトの宣言は、オーバーレイしているダイレクト I/O オブジェクト型のピンの方向（入力か出力か）より優先します。
- **quadrature**、**dualslope** の各入力オブジェクトは、タイマー/カウンタ 1 上の他の入力オブジェクトと多重化することはできません。**edgelog** 入力はタイマー/カウンタを両方とも使用し、他のタイマー/カウンタとは併用できません。
- **bitshift** I/O オブジェクトは、同じ I/O ピン上でタイマー/カウンタ・オブジェクトとして宣言することはできません。ダイレクト I/O オブジェクトは **bitshift** I/O オブジェクトにオーバーレイできます。隣り合わせの 2 つの **bitshift** I/O オブジェクトは、どの I/O ピンも共有できません。

Neuron チップまたはスマート・トランシーバ上で混在可能な I/O オブジェクト型の組み合わせ例を示します。

A) **parallel** I/O オブジェクト型 1 個（**IO_0** を使用）

または

B) **muxbus** I/O オブジェクト型 1 個（**IO_0** を使用）

または

C) その他の I/O オブジェクトの組み合わせ

1) a) タイマー/カウンタ入力 1～4 個（**IO_4**、**IO_5**、**IO_6**、**IO_7** で多重化）。IO_6 は **quadrature** 入力に使用。
または
b) タイマー/カウンタ出力 1 個（**IO_0** を使用）

かつ

2) a) タイマー/カウンタ入力 1 個（**IO_4** を使用）。**IO_4** は **quadrature** 入力に使用。
または
b) タイマー/カウンタ出力 1 個（**IO_1** を使用）

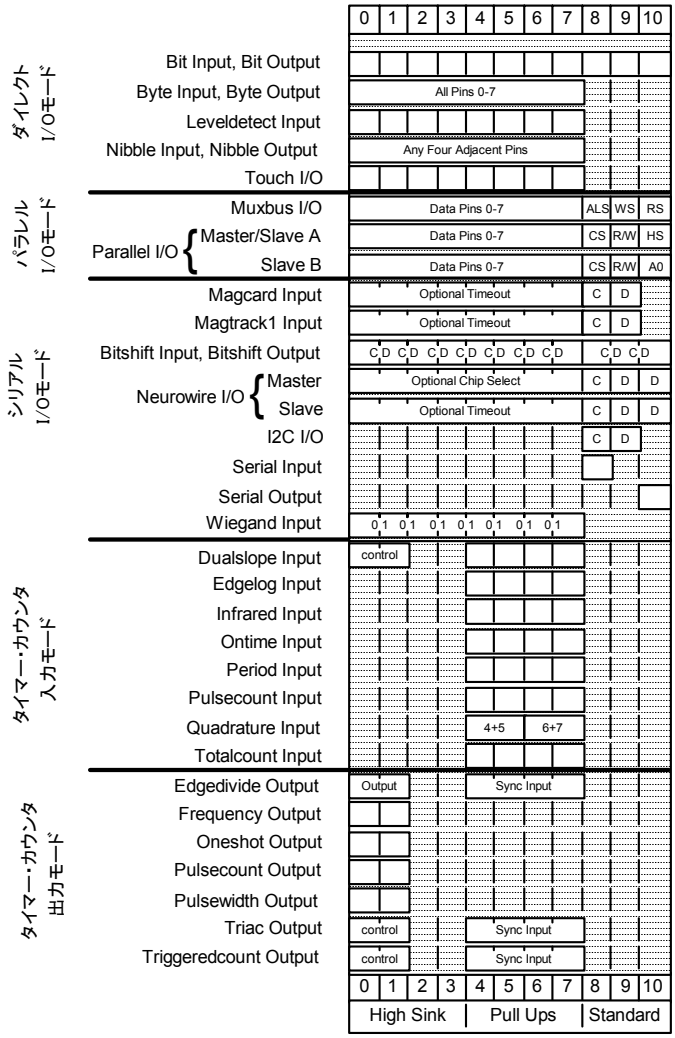
かつ

3) a) **neurowire** I/O オブジェクト型 1 個（**IO_8**、**IO_9**、**IO_10** を使用）と **IO_0** から **IO_7** までの中の 1 個
または
b) シリアル I/O オブジェクト型 1 個（**IO_8**、**IO_10** を使用）

かつ

4) 任意のピン（**IO_0**～**IO_10**）を使った任意のダイレクト I/O オブジェクト型。

表 2.2　I/O デバイス



Bitshift, Neurowire: C=Clock D=Data

タイマー/カウンタ 1 デバイス

次のいずれかに指定

- **IO_6 input quadrature**
- **IO_4 input edgelog**
- **IO_0 output [triac|triggeredcount|edgelog] sync([IO_4...IO_7])**
- **IO_0 output [frequency|oneshot|pulsecount|pulsewidth]**
- **最大 4 個までの次の宣言:**
  IO_4 input [ontime|period|pulsecount|totalcount|dualslope|
  [IO_5...IO_7]input[ontime|period|pulsecount|totalcount|dualslope|infrared
- **IO_4 input edgelog**

タイマー/カウンタ 2 デバイス

次のいずれかに指定

- **IO_4 input quadrature**
- **IO_4 input edgelog**
- **IO_1 output [triac|triggeredcount|edgedivide] sync(IO_4)**
- **IO_1 output [frequency|oneshot|pulsecount|pulsewidth]**
- **IO_4 input [ontime|period|pulsecount|totalcount|dualslope|infrared] ded**

I/O オブジェクトのオーバーレイ

同一のピンに対して複数の I/O オブジェクトを宣言することもできます。次の例の最初の宣言では、**nibble** オブジェクトを使って連続した 4 本のピンを１回の操作で読み取るようにしています。個々のピンを読み取ることもできるように、その後に 4 つの **bit** オブジェクトを宣言しています。

```
IO_4 input nibble io_all_points;
IO_4 input bit io_point_1;
IO_5 input bit io_point_2;
IO_6 input bit io_point_3;
IO_7 input bit io_point_4;
```

oneshot 出力オブジェクトで使用するピンのレベルをモニター（読み取り）するプログラムを作成するには、次のように宣言します。

```
IO_1 output oneshot clock (3) io_break_high;
IO_1 input bit io_break_high_level;
```

I/O オブジェクト型をオーバーレイに関して分類すると、ハード・ピン方向 I/O オブジェクトとソフト・ピン方向 I/O オブジェクトに分かれます。「ソフト」ピン方向 I/O オブジェクト（**bit**、**nibble**、**byte** オブジェクト型）は、それ以降にある宣言によってピン方向（入力か出力か）が変更される可能性があります。同じピンに対して複数のソフト・ピン方向 I/O オブジェクトが宣言されていると、最後に宣言されたソフト・ピン方向 I/O オブジェクトによって実行時のピン方向の初期状態が決まります。一方、「ハード」ピン方向 I/O オブジェクト（その他の I/O オブジェクト型）は、それ以降にどのような宣言があっても影響されません。

io_set_direction()関数を使用すれば、アプリケーションの実行時に **bit**、**nibble**、**byte** 型 I/O ピンの方向を変更できます。**io_set_direction()**については、『Neuron C Reference Guide』を参照してください。

前に示した **oneshot** 出力オブジェクトと **bit** 出力オブジェクトの例では、**oneshot** がハード・ピン方向 I/O オブジェクトで、**bit** がソフト・ピン方向 I/O オブジェクトです。つまり、どのような順序で宣言されていたとしても、**oneshot** オブジェクトによって実行時の **IO_1** のピン方向が決まります。ピン **IO_1** はリセット後の初期化中に設定されます。

例えば、次のような宣言を含むプログラムがあるとします。

```
IO_2 input bit io_point_1;
IO_2 output bit io_point_2;
```

後にある宣言が出力になっていますから、ピン **IO_2** は出力 I/O オブジェクトになります。この後に **io_point_2** に対する **io_out()**の呼び出しがあれば、このピンのレベルが設定されます。この出力オブジェクトに対応するピンの実際のレベルを調べるには、**io_point_1** に対して **io_in()**呼び出しを実行します。ただし、ここでは **io_set_direction()**は呼び出されていないものとします。

入出力の実行：関数とイベント

入力オブジェクトにアクセスしてその値を得るには、2 つの方法があります。1 つは **io_in()**関数を使う方法で、もう 1 つはオブジェクトに関連しているイベントを **when** 節の中で使用する方法です。以下のセクションで、それぞれの方法について説明します。

入出力関数

Neuron C アプリケーションは、一度 I/O オブジェクトを宣言すると、次からは Neuron C で用意している入出力関数を使ってオブジェクトにアクセスできるようになります。これらの関数は Neuron C のコンパイラに組み込まれているので、宣言やリンクの必要はありません。関数のパラメータに対する型チェックはコンパイラが行います。Neuron C で用意している入出力関数は次のとおりです。

io_change_init()	**io_changes** イベントで使用する入力オブジェクトの値を初期化します。
io_edgelog_preload()	タイマー/カウンタのロード済み値をセットします。
io_in()	I/O オブジェクトからデータを読み込みます。
io_in_ready()	**parallel** I/O オブジェクトからデータブロックを読み取れるようになったときに評価値が TRUE になるイベント関数です。
io_in_request()	**dualslope** I/O オブジェクトに対する I/O 入力サイクルを開始します。
io_out()	I/O オブジェクトへデータを書き込みます。
io_out_request()	**parallel** I/O オブジェクト用の書き込みトークンを要求します。
io_preserve_input()	リセットまたは **io_select()**の後でタイマー/カウンタから得られた最初の値を有効にします。
io_select()	多重化された入力オブジェクトの 1 つを選択します（「入出力の多重化」のセクションを参照してください）。
io_set_clock()	特定のオブジェクトに設定されているクロックを変更します。
io_set_direction()	**bit**、**nibble**、**byte** 型などの I/O ピンの方向を変更します。

詳しくは、『Neuron C Reference Guide』を参照してください。

io_in( )関数

io_in()のシンタックスは以下のとおりです。

return-value = **io_in (** *io-object-name* [**,** *args*] **)**

io-object-name　I/O オブジェクト名を指定します。I/O オブジェクト宣言の中の *io-object-name* に対応します。

args　I/O オブジェクトの型に応じた引数を指定します。引数の中には、I/O オブジェクトの宣言でも指定できるものがあります。両方で指定されている場合、この関数が呼び出されている間だけここで指定した引数が優先されます。なお、関数引数にも宣言にも値が指定されていなければ、デフォルト値になります。

次のコード例の **io_in()**関数は、**io_part_detector** の値を戻します。

```
part_detected = io_in(io_part_detector);
```

各オブジェクトの **io_in()**に関する規則については、『Neuron C Reference Guide』を参照してください。

io_out( )関数

デバイスに対して信号を送る必要があるときは、出力オブジェクトを宣言して **io_out()**組み込み関数を使用します。

io_out( )のシンタックスは以下に示すとおりです。

io_out (*io-object-name*, *output-value* [, *args*] **)**

例えば、照明ランプのスイッチを切り替える場合などに **io_out()**を使います。この場合、**nv_lamp_state** は入力ネットワーク変数で、値は LONWORKS ネットワーク内の他の場所から得られます。

```
io_out(io_lamp_out,
       (nv_lamp_state != ST_OFF) ? 1 : 0);
```

以下に **IO_0** ピンに接続された LED の表示を制御するコード例を示します。宣言のシンタックスは以下のとおりです。

```
#define ON 1
#define OFF 0
IO_0 output bit io_display_LED;
// or
IO_0 output bit io_display_LED = ON;
```

上の例の 2 番目の宣言では、「初期化子」を使用しています。このように指定されていると、リセットが発生した後、システムは **io_display_LED** オブジェクトの出力値を 1 に初期化します。デフォルトの初期値は 0 です。

I/O オブジェクトの宣言を行ったので、これで **io_out()**関数を使って **io_display_LED** の状態を制御できるようになります。

```
if (flow_total > 500)
   io_out(io_display_LED, ON);
```

input_is_new 変数

タイマー/カウンタ入力オブジェクトに対する **io_in()**呼び出しの戻り値が更新されると、組み込み変数 **input_is_new** の値が TRUE になります。これは暗黙的呼び出しの場合も同様です。暗黙的 **io_in()**呼び出しについては、後述の「I/O イベント」の説明を参照してください。**input_is_new** 変数のデータ型は

unsigned short です。更新が発生する頻度は、I/O オブジェクト型によって異なります。

以下に、タイマー/カウンタ I/O デバイスの 1 つを使った例を示します。この例では、ピン **IO_7** に光学流量計が接続されているものとします。この計測器は、液体の量に比例したパルス数を出力します。そのパルスを数えることで、総量が何ガロンであるかがわかります。この例では、Neuron チップまたはスマート・トランシーバのクロック・スピードは 10MHz と仮定します。

パルス数を数えるには、**pulsecount** オブジェクトを使用します。**pulsecount** 入力オブジェクトは入力エッジをカウントし、約 0.8388608（正確には $2^{23}/10^7$ 秒）毎にその数をラッチします。この I/O オブジェクトに対して **io_in()**関数を使うと、常に現在ラッチされている値を読み込むことになります。総流量を計算するためには、ラッチされた値を合計しなければなりません。これには、**input_is_new** 変数を使用してください。**input_is_new** 変数は、**io_in()**関数の後で新規に計測された値があれば（この場合は 0.8388608 秒毎に）TRUE になります。

```
IO_7 input pulsecount io_flow_sensor;
       // 451 pulses/gallon
long volume_total, volume_temp;

.
.
.
{
   volume_temp = io_in(io_volume_sensor);
   if (input_is_new)
      volume_total += volume_temp;
}
.
.
.
```

I/O イベント

入力オブジェクトにアクセスするには、**io_in()**関数を使う方法の他に、入力オブジェクトを扱う定義済みイベントを使用する方法があります。入出力関連の定義済みイベントには、**io_changes** と **io_update_occurs** の 2 つがあります。どちらのイベントも、内部では **io_in()**関数を呼び出しています。これらのイベントは必ず入力オブジェクトと一緒に使用され、さまざまな形式指定することができます。**io_update_occurs** でも **io_changes** でも、評価時に内部で **io_in()**関数を呼び出してオブジェクトの入力値を取得します。ここで得られた入力値にアクセスするには、タスクの中でキーワード **input_value** を使います。これらの I/O イベントとキーワードについて、以下のセクションで説明します。

io_changes イベント

このイベントは、指定した入力オブジェクトから読み込まれた値が変化したときに TRUE になります。状態の変化には、次の 3 種類があります。

- なんらかの変化が起こった（任意の変更）
- 変化量の絶対値が特定の値以上あった
- 値が特定の値になった

このイベントの Neuron C シンタックスは以下のとおりです。

io_changes(*io-object-name***)** [**by** *expr* | **to** *expr*]

このイベントを使用すると、入力オブジェクトから読み込まれた現在値が参照値と比較されます（**to** オプションがある場合を除く）。「参照値」とは、**change** イベントが最後に TRUE と評価されたときに読み込まれ、ファームウエアによって保存された値です。**by** オプションまたは **to** オプションのどちらも使用していない **io_changes** イベントでは、状態の変化は現在の値が参照値と異なるときに発生します。オプション形式での比較は前述のとおりです。**io_changes** イベントをオプション付きで使用するとき、*expr* 式は定数でなくてもかまいませんが、定数の式を使用した方が効率が高くなります。

例えば、**io_changes** イベントを使用すると、**io_switch_in** 入力ビット・オブジェクトの変化を検出できます。

```
when (io_changes(io_switch_in))
```

もし **io_part_detector** がパーツを検出した（値が TRUE あるいは 1 になった）ときにだけ特定の処理をしたいのであれば、次のような **when** 節を使うこともできます。

```
when (io_changes(io_part_detector) to TRUE)
{
   .
   .
   .
}
```

io_update_occurs イベント

このイベントのシンタックスは以下のとおりです。

io_update_occurs (*io-object-name***)**

io_update_occurs イベントは、*io-object-name* に指定した入力オブジェクトから読み込んだ値が更新されていれば TRUE になります。**io_update_occurs** イベントは、タイマー/カウンタ入力オブジェクトにしか使用できません。どのようなタイミングでイベントが起こるかは、入力オブジェクト型によって異なります。

dualslope	変換が完了するとイベントが発生し、値が変化します。
ontime, period	指定されただけの時間が経過すると、イベントが発生します。
pulsecount	0.8388608 秒毎にイベントが発生します。つまり、新しいパルスカウント値があるときにイベントが発生します。
quadrature	少なくとも 1 カウントの変化があると、直後にイベントが発生します。

タイマー/カウンタ入力デバイスに対する **io_changes** イベントは、そのデバイスの値が前の値とは異なる新しい値になったときに発生します。タイマー/カウンタ・デバイスでは、**io_changes** イベントは入力オブジェクトの型に応じて次のように発生します。

dualslope	変換が完了するとイベントが発生します。
ontime, **period**	計測時間が前回の値とは異なるときにイベントが発生します。
pulsecount	計測されたカウント数が最新値とは異なるときにイベントが発生します。
quadrature	計測されたカウント数が最新値とは異なるときにイベントが発生します。

input_value 変数

input_value は **signed long** 型の組み込み変数です（**input_value** は、他の C の変数と同じ方法で型変換できます）。この変数は次のように使用します。

```
when (io_update_occurs(io_dev))
{
   if (input_value > 2) {
      // code
   }
}
```

照明ランプ・デバイスは、**input_value** の値（スイッチの値）に基づいて **nv_switch_state** ネットワーク変数の値を設定できます。

```
when (io_changes(io_switch_in))
{
   nv_switch_state
       = (input_value == SWITCH_ON) ? ST_ON : ST_OFF;
}
```

input_value 変数の値は、それが使われているコンテキストに依存します。以下に示す **when** 節の組み合わせは正しいシンタックスです。両方のイベントが同じ I/O オブジェクトを参照するため、どちらのイベントが発生しても **input_value** がどのオブジェクトの入力値になっているかがはっきりしています。

```
when (io_changes(io_dev) to 4)
when (io_changes(io_dev) to 3)
{
   x = input_value;
}
```

しかし、以下に示す **when** 節の組み合わせでは、どのオブジェクトの値が **input_value** になるのかがわかりません。例えば、最初の **when** 節が TRUE と評価されたときには、**input_value** は **io_dev2** の値になります。また、2 番目の **when** 節が TRUE になったのであれば、**input_value** は **io_dev1** の値です。このような組み合わせは、正しいシンタックスではありません。

```
when (io_update_occurs(io_dev2))
when (io_update_occurs(io_dev1))
{
   x = input_value;
}
```

さらに、**input_value** は **io_update_occurs** または **io_changes** イベントが発生したときにだけ有効であることに注意してください。次の例のように **when** 節を組み合わせた場合、**timer_expires** イベントでは入出力が実行されませんか

ら、**input_value** の値は不定になります。このような場合は、**io_in()**を使って値を取得してください。

```
when (timer_expires(t))
when (io_update_occurs(io_dev))
{
   x = input_value;
   // use  x=io_in(io_dev) instead of input_value
}
```

入力オブジェクトへのアクセス方法の選び方

これまでに、入力オブジェクトの値が新しくなったかどうかを調べる 2 つの方法を紹介しました。1 つは **io_update_occurs** イベントと **input_value** 変数を使う方法で、もう 1 つは **io_in()**関数と **input_is_new** 変数を使う方法です。次の 2 つのプログラム例では、それぞれの方法を使って同じ機能を実現しています。

リスト 2-1　io_update_occurs/input_value

```
IO_5 input pulsecount io_dev;

when (io_update_occurs(io_dev))
{
   if (input_value > 2) {
      // code
   }
}
```

リスト 2-2　io_in( )/input_is_new

```
stimer t;
IO_5 input pulsecount io_dev;
when (timer_expires(t))
{
   // code
   if ((io_in(io_dev) > 2) && input_is_new) {
      // code
   }
}
```

どちらの方法が適しているかは、個々のケースに依存します。I/O イベントを使う方法（リスト 2-1 の **when** 節を使用したもの）の方が簡単で、いつ入出力関数を実行するかはスケジューラに任されています。できれば、こちらの方法を使ってください。単一ブロック内で複数のイベントを処理する場合には、リスト 2-2 に示したような **input_is_new** 変数と **io_in()**関数の明示的な実行が必要になるでしょう。

注意事項

I/O イベントを含む **when** 節のタスクの中で **io_in()**関数を呼び出して入力値を取得すると、同期の問題が生ずる恐れがあります。例えば、入出力のサンプリング時間の終わり近くで **when** 節が TRUE と評価された場合、**io_in()**呼び出しが実行されたときにはすでに次のサンプリング時間になっていて、得られた値が適切なものにはなっていない可能性があります。

```
when (io_update_occurs(dev))
{
      // code
      io_in(dev); // Use input_value instead
            // of io_in() to retrieve
            // the value obtained when
            // the io_update_occurs
            // event was TRUE
}
```

I/O 計測、出力、関数の相関関係

ダイレクト、シリアル、パラレル I/O オブジェクト

ダイレクト I/O オブジェクトの入力レベルは **io_in()**関数が実行された時点、あるいはオブジェクトを参照している **when** 節が評価された時点で計測されます。

シリアル I/O オブジェクトとパラレル I/O オブジェクトの入力レベルは、**io_in()**関数が呼び出された時点でサンプリングされます。入力クロックが 40MHz の場合、出力レベルは **io_out()**関数の実行後およそ 12.5～25 マイクロ秒でセットされます（この値は、比較的遅いクロック・スピードでの計測値です）。『FT 3120 and FT 3150 Smart Transceivers Databook』に詳しいタイミング図が掲載されていますので、参照してください。

タイマー/カウンタ I/O オブジェクト

タイマー/カウンタ入力オブジェクトの値は定期的にラッチされ、その間隔はオブジェクト型やオブジェクトのクロックに依存します。**io_in()**関数または I/O **when** 節が使用される時点と、データがラッチされる時点との相関関係は、アプリケーションに依存します。一度値がラッチされると、ハードウエアのタイミングに基づいて新しい値がラッチされるまで、**io_in()**はその後の呼び出しでも同じ値を返し続けます。

period 入力と **ontime** 入力オブジェクト型は、入力信号の立ち下がりエッジで新しい値をラッチします。ただし、**invert** キーワードを使用しているときには、これらのオブジェクト型は入力信号の立ち上がりエッジで新しい値をラッチします。**pulsecount** 入力オブジェクトは、0.8388608 秒毎に新しい値をラッチします（本章で後述する「入力クロック周波数とタイマー精度」のセクションを参照してください）。

一般的には、タイマー/カウンタ出力オブジェクトに書き込まれた新しい値は、現行の出力信号周期の終わりに有効になります。ただし、**oneshot** 出力と、無効になっている I/O オブジェクト（制御値がゼロのもの）は例外です。このようなオブジェクトは、**io_out()**関数から戻ってくると新しい値が有効になります。

詳しくは、『Smart Transceivers Databook』を参照してください。

出力オブジェクト

以下のタイマー/カウンタ出力オブジェクト型は、現行の出力信号周期の終わりで新しい出力値になります。

edgedivide 出力
frequency 出力
pulsewidth 出力

triac 出力
triggeredcount 出力

以下のタイマー/カウンタ出力オブジェクト型は、**io_out()**関数から戻ったときに新しい出力値になります。

oneshot 出力
pulsecount 出力

どのタイマー/カウンタ出力オブジェクトでも、**io_out()**関数から戻ったときの出力値は 0 になっています。

入出力の多重化

2 つあるタイマー/カウンタ回路のうちの 1 つでは、入力はピン **IO_4** から **IO_7** までの間で多重化でき、出力のときには **IO_0** を使用します。このタイマー/カウンタのことを「多重化」タイマー/カウンタと呼びます。もうひとつののタイマー/カウンタ回路デバイスは、入力が **IO_4** だけで、出力は **IO_1** です。この 2 番目のタイマー/カウンタ回路を「専用」タイマー/カウンタと呼びます。図 2.2 に多重化されたタイマー/カウンタ回路と、専用のタイマー/カウンタ回路の信号流れ図を示します。

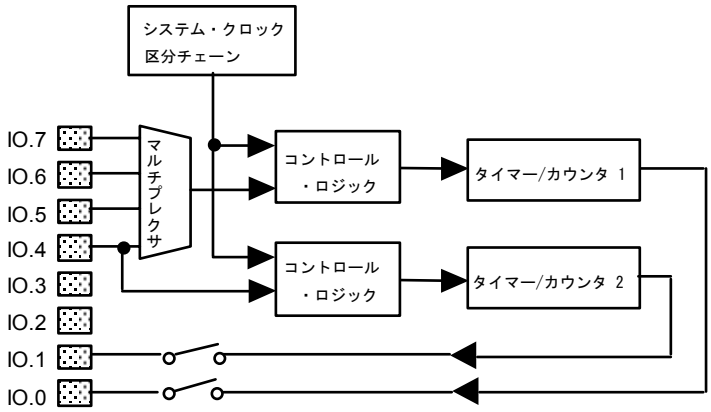


図 2.2　タイマー/カウンタ回路の流れ図

タイマー/カウンタ I/O オブジェクトの入出力関数

多重化された I/O オブジェクトでは、プログラムの中で最後に宣言されたタイマー/カウンタ I/O オブジェクトが、リセット後最初に有効になります。オブジェクトの選択を変更するには、**io_select()**関数を使用します。この関数には、多重化されたピンのうちどれがタイマー/カウンタ回路を使用するかを指定します。**io_select()**のシンタックスは以下のとおりです。

io_select (*io-object-name* [**,** *clock*] **)**

io-object-name　　オブジェクト名を指定します。I/O 宣言部分の *io-object-name* に対応します。

clock　　クロック・セレクタを指定します。ここには 0 から 7 までの値を指定でき、オブジェクトの宣言文で指定したクロック・セレクタ値と異なっていても構いません。この指定はオプションです。省略して **io_select()** を実行すると、オブジェクト宣言のときに *clock* で指定したクロック値がセットされます。

タイマー/カウンタ I/O オブジェクトの中には、宣言シンタックスの中に *clock* 引数があり、クロックを指定できるようになっているものがあります。このクロック値は、**io_set_clock()**関数を使って別のクロック値に設定し直すことができます。**io_set_clock()**関数のシンタックスは以下のとおりです。

io_set_clock (*io-object-name*, *clock* **)**

io-object-name　　オブジェクト名を指定します。I/O 宣言部分の *io-object-name* に対応します。

clock　　クロック・セレクタ値を指定します。ここには 0 から 7 までの値を指定でき、オブジェクトの宣言文で指定したクロック・セレクタ値と異なっていても構いません。ただし、I/O オブジェクトの中には、全てのクロック値を指定すると正しく動作しないものもあります。特定の I/O オブジェクトについては『Neuron C Reference Guide』を参照してください。

多重化されたオブジェクトに対して **io_set_clock()**を使用すると、オブジェクト自身が選択された状態であるかどうかに関わらず、クロックが変更されます。

以下に、**io_select()**と **io_set_clock()**の使用例を示します。

```
IO_1 output pulsecount clock(3) out_pc;
IO_5 input period clock(2) in_period;
IO_6 input ontime clock(3) in_ontime;

when (reset)
{
   io_set_clock(out_pc, 5);
   io_select(in_ontime);
}

when (io_update_occurs(in_ontime))
{
   io_select(in_period, 3);
}
```

io_select()を使って I/O オブジェクトに新しいクロックを設定すると、このクロックは再び新しい値に設定されるまで有効です。同じ I/O オブジェクトに対して **io_select()**が呼び出されたときに *clock* 引数が指定されていなければ、宣言文で指定したクロック値にリセットされます。

io_select()関数を使って選択したことのない I/O オブジェクトに対して **io_in()** または **when** 節を使って入力計測を行うと「範囲外」（65,535）のデータ値が返され、**input_is_new** 変数と **io_update_occurs** イベントは FALSE のままになります。

io_select()呼び出しと Neuron のリセットによって新たに I/O オブジェクトが選択されると、不完全な計測値を避けるため、最初に計測された値は廃棄され

ます。ただし、**io_in()**呼び出しの前に **io_preserve_input** 関数を呼び出す場合は破棄されません。したがって、実際に **io_update_occurs** イベントが発生するのは、2 番目の計測が読み込まれたときということになります。**io_update_occurs** イベントか **input_is_new** 変数を使用して、**io_select()**の呼び出し後の実計測を確認してから値を取得するようにしてください。

多重化タイマー/カウンタ回路での **io_select()**の使用例を以下に示します。多重化 I/O オブジェクトでは、プログラム中で最後に宣言された I/O オブジェクトがリセット直後に有効になります。

コード例

```
// I/O Definitions
IO_5 input period mux clock (2) io_pcount_2;
IO_4 input period mux clock (2) io_pcount_1;


static long variable1, variable2;


// The following occurs only when the
// io_pcount_1 is selected
when (io_update_occurs(io_pcount_1))
{
   variable1 = input_value;
   io_select(io_pcount_2);
   // select next I/O object
}


// The following occurs only when the
// io_pcount_2 is selected
when (io_update_occurs(io_pcount_2))
{
   variable2 = input_value;
   io_select(io_pcount_1);
   // select next I/O object
}
```

次の例では、ピン **IO_5** 上でオン時間計測を行う **ontime** 入力オブジェクトと、ピン **IO_6** 上で周期時間計測を行う **period** 入力オブジェクトとの間で、タイマー/カウンタを多重化して使用しています。**ontime** 入力オブジェクトは広い範囲をカバーすることもあるので、この例では「自動範囲設定」を使用しています。入力計測値が指定した値の範囲外にあるときには、クロック値が 4 または 2 に切り替わります。クロックがこの 2 つの値のどちらかに設定されているため、**ontime** オブジェクトを再選択するときには変数を使ってクロックを設定しています。

コード例

```
unsigned long slope1Raw, cycleAValue;
int slope1Clock = 2;
IO_5 input ontime clock (2) ioSlope1;
IO_6 input period clock (1) ioCycleA;
// Following reset, the ioCycleA object is selected
// because it is the last object declared using the mux
```

```
 when (io_update_occurs(ioSlope1)) {
   if (input_value > 0x4000 && slope1Clock == 2) {
       // Range down (slower)
     slope1Clock = 4;
     io_set_clock(ioSlope1, 4);
   } else if (input_value < 0x4000 && slope1Clock == 4) {
     // Range up (faster)
      slope1Clock = 2;
     io_set_clock(ioSlope1, 2);
   } else {
     // Save the measured value, select the other object
     slope1Raw = input_value;
     io_select(ioCycleA);
   }
   // If auto-ranging has occurred, another measurement
   // will be made.  Otherwise, the ioCycleA object
   // will be measured next.
}


when (io_update_occurs(ioCycleA)) {
   cycleAValue = input_value;
   // Now select the ioSlope1 object,
   // using the current clock range computed above
   io_select(ioSlope1, slope1Clock);
}
```

デバイスの自己記録

アプリケーションには、デバイスについて記述したテキスト文字列を含めることができます。このテキスト文字列は、どんなネットワークツールからでもアクセスできます。ネットワーク・インテグレータがデバイスを設計、インストールする際、この文字列を参照して、それが正しいデバイスかどうかを確認します（このテキスト文字列は、デバイスの自己記録（SD）文字列に追記されます）。SD 文字列の一部は Neuron C コンパイラによって自動的に生成され、ここにはアプリケーションの機能ブロックによって実装される機能プロファイルが記録されます。次のコンパイラ指令を使用すると、SD 文字列にテキストを追加できます。コンパイラ指令については、『Neuron C Reference Guide』の「Compiler Directives」を参照してください。

#pragma set_node_sd_string *C-string-const*

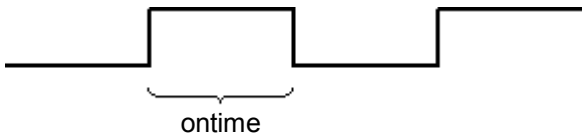
プログラム例

このセクションでは、Neuron C の機能、および適切なプログラミング・スタイルを理解してもらうため、次の 3 つのプログラムを紹介します。

1. サーモスタット用インターフェース
2. 簡単な照明調節インターフェース
3. 7セグメント LED 表示インターフェース

例１：サーモスタット用インターフェース

このサーモスタットは、**IO_4** ピンに入力される波形のパルス幅を調べることによって、サーミスタの抵抗値を計測します。I/O オブジェクトの宣言には、波形のオン時間を測るように **ontime** オブジェクトを使用します。オン時間から温度への変換は、$T = mx + b$ で行います。



また、この例ではダイヤルを使って温度を設定するため、**quadrature** 入力を生成するシャフト・エンコーダも使用します（図 2.3 参照）。**quadrature** 入力オブジェクト型は、**io_update_occurs** イベントと共に使用します。この入力オブジェクトの値は、最後の入力からの回転オフセットの変化を表しています。シャフト・エンコーダは、一般的に 360 度の回転に対して 16～256 カウントのオフセットを生成します。オフセットの計測結果が 0 以外であれば、**io_update_occurs** イベントは TRUE になります。以下に示すアプリケーションの **when（io_update_occurs…）**節のタスクは、**quadrature** 入力ダイヤルが前に計測した位置から動かされたときにだけ実行されます。

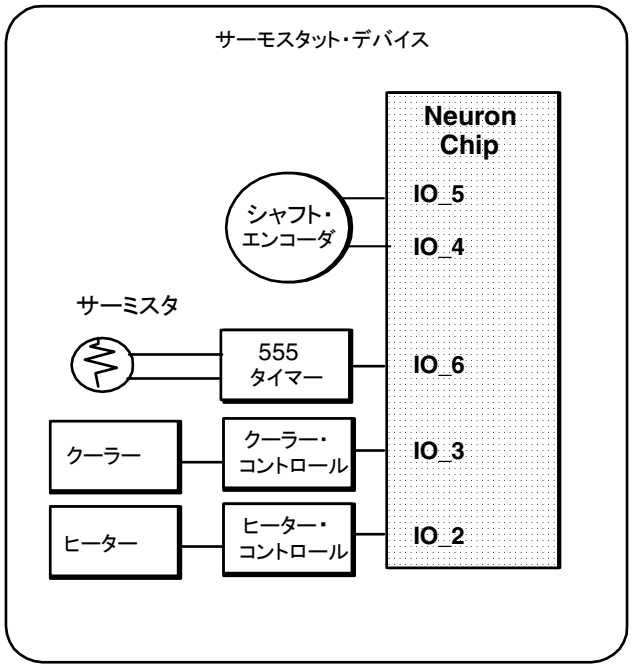


図 2.3　サーモスタット・デバイスの例

quadrature I/O オブジェクトを **io_changes** イベントに使用することはめったにありません。**io_changes** イベントは計測されたカウントに「変化」があったときにだけ TRUE になりますから、入力オブジェクトがある一定の割合で変化しているかぎり TRUE にはなりません（この例では、I/O オブジェクトの一般的な使用方法に重点をおいて説明するため、ネットワーク変数についての情報を割愛しています。ネットワーク変数については、第 3 章「ネットワーク変数を使ったデバイス間通信」で説明します）。

```
// THERMOS.NC -- LONWORKS thermostat device

// Uses a thermistor to measure temperature, and a
// quadrature encoder to enter setpoint.  Activates either
// heating or cooling equipment via bit outputs.


///////////////// Compiler Pragmas /////////////////
#pragma enable_io_pullups
                  // for quadrature input on IO_4 and IO_5


///////////////// Include Files /////////////////
#include <stdlib.h>
                  // for muldiv()

//////////////////// Timers /////////////////
stimer repeating tmCheckHeatOrCool;
                  // Automatically repeating timer


////////////////// Constants ///////////////////////
#define TEMP_DEG_F(t)  (((long)t - 32L) * 50 / 9 + 2740)
            // macro to convert degrees F to SNVT_temp

const SNVT_temp DESIRED_TEMP_MAX = TEMP_DEG_F(84);
const SNVT_temp DESIRED_TEMP_MIN = TEMP_DEG_F(56);
const SNVT_temp BAND_SIZE = 10;
// Guardband of +/- 1 deg C around desired temperature


///////////////// I/O Objects /////////////////////
IO_6 input ontime clock (1) invert ioTempRaw;
IO_4 input quadrature ioShaftIn;
IO_2 output bit ioHeatingOn = FALSE;
IO_3 output bit ioCoolingOn = FALSE;


///////////////// Global Variables ////////////////////
SNVT_temp newTemp    = TEMP_DEG_F(70);// init to 70 deg F
SNVT_temp desiredTemp = TEMP_DEG_F(70);


enum {
   OFF, HEATING, COOLING
} equip = OFF;   // current state of HVAC equipment


////////////////// Tasks /////////////////////
// I/O update task --
// read thermistor voltage-to-frequency converter

when (io_update_occurs(ioTempRaw)) {
   // An update occurs periodically as the ontime is
   // sampled. The new sample is placed in 'input_value.'
   // Calculation is performed using 32-bit intermediate
   // math, then the result stored as a SNVT_temp. The
   // input is scaled based on the temperature coefficient
   // of the thermistor.
   newTemp = muldiv(input_value, 25000, 9216) + 2562;
}
```

```
////////////////////////////////////////////////////
//  I/O update task -- read quadrature encoder
// A quadrature input is used as a dial to select a new
// temperature setting.

when (io_update_occurs(ioShaftIn)) {
// An update occurs for a quadrature I/O object when the
// accumulated offset is nonzero. The value is placed in
// 'input_value' by the io_update_occurs event.
    desiredTemp += input_value;     // Assumes no overflow
    desiredTemp = min(DESIRED_TEMP_MAX, desiredTemp);
    desiredTemp = max(DESIRED_TEMP_MIN, desiredTemp);
}

////////////////////////////////////////////////////
//  Timer task -- execute control algorithm
// A timer is used to decide periodically whether to
// activate heating or cooling. The temperature comparison
// is done only every five minutes to prevent cycling the
// equipment too frequently. There are two digital outputs:
// one for activating the heating equipment, and one for
// activating the cooling equipment.
when (timer_expires(tmCheckHeatOrCool)) {
    switch (equip) {
    case HEATING:
        if (newTemp > desiredTemp) {   // if too hot
            equip = OFF;              // turn off heater
            io_out(ioHeatingOn, FALSE);
        }
        break;


    case OFF:
        if (newTemp < desiredTemp - BAND_SIZE) {
            equip = HEATING;          // if too cold, then
            io_out(ioHeatingOn, TRUE);  // turn on heater

        } else if (newTemp > desiredTemp + BAND_SIZE) {
            equip = COOLING;          // if too hot, then
            io_out(ioCoolingOn, TRUE);  // turn on cooler
        }
        break;


    case COOLING:
        if (newTemp < desiredTemp) {   // if too cold
            equip = OFF;              // turn off cooler
            io_out(ioCoolingOn, FALSE);
        }
        break;
    }
}


////////////////////////////////////////////////////
//  Reset task -- Set the repeating timer to 300 seconds

when (reset) {
    tmCheckHeatOrCool = 300;  // 5 minutes, repeating
}
```

例２：簡単な照明調節インターフェース

以下の例は、簡単な照明調節用の Neuron C プログラムです。この例では 2 つの I/O オブジェクトを使用しています。1 つはランプの明るさを制御するトライアック制御回路への **triac** 出力オブジェクトで、もう 1 つは照明レベルを選ぶために使用するシャフト・エンコーダ用の **quadrature** 入力オブジェクトです（図 2.4 参照）。**triac** 出力オブジェクトの値が 1 のときを最大照度、値が 320 のときを最小照度（オフ）とします（ライン周波数は 60Hz）。電源が入ったときの初期値は完全なオフ状態（65535）です。

when 節の中で **io_update_occurs** イベントをどのように使用しているかに注目してください。このイベントの評価時に内部で **io_in()**呼び出しが起こり、タスクの中で **input_value** 組み込み関数を使って計測値にアクセスできるようになります。

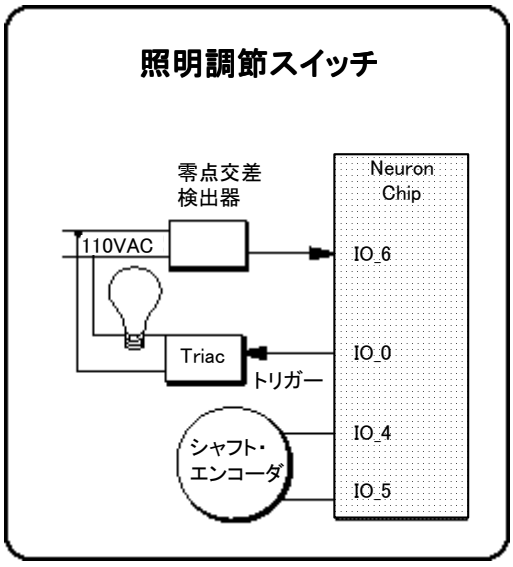


図 2.4　照明調節デバイスの例

```
// DIMMER.NC -- LONWORKS triac dimmer control

// Uses a triac output to control an incandescent lamp
// Uses a shaft encoder input to set desired lighting level


//////////////////// Compiler Pragmas /////////////////
#pragma enable_io_pullups

////////////////////// I/O Objects ////////////////////
IO_0 output triac pulse sync (IO_6) clock (6) ioLampTriac;
IO_4 input quadrature ioShaftIn;


////////////////////// Constants ////////////////////
// These constants are appropriate for 60Hz line frequency
const unsigned long MIN_BRIGHTNESS = 320;
const unsigned long MAX_BRIGHTNESS = 1;
```

```
//////////////////// Global Variables /////////////////
signed long currentBrightness;


/////////////////////////// Tasks ///////////////////////


//  Reset task -- turn the lamp off
when (reset) {
    io_out(ioLampTriac, MIN_BRIGHTNESS);
    currentBrightness = MIN_BRIGHTNESS;
}


//  I/O update task -- read quadrature input dial
//                  to select the light level
when (io_update_occurs(ioShaftIn)) {
    // An update occurs for a quadrature input
    // object when the accumulated offset is
    // nonzero.  The sample value is in
    // 'input_value'.  The value is subtracted
    // since a lower value means more light.

    currentBrightness -= input_value;

    // Look for underflow or overflow
    if (currentBrightness < MAX_BRIGHTNESS)
       currentBrightness = MAX_BRIGHTNESS;
    else if (currentBrightness > MIN_BRIGHTNESS)
       currentBrightness = MIN_BRIGHTNESS;

    // Change the triac setting to the
    // desired brightness level
    io_out(ioLampTriac, currentBrightness);
}
```

例３：７セグメント LED 表示インターフェース

次の例では、複数の文字を表示する装置（マルチキャラクタ・ディスプレイ）を **neurowire** ポートに接続しています。表示装置には、8 ビットの設定レジスタと 24 ビットの表示レジスタがあります。この機器構成は、以下のように定義できます。

```
IO_2 output bit ioEnable = 1;
IO_8 neurowire master select(IO_2) ioDisplay;
unsigned char displayReg[3];
unsigned char configReg;
      .
      .
      .
io_out(ioDisplay, &configReg, 8);
io_out(ioDisplay, displayReg, 24);
```

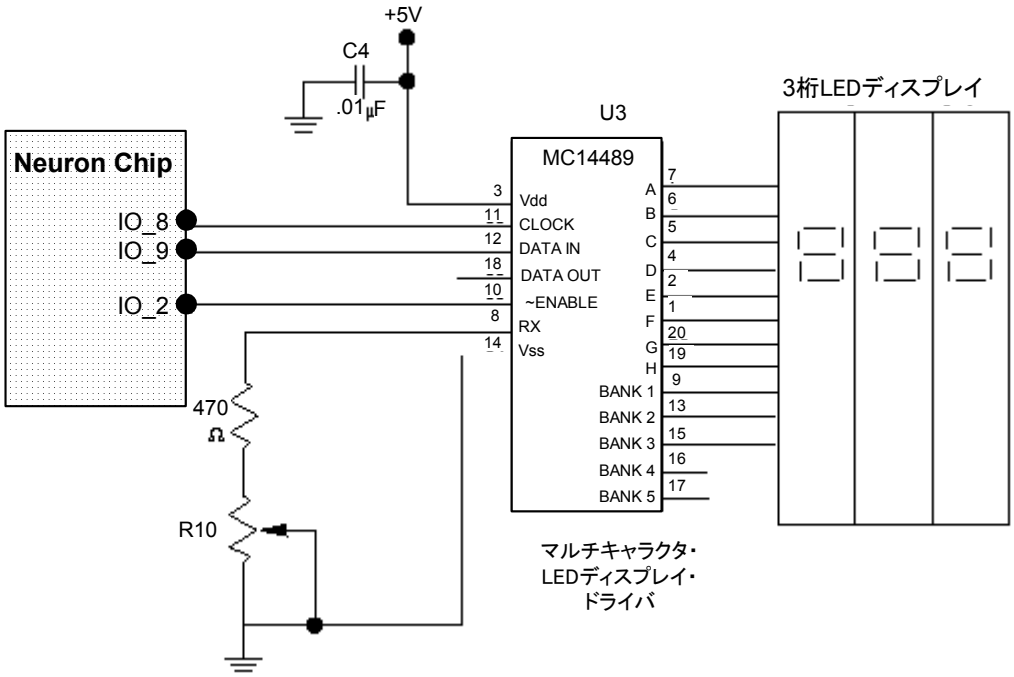


図 2.5　表示装置への Neurowire 接続

入力クロック周波数とタイマー精度

Neuron チップとスマート・トランシーバの入力クロックの周波数は、メーカーとバージョンによって、40MHz、20MHz、10MHz、5MHz、2.5MHz、1.25MHz、625 kHz のいずれかになります。次にリストするタイマーは、「固定タイマー」です。固定タイマーは、どの入力クロックが選択されていても、同じ時間間隔を維持します。ただし、入力クロックが遅くなれば、それだけタイマーの精度は低下します。後述する「スケール・タイマー」の時間長は、入力クロックに比例した長さになります。

固定タイマー

特に断わらないかぎり、本書に出てくるタイマーはすべて固定時間間隔のタイマーです。次に説明するタイマーはハードウエアに組み込まれているもので、その周期は Neuron チップまたはスマート・トランシーバの入力クロック周波数とは独立しています。ただし、これらのタイマーの精度は、Neuron チップまたはスマート・トランシーバの入力クロックの精度と周波数によって変化します。

- 先取りモードのタイムアウト・タイマー。
- パルスカウント入力タイマー。**pulsecount** 入力オブジェクトのカウント間隔を決定します。この間隔は $2^{23}/10^7$ 秒（約 0.8388608 秒）です。
- トライアック・パルス・タイマー。**triac** 出力オブジェクトのパルスを生成します。

以下に説明するタイマーはソフトウエアで実装しているもので、その周期は Neuron チップまたはスマート・トランシーバの入力クロックとは独立しています。これらのタイマーの精度については、次のセクションで説明します。

- アプリケーション秒タイマー（Neuron C プログラムの中で **stimer** で宣言されたもの）。
- アプリケーション・ミリ秒タイマー（Neuron C プログラムの中で **mtimer** で宣言されたもの）。

スケール・タイマーと I/O オブジェクト

入力クロックを利用するタイマーと I/O オブジェクトは、入力クロックに比例して変化します。例えば、2400bps に設定されたシリアルオブジェクトを 2.5MHz（1/4 の速度）の発振器で使用すると、実際の動作は 600bps になります。以下のタイマーは、入力クロックによって周期が変わります。

- ビットシフト・クロック
- neurowire マスター・クロック
- シリアル・クロック
- ウォッチドッグ・タイマー

注意： 構成可能 EEPROM 書き込みタイマーの精度は、入力クロックの速度によって異なります。詳しくは、本章で後述する「EEPROM 書き込みタイマー」のセクションを参照してください。

ソフトウエア・タイマーの精度計算

ミリ秒タイマーの精度

以下に、ミリ秒タイマーの精度範囲の計算式を示します。精度は、タイマーがセットされてからシステムがアプリケーションにイベントを送るまでの最短時間（**L**）と最長時間（**H**）で決まります。後述するように、**L** と **H** は期待時間（**E**）から計算されます。

時間切れのイベントを検知するために付加される時間遅れは、アプリケーションに依存するため、これらの計算式には含まれていません。例えばアプリケーションが、ある **when** 節のタスクを実行している間にイベントが知らされても、タスクの実行が完了してアプリケーションの制御がスケジューラに戻るまでそのイベントは検知されません。

注意： あるイベントが Neuron ファームウエアによって呼び出されると、スケジューラや他のイベント（例：**io_changes**、**nv_update_occurs**）から見える状態になります。

10MHz クロックの場合

次の計算式で使用している *floor()*関数は、引数に指定された値以下で最大の整数を返します。例えば、*floor*(3.3)=3、*floor*(3.0)=3 となります。10MHz の場合、ミリ秒タイマーの期待時間は次のようになります。

$$E = .8192 * floor((D/.82) + 1)$$

ここで、*D* はタイマーで指定した時間です。例えば、100 ミリ秒のタイムアウトの場合、*E* の値は 99.94 ミリ秒になります。

10MHz での最短時間は、次のようになります。

$$L = E - 12\text{ms}$$

最長時間は、次のとおりです。

$$H = E + 12\text{ms}$$

他のクロック・スピードの場合

次の計算式を使うと、他の入力クロックレートを選択した場合のミリ秒タイマーの精度を求めることができます。これらの計算式に出てくる *S* は、次の表のように入力クロック・スピードによって変わります。

S=入力クロックレート	
0.25	40 MHz
0.5	20 MHz
1	10 MHz
2	5 MHz
4	2.5 MHz
8	1.25 MHz
16	625 kHz

E = .8192 * floor ((floor(D/S)*S)/.82) +1)

E を決定する要素には 2 つあります。1 つは入力クロック・スピードが遅くなるのに伴って入力クロックの単位が粗くなることです。例えば、1/16 のスピードの場合では、ミリ秒の単位時間は 16 ミリ秒になります（クロックの刻みが 16 ミリ秒毎）。もう 1 つの要素は、ハードウエアのクロック刻みが 819.2 マイクロ秒間隔でも、ソフトウエアはそれを 820 マイクロ秒として扱うことです。つまり、タイマーの時間長が実際には指定された期間の 0.999 倍になってしまいます。

例えば、2.5MHz の場合にタイムアウト時間を 99 ミリ秒と指定しても、実際の期待時間は 96.67 ミリ秒になります。

最短時間と最長時間を計算するための正確な計算式は、次のようになります。

$L = E - (11*S + 1)$

$H = E + (11*S + 1)$

2.5MHz クロックでタイムアウト時間に 99 ミリ秒を指定した場合、最長時間が 141.67 ミリ秒、最短時間が 51.67 ミリ秒になります。

注意： 上の計算式の「11」という数字は、通常考えられる最悪の状態を基にしたものです。最悪の場合、つまりタイマー、ネットワーク変数、アドレスなどを最大限利用しているときには、この数が最高 32 になります。

さらに「ネットワーク管理遅延（NMD）」の影響で、最長時間はもっと長くなる可能性があります。NMD は、ネットワーク管理コマンド処理によって引き起こされる遅延です。通常この要素は 0 です。しかし、ネットワーク管理メッセージを処理するデバイスでは、タイムアウトの上限が無視できないほど増加することがあります。例えば、デバイスにドメインを 1 つ追加すると、300 ミリ秒から(300+838*S)ミリ秒の NMD が発生します。通常、この種のネットワーク管理操作は何度も起こるものではありません。ネットワーク管理コマンドを続けて送信する前に、可能であればデバイスをオフラインにしてみてください。

イベントの時間長を計測するため、イベントの前後でタイマーがポーリングされます。ただし、イベントの時間長が 50 ミリ秒よりも短ければ、**get_tick_count()**関数を使用してください（『Neuron C Reference Guide』を参照してください）。

繰り返しタイマー

繰り返しタイマーには、D と E の差異によって発生する蓄積ドリフトがあります。繰り返しタイマーの N 番目のタイムアウトは、次の計算式で求められる L_R と H_R の範囲で発生します。

$E_R = E * N$

および

$L_R = E_R - (11*S + 1)$

$H_R = E_R + (11*S + 1)$

繰り返しタイマーでは、次の条件が成立すると、中間にあるタイムアウト・イベントが失われることに注意してください。

$$\text{abs}(A_R - E_R) \geq E$$

$$E_R - A_R > E$$

ここで、A_Rは繰り返しタイマーの実時間長です。

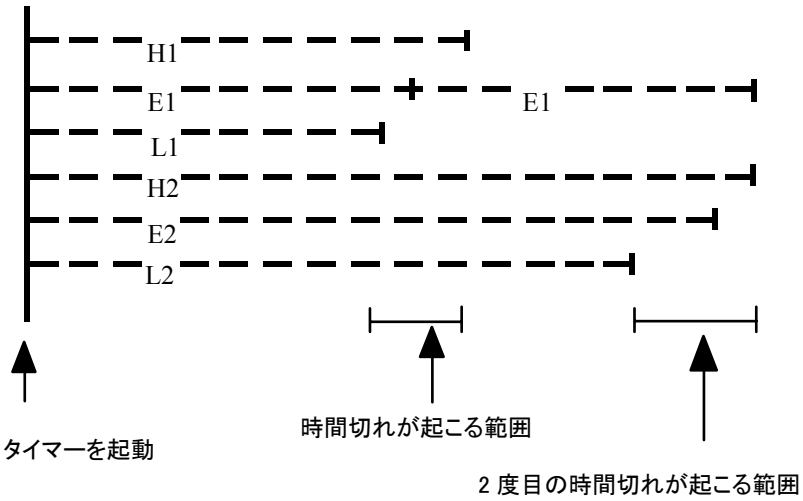


図 2.6　タイムアウト・イベントの期待時間、最短時間、最長時間

秒タイマーの精度

秒タイマーは、1 秒タイマーに依存しています。1 秒タイマーの仕組みは、すでに説明したミリ秒タイマーの仕組みと基本的に同じです。*D* という時間長の 1 秒タイマーは、*D*-1 秒から *D* 秒までの範囲でタイムアウトします。ここでいう「秒」は、ミリ秒タイマーの *L* と *H* の計算式を使って、1001 ミリ秒として定義したものです。

例えば、625kHz での「1 秒」は 991.23 ミリ秒となります。したがって、10 秒タイマーは 8.74 秒から 10.09 秒までの範囲でタイムアウトします。

繰り返し 1 秒タイマーでは、最初のタイムアウトが *D*-1 秒から *D* 秒までの範囲で発生し、その次からのタイムアウトは *D* 秒毎に発生します。つまり、10 秒の繰り返しタイマーの 5 番目のタイムアウトは、48.39 秒から 49.74 秒の間で発生します。

遅延関数

指定された時間だけアプリケーションの実行を中断し、アプリケーションが直接タイミングを調整するための関数が 2 つあります。これら 2 つの関数は、プログラムの中でタイミングを取るための簡潔な方法を提供します。

delay()

scaled_delay()

delay()関数は、入力クロック・スピードとは関係ない固定長の遅れを発生します。この関数は、**wink** 機能や I/O デバウンシングと一緒に使用します。プロトタイプは次のとおりです。

void delay (unsigned long *count*);

count	1 から 33,333 までの値を指定します。遅延時間を求める計算式については、『Neuron C Reference Guide』を参照してください。33,334..65,535 の範囲の値を指定することもできますが、ウォッチドッグ・タイマーがリセットされることに注意してください。

コード例

```
when (io_changes(io_switch))
{
   delay(400);  // wait 10msec for debounce
        .
        .
        .
}
```

scaled_delay()関数は、入力クロック・スピードと比例する遅延時間を発生します。

scaled_delay()関数のシンタックスは以下のとおりです。

void scaled_delay (unsigned long *count*);

count	1 から 33,333 までの値を指定します。遅延時間を求める計算式については、『Neuron C Reference Guide』を参照してください。

EEPROM 書き込みタイマー

構成可能 EEPROM 書き込みタイマーは、入力クロック・スピードと共に精度が落ちていきます。*n* ミリ秒のタイムアウトの精度は、次の式で求めることができます。

duration = *n* * delay(43)

例えば、625kHz での 20 ミリ秒の EEPROM の書き込みは、実際には 55.2 ミリ秒かかることになります。.

3

ネットワーク変数を使った デバイス間通信

この章では、ネットワーク変数を使って LONWORKS デバイスが互いに通信する方法について説明します。ここでは、ネットワーク変数の宣言や別々のデバイスにあるネットワーク変数をどのように接続するかについて説明しています。さらに、Sync 型ネットワーク変数の使用方法、ネットワーク変数のポーリング処理過程、認証ネットワーク変数についても説明します。

はじめに

ある LONWORKS デバイスが他の LONWORKS デバイスと通信するには、ネットワーク変数またはアプリケーション・メッセージを使用します。本章では、ネットワーク変数に焦点を当てます。ネットワーク変数は、相互運用可能なオープン・インターフェースを提供し、プログラミングとインストールを簡単にし、プログラムのメモリ使用量を抑えます。そのため、多くのプログラムではネットワーク変数を使って通信し、必要に応じてアプリケーション・メッセージを使用します。詳細については、第 6 章「アプリケーション・メッセージを使ったデバイス間通信」を参照してください。本書では、2 つの方法をそれぞれ別の章で説明しますが、1 つのプログラムの中でネットワーク変数とアプリケーション・メッセージの両方を使用することもできます。

本章は、以下の内容で構成されています。

- 「概要」: ネットワーク変数を読み書きするときのデバイスの動作について、ネットワーク変数の宣言方法を含めて概説します。また、別々のデバイス上にあるネットワーク変数がどのように接続するかについても説明しています。
- 「ネットワーク変数の宣言」：ネットワーク変数の宣言のシンタックスと、関連する概念について説明します。
- 「ネットワーク変数の接続」：ネットワーク変数読み込みデバイスが書き込みデバイスに接続し、値を読み取る方法について説明します（この手順については、第 1 章「概要」でも簡単に説明しました）。
- 「ネットワーク変数のイベント」: ネットワーク変数に関するスケジュール・イベントについて説明します。ここで説明するイベントは、**nv_update_completes**、**nv_update_fails**、**nv_update_occurs**、**nv_update_succeeds** の 4 つです。
- 「Sync 型ネットワーク変数」：SYNC 型ネットワーク変数の動作について説明します。
- 「ネットワーク変数の完了イベントの処理」：2 種類のモードの完了イベント・チェック、およびアプリケーション・プログラムでこれらの異なるモードを使用するためのガイドラインについて説明します。
- 「ネットワーク変数のポーリング」：ネットワーク変数の最新データを得るため、読み込みデバイスから書き込みデバイスをポーリングする方法について説明します。
- 「ネットワーク変数の明示的伝達」：Neuron ファームウエア・スケジューラによるネットワーク変数の更新の自動伝達を許可せず、アプリケーション・プログラムがネットワーク変数の伝達を明示的に制御する方法について説明します。
- 「ネットワーク変数のモニター」：デバイスをモニターする上での特別な考慮事項について説明します。
- 「認証機能」：ネットワーク変数の認証機能を使って、ネットワークのセキュリティを向上させる方法について説明します。認証機能を使用すると、読み込みデバイスのネットワーク変数の値を更新しようとしている書き込みデバイスが正しいデバイスであるかどうかを識別できます。また、認められていないデバイス構成を防ぐのにも認証機能を使用します。
- 「型の変更が可能なネットワーク変数」：インストール時に型の変更が可能なネットワーク変数の実装方法について説明します。

概要

第 1 章「概要」で説明したように、ネットワーク変数とはネットワーク上で複数のデバイスと接続するオブジェクトです。Neuron チップまたはスマート・トランシーバ上で実行する Neuron C アプリケーション・プログラムでは、最大 62 のネットワーク変数を宣言することができます。ホスト・アプリケーションでは、さらに多くのネットワーク変数を宣言できます。ホスト・アプリケーションについては、この後で詳しく説明します。

ネットワーク変数は、各 Neuron チップまたはスマート・トランシーバ上で動作するプログラムの中で定義されます。例えば、**nv_lamp_state** という名前の 1 つのネットワーク変数を持つ照明ランプ・プログラムを考えてみてください（図 3.1 参照）。また、**nv_switch_state** という名前の 1 つのネットワーク変数を持つスイッチ・プログラムを考えてみてください。3 つのランプ・デバイスにはそれぞれ同じランプ・プログラムがインストールされ、以下の図の 2 つのスイッチ・デバイスにもそれぞれ同じスイッチ・プログラムがインストールされます。

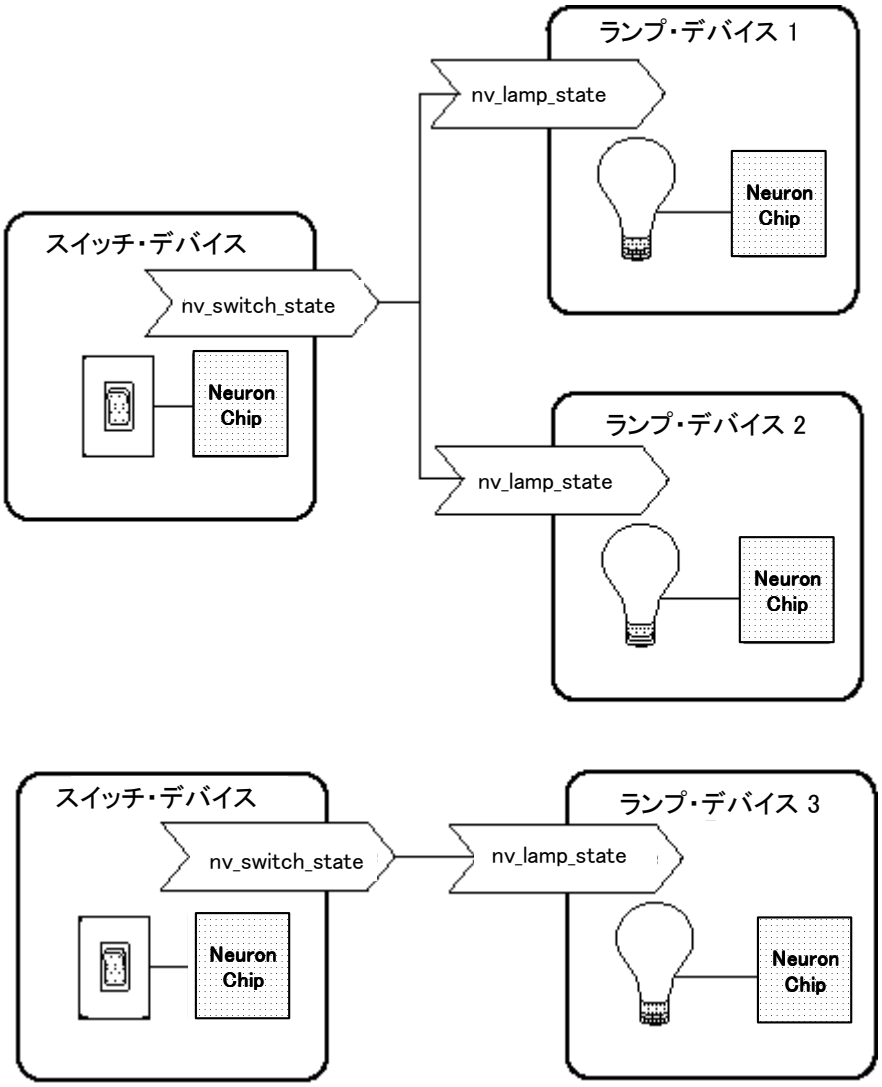


図 3.1　5 つのデバイスを使用したネットワーク開発例

これら 2 つのネットワーク変数は、それぞれのプログラムで次のように宣言されています。

```
network output SNVT_switch nv_switch_state;

network input SNVT_switch nv_lamp_state;
```

書き込みデバイスと読み込みデバイスの動作

書き込みデバイスとは、ネットワーク変数の値を変えることのできるデバイスのことです。ネットワーク変数に変更があると、接続されているすべての読み込みデバイス中のネットワーク変数は、その変更に応じた値に更新されます。一般的に読み込みデバイスは、ネットワーク変数のコピーを読み込むだけで書き込むことはありません。読み込みデバイスで入力ネットワーク変数の宣言に初期値を設定したり、プログラムの中で変数のローカル・コピーを修正することはできますが、どちらの場合も新しい値は他のデバイスに伝達されません。

書き込みデバイスも自分のネットワーク変数の最新のコピーから値を読み込むことがあります。ただし、自分が最後に書き込んだ値を見るというだけであり、同じネットワーク変数への書き込みデバイスが 2 つあっても、お互いの値を変更することはできません。

書き込みデバイスが値を出力ネットワーク変数に書き込むと、その変数の読み込みデバイスとして設定されているすべてのデバイスに対して新しい値を通知する LONWORKS メッセージを Neuron ファームウエアが送信します。デフォルトでは、確認応答付き（ACKD）サービスを使ったメッセージが送信されます。すべての読み込みデバイスが更新情報を同時に受け取るとは限らない点に注意してください。例えば、ネットワーク・アプリケーションを設計するときには、更新メッセージ受信の失敗や時間遅れを考慮して設計しなければなりません。

注意： ここでは、「書き込みデバイス」と「読み込みデバイス」という用語を使っています。書き込みデバイスとは、ある特定のネットワーク変数（出力ネットワーク変数）に対して書き込むデバイスのことです。また、読み込みデバイスとは、ある特定のネットワーク変数（入力ネットワーク変数）を読み込むものです。ほとんどの場合、デバイスのプログラム中には入力ネットワーク変数と出力ネットワーク変数の両方が宣言されています。したがって、1 つのデバイスがネットワーク変数に応じて、書き込みデバイスになったり、読み込みデバイスになったりします。

更新発生時の処理

読み込みデバイスがネットワーク変数の新しい値を受け取っても、すぐに受信やメッセージの処理を開始するわけではありません。同様に、出力ネットワーク変数に新しい値が代入されても、すぐにメッセージが送信されるわけでもありません。これらの更新は、アプリケーション・プログラム中のクリティカル・セクションが終了したところで実行されます。言い換えると、ネットワーク変数の更新が起こらないようになっているアプリケーション・プログラム部分が「クリティカル・セクション」です。

タスクはクリティカル・セクションの一例です。タスクが一度開始されると、完了するまで中断されずに実行します。ネットワーク変数の更新情報を受信したり更新要求があったときには、スケジューラが各クリティカル・セクションが終わったところでポストします。**post_events()**関数を使えば、1 つのタスクの中を複数のクリティカル・セクションに分割することもできます。**post_events()**関数を使用すると、ネットワーク変数の更新を他のデバイスに送ったり、受信したネットワーク変数の更新を処理したりするための「クリティカル・セクションの境界」を作り出します。そのため処理能力が向上し、応答時間がより速くなります。**post_events()**について詳しくは、第 7 章「追加機能」を参照してください。

ネットワーク変数の宣言

ネットワーク変数を宣言するためのシンタックスは以下のとおりです。最初の形式の宣言は、単純なネットワーク変数用で、2 番目の形式はネットワーク変数の配列用です。

network input | **output** [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier*
[= *initial-value*] [*nv-property-list*] **;**

network input | **output** [*netvar-modifier*] [*class*] *type*
[*connection-info*] *identifier* **[***array-bound***]**
[= *initializer-list*] [*nv-property-list*] **;**

注意： 「*array_bound*」を囲っているブラケット（角カッコ）は、省略可能を意味するものではありません。このブラケットは必須のもので、プログラムの一部になっている必要があります。

1 つの Neuron C プログラムまたは ShortStack™ Micro Server を使用したアプリケーション内で、1 つのデバイスに対して宣言できるネットワーク変数の数は、最大 62 個（配列要素を含む）です。LONWORKS ネットワーク・インターフェースおよびネットワーク接続されたホストプロセッサを使用しているときには、最大 4,096 個までのネットワーク変数を宣言できます。詳しくは、『LNS® Programmer's Guide』および『Host Application Programmer's Guide』を参照してください。

上に示したシンタックスの 2 番目の形式は、ネットワーク変数配列の宣言ですが、一次元にしかならないことに注意してください。また、*array-bound* は定数でなければなりません。配列の各要素は、イベント、ネットワークへの送信といった目的に対しては、独立したネットワーク変数として扱われます。したがって、デバイス上では各要素が別々のネットワーク変数であるとして数えられ、最大数を越えていないかどうかがチェックされます。配列の各要素は、「個別にバインド可能な」ネットワーク変数です。

デバイスの設計が完成したら、異なるデバイス上にある出力ネットワーク変数と入力ネットワーク変数とを接続します。これについては、後述の「ネットワーク変数の接続」のセクションで説明します。接続後、その情報を使用してネットワーク管理ツールが適切なネットワーク・アドレスを生成します。ここで生成されたアドレスはデバイスにダウンロードされ、これによって書

き込みデバイスの送信した更新情報が読み込みデバイスとして設定されているすべてのデバイスに届いたことを確認できます。

照明ランプとスイッチの例では、1 列目にある出力ネットワーク変数が 2 列目の入力ネットワーク変数に接続します。

出力 (デバイス/変数名)	入力 (デバイス/変数名)
switch1/nv_switch_state	lamp1/nv_lamp_state lamp2/nv_lamp_state
switch2/nv_switch_state	lamp3/nv_lamp_state

ネットワーク変数修飾子

ネットワーク変数の宣言には、次のオプション修飾子があります。

sync|synchronized

この修飾子を付けて宣言すると、ネットワーク変数に代入されたすべての値が代入の順番通りに伝達されます。ただし、同じ Sync 型ネットワーク変数が 1 つのクリティカル・セクションで何度か更新されたときには、最新値だけが送信されます。

このキーワードを指定しないでネットワーク変数を宣言したときには、スケジューラがすべての代入値を伝達するとは限りません。例えば、新しい値を伝達するよりも頻繁にネットワーク変数が更新されたり、更新イベントの処理よりも頻繁にネットワーク変数が更新されたりすると、スケジューラはどこか途中の値を捨ててしまいます。ただし、ネットワーク変数の最新値は、デバイスがリセットされない限り捨てられることはありません。Sync 型ネットワーク変数については、後述の「Sync 型ネットワーク変数」のセクションを参照してください。

polled

ネットワーク変数の読み込みデバイスからのポーリング要求があったときにのみ、その値を送信します。このキーワードを指定しないでネットワーク変数を宣言したときには、変数に値が代入される度に最新値が伝達されます（書き込みデバイスの出力ネットワーク変数に **polled** の指定があるかどうかにかかわらず、読み込みデバイスは書き込みデバイスに対して出力をポーリングできます）。また、**polled** を指定して宣言した出力ネットワーク変数の値は、**propagate()**関数（『Neuron C Reference Guide』の「Functions」の章を参照してください）を使ってネットワーク上に送信することもできます。

注意： **polled** キーワードは出力ネットワーク変数のみで用います。ただし、ShortStack アプリケーションの開発中に Neuron C のモデル・ファイルとして使用されるプログラムでは、このキーワードを使用できます。詳細については、『ShortStack User's Guide』を参照してください。

changeable_type　ネットワーク管理ツールによってネットワーク変数の型を変更できることを宣言します。この機能の使用については、本章の「型変更可能なネットワーク変数」を参照してください。**changeable_type** 修飾子はネットワーク変数の宣言に 1 回だけ使用でき、**sync** 修飾子または **polled** 修飾子のどちらかを使用する場合には、その後に指定する必要があります。

sd_string (*C-string-const* **)**
ネットワーク変数の自己記録文字列（Neuron C コンパイラが自動的に生成する自己記録テキストを含む、最大 1023 バイトの文字列）を設定します。ANSI C で定義されている連結文字列定数の機能を使用できます。この修飾子は、ネットワーク変数の宣言に 1 回だけ使用できます。**sync**、**polled**、**changeable_type** のいずれかの修飾子を使用する場合、**sd_string** 修飾子はその後に指定してください。

ネットワーク変数クラス

ネットワーク変数は、Neuron C の記憶クラスの 1 つです。ネットワーク変数は、以下の記憶クラスと組み合わせることもできます。

const　アプリケーション・プログラムから変更できないネットワーク変数を指定します。出力ネットワーク変数の宣言に **const** を指定すると、その変数は ROM または EEPROM に保存されます。また入力ネットワーク変数の宣言に **const** を指定した場合には、RAM に保存されます。

const を出力ネットワーク変数と共に使用する場合は、**polled** 修飾子（上記参照）についても考慮してください。

eeprom　値を EEPROM（またはフラッシュ・メモリ）に保存することをアプリケーションから指示するためのキーワードです。EEPROM やフラッシュ・メモリに保存されたネットワーク変数の値は、停電があっても保存されます。ただし、**eeprom** ネットワーク変数には、変更に関する制約があることに注意してください。**eeprom** クラスのネットワーク変数に指定した初期化子は、プログラムがロードされたときに実行されます。そして、リセットされてもこれらの変数は初期化されず、アプリケーション・イメージが再ロードされるまで初期化されることはありません。

注意：　Neuron コアの EEPROM は、通常少なくとも 10,000 回のデータ削除と書き込みを繰り返してもデータが失われないように設計されています。これは Neuron チップまたはスマート・トランシーバのモデルによって異なる場合があるため、正確な仕様については該当するデータ・ブックを参照してください。外部フラッシュ・メモリの仕様については、そのメーカーのデータシート等を参照してください。

config　　他のデバイスからしか変更できないような EEPROM 中の **const** ネットワーク変数を指定します。このネットワーク変数クラスは、ネットワーク管理ツールやネットワーク・コントローラがアプリケーションの設定をするために使用します。**config** 修飾子は古いアプリケーションをサポートするためのものであり、完全に管理された構成プロパティではありません。Neuron C バージョン 2 では **config_prop** キーワード（以下を参照）を使用して、完全に管理された構成プロパティを宣言します。構成プロパティの宣言と仕様については、第 4 章「構成プロパティを使ったデバイス動作の構成」を参照してください。

注意：　**config** キーワードは入力ネットワーク変数のみで用います。

config_prop | **cp**　　ネットワーク変数が、Neuron C バージョン 2 で完全に管理される構成プロパティであることを宣言します。構成プロパティの宣言と仕様については第 4 章で説明しています。

ネットワーク変数宣言にクラス指定がなければ、そのネットワーク変数はグローバル変数になります。グローバル変数は Neuron チップの RAM 領域に保存されるので、電源が切れると値は消失してしまいます。

ネットワーク変数のコネクション情報

connection-info　　ネットワーク変数の接続に関するオプション属性を指定するためのフィールドです。ここには、次のようなオプション・フィールドがあります。

bind_info (
　　[offline]
　　[unackd | unackd_rpt | ackd　[(config | nonconfig)]]
　　[authenticated | nonauthenticated　[(config | nonconfig)]]
　　[priority | nonpriority　[(config | nonconfig)]]
　　[rate_est (*const-expr*)]
　　[max_rate_est (*const-expr*)]
　　)

各フィールドについては、『Neuron C Reference Guide』にある「コネクション情報」のセクションで説明します。フィールドの指定順序には決まりはありません。これらのコネクション情報は、デバイスがインストールされた後でネットワーク管理ツールを使って書き換えることができます。ただし、**nonconfig** オプションが指定されているネットワーク変数のコネクション情報は書き換えられません。

ネットワーク変数の初期化子

initial-value

または

initializer-list

ネットワーク変数に対する初期化子（または初期化子のリスト）を指定します。ここに指定した初期値は、**eeprom** や **config** クラスのネットワーク変数ではアプリケーション・イメージの一部としてロードされます。**const**、**eeprom**、**config**、**config_prop** 以外のクラスのネットワーク変数では、電源を入れた直後やリセットのときにもこの初期値が設定されます。ネットワーク変数は、特に入力ネットワーク変数は、適切なデフォルト値に初期化するようにしてください。

コード例

```
network input SNVT_temp nv_temp = 2960;   // 22 C, 72 F
```

デバイスがリセットされる場合、変数がネットワークから更新される前に、リセット後の計算に初期値を使用しても問題がないこと、およびこれらの計算によってデバイスが危険な状態やエラー状態を引き起こすことがないことを考慮して、初期値を選択してください。デフォルトの初期化値は 0 です。リセット時にゼロに初期化すると、コード領域とデバイスの起動実行時間を節約できるという利点があります。ネットワーク変数が **input** でも **output** でも、初期化値はネットワークに伝達されません。

ネットワーク変数型

ネットワーク変数型には、2 つの目的があります。1 つは、変数が正しく使用されているかをコンパイル時に確認できます。もう 1 つは、デバイスをネットワークにインストールしたときのネットワーク変数の接続が正しいかどうかを型の整合性から確認できます。ネットワーク変数には、第 1 章「概要」で説明した変数型のうちポインタ以外の変数型を使用できます。型は次のとおりです。

- 「標準のネットワーク変数型（SNVT）」：SNVT は、摂氏、ボルト、メートルなどの標準数量のデータ・エンコーディング、スケーリング、および単位を定義する標準型です。各 SNVT には *SNVT index*「SNVT インデックス」と呼ばれる一意の識別があります。SNVT の定義をすべて表示するには、NodeBuilder リソース・エディタを使用します。詳細は『NodeBuilder User's Guide』を参照してください。また、Echelon NodeBuilder ソフトウエアのプログラム・フォルダにある NodeBuilder ツールに付属の『LONMARK SNVT and SCPT Guide』にも SNVT の定義を掲載しています。
- 「ユーザ・ネットワーク変数型（UNVT）」：UNVT は、NodeBuilder リソース・エディタを使用して定義する型です。詳細は『NodeBuilder User's Guide』を参照してください。
- **typedef**：Neuron C には例えば次のような定義済みの型が用意されています。

  typedef enum {FALSE, TRUE} boolean;

  また、他の型定義を定義して、ネットワーク変数型として使用することもできます。

  しかし相互運用性のあるデバイスとするためには、**typedef** の代わりに、リソース・ファイルで定義されている SNVT および UNVT を使用するようにしてください。

- 第 1 章で説明した型のうち、ポインタ以外のすべての変数型を使用できます。使用できる変数型は次のとおりです。

  [signed] long int
  unsigned long int
  signed char
  [unsigned] char
  [signed] [short] int
  unsigned [short] int
  enum

  上記の変数型を使用した構造体および共用体。

  しかし相互運用性のあるデバイスとするためには、これらのベース型の代わりに、リソース・ファイルで定義されている SNVT および UNVT を使用するようにしてください。

- 上記の変数型を使用した一次元配列（要素数は最大 62）。

ネットワーク変数が構造体になっているとき、ネットワーク変数に書き込みデバイスからの変更があると、その変更が部分的なものでも全体に渡るものでも、次のクリティカル・セクションの後で読み込みデバイスの構造体全体が更新されます。

一次元配列を使ってネットワーク変数を宣言できます。この場合、配列の各要素は個々にバインド可能なネットワーク変数になります。詳しくは、『Neuron C Reference Guide』にある **poll()**関数、組み込み **nv_array_index** 変数、および **nv_update_completes**、**nv_update_fails**、**nv_update_occurs**、**nv_update_succeeds** の各イベントを参照してください。

配列になっているネットワーク変数の要素が書き込みデバイスによって変更されると、次のクリティカル・セクションの後では変更要素だけが更新されます。

ネットワーク変数の最大サイズは 31 バイトです。ネットワーク変数配列の場合、各要素が 31 バイトに制限されます。

Neuron C コンパイラは、アプリケーション・イメージの中に SNVT として宣言されている全ネットワーク変数の SNVT インデックスを格納します。オプションとして、全ネットワーク変数のネットワーク変数名も含めることができます。ネットワーク変数名は、デバイスのインターフェース・ファイルに常に含まれていますが、デバイス・インターフェース・ファイルを紛失してしまい、デバイスをインストールする必要が生じたときに上記の情報が役に立ちます。これらのオプションは、次のコンパイラ指令を使用して制御できます。詳細は『Neuron C Reference Guide』の「Compiler Directives」の章を参照してください。

#pragma disable_snvt_si

#pragma enable_sd_nv_names

ネットワーク変数宣言の例

以下は、ネットワーク変数宣言の例です。

```
network input SNVT_temp temp_set_point;
network output SNVT_switch primary_heater;
network output int current_temp;
```

以下は、優先ネットワーク変数宣言の例です。

```
network output SNVT_alarm bind_info(priority)
            fire_alarm;
network input boolean bind_info (priority(nonconfig))
            fire_alarm;
```

以下は、確認応答なし（unackd）サービスを使ったネットワーク変数宣言の例です。

```
network output SNVT_lev_cont bind_info(unackd)
            control_dial;
```

このネットワーク変数宣言では、control_dial のチューニング時に多数のメッセージが生成されるため、各メッセージに対する確認をいちいち受け取る必要がないか、受け取りたくないことを想定し、確認応答なしサービスを使用しています。さらに、このアプリケーションではメッセージのごく一部が受信されなかったとしてもそれほど問題にはならないという理由もあります。

ネットワーク変数の接続

ネットワーク変数の「接続（バインド）」は、デバイスの Neuron C アプリケーションとは独立に管理できます。ネットワーク変数を接続するには、「バインドツール」と呼ばれるネットワーク管理ツールを使用します。バインドツールは、LonMaker 統合ツールまたはその他のネットワーク管理ツールの中に組み込まれています。

最初に、バインドツールは共通のネットワーク変数を共有しているすべてのデバイスを探します。次に、各ネットワーク変数について正しいデバイス間の情報の流れを確立できるように、該当する全デバイスにアドレスを割り当てます。

is_bound( )関数の使用

Neuron C アプリケーションでは **is_bound()**関数を呼び出して、ネットワーク変数がネットワーク管理ツールによって接続されているかどうかを調べることができます。これによって、未接続のネットワーク変数が不必要に処理されることを避けることができます。未接続の出力ネットワーク変数が更新されると、実際には更新は行われませんが、**nv_update_succeeds** イベントは TRUE になります（第 6 章「アプリケーション・メッセージを使ったデバイス間通信」の「ネットワーク変数の完了イベントの処理」のセクションも参照してください）。この関数を使用すれば、出力ネットワーク変数が接続しているかどうかで結果が変わってしまうようなコードの実行を防ぐことができます。また、入力ネットワーク変数を使う前にその変数が接続されているかどうか（つまり正しい値かどうか）を確認するために **is_bound()**関数を使うこともできます。コード例を以下に示します。

```
network output SNVT_lev_switch heater_2;
void turn_on_heater_2(void) {
      // turn on secondary heater if one is connected
      if (is_bound(heater_2))
            heater_2 = ST_ON;
}
```

ネットワーク変数のイベント

イベント・スケジューリングのメカニズムと定義済みイベントについては第2章「シングル・デバイスでの機能」で説明しました。ネットワーク変数に関する定義済みイベントには、次の4つがあります。

nv_update_completes [(*network-var-reference*)]
nv_update_fails [(*network-var-reference*)]
nv_update_occurs [(*network-var-reference*)]
nv_update_succeeds [(*network-var-reference*)]

nv_update_occurs イベントは、入力ネットワーク変数だけに使用します。他の3つのイベント（**nv_update_completes**、**nv_update_fails**、**nv_update_succeeds**）は、更新については出力ネットワーク変数、ポーリングについては入力ネットワーク変数に対して使用します。

イベント式には、ネットワーク変数名（*network-var-reference*）、*network_var* [*index*]という形式のネットワーク変数配列要素、ネットワーク変数配列名、またはネットワーク変数の範囲を指定できます。イベントに配列名を指定すると、各要素ごとにイベントが発生します。

ネットワーク変数の範囲を受け入れるイベントの形式には、以下のようなシンタックスを使います。範囲は2つのネットワーク変数またはネットワーク変数要素の参照で構成されており、これらは範囲であることを示す2つのドット文字"**..**"で区切られています。このシンタックスは上記の4つのイベント名」すべてに適用されます。各ネットワーク変数にはコンパイラによってグローバル・インデックスが代入されます。ネットワーク変数の配列には、各要素に対応する連続したインデックスが代入されます。範囲のイベントは、*network-var-1* のグローバル・インデックスと *network-var-2* のグローバル・インデックスの間の範囲にグローバル・インデックスが含まれているネットワーク変数すべてに適用されます。*network-var-1* のグローバル・インデックスの値は、*network-var-2* のグローバル・インデックスよりも小さい値でなければなりません

event-name [**(***network-var-1 .. network-var-2***)**]

次に、これらのイベントについて詳しく紹介します。なお、ここで説明するイベントはどれもネットワーク変数の更新またはポーリングが完了したかどうかに関連しているので、これらのイベントをネットワーク変数の「完了イベント」と呼ぶことにします。完了イベントの使用方法の詳細については、第6章「アプリケーション・メッセージを使ったデバイス間通信」も参照してください。

nv_update_occurs イベント

入力ネットワーク変数が新しい値を受け取ると、**nv_update_occurs** イベントの値がTRUEになります。このイベントの式として特定のネットワーク変数が指定されていなければ、デバイス上のどのネットワーク変数の更新情報であってもこのイベントがTRUEになります。

注意： 組み込み変数 **nv_in_addr** について詳しくは、『Neuron C Reference Guide』を参照してください。

nv_update_occurs イベントは、さまざまな状況で使用されます。例えば照明ランプのプログラムでは、次のような使い方をしています。

```
// Use the network variable's value
// as the new state for the lamp

when (nv_update_occurs(nv_lamp_state)) {
      io_out(ioLED, nv_lamp_state.state);
}
```

サーモスタット・デバイスが新しい温度設定値を受け取ったとき、現在の温度をチェックし、必要であればヒーターのスイッチを入れたり切ったりするコード例です。

```
network input SNVT_temp tempSetPoint;
network output SNVT_switch primaryHeater;
network output SNVT_temp currentTemp;

when (nv_update_occurs(tempSetPoint)) {
   primaryHeater.state = currentTemp < tempSetPoint;
}
```

nv_update_succeeds と nv_update_fails イベント

ネットワーク変数の更新やポーリングが失敗すると、**nv_update_fails** イベントの値が TRUE になります。**nv_update_fails** イベントに特定のネットワーク変数が指定されていなければ、そのデバイス上のどのネットワーク変数への更新やポーリングが失敗したとしてもイベントが TRUE になります。複数のネットワーク変数が指定されているときには、ネットワーク変数の更新およびポーリングに失敗するたびにイベントが TRUE になります。

同様に **nv_update_succeeds** イベントは、出力ネットワーク変数の更新値の送信が成功するか、ポーリングされた値がすべての書き込みデバイスから受信された場合に TRUE になります。

nv_update_fails イベントは、あらゆる出力ネットワーク変数に使用できます。以下は、1 つの出力ネットワーク変数に **nv_update_fails** イベントを使用した例です。

```
when (nv_update_fails(nv_switch_state))
{
   // take some corrective action
}
```

ネットワークの更新が失敗したか成功したかを調べる例をもう一つ示します。

```
boolean heater_failed;
network output SNVT_switch nv_heater_1;

when (nv_update_fails(nv_heater_1))
{
   heater_failed.state = TRUE;
        // remember update failure
}

when (nv_update_succeeds(nv_heater_1))
{
   heater_failed.state = FALSE;
        // heater device received update
}
```

nv_update_completes イベント

nv_update_completes イベントは、出力ネットワーク変数の更新やポーリングが行われたとき、それが成功しても失敗しても TRUE になります。以下の例では、ネットワーク変数の更新が完了したかどうかを調べています。

```
IO_7 input ontime invert clock(2) io_temperature_sensor;
network output SNVT_temp nv_current_temp;


when (nv_update_completes(nv_current_temp))
{     // latest temperature has been sent out
      ontime_t sensor_value;

      // send another update
      sensor_value = io_in(io_temperature_sensor);
      nv_current_temp = (sensor_value * 221) / 642
                                    + 211 + C_TO_K;
            // tenths of a degree,C
}
```

プログラムにおいて、任意のネットワーク変数の **nv_update_completes** または **nv_update_succeeds** を調べる場合、そのプログラムは完全完了イベント評価を行っていることになります。詳細規則については、本章で後述する「完全完了イベント評価」を参照してください。

サンプル・プログラム

以下のプログラム例では、ネットワーク変数宣言の使い方とイベント処理方法を示しています。このプログラムの一部は、前のセクションでも紹介しました。

```
// therm.nc:  Sample program for a thermostat device
// that is connected to two heater devices and a
// temperature setpoint device.

#include <io_types.h>
#define C_TO_K 2740

// temperature sensor I/O object declaration
IO_7 input ontime invert clock(2) io_temperature_sensor;
IO_2 output bit io_failure_light;
   // LED for heater failure


// Example declarations of network variables using SNVTs
network input SNVT_temp nv_set_point;
   // tenths of a degree C+2740,
   // received from setpoint device

network output SNVT_switch nv_heater_1;
   // control heaters (on/off)

network output SNVT_switch nv_heater_2;

network output SNVT_temp nv_current_temp;
   // exported to other devices


// Function prototype declaration
void heaters_on(boolean state);
```

```
// Example of receiving a network variable update event
when (nv_update_occurs(nv_set_point))
{
   heaters_on(nv_current_temp < nv_set_point);
}


// Example of testing network variable update completion
when (nv_update_completes(nv_current_temp))
{
   ontime_t sensor_value;

   //  latest temperature has been sent out on the network
   //  send another update
   sensor_value = io_in(io_temperature_sensor);
   nv_current_temp = (sensor_value * 221) / 642
                              + 211 + c_to_k;
        // tenths of a degree,C
}

// Example of testing NV update failure and success
boolean heater_device_failed;
       // true if we cannot communicate with heater

when (nv_update_fails(nv_heater_1))
when (nv_update_fails(nv_heater_2))
{
   heater_device_failed = TRUE; // remember device failure
   io_out(io_failure_light, 0); // turn on error indicator
}


when (nv_update_succeeds(nv_heater_1))
when (nv_update_succeeds(nv_heater_2))
{
   heater_device_failed = FALSE;
             // heater device received update
   io_out(io_failure_light, 1);
             // turn off error indicator
}


// Example of polling a network variable.
//(See section on Polling, later in this chapter)
// when this device starts running, get latest value of
// setpoint

when (reset)
{
   poll(nv_set_point);
   io_out(io_failure_light, 1);    // clear error light
   heater_device_failed = FALSE;
}


// Example of using is_bound() function
// control heaters


void heaters_on (boolean state)
{

   // update primary heater NV
   nv_heater_1.state = state;
```

```
    if (is_bound(nv_heater_2))
       // update secondary heater NV only if it is bound
       nv_heater_2.state = state;
}
```

Synchronous（Sync 型）ネットワーク変数

出力ネットワーク変数が更新された場合、Neuron ファームウエアでは、最後に出力に割り当てられた値が伝達され、接続中の入力ネットワーク入力変数がイベントとして受信するようになっています。つまり、短時間に何度も値の更新があったとすると、最後に割り当てられた値だけが確実に伝達され、入力ネットワーク変数がイベントとして受信します。出力ネットワーク変数に対するすべての更新を伝達してイベントとして送受信させる必要がある場合には、ネットワーク変数のサブクラスである「Sync 型」ネットワーク変数を使用してください。

Synchronous（Sync 型）ネットワーク変数の宣言

Sync 型ネットワーク変数を宣言するには、宣言に **synchronized** または **sync** のキーワードを含めます。宣言例は以下に示すとおりです。

```
network output sync SNVT_temp nv_rel_temp;
```

次の例では、ネットワーク変数を Sync 型として宣言し、ネットワーク変数に発生した更新をすべて通知するように設定しています（1 つでも警報器がオフになれば、最後にオフになった警報器だけでなく、オフになったすべての警報器についての警告を受信します）。

```
// ensure multiple alarms are handled serially
network output sync SNVT_alarm sensor_alarm;
```

Sync 型出力ネットワーク変数が Sync 型入力ネットワーク変数に接続している必要はありません。入力ネットワーク変数は、Sync 属性が指定されているかいないかにかかわらず、すべて同期しています。

Sync 型と Nonsync 型ネットワーク変数

ほとんどのアプリケーションは Nonsync 型ネットワーク変数で十分ですから、できるだけ Nonsync 型を使用してください。通常のアプリケーションでは、ネットワーク変数に対するすべての更新値は必要ありません。必要なのは「最新値」だけです。多数の Sync 型ネットワーク変数が頻繁に更新されると、プログラムがバッファ不足を起こす可能性があり、処理の遅れの原因になります（本章で後述する「先取りモード」のセクションを参照してください）。デバイスのバッファリング、チャネル速度、ネットワーク負荷に依存しますが、Sync 型ネットワーク変数を数多く使用すると、それに反比例してアプリケーションの性能が落ちることになります。

相対「デルタ」データ値を使用するプログラムでは、中間のデータ値を保存するために、Sync 型ネットワーク変数を使用する必要が生じることがあります。絶対データ値を扱うプログラムであれば、Nonsync 型ネットワーク変数で十分です。

Nonsync 型ネットワーク変数は、次の出力バッファが使用可能になったときにネットワークに送出されます。もしプログラムがこれよりも早く変数をふ

たたび更新すれば、最新の値だけが送出されます。Sync 型出力ネットワーク変数では、出力バッファがどれも使用できなければ、使用できるようになるまでアプリケーションが待機します。この場合、スケジューラは先取りモードになります（先取りモードについては、次のセクションにある「先取りモード」の項を参照してください）。

入力ネットワーク変数では、変数の更新が常にアプリケーションのイベントとなります。Sync 型を指定しているかどうかにかかわらず、入力ネットワーク変数はすべて同期することに注意してください。

Synchronous（Sync 型）ネットワーク変数の更新

Sync 型ネットワーク変数は次のクリティカル・セクションの終わりで常に更新されます。このとき、バッファが使用できない状態であれば、使用できるようになるまでスケジューラが待機します。一方、Nonsync 型ネットワーク変数の場合は、スケジューラがアプリケーション・バッファを使用できるときにだけ、次のクリティカル・セクションの終わりで更新されます。Sync 型ネットワーク変数とは異なり、常に次のクリティカル・セクションの終わりに値が更新されるわけではありません。前にも指摘しましたが、複数の更新が発生したところでは、中間値はネットワークに伝達されません。

先取りモード

Sync 型出力ネットワーク変数の更新が発生したときにアプリケーション・バッファが使用できないと、スケジューラは「先取りモード」になります。Sync 型出力ネットワーク変数の更新値は必ず送出しなければならないので、アプリケーションの出力バッファが使用可能になるまで、システムは完了イベント（**msg_arrives** や **nv_update_occurs** イベント）や応答イベントの処理を実行します。

イベントに対する **when** 節に **preempt_safe** キーワードを指定していない限り、その他のイベントは処理されません。**when** 節のシンタックスについては、第 2 章「シングル・デバイスでの機能」を参照してください。「先取りモード」の詳しい説明や **preempt_safe** キーワードの使用法については、第 6 章「アプリケーション・メッセージを使ったデバイス間通信」を参照してください。

システムが先取りモードに入ると、アプリケーションの処理が遅れます。どれくらい遅れるかは、アプリケーションの出力バッファ・スペースが空くまでの時間によります。また、この遅れはネットワークのトラフィック、チャネルのビットレートなどによっても異なります。

ネットワーク変数に対する完了イベントの処理

ネットワーク変数には、完了イベントをチェックするための 2 つのモードがあります。1 つは部分完了イベント評価（これがデフォルトです）で、もう 1 つは完全完了イベント評価です。メッセージ・タグでは、完全完了イベント評価だけが使用できます（第 6 章「アプリケーション・メッセージを使ったデバイス間通信」を参照してください）。

部分完了イベント評価

プログラムで部分完了イベント評価を使用する場合は、各ネットワーク変数に対してどのように完了イベントを処理するかについて、次の 2 つの選択が可能です。

1 どの完了イベントのチェックも行いません。

2 失敗イベント（**nv_update_fails**）だけをチェックします。

例えば、2 つのネットワーク変数を含むプログラムがあったとすると、次のような設定ができます。

- ネットワーク変数 1：プログラムは完了イベントのチェックをしません。
- ネットワーク変数 2：プログラムは失敗だけをチェックします。

完全完了イベント評価

完全完了イベント評価は、メッセージ・タグの完了イベント処理に適用できる処理方法で（第 6 章「アプリケーション・メッセージを使ったデバイス間通信」を参照してください）、ネットワーク変数にも適用できます。プログラムで完全完了イベント評価を使用する場合は、各ネットワーク変数に対してどのように完了イベントを処理するかについて、次の 3 つの選択が可能です

1 どの完了イベントのチェックも行いません。

2 失敗と成功のイベント（**nv_update_fails**、**nv_update_succeeds**）をチェックします。

3 更新完了イベント（**nv_update_completes**）をチェックします。

例えば、3 つのネットワーク変数を含むプログラムでは、次の選択が可能です。

- ネットワーク変数 1：プログラムは完了イベントのチェックをしません。
- ネットワーク変数 2：プログラムは失敗と成功のイベントをチェックします。
- ネットワーク変数 3：プログラムは更新完了イベントのみをチェックします。

注意： ネットワーク変数に完全完了イベント評価機能を使用する場合は、ネットワーク変数のすべての完了コード処理を完全完了イベント評価にする必要があります（これはすべてのネットワーク変数に対してイベントを確認しなければならないということではありません。各プログラムでは部分完了イベント評価または完全完了イベント評価のどちらか片方だけを使用できるが、両方を混在させることはできないという意味です）。Neuron C コンパイラは、完全完了イベント機能をプログラム単位で扱います。

長所と短所

あるプログラムの中で完全完了イベント評価を使ってネットワーク変数の完了イベントを処理すると、部分完了イベント評価を使った場合よりもプログラムのコード・サイズが大きくなり、プログラム効率も悪くなります。一度でも **nv_update_completes** のような完全イベント評価機能を選択するイベントを使用すれば、そのプログラムのすべてのネットワーク変数イベントが完

全完了イベント評価機能になります。例えば、完全完了イベント評価を使ったプログラムの中では、**nv_update_fails** イベントだけをチェックすることはできません。なぜなら、この特性は部分完了イベント評価でしか使用できないからです。

ネットワーク変数のポーリング

この章で前述したように、ネットワークの更新は、書き込みデバイスがネットワーク変数に値を代入したときに発生します。つまり、通常ネットワーク変数の更新は書き込みデバイスから開始されるわけです。

ただし、読み込みデバイスが書き込みデバイスに対してネットワーク変数の最新値を送るように要求することもできます。「ポーリング」という用語は、読み込みデバイスから開始されるネットワーク変数の更新処理を意味しています。

デバイスのプログラムは、いつでも任意の入力ネットワーク変数をポーリングできます。これには、電源を入れた時や、オフラインからオンラインになったときも含まれます。ただし、電源が入ったときにネットワーク変数を伝達するように設定されているデバイスが複数存在する場合、多くのデバイスの電源が同時に投入されると、ネットワークに負荷がかかりますから注意してください。

入力ネットワーク変数をプログラムからポーリングするには、ネットワーク・バインドツールは、別の方法を使って書き込みデバイスと読み込みデバイス間の出力ネットワーク変数に接続しなければなりません。このためには、読み込みデバイスに追加のアドレス・テーブル・エントリが必要になります。ポーリングを行っていなかった既存のアプリケーションにポーリングを追加する場合は、そのデバイス用の新しいデバイス・インターフェース・ファイルを作成し、以前のバージョンを使用していたネットワーク管理ツールにそのファイルをインポートする必要があります。

読み込みデバイスがポーリングを要求するには、**poll()**関数を使用します。シンタックスは以下のとおりです。

poll ([network-var]**);**

network-var ネットワーク変数識別子を指定します。

ネットワーク変数名が指定されていないと、デバイス上のすべての入力ネットワーク変数に対してポーリング要求が生成されます。入力ネットワーク変数では、明示的な **polled** 宣言ができないことに注意してください。

また、*network_var* 識別子には、ネットワーク変数配列の識別子や *network_var* [*index*]といった形式のネットワーク変数配列要素も指定できます。インデックスなしでネットワーク変数配列名を指定すると、その配列の要素すべてに対するポーリングが生成されます。

ポーリングによって得られた新しい値は、**poll()**関数呼び出しの直後には使用できません。ネットワーク変数を指定した **nv_update_occurs** を **when** 節に使用するか、他の条件分岐シンタックスを使って、ポーリングした結果の値を受け取ってください。

コード例

```
when (timer_expires(t)) {
   poll(nv_cooling_mode);
   . . .
}

when (nv_update_occurs(nv_cooling_mode)) {
   . . .
}
```

以下の例は、リセット・イベントの後で入力ネットワーク変数 **nv_lamp_state** をポーリングする照明ランプのプログラムです。デバイスがリセットされると、デバイスは **nv_lamp_state** の最新値を調べ、その値を使用します。

```
// LAMP.NC -- Sample lamp actuator program,
// polls the switch on reset

////////////////// Network Variables ////////////////////
network input SNVT_switch nv_lamp_state = {0,0};


////////////////////////// Constants ////////////////////
#define LED_ON     1
#define LED_OFF    0


////////////////////// I/O Objects ////////////////////
IO_0 output bit ioLED = LED_OFF;


///////////////////////////// Tasks /////////////////////
// NV update task -- handle update to lamp state
// Use the network variable's value as the new state
// for the lamp
when (nv_update_occurs(nv_lamp_state)) {
      io_out(ioLED,
            nv_lamp_state.value && nv_lamp_state.state
                  ? LED_ON : LED_OFF);
}


////////////////////////////////////////////////////////
// Reset task -- request last value from any switch attached
when (reset) {
   poll(nv_lamp_state);
}
```

ポーリングされるネットワーク変数の宣言

ポーリングの要求は、読み込みデバイスが生成します。書き込みデバイスがネットワーク変数に頻繁に値を代入したとしても、読み込みデバイス側はそれらの更新を特定の時間にだけ受け取るようにしたいことがあります。このような場合、出力ネットワーク変数を **polled** として宣言します。

network output polled *type netvar*;

このように宣言すると、出力ネットワーク変数の値に変更があってもその値は伝達されません。この出力ネットワーク変数の値は、読み込みデバイスからのポーリング要求に応答する場合か、そのネットワーク変数に対して **propagate()**関数が呼び出された場合にのみ送信されます。

コード例

照明ランプ・プログラムとスイッチ・プログラムの例は、スイッチの状態を表すネットワーク変数がポーリングによって更新されるように書き直すことができます。以下にポーリングを使うように書き直したプログラムを示します。

リスト 3-1　ポーリング版ランプ・プログラム

```
// LAMP.NC -- Sample lamp actuator program,
// polls the switch periodically


//////////////////// Network Variables ////////////////
network input SNVT_switch nv_lamp_state = {0,0};


////////////////////////// Constants ////////////////////
#define LED_ON     1
#define LED_OFF    0


////////////////////// I/O Objects ////////////////////
IO_0 output bit ioLED = LED_OFF;


////////////////////////// Timers ////////////////////////
mtimer tmPoll;


///////////////////////////// Tasks /////////////////////
//   NV update task -- handle update to lamp state
//   Use the network variable's value as the new
// state for the lamp
when (nv_update_occurs(nv_lamp_state)) {
      io_out(ioLED,
            nv_lamp_state.value && nv_lamp_state.state
                  ? LED_ON : LED_OFF);
      tmPoll = 500;  // Wait 500 msec before polling again
}


////////////////////////////////////////////////////////
// Reset and timer task
// request last value from any switch attached
when (reset)
when (timer_expires(tmPoll) ) {
   poll(nv_lamp_state);
}
```

リスト 3-2　ポーリング版スイッチ・プログラム

```
// SWITCH.NC -- Sample switch sensor program
// Only transmits switch state when polled by the lamp


////////////////////// Compiler Pragmas /////////////////
#pragma enable_io_pullups
```

```
//////////////////// Network Variables  ////////////////
network output polled SNVT_switch nv_switch_state = {0,0};


////////////////////////// Constants
//////////////////////////
#define BUTTON_DOWN 1
#define BUTTON_UP   0


////////////////////// I/O Objects ////////////////////
IO_4 input bit ioButton = BUTTON_UP;


///////////////////////////// Tasks ///////////////////////
// I/O task -- handle pushbutton down event
// Just toggle the network variable (nv_switch_state).
// In this case, no message is sent until a poll request
// is received from a reader device
when (io_changes(ioButton) to BUTTON_DOWN)
{    // button pressed
   nv_switch_state.state = !(nv_switch_state.state);
}    // toggle state
```

ネットワーク変数の明示的な伝達

本章で前述したように、書き込みデバイスがネットワーク変数に値を代入したとき、ネットワーク変数の更新が発生します。通常、値が変更されると、コンパイラが生成したコードによって自動的にネットワーク変数が更新されるようになっています。

この他に、出力ネットワーク変数をネットワークに送出するように、アプリケーションから明示的に要求することもできます。この機能は、変数を直接変更しない場合や、ネットワーク変数へのポインタを使用している場合などに使用します。このように出力デバイスが明示的にネットワーク変数を更新する処理手順のことを「伝達」と呼んでいます。

デバイスのプログラムは、いつでも任意の出力ネットワーク変数を伝達できます。これには、電源を入れた時や、オフラインからオンラインになったときも含まれます。ただし、電源が入ったときにネットワーク変数を伝達するように設定されているデバイスが複数存在する場合、多くのデバイスの電源が同時に投入されると、ネットワークに負荷がかかりますから注意してください。

アプリケーションから伝達を要求するためには、**propagate()**関数を使用します。シンタックスは以下のとおりです。

propagate ([*network-var*]**);**

network-var　　出力ネットワーク変数識別子を指定します。

ネットワーク変数名が指定されていないと、デバイスのすべての出力ネットワーク変数が伝達されます。また、*network_var* 識別子には、ネットワーク変数配列の識別子や *network_var* [*index*]といった形式のネットワーク変数配列要素も指定できます。インデックスなしでネットワーク変数配列名を指定すると、その配列の全要素が伝達されます。

propagate()関数は、**const** として宣言しているネットワーク変数を送信するために使用することがあります。ネットワーク変数の伝達は、通常は値への代入によって起こるのですが、**const** 変数には代入ができないため、明示的に要求しない限り伝達はできません。詳しくは、『Neuron C Reference Guide』の **propagate()**関数の説明を参照してください。

コード例

```
// The variable below is a special node ID
network output const unsigned long nodeID = 24221;

when (some-special-event)
{
   propagate(nodeID);
}
```

propagate()関数は、ネットワーク変数へのポインタを使用しているときにも便利です。例えば、複雑な値の組み合わせの計算をして、その結果をネットワーク変数の構造体の中に保存する関数 **f()**を定義してあったとします。この関数はデバイスの中の類似した変数を扱うように設計されており、関数への引数は変数へのポインタになっていると仮定します。

この関数をプログラミングするとき、変数をポインタを使って参照すれば、効率のよいコードが作成できます。しかし Neuron C コンパイラは、ポインタの参照先が内部変数であるのかネットワーク変数であるのかという区別はしません。したがって、ポインタを介してネットワーク変数が更新されても自動的に伝達されないので、伝達を明示的に要求する必要があります。

さらに次のプログラム例では、ネットワーク変数へのポインタを区別できないため、ネットワーク変数へのポインタを **const** データへのポインタとして扱っていることに注意してください。このように、ポインタが変数を変更するのを回避しています。Neuron C では、通常は **const** 属性の指定を取り除くことはできません。ただし、**#pragma relaxed_casting_on** 指令を使用すると、この型変換が許可されるようになります。明示的な型変換でも、変数の代入や関数パラメータの引き渡しによる暗黙的な型変換でも、どちらも使用可能です。

コード例

```
typedef struct complex_struct {
      .... // struct definition here
} complex_type;

network output complex_type nv1, nv2, nv3;

void f(complex_type *p) {
      .... // calculations & modification of (*p).
      .... // Neuron C cannot distinguish between pointers
      .... // to network variables and pointers to
      .... // non-network variables.
      .... // Thus, any modifications here do not cause any
      .... // propagation of an NV.
}
```

```
when (some-event)
{
#pragma relaxed_casting_on
      // Without pragma above, this would result in
      // an error, because the address of a network
      // variable is treated as 'const <type> *'.
      // Passing such a type as the function parameter
      // results in an implicit cast, since the function
      // prototype defines the variable as '<type> *'.
      f(&nv1);
      propagate(nv1);   // Explicit propagation needed
                  // since f() modified nv1 via pointer.
      f(&nv2);
      propagate(nv2);
      f(&nv3);
      propagate(nv3);
}
```

ネットワーク変数のモニター

LONWORKS デバイスの 1 つに、多数のデバイスからデータを受け取るモニター・デバイスがあります。監視されるデバイスは、通常は同じ種類のデバイスです。例えば、警報表示デバイスは多数の警報センサー・デバイスをモニターしています。すべてのセンサー・デバイスが **SNVT_switch** 出力として宣言された出力ネットワーク変数を持っているとすると、モニター・デバイスの入力ネットワーク変数も **SNVT_switch** 入力として宣言されているはずです。

一般的に、モニター・デバイスは、そのデバイスにある入力ネットワーク変数が変化するのを待ちます。変更が発生すると、どのデバイスからの変更であるかを識別します。ネットワーク変数を変更したデバイスを判別する方法は、センサー出力とモニター入力との接続方法によって異なります。

以下にネットワーク・モニター・デバイスの例をいくつか示します。この例では、どのセンサー・デバイスも **SNVT_switch** 出力ネットワーク変数を 1 つ持っており、その出力ネットワーク変数をネットワーク・モニター・デバイスが監視しているものとします。

- ネットワーク変数を配列として宣言し、その配列の各要素を別々のセンサーに接続します。配列全体に対する **nv_update_occurs** イベントを待ち、**nv_array_index** 組み込み変数を使ってどのデバイスが変更を発生させたかを判別します。以下にコード例を示します。

  コード例

  ```
  network input SNVT_switch nv_alarm_array[50];
  SNVT_switch alarm_value;
  unsigned int  alarm_device;

  when (nv_update_occurs(nv_alarm_array))
  {
        alarm_device =  nv_array_index;
        alarm_value = nv_alarm_array[alarm_device];

        // Process alarm_device and alarm_value
  }
  ```

この方法は、デバイス数がモニター・デバイスのネットワーク変数の数の上限を越えないときに使ってください。ネットワーク変数は、Neuron チップをホストとしたデバイスで 62 個、ホスト・ベースのデバイスで 4,096 個まで使用できます。

- 入力ネットワーク変数をモニター・デバイス上の単一入力として宣言し、センサー・デバイスでは出力ネットワーク変数をポーリング出力として宣言します。すべてのセンサー出力をモニター入力に接続するコネクションを 1 つ作成します。次の章で説明する明示的なアドレス指定と明示的メッセージを使って、各センサーにポーリングを行います。ポーリングによって更新値を受け取るので、モニター・デバイスはいつでもネットワーク変数の更新元がわかります。

  この方法は、ポーリング・ループによって発生する遅れ時間がアプリケーションにとって受容範囲である限り、デバイスがいくつあっても適用できます。

- ネットワーク変数入力を単一入力として宣言し、すべてのセンサー出力をモニター入力に接続するコネクションを 1 つ作成します。入力ネットワーク変数への **nv_update_occurs** イベントを待ち、イベントが起こったら **nv_in_addr** 組み込み変数を使って変更元デバイスのソース・アドレスを識別します。以下は、ネットワーク・モニター・デバイスのコード例です。

  コード例

```
network input SNVT_switch nv_alarm_in;
SNVT_switch alarm_value;
nv_in_addr_t alarm_device_addr;

when (nv_update_occurs(nv_alarm_in)) {
    alarm_device_addr = nv_in_addr;
    alarm_value = nv_alarm_in;
       // Process alarm_device_addr and alarm_value
       // Look up alarm_device_addr in a configuration
       // property set by a plug-in at installation time
}
```

この方法は、デバイスの数に関わらず使用できます。

nv_in_addr 組み込み変数の内容については『Neuron C Reference Guide』を参照してください。

認証機能（Authentication）

「認証機能」は、1 つの書き込みデバイスと、1～63 個までの読み込みデバイスとの間の特殊な確認サービスです。認証機能は、読み込みデバイスが書き込みデバイスの身元を確認するために使用します。この種のサービスは、次のようなときに役に立ちます。例えば、電子ロックを持つデバイスが、そのロックを解除するメッセージを受けたとします。この「解除」のメッセージをデバイスの所有者が送ったのか、それとも誰かがシステムへの侵入目的で送ったのか、電子ロック・デバイスが判断するのに認証機能を利用します。

認証機能を使用すると、トランザクションあたりのメッセージ数が 2 倍になります。認証機能は、確認応答付きの更新やネットワーク変数のポーリングにも使用できます。確認応答なしの更新や繰り返し更新には使用できませんから、注意してください。通常の確認応答付きメッセージには、更新メッセージと確認メッセージの 2 つのメッセージが必要ですが、認証機能のメッセ

ージでは図 3.7 で示すように 4 つのメッセージが必要になります。このことは、システムの応答時間と処理能力に影響します。

次の各セクションでは、認証機能を使用するためのデバイスの設定方法と、認証機能の動作について説明します。

認証を使用するためのデバイスの設定

認証機能付きネットワーク変数を使用するか、認証機能付きメッセージを送信するようにデバイスを設定するには、次のステップに従います。

1 ネットワーク変数を認証機能付きとして宣言します。認証機能を付けるアプリケーション・メッセージについては、**msg_out** オブジェクトの **authenticated** フィールドを TRUE にします。

2 ネットワーク管理ツールを使って、デバイスに対する認証キーを指定します。開発中のキーのインストールには、LonMaker ツールを使用します。

これらのステップについては、次の各セクションで詳しく説明します。

認証機能付き変数とメッセージの宣言

ネットワーク変数のコネクション情報には、**authenticated**（または **auth**）キーワードが含まれています。シンタックスは以下のとおりです。

bind_info ([authenticated | nonauthenticated]
[(config | nonconfig)])

注意： キーワード **authenticated** は、短縮して **auth** と指定することもできます。同様にキーワード **nonauthenticated** は **nonauth** と短縮できます。

宣言に **config** キーワードも含めると、デバイスがインストールされた後でもネットワーク変数の認証状態をネットワーク管理ツールで変更できます。**nonconfig** キーワードを含めると、ネットワーク変数の認証状態は変更できなくなります。

コード例:

```
network output boolean
   bind_info(auth(nonconfig)) nv_safe_lock;
```

この宣言には **nonconfig** キーワードが含まれているため、**nv_safe_lock** ネットワーク変数の更新に対する認証機能がオフになることはありません。

認証キーの設定

認証機能付きネットワーク変数を読み込むデバイスも書き込むデバイスも、同じ認証キーを持っている必要があります。この 48 ビットの認証キーをどのように使用するかは、この後で説明します。

認証キーは、初期設定の間にデバイスに送信されます。その後でキーを変更するときには、新しいキー自身がネットワーク上に送出されるわけではありません。ネットワーク管理ツールは、ネットワーク管理メッセージを使って、ネットワークを通じて安全にデバイスのキーを変更することができます。

認証機能の処理手順

認証機能の例を順に説明していきます。図 3.7 には、その処理過程を示します。

1 デバイス A は、ackd サービスを使ってデバイス B 上のネットワーク変数に更新情報を送ります。このときのデバイス B のネットワーク変数の宣言には auth キーワードが含まれているとします。デバイス A は、チャレンジ・メッセージの受信を待ち、受信がなければ、初期更新のリトライを送信します。

2 デバイス B は 64 ビットの乱数を発生し、デバイス A に対してこの乱数を含むチャレンジ・パケットを送り返します。次にデバイス B は暗号化アルゴリズム（Neuron ファームウエアに組み込まれています）を使って、48 ビットの認証キーとメッセージ・データから乱数の変換を行います。この変換結果はデバイス B に保存されます。

3 デバイス A も暗号化アルゴリズム（Neuron ファームウエアに組み込まれています）を使って、48 ビットの認証キーとメッセージ・データから乱数（デバイス B から送り返されたもの）の変換を計算します。そして、デバイス A は計算した変換値をデバイス B に送ります。

4 デバイス B は、自分で計算した変換値とデバイス A から送られてきた変換値を比較します。もし 2 つの数字が一致すれば、送り手の身元が確認されたことになり、デバイス B は要求された動作を行い、デバイス A に確認応答メッセージを送ります。もし 2 つの数字が一致しなければ、デバイス B は要求された動作を行わず、エラー・テーブルにエラーを記録します。

デバイス B が送った確認応答メッセージがデバイス A まで届かずに消失してしまったときには、デバイス A が同じメッセージを再送します。デバイス B は認証が成功していることを覚えているので、もう一度確認応答メッセージだけを送ります。

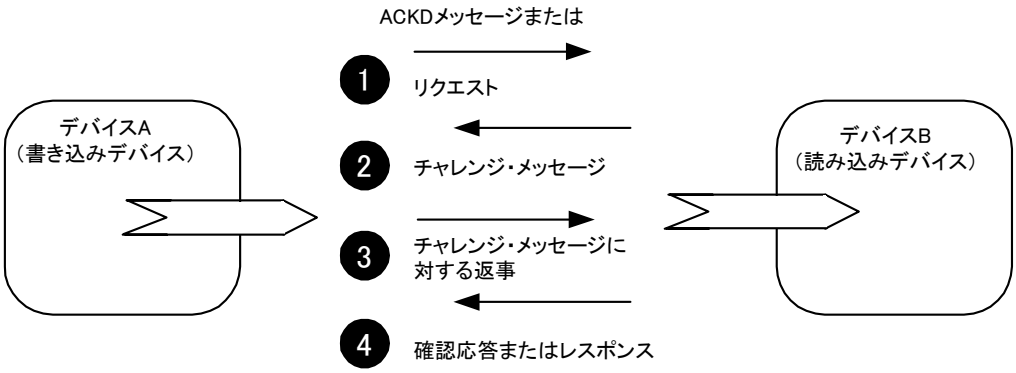


図 3.7　認証機能の処理

デバイス A の出力ネットワーク変数が複数の入力ネットワーク変数に接続していると、デバイス A が更新情報を送った後で各読み込みデバイスがそれぞれ異なる 64 ビットの乱数をチャレンジ・パケットに乗せてデバイス A に送ります。すると、デバイス A は受け取った各乱数について変換を行い、それぞれの読み込みデバイスに対してその返事を送らなければなりません。

認証機能の大きな強みは、単純な記録や特定の機能（例：ロックの解除）を実現したコマンドの繰り返しでは破ることができない点にあります。この認証機能では、メッセージやコマンドは暗号化されていませんから、メッセー

ジの内容を調べようと思えば読むことはできます。したがって、特定のメッセージやコマンドを非公開にする必要はありません。

デバイスの認証キーを最初にインストールするときには、デバイスをネットワーク管理ツールに直接接続し、同じネットワーク上で他のデバイスを使用しないようにしてください。これによって、侵入者によって盗聴される恐れのある大きなネットワーク上でキーが送信されるのを防ぐことができます。いったんデバイスに認証キーをインストールすると、その後でネットワーク管理ツールによるキーの変更を行うときは、既存のキーに加算する数だけがネットワーク経由で送信されます。

型変更可能なネットワーク変数

インストール時に型やサイズを変更できるネットワーク変数は「型変更可能なネットワーク変数」と呼ばれ、ユーザが作成することも可能です。型は、リソース・ファイル（リソース・ファイル内の任意の SNVT または UNVT）で定義されている任意のネットワーク変数型に変更できます。型変更可能なネットワーク変数は、機能ブロックのメンバまたは構成プロパティ（構成ネットワーク変数）であるネットワーク変数に対してのみ作成できます。ネットワーク変数の型を変更するには、通常、作成時に用意したプラグインを使用します。ネットワーク変数の型が変更された場合、この変数を他のネットワーク変数に接続することはできなくなります。型を変更することによって、接続が無効になるためです。型変更可能なネットワーク変数を作成するには、次のステップに従ってください。

1 **changeable_type** キーワードを使用して、ネットワーク変数を宣言します。このキーワード情報は、デバイス・インターフェースの説明に記載されます。この情報は、実装されたネットワーク変数の型が、ネットワーク管理ツールを使用して変更できるということを伝えるためのものです。ネットワーク変数には初期の型を宣言する必要があります。また、初期の型のサイズはアプリケーションがサポートできる最大のネットワーク変数のサイズに等しくする必要があります。

   例えば、次の宣言では、**SNVT_volt_f** という初期の型を使用して、型の変更可能なネットワーク変数を宣言しています。この型は 4 バイトの浮動小数点数値であるため、このネットワーク変数は 4 バイト以下のネットワーク変数に変更できます。

   network output changeable_type SNVT_volt_f ...

2 デバイス・テンプレートのプログラム ID に、変更可能なインターフェース・ビットを設定します。このビットを設定するには、『NodeBuilder User's Guide』の説明に従ってデバイス・テンプレートを作成するときに、標準プログラム ID カリキュレータに **Has changeable interface** を設定します。

3 型の変更可能なネットワーク変数に適用される **SCPTnvType** 構成プロパティを宣言します。構成プロパティの詳細については、第 4 章「構成プロパティを使ったデバイス動作の構成」を参照してください。この構成プロパティは、ネットワーク変数の型が変更されたことをネットワーク管理ツールがアプリケーションに通知するために使用します。

   この構成プロパティが変更されたときは、アプリケーションに通知する必要があります。通知を実行するには、**reset_required** 修飾子を使用して構成プロパティを宣言し、リセット・ディレクタ関数の **SCPTnvType** の値を確認し、FTP を経由して構成プロパティを実装し、**SCPTnvType** 値が変更されているかどうかを **stop_transfer()**関数で確認するか、**SCPTnvType** 構成プロパティ

を構成ネットワーク変数として実装し、**nv_update_occurs(***nv-name***)**イベントのタスクの現在の型を確認します。

例えば、次のコードでは、**SCPTnvType** 構成プロパティを使用して型変更可能な出力ネットワーク変数を宣言します。

```
SCPTnvType cp_family cp_info(reset_required) nvType;
network output changeable_type SNVT_volt_f nvo1
      nv_properties { nvType };
```

4 型変更可能なネットワーク変数に適用される **SCPTmaxNVLength** 構成プロパティを宣言することもできます。この構成プロパティは、型変更可能なネットワーク変数がサポートしている型の最大長をネットワーク管理ツールに伝えます。この値は定数なので、構成プロパティは **const** 修飾子を使用して宣言します。例えば次のコードを指定すると、**SCPTmaxNVLength** 構成プロパティが前のステップの例に追加されます。

```
SCPTnvType cp_family cp_info(reset_required) nvType;
const SCPTmaxNVLength cp_family nvMaxLength;
network output changeable_type SNVT_volt_f nvo1
      nv_properties { nvType,
                      nvMaxLength=sizeof(SNVT_volt_f) };
```

5 コードを Neuron C アプリケーションに実装し、**SCPTnvType** の値の変更を処理します。必要なコードについては次のセクションで説明します。

6 LonMaker ブラウザは、ネットワーク変数の型を変更するためのユーザ・インターフェースを提供します。通常はこのインターフェースをカスタマイズして、使用しているデバイスでネットワーク変数型を変更できるようにします。例えば、カスタム・インターフェースによって、変更可能な型をアプリケーションがサポートしている型だけに制限すると、構成エラーを防ぐことができます。カスタム・インターフェースを提供するには、プラグインにコードを実装して、ネットワーク変数の型を変更するためのユーザ・インターフェースを提供します。必要なプラグインの変更については、『LNS Plug-in Programmer's Guide』を参照してください。

警告: ネットワーク変数を型変更可能なネットワーク変数として宣言すると、デバイス内のネットワーク変数の自己識別データがすべて書き込み可能なメモリに配置されます。このため、デバイスにアプリケーション用の書き込み可能な外部 EEPROM またはフラッシュ・メモリがない場合には、Neuron 3150 チップまたは FT 3150 スマート・トランシーバを使用したデバイスのメモリにアプリケーションを収めることが困難になる場合があります。

SCPTnvType CP の変更の処理

プラグインまたは LonMaker ブラウザによってネットワーク変数の型が変更されると、そのネットワーク変数に関連付けられている **SCPTnvType** 構成プロパティに新しい値を書き込むことによって、変更がなされたことがアプリケーションに通知されます。**SCPTnvType** 型の定義は以下のとおりです。

```
typedef struct {
  unsigned short          type_program_ID[8];
  unsigned short          type_scope;
  unsigned short          type_index;
  nv_type_category_t      type_category;
  unsigned short          type_length;
  signed long            scaling_factor_a;
  signed long            scaling_factor_b;
  signed long            scaling_factor_c;
} SCPTnvType;
```

アプリケーションは **SCPTnvType** 値の変更を検出したとき、以下の「型変更の検証」の説明に従って、変更が有効であるかどうかを確認する必要があります。有効である場合は、以下の「型変更の処理」の説明に従って変更を処理する必要があります。変更が有効でないか、サポートされていないことがわかった場合には、「型変更の拒否」の説明に従ってエラーを報告する必要があります。変更が有効で、アプリケーションによってサポートされており、この変更によってネットワーク変数のサイズも変更される場合には、以下の「サイズ変更の処理」の説明に従ってサイズの変更を実装する必要があります。

型変更の検証

アプリケーションが **SCPTnvType** の特定の値をサポートしているかどうかを判断する方法はいくつかあります。1 つの方法は、**type_program_ID** フィールドと **type_scope** フィールドに指定されている特定の型を調べることです。また、**type_category** フィールドと **type_length** フィールドに定義されている特定の型のカテゴリを調べることもできます。

type_program_ID と **type_scope** の値は、プログラム ID テンプレートとリソース範囲をそれぞれ指定し、これらの組み合わせにより、リソース・ファイルのセットを一意に識別します。**type_index** の値は、そのリソース・ファイル・セット内のネットワーク変数の型を識別します。**type_scope** の値が 0 の場合、**type_index** の値は SNVT インデックスになります。**type_program_ID** と **type_scope** の値は、ネットワーク変数の現在の型を照会あるいは変更するよう要求しているネットワークツールのタイプに加え、アプリケーションのタイプも一意に識別します。**SCPTnvType** 構造体内の残りのフィールドによってアプリケーションに関する十分な情報が提供されている場合、これらの値は無視できます。

type_category 列挙体は次のように**<snvt_nvt.h>**インクルード・ファイルに定義されています。

```
typedef enum nv_type_category_t {
      /*  1 */    NVT_CAT_SIGNED_CHAR = 1,   // Signed Char
      /*  2 */    NVT_CAT_UNSIGNED_CHAR,     // Unsigned Char
      /*  3 */    NVT_CAT_SIGNED_SHORT,      // 8-bit Signed Short
      /*  4 */    NVT_CAT_UNSIGNED_SHORT,    // 8-bit Unsigned Short
      /*  5 */    NVT_CAT_SIGNED_LONG,       // 16-bit Signed Long
      /*  6 */    NVT_CAT_UNSIGNED_LONG,     // 16-bit Unsigned Long
      /*  7 */    NVT_CAT_ENUM,              // Enumeration
      /*  8 */    NVT_CAT_ARRAY,             // Array
      /*  9 */    NVT_CAT_STRUCT,            // Structure
      /* 10 */    NVT_CAT_UNION,             // Union
      /* 11 */    NVT_CAT_BITFIELD,          // Bitfield
      /* 12 */    NVT_CAT_FLOAT,             // 32-bit Floating Point
      /* 13 */    NVT_CAT_SIGNED_QUAD,       // 32-bit Signed Quad
      /* 14 */    NVT_CAT_REFERENCE,         // Reference
      /* -1 */    NVT_CAT_NUL = 1            // Invalid Value
} nv_type_category_t;
```

この列挙体は、例えばそれが **signed short**、浮動小数点、または構造体であるかどうかなど、型については記述しますが、構造体や共用体のフィールド、あるいはその他の類似の詳細情報は提供しません。**type_length** フィールドは、すべての型に設定されていますが、必要になるのは構造体または共用体の型のバイト数を指定するとき必要です。すべての数量型をサポートするには、**NVT_CAT_SIGNED_CHAR** と **NVT_UNSIGNED_LONG** 間、および **NVT_CAT_SIGNED_QUAD** の **type_category** 値を評価します。さらに浮動小数点型をサポートするには、**NVT_FLOAT** の **type_category** 値も評価してください。

型の変更の処理

型の変更に必要な処理は、アプリケーションがサポートしている型の範囲によって異なります。例えば、異なる種類の浮動小数点型間での変更のみをアプリケーションがサポートしている場合、通常は追加の処理は必要とされません。ただし、異なる数量型間の変更をサポートしている場合には、スケーリング係数とネットワーク変数の型の長さを使用して、生のネットワーク変数値を、スケーリングされた値に変換する必要が生じることがあります。例えば **SNVT_lev_cont** 型は、0～100%のパーセント率を表す、0.5%刻みの **unsigned short** 値です。実際のデータ値（「生の」値とも呼ばれる）は 0～200 の範囲の変数です。**SNVT_lev_cont** のスケーリング係数は、a=5、b= -1、c=0 として定義されています。スケーリングされた固定小数点データに生のデータを変換するには、次の計算式を使用します。

$$scaled = (a * (10 ** b) * (raw + c))$$

アプリケーションは、型変換可能な入力ネットワーク変数の生のデータを、スケーリングされた実際の値に内部で変換し、例えば上の計算式で浮動小数点データ項目として使用できます。データを生の値に戻し、出力ネットワーク変数に使用するには、次の逆スケーリング計算式を使用します。

$$raw = (scaled / (a * (10 ** b))) - c$$

型の変更は、型変換操作やポインタ操作を利用して処理できます。後述の「変更可能な型の例」も参照してください。通常この方法は、それぞれサイズの異なる型をサポートしているネットワーク変数を取り扱うのに最適な方法です。別の方法として、考え得るサポート可能なすべての型の共用体を使用し、ポインタを使用してこの共用体の定義をネットワーク変数に重ねてみることができます。

アプリケーションが構造体や共用体などの非数量型への変更をサポートしている場合、処理はさらに複雑になります。サポートする各非数量型ごとにハードコーディングを行うか、アプリケーション内のリソース・ファイルのサ

ポートを実装する必要があります。後者の方法は、Neuron C アプリケーションでは選択できません。

サイズ変更の処理

SCPTnvType 構成プロパティを変更すると、ネットワーク変数の型のサイズが変更される場合は、アプリケーションでネットワーク変数の組み込み **nv_len** プロパティを使用して、新しい長さを明示的に設定する必要があります。また、**<modnvlen.h>**インクルード・ファイルも含める必要があります。

nv_len 組み込みプロパティは、以下のようにしてアクセス（変更）できます。

```
size_t oldNVLen, newNVLen;
oldNVLen = nv-name :: nv_len;
nv-name :: nv_len = newNVLen;
```

警告： Neuron C コンパイラによって、**nv_len** プロパティを使用したネットワーク変数長の変更が検出された場合、ネットワーク変数の固定構成テーブルを、書き込み可能メモリに配置する要求がリンカに送られます。このため、デバイスにアプリケーション用の EEPROM またはフラッシュ・メモリなどの書き込み可能な外部メモリがない場合には、Neuron 3150 チップまたは FT 3150 スマート・トランシーバを使用したデバイスのメモリにアプリケーションを収めることが困難になる場合があります。

型変更の拒否

ネットワーク管理ツールが、型変更可能なネットワーク変数の型をアプリケーションによってサポートされていない型に変更しようとしたとき、アプリケーションは次の操作を行う必要があります。

- エラーを報告し、機能ブロックを無効にします。これは、デバイス・オブジェクト機能ブロックの **nvoStatus** 出力にある機能ブロックに対して **locked_out** と **disabled** の状態を設定するか、エラー・コードを設定してオフラインにすることで設定します。機能ブロックの状態を設定しても、デバイスの残りの機能ブロックは、正常通りに動作を継続します。機能ブロックを使用してエラー・コードを設定することで、さらに正確なエラー状況をネットワーク・インテグレータに提供することも可能です。機能ブロックの使用の詳細については、第 5 章「機能ブロックを使用したデバイス・インターフェースの実装」を参照してください。

  例えば、次のコード例では、型の変更を検証し、型の変更が無効の場合はエラー・コードを設定して、機能ブロックの状態を変更します（**fblockData[ ]** 配列を使用するには、『NodeBuilder User's Guide』の説明に従って NodeBuilder コード・ウィザードを使って定義します）。

```
#define TYPE_ERROR 1
#define NV_LENGTH_MISMATCH 2

if  (   (nv-name::nvType.type_category == NVC_CAT_UNION)
     || (nv_name::nvType.type_category == NVC_CAT_ARRAY) ) {
       error_log(TYPE_ERROR);
       fblockData[fb-index].objectStatus.locked_out = TRUE;
       fblockData[fb-index].objectStatus.disabled = TRUE;
} else if (nv-name::nv_len > nv-name::nvMaxLength) {
       error_log(NV_LENGTH_MISMATCH);
       fblockData[fb-index].objectStatus.locked_out = TRUE;
       fblockData[fb-index].objectStatus.disabled = TRUE;
}
```

- **SCPTnvType** の値を以前の値にリセットします。

変換可能な型の例

次の例では、**nvo1** 出力ネットワーク変数の型の変更、具体的には、浮動小数点型と **unsigned long** 型間の変更がサポートされています。型の変更が可能なネットワーク変数の宣言では、ネットワーク変数の初期の型として、4 バイトの型である浮動小数点型が使用されます。これは、サポートされている 2 種類の型のうち、サイズが大きい方の型を使用するためです。アプリケーションでは **SCPTnvType** プロパティの **type_category** フィールドを使用して、変数の処理方法を判別します。

```
#include <control.h>
#include <float.h>
#include <modnvlen.h>
#include <snvt_nvt.h>
#pragma relaxed_casting_on

#define TYPE_ERROR 1
#define NV_LENGTH_MISMATCH 2

SCPTnvType cp_family cp_info(reset_required) nvType;
const SCPTmaxNVLength cp_family nvMaxLength;
network output changeable_type SNVT_volt_f nvo1
       nv_properties {
              nvType,
              nvMaxLength=sizeof(SNVT_volt_f)
       };

fblock SFPTopenLoopSensor {
       nvo1 implements nvoValue;
} fbSensor external_name("Sensor");

SCPTnvType nvTypeLastGood;
float_type float_val;
unsigned long ul_val;

if (nvo1::nv_len>nvo1::nvMaxLength) {
       nvType = nvTypeLastGood;
       error_log(TYPE_ERROR);
       fblockData[fbSensor::global_index].objectStatus
                     .locked_out = TRUE;
       fblockData[fbSensor::global_index].objectStatus
                     .disabled = TRUE;

} else switch (nvo1::nvType.type_category) {
       case NVT_CAT_UNSIGNED_LONG:
              nvTypeLastGood = nvType;
              nvo1::nv_len = nvo1::nvType.type_length;
              ul_val = *((unsigned long *)(&nvo1));
       break;

       case NVT_CAT_FLOAT:
              nvTypeLastGood = nvType;
              nvo1::nv_len = nvo1::nvType.type_length;
              float_val = *((float_type *)(&nvo1));
       break;
```

```
        default:
                // handle setting to an unsupported type
                nvType = nvTypeLastGood;
                error_log(TYPE_ERROR);
                fblockData[fbSensor::global_index].objectStatus
                                .locked_out = TRUE;
                fblockData[fbSensor::global_index].objectStatus
                                .disabled = TRUE;
        break;
}
```

4

構成プロパティを使ったデバイス動作の構成

本章では、構成プロパティの宣言と使用方法について説明します。構成プロパティとは、デバイスの動作を構成するデバイス・インターフェースの一部です。構成プロパティは、デバイスのネットワークへのインストール中およびインストール後に、ネットワーク管理ツールを使用して設定します。

概要

構成プロパティは、ネットワーク変数と同様、デバイスのインターフェースの一部を成すデータ項目です。構成プロパティの変更にはネットワーク管理ツールを使用します。構成プロパティは、標準化されたネットワーク・インターフェースを提供することで、相互運用性の高いデバイスのインストールや構成を実現します。ネットワーク変数と同様、構成プロパティにも配慮の行き届いたインターフェースが用意されています。各構成プロパティの型は、リソース・ファイルで定義します。リソース・ファイルは、構成プロパティのデータのエンコーディング、スケーリング、単位、デフォルト値、範囲、および動作を、各型に基づいて指定しています。標準のリソース・ファイル・セットには、さまざまな標準構成プロパティ型（SCPT）が定義されています。独自のユーザ構成プロパティ型（UCPT）を作成することもできます。ユーザ構成プロパティ型を定義するには NodeBuilder リソース・エディタを使用してリソース・ファイルを作成します。

構成プロパティの宣言

構成プロパティの実装方法には 2 種類あります。*configuration network variable*（構成ネットワーク変数）と呼ばれる方法では、ネットワーク変数を使用して、構成プロパティを実装します。これには他のネットワーク変数と同じように、別の LONWORKS デバイスから構成プロパティを変更できるという利点があります。また、Neuron C のイベント処理方法を利用して、イベントの更新を構成プロパティに通知することもできます。

構成ネットワーク変数の欠点は、それぞれの長さが最大 31 バイトに制限されており、Neuron チップまたはスマート・トランシーバがホストしているデバイスでは、ネットワーク変数の数が最大 62 に制限されることです。

構成プロパティを構成ネットワーク変数として実装するには、次の「構成ネットワーク変数の宣言」のセクションで説明するとおり、**network … config_prop** のシンタックスを使用して宣言します。

構成プロパティを実装する 2 番目の方法では、*configuration file*（構成ファイル）内にデバイスの構成プロパティを実装します。構成ファイル内に実装されるすべての構成プロパティは、それぞれが外部に接触している個別のデータ項目となるのではなく、「値ファイル」と呼ばれる 1 つまたは 2 つのデータのブロックにまとめられます。値ファイルにはさまざまな長さの構成プロパティ・レコードが結合されています。各値ファイルは、アプリケーションがアクセスできるデバイス内のメモリ領域に、連続したバイトとして収まる必要があります。2 つの値ファイルがあるときは、1 つのファイルに書き込み可能な構成プロパティが含まれ、もう 1 つのファイルに読み取り専用データが含まれます。ネットワーク管理ツールが値ファイル内のデータ項目にアクセスするのを許可するために、*template file*「テンプレート・ファイル」も用意されています。これは、値ファイル内の要素を記述したテキスト文字の配列です。

構成プロパティを構成ファイルとして実装することの利点は、構成プロパティのサイズや構成プロパティの数に制限が課されないことです。ただし、使用できるデバイス上のメモリ領域によって制限が課されることはあります。構成ファイルを使用した場合の欠点は、構成ファイルとして実装されている構成プロパティに対して、他のデバイスが接続やポーリングをすることがで

きないことです。このため、構成ファイル内に実装されている構成プロパティを変更するにはネットワーク管理ツールが必要となり、構成ファイル内に実装されている構成プロパティが更新されても、イベントは自動的に生成されません。アプリケーションを使うと、構成プロパティが更新されたときにネットワーク管理ツールによってデバイスをリセットするか、機能ブロックを無効にするか、またはデバイスをオフラインにすることで強制的に更新が通知されるように設定することができます。ただし、構成プロパティが変更された後で行われる通知はリセットの通知だけです。別の方法としては、アプリケーションが LONWORKS ファイル転送プロトコル（FTP）を利用して構成ファイルのアクセスを実装し、**stop_transfer()**関数をモニターすることによって、通知を強制することもできます。このオプションを使用する場合、FTP サーバ・コードに追加のコード領域が必要になります。

構成プロパティを構成ファイルの一部として実装するには、「ファイル内の構成プロパティの宣言」で説明している **cp_family** のシンタックスを使用して宣言します。

ファイル内の構成プロパティの宣言

構成ファイル内に実装する構成プロパティは、CP ファミリを使用して宣言します。CP ファミリの宣言は、「メタ」宣言として捉えることができます。この宣言では、プログラム内で使用する型のコンストラクトを定義します。型やその他の設定が同じでありながら、異なるネットワーク変数、機能ブロック（第 5 章「機能ブロックを使ったデバイス・インターフェースの実装」を参照してください）、またはデバイス自体に個別に適用される多くの構成プロパティを、ここでまとめて定義します。CP ファミリは、ゼロのメンバ、1 つのメンバ、または複数のメンバを持つことができます。後述されるように、CP ファミリのメンバが宣言されるまでは、コードやデータは生成されません。この点において、CP ファミリは C 言語の **typedef** に似ていますが、型定義以外の機能も備えています。ファミリのメンバはすべて同じ属性を共有しますが、値を共有することはありません。

CP ファミリを宣言するためのシンタックスは以下のとおりです。

[**const**] *type* **cp_family** [*cp-modifiers*] *identifier* [**=** *initial-value*] **;**

コード例

```
SCPTgain cp_family cpGain = { 2, 3 };
```

CP ファミリの「型」は、**int** や **char** などの標準の C の型として扱うことはできません。宣言では、リソース・ファイルの構成プロパティ型（CPT）を使用する必要があります。構成プロパティ型には、標準構成プロパティ型（SCPT）とユーザ構成プロパティ型（UCPT）があり、現在 200 以上の SCPT 定義が用意されています。UCPT を使用すると、特定のメーカーに対応した独自の型を作成できます。SCPT 定義は **standard.typ** ファイルに格納されています。このファイルは NodeBuilder ツールに含まれている標準リソース・ファイル・セットの一部です。他にも UCPT 定義を含む多くの類似のリソース・ファイルが多数用意されており、NodeBuilder リソース・エディタによってコンピュータ上で管理されます。詳細については『NodeBuilder User's Guide』を参照してください。

構成プロパティ型も ANSI C の **typedef** に似ていますが、それ以外にも多くの機能を提供します。構成プロパティ型は、標準化された型の意味も定義しま

す。リソース・ファイル内の構成プロパティ定義には、デフォルト値、有効な最小値と最大値、無効な値（オプション）、ならびに地域や言語に対応した文字列リファレンス、追加のコメント、および構成プロパティ型に関連付けられる単位文字列が含まれています。

cp-modifiers は **cp_info** キーワードで始まり、その後にかっこで囲まれたオプションのキーワードのリストが続きます。キーワードとその意味については、『Neuron C Reference Guide』の「Configuration Property and Network Variable Declarations」の章を参照してください。。

CP ファミリの宣言における *initial-value* はオプションです。*initial-value* が宣言に指定されていない場合は、リソース・ファイルで指定されているデフォルト値が使用されます。指定されている *initial-value* はファミリに属する 1 つのメンバの初期値ですが、ファミリ・メンバがインスタンス化されるたびにこの初期値がコンパイラによって複製されます。

CP ファミリ・メンバの初期化ルールは以下に示すとおりです。この初期化ルールは、リンク・イメージの値ファイルや、デバイス・インターフェース・ファイルに格納されている値ファイルに初期値を設定するときに使用します。ネットワーク管理ツールは初期値を「デフォルト値」として設定し、時には構成プロパティ（または構成プロパティのサブセット）をデフォルト値に戻すこともできます。構成管理ツールがどのように構成プロパティのデフォルト値を使用するかについては、『LonMaker User's Guide』など、ネットワーク管理ツールのドキュメントを参照してください。以下の初期化ルールでは、構成プロパティにあてはまる最初のルールをコンパイラが使用します。

1 構成プロパティがインスタンス化において明示的に初期化される場合は、これが、使用される初期値になります。

2 構成プロパティが CP ファミリの宣言において明示的に初期化される場合は、ファミリの初期化子が使用されます。

3 構成プロパティが機能ブロックを利用し、機能ブロックを定義している機能プロファイルが、関連付けられている構成プロパティ・メンバのデフォルト値を指定している場合は、機能プロファイルのデフォルトが使用されます。

4 構成プロパティの型がデフォルト値を定義している場合は、そのデフォルト値が初期値として使用されます。このルールは型を継承する構成プロパティ型には適用されません。本章の「型を継承する構成プロパティ」を参照してください。

5 上のルールのいずれの初期値も使用できない場合には、すべてゼロの値が使用されます。

cp_family 宣言は繰り返しが可能です。宣言は 2 回以上繰り返すことができ、重複する宣言がすべての点において一致する限り、コンパイラは単一の宣言として扱います。

構成ネットワーク変数の宣言

構成ネットワーク変数の宣言のシンタックスは、第 3 章「ネットワーク変数を使ったデバイス間通信」で述べているように、非構成ネットワーク変数の宣言のシンタックスに似ています。

構成ネットワーク変数を宣言するための完全なシンタックスは、以下のとおりです。宣言は、ネットワーク変数の型の後に **config_prop** キーワードを含

めることで、他のネットワーク変数の宣言とは区別されます。**config_prop** キーワードを **cp** と省略することもできます。

network input [*netvar-modifier*] [*class*] *type* **config_prop** [*cp-modifiers*] [*connection-info*] *identifier* [= *initial-value*] **;**

network input [*netvar-modifier*] [*class*] *type* **config_prop** [*cp-modifiers*] [*connection-info*] *identifier* **[***array-bound***]** [= *initializer-list*] **;**

コード例

```
network input SCPTupdateRate config_prop nciUpdateRate;
network input SCPTbypassTime cp nciBypassTime = ...
```

このシンタックスの *netvar-modifier* と *class* の部分は前の章で詳しく説明しました。これらは他のネットワーク変数と同じように構成ネットワーク変数にも適用されます。ただし、*class* を **config** にすることはできません（**config** ネットワーク変数は、完全に管理された構成プロパティではなく、手動で管理されます。**config** キーワードは古いキーワードなので、新規の開発で使用することは奨励できません。これは古いアプリケーションを Neuron C バージョン 2 で使用できるようにするためのものです）。

構成 CP ファミリのメンバと同じように、構成ネットワーク変数はリソース・ファイル内の構成プロパティ型によって定義されている「型」を使用して宣言する必要があります。型は標準（SCPT）型でもユーザ（UCPT）型でも構いません。**config_prep** キーワードの後にオプションで指定できる *cp-modifiers* 節も、本章で前述した CP ファミリの宣言と全く同じです（*cp-modifiers* シンタックスと意味については、『Neuron C Reference Guide』を参照してください）。

構成ネットワーク変数の *connection-info* は前章で説明したネットワーク変数の接続情報と全く同じです。他のネットワーク変数と同様、構成ネットワーク変数は配列にすることができます。配列の各要素は構成プロパティによって個別に扱われます。

構成ネットワーク変数の宣言には、他のネットワーク変数の宣言と同様に *initial-value* または *initializer-list* を含めることができます。他のネットワーク変数とは異なり、構成ネットワーク変数は、それ自体にネットワーク変数プロパティ・リストを含めることができません。つまり、他の構成プロパティに適用される構成プロパティを定義することはできません。

ネットワーク変数の配列を構成プロパティとして使用するときは、そのネットワーク変数の配列に対するコンパイラの初期化ルールに特別な注意を払う必要があります。配列の要素は他の変数や配列変数の宣言の場合と同じように、宣言内で初期化できます。配列の要素が初期化されない場合には、デフォルトによりゼロに初期化されます。ただし、配列の各要素はプロパティ節でプロパティとして指定したときにも初期化できます。この宣言は、宣言内の初期化よりも優先されますが、優先されるのはそのプロパティ節に指定されている要素のみです。

構成プロパティのインスタンス化

cp_family の宣言は、宣言の結果として実際の変数が作成されることはないという点において、C 言語の **typedef** に似ています。型定義は、その後の宣言で使用されたときにインスタンス化されます。このとき宣言自体が別の

typedef であってはなりません。この時点で、変数は「インスタンス化」されます。つまり変数が宣言され、メモリが確保されて変数に割り当てられます。その後、変数はプログラムの実行可能コード内の式で使用できるようになります。

CP ファミリ・メンバのインスタンス化は、CP ファミリの宣言の識別子がプロパティ・リストで使用されるときに行われます。ただし、構成ネットワーク変数は宣言時に既にインスタンス化されています。構成ネットワーク変数の場合、プロパティ・リストは構成プロパティとオブジェクトまたは適用先のオブジェクト間の関係を識別するためだけに使用されます。

構成プロパティは、デバイス、機能ブロック、またはネットワーク変数に適用できます。いずれの場合にも、構成プロパティは「プロパティ・リスト」を通じて対応するオブジェクトに適用されます。デバイスとネットワーク変数のプロパティ・リストについては、次の各セクションで説明します。機能ブロックのプロパティ・リストについては、第 5 章「機能ブロックを使ったデバイス・インターフェースの実装」で説明します。1 つのオブジェクトに適用できる構成ファイルは、SCPT 型、UCPT 型にかかわらず 1 つだけです。このことはネットワーク変数、機能ブロック、またはデバイス全体のいずれのオブジェクトにも当てはまります。

デバイスのプロパティ・リスト

デバイスのプロパティ・リストは、適用されるデバイスの CP ファミリ宣言と構成ネットワーク変数宣言から成る構成プロパティのインスタンスを宣言します。デバイスのプロパティ・リストの完全なシンタックスは以下のとおりです。

device_properties { *property-reference-list* **} ;**

property-reference-list :
property-reference-list **,** *property-reference*
property-reference

property-reference :
property-identifier [= *initializer*] [*range-mod*]
property-identifier [*range-mod*] [= *initializer*]

range-mod : **range_mod_string (** *concatenated-string-constant* **)**

property-identifier : *identifier* **[** *constant-expression* **]**
identifier

コード例

```
SCPTlocation cp_family cpLocation;

device_properties {
      cpLocation = { "Unknown" }
};
```

デバイスのプロパティ・リストは **device_properties** のキーワードで始まり、コンマで区切られたプロパティ参照のリストがその後に続きます。各プロパティ参照には、既に宣言されている CP ファミリの名前、または構成ネットワーク変数の名前を指定する必要があります。ネットワーク変数が配列の場

合には、単一の配列要素のみをデバイスのプロパティとして選択できるため、その場合にはプロパティ参照の一部として配列のインデックスを含める必要があります。

コード例

```
network input SCPTlocation cp cpLocation[5];

device_properties {
      cpLocation[0] = { "Unknown" }
};
```

property-identifier の後にはオプションの *initializer* と *range-mod* が指定される場合があります。これらの要素については、『Neuron C Reference Guide』の「Configuration Properties and Network Variables」の章で詳しく説明しています。

デバイスのプロパティ・リストは、関数の宣言、タスクの宣言、またはグローバル・データの宣言と同様、ファイル・レベルのスコープで指定します。Neuron C プログラムではデバイスのプロパティ・リストを複数使用することができます。Neuron C コンパイラはこれらのリストを結合して、1 つのデバイス・プロパティ・リストを作成します。ただし、デバイスに対応させる SCPT 型または UCPT 型の構成プロパティは複数使用することはできません。

デバイスのプロパティ・リストにおいて同じ型の特定の構成を 2 つの異なるモジュールが指定した場合には、コンパイル時にエラーが発生します。

コード例

```
UCPTsomeDeviceCp cp_family cpSomeDeviceCp;
SCPTlocation cp_family cpLocation = {""};

device_properties {
      cpSomeDeviceCp,
      cpLocation = { "Unknown" }
            // This instantiation overrides the
            // empty string initializer with its own
};
```

ネットワーク変数のプロパティ・リスト

ネットワーク変数のプロパティ・リストは、そのネットワーク変数に適用される CP ファミリの宣言と構成ネットワーク変数の宣言から成る構成プロパティのインスタンスを宣言します。構成ネットワーク変数のプロパティ・リストの完全なシンタックスは以下のとおりです。

nv_properties { *property-reference-list* **}**

property-reference-list :

property-reference-list **,** *property-reference*
property-reference

property-reference :

property-identifier [= *initializer*] [*range-mod*]
property-identifier [*range-mod*] [= *initializer*]

range-mod : **range_mod_string (** *concatenated-string-constant* **)**

property-identifier : [*property-modifier*] *identifier* **[** *constant-expression* **]**
[*property-modifier*] *identifier*

property-modifier : **static** | **global**

コード例

```
// CP for heartbeat and throttle (default 1 min each)
SCPTmaxSndT cp_family cpMaxSendT = { 0, 0, 1, 0, 0 };
SCPTminSndT cp_family cpMinSendT = { 0, 0, 1, 0, 0 };

// NV with heartbeat and throttle:
network output SNVT_lev_percent nvoValue
nv_properties {
      cpMaxSendT,
      // override default for minSendT to 30 seconds:
      cpMinSendT = { 0, 0, 0, 30, 0 }
};
```

ネットワーク変数のプロパティ・リストは **nv_properties** キーワードで始まり、デバイスのプロパティ・リストと全く同じように、コンマで区切られたプロパティ参照のリストがその後に続きます。各プロパティ参照には、あらかじめ宣言した CP ファミリまたは構成ネットワーク変数の名前を指定する必要があります。シンタックスの残りの部分は前述のデバイスのプロパティ・リストとほとんど同じです。

property-identifier の後には、*initializer* と *range-mod* がオプションとして指定される場合があります。これらのオプションの要素については、『Neuron C Reference Guide』で詳しく説明しています。

同じネットワーク変数に適用される構成プロパティは、SCPT 型、UCPT 型にかかわらず、複数使用することはできません。ネットワーク変数のプロパティ・リストにおいて特定の構成プロパティ型を複数のプロパティに使用すると、コンパイル時にエラーが発生します。

デバイスのプロパティとは異なり、ネットワーク変数のプロパティは 2 つ以上のネットワーク変数間で共有できます。**global** キーワードを使用すると、2 つ以上のネットワーク変数間で共有される CP ファミリ・メンバが作成されます。**static** キーワードを使用すると、ネットワーク変数配列のすべてのメンバ間で共有される CP ファミリ・メンバが作成されます。ただし、配列に属さないその他のネットワーク変数は CP ファミリ・メンバを共有できません。詳細については、後述の「構成プロパティの共有」を参照してください。

プログラムからのプロパティ値へのアクセス

構成プロパティは、他の変数と同じように、プログラムからアクセスできます。例えば、構成プロパティを関数のパラメータとして使用したり、構成プロパティのアドレスを使用することができます。

ただし、式の中で CP ファミリ・メンバを使用するには、どのファミリ・メンバにアクセスするのかを指定する必要があります。これは、1 つの CP ファミリに属する同名の複数のメンバが、異なるネットワーク変数に適用される場合があるためです。ネットワーク変数のプロパティ・リストから構成プロパティにアクセスするためのシンタックスは次のとおりです。

nv-context **::** *property-identifier*

nv-context : *identifier* **[** *index-expr* **]**
identifier

コード例

```
// CP for heartbeat and throttle (default 1 min each)
SCPTmaxSndT cp_family cpMaxSendT = { 0, 0, 1, 0, 0 };
SCPTminSndT cp_family cpMinSendT = { 0, 0, 1, 0, 0 };


// NV with heartbeat and throttle:
network output SNVT_lev_percent nvoValue
nv_properties {
      cpMaxSendT,
      // override default for cpMinSendT to 30 seconds:
      cpMinSendT = { 0, 0, 0, 30, 0 }
};


void f(void)
{
      ...
      if (nvoValue::cpMaxSendT.seconds > 0) {
            ...
      }
}
```

特定の CP ファミリ・メンバはそのメンバの前に付けられる修飾子によって識別されます。この修飾子は *context*（コンテキスト）と呼ばれます。コンテキストの後には、*context operator*（コンテキスト演算子）と呼ばれる 2 つの連続したコロン文字が続き、その後にプロパティの名前を指定します。同じネットワーク変数に適用される同じ構成プロパティ型のプロパティを複数使用することはできないため、各プロパティは特定のコンテキスト内で一意になります。したがって、コンテキストはプロパティを一意に識別します。例えば、10 の要素を持つネットワーク変数の配列 **nva** は、**xyz** という名前の CP ファミリを参照するプロパティ・リストを使用して宣言できます。この場合、**xyz** の CP ファミリには同じ名前を持つ 10 の異なるメンバが存在することになります。そこで、**nva[4]::xyz** や **nva[j]::xyz** のようにコンテキストを追加することで、CP ファミリの各メンバは一意に識別されます。

CP ファミリはデバイスのプロパティとしても使用できるため、デバイス用に定義される特別なコンテキストがあります。デバイスのコンテキストは、先頭にコンテキスト識別子を持たないコンテキスト演算子（2 つの連続するコロン文字）です。

構成ネットワーク変数はその変数識別子を使用して一意にアクセスできますが、CP ファミリ・メンバと同じように、コンテキスト式を使用してアクセスすることもできます。

構成プロパティの高度な機能

このセクションでは、構成プロパティの高度な機能を紹介します。高度な機能の 1 つに、ネットワーク変数の配列のサポートが挙げられます。もう 1 つの機能は、インスタンス化を行う時点での構成プロパティの初期化です。

さらに別の高度な機能に、構成プロパティの共有があります。この機能を利用すると、1 つの構成プロパティを複数のネットワーク変数または機能ブロックに適用できます（機能ブロックの詳細については、第 5 章「機能ブロックを使ったデバイス・インターフェースの実装」を参照してください）。ただし、ネットワーク変数と機能ブロックの両方に 1 つの構成プロパティ（または構成プロパティ・ファミリのメンバ）を適用することはできません。

このセクションで説明する最後の高度な機能は、型を継承する構成プロパティです。構成プロパティ型（CPT）の中には、それ自身の型が、適用先のネットワーク変数によって定義されるものがあります。

配列に適用される構成プロパティ

構成プロパティがネットワーク変数の配列に適用されるとき、コンパイラは配列のメンバごとに構成プロパティを複製します（これは「構成プロパティの共有」で述べているとおり、プロパティが共有される場合には当てはまりません）。4 つの要素を持つ次のようなネットワーク変数の配列を考えてみましょう（各要素はセンサーに対応しているとします）。

```
network output SNVT_volt nvo[4];
```

ここで、各センサーの出力に SCPTmaxSendTime 構成プロパティを用意したいとします。これらの出力を使用して、出力ネットワーク変数の更新の最大間隔（秒単位）を表します。構成プロパティ・ファミリを使用する場合、これは次のように宣言することで実現できます。このように構成プロパティを定義すると、Neuron C コンパイラがネットワーク変数の配列の各要素に対して個別のファミリ・メンバを自動的に作成します。

```
SCPTmaxSendTime cp_family cpMaxSendTime;
network output SNVT_volt nvo[4]
      nv_properties { cpMaxSendTime };
```

もう 1 つの方法として、SCPTmaxSendTime 構成プロパティに個別のネットワーク変数の配列を使用する方法があります。例えば、以下に示すネットワーク変数の配列の宣言では、4 つの要素を持つ **cpMaxSendTime** 配列を指定します。これらの各要素の構成プロパティは、**nvo** 配列内の要素に対応しています。

```
network input cp SCPTmaxSendTime cpMaxSendTime[4];
network output SNVT_volt nvo[4]
      nv_properties { cpMaxSendTime[0] };
```

ここに示すような方法で、構成プロパティに対してネットワーク変数の配列を使用するときは、**nv_properties** 節内の構成プロパティ参照にインデックスを含める必要があります。このインデックスは、「開始インデックス」としてコンパイラが使用するものです。コンパイラは構成プロパティ・ネットワーク変数配列の要素を、プロパティの適用先となる、基のネットワーク変数配列の要素に自動的に割り当てます。また、必要な要素のメンバが揃っているかどうかも確認します。次のコードはエラーになります。

```
network input cp SCPTmaxSendTime cpMaxSendTime[3];
                          // Insufficient # of elements
network output SNVT_volt nvo[4]
      nv_properties { cpMaxSendTime[0] };
```

nv_properties 節内の構成プロパティ参照のインデックスは「開始」インデックスです。このインデックスをゼロにする必要はありません。例えば、nvo1

と nvo2 という名前の 2 つのネットワーク変数の配列があり、それぞれの配列が **SCPTmaxSendTime** プロパティを含んでいるとすると、次のような宣言をします。ここでは、構成プロパティ・ネットワーク変数の配列の一部が出力ネットワーク変数の 1 つの配列に使用され、残りの部分が出力ネットワーク変数のもう 1 つの配列に使用されます（この例では **cpMaxSendTime** 配列のすべてのメンバが使用されていますが、すべて使う必要はありません）。

```
network input cp SCPTmaxSendTime cpMaxSendTime[7];
network output SNVT_volt nvo1[4]
      nv_properties { cpMaxSendTime[3] };
network output SNVT_vold nvo2[3]
      nv_properties { cpMaxSendTime[0] };
```

インスタンス化における構成プロパティの初期化

固定の型の構成プロパティは、宣言時に初期化できます。ネットワーク変数の配列を構成プロパティの配列として使用するときは、次の例のようになります。以下に示す 4 つの構成プロパティは、いずれも 10 の値（起動時の遅延値は秒数です）に初期化されます。

```
network input cp SCPTpwrUpDelay nvcp[4] = {10, 10, 10, 10};
```

構成プロパティをインスタンス化時に初期化する必要はありませんが、こうすると便利な場合があります。例えば、2 つのネットワーク変数 **nva** と **nvb** を宣言し、構成プロパティ **nvcp[0]**を **nva** に関連付け、**nvcp[1]**を **nvb** に関連付けたいとします。さらに、これら 2 つのインスタンスにおいて、起動時の遅延プロパティをそれぞれ 5 秒と 10 秒にしたいとします。次に、宣言の初期値よりも **nva** のプロパティのインスタンス化における初期値の方を「優先」させ、同時に **nvcp[1]**の 10 への以前の初期化をそのまま利用したいとします。

```
network output SNVT_volt nva = 0
      nv_properties { nvcp[0] = 5 };
network output SNVT_amp nvb = 0
      nv_properties { nvcp[1] };
```

最後に、2 つのメンバを持つネットワーク変数配列 **nvc** を考えます。ここでは **nvcp[2]**と **nvcp[3]**をそれぞれ **nvc[0]**と **nvc[1]**の構成プロパティとして使用します。また、これらの構成プロパティをどちらも 60 秒に初期化したいとします。この場合は次のように宣言します。

```
network output SNVT_count nvc[2] = {100, 100}
      nv_properties { nvcp[2] = 60 };
```

nvc ネットワーク変数は配列であるため、**nvcp[2]**のプロパティ参照は、コンパイラがプロパティの自動割り当てを行うための「開始点」として扱われます（前述の「配列に適用される構成プロパティ」を参照してください）。コンパイラは自動的に **nvcp[2]**への参照を複製し、**nvcp[0]**に適用します。そして、**nvc** 配列のその後の各要素にも複製（**nvcp[3]→nvc[1]**など）が行われます。このとき初期化値も複製されます（したがって、この場合には **nvcp[3]** も 60 に初期化されます）。このため、各要素の構成プロパティに異なる初期値を適用することはできません。

構成プロパティの型の中には（SCPTdefOutput など）「型を継承する」ものがあります。つまり、SCPT の定義自体は構成プロパティのデータ型を指定しません。その代わりに、構成プロパティのデータ型は適用先のネットワーク変数から継承されます。この場合、許可される唯一の明示的初期化は、宣言ではなく、プロパティ・リスト内のインスタンス化になります。この方法

については、本章で後述する「型を継承する構成プロパティ」を参照してください。

構成プロパティの共有

通常、構成プロパティのインスタンス化は、1 つのデバイス、機能ブロック、またはネットワーク変数に固有のものになります。例えば、5 つの異なるネットワーク変数のプロパティ・リストに含まれている CP ファミリは 5 つのインスタンス化を持ち、各インスタンス化は 1 つのネットワーク変数に固有のものになります。同様に、同じ CP ファミリ名をプロパティ・リストに持つ 5 つの要素のネットワーク変数の配列は、CP ファミリの 5 つのメンバをインスタンス化し、各インスタンスがネットワーク変数配列の各要素に適用されます。

インスタンス化の動作は **static** キーワードと **global** キーワードを使用して変更できます。**global** キーワードを使用すると、1 つの CP ファミリのメンバが、その CP ファミリ名をプロパティ・リストに含むすべてのネットワーク変数間で共有されるようになります（CP ファミリにはこのようなグローバル・メンバを 1 つだけ指定でき、そのメンバは、メンバをプロパティ・リストでインスタンス化するすべてのネットワーク変数間で共有されます）。

static キーワードを使用すると、ネットワーク変数配列のすべての要素間で 1 つの CP ファミリが共有されますが、**static** メンバは配列の外では共有されません。

コード例

```
// CP for throttle (default 1 minute)
SCPTmaxSndT cp_family cpMaxSendT = { 0, 0, 1, 0, 0 };

// NVs with shared throttle:
network output SNVT_lev_percent nvoValue1
      nv_properties {
            global cpMaxSendT
      };

network output SNVT_lev_percent nvoValue2
      nv_properties {
            global cpMaxSendT // the same as the one above
      };

network output SNVT_lev_percent nvoValueArray[10]
      nv_properties {
            static cpMaxSendT // shared among the array
                              // elements only
      };
```

上記の記述はインスタンス化と共有 CP ファミリ・メンバに関するものですが、構成ネットワーク変数も似たような方法を使用して共有することができます。**static** キーワードを配列のプロパティ・リスト内で使用して、ネットワーク変数配列のメンバ間で構成ネットワーク変数を共有します。**global** キーワードは構成ネットワーク変数のプロパティ・リスト内で使用して、2 つ以上のネットワーク変数間でプロパティを共有します。

この点において構成ネットワーク変数と CP ファミリ・メンバの唯一の違いは、構成ネットワーク変数は **global** キーワードを使わなければ 2 つ以上のプロパティ・リストで使用できないことです。これは、ネットワーク変数のイ

ンスタンスが 1 つしかないためです（一方、CP ファミリは複数のインスタンスを持つことができます）。

デバイスに適用される構成プロパティは、1 つのアプリケーションにつき 1 つのデバイスしかないため、共有できません。

型を継承する構成プロパティ

完全な型定義を含む代わりに、適用するネットワーク変数の型定義を使用して構成プロパティの型を定義することができます。別の変数型を使用する構成プロパティの型は、「型を継承する構成プロパティ」と呼ばれます。型を継承する構成プロパティの CP ファミリ・メンバがプロパティ・リストに含まれるとき、CP ファミリ・メンバのインスタンス化はネットワーク変数の型を使用します。同様に、構成ネットワーク変数も型を継承できます。ただし、構成ネットワーク変数の配列には、配列の各要素は同じ型を継承する必要があるという制限があります。

型を継承する構成プロパティの型はインスタンス化が行われるまでわからないため、構成プロパティの初期化子のオプションは、宣言内ではなく、プロパティ・リスト内で指定する必要があります。同様に、プロパティの異なるインスタンス化には異なる *range-mod* 文字列が適用される可能性があるため、型を継承する構成プロパティに対しては、宣言内でなく、プロパティ・リスト内に *range-mod* オプションを指定する必要があります。

型を継承する構成ネットワーク変数を共有する場合（前セクションを参照してください）は、同じ型のネットワーク変数間のみで共有できます。

型を継承する構成プロパティは、デバイス・プロパティとしては使用できません。これは、継承する型がデバイスにないためです。

型を継承する構成プロパティの代表的な例に、**SCPTdefOutput** 構成プロパティ型があります。**SFPTopenLoopSensor** 機能プロファイルは **SCPTdefOutput** 構成プロパティをオプションの構成プロパティとしてリストします。この構成プロファイルの中で、センサーの主要なネットワーク変数のデフォルト値を定義します。ただし、**SFPTopenLoopSensor** 機能プロファイル自体は主要なネットワーク変数の型を定義しません。

次の例では、オプションの **SCPTdefOutput** 構成プロパティを使用して、**SFPTopenLoopSensor** 機能ブロックを実装します。構成プロパティは、以下に示すように、適用先のネットワーク変数（この場合には **SNVT_amp**）から型を継承します。

コード例

```
SCPTdefOutput cp_family cpDefaultOutput;

network output SNVT_amp nvoAmpere nv_properties {
      cpDefaultOutput = 123
};

fblock SFPTopenLoopSensor {
      nvoAmpere implements nvoValue;
} fbAmpereMeter;
```

初期値（123）は宣言内ではなく、構成プロパティのインスタンス化でのみ指定できます。**cpDefaultOutput** は型を継承する構成プロパティなので、インスタンス化されるまで型がわからないためです。

型変更可能なネットワーク変数のための、型を継承する構成プロパティ

型を継承する構成プロパティは、型を変更できるネットワーク変数と組み合わせることもできます。このようなネットワーク変数の型は、デバイスがネットワークにインストールされたときに、ネットワーク・インテグレータによって動的に変更できます。

コード例

```
SCPTdefOutput cp_family cpDefaultOutput;
SCPTnvType cp_family   cpNvType;

network output changeable_type SNVT_amp nvoValue
      nv_properties {
            cpDefaultOutput = 123,
            cpNvType
      };

fblock SFPTopenLoopSensor {
      nvoValue implements nvoValue;
} fbGenericMeter;
```

nvoValue 主要ネットワーク変数は、型の変更ができますが、デフォルトの型（前の例では **SNVT_amp**）を実装する必要があります。型を継承する **SCPTdefOutput** 構成プロパティは、初期の型から型情報を継承するため、**cpDefaultOutput** の初期化子はこのインスタンス化に固有である必要があります。さらに、初期化子はこの初期の型に対して有効な型でなければなりません。

ネットワーク・インテグレータが、ランタイム時に基になるネットワーク変数の型を変更した場合（例えば **SNVT_volt** など）、この構成プロパティを読み書きするときに新しい型に対応するフォーマット･ルールを適用するのは、ネットワーク管理ツールの責任になります。ネットワーク管理ツールは、型を継承する構成プロパティに、ネットワーク変数の新しい型（つまり構成プロパティの新たに継承された型）に対応する適切な初期値を設定しなければなりません。

5

機能ブロックを使ったデバイス・インターフェースの実装

本章では、機能ブロックを使用してデバイスにタスク指向のインターフェースを提供する方法について説明します。機能ブロックを使用すると、ネットワーク変数や構成プロパティをグループ化し、1 つのタスクとして実行することができます。

概要

LONWORKS デバイスのデバイス・インターフェースは、機能ブロック、ネットワーク変数、および構成プロパティで構成されています。各「機能ブロック」はネットワーク変数と構成プロパティから成り、全体で 1 つのタスクを実行することができます。これらのネットワーク変数と構成プロパティは、「機能ブロック・メンバ」と呼ばれます。

機能ブロックを使用すると、デバイスを簡単にインストールできるようになります。インテグレータは、デバイスを 1 つのデバイス・アプリケーションとしてではなく、タスク指向の機能ブロックの集まりとして扱うことができます。機能ブロックをネットワーク内の他の機能ブロックと組み合わせると、ネットワーク内の異なる機能ブロックの関係を簡単に見ることができるようになります。例えば、次の図は機能ブロックを使用して実装されている温度制御装置をインテグレータの視点で示したものです。この例は、7 つの機能ブロックと、そのネットワーク接続を示しています。これらの機能ブロックは実際には 2 つのデバイスに実装されますが、機能図には入力センサーがどのように温度制御装置にデータを提供し、それを受けて温度制御装置がどのように出力をアクチュエータとプロセス・モニターに提供するかが明確に示されています。

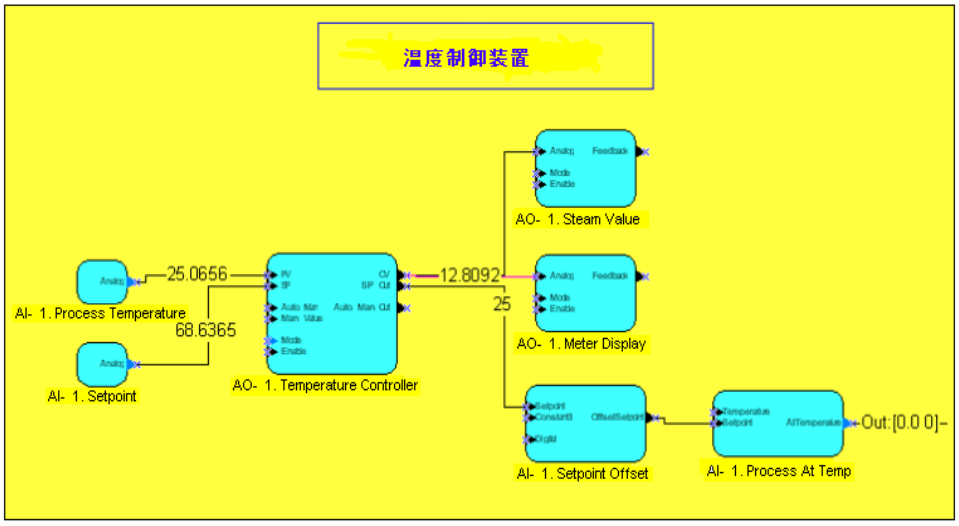


図 5.1　機能ブロックを使用したネットワークの設計

機能ブロックは「機能プロファイル」によって定義されます。機能プロファイルは共通の機能を動作の単位ごとに記述したものです。各機能プロファイルは必須のネットワーク変数とオプションのネットワーク変数、および必須の構成プロパティとオプションの構成プロパティを定義します。各機能ブロックは機能プロファイルのインスタンスを実装します。機能ブロックは、機能プロファイルが定義した必須のネットワーク変数と構成プロパティをすべて実装する必要があり、機能プロファイルが定義したオプションのネットワーク変数と構成プロパティは実装してもしなくても構いません。機能ブロックは、機能プロファイルが定義したものではないネットワーク変数と構成プロパティを実装することもできます。これらは「固有の実装」のネットワーク変数および構成プロパティと呼ばれます。

例えば次の図は、標準機能プロファイル番号 3050 のコンスタント・ライト・コントローラの機能プロファイルを示したものです。このプロファイルは 2 つの必須の入力、1 つの必須の出力、および 1 つのオプションの入力を定義しています。また、1 つの必須の構成プロパティと 8 つのオプションの構成プロパティも定義しています。

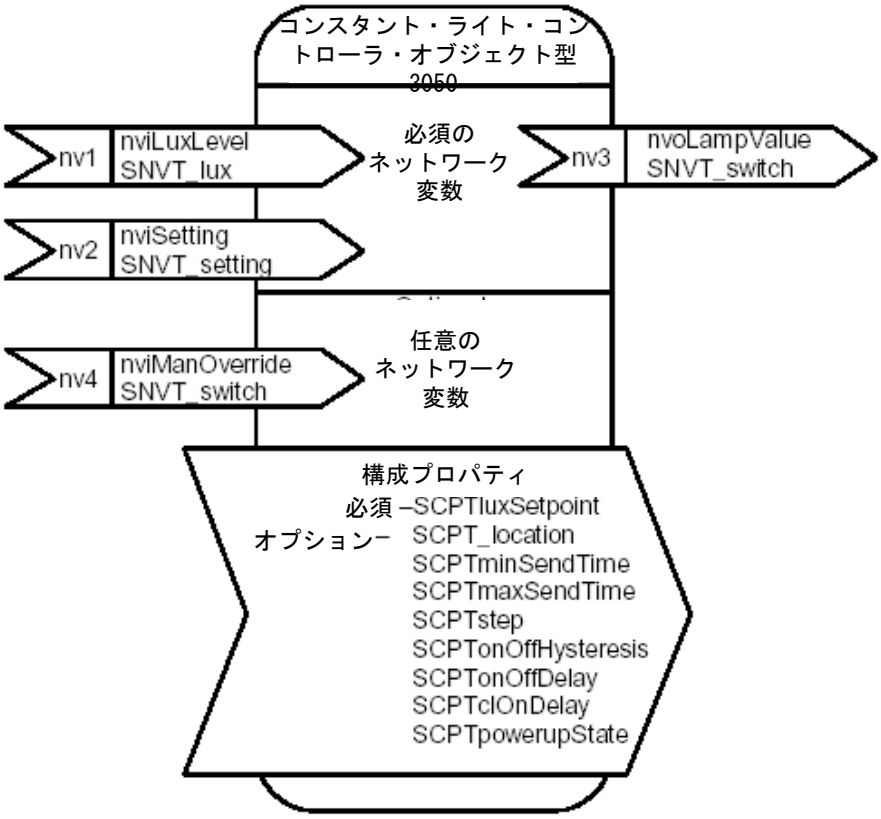


図 5.2　コンスタント・ライト・コントローラの機能プロファイル

機能プロファイルは「リソース・ファイル」で定義します。標準のリソース・ファイルに定義されている標準機能プロファイルを使用することも、独自のリソース・ファイルに独自の機能プロファイルを定義することもできます。リソース・ファイルに定義されている機能プロファイルは「関数プロファイル・テンプレート」とも呼ばれます。

Neuron C アプリケーション内で機能ブロックを宣言するには、**fblock** 宣言を使用します。これらの宣言については、本章で詳しく説明します。

機能ブロックを宣言しても、コンパイラが実行可能コードを生成するようなことはありません。ただし、機能ブロックのさまざまな要件をサポートするためのデータ構造が作成されます。機能ブロックは主にネットワーク変数と構成プロパティ間の関連付けを行います。コンパイラは次にこれらの関連付けを使用して、デバイスおよびデバイスに関連付けられているデバイス・インターフェース・ファイル（拡張子.xif）内に、自己記録（SD）データと自己識別（SI）データを作成します。

デバイス・インターフェース・ファイル内の機能ブロック情報（SD データと SI データ）は、デバイスに含まれている機能ブロックの存在と名前をネット

ワーク管理ツールに伝えます。ここには、デバイス内のどのネットワーク変数と構成プロパティが各機能ブロックのメンバであるかという情報も含まれています。

機能ブロックの宣言

機能ブロックを宣言するための完全なシンタックスは以下のとおりです。

fblock *FPT-identifier* **{** *fblock-body* **}** *identifier* [*array-bounds*] [*ext-name*] [*fb-property-list*] **;**

array-bounds : **[** *const-expr* **]**

fblock-body : *fblock-member-list* [**;** *director-function*]

fblock-member-list : *fblock-member-list* **;** *fblock-member*
fblock-member

fblock-member : *nv-reference* **implements** *member-name*
nv-reference impl-specific

impl-specific : **implementation_specific (** *const-expr* **)** *member-name*

nv-reference : *nv-identifier array-index*
nv-identifier

array-index : **[** *const-expr* **]**

director-function : **director** *identifier* **;**

ext-name : **external_name (** *C-string-const* **)**
external_resource_name (*C-string-const* **)**
external_resource_name (*const-expr* **:** *const-expr* **)**

機能ブロックの宣言は、**fblock** キーワードで始まり、リソース・ファイルで定義されている機能プロファイル名がその後に続きます。機能ブロックは機能プロファイルの実装です（プロファイルを「インスタンス化」します）。機能プロファイルは抽象的なネットワーク変数と構成プロパティのメンバ、「機能プロファイル・キー」と呼ばれる一意のキー、およびその他の情報を定義します。ネットワーク変数と構成プロパティのメンバは、必須のメンバとオプションのメンバに分類されます。必須のメンバは必ず実装する必要がありますが、オプションのメンバは実装してもしなくてもかまいません。

機能ブロックの宣言は次にメンバ・リストへと続きます。このメンバ・リストでは、すでにアプリケーションで宣言したネットワーク変数を、プロファイルのネットワーク変数のメンバと関連付けます。**implements** キーワードは、アプリケーションのネットワーク変数をプロファイル・ネットワーク変数のメンバと関連付けます。

必須のプロファイル・ネットワーク変数メンバは、必ず Neuron C プログラム内の実際のネットワーク変数によって実装される必要があります。各ネットワーク変数（またはネットワーク変数の配列の場合には各配列の要素）は、1 つのプロファイル・メンバしか実装できず、1 つの機能ブロックとしか関連付けることができません。

コード例

```
network output SNVT_amp nvoAmpere;

fblock SFPTopenLoopSensor {
      nvoAmpere implements nvoValue;
} fbAmpereMeter;
```

Neuron C プログラムは、プロファイルが定義したメンバのリストにはない追加のネットワーク変数を機能ブロックに実装することもできます。プロファイルの範囲を越えたこのような追加のネットワーク変数のメンバは、*implementation-specific*「固有の実装」メンバと呼ばれます。メンバ・リスト内のこれらの追加のメンバは、implementation_specific キーワードに続いて一意のインデックス番号、一意の名前をメンバ・リストに指定することで宣言します。機能プロファイル内の各ネットワーク変数は、このインデックス番号とメンバ名を各プロファイル・ネットワーク変数のメンバに代入します。プロファイルが既に使用しているインデックス番号またはメンバ名を固有の実装のメンバに使用することはできません。

コード例

```
network output SNVT_amp nvoAmpere;
network output polled SNVT_time_stamp nvoInstallDate;


fblock SFPTopenLoopSensor {
      nvoAmpere implements nvoValue;
      nvoInstallDate implementation_specific(128)
                  nvoInstall;
} fbAmpereMeter;
```

上の例では、**SFPTopenLoopSensor** 機能プロファイルの必須ネットワーク変数 **nvoValue** を実装し、さらに **nvoInstall** というメンバ名を持つ固有の実装の **SNVT_time_stamp** ネットワーク変数を追加します。通常、メンバ名（この例では **nvoInstall**）は、本章の「プログラムからの機能ブロックのメンバとプロパティへのアクセス」で後述されているように、メンバのネットワーク変数を参照するために使用されます。ただし、ネットワーク変数名 **nvoInstallDate** は、ネットワーク変数の自己記録（SD）データとデバイス・インターフェース・ファイルによってネットワーク・インテグレータが参照できる名前になります。ネットワーク管理ツールでは、プロファイルの定義を使うたびに **nvoInstall** の名前が機能ブロックのメンバとして表示されます。

メンバ・リストの最後には、ディレクタ関数の指定を許可するオプションの項目があります。ディレクタ関数の指定は **director** キーワードで始まり、関数の名前を示す識別子がそれに続き、セミコロンで終わります。

コード例

```
network output SNVT_amp nvoAmpere;

extern void MeterDirector(unsigned fbIdx, unsigned cmd);

fblock SFPTopenLoopSensor {
      nvoAmpere implements nvoValue;
      director MeterDirector;
} fbAmpereMeter;
```

ディレクタの詳細については、本章の「ディレクタ関数」を参照してください。

メンバ・リストの後には、機能ブロック自身の名前を使用した機能ブロックの宣言が続きます。機能ブロックは単一の宣言にすることも、1 次元の配列にすることもできます。

以下の例に示すように、機能ブロックが配列として実装される場合、その機能ブロックのメンバを実装する各ネットワーク変数は、少なくとも同じサイズの配列として宣言する必要があります。配列ネットワーク変数の要素を使用して **fblock** 配列のメンバを実装するときは、配列要素の範囲にある最初のネットワーク変数配列要素の開始インデックスを implements 文で指定する必要があります。Neuron C コンパイラは自動的に残りのネットワーク変数配列要素を **fblock** 配列要素に追加し、各要素を順番に配置します。

コード例

```
network output SNVT_lev_percent nvoValue[6];

// The following declares an array of four fblocks,
// with "nvoAnalog" members implemented by the
// network variables nvoValue[2] .. nvoValue[5],
// respectively.
fblock SFPTanalogInput {
      nvoValue[2] implements nvoAnalog;
}
```

各機能ブロックには、外部名を指定できます。外部名を指定するには、**external_name** キーワードを使用し、その後に最高 16 文字の文字列をかっこで囲んで指定します。この文字列はネットワーク管理ツールが参照するデバイス・インターフェースの一部として扱われます。

別の方法として、**external_resource_name** キーワードを使用して、リソース・ファイル内の言語文字列を使った外部名を指定することもできます。この場合、デバイスのインターフェース情報にはスコープとインデックスのペア（最初の番号はスコープで、コロンに続く次の番号がインデックス）が含まれます。スコープとインデックスのペアは、リソース・ファイル内の言語文字列を識別します。ネットワーク管理ツールは、この言語文字列にアクセスして、機能ブロックの名前を求めます。スコープとインデックスのペアを使用すると、メモリ要件を削減し、機能ブロックに言語依存名を提供できるようになります。外部名の詳細については、『Neuron C Reference Guide』を参照してください。

コード例

```
#define     NUM_AMMETERS      4

network output SNVT_amp nvoAmpere[NUM_AMMETERS];


fblock SFPTopenLoopSensor {
      nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_AMMETERS]
external_name("AmpereMeter");
```

機能ブロックのプロパティ・リスト

機能ブロックの終わりには、前の章で説明したデバイスのプロパティ・リストやネットワーク変数のプロパティ・リストに似たプロパティ・リストがあります。機能ブロックのプロパティ・リストには、適用される機能プロファイルによって定義された必須のプロパティをすべて含めなければなりません。固有の実装のプロパティは、特別なキーワードなしでリストに追加することができます。1 つの機能ブロックに特定の SCPT 型または UCPT 型のプロパティを複数実装することはできません。

機能ブロックのプロパティ・リストは、機能ブロック全体に適用される必須プロパティとオプションのプロパティのみを含めます。プロファイルの個別の抽象化ネットワーク変数のメンバに適用されるプロパティは、*fb-property-list* ではなく、メンバを実装するネットワーク変数の *nv-property-list* に含める必要があります。

機能ブロックのプロパティ・リストの完全なシンタックスは、次のとおりです。

fb_properties { *property-reference-list* **}**

property-reference-list :
property-reference-list **,** *property-reference*
property-reference

property-reference :
property-identifier [= *initializer*] [*range-mod*]
property-identifier [*range-mod*] [= *initializer*]

range-mod : **range_mod_string (** *C-string-constant* **)**

property-identifier : [*property-modifier*] *identifier* **[** *constant-expression* **]**
[*property-modifier*] *identifier*

property-modifier : **static** | **global**

機能ブロックのプロパティ・リストは **fb_properties** キーワードで始まり、各プロパティ参照がコンマで区切られたリストが続きます。これはデバイスのプロパティ・リストやネットワーク変数のプロパティ・リストと全く同じです。各プロパティ参照の名前は以前に宣言した CP ファミリであるか、既に宣言した構成ネットワーク変数でなければなりません。シンタックスの残りの部分は、前の章で説明したネットワーク変数のプロパティ・リストによく似ています。

property-identifier の後には、オプションの *initializer* が続く場合があります。このオプションが指定されている場合、CP ファミリ・メンバのインスタンス化初期化子により、ファミリの宣言時に指定された初期化子が優先されます。したがって、この方法を使用すると、一部の CP ファミリ・メンバだけを特別に初期化し、残りのファミリ・メンバにはより一般的な初期値を適用することができます。ネットワーク変数が複数の場所で初期化される場合（宣言およびプロパティ・リスト内での使用で初期化される場合）、これらの初期値は一致する必要があります。

機能ブロック・プロパティ・リストのシンタックスの詳細については『Neuron C Reference Guide』を参照してください。

コード例

```
SCPTdefOutput     cp_family cpDefaultOutput;
SCPTbrightness    cp_family cpDisplayBrightness;


network output SNVT_amp nvoAmpere;
network output polled SNVT_time_stamp nvoInstallDate;


fblock SFPTopenLoopSensor {
      nvoAmpere implements nvoValue;
      nvoInstallDate implementation_specific(128)
                  nvoInstall;
} fbAmpereMeter external_name("AmpereMeter")
      fb_properties {
            cpDefaultOutput,        // optional CP
            cpDisplayBrightness    //implementation-spec.
      };
```

この例では、オープン・ループのセンサーを電流計として実装しています。必須のネットワーク変数 **nvoValue** が実装されていますが、オプションのネットワーク変数はありません。本章で前述した固有の実装メンバ **nvoInstall** およびオプションの構成プロパティ **SCPTdefOutput** が実装されており、2 番目の固有の実装の **SCPTbrightness** 構成プロパティも含まれています。

上の例における CP ファミリ名（**cpDefaultOutput** と **cpDisplayBrightness**）には外部との関連は存在しません。これらの名前はデバイスのソース・コード内で、構成プロパティを参照するためだけに使用されます。詳細については、本章で後述する「プログラムからの機能ブロックのメンバとプロパティへのアクセス」を参照してください。

共有機能ブロック・プロパティ

ネットワーク変数のプロパティを共有できるのと同様、機能ブロックのプロパティも 2 つ以上の機能ブロック間で共有できます。**global** キーワードを使用すると、2 つ以上の機能ブロック間で共有される CP ファミリ・メンバが作成されます。このグローバル・メンバはネットワーク変数間で共有されるグローバル・メンバとは異なります。**static** キーワードを使用すると、機能ブロック配列のすべてのメンバ間では共有されますが、配列外の他の機能ブロック間では共有されない CP ファミリ・メンバが作成されます。

例えば、3 つの **SFPTopenLoopSensor** 機能ブロックの配列を使用して実装される 3 位相式の電流計を考えます。ハードウエアには各位相ごとに個別のアンプが含まれていますが、3 つの位相すべてが 1 つのアナログ・デジタル変換器を共有するものとします。したがって各位相には個別の利得係数が存在しますが、3 つの位相すべてに対して 1 つのプロパティを共有してサンプル・レートを指定します。

コード例

```
#define     NUM_AMMETERS      3

SCPTgain    cp_family cpGain;
SCPTupdateRate cp_family cpUpdateRate;


network output SNVT_amp nvoAmpere[NUM_AMMETERS];
```

```
fblock SFPTopenLoopSensor {
      nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_AMMETERS]
external_name("AmpereMeter")
      fb_properties {
            cpGain,
            static cpUpdateRate
      };
```

さらに、同デバイスには電流計の実装をミラーリングした 3 位相式の電圧計が含まれているものとします。また、6 つのメーターすべてに対してローカルで起動される、バイパス・モードの長さを制限する **SCPTbypassTime** 構成プロパティが存在するものとします。

次の例は、6 つのメーターすべてと、構成プロパティを参照するすべての **fblock** 間で共有されるグローバルの **SCPTbypassTime** 構成プロパティ、および対応する **fblock** 配列のメンバ間で共有される 2 つの静的 **SCPTupdateRate** 構成プロパティを実装しています。

コード例

```
#define     NUM_PHASES  3

SCPTgain       cp_family cpGain;
SCPTupdateRate cp_family cpUpdateRate;
SCPTbypassTime cp_family cpBypassTime;


network output SNVT_amp nvoAmpere[NUM_PHASES];
network output SNVT_volt nvoVolt[NUM_PHASES];


fblock SFPTopenLoopSensor {
      nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_PHASES]  external_name("AmpereMeter")
      fb_properties {
            cpGain,
            static cpUpdateRate,
            global cpBypassTime
      };


fblock SFPTopenLoopSensor {
      nvoVolt[0] implements nvoValue;
} fbVoltMeter[NUM_PHASES]  external_name("AmpereMeter")
      fb_properties {
            cpGain,
            static cpUpdateRate,
            global cpBypassTime
      };
```

スコープ・ルール

固有の実装のネットワーク変数または構成プロパティを標準の機能プロファイルまたはユーザ機能プロファイルに追加するときは、追加項目のリソース定義のスコープの値が機能プロファイルのスコープの値以下になるようにする必要があります。

例えば、固有の実装のネットワーク変数または構成プロパティを標準機能ブロック（SFPT、スコープ 0）に追加する場合は、標準型（SNVT または SCPT）を使って構成プロパティを定義する必要があります。

2 番目の例として、メーカーのスコープ（スコープ 3）のリソース・ファイルに基づいて機能ブロックを実装する場合は、同じスコープ 3 のリソース・ファイルに定義されている固有の実装のネットワーク変数または構成プロパティを追加できます。また、SNVT または SCPT によって定義されている固有の実装のネットワーク変数または構成プロパティを追加することもできます。

固有の実装のメンバは継承を使用して標準機能プロファイルに追加することができます。これを行うには、次のステップに従います。

1 NodeBuilder リソース・エディタを使用して、継承元の標準機能プロファイルと同じ機能プロファイル・キーを持つユーザ機能プロファイルを作成します。

2 機能プロファイル定義の **Inherit members from scope** を **0** に設定します。これによって、標準機能プロファイルのすべてのメンバがユーザ機能プロファイルの一部になります。

3 新しいユーザ機能プロファイルに基づいて機能ブロックを宣言します。

4 固有の実装のメンバを機能ブロックに追加します。

このようにする代わりに、標準機能プロファイルのメンバから継承する機能プロファイルを作成し、独自のプロファイル固有のメンバを機能プロファイルに追加することもできます。こうすると、固有の実装のメンバを使用するよりも文書化が簡単になり、再使用性も向上します。これを行うには、次のステップに従います。

1 NodeBuilder リソース・エディタを使用して、継承元の標準機能プロファイルと同じ機能プロファイル・キーを持つユーザ機能プロファイルを作成します。

2 機能プロファイル定義の **Inherit members from scope** を **0** に設定します。これによって、標準機能プロファイルのすべてのメンバがユーザ機能プロファイルの一部になります。

3 追加のメンバを新しいユーザ機能プロファイルに追加します。

4 新しいユーザ機能プロファイルに基づいて機能ブロックを宣言します。

プログラムからの機能ブロックのメンバとプロパティへのアクセス

機能ブロックのネットワーク変数と構成プロパティのメンバには、他の変数と同じようにプログラムからアクセスできます。例えば、メンバは式内、関数のパラメータとして、あるいはアドレス演算子または増分演算子のオペランドとして使用できます。機能ブロックのネットワーク変数のメンバまたは機能ブロックのネットワーク変数の構成プロパティにアクセスするには、他の変数と同じように、プログラム内でネットワーク変数を参照します。

ただし、CP ファミリのメンバを使用するには、どのファミリ・メンバにアクセスするのかを指定する必要があります。これは、複数の機能ブロックが同じ CP ファミリのメンバを持つことができるためです。機能ブロックのプロパティ・リストから構成プロパティにアクセスするためのシンタックスは、次のとおりです。

fb-context **::** *property-identifier*

fb-context : *identifier* **[** *index-expr* **]**
identifier

特定の CP ファミリのメンバは、その前に付けられている修飾子によって識別されます。この修飾子は「コンテキスト」と呼ばれます。コンテキストの後には 2 つの連続するコロン文字に続いて、プロパティの名前を指定します。同じ機能ブロックに適用される、同じ SCPT 型または UCPT 型のプロパティは 1 つしか使用できないため、各プロパティは特定のコンテキスト内で一意になります。コンテキストはプロパティを一意に識別します。例えば、10 の要素を持つ機能ブロック配列 **fba** は、**xyz** という名前の CP ファミリを参照するプロパティ・リストを使用して宣言できます。これによって、CP ファミリ **xyz** には同じ名前を持つ 10 の異なるメンバが存在することになります。ただし、**fba[4]::xyz** や **fba[j]::xyz** などのコンテキストを追加することで、CP ファミリのメンバは一意に識別されます。

ネットワーク変数のプロパティと同様、構成ネットワーク変数はその変数識別子を経由して一意にアクセスできますが、CP ファミリのメンバと同じように、コンテキスト式を経由してアクセスすることもできます。

また、機能ブロックのネットワーク変数のメンバにも、類似のシンタックスを使用してアクセスできます。機能ブロックのメンバにアクセスするためのシンタックスは、以下に示すとおりです（*fb-context* のシンタックス要素は上記で定義されています）。

fb-context **::** *member-identifier* [**[** *index-expr* **]**]

機能ブロックのネットワーク変数のメンバのプロパティにも、このシンタックスを使用してアクセスできます。機能ブロックのメンバのプロパティにアクセスするためのシンタックスは、以下に示すとおりです（*fb-context* のシンタックス要素は上記で定義されています）。

fb-context **::** *member-identifier* [**[** *index-expr* **]**] **::** *property-identifier*

コード例

```
#define     NUM_AMMETERS      3

SCPTmaxSndT cp_family cpMaxSendTime;
SCPTminSndT cp_family cpMinSendTime;
SCPTgain    cp_family cpGain;
SCPTupdateRate cp_family cpUpdateRate;

network output SNVT_amp nvoAmpere[NUM_AMMETERS]
      nv_properties {
            cpMaxSendTime,
            cpMinSendTime
      };

fblock SFPTopenLoopSensor {
      nvoAmpere[0] implements nvoValue;
} fbAmpereMeter[NUM_AMMETERS]
external_name("AmpereMeter")
      fb_properties {
            cpGain,
            static cpUpdateRate
      };
```

次の構造体はすべて有効なコード例です。

```
nvoAmpere[2] = 123;

fbAmpereMeter[2]::nvoValue = 123;

fbAmpereMeter[0]::cpGain.multiplier = 2L;

nvoAmpere[2]::cpMaxSendTime.seconds = 30;

fbAmpereMeter[2]::nvoValue::cpMaxSendTime.hours = 0;

z = ((SCPTmaxSendT *)&nvoAmpere[2]::cpMaxSendTime)->days;
```

例に示すように、CP ファミリのメンバにポインタを使用することができますが、構成プロパティは EEPROM に格納されます。これによって、『Neuron C Reference Guide』で**#pragma relaxed_casting_on** 指令について説明されているとおり、コンパイラによって特別なルールが適用されます。

cpGain は静的な構成プロパティなので、次の式は常に TRUE になります。

```
fbAmpereMeter[0]::cpGain.multiplier ==
                    fbAmpereMeter[1]::cpGain.multiplier
```

次の式は誤っているため、コンパイラ・エラーが発生します。

```
//    '.' instead of '::'
fbAmpereMeter[0].cpGain.multiplier = 123;

//    reference of CP family, not CP family member
cpGain.multiplier = 123;

//    '::' instead of '.'
fbAmpereMeter[0]::cpGain::multiplier = 123;
```

Neuron C には、機能ブロック用の組み込みプロパティもいくつか用意されています。組み込みプロパティは以下に示すとおりです（*fb-context* のシンタックス要素は上記で定義されています）。

fb-context **:: global_index**

fb-context **:: director (** *expr* **)**

global_index プロパティは、コンパイラによって代入されるグローバル・インデックスに対応する **unsigned short** の値です。グローバル・インデックスは読み取り専用の値で、範囲は 0（ゼロ）～62 です。各 **fblock** と **fblock** 配列の要素はそれぞれ一意のインデックスを持ちます。**fblock** インデックスの順序は、**fblock** 宣言がコンパイルされる順序と同じになります。

例に示すように **director** プロパティを使用すると、機能ブロックの宣言に使用されているディレクタ関数が呼び出されます。実際のディレクタ関数の最初のパラメータはコンパイラによって自動的に提供され（最初の引数は機能ブロックのグローバル・インデックスです）、上のシンタックスに示されている *expr* はディレクタ関数の 2 番目のパラメータになります。この 2 番目のパラメータは、通常は **unsigned uCommand** と呼ばれますが、コンパイラは特別な解釈を一切行うことなく **unsigned** 型のオプションの値を許可します。

director プロパティは、ディレクタが個々の **fblock** に対して定義されているか、さまざまな **fblock** 間で共有されるか、またはまったく定義されないかにかか

わらず、上記のいずれのケースにも使用できます。最後のケースでは、動作は何も発生しません。

director プロパティと **global_index** プロパティの詳細、およびこれらの使用例については、以下の「ディレクタ関数」を参照してください。

ディレクタ関数

各機能ブロックには「ディレクタ関数」を作成できます。ディレクタ関数とは、**enable**、**disable**、**reset**、**test** など、機能ブロックに関連付けられている動作を提供できる関数です。機能ブロックとの関連付けによって、Node Object 機能ブロックにより定義されている標準のリクエスト関数を簡単に実装できるようになります。これらのリクエスト関数を使用すると、ネットワーク管理ツールはデバイス上のオプションの機能ブロックを有効化、無効化、リセット、またはテストするリクエストをデバイスに送信できるようになります。Node Object を実装すると、ディレクタ関数を使って、これらのリクエストを適切な機能ブロック関数に伝えることができます。NodeBuilder コード・ウィザードによって生成される Node Object の実装には、Node Object Request 入力からの入力に基づいて機能ブロックのディレクタ関数を呼び出すためのコードが含まれています。

ディレクタ関数は以下に示す関数プロトタイプに一致する必要があります。最初のパラメータはディレクタを呼び出すための機能ブロックのグローバル・インデックスで、2 番目のパラメータはディレクタが操作を行うためのコマンド・コードです。

void *director-name* **(unsigned** *fbIndex*, **unsigned** *command*);

ディレクタ関数は、機能ブロックのメンバ・リストの最後にオプションの宣言文を使用することで機能ブロックに追加します。

コード例

```
void myDirector (unsigned fbIndex, unsigned command);

fblock . . . {
      /* Member NVs, "implements" . . . */

      director myDirector;
} myFB;

void myDirector (unsigned fbIndex, unsigned command) {
   . . . /* whatever */
}
```

ディレクタ関数は、機能ブロックを適切に使用する上で重要な役割を果たします。機能ブロックはネットワーク変数とプロパティが集まった 1 つの機能単位です。ネットワーク管理ツールは、特定の機能ブロックを有効化、無効化、リセット、またはテストするリクエストをデバイスに送信できます。デバイスは次にこのリクエストを、機能ブロックを実装している特定の機能ブロックのコードに送ります。ディレクタ関数は、デバイスが自身の機能ブロックを管理し、イベントやコマンドが適切なオブジェクトに送られることを保証する作業を容易にします。

コード例

SFPTnodeObject機能ブロックの実装は、必須メンバである**nviRequest**ネットワーク変数の入力を経由してリクエストを受け取ります。これらのリクエストの例には、RQ_DISABLEDリクエストやRQ_ENABLEDリクエストがあります。これらはオブジェクトに対し、それぞれ無効状態または有効状態に切り替わるようリクエストを出します。これらのリクエストは、Node Object機能ブロック、Node Object機能ブロック以外の個別の機能ブロック、またはデバイスに実装されているすべての機能ブロックに適用できます。
SFPTnodeObjectの実装は、受信したコマンドのスコープを検査し、そのコマンドを正しいディレクタ関数に配信します。

```
when (nv_update_occurs(nviRequest))
{
     if (nviRequest.object_id == MyNodeObj::global_index) {
          // NodeObject must handle this:
          MyNodeObj::director(nviRequest.object_request);
     } else {
          // route the command to the best director:
          ... (see below)
     }
}
```

ネットワーク変数の更新を受信したときは、組み込み **fblock_index_map** 変数を使用して、そのネットワーク変数を含む機能ブロックを判別できます。このマッピング配列には各ネットワーク変数用の要素が含まれており、対応する要素にはネットワーク変数をメンバとする機能ブロックのグローバル・インデックスが含まれています。ネットワーク変数が機能ブロックのメンバでない場合、対応する要素は「機能ブロックではない」ことを意味する 0xFF になります。

機能ブロックのインデックスを指定して、機能ブロックのディレクタを直接呼び出すこともできます。これを行うには、組み込み仮想関数 **fblock_director()**を呼び出します。この関数は個別のディレクタと同じプロトタイプを持つ仮想関数です。**fblock_director()**関数は、呼び出しの対象となる実際のディレクタ関数を自動的に選択します。機能ブロックにディレクタがない場合、**fblock_director()**関数は何も行わずに戻ります。

上記の例でこの構造体を使用すると、受信したコマンドを最も適切なディレクタに配信するコードを書くことができます。

コード例

```
when (nv_update_occurs(nviRequest))
{
     if (nviRequest.object_id == MyNodeObj::global_index) {
          // NodeObject must handle this:
          MyNodeObj::director(nviRequest.object_request);
     } else {
          // route the command to the best director:
          fblock_director(nviRequest.object_id,
                          nviRequest.object_request);
     }
}
```

同様に、ネットワーク変数の更新の適用先となる機能ブロックを管理しているディレクタ関数に通知を行うことで、1 つのタスクによってすべてのネットワーク変数の更新を処理できます。

```
#define     CMD_NV_UPDATE     17

when (nv_update_occurs)
{
      fblock_director(fblock_index_map[nv_in_index],
            CMD_NV_UPDATE);
}
```

ディレクタ関数の使用方法や、ディレクタ関数の 2 番目のパラメータの解釈方法には制限はありません。ディレクタ関数は、Node Object を実装する便利な手段ですが、その他の用途にもこの関数を自由に使用できます。

6

アプリケーション・メッセージを使ったデバイス間通信

本章では、アプリケーション・メッセージの使い方について説明します。アプリケーション・メッセージは、単独で使うことも、ネットワーク変数と併用することもできます。アプリケーション・メッセージの特殊用法であるリクエスト/レスポンス・メカニズムについてもこの章で説明します。その他に、先取りモード、非同期イベント処理や直接イベント処理、メッセージおよびネットワーク変数を使った完了イベント処理、メッセージの認証機能についても説明します。

アプリケーション・メッセージは、デバイス独自（相互運用不可）のインターフェースを作成するために使用します。アプリケーション・メッセージのメカニズムは、外部フレーム・メッセージ（独自のゲートウェイ用）や、明示的にアドレス指定されたネットワーク変数メッセージにも使用できます。

アプリケーション・メッセージについて

アプリケーション・メッセージは、デバイス独自（相互運用不可）のインターフェースを作成する場合に使用します。アプリケーション・メッセージのメカニズムは、外部フレーム・メッセージ（独自のゲートウェイ用）や、明示的にアドレス指定されたネットワーク変数メッセージにも使用できます。

アプリケーション・メッセージで相互運用ができるのは、LONWORKS ファイル転送プロトコルだけです。このプロトコルは、デバイス間、またはデバイスとツール間で大きなブロックのデータを交換することができます。また、構成ファイルの実装にこのプロトコルを使用することもできます。

前の各章で説明したとおり、機能ブロック、ネットワーク変数、および構成プロパティは、相互運用可能なデバイスのオープン・インターフェースを提供します。一方デバイスのインターフェースには、相互運用可能な部分と独自の部分の両方を含めることができます。例えば、デバイスは製造中にだけ使用する独自のインターフェースと、フィールドで使用する相互運用可能なインターフェースを実装できます。

アプリケーション・メッセージには独自のメッセージ・コードが含まれています。このコードの後には可変長のデータ・フィールドが続きます。1 つのメッセージ・コードは、1 バイトから 25 バイトまでのデータを含めることができます。

アプリケーション・メッセージにリクエスト/レスポンス・サービスを使用すると、あるデバイス上のアプリケーションを有効にして、別のデバイス上のアプリケーションがそれに応答するように設定することができます。リクエスト/レスポンス・メカニズムはネットワーク変数のポーリングに似ています。ネットワーク変数がポーリングされると、ポーリングされたデバイスにあるアプリケーション･スケジューラがネットワーク変数の最新値を提供します。ここではアプリケーション・プログラムは介在しておらず、プログラムはポーリングのリクエストがあったことも知りません。これに対して、リクエスト・サービスのアプリケーション・メッセージが送信されたときには、リモート・デバイスにあるアプリケーション・プログラムがリクエスト・メッセージを受信した結果、何らかの動作を行い、それに対するレスポンスとして新しい値を返します。また、リクエスト/レスポンス・サービスは、1 つのデバイス上のアプリケーションが別のデバイス上で動作を開始する手段を提供するため、このサービスを使用してリモート手続き呼び出しを実装することもできます。

アプリケーション･メッセージが使う EEPROM テーブル領域はネットワーク変数が使う領域よりも小さいのですが、同じようなタスクを実行する場合、アプリケーション・メッセージの方がネットワーク変数よりも常に多くのコード領域を必要とします。これは、Neuron ファームウエアにネットワーク変数用のサポート・コードがより多く組み込まれているためです。また、タスクを実行する方法は、アプリケーション・メッセージの方が複雑です。プログラマは、メッセージの生成、送信、および受信を明示的に指定しなければなりません。サービス・タイプ、認証機能、優先度といったメッセージ属性はコンパイル時またはランタイム時に定義され、デバイスがインストールされた後ではネットワーク管理ツールを使っても構成を変更できません（ただし、これらの属性をメッセージごとに設定することはできます）。

アプリケーション・メッセージでは、ネットワーク変数に収まらないデータの転送が許可されます。ネットワーク変数がサポートしているデータ長は 31 バイトまでですが、アプリケーション・メッセージでは、1 つのメッセージ内で最高 228 バイトのデータを転送できます。ただし、大きなメッセージはほとんどの LONWORKS ルータを通過できないことに注意してください。これは、通常はルータが長いデータを処理する設定になっていないためです。

Neuron ソフトウエアの階層

プログラム中でネットワーク変数を使用すると、表には現れませんが実際にはメッセージが生成・送信されています。これは *implicit*（暗黙的）メッセージ通信と呼ばれます。図 6.1 に示すように、通信ソフトウエアはアプリケーション層（スケジューラを含む）、ネットワーク層、メディア・アクセス制御（MAC）層の 3 階層から構成されています。ソフトウエアの各階層は、LonTalk プロトコルの階層と対応しており、Neuron チップまたはスマート・トランシーバ上にある各プロセッサが処理します。

これらの階層のうちで、アプリケーション層だけがプログラム可能であることに注意してください。ネットワーク層が提供する情報の中には、スケジューラを使ってプログラムからアクセスすることができるものがあります。これらのサービスについては、この章で後述します。

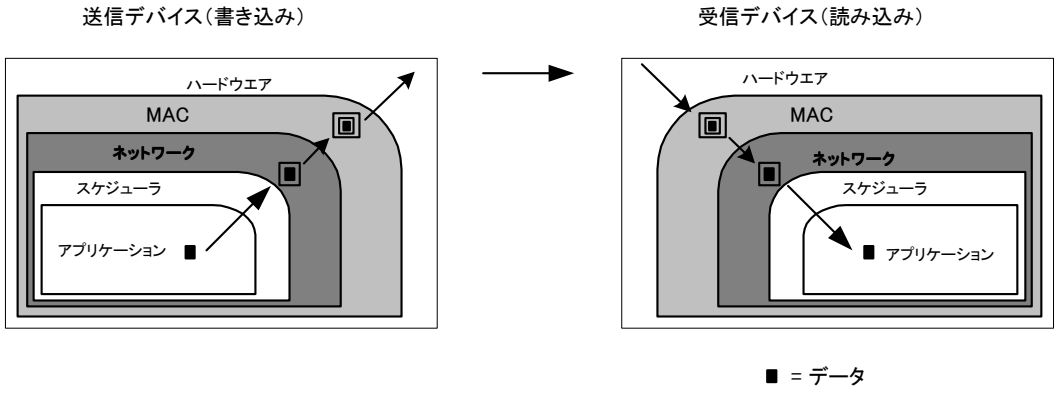


図 6.1　メッセージの送信

暗黙的メッセージとネットワーク変数

図 6.1 では、デバイスが出力ネットワーク変数に値を代入したときに何が起こるかを説明しています。まず、アプリケーション・プログラムがネットワーク変数に値を代入します。すると、スケジューラはネットワーク変数メッセージを生成して、ネットワーク層にメッセージを渡します。ネットワーク層では、ネットワーク変数メッセージにアドレス指定情報を加え、そのメッセージを MAC 層に渡します。MAC 層では、そのネットワーク変数メッセージにさらに情報を追加し、通信チャネルに対してそのメッセージを送信します。

デバイスがネットワーク変数メッセージを受信すると、次のようにメッセージを処理していきます。まず、MAC 層がメッセージを調べます。次にネットワーク層がメッセージに含まれているアドレス指定情報をチェックして、自分のデバイス宛のものかどうか確認します。そうであれば、ネットワーク変

数の情報をスケジューラに渡します。スケジューラはアプリケーション・プログラム内の適切なタスクが新しい値を使用するための準備をします。

これらのメッセージは自動的に送受信されるため、「暗黙的メッセージ」と呼ばれます。逆にアプリケーションが送信するメッセージは「明示的メッセージ」と呼ばれます。

アプリケーション・メッセージ

Neuron C では、メッセージを明示的に作成することができます。ネットワーク変数が提供している暗黙的メッセージ機能を使用する代わりに、メッセージを手動で作成し、送信することが可能です。この種のメッセージは「明示的」メッセージと呼ばれます。この明示的メッセージはメッセージ・コードを使用して指定する必要があります。「メッセージ・コード」はメッセージをアプリケーション・メッセージ、外部フレーム・メッセージ、またはネットワーク変数メッセージとして指定します。次の各セクションでは、アプリケーション・メッセージで使用するオブジェクト、関数、およびイベントの使用方法について説明します。アプリケーション・メッセージの特殊な使用方法であるリクエスト/レスポンス・メカニズムについては、アプリケーション・メッセージの一般的な説明の後で説明します。アプリケーション・メッセージに使用するメカニズムは、外部フレーム・メッセージ（独自のゲートウェイ用）および明示的にアドレス指定されたネットワーク変数メッセージを作成・送信する目的にも使用できます。

アプリケーション・メッセージは定義済みメッセージ・オブジェクトを使用して作成し、関数呼び出しと定義済みイベントを使用して処理する必要があります。この後のセクションで説明する項目を以下に示します。各セクションの中は、オブジェクト、関数、イベントという項目に分かれています。

機能		Neuron C の機能
1	メッセージの作成	**msg_out** オブジェクト
2	メッセージの送信	**msg_send()**関数 **msg_cancel()**関数
3	メッセージの受信	**msg_arrives** イベント **msg_receive()**関数 **msg_in** オブジェクト
4	確認応答付きサービスのメッセージ送信の後	**msg_completes** イベント **msg_succeeds** イベント **msg_fails** イベント
5	リクエスト/レスポンス・サービスのメッセージに対するレスポンスの送信	**resp_out** オブジェクト **resp_send()** 関数 **resp_cancel()** 関数 **resp_arrives** イベント **resp_receive()** 関数 **resp_in** オブジェクト
6	明示的なバッファの割り当て	**msg_alloc()**関数 **msg_alloc_priority()**関数 **msg_free()**関数 **resp_alloc()**関数

メッセージの作成

アプリケーション・メッセージは **msg_out** 発信メッセージ・オブジェクトを使用して作成します。この定義は Neuron C に組み込まれています。メッセージの送信には **msg_send()**関数を使用します。一度に作成できるのは、1 つの発信メッセージ（またはレスポンス）と 1 つの着信メッセージ（またはレスポンス）だけです。つまり、1 つのプログラムの中で並行して 2 つのメッセージを作成し、その両方を送ったり、2 つの入力メッセージを同時に処理することはできません。

msg_out オブジェクトの定義

発信メッセージは Neuron C コンパイラで以下のように定義されています。

```
typedef enum {FALSE, TRUE} boolean;
typedef enum {ACKD, UNACKD_RPT,
        UNACKD, REQUEST}  service_type;

struct {
     boolean priority_on;      // TRUE if a priority message
                               // (default:FALSE)
     msg_tag tag;              // message tag (required)
     int       code;           // message code (required)
     int       data[MAXDATA]   // message data
(default:none)
     boolean authenticated;    // TRUE if to be authenticated
                               // (default:FALSE)
     service_type service;     // service type
(default:ACKD)
     msg_out_addr dest_addr;   // see include file msg_addr.h
                               // (optional field)
} msg_out;
```

priority_on　メッセージを優先メッセージとして送信するときに TRUE に設定します。このフィールドに FALSE を指定するか、何も指定しないと、メッセージには優先権が与えられません。このフィールドを指定するときには、必ずメッセージ・オブジェクトの最初のフィールドとして設定し、タグよりも前にしてください。デフォルトは、FALSE（非優先）です。

tag　メッセージに対するメッセージ・タグ識別子を指定します。このフィールドは省略できません。本章の「メッセージ・タグ」を参照してください。

code　メッセージ・コードを表す数値を指定します。このフィールドは省略できません。本章の「メッセージ・コード」を参照してください。

data

アプリケーションのデータを指定します。このフィールドはオプションで、メッセージ・タグとメッセージ・コードだけのメッセージも作成できます。ネットワーク・バッファにはオーバーヘッドがあるため、MAXDATA は 228 以下に設定してください。データの最大長（MAXDATA）は、app_buf_out_size プラグマの設定によって決まります（第 8 章 参照）。

MAXDATA = app_buf_out_size – 6

または、

MAXDATA = app_buf_out_size - 17

（このプログラムのメッセージまたはネットワーク変数に明示的にアドレスを指定する場合）

注意： Neuron ファームウエアは、データ配列のどの場所に代入があったかを把握しており、それに応じて自動的に発信メッセージの長さを設定することに注意してください。

authenticated

認証機能を利用するメッセージを送信するときに TRUE を指定します。認証を必要としないメッセージを送信する場合は、このフィールドに FALSE を指定するか、何も指定しません。デフォルトは FALSE（認証なし）です。

service

次のサービス・タイプのうちの 1 つを指定します。

ACKD - （デフォルト）リトライを伴う確認応答付きサービス

REQUEST - リクエスト/レスポンス・プロトコル

UNACKD - 確認応答なしサービス

UNACKD_RPT - 反復サービス（メッセージは繰り返し送られます）

注意： 認証機能を使用するメッセージには、**UNACKD** や **UNACKD_RPT** サービスを使用できません。**ACKD** または **REQUEST** サービス・タイプを使用してください。

dest_addr　送信先アドレスを明示的に指定する **msg_out** 内のオプションのフィールドです。**dest_addr** をセットしていない場合、メッセージは「タグ」に結び付けられているアドレスに送信されます（タグが結びつけられている場合）。詳しくは、本章で後述する「明示的なアドレス指定」のセクションを参照してください。

注意：　このフィールドを使用するには、**<addrdefs.h>**と**<msg_addr.h>**をインクルードする必要があります。

メッセージ・タグ

「メッセージ・タグ」はアプリケーション・メッセージの接続点です。着信アプリケーション・メッセージは、いつでも **msg_in** という共通のメッセージ・タグで受信するようになっていますが、発信する明示的メッセージを使用する場合には、1 つ以上のメッセージ・タグを宣言する必要があります。着信タグおよび各発信タグに対する一意のネットワークアドレスの割り当てには、ネットワーク管理ツールを使用できます。

メッセージ・タグ宣言には、コネクション情報を指定するオプション・フィールドがあります。メッセージ･タグの宣言シンタックスは次のとおりです。

msg_tag [*connection-info*] *tag-identifier* [**,** *tag-identifier ...*]**;**

connection-info フィールドは、コネクション情報に接続オプションを設定するための省略可能なフィールドで、次のような形式で指定します。

bind_info (*options*)

メッセージ・タグに設定できる接続オプションは、以下のとおりです。

nonbind　このオプションを指定すると、暗黙的アドレス指定情報を含まないメッセージ・タグになります。したがって、このメッセージ・タグはアドレス・テーブル・エントリを消費しません。このオプションは、明示的にアドレス指定したメッセージを作成するときの送信先タグとして使用します。

rate_est (*const-expr***)**　推定メッセージ・レートを指定します。これはメッセージ・タグを転送するのに予想されるメッセージ・レートで、1 秒間のメッセージ数の 10 倍の値を指定します。ここには 0 から 18780 までの値（0～1878.0 メッセージ/秒）を指定できます。

max_rate_est (*const-expr***)**　最大推定メッセージ・レートを指定します。これはメッセージ・タグを転送するのに予想される最大メッセージ・レートで、1 秒間の最大メッセージ数の 10 倍の値を指定します。ここには 0 から 18780 までの値（0～1878.0 メッセージ/秒）を指定できます。

tag-identifier　メッセージ・タグ名を Neuron C の識別子の形式で指定します。

rate_est と **max_rate_est** は、いつでも指定できるとは限りません。例えば、メッセージの出力レートはデバイスがインストールされている特定のネットワークに左右されることがよくあります。これらの値はネットワーク・デバイスを解析するためにネットワーク管理ツールが使用するもので、オプションの項目です。また、**rate_est** と **max_rate_est** には 0 から 18780 までの値を指定できますが、すべての値を使用できるわけではありません。これらの値はコード化されて値 n にマッピングされます。n は 0 から 127 までの範囲の値です。コード化された値だけがデバイスにある自己識別（SI）データに保存されます。実際の値は、このコード化された値から再構成されます。コード化された値がゼロであれば、実際の値は未定義です。コード化された値が 1 から 127 の間にあれば、実際の値は a=2(n/8)-5 の結果を小数点以下 1 桁で四捨五入した数字になります。この式で求めた実際の値は、秒当たりのメッセージ数になります。

各発信メッセージの **msg_out.tag** フィールドにはメッセージ・タグを割り当てる必要があります。これは各発信メッセージがどの接続点（アドレス・テーブルのエントリに対応する）を使用するかを指定するためのものです。一度タグ・フィールドを割り当てたら、そのメッセージは送信するか、取り消すかのどちらかを実行しなければなりません。

アドレス指定以外のメッセージ・タグの役割としては、完了イベントと発信メッセージによるレスポンスとの関連付けが挙げられます。例えば、**tag1** メッセージ・タグを使って送信されたメッセージとメッセージ完了イベントを関連付けるには、次の **when** 節を使用します。

```
when (msg_completes(tag1))
```

メッセージ・タグをイベントに関連付けておくと、特定の発信メッセージに対応するイベントが起こったときにだけ、イベントの評価が TRUE になります。

メッセージ・コード

「メッセージ・コード」は、メッセージの数値識別子です。アプリケーション・メッセージには、必ずメッセージ・コードが指定されていなければなりません。このコードは、受信側のアプリケーションが各メッセージの内容を解釈するために使用します。

メッセージ・コードはアプリケーション・メッセージだけでなく、すべての LonTalk メッセージによっても使用されます。メッセージ・コードの値は表 6.1 に示す範囲になります。アプリケーションには、0～62 と 64～78 のコードを使用してください。一般に、低い方の数値は独自のアプリケーション・メッセージに使用し、高い方の数値は他のネットワークに接続するアプリケーション・レベルの独自ゲートウェイに使用します。

表 6.1　メッセージ・コードの範囲

メッセージのタイプ	メッセージ・コード	説明
ユーザ・アプリケーション・メッセージ	**0〜47**	一般アプリケーション・メッセージ。メッセージ・コードの解釈は、アプリケーション依存です。
標準アプリケーション・メッセージ	**48〜62**	LONMARK Interoperability Association が定義する標準アプリケーション・メッセージ。
応答側オフライン	**63**	アプリケーションのレスポンス・メッセージに使用します。レスポンスの送信側がオフライン状態にあり、リクエストを処理できないことを意味します。
外部フレーム	**64〜78**	他のネットワークへのアプリケーション・レベルのゲートウェイが使用します。メッセージ・コードの解釈は、アプリケーション依存です。
外部応答側オフライン	**79**	外部フレーム・レスポンスが使用します。レスポンスの送信側がオフライン状態にあり、リクエストを処理できないことを意味します。
ネットワーク診断メッセージ	**80〜95**	ネットワークを診断するためにネットワーク管理ツールが使用します。
ネットワーク管理メッセージ	**96〜127**	ネットワークをインストールし、保守するためにネットワーク管理ツールが使用します。
ネットワーク変数	**128〜255**	メッセージ・コードの下位 6 ビットには、ネットワーク変数セレクタの上位 6 ビットが含まれています。最初のテータ・バイトには、セレクタの下位 8 ビットが含まれています。

アプリケーション・メッセージの作成例

```
msg_tag motor;

#define MOTOR_ON 0
#define ON_FULL 100

msg_out.tag = motor;
msg_out.code = MOTOR_ON;
msg_out.data[0] = ON_FULL;
```

データのブロック転送

memcpy()関数を使用すると、メッセージ・データのブロックを **msg_out** オブジェクトまたは **resp_out** オブジェクトに転送できます（本章で後述する「リクエスト/レスポンス・メカニズムの使用」を参照してください）。これは、**msg_out** または **resp_out** オブジェクトのアドレスを使用する唯一のケースです。

データのブロックを **msg_out** オブジェクトにコピーするには、次のシンタックスを使用します。

void memcpy (msg_out.data, &*s*, sizeof(*s*));

resp_out オブジェクトのシンタックスも同様です。

msg_out.data コピー先を指定します。ここには、メッセージ・オブジェクトの特定のフィールドを指定することもできます（例：**&msg_out.data[3]**）。

&*s* コピーするデータを含む構造体へのポインタを指定します。このフィールドには、ポインタ、配列、または配列要素へのポインタ（例：**&a[5]**）などを指定できます。

sizeof(*s*) コピー元の構造体の大きさを指定します。

memcpy()関数は、**msg_in** または **resp_in** オブジェクトからデータ・ブロックをコピーするときにも使用します。これは、**msg_in** または **resp_in** オブジェクトのアドレスを使用する唯一のケースです。

void memcpy (&*s*, msg_in.data, sizeof (*s*));

&*s* コピー先の構造体へのポインタを指定します。このフィールドには、ポインタ、配列、または配列要素へのポインタ（例：**&a[5]**）が指定できます。

msg_in.data コピー元を指定します。ここにもメッセージ・オブジェクトの特定の領域を指定できます（例：**&msg_in.data[3]**）。

sizeof(*s*) コピー先の構造体の大きさを指定します。

長さが不明もしくは可変のメッセージでは、**msg_in.len** もしくは **resp_in.len** を用いて、**sizeof(s)**で限定することで、*s* の終わりを過ぎて書き込むことを防ぎます。

データのブロック転送例

```
msg_tag motor;
#define MOTOR_ON 0

typedef enum {
   MOTOR_FWD,
   MOTOR_REV
} motor_dir;
```

```
struct {
   long     motor_speed;
   motor_dir motor_direction;
   int      motor_ramp_up_rate;
}  motor_on_message;

when(some_event) {
      msg_out.tag = motor;
      msg_out.code = MOTOR_ON;
      motor_on_message.motor_direction = MOTOR_FWD;
      motor_on_message.motor_speed = 500;
      motor_on_message.motor_ramp_up_rate = 100;
      memcpy(msg_out.data, &motor_on_message,
            sizeof (motor_on_message));
      msg_send();
}
```

メッセージの送信

メッセージの送信と取り消しには次の関数を使用できます。

msg_send()

msg_cancel()

msg_send()関数のシンタックスは次のとおりです。

void msg_send(void);

この関数は **msg_out** オブジェクトを使用してメッセージを送信します。**msg_out** オブジェクトは、**msg_send()**関数を呼び出す前に作成する必要があります。この関数にはパラメータはなく、戻り値もありません。

以下にメッセージ送信のコード例を示します。

```
msg_tag motor;
#define MOTOR_ON 0
#define ON_FULL 100     // (100 percent)

when (io_changes(switch1) to ON)
{
      // Send a message to the motor
      msg_out.tag = motor;
      msg_out.code = MOTOR_ON;
      msg_out.data[0] = ON_FULL;
      msg_send();
}
```

msg_cancel()関数（発信メッセージの取り消し）のシンタックスは以下のとおりです。

void msg_cancel(void);

この関数は、作成した **msg_out** オブジェクトのメッセージを取り消します。割り当てられていたバッファは解放され、そのバッファが他のメッセージの作成に利用できるようになります。この関数にはパラメータはなく、戻り値もありません。

タスクの終了までに作成したメッセージを送信しないと、そのメッセージは自動的に取り消されます。

メッセージの受信

通常メッセージを受信するには、**msg_arrives** 定義済みイベントを使用します。**msg_receive()**関数を使用してメッセージを受信することもできます。

msg_arrives イベント

msg_arrives は、メッセージ受信用の定義済みイベントです。**msg_arrives** イベントのシンタックスは以下のとおりです。

msg_arrives [**(***message-code***)**]

メッセージが到着すると、このイベントの値が TRUE になります。メッセージ・コードのオプションを指定して、イベントの範囲を限定することもできます。この場合、到着したメッセージが指定されたコードを含んでいる場合にだけ、イベントが TRUE になります。

非限定 **msg_arrives** イベントと限定 **msg_arrives** イベントを混在して使用するときには、**#pragma scheduler_reset** 指令を指定し、限定イベントの **when** 節がすべて評価されてから、非限定イベントの **when** 節が評価されるようにします。

次の例に示すように、プログラムにはデフォルトのケースを指定して、イベントのキューのロックアップが発生しないよう配慮してください。これについては、本章で後述する「デフォルトの **when** 節の重要性」のセクションで詳しく説明します。

以下のリスト 6-1 は、このイベントの使用例です。

リスト 6-1　msg_arrives イベントの使用例

```
#pragma scheduler_reset
when (msg_arrives(1))
{
      io_out(sprinkler, ON);
}

when (msg_arrives(2))
{
      io_out(sprinkler, OFF);
}

when (msg_arrives)      // default case for
            // handling unexpected message codes
{
      // Do nothing, just discard it
}
```

着信メッセージ・キューがブロックされるのを防ぐため、リスト 6-1 に示すようなアプリケーション・メッセージを受信するプログラムには非限定 **msg_arrives** イベントを使用してデフォルトの **when** 節を含めるようにしてください。これについては、本章で後述する「デフォルトの **when** 節の重要性」でさらに詳しく説明します。

msg_receive( )関数

msg_receive()関数のシンタックスは以下のとおりです。

boolean msg_receive(void);

この関数は **msg_in** オブジェクトを使用してメッセージを受信します。新しいメッセージを受信したときは TRUE を返し、それ以外のときは FALSE を返します。

メッセージを受信していなければ、この関数はメッセージの受信を待たずに戻ります。バイパス・モード（直接イベント処理とも呼ばれる）のときなど、単一のタスクの中で複数のメッセージを受信するときにもこの関数を使用する必要が生じることがあります。すでに「受信された」メッセージがあれば、前に到着した方のメッセージは廃棄されます（そのメッセージが使用しているバッファが解放されます）。

msg_receive()や **resp_receive()**関数の呼び出しには、**post_events()**の呼び出しが伴います。したがって、**msg_receive()**や **resp_receive()**を呼び出すと、そこがクリティカル・セクションの境界になります（後述の「レスポンスの受信」のセクションを参照してください）。

msg_receive()関数を使用すると、すべてのメッセージは生（raw）形式で受信されます。このため、**online**、**offline**、**wink** などの特殊イベントは使用できますが、メッセージ・コードをチェックして、これらのイベントを明示的に調べる必要があります。このため、アプリケーション・プログラムが **wink**、**online**、**offline** などの特殊イベントを使用する場合、**msg_receive()**関数は使用できません。

着信メッセージのフォーマット

着信メッセージのオブジェクト名は **msg_in** です。この定義は Neuron C に組み込まれています。オブジェクトの該当するフィールドを調べ、メッセージを読み込みます。

msg_in オブジェクトのフィールドは読み取り専用で、値を代入することはできません。着信メッセージの事前定義を以下に示します。

```
typedef enum {FALSE, TRUE} boolean;
typedef enum {ACKD, UNACKD_RPT,
        UNACKD, REQUEST}   service_type;

struct  {
     int code;           // message code
     int len;            // length of message data
     int data[MAXDATA];  // message data
     boolean authenticated;    // TRUE if message was
                         // authenticated
     service_type service;     // service type used by sender
     msg_in_addr addr;   // see <msg_addr.h> include file
     boolean duplicate;  // the message is a duplicate
     unsigned rcvtx;     // the message's receive tx ID
} msg_in;
```

警告： **msg_out** オブジェクトに値を代入すると、**msg_in** オブジェクトのフィールドが無効になることがあります。メッセージの受信後、メッセージの

送信を開始する前に、**msg_in** オブジェクトの必要なフィールドを検査するか、値を保存しておく必要があります。

code メッセージ・コードを表す数値です。詳しくは、前述の「メッセージ・コード」のセクションを参照してください。

len メッセージ・データの長さです。

data アプリケーションのデータを指定します。このフィールドは、**len** の値が **0** よりも大きいときにだけ有効です。データの最大長（MAXDATA）は**#pragma app_buf_in_size 指令**によって決まります（第 8 章を参照してください）。

MAXDATA = app_buf_in_size – 6

または、

MAXDATA = app_buf_in_size - 17

（メッセージまたはネットワーク変数に明示的にアドレスを指定する場合）

authenticated メッセージが認証された場合、このフィールドの値は TRUE になります。認証されなかった場合には、FALSE になります。

service 次のサービス・タイプのうちの 1 つを指定します。

ACKD - （デフォルト）リトライを伴う確認応答付きサービス

UNACKD - 確認応答なしサービス

UNACKD_RPT - 反復サービス（メッセージは繰り返し送られます）

REQUEST - リクエスト・サービス。このサービスを使ってメッセージが送信されていると、受信側のデバイスは送信側にレスポンスを返し、送信側はそのレスポンスを処理します。リクエスト/レスポンス・メカニズムについて詳しくは、本章で後述します。

addr 着信メッセージにあるオプションのフィールドです。アプリケーション・プログラムがメッセージの送信元と送信先を判断するために使用します。**msg_in_addr** 型は、<**msg_addr.h**>インクルード・ファイルの中で定義されています。

このフィールドを使用するには、<**msg_addr.h**>ファイルをインクルードする必要があります。

duplicate このブール型フラグが TRUE のとき、アプリケーションに渡されるメッセージは重複リクエスト・メッセージになります。アプリケーション・レスポンスの中に 1 バイトのメッセージ・コードの他にもデータが含まれているとき、このメッセージが渡されます。

rcvtx 受信トランザクション ID です。この ID は、デバイスのトランザクション・データベースの中でメッセージによって使用されます。

デフォルトの when 節の重要性

前に掲載したリスト 6-1 では、メッセージを使用した場合の重要なテクニックを説明しています。アプリケーション・メッセージを受信するプログラムは、不要なメッセージを受信してそれを破棄する機能を持っていなければなりません。メッセージを廃棄するには、リスト 6-1 に示した形式を使用するか、**switch** 文の default 条件文で対処します。

不必要なメッセージを廃棄する処理を忘れると、アプリケーションが到着したメッセージを処理しないまま、そのメッセージがキューの先頭に残ってしまいます。そのため、他のメッセージやネットワーク変数の到着イベントが起こらず、デバイスがリセットされるまでずっとそのデバイスはロックされた状態になります。例えば、サービス・ピン・メッセージは、すべてのデバイスに送信されますが、ほとんどのデバイスには関係ないメッセージです。サービス・ピン・メッセージが必要なデバイスはネットワーク管理ツールなどに限られます。その他のデバイスでは、このメッセージを廃棄しなければなりません。

プログラムがメッセージを処理しない場合（**when(msg_arrives)**を使った暗黙的プログラムでも、**msg_receive()**を使った明示的プログラムでも）、スケジューラは自動的にすべての着信メッセージを廃棄します。

ネットワーク変数だけを使っているデバイスでは、この現象を気にする必要はないことに注意してください。ネットワーク変数だけを使っていれば、スケジューラがすべての着信メッセージを処理するためです。

コード例

次のコード例では、ネットワーク変数の代わりにアプリケーション・メッセージを使用するように照明ランプとスイッチのプログラムを書き換えています。

ランプ・プログラム

最初に照明デバイスのランプ・プログラムを示します。

```
// lamp.nc - Generic program for a lamp
// The lamp's state is governed by an incoming
// application message

#define LAMP_ON 1
#define LAMP_OFF 2
#define OFF 0
#define ON 1


// I/O declaration
IO_0 output bit io_lamp_control;


when (msg_arrives) {
    switch (msg_in.code) {
    case LAMP_ON:
        io_out(io_lamp_control, ON);
      break;
    case LAMP_OFF:
        io_out(io_lamp_control, OFF);
      break;
    } //end switch
} //end when
```

スイッチ・プログラム

スイッチ・デバイスに使用するスイッチ・プログラムを示します。

```
// switch.nc - Generic program for a switch
// Send a message when the switch changes state

#define LAMP_ON  1
#define LAMP_OFF 2
#define OFF 0
#define ON 1

// I/O Declaration
IO_4 input bit io_switch_in;

// Message tag declaration
msg_tag TAG_OUT;


// Event-driven code
when (reset) {
   io_change_init(io_switch_in);
}

when (io_changes(io_switch_in)) {
   // Set up message code based on the switch state
   msg_out.code = (input_value == ON) ? LAMP_ON : LAMP_OFF;

   // Set up message tag and send message
   msg_out.tag = TAG_OUT;
   msg_send();
}
```

メッセージ・タグの接続

各デバイスには、デフォルトの **msg_in** 入力メッセージ・タグがあります。ネットワーク・インテグレータはネットワーク管理ツールを使用して、発信メッセージ用のメッセージ・タグを **msg_in** 入力メッセージ・タグに接続します。例えば、上記の 2 つのデバイスのメッセージ・タグの接続は、次のようになります。

TAG_OUT connects to msg_in

（スイッチ デバイス上）　　（照明 デバイス上）

明示的アドレス指定

アプリケーション・メッセージおよびネットワーク変数に対して明示的に送信先アドレスを指定するには、**<msg_addr.h>**と**<addrdefs.h>**のインクルード・ファイル内のデータ構造体を使用します。発信メッセージに明示的アドレス指定を使用するには、**msg_out** オブジェクトに含まれる **dest_addr** 共用体のいずれかの要素の該当するフィールドすべてに適切な値を代入してから、**msg_send()**を呼び出します。メッセージ・タグからアドレス指定情報を決めるわけではありませんが、メッセージにはメッセージ・タグが依然として必要です。このため、どのようにメッセージ・タグがバインドされていても、明示的に指定されているアドレスはタグにより指定されているアドレスより優先されます。

明示的な送信先アドレスを割り当てたときには、応答イベントや完了イベントの処理との関連付けのためだけにメッセージ・タグが使用されます。ただし、明示的にアドレス指定したメッセージにメッセージ・タグを使用するだけであっても、標準メッセージ・タグを使用している限り、アドレス・テーブル・エントリが消費されます。Neuron リソースをより効率的に利用するには、アドレス指定情報を持たず、アドレス・テーブル・エントリを消費しない *non-bindable*（バインドしない）メッセージ・タグを使用します。このようなメッセージ・タグは、次のシンタックスを使用して宣言します。

msg_tag bind_info(nonbind [, *other-info***]) *tag-name*;**

バインドしないオプションの詳細については、本章の「メッセージ・タグ」を参照してください。

明示的にアドレスを指定すると、Neuron ファームウエアが必要とするバッファのサイズに影響します。詳細については、第 8 章の「表 8.1 バッファ数とバッファ・サイズ」を参照してください。

ネットワーク変数の更新に対応する明示的メッセージを作成し、送信先アドレスを明示的に設定すると、明示的アドレス指定を使ってネットワーク変数の更新を送信できます。ネットワーク変数の更新を送るための明示的なメッセージのフォーマットについては、『FT 3120 and FT 3150 Smart Transceivers Databook』を参照してください。また、アドレス指定の詳細については同じドキュメントの「A.3、The Address Table」のセクションを参照してください。

確認応答付きサービスを使ったメッセージ送信

あるデバイスが確認応答付きサービス（デフォルト）でメッセージを送信すると、すべての受信デバイスがメッセージの受信を送信デバイスに通知しなければなりません。図 6.2 に示すように、この確認応答はネットワーク・プロセッサが送出します。確認応答メッセージはデータのないメッセージで、メッセージを発信したデバイス上のネットワーク・プロセッサに対して送られます。

メッセージの確認応答については、アプリケーション層は何もしないことに注意してください。では、プログラムからメッセージ送信の完了および失敗を知るにはどうすればいいのでしょうか。以下のセクションで、この質問に答えます。

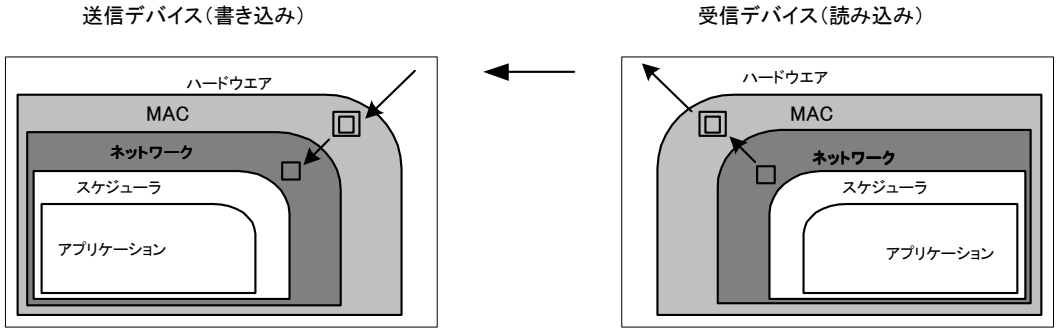


図 6.2　メッセージの確認応答

メッセージの完了イベント

送信デバイスがメッセージの完了を知るためのイベントには、次の 3 つがあります。

msg_completes [**(***msg-tag-name***)**]

msg_succeeds [**(***msg-tag-name***)**]

msg_fails [**(***msg-tag-name***)**]

これら 3 つのイベントはすべて、*msg-tag-name* にメッセージ・タグ名を指定し、イベントの適用範囲を限定することができます。*msg-tag-name* オプションが指定されていないときには、イベントはすべてのメッセージに適用されます。

非限定メッセージの完了イベントを用いる場合、どのメッセージ・タグがイベントに対応しているかを判断するために、組み込み変数 **msg_tag_index** が使われることがあります。詳細情報については『Neuron C Reference Guide』の「Predefined Events」の章を参照してください。

msg_completes イベントは最も一般的に使用されるイベントです。発信メッセージが完了すると、成功・失敗に関わらず、このイベントの値が TRUE になります。

msg_succeeds イベントは、メッセージの送信が成功したときに TRUE になります。また、**msg_fails** イベントは、メッセージの送信が失敗したとき（リトライもすべて失敗したとき）に TRUE になります（表 6.2 に各サービス・タ

イプごとに、何が「成功」で何が「失敗」であるかを詳しく説明しています）。1 つのメッセージ送信につき、これら 3 つのイベントうちの 1 つだけが TRUE になります。したがって、イベント処理の順番が重要になります。例えば、**msg_ completes** イベントが **msg_succeeds** や **msg_fails** イベントより前に処理されるようになっていると、**msg_succeeds** と **msg_fails** イベントは TRUE になることはありません。

注意： 本章で後述する「**resp_arrives** と **msg_succeeds**」のセクションを参照してください。

これらのイベントは、確認応答付きサービスやリクエスト/レスポンス・サービス（後述の「リクエスト/レスポンス・メカニズム」のセクションを参照してください）でメッセージを送信するときに意味があります。確認応答なしサービスや反復サービスでメッセージを送信する場合は、メッセージがネットワーク・プロセッサから送信デバイス上のメディア・アクセス制御（MAC）プロセッサに転送された時点で **msg_succeeds** と **msg_completes** イベントが必ず TRUE になります。

表 6.2　成功/失敗の完了イベント

サービス・タイプ	成功 =	失敗=
確認なし	メッセージが MAC プロセッサに転送された。	*
反復	*N* 個のメッセージが MAC プロセッサに転送された。（*N* は繰り返しの回数）。	*
確認	すべての確認応答を送信デバイスのネットワーク・プロセッサが受信した。	1 つ以上の確認応答が受信されていない。これはメッセージとネットワーク変数の両方に適用されます。
リクエスト/レスポンス	すべてのレスポンスを送信デバイスのアプリケーション・プロセッサが受信した。	メッセージ：1 つ以上のレスポンスが到着していない。 ネットワーク変数のポーリング： (a) 1 つ以上のレスポンスが到着していない。 (b) どのレスポンスにも有効なデータが入っていない。
* いずれの場合でも、Neuron ファームウエアでアドレス・エラーが起こると、失敗イベントが発生します（『Neuron C Reference Guide』を参照してください）。また、バインドされていないネットワーク変数またはメッセージへの送信は、成功イベントになります。		

メッセージの完了イベント処理

メッセージを送信するとき、完了イベントをチェックするかどうかを選択できます。ただし、完了イベントのチェックを行うときには、次のような注意事項があります。

まず、**msg_succeeds** または **msg_fails** のどちらかをチェックする場合には、両方のイベントをチェックしてください。あるいは、単に **msg_completes** だけを使ってください。

次に、完了イベントに特定のメッセージ・タグを指定した場合、常にそのメッセージ・タグに対する完了イベントが処理されます。したがって、プログラムは特定のメッセージ・タグに対する完了イベントを処理し、その他のメッセージ・タグに対する完了イベントを無視することができます。次のプログラム例では、TAG1 に対する完了イベントを処理し、TAG2 に対する完了イベントは処理していません。

```
when (io_changes(dev1))
{
   .
   .
   .
   msg_out.tag = TAG1;
   .
   .
   .
   msg_send();
}


when (msg_completes(TAG1))
{
   .
   .
   .
}


when (io_changes(dev2))
{
   .
   .
   .
   msg_out.tag = TAG2;
   .
   .
   .
   msg_send();
}
```

3 つめの制限は、*unqualified*（非限定）完了イベントに関するものです。この完了イベントは、すべてのメッセージを暗黙的にチェックします。非限定完了イベントを使用するときには、すべての確認メッセージを処理する必要があります。この処理は、メッセージ・タグごとに明示的に行うか、メッセージが送信されるたびに非限定イベントを使用することで暗黙的に行われます。

以下のコード例で、メッセージ・タグによる完了イベントの正しい処理を示します。

```
int failures[2], success;
msg_tag TAG1, TAG2;


when (io_changes(toggle))
{
   msg_out.tag = TAG1;
   msg_out.code = TOGGLE_STATE;
   msg_out.data[0] = input_value;
   msg_send();
```

```
   msg_out.tag = TAG2;
   msg_out.code = TOGGLE_STATE;
   msg_out.data[0] = input_value;
   msg_send();
}


when (msg_fails(TAG1))
{
   failures[0]++;
}


when (msg_fails(TAG2))
{
   failures[1]++;
}


when (msg_succeeds)     // any message qualifies
{
   success++;
}
```

先取りモードとメッセージ

発信メッセージ用に使用できるアプリケーション・バッファがないと、Neuron ファームウエアは「先取りモード（プリエンプションモード）」になります。先取りモードでは、アプリケーション・バッファが必要になると、システムはアプリケーション・プログラムを止め、完了イベント、応答イベント、およびネットワーク変数とメッセージの受信イベントだけを処理し、バッファの空きを作ろうとします。

これ以外の定義済みイベントやユーザ定義イベントは、イベント式を含む **when** 節と共に **preempt_safe** キーワードを使用していない限り処理されません。**when** 節のシンタックスについては、第 2 章「シングル・デバイスでの機能」を参照してください。

ウォッチドッグ・タイマーは、待ち時間中も更新されています。プログラムの待ち時間が設定時間を超えてしまうと、デバイスがリセットされます。この設定可能なタイマーは、Max Free Buffer Wait（上限なしバッファ待ち）タイマーと呼ばれます。バッファ待ち時間タイムアウト（先取りモード・タイムアウトとも呼ばれる）は、デバイスが完全に送信できなくなった場合にのみ発生するようにします。このような状態は、極端なネットワーク負荷やネットワーク障害があったときにも起こることがあります。

またバッファ待ちタイムアウトは、プログラムが完了イベントを正しく解放していないときにも発生します。このような状態を回避するには、**if (nv_update_ completes)**といった完了イベントをバイパス・モードでチェックすることはせず、**when** 節の中で対応する完了イベントをチェックするようにしてください。

ネットワーク変数を使用しているときには、次のどちらかの場合にだけシステムが先取りモードになります。

- Sync 型ネットワーク変数が更新されているとき
- **flush_wait()**が呼ばれたとき

システムが先取りモードになっているときに、メッセージの完了イベントを使った **when** 節のタスクの中でメッセージを送信しようとすると、その新しいメッセージ用のバッファが取れないためにデバイスがリセットされてしまいます。

したがって、以下のようなプログラムは作成しないでください。

```
when (TOGGLE_ON)
{
   // build a message
   // send the message
}

when (msg_completes)
{
   msg_out.tag = t;      // This sequence is not
                         // recommended.
   msg_out.code = 1;     // Causes a device reset
                         // if the system is
                         // already in preemption
                         // mode
}
```

このシーケンスを使用する代わりに、**msg_completes** イベントを使用しない **when** 節を使用して、タスクの中でメッセージを作成し、**msg_send()**を呼び出すようにしてください。同期出力ネットワーク変数を更新すると、この更新を処理するのに十分なアプリケーション出力バッファがない場合に、クリティカル・セクションの境界で先取りモードになります。例えば、3 つの同期出力ネットワーク変数をクリティカル・セクションで更新するとします。ここで、使用可能なアプリケーション出力バッファが 2 つしかないと、このクリティカル・セクションの終了時に先取りモードになります。未処理のネットワーク変数の更新がすべてバッファに取り込まれると、先取りモードが終了し、通常の操作に戻ります。

バッファ割り当てが暗黙的に行われる場合（最初に **msg_alloc** を呼び出さずに明示的メッセージを作成する場合）、使用可能なアプリケーション出力バッファがないと、**msg_out** に最初に値を代入する際に先取りモードになります。完了イベントが処理されてバッファが使用可能になると、すぐに先取りモードは終了します。デバイスが先取りモードになっていると、ネットワーク変数が優先ネットワーク変数でもそうでなくても、更新処理は行われません。そのため、ある一定時間内に優先ネットワーク変数の更新が実行されることを想定しているプログラムでは、非優先の Sync 型ネットワーク変数やバッファ割り当てを明示的に行わないメッセージは使用しないでください。

明示的にバッファを割り当てたり解放したりするには、本章で後述される「アプリケーション・バッファの割り当て」のセクションで説明する関数を使用してください。

プログラムが先取りモードになっているかどうかは、次の関数を使って判別できます。

boolean preemption_mode (void);

この関数は、デバイスが先取りモードになっていると TRUE を返します。

非同期イベント処理と直接イベント処理

イベントは、**when(msg_completes)**、**when(msg_fails)**、**when(msg_succeeds)**という **when** 節とイベントを使ってチェックできます。このタイプのイベント処理は、実際の実行の順序をスケジューラが管理しているため、非同期イベント処理」と呼んでいます。もう 1 つのテクニックは **if** 文や **while** 文を使って完了イベントをタスク内でチェックするもので、「直接イベント処理」と言います。

次のコードは、一方向の非同期イベント処理の例で、直接イベント処理を併用することはできません。メッセージ完了イベント **when** 節のタスクの中では、メッセージ完了イベントのチェックをしないでください。

```
when (msg_completes)
{
   post_events();
   if (msg_completes)    // not recommended
      x = 4;
}
```

非同期イベント処理と直接イベント処理は 1 つのプログラムの中で併用することができます。非同期イベント処理は、イベント処理として典型的な方法です。この方法を使用した方が、アプリケーション・プログラムが小さくなります。アプリケーション・プログラムの中で非同期イベント処理から直接イベント処理に変わるところがあれば、その前に **flush_wait()**関数を呼んでください。**flush_wait()**関数を使用すれば、直接イベント処理に移る前に、処理の終わっていない完了イベントや応答イベントがすべて確実に処理されます。

次のプログラム例では、メッセージを送信し、完了イベントを直接的に処理しています（**when** 節ではなくタスクの内側でイベントをチェックしています）。

```
msg_tag motor;
#define MOTOR_ON 0

when (x==3)
{
   // send a message
   flush_wait();
   msg_out.tag = motor;
   msg_out.code = MOTOR_ON;
   msg_send();

   // check completion status
   while (!msg_succeeds(motor)) {
      post_events();
      if (msg_fails(motor))
         node_reset();
   }
}
```

リクエスト/レスポンス・メカニズム

リクエスト/レスポンス・メカニズムは、あるデバイスで実行されているアプリケーションが他のデバイス上のアプリケーションにデータをリクエストするためのメッセージ送受信です。このメカニズムは、入力ネットワーク変数をポーリングするときに Neuron ファームウエアが自動的に使用しています。

なお、明示的メッセージを使用するアプリケーション・プログラムでも利用できます。

「リクエスト」はリクエスト・サービスを使用するメッセージです。リクエスト・メッセージの送信は、ネットワーク変数のポーリングに似ています。ポーリングでは、特定のネットワーク変数の最新値をスケジューラから受け取るようになっていました。これに対してリクエストでは、相手のデバイス上のアプリケーションに対して「リクエスト時点でのリクエスト」を評価してレスポンスを送り返すように要求します。

レスポンスの作成、送信、受信に関する関数、イベント、オブジェクトは、前のセクションで説明したメッセージの作成、送信、受信とほぼ同じです。これらについて以下に要約します。

以下は、リクエストの送信プログラム例です。

```
msg_tag motor;
#define MOTOR_STATE 1

when (io_changes(switch1) to 0)
{
   //send a request to the motor
   msg_out.tag = motor;
   msg_out.service = REQUEST;
   msg_out.code = MOTOR_STATE;
   msg_send();
}
```

リクエストは､図 6.1 に示したようにパッケージに組み立てられていきます。受信デバイス上のアプリケーション・プログラムは **when** 節（または **msg_receive()**関数）を使ってリクエストを受け取り、このリクエストに対するレスポンスを作成します。リクエストとレスポンスの作成について、図 6.3 に示します。

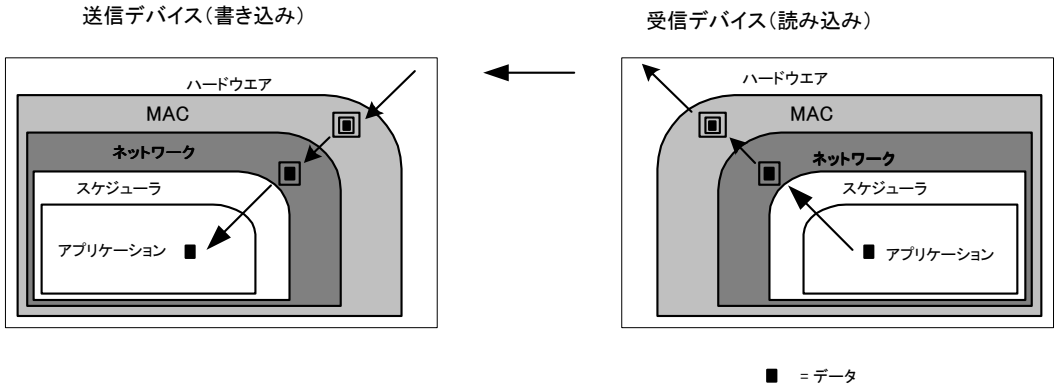


図 6.3　レスポンスの送信

レスポンスの作成

リクエスト・メッセージに対して、レスポンスを作成することができます。図 6.3 に示すように、レスポンスには送信デバイスのアプリケーション・プロセッサに送られるデータが含まれています。レスポンスは確認応答とは異なります（図 6.2 を参照）。確認応答にはデータ部分がありませんし、送信デバイス上のネットワーク・プロセッサまでにしか送られません。

レスポンスを発信するために使用するオブジェクトの名前は **resp_out** です。レスポンスに優先権があるかどうかと認証機能を使用するかどうかは、リクエスト側の設定を受け継ぎます。レスポンスはリクエストの送信元に送り返されるので、メッセージ・タグは必要ありません。それと同じ理由で、レスポンスを明示的にアドレス指定することはできません。

組み込み発信レスポンス・オブジェクトの定義を以下に示します。

```
struct {
      int code;          // message code
      int data[MAXDATA];// message data
} resp_out;
```

code　メッセージ・コードを 0～79 の範囲で表します。このフィールドは省略できません。このコード・フィールドに使用される数値の範囲の詳細については、本章の「メッセージ・コード」のセクションを参照してください。

data　メッセージ・データです。このフィールドはオプションです。データの最大長（MAXDATA）は、**#pragma app_buf_in_size** 指令（第 8 章「メモリ管理」を参照）によって決まります。

MAXDATA = app_buf_in_size – 6

または、

MAXDATA = app_buf_in_size - 17
（明示的にアドレスを指定する場合）

注意： Neuron チップファームウエアはデータ配列のどの場所に代入があったかを把握しており、それに応じて自動的に発信メッセージの長さを設定することに注意してください。

レスポンスの送信

レスポンスを送信するには、**resp_send()**関数を使用します。着信リクエストを処理したクリティカル・セクションでレスポンスを送信してください。レスポンスはリクエストが到着したアプリケーション入力バッファ内で作成されます。このため、一度レスポンスの作成を始めると、着信リクエストの内容を見ることはできなくなります。また、他のメッセージの送受信の割り込みもできません。このレスポンス送信のときにだけ、発信メッセージがアプリケーション入力バッファを使用します。

resp_send()関数のシンタックスは以下のとおりです。

void resp_send (void);

この関数は **resp_out** オブジェクトを使ってレスポンスを送信します。

注意： アプリケーションがアプリケーション入力バッファにレスポンスを作成している間、ネットワーク・プロセッサはネットワーク出力バッファを

使用してレスポンス・パケットを作成します。このため、ネットワーク出力バッファのサイズは他の発信メッセージだけでなく、発信レスポンスも格納できる大きさにする必要があります。

レスポンスの受信

通常のプログラムは、レスポンスの受信に定義済みイベント **when(resp_arrives)**を使用します。また、**resp_receive()**関数も使用できます。

resp_arrives イベント

応答を受信するには、定義済みイベント **resp_arrives** を使用します。

resp_arrives イベントのシンタックスは以下のとおりです。

resp_arrives [**(***msg-tag-name***)**]

レスポンスが到着すると、このイベントの値が TRUE になります。*msg-tag-name* にメッセージ・タグ名を指定して、イベントの適用範囲を限定することもできます。これによってイベントは、名前付きメッセージ・タグを使用した送信済みリクエストに対応するレスポンス・メッセージに制限されます。イベントを限定するメッセージ・タグ名がないときは、到着するすべてのレスポンス･メッセージに対してイベントの評価が TRUE になります。

resp_receive( )関数

resp_receive()関数のシンタックスは以下のとおりです。

boolean resp_receive(void);

この関数は **resp_in** オブジェクトを使用して応答を受信します。レスポンスを受信したときは TRUE を返し、それ以外のときは FALSE を返します。レスポンスは、受信したタスクの終了時に自動的に廃棄されます。

resp_receive()の呼び出しには **post_events()**の呼び出しが伴います。そのため、**resp_receive()**関数を呼び出したところは、クリティカル・セクションの境界になります。

レスポンスのフォーマット

着信レスポンスのオブジェクト名は **resp_in** です。

着信応答の構造体は、Neuron C コンパイラで次のように定義されています。

```
struct {
      int code;   // message code
      int len;    // length of message data
      int data[MAXDATA];      // message data
      resp_in_addr addr;      // explicit address - see the
                              // <msg_addr.h> include file
} resp_in;
```

code メッセージ・コードを 0～79 の範囲で表します。詳しくは、本章の「メッセージ・コード」のセクションを参照してください。

len メッセージ・データの長さです。

data

メッセージ・データです。このフィールドは **len** が **0** よりも大きいときにだけ有効です。データの最大長（MAXDATA）は、**#pragma app_buf_in_size** プラグマ（第 8 章を参照）によって決まります。

MAXDATA = app_buf_in_size – 6

または、

MAXDATA = app_buf_in_size - 17

（メッセージやネットワーク変数に明示的にアドレスを指定する場合）

addr

着信メッセージのオプション・フィールドで、アプリケーション・プログラムがメッセージの送信元と送信先を判断するときに使用できます。**resp_in_addr** 型は、<**msg_addr.h**>インクルード・ファイルの中で定義されています。

このフィールドを使用するときは、<**addrdefs.h**>と<**msg_addr.h**>ファイルをインクルードする必要があります。

コード例

ここでは、リクエストを送信して、レスポンスを非同期イベント処理を使って受信するプログラム例を示します（このリクエストを受信してレスポンスを送信するコードは、この次の例にあります）。

```
msg_tag  tag1;
#define DATA_REQUEST 0


when (io_changes(toggle))
{
   msg_out.tag = TAG1;
   msg_out.code = DATA_REQUEST;
   msg_out.service = REQUEST;
   msg_send();
}


when (resp_arrives(TAG1))
{
   if (resp_in.code == OK)
      process_response(resp_in.data[0]);

}
```

上のリクエストに対するレスポンスを送信するプログラムです。

```
#define DATA_REQUEST 0
#define OK 1
```

```
when (msg_arrives(DATA_REQUEST))
{
      int x, y;
      x = msg_in.data[0];
      y = get_response(x);
      resp_out.code = OK;     // msg_in no longer
                              // available
      resp_out.data[0] = y;
      resp_send();
}
```

次の例では、リクエストを送信して、レスポンスを直接イベント処理で受信しています。

```
int x;
msg_tag motor;
#define MOTOR_ON 0
#define DO_MOTOR_ON 3

when (command == DO_MOTOR_ON)
{
   // send a request
   msg_out.tag = motor;  // construct the message
   msg_out.code = MOTOR_ON;
   msg_out.service = REQUEST;
   msg_send();     // send the message


   // wait for completion
   while (!msg_succeeds(motor)) {
      post_events();
      if (msg_fails(motor))
         node_reset();
      else if (resp_arrives(motor)) {
         x = x + resp_in.data[0];
         resp_free();         // optional
      }
   }
}
```

resp_arrives と msg_succeeds の比較

resp_arrives と完了イベント（**msg_succeeds**、**msg_fails** および **msg_completes**）とでは異なる情報を提供しますから、これらのイベントの両方を同じリクエスト・トランザクションに使用できます。ここでは、この相異点について説明します。

マルチキャスト（グループ）アドレス指定を使って 6 つのデバイスに対して 1 個のリクエストを送り、3 個のレスポンスを受信し、残り 3 個を受信していないものとします。**resp_arrives** イベントはレスポンスを受信する毎に値が TRUE になりますから、この場合は 3 回 TRUE になっています。ところが、すべてのレスポンスが到着してはいないため、**msg_succeeds** の値は TRUE になりません。**msg_fails** イベントは、到着設定時間までにすべてのレスポンスが受信されないと、TRUE になります（言い換えれば、**msg_succeeds** が TRUE になるためには、すべてのレスポンスの受信が必要です）。

レスポンスは、常にメッセージ完了イベント（**msg_completes**、**msg_fails**、**msg_succeeds**）よりも前に到着します。

idempotent（べき等）リクエストと non-idempotent（非べき等）リクエスト

idempotent（べき等）トランザクションとは、繰り返し実行可能なトランザクションです。例えば、「照明を点灯する」という命令を繰り返し実行しても、その結果に変更がありません（照明は点灯したままです）。

non-idempotent（非べき等）トランザクションは、繰り返し実行するとその結果が変わるものをいいます。例えば「照度を 10%上げる」という命令は、非べき等トランザクションになります。この命令に 10 回レスポンスを送信するのと、1 回レスポンスを送信するのでは、結果が異なります。

LonTalk メッセージは「idempotent」属性をサポートしていません。LonTalk では、リクエストに対するレスポンスの中にアプリケーション・データが含まれているかどうかによって、受信デバイスがこの属性を暗黙的に指定します。

レスポンスの中にアプリケーション・データが含まれていなければ、Neuron ファームウエアは、そのリクエストが「非べき等」であり、アプリケーションに対して繰り返し指示することはできないと判断します。この場合、ファームウエアは繰り返し送信されるリクエストに対して元のリクエストを送り、アプリケーションには繰り返し送信されるリクエストを転送しません。この方法では、メッセージが重複しているかどうかをアプリケーションがテストする必要がないため、データのないレスポンスに対するアプリケーションの処理が簡単になります。

レスポンスの中にアプリケーション・データが含まれていると、Neuron ファームウエアは、そのリクエストが「べき等」であり、アプリケーションに対して繰り返し指示することができると判断します。この場合は、Neuron チップファームウエアは繰り返し送信されるリクエストをアプリケーションに送るので、アプリケーションはレスポンスを再生成する必要があります。これによって、アプリケーションは繰り返しリクエストに対するレスポンスを更新することができます。アプリケーションがこれらの繰り返しリクエスト・メッセージを「非べき等」として扱うこともできます。その場合には、受信トランザクション・インデックスを使ってレスポンスをバッファに保存し、重複したリクエストの到着に対して、これらのレスポンスを再発行するようにします。以下にコード例を示します。

コード例

```
#define OK 1
#define MAXRESP 10


struct RespBuffer {
      int code;
      unsigned int len;
      int data[MAXRESP];
} resp_buffer[16];
```

```
when (msg_arrives) {
      struct RespBuffer *buf_p;

      if (msg_in.service == REQUEST) {
            buf_p = &resp_buffer[msg_in.rcvtx];
            if (!msg_in.duplicate) {
                  int i;

                  // Process initial request
                  // . . .

                  // Now save response
                  buf_p->code = OK;
                  buf_p->len = MAXRESP;
                  for (i=0; i<MAXRESP; i++) {
                        buf_p->data[i] = get_resp_data();
                  }
            }


            // Generate the response.
            resp_out.code = buf_p->code;
            memcpy(resp_out.data, buf_p->data,
buf_p->len);
            resp_send();
      }
}
```

上の例では、**msg_in** オブジェクトの **rcvtx** フィールドで、どの受信トランザクション・インデックスにリクエストが属しているかも指定しています。

アプリケーション・バッファ

Neuron C アプリケーションが使用する着信用と発信用バッファの数は、コンパイル時に設定できます。Neuron 3120®チップと Neuron 3120E1 チップを除く Neuron チップとスマート・トランシーバの全モデルのデフォルトでは、優先アプリケーション出力バッファ 2 個、非優先アプリケーション出力バッファ 2 個、アプリケーション入力バッファ 2 個となっています。バッファ割り当てについては、第 8 章「メモリ管理」を参照してください。レスポンスの処理を最も効率よく行うには、アプリケーション入力バッファ数を予想されるレスポンスの数に合わせておきます。1 つのリクエストに対して不釣り合いに多いレスポンス（例えば 10 より多い）が返ってくるような場合は、アプリケーション入力バッファ数が限定されているためにレスポンスのいくつかが受信されなくなる可能性もあります。

注意： 作成したプログラムが Neuron 3120 チップまたは Neuron 3120E1 チップ用とリンクしているときには、これらのチップのメモリが限られているため、リンカが出力バッファのデフォルト数を調整して、優先出力バッファ 1 個、非優先出力バッファ 1 個にします。入力バッファ数は 2 個のままです。

注意： また、同じバッファが着信メッセージとレスポンスの両方の処理に使用されることに注意してください。直接イベント処理をしている場合（スケジューラによるサービスをバイパスしている場合）、メッセージが処理され

た後でアプリケーション・バッファがきちんと解放されるように、レスポンスと同様のメッセージのチェックを確実に行ってください。

アプリケーション・バッファの割り当て

通常、アプリケーションがメッセージを作成するときに、Neuron チップファームウエアがアプリケーション出力バッファを自動的に割り当てます。また、アプリケーションがクリティカル・セクションを抜ける時に、処理の終わっていないメッセージ・バッファをファームウエアが自動的に解放します。

次の関数を使用すれば、明示的にアプリケーション・バッファを割り当てたり解放したりできます。

boolean msg_alloc (void);

boolean msg_alloc_priority (void);

void msg_free (void);

メッセージは優先パスと非優先パスの 2 種類のパスのどちらかを通って移動します。その名前からわかるように、優先パスは非優先パスより先に処理されます。したがって、**msg_alloc_priority()**関数を使ってアプリケーション出力バッファを割り当てれば、ネットワークが渋滞していてもメッセージの伝達が成功しやすくなります。

msg_alloc()と **msg_alloc_priority()**関数は、**msg_out** オブジェクトがバッファに割り当てられれば TRUE を返し、割り当てられなければ FALSE を返します。送信メッセージの作成時にアプリケーション出力バッファが空くのを待たないプログラムを作成するには、これらの関数を使用してください。そうすれば、これらの関数が FALSE を返した場合にはとりあえず他の処理を行い、後でリトライするという方法を取ることができます。

msg_alloc_priority()関数は、優先アプリケーション出力バッファを割り当てます。**msg_alloc()**関数は、非優先アプリケーション出力バッファを割り当てます。システムのデフォルト値を使っていれば、それぞれのタイプについて最大 2 個までのバッファを同時に使用できます（割り当て可能なバッファの最大数は、**#pragma app_buf_out_count** 指令および**#pragma app_buf_out_priority_count** 指令で設定できます。これらの指令の詳細については、第 8 章「メモリ管理」と『Neuron C Reference Guide』を参照してください）。

msg_free()関数は、**msg_in** オブジェクトが使用しているアプリケーション入力バッファを解放します。通常はタスクの終了時に自動的にメッセージ・バッファが解放されるため、自分で解放する必要はありません。タスクの中で受信したメッセージの処理が終了してアプリケーション入力バッファが必要なくなった後、タスクを終了する前に何かの都合で明示的にバッファを解放したいときに使用してください。

通常、**msg_out** オブジェクトのフィールドのどこかに値を代入すると、アプリケーション出力バッファが割り当てられます。このとき、アプリケーション・バッファが使用できない状況になっていると、バッファが使用可能になるまでアプリケーションの処理が休止します（先取りモード）。**msg_alloc()** 関数を使用すれば、バッファが使用できないときにもプロセスは休止しません。この関数は、使用できるアプリケーション・バッファがなければ FALSE

を返し、処理は続行します。このように、発信アプリケーション・バッファが使用できなくても、アプリケーションはバッファが使用できるようになるまで待たずに他の処理ができます。

アプリケーション入力バッファは、**msg_receive()**が呼び出されたクリティカル・セクションの終了時に解放されます。これより前にアプリケーション・バッファを解放したければ、アプリケーション・プログラムで **msg_free()**を呼び出します。この関数を呼び出せば、**msg_in** オブジェクトには受信メッセージが含まれませんから、ネットワーク・プロセッサは他の着信メッセージ用にこのアプリケーション入力バッファを使うことができます。**msg_alloc()** と **msg_free()**関数は、標準のメモリ割り当て関数とは異なることに注意してください。**msg_alloc()**関数によって割り当てられたアプリケーション出力バッファを **msg_free()**関数で解放することはできません。しかし、**msg_send()** や **msg_cancel()**の呼び出しは **msg_alloc()**によって割り当てられた出力バッファを自動的に解放し、**msg_free()**呼び出しは **msg_receive()**によって割り当てられた入力バッファを自動的に解放します。

レスポンスのバッファ割り当てには、次のような関数があります。

boolean resp_alloc (void);

void resp_free (void);

次の例では、2 つのメッセージを作成するタスクを示します。各メッセージの作成や送信の前に、**msg_alloc()**を使ってバッファが利用できるかどうかをチェックしています。

```
msg_tag motor1;
msg_tag motor2;
#define MOTOR_ON 0

when (x == 2)
{
    if(msg_alloc() == FALSE)
        return;

    msg_out.tag = motor1;
    msg_out.code = MOTOR_ON;
    msg_send();

    if(msg_alloc() == FALSE)
        return;

    msg_out.tag = motor2;
    msg_out.code = MOTOR_ON;
    msg_send();
}
```

7

その他の機能

本章では、第 6 章までに説明しなかった Neuron C の機能について説明します。ここでは、スケジューラのリセット・メカニズムについてさらに詳しく説明します。Neuron ファームウエアの持つスケジューリング・アルゴリズムと異なるアルゴリズムが必要な場合は、バイパス・モードで実行し、**post_events()**関数を使用します。この方法も本章で説明しています。この他、スリープ・モード、エラー処理、ステータス・レポートについても説明します。

スケジューラ

第 2 章「シングル・デバイスでの機能」では、図 7.1 に示す Neuron ファームウエア・スケジューラの基本機能について紹介しました。優先 **when** 節は、スケジューラが起動するたびに指定された順序で評価されます。優先 **when** 節のどれかが TRUE と評価されると、そのタスクを実行した後、スケジューラの処理は最初に戻ります。優先 **when** 節のどれも TRUE と評価されなかった場合、非優先 **when** 節をラウンドロビン方式で選択して評価します。非優先 **when** 節が TRUE と評価されると、そのタスクが実行されます。非優先 **when** 節が FALSE と評価されれば、その節のタスクは無視されます。いずれの場合も、スケジューラは「評価ループの始まり」に戻ります。

スケジューラのリセット・メカニズム

スケジューラのリセット・メカニズムは通常無効になっています。リセット・メカニズムを有効化すると、以下のいずれかにあてはまるときに非優先 **when** 節のラウンドロビン部分が最初の非優先 **when** 節にリセットされます。

- キューの先頭に新規のネットワーク変数更新情報があるとき
- 新規のタイマー時間切れイベントが起こったとき
- キューの先頭に新規のメッセージがあるとき

これらのイベントはどんなときにでも起こる可能性はありますが、スケジューラはループの先頭（図 7.1 の中の「評価ループの始まり」ラベルがあるところ）から調べていきます。

リセット・メカニズムをオフにすると、非優先 **when** 節はプログラム上の順序で評価されます。そして、非優先 **when** 節の最後に到達すると、スケジューリングのループの最初にある非優先 **when** 節に戻ります。

when 節が評価される順番を指定したい場合は、次のコンパイラ指令を使用し、リセット・メカニズムを起動してください。

```
#pragma scheduler_reset
```

警告: スケジューラのリセット・メカニズムが有効になっている場合、スケジューラが頻繁にリセットされると、プログラムの後ろの方にある **when** 節が一度も実行されなくなる危険があります。ネットワーク変数やメッセージの処理タスクは到着した順序でしかバッファを処理できないため、アプリケーションのバッファが不足してしまう場合があります。したがって、スケジューラのリセット・メカニズムが有効になっている場合には、最も頻繁に実行されるものが最後に来るようにプログラムの **when** 節とタスクを配置するか、たまにしか実行されないものを **priority** キーワードを使用して宣言する必要があります。

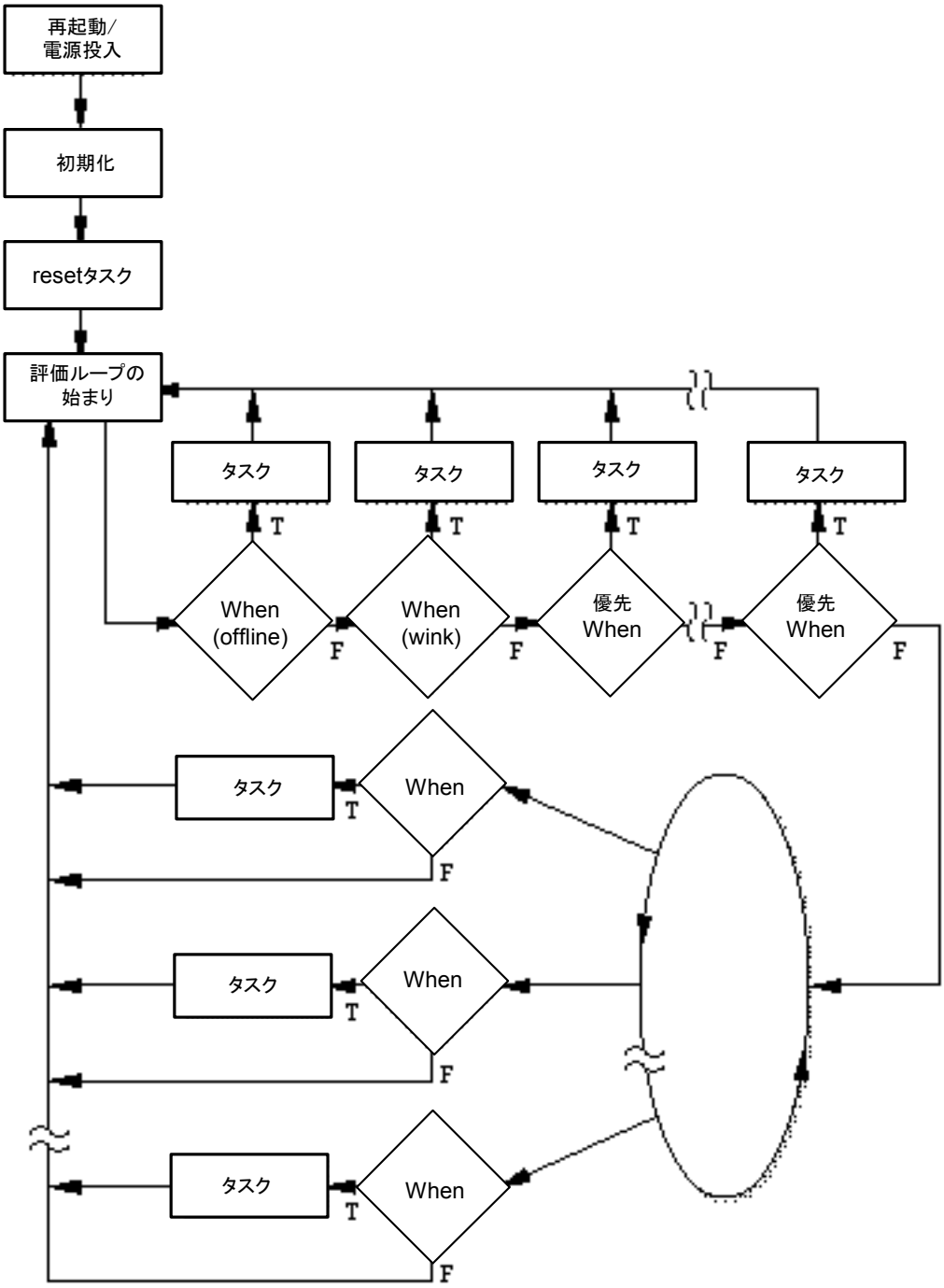


図 7.1　Neuron ファームウエアのスケジューリング

着信メッセージとネットワーク変数の更新には、アプリケーションの入力バッファを使用します。このバッファはそれぞれ **when(msg_arrives)**タスクと **when(nv_update_occurs)**タスクによって処理されます。

発信メッセージ、ネットワーク変数の更新、およびネットワーク変数のポーリング情報はアプリケーションの出力バッファに格納されます。次のいずれかのタスクを使用して完了イベントを確認すると、完了イベント・タスクに対応したアプリケーションの出力バッファがスケジューラによって自動的に処理され、解放されます。

when (nv_update_completes)

when (nv_update_succeeds)

when (nv_update_fails)

when (msg_completes)

when (msg_succeeds)

when (msg_fails)

対応する完了イベント・タスクがない場合は、対応するイベントが破棄されたときにスケジューラによって出力バッファが自動的に解放されます。

いずれの場合においても、着信イベント・キューまたは完了イベント・キューの先頭にあるアプリケーション・バッファを処理する **when** 節に到達しなかった場合（スケジューラが頻繁にリセットされる場合など）は、アプリケーション・バッファが一度も処理されず、解放されなくなるため、いつまでもキューがブロックされ、残ることになります。

したがって、スケジューラのリセット・メカニズムを使用するときは、頻繁に発生するイベント（継続的に発生する I/O イベントや、継続的に時間切れになる短い間隔のタイマー・イベントなど）がメッセージ通信イベントの処理をロックアウトしないように注意して各 **when** 節を配置してください。

コード例

リセット・メカニズムを有効にすると、イベントを意図した順序で確実に処理できます。例えば、特定のイベントが必ず最初にチェックされるように指定することができます。次のプログラムでは、非限定イベントがイベントを全部捕まえてしまう前に、特定のイベントがチェックされるようにしています。

```
#pragma scheduler_reset
network input int NV1, NV2, NV3, NV4;

when (nv_update_occurs(NV1))
{
   . . .
}

when (nv_update_occurs(NV2))
{
   . . .
}
```

```
when (nv_update_occurs)
{                   // provides a generic check
   . . .            // for all network variable
                    // updates
}
```

NV1 の更新を受信すると、1 番目と 3 番目のイベントが TRUE になります。同様に、**NV2** の更新があると、2 番目と 3 番目のイベントが TRUE になります。このようにネットワーク変数の更新があったとき、**when** 節が順番どおりに評価されることが重要になります。**scheduler_reset** を指定しておけば、新しいネットワーク変数の更新がキューの先頭にあるときには評価ループの最初に戻りますから、**NV1** の **nv_update_occurs** が必ず最初にチェックされます。

バイパス・モード

前述のとおり、Neuron C プログラムはすべてイベント駆動であり、スケジューラがそれを処理しています。ただし、プログラムの中でスケジューラに制御を戻さずにイベントを処理することもできます。「バイパス・モード」とは、1 つの **when** 節を常に TRUE にしておき、**when** 節のタスクが終了しないようにしておくプログラミング技法です。この場合、1 つのタスクですべてのイベントを処理しなければなりません。

バイパス・モードが必要になることは滅多にありません。バイパス・モードは Neuron ファームウエアのスケジューラが使うスケジューリング･アルゴリズムと異なるものが必要になったときに使用します。バイパス・モードになっているときは、イベント処理に対する全責任がプログラムにあります。バイパス・モードのタスクの中では、**post_events()**関数（次のセクションを参照してください）を使ってクリティカル・セクションを定義し、**if**、**while**、**for** 式を使って定義済みイベントをチェックします。

post_events( )関数

クリティカル･セクションの境界を定義するには、**post_events()**関数を使用します。この関数が呼ばれたとき、ネットワーク変数の更新が情報を送受信し、処理を実行します。

注意： **post_events()**関数は、スケジューリング・ループの先頭で自動的に呼ばれます。

post_events()関数が呼ばれると、次の事項が実行されます。

- ネットワーク変数の更新を送信します。Sync 型ネットワーク変数の更新は、すべて送信します。Nonsync 型ネットワーク変数の更新は、使用可能なアプリケーション出力バッファ数だけ送信します。このとき送られなかった更新情報は、次に **post_events()**が呼ばれた時に送信されます。
- ネットワーク変数の更新を受信します。
- 新規着信メッセージを処理します。
- タイマーが時間切れになっていないかを確認します。
- ウォッチドッグ・タイマーをリセットします（タイムアウトを防ぎます）。ウォッチドッグ･タイマーについては、次のセクションを参照してください。

ネットワーク変数の変更直後に **post_events()**関数を呼ぶと、ネットワーク処理の性能が向上します。**post_events()**関数を呼び出すと、**when** 節のタスクが終了する前にネットワーク・プロセッサが発信パケットのフォーマット処理を開始できます。そのため、並列処理能力を高め、Neuron チップおよびスマート・トランシーバのマルチプロセッサ・アーキテクチャを最大限に活用できます。

ウォッチドッグ・タイマー

ウォッチドッグ・タイマーは、入力クロックを 40MHz とすると 0.21 から 0.42 秒の範囲でタイムアウトします（この範囲は入力クロックに反比例します）。ハードウエア・タイマーの期間は 0.21 秒ですが、現在の期間が始まってからウォッチドッグ・タイマーが再トリガされていない場合にのみ、現在の期間の終わりにタイムアウトが発生します。ソフトウエアのタイマー・リトリガはハードウエアのタイムアウト期間と非同期です。そのため、ソフトウエアの観点では、再トリガされてからタイムアウトまでの最小時間は 1 つの期間、つまり 0.21 秒で、再トリガされてからタイムアウトまでの最大時間は 2 つの期間、つまり 0.42 秒になります。

ウォッチドッグ・タイマーの目的は、ソフトウエアがタイマーを再トリガしなくなるような無限ループやその他の問題によってソフトウエアに障害が発生したときに、1 秒という時間内でデバイスをリセットすることです。スケジューラによってウォッチドッグ・タイマーが定期的にリセットされることが保証されるため、アプリケーション・プログラムがウォッチドッグ・タイマーを気にする必要はほとんどありません。しかし、プログラムが非常に長いタスクに入るようなことがあると、ウォッチドッグ・タイマーが時間切れになって、デバイスがリセットされてしまう可能性があります。

ウォッチドッグ・タイマーがタイムアウトしないようにするには、時間のかかるタスク（またはバイパス・モードのときなど）の中で定期的に **watchdog_update()**関数を呼び出します。その他ウォッチドッグ・タイマーを更新するものとしては **post_events()**、**msg_receive()**、**resp_receive()**の各関数、**pulsecount** 出力オブジェクトを持つ **io_out()**関数および **magcard**、**magtrack1**、**neurowire slave**、**wiegand** の各入力オブジェクトを持つ **io_in()**関数が挙げられます。

注意： **watchdog_update()**関数は注意して使用し、可能であれば、ループ内での使用は避けてください。ループが終了しなくなるようなソフトウエアまたはハードウエアの障害が発生すると、デバイスが応答しなくなることがあります。また、ループ本体ではウォッチドッグ・タイマーを継続的に再トリガしてしまい、このような状況からの回復が不可能になります。

注意： EEPROM に書き込むファームウエア関数は、ウォッチドッグ・タイマーを自動的には更新しません。

watchdog_update()関数の使用例を以下に示します。

```
when (TRUE)
{
   post_events();
   if (nv_update_occurs(NV1)){
      .
      .
      .
   } else if (nv_update_occurs(NV3)){
      . // long task
      .
      .
      watchdog_update();
      . // more long task
      .
      .
   }
}
```

その他の定義済みイベント

次の 3 つの定義済みイベントは、ネットワーク管理メッセージから送られるイベントです。

offline

online

wink

offline イベントは、ネットワークツールから **offline** ネットワーク管理コマンドを受信したときに発生します。このイベントは最優先 **when** 節として扱われます。**online** イベントは、ネットワークツールから **online** メッセージを受信したときに発生します。**wink** イベントは、ネットワーク管理ツールから **wink** コマンドを受信したときに発生します。

offline イベントはデバイスをオフライン状態にするために使用され、緊急のとき、保守用、その他のシステム全体に関わる状態への対応として発生します。一度オフラインになると、デバイスがリセットされるかオンライン状態に復帰するまで、ネットワーク管理メッセージにしか応答しなくなります(リセットは、Neuron チップまたはスマート・トランシーバのリセット・ラインをアクティブにしてデバイスのリセットを物理的に行うか、**reset** ネットワーク管理メッセージを使用します)。**when(offline)**節のタスクが実行された後、デバイスがリセットされるかオンライン状態に復帰するまで、アプリケーション・プログラムも実行されません。

online と **offline** イベントの使用例を以下に示します。

```
when (offline)
{
   x();      // Clean up before going offline.
}            // Device goes offline here; application
             // program stops running.

when (online)
{
   y();      // Start up again (poll inputs,
             // and so on)
}
```

アプリケーションには、オフラインまたはオンラインへの状態の切り替えを拒否する手段がありません。アプリケーションは周辺ハードウエアの無効化やタイマーの停止など、状態を切り替える準備を行います。そのタスクが終了すると、状態の切り替えが有効になります。

デバイスは、**when(online)**節が TRUE に評価されなくても、オンライン状態に切り替えることができます。デバイスがオフラインからソフトオフライン状態になる場合にデバイスをリセットすると、ソフトオフライン状態は失われ（破棄され）、デバイスは通常のオンライン動作に戻ります。この状況に対処するには、以下の方法を使用します。

```
void HandleOnline (void)
{
   ...
}


when (reset)
{
      // regular reset code here:
      .. .
      // handle case of device going online
      if (online) {
            HandleOnline();
      }
}


when (online)
{
      HandleOnline();
}
```

バイパス・モードでのオフライン

バイパス・モードのときのように **offline** イベントを **when** 節の外でチェックする場合には、**offline_confirm()**関数を使ってください。**offline_confirm()**関数はデバイスの状態をオフラインに設定して直ちに戻ります。デバイスのクリーンアップの完了とオフラインへの移行を確認するには、この関数を使用してください。

次のプログラム例では、バイパス・モードでデバイスがオフラインになっています。ただし、プログラムの処理は続行します。バイパス・モードでは、デバイスがオフラインのときに処理するイベントあるいは処理しないイベントを選択できます。

バイパス・モードで **offline_confirm()**を使用する例を以下に示します。

```
when (TRUE)
{
      while (TRUE) {
            post_events();
            if (online)
                  continue;
            if (nv_update_occurs) {
                  ...
            } else if (offline) {
                  x();
                  offline_confirm();
                  // Wait for online
                  while (!online) {
                        post_events();
                  }
            } else {
                  ...
            }
      }
}
```

Wink イベント

wink イベントは、ネットワーク管理ツールが送出する *wink* ネットワーク管理メッセージに応答します。ネットワーク・インテグレータが特定のデバイスを物理的に識別したい場合、ネットワーク管理ツールを使用して *wink* メッセージをデバイスに送信します。デバイスが構成済みか未構成かに関わらず、*wink* メッセージを受信すると **wink** イベントが TRUE になります。

未構成デバイスでは、**wink** イベントが評価される前に I/O と変数が初期化されます。ただし、**when(reset)**節のタスク中の初期化は行われません。また、未構成デバイスではスケジューラが動いていないので、イベントは直接イベント処理でしか処理できません。また、ネットワーク変数の更新やメッセージも、デバイスが未構成であるために送信されません。**wink** タスクの中では、タイマー・オブジェクトをセットすることも、読み込むこともできます。また、最初に **post_events()**を呼び出すと、**timer_expires()**イベントを明示的にチェックすることもできます。

スリープ・モード

「スリープ・モード」を使用すると、Neuron チップまたはスマート・トランシーバを低電力状態にすることができます。Neuron チップまたはスマート・トランシーバをスリープ・モードにするには、次のステップを実行します。

1. 未処理のネットワーク変数の更新および未処理の発信/着信メッセージをすべてフラッシュします。
2. フラッシュが完了したときに、Neuron チップまたはスマート・トランシーバをスリープ・モードにします。スリープ・モードになった Neuron チップまたはスマート・トランシーバは、サービス・ピンがアクティブになったとき、または I/O ピン（選択したピンは変更可）や通信チャネルに何か動作があったときにウェイクアップします。

コード例

```
mtimer m_30;
network output SNVT_switch nvoValue;
static SNVT_switch temp;

when (timer_exp(m_30))
{
   nvoValue = temp;
   flush(TRUE);
}


when (flush_completes)
{
   sleep(COMM_IGNORE);
}
```

Neuron チップまたはスマート・トランシーバのフラッシュ処理

flush()関数は、Neuron ファームウエアに対してすべての発信/着信メッセージの処理を完了するよう指示を出します。フラッシュが完了すると、**flush_completes** イベントが TRUE になります。

flush( )関数と flush_cancel( )関数

flush()関数を呼び出すと、Neuron ファームウエアは発信/着信メッセージの状態をすべてモニターするようになります。**flush()**関数のシンタックスは次のとおりです。

flush (boolean *comm-ignore***);**

comm-ignore TRUE を指定すると、Neuron ファームウエアがフラッシュ中に通信チャネルの活動状態を無視します。FALSE を指定すると、Neuron ファームウエアは着信メッセージを受信します。このパラメータには、この後に使用する **sleep()**関数の **comm_ignore** パラメータと同じ値を指定してください。

フラッシュの実行中もプログラムは続行しています。フラッシュ中にプログラムが新しいメッセージを作成することはできますが、単にフラッシュの完了を遅らせるだけです。

comm_ignore オプションが TRUE に設定されていると、フラッシュ中に新しく到着したパケットがあったとき、それらが確認応答、レスポンス、チャレンジ、リプライでない限り廃棄されます。

flush_cancel()関数を呼び出すと、進行中のフラッシュ操作を取り消すことができます。

flush_completes イベント

フラッシュが完了すると、次の定義済みイベントの値が TRUE になります。

flush_completes イベントのシンタックスは以下のとおりです。

flush_completes

このイベントは、発信用ネットワーク・バッファとアプリケーション・バッファがすべて解放され、着信メッセージにもネットワーク変数の更新にも未処理状態のものがなくなったときに TRUE になります。

注意： デバイスをスリープ・モードにする準備として **flush_wait()**関数を使用しないでください。**flush_wait()**関数は未処理状態のネットワーク変数の更新や着信メッセージはチェックしません。

デバイスのスリープ

flush_completes イベントが TRUE になったときに、**sleep()**関数を使って Neuron チップまたはスマート・トランシーバをスリープ状態にできます。**sleep()**関数のシンタックスは以下のとおりです。

sleep (*flags***)**

sleep (*flags***,** *io-object-name***)**

sleep (*flags***,** *io-pin***)**

flags

スリープ状態の属性を指定するフラグを指定します。フラグが必要ないときは、ここに 0 を指定します。複数のフラグを使用するときには、各フラグを OR でつないでください。ここに指定できるフラグは次のいずれかです。

COMM_IGNORE	着信メッセージを無視します。
PULLUPS_ON	すべての内部プルアップ抵抗を使用可能にします（デフォルトではプルアップは使用不可になっています。この方が電力消費が少なくなります）。
TIMERS_OFF	プログラムにあるすべてのタイマー・オブジェクト（**mtimer** や **stimer** で宣言したもの）を停止します。

io-object-name

I/O ピンに対応する入力オブジェクトを指定します。ただし、適用できる I/O ピンは IO_4 から IO_7 ピンまでの 1 本です。選択したピンで何らかの I/O が発生すると、デバイスがウェイクアップします。このパラメータを省略すると、デバイスがスリープ・モードに入った後の I/O は無視されます。この I/O オブジェクトは、ウェイクアップ機能のためだけに使用することもできますし、他の入出力処理のためにも使用できます。

io-pin

IO_4 から IO_7 ピンまでの 1 本を指定します。指定したピン上で何らかの I/O が発生すると、Neuron チップまたはスマート・トランシーバがウェイクアップ

します。このパラメータを省略した場合、デバイスがスリープ・モードに入った後の I/O は無視されます。

例えば、タイマーを停止し、プルアップ抵抗を使用可能にしてスリープ状態にするには、以下のように **sleep()**を呼び出します。

sleep(TIMERS_OFF | PULLUPS_ON);

コード例

```
IO_4 input bit wakeup_pin1;

when timer_expires(timer_2)
{
    sleep(COMM_IGNORE, wakeup_pin1);
       //or, sleep (COMM_IGNORE, IO_4);
}
```

フラッシュが完了する前にデバイスを強制的にスリープ・モードにすることもできます。これについては、次の「強制スリープ」のセクションで説明します。

Neuron チップまたはスマート・トランシーバをウェイクアップするイベントが発生したとき、プログラムはスリープ関数の呼び出し直後から再開します。**sleep()**関数呼び出しがタスクの最後にあれば、プログラムはウェイクアップ後にスケジューラに戻ります。

トランシーバがパケットを受信すると、（**COMM_IGNORE** が指定されていない限り）デバイスはウェイクアップします。ここで受け取るパケットは、そのデバイス宛のものである必要はありません。なお、ウェイクアップ後に再びデバイスをスリープ状態にする方法についても必ず考慮してください。

デバイスが受信タイマー間隔よりも短い時間だけスリープ状態になるような場合、**COMM_IGNORE** オプションを指定していると、メッセージやネットワーク変数更新のイベントを重複して受信する可能性があります。デフォルトの受信タイマーは、デバイスのインストール時にネットワーク管理ツールによって設定します。この値はハードウエアで 768ms に設定されています。これは、デバイスへのネットワーク接続に応じてネットワーク管理ツールが増加できる最小値です。

強制スリープ

フラッシュ操作が完了していなくても、強制的にデバイスをスリープ状態にすることができます。極端なネットワークの輻輳などの特殊なネットワーク状態では、フラッシュに長時間かかることもあります。アプリケーションは、フラッシュ完了を待つのを止めてフラッシュが完了したかどうかに関係なくスリープ状態にし、電源を過大に消費しないようにできます。

デバイスを強制的にスリープ状態にするには、**flush_completes** イベントを待たずに **sleep()**関数を呼びます。デバイスを強制的にスリープ状態にするコード例を以下に示します。

```
...
   flush(TRUE);          // start flush; ignore
                                 // incoming packets
   flush_timeout = 300; // start flush timeout
                                 // timer (300 msec)
}


when (timer_expires(flush_timeout))
when (flush_completes)
{     // Ready to go to sleep since the flush
            // either completed or timed out
      flush_timeout = 0;       // First, turn off timer
                                 // if not expired
      sleep(COMM_IGNORE);
}
```

強制的にスリープ・モードにすると、次のようなことが起こります。

1 処理途中のネットワーク変数更新、未完了のアプリケーション出力バッファ、未完了のネットワーク出力バッファは、どれも送信されずに開放されます。

2 **COMM_IGNORE** オプションを指定すると、着信ネットワーク・バッファがすべて解放されます。

3 未完了の着信アプリケーション・バッファが残っていると、デバイスはスリープ状態になりません（**COMM_IGNORE** オプション指定には関係ありません）。この機能は、デバイスが再起動したときに古いメッセージを受信することを防ぐためにあります。上記で示した例では、キューに既に入っている着信メッセージを処理するために 300 ミリ秒だけ待つようにしています。さらに、**flush()**関数呼び出しで **COMM_IGNORE** パラメータを TRUE に設定しているので、新規に着信メッセージを受け取ることはありません。したがって、タイムアウトになるまでの 300 ミリ秒間に **flush()**呼び出し以前にあった処理途中の着信メッセージはすべて処理できると仮定して、デバイスをスリープ状態にしています。

エラー処理

アプリケーション・エラーから回復するか、アプリケーション・エラーを報告するには、デバイスのリセット、アプリケーションの再起動、アプリケーションのオフラインへの移行、機能ブロックの無効化、機能ブロックのステータスの変更、エラー・ログ記録のいずれかを実行します。なお、Neuron ファームウエアが検出したシステム・エラーはログに記録されます。これらの動作は組み合わせて使用することも可能です。例えば、エラーをログに記録してからアプリケーションをオフラインに移行することができます。または、機能ブロックを無効にして機能ブロックのステータスを変更することもできます。

デバイスのリセット

node_reset()関数を呼び出して、デバイスをリセットできます。この関数は、Neuron チップまたはスマート・トランシーバにあるすべてのプロセッサ（アプリケーション、ネットワーク、MAC）を直ちにリセットします。Neuron リセット・ピンをローに駆動するため、これを外部のトランシーバや論理回路のリセットに利用できます。通常この動作は、ハードウエアのリセットを必要とする致命的なエラーに対して行います。

デバイスをリセットすると、すべての初期化処理が実行されます。初期化に必要な時間の長さは、アプリケーション・プログラムの大きさと、オフチップ・メモリの容量に依存しますが、18 秒未満でアプリケーションがオンラインになる必要があります。リセット時間の詳細な計算式については『FT 3120 and FT 3150 Smart Transceivers Databook』を参照してください。

デバイスのリセットには、多くの欠点があります。まず、デバイスをリセットすると、EEPROM に保存されていないステータス情報が失われます。処理途中の発着信メッセージおよびネットワーク変数の更新は、すべて無効になります。ネットワーク・プロセッサは、重複したパケットを受信する可能性があります。さらに、ネットワーク・プロセッサが確認応答を返していても、アプリケーションがまだ処理していないパケットは失われます。

アプリケーションの再起動

application_restart()関数を使用してアプリケーション・プロセッサをリセットすることはできますが、ネットワーク・プロセッサや MAC プロセッサはリセットできません。通常この動作は、外部のハードウエアをリセットすることなく、アプリケーションを再起動するだけで回復できるアプリケーション・エラーに対して行います。

この関数が呼ばれると、Neuron ファームウエアはすべてのタイマー・オブジェクトをクリアし、I/O オブジェクト、未構成のネットワーク変数、および **static** 変数の初期化を行ってから、**when(reset)**節を実行します。同期のためのクリーンアップ処理を終えてから、アプリケーションが再起動します。処理中の発信メッセージは、どれも中断されます。着信メッセージには影響ありません。未処理の完了イベントとレスポンスは廃棄されます。アプリケーションの再起動でネットワーク・ステータス情報が失われることはありません。アプリケーション・プロセッサだけがリセットされるため、ネットワークと MAC プロセッサはネットワーク通信の処理を続行します。

アプリケーションのオフラインへの移行

go_offline()関数を使用すると、デバイスをオフラインに移行できます。通常この動作は、デバイスをリセットするか、アプリケーションを再起動してもエラーが修正されない場合、およびエラーがデバイス上の特定の機能ブロックに特定されない場合に実行します。

go_offline()関数を呼び出すと、未処理のトランザクションがすべて終了し、アプリケーションの処理がすべて停止します。**when(offline)**タスクで **flush_wait()**関数を呼び出すと、未処理のトランザクションが正常に完了することを保証できます。

Neuron ファームウエアは、デバイスがオフラインのときも継続して実行され、ネットワーク管理ツールを使用しているネットワーク・インテグレータがデバイスのステータスをテストし、必要な修正を施してアプリケーションをオンラインに戻すことができるようになっています。アプリケーション・エラーは後述の「アプリケーション・エラーのログ」の説明に従ってログに記録することで、オフラインになった理由をネットワーク・インテグレータに伝えることができます。

機能ブロックの無効化

デバイスをリセットするか、アプリケーションを再起動しても修正されないエラーのうち、デバイス上の特定の機能ブロックまたは機能ブロックのセットに特定されるものに対しては、個別の機能ブロックを無効にできます。機能ブロックのステータスは、Neuron C 言語に組み込まれていませんが、機能ブロックのステータスを管理するためのコードは NodeBuilder コード・ウィザードによって自動的に生成されます。コード・ウィザードを使うと、アプリケーション内の機能ブロックごとの機能ブロック・ステータスを保持する **fblockData[]**配列が作成されます。この配列のメンバは **SNVT_obj_status** 型を使用して宣言します。機能ブロックを無効にするには、次のコードを使用します。

```
fblockData[fblockIndex].objectStatus.disabled = TRUE;
```

fblockIndex パラメータには、無効にする機能ブロックのインデックスを指定します。

アプリケーションには、機能ブロックのステータスをテストするためのコードを含める必要があります。コード・ウィザードによって生成される **fblockNormalNotLockedOut()**関数を使用すると、機能ブロックのステータスをテストできます。この関数のシンタックスは以下のとおりです。

boolean fblockNormalNotLockedOut(TFblockIndex ***fblockIndex*****);**

fblockIndex パラメータにはテストする機能ブロックのインデックスを指定します。

例えば、以下の例ではネットワーク変数の入力に関連付けられている機能ブロックのステータスがテストされます。

```
if
(fblockNormalNotLockedOut(fblock_index_map[nv_in_index]))
{
      . . .
}
```

fblockNormalNotLockedOut()関数の他の使用例については、『NodeBuilder User's Guide』の付録、「NodeBuilder Example」を参照してください。

以下の「機能ブロックのステータスの変更」で説明するように、機能ブロックのステータスを変更すると、機能ブロックを無効にする理由をネットワーク・インテグレータに伝えることができます。

機能ブロックのステータスの変更

Node Object 機能ブロックの **nvoStatus** 出力を使用すると、機能ブロックのエラー状態を報告できます。デバイス上の各機能ブロックにはそれぞれ独立したステータスが存在するため、ネットワーク管理ツールでは Node Object 機能ブロックの **nviRequest** 入力を使用して、個々の機能ブロックのステータスをリクエストします。リクエストされたステータスは、**nvoStatus** 出力を経由して報告されます。

機能ブロックのステータスは Neuron C 言語に組み込まれているわけではありませんが、「機能ブロックの無効化」で説明しているように、機能ブロックのステータスを管理するためのコードは NodeBuilder コード・ウィザード

によって自動的に生成されます。機能ブロックのステータスは、**fblockData[ ]** 配列内の適切なフィールドを設定することで更新できます。各フィールドの説明については、『LONMARK SNVT and SCPT Guide』または NodeBuilder リソース・エディタの **SNVT_obj_status** 型の定義を参照してください。

例えば、次の文は *fblockIndex* によって指定した機能ブロックのステータスを更新して、機械的な障害を報告します。

```
fblockData[fblockIndex].objectStatus.mechanical_fault = TRUE;
```

アプリケーション・エラーのログ

デバイスのエラー状態は、**error_log()**関数を使用して報告します。この関数には、1 から 127 までのエラー番号が渡されます。また、この関数は EEPROM にある専用の領域に最新のエラー番号を書き込みます。ネットワーク管理ツールは、*query status*（ステータスを問い合わせる）ネットワーク診断用コマンドを使ってこのエラーを読むことができます。**error_log()**関数のシンタックスは以下のとおりです。

void error_log (unsigned int *error_num***);**

1 から 127 のエラー番号の値はアプリケーションによって定義されます。この範囲の番号をデバイスのエラー状態に割り当てて、割り当てた内容をデバイスの記録の一部として記録することができます。

LonBuilder Neuron C デバッガは、最新の 25 個までのエラー・メッセージを管理します。Neuron エミュレータでは、Neuron ファームウエアのエラーをログに書き込んでから PC が最新値を取り出すまでに最高 70ms の遅れが発生します。

システム・エラー

Neuron ファームウエアはアプリケーション・エラーの報告に使われるものと同じエラー・ログを使用してシステム・エラーを報告します。システム・エラーには、プログラミング・エラー、ネットワーク・エラー、システム矛盾があります。アプリケーション・エラーと同じように、ネットワーク管理ツールでは *query status*（ステータスを問い合わせる）ネットワーク診断用コマンドを使用して、エラー・ログから最新値を取得できます。

システム・エラーには 128～255 の番号が付いています。各システム・エラーのメッセージについては、『NodeBuilder Errors Guide』を参照してください。

エラー情報へのアクセス

アプリケーション・プログラムからは、ネットワーク管理ツールが使用できるものと同じ診断ステータス情報にアクセスできます。ステータス情報はステータス構造体に保存されており、**retrieve_status()**関数を使用してこの情報を検索できます。**retrieve_status()**関数のシンタックスは以下のとおりです。

void retrieve_status (status_struct ****status-p*);**

ステータス構造体の各フィールドについては、『Neuron C Reference Guide』を参照してください。ステータス構造体の特定のフィールド（統計情報、リセット発生レジスタ、エラー・ログ）をクリアするには、**clear_status()**関数を使用します。

コード例

```
#define unconfigured          0x02
#define config_on_line        0x04
#define config_off_line 0x0C
#define power_up_reset        0b1
#define power_up_reset_mask   0b1
#define external_reset        0b10
#define external_reset_mask   0b11
#define WDT_reset       0b1100
#define WDT_reset_mask        0b1111
#define SI_reset        0b10100
#define SI_reset_mask         0b11111

#include <status.h>
status_struct status;   // structure type defined
                        // in <status.h>

unsigned long     transmission_errors;
unsigned long     transaction_timeouts;
unsigned long     receive_transaction_full;
unsigned long     lost_messages;
unsigned long     missed_messages;
unsigned long     reset_cause;
unsigned short    node_state;
unsigned short    version;
unsigned short    error_log;
unsigned short    model_number;


retrieve_status(&status);
      // obtain device status structure
transmission_errors = status.status_xmit_errors;
      // number of received packets with CRC errors
transaction_timeouts =
      status.status_transaction_timeouts;
      // number of timeouts using Ackd or Req/Resp
      // transactions

receive_transaction_full =
      status.status_rcv_transaction_full;
      // number of times incoming message (other than
      // Unackd) was lost due to receive transaction
      // database overflow

lost_messages = status.status_lost_msgs;
      // number of times incoming message was lost
      // because there was no application buffer
missed_messages = status.status_missed_msgs;
      // number of times incoming message was lost
      // because there was no network buffer
reset_cause = status.status_reset_cause;


if ((reset_cause & power_up_reset_mask) ==
      power_up_reset) {
      // last reset was a power_up
}
```

```
if ((reset_cause & external_reset_mask) ==
      external_reset) {
      // last reset was from the NEURON RESET pin
}


if ((reset_cause & WDT_reset_mask) ==
      WDT_reset ) {
      // last reset was from the watchdog timer
      // timing out
}


if ((reset_cause & SI_reset_mask) == SI_reset ) {
      // last reset was software initiated by a
      // call to node_reset()
}


node_state = status.status_node_state;
if (node_state == unconfigured) {
      // this device has not been configured
}
if (node_state == configured_online) {
      // this device is running its application
}
if (node_state == configured_offline) {
      // this device is not running its application
}


version = status.status_version_number;
      // version number of Neuron firmware
error_log = status.status_error_log;
      // most recent error logged by system
model_number = status.status_model_number;
      // model number of Neuron Chip or Smart Transceiver
```

8

メモリ管理

本章では、オンチップ EEPROM、アプリケーション・バッファ、ネットワーク・バッファといったアプリケーションが使用するシステム・メモリ・リソースについて説明します。ここでは，メモリ・リソースの再割り当ての方法、およびどういう時に再割り当てが必要になるかを説明します。

オンチップ EEPROM の再割り当て

Neuron C コンパイラは、オンチップ EEPROM に 4 つのテーブルを生成します。Neuron ファームウエアやネットワーク管理ツールはこれらのテーブルを使用して、デバイスのネットワーク構成を定義します。4 つのテーブルのうちの 2 つはドメイン・テーブルとアドレス・テーブルと呼ばれ、デフォルトでは最大サイズで生成されます。つまり、ドメイン・テーブルには 2 つのエントリ、アドレス・テーブルには 15 のエントリが生成されます。より小さいサイズを指定するには、**#pragma num_domain_entries** 指令と**#pragma num_addr_table_entries** 指令を使用します。3 番目のテーブルである別名（エイリアス）テーブルにはデフォルトのサイズはありませんが、**#pragma num_alias_table_entries** 指令を使用してサイズを指定する必要があります。詳細については『Neuron C Reference Guide』の「Compiler Directives」の章と以下の説明を参照してください。

4 つ目のテーブルはネットワーク変数構成テーブルで、プログラム内でネットワーク変数が宣言されると、ネットワーク変数 1 つにつきテーブル・エントリが 1 つずつ生成されます。ネットワーク変数配列の各要素は別々にカウントされます。ネットワーク変数構成テーブルの最大エントリ数は 62 です。1 つのエントリにつき 3 バイトの EEPROM を使用します。このテーブルのサイズはネットワーク変数を追加または削除しない限り、変更できません。

プログラムがデフォルトのメモリ領域に収まらないとき、Neuron 3150 チップまたは FT 3150 スマート・トランシーバを使っている場合、プログラムの一部をメモリ内の他の領域に移すことができます。ただし、ドメイン・テーブル、アドレス・テーブル、別名テーブル、およびネットワーク変数構成テーブルはオンチップ EEPROM に配置する必要があります。本章の「オフチップ・メモリ」のセクションを参照してください。

アドレス・テーブル

アドレス・テーブルには、デバイスがネットワーク変数の更新やポーリングを送信する宛先のネットワーク・アドレス、または暗黙的にアドレス指定したアプリケーション・メッセージの宛先ネットワーク・アドレスのリストが書き込まれています。アドレス・テーブルを変更するには、ネットワーク管理ツールを使用してネットワーク管理メッセージを送信します。

注意： アドレス・テーブルについては『FT 3120 and FT 3150 Smart Transceivers Databook』を参照してください。

デフォルトでは、アドレス・テーブルには 15 個のエントリを登録できます。アドレス・テーブルはオンチップ EEPROM 上にあり、各エントリは 5 バイトずつ消費します。アドレス・テーブルのエントリ数を減らすには、以下のコンパイラ指令を使用します。

#pragma num_addr_table_entries *nn*
（*nn* は 0 から 15 までの数値）

デバイスが要求できるアドレス・テーブルのエントリの最大数は、そのデバイスの接続（ネットワーク変数とメッセージ・タグ）に必要な各宛先エント

リの予想最大数によって決定します。出力が、ポーリングされる出力として宣言されていない場合は、接続する出力ネットワーク変数またはメッセージ・タグに宛先エントリが必要となります。入力がポーリングされるかグループ接続のメンバになっている場合には、入力に対しても宛先エントリが必要になります。2 つの宛先エントリが異なるサービス型、異なる宛先アドレス、または繰り返しタイマーなどの異なるトランスポート属性を使用している場合、これらの宛先エントリは異なるものになります。同じ宛先エントリを使用する複数のネットワーク変数は、共通のアドレス・テーブル・エントリを共有します。複数の接続でアドレス・テーブル・エントリを共有できる場合には、消費するアドレス・テーブル・エントリが少なくなります。この機能は、デバイスをインストールするネットワーク管理ツールが共有エントリ（LonMaker 統合ツールを含むすべての LNS ツールにこの機能があります）を生成する場合にのみ利用できます。

通常、アドレス・テーブルのエントリ数は、可能な限り最大の 15 個にしてください。

別名（エイリアス）テーブル

別名テーブルは、以下に示す**#pragma num_alias_table_entries** コンパイラ指令を使用して指定されている別名テーブルのサイズに基づいて生成されます。このコンパイラ指令を使用すると、別名テーブルのサイズをゼロから 62 の範囲のエントリ数に設定できます。1 つの別名エントリにつき、オンチップ EEPROM を 4 バイト使用します。別名はネットワーク変数を抽象化したもので、ネットワーク管理ツールと Neuron ファームウエアによって管理されます。Network 管理ツールは別名を使用して、アドレス・テーブルとネットワーク変数テーブルだけでは作成できない接続を作成し、デバイスをネットワークにインストールする際の柔軟性をもたらします。この機能は、Neuron ファームウエアのバージョン 6 以降でサポートされています。

#pragma num_alias_table_entries *nn*
（*nn* は 0 から 62 までの数値）

注意： 別名テーブルについては、『Smart Transceivers Databook』を参照してください。

通常、別名テーブルのサイズは Neuron チップ内蔵のオンチップ・メモリの使用可能なサイズに設定します。これは普通、最大サイズの 62 エントリよりも小さくなります。使用可能サイズを判断するには、リンク・マップとオンチップ EEPROM エリアを確認します。オンチップ EEPROM プールに使用可能なメモリとしてメモリの容量が表示されている場合は、このメモリを別名テーブルに使用できます。まずプール内のバイト数を 4 で割り、小数点以下を切り捨てます。次に、この値を既に使用されている別名の数に追加し、使用可能な合計別名エントリ数を求めます。ただし、この値は 62 以下に制限してください。

Neuron チップの外部に付けたメモリ（オフチップ EEPROM、ROM、フラッシュなど）でさらにプログラム・コード用にメモリを使用できる Neuron3150 チップ用にプログラムが作られている場合には、別名テーブルのサイズをさらに増加できます。使用可能な追加エントリ（前の段落で求めた数を超える数）を求めるには、オンチップ・プログラム・エリアのバイト・サイズを確

認し、それを 4 で割って切り捨てます。ただし、オンチップ EEPROM にプログラム・コードがあり、プログラム・コードをオフチップに移動できることが前提となります。求めた結果は、前の段落で求めた数とは別に利用できる別名エントリ数となります。ここでも、別名の合計数は 62 以下になるようにしてください。

ドメイン・テーブル

デフォルトでは、2 つのドメインが登録できるドメイン・テーブルが構成されます。ドメインはオンチップ EEPROM 上にあり、各エントリは 15 バイトずつ使用します。ドメイン・テーブルのエントリ数は、アプリケーションではなく、デバイスがインストールされているネットワークによって決まります。ドメイン・テーブルのサイズを小さくするには、次のコンパイラ指令を使用します。

```
#pragma num_domain_entries 1
```

注意： ドメイン・テーブルについては、『Smart Transceivers Databook』を参照してください。

注意： 通常、ドメイン・テーブルは 2 つのエントリを使用します。LONMARK 相互運用性協会では、すべての相互運用可能な LONWORKS デバイスのドメイン・テーブルが 2 つのエントリを持つことを義務づけています。ドメイン・テーブルを 1 つのエントリに減らすと、認証を受けることができなくなります。

バッファ割り当て

コンパイラ指令（Pragma）を使用すると、バッファ数、バッファ・サイズ、受信トランザクション数といった Neuron ファームウエアのメモリ・リソースを設定できます。これらの値はコンパイル時にのみ設定でき、実行中に変更することはできません。図 8.1 は、アプリケーション・バッファやネットワーク・バッファがどの段階で使用されるかを説明しています。「アプリケーション・バッファ」は、アプリケーション・プロセッサとネットワーク・プロセッサの間で使用されます。また、「ネットワーク・バッファ」は、ネットワーク・プロセッサとメディア・アクセス・コントロール（MAC）プロセッサの間で使用されます。

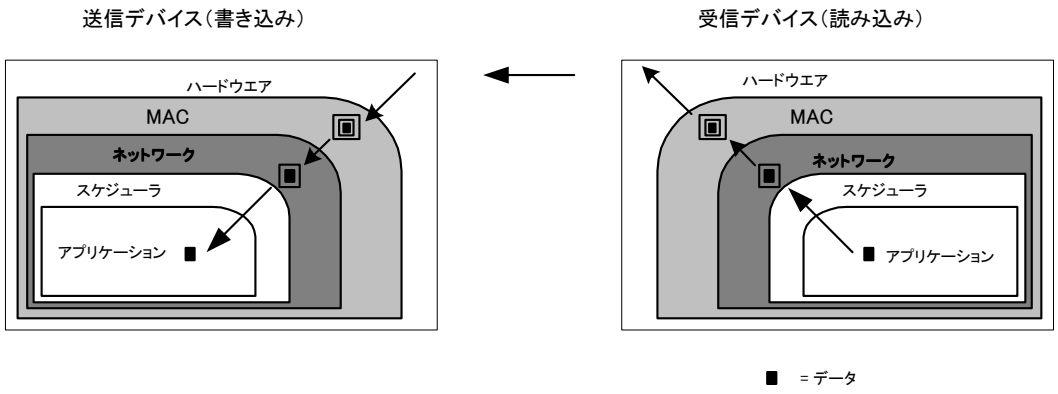


図 8.1　アプリケーション・バッファとネットワーク・バッファ

このセクションではバッファ割り当てについて概説しますが、実際のバッファ割り当てはアプリケーションのニーズによって異なります。

バッファ・サイズ

アプリケーション・メッセージを使用する場合は、アプリケーションや Neuron ファームウエアが生成または受信できる処理対象のメッセージの中で一番大きなものに対応できるだけのバッファ・サイズが必要になります。場合によっては、バッファ・サイズを増やす必要もあります。ネットワーク変数だけを使用する場合は、宣言された最大のネットワーク変数のサイズと、Neuron ファームウエアが必要とする最小のサイズに基づいて、コンパイラがバッファ・サイズを決定します。

図 8.2 には、アプリケーション・バッファとネットワーク・バッファの基本構成要素を示します。アプリケーション・バッファには、アプリケーション・メッセージ・データが使用する領域とシステム・オーバーヘッドの領域があります。ネットワーク・バッファには、アプリケーション・メッセージ・データ、プロトコル層 L2 から L5 におけるオーバーヘッド、システム・オーバーヘッドの領域があります。

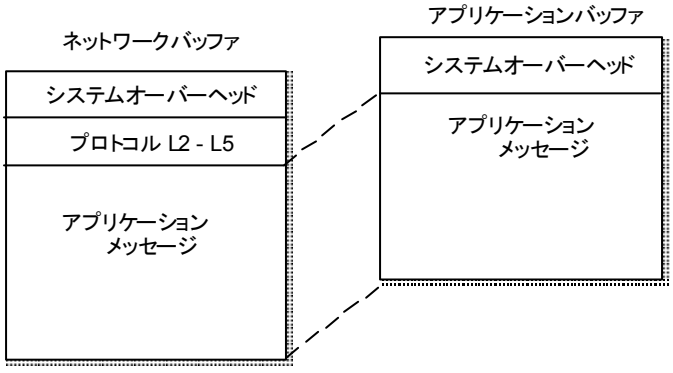


図 8.2　アプリケーション・バッファとネットワーク・バッファの
システム・オーバーヘッドとプロトコル・オーバーヘッド

アプリケーション・バッファ・サイズ

アプリケーション・バッファのサイズは、次の式で求められます。

message_size + システム・オーバーヘッド（5 バイト）

明示的アドレスを使用していれば、システム・オーバーヘッドはさらに 11 バイト大きくなります。

アプリケーション・メッセージの *message_size* は、メッセージ・コード（1 バイト）+ データのバイト数です。ネットワーク変数の *message_size* は、ネットワーク変数のバイト数 +2 バイト です。

表 8.1 は、アプリケーション・バッファのサイズとして指定できる値のリストです。例えば、*message_size* が 40 であれば、最低 45 バイトのアプリケーション・バッファが必要です。それよりも大きなアプリケーション・バッファの中で一番小さいサイズを調べると、50 バイトということになります。

アプリケーション・バッファは、ネットワーク管理ツールがデバイスを構成するために使用するネットワーク管理メッセージを受信するためにも使用されます。最も大きなネットワーク管理メッセージに対応するには、22 バイト（明示的アドレス指定を行う場合には 34 バイト）の最小入力アプリケーション・バッファ・サイズが必要です。

ネットワーク・バッファ・サイズ

ネットワーク・バッファに必要なサイズは、次の式で求めた値と同じかそれより小さくなります。

message_size + システム・オーバーヘッド（6 バイト） + プロトコル・オーバーヘッド（20 バイト）

プロトコル・オーバーヘッドは、メッセージ当たり 5〜20 バイトです。この計算式は最大値で計算しています。40 バイトのメッセージには、少なくとも 66 バイトのネットワーク・バッファが必要になります（表 8.1 を参照してください）。

ネットワーク・バッファは、ネットワーク管理ツールがデバイスを構成するためのネットワーク管理メッセージを受信し、それに応答するためにも使用されます。最も大きなネットワーク管理メッセージに対応するには、42 バイトの最小入力ネットワーク・バッファ・サイズと 50 バイトの出力ネットワーク・バッファ・サイズが必要です。

エラー

入力メッセージが着信用ネットワーク・バッファに収まったとしても、着信用アプリケーション・バッファに収まらなければ、そのメッセージは廃棄されます。この場合にはファームウエアによって APP_BUF_TOO_SMALL のエラー・コードがログに記録されます。確認応答付きサービスの入力メッセージに対する確認応答は送信されませんし、リクエスト・サービスのメッセージへのレスポンスも送信されません。

出力メッセージが発信用アプリケーション・バッファに収まっても発信用ネットワーク・バッファに収まらなければ、NET_BUF_TOO_SMALL エラーが記録され、そのデバイスはリセットされます。

バッファ数

ほとんどの場合、出力アプリケーション・バッファの数はデフォルトの値で十分です。Sync 型ネットワーク変数出力を使用している場合、出力側のアプリケーション・バッファ数を増やすと、先取りモードに入る確率が低くなります（第 3 章「ネットワーク変数を使ったデバイス間通信」の「先取りモード」を参照してください）。

入力ネットワーク・バッファとして必要な数は、使用しているサービス・タイプとデバイス間のコネクション・タイプによって異なります。認証機能を使用している場合は、メッセージの数が倍増するため、ネットワーク・バッファ数が足りなくなることもあります。デバイスが「ユニキャスト」コネクション（1 つのデバイスが 1 つのデバイスにネットワーク変数やネットワーク・メッセージを送信する）を使用する場合、ネットワーク・バッファの数は、デフォルトの値で十分でしょう。「マルチキャスト」確認応答付きサービスまたはマルチキャストリクエスト/レスポンス・サービス（1 つのデバイスが複数のデバイスに対してメッセージを送信し、各デバイスから応答を受信する）を使用する場合、ネットワーク入力バッファ数は少なくとも送信先グループの最大デバイス数にしておきます。例えば、あるデバイスが異なる 63 個のデバイスに対して確認応答付きサービスやリクエスト/レスポンス・サービスのメッセージを送信する場合、送信デバイスはほとんど同時に 63 個の確認応答やレスポンスを受信する可能性があります。一般に大規模な確認応答付きサービスによるメッセージ配信はあまり使用しないようにします。代わりに反復サービスによるメッセージを送信することで、ネットワーク・トラフィックに負担をかけずに同程度のメッセージ配信の信頼性を達成できます。反復サービスを使用した場合、ネットワーク変数の更新時も確認応答は生成されないため、入力バッファは不要になります。

必要になるネットワーク入力バッファ数の正確な値は、ビットレートや入力クロックによって異なりますので、効率のよい最小限のバッファ数を決めるためには何回か試験を実施する必要があります。

バッファ割り当てのためのコンパイラ指令

このセクションでは、各バッファのサイズや数を設定するためのコンパイラ指令について説明します。

コンパイラ指令を使ってバッファ・サイズを設定したとき、そのサイズ設定がネットワーク管理メッセージの受信や応答が利用できないほど小さいと、コンパイラは警告メッセージを表示します。

発信用アプリケーション・バッファ

以下のコンパイラ指令では、発信するメッセージおよびネットワーク変数がアプリケーション・プロセッサとネットワーク・プロセッサの間で使用する優先/非優先バッファの数とサイズを設定します。デフォルトと指定できる値については、表 8.1 を参照してください。

#pragma app_buf_out_size *n*

発信する優先/非優先のアプリケーション・メッセージおよびネットワーク変数が使用するアプリケーション・バッファのサイズをバイト単位で設定します。

#pragma app_buf_out_count *n*

発信する非優先アプリケーション・メッセージおよび非優先ネットワーク変数が使用できるアプリケーション・バッファの数を設定します。

#pragma app_buf_out_priority_count *n*

発信する優先アプリケーション・メッセージおよび優先ネットワーク変数が使用できるアプリケーション・バッファの数を設定します。

発信用ネットワーク・バッファ

以下のコンパイラ指令は、発信するアプリケーション・メッセージやネットワーク変数がネットワーク・プロセッサと MAC プロセッサの間で使用する優先/非優先バッファの数とサイズを設定します。デフォルトと指定できる値については、表 8.1 を参照してください。

#pragma net_buf_out_size *n*

発信する優先/非優先のアプリケーション・メッセージおよびネットワーク変数が使用するネットワーク・バッファのサイズをバイト単位で設定します。このサイズは 42 バイト以上にしてください。これより小さいサイズなっていると、デバイスがネットワーク管理ツールからのネットワーク管理メッセージに正常に応答できなくなる可能性があります。

#pragma net_buf_out_count *n*

発信する非優先メッセージおよび非優先ネットワーク変数が使用できるネットワーク・バッファの数を設定します。

#pragma net_buf_out_priority_count *n*

発信する優先メッセージおよび優先ネットワーク変数が使用できるネットワーク・バッファの数を設定します。

着信用ネットワーク・バッファ

以下のコンパイラ指令は、到着したアプリケーション・メッセージおよびネットワーク変数が MAC プロセッサとネットワーク・プロセッサの間で使用するバッファの数とサイズを設定します。デフォルト値と指定できる値については、表 8.1 を参照してください。

#pragma net_buf_in_size *n*

到着したアプリケーション・メッセージおよびネットワーク変数が使用できるネットワーク・バッファの大きさをバイト単位で設定します。ネットワーク管理ツールからのネットワーク管理メッセージを受信するには、少なくとも 50 バイト必要です。

#pragma net_buf_in_count *n*

到着したアプリケーション・メッセージおよびネットワーク変数が使用できるネットワーク・バッファの数を設定します。

着信用アプリケーション・バッファ

以下のコンパイラ指令は、着信したアプリケーション・メッセージおよびネットワーク変数がネットワーク・プロセッサとアプリケーション・プロセッサの間で使用するバッファの数とサイズを設定します。デフォルト値と指定できる値については、表 8.1 を参照してください。

#pragma app_buf_in_size *n*

着信したアプリケーション・メッセージおよびネットワーク変数が使用できるアプリケーション・バッファの大きさをバイト単位で設定します。このサイズは 22 バイト以上（明示的アドレス指定を使用している場合は 34 バイト以上）にしてください。

#pragma app_buf_in_count *n*

着信したアプリケーション・メッセージおよびネットワーク変数が使用できるアプリケーション・バッファの数を設定します。

受信トランザクション数

ネットワーク・プロセッサによって同時に処理される着信トランザクションの数は、受信トランザクション配列によって決まります。配列のエントリ数は、次のコンパイラ指令によって設定します。受信トランザクションのエントリのサイズは 13 バイトです。デフォルトと指定できる値については、表 8.1 を参照してください

#pragma receive_trans_count *n*

受信トランザクション配列のエントリ数を設定します。受信トランザクション・ブロックのサイズは 13 バイトです。

確認応答なしの反復サービスまたは確認応答付きサービスやリクエスト/レスポンス・サービスを使用する着信メッセージには、それぞれ受信トランザクション・エントリが必要です。確認応答なしサービスの場合、受信トランザクション・エントリは必要ありません。また、受信トランザクション・エントリは、送信元アドレス/送信先アドレス/優先属性が違うものに対してそれぞれ割り当てる必要があります。各受信トランザクション・エントリには、現在のトランザクション番号が含まれています。送信元アドレス/送信先アドレス/優先属性が既存の受信トランザクションの値と同じで、メッセージのトランザクション番号とエントリのトランザクション番号が一致している場合、重複メッセージであると判断されます。

受信タイマーがタイムアウトになると、受信トランザクション・エントリは解放されます。受信タイマーの時間長は送信先デバイスによって決まり、そのメッセージのアドレス指定モードによって異なります。グループ・アドレス指定メッセージでは、受信タイマーがアドレス・テーブルに登録されています。Neuron ID アドレス指定メッセージの受信タイマーは固定で、8 秒間にセットされています。その他のアドレス指定モードの場合は、構成データ構造体内の非グループ（non-group）受信タイマーが使用されます。

表 8.1　バッファ数とバッファ・サイズ(1/3)		
プラグマ	**指定できる値**	**デフォルト**
app_buf_out_size	20, 21, 22, 24, 26, 30, 34, 42, 50, 66, 82, 114, 146, 210, 255 バイト	A
app_buf_out_count	1, 2, 3, 5, 7, 11, 15, 23, 31, 47, 63, 95, 127, 191	E
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 バイト （奨励される最小値は 42 バイト以上です）	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 バイト （奨励される最小値は 50 バイト以上です）	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 バイト （明示的アドレス指定を使用する場合は、34 バイト以上の最小値、それ以外の場合は 22 バイト以上の最小値が奨励されます）	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

表 8.1　バッファ数とバッファ・サイズ(2/3)

A. app_buf_out_size のデフォルト値

発信メッセージが **msg_send()**で送信されるとき

明示的アドレス指定を使用する場合
A = 66

明示的アドレス指定を使用しない場合
A = 50

発信する明示的メッセージが送信されないとき

ネットワーク変数に対して明示的アドレス指定を使用する場合
A = max(34, 19+sizeof(最も大きな出力ネットワーク変数))

明示的アドレス指定を使用しない場合
A = max(20, 8+sizeof(最も大きな出力ネットワーク変数))

B. net_buf_out_size のデフォルト値

発信メッセージが **msg_send()**または **resp_send()**で送信される場合
B = 66

そうでない場合
B = max(42, 22+sizeof(最も大きなネットワーク変数))

注意：　応答はアプリケーションによってアプリケーション入力バッファ内に作成されますが、ネットワーク・プロセッサはネットワーク出力バッファを使用して応答パケットを作成します。このため、ネットワーク出力バッファは他の発信メッセージだけでなく、発信応答にも対応できる大きさにする必要があります。

C. app_buf_in_size のデフォルト値

明示的メッセージ関数やイベントを使用するとき（発信、着信とも）

明示的アドレス指定を使用する場合
C = 66

明示的アドレス指定を使用しない場合
C = 50

明示的メッセージの関数やイベントを使用しないとき

ネットワーク変数に対して明示的アドレス指定を使用する場合
C = max(34, 19+sizeof(最も大きなネットワーク変数))

明示的アドレス指定を使用しない場合
C = max(22, 8+sizeof(最も大きなネットワーク変数))

表 8.1　バッファ数とバッファ・サイズ(3/3)

D.　receive_trans_count のデフォルト値

アプリケーション・プログラムが明示的メッセージを受信する場合

D = max(8, min(16, 非構成入力ネットワーク変数の数 + 2))

アプリケーション・プログラムが明示的メッセージを受信しない場合

D = min(16, 非構成入力ネットワーク変数の数 + 2)

E.　app_buf_out_count, app_buf_out_priority_count, net_buf_out_count, net_buf_out_priority_count のデフォルト値

アプリケーションが Neuron 3120 チップまたは Neuron 3120E1 チップにリンクされる場合

E = 1

アプリケーションがその他の Neuron チップまたはスマート・トランシーバにリンクされる場合

E = 2

注意：　優先バッファ数をゼロに設定すると、すべてのネットワーク変数は非優先として扱われます。**app_buf_out_priority_count** または **net_buf_out_priority_count** がゼロでない場合は、もう一方もゼロであってはいけませんし、この 2 つの発信トランザクション・バッファは自動的に RAM の中に割り当てられます。優先出力バッファがない場合には、発信トランザクション・バッファが 1 つだけ割り当てられます。発信トランザクション・バッファのサイズは、Neuron ファームウエアのバージョン 4 と 6 以降では 28 バイトで、それ以前のバージョンでは 18 バイトです。

注意：　Neuron C コンパイラが次のいずれかを参照している場合は、プログラムで明示的アドレス指定が使用されているものと判断されます。

msg_in.addr
resp_in.addr
msg_out.dest_addr
nv_in_addr

Neuron チップ・メモリの使用

このセクションでは、オフチップ（Neuron チップの外部につけた）・メモリと共に Neuron チップまたはスマート・トランシーバを使用している場合と、オフチップ・メモリを使用せずに Neuron チップまたはスマート・トランシーバを使用している場合の 2 つの状況について説明します。

オフチップ・メモリを使用したチップ

Neuron 3150 チップと FT 3150 スマート・トランシーバのオンチップ・メモリは、RAM と EEPROM で構成されています。

これらのチップのオフチップ・メモリには、ROM、RAM、EEPROM、NVRAM、フラッシュ・メモリの 1 つまたは複数を使用することができます。デバイスを定義するときは、各領域の開始ページ番号とページの数を指定します（1 ページは 256 バイト）。ROM を使用する場合、その開始番地は 0000 に固定されています。ROM を使用しない場合は、この領域をフラッシュ・メモリまたは NVRAM メモリが使用し、開始番地は 0000 になります。メモリ中の各領域の順番は、図 8.3 のとおりです。これらの領域は連続である必要はありませんが、重ねることはできません。

メモリ・マップド I/O デバイスは Neuron 3150 チップおよび FT 3150 スマート・トランシーバに接続することができます。これらのデバイスは、図 8.3 で黒く塗られた部分に該当するメモリ・アドレス内ならどこでも入出力を行えます。

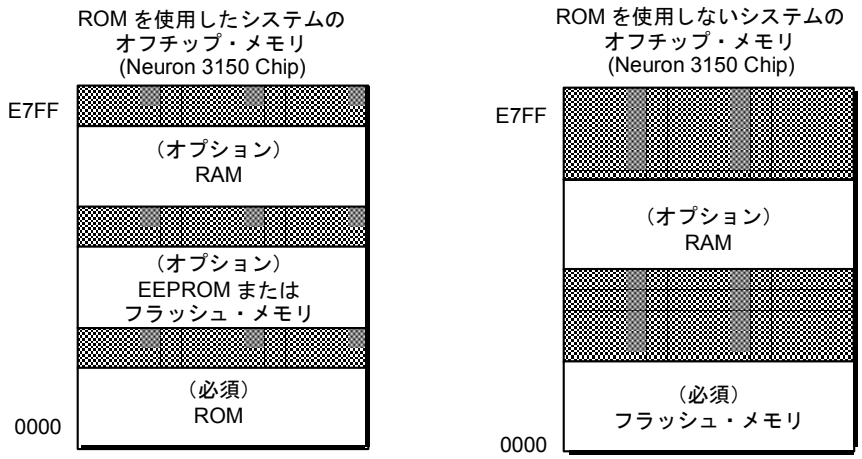


図 8.3 Neuron 3150 チップと
FT 3150 スマート・トランシーバのオフチップ・メモリ

オフチップ・メモリを使用しないチップ

Neuron 3120 チップと FT 3120 スマート・トランシーバのオンチップ・メモリは、ROM、RAM、および EEPROM で構成されています。これらのデバイスは、オフチップ・メモリをサポートしていません。図 8.4 にメモリ・マップ図を示します。

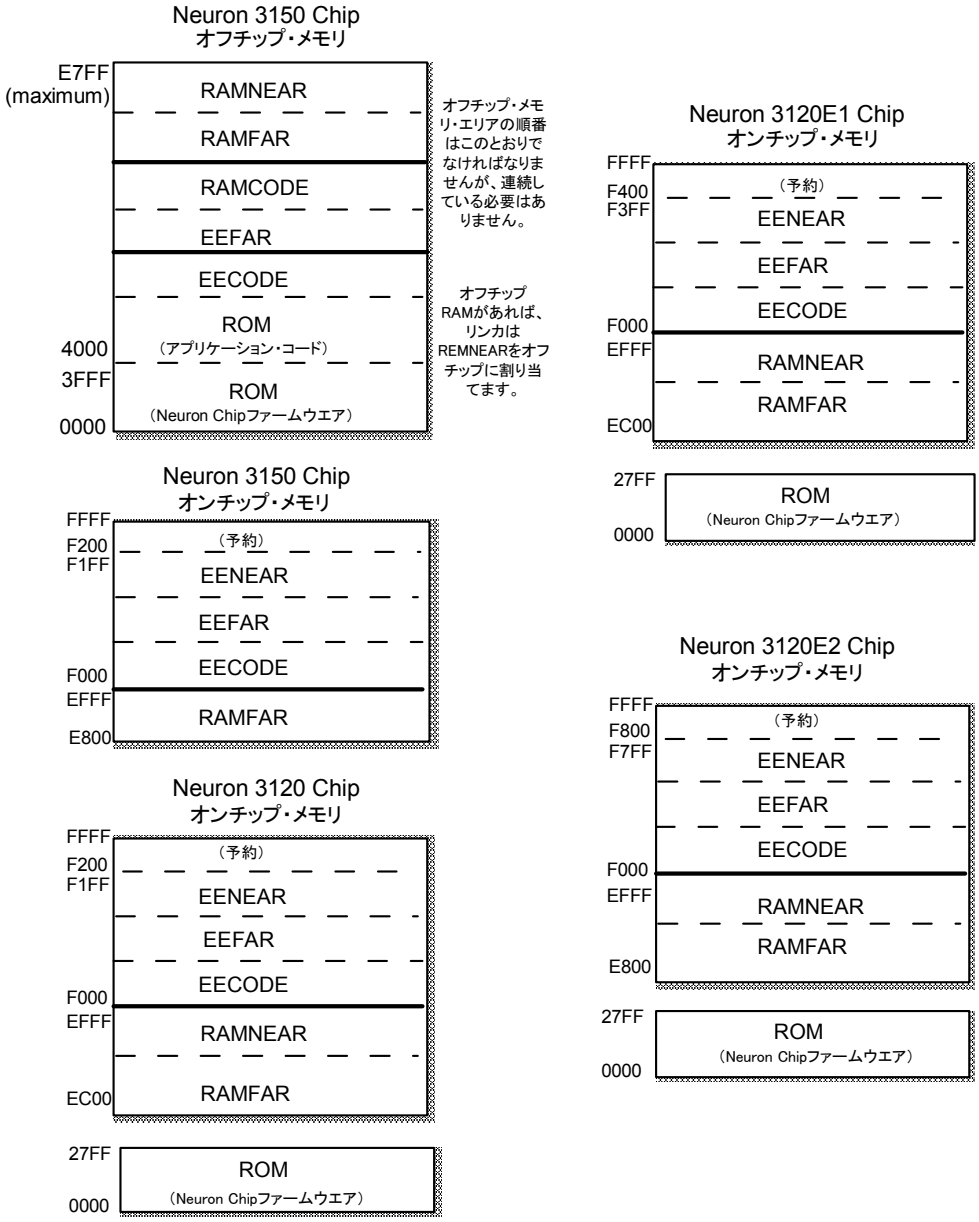


図 8.4　リンカによって定義されている領域を示す、
さまざまなチップのメモリ・マップ

メモリ領域

Neuron チップには、次の 3 種類のメモリ領域があります。

- ROM : デバイス上のプログラムを実行する前に初期化される不揮発性メモリ。ROM の内容はプログラムから変更できません。ROM は Neuron ファームウエアが使用し、アプリケーション・コードと定数を書き込むこともできます（Neuron 3150 チップまたは FT 3150 スマート・トランシーバのみ）。

  オフチップ ROM（Neuron 3150 チップまたは FT 3150 スマート・トランシーバのみ）には、ROM、PROM、EEPROM、フラッシュ・メモリ、または不揮発性 RAM など、どのような種類の不揮発性メモリを使っても構いません。オフチップ ROM またはそれに代わって使用するメモリに Neuron ファームウエアを格納するためには、その開始番地を 0x0000 とし、書き込み遅延を 0（ゼロ）ms にしなければなりません。この条件によって、EEPROM は Neuron ファームウエアの格納用に使用できません。アプリケーション・コードやデータの格納にオフチップ EEPROM を使用することは可能です。

- EEPROM：プログラム実行中に内容を変更できる不揮発性メモリ。オンチップ EEPROM へのメモリ書き込みに必要な時間は、通常、1 バイト当たり 20 ミリ秒です。EEPROM には、アプリケーション・コード、定数および EEPROM 変数を書き込むことができます。

  オフチップ EEPROM には、EEPROM、フラッシュ・メモリ、または不揮発性 RAM を使用できます。フラッシュ・メモリを使用する場合は、LonBuilder ツールのメモリ・マップや NodeBuilder ツールのハードウエア・デバイス・テンプレートをフラッシュ・メモリ用に構成してください。この領域へのメモリ書き込みがあると、Neuron ファームウエアはメモリの書き込みが完了できるように遅延を発生させます。EEPROM で実装する場合、オフチップ EEPROM への書き込みに対する遅延時間は、LonBuilder Hardware Properties ウィンドウまたは NodeBuilder Hardware Template Properties ダイアログを使って 0～255 ミリ秒に設定します。オフチップ・フラッシュ・メモリへの書き込みに対する遅延時間は、64 または 128 バイトのセクタにつき 10 ミリ秒に固定されます。

  EEPROM 領域をフラッシュ・メモリで使用した場合、フラッシュ・メモリが ROM 領域の代わりになることがあります。この場合、フラッシュ・メモリ内のシステム・エリア（本章の「メモリ・エリア」を参照）には書き込めませんが、ユーザ・エリアに書き込むことはできます。フラッシュ・メモリのサポートについては、後述の「フラッシュ・メモリの使用」を参照してください。

  Neuron チップまたはスマート・トランシーバをリセットしても、EEPROM はクリアされません。

- RAM：プログラム実行中に変更可能な揮発性メモリ。RAM には、アプリケーション・コード、定数、または変数を書き込むことができます。

  Neuron ハードウエアでは、低速デバイスに対するウエイト状態は実装されていません。このメモリは、設定された入力クロックレートでの 1 マシンサイクル内に読み書き可能でなければなりません。

  オフチップ RAM 領域はコードが使用できます。コード用のオフチップ RAM 部分は、リセットしてもクリアされません。コードが使用しない方の RAM 領域は、チップのリセット時にゼロに初期化されます。

メモリ・エリア

Neuron ファームウエアと Neuron リンカは、メモリ領域を次のようなメモリ・エリアに分割して使用します。

- ROM 領域には、システム・エリアとユーザ・エリア（Neuron 3150 チップと FT 3150 スマート・トランシーバのみ）があります。システム・エリアは、Neuron 3150 チップで 16K バイト（またはそれ以上）です。ユーザ・エリアも ROM と呼ばれます。フラッシュ・メモリを使用していなければ、Neuron C コンパイラとリンカは、実行コードと定数データをユーザ・エリアに置きます。フラッシュ・メモリを ROM として使用している場合、ユーザ・コードは EECODE エリアに置かれます。
- EEPROM 領域には、次の 3 つのエリアがあります。

  EECODE
  EEFAR
  EENEAR

  オンチップ EEPROM とオフチップ EEPROM の両方があるときは、それぞれが EECODE と EEFAR のセクションを独自に持ちます。EENEAR のセクションは 1 つだけで、オンチップ上にあります。これらはどれもユーザ・エリアです。

  EECODE には、実行コードと定数データがあります。あるオブジェクトを強制的に EEPROM に置くようにコンパイラに指示するには、Neuron C の **eeprom** キーワードを使用します。

  EEFAR エリアには、**config** や **eeprom** キーワードと一緒に **far** キーワードを指定して宣言された変数が置かれます。このエリアには、**config_prop**（または **cp**）キーワードを使用して宣言された構成プロパティ・ネットワーク変数、および **cp_family** キーワードを使用して宣言された構成プロパティの変更可能な構成プロパティ・ファイルも含まれます。

  Neuron C では **offchip** キーワードや **onchip** キーワードを使用すると、特定のオブジェクトをそれぞれ強制的にオフチップまたはオンチップの EEFAR エリアに置くように、コンパイラとリンカに指示できます。

  EENEAR エリアには、**config** または **eeprom** キーワードで宣言された変数が含まれます。総サイズはデフォルトで 255 バイトに制限されています。
- RAM 領域には、次の 3 つのエリアがあります。

  RAMCODE

  RAMFAR

  RAMNEAR

  RAMCODE はオフチップのみに配置できます（Neuron 3150 チップまたは FT 3150 スマート・トランシーバのみ）。このエリアには、実行コードと定数データが含まれます。宣言文で **ram** キーワードを使用すれば、明示的に実行コードと定数データをこのエリアに置くことができます。

  RAMFAR はオンチップにあることも、オフチップにあることもあります。オンチップ RAM には、1 つか 2 つの RAMFAR のセクションがあります。オフチップ RAM があれば、オンチップ RAMFAR エリアは 1 つだけです。RAMFAR エリアは変数が使用します。

  Neuron C で **offchip** キーワードおよび **onchip** キーワードを使用すると、特定のオブジェクトをそれぞれ強制的にオフチップまたはオンチップの RAMFAR エリアに置くように、コンパイラとリンカに指示できます。

RAMNEAR エリアは 1 つしかありません。このエリアは、オンチップ（すべてのチップ）またはオフチップ（Neuron 3150 チップおよび FT 3150 スマート・トランシーバのみ）にあります。RAMNEAR エリアの場所は、リンカによって自動的に判断されます。RAMNEAR エリアは、すべての Neuron C 変数がデフォルトで使用する領域です。このエリアの総サイズは 256 バイトに制限されています。ただし、ユーザがバッファに割り当てたメモリ量によっては、使用可能な最大サイズが 256 バイト以下になることもあります。本章の「バッファ割り当てのためのコンパイラ指令」と「メモリ使用法を変更する特殊キーワード」を参照してください。

デフォルトのメモリ使用法

Neuron C プログラムの中で、変数や関数の宣言などに特殊キーワードが指定されていなければ、リンカは次の規則に従ってプログラムの各部分をメモリ内に配置します。

全ての実行コード・オブジェクト（関数、**when** 節、タスク）は、文字列定数および **const** として宣言されたデータと一緒に ROM または EECODE エリアに置きます。リンカは、これらのオブジェクトを収められる場所ならばどこにでも配置します。Neuron 3150 チップまたは FT 3150 スマート・トランシーバでは、リンカは最初にそのオブジェクトをオフチップ ROM のユーザ・エリアに置いてみます。オブジェクトが ROM に入らなければ、メモリのオフチップ EECODE エリアへの配置を試みます。最後に、メモリのオンチップ EECODE エリアへの配置を試みます。

config、**config_prop**、**cp**、**eeprom** キーワードを使って宣言されたデータ・オブジェクトは、通常、メモリのオンチップ EENEAR エリアに配置されます。これらのキーワードを持たないその他のデータ・オブジェクトは通常メモリの RAMNEAR エリアに配置されます。

cp_family キーワードを使用して宣言したデータ・オブジェクト（構成プロパティ・ファミリ）は、メモリに格納される複数の項目を作成します。各構成プロパティ・ファミリのメンバのインスタンスは、テンプレート・ファイルに記述子のエントリを作成し、書き込み可能の値ファイルまたは読み取り専用の値ファイルにデータ値のエントリを作成します。構成テンプレート・ファイルは ROM エリアまたは EECODE エリアに配置されます。書き込み可能な値ファイルはメモリの EEFAR エリアに配置されます。読み取り専用の値ファイルはメモリの ROM エリアまたは EECODE エリアに配置されます。

次のセクションで説明する特殊キーワードを使用すると、リンカによる変数や関数の配置を変更できます。

メモリ使用法を変更する特殊キーワード

リンク時にプログラムの一部がデフォルトのメモリ領域に入らないというエラー・メッセージが表示された場合には、Neuron C の特殊キーワードを使った変数や関数の宣言を行います。これらのキーワードは、変数や関数をメモリ内の他の領域に移動するための指定です。ここでは、**eeprom**、**far**、**offchip**、**ram**、**uninit** といった特殊キーワードについて説明します。

EENEAR および RAMNEAR エリアでの直接的なメモリの読み書き（アプリケーションが行うもの）には Neuron チップやスマート・トランシーバの特殊なアドレッシング・モードが使用されるので、効率的なコードを生成できます（1 命令当たりのバイト数が減少し、読み書きのサイクルが少なくなりま

す）。ただし、ポインタを使ったメモリへの間接アクセスは、NEAR エリアでも FAR エリアでも同じです。

eeprom キーワード（関数とデータの宣言用）

Neuron 3150 チップおよび FT3150 スマート・トランシーバでは、関数および **const** 型データはデフォルトにより ROM に配置されます。ROM が一杯になったときや使用できる ROM がない場合、残りの関数や **const** 型データは、最初にオフチップ EECODE、次にオンチップ EECODE の順で EECODE エリアに配置されます。ただし、関数の定義や **const** 型データの宣言に **eeprom** キーワードを指定すれば、これらをメモリの ROM エリアから EECODE エリアに明示的に移すように指示できます。以下にコード例を示します。

```
eeprom int fn() { ... statements ... }

eeprom const type varname = {inits};
```

eeprom キーワードは、インストール後にネットワーク管理ツールによって変更される可能性があまりない関数に使用すると便利です。例えば、キャリブレーションなどの構成が必要な **const** 型データ構造体は、EEPROM に格納することで、ネットワーク管理ツールを使用して変更できるようになります。

このキーワードを指定すると、RAMNEAR ではなく EENEAR に変数が配置されるため、電源が切れても値が保存されます。ただし、**eeprom** 変数には変更に関する制限があります。各 EEPROM がサポートしている書き込み回数の制限については、『Smart Transceivers Databook』を参照してください。

前述のように、宣言に **eeprom** キーワードを指定すれば、変数をメモリの RAMNEAR エリアから EENEAR エリアに移すように指示できます。例えば、次の宣言では *varname* が EENEAR エリアに移動されます。

```
eeprom int varname;
```

eeprom または **config** キーワードのどちらかを使えば、ネットワーク変数を EENEAR エリアに置くことができます。上の例と同様に、**far** キーワードもネットワーク変数に使用できます。

EENEAR エリアのサイズは最大 255 バイトに制限されています（ただし他の要因によってこのエリアがさらに制限される場合もあります）。追加のオンチップ EEPROM およびすべてのオフチップ EEPROM は、EEFAR エリアとして扱われます。変数を指定して EEFAR エリアに移す方法については、以下の **far**、**offchip**、**onchip** の各キーワードの説明を参照してください。

eeprom クラス変数が初期化されるのは、LonMaker 統合ツールやその他のネットワーク管理ツールなどの外部システムからアプリケーション・イメージがロードされたときです。また、プログラムの再ロードのときにも、すべての **eeprom** 変数が初期化されます。デバイスの再スタートや電源投入のときには、**eeprom** 変数は初期化されません。ただし、この規則にも例外があり、これについては後述の **uninit** キーワードの説明を参照してください。

オンチップ EEPROM に値を書き込むと、通常はその値が有効になるまでに約 20 ミリ秒かかります（ただし、この時間はチップによって異なります）。この書き込み時間が足りないと、値が書き込まれないか、書き込まれたとしても不揮発ではなくなる可能性があります（停電によって電源が切られたとき、デバイスが外部からリセットされたとき、またはウォッチドッグ・タイマー

のタイムアウトが起こってデバイスがリセットされたときなどに、書き込み時間が足りなくなることがあります）。

far キーワード（データの宣言用）

データ・オブジェクトがメモリの RAMNEAR エリアに入らないとき、次のようなリンカ・エラー・メッセージが表示されます。

```
Error:  No more memory in RAMNEAR area
Error:  Could not relocate segment in file '<program>.no'
```

Neuron C のデータ宣言に **far** キーワードを加えれば、リンカはそのオブジェクトをメモリの RAMFAR エリアに置きます。例えば、次の宣言では *varname* が RAMFAR エリアに移動します。

```
far int varname;
```

同様に、**config** または **eeprom** オブジェクトがメモリの EENEAR エリアに入らないとき、次のようなメッセージが表示されます。

```
Error:  No more memory in EENEAR area
Error:  Could not relocate segment in file '<program>.no'
```

Neuron C のデータ宣言に **far** キーワードを加えれば、リンカはそのオブジェクトをメモリの EEFAR エリアに置きます。例えば、次の宣言では *varname* が EEFAR エリアに移動します。

```
far eeprom int varname;
```

この種の宣言を使用すると、例えば大きすぎて EENEAR エリアに収まらないデータ・テーブルをメモリの EEFAR エリアに移動できます。

一般的なガイドラインとして、頻繁にアクセスするデータはできるだけ NEAR エリアに残すようにします。NEAR エリアを使用する方が、FAR エリアを使用するよりも実行サイクルが少ないため、生成する命令が少なくてすみます。非定数インデックスやポインタを使って参照される配列は、効率を犠牲にすることなくメモリの FAR エリアに配置することができます。

offchip キーワード（関数とデータの宣言用）

Neuron リンカがコード、**const** 型データ、**far** 変数を配置するとき、通常はオフチップ・エリアに配置し、必要な場合にだけオンチップ・エリアに配置します。ただしフラッシュ・メモリ用にリンクするときには、リンカのデフォルトの動作が異なります。詳しくは、後述の「フラッシュ・メモリの使用」を参照してください。どのデータ宣言または関数宣言でも、**offchip** キーワードを使用すれば、これらのオブジェクトの配置を明示的に制御できます。

適切なオフチップ・メモリ・エリアが使用可能であれば、オブジェクトはそのエリアに配置されます。このメモリ・エリアが使用可能でない場合には、リンカは処理を中止してその内容を示すエラー・メッセージを表示します。**offchip** キーワードの使用例を以下に示します。

```
far offchip int a;            // offchip RAMFAR

far eeprom offchip int b;     // offchip EEFAR
```

```
const eeprom offchip int c = init;
      // offchip EECODE (no need for far kwd)
eeprom offchip void fn () {...}
      // offchip EECODE
```

onchip キーワード（関数とデータの宣言用）

Neuron リンカがコード、**const** 型データ、**far** 変数を配置するとき、通常はオフチップ・エリアに配置し、必要な場合にだけオンチップ・エリアに配置するようになっています。どのデータ宣言または関数宣言でも、**onchip** キーワードを使用すれば、これらのオブジェクトの配置を明示的に制御できます。適切なオンチップ・メモリ・エリアが使用可能であれば、オブジェクトはそのエリアに配置されます。このメモリ・エリアが使用可能でない場合は、リンカは処理を中止してその内容を示すエラー・メッセージを表示します。詳しくは、後述の「フラッシュ・メモリの使用」を参照してください。**onchip** キーワードの使用例を以下に示します。

```
far onchip int a;              // onchip RAMFAR
far eeprom onchip int b;       // onchip EEFAR
const eeprom onchip int c = init;
      // onchip EECODE (no need for far kwd)
eeprom onchip void fn () {...}
      // onchip EECODE - would be in EEPROM
      // even without the eeprom keyword
```

オフチップ・フラッシュ・メモリを使用しているとき、**onchip** キーワードを使用すると、データをオンチップ EEPROM へ移すことができます。オンチップ EEPROM は、オフチップ・フラッシュ・メモリよりも書き込み可能サイクル数が多いので、頻繁に更新される EEPROM 変数はオンチップ・エリアに配置するようにします。最大書き込み数については、オフチップ・フラッシュ・メモリのメーカーのデータブックおよび『Smart Transceivers Databook』を参照してください。

ram キーワード（関数用）

const 型データと同様に、関数およびその他の実行コードも、Neuron 3150 チップまたは FT 3150 スマート・トランシーバ上で ROM が使用可能であればすべて ROM に配置され、次にオフチップまたはオンチップ EECODE に配置されるようになっています。Neuron C の関数定義に **ram** キーワードを指定すると、関数をメモリのオフチップ RAMCODE エリアに移すことができます。この RAMCODE エリアは、Neuron 3150 チップまたは FT 3150 スマート・トランシーバに接続されたオフチップ RAM メモリでのみ使用できます。ただし、電源が切れてもデバイスの状態を保護しておきたければ、RAM は不揮発性でなければなりません（バッテリ・バックアップ機能付き RAM など）。Neuron チップのリセット時に RAM の内容を保護するための注意事項については、『Neuron 3150 Chip External Memory Interface』技術資料を参照してください。

ram キーワードは関数名の前であればどこに指定しても構いません。例えば、次のように指定します。

```
ram int fn() { ... statements ... }
```

ram キーワードは、インストール後にネットワーク管理ツールが頻繁に変更する可能性がある関数に使用してください。

uninit キーワード（データ宣言用）

uninit キーワードは、**eeprom** 変数宣言と一緒に使用できます。このキーワードを指定すると、EENEAR や EEFAR メモリ・エリアのデータが、プログラムのロードやチップのリセットによる影響を受けなくなります。このキーワードの使用には 2 つの利点があります。データベースやキャリブレーションなどの用途には、広範囲のメモリ・エリアを割り当てる必要が生じることがありますが、このようなとき、プログラムのロードやチップのリセットが起こっても内容が書き変わらずに残っていると便利です。さらに、EEPROM の **uninit** エリアはロードされないので、ロード時間が短縮できます。例えば、**uninit** キーワードを指定してこのようなメモリ・エリアを 500 バイト確保するには、次のように指定します

```
uninit eeprom int datablock[500];
```

プログラムの再リンク

コンパイラは、アプリケーション・コードをメモリの適切なエリアに置くように指示します。リンカはデータをメモリに配置し、グローバル・シンボリック・アドレス参照を提供します。このようなメモリの割り当ては、コンパイル時の宣言の順番で行われます。このため、リンクからリンクまで同じアドレスを保持するには、同じ順序で宣言を行うようにしてください。

フラッシュ・メモリの使用

Neuron 3150 チップおよび FT 3150 スマート・トランシーバの Neuron ファームウエア・バージョン 6 以降では、フラッシュ・メモリを使用できます。Neuron ファームウエアは、Atmel AT29C256 および AT29C257（32Kx8、64 バイト・セクタサイズ）、AT29C512（64Kx8、128 バイト・セクタサイズ）、AT29C010（128Kx8、128 バイト・セクタサイズ）のフラッシュ・メモリのみをサポートしています。フラッシュ・メモリを使用するための指定は、LonBuilder ツールのメモリ・マップまたは NodeBuilder ツールのハードウエア・テンプレートを使って行います。これらの機能やフラッシュ・メモリを指定する方法については、『LonBuilder User's Guide』と『NodeBuilder User's Guide』を参照してください。Neuron 3150 チップおよび FT 3150 スマート・トランシーバでフラッシュ・メモリを使用する場合、クロック・スピードは 1.25MHz 以上でなければなりません。

フラッシュ・メモリは、EEPROM メモリ領域としてだけ使用することも、ROM と EEPROM の両方のメモリ領域として使用することもできます。EEPROM メモリ領域としてフラッシュ・メモリを使用する場合は、EEPROM と通常関連付けられているすべてのメモリ・エリア（EECODE と EEFAR）を含めることが可能です。ただし、フラッシュ・メモリには書き込み回数に制限があることに注意してください（通常は 1000 回までですが、チップによって異なります。詳細については、チップのメーカーのデータブックを参照してください）。このため、コンパイラ、リンカ、ファームウエアなどがフラッシュ・メモリでの **eeprom** クラス変数をサポートしていても、プログラム作成時には、これらの変数をできる限り変更しないようにして、フラッシュ・メモリに配置される変数の変更回数が制限数を超えないようにしなければなりません。

フラッシュ・メモリを Neuron ファームウエア ROM として使用する場合、ROMCODE エリア・サイズはシステム・イメージのサイズ（16K バイト以上）に固定されるので、この中にアプリケーション・コードやデータを含めることはできません。残りのフラッシュ・メモリ・アドレスの部分は、EECODE と EEFAR とに分割されます。ただし、フラッシュ・メモリにはシステム・イメージとアプリケーションの読み/書きデータの両方を配置しないようにします。フラッシュ・メモリに書き込みを行うと、チップのプログラミング間隔（＜10 ミリ秒）の間はシステム・イメージを読み出すことができないからです。Neuron ファームウエアではプログラミング間隔中に自動的にファームウエア自身をロックしてフラッシュ・メモリを使用できないようにするため、すべてのシステム関数に遅延が生じ、ロックされている間のネットワーク・メッセージが失われることになります。データをフラッシュ・メモリに配置する必要がある場合は、**offchip** キーワードを使って、最も変更頻度の少ない変数をフラッシュ・メモリに移すようにします。

フラッシュ・メモリはセクタ単位（64 か 128 バイト）で書き込まれるため、フラッシュ・メモリ内の **eeprom** 変数の書き込み可能回数は、その変数が置かれているセクタ全体に対する書き込み回数ということになります。例えば、1 バイトの **eeprom** 変数が書き込み保証サイクル 1000 回のオフチップ EEPROM に置かれているとします。この変数は、1000 回の更新ができるはずです。次に、この変数を同じような書き込み制限のオフチップのフラッシュ・メモリ（64 バイト・セクタ）に置いたとすると、そのセクタ内にあるすべての変数の書き込み総数が 1000 回までという保証になるため、セクタ内の全部のデータが書き込み保証回数に影響します。ほぼ同じ頻度で変更される 1 バイト変数が 64 個あったとすると、各変数の書き込み可能回数は 1,000/64 回で、約 15 回ということになります。

フラッシュ・メモリ内のデータ変更にはフラッシュ・プログラミング・サイクルが必要なため、Neuron C リンカは、なるべくすべてのデータ・オブジェクトをオンチップ・メモリに配置するというリンク・アルゴリズムを使用します。オブジェクトをオフチップ・メモリに配置するには、データ・オブジェクトの宣言に **offchip** キーワードを指定します。リンカがデータをフラッシュ・メモリに配置すると、警告メッセージが表示されます。

アプリケーションからフラッシュ・メモリへ直接書き込みを行うと、フラッシュ・メモリが書き込み保護されている場合でも必ずプログラミング・サイクルとなります。これは、アプリケーションが Neuron ファームウエアをバイパスしてフラッシュ・メモリにアクセスするときに発生するエラー状態です。Neuron ファームウエアでは、フラッシュ・メモリのソフトウエア書き込み保護機能を備えているため、無効な書き込みによってフラッシュ・メモリの内容が書き換えられることはありません。ただし、プログラミング・サイクルは開始されます。プログラミング・サイクルの間、フラッシュ・メモリからは無効なデータが提供されることになるため、ウォッチドッグがリセットされます。このリセットは、書き込みサイクルの間に起こる可能性があり、最悪の場合はリセット状態が無限に続きます。このため、システム・イメージをフラッシュ・メモリに置くときには、少なくとも書き込みサイクル（通常 10 ミリ秒）中は Neuron のリセット状態を延長できるようなハードウエア・メカニズムが必要になります。これを実現しているのが、『Smart Transceivers Databook』で説明している LVI 回路です。詳しくは、『3150 Chip External Memory Interface』技術資料を参照してください。

EEPROM の書き込み中に電源の切断、投入、および何らかのリセットが起こった場合、データの破損はその書き込みが行われていたエリアにのみ集中します。ただしフラッシュ・メモリでは、常に 1 つのセクタ全体が一度にプロ

グラムされるため、障害が発生した際に、書き込まれていたセクタ内の全データが影響を受けている可能性あります。チェックサムなしの重要な読み書きデータは、重複チェック、ボータ、ジャーナルなどの方法を使って、アプリケーション・プログラムでデータを保護するようにしてください。

さらに、フラッシュ・メモリのロードはセクタ単位で行われるため、ロード・イメージのデータが連続して配置されていることが重要になります。つまり、**uninit eeprom** データが初期化された **eeprom** データと混在しているような状態は避けるべきです。リンカは、プログラム内で宣言された順番にデータを処理するので、プログラムする際に **uninit eeprom** の宣言をまとめて、ロード時間を短縮するようにしてください。

eeprom_memcpy()関数

EEPROM メモリはフラッシュ・メモリと同様に、構造体から構造体への代入も含め、直接の代入によって書き込むことができます。この場合コンパイラは、目的の変数が EEPROM メモリ内にあることを認識し、適切なファームウエア関数を使って、適切な遅延、ハードウエア・インターフェース手順を実行して、メモリに正しく書き込むことができます。

これに対してポインタを介して EEPROM メモリに書き込みを行う場合には、コンパイラはそのポインタがどのメモリを指しているのかを追跡していません。そこで、ポインタを使った書き込みが起こらないように、EEPROM 変数のアドレスはコンパイラによって自動的に「**const ***」型に変換されます。

```
eeprom int x;

.... &x ... // '&x' is 'const int *'
```

通常コンパイラは、どのようなポインタからも **const** 属性を取り除くようなことはしません。これは、暗黙的、明示的どちらの型変換の場合にもあてはまります。暗黙的な型変換は、ある値を異なる型の変数に代入したときや、次の例に示すように、関数に渡されたパラメータとその関数の仮引数のデータ型が異なる場合に起こります。

```
eeprom int x;
int *p;
void f (int *p);
p = &x;      // implicit cast, compiler error
f(&x);       // another erroneous implicit cast
p = (int *)&x;    // explicit cast, also error
```

Neuron C コンパイラのこの動作は、ANSI C で規定されている動作よりも厳密です。ただし、プログラム中に**#pragma relaxed_casting_on** 指令が指定されていると、このような暗黙的または明示的な型変換が起こっても、コンパイラからは警告メッセージが表示されるだけになります。さらに、この警告メッセージも表示しないようにするには、**#pragma warnings_off** 指令を使用します。これらの指令の後にそれぞれ対応する **relaxed_casting_off** と **warnings_on** という指令をプログラム内で使用すると、指定した場所以降のコンパイラの動作をデフォルト状態に戻すことができます。

上記の機能を使用することは、EEPROM やフラッシュ・メモリへの書き込みを、コンパイラによる検査を迂回して実行する（ファームウエアを無視して書き込む）ことになるので、危険なことではあります。EEPROM やフラッシュ・メモリを参照している可能性のあるポインタを使っているときのために、**eeprom_memcpy()**という書き込み用の関数が用意されています。この関数の

パラメータは **memcpy()**関数のパラメータと同じです。ただし、**eeprom_memcpy()**関数では、書き込み先アドレスとして EEPROM やフラッシュ・メモリをサポートしていますが、通常の **memcpy()**関数ではこれをサポートしていません。**eeprom_memcpy()**関数のパラメータ長の制限は 255 バイトです。

注意： 長すぎるパラメータを指定してこの関数を古いバージョンのファームウエアで使用すると、ウォッチドッグ・タイマーがタイムアウトになり、デバイスがリセットされてしまう場合があります。10 MHz で動作しているデバイスに使用できる安全な長さは 32 バイトです。

バージョン 7 以降のファームウエアを実行している Neuron 3120xx チップと FT 3120 スマート・トランシーバ、およびバージョン 12 以降のファームウエアを実行している Neuron 3150 チップと FT 3150 スマート・トランシーバでは、**eeprom_memcpy()**の使用中にウォッチドッグ・タイマーがタイムアウトになることはなく、パラメータの長さも 255 バイトまで使用できます。

メモリの使用

このセクションでは、プログラム内の各要素が使用するメモリ量について概説します。プログラムが使用する実メモリについては、リンク・サマリーを参照してください。

RAM の使用

RAM は次のように使用されます。

- コード　コード・サイズ（**ram** キーワードで宣言されている関数）
- **config_prop**　0
- **cp_family**　0
- **fblock**　0
- **io_changes**　3 バイト（任意の型）
- I/O オブジェクト　0
- **msg_tag**　0
- **mtimer**　4 バイト
- 次のサイズはプログラム内で宣言されたグローバル・データと静的データに適用されます（**eeprom** 変数と **config** 変数を除く）。ここでのバイト数は、ネットワーク変数にも当てはまります。

  char　1 バイト

  int　1 バイト

  enum　1 バイト

  long　2 バイト

構造体	各要素のサイズの合計。連続した 8 ビット（またはそれ以下）のビットフィールドには 1 バイトを使用します。ビットフィールドがバイトの境界をまたがることはできません。パディングも実行されません。**float_type** と **s32_type** の拡張演算構造体は、それぞれ 4 バイトを使用します。
共用体	最大の要素サイズ

EEPROM の使用

EEPROM のうちの約 65 バイトは、システム・オーバーヘッドとして使用され、この大きさはファームウエアのバージョンによって異なります。その他の EEPROM またはフラッシュ・メモリ領域は次のように使用されます。

- ドメイン・テーブル・エントリには、ドメイン・アドレス、サブネット番号、デバイス番号、認証キーといった設定可能な情報が含まれており、ドメイン 1 つが EEPROM を 15 バイト使用します。システムは最大 2 個のドメイン・テーブル・エントリを持つことができ、少なくとも 1 つのエントリが必要です。デフォルトのドメイン・テーブル・エントリ数は 2 つです。前述の「ドメイン・テーブル」のセクションを参照してください。
- アドレス・テーブルの各エントリが 5 バイト使用します。アドレス・テーブルのエントリ数は、最大 15 個です。エントリ数の最小値は 0 で、デフォルトは 15 です。前述の「アドレス・テーブル」のセクションを参照してください。
- 各ネットワーク変数宣言（入力または出力）が、設定情報に 3 バイト使用します。さらに、固定情報用に読み取り専用メモリが 3 バイト使用します。SNVT の自己識別（SI）機能を使用している場合、オーバーヘッドの 7 バイトと、ネットワーク変数毎に最低 2 バイトをさらに使用します。
- 各ネットワーク変数別名テーブルは、エントリごとに 4 バイトを使用します。このテーブルにはデフォルトのサイズがありません。前述の「別名テーブル」のセクションを参照してください。
- プログラムの中で **eeprom** および **config** として宣言された変数が、変数のサイズに応じた量の EEPROM を使用します。これには **config_prop**（または **cp**）クラスのネットワーク変数と、**cp_family** キーワードを使用して宣言した変更可能な構成パラメータが含まれます。
- **when** 節テーブルは CODE メモリ・エリア（可能な場合には ROM、もしくは EEPROM）に配置されます。各 **when** 節は 1 つのエントリにつき 3 バイトから 6 バイト（ほとんどは 3 バイト）使用します。このコード・スペースは、**if** 文で生成される同機能のコードよりも一般に若干小さくなります。ユーザ定義イベントを含む **when** 節では、さらにコード・スペースを必要とする可能性があります。
- 読み取り専用の値ファイルは通常 CODE メモリ・エリア（可能な場合には ROM、もしくは EEPROM）に配置されます。構成値ファイルは必要なデータ・バイト数のみを使用し、オーバーヘッドはありません。構成テンプレート・ファイルも CODE メモリ・エリアに配置されます。テンプレート・ファイルは数バイトを使用して、値ファイル内の各構成プロパティについて記述します。このバイト数は、構成プロパティの型と特性によって異なりますが、通常は構成プロパティのインスタンスごとに 12 バイト以上になります。

メモリ・マップド I/O の使用方法

Neuron 3150 チップまたは FT 3150 スマート・トランシーバには、メモリ・マップド I/O デバイスを接続できます。これらのデバイスは、ROM、EEPROM、RAM の構築済みメモリ・マップ・エリア以外の部分にあたるメモリ・アドレスに割り当てます。

Neuron C プログラムからメモリ・マップド I/O にアクセスするには、デバイスの制御アドレス・ブロックへの定数ポインタを宣言します。以下の例では、あるメモリ・マップド I/O デバイスが、2 つのコントロール・レジスタと 16 ビットのデータ・レジスタを、それぞれアドレス *x*、*x+1*、*x+2*、*x+3* 番地に持つものと仮定しています。デバイスは、0x8800 から 0x8803 番地に応答するように接続されています。以下に、デバイスにアクセスする部分の Neuron C のコード例を示します。

```
typedef struct {
      unsigned short int      controlReg1;
      unsigned short int      controlReg2;
      unsigned long int dataReg;
}  *PMemMapDev;

const PMemMapDev pDevice = (PMemMapDev) 0x8800;

// Read from device ...
unsigned int x, y;
unsigned long z;

x = pDevice->controlReg1;
y = pDevice->controlReg2;
z = pDevice->dataReg;

// Write to device ...
unsigned int x, y;
unsigned long z;

pDevice->controlReg1 = x;
pDevice->controlReg2 = y;
pDevice->dataReg = z;
```

Neuron チップに収まらないプログラムの解決方法

ここでは、Neuron チップにプログラムが収まらなかったときに EEPROM の必要量を減らす手法とテクニックについて説明します。これらのテクニックの中には、Neuron 3120 チップまたは FT 3120 スマート・トランシーバ専用のものもありますが、ほとんどは Neuron C 言語のどのプログラムにも適用できます。これらのテクニックは、できるだけここに説明した順序で実行するようにしてください。

リンク・サマリーには、プログラムの現時点におけるメモリ使用についての情報が入っています。また、サマリー情報の中には、必要とする追加メモリの推定値も含まれています。オプションの指定をすれば、リンク・サマリーを BUILD.LOG ファイルに出力することもできますし、リンク・マップ・ファイルに含めることもできます。

アドレス・テーブル・エントリ数の削減

すべてのデバイスが接続された状態で、Neuron アプリケーション・プログラムに必要なアドレス・テーブル・エントリ数の最小値は、ポーリングされていない出力ネットワーク変数の数と、ポーリングされている入力ネットワーク変数の数と、バインド可能メッセージ・タグの数の合計であるという規則を覚えておいてください（バインド可能メッセージ・タグとは、宣言の中に**bind_info(nonbind)**を含まないメッセージ・タグのことです）。例えば、メッセージ・タグ 1 個と出力ネットワーク変数 2 個（そのうちの 1 つは要素 4 つの配列）を持つアプリケーションでは、アドレス・テーブル・エントリが 6 個以上必要です。

ただし、複数のグループに属している入力ネットワーク変数があると、各グループに 1 エントリが必要になるため、さらにアドレス・テーブル・エントリが必要になる場合があります。また、デバイス上の入力ネットワーク変数または出力ネットワーク変数に関連付けられている別名ネットワーク変数に対しても、アドレス・テーブル・エントリが必要になる場合があります。さらに、デバイスの「**msg_in**」タグへの各グループ接続も、アドレス・テーブル・エントリを使用します。

プログラムが明示的メッセージを受信せず(「**msg_in**」タグへの接続がない)、少数のネットワーク変数が（グループ接続ではなく）ポイント・トゥー・ポイントを使用する場合、アドレス・テーブル・エントリ数を簡単に減らせます。その他の状況でアドレス・テーブル・エントリ数を減らせるかどうかを判断するには、さらに解析する必要があります。

アドレス・テーブル・エントリ数のデフォルトは 15 です。この値を減らすには、**#pragma num_addr_table_entries** 指令を使用します（『Neuron C Reference Guide』の「Compiler Directives」の章を参照してください）。アドレス・テーブル・エントリ数を減らすと、1 エントリ当たり 5 バイトの EEPROM を節約できます。

不必要な自己識別データの削除

Neuron C コンパイラは、自己識別用データをデバイスのプログラム領域に置きます。Neuron 3120 チップまたは FT 3120 スマート・トランシーバの場合、そのプログラム領域は EEPROM にあります。プログラムが SNVT を使用していなければ、自己識別データを削除しても構いません。自己識別データを削除するには、次のコンパイラ指令を指定します。

```
#pragma disable_snvt_si
```

不要なネットワーク変数名の削除

Neuron C コンパイラは、プログラム内の**#pragma enable_sd_nv_names** コンパイラ指令に遭遇したときに、ネットワーク変数名に関する情報をデバイスのプログラム領域に配置します。Neuron 3120 チップおよび FT 3120 スマート・トランシーバでは、これによって EEPROM が消費されます。

この指令を削除すると、ネットワーク変数名の各文字に対して 1 バイト、さらに各ネットワーク変数に対して 1 バイト分の EEPROM 領域を節約できます。デバイスがインストールされたときにネットワーク変数名に関する追加情報

がない場合には、ネットワーク管理ツールによって「NVI1」「NVO7」などの一般的な名前が自動的に割り当てられます。

自動生成された一般的なネットワーク変数名ではなく、わかりやすい名前を使用するには、デバイスに外部インターフェース・ファイルを提供するようにしてください。ネットワーク管理ソフトウエアはファイル（.XIF や.XFB などの拡張子）からネットワーク変数名を抽出することができ、デバイスのコード領域を消費する必要もありません。

定数データの適切な宣言

定数データの宣言において **const** キーワードまたは **eeprom** キーワードを使用することは非常に重要です。これらのキーワードのどちらかがないと、コンパイラはデータが RAM に配置されているものと見なすため、アプリケーション・プログラムがリセットされるたびにランタイム時にデータを初期化する必要が生じます。これによってコード領域が多く消費されるだけでなく、RAM メモリも不当に消費され、アプリケーションがリセット処理を完了するのに要する時間も不当に長くなってしまいます。

次の例を考えてください。この例は、長さが 4 バイトの不適切に宣言されたデータ・テーブルを示しています。この宣言では **const** キーワードを使用していないため、コンパイラによってこの宣言は RAM に配置されます。このため、アプリケーション・プロセッサがリセットされるたびに初期化が必要となります。初期化のための実行コードを作成するのに追加の 9 バイトが必要となり、テーブルの初期値を保持するためにさらに 4 バイトがコードに使用されます。

また、データ・テーブルに RAM を使用すると、意図しないプログラミング・エラーによってこの内容が誤って変更されてしまう可能性があります。

不適切な宣言例:

```
int lookup_table[4] = {1, 4, 7, 13};
```

データ・テーブルの適切な宣言では、データの値に 4 バイトの読み取り専用メモリ（コード領域）しか使用しません。

適切な宣言例:

```
const int lookup_table[4] = {1, 4, 7, 13};
```

効率的な定数値の使用

Neuron チップおよびスマート・トランシーバの CPU アーキテクチャでは、0～7 の範囲の定数を使用すると、より大きな 8 ビットの定数を使用するよりも効率が向上します。これらの定数値を使用した命令はより小さく高速になるため、一連の定数値を選択するときは、定数が 0 で始まるように正規化してください。列挙型（**enum**）は、デフォルトによりゼロで正規化されます。

また、アプリケーションがリセットされるときには Neuron ファームウエアが自動的に RAM をゼロに初期化するため、定数のシーケンスにはゼロの初期値を使用するようにします。次の「Neuron ファームウエアのデフォルト初期化動作の利用」のセクションでは、この特性を活用する方法について説明します。

もう 1 つ考慮すべき点に、比較があります。これは特に **while** 文の中など、ループ制御式で使用するときに重要です。定数がゼロであると、式と定数は最も効率的に比較できます。ループのテストをゼロと比較できない場合には、1 と比較するようにしてください。1 との同等比較はゼロと比較した場合ほど効率が良くありませんが、その他の定数との比較よりも効率的です。

Neuron ファームウエアのデフォルト初期化動作の利用

Neuron ファームウエアは、チップがリセットされたとき、および Neuron C の **application_restart()**関数が呼ばれたときに、すべての RAM 変数をゼロにします。この動作を行ってから Neuron C アプリケーション・プログラムの起動が始まります。Neuron C アプリケーション・プログラムが起動すると、最初にすべての RAM 変数をゼロ以外の値に初期化するコードを実行します。次に **when(reset)**節のタスクがあれば、それを実行します。

つまり、RAM 変数をゼロに設定するためのコンパイル時の初期化子をわざわざ指定する必要はありません。また、**when(reset)**節のタスクの中に RAM 変数をゼロにするコードがあれば、それも必要ありませんから取り除くようにしてください。

出力 I/O オブジェクトのコンパイル時の初期化子の指定も、コード・スペースを消費しません。このことは初期化子の値に関係なく成立します。I/O に対してコンパイル時の初期化子を使用した方が、**when(reset)**節のタスクの中で **io_out()**を呼び出すよりもコード・スペースが小さくなります。

最後に、リセット時にはすべての Neuron C アプリケーション・タイマーが自動的にオフになりますから、**when(reset)**節のタスクの中にアプリケーション・タイマーをオフにするコードがあれば、全て取り除いてください。

Neuron C ユーティリティ関数の効果的利用

Neuron C ユーティリティ関数を使用して、コード量を少なくすることができます。例えば、**min()**、**max()**および **abs()**関数は、一般的な演算用のユーティリティ関数です。これらの関数を使用すれば、C の演算子を使ってインラインで演算するよりもコード・スペースが小さくなります。

Neuron C ユーティリティには、**high_byte()**、**low_byte()**、**make_long()**、**swap_bytes()**などのように、バイトを操作する関数があります。また、**clr_bit()**、**reverse()**、**rotate_long_left()**、**rotate_long_right()**、**rotate_short_left()**、**rotate_short_right()**、**set_bit()**、**tst_bit()**などのように、ビットを操作する関数もあります。

さらに正確な演算を提供するために、Neuron C には **muldiv()**、**muldivs()**、**muldiv24()**、および **muldiv24s()**の関数が用意されています。これらの関数を使用すると、乗算の後に除算を行い、その中間結果と演算を 32 ビットまたは 24 ビットの精度で得ることができます。

Neuron C 関数には **timers_off()**関数のようなユーティリティも含まれています。この関数を使用すると、1 回の関数呼び出しですべてのアプリケーション・タイマーをオフにすることができます。この関数呼び出しは、各タイマーにゼロを代入していくよりも実行時間は長くなりますが、コード・スペースは少なくてすみます。このため、プログラムに 1 つのアプリケーション・タイマーが含まれていて、ゼロを代入することでそれをオフにしている場合

には、この関数を代わりに使用して、コード領域を節約することを考えてください。

その他の関数には **bcd2bin()**と **bin2bcd()**、**delay()**と **scaled_delay()**、および **random()**があります。

Neuron C の関数は『Neuron C Reference Guide』で例を交えて詳しく説明しています。

ライブラリ使用の注意

アプリケーション・メモリに配置されるシステム関数には注意が必要です（メモリに配置される各チップおよび各バージョンのファームウエア関数のリストについては、『Neuron C Reference Guide』の表を参照してください）。構造体での **signed** ビットフィールドのようなライブラリ関数の使用につながるものは避けてください。

より効率的なデータ型の使用

マシン・アーキテクチャや命令セットにデータ項目や処理操作がマッチすればするほど、Neuron C コンパイラはさらに効率的なコードを生成します。可能であれば、グローバル変数ではなくローカル変数、**long** よりも **short**、**signed** よりも **unsigned** を使用するように変数を変更してください。

例えば、次に示す関数は、配列 **a** の中で **value** に等しい値を検索し、そのインデックス値を返します。

```
<type> find(int a[], int value, <type> count)  {
      <type>   i;
      for (i=0; i<count; ++i) {
            if (a[i] == value) break;
      }
      return i;
}
```

この関数をコンパイルすると（LonBuilder 3.0 Neuron C コンパイラを使用）、各データ型に応じて次のようなサイズのコードが生成されます。

```
<type>が signed short の場合：    25 バイト
<type>が unsigned short の場合：  24 バイト
<type>が signed long の場合：     34 バイト
<type>が unsigned long の場合：   34 バイト
```

上記の **unsigned short** 以外はすべて、ファームウエアのヘルパー関数を何回か呼び出します。つまり、上記のコード・サイズの数値だけではなく、**unsigned short** を使用したコード・シーケンスは見た目よりも実行効率がずっといいことを意味しています。このように、最も効率のよいコードを生成するデータ型は **unsigned short** です。これは、Neuron チップおよびスマート・トランシーバの命令セットが、最も効率よく処理できるのが 8 ビットの符号なし整数であるためです。

また、Neuron チップおよびスマート・トランシーバの CPU に採用されているスタック・アーキテクチャに注意を向けると、効率的にコンパイルできるコードの作成方法を理解するのに役立ちます。一般的なガイドラインとして、アクティブなローカル変数とパラメータを合計したサイズは 8 バイト以下に

抑えるようにしてください。これによって、最小サイズの命令を使用してすべてのローカル変数やパラメータにアクセスし、保存できるようになります。次の「宣言順序による影響」のセクションでは、最初のローカル変数に効率よくアクセスする方法と、この特性を利用する方法について説明しています。

Neuron C 言語では、配列、構造体、共用体などの集合体をローカルのデータ・スタックで宣言できます。このようなローカル変数の集合体が宣言される結果、コンパイラはより大きくて処理の遅い命令を使用してこれらのデータ項目にアクセスするようになります。したがって、このような変数はローカル変数ではなく静的項目として宣言するのが一番です。データのメモリが制限されており、これらの集合体をローカル変数として宣言する必要がある場合には、集合体以外のローカル変数の後でこれらを宣言するようにしてください。そうすることで、コンパイラは集合体以外の変数に短い命令を使用するようになります。

宣言順序による影響

関数内での自動変数の宣言順序がコード・サイズに影響することがあります。コンパイラは、最初に宣言された変数を実行時スタックの一番上に置き、二番目以降をその下に順に置いていきます。スタックの一番上の変数にアクセスするとき、特にその変数が **short** 型であれば、Neuron コンパイラは最も効率のよいコードを生成します。もっともコード効率が悪くなるのは、スタックの一番下に置かれた変数にアクセス（ロードおよび記憶）する場合です。したがって、効率のよいコードを生成しようとするなら、最も頻繁にアクセスされる変数を最初に宣言しておきます。

次のコード例を参考にしてください。

```
void arrayinit(int a[], int initval,
               unsigned count) {
      int j;
      unsigned i;
      j = initval;
      for (i=0; i<count; ++i) {
            a[i] = j;
      }
}
```

この関数によって生成されるコード・サイズは、23 バイトです。ただし、**i** は **j** よりも頻繁に使用されているので変数 **i** を先に宣言しておけば、生成されるコードは 21 バイトになります。

ファースト・アクセス機能オプション

Neuron C では通常、配列へのアクセス（ロードおよび記憶）に ANSI 標準 C の規則を使用します。この標準規則では、配列のインデックスを符号付き数値として解釈できるうえ、配列の宣言境界を越えるような配列インデックス値も許可しています。配列にこのようなインデックス方法を使用すると、配列参照のためのコード・サイズが大きくなります。

Neuron のマシン命令セットでは、以下の規則に従えば、サイズの小さな配列へのアクセスに対して効率のよいコードを生成できます。

1　配列インデックスが **signed short** 型のとき、コンパイラが **unsigned** 型に変換できること。

2　プログラムが配列の境界外へのアクセスを行わず、配列の境界を越えた配列要素のアドレスも計算しないこと。ANSI 標準 C では、ポインタを使ったループを終了させるときなどのために、このようなアドレスの計算ができるようになっています。同様の計算を **fastaccess** 配列で行うと、予期できない結果になります。

配列へのアクセスにこれらの追加規則を適用することをコンパイラに指示するには、配列の宣言に **fastaccess** キーワードを指定します。また、配列の総サイズは 254 バイト以下にします。**fastaccess** キーワードは、どの宣言文にでも使用でき、宣言文に含まれる全配列に対して適用されます。例えば、次のように宣言すれば、配列 **a1** と配列 **a2** の両方が **fastaccess** 配列になります。

```
fastaccess int a1[4], a2[12];
```

fastaccess キーワードは、**network**、**far**、**eeprom**、**const** など、他の宣言文シンタックスと組み合わせて使用できます。**fastaccess** 配列は、グローバル・メモリ内だけでなく、ローカル・プロシージャや関数スタックにも使用できます。ただし、**fastaccess** 機能はポインタを使ったインデックス演算には使用できません。

fastaccess 配列をグローバル・メモリ内で使用した場合の欠点は、リンカがこれらのデータ項目をページ境界を越えないように配置することです（メモリ・ページは 256 バイトです）。この結果、たくさんのグローバル配列を **fastaccess** として宣言すると、メモリの断片化が起こり、メモリ使用量が増えることになります。

重複する式の削除

Neuron C コンパイラは、共通の計算式を自動的に削除することはありません。このような最適化は手動で行ってください。ほとんどの場合、これでコード・サイズを減らすことができます。次の最適化前と最適化後の例では、必要なコード・スペースに 4 バイトの違いがあります。最適化後の例における **temp** 変数は、スタック上の一番上の変数になるように宣言されます。

最適化前（コンパイルされたコードは 28 バイト）

```
int a, b, c, d, e;
void f(void) {
    d = (a * 2) +  (b * c * 4);
    e = a - (b * c * 4);
}
```

最適化後（コンパイルされたコードは 24 バイト）

```
int a, b, c, d, e;
void f(void) {
    int temp;
    temp = b * c * 4;
    d = (a * 2) + temp;
    e = a - temp;
}
```

気づきにくい共通の計算式のもう 1 つの形式に、配列のインデックスがあります。配列要素に対するインデックスの繰り返しを避ける例を、以下の最適化前と最適化後のコードに示します。一時的なポインタ変数を使用するとコード領域が節約されるだけでなく、コード自体も簡潔化されます（最適化前の例では配列にアクセスするたびに合計 3 回の乗算が行われているのに対し、最適化後の例では 1 回の乗算しか行われていません）。

最適化前（コンパイルされたコードは 46 バイト）

```
struct s {
      int x, y, z;
} a[5];


void f(int i) {
      a[i+2].x = 3;
      a[i+2].y = 5;
      a[i+2].z = 7;
}
```

最適化後（コンパイルされたコードは 33 バイト）

```
struct s {
      int x, y, z;
} a[5];


void f(int i) {
      struct s *p;
      p = &(a[i+2]);
      p->x = 3;
      p->y = 5;
      p->z = 7;
}
```

関数ライブラリの使用

関数呼び出しは、コード・サイズと実行時間オーバーヘッドが相対的に軽いものです。複雑なコードの中の 1 行でも 2 回以上使用される行があれば、それを同機能の関数に置き換えて、コード・サイズを減らすことができます。ネットワーク変数を伴うコードのいくつかは、見た目は複雑でなくても、そこで行われる操作は複雑である可能性があります。

例えば、ネットワーク変数配列の一部に含まれる構造体の 1 つの要素をインクリメントすることを考えてみましょう。この操作には、かなりの量のコードが生成されます。同じコードを 2 回使用する代わりに 1 個の関数呼び出しを使うと、若干の性能低下はありますが、コード・スペースは節約できます。

また、グローバル変数を使わずに関数の実引数として式や関数を渡すことも考えてみてください。引数へのアクセスは、一般的にグローバルへのアクセスよりも効率的です（少なくとも低下しません）。

別の初期化シーケンスの使用

#pragma disable_mult_module_init 指令を使用すると、EEPROM のコード領域を 2～3 バイト節約できます。この指令を指定すると、コンパイラが必要な初期化コードを special init やイベント・ブロックの中に直接生成します。通

常、この初期化コードは special init やイベント・ブロックから呼び出す別手続きになっています。

この指令を使用したことによって選択されるインライン方式は、メモリ使用に関してより効率的です（初期化コードがあれば 3 バイト、初期化コードがなければ 2 バイトの節約になります）。ただし、この指令を使用したときの欠点もあります。それは、（1）インラインの初期化エリアには長さの制限があること、（2）プログラムの初期化コードからアプリケーション・ライブラリやカスタム・イメージの初期化コードへのリンクがない可能性があることです（Neuron 3120 チップまたは FT 3120 スマート・トランシーバでは一般に問題になりません）。

ドメイン数の削減

アプリケーション・デバイスが常に単一のドメインのメンバであることがわかっていれば、**#pragma num_domain_entries 1** 指令を使用して 15 バイトの EEPROM を節約できます。ドメイン・エントリのデフォルト数は **2** で、各ドメイン・エントリは 15 バイトの EEPROM を使用します。

注意： LONMARK 相互運用性協会では、すべての相互運用可能デバイスは 2 つのドメインのメンバであることを義務づけています。ドメインの数を 1 つに削減すると、15 バイトの EEPROM 領域を節約できますが、デバイスが LONMARK の相互運用可能ガイドラインに準拠しなくなります。

C 演算子の効果的使用

ANSI C 言語には、演算子が豊富に用意されています。これらを効果的に使用することで非常に効率的なコードが作成できます。

例えば、同じ左辺に対する二者択一式の代入では、**if-else** シンタックスを使用するよりも C の **? :** 演算子を使用する方がコード・スペースが小さくなります。特に左辺の式が複雑な場合に有効です。

また、複数の **if-else** 節を使用すると、**switch** 節を使用するよりもコード領域の使用効率が若干向上します。次の最適化前と最適化後の例では、2 バイトのコードを節約できます。

最適化前（コンパイルされたコードは 40 バイト）：

```
void f (unsigned c) {
      switch (c) {
      case '1':
            f1();
      break;
      case '2':
            f2();
      break;
      case '3':
            f3();
      break;
      case '4':
            f4();
      break;
      default:
            f5();
      break;
      }
}
```

最適化後（コンパイルされたコードは 38 バイト）：

```
void f (unsigned c) {
      if (c == '1') {
            f1();
      } else if (c == '2') {
            f2();
      } else if (c == '3') {
            f3();
      } else if (c == '4') {
            f4();
      } else {
            f5();
      }
}
```

コードの効率を改善できるもう 1 つの C 言語演算子に連鎖式代入があります。連鎖式代入では、代入される値を代入後も使用できるという事実を利用しています。連鎖式代入では、代入される値の再ロードや再コンパイルが不要になります。これは次の最適化前と最適化後の例で示されています。

最適化前（コンパイルされたコードは 14 バイト）：

```
mtimer t1;
unsigned long int l;


void f(void) {
      t1 = 5000;
      l = 5000;
}
```

最適化後（コンパイルされたコードは 13 バイト）：

```
mtimer t1;
unsigned long int l;
```

```
void f(void) {
      t1 = l = 5000;
}
```

論理演算子**&&**および || を複雑な条件に使用すると、ビット演算子**&**および **|** を使用する類似の式と比較して、処理時間が常に高速になります。また、論理演算子を使用すると、特にゼロとの同等性または不同等性のテストが条件式の一部である場合はコードが少なくなります。以下に最適化前と最適化後の例を示します。

最適化前（コンパイルされたコードは 15 バイト）:

```
void f (int a, int b, int c) {
      if ((a < 0) | (b == 0) | (c > 0)) {
            // take some action
      }
}
```

最適化後（コンパイルされたコードは 12 バイト）:

```
void f (int a, int b, int c) {
      if ((a < 0) || (b == 0) || (c > 0)) {
            // take some action
      }
}
```

Neuron C 拡張機能の効果的使用

Neuron C 言語では、効率的なコードを作成するのに役立つ機能を数多く採用しています。

例えば、プログラムに入力ネットワーク変数が 2 つあり、このどちらかの変数が更新されたときに実行されるタスクが 1 つある場合、以下の最適化後の例に示したコードの方が効率的になります。同様に、配列名だけ指定した **nv_update_occurs** イベントを **when** 節に指定した方が、配列の各要素に **when** 節を使うよりも効率的です。

最適化前（コンパイルされたコードは 6 バイト）:

```
when (nv_update_occurs(var1))
when (nv_update_occurs(var2))
{
   // task ...
}
```

最適化後（コンパイルされたコードは 3 バイト）:

```
when (nv_update_occurs)
   // Use "unqualified" event to cover all variables
{
   // task ...
}
```

ただし、上に示す非限定イベントを使用せずに特定の **nv_update_occurs** イベントを使用する必要がある場合は、次のガイドラインに従ってください。

2 つのネットワーク変数 **nviA** と **nviB** を宣言するプログラムを考えてみてください。

```
network input SNVT_switch nviA, nviB;
```

次のコード例の機能はどれも同じです。いずれも、2 つのネットワーク変数のどちらかに対する着信ネットワーク変数の更新に応答します。最初の実装は一番非効率的で、最後の実装（上の例の最適化前に相当）が最も効率的です。

バリエーション 1（コンパイルされたコードは 15 バイト）：

```
when (nv_update_occurs(nviA) || nv_update_occurs(nviB))
{
      // body of task
}
```

バリエーション 2（コンパイルされたコードは 9 バイト）：

```
when (nv_update_occurs(nviA..nviB))
{
      // body of task
}
```

バリエーション 3（コンパイルされたコードは 6 バイト）：

```
when (nv_update_occurs(nviA))
when (nv_update_occurs(nviB))
{
      // body of task
}
```

Neuron 3120 チップのシステム・ライブラリ

すべてのバージョンの Neuron 3120 チップと FT 3120 スマート・トランシーバで、すべてのアプリケーション・コードがオンチップ EEPROM に配置されます。さらに、各種 I/O 関数やライブラリ関数を使用しているかどうかをリンカが調べて、オンチップ EEPROM 上に移します。ライブラリ関数の詳細については『Neuron C Reference Guide』を参照してください。

ライブラリ関数の使用を必要とするチップおよびファームウエアのバージョンにリンクされているアプリケーションには、Neuron 3150 チップまたは FT 3150 スマート・トランシーバにリンクされている同じアプリケーションよりも多くのオンチップ EEPROM が必要になることがあります。チップとファームウエアによっては、これらの関数が Neuron ファームウエアではなくシステム・ライブラリに存在する場合があるためです。リンク・マップを検討すれば、これらの関数によって使用される EEPROM メモリが計算できます。これらの関数に必要な Neuron 3120 チップまたは FT 3120 スマート・トランシーバの EEPROM の量を求めるには、以下の手順に従います。

1. LonBuilder または NodeBuilder で、リンク・マップを生成するためのオプションを選択します。
2. デバイスに適した Neuron チップのモデルを選択します。
3. **Build** コマンドを選択します。

ビルドが完了すると、このデバイスのリンク・マップのリンク・サマリー領域に、システム・ライブラリ関数に必要な Neuron 3120 チップまたは FT 3120

スマート・トランシーバの EEPROM サイズが書き込まれます。リンク・マップの詳細については『Neuron C Reference Guide』を参照してください。

付録A

Neuron C ツールの
スタンドアロンでの使用

付録 A では、Neuron C ツールをスタンドアロン・プログラムとしてコマンド・ラインから使用する方法について説明します。

これらのツールのオプション一覧を表示したい場合は、コマンド・プロンプトで各ツールの名前を入力してください。例えば、コマンド・プロンプトで **ncc** と入力すると、Neuron C コンパイラのコマンド・ライン・オプションが表示されます。

スタンドアロン・ツール

以下に示す Neuron C ツールは、「スタンドアロン」として、統合開発環境の外で、コマンド・プロンプトまたはコマンド・ウィンドウのみで使用できます。

説明	ツール名
Neuron アセンブラ（LonBuilder & NodeBuilder ツール）	NAS
Neuron アセンブラ（LonBuilder ツール）拡張メモリ	NASX
Neuron C コンパイラ（LonBuilder ツール）	APC
Neuron C コンパイラ（LonBuilder ツール）拡張メモリ	APCX
Neuron C コンパイラ（NodeBuilder ツール）	NCC
Neuron エクスポータ（NodeBuilder ツールのみ）	NEX
Neuron ライブラリアン（LonBuilder & NodeBuilder ツール）	NLIB
Neuron リンカ（LonBuilder & NodeBuilder ツール）	NLD
Neuron リンカ（LonBuilder ツール）拡張メモリ	NLDX
プロジェクト作成（NodeBuilder ツールのみ）	PMK

NodeBuilder スタンドアロン・ツールはいずれも共通のコマンド・ライン技術を共有するため、使用にあたっていくつかの共通点があります。これらの共通点については、次の「共通のスタンドアロン・ツールの使用」のセクションで説明します。その後の各セクションでは、上のリストの各ツールを簡単に紹介します。

注意： NodeBuilder 開発ツールのユーザは、Neuron ライブラリアンとプロジェクト作成ユーティリティ以外のコマンド・ライン・ツールを使用しないでください。作成ツールはプロジェクト作成ユーティリティ、**pmk.exe** によって制御する必要があるためです。このユーティリティは作成プロセスを管理するだけでなく（必要な作成ステップ数を最小化します）、プログラム ID の管理と自動起動 ID 処理も行います。このユーティリティを使用しないと、これら２つの重要なタスクを手動で行わなければならなくなります。プロジェクト作成ユーティリティの詳細については、『NodeBuilder User's Guide』を参照してください。

共通のスタンドアロン・ツールの使用

共通点

すべてのツールには次の共通点があります。

- ツール名の後にコマンドのスイッチまたは引数が指定されていない場合は、ツールの使用法が表示されます。

  例

  C:¥>**NAS**

  ツールの応答

  ```
  Neuron (R) Assembler, version 3.10.13, build 49
  Copyright (C) Echelon Corporation 1989-2001

  Usage:  [optional command(s)] argument

  ...  （以下省略）
  ```

- ほとんどのコマンド・スイッチには、短い形式と長い形式の 2 種類があります。短い形式の前にはスラッシュ「/」またはダッシュ「-」を 1 つ指定し、その後に、コマンドを識別する文字を 1 文字指定する必要があります（大文字と小文字は区別します）。

  短い形式の例

  C:¥>**NCC -DMYMACRO ...**

  コマンドの長い形式の前には 2 つのダッシュ「--」を付けて、その後に大文字と小文字の区別された語句のコマンド名を指定します。

  長い形式の例

  C:¥>**NCC --define=MYMACRO ...**

  短いコマンド・スイッチは、半角のスペースまたは等号を使用して、対応する値と区別できます。短いコマンド・スイッチには区切り文字は必要ありません。上の例のように、コマンドの識別子のすぐ後に値を指定できます。

  長いコマンド・スイッチには半角のスペースまたは等号の区切り文字が必要です。

- 複数のコマンド・スイッチは半角スペースで区切ることができます。

  例

  C:¥>**NCC --define=MYMACRO1 --define=MYMACRO2 ...**

- ブール型のコマンドには値を指定する必要がありません。何も指定しないと、yes が指定されているときと同じ結果になります。ブール型コマンドに指定できる値には、yes、on、1、no、off、0 があります。

例

C:¥>**NCC --kerneldbg=yes ...**

上の例は、下の例と等しくなります（ブール型コマンドはデフォルトで yes になるためです）。

C:¥>**NCC --kerneldbg ...**

- コマンドはコマンド・ラインから読み取ることができますが、コマンド・ファイル（スクリプト）から読み取ることもできます。コマンド・ファイルには、空の行、セミコロンで始まる行（コマンド・ライン）、または各行に1つのコマンド・スイッチ（値を含む場合もある）を含む行で構成されています。

  短いコマンド・シンタックスは対話的なコマンド・ラインでよく使用されます。一方、長いコマンド・ライン・シンタックスはわかりやすいため、コマンド・ファイルでの使用が好まれます。

  コマンド・ファイルの例

```
; Example command file, containing
; some of the Exporter's commands
; Created Wednesday, November 21, 2001, 20:42:20

--bootflags=1024
--infolder=d:¥lm¥Source¥EPR¥23305¥Development¥IM
--outfolder=d:¥lm¥Source¥EPR¥23305¥Development
--basename=23305
```

- ほとんどのツールは、引数を追加する必要があります。これらの引数は、コマンド・ライン内のどの場所に指定しても、またスクリプト内の別の行に指定してもかまいません。

  コマンド・ラインの最後に指定した引数の例

  C:¥>NCC **--define=MYMACRO mycode.nc**

基本的なコマンドの共通セット

前述のシンタックス面での共通点に加えて、スタンドアロン・ツールでは基本的なコマンドの共通セットも共有しています。これらの共通コマンドの一部を以下に表示します。使用可能なコマンドの全リストを取得するには、コマンドを指定せずに、スタンドアロン・ツール名を入力します。

-@ file-pathname （コマンド・ファイルを含む）

-@（または**--file**）コマンドは、コマンド・ファイル（スクリプト）を指定します。コマンドはこのスクリプトから読み取られ、@コマンドの場所からコマンド・ラインを入力しているのと同じ効果が得られます。スクリプト自体が他のスクリプトを参照することもできます。

--defloc *dir* （オプションのデフォルト・コマンド・ファイルの場所）

コマンド・ライン・ツールではデフォルトのスクリプトも検索します。このスクリプトは、コマンド・ラインで指定された他のコマンドがすべて処理された後に追加で読み取られるファイルです。これらのデフォルトのスクリプト・ファイルは、**-@**コマンドを使用して指定する必要がありません。次の表に示すように、定義済みの名前があるためです。コマンド・ライン・ツールは、デフォルトのスクリプトが現在の作業ディレクトリに存在するものと見なします（存在しなくてもエラーにはなりません）。デフォルトのスクリプトの場所（名前ではない）を指定するには、**--defloc** コマンドを使用します。NodeBuilder 開発ツールは NodeBuilder デバイス・テンプレート・ファイル（拡張子は**.nbdt**）をデフォルトのスクリプトの場所として使用します。

作成ツール	コマンド名	デフォルトのスクリプト名
Neuron C コンパイラ	NCC	LonNCC32.def
Neuron アセンブラ	NAS	LonNAS32.def
Neuron リンカ	NLD	LonNLD32.def
Neuron エクスポータ	NEX	LonNEX32.def
Neuron ライブラリアン	NLIB	LonLIB32.def

--mkscript *scriptfile*　　（*scriptfile* にコマンド・スクリプトを生成）

--mkscript コマンドは、作成ツールが受け取ったコマンドをすべて含む追跡ファイルを生成します。これらのコマンドはどこから送られたものであっても構いません。デフォルト・スクリプト内でこの機能を使用すると、プロジェクト・マネージャが使用するコマンド・シーケンスを取り込むことができます。これにより、マシンが生成した作成スクリプトを簡単に取得することができます。

注意： *scriptfile* に指定するスクリプト・ファイルはデフォルトのスクリプト・ファイル、またはその他の保持しておきたいスクリプト・ファイルを上書きしないような方法で指定してください。**--mkscript** コマンドを使用すると、コマンドの流れが常に追跡されるため、警告を表示することなく既存のファイルが上書きされます。

--warning *text*　　（警告として *text* を表示）

このコマンドは、スクリプト・ファイルの中でのみ使用できるコマンドで、*text* のメッセージを警告として表示します。前述の**--mkscript** コマンドは、監視対象のコマンド・ストリームをツールが実行できず、エラーも発生しなかった場合、**--warning** コマンドを自動的に挿入します。

その後マシンで生成したスクリプト・ファイルを使用すると、このスクリプトはマシンで生成されたものであり、エラーの可能性があるコマンド・ストリームを含んでいることを示す警告が表示されます。

スタンドアロン・ツールのコマンド・スイッチ

このセクションでは、最も役に立つ一般的なコマンド・スイッチについて、スタンドアロン・ツールごとに説明します。

Neuron C コンパイラ

Neuron C コンパイラの名前は **ncc.exe** です。このスタンドアロンのコンパイラはコマンド・プロンプトから実行し、Neuron のアセンブリ・ソース・ファイルを生成します。コンパイラのコマンド・ラインには、実行ファイル名、オプションのコマンド・スイッチ、コンパイルするファイル名の順に指定します。

例

C:¥>NCC mycode.nc

よく使用するスイッチに**-D（--define）**スイッチと**-I（--include）**スイッチがあります。**-D** スイッチは、シンボルをコマンド・ラインから定義するときに使用します。ここで定義したシンボルは、**#ifdef** 指令や**#ifndef** 指令を使ってプログラム内でテストできます。

-I（--include）スイッチは、インクルード・ファイルを含むディレクトリを指定するときに使用します。**-I** を複数使用することで、複数のインクルード・ディレクトリを指定することもできます。複数の**-I** スイッチを指定した場合、指定した順番にディレクトリが検索されます。

例

C:¥>ncc -DVERSION5 -I..¥include -Id:¥include mycode.nc

.nc の拡張子を持つファイル名を呼び出すと、Neuron C コンパイラは Neuron C の規則に従ってコードを生成します。ただし、ライブラリとカスタム・システム・イメージには Neuron C コードを使用できません。「基本 C」ファイルをコンパイルして「基本 C」規則に従ったコードを生成するには、以下のコマンド・ライン例に示すように、ファイル名の拡張子を**.c** にする必要があります。

例

C:¥>ncc -I..¥include mycode.c

最後の例では、myincs という名前のサブディレクトリに myinc.h というインクルード・ファイルを持つ myfile.nc をコンパイルし、条件付きコンパイルの目的で OPTION1 シンボルを定義しています。

例

C:¥>ncc –Imyincs -DOPTION1 myfile.nc

コンパイル時にエラーがなければ、このコマンドによって複数の出力ファイルが作成されます。これらのファイルはすべて、Neuron C ソース・ファイルに使用された基本名を共有しています。上の「基本 C」の例では、生成されたファイルにはすべて myfile という基本名が付きますが、それぞれの拡張子は異なります。

「基本 C」のコンパイルでは、.ns の拡張子を持つアセンブリ・ソース・ファイル以外、生成されたファイルはすべて破棄できます。アセンブリ・ソース・ファイルは Neuron C のコンパイル時に他のツールによって必要とされることがあるため、破棄できません。

Neuron アセンブラ

Neuron アセンブラの名前は **nas.exe** です。Neuron アセンブラは、Neuron C コンパイラをサポートする目的のみに提供されているもので、Neuron アセンブリ言語のアプリケーションを生成するために使用するものではありません。

Neuron アセンブラをコマンド・プロンプトから実行すると、Neuron オブジェクト・ファイルが生成されます。アセンブラのコマンド・ラインには、実行ファイルの名前、オプション・スイッチ、アセンブルするファイルの名前の順に指定します。よく使用するアセンブラ・スイッチに、**-l** スイッチ（長い形式は**--listing**）があります。このオプションを指定すると、アセンブラ・リストが作成されます。

前のセクションの例を引き続き使用すると、myfile.ns ファイルをアセンブルして myfile.nl リスト・ファイルと myfile.no オブジェクト・ファイルを生成するコマンドは、次のようになります。オブジェクト・ファイルが作成されたら、中間アセンブリ・ファイルである myfile.ns ファイルは削除して、ディスク容量を節約できます。

例

C:¥>**NAS -l myfile.ns**

Neuron リンカ

Neuron リンカの名前は **nld.exe** です。リンカをコマンド・プロンプトから実行すると、Neuron 実行ファイルを生成できます。リンカのコマンド・ラインには、実行ファイル名、１つ以上のスイッチ、リンクするオブジェクト・ファイルの名前の順に指定します。正しいリンクを生成するには、いくつかのスイッチを組み合わせて使用する必要があります。

-a（--appimage）スイッチは、Neuron アプリケーション・プログラムをリンクするときに必ず使用してください。

-t（--neurontype）スイッチには、アプリケーションのリンクの対象となる Neuron チップまたはスマート・トランシーバの名前を指定します。

例

C:¥>**NLD -a -t3120E2 ....**

Neuron 3150 チップまたは FT 3150 スマート・トランシーバ用にリンクするときは、スイッチを使用して、外部メモリ・マップを指定する必要があります。スイッチは、外部の RAM、EEPROM、ROM などのエリアの開始または終了番地を指定するときに使用します。各スイッチの後には、エリアの最初（または最後）のページ番号のアドレスを 16 進で指定します。 1 ページは 256 バイトですから、16 進 4 桁のバイト・アドレスの上位 2 桁がページ番号になります。

-r および**-R** スイッチは、それぞれ外部 RAM の最初および最後のページを指定します。**-e** および**-E** スイッチは、それぞれ外部 EEPROM の最初と最後のページを指定します。**-Z** スイッチは、ROM の最後のページを指定します（ROM は必ず 0000 番地から始まるため、ROM の最初のページを指定するスイッチはありません）。外部メモリを持たないエリアに対しては、これらのスイッチを使用しないでください。ここにはメモリ・マップド I/O エリアを指定することはできませんし、メモリ・マップド I/O エリアは指定した外部メモリ・エリアの範囲外でなければなりません。

例えば、外部 ROM を 0000～7FFF、外部 EEPROM なし、外部メモリ・マップド I/O デバイスを 8000～97FF、外部 RAM を 9800～9FFF にしてリンクするには、次のようにスイッチを指定します。

C:¥>**nld –Z=7F –r=98 –R=9F ....**

リンカには、リンクするアプリケーションが使用するシステム・イメージに対応したシンボル・テーブルが必要です。これには**-p** スイッチを使い、半角スペースを 1 つ空けて、イメージのシンボル・ファイルのパス名を指定します。

例えば、3150 カスタム・デバイスにリンクする場合、イメージの名前は SYS3150 になります。Neuron C ソフトウエアがインストールされているディレクトリが C:¥LonWorks だとすると、このメイン・ディレクトリにはいくつかのサブディレクトリがあります。IMAGES ディレクトリには **VER*nnn*** という名前のサブディレクトリがあります。ここで、***nnn*** はゼロで始まらない 2～255 の数字です。

それぞれの **VER*nnn*** サブディレクトリには、標準イメージ・ファイルが保存されています。例えば、ソフトウエアが C:¥LonWorks（デフォルト）にインストールされている場合、ファームウエアのバージョン 7 にリンクするスイッチは次のようになります。

C:¥>**NLD -p C:¥LONWORKS¥IMAGES¥VER7¥SYS3150.SYM ....**

これまでに説明したスイッチは、Neuron リンカに最小限必要なものです。

-o スイッチを使用すると、デフォルトの出力ファイル名とは異なるファイルを指定できます。デフォルトの出力ファイル名は、リンク・コマンド・ラインに指定した最初のオブジェクト・ファイル名から拡張子を除いたものと同じになります。

-A スイッチを使用すると、EEPROM の代わりにフラッシュ・メモリを使って EEPROM を実装できます。**-A** スイッチの後にフラッシュ・メモリ・デバイスのセクタ・サイズ（64 または 128）を指定してください。

ライブラリをリンクに含めるには、**-l** スイッチの 1 つ以上のインスタンスを指定し、その後にライブラリ名を指定します。各ライブラリに**-l** スイッチを指定すると、複数のライブラリにリンクできます。

NodeBuilder 開発ツールを使用している場合は、NodeBuilder プロジェクト内でのライブラリの使用の詳細について、『NodeBuilder User's Guide』を参照してください。NodeBuilder ツールは、スタンドアロンのツールを使用しない限り、カスタム・ライブラリの作成をサポートしていません。ただし、カスタム・ライブラリを使用することはできます（詳細については付録 B を参照してください）。

カスタム・ライブラリの作成の詳細については、後述の「Neuron ライブラリアン」および付録 B を参照してください。

Neuron エクスポータ

Neuron エクスポータの名前は **nex.exe** です。エクスポータはコンパイラとリンカから入力を受け取り、デバイス・ファイル・セットを生成します。デバイス・ファイル・セットには、デバイス・インターフェース・ファイル（拡張子は.xif と.xfb）と必要なイメージ・ファイル（拡張子は.nri、.nfi、.nxe、.nei、および.apb）が含まれています。

エクスポータのコマンド・ラインには、スイッチを指定します。エクスポートするファイルのセットを正しく生成するには、複数のスイッチを組み合わせて使用する必要があります。

ブート ID を指定するには、**-t** スイッチ（**--bootid**）を使用します。スイッチの後には、エクスポートされるイメージ内のブート ID を示す 10 進数 0～65535 を指定します。この範囲内の値であればどれでも使用できますが、それぞれ一意のブート ID 値を使用して作成する必要があります。Neuron チップとスマート・トランシーバのリセット手順とブート ID の意味については、『FT 3120 and FT 3150 Smart Transceivers Databook』を参照してください。

-C（--clock）コマンドは、Neuron クロック・レートをエンコード値として指定するために使用します。例えば 10 MHz は「5」と指定します。全クロック・レートのリストについては、LonWorks Types ディレクトリ（デフォルトでは C:¥LonWorks¥Types）にある LonWorksUI.xml ファイルの ClockSpeeds フィールドを参照してください。

-c（--xcvr）コマンドを使うと、使用するトランシーバ・タイプをトランシーバの標準 ID で指定できます。例えば、TP/FT-10 フリー・トポロジ型トランシーバには「7」を使用します。標準の全トランシーバ ID のリストについては、LONWORKS Types ディレクトリ（デフォルトでは C:¥LonWorks¥Types）にある StdXcvr.xml ファイルの std_id フィールドを参照してください。

-I（--infolder）コマンドと**-O（--outfolder）**コマンドは、それぞれ入力ファイル（コンパイラとリンカが生成）および出力ファイル（エクスポータが生成）の場所を指定します。

-b（--basename）コマンドは、入力フォルダと出力フォルダに含まれている入力ファイルと出力ファイルの基本名を指定します。

--create*XXX* スイッチは、ファイル・タイプの生成を有効にするために使用します。ここで、***XXX*** には希望するファイル・タイプの拡張子を小文字で指定します。複数の**--create*XXX*** スイッチを使用して、複数のファイル・タイプを生成することもできます。

次の例は、TP/FT-10 フリー・トポロジ型トランシーバを使用する 10MHz ベースのデバイスに対して MyDevice という基本名を使用したデバイス・ファイルセットをエクスポートします。

```
C:¥>NEX -@commands.nex --bootid=12345
```

コマンド・ラインで参照している commands.nex スクリプト・ファイルには次のコマンドが含まれています。

```
; Sample Exporter command file
--clock=5
--xcvr=7
--infolder=MyProject¥MyDevice¥Development¥IM
--outfolder=MyProject¥MyDevice¥Development
--basename=MyDevice
--createxif=yes
--createxfb=yes
--createnei=yes
--createnxe=yes
--createapb=yes
--createnri=yes
```

Neuron ライブラリアン

Neuron ライブラリアンの名前は **nlib.exe** です。ライブラリアンを使用すると、ライブラリの作成・管理、オブジェクト・ファイルの既存のライブラリへの追加、既存のライブラリからの削除といった操作を実行することができます。ライブラリは「基本 C」関数で構成されています。Neuron C のコードをライブラリに含めることはできません。ただし、Neuron C 中の「基本 C」コードを含めることはできます。ライブラリアンは本書のこのセクションで説明しているツールの中で、唯一必須ではないツールです。ライブラリアンは LONWORKS デバイスの生成には必要ありません。

ライブラリアンが作成または変更したライブラリは、LonBuilder または NodeBuilder プロジェクト・マネージャ内、またはスタンドアロンの nld.exe Neuron リンカと共に使用できます。スタンドアロン・リンカでライブラリを使用する場合は、リンカのコマンド・セットの**-l** コマンドと**-L** コマンドを参照してください。プロジェクト・マネージャ内でライブラリを使用する場合は、『NodeBuilder User's Guide』または『LonBuilder User's Guide』を参照してください。

ライブラリアンをコマンド・プロンプトから実行するには、コマンド名、オプションのスイッチのセット、ライブラリ名、およびオプションのオブジェクト・ファイル名のリストを指定します。

新しいライブラリを作成するには、次のコマンド・ラインを入力します。

C:¥>**nlib -c -a** *library-name object-file* [*object-file ...*]

モジュールを既存のライブラリに追加するには、次のコマンド・ラインを入力します。

C:¥>**nlib -a** *library-name object-file* [*object-file ...*]

既存のライブラリにある既存のモジュールを置換（または更新）するには、次のコマンド・ラインを入力します。

C:¥>**nlib -u** *library-name object-file* [*object-file ...*]

既存のライブラリの内容をレポートとして出力するには、次のコマンドを入力します。レポートはコンソールに出力されますが、ファイルに出力することもできます。

C:¥>**nlib -r** *library-name*

サマリ・レポートを作成するには、次のコマンド・ラインを入力します。

C:¥>**nlib -s** *library-name*

例えば以下に示すコマンドでは、長い形式のスイッチを使用して、zorro.no と garcia.no のオブジェクト・ファイルを mylib.lib ライブラリに追加しています。

C:¥>**nlib --add mylib.lib zorro.no garcia.no**

スクリプト・ファイルを使用すると、ライブラリアンの入力を指定できます。例えば、「f1.no」～「f10.no」の 10 個のオブジェクト・ファイルを mylib.lib ライブラリにまとめるには、次のコマンド・ラインとコマンド・ファイルを入力します。

C:¥>**nlib -c2 -a mylib.lib @mylib.lst**

mylib.lst ファイルの内容は次のとおりです。

```
f1.no
f2.no
f3.no
f4.no
f5.no
f6.no
f7.no
f8.no
f9.no
f10.no
```

ライブラリアンのコマンド・ラインには、希望に応じて複数のスクリプト・ファイルを指定できます。また、オブジェクト・ファイル名だけを指定することも、これら 2 つを組み合わせることもできます。

ライブラリの使用の詳細については、付録 B「Neuron C 関数ライブラリ」を参照してください。「基本 C」ファイルのコンパイル方法については、本章で前述した「Neuron C コンパイラ」を参照してください。コンパイルした出力を Neuron オブジェクト・ファイルにアセンブルする方法については、本章で前述した「Neuron アセンブラ」を参照してください。前の例に示すように、Neuron オブジェクト・ファイルは.no の拡張子を使用してカスタム・ライブラリに追加できます。

付録B

Neuron C 関数ライブラリ

付録 B では、Neuron C ツールを使って独自の関数ライブラリやデータ・ライブラリを作成し、使用する方法について説明します。ライブラリの作成・使用には、LonBuilder 開発ツールと NodeBuilder 開発ツールのどちらも使用できます。

定義

アプリケーション・プログラム　コンパイルとアセンブルの後、システム・ファームウエア・イメージにリンクされる Neuron C のソース・プログラム。アプリケーション・プログラムは、スタンドアロン型の実行プログラムではありません。アプリケーション・プログラムはシステム・イメージへの外部参照を含んでいるため、対応するシステム・イメージが含まれているデバイスのメモリにロードして使用します。

ライブラリ　**nlib.exe** Neuron ライブラリアンが生成するファイル。Neuron アセンブラが生成する「基本 C」オブジェクト・ファイルで構成されています。Neuron リンカは、これらのオブジェクト・ファイルを必要に応じてライブラリから抽出し、Neuron C アプリケーション・プログラムに結合します。

基本 C　Neuron C 言語は、ANSI 標準 C 言語のサブセットに拡張を加えた言語セットです。本書の「基本 C」という用語は、Neuron C の拡張部分を除いたサブセットの言語を指しています。

スタンドアロン型ツール　「スタンドアロン」という用語は、Windows のコマンド・プロンプトからツールが使用でき、プロジェクト・マネージャの環境外でも利用可能であることを意味します。この付録で説明するツールはすべてスタンドアロン方式で実行できます。

LonBuilder によるライブラリのサポート

付録 A「Neuron C ツールのスタンドアロンでの使用」で説明するとおり、LonBuilder Neuron C のコンパイラ、アセンブラ、およびライブラリアンを使って「基本 C」ソース・ファイルから Neuron ライブラリを作成することができます。これらのライブラリをアプリケーション・プログラムの中で利用するには、LonBuilder プロジェクト・マネージャを使用します。

LonBuilder プロジェクト・マネージャは、ライブラリを構成しているソース・ファイルの変更を検知しないため、いつライブラリを再構築しなければならないかを判断できません。ライブラリを作成するときは、標準のソフトウエア・エンジニアリング技術を使用してソフトウエアを管理し、常に最新のソフトウエア・バージョンを使用するように考慮してください。サードパーティ製の「make」プログラムを使用すると、ソフトウエアの従属性を簡単に管理できます。

LonBuilder プロジェクト・マネージャは、「基本 C」で書かれた Neuron オブジェクト・コードやデータから成るライブラリとアプリケーション・プログラムとのリンクを完全にサポートします。プロジェクト・マネージャは、ライブラリを使用するときにすべての従属性をチェックし、ライブラリに変更があればそれを自動的に検知して、影響のあるデバイスのアプリケーション・プログラムを再リンクします。

ライブラリ名は、LonBuilder ソフトウエアと一緒にインストールされる ASCII ファイルを経由してプロジェクト・マネージャに通知されます。

LonBuilder システム・ディレクトリにはいくつかのサブディレクトリがあります。IMAGES という名前のサブディレクトリには、Neuron リンカが使用するファイルが含まれています。IMAGES ディレクトリには、DEFAULT.VER と STDLIBS.LST という 2 つのファイルがあります。さらに IMAGES ディレクトリの下にも **VER*nnn***（***nnn*** はゼロで始まらない 2～255 の数字）という名前のサブディレクトリが作成されます。

前述の STDLIBS.LST ファイルには、LonBuilder プロジェクト・マネージャが使用するライブラリ名が書き込まれています。このファイルは LonBuilder エディタを使用して変更します。ライブラリ名は 1 行に 1 つずつ指定し、空白行があったり、ライブラリ名の先頭に空白があってはいけません。

ライブラリを追加するには、STDLIBS.LST ファイルを変更し、それを適切な **VER*nnn*** ディレクトリに配置します。1 つのライブラリからシステム・イメージごとに異なるバージョンを作成することも可能です。システム・イメージ・ファイルに対応している **VER*nnn*** ディレクトリは、デバイスをリンクするときにプロジェクト・マネージャがライブラリを検索する場所でもあります。ただし、アプリケーションがライブラリ・コードを全く必要としない場合、リンク時にライブラリ・ファイルが存在している必要はありません。

上で説明したようにライブラリ名をリスト・ファイルに追加すれば、そのライブラリのシンボルを参照する Neuron C プログラムがあると、リンク時に該当モジュールが自動的に読み込みます。これは、プログラムのリンク先デバイスが Neuron 3120xx であるか 3150 チップまたはスマート・トランシーバであるかには無関係です。

リンカは、システム・イメージ・ファイルにリンクして結合するオブジェクト・ファイルの中に未解決のシンボルがある場合にのみ、ライブラリを検索します。未解決のシンボル１つ１つに対して、ライブラリ・リスト内の各ライブラリが順にチェックされます。シンボルがライブラリ内に見つかると、対応するオブジェクト・ファイルがライブラリから抽出され、そのオブジェクト・ファイルに含まれているすべてのオブジェクトがリンクに追加されます。

ライブラリ内のオブジェクト・ファイルの中にも、他のシンボルが含まれていることがあります。この中に未定義のシンボルがあれば、このシンボルの参照も解決する必要があります。リンカは再び全ライブラリを検索し、抽出したオブジェクト・ファイルをリンクに追加します。このプロセスは、リンクに含まれる全てのシンボルが解決されるか、ライブラリ・リストを検索し終わるまで続きます。

ライブラリ・リストは、STDLIBS.LST ファイルに含まれているライブラリ名で構成されています。カスタム・ライブラリには、STDLIBS.LST ファイルに既に含まれている Echelon の標準ライブラリ名を使用しないでください。

オブジェクト・ファイルには複数のプロシージャやデータ項目が含まれていることがあります。オブジェクト・ファイル内のそのようなコンポーネントは、必要であるかどうかにかかわらず、すべてリンクされます。したがってライブラリを作成する際は、ライブラリに結合する前に互いに関連のないプロシージャやデータ項目を別の C ソース・ファイルに入れておくと効率がよくなります。

NodeBuilder によるライブラリの使用

付録 A「Neuron C ツールのスタンドアロンでの使用」で説明するとおり、NodeBuilder Neuron C のコンパイラ、アセンブラ、およびライブラリアンを使って「基本 C」ソース・ファイルから Neuron ライブラリを作成することができます。これらのライブラリをアプリケーション・プログラムの中で利用するには、NodeBuilder プロジェクト・マネージャの NodeBuilder デバイス・テンプレートを使用します。

NodeBuilder プロジェクト・マネージャは、ライブラリを構成しているソース・ファイルの変更を検知しないため、いつライブラリを再構築しなければならないかを判断できません。ライブラリを作成するときは、標準のソフトウエア・エンジニアリング技術を使用してソフトウエアを管理し、常に最新のソフトウエア・バージョンを使用するように考慮してください。サードパーティ製の「make」プログラムを使用すると、ソフトウエアの従属性を簡単に管理できます。

NodeBuilder ツールは、「基本 C」で書かれた Neuron オブジェクト・コードやデータから成るライブラリとアプリケーション・プログラムとのリンクを完全にサポートします。NodeBuilder ツールは、ライブラリを使用するときにすべての従属性をチェックし、ライブラリに変更があればそれを自動的に検知して、ビルド関数が呼ばれたときにデバイスのアプリケーション・プログラムを再リンクします。

ライブラリを NodeBuilder プロジェクトに認識させるには、ライブラリを Project ペインの Libraries フォルダに追加します。NodeBuilder プロジェクトやデバイス・テンプレート内でのライブラリの使用の詳細については、『NodeBuilder User's Guide』を参照してください。

ライブラリ内のシンボルを参照する Neuron C プログラムがあると、リンク時に該当モジュールが自動的に読み込まれます。

リンカは、システム・イメージ・ファイルにリンクして結合するオブジェクト・ファイルで、アプリケーション・プログラムが必要とするシンボルをすべて定義していない場合にのみ、ライブラリを検索します。各ライブラリは、リンカのコマンド・ラインで指定されている順にチェックされます。NodeBuilder プロジェクト作成ユーティリティは、ユーザ定義のライブラリを標準ライブラリの前に配置するため、標準ライブラリのシンボルよりも、ユーザ定義ライブラリに定義およびエクスポートされているシンボルが優先されます。この機能を使用すると、標準のシンボルが誤って上書きされてしまう危険があるため、注意してください。シンボルがライブラリ内に見つかると、対応するオブジェクト・ファイルがライブラリから抽出され、そのオブジェクト・ファイルに含まれているすべてのオブジェクトがリンクに追加されます。

ライブラリ内のオブジェクト・ファイルの中にも、他のシンボルが含まれていることがあります。この中に未定義のシンボルがあれば、このシンボルの参照も解決する必要があります。リンカは再び全ライブラリを検索し、抽出したオブジェクト・ファイルをリンクに追加します。このプロセスは、リンクに含まれる全てのシンボルが解決されるか、ライブラリ・リストを検索し終わるまで続きます。

ライブラリ使用の利点と欠点

ライブラリの使用には、次の各セクションで説明するように、利点と欠点があります。

ライブラリの利点

ユーティリティ・ルーチンや定数データ・テーブルをライブラリに入れて使用した場合、次のような利点があります。

1. ライブラリを使用すると、ユーティリティ・ルーチンを毎回再コンパイルする必要がないため、コンパイル時間が短くなります。
2. ライブラリには、モジュール化、カプセル化、再利用といった特長があります。これらのソフトウエア技術によって製品の品質を高め、開発コストを削減できます。
3. ライブラリには、関連する定数テーブルやプロシージャを一緒に入れておくことができます。ライブラリを正しく作成すれば、アプリケーションで使用されるモジュールだけがリンクされます。したがって、使用しないモジュールによってコード・スペースが消費される心配がありません。
4. ライブラリの中では、near RAM や EEPROM エリアを含むすべての Neuron チップまたはスマート・トランシーバのメモリ・スペースのデータ・オブジェクトを宣言できます。さらに、初期化済み RAM 変数をライブラリに入れておくことも可能です。初期化の規則は、Neuron C アプリケーション・プログラムの規則と同じです。

ライブラリの欠点

ライブラリの使用には以下のような欠点があります。

1. LonBuilder ツールも NodeBuilder ツールも、ライブラリの内容についてのデバッグはできません。（ただし、アプリケーション・プログラムの中でデータを「extern」として宣言してある場合は、ライブラリ内のデータ・オブジェクトの内容も、Neuron C デバッガを使って調べることができます。）ライブラリに登録する前に、各プロシージャを充分デバッグしておいてください。
2. LonBuilder および NodeBuilder のプロジェクト・マネージャでは、ライブラリ、およびそのコンポーネント・ファイルの従属性を管理することはできません。ライブラリの更新については、ユーザの責任で管理する必要があります。
3. ライブラリには、「基本 C」の関数およびデータ・オブジェクトだけが登録できます。ネットワーク変数、I/O オブジェクト、タイマー、**when** 文などの Neuron C 拡張機能を含めたり、参照したりすることはできません。『Neuron C Reference Guide』に記載されている関数のうち、ネットワーク変数、メッセージ、および入出力関連の関数以外は、「基本 C」関数としてライブラリ・モジュールからでも使用できます。Neuron C 組み込み変数にも同様の制約があります。この欠点を克服する手段については、本章で後述される「ライブラリから Neuron C 関数を実行する」のセクションを参照してください。
4. アプリケーションにリンクされたライブラリ・オブジェクトは、そのアプリケーションの一部となります。アプリケーションに変更があると、ライブラリ・オブジェクトはアプリケーションに再リンクされるので、結果としてオブジェクトの位置に変更が生じる可能性があります。したがって、アプリケーションが再リンクされる度に、ライブラリ・オブジェクトをデバイスのメ

モリに再ロードする必要があります。この作業は、ネットワークを介して、または不揮発性メモリ・デバイスをプログラムすることによって実行できます。

ライブラリアンを使ったライブラリの作成

ライブラリを作成するには、付録 A「Neuron C ツールのスタンドアロンでの使用」で説明しているスタンドアロン・バージョンの Neuron C ツールを使用します。ライブラリを構成する「基本 C」ソース・ファイルをコンパイルし、アセンブルするには、スタンドアロン型のコンパイラとアセンブラが必要です。このコンパイルとアセンブルのプロセスにより、各「基本 C」のソース・ファイルに対応する Neuron オブジェクト・ファイル（拡張子は.no）が生成されます。次に **nlib.exe** Neuron ライブラリアンを使用して Neuron オブジェクト・ファイルを結合し、Neuron リンカが使用できるライブラリを生成します。

ライブラリを作成する際には、次のガイドラインに従ってください。

1 「基本 C」ファイルに対応するライブラリ・モジュール内のシンボルが 1 つでも参照された場合は、そのモジュール全体がリンクに含まれます。このため、互いに関連のない関数は別にして、アプリケーション・プログラムがライブラリのコンポーネントを使用するときのプログラムの使用領域を最小限に抑えるようにしてください。

例えば、**strcpy()**、**strcat()**、**strlen()**関数を含んだ文字列ライブラリを作成する場合、アプリケーション・プログラムがこれらの関数全部をリンクするとは限らないため、各関数をそれぞれ別のファイルに配置して、コード領域を最小化します。この 3 つの関数を同じファイルに配置すると、このうちどれか 1 つでも使用されたときにすべての関数がリンクされることになります。

2 ライブラリ・モジュール内の関数やデータ項目をアプリケーション・プログラムまたはその他のライブラリ・モジュールから隠蔽したい場合は、**static** キーワードで宣言します。こうすると、シンボルを効果的に隠蔽することができます。

3 インクルード・ファイルには **extern** 関数プロトタイプおよび **extern** データ宣言を使用して、ライブラリのユーザが各自のプログラムに取り込むことができるようにします。コンパイラが正しい呼び出し順を確立し、ライブラリ内のデータ・オブジェクトや関数をリンクするための適切なアセンブラ・コマンドを使用するには、各宣言に **extern** キーワードが必要です。

さらに、関数やデータ・オブジェクトを定義するライブラリ・ソース・ファイルにも、これらのインクルード・ファイルを含めておく必要があります。こうすると、Neuron C コンパイラに対して、**extern** 宣言およびプロトタイプが実際の宣言および関数定義と一致していることを明確に伝えることができます。宣言の内容が一致していれば、**extern** 宣言の後に実際の宣言を行っても問題ありません。この方法を利用すれば、誤った **extern** プロトタイプなどが原因で、誤ったパラメータを使用して関数を呼び出すようなことがなくなります。パラメータが誤っていると、データ・スタックが上書きされる恐れがあります。その結果、デバイスのウォッチドッグ・タイマーが繰り返しリセットされたり、何度も変数が上書きされたり、その他のソフトウエアの障害が発生する可能性があります。

4 LonBuilder プロジェクト・マネージャは、リンカが STDLIBS.LST ファイル内のライブラリをすべて使用するように常に指示します。NodeBuilder プロジェ

クト作成ユーティリティも、STDLIBS.LST ファイル内のすべてのライブラリを使用するようリンカに指示しますが、NodeBuilder デバイス・テンプレートで指定されているユーザ定義ライブラリのリストの方が優先されます。このため、ライブラリ間でシンボルが競合しないように、ライブラリ・オブジェクトの名前が確実に一意になるような名前付けの規則を決めておいてください。**nlib -r** コマンド・オプション（付録 A を参照）を使用すると、既存の各ライブラリで定義されているシンボルのリストが生成されます。

ライブラリから Neuron C 関数を実行する

ライブラリに登録する「基本 C」コードは、ネットワーク変数、メッセージ、I/O、タイマーといった Neuron C オブジェクトを参照できません。しかし、標準 I/O デバイス管理、メッセージ作成、タイマー操作などの Neuron C 関連の機能をライブラリに含めることはできます。

ライブラリ関数から Neuron C オブジェクトにアクセスするには、アプリケーション・プログラムからアプリケーション関数を呼び出して Neuron C の操作を実行するようにします。ライブラリは、Neuron C アプリケーション・プログラムの中にあるアプリケーション関数を呼び出すことによって、Neuron C 操作を効果的に実行できます。

例えば、標準 LCD ディスプレイの管理ルーチンのライブラリがあるとします。このライブラリには、情報をフォーマットするルーチンや、アプリケーション・プログラムからのコマンドに応答してディスプレイを管理するルーチンなどが含まれます。デバイスを更新するための I/O 操作は、ライブラリ・コードの中で自動的に実行できればいいのですが、「基本 C」の制限のために、必要な Neuron C コードをライブラリの一部として実装することはできません。

ここで、ディスプレイの I/O 操作は、Neurowire デバイス・インターフェースを使用すると仮定します。ライブラリには、Neuron C アプリケーション・プログラムに必要なインクルード・ファイルを含めておきます。インクルード・ファイルの中には、ディスプレイの I/O 操作に必要な Neurowire I/O 宣言や関数定義などが含まれるはずです。このようにしておけば、アプリケーション・プログラムからの指示がなくても、ライブラリが必要に応じて各関数にアクセスできます。またアプリケーション・プログラムも、ライブラリのディスプレイ管理ソフトウエアに対して必要なパラメータを引き渡す必要がなくなります。

ここで説明したように、Neuron C 機能に依存する部分にはユーティリティ関数を作成すると便利です。Neuron C コードと「基本 C」コードとを区別し、Neuron C コードの方をユーティリティ用のインクルード・ファイルに配置すると、ユーティリティはそれがカプセル化されたユーティリティであるかのように効果的に機能します。

付録C

Neuron C
カスタム・システム・イメージ

付録 C では、Neuron C ツールを使って独自のカスタム・システム・イメージを作成し、使用する方法について説明します。カスタム・システム・イメージの作成・使用には、LonBuilder 開発ツールと NodeBuilder 開発ツールのどちらも使用できます。

定義

アプリケーション・プログラム

コンパイルとアセンブルの後、システム・ファームウエア・イメージにリンクされる Neuron C のソース・プログラム。アプリケーション・プログラムは、スタンドアロン型の実行プログラムではありません。アプリケーション・プログラムはシステム・イメージへの外部参照を含んでいるため、対応するファームウエア・イメージが含まれているデバイスのメモリにロードして使用します。

カスタム・システム・イメージ

Neuron アセンブラを使って作成した「基本 C」オブジェクト・ファイルを、標準システム・イメージ（または別のカスタム・システム・イメージ）に結合して生成したファイルの集合。カスタム・システム・イメージに追加されたコードやデータ・オブジェクトは、カスタム・イメージにリンクされたアプリケーション・プログラムから参照可能です。カスタム・システム・イメージの作成には、Neuron リンカを使用できます。

ライブラリ

Neuron アセンブラを使って作成したオブジェクト・ファイルから構成されるファイルの集合。Neuron リンカは、これらのオブジェクト・ファイルを必要に応じてライブラリから抽出し、Neuron C アプリケーション・プログラムにリンクします。

基本 C

Neuron C 言語は、ANSI 標準 C 言語のサブセットに拡張を加えた言語セットです。本書の中の「基本 C」という用語は、Neuron C の拡張部分を除いた C 言語サブセットを指します。

スタンドアロン型ツール

「スタンドアロン」とは、ツールが Windows のコマンド・プロンプトから利用でき、プロジェクト・マネージャの環境外で使用できることを意味します。この付録で説明するツールはすべてスタンドアロン方式で実行できます。

標準システム・イメージ

LonBuilder および NodeBuilder ソフトウエアに含まれている、Neuron ファームウエア・ファイルの集合。現在、標準システム・イメージとして定義されているファイルは以下のとおりです。

BIN3120： Neuron 3120 Chip に搭載されているファームウエア。
B3120E1： Neuron 3120E1 Chip に搭載されているファームウエア。
B3120E2： Neuron 3120E2 Chip に搭載されているファームウエア。
B3120E3： Neuron 3120E3 Chip に搭載されているファームウエア。
B3120E4： Neuron 3120E4 Chip または FT 3120 スマート・トランシーバに搭載されているファームウエア

B3120E5： 3120E5 Chip に搭載されているファームウエア。
B3120A20：Neuron 3120A20 Chip に搭載されているファームウエア。

E3120E1： エミュレータのみ。Neuron 3120E1 Chip をエミュレートする。
E3120E2： エミュレータのみ。Neuron 3120E2 Chip をエミュレートする。
E3120E3： エミュレータのみ。Neuron 3120E3 Chip をエミュレートする。
E3120E4： エミュレータのみ。Neuron 3120E4 Chip
または FT 3120 スマート・トランシーバをエミュレートする。
E3120E5： エミュレータのみ。Neuron 3120E5 Chip をエミュレートする。
E3120A20：エミュレータのみ。Neuron 3120A20 Chip をエミュレートする。

EMU3120：エミュレータのみ。Neuron 3120 Chip をエミュレートする。
EMU3150：エミュレータのみ。Neuron 3150 Chip
および FT 3120 スマート・トランシーバをエミュレートする。
SYS3150： Neuron 3150 Chip または FT 3120 スマート・トランシーバ
を使用している全デバイス用。

システム・イメージ　Neuron チップまたはスマート・トランシーバの動作および LonTalk プロトコルの実装に必要な Neuron ファームウエアを含んだファイルの集合。このイメージ内のソフトウエアは、リンク済みの実行プログラムです。外部参照を含めることはできません。

LonBuilder によるカスタム・システム・イメージのサポート

本章で後述する「カスタム・システム・イメージの作成」で説明するとおり、LonBuilder Neuron C のコンパイラ、アセンブラ、およびライブラリアンを使ってカスタム・システム・イメージを作成することができます。これらのカスタム・システム・イメージをアプリケーション・プログラムの中で利用するには、LonBuilder プロジェクト・マネージャを使用します。

LonBuilder プロジェクト・マネージャは、カスタム・システム・イメージを構成しているソース・ファイルの変更を検知しないため、いつカスタム・システム・イメージを再構築しなければならないかを判断できません。カスタム・システム・イメージを作成するときは、標準のソフトウエア・エンジニアリング技術を使用してソフトウエアを管理し、常に最新のソフトウエア・バージョンを使用するように考慮してください。サードパーティ製の「make」プログラムを使用すると、ソフトウエアの従属性を簡単に管理できます。

LonBuilder プロジェクト・マネージャは、カスタム・システム・イメージを標準システム・イメージとほぼ同様に扱います。プロジェクト・マネージャは、システム・イメージに関する従属性をチェックし、システム・イメージに変更があればそれを自動的に検知して、影響のあるデバイスを再リンクします。

プロジェクト・マネージャがデバイスを作成する際、デバイス用のシステム・イメージ情報を読み取り、必要な関連ハードウエアの仕様をチェックします。**Target HW**（ターゲット・ハードウエア）エントリには、デバイスのハードウエアの仕様が表示されます。さらにハードウエア・ウィンドウには **HW Prop. Name**（ハードウエア仕様名）エントリがあり、このハードウエア仕様が使用するハードウエア仕様名が表示されます。

ハードウエア特性ウィンドウには、「**NEURON Chip Firmware**」と「**Firmware Version**」という 2 つのエントリがあります。これらの情報は、プロジェクト・

マネージャが、どのシステム・イメージをデバイスが使用しているかを調べるために使用します。

NEURON Chip Firmware エントリに文字列が入力されていれば、この文字列がシステム・イメージの名前（拡張子なし）として使用されます。カスタム・システム・イメージ・ファイルはここで指定します。このフィールドが空白であれば、プロジェクト・マネージャが適切な標準システム・イメージ名（前述の「定義」のセクションを参照）を割り当てます。

Firmware Version エントリでは、システム・イメージのバージョン番号を指定します。システム・イメージ名を入力したかどうかに関係なく、バージョン番号を指定することができます。**Firmware Version** がゼロの場合、プロジェクト・マネージャは、IMAGES ディレクトリにある DEFAULT.VER ファイルに書き込まれているデフォルト値からバージョン番号を調べます。ゼロ以外のときは、**Firmware Version** で指定したバージョン番号が使われます。

DEFAULT.VER ファイルには、ファームウエア・イメージ・ファイルの名前と番号が対になって保存されています。プロジェクト・マネージャはこのリストを参照して、各イメージ名に対応したデフォルト番号を選択します。イメージ名に対応したレコードがなければ、デフォルトとして、イメージ名の代わりに「*」で始まる行のバージョン番号を使用します。詳しくは、『LonBuilder User's Guide』の第 7 章「Using Multiple Firmware Versions」を参照してください。

プロジェクト・マネージャは、上記で説明した規則に従ってシステム・イメージ名とバージョン番号を決定し、これに対応するイメージをリンクするようにリンカに指示します。イメージ・ファイルは、IMAGES ディレクトリ内の **Ver*nnn*** サブディレクトリに保存してください。ここで、***nnn*** はイメージのバージョン番号に対応します。

LonBuilder ツールでカスタム・システム・イメージを使用するときは、次のガイドラインに従ってください。

1 LonBuilder ハードウエア管理ソフトは、ファームウエア・イメージの従属性をチェックしません。したがって、システム・イメージ・ファイルを変更しても、**Navigator Target HW** ウィンドウの **To-Do** には、インストールされていることを示す「Install」とは表示されません。同様にプロジェクト・マネージャも、ハードウエアにインストールされたシステム・イメージが正しいかどうかのチェックは行いません。ファームウエア・イメージ・ファイルを 1 つでも変更したら、必ず関連のあるデバイスを手動でインストールしてください。「Automatic Install」を実行しただけでは、LonBuilder デバイスの電源を入れ直さない限り、不十分です。「Install All」を実行するか、影響のあるデバイスをすべて手動でインストールするのが最も安全な方法です。エミュレータに対する「Install」（またはカスタム・デバイス ROM の再プログラム）を実行しないと、システムに不一致が生じる可能性があります。その結果、「保護メモリ領域への書き込み試行」や「ウォッチドッグ・タイマーのタイムアウト」などの診断エラーが発生して、デバイスのプログラムが異常終了することになります。ただし該当のイメージ・ファイルが変更されたときには、プロジェクト・マネージャの状態として、Target HW ウィンドウの To-Do に「Build」と表示されます。

2 システム・イメージを構成しているファイルはまとめて保持しておく必要があります。システム・イメージ・ファイルを変更するときは、これら全ファイルを一緒に更新してください。更新されていないファイルがあると、前の項目と同様のエラーが発生します。システム・イメージを構成しているファ

イルを以下に示します。

image.nx	Intel 16 進形式のロード可能イメージ
image.nxb	バイナリ形式の *image*.nx
image.sym	シンボル・テーブル・ファイル
image.ib	エミュレータ・イメージ・ビットマップ（LonBuilder エミュレータのみ）

3 LonBuilder Neuron Emulator ハードウエアも、システム・イメージ・ファイルを使用していますが、カスタム・デバイスとは異なるファイルを使用します。カスタム・システム・イメージを使用するときには、2 種類のファイル（エミュレータ用とカスタム・デバイス用）を同時に作成するようにしてください。

NodeBuilder によるカスタム・システム・イメージの使用

本章で後述する「カスタム・システム・イメージの作成」で説明するとおり、NodeBuilder の Neuron C コンパイラ、アセンブラ、およびライブラリアンを使ってカスタム・システム・イメージを作成することができます。これらのカスタム・システム・イメージをアプリケーション・プログラムの中で利用するには、NodeBuilder プロジェクト・マネージャの NodeBuilder デバイス・テンプレートを使用します。

NodeBuilder プロジェクト・マネージャは、カスタム・システム・イメージを構成しているソース・ファイルの変更を検知しないため、いつカスタム・システム・イメージを再構築しなければならないかを判断できません。カスタム・システム・イメージを作成するときは、標準のソフトウエア・エンジニアリング技術を使用してソフトウエアを管理し、常に最新のソフトウエア・バージョンを使用するように考慮してください。サードパーティ製の「make」プログラムを使用すると、ソフトウエアの従属性を簡単に管理できます。

NodeBuilder プロジェクト・マネージャは、カスタム・システム・イメージを標準システム・イメージとほぼ同様に扱います。NodeBuilder プロジェクト・マネージャは、カスタム・システム・イメージに関する従属性をチェックし、システム・イメージに変更があればそれを自動的に検知して、影響のあるデバイスを再リンクします。

デバイス・ターゲット用のファームウエア・イメージの選択と、カスタム・ファームウエア・イメージの使用の詳細については、『NodeBuilder User's Guide』と NodeBuilder のヘルプ・ファイルを参照してください。

NodeBuilder ツールでカスタム・システム・イメージを使用するときは、次のガイドラインに従ってください。

1 新しいバージョンのカスタム・システム・イメージを作成するときは、カスタム・システム・イメージを使用してカスタム・デバイス PROM を再プログラムします。PROM を再プログラムしないと、PROM バージョンと NodeBuilder バージョンのシステム・イメージに不一致が生じる可能性があります。その結果、ウォッチドッグ・タイマーのタイムアウトや診断エラーなどが起こり、デバイスのプログラムが異常終了することになります。

2 システム・イメージを構成しているファイルはまとめて保持しておく必要があります。例えば、システム・イメージ・ファイルを適切な **Ver*nnn*** ディレクトリにコピーするときには、これらをすべてまとめてコピーします。シス

テム・イメージ・ファイルを変更するときは、これら全ファイルを一緒に更新してください。更新されていないファイルがあると、前の項目と同様のエラーが発生します。システム・イメージを構成しているファイルを以下に示します。

image.nx	Intel16 進形式のロード可能イメージ
image.nxb	バイナリ形式の *image*.nx
image.sym	シンボル・テーブル・ファイル

カスタム・システム・イメージ使用の利点と欠点

開発者にとって、カスタム・システム・イメージの使用には、利点もあれば欠点もあります。ライブラリを使用する方が適切な場合もあれば、カスタム・システム・イメージを使用する方が適切な場合もあります。カスタム・システム・イメージの使用に関する利点と欠点については、このセクションと付録 B「Neuron C 関数ライブラリ」で説明します。

カスタム・システム・イメージの利点

ユーティリティ・ルーチンや定数データ・テーブルをカスタム・システム・イメージに入れて使用した場合、次のような利点があります。

1. カスタム・イメージを使用すると、ユーティリティ・ルーチンを毎回再コンパイルする必要がないため、コンパイル時間が短くなります。
2. カスタム・イメージには、モジュール化、カプセル化、再利用といった特長があります。これらのソフトウエア技術によって製品の品質を高め、開発コストを削減できます。
3. いったんカスタム・イメージを作成しておけば、それを複数のアプリケーション・プログラムで利用できます。例えば、Neuron 3150 チップや FT 3150 スマート・トランシーバで ROM やフラッシュ・メモリにカスタム・イメージをプログラムしておいて、後日、このカスタム・イメージを使用するアプリケーション・プログラムを EEPROM やフラッシュ・メモリにネットワーク転送するといったことが可能になります。これは、Neuron C フィールド・コンパイラのユーザが、アプリケーション I/O ハードウエアのサポートを組み込んだカスタム・システム・イメージを乗せたデバイスを配布する場合に使われる方法です。このようにすると、追加機能を標準関数として提供することになるため、エンド・ユーザがアプリケーション・コードから呼び出すことができるようになります。

カスタム・システム・イメージの欠点

カスタム・システム・イメージの使用には以下のような欠点があります。

1. LonBuilder ツールも NodeBuilder ツールも、システム・イメージの内容についてのデバッグはできません（ただし、アプリケーション・プログラムの中でデータを「**extern**」として宣言してある場合は、システム・イメージ内のデータ・オブジェクトの内容も、Neuron C デバッガを使って調べることができます）。カスタム・システム・イメージ内に組み込む前に、各プロシージャを十分デバッグしておいてください。デバッガの関数は、カスタム・システム・イメージを使用するアプリケーション・プログラムには影響を受けません。

2 LonBuilder および NodeBuilder のプロジェクト・マネージャでは、カスタム・システム・イメージ、およびそのコンポーネント・ファイルの従属性を管理することはできません。カスタム・システム・イメージの更新については、ユーザの責任で管理する必要があります。

3 カスタム・システム・イメージには、「基本 C」の関数およびデータ・オブジェクトだけを登録できます。ネットワーク変数、I/O オブジェクト、タイマー、**when** 文などの Neuron C 拡張機能は使用できません。『Neuron C Reference Guide』に記載されている関数のうち、ネットワーク変数、メッセージ、および入出力関連の関数以外は、「基本 C」関数として使用できます。Neuron C 組み込み変数にも同様の制約があります。

4 カスタム・システム・イメージは、Neuron 3120xx チップには使用できません。

5 カスタム・システム・イメージには多くのユーティリティ・ルーチンやデータ・テーブルを含めることができます。アプリケーション・プログラムによっては必要ないものも含まれており、それだけメモリ・スペースが無駄に使用されることになります。

6 カスタム・システム・イメージは、未解決のシンボルへの参照を含むことができません。カスタム・システム・イメージ内のプロシージャやデータ・オブジェクトは、同じイメージ内のプロシージャやオブジェクトか、ベース・イメージを構成しているプロシージャやオブジェクトしか参照できません（通常、ベース・イメージは、標準システム・イメージのうちの 1 つであるか、標準システム・イメージを基にした別のカスタム・システム・イメージです。カスタム・システム・イメージは別のカスタム・システム・イメージの上にも作成できます）。

7 カスタム・システム・イメージには、Neuron チップの RAM メモリの使用に関する制限があるうえ、EEPROM メモリも使用できません。関数および定数データ・オブジェクトは、ROM またはフラッシュ・メモリにしか配置できません。**far** RAM の一部だけが使用可能で、使用できるのは（カスタム・システム・イメージ全体で）合計 64 バイトです。**near** RAM や **near** EEPROM、または **far** EEPROM を宣言したり、直接使用することはできません。また、カスタム・システム・イメージには、初期化済みの RAM 変数が含まれている可能性があることに注意してください。初期化の規則は Neuron C アプリケーション・プログラムの規則と同じです。

8 システム・イメージを作成するプロセスは複雑で、いろいろな詳細規則があります。例えば、全部のファイルを同時に更新管理する、各デバイス上のイメージとイメージ・バージョン・ファイルを同期して管理するなどです。LonBuilder と NodeBuilder プロジェクト・マネージャには、これらの管理機能がありません。そのため、カスタム・システム・イメージを作成する際には間違いが起こりやすく、間違いであることが発見しにくいことがあります。

カスタム・システム・イメージの作成

カスタム・システム・イメージを作成するには、付録 A「Neuron C ツールのスタンドアロンでの使用」で説明するとおり、スタンドアロン・バージョンの Neuron C コンパイラ、Neuron アセンブラ、および Neuron リンカの各ツールを使用します。システム・イメージのカスタム部分を構成する「基本 C」ソース・ファイルをコンパイルしてアセンブルするには、スタンドアロン型のコンパイラとアセンブラが必要です。このコンパイルとアセンブルのプロセスにより、各「基本 C」ソース・ファイルに対応する Neuron オブジェクト・ファイルが生成されます。次に Neuron リンカ **nld.exe** を使用して Neuron オブジェクト・ファイルを結合し、カスタム・システム・イメージを作成します。

標準システム・イメージを新しいカスタム・システム・イメージのベースとして使用することも、以前に作成した別のカスタム・システム・イメージを使用することもできます。

カスタム・システム・イメージを作成するときは、次の手順に従ってください。

1 付録 A で説明しているスタンドアロンの Neuron リンカを実行します。このリンカ・スイッチの他に、カスタム・システム・イメージ作成用の特別なスイッチもあります。これらのスイッチについて、以下に説明します。

- **-c** スイッチを指定します。これはリンカに指定する最初のスイッチになる必要があります。**-c** スイッチには引数はありません。

- ベース・システム・イメージは、**-b** スイッチを使用して指定します。**-b** スイッチの後にはベース・イメージのファイル名またはパス名（拡張子なし）を指定します。LonBuilder エミュレータのカスタム・システム・イメージを作成する場合は、Emu3150 標準システム・イメージ、または Emu3150 イメージに基づいたカスタム・システム・イメージを指定します。LonBuilder エミュレータのカスタム・システム・イメージを作成しない場合は、Sys3150 システム・イメージ、または Sys3150 システム・イメージに基づいたカスタム・システム・イメージを指定します。例えば、Neuron 3150 チップ・ベースのカスタム・デバイスとして標準のバージョン 13 システム・イメージを使用するには、次のようにスイッチを指定します。

  -b ¥LonWorks¥Images¥Ver13¥SYS3150

- カスタム・システム・イメージが使用できるのは、ROM および far オンチップ RAM の一部のみです。したがって、カスタム・システム・イメージの作成に必要なメモリ・マップ・スイッチは、**-Z** スイッチだけです。リンカがカスタム・システム・イメージを作成するとき、カスタム・システム・イメージ用に予約されている ROM の終わりを **-Z** スイッチに指定します。ベース・イメージ用には、0x0000 番地からある番地 *x* までの ROM エリアが予約されています（標準システム・イメージでは *x* は 0x3FFF）。

  -Z スイッチの後に値 y_p を指定すると、ROM の *x+1* から *y* までのエリアが作成中のカスタム・システム・イメージ用に予約されます。y_p は *y* 番地に該当するページ番号です。例えば、作成中のカスタム・システム・イメージに 0x4000 から 0x4FFF までの番地を予約するための**-Z** スイッチの値は「4F」になります。予約エリアには、少なくともイメージ作成に必要なオブジェクト・ファイル分の大きさが必要です。**-Z** スイッチの最小値は「40」で、これは標準システム・イメージからちょうど 1 ページ（256 バイト）分の追加 ROM 領域を使うという指示になります。

  新しく予約したエリア内の未使用の領域は、このカスタム・システム・イメージの将来のバージョンのために確保されます。リンカが後でこのカスタム・システム・イメージを使用して Neuron C アプリケーション・プログラムをリンクしようとすると、新しく予約された ROM エリアが自動的にリンカに知らされます。アプリケーション・プログラムは、予約されていない ROM の残りの部分のみを使用できます。

- **-V** スイッチを指定して、128～255 の範囲でカスタム・システム・イメージにバージョン番号を割り当てます。**-V** スイッチの後には希望のバージョン番号を 10 進数で指定します。作成したカスタム・システム・イメージ・ファイルは、IMAGES ディレクトリにある、選択したバージョン番号に対応する **VER*nnn*** サブディレクトリに移動します。該当する **VER*nnn*** ディレクトリがない場合は、ディレクトリを新たに作成します。

- エミュレータのカスタム・システム・イメージを作成している場合は、**-i** スイッチを指定します。これによって、入力ベース・イメージ内の追加ファイル（拡張子は.ib）を参照し、出力カスタム・システム・イメージ内に追加ファイル（この拡張子も.ib）を出力するようリンカに指示します。この追加ファイルはエミュレータがブレークポイントの管理、エラーのトラップなどを行うために使用します。

次に示すコマンドを入力すると、SYS3150 標準システム・イメージをベースにして、Objs.lst スクリプト・ファイルに含まれているオブジェクト・ファイルを組み込んだカスタム・システム・イメージが作成されます。出力ファイルは、"-o"スイッチに指定されている「myimage.*」ファイルになります。

```
nld –c –b c:¥LonWorks¥Images¥Ver13¥sys3150 –t 3150 –Z 4F –V 128 –o myimage -@objs.lst
```

上記の方法でリンカを使用すると、次のカスタム・システム・イメージ・ファイルが作成されます。

- myimage.sym ファイルには、エクスポートしたイメージのシンボルが含まれています。この情報は、アプリケーション・プログラムをイメージにリンクする際に使用されます。

- myimage.nx ファイルはシステム・イメージを含んだ Intel 16 進形式のファイルです。

- myimage.map ファイルには（作成を指定した場合）、イメージのリンク・マップとレポートが含まれます。

2 テキストのシステム・イメージ・ファイルをバイナリ・イメージに変換します。これを行うには、Echelon ディベロッパーのツールボックス（www.echelon.com/toolbox）から **nxcvt.exe** イメージ変換ユーティリティをダウンロードし、このユーティリティを使用してファイルを変換します。例えば、前述の例の myimage.nx ファイルをバイナリ形式の myimage.nxb ファイルに変換するには、次のコマンド・ラインを使用してイメージ名（拡張子なし）を指定します。

nxcvt myimage

3 システム・イメージ・ファイルは、ステップ 1 で指定したバージョン番号に対応する **Ver*nnn*** ディレクトリに移動します。

カスタム・システム・イメージを作成するには、次のガイドラインに従ってください。

1 ライブラリを使ってカスタム・システム・イメージを作成することもできますが、ライブラリを参照するだけであり、ライブラリ全体の内容がカスタム・システム・イメージに自動的に転送されるわけではありません。カスタム・システム・イメージの作成にライブラリを使用するには、ライブラリ内で定義されているグローバル変数をどれも **far**（RAM）変数として宣言しておく必要があります。

2 カスタム・システム・イメージから PROM を直接プログラムすることはできません。PROM をプログラムするには、LonBuilder または NodeBuilder ツールを使用して、カスタム・システム・イメージに基づいた ROM イメージ・ファイルを作成する必要があります。カスタム・システム・イメージを PROM に読み込むことだけが目的の場合は、空のアプリケーションを使用してイメージを作成します。

3 **static** キーワードを使用して、アプリケーション・プログラムから隠蔽する関数やデータ項目を宣言します。これによって、シンボルが効果的に隠蔽されます。

4 インクルード・ファイルには **extern** 関数プロトタイプおよび **extern** データ宣言を使用して、カスタム・システム・イメージのユーザが各自のプログラムに取り込むことができるようにします。コンパイラが正しい呼び出し順を確立し、カスタム・システム・イメージ内のデータ・オブジェクトや関数をリンクするための適切なアセンブラ・コマンドを使用するには、各宣言に **extern** キーワードが必要です。

さらに、関数やデータ・オブジェクトを定義するカスタム・システム・イメージ・ソース・ファイルにも、これらのインクルード・ファイルを含めておく必要があります。こうすると、Neuron C コンパイラに対して、**extern** 宣言およびプロトタイプが実際の宣言および関数定義と一致していることを明確に伝えることができます。宣言の内容が一致していれば、**extern** 宣言の後に実際の宣言を行っても問題ありません。この方法を利用すれば、誤った **extern** プロトタイプなどが原因で、誤ったパラメータを使用して関数を呼び出すようなことがなくなります。パラメータが誤っていると、データ・スタックが上書きされる恐れがあります。その結果、デバイスのウォッチドッグ・タイマーが繰り返しリセットされたり、何度も変数が上書きされたり、その他のソフトウエアの障害が発生する可能性があります。

5 カスタム・システム・イメージが使用できる RAM は最大 64 バイトです。あるカスタム・システム・イメージをベースにして別のカスタム・イメージを作成した場合、必要な RAM は累積されていくため、2 つ（またはそれ以上）のイメージが使用する RAM が合計で 64 バイト以下にする必要があります。この 64 バイト・エリア内で未使用の RAM は、リンク後にアプリケーション・プログラムが使用できます。カスタム・システム・イメージは far RAM 変数しか使用できません。これらの変数の **extern** 宣言には必ず **far** キーワードを指定してください（上記の項目 4 を参照）。

6 カスタム・システム・イメージから EEPROM 変数を使用することはできません。

広い RAM 領域の割り当て

リンク時にカスタム・システム・イメージに使用できる RAM 領域の総量は 64 バイトです。ただし、カスタム・システム・イメージが使用する関数によっては、これよりも多くの RAM を必要とする場合があります。このようなときには、カスタム・システム・イメージ用の大きな RAM ブロックをアプリケーション・プログラムで宣言します。アプリケーション・プログラムはリセット時にこれをカスタム・システム・イメージに知らせることができます。この機能は、カスタム・イメージに関連付けられているインクルード・ファイル内のリセット・ルーチンに配置できます。カスタム・イメージを使用しているアプリケーションは、これによってインクルード・ファイルを使用し、**when(reset)**タスクからリセット・ルーチンを呼び出すことができます。

RAM を必要とするカスタム・システム・イメージ関数が呼び出されるたびに、この RAM ブロックへのポインタをパラメータとして引き渡すようにします。さらに効率を高めるには、カスタム・システム・イメージの RAM 領域を 2 バイト使ってメモリ・ブロックへのグローバル・ポインタを宣言し、NULL に初期化しておきます。アプリケーション・プログラムがリセットされたときには、グローバル・ポインタ変数が適切なメモリ・ブロックを指すようにアプリケーション・プログラムを設定するか、少なくともこのポインタを指定して初期化関数を呼び出すようにします。

Neuron C 関数の実行

カスタム・システム・イメージに組み込む「基本 C」コードは、ネットワーク変数、メッセージ、I/O、タイマーといった Neuron C オブジェクトを参照できません。しかし、標準 I/O デバイス管理、メッセージ作成、タイマー操作などの Neuron C 関連の機能をカスタム・システム・イメージに含めることができます。

カスタム・システム・イメージ関数から Neuron C オブジェクトにアクセスするには、アプリケーション・プログラムからアプリケーション関数を呼び出して Neuron C 操作を実行するようにします。アプリケーション関数へのポインタに設定する RAM 変数は、カスタム・システム・イメージから宣言できます。カスタム・システム・イメージは、Neuron C アプリケーション・プログラムの中にある関数を呼び出すことによって、Neuron C 操作を効果的に実行できます。

例えば、標準 LCD ディスプレイの管理ルーチンを含んだカスタム・システム・イメージがあるとします。このカスタム・システム・イメージには、情報をフォーマットするルーチンや、アプリケーション・プログラムからのコマンドに応答してディスプレイを管理するルーチンなどが含まれています。デバイスを更新するための I/O 操作は、カスタム・システム・イメージ・コードの中で自動的に実行されるようになっているのが理想的です。また、このようなカスタム・システム・イメージは RAM 内の大きなバッファにアクセスする必要が生じることがあります。しかし、これらは「基本 C」の制限とカスタム・システム・イメージの RAM メモリの制限のために、カスタム・システム・イメージの中だけでは実装できません。アプリケーションでは、前述のように RAM バッファを使用することができます。

ここで、ディスプレイの I/O 操作に Neurowire デバイス・インターフェースを使用すると仮定します。カスタム・システム・イメージには、Neuron C アプリケーション・プログラムに必要なインクルード・ファイルを含めておきます。インクルード・ファイルの中には、ディスプレイの I/O 操作に必要な Neurowire I/O 宣言や関数定義などが含まれるはずです。さらに、関数へのポインタやメモリ・バッファへのポインタを初期化するための、アプリケーションの **when (reset)**タスクから呼び出されるルーチンを入れておくこともできます。このようにしておけば、アプリケーション・プログラムからの指示がなくても、カスタム・システム・イメージが必要に応じて関数やメモリ・バッファにアクセスできます。また、アプリケーション・プログラムも、カスタム・システム・イメージのディスプレイ管理ソフトウエアと対話が必要になるたびに特別なパラメータを引き渡す必要がなくなります。

カスタム・システム・イメージにはリンカによって作成されたときに未解決の外部記号を含めることができないため、このようなポインタによる関数への間接アクセスが必要になります。ただし、ディスプレイ管理ソフトウエア

をライブラリとして使用した場合には、前に説明したようにこのような I/O 関数の名前を指定するだけでライブラリからアクセスできるので、関数へのポインタを設定し直す必要はありません。

索引

C

D

E

F

G

I

L

M

N

O

P

Q

R

S

T

U

V

W

あ

い

う

え

お

か

き

く

け

こ

さ

し

す

せ

そ

た

ち

て

と

に

ね

へ

ほ

ま

み

め