目次

1 IEC 61131-3 の歴史

IEC 61131-3 は、産業用オートメーションのための、ベンダに依存しない最初の標準プログラミング言語です。この言語は、世界中で認められる標準や規格を策定する国際電気標準会議 (IEC) によって確立されました。

歴史: IEC は 1906 年に創設されました。当初はロンドンを拠点としていましたが、1948 年に本部をジュネーブに移しました。IEC は本来、計量単位、特にガウス、ヘルツ、およびウェーバーの規格を調整することに関与していました。1938 年、IEC は電気技術用語を標準化するために、多国語の国際辞書を出版しました。その業務は今日まで続いており、国際電気標準用語集は、電気および電子業界において重要な役割を果たしています。

この規格の構成は次の通りです。

第 1 部:	一般的な情報; 現在のバージョンは 3.2004
第 2 部:	装置の要件とテスト; 現在のバージョンは 2.2004
第 3 部:	プログラミング言語; 現在のバージョンは 12.2003
第 3 部 付属書 1:	プログラマブルロジックコントローラ用プログラミング言語のアプリケーションと実装のガイドライン; 現在のバージョンは 4.2005
第 5 部:	通信; 現在のバージョンは 11.2001
第 7 部:	ファジー制御プログラミング; 現在のバージョンは 11.2001

IEC 61131-3 は産業およびプロセス制御のシステムとして、ヨーロッパでは確固たる基盤が確立されており、北米およびアジアでも急速に普及してきています。

IEC 61131-3 は、産業用制御とオートメーションがソフトウェアソリューションによって管理されるようになることで、その要件がますます複雑化していると考えています。

標準のプログラミング言語を適用することは、分析、設計、製造、テスト (検証)、インストール、運用、およびメンテナンスの要件を含むソフトウェアライフサイクルにプラスの影響を与えます。

この場合、メンテナンスに及ぼす影響が重要な役割を果たします。というのは、制御ソフトウェアのアップグレードなどのメンテナンスに要する時間は、元の計画で想定されたものより 2〜4 倍も長くなるのが一般的なためです。

POU (プログラム構成要素) コンポーネントは、個別での実行が可能なタスクとして定義することができます。また、タスクは、1 つの大型コントローラに集約するのではなく、複数の PLC (プログラマブルロジックコントローラ) に分散することもできます。

IEC 61131-3 は、1 つの制御プログラムで複数のプログラミング言語のサポートを提供します。これにより制御プログラムの開発者は、特定のタスクを解決するのに最も適した言語を選択でき、生産性を大幅に高めることができます。

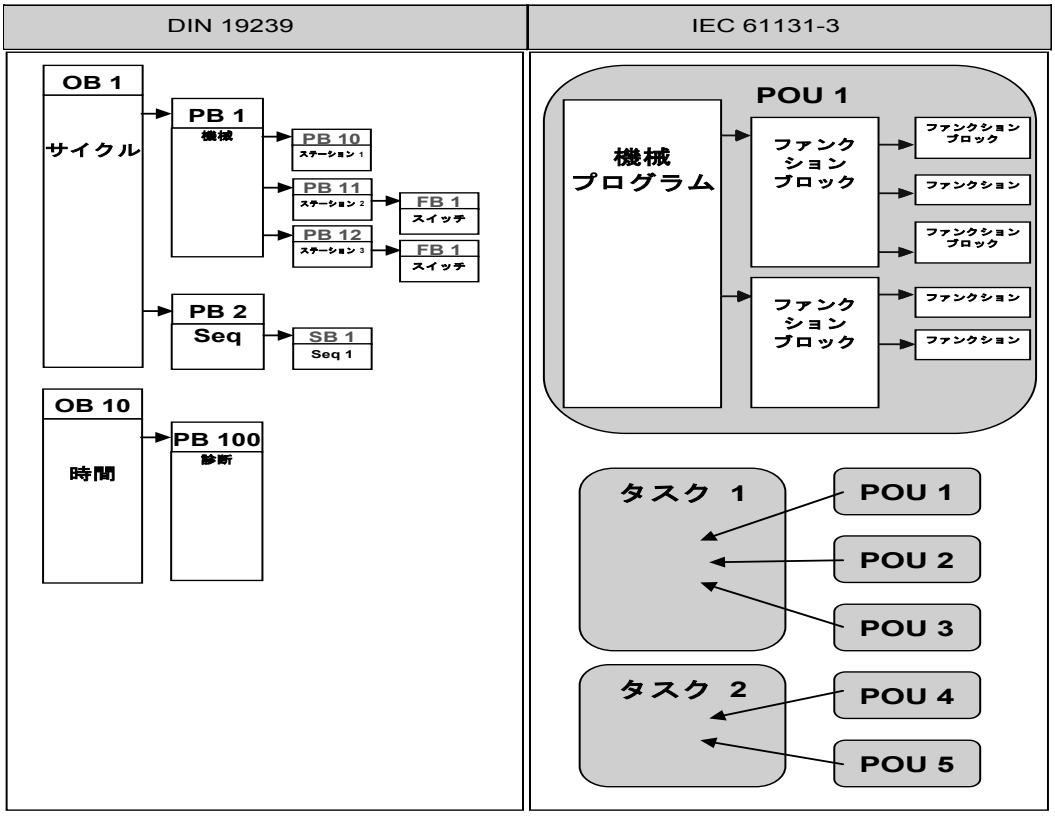
エンジニアリングプロセスに携わる担当者が容易に理解して学習できる、ベンダに依存しない標準言語が設計されました。ヨーロッパでは、次のグラフィカルなドキュメンテーション形式が確立されています。

- 制御およびインターロック用 CFC (コンティニュアスファンクションチャート)
- シーケンス制御用 SFC (シーケンシャルファンクションチャート)

2 POU (プログラム構成要素)

IEC 61131-3 は、PLC プログラミングのための基本モデルを導入しています。古い PLC で通常使用される DIN 19239 に基づいたブロックモデルは、すでに外されています。下のテーブルはこれらの違いを示しています。

DIN 19239	IEC 61131-3
➢ OB (オーガニゼーションブロック) ➢ PB (プログラムブロック) ➢ FB (ファンクションブロック) ➢ SB (シーケンスブロック) ➢ DB (データブロック)	➢ プログラム (PRG) ➢ ファンクションブロック (FB) ➢ ファンクション
OB: OB は周期的または時間起動型の プログラムシーケンスを制御	タスク: タスクは時系列的なプログラム シーケンスを制御



タスク	説明
タスク 1 … n	タスクは定期的に処理されるか、またはイベントの結果として処理されます。タスクには優先度も指定され、この優先度に従ってリソース内に割り付けされます。 タスクの宣言は、タスクの名前、タスクの優先度のエントリ、タスクの実行条件のエントリで構成されます。条件は、タスクが実行されるまでの時間の間隔、イベント (デジタル入力での立上りやグローバル変数の FALSE/TRUE の切り換わり)、または遮

タスク	説明
タスク 1 … n	断のいずれかです。タスクごとに、タスクから呼び出すプログラムのシーケンスを指定することができます。これらのプログラムは記述された順序で実行されます。

POU	説明
POU:	IEC 61131-3 では、プログラム、ファンクションブロック、およびファンクションは、プログラム構成要素 (POU) と呼ばれます。POU のプロパティにより、ユーザプログラムの大規模なモジュラ化と、実装およびテスト済みソフトウェアブロックの再使用を可能にします。POUはタスク宣言によって呼び出すことができます。
プログラム:	主要なプログラムを形成するプログラムは、PSS 4000 の構造化プログラミングに大きな役割を果たします。プログラムは呼び出されず、タスクに割り付けられます。プログラムが割り付けられたタスクはリソースに割り付けられます。プログラムから FB とファンクションを呼び出すことができます。
ファンクションブロック:	ファンクションブロックにより、PSS 4000 の構造化プログラミングはさらに強化されます。FB にはデータを転送するための入力変数および出力変数が格納されます。FB はプログラムおよび FB によって呼び出され、呼び出された FB は他の FB およびファンクションを呼び出すことができます。
ファンクション:	ファンクションには次の規則が適用されます。ファンクションが呼び出される回数や呼び出される時点に関わらず、同じ入力値では常に同じ出力値とファンクション値が提供されます。ファンクションには FB のようにメモリはありません。 ファンクションは、プログラム、FB、またはファンクションによって呼び出される他、自身で他のファンクションを呼び出すこともできます。

2.1 FBとファンクションのインスタンス化

FB とファンクションは、データ型のように、使用前にシステムに認識させる必要があります。これは、一般ファンクション (「一般ファンクション」を参照) には当てはまりません。一般ファンクションの場合は、すでに PSS 4000 で使用できるよう定義されています。

- FB/FUN/スタンダード FB を使ったインスタンス:
  データ型と同様、定義する必要があります。結果として、計算規則 (コマンド文字列) が戻されます。データ構造 (変数) は複製されます。
- 一般ファンクションを使ったインスタンス:
  不要 - 定義は PSS 4000 に認識されています。

3 プログラミング言語

IEC 61131-3 にはテキスト言語とグラフィカル言語、さらにシーケンシャルファンクションチャートが含まれています。シーケンシャルファンクションチャートはテキストですが、グラフィカルに使用することもできます。これらの IEC 61131-3 言語に加えて、PSS 4000 には PNOZmulti の使い慣れた Multi Editor も含まれています。グラフィカル言語は既製の入力、ロジック、および出力ファンクションと連携して機能します。

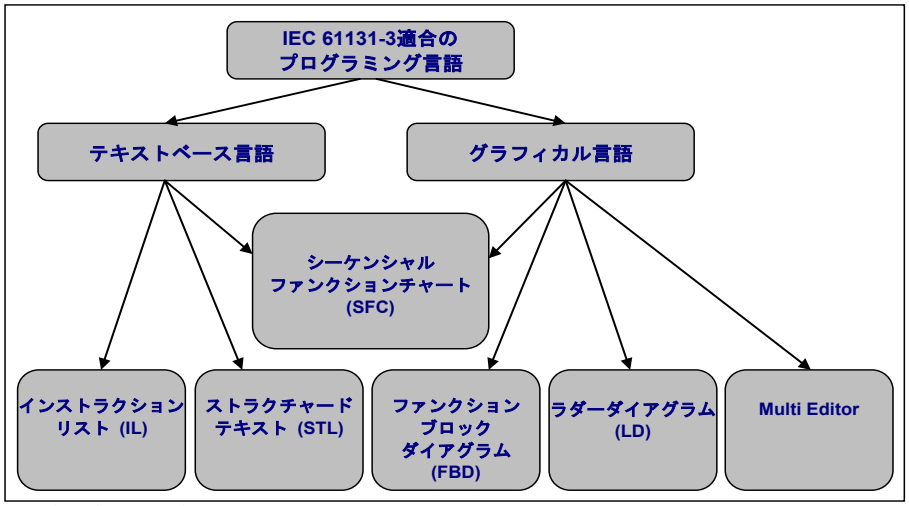


図: プログラミング言語

3.1 IL:

インストラクションリストプログラミング言語でプログラミングするためのエディタ。テキスト言語 IL のコードは一連の命令で構成されます。各命令 (オペレータ) は新しい行で始まり、そのあとにオペランドが続きます。IL のオペランドは変数、定数、またはファンクションブロックのインスタンス名で、これらは PAS4000 に別々の色で示されます。

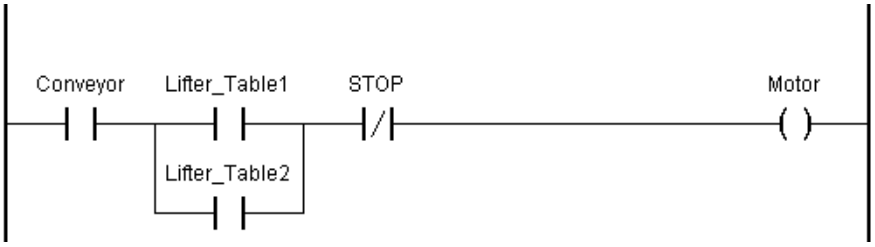
IL では分岐コマンドやラベルも使用できます。プログラムを実行すると、プログラムは分岐コマンドのある行で分岐し、ラベルのある行に移動します。

```
LD TRUE
ST b_Start

LD T#1000ms
ST t_Time_value // set time value

CAL T_clockTP( // Standard-Function_Block “Timer as Impuls
in := b_Start,
pt := t_Time_value,
q => b_Output
)
```

3.2 LD:

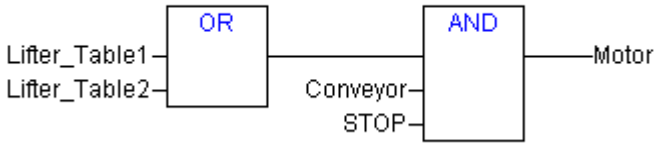
ラダーダイアグラムプログラミング言語でプログラミングするためのエディタ。この言語のコードは接点とコイルで構成されます。接点は電流を左から右に導通します。コイルは受電値を保存します。両方とも BOOL 値の変数です。

接点とコイルはラインでつながれ、電源レールによって左右に拘束されます。



3.3 FBD:

ファンクションブロックダイアグラムプログラミング言語でプログラミングするためのエディタ。この言語のプログラムコードはファンクションとファンクションブロックで構成され、ファンクションとファンクションブロックはラインで相互接続されるか、または変数に接続されます。



3.4 STL:

ストラクチャードテキストプログラミング言語でプログラミングするためのエディタ。
このテキスト言語のプログラムコードは命令と式で構成されます。コードの各行はセミコロンで終わります。

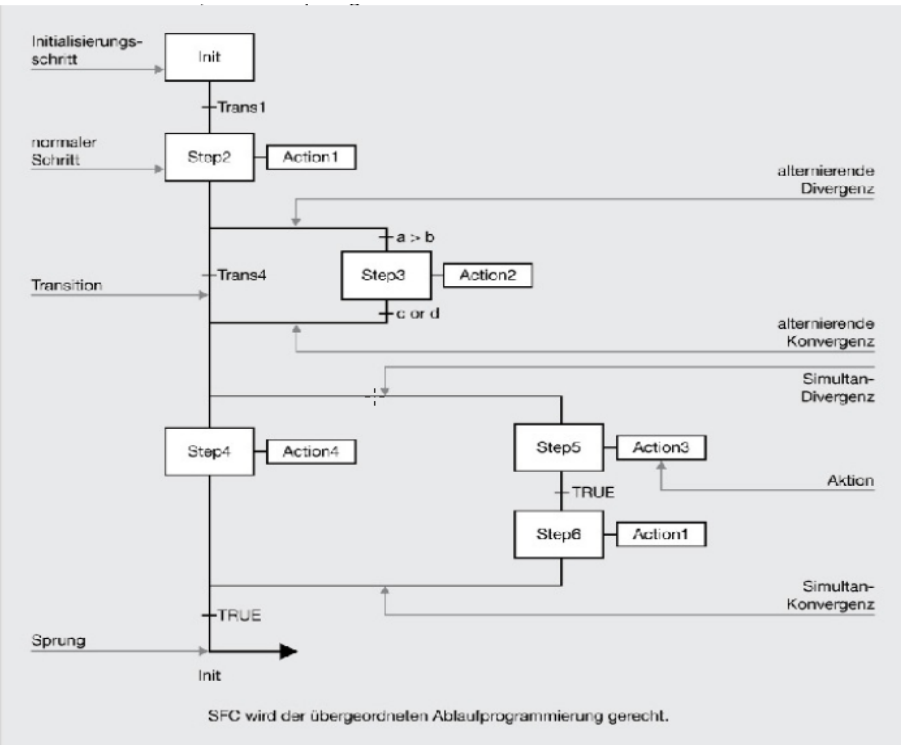
式は命令の一部を形成し、その命令を実行するための値を提供する接点です。式はオペレータとオペランドで構成されます。オペランドのオペレータは、優先度の最も高いものから順に来るように適用しなければなりません。オペランドは定数、変数、またはファンクションの名前です。

```
IF (b_MACHINE_ON = TRUE) THEN
    di_TARGET_POSITION := di_TARGET_POSITION + 100;
    b_INPUT1 := b_INPUT1 AND b_INPUT2;
    b_OUTPUT1 := TRUE;
ELSIF b_MACHINE_ON = FALSE THEN
    b_OUTPUT1 := FALSE;

END_IF;
```

3.5 SFC:

SFC (シーケンシャルファンクションチャート) でプログラミングするためのエディタ。以前は、この言語を SB (シーケンスブロックまたはシーケンサ) と呼ぶこともありました。グラフィカル言語 SFC のプログラムコードはステップとトランジションで構成され、これらのステップとトランジションはラインで接続されます。1 つのステップに 1 つ以上のブロックを割り付けることができます。この種のプログラミングは、主に設備シーケンスを構造的に表すのに使用されます。複雑なシーケンスをより管理しやすくするのに役立ちます。

シーケンスは初期化のステップから始まり、これによりプラントが定義済みの初期化状態になります。あるステップから次のステップへの移行は、トランジション (条件を有効にするステップ) によって制御されます。

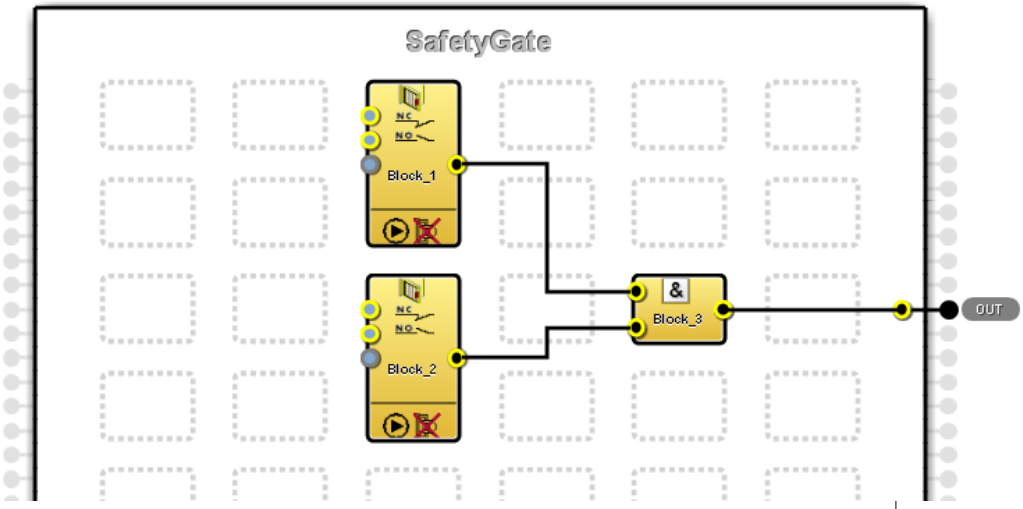


SFC wird der übergeordneten Ablaufprogrammierung gerecht.

3.6 Multi Editor:

PNOZmulti の使い慣れたグラフィックエディタ。

プログラムコードは既製のファンクション、ロジック、および出力ファンクションで構成され、これらの要素はラインで接続されます。非常にシンプルな原理で、管理が簡単です。この種のプログラミングは、シンプルな安全制御機能および監視機能に役立ちます。



4 変数とデータ型

4.1 変数タイプ

IEC 61131-3 は 7 種類の変数を認識し、これらは関連するキーワードで識別されます。制御プログラムの POU 内または POU 間での変数の使用方法は、変数タイプを使って記述します。

変数タイプ	説明	アクセス権		アプリケーション		
		内部 POU	外部 POU	PRG	FB	FUN
VAR	POU 内でのみ使用されるローカル変数です。	読み取り／書き込み	-	X	X	X
VAR_INPUT	入力変数はPOU内では読み取り専用ですが、呼び出された POU 内では読み取り／書き込み可能です。イネーブルパラメータとして機能します。	読み取り専用	読み取り／書き込み		X	X
VAR_ OUTPUT	出力変数は POU 内では読み取り／書き込み可能ですが、呼び出された POU 内では読み取り専用です。イネーブルパラメータとして機能します。	読み取り／書き込み	読み取り専用		X	X
VAR_IN_ OUT	入出力変数は、POU 内および呼び出された POU の両方で読み取り／書き込み可能です。イネーブルパラメータとして機能します。	読み取り／書き込み	読み取り／書き込み		X	X
VAR_ GLOBAL	この変数は「グローバル変数エディタ」を使って宣言され、すべての POU で読み取り／書き込み可能です。POU 内では VAR_EXTERNAL として宣言する必要があります。	読み取り／書き込み	読み取り／書き込み	X	X	
VAR_ EXTERNAL	POU 内でグローバル変数 (VAR_GLOBAL) へのアクセスを有効にするには、VAR_EXTERNAL として宣言する必要があります。	読み取り／書き込み	読み取り／書き込み	X	X	
VAR_TEMP	POU 内でのみ使用される一時ローカル変数です。POU が呼び出されるたびに初期化され、初期値などにリセットされます。	読み取り／書き込み	-	X	X	X

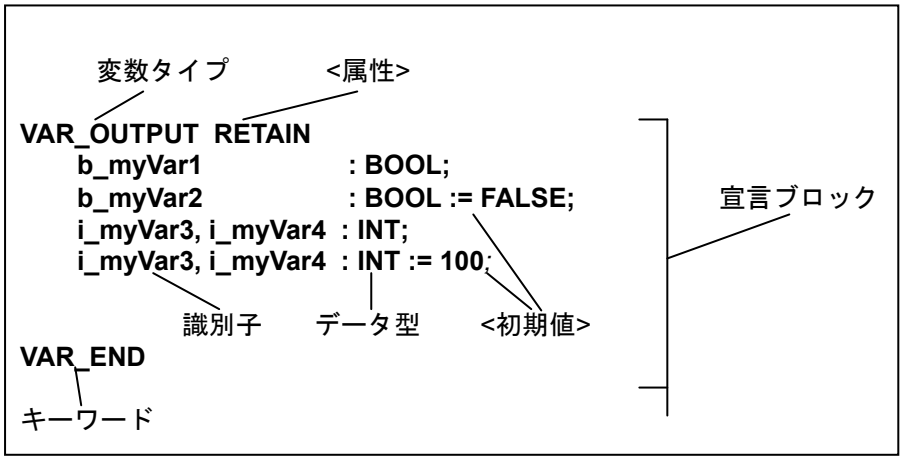
変数の宣言の例:

```
FUNCTION_BLOCK Station1
VAR_INPUT
END_VAR
VAR_OUTPUT
END_VAR

VAR
b_Start:BOOL;
b_Stop: BOOL;
END_VAR
```

4.1.1 変数の宣言

宣言はテキストとして作成され、すべてのプログラミング言語で同じです。ほとんどのデータ型は POU の宣言セクションで宣言されますが、グローバル変数だけは「グローバル変数エディタ」内で個別に宣言されます。

原則として、変数の宣言は次に示すようになります。



4.1.2 変数タイプの属性

属性 **RETAIN**、**PERSISTENT**、**PERSISTENT/RETAIN**、**R_EDGE**、**F-EDGE**、**READ_ONLY**、および **READ_WRITE** は変数タイプのすぐ後ろに指定します。次のテーブルは、各変数タイプに使用できる属性を示しています。

例:

```
VAR_GLOBAL RETAIN
      G_b_myGlobalRetain:   BOOL;
END VAR
```

変数	PERSISTENT	RETAIN	PERSISTENT RETAIN	CONSTANT	R_EDGE F_EDGE	READ_ONLY READ_WRITE
VAR	O	O	O	O	-	-
VAR_INPUT	-	-	-	-	O	-
VAR_OUTPUT	O	O	O	-	-	-
VAR_IN_OUT	-	-	-	-	-	-
VAR_GLOBAL	O	O	O	O	-	-
VAR_EXTERNAL	-	-	-	-	-	-
VAR_TEMP	-	-	-	-	-	-

4.1.3 Retain、Persistent、およびRetain/Persistent

次のテーブルは、各属性が**ダウンロード**、**ウォームリセット**、**コールドリセット**、または**オリジナルリセット**の間に使用された場合の、変数タイプの動作を示しています。

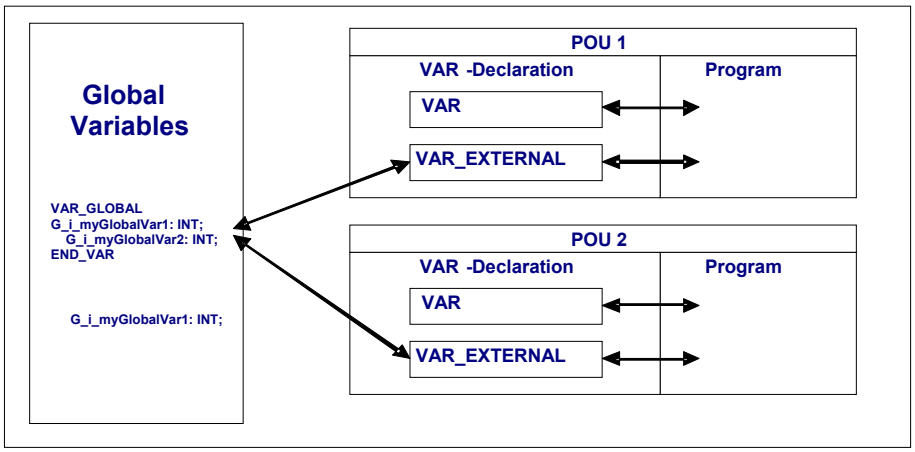
キーワード	ダウンロード	ウォームリセット	コールドリセット	オリジナルリセット
	初期値へのリセット (初期値が使用できない場合はデフォルト値へのリセット)			変数削除
	O	O	O	O
RETAIN	O	X	O	O
PERSISTENT	X	O	O	O
RETAIN PERSISTENT	X	X	O	O

変数タイプと属性の例:

```
VAR_OUTPUT RETAIN
END_VAR

VAR RETAIN
END_VAR

VAR CONSTANT
END_VAR
```

4.1.4 グローバル変数

グローバル変数にはリソース内のすべての POU からアクセスできるため、個々の POU 間でデータを交換できます。



グローバル変数の例:

Resource global variables | Global_Variables

```
VAR_GLOBAL
G_i_myGlobalVar1 :INT;
G_i_myGlobalVar2 :INT;
END_VAR
```

4.2 基本データ型 (Elementary data type)

基本データ型のテーブル:

ST リソースの基本データ型			
データ型	**説明**	**ビット数**	**値の範囲**
BOOL	シングルビット	1	0、1、TRUE、FALSE
BYTE	8 ビットのビット配列	8	0 … FF
WORD	16 ビットのビット配列	16	0 … FFFF
DWORD	32 ビットのビット配列	32	0 … FFFF_FFFF
LWORD	64 ビットのビット配列	64	0 … FFFF_FFFF_FFFF_FFFF
SINT	短整数	8	-128 ... +127
INT	整数	16	-32.768 ... +32.767
DINT	倍精度整数	32	-2.147.483.648 ... +2.147.483.647
LINT	長整数	64	-2^{63} ... $+2^{63}-1$
USINT	符号なし短整数	8	0 ... 255
UINT	符号なし整数	16	0 ... 65.535
UDINT	符号なし倍精度整数	32	0 ... 4.294.967.295
ULINT	符号なし倍精度整数	64	0 ... $+2^{64}-1$
REAL	実数	32	0.0 … X
LREAL	長実数	64	0.0 … X
TIME	スタンダード FB TP、TON、TOF の時間	32	0 ... 4.294.967.295
DATE	日付		D#1970-01-01...D#294247-01-10
TIME_OF_DAY または TOD	1 日のうちの時間		0 ... TOD#23:59:59.999
DATE_AND_TIME または DT	日付と時間		DT#1970-01-01-00:00:00 ...DT #2106-01-19-3:14:08

FS リソースの基本データ型			
データ型	**説明**	**ビット数**	**値の範囲**
SAFEBOOL	シングルビット	1	0、1、TRUE、FALSE
SAFEBYTE	8 ビットのビット配列	8	0 … FF
SAFEWORD	16 ビットのビット配列	16	0 … FFFF
SAFEDWORD	32 ビットのビット配列	32	0 … FFFF_FFFF
SAFESINT	短整数	8	-128 ... +127
SAFEINT	整数	16	-32.768 ... +32.767
SAFEDINT	倍精度整数	32	-2.147.483.648 ... +2.147.483.647
SAFEUSINT	符号なし短整数	8	0 ... 255
SAFEUINT	符号なし整数	16	0 ... 65.535
SAFEUDINT	符号なし倍精度整数	32	0 ... 4.294.967.295

データ型宣言の例:

```
VAR
b_START:BOOL;
b_STOP:BOOL;
di_Speed_actual:DINT;
di_Speed_target:DINT;
END_VAR
```

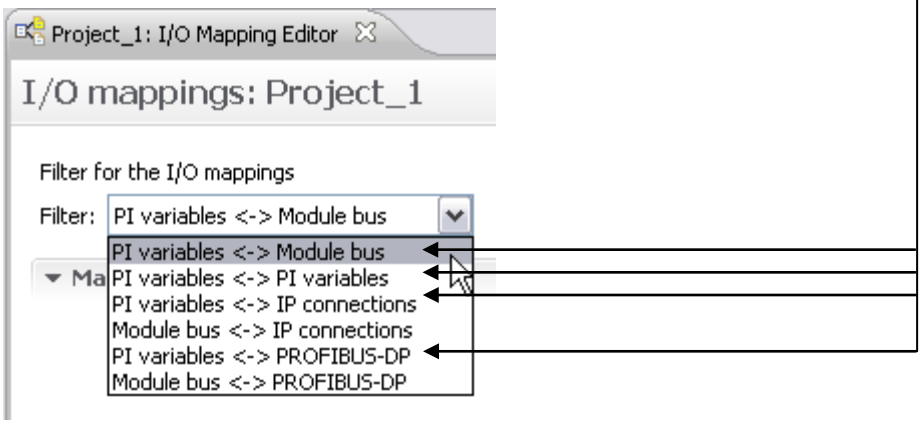
4.2.1 I/Oアドレス指定の変数

固定の入力アドレスおよび出力アドレスへのアクセスには、IEC 61131-3 に従ってキーワード **AT** を使用します。

```
PROGRAM Main_Program
VAR
(*Prozess-Image-Variables:Inputs
for PSSu PLC_left*)
I_b_Digital_Input_1 AT %I*:BOOL;
I_b_Digital_Input_2 AT %I*:SAFEBOOL;
I_w_Analogue_Input_1 AT %I*:WORD;
I_w_Analogue_Input_2 AT %I*:SAFEWORD;

(*Prozess-Image-Variables:Outputs
for PSSu PLC_left*)
I_b_Digital_Output_1 AT %Q*:BOOL;
I_b_Digital_Output_2 AT %Q*:SAFEBOOL;
I_w_Analogue_Output_1 AT %Q*:WORD;
I_w_Analogue_Output_2 AT %Q*:SAFEWORD;
END_VAR
```

絶対 I/O アドレスは、I/O マッピングによってのみ定義されます (「I/O マッピング」を参照)。

☞ I/O マッピングでは、これらの変数は **PI-変数** (Process-Image-Variables) と呼ばれます。



4.2.2 エッジ検出の変数 (R_EDGE、F_EDGE)

属性 R_EDGE および F_EDGE は BOOL 値の入力変数で、それぞれエッジの立上りと立下りを示します。これらの属性は、変数タイプ VAR_INPUT の個々の変数のファンクションブロックで使用できます。例:

```
VAR_INPUT
myRisingEdge: BOOL R_EDGE;
myFallingEdge: BOOL F_EDGE;
END_VAR
```

4.2.3 定数値の変数 (CONSTANT)

ユーザプログラムで固定値を使用するには 2 つのオプションがあります。

- POU のステートメントの部分に直接入力する
- CONSTANT 属性を使った変数を宣言し、固定値を割り付ける

変数に固定値を割り付けるには、宣言ブロックに進み、データ型のキーワードの後にスペースを 1 つ入力してから、CONSTANT 属性を挿入します。変数の固定値は、データ型および割り付けオペレータ「**:=**」の後ろに記述します。固定値はデータ型の値の範囲を超えてはなりません。

```
VAR CONSTANT
b_myconst1: BOOL := TRUE;
i_myconst2: INT := INT#-273;
di_myconst3: DINT := -1234567;
END_VAR
```

CONSTANT 属性は常に宣言ブロック内のすべての変数を参照します。あるデータ型に 1 つの定数値と複数の定数以外の値を宣言する場合、2 つの宣言ブロックを作成する必要があります。CONSTANT 属性は、VAR_GLOBAL データ型、および VAR データ型のプログラム、ファンクションブロック、ファンクションで使用できます。

4.3 派生データ型 (derived data type)

派生データ型 (derived data type) はユーザによって宣言されるデータ型で、基本データ型をベースにしています。ただし、派生データ型には変更または拡張されたプロパティがあります。派生データ型を宣言する理由は、データ型のプロパティをモニタリングするためです。変数が、指定されたプロパティと一致しない場合は、リンク時または実行時にエラーメッセージがトリガされます。

派生データ型を、新しい派生データ型のベースとして再使用することもできます。

派生データ型の宣言はPOUの宣言部分で行い、キーワードTYPEとEND_TYPEの間に定義します。

派生データ型は、基本データ型と同様、変数宣言内で使用でき、キーワードTYPE から END_TYP までの間に定義します。

派生データ型にはさまざまな種類があります。

- 直接派生したデータ型
- 「範囲」データ型 (RANGE)
- 「列挙」データ型 (ENUM)
- 「配列」データ型 (ARRAY)
- 「構造」データ型 (STRUCT)
- PSS_ERROR データ型

4.3.1 直接派生したデータ型

直接派生したデータ型は 1 つの基本データ型に対応していますが、別の名前、および場合によっては別の初期値を持ちます。
直接派生したデータ型の変数は、基本データ型を使用できるすべての場所で使用できます。つまり、「mytype」データ型の変数は、INT データ型の変数を使用できるあらゆる場所で使用できます。

説明	例
基本データ型 INT と同じプロパティ (この例では 0) を持つ派生データ型である「mytype」の宣言。	**TYPE** **i_mytype : INT;** **END_TYPE**
ただし、初期値は変更することもできます (この例では 3)。	**TYPE** **di_mytype : DINT :=3;** **END_TYPE**

4.3.2 「範囲」データ型 (RANGE)

「範囲」データ型は、制限された範囲内で基本データ型の整数値 (ANY_INT) に対応します。

説明	例
値の幅は-20～35	**TYPE** **di_mytype : DINT (-20..35);** **END_TYPE**
値の幅は-20～35 で初期値が 5	**TYPE** **di_mytype : DINT (-20..35) := 5;** **END_TYPE**

4.3.3 「列挙」データ型 (ENUM)

「列挙」データ型は、変数が受け入れることのできる値をすべてリストします。

説明	例
値「red」、「amber」、「green」を持つ派生データ型「traffic light」の宣言。初期値は、自動的に列挙の最初の値を取ります。この例では「red」。	**TYPE** **e_Ampel : (red, yellow, green);** **END_TYPE**
ただし、初期値を定義することもできます。この例では「amber」。	**TYPE** **e_Ampel : (red, yellow, green):= yellow;** **END_TYPE**
さまざまな列挙の値は、データ型の識別子を記述することで区別できます。2 つの列挙に同じ値が含まれている場合は、データ型の識別子を入力する必要があります。例は、2 つの似通った列挙。	**TYPE** **e_Ampel : (red, yellow, green);** **LED : (red, yellow, green);** **END_TYPE**
POU のステートメント部分で指定する場合、「traffic light」と「LED」の「amber」値はデータ型の識別子を記述することで区別されます。	**e_Ampel#yellow** **e_LED#yellow**
「列挙」データ型は明示的に宣言する必要はなく、匿名の宣言でも構いません。つまり、「列挙」データ型に識別子を定義する代わりに、変数を宣言するときに列挙を記述します。	**VAR** **e_myvar : (red, yellow, green):= yellow;** **END_VAR**

4.3.4 「配列」データ型 (ARRAY)

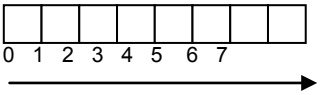
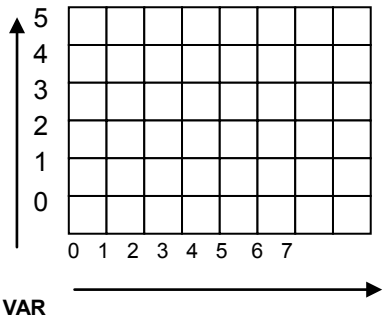
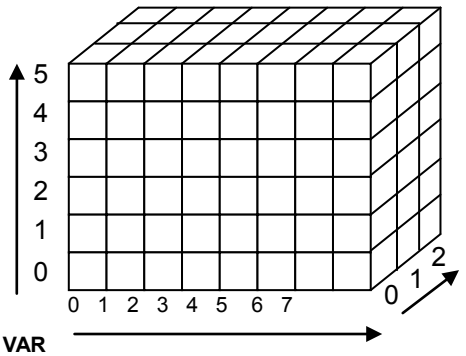
同じ基本データ型または派生データ型のいくつかの変数を 1 つの配列にまとめることができます。また、ファンクションブロックインスタンスの配列を作成することもできます。

- 配列は、同じデータ型の変数の集合である
- グローバル変数またはローカル変数として宣言できる
- 設計内の 1 次元または 2 次元、さらに 3 次元も指定できる

配列の個々のファンクションにはインデックスを使ってアクセスできます。

配列をデータ型として宣言するには、まず識別子を記述する必要があります。

コロンの後にキーワード「ARRAY」を指定し、その後、配列ファンクションに対応させるインデックスを角括弧で囲んで記述します。インデックスには整数とゼロのみ使用できます。配列に5つのファンクションを含める場合は、たとえば、インデックスとして「0.0.4」または「32..36」と入力できます。最後に、キーワード「OF」の後ろにデータ型を記述します。

説明	例
1 次元の配列 (8 配列)	 **VAR** **a_di_mytype : ARRAY [0..7] OF DINT;** **END_VAR**
2 次元の配列: (8 x 6 = 48 配列)	 **VAR** **a_di_mytype: ARRAY [0..7, 0..5] OF DINT;** **END_VAR**
3 次元の配列 (8 x 6 x 3 = 144 配列)	 **VAR** **a_di_mytype: ARRAY [0..7] OF DINT;** **END_VAR**

説明	例
初期値: 配列ファンクションごとに別々の初期値を指定できます。	**TYPE** **a_di_mytype : ARRAY [0..7] OF DINT := [3, -4, 88, 16, -89, 5, 5, 0];** **END_TYPE**
省略構文の初期値: 丸括弧の中の値を繰り返す回数を、括弧の前に記述します。	すべての配列ファンクションの初期値は 3: **TYPE** **a_di_mytype : ARRAY [0..7] OF DINT := [8(3)];** **END_TYPE** 初期値の例 [1, 2, 3, 4, 4, 1, 2, 3]: **TYPE** **a_di_mytype : ARRAY [0..7] OF DINT := [1,2,3,2(4),1,2,3];** **END_TYPE**
多次元配列の初期値: 配列ファンクションごとに別々の初期値を指定できます。	2 次元配列の例: ARRAY [0.0.7, 0.0.5,] 初期値は次の順序で記述する: Element [0,...7, 0,...7,0,⋯7,0,⋯7,0,⋯7]; **TYPE** **a_di_mytype : ARRAY [0..7, 0..5] OF DINT := [3, -4, 88, 16, -89, 5, 5, 0, 3, -4, 88, 16, -89, 5];** **END_TYPE**
多次元配列の省略構文の初期値: 丸括弧の中の値を繰り返す回数を、括弧の前に記述します。	2 次元配列の例: ARRAY [0.0.7, 0.0.5,] 初期値は次の順序で記述する: Element [0,...7, 0,...7,0,⋯7,0,⋯7,0,⋯7]; **TYPE** **a_di_mytype : ARRAY [0..7, 0..5] OF DINT := [1,2,3,4,5,6,7,8,9,36(0),1,2,3];** **END_TYPE**

☞ データ型「DINT」の変数のみ、データ型の初期値指定を省略できます。たとえば、データ型「INT」の場合、初期値は次のように入力します。[# 1 INT, INT # 5, etc.]

☞ 初期値が宣言されてもその数が配列ファンクションの数に満たなかった場合は、そのデータ型の開始値が残りの配列ファンクションに適用されます。記述された初期値の数が配列ファンクションの数を超えている場合は、余分な初期値は無視されます。どちらの場合も、PAS4000 から警告が出されます。

☞ 「ARRAY」データ型の変数の個々のファンクションは、ベースとなった基本データ型の変数を使用できるすべての場所で使用できます。

4.3.5 「構造」データ型 (STRUCT)

「構造」データ型は、さまざまな基本データ型や派生データ型の複数のファンクションの組み合わせです。ファンクションブロックインスタンスも構造ファンクションになり得ます。個々の構造ファンクションにアクセスするには、変数名の後に構造ファンクションの名前を記述します。

説明	例
構造の宣言: 構造ファンクションの初期値は、自動的にそれらのデータ型の初期値と同じになります。	**TYPE** **st_AnalogInput : STRUCT** **MinValue : INT;** **MaxValue : INT;** **READY : BOOL;** **END_STRUCT;** **END_TYPE**
初期値の定義:	**TYPE** **st_AnalogInput : STRUCT** **MinValue : INT :=45;** **MaxValue : INT:=90;** **Ready : BOOL:=TRUE;** **END_STRUCT;** **END_TYPE**
構造がデータ型のベースとして使用された場合、初期値の定義は次のようになります。	**TYPE** **st_mytype : st_AnalogInput := (MinValue := 50, MaxValue := 100, Ready := FALSE);** **END_TYPE**

☞ 構造の個々のファンクションに初期値が定義されていない場合は、ベースとなったデータ型の初期値がこれらのファンクションに適用されます。記述された初期値の数が配列ファンクションの数を超えている場合は、余分な初期値は無視されます。どちらの場合も、PAS4000 から警告が出されます。

☞ 「構造」データ型の変数の個々のファンクションは、ベースとなった基本データ型の変数が使用できる状況であればどのような状況でも使用できます。「構造」データ型は匿名で宣言できません。

4.4 Präfix/Suffix für Variablen

変数およびファンクションの名前に変数の内容が反映されるようにします。
また､変数名には1つまたは複数のプレフィックスをつける必要があります。
これらのプレフィックスはデータ型および変数のプロパティによって異なり、
アンダーバーで区切ります。

基本データ型: ST-データ型	FS-データ型	プレフィックス	例
BOOL	SAFEBOOL	**b_**	b_MyFirstBool : BOOL;
BYTE	SAFEBYTE	**by_**	by_MyFirstByte : BYTE;
WORD	SAFEWORD	**w_**	w_MyFirstWord : WORD;
DWORD	SAFEDWORD	**dw_**	dw_MyFirstDWord : DWORD;
LWORD	SAFESINT	**lw_**	lw_MyFirstLWord : LWORD;
SINT	SAFEINT	**si_**	si_MyFirstSInt : SINT;
INT	SAFEDINT	**i_**	i_MyFirstInt : INT;
DINT	SAFEUSINT	**di_**	di_MyFirst: Dint : DINT;
LINT	SAFEUINT	**li_**	li_MyFirstLInt : LINT;
USINT	SAFEUDINT	**usi_**	usi_MyFirstUSInt : USINT;
UINT	-	**ui_**	ui_MyFirstUInt : UINT;
UDINT	-	**udi_**	udi_MyFirstUDInt : UDINT;
ULINT	-	**uli_**	uli_MyFirstULInt : ULINT;
REAL	-	**r_**	r_MyFirstReal : REAL;
LREAL	-	**lr_**	lr_MyFirstLReal : LREAL;
STRING	-	**s_**	s_MyFirstString : STRING;
TIME	-	**t_**	t_MyFirstTime : TIME;
DATE	-	**d_**	d_MyFirstDate : DATE;
TIME_OF_DAY または TOD	-	**tod_**	tod_MyFirstTod : TOD;
DATE_AND_TIME または DT	-	**dt_**	dt_MyFirstDt : DT;
ENUM	-	**e_**	e_MyNumbers: (red,yellow,green);

派生データ型: データ型	プレフィックス	例
POINTER	**p_**	p_dw_MyFirstPointer : POINTER TO DWORD;
ARRAY	**a_**	a_di_MyFirstArray : ARRAY [0..9] OF DINT;
STRUCT	**st_**	st_MyFirstStruct : STRUCT;
POINTER	**p_**	p_dw_MyFirstPointer : POINTER TO DWORD;

E/Amapping „AT “ = PII / PIO の変数 AT-変数	プレフィックス	例
PII =入力	**I_**	I_b_MyFirstInput AT %I* : BOOL;
PIO =出力	**Q_**	Q_w_MyFirstOutput AT %Q* : WORD;

グローバル変数	プレフィックス	例
基本データ型	**G_**	G_b_MyFirst_GlobalVar: BOOL;
PII =入力	**G_I_**	G_I_b_MyFirst_GlobalVar AT%I: BOOL;
PIO =出力	**G_Q_**	G_Q_b_MyFirst_GlobalVar AT%Q: BOOL;

スタンダードファンクションブロック: FB 名	プレフィックス	例
SR	**sr_**	sr_MyFirstSr_Filpflop: SR;
RS	**rs_**	rs_MyFirstRs_Filpflop: RS;
R_TRIG	**rt_**	rt_MyFirstR_Trig : R_TRIG;
F_TRIG	**ft_**	ft_MyFirstF_Trig : F_TRIG;
CTU … CTUD_ULINT	**C_i** **…** **C_uli**	C_i_MyFirstCtu: CTU; … C_uli_MyFirstCtu: CTUD_ULINT;
TP	**T_**	T_MyFirstTp: TP;
TON	**T_**	T_MyFirstTon: TON;
TOF	**T_**	T_MyFirstTof: TOF;

5 演習: 選択肢問題

問題 1

コンベヤベルトシステムの 20 個のスイッチに POU を作成する必要があります。POUには、さまざまなセンサまたはアクチュエータ用の入力変数および出力変数を含めなければなりません。また、診断用など、さまざまなスイッチ条件を認識できるようにする必要があります。どのような種類の POU が必要ですか？

	正解
1. プログラム	○
2. ファンクションブロック	○
3. ファンクション	○
4. タスク	○

問題 2

さまざまな信号や制御ランプの点滅を有効にするためのパルスジェネレータが必要です。どのような種類の POU を作成する必要がありますか？

	正解
1. プログラム	○
2. ファンクションブロック	○
3. ファンクション	○
4. タスク	○

問題 3

ドライブ用に単純な PI コントローラをプログラミングする必要があります。これは、ドライブ自体にはメモリを持たずに、設定／実速度に基づいて転送速度を規制するためのものです。このような場合、どのような POU が最も適していますか？

	正解
1. プログラム	○
2. ファンクションブロック	○
3. ファンクション	○
4. タスク	○

問題 4

ファンクションブロックおよびファンクション POU を呼び出せるのは、どの POU からですか？

	正解
1. プログラム POU のみ	○
2. プログラムおよびファンクションブロック POU	○
3. 2. プログラムおよびファンクション POU	○
4. すべての POU (プログラム、ファンクションブロック、ファンクション)	○

問題 5

プログラム POU はどの POU から呼び出せますか？

	正解
1. どの POU からも呼び出せない	○
2. プログラム POU のみ	○
3. プログラムおよびファンクションブロック POU	○
4. すべての POU (プログラム、ファンクションブロック、ファンクション)	○

問題 6

変数タイプ「VAR」および「VAR_TEMP」についての記述で正しいものはどれですか？

	正解
1. これらは自身の POU の中のローカル変数です。 両方とも、POU が呼び出されるたびに初期値にリセットされます。	❍
2. これらは自身の POU の中のローカル変数です。 「VAR_TEMP」だけが POU が呼び出されるたびに初期値にリセットされます。「VAR」はその値を維持します。	❍
3. これらは自身の POU の中のローカル変数です。 「VAR」だけが POU が呼び出されるたびに初期値にリセットされます。「VAR_TEMP」はその値を維持します。	❍
4. これらは自身の POU の中のローカル変数です。 両方とも、その値を維持します。	❍

問題 7

データを他の POU に転送するための入出力変数を含むファンクションブロックを作る必要があります。これらの変数のうちの 1 つの値は、この FB および呼び出されたPOUの両方で上書きされる必要があります。どの変数タイプを選択しますか？

	正解
1. VAR_INPUT	❍
2. VAR_OUTPUT	❍
3. VAR_IN_OUT	❍
4. このファンクションは不可能	❍

問題 8

さまざまな POU で使用できるよう、点滅装置 (1 Hz) に変数を設定する必要があります。どの変数タイプを選択しますか？

	正解
1. VAR_GLOBAL	○
2. VAR_TEMP	○
3. VAR	○
4. VAR_IN_OUT	○

問題 9

-100 000～+100 000 の数字を PSS 4000 で処理する必要があります。どのデータ型を選択できますか？

	正解
1. DINT	○
2. UDINT	○
3. INT	○
4. UINT	○

問題 10

変数はどのように宣言しますか。下の空欄に適切なプレフィックスを記入してください。

```
VAR
    .....Start-Button AT %I* :BOOL;
    .....Pressure_switch AT %I* :WORD;
    .....Pressure_monitoring :UDINT;
    .....Machine_datas :ARRAY [0..99] OF DINT;
END_VAR
```

6 一般ファンクションとスタンダードファンクションブロック (IEC 61131 - 適合リスト)

6.1 一般ファンクション

一般ファンクションは数多くの用途に使用できます。ファンクションの呼び出しをサポートするすべてのプログラミング言語で使用できます。ファンクションに、「ストラクチャードテキスト (ST)」プログラミング言語で異なる名前 (「SUB」や「-」など) がある場合、その両方が表示されます。

下のテーブルに関するメモ:

- 「**ANY**」データ型: すべてのデータ型 (基本データ型および派生データ型)。
- 「**ANY_BIT**」データ型: BOOL、BYTE、WORD、DWORD、LWORD。
- 「**ANY_NUM**」データ型: 次以外のすべてのデータ型: BOOL、BYTE、WORD、DWORD、LWORD。

「ステップ」の列には、ファンクションがどのバージョンからサポートされているかが記載されています。

ステップ = 0: PAS4000 バージョン 1.0 以降

ステップ = 1: PAS4000 バージョン 1.1 以降

ステップ = 2: PAS4000 バージョン 1.2 以降

ステップ = 3: PAS4000 バージョン 1.3 以降

ステップ = 4: PAS4000 バージョン 1.4 以降

…

…

…

ステップ=99: 計画中、サポート予定

PAS4000 の一般ファンクションのテーブル:

ステップ	一般ファンクションの名前	変数 (許可されたデータ型)	説明	
	型変換ファンクション			
	*_TO_**	ANY	データ型の型変換。例: BYTE_TO_INT または TIME_TO_DINT など。	
	TRUNC	ANY_NUM	切り捨てファンクション:REAL 値の少数値は切り捨てられ、整数値になります。	
	数値ファンクション			
	ABS	ANY_NUM	絶対値の算出。値は維持され、符号は正に変わります。	
	SQRT	ANY_NUM	平方根の算出。	
	EXP	ANY_NUM	指数関数の算出。	
	-LN	ANY_NUM	自然対数の算出。	
	LOG	ANY_NUM	10 を底とする対数の算出。	
	角度ファンクション			
	SIN	ANY_NUM	正弦の算出。(ラジアンでは IN を記述)	
	COS	ANY_NUM	余弦の算出。(ラジアンでは IN を記述)	
	TAN	ANY_NUM	正接の算出。(ラジアンでは IN を記述)	
	ASIN	ANY_NUM	逆正弦の算出 (主要値)。(ラジアンでは IN を記述)	
	ACOS	ANY_NUM	逆余弦の算出 (主要値)。(ラジアンでは IN を記述)	
	ATAN	ANY_NUM	逆正接の算出 (主要値)。(ラジアンでは IN を記述)	
	算術ファンクション			
	ADD + (ST)	ANY_NUM	加算。任意の数の入力パラメータを指定できます。	
	算術ファンクション			
	SUB - (ST)	ANY_NUM	減算。任意の数の入力パラメータを指定できます。	
	MUL * (ST)	ANY_NUM	乗算。任意の数の入力パラメータを指定できます。	
	DIV / (ST)	ANY_NUM	除算。任意の数の入力パラメータを指定できます。	
	MOD	ANY_NUM	除算の余りの算出。 例: If IN2 = 0, then OUT = 0 If IN2 <> 0, then OUT := IN1 - (IN1 / IN2) * IN2	
	EXPT ** (ST)	ANY_NUM	入力パラメータの累乗。 例: OUT := IN1IN2 (IN1 と OUT が同じデータ型であることが必須)	
	MOVE := (ST)	ANY	入力パラメータの出力パラメータへのアロケーション。 例: OUT := IN	
	シフトおよびローテートファンクション			
	SHL	ANY	左シフト: IN は左に N ビットシフトされ、ゼロは右側から埋められます。N はゼロ以上にする必要があります。	
	SHR	ANY	右シフト: IN は右に N ビットシフトされ、ゼロは左側から埋められます。N はゼロ以上にする必要があります。	
	ROR	ANY	右ローテート: IN は右に N ビット回されます。ビット列の最下位ビット (ビット 0) は、最上位ビット (ビット 15) に回されます。N はゼロ以上にする必要があります。	

ステップ	一般ファンクションの名前	変数 (許可されたデータ型)	説明
	ROL	ANY	左ローテート: IN は左に N ビット回されます。ビット列の最上位ビット (ビット 15) は、最下位ビット (ビット 0) に回されます。 N はゼロ以上にする必要があります。
	選択ファンクション		
	SEL	ANY	バイナリの選択: 例: If G = 0, then OUT := IN0 If G = 1, then OUT:= IN1
	MAX	ANY_NUM	最大値の定義: 最大入力パラメータの定義。任意の数の入力パラメータを指定できます。 例: OUT := MAX (IN1, IN2, ..., INn)
	MIN	ANY_NUM	最小値の定義: 最小入力パラメータの定義。任意の数の入力パラメータを指定できます。 例: OUT := MIN (IN1, IN2, ..., INn)
	LIMIT	ANY_NUM	LIMIT - リミッタ: MN は下限値、MX は上限値で、IN は測定値などです。このファンクションは、IN が制限値内である限り、OUT が IN と等価となるようにします。値が下限値の MN を下回る場合、OUT は MN と等価になります。値が上限値の MX を上回る場合、OUT は MX と等価になります。 例: OUT := MIN (MAX (IN, MN), MX)
	選択ファンクション		
	MUX	ANY_NUM	マルチプレクサ: 入力パラメータ K は、出力に切り換える入力パラメータを決定します。K は 0〜n-1 の範囲内であることが必要です。任意の数の入力パラメータを指定できます。 例: If K = i, then OUT := INi
	比較ファンクション		
	GT > (ST)	ANY	「>」比較: IN1 が IN2 よりも大きく、IN2 が IN3 よりも大きいといった場合、OUT は 1 になります。任意の数の入力パラメータを指定できます。 例: OUT := 1, if IN1 > IN2 > ...> INn
	GE >= (ST)	ANY	「>=」比較: IN1 が IN2 よりも大きいかまたは等しく、IN2 が IN3 よりも大きいかまたは等しいといった場合、OUT は 1 になります。任意の数の入力パラメータを指定できます。 例: OUT := 1, if IN1 >= IN2 >= ...>= INn
	EQ = (ST)	ANY	「=」比較: IN1 が IN2 と等しく、IN2 が IN3 と等しいといった場合、OUT は 1 になります。任意の数の入力パラメータを指定できます。 例: OUT := 1, if IN1 = IN2 = ...= Inn
	NE <> (ST)	ANY	「<>」比較: IN1 が IN2 と等しくない場合、OUT は 1 になります。 例: OUT := 1, if IN1 <> IN2
	LT < (ST)	ANY	「<」比較: IN1 が IN2 よりも小さく、IN2 が IN3 よりも小さいといった場合、OUT は 1 になります。任意の数の入力パラメータを指定できます。 例: OUT := 1, if IN1 < IN2 < ...< Inn

ステップ	一般ファンクションの名前	変数 (許可されたデータ型)	説明
	LE <= (ST)	ANY	「Less than or equal to」比較: IN1 が IN2 よりも小さいかまたは等しく、IN2 が IN3 よりも小さいかまたは等しいといった場合、OUT は 1 になります。任意の数の入力パラメータを指定できます。 例: OUT := 1, if IN1 <= IN2 <= ...<= Inn
	時間と日付ファンクション		
	ADD_TIME + (ST)	TIME	時間の加算
	ADD_TOD_ TIME	TIME / DATE	時間または日付と時間の加算
	ADD_DT_ TIME	TIME / DATE	日付と時間の加算
	SUB_TIME + (ST)	TIME	時間の減算
	SUB_TOD_ TIME	TIME / DATE	時間または日付と時間の減算
	SUB_DT_ TIME	TIME / DATE	日付と時間の減算
	SUB_DATE _DATE	DATE	2 つの日付の減算
	時間と日付ファンクション		
	SUB_DT_ DT	TIME / DATE	時間指定のある 2 つの日付の減算
	SUB_TOD_ TOD	TIME / DATE	2 つの時間または日付の減算
	MULTIME	TIME	時間の乗算
	DIVTIME	TIME	時間の除算
	CONCAD_ DATE_ TOD	TIME / DATE	日付と時間の結合: 日付 (DATE) と一日のうちの時間 (TOD) が日付と時間 (DT) に結合されます。

6.2 スタンダードファンクションブロック

PAS4000 では、次のスタンダードファンクションブロックを使用することができます。

スタンダードファンクションブロックの名前	変数 (許可されたデータ型)	説明
フリップフロップ		
SR	BOOL	SR-フリップフロップ: 先行の設定 (優先の設定)
RS	BOOL	RS-フリップフロップ: 先行のリセット (優先のリセット)
パルス立上り／立下り		
R_EDGE	BOOL	エッジ情報: 立上り
F_EDGE	BOOL	エッジ情報: 立下り
R_TRIG	BOOL	立上りでトリガ
F_TRIG	BOOL	立下りでトリガ
カウンタ		
CTU	INT	整数形式の加算カウンタ
CTU_DINT	DINT	倍精度浮動小数点形式ダブルの加算カウンタ
CTU_LINT	LINT	長整数形式の加算カウンタ
CTU_UDINT	UDINT	倍精度浮動小数点形式ダブルの加算カウンタ (符号なし)
CTU_ULINT	ULINT	長整数形式の加算カウンタ(符号なし)
CTD	INT	減算カウンタ
CTD_DINT	DINT	倍精度浮動小数点形式ダブルの減算カウンタ
CTD_LINT	LINT	長整数形式の減算カウンタ
CTD_UDINT	UDINT	倍精度浮動小数点形式ダブルの減算カウンタ (符号なし)
CTD_ULINT	ULINT	長整数形式の減算カウンタ(符号なし)
CTUD	INT	整数形式の加算／減算カウンタ
CTUD_DINT	DINT	倍精度浮動小数点形式ダブルの加算／減算カウンタ
CTUD_LINT	LINT	長整数形式の加算／減算カウンタ(符号なし)
カウンタ		
CTUD_UDINT	UDINT	倍精度浮動小数点形式ダブルの加算／減算カウンタ (符号なし)
CTUD_ULINT	ULINT	符号なし長整数形式の加算／減算カウンタ
タイマ		
TP	TIME	パルス幅
TON	TIME	スイッチオンディレイ時間
TOF	TIME	スイッチオフディレイ時間

スタンダードファンクションブロックの使用例:

```
FUNCTION_BLOCK
VAR
    b_Start :BOOL;
    b_Output AT%Q*:BOOL;
    t_Time_value:TIME;
    T_ClockTP:TP;
END_VAR
```

スタンダード FB
TP = パルス幅

```
LD TRUE
ST b_Start

    LD T#1000ms
    ST t_Time_value

    CAL T_ClockTP(
    in := b_Start,
    pt := t_Time_value,
    q => b_Output
    )
    END_FUNCTION_BLOCK
```

スタンダード FB
の呼び出し

6.2.1 インスタンス化

IEC 61131-3 では、ファンクションブロックのインスタンス化が利用できます。インスタンスとは、FB が呼び出されたときにすべての内部変数および入出力変数が保存される構造を指します。つまり、プログラムコードは転送されますが、データ構造は複製されます。

「TP」FB を 3 回呼び出すプログラムには、この FB のインスタンスが 3 つ含まれています (呼び出し 1 回につき 1 つ)。

このオブジェクト指向型プロシージャにより、プログラム評価は呼び出しごとに行われ、副次的な影響がなくなります。すべてのインスタンスが同じプログラムコードを使用することが重要です。そうすることで、プログラムコードに加えるあらゆる変更が、すべての呼び出しに同様の影響を与えます。

PAS4000 のような最新のソフトウェアツールは、自動宣言を使ったインスタンス化を実装するのに役立ちます。自動宣言では、FB を呼び出す際にインスタンス名が単純に定義され、これが呼び出しのデータ構造を管理します。

FB のインスタンス化の例:

```
VAR INPUT
    t_Pulse_duration, t_Pause_time:TIME;
END_VAR
VAR OUTPUT
    b_Clock_output:BOOL;
END_VAR

VAR
    b_End_time:BOOL;
    T_ON :TP; (*1. Instance of TP*)
    T_OFF :TP; (*2. Instance of TP*)
    b_Enable1, b_Enable2:BOOL;
END_VAR
    LDN T_OFF.q (*with Output „q“ of TP „OFF“*)
    ST b_ Enable 1

    CAL T_ON( (*Call of 1. Instance*)
    in := b_ Enable1,
    pt := t_Pulse_duration
    )
    LD T_ON.q (*with Output „q“ of TP „ON“*)
    ST b_Clock_output
    STN b_ Enable2
    CAL T_OFF( (*Call of 2. Instance*)
    in := b_ Enable2,
    pt := t_Pulse_duration
    )   )
    END_FUNCTION_BLOCK
```

FB「TP」の最初のインスタンス

FB「TP」の2番目のインスタンス

最初のインスタンスの呼び出し

2番目のインスタンスの呼び出し

7 システムファンクション

PAS4000 では、次のシステムファンクションを使用することができます。

システムファンクションの名前	変数 (許可されたデータ型)	説明
フリップフロップ		
pssGetMsTickCounter	UDINT	**ミリ秒のカウンタの読み取り** pssGetMsTickCounter ファンクションを使って、ミリ秒のカウンタからカウントを要求できます。リソースが停止状態にない場合、カウンタは毎ミリ秒増加します。 カウンタの最大値に達すると 0 に切り替わり、カウントを続けます。 ホットスタートの後は、カウンタは停止したポイントからカウントを再開します。その 他のすべてのケース (ウォームリスタート、ウォームリセット、コールドリスタート、コールドリセット、オリジナルリセット、およびダウンロード) では、カウンタはゼロにリセットされます。
VALID	BOOL	有効なビットがポーリングされます。
IO_DatatypeMerger	ANY_NUM	
IO_DatatypeSplitter	ANY_NUM	

8 プログラム内の定数

定数の形式
BOOL#0 … 1
BYTE#0 … 255
WORD#0 … 65.535
DWORD#0 … 4.294.967.295
LWORD#0 … $+2^{64}$-1
SINT#-128 ... +127
INT#-32.768 ... +32.767
DINT#-2.147.483.648 ... +2.147.483.647
LINT#-2^{63} ... $+2^{63}$-1
USINT#0 ... 255
UINT#0 ... 65.535
UDINT#0 ... 4.294.967.295
ULINT#0 ... $+2^{64}$-1
REAL#0.0 … X
LREAL#0.0 … X
TIME#0 ... 4.294.967.295 T#0 ... 4.294.967.295
DATE#1970-01-01 ... 294247-01-10 D#1970-01-01 ... 294247-01-10
TIME_OF_DAY TOD#0 ... 23:59:59.999
DATE_AND_TIME#1970-01-01-00:00:00 ...2106-01-19-3:14:08 DT#1970-01-01-00:00:00 ... 2106-01-19-3:14:08

9 リソース

リソースはコードを実行する実体を指し、FS リソース、ST リソース、VISU リソース、MC リソースなどがあります。

PSS 4000 には、フェイルセーフ (FS) IEC 61 131 コードを実行するリソースと、非フェイルセーフ (ST) IEC 61 131 コードを実行するリソースの 2 つがあります。

プロジェクトは複数の PSS 4000 ヘッドモジュールに配布されます。その結果、2 つの追加リソース (FS と ST) が PSS 4000 の各ヘッドモジュールに発生します。

これにより、各リソースには最大 9 つのタスクを含めることができます (「タスク」の項を参照)。各タスクには任意の数の POU を含めることができます。

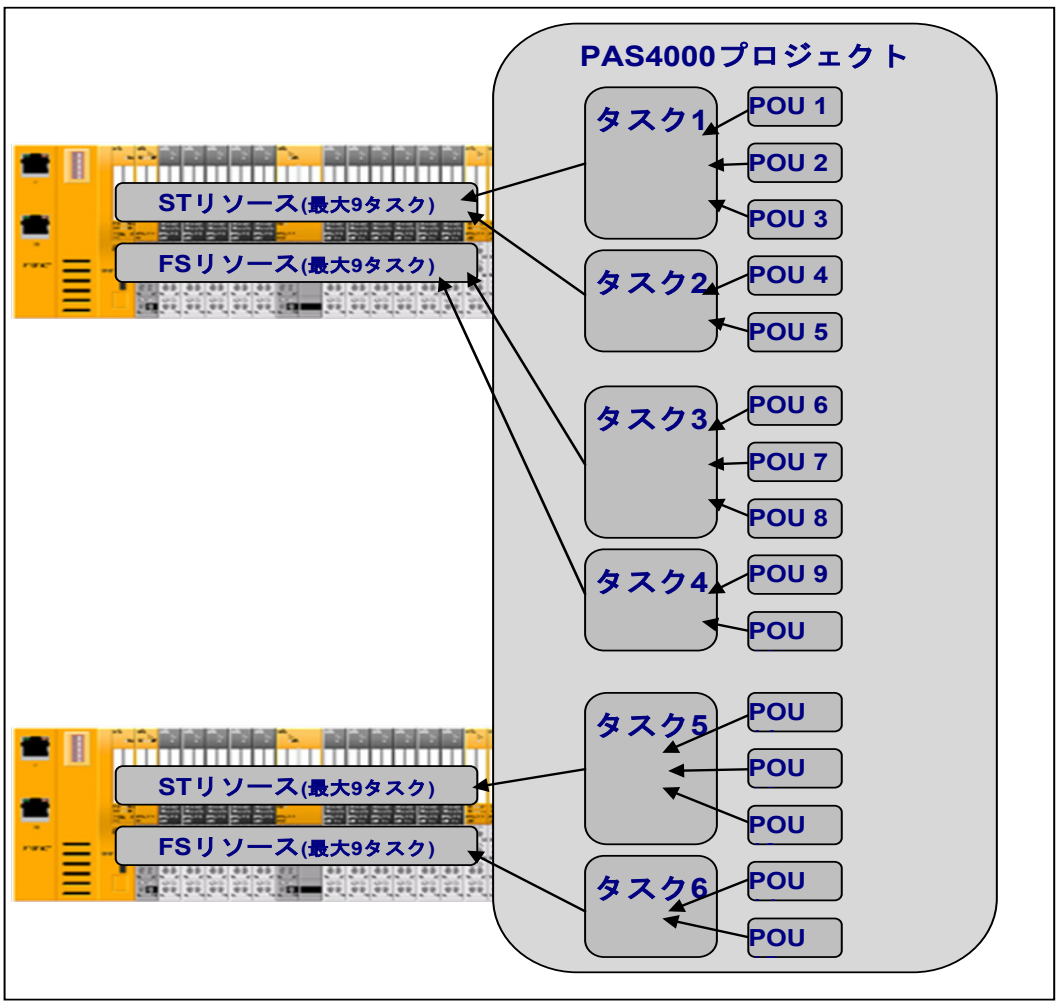


図: リソース、タスク、POU

9.1 プログラミングモデル

プログラミングモデルはプログラム作成の基本原則を定義します。

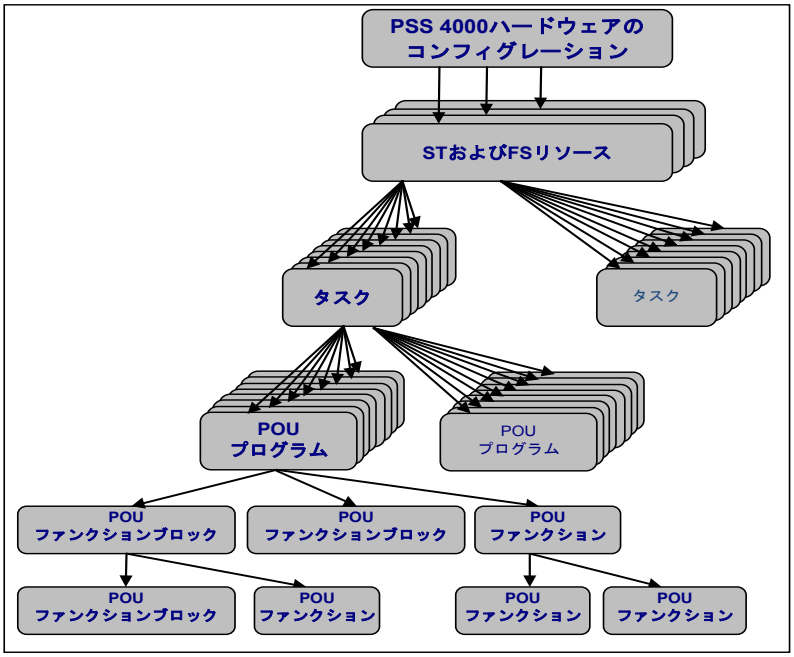
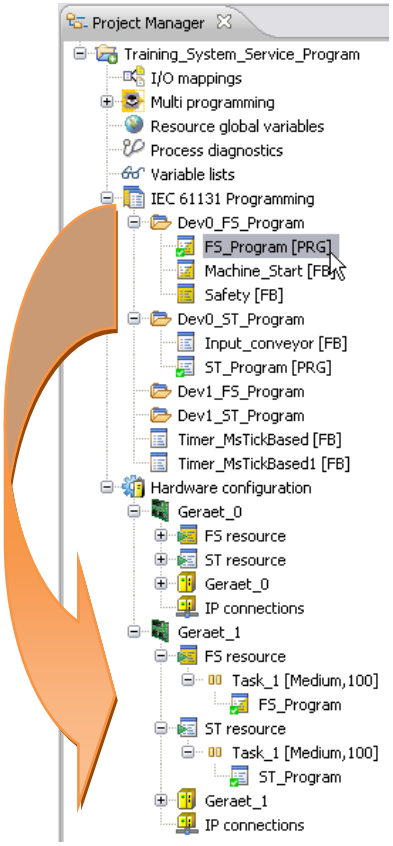


図: プログラミングモデル

9.2 リソースの割り付け:

プログラム型プログラム構成要素のリソースへの割り付け。



コンポーネントモデル内の項:

リソースによって実行されるコンポーネントブロック／アスペクトブロックの定義。

ロジックブロック:

ロジックブロックもユーザが割り付けできますが、必須ではありません。ユーザによって割り付けられない場合は、最も適したリソースによって自動的に実行されます。

10 タスク

PSS 4000 は、プリエンプティブスケジューリングを使ったマルチタスクシステムです。つまり、1 つのリソースで最大 9 個のタスクを使用できます。タスクの実行はシステムによって構成され、タスクのプロパティがタスクを実行するタイミングと順番に影響します。優先度の高いタスクが保留になっている場合、優先度の低いタスクの実行は中断されます。

10.1 タスクの実行

タスクを実行する際は、次の規則が適用されます。

- タスクの実行:
  たとえば、指定された間隔が経過した場合など、条件が満たされたタスクは実行されます。制御がイベント起動の場合、選択した変数は FALSE から TRUE へと切り換わります。複数のタスクの条件が満たされた場合は、優先度が最も高いタスクが実行されます。
- タスクの中断:
  あるタスクが処理されているときに、それよりも優先度の高いタスクの条件が満たされた場合、優先度の低いほうのタスクは、優先度の高いタスクが完了するまで中断されます。また、プログラムがシステムファンクションの実行待ち状態である場合なども、タスクは中断されます。
- タスクへの POU の割り付け:
  1 つのタスクに任意の数のプログラム型 POU を割り付けることができます。
- デバイスあたりのタスクの数:
  プロジェクト内の各デバイスは個別にタスクの実行を制御します。デバイスごとに FS リソースと ST リソースを 1 つずつ持つことができ、それぞれのリソースには最大 9 個のタスクを含めることができます。
- 優先度:
  3 つの優先度を使用できます (高、中、低)。タスクは優先度に従って実行されます。優先度が同じ定周期タスクの場合は、周期長によって決まり (Implicit の優先度)、周期長が最も短いタスクが最初に実行されます。優先度も周期長も同じタスクの場合、順番はランダムになります。
- FS リソースと ST リソースの連携:
  デバイスに FS リソースと ST リソースがある場合は、どちらも同じ CPU によって実行されます。
- 保留中のタスクの確認:
  システムクロックパルスごとに、システムは実行が保留されているタスクがないかどうかを確認し、実行の順序を定義し直します。あるタスクが処理されているときに、それよりも優先度の高いタスクが保留されている場合は、実行中のタスクが中断されます。
- プログラム型 POU の実行順序:
  1 つのタスクに複数のプログラムが割り付けられている場合、プログラムはプログラム名のアルファベット順に実行されます。

10.2 タスクのプロパティ

タスクのプロパティはタスクコンフィグレーションで定義されます。タスクの宣言は、タスクの名前、タスクの優先度のエントリ、タスクの実行条件のエントリで構成されます。条件は、タスクが実行されるまでの時間の間隔、イベント (デジタル入力での立上りやグローバル変数の FALSE/TRUE の切り換わり)、または遮断のいずれかです。

タスクごとに、タスクから呼び出すプログラムのシーケンスを指定することができます。これらのプログラムは記述された順序で実行されます。

10.3 優先度

各タスクの優先度を定義できます。優先度は、リソースの他のタスクと比較した各タスクの重要度です。つまり、リソースのタスクが実行される順序は優先度で決まります。優先度には、高、中、低があります。1 つのリソースに最大 9 個のタスクを含めることができ、高、中、低の優先度を 3 つずつ指定できます。次の順序が適用されます。

1. FS タスク、優先度 高
2. ST タスク、優先度 高
3. FS タスク、優先度 中
4. ST タスク、優先度 中
5. FS タスク、優先度 低
6. ST タスク、優先度 低

☞ 各優先度 (高、中、低) は、各リソースに 3 回まで適用できます。つまり、リソースに 9 個のタスクが作成されている場合、優先度の高、中、低がそれぞれ 3 つずつとなります。

10.4 タスクのタイプ

タスクには 3 つのタイプがあります。

- フリーホイーリングタスク (Freewheeling task: 周期的)
- 定周期タスク (Periodic task: 時間起動)
- イベント起動タスク (Event-driven task: 中断)

フリーホイーリングタスク:

フリーホイーリングタスクは、先行タスクの実行が完了すると、再度実行されます。そのため、実行の間隔は単に制御プログラムの長さで決まります。また、優先度の高いタスク (たとえば定周期タスク) が原因で延長されることもあります。

2 つの例を次に示します。

1. フリーホイールの FS タスクと ST タスクが 1 つずつ
2. フリーホイールの FS タスクと ST タスク、および優先度が高の定周期タスク (20 ms) が 1 つ

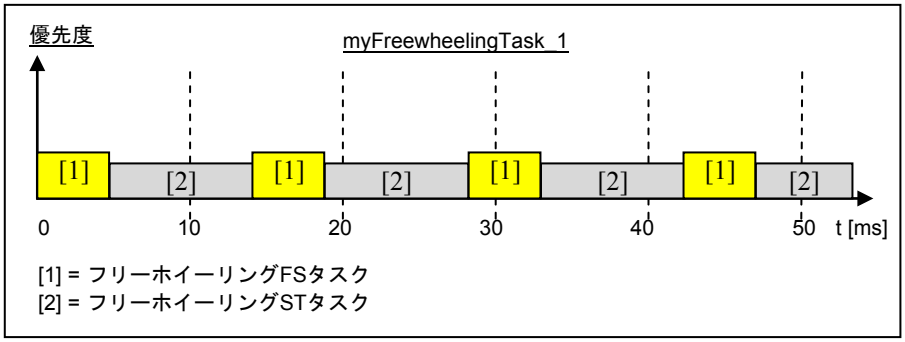


図: 例 1

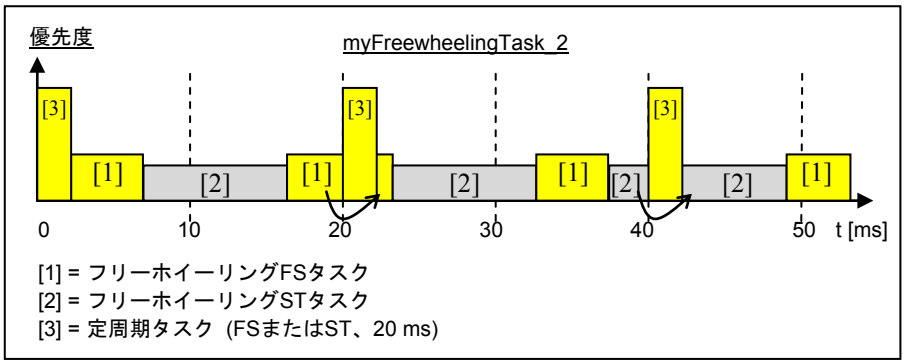


図: 例 2

定周期タスク:

定周期タスクは、定義可能な周期長の間に 1 回だけ実行されます。実行の間隔はさまざまです。実行が周期長の間に完了しなかった場合、オペレーティングステータス「Resource in RUN condition with task fault」が表示されます。周期長はタスクが実行される周期を決定します。また、副次的な優先度の役

割も果たします。タスクの優先度が同じ場合は、周期長の短いタスクが先に実行されます。値の範囲:

- ST タスク: 2 ms～71 min
- FS タスク: 4 ms～71 min

3 つの例を次に示します。

1. 周期長が 10 ms の定周期タスク 1 つ
2. 許可済みの呼び出しのある定周期タスク (優先度高)
   1 つと 2 つ目の定周期タスク (優先度低)
3. 未許可の呼び出しのある定周期タスク (優先度低)
   1 つと 2 つ目の定周期タスク (優先度高)

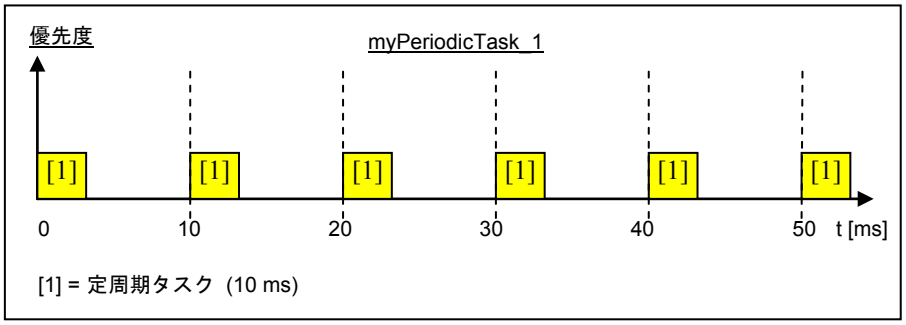


図: 例 1

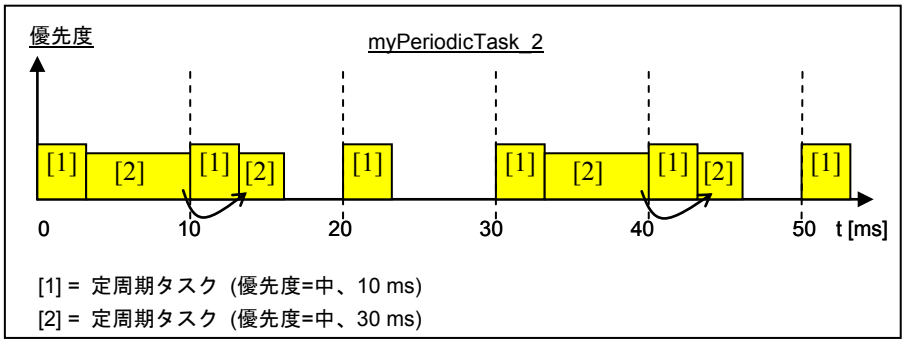


図: 例 2

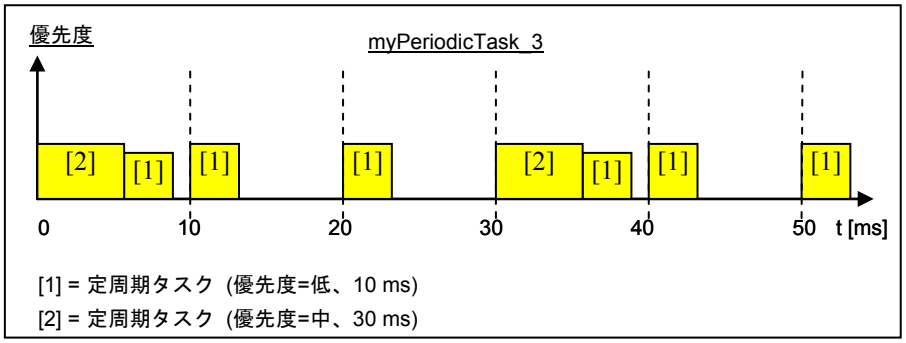


図: 例 3

イベント起動タスク:

イベント起動タスクは、定義可能なイベント (BOOL 値変数での立上り／立下り、リソースの開始／停止など) が発生したときに実行されます。

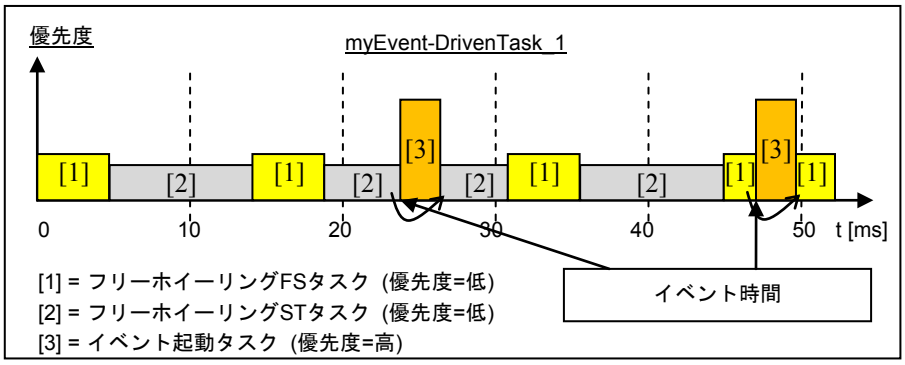


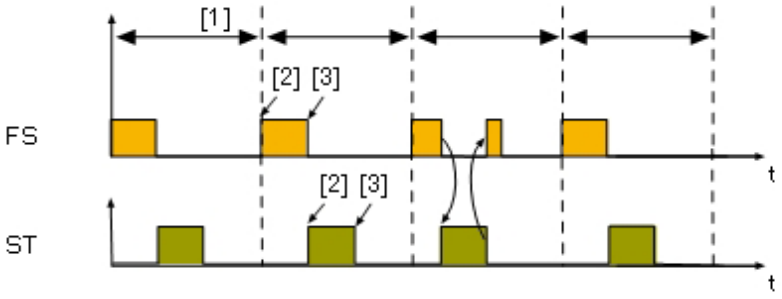
図: 例 1

10.5 タスクのデフォルトコンフィグレーション

☞ ほとんどのプロジェクトはデフォルトコンフィグレーションで十分なため、ユーザが「タスク」の問題に対応する必要はありません。

デフォルトコンフィグレーションでは、デバイス上で FS プログラムが最初に実行され、次に ST プログラムが実行されるよう設計されています。FS プログラムは常に同じ順序で実行されます。これはST プログラムも同じです。

FS プログラムと ST プログラムは 50 ms の周期長の間に 1 回実行されます。



[1] 周期長、[2] ブローバル変数と入力 (PII) および出力 (PIO) のプロセスイメージの読み取り、[3] グローバル変数の出力と出力 (PIO) のプロセスイメージ

☞ たとえば、FS プログラムがシステムファンクションの実行待ち状態となる状況が発生することがあります。この場合、FS プログラムの実行は中断され、ST プログラムが実行されます。

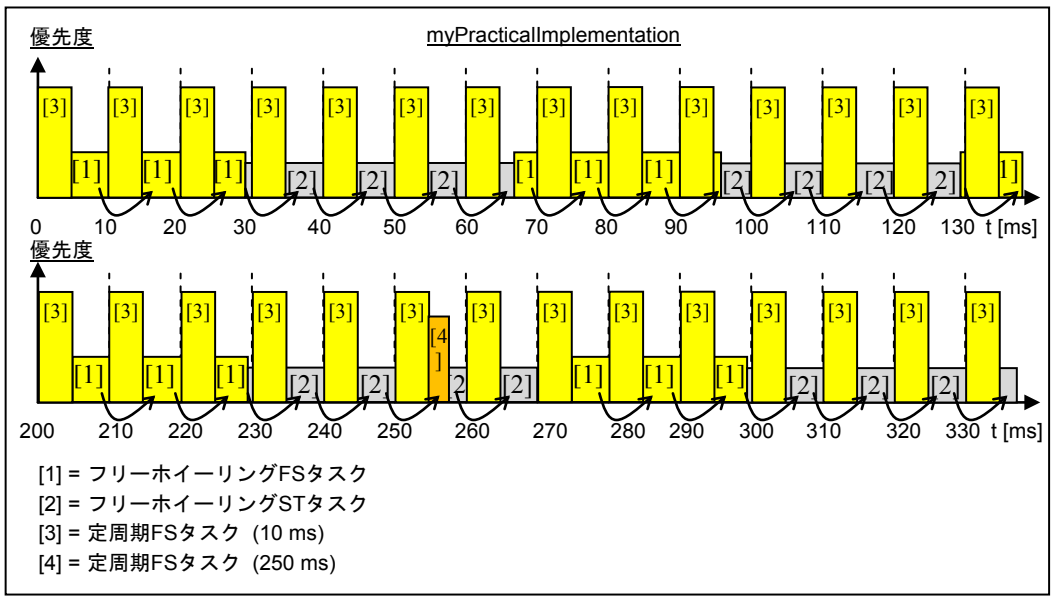
10.6 ネームスペース内のタスク情報

各タスクの情報はプロジェクトのネームスペースにあり、それらは変数表示内および OPC サーバで呼び出すことができます。

- 現在のタスク実行時間:
  アドレス:
  prj::<**プロジェクト名**>.Exec:<**リソース名**>.Task:ProcessingTime:<**タスク名**>.Current
  データ型: UDINT
- リソースが起動してからの最小タスク実行時間
  アドレス:
  prj::<**プロジェクト名**>.Exec:<**リソース名**>.Task:ProcessingTime:<**タスク名**>.Minimum
  データ型: UDINT
- リソースが起動してからの最大タスク実行時間
  アドレス:
  prj::<**プロジェクト名**>.Exec:<**リソース名**>.Task:ProcessingTime:<**タスク名**>.Maximum
  データ型: UDINT
- タスクステータス:
  アドレス:
  prj::<**プロジェクト名**>.Exec:<**リソース名**>.Task:State:<**タスク名**>.
  データ型: UDINT
  Key:
  1 =タスクは停止状態
  2 =タスクは実行状態

10.7 実践的な実装

FS/ ST	番号	タスクのタイプ	優先度	アプリケーション
FS	1	フリーホイール FS タスク	低	瞬時の処理を必要としない安全アプリケーション用の FS-POU。例: - 非常停止、- グラブワイヤ、- 安全扉、- 回転ドア
	2	定周期 FS タスク	低	250 ms、高速点滅レート 2 Hz
				500 ms、平均点滅レート 1 Hz
	1	定周期 FS タスク	中	10 ms、瞬時の処理を必要とする安全アプリケーション用。例: - ライトカーテン、- ライトグリッド、- ミューティング、- スキャナ、- 安全カメラ
	2	イベント起動 FS タスク	中	瞬時の処理を必要とする安全アプリケーション用の中断処理
	3	イベント起動 FS タスク	高	瞬時の処理を行うことが非常に重要な安全アプリケーション用の中断処理
	9	すべての FS タスクの総数		
ST	1	フリーホイール ST タスク	低	ST POU
	2	定周期 ST タスク	低	250 ms、高速点滅レート 2 Hz
				500 ms、平均点滅レート 1 Hz
	1	定周期 ST タスク	中	1 s: - 遅い点滅レート 0.5 Hz、- 診断用 ST-POU
	2	イベント起動 ST タスク	中	瞬時の処理を必要とする一般アプリケーション用の中断処理
	3	イベント起動 ST タスク	高	瞬時の処理を行うことが非常に重要な一般アプリケーション用の中断処理
	18	すべてのタスクの総数		



11 I/Oマッピング

すべての情報 (内部／外部プロセス信号) にとってハードウェアへの割り当て (マッピング) が 1 つあるだけで十分なことから、すべてのユーザが情報に直接アクセスできます。これは、一般プロセス信号と安全関連信号の両方に当てはまります。これにより柔軟性が高まりまが、このことは、後で変更を加える場合に特に重要です。

- プログラム内でプロセス信号の装置 ID が使用されていて、それとは別のデバイスに接続する場合は、マッピングテーブルだけを変更します。これによりデータの一貫性を容易に保つことができ、特に安全関連プログラムにおいて、プログラムのチェックサムを維持できます。
- ハードウェアは ID によって管理できるため、メーカは機械のオプションやタイプの処理が非常に簡単になります。
- シンプルで、一度で済む再設定により、処理がより簡単になります。配布可能なアプリケーションに集中型マッピングテーブルを使用している場合でも、さまざまな割り付けと一致させるための追加の作業は不要です (プログラム、診断、視覚化)。
- 試運転または変更後の管理作業がなくなり、試運転リストを一致させるという非常に生産性のない作業が不要になりました。

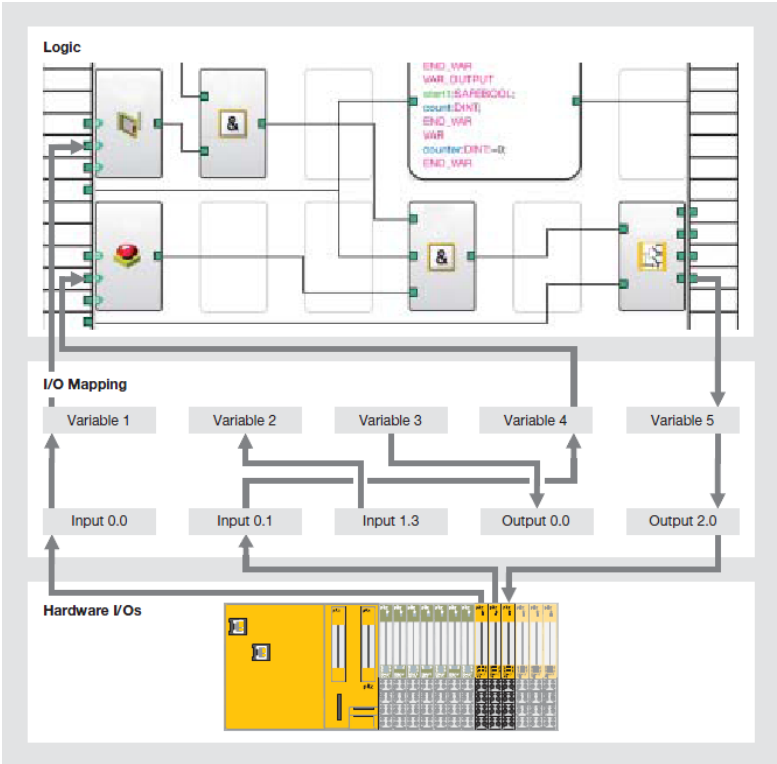


図: I/O マッピング - ハードウェアマッピング

プロジェクト全体の集中型マッピングテーブルにより、次のことが実現されます。

- シンプルな指向性
- アプリケーション (ユーザプログラム／診断／視覚化など) と周辺機器との間のデータの一貫性

重要なことは、設定を 1 つの集中型制御システムで行ったか、あるいは複数の制御システムで行ったかにかかわらず、1 つの集中型マッピングテーブルが 1 つのオートメーションプロジェクトで使用されることです。プロジェクトで使用される制御システムやパッシブバスインタフェースの数に関係なく、ユーザには常に、分散型システムを集中表示したオーバービューが表示されます。さらに、集中型マッピングテーブルは、これまで非常に複雑だった通信機能に対する明確なソリューションとなります。下の図は、3 つの制御システムを持つアプリケーションに基づいた、集中型マッピングテーブルのオペレーションの原理モードを示しています。その他の利点: オートメーションシステム PSS 4000 を使用してオートメーション機能を複数の制御システムに分ける手法は、1 つの制御システムを使用した以前のソリューションと同様にシンプルです。

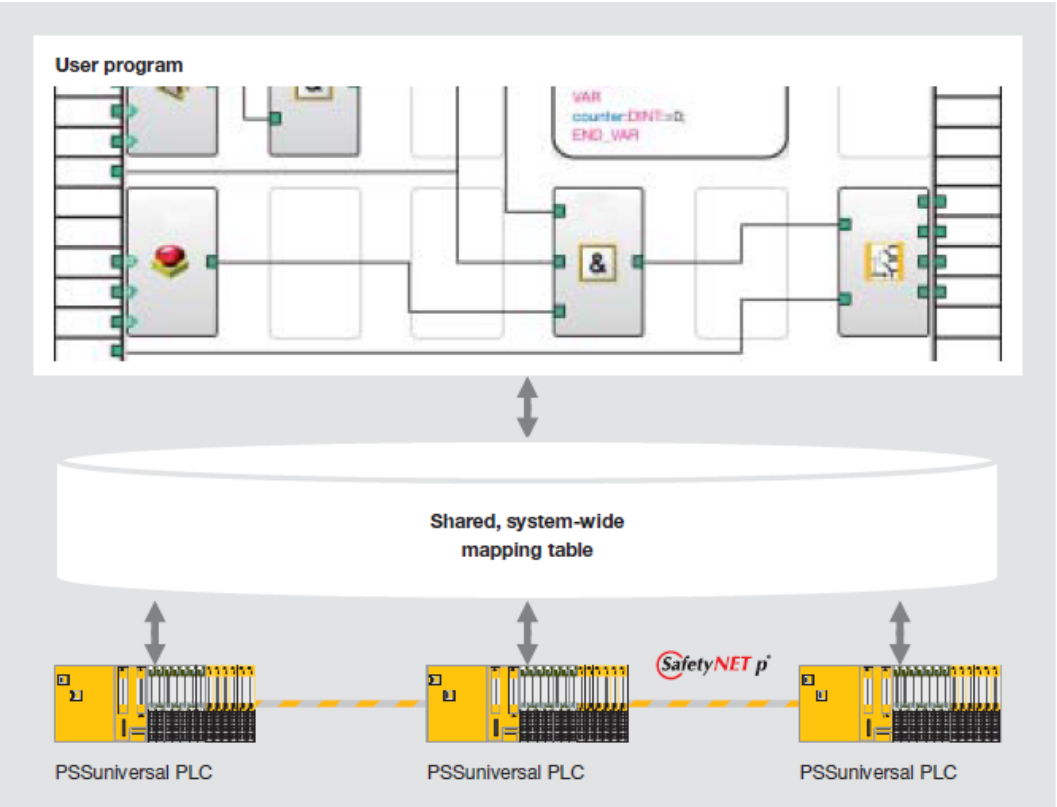


図: プロジェクト全体の集中型マッピングテーブル

空白ページ

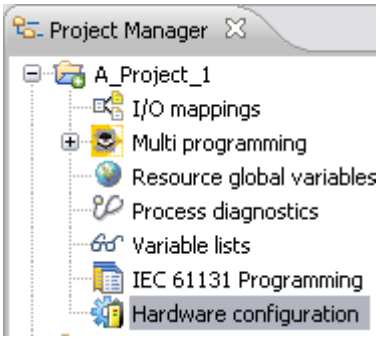
12 PSS 4000 のコンフィグレーション (リソース、タスク、I/Oマッピング)

POU を作成し、変数を宣言したら、次にハードウェアリソースとタスクを定義する必要があります。リソースが計画され、タスクが割り付けられたら、I/Oマッピングが実行されます。つまり、キーワード AT %I または AT %Q が指定された変数が、関連するデバイスの絶対 I/O アドレスに割り付けられます。

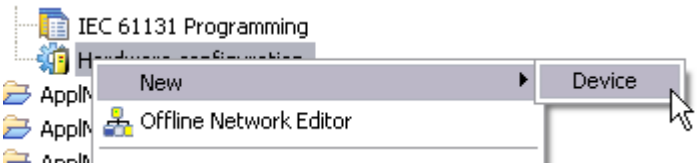
12.1 手順

12.1.1 デバイスとI/Oモジュールの定義 (リソース)

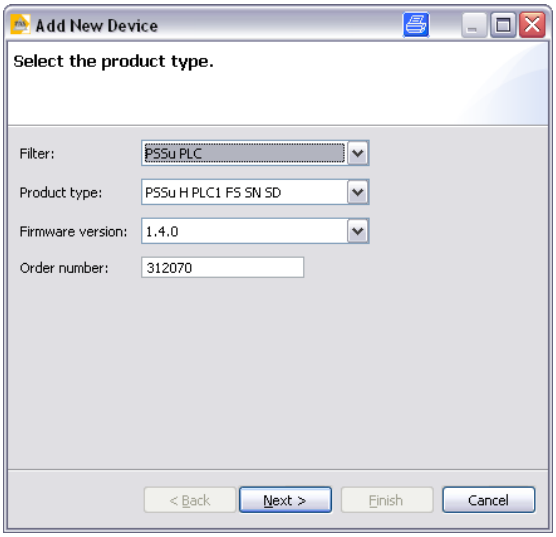
❶ [**Project Manager**] ウィンドウの [**Hardware configuration**] を右クリックします。



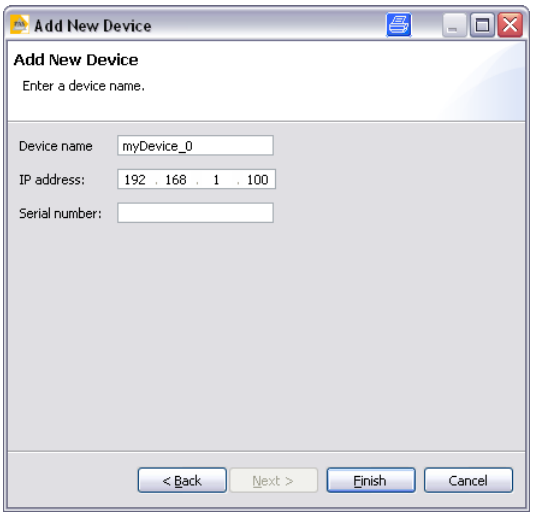
❷ [**New**] ⇒ [**Device**] を選択します。



❸ デバイス [**PSSu PLC**] を選択します。

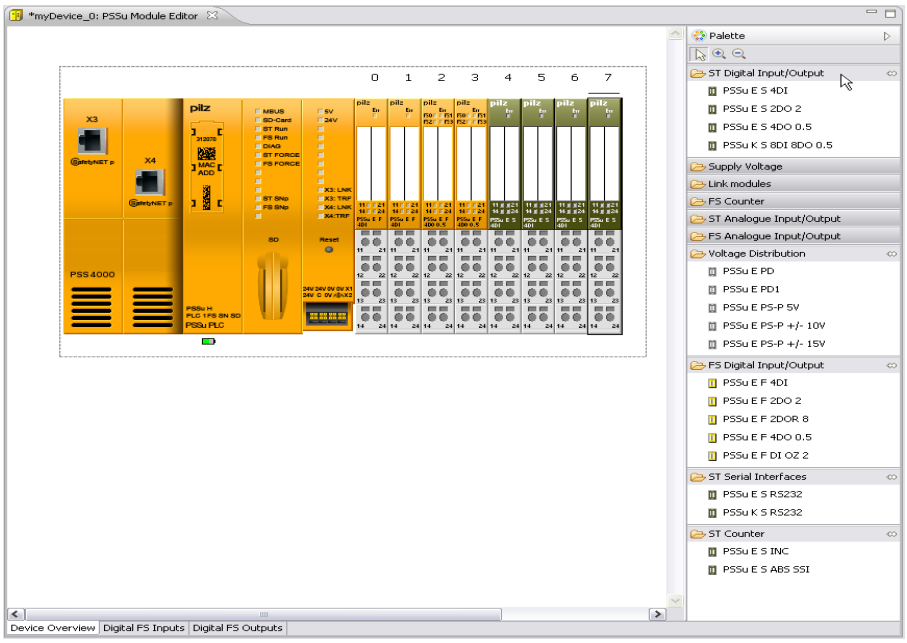


❹ Next > をクリックして次のダイアログを表示します。



❺ 必要なデバイス名と IP アドレスを入力します。

❻ 入力を確認し、 Finish をクリックしてウィンドウを閉じます。



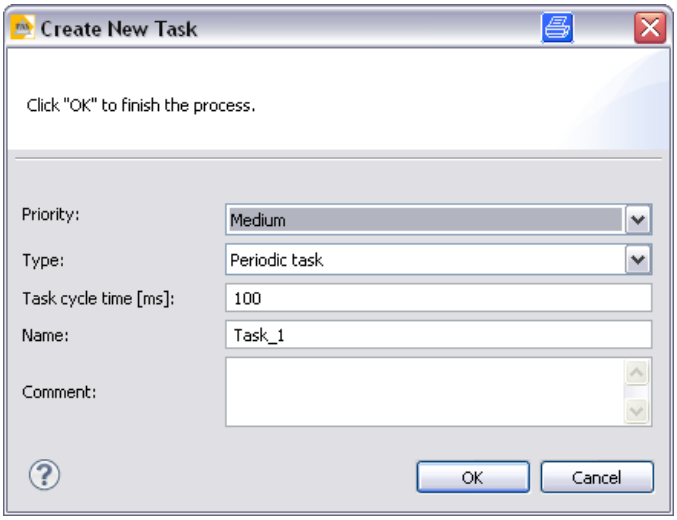
❼ モジュールパレットから必要なモジュールを選択します。

12.1.2 タスクの定義

❶ **[Project Manager]** ウィンドウでリソースを右クリックします。たとえば、**[myDevice_0]** で **[ST resource]** を右クリックします。

❷ **[New]** ⇒ **[New Task]** を選択します。

❸ タスクの優先度、タイプ、および値を選択し、タスク名とコメント (必要な場合) を入力します。

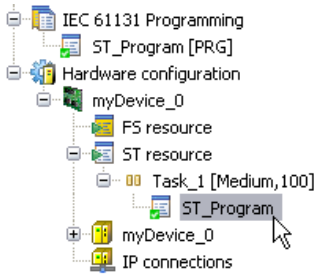


❹ 入力を確認し、 OK をクリックしてウィンドウを閉じます。

12.1.3 展開 (POUのハードウェアへの割り付け)

展開とは、ソフトウェアをハードウェアに配布するためのプロセスのことです。保守の行き届いた環境で信頼性が高く安全なオペレーションを確実に行えること必要です。

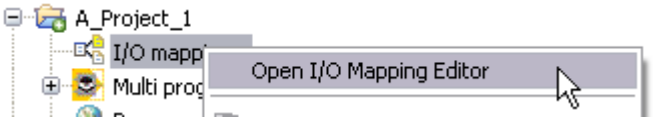
❶ **[Project Manager]** ウィンドウで、目的の POU (**[ST_Program[PRG]**])をクリックし、この POU を目的のデバイス (**[myDevice_0]**) までドラッグします。



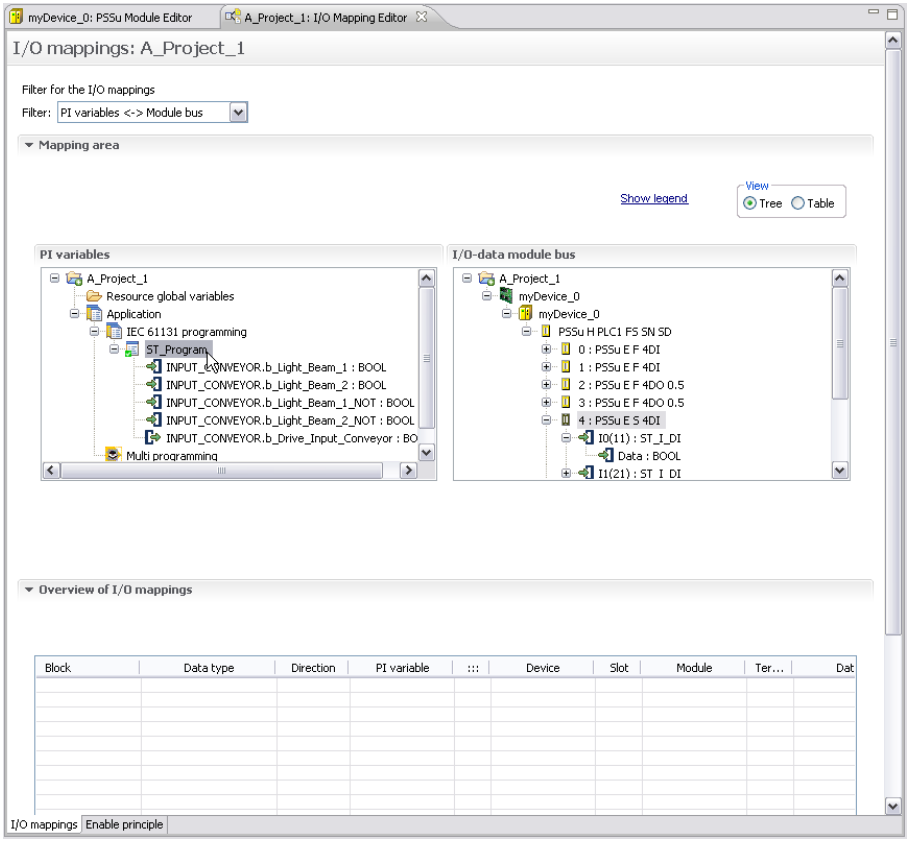
12.1.4 I/Oマッピングの定義

❶ [**Project Manager**] ウィンドウの [**I/O mappings**] を右クリックします。

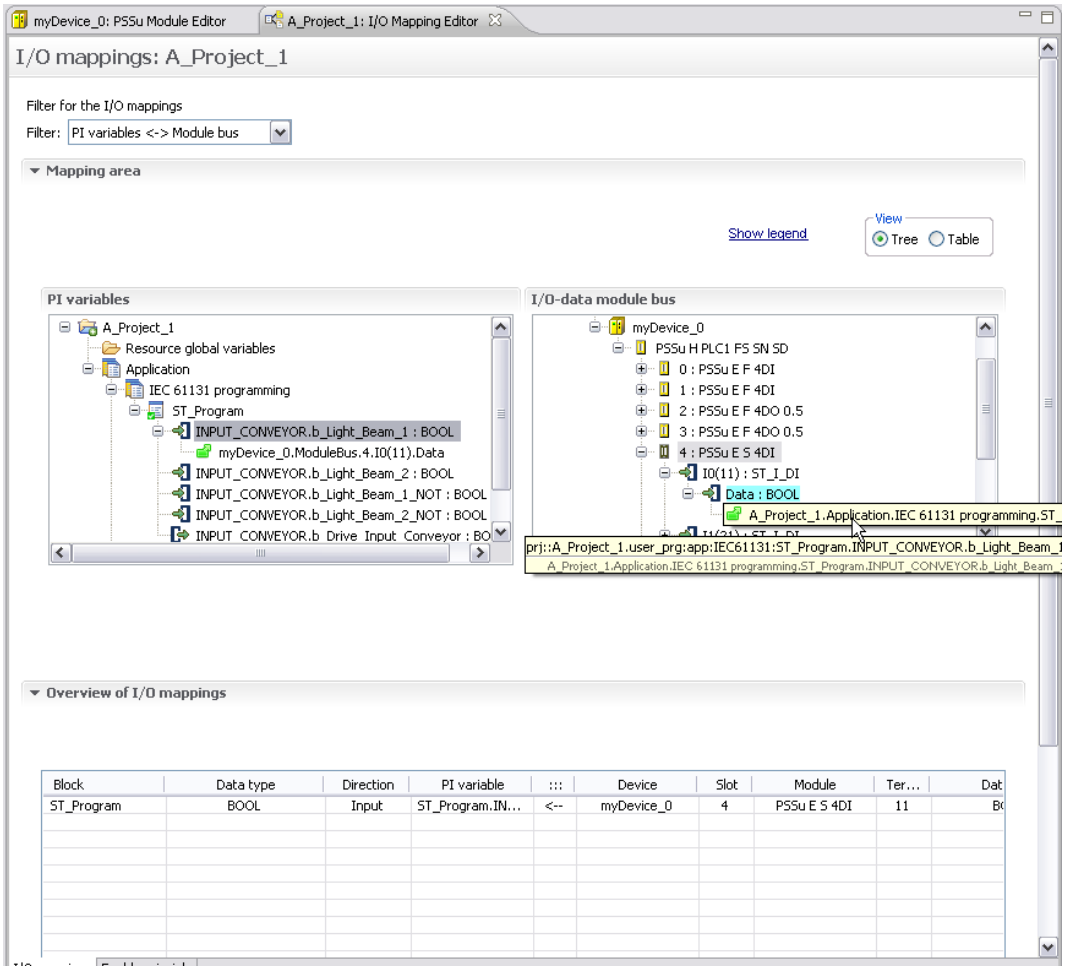
❷ [**Open I/O Mapping Editor**] を選択します。



❸ デバイス [**PSSu PLC**] を選択します。



❹ 左側のウィンドウで変数をクリックし、右側のウィンドウの関連するハードウェアアドレスまでドラッグして、変数を絶対ハードウェアアドレスにマッピングします。


myDevice_0: PSSu Module Editor
A_Project_1: I/O Mapping Editor
I/O mappings: A_Project_1
Filter for the I/O mappings
Filter: PI variables <-> Module bus
Mapping area
Show legend
View
Tree
Table
PI variables
A_Project_1
Resource global variables
Application
IEC 61131 programming
ST_Program
INPUT_CONVEYOR.b_Light_Beam_1 : BOOL
myDevice_0.ModuleBus.4.IO(11).Data
INPUT_CONVEYOR.b_Light_Beam_2 : BOOL
INPUT_CONVEYOR.b_Light_Beam_1_NOT : BOOL
INPUT_CONVEYOR.b_Light_Beam_2_NOT : BOOL
I/O-data module bus
myDevice_0
PSSu H PLC1 FS SN SD
0 : PSSu E F 4DI
1 : PSSu E F 4DI
2 : PSSu E F 4DO 0.5
3 : PSSu E F 4DO 0.5
4 : PSSu E S 4DI
IO(11) : ST_I_DI
Data : BOOL
A_Project_1.Application.IEC 61131 programming.ST_
Overview of I/O mappings
Block
Data type
Direction
PI variable
:::
Device
Slot
Module
Ter...
ST_Program
BOOL
Input
ST_Program.IN...
<--
myDevice_0
4
PSSu E S 4DI
11

13 その他

13.1 キーワード

キーワードとは、その用途が一意に定義されている文字列です。その用途以外に使用できません。大文字と小文字は区別されません。予約済みキーワードは次の通りです。

- 基本データ型と派生データ型の名前 (たとえば BOOL)
- 一般ファンクションの名前とそれらの入出力パラメータ
- ユーザによって作成されたファンクションの名前
- スタンダードファンクションブロックの名前とそれらの入出力パラメータ
- ユーザによって作成されたファンクションブロックタイプの名前
- IL プログラミング言語のオペレータの名前 (IL オペレータは、IL だけでなく、他のすべてのプログラミング言語でプログラミングする際のキーワードとなります)。
  - ACTION...END_ACTION
  - ARRAY...OF
  - AT
  - CASE...OF...ELSE...END_CASE
  - CONFIGURATION...END_CONFIGURATION
  - CONSTANT
  - EN, ENO
  - EXIT
  - FALSE
  - F_EDGE
  - FOR...TO...BY...DO...END_FOR
  - FUNCTION...END_FUNCTION
  - FUNCTION_BLOCK...END_FUNCTION_BLOCK
  - IF...THEN...ELSIF...ELSE...END_IF
  - INITIAL_STEP...END_STEP
  - NOT, MOD, AND, XOR, OR
  - PROGRAM...WITH...
  - PROGRAM...END_PROGRAM
  - R_EDGE
  - READ_ONLY, READ_WRITE
  - REPEAT...UNTIL...END_REPEAT
  - RESOURCE...ON...END_RESOURCE
  - RETAIN,
  - RETENTIVE
  - RETURN
  - STEP...END_STEP
  - STRUCT...END_STRUCT
  - TASK
  - TRANSITION...FROM...TO...END_TRANSITION
  - TRUE
  - TYPE...END_TYPE
  - VAR...END_VAR
  - VAR_INPUT...END_VAR
  - VAR_OUTPUT...END_VAR
  - VAR_IN_OUT...END_VAR
  - VAR_TEMP...END_VAR
  - VAR_EXTERNAL...END_VAR
  - VAR_ACCESS...END_VAR
  - VAR_CONFIG...END_VAR
  - VAR_GLOBAL...END_VAR
  - WHILE...DO...END_WHILE
  - WITH

13.2 コメント

コメントは、プロジェクトを詳細に記述する場合に使用する説明のテキストです。すべての Unicode 文字を使用できます (UTF-8 形式)。コメントは、たとえば新しいブロックを作成するときなど、PAS4000 の数多くの場面で入力できます。テキスト言語 (たとえば IL、STL) を使ってプログラミングする場合、コメントはプログラムコードの中に入力できます。

コメントは「(*」で始まり「*)」で終わるか、または「//」で始まり行の最後で終わります。ネストされたコメントも使用できます。
例: (*(*コメント*)*)

13.3 スペースの使用

ユーザプログラムでは任意の場所に 1 つ以上のスペースを入力することができます。ただし、次の場所は例外です。

- キーワード内
- 定数内
- 識別子内
- 「列挙」データ型のファンクション内
- 「%I*」および「%Q*」内
- 関連する区切り記号 (たとえば「:=」) の間

スペースには、タブや改行などの空白を作成する制御文字も含まれます。

13.4 区切り記号

区切り記号は、一部がコンテキストに依存する意味を持つ定義済みの特殊文字です。たとえば、+記号は数字の接頭辞として使用したり、IL プログラミング言語で加算オペレータとして使用したりできます。

13.5 プログラミング言語

13.5.1 IL - インストラクションリスト

下の図は、IL 命令の基本構造を示しています。

オペレータ／修飾子	変数	//コメントまたは (*コメント*)

または

ラベル:	//コメントまたは (*コメント*)

下のテーブルに関するメモ:

- 「ANY」データ型: すべての基本データ型および派生データ型。
- 「ANY_BIT」データ型: BOOL、BYTE、WORD、DWORD、LWORD。
- 「ANY_NUM」データ型: BOOL、BYTE、WORD、DWORD、LWORD 以外のすべての基本データ型。

13.5.1.1 IL命令セット

オペレータ	修飾子	変数 (許可されたデータ型)	説明
LD	N	ANY	バイナリ命令 (BOOL) の開始、またはオペランドの値にアキュムレータを設定
ST	N	ANY	現在の結果をオペランドの場所に格納
S		BOOL	条件が満たされた場合にオペランドを設定
R		BOOL	条件が満たされた場合にオペランドをリセット
AND	N, (	ANY_BIT	AND 命令
OR	N, (	ANY_BIT	OR 命令
XOR	N, (	ANY_BIT	EXCLUSIVE OR 命令
NOT		ANY_BIT	否定
ADD		ANY_NUM	加算
SUB		ANY_NUM	減算
MUL		ANY_NUM	乗算
DIV		ANY_NUM	除算
GT		ANY	比較: >
GE		ANY	比較: >=
EQ		ANY	比較: =
NE		ANY	比較: <>
LE		ANY	比較: <=
LT		ANY	比較: <
JMP	C, CN	ラベル	ラベルへの分岐
CAL	C, CN	ファンクションブロックインスタンス名	ファンクションまたはファンクションブロックの呼び出し
RET	C, CN	-	ブロックの終了
)		-	括弧内の命令の終了

13.5.1.2 プログラムの例

スイッチオンディレイ時間「TON」が経過したら、出力の点滅のステータスが反転します(「0」は「1」に、「1」は「0」になる)。その結果、「出力の点滅」変数は「時間値」変数に記述されているパルスと停止時間で点滅します。

```
FUNCTION_BLOCK Flasher
VAR INPUT
    t_time_value:TIME;
END_VAR
VAR OUTPUT
    b_Flasher_output:BOOL;
END_VAR

VAR
    b_End_time:BOOL;
    T_delayON :TON;
END_VAR
VAR_TEMP
    b_Start:BOOL;
END_VAR
```

```
    LDN b_End_time (*If time is over
                    => switch off b_Start*)
ST b_Start

    CAL T_delayON( (*Standard-FB “TON” *)
    in := b_Start,
    pt := t_time_value
    q => b_End_time
    )

    LDN b_End_time (*If time is over
                    => negate b_Flasher_output *)
    JMPC End
    LD b_Flasher_output
    STN b_Flasher_output
    End:
    END_FUNCTION_BLOCK
```

```
PROGRAM myFirstProgram
VAR
    FB_Flasher_1Hz:Flasher;
    b_Flasher_output_1Hz BOOL;
    FB_Flasher_2Hz:Flasher;
    b_Flasher_output_2Hz BOOL;
END_VAR
```

```
    CAL FB_Flasher_1Hz ( (*1Hz Flasher*)
    t_time_value := T#500ms,
    b_Flasher_output => b_Flasher_output_1Hz
    )
    CAL FB_Flasher_2Hz ( (*2Hz Flasher*)
    t_time_value := T#250ms,
    b_Flasher_output => b_Flasher_output_2Hz
    )
    END_PROGRAM
```

13.5.2 STL - ストラクチャードテキスト

ST 命令の一般的なプロパティは次の通りです。

- ST 命令はセミコロン「;」で区切る
- ST 命令の最後はスペースで定義されるため、STL には行ベースのエントリ形式はない
- コメントは IL と同じ形式で入力する: (*コメント*) または//コメント

下のテーブルに関するメモ:「ANY」データ型は、すべての基本データ型 (BOOL、INT など) と派生データ型を意味します。

13.5.2.1 STキーワード

キーワード	キーワード拡張	例	説明
:=		**a := 10;**	割り付け
		FBName(Par1:=1, Par2:=2);	ファンクションまたはファンクションブロックの呼び出しにキーワードは不要
RETURN			呼び出し元の POU に戻る

IF	THEN
	ELSEIF
	ELSE
	END_IF
		IF a<b THEN
		c := 1;
		ELSEIF a=b THEN
		c := 2;
		ELSE
		c := 3;
		END_IF;
			代替に分岐

CASE	OF ELSE END_CASE	**CASE a OF** **1:c := 100;** **2:c := 101;** **ELSE** **c := FBName(Par1:=1, Par2:=2);** **END_CASE;**	値「f」に基づくステートメントブロックの選択
FOR	TO BY DO END_FOR	**FOR a=1 TO 100 BY 5** **DO a := a-5;** **END_FOR;**	ループ文: 開始および終了条件のある1つ以上の文の反復
WHILE	DO END_WHILE	**WHILE a>1** **DO a := a-1;** **END_WHILE;**	ループ文: 終了条件のある 1 つ以上の文の反復
REPEAT	UNTIL END_REPEAT	**REPEAT a := a*10 UNTIL a>1000;** **END_REPEAT;**	ループ文: ダウンストリーム終了条件のある1つ以上の文の反復
EXIT			ループのブレーク
;			空白の文

13.5.2.2 STオペレータ

オペレータ	説明
( )	括弧
+	加算
-	減算
*	乗算
/	除算
**	累乗
<	「<」比較
>	「>」比較
<=	「<=」比較
>=	「>=」比較
=	「=」比較
<>	「<>」比較
& AND	BOOL 値 AND 命令
OR	BOOL 値 OR 命令
XOR	BOOL 値 EXCLUSIVE OR 命令
NOT	否定

空白ページ

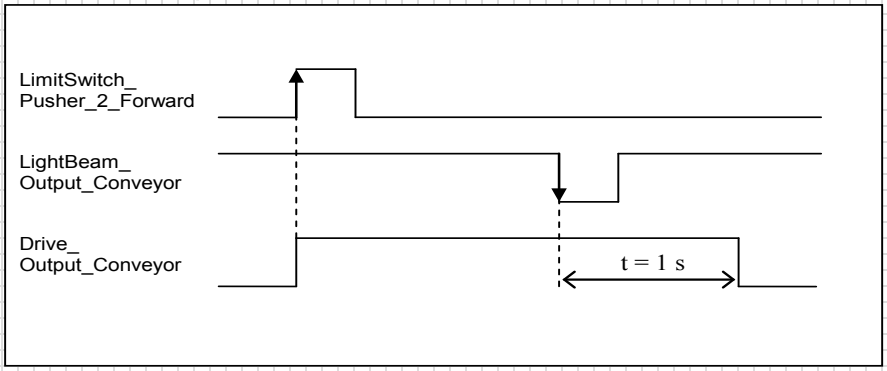
14 プログラミング演習 1

I. 演習の概要

出力コンベヤは、前面のリミットスイッチ「LimitSwitch_Pusher_2_Forward」によってスイッチをオンにし、出力コンベヤの光線装置が遮光された後で時間制御を使ってオフにする必要があります (図を参照)。

II. 図

図 1.14: プログラム構造



III. プログラム構造

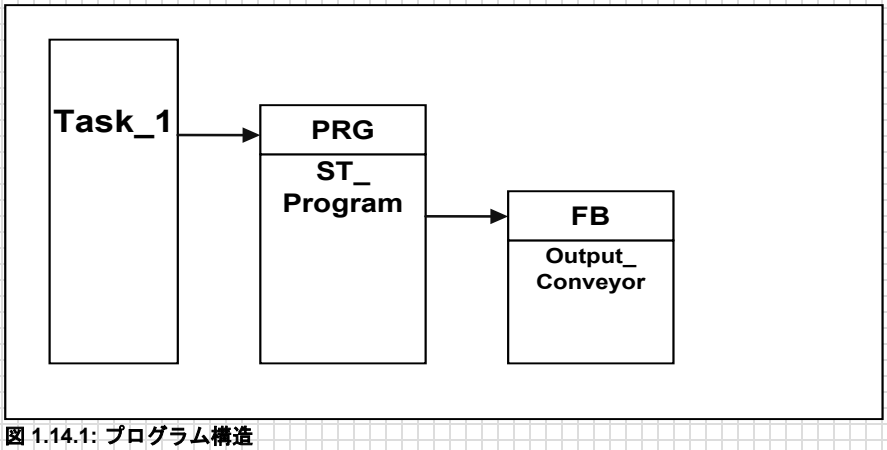


図 1.14.1: プログラム構造

IV. I/O マッピングの変数

変数名／ データ型	デバイス名	スロット 番号	ハードウェア アドレス	センサ／ アクチュエータ
I_b_LimitSwitch_Pusher_2_ Forward BOOL	Geraet_1	2	I2 (14)	リミットスイッチ: 右側の方は前進 位置
I_b_LightBeam_Output_ Conveyor BOOL	Geraet_1	2	I1 (21)	出力コンベヤ上の 光線装置
Q_b_Drive_Output_Conveyor BOOL	Geraet_1	3	O2 (14)	出力コンベヤ上の ドライブの コンタクタ

V. 必要な一般ファンクション

インスタンス名	スタンダード **FB**	キー
ft_LB_falling_Ed	F_TRIG	立下りでトリガ
T_Conveyor_Delay	TON	スイッチオンディレイ
…	…	...

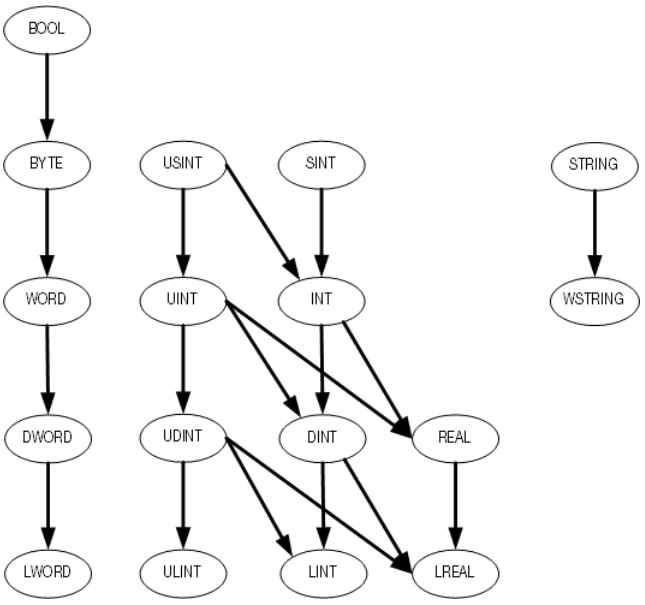
15 型変換

15.1 Implicit型変換

代表的なブロック (ファンクションまたはファンクションブロック) では、入出力パラメータに特定の基本データ型を指定する必要があります。パラメータのデータ型が指定と一致しない場合は、自動型変換を行えるかどうかのテストが実行されます。これを「Implicit 型変換」と言います。

型変換によって情報が失われない場合は、型変換が常に実行されます。型変換によって情報が失われる場合は、プログラミング中にエラーメッセージが表示されます。この場合、ユーザは入出力パラメータのデータ型を変更するか、または Explicit 型変換を実行する必要があります。

この図は、考えられるすべての Implicit 型変換を示しています。

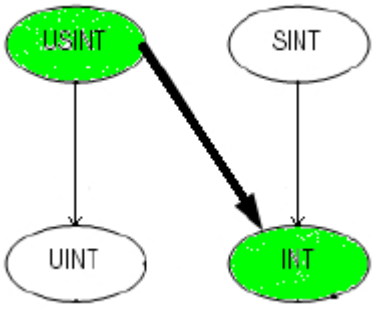


オーバーロードされたファンクションの場合、入力パラメータに複数のデータ型を指定することができます。ただし、すべての入力パラメータが同じデータ型を含んでいる必要があります。ファンクションの結果に互換性のあるデータ型がある場合は、異なるデータ型も許容されます。ファンクションの結果のデータ型の定義は、次の規則に基づきます。

- 入力パラメータのデータ型が同じであれば、ファンクションの結果も同じデータ型となる。
- 入力パラメータのデータ型が異なる場合、これらのデータ型を組み合わせることのできるデータ型が結果に含まれている必要がある。これには、上の図を使用でき、矢印の向きに沿うチャンネルのみが許可される。

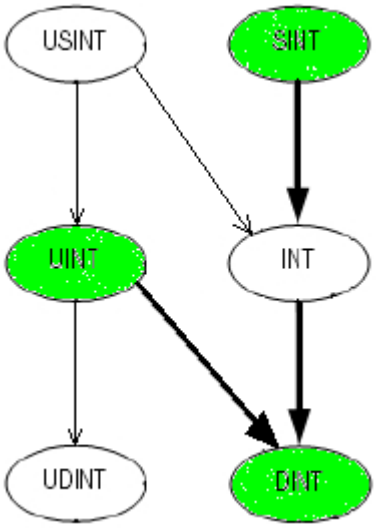
例 1:

入力パラメータのデータ型は USINT と INT なので、ファンクションの結果のデータ型は INT になります。



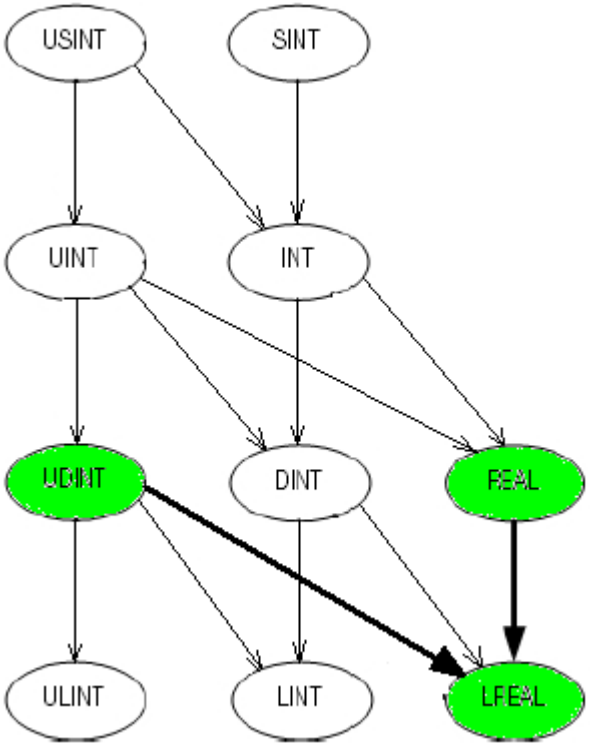
例 2:

入力パラメータのデータ型は UINT と SINT なので、ファンクションの結果のデータ型は DINT になります。



例 3:

入力パラメータのデータ型は UDINT と REAL なので、ファンクションの結果のデータ型は LREAL になります。



☞

ファンクションの結果に互換性のあるデータ型が見つからない場合は、入力パラメータのデータ型は許可されません。プログラミング中に、関連するエラーメッセージが表示されます。この場合、ユーザは入力パラメータのデータ型を変更するか、または Explicit 型変換を実行する必要があります。

15.2 Explicit型変換

基本データ型はすべて別の基本データ型に変換できます。この目的で、型変換ファンクションが利用できます。BOOL から BYTE など、一部の型変換は情報を失うことなく実行できます。一方、BYTE から BOOL など、情報が失われる型変換もあります。

情報が失われない型変換は、ユーザが明示的に実行する必要はありません。たとえば、ファンクションの入力パラメータが必要なデータ型と一致しないため型変換が必要になる場合は、変換は自動的 (暗黙的) に実行されます。

下のテーブルは、考えられるすべての型変換と、それらがデータを損失することなく実行できるかどうかを示しています。

Explicit 型変換の例:

BYTE_TO_**INT**
WORD_TO_**INT**
DWORD_TO_**INT**
LWORD_TO_**INT**
UINT_TO_**SINT**
UDINT_TO_**SINT**
ULINT_TO_**SINT**
SINT_TO_**UINT**
INT_TO_**UINT**
DINT_TO_**INT**
LINT_TO_**INT**
REAL_TO_**INT**
LREAL_TO_**INT**
TIME_TO_**INT**
TIME_OF_DAY_TO_
DATE_TO_**INT**
DATE_AND_TIME_TO_

Datentyp	BOOL	BYTE	WORD	DWORD	LWORD	USINT	UINT	UDINT	ULINT	SINT	INT	DINT	LINT	REAL	LREAL	TIME	TIME_OF_DAY	DATE	DATE_AND_TIME
BOOL	■																		
BYTE		■																	
WORD			■																
DWORD				■															
LWORD					■														
USINT						■													
UINT							■												
UDINT								■											
ULINT									■										
SINT										■									
INT											■								
DINT												■							
LINT													■						
REAL														■					
LREAL															■				
TIME																■			
TIME_OF_DAY																	■		
DATE																		■	
DATE_AND_TIME																			■

黒　　= 型変換は不要
グレー = 情報が失われない型変換で、暗黙に実行される
白　　= 情報が失われる型変換で、明示的に実行する必要あり

```
WORD_TO_INT(
in := w_myWordVariables,
)
ST i_myIntVariables
```

16 演習: 選択肢問題

問題 1.

POU: インスタンス化が必要な POU はどれですか？

	正解
1. プログラム	○
2. ファンクションブロック	○
3. ファンクション	○
4. すべての POU	○

問題 2.

問題 1 の POU はなぜインスタンス化が必要ですか？

	正解
1. データ構造 (変数) を複製するため。そうすることにより、各インスタンスが異なる値を取得することができる	○
2. プログラムコードを複製するため。そうすることにより、各インスタンスが異なるファンクションを持つことができる	○
3. プログラムコードとデータ構造 (変数) を複製するため。そうすることにより、各インスタンスが異なる値を取得し、異なるファンクションを持つことができる	○
4. プログラムコードとデータ構造が変更されないようにするため。そうすることにより、各インスタンスが同じ値を取得し、同じファンクションを持つことができる	○

問題 3.

リソース: 1 つの PAS4000 プロジェクト

(制御プログラム) はいくつのリソースに割り付けることができますか？

	正解
1. 1 つの FS リソース	○
2. 1 つの ST リソース	○
3. 1 つの FS リソースと ST リソース	○
4. 複数の FS リソースと ST リソース	○

問題 4.

1 つの FS リソースおよび ST リソース上で、異なる優先度のタスクをいくつ実行できますか？

	正解
1. 6 タスク (優先度 高、中、低は任意の数だけ割り付けることができる)	○
2. 6 タスク (2 タスク = 優先度 高、2 タスク = 優先度 中、2 タスク = 優先度 低)	○
9 タスク (優先度 高、中、低は任意の数だけ割り付けることができる)	○
4. 9 タスク (3 タスク = 優先度 高、3 タスク = 優先度 中、2 タスク = 優先度 低)	○

問題 5.

データ型からの Implicit 型変換でプログラミングエラーが生じないのはどれですか？

	正解
1. LD SINT ST USINT	○
2. LD SINT ST DINT	○
3. LD SINT ST UDINT	○
4. LD SINT ST WORD	○

問題 6.

プログラミングエラーが生じるため、データ型の Explicit 型変換を実行する必要があるのはどれですか？

	正解
1. WORD_TO_INT	○
2. UINT_TO_INT	○
3. UDINT_TO_DINT	○
4. DINT_TO_TIME	○

17 ライブラリ

再利用可能なプログラム構成要素やデバイスの集合 (たとえば、ソフトウェアツール内など)

17.1 ライブラリマネージャ

すべてのライブラリが表示され、整理されるウィンドウ

17.1.1 パレット

ライブラリファンクションを編成したもの。迅速なアクセスを可能にする

17.1.2 ピルツライブラリ

ピルツから購入したファンクションを含むライブラリ

17.1.3 システムライブラリ

ピルツが提供する再利用できるファンクションを含むライブラリ

17.1.4 ユーザライブラリ

ユーザが作成したファンクションを含むライブラリ

18 プログラミング演習 2

I. 演習の概要

2 つのアナログ信号 (Pot. 1 と Pot. 2) を比較し、両者の入力電圧の違いをインジケータに表示します。

例 1:

Pot_1 = 7 V で Pot_2 = 5 V であれば、インジケータ = 2 V

例 2:

Pot_1 = 1 V で Pot_2 = 8 V であれば、インジケータ = 7 V

例 3:

Pot_1 = 5 V で Pot_2 = 5 V であれば、インジケータ = 0 V

II. プログラム構造

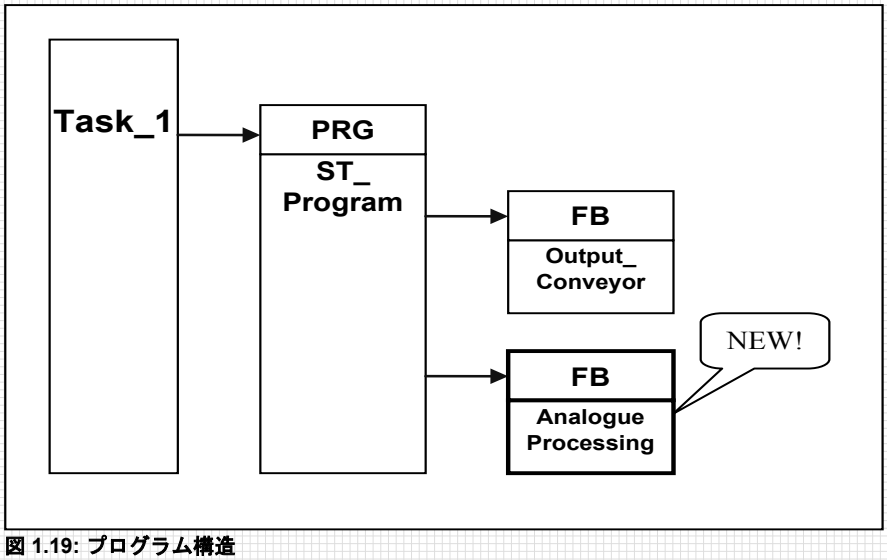


図 1.19: プログラム構造

III. I/O マッピングの変数

変数名／ データ型	デバイス名	スロット 番号	ハードウェア アドレス	センサ／ アクチュエータ
I_w_Analogue_InputValue_1 WORD	Geraet_1	5	I0 (11, 14)	Pot_UPPER
I_w_Analogue_InputValue_2 WORD	Geraet_1	5	I1 (21, 24)	Pot_LOWER
Q_w_Analogue_OutputValue_1 WORD	Geraet_1	6	O0 (11)	インジケータ
Q_w_Analogue_OutputValue_2 WORD	Geraet_1	6	O1 (21)	-

IV. 必要な一般ファンクション

インスタンス名	一般ファンクション	キー
不要	WORD_TO_INT	データ変換
不要	INT_TO_WORD	データ変換
…	…	…

第 1 章の
終わり