SIEMENS

SIMOTION

SIMOTION SCOUT
SIMOTION ST 構造化テキスト

プログラミング/操作マニュアル

はじめに

はじめに 1

STの入門 2

STの基本原理 3

ファンクション、
ファンクションブロック、
およびプログラム 4

SIMOTIONへのSTの統合 5

エラーソースおよび
プログラムのデバッグ 6

付録 A

03/2007

安全性に関する基準

本書には、ユーザーの安全性を確保し製品の損傷を防止するうえ守るべき注意事項が記載されています。
ユーザーの安全性に関する注意事項は、安全警告サインで強調表示されています。このサインは、物的損傷に関する注意事項には表示されません。

⚠危険
回避しなければ、直接的な死または重傷に至る危険状態を示します。

⚠警告
回避しなければ、死または重傷に至るおそれのある危険な状況を示します。

⚠注意
回避しなければ、軽度または中度の人身傷害を引き起こすおそれのある危険な状況を示します（安全警告サイン付き）。

注意
回避しなければ、物的損傷を引き起こすおそれのある危険な状況を示します（安全警告サインなし）。

通知
回避しなければ、望ましくない結果や状態が生じ得る状況を示します（安全警告サインなし）。

複数の危険レベルに相当する場合は、通常、最も危険度の高い（番号の低い）事項が表示されることになっています。安全警告サイン付きの人身傷害に関する注意事項があれば、物的損傷に関する警告が付加されます。

有資格者

装置/システムのセットアップおよび使用にあたっては必ず本マニュアルを参照してください。機器のインストールおよび操作は有資格者のみが行うものとします。有資格者とは、法的な安全規制/規格に準拠してアースの取り付け、電気回路、設備およびシステムの設定に携わることを承認されている技術者のことをいいます。

使用目的

以下の事項に注意してください。

⚠警告
本装置およびコンポーネントはカタログまたは技術的な解説に詳述されている用途にのみ使用するものとします。また、Siemens 社の承認または推奨するメーカーの装置またはコンポーネントのみを使用してください。 本製品は輸送、据付け、セットアップ、インストールを正しく行い、推奨のとおりに操作および維持した場合にのみ、正確かつ安全に作動します。

商標

®マークのついた称号はすべて Siemens AG の商標です。本書に記載するその他の称号は商標であり、第三者が自己の目的において使用した場合、所有者の権利を侵害することになります。

免責事項

本書のハードウェアおよびソフトウェアに関する記述と、実際の製品内容との一致については検証済みです。
しかしなお、本書の記述が実際の製品内容と異なる可能性もあり、完全な一致が保証されているわけではありません。記載内容については定期的に検証し、訂正が必要な場合は次の版て更新いたします。

Siemens AG
Automation and Drives
Postfach 48 48
90327 NÜRNBERG
ドイツ

℗ 09/2007

はじめに

対象製品

本書は SIMOTION プログラミングのマニュアルパッケージの一部です。

本書は以下の SIMOTION SCOUT V4.1 に使用できます。

- SIMOTION SCOUT V4.1 (SIMOTION 製品シリーズのエンジニアリングシステム)、
  および、併用する以下の製品
- SIMOTION Kernel V4.1、V4.0、V3.2、V3.1、V3.0、または V2.1
- SIMOTION テクノロジーパッケージ Cam、Path (カーネル V4.1 現在)、Cam_ext (カーネル V3.2 現在)、および各カーネルのバージョンの TControl(カーネル V3.0 までのテクノロジーパッケージ Gear、Position、および基本 MC を含む)

本書の目的と構成

下記のリストは、このマニュアルの内容を章ごとにまとめたものです。

- **はじめに** (第 1 章)
- **ST の入門** (第 2 章)
  プログラムおよびサンプルプログラムを作成するための必要条件
- **ST の基本原理** (第 3 章)
  ST プログラミング言語の要素、変数とデータタイプの宣言、ステートメント
- **ファンクション、ファンクションブロック、およびプログラム** (第 4 章)
  プログラミングおよびプログラムオーガニゼーションユニット(POU)の呼び出し
- **SIMOTION SCOUT への ST の統合** (第 5 章)
  変数の動作、入力および出力へのアクセス、タスクシステム、システムファンクションのアプリケーション事例集
- **エラーソースおよびプログラムテスト** (第 6 章)
  エラーソース、効率的なプログラミング、およびプログラムテストに関する情報
- **付録**
  - **形式言語記述** (付録 A.1)
  - **コンパイラエラーメッセージと修正方法** (付録 A.2)
  - **ユニット例のテンプレート** (付録 A.3)
- **索引**

すぐに開始したい場合は、第 2 章から始めてください。

SIMOTION ドキュメンテーション

SIMOTION ドキュメンテーションの一覧は、別途、参考文献一覧として掲載されています。

このマニュアルは、提供される SIMOTION SCOUT とともに電子マニュアルとして収録されます。

SIMOTION 取扱説明書は 9 個のマニュアルパッケージで構成され、そのパッケージには約 60 の SIMOTION マニュアルとその他の製品(たとえば SINAMICS)に関するマニュアルが含まれています。

SIMOTION V4.1 では、以下のドキュメンテーションパッケージを使用できます。

- SIMOTION エンジニアリングシステム
- SIMOTION システムおよび機能
- SIMOTION 診断
- SIMOTION プログラミング
- SIMOTION プログラミング - リファレンス
- SIMOTION C2xx
- SIMOTION P350
- SIMOTION D4xx
- SIMOTION 追加ドキュメンテーション

ホットラインおよびインターネットアドレス

技術上のご質問がある場合は、弊社のホットライン(世界中どこでも可能です)にお問い合わせください。

A&D テクニカルサポート:

- 電話番号:+49 (180) 50 50 222
- FAX 番号: +49 (180) 50 50 223
- 電子メール:adsupport@siemens.com
- インターネット: http://www.siemens.de/automation/support-request

ご質問やご提案がある場合や、ドキュメンテーションの間違いにお気付きの場合は、次の連絡先宛にファックスまたは電子メールでお知らせください。

- FAX 番号: +49 (9131) 98 63315
- 電子メール: docu.motioncontrol@siemens.com

Siemens インターネットアドレス

SIMOTION 製品、製品サポート、および FAQ に関する情報は、インターネットの次のアドレスに掲載されています。

- 一般情報:
  - **http://www.siemens.de/simotion**(ドイツ)
  - **http://www.siemens.com/simotion**(世界共通)
- 製品サポート:
  - **http://support.automation.siemens.com/WW/view/en/10805436**

その他のサポート

弊社は、SIMOTION の習得のための入門コースも提供しています。

お客様の地域のトレーニングセンターか、D-90027 Nuremberg/Germany、Tel +49 (911) 895 3202 の本部トレーニングセンターにお問い合わせください。

目次

はじめに **3**

1 はじめに **17**

1.1 高級プログラミング言語 17

1.2 テクノロジーコマンドを備えたプログラミング言語 17

1.3 実行レベル 17

1.4 プログラムの作成およびテスト用ツールを備えたST工ディタ 18

2 STの入門 **19**

2.1 SCOUTへのSTの統合 19
2.1.1 ワークベンチの要素の概要 20

2.2 プログラム作成の必要条件 21

2.3 STエディタとコンパイラの操作 22
2.3.1 STソースファイルの挿入 22
2.3.2 既存のSTソーフファイルを開く 23
2.3.3 STソースファイルの特性の変更 24
2.3.4 ST工ディタの操作 25
2.3.4.1 構文の色付け 25
2.3.4.2 ドラッグアンドドロップ 26
2.3.4.3 キーボードショートカット 27
2.3.4.4 ST工ディタの設定 27
2.3.4.5 コマンドライブラリの使用 28
2.3.5 STコンパイラの起動 29
2.3.5.1 エラーの修正のためのヘルプ 29
2.3.5.2 ST工ディタのツールバー 29
2.3.6 STコンパイラの設定 30
2.3.6.1 STコンパイラの設定 30
2.3.6.2 コンパイラのグローバル設定 31
2.3.6.3 STコンパイラのローカル設定 32
2.3.6.4 警告クラスの意味 34
2.3.7 STソースファイルのノウハウ保護 35
2.3.8 プリプロセッサ定義の作成 35
2.3.9 STソースファイルのエクスポート、インポート、および印刷 37
2.3.9.1 STソースファイルのXMLフォーマットでのエクスポート 37
2.3.9.2 テキストファイル(ASCII)のSTソースファイルとしてのインポート 37
2.3.9.3 STソースファイルへのXMLデータのインポート 38
2.3.9.4 STソールファイルの印刷 38
2.3.10 外部工ディタの使用 39
2.3.11 STソースファイルのメニュー 41
2.3.11.1 STソースファイルのメニュー 41
2.3.11.2 STソースファイルのコンテキストメニュー 41

2.4 サンプルプログラムの作成 43
2.4.1 必要条件 43
2.4.2 プロジェクトを開く、または作成する 43

2.4.3 ハードウェアを認識させる ... 44
2.4.4 STエディタによるソーステキストの入力 ... 45
2.4.4.1 エディタの機能 ... 47
2.4.4.2 サンプルプログラムのソーステキスト ... 47
2.4.5 サンプルプログラムのコンパイル ... 48
2.4.5.1 コンパイラの起動 ... 48
2.4.5.2 エラーの修正 ... 48
2.4.5.3 例: STソースファイルのコンパイル中のエラーメッセージ ... 49
2.4.6 サンプルプログラムの実行 ... 50
2.4.6.1 実行レベルへのサンプルプログラムの割り付け ... 50
2.4.6.2 ターゲットシステムへの接続の確立 ... 51
2.4.6.3 ターゲットシステムへのサンプルプログラムのダウンロード ... 52
2.4.6.4 サンプルプログラムの起動とテスト ... 53

3 STの基本原理 ... 55

3.1 言語記述リソース ... 55
3.1.1 構文ダイアグラム ... 55
3.1.2 構文ダイアグラムのブロック ... 56
3.1.3 ルールの意味(セマンティクス) ... 56

3.2 言語の基本要素 ... 57
3.2.1 ST文字セット ... 57
3.2.2 STの識別子 ... 57
3.2.2.1 識別子のルール ... 57
3.2.2.2 識別子の例 ... 58
3.2.3 予約識別子 ... 59
3.2.3.1 保護識別子 ... 60
3.2.3.2 その他の予約識別子 ... 64
3.2.4 数字およびブール値 ... 66
3.2.4.1 整数 ... 66
3.2.4.2 浮動小数点数 ... 67
3.2.4.3 指数 ... 67
3.2.4.4 ブール値 ... 67
3.2.4.5 数字のデータタイプ ... 68
3.2.5 文字列 ... 68

3.3 STソースファイルの構造 ... 69
3.3.1 ステートメント ... 71
3.3.2 説明 ... 71

3.4 データタイプ ... 72
3.4.1 基本データタイプ ... 73
3.4.1.1 基本データタイプ ... 73
3.4.1.2 基本データタイプの値の範囲限界 ... 75
3.4.1.3 一般的なデータタイプ ... 76
3.4.1.4 基本システムデータタイプ ... 76
3.4.2 ユーザ定義データタイプ ... 77
3.4.2.1 ユーザ定義データタイプ ... 77
3.4.2.2 ユーザ定義データタイプの構文(タイプ宣言) ... 77
3.4.2.3 基本データタイプまたは派生データタイプの派生 ... 79
3.4.2.4 ARRAY派生データタイプ ... 80
3.4.2.5 列挙子派生データタイプ ... 82
3.4.2.6 STRUCT (構造体)派生データタイプ ... 83
3.4.3 テクノロジーオブジェクトデータタイプ ... 85
3.4.3.1 テクノロジーオブジェクトのデータタイプの説明 ... 85
3.4.3.2 軸のプロパティの継承 ... 86
3.4.3.3 テクノロジーオブジェクトデータタイプの使用例 ... 86

3.4.4 システムデータタイプ 87

3.5 変数宣言 88
3.5.1 変数宣言の構文 88
3.5.2 すべての変数宣言の概要 89
3.5.3 変数またはデータタイプの初期化 90
3.5.4 定数 94

3.6 値割り付けおよび式 95
3.6.1 値割り付け 95
3.6.1.1 値割り付けの構文 95
3.6.1.2 基本データタイプの変数を使用した値割り付け 97
3.6.1.3 STRING基本データタイプの変数を使用した値割り付け 97
3.6.1.4 ビットデータタイプの変数を使用した値割り付け 99
3.6.1.5 列挙子派生データタイプの変数を使用した値割り付け 101
3.6.1.6 ARRAY派生データタイプの変数を使用した値割り付け 101
3.6.1.7 STRUCT派生データタイプの変数を使用した値割り付け 102
3.6.2 式 103
3.6.2.1 式の結果 103
3.6.2.2 式の解釈順序 103
3.6.3 オペランド 105
3.6.4 演算式 106
3.6.4.1 演算式の例 108
3.6.5 関係式 109
3.6.6 論理式およびビットシリアル式 111
3.6.7 演算子の優先度 113

3.7 制御ステートメント 114
3.7.1 IFステートメント 114
3.7.2 CASEステートメント 115
3.7.3 FORステートメント 118
3.7.3.1 FORステートメントの処理 118
3.7.3.2 FORステートメントのルール 119
3.7.3.3 FORステートメントの例 119
3.7.4 WHILEステートメント 120
3.7.5 REPEATステートメント 121
3.7.6 EXITステートメント 122
3.7.7 RETURNステートメント 122
3.7.8 WAITFORCONDITIONステートメント 123
3.7.9 GOTOステートメント 124

3.8 データタイプの変換 125
3.8.1 基本データタイプの変換 125
3.8.1.1 暗黙的なデータタイプ変換 126
3.8.1.2 明示的なデータタイプ変換 128
3.8.2 その他の変換 129

4 ファンクション、ファンクションブロック、およびプログラム 131

4.1 ファンクション、ファンクションブロック、およびプログラム 131

4.2 ファンクションおよびファンクションブロックの作成と呼び出し 131
4.2.1 ファンクションの定義 132
4.2.2 ファンクションブロックの定義 133
4.2.3 FBおよびFCの宣言セクション 133
4.2.4 FBおよびFCのステートメントセクション 136
4.2.5 ファンクションおよびファンクションブロックの呼び出し 137
4.2.5.1 パラメータ転送の原理 137
4.2.5.2 入力パラメータへのパラメータの転送 138

4.2.5.3 入/出力パラメータへのパラメータの転送 ... 138
4.2.5.4 出力パラメータへのパラメータの転送(FBの場合のみ) ... 139
4.2.5.5 パラメータのアクセスタイム ... 140
4.2.5.6 ファンクションの呼び出し ... 140
4.2.5.7 ファンクションブロックの呼び出し(インスタンスの呼び出し) ... 141
4.2.5.8 FBの外部でのFBの出力パラメータのアクセス ... 143
4.2.5.9 FBの外部でのFBの入力パラメータのアクセス ... 143
4.2.5.10 FBの呼び出しにおけるエラーソース ... 144

4.3 ファンクションとファンクションブロックの比較 ... 145
4.3.1 例の説明 ... 145
4.3.2 コメント付きソースファイル ... 146

4.4 プログラム ... 148

4.5 式 ... 149

5 SIMOTIONへのSTの統合 ... 151

5.1 ソースファイルセクション ... 151
5.1.1 ソースファイルセクションの使用 ... 151
5.1.1.1 インターフェースセクション ... 152
5.1.1.2 実装セクション ... 153
5.1.1.3 プログラムオーガニゼーションユニット(POU) ... 154
5.1.1.4 ファンクション(FC) ... 154
5.1.1.5 式 ... 155
5.1.1.6 ファンクションブロック(FB) ... 156
5.1.1.7 プログラム ... 157
5.1.1.8 宣言セクション ... 157
5.1.1.9 ステートメントセクション ... 158
5.1.1.10 データタイプ定義 ... 158
5.1.1.11 変数宣言 ... 159
5.1.2 STソースファイル間のインポートとエクスポート ... 161
5.1.2.1 ユニット識別子 ... 161
5.1.2.2 インターフェースセクションでの指定 ... 161
5.1.2.3 インポートユニットのインターフェースまたは実装セクションでのUSESステートメントの使用 ... 162
5.1.2.4 インポートユニットの例 ... 164
5.1.2.5 エクスポートユニットのインターフェースセクションでの指定 ... 165
5.1.2.6 エクスポートユニットの例 ... 166

5.2 SIMOTIONの変数 ... 167
5.2.1 変数モデル ... 167
5.2.1.1 同名の変数の使用 ... 169
5.2.1.2 ユニット変数 ... 170
5.2.1.3 非保持型ユニット変数 ... 171
5.2.1.4 保持型ユニット変数 ... 173
5.2.1.5 ローカル変数(スタティック変数およびテンポラリ変数) ... 174
5.2.1.6 スタティック変数 ... 175
5.2.1.7 テンポラリ変数 ... 176
5.2.2 名前空間 ... 177
5.2.3 グローバルデバイス変数の使用 ... 178
5.2.4 変数タイプのメモリ範囲 ... 178
5.2.4.1 例 ... 180
5.2.4.2 ローカルデータスタックの変数のメモリ必要条件(Kernel V3.1 以降) ... 183
5.2.4.3 ローカルデータスタックの変数のメモリ必要条件(Kernel V3.0 以前) ... 183
5.2.5 変数の初期化の時期 ... 184
5.2.5.1 保持性グローバル変数の初期化 ... 185
5.2.5.2 非保持性グローバル変数の初期化 ... 186

5.2.5.3 ローカル変数の初期化 187
5.2.5.4 ファンクションブロック(FB)のインスタンス初期化 187
5.2.5.5 テクノロジーオブジェクトのシステム変数初期化 188
5.2.5.6 グローバル変数のバージョンIDとダウンロード中の初期化 189
5.2.5.7 スタティックプログラム変数の初期化 190
5.2.6 変数およびHMIデバイス 191

5.3 入力および出力(プロセスイメージ、I/O変数)へのアクセス 192
5.3.1 入力と出力へのアクセス概要 192
5.3.2 直接アクセスとプロセスイメージアクセスの重要な機能 193
5.3.3 周期的タスクの直接アクセスおよびプロセスイメージ 195
5.3.3.1 直接アクセスのI/Oアドレスとサイクリックタスクのプロセスイメージのルール 196
5.3.3.2 直接アクセスまたはサイクリックタスクのプロセスイメージ用のI/O変数の作成 196
5.3.3.3 I/Oアドレス入力の構文 198
5.3.3.4 I/O変数の指定可能なデータタイプ 198
5.3.4 BackgroundTaskの固定プロセスイメージへのアクセス 199
5.3.4.1 BackgroundTaskの固定プロセスイメージへの絶対アクセス(絶対PIアクセス) 200
5.3.4.2 絶対プロセスイメージアクセスの識別子の構文 201
5.3.4.3 BackgroundTaskの固定プロセスイメージへのシンボリックアクセス(シンボリックPIアクセス) 202
5.3.4.4 シンボリックプロセスイメージ(PI)アクセスに指定可能なデータタイプ 202
5.3.4.5 シンボリックPIアクセスの例 203
5.3.4.6 BackgroundTaskの固定プロセスイメージへのアクセス用のI/O変数の作成 203
5.3.5 I/O変数へのアクセス 204

5.4 ライブラリの使用 205
5.4.1 ライブラリのコンパイル 205
5.4.2 ライブラリのノウハウ保護 206
5.4.3 ライブラリのデータタイプ、ファンクション、およびファンクションブロックの使用 207
5.4.4 名前空間 208

5.5 基準データ 210
5.5.1 クロスリファレンスリスト 210
5.5.1.1 クロスリファレンスリストの作成 210
5.5.1.2 クロスリファレンスリストの内容 211
5.5.1.3 クロスリファレンスリストでの作業 211
5.5.2 Program structure 212
5.5.2.1 プログラム構造の内容 212
5.5.3 コード属性 213
5.5.3.1 コード属性の内容 214

5.6 プラグマによるプリプロセッサおよびコンパイラの制御 214
5.6.1 プリプロセッサの制御 215
5.6.1.1 プリプロセッサステートメント 215
5.6.1.2 プリプロセッサステートメントの例 218
5.6.2 属性によるコンパイラ出力の制御 219

5.7 ジャンプステートメントおよびラベル 221

6 エラーソースおよびプログラムのデバッグ 223

6.1 エラーの回避と効率的なプログラミングに関する注意事項 223

6.2 プログラムのデバッグ 223
6.2.1 プログラムのテストモード 224
6.2.1.1 SIMOTIONデバイスのモード 224
6.2.1.2 デバッグモードに関する重要な情報 225
6.2.1.3 [emergency stop parameterization]パラメータ 226
6.2.2 シンボルブラウザ 227
6.2.2.1 シンボルブラウザの特徴 227

6.2.2.2 シンボルブラウザの使用 ... 227
6.2.3 ウォッチテーブルでの変数の監視 ... 230
6.2.3.1 ウォッチテーブルの変数 ... 230
6.2.3.2 ウォッチテーブルの使用 ... 230
6.2.4 プログラム実行 ... 232
6.2.4.1 プログラム実行: コード位置と呼び出しパスの表示 ... 232
6.2.4.2 [call stack program run]パラメータ ... 232
6.2.4.3 プログラム実行ツールバー ... 233
6.2.5 プログラムステータス ... 233
6.2.5.1 プログラムステータスの特徴 ... 233
6.2.5.2 ステータスプログラムの使用 ... 234
6.2.5.3 プログラムステータスの呼び出しパス ... 236
6.2.5.4 [call path status program]パラメータ ... 237
6.2.5.5 プログラムステータスの更新 ... 237
6.2.6 ブレークポイント ... 237
6.2.6.1 ブレークポイント設定の一般的な手順 ... 237
6.2.6.2 デバッグタスクグループの定義 ... 238
6.2.6.3 [Debug settings]パラメータ ... 240
6.2.6.4 [Debug table]パラメータ ... 241
6.2.6.5 ブレークポイントの設定 ... 241
6.2.6.6 [Breakpoints]ツールバー ... 243
6.2.6.7 単一ブレークポイントの呼び出しパスの定義 ... 243
6.2.6.8 [Breakpoint call path]パラメータ ... 245
6.2.6.9 すべてのブレークポイントに呼び出しパスを定義 ... 245
6.2.6.10 [Call Path of all Breakpoints per POU]パラメータ ... 247
6.2.6.11 ブレークポイントの有効化 ... 248
6.2.6.12 呼び出しスタックの表示 ... 249
6.2.6.13 [Breakpoints call stack]パラメータ ... 249
6.2.7 トレース ... 250

A 付録 ... 251

A.1 形式言語記述 ... 251
A.1.1 言語記述リソース ... 251
A.1.1.1 書式付きルール(語彙ルール) ... 251
A.1.1.2 書式なしルール(構文ルール) ... 252
A.1.2 基本要素(端子) ... 253
A.1.2.1 アルファベット、数字、およびその他の文字 ... 253
A.1.2.2 ルールにおけるフォーマット文字およびセパレータ ... 253
A.1.2.3 定数のフォーマット文字およびセパレータ ... 255
A.1.2.4 プロセスイメージアクセスのために事前定義された識別子 ... 256
A.1.2.5 Taskstartinfoの識別子 ... 256
A.1.2.6 演算子 ... 257
A.1.2.7 予約語 ... 257
A.1.3 ルール ... 265
A.1.3.1 識別子 ... 265
A.1.3.2 定数(リテラル)の表記 ... 266
A.1.3.3 説明 ... 273
A.1.3.4 STソースファイルセクション ... 274
A.1.3.5 STソースファイルの構造 ... 274
A.1.3.6 プログラムオーガニゼーションユニット(POU) ... 276
A.1.3.7 宣言セクション ... 277
A.1.3.8 宣言ブロックの構造 ... 279
A.1.3.9 STのデータタイプ ... 286
A.1.3.10 ステートメントセクション ... 290
A.1.3.11 値割り付けおよび演算子 ... 291
A.1.3.12 ファンクションおよびファンクションブロックの呼び出し ... 297
A.1.3.13 制御ステートメント ... 300

A.2 コンパイラエラーメッセージと修正方法 305
A.2.1 ファイルアクセスエラー(1000 – 1003、1100) 305
A.2.2 スキャナエラー(2001 – 2002) 305
A.2.3 POUにおける宣言エラー (3002 – 3027) 306
A.2.4 タイプ宣言における宣言エラー (4001 – 4051) 307
A.2.5 変数宣言における宣言エラー (5001 – 5014、5100 – 5111、5500 – 5507) 308
A.2.6 式のエラー (6001 – 6140) 309
A.2.7 構文エラー、式のエラー(7000 – 7014) 318
A.2.8 ソースファイルをリンクする際のエラー(8001) 318
A.2.9 別のUNITまたはテクノロジーパッケージのインターフェースをロードする間のエラー (10000 – 10036、10100 – 10101) 319
A.2.10 実装制限(15001 – 15007、15152、15152) 321
A.2.11 警告(16001 – 16601) 322
A.2.12 情報 325
A.3 ユニット例のテンプレート 326
A.3.1 予備情報 326
A.3.2 インターフェースでのタイプ定義 327
A.3.3 インターフェースでの変数宣言 328
A.3.4 実装 329
A.3.5 ファンクション 330
A.3.6 ファンクションブロック(Function block) 331
A.3.7 プログラム 332
A.3.8 初期化に関する注意事項 333
索引 335

表

表 2-1 STエディタのキーボードショートカット 27
表 2-3 STエディタのファンクション 30
表 2-4 コンパイラのグローバル設定のパラメータ 31
表 2-5 STコンパイラのローカル設定のパラメータ 33
表 2-8 STソースファイルのメニュー 41
表 2-9 STソースファイルのコンテキストメニュー 41
表 3-1 STプログラミング言語の保護識別子 60
表 3-2 ST言語のその他の予約識別子 64
表 3-3 文字列内の特殊文字を表す 2 文字の組み合わせ 69
表 3-7 基本データタイプのビット幅と値の範囲 73
表 3-12 基本データタイプの派生の例 79
表 3-13 一次元配列の例 81
表 3-17 テクノロジーオブジェクトのデータタイプ(TOデータタイプ) 85
表 3-18 テクノロジーオブジェクトデータタイプの無効な値に関するシンボリック定数 85
表 3-22 宣言ブロックのキーワード 90
表 3-25 定数の例 95
表 3-29 算術演算子 107
表 3-32 関係式の例 110
表 3-44 WAITFORCONDITIONステートメントの例 124
表 3-45 数値データタイプおよびビットデータタイプのタイプ変換 125

表 3-46 式および値割り付けのデータタイプの例 ... 127
表 4-4 インスタンスの宣言、FBの呼び出し、および出力パラメータへのアクセスの例 ... 142
表 4-6 FBとFCの違いの例 ... 146
表 4-7 前の例のFBとFCの違いの例 ... 147
表 5-1 インターフェースセクションの構文 ... 152
表 5-2 実装セクションの構文 ... 153
表 5-3 ファンクション(FC)の構文 ... 154
表 5-4 式の構文 ... 155
表 5-5 ファンクションブロックの構文 ... 156
表 5-6 プログラムの構文 ... 157
表 5-7 宣言セクションの構造 ... 157
表 5-8 ステートメントセクションの構造 ... 158
表 5-9 データタイプ定義の構文 ... 158
表 5-10 変数宣言の構文 ... 159
表 5-11 インターフェースセクションまたは実装セクションへのUSESステートメントの配置に関する影響 ... 163
表 5-12 インポートユニットの例 ... 164
表 5-13 エクスポートユニットの例 ... 166
表 5-14 識別子の有効性の例 ... 170
表 5-15 非保持型ユニット変数の例 ... 172
表 5-17 ソースファイルセクションに応じたスタティック変数とテンポラリ変数の宣言キーワード ... 174
表 5-19 さまざまな変数タイプに割り付けられたメモリ範囲とその初期化 ... 179
表 5-20 • 変数タイプのメモリ領域の例、Kernel V3.1 以降(パート 1) ... 180
表 5-21 • 変数タイプのメモリ領域の例、Kernel V3.1 以降(パート 2) ... 181
表 5-22 • 変数タイプのメモリ領域の例、Kernel V3.1 以降(パート 3) ... 182
表 5-23 ダウンロード中の保持性グローバル変数の初期化 ... 185
表 5-24 ダウンロード中の非保持性グローバル変数の初期化 ... 186
表 5-25 プログラム編成ユニットの呼び出し時のローカル変数初期化 ... 187
表 5-26 ダウンロード中のテクノロジーオブジェクトシステム変数の初期化 ... 188
表 5-27 グローバル変数のバージョンIDとダウンロード中の初期化 ... 189
表 5-28 直接アクセスとプロセスイメージアクセスの重要な機能 ... 193
表 5-34 プログラム構造の表示エレメント ... 213
表 5-38 コンパイラオプションの属性の例 ... 220
表 6-5 デバッグタスクグループ内のタスクに応じて有効にされたブレークポイントに達する動作 ... 239
表 6-10 [Breakpoint call path]パラメータの説明 ... 250
表 A-2 語彙ルールにおけるフォーマット文字およびセパレータ ... 253
表 A-3 構文ルールにおけるフォーマット文字およびセパレータ ... 254

表 A-8 基本STシステムのSTキーワードおよび定義済み識別子 257
表 A-9 ファイルアクセスエラー 305
表 A-10 スキャナエラー(2001 – 2002)....305
表 A-11 POUにおける宣言エラー (3002 – 3027) 306
表 A-12 タイプ宣言における宣言エラー (4001 – 4051) 307
表 A-13 変数宣言における宣言エラー (5001 – 5014、5100 – 5111、5500 – 5507)....308
表 A-14 式のエラー (6001 – 6140)....309
表 A-15 構文エラー、式のエラー(7000 – 7014) 318
表 A-16 ソースファイルをリンクする際のエラー(8001)....318
表 A-17 別のUNITまたはテクノロジーパッケージのインターフェースをロードする間のエラー (10000 – 10036、10100 – 10101) 319
表 A-18 実装制限(15001 – 15007、15152、15153) 321
表 A-19 警告(16001 – 16601)....322
表 A-20 情報 325

はじめに 1

今日の自動化システムでは、従来の開ループ制御タスクと閉ループ制御タスクに加えて、データ管理関数や複雑な数学計算を処理する必要性がますます高まっています。ST (構造化テキスト)は、こうしたタスク向けに特別に設計されています。IEC 61131-3 (ドイツ標準 DIN EN-61131-3)に合わせて標準化されているため、このプログラミング言語を使用することで、プログラマとしての仕事がより容易になります。

1.1 高級プログラミング言語

ST は、PASCAL ベースの高級プログラミング言語です。この言語は IEC 61131-3 標準に基づいており、プログラマブルコントローラ(PLC)用のプログラミング言語を統一したものです。ST は、この標準の *構造化テキスト* 部分に基づいています。

プログラム制御システムに ST などの高級言語を使用すると、ユーザにたとえば次のような幅広い可能性が提供されます。

- データ管理
- プロセス最適化
- 数学/統計計算

1.2 テクノロジーコマンドを備えたプログラミング言語

SIMOTION ST プログラミング言語は、IEC 61131-3 に準拠しているだけでなく、SIMOTION デバイス、モーションコントロール、およびテクノロジーのコマンドも含んでいます。

テクノロジーオブジェクトは、軸の位置決めや出力カムのパラメータの割り付けなどの技術的機能を表します。テクノロジーコマンドは、テクノロジーオブジェクトによって提供される言語コマンドです。こうしたコマンドは、たとえば軸を位置決めするために、カム機能を有効化したり、モーションシーケンスを制御したりするのに使用することができます。

1.3 実行レベル

SIMOTION 実行システムには、ユーザプログラムの作成に関係する各種のタスクを最適にサポートするため、さまざまな実行レベル(周期的、同期、時間制御、アラーム制御、および連続)が提供されています。

SIMOTION SCOUT は、SIMOTION 製品ファミリのエンジニアリングシステムです。ST は、ユーザプログラムを作成するための高級言語です。ST では、さまざまな実行レベルのユーザプログラムを開発できます。

ユーザプログラムをシステムクロックまたは定義済みタイムサイクルと同期的に実行したい場合、ユーザプログラムの実行を時間で駆動することができます。ユーザプログラムを特定のイベントに対して 1 回起動し実行する場合、ユーザプログラムを割り込みで駆動することができます。あるいは、ラウンドロビン実行レベルで連続的または周期的に実行することもできます。

1.4 プログラムの作成およびテスト用ツールを備えた ST エディタ

プログラムの作成用に、使いやすいテキストエディタが用意されています。

ST コンパイラは、編集済みプログラムを実行可能コードに変換し、構文エラーがあればプログラム行とエラーソースを指定してそれを知らせます。

SIMOTION SCOUT には、ST プログラムのテスト用のテストファンクションが提供されています。プログラムはオンラインでテストおよび視覚化することができます。

ST の入門

2

この章では、プログラムを作成し、実行可能コードにコンパイルして実行し、テストを行う方法を簡単な例を使用して説明します。

2.1 SCOUT への ST の統合

ST のプログラミング環境は以下のコンポーネントで構成されます。

- プログラムを作成するための**エディタ**。ファンクション(FC)、ファンクションブロック(FB)、ユーザ定義データタイプ(UDT)などで構成されます。
- 以前に編集した ST プログラムを実行可能マシンコードに変換するための**コンパイラ**
- 実行中のプログラムにおける論理プログラムエラーの検索を支援する**プログラムステータス**
- コンパイラのエラーメッセージなどが表示される**詳細ビュー**。詳細ビューの重要なタブは**[Symbol browser]**です。このタブでは、変数の監視と変更を行うことができます。

個々のコンポーネントは使いやすいものになっています。コンポーネントは、SIMOTION SCOUT ワークベンチに直接統合されています。

ワークベンチとそのツールの操作の詳細については、『SIMOTION SCOUT 設定マニュアル』を参照してください。

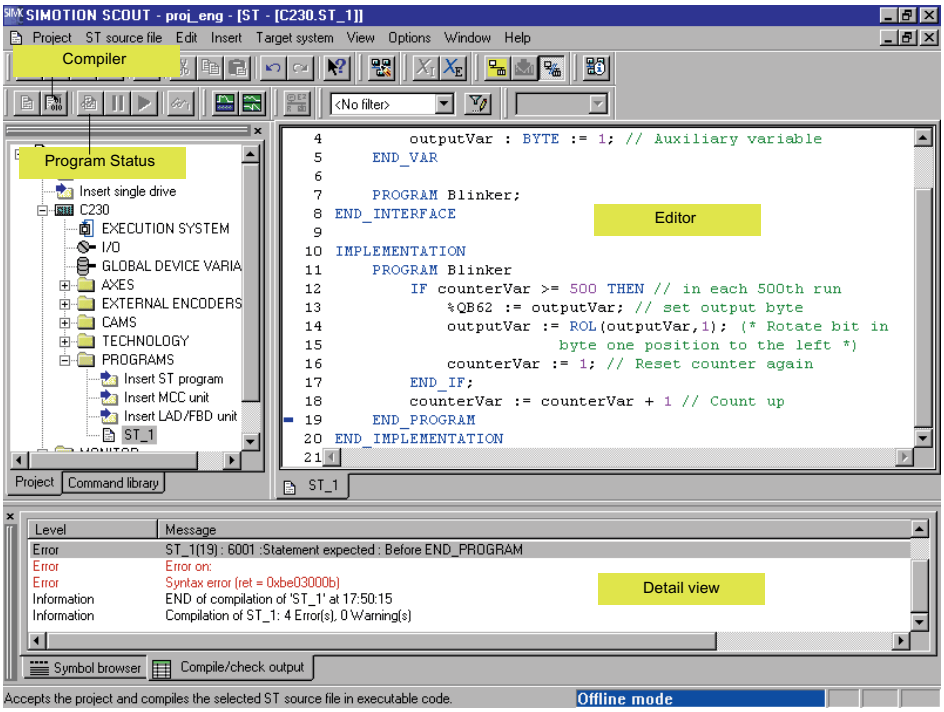


図 2-1 ST の開発環境

2.1.1 ワークベンチの要素の概要

ワークベンチは SIMOTION SCOUT の枠組みを表します。ワークベンチのツールを使用すると、マシンを用途に合わせて設定、最適化、およびプログラミングするのに必要なすべての手順を実行できます。

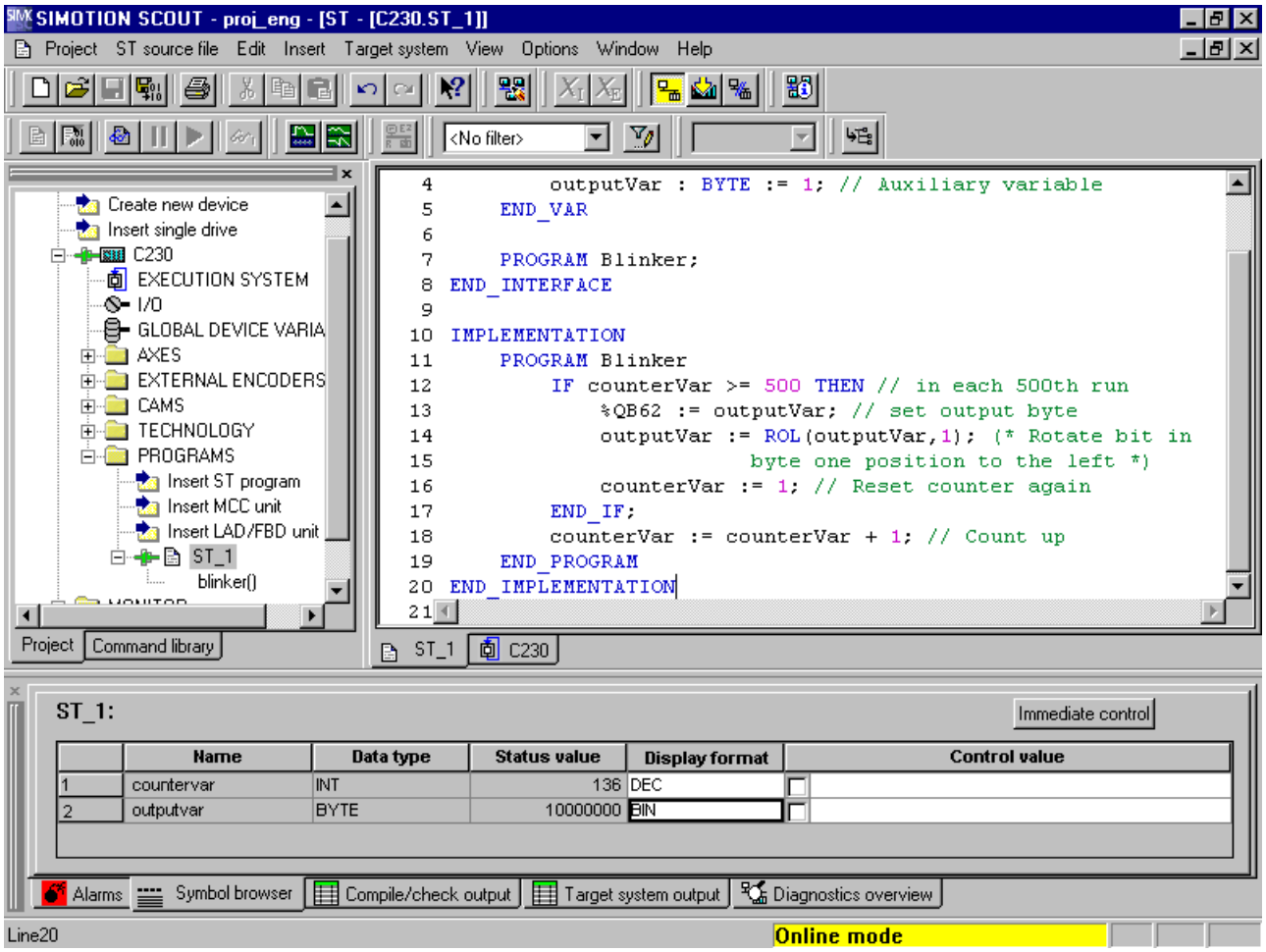


図 2-2 ワークベンチの要素

ワークベンチには、次の要素があります。

- メニュー

メニューには、ワークベンチの制御やツールの読み出しなどができるメニューコマンドがあります。

- ツールバー

いずれかのツールバーの対応するボタンをクリックすると、使用可能なメニューコマンドの多くを実行できます。

- プロジェクトナビゲータ

プロジェクトナビゲータには、プロジェクト全体とその要素(CPU、軸、プログラム、カムなど)がツリー構造で表示されます。

- 作業エリア

このウィンドウでは、独力で(プログラミングにより)、またはウィザードを使用して(設定を行うことにより)特定のタスクを実行することができます。

- 詳細ビュー

詳細ビューには、プログジェクトナビゲータで選択した要素に関する関連情報が表示されます(たとえば、プログラムまたは[Compile/Test Output]ウィンドウのすべてのグローバグ変数など)。

2.2 プログラム作成の必要条件

このセクションでは、プログラムを作成する前に満たす必要がある一般的な条件を説明します。詳細については、『SIMOTION SCOUT 設定マニュアル』および SIMOTION モーションコントロール機能の説明を参照してください。

プロジェクトを追加する、または開く

プロジェクトは、データ管理階層構造の最上位レベルにあります。たとえば、SIMOTION SCOUT では、生産機械に属しているすべてのデータはプロジェクトディレクトリに保存されます。

つまり、プロジェクトは、1 つの機械に属するすべての SIMOTION デバイス、ドライブなどをひとまとめにしたものです。

プロジェクトを作成したら、以下のことを実行できます。

- ハードウェアの設定
- テクノロジーオブジェクトの挿入と設定

ハードウェアの設定

プロジェクト内で、以下を含め、使用されているハードウェアをシステムに認識させる必要があります。

- SIMOTION デバイス
- 中央 I/O (I/O アドレスを使用)
- 分散 I/O (I/O アドレスを使用)

SIMOTION デバイスを設定しないと、ST ソースファイルを挿入および編集することはできません。

テクノロジーオブジェクトの挿入と設定

軸や出力カムなどの機能は、SIMOTION ではテクノロジーオブジェクト(TO)によって表されます。

テクノロジーオブジェクトを挿入し、設定すると、システムファンクションを使用してテクノロジーオブジェクトをプログラミングし、そのシステム変数にアクセスできるようになります。

2.3 ST エディタとコンパイラの操作

このセクションでは、ST エディタとコンパイラの使用方法を学習します。

下記も参照

STソースファイルのメニュー (ページ 41)
STソースファイルのコンテキストメニュー (ページ 41)
キーボードショートカット (ページ 27)

2.3.1 ST ソースファイルの挿入

ST ソースファイルは、ソースファイルを実行する SIMOTION デバイスに割り付けます。

実行手順

1. プロジェクトナビゲータで、該当する SIMOTION デバイスを開きます。
2. **[PROGRAMS]**フォルダを選択します。
3. **[Insert|Program|ST source file]**メニューを選択します。
4. ST ソースファイルの名前を入力します。

   プログラムソースファイルの名前は識別子のルールを満たしている必要があります。ソースファイルの名前は、アルファベット(A～Z、a～z)、数字(0～9)、または下線(_)を任意の順序で組み合わせたものです。最初の文字は、アルファベットまたは下線にする必要があります。アルファベットの大文字と小文字は区別されません。

   名前の許容される長さは、SIMOTION Kernel のバージョンによって異なります。

   – SIMOTION Kernel バージョン V4.1 の場合: 最大 128 文字
   – SIMOTION Kernel バージョン V4.0 まで: 最大 8 文字

   名前は SIMOTION デバイス内で一意である必要があります。

   保護識別子または予約識別子(予約識別子 (ページ 59)を参照)は使用できません。

   既存のプログラムソース(たとえば、ST ソースファイル、MCC ユニットなど)が表示されます。
5. 必要な場合、その他のタブを選択して、ローカル設定(この ST ソースファイルにのみ有効)を行います。
   – **[Compiler]**タブ: コンパイラのローカル設定(STコンパイラのローカル設定 (ページ 32)を参照)
   – **[Additional settings]**タブ: プリプロセッサの定義(プリプロセッサ定義の作成 (ページ 35)を参照)
6. **[Open editor automatically]**チェックボックスを選択します。
7. **[OK]**を選択して確定します。

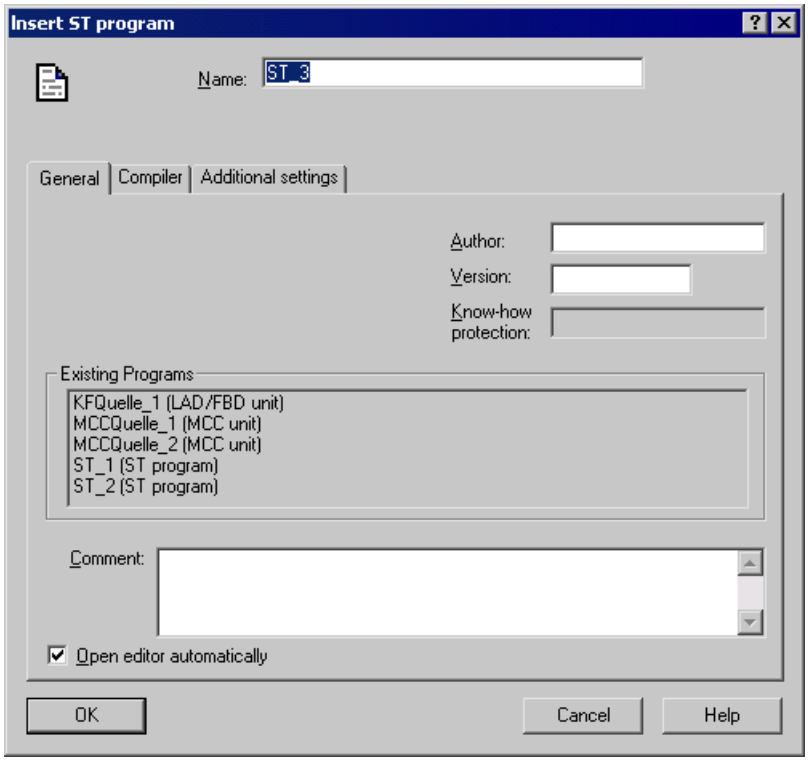


図 2-3　　ST ソースファイルの挿入

通知

SIMOTION Kernel V4.0 までのバージョンでは、プログラムソースファイル名の許容される長さに違反しても、プログラムソースファイルの一貫性チェックまたはダウンロードが実行されるまで違反が検出されないことがあります!

2.3.2　既存の ST ソーフファイルを開く

実行手順

1. プロジェクトナビゲータで、該当する SIMOTION デバイスのサブツリーを開きます。
2. **[PROGRAMS]**フォルダを開きます。
3. 目的の ST ソースファイルを選択します。
4. **[Edit|Open object]**メニューコマンドを選択します。

注記

必要な ST ソースファイルをダブルクリックしてファイルを開くこともできます。

2.3.3 ST ソースファイルの特性の変更

手順

1. SIMOTION デバイスの下で、**[PROGRAMS]**フォルダを開きます。
2. 目的の ST ソースファイルを選択します。
3. **[Edit|Object Properties]**メニューコマンドを選択します。
4. 必要な場合、その他のタブを選択して、ローカル設定(この ST ソースファイルにのみ有効)を行います。
   - **[Compiler]**タブ: コードの生成とメッセージの表示のためのコンパイラのローカル設定 (STコンパイラのローカル設定 (ページ 32)を参照)
   - **[Additional settings]**タブ: プリプロセッサの定義 (プリプロセッサ定義の作成 (ページ 35)を参照)
   - **[Compilation]**タブ: 現在のコンパイラオプションの表示

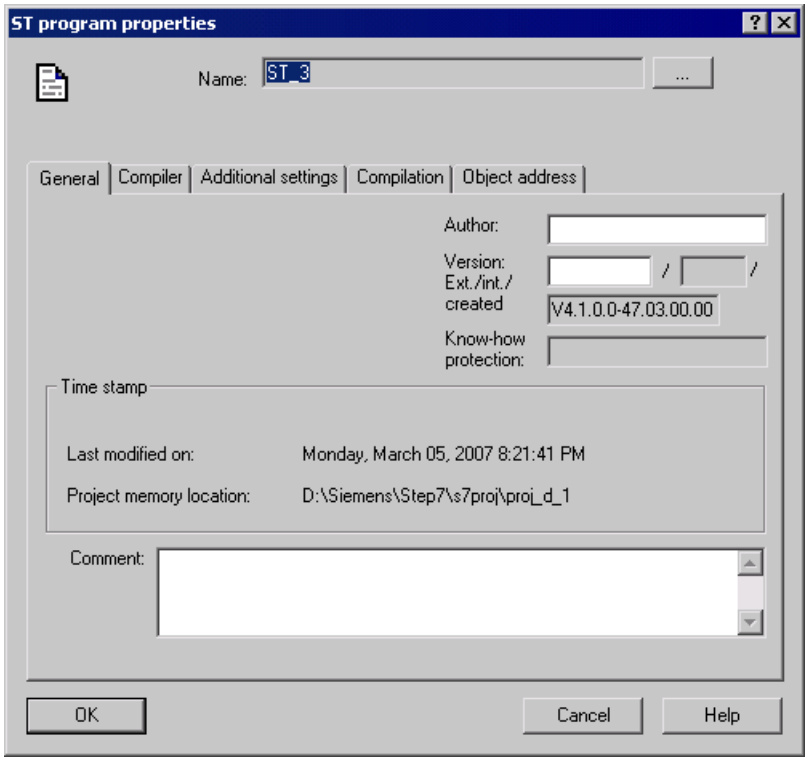


図 2-4 ST ソースファイルの特性の変更

ST ソースファイルの名前の変更

ここでは、ST ソースファイルの名前を変更することもできます。これを行うには、[...]ボタンをクリックします。

プログラムソースファイルの名前は識別子のルールを満たしている必要があります。ソースファイルの名前は、アルファベット(A～Z、a～z)、数字(0～9)、または下線(_)を任意の順序で組み合わせたものです。最初の文字は、アルファベットまたは下線にする必要があります。アルファベットの大文字と小文字は区別されません。

名前の許容される長さは、SIMOTION Kernel のバージョンによって異なります。

- SIMOTION Kernel バージョン V4.1 の場合: 最大 128 文字
- SIMOTION Kernel バージョン V4.0 まで: 最大 8 文字

名前は SIMOTION デバイス内で一意である必要があります。

保護識別子または予約識別子(予約識別子 (ページ 59)を参照)は使用できません。

既存のプログラムソース(たとえば、ST ソースファイル、MCC ユニットなど)が表示されます。

通知

SIMOTION Kernel V4.0 までのバージョンでは、プログラムソースファイル名の許容される長さに違反しても、プログラムソースファイルの一貫性チェックまたはダウンロードが実行されるまで違反が検出されないことがあります!

2.3.4 ST エディタの操作

ST エディタを使用すると、以下のオペレータ制御を通じて、ST ソースファイル、変数、およびテクノロジーオブジェクトをより容易に操作できるようになります。

- 構文の色付け
- ドラッグアンドドロップ
- メニューコマンドおよびショートカット

下記も参照

キーボードショートカット (ページ 27)

2.3.4.1 構文の色付け

構文の色付け

ST エディタでは、言語要素はさまざまな色で表されます。

- 青: キーワードおよびコンパイラ内蔵ファンクション
- マゼンタ: 数字、値
- 緑:説明
- 黒: テクノロジーオブジェクト、ユーザコード、変数

2.3.4.2 ドラッグアンドドロップ

ドラッグアンドドロップ

ドラッグアンドドロップ操作(マウスの左ボタンを押しながらドラッグ)を使用すると、以下のことを実行できます。

- ST ソースファイル内で、または開いている別の ST ソースファイルに選択したテキスト領域を移動する。
- シンボルブラウザから ST ソースファイルに変数の名前をコピーする。
- プロジェクトナビゲータから ST ソースファイルに名前(たとえば、テクノロジーオブジェクト、ファンクション、ファンクションブロックなどの名前)をコピーする。
- コマンドライブラリから ST ソースファイルにシステムファンクションをコピーする。

シンボルブラウザから ST ソースファイルに変数の名前をコピーするには

1. シンボルブラウザで、目的の変数の行全体を選択します。これを行うには、行の先頭にある行番号をクリックします。
2. マウスの左ボタンを押し、ST ソースファイル内の目的の位置に行番号をドラッグします。

   ST ソースファイルに選択した変数の名前が挿入されます。

プロジェクトナビゲータから ST ソースファイルに要素(たとえば、テクノロジーオブジェクト、ファンクション、ファンクションブロックなど)の名前をコピーするには

1. プロジェクトナビゲータで[Project]タブを選択します。
2. プロジェクトナビゲータで要素を選択します。
3. マウスの左ボタンを押し、ST ソースファイル内の目的の位置に要素をドラッグします。

   ST ソースファイルに選択した要素の名前が挿入されます。

コマンドライブラリから ST ソースファイルにシステムファンクションをコピーするには

1. プロジェクトナビゲータで[Command Library]タブを選択します。
2. コマンドライブラリでシステムファンクションを選択します。
3. マウスの左ボタンを押し、ST ソースファイル内の目的の位置にシステムファンクションをドラッグします。

   ST ソースファイルにシステムファンクションがそのパラメータと共に挿入されます。

2.3.4.3 キーボードショートカット

ST エディタにはキーボードショートカットも用意されています。コマンドは、**[Edit]**メニューと**[ST editor]**メニューを使用して呼び出すことができます。

表 2-1 ST エディタのキーボードショートカット

キーボードショートカット	説明
CTRL+Z	最後のアクションを元に戻す(**[Edit|Undo]**メニュー)
CTRL+Y	最後のアクションをやり直す(**[Edit|Redo]**メニュー)
CTRL+C	選択した領域をクリップボードにコピーする (**[Edit|Copy]**メニュー)
CTRL+V	クリップボードの内容を貼り付ける(**[Edit|Paste]**メニュー)
CTRL+A	すべて選択する(**[Edit|Select all]**メニュー)
CTRL+F	ST ソースファイル内のテキストを検索する(**[Edit|Find]**メニュー)
CTRL+H	ST ソースファイル内のテキストを置換する(**[Edit|Replace]**メニュー)
CTRL+X	選択した領域を切り取る(**[Edit|Cut]**メニュー)
CTRL+B	ST ソースを保存およびコンパイルする(**[ST source|Save and compile]**メニュー)
CTRL+F4	ST ソースを閉じる(**[ST source|Close]**メニュー)
ソースファイルの エクスポート/インポート	ST ソースを ST フォーマットのファイルとしてエクスポートし、他のプロジェクトに インポートすることが可能(**[ST source|Export]**メニュー)
DEL	選択した領域を削除する(**[Edit|Delete]**メニュー)

2.3.4.4 ST エディタの設定

手順:

1. **[Tools|Settings]**メニューを選択します。
2. **[ST editor / Scripting]**タブを選択します。
3. 設定を入力します。
4. **[OK]**または**[Accept]**をクリックして確定します。

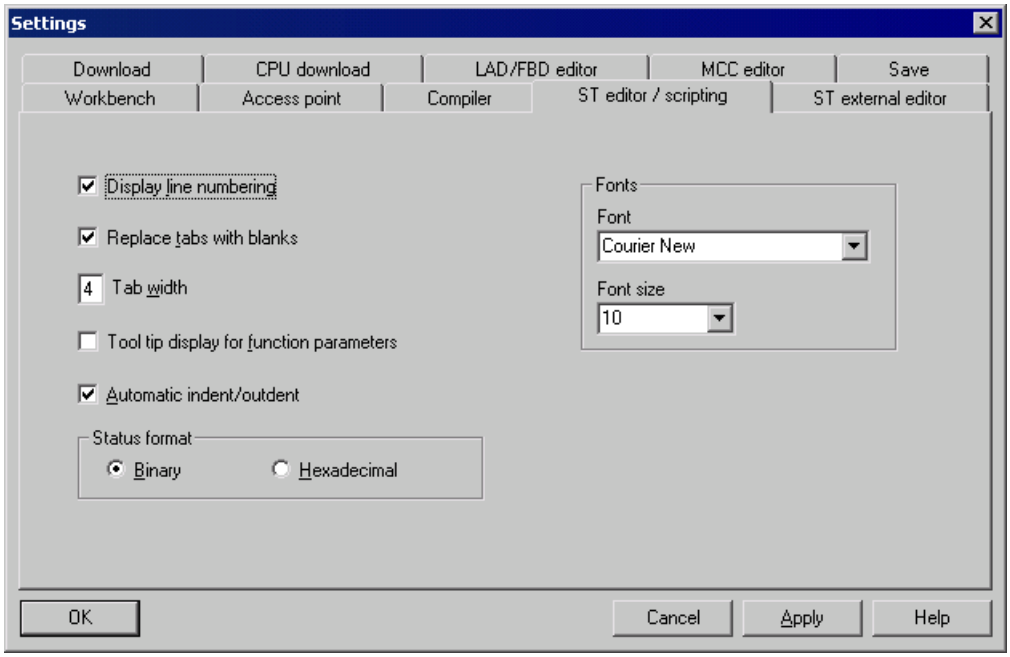


図 2-5 [ST Editor / Scripting]

設定はスクリプトエディタにも適用されます。

以下の表に、個々のパラメータの説明を示します。

表 2-2 [ST Editor / Scripting]のパラメータ設定

パラメータ	説明
Display line numbering	有効にすると、行番号が表示されます。
Replace tabs with blanks	有効にすると、タブのエントリが対応する数の空白で置換されます。
Tab width	タブが進む文字数
Tooltip display for function parameters	有効にすると、パラメータがファンクションのツールヒントとして表示されます。
Automatic indent/outdent	有効にすると、ソースファイルセクションとブロックが自動的にインデントされます。
Status format	プログラムステータスに変数値を表示するフォーマット(プログラムステータスの特徴 (ページ 233)を参照) (STエディタの場合のみ)
Font	ST エディタにテキストを表示するフォント。PC にインストールされているすべての非等間隔フォントを選択できます。
Font size	ST エディタにテキストを表示するフォントサイズ(pt)

2.3.4.5 コマンドライブラリの使用

コマンドライブラリは、プロジェクトナビゲータのタブです。コマンドライブラリには、使用可能なシステムファンクション、システムファンクションブロック、および演算子が含まれています。

コマンドライブラリから ST エディタウィンドウにこれらの要素をドラッグアンドドロップすることができます。

2.3.5 ST コンパイラの起動

必要条件

ST エディタを使用して ST ソースファイルが開かれていること。

実行手順

1. ST エディタを含むウィンドウをクリックします。ダイナミックな ST ソースファイルメニューが表示されます。
2. **[ST source file|Save and compile]**メニューコマンドを選択します。

注記

ST ソースファイルのメニューはダイナミックです。このメニューは、ST エディタのウィンドウが有効になっている場合にのみ表示されます。

コンパイラによって ST ソースファイルの構文がチェックされます。詳細ビューの**[Compile/check output]**タブに、ソーステキストの成功したコンパイル、またはコンパイラエラーが表示されます。エラーの詳細には、 ST ソースファイルの名前、エラーが発生した行の番号、エラー番号、および エラーの説明が含まれます。

2.3.5.1 エラーの修正のためのヘルプ

エラーの修正中にヘルプを表示するには

- 詳細ビューの**[Compile/check output]**タブで、エラーメッセージをダブルクリックします。

ST ソースファイルの関連する行にカーソルが置かれます。

2.3.5.2 ST エディタのツールバー

このツールバーには、重要なプログラミングコマンドボタン([Insert ST source file]、[Accept and compile]、[Monitor]など)が含まれています。

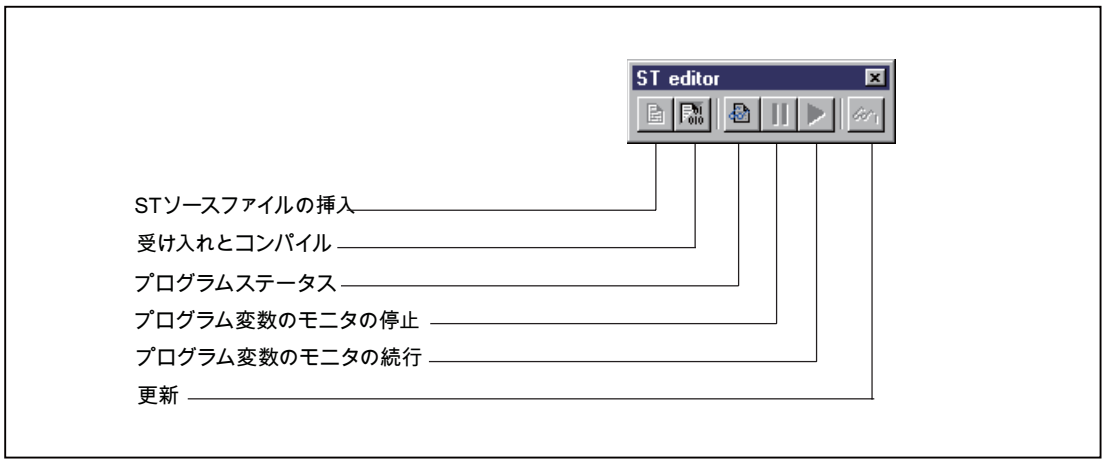
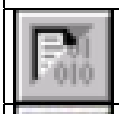
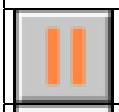


図 2-6 ST エディタのツールバー

ST エディタのファンクションバーを介して主要な機能を実行できます。

表 2-3　　ST エディタのファンクション

ファンクション	説明
	このボタンをクリックすると、新しい ST ソースファイルが作成されます。このボタンは、ST ソースファイルを保存する[PROGRAMS]フォルダがプロジェクトナビゲータで選択されている場合にのみ表示されます。
	このボタンをクリックすると、現在の ST ソースファイルが保存およびコンパイルされます。 コンパイル時に、ST ソースファイルは実行可能なプログラムコードでコンパイルされます。
	有効な ST ソースファイルの現在のステータスを監視するには、このボタンをクリックします。 ステータスを監視している間、プログラム実行中のプログラム変数値を監視することができます。 ST エディタは 2 つのウィンドウに分割されます。右側のペインには、ST ソースファイル(左側のペイン)で選択したプログラム変数の現在値が表示されます。 **ターゲットシステムでプロジェクトとプログラムを使用できる必要があり、また、ターゲットシステムへの ONLINE 接続が有効になっている必要があります。**
	このボタンをクリックすると、プログラム変数の監視が停止します。
	このボタンをクリックすると、プログラム変数の監視が続行されます。

2.3.6　ST コンパイラの設定

2.3.6.1　ST コンパイラの設定

ST コンパイラの設定は、次のように定義できます。

- SIMOTIONプロジェクト、すべての適用可能なプログラミング言語にグローバルに定義。コンパイラのグローバル設定 (ページ 31)を参照してください。
- SIMOTIONプロジェクト内の個々のSTソースにローカルに定義。STコンパイラのローカル設定 (ページ 32)を参照してください。

2.3.6.2 コンパイラのグローバル設定

グローバル設定は、SIMOTION プロジェクト内のすべてのプログラミング言語に対して有効です。

手順

1. **[Tools|Settings]**メニューを選択します。.
2. **[Compiler]**タブを選択します。
3. 以下の表に従って設定を定義します。
4. **[OK]**を選択して確定します。

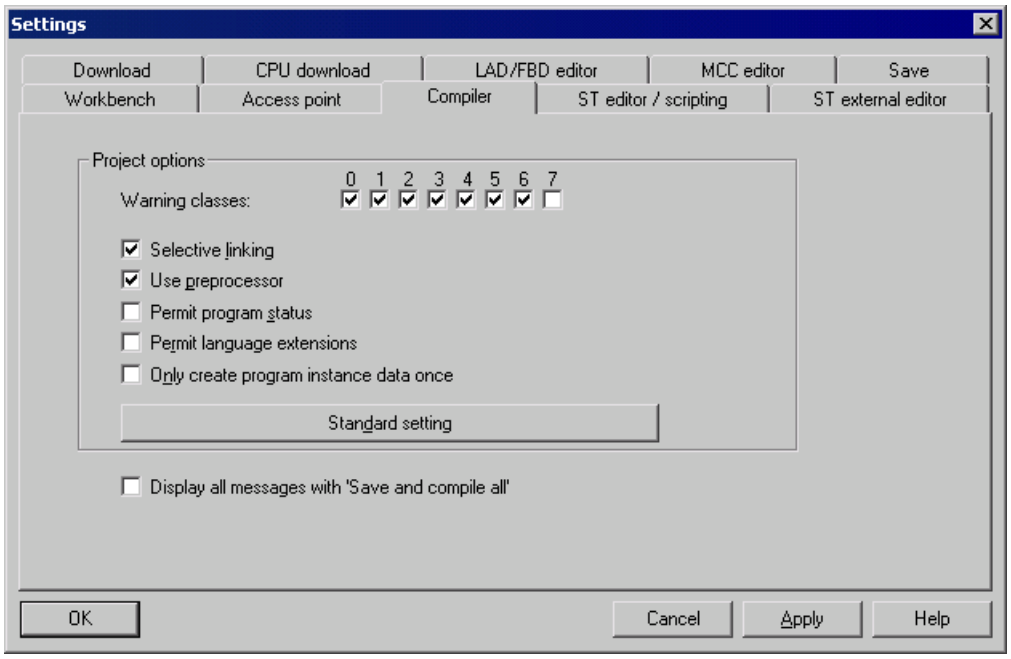


図 2-7 コンパイラのグローバル設定

パラメータ

表 2-4 コンパイラのグローバル設定のパラメータ

パラメータ	説明
Warning classes[1]	**有効**: エラーメッセージ加えて、コンパイラから選択したクラスの警告メッセージが出力されます。 **無効**: コンパイラは各クラスの警告メッセージを抑制します。 警告クラスの意味 (ページ 34)も参照してください。
Selective linking[1]	**有効**(標準): 実行可能プログラムから未使用のコードが削除されます。 **無効**: 実行可能プログラムに未使用のコードが保持されます。
Use preprocessor[1]	**有効**: プリプロセッサが使用されます(プリプロセッサの制御 (ページ 215)を参照)。 **無効**(標準): プリプロセッサは使用されません。

パラメータ	説明
Permit program status[1]	**有効**: 追加のプログラムコードが生成されてプログラム変数(ローカル変数を含む)の監視が有効になります(プログラムステータスの特徴 (ページ 233)を参照)。 **無効**(標準): プログラムステータスは実行できません。
Permit language extensions[1]	**有効**: IEC 61131-3 に準拠しない言語要素を使用できます。 **無効**(標準): IEC 61131-3 に準拠する言語要素だけを使用できます。
Only create program instance data once[1]	**有効**: プログラムのローカル変数がユニットのユーザメモリに 1 回だけ格納されます。プログラム内で追加のプログラムを呼び出すときはこの設定が必要です。 **無効**(標準): 各タスクのユーザメモリにおけるタスク割り付けに従ってプログラムのローカル変数が格納されます。 変数タイプのメモリ範囲 (ページ 178)を参照してください。
Display all messages with *Save and compile all*[2]	ここでは、SIMOTION SCOUT で**[Save and compile all]**コマンドを呼び出したときにワークベンチの詳細ビューに表示されるエラーログのスコープを制限することができます。 **有効**: ST ソースファイルを 1 回コンパイルした場合と同じような詳細ログが作成されます。 **無効**: 圧縮されたエラーログが作成されます。
[1] ローカル設定も可能(STコンパイラのローカル設定 (ページ 32)を参照) [2] ユーザ固有の設定。ユーザが処理するすべての SIMOTION プロジェクトに有効です。	

通知

設定が有効になるように、プロジェクトを再コンパイルする必要があることがあります。

2.3.6.3 ST コンパイラのローカル設定

ローカル設定は、ST ソースファイルごとに個別に設定します。ローカル設定はグローバル設定を上書きします。

手順

1. STソースファイルの[Properties]ウィンドウを開きます(STソースファイルの特性の変更 (ページ 24)を参照)。

   プロジェクトナビゲータで ST ソースファイルを選択し、**[Edit|Object properties]**メニューコマンドを選択します。
2. **[Compiler]**タブを選択します。
3. 以下の表に従って設定を定義します。
4. **[OK]**を選択して確定します。

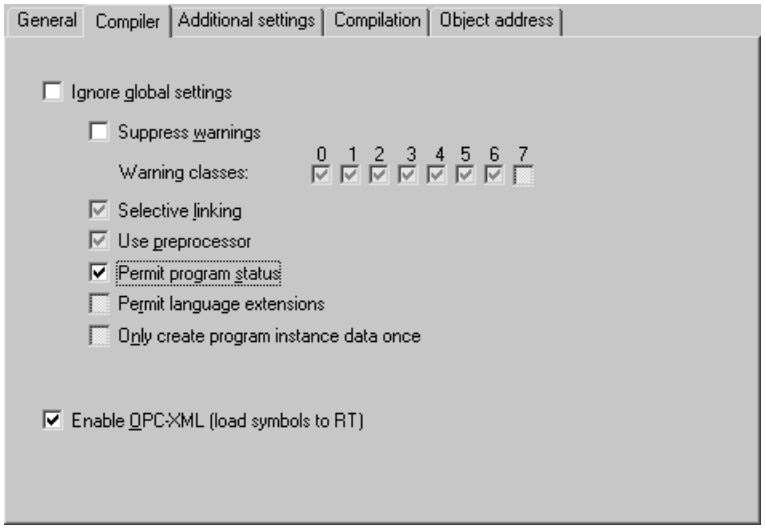


図 2-8　　ST コンパイラのローカル設定

パラメータ

表 2-5　　ST コンパイラのローカル設定のパラメータ

パラメータ	説明
Ignore global settings	影響: • Warning classes • Selective linking • Use preprocessor • Permit program status • Permit language extensions • Only create program instance data once **有効**: 選択したローカル設定だけが適用されます。グローバル設定は無視されます。 **無効**: それぞれのグローバル設定を採用できます。対応するチェックボックスはグレー表示されます。
Suppress warnings	エラーメッセージに加えて、コンパイラから警告が出力されます。この出力される警告メッセージのスコープを設定することができます。 **有効**: コンパイラは、警告クラスのグローバル設定の選択に従って警告メッセージを出力します。警告クラスのチェックボックスは選択できなくなります。 **無効**: コンパイラは、警告クラスのその後の選択に従って警告メッセージを出力します。
Warning classes[1]	[Suppress warnings]が無効の場合のみ **有効**: コンパイラから選択したクラスの警告メッセージが出力されます。 **無効**: コンパイラは各クラスの警告メッセージを抑制します。 **グレーの背景**: 表示されているグローバル設定が採用されます([Ignore global settings]が無効の場合のみ)。 警告クラスの意味 (ページ 34)を参照してください。
Selective linking[1]	**有効**: 実行可能プログラムから未使用のコードが削除されます。 **無効**: 実行可能プログラムに未使用のコードが保持されます。 **グレーの背景**: 表示されているグローバル設定が採用されます([Ignore global settings]が無効の場合のみ)。

パラメータ	説明
Use preprocessor[1]	**有効**: プリプロセッサが使用されます(プリプロセッサの制御 (ページ 215)を参照)。 **無効**: プリプロセッサは使用されません。 **グレーの背景**: 表示されているグローバル設定が採用されます([Ignore global settings]が無効の場合のみ)。
Permit program status[1]	**有効**: 追加のプログラムコードが生成されてプログラム変数(ローカル変数を含む)の監視が有効になります(プログラムステータスの特徴 (ページ 233)を参照)。 **無効**: プログラムステータスは実行できません。 **グレーの背景**: 表示されているグローバル設定が採用されます([Ignore global settings]が無効の場合のみ)。
Permit language extensions[1]	**有効**: IEC 61131-3 に準拠しない言語要素を使用できます。 **無効**: IEC 61131-3 に準拠する言語要素だけを使用できます。 **グレーの背景**: 表示されているグローバル設定が採用されます([Ignore global settings]が無効の場合のみ)。
Only create program instance data once[1]	**有効**: プログラムのローカル変数がユニットのユーザメモリに 1 回だけ格納されます。プログラム内で追加のプログラムを呼び出すときはこの設定が必要です。 **無効**: 各タスクのユーザメモリにおけるタスク割り付けに従ってプログラムのローカル変数が格納されます。 **グレーの背景**: 表示されているグローバル設定が採用されます([Ignore global settings]が無効の場合のみ)。 変数タイプのメモリ範囲 (ページ 178)を参照してください。
Enable OPC-XML	SIMOTION デバイスで、ST ソースファイルのインターフェースセクションのユニット変数に関するシンボル情報を使用できます(*_exportUnitDataSet* および *_importUnitDataSet* ファンクションに必要。*『SIMOTION 基本機能』*機能マニュアルを参照)。
[1] グローバル設定も可能(コンパイラのグローバル設定 (ページ 31)を参照)	

2.3.6.4 警告クラスの意味

以下の表に、警告クラスとその意味をリストします。

表 2-6 警告クラスの意味

警告クラス	意味
0	非参照/未使用のコード、データなど
1	非表示の識別子
2	タイプ変換に関する警告、データ変更に関する警告
3	設定したコンパイラオプションに関する警告
4	セマフォ(エラーが発生する可能性のある機能)に関する警告
5	アラーム機能に関する警告
6	ライブラリのコンストラクト(宣言したユニット変数)に関する警告
7	プリプロセッサのメッセージ(プリプロセッサの制御 (ページ 215)を参照)

2.3.7 ST ソースファイルのノウハウ保護

権限のない第三者によるアクセスから ST ソースファイルを保護することができます。このような保護された ST ソースファイルは、パスワードを入力しないと開いて表示することはできません。

ノウハウ保護の適用方法については、オンラインヘルプを参照してください。

注記

ノウハウ保護された ST ソースファイルを通常のテキストファイルとして エクスポートすることはできません。

ただし、XML フォーマットで エクスポートすることはできます。ST ソースファイルは暗号化されて エクスポートされます。暗号化された XML ファイルをインポートする場合、ログインとパスワードを含め、ノウハウ保護はそのままです。

下記も参照

ライブラリのノウハウ保護 (ページ 206)

2.3.8 プリプロセッサ定義の作成

STソースファイルの[Properties]ダイアログボックスで、プリプロセッサ定義を作成することができます(プリプロセッサの制御 (ページ 215)を参照)。これにより、ノウハウ保護されたSTソースファイルを使用してプリプロセッサの制御もできるようになります(STソースファイルのノウハウ保護 (ページ 35)を参照)。

[Properties]ダイアログボックスにおけるプリプロセッサ定義の作成

1. STソースファイルの[Properties]ウィンドウを開きます
   (STソースファイルの特性の変更 (ページ 24)を参照)。

   プロジェクトナビゲータで ST ソースファイルを選択し、**[Edit|Object properties]**メニューコマンドを選択します。
2. **[Additional settings]**タブを選択します。
3. プリプロセッサ定義(以下の表に示す構文)を入力します。
4. **[OK]**を選択して確定します。

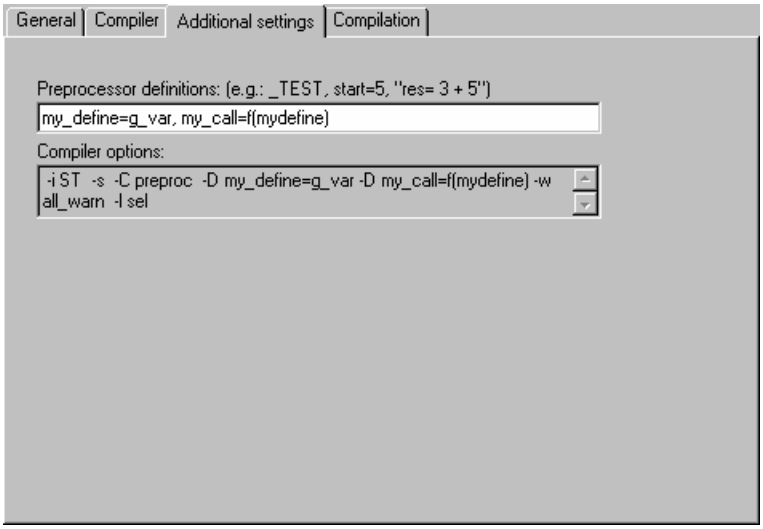


図 2-9　　プリプロセッサ定義

表 2-7　　プリプロセッサ定義の構文

構文	意味
Identifier=text *'Identifier=text'* *"Identifier=text"*	指定した *identifier* が定義され、ST ソースファイルで、指定した *text* によって置換されます。 テキストで使用できる文字については、プリプロセッサステートメント (ページ 215)を参照してください。 (たとえばテキスト内で)式に空白が含まれる場合、構文 *"Identifier=text"* を使用する必要があります。
Identifier	指定した *identifier* が定義され、ST ソースファイルで、空のテキストによって置換されます。

注記

プラグマを使用して ST ソースファイル内で作成されたプリプロセッサ定義は、[Properties] ダイアログボックスの定義を上書きします。

プリプロセッサステートメント (ページ 215)の情報に注意してください!

2.3.9 ST ソースファイルのエクスポート、インポート、および印刷

ここでは、ST ソースファイルのエクスポート、インポート、および印刷の概要を説明します。

ST ソースファイルを ASCII ファイルとしてエクスポートするには

1. ST ソースファイルを開きます。
2. カーソルが ST エディタ内にあることを確認します。
3. **[ST source file|Export]**メニューコマンドを選択します。
4. ASCII ファイルのパスとファイル名を入力し、**[Save]**をクリックして確定します。

ST ソースファイルが ASCII ファイルとして保存されます。ファイル名にはデフォルト拡張子*.st が付与されます。

注記

ノウハウ保護された ST ソースファイルを通常のテキストファイルとしてエクスポートすることはできません。ただし、XML フォーマットでエクスポートすることはできます。

2.3.9.1 ST ソースファイルの XML フォーマットでのエクスポート

ST ソースファイルを XML フォーマットでエクスポートするには、以下の手順に従ってください。

1. プロジェクトナビゲータで ST ソースファイルを選択します。
2. コンテキストメニュー**[Expert|Save project and export object]**を選択します。
3. XML エクスポートのパスを指定し、**[OK]**で確定します。

指定したパスに、ST ソースファイルの名前が付いた XML ファイルと、関連する追加の XML ファイルを含むフォルダが保存されます。

注記

ノウハウ保護された ST ソースファイルも XML フォーマットでエクスポートできます。ST ソースファイルは暗号化されてエクスポートされます。暗号化された XML ファイルをインポートする場合、ログインとパスワードを含め、ノウハウ保護はそのままです。

2.3.9.2 テキストファイル(ASCII)の ST ソースファイルとしてのインポート

ASCII ファイルを ST ソースファイルとしてインポートするには

1. プロジェクトナビゲータで、該当する SIMOTION デバイスの下にある**[PROGRAMS]**フォルダを選択します。
2. **[Insert|External source|ST source file]**メニューを選択します。
3. インポートする ASCII ファイルを選択し、**[Open]**をクリックして確定します。

   ST ソースファイルを挿入するためのダイアログボックスが表示されます。
4. STソースファイルの名前を入力し、追加オプションを選択します(STソースファイルの挿入 (ページ 22)を参照)。

ASCII ファイルが現在のプロジェクトディレクトリに ST ソースファイルとして組み込まれ、ファイルを開くことができます。

2.3.9.3 ST ソースファイルへの XML データのインポート

XML データを ST ソースファイルにインポートするには、以下の手順に従ってください。

1. 適用可能な場合、新しいSTソースファイルを挿入します(STソースファイルの挿入 (ページ 22)を参照)。
2. プロジェクトナビゲータで ST ソースファイルを選択します。
3. コンテキストメニュー**[Expert|Import object]**を選択します。
4. インポートする XML データを選択します。

インポートした XML データにより、選択した ST ソースファイルの既存のデータが上書きされます。プロジェクト全体が保存され、再コンパイルされます。

注記

インポートする XML データがノウハウ保護されている ST ソースファイルからエクスポートされたものである場合、暗号化された XML ファイルをインポートする間、ログインとパスワードを含め、ノウハウ保護はそのままになります。

2.3.9.4 ST ソールファイルの印刷

ST ソースファイルを印刷するには

1. ST ソースファイルを開きます。
2. カーソルが ST エディタ内にあることを確認します。
3. **[Project|Print]**メニューを選択します。

プログラムが名前と日付と共に印刷されます。

2.3.10 外部エディタの使用

使用できる外部エディタ

デフォルトの ST エディタの代わりに、次の機能をサポートする他の ASCII エディタを使用することができます。

- 有効なウィンドウで外部プログラム(たとえばコンパイラ)を呼び出し、実行できる。

また、外部エディタでは、ST ソースファイルの特定のテキスト節を色で強調表示できる必要があります(構文の色付け)。

注記

外部エディタを使用する場合、ダイナミックな ST ソースファイルメニューとそのエントリは使用できません。対応するツールバーも無効になります。

外部エディタから ST ソースファイルのコンパイルを開始できる必要があります。

ST エディタではステータスプログラムが続行します。

外部エディタを使用するための設定

設定は SCOUT ワークベンチで行います。

1. **[Tools|Settings]**メニューを選択します。
2. **[ST external editor]**タブを選択します(図を参照)。
3. **[Use external ST editor]**チェックボックスを有効にします。
4. 外部エディタのパスを入力します。
   - **[Browse...]**をクリックし、エディタのパスとファイル名を選択します。

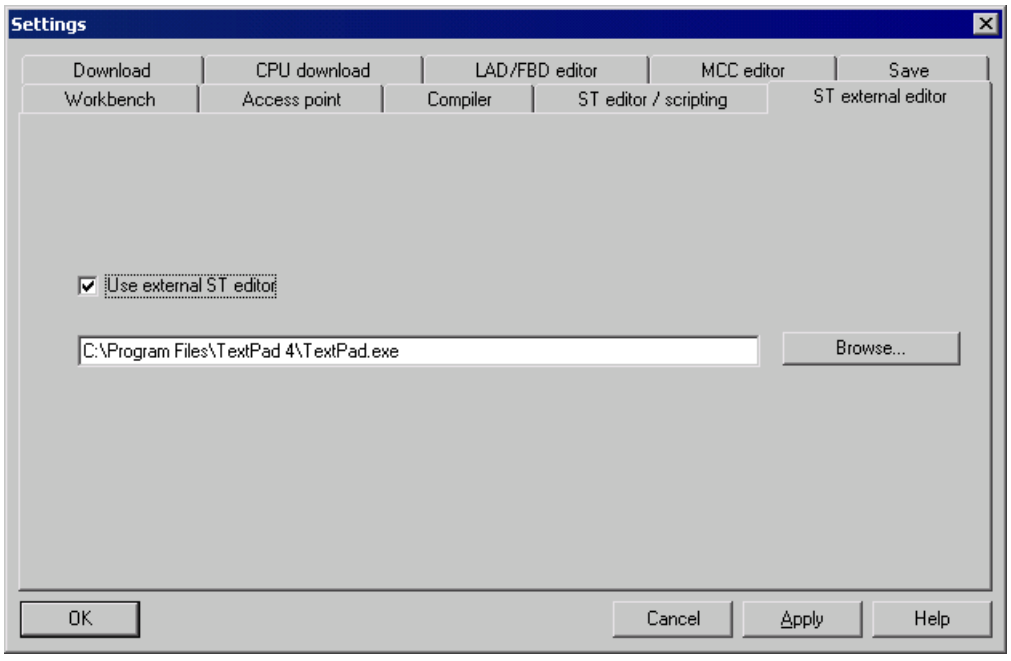


図 2-10 外部エディタを使用するための設定

外部エディタでの設定

以下の注意は一般的な性質のものです。外部エディタのオペレータ指示を比較してください。

1. 外部エディタに ST コンパイラへのパスを設定します。コンパイラは STEP7 のインストールディレクトリ s7bin\u7wstcax.exe に置かれています。
2. 各種のエディタには構文ファイルが用意されています。これらの構文ファイルを使用すると、エディタのテキスト節を色で強調表示することができます(構文の色付け)。
   構文ファイルを関連ディレクトリにコピーし、エディタをしかるべく設定します。

外部エディタを使用する際の注意事項

注意

プロジェクトを閉じるか、SIMOTION SCOUT を終了する前に、外部エディタのすべてのウィンドウを閉じます。ウィンドウを閉じないと、データが失われる可能性があります!

2.3.11 ST ソースファイルのメニュー

2.3.11.1 ST ソースファイルのメニュー

有効なアプリケーション/エディタまたはモード(ONLINE/OFFLINE)により、特定のコマンドが表示されなかったり選択できなかったりします。メニューは、作業エリアで ST エディタが有効になっている場合にのみ表示されます。

次のファンクションを選択できます。

表 2-8 ST ソースファイルのメニュー

ファンクション	意味/注
Close	**[Close]**を選択すると、有効な ST ソースファイルが閉じます。
Characteristics	有効な ST ソースファイルの特性を表示するには、**[Properties]**を使用します。このウィンドウには、名前、変更日、および格納先が表示されます。
Accept and compile	**[Save and compile]**を選択すると、有効な ST ソースファイルが実行可能コードに変換され、保存されます。
Use preprocessor	オプションとして、プリプロセッサがコンパイル前に ST ソースファイルをスキャンして、たとえばファイル内の文字列を置換するということが可能です。これは、コンパイル中に考慮されます。このメニューコマンドを使用すると、プリプロセッサステートメントを明確に実行することができます。
Export	**[Export]**を選択すると、たとえば他のプロジェクトにプログラムをインポートするために、有効な ST ソースファイルが ST フォーマットのファイルとして エクスポートされます。
Program status On/Off	**[Program status On/Off]**を選択すると、有効な ST ソースファイルの現在のステータスが監視されます。ステータスを監視している間、プログラム実行中のプログラム変数値を監視することができます。ST エディタは 2 つのウィンドウに分割されます。右側のペインには、ST ソースファイル(左側のペイン)で選択したプログラム変数の現在値が表示されます。 **ターゲットシステムでプロジェクトとプログラムを使用できる必要があり、また、ターゲットシステムへの ONLINE 接続が有効になっている必要があります。**

2.3.11.2 ST ソースファイルのコンテキストメニュー

有効なアプリケーション/エディタまたはモード(ONLINE/OFFLINE)により、特定のコマンドが表示されなかったり選択できなかったりします。

次のファンクションを選択できます。

表 2-9 ST ソースファイルのコンテキストメニュー

ファンクション	意味/注
Close	**[Close]**を選択すると、有効な ST ソースファイルが閉じます。
Cut	選択したオブジェクトを削除し、クリップボードに保存するには、**[Cut]** を選択します。
Copy	選択したオブジェクトをコピーするには、**[Copy]**を用します。オブジェクトはクリップボードに格納されます。
Inserting	クリップボードの内容を現在のカーソル位置に貼り付けるには、**[貼り付け]**を選択します。
Deleting	現在の ST ソースファイルを削除するには、**[Delete]**を使用します。ST ソースファイルのすべてのデータが永久的に削除されます。
Rename	現在の ST ソースファイルの名前を変更するには、**[Rename]**を使用します。名前の変更と共にこの名前に対する参照を変更することはできないこと、また、新しい名前は ST の規則に準拠していなければならないことに注意してください。

ファンクション	意味/注
Save variables	このメニューコマンドでは、保持型変数、ユニット変数、およびグローバル変数を保存できます。ターゲットデバイスの RAM/ROM メモリからこれらの変数を保存し、データ媒体に XML ファイルとして格納することができます。これらの変数を復元するときは、データ媒体からターゲットデバイスの RAM/ROM メモリに書き込むことができます。
Restore variables	このメニューコマンドでは、以前にエクスポートした変数から保持型変数、ユニット変数、およびグローバル変数を復元することができます。これらの変数を復元するときは、データ媒体からターゲットデバイスの RAM/ROM メモリに書き込むことができます。
エキスパート	
Import object	**[Import object]**を選択すると、選択的な XML エクスポートで以前に作成した別のプロジェクトから ST ソースファイルのデータがインポートされます。
Save project and export object	ST ソースファイルの選択したデータを XML フォーマットでエクスポートするには、**[Save project and export object]**を使用します。エクスポートデータは、他のプロジェクトに再度インポートできます。
Accept and compile	選択した ST ソースファイルを保存およびコンパイルするには、**[Accept and compile]**を使用します。
Run preprocessor	オプションとして、プリプロセッサがコンパイル前に ST ソースファイルをスキャンして、たとえばファイル内の文字列を置換するということが可能です。これは、コンパイル中に考慮されます。このメニューコマンドを使用すると、プリプロセッサステートメントを明確に実行することができます。
Program status On/Off	**[Program status On/Off]** を選択すると、有効な ST ソースファイルの現在のステータスが監視されます。ステータスを監視している間、プログラム実行中のプログラム変数値を監視することができます。ST エディタは 2 つのウィンドウに分割されます。右側のペインには、ST ソースファイル(左側のペイン)で選択したプログラム変数の現在値が表示されます。 ターゲットシステムでプロジェクトとプログラムを使用できる必要があり、また、ターゲットシステムへの ONLINE 接続が有効になっている必要があります。
Export	**[Export]**を選択すると、たとえば他のプロジェクトにプログラムをインポートするために、有効な ST ソースファイルが ST フォーマットのファイルとしてエクスポートされます。
ノウハウ保護	
Set	ST ソースファイルにノウハウ保護を設定するには、**[Set know-how protection]**を使用します。保護されたソースは、指定されたログオンおよびパスワードを使用しないと開いて修正することはできません。
Deleting	**[Delete know-how protection]**を選択すると、パスワードを入力しなくても保護された ST ソースファイルを開いて読み取ることができるように、ファイルが解放されます。
基準データ	
Create	選択した ST ソースファイルの基準データを作成するには、**[Create reference data]**を選択します。基準データには、使用されている指定子に関する情報と、その宣言、適用、ファンクション呼び出し、および呼び出しのネストの詳細が含まれています。
表示	
Cross references	**[Display cross references]**を選択すると、選択した ST ソースファイルのクロスリファレンスリストが詳細ビューに表示されます。クロスリファレンスを表示するには、基準データを作成しておく必要があります。
Program structure	選択した ST ソースファイルのプログラム構造を詳細ビューに表示するには、**[Display program structure]**を選択します。プログラム構造を表示するには、最初に基準データを作成しておく必要があります。
Print	選択した ST ソースファイルを印刷するには、**[Print]**を使用します。
Print preview	印刷するページのプレビューを行うには、**[印刷プレビュー]**を選択します。
Characteristics	有効な ST ソースファイルの特性を表示するには、**[Properties]**を使用します。このウィンドウには、名前、変更日、および格納先が表示されます。

2.4 サンプルプログラムの作成

このセクションでは、短いプログラムを作成して、開始とテストを含めた関連手順を示します。テストについては プログラムのデバッグ (ページ 223)で説明しています。

ファンクション

*Blinker*プログラムは、ターゲットシステムの出力バイトにビットを設定し、このバイト内でビットを回転します。これにより、出力バイトの各ビットが連続して設定、リセットされます。バイトの最後のビットの後、最初のビットが再度設定されます。プログラムの結果は、ターゲットシステムの出力で確認できます。

2.4.1 必要条件

サンプルプログラムを作成するには以下が必要です。

- SIMOTION プロジェクト、および
- 出力をアドレス 62 に設定した、プロジェクト内の SIMOTION デバイス(たとえば、SIMOTION C230-2)

2.4.2 プロジェクトを開く、または作成する

プロジェクトには、ハードウェアと設定に関するすべての情報が含まれています。これには、ハードウェアの制御に使用するプログラムも含まれます。

実行手順

プロジェクトが存在しない場合は、以下の手順に従ってください。

1. メニューバーで[Project]を選択します。
2. [New]または[Open]を選択します。
3. 新規プロジェクトの名前を指定し、[OK]をクリックして確定します。

詳細については、オンラインヘルプを参照してください。

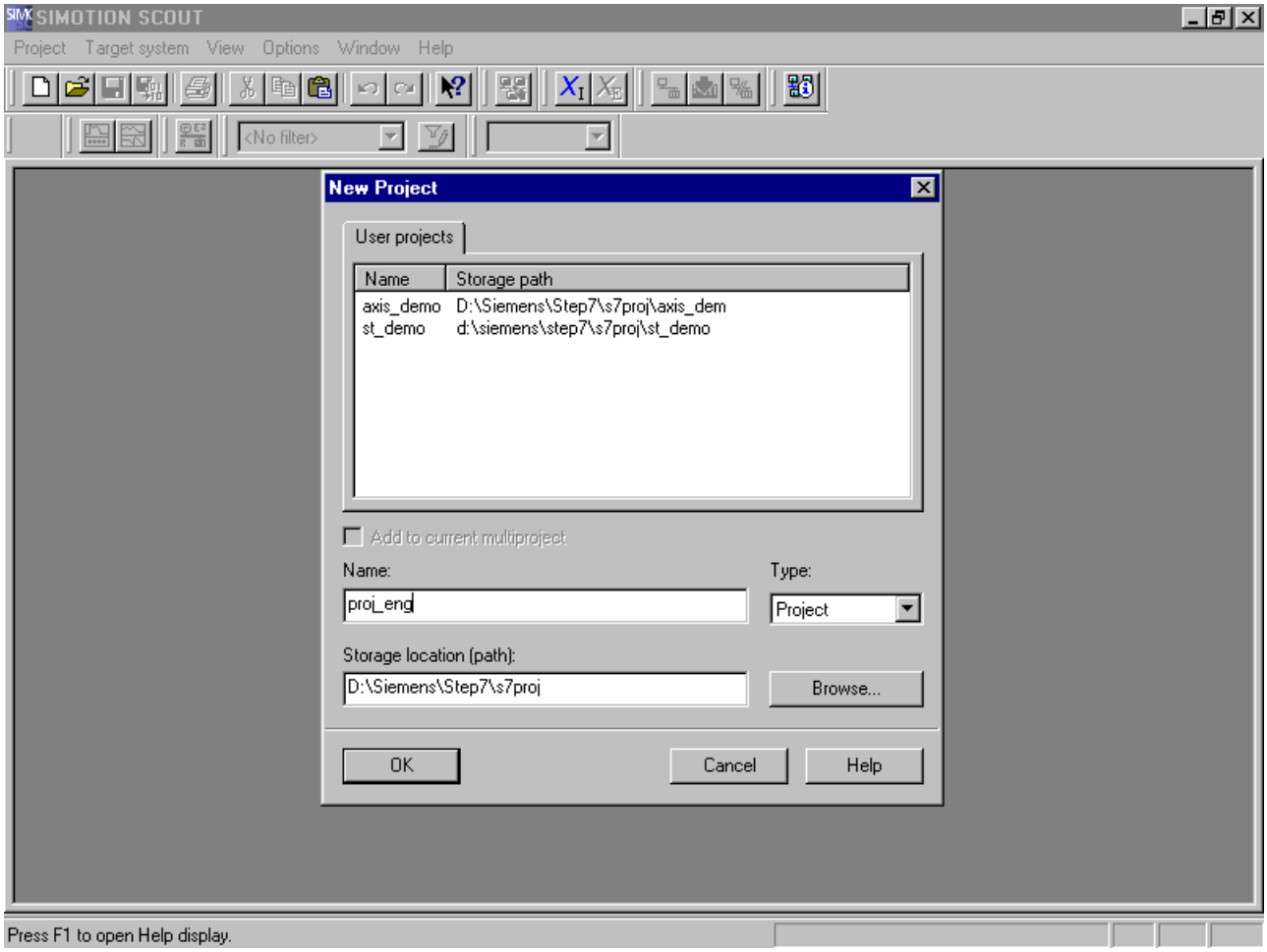


図 2-11　　新規プロジェクトの作成

2.4.3　ハードウェアを認識させる

実行手順

1. 新しい SIMOTION デバイス(たとえば、C240 V4.1)を作成および設定します。
2. HW Config で出力をアドレス 62 に設定します。

手順 1 と 2 の詳細については、オンラインヘルプを参照してください。

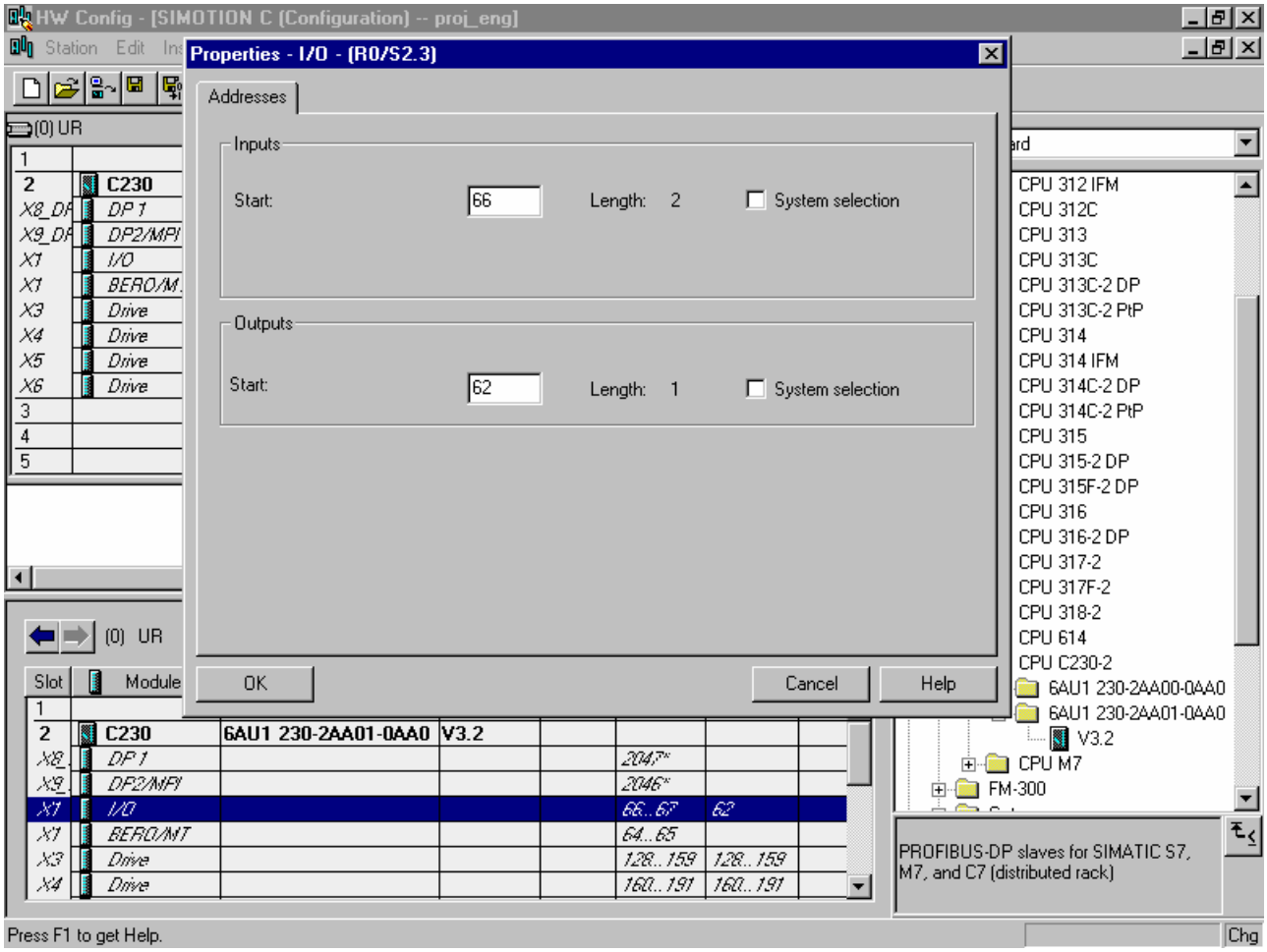


図 2-12　　HW Config での変更

2.4.4　ST エディタによるソーステキストの入力

実行手順

1. プロジェクトナビゲータで、SIMOTION デバイスのツリーを開きます(プログラムを実行する SIMOTION デバイスにプログラムが割り付けられます)。
2. **[PROGRAMS]**フォルダを選択し、**[Insert|Program|ST source file]**を選択します。
3. 最大 128 文字から成る ST ソースファイル名(たとえば、**ST_1**)を入力し(図を参照)、**[OK]**をクリックして入力を確定します。

   作業エリアに ST エディタが表示されます。ナビゲータに ST ソースファイル **ST_1** が挿入されます。

4. できれば行をインデントして、サンプルプログラムのソーステキスト (ページ 47)からソーステキストを入力します。インデントするには、TABキーを押します。

   STエディタの機能については、STエディタの操作 (ページ 25)で説明しています。STソースファイルの構造については、STソースファイルの構造 (ページ 69)および ソースファイルセクション (ページ 151)で詳しく説明しています。

5. できるだけ頻繁にコメントを使用します。コメントが 1 行のテキストに収まる場合、//文字の後にコメントを入力します。コメントが複数行にわたる場合は、文字のペア(*と*)の間にコメントを挿入します。

6. **[Project|Save]**を使用してプロジェクト一式を保存します。

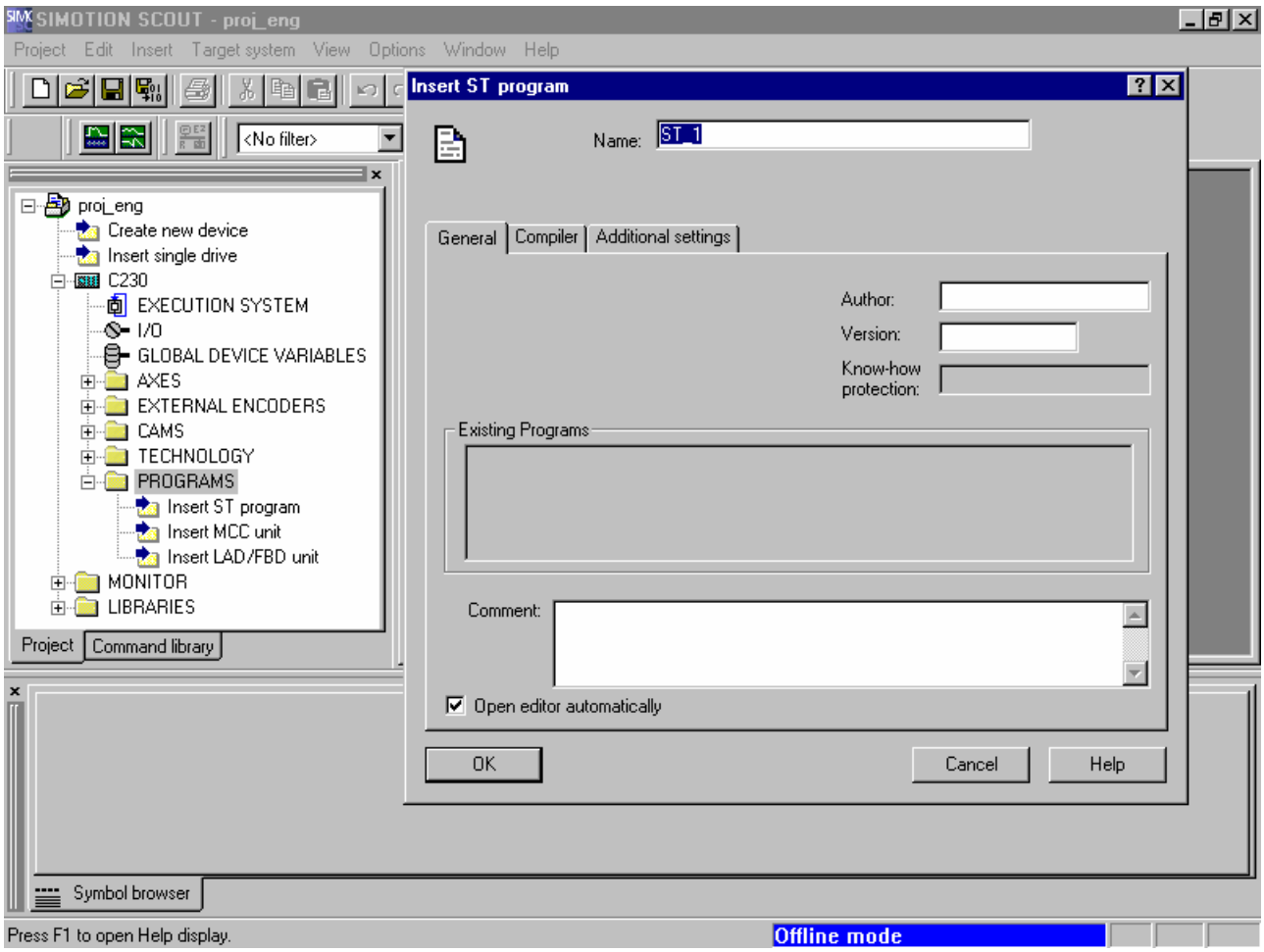


図 2-13　　ST ソースファイルの命名

下記も参照

STエディタとコンパイラの操作 (ページ 22)

2.4.4.1 エディタの機能

単純なテキスト入力に加えて、ST エディタには、ソーステキストの機能を文書化するための以下の高度/便利な機能が用意されています。

- 標準の Windows ユーザ機能(たとえば、Ctrl+Z による Undo や Ctrl+Y による Redo)
- 構文の色付け(言語要素によって異なる色を表示)
- ページ番号、ソースファイル名、印刷日を含む適切なレイアウトでのソースファイルの印刷
- ソースファイルのエクスポート/インポート
- ソースファイルのアーカイブ(プロジェクト経由)

機能の詳細については、ST エディタの操作 (ページ 25)および ST コンパイラの設定 (ページ 30)を参照してください。

下記も参照

ST エディタとコンパイラの操作 (ページ 22)

2.4.4.2 サンプルプログラムのソーステキスト

以下の表は、サンプルプログラムのソースコードを示します。実行可能コードを作成するには、同じようにソースコードを入力する必要があります。

表 2-10 サンプルプログラム Blinker

```
INTERFACE
    VAR_GLOBAL
        counterVar : INT  := 1;        // カウンタ変数
        outputVar  : BYTE := 1;        // 補助変数
    END_VAR

    PROGRAM Blinker;
END_INTERFACE

IMPLEMENTATION
    PROGRAM Blinker
        IF counterVar >= 500 THEN // 500 パスごとに
            %QB62 := outputVar; // 出力バイトを設定する
            outputVar := ROL(outputVar,1); (* バイト内のビットを
                                1 桁左に回転する *)
            counterVar := 1; // カウンタのリセット
        END_IF;
        counterVar := counterVar + 1; // カウンタの増分
    END_PROGRAM
END_IMPLEMENTATION
```

2.4.5 サンプルプログラムのコンパイル

プログラムを実行またはテストするには、実行可能マシンコードにプログラムをコンパイルしておく必要があります。このタスクは ST コンパイラが実行します。

2.4.5.1 コンパイラの起動

プログラムを実行またはテストするには、実行可能マシンコードにプログラムをコンパイルしておく必要があります。このタスクは ST コンパイラが実行します。

コンパイラは次のように起動します。

1. ST エディタを含むウィンドウをクリックして、**[ST source file]**メニューを表示します。このメニューはダイナミックなメニューで、ST エディタのウィンドウが有効な場合にのみ表示されます。
2. **[ST source file|Save and compile]**メニューコマンドを選択して、コンパイラを起動します。

2.4.5.2 エラーの修正

コンパイラによって ST ソースファイルの構文がチェックされます。詳細ビューの**[Compile/check output]**タブに、ソーステキストの成功したコンパイル、またはコンパイラエラーが表示されます。エラーの詳細には、 ST ソースファイルの名前、エラーが発生した行の番号、エラー番号、および エラーの説明が含まれます。

サンプルプログラムのエラーを修正するには、以下の手順に従ってください。

1. エラーメッセージをダブルクリックします。ST ソースファイルの関連する行にカーソルが置かれます。以下の図の例を参照してください。
2. 1 つ目のエラーのデバッグを開始します。
3. コンパイル操作をやり直します。
4. エラーが表示されなくなるまで(**0 エラー**)操作全体を繰り返します。

コンパイルに成功した場合、**Blinker** という名前のユーザプログラムを生成したことになります。これは、プロジェクトナビゲータの ST_1 の下に表示されます。

2.4.5.3 例: ST ソースファイルのコンパイル中のエラーメッセージ

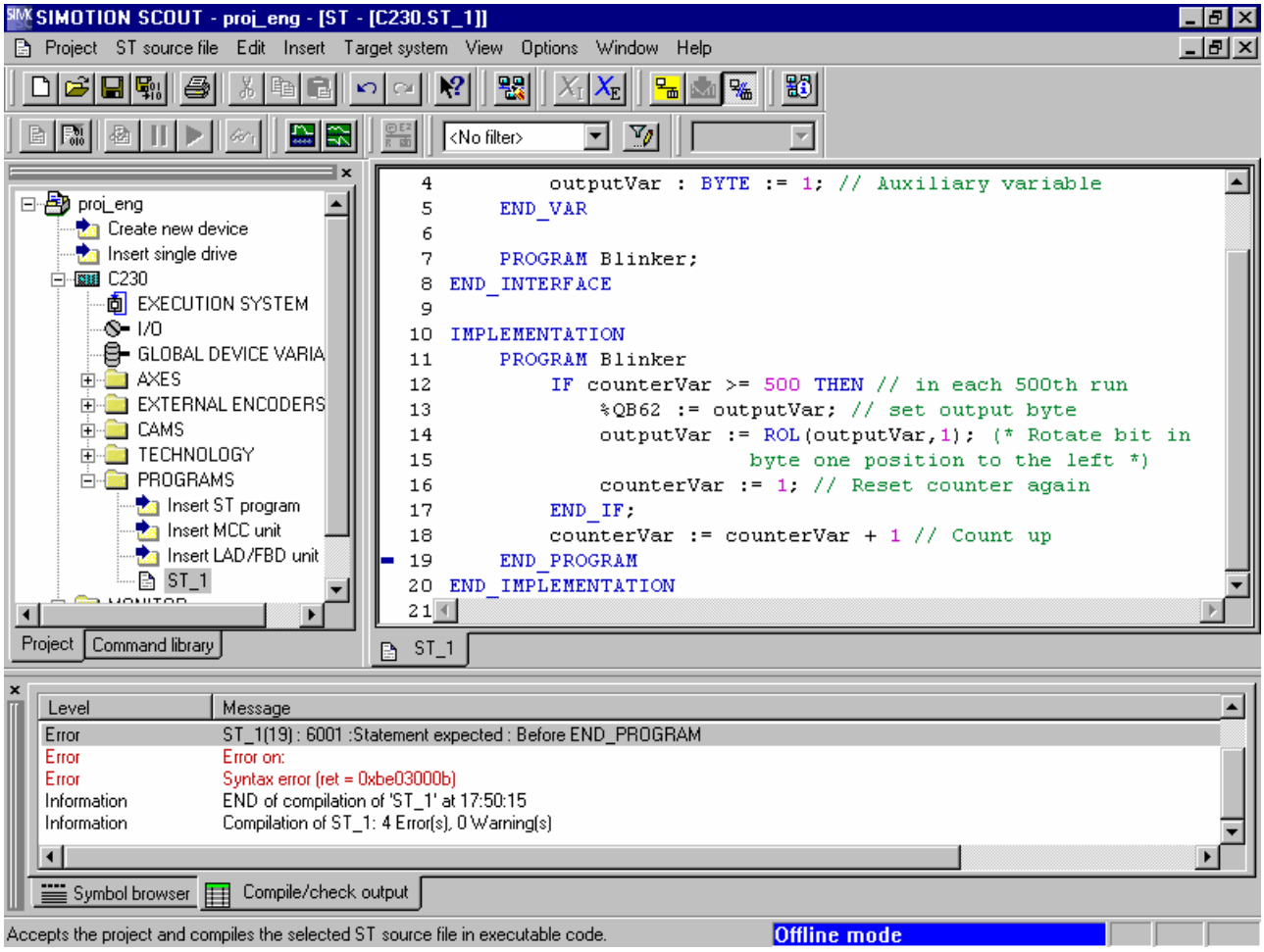


図 2-14 ST ソースファイルのコンパイル中のエラーメッセージ

図は、STソースファイルST_1 (サンプルプログラムのソーステキスト (ページ 47)を参照)のコンパイルの例を示します。ここでは、1 つの変更を加えています。18 行目のステートメントcounterVar := counterVar + 1 の末尾のセミコロンが欠けています。

コンパイラは、欠けているセミコロンの後からコンパイルを続行するため、19 行目までエラーを検出しません。

欠けているセミコロンを追加すると、ST ソースファイルはエラーなく実行されます。

すべてのコンパイラエラーメッセージの詳細なリストは、コンパイラエラーメッセージと修正方法 (ページ 305)を参照してください。

2.4.6 サンプルプログラムの実行

プログラムを実行するには、実行レベルまたはタスクにプログラムを割り付けておく必要があります。これを行うと、ターゲットシステムへの接続を確立し、ターゲットシステムにプログラムをダウンロードし、プログラムを起動することができます。

2.4.6.1 実行レベルへのサンプルプログラムの割り付け

実行レベルでは、プログラムを実行する順序を指定します。各実行レベルには、プログラムを割り付けることができる 1 つまたは複数のタスクが含まれています。

タスクへのプログラムの割り付けは、コンパイルの後、プログラムをターゲットシステムにロードする前にのみ実行できます。

1. プロジェクトナビゲータで**実行システム**要素をダブルクリックすると、実行システムとプログラム割り付けを含むウィンドウが作業エリアに表示されます。
2. プログラム割り付けのために **BackgroundTask** をクリックして選択します。

   ウィンドウの左側のプログラム割り付けに、タスクに割り付けることができるすべてのコンパイル済みプログラムが表示されます。
3. **[Programs]**リストで、サンプルプログラム **ST_1.blinker** をクリックします。次に、[>>] ボタンをクリックして、BackgroundTask にプログラムを割り付けます。

   以下の図に結果が表示されます。**[Programs used]**リストボックスにプログラム **ST_1.blinker** が表示されます。

実行システム、およびタスクへのプログラムの割り付けの詳細については、*『SIMOTION モーションコントロール、基本機能*、機能の記述』を参照してください。

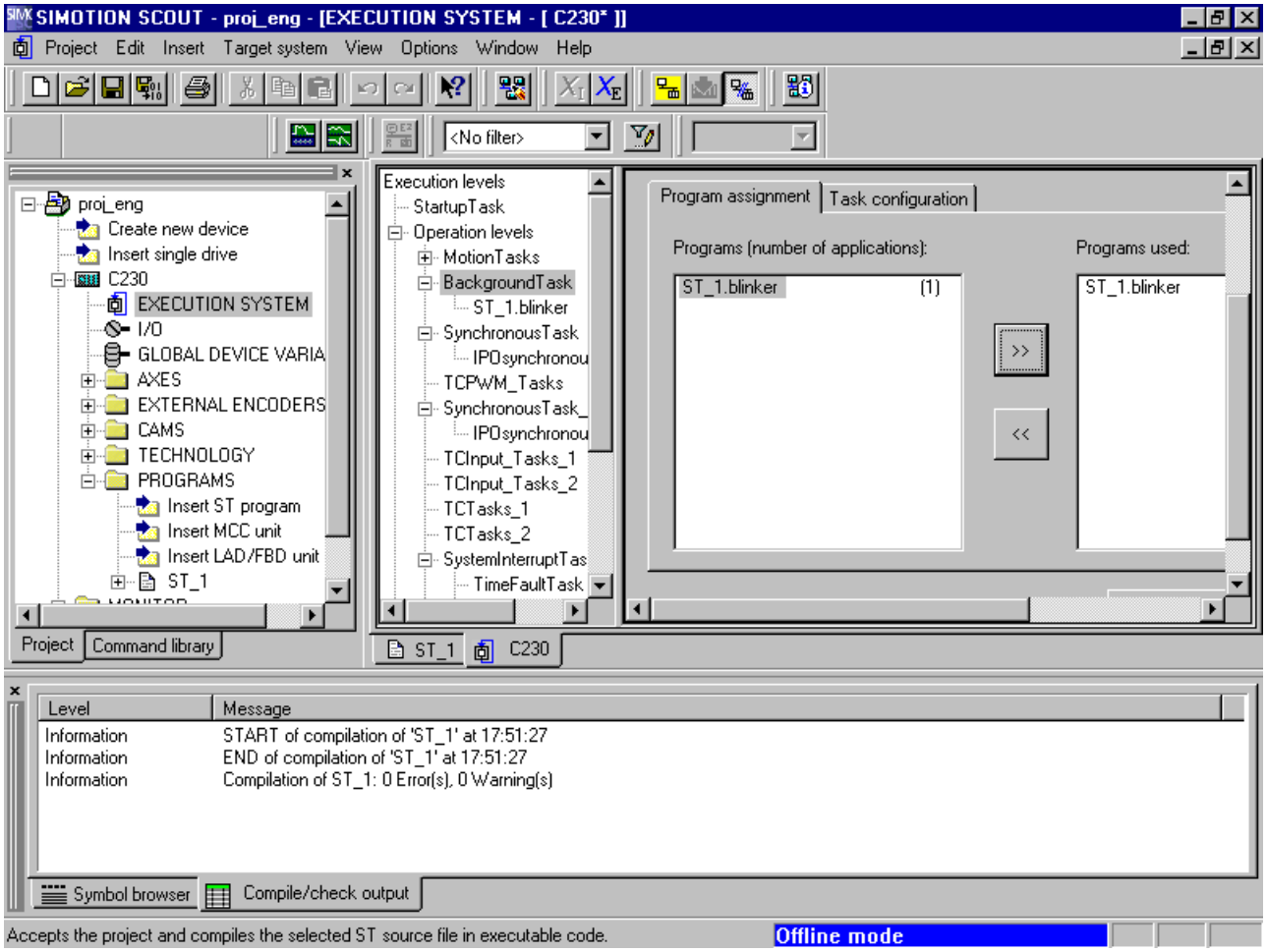


図 2-15　BackgroundTask へのサンプルプログラムの割り付け

2.4.6.2　ターゲットシステムへの接続の確立

ターゲットシステムへの接続をセットアップするには、PC インターフェースカードを設定し、ターゲットシステムに接続しておく必要があります。

ターゲットシステムに接続するには、以下の手順に従ってください。

1. **[Project|Connect to target system]**メニューコマンドを選択します。

   詳細ビューに**[Diagnostics overview]**タブが開きます。[Diagnostics overview]には、接続先デバイスの動作状態、メモリ割り付け、および CPU 使用率が表示されます。ターゲットシステムに接続されていることは画面の右下で確認できます。

注記

詳細については、『SIMOTION SCOUT 設定マニュアル』および SIMOTION SCOUT のオンラインヘルプを参照してください。

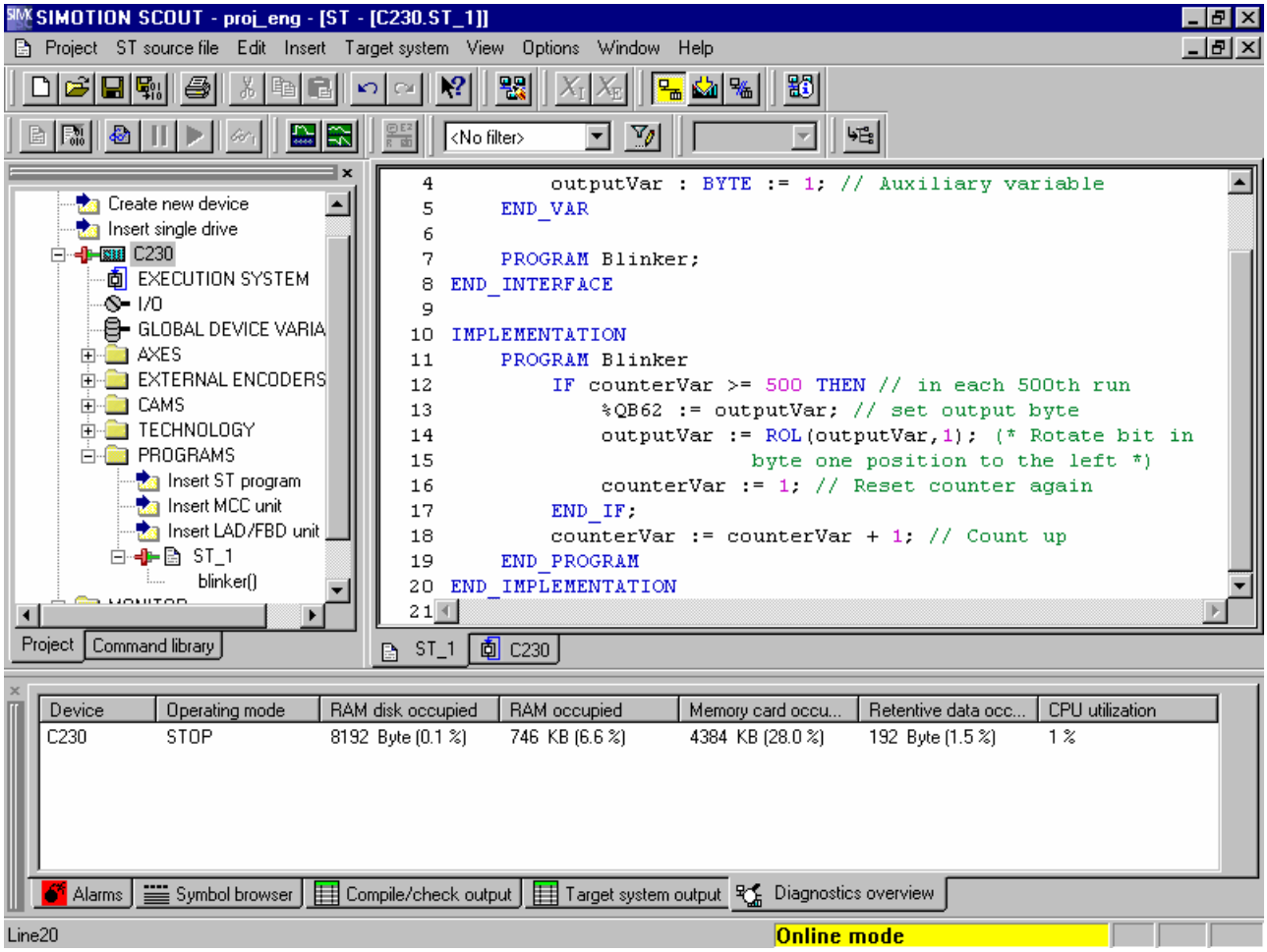


図 2-16　　ターゲットシステムへの接続の確立

2.4.6.3　ターゲットシステムへのサンプルプログラムのダウンロード

サンプルプログラムをターゲットシステムにダウンロードするには、以下の手順に従ってください。

1. ターゲットシステムを **STOP** に切り替えます。
2. **[Target system|Download|Project to target system]**メニューコマンドを選択します。

   詳細ビューの[Target system output]ウィンドウが開き、ダウンロードの結果が表示されます。さらに、ダウンロード操作が成功したことを確認するメッセージが表示されます。
3. ダウンロードメッセージを確認します。

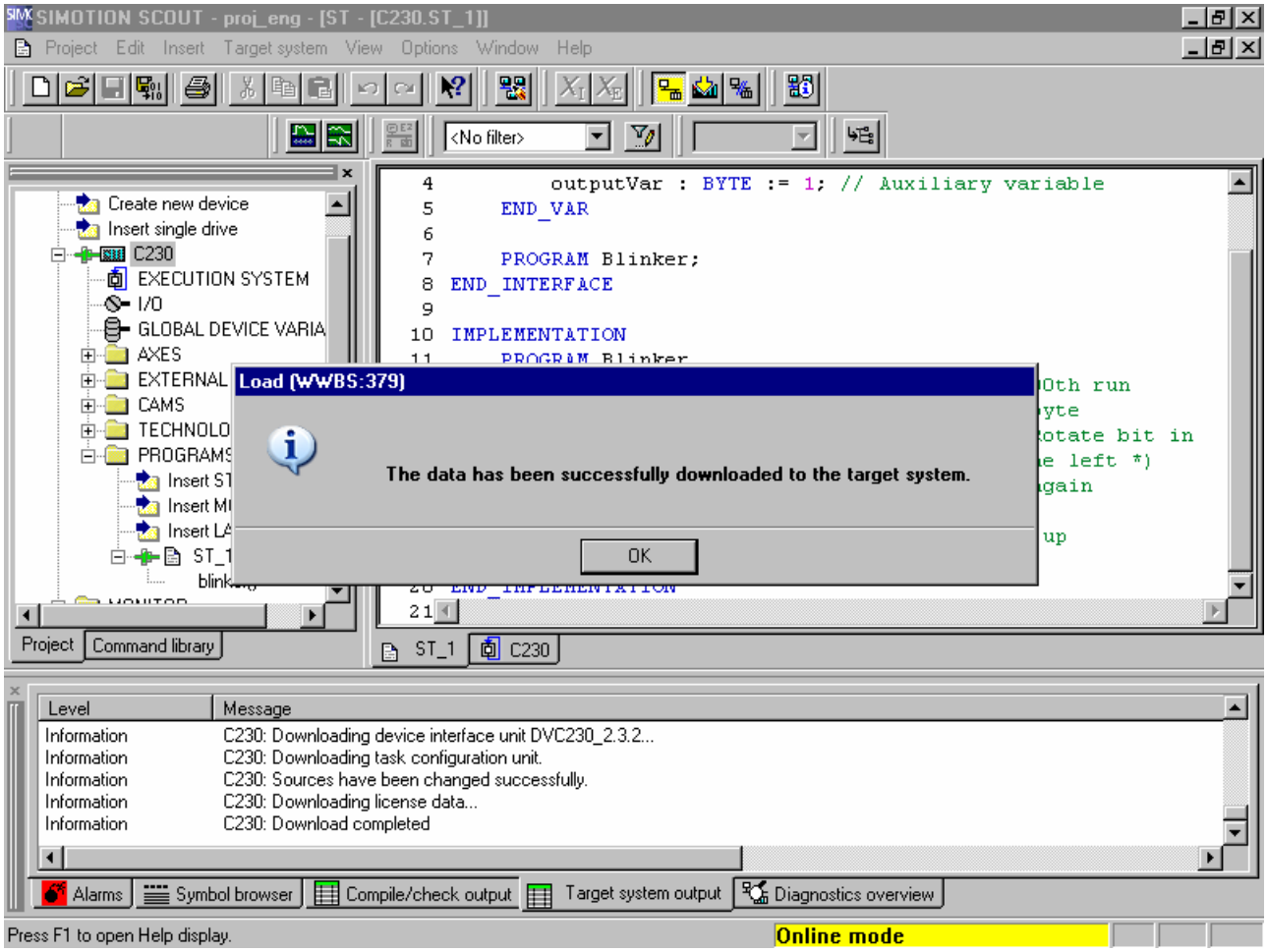


図 2-17　　ターゲットシステムへのサンプルプログラムのダウンロード

2.4.6.4　サンプルプログラムの起動とテスト

サンプルプログラムの起動

サンプルプログラムを起動するには、以下の手順に従ってください。

- ターゲットシステムを RUN に切り替えます(ハードウェアの説明を参照)。

ターゲットシステムの出力でランプが連続して点滅します。

サンプルプログラムのテスト

プログラムのデバッグ (ページ 223)を参照してください。

ST の基本原理

3

このセクションでは、ST で使用できる言語リソースとその使用方法を説明します。ファンクション、ファンクションブロック、およびタスク制御システムについては以下の章で説明しています。すべての構文ダイアグラムを含む完全な形式言語記述は、付録*「ルール」*を参照してください。

下記も参照

ルール (ページ 265)

3.1 言語記述リソース

本マニュアルの以下のセクションでは、言語記述の基礎として構文ダイアグラムを使用します。構文ダイアグラムを使用すると、ST の構文(すなわち文法)構造に関する理解が深まります。

3.1.1 構文ダイアグラム

構文ダイアグラムは、言語構造を図で表したものです。構造は一連のルールによって記述されています。既存のルールの上にルールを構築することができます。

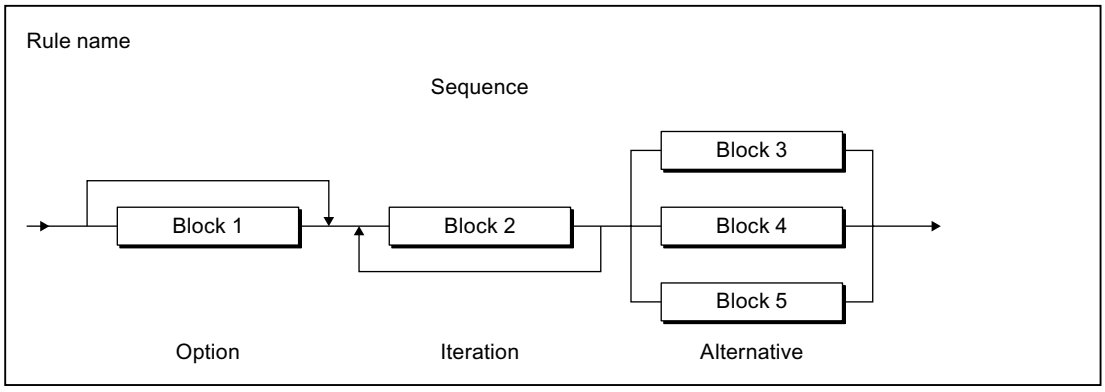


図 3-1 構文ダイアグラム

前の図の構文ダイアグラムは、左から右に読みます。以下のルール構造を守る必要があります。

- シーケンス: ブロックのシーケンス
- オプション: スキップ可能なステートメント
- 反復: 1 つまたは複数のステートメントの繰り返し
- 代替:ブランチ

3.1.2 構文ダイアグラムのブロック

ブロックは、1 つの基本要素、またはそれ自体複数のブロックから構成される 1 つの要素です。図は、ブロックを表すのに使用されるシンボルタイプを示します。

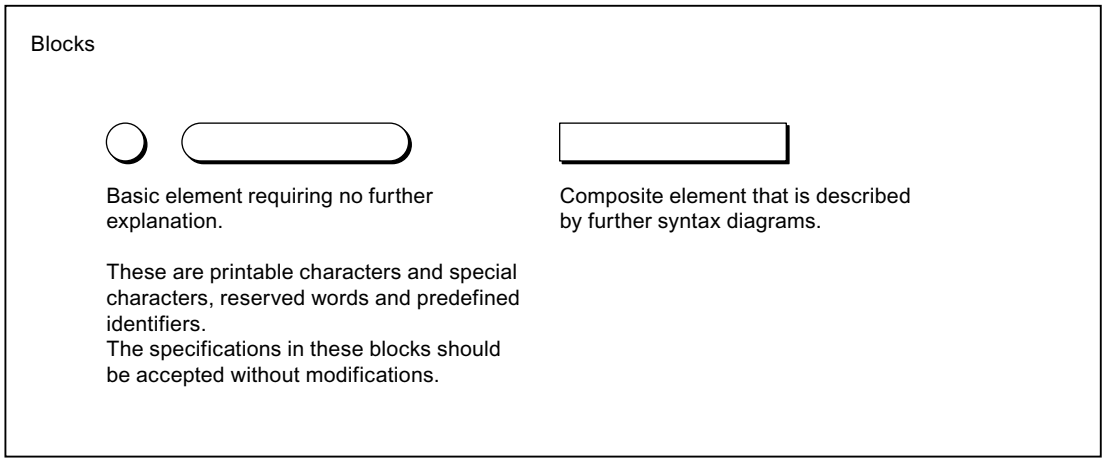


図 3-2 ブロック

ソーステキストを入力するとき、すなわち構文ダイアグラムのブロックまたは要素をソーステキストに変換するときは、書式付きルールと書式なしルールを守る必要があります(言語記述リソース (ページ 251)を参照)。

下記も参照

形式言語記述 (ページ 251)

3.1.3 ルールの意味(セマンティクス)

ルールで表すことができるのは言語の形式的な構造だけです。意味(すなわち、セマンティクス)は必ずしも明らかではありません。このため、意味が重要な場合は、ルールのそばに関連情報が書き込まれます。たとえば次のようになります。

- 同じ種類の要素で意味が異なる場合、追加の名前が付加されます。たとえば、すべての *10 進数字列*要素に関する *日付*ルールでは、追加(*年*、*月*、または *日*)が指定されます(リテラル (ページ 266)を参照)。名前は使用法を示します。
- ルールの隣に重要な制限事項が注記されます。たとえば、- (マイナス)に関する *整数*ルールでは、マイナスを出現させることができるのはSINT、INT、およびDINTデータタイプの 10 進数字列の前だけであることが注記されます(リテラル (ページ 266)を参照)。

下記も参照

形式言語記述 (ページ 251)

3.2 言語の基本要素

ST 言語の基本要素には、ST 文字セット、ST 文字セットから構築された予約識別子(たとえば、言語コマンド)、自己定義型識別子、および数字が含まれます。

ST 文字セットと予約識別子は、言語で記述され、別のルールによって記述されるものではないため、基本要素(端子)です。自己定義型識別子と数字は、他のルールによって記述されるため、端子ではありません。

構文ダイアグラムでは、端子は円または楕円形のシンボルによって表され、複合要素は長方形によって表されます(構文ダイアグラムのブロック (ページ 56)を参照)。以下に主要な端子を示します。完全な概要は、基本要素(端子) (ページ 253)を参照してください。

3.2.1 ST 文字セット

ST では、ASCII 文字セットの以下の**アルファベットと数字**が使用されます。

- A～Z、a～z のアルファベット
- 0～9 のアラビア数字

アルファベットと数字は最も一般的に使われる文字です。たとえば、識別子(STの識別子 (ページ 57)を参照)は、アルファベット、数字、下線の組み合わせで構成されます。下線は特殊文字の 1 つです。

特殊文字は、STでは決まった意味を持っています(形式言語記述 (ページ 251)、基本要素(端子) (ページ 253)を参照)。

3.2.2 ST の識別子

識別子は ST における名前です。この名前は、言語コマンドなど、システムによって定義することができます。ただし、たとえば定数、変数、ファンクションなどの場合、名前をユーザ定義することもできます。

3.2.2.1 識別子のルール

識別子は、アルファベット(A～Z、a～z)、数字(0～9)、または下線(_)を任意の順序で組み合わせたものです。最初の文字は、アルファベットまたは下線にする必要があります。アルファベットの大文字と小文字は区別されません(たとえば、Anna と AnNa はコンパイラにより同じものと見なされます)。

識別子は、形式的には以下の構文ダイアグラムで表すことができます。

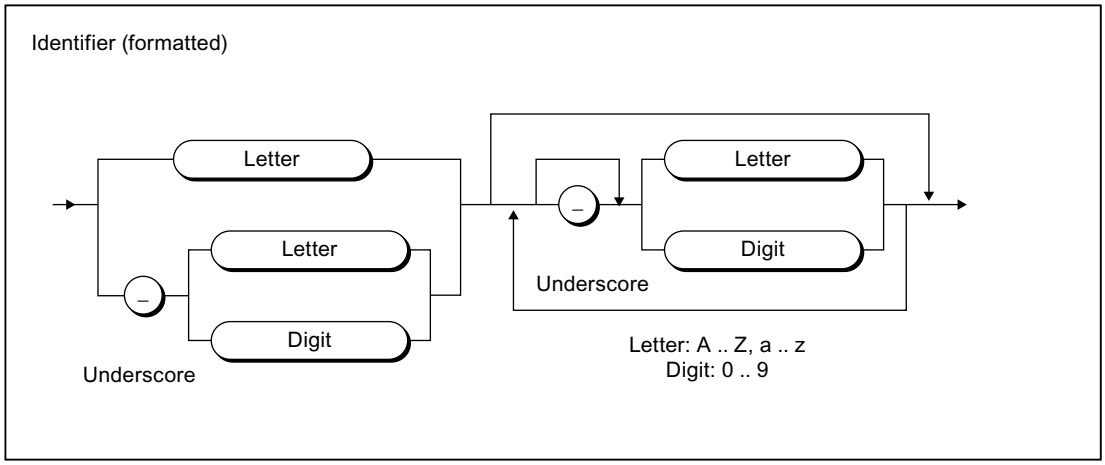


図 3-3　　構文:識別子

名前を割り付けるときは、プログラムが分かりやすくなる、意味のある一意の名前を選択するのが最善です。

図の構文ダイアグラムは、識別子の最初の文字がアルファベットまたは下線でなければならないことを示しています。下線の次はアルファベットまたは数字である必要があります。すなわち、複数の下線が連続することはできません。この後には、任意の数字、あるいは下線、アルファベット、または数字の並びを続けることができます。ここでもやはり、唯一の例外として、2 つの下線を続けて使用することはできません。

3.2.2.2　識別子の例

有効な識別子の例

以下の名前は有効な識別子です。

```
x              y12            _sum           temperature    R_CONTROLLER3
name           area           myFB           table
```

無効な識別子の例

以下の名前は有効な識別子では**ありません**。

無効な識別子	理由
4ter	最初の文字は、アルファベットまたは下線にする必要があります。
*#AB	特殊文字(下線以外)は使用できません。
RR__20	2 つの下線が連続することはできません。
S value	空白は特殊文字であるため、使用できません。
Array	ARRAY は形式的には有効な識別子ですが、これは予約識別子です。すなわち、定義済みとしてのみ使用できます。つまり、たとえば変数などに独自の目的でこの名前を使用することはできません。

使用できない識別子

以下の識別子は定義しないでください。

- **予約識別子**と同じ識別子

  詳細については、予約識別子 (ページ 59)を参照してください。

- **タスクの名前**と一致する識別子

  詳細については、*『SIMOTION 基本機能』*機能マニュアルを参照してください。

注記

できれば、_ (下線)、**struct**、**enum**、または**コマンド**で始まる識別子を定義するのは避けてください。

これらは有効な識別子ですが、使用すると、後で(追加の)テクノロジーパッケージをダウンロードするときにエラーが発生する可能性があります。基本システムおよびテクノロジーパッケージのコマンドワード、パラメータ、またはデータタイプはこれらの文字で始まっています。

3.2.3 予約識別子

予約識別子は定義済みとしてのみ使用できます。予約識別子の名前を使用して変数やデータタイプを宣言することはできません。

大文字と小文字の表記は区別されません。

すべての予約識別子のリストは、『SIMOTION 基本機能』機能マニュアルを参照してください。

- STプログラミング言語の保護識別子および予約識別子
  (保護識別子 (ページ 60)および その他の予約識別子 (ページ 64)も参照)
- 一般的な標準ファンクションおよびこのファンクションに定義されたデータタイプ
  (エラーソースおよびプログラムのデバッグ (ページ 223)も参照)
- 一般的なシステムファンクションブロック
- SIMOTION デバイスのシステムファンクション、システム変数、およびデータタイプ
  (SIMOTION デバイスのリストマニュアルも参照)
- テクノロジーオブジェクトのシステムファンクション、システム変数、およびデータタイプ
  (テクノロジーパッケージのリストマニュアルも参照)

3.2.3.1 保護識別子

ST 言語の保護識別子と予約識別子を表にリストしています。

すべての予約語の簡単な説明は、付録*「予約語」*、および付録*「ルール」*の構文ダイアグラムと例を参照してください。

表 3-1 ST プログラミング言語の保護識別子

A	
ABS	ANYTYPE_TO_LITTLEBYTEARRAY
ACOS	ARRAY
AND	AS
ANYOBJECT	ASIN
ANYOBJECT_TO_OBJECT	AT
ANYTYPE_TO_BIGBYTEARRAY	ATAN
B	
BIGBYTEARRAY_TO_ANYTYPE	BY
BOOL	BYTE
BOOL_TO_BYTE	BYTE_TO_BOOL
BOOL_TO_DWORD	BYTE_TO_DINT
BOOL_TO_WORD	BYTE_TO_DWORD
BOOL_VALUE_TO_DINT	BYTE_TO_INT
BOOL_VALUE_TO_INT	BYTE_TO_SINT
BOOL_VALUE_TO_LREAL	BYTE_TO_UDINT
BOOL_VALUE_TO_REAL	BYTE_TO_UINT
BOOL_VALUE_TO_SINT	BYTE_TO_USINT
BOOL_VALUE_TO_UDINT	BYTE_TO_WORD
BOOL_VALUE_TO_UINT	BYTE_VALUE_TO_LREAL
BOOL_VALUE_TO_USINT	BYTE_VALUE_TO_REAL
C	
CASE	CTD_UDINT
CONCAT	CTU
CONCAT_DATE_TOD	CTU_DINT
CONSTANT	CTU_UDINT
COS	CTUD
CTD	CTUD_DINT
CTD_DINT	CTUD_UDINT

D	
DATE	DO
DATE_AND_TIME	DT
DATE_AND_TIME_TO_DATE	DT_TO_DATE
DATE_AND_TIME_TO_TIME_OF_DAY	DT_TO_TOD
DELETE	DWORD
DINT	DWORD_TO_BOOL
DINT_TO_BYTE	DWORD_TO_BYTE
DINT_TO_DWORD	DWORD_TO_DINT
DINT_TO_INT	DWORD_TO_INT
DINT_TO_LREAL	DWORD_TO_REAL
DINT_TO_REAL	DWORD_TO_SINT
DINT_TO_SINT	DWORD_TO_UDINT
DINT_TO_UDINT	DWORD_TO_UINT
DINT_TO_UINT	DWORD_TO_USINT
DINT_TO_USINT	DWORD_TO_WORD
DINT_TO_WORD	DWORD_VALUE_TO_LREAL
DINT_VALUE_TO_BOOL	DWORD_VALUE_TO_REAL
E	
ELSE	END_REPEAT
ELSIF	END_STRUCT
END_CASE	END_TYPE
END_EXPRESSION	END_VAR
END_FOR	END_WAITFORCONDITION
END_FUNCTION	END_WHILE
END_FUNCTION_BLOCK	EXIT
END_IF	EXP
END_IMPLEMENTATION	EXPD
END_INTERFACE	EXPRESSION
END_LABEL	EXPT
END_PROGRAM	
F	
F_TRIG	FOR
FALSE	FUNCTION
FIND	FUNCTION_BLOCK
G	
GOTO	

I	
IF	INT_TO_SINT
IMPLEMENTATION	INT_TO_TIME
INSERT	INT_TO_USINT
INT	INT_TO_UDINT
INT_TO_BYTE	INT_TO_UINT
INT_TO_DWORD	INT_TO_WORD
INT_TO_DINT	INT_VALUE_TO_BOOL
INT_TO_LREAL	INTERFACE
INT_TO_REAL	
L	
LABEL	LREAL_TO_REAL
LEFT	LREAL_TO_SINT
LEN	LREAL_TO_UDINT
LIMIT	LREAL_TO_UINT
LITTLEBYTEARRAY_TO_ANYTYPE	LREAL_TO_USINT
LN	LREAL_VALUE_TO_BOOL
LOG	LREAL_VALUE_TO_BYTE
LREAL	LREAL_VALUE_TO_DWORD
LREAL_TO_DINT	LREAL_VALUE_TO_WORD
LREAL_TO_INT	
G	
MAX	MOD
MID	MUX
MIN	
N	
NOT	
O	
OF	OR
P	
PROGRAM	
R	
R_TRIG	REAL_VALUE_TO_BYTE
REAL	REAL_VALUE_TO_DWORD
REAL_TO_DINT	REAL_VALUE_TO_WORD
REAL_TO_DWORD	REPEAT
REAL_TO_INT	REPLACE
REAL_TO_LREAL	RETAIN
REAL_TO_SINT	RETURN
REAL_TO_TIME	RIGHT
REAL_TO_UDINT	ROL
REAL_TO_UINT	ROR
REAL_TO_USINT	RS
REAL_VALUE_TO_BOOL	RTC

S	
SEL	SINT_TO_UINT
SHL	SINT_TO_USINT
SHR	SINT_TO_WORD
SIN	SINT_VALUE_TO_BOOL
SINT	SQRT
SINT_TO_BYTE	SR
SINT_TO_DINT	STRING
SINT_TO_DWORD	STRUCT
SINT_TO_INT	StructAlarmId
SINT_TO_LREAL	STRUCTALARMID_TO_DINT
SINT_TO_REAL	StructTaskId
SINT_TO_UDINT	
T	
TAN	TOD
THEN	TOF
TIME	TON
TIME_OF_DAY	TP
TIME_TO_INT	TRUE
TIME_TO_REAL	TRUNC
TO	TYPE
U	
UDINT	UINT_TO_USINT
UDINT_TO_BYTE	UINT_TO_WORD
UDINT_TO_DINT	UINT_VALUE_TO_BOOL
UDINT_TO_DWORD	UNIT
UDINT_TO_INT	UNTIL
UDINT_TO_LREAL	USELIB
UDINT_TO_REAL	USEPACKAGE
UDINT_TO_SINT	USES
UDINT_TO_UINT	USINT
UDINT_TO_USINT	USINT_TO_BYTE
UDINT_TO_WORD	USINT_TO_DINT
UDINT_VALUE_TO_BOOL	USINT_TO_DWORD
UINT	USINT_TO_INT
UINT_TO_BYTE	USINT_TO_LREAL
UINT_TO_DINT	USINT_TO_REAL
UINT_TO_DWORD	USINT_TO_SINT
UINT_TO_INT	USINT_TO_UDINT
UINT_TO_LREAL	USINT_TO_UINT
UINT_TO_REAL	USINT_TO_WORD
UINT_TO_SINT	USINT_VALUE_TO_BOOL
UINT_TO_UDINT	

V	
VAR VAR_GLOBAL VAR_IN_OUT VAR_INPUT	VAR_OUTPUT VAR_TEMP VOID
WAITFORCONDITION WHILE WITH WORD WORD_TO_BOOL WORD_TO_BYTE WORD_TO_DINT WORD_TO_DWORD	WORD_TO_INT WORD_TO_SINT WORD_TO_UDINT WORD_TO_UINT WORD_TO_USINT WORD_VALUE_TO_LREAL WORD_VALUE_TO_REAL
X	
XOR	

下記も参照

予約語 (ページ 257)
ルール (ページ 265)

3.2.3.2 その他の予約識別子

表は、将来の拡張のために予約されているその他の予約識別子を示します。

表 3-2 ST 言語のその他の予約識別子

A	
ACTION ADD ADD_DT_TIME	ADD_TIME ADD_TOD_TIME
B	
BCD_TO_BYTE BCD_TO_DINT BCD_TO_DWORD BCD_TO_INT	BCD_TO_LWORD BCD_TO_SINT BCD_TO_WORD BYTE_TO_BCD
C	
CONFIGURATION CTD_LINT CTD_ULINT CTU_LINT	CTU_ULINT CTUD_LINT CTUD_ULINT
D	
DINT_TO_BCD DIV	DIVTIME DWORD_TO_BCD

E	
EN	END_STEP
END_ACTION	END_TRANSITION
END_CONFIGURATION	ENO
END_RESOURCE	EQ
F	
F_EDGE	FROM
G	
GE	GT
I	
INITIAL_STEP	INT_TO_BCD
L	
LE	LWORD
LINT	LWORD_TO_BCD
PM	
G	
MUL	MULTIME
N	
MS	
R	
R_EDGE	RESOURCE
S	
SEMA	SUB_DT_DT
SINT_TO_BCD	SUB_DT_TIME
STEP	SUB_TIME
SUB	SUB_TOD_TIME
SUB_DATE_DATE	SUB_TOD_TOD
T	
TRANSITION	
U	
ULINT	
V	
VAR_ACCESS	VAR_EXTERNAL
VAR_ALIAS	VAR_OBJECT
W	
WORD_TO_BCD	

3.2.4 数字およびブール値

ST ではさまざまな方法で数字を書くことができます。数字には、符号、小数点、または指数を含めることができます。すべての数字に以下のルールが適用されます。

- 数字の中にコンマや空白があってはいけません。
- 視覚的な区切りとして下線(_)を使用することができます。
- 数字の前にプラス(+)またはマイナス(–)を付けることができます。符号を使用しない場合、数字は正と想定されます。
- 数字が特定の最大値または最小値に違反することはできません。

3.2.4.1 整数

整数は小数点も指数も含みません。したがって、整数は、前に符号を付けることができる、数字の桁の並びです。

以下は有効な整数です。

```
0                1               +1              -1
743              -5280           60_000          -32_211_321
```

以下の整数は、示した理由のため**無効**です。

`123,456`	コンマは使用できません。
`36.`	整数に小数点を含めることはできません。
`10 20 30`	空白は使用できません。

ST では、整数はさまざまな記数法で表すことができます。これは、記数法のキーワード接頭語を挿入することにより実現します。

以下を使用します。

- 2 進法の場合、2#
- 8 進法の場合、8#
- 16 進法の場合、16#

10 進数 15 の有効な表現は次のようになります。

```
2#1111           8#17            16#F
```

3.2.4.2 浮動小数点数

浮動小数点数は小数点または指数(あるいは両方)を含むことができます。小数点は、数字の桁と桁の間にある必要があります。したがって、浮動小数点数が小数点で始まったり、小数点で終わったりすることはできません。

以下は有効な浮動小数点数です。

```
0.0             1.3             -0.2            827.602
0000.0          +0.000743       60_000.15       -315.0066
```

以下の浮動小数点数は**無効**です。

`1.`	小数点の前後に数字の桁がある必要があります。
`1,000.0`	コンマは使用できません。
`1.333.333`	小数点を 2 つ使用することはできません。

3.2.4.3 指数

小数点の位置を定義するには、指数を含めることができます。小数点が表示されない場合、小数点は桁の右側にあると想定されます。指数自体は正の整数または負の整数である必要があります。Base 10 はアルファベット E で表されます。

3 x 10^8 の大きさは、ST では以下の正確な浮動小数点数によって表すことができます。

```
3.0E+8      3.0E8       3e+8        3E8         0.3E+9
0.3e9       30.0E+7     30e7
```

以下の浮動小数点数は、**無効**です。

`3.E+8`	小数点の前後に数字の桁がある必要があります。
`8e2.3`	指数は整数である必要があります。
`.333e-3`	小数点の前後に数字の桁がある必要があります。
`30 E8`	空白は使用できません。

3.2.4.4 ブール値

ブール値はビット定数です。ブール値は、0 または 1 の値、あるいはキーワード FALSE または TRUE で表す必要があります。

例:

```
a := 1;         // a := TRUE と等価
b := FALSE; // b := 0 と等価
```

3.2.4.5 数字のデータタイプ

コンパイラにより、(式または値割り付け内の)値と用途に応じて、数字に適した基本データタイプが自動的に選択されます。

データタイプは直接指定することもできます。数字の前にデータタイプ(数値データタイプまたはビットデータタイプ)と文字"#"を配置します。

例:

```
INT#255                 INT#16#FF               INT#8#377
WORD#255                WORD#16#FF              WORD#8#377
REAL#255                REAL#16#FF              REAL#8#377
REAL#255.0              REAL#2.55E2             LREAL#255.0
```

注記

浮動小数点数は REAL および LREAL データタイプにのみ割り付けることができます。

3.2.5 文字列

文字列とは

文字列は、最初と最後をアポストロフィで囲まれた 0 個以上の文字の並びです。文字列では、各文字は 1 バイト(8 ビット)でコード化されます。

次のように文字を入力することができます。

- 印刷可能文字(ASCII コード$20～$7E、$80～$FF)として。ドル記号(ASCII コード$24)とアポストロフィ(ASCII コード$27)は除く。これらは文字列内で特殊な機能を持つため。
- 前にドル記号($)の付いた、関連する文字の 2 桁の 16 進 ASCII コードとして。
- 以下の表に準拠した 2 文字の組み合わせとして。

表 3-3　　文字列内の特殊文字を表す 2 文字の組み合わせ

文字の組み合わせ			意味
$$.			ドル記号$ ($24)
$'.			アポストロフィ' ($27)
$L	または	$l	ラインフィード LF ($0A)
$N	または	$n	キャリッジリターン + ラインフィード CR + LF ($0D$0A)
$P	または	$p	フォームフィード FF ($0C)
$R	または	$r	キャリッジリターン CR ($0D)
$T	または	$t	水平タブ(HT) ($09)

例:

''	空の文字列(長さ 0)
'A'	アルファベット A を含む長さ 1 の文字列
' '	空白を含む長さ 1 の文字列
'$''	アポストロフィを含む長さ 1 の文字列
'RL' '$0D$0A'	文字 CR と LF を含む長さ 2 の文字列の 2 つの等価表現
'$$1.00'	$1.00 を含む長さ 5 の文字列
'Text$R$L'	Text という語とその後に文字 CR と LF を含む長さ 6 の文字列
'ÄÖÜ' '$C4$D6$FC'	ドイツ語のウムラウト ÄÖü (ダイアレシスの付いた A、O、u)を含む長さ 3 の文字列の 2 つの等価表現

3.3　ST ソースファイルの構造

STソースは、基本的に、連続するテキストで構成されます。このテキストは、テキストを複数の論理セクションに分割することにより構造化することができます。これに関する詳細なルールは、ソースファイルセクション (ページ 151)を参照してください。

以下に概要を示します。

- **ST ソースファイル**は、プロジェクトで作成し、複数回出現させることができる論理ユニットです。論理ユニットは、しばしばユニットと呼ばれます。
- ST ソースファイルの論理セクションは**セクション**と呼ばれます(表を参照)。
- **ユーザプログラム**は、すべてのプログラムソース(たとえば、ST ソースファイル、MCC ユニットなど)を合わせたものです。

ST ソースファイルの各論理セクションには、特定のキーワードで示される開始と終了があります。

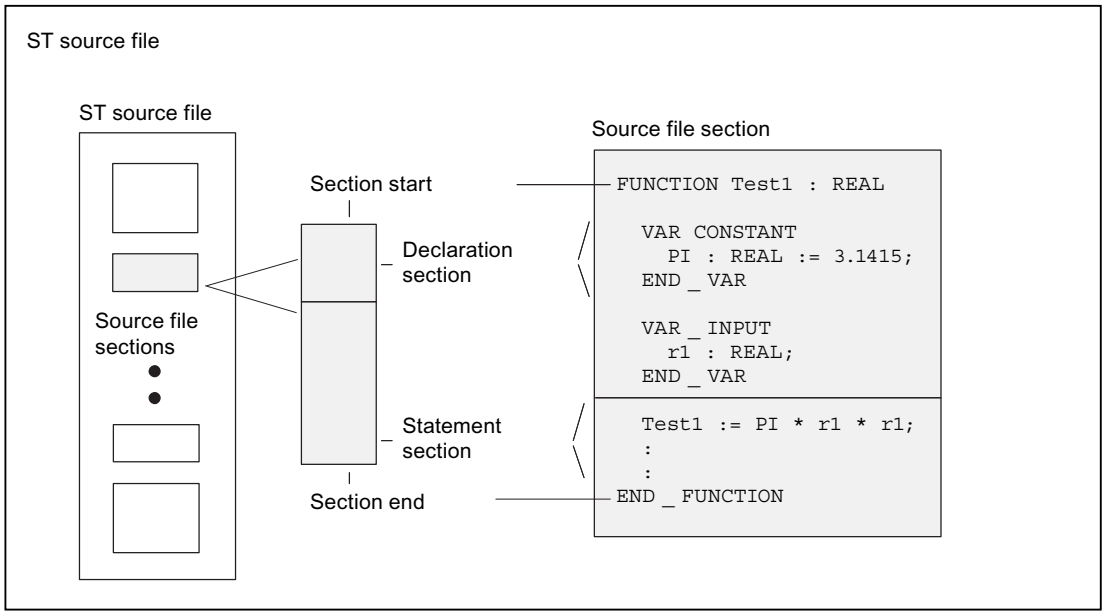


図 3-4　　ST ソースファイルの構造

すべてのファンクションを自身でプログラミングする必要はありません。SIMOTION のシステム部品を使用することもできます。このシステム部品は、システムファンクションやテクノロジーオブジェクトのファンクション(TO ファンクション)など、事前にプログラミングされたセクションです。

表 3-4　　ST ソースファイルの主なセクション

ソースファイルセクション	説明
ユニットステートメント(オプション)	ST の名前が含まれます。
インターフェースセクション	変数、タイプ、およびプログラムオーガニゼーションユニット(POU)をインポートおよびエクスポートするためのステートメントが含まれます。
実装セクション	ST ソースファイルの実行可能セクションが含まれます。
POU (プログラムオーガニゼーションユニット)	ST ソースファイルの単一の実行可能セクション(プログラム、ファンクション、ファンクションブロック)
宣言セクション	宣言(たとえば、変数やタイプの宣言)が含まれます。さらに、宣言セクション自体をインターフェースセクション、実装セクション、POU に含めることができます。
ステートメントセクション	POU の実行可能ステートメントが含まれます。

注記

オンラインヘルプでは、広範囲にわたり注釈の付いた*ユニット例のテンプレート*も入手でき、 新しい ST ソースファイルのテンプレートとして使用することができます。

ST エディタのヘルプを呼び出し、関連するリンクをクリックします。ST エディタの開いているウィンドウにテキストをコピーし、要件に従ってテンプレートを修正します。

*ユニット例のテンプレート*には、このテンプレートのコピーが含まれています。

3.3.1 ステートメント

プログラムオーガニゼーションユニット(POU – プログラム、ファンクション、ファンクションブロック)のステートメントセクションは、繰り返される単一のステートメントで構成されます。ステートメントセクションは POU の宣言セクションの後にあり、POU の最後で終了します。最初と最後に明示的なキーワードはありません。

ST には 3 つの基本的なタイプのステートメントがあります。

- 値割り付け

  変数への式の割り付け。変数宣言 (ページ 88)を参照してください。

- 制御ステートメント

  ステートメントの反復または分岐。制御ステートメント (ページ 114)を参照してください。

- サブルーチンの実行

  ファンクション(FC)およびファンクションブロック(FB)。ファンクション、ファンクションブロック、およびプログラム (ページ 131)を参照してください。

表 3-5　　ステートメントの例

```
...
// 値割り付け
 Status := 17;

// 制御ステートメント
 IF a = b THEN
     FOR c := 1 TO 10 DO
         b := b + c;
     END_FOR;
 END_IF;

// ファンクション呼び出し
 retVal := Test1(10.0);
...
```

3.3.2 説明

コメントは、文書化目的のため、また、ユーザがソースファイルセクションを理解するのを手助けするために使用します。

コンパイル後は、コメントはプログラムの実行に対して何の意味も持ちません。

2 つのタイプのコメントがあります。

- 行コメント
- ブロックコメント

行コメントの前には//を付けます。コンパイラは、行の最後まで続くその後のテキストをコメントとして処理します。

先頭に(*、最後に*)を付けると、複数行にわたるブロックコメントを入力することができます。

コメントを挿入するときは、以下の点に注意してください。

- コメントには完全な拡張 ASCII 文字セットを使用できます。
- 行コメント内では文字のペア(*と*)は無視されます。
- ブロックコメントのネストはできません。ただし、ブロックコメントに行コメントをネストすることはできます。
- コメントはどの位置でも挿入できますが、識別子の名前など、守るべきルールには挿入できません。このルールの詳細については、言語記述リソース (ページ 251)を参照してください。

表 3-6　　コメントの例

```
// これは 1 行コメントである。
a := 5;

// これは続けて複数回使用される
// 1 行コメントの例である。
b := 23;

(* 上記の例の方が複数行コメントとして
   編集が容易である。
 *)
c := 87;
```

3.4　データタイプ

データタイプは、プログラムソースで変数または定数の値を使用する方法を決定するために使用します。

ユーザは以下のデータタイプを使用できます。

- 基本データタイプ
- ユーザ定義データタイプ(UDT)
  - 単純派生
  - 配列
  - 列挙子
  - 構造体(Struct)
- テクノロジーオブジェクトデータタイプ
- システムデータタイプ

下記も参照

基本データタイプ (ページ 73)

テクノロジーオブジェクトのデータタイプの説明 (ページ 85)

システムデータタイプ (ページ 87)

3.4.1 基本データタイプ

3.4.1.1 基本データタイプ

基本データタイプは、これ以上小さいユニットに分割できないデータの構造を定義します。基本データタイプでは、固定長を持つメモリ領域と、ビットデータ、整数、浮動小数点数、時間、時刻、日付、文字列の状態を記述します。

すべての基本データタイプを以下の表に記載します。

表 3-7　基本データタイプのビット幅と値の範囲

タイプ	予約語	ビット幅	値の範囲
ビットデータタイプ このタイプのデータは、1 ビット、8 ビット、16 ビット、または 32 ビットのいずれかを使用します。このデータタイプの変数の初期化値は 0 です。			
ビット	BOOL	1	0、1 または FALSE、TRUE
バイト	BYTE	8	16#0 ~ 16#FF
ワード	WORD	16	16#0 ~ 16#FFFF
倍長ワード	DWORD	32	16#0 ~ 16#FFFF_FFFF
数値データタイプ これらのデータタイプは、数値を処理するために使用できます。このデータタイプの変数の初期化値は 0 (すべての整数) または 0.0 (すべての浮動小数点数)です。			
単精度整数	SINT	8	-128 ~ 127 (-2**7 ~ 2**7-1)
符号なし単精度整数	USINT	8	0 ~ 255 (0 ~ 2**8-1)
整数	INT	16	-32_768 ~ 32_767 (-2**15 ~ 2**15-1)
符号なし整数	UINT	16	0 ~ 65_535 (0 ~ 2**16-1)
倍長整数	DINT	32	-2_147_483_648 ~ 2_147_483_647 (-2**31 ~ 2**31-1)
符号なし倍長整数	UDINT	32	0 ~ 4_294_96_7295 (0 ~ 2**32-1)
浮動小数点数 (IEEE -754 に準拠)	REAL	32	-3.402_823_466E+38 ~ -1.175_494_351E−38、 0.0、 +1.175_494_351E−38 ~ +3.402_823_466E+38 精度: 24 ビット仮数、小数点第 6 位まで対応
倍長浮動小数点数 (IEEE-754 に準拠)	LREAL	64	−1.797_693_134_862_315_8E+308 ~ −2.225_073_858_507_201_4E308、 0.0、 +2.225_073_858_507_201_4E−308 ~ +1.797_693_134_862_315_8E+308 精度: 53 ビット仮数、小数点第 15 位まで対応

タイプ		予約語	ビット幅	値の範囲
時間データタイプ これらのデータタイプは、さまざまな日付と時刻の値を表すために使用します。				
	1 ミリ秒ずつ増加する時間	TIME	32	T#0d_0h_0m_0s_0ms ~ T#49d_17h_2m_47s_295ms 日、時、分、秒の値に最大 2 桁、ミリ秒の値に最大 3 桁 T#0d_0h_0m_0s_0ms で初期化
	1 日ずつ増加する日付	DATE	32	D#1992-01-01 ~ D#2200-12-31 うるう年は考慮され、年は 4 桁で表示、月と日はそれぞれ 2 桁で表示します D#0001-01-01 で初期化
	1 ミリ秒のステップでの時刻	TIME_OF_DAY (TOD)	32	TOD#0:0:0.0 ~ TOD#23:59:59.999 時、分、秒の値に最大 2 桁、ミリ秒の値に最大 3 桁 TOD#0:0:0.0 で初期化
	日付と時刻	DATE_AND_TIME (DT)	64	DT#1992-01-01-0:0:0.0 ~ DT#2200-12-31-23:59:59.999 DATE_AND_TIME はデータタイプ DATE と TIME から成ります。 DT#0001-01-01-0:0:0.0 で初期化
文字列データタイプ このタイプのデータは、各文字が指定されたバイト数でエンコードされた文字列を表します。 文字列の長さは、宣言時に定義することができます。長さは[ ]で囲んで示します。たとえば STRING[100]となります。デフォルト設定では 80 文字が指定されています。 割り当てられた(初期化された)文字数は、宣言された長さよりも短くなることがあります。				
	1 バイト文字による文字列	STRING	8	ASCII コード$00 ~ $FF のすべての文字を使用できます。 デフォルト' ' (空の文字列)

通知

変数を他のシステムにエクスポートする場合は、ターゲットシステムでの対応するデータタイプの値の範囲を考慮する必要があります。

下記も参照

基本データタイプの値の範囲限界 (ページ 75)
基本データタイプの値の範囲限界 (ページ 75)
一般的なデータタイプ (ページ 76)
基本システムデータタイプ (ページ 76)

3.4.1.2 基本データタイプの値の範囲限界

特定の基本データタイプの値の範囲限界は、定数として使用可能です。

表 3-8 基本データタイプの値の範囲限界に使用するシンボリック定数

シンボリック定数	データタイプ	値	16 進表記
SINT#MIN	SINT	-128	16#80
SINT#MAX	SINT	127	16#7F
INT#MIN	INT	-32768	16#8000
INT#MAX	INT	32767	16#7FFF
DINT#MIN	DINT	-2147483648	16#8000_0000
DINT#MAX	DINT	2147483647	16#7FFF_FFFF
USINT#MIN	USINT	0	16#00
USINT#MAX	USINT	255	16#FF
UINT#MIN	UINT	0	16#0000
UINT#MAX	UINT	65535	16#FFFF
UDINT#MIN	UDINT	0	16#0000_0000
UDINT#MAX	UDINT	4294967295	16#FFFF_FFFF
REAL#MIN	REAL	+1.175_494_351E−38	16#0080_0000
REAL#MAX	REAL	+3.402_823_466E+38	16#7F7F_FFFF
LREAL#MIN	LREAL	+2.225_073_858_507_201_4E−308	16#0010_0000_0000_0000
LREAL#MAX	LREAL	+1.797_693_134_862_315_8E+308	16#7FEF_FFFF_FFFF_FFFF
T#MIN TIME#MIN	TIME	T#0ms	16#0000_0000
T#MAX TIME#MAX	TIME	T#49d_17h_2m_47s_295ms	16#FFFF_FFFF
TOD#MIN TIME_OF_DAY#MIN	TOD	TOD#00:00:00.000	16#0000_0000
TOD#MAX TIME_OF_DAY#MAX	TOD	TOD#23:59:59.999	16#0526_5BFF

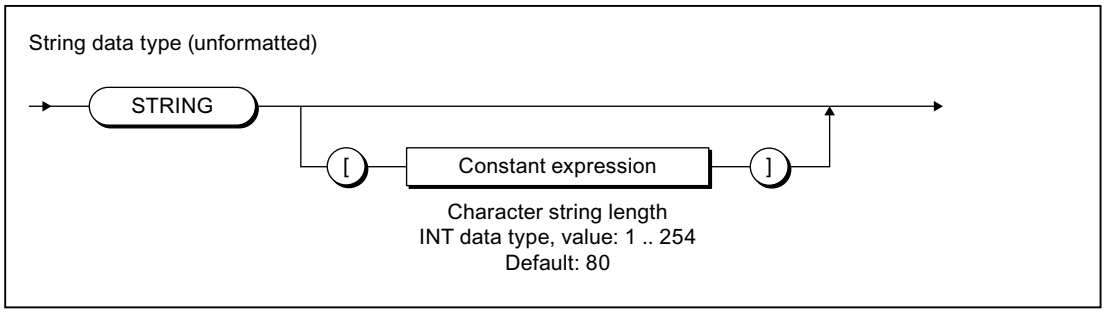


図 3-5 構文: STRING データタイプ

3.4.1.3 一般的なデータタイプ

一般的なデータタイプは、システムファンクションとシステムファンクションブロックの入力と出力のパラメータによく使用されます。サブルーチンは、この一般的なデータタイプに含まれている各データタイプの変数を使用して呼び出すことができます。

以下の表に、使用可能な一般的なデータタイプを記載します。

表 3-9　　一般的なデータタイプ

一般的なデータタイプ	含まれているデータタイプ
ANY_BIT	BOOL、BYTE、WORD、DWORD
ANY_INT	SINT、INT、DINT、USINT、UINT、UDINT
ANY_REAL	REAL、LREAL
ANY_NUM	ANY_INT、ANY_REAL
ANY_DATE	DATE、TIME_OF_DAY (TOD)、DATE_AND_TIME (DT)
ANY_ELEMENTARY	ANY_BIT、ANY_NUM、ANY_DATE、TIME、STRING
ANY	ANY_ELEMENTARY、ユーザ定義のデータタイプ(UDT)、システムデータタイプ、テクノロジーオブジェクトのデータタイプ

注記

一般的なデータタイプを、変数やタイプ宣言でタイプ識別子として使用することは**できません**。

一般的なデータタイプは、ユーザ定義のデータタイプ(UDT)が基本データタイプから直接取得されている場合に保持されます(SIMOTION ST プログラミング言語でのみ可能)。

3.4.1.4 基本システムデータタイプ

SIMOTION システムでは、以下の表で指定されているデータタイプは基本データタイプと同様に扱われます。これらのデータタイプは、多数のシステムファンクションで使用されます。

表 3-10　　基本システムデータタイプとその用途

識別子	ビット幅	用途
StructAlarmId	32	メッセージをプロジェクト単位で固有に識別するために使用する alarmId データタイプ。alarmId は、メッセージの生成に使用されます。 『*SIMOTION 基本機能*機能マニュアル』を参照してください。 STRUCTALARMID#NIL で初期化
StructTaskId	32	実行システムでタスクをプロジェクト単位で固有に識別するために使用する taskId データタイプ。 『*SIMOTION 基本機能*機能マニュアル』を参照してください。 STRUCTTASKID#NIL の初期化

表 3-11　　基本システムデータタイプの無効な値に使用するシンボリック定数

シンボリック定数	データタイプ	意味
STRUCTALARMID#NIL	StructAlarmId	無効な AlarmId
STRUCTTASKID#NIL	StructTaskId	無効な TaskId

3.4.2 ユーザ定義データタイプ

3.4.2.1 ユーザ定義データタイプ

ユーザ定義データタイプ(UDT)は、以下のセクションまたはユニットの後続のソースファイルセクションの宣言セクション(STソースファイルの構造 (ページ 69)および ソースファイルセクション (ページ 151)を参照)でコンストラクトTYPE/END_TYPEを使用して作成します。

- インターフェースセクション
- 実装セクション
- プログラムオーガニゼーションユニット(POU)

作成したデータタイプは宣言セクションで引き続き使用することができます。ソースファイルセクションによって、タイプ宣言の範囲が決まります。

下記も参照

ユーザ定義データタイプの構文(タイプ宣言) (ページ 77)

基本データタイプまたは派生データタイプの派生 (ページ 79)

ARRAY派生データタイプ (ページ 80)

列挙子派生データタイプ (ページ 82)

STRUCT (構造体)派生データタイプ (ページ 83)

3.4.2.2 ユーザ定義データタイプの構文(タイプ宣言)

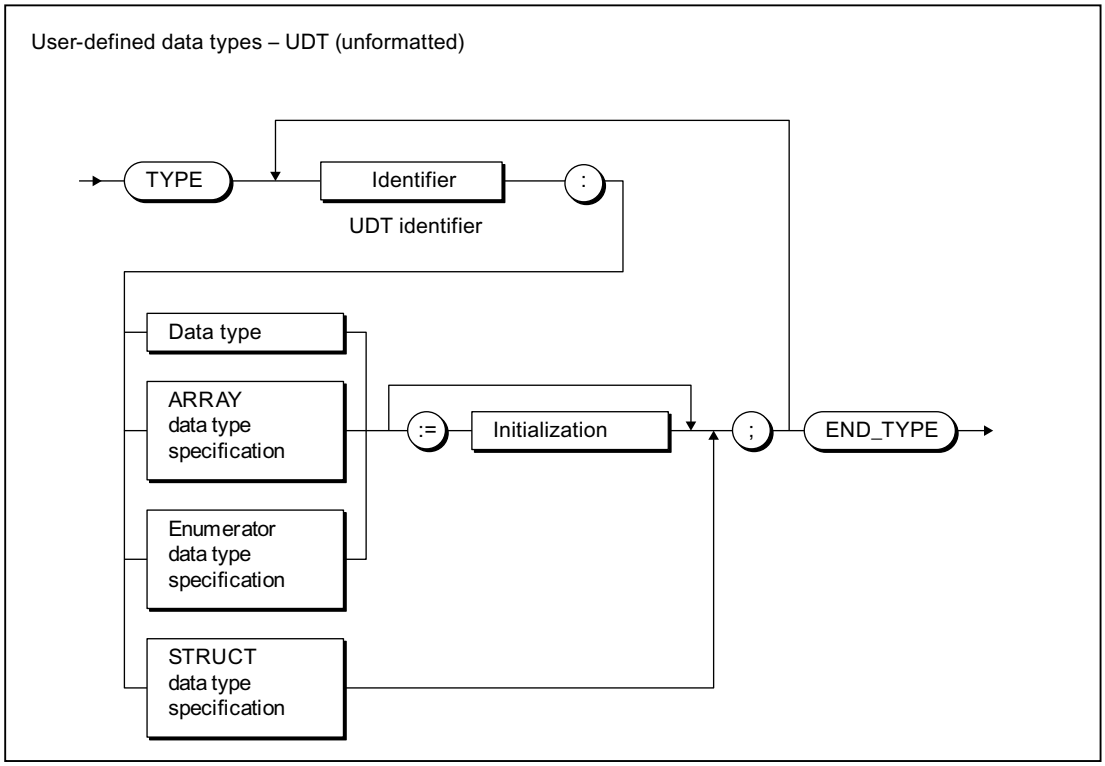


図 3-6 構文: ユーザ定義データタイプ

UDT の宣言は、キーワード TYPE で取り込まれます。

宣言するデータタイプごとに、この後に以下が続きます(図を参照)。

1. 名前:

   データタイプの名前は識別子のルールに準拠している必要があります。

2. データタイプ指定

   *データタイプ*という用語は以下で構成されます(基本データタイプまたは派生データタイプの派生 (ページ 79)を参照)。

   – 基本データタイプ
   – 以前に宣言した UDT
   – テクノロジーオブジェクトデータタイプ
   – システムデータタイプ

   以下のデータタイプ指定も可能です。

   – 配列データタイプ指定(ARRAY派生データタイプ (ページ 80)を参照)
   – 列挙子データタイプ指定(列挙子派生データタイプ (ページ 82)を参照)
   – STRUCTデータタイプ指定(STRUCT (構造体)派生データタイプ (ページ 83)を参照)

   括弧内に示した参照は、各データタイプ指定の詳細が記載されている以下のセクションを指します。

3. 初期化(オプション)

   データタイプの初期化値を指定することができます。その後でこのデータタイプの変数を宣言すると、変数に初期化値が割り付けられます。

   例外: STRUCT データタイプ指定の場合、個々のコンポーネントはデータタイプ指定内で初期化されます。

   変数またはデータタイプの初期化 (ページ 90)を参照してください。

UDT の完全な宣言は、キーワード END_TYPE で終了します。TYPE/END_TYPE コンストラクト内ではデータタイプをいくつでも作成できます。定義済みデータタイプを使用して変数またはパラメータを宣言することができます。

UDT は、図の構文が守られる限り、任意の方法でネストすることができます。たとえば、以前に定義した UDT やネストされた構造体をデータタイプ指定として使用することができます。タイプ宣言は、連続的にのみ使用でき、ネストされた構造体内では使用できません。

注記

すべての変数宣言の概要 (ページ 89)では、変数とパラメータを宣言する方法を学習できます。値割り付けの構文 (ページ 95)では、UDTを使用して値を割り付ける方法を学習できます。

以下に、UDT の個々のデータタイプ指定の説明と、その使用例を示します。

3.4.2.3 基本データタイプまたは派生データタイプの派生

データタイプの派生では、TYPE/END_TYPE コンストラクトで定義するデータタイプに基本データタイプまたはユーザ定義データタイプ(UDT)を割り付けます。

TYPE identifier : Elementary data type { := initialization } ; END_TYPE

TYPE identifier : User-defined data type { := initialization } ; END_TYPE

データタイプを宣言したら、派生データタイプの識別子の変数を定義することができます。これは、変数を*基本データタイプ*データタイプとして宣言するのと同等です。

表 3-12　　基本データタイプの派生の例

```
TYPE
    I1: INT;      // 基本データタイプ
    R1: REAL;     // 基本データタイプ
    R2: R1;       // 派生データタイプ(UDT)
END_TYPE
VAR
    // これらの変数は、
    // INT タイプの変数を使用できるところではどこでも使用できる。
    myI1 : I1;
    myI2 : INT;   // 派生データタイプなし!

    // これらの変数は、
    // REAL タイプの変数を使用できるところではどこでも使用できる。
    myR1 : R1;
    myR2 : R2;
END_VAR
myI1 := 1;
myI2 := 2;
myR1 := 2.22;
myR2 := 3.33;
```

3.4.2.4 ARRAY 派生データタイプ

ARRAY 派生データタイプは、同じデータタイプの定義された数のコンポーネントを TYPE/END_TYPE コンストラクトで結合します。以下の図の構文ダイアグラムは、このデータタイプを示しています。このデータタイプは、予約識別子 OF の後でより正確に指定します。

TYPE identifier : ARRAY data type specification { := initialization } ; END_TYPE

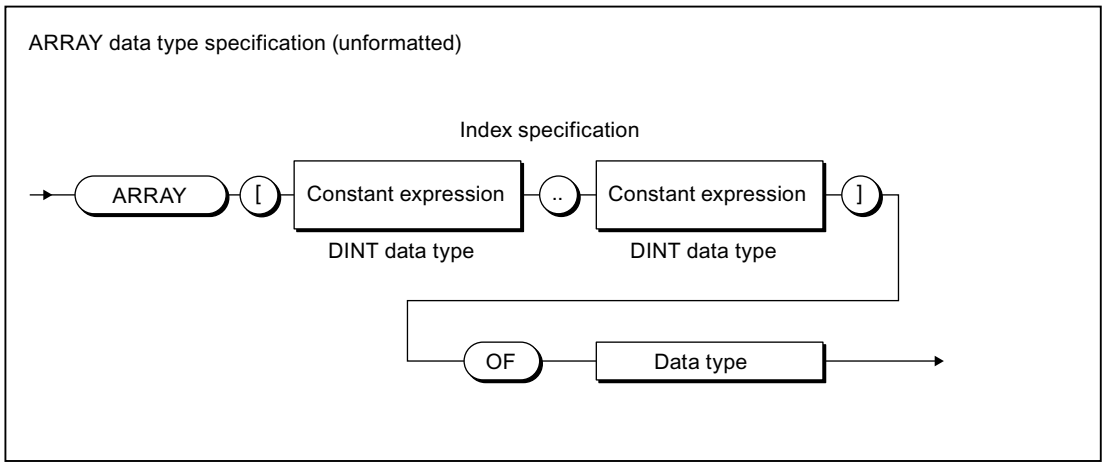


図 3-7　　構文: 配列データタイプ指定

インデックス指定には配列制限を記述します。

- 配列制限では、インデックスの最小値と最大値を指定します。これらの値は、定数または定数式を使用して指定できます。データタイプはDINTです(または、暗黙的にDINTに変換することができます。基本データタイプの変換 (ページ 125)を参照してください)。
- 配列制限は 2 つのピリオドで区切る必要があります。
- インデックス指定全体を角括弧で囲みます。
- インデックス自体にはDINTデータタイプの整数値を指定できます(または、暗黙的にDINTに変換することができます。基本データタイプの変換 (ページ 125)を参照してください)。

注記

ランタイム中に配列制限に違反すると、プログラムで処理エラーが発生します (*『SIMOTION 基本機能』*機能マニュアルを参照)。

配列コンポーネントのデータタイプはデータタイプ指定を使用して宣言します。この章で説明しているすべてのオプションをデータタイプとして(たとえば、ユーザ定義データタイプ(UDT)としても)使用することができます。

いくつかの異なる ARRAY タイプがあります。

- 一次元 ARRAY タイプは、昇順に整列されたデータ要素のリストです。
- 二次元 ARRAY タイプは、行と列から成るデータテーブルです。1 つ目の次元は行番号を参照し、2 つ目の次元は列番号を参照します。
- 高次元 ARRAY タイプは二次元 ARRAY タイプの拡張で、追加の次元を含んでいます。

表 3-13　　一次元配列の例

```
TYPE
    x : ARRAY [0.0.9] OF REAL;
    y : ARRAY[1..10] OF C1;
END_TYPE
```

二次元配列は行と列を含むテーブルに相当します。二次元配列または多次元配列は、マルチレベルのタイプ宣言によって作成できます。例を参照してください。

表 3-14　　多次元配列の例

```
TYPE
    a : ARRAY[1..3] OF INT;    // 1 次元配列(3 列):
    matrix1: ARRAY[1..4] OF a;      // 2 次元フィールド
                                    // (4 行、3 列)
    b: ARRAY[4.0.8] OF INT;    // 1 次元配列(5 列):
    matrix2: ARRAY[10..16] OF b;    // 2 次元フィールド
                                    // (7 行、5 列)
END_TYPE

VAR
    m: matrix1;       // 2 次元フィールドデータタイプの変数 m
    n: matrix2;       // 2 次元フィールドデータタイプの変数 m
END_VAR

m[4][3] := 9;          // Matrix1 の 4 行目、3 列目への書き込み
n[16][8] := 10;        // Matrix2 の 7 行目、5 列目への書き込み
```

例では、以下を定義できます。

1. 整数を含む一次元配列として、テーブル列 a[1] ~ a[3]
2. やはり配列として、テーブル行 matrix1[1] ~ matrix2[4]。ただし、データタイプ指定として、テーブル列を使用してちょうど作成した配列をとります。

   データタイプ指定で配列を指定すると、2 つ目の次元が作成されます。この方法でさらに次元を作成することができます。

今度は、テーブルに作成したデータタイプを使用して変数を宣言します。角括弧を使用して(この場合は、行と列を指定して)テーブルの各次元をアドレス指定します。

3.4.2.5 列挙子派生データタイプ

列挙子データタイプの場合、TYPE/END_TYPE コンストラクトで定義するデータタイプに、限られたセットの識別子または名前を割り付けます。

TYPE identifier : Enumerator data type specification { := initialization } ; END_TYPE

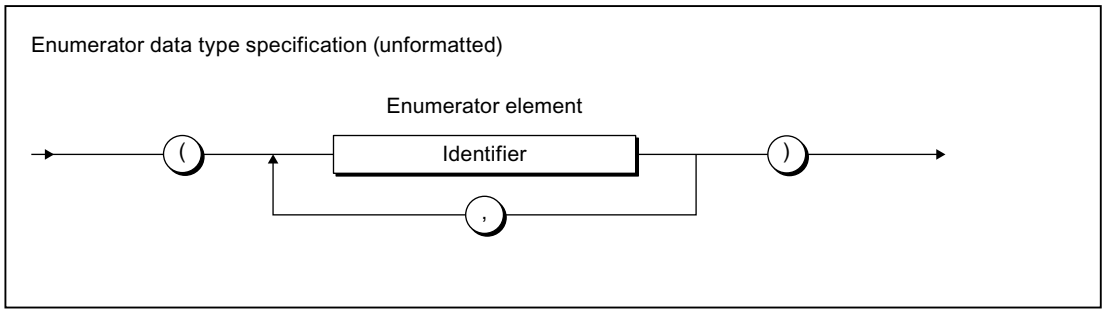


図 3-8　　構文: 列挙子データタイプ指定

*識別子*データタイプを宣言したら、列挙子データタイプの変数を定義することができます。ステートメントセクションでは、定義済み識別子のリストからの要素(列挙子要素)だけをこの変数に割り付けることができます。

データタイプは直接指定することもできます。列挙子要素の前に列挙子データタイプの識別子と"#"記号を配置します(表*「列挙子データタイプの例」*を参照)。

列挙データタイプの最初と最後の値は、それぞれ *enum_type*#MIN と *enum_type*#MAX を使用して取得することができます。*enum_type* は列挙データタイプの識別子です。

列挙要素の数値は、ENUM_TO_DINT 変換ファンクションを使用して取得することができます。

表 3-15　　列挙子データタイプの例

```
TYPE
    C1: (RED, GREEN, BLUE);
END_TYPE

VAR
    myC11, myC12, myC13 : C1;
END_VAR

myC11 := GREEN;
myC1l := C1#GREEN;
myC12 := C1#MIN;        // RED
myC13 := C1#MAX;        // BLUE
```

注記

列挙子データタイプはシステムデータタイプとしても存在します。

列挙子データタイプは構造体のコンポーネントとなることができます。つまり、列挙子データタイプは、ユーザ定義データ構造体のどの下位レベルでも使用されます。

3.4.2.6 STRUCT (構造体)派生データタイプ

STRUCT 派生データタイプ、すなわち構造体は、TYPE/END_TYPE コンストラクト内に一定数のコンポーネントの領域を含んでいます。これらのコンポーネントのデータタイプはさまざまです。

TYPE identifier : STRUCT data type specification; END_TYPE

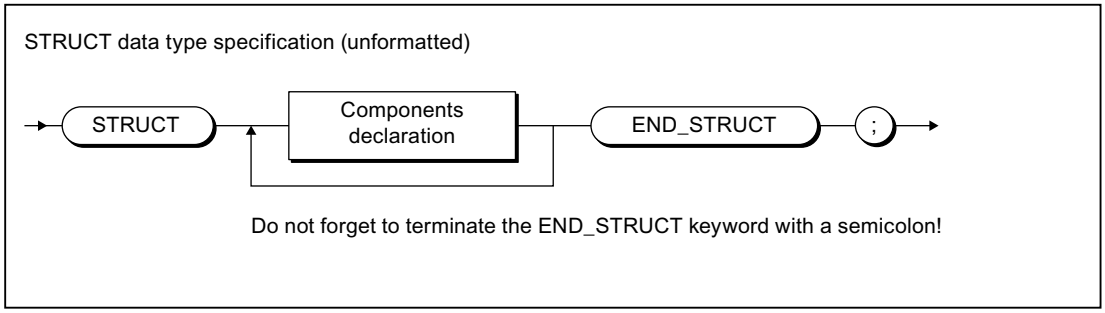


図 3-9　　構文: STRUCT データタイプ指定

以下の図にコンポーネント宣言の構文を示します。

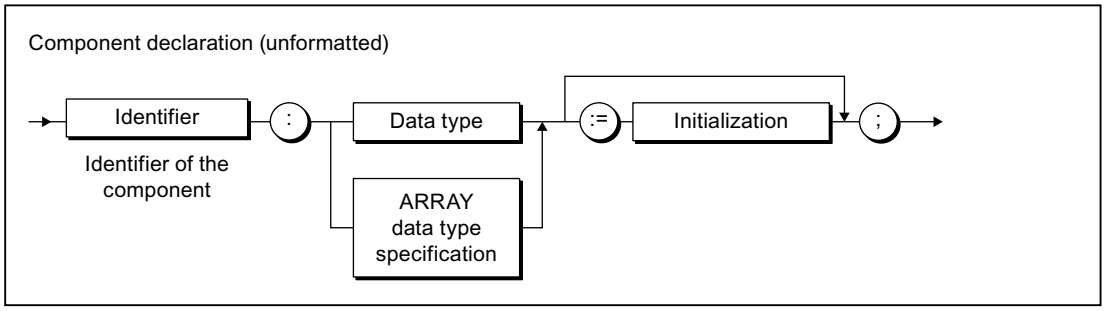


図 3-10　　構文: コンポーネント宣言

データタイプとして以下を使用できます。

- 基本データタイプ
- 以前に宣言した UDT
- システムデータタイプ
- テクノロジーオブジェクトデータタイプ
- 配列データタイプ指定

コンポーネントに初期化値を割り付けることもできます。変数またはデータタイプの初期化の場合のように処理してください(変数またはデータタイプの初期化 (ページ 90)を参照)。

注記

以下のデータ指定をコンポーネント宣言の内部で直接使用することはできません。

- STRUCT データタイプ指定
- 列挙子データタイプ指定

解決方法: 上記の指定を使用して UDT (ユーザ定義データタイプ)を事前に宣言し、これをコンポーネント宣言で使用します。

これにより、STRUCT データタイプをネストできるようになります。

STRUCT データタイプはシステムデータタイプとしても存在します。

この例は、UDT の定義方法と、変数宣言内でこのデータタイプを使用する方法を示します。

表 3-16　　STRUCT 派生データタイプの例

```
TYPE      // UDT の定義
    S1: STRUCT
        var1 : INT;
        var2 : WORD := 16#AFA1;
        var3 : BYTE := 16#FF;
        var4 : TIME := T#1d_1h_10m_22s_2ms;
    END_STRUCT;
END_TYPE

VAR
    myS1 : S1;
END_VAR

myS1.var1 := -4;
myS1.var4 := T#2d_2h_20m_33s_2ms;
```

3.4.3 テクノロジーオブジェクトデータタイプ

3.4.3.1 テクノロジーオブジェクトのデータタイプの説明

テクノロジーオブジェクト(TO)のデータタイプを使用して、変数を宣言することができます。以下の表は、個々のテクノロジーパッケージで使用可能なテクノロジーオブジェクトのデータタイプを記載しています。

たとえば、データタイプ *posAxis(位置決め軸タイプ)*を使用して変数を宣言し、位置決め軸の適切なインスタンスに割り当てることができます。このような変数はよく参照と呼ばれます。

表 3-17　テクノロジーオブジェクトのデータタイプ(TO データタイプ)

テクノロジーオブジェクト	データタイプ	テクノロジーパッケージの内容
ドライブ軸	driveAxis	CAM[12]、PATH、CAM_EXT
外部エンコーダ	externalEncoderType	CAM[12]、PATH、CAM_EXT
測定入力	MeasuringInputType (測定入力タイプ)	CAM[12]、PATH、CAM_EXT
出力カム	outputCamType	CAM[12]、PATH、CAM_EXT
カムトラック(V3.2 以降)	_camTrackType	CAM、PATH、CAM_EXT
位置決め軸	posAxis	CAM[13]、PATH、CAM_EXT
Following axis	followingAxis	CAM[14]、PATH、CAM_EXT
従動オブジェクト	followingObjectType (フォローイングオブジェクトタイプ)	CAM[14]、PATH、CAM_EXT
カム	camType	CAM、PATH、CAM_EXT
パス軸(V4.1 以降)	_pathAxis	PATH、CAM_EXT
パスオブジェクト(V4.1 以降)	_pathObjectType	PATH、CAM_EXT
固定ギア(V3.2 以降)	_fixedGearType	CAM_EXT
追加オブジェクト(V3.2 以降)	_additionObjectType	CAM_EXT
数式オブジェクト(V3.2 以降)	_formulaObjectType (数式オブジェクトタイプ)	CAM_EXT
センサ(V3.2 以降)	_sensorType	CAM_EXT
コントローラオブジェクト (V3.2 以降)	_controllerObjectType (コントローラオブジェクトタイプ)	CAM_EXT
温度チャンネル	TemperatureControllerType (温度コントローラタイプ)	TControl
すべてのテクノロジーオブジェクトを割り当て可能な一般的なデータタイプ	ANYOBJECT	

1) バージョン V3.1 以降には、BasicMC、位置とギアテクノロジーパッケージは収録されていません。
2) バージョン V3.0 までは、BasicMC、位置とギアテクノロジーパッケージも収録されていました。
3) バージョン V3.0 までは、位置とギアテクノロジーパッケージも収録されていました。
4) バージョン V3.0 までは、ギアテクノロジーパッケージも収録されていました。

テクノロジーオブジェクトの エレメント(コンフィグレーションデータとシステム変数)には構造体を経由してアクセスすることができます(『SIMOTION 基本機能機能マニュアル』を参照)。

表 3-18　テクノロジーオブジェクトデータタイプの無効な値に関するシンボリック定数

シンボリック定数	データタイプ	意味
TO#NIL	ANYOBJECT	無効なテクノロジーオブジェクト

下記も参照

軸のプロパティの継承 (ページ 86)

テクノロジーオブジェクトデータタイプの使用例 (ページ 86)

3.4.3.2 軸のプロパティの継承

軸の継承とは、TO driveAxis のすべてのデータタイプ、システム変数、ファンクションを TO positionAxis に完全に含めることです。同様に、位置決め軸は TO followingAxis に完全に含められ、フォローイング軸は TO pathAxis に完全に含められます。これには次のような効果があります。

- ファンクションまたはファンクションブロックが driveAxis データタイプの入力パラメータを予期する場合、呼び出し時に位置決め軸、フォローイング軸、またはパス軸を使用することもできます。
- ファンクションまたはファンクションブロックが posAxis データタイプの入力パラメータを予期する場合、呼び出し時にフォローイング軸またはパス軸を使用することもできます。

3.4.3.3 テクノロジーオブジェクトデータタイプの使用例

以下に、テクノロジーオブジェクトデータタイプの変数を任意で使用した例を示します(TO データタイプの変数を強制的に使用した例は、*『SIMOTION 基本機能』*機能マニュアルを参照してください)。2 つ目の例は、TO データタイプの変数を使用しない場合を示します。

軸を位置決めできるようプログラムの主要部分で軸を有効にするため、TO ファンクションを使用します。位置決め操作の後、構造体アクセスを使用して軸の現在の位置が記録されます。

最初の例では、TO データタイプの変数を使用して、TO データタイプの使用を示します。

表 3-19 テクノロジーオブジェクトデータタイプの使用例

```
VAR
    myAxis : posAxis;   // 軸の宣言変数
    myPos : LREAL;      // 軸の位置の変数
    retVal: DINT;      // TO ファンクションの
                // 戻り値の変数
END_VAR
myAxis := Axis1;       // 名前 Axis1 はプロジェクトナビゲータで軸を
                       // 設定したときに定義済みである。

// TO データタイプの変数によるファンクションの呼び出し:
retVal := _enableAxis(axis := myAxis, commandId := _getCommandId());

// 軸を位置決めする。
retVal := _pos(axis := myAxis,
               position := 100,
               commandId:= _getCommandId() );

// 構造体アクセスを使用して位置をスキャンする。
myPos := myAxis.positioningState.actualPosition;
```

2 つ目の例は、TO データタイプの変数を使用しません。

表 3-20　　テクノロジーオブジェクトの使用例

```
VAR
    myPos : LREAL;       // 軸の位置の変数
    retVal: DINT;  // TO ファンクションの戻り値の変数
END_VAR

// TO データタイプの変数によらないファンクションの呼び出し
// 名前 Axis1 はプロジェクトナビゲータで軸を
// 設定したときに定義済みである。
retVal := _enableAxis(axis := Axis1,
                      commandId:= _getCommandId() );

// 軸を位置決めする。
retVal := _pos(axis := Axis1
               position := 100,
               commandId:= _getCommandId() );

// 構造体アクセスを使用して位置をスキャンする。
myPos := Axis1.positioningState.actualPosition;
```

テクノロジーオブジェクトの設定の詳細については、SIMOTION モーションコントロール機能の説明を参照してください。

3.4.4　システムデータタイプ

事前の宣言を行うことなく使用できる多くのシステムデータタイプがあります。また、インポートした各テクノロジーパッケージには、システムデータタイプのライブラリが用意されています。

追加のシステムデータタイプ(主に列挙子および STRUCT データタイプ)は以下で使用されています。

- 一般的な標準ファンクションのパラメータ(*『SIMOTION 基本機能』*機能マニュアルを参照)
- 一般的な標準ファンクションのモジュールのパラメータ(*『SIMOTION 基本機能』*機能マニュアルを参照)
- SIMOTION デバイスのシステム変数(関連するパラメータマニュアルを参照)
- SIMOTION デバイスのシステムファンクションのパラメータ(関連するパラメータマニュアルを参照)
- テクノロジーオブジェクトのシステム変数および設定データ(関連するパラメータマニュアルを参照)
- テクノロジーオブジェクトのシステムファンクションのパラメータ(関連するパラメータマニュアルを参照)

3.5 変数宣言

変数は、STソースファイルで使用できる可変の内容を含むデータ項目を定義します。変数は、自由に選択できる識別子(たとえば、*myVar1*)とデータタイプ(たとえば、BOOL)で構成されます。予約識別子(予約識別子 (ページ 59)を参照)は識別子として使用しないでください。

3.5.1 変数宣言の構文

変数は、常にソースファイルセクションの宣言セクションで同じパターンに従って作成します。

1. 適切なキーワード(たとえば、VAR、VAR_GLOBALなど。すべての変数宣言の概要 (ページ 89)を参照)で宣言ブロックを開始します。
2. この後に、実際の変数宣言を続けます(図を参照)。これは必要な数だけ作成することができます。順番は任意です。
3. END_VAR で宣言ブロックを終了します。
4. さらに宣言ブロックを作成することができます(やはり同じキーワードを使用します)。

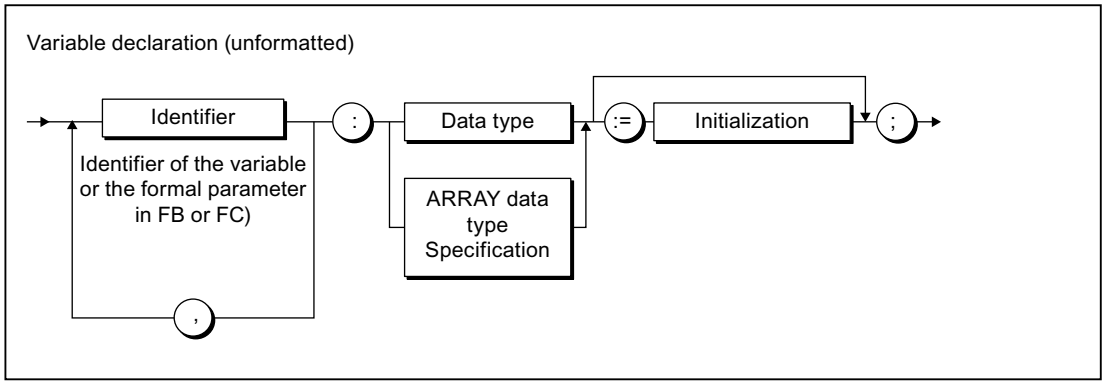


図 3-11 構文: 変数宣言

以下に注意してください。

- 変数名は識別子である必要があります。すなわち、変数名はアルファベット、数字、または下線のみを含むことができ、特殊文字を含むことはできません。
- データタイプとして以下を使用できます。
  - 基本データタイプ
  - UDT (ユーザ定義データタイプ)
  - システムデータタイプ
  - テクノロジーオブジェクトデータタイプ
  - 配列データタイプ指定
  - ファンクションブロックの指定(インスタンスの宣言。ファンクションおよびファンクションブロックの呼び出し (ページ 137)を参照)
- 宣言ステートメントで変数に初期値を割り付けることができます。これを初期化といいます(変数またはデータタイプの初期化 (ページ 90)を参照)。

次の場合は示したパターンから外れます。

- 定数宣言の場合(定数は値を使用して初期化する**必要があります**。定数 (ページ 94)を参照)
- プロセスイメージアクセスの場合(すべての変数宣言の概要 (ページ 89)を参照)
  - 絶対プロセスイメージアクセスの場合、変数宣言は必要ありません。
  - シンボリックプロセスイメージアクセスの場合、初期化はできません。

表 3-21　　変数宣言の例

```
VAR CONSTANT
    PI : REAL := 3.1415;
END_VAR

VAR
    // 変数の宣言 ...
    var1 : REAL;
    // ... または、同じタイプの複数の変数がある場合:
    var2, var3, var4 : INT;
    // 1 次元配列の宣言:
    a1 : ARRAY[1..100] OF REAL;
    // 文字列の宣言:
    str1 : STRING[40];
END_VAR
```

3.5.2　すべての変数宣言の概要

変数およびパラメータの宣言では、変数の名前、データタイプ、および初期値を指定します。これらの宣言は、必ず以下のソースファイルセクションの宣言セクションで実行します。

- インターフェースセクション
- 実装セクション
- POU (プログラム、ファンクション、ファンクションブロック、式)

ソースファイルセクションでは、宣言できる変数(表を参照)とその範囲も決定します。

ソースファイルセクションの関連情報は、STソースファイルの構造 (ページ 69)および ソースファイルセクション (ページ 151)を参照してください。

表 3-22　　宣言ブロックのキーワード

キーワード	意味	以下の宣言セクションでの宣言
VAR	テンポラリ変数またはスタティック変数の宣言 変数モデル (ページ 167)を参照	任意の POU
VAR_GLOBAL	ユニット変数の宣言 変数モデル (ページ 167)を参照	インターフェースセクション 実装セクション
VAR_IN_OUT	入/出力パラメータの変数宣言。POU はこの変数に直接アクセスし(参照を使用)、変数を直ちに変更することができます。 「変数の定義」 (ページ 132)、「ファンクションブロックの定義」 (ページ 133)を参照	ファンクション ファンクションブロック (Function block) 式
VAR_INPUT	入力パラメータの変数宣言。値は外部から提供され、POU 内では変更できません。 ファンクションの定義 (ページ 132)、ファンクションブロックの定義 (ページ 133)を参照	ファンクション ファンクションブロック (Function block) 式
VAR_OUTPUT	出力パラメータの変数宣言。値はファンクションブロックから送信されます。 「ファンクションの定義」 (ページ 132)、「ファンクションブロックの定義」 (ページ 133)を参照	ファンクションブロック (Function block)
VAR_TEMP	テンポラリ変数の宣言 変数モデル (ページ 167)を参照	プログラム ファンクションブロック (Function block)
RETAIN	保持型変数の宣言 変数モデル (ページ 167)を参照	インターフェースおよび実装セクションの VAR_GLOBAL の補足としてのみ
CONSTANT	定数の宣言 定数 (ページ 94)を参照	以下の補足としてのみ • FB、FC、またはプログラムの VAR • インターフェースまたは実装セクションの VAR_GLOBAL

3.5.3　変数またはデータタイプの初期化

宣言内での変数またはデータタイプに対する初期値の割り付けはオプションです(*「構文: 変数宣言」*または*「構文: ユーザ定義データタイプ」*の図を参照)。

- 変数宣言で初期化を指定していない場合、データタイプ宣言で変数に指定した初期化値がコンパイラにより自動的に割り付けられます。
- データタイプ宣言で初期化を指定していない場合、コンパイラにより変数またはデータタイプにゼロの値が割り付けられます。例外:
  - 時刻データタイプの場合: 初期化値
  - 列挙データタイプの場合: 1. 列挙の値

データタイプ指定の後に値を割り付けることにより(:=)、初期値を使用して変数またはユーザ定義データタイプを事前割り付けします(*「構文: 変数の初期化」*の図を参照)。

- *「構文: 定数式」*の図に従って定数式に基本データタイプ(または基本データタイプから派生したデータタイプ)を割り付けます。
- *「構文: フィールド初期化リスト」*の図に従ってフィールド(ARRAY)にフィールド初期化リストを割り付けます。
- *「構文: 構造体初期化リスト」*の図に従って構造体(STRUCT)の個々のコンポーネントに構造体初期化リストを割り付けます。
- 列挙子データタイプに列挙子要素を割り付けます。

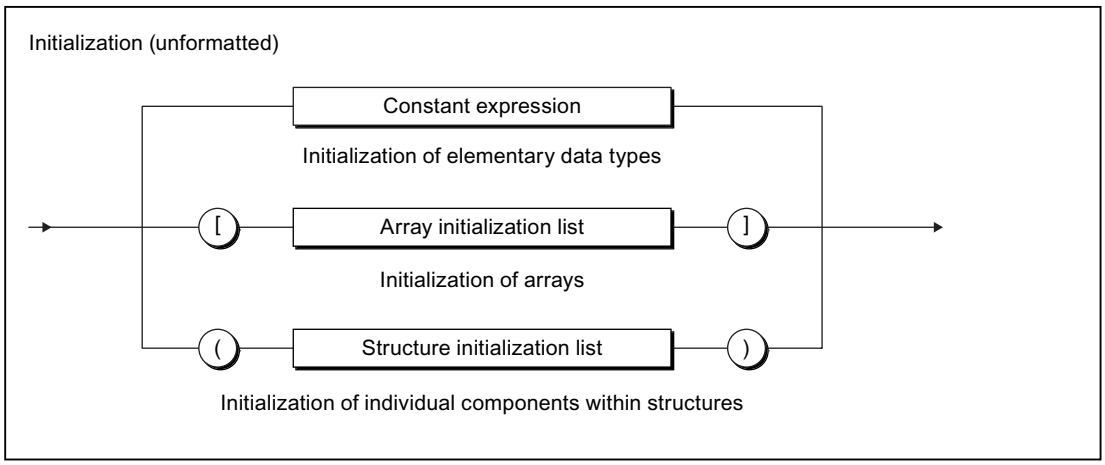


図 3-12　　構文: 変数の初期化

変数に割り付けた初期化値は、コンパイル時に定数式から計算されます。構文は図を参照してください。定数式の構文については、*「構文: 定数式」*の図を参照してください。

変数リスト(a1、a2、a3、.. : INT := ..)は共通の値を使用して初期化できることに注意してください。この場合、変数を個別に初期化する必要はありません(a1 : INT := .. ; a2 : INT := .. ;など)。

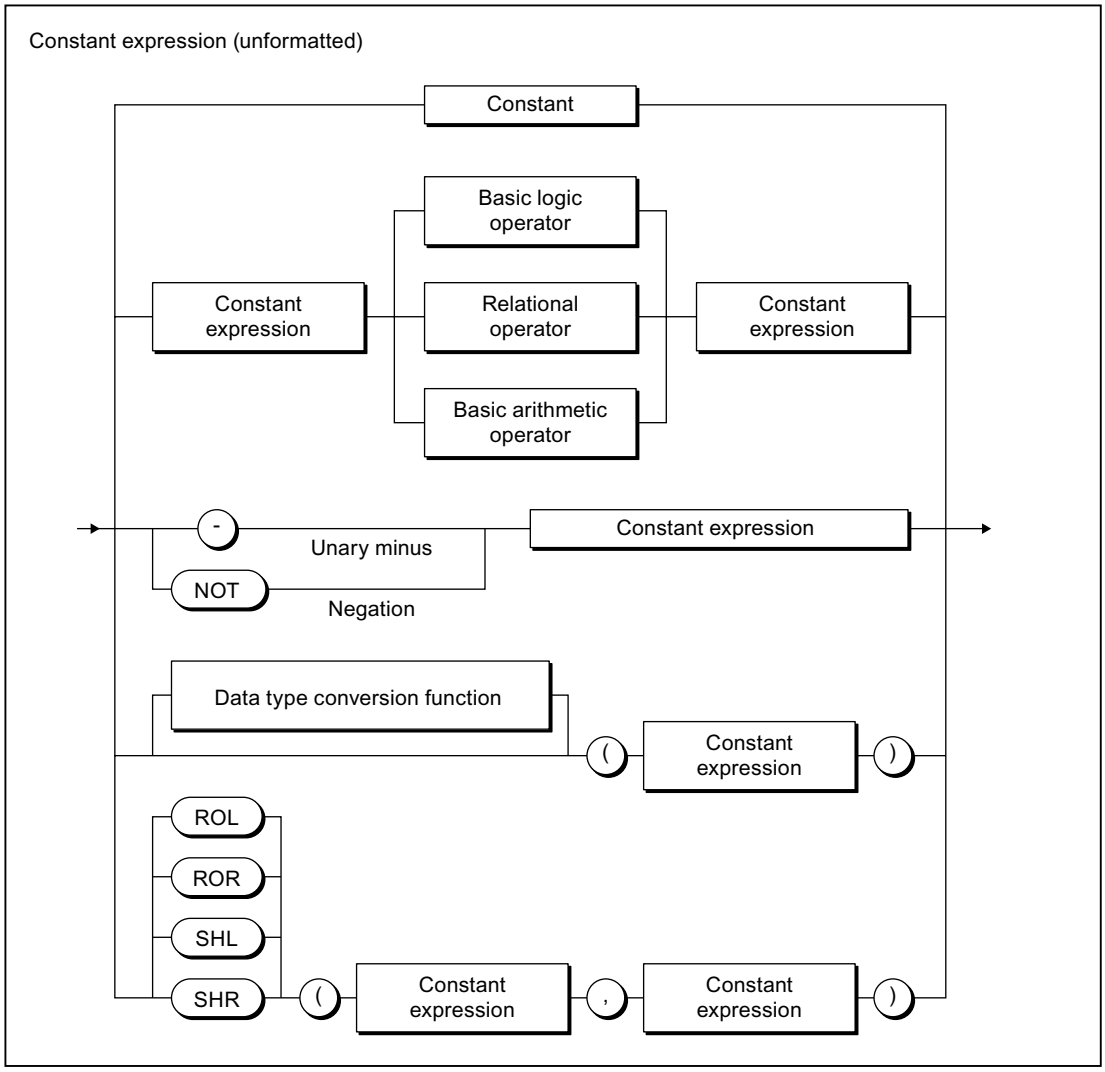


図 3-13　　構文: 定数式

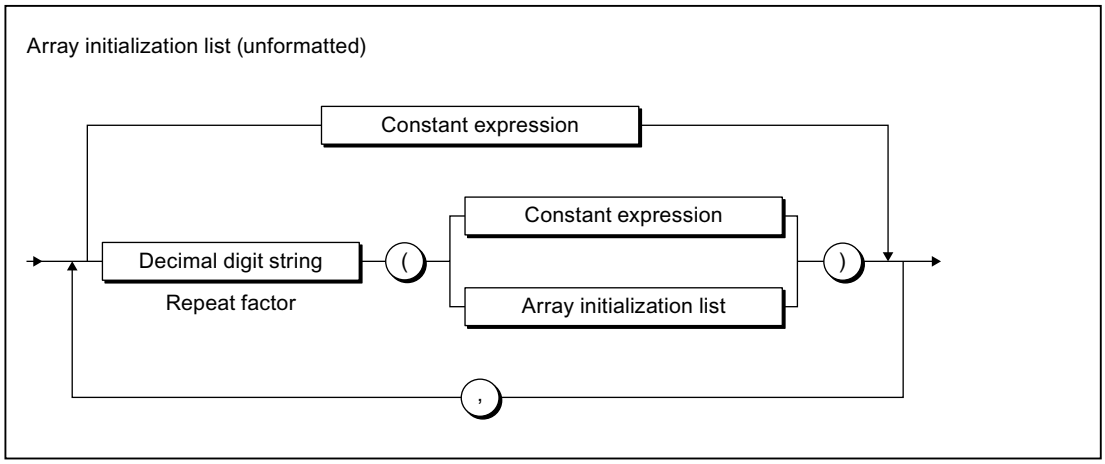


図 3-14　　構文: 配列初期化リスト

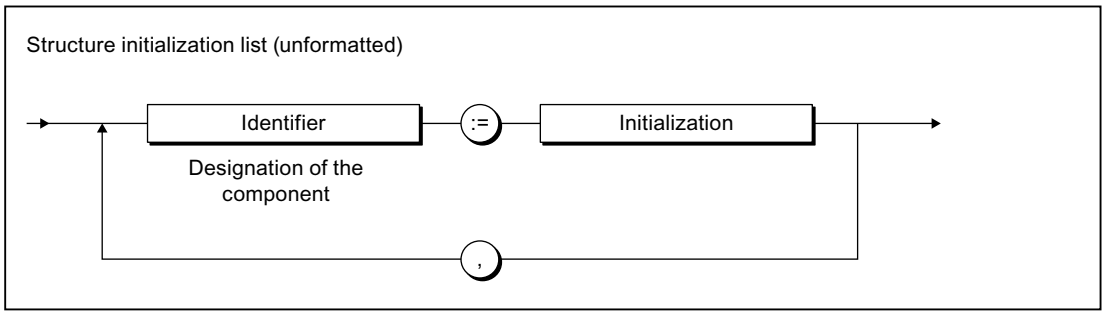


図 3-15　　構文: 構造体初期化リスト

表 3-23　　変数の初期化の例

```
VAR
    //変数の宣言...
    var1 : REAL := 100.0;
    // ... または、同じタイプの複数の変数がある場合:
    var2, var3, var4 : INT := 1;
    var5 : REAL := 3 / 2;
    var6 : INT  := 5 * SHL(1, 4)
    myC1 : C1 := GREEN;
    array1 : ARRAY [0..4] OF INT  := [1, 3, 8, 4, 0];
    array2 : ARRAY [0..5] OF DINT := [6 (7)];
    array3 : ARRAY [0..10] OF INT := [2 (2(3),3(1)),0];
                // [2(3),3(1),2(3),3(1)),0]と等価
                // 以下のように初期化:
                // 配列要素 0, 1        3 で初期化
                // 配列要素 2, 3, 4        1 で初期化
                // 配列要素 5, 6        3 で初期化
                // 配列要素 7, 8, 9        1 で初期化
                // 配列要素 10           0 で初期化
    myAxis : PosAxis := TO#NIL;
END_VAR
```

表 3-24　　データタイプの初期化の例

```
TYPE
    // 派生データタイプの初期化
    type1 : REAL := 10.0;
    // 列挙データタイプの初期化
    cmyk_colour : (cyan, magenta, yellow, black) := yellow;
    // 構造体の初期化
    var_rgb_colour : STRUCT
            red, green, blue : USINT := 255;// 白
        END_STRUCT;
    new_colour : var_rgb_colour := (red := 0, blue := 0);//緑
END_TYPE
```

テクノロジーオブジェクト(TO)データタイプの変数は、コンパイラにより TO#NIL を使用して初期化されます。変数の初期化に対するタスクの影響については、*『SIMOTION 基本機能』*機能マニュアルで説明しています。

3.5.4 定数

定数は、プログラムのランタイム時に変更できない固定値を持つデータです。定数は変数と同様に宣言します。

- ローカル定数の場合、POU の宣言セクションで宣言します(*「構文: POU の定数ブロック」*および *「構文: 定数の宣言」*の図を参照)。
- ユニット定数の場合、STソースファイルのインターフェースまたは実装セクションで宣言します(*「構文: インターフェースまたは実装セクションのユニット定数」*および *「構文: 定数の宣言」*) インターフェースセクションで宣言したユニット定数は、他のSTソースファイルにインポートすることができます(変数モデル (ページ 167)を参照)。

ソースファイルセクションでは、定数宣言の範囲も決定します。

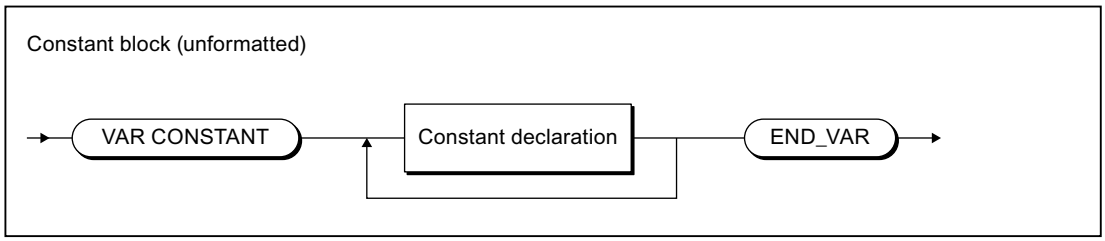


図 3-16　構文: POU の定数ブロック

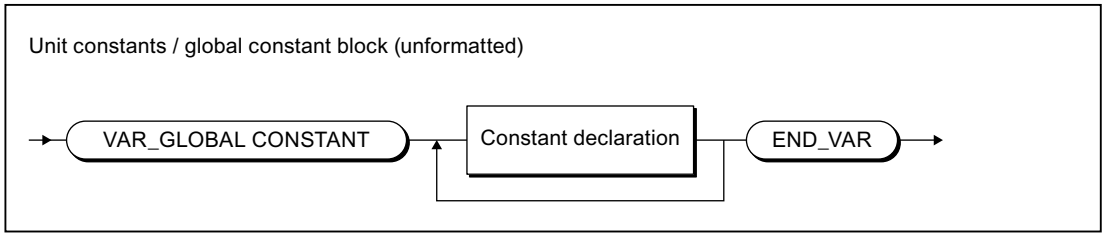


図 3-17　構文: インターフェースまたは実装セクションのユニット定数

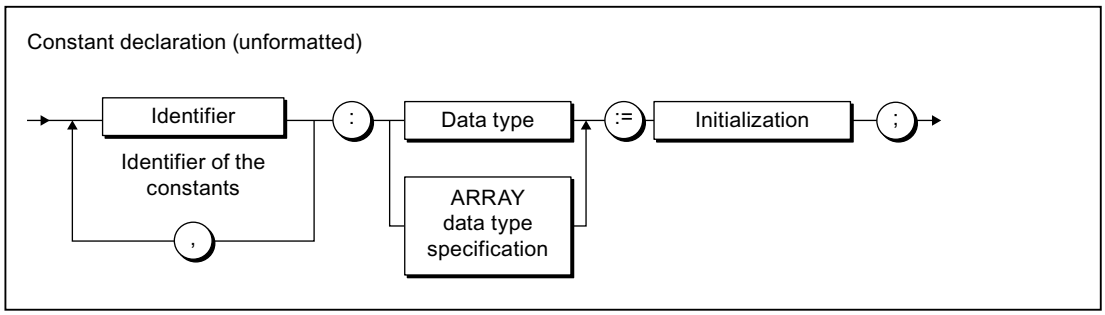


図 3-18　構文: 定数の宣言

定数に割り付けた値は、コンパイル時に定数式から計算されます。定数式の構文については、*「構文: 定数式」*の図を参照してください。

表 3-25　　定数の例

```
VAR CONSTANT
    PI : REAL := 3.1415;
    intConst      : INT := 10;
    sintConst     : SINT := 0;
    dintConst     : DINT := 10_000;
    timeConst     : TIME := TIME#1h;
    strConst      : STRING[40] := 'Example of a string';
    Two_PI        : REAL := 2 * PI;
END_VAR
```

3.6　値割り付けおよび式

値割り付けは、例の一部として挙げたステートメントの文字列:=を使用して(ステートメント (ページ 71)の表*「ステートメントの例」*を参照)、またはソースファイルセクションの宣言セクションで変数を初期化するときに、既に作成しています。

ただし、これは、値割り付けを定式化するために使用できるオプションのほんの一部です。本マニュアルのこのセクションでは、説明のため多くの例を使用してこの重要なトピックについて詳しく記述します。

注記

演算式および論理式では、結果は常に式の最多数のフォーマットで計算され、結果のデータタイプに変換されます。値割り付けでは、暗黙の変換が常に可能なわけではありません。このエラーソースとその解決方法の詳細については、*『SIMOTION 基本機能』*機能マニュアルを参照してください。

下記も参照

エラーの回避と効率的なプログラミングに関する注意事項 (ページ 223)

3.6.1　値割り付け

3.6.1.1　値割り付けの構文

値割り付けは、式の値を変数に割り付けるために使用します。前の値は上書きされます。値を正しく割り付けるには、宣言セクションで変数を宣言しておく必要があります(変数宣言の構文 (ページ 88)を参照)。

以下の構文ダイアグラムに示すように、割り付け記号:=の右側で式が評価されます。結果は、名前が割り付け記号の左側にある変数(ターゲット変数)に格納されます。形式的な視点からサポートされているすべてのターゲット変数を図に示します。

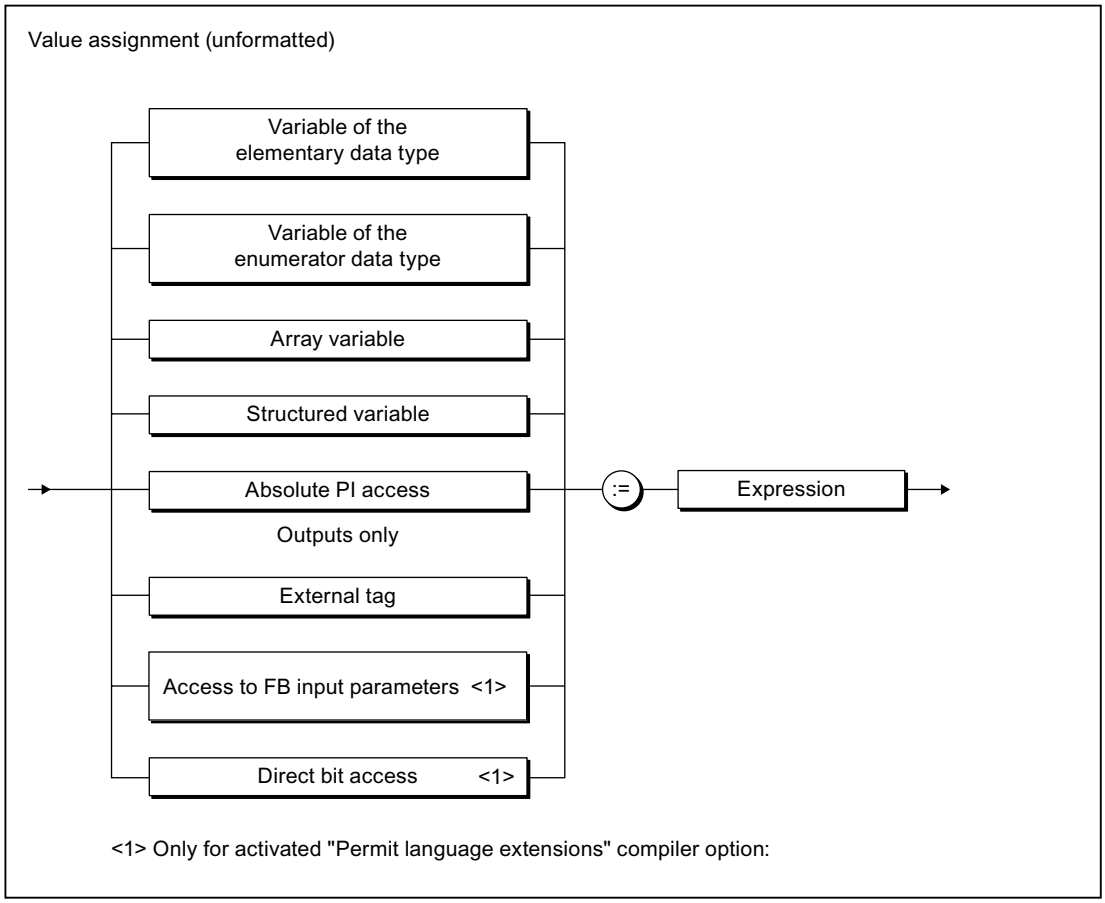


図 3-19　　構文: 値割り付け

以下には、値割り付けの左側についての説明と例が含まれています。

- 基本データタイプの変数を使用した値割り付け (ページ 97),
- 列挙子派生データタイプの変数を使用した値割り付け (ページ 101)
- ARRAY派生データタイプの変数を使用した値割り付け (ページ 101)
- STRUCT派生データタイプの変数を使用した値割り付け (ページ 102)
- (プロセスイメージのアドレスに対する)絶対PIアクセスを使用した値割り付け。参照: BackgroundTaskの固定プロセスイメージへの絶対アクセス(絶対PIアクセス) (ページ 200).

値割り付けの右側、すなわち式を形成する方法は、式 (ページ 103)で説明しています。

3.6.1.2 基本データタイプの変数を使用した値割り付け

以下のいずれかの条件を満たした場合、基本データタイプを含む式(「基本データタイプ」も参照)を変数に割り付けることができます。

- 式とターゲット変数が同じデータタイプを持っている。

  STRING データタイプに関する以下の情報に注意してください。

- 式のデータタイプをターゲット変数のデータタイプに暗黙的に変換できる(基本データタイプの変換 (ページ 125)、および *『SIMOTION基本機能』*機能マニュアルの *「数値データタイプおよびビットデータタイプの変換用ファンクション」*を参照)。

例

```
elemVar         := 3*3;
elemVar         := elemVar1;
```

下記も参照

STRING基本データタイプの変数を使用した値割り付け (ページ 97)

ビットデータタイプの変数を使用した値割り付け (ページ 99)

3.6.1.3 STRING 基本データタイプの変数を使用した値割り付け

STRING データタイプの変数間の割り付け

異なる長さを使用して宣言した STRING データタイプ(文字列)の変数間の割り付けに制限はありません。ターゲット変数の宣言された長さが割り付けた文字列の現在の長さより短い場合、文字列はターゲット変数の長さに切り捨てられます。

例外: 入力/出力割り付け(入/出力パラメータへのパラメータの転送)には、次のことが適用されます。すなわち、割り付けた変数(実パラメータ)の宣言された長さがターゲット変数(仮入/出力パラメータ)の宣言された長さより長いか同じである必要があります。入/出力パラメータへのパラメータの転送 (ページ 138)を参照してください。

関連項目 基本データタイプの値の範囲限界 (ページ 75)

例:

```
string20 := 'ABCDEFG';
string20 := string30;
```

文字列の要素へのアクセス

配列[1..n]の要素と同じ方法で文字列の個々の要素をアドレス指定することができます。これらの要素は、基本データタイプ BYTE に暗黙的に変換されます。このようにして、文字列の要素と BYTE データタイプの変数間の割り付けを行うことができます。

例:

```
byteVar := string20[5];
string20[10] := byteVar;
```

以下の特殊な場合を考慮する必要があります。

1. BYTE データタイプの変数を文字列要素に割り付ける場合
   (例: `stringVar[n:] := byteVar`)
   – 値を割り付ける文字列要素が、文字列の宣言された長さの範囲外にある。

     文字列は変更されないままになり、TSI#ERRNO が 1 に設定されます。
   – 値を割り付ける文字列要素が、文字列の割り付けられた長さの範囲外だが (n > LEN(stringVar))、宣言された長さの範囲内にある。

     文字列の長さが調整され、LEN(stringvar)と n の間の文字列要素が$00 に設定されます。
2. 文字列要素を BYTE データタイプの変数に割り付ける場合
   (`byteVar := stringVar[n:]`)
   – 変数を割り付ける文字列要素が、文字列の割り付けられた長さの範囲外にある (n > LEN(stringVar))。

     変数が 16#00、TSI#ERRNO が 2 に設定されます。

文字列の編集

文字列の連結、文字の置換や抽出など、文字列の編集を行うには、さまざまなシステムファンクションを使用することができます。*『SIMOTION 基本機能』*機能マニュアルを参照してください。

数字と文字列間の変換

数値データタイプと文字列の変数間の変換には、さまざまなシステムファンクションを使用することができます。基本データタイプの変換 (ページ 125)および *『SIMOTION基本機能』*機能マニュアルを参照してください。

3.6.1.4 ビットデータタイプの変数を使用した値割り付け

ビットデータタイプ変数の個々のビットへのアクセス

BYTE、WORD、または DWORD データタイプの変数の個々のビットにアクセスすることもできます。

- 標準ファンクションを使用(*『SIMOTION 基本機能』*機能マニュアルを参照)

  _getBit、_setBit、および_toggleBit の各ファンクションを使用して、ビット文字列のビットの読み取り、書き込み、または反転を行うことができます。

  変数を介してビットの番号を指定することができます。

- 直接ビットアクセスを使用

  変数の後ろに区切り点を使用して、定数としてアクセスしたい変数のビットを定義することができます。

  定数を介してのみ、ビットの番号を指定できます。

  このオプションを使用できるためには、[Permit language extensions]コンパイラオプションを有効にする必要があります(コンパイラのグローバル設定 (ページ 31)および ST コンパイラのローカル設定 (ページ 32)を参照)。

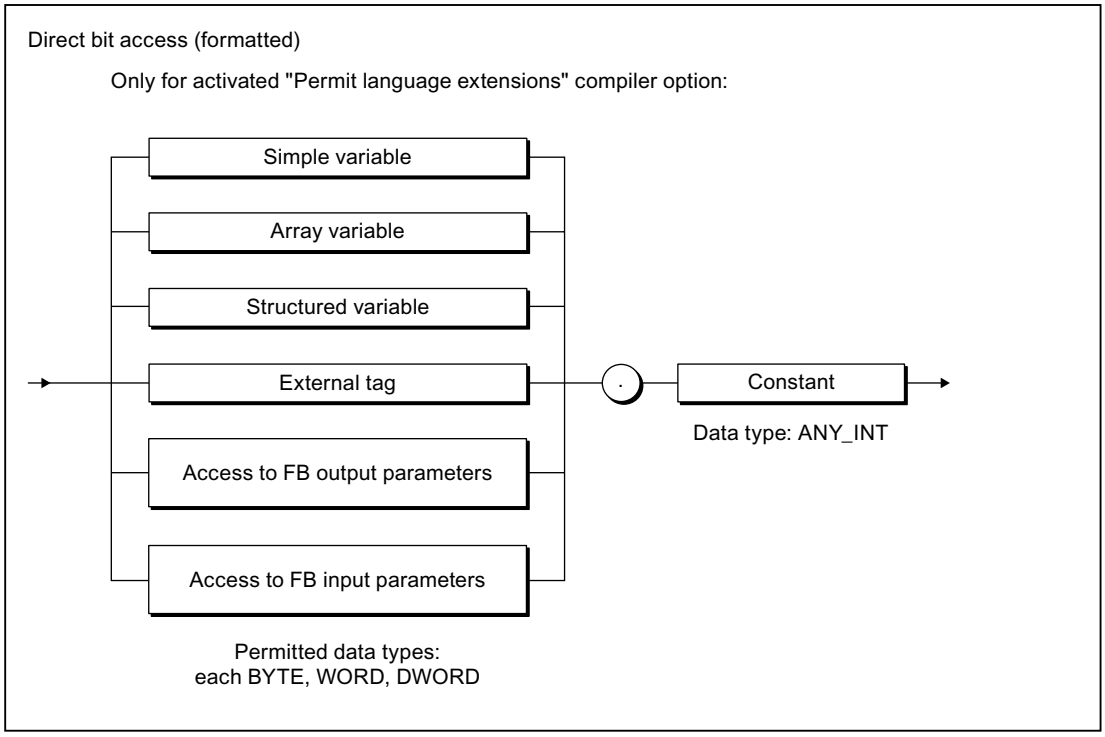


図 3-20 構文: 直接ビットアクセス

表 3-26　　直接ビットアクセスの例

```
// [Permit language extensions]コンパイラオプションが有効にされている場合のみ
FUNCTION f : VOID
    VAR CONSTANT
        BIT_7 : INT := 7;
    END_VAR
    VAR
        dw : DWORD;
        b: BOOL;
    END_VAR
    b := dw.BIT_7;  // ビット 7 へのアクセス
    b := dw.3;      // ビット 3 へのアクセス
//  b := dw.33;     // コンパイラエラー
                    // ビット 33 は使用できない。
END_FUNCTION
```

通知

I/O 変数またはシステム変数のビットへのアクセスは、他のタスクによって割り込まれることがあります。したがって、一貫性が保たれる保証はありません。

ビットデータタイプの変数の編集

以下のことが可能です。

1. 同じデータタイプの複数の変数を上位データタイプの 1 つの変数に(たとえば、BYTE データタイプの 2 つの変数を WORD データタイプの 1 つの変数に)結合することができます。これを行うには、WORD_FROM_2BYTE など、さまざまなシステムファンクションを使用できます。
2. 1 つの変数を下位データタイプの複数の変数に(たとえば、DWORD データタイプの 1 つの変数を BYTE データタイプの 4 つの変数に)分割することができます。これを行うには、DWORD_TO_4BYTE など、さまざまなシステムファンクションを使用できます。
3. 変数内でビットを回転またはシフトさせることができます。これを行うには、ビット文字列の標準ファンクション ROL、ROR、SHL、および SHR を使用できます。

これらのシステムファンクションおよびシステムファンクションブロックについては、『*SIMOTION 基本機能*』機能マニュアルで説明しています。

論理演算子

ビットデータタイプの変数は、論理演算子を使用して結合することができます。論理式およびビットシリアル式 (ページ 111)を参照してください。

3.6.1.5 列挙子派生データタイプの変数を使用した値割り付け

列挙子派生データタイプの各式と各変数(関連項目: 列挙子派生データタイプ (ページ 82))には、同じタイプの別の変数を割り付けることができます。

```
type1 := BLUE;
```

3.6.1.6 ARRAY 派生データタイプの変数を使用した値割り付け

配列は、すべての同じタイプの複数の次元と配列要素で構成されます(関連項目: ARRAY派生データタイプ (ページ 80)).

変数に配列を割り付けるには、さまざまな方法があります。完全な配列、個々の要素、または配列の一部を割り付けることができます。

- コンポーネントのデータタイプと配列制限(可能な最小および最大の配列インデックス)の両方が同じである場合、完全な配列を別の配列に割り付けることができます。有効な割り付けは次のようになります。

```
array_1 := array_2;
```

- 配列名とそれに続けて角括弧で囲んだインデックス値を指定することにより、個々の配列要素をアドレス指定することができます。インデックスは、SINT、USINT、INT、UINT、または DINT データタイプの演算式である必要があります。

```
elem1        := array [i];
array_1 [2]  := array_2 [5];
array [j]    := 14;
```

- 配列の各次元の角括弧のペアを右から省略すると、有効なサブ配列に対する値割り付けを取得することができます。これにより、次元数が残りのインデックスの数と等しい配列の一部の領域がアドレス指定されます(以下の例を参照)。

  結果として、マトリクス内の行と個々のコンポーネントを参照できますが、閉じた列(から...までという意味で閉じた)は参照できません。有効な割り付けは次のようになります。

```
matrix1[i] := matrix2[k];
array1 :=  matrix2 [k];
```

3.6.1.7 STRUCT 派生データタイプの変数を使用した値割り付け

STRUCTデータタイプ指定を含むユーザ定義データタイプの変数は、呼び出された構造体変数です(STRUCT (構造体)派生データタイプ (ページ 83)も参照)。この変数は、完全な構造体またはこの構造体のコンポーネントを表すことができます。

構造体変数の有効なパラメータは次の通りです。

```
struct1                 //構造体の識別子
struct1.elem1           //構造体コンポーネントの識別子
struct1.array1          //構造体内の単純配列
                        //の識別子
struct1.array1[5]       //構造体内の配列コンポーネント
                        //の識別子
```

変数に構造体を割り付けるには、2 つの方法があります。完全な構造体または構造体コンポーネントを参照できます。

- 完全な構造体を別の構造体に割り付けることができます。これは両方の構造体コンポーネントのデータタイプと名前が一致している場合に限ります。

  有効な割り付けは次のようになります。

```
struct1 := struct2;
```

- タイプ互換変数、タイプ互換式、または別の構造体コンポーネントを各構造体コンポーネントに割り付けることができます。

  有効な割り付けは次のようになります。

```
struct1.elem1               := Var1;
struct1.elem1               := 20;
struct1.elem1               := struct2.elem1;
struct1.array1              := struct2.array1;
struct1.array1[10]          := 100;
```

注記

ファンクションブロックの出力変数、すなわちファンクションブロックの結果にアクセスするのにも、*FBInstanceName.OutputParameter*フォーマットの構造体変数、たとえば *myCircle.circumference*を使用します。ファンクションブロックの詳細については、ファンクションの定義 (ページ 132)および ファンクションブロックの定義 (ページ 133)*の説明を参照してください。*

構造体変数のその他の用途としては、TO 変数および基本システムの変数にアクセスすることがあります。

3.6.2 式

式とは、プログラムのコンパイルまたは実行時に計算される値を表します。式は、オペランド(定数、変数、ファンクション値など)と演算子(*、/、+、-など)で構成されます。

オペランドのデータタイプと演算子によって式の種類が決まります。

ST では以下の種類の式を使用します。

- 演算式
- 関係式
- 論理式
- ビットシリアル式

3.6.2.1 式の結果

式の結果は次のように処理することができます。

- 変数に割り付ける。
- 制御ステートメントの条件として使用する。
- ファンクションまたはファンクションブロック呼び出しのパラメータとして使用する。

注記

ARRAY宣言での変数の初期化およびインデックスの指定には、以下の要素のみを含む式を使用することができます(初期化式については、*「構文: 定数式」*の図を参照 変数またはデータタイプの初期化 (ページ 90)。以下参照)。

- 定数
- 基本算術演算
- 論理演算および関係演算
- ビット文字列の標準ファンクション

3.6.2.2 式の解釈順序

式の解釈順序は以下に応じて決まります。

- 使用されている演算子の優先度
- 左から右のルール
- 丸括弧の使用(同じ優先度の演算子の場合)

式は特定の**ルール**に従って処理されます。

- 演算子は優先度に従って実行されます
  (演算子の優先度 (ページ 113)の表を参照)。
- 同じ優先度の演算子は左から右に実行されます。
- 識別子の前のマイナス符号は-1 による乗算を示します。

- 算術演算子の直後に別の算術演算子を使用することはできません。
  したがって、*a * -b* という式は無効ですが、*a * (-b)*は許可されます。
- 丸括弧は演算子の優先順位を無効にします。すなわち、丸括弧は最も高い優先度を持ちます。
- 丸括弧で囲まれた式は個々のオペランドとして扱われ、常に最初に評価されます。
- 開く丸括弧の数は閉じる丸括弧の数と等しくなければなりません。
- 文字または論理データに対して算術演算を使用することはできません。したがって、*(n<=0) + (n<0)*などの式は無効です。

表 3-27　　式の例

```
testVar           // オペランド
A AND (B)         // 論理式
A AND (NOT B)     // 否定を含む論理式
(C) < (D)         // 関係式
3+3*4/2           // 演算式
```

3.6.3 オペランド

定義

オペランドは、式を定式化するために使用できるオブジェクトです。オペランドは以下の構文ダイアグラムで表すことができます。

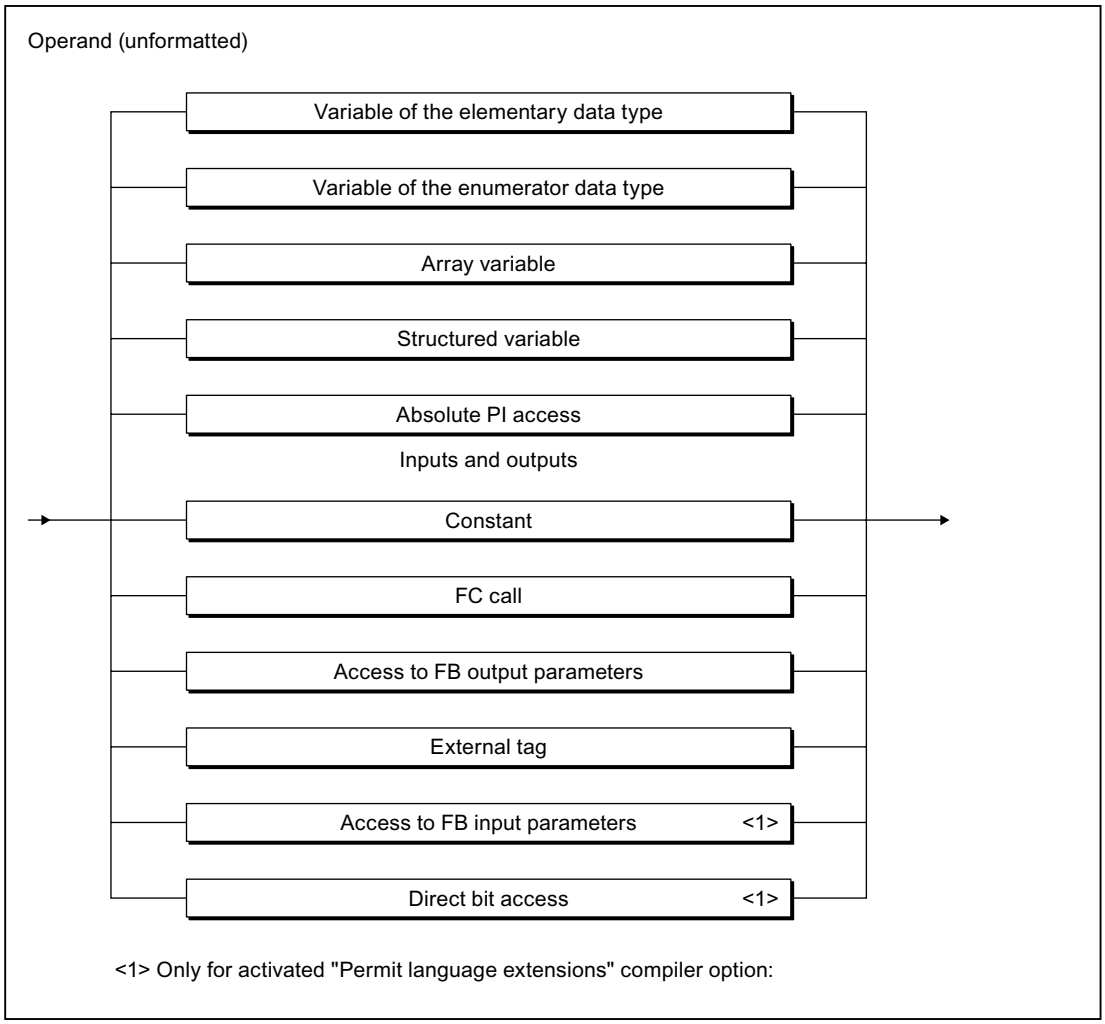


図 3-21 構文:オペランド

表 3-28 オペランドの例

```
intVar
5
%I4.0
PI
NOT TRUE
axis1.motionStateData.actualVelocity
```

3.6.4 演算式

演算式とは算術演算子で形成される式のことです。演算式を使用すると、数値データタイプの処理が可能になります。

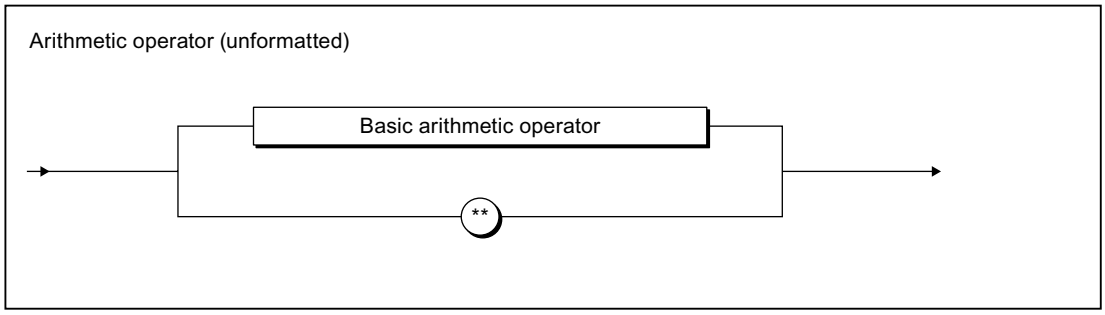


図 3-22　構文: 算術演算子

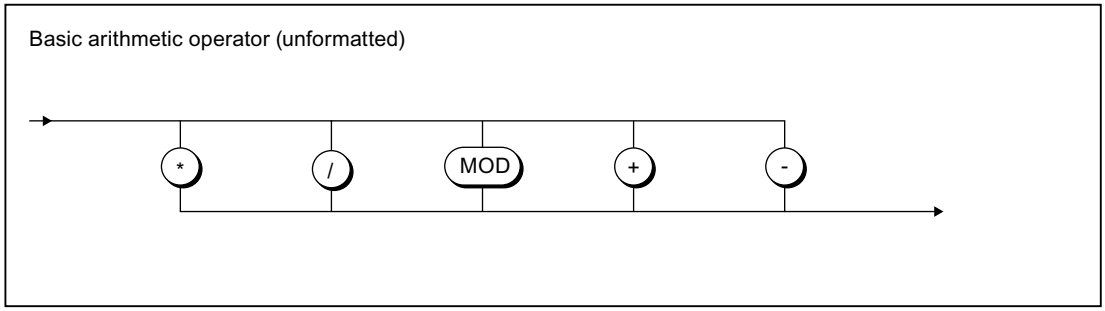


図 3-23　構文: 基本算術演算子

以下の表は、可能な演算子のリストと、結果が割り付けられるタイプをオペランドに応じて示しています。ここでは、で定義された一般的なデータタイプを使用しています。

注記

標準数値ファンクションを使用するとその他の演算が可能です。*『SIMOTION 基本機能』*機能マニュアルの*「標準数値ファンクション」*を参照してください。

演算式の演算子は、その優先度の順に処理されます(演算子の優先度 (ページ 113)を参照)。

読みやすさを高めるため、負の数字は絶対に必要というわけではなくても丸括弧で囲むことをお勧めします。

表 3-29　　算術演算子

命令	演算子	データタイプ		
		第一オペランド	第二オペランド	結果 [1]
べき乗 (ファンクション EXPT も参照)	**	ANY_REAL[2]	ANY_REAL	ANY_REAL[3]
単項マイナス	–	ANY_NUM	(なし)	ANY_NUM
		ANY_BIT	(なし)	ANY_BIT
乗算	*	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT	ANY_BIT	ANY_BIT
		TIME	ANY_NUM	TIME
除算	/	ANY_NUM	ANY_NUM[4]	ANY_NUM
		ANY_BIT	ANY_BIT[4]	ANY_BIT
		TIME	ANY_NUM[4]	TIME
		TIME	TIME[4]	UDINT
モジュロ除算	MOD	ANY_INT	ANY_INT[4]	ANY_INT
		ANY_BIT	ANY_BIT[4]	ANY_BIT
加算	+	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT	ANY_BIT	ANY_BIT
		TIME	TIME	TIME[5]
		TOD	TIME	TOD[5)]
		DT	TIME	DT[6]
減算	–	ANY_NUM	ANY_NUM	ANY_NUM
		ANY_BIT	ANY_BIT	ANY_BIT
		TIME	TIME	TIME
		TOD	TIME[7]	TOD
		DATE	DATE	TIME[8]
		TOD	TOD	TIME[8]
		DT	TIME	DT
		DT	DT	TIME[8]

[1] 結果のデータタイプは、オペランドの最も強力なデータタイプによって決まります。

[2] 第一オペランドはゼロより大きくなければなりません。
SIMOTION Kernel のバージョン V4.1 の例外:
– 第二オペランドが整数の場合、第一オペランドはゼロより小さくても構いません。
– 第二オペランドが正の数の場合、第一オペランドはゼロと等しくても構いません。
SIMOTION Kernel のバージョン V4.0 までは次のことが適用されます: 第一オペランドがゼロと等しい場合、ExecutionFaultTask(実行エラータスク)によってエラーメッセージが捕捉されることがあります。

[3] 第一オペランドのデータタイプ

[4] 第二オペランドはゼロと等しくてはいけません。

[5] オーバーフローによると思われる加算

[6] データの修正による加算

[7] 計算前の TOD に対する TIME の制限

[8] これらの演算は、TIME データタイプの最大値のモジュロに基づきます。

注記

一般的な ANY_REAL データタイプの変数を使用した演算で値範囲の制限を超過した場合、結果には IEEE 754 に準拠した等価のビットパターンが含まれます。

演算で値範囲を超過したかどうかを明確にするには、ファンクション*_finite* を使用して結果を確認することができます(*『SIMOTION 基本機能』*機能マニュアルを参照)。

3.6.4.1 演算式の例

数字を含む演算式の例

i と *j* がそれぞれ 11 と-3 の値を持つ整数変数(たとえば、INT データタイプの変数)であると想定すると、この例の整数式とそれに対応する値は以下のように示されます。

式	規格値
i + j	8
i - j	14
i * j	-33
i MOD j	-2
i / j	-3

時刻指定を含む有効な演算式の例

以下の変数を想定します。

変数	内容	データタイプ
t1	T#1D_1H_1M_1S_1MS	TIME
t2	T#2D_2H_2M_2S_2MS	TIME
d1	D#2004-01-11	DATE
d2	D#2004-02-12	DATE
tod1	TOD#11:11:11.11	TIME_OF_DAY
tod2	TOD#12:12:12.12	TIME_OF_DAY
dt1	DT#2004-01-11-11:11:11.11	DATE_AND_TIME
dt2	DT#2004-02-12-12:12:12.12	DATE_AND_TIME

これらの変数を含む式とその値を例で示します。

式	規格値
t1 + t2	T#3d3h3m3s3ms
dt1 + t1	DT#2004-01-12-12:12:12.111
t1 - t2	T#48D_16H_1M_46S_295MS
t1 * 2	T#2D_2H_2M_2S_2MS
t1 / 2	T#12H_30M_30S_500MS
DATE_AND_TIME_TO_TIME_OF_DAY(dt1)	TOD#11:11:11.11
DATE_AND_TIME_TO_DATE(dt1)	D#2004-01-11

3.6.5 関係式

定義

関係式とは、関係演算子(図を参照)で形成される、BOOL データタイプの式のことです。

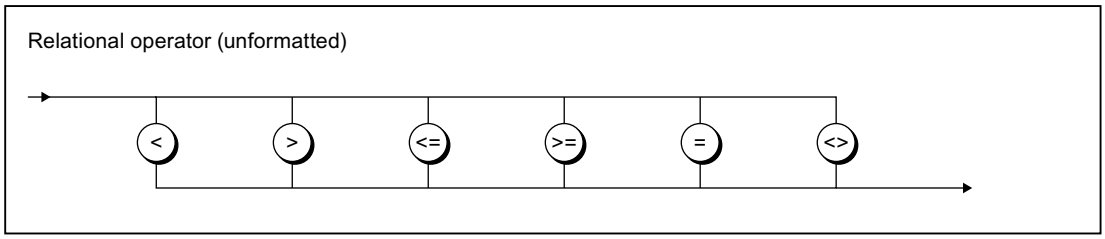


図 3-24　　構文: 関係演算子

関係演算子は、2 つのオペランドの値を比較し(表を参照)、結果としてブール値を返します。
第一オペランド 演算子 第二オペランド -> ブール値

表 3-30　　関係演算子の意味

演算子	意味
>	1. オペランドが第二オペランド**より大きい**
<	1. オペランドが第二オペランド**より小さい**
>=	1. オペランドが第二オペランド**より大きいか、等しい**
<=	1. オペランドが第二オペランド**より小さいか、等しい**
=	1. オペランドが第二オペランドと**等しい**
<>	1. オペランドが第二オペランドと**等しくない**

関係式の結果は次のようになります。

- 比較条件が満たされた場合、1 (TRUE)
- 比較条件が満たされなかった場合、0 (FALSE)

以下の表は、2 つのオペランドのデータタイプと関係演算子の許容される組み合わせを示します。

表 3-31　関係式: データタイプと関係演算子の許容される組み合わせ

データタイプ		使用できる関係演算子
1. オペランド	2. オペランド	
ANY_NUM	ANY_NUM[1]	<, >, <=, >=, =, <>
ANY_BIT	ANY_BIT	<, >, <=, >=, =, <>
DATE	DATE	<, >, <=, >=, =, <>
TIME_OF_DAY (TOD)	TIME_OF_DAY (TOD)	<, >, <=, >=, =, <>
DATE_AND_TIME (DT)	DATE_AND_TIME (DT)	<, >, <=, >=, =, <>
TIME	TIME	<, >, <=, >=, =, <>
STRING	STRING[2]	<, >, <=, >=, =, <>
列挙子データタイプ	列挙子データタイプ [3]	=, <>
ARRAY	ARRAY[3]	=, <>
構造体(STRUCT)	構造体(STRUCT)[3]	=, <>

[1] 両方のオペランドを暗黙の変換を通じて最も強力なデータタイプに変換する必要があります(基本データタイプの変換 (ページ 125)、および *『SIMOTION基本機能』*機能マニュアルの *「数値データタイプおよびビットデータタイプの変換用ファンクション」*を参照)。

[2] STRING データタイプの変数は、文字列の宣言された長さと関係なく比較することができます。長さが異なる STRING データタイプの 2 つの変数を比較するには、短い方の文字列を、右側に$00 を挿入することにより長い方の文字列の長さに拡張します。比較は左から始まり右向きに行われます。この比較は、各文字の ASCII コードに基づきます。例: 'ABC' < 'AZ' < 'Z' < 'abc' < 'az' < 'z'

[3] 第一オペランドのデータタイプ

関係式、およびBOOLデータタイプの変数または定数は、論理演算子を使用して論理式に結合することができます(論理式およびビットシリアル式 (ページ 111)を参照)。
これにより、*If a < b and b < c, then …*などの問い合わせを実装できるようになります。

通知

式では、関係演算子は論理演算子より高い優先度を持ちます(演算子の優先度 (ページ 113)を参照)。したがって、関係式のオペランドが論理式またはビットシリアル式である場合、オペランドを角括弧で囲む必要があります。

REAL 変数や LREAL 変数(さらに、軸位置などの対応するシステム変数)を比較すると、エラーが発生する可能性があることに注意してください。

表 3-32　関係式の例

```
IF A = 2 THEN
     //...
END_IF;
var_1 := B < C;          // BOOL データタイプの var_1
IF D < E OR var_2 THEN  // BOOL データタイプの var_2
     // ...
END_IF;
```

3.6.6 論理式およびビットシリアル式

定義

論理演算子 AND、&、XOR、および OR では、一般的なデータタイプ ANY_BIT (BOOL、BYTE、WORD、または DWORD)のオペランドおよび式を結合することができます。

論理演算子 NOT では、データタイプ ANY_BIT のオペランドおよび式を否定することができます。

以下の表は、使用可能な演算子に関する情報を示します。

表 3-33 論理演算子

命令	演算子	1. オペランド	2. オペランド	結果 [1]
否定	NOT	ANY_BIT	-	ANY_BIT
結合	AND または&	ANY_BIT	ANY_BIT	ANY_BIT
排他的論理和	XOR	ANY_BIT	ANY_BIT	ANY_BIT
論理和	OR	ANY_BIT	ANY_BIT	ANY_BIT
[1] 結果のデータタイプは、オペランドの最も強力なデータタイプによって決まります。				

式には以下を指定します。

- BOOL データタイプのオペランドだけを使用する場合、**論理式**

  演算子は、以下の真偽表に示した効果を持ちます(以下の表を参照)。

  論理式の結果は 1 (TRUE)または 0 (FALSE)です。

- BYTE、WORD、または DWORD データタイプのオペランドを使用する場合、**ビットシリアル式**

  演算子は、オペランドの個々のビットに対して、以下の真偽表に示した効果を持ちます。

表 3-34 論理演算子の真偽表

オペランド (BOOL データタイプ)		結果(BOOL データタイプ)				
a	b	NOT a	NOT b	a AND b a & b	a XOR b	A OR b
0	0	1	1	0	0	0
0	1	1	0	0	1	1
1	0	0	1	0	1	1
1	1	0	0	1	0	1

例

表 3-35　　論理式

式(n = 10 と仮定)	規格値
(n>0) AND (n<20)	TRUE
(n>0) AND (n<5)	FALSE
(n>0) OR (n<5)	TRUE
(n>0) XOR (n<20)	FALSE
NOT ((n>0) AND n<20))	FALSE

表 3-36　　ビットシリアル式

式	規格値
2#01010101 AND 2#11110000	2#01010000
2#01010101 OR 2#11110000	2#11110101
2#01010101 XOR 2#11110000	2#10100101
OT 2#01010101	2#10101010

問い合わせにおける式(value1 を 2#01、value2 を 2#11 と仮定)

```
IF (value1 AND value2) = 2#01 THEN...
```

ビットシリアル式から 2#01 が返されるため、条件は TRUE を返します。

3.6.7 演算子の優先度

式 (ページ 103)では、式を定式化するための一般的なルールを説明しました。以下の表は、式内の個々の演算子の優先度を示します。

命令	シンボル	優先度
丸括弧	(式)	最高
ファンクション評価	**識別子**(引数リスト) たとえば、LN(a)、EXPT (a,b)など	
否定 補集合	– **NOT**	
べき乗	**	
乗算 除算 モジュロ	* / **MOD**	
加算 減算	**+** **–**	
比較	**<, >, <=, >=**	
等しい 等しくない	= <>	
論理 AND	**&、AND**	
論理 EXCLUSIVE OR	**XOR**	
論理 OR	**OR**	最低

3.7 制御ステートメント

すべてのステートメントが開始から終了まで順に実行されるようにプログラミングできるソースファイルセクションはほとんどありません。条件が真の場合のみ実行されるステートメント(代替)もあれば、繰り返し実行されるステートメント(ループ)もあるのが普通です。ソースファイルセクション内のプログラム制御ステートメントは、これを実現するための手段です。

3.7.1 IF ステートメント

IF ステートメントは条件文です。IF ステートメントでは、1 つまたは複数のオプションを指定し、そのステートメントセクションのうち 1 つを実行のために選択します(あるいは何も選択しません)。

条件文が実行されると、指定した論理式が評価されます。式の値が TRUE の場合、条件は満たされ、式の値が FALSE の場合、条件は満たされません。

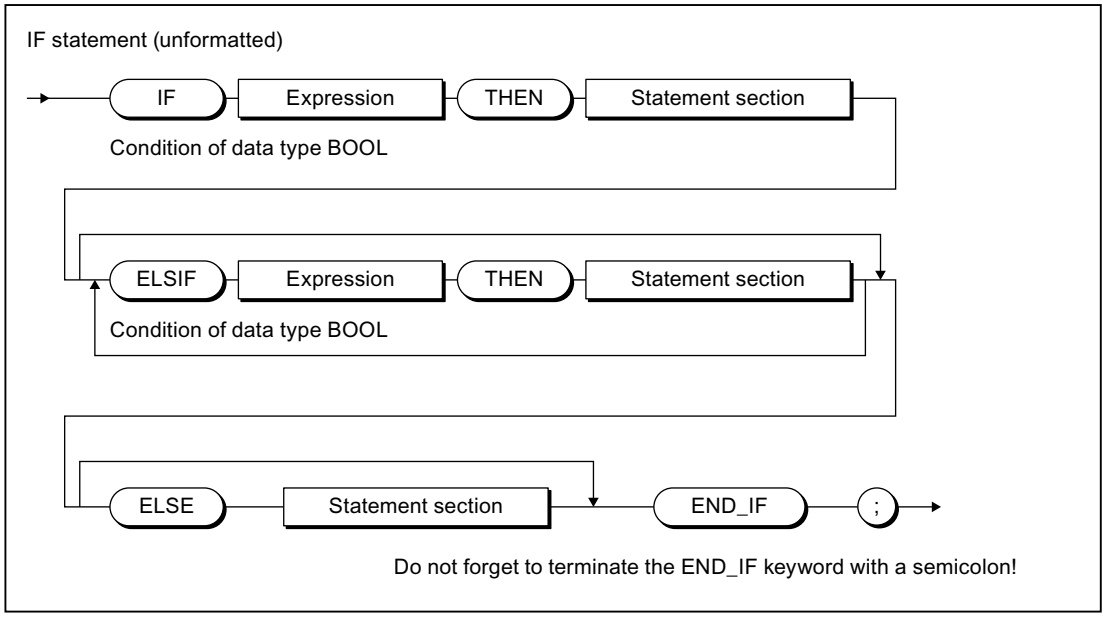


図 3-25　　構文: IF ステートメント

IF ステートメントは、以下のルールに従って処理されます。

1. 最初の式の値が TRUE の場合、THEN の後のステートメントセクションが実行されます。

   END_IF の後で続けてプログラムが再開されます。

2. 最初の式の値が FALSE の場合、ELSIF 分岐の式が実行されます。いずれかの ELSIF 分岐の論理式が TRUE の場合、THEN に続くステートメントセクションが実行されます。

   END_IF の後で続けてプログラムが再開されます。

3. ELSIF 分岐のどの論理式も TRUE でない場合、ELSE の後の一連のステートメントが実行されます(あるいは、ELSE 分岐がない場合は、ステートメントはそれ以上実行されません)。

   END_IF の後で続けてプログラムが再開されます。

任意の数の ELSIF ステートメントをプログラミングできます。

ELSIF 分岐または ELSE 分岐がないこともあります。これは、ステートメントなしで分岐が存在する場合と同様に解釈されます。

注記

一連の IF ステートメントではなく 1 つまたは複数の ELSIF 分岐を使用する利点は、有効な式の後の論理式はもはや評価されなくなるということです。これは、プログラムに必要とされる処理時間を短縮し、不要なプログラムルーチンが実行されるのを回避するのに役立ちます。

表 3-37　　IF ステートメントの例

```
IF A=B THEN
    n:= 0;
END_IF;

IF temperature < 5.0 THEN
    %Q0.0 := TRUE;
ELSIF temperature > 10.0 THEN
    %Q0.2 := TRUE;
  ELSE
    %Q0.1 := TRUE;
END_IF;
```

3.7.2　CASE ステートメント

CASE ステートメントは、n 個のプログラムセクションのうち 1 つを選択するために使用します。

この選択により、選択式(セレクタ)が決まります。

- 一般的なデータタイプ ANY_INT の式
- 列挙データタイプの変数(列挙子)

選択は値のリスト(値リスト)から行います。それにより、プログラムのセクションが各値または値のグループに割り付けられます。

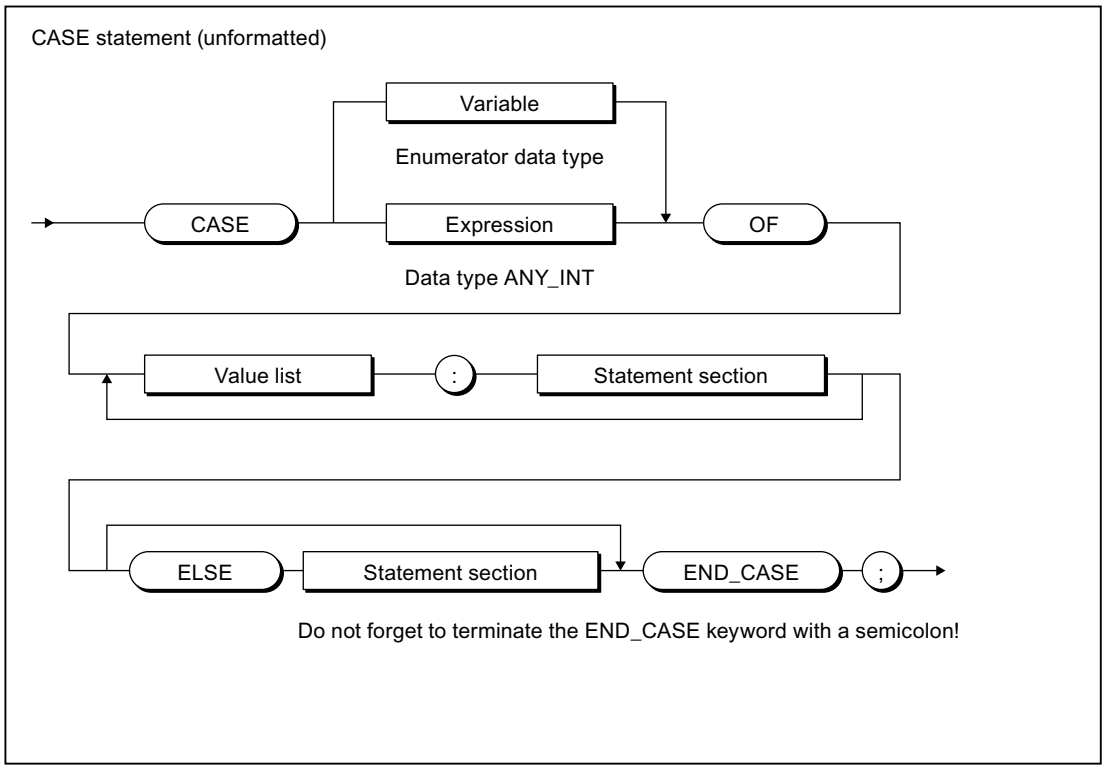


図 3-26　　構文: CASE ステートメント

CASE ステートメントは、以下のルールに従って処理されます。

1. 選択式(セレクタ)が計算されます。選択式は、一般的なデータタイプ ANY_INT (整数)または列挙子データタイプの値を返す必要があります。
2. 次に、値リストにセレクタ値が含まれるかどうかを判別するためにチェックが実行されます。リストの各値は、選択式に許可されている値の 1 つを表します。
3. 一致が検出された場合、リストで割り付けられているプログラムセクションが実行されます。
4. ELSE 分岐はオプションです。これは、一致が検出されない場合に実行されます。
5. ELSE 分岐が欠落していて、一致が検出されない場合、END_CASE の後でプログラムが再開されます。

値リストには、選択式に許可されている値が含まれます。

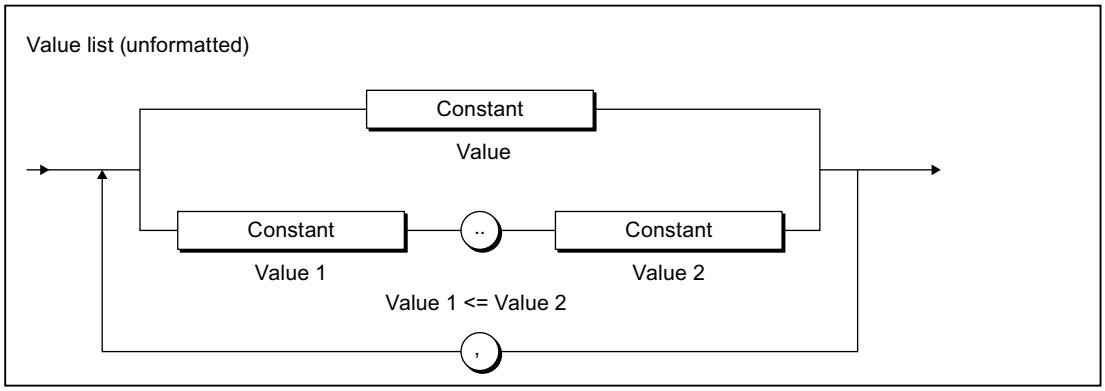


図 3-27　　構文: 値リスト

値リストを定式化するときは、以下の点に注意してください。

- 各値リストは、定数(*value*)、定数のリスト(*value1*、*value2*、*value3 など*)、または定数の範囲(*value1 ~ value2*)から始まることができます。
- 値リストの値は、セレクタの列挙データタイプの整数値または定数/要素である必要があります。

注記

CASE ステートメントの値リストでは、値は 1 回だけ発生するはずです。

値が複数回発生する場合、コンパイラからアラームが発行され、値が最初に発生した値リストに対応するステートメントのセクションだけが実行されます。

以下の例は、CASE ステートメントの使用を示します。

表 3-38　　CASE ステートメントの例

```
CASE intVar OF
    1    : a := 1;
    2,3  : b := 1;
    4..9 : c := 1; d:=2;
  ELSE
    e := 5;
END_CASE;
```

3.7.3 FOR ステートメント

FOR ステートメントまたは反復ステートメントは、一連のステートメントをループで実行します。それにより、各パスの変数(カウント変数)に値が割り付けられます。カウント変数は、SINT、INT、または DINT タイプのローカル変数である必要があります。

FOR を使用したループの定義には、開始および終了値の指定が含まれます。どちらの変数も、カウント変数と同じデータタイプである必要があります。

注記

プログラミング段階でループパスの数が分かっている場合は、FORステートメントを使用します。パスの数が分からない場合は、WHILEまたはREPEATステートメントの方が適しています(WHILEステートメント (ページ 120)および REPEATステートメント (ページ 121)を参照)。

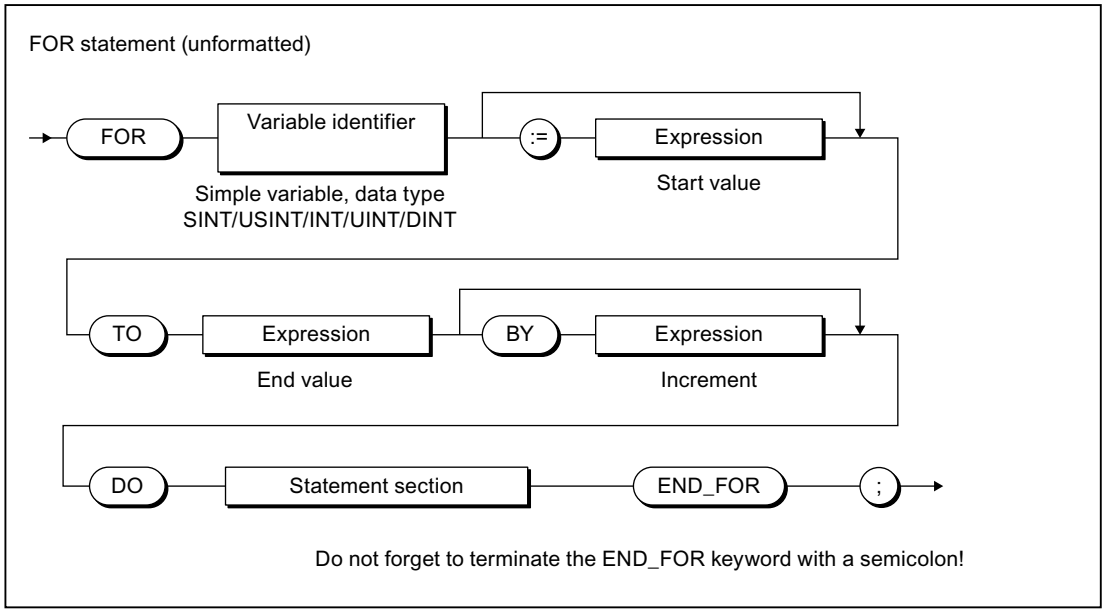


図 3-28　　構文: FOR ステートメント

3.7.3.1 FOR ステートメントの処理

FOR ステートメントは、以下のルールに従って処理されます。

1. ループの開始時にカウント変数が**開始値**に設定され、**終了値**に達するまで毎回のループパスごとに、指定した増分だけカウント変数が増加(正の増分)または減少(負の増分)します。最初のループパスの後の開始値を**現在値**といいます。
2. 各パスで、以下の条件が真かどうかシステムによりチェックが行われます。
   - **開始値または現在値 <= 終了値(正の増分の場合)**、または
   - **開始値または現在値 >= 終了値(負の増分の場合)**

   条件が満たされた場合、ステートメントのシーケンスが実行されます。

   条件が満たされない場合、ループおよび一連のステートメントはスキップされ、END_FOR の後でプログラムが再開されます。
3. ステップ 2 の結果、FOR ループが実行されない場合、カウント変数は現在値を保持します。

3.7.3.2 FOR ステートメントのルール

FOR ステートメントには以下のルールが適用されます。

- *BY [increment]*の指定は省略できます。増分を指定しない場合、デフォルトは+1 になります。
- 開始値、終了値、および増分は式です(式 (ページ 103)を参照)。式は、FORステートメントの最初に 1 回評価されます。
- 開始値と終了値が DINT データタイプの場合、(終了値 - 開始値)の値は倍精度整数の最大の値範囲より小さくなければ(つまり、2**31-1 より小さくなければ)なりません。
- セレクタが真である最初の選択ステートメントだけが実行されます。
- カウント変数には、ループの終了をトリガする値が含まれます。すなわち、カウント変数はループが終了する前に増分します。
- ループの実行中は終了値と増分値を変更することはできません。

3.7.3.3 FOR ステートメントの例

表 3-39 FOR ステートメントの例

```
FOR k := 1 TO 10 BY 2 DO
    l:=l+1;
    // ...
END_FOR;
```

3.7.4 WHILE ステートメント

WHILE ステートメントを使用すると、反復条件の制御の下で一連のステートメントを繰り返し実行することができます。反復条件は、論理式のルールに従って定式化します。

注記

WHILEステートメントは、プログラミング段階でループパスの数が分かっていない場合に使用します。パスの数が分かっている場合は、FORステートメントの方が適しています (FORステートメント (ページ 118)を参照)。

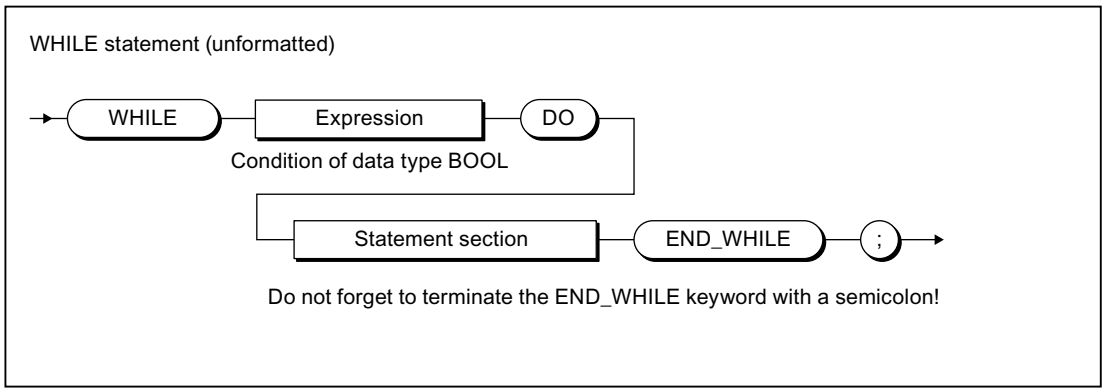


図 3-29 構文: WHILE ステートメント

DO の後のステートメントセクションは、反復条件の値が TRUE になるまで繰り返されます。

WHILE ステートメントは、以下のルールに従って処理されます。

1. ステートメントセクションが実行される**前**に、反復条件がその都度評価されます。
2. 値が TRUE の場合、ステートメントセクションが実行されます。
3. 値が FALSE の場合、WHILE ステートメントは終了し(これは、条件が最初に評価されるときに発生する可能性があります)、END_WHILE の後でプログラムが再開されます。

表 3-40 WHILE ステートメントの例

```
WHILE Index <= 50 DO
      Index:= Index + 2;
END_WHILE;
```

3.7.5 REPEAT ステートメント

REPEAT ステートメントを使用すると、REPEAT と UNTIL の間にプログラミングした一連のステートメントが、終了条件が真になるまで繰り返し実行されます。終了条件は、論理式のルールに従って定式化します。

注記

REPEATステートメントは、プログラミング段階でループパスの数が分かっていない場合に使用します。パスの数が分かっている場合は、FORステートメントの方が適しています (FORステートメント (ページ 118)を参照)。

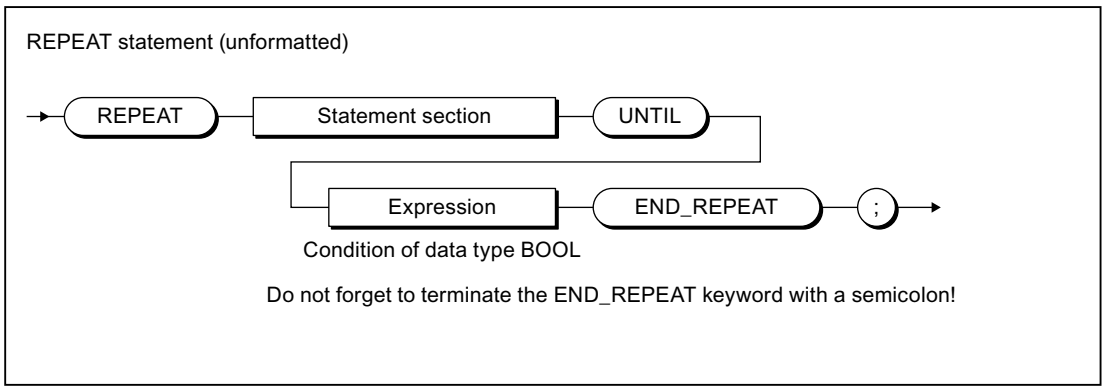


図 3-30　　構文: REPEAT ステートメント

条件は、ステートメントセクションが実行された**後**でチェックされます。つまり、開始時に終了条件が真であっても、ステートメントセクションは少なくとも 1 回実行されます。

REPEAT ステートメントは、以下のルールに従って処理されます。

1. ステートメントセクションが実行された**後**で、反復条件がその都度評価されます。
2. 値が FALSE の場合、ステートメントセクションは再度実行されます。
3. 値が TRUE の場合、REPEAT ステートメントの実行は終了し、END_REPEAT の後でプログラムの実行が再開されます。

表 3-41　　REPEAT ステートメントの例

```
Index:= 1;
REPEAT
    Index:= Index + 2;
UNTIL Index > 50
END_REPEAT;
```

3.7.6 EXIT ステートメント

EXIT ステートメントは、終了条件が真か偽かに関係なく、任意のポイントでループ(FOR、WHILE、または REPEAT ループ)を終了させるために使用します。

このステートメントは、EXIT ステートメントを直接囲むループから直ちにジャンプする効果を持ちます。

ループが終了した後(たとえば、END_FOR の後)に、プログラムは再開します。

表 3-42 EXIT ステートメントの例

```
Index:= 1;
FOR Index := 1 to 51 BY 2 DO
    IF %I0.0 THEN
        EXIT;
    END_IF;
END_FOR;
// EXIT の実行後または FOR ループの通常の終了後に
// 以下の値割り付けを行う
// 実行の場合:
Index_find := Index_2;
```

3.7.7 RETURN ステートメント

RETURN ステートメントを使用すると、現在処理されている POU (プログラム、ファンクション、ファンクションブロック)が終了します。

ファンクションまたはファンクションブロックが終了すると、ファンクションまたはファンクションブロックを呼び出した位置の後の上位 POU でプログラムの実行が続行されます。

表 3-43 RETURN ステートメントの例

```
Index:= 1;
FOR Index := 1 to 51 BY 2 DO
    IF %I0.0 THEN
        RETURN;
    END_IF;
END_FOR;
// 実行の FOR ループの通常の終了後、
// ただし RETURN の実行後ではなく、
// 以下の値割り付けを行う:
Index_find := Index_2;
```

3.7.8 WAITFORCONDITION ステートメント

WAITFORCONDITION ステートメントは、プログラム可能なイベントまたは条件を待機するために MotionTask で使用することができます。このステートメントは、条件(式)が真になるまで、ステートメントを呼び出したタスクの実行を保留します。
WAITFORCONDITION に関する情報、およびこれに関連する式は、*『SIMOTION 基本機能』*機能マニュアルを参照してください。

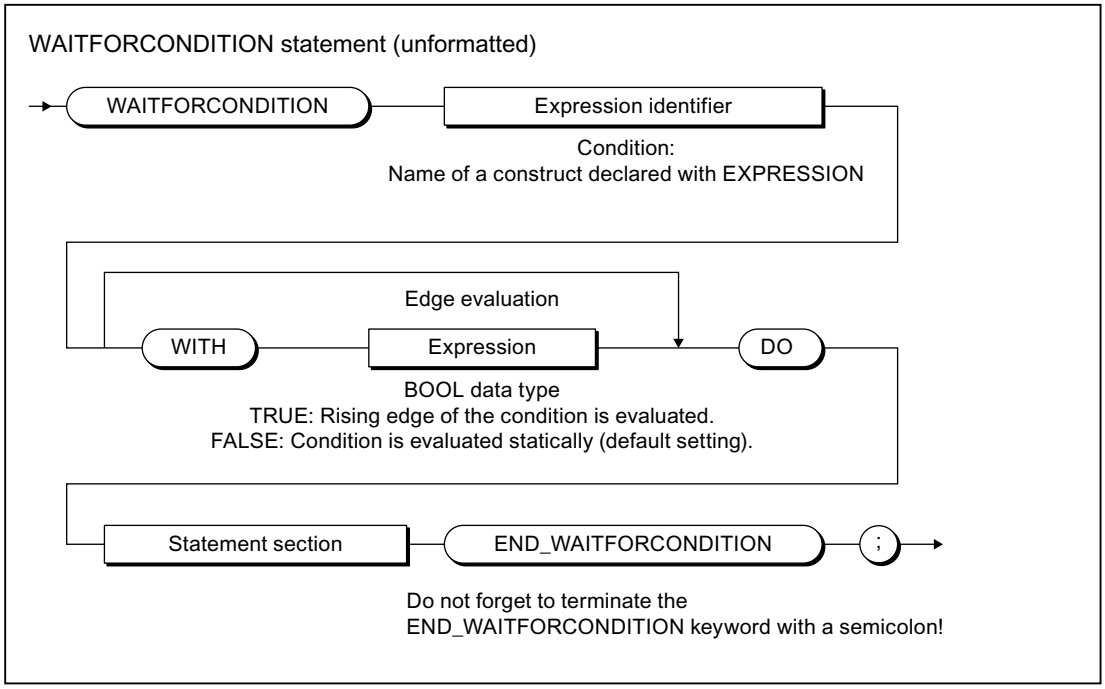


図 3-31　　構文: WAITFORCONDITION ステートメント

*式識別子*は EXPRESSION で宣言したコンストラクトです。その値は、条件が満たされたとみなすかどうかを(必要であれば *WITH エッジ評価*と共に)定義します。

*WITH エッジ評価*シーケンスはオプションです。*エッジ評価*は BOOL データタイプの式です。これは、*式識別子*の値を解釈する方法を決定します。

- *エッジ評価* = TRUE: *式識別子*の立ち上がりが解釈されます。すなわち、*式識別子*の値が FALSE から TRUE に**変化する**と、条件が満たされます。
- *エッジ評価* = FALSE: *式識別子*のスタティック値が解釈されます。すなわち、*式識別子*の値が TRUE **である**と、条件が満たされます。

*WITH エッジ評価*を指定しない場合、デフォルト設定は FALSE になります。すなわち、*式識別子*のスタティック値が評価されます。

立ち下がりをチェックするには、*NOT 式識別子*を使用することができます。

ステートメントセクションは、少なくとも 1 つのステートメント(空のステートメントも可)を含んでいる必要があります。

表 3-44　WAITFORCONDITION ステートメントの例

```
// ...
// 式の名前でステートメントを呼び出す
WAITFORCONDITION myExpression WITH TRUE DO
// ここでは、少なくとも 1 つのステートメントがより高い優先度で実行される。たとえば、
    %Q0.0 := TRUE;
END_WAITFORCONDITION;
// ...
```

完全な例は、*『SIMOTION 基本機能』*機能マニュアルを参照してください。

3.7.9　GOTO ステートメント

GOTOステートメントを使用すると、ステートメントで指定したジャンプラベルまでジャンプします(ジャンプステートメントおよびラベル (ページ 221)を参照)。

GOTO ステートメントを使用してジャンプステートメントをプログラミングし、ジャンプ先のジャンプラベルを指定します。ジャンプは POU 内でのみ使用できます。

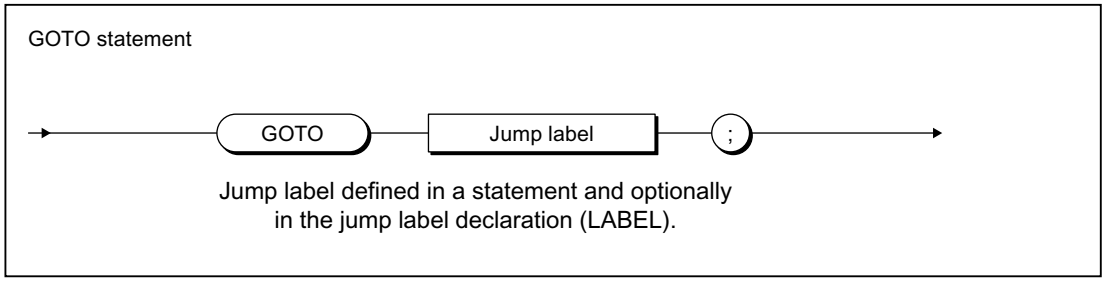


図 3-32　構文: GOTO ステートメント

注記

GOTO ステートメントは、特別な場合(たとえば、トラブルシューティングのためなど)にのみ使用してください。構造化プログラミングのルールによっては、絶対に使用しないでください。

ジャンプは POU 内でのみ使用できます。

以下のジャンプは不正です。

- 下位の制御構造(WHILE、FOR など)へのジャンプ
- WAITFORCONDITION 構造からのジャンプ
- CASE ステートメント内でのジャンプ

ジャンプラベルは、ジャンプラベルを使用する POU でのみ宣言できます。ジャンプラベルを宣言した場合、宣言したジャンプラベルだけを使用できます。

3.8 データタイプの変換

このセクションでは、基本データタイプ間を暗黙的および明示的に変換する方法を説明します。その他の変換の可能性についても概説します。

3.8.1 基本データタイプの変換

以下の表は、数値データタイプおよびビットデータタイプ間の変換の概要を示します。以下の 2 つの変換に区別されます。

- 暗黙的な変換: 1 つの式でさまざまなデータタイプを使用している場合、またはコンパイラによって値が割り付けられる場合、変換は自動的に行われます。
- 明示的な変換: ユーザが変換ファンクションを呼び出したときに変換が実行されます (*『SIMOTION 基本機能』*機能マニュアルを参照)。

表 3-45 数値データタイプおよびビットデータタイプのタイプ変換

ソースデータタイプ	ターゲットデータタイプ												
	BOOL	BYTE	WORD	DWORD	USINT	UINT	UDINT	SINT	INT	DINT	REAL	LREAL	STRING
BOOL	–	Im/Ex	Im/Ex	Im/Ex	Val	Val	Val	Val	Val	Val	Val	Val	–
BYTE	Ex	–	Im/Ex	Im/Ex	Ex	Ex	Ex	Ex	Ex	Ex	Val	Val	Elem
WORD	Ex	Ex	–	Im/Ex	Ex	Ex	Ex	Ex	Ex	Ex	Val	Val	–
DWORD	Ex	Ex	Ex	–	Ex	Ex	Ex	Ex	Ex	Ex	Ex/Val	Val	–
USINT	Val	Ex	Ex	Ex	–	Im/Ex	Im/Ex	Ex	Im/Ex	Im/Ex	Im/Ex	Im/Ex	–
UINT	Val	Ex	Ex	Ex	Ex	–	Im/Ex	Ex	Ex	Im/Ex	Im/Ex	Im/Ex	–
UDINT	Val	Ex	Ex	Ex	Ex	Ex	–	Ex	Ex	Ex	Ex	Ex	Ex
SINT	Val	Ex	Ex	Ex	Ex	Ex	Ex	–	Im/Ex	Im/Ex	Im/Ex	Im/Ex	–
INT	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	–	Im/Ex	Im/Ex	Im/Ex	–
DINT	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	Ex	–	Ex	Im/Ex	Ex
REAL	Val	Val	Val	Ex/Val	Ex	Ex	Ex	Ex	Ex	Ex	–	Im/Ex	Ex
LREAL	Val	Val	Val	Val	Ex	Ex	Ex	Ex	Ex	Ex	Ex	–	Ex
STRING	–	Elem	–	–	–	–	Ex	–	–	Ex	Ex	Ex	–

Im: 暗黙的なデータタイプ変換が可能
Ex: Quelldatentyp_TO_Zieldatentyp タイプ変換ファンクションを使用して明示的なデータタイプ変換が可能
Val:Quelldatentyp_VALUE_TO_Zieldatentyp タイプ変換ファンクションを使用して明示的なデータタイプ変換が可能
Elem: STRING データタイプの要素を使用した暗黙的なデータタイプ変換

日付および時刻データタイプの変換ファンクションに関する情報: *『SIMOTION 基本機能』*機能マニュアルを参照してください。

3.8.1.1 暗黙的なデータタイプ変換

たとえば REAL から LREAL、INT から REAL など、値の範囲を拡大しても値の損失が起こらない場合は常に暗黙的なデータタイプ変換が可能です。結果は常に定義されています。

以下の図は、すべての暗黙的なタイプ変換チェーンをグラフィックベースで表示したものです。タイプ変換チェーンの各段階(左から右、または上から下に読みます)は常に値の範囲の拡大を表します。

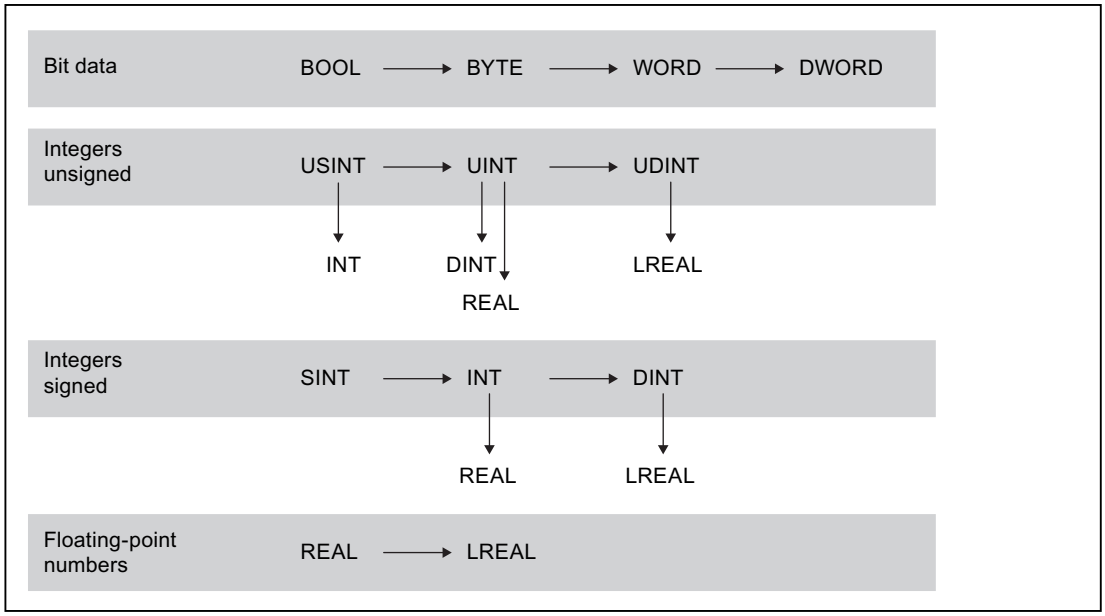


図 3-33 暗黙的なタイプ変換チェーン(左から右に 1 または複数レベル、あるいは上から下に 1 レベル)

以下の暗黙的なタイプ変換がサポートされています。

1. 1 つまたは複数のレベルにわたる横方向(左から右)の変換(たとえば、USINT から UDINT)
2. 1 つのレベルでの縦方向(上から下)の変換(たとえば、UINT から REAL)

暗黙的なタイプ変換は、以下の順序で結合することができます(たとえば、INT から LREAL)。

その他のすべてのタイプ変換は暗黙的には実行できません(たとえば、UDINT から REAL)。つまり、明示的なファンクションを使用する必要があります(*『SIMOTION 基本機能』*機能マニュアルを参照)。

注記

演算式では、結果は常に式に含まれる最多数のフォーマットで計算されます。

以下の場合にのみ式に値を割り付けることができます。

- 計算された式と割り付ける変数が同じデータタイプである。
- 計算された式のデータタイプを、割り付ける変数のデータタイプに暗黙的に変換できる。

このエラーソースとその解決方法の詳細: *『SIMOTION 基本機能』*機能マニュアルを参照してください。

表 3-46　　式および値割り付けのデータタイプの例

```
VAR
    usint_var : USINT;
    real_var : REAL;
byte_var : BYTE;
    string_var : STRING[80];
END_VAR

usint_var := 234 / 10;       // 式のデータタイプ: USINT
                             // 結果 = 23

real_var := 234 / 10;        // 暗黙的な変換が可能
                             // 結果 = 23

usint_var := 234 / SINT#10;  // 式のデータタイプ:INT
                             // 暗黙的な変換および
                             // 値割り付けは不可能

real_var := 234 / 10.0;      // 式のデータタイプ:REAL
                             // 結果 = 23.4

usint_var := 234 / 10.0;     // 暗黙的な変換および
                             // 値割り付けは不可能

byte_var := string_var[5];   // 暗黙的な変換が可能
                             // 値割り付けは不可能

string_var[10] := byte_var;  // 暗黙的な変換および
                             // 値割り付けは不可能
```

注記

適用可能な場合、数字にデータタイプを明示的に指定します(たとえば、数字 127 を USINT ではなく UINT データタイプにする場合、UINT#127)。

3.8.1.2 明示的なデータタイプ変換

明示的な変換は、LREAL から REAL に変換する場合のように、値の範囲が縮小されたり、精度が低下したりするなど、情報が失われる可能性がある場合に常に必要となります。

数値データタイプとビットデータタイプの変換ファンクションは、*『SIMOTION 基本機能』*機能マニュアルにリストされています。

コンパイラは、精度の損失に関わる変換を検出すると、警告を出力します。

重要

プログラムが稼働していると、タイプ変換によってエラーが引き起こされる可能性があります。これにより、タスク設定で設定したエラー応答がトリガされます(*『SIMOTION 基本機能』*機能マニュアルを参照)。

DWORD を REAL に変換するときは特別な注意が必要です。DWORD のビット文字列は REAL 値としてチェックなしで取得されます。DWORD のビット文字列が、IEEE に準拠した正規化浮動小数点数のビットパターンと一致していることを確認する必要があります。これを行うには、*_finite* ファンクションと*_isNaN* ファンクションを使用することができます。これを行うには、*_finite* ファンクションと*_isNaN* ファンクションを使用することができます。

文字列が一致していないと、REAL を算術演算に最初に使用したときに(たとえば、プログラム内で、またはシンボルブラウザで監視を行う際)、直ちにエラーがトリガされます(上記を参照)。

注記

変換中に値の範囲の制限を超過した場合、以下が当てはまります。

- アンダーフロー(LREAL 数の絶対値が最も小さい正の REAL 数より小さい): 結果は 0.0 です。
- オーバーフロー(LREAL 数の絶対値が最も大きい正の REAL 数より大きい):
  タスク設定中に指定したエラー応答がトリガされます。

3.8.2 その他の変換

ST のシステムファンクションでは、以下の変換を行うこともできます。

- **ビット文字列データタイプの結合**

  このファンクションは、ビット文字列データタイプの複数の変数を上位データタイプの 1 つの変数に結合します。

- **ビット文字列データタイプの分割**

  このファンクションブロックは、ビット文字列データタイプの 1 つの変数を上位データタイプの複数の変数に分割します。

- **任意のデータタイプとバイト配列間の変換**

  これは、通常、各種デバイス間のデータ交換のための定義済み送信フォーマットを作成するために使用します。

  詳細情報(バイト配列の配置に関する情報、アプリケーション事例集など): *『SIMOTION 基本機能』*機能マニュアルを参照してください。

- **テクノロジーオブジェクトデータタイプの変換**

  これは、階層的 TO データタイプ(driveAxis、posAxis、または followingAxis)の変数、あるいは一般的な ANYOBJECT タイプの変数を互換性のある TO データタイプに変換します。

アプリケーション事例集およびその他の情報: *『SIMOTION 基本機能』*機能マニュアルを参照してください。

ファンクション、ファンクションブロック、およびプログラム

4

4.1 ファンクション、ファンクションブロック、およびプログラム

この章では、ユーザ定義のファンクションおよびファンクションブロックの作成方法と呼び出し方法を説明します。システムでは、タイプ変換、三角法、およびビット文字列操作のための標準ファンクションが既に使用可能になっています。『*SIMOTION 基本機能*』機能マニュアルに、システムファンクションおよびテクノロジーオブジェクトのファンクション(TO ファンクション)の使用方法を記載しています。

ファンクション(FC)は、スタティックデータを含まない論理ブロックです。ファンクションを終了するとすべてのローカル変数がその値を失い、ファンクションを次回呼び出すと変数は再初期化されます。

ファンクションブロック (FB)は、スタティックデータを含むコードブロックです。FB はメモリを持っているため、いつでも、またユーザプログラムのどこからでも、その出力パラメータにアクセスすることができます。ローカル変数は、呼び出しと呼び出しの間にその値を保持します。

プログラムは FB に似ていますが、パラメータを持っていません。ただし、プログラムには実行レベルとタスクを割り付けることができます(『*SIMOTION 基本機能*』機能マニュアルを参照)。

FC と FB はパラメータを割り付けることができるカプセル化されたソースファイルセクションであるため、再利用できるという利点があります。

ファンクション、ファンクションブロック、およびプログラムはプログラムオーガニゼーションユニット(POU)です。すなわち、これらは実行可能なソースファイルセクションです。すべてのソースファイルセクションの概要は、ソースファイルセクションの使用 (ページ 151)を参照してください。

4.2 ファンクションおよびファンクションブロックの作成と呼び出し

以下の記述では、ファンクション(FC)およびファンクションブロック(FB)の作成方法と呼び出し方法を説明します。FCとFBの違いを示した完全な例は、ファンクションとファンクションブロックの比較 (ページ 145)を参照してください。

明記されたソースファイルセクションを定義し呼び出す順序は、ソースファイルセクションの使用 (ページ 151)で指定されています。

FCとFBのエクスポートとインポートの方法は、STソースファイル間のインポートとエクスポート (ページ 161)セクションで説明しています。

4.2.1 ファンクションの定義

ファンクションは、ファンクションを呼び出すソースファイル(プログラム、FB、または FC)のセクションの前にある実装セクションの宣言部分で定義します。

以下の構文を使用します。

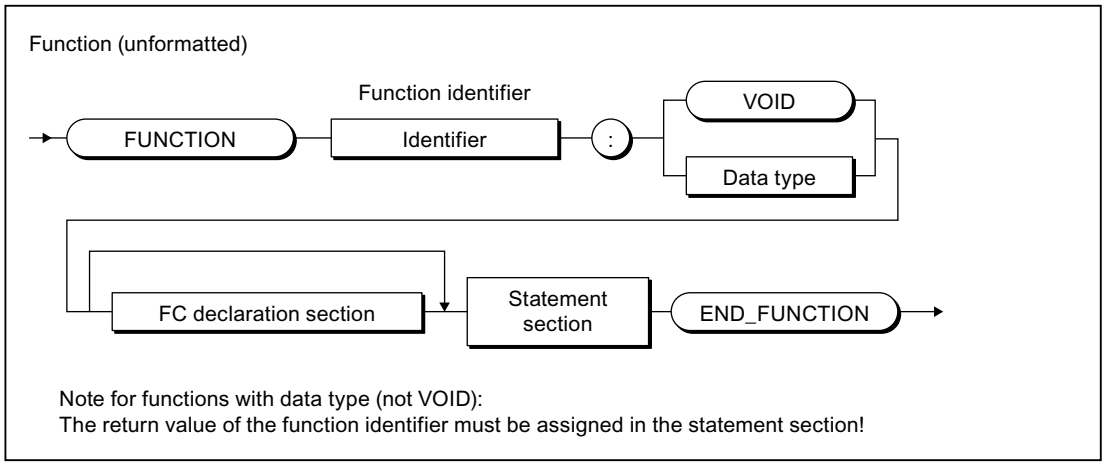


図 4-1 構文: ファンクション(FC)

FUNCTION キーワードの後に、FC 名としての識別子、戻り値のデータタイプと続けます。FC が戻り値を持たない場合は、データタイプとして VOID を入力します。

その後で、以下を入力します(コメント付きソースファイル (ページ 146)の例を参照)。

- 宣言セクション(オプション)
- ステートメントセクション
- END_FUNCTION キーワード

4.2.2 ファンクションブロックの定義

ファンクションブロックは、FB を呼び出すソースファイル(プログラム、FB、または FC)のセクションの前にある実装セクションの宣言部分で定義します。

以下の構文を使用します。

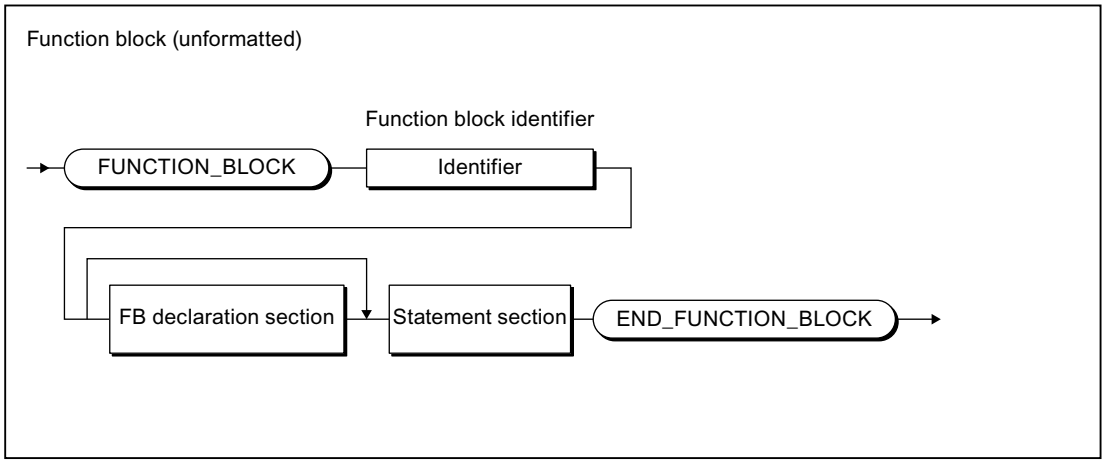


図 4-2 構文: ファンクションブロック(FB)

FUNCTION_BLOCK キーワードの後に FB 名としての識別子を入力します。

その後で、以下を入力します(コメント付きソースファイル (ページ 146)の例を参照)。

- 宣言セクション(オプション)
- ステートメントセクション
- END_FUNCTION キーワード

4.2.3 FB および FC の宣言セクション

宣言セクションは、それぞれ個別のキーワードペアによって識別されるさまざまな宣言ブロックに細分されます。各ブロックには、定数、ローカル変数、パラメータなど、類似したデータの宣言リストが含まれます。それぞれのタイプのブロックは 1 回だけ出現することができます。また、各ブロックは任意の順序で出現することができます。

FCおよびFBの宣言セクションでは、以下のオプションが使用可能です(コメント付きソースファイル (ページ 146)の例も参照)。

表 4-1　　FC および FB の宣言ブロック:オプション

データ	構文	FB	FC
定数	VAR CONSTANT *宣言リスト* END_VAR	X	X
入力パラメータ	VAR_INPUT *宣言リスト* END_VAR	X	X
入/出力パラメータ	VAR_IN_OUT *宣言リスト* END_VAR	X	X
出力パラメータ	VAR_OUTPUT *宣言リスト* END_VAR	X	–
ローカル変数 (FC および FB の場合)	VAR *宣言リスト* END_VAR	X (スタティック)	X (テンポラリ)
ローカル変数 (FB の場合)	VAR_TEMP *宣言リスト* END_VAR	X (テンポラリ)	-
宣言リスト: 宣言するタイプの識別子のリスト			

パラメータはローカルデータで、ファンクションブロックまたはファンクションの仮パラメータです。FB または FC を呼び出すと、仮パラメータが実パラメータによって置換されるため、呼び出されたソースファイルセクションと呼び出し側ソースファイルセクション間で情報を交換する手段が提供されます。

- 仮入力パラメータは、実入力値を受け取ります(データフロー内部)。
- 仮出力パラメータ(FB の場合のみ)は、出力値の転送に使用されます(データフロー外部)。
- 仮入/出力パラメータは、入力および出力パラメータとして機能します。

以下の図は、FB または FC のパラメータ宣言の構文を示します。

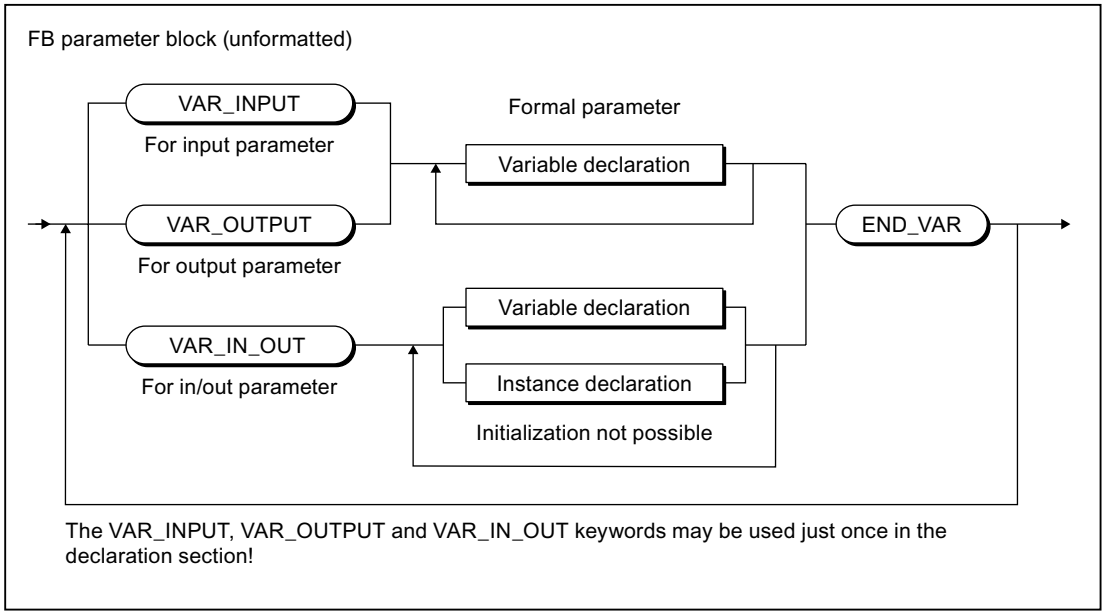


図 4-3　　構文: FB パラメータブロック

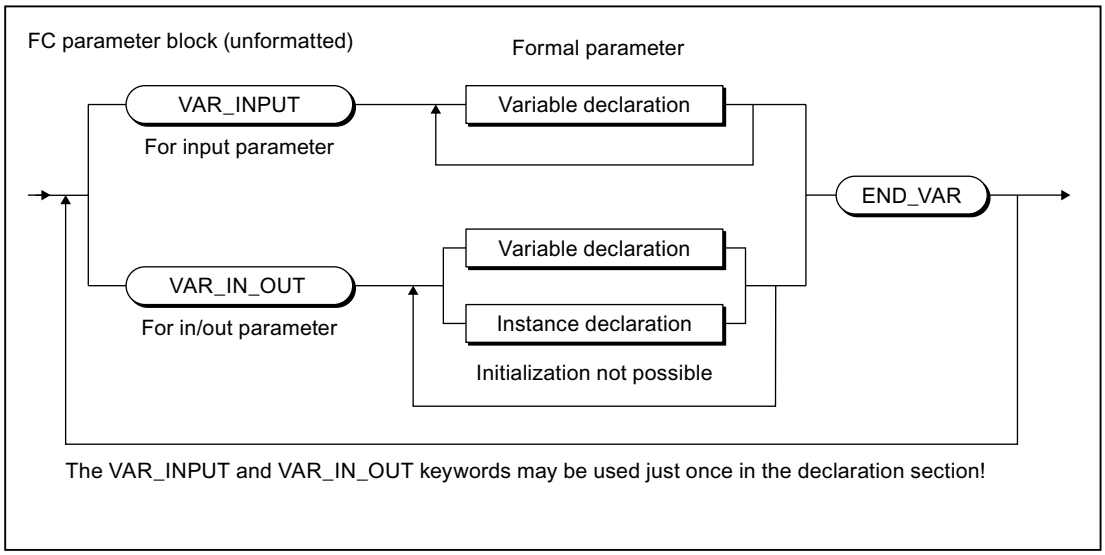


図 4-4　　構文: FC パラメータブロック

宣言したパラメータは、FB または FC 内の他の変数と同様に使用することができます。ただし、例外として、 入力パラメータに値を割り付けることはできません。

FB または FC の外部からは、以下にアクセスすることができます。

- 構造体変数を使用して、FBの入力および出力パラメータにアクセスすることができます(ユーザ定義データタイプ (ページ 77)を参照)。

  入力パラメータへのアクセスは、[Permit language extensions]コンパイラオプションが有効にされている場合のみ可能です(コンパイラのグローバル設定 (ページ 31)または STコンパイラのローカル設定 (ページ 32)を参照)。

  出力パラメータへのデータアクセスは標準として可能です。

- 式でファンクションを使用し、これをたとえば変数に割り付けることにより、FC の戻り値にアクセスすることができます(ファンクション名を指定すると、ファンクションが呼び出され、同時に結果が返されます)。

4.2.4 FB および FC のステートメントセクション

FC または FB のステートメントセクションには、FC または FB を呼び出すと実行されるステートメントが含まれます。ステートメントセクションを作成するための形式的なルールと比べて違いはありませんが、以下の表の情報に注意する必要があります。

注記

パラメータの効率的な使用に関するヒントは、*『SIMOTION 基本機能』*機能マニュアルの*「ランタイム最適化プログラミング」*を参照してください。

表 4-2　FC および FB でのパラメータと変数の使用

パラメータ/変数	使用
入力パラメータ	FC または FB の呼び出しを使用して、入力パラメータに現在の値を割り付けます。この値は FC または FB 内でデータ処理(計算など)に使用されますが、この値自体を変更することはできません。 [Permit language extensions]コンパイラオプションが有効にされている場合のみ(コンパイラのグローバル設定 (ページ 31)または STコンパイラのローカル設定 (ページ 32)を参照): 構造体変数を使用して、さらにFBの外部から(たとえば、呼び出し側ソースファイルセクションで)、FBの入力パラメータを読み書きすることができます。
入/出力パラメータ	FB または FC の呼び出しのために入/出力パラメータに変数を割り付けます。FC または FB はこの変数に直接アクセスし、それを直ちに変更することができます。タイプ変換はサポートされていません。 入/出力パラメータに割り付けた変数は、直接読み書きができる必要があります。したがって、システム変数(SIMOTION デバイスまたはテクノロジーオブジェクトの)、I/O 変数、またはプロセスイメージアクセスを入/出力パラメータに割り付けることはできません。
出力パラメータ (FB の場合のみ)	**=>**演算子を使用して、FB の呼び出しのために入/出力パラメータに変数を割り付けます。FB を閉じると、出力パラメータ(結果)の値が変数に転送されます。構造体変数を使用して、さらに FB の外部から(たとえば、呼び出し側ソースファイルセクションで)、FB の出力パラメータを読み取ることもできます。 ファンクション名は戻り値を受け取るため、FC は仮出力パラメータを持っていません。ファンクション名自体がある意味で出力パラメータになります。

パラメータ/変数	使用
ローカル変数	ローカル変数は、ブロック内でのみ宣言し、使用する変数です。 FC ではすべてのローカル変数(VAR ... END_VAR)が一時的です。すなわち、変数は FC が終了するとその値を失います。FC を次回呼び出すと、変数は再初期化されます。 FB では、スタティックローカル変数とテンポラリローカル変数が区別されます。 • スタティック変数(VAR ... END_VAR)は、FB が閉じられたときにその値を保持します。 • テンポラリ変数(VAR_TEMP ... END_VAR)は、FB を閉じると値を失います。FB を次回呼び出すと、変数は再初期化されます。 ブロックを呼び出すことによって直接、ローカル変数の値を問い合わせることはできません。これは、出力パラメータを使用してのみ可能です。

4.2.5 ファンクションおよびファンクションブロックの呼び出し

ここでは、ファンクションおよびファンクションブロックの呼び出しの概要を提供します。

4.2.5.1 パラメータ転送の原理

FC または FB を呼び出すと、呼び出し側ブロックと呼び出されたブロック間でデータ交換が行われます。転送するパラメータは、呼び出しにおけるパラメータリストとして指定する必要があります。パラメータは丸括弧で囲んで記述します。パラメータが複数の場合はコンマで区切ります。

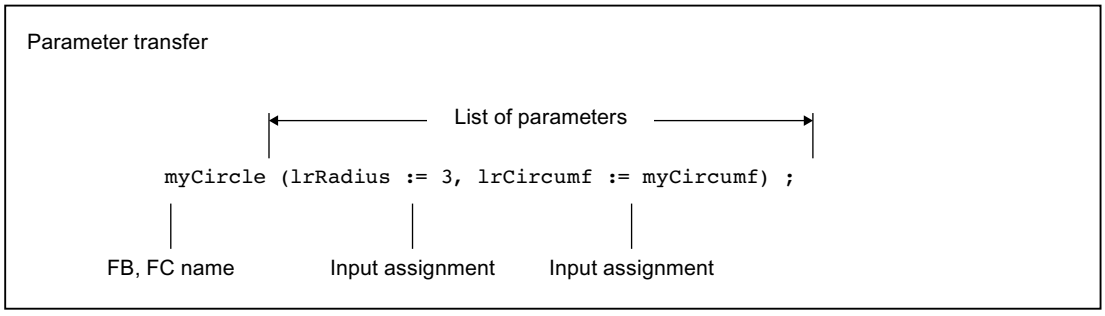


図 4-5 呼び出しのためのパラメータ転送の原理

入力パラメータおよび入/出力パラメータは、通常、値割り付けとして指定します。この方法で、呼び出したブロックの宣言セクションで定義したパラメータ(仮パラメータ)に値(実パラメータ)を割り付けます。

出力パラメータの割り付けは、**=>**演算子を使用して行います。この方法で、呼び出したブロックの宣言セクションで定義した出力パラメータ(仮パラメータ)に変数(実パラメータ)を割り付けます。

4.2.5.2 入力パラメータへのパラメータの転送

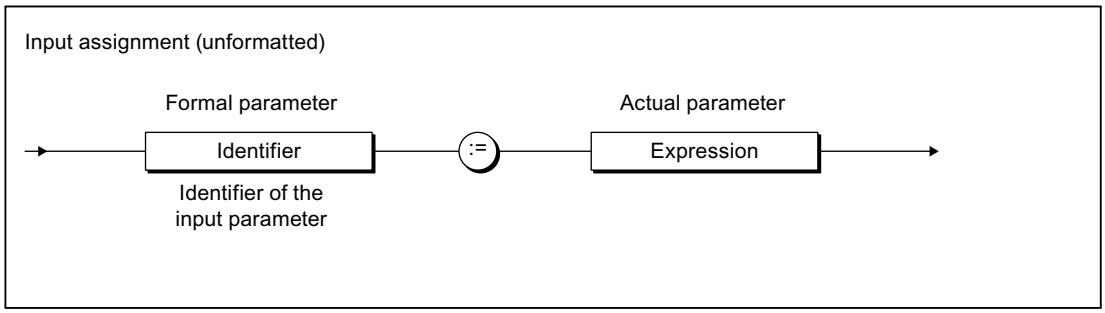


図 4-6 構文: 入力割り付け

入力割り付けを使用して、FB または FC の仮入力パラメータにデータ(実パラメータ)を転送します。実パラメータは式の形式で指定できます。FB または FC 内のステートメントで仮入力パラメータを使用することはできますが、パラメータの値を変更することはできません。

省略形式のパラメータ転送がサポートされていますが、ユーザ定義の FB と併せて適用はしないでください。この省略形式は、一部の FC でのみ必要となります。*『SIMOTION 基本機能』*機能マニュアルを参照してください。

FB では、実パラメータの割り付けはオプションです。入力割り付けを指定しない場合、FB はメモリを含むソースファイルセクションであるため、最後の呼び出しの値は保持されます。

FC では、仮パラメータの宣言に初期化式を指定した場合は実パラメータの割り付けはオプションです。

ファンクションの呼び出し (ページ 140)または ファンクションブロックの呼び出し(インスタンスの呼び出し) (ページ 141)の例も参照してください。

また、FBの入力パラメータは、FBの外部からいつでも読み書きすることができます。詳細情報: FBの外部でのFBの入力パラメータのアクセス (ページ 143).

4.2.5.3 入/出力パラメータへのパラメータの転送

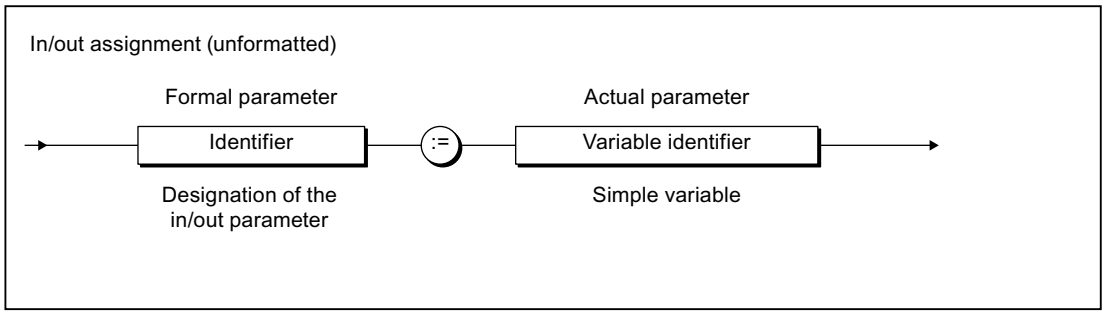


図 4-7 構文: 入力/出力割り付け

入力/出力割り付けを使用して、FB または FC の仮入/出力パラメータにデータ(実パラメータ)を転送します。仮入/出力パラメータには同じタイプの変数だけを割り付けることができ、データタイプ変換はできません。

仮入/出力パラメータは、FC または FB 内のステートメントで使用および変更することができます。FC または FB は実パラメータの変数に直接アクセスし、それを直ちに変更することができます。

ファンクションの呼び出し (ページ 140)または ファンクションブロックの呼び出し(インスタンスの呼び出し) (ページ 141)の例も参照してください。

入力/出力割り付けで STRING データタイプを使用するときは、実パラメータの宣言された長さが仮入/出力パラメータの長さより長いか等しい必要があります(以下の例を参照)。

表 4-3　　入力/出力割り付けにおける STRING データタイプの使用例

```
FUNCTION_BLOCK REF_STRING
    VAR_IN_OUT
        io : STRING[80];
    END_VAR
;    // ステートメント
END_FUNCTION_BLOCK

FUNCTION_BLOCK test
    VAR
        my_fb : REF_STRING;
        str1 : STRING[100];
        str2 : STRING[50];
    END_VAR
    my_fb(io := str1);    // 許可される呼び出し
    my_fb(io := str2);    // 許可されない呼び出し
                          // コンパイラエラーメッセージ

END_FUNCTION_BLOCK
```

入/出力パラメータに割り付けた変数は、直接読み書きができる必要があります。したがって、システム変数(SIMOTION デバイスまたはテクノロジーオブジェクトの)、I/O 変数、またはプロセスイメージアクセスを入/出力パラメータに割り付けることはできません。

パラメータのさまざまなアクセスタイムに注意してください!

4.2.5.4　出力パラメータへのパラメータの転送(FB の場合のみ)

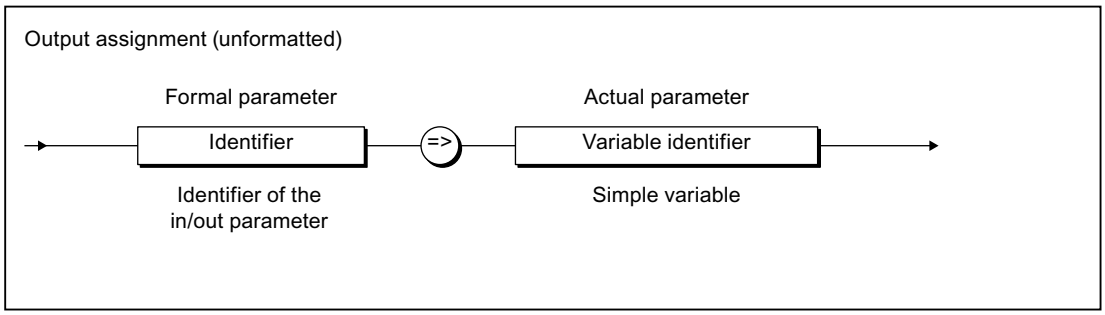


図 4-8　　構文: 出力割り付け

FB を閉じたときに FB の仮出力パラメータを仮出力パラメータの値を受け取る変数(実パラメータ)に割り付けるには、出力割り付けを使用します。

仮出力パラメータは、FB 内のステートメントで使用および変更することができます。

ファンクションの呼び出し (ページ 140)または ファンクションブロックの呼び出し(インスタンスの呼び出し) (ページ 141)の例も参照してください。

パラメータの転送では、出力割り付けはオプションです。FBの出力パラメータは、いつでも、またFBの外部でも読み書き可能です。詳細情報: FBの外部でのFBの出力パラメータのアクセス (ページ 143)).

4.2.5.5 パラメータのアクセスタイム

アクセスのタイプ、したがってパラメータのアクセスタイムはさまざまです。

- 入力割り付けの場合、実パラメータの値が仮パラメータにコピーされます。配列などの大きい構造がコピーされ、FC または FB が頻繁に呼び出される場合、これによりパフォーマンスが制限される可能性があります。
- 入力/出力割り付けでは値はコピーされません。この場合、仮パラメータのメモリアドレスと実パラメータのメモリアドレスの間にリンクが確立されます。したがって、変数の転送は入力割り付けより高速です(特に、大容量のデータが関係する場合)。ただし、FB から変数へのアクセスは入力割り付けより低速になります。
- ユニット変数を使用する場合、この変数はSTソースファイル全体で有効なため、ファンクションまたはファンクションブロックには何もコピーされません(変数モデル (ページ 167)を参照)。

注記

入力パラメータの代わりに入/出力パラメータを使用した方が高速なのは、ファンクションブロックに大容量のデータが渡される場合だけです。

パラメータではなく主にユニット変数を使用する場合、結果のプログラム構造が複雑で分かりにくくなります。オブジェクト指向、データのカプセル化、変数名の複数の使用(有効性の範囲のカプセル化)などは使用できなくなります。

4.2.5.6 ファンクションの呼び出し

ファンクションは次のように呼び出します。

- 戻り値を持つファンクション(VOID 以外のデータタイプ)

  値割り付けの右側にファンクションを配置します。ファンクションは式内のオペランドとして出現することもできます。ファンクションを呼び出すと、該当するポイントでその戻り値が使用されて式が計算されます。

  例

```
y:=sin(x);
y := sin(in := x);
y := sqrt (1 - cos(x) * cos(x));
```

- 戻り値を持たないファンクション(VOID データタイプ)

  割り付けはファンクション呼び出しのみで構成されます。

  in1 および in2 入力パラメータと inout 入/出力パラメータを持つ funct1 ファンクションが既に定義されている場合、以下の例は有効です。

  例:

```
funct1 (in1 := var11, in2 := var12, inout1 := var13);
```

注記

ファンクション自体では、結果(戻り値)はファンクション名に割り付けられます(VOID データタイプ以外)。

4.2.5.7 ファンクションブロックの呼び出し(インスタンスの呼び出し)

ファンクションブロック(FB)を呼び出す前に、インスタンスを宣言する必要があります。変数を宣言し、データタイプとしてファンクションブロックの名前を入力します。このインスタンスは以下のように宣言します。

- ローカルに宣言(POU の宣言セクションの VAR / END_VAR 内)
- グローバルに宣言(実装セクションのインターフェースの VAR_GLOBAL / END_VAR 内)
- 入/出力パラメータとして宣言(ファンクションブロックまたはファンクションの宣言セクションの VAR_IN_OUT / END_VAR 内)

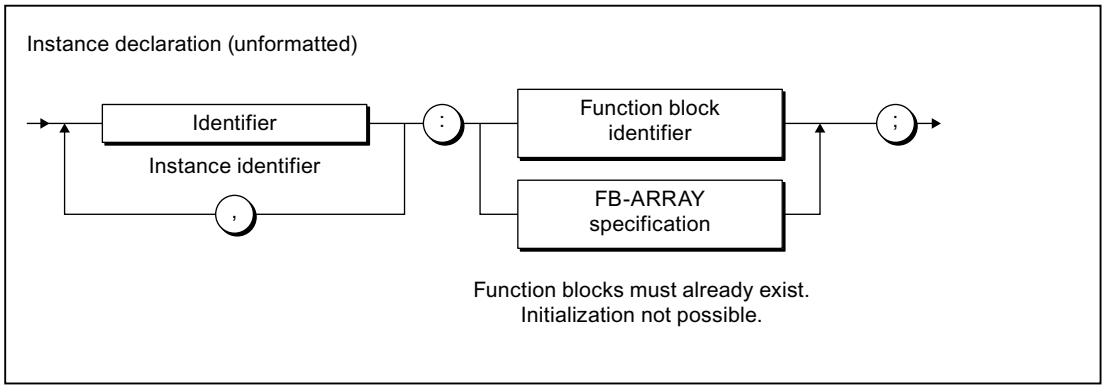


図 4-9 構文: インスタンスの宣言

インスタンスの宣言は、たとえば次のように配列とすることも可能です。

```
FB_inst : ARRAY [1..2] OF FB_name.
```

注記

変数のタイプによって初期化時間が異なることに注意してください。

ファンクションブロックのインスタンスは POU のステートメントセクションで呼び出します(構文については、図を参照してください)。FB パラメータは、コンマで区切られた、入力および入力/出力割り付けです。

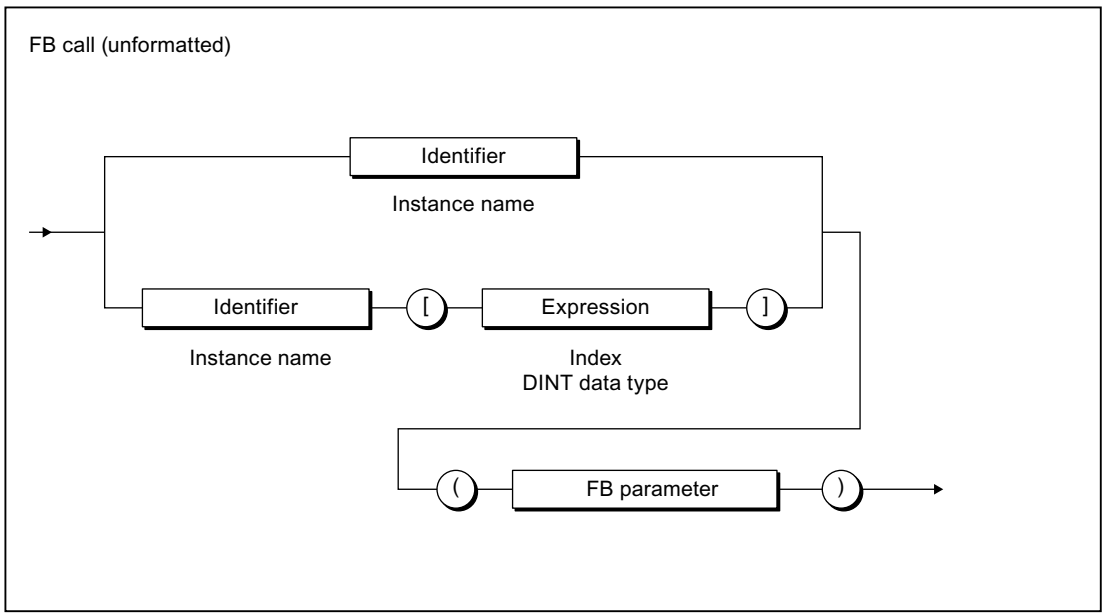


図 4-10　　FB の呼び出し構文

*電源およびモータ*ファンクションブロックが既に定義されているとすると、以下の表の例を適用できます。

- FB 電源:
  入力パラメータ in1、in2; 入/出力パラメータ inout; 出力パラメータ out
- FB モータ:
  入/出力パラメータ inout1、inout2; 出力パラメータ out1、out2

表 4-4　　インスタンスの宣言、FB の呼び出し、および出力パラメータへのアクセスの例

```
VAR
    Supply1, Supply2: Supply;
    Motor1 : Motor;
END_VAR

Supply1 (in1 := var11, in2 := expr12, inout := var13, out => var14) ;
Supply2 (in1 := var21, in2 := expr22, inout := var23, out => var24) ;
Motor1 (inout1 := var31, inout2 := var32, out1 => var33, out2 => var34);
// ...
var15 := PowerSupply1.out;
var25 := PowerSupply2.out;
var35 := Motor1.out1;
var36 := Motor1.out2;
var41 := Motor1.out1 * Motor1.out2 * (Supply1.out + Supply2.out);
```

4.2.5.8 FB の外部での FB の出力パラメータのアクセス

FBの呼び出しのための 出力割り付け (ページ 139)に加えて、FBの外部でFBの出力パラメータにアクセスすることが常に可能です。

これを行うには、*FBインスタンス名.出力パラメータ*形式の 構造体変数 (ページ 83)(たとえば、*Supply1.out*)を使用します。

値割り付けでは FB 自体のインスタンス名は使用しないでください!

下記も参照

ユーザ定義データタイプ (ページ 77)

4.2.5.9 FB の外部での FB の入力パラメータのアクセス

FBの呼び出しのための 入力割り付け (ページ 138)に加えて、FBの外部でFBの入力パラメータを読み書きすることが常に可能です。

これを行うには、*FBインスタンス名.入力パラメータ*形式の 構造体変数 (ページ 83)(たとえば、*Supply1.in1*)を使用します。

通知
このオプションを使用できるためには、[Permit language extensions]コンパイラオプション(コンパイラのグローバル設定 (ページ 31)および STコンパイラのローカル設定 (ページ 32)を参照)が有効にされている必要があります。

値割り付けでは FB 自体のインスタンス名は使用しないでください!

表 4-5　　入力パラメータへの割り付けの例

```
// [Permit language extensions]コンパイラオプションが有効にされている場合のみ
VAR
    var_fb   : _WORD_TO_2BYTE;
    var_word : WORD;
END_VAR
var_fb.wordin := var_word;
// ..
var_fb();
```

4.2.5.10 FB の呼び出しにおけるエラーソース

ファンクションブロックのインスタンスを呼び出すときは、以下の点に注意してください。

- **メモリに直接格納される変数を持つ入/出力パラメータだけを割り付けます。**

  以下の変数だけが実パラメータとして許容されます。

  – グローバル変数(ユニット変数およびグローバルデバイスユーザ変数)

  – ローカル変数

  – TO データタイプの変数(TO インスタンス)

  特に以下を使用できます。

  – システム変数(TO 変数)

  – エンジニアリングシステムのテクノロジーオブジェクトの名前

  – I/O 変数

  – 絶対プロセスイメージアクセスおよびシンボリックプロセスイメージアクセス

- **入/出力パラメータとしてファンクション(FC)は使用しないでください。**

  入力/出力割り付けで、FC の戻り値、すなわち FC の呼び出しを実パラメータとすることはできません。まずローカル変数に FC の結果を格納し、その後でこの変数を入力/出力割り付けで実パラメータとして使用する必要があります。

- **入/出力パラメータとして定数は使用しないでください。**

  値は書き戻されるため、入力/出力割り付けの実パラメータとしては変数だけを使用できます。

- **入/出力パラメータを初期化することはできません。**

4.3 ファンクションとファンクションブロックの比較

ユーザ定義のファンクションブロック(FB)とファンクションの違いについて、完全な例を使用して以下に簡潔に示します。

4.3.1 例の説明

以下の例は、FBとFCの違いを示します。簡単にするため、パラメータの各タイプを 1 回だけ使用しています。ただし実際には、パラメータは何回でも定義できます。使用されている用語は、ファンクションの定義 (ページ 132)および ファンクションブロックの定義 (ページ 133)の詳細な説明で定義されています。

半径入力変数に対して円の円周と面積を計算する際に使用するため、実装セクションの宣言部分で FB および FC としてブロックを作成します。

- 半径に入力パラメータを定義します。
- 円の円周に入/出力パラメータを定義します。すなわち、FB または FC の呼び出し中、転送された変数の値が直接割り付けられます。
- FB および FC に円の面積を定義するにはいくつかの方法があります。
  - FB の場合、出力パラメータを定義します。
  - FC の場合、その戻り値を使用します。戻り値のデータタイプを適切に定義します。
- FB および FC の各呼び出しはカウンタ(ローカル変数)に記録されます。例の状態についての説明: この値は FB でのみカウントが継続されます。
- プログラムセクションでは、FB または FC を呼び出し、以下の仮パラメータに実パラメータを割り付けます。
  - FB の場合: 入力、入力/出力、および出力パラメータ
  - FC の場合: 入力および入/出力パラメータ

  FB または FC を呼び出した後は、以下で円周と面積の値を使用できます。
  - FB の場合: 入力/出力および出力パラメータの実パラメータ。

    出力パラメータは FB の外部でも読み取ることができます。
  - FC の場合: ファンクションの戻り値、および入/出力パラメータの実パラメータ。

4.3.2 コメント付きソースファイル

表 4-6 FB と FC の違いの例

ファクションブロック(FB)

```
INTERFACE
  PROGRAM CircleCalc1;
END_INTERFACE
IMPLEMENTATION
  FUNCTION_BLOCK Circle1
    // 定数宣言
    VAR CONSTANT
      PI : LREAL := 3.1415 ;
    END_VAR
    // 入力パラメータ
    VAR_INPUT
      半径: LREAL;
    END_VAR
    // 入/出力パラメータ
    VAR_IN_OUT
      circumference : LREAL;
    END_VAR
    // 出力パラメータ
    VAR_OUTPUT
      Area : LREAL;
    END_VAR
    // ローカル変数、スタティック
    VAR
      Counter : DINT;
      (* 呼び出しと呼び出しの間に
      変数はその値を保持する *)
    END_VAR
    // カウンタの呼び出し
    Counter := counter + 1 ;
    Circumference := 2 * PI * Radius ;
    Area := PI * Radius**2 ;
  END_FUNCTION_BLOCK
  PROGRAM CircleCalc1
    VAR
      myCircle1        : Circle1 ;
      myArea1, myArea2 : LREAL;
      myCircf          : LREAL;
    END_VAR;
    myCircle1(Radius := 3
      , Circumference := myCircf
      , Area => myArea1) ;
    myArea2 := myCircle1.Area ;
    // myCircf の値は 18,849
    // myArea1 の値は 28,274
    // myArea2 の値は 28,274
  END_PROGRAM
END_IMPLEMENTATION
```

ファンクション(FC)

```
INTERFACE
  PROGRAM CircleCalc2;
END_INTERFACE
IMPLEMENTATION
  FUNCTION Circle2 :LREAL
    //定数の宣言
    VAR CONSTANT
      PI : LREAL := 3.1415 ;
    END_VAR
    // 入力パラメータ
    VAR_INPUT
      半径: LREAL;
    END_VAR
    //入/出力パラメータ
    VAR_IN_OUT
      circumference : LREAL;
    END_VAR
    // 出力パラメータ
    // 不可能

    // ローカル変数、テンポラリ
    VAR
      Counter : DINT;
      (* 呼び出しのたびに変数が
      0 で初期化される *)
    END_VAR
    // カウンタの呼び出し
    Counter := Counter + 1 ;
    Circumference := 2 * PI * Radius ;
    Circle2 := PI * Radius**2 ;
  END_FUNCTION
  PROGRAM CircleCalc2
    VAR

      myArea  : LREAL;
      myCircf          : LREAL;
    END_VAR;
    myArea := Circle2(Radius := 3
      , Circumference := myCircf);


    // myCircf の値は 18,849
    // myArea の値は 28,274

  END_PROGRAM
END_IMPLEMENTATION
```

表 4-7　　前の例の FB と FC の違いの例

ファクションブロック(FB)	ファンクション(FC)
説明	
定義用に予約された語 FUNCTION_BLOCK および END_FUNCTION_BLOCK	定義用に予約された語 FUNCTION および END_FUNCTION
戻り値は使用できません。	名前の後に戻り値のデータタイプを指定する必要があります(戻り値がない場合は VOID データタイプ)。
入力パラメータを使用して FB に値を転送することができます。	入力パラメータを使用して FC に値を転送することができます。
入/出力パラメータを使用して FB の転送された変数を読み書きすることができます。	入/出力パラメータを使用して FC の転送された変数を読み書きすることができます。
出力パラメータを使用して FB から値を返すことができます。	出力パラメータは使用できません。
ローカル変数はスタティックです。すなわち、この変数は FB の呼び出しと呼び出しの間にその値を保持します。 *Counter* ローカル変数は増分され、FB の終了時にその値が保持されます。したがって、この変数は FB を呼び出すたびに増分されます。 この動作を確認する方法: ローカル変数の値を FB のグローバル変数に割り付けます。FB の呼び出しを何度か繰り返した後でグローバル変数の値を監視します。	ローカル変数は一時的です。すなわち、この変数はファンクションが終了するとその値を失います。 *Counter* ローカル変数は増分されますが、FC の終了時にその値が失われます。変数は FC の次回呼び出し時に初期化されます(例では 0 に初期化されます)。 この動作を確認する方法: ローカル変数の値を FC のグローバル変数に割り付けます。FC の呼び出しを何度か繰り返した後でも、グローバル変数の値は変更されないままです。
ステートメントセクションでは、出力または入/出力パラメータに結果(戻り値)を割り付けます。	ステートメントセクションでは、ファンクション名に結果(戻り値)を割り付けます(VOID データタイプを指定した場合を除く)。
呼び出しを実行するブロックの宣言セクションでは、FB のインスタンスを宣言します。変数を宣言し、そのデータタイプとして FB の名前を指定します。宣言したインスタンス名を使用して、FB を呼び出し、その出力パラメータにアクセスします。 ステートメントセクションでは FB 自体の名前は使用しないでください。	
• FB インスタンスを呼び出すときに、入/出力パラメータに変数を割り付けます。 • 呼び出しを使用すると、変数に出力パラメータを割り付けることができます。 • FB の外部であっても、FB の出力パラメータを読み取ることができます。このためには、次のフォーマットの構造体変数を使用します: *FB-instancename.outputparameter*。	• FB インスタンスを呼び出すときに、入/出力パラメータに変数を割り付けます。 • FC の戻り値を取得するには、 – 変数にファンクションを割り付けます。 – 値割り付けの右側の式でファンクションを使用します。
呼び出しを実行するプログラムから FB の入力/出力変数および出力パラメータ以外の変数にアクセスすることはできません。 例外: [Permit language extensions]コンパイラオプションが有効にされている場合(コンパイラのグローバル設定 (ページ 31)または STコンパイラのローカル設定 (ページ 32)を参照)、呼び出されたプログラムはFBの入力パラメータにもアクセスできます。このためには、次のフォーマットの構造体変数を使用します: *FB-instancename.inputparameter*。	呼び出しを実行するプログラムから戻り値以外の変数にアクセスすることはできません。

4.4 プログラム

プログラム、すなわちキーワード PROGRAM と END_PROGRAM の間のステートメントは、FB と似ています。たとえば、スタティックローカル変数(VAR...END_VAR)またはテンポラリローカル変数(VAR_TEMP...END_VAR)を作成することができます。

FBまたはFCとは対照的に、プログラムはSIMOTION SCOUTのタスクまたは実行レベルに割り付けることができます。タスクの詳細については、SIMOTIONへのSTの統合 (ページ 151)および『SIMOTION基本機能』機能マニュアルを参照してください。

プログラムをパラメータで呼び出すことはできません。したがって、FBおよびFCと異なり、プログラムは仮パラメータを持ちません。ただし、FBおよびFCと同様、プログラムはSTソースファイルの一部です。例は コメント付きソースファイル (ページ 146)の表「FBとFCの違いの例」に、ソースファイルセクションに関するその他の情報は ソースファイルセクション (ページ 151)に示しています。

プログラムは、異なるプログラムまたはファンクションブロック内で呼び出すこともできます。この場合、以下のコンパイラオプションを有効にする必要があります(コンパイラのグローバル設定 (ページ 31)および STコンパイラのローカル設定 (ページ 32)を参照)。

1. 呼び出し側プログラムまたはファンクションブロックのプログラムソースの[Permit language extensions]、および
2. 呼び出し側プログラムのプログラムソースの[Create program instance data only once]

パラメータと戻り値を持つファンクションの場合のように呼び出しを実行します。以下の例を参照してください。

表 4-8　　プログラムにおけるプログラムの呼び出しの例

```
PROGRAM my_prog
    ; // ...
END_PROGRAM

PROGRAM main_prog
    ; // ...
    my_prog();
    ; // ...
END_PROGRAM
```

4.5 式

式は、ファンクション宣言の特殊なケースです。

- 戻り値のデータタイプを BOOL として定義し、明示的には指定しません。

式は、WAITFORCONDITIONステートメントと一緒に使用します(WAITFORCONDITIONステートメント (ページ 123)を参照)。

式は、ST ソースファイルの実装セクションでのみ宣言できます。

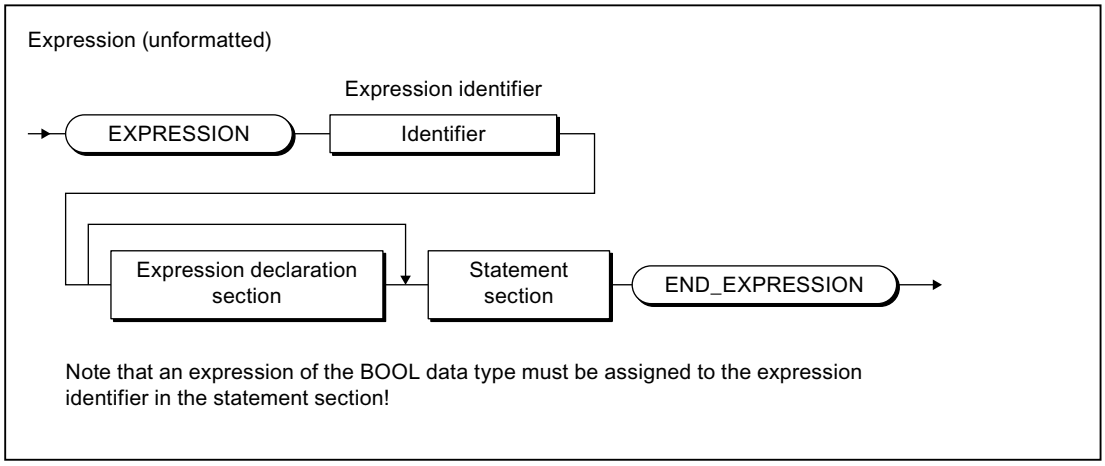


図 4-11 構文:式

オプションで、宣言セクションにローカル(テンポラリ)変数を宣言することができます。その他の宣言(たとえば、入力パラメータや定数の宣言)はできません。

ステートメントセクションでは、以下にアクセスすることができます。

- 式のローカル変数
- ユニット変数
- グローバルデバイス変数、I/O 変数、およびプロセスイメージ

式のステートメントセクションで、式の名前に BOOL データタイプの式を割り付ける必要があります(図を参照)。

注記

式のステートメントセクションにファンクション呼び出しやループを含めることはできません。

SIMOTION への ST の統合 5

このセクションでは、ST プログラムと SIMOTION SCOUT の相互運用性について説明します。

5.1 ソースファイルセクション

ソースファイルセクションの意味の概要は STソースファイルの構造 (ページ 69)で説明しました。このセクションでは、セクションの構文や、セクションを使用して複数のSTソースファイル間でデータをインポートおよびエクスポートする方法などの詳細を説明します。

5.1.1 ソースファイルセクションの使用

ST ソースファイルをコンパイルできるよう、ソースファイルセクション(モジュール)の特定の構造ルールと構文ルールに従う必要があります。ここでは、いくつかの一般的なガイドラインを示します。ソースファイルセクションに関する詳細は、このセクションの後の方で説明します。

- ソースファイルを作成するときは、常にソースファイルセクションの順序に注意を払う必要があります。呼び出されるセクションが必ず呼び出し側セクションの前になければなりません。そうでないと、呼び出されるセクションは呼び出し側セクションを認識しません。

  たとえば、変数は常にそれが使用される前に宣言され、また、ファンクションはそれが呼び出される前に定義されている必要があります。

- 最も一般的なソースファイルセクションのソーステキスト(プログラム、ファンクション、またはファンクションブロック)は、以下で構成されます。
  - 予約語および識別子によるセクションの開始
  - 宣言セクション(オプション)
  - ステートメントセクション
  - 予約語によるセクションの終了
- ソースファイルセクションの識別子(以下、*name*または*name_list*)は、識別子の一般的な構文ルール(STの識別子 (ページ 57))に従います。

注記

すべての可能なソースファイルセクションを含むテンプレートをオンラインヘルプから入手することができます。

5.1.1.1 インターフェースセクション

インターフェースセクションには、データ(データタイプ、変数、ファンクションブロック、ファンクション、およびプログラム)をインポートおよびエクスポートするためのステートメントが含まれます。テクノロジーパッケージとライブラリもダウンロードできます。

インターフェースセクションは以下の構文を持っています。

表 5-1 インターフェースセクションの構文

構文	INTERFACE // インターフェースステートメント(オプション) END_INTERFACE セクションの個々の識別子を指定することはできません。 オプションで、予約語 INTERFACE と END_INTERFACE の間に以下の順序でインターフェースステートメントが存在します。 1. 使用しているテクノロジーパッケージの指定。構文: USEPACKAGE tp-name [AS namespace]; 詳細については、*『SIMOTION 基本機能』*機能マニュアルを参照してください。 2. 使用しているライブラリの指定。構文: USELIB library-name-list [AS namespace]; 詳細については、ライブラリのデータタイプ、ファンクション、およびファンクションブロックの使用 (ページ 207)を参照してください。 3. 他のユニットのエクスポートされたコンポーネントを使用するための、他のユニットへの参照。 構文: USES unit_name-list; 詳細については、STソースファイル間のインポートとエクスポート (ページ 161)を参照してください。 4. エクスポートのための宣言および指定 – データタイプ定義 (ページ 158) ST ソースファイル全体で有効で、エクスポートされるユーザ定義データタイプ – 変数宣言 (ページ 159) ST ソースファイル全体で有効で、エクスポートされるユニット変数およびユニット定数 使用できるキーワード: 変数宣言 (ページ 159)の表*「変数宣言の構文」*の*「構文」*を参照してください。 – エクスポートするプログラムオーガニゼーションユニット(POU)に関する情報。構文: FUNCTION fc_name; FUNCTION_BLOCK fb_name; PROGRAM program_name;
シーケンス	インターフェースセクションは、ST ソースファイルの最初のセクションです [1]。 インターフェースステートメント 1 ~ 4 の順序は固定されています。 番号 4 の中では、任意の順序を使用できます。データタイプ定義および変数定義の個々の宣言ブロックは、複数回出現することができます。 注意点: 識別子は、それが使用される前に宣言されている必要があります。
頻度	ST ソースファイルごとに 1 回
必須セクション	あり
[1] オプションで、インターフェースセクションの前にユニットステートメントを置くことが可能(STソースファイル間のインポートとエクスポート (ページ 161)を参照)	

データをインポートおよびエクスポートするためのインターフェースソースファイルセクションの使用の詳細については、STソースファイル間のインポートとエクスポート (ページ 161)を参照してください。

5.1.1.2 実装セクション

実装セクションには、STソースファイルの主要部分から成る実行可能セクションが含まれます。STソースファイル間のインポートとエクスポート (ページ 161)

実装セクションは以下の構文を持っています。

表 5-2 実装セクションの構文

構文	IMPLEMENTATION // 実装ステートメント(オプション) END_IMPLEMENTATION セクションの個々の識別子を指定することはできません。 オプションで、予約語 IMPLEMENTATION と END_IMPLEMENTATION の間に以下の順序で実装ステートメント(ST ソースファイルの主要部分)が存在します。 1. 他のユニットのエクスポートされたコンポーネントを使用するための、他のユニットへの参照。構文: USES unit_name-list; 詳細については、STソースファイル間のインポートとエクスポート (ページ 161)を参照してください。 2. 宣言 – データタイプ定義 (ページ 158) ST ソースファイル全体で有効なユーザ定義データタイプ(UDT) – 変数宣言 (ページ 159) ST ソースファイル全体で有効なユニット変数およびユニット定数 使用できるキーワード: 変数宣言 (ページ 159)の表 *「変数宣言の構文」*の「構文」を参照してください。 3. プログラムオーガニゼーションユニット(POU) (ページ 154)
シーケンス	常にインターフェースセクションの後に続きます。 上記の実装ステートメントの順序は強制的です。番号 2 と 3 の中では、任意の順序を使用できます。 注意点: 識別子は、それが使用される前に宣言されている必要があります。
頻度	ST ソースファイルごとに 1 回
必須セクション	あり

5.1.1.3 プログラムオーガニゼーションユニット(POU)

POU は、実行可能なソースファイルセクションです。

- ファンクション(FC) (ページ 154)
- 式 (ページ 155)
- ファンクションブロック(FB) (ページ 156)
- プログラム (ページ 157)

注記

呼び出された POU は、呼び出し側 POU によって認識されるよう、常に呼び出し側 POU の前にあります。

5.1.1.4 ファンクション(FC)

FC はプログラムオーガニゼーションユニット(POU)として分類されます。ファンクションは、プログラムおよびファンクションブロックから呼び出すことができるデータタイプを含む、パラメータ化されたソースファイルセクションです。ファンクションを終了するとすべての内部変数がその値を失い、ファンクションを次回呼び出すと変数は再初期化されます。

FC は以下の構文を持っています。

表 5-3 ファンクション(FC)の構文

構文	`FUNCTION name : function_data_type`
	`// 宣言セクション`
	`// ステートメントセクション`
	`END_FUNCTION`
	name はファンクションの識別子を表し、*function_data_type* は戻り値のデータタイプを表します。
	宣言セクションで変数の宣言に使用できるキーワード: 変数宣言 (ページ 159)の表*「変数宣言の構文」*の「構文」を参照してください。
	function_data_type <> VOID のファンクションについての注意点: ステートメントセクションで、ファンクションの識別子に *function_data_type* データタイプの式を割り付ける必要があります!

シーケンス	FC は実装セクションでのみ定義できます。 順序についての注意点: FC は FC を呼び出す POU の前にある必要があります! 宣言セクション (ページ 157)は ステートメントセクション (ページ 158)の前にある必要があります。
頻度	ST ソースファイルごとに任意の回数
必須セクション	なし

FCの詳細については、ファンクションおよびファンクションブロックの作成と呼び出し (ページ 131)を参照してください。

5.1.1.5 式

式は、指定した BOOL データタイプの戻り値を持つ、ファンクション宣言の特殊なケースです。ファンクション名に割り付けられた予約語 EXPRESSION <式識別子> ... END_EXPRESSION 内の式が評価されます。

WAITFORCONDITION コンストラクトを使用すると、プログラム可能なイベントまたは条件を MotionTask で直接待機することができます。ステートメントは、条件(式)が真になるまで、ステートメントを呼び出したタスクを保留します。

式は以下の構文を持っています。

表 5-4　式の構文

構文	EXPRESSION name
	// 宣言セクション
	// ステートメントセクション
	END_EXPRESSION
	name は、式識別子を表します。
	宣言セクションで変数の宣言に使用できるキーワード: 変数宣言 (ページ 159)の表*「変数宣言の構文」*の「構文」を参照してください。
	注意点: ステートメントセクションで、式識別子に BOOL データタイプの式を割り付ける必要があります!

シーケンス	式は ST ソースファイルの実装セクションでのみ宣言できます。 したがって、式は、WAITFORCONDITION 制御構造から式を呼び出すプログラムの前にある必要があります。 宣言セクション (ページ 157)は ステートメントセクション (ページ 158)の前にある必要があります。
頻度	ST ソースファイルごとに任意の回数
必須セクション	なし

式の詳細については、式 (ページ 149)を参照してください。WAITFORCONDITIONステートメントと併せて、*『SIMOTION基本機能』*機能マニュアルを参照してください。

5.1.1.6 ファンクションブロック(FB)

FB はプログラムオーガニゼーションユニット(POU)として分類されます。FB は、プログラムから呼び出し可能なスタティックデータと割り付け済みパラメータを持つソースファイルセクションです(内部変数は呼び出しと呼び出しの間にその値を保持します)。FB はメモリを持っているため、いつでも、またユーザプログラムのどこからでも、その出力パラメータにアクセスすることができます。

FB は以下の構文を持っています。

表 5-5 ファンクションブロックの構文

構文	`FUNCTION_BLOCK name`
	`// 宣言セクション`
	`// ステートメントセクション`
	`END_FUNCTION_BLOCK`
	name は、ファンクションブロックの識別子を表します。
	宣言セクションで変数の宣言に使用できるキーワード: 変数宣言 (ページ 159)の表*「変数宣言の構文」の「構文」*を参照してください。

特殊機能	ファンクションブロック(FB)を呼び出す前に、インスタンスを宣言する必要があります。変数を宣言し、データタイプとしてファンクションブロックの識別子を入力します。インスタンスは、ローカルに(プログラムまたはファンクションブロックの宣言セクションの VAR / END_VAR 内で)、またはグローバルに(インターフェースまたは実装セクションの VAR_GLOBAL / END_VAR 内で)宣言することができます。 FC で FB のインスタンスを宣言することはできません。
シーケンス	FB は実装セクションでのみ定義できます。 順序についての注意点: FB は FB を呼び出す POU の前にある必要があります! 宣言セクション (ページ 157)は ステートメントセクション (ページ 158)の前にある必要があります。
頻度	ST ソースファイルごとに任意の回数
必須セクション	なし

FBの詳細については、ファンクションおよびファンクションブロックの作成と呼び出し (ページ 131)を参照してください。

5.1.1.7 プログラム

プログラムはプログラムオーガニゼーションユニット(POU)として分類されます。プログラムは、そのタスク割り付けに従ってターゲットシステム上で呼び出され(*『SIMOTION 基本機能』*機能マニュアルの*「実行システムの設定」*を参照)、FC と FB を呼び出すことができます。

プログラムは以下の構文を持っています。

表 5-6 プログラムの構文

構文	PROGRAM name
	// 宣言セクション
	//ステートメントセクション
	END_PROGRAM
	name は、プログラムの名前を表します。
	宣言セクションで変数の宣言に使用できるキーワード:
	変数宣言 (ページ 159)の表*「変数宣言の構文」*の*「構文」*を参照してください。

シーケンス	プログラムは実装セクションでのみ定義できます。 プログラムは、式、FC、および FB の後に置くと便利です。これにより、プログラムがソースファイルセクションを認識し、使用できるようになります。 宣言セクション (ページ 157)は ステートメントセクション (ページ 158)の前にある必要があります。
頻度	ST ソースファイルごとに任意の回数
必須セクション	なし

プログラムの詳細については、プログラム (ページ 148)を参照してください。

5.1.1.8 宣言セクション

プログラムオーガニゼーションユニット(POU)の宣言セクションには、POU のデータタイプ定義と変数宣言が含まれます。

宣言セクションは以下の構造を持っています。

表 5-7 宣言セクションの構造

構造	// データタイプ定義 // 変数宣言
シーケンス	宣言セクションの最初または最後に明示的なキーワードはありません。宣言セクションは、各プログラムオーガニゼーションユニット(POU)のキーワードの後に開始し、ステートメントセクションの最初の実行可能ステートメントで終了します。 宣言セクションには、以下が任意の順序で含まれます。 • データタイプ定義 (ページ 158) POU でローカルに有効なユーザ定義データタイプ(UDT) • 変数宣言 (ページ 159) POU でローカルに有効な変数と定数 POUのタイプに応じて使用できるキーワード: 変数宣言 (ページ 159)の表*「変数宣言の構文」*の*「構文」*を参照してください。 注意点: 識別子は、それが使用される前に宣言されている必要があります。
頻度	POU ごとに 1 回
必須セクション	なし

5.1.1.9 ステートメントセクション

POU のステートメントセクションは、個々の(実行可能な)ステートメントで構成されます。

ステートメントセクションは以下の構造を持っています。

表 5-8 ステートメントセクションの構造

構造	// ステートメント
シーケンス	ステートメントセクションの最初または最後に明示的なキーワードはありません。ステートメントセクションは、宣言セクションの後に開始し、各 POU のキーワードで終了します。
頻度	POU ごとに 1 回
必須セクション	なし

5.1.1.10 データタイプ定義

データタイプ定義では、ユーザ定義データタイプ(UDT)を指定します。変数の宣言に UDT を使用することができます。UDT は、FC、FB、およびプログラムのインターフェースセクション、実装セクション、および宣言セクションで定義できます。

データタイプ定義は以下の構文を持っています。

表 5-9 データタイプ定義の構文

構文	TYPE name : data_type_specification; // ... END_TYPE *name* は、変数の宣言に使用する個々のデータタイプの名前を表します。 *data_type_specification* は、任意のデータタイプまたは構造体を表します。TYPE と END_TYPE の間には個々のデータタイプがいくつでも出現することができます。
シーケンス	UDT は次のように定義できます。 • インターフェースセクション内 ST ソースファイル内で UDT が認識され、エクスポートされます。 ユニット変数の宣言のためにインターフェースおよび実装セクションで、また、ローカル変数の宣言のためにすべての POU で UDT を使用することができます。 さらに、ST ソースファイルへの参照を含むすべてのユニット(USES ステートメントを含む ST)でユニット変数を使用することができます。 • 実装セクション内 ST ソースファイル内で UDT が認識されます。 ユニット変数の宣言のために実装セクションで、また、ローカル変数の宣言のためにすべての POU で UDT を使用することができます。 • POU (FC、FB、プログラム)の宣言セクション内 UDT は POU 内でローカルにのみ認識されます。 ローカル変数の宣言のために POU 内でのみ変数を使用することができます。 UDT は、変数宣言でそれが使用される前に定義されている必要があります。
頻度	TYPE / END_VAR 宣言ブロックは、ソースファイルセクションに複数回出現することができます。宣言ブロック内では、任意の数の UDT を使用できます。
必須セクション	なし

UDTの詳細については、ユーザ定義データタイプ (ページ 77)を参照してください。

5.1.1.11 変数宣言

宣言セクションには変数宣言が含まれます。さらに、宣言セクション自体を、POU (FC、FB、プログラム)、インターフェースセクション、実装セクションに含めることができます。

変数宣言は以下の構文を持っています。

表 5-10 変数宣言の構文

構文	`variable_type`
	` name_list : data_type;`
	` // ...`
	`END_VAR`
	variable_type は、宣言している変数タイプのキーワードを表します。使用できるキーワードは、ソースファイルセクションによって異なります。
	• ST ソースのインターフェースおよび実装セクション
	VAR_GLOBAL: 非保持型ユニット変数
	VAR_GLOBAL CONSTANT: ユニット定数
	VAR_GLOBAL RETAIN: 保持型ユニット変数
	• ファンクションの宣言セクション
	VAR: ローカル変数
	VAR CONSTANT: ローカル定数
	VAR_INPUT:入力パラメータ
	VAR_IN_OUT: 入/出力パラメータ
	• ファンクションブロックの宣言セクション
	VAR: ローカル変数
	VAR CONSTANT: ローカル定数
	VAR_TEMP: テンポラリ変数
	VAR_INPUT:入力パラメータ
	VAR_OUTPUT:出力パラメータ
	VAR_IN_OUT: 入/出力パラメータ
	• プログラムの宣言セクション
	VAR: ローカル変数
	VAR CONSTANT: ローカル定数
	VAR_TEMP: テンポラリ変数
	• 式の宣言セクション
	VAR: ローカル変数
	VAR CONSTANT: ローカル定数
	VAR_INPUT:入力パラメータ
	VAR_IN_OUT: 入/出力パラメータ
	name_list は、宣言する *data_type* データタイプの識別子のリストです。

シーケンス	変数は次のように宣言します。 • ST ソースのインターフェースセクション内 使用できるキーワード: 表の「構文」を参照してください。 ST ソースファイル内でユニット変数が認識され、エクスポートされます。 ST ソースファイルのすべての POU でユニット変数を使用することができます。 さらに、ユニット変数を参照するすべてのユニット(USES ステートメントを含む ST)でユニット変数を使用することができます。 • ST ソースの実装セクション内 使用できるキーワード: 表の「構文」を参照してください。 ST ソースファイル内でユニット変数が認識されます。 ST ソースファイルのすべての POU でユニット変数を使用することができます。 • POU (FC、FB、プログラム、式)の宣言セクション内 POU のタイプに応じて使用できるキーワード: 表の *「構文」*を参照してください。 変数は POU 内でローカルにのみ認識されます。 ローカル変数の宣言のために POU 内でのみ変数を使用することができます。 例外: – FB の外部からファンクションブロックの出力パラメータにアクセスすることもできます。 – [Permit language extensions]コンパイラオプションが有効にされている場合、FBの外部からファンクションブロックの入力パラメータにアクセスすることができます。コンパイラのグローバル設定 (ページ 31)および STコンパイラのローカル設定 (ページ 32)を参照してください。 変数は、それが使用される前に宣言されている必要があります。
頻度	特定の変数タイプの variable_type / END_VAR 宣言ブロックが出現できる回数は、関連するソースファイルセクションによって異なります。 • ST ソースのインターフェースおよび実装セクション 宣言ブロックは複数回出現することができます。 • POU (FC、FB、プログラム、式)の宣言セクション 各宣言ブロック(VAR CONSTANT / END_VAR 以外)は宣言セクションに 1 回だけ出現することができます。 関連するソースファイルセクションに応じて使用できる宣言ブロックおよびキーワード: 表の *「構文」*を参照してください。 宣言ブロック内では、任意の数の変数宣言を行うことができます。
必須セクション	なし

変数宣言の詳細については、変数宣言 (ページ 88)を参照してください。

5.1.2 ST ソースファイル間のインポートとエクスポート

ST には、他のソースファイルのグローバル変数、データタイプ、ファンクション(FC)、ファンクションブロック(FB)、およびプログラムにアクセスできる、ユニットコンセプトが適用されます。したがって、たとえば、再利用可能なサブルーチンをコンパイルし、それを使用可能にすることができます。

5.1.2.1 ユニット識別子

以下では、ユニットとはプログラムソースファイル(たとえば、ST ソースファイル、MCC ソースファイルなど)のことを指します。

SIMOTION SCOUT に定義された ST ソースファイルの名前が識別子として適用されます。

オプションで、ユニット命令を使用して、最初のステートメント(インターフェースセクションの前)としてユニット識別子を繰り返すことができます。構文:

```
UNIT name;
```

name は、SIMOTION SCOUT に定義された ST ソースファイルの名前に対応します。
ST ソースの追加または ST ソースの特性の変更を参照してください。

そこで指定されている名前が ST ソースファイルの名前と異なる場合、ユニットステートメントは無視されます。

5.1.2.2 インターフェースセクションでの指定

インターフェースセクションの構文表に、ユニット(ST ソース)のインターフェースセクションで使用できるステートメントをリストしています。インターフェースセクションで行うことができる指定の例は次の通りです。

- 他のユニットのエクスポートされたコンポーネントを使用するため、他のユニットを参照することができます。構文は USES *unit_name-list* です。このタイプのユニットをインポートユニットと呼びます(詳細については以下を参照)。

  キーワード USES は実装セクションでも使用できます。

- エクスポートするユニット変数、ユニット定数、およびデータタイプを宣言し、エクスポートするファンクション、ファンクションブロック、およびプログラムを指定します。以下、このタイプのユニットをエクスポートユニットと呼びます。詳細については、以下を参照してください。

ユニットはインポートユニットとエクスポートユニットのどちらにもなることができます。

注記

ユニット(STソースファイル)のインターフェースセクションで指定を行うには、表*「インターフェースセクションの構文」*(インターフェースセクション (ページ 152))に示された順序を守る必要があります。この順序を守らないと、STソースファイルをエラーなくコンパイルすることができなくなります。

5.1.2.3 インポートユニットのインターフェースまたは実装セクションでの USES ステートメントの使用

インポートユニットのインターフェースまたは実装セクションに次のステートメントを入力します。

```
USES unit_name-list
```

unit_name-list は、モジュールのインポート元のユニットをコンマで区切ったリストです。

例:

```
USES unit_1, unit_2, unit_3;
```

これにより、インポートしたユニット(たとえば、ST ソースファイル、MCC ユニットなど)のインターフェースセクションで指定または宣言した以下の要素にアクセスできるようになります。

- ファンクションおよびファンクションブロック
- ユーザ定義データタイプ(UDT)
- ユニット変数およびユニット定数

インポートした要素は、現在のユニットに存在したかのように使用することができます。

注記

キーワード USES は、ユニットのインターフェースセクションまたは実装セクションに 1 回だけ出現することができます。複数のユニットをインポートするときは、キーワード USES の後に、コンマで区切ったリストとしてユニットを入力します。

USES ステートメントは、UNIT のインターフェースセクションまたは実装セクションに出現することができます。これは、広範囲にわたり影響を及ぼします。

表 5-11　インターフェースセクションまたは実装セクションへの USES ステートメントの配置に関する影響

影響	USES ステートメント	
	インターフェースセクション	実装セクション
継承	現在のユニットは、インポートされたユニットのエクスポートを続行します。インポートしたユニットは、現在のユニットにアクセスする他のすべてのユニットによって継承されます。 例: 1. **インターフェース**セクションでユニット B がユニット A をインポートします。 2. 今度はユニット C がユニット B をインポートします。 3. その後、ユニット C はユニット A も自動的にインポートします。 A --> B --> C ==> A --> C 継承のため、ユニット A をユニット C に明示的にインポートしてはいけません。	継承が中断されます。 例: 1. **実装**セクションでユニット B がユニット A をインポートします。 2. 今度はユニット C がユニット B をインポートします。 3. その後、ユニット C はユニット A に自動的にはアクセスしません。 ユニット C がユニット A にアクセスしたい場合、ユニット C はユニット A を明示的にインポートする必要があります。
変数宣言	以下で、インポートしたデータタイプのユニット変数の宣言を行うことができます。 • インターフェースセクション • 実装セクション	インポートしたデータタイプのユニット変数の宣言は、実装セクションでのみ可能です。

注記

ユニット変数の使用に関するヒントは、『SIMOTION 基本機能』機能マニュアルを参照してください。

5.1.2.4 インポートユニットの例

以下に、インポートユニット(*myUnit_B*)の例を示します。インポートユニットは、エクスポートユニットの例 (ページ 166)からユニット*myUnit_A*をインポートします。

表 5-12 インポートユニットの例

```
UNIT myUnit_B;          // オプション、ST ソースファイルの名前
INTERFACE
    // ... 必要な場合、USES ステートメント
    PROGRAM myProgram_B;
    // エクスポートするプログラムの指定、FB、FC
    // データタイプおよびユニット変数
END_INTERFACE

IMPLEMENTATION
    USES myUnit_A;     // インポートするユニットの指定

    VAR_GLOBAL
        myInstance : myFB;          // インポートした FB の
                                    // インスタンスの宣言
        myColor : Color;            // インポートしたデータタイプの
                                    // 変数を宣言する
    END_VAR

    PROGRAM myProgram_B

        myColor := GREEN;           // インポートするデータタイプの変数への
                                    // 値割り付け
        Pass:= Pass + 1; // インポートした変数への
                                    // 値割り付け
    END_PROGRAM
END_IMPLEMENTATION
```

5.1.2.5 エクスポートユニットのインターフェースセクションでの指定

エクスポートユニットのインターフェースセクションには、以下のコンストラクトを入力することができます(ここでは、コンストラクトの構文の概略だけを示します。詳細については、ソースファイルセクションの使用 (ページ 151)を参照してください)。

1. 変数およびタイプ宣言
   - TYPE

     完全な宣言を含むユーザ定義データタイプ
   - VAR_GLOBAL、VAR_GLOBAL RETAIN、または VAR_GLOBAL CONSTANT

     完全な宣言を含むグローバル変数

   変数およびタイプは、エクスポートする POU より前にインターフェースセクションで宣言する必要があります。
2. エクスポートする POU (ファンクション、ファンクションブロック、およびプログラム)

   関連キーワードを使用して、エクスポートする POU (ファンクション、ファンクションブロック、またはプログラム)を指定します。各エントリはセミコロンで閉じます。
   - FUNCTION_BLOCK *fb_name* ;
   - FUNCTION *fc_name*;
   - PROGRAM *program_name* ;

指定の順序は任意です。POU 自体は、ST ソースファイルの実装セクションでプログラミングします。

実行システムで ST ソースファイルのプログラムをタスクに割り付けることができるよう、プログラムをインターフェースセクションにリストする必要があります(*『SIMOTION 基本機能』*機能マニュアルの*「実行システムの設定」*を参照)。ST ソースファイルのインターフェースセクションでプログラムをエクスポートできない場合、コンパイラから警告メッセージが出力されます。

ST ソースファイルでのみ使用されるファンクションとファンクションブロックは、インターフェースセクションにリストしないでください。

5.1.2.6 エクスポートユニットの例

以下に、エクスポートユニット(*myUnit_A*)の例を示します。エクスポートユニットは、*myUnit_B* (インポートユニットの例 (ページ 164)を参照)によってインポートされます。

表 5-13 エクスポートユニットの例

```
UNIT myUnit_A;     // オプション、ST ソースファイルの名前

INTERFACE
    // ... ここでは USES ステートメントも可能
    TYPE         // エクスポートするデータタイプの宣言
        Color: (RED, GREEN, BLUE);
    END_TYPE
    VAR_GLOBAL
        Pass : INT := 1;// エクスポートするユニット変数の
                        // 宣言
    END_VAR
    FUNCTION myFC;          // FC のステートメントをエクスポートする
    FUNCTION_BLOCK myFB;    // FB のステートメントをエクスポートする
    PROGRAM myProgram_A;    // プログラムのステートメントをエクスポートする
                            // (実行システムを備えたインターフェースに対して)
END_INTERFACE

IMPLEMENTATION
    Function myFC : LREAL   // ファンクションの完全な記述
        ;     // ... (ステートメント)
    END_FUNCTION

    Function_BLOCK myFB      // ファンクションブロックの完全な記述
        ;     // ... (ステートメント)
    END_FUNCTION_BLOCK

    PROGRAM myProgram_A     // プログラムの完全な記述
        ;     // ... (ステートメント)
    END_PROGRAM
END_IMPLEMENTATION
```

5.2 SIMOTION の変数

ここでは、ST で使用可能な変数の要約を示します。

5.2.1 変数モデル

以下の表は、ST を使用したプログラミングで使用可能なすべての変数タイプを示します。

- SIMOTION デバイスおよびテクノロジーオブジェクトのシステム変数
- グローバルユーザ変数(I/O 変数、デバイスグローバル変数、ユニット変数)
- ローカルユーザ変数(プログラム、ファンクション、またはファンクションブロック内の変数)

システム変数

変数タイプ	意味
SIMOTION デバイスのシステム変数	各 SIMOTION デバイスおよびテクノロジーオブジェクトは、固有のシステム変数を持っています。この変数は次のようにアクセスできます。 • すべてのプログラムから SIMOTION デバイス内でアクセス • HMI デバイスからアクセス システム変数はシンボルブラウザで監視できます。
テクノロジーオブジェクトのシステム変数	

グローバルユーザ変数

変数タイプ	意味
I/O 変数	SIMOTION デバイスまたは周辺機器の I/O アドレスにシンボリック名を割り付けることができます。これにより、I/O に対する以下の直接アクセスおよびプロセスイメージアクセスが可能になります。 • すべてのプログラムから SIMOTION デバイス内でアクセス • HMI デバイスからアクセス この変数は、プロジェクトナビゲータで I/O 要素を選択した後にシンボルブラウザで作成します。 I/O 変数はシンボルブラウザで監視できます。
グローバルデバイス変数	すべての SIMOTION デバイスプログラムおよび HMI デバイスがアクセスできるユーザ定義変数です。 この変数は、プロジェクトナビゲータで GLOBAL DEVICE VARIABLES 要素を選択した後にシンボルブラウザで作成します。 グローバスデバイス変数は保持型として定義することができます。つまり、SIMOTION デバイスの電源が切断されても、変数は格納されたままになります。 グローバルデバイス変数はシンボルブラウザで監視できます。

変数タイプ	意味
ユニット変数	すべてのプログラム、ファンクションブロック、およびファンクション(たとえば、ST ソース、MCC ソース、LAD/FBD ソースなど)がユニット内でアクセスできるユーザ定義変数です。 この変数はユニットで宣言します。 • インターフェースセクション内: この変数を他のユニット(ST ソースファイル、MCC ソースファイル、LAD/FBD ソースファイル)にインポートすることができます。この変数は HMI デバイスでも標準として使用できます。 • 実装セクション内: 関連するユニット内でのみこの変数にアクセスすることができます。 ユニット変数は保持型として宣言することができます。つまり、SIMOTION デバイスの電源が切断されても、変数は格納されたままになります。 ユニット変数はシンボルブラウザで監視できます。

ローカルユーザ変数

変数タイプ	意味
	ユーザ定義変数が定義されたプログラム(またはファンクション、ファンクションブロック)内からアクセスできるユーザ定義変数です。
プログラムの変数(プログラム変数)	変数をプログラムで宣言します。変数は、このプログラム内でのみアクセスできます。スタティック変数とテンポラリ変数が区別されます。 • スタティック変数は、変数を格納するメモリ領域に応じて初期化されます。このメモリ領域はコンパイラオプションを使用して指定します。デフォルトでは、スタティック変数は、プログラムが割り付けられるタスクに応じて初期化されます(*『SIMOTION 基本機能』*機能マニュアルを参照)。 スタティック変数はシンボルブラウザで監視できます。 • テンポラリ変数は、タスクのプログラムを呼び出すたびに初期化されます。 テンポラリ変数をシンボルブラウザで監視することはできません。
ファンクションの変数 (FC 変数)	変数をファンクション(FC)で宣言します。変数は、このファンクション内でのみアクセスできます。 FC 変数は一時的です。つまり、この変数は、FC を呼び出すたびに初期化されます。FC をシンボルブラウザで監視することはできません。
ファンクションブロックの変数(FB 変数)	変数をファンクションブロック(FB)のソースで宣言します。変数は、このファンクションブロック内でのみアクセスできます。スタティック変数とテンポラリ変数が区別されます。 • スタティック変数は、FB が終了したときにその値を保持します。この変数は、FB のインスタンスが初期化されたときだけ初期化されます。これは、FB のインスタンスを宣言した変数タイプに依存します。 スタティック変数はシンボルブラウザで監視できます。 • テンポラリ変数は、FB が終了するとその値を失います。FB を次回呼び出すと、変数は再初期化されます。 テンポラリ変数をシンボルブラウザで監視することはできません。

詳細については、以下のマニュアルを参照してください。

- 対応するリストマニュアルには、SIMOTION テクノロジーパッケージおよび SIMOTION デバイスのすべてのシステム変数に関する簡潔な情報が記載されています。
- テクノロジーオブジェクトのシステム変数の使用についての詳細は、*『SIMOTION モーションコントロールテクノロジーオブジェクト』*機能マニュアルを参照してください。
- 『SIMOTION 基本機能』機能マニュアルには、システム変数と設定データへのアクセス方法に関する情報が記載されています。

- 本マニュアルには、以下に関する情報が記載されています。
  - I/O変数によるI/Oアドレスへのアクセス(周期的タスクの直接アクセスおよびプロセスイメージ (ページ 195)を参照)
  - プロセスイメージアクセス(BackgroundTaskの固定プロセスイメージへのアクセス (ページ 199)を参照)
  - グローバルデバイス変数の作成と使用(グローバルデバイス変数の使用 (ページ 178)を参照)
  - ユニット変数およびローカル変数(スタティック変数およびテンポラリ変数)の使用

注記

STソースファイルをターゲットシステムにダウンロードし、タスクを実行すると、変数の初期化、ひいては変数の内容に影響することに注意してください。変数の初期化の時期 (ページ 184)を参照してください。

5.2.1.1 同名の変数の使用

同じ名前のユニット変数およびローカル変数(プログラム変数、FB 変数、FC 変数)を使用することができます。有効性の範囲の小さい方が、有効性の範囲が大きい方に常に優先されます。これにより、異なるソースファイルセクションで同じ名前を使用できるようになります。

ただし、ローカル変数が同じ名前のユニット変数を隠蔽する場合、コンパイラから警告が出力されます。

たとえば、ユニット変数(広い有効性範囲)と FC 変数(狭い有効性範囲)に同じ名前を使用する場合、ファンクションで宣言した変数だけがこのソースファイルセクション内で有効になります。ユニット変数は、同じ名前のローカル変数が宣言されていない POU でのみ有効です。例を参照してください。

注記

POU の識別子は、変数名と同様に処理されます。

- ユニット変数が POU の識別子と同じであることはできません。
- POU のローカル変数がその識別子と同じであることはできません。
- その他の場合、警告メッセージが生成されます。

表 5-14　　識別子の有効性の例

```
TYPE
    type_a : (enum1, enum2, enum3);
END_TYPE

VAR_GLOBAL
    var_a, var_b : DINT; // ユニット変数
END_VAR

FUNCTION fc_1 : VOID
    VAR
        var_a : type_a;      // 宣言により UNIT 変数を隠蔽する
        var_c : DINT;   // ローカル変数
    END_VAR
    // 使用できるステートメント
    var_a := enum2;          // ローカル変数アクセス
    var_b := 100;            // ユニット変数アクセス
    var_c := -1;             // ローカル変数アクセス

    // 不正なステートメント
    // var_a := 200;
END_FUNCTION

FUNCTION fc_2 : VOID
    VAR
        var_b : type_a;      // 宣言により UNIT 変数を隠蔽する
        var_c : type_a; // ローカル変数
    END_VAR
    // 使用できるステートメント
    var_a := -100;          // ユニット変数アクセス
    var_b := enum3;         // ローカル変数アクセス
    var_c := enum1;         // ローカル変数アクセス
    // 不正なステートメント
    // var_b := 200;
END_FUNCTION
```

5.2.1.2　ユニット変数

ユニット変数は ST ソースファイル全体を通して有効です。すなわち、この変数は、どのソースファイルセクションでもアクセスできます。

ユニット変数は、ST ソースファイルのインターフェースおよび実装セクションで宣言します。宣言の位置によって、ユニット変数の有効性が決まります。

- ユニット変数をインターフェースセクションで宣言すると、他のプログラムソース(たとえば、STソースファイル、MMCユニットなど)で使用できる変数が作成されます。プログラムソースファイル間のインポートとエクスポートの詳細については、STソースファイル間のインポートとエクスポート (ページ 161)を参照してください。

  デフォルトでは、このユニット変数は HMI デバイスでも使用できます。HMI デバイスにエクスポートできるユニット変数の全サイズは、ユニットごとに 64 KB に制限されています。

- ユニット変数を実装セクションで宣言すると、現在のソースファイルのすべてのプログラムオーガニゼーションユニット(POU)で使用できる変数が作成されます。

宣言ブロック内のプラグマを使用すると、ユニット変数のHMIエクスポートのデフォルト設定を変更することができます(変数およびHMIデバイス (ページ 191)および 属性によるコンパイラ出力の制御 (ページ 219)を参照)。

たとえば電源異常の場合などは、異なる動作のユニット変数を定義できます。

- 非保持型ユニット変数(VAR_GLOBAL キーワード): 電源異常の場合は値が失われます。
- 保持型ユニット変数(VAR_GLOBAL RETAIN キーワード): 電源異常の場合でも値が保持されます。
- ユニット定数(VAR_GLOBAL CONSTANTキーワード): 値は変更されないままです(定数 (ページ 94)を参照)。

ユニット変数の効率的な使用に関するヒントは、*『SIMOTION 基本機能』*機能マニュアルを参照してください。

5.2.1.3 非保持型ユニット変数

電源異常の場合、非保持型ユニット変数はその値を失います。

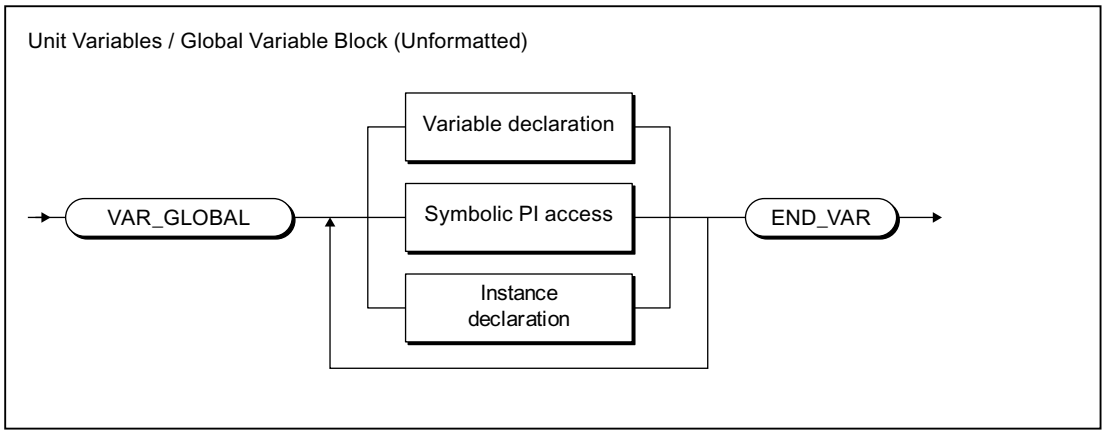


図 5-1 構文: ユニット変数

この宣言ブロックは、インターフェースまたは実装セクション内に複数回出現することができます。変数宣言に変数名とデータタイプを指定します(すべての変数宣言の概要 (ページ 89)および 変数またはデータタイプの初期化 (ページ 90)を参照)。

宣言のスコープとHMIエクスポートについては、ユニット変数 (ページ 170)を参照してください。

注記

非保持型ユニット変数の初期化について:

- 非保持性グローバル変数の初期化 (ページ 186)を参照してください。
- ダウンロード中の動作を設定することができます(**[Options|Settings]**メニューコマンド、**[Project Download]**タブ、**[Initialize all non-retentive data]**チェックボックス)。
- バージョンIDのタイプ、したがってダウンロード時の初期化動作は、SIMOTION Kernelのバージョンに依存します。グローバル変数のバージョンIDとダウンロード中の初期化 (ページ 189)を参照してください。

表 5-15　　非保持型ユニット変数の例

```
INTERFACE
    VAR_GLOBAL    // これらの変数はエクスポートできる
        rotation1        : INT;
        field1           : ARRAY[1.0.10] OF REAL;
        flag1            : BOOL;
        motor1           : motor;    // インスタンスの宣言
    END_VAR
END_INTERFACE
IMPLEMENTATION
    VAR_GLOBAL    // これらの変数はエクスポートできない
                  //  MotionTask
        rotation2        : INT;
        field2           : ARRAY[1.0.10] OF REAL;
        flag2            : BOOL;
        motor2           : motor;    //インスタンスの宣言
    END_VAR
END_IMPLEMENTATION
```

下記も参照

変数の初期化の時期 (ページ 184)

5.2.1.4 保持型ユニット変数

保持型ユニット変数を使用すると、電源異常の間であっても、変数値を同じ状態で格納しておくことができます。

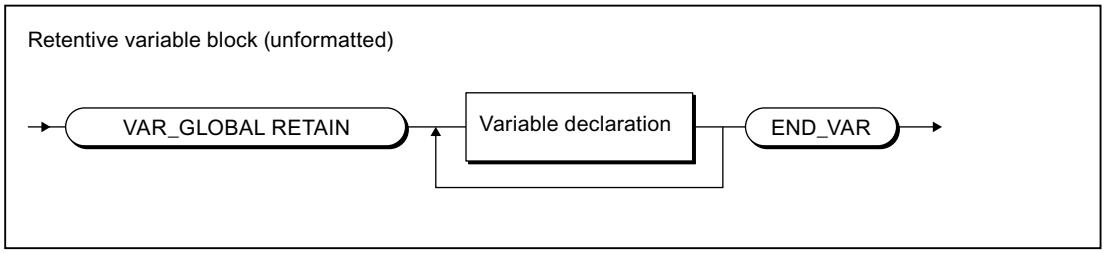


図 5-2 構文: 保持型変数ブロック

この宣言ブロックは、インターフェースまたは実装セクション内に複数回出現することができます。変数宣言に変数名とデータタイプを指定します(すべての変数宣言の概要 (ページ 89)および 変数またはデータタイプの初期化 (ページ 90)を参照)。

宣言のスコープとHMIエクスポートについては、ユニット変数 (ページ 170)を参照してください。

注記

- 保持型ユニット変数の初期化について:
  - 保持性グローバル変数の初期化 (ページ 185)を参照してください。
  - ダウンロード中の動作を設定することができます(**[Options|Settings]**メニューコマンド、**[Project Download]**タブ、**[Initialize all retentive data]**チェックボックス)。
  - バージョンIDのタイプ、したがってダウンロード時の初期化動作は、SIMOTION Kernelのバージョンに依存します。グローバル変数のバージョンIDとダウンロード中の初期化 (ページ 189)を参照してください。
- 保持型変数に使用可能なメモリ容量は、デバイスによって異なります(SIMOTION SCOUT 設定マニュアルの量のフレームワークを参照)。
  限られたメモリ空間を効率的に使用するには、メモリを単一の ST ソースファイルで使用し、変数を降順にソートします!
- SIMOTION SCOUT の保持メモリの容量使用率をチェックします。
  オンラインモードで、チェックする SIMOTION デバイスの**デバイス診断**を呼び出します(オンラインヘルプを参照)。**[Retentive data]**の下の**[System utilization]**タブで、使用可能なメモリ容量を確認することができます。

表 5-16 保持型変数の例

```
VAR_GLOBAL RETAIN
    Measuring field : ARRAY[1.0.10] OF REAL;
    Pass : INT;
    スイッチ BOOL;
END_VAR
```

5.2.1.5 ローカル変数(スタティック変数およびテンポラリ変数)

ローカル変数は、変数を 変数の初期化の時期 (ページ 184)宣言したソースファイルセクション(プログラム、FC、またはFB)でのみ有効です。以下の変数が区別されます。

- スタティック変数 (ページ 175)

  スタティック変数は、ソースファイルセクションのすべてのパスにわたりその値を保持します(ブロックメモリ)。

- テンポラリ変数 (ページ 176)

  テンポラリ変数は、ソースファイルセクションを再度呼び出すたびに初期化されます。

注記

ローカル変数を宣言したソースファイルセクションの外部からローカル変数にアクセスすることはできません。

以下の表は、スタティック変数とテンポラリ変数の宣言の概要を示します。これらの変数を宣言できるソースファイルセクションと、変数の宣言に使用できるキーワードを示しています。

表 5-17　ソースファイルセクションに応じたスタティック変数とテンポラリ変数の宣言キーワード

ソースファイルセクション	宣言のキーワード	
	スタティック変数	テンポラリ変数
ファンクション	–	VAR / END_VAR または VAR_INPUT / END_VAR または VAR_IN_OUT / END_VAR[2]
式	–	VAR / END_VAR または VAR_INPUT / END_VAR または VAR_IN_OUT / END_VAR[2]
ファンクションブロック (Function block)	VAR / END_VAR[1] または VAR_INPUT / END_VAR[1] または VAR_OUTPUT / END_VAR[1]	VAR_TEMP / END_VAR または VAR_IN_OUT / END_VAR[2]
プログラム	VAR / END_VAR[3]	VAR_TEMP / END_VAR

[1] 変数の初期化は、宣言したインスタンスの初期化に依存します(変数の初期化の時期 (ページ 184)を参照)。

[2] 渡された変数の参照(ポインタ)は一時的です。

[3] 変数の初期化は、変数を格納するメモリ領域に依存します。スタティックプログラム変数の初期化 (ページ 190)を参照してください。

注記

STソースファイルをターゲットシステムにダウンロードし、タスクを実行すると、変数の初期化、ひいては変数の内容に影響することに注意してください。変数の初期化の時期 (ページ 184)を参照してください。

表 5-18　スタティック変数とテンポラリ変数の例

```
IMPLEMENTATION
    FUNCTION testFkt
        VAR             // テンポラリ変数の宣言
            flag : BOOL;
        END_VAR
    END_FUNCTION
    FUNCTION_BLOCK testFbst;
        VAR             // スタティック変数の宣言
            rotation1        : INT;
        END_VAR

        VAR_TEMP        // テンポラリ変数の宣言
            help1, help2 : REAL;
        END_VAR
    END_FUNCTION_BLOCK
    PROGRAM testPrg;
        VAR             // スタティック変数の宣言
            rotation2        : INT;
        END_VAR

        VAR_TEMP        // テンポラリ変数の宣言
            help1, help2 : REAL;
        END_VAR
    END_PROGRAM
END_IMPLEMENTATION
```

下記も参照

ローカル変数の初期化 (ページ 187)

5.2.1.6　スタティック変数

スタティック変数は、ソースファイルセクションが終了したときにその最新値を保持します。この値は、次回の呼び出しで再度使用されます。

以下のソースファイルセクションがスタティック変数を含んでいます。

- プログラム
- ファンクションブロック

スタティック変数はスタティック変数ブロックで宣言します。

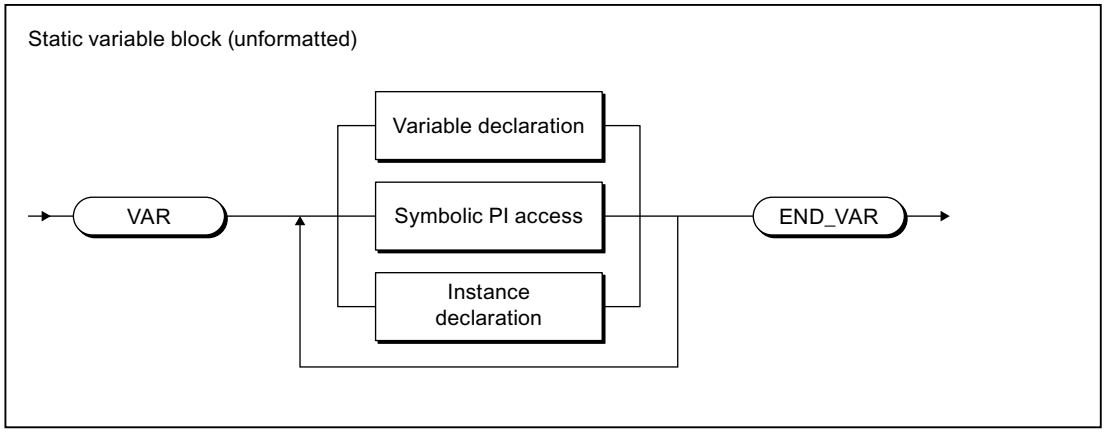


図 5-3　　構文: スタティック変数ブロック

スタティック変数ブロックでは、図の構文に従って以下のことを実行できます。

- 変数(名前およびデータタイプ)の宣言。オプションで初期化付き
- BackgroundTask のプロセスイメージへのシンボリックアクセスの宣言
- ファンクションブロックのインスタンスの宣言

スタティック変数を初期化する場合

- ファンクションブロック内: 宣言したインスタンスの初期化に依存
- プログラム内: プログラムを割り付ける実行動作に依存(*『SIMOTION 基本機能』*機能マニュアルを参照)。

  ローカル変数の初期化 (ページ 187)を参照してください。

5.2.1.7　テンポラリ変数

テンポラリ変数は、ソースファイルセクションを呼び出すたびに初期化されます。この変数の値は、ソースファイルセクションの実行中だけ保持されます。

以下のソースファイルセクションがテンポラリ変数を含んでいます。

- プログラム

  FB およびプログラムのテンポラリ変数ブロックでテンポラリ変数を宣言します(*「構文: FB またはプログラムのテンポラリ変数ブロック」*の図を参照)。

- ファンクションブロック

  FB およびプログラムのテンポラリ変数ブロックでテンポラリ変数を宣言します(*「構文: FB またはプログラムのテンポラリ変数ブロック」*の図を参照)。

- ファンクション

  FC のテンポラリ変数ブロックでテンポラリ変数を宣言します(*「構文: FC のテンポラリ変数ブロック」*の図を参照)。

- 式

  FC のテンポラリ変数ブロックでテンポラリ変数を宣言します(*「構文: FC のテンポラリ変数ブロック」*の図を参照)。

ファンクションおよび式では、FB のテンポラリ変数ブロックでテンポラリ変数を宣言します(*「構文: FB またはプログラムのテンポラリ変数ブロック」*の図を参照)。

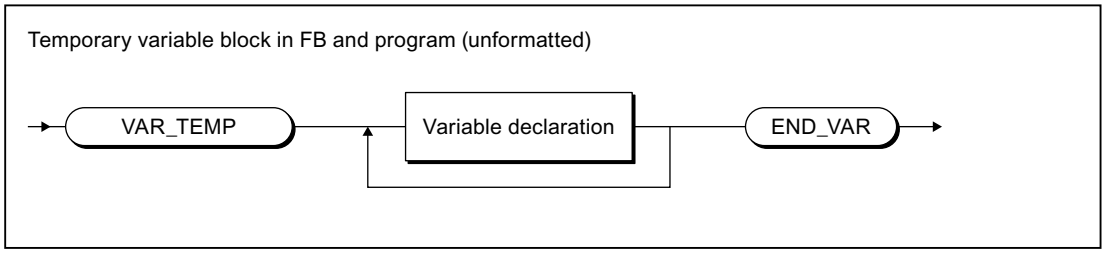


図 5-4　　構文: FB またはプログラムのテンポラリ変数ブロック

ファンクションおよび式では、FC のテンポラリ変数ブロックでテンポラリ変数を宣言します(*「構文: FC のテンポラリ変数ブロック」*の図を参照)。

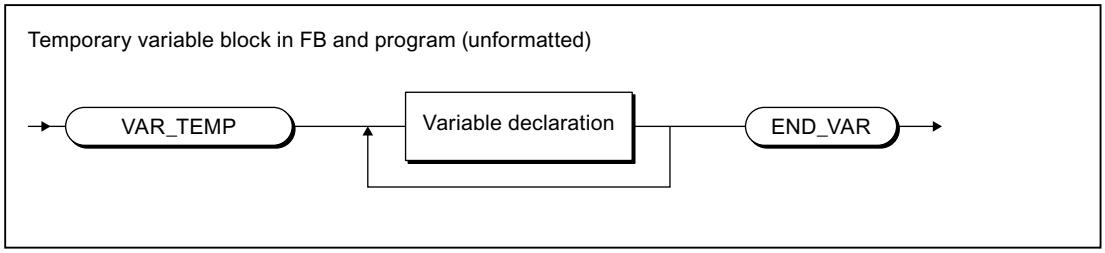


図 5-5　　構文: FC のテンポラリ変数ブロック

5.2.2　名前空間

ST ソースファイルの外部で定義された変数にはその名前を使用してアクセスします。ただし、ST ソースファイルで同じ名前の変数を宣言した場合、ソースの外部で定義された変数は隠蔽され、ST ソースファイルで定義した変数にデフォルトでアクセスすることになります。

デバイス固有の変数やプロジェクト固有の変数、またはライブラリ変数にアクセスできるよう、名前空間を使用することができます(名前空間 (ページ 208)を参照)。

下記も参照

グローバルデバイス変数の使用 (ページ 178)

5.2.3 グローバルデバイス変数の使用

グローバルデバイス変数は、SIMOTION デバイスのすべてのプログラムソース(たとえば、ST ソースファイル、MCC ユニットなど)からアクセスできるユーザ定義変数です。

グローバルデバイス変数は、詳細ビューのシンボルブラウザタブで作成します。これを行うには、オフラインモードで作業している必要があります。

以下に手順の概要を示します。

1. SIMOTION SCOUT のプロジェクトナビゲータで、SIMOTION デバイスサブツリーの **GLOBAL DEVICE VARIABLES** 要素を選択します。
2. 詳細ビューで、**[Symbol browser]**タブを選択し、変数表の最後(空の行)まで下にスクロールします。
3. 表の最後の(空の)行で、以下を入力するか選択します。
   - 変数の**名前**
   - 変数の**データタイプ**(要素のデータタイプだけを使用できます)
4. オプションで、以下の入力を行うことができます。
   - **[Retain]**チェックボックスの選択(これにより変数が保持型として宣言され、電源異常後も値が保持されます)
   - **[Array length]** (配列サイズ)
   - **[Initial value]** (配列の場合、要素ごと)
   - **[Display format]** (配列の場合、要素ごと)

シンボルブラウザ、または SIMOTION デバイスのプログラムを使用してこの変数にアクセスできるようになりました。

ST ソースファイルでは、他の変数と同じようにグローバルデバイス変数を使用することができます。

注記

同じ名前(たとえば、*var-name*)のユニット変数またはローカル変数を宣言している場合、*_device.var-name* を使用してグローバルデバイス変数を指定します。

グローバルデバイス変数の代わりに、個々のユニットでユニット変数を宣言することができます。これを他のユニットにインポートします。これには以下の利点があります。

1. 変数の構造体を使用できます。
2. STOP-RUN の移行中に変数を初期化できます(StartupTask(スタートアップタスク)のプログラム経由)。
3. 新しく作成したグローバルユニット変数の場合、RUN でのダウンロードも可能です。

『SIMOTION 基本機能』機能マニュアルを参照してください。

5.2.4 変数タイプのメモリ範囲

それぞれの変数タイプは、初期化の時期が異なるさまざまなメモリ領域に格納されます。表は以下のことを示しています。

- ST ソースファイルで宣言した変数タイプの使用可能なメモリ領域(SIMOTION Kernel のバージョンに依存)
- 各メモリ領域の初期化の時期

例を使用した説明は、*変数タイプのメモリ領域の例、Kernel V3.1(パート 1～3)*の図を参照してください。

表 5-19　さまざまな変数タイプに割り付けられたメモリ範囲とその初期化

メモリ領域	割り付けられた変数タイプ	初期化
保持メモリ	保持型ユニット変数	ダウンロード設定を使用したダウンロード中
ユニットのユーザメモリ	• 非保持型ユニット変数 • 関連するスタティック変数(VAR、VAR_INPUT、VAR_OUTPUT)を含め、VAR_GLOBAL を使用して宣言したファンクションブロックインスタンス さらに、**[Create program instance data only once]** コンパイラオプションが有効にされている場合: • VAR を使用して宣言した、ユニットプログラムのローカル変数 • 関連するスタティック変数(VAR、VAR_INPUT、VAR_OUTPUT)を含め、ユニットプログラム内で VAR を使用して宣言したファンクションブロックインスタンス	• デバイスをオンにしたとき • ダウンロード設定を使用したダウンロード中
タスクのユーザメモリ	**[Create program instance data only once]**コンパイラオプションが無効にされている場合(標準): • 割り付けたプログラムの VAR を使用して宣言したローカル変数 • 関連するスタティック変数(VAR、VAR_INPUT、VAR_OUTPUT)を含め、割り付けたプログラム内で VAR を使用して宣言したファンクションブロック変数	タスクの実行動作による • 連続タスク: タスクを起動するたび • 周期的タスク: STOP から RUN への移行中に 1 回
タスクのローカルデータスタック (SIMOTION Kernel V3.1 以降)	• タスクで呼び出したプログラムへの参照(ポインタ) • タスクで呼び出したプログラムの VAR_TEMP を使用して宣言したローカル変数	タスクでプログラムを呼び出すたび
	• 呼び出したファンクションブロックインスタンスへの参照(ポインタ) • VAR_TEMP を使用して宣言した、ファンクションブロックのローカル変数 • VAR_IN_OUT を使用して宣言した、ファンクションブロックの入/出力パラメータ [1]	ファンクションブロックインスタンスを呼び出すたび
	• VAR または VAR_INPUT を使用して宣言した、呼び出したファンクションの変数 • VAR_IN_OUT を使用して宣言した、ファンクションの入/出力パラメータ [1] • 呼び出したファンクションの戻り値	ファンクションを呼び出すたび
タスクのローカルデータスタック (SIMOTION Kernel V3.0 まで)	• すべての関連する変数(VAR、VAR_TEMP)を含め、タスクで呼び出したプログラムのコピーされたデータ	タスクでプログラムを呼び出すたび
	• すべての関連する変数(VAR、VAR_INPUT、VAR_OUTPUT、VAR_IN_OUT[1]、VAR_TEMP)を含め、呼び出したファンクションブロックのインスタンスからコピーされたデータ	ファンクションブロックインスタンスを呼び出すたび
	• VAR、VAR_INPUT、または VAR_IN_OUT を使用して宣言した、呼び出したファンクションの変数 [1] • 呼び出したファンクションの戻り値	ファンクションを呼び出すたび
[1] 渡された変数への参照(ポインタ)		

5.2.4.1 例

表 5-20 • 変数タイプのメモリ領域の例、Kernel V3.1 以降(パート 1)

```
INTERFACE
// インターフェースセクションのステートメントでは
// エクスポートするソースコンテンツを指定する。
    FUNCTION FC1;
    FUNCTION_BLOCK FB1;
    PROGRAM p1;

    // インターフェースセクションのユニット変数は
    // HMI デバイスにも表示される。
    VAR_GLOBAL                // 非保持型ユニット変数は
                              // UNIT のユーザメモリにある
        u1_if : INT;
    END_VAR
    VAR_GLOBAL CONSTANT       // ユニット定数は
                              // UNIT のユーザメモリにある
    END_VAR
    VAR_GLOBAL RETAIN         // 保持型ユニット変数は
                              // 保持(パワーフェイルセーフ)メモリにある
    END_VAR
END_INTERFACE

IMPLEMENTATION
// 実装セクションにはさまざまなプログラムオーガニゼーションユニット(POU)の
// 実行可能なコードセクションが含まれる。
// POU はプログラム、FC、または FB であることが可能。
    // 実装セクションのユニット変数は
    // ソースファイル内でのみ使用できる。
    VAR_GLOBAL                // 非保持型ユニット変数は
                              // UNIT のユーザメモリにある
        u1_glob : INT;
    END_VAR
    VAR_GLOBAL CONSTANT       // ユニット定数
                              // UNIT のユーザメモリにある
    END_VAR
    VAR_GLOBAL RETAIN         // 保持型ユニット変数
                              // 保持(パワーフェイルセーフ)メモリにある
    END_VAR
//-----------------------------------------------------------------------
```

表 5-21　　• 変数タイプのメモリ領域の例、Kernel V3.1 以降(パート 2)

```
// 続き
//----------------------------------------------------------------------
    FUNCTION_BLOCK FB1 // ファンクションブロックインスタンスの宣言により
    // データの配置場所を決定する:
    // -      ユニットに VAR_GLOBAL として:
         ユニットのユーザメモリ内
    // -      プログラムに VAR として:
        タスクのユーザメモリ内
    // - ファンクションブロックに VAR として:
    //      ユニットまたはタスクのユーザメモリ内、
    //      上位の FB のインスタンス宣言に依存
    // インスタンスを呼び出すと、インスタンスデータへのポインタが
    // 呼び出し側タスクのスタックに置かれる

        VAR_INPUT         // 入力パラメータは
                          // ユーザメモリにあり、
                          // インスタンスを呼び出すと書き込まれる
            fb_in         : INT;
        END_VAR
        VAR_OUTPUT        // 出力パラメータは
                          // ユーザメモリにあり、
            fb_out         : INT;
        END_VAR
        VAR_IN_OUT        // 入/出力パラメータの
                          // 参照はユーザメモリにあり、
                          // インスタンスを呼び出すと書き込まれる
            fb_in_out     : INT;
        END_VAR

        VAR               // スタティック変数は
                          // ユーザメモリにあり、
                          // FB でローカルに使用できる
             fb_var1     : INT;
        END_VAR

        VAR_TEMP          // テンポラリ変数は
                          // 呼び出し側タスクのスタックにあり、
                          // 呼び出しのたびに初期化される
            fb_temp1     : INT;
        END_VAR

        // コードはユニットのユーザメモリにある
        fb_var1 := fb_var1 + 1;
        fb_out := fb_var1;
        fb_temp1 := fb_in_out;
        fb_in_out := fb_temp1 + fb_in;
    END_FUNCTION_BLOCK
//----------------------------------------------------------------------
```

表 5-22　• 変数タイプのメモリ領域の例、Kernel V3.1 以降(パート 3)

```
// 続き
//-----------------------------------------------------------------------
    FUNCTION FC1 : INT     // ファンクションデータは
        // 呼び出し側タスクのスタックにあり、
        // ファンクションを呼び出すたびに初期化される。
        // 戻り値は呼び出し側タスクのスタックにある

        VAR_INPUT         //入力パラメータ
                          // 呼び出し側タスクのスタックにあり、
                          // ファンクションを呼び出すと書き込まれる
            fc_in         : INT;
        END_VAR

        VAR               // テンポラリ変数は
                          // 呼び出し側タスクのスタックにあり、
            fc_var         : INT;
        END_VAR
        // コードはユニットのユーザメモリにある
        fc_var := 567;
        fc1 := fc_in + fc_var;
    END_FUNCTION

    PROGRAM p1
        VAR               // 変数は
                          // タスクのユーザメモリにある
            p_var         : INT;
            p_varFB    : FB1;
        END_VAR

        VAR_TEMP          // テンポラリ変数は
                          // タスクのスタックにあり、
                          // タスクパスごとに初期化される
            p_temp         : INT;
        END_VAR

    // コードはユニットのユーザメモリにある
        p_temp := p_var;
        p_varFB (fb_in_out := p_temp);
        u1_glob := 4711;
    END_PROGRAM
END_IMPLEMENTATION
```

5.2.4.2 ローカルデータスタックの変数のメモリ必要条件(Kernel V3.1 以降)

変数タイプのメモリ範囲 (ページ 178)に、タスクのローカルデータスタックに格納された変数をリストしています。タスクの設定で、タスクごとにスタックサイズを設定します。

ローカルスタックのメモリの必要条件については、以下の点に注意してください。

- テンポラリローカル変数は、スタック上にそれ自身のサイズを必要とします。
- グローバル変数とスタティックローカル変数は、スタック上にリソースを必要としません。

  ただし、ファンクションの入力パラメータとしてこれらの変数を使用する場合、スタック上にそれ自身のデータサイズが必要となります。
- タスクでファンクションを複数回呼び出しても、スタックのリソースは 1 回しか使用されません。
- BOOL タイプの変数は、スタック上に 1 バイトを必要とします。

[Program Structure]ファンクションを使用して、ローカルデータスタック内のPOUのメモリ必要条件に関する情報を取得することができます(Program structure (ページ 212)を参照)。

5.2.4.3 ローカルデータスタックの変数のメモリ必要条件(Kernel V3.0 以前)

変数タイプのメモリ範囲 (ページ 178)に、タスクのローカルデータスタックに格納された変数をリストしています。タスクの設定で、タスクごとにスタックサイズを設定します。

ローカルスタックのメモリの必要条件については、以下の点に注意してください。

- プログラムのスタティックローカル変数は、スタック上にその 2 倍のサイズを必要とします。
- FB のスタティックローカル変数は、呼び出しの深さに応じて、スタック上にその数倍のサイズを必要とします。
- (プログラム、FB、および FC の)テンポラリローカル変数は、スタック上にそれ自身のサイズを必要とします。
- グローバル変数は、スタックメモリ空間を占有しません。

  ただし、ファンクションまたはファンクションブロックの入力パラメータとしてこの変数を使用する場合、この変数はスタック上にその通常の空間を占有します。
- タスクでファンクションを複数回呼び出しても、スタックのリソースは 1 回しか使用されません。
- BOOL タイプの変数は、スタック上に 1 バイトを必要とします。

注記

ファンクションブロックインスタンスを呼び出すと、インスタンスを VAR_GLOBAL インスタンスとして宣言していても、すべてのインスタンスデータがローカルデータスタックにコピーされます(、ファンクションブロック呼び出しの最適化と比較してください)。

5.2.5 変数の初期化の時期

変数初期化の時期は、以下により決まります。

- 変数が割り当てられるメモリ領域
- オペレータの操作(ソースファイルのターゲットシステムへのダウンロードなど)
- プログラムが割り当てられたタスクの実行動作(シーケンシャル、サイクリック)

すべての変数タイプと変数初期化の時期を、以下の表に記載します。タスクに関する基本情報については、『*SIMOTION 基本機能*機能マニュアル』を参照してください。

ダウンロード中の変数初期化の動作は、次のように設定します。これを実行するには、**[Options|Settings]**メニューコマンド、**[Download]**タブの順に選択します。

注記

ユニット変数またはグローバルデバイス変数の値は、SIMOTION デバイスから SIMOTION SCOUT にアップロードし、XML フォーマットで保存することができます。

1. *[_saveUnitDataSet]*ファンクションを使用して、ユニット変数またはグローバルデバイス変数の必要なデータセグメントをデータセットとして保存します。
2. SIMOTION SCOUT で**[Save variables]**ファンクションを使用します。

[Restore variables] ファンクションを使用して、これらのデータレコードと変数を SIMOTION デバイスに再びダウンロードすることができます。

詳細については、『SIMOTION SCOUT 設定マニュアル』を参照してください。

これにより、データがプロジェクトのダウンロードにより初期化されていても、(たとえば SIMOTION SCOUT のバージョン変更により)使用不可能になっていても、これらのデータを取得することができます。

下記も参照

保持性グローバル変数の初期化 (ページ 185)
非保持性グローバル変数の初期化 (ページ 186)
ローカル変数の初期化 (ページ 187)
テクノロジーオブジェクトのシステム変数初期化 (ページ 188)
グローバル変数のバージョンIDとダウンロード中の初期化 (ページ 189)

5.2.5.1 保持性グローバル変数の初期化

保持性変数は、停電の後も最後の値を保持します。その他すべてのデータは、デバイスが再び電源投入されると再初期化されます。

保持性グローバル変数は、以下の場合に初期化されます。

- 保持性データのバックアップが失敗するか、バッファが故障した場合
- ファームウェアが更新された場合
- メモリリセット(MRES)が実行された場合
- SIMOTION P350 のリスタートファンクション(Del. SRAM)の実行時
- *[_resetUnitData]*ファンクションの適用時(カーネル V3.2 現在)。保持性データの異なるデータセグメントを選択することが可能。
- 以下に説明するように、ダウンロードが実行された場合

表 5-23 ダウンロード中の保持性グローバル変数の初期化

変数タイプ	変数初期化の時期
保持性グローバルデバイス変数	ダウンロード時の動作は、*[Initialization of all non-retentive data]*設定により異なります [1]。 • **[Yes]**[2] の場合: すべての保持性グローバルデバイス変数が初期化されます。 • **[No]**[3] の場合: – SIMOTION Kernel のバージョン V3.2 現在: 保持性グローバルデバイス変数に個別のバージョン ID があります。バージョン ID が変更されると、保持性グローバルデバイス変数は初期化されます。 – SIMOTION Kernel のバージョン V3.1 まで: すべてのグローバルデバイス変数(保持性と非保持性)に、共通のバージョン ID があります。バージョン ID が変更されると、すべてのグローバルデバイス変数が初期化されます。
ユニットの保持性ユニット変数	ダウンロード時の動作は、*[Initialization of all non-retentive data]*設定により異なります [1]。 • **[Yes]**[2] の場合: すべての保持性ユニット変数(すべてのユニット)が初期化されます。 • **[No]**[3] の場合: – SIMOTION Kernel のバージョン V3.2 現在: インターフェースセクションまたは実装セクションの保持性ユニット変数について、各宣言ブロックに個別のバージョン識別子があります。このバージョン識別子が変更されると、インターフェースセクションまたは実装セクションの保持性ユニット変数が初期化されます。 – SIMOTION Kernel のバージョン V3.1 まで: ユニットのすべてのユニット変数に、共通のバージョン ID があります(保持性と非保持性、インターフェースセクションと実装セクション内)。このバージョン ID が変更されると、このユニットのすべてのユニット変数が初期化されます。
[1]**[Options|Settings]**メニューコマンドの、**[Download]**タブ。 [2]該当するチェックボックスが選択されます。 [3]該当するチェックボックスがクリアされます。	

下記も参照

グローバル変数のバージョンIDとダウンロード中の初期化 (ページ 189)

5.2.5.2 非保持性グローバル変数の初期化

非保持性グローバル変数は、停電中にその値が失われます。以下の場合に初期化されます。

- 保持性変数の初期化時。たとえば、ファームウェアの更新時や通常のリセット(MRES)時。
- 電源投入中
- *[_resetUnitData]*ファンクションを適用することにより(カーネル V3.2 現在)、非保持性データの異なるデータセグメントを選択することができます。
- 以下に説明するように、ダウンロードが実行された場合

表 5-24 ダウンロード中の非保持性グローバル変数の初期化

変数タイプ	変数初期化の時期
非保持性グローバルデバイス変数	ダウンロード時の動作は、*[Initialization of all non-retentive data]*設定により異なります [1]。 • **[Yes]**[2] の場合: すべての非保持性グローバルデバイス変数が初期化されます。 • **[No]**[3] の場合: – SIMOTION Kernel のバージョン V3.2 現在: 非保持性グローバルデバイス変数に個別のバージョン ID があります。バージョン ID が変更されると、非保持性グローバルデバイス変数は初期化されます。 – SIMOTION Kernel のバージョン V3.1 まで: すべてのグローバルデバイス変数(保持性と非保持性)に、共通のバージョン ID があります。バージョン ID が変更されると、すべてのグローバルデバイス変数が初期化されます。
ユニットの非保持性ユニット変数	ダウンロード時の動作は、*[Initialization of all non-retentive data]*設定により異なります [1]。 • **[Yes]**[2] の場合: すべての非保持性ユニット変数(すべてのユニット)が初期化されます。 • **[No]**[3] の場合: – SIMOTION Kernel のバージョン V3.2 現在: インターフェースセクションまたは実装セクションの非保持性ユニット変数について、各宣言ブロックに個別のバージョン識別子があります。このバージョン識別子が変更されると、インターフェースセクションまたは実装セクションの非保持性ユニット変数が初期化されます。 – SIMOTION Kernel のバージョン V3.1 まで: ユニットのすべてのユニット変数に、共通のバージョン ID があります(保持性と非保持性、インターフェースセクションと実装セクション内)。このバージョン ID が変更されると、このユニットのすべてのユニット変数が初期化されます。

[1] **[Options|Settings]**メニューコマンドの、**[Download]**タブ。

[2] 該当するチェックボックスが選択されます。

[3] 該当するチェックボックスがクリアされます。

下記も参照

グローバル変数のバージョンIDとダウンロード中の初期化 (ページ 189)

5.2.5.3 ローカル変数の初期化

ローカル変数は以下の場合に初期化されます。

- 保持性ユニット変数が初期化された場合
- 非保持性ユニット変数が初期化された場合
- 以下に説明する場合

表 5-25 プログラム編成ユニットの呼び出し時のローカル変数初期化

変数タイプ	変数初期化の時期
ローカルプログラム変数	プログラムのローカル変数には、以下の異なる初期化があります。 • スタティック変数(VAR)は、自身が格納されるメモリ領域に従って初期化されます。スタティックプログラム変数の初期化を参照してください。 • テンポラリ変数(VAR_TEMP)は、タスクのプログラムが呼び出されるたびに初期化されます。
ファンクションブロック(FB)のローカル変数	ファンクションブロックのローカル変数には、以下の異なる初期化があります。 • スタティック変数(VAR、VAR_IN、VAR_OUT)は、FB インスタンスが初期化された場合にのみ初期化されます。 • テンポラリ変数(VAR_TEMP)は、FB インスタンスが呼び出されるたびに初期化されます。
ファンクション(FC)のローカル変数	ファンクションのローカル変数は一時変数であり、ファンクションが呼び出されるたびに初期化されます。

下記も参照

ファンクションの定義 (ページ 132)
ファンクションブロックの定義 (ページ 133)
スタティックプログラム変数の初期化 (ページ 190)

5.2.5.4 ファンクションブロック(FB)のインスタンス初期化

ファンクションブロックインスタンスの初期化は、宣言の場所により決まります。

- 非保持性ユニット変数同様
- プログラムのローカル変数同様
- ファンクションブロックのローカル変数同様

FB はファンクションブロック内で宣言できます。

注記

ローカルデータスタックにある POU のメモリ要件に関する情報は、*[Program Structure]* ファンクションを使用して入手できます。

下記も参照

ファンクションおよびファンクションブロックの呼び出し (ページ 137)
Program structure (ページ 212)

5.2.5.5 テクノロジーオブジェクトのシステム変数初期化

通常テクノロジーオブジェクトのシステム変数には、保持性がありません。テクノロジーオブジェクトに応じて、数個のシステム変数が保持性メモリ領域に格納されます(絶対値エンコーダのキャリブレーションなど)。

初期化動作(ダウンロード中を除く)は、保持性または非保持性グローバル変数の初期化動作と同じです(保持性グローバル変数、非保持性グローバル変数を参照)。

以下の変数のダウンロード中の動作について説明します。

- 非保持性システム変数
- 保持性システム変数

表 5-26　ダウンロード中のテクノロジーオブジェクトシステム変数の初期化

変数タイプ	**変数初期化の時期**
非保持性システム変数	ダウンロード中の動作は、*[Initialization of all non-retentive data]*の設定により異なります [1]。 • **[Yes]**[2] の場合: すべてのテクノロジーオブジェクトが初期化されます。 – すべてのテクノロジーオブジェクトが再構築され、すべての非保持性システム変数が初期化されます。 – すべての技術上のアラームがクリアされます。 • **[No]**[3] の場合: SIMOTION SCOUT で変更されたテクノロジーオブジェクトだけが初期化されます。 – 問題のテクノロジーオブジェクトは再構築され、非保持性システム変数が初期化されます。 – 関連するテクノロジーオブジェクトで保留中のすべてのアラームはクリアされます。 – *[Power On]*によってのみ認識可能なアラームが初期化されないテクノロジーオブジェクト上で保留中の場合、ダウンロードは中止されます。
保持性システム変数	*[Initialization of all retentive data]*の設定 [1] は、ダウンロード中の動作にまったく影響を与えません。 テクノロジーオブジェクトが SIMOTION SCOUT で変更された場合にのみ、その保持性システム変数が初期化されます。 その他すべてのテクノロジーオブジェクトの保持性システム変数は保持されます(絶対値エンコーダのキャリブレーションなど)。
[1] **[Options|Settings]**メニューコマンドの、**[Download]**タブ。 [2] 該当するチェックボックスが選択されます。 [3] 該当するチェックボックスがクリアされます。	

5.2.5.6 グローバル変数のバージョン ID とダウンロード中の初期化

表 5-27 グローバル変数のバージョン ID とダウンロード中の初期化

データセグメント		SIMOTION Kernel バージョン V3.2 現在	SIMOTION Kernel のバージョン V3.1 まで
グローバルデバイス変数			
	保持性グローバルデバイス変数 非保持性グローバルデバイス変数	• グローバルデバイス変数の各データセグメントに個別のバージョン ID があります。 • このバージョン ID は、データセグメント内で以下の変更に応じて変更されます。 – 変数の追加または削除 – 変数のデータタイプ変更 • このバージョン ID は、以下の場合には変更されません。 – 他のデータセグメント内の変更 – 初期化値に対する変更 [1] • ダウンロード中は [2]、以下のルールが当てはまります。データセグメントのバージョン ID が変更された場合にのみ、データセグメントが初期化されます。 • データのバックアップと初期化にファンクションを使用することができます。	• グローバルデバイス変数のすべてのデータセグメントに共通のバージョン ID があります。 • このバージョン ID は、データセグメント内で変数の宣言が変更された場合にのみ変更されます。 • ダウンロード中は [2]、以下のルールが当てはまります。バージョン ID が変更された場合、すべてのデータセグメントが初期化されます。 • データのバックアップと初期化にファンクションを使用することはできません。
ユニットのユニット変数			
	インターフェースセクションの保持性ユニット変数 実装セクションの保持性ユニット変数 インターフェースセクションの非保持性ユニット変数 実装セクションの非保持性ユニット変数	• データセグメント内の各宣言ブロックに、個別のバージョン ID があります。 • このバージョン ID は、データブロック内で以下の変更に応じて変更されます。 – 変数の追加または削除 – 変数のデータタイプ変更 – データブロックで使用されるデータタイプ定義の変更(個別のまたはインポートされた [3] ユニット) • このバージョン ID は、以下の場合には変更されません。 – 他の宣言ブロックの追加 – 他のデータブロックでの変更 – 初期化値に対する変更 [1] – データブロックで使用されないデータタイプ定義の変更 – ファンクションの変更 • ダウンロード中は [2]、以下のルールが当てはまります。データブロックのバージョン ID が変更された場合にのみ、データブロックが初期化されます。 • データバックアップと初期化のファンクションでは、宣言ブロックのバージョン ID が考慮されます。	• ユニット内のすべてのグローバル変数に共通のバージョン ID があります。 • このバージョン ID は、以下が変更されるとそれに応じて変更されます。 – データセグメント内の変数宣言 – ユニット内のグローバルデータタイプの宣言 – インポートされたインターフェースセクション内の宣言 [3] • ダウンロード中は [2]、以下のルールが当てはまります。バージョン ID が変更された場合、すべてのデータセグメントが初期化されます。 • データバックアップでのファンクションの使用は、以下の場合にのみ可能です。インターフェースセクションの非保持性ユニット変数

[1] 変更された初期化値は、問題のデータセグメントが初期化されるまで有効にはなりません。

[2] *[initialization of all retentive data]*が [No]でかつ *[Initialization of all non-retentive data]*が [No] の場合、他の設定については 「*変数初期化の時期*」の保持性と非保持性グローバル変数のセクションを参照してください。

[3] インポートとエクスポートについては、以下の関連項目リンクを参照してください。

下記も参照

STソースファイル間のインポートとエクスポート (ページ 161)

インポートユニットのインターフェースまたは実装セクションでのUSESステートメントの使用 (ページ 162)

5.2.5.7 スタティックプログラム変数の初期化

以下のバージョンが以下のスタティック変数に影響を与えます。

- VAR で宣言されたユニットプログラムのローカル変数
- ユニットプログラム内で VAR によって宣言されたファンクションブロックインスタンス。関連付けられたスタティック変数(VAR、VAR_INPUT、VAR_OUTPUT)など。

初期化動作は、スタティック変数が格納されているメモリ領域によって決まります。この動作は、**[Create program instance data only once]**コンパイラオプションにより決まります。

- **[Create program instance data only once]** (標準)コンパイラオプションが有効にされていない場合:

  スタティック変数は各タスクのユーザメモリに格納され、プログラムに割り当てられます。

  したがって変数の初期化は、プログラムが割り当てられたタスクの実行動作によって異なります(『SIMOTION 基本機能機能マニュアル』を参照)。

  – シーケンシャルタスク(MotionTasks、UserInterruptTask(ユーザ割り込みタスク)、SystemInterruptTask(システム割り込みタスク)、StartupTask(スタートアップタスク)、ShutdownTask(停止タスク))。スタティック変数は、タスクが開始されるたびに初期化されます。

  – サイクリックタスク(BackgroundTask、SynchronousTask(同期制御タスク)、TimerInterruptTask(タイマー割り込みタスク))。スタティック変数は、STOP から RUN に移行中にのみ 1 回だけ初期化されます。

- **[Create program instance data only once]**コンパイラオプションが有効な場合:

  スタティック変数は、タスクのユーザメモリに 1 回だけ格納されます。したがって、これらの変数は非保持性ユニット変数とともに初期化されます(「非保持性グローバル変数の初期化」を参照)。

  この設定は、プログラムがプログラム内で呼び出される場合に必要です。

下記も参照

コンパイラのグローバル設定 (ページ 31)

STコンパイラのローカル設定 (ページ 32)

非保持性グローバル変数の初期化 (ページ 186)

5.2.6 変数および HMI デバイス

使用可能な場合、HMI デバイスに以下の変数がエクスポートされます。

- SIMOTION デバイスのシステム変数
- テクノロジーオブジェクトのシステム変数
- I/O 変数
- グローバルデバイス変数
- インターフェースセクションの保持型および非保持型ユニット変数(デフォルト設定)

  このデフォルト設定は、以下のプラグマを使用して宣言ブロックごとに変更できます。

  ```
  { HMI_Export := FALSE; }
  ```

  このような識別された宣言ブロックのユニット変数は HMI デバイスにエクスポートされません。ダウンロード中、このユニット変数に対しては HMI の一貫性テストも省略されます。

  HMI デバイスにエクスポートできるユニット変数の全サイズは、ユニットごとに 64 KB に制限されています。

  属性によるコンパイラ出力の制御 (ページ 219)を参照してください。

以下の変数は HMI デバイスにエクスポート**されず**、HMI デバイスでは使用**できません**。

- 実装セクションの保持型および非保持型ユニット変数(デフォルト設定)

  このデフォルト設定は、以下のプラグマを使用して宣言ブロックごとに変更できます。

  ```
  { HMI_Export := TRUE; }
  ```

  このような識別された宣言ブロックのユニット変数は、HMI デバイスにエクスポートされます。結果として、ダウンロード中にユニット変数に対して HMI の一貫性テストが行われます。

  HMI デバイスにエクスポートできるユニット変数の全サイズは、ユニットごとに 64 KB に制限されています。

  属性によるコンパイラ出力の制御 (ページ 219)を参照してください。

- POU のローカル変数

5.3 入力および出力(プロセスイメージ、I/O 変数)へのアクセス

5.3.1 入力と出力へのアクセス概要

SIMOTION デバイスの入力と出力、集中型 I/O と分散型 I/O にアクセスすることができます。

- I/O 変数を使用した直接アクセス経由

  I/O 変数(名前と I/O アドレス)を定義します。アドレス範囲全体を使用できます。

  シーケンシャルプログラミングによる直接アクセスを使用するほうが望ましいといえます(MotionTasks 内)。特定の時点での現在の入力値と出力値へのアクセスは、この場合には特に重要です。

- I/O 変数を使用したサイクリックタスクのプロセスイメージ経由

  SIMOTION デバイスの RAM にあるメモリ領域。ここに、SIMOTION デバイスのアドレス空間がマッピングされます。ミラーイメージは割り当てられたタスクによりリフレッシュされ、サイクル全体を通じて一貫性を持ちます。割り当てられたタスクをプログラミングする場合にこの方法が選択されます(サイクリックプログラミング)。

  I/O 変数(名前と I/O アドレス)を定義して、この変数にタスクを割り当てます。
  SIMOTION デバイスのアドレス領域全体を使用することができます。

  この I/O 変数に直接アクセスすることも引き続き可能です。直接アクセスは*_direct.varname* を使用して指定します。

- BackgroundTask の固定プロセスイメージの使用

  SIMOTION デバイスの RAM にあるメモリ領域。ここに、I/O アドレス空間のサブセットがマッピングされます。ミラーイメージは BackgroundTask によりリフレッシュされ、サイクル全体を通じて一貫性を持ちます。BackgroundTask をプログラミングする場合にこの方法が選択されます(サイクリックプログラミング)。

  アドレス範囲 0～ 63 を使用できます。サイクリックタスクのプロセスイメージは除きます。

注記

プロセスイメージによるアクセスのほうが、直接アクセスより効率的です。

5.3.2 直接アクセスとプロセスイメージアクセスの重要な機能

表 5-28 直接アクセスとプロセスイメージアクセスの重要な機能

	直接アクセス	サイクリックタスクのプロセスイメージへのアクセス	BackgroundTask の固定プロセスイメージへのアクセス
指定可能なアドレス範囲	SIMOTION デバイスのアドレス範囲全体 **例外:** 複数バイトで構成された I/O 変数では、アドレス 63 と 64 を連続して指定することはできません(例: PIW63 または PQD62 は指定できません)。 使用するアドレスは、I/O に存在し、適切にコンフィグレーションされている必要があります。		0 .. 63, サイクリックタスクのプロセスイメージで使用されるアドレスは除く。 I/O に存在しないアドレスまたはコンフィグレーションされていないアドレスも使用できます。
割り当てられたタスク	なし	選択するサイクリックタスク: • SynchronousTask(同期制御タスク)、 • TimerInterruptTask(タイマー割り込みタスク)、 • BackgroundTask	BackgroundTask
更新	• SIMOTION デバイス C230-2、C240、P350 のオンボード I/O: 更新は 125 µs のサイクルクロックで発生します。 • PROFIBUS DP、PROFINET、P-Bus、DRIVE-CLiQ 経由の I/O、および D4xx SIMOTION デバイスのオンボード I/O: 更新は位置制御サイクルクロックで発生します。 入力はサイクルクロックの開始時に読み取られます。 出力はサイクルクロックの終了時に書き込まれます。	更新は割り当てられたタスクにより発生します。 • 入力は割り当てられたタスクの開始前に読み取られ、プロセス入力イメージに転送されます。 • プロセス出力イメージは、割り当てられたタスクが完了した後に出力に書き込まれます。	*BackgroundTask* に更新が行われます。 • 入力は *BackgroundTask* が開始される前に読み取られ、プロセス入力イメージに転送されます。 • プロセス出力イメージは、BackgroundTask の完了時に出力に書き込まれます。
一貫性	–	割り当てられたタスクのサイクル全体を通じて **例外:** 出力への直接アクセスが発生します。	*BackgroundTask* のサイクル全体を通じて **例外:** 出力への直接アクセスが発生します。
	一貫性は基本データタイプについてのみ確保されます。 配列の使用時、データの一貫性を確保する責任はユーザにあります。		
用途	MotionTasks に適切	割り当てられたタスクに適切	BackgroundTask に適切
変数として宣言	シンボルブラウザでデバイス全体について必要		可能、ただし必要ではない。 • シンボルブラウザでデバイス全体について • ユニット変数として • プログラムのローカル変数として
出力の書き込み保護	可能。**[Read only]**ステータスを選択できます。	サポートされていません。	サポートされていません。

	直接アクセス	サイクリックタスクのプロセスイメージへのアクセス	BackgroundTask の固定プロセスイメージへのアクセス
配列の宣言	可能。		サポートされていません。
エラーの場合の反応	ユーザプログラムからアクセス中のエラー。代替応答が可能。 • CPU 停止 [1] • 置換値 • 最終値	プロセスイメージの生成中のエラー。代替応答が可能。 • CPU 停止 [1] • 置換値 • 最終値	プロセスイメージの生成中のエラー。応答は以下のとおりです。CPU 停止 [1]. **例外:** 直接アクセスが同じアドレスで作成されている場合、そのアドレスに設定されている動作が適用されます。
	『SIMOTION 基本機能』マニュアルの機能の説明を参照してください。		
絶対アドレスの使用	サポートされていません。		サポートされています。
アクセス			
• RUN モードで	制限事項なし。	制限事項なし。	制限事項なし。
• StartupTask(スタートアップタスク)中	制限事項付きで可能。 • 入力を読み取ることはできます。 • 出力は StartupTask(スタートアップタスク)が完了するまで書き込まれません。	制限事項付きで可能。 • 入力は StartupTask(スタートアップタスク)の開始時に読み取られます。 • 出力は StartupTask(スタートアップタスク)が完了するまで書き込まれません。	制限事項付きで可能。 • 入力は StartupTask(スタートアップタスク)の開始時に読み取られます。 • 出力は StartupTask(スタートアップタスク)が完了するまで書き込まれません。
• ShutdownTask(停止タスク)中	制限事項なし。	制限事項付きで可能。 • 入力は最後の更新のステータスを保持します。 • 出力は書き込まれなくなります。	制限事項付きで可能。 • 入力は最後の更新のステータスを保持します。 • 出力は書き込まれなくなります。

[1] PeripheralFaultTask(ペリフェラルエラータスク)を呼び出します。

5.3.3 周期的タスクの直接アクセスおよびプロセスイメージ

入力と出力への直接アクセスと、サイクリックタスクのプロセスイメージへのアクセスは、常に I/O 変数により発生します。

機能:「直接アクセスとプロセスイメージアクセスの重要な機能」を参照してください。

表 5-29 直接アクセス用の SIMOTION デバイスのアドレス範囲と、SIMOTION Kernel バージョンによるサイクリックタスクのプロセスイメージ

SIMOTION デバイス	SIMOTION Kernel バージョンのアドレス範囲		
	V3.0 まで	V3.1、V3.2	V4.0 以降
C230-2	0 .. 1023	0 .. 2047 [4]	0 .. 2047 [4]
C240	–	–	0 .. 4096 [4]
D410 [1]	–	–	0 .. 16383 [4 5]
D425[2]	–	0 .. 4095 [4]	0 .. 16383 [4 5]
D435[3]	0 .. 1023	0 .. 4095 [4]	0 .. 16383 [4 5]
D445[2]	–	0 .. 4095 [4]	0 .. 16383 [4 5]
P350	0 .. 1023	0 .. 2047 [4]	0 .. 4095 [4]

[1] V4.1 現在で使用可能。

[2] V4.2 現在で使用可能。

[3] V4.0 現在で使用可能。

[4] 分散 I/O (PROFIBUS DP 経由)の場合、転送量は PROFIBUS DP ライン当たり 1024 バイトに制限されます。

[5] 分散 I/O (PROFINET 経由)の場合、転送量は PROFINET セグメント当たり 4096 バイトに制限されます。

注記

直接アクセスの I/O アドレスとサイクリックタスクのプロセスイメージのルールに従ってください。

下記も参照

直接アクセスとプロセスイメージアクセスの重要な機能 (ページ 193)

直接アクセスのI/Oアドレスとサイクリックタスクのプロセスイメージのルール (ページ 196)

5.3.3.1 直接アクセスの I/O アドレスとサイクリックタスクのプロセスイメージのルール

直接アクセスの I/O 変数アドレスとサイクリックタスクのプロセスイメージに関する、以下のルールに従う必要があります。これらのルールへの準拠は、SIMOTION プロジェクトの一貫性チェック中にチェックされます(ダウンロード中など)。

1. I/O 変数に使用するアドレスは I/O 内に存在し、HW コンフィグレーションによって適宜コンフィグレーションされている必要があります。
2. 複数バイトで構成されている I/O 変数に、アドレス 63 と 64 を連続して指定することはできません。

   以下の I/O アドレスは指定できません。

   入力: PIW63、PID61、PID62、PID63

   出力: PQW63、PQD61、PQD62、PQD63
3. 複数バイトで構成される I/O 変数のすべてのアドレスは、HW コンフィグレーションでコンフィグレーションされたアドレス領域内に存在している必要があります。
4. I/O アドレス(入力または出力)は、データタイプ BYTE、WORD、または DWORD の単一 I/O 変数か、これらのデータタイプの配列によってのみ使用することができます。データタイプ BOOL の I/O 変数による個々のビットへのアクセスは可能です。
5. 複数のプロセス(I/O 変数、テクノロジーオブジェクト、PROFIBUS メッセージフレームなど)が 1 つの I/O アドレスにアクセスする場合、以下のことが当てはまります。
   - すべてのプロセスは、同じデータタイプ(BYTE、WORD、または DWORD、もしくはこれらのデータタイプの ARRAY)によってアクセスする必要があります。個々のビットへのアクセスは、このことに関係なく可能です。
   - 唯一のプロセスだけが、出力の I/O アドレスに書き込みアクセス権を持ちます(データタイプ BYTE、WORD、または DWORD)。書き込みアクセスの別のプロセスで使用される I/O 変数を使用した出力への読み取りアクセスは可能です。
   - 複数プロセスからの異なるビットアドレスへの書き込みアクセスは可能です。ただし、この場合にはデータタイプ BYTE、WORD、または DWORD による書き込みアクセスはできません。

5.3.3.2 直接アクセスまたはサイクリックタスクのプロセスイメージ用の I/O 変数の作成

I/O 変数は、詳細ビューのシンボルブラウザで作成します。これを実行するには、オフラインモードで作業する必要があります。

手順の概要を示します。

1. SIMOTION SCOUT のプロジェクトナビゲータで、SIMOTION デバイスのサブツリー内の I/O エレメントを選択します。
2. 詳細ビューで、[Symbol browser]タブを選択し、変数テーブルの末尾(空の行)までスクロールダウンします。
3. テーブルの最後の(空の)行に、以下を入力または選択します。
   - 変数の**[Name]**
   - I/O アドレスを入力するための構文での構文に従う**[I/O address]**
   - 出力のオプション:

     出力に読み取りアクセス権だけが必要な場合は、[Read only]チェックボックスを有効にします。

     これにより、他のプロセス(出力カムの出力、PROFIBUS メッセージフレームなど)により既に読み取り中の出力も読み取ることができます。

読み取り専用の出力変数を、サイクリックタスクのプロセスイメージに割り当てることはできません。

- 変数の**[Data type]**(I/O 変数の指定可能なデータタイプを参照)。

4. オプションで、以下を入力または選択することもできます(データタイプ BOOL についてはできません)。
   - **[Array length]** (配列のサイズ)
   - **[Process image]**または直接アクセス:

     [Read only]チェックボックスがクリアされている場合にのみ割り当てることができます。

     プロセスイメージの場合、I/O 変数を割り当てるサイクリックタスクを選択します。タスクは選択できるようにするため、ランタイムシステムで有効にされ、プログラムを割り当てられている必要があります。

     直接アクセスの場合は、空白のエントリを選択します。
   - エラー状況での動作に関する**[Strategy]**(『*SIMOTION 基本機能*機能マニュアル』を参照)。
   - **[Substitute value]** (配列の場合、各エレメントについて)
   - **[Display format]** (配列の場合、各エレメントについて)

この変数には、シンボルブラウザまたは SIMOTION デバイスの任意のプログラムを使用してアクセスできるようになりました。

通知

サイクリックタスクのプロセスイメージに関して、以下の事項に注意してください。

- 1 つの変数は 1 つのタスクにだけ割り当てることができます。
- 入力と出力の各バイトは、1 つの変数にだけ割り当てることができます。

データタイプ BOOL については、以下に注意してください。

- サイクリックタスクのプロセスイメージとエラーの方針を定義することはできません。バイト全体の I/O 変数によって定義された動作が適用可能です(デフォルト: 直接アクセスまたは CPU 停止)。
- I/O 変数の個々のビットも、ビットアクセスファンクションを使用してアクセスできます。

注記

I/O 変数はオフラインモードでのみ作成できます。SIMOTION SCOUT で I/O 変数を作成し、それらをプログラムソースで使用することができます(ST ソース、MCC チャート、LAD/FBD ソースなど)。

出力は読み書きできますが、入力は読み取り専用であることに注意してください。

新しいまたは更新された I/O 変数をモニタまたは変更する前に、まずプロジェクトをターゲットシステムにダウンロードする必要があります。

下記も参照

I/Oアドレス入力の構文 (ページ 198)

I/O変数の指定可能なデータタイプ (ページ 198)

5.3.3.3 I/O アドレス入力の構文

I/O アドレス入力の構文(データタイプと入力/出力識別子による)

データタイプ	入力	出力	指定可能なアドレス範囲					
			直接アクセス		プロセスイメージ		たとえば直接アクセス D435 V4.1	
BOOL	PIn.x	PQn.x	n:	0 .. *MaxAddr* 0 .. 7		-[1]	n: x:	0 .. 16383 0 .. 7
BYTE	PIBn	PQBn	x :	0 .. *MaxAddr*	n:	0 .. *MaxAddr*	n:	0 .. 16383
WORD	PIWn	PQWn	n:	0 .. 62 64 .. *MaxAddr* - 1	n:	0 .. 62 64 .. *MaxAddr* - 1	n:	0 .. 62 64 .. 16382
DWORD	PIDn	PQDn	n:	0 .. 60 64 .. *MaxAddr* - 3	n:	0 .. 60 64 .. *MaxAddr* - 3	n:	0 .. 60 64.. 16380
n = 論理アドレス x = ビット数								
MaxAddr =	SIMOTION Kernel バージョンに依存する SIMOTION デバイスの最大 I/O アドレス							
例:	アドレス 1022、データタイプ WORD の入力: PIW1022 論理アドレス 63、ビット 3、BOOL データタイプの出力: PQ63.3							
[1] データタイプ BOOL の場合、サイクリックタスクのプロセスイメージを定義することはできません。バイト全体の I/O 変数によって定義された動作が適用可能です(デフォルト: 直接アクセス)。								

5.3.3.4 I/O 変数の指定可能なデータタイプ

I/O アドレスのデータタイプに依存する、直接アクセスとサイクリックタスクのプロセスイメージの I/O 変数に指定可能なデータタイプ

I/O アドレスのデータタイプ	I/O 変数に指定可能なデータタイプ
BOOL (PIn.x、PQn.x)	BOOL
BYTE (PIBn、PQBn)	BYTE、SINT、USINT
WORD (PIWn、PQWn)	WORD、INT、UINT
DWORD (PIDn、PQDn)	DWORD、DINT、UDINT

5.3.4 BackgroundTask の固定プロセスイメージへのアクセス

BackgroundTask の固定プロセスイメージは、SIMOTION デバイス上のランダムアクセスメモリ(RAM)内のメモリ領域で、ここに I/O アドレス空間のサブセットがマッピングされます。この方法は、BackgroundTask (サイクリックプログラミング)のプログラミングに適しています。サイクル全体を通じて一貫性があるからです。

機能: 直接アクセスとプロセスイメージアクセスの重要な機能の表を参照してください。

すべての SIMOTION デバイス用の BackgroundTask の固定プロセスイメージのサイズは 64 バイトです(アドレス範囲 0～63)。

通知
サイクリックタスクのプロセスイメージによりアクセスされるアドレスは使用できません。これらのアドレスは、BackgroundTask の固定プロセスイメージにより読み書きすることはできません。 直接アクセスとサイクリックタスクのプロセスイメージに関する I/O アドレスのルールは適用**されません**。BackgroundTask の固定プロセスイメージへのアクセスは、プロジェクトの一貫性チェック中に考慮されません(ダウンロード中など)。 I/O に存在しないアドレスまたは HW コンフィグレーションでコンフィグレーションされていないアドレスは、通常のメモリアドレスのように扱われます。

BackgroundTask の固定プロセスイメージには、以下によりアクセスできます。

- 絶対 PI(プロセスイメージ)アクセス: 絶対 PI アクセス識別子には、入力/出力のアドレスとデータタイプが含まれています。
- シンボリック PI アクセス: 関連する絶対 PI アクセスを参照する変数を宣言します。
- I/O 変数: シンボルブラウザで、対応する絶対 PI アクセスを参照するデバイス全体の有効な I/O 変数を定義します。

すべてのオプションをこのセクションで説明します。

通知

入力と出力がリトルエンディアンのバイト順序で動作するかに注意してください(たとえば SIMOTION デバイス C230-2 の統合デジタル入力)。以下の条件が満たされます。

1. 入力と出力がアドレス 0～62 にコンフィグレーションされている。
2. 直接アクセスの I/O 変数(データタイプ WORD、INT、または UINT)が、これらの入力と出力に作成されている。
3. また、これらの入力と出力に BackgroundTask の固定プロセスイメージによりアクセスする。

すると、以下が有効になります。

- データタイプ WORD によるアクセスは、I/O 変数と BackgroundTask の固定プロセスイメージによるアクセスと同じ結果をもたらす。
- *[_getInOutByte]*ファンクションを使用した個々のバイトへのアクセス(『*SIMOTION 基本機能*機能マニュアル』を参照)により、個々のバイトがリトルエンディアンの順序で並びます。
- BackgroundTask の固定プロセスイメージによる個々のバイトやビットへのアクセスにより、これらのビットがビッグエンディアンの順序で並びます。

リトルエンディアンとビッグエンディアンの詳細については、『*SIMOTION 基本機能*機能マニュアル』を参照してください。

下記も参照

直接アクセスとプロセスイメージアクセスの重要な機能 (ページ 193)

直接アクセスのI/Oアドレスとサイクリックタスクのプロセスイメージのルール (ページ 196)

5.3.4.1 BackgroundTask の固定プロセスイメージへの絶対アクセス(絶対 PI アクセス)

BackgroundTask の固定プロセスイメージへの絶対アクセスを、該当するアドレスの識別子を直接使用して(暗黙のデータタイプにより)実行します。この識別子の構文は次のセクションで説明します。

この識別子は、通常の変数と同じ方法で使用できます。

注記

出力は読み書きできますが、入力は読み取りアクセス専用です。

5.3.4.2 絶対プロセスイメージアクセスの識別子の構文

絶対プロセスイメージアクセスの識別子の構文

データタイプ	入力	出力	指定可能なアドレス範囲	
BOOL	%In.x または %IXn.x[1]	%Qn.x または %QXn.x[1]	n: x :	0 .. 63[2] 0 .. 7
BYTE	%IBn	%QBn	n:	0 .. 63[2]
WORD	%IWn	%QWn	n:	0 .. 63[2]
DWORD	%IDn	%QDn	n:	0 .. 63[2]
n = 論理アドレス x = ビット番号				
例:	アドレス 62、データタイプ WORD の入力: %IW62 論理アドレス 63、ビット 3、BOOL データタイプの出力: %Q63.3			
[1] 構文%IXn.x または%QXn.x は、I/O 変数の定義時に指定できません。 [2] サイクリックタスクのプロセスイメージで使用されるアドレスを除く。				

同じタイプの変数の割り付けの例を以下にいくつか示します。

表 5-30　絶対 CPU メモリアクセスの例

```
status1 := %I1.1; // BOOL データタイプ
status2 := %IB10; // BYTE データタイプ
status3 := %IW20; // WORD データタイプ
status4 := %ID20; // DWORD データタイプ

%Q1.1 := status1; // BOOL データタイプ
%QB20 := status1; // BYTE データタイプ
%QW20 := status1; // WORD データタイプ
%QS20 := status1; // DWORD データタイプ
```

5.3.4.3 BackgroundTask の固定プロセスイメージへのシンボリックアクセス(シンボリック PI アクセス)

絶対プロセスイメージアクセスを指定することなく、PI にシンボリックにアクセスすることもできます。

以下の変数としてシンボリックアクセスを宣言することができます。

- プログラムのスタティック変数(宣言セクションの VAR / END_VAR 構造内)
- ユニット変数(ST ソースファイルのインターフェースまたは実装セクションの VAR_GLOBAL / END_VAR 構造内)

PI アクセスのためのシンボリック名を宣言する構文を図に示します。

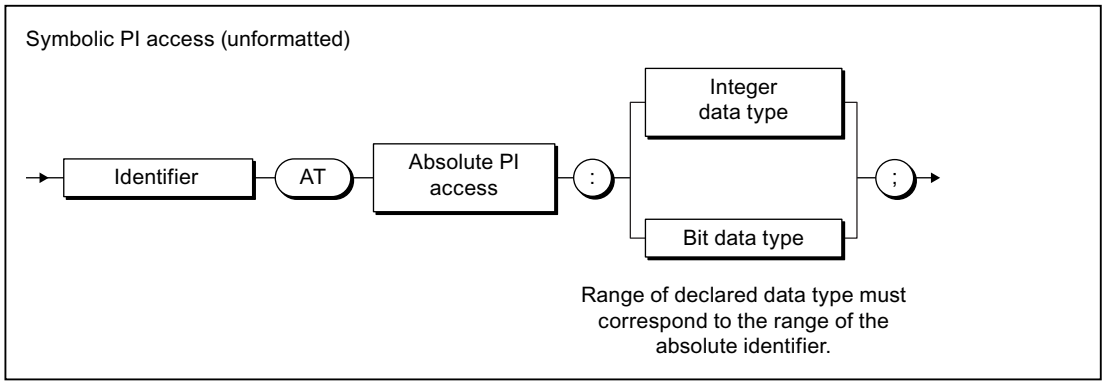


図 5-6　　プロセスイメージへのシンボリックアクセスの宣言

宣言したデータタイプの範囲は、絶対 PI アクセスの範囲と一致している必要があります(以下の表を参照)。数値データタイプを宣言したら、プロセスイメージの内容を整数としてアドレス指定することができます。

5.3.4.4 シンボリックプロセスイメージ(PI)アクセスに指定可能なデータタイプ

絶対 PI アクセスのデータタイプに基づきシンボリック PI アクセスに指定可能なデータタイプ

絶対 PI アクセスのデータタイプ	シンボリック PI アクセスに指定可能なデータタイプ
BOOL (%In.x、%IXn.x、%Qn.x.、%QXn.x)	BOOL
BYTE (%IBn、%QBn)	BYTE、SINT、USINT
WORD (%IWn、%QWn)	WORD、INT、UINT
DWORD (%IDn、%PQDn)	DWORD、DINT、UDINT

5.3.4.5 シンボリック PI アクセスの例

シンボリック PI アクセスの例

たとえば、CPU のメモリ領域%IB10 に頻繁にアクセスする必要があり、プログラムの変更に柔軟に対応したい場合、この CPU メモリ領域を使用して次のように *myInput* 変数を宣言します。

```
VAR
 myInput AT %IB10 : BYTE;
END_VAR
```

あるいは、メモリ領域の整数値を使用したい場合は、次のように宣言します。

```
VAR
 myInput AT %IB10 : SINT;
END_VAR
```

後でプログラムで%IB10 とは別の CPU メモリ領域を使用したい場合は、変数宣言の絶対 PI アクセスを変更する必要があるだけです。

5.3.4.6 BackgroundTask の固定プロセスイメージへのアクセス用の I/O 変数の作成

I/O 変数は、詳細ビューのシンボルブラウザで作成します。これを実行するには、オフラインモードで作業する必要があります。

手順の概要を示します。

1. SIMOTION SCOUT のプロジェクトナビゲータで、SIMOTION デバイスのサブツリー内の I/O エレメントを選択します。
2. 詳細ビューで、[Symbol browser]タブを選択し、変数テーブルの末尾(空の行)までスクロールダウンします。
3. テーブルの最後の(空の)行に、以下を入力または選択します。
   - 変数の**[Name:**
   - **[I/O address]**の下で、絶対 PI アクセスの識別子の構文に基づく絶対 PI アクセス (例外: 構文%IXn.x または%QXn.x は、BOOL データタイプには指定できません。
   - I/O 変数の**[Data type]**(シンボリック PI アクセスに指定可能なデータタイプを参照)。
4. オプションで、表示フォーマットを選択します。

この変数には、シンボルブラウザまたは SIMOTION デバイスの任意のプログラムを使用してアクセスできるようになりました。

注記

I/O 変数はオフラインモードでのみ作成できます。SIMOTION SCOUT で I/O 変数を作成し、これらの変数をプログラムソースで使用します。

出力は読み書きできますが、入力は読み取り専用であることに注意してください。

新しいまたは更新された I/O 変数をモニタまたは変更する前に、まずプロジェクトをターゲットシステムにダウンロードする必要があります。

I/O 変数は、その他の変数と同じように使用することができます。

注記

同じ名前のユニット変数またはローカル変数を宣言した場合は(*var-name* など)、I/O 変数を *_device.var-name* (事前定義されたネームスペース)で指定します。

下記も参照

絶対プロセスイメージアクセスの識別子の構文 (ページ 201)
シンボリックプロセスイメージ(PI)アクセスに指定可能なデータタイプ (ページ 202)

5.3.5 I/O 変数へのアクセス

I/O 変数は、その他の変数と同じように使用することができます。

通知
一貫性は基本データタイプについてのみ確保されます。 配列の使用時、データの一貫性を確保する責任はユーザにあります。

注記

同じ名前のユニット変数またはローカル変数を宣言した場合は(*var-name* など)、I/O 変数を *_device.var-name*(事前定義されたネームスペース、「*ネームスペース*」の*事前定義されたネームスペース*の表を参照)。

サイクリックタスクのプロセスイメージを作成した I/O 変数に直接アクセスすることができます。直接アクセスは*_direct.var-name* を使用して指定します。_device.varname

変数へのアクセス中にエラーが発生したときのデフォルト動作を変更する場合は、*[_getSafeValue]*ファンクションと*[_setSafeValue]*ファンクションを使用できます(『*SIMOTION 基本機能*機能マニュアル』を参照)。

I/O 変数へのアクセスに関するエラーの詳細については、『*SIMOTION 基本機能*機能マニュアル』を参照してください。

5.4 ライブラリの使用

ライブラリには、すべての SIMOTION デバイスから使用するユーザ定義タイプ、ファンクション、およびファンクションブロックが提供されています。

ライブラリはすべてのプログラミング言語で作成できます。また、ライブラリはすべてのプログラムソース(たとえば、ST ソースファイル、MCC ユニットなど)で使用できます。

ライブラリの挿入と管理の詳細については、オンラインヘルプを参照してください。

5.4.1 ライブラリのコンパイル

ライブラリではすべての ST コマンドを使用できますが、例外として表にリストされたコマンドは使用できません。また、一部の変数はアクセスが許可されていません。これらの変数もこの表にリストしています。

表 5-31 ライブラリにおける不正な ST コマンドと変数アクセス

禁止されているコマンド • *_getTaskId* ファンクション(*『SIMOTION 基本機能』*機能マニュアルを参照) • *_getAlarmId* ファンクション(*『SIMOTION 基本機能』*機能マニュアルを参照) • *_checkEqualTask* ファンクション(*『SIMOTION 基本機能』*機能マニュアルを参照) • SIMOTION Kernel V3.0 までのバージョンを対象とする以下のファンクション。これらのファンクションを使用すると、設定したメッセージのタスクの名前が転送されます。 – タスク制御コマンド – タスクのランタイム測定用コマンド – メッセージプログラミング用コマンド これらのファンクションを使用すると、設定したメッセージのタスクの名前が転送されます。 • ライブラリがデバイス依存でない場合(すなわち、SIMOTION デバイスまたは SIMOTION Kernel バージョンへの参照なしでコンパイルされている場合): – SIMOTION デバイスのシステムファンクション(SIMOTION デバイスのパラメータマニュアルを参照)。 – バージョン依存のシステムファンクション
禁止されている変数アクセス • ユニット変数(保持型または非保持型) • グローバルデバイス変数(保持型および非保持型) • I/O 変数 • テクノロジーオブジェクトのインスタンスとそのシステム変数 • タスク名および設定したメッセージの変数(*_task* および *_alarm* 名前空間。表*「事前定義された名前空間」*の*「名前空間」*を参照) • ライブラリがデバイス依存でない場合(すなわち、SIMOTION デバイスまたは SIMOTION Kernel バージョンへの参照なしでコンパイルされている場合): – SIMOTION デバイスのシステム変数(SIMOTION デバイスのパラメータマニュアルを参照) – テクノロジーオブジェクトの設定データ(関連する SIMOTION テクノロジーパッケージの設定データのパラメータマニュアルを参照)

ライブラリは次のようにコンパイルします。

1. プロジェクトナビゲータでライブラリを選択します。
2. ライブラリのコンパイルが必要な SIMOTION デバイスおよびテクノロジーパッケージを選択します([Edit|Object properties]メニューコマンド、[TPs/TOs]タブ。『SIMOTION 基本機能』機能マニュアルを参照)。
3. [Save and compile]コンテキストメニューを選択します。

注記

ライブラリでは**プログラムステータス**のデバッグ機能は使用できません。

5.4.2 ライブラリのノウハウ保護

第三者による未許可のアクセスからライブラリとそのソースファイルを保護することができます。このような保護されたライブラリまたはソースファイルは、パスワードを入力しないと開いて表示することができません。

以下のことが可能です。

- ライブラリの個々のソースにノウハウ保護を提供できます。

  ソースだけが未許可のアクセスから保護されます。

  ライブラリをコンパイルする、SIMOTION Kernel のバージョンとテクノロジーパッケージを含めた SIMOTION デバイスの設定は、引き続きユーザが変更および調整できます。『SIMOTION 基本機能』機能マニュアルを参照してください。

  したがって、ユーザはその他の SIMOTION デバイスおよびカーネルバージョンのライブラリを使用することができます。

- ライブラリにノウハウ保護を提供できます。

  その結果、以下のものが未許可のアクセスから保護されます。

  – ライブラリのすべてのソース
  – ライブラリをコンパイルする、SIMOTION Kernel のバージョンとテクノロジーパッケージを含めた SIMOTION デバイスの設定

  したがって、ユーザはその他の SIMOTION デバイスおよびカーネルバージョンのライブラリを使用できなくなります。

  この設定は、これを目的とする場合のみ使用してください。

ノウハウ保護の適用方法については、オンラインヘルプを参照してください。

注記

ノウハウ保護されたソースファイルを通常のテキストファイルとして エクスポートすることはできません。

ただし、XML フォーマットで エクスポートすることはできます。ソースファイルは暗号化されて エクスポートされます。暗号化された XML ファイルをインポートする場合、ログインとパスワードを含め、ノウハウ保護はそのままです。

5.4.3 ライブラリのデータタイプ、ファンクション、およびファンクションブロックの使用

ライブラリのデータタイプ、ファンクション、またはファンクションブロックを使用する前に、それらを ST ソースファイルに認識させる必要があります。これを行うには、ST ソースファイルのインターフェースセクションで以下のコンストラクトを使用します。

```
USELIB library-name [AS namespace];
```

この場合、*library-name* はプロジェクトナビゲータに表示されるライブラリ名です。

複数のライブラリを指定するときは、コンマで区切ったリストとしてライブラリを入力します。例えば次のように入力します。

```
USELIB library-name_1 [AS namespace_1],
    library-name_2 [AS namespace_2],
    library-name_3 [AS namespace_1], ...
```

オプションの*AS namespace*アドオンを使用して名前空間を定義することができます(名前空間 (ページ 208)を参照)。

- その後で、SIMOTION デバイスの ST ソースファイル(PROGRAMS フォルダ内)と同じ名前を持つ、ライブラリのデータタイプ、ファンクション、およびファンクションブロックにアクセスすることができます。
- 異なる名前になるように、名前空間を使用してライブラリのデータタイプ、ファンクション、およびファンクションブロックの名前を変更することもできます。

異なるライブラリに同じ名前空間を割り付けることもできます。

表 5-32　ライブラリを含む名前空間の使用例

```
INTERFACE
    USELIB Bib_1 AS NS_1, Bib_2 AS NS_2;
    PROGRAM Main_Program;
END_INTERFACE

IMPLEMENTATION
    FUNCTION Function1 : VOID
        VAR
            ComID : CommandIdType;
        END_VAR
        ComId := _getCommandId();
    END_FUNCTION

    PROGRAM Main_program
        function1();          // このソースからのファンクション
        NS_1.Var1:=1;
        NS_2.Var1:=2;
        NS_1.function1();     // Bib1 ライブラリからのファンクション
        NS_2.function1();     // Bib2 ライブラリからのファンクション
    END_PROGRAM
END_IMPLEMENTATION
```

5.4.4 名前空間

名前を使用すると、ST ソースファイルの外部で定義されたタイプ、ユニット変数、ファンクション、およびファンクションブロック(ライブラリ内、テクノロジーパッケージ内、SIMOTION デバイス上)にアクセスすることもできます。

コンパイル時に、ST コンパイラは現在の POU から順に識別子を探します。このため、ST ソースファイルで宣言したデータタイプ、変数、ファンクション、またはファンクションブロックにより、ソースファイルの外部で定義された同名の変数などが隠蔽されます。
したがって、デフォルトでは、ST ソースファイルのデータタイプ、変数、ファンクション、またはファンクションブロックにアクセスすることになります。

ただし、ライブラリおよびテクノロジーパッケージのデータタイプ、ユニット変数、ファンクション、およびファンクションブロックにアクセスしたい場合は、名前空間を定義することができます。

```
USELIB library-name_1 [AS lib_namespace_1],
     library-name_2 [AS lib_namespace_2],
     library-name_3 [AS lib_namespace_1], ...
USEPACKAGE tp-name_1 [AS tp_namespace_1],
     tp-name_2 [AS tp_namespace_2],
     tp-name_3 [AS tp_namespace_1], ...
```

名前空間を使用すると、異なるライブラリ内で名前を一貫させることもできます。

ライブラリまたはテクノロジーパッケージのデータタイプ、ファンクション、またはファンクションブロックを使用したい場合、名前の前に名前空間の識別子をピリオドで区切って配置します。たとえば、*namespace.var-name*、 *namespace.fc-name* のように指定します。

デバイス固有変数、プロジェクト固有変数、TaskID 変数、AlarmID 変数には名前空間が事前定義されています。必要な場合、変数名の前にその指定をピリオドで区切って記述します。たとえば、*_device.var-name* または *_task.task-name* のように指定します。

表 5-33　事前定義された名前空間

名前空間	説明
_alarm	AlarmId 用: _alarm.*name* 変数には、*name* 識別子を持つメッセージの AlarmId が含まれます(*『SIMOTION 基本機能』*機能マニュアルを参照)。
_device	デバイス固有の変数用(SIMOTION デバイスのグローバルデバイス変数、I/O 変数、およびシステム変数)
_direct	I/O変数への直接アクセス用。周期的タスクの直接アクセスおよびプロセスイメージ (ページ 195) *を参照。* _device のローカル名前空間。_device._direct.*name* のようなネストができます。
_project	プロジェクト内の SIMOTION デバイスの名前用。他のデバイスのテクノロジーオブジェクトと一緒にのみ使用します。 テクノロジーオブジェクトの一意のプロジェクト規模の名前の場合、この名前とそのシステム変数にも使用されます。
_task	TaskID 用: _task.*name* 変数には、*name* 識別子を持つタスクの TaskId が含まれます(*『SIMOTION 基本機能』*機能マニュアルを参照)。
_to	テクノロジーオブジェクトおよびそのシステム変数と設定データ用(*『SIMOTION 基本機能』*機能マニュアルを参照)。

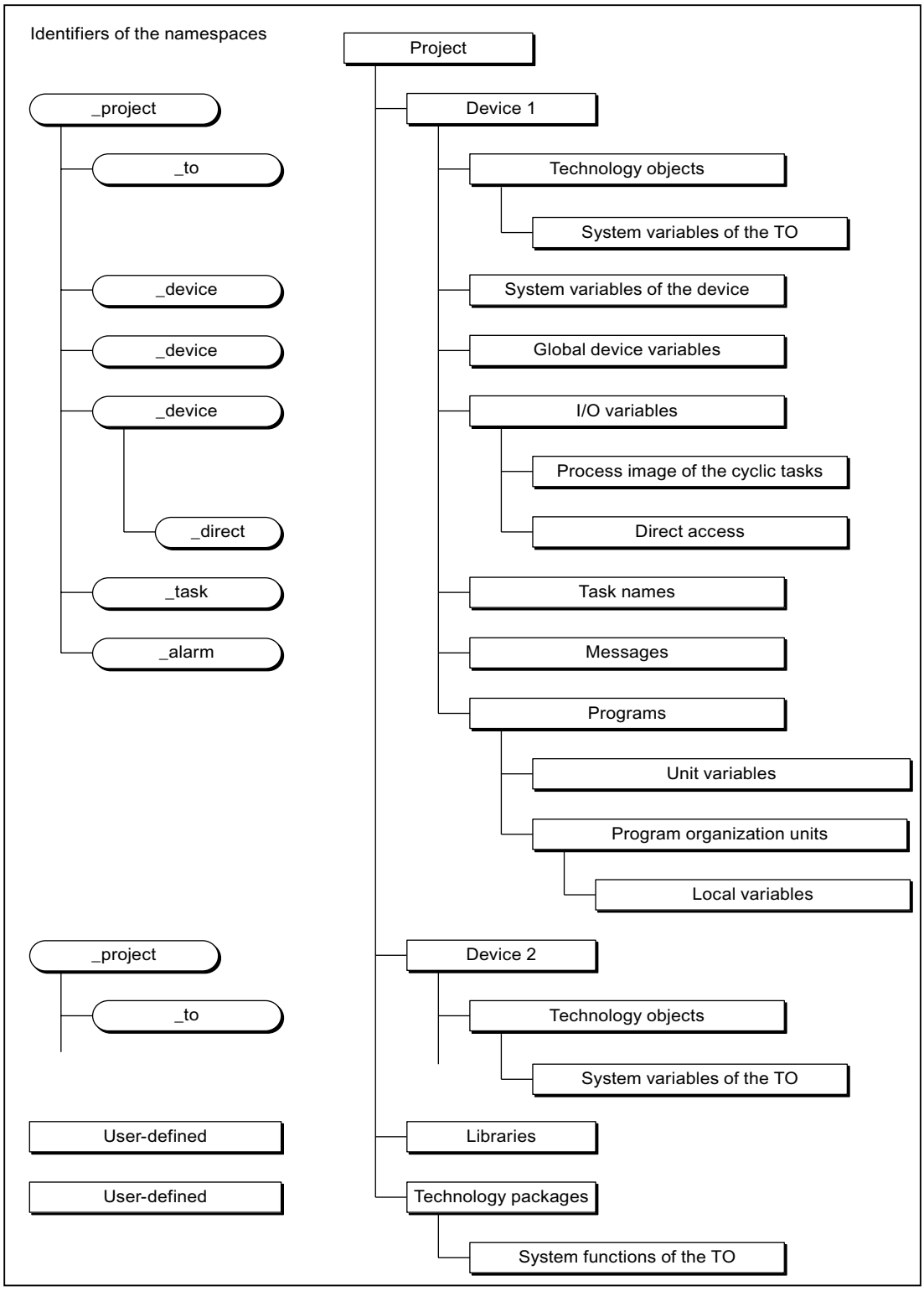


図 5-7　　変数の名前空間および階層

5.5 基準データ

基準データは、以下の概要を提供します。

- 宣言と使用に関する情報を持つ利用識別子
  (クロスリファレンスリスト)
- ファンクション呼び出しとそのネスト
  (プログラム構造)
- プログラムソースのさまざまなデータ領域に関するメモリ要件
  (コード属性)

下記も参照

クロスリファレンスリスト (ページ 210)
Program structure (ページ 212)
コード属性 (ページ 213)

5.5.1 クロスリファレンスリスト

クロスリファレンスリストには、プログラムソース(ST ソースファイル、MCC ソースファイルなど)内のすべての識別子が表示されます。

- 変数、データタイプ、またはプログラム構成ユニット(プログラム、ファンクション、ファンクションブロック)として宣言
- 以前に定義されたタイプ識別子として宣言で使用
- プログラム構成ユニットのステートメントセクションの変数として使用

必要に応じて以下についてクロスリファレンスリストを生成することができます。

- 個々のプログラムソース(ST ソースファイル、MCC ソースファイル、LAD/FBD ソースなど)
- SIMOTION デバイスのすべてのプログラムソース
- プロジェクトのプログラムソースとライブラリ
- ライブラリ(すべてのライブラリ、1 つのライブラリ)

5.5.1.1 クロスリファレンスリストの作成

クロスリファレンスリストを作成するには

1. プロジェクトナビゲータで、クロスリファレンスリストを作成する工レメントを選択します。
2. メニュー**[Edit|Reference data|Create]**を選択します。

クロスリファレンスリストは、詳細ビューのクロスリファレンスリストのタブに表示されます。

5.5.1.2 クロスリファレンスリストの内容

作成されたクロスリファレンスリストには、各識別子について以下が表示されます。

- 識別子名(構造体と列挙の場合は、個々のコンポーネントと工レメントも表示)
- タイプ(データタイプ、POU タイプ)
- 宣言場所(プログラムソースの名前、テクノロジーパッケージの名前など)
- 識別子の現在の使用に関する情報:
  - 使用のタイプ(R = 読み取りアクセス、W = 書き込みアクセス、変数タイプ = 宣言)
  - プログラムソースのパス詳細(SIMOTION デバイス、プログラムソースの名前)
  - プログラムソースの領域(実装セクション、POU 名)
  - プログラムソースのプログラム言語
  - ST ソースの行番号(または MCC チャートのブロック番号、または LAD/FBD ソースの参照番号)

注記

生成されたクロスリファレンスリストは自動的に保存され、プロジェクトナビゲータで適切な工レメントを選択すると選択して表示することができます。クロスリファレンスリストを表示するには、**[Edit|Reference data|Display|Cross-Reference List]**メニューコマンドを選択します。

クロスリファレンスリストは作成されると、選択的に更新されます(プロジェクトナビゲータで選択された工レメントに基づく)。もしあれば、他の既存のクロスリファレンスデータが保持され、表示されます。

5.5.1.3 クロスリファレンスリストでの作業

クロスリファレンスリストでは、以下の作業ができます。

- 列の内容をアルファベット順に並べ替える
- フィルタ機能を設定する(マウスの右ボタンで呼び出すコンテキストメニュー経由)
- 内容をクリップボードにコピーして、たとえばスプレッドシートプログラムに貼り付ける
- 内容を印刷する
- 参照されたプログラムソースを開いて、ST コマンド(または MCC エレメントか LAD/FBD エレメント)の関連する行にカーソルを置く。
  - クロスリファレンスリストの対応する行をダブルクリックする。

    または

  - カーソルをクロスリファレンスリストの対応する行に置いて、**[Go to application]**ボタンをクリックする。

クロスリファレンスリストの詳細については、オンラインヘルプを参照してください。

5.5.2 Program structure

プログラム構造には、選択したエレメント内のすべてのファンクション呼び出しとそのネストが表示されます。

クロスリファレンスリストが正しく作成されると、以下についてプログラム構造を選択して表示することができます。

- 個々のプログラムソース(ST ソースファイル、MCC ソースファイル、LAD/FBD ソースなど)
- SIMOTION デバイスのすべてのプログラムソース
- プロジェクトのプログラムソースとライブラリ
- ライブラリ(すべてのライブラリ、1 つのライブラリ、ライブラリ内の個々のプログラムソース)

以下の手順に従います。

1. プロジェクトナビゲータで、プログラム構造を表示するエレメントを選択します。
2. メニュー**[Edit|Reference data|Display|Program structure]**を選択します。

   [Cross references]タブは、詳細ビューでは[Program structure]タブに換わります。

5.5.2.1 プログラム構造の内容

以下を表示するツリー構造が現れます。

- それぞれ基本として
  - プログラムソースで宣言されたプログラム構成ユニット(プログラム、ファンクション、ファンクションブロック)、または
  - 使用される実行システムタスク
- その下に、このプログラム構成ユニットまたはタスクで参照されるサブルーチン

エントリの構造については、以下の表を参照してください。

表 5-34　　プログラム構造の表示エレメント

エレメント	説明
基本 (宣言された POU または使用されるタスク)	リストはカンマで区切られます。 • プログラム構成ユニット(POU)またはタスクの識別子 • POU またはタスクが宣言されたプログラムソースのアドオン[UNIT]を伴う識別子 • 最大および最小スタック要件(ローカルデータスタック上の POU またはタスクのメモリ要件)、バイト単位[Min, Max] • 最大および最小スタック全体の要件(すべての呼び出された POU を含むローカルデータスタック上の POU またはタスクのメモリ要件)、バイト単位[Min, Max]
参照された POU	リストはカンマで区切られます。 • 呼び出された POU の識別子 • オプション: POU が宣言されたプログラムソース/テクノロジーパッケージの識別子: アドオン(UNIT): ユーザ定義のプログラムソース アドオン(LIB): ライブラリ アドオン(TP): テクノロジーパッケージからのシステムファンクション • ファンクションブロック専用: インスタンスの識別子 • ファンクションブロック専用: インスタンスが宣言されたプログラムソースの識別子: アドオン(UNIT): ユーザ定義のプログラムソース アドオン(LIB):ライブラリ • POU が呼び出された(コンパイル済み)ソースの行。複数の行は「|」で区切られます。

5.5.3　コード属性

コード属性で、プログラムソースのさまざまなデータ領域のメモリ要件についての情報を検索することができます。

クロスリファレンスリストが正しく作成されていれば、コード属性を以下について選択して表示することができます。

- 個々のプログラムソース(ST ソースファイル、MCC ソースファイル、LAD/FBD ソースなど)
- SIMOTION デバイスのすべてのプログラムソース
- プロジェクトのプログラムソースとライブラリ
- ライブラリ(すべてのライブラリ、1 つのライブラリ、ライブラリ内の個々のプログラムソース)

以下の手順に従います。

1. プロジェクトナビゲータで、コード属性を表示するエレメントを選択します。
2. **[Edit|Reference data|Display|Code attributes]メニューを選択します。**

   [Cross references]タブは、詳細ビューでは**[Code attributes]**タブに換わります。

5.5.3.1 コード属性の内容

すべての選択されたプログラムソースファイルのテーブルに、以下のものが表示されます。

- プログラムソースファイルの識別子
- プログラムソースファイルの以下のデータ領域に関するメモリ要件
  - **[Dynamic data]**: すべてのユニット変数(保持性および非保持性、インターフェースセクションと実装セクション内)
  - **[Retain data]**: インターフェースセクションと実装セクション内の保持性ユニット変数
  - **Interface data**: インターフェースセクション内のユニット変数(保持性および非保持性)
- 参照されたソースの数

5.6 プラグマによるプリプロセッサおよびコンパイラの制御

プラグマは、ST ソースファイルのコンパイルに影響を及ぼす ST ソースファイルテキスト(ステートメントなど)を挿入するために使用します。

プラグマは中括弧{と}で囲み、中に
を含めることができます(図を参照)。

- プリプロセッサを制御するためのプリプロセッサステートメント
  (プリプロセッサの制御 (ページ 215)を参照)
- コンパイラを制御するためのコンパイラオプションの属性
  (属性によるコンパイラ出力の制御 (ページ 219)を参照)

ST ソースファイルに含まれるプラグマは、プリプロセッサまたはコンパイラによって評価され、制御ステートメントとして解釈されます。

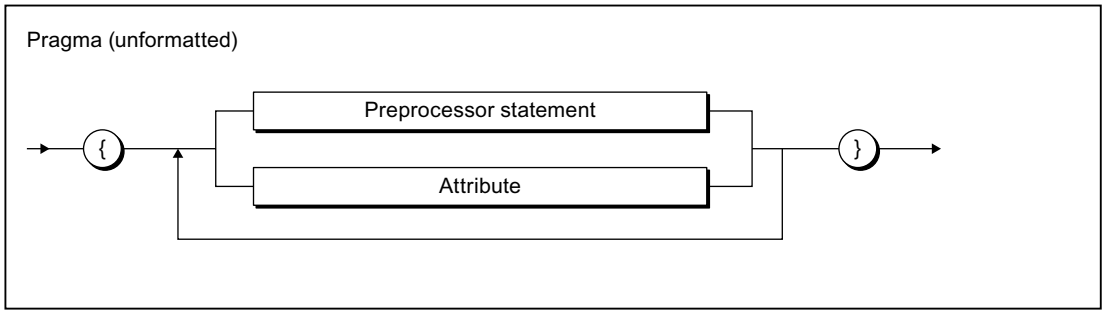


図 5-8 プラグマ構文

注記

必ず正しいプラグマ構文を使用してください(たとえば、属性の大文字および小文字表記など)。

認識されないプラグマは、警告メッセージなく無視されます。

5.6.1 プリプロセッサの制御

プリプロセッサは、コンパイルのために ST ソースファイルを準備します。たとえば、文字列を識別子の置換テキストとして定義したり、ソースプログラムのセクションをコンパイルのために非表示/表示したりすることができます。

デフォルトでは、プリプロセッサは無効にされています。STコンパイラの設定を使用して有効にすることができます(STコンパイラの設定 (ページ 30)を参照)。

- すべてのプロジェクトに対してグローバルに有効化
- 単一の ST ソースファイルに対してローカルに有効化

警告クラス 7 を有効にすると、プリプロセッサの作業に関する情報が表示されます(STコンパイラの設定 (ページ 30)、警告クラスの意味 (ページ 34)、および 属性によるコンパイラ出力の制御 (ページ 219)を参照)。

下記も参照

STコンパイラのローカル設定 (ページ 32)

5.6.1.1 プリプロセッサステートメント

プリプロセッサは、プラグマ (ページ 214)のステートメントを使用して制御することができます。以下の構文ダイアログで指定されたステートメントを使用できます。これらのステートメントは、STソースファイルのその後のすべての行で動作します。

ステートメントは、SIMOTION デバイスまたはライブラリの ST ソースファイルで使用することができます。

STソースファイルの特性ダイアログボックスで、プリプロセッサの定義を作成することができます(プリプロセッサ定義の作成 (ページ 35)を参照)。これにより、ノウハウ保護されたSTソースファイルを使用してプリプロセッサの制御もできるようになります(STソースファイルのノウハウ保護 (ページ 35)を参照)。

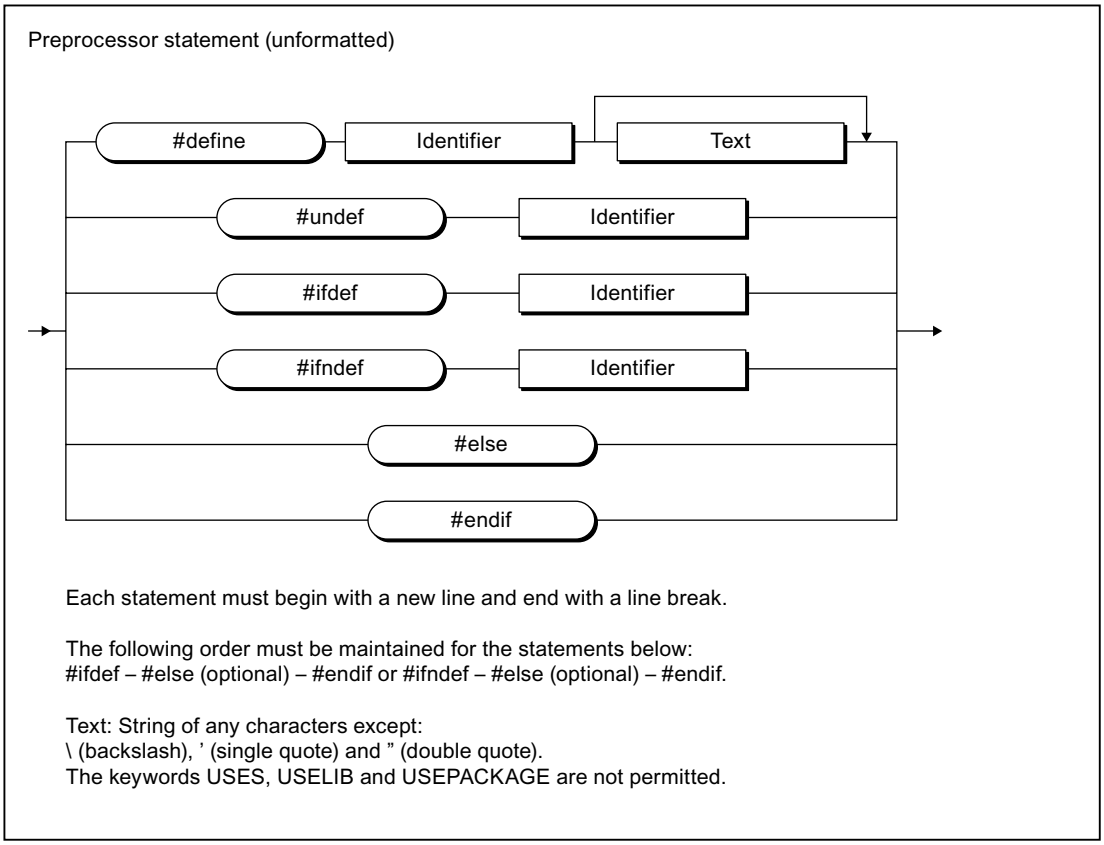


図 5-9　　プリプロセッサステートメントの構文

表 5-35　　プリプロセッサステートメント

ステートメント	意味
#define	指定した識別子が指定したテキストによって下位で置換されます。
#undef	識別子の置換ルールが取り消されます。
#ifdef	変形フォーメーション(条件付きコンパイル)用 指定した識別子を定義する場合、その後のプログラム行(#else または#endif を含む次のプラグマまで)がコンパイラによってコンパイルされます。
#ifndef	変形フォーメーション(条件付きコンパイル)用 指定した識別子を定義**しない**場合、その後のプログラム行(#else または#endif を含む次のプラグマまで)がコンパイラによってコンパイルされます。
#else	変形フォーメーション(条件付きコンパイル)用 #ifdef または#ifndef への代替分岐 #ifdef または#ifndef による先行問い合わせの条件が満たされなかった場合、その後のプログラム行(#endif を含む次のプラグマまで)がコンパイラによってコンパイルされます。
#endif	#ifdef または#ifndef による変形フォーメーションを終了します。

注記

#define ステートメントを含むプラグマの場合、以下の点に注意してください。

ST ソースファイルのインターフェースセクションにある、#define ステートメントを含むプラグマはエクスポートされます。USES ステートメントを使用すると、定義した識別子を同じ SIMOTION デバイスまたは同じライブラリの他の ST ソースファイルにインポートすることができます。

ライブラリのプラグマで定義した識別子を SIMOTION デバイスの ST ソースファイルにインポートすることはできません。

予約識別子の再定義はできません。

ST ソースの特性ダイアログボックスでプリプロセッサ定義を作成することもできます。同じ識別子で定義が異なる場合、ST ソースファイル内の#define ステートメントが優先します。

5.6.1.2 プリプロセッサステートメントの例

表 5-36 プリプロセッサステートメントの例

```
INTERFACE
    FUNCTION_BLOCK fb1;
    VAR_GLOBAL
        g_var     : int;
    END_VAR

{
#define my_define g_var
#define my_call f(my_define)
}
END_INTERFACE

IMPLEMENTATION

FUNCTION f :[Int]
    VAR_INPUT
        i: INT;
    END_VAR
    f := i;
END_FUNCTION

FUNCTION_BLOCK fb1
    VAR_INPUT
        i_var : int;
    END_VAR
    VAR_OUTPUT
        o_var : int;
    END_VAR
    my_define := i_var;     // プロプロセッサの後:
                            // g_var := i_var
{
#undef my_define
}
    o_var := my_call +1;    // プリプロセッサの後:
                            // o_var := f(g_var) + 1;
{
#ifdef my_define
}
    my_define := i_var;     // プロプロセッサの後:
                            // my_define が定義されていない。
{
#endif
}
END_FUNCTION_BLOCK

END_IMPLEMENTATION
```

5.6.2 属性によるコンパイラ出力の制御

プラグマ (ページ 214)内の属性を使用すると、コンパイラ出力を制御することができます。

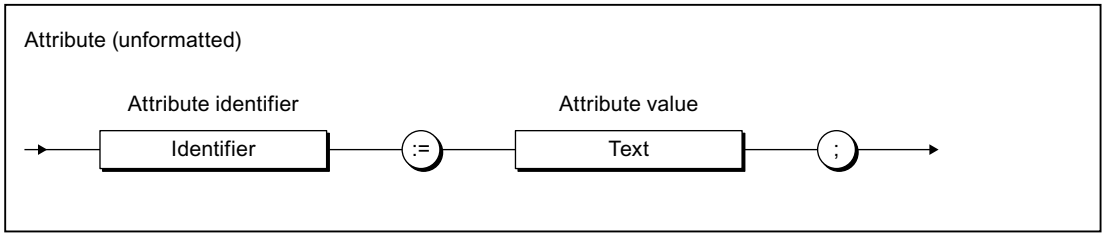


図 5-10 コンパイラ出力の属性の構文

表 5-37 コンパイラ出力で使用できる属性

属性識別子	属性値	意味
_U7_PoeBld_CompilerOption	この属性は、ST ソース内のコンパイラ警告の出力に影響します。これは、ST ソースのその後のすべての行に影響します。	
	warning:*n*:off	番号 *n* で指定した警告は表示されません。
	warning:*n*:on	番号 *n* で指定した警告が表示されます。
	n の許容される値 *n* = 0 ~ 7: 警告クラス(警告クラスの意味 (ページ 34)を参照) *n* = 16000 以上: 警告の番号	
HMI_Export	この属性は、HMI デバイスで使用可能なユニット変数をデフォルトで変更します。この変数は、VAR_GLOBAL または VAR_GLOBAL RETAIN 宣言ブロックの関連キーワードの直後に配置する必要があります。これは、関連する宣言ブロックで宣言したユニット変数にのみ影響します。 変数およびHMIデバイス (ページ 191)を参照	
	FALSE	ST ソースのインターフェースセクション内。関連する宣言ブロックで宣言したユニット変数を HMI デバイスで使用することは**できません**。
	TRUE	ST ソースの実装セクション内。関連する宣言ブロックで宣言したユニット変数を HMI デバイスで使用することができます。

通知
属性には、必ず正しい大文字および小文字表記を使用してください!

表 5-38　　コンパイラオプションの属性の例

```
INTERFACE
VAR_GLOBAL
    { HMI_Export := FALSE; }
    x : DINT;
END_VAR
FUNCTION_BLOCK fb1;
END_INTERFACE

IMPLEMENTATION
VAR_GLOBAL
    { HMI_Export := TRUE; }
    y : DINT;
END_VAR
FUNCTION_BLOCK fb1
    VAR_INPUT
        i_var : int;
    END_VAR
    VAR_OUTPUT
        o_var : int;
    END_VAR
    { _U7_PoeBld_CompilerOption := warning:2:on; }
    o_var := REAL_TO_INT(1.0);        // 警告 16004
    { _U7_PoeBld_CompilerOption := warning:2:off; }
    o_var := REAL_TO_INT(1.0);        // 警告 16004 なし
    { _U7_PoeBld_CompilerOption := warning:16004:on; }
    o_var := REAL_TO_INT(1.0);        // 警告 16004
    { _U7_PoeBld_CompilerOption := warning:16004:off; }
    o_var := REAL_TO_INT(1.0);        // 警告 16004 なし
    { _U7_PoeBld_CompilerOption := warning:2:off;
     _U7_PoeBld_CompilerOption := warning:16004:on; }
    o_var := REAL_TO_INT(1.0);        // 警告 16004
END_FUNCTION_BLOCK

END_IMPLEMENTATION
```

5.7 ジャンプステートメントおよびラベル

制御ステートメント(制御ステートメント (ページ 114)を参照)の他に、ジャンプステートメントを使用することもできます。

GOTO ステートメントを使用してジャンプステートメントをプログラミングし、ジャンプ先のジャンプラベルを指定します。ジャンプは POU 内でのみ使用できます。

プログラムを再開させるステートメントの前に(コロンで区切って)ジャンプラベルを入力します。

あるいは、POU でジャンプラベルを宣言することもできます(POU の構造 LABEL/END_LABEL を使用)。そうすると、宣言したジャンプラベルだけをステートメントセクションで使用することができます。

ジャンプステートメントおよびラベルの構文

表 5-39　ジャンプステートメントの構文の例

```
FUNCTION func : VOID
    VAR
        x, y, z BOOL;
    END_VAR
    LABEL
        lab_1, lab_2;     // ジャンプラベルの宣言
    END_LABEL
    x := y;
    lab_1 : y := z;        // ステートメントを持つジャンプラベル
    IF x = y THEN
        GOTO lab_2;        // ジャンプステートメント
    END_IF;
    GOTO lab_1;            // ジャンプステートメント
    lab_2 : ;              // ステートメントが空のジャンプラベル
END_FUNCTION
```

注記

GOTO ステートメントは、特別な場合(たとえば、トラブルシューティングのためなど)にのみ使用してください。構造化プログラミングのルールによっては、絶対に使用しないでください。

ジャンプは POU 内でのみ使用できます。

以下のジャンプは不正です。

- 下位の制御構造(WHILE、FOR など)へのジャンプ
- WAITFORCONDITION 構造からのジャンプ
- CASE ステートメント内でのジャンプ

ジャンプラベルは、ジャンプラベルを使用する POU でのみ宣言できます。ジャンプラベルを宣言した場合、宣言したジャンプラベルだけを使用できます。

エラーソースおよびプログラムのデバッグ 6

この章では、プログラミングエラーのさまざまな原因を説明し、効率的にプログラミングを行う方法を示します。プログラムのテストで使用できるオプションについても学習します。すべての起こり得るコンパイラエラーメッセージ、すなわちコンパイラエラーは、コンパイラエラーメッセージと修正方法 (ページ 305)を参照してください。可能な対応と修正方法をエラーごとに記載しています。

6.1 エラーの回避と効率的なプログラミングに関する注意事項

*『SIMOTION 基本機能』*機能マニュアルには、コンパイラを妨害したり、プログラムの適切な実行を妨げたりする、いくつかの一般的なエラーソースをリストしています。たとえば以下に関する注意事項を挙げています。

- 演算式を割り付けるためのデータタイプ
- 周期的タスクにおける起動ファンクション
- 周期的タスクにおける待機時間
- ダウンロード時のエラー
- CPU が RUN に切り替わらない
- CPU が STOP に移行する
- ローカルデータスタックのサイズ
- その他

さらに、効率的なプログラミングに関する注意事項(特に以下の場合について)も挙げています。

- ランタイム指向プログラミング
- 変更最適化プログラミング

6.2 プログラムのデバッグ

コンパイル手順中は、ST コンパイラによって構文エラーが検出され、表示されます。プログラムの実行時にランタイムエラーがあると、システムアラームによって表示されるか、動作モードが STOP になります。論理プログラミングエラーは、ST のテストファンクション、たとえばシンボルブラウザ、ステータスプログラム、トレースなどを使用して検出することができます。

テストファンクションを使用して以下に示すのと同じ結果を実現するには、サンプルプログラムの作成 (ページ 43)のサンプルプログラムを使用することをお勧めします。

6.2.1 プログラムのテストモード

6.2.1.1 SIMOTION デバイスのモード

さまざまな SIMOTION デバイスモードが、プログラムテストで使用できます。

SIMOTION デバイスのモードを選択するには

1. プロジェクトナビゲータで SIMOTION デバイスを強調表示します。
2. コンテキストメニューから**[Test mode]**を選択します。
3. 必要なモード(表を参照)を選択し、**[OK]**で確定します。

デバッグモードを選択した場合は、さらに入力するよう指示されます。関連セクション: 「デバッグモードに関する重要な情報」

表 6-1 SIMOTION デバイスのモード

設定	意味
Process mode	SIMOTION デバイスでのプログラム実行は、最大システムパフォーマンスに最適化されています。 以下の診断ファンクションが利用できます。 • シンボルブラウザまたはウォッチテーブルでのモニタ変数 • プログラムステータス • ドライブとファンクションジェネレータ用の測定機能付きトレースツール(オンラインヘルプを参照) システムパフォーマンスを最適化するために、以下の制限事項が適用されます。 • プログラムステータスの場合: – SIMOTION Kernel のバージョン V4.0 現在: **タスク当たり**モニタ可能なのは 1 つのプログラムソース(ST ソース、MCC ソース、LAD/FBD ソースなど)のみです。 – SIMOTION Kernel のバージョン V3.2 まで: モニタ可能なのは 1 つのプログラムソース(ST ソース、MCC ソース、LAD/FBD ソースなど)のみです。 • トレースの場合: 各 SIMOTION デバイスで 1 つのトレースのみ可能
Test mode	診断ファクション(プロセスモードを参照)がすべて使用できます。 • プログラムステータスの場合: – SIMOTION Kernel のバージョン V4.1 現在: 1 つのタスク当たり複数のプログラムソース(ST ソース、MCC ソース、LAD/FBD ソースなど)をモニタ可能 – SIMOTION Kernel のバージョン V4.0 まで: **タスク当たり**モニタ可能なのは 1 つのプログラムソース(ST ソース、MCC ソース、LAD/FBD ソースなど)のみです。 • トレースの場合: SIMOTION デバイスで **5 つ**以上のトレースは不可能 **注** ランタイムとメモリ利用率は、診断ファンクションの使用が増えるにつれ増加します。
Debug mode	このモードは SIMOTION Kernel V3.2 以降で利用可能です。 テストモードの診断ファンクションのほかに、以下の機能を使用することができます。 • ブレークポイント Debug mode では、プロジェクトの複数の SIMOTION デバイスを切り替えることはできません。 **Debug mode に関する次の注意事項に留意してください。**

6.2.1.2 デバッグモードに関する重要な情報

⚠ 警告

適切な安全規則に従う必要があります。

適度に短いモニタ時間で有効にされたライフサインモニタファンクションでのみ、デバッグモードを使用してください。

そうしないと、PC と SIMOTION デバイス間で通信リンク問題が発生した場合、軸は制御不能な動作を開始する可能性があります。

このファンクションは、試運転、診断、サービス目的でのみリリースされています。通常このファンクションは、認定技術者だけが使用すべきです。上位レベルの制御を安全にシャットダウンしても、効果はありません。

「スペースキーによる緊急停止」機能は、動作モードのすべてで保証されているわけではありません。したがって、ハードウェアに緊急停止回路を設置する必要があります。適切な対策が、ユーザにより実施される必要があります。

安全対策注意事項の確認

デバッグモードを選択した後、安全対策注意事項を確認する必要があります。また、ライフサインモニタと緊急停止をパラメータ設定することができます。

以下のように実行します。

1. **[Settings]**ボタンをクリックします。

   [Emergency stop]パラメータ設定ウィンドウが開きます。

2. 次のセクションで説明しているとおり、このウィンドウで安全対策注意事項を読み、ライフサインモニタと緊急停止をパラメータ設定します。

ライフサインモニタと緊急停止のパラメータ設定

[Emergency stop]パラメータ設定ウィンドウで、以下に説明するとおりに操作します。

1. 警告をよく読みます。
2. **[Safety notes]**ボタンをクリックして、詳細な安全対策注意事項のウィンドウを開きます。
3. ライフサインモニタのデフォルトには何も変更をしないでください。

   変更は特別な場合にのみ行うものであり、すべての危険についての警告をよく認識する必要があります。

4. **[Accept]**をクリックして、安全対策注意事項を読み、ライフサインモニタを正しくパラメータ設定したことを確認します。

通知

スペースキーを押すと、デバッグモードでは緊急停止と解釈されます。

緊急停止は、別の Windows アプリケーションに切り替えるとき常にトリガされます。

6.2.1.3 [emergency stop parameterization]パラメータ

表 6-2 [emergency stop parameterization]パラメータの説明

フィールド	説明
サインオブライフモニタ	SIMOTION デバイスおよび SIMOTION SCOUT は、定期的にサインオブライフ信号を交換して、接続が正しく機能していることを確認します。設定した監視時間よりも長い間サインオブライフ信号が中断されると、SIMOTION デバイスは STOP 状態になります。 • **[Active]**チェックボックス: このチェックボックスを選択すると、サインオブライフモニタが有効になります。 • [Monitoring time]: タイムアウトを入力します。 **注意点** できれば、ライフサインモニタのデフォルトは変更しないでください。 特殊な状況、およびすべての危険警告を守った上でのみ、変更は行ってください。
安全に関する情報	**警告を守ってください!** 追加の安全に関する情報を取得するには、ボタンをクリックします。

6.2.2 シンボルブラウザ

6.2.2.1 シンボルブラウザの特徴

シンボルブラウザでは、名前、データタイプ、および変数値を表示し、必要な場合はこれらを変更することができます。特に、 以下の変数を確認することができます。

- プログラムまたはファンクションブロックのユニット変数およびスタティック変数
- SIMOTION デバイスまたはテクノロジーオブジェクトのシステム変数
- I/O 変数またはグローバルデバイス変数

これらの変数に対して、以下のことを実行できます。

- 変数値のスナップショットの表示
- 変数値が変化する際の変数値の監視
- 変数値の変更(修正)

ただし、ターゲットシステムにプロジェクトがロードされていて、ターゲットシステムへの接続が確立されている場合、シンボルブラウザでは変数値を表示/変更できるだけです。

6.2.2.2 シンボルブラウザの使用

必要条件

- ターゲットシステムへの接続が確立されていて、ターゲットシステムにプロジェクト(サンプルプログラムを含むプロジェクトなど)がダウンロードされていることを確認してください。サンプルプログラムの実行 (ページ 50)を参照してください。
- ユーザプログラムを実行できますが、ユーザプログラムを実行する必要はありません。プログラムを実行しない場合、変数の初期値だけが表示されます。

手順は、監視する変数を格納するメモリ領域に依存します。変数タイプのメモリ範囲 (ページ 178)を参照してください。

ユニットのユーザメモリの変数

シンボルブラウザを使用して、ユニットのユーザメモリに含まれる、たとえば以下の変数を監視することができます。

- プログラムソースファイル(ユニット)のインターフェースセクションのユニット変数
- プログラムソースファイル(ユニット)の実装セクションのユニット変数
- インスタンスがユニット変数として宣言されているファンクションブロックのスタティック変数

以下の手順に従ってください。

1. プロジェクトナビゲータでプログラムソースファイルを選択します(たとえば、ST_1)。
2. 詳細ビューで[Symbol browser]タブをクリックします。

   シンボルブラウザに、ユニットのユーザメモリに含まれるすべての変数が表示されます。

タスクのユーザメモリの変数

シンボルブラウザを使用して、関連タスクのユーザメモリに含まれる、たとえば以下の変数を監視することができます。

- プログラムのスタティック変数
- インスタンスがプログラムのスタティック変数として宣言されているファンクションブロックのスタティック変数

以下の手順に従ってください。

1. SIMOTION SCOUT のプロジェクトナビゲータで、SIMOTION デバイスのサブツリーの **EXECUTION SYSTEM** 要素を選択します。
2. 詳細ビューで**[Symbol browser]**タブをクリックします。

   シンボルブラウザに、割り付け済みプログラムを含むすべてのタスクが表示されます。タスクのユーザメモリに含まれる変数が下に表示されます。

注記

ステータスプログラムを使用して、(ユニット変数とスタティック変数と一緒に)テンポラリ変数を監視することができます(プログラムステータスの特徴 (ページ 233)を参照)。

システム変数およびグローバルデバイス変数

シンボルブラウザでは、以下の変数を監視することもできます。

- SIMOTION デバイスのシステム変数
- テクノロジーオブジェクトのシステム変数
- I/O 変数
- グローバルデバイス変数

以下の手順に従ってください。

1. SIMOTION SCOUT のプロジェクトナビゲータで該当する要素を選択します。
2. 詳細ビューで**[Symbol browser]**タブをクリックします。

   対応する変数がシンボルブラウザに表示されます。

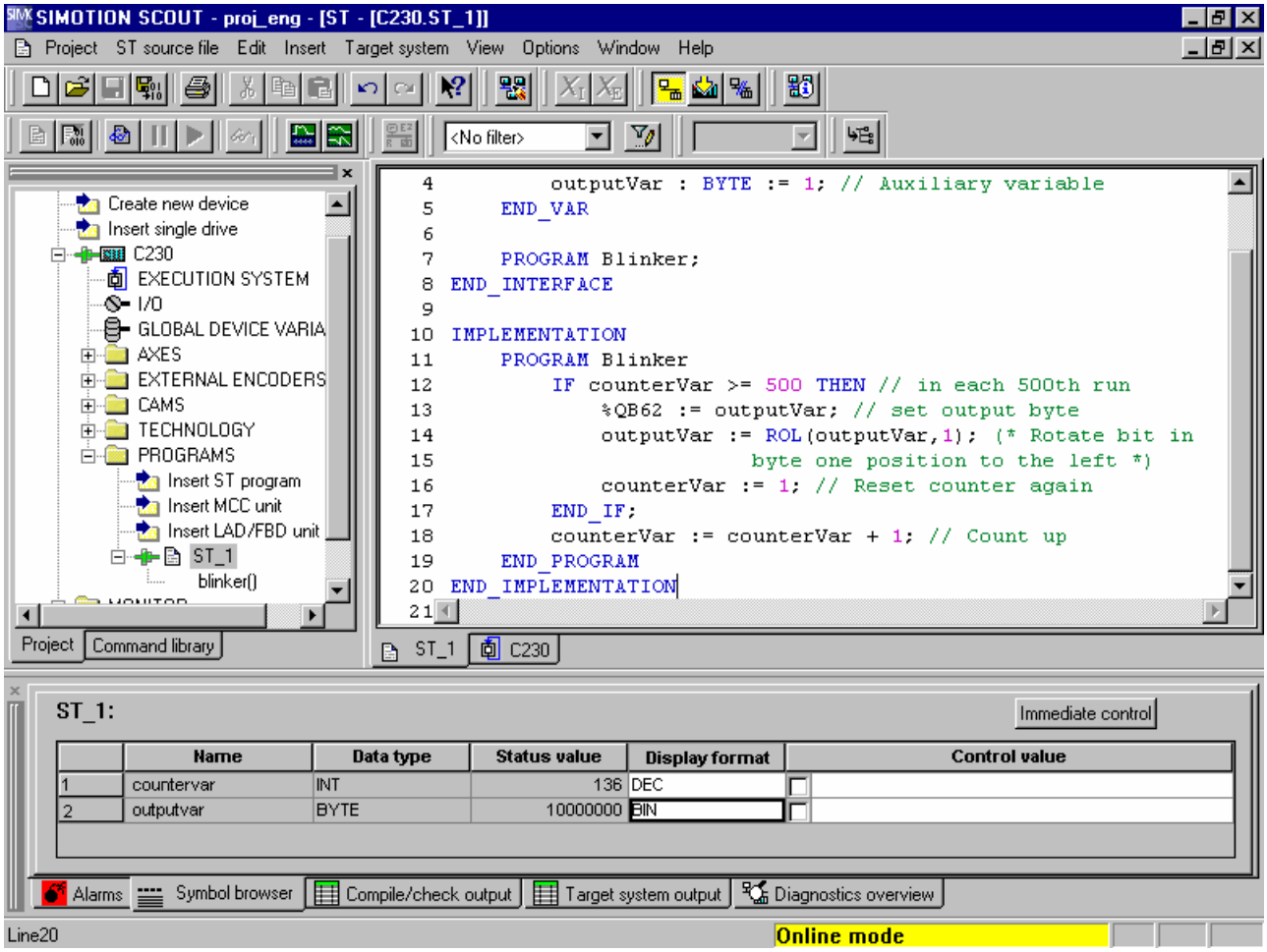


図 6-1　　シンボルブラウザを使用した変数の内容の表示

ステータス、および変数の制御

[Status value]列には、現在の変数値が表示されます。これは定期的に更新されます。

1 つまたは複数の変数の値を変更することができます。変数を変更するには、以下の手順に従ってください。

1. [Control value]列に値を入力します。
2. この列のチェックボックスを有効にします。
3. [Immediate control]ボタンをクリックします。

入力した値が、選択した変数に書き込まれます。

通知

複数の変数の値を変更するときは、以下の点に注意してください。

値は変数に順次書き込まれます。次の値が書き込まれるまでには数ミリ秒かかることがあります。シンボルブラウザウでは変数は上から下に変更されます。したがって、一貫性が保たれる保証はありません。

シンボルブラウザの表示の固定

有効なオブジェクトのシンボルブラウザの表示を固定することができます。

- これを行うには、シンボルブラウザの右上隅にある[Retain display]アイコンをクリックします。表示されるシンボルが に変わります。

  このオブジェクトの変数は、プロジェクトナビゲータで別のオブジェクトを選択しても、引き続きシンボルブラウザに表示され、更新されます。

- 表示を削除するには、もう一度 アイコンをクリックします。表示されるシンボルが に戻ります。

6.2.3 ウォッチテーブルでの変数の監視

6.2.3.1 ウォッチテーブルの変数

シンボルブラウザでは、プロジェクト内のオブジェクトの変数だけが表示されます。プログラムステータスでは、自由に選択可能な監視領域内の ST ソースファイルの変数だけが表示されます。

それに対して、ウォッチテーブルでは、さまざまなソース(たとえば、プログラムソース、テクノロジーオブジェクト、SINAMICS ドライブなど。異なるデバイス上であっても可)から選択した変数をグループとして監視することができます。

変数のデータタイプはオフラインモードで確認できます。変数の値はオンラインモードで表示および変更できます。

6.2.3.2 ウォッチテーブルの使用

ウォッチテーブルでさまざまなプログラムソース、テクノロジーオブジェクト、SIMOTIONデバイスなど(異なるデバイス上であっても可)からの変数をグループ化し、それらをまとめて監視し、必要な場合は変更することができます。

ウォッチテーブルの作成

ウォッチテーブルを作成し、変数を割り付ける手順

1. プロジェクトナビゲータで、[Monitor]フォルダを選択します。
2. [Insert|Watch table]を選択してウォッチテーブルを作成し、ウォッチテーブルの名前を入力します。[Monitor]フォルダにこの名前のウォッチテーブルが表示されます。
3. プロジェクトナビゲータで、ウォッチテーブルに変数を移動するオブジェクトをクリックします。
4. シンボルブラウザで、左側の列で番号をクリックして、対応する変数行を選択します。
5. コンテキストメニューから、[Move variable to watch table]項目と、該当するウォッチテーブル(たとえば、Watch table_1)を選択します。
6. ウォッチテーブルをクリックすると、[Watch table]タブの詳細ビューに、選択した変数がウォッチテーブル内にあることが表示されます。
7. 手順 3~6 を繰り返して、さまざまなオブジェクトの変数を監視します。

ターゲットシステムと接続している場合、変数の内容を監視することができます。

ステータス、および変数の制御

[Status value]列には、現在の変数値が表示されます。これは定期的に更新されます。

1 つまたは複数の変数の値を変更することができます。変数を変更するには、以下の手順に従ってください。

1. [Control value]列に値を入力します。
2. この列のチェックボックスを有効にします。
3. [Immediate control]ボタンをクリックします。

入力した値が、選択した変数に書き込まれます。

通知

複数の変数の値を変更するときは、以下の点に注意してください。

値は変数に順次書き込まれます。次の値が書き込まれるまでには数ミリ秒かかることがあります。ウォッチテーブルでは変数は上から下に変更されます。したがって、一貫性が保たれる保証はありません。

ウォッチテーブルの表示の固定

有効なウォッチテーブルの表示を固定することができます。

- これを行うには、詳細ビューの[Watch table]タブの右上隅にある[Retain display]アイコンをクリックします。表示されるシンボルが に変わります。

  このウォッチテーブルは、プロジェクトナビゲータで別のウォッチテーブルを選択しても、引き続き表示されます。

- 表示を削除するには、もう一度 アイコンをクリックします。表示されるシンボルが に戻ります。

6.2.4 プログラム実行

6.2.4.1 プログラム実行: コード位置と呼び出しパスの表示

MotionTask が現在実行しているコード位置(たとえば、ST ソースファイルの行)をその呼び出しパスと共に表示することができます。

以下の手順に従ってください。

1. **[Program run]**ツールバーの**[Show program run]**ボタンをクリックします。

   開かれるウィンドウで、必要な MotionTask を選択します。

2. ウィンドウを開いたままにします。

   [ST editor]ツールバーの**[Update]**ボタンをクリックします。

ウィンドウに以下の内容が表示されます。

- プログラムのソースと POU を示す、実行されているコード位置(たとえば、ST ソースファイルの行)が表示されます。
- 実行されているコード位置を呼び出す、他の POU のコード位置が再帰的に表示されます。

下記も参照

[call stack program run]パラメータ (ページ 232)

6.2.4.2 [call stack program run]パラメータ

設定したすべてのタスクについて以下を表示することができます。

- 現在実行されているプログラムコード位置(たとえば、ST ソースファイルの行)
- このコード位置の呼び出しパス

表 6-3 [call stack program run]パラメータの説明

フィールド	説明
Selected CPU	選択した SIMOTION デバイスが表示されます。
Refresh	ボタンをクリックすると、SIMOTION デバイスから現在のコード位置が読み込まれ、開いているウィンドウに表示されます。
Calling task	実行されているコード位置を判別するタスクを選択します。 実行システムの設定したすべてのタスク。
Current code position	実行されているプログラムコード位置(たとえば、ST ソースファイルの行)が(プログラムソースファイルの名前、行番号、POU の名前と共に)表示されます。
is called by	選択したタスク内の実行されているコード位置を呼び出すコード位置が(適用可能な場合、プログラムソースファイルの名前、行番号、POU の名前、ファンクションブロックインスタンスの名前と共に)再帰的に表示されます。

6.2.4.3 プログラム実行ツールバー

このツールバーでは、MotionTask が現在実行しているコード位置(たとえば、ST ソースファイルの行)をその呼び出しパスと共に表示することができます。

図 6-2 プログラム実行ツールバー

6.2.5 プログラムステータス

6.2.5.1 プログラムステータスの特徴

ステータスプログラムを使用すると、プログラムの実行中、サイクルに対して正確に変数値を監視することができます。

ST ソースファイル内の監視領域を選択し、そこでグローバル変数とスタティックローカル変数に加え、さらにテンポラリローカル変数(たとえば、ファンクション内の変数)を監視することができます。

以下の変数の値が表示されます。

- 単純データタイプの変数(INT、REAL など)
- 割り付けが行われている場合、構造体の個々の要素
- 割り付けが行われている場合、配列の個々の要素
- 列挙データタイプの変数

SIMOTION Kernel V3.1 以降では、ファンクションまたはファンクションブロックインスタンスを呼び出す ST ソースファイル内の位置(呼び出しパス)を選択することができます。これにより、この呼び出しについて特別に変数値を確認することができます。

注記

バッファ容量の制限と実行時の改ざんを最小にする必要から、以下の変数を表示することはできません。

- 完全な配列
- 完全な構造体

ただし、ST ソースファイルで割り付けが行われている場合、個々の配列要素または個々の構造体要素は表示されます。

通知

プログラムステータスは、追加の CPU リソースを必要とします。

ステータスプログラムを使用して複数のプログラムを同時に監視したい場合は、以下の点に注意してください。

- テストモードを有効にする必要があります(SIMOTIONデバイスのモード (ページ 224)を参照)。
- SIMOTION Kernel V4.0 までのバージョンでは、各種のタスクにプログラムを割り付ける必要があります。

6.2.5.2 ステータスプログラムの使用

ステータスプログラムを操作するには、特殊なモードで動作するようにシステムに指示しておく必要があります。

1. コンパイル中に ST ソースファイルが追加のデバッグコードを生成することを確認します。
   - プロジェクトナビゲータで ST ソースファイルを選択し、**[Edit|Object properties]**メニューコマンドを選択します。
   - **[Compiler]**タブを選択し、**[Local settings active]**チェックボックスを有効にします。
   - **[Enable Status program]**チェックボックスが有効になっていることを確認し、**[OK]**で確定します。
2. ST ソースファイルを開き、**[ST source file|Save and compile]**を使用して再コンパイルします。
3. 通常の方法でプログラムをダウンロードし起動します。
4. **[Status program]**ボタン(以下の図を参照)をクリックして、デバッグモードを起動します。

ST エディタウィンドウが縦に分割されます。左側のペインには ST ソースファイル、右側のペインには選択した変数とその値が表示されます。

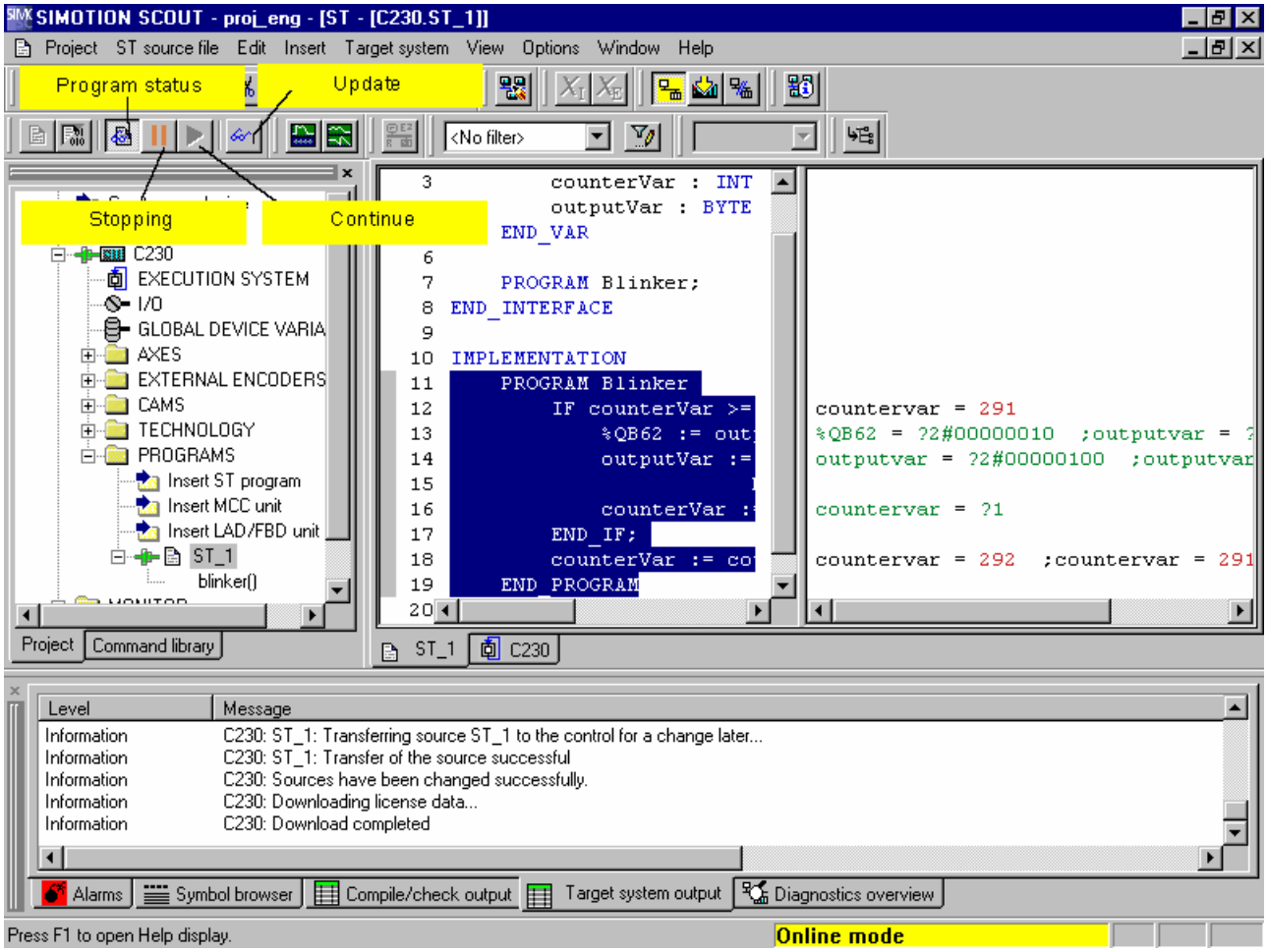


図 6-3　　プログラムステータステストモードの ST プログラムの一部

プログラムステータスをテストするには、以下の手順に従ってください。

1. エディタで、ST ソースファイルのテストしたいセクションを選択します。
2. SIMOTION Kernel バージョン V3.2 現在

   プログラムソースファイル内の複数の位置、または複数のタスクによって呼び出される POU のセクションを選択した場合

   プログラムステータスの呼び出しパスを入力します。

選択したセクションについて、画面の右側のペインに変数とその値が表示されます。
これらは定期的に更新されます。

- 現在のパスで変更された値は**赤**で表示されます。
- 変更されていない値は**黒**で表示されます。
- 値を持たない変数、たとえば未使用の IF 分岐内の変数は**緑**で表示されます。

変数値の表示が非常に素早く変わる場合

- 表示を停止するには、**[Stop monitoring program variables]**ボタンをクリックします(図を参照)。
- 表示を継続するには、**[Continue monitoring program variables]**ボタンをクリックします(図を参照)。

下記も参照

プログラムステータスの呼び出しパス (ページ 236)

6.2.5.3 プログラムステータスの呼び出しパス

SIMOTION Kernel V3.1 以降では、ファンクションおよびファンクションブロックの変数値を監視するときに、呼び出しパスを指定することができます。これにより、この呼び出しについて特別に変数値を確認することができます。

この目的のため、以下の場合、**[Call path]**ウィンドウが自動的に開きます。

- ファンクションのセクションを選択した場合

  SIMOTION デバイスのプログラムソースファイル(たとえば、ST ソースファイル)のさまざまなポイントでファンクションが呼び出されます。
- ファンクションブロックのセクションを選択した場合

  複数のファンクションブロックインスタンスがあるか、SIMOTION デバイスのプログラムソースファイル(たとえば、ST ソースファイル)のさまざまなポイントでインスタンスが呼び出されます。
- プログラムのセクションを選択した場合

  複数のタスクにプログラムが割り付けられます。

呼び出しパスを選択する方法

[Call path status program]ウィンドウに、POU のマーク付きセクション(コード位置)が(ST ソースファイルの名前、行番号、POU の名前と共に)表示されます。

1. 複数のタスクでコード位置を呼び出す場合
   - タスクを選択します。
2. 呼び出すコード位置を選択します(呼び出し側 POU 内)。

   以下から選択することができます。
   - 選択したタスク内の呼び出すコード位置(プログラムソースの名前、行番号、POU の名前を含む)。

     選択した呼び出すコード位置が今度は複数のコード位置によって呼び出される場合、同様に処理する追加の行が表示されます。
   - **すべて:**

     表示されたすべてのコード位置が選択されます。さらに、表示されたコード位置を呼び出すすべてのコード位置(階層の最上位まで)が選択されます。

下記も参照

[call path status program]パラメータ (ページ 237)

6.2.5.4 [call path status program]パラメータ

表 6-4 [Program status call path]パラメータの説明

フィールド	説明
Calling task	タスクを選択します。 選択したコード位置を呼び出すすべてのタスクを選択できます。
Current code position	POU の選択したセクション(コード位置)が(ST ソースファイルの名前、行番号、POU の名前と共に)表示されます。
is called by	呼び出すコード位置を選択します。 以下を使用することができます。 • 選択したタスク内の呼び出すコード位置(プログラムソースの名前、行番号、POU の名前を含む)。 選択した呼び出すコード位置が今度は複数のコード位置によって呼び出される場合、同様に処理する追加の行が表示されます。 • **すべて**: 表示されたすべてのコード位置が選択されます。さらに、表示されたコード位置を呼び出すすべてのコード位置(階層の最上位まで)が選択されます。

6.2.5.5 プログラムステータスの更新

表示された値を強制的に更新することができます。**[Update]**ボタンをクリックします。

6.2.6 ブレークポイント

6.2.6.1 ブレークポイント設定の一般的な手順

ブレークポイントはプログラムソース(ST ソースファイル、MCC ソースファイル、LAD/FBD ソース)内で設定できます。有効にされたブレークポイントに達すると、ブレークポイントが呼び出される POU にあるタスクは停止されます。ローカル変数の値は、詳細ビューの**[Status variables]**タブで確認することができます。ファンクションブロックの場合は、入/出力パラメータ(VAR_IN_OUT)をモニタすることはできません。グローバル変数は、引き続きシンボルブラウザでモニタすることができます。

前提条件:

- POU のプログラムソース(ST ソースファイル、MCC チャート、LAD/FBD プログラム)が開いている。

以下のように実行します。

以下の手順に従います。

1. [Debug mode]
   を関連付けられたSIMOTIONデバイスについて選択します(SIMOTIONデバイスのモード (ページ 224)を参照)。
2. デバッグタスクグループを指定します。
3. ブレークポイントを設定します。
4. 呼び出しパスを設定します。
5. ブレークポイントを有効にします。

6.2.6.2 デバッグタスクグループの定義

有効にされたブレークポイントに達すると、デバッグタスクグループに割り当てられたすべてのタスクが停止されます。

前提条件

- 関連する SIMOTION デバイスがデバッグモードになっている。

以下のように実行します。

デバッグタスクグループにタスクを割り当てるには

1. プロジェクトナビゲータで関連する SIMOTION デバイスを強調表示します。
2. コンテキストメニューから[Debug task group]を選択します。

   [Debug Settings]ウィンドウが開きます。
3. ブレークポイントに達したら停止させるタスクを選択します。
   - RUN 状態にある個々のタスクを停止するだけの場合: [Debug task group]選択オプションを有効にします。

     ブレークポイントに達すると停止させるすべてのタスクを[Tasks to be stopped]リストに割り当てます。
   - HALT 状態にある個々のタスクを停止するだけの場合: [All tasks]選択オプションを有効にします。

     この場合、プログラム実行の再開後に出力とテクノロジーオブジェクトを再び解放するかどうかも選択します。

有効にされたブレークポイントに達する異なる動作の詳細については、 以下の表を参照してください。

表 6-5　デバッグタスクグループ内のタスクに応じて有効にされたブレークポイントに達する動作

特性	停止させるタスク	
	選択したタスク	すべてのタスク
動作状態	RUN	STOP
停止されるタスク	デバッグタスクグループ内のタスクのみ	すべてのタスク
出力	有効	無効
タスクのランタイム測定	すべてのタスクについて有効	すべてのタスクについて無効
ウォッチドッグ	デバッグタスクグループ内のタスクについて無効	すべてのタスクについて無効
テクノロジ	閉ループ制御が有効	閉ループ制御が無効
リアルタイムクロック	動作を続行	動作を続行
プログラム実行再開時の動作	デバッグタスクグループのタスクは動作を続行	すべてのタスクが動作を続行 出力とテクノロジーオブジェクトの動作は、['Continue' activates the outputs]チェックボックスによって異なります。 • [Active]の場合: すべての出力とテクノロジーオブジェクトが解放されます。 • [Inactive]の場合: すべての出力とテクノロジーオブジェクトは、別のプロジェクトがダウンロードされると初めて解放されます。

注記

有効なブレークポイントがない場合、デバッグタスクグループに対する変更だけを行うことができます。

以下のように実行します。

1. ブレークポイントを設定します(「ブレークポイントの設定」を参照)。
2. 呼び出しパスを定義します(「単一ブレークポイントの呼び出しパスの定義」を参照)。
3. ブレークポイントを有効にします(「ブレークポイントの有効化」を参照)。

下記も参照

ブレークポイントの設定 (ページ 241)
単一ブレークポイントの呼び出しパスの定義 (ページ 243)
ブレークポイントの有効化 (ページ 248)
[Debug settings]パラメータ (ページ 240)

6.2.6.3 [Debug settings]パラメータ

デバッグタスクグループを定義するには、このウィンドウを使用します。有効にしたブレークポイントに達すると、デバッグタスクグループに割り付けたすべてのタスクが停止します。

このためには、関連する SIMOTION デバイスがデバッグモードになっていることが必要です。

表 6-6 [Debug settings]パラメータの説明

フィールド	説明
Debug task group	(RUN 状態の)個々のタスクを停止したいだけの場合、この選択オプションを選択します。 ブレークポイントに達したときに停止させるすべてのタスクを**[Tasks to be stopped]**リストに割り付けます。 有効にしたブレークポイントに達したときの動作は、次の表に詳しく示しています:「有効にしたブレークポイントに達したときの、デバッグタスクグループのタスクに応じた動作」
All tasks	(STOP 状態の)すべてのユーザタスクを停止したい場合、この選択オプションを選択します。 この場合、プログラムの実行の再開後に出力を再度解放するかどうかも選択します。 有効にしたブレークポイントに達したときの動作は、次の表に詳しく示しています:「有効にしたブレークポイントに達したときの、デバッグタスクグループのタスクに応じた動作」
*[Resume]*は出力を有効にします。	**[All tasks]**を選択した場合のみ。 このチェックボックスを選択すると、プログラムの実行を再開した後に出力とテクノロジーオブジェクトが再度解放されます。 すべての出力とテクノロジーオブジェクトを解放できるのは、チェックボックスを無効にしたプロジェクトをダウンロードした後だけです。

通知
デバッグタスクグループを変更することができるのは、どのブレークポイントも有効になっていない場合だけです。

6.2.6.4 [Debug table]パラメータ

デバッグテーブルには、SIMOTION デバイスのプログラムソース内のすべてのデバッグポイント(ブレークポイントなど)が表示されます。

表 6-7 [Debug table]パラメータの説明

フィールド		説明
デバッグポイント(テーブル)		
	Active	
	Source, line (POU)	設定されたデバッグポイントと共にコード位置(ST ソースファイルの名前、行番号、POU の名前を含む)が表示されます。
	Debug type	デバッグポイントのタイプが表示されます(たとえば、ブレークポイント、トレースポイントなど)。
	Call path	存在する場合、呼び出しパスが表示されます。
すべてのブレークポイント...		
	activate	SIMOTION デバイスのプログラムソース内のすべてのブレークポイントを有効にするには、このボタンをクリックします。
	Deactivating	SIMOTION デバイスのプログラムソース内のすべてのブレークポイントを無効にするには、このボタンをクリックします。
	deleting	SIMOTION デバイスのプログラムソース内のすべてのブレークポイントをクリアするには、このボタンをクリックします。

6.2.6.5 ブレークポイントの設定

前提条件:

1. POU のプログラムソース(ST ソースファイル、MCC チャート、LAD/FBD プログラム)が開いている。
2. 関連する SIMOTION デバイスが**デバッグモード**になっている(「SIMOTION デバイスの動作モード」を参照)。
3. デバッグタスクグループが定義されている(「デバッグタスクグループの定義」を参照)。

以下のように実行します。

ブレークポイントを設定するには

1. ブレークポイントが設定されていないコード位置を選択します。
   - SIMOTION ST の場合: ステートメントを含む ST ソースファイル内の行にカーソルを置きます。
   - SIMOTION MCC の場合: MCC チャートで MCC コマンドを選択します(モジュールやコメントブロックは除く)。
   - SIMOTION LAD/FBD の場合: LAD/FBD プログラムのネットワーク内にカーソルを設定します。
2. [Breakpoints]ツールバーで**[Set/remove breakpoints]**ボタンをクリックします(または Ctrl+H ショートカットを使用します)。

ブレークポイントを削除するには、[Set/remove breakpoint]をもう一度クリックします。

注記

以下の場所にはブレークポイントを設定できません。

- SIMOTION ST の場合: コメントしかない行
- SIMOTION MCC の場合: モジュール上またはコメントブロックコマンド
- SIMOTION LAD/FBD の場合: ネットワーク内
- 他のデバッグポイント(トリガポイントなど)が設定されたコード位置

以下の手順を実行して、すべてのデバッグポイントをリストできます。プロジェクトナビゲータで該当する SIMOTION デバイスを選択し、コンテキストメニューから[Debug table]を選択します。

すべてのブレークポイントを削除するには、[Breakpoints]ツールバーまたはデバッグテーブルで対応するボタンをクリックします。

プログラムステータス診断ファンクションとブレークポイントを、プログラムのソースファイルや POU で組み合わせて使用することができます。ただし、プログラム言語によって以下の制限事項があります。

- SIMOTION ST の場合: SIMOTION Kernel のバージョン V3.2 では、プログラムステータスをテストするマークされた ST ソースファイル行に、ブレークポイントを設定することはできません。
- SIMOTION MCC と LAD/FBD の場合: プログラムステータスをテストする MCC チャート(または LAD/FBD プログラムのネットワーク)のコマンドに、ブレークポイントを設定することはできません。

以下のように実行します。

1. 呼び出しパスを定義します(「単一ブレークポイントの呼び出しパスの定義」を参照)。
2. ブレークポイントを有効にします(「ブレークポイントの有効化」を参照)。

下記も参照

SIMOTIONデバイスのモード (ページ 224)
デバッグタスクグループの定義 (ページ 238)
単一ブレークポイントの呼び出しパスの定義 (ページ 243)
ブレークポイントの有効化 (ページ 248)

6.2.6.6 [Breakpoints]ツールバー

このツールバーには、ブレークポイントを設定し有効にするための重要なオペレータ操作が含まれています。

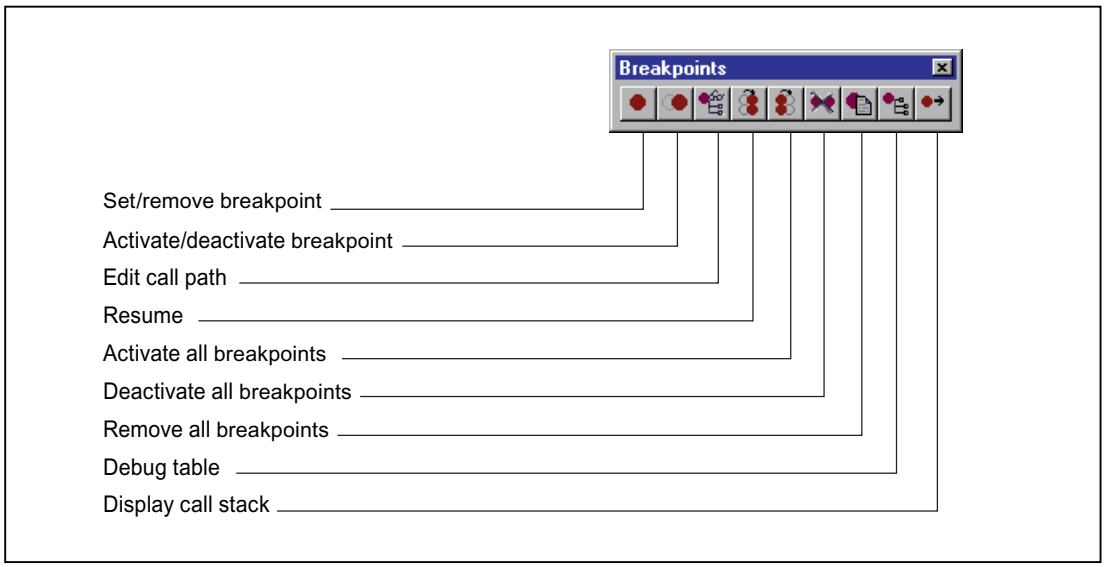


図 6-4 [Breakpoints]ツールバー

6.2.6.7 単一ブレークポイントの呼び出しパスの定義

前提条件:

1. POU のプログラムソース(ST ソースファイル、MCC チャート、LAD/FBD プログラム)が開いている。
2. 関連する SIMOTION デバイスがデバッグモードになっている (「SIMOTION デバイスの動作モード」を参照)。
3. デバッグタスクグループが定義されている(「デバッグタスクグループの定義」を参照)。
4. ブレークポイントが設定されている(「ブレークポイントの設定」を参照)。

以下のように実行します。

単一ブレークポイントの呼び出しパスを定義するには、以下のように実行します。

1. ブレークポイントが既に設定されているコード位置を選択します。
   - SIMOTION ST の場合: カーソルを ST ソースの適切な行に設定します。
   - SIMOTION MCC の場合: MCC チャートで適切なコマンドを選択します。
   - SIMOTION LAD/FBD の場合: LAD/FBD プログラムの適切なネットワークにカーソルを設定します。
2. ブレークポイントを設定する ST ソースファイルの行にカーソルを置きます。
3. [Breakpoints]ツールバーの**[Edit call path]**ボタンをクリックします。

   [Call path breakpoint]ウィンドウに、マークされたコード位置が表示されます(ST ソースファイルの名前、行番号、POU の名前とともに)。
4. コード位置が複数タスクで呼び出される場合:
   - 該当するタスクを選択します。

     このタスクはデバッグタスクグループに存在している必要があります。
5. 呼び出し元の POU で、呼び出すコード位置を選択します。

   以下の中から選択することができます。
   - 選択したタスク内で呼び出されるコード位置(プログラムソース、行番号、POU の名前とともに)。

     選択した呼び出しコード位置がさらに複数のコード位置により呼び出される場合は、以降の行が表示され同様に操作します。
   - **All:**

     表示されたすべてのコード位置が選択されます。さらに、すべてのコード位置(階層のトップレベルまで)が選択され、そこから表示されたコード位置が呼び出されます。
6. コード位置に複数回達した後に初めてブレークポイントが有効になる場合は、その回数を選択します。

以下のように実行します。

- ブレークポイントを有効にします(「ブレークポイントの有効化」を参照)。

注記

選択するタスクに応じて、[Call stack]機能で呼び出しパスを表示することができます。**「呼び出しスタックの表示」**を参照してください。

下記も参照

SIMOTIONデバイスのモード (ページ 224)
デバッグタスクグループの定義 (ページ 238)
ブレークポイントの設定 (ページ 241)
[Breakpoint call path]パラメータ (ページ 245)
[Call Path of all Breakpoints per POU]パラメータ (ページ 247)
呼び出しスタックの表示 (ページ 249)

6.2.6.8 [Breakpoint call path]パラメータ

表 6-8 [Breakpoint call path]パラメータの説明

フィールド	説明
Selected CPU	選択した SIMOTION デバイスが表示されます。
Calling task	タスクを選択します。 以下を使用することができます。 • 設定されたブレークポイントがあるコード位置を呼び出すタスク。 選択したタスクはデバッグタスクグループ内にある必要があります。 • [All] 設定されたブレークポイントがあるコード位置を呼び出す、デバッグタスクグループのすべてのタスクが選択されます。
Current code position	設定されたブレークポイントと共にコード位置(ST ソースファイルの名前、行番号、POU の名前を含む)が表示されます。
is called by	呼び出すコード位置を選択します。 以下を使用することができます。 • 選択したタスク内の呼び出すコード位置(適用可能な場合、プログラムソースファイルの名前、行番号、POU の名前、ファンクションブロックインスタンスの名前を含む) 選択した呼び出すコード位置が今度は複数のコード位置によって呼び出される場合、同様に処理する追加の行が表示されます。 • すべて: 表示されたすべてのコード位置が選択されます。さらに、表示されたコード位置を呼び出すすべてのコード位置(階層の最上位まで)が選択されます。
Activate breakpoint for ... execution	特定の回数だけコード位置に達するまでブレークポイントを有効にしたくない場合、この数字を設定します。

通知

デバッグタスクグループを変更することができるのは、どのブレークポイントも有効になっていない場合だけです。

下記も参照

ブレークポイントの設定 (ページ 241)
単一ブレークポイントの呼び出しパスの定義 (ページ 243)

6.2.6.9 すべてのブレークポイントに呼び出しパスを定義

この手順では、以下のことができます。

- POU (MCC チャート、LAD/FBD プログラム、ST ソースファイル内の POU など)にあるすべてのブレークポイントの場合: 呼び出しパスを一致させます(以降も同様)。
- このプログラムソース(ST ソースファイル、MCC ソースファイル、LAD/FBD ソースファイル)のすべての以降のブレークポイントの場合: デフォルト設定を選択します。

前提条件

- POU のプログラムソース(ST ソースファイル、MCC チャート、LAD/FBD プログラム)が開いている。
- 関連する SIMOTION が、SIMOTION デバイスのデバッグモード動作モードになっている。
- デバッグタスクグループが定義されている(「デバッグタスクグループの定義」を参照)。

以下のように実行します。

POU のすべてのブレークポイントの呼び出しパスを定義するには、以下のように実行します。

1. ブレークポイントが設定されて**いない**コード位置を選択します。
   - SIMOTION ST の場合: カーソルを ST ソースの適切な行に設定します。
   - SIMOTION MCC の場合: MCC チャートで適切なコマンドを選択します。
   - SIMOTION LAD/FBD の場合: LAD/FBD プログラムの適切なネットワークにカーソルを設定します。
2. **[Breakpoints]**ツールバーの**[Edit call path]**ボタンをクリックします。

   [Call path/All breakpoints per POU task selection]ウィンドウに、選択した MCC コマンド(コード位置)が表示されます(MCC ソースファイル、行番号、MCC チャートの名前とともに)。
3. 該当する MCC コマンド(コード位置)が複数タスクで呼び出される場合:

   呼び出し側のタスクを選択します。以下を使用することができます。
   - 設定されたブレークポイントのあるコード位置のタスクが呼び出されます。
     選択したタスクはデバッグタスクグループに存在している必要があります。
   - **All**
     ブレークポイントが設定されたコード位置にある、デバッグタスクグループのすべてのタスクが選択されます。このタスクはデバッグタスクグループに存在している必要があります。
4. 呼び出し側のコード位置を選択します。

   以下を使用することができます。
   - 選択されたタスク内にある呼び出し側のコード位置(プログラムソースファイルの名前、行番号、POU の名前、該当する場合はファンクションブロックインスタンスの名前とともに)
     選択した呼び出しコード位置がさらに複数のコード位置により呼び出される場合は、以降の行が表示され同様に操作します。
   - **All:**
     表示されたすべてのコード位置が選択されます。さらに、すべてのコード位置(階層のトップレベルまで)が選択され、そこから表示されたコード位置が呼び出されます。
5. コード位置に複数回達した後に初めてブレークポイントが有効になる場合は、その回数を選択します。
6. **[OK]**を選択して確定します。

以下のように実行します。

- ブレークポイントを有効にします(「ブレークポイントの有効化」を参照)。

注記

選択するタスクに応じて、[Call stack]機能で呼び出しパスを表示することができます。
呼び出しスタックの表示.

下記も参照

呼び出しスタックの表示 (ページ 249)

デバッグタスクグループの定義 (ページ 238)

SIMOTIONデバイスのモード (ページ 224)

単一ブレークポイントの呼び出しパスの定義 (ページ 243)

6.2.6.10 [Call Path of all Breakpoints per POU]パラメータ

ここでは、POU 内にあるすべてのブレークポイントの呼び出しパスに対して事前設定を定義することができます。さらに、この POU の以前のすべてのブレークポイントについてこの設定を受け入れることもできます。

表 6-9 [Call Path of all Breakpoints per POU]パラメータの説明

フィールド	説明
Selected CPU	選択した SIMOTION デバイスが表示されます。
Calling task	タスクを選択します。 以下を使用することができます。 • 現在の POU を呼び出すタスク。 選択したタスクはデバッグタスクグループ内にある必要があります。 • **[All]** 現在の POU を呼び出す、デバッグタスクグループのすべてのタスクが選択されます。
Current POU	カーソルが置かれている POU が(ST ソースファイルの名前、POU の名前と共に)表示されます。
is called by	呼び出すコード位置を選択します。 以下を使用することができます。 • 選択したタスク内の呼び出すコード位置(適用可能な場合、プログラムソースファイルの名前、行番号、POU の名前、ファンクションブロックインスタンスの名前を含む) 選択した呼び出すコード位置が今度は複数のコード位置によって呼び出される場合、同様に処理する追加の行が表示されます。 • **すべて:** 表示されたすべてのコード位置が選択されます。さらに、表示されたコード位置を呼び出すすべてのコード位置(階層の最上位まで)が選択されます。
Activate breakpoint for ... execution	特定の回数だけコード位置に達するまでブレークポイントを有効にしたくない場合、この数字を設定します。
Apply this call path to all previous breakpoints of this POU	現在の POU の以前のすべてのブレークポイントに呼び出しパスを適用したい場合、**[Apply]**ボタンをクリックします。既存の設定が上書きされます。

下記も参照

ブレークポイントの設定 (ページ 241)

単一ブレークポイントの呼び出しパスの定義 (ページ 243)

6.2.6.11 ブレークポイントの有効化

ブレークポイントは、プログラム実行に影響を与える場合に有効にする必要があります。

前提条件

1. POU のプログラムソース(ST ソースファイル、MCC チャート、LAD/FBD プログラム)が開いている。
2. 関連する SIMOTION デバイスがデバッグモードになっている (「SIMOTION デバイスの動作モード」を参照)。
3. デバッグタスクグループが定義されている(「デバッグタスクグループの定義」を参照)。
4. ブレークポイントが設定されている(「ブレークポイントの設定」を参照)。
5. 呼び出しパスが定義されている(「単一ブレークポイントの呼び出しパスの定義」を参照)。

以下のように実行します。

単一ブレークポイントを有効にするには

1. ブレークポイントが既に設定されているコード位置を選択します。
   - SIMOTION ST の場合: カーソルを ST ソースの適切な行に設定します。
   - SIMOTION MCC の場合: MCC チャートで適切なコマンドを選択します。
   - SIMOTION LAD/FBD の場合: LAD/FBD プログラムの適切なネットワークにカーソルを設定します。
2. **[Breakpoints]**ツールバーの**[Activate/deactivate breakpoint]**ボタンをクリックします。

   ブレークポイントを無効にするには、**[Activate/deactivate breakpoint]**ボタンをもう一度クリックします。

すべてのブレークポイントを有効にするには

- **[Breakpoints]**ツールバーの**[Activate all breakpoints]**ボタンをクリックします。

  ブレークポイントを無効にするには、**[Deactivate all breakpoints]**ボタンをクリックします。

有効にされたブレークポイントに達すると、デバッグタスクグループに割り当てられたタスクが停止されます。この動作は、[Define debug task group]に記述されたデバッグタスクグループ内のタスクに応じて異なります。

注記

ブレークポイントは、デバッグテーブルで有効または無効にすることもできます。

1. プロジェクトナビゲータで該当する SIMOTION デバイスを選択し、コンテキストメニューから**[Debug table]**を選択します。
2. 有効または無効にするブレークポイントに応じて、以下の操作を実行するウィンドウが開きます。
   - 単一ブレークポイントの場合: 対応するチェックボックスを選択またはクリアします。
   - すべてのブレークポイントの場合: 対応するボタンをクリックします。

プログラム実行を再開するには

- **[Breakpoints]**ツールバーの**[Resume]**ボタンをクリックします(ショートカットは Ctrl+F8)。

下記も参照

SIMOTIONデバイスのモード (ページ 224)

デバッグタスクグループの定義 (ページ 238)

[Breakpoint call path]パラメータ (ページ 245)

ブレークポイントの設定 (ページ 241)

単一ブレークポイントの呼び出しパスの定義 (ページ 243)

6.2.6.12 呼び出しスタックの表示

[Call stack]機能を使用して、選択したタスクに応じた呼び出しパスを表示させることができます。

以下のように実行します。

呼び出しスタックを表示するには

1. ブレークポイントが既に設定されているコード位置を選択します。
   - SIMOTION ST の場合: カーソルを ST ソースの適切な行に設定します。
   - SIMOTION MCC の場合: MCC チャートで適切なコマンドを選択します。
   - SIMOTION LAD/FBD の場合: LAD/FBD プログラムの適切なネットワークにカーソルを設定します。
2. [Breakpoints]ツールバーの**[Display callstack]**ボタンをクリックします。

   [Callstack breakpoint]ダイアログが開きます。

   プログラムがブレークポイントで停止されると、現在の呼び出しスタックが、呼び出し側のタスクと指定された反復回数とともに表示されます。呼び出しスタック自身(呼び出しパス)を変更することはできません。

   プログラムがブレークポイントで停止されない場合、ダイアログは表示されません。
3. 別のタスクの呼び出しスタックを表示するには、呼び出し側のタスクを変更します。

   前のデータは上書きされます。
4. [Breakpoints]ツールバーの**[Continue]**ボタンを使用して、次のブレークポイントにジャンプします。

   新しい呼び出しスタックが表示されます。前のデータは上書きされます。
5. **[OK]**を選択して確定します。

6.2.6.13 [Breakpoints call stack]パラメータ

有効にしたブレークポイントに達したら、デバッグタスクグループのタスクごとに以下を表示することができます。

- タスクが停止したプログラムコード位置(たとえば、ST ソースファイルの行)
- このコード位置の呼び出しパス

表 6-10 [Breakpoint call path]パラメータの説明

フィールド	説明
Selected CPU	選択した SIMOTION デバイスが表示されます。
Calling task	タスクが停止されたコード位置を表示するタスクを選択します。 デバッグタスクグループのすべてのタスクを選択することができます。
Current code position	選択したタスクが停止されたプログラムコード位置(たとえば、ST ソースファイルの行)が(プログラムソースファイルの名前、行番号、POU の名前と共に)表示されます。
is called by	選択したタスク内の現在のコード位置を呼び出すコード位置が(適用可能な場合、プログラムソースファイルの名前、行番号、POU の名前、ファンクションブロックインスタンスの名前と共に)再帰的に表示されます。

下記も参照

ブレークポイントの有効化 (ページ 248)

6.2.7 トレース

トレースツールを使用すると、時間と共に変化する変数値(たとえば、ユニット変数、ローカル変数、システム変数、I/O 変数)を記録および格納することができます。これにより、たとえば軸などの最適化を文書化することができます。

記録時間の設定、最大 4 つのチャンネルの表示、トリガ条件の選択、タイミング調整のパラメータ設定、カーブのさまざまな表示とスケーリングの選択などを行うことができます。

等時間間隔の記録の他に、**コード位置での記録**を選択することもできます。これにより、プログラムが ST ソースファイルの特定のポイントを通過するときにいつでも、変数の値を記録できるようになります。

トレースツールについてはオンラインヘルプで詳しく説明しています。

付録 A

A.1 形式言語記述

この章では、ST の基本要素の概要と、言語要素を含むすべての構文ダイアグラムの完全なコンパイルを示します。この付録には、ST 言語の基本機能が要約されています。

A.1.1 言語記述リソース

個々のセクションでは、言語記述の基礎として構文ダイアグラムを使用します。構文ダイアグラムを使用すると、ST の構文(すなわち文法)構造に関する理解が深まります。

構文ダイアログの使用に関する説明は、*「言語記述リソース」*に示しました。以下では、上級ユーザにとって興味のある、書式付きルールと書式なしルールの違いに関する情報を示します。

A.1.1.1 書式付きルール(語彙ルール)

語彙ルールには、語彙分析中にコンパイラによって処理される要素の構造が記述されています。つまり、表記が書式付きで、ルールに従う必要があります。特に、以下のことを意味します。)

- フォーマット文字は挿入できません。
- ブロックコメントおよび行コメントは挿入できません。
- 識別子の属性は挿入できません。

以下の図は、正当な識別子の語彙ルールを示します。

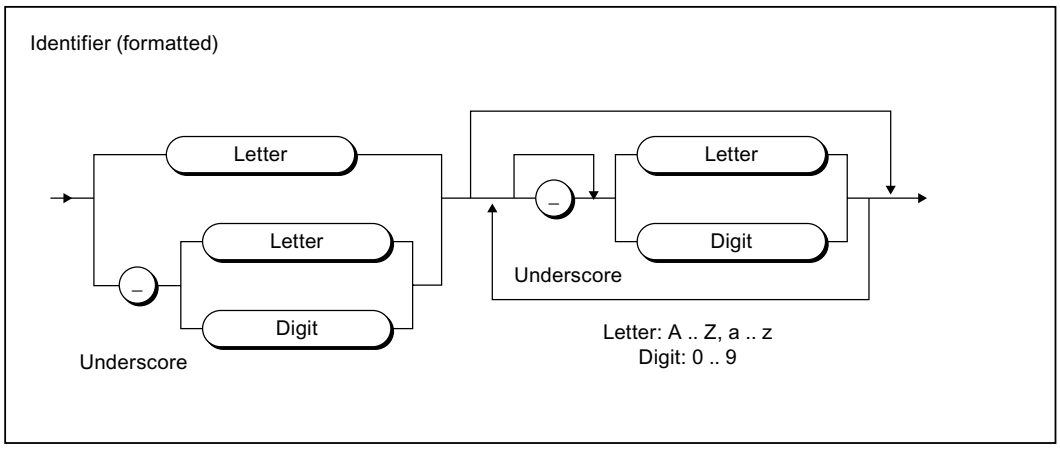


図 A-1 語彙ルールの例

このルールに従った有効な例

```
R_CONTROLLER3
_A_ARRAY
_100_3_3_10
```

A.1.1.2 書式なしルール(構文ルール)

構文ルールは語彙ルールの上に構築され、ST の構造が記述されています。このルールの枠組み内で、書式なしで ST プログラムを作成することができます。

書式なしということは以下を意味します。

- フォーマット文字をどこでも挿入できます。
- ブロックコメントおよび行コメントを挿入できます。

以下の例は、ステートメントで値を割り付けるための構文ルールを示します。

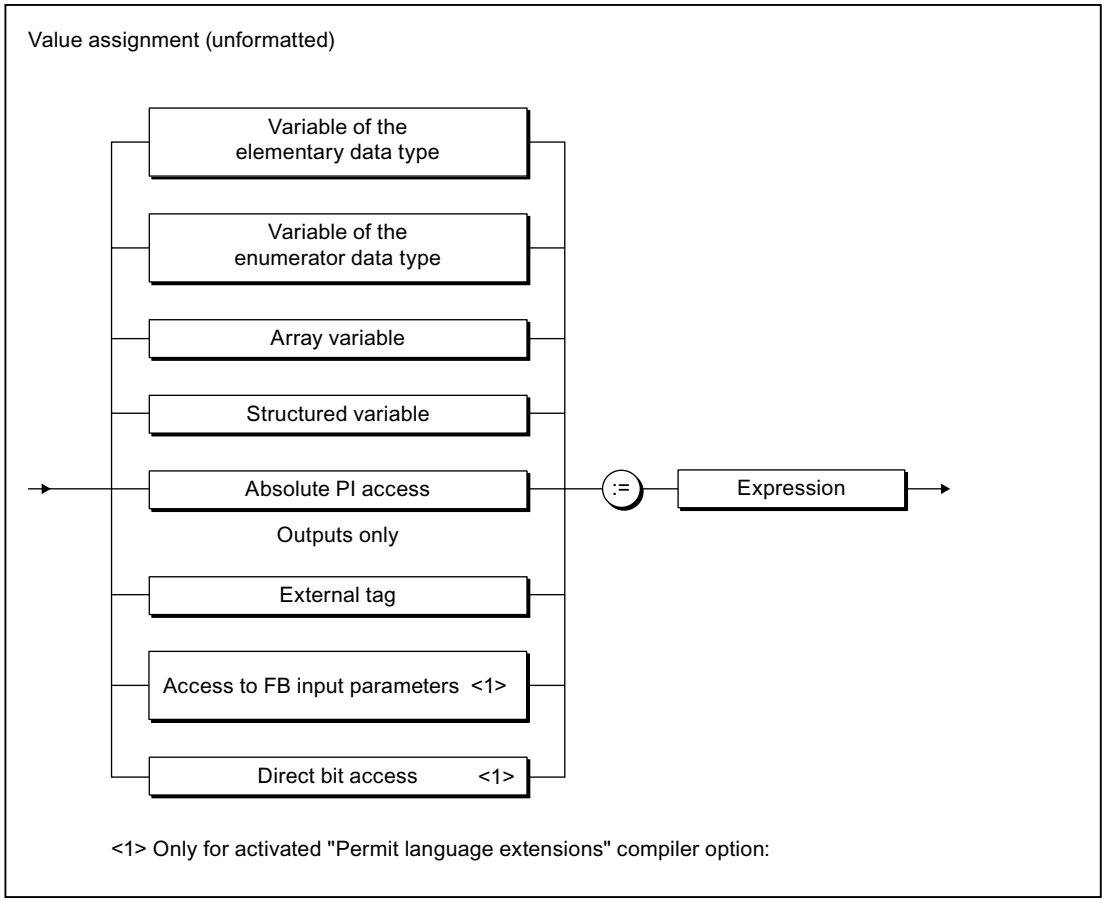


図 A-2 構文ルールの例

このルールに従った有効な例:

```
VARIABLE_1 := 100; SWITCH := FALSE;
//'これはコメントである
VARIABLE_2:=3.2 +VARIABLE_1;
```

A.1.2 基本要素(端子)

端子は、追加のルールではなく、言語で宣言する基本要素です。構文ダイアログでは、端子は楕円形または円によって表されます。

A.1.2.1 アルファベット、数字、およびその他の文字

アルファベットと数字は最も一般的に使われる文字です。たとえば、*識別子*は、アルファベット、数字、下線の組み合わせで構成されます。下線は特殊文字の 1 つです。

表 A-1 アルファベットおよび数字

文字	サブグループ	文字セットの要素
アルファベット	大文字	A..Z
	小文字	a.. z
数字	10 進数字	0.. 9
8 進数字	8 進数字	0.. 7
16 進数字	16 進数字	0.. 9, A.. F, a.. f
Bit	2 進数字	0, 1

コメントには完全な拡張 ASCII 文字セットを使用できます。10 進等価値 32 (空白)から始まるすべての印刷可能な ASCII コード文字を使用することができます。

言語コマンド、識別子、定数、式、および演算子の場合、特定のルールに従ってのみ、特殊文字、すなわちアルファベットと数字以外の文字を使用することができます。

A.1.2.2 ルールにおけるフォーマット文字およびセパレータ

フォーマット文字とセパレータは、書式付き(語彙)ルールと書式なし(構文)ルールとで使い方が異なります。言語記述リソース (ページ 251)に、構文ルールと語彙ルールの違いを説明しています。

以下の表に、語彙ルールと構文ルールのフォーマット文字とセパレータを示します。フォーマット文字とセパレータを端子として使用する際のすべてのルールの説明とリストも示しています(ルール (ページ 265)を参照)。

表 A-2 語彙ルールにおけるフォーマット文字およびセパレータ

文字	説明	語彙ルール
:	時間、分、秒の間の区切り	時刻情報
.	浮動小数点表示、時間間隔表示、絶対アドレス指定の区切り	浮動小数点表示、時刻情報、10 進表示、ローカルまたはグローバルインスタンスへのアクセス
_ 下線	識別子の区切り、定数の数値の区切り	識別子、10 進数字列、2 進数字列、8 進数字列、16 進数字列、シーケンス表示
%	CPU メモリアクセス時の直接識別子の接頭語	単純メモリアクセス
//	コメント	行コメント
(**)	コメント	ブロックコメント

表 A-3　構文ルールにおけるフォーマット文字およびセパレータ

文字	説明	構文ルール
:	タイプ情報の区切り	ファンクション、変数宣言、コンポーネント宣言、CASE ステートメント、インスタンス宣言
;	宣言またはステートメントの終了	定数ブロック、ステートメント、変数宣言、インスタンス宣言、コンポーネント宣言、ステートメントセクション
,	リストの区切り	変数宣言、配列初期化リスト、インスタンス宣言、ARRAY(配列)データタイプ指定、FB パラメータ、FC パラメータ、値リスト
..	範囲情報	配列データタイプ指定、値リスト
.	構造体アクセス	構造体変数
( )	配列の初期化リスト、式の丸括弧、ファンクション呼び出し、ファンクションブロック呼び出し	配列初期化リスト、式、単純乗算、オペランド、指数、FB 呼び出し、ファンクション呼び出し
[ ]	配列宣言、配列の構造体変数セクション	配列データタイプ指定

下記も参照

言語記述リソース (ページ 55)

A.1.2.3 定数のフォーマット文字およびセパレータ

以下に、定数のすべてのフォーマット文字およびセパレータと、それらを使用する語彙ルールに関する情報を示します。

表 A-4 定数のフォーマット文字およびセパレータ

文字	コードの用途	語彙ルール
2#	整数定数	2 進数字列
8#	整数定数	8 進数字列
16#	整数定数	16 進数字列
E	浮動小数点定数の区切り	指数
E	浮動小数点定数の区切り	指数
D#	時刻情報	日付
DATE#	時刻情報	日付
DATE_AND_TIME#	時刻情報	日付と時刻
DT#	時刻情報	日付と時刻
T#	時刻情報	持続時間
TIME#	時刻情報	持続時間
TIME_OF_DAY#	時刻情報	時刻
TOD#	時刻情報	時刻
d	時間間隔(日)の区切り	日(ルール: シーケンス表示)
h	時間間隔(時間)の区切り	時間 (ルール: シーケンス表示)
m	時間間隔(分)の区切り	分 (ルール: シーケンス表示)
ms	時間間隔(ミリ秒)の区切り	ミリ秒 (ルール: シーケンス表示)
s	時間間隔(秒)の区切り	秒 (ルール: シーケンス表示)

A.1.2.4 プロセスイメージアクセスのために事前定義された識別子

以下は、CPU メモリ領域(絶対識別子)にアクセスするのに使用できる、ST のすべての事前定義された変数のリストです。出力は読み書きできますが、入力は読み取りだけが可能です。

表 A-5 絶対識別子

識別子	説明	語彙ルール
%In.x または %IXn.x	バイトアドレスおよびビットアドレスを持つ CPU 入力範囲	絶対 PI アクセス
%IBn	バイトアドレスを持つ CPU 入力範囲	絶対 PI アクセス
%IWn	ワードアドレスを持つ CPU 入力範囲	絶対 PI アクセス
%IDn	ダブルワードアドレスを持つ CPU 入力範囲	絶対 PI アクセス
%Qn.x または %QXn.x	バイトアドレスおよびビットアドレスを持つ CPU 出力範囲	絶対 PI アクセス
%QBn	バイトアドレスを持つ CPU 出力範囲	絶対 PI アクセス
%QWn	ワードアドレスを持つ CPU 出力範囲	絶対 PI アクセス
%QDn	ダブルワードアドレスを持つ CPU 出力範囲	絶対 PI アクセス

A.1.2.5 Taskstartinfo の識別子

Taskstartinfo には以下の識別子が定義されています。

表 A-6 Taskstartinfo の識別子

識別子	データタイプ	説明
TSI#alarmNumber	DINT	アラーム番号のスキャン
TSI#commandId.high	UDINT	commandId のスキャン(最上位ワード)
TSI#commandId.low	UDINT	commandId のスキャン(最下位ワード)
TSI#currentTaskId	StructTaskId	現在のタスクの TaskId のスキャン
TSI#cycleTime	TIME	現在のタスクの設定済みサイクルタイムのスキャン
TSI#details	DWORD	詳細情報のスキャン
TSI#executionFaultType	UDINT	実行エラーのタイプのスキャン
TSI#interruptId	UDINT	トリガイベントのスキャン
TSI#logBaseAdrIn	DINT	論理ベースアドレスのスキャン
TSI#logBaseAdrOut	DINT	論理ベースアドレスのスキャン
TSI#logDiagAddr	DINT	論理診断アドレスのスキャン
TSI#shutDownInitiator	UDINT	STOP への移行の原因のスキャン
TSI#startTime	DT	開始時間のスキャン
TSI#taskId	StructTaskId	トリガタスクの TaskId のスキャン
TSI#toInst	ANYOBJECT	TO インスタンスのスキャン

A.1.2.6 演算子

以下は、ST のすべての演算子と、演算子を使用する際の構文ルールのリストです。

表 A-7 ST の演算子

演算子	説明	ルール
:=	割り付け演算子 (初期化値にも使用)	値割り付け、入力割り付け、入力/出力割り付け、変数宣言、定数宣言、ユーザ定義データタイプ、コンポーネント宣言
+, –	算術演算子: 単項演算子、符号	式、指数
+, –, *, / MOD	基本算術演算子	式、基本算術演算子
**	算術演算子: 指数演算子	式
NOT	論理演算子: 否定	式、オペランド
AND, &, OR, XOR	基本論理演算子	基本論理演算子
<, >, <=, >=, =, <>	関係演算子	関係演算子
=>	割り付け演算子	出力割り付け

A.1.2.7 予約語

以下は、基本 ST システムのキーワード、定義済み識別子、および標準ファンクションのアルファベット順のリストです。それらの説明と、それらを端子として使用する*ルール*の中から構文ルールも示しています。例外は、標準ファンクションです。これは、*ファンクション呼び出し*の構文ルールに標準ファンクション名として暗黙的にのみ組み込まれています。

注記

変数にキーワードまたは定義済み識別子の名前を割り付けてはいけません。識別子の詳細については、*「ST の識別子」*を参照してください。テクノロジーオブジェクトに予約されている識別子、およびその他の予約識別子の概要は、*「予約識別子」*を参照してください。

表 A-8 基本 ST システムの ST キーワードおよび定義済み識別子

キーワード/識別子	説明	ルール
ABS	標準数値ファンクション	ファンクション呼び出し
ACOS	標準数値ファンクション	ファンクション呼び出し
AND	論理演算子	基本論理演算子
ANYOBJECT	テクノロジーオブジェクトの一般データタイプ	TO データタイプ
ANYOBJECT_TO_OBJECT	タイプ変換のための標準ファンクション(テクノロジーオブジェクト)	ファンクション呼び出し
ANYTYPE_TO_BIGBYTEARRAY	標準ファンクション(マーシャリング)	ファンクション呼び出し
ANYTYPE_TO_LITTLEBYTEARRAY	標準ファンクション(マーシャリング)	ファンクション呼び出し
ARRAY	配列の指定の取り込みと、それに続く[と]で囲まれたインデックスリスト	配列データタイプ指定

キーワード/識別子	説明	ルール
AS	名前空間の取り込み	–
ASIN	標準数値ファンクション	ファンクション呼び出し
AT	予約識別子	–
ATAN	標準数値ファンクション	ファンクション呼び出し
BIGBYTEARRAY_TOANYTYPE	標準ファンクション(マーシャリング)	ファンクション呼び出し
BOOL	バイナリデータの基本データタイプ	ビットデータタイプ
BOOL_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_VALUE_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_VALUE_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_VALUE_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_VALUE_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_VALUE_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_VALUE_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_VALUE_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BOOL_VALUE_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BY	増分の取り込み	FOR ステートメント
BYTE	基本データタイプ	ビットデータタイプ
BYTE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_VALUE_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
BYTE_VALUE_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
CASE	選択のための制御ステートメントの取り込み	CASE ステートメント
CONCAT	文字列の編集のための標準ファンクション	ファンクション呼び出し
CONCAT_DATE_TOD	タイプ変換のための標準ファンクション	ファンクション呼び出し
CONSTANT	定数定義の取り込み	定数ブロック
COS	標準数値ファンクション	ファンクション呼び出し
CTD	ダウンカウンタ	ファンクションブロック呼び出し
CTD_DINT	ダウンカウンタ	ファンクションブロック呼び出し
CTD_UDINT	ダウンカウンタ	ファンクションブロック呼び出し
CTU	アップカウンタ	ファンクションブロック呼び出し
CTU_DINT	アップカウンタ	ファンクションブロック呼び出し
CTU_UDINT	アップカウンタ	ファンクションブロック呼び出し
CTUD	アップ/ダウンカウンタ	ファンクションブロック呼び出し

キーワード/識別子	説明	ルール
CTUD_DINT	アップ/ダウンカウンタ	ファンクションブロック呼び出し
CTUD_UDINT	アップ/ダウンカウンタ	ファンクションブロック呼び出し
DATE	データの基本データタイプ	時刻タイプ
DATE_AND_TIME	日付と時刻の基本データタイプ	DATE_AND_TIME
DATE_AND_TIME_TO_DATE	タイプ変換のための標準ファンクション	ファンクション呼び出し
DATE_AND_TIME_TO_TIME_OF_DAY	タイプ変換のための標準ファンクション	ファンクション呼び出し
DELETE	文字列の編集のための標準ファンクション	ファンクション呼び出し
DINT	-2**31 ~ 2**32-1 の値範囲の倍精度整数の基本データタイプ	数値データタイプ
DINT_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
DINT_VALUE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
DO	FOR ステートメントのステートメントセクションの取り込み	FOR ステートメント、 WHILE ステートメント
DT	DATE_AND_TIME の省略表記	DATE_AND_TIME
DT_TO_DATE	タイプ変換のための標準ファンクション	ファンクション呼び出し
DT_TO_TOD	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD	ダブルワードの基本データタイプ	ビットデータタイプ
DWORD_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
DWORD_VALUE_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
ELSE	条件が真でない場合に実行する句の取り込み	IF ステートメント、 CASE ステートメント
ELSIF	代替条件の取り込み	IF ステートメント
END_CASE	CASE ステートメントの終了	CASE ステートメント
END_EXPRESSION	EXPRESSION ステートメントの終了	ファンクション
END_FOR	FOR ステートメントの終了	FOR ステートメント

キーワード/識別子	説明	ルール
END_FUNCTION	ファンクションの終了	ファンクション
END_FUNCTION_BLOCK	ファンクションブロックの終了	ファンクションブロック (Function block)
END_IF	IF ステートメントの終了	IF ステートメント
END_IMPLEMENTATION	実装セクションの終了	実装セクション
END_INTERFACE	インターフェースセクションの終了	インターフェースセクション
END_LABEL	LABEL ステートメントの終了	–
END_PROGRAM	プログラムセクションの終了	プログラムセクション
END_REPEAT	REPEAT ステートメントの終了	REPEAT ステートメント
END_STRUCT	構造体の指定の終了	STRUCT データタイプ指定
END_TYPE	UDT の終了	ユーザ定義データタイプ
END_VAR	宣言ブロックの終了	テンポラリ変数ブロック、スタティック変数ブロック、パラメータブロック、定数ブロック
END_WAITFORCONDITION	プログラム可能なイベントを待機するタスクの制御ステートメントの終了	WAITFORCONDITION ステートメント
END_WHILE	WHILE ステートメントの終了	WHILE ステートメント
ENUM_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
EXIT	ループの実行からの直接の終了	EXIT
EXP	標準数値ファンクション	ファンクション呼び出し
EXPD	標準数値ファンクション	ファンクション呼び出し
EXPRESSION	待機タスクに対するプログラム可能イベント	ファンクション
EXPT	標準数値ファンクション	ファンクション呼び出し
FIND	文字列の編集のための標準ファンクション	ファンクション呼び出し
F_TRIG	立ち下がりの検出	ファンクションブロック呼び出し
FALSE	事前定義されたブール定数: 論理条件が偽、値が 0 と等しい	–
FOR	ループ実行のための制御ステートメントの取り込み	FOR ステートメント
FUNCTION	ファンクションの取り込み	ファンクション
FUNCTION_BLOCK	ファンクションブロックの取り込み	ファンクションブロック(Function block)
GOTO	ジャンプ	–
IF	選択のための制御ステートメントの取り込み	IF ステートメント
IMPLEMENTATION	IMPLEMENTATION セクションの取り込み	IMPLEMENTATION セクション
INSERT	文字列の編集のための標準ファンクション	ファンクション呼び出し
INT	-2**15～2**15-1 の値範囲の単精度整数の基本データタイプ	数値データタイプ
INT_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_TIME	タイプ変換のための標準ファンクション	ファンクション呼び出し

キーワード/識別子	説明	ルール
INT_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
INT_VALUE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
INTERFACE	インターフェースセクションの取り込み	インターフェースセクション
LABEL	ジャンプラベルの定義	–
LEFT	文字列の編集のための標準ファンクション	ファンクション呼び出し
LEN	文字列の編集のための標準ファンクション	ファンクション呼び出し
LIMIT	選択のための標準ファンクション	ファンクション呼び出し
LITTLEBYTEARRAY_TOANYTYPE	標準ファンクション(マーシャリング)	ファンクション呼び出し
LN	標準数値ファンクション	ファンクション呼び出し
LOG	標準数値ファンクション	ファンクション呼び出し
LREAL	64 ビット倍精度浮動小数点数の基本データタイプ(long real)	数値データタイプ
LREAL_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_VALUE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_VALUE_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_VALUE_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
LREAL_VALUE_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
MAX	選択のための標準ファンクション	ファンクション呼び出し
MID	文字列の編集のための標準ファンクション	ファンクション呼び出し
MIN	選択のための標準ファンクション	ファンクション呼び出し
MOD	除算の剰余の算術演算子	基本算術演算子、単純な乗算
MUX	選択のための標準ファンクション	ファンクション呼び出し
NOT	論理演算子(単項演算子に帰属)	式、オペランド
OF	データタイプ指定の取り込み	配列データタイプ指定、CASE ステートメント
OR	論理演算子	基本論理演算子
PROGRAM	PROGRAM セクションの取り込み	プログラム
R_TRIG	立ち上がりの検出	ファンクションブロック呼び出し
REAL	32 ビット単精度浮動小数点数の基本データタイプ(real)	数値データタイプ
REAL_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し

キーワード/識別子	説明	ルール
REAL_TO_TIME	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_VALUE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_VALUE_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_VALUE_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
REAL_VALUE_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
REPEAT	ループ実行のための制御ステートメントの取り込み	REPEAT ステートメント
REPLACE	文字列の編集のための標準ファンクション	ファンクション呼び出し
RETAIN	バッファ型変数の宣言	保持型変数ブロック
RIGHT	文字列の編集のための標準ファンクション	ファンクション呼び出し
RETURN	サブルーチンから戻るための制御ステートメント	RETURN ステートメント
ROL	ビット文字列の標準ファンクション	ファンクション呼び出し
ROR	ビット文字列の標準ファンクション	ファンクション呼び出し
RS	双安定ファンクションブロック (優先度のリセット)	ファンクションブロック呼び出し
RTC	リアルタイムクロック	ファンクションブロック呼び出し
SEL	選択のための標準ファンクション	ファンクション呼び出し
SHL	ビット文字列の標準ファンクション	ファンクション呼び出し
SHR	ビット文字列の標準ファンクション	ファンクション呼び出し
SIN	標準数値ファンクション	ファンクション呼び出し
SINT	-128 ~ 127 の値範囲の短整数の基本データタイプ	数値データタイプ
SINT_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
SINT_VALUE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
SQRT	標準数値ファンクション	ファンクション呼び出し
SR	双安定ファンクションブロック(優先度の設定)	ファンクションブロック呼び出し
STRING		
STRUCT	構造体の指定の取り込みと、それに続くコンポーネントリスト	STRUCT データタイプ指定
StructAlarmId	AlarmId のデータタイプ	–
StructAlarmId_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し

キーワード/識別子	説明	ルール
StructTaskId	TaskId のデータタイプ	–
TAN	標準数値ファンクション	ファンクション呼び出し
THEN	条件が真の場合のその続のアクションの取り込み	IF ステートメント
TIME	時刻情報の基本データタイプ	時刻タイプ
TIME_OF_DAY	時刻の基本データタイプ	時刻タイプ
TIME_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
TIME_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
TO	終了値の取り込み	FOR ステートメント
TOD	TIME_OF_DAY の省略表記	時刻タイプ
TOF	OFF 遅延	ファンクションブロック呼び出し
TON	ON 遅延	ファンクションブロック呼び出し
TP	パルス	ファンクションブロック呼び出し
TRUE	事前定義されたブール定数: 論理条件が真、値が 0 と等しくない	–
TYPE	UDT の取り込み	ユーザ定義データタイプ
UDINT	0～2**32 -1 の値範囲の符号なし倍精度整数の基本データタイプ	数値データタイプ
UDINT_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
UDINT_VALUE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT	0～2**16-1 の値範囲の符号なし単精度整数の基本データタイプ	数値データタイプ
UINT_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
UINT_VALUE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
UNIT	UNIT セクションの取り込み	ユニットセクション

キーワード/識別子	説明	ルール
UNTIL	REPEAT ステートメントの終了条件の取り込み	REPEAT ステートメント
USELIB	ライブラリ名の取り込み	–
USEPACKAGE	パッケージ名の取り込み	–
USES	他のユニットへの参照の取り込み	–
USINT	0～256 の値範囲の符号なし短整数の基本データタイプ	数値データタイプ
USINT_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_TO_WORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
USINT_VALUE_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
VAR	ローカル変数の宣言ブロックの取り込み	スタティック変数ブロック
VAR_GLOBAL	ユニット変数(グローバル変数)の宣言ブロックの取り込み	ユニット変数
VAR_IN_OUT	宣言ブロックの取り込み	パラメータブロック
VAR_INPUT	宣言ブロックの取り込み	パラメータブロック
VAR_OUTPUT	宣言ブロックの取り込み	パラメータブロック
VAR_TEMP	宣言ブロックの取り込み	パラメータブロック
VOID	ファンクション呼び出しで戻り値なし	ファンクション
WAITFORCONDITION	プログラム可能なイベントを待機するタスクの制御ステートメントの取り込み	WAITFORCONDITION ステートメント
WHILE	ループ実行のための制御ステートメントの取り込み	WHILE ステートメント
WITH	WAITFORCONDITION と組み合わせて使用	WAITFORCONDITION ステートメント
WORD	ワードの基本データタイプ	ビットデータタイプ
WORD_TO_BOOL	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_TO_BYTE	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_TO_DINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_TO_DWORD	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_TO_INT	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_TO_SINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_TO_UDINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_TO_UINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_TO_USINT	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_VALUE_TO_LREAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
WORD_VALUE_TO_REAL	タイプ変換のための標準ファンクション	ファンクション呼び出し
XOR	論理演算子	基本論理演算子

A.1.3 ルール

ST 言語の以下の構文ルールは、書式付き表記を使用するルール(語彙ルール)と書式なし表記を使用するルール(構文ルール)に細分されます。構文ルールと語彙ルールの違いは、*「言語記述リソース」*で説明しています。

A.1.3.1 識別子

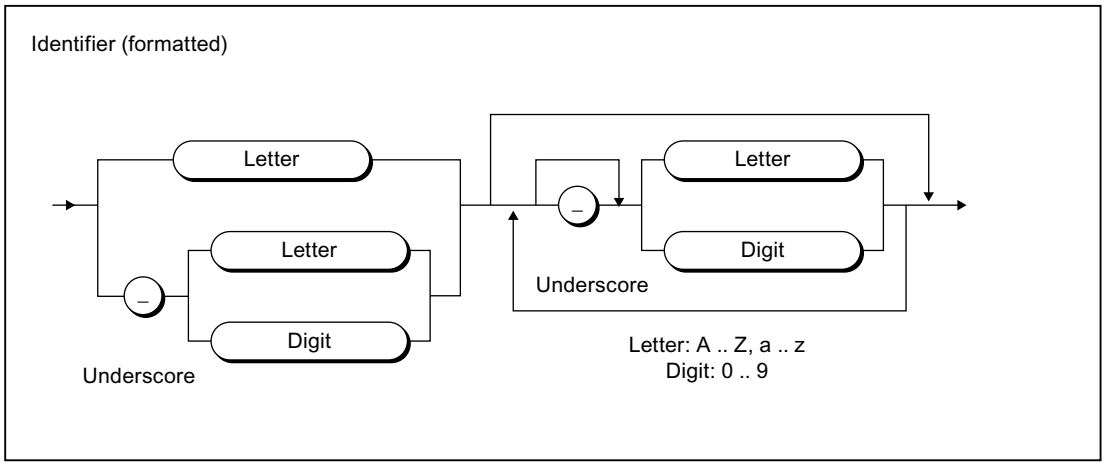


図 A-3 識別子

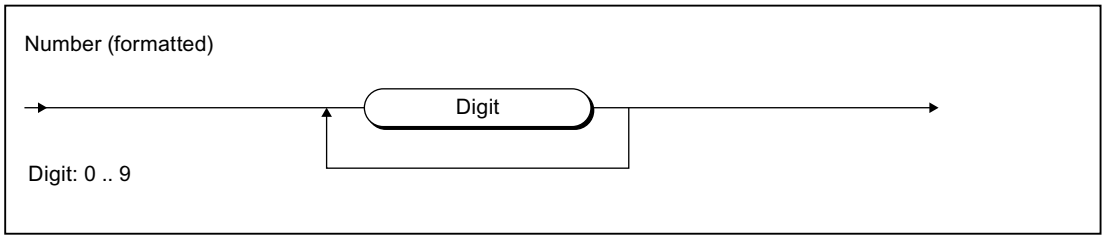


図 A-4 st_num1_001

A.1.3.2 定数(リテラル)の表記

リテラル

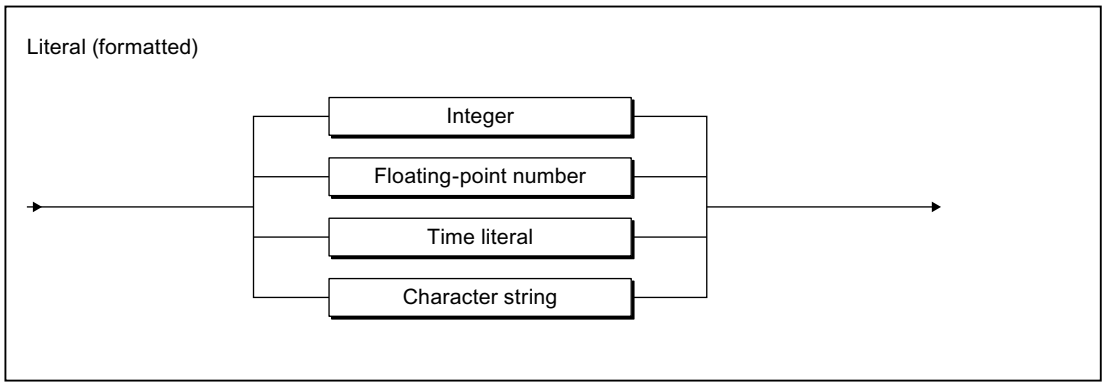


図 A-5 st_lit1_001

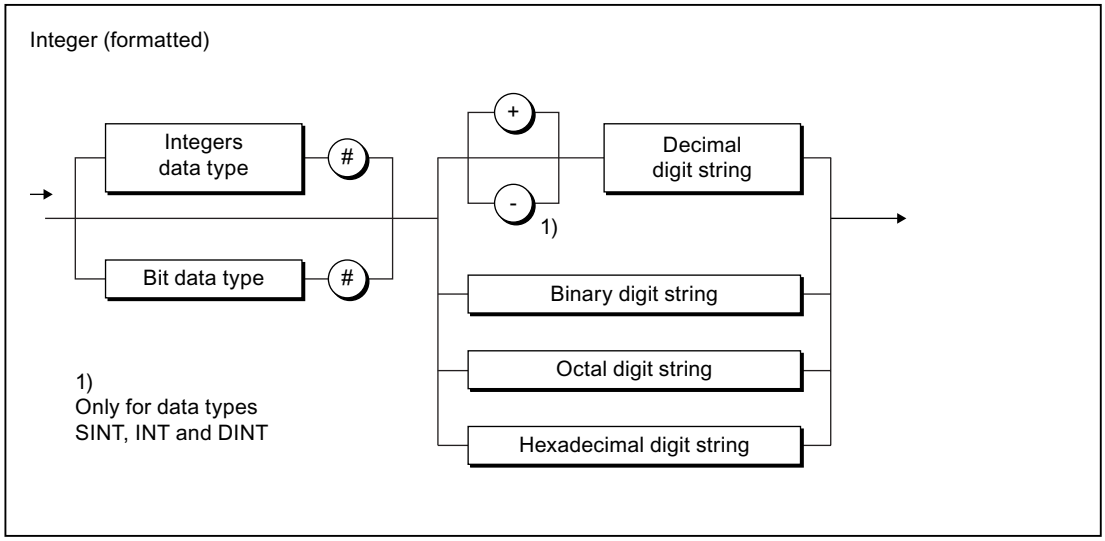


図 A-6 st_int1_001

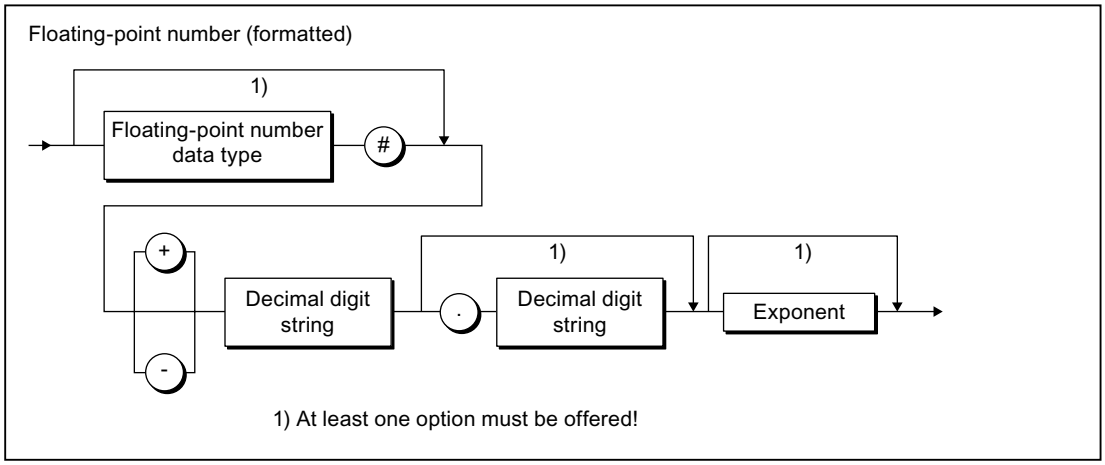


図 A-7 st_rea1_001

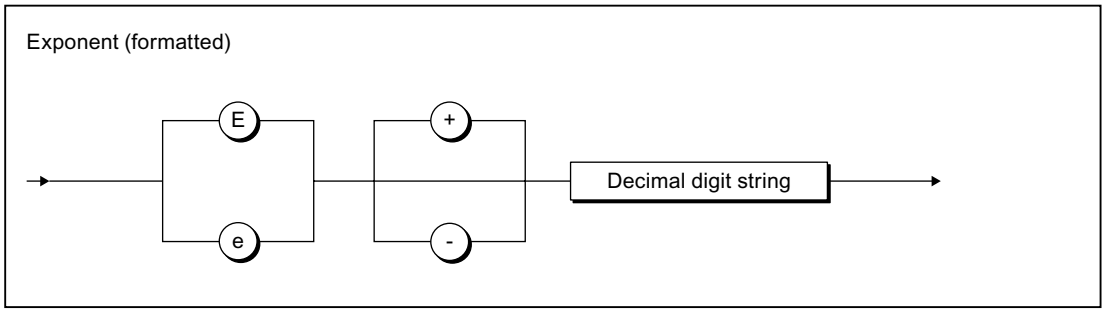


図 A-8 st_exp1_001

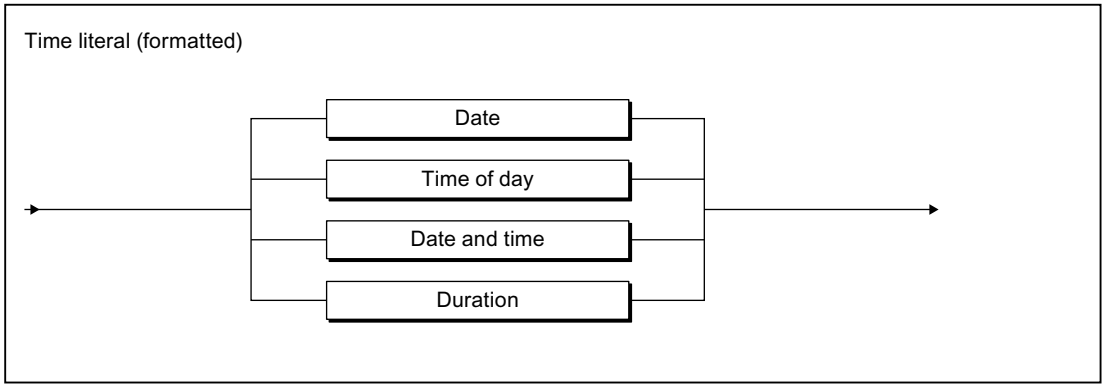


図 A-9 st_lit2_001

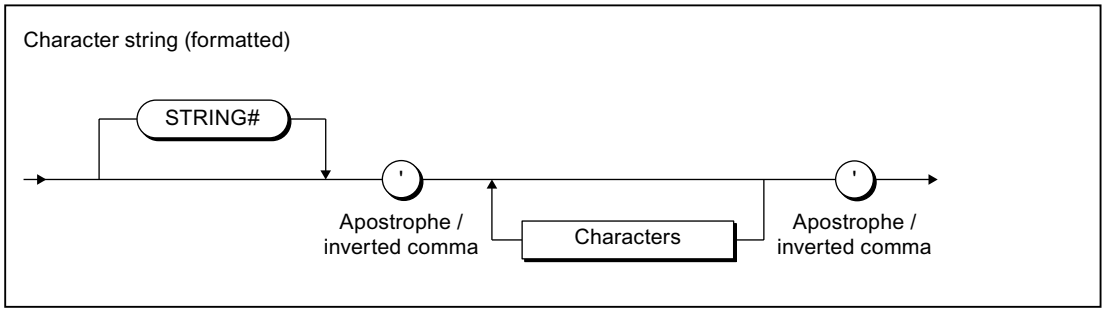


図 A-10 st_str5_001

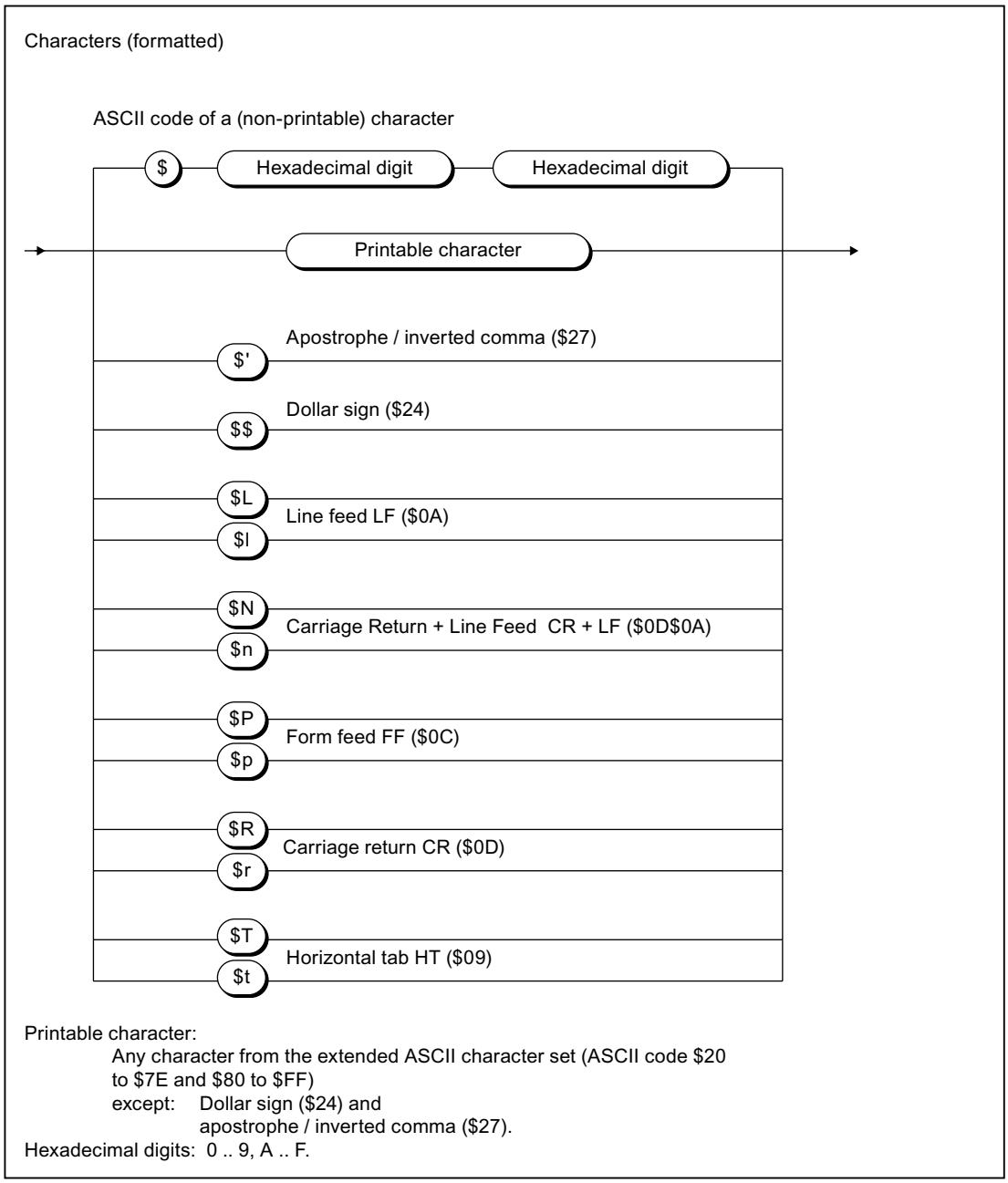


図 A-11 st_str4_001

数字列

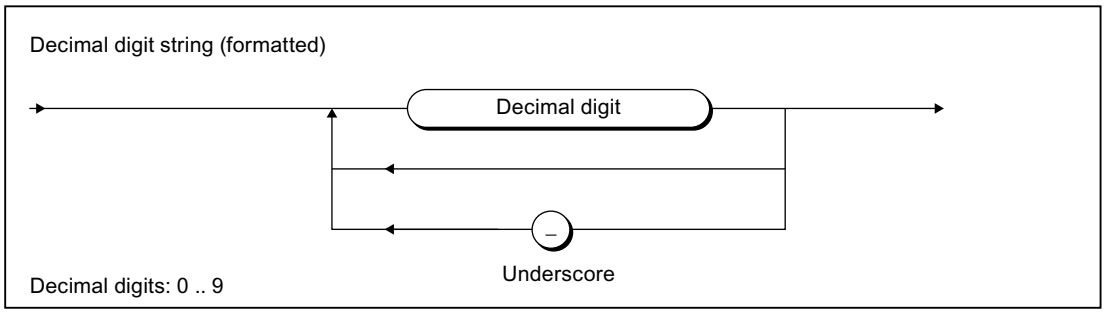


図 A-12 st_dez1_001

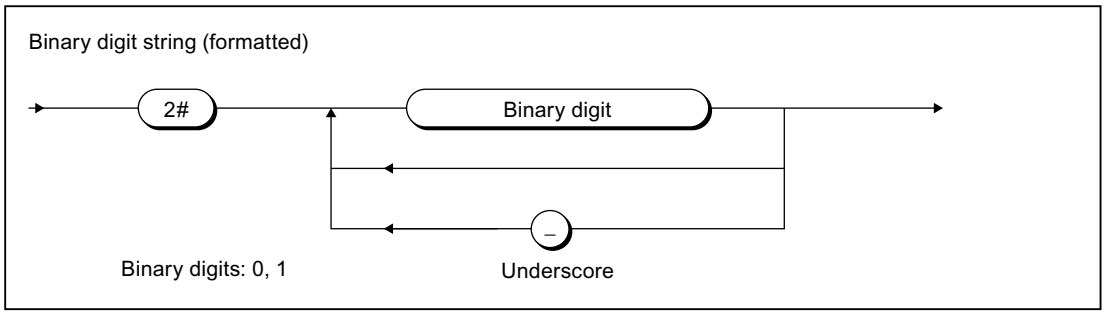


図 A-13 st_bin1_001

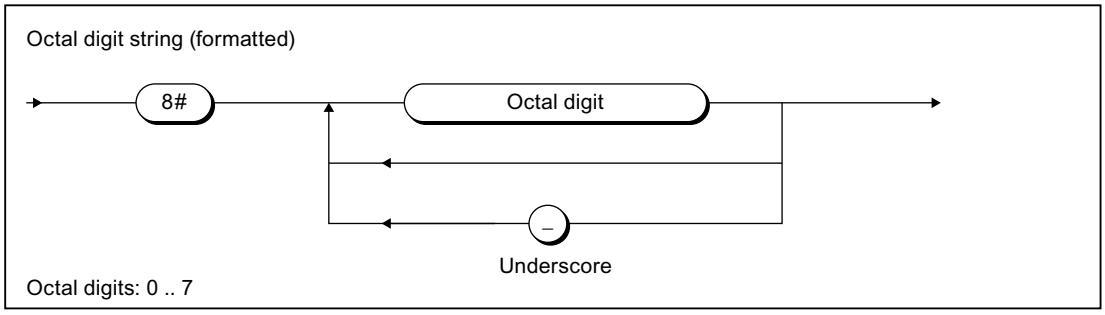


図 A-14 st_okt1_001

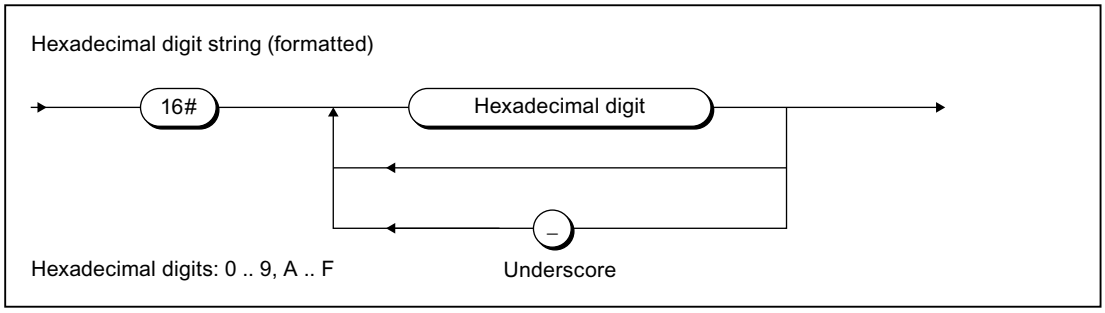


図 A-15 st_hex1_001

日付と時刻

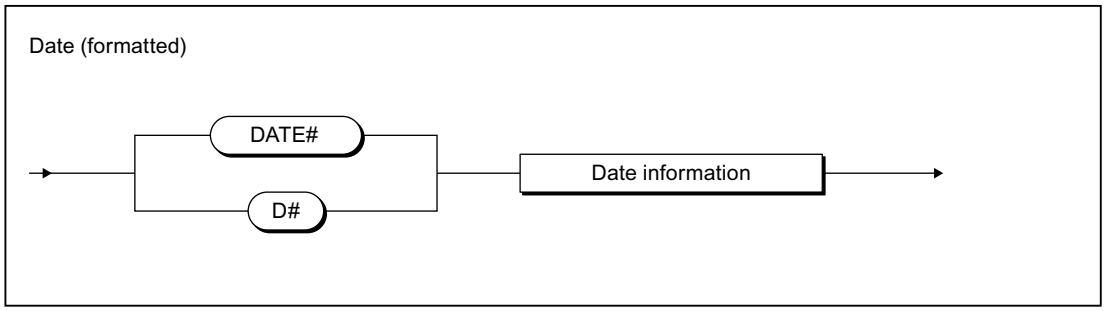


図 A-16　st_dat1_001

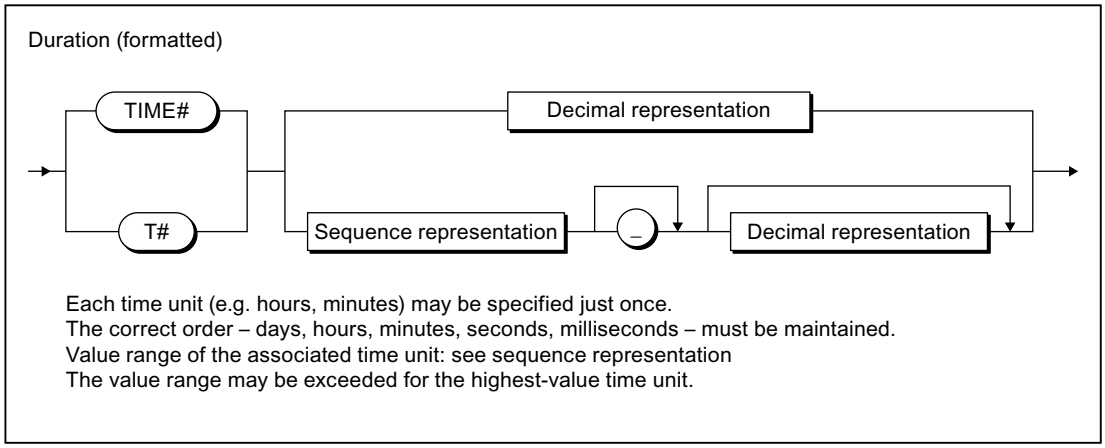


図 A-17　st_zei1_001

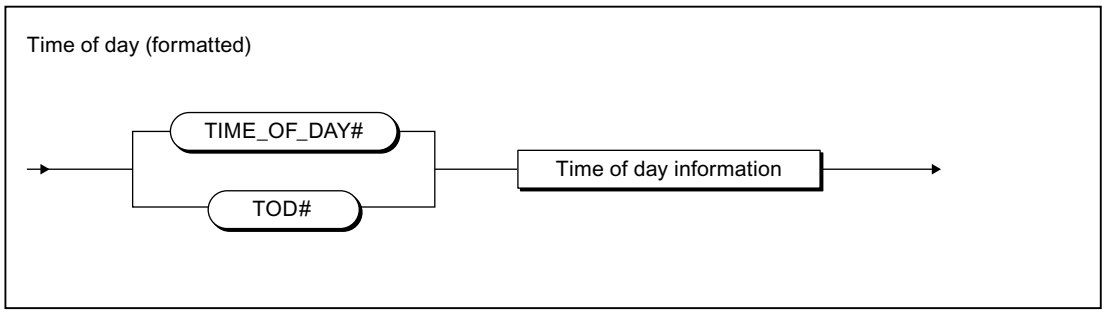


図 A-18　st_tag1_001

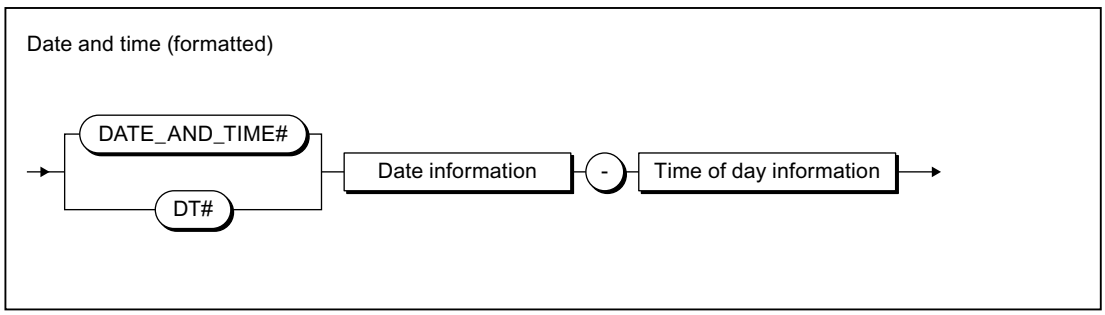


図 A-19　st_dat2_001

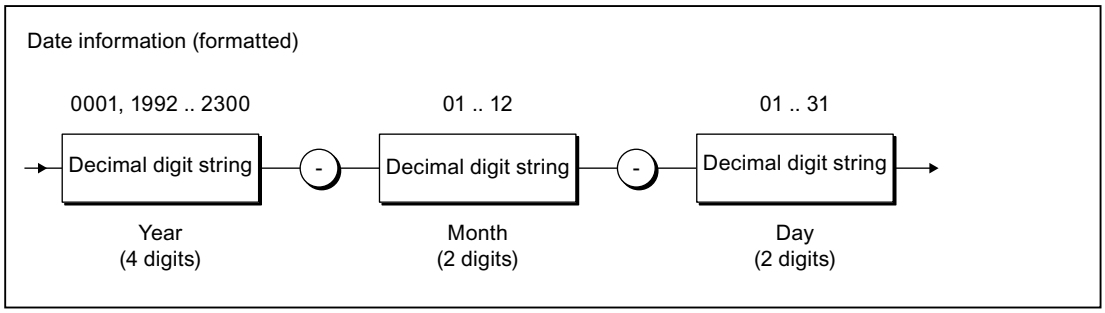


図 A-20 st_dat3_045

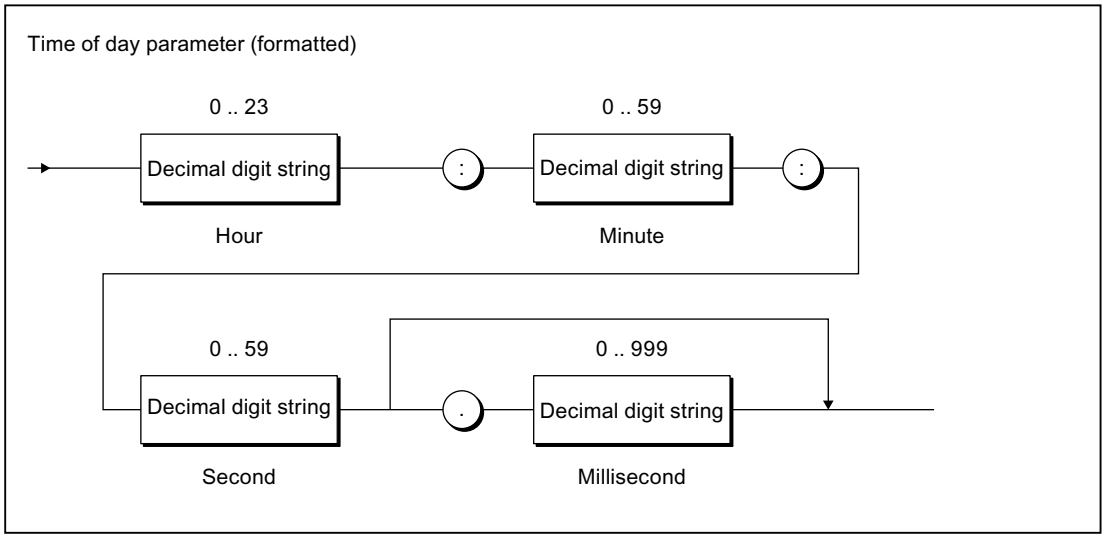


図 A-21 st_tag2_001

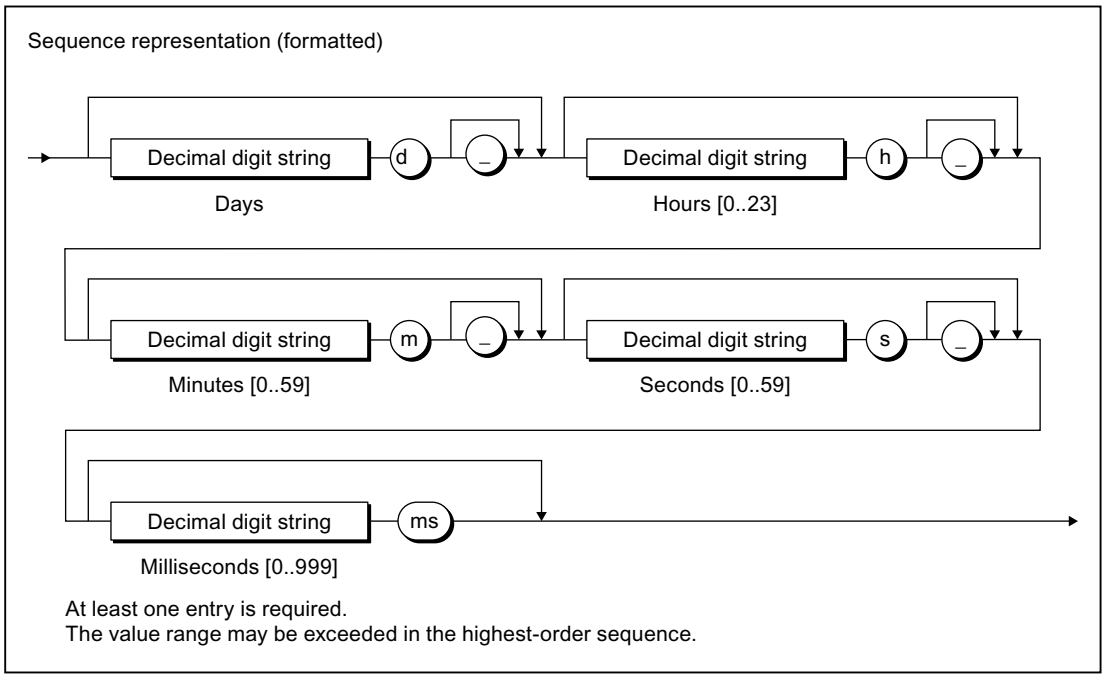


図 A-22 st_stu1_001

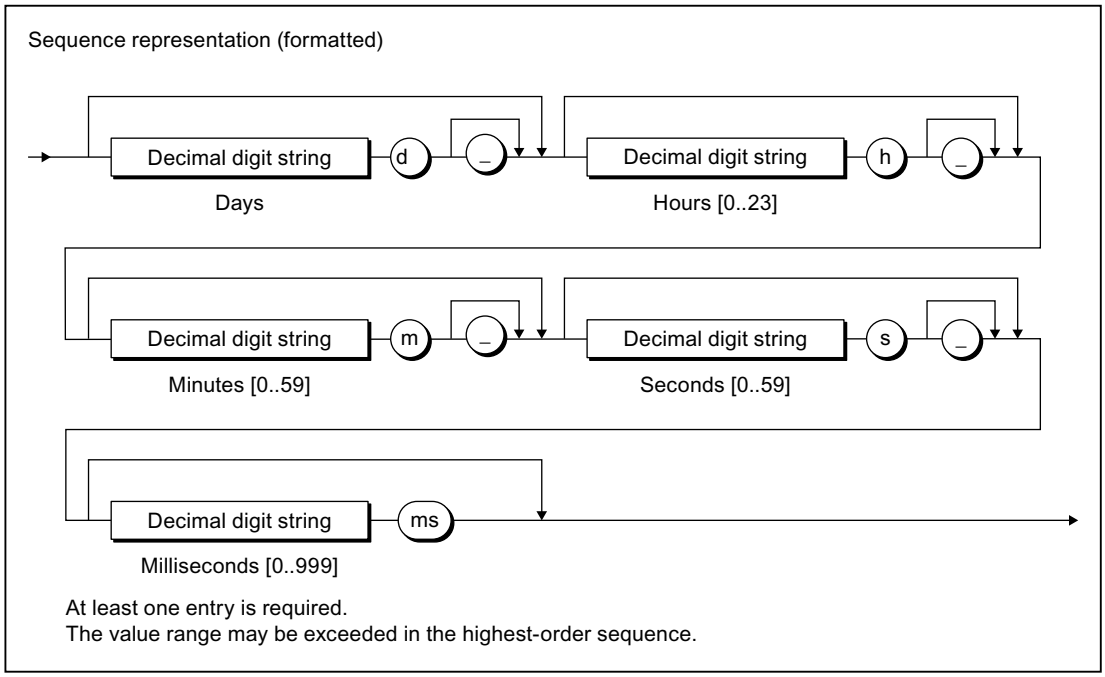


図 A-23 st_dez2_001

A.1.3.3 説明

コメントを挿入するときは、以下の点に注意してください。

- 行コメントのネストはできません。
- ブロックコメントのネストはできません。ただし、ブロックコメントに行コメントをネストすることはできます。
- 書式なし(構文)ルールでは、コメントはどこでも挿入できます。
- 書式付き(語彙)ルールでは、コメントは挿入できません。

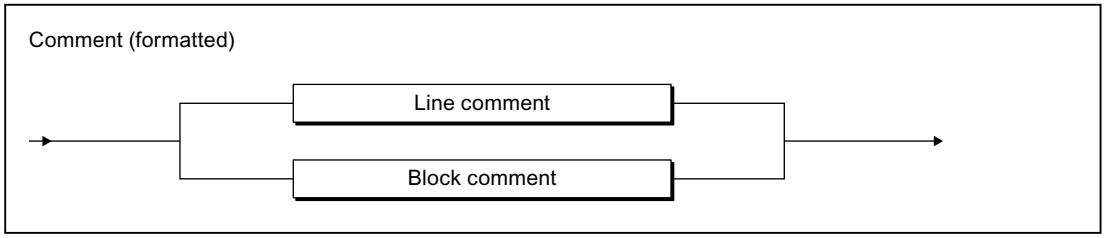


図 A-24 st_kom1_001

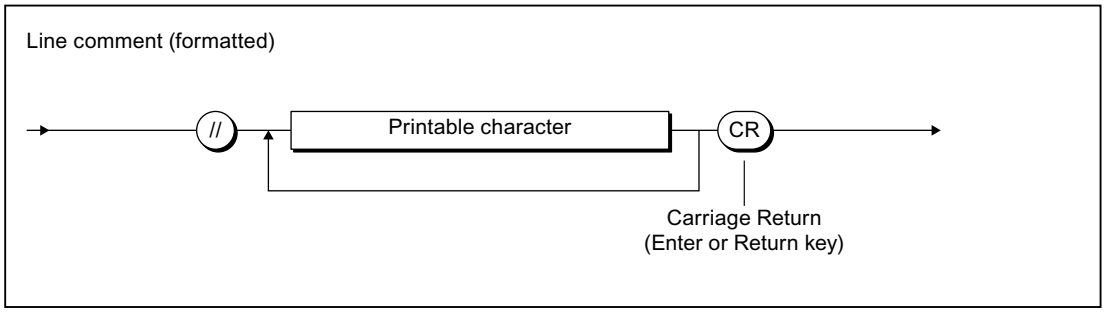


図 A-25 st_zei2_001

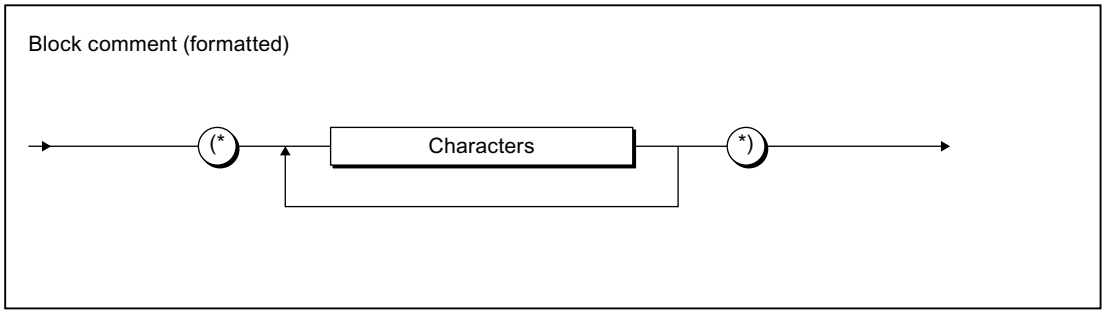
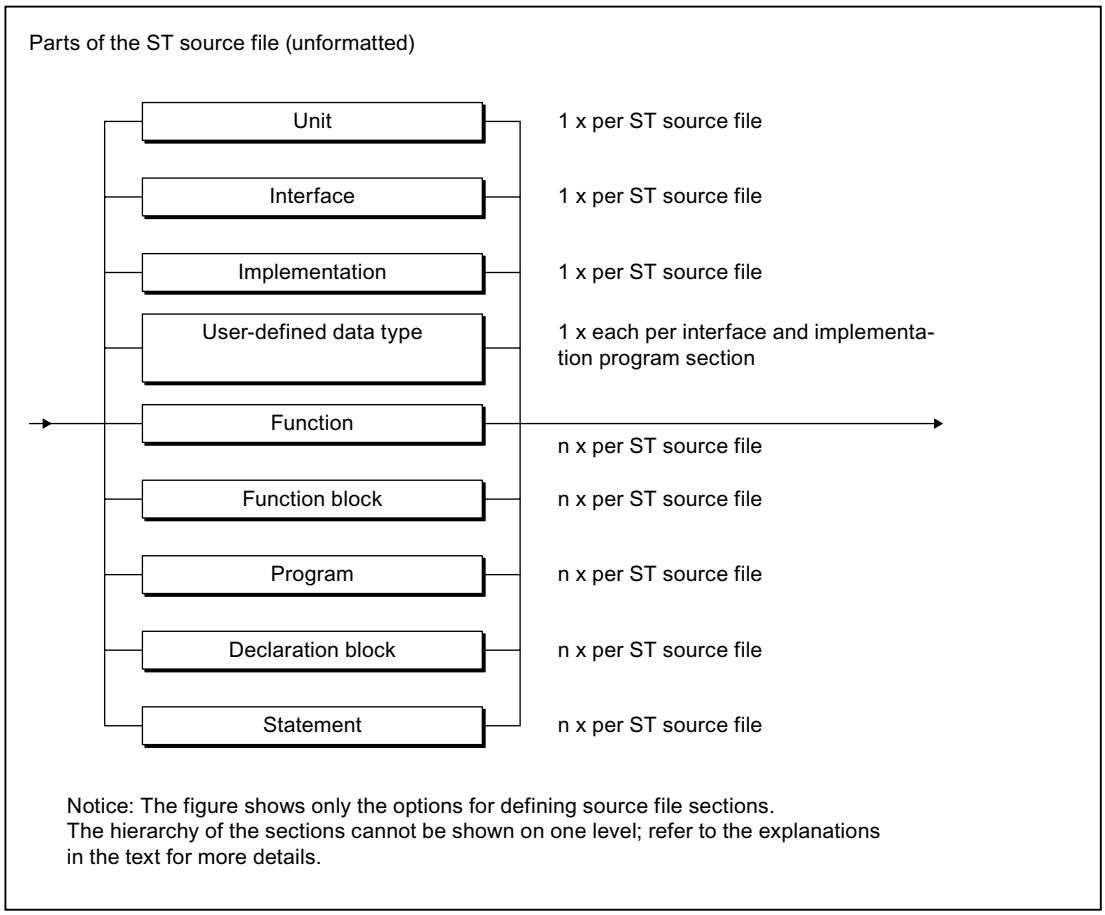


図 A-26 st_blo1_001

A.1.3.4 ST ソースファイルセクション



st_pro2_001

A.1.3.5 ST ソースファイルの構造

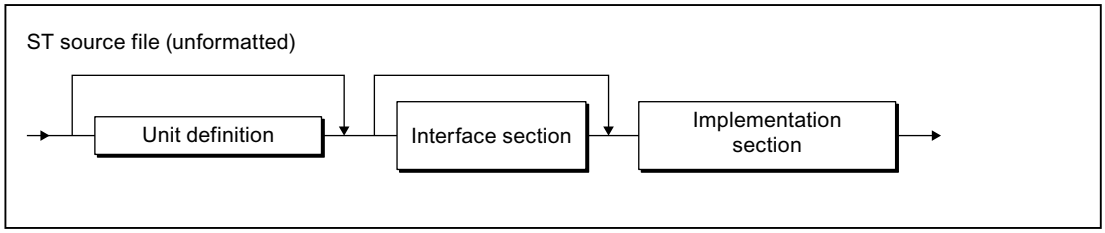


図 A-27 st_pro1_001

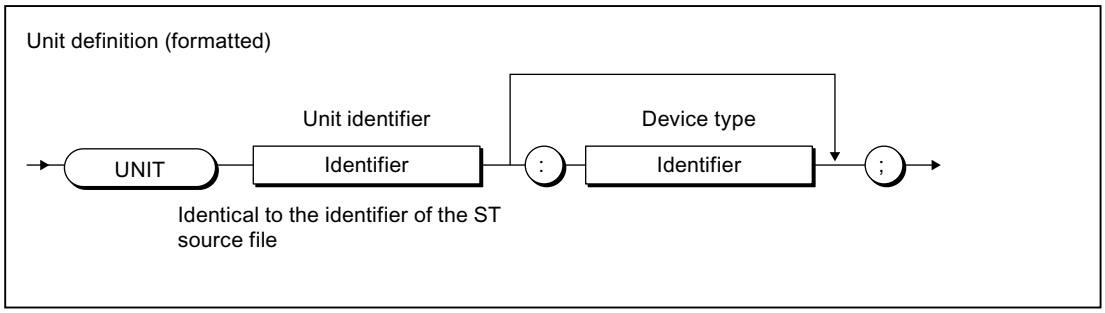


図 A-28 st_uni1_001

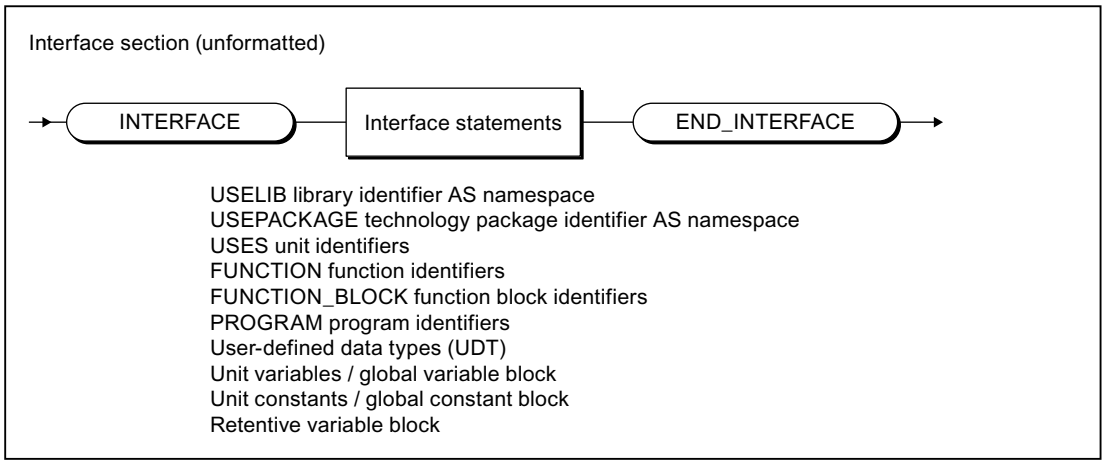


図 A-29 st_int3_012

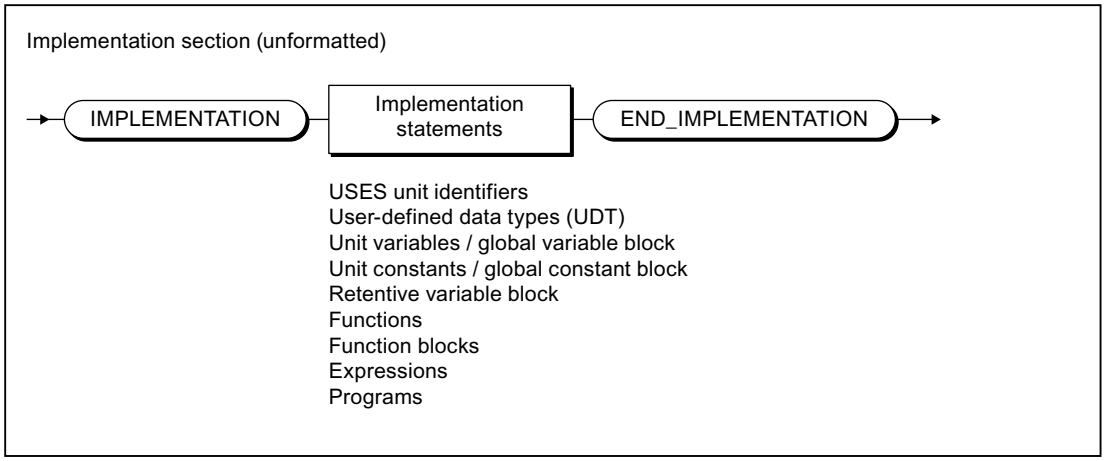


図 A-30 st_impl_088

A.1.3.6 プログラムオーガニゼーションユニット(POU)

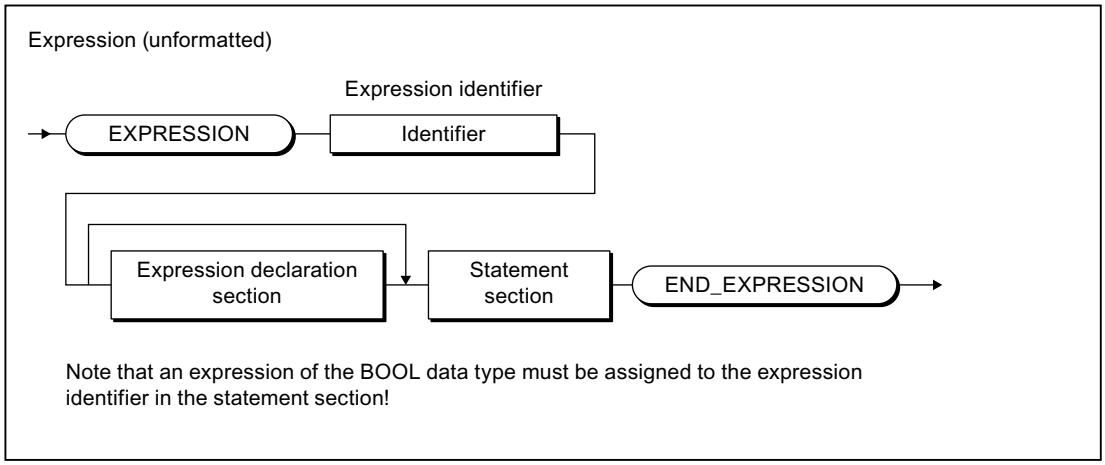


図 A-31 式

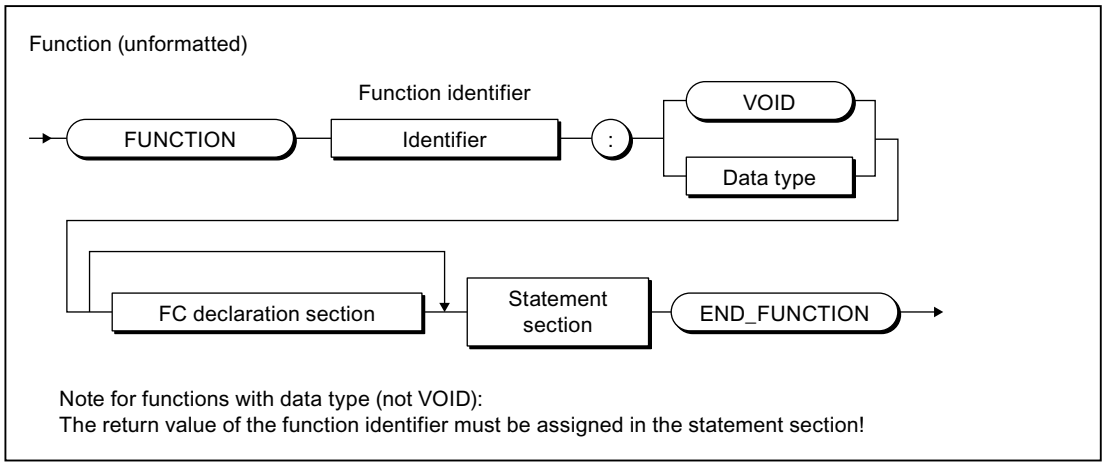


図 A-32 ファンクション(FC)

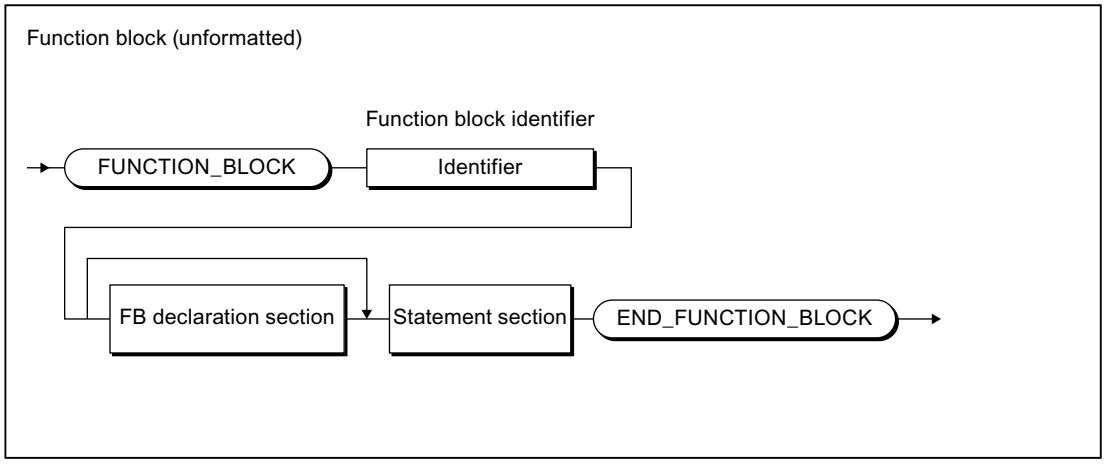


図 A-33 ファクションブロック(FB)

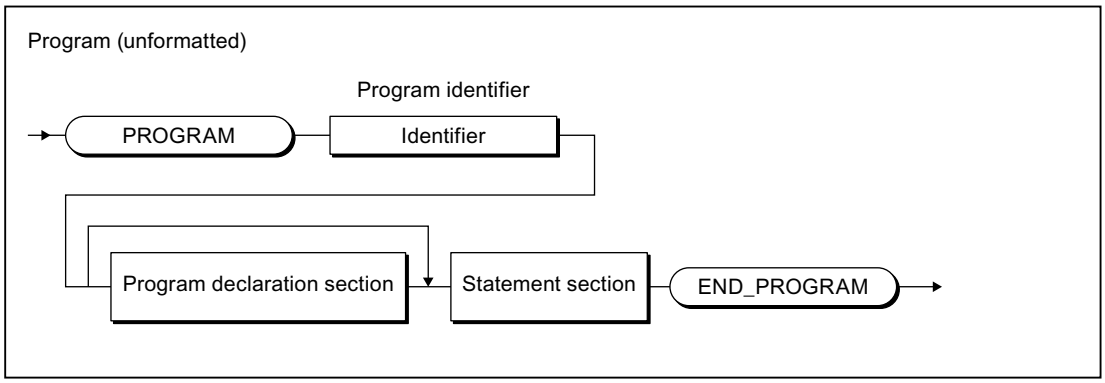


図 A-34　　プログラム

A.1.3.7　宣言セクション

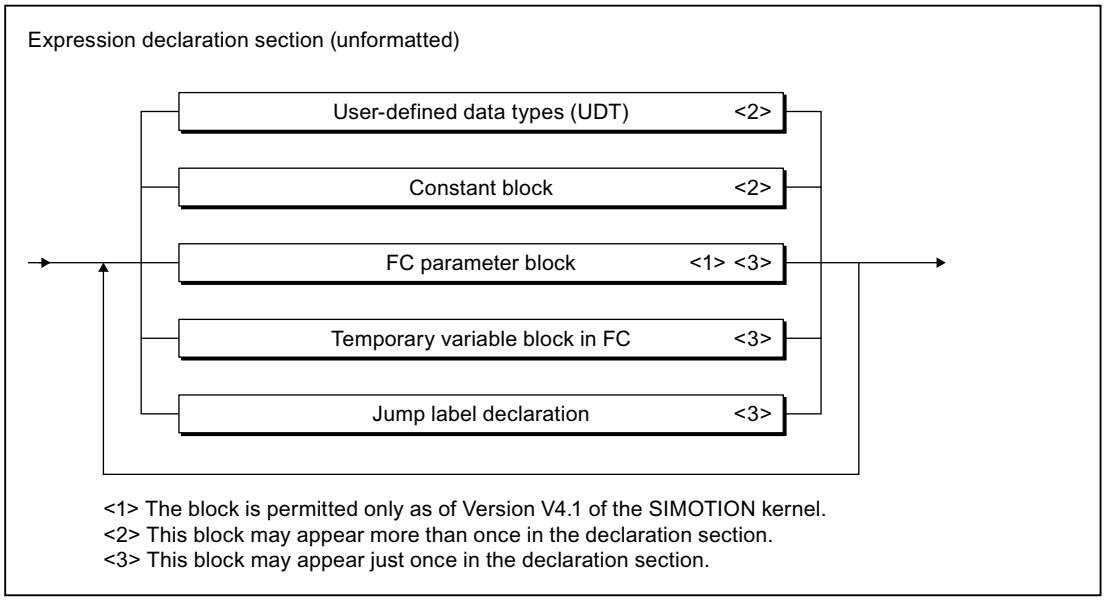


図 A-35　　st_exv1_001

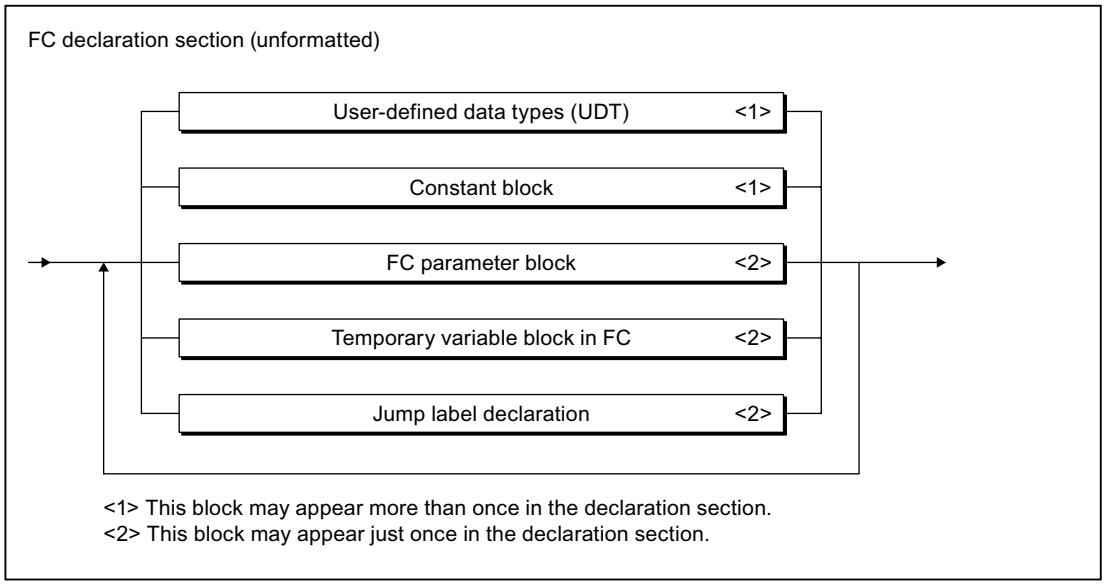


図 A-36 st_fcv1_001

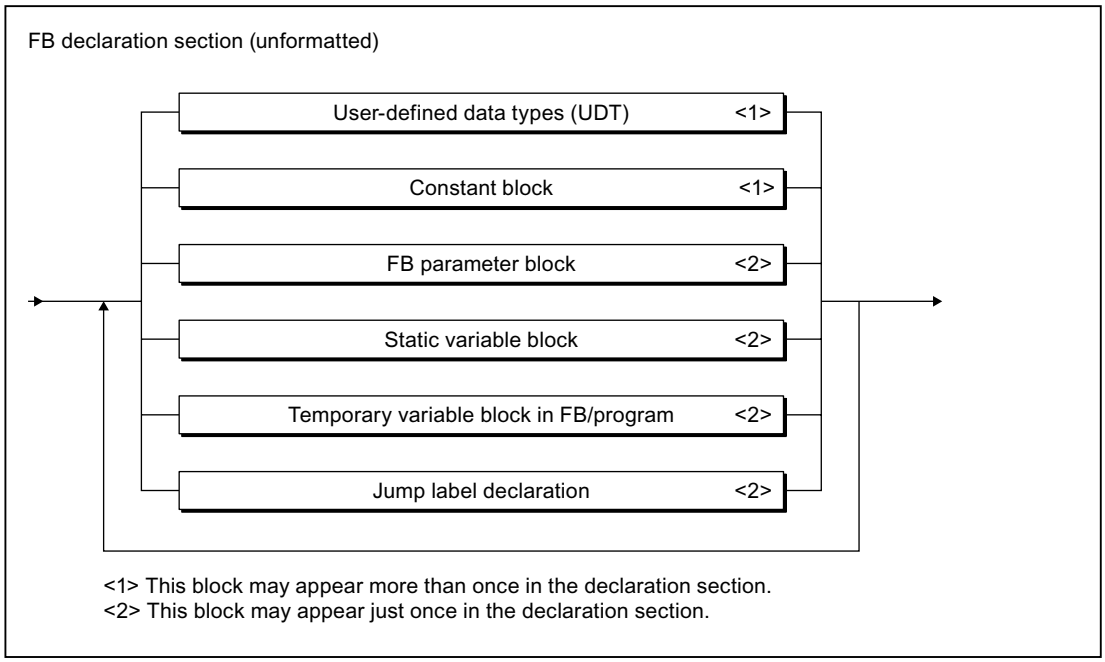


図 A-37 st_fbv1_001

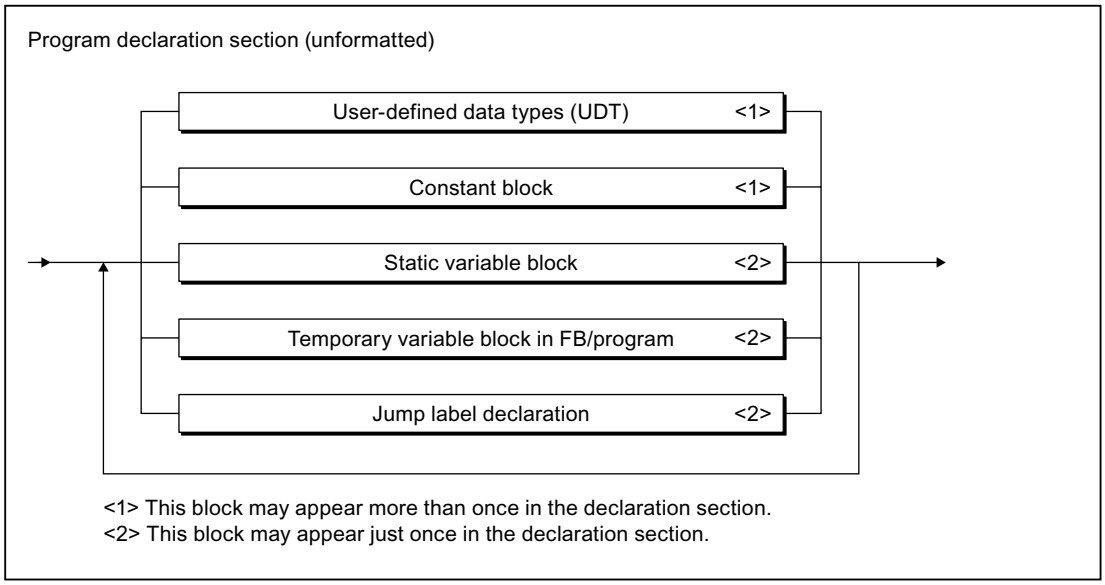


図 A-38　　st_tei1_001

A.1.3.8　宣言ブロックの構造

定数ブロック

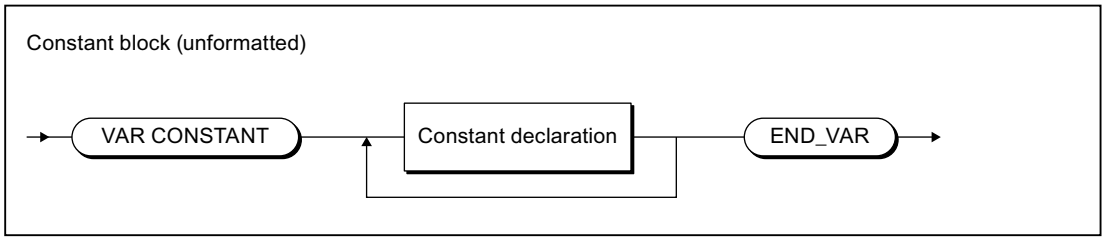


図 A-39　　定数ブロック

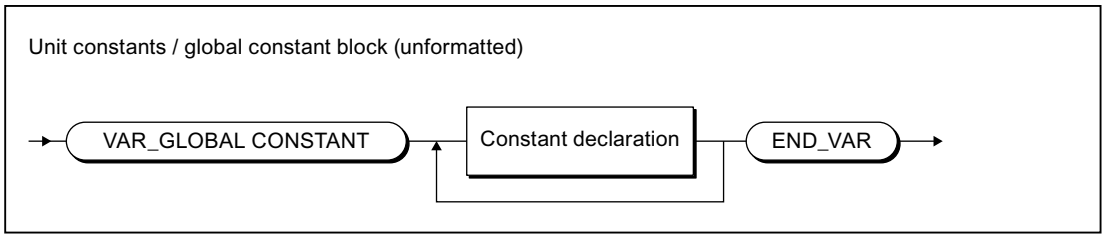


図 A-40　　ユニット定数/グローバル定数ブロック

変数ブロック

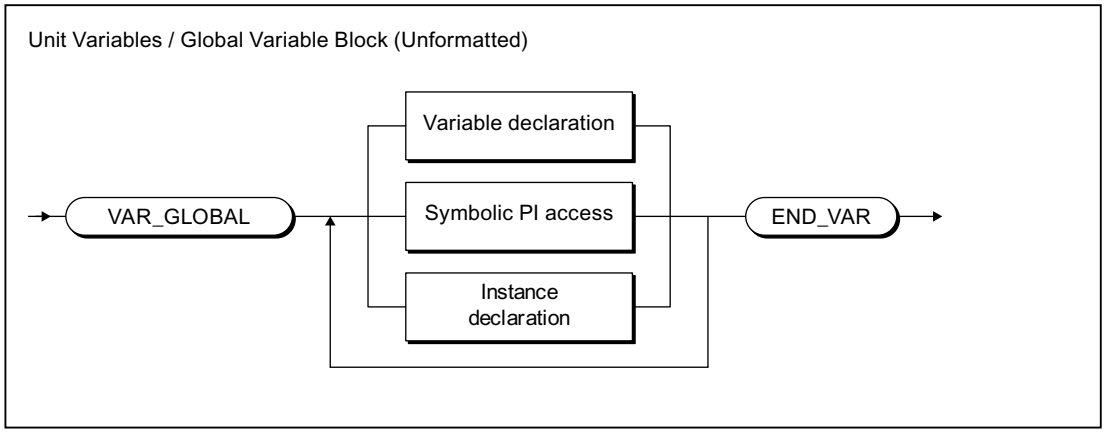


図 A-41　　ユニット変数/グローバル変数ブロック

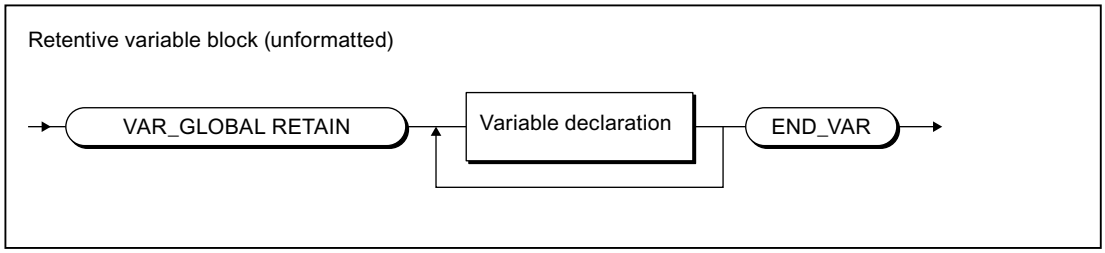


図 A-42　　保持型変数ブロック

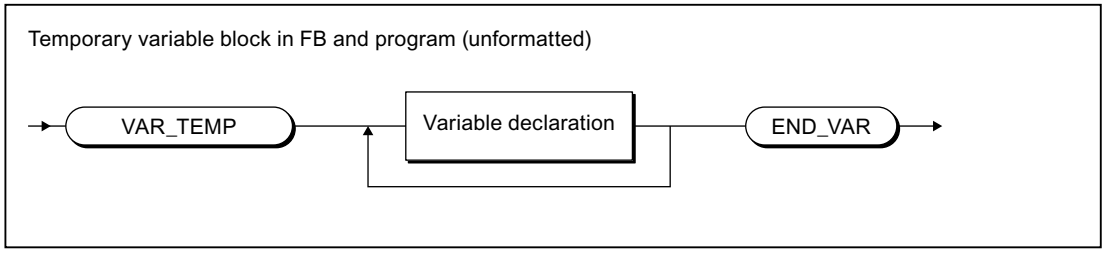


図 A-43　　FC のテンポラリ変数ブロック

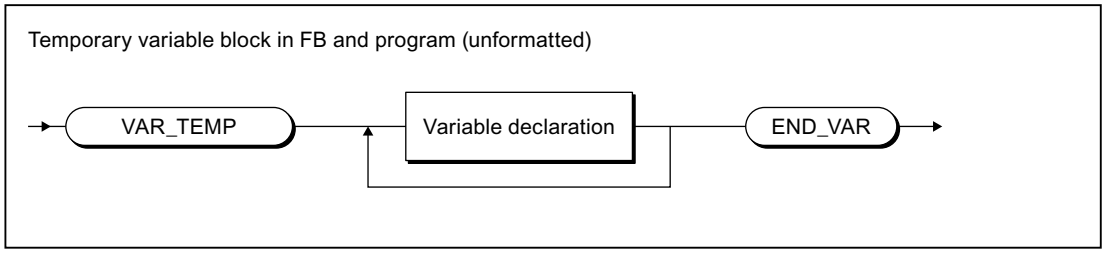


図 A-44　　FB/プログラムのテンポラリ変数ブロック

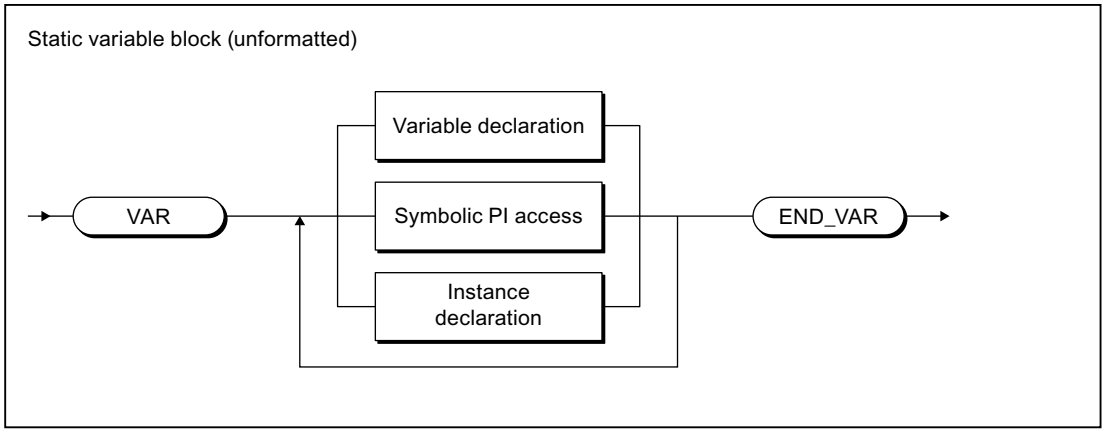


図 A-45　　スタティック変数ブロック

パラメータフィールド

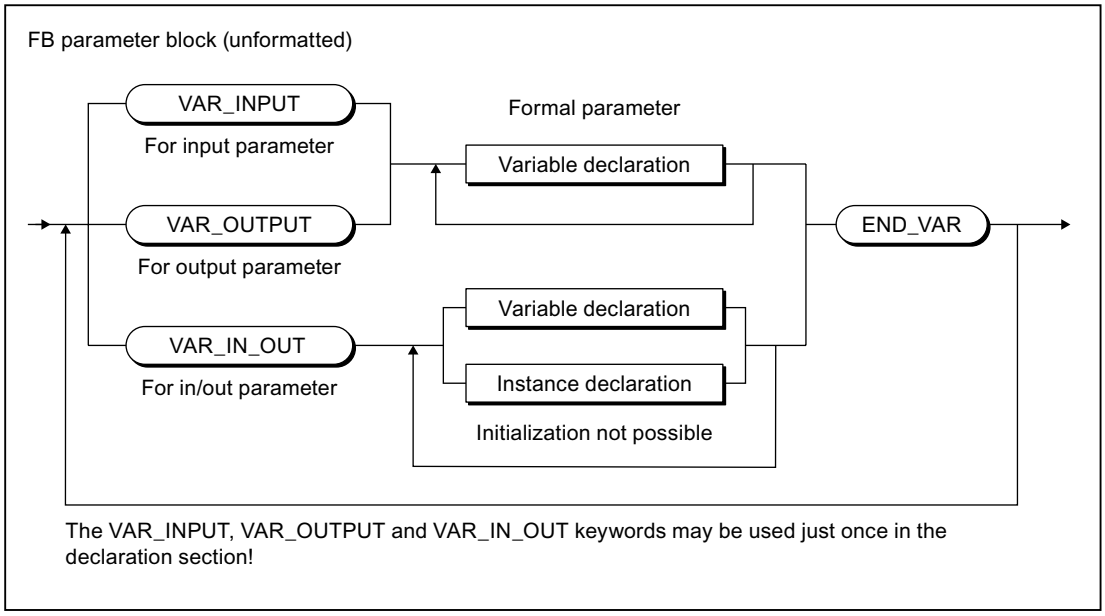


図 A-46　　st_par1_001

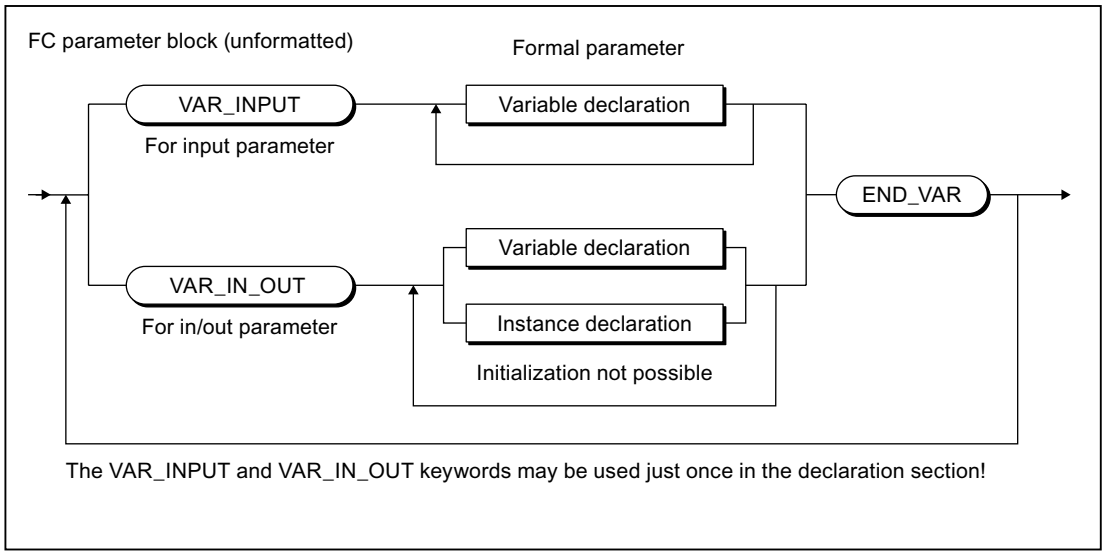


図 A-47 st_par2_001

ジャンプラベル

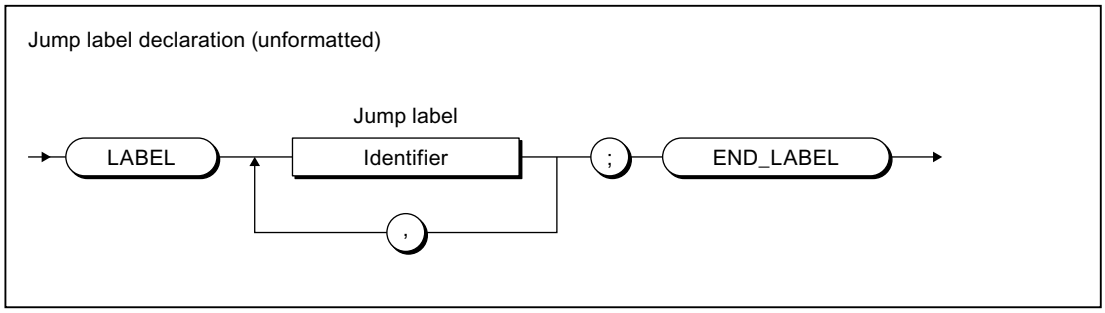


図 A-48 st_lab1_101

宣言

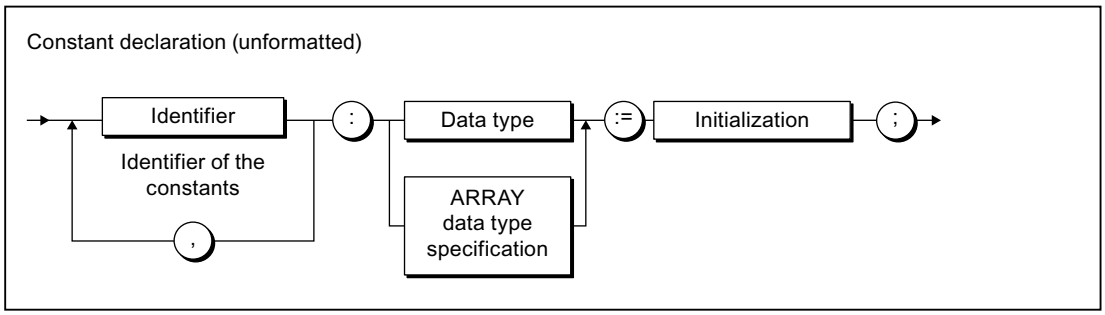


図 A-49　　定数の宣言

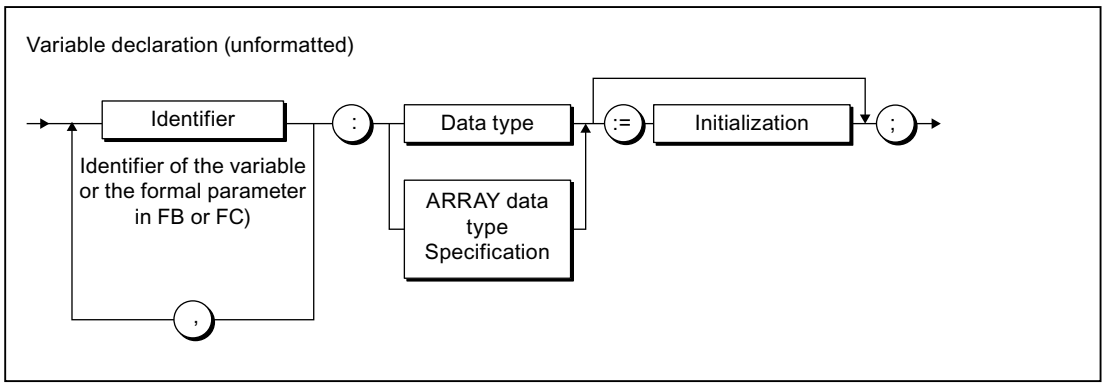


図 A-50　　変数宣言

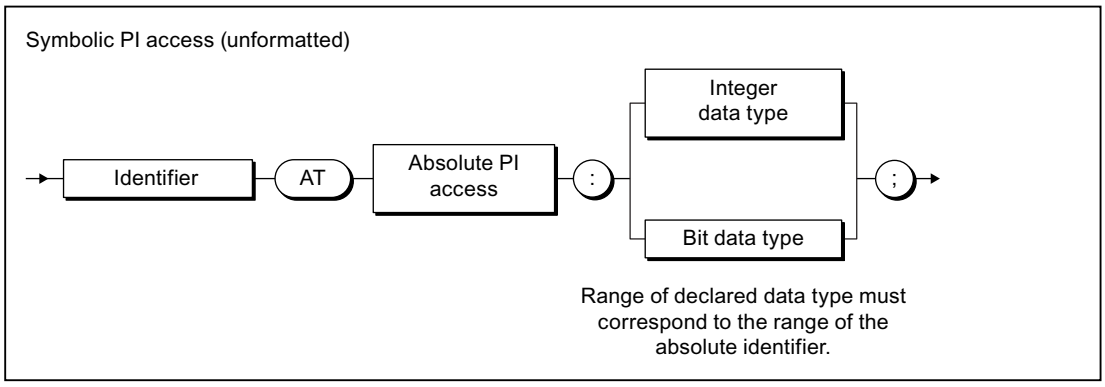


図 A-51　　シンボリック PI アクセス

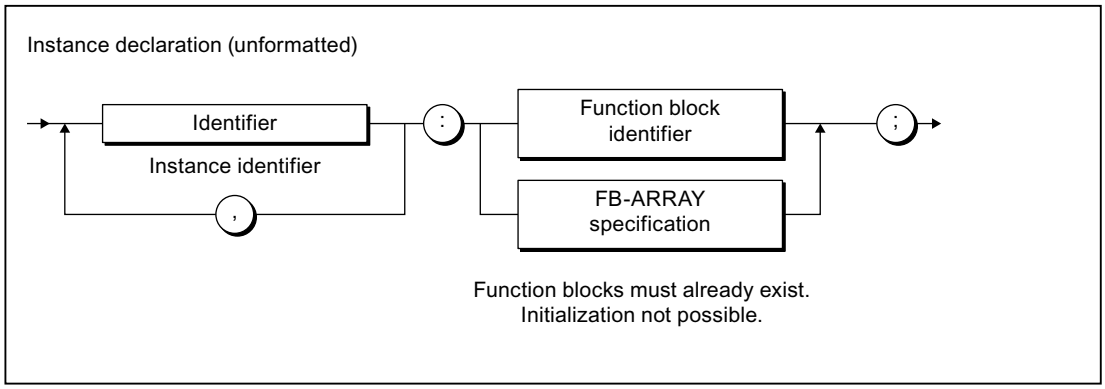


図 A-52　　インスタンスの宣言

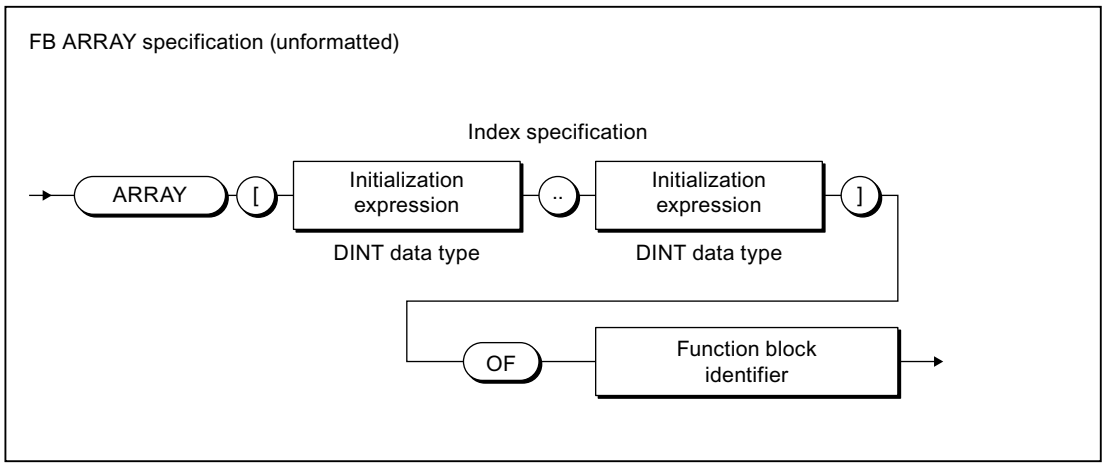


図 A-53　　FB ARRAY 指定

初期化

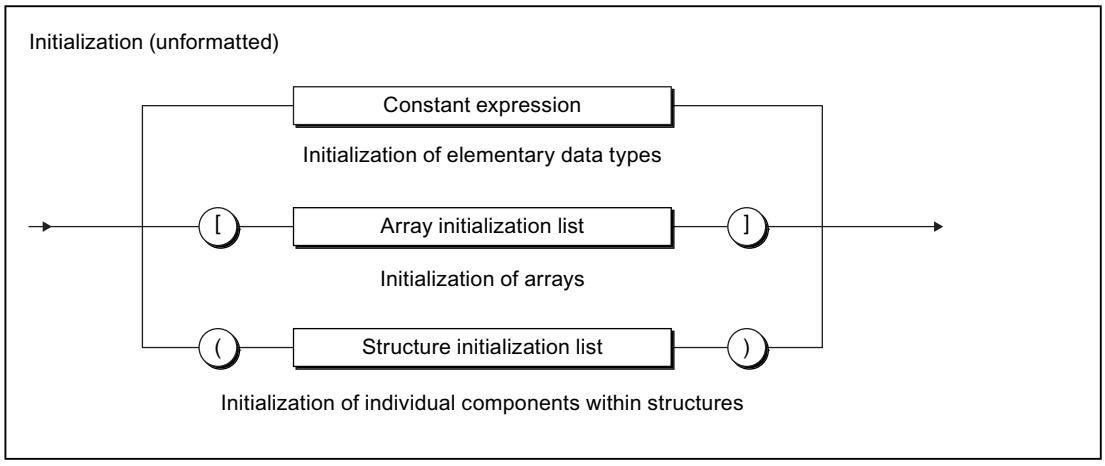


図 A-54　　初期化

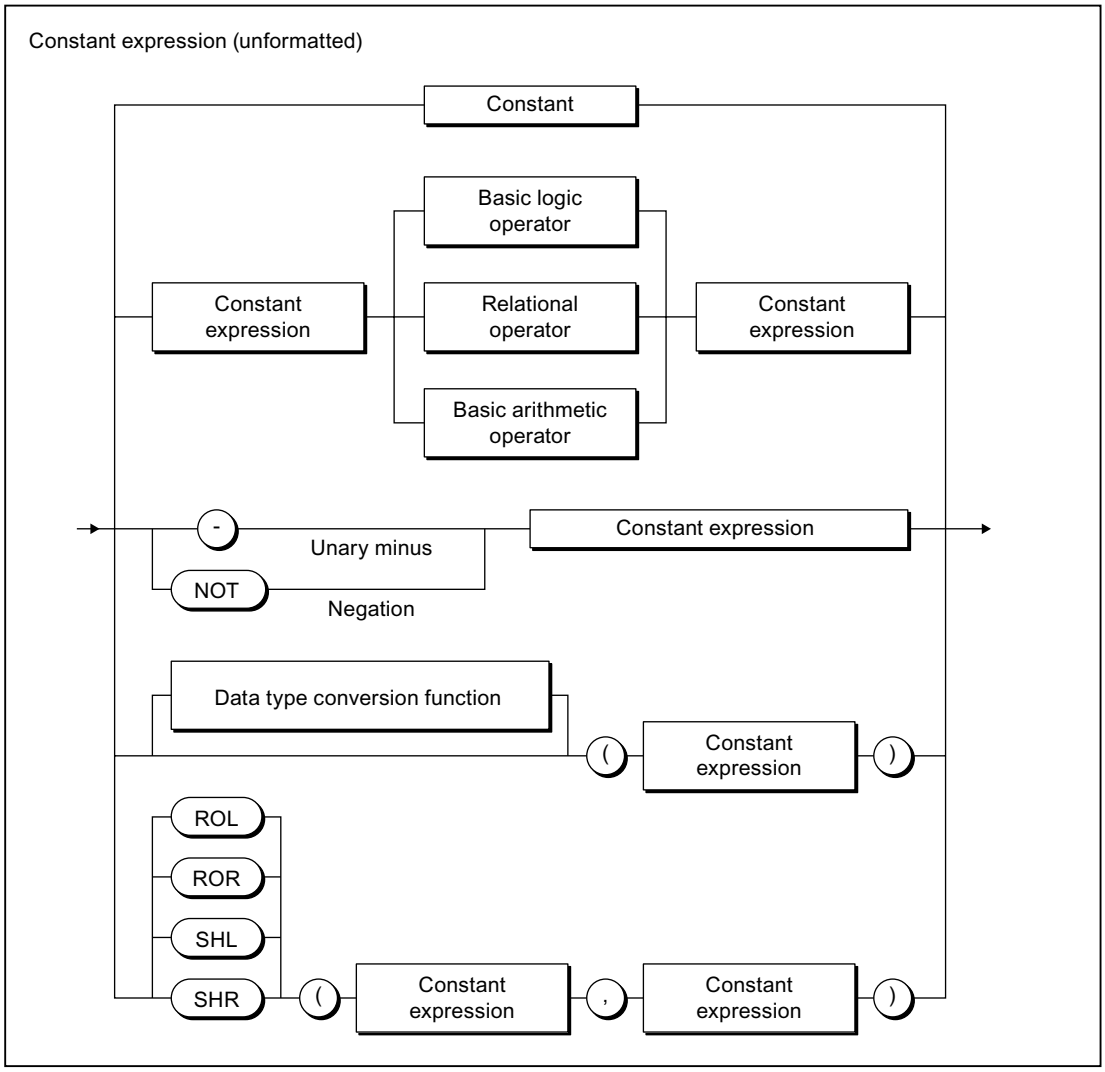


図 A-55　　定数式

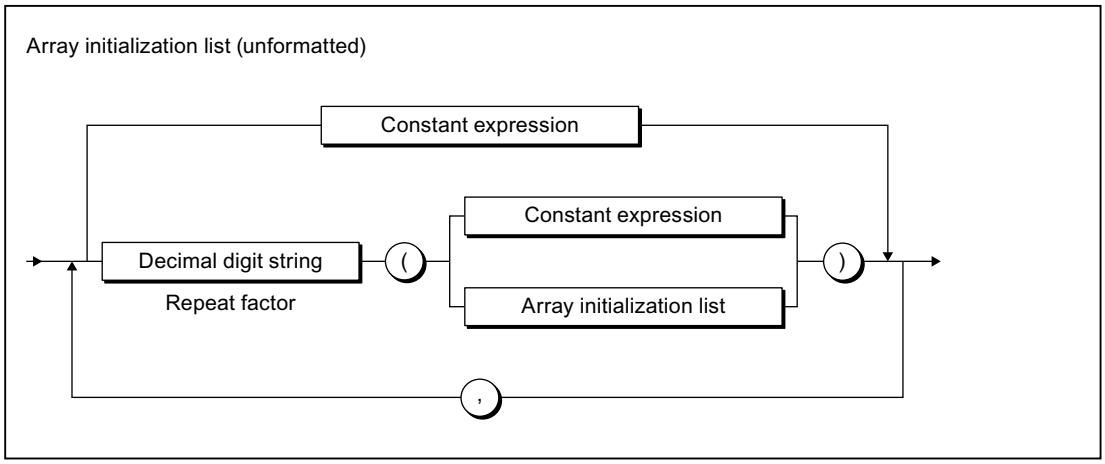


図 A-56　　配列初期化リスト

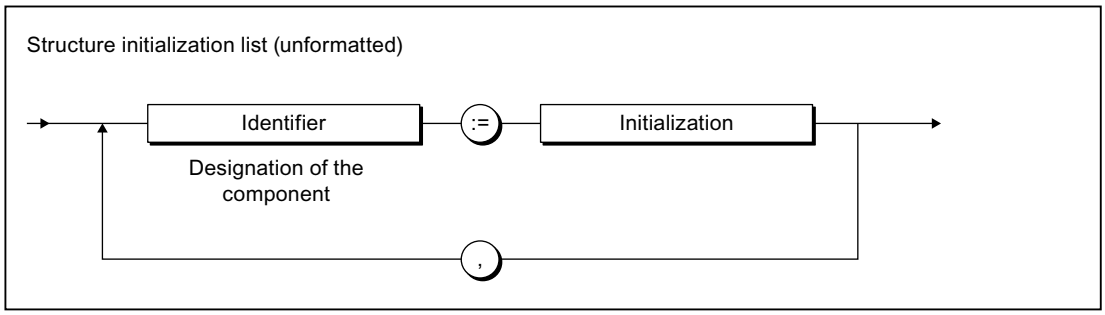


図 A-57　構造体初期化リスト

A.1.3.9　ST のデータタイプ

基本データタイプ

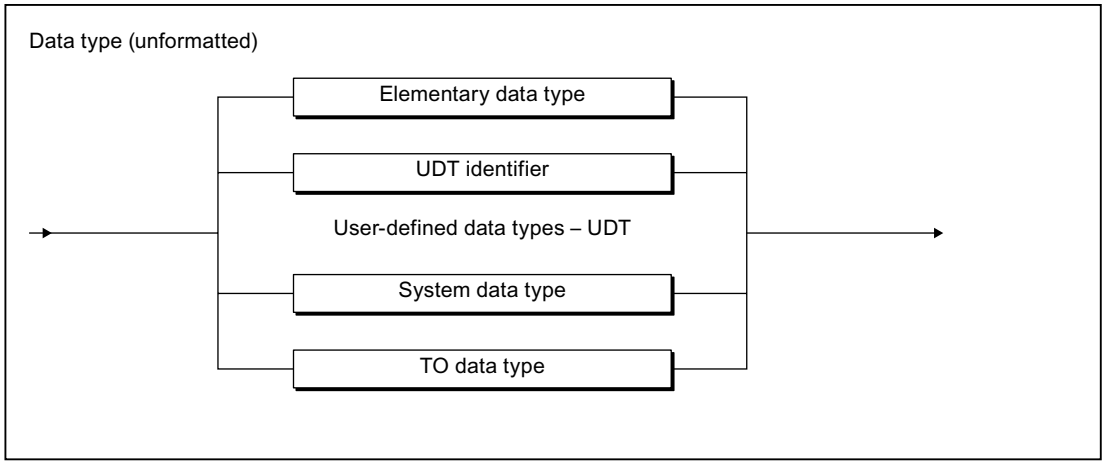


図 A-58　st_dat5_001

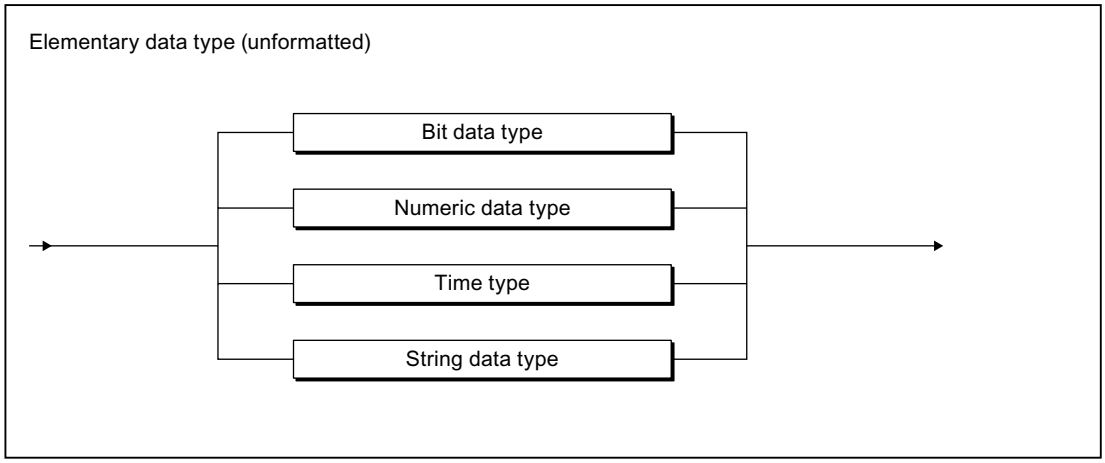


図 A-59　st_ele1_001

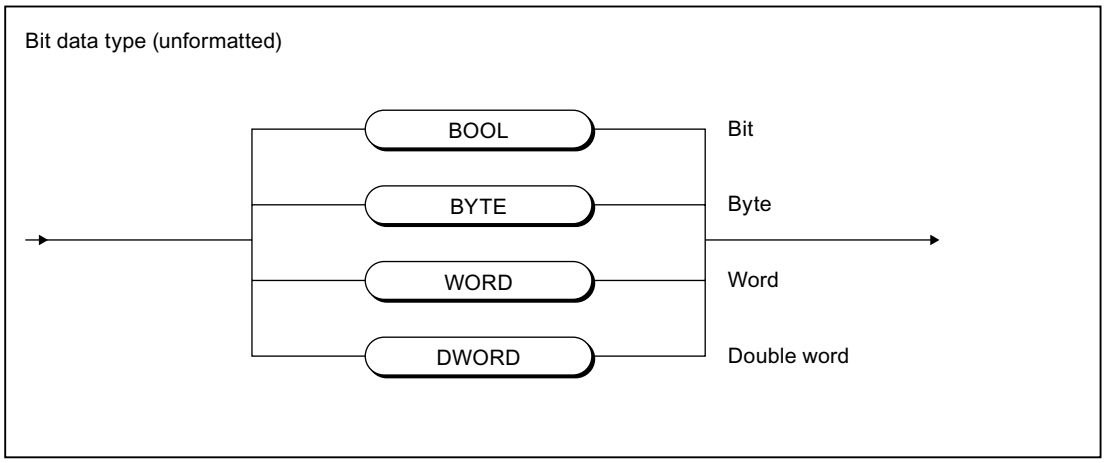


図 A-60 st_bit1_001

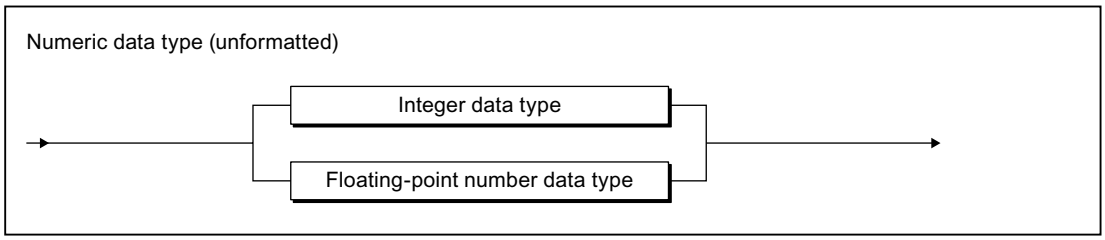


図 A-61 st_num2_001

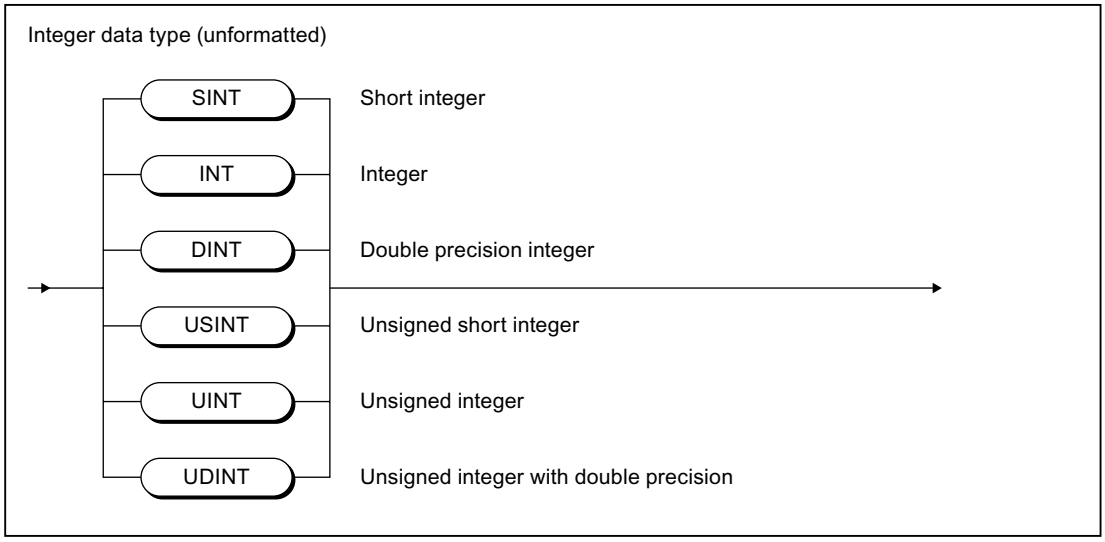


図 A-62 st_num3_001

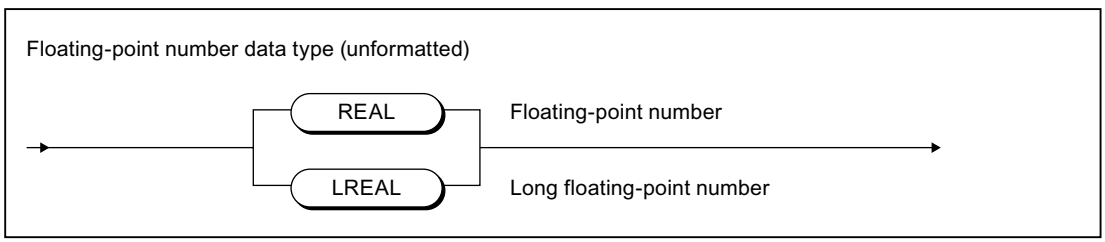


図 A-63 st_num4_001

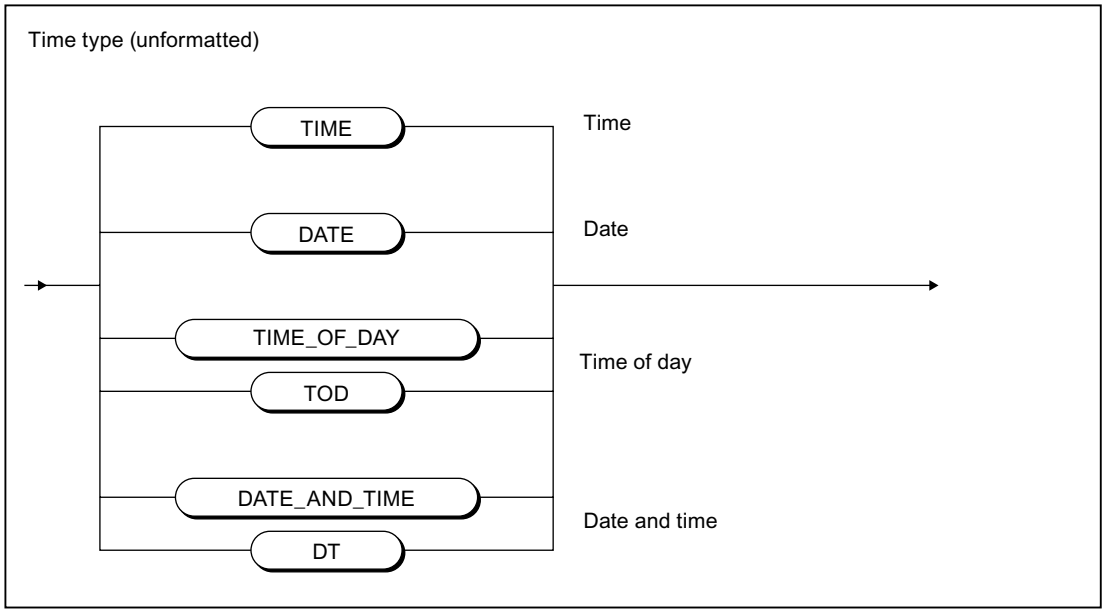


図 A-64　　st_zei3_001

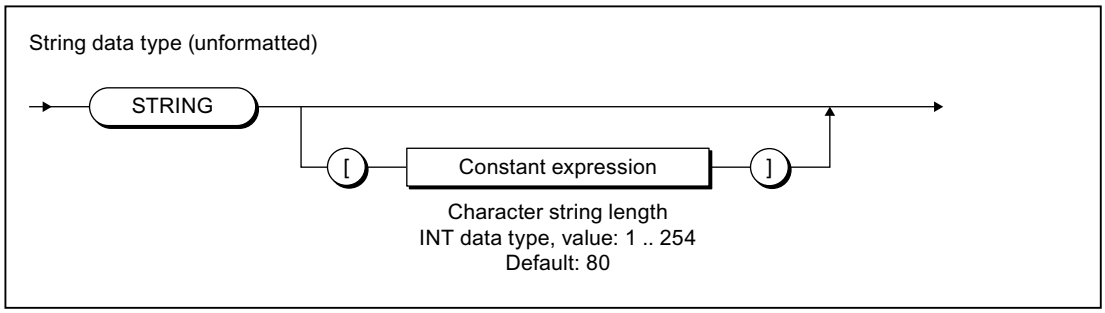


図 A-65　　STRING データタイプ

ユーザ定義データタイプ

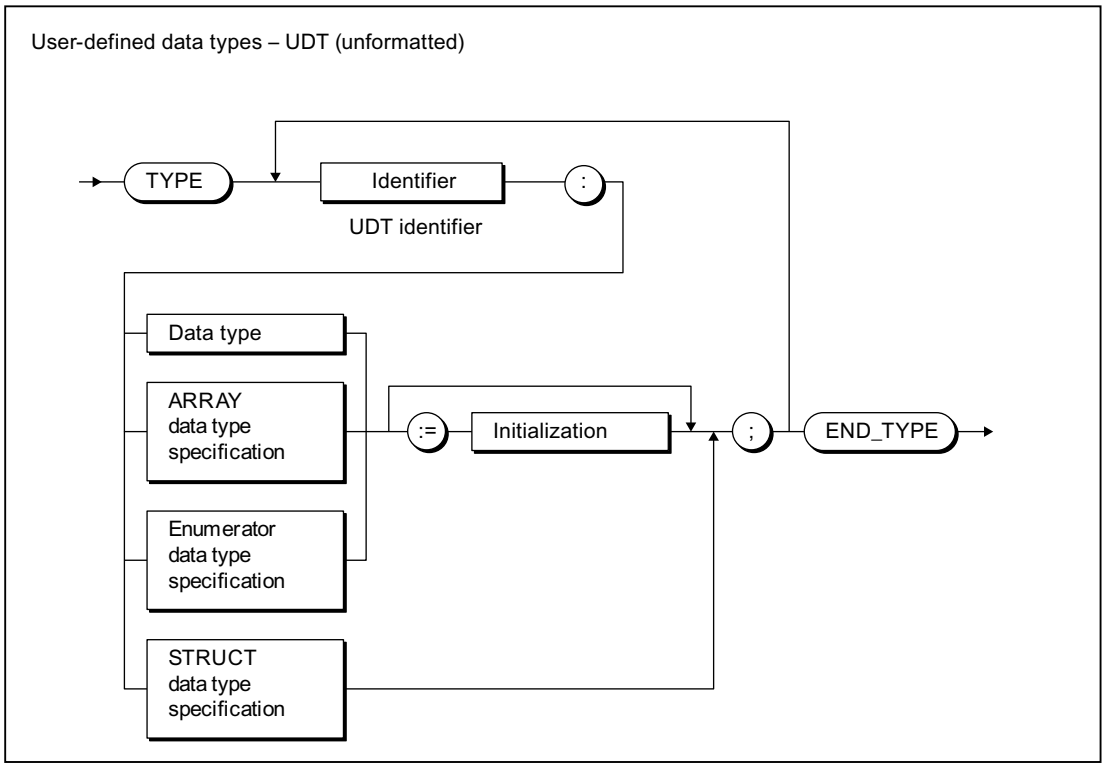


図 A-66　　ユーザ定義データタイプ

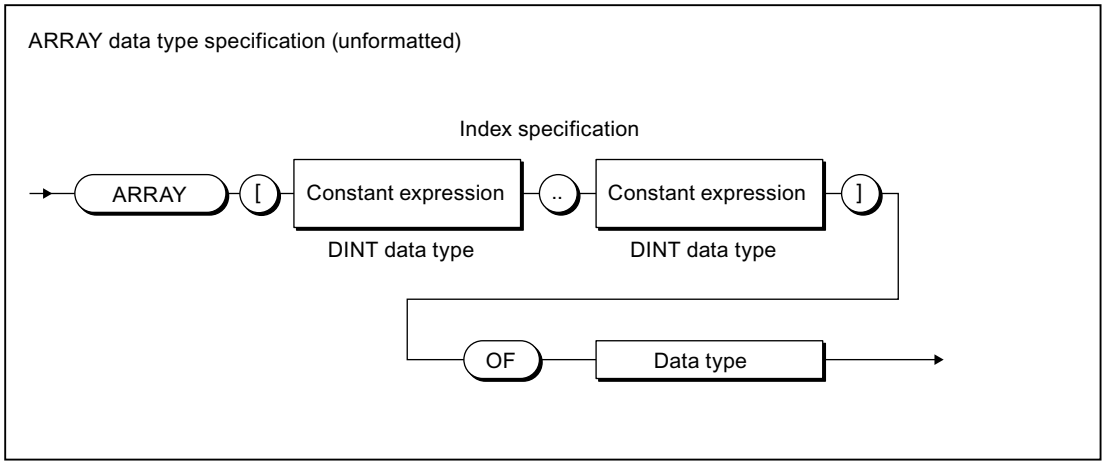


図 A-67　　配列データタイプ指定

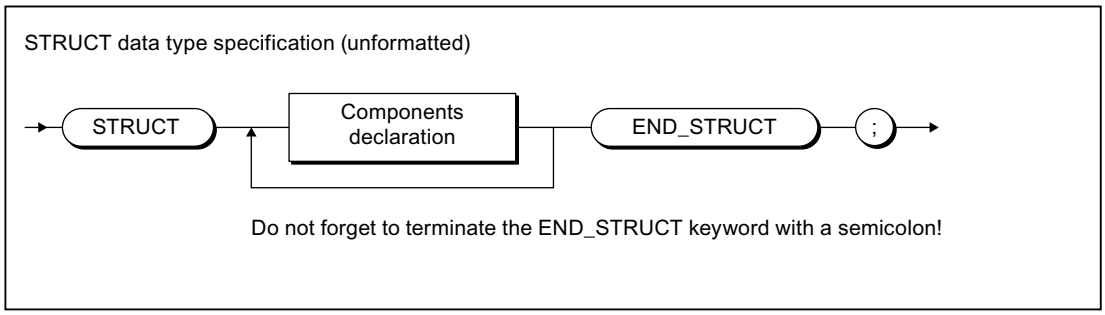


図 A-68　　STRUCT データタイプ指定

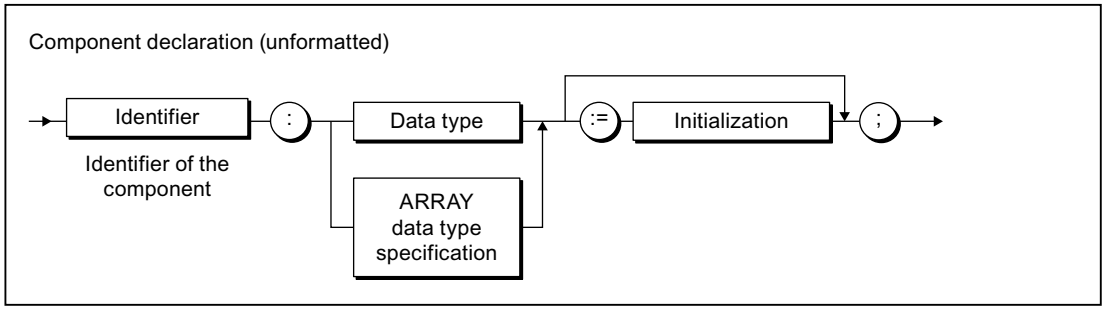


図 A-69　　コンポーネント宣言

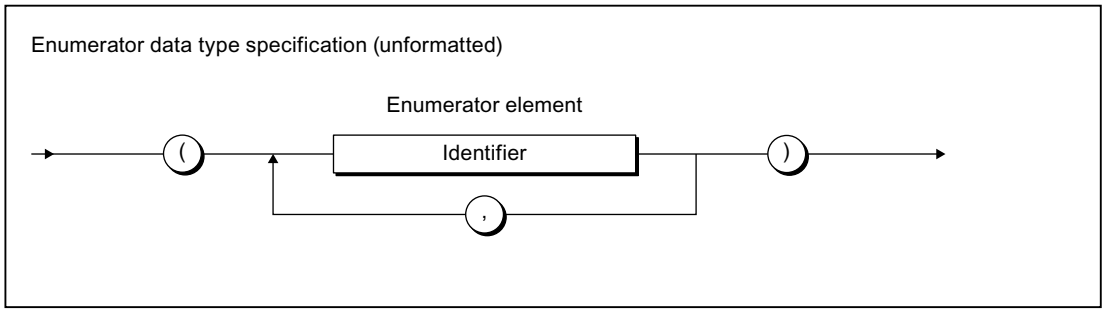


図 A-70　　列挙子データタイプ指定

A.1.3.10　ステートメントセクション

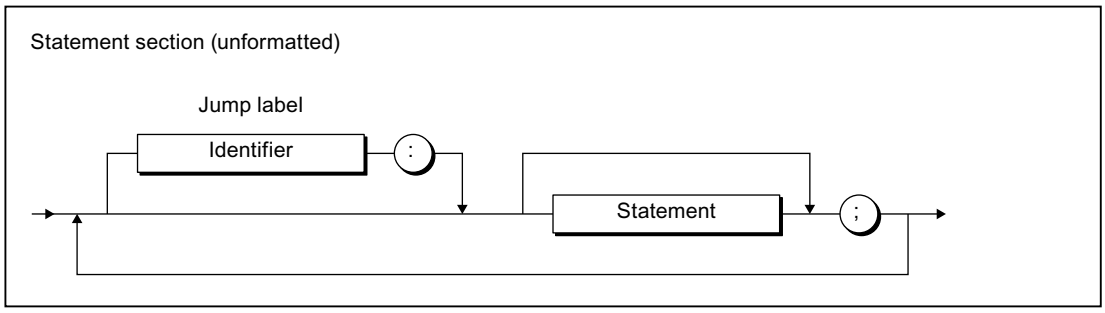


図 A-71　　st_anw2_002

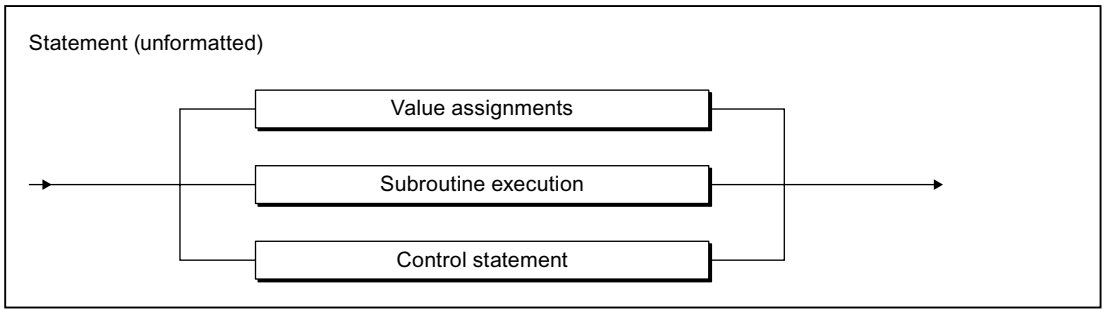


図 A-72　　st_anw3_001

A.1.3.11　値割り付けおよび演算子

値割り付けおよび式

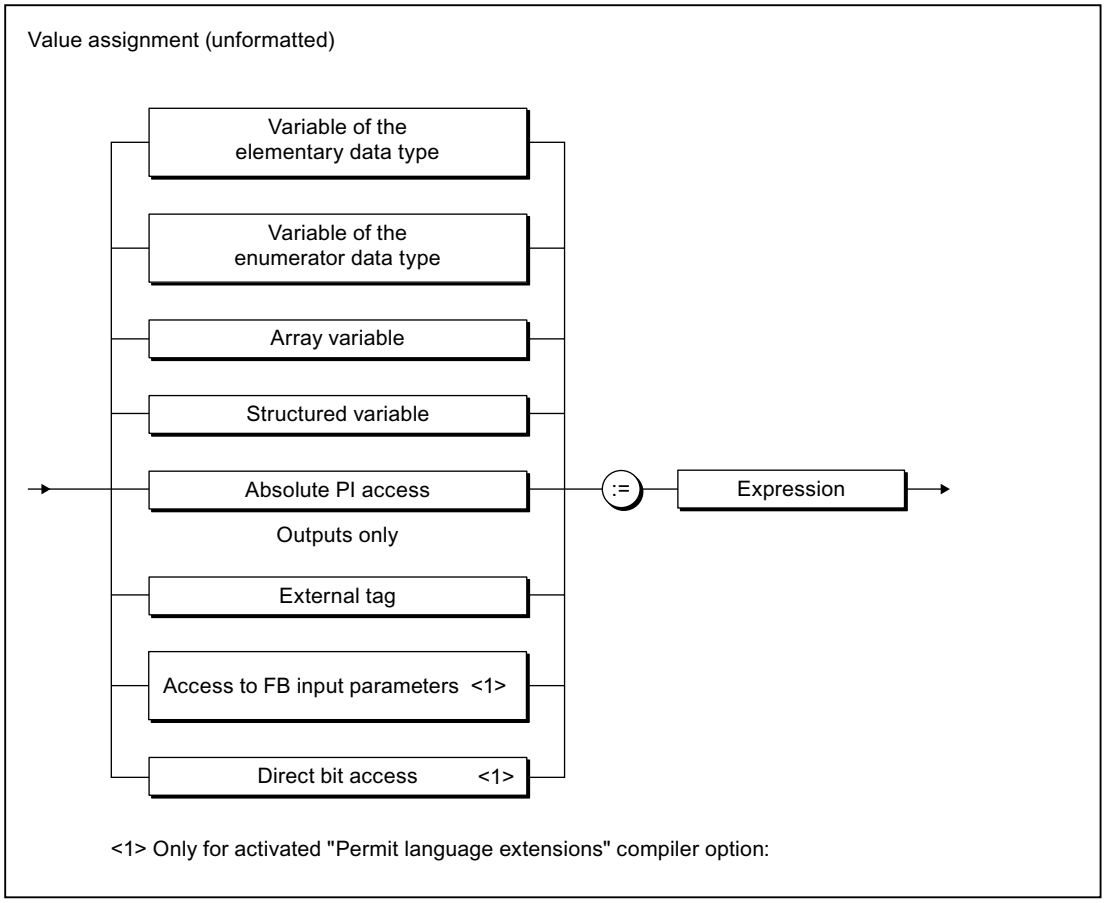


図 A-73　　値割り付け

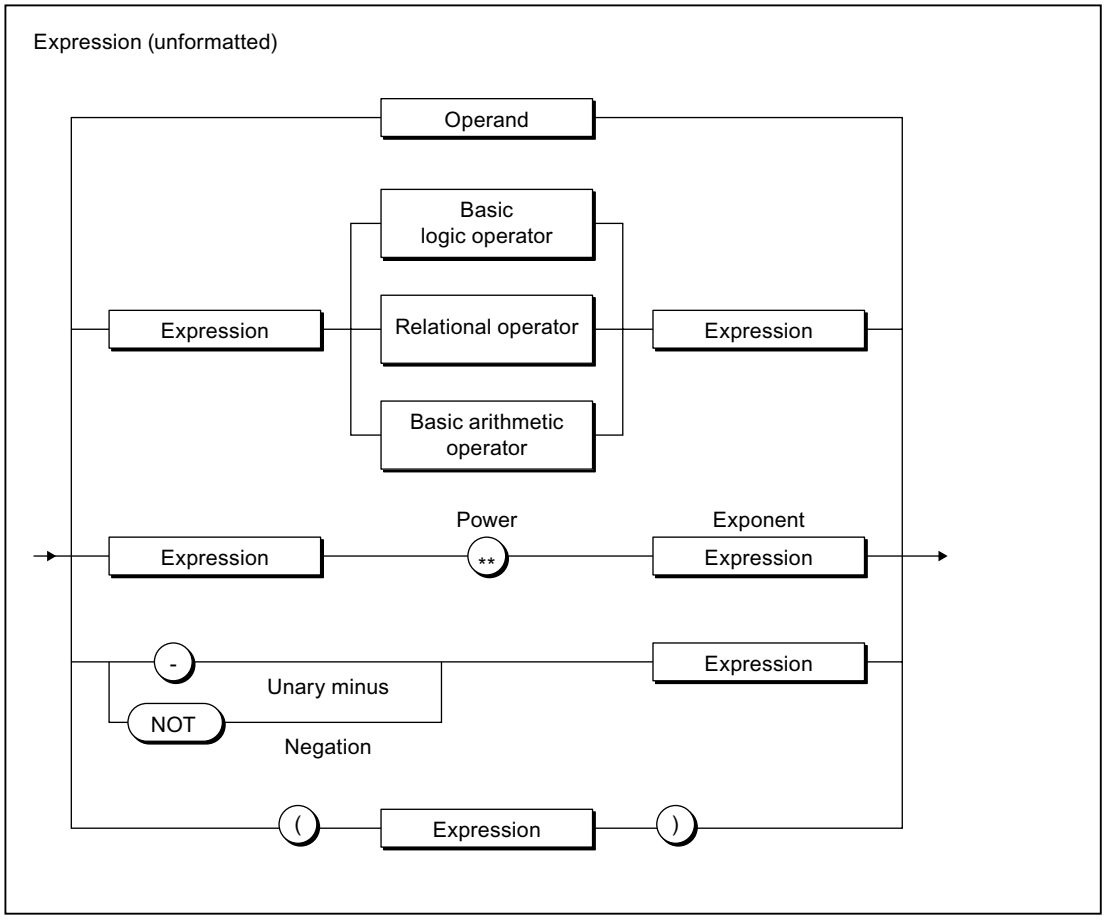

Expression (unformatted)
Operand
Basic
logic operator
Expression
Relational operator
Expression
Basic arithmetic
operator
Power
Exponent
Expression
**
Expression
-
Expression
Unary minus
NOT
Negation
(
Expression
)

図 A-74　　式

オペランド

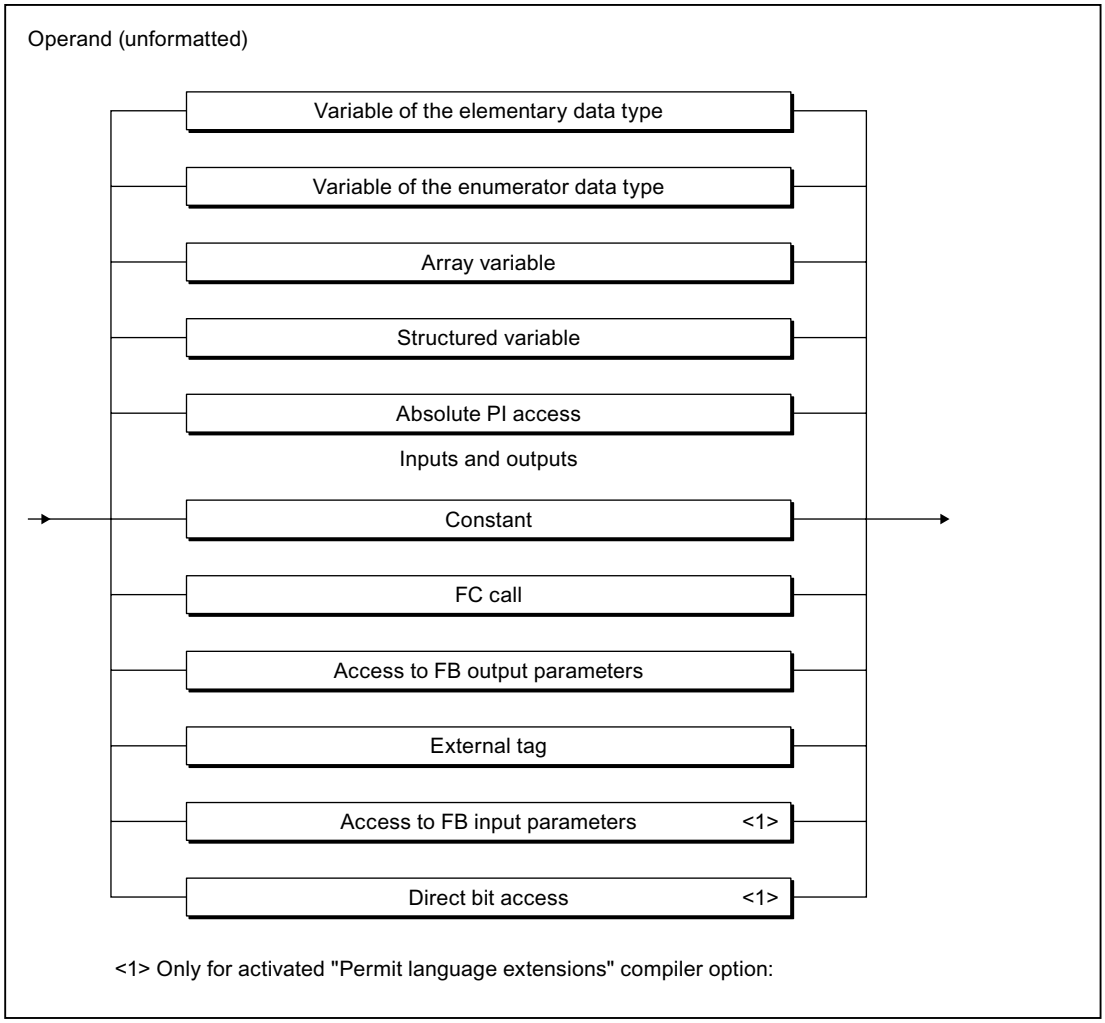


図 A-75　　オペランド

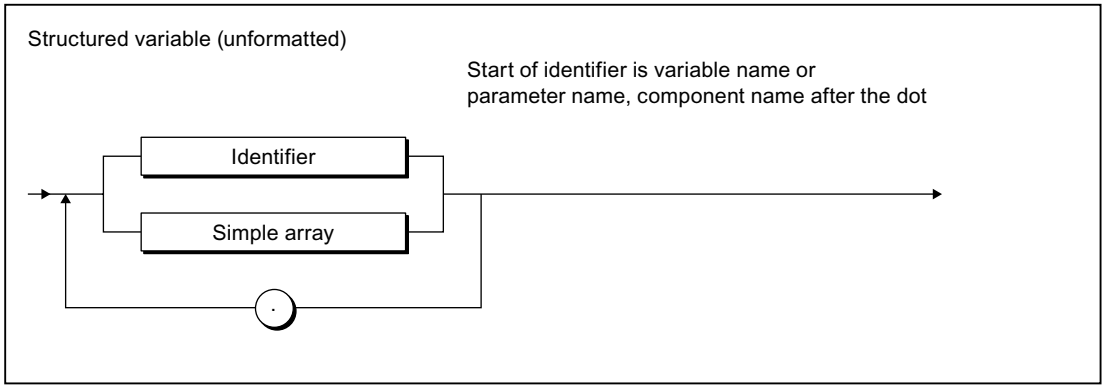


図 A-76　　構造体変数

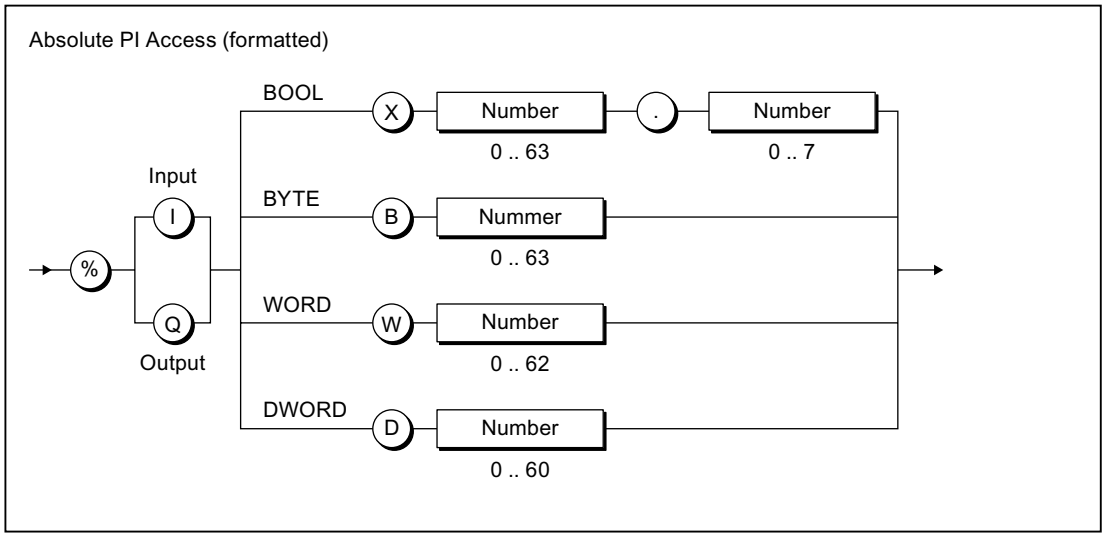


図 A-77　　絶対 PI アクセス

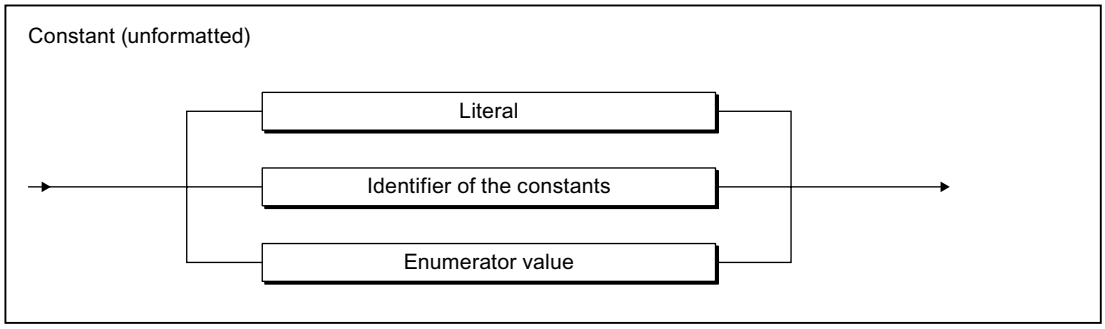


図 A-78　　定数

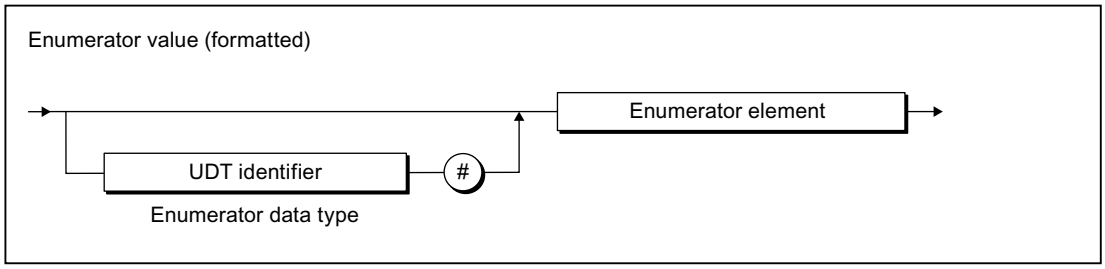


図 A-79　　列挙子値

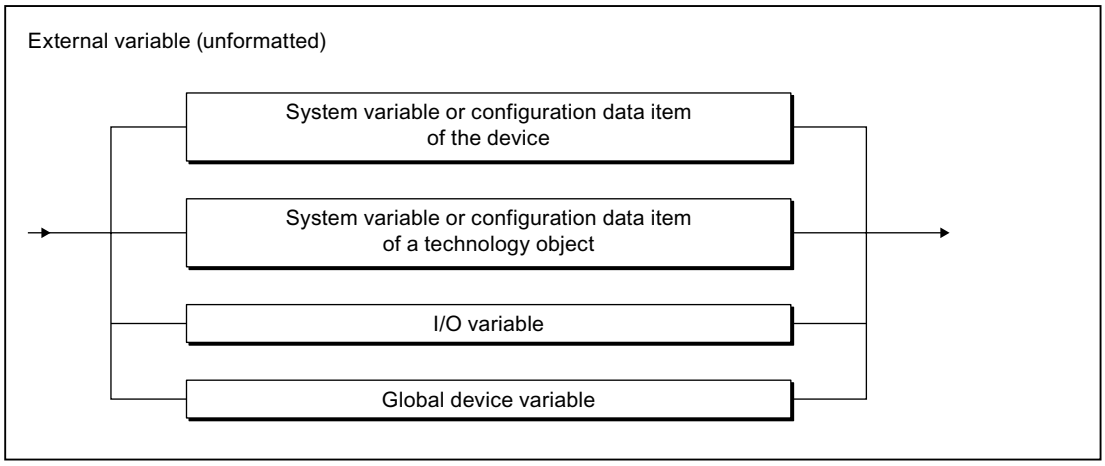


図 A-80　　外部タグ

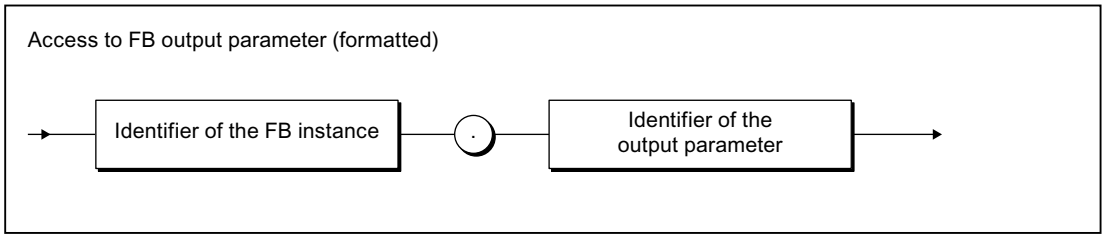


図 A-81　　FB 出力パラメータへのアクセス

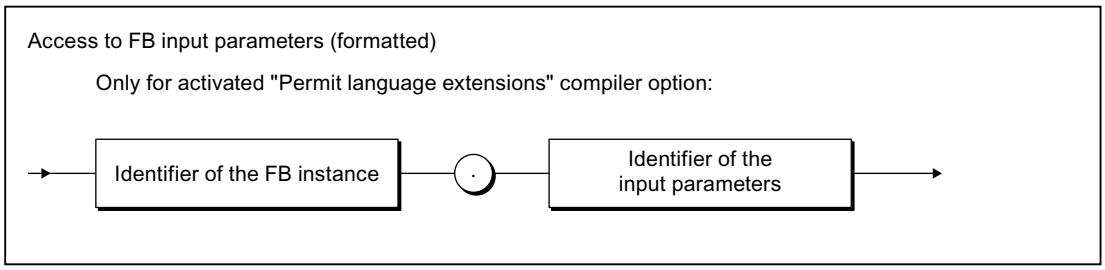


図 A-82　　FB 入力パラメータへのアクセス

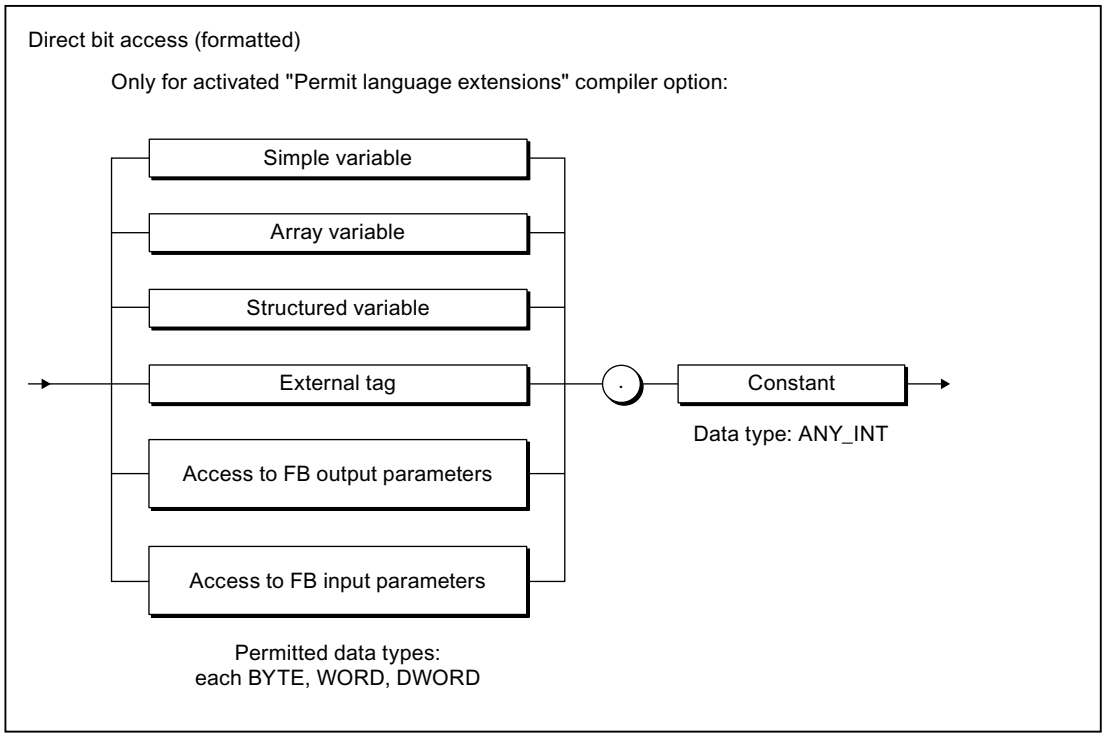


図 A-83　　ビットアクセス

演算子

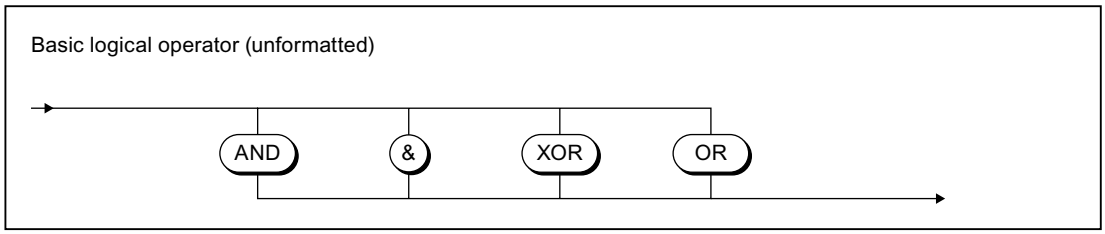


図 A-84　　st_log1_001

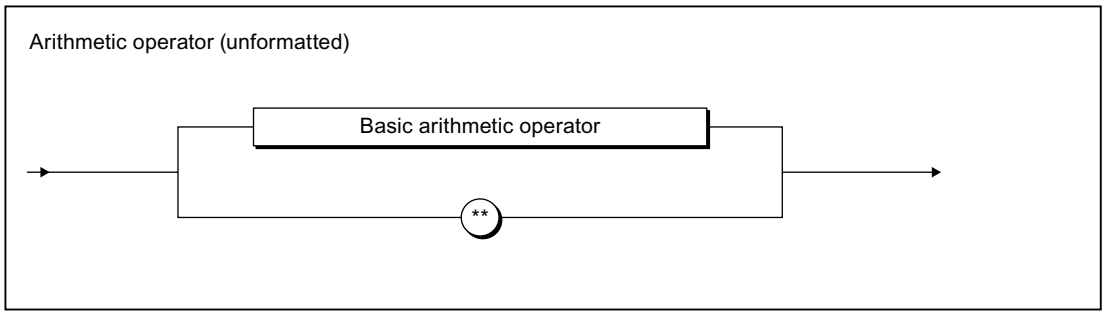


図 A-85　　算術演算子

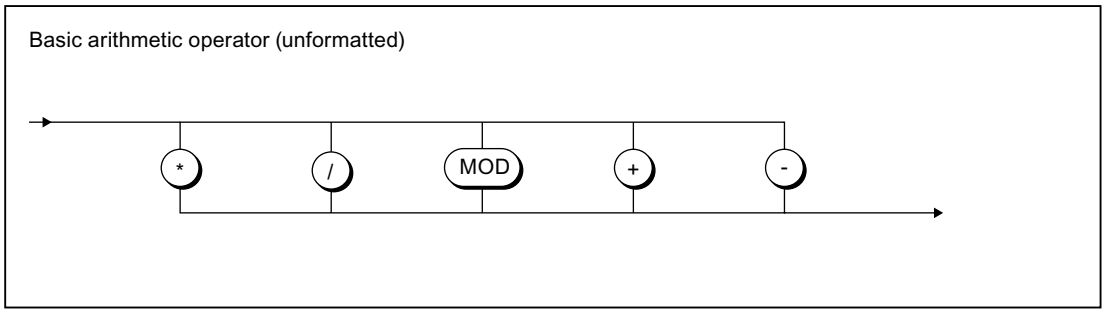


図 A-86　　基本算術演算子

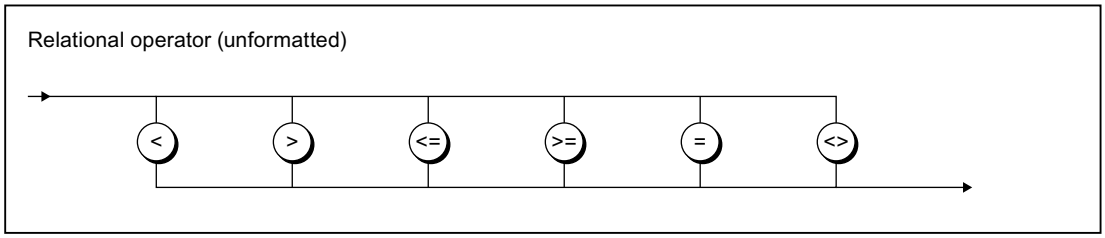


図 A-87　　関係演算子

A.1.3.12　ファンクションおよびファンクションブロックの呼び出し

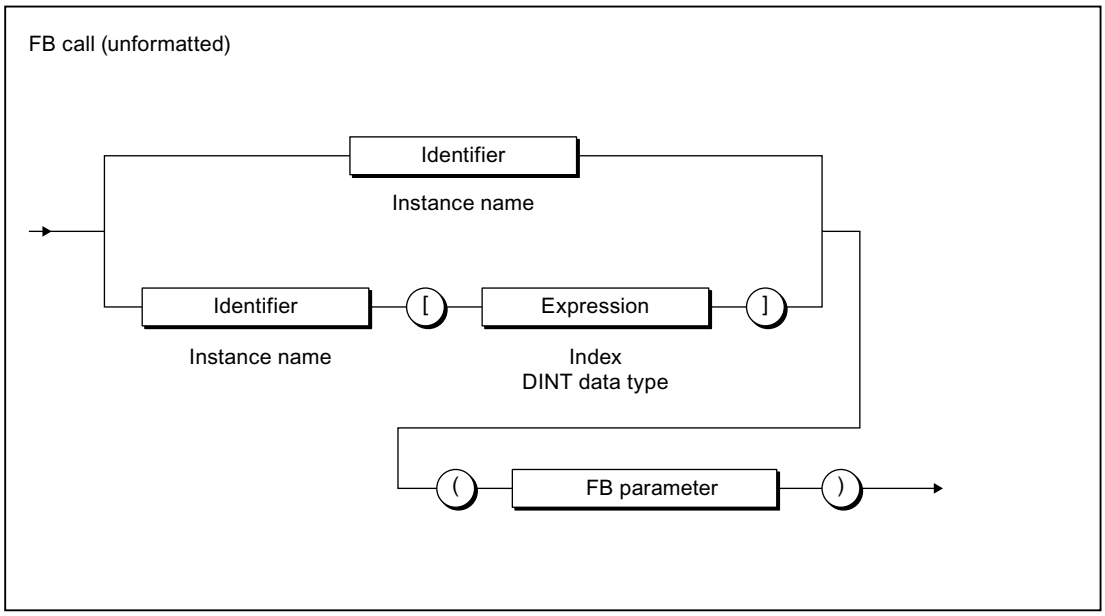


図 A-88　　FB 呼び出し

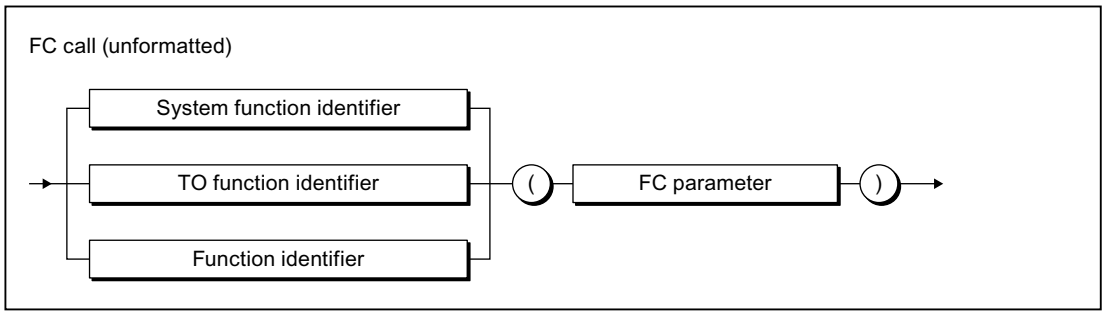


図 A-89　　FC 呼び出し

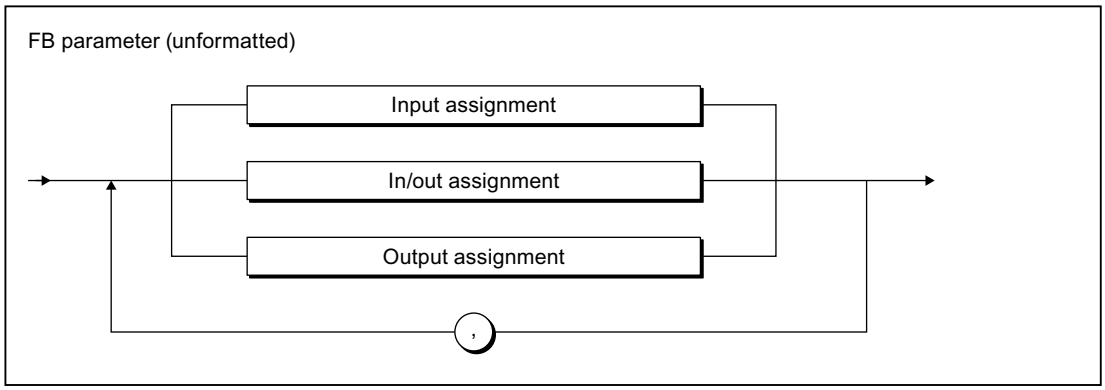


図 A-90　　FB パラメータ

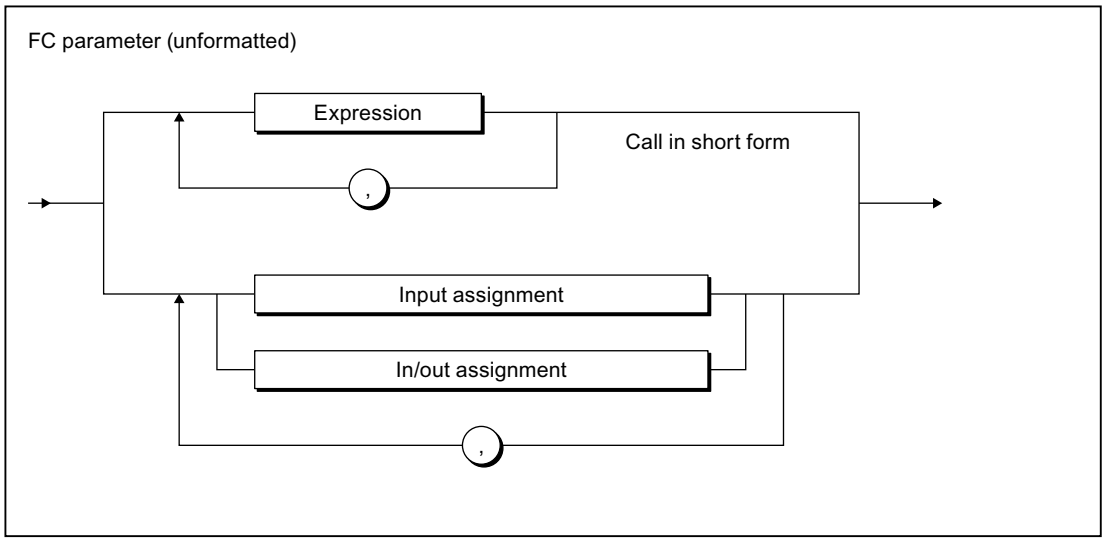


図 A-91　　FC パラメータ

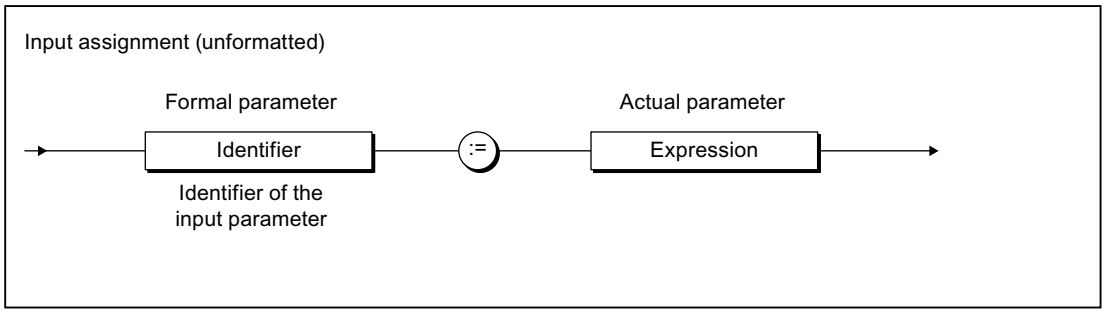


図 A-92　　入力割り付け

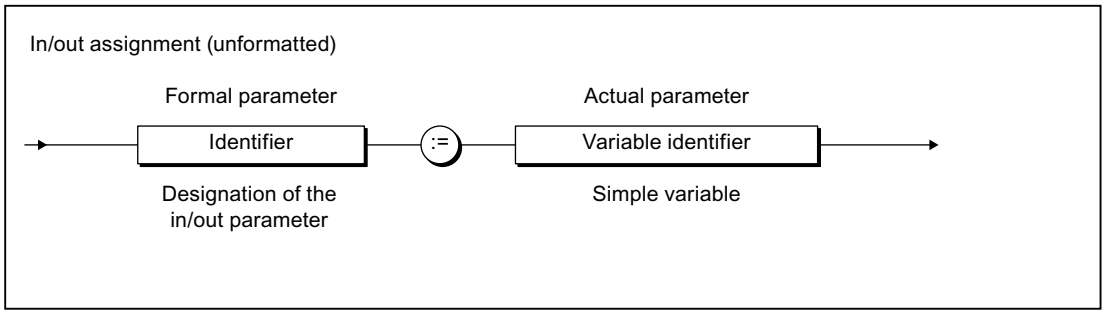


図 A-93　　入力/出力割り付け

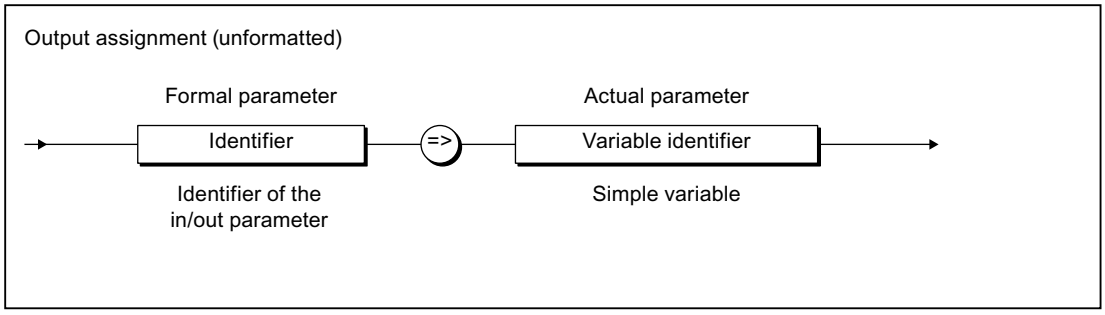


図 A-94　　出力割り付け

A.1.3.13 制御ステートメント

分岐

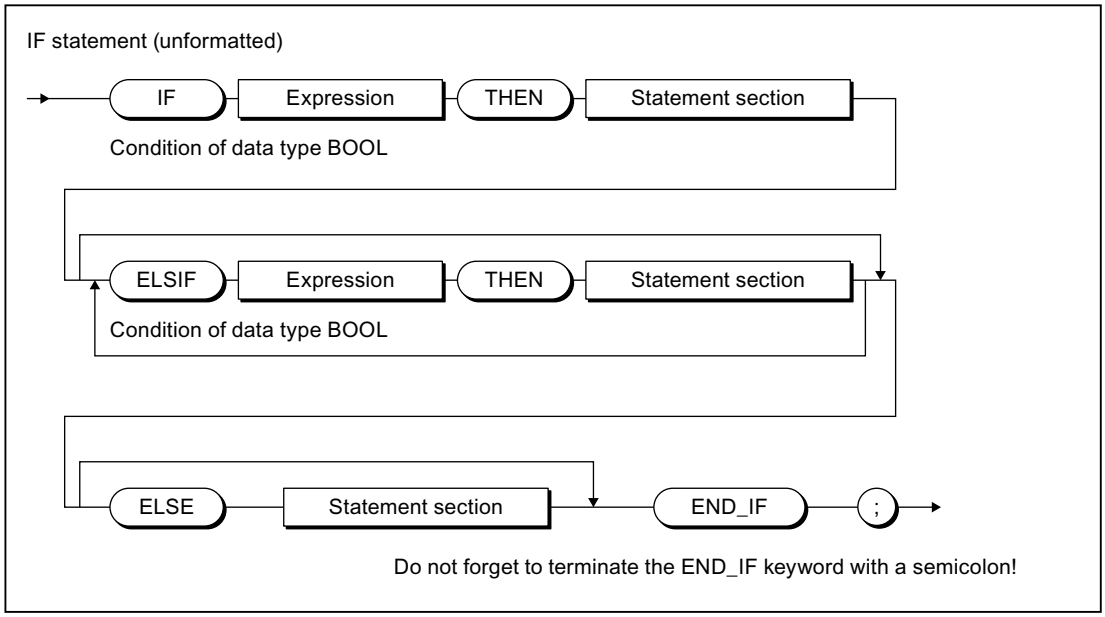


図 A-95 IF ステートメント

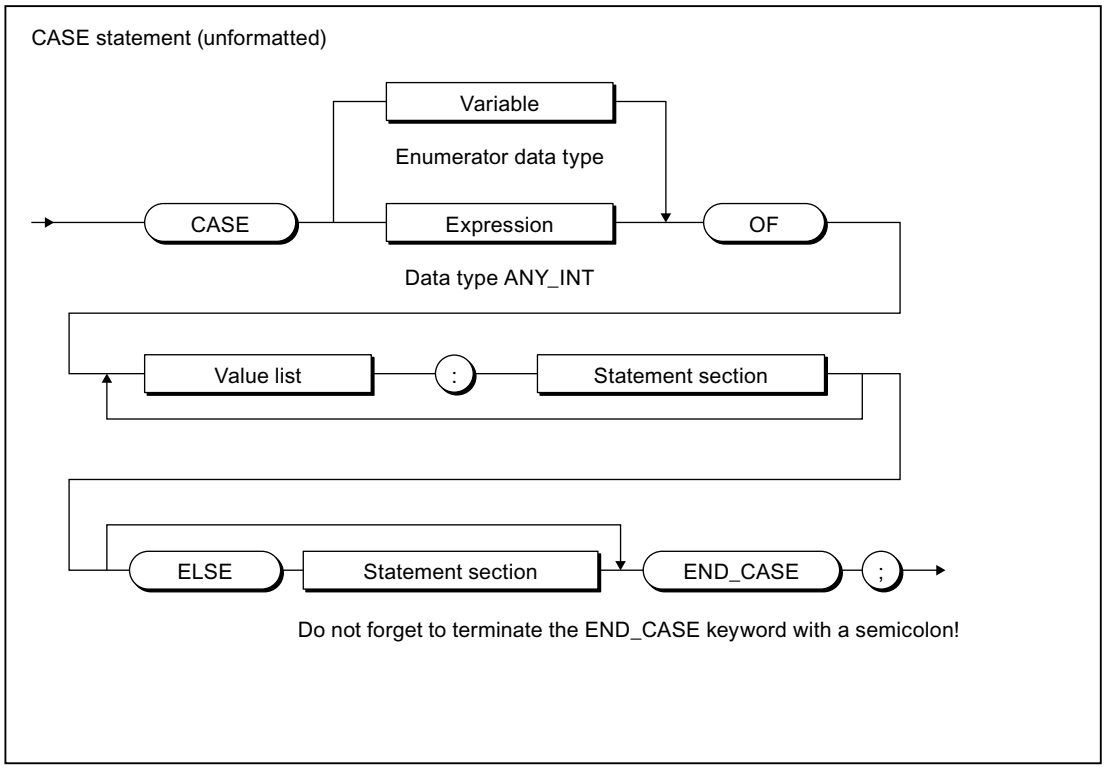


図 A-96 CASE ステートメント

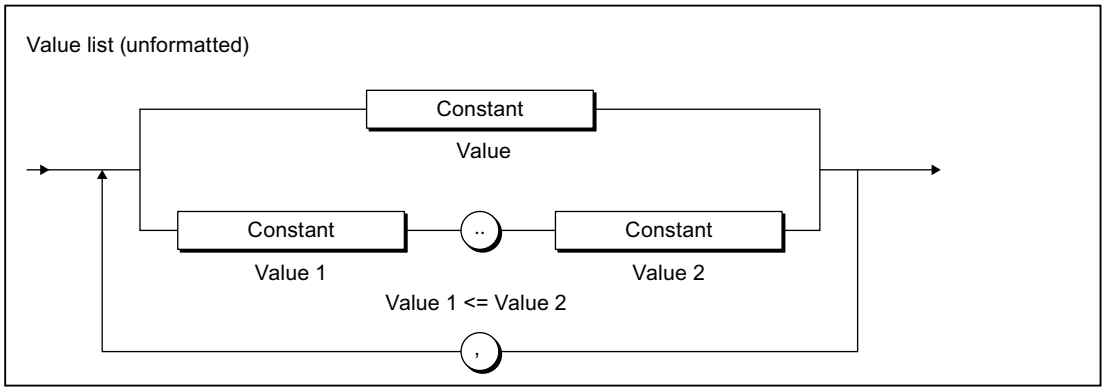

Value list (unformatted)
Constant
Value
Constant
..
Constant
Value 1
Value 2
Value 1 <= Value 2
,

図 A-97　　値リスト

反復ステートメントおよびジャンプステートメント

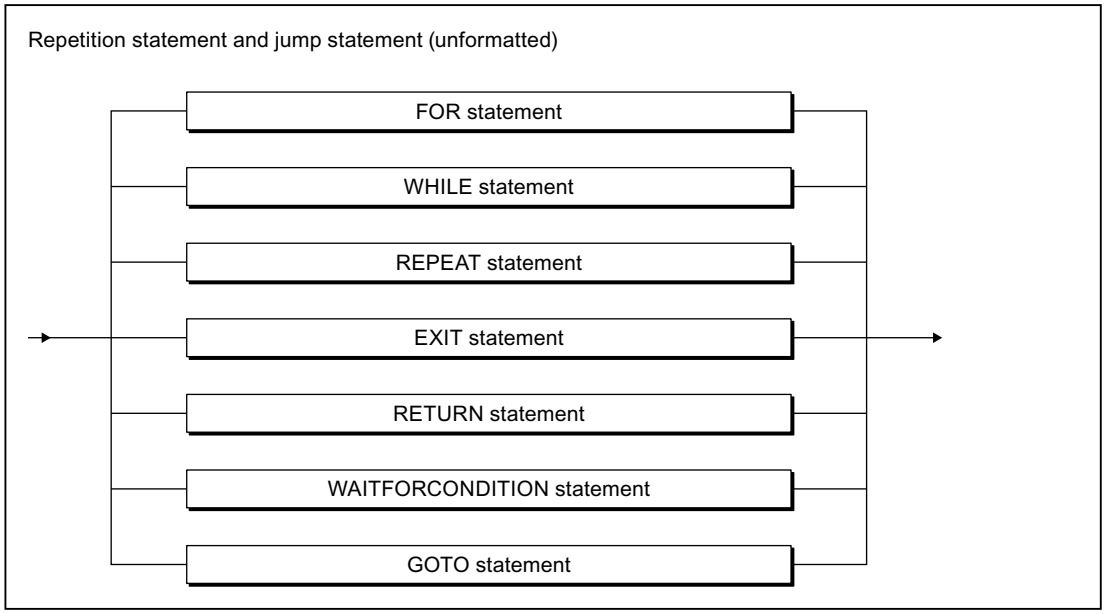


図 A-98　　st_wie1_002

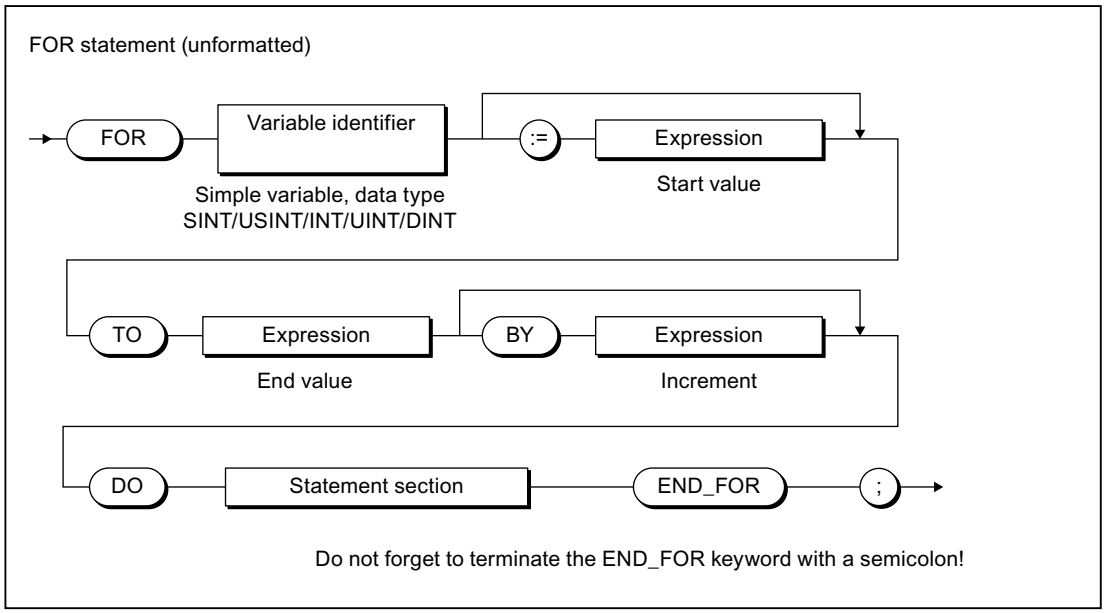


図 A-99　　FOR ステートメント

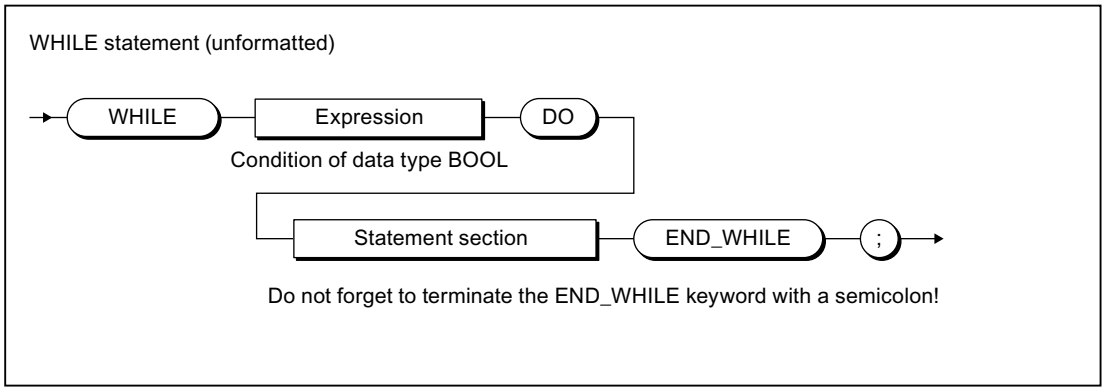


図 A-100　WHILE ステートメント

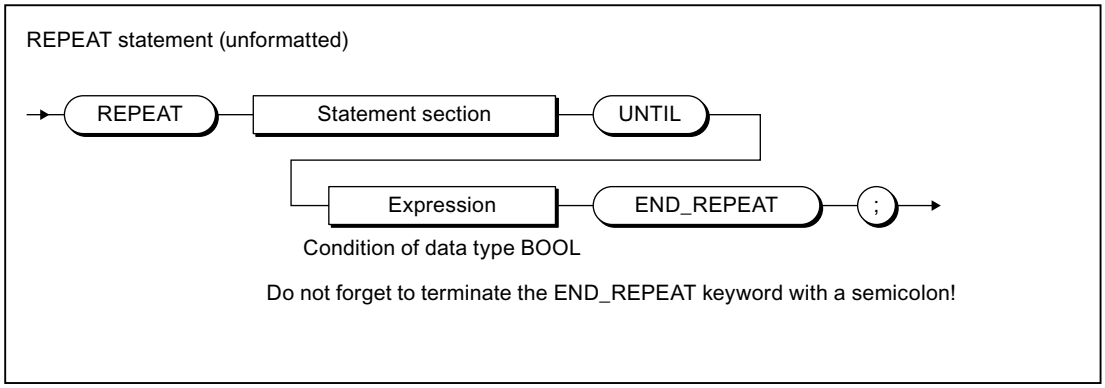


図 A-101　REPEAT ステートメント

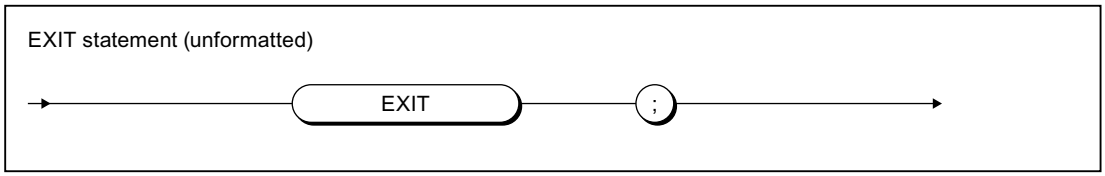


図 A-102　st_exi1_065

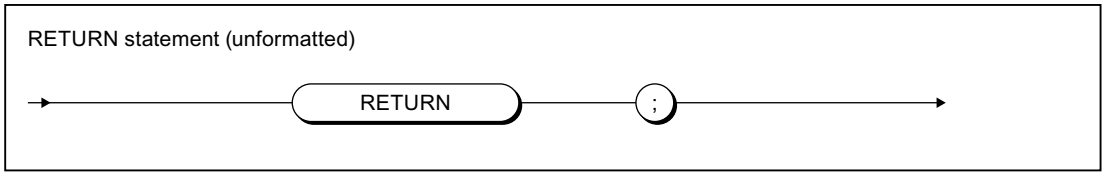


図 A-103　st_ret1_066

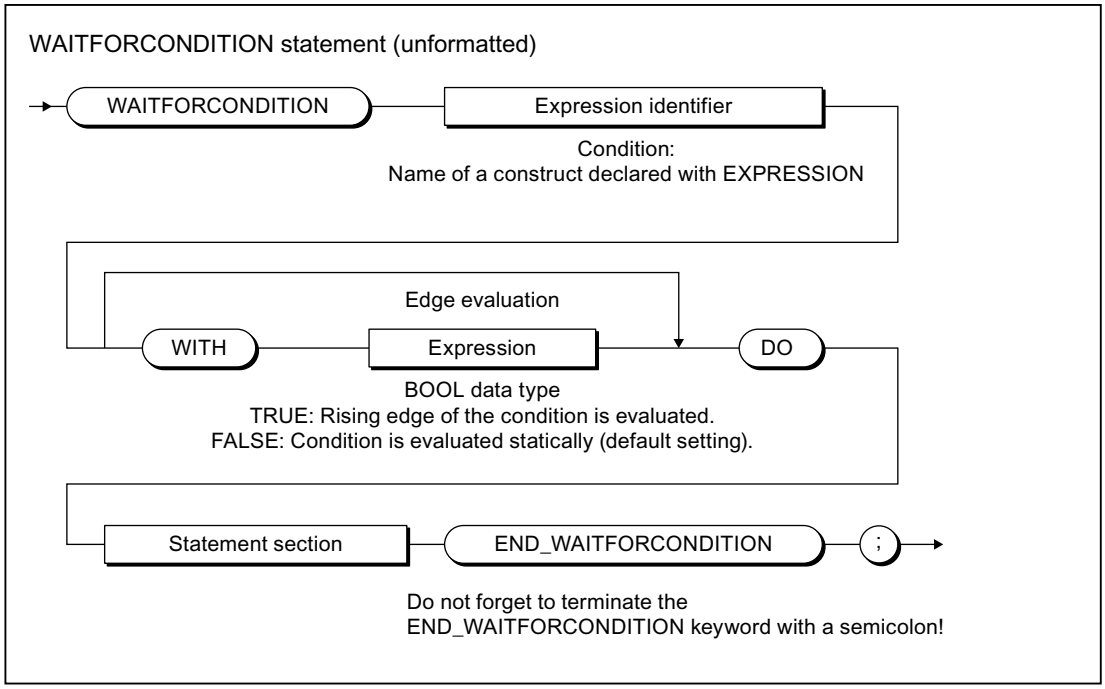


図 A-104　　WAITFORCONDITION ステートメント

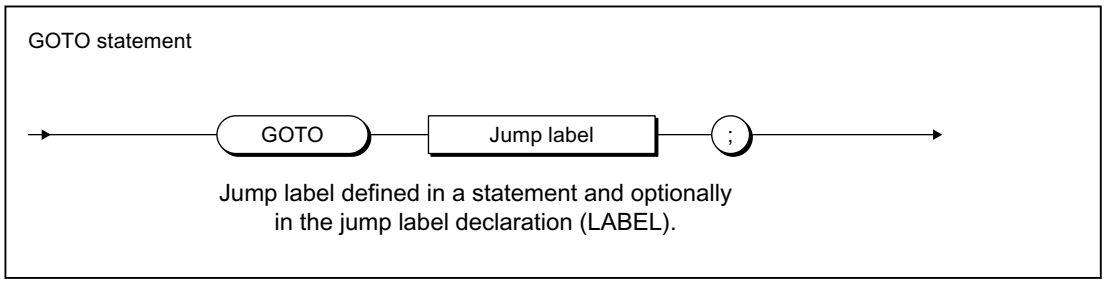


図 A-105　　GOTO ステートメント

A.2 コンパイラエラーメッセージと修正方法

このセクションでは、コンパイラエラーメッセージとその修正の概要を説明します。

A.2.1 ファイルアクセスエラー(1000 – 1003、1100)

表 A-9 ファイルアクセスエラー

エラー	説明
1000	ファイルアクセスで読み取り/書き込みエラーが発生しました。
1001	テキスト形式のエラーメッセージを含むファイルをロードできません。エラーメッセージテキストを出力できません。エラー番号を使用してオンラインヘルプを参照してください!
1002	作成されたコードを格納できませんでした。いくつかのウィンドウを閉じ、再コンパイルしてください!
1003	ファイルを開くときに読み取り/書き込みエラーが発生しました。アプリケーションを閉じ、再試行してください!
1100	プリプロセッサ定義を述べるためのオプションに、定義済みトークンとして無効な識別子が含まれています。 呼び出しオプションの正しい構文: -D identifier[=[text]] 例: • -D myident // myident の定義; これは#ifdef を使用して問い合わせできます。 • -D myident= // myident は空の文字列として定義されます。 • -D "myident=This is a text" // myident は文字列'This is a text'として定義されます。引用符は、置換テキストに空白が含まれる場合のみ使用する必要があります。

A.2.2 スキャナエラー(2001 – 2002)

表 A-10 スキャナエラー(2001 – 2002)

エラー	説明
2001	指定した文字が不正です。
2002	指定した識別子に不正な文字または文字の組み合わせが含まれています。IEC 61131 に従って、識別子をアルファベットまたは下線で開始する必要があります。その後に任意の数のアルファベット、数字、または下線を続けることができますが、複数の下線を続けて使用することはできません。

A.2.3 POU における宣言エラー (3002 – 3027)

表 A-11 POU における宣言エラー (3002 – 3027)

エラー	説明
3002	ロードユニットのコードセクションを識別するためのキーワード"IMPLEMENTATION"が期待されます。
3003	このコンテキストでは指定した宣言ブロックは使用できません。
3004	変数宣言ブロック VAR、VAR_INPUT、VAR_OUTPUT、VAR_IN_OUT、VAR CONSTANT、VAR_GLOBAL、VAR_GLOBAL CONSTANT、および VAR_GLOBAL RETAIN は、各 POU のインターフェースおよび実装セクションで 1 回だけ使用できます。
3005	TASK ステートメント: 指定したタスクのソースファイルには既にタスクリンクが作成されています。さらにタスクリンクを作成することはできません。
3006	タスクに間違ったスタックサイズが指定されました。正の整数だけを使用できます。
3007	指定した識別子はタスク識別子である必要があります。タスク設定を参照してください。
3008	指定した識別子はプログラム識別子である必要があります。ステートメント PROGRAM xx ... END_PROGRAM で宣言を行います。
3009	EXPRESSION キーワードの後に識別子を続ける必要があります。ステートメント EXPRESSION xx ... END_EXPRESSION で宣言を行います。
3010	指定した識別子は EXPRESSION 識別子ではありません。ステートメント EXPRESSION xx ... END_EXPRESSION を使用して宣言を行ったかどうかチェックしてください。
3011	ユニットでは TASK ステートメントは使用できません。ワークベンチのタスク設定を使用してください。
3012	指定した識別子は既に別の位置で宣言されています。この識別子をファンクション識別子として再度使用することはできません。
3013	指定した識別子は既に別の位置で宣言されています。この識別子をファンクションブロック識別子として再度使用することはできません。
3014	UNIT ステートメントが期待されます。以下の形式を使用できます。 • UNIT myunit; • UNIT myunit : dvtype; UNIT ステートメントは ASCII ファイルレベルでコンパイルする場合のみ必要となります。ワークベンチからコンパイラを呼び出すときはこれはオプションです。
3015	ソースファイルが END_IMPLEMENTATION で終わっていません。ソースファイルの構造を確認してください!
3016	キーワード END_IMPLEMENTATION の後にさらにステートメントを指定することはできません。
3017	タスクの宣言が END_TASK で終わっていません。ソースファイルの構造を確認してください!
3018	POU の宣言が END_FUNCTION、END_FUNCTION_BLOCK、または END_PROGRAM で終わっていません。ソースファイルの構造を確認してください!
3019	キーワード FUNCTION、FUNCTION_BLOCK、または PROGRAM で始まる POU が期待されます。
3020	タスクリンクステートメントが期待されます。設定: TASK tname ... END_TASK;
3022	キーワード INTERFACE が期待されます。ソースファイルの構造を確認してください。
3023	キーワード INTERFACE または IMPLEMENTATION が期待されます。ソースファイルの構造を確認してください。
3024	TASK ステートメントの構文エラーです。正しい構造: TASK tname ... END_TASK;
3025	指定した識別子は既に別の位置で宣言されています。この識別子をプログラム識別子として再度使用することはできません。
3026	WAITFORCONDITION ステートメントを再帰的に使用することはできません。WAITFORCONDITION ステートメント内で WAITFORCONDITION ステートメントを 2 回使用しようと試みられました。これはできません。

A.2.4 タイプ宣言における宣言エラー (4001 – 4051)

表 A-12 タイプ宣言における宣言エラー (4001 – 4051)

エラー	説明
4001	指定した識別子は上書きできない標準ファンクション識別子です。別の識別子を選択してください。
4002	指定した識別子は既に使用されています。タイプ識別子として使用することはできません。別の識別子を選択してください。
4003	指定した識別子は既に使用されています。定数識別子として使用することはできません。別の識別子を選択してください。
4004	指定した初期化値は間違ったフォーマットです。データタイプ宣言と一致する初期化値を選択してください。
4005	タイプ宣言の構文エラーです。
4006	構造体宣言における構造体要素指定の構文エラーです。
4007	ARRAY データタイプの宣言の構文エラーです。
4008	識別子リスト指定の構文エラーです。識別子はコンマで区切る必要があります。
4009	指定した定数識別子に別の値が割り付けられています。列挙データタイプを宣言するとこれが起こります。異なる列挙データタイプの同じ列挙要素をタイプ宣言の同じ位置に配置する必要があります。
4010	指定したタイプ識別子を使用する POU はエクスポートされていますが、指定したタイプ識別子がソースファイルからエクスポートされていません。別のデータタイプを使用するか、実装セクションでデータタイプを宣言してください。
4011	定数宣言では初期化値を指定する必要があります。例: x : DINT := 5;
4012	指定したデータタイプは POU の外部で宣言する必要があります。VAR_INPUT、VAR_OUTPUT、および VAR_IN_OUT の場合、タイプ識別子を POU でローカルに宣言してはいけません。タイプ識別子は、パラメータ転送目的のため、POU の外部でも認識される必要があります。
4013	指定した値が列挙データタイプで複数回使用されています。ただし、列挙データタイプの値は異なっている必要があります。
4050	データタイプまたは変数の宣言により、指定した最大許容データサイズより大きいデータタイプが作成されます。
4051	変数宣言では、指定した最大許容メモリサイズより大きいメモリ領域が必要です。

A.2.5 変数宣言における宣言エラー (5001 – 5014、5100 – 5111、5500 – 5507)

表 A-13 変数宣言における宣言エラー (5001 – 5014、5100 – 5111、5500 – 5507)

エラー	説明
5001	指定した定数値により値範囲の超過が引き起こされ、指定した定数値を要求したタイプに変換することはできません。
5002	指定した識別子は既に使用されています。変数識別子として使用することはできません。別の識別子を選択してください。
5003	変数宣言の構文エラーです。
5004	データタイプの指定が期待されます(単純データタイプまたは派生データタイプ)。
5005	指定した定数値のデータタイプが間違っているか、指定した定数値により値範囲の超過が引き起こされます。
5006	配列初期化のための初期化値の数をチェックしてください。
5007	時刻と日付のリテラルの指定における構文エラーです。
5008	指定した位置にファンクションブロックインスタンスを作成することはできません。たとえば、ファンクションに FB インスタンスを作成することはできません。また、ファンクションブロックの出力パラメータ (VAR_OUTPUT)が FB インスタンスであることはできません。
5009	宣言で指定したデータタイプを絶対アドレスを持つ変数に割り当てることはできません。一致するビット幅を持つ整数またはビットデータタイプを使用する必要があります。
5010	変数にメモリアドレスを割り付けようと試みられました。これは指定した位置では行えません。この割り付けは、ユニットの VAR_GLOBAL 宣言内、または PROGRAM の VAR 宣言内でのみ使用してください。
5012	指定した変数に初期化値を事前に割り付けることはできません。
5014	データ構造の間違った初期化です。コンポーネントの初期化値が複数回指定されました。
5100	指定した変数に初期化値を事前に割り付けることはできません。
5110	特殊文字は、次の方法で$...を使用して指定することができます: $$、$'、$L、$N、$P、$R、$T。さらに、$xx を使用して文字の数値を指定することができます。ここで xx は、文字コードの 2 桁の 16 進指定を表します。
5111	特殊文字は$...を介してのみ指定できます。これは、$L、$N、$P、$R、$T に影響します。
5500	指定したジャンプラベル識別子は既に定義されています。別の名前を選択してください。
5501	指定したジャンプラベル識別子は定義されていません。LABEL 宣言にこの識別子を組み込んでください。
5502	ジャンプラベル識別子が複数回割り付けられています。しかし、各ジャンプラベルをラベルとして使用できるのは 1 回だけです。
5503	ジャンプ先としてジャンプラベルが指定されていますが、関連するラベルが存在しません。
5504	下位の制御構造(たとえば、WHILE ループ)ではジャンプはできません。指定したジャンプラベルをこの位置で使用することはできません。
5505	下位の制御構造(たとえば、WHILE ループ)ではジャンプはできません。指定したジャンプ先に到達することはできません。
5506	WAITFORCONDITION ブロックではジャンプはできません。指定したジャンプラベルをこの位置で使用することはできません。
5507	WAITFORCONDITION ブロックではジャンプはできません。指定したジャンプ先に到達することはできません。

A.2.6 式のエラー (6001 – 6140)

表 A-14 式のエラー (6001 – 6140)

エラー	説明
6001	構文エラー: a := b*c;などの、セミコロンで終了するステートメントが期待されます。
6002	構文エラー: x < y などの式が期待されます。
6003	指定した識別子は変数識別子ではありません。変数識別子を指定する必要があります。指示された識別子がカバーされているかどうかチェックしてください。 V4.0 以前では、警告 16021 にかかわらず、同名のプログラムまたはファンクションブロック内でグローバルデバイス識別子へのアクセスが可能でした。
6004	配列アクセスのためのインデックスは DINT データタイプでなければなりません。適切なタイプ変換を実行するか、別の式を使用してください。
6005	式にタイプの不一致がありす。一方のオペランドを計算のタイプに変換できないか、結果の割り付けによりタイプの不一致が起こっています。
6006	指定した変数にアクセスすることはできません。したがって、式で変数を使用することはできません。考えられる原因: • 変数を読み取れない。 • ファンクションまたはファンクションブロックのローカル変数に外部からアクセスしようとしている。
6007	指定された変数を書き込めません。値割り付けはできません。
6008	指定したファンクションは戻り値を提供しません。したがって、式で適用することはできません(VOID の戻り値で宣言したファンクション)。
6009	指定した識別子はファンクションまたはファンクションブロックインスタンスを参照しません。したがって、この識別子をファンクション識別子として使用することはできません。
6010	指定した識別子はファンクションまたはファンクションブロックの宣言に入力パラメータ(VAR_INPUT)または入/出力パラメータ(VAR_IN_OUT)として組み込まれていません。この識別子をファンクション呼び出しで使用することはできません。
6011	呼び出し内のファンクション割り付けの数が宣言と異なるか、必要な呼び出しパラメータが呼び出し内で欠落しています。
6012	この位置では構文的に RETURN は使用できません。RETURN はファンクションでのみ使用できます。
6013	この位置では構文的に EXIT は使用できません。EXIT は FOR、WHILE、および REPEAT 内でのみ使用できます。
6014	指定したインデックス値は配列制限の範囲外です。配列宣言と一致するインデックス値だけを使用できます。
6015	指定されたタスク制御コマンドをタスクに適用できません。これはこのタイプのタスクには許可されていません。
6016	指定したタスクは実行システムで無効にされています。タスクを使用するには有効にする必要があります。
6017	タスク内でプログラムを指定するときの構文エラーです。プログラムは、名前でリストし、コンマで区切る必要があります。
6018	指定した識別子は PROGRAM を参照しません。したがって、この識別子をプログラム識別子として使用することはできません。
6019	タスクにプログラムが複数割り付けられています。割り付けは 1 回のみ可能です。
6020	直接表示した変数を指定するときの構文エラーです。入力の構文は%Ix.y、出力の構文は%Qx.y である必要があります。
6021	直接表示した変数の指定したバイトオフセットは、許容されるアドレス空間の範囲外です。
6022	直接表示した変数の指定したバイトオフセットは、許容されるアドレス空間の範囲外です。0～7 の値を使用できます。
6023	ファンクションの戻り値が割り付けられませんでした。ただし、割り付けは必須です。
6024	指定した識別子を持つ変数がタスク開始情報に組み込まれていません。

エラー	説明
6025	CASE ステートメントの条件変数および条件値は SINT、INT、DINT、USINT、UINT、または UDINT データタイプでなければなりません。条件値を条件変数のデータタイプに暗黙的に変換できる必要があります。
6026	指定したメッセージ識別子がメッセージ設定に含まれていません。メッセージ設定に切り替え、識別子を追加してください。
6027	システム変数はテクノロジーオブジェクトの参照を使用して直接的にのみアクセスできます。構造体または配列を使用してアクセスすることはできません。TO タイプのローカル変数を作成し、この変数に TO の参照を割り付けてください。そうすると、このローカル TO 変数を使用して必要なシステム変数にアクセスすることができます。
6028	指定した演算の式にタイプの不一致があります。一方のオペランドを計算のタイプに変換できないか、結果の割り付けによりタイプの不一致が起こっています。式の指定したデータタイプが期待されます。
6029	指定したファンクションパラメータはデフォルト値を持っていません。したがって、ファンクションを呼び出すときに必ず値を指定する必要があります。
6030	入/出力パラメータ(VAR_IN_OUT)に式を転送しようと試みられました。これはできません。ユーザ変数を入/出力パラメータとして指定する必要があります。
6031	入/出力パラメータ(VAR_IN_OUT)にシステム変数(TO、I/O 直接アクセス)を転送しようと試みられました。これはできません。ユーザ変数を入/出力パラメータとして指定する必要があります。
6032	入/出力パラメータ(VAR_IN_OUT)にプロセスイメージの変数を転送しようと試みられました。これはできません。ユーザ変数を入/出力パラメータとして指定する必要があります。
6033	一致しないデータタイプを持つ変数を入/出力パラメータ(VAR_IN_OUT)に転送しようと試みられました。しかし、暗黙的なタイプ変換はできません。正しいデータタイプを持つユーザ変数を入/出力パラメータとして指定する必要があります。
6034	入/出力パラメータ(VAR_IN_OUT)に読み取り専用変数を転送しようと試みられました。これはできません。入/出力パラメータは読み取り/書き込みである必要があります。
6035	入/出力パラメータ(VAR_IN_OUT)に定数を転送しようと試みられました。これはできません。入/出力パラメータはユーザ変数である必要があります。
6036	定数に演算が適用されています。定数の値はファンクションの値範囲の範囲外です。たとえば次のことが行われています。 • 負の数に対する SQRT の適用 • 0 以下の数での対数関数の使用 • [0..1]の間隔の範囲外の数での ASIN または ACOS の使用
6037	定数をゼロで除算しようと試みられました。この演算は行えません。
6038	指定したファンクションパラメータが引数リストに複数回出現します。
6039	指定したファンクション/ファンクションブロックは使用できません。ファンクションパラメータの宣言が欠落しています。INTERFACE でプロトタイプだけが指定されました。
6040	セマフォとしては単純な変数だけを使用できます。インデックスの作成はできません。
6041	メッセージファンクションには、指定したデータタイプの補助値が必要です。タイプ変換はできません。
6042	アラームファンクションでは、メッセージ番号を指定することが必要です。指定したメッセージ番号は無効です。
6050	指定した演算/変数の式にタイプの不一致があります。一方のオペランドを計算のタイプに変換できないか、結果の割り付けによりタイプの不一致が起こっています。ソースファイルタイプとターゲットタイプ間の変換はできません。
051	指定した演算の式にタイプの不一致があります。一方のオペランドを他方のオペランドのタイプに変換して計算を実行できないか、これらのオペランドタイプをこの演算に使用できません。
6052	式にタイプの不一致がありす。指定したデータタイプを演算に使用することはできません(マーシャリング関数を参照)。
6053	指定した演算の式にタイプの不一致があります。指定したデータタイプに対してこの演算は行えません。
6060	ファンクション呼び出しで、ファンクション引数の割り付けと設定パラメータの割り付けが混ざっています。一方の形式のファンクション呼び出しを使用してください。例: • f (x, y); または • f (in1 := x, in2 := y);

エラー	説明
6070	設定データへのアクセスは、完全に指定されている変数でのみ可能です。選択したテクノロジーオブジェクトの設定データに従って名前を追加してください。
6080	指定した変数は直接アクセスできる入力または出力変数ではありません。このような変数は、各デバイスの I/O コンテナで宣言する必要があり、PI*または PQ*という構文を持っている必要があります。
6100	指定したコンストラクトはデバイスタイプが設定されている場合にのみコンパイルできます。ユニットステートメントにデバイスタイプを追加するか、プログラムコンテナでデバイスタイプを設定してください。
6110	指定したコンストラクトをライブラリで使用することはできません。
6111	指定したコンストラクトをライブラリで使用することはできません。
6112	指定したコンストラクトをライブラリで使用することはできません。
6113	ライブラリではテクノロジーオブジェクトおよびデバイスへのアクセスは許可されていません。
6130	CASE ステートメントで指示したデータタイプの場合、間隔の指定はできません。
6140	ENUM_TO_DINT で定数を指定するには、enum_type#value 形式のデータタイプを指定する必要があります。

構文エラー、式のエラー (7000 – 7014)

エラー	説明
7000	構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 • 括弧が欠けている。
7001	指定した識別子が定数を参照しません。1 つの値または識別子ごとに 1 つの定数を入力してください。
7002	符号付き整数が期待されます。SINT、INT、または DINT データタイプの整数が可能です。
7003	間隔を指定するときは、初期値が終了値より小さいか等しい必要があります。これは、CASE 選択条件における配列の宣言および間隔の指定に当てはまります。
7004	初期化値が期待されます。値は定数である必要があります。定数は次のように割り付けることができます。 • 値ごとに直接 • 先行する定数宣言を介してシンボリックに 定数のみを含む式として
7009	WHILE、REPEAT、および IF の条件としては、BOOL データタイプを提供する式が期待されます。これは、BOOL データタイプの変数として、または比較式を介して指定することができます。BOOL データタイプの戻り値を持つファンクションを指定することもできます。
7010	構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 括弧が欠けている。
7011	構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 括弧が欠けている。
7012	指定された行から始まる、ステートメントの構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 • 括弧が欠けている。
7013	構文エラーが発生しました。不正なコンストラクトが使用されています。
7014	構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 • 括弧が欠けている。

ソースファイルをリンクする際のエラー (8001)

エラー	説明
8001	指定した POU が INTERFACE セクションにエクスポートされましたが、IMPLEMENTATION セクションが存在しません。エクスポートステートメントを削除するか、有効な実装を指定してください。

別の UNIT またはテクノロジーパッケージのインターフェースをロードする間のエラー (10000 – 10036, 10100 – 10101)

エラー	説明
10000	指定したユニットのファイルフォーマットが無効です。ユニットが古いバージョンのコンパイラを使用して作成されたか、互換性のないオプションを使用してコンパイルされた可能性があります。ユニットが関係する場合、まずユニットをコンパイルする必要があります。その後で、現在のコンパイルを繰り返します。パッケージが関係する場合、新しいバージョンをインストールする必要があります。
10001	ユニット名のフォーマットが無効です。ユニット名は最大 8 文字で構成されます。ST の識別子のルールが適用されます。
10002	指定した識別子が、インポートした 2 つの異なるユニットに含まれます。インポートリストから 1 つのユニットを削除するか、インポートしたユニットの識別子間に一意の関係を確立します。これを行うには、インターフェースセクションで、エクスポートされたユニットを修正します。
10003	指定したデータタイプのメモリレイアウトが無効です。ユニットが古いバージョンのコンパイラを使用して作成されたか、互換性のないオプションを使用してコンパイルされた可能性があります。ユニットが関係する場合、まずユニットをコンパイルする必要があります。その後で、現在のコンパイルを繰り返します。パッケージが関係する場合、新しいバージョンをインストールする必要があります。
10004	指定されたユニットのエクスポートされた識別子をロードできませんでした。いくつかのアプリケーションを閉じ、再試行してください。
10005	パッケージのロード時に再帰が検出されました。指定したパッケージは USEPACKAGE を使用して既にロードされており、2 回指定することはできません。
10006	ユニットのロード時に再帰が検出されました。指定したユニットは USES を使用して既にロードされており、2 回指定することはできません。
10007	ユニットで参照できるインポート済みユニットの最大数を超過しました。ロードユニットあたり最大 223 個のインポート済みユニットを参照できます。USES を使用して直接インポートしたユニットと間接的にインポートされたユニットの両方がカウントされます。
10008	ユニットで参照できるインポート済みパッケージの数を超過しました。ロードユニットあたり最大 127 個のインポート済みパッケージを参照できます。
10009	指定したパッケージはユニットでは使用されますが、デバイスでは使用できません。このエラーメッセージは、[implicit package utilization]オプションを使用してコンパイルし、デバイスで指定したパッケージと異なる内容を持つ USEPACKAGE ステートメントをプログラミングした場合に発生します。
10010	指定したパッケージはユニット a では使用されますが、ユニット b では使用されません。このエラーメッセージは、USES を使用して相互に参照する各ユニットで、USEPACKAGE を使用して異なるパッケージが指定された場合に発生します。USEPACKAGE ステートメントを修正してください。
10011	指定したユニットは、直接的あるいは 1 つまたは複数のユニットを介して間接的に自身により使用されます。USES ステートメントを修正してください。
10012	指定したユニットは、異なるコンパイルバージョンの複数のユニットに直接的または間接的にインポートされます。指定したユニットを USES ステートメントで参照するすべてのユニットを再コンパイルしてください。
10013	指定したユニットがまだコンパイルされていないか、最後のコンパイル中にエラーが発生しました。まずこのユニットをコンパイルして、コンパイルを正常に終了させてください。
10014	指定したテクノロジーオブジェクト(TO)のタイプは、USEPACKAGE でのコンパイル中に以前に指定されたパッケージによりサポートされていません。TO タイプを含むパッケージを使用してください。
10015	ユニットで参照できるテクノロジーオブジェクト(TO)の最大数を超過しました。最大 65535 個の TO を参照することができます。

エラー	説明
10016	デバイスタイプパラメータを使用できません。コンパイルするユニットをデバイスに割り付けない場合は、ステートメント UNIT xx : dvtype;を使用してください。
10017	デバイスタイプが一意的に指定されていません。ユニットで、ステートメント UNIT xx : dvtype;により、デバイスに対するユニットの割り付けを介して決定されたデバイスタイプとは異なるデバイスタイプを指定します。
10018	指定されたユニットを検出できませんでした。ワークベンチの PROGRAM コンテナでユニット名を使用できるかどうか、または指定したファイルが現在の作業ディレクトリに含まれてるかどうかをチェックしてください (u7bt00ax のみ - コマンド行)。
10019	指定されたテクノロジーパッケージを検出できませんでした。先行するエラー出力を確認してください。
10020	テクノロジーパッケージのロード時に発生したエラーです。その後のエラー出力を確認してください。
10021	指定したソースファイルでテクノロジーパッケージが使用されていますが、デバイスで選択されていません。USEPACKAGE ステートメントを修正するか、デバイスでテクノロジーパッケージを選択してください。
10022	指定したテクノロジーパッケージが別のバージョンと共に使用されています。デバイスで(さらに、必要に応じてライブラリで)テクノロジーパッケージの選択の設定を修正してください。デバイスでは、テクノロジーパッケージの 1 つのバージョンだけを使用することができます。
10030	デバイスタイプが一意的に指定されていません。ユニットで、ステートメント UNIT xx : dvtype;により、ライブラリコンテナに対するユニットの割り付けを介して決定されたデバイスタイプとは異なるデバイスタイプを指定します。
10031	指定したライブラリは、直接的あるいは 1 つまたは複数のライブラリを介して間接的に自身により使用されます。USELIB ステートメントを修正してください。
10032	指定されたライブラリを検出できませんでした。プロジェクトをチェックしてください。
10033	ライブラリのロード時に再帰が検出されました。指定したライブラリは USELIB を使用して既にロードされており、2 回指定することはできません。
10034	指定したライブラリがまだコンパイルされていないか、最後のコンパイル中にエラーが発生しました。まずこのライブラリをコンパイルして、コンパイルを正常に終了させてください。
10035	指定されたライブラリを検出できませんでした。ワークベンチのプロジェクトでライブラリ名を使用できるかどうか、または指定したファイルが現在の作業ディレクトリに含まれているかどうかをチェックしてください (u7bt00ax のみ - コマンド行)。
10036	指定したパッケージはソースファイルでは使用されますが、ライブラリでは使用できません。ライブラリは、通常、ライブラリコンテナで指定したパッケージバージョンに対してコンパイルされます。ライブラリで指定したパッケージと異なる内容を持つ USEPACKAGE ステートメントがプログラミングされています。正しいパッケージバージョンを選択するか、ソースファイルから USEPACKAGE ステートメントを削除してください。
10100	指定したタイプのテクノロジーオブジェクトが、ソースファイルによって参照された複数のパッケージに含まれています。要件に合うテクノロジーパッケージを選択してください。
10101	指定したテクノロジーオブジェクトは、ロードしたパッケージによってサポートされているテクノロジーオブジェクトのタイプと互換性がありません。パッケージを更新するか、テクノロジーオブジェクトのタイプを変更してください。

実装制限(15001 – 15007、15152、15153)

エラー	説明
15001	指定したコンストラクトは、コンパイラの現在のバージョンでサポートされていません。
15002	現在選択されているデバイスは、指定したファンクションをサポートしていません。このファンクションを使用したい場合は、別のデバイスバージョンを選択してください。これを行うには、ハードウェアカタログで CPU を置換し、必要に応じてファームウェアを更新してください。
15003	指定した識別子はサポートされていないキーワードなので、以後のコンパイラバージョンとの互換性を確保するため、ユーザ固有として使用することはできません。
15004	指定した識別子はサポートされていない標準ファンクションを示すので、以後のコンパイラバージョンとの互換性を確保するため、ユーザ固有識別子として使用することはできません。
15005	指定した識別子はサポートされていない標準ファンクションを示すので、以後のコンパイラバージョンとの互換性を確保するため、ユーザ指定識別子として使用することはできません。
15006	指定したコンストラクトは、MCC を使用して生成したソースファイルでのみ使用できます。ST で使用することはできません。
15007	ソース/ライブラリ/パッケージは、実装セクションでは名前空間を指定せず直接的または間接的に使用します。インターフェースセクションでは、ソース/ライブラリ/パッケージは名前空間と共に使用します。指定したソース/ライブラリ/パッケージのインターフェースセクションで名前空間を指定することにより、この衝突を解決してください。
15152	条件付きコンパイルで隠蔽したソースファイルセクションで USES、USELIB、または USEPACKAGE ステートメントが検出されました。これは不正です。これらのステートメントを含むソースファイルセクションを条件付きでコンパイルすることはできません。
15153	コード生成中、指定した定義は考慮されません。各キーワードを異なる方法で定義することはできません。

警告(16001 – 16601)

エラー	説明
16001	(警告クラス: 0) [Selective Linking]コンパイラオプションに関連してのみ。指定したファンクション、ファンクションブロック、またはプログラムは現在のユニットでエクスポートも呼び出しもされません。コードは生成されません。
16002	(警告クラス: 0) [Selective Linking]コンパイラオプションに関連してのみ。指定したユニットには、エクスポートされた PROGRAM もタスクリンクも含まれていません。ユニットのコードは生成されません。
16003	(警告クラス: 2) 比較演算のオペランドに明示的なタイプ定義が含まれていません。発行された警告メッセージで、比較にリストされたデータタイプを確認できます。使用している定数のデータタイプを<タイプ>#<値>を使用して明示的に指定してください。
16004	(警告クラス: 2) 指定したタイプ変換を行うと、表示幅が縮小されたり、ターゲットデータタイプの精度が不十分であったりするために、変数値が変更されることがあります。
16005	(警告クラス: 2) タイプ変換中、変数値の依存性により符号が変更されることがあります。
16006	(警告クラス: 2) 指定した値は、表示幅が足りないため次の表示可能な値に丸められます。
16007	(警告クラス: 2) タイプ変換中に精度の損失が発生しました。すべての小数位が考慮されるわけではありません。
16008	(警告クラス: 2) 指定した変数の初期化中に精度の損失が発生しました。定数は指定したデータタイプに変換されます。すべての小数位が考慮されるわけではありません。

エラー	説明
16009	(警告クラス: 0) [Selective Linking]コンパイラオプションに関連してのみ。指定したユニットに、エクスポートされた PROGRAM またはタスクリンクが含まれていません。ユニットコードにアクセスすることはできません。関連する POU を呼び出すことはできません。
16010	(警告クラス: 0) 指定したプログラムはユニットにエクスポートされません。したがって、このプログラムを実行レベルの設定で使用することはできません。
16011	(警告クラス: 0) ソースファイルに、エクスポートされたグローバル変数が含まれていません。ターゲットシステムにデータはロードされません。
16012	(警告クラス: 0) 指定したソースファイルの名前が、選択したデバイスの PROGRAMS コンテナから継承されました。UNIT ステートメントのソースファイルの識別子は無視されました。
16013	(警告クラス: 2) マーシャリング機能のため、指定されたデータタイプを移植変換できません。このデータタイプと関連する SIMOTION デバイスだけを使用するか、データタイプの明示的な変換を実行してください。
16014	(警告クラス: 2) 指定した演算で、符号付きと符号なしの間でデータタイプ変換が実行されます。この場合、ビット文字列が採用されるため、結果の数値が指定した値と異なることがあります。
16020	(警告クラス: 1) 宣言により、自身のソースファイルまたはインポートしたソースファイルでグローバルに定義した指定した識別子が隠蔽されます。この識別子がローカルに宣言されている POU からグローバル識別子にアクセスすることはできなくなります。
16021	(警告クラス: 1) 宣言により、デバイスで定義した指定した識別子が隠蔽されます。グローバルデバイス識別子は_device.<名前>を使用してアクセスできます。
6022	(警告クラス: 1) 宣言により、プロジェクト(たとえば、テクノロジーオブジェクトやデバイス)で定義した指定した識別子が隠蔽されます。グローバルプロジェクト識別子は_project.<名前>を使用してアクセスできます。
16023	(警告クラス: 1) 宣言により、テクノロジーオブジェクトのデータタイプの指定した識別子が隠蔽されます。データタイプ識別子にアクセスすることはできなくなります。
16024	(警告クラス: 1) 宣言により、デバイス上のテクノロジーオブジェクトへのアクセスが隠蔽されます。この TO は_to.<名前>を使用してアクセスできます。
16025	(警告クラス: 1) 宣言により、同名の IEC 標準ファンクションが隠蔽されます。現在のコンテキストでこのファンクションにアクセスすることはできなくなります。
16030	(警告クラス: 1) CASE ステートメントでラベルが複数回指定されました。最初のラベルだけが評価されます。その他の指定は効果がありません。
16102	(警告クラス: 3) デバッグ情報が生成されなかったため、プログラムステータスファンクションのコードを出力するためのオプションは無視されました。コンパイラオプションを介してデバッグ情報の出力が無効にされました。
16103	(警告クラス: 3) プログラムステータスファンクションのライブラリにコードを出力するためのオプションは無視されます。個々のソースファイルのオプションで定義されたように、プログラムステータスのコードが生成されます。

エラー	説明
16150	(警告クラス: 7) 指定した識別子に新しい定義が作成されました。したがって、前の定義は無効です。この警告は、プリプロセッサの作業の追跡を可能にします。
16151	(警告クラス: 7) #undef を使用して、指定した識別子の定義を削除しようと試みられました。しかし、識別子が定義されていないか、定義は既に削除されています。この警告は、プリプロセッサの作業の追跡を可能にします。
16152	(警告クラス: 7) コード生成中、指定した定義は考慮されません。これは、コンパイルしたソースに対してプリプロセッサが無効にされていることが原因である可能性があります。
16170	(警告クラス: -) コード生成中、USES を使用してインポートしたソースからの定義は考慮されません。
16200	(警告クラス: 4) セマフォを使用するには、グローバル変数で、異なるタスクからグローバル変数にアクセスできるようにする必要があります。セマフォを介してローカルタスク操作を妨害する必要はありません。
16300	(警告クラス: 5) 補助値が、メッセージに設定されたデータタイプに変換できないデータタイプを持っています。
16301	(警告クラス: 5) メッセージの出力中、指定した補助値は評価されません。
16302	(警告クラス: 5) メッセージ設定から補助値のデータタイプを判別することができません。指定したデータタイプが使用されます。
16303	(警告クラス: 5) メッセージ設定で補助値が必要ですが、ファンクションに補助値が指定されていません。対応するデータタイプのデフォルト値が追加されました。
16400	(警告クラス: 6) ライブラリでグローバル変数が宣言されています。ライブラリを複数回使用できない可能性があります。
16420	(警告クラス: 6) ファンクション内に戻り値が割り付けられていません。このようなファンクションを呼び出すと、ランダムな値が返されます。
16421	(警告クラス: 6) コードで割り付けも読み取りもされていない変数が宣言されています。
16450	(警告クラス: –) 保持メモリ範囲でグローバル変数が作成されています。指定した位置ではこのような宣言はできません。
16451	(警告クラス: –) 0 以外の値を含む大きい配列を初期化すると、コントローラのデータ量が大きくなります。これにより、ロード時間が長くなり、メモリ使用量が多くなります。
16452	(警告クラス: –) 指定したプログラムに、初期化する大量のインスタンスデータが含まれています。初期化コードとユーザコードの両方が実行されているため、タスクの起動時にランタイム違反が引き起こされる可能性があります。特に、SynchronousTask(同期制御タスク)の場合は注意が通知されます。
16600	(警告クラス: 6) 指定した変数は初期化リストに含まれていません。デフォルトの初期化値が使用されます。
16601	(警告クラス: 6) 指定した変数は初期化リストに含まれていません。デフォルトの初期化値が使用されます。

情報

エラー	説明
32010	(警告クラス: 6) 指定したジャンプラベル識別子は宣言されていますが、使用されていません。
32020	(警告クラス: –) 指定した変数は、このソースファイル、または指示したデータタイプを持つ別のソースファイルでグローバルに宣言されました。この情報は、コンパイルエラーの原因を探すときに役立ちます。これは、エラーメッセージと一緒に発行されます。
32021	(警告クラス: –) 指定した変数は、デバイスで I/O 変数、グローバルデバイス変数、またはシステム変数として宣言されました。この情報は、コンパイルエラーの原因を探すときに役立ちます。これは、エラーメッセージと一緒に発行されます。
32022	(警告クラス: –) 指定した変数は、プロジェクトでグローバル識別子として宣言されました。この情報は、コンパイルエラーの原因を探すときに役立ちます。これは、エラーメッセージと一緒に発行されます。
32023	(警告クラス: –) 今までのところ、指定した識別子の有効な宣言は検出されていません。この情報は、エラーメッセージと一緒に発行されます。
32024	(警告クラス: 0) 指定した変数は、現在のユニットまたはインポートユニットでグローバル識別子として宣言されました。この情報は、コンパイルエラーの原因を探すときに役立ちます。これは、エラーメッセージと一緒に発行されます。
32030	(警告クラス: 0) 指定した配列初期化が IEC 61131-3 に準拠していません。ポータブルプログラムの場合、配列初期化値を角括弧で囲む必要があります。標準に準拠したフィールド初期化の例: x : ARRAY [0 to 1] OF INT := [1, 2];
32050	(警告クラス: 0) HMI を介して到達することができる最大サイズは 65536 バイトです。指定した変数でこの制限を超過しました。後続のすべての変数にも到達できません。
32300	(警告クラス: 1) CASE ステートメントでラベルが複数回指定されました。最初のラベルだけが評価されます。その他の指定は効果がありません。
32650	(警告クラス: 7) 指定した識別子はその後に出力テキストによって置換されます。この情報を使用すると、プリプロセッサの作業を追跡できます。
32651	(警告クラス: 7) #undef を使用して、指定した識別子の定義が削除されました。この情報を使用すると、プリプロセッサの作業を追跡できます。
32652	(警告クラス: 7) ソースファイルで識別子は指定した置換テキストと一緒に使用されます。コンパイルは置換テキストを使用して行われます。この情報を使用すると、プリプロセッサの作業を追跡できます。
32653	(警告クラス: 7) 指定した識別子はその後に出力テキストによって置換されます。この情報は、USES ステートメントを使用して追加の置換をロードすると表示されます。この情報を使用すると、プリプロセッサの作業を追跡できます。

A.2.7 構文エラー、式のエラー(7000 – 7014)

表 A-15 構文エラー、式のエラー(7000 – 7014)

エラー	説明
7000	構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 • 括弧が欠けている。
7001	指定した識別子が定数を参照しません。1 つの値または識別子ごとに 1 つの定数を入力してください。
7002	符号付き整数が期待されます。SINT、INT、または DINT データタイプの整数が可能です。
7003	間隔を指定するときは、初期値が終了値より小さいか等しい必要があります。これは、CASE 選択条件における配列の宣言および間隔の指定に当てはまります。
7004	初期化値が期待されます。値は定数である必要があります。定数は次のように割り付けることができます。 • 値ごとに直接 • 先行する定数宣言を介してシンボリックに 定数のみを含む式として
7009	WHILE、REPEAT、および IF の条件としては、BOOL データタイプを提供する式が期待されます。これは、BOOL データタイプの変数として、または比較式を介して指定することができます。BOOL データタイプの戻り値を持つファンクションを指定することもできます。
7010	構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 括弧が欠けている。
7011	構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 括弧が欠けている。
7012	指定された行から始まる、ステートメントの構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 • 括弧が欠けている。
7013	構文エラーが発生しました。不正なコンストラクトが使用されています。
7014	構文エラーが発生しました。考えられる原因: • 制御構造が正しく終了していない(たとえば、END_IF が欠けている)。 • ステートメントが;で終了していない。 • 括弧が欠けている。

A.2.8 ソースファイルをリンクする際のエラー(8001)

表 A-16 ソースファイルをリンクする際のエラー(8001)

エラー	説明
8001	指定した POU が INTERFACE セクションにエクスポートされましたが、IMPLEMENTATION セクションが存在しません。エクスポートステートメントを削除するか、有効な実装を指定してください。

A.2.9 別の UNIT またはテクノロジーパッケージのインターフェースをロードする間のエラー (10000 – 10036、10100 – 10101)

表 A-17 別の UNIT またはテクノロジーパッケージのインターフェースをロードする間のエラー (10000 – 10036、10100 – 10101)

エラー	説明
10000	指定したユニットのファイルフォーマットが無効です。ユニットが古いバージョンのコンパイラを使用して作成されたか、互換性のないオプションを使用してコンパイルされた可能性があります。ユニットが関係する場合、まずユニットをコンパイルする必要があります。その後で、現在のコンパイルを繰り返します。パッケージが関係する場合、新しいバージョンをインストールする必要があります。
10001	ユニット名のフォーマットが無効です。ユニット名は最大 8 文字で構成されます。ST の識別子のルールが適用されます。
10002	指定した識別子が、インポートした 2 つの異なるユニットに含まれます。インポートリストから 1 つのユニットを削除するか、インポートしたユニットの識別子間に一意の関係を確立します。これを行うには、インターフェースセクションで、エクスポートされたユニットを修正します。
10003	指定したデータタイプのメモリレイアウトが無効です。ユニットが古いバージョンのコンパイラを使用して作成されたか、互換性のないオプションを使用してコンパイルされた可能性があります。ユニットが関係する場合、まずユニットをコンパイルする必要があります。その後で、現在のコンパイルを繰り返します。パッケージが関係する場合、新しいバージョンをインストールする必要があります。
10004	指定されたユニットのエクスポートされた識別子をロードできませんでした。いくつかのアプリケーションを閉じ、再試行してください。
10005	パッケージのロード時に再帰が検出されました。指定したパッケージは USEPACKAGE を使用して既にロードされており、2 回指定することはできません。
10006	ユニットのロード時に再帰が検出されました。指定したユニットは USES を使用して既にロードされており、2 回指定することはできません。
10007	ユニットで参照できるインポート済みユニットの最大数を超過しました。ロードユニットあたり最大 223 個のインポート済みユニットを参照できます。USES を使用して直接インポートしたユニットと間接的にインポートされたユニットの両方がカウントされます。
10008	ユニットで参照できるインポート済みパッケージの数を超過しました。ロードユニットあたり最大 127 個のインポート済みパッケージを参照できます。
10009	指定したパッケージはユニットでは使用されますが、デバイスでは使用できません。このエラーメッセージは、[implicit package utilization]オプションを使用してコンパイルし、デバイスで指定したパッケージと異なる内容を持つ USEPACKAGE ステートメントをプログラミングした場合に発生します。
10010	指定したパッケージはユニット a では使用されますが、ユニット b では使用されません。このエラーメッセージは、USES を使用して相互に参照する各ユニットで、USEPACKAGE を使用して異なるパッケージが指定された場合に発生します。USEPACKAGE ステートメントを修正してください。
10011	指定したユニットは、直接的あるいは 1 つまたは複数のユニットを介して間接的に自身により使用されます。USES ステートメントを修正してください。
10012	指定したユニットは、異なるコンパイルバージョンの複数のユニットに直接的または間接的にインポートされます。指定したユニットを USES ステートメントで参照するすべてのユニットを再コンパイルしてください。
10013	指定したユニットがまだコンパイルされていないか、最後のコンパイル中にエラーが発生しました。まずこのユニットをコンパイルして、コンパイルを正常に終了させてください。
10014	指定したテクノロジーオブジェクト(TO)のタイプは、USEPACKAGE でのコンパイル中に以前に指定されたパッケージによりサポートされていません。TO タイプを含むパッケージを使用してください。
10015	ユニットで参照できるテクノロジーオブジェクト(TO)の最大数を超過しました。最大 65535 個の TO を参照することができます。
10016	デバイスタイプパラメータを使用できません。コンパイルするユニットをデバイスに割り付けない場合は、ステートメント UNIT xx : dvtype;を使用してください。

エラー	説明
10017	デバイスタイプが一意的に指定されていません。ユニットで、ステートメント UNIT xx : dvtype;により、デバイスに対するユニットの割り付けを介して決定されたデバイスタイプとは異なるデバイスタイプを指定します。
10018	指定されたユニットを検出できませんでした。ワークベンチの PROGRAM コンテナでユニット名を使用できるかどうか、または指定したファイルが現在の作業ディレクトリに含まれてるかどうかをチェックしてください (u7bt00ax のみ - コマンド行)。
10019	指定されたテクノロジーパッケージを検出できませんでした。先行するエラー出力を確認してください。
10020	テクノロジーパッケージのロード時に発生したエラーです。その後のエラー出力を確認してください。
10021	指定したソースファイルでテクノロジーパッケージが使用されていますが、デバイスで選択されていません。USEPACKAGE ステートメントを修正するか、デバイスでテクノロジーパッケージを選択してください。
10022	指定したテクノロジーパッケージが別のバージョンと共に使用されています。デバイスで(さらに、必要に応じてライブラリで)テクノロジーパッケージの選択の設定を修正してください。デバイスでは、テクノロジーパッケージの 1 つのバージョンだけを使用することができます。
10030	デバイスタイプが一意的に指定されていません。ユニットで、ステートメント UNIT xx : dvtype;により、ライブラリコンテナに対するユニットの割り付けを介して決定されたデバイスタイプとは異なるデバイスタイプを指定します。
10031	指定したライブラリは、直接的あるいは 1 つまたは複数のライブラリを介して間接的に自身により使用されます。USELIB ステートメントを修正してください。
10032	指定されたライブラリを検出できませんでした。プロジェクトをチェックしてください。
10033	ライブラリのロード時に再帰が検出されました。指定したライブラリは USELIB を使用して既にロードされており、2 回指定することはできません。
10034	指定したライブラリがまだコンパイルされていないか、最後のコンパイル中にエラーが発生しました。まずこのライブラリをコンパイルして、コンパイルを正常に終了させてください。
10035	指定されたライブラリを検出できませんでした。ワークベンチのプロジェクトでライブラリ名を使用できるかどうか、または指定したファイルが現在の作業ディレクトリに含まれているかどうかをチェックしてください (u7bt00ax のみ - コマンド行)。
10036	指定したパッケージはソースファイルでは使用されますが、ライブラリでは使用できません。ライブラリは、通常、ライブラリコンテナで指定したパッケージバージョンに対してコンパイルされます。ライブラリで指定したパッケージと異なる内容を持つ USEPACKAGE ステートメントがプログラミングされています。正しいパッケージバージョンを選択するか、ソースファイルから USEPACKAGE ステートメントを削除してください。
10100	指定したタイプのテクノロジーオブジェクトが、ソースファイルによって参照された複数のパッケージに含まれています。要件に合うテクノロジーパッケージを選択してください。
10101	指定したテクノロジーオブジェクトは、ロードしたパッケージによってサポートされているテクノロジーオブジェクトのタイプと互換性がありません。パッケージを更新するか、テクノロジーオブジェクトのタイプを変更してください。

A.2.10 実装制限(15001 – 15007、15152、15152)

表 A-18 実装制限(15001 – 15007、15152、15153)

エラー	説明
15001	指定したコンストラクトは、コンパイラの現在のバージョンでサポートされていません。
15002	現在選択されているデバイスは、指定したファンクションをサポートしていません。このファンクションを使用したい場合は、別のデバイスバージョンを選択してください。これを行うには、ハードウェアカタログで CPU を置換し、必要に応じてファームウェアを更新してください。
15003	指定した識別子はサポートされていないキーワードなので、以後のコンパイラバージョンとの互換性を確保するため、ユーザ固有として使用することはできません。
15004	指定した識別子はサポートされていない標準ファンクションを示すので、以後のコンパイラバージョンとの互換性を確保するため、ユーザ固有識別子として使用することはできません。
15005	指定した識別子はサポートされていない標準ファンクションを示すので、以後のコンパイラバージョンとの互換性を確保するため、ユーザ指定識別子として使用することはできません。
15006	指定したコンストラクトは、MCC を使用して生成したソースファイルでのみ使用できます。ST で使用することはできません。
15007	ソース/ライブラリ/パッケージは、実装セクションでは名前空間を指定せず直接的または間接的に使用します。インターフェースセクションでは、ソース/ライブラリ/パッケージは名前空間と共に使用します。指定したソース/ライブラリ/パッケージのインターフェースセクションで名前空間を指定することにより、この衝突を解決してください。
15152	条件付きコンパイルで隠蔽したソースファイルセクションで USES、USELIB、または USEPACKAGE ステートメントが検出されました。これは不正です。これらのステートメントを含むソースファイルセクションを条件付きでコンパイルすることはできません。
15153	コード生成中、指定した定義は考慮されません。各キーワードを異なる方法で定義することはできません。

A.2.11 警告(16001 – 16601)

表 A-19 警告(16001 – 16601)

エラー	説明
16001	(警告クラス: 0) [Selective Linking]コンパイラオプションに関連してのみ。指定したファンクション、ファンクションブロック、またはプログラムは現在のユニットでエクスポートも呼び出しもされません。コードは生成されません。
16002	(警告クラス: 0) [Selective Linking]コンパイラオプションに関連してのみ。指定したユニットには、エクスポートされた PROGRAM もタスクリンクも含まれていません。ユニットのコードは生成されません。
16003	(警告クラス: 2) 比較演算のオペランドに明示的なタイプ定義が含まれていません。発行された警告メッセージで、比較にリストされたデータタイプを確認できます。使用している定数のデータタイプを<タイプ>#<値>を使用して明示的に指定してください。
16004	(警告クラス: 2) 指定したタイプ変換を行うと、表示幅が縮小されたり、ターゲットデータタイプの精度が不十分であったりするために、変数値が変更されることがあります。
16005	(警告クラス: 2) タイプ変換中、変数値の依存性により符号が変更されることがあります。
16006	(警告クラス: 2) 指定した値は、表示幅が足りないため次の表示可能な値に丸められます。
16007	(警告クラス: 2) タイプ変換中に精度の損失が発生しました。すべての小数位が考慮されるわけではありません。
16008	(警告クラス: 2) 指定した変数の初期化中に精度の損失が発生しました。定数は指定したデータタイプに変換されます。すべての小数位が考慮されるわけではありません。
16009	(警告クラス: 0) [Selective Linking]コンパイラオプションに関連してのみ。指定したユニットに、エクスポートされた PROGRAM またはタスクリンクが含まれていません。ユニットコードにアクセスすることはできません。関連する POU を呼び出すことはできません。
16010	(警告クラス: 0) 指定したプログラムはユニットにエクスポートされません。したがって、このプログラムを実行レベルの設定で使用することはできません。
16011	(警告クラス: 0) ソースファイルに、エクスポートされたグローバル変数が含まれていません。ターゲットシステムにデータはロードされません。
16012	(警告クラス: 0) 指定したソースファイルの名前が、選択したデバイスの PROGRAMS コンテナから継承されました。UNIT ステートメントのソースファイルの識別子は無視されました。
16013	(警告クラス: 2) マーシャリング機能のため、指定されたデータタイプを移植変換できません。このデータタイプと関連する SIMOTION デバイスだけを使用するか、データタイプの明示的な変換を実行してください。
16014	(警告クラス: 2) 指定した演算で、符号付きと符号なしの間でデータタイプ変換が実行されます。この場合、ビット文字列が採用されるため、結果の数値が指定した値と異なることがあります。

エラー	説明
16020	(警告クラス: 1) 宣言により、自身のソースファイルまたはインポートしたソースファイルでグローバルに定義した指定した識別子が隠蔽されます。この識別子がローカルに宣言されている POU からグローバル識別子にアクセスすることはできなくなります。
16021	(警告クラス: 1) 宣言により、デバイスで定義した指定した識別子が隠蔽されます。グローバルデバイス識別子は_device.<名前>を使用してアクセスできます。
6022	(警告クラス: 1) 宣言により、プロジェクト(たとえば、テクノロジーオブジェクトやデバイス)で定義した指定した識別子が隠蔽されます。グローバルプロジェクト識別子は_project.<名前>を使用してアクセスできます。
16023	(警告クラス: 1) 宣言により、テクノロジーオブジェクトのデータタイプの指定した識別子が隠蔽されます。データタイプ識別子にアクセスすることはできなくなります。
16024	(警告クラス: 1) 宣言により、デバイス上のテクノロジーオブジェクトへのアクセスが隠蔽されます。この TO は_to.<名前>を使用してアクセスできます。
16025	(警告クラス: 1) 宣言により、同名の IEC 標準ファンクションが隠蔽されます。現在のコンテキストでこのファンクションにアクセスすることはできなくなります。
16030	(警告クラス: 1) CASE ステートメントでラベルが複数回指定されました。最初のラベルだけが評価されます。その他の指定は効果がありません。
16102	(警告クラス: 3) デバッグ情報が生成されなかったため、プログラムステータスファンクションのコードを出力するためのオプションは無視されました。コンパイラオプションを介してデバッグ情報の出力が無効にされました。
16103	(警告クラス: 3) プログラムステータスファンクションのライブラリにコードを出力するためのオプションは無視されます。 個々のソースファイルのオプションで定義されたように、プログラムステータスのコードが生成されます。
16150	(警告クラス: 7) 指定した識別子に新しい定義が作成されました。したがって、前の定義は無効です。この警告は、プリプロセッサの作業の追跡を可能にします。
16151	(警告クラス: 7) #undef を使用して、指定した識別子の定義を削除しようと試みられました。しかし、識別子が定義されていないか、定義は既に削除されています。この警告は、プリプロセッサの作業の追跡を可能にします。
16152	(警告クラス: 7) コード生成中、指定した定義は考慮されません。これは、コンパイルしたソースに対してプリプロセッサが無効にされていることが原因である可能性があります。
16170	(警告クラス: -) コード生成中、USES を使用してインポートしたソースからの定義は考慮されません。
16200	(警告クラス: 4) セマフォを使用するには、グローバル変数で、異なるタスクからグローバル変数にアクセスできるようにする必要があります。セマフォを介してローカルタスク操作を妨害する必要はありません。
16300	(警告クラス: 5) 補助値が、メッセージに設定されたデータタイプに変換できないデータタイプを持っています。
16301	(警告クラス: 5) メッセージの出力中、指定した補助値は評価されません。

エラー	説明
16302	(警告クラス: 5) メッセージ設定から補助値のデータタイプを判別することができません。指定したデータタイプが使用されます。
16303	(警告クラス: 5) メッセージ設定で補助値が必要ですが、ファンクションに補助値が指定されていません。対応するデータタイプのデフォルト値が追加されました。
16400	(警告クラス: 6) ライブラリでグローバル変数が宣言されています。ライブラリを複数回使用できない可能性があります。
16420	(警告クラス: 6) ファンクション内に戻り値が割り付けられていません。このようなファンクションを呼び出すと、ランダムな値が返されます。
16421	(警告クラス: 6) コードで割り付けも読み取りもされていない変数が宣言されています。
16450	(警告クラス: –) 保持メモリ範囲でグローバル変数が作成されています。指定した位置ではこのような宣言はできません。
16451	(警告クラス: –) 0 以外の値を含む大きい配列を初期化すると、コントローラのデータ量が大きくなります。これにより、ロード時間が長くなり、メモリ使用量が多くなります。
16452	(警告クラス: –) 指定したプログラムに、初期化する大量のインスタンスデータが含まれています。初期化コードとユーザコードの両方が実行されているため、タスクの起動時にランタイム違反が引き起こされる可能性があります。特に、SynchronousTask(同期制御タスク)の場合は注意が通知されます。
16600	(警告クラス: 6) 指定した変数は初期化リストに含まれていません。デフォルトの初期化値が使用されます。
16601	(警告クラス: 6) 指定した変数は初期化リストに含まれていません。デフォルトの初期化値が使用されます。

A.2.12 情報

表 A-20 情報

エラー	説明
32010	(警告クラス: 6) 指定したジャンプラベル識別子は宣言されていますが、使用されていません。
32020	(警告クラス: –) 指定した変数は、このソースファイル、または指示したデータタイプを持つ別のソースファイルでグローバルに宣言されました。この情報は、コンパイルエラーの原因を探すときに役立ちます。これは、エラーメッセージと一緒に発行されます。
32021	(警告クラス: –) 指定した変数は、デバイスで I/O 変数、グローバルデバイス変数、またはシステム変数として宣言されました。この情報は、コンパイルエラーの原因を探すときに役立ちます。これは、エラーメッセージと一緒に発行されます。
32022	(警告クラス: –) 指定した変数は、プロジェクトでグローバル識別子として宣言されました。この情報は、コンパイルエラーの原因を探すときに役立ちます。これは、エラーメッセージと一緒に発行されます。
32023	(警告クラス: –) 今までのところ、指定した識別子の有効な宣言は検出されていません。この情報は、エラーメッセージと一緒に発行されます。
32024	(警告クラス: 0) 指定した変数は、現在のユニットまたはインポートユニットでグローバル識別子として宣言されました。この情報は、コンパイルエラーの原因を探すときに役立ちます。これは、エラーメッセージと一緒に発行されます。
32030	(警告クラス: 0) 指定した配列初期化が IEC 61131-3 に準拠していません。ポータブルプログラムの場合、配列初期化値を角括弧で囲む必要があります。標準に準拠したフィールド初期化の例: x : ARRAY [0 to 1] OF INT := [1, 2];
32050	(警告クラス: 0) HMI を介して到達することができる最大サイズは 65536 バイトです。指定した変数でこの制限を超過しました。後続のすべての変数にも到達できません。
32300	(警告クラス: 1) CASE ステートメントでラベルが複数回指定されました。最初のラベルだけが評価されます。その他の指定は効果がありません。
32650	(警告クラス: 7) 指定した識別子はその後に出力テキストによって置換されます。この情報を使用すると、プリプロセッサの作業を追跡できます。
32651	(警告クラス: 7) #undef を使用して、指定した識別子の定義が削除されました。この情報を使用すると、プリプロセッサの作業を追跡できます。
32652	(警告クラス: 7) ソースファイルで識別子は指定した置換テキストと一緒に使用されます。コンパイルは置換テキストを使用して行われます。この情報を使用すると、プリプロセッサの作業を追跡できます。
32653	(警告クラス: 7) 指定した識別子はその後に出力テキストによって置換されます。この情報は、USES ステートメントを使用して追加の置換をロードすると表示されます。この情報を使用すると、プリプロセッサの作業を追跡できます。

A.3 ユニット例のテンプレート

A.3.1 予備情報

この付録では、注釈の付いた包括的なテンプレートを示します。このテンプレートは、オンラインヘルプで呼び出すことができ、 新しい ST ソースファイルのテンプレートとして使用することができます。

```
//-----------------------------------------------------------------------------
//    テンプレートの最後にユーザデータの INITIALIZATION に関する
//    注意事項がある
//-----------------------------------------------------------------------------
INTERFACE
//    INTERFACE および END_INTERFACE/キーワードの間に追加したすべての
//    ステートメントを使用して、他のソース(ユニット)にもあるどのソースコンテンツ
//    (変数、ファンクション、ファンクションブロックなど)を使用できるか、
//    またはエクスポートするかを定義する。

    USEPACKAGE cam;
    // 使用するテクノロジーパッケージがここで認識され、
    // したがってソースで使用できるようにされる。適切なパッケージが
    // 組み込まれている場合に限り、このユニットでテクノロジーオブジェクト(TO)
    // 固有のコマンドを使用できる。
    // USEPACKAGE カムを使用するソースファイルを USES を介して統合すると、
    // USEPACKAGE が"継承"される。そのときは、USEPACKAGE を省略できる。
    // この例で使用しているパッケージは"cam"である。ただし、他の
    // テクノロジーパッケージを使用することもできる(マニュアルを参照)。

    // USELIB testlib;

    // ライブラリファンクションをソースファイルで使用する場合、ソースにも
    // ライブラリファンクションを認識させる必要がある。"testlib"という名前の
    // ライブラリがプロジェクトに存在しない場合、
    // エラーメッセージ
    // "エラー10035、"testlib.lib"ライブラリをロードできませんでした"
    // "エラー10032、"testlib"ライブラリをロードできませんでした"
    // が出力される。
    // ライブラリを使用していない場合、この行は
    // 削除できる...

    // USES ヘッダー;

    // USES は、異なるソース(NAME、ここでは"header") から エクスポートされたコンテンツ
    // をインポートするために使用し、"Template"で使用できるようにされる。
    // "header"という名前のソースがプロジェクトに存在しない場合、
    // エラーメッセージ
    // "エラー10018、"header"ソースをロードできませんでした"
    // が出力される。この場合、"header"の代わりに
    // 既存のソースファイルの NAME を使用する必要がある。
```

A.3.2 インターフェースでのタイプ定義

```
// ******************************************************
// * INTERFACE でのタイプ定義                  *
// ******************************************************

VAR_GLOBAL CONSTANT
    PI : REAL := 3.1415;
    ARRAY_MAX : INT := 3;
END_VAR
// グローバル定数の宣言。ソースファイルでは
// 識別子にその他の値を割り付けることはできない。

// ユーザ定義変数タイプ(UDT)は
// TYPE と END_TYPE の間で定義する。
TYPE
    array1dim : ARRAY [0..ARRAY_MAX] OF INT;
    // INT タイプの 4 つの配列要素を持つ 1 次元配列の、
    // "array1dim"という名前での定義。すべてのソースファイルセグメントで
    // データタイプとして"array1dim"を使用すると、
    // 1 次元配列を INT によって宣言できる。

    array2dim : ARRAY [0..3] OF array1dim;
    // 2 次元配列は 1 次元配列の配列である。
    // ここでは、"array2dim"という名前で、INT タイプの
    // 16 個の要素を持つ 2 次元フィールドが発生する。

    enumTrafficLight : (RED, YELLOW, GREEN);
    // ユーザ定義変数タイプとしての列挙子"enumTrafficLight"
    // の定義。このタイプの変数は、"RED"、"YELLOW"、
    // および"GREEN"の値だけを受け取ることができる。

    structCollection : STRUCT
        toAxisX : posaxis;
        aInStruct1dim : array1dim;
        eTrafficInStruct : enumTrafficLight;
        iCounter : INT;
        bStatus : WORD;
    END_STRUCT;
    // ここではユーザ定義構造体を作成する。基本データタイプ
    // (ここでは INT および WORD)または定義済み
    // ユーザデータタイプ(ここでは"array1dim"および"enumTrafficLight")を
    // 1 つの構造体に結合することができる。さらに、
    // テクノロジーオブジェクトのタイプも使用できる。
    // 例では、構造体に位置決め軸(posAxis)の要素が
    // 含まれている。
    // 定義では、変数は必ずサイズの昇順に
    // 並べる
    // (ARRAY、STRUCT、LREAL、DWORD、INT、BOOL ...)

    arrayOfStruct : ARRAY [0..5] OF structCollection;
    // ネストも可能。"arrayOfStruct"タイプには、
    // "structCollection"タイプの 6 つの要素から成るフィールドが含まれる。
END_TYPE
```

A.3.3 インターフェースでの変数宣言

```
    // ******************************************************
    // * INTERFACE での変数宣言                    *
    // ******************************************************
    VAR_GLOBAL        // UNIT のユーザメモリ内。
                      // HMI サービスを使用しても表示される。
        g_aMyArray : ARRAY [0..11] OF REAL := [3 (2(4), 2(18))];
        // 前回のタイプ宣言を使用しない 1 次元配列の
        // 宣言の例。ここで実行した初期化は
        // 次のように読み込まれる。
        // 2 つの要素がそれぞれ値 4 で初期化され、
        // 2 つの要素がそれぞれ値 18 で初期化される。このパターンが
        // "g_aMyArray"フィールドで 3 回続けて使用される。
        // したがって、フィールド要素は次のように割り付けられる。
        // 4, 4, 18, 18, 4, 4, 18, 18, 4, 4, 18, 18.
        g_aMy2dim : array2dim;
        // 2 次元配列の宣言の例
        g_aMy1dim : array1dim;
        // タイプ宣言を使用した 1 次元配列の
        // 宣言の例。
        g_sMyStruct : structCollection;
        // タイプの変数、または user_struct の構造体
        // を持つ変数。
        g_aMyArrayOfStruct : arrayOfStruct;
        // ここで生成された変数には、TYPE/END_TYPE セクション
        // で宣言した構造体要素からのフィールドが
        g_tMyTime : TIME := T#0d_1h_5m_17s_4ms;
        // ...基本時刻タイプおよび派生データタイプとして含まれる(以下を参照)。
        g_eMyTraffic : enumTrafficLight := RED;
        // ここで"enumTrafficLight"タイプの列挙子を作成し、
        // 値"RED"を割り付ける。
        g_iMyInt : INT := -17;
        // 変数宣言で基本数値データタイプの変数を
        // 宣言することもできる...
    END_VAR
    VAR_GLOBAL RETAIN
    END_VAR
    // アドオンの RETAIN を使用して宣言した変数は使用している
    // ハードウェアプラットフォームの RETAIN データ領域に格納されるため、
    // ネットワーク障害から守られる。
    // ここでは、VAR、VAR CONSTANT、VAR_TEMP、VAR_INPUT、VAR_OUTPUT、
    // および VAR_IN_OUT の宣言はできない。
    // このセクションで宣言し、エクスポートする変数は、USES "テンプレート"を使用して
    // 別のソースファイル(UNIT)に再インポートできる。
    FUNCTION FC_myFirst;
    FUNCTION_BLOCK FB_myFirst;
    PROGRAM myPRG;
    // 実装部分で定義したファンクションブロック(FB)、
    // ファンクション(FC)、およびプログラムは、他のユニットで使用できるように、
    // このインターフェース部分でエクスポートされる。
    // エクスポートされない FB および FC はこのソースファイルでのみ使用できる
    // ("情報の隠蔽"、他のユニットが絶対に必要とするものだけを
    // インターフェースに配置)。
    // エクスポートされていないプログラムを、削除された TASK に
    // 割り付けることはできない..
END_INTERFACE
```

A.3.4 実装

```
    // ******************************************************
    // * IMPLEMENTATION セクション                          *
    // ******************************************************

IMPLEMENTATION
    // ユニットの IMPLEMENTATION セクションでは、実行コードセクションは
    // さまざまなプログラムオーガニゼーションユニット(POU)に格納される。
    // POU はプログラム、FC、または FB であることが可能。

    VAR_GLOBAL CONSTANT
    END_VAR

    TYPE
    END_TYPE
    // タイプ定義は IMPLEMENTATION セクションでも行うことができる。
    // しかし、この定義を別のソースファイルにインポートすることはできない。
      ただし、タイプ定義をソースファイル"Template"の
    // すべての POU の変数に使用することはできる。タイプの定義は
    // 変数の宣言前に実行する必要がある。
    VAR_GLOBAL         // UNIT のユーザメモリ内
        g_boDigInput1 : BOOL;
        // "EXPRESSION"の例のブール変数(以下を参照)。
    END_VAR


    VAR_GLOBAL RETAIN
    END_VAR
    // アドオンの RETAIN を使用して宣言した変数は使用している
    // ハードウェアプラットフォームの RETAIN データ領域に格納されるため、
    // ネットワーク障害から守られる。
    // IMPLEMENTATION セクションでの変数宣言。
    // ここでは、VAR、VAR CONSTANT、VAR_TEMP、VAR_INPUT、VAR_OUTPUT、
    // および VAR_IN_OUT の宣言はできない。

    EXPRESSION xCond
        xCond := g_boDigInput1;
    END_EXPRESSION
    // EXPRESSION の定義。
    // EXPRESSION は、戻り値 TRUE および FALSE だけを認識する、
    // ファンクションの特殊なケースである。これは、
    // WAITFORCONDITON ステートメントと一緒に使用し(myPRG を参照)、
    // プログラムを MotionTask の一部として実行する場合にのみ使用する
    // 必要がある。"dig_input_1" (デジタル入力またはプログラム内の条件
    // で一般的)が値 1 をとる場合、EXPRESSION の戻り値は
    // TRUE である。
```

A.3.5 ファンクション

```
// ******************************************************
// * FUNCTION                                           *
// ******************************************************

// FB または FC の宣言は、ブロックのコードが呼び出し側ポイント
// に既知になるよう、実際の使用(呼び出し)の前にソースファイルに
// 配置する必要がある。

FUNCTION FC_myFirst :INT
    // ここから POU FUNCTION のステートメントセクションが始まる。この場合、
    // ファンクションの戻り値のタイプは整数である。
    // 呼び出しごとに呼び出し側タスクのスタックが初期化される。
      戻り値はスタックに配置され、
    // FUNCTION によって書き込まれる。

    VAR CONSTANT
    END_VAR

    TYPE
    END_TYPE
    // タイプ宣言は POU でも行うことができる。
    // 基本的な違いはタイプ宣言の
    // 有効性である。POU で宣言したタイプは
    // 関連する POU 内の変数にのみ使用できる。

    VAR_INPUT   // 呼び出し側タスクのスタックにおいて、
                // 呼び出し時にスタックに配置される。割り付けはオプション。
    END_VAR

    VAR         // 呼び出し側タスクのスタックにおいて、
                // FUNCTION で使用される。
    END_VAR
    // FC での変数宣言。
    // ここでは、VAR_TEMP、VAR_GLOBAL、VAR_GLOBAL CONSTANT、
    // VAR_GLOBAL RETAIN、VAR_OUTPUT、および VAR_IN_OUT の宣言は
    // できない。

    // FC および FB でデータを受け取るにはユニットグローバル変数
    // を使用するのがランタイムにとって最速である。入力パラメータ
    // VAR_INPUT の使用および戻り値を介した戻りは、値が個々に
    // コピーされるためそれと比べ低速である。

    // コメント: VAR および VAR CONSTANT で宣言した変数は
    // テンポラリである。次回呼び出し時、FB とは対照的に、最新の呼び出し
    // からの内容は使用できなくなる。

    // ****************************************
    // * FC コードまたはステートメントの領域 *
    // ****************************************)
    // コードはユーザメモリにある。
    g_eMyTraffic := YELLOW; // たとえば、信号機を変更する。

    FC_myFirst := 17;
    // この例では、ファンクションは呼び出し側プログラムに
    // 値"17"を返す。

END_FUNCTION
```

A.3.6 ファンクションブロック(Function block)

```
// ****************************************************
// * FUNCTION_BLOCK                                   *
// ****************************************************

// FB または FC の宣言は、ブロックのコードが呼び出し側ポイント
// に既知になるよう、実際の使用(呼び出し)の前にソースファイルに
// 配置する必要がある。

FUNCTION_BLOCK FB_myFirst
    // ここから FUNCTION_BLOCK POU のステートメントセクションが始まる。
    // インスタンスデータは、UNIT または TASK のユーザメモリ内の、
    // インスタンスが形成される場所に依存し(テンプレートの最後のコメント
    // を参照)、STOP->RUN、または TASK の起動により初期化される。

    // 呼び出し中にインスタンスデータへのポインタが転送される。

    VAR CONSTANT
    END_VAR
    // VAR および VAR CONSTANT で宣言した変数は
    // スタティックである。すなわち次回ブロック呼び出し時、その内容は
    // 引き続き使用可能であり、有効である。

    TYPE
    END_TYPE
    // タイプ定義は POU でも行うことができる。
    // 基本的な違いはタイプ定義の
    // 有効性である。POU で定義したタイプは
    // 関連する POU 内の変数にのみ使用できる。

    VAR_INPUT     // UNIT または TASK のユーザメモリにおいて、
                  // 呼び出し時の割り付けはオプションである。
    END_VAR

    VAR_IN_OUT    // UNIT または TASK のユーザメモリにおいて、
                  // 呼び出し時に参照を割り付ける必要がある。
    END_VAR

    VAR_OUTPUT    // UNIT または TASK のユーザメモリ内。
    END_VAR

    VAR           // UNIT または TASK のユーザメモリにおいて、
                  // FB で使用できる。
    END_VAR

    VAR_TEMP      // 呼び出し側タスクのスタックにおいて、
                  // 呼び出しごとに初期化される。
    END_VAR

    // FB での変数宣言。
    // ここでは、VAR_GLOBAL、VAR_GLOBAL CONSTANT、および
    // VAR_GLOBAL RETAIN の宣言はできない。

    // ****************************************
    // * FB コードまたはステートメントの領域 *
    // ****************************************

    g_eMyTraffic := GREEN;        // たとえば、信号機を変更する。
END_FUNCTION_BLOCK
```

A.3.7 プログラム

```
    // ****************************************************
    // * PROGRAM                                          *
    // ****************************************************

    PROGRAM myPRG
        // ここから POU PROGRAM のステートメントセクションが始まる。

        VAR CONSTANT
        END_VAR

        TYPE
        END_TYPE
        // タイプ定義は POU でも行うことができる。
        // 基本的な違いはタイプ定義の
        // 有効性である。POU で定義したタイプは
        // 関連する POU 内の変数にのみ使用できる。
        VAR     // TASK のユーザメモリ内。
            instFBMyFirst : FB_myFirst;
            // FB を呼び出すことができるためには、スタティック変数用の領域
            // (インスタンスの形成)を生成する必要がある。これによって
            // FB の"メモリ"を処理する必要がある。

            retFCMyFirst : INT;
            // ファンクションの戻り値の変数。
        END_VAR

        VAR_TEMP    // タスクのスタックでは、各パスで初期化される。
        END_VAR
        // PROGRAM での変数宣言。
        // ここでは、VAR_GLOBAL、VAR_GLOBAL CONSTANT、
        // VAR_GLOBAL RETAIN、VAR_INPUT、VAR_OUTPUT、および VAR_IN_OUT の宣言は
        // できない。

        // コメント: VAR を介して宣言したローカル変数
        // がテンポラリ変数かどうかは、PROGRAM を使用する
        // タスクコンテキストに依存する。
        //
        // 非周期的タスク(StartupTask(スタートアップタスク)、ShutdownTask(停止タスク)、MotionTask(モーションタスク)、
        // SystemInterruptTask(システム割り込みタスク)、および UserInterruptTask(ユーザ割り込みタスク))では、
        // VAR および VAR_TEMP の前回の内容はもはや使用できない。
        // したがって、変数はテンポラリである。
        //
        // その他の周期的タスク(BackgroundTask(バックグラウンドタスク)、IPOSynchronousTask(IPO 同期制御タスク)、
        // IPOsynchronousTask_2(IPO 同期制御タスク_2)、および TimerInterruptTask(タイマー割り込みタスク))では、
        // VAR セクションで宣言した変数の内容は次の実行で
        // 同じままである。したがって、変数はスタティックである。
        // VAR_TEMP からの変数は常にテンポラリである。

        instFBMyFirst ();
        // 有効なインスタンスを使用した FB 呼び出し。

        retFCMyFirst := FC_myFirst ();
        // FC 呼び出しおよび戻り値の割り付け。

        WAITFORCONDITION xCond WITH TRUE DO
            // 関連する EXPRESSION で定義した条件"xcond"が論理的に
            // 真である場合、ここでプログラミングしたステートメントが
            // 直ちに実行される状態になる。
            ;
        END_WAITFORCONDITION;
        // WAITFORCONDITION は、通常、MotionTask でのみ使用する。
          MotionTask はその場所にとどまり、
        // EXPRESSION で定義した条件が高優先度でチェックされる。

    END_PROGRAM

END_IMPLEMENTATION
//-----------------------------------------------------------------------------
```

A.3.8 初期化に関する注意事項

```
//    ユーザデータの初期化について
//    * ユーザデータ(基本データタイプ、構造体、および配列の変数)
//    * は異なる時期に初期化される。この時期は、
//    * データの位置(すなわち、メモリ領域)に依存する。
//    * タスクのメインメモリ(スタック)間および
//    * TASK のユーザメモリ内で常に区別される。TASK のユーザメモリと
//    * UNIT のユーザメモリがある。

//    タスクのメインメモリ(スタック)内のデータ:
//    ==========================================
//    各タスクはスタックデータ(ファンクション呼び出しのパラメータ、
//    テンポラリ変数)用の予約メモリを持つ。TASK のスタックサイズは、
//    コンパイラによって計算され、実行システムのタスク設定
//    ([Reserve for Download in the RUN])でユーザが影響を及ぼすことができる。
//    * TASK のメインメモリ(スタック)には以下のデータが含まれる:
//      -     FUNCTION の VAR
//      -     FUNCTION_BLOCK および PROGRAM の VAR_TEMP
//      -     FUNCTION の VAR_INPUT および戻り値
//    * これらは呼び出しごとに初期化される(必要な場合、削除/ゼロに設定、および
//    プログラムから)。

//    ユーザメモリ(ヒープ)は、UNIT ごと、および TASK ごとに
//    別々に管理される:
//    ================================================================================
//    * UNIT のユーザメモリには以下のデータが含まれる:
//      -     INTERFACE および IMPLEMENTATION からの VAR_GLOBAL
//    * これらは初期化される(必要な場合、削除/ゼロに設定、および
//      プログラムから初期値を書き込み):
//      -     起動中
//      -     ロード中(すべての非保持型データの初期化を
//            選択した場合)
//
//    * TASK のユーザメモリには以下のデータが含まれる:
//          PROGRAM の VAR
//    * これらは初期化される(必要な場合、削除/ゼロに設定、および
//      プログラムから初期値を書き込み):
//      -     周期的タスクの場合、STOP->RUN 時に 1 回
//      -     非周期的タスクの場合、タスクの起動時
//
//    * FUNCTION_BLOCK のインスタンスデータ(VAR_INPUT、VAR_OUTPUT、
//      VAR_IN_OUT (参照)、VAR)は、UNIT または TASK のユーザメモリ内の、
//      FB のインスタンスが形成される場所に依存する。
//     FB のインスタンス化
//      -     VAR_GLOBAL 内:              インスタンスは UNIT のユーザメモリに置かれる
//      -     PROGRAM の VAR 内:          インスタンスは TASK のユーザメモリに置かれる
//      -     FB の VAR 内:               インスタンスは上位の FB に応じてユーザメモリに
//                                        置かれる
//    * インスタンスデータは上記のように初期化される。
//    どのデータ領域にどの変数タイプが置かれるかは、
//     テンプレートのコメントで入手可能。
//--------------------------------------------------------------------------------
```

索引

#

–, 107
#define, 216
#else, 216
#endif, 216
#ifdef, 216
#ifndef, 216
#undef, 216

*

*, 107
**, 107

/

/, 107

_

_additionObjectType, 85
_alarm, 208
_camTrackType, 85
_controllerObjectType(コントローラオブジェクトタイプ), 85
_device, 204, 208
_direct, 192, 204, 208
_fixedGearType, 85
_formulaObjectType(数式オブジェクトタイプ), 85
_getSafeValue
　用途, 204
_project, 208
_sensorType, 85
_setSafeValue
　用途, 204
_task, 208
_to, 208
_U7_PoeBld_CompilerOption, 219

+

+, 107

<

<, 109
<=, 109
<>, 109

=

=, 109

>

>, 109
>=, 109

A

ANY, 76
ANY_BIT, 76
ANY_DATE, 76
ANY_ELEMENTARY, 76
ANY_INT, 76
ANY_NUM, 76
ANY_REAL, 76
ANYOBJECT, 85

B

BackgroundTask のプロセスイメージ, 192
BOOL, 73
BYTE, 73

C

camType, 85
CASE ステートメント
　説明, 115
CPU メモリアクセス
　プロセスイメージアクセスのための識別子, 256
　変数モデル, 167

D

DATE, 74
DATE_AND_TIME, 74
DINT, 73
DINT#MAX, 75
DINT#MIN, 75
driveAxis, 85
DT, 74
DWORD, 73

E

EXIT ステートメント
　説明, 122
EXPRESSION
　構文, 149
　説明, 155
externalEncoderType, 85

F

FB。「ファンクションブロック」を参照, 17
FB/FC 変数
　変数モデル, 167
　定義, 174
FB のインスタンスの宣言
　構文, 141
FC。「ファンクション」を参照, 17
Floating-point number
　データタイプ, 73
followingAxis, 85
followingObjectType(フォローイングオブジェクトタイプ), 85
FOR ステートメント
　説明, 118

G

GOTO ステートメント, 221

H

HMI_Export, 219

I

I/O アドレス空間へのシンボリックアクセス
　プロセスイメージ, 202
I/O 変数
　BackgroundTask のプロセスイメージ, 199
　作成, 195, 203
　直接アクセス, 192
　プロセスイメージ, 192
I/O 変数
　「プロセスイメージ」も参照, 17
IF ステートメント
　説明, 114
INT, 73
INT#MAX, 75
INT#MIN, 75

L

LABEL 宣言, 221
LREAL, 73
LREAL#MAX, 75
LREAL#MIN, 75

M

MeasuringInputType(測定入力タイプ), 85
MOD, 107

O

outputCamType, 85

P

posAxis, 85

R

REAL, 73
REAL#MAX, 75
REAL#MIN, 75
REPEAT ステートメント
　説明, 121
RETAIN, 171
RETURN ステートメント
　説明, 122
RUN
　変数初期化に対する影響, 184

S

SCOUT ワークベンチ。「ワークベンチ」を参照, 17
SINT, 73
SINT#MAX, 75
SINT#MIN, 75

STOP から RUN
変数初期化に対する影響, 184
STRING, 74
エレメント, 98
ファンクションの編集, 98
割り付け, 97
StructAlarmId, 76
STRUCTALARMID#NIL, 76
StructTaskId, 76
STRUCTTASKID#NIL, 76
ST エディタ。「エディタ」を参照, 17
ST コンパイラ。「コンパイラ」を参照, 17
ST ソースファイル
インポート, 37
エクスポート, 37
テンプレート(例), 326
印刷, 38
ST ソースファイル。「ソースファイル」を参照, 17

T

T#MAX, 75
T#MIN, 75
TIME, 74
TIME#MAX, 75
TIME#MIN, 75
TIME_OF_DAY, 74
TIME_OF_DAY#MAX, 75
TIME_OF_DAY#MIN, 75
TO#NIL, 85
TO。「テクノロジーオブジェクト」を参照, 17
TOD, 74
TOD#MAX, 75
TOD#MIN, 75

U

UDINT, 73
UDINT#MAX, 75
UDINT#MIN, 75
UDT
「ユーザ定義データタイプ」を参照, 77
UINT, 73
UINT#MAX, 75
UINT#MIN, 75
UNIT, 161
USELIB, 152
USEPACKAGE, 152
USES, 152, 153, 161, 162
USINT, 73
USINT#MAX, 75
USINT#MIN, 75

W

WHILE ステートメント
説明, 120
WORD, 73

ア

アクセスタイム
パラメータ, 140

イ

インターフェース
ソースファイルセクション, 152, 161
インポート
ST ソースファイル, 37

ウ

ウォッチテーブル, 230

エ

エクスポート
ST ソースファイル, 37
エディタ
ツールバー, 29
プログラムの例, 46
操作, 47
エラー
FB または FC の呼び出し, 144
配列データタイプ指定, 80
エラーメッセージ
スキャナエラー, 305
ソースファイルをリンクする際のエラー, 318
タイプ宣言における宣言エラー, 307
ファイルアクセスエラー, 305
別の UNIT またはテクノロジーパッケージのインターフェースをロードする間のエラー, 319
変数宣言における宣言エラー, 308
実装制限, 321
宣言エラー, 306
式のエラー, 309
情報, 325
構文エラー、式のエラー, 318
警告, 322

オ

オペランド

構文, 293

カ

関連資料, 4

キ

基準データ, 210
基本データタイプ
概要, 73

グ

グローバルデバイスデータ
データモデル, 167
定義, 178
グローバルユーザデータ
変数モデル, 167
定義, 178
グローバル変数ブロック。「ユニット変数」を参照, 17

ク

クロスリファレンスリスト, 210

ケ

継承
テクノロジーオブジェクトの場合, 86

コ

コード属性, 213
コマンド
ST プログラミング言語の概要, 64
基本システムの概要, 257
コンパイラ, 48
エラーの修正, 29, 48
スキャナエラー, 305
ソースファイルをリンクする際のエラー, 318
タイプ宣言における宣言エラー, 307
ファイルアクセスエラー, 305
別の UNIT またはテクノロジーパッケージのインターフェースをロードする間のエラー, 319
変数宣言における宣言エラー, 308
実装制限, 321
宣言エラー, 306
属性, 219
式のエラー, 309
情報, 325
有効化, 48
構文エラー、式のエラー, 318
警告, 322
起動, 29
コンパイル
ST ソースファイル, 205

サ

サイクリックタスクのプロセスイメージ, 192
サイクリックプログラムの実行
I/O アクセスに対する効果, 192, 199
変数初期化に対する影響, 184
参照, 85

シ

シーケンシャルプログラムの実行
I/O アクセスに対する効果, 192
変数初期化に対する影響, 184

ジ

時間データタイプ
概要, 74

シ

システム関数
継承, 86
システムファンクション
「テクノロジーオブジェクト」も参照, 17
システム変数
継承, 86
システム変数
変数モデル, 167

ジ

ジャンプラベル
構文, 282

シ

初期化
変数初期化の時期, 184
新規作成
I/O 変数, 195, 203

シンボルブラウザ, 227

ス

数値データタイプ, 73
ステータス
　プログラム(テストツール), 233

セ

整数
　データタイプ, 73
セクション
　構文, 274
セパレータ, 253

ソ

ソースファイル
　構造, 69
ソースファイルセクション
　インターフェース, 152, 161
　ステートメントセクション, 158
　データタイプ宣言, 158
　ファンクション, 154
　ファンクションブロック(Function block), 156
　プログラム, 157
　プログラムオーガニゼーションユニット, 154
　ユニットステートメント, 161
　命令, 71
　変数宣言, 159
　実装, 153
　宣言セクション, 157

タ

ターゲット変数, 95
タイプ変換ファンクション, 125
タイプ宣言, 78

ダ

ダウンロード
　変数初期化に対する影響, 184

タ

タスク
　プログラムの割り付け, 50
　変数初期化に対する影響, 184

チ

直接アクセス, 192
　機能, 193

テ

定数
　時間指定, 74

デ

データタイプ
　TYPE, 83
　基本, 73
　継承, 86
　時間, 74
　数値, 73
　テクノロジーオブジェクト, 85
　ビットデータタイプ, 73
　ユーザ定義, 83
　列挙子, 82
　単純タイプの派生, 79
　変換, 125
　明示的な変換, 128
　暗黙的な変換, 126
　構文, 286, 289
データタイプ指定
　STRUCT, 83
　列挙子, 82
　基本, 79
データモデル, 167

テ

テクノロジーオブジェクト
　継承, 86
　データタイプ, 85
テンプレート
　ST ソースファイル, 326

ト

トレースツール, 250

ノ

ノウハウ保護, 35

ハ

ハードウェア
設定, 44

パ

パラメータ
アクセスタイム, 140
ファンクションおよびファンクションブロック, 134
ブロック(構文), 134
宣言, 133
宣言、概要, 89
転送(入/出力パラメータ), 138
転送(入力パラメータ), 138
転送(出力パラメータ), 139
転送(原理), 137
パラメータフィールド
構文, 281

ビ

ビットデータタイプ, 73
ビット定数, 67

フ

ファイル。「ソースファイル」を参照, 17
ファンクション
TO。「テクノロジーオブジェクト」を参照, 17
システムファンクション。「テクノロジーオブジェクト」を参照, 17
ソースファイルセクション, 154
ローカル変数, 137
例, 145
入力パラメータ, 136
呼び出し, 140
呼び出しパス, 236
呼び出し中のエラーソース, 144
定義, 132
構文, 132
構造, 132
ファンクションブロック
FC との違い, 145
ファンクションブロック(Function block)
インスタンス, 141
ソースファイルセクション, 156
ローカル変数, 137
例, 145
入/出力パラメータ, 136
入力パラメータ, 136
出力パラメータ, 136
名前, 141
呼び出し, 141
呼び出し、構文, 142
呼び出しパス, 236
呼び出し中のエラーソース, 144
定義, 133
構文, 133
構造, 133

ブ

ブール値, 67

フ

フォーマット文字, 253

プ

プラグマ
プリプロセッサステートメント, 215
属性, 219
プリプロセッサ
プリプロセッサステートメント, 215
使用, 31, 34
制御, 215
有効化, 31, 34
警告クラス, 34
プリプロセッサステートメント
例, 218

ブ

ブレークポイント, 237
削除, 241
設定, 241
ツールバー, 243
無効化, 248
有効化, 248
呼び出しパス, 243, 245, 249

プ

プログラミング環境, 19
プログラム
エラーの検出, 223
コンパイル, 48
ステータス(テストツール), 233
ソースファイルセクション, 157

ターゲットシステムへの接続, 51
ダウンロード, 52
タスクの割り付け, 50
テスト, 223
作成(例), 46
呼び出しパス, 236
実行, 50, 53
起動, 50, 53
プログラムオーガニゼーションユニット
ソースファイルセクション, 154
構文, 276
プログラム構造, 212
プログラムセクション。「ソースファイルセクション」を参照, 17
プログラムのテスト, 223
プログラムの構造化, 114
プログラムヘンスウ
データモデル, 173
変数モデル, 167
定義, 174
プログラム実行, 232
ツールバー, 233
プロジェクト
表示, 43
プロセスイメージ
「直接アクセス」も参照, 17
機能, 193
原理と用途, 192, 199
シンボリックアクセス, 202

ブ

ブロック, 56

プ

プロトタイプ, 165

ベ

べき乗, 107

ヘ

変数
初期化の時期, 184
プロセスイメージ, 192, 199

メ

メモリの必要条件, 178, 183

ユ

ユーザ定義データタイプ
構文, 78
ユニット
ソースファイルセクション, 161
テンプレート(例), 326
ユニット変数
変数モデル, 167
定義, 170
ユニット定数
定義, 171

ヨ

呼び出しパス
ブレークポイント, 243, 245, 249

ラ

ライブラリ, 205
コンパイル, 205
使用, 207

ル

ルール
セマンティクス, 56
書式なし, 252, 265
書式付き, 251, 265

ロ

ローカルデータスタック, 178, 183
ローカル変数
変数モデル, 167

ワ

ワークベンチ
プログラミング環境, 19
要素, 20

予

予約識別子, 59, 257

例

例、完全
FB と FC, 145
ST ソースファイル(テンプレート), 326
TO データタイプの使用, 86
ユーザ定義データタイプ, 84
出力バイトのビットの回転, 43

保

保持型変数
変数モデル, 167
定義, 173

値

値割り付け
構文, 291
説明, 95

入

入/出力パラメータ
ファンクションブロック(Function block), 136
転送, 138
入力/出力割り付け
構文, 139
入力パラメータ
ファンクション, 136
ファンクションブロック(Function block), 136
ファンクションブロックでのアクセス, 143
転送, 138
入力割り付け
構文, 138

出

出力パラメータ
ファンクションブロック(Function block), 136
ファンクションブロックでのアクセス, 143
転送, 139

分

分岐
構文, 300

列

列挙子, 82
列挙子データタイプ, 82

初

初期化
構文, 284

制

制御ステートメント, 114

単

単一要素変数, 97
単純データタイプ
派生, 79
単純データタイプの派生, 79

印

印刷
ST ソースファイル, 38

反

反復ステートメントおよびジャンプステートメント
構文, 302

名

名前, 57
名前空間, 208

呼

呼び出しパス
プログラムステータス, 236
プログラム実行, 232

命

命令
ソースファイルセクション, 71, 158

基

基本機能, 106
基本要素
    Of ST, 57

変

変数, 88
    ARRAY, 101
    ARRAY, 102
    FB のインスタンスの宣言, 141
    ウォッチテーブル, 230
    スタティック, 174
    テンポラリ, 174
    バッテリバックアップ, 173
    パラメータ宣言, 133
    ファンクションブロック(Function block), 137
    保持型, 173
    列挙子データタイプ, 101
    列挙子データタイプ, 101
    初期化、説明, 90
    同名, 169
    基本, 97
    宣言, 89
    宣言(ソースファイルセクション), 159
    有効性, 167
    有効性範囲の隠蔽, 169
    構造体, 102
    機能, 137
変数ブロック
    構文, 280

多

多要素変数, 101, 102

定

定数
    Bit, 67
    グローバルに有効, 171
    シンボリック名, 94
    フォーマット文字およびセパレータ, 255
    ユニット定数, 171
    リテラル、構文, 266
    定数のデータタイプ, 72
    数字列、構文, 269
    整数, 66
    日付と時刻、構文, 270
    浮動小数点数, 67
定数ブロック
    構文, 279

実

実数。「浮動小数点数」を参照, 17
実装
    ソースファイルセクション, 153

宣

宣言
    パラメータ, 89
    変数, 89
    構文, 283
宣言セクション
    構文, 277

属

属性
    コンパイラオプション, 219

式

式
    定式化のルール, 103, 113
    算術, 106
    論理, 113
    関係式, 109, 113
論理式; ビットシリアル式
    論理; 式 ビットシリアル, 111

指

指数
    説明, 67

数

数字
    数字のデータタイプ, 72
    表記法, 66
    説明, 66

整

整数
    表記法, 66
    説明, 66

整数。「整数」を参照, 17

文

文字セット, 57, 253

明

明示的なデータタイプ変換, 128

時

時刻タイプ
　変換, 125
　機能, 106

暗

暗黙的なデータタイプ変換, 126

有

有効性範囲の隠蔽, 169

構

構文ダイアグラム, 55
構造
　構文, 274
構造体変数, 101, 102

標

標準ファンクション, 106

派

派生データタイプ
　「ユーザ定義データタイプ」を参照, 17

浮

浮動小数点数
　表記法, 67
　説明, 67

演

演算子, 257
　優先度, 113
　構文, 296
　関係演算子, 109

直

直接アクセス
　「I/O 変数」も参照, 17
　変数モデル, 167

端

端子, 57

算

算術演算子, 106

絶

絶対識別子
　概要, 256

継

継承
　インポート/エクスポート中, 163

複

複合データタイプ, 83
　ARRAY; 配列 データタイプ, 80

言

言語記述
　リソース, 55, 251, 253

記

記数法
　表記法, 66

説

説明, 71

ソースファイルセクション, 71
構文, 273

識

識別子
ST 用に予約, 64, 257
事前定義, 256
定式化のルール, 57
構文, 57, 265

警

警告クラス, 34, 215

配

配列
値割り付け, 101, 102
配列データタイプ指定
エラーソース, 80

関

関係式, 109