

ISaGRAF

Version 3.5

ユーザ ガイド

ICS Triplex ISaGRAF Inc.

はじめに

本ドキュメントの内容は予告無く変更されることがあります。また**ICS Triplex ISaGRAF Inc.**社は、その情報について責任を負うものではありません。

本ドキュメントに記載されているソフトウェア(あらゆるデータベース内の全ての情報を含む)はライセンス契約または非開示契約に基づいて提供されており、当該契約の規定に従ってのみ使用または複製することが可能です。ライセンス契約または非開示契約において、無断で本ソフトウェアを複製することは法律に違反します。

いかなる目的にも**ICS Triplex ISaGRAF Inc.**社の文書による明確な許可なく、これらのページの一部分または全部を複製することは、電子的であれ機械的であれ、その形態手段を問わずできません。

© 1994-2004 **ICS Triplex ISaGRAF Inc.** All rights reserved.
Published in Canada by **ICS Triplex ISaGRAF Inc.**

ISaGRAF は**ICS Triplex ISaGRAF Inc.** 社の登録商標です。

MS-DOS は Microsoft 社の登録商標です。

Windows は Microsoft 社の登録商標です。

Windows は Microsoft 社の登録商標です。

OS-9 and ULTRA-C は Microware 社の登録商標です。

VxWorks and Tornado Wind River Systems 社の登録商標です。

他のすべての商標または製品名はそれぞれの所有者の商標あるいは登録商標です。

本書に記載されていない最新の情報は、ワークベンチの README ヘルプファイルに記載されています。そちらもあわせてご覧下さい。

目次

A.	ワークベンチ ユーザガイド	A-11
A.1	はじめに	A-12
A.1.1	ISaGRAF のインストール	A-12
A.1.2	ライセンス登録	A-16
A.1.3	オンラインドキュメンテーションの使い方	A-19
A.1.4	サンプル アプリケーション	A-20
A.2	プロジェクトの管理	A-26
A.2.1	プロジェクトの新規作成と編集	A-27
A.2.2	複数プロジェクトグループの管理	A-30
A.2.3	オプション	A-31
A.2.4	ツール	A-31
A.2.5	ツールバーアイコン	A-31
A.3	プログラム管理	A-33
A.3.1	プロジェクト構成要素	A-33
A.3.2	プログラムの管理:「ファイル」「ツール」メニューコマンド	A-36
A.3.3	「コード生成」メニューコマンド	A-42
A.3.4	その他の ISaGRAF ツール:「プロジェクト」メニューコマンド	A-45
A.3.5	ツールメニューへのユーザコマンドの追加	A-45
A.3.6	アプリケーションのデバッグ、シミュレーション:「デバッグ」メニューコマンド	A-46
A.3.7	ツールバーアイコン	A-49
A.4	SFC エディタの使い方	A-51
A.4.1	SFC 言語メイントピック	A-52
A.4.2	SFC チャートの入力	A-54
A.4.3	SFC チャートの編集	A-56
A.4.4	レベル2プログラムの入力	A-58
A.4.5	SFC ギャラリーの使い方	A-62
A.4.6	ツールバーアイコン	A-63

A.5	フローチャートエディタの使い方	A-64
A.5.1	フローチャート(FC) 言語の基本要素	A-64
A.5.2	フローチャートの入力	A-66
A.5.3	フローチャートの編集	A-68
A.5.4	レベル2プログラムの入力	A-69
A.5.5	Quick LD によるプログラミング	A-71
A.5.6	表示オプション	A-71
A.5.7	ツールバーアイコン	A-72
A.6	Quick LDエディタの使い方	A-73
A.6.1	LD言語の基本事項	A-73
A.6.2	LDダイアグラムの入力	A-76
A.6.3	LDダイアグラムの修正	A-79
A.6.4	表示オプション:「オプション」メニューコマンド	A-80
A.6.5	オンラインヘルプ	A-83
A.6.6	ツールバーアイコン	A-83
A.7	FBD/LDエディタの使い方	A-84
A.7.1	FBD/LD 言語の基本事項	A-84
A.7.2	FBDダイアグラムの入力	A-87
A.7.3	FBDダイアグラムの編集:「編集」メニューコマンド	A-89
A.7.4	表示オプション	A-91
A.7.5	スタイルと編集箇所の記録	A-93
A.7.6	オンラインヘルプ	A-95
A.7.7	FBD ダイアグラムの印刷	A-95
A.7.8	ツールバーアイコン	A-95
A.8	テキストエディタの使い方	A-96
A.8.1	「編集」メニューコマンド	A-96
A.8.2	構文の着色	A-97
A.8.3	オプション	A-97
A.8.4	ツールバーアイコン	A-98
A.9	プログラムエディタ共通項目について	A-99
A.9.1	他の ISaGRAF ツールのコール	A-99
A.9.2	プログラムのパラメータ定義	A-99
A.9.3	その他の「ファイル」メニューコマンド	A-100
A.9.4	プログラムの修正履歴の更新	A-101
A.9.5	変数辞書から変数選択	A-101
A.9.6	アウトプットウィンドウ	A-102

A.10	辞書エディタの使い方	A-104
A.10.1	辞書エディタメインウィンドウ	A-106
A.10.2	変数の管理	A-108
A.10.3	オブジェクトの記述	A-110
A.10.4	クイック変数登録	A-113
A.10.5	SCADA Modbus アドレスマップ	A-114
A.10.6	他のアプリケーションとのデータ交換	A-115
A.10.7	ツールバーアイコン	A-119
A.11	I/O接続エディタの使い方	A-120
A.11.1	I/Oボードの定義	A-121
A.11.2	ボードパラメータの設定	A-122
A.11.3	I/O変数とI/Oチャンネルの接続	A-123
A.11.4	直接表現変数 (Directly Represented Variables) ..	A-123
A.11.5	ナンバリング	A-124
A.11.6	チャンネル単位のプロテクション設定	A-124
A.11.7	ツールバーアイコン	A-125
A.12	数値変換テーブルの作成	A-126
A.12.1	変換テーブルのメインコマンド	A-126
A.12.2	テーブルへの変換ポイント入力	A-127
A.12.3	ルールと制限	A-128
A.13	コードジェネレータの使い方	A-129
A.13.1	「ファイル」コマンドメニュー	A-129
A.13.2	コンパイラオプション	A-130
A.13.3	Cソースコード生成	A-133
A.13.4	コード生成情報のモニタ:「編集」メニューコマンド	A-134
A.13.5	リソースファイルの定義	A-134
A.14	クロスリファレンスの使い方	A-140
A.14.1	ツールバーアイコン	A-143
A.15	グラフィックデバッガの使い方	A-144
A.15.1	デバッガウィンドウ	A-144
A.15.2	アプリケーションの制御	A-145
A.15.3	オプション	A-148
A.15.4	書き込みコマンド	A-149
A.15.5	ロック状態とデバイス値の表示	A-152
A.15.6	オンライン修正の詳細	A-153

はじめに

A.15.7	DDE通信	A-158
A.15.8	ツールバーアイコン	A-158
A.16	変数のスパイ	A-159
A.16.1	ツールバーアイコン	A-160
A.17	ST、IL プログラムのデバッグ	A-161
A.18	SpotLight の使い方	A-163
A.18.1	グラフィックレイアウトの構築	A-163
A.18.2	リストレイアウト	A-166
A.18.3	スタイル設定	A-166
A.18.4	「ファイル」メニューコマンド	A-168
A.18.5	バージョン 3.2 ユーザへの注意事項	A-168
A.18.6	ツールバーアイコン	A-169
A.19	アプリケーションのアップロード	A-170
A.19.1	プロジェクトのアップロード	A-170
A.19.2	通信設定	A-171
A.19.3	アップロードに備えた準備	A-171
A.19.4	ターゲットにおける圧縮ソースの保管	A-173
A.19.5	ターゲットに必要なメモリ容量	A-173
A.19.6	アップロードプロジェクトに関して	A-173
A.19.7	バージョン間の互換性	A-174
A.20	デバッグ(診断)ツールの使い方	A-175
A.21	シミュレータの使い方	A-177
A.21.1	デバッグとのリンク	A-177
A.21.2	I/O シミュレーション	A-177
A.21.3	ライブラリコンポーネント	A-178
A.21.4	オプション	A-180
A.21.5	入力信号状態の保存／読み込み	A-181
A.21.6	サイクルプロファイル	A-181
A.21.7	シミュレーション スクリプト	A-182
A.21.8	ツールバーアイコン	A-189

A.22	ライブラリ管理ユーティリティの使い方	A-190
A.22.1	ライブラリ要素の管理:「ファイル」メニューコマンド	A-191
A.22.2	I/O 構成ライブラリ	A-195
A.22.3	I/O 装置機器ライブラリ	A-196
A.22.4	I/Oボードライブラリ	A-196
A.22.5	IEC言語によるファンクション、ファンクションブロック	A-199
A.22.6	C言語ファンクションとファンクションブロック	A-201
A.22.7	C言語数値変換関数	A-202
A.22.8	ツールバーアイコン	A-203
A.23	アーカイブユーティリティの使い方	A-204
A.23.1	アーカイブユーティリティの呼び出し	A-204
A.23.2	オプション	A-205
A.23.3	バックアップと復元	A-205
A.23.4	アーカイブファイルの拡張子	A-205
A.24	プロジェクトドキュメントの印刷	A-207
A.24.1	ドキュメントの目次のカスタマイズ	A-207
A.24.2	オプション	A-208
A.25	パスワードプロテクション	A-211
A.26	アドバンスド プログラミングテクニック	A-214
A.26.1	ISaGRAF ツールの詳細情報	A-214
A.26.2	ロックされたI/OとバーチャルI/O	A-214
A.26.3	通信リンク設定(ワークベンチターゲット)の動作確認 ...	A-217
A.26.4	ISaGRAF のディレクトリ構成	A-217
A.26.5	アプリケーションシンボル	A-219
A.26.6	ISaGRAF ワークベンチI/O無制限版(WDL)の制限 ..	A-223
B.	言語リファレンス	B-226
B.1	プロジェクトの構成	B-227
B.1.1	プログラム	B-227
B.1.2	サイクリック及びシーケンシャルなセクション	B-227
B.1.3	チャイルドSFC、FCプログラム	B-228
B.1.4	ファンクションとサブプログラム	B-229

はじめに

B.1.5	ファンクションブロック	B-230
B.1.6	言語の説明	B-231
B.1.7	実行ルール	B-231
B.2	共通オブジェクト	B-233
B.2.1	基本タイプ	B-233
B.2.2	定数表現	B-233
B.2.3	変数	B-235
B.2.4	コメント	B-239
B.2.5	定義ワード	B-239
B.3	SFC 言語	B-241
B.3.1	SFC チャートメインフォーマット	B-241
B.3.2	SFC 基本コンポーネント	B-241
B.3.3	分岐と結合	B-244
B.3.4	マクロステップ	B-246
B.3.5	ステップの中でのアクション	B-247
B.3.6	トランジションの条件	B-252
B.3.7	SFCの動作の原則	B-254
B.3.8	SFC プログラム階層化	B-254
B.4	フローチャート言語	B-256
B.4.1	FC の構成要素	B-256
B.4.2	FC 組み合わせ構造	B-259
B.4.3	FC の実行時の原則	B-260
B.4.4	FC の文法チェック	B-260
B.5	FBD言語	B-261
B.5.1	FBDダイアグラムのフォーマット	B-261
B.5.2	RETURN ステートメント	B-262
B.5.3	ジャンプとラベル	B-262
B.5.4	論理の反転	B-263
B.5.5	FBDからのファンクション、ファンクションブロックのコール ...	B-263
B.6	LD言語	B-265
B.6.1	母線と接続線	B-265
B.6.2	マルチプル接続(並列接続)	B-266
B.6.3	LDの基本的な接点とコイル	B-267
B.6.4	RETURN ステートメント	B-272

B.6.5	ジャンプ とラベル	B-273
B.6.6	LDの中でのブロック.....	B-273
B.6.7	ラダーにおける"In Line"ファンクションブロック	B-275
B.7	ST言語.....	B-277
B.7.1	STの主な文法.....	B-277
B.7.2	式と括弧.....	B-278
B.7.3	ファンクションやファンクションブロックのコール	B-278
B.7.4	ST専用ブール演算子	B-280
B.7.5	ST基本ステートメント.....	B-282
B.7.6	STの拡張.....	B-287
B.8	IL言語	B-293
B.8.1	ILの主な文法.....	B-293
B.8.2	ILオペレータ(命令).....	B-294
B.9	標準命令、ファンクションブロックとファンクション.....	B-300
B.9.1	標準命令	B-300
B.9.2	標準ファンクションブロック	B-322
B.9.3	標準ファンクション	B-341
C.	ターゲットユーザガイド	C-382
C.1	はじめに.....	C-383
C.2	ターゲットのインストレーション.....	C-384
C.3	DOSターゲットの使い方	C-385
C.3.1	ISaGRAF ターゲットの実行: ISA.EXE	C-385
C.3.2	DOSターゲットの特徴.....	C-387
C.4	OS-9ターゲットの使い方	C-391
C.4.1	シングルタスクモード ISaGRAF の実行: isa.....	C-391
C.4.2	マルチタスクモード ISaGRAF の実行: isaker, isatst, isanet	C-392
C.4.3	OS9ターゲットの特徴.....	C-397

C.5	VxWorksターゲットの使い方	C-402
C.5.1	システムリソースマネージャ: isassr.o	C-402
C.5.2	isa.o, isakerse.o, isakeret.o に共通の特徴	C-402
C.5.3	シングルトaskモード ISaGRAF の実行: isa.o	C-403
C.5.4	マルチタスクモード ISaGRAF の起動: isakerse.o , isakeret.o	C-405
C.5.5	VxWorksターゲットの特徴	C-410
C.6	NT ターゲットの使い方	C-414
C.6.1	ISaGRAF ターゲットの実行	C-414
C.6.2	各種オプションの概要	C-414
C.6.3	NTターゲットの特徴	C-419
C.6.4	ユーザインタフェース	C-424
C.7	C言語プログラミング	C-429
C.7.1	概要	C-429
C.7.2	C言語変換関数	C-430
C.7.3	C言語ファンクション	C-435
C.7.4	C言語ファンクションブロック	C-442
C.7.5	コンパイルとリンクのテクニック	C-458
C.8	MODBUS リンク	C-465
C.8.1	MODBUS ネットワークとプロトコル	C-465
C.8.2	ISaGRAF ターゲットとの MODBUS 通信	C-466
C.9	電源異常時の管理	C-472
C.9.1	基本事項	C-472
C.9.2	アプリケーション変数のバックアップ	C-473
C.9.3	プログラム状態のバックアップ	C-477
C.10	付録: エラーリストと説明	C-478
D.	用語集	D-488
E.	索引	E-502

A. ワークベンチ ユーザガイド

A.1 はじめに

ISaGRAF は、各種制御機器の制御プログラムを開発するワークベンチと制御プログラムを実行するターゲットからなるソフトウェアです。

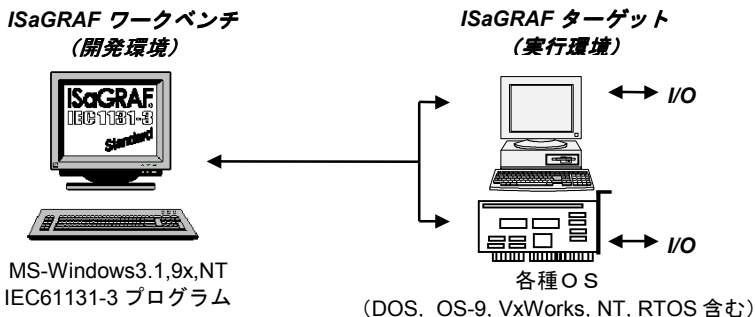
ワークベンチ(開発環境)は制御プログラミングに IEC61131-3 を採用し、さらに開発に必要なさまざまなツールを備えているので効率的に制御プログラムの開発を行えます。

ターゲット(実行環境)はワークベンチで開発された制御プログラムがパソコンやボード上でリアルタイムに実行されるソフトウェアで、CPU、OS に依存しないマルチプラットフォームでの実行が可能です。

本章では ISaGRAF ワークベンチのインストールに関する説明を行います。また、ISaGRAF の簡単なアプリケーションも取り入れて、ISaGRAF をすぐに使っていただけるような解説を行ないます。

A.1.1 ISaGRAF のインストール

ISaGRAF のセットアップは、SETUP.EXE というセットアッププログラムを使用していきます。このセットアッププログラムにより、ワークベンチの必要なファイルがハードディスクにインストールされます。ISaGRAF ターゲットのインストールに関しては ISaGRAF ターゲットユーザガイドの章も参照してください。



必要な環境条件

ISaGRAF ワークベンチのインストール条件としては Windows が実行可能な以下の条件を満足する環境が必要となります。

ハードウェア環境(ワークベンチ)

- パソコンはDOS/VあるいはNEC PC- 9821シリーズ
- CPUは80486以上
- メモリは8MB以上
- フロッピーディスクドライブは3.5インチ1.44MBドライブ
- CD-ROMドライブ

- ハードディスク空容量は20MB以上
- ビデオドライバはVGA、SVGA
- マウス
- プリンタ用パラレルポート LPT1 (プロテクションキーに必要)

ISaGRAF ワークベンチのインストールの前には以下のソフトがインストールされている必要があります。

ソフトウェア環境(ワークベンチ)

- Windows3.1 の 386 エンハンスドモード
- Windows95 以降
- WindowsNT version 3.51 または 4.00 以降



ワークベンチ インストレーションの手順

以下のインストールの手順は MS-Windows3.1 をベースに記述されていますが、MS-Windows9X/NT の場合もこれに相当する操作を行なって下さい。

ステップ1: Windowsの起動

Windows を起動してください。

インストールは Windows 上から実行する必要があります。

ステップ2: CD-ROMの挿入

CD-ROMをCD-ROMドライブに挿入します。

ステップ3: インストールファイルの実行

インストーラは自動的に起動します。

手動で起動する場合は、プログラムマネージャの“ファイル”メニューまたはスタートメニューの“ファイル名を指定して実行”を選択します。そこで、“X:\SETUP.EXE”をプログラムコマンドラインで入力します。

Xにはインストール時のドライブを指定します。(例えば、D:\SETUP.EXE)

または、エクスプローラ上から“SETUP.EXE”を直接ダブルクリックして実行します。

ステップ4: インストール項目の選択



上図のように ISaGRAF の製品メニューダイアログボックスが表示されます。CD-ROMからこれらの内容をインストールすることができます。日本語版ワークベンチをインストールする場合は、Language 選択ボックスで"Japanese"を選択してください。

選択後、Install ボタンを選択するとインストールを開始します。

画面の指示に従ってインストレーションを進めます。ISaGRAF のインストレーションディレクトリは過去のバージョンがハードディスクに存在する場合は、新規のディレクトリを指定して下さい。

ワークベンチインストレーションで以下のどのモジュールが必要か質問されます。

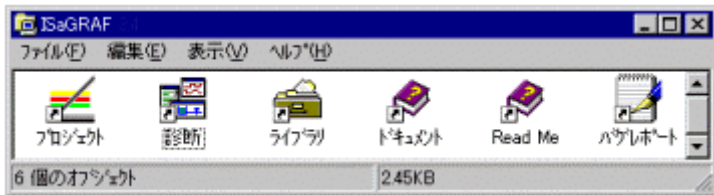
- ISaGRAF 実行プログラム
- オンラインドキュメント及びヘルプファイル
- ISaGRAF 標準ライブラリ
- ISaGRAF サンプルプログラム

初めてのインストレーションでは全部のモジュールを選択することをお薦めします。

ISaGRAF ワークベンチがインストールされるディレクトリは、デフォルトが"¥ISAWIN"となっています。全ファイルがコピーされた後は、プログラムマネージャには次の図に示すグループが追加されます。ハードディスクにコピーされた圧縮ライブラリファイルはコピー後に展開されます。

Windows のスタートメニュー(プログラムマネージャ)に"ISaGRAF 3.4"のグループ(あるいは、フォルダー)が登録され、インストール先の \EXE サブディレクトリには ISA.ini ファイルが作成されます。(ISaGRAF インストール用のサブディレクトリ以外にはファイルはコピーされません。)

注意: Windows ディレクトリに ISA.ini ファイルがある場合はこれを削除して下さい。(体験版デモプログラムを以前にインストールしていると、ISA.ini ファイルが Windows ディレクトリにコピーされている可能性があります。)



主な ISaGRAF のアイコンは以下の通りです。

プロジェクト:..... プロジェクト管理管理
診断:..... エンドユーザ用の診断ツール
ライブラリ:..... ライブラリ管理
ドキュメント:..... ISaGRAF のオンラインドキュメント
Read Me:..... ISaGRAF の新バージョンに関する情報
バグレポート:..... 提出用バグレポート様式

もし、バグなどの問題に直面した場合はバグレポート報告の様式に必要事項を記入の上、送付願います。

⇒ システムファイルの更新

ISaGRAF は DOS の環境変数は使いませんが、以下の内容が CONFIG.SYS の項目に入っていない場合は、追加する必要があります。既に存在している場合は数字が"20"以上であることを確認して、必要があれば変更して下さい。ISaGRAF の実行ディレクトリは path に含める必要はありません。

```
files=20
buffers=20
```

ISaGRAF ワークベンチは ISaGRAF ターゲットとの通信にデフォルトでシリアル通信(COM1)を使用します。もし、COM1 が既にマウスなどで使用中の場合は、可能であればマウスを COM2 に変更願います。

⇒ Windows NT ユーザに対する注意事項:

V3.3 以前のバージョンの ISaGRAF ワークベンチを WindowsNT 3.51, 4.00 上にインストールする場合、指定されたファイルに以下のライン(NT=1)をマニュアルで追加する必要があります。追加場所はファイル名 ISA.ini のセクション [WS001]でデフォルトのディレクトリは\ISAWIN\EXE です。

```
[WS001]
NT=1
Isa=C:\ISAWIN
IsaExe=C:\ISAWIN\EXE
IsaApl=C:\ISAWIN\APL1
IsaTmp=C:\ISAWIN\TMP
```

なお、この設定は COM1 などを使用したシリアル通信の際に必要となります。

また、この設定は、V3.4 以降のバージョンでは自動的に OS を認識して挿入されますので、ユーザが設定する必要はありません。

A.1.2 ライセンス登録

現在インストールされている ISaGRAF は制御アプリケーションを構築することが可能です。ライセンスが必要になるまでの30日間は試用期間として使用することが可能です。ライセンスを保持していない場合、プログラムや変数をエクスポートしたり、ターゲットに対するワークベンチプロジェクトソースコードのダウンロード、およびアップロードを行うことが出来ません。これらはライセンスを取得することで可能となります。

ISaGRAF のライセンスにはハードウェアキーもしくはソフトウェアキーを使用することができます。ハードウェアキーを使用する場合、お使いになるマシンのパラレルポート、もしくは USB ポートを使用していただくことになります。これらのキーは使用するワークベンチの種類によって数種類用意されています。ソフトウェアキーを使用する場合は、**ICS Triplex ISaGRAF Inc.**社から発行されるライセンスコードが必要となり、ISaGRAF 付属のライセンスマネージャよりライセンス登録を行います。

ハードウェアキー（パラレルポートタイプ）はどのパラレルポートにも使用可能です。もし、2つ以上のパラレルポートが使用可能ならハードウェアキーとプリンタは別々のポートを使用することをお勧めします。マシンとプリンタの環境によってはオフラインプリンタに接続されている場合、ハードウェアキーが認識されない場合があります。このような場合、プリンタとの接続をはずすかもしくはオンライン状態でスタートさせてから、ISaGRAF 自体を再起動させて下さい。

Note: Windows NT 上でハードウェアキーを使用する場合、Sentinel driver をインストールする必要があります。ISaGRAF CD-ROM のルートフォルダにある Sentinel folder フォルダ内の Setup.exe を起動することでこのドライバのインストールを行います。

ISaGRAF では以下の2種類のタイプが利用可能です。

- IO 数制限版（1 点から 4 0 9 6 点まで）
- IO 数無制限版

どちらのタイプについても ST 言語と IL 言語の使用が含まれています。しかし、その他の言語を使用する場合は以下の言語を指定しライセンスを取得する必要があります。

- SFC (Sequential Function Chart)
- FC (Flow Chart)
- FBD (Function Block Diagram)
- LD (Ladder Diagram)

ライセンスはそのマシンでのみ有効となりますが、当該マシンから別のマシンに移動させることも可能です。

ライセンスマネージャを立ち上げるには

- Windows のスタートメニューから「プログラム」を選択し、ISaGRAF3.5 の「ライセンス」を選択します。

A.1.2.1 ライセンスの追加

ISaGRAF のライセンスを取得することができます。.

ISaGRAF のライセンスを取得する為には

ライセンス登録を行うにはユーザコードのセットとレジストレーションキーのセットが必要となります。

1. 「ライセンス」タブ上で、登録可能な項目リストから ISaGRAF を選択します。
2. Click →で選択項目リストに選択した項目が移動されます。

ここで以下のタイプを1つ選択します。

- IO 数制限版（1 点から 4 0 9 6 点まで）
- IO 数無制限版

セットアップコードとユーザコード1、ユーザコード2がそれぞれのフィールドに表示されます。

3. ライセンス情報の送付
 - a) 「送信」をクリックしてください。

アドレスが入力された E-mail が表示されます。この E-mail にはコンタクト情報と購入番号が含まれたセットアップコードと2つのユーザコードが記述されています。追加購入に関してはクレジットカード番号が必要です。

- b) 必要な情報が記述されている場合、送信します。

レジストレーションキー1、2と一緒にセットアップコードとユーザコードを E-mail で返信します。

4. 返信メール受信時は、セットアップコードとユーザコードがライセンスマネージャウィンドウのものと同じかどうか確かめてください。その後カット&ペーストでそれぞれのフィールドにレジストレーションキーをコピーします。
5. 「決定」をクリックして下さい。

ライセンス登録が成功していれば、選択項目リストの ISaGRAF は灰色表示になります。

6. ライセンス登録を有効にする場合は、ISaGRAF を再起動して下さい。

A.1.2.2 ライセンスの移動

あるマシンから別のマシンにライセンスを移動できます。

新しいマシンにライセンスを移動させるには

あるマシンから別のマシンへのライセンスの移動には、ライセンス移動用のディスクが必要です。また新しいマシンにインストールする前に現在ライセンス登録されているマシンからライセンスを消去しライセンス移動用ディスクにコピーします。

1. 新規のマシンから、ライセンス移動用ディスクの準備をします。
 - a) ISaGRAF をインストールします。
 - b) マシンのドライブにフロッピーディスクを挿入します。
 - c) Windows のスタートメニューから「プログラム」を選択し ISaGRAF3.5 の「ライセンス」を選択します。
 - d) 「ライセンス移動」タブで、フロッピーディスクが入っているドライブを選択し「**ライセンスディスクの作成**」を選択します。

これでライセンス移動用ディスクが作成されます。

- e) ライセンス移動用ディスクを新規マシンのドライブから取り出します。
2. ライセンスを現在のマシンからライセンス移動用ディスクに移動します。
 - a) 現在ライセンス登録されているマシンのドライブにライセンス移動用ディスクを挿入します。
 - b) Windows のスタートメニューから「プログラム」を選択し ISaGRAF3.5 の「ライセンス」を選択します。
 - c) 「ライセンス移動」タブで、フロッピーディスクが入っているドライブを選択し「**ライセンスを移動する**」を選択します。

ライセンスはマシンから削除され、ディスクにコピーされます。

- d) ライセンス移動用ディスクをマシンのドライブから取り出します。
3. 新規マシンへのライセンスのインストール
 - a) 新しいマシンのドライブにライセンスがコピーされたライセンス移動用ディスクを挿入します。
 - b) Windows のスタートメニューから「プログラム」を選択し、ISaGRAF3.5 の「ライセンス」を選択します。
 - c) 「ライセンス移動」タブで、フロッピーディスクが入っているドライブを選択し、**移動の完了**」を選択します。

新しいマシンにライセンスが移動され、ISaGRAF 3.5 が利用可能になります。

A.1.2.3 ライセンスの削除

マシンからライセンスを削除することが出来ます。

マシンからライセンスを削除するには

1. Windows のスタートメニューから「プログラム」を選択し ISaGRAF3.5 の「ライセンス」を選択します。
2. 「ライセンスの削除」タブで、ライセンス登録されている項目リストからライセンスを削除したい項目を選択します。
3. をクリックすると選択リストに項目が移動します。

ユーザコード1, 2と一緒にセットアップコードがフィールドに表示されます。

4. ライセンス情報の送信
 - a) 「送信」をクリックしてください。

アドレスが入力された E-mail が表示されます。この E-mail にはコンタクト情報と購入番号が含まれたセットアップコードと2つのユーザコードが記述されています。

- b) 必要な情報が記述されている場合、送信します。

レジストレーションキー1, 2と一緒にセットアップコードとユーザコードを E-mail で返信します。

5. 受信時は、セットアップコードとユーザコードがライセンスマネージャウィンドウのものと同じかどうか確かめてください。その後カット&ペーストでそれぞれのフィールドにレジストレーションキーをコピーします。その後「決定」をクリックして下さい。

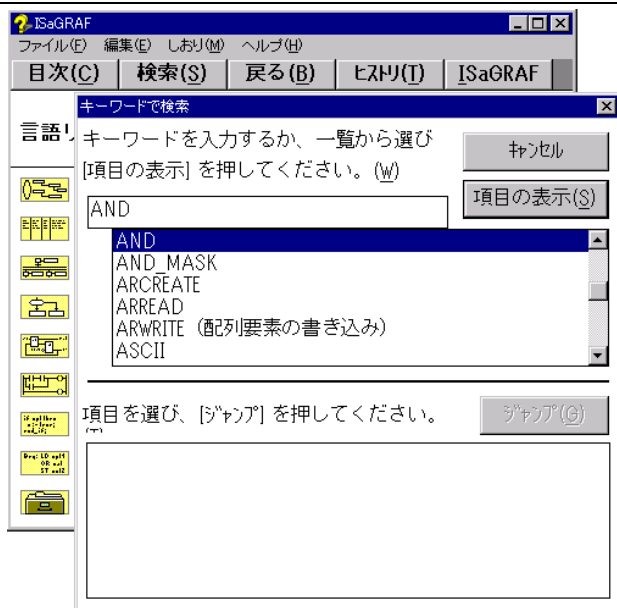
確認コードが当該フィールドに表示されます。

6. 名前、住所、電話番号と一緒に確認コードが記述された E-mail を返信してください。

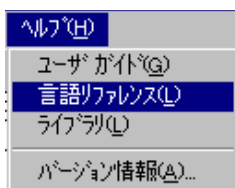
A.1.3 オンラインドキュメンテーションの使い方

オンラインドキュメンテーションは以下のトピックに関して ISaGRAF ワークベンチにインストールされています。

- ISaGRAF 言語リファレンスマニュアル
- ユーザガイド (ISaGRAF 全般の情報)
- 標準ライブラリ内の要素に関する技術メモ



各 ISaGRAF ウィンドウから“ヘルプ”メニューを選択することで現在使用中のプログラム言語やツールに関して適切なオンラインドキュメントを引出すことができます。



また、プログラム言語リファレンスマニュアルはウィンドウズのプログラムマネージャの“ドキュメント”アイコンから引出すことができます。



A.1.4 サンプル アプリケーション

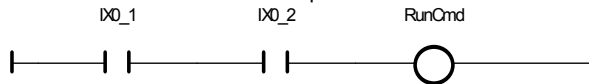
ここでは、コンパクトなアプリケーションを設計、作成、テストするための全ての基本的操作に関して順を追って解説します。
以下の図はLD(ラダー図)とSFC(シーケンシャルファンクションブロック)の表現で書かれたアプリケーションを示します。

詳細な解説は後の章で説明がありますので、ここではプログラム作成からシミュレーションまでの流れをつかむことができます。

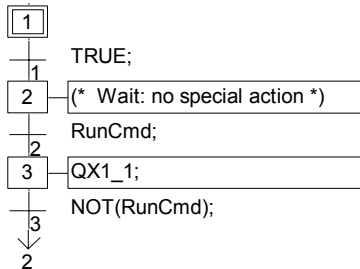
変数辞書への登録(ブール型変数):

変数名	コメント
IX0_1, IX0_2:	入力変数:プロセスコマンド
RunCmd:	内部変数:"run/stop"コマンド
QX1_1:	出力変数:プロセスの状態

プログラム名: Command: Begin セクション – QuickLD 言語
 "run/stop" 内部コマンド生成



プログラム名: RunStop: シーケンシャル セクション- SFC 言語
 プロセスのロック



Start ISaGRAF ワークベンチの起動

ISaGRAF ワークベンチを起動するために ISaGRAF グループ(フォルダ)内の“プロジェクト”アイコンをダブルクリックします。これで、プロジェクト管理ウィンドウが開きます。



プロジェクトの作成

「ファイル」メニューの「新規作成」コマンドで“RunStop”という名前でプロジェクトを作成します。

ダイアログボックスでは以下のように入力します。

プロジェクト名: "RunStop"
 I/O構成の選択: "Sim_Boo"
 "OK" ボタン

これで、プロジェクトが作成されます。

(I/O構成の選択を行なうことで、I/O変数の辞書登録は自動的に行なわれません。)



プログラム管理ウィンドウを開く

プロジェクト内のプログラムはプロジェクト管理ウィンドウで「ファイル」メニューの「編集」コマンドを選択するか、プロジェクト名をダブルクリックすることでプログラム管理ウィンドウを開くことができます。（この時点ではプログラムはまだ作成されていません。）



プログラムの作成

ここで、「ファイル」メニューの「新規作成」コマンドを選択してまず最初のプログラムを作成します。ダイアログボックスが開きますの以下のように入力、選択します。

プログラム名の入力: **"Command"**.

言語の選択: **"Quick LD"**

プログラムセクションの選択: **"Begin"** セクション

"OK" ボタン

同様に、2つめのプログラムを作成します。

「ファイル」メニューの「新規作成」コマンドを選択して、まず最初のプログラムを作成します。ダイアログボックスが開きますの以下のように入力、選択します。

プログラム名の入力: **"RunStop"**.

言語の選択: **"SFC"**

プログラムセクションの選択: **"シーケンシャル"** セクション

"OK" ボタン

これで、2つのプログラムが作成されました。これらはプログラム管理ウィンドウに表示されます。



変数の宣言(辞書エディタを開く)

プログラムの内容を作る前にプログラムで使用される内部変数を事前に宣言しておく必要があります。このために「ファイル」-「辞書」メニューコマンドで辞書エディタを開きます。なお、I/O変数に関しては、プロジェクトを新規作成したときI/O構成を選択したため、既に自動的に宣言されています。（プロジェクトを作成するときに“sim__boo”を選択したためシミュレーション用のI/Oが自動生成されています。）



グローバル変数の登録

辞書エディタで「ファイル」-「その他」コマンドから“グローバル変数”、“ブール型”を選択します。この操作でグローバル変数のブール型変数用のエディタが開きます。（同様の操作のためにツールバーアイコンの選択も可能です。）



「編集」-「新規作成」コマンドにより新規のブール型変数を作成（登録）することができます。（同様の操作のためにツールバーアイコンから“オブジェクトの挿入”を選択することも可能です。）ダイアログボックスが開いたら以下のように入力を行います。

名前: **RunCmd**

コメント: **Run/Stop コマンド(内部)**

属性: **"内部"** 属性の選択

"保存" ボタンの選択で変数が作成されます。

"キャンセル" ボタンでダイアログボックスでの設定内容をキャンセルします。

最後に、修正内容の保存をしてから辞書エディタを終了します。「ファイル」-「終了」コマンドを選択して「修正内容を保存しますか？」に対して“はい”を選択します。



Quick LD プログラムの編集

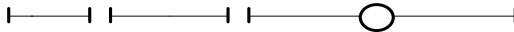
"Command"LDプログラムを編集してみましょう。プログラム管理ウィンドウでプログラム名をダブルクリックするか、「プログラムの編集」ボタンをクリックします。



Quick LD エディタが開きます。ウィンドウのサイズを必要に応じて最大化などを選択して調整します。

F2 F3 ファンクションキー F2、F3 キーを入力(又は、マウスでツールバーボタンをクリック)します。

(' ')



LD図のシンボルに対応した変数を割り付けます。キーボードのカーソルキーやマウスでシンボルを選択して、ENTER キーの入力をします。変数の選択用のダイアログボックスが開きます。(選択する変数のスコープが“グローバル”が選択されて、変数の型が“ブール型”となっていることを確認して下さい。)

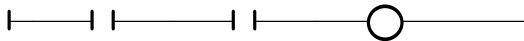
最初のa接点には IX0_1 を選択して “OK” ボタンを入力

次のa接点には IX0_2 を選択して “OK” ボタンを入力。

コイルには RunCmd を選択して “OK” ボタンを入力

これで、LDプログラムの入力ことができました。結果は以下のようになります。

IX0_1 IX0_2 RunCmd



プログラムの作成が完了しましたので、プログラムエディタを終了します。終了時に修正内容を保存して下さい。



SFC プログラムの編集

"RunStop" SFC プログラムを編集するために、プログラム管理ウィンドウでプログラム名をダブルクリックします。



SFCプログラムエディタが開きます。必要に応じてウィンドウサイズを変更又は最大化します。

SFCエディタのオプションを選択します。

「オプション」メニューで「レイアウト」コマンドを選択してすべてのオプションを選択すると、言語のツールバーやカーソルの座標がSFCエディタのステータスバーに表示されます。

SFCの初期ステップは既に存在しています。



最初のトランジションの入力のためにツールバーアイコンから対応するアイコンを選択します。

セル座標 (0,1) を選択して、ファンクションキーF4 あるいは、ツールバーボタンを押してトランジションを配置させます。

はじめに

以降、残りのステップとトランジションをプログラムします。セルの選択は自動的に下のセルに移動しますので、F4、F3キーを交互に押すことでステップとトランジションは交互に入力が可能です。

セル座標(0,2)を指定します。ステップを配置します。

セル座標(0,3)を指定します。トランジションを配置します。

セル座標(0,4)を指定します。ステップを配置します。

セル座標(0,5)を指定します。トランジションを配置します。



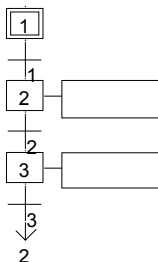
ジャンプシンボルをセットします。セル座標(0,6)を指定し、ツールバーから左のアイコンを選択するか、ファンクションキーのF5を選択します。

ジャンプ先のステップを選択します。

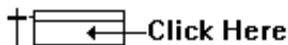
“GS2”を選択して“OK”ボタンを選択します。



以下のようなSFCチャートが完成しました。ツールバーの“ズーム”ボタンをクリックしてセルの表示サイズを拡大し、レベル2のコードを表示できるようにします。



次に“レベル2”プログラミングの入力をステップやトランジションに対して行いません。トランジションシンボルの下図の部分をクリックまたはエンターキーを押すか、「編集」-「レベル2の編集」コマンドを選択します。:



(0,3)

セル座標(0,3)のダブルクリック

ウィンドウが分割され、右側にテキストエディタが開きます。ここで、STプログラム（又は、LDプログラム）を入力することができます。トランジション“2”の条件として以下の入力を行います。

RunCmd;

同様の操作をステップ“3”に対して行います。

^TAB

“Control + Tab” でフォーカスが SFC チャートに移ります。ステップ3を選択します。

(0,4)

セル座標 (0,4)をダブルクリック

レベル2エディタウィンドウに、ステップ“3”に以下の入力を行います。

QX1_1;

- (0,5) 次にトランジション“3”の入力を行います。ウィンドウをスクロールして表示させます。
セル座標 (0,5)をダブルクリック
テキストエディタが開きます。トランジション“3”の条件として以下の入力を行います。

Not (RunCmd);

- ▲F4 "Control + F4" でレベル2のウィンドウをクローズします。

以上でSFCプログラムが完了しました。「ファイル」-「終了」で修正内容を保存してSFCエディタを終了します。
プロジェクトに必要な2つのプログラムの作成が終了しましたので、次にプログラム実行できるコードを生成します。



アプリケーションコード生成

プログラム管理ウィンドウの「コード生成」-「アプリケーションコード生成」コマンドかツールバーからアイコンを選択して中間コードを生成します。

コード生成が完了すると、ダイアログボックスが表示され“終了”、“継続”の選択ができますが、ここで“終了”を選択します。

(もし、入力ミスなどがありますと、コード生成中のコンパイラがエラーを検出します。この場合は、エラーメッセージ: 赤色表示 をダブルクリックして対象となっているプログラムの修正を行った後に再度コード生成を行います。)



シミュレーション

プログラム管理ウィンドウで「デバッグ」-「シミュレート」コマンドかツールバーのアイコンを選択して、ISaGRAF カーネルシミュレータを起動します。シミュレータウィンドウ(仮想I/Oパネル)上からアプリケーションをテストすることができます。この例では、入力の1、2(緑色)が押されると出力(赤色)が点灯します。

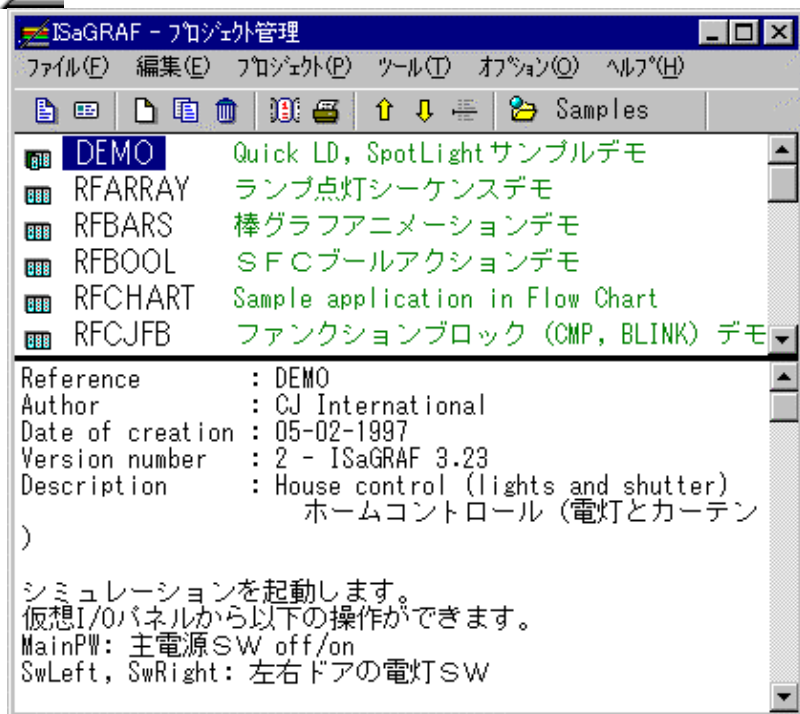
プログラム管理ウィンドウでプログラムをダブルクリックすることでプログラムがモニタモードで開きます。

シミュレーションデバッグを終了するには、デバッグウィンドウの「ファイル」-「終了」コマンドを選択します。

A.2 プロジェクトの管理



ISaGRAF プロジェクト管理ツールの立ち上げには ISaGRAF グループの“プロジェクト”アイコンをダブルクリックします。プロジェクト管理ウィンドウが開きます。



一つのプロジェクトは ISaGRAF ターゲット上で一つのアプリケーションに相当します。プロジェクト管理ウィンドウの上部には存在しているプロジェクトのリストを、下部ウィンドウには選択されているプロジェクトの記述(解説)が表示されます。



ウィンドウのリサイズ

上下のウィンドウ間のセパレータをクリックしてウィンドウのサイズを上下に変更することができます。ただし、下部のプロジェクトの記述ウィンドウは最小でも1行のテキストが表示されます。



プロジェクト間セパレータの挿入

プロジェクト間にセパレータを挿入することができます。このセパレータによってプロジェクトのグループ分けを行うことができます。「編集」-「セパレータの挿入/消去」コマンドにより選択プロジェクトの直前にセパレータが挿入又は消去できます。



リスト中のプロジェクトの移動

リスト中のプロジェクトの上下の移動を行うことができます。プロジェクトを移動させるには、プロジェクトを選択（ハイライト）して、このプロジェクトを移動先までドラッグさせます。この際に、左隅に矢印アイコンが表示されて移動先を示します。「編集」-「リストの上／下へ移動」コマンドによって移動も行えます。セパレータがプロジェクトの直前にある場合は、プロジェクトと共にセパレータも移動します。

A.2.1 プロジェクトの新規作成と編集

プロジェクト管理ウィンドウのメニューを使って、プロジェクトの新規作成、編集、管理が行えます。



プロジェクトの新規作成

「ファイル」-「新規作成」コマンドにより、新規のプロジェクトを作成することができます。空の新しいプロジェクトが開きます。ライブラリに登録済みのI/O構成の選択を行うことができます。もし、I/O構成の選択を行うと、辞書エディタやI/O接続エディタで入出力変数が自動的に設定されます。プロジェクト名は以下のルールに従って入力します。

- プロジェクト名は最大半角8文字
- 最初の文字は英文字（a-z）
- 以降の文字は英文字、数字、_ のいずれか
- 大文字、小文字の区別なし（全角入力はできません）

プロジェクトが作成されると、「編集」-「コメントの設定」コマンドでプロジェクトコメントを記入します。このコメントはプロジェクトの右側に表示されます。



プロジェクト記述の編集

「プロジェクト」-「プロジェクトの記述」コマンドの選択でプロジェクトを記述するためのテキストエディタが開きます。プロジェクトを管理する上でもプロジェクト間での違いを表現する上でもプロジェクトのライフサイクルにわたり活用されます。



プロジェクトの編集

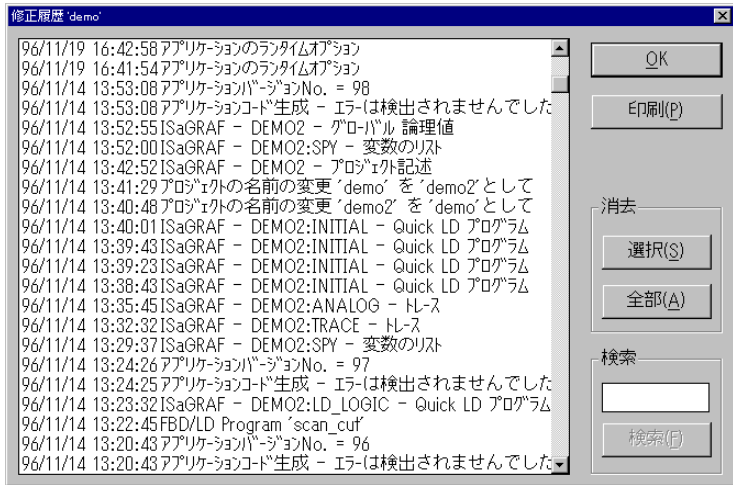
「ファイル」-「開く」コマンドにより、選択されたプロジェクトの編集が可能になります。即ち、プログラム管理ウィンドウが開きます。プログラム管理ウィンドウからはプロジェクトの全ての内容（プログラム、パラメータ、I/O接続、グラフィックス、コンパイラ設定...）を管理することができます。「開く」の代わりにプロジェクト名をマウスでダブルクリックすることでプログラム管理ウィンドウを開くことができます。



修正履歴

「プロジェクト」-「修正履歴」コマンドにより、プロジェクトに関する修正履歴のファイルの内容を表示及び印刷ができます。修正履歴内容の一部、あるいは全体の削除も行えます。

プロジェクト修正履歴ファイルにはプロジェクト内のプログラムの全ての修正内容（コード生成、アプリケーションダウンロード...）が日付、時刻、タイトルを含む形式で保存されます。修正履歴ファイルには最大500件の内容が保管されます。



修正履歴ダイアログボックスでは、以下の操作が可能です。

OK ダイアログボックスを閉じる

印刷 修正履歴内容の印刷

消去—選択 選択されたリストを削除

消去—全部 全リストの削除

検索 入力された文字列の検索

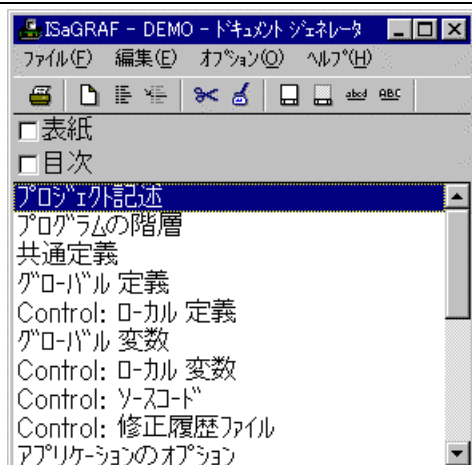
文字列検索時には、“**検索**”ボタン上の入力フィールドに検索したい文字列を入力します。入力文字列は大文字・小文字の区別があります。検索がリストの最後にきても引き続きリストのトップから検索を続けます。

注意: プロジェクトの修正履歴は個々のプログラムに付属している修正履歴のまとめとなっています。



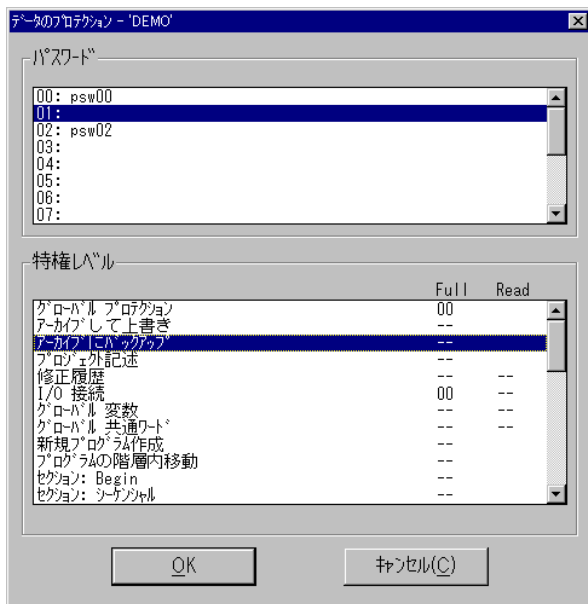
ドキュメント印刷

「プロジェクト」-「印刷」コマンドにより、選択中のプロジェクトに関する目次付きの完全ドキュメントを印刷することができます。このドキュメントは多くのセクション（プログラムソースコード、クロスリファレンス、修正履歴...）から成り立っています。部分的なドキュメント作成時は目次の中から必要なセクションのみを選択して（残して）印刷することが可能です。



パスワードプロテクション

「プロジェクト」-「パスワードセット」コマンドにより、選択中のプロジェクトに対して最大15種類のパスワードを設定できます。パスワードによって個別の機能毎に保護をかけることが可能です。ここで設定されたパスワードは他のプロジェクトやライブラリプロジェクトには一切無効となります。

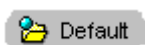


A.2.2 複数プロジェクトグループの管理

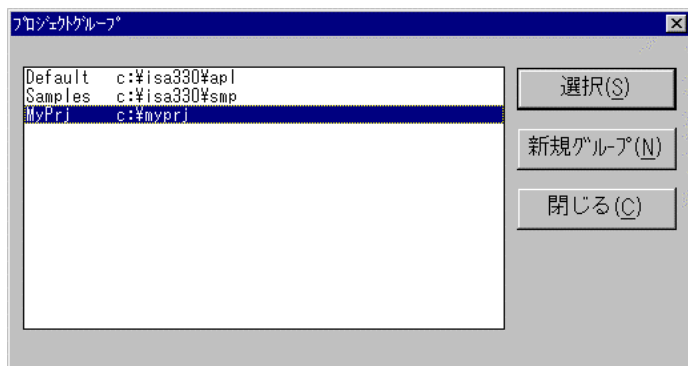
ISaGRAF プロジェクトは、ある一つのディレクトリに全プロジェクト(各プロジェクトは一つのサブディレクトリ)が保存されています。プロジェクトグループはその名前によって識別されます。デフォルトでは以下の2つのプロジェクトグループが作られます。

"Default" "ISAWIN\APL" : デフォルト作業エリア用サブディレクトリ
"Samples" "ISAWIN\SMP" : ISaGRAF ワークベンチに含まれるサンプルプロジェクト用サブディレクトリ

選択されているプロジェクトグループ名はツールバー上に表示されています。このボタンの選択で別のプロジェクトグループに切り替えることができます。



なお、「ファイル」-「プロジェクトグループの選択」コマンドにより、以下のダイアログボックスが開き既存のプロジェクトグループの選択や新規のプロジェクトグループの作成を行うことができます。



プロジェクト管理リストの内容を別のプロジェクトグループに切り替えるためには、目的のプロジェクトグループリストを選び、「選択」ボタンを押します。プロジェクトグループを選択してダブルクリックしても選択ができます。「新規グループ」ボタンにより、新規のプロジェクトグループを定義することができます。この際、プロジェクトグループ名を存在しているサブディレクトリ名に割り当てたり、新規サブディレクトリの作成が行えます。

注意: あるプロジェクトのウィンドウ(例えば、プログラムエディタや辞書エディタ)が開いているときは、別のプロジェクトグループへの切り替えや新規プロジェクトグループの作成はできません。

A.2.3 オプション

「オプション」メニューの「**プロジェクトマネージャを開き続ける**」がチェックされている場合は、プロジェクトを開いても、プロジェクト管理ウィンドウは開いたままですが、チェックをはずした場合は、プロジェクトを開くとプロジェクト管理ウィンドウは、自動的に閉じます。

「オプション」メニューで「**ツールバー表示**」がチェックされている場合はツールバーボタンが表示されます。

「オプション」メニューの「**フォント**」コマンドにより、プロジェクト記述や ISaGRAF テキストエディタのテキストフォントを選択できます。

A.2.4 ツール













「**ツール**」メニューコマンドから他の ISaGRAF ユーティリティを起動させることができます。「**ツール**」-「**アーカイブ**」-「**プロジェクト**」コマンドにより、プロジェクトのバックアップ、復元が行えます。「**ツール**」-「**アーカイブ**」-「**共通データ**」コマンドにより、全プロジェクトに共通なデータ(例えば、共通定義ワード)のバックアップ、復元ができます。

「**ツール**」-「**ライブラリ**」コマンドにより、ライブラリ管理ウィンドウが開きます。

「**ツール**」-「**IL プログラムのインポート**」コマンドにより、PLC Open で規定されたファイルフォーマットの IL プログラムのテキストファイルが1つのプロジェクトとしてインポートされます。

A.2.5 ツールバーアイコン

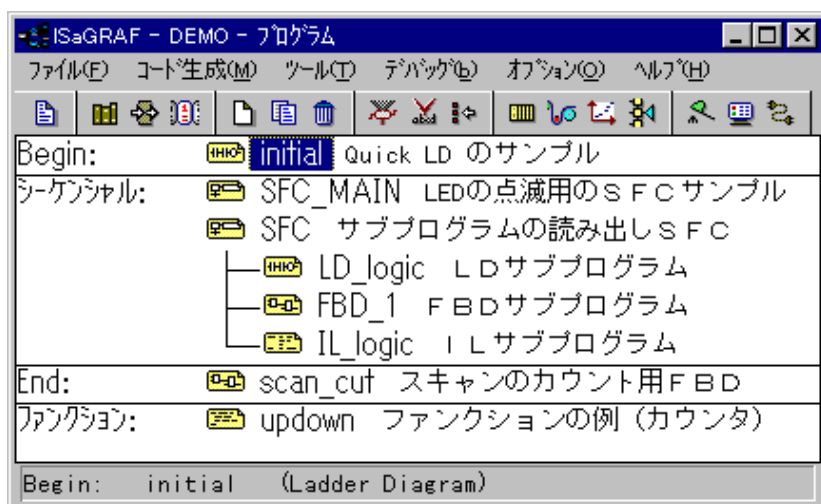
以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。

-  開く
-  プロジェクト記述の編集
-  新規プロジェクト作成
-  プロジェクトのコピー
-  プロジェクト削除
-  修正履歴
-  全ドキュメントの印刷
-  リストの上に移動
-  リストの下に移動
-  セパレータ挿入/消去
-  プロジェクトグループの選択
-  ISaGRAF 3.4.

A.3 プログラム管理



プログラム管理ウィンドウはプロジェクト(アプリケーション)で使われる全**プログラム(モジュール、プログラムユニット)**を表示します。ここでは、プロジェクト構成のデザイン、プログラムの編集エディタの起動、コンパイラやデバッカーの起動などが行なえます。このウィンドウはアプリケーションを開発する場合のワークベンチの中心的なウィンドウです。プロジェクト管理ウィンドウの「ファイル」-「開く」(あるいはプロジェクト名をダブルクリック)で開くことができます。



A.3.1 プロジェクト構成要素

プロジェクトを構成している要素はプログラムです。プログラムは制御の内容を論理的に記述したものです。グローバル変数(例えばI/O変数)はアプリケーション全体のいかなるプログラムからも使えます。ローカル変数は1つのプログラムからのみアクセスできます。プログラムは階層的に構成されて3種類のセクション(Begin、シーケンシャル、End)に分割されています。プログラム管理ウィンドウはプロジェクト内のプログラムのこの階層的な構成、関連づけを示します。どのセクションにも階層の左側にくるプログラムはトップレベルプログラムと呼ばれます。

■ トップレベルプログラム

3つのどのセクションにおいてもトップレベルプログラムは階層ツリーの左端に位置します。トップレベルのプログラムは、ランタイムサイクル中(スキャン時)は常に活性状態(アクティブ)となります。以下の順で実行されます。

- (入力取り込み)
- **Begin**セクションのトップレベルプログラム実行
- **シーケンシャル**セクションのトップレベルプログラム実行
- **End**セクションのトップレベルプログラム実行
- (出力更新)

Beginと**End**セクションのプログラムには時間に依存しない、サイクリックに実行される処理が記述されます。**シーケンシャル**セクションのプログラムには、基本となる処理を時系列的に処理をする、シーケンシャルな処理を記述します。**Begin**セクションのトップレベルのプログラムは毎サイクルの最初に実行されます。同様に**End**セクションのトップレベルのプログラムは毎サイクルの最後に実行されます。**シーケンシャル**セクションのトップレベルプログラムは **SFC** か **FC** の実行ルールにのっとって実行されるので、必ず**SFC**あるいは**FC言語**で記述されなければなりません。一方、**Begin**、**End**セクションのトップレベルプログラムは**SFC**あるいは**FC言語**での記述はできません。全てのセクションでどのプログラムも一つの**サブプログラム**を持つことができます。

□ ファンクション、ファンクションブロック セクション

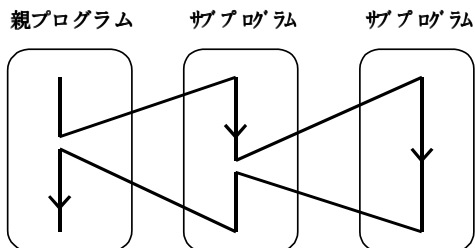
ファンクションセクションや**ファンクションブロックセクション**はどのセクションのどのプログラムからでも呼び出せるサブプログラムとなります。ファンクション、ファンクションブロックセクションのプログラムは**SFC/FC言語**では記述できません。**ファンクション**は複数の入力パラメータと1つの出力パラメータ(戻り値)を持ちます。ファンクションの中では変数の内容は毎回消されてしまいます。即ち、ファンクションからファンクションブロックをコールすることはできません。

ファンクションブロックは入力パラメータと内部(hidden)のスタティックデータから出力パラメータ(戻り値)を出力します。この内部のスタティックデータはファンクションブロックが呼ばれる毎にコピーされます。(即ち、使用するファンクションブロック毎にインスタンスが生成されます。)

□ サブプログラム

サブプログラムは、ファンクションセクションのプログラムと同じ性質をもっていますが、一つの親プログラムしか呼び出して実行することはできません。親のプログラムは、**SFC**、**FC**を含む全ての **IEC 言語**で記述できます。一方、サブプログラムは、**SFC** と **FC 言語**で記述することはできません。

親プログラムの実行はサブプログラムの処理が完了するまで停止しています(下図参照)。



どのセクションのどのプログラムも、一つ以上のサブプログラムを持つことができますが、サブプログラムの親は1つのプログラムとなります。サブプログラムは**ローカル変数、ローカル定義ワード**を持つことができます。

■ チャイルドSFCとFCプログラム

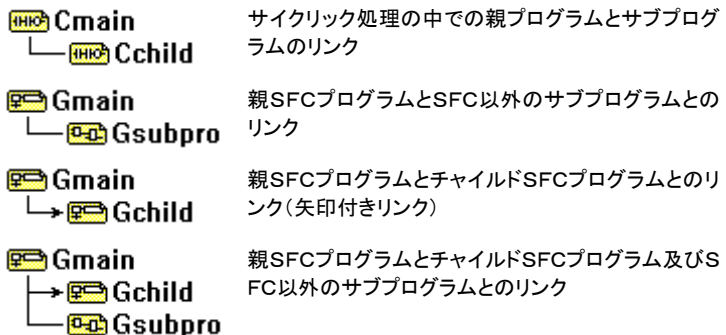
チャイルドSFCプログラムは親プログラムから起動(start)、停止(stop)、凍結(freeze)がかけられるパラレル処理プログラムです。親プログラムもチャイルドプログラムも共に**SFC言語**で記述される必要があります。

- 親SFCプログラムがチャイルドSFCプログラムを**起動**すると**SFCTークン**をチャイルドプログラムのイニシャルステップにセットします。チャイルドSFCの起動は、**GSTART**ステートメントまたは**＜チャイルドプログラム＞(S)**;で行ないます。
- 親SFCプログラムがチャイルドSFCプログラムを**停止**させるとチャイルドプログラムに存在していたSFCTークンが全てクリアされます。チャイルドSFCの停止は**GKILL**ステートメントまたは**＜チャイルドプログラム＞(R)**;で行ないます。
- 親SFCプログラムがチャイルドSFCプログラムを**凍結**させるとチャイルドプログラムに存在していたSFCTークンの全てを取り除きますが、この時トークンの場所を全て記憶しておきます。このコマンドは**GFREEZE**ステートメントで行ないます。
- 親SFCプログラムが凍結されたチャイルドSFCプログラムを**再起動**させるとチャイルドプログラムが凍結されたときに取り除かれたSFCTークンを全てもとの場所に戻します。このコマンドは**GRST**ステートメントで行ないます。

シーケンシャルセクションでのFC言語プログラムは別のFCサブプログラムをコントロールすることができます。FCサブプログラムが実行されているときは親のFCプログラムの処理は停止しています。従って、親FCプログラムとそこから読み出されるFCサブプログラムは同時実行はされません。







■ プログラムとサブプログラムのリンク

サブプログラムとチャイルドプログラムは、階層ツリーの中で、親プログラムとリンクで結ばれて表示されます。親SFCプログラムとチャイルドSFCプログラムとのリンクは矢印付きのラインで接続されます。この印はプログラムが**パラレル処理**されることを意味します。



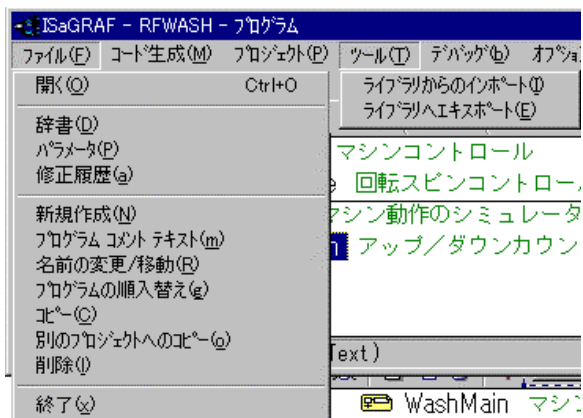
プログラミング言語

各プログラムは1つの言語で記述されます。プログラム新規作成時に選択した言語は変更できません。ただし、**FBD** ダイアグラムは **LD** シンボルを含むことができます。また、**LD** プログラムはファンクションブロックコールを含めることができます。グラフィック言語としては SFC (Sequential Function Chart), FC (Flow Chart), FBD (Functional Block Diagram), LD (Ladder Diagram)があります。テキスト言語として ST (Structured Text), IL (Instruction List)があります。SFC,FC 言語はシーケンシャルセクションでの親プログラム、チャイルドプログラム(FCではサブプログラム)として使われます。プログラムの言語はプログラム管理ウィンドウにプログラム名と共に以下のアイコンで表示されます。以下にプログラム言語を表わすアイコンを示します。:

	SFC	シーケンシャルファンクションチャート
	FC	フローチャート
	FBD	ファンクションブロックダイアグラム
	LD	ラダーダイアグラム (Quick LD エディタを使用)
	ST	ストラクチャードテキスト
	IL	インストラクションリスト

A.3.2 プログラムの管理:「ファイル」「ツール」メニューコマンド

「ファイル」と「ツール」メニューはプログラムを作成、更新、変更する以下のコマンドを含んでいます。



プログラムの新規作成

「ファイル」-「新規作成」コマンドにより、プログラムを指定したセクションに新規作成することができます。

新規プログラム

名前:

コメント:

言語: FBD : ファンクションブロック・ダイアグラム

スタイル: Begin : メインプログラム

OK Cancel(C)

まず、プログラム名を以下のルールに従って入力します。

- プログラム名は最大半角8文字
- 最初の文字は英文字(a-z)
- 以降の文字は英文字、数字、'_' のいずれか
- 大文字、小文字の区別なし(全角入力はできません)

次に、使用するプログラム言語選択します。

- SFC シーケンシャル・ファンクション・チャート
- FC フローチャート
- FBD ファンクションブロック・ダイアグラム
- LD ラダー・ダイアグラム
- ST ストラクチャード・テキスト(構造化テキスト)
- IL インストラクション・リスト(命令リスト)

最後に、プログラムのスタイル／セクションを選択します。

- Begin Beginセクションのトップレベル
- シーケンシャル シーケンシャルセクションのトップレベル
- End Endセクションのトップレベル
- ファンクション ファンクションセクション
- ファンクションブロック ファンクションブロックセクション
- ...の子 既存のプログラムのサブプログラム、チャイルドSF・FCプログラム

以上の5つのセクションから一つを選択すると、**Begin、End、シーケンシャル、ファンクション、ファンクションブロック**のいずれかのセクションのトップレベルにプログラムを位置づけることができます。"**...の子**"は、作成するプログラムがチャイルド**SFC**プログラム、**FC**サブプログラム、あるいはサブプログラムのときに選択します。この場合は親プログラム名を選択入力する必要があります。シーケンシャルプログラムのトップレベルはSFC／FC言語である必要があります。SFC／FC言語プログラムは Begin、End セクションのトップレベル、あるいは、そのサブプログラムには使えません。

各プログラムへのコメント入力

ISaGRAF ではプロジェクト中の各プログラムに対してコメントを入力することができます。このコメントはプログラムの横に小さめのフォントで表示されます。コメ

ントの入力には「ファイル」-「プログラムコメントテキスト」コマンドを選択します。ここで、新しくコメントの入力、又は、修正を行います。



プログラムの編集

「ファイル」メニューの「開く」コマンドにより、エディタウィンドウを開いてプログラムの内容を修正することができます。ISaGRAF はプログラム言語に応じたエディタを開きます。プログラムの編集は個別のウィンドウに表示されます。複数のエディタウィンドウを同時に開くことも可能です。プログラムが選択されている状態で**ENTER**キー入力、あるいはプログラム名のマウスによる**ダブルクリック**によりプログラムエディタウィンドウを開くことができます。



修正履歴ファイルの編集

「ファイル」-「修正履歴」コマンドにより、各プログラムに付属する修正履歴ファイルを編集したり、印刷することができます。編集中のプログラムを終了する際に修正履歴メモを記入するためのダイアログボックスが開きますので必要なメモを書き込むことができます。また、コード作成、ベリファイを行なった結果も修正履歴ファイルに自動的に書き込まれます。

なお、各エディタでの「オプション」-「修正履歴の更新」コマンドがセットされていない場合は、編集中のプログラム終了時にダイアログボックスは開きません。デフォルトはこのコマンドがセットされています。



変数辞書

「ファイル」-「辞書」コマンドにより、プロジェクトで使われる変数や定義ワードの辞書ファイルの編集が行なえます。変数にはその有効範囲によりグローバルとローカルがあり、ローカル変数は選択中のプログラムからのみアクセスできます。グローバル変数はプロジェクト全体からアクセスできます。定義ワードはプログラムで使われる文字列(数字)の置き換えを意味します。例えば、定数、キーワード、論理値状態を別の言葉(ワード)で置き換える際に使います。



ファンクション、サブプログラム、ファンクションブロックのパラメータ

「ファイル」-「パラメータ」コマンドにより、選択されているサブプログラムあるいはファンクション、ファンクションブロックセクションのプログラムの入力・出力パラメータを定義することができます。選択されているプログラムがBegin、Endセクションのトップレベルの親プログラムあるいはSFC・FC プログラムの場合は「パラメータ」コマンドは無効となります。ファンクション、ファンクションブロック、サブプログラム、最大32の入出力パラメータを持てます。ファンクションとサブプログラムは、1つの出力パラメータしか持てません。そしてそのパラメータ名はそのファンクションやサブプログラムの名前と同じでなければなりません。ファンクションブロックでは出力パラメータを複数持つことができます。



パラメータダイアログボックスの上部のウィンドウにはパラメータ(入力、出力パラメータ)が表示されています。表示の順番は呼び出しのルールに則って、1番目の入力パラメータから始まり、最後が出力パラメータです。下部のウィンドウには現在選択されているパラメータに関する詳細情報が表示されています。パラメータのデータタイプとして ISaGRAF がサポートするいかなるデータタイプも扱うことができます。

出力パラメータはパラメータリストの最後である必要があります。パラメータ名には以下のルールがあります。

- パラメータ名最大長は半角16文字
- 最初の文字は英文字(a-z)
- 以降の文字は英文字、数字、_ のいずれか
- 大文字、小文字の区別なし

1つのファンクションやファンクションブロック内で同一のパラメータ名は許されません。異なるファンクション間の同一のパラメータ名は使用できます。

“挿入”ボタンは新しいパラメータを選択中のパラメータの前に挿入します。“削除”ボタンは選択中のパラメータを削除します。“アレンジ”ボタンは自動でパラメータの順番を並び替えて出力パラメータが最後にくるようにします。

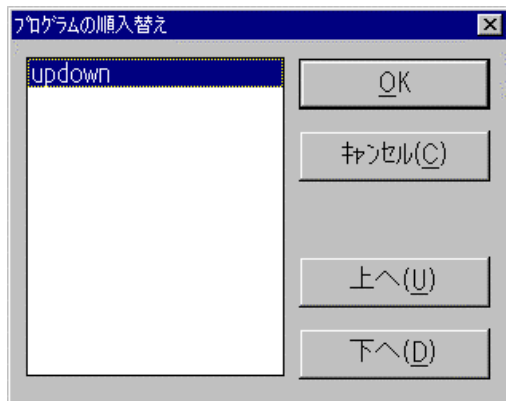
プロジェクト内でのプログラムの移動

「ファイル」-「名前の変更／移動」コマンドにより、プログラム名の変更やプログラムを別のセクションへ移動することができます。ただし、プログラム言語は変更することはできません。

このコマンドを実行すると、プログラムの新規作成時と同じダイアログボックスが開きます。選択されたプログラムの属性が選択表示されています。ここで、プログ

ラム名の変更、あるいは移動させたいセクション、親プログラムの選択を行なうことができます。

「ファイル」-「プログラム順入替え」コマンドにより、同一レベルのプログラム間での順番の入れ替えを行なうことができます。例えば、トップレベルのプログラムが選択されているときは同じセクションに属するトップレベル間でプログラムの順番が入れ替えができます。サブプログラムのレベルでは同一の親プログラムをもつプログラム間で順番の入れ替えができます。プログラムの順入替えダイアログボックスで“上へ”あるいは“下へ”ボタンを選択することでプログラムの順を入れ替えます。



プログラムのコピー

「ファイル」メニューの「コピー」コマンドにより、選択されているプログラムを別名のプログラムにコピーできます。

「コピー」コマンドを実行すると、プログラムの新規作成時のダイアログボックスが開きます。選択されたプログラムの属性が表示されます。ここで、コピー先のプログラム名の入力、あるいはコピー先のセクション、親プログラムの選択を行なうことができます。コピー先のプログラムが存在していない場合は指定された場所にプログラムがコピーされ、既にプログラムが存在している場合は上書きされます。プログラムのコピーにより、プログラムに加えて辞書ファイル内容もまとめてコピーされます。コピー先のプログラムの言語の種類は、オリジナルのものと同じでなければなりません。“OK”ボタンによりプログラムがコピーされます。“キャンセル”ボタンによりプログラムはコピーされずにダイアログボックスが閉じます。

「ファイル」メニューの「別のプロジェクトへのコピー」コマンドにより、選択されているプログラムを同一名のまま別のプロジェクトにコピーすることができます。チャイルドプログラム、サブプログラムもまとめてコピーされます。コピー先のプロジェクトで同一名のプログラム及びチャイルド・サブプログラムが存在していないことを確認しておく必要があります。このコマンドではプログラムは上書きされません。このコマンドにより、プログラムに加えて変数辞書ファイル(ローカル変数、ローカル定義ワードのみ)もあわせてコピーされます。



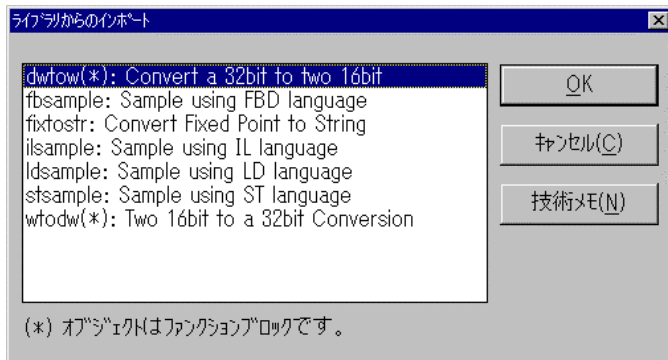
プログラムの削除

「ファイル」-「削除」コマンドにより、選択されているプログラムを削除することができます。チャイルドあるいはサブプログラムを持った親プログラムを選択して削除することはできません。この場合は最初にチャイルド・サブプログラムを削除した後に親プログラムを削除する必要があります。プログラムの削除の際に辞書ファイルで定義されているローカルな定義内容もあわせて削除されます。



ライブラリからのファンクション、ファンクションブロックのインポート

「ツール」-「ライブラリからのインポート」コマンドによりIEC言語で書かれたライブラリのファンクション又はファンクションブロックを現在のプロジェクトに取り込むことができます。ライブラリ内で宣言されているローカル変数、定義も一緒にインポートされます。ライブラリ内で既にペリファイ、コンパイル済みの場合はオブジェクトコードもあわせてコピーされます。この場合は開いているプロジェクト内でのコンパイルは省略されます。このコマンドによりIEC言語で書かれたライブラリから**ファンクション、ファンクションブロックセクション**のトップレベルにコピーすることができます。インポートされた関数はプロジェクトの階層構造の中の所定のセクションに「**名前の変更／移動**」コマンドにより移すことができます。既に存在するファンクションやファンクションブロックへの上書きを避けるために、インポートされるライブラリは名前の変更が必要な場合があります。この場合、ファンクションの戻り値の変数名も忘れずにファンクション名に合わせておく必要があります。

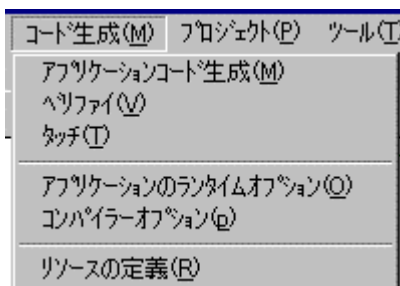


☐ ファンクション、ファンクションブロックのライブラリへのエクスポート

「ツール」-「ライブラリへのエクスポート」コマンドにより、現在オープン中のプロジェクトのファンクション、又は、ファンクションブロックを対応したライブラリへコピーすることができます。ファンクション内で宣言されたローカル変数、定義ワードは一緒に移されます。エクスポートされたファンクション、ファンクションブロックはライブラリ管理ウィンドウ上でコンパイルする必要があります。エクスポートされるファンクション、ファンクションブロックは、グローバル変数を使うことはできません。

A.3.3 「コード生成」メニューコマンド

「コード生成」メニューにはプロジェクトの実行コードを生成するためのコマンドやそのオプションコマンドが含まれています。詳細は「コード生成の使い方」の章を参照願います。



アプリケーションコード生成

「コード生成」-「アプリケーションコード生成」コマンドにより、設定済みのコンパイラオプションに従ってプロジェクトコードが生成されます。プロジェクトコード生成時には文法チェックがなされていないコードに関してはベリファイもされます。インクリメンタルコンパイルのため既にコンパイルされてるコードについての再コンパイルは行いません。



プログラムのベリファイ

「コード生成」-「ベリファイ」コマンドにより、プログラム管理ウィンドウで選択されているプログラムの文法チェックベリファイができます。もし、エラーが検出されなかった場合は、コードジェネレーション時には内容や辞書に変化がない限り再度プログラムベリファイがされることはありません。

☐ プログラム修正の仮操作

「コード生成」-「タッチ」コマンドにより各プログラムに修正が加わったことをシミュレートし、次のコード生成時には全プログラムがベリファイ、コンパイルされます。



アプリケーションのランタイムオプション

「コード生成」-「アプリケーションのランタイムオプション」コマンドにより、中間コード生成オプション及びランタイムパラメータの設定を行なうことができます。以下に個々のパラメータの解説を行ないます。詳細は「コード生成の使い方」を参照願います。

● サイクルタイム

サイクルタイムとは2つのランタイムサイクル間の時間間隔設定値(周期)のことを指し、単位にミリセカンド(ms)を用います。ランタイムサイクルがこの設定値(許容値)よりも長くなった場合はエラー(オーバーフローエラー)を出します。マルチタスク環境ではこの設定値を有効に活用することができます。即ち、設定されたサイクルタイムよりランタイムサイクルが短い場合は残りの時間はその他のタスクに処理時間をまわすことができます。(その間は ISaGRAF ターゲットはCPUに負荷をかけません。)

“**サイクルのトリガ**”オプションがセットされていない場合は、ランタイムサイクルは連続的に繰り返されます。即ち、他のタスクに処理時間をまわすことができません。このオプションがセットされないモードはターゲットがシングルタスク環境の場合を前提としています。

- **ランタイムエラー**

“ランタイムエラー” オプションがセットされているとき、デバッガーがランタイム中のエラーを検出します。このオプションは新しく作られたプロジェクトをデバッグする際に有効です。デバッグが終わればこのオプションをはずすことで不必要なコミュニケーションのための処理がなくなりサイクル時間が短縮されます。(ただし、コントロールカーネルタスクと通信タスクが同一タスクの場合)ランタイムエラーオプションがセットされているときは“**保管エラー数**”がセットできます。この値はエラーキュー内に保管しているエラー数を示します。

- **スタートモード**

“スタートモード”パラメータによりランタイムの実行モードを設定します。“**サイクル**”モードにセットされている場合は、プログラムがダウンロードされた後にデバッガーからのサイクルの実行コマンドを待ちます。1サイクル実行コマンドがくると1サイクルのみ実行して再び次の実行コマンドを待ちます。“**リアルタイム**”モードにセットされている場合(デフォルト)はプログラムがダウンロードされた後に即、実行状態になります。

- **保持変数**

“**保持変数**”オプションがセットされている場合は、指定された保持変数がバックアップメモリに常に保持されます。プログラムの再起動時に保持されている変数内容を初期値にすることができます。メモリ領域の指定はターゲットハードウェアにより指定方法が異なりますのでターゲットのハードウェアマニュアルを参照願います。

□ **コンパイラオプション**

「**コード生成**」-「**コンパイラオプション**」コマンドにより、ISaGRAF のコードジェネレータがターゲット実行コードの種類の選択、及び最適化をすることができます。詳細は「コード生成の使い方」を参照願います。

□ **リソース定義**

「**コード生成**」-「**リソースの定義**」コマンドにより、ターゲット実行コードと一緒に使えるデータ(あるいは参照するファイル)をリソースファイルとして定義しダウンロードできるようになります。詳細なリソース定義ファイルの設定フォーマットなどは「コード生成の使い方」を参照願います。

リソース(ダウンロードコードに添付されるデータ)の定義で、相対パス名の指定が可能になりました。入力ファイルの場所を、プロジェクトディレクトリからの相対パスとして、“.”を使用して指定することが可能です。これは、“FROM”ステートメント中で、テキストファイル、バイナリファイルいずれのリソースの指定にでも使用できます。

A.3.4 その他の ISaGRAF ツール:「プロジェクト」メニューコマンド

「プロジェクト」メニューには ISaGRAF ツールを実行するためのコマンドが含まれています。詳細は各ツールの使い方を参照願います。



I/O変数の接続

「ツール」-「I/O接続」コマンドにより、I/O接続エディタを開くことができます。ここでは、宣言されているI/O変数を対応したI/Oボードとのマッピングを行います。論理的な変数を物理的に割り当てることになります。



クロスリファレンス

「プロジェクト」-「クロスリファレンス」コマンドにより、プロジェクトのクロスリファレンスエディタが起動し、リファレンスの表示や、リストの印刷を行うことができます。クロスリファレンスでプログラムのソースコード中の全変数の使われている場所を見ることができます。特定の変数の使われている場所の検索に有効に使えます。



プロジェクトの記述メモ

「プロジェクト」-「プロジェクトの記述」コマンドにより、プロジェクトの記述用テキストの編集を行うことができます。なお、このコマンドはプロジェクト管理ウィンドウからも起動することができます。



プロジェクトドキュメント印刷

「プロジェクト」-「プロジェクトドキュメントの印刷」コマンドにより、開いているプロジェクトに関するドキュメントを印刷できます。このドキュメントにはプログラム、変数、パラメータなどが含まれます。ユーザは必要な印刷項目を選択してカスタマイズされたドキュメントを印刷することができます。なお、「プロジェクト管理」にも同じコマンドがあります。



修正履歴

「コード生成」-「修正履歴」コマンドによりプロジェクトのこれまでの修正履歴のダイアログボックスが開きます。詳細は「プロジェクト管理」を参照願います。

A.3.5 ツールメニューへのユーザコマンドの追加

ISaGRAF では、「ツール」メニューにユーザコマンドを追加することができます。`\\ISAWIN\\COM\\ISA.MNU` テキストファイルを編集することにより、ユーザコマンドが追加できます。最大10個のコマンドを追加できます。コマンドに対するコメント行は“;”で開始します。各コマンドは以下に示す2行のテキストラインで構成されます。以下の文法に従います。

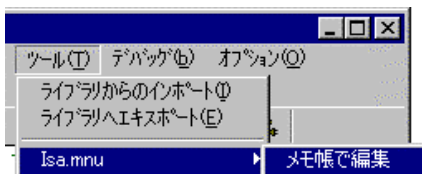
M=メニューに表示される文字列
C=コマンドライン

Mで記述した文字が、「ツール」メニューに表示されます。コマンドラインにはDOSやWindowsの実行モジュールを引数を含わせて記述できます。扱える引数としては“%A”は開かれているプロジェクト名、“%P”は選択中のプログラム名です。

以下の例ではメモ帳で選択されたプログラムを編集する場合のコマンドラインとなります。

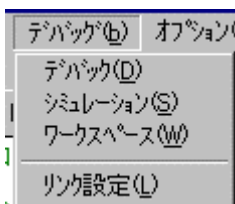
```
M=メモ帳で編集  
C=Notepad.exe \isawin\apl\%A\%P.lsf
```

この様に、設定した場合ツールバーは下記のようにになります。



A.3.6 アプリケーションのデバッグ、シミュレーション:「デバッグ」メニューコマンド

「デバッグ」メニューには ISaGRAF アプリケーションのシミュレーション、オンラインデバッグに必要なコマンドが含まれています。



シミュレーション

シミュレーションは、プログラムが実際動作するターゲットマシンなしでプログラムをチェックすることができる便利なツールです。「デバッグ」-「シミュレーション」コマンドにより、デバッグウィンドウがシミュレーションモードで起動されます。これと同時にカーネルシミュレータと呼ばれるウィンドウが起動されます。カーネルシミュレーションウィンドウとはアプリケーションで入出力I/O変数が使われていて、これらがI/O接続がなされている場合の仮想I/Oパネルを指します。シミュレーションを行う前にはシミュレーション用のプロジェクトコードが生成されている必要があります。(コード生成のコンパイラオプションでシミュレーション用コードの選択が必要)

デバッグウィンドウが開かれるときにプログラム管理ウィンドウは一旦、閉じられます。シミュレーションウィンドウが開いた後に、プログラム管理ウィンドウは再びデバッグモードで開きます。なお、オープン中のプロジェクト内のプログラムエディタ、コード生成ウィンドウ、I/O接続ウィンドウなどが開いているときはデバッグウィンドウは開きません。デバッグの前にはこれらを全て閉じる必要があります。このコマンドはプログラムエディタから起動することもできます。



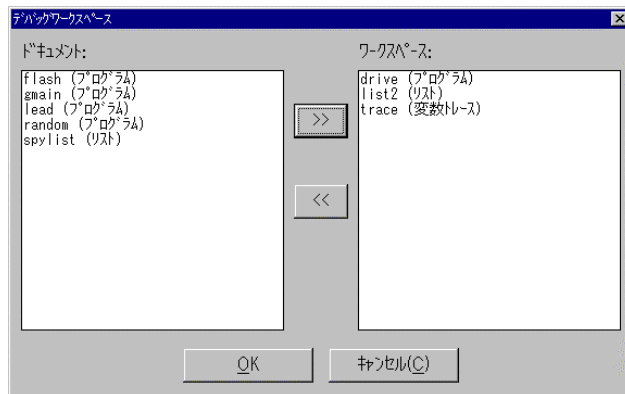
オンラインデバッグ

「デバッグ」-「デバッグ」コマンドにより、デバッガーメインウィンドウを起動します。オンラインデバッグ前にはターゲット用のアプリケーションコードが生成されている必要があります。デバッグウィンドウが開かれるときにプログラム管理ウィンドウは一旦、閉じられます。ワークベンチのデバッガーがターゲットアプリケーションと通信ができたときに再びデバッグモードで開きます。なお、オープン中のプロジェクト内のプログラムエディタ、コードジェネレーションウィンドウ、I/O接続ウィンドウなどが開いているときはデバッグウィンドウは開きません。デバッグの前にはこれらを全て閉じる必要があります。このコマンドもプログラムエディタから起動することができます。

シミュレーションとの違いは、カーネルシミュレーションウィンドウが開かないことだけです。これ以外は同じシミュレーション機能を持ちます。

■ デバッグ用ワークスペース設定

「デバッグ」-「ワークスペース」メニューコマンドにより、初期のデバッグワークスペースに表示されるドキュメントのリストを設定することができます。ドキュメントにはプログラム、SpotLight グラフィック、変数リスト、以前のバージョンで使われていたグラフィック画面、タイムチャートなどが含まれます。ワークスペースに設定されたこれらのドキュメントはシミュレーションやオンラインデバッグを行うときに自動的に開かれます。

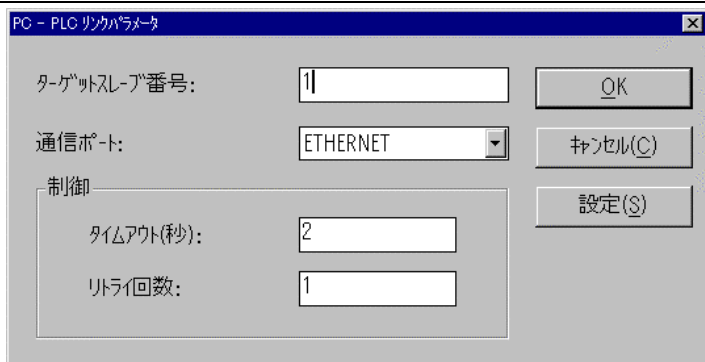


このダイアログボックスでは、左側のボックスには存在しているすべてのドキュメントが表示されます。これらの中から自動で表示したいドキュメントを右側のワークスペースボックスに“>>”、“<<”ボタンを選択して移動させます。各プロジェクトには1つのワークスペースを持つことができます。



通信リンク設定

「デバッグ」-「リンク設定」コマンドにより、ISaGRAF ワークベンチとターゲット間の通信パラメータの設定を行ないます。リンクパラメータダイアログボックスで以下の通信パラメータを設定します。これらの設定内容は接続される ISaGRAF ターゲット側の設定内容と一致している必要があります。



PC - PLC リンクパラメータ

ターゲットスレーブ番号: 11

通信ポート: ETHERNET

制御

タイムアウト(秒): 2

リトライ回数: 1

OK

キャンセル(C)

設定(S)

ターゲットスレーブNo. は同時に実行中の ISaGRAF ターゲットのID番号になります。1～255の値を入力できます。ターゲットスレーブNo. の設定に関してはターゲットマニュアルを参照願います。

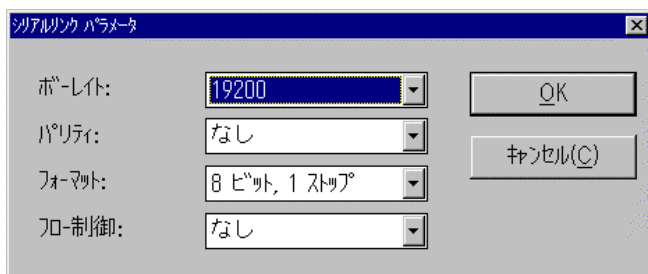
通信ポートはワークベンチデバッガーと ISaGRAF ターゲットとの通信ポートを設定します。標準としてCOM1～COM4、Ethernetがあります。EthernetはWinsock Ver1. 1を使ったTIC/IP通信となります。Ethernet通信はSunPC-NFS Ver5. 0で動作確認済みです。

タイムアウトはデバッガーとターゲット間の通信のタイムアウト条件を設定します。タイムアウト時間はデバッガーからの質問に対するレスポンスまでの間隔を指します。**単位は秒**です。

リトライはデバッガーがターゲット間と通信を行なう際の通信エラーを検出する前の通信トライの回数です。

シリアルリンク通信パラメータの設定

通信ポートにCOM1～4が選択されたときに**“設定”**ボタンを選択すると、シリアルリンク通信パラメータの設定ダイアログボックスが開きます。ここでは、以下のパラメータを設定することができます。



シリアルリンク パラメータ

ボーレート: 19200

パリティ: なし

フォーマット: 8 ビット, 1 ストップ

フロー制御: なし

OK

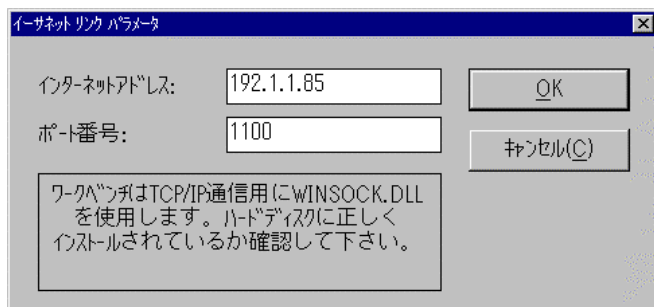
キャンセル(C)

- **ボーレート** デバッガーとターゲット間の通信速度(ビット/秒)を示します。最大19200baudが設定可能です。
- **パリティ** 通信パリティで偶数、奇数、なしのいずれかが選択可能です。
- **フォーマット** ... データビット数とストップビット数を指定します。データビット数は7、8のいずれかを、ストップビット数は1、2のいずれかの選択が可能です。

- **フロー制御** **フロー制御**の選択を行ないます。ISaGRAF ワークベンチはハードウェアハンドシェイクを可能にするためにCTS、DSR信号を制御することも可能です。

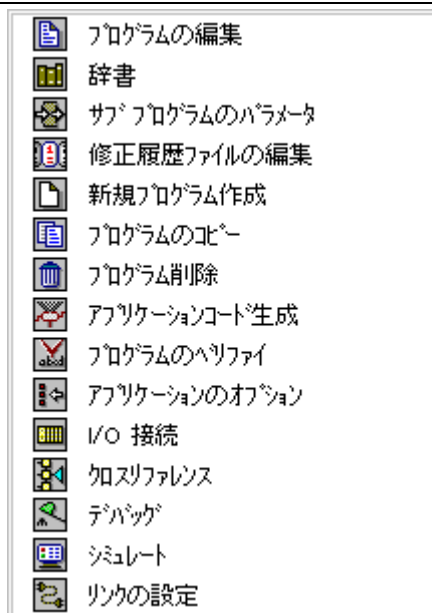
Ethernet/パラメータ設定

通信ポートにEthernetが選択されたときは、“**設定**”ボタンによりTCP/IP通信のためのインターネットアドレス及びインターネットポート選択のダイアログボックスが開きます。これらのフィールドはソケットインタフェースにより定義済みの標準フォーマットです。ワークベンチはTCP/IP通信にWinsock. dllのver1. 1を使います。ワークベンチの環境にはこのソケットが正しくインストールされている必要があります。デフォルトのポート番号は**1100**です。



A.3.7 ツールバーアイコン

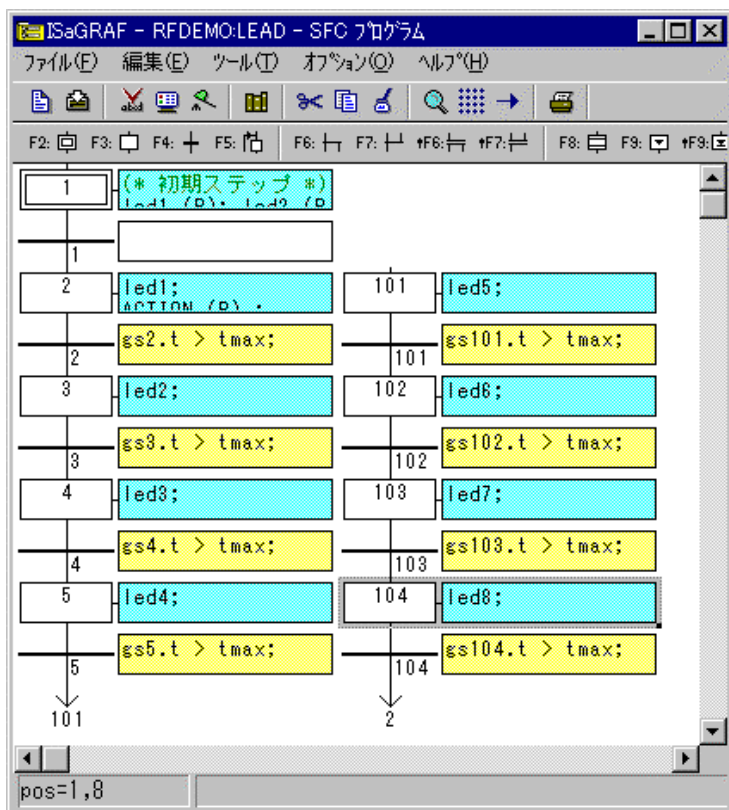
以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.4 SFC エディタの使い方



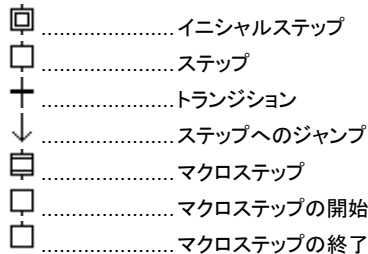
SFC言語はシーケンシャルなプロセスオペレーションを記述するときに使います。プロセスのステップを表現する部分と、アクティブなステップを変更する条件となるトランジションを表現するグラフィックシンボルから成り立っています。SFC プログラムは、ISaGRAF の SFC グラフィックエディタによって記述します。SFC言語はIEC 61131-3標準のコアとなる部分です。その他の言語はステップ内のアクションを記述するため、あるいは、トランジションの論理的条件を記述するために使われる場合があります。SFC言語エディタは、チャートを作成するグラフィックエディタとテキスト言語によるプログラミングのどちらも扱えます。よって、チャートの作成からアクション部分、トランジション条件部分のプログラミングまで連続的に行なうことができます。



A.4.1 SFC 言語メイントピック

SFCは、シーケンシャルなプロセスを表現するには最適な言語です。SFC言語はプロセスのサイクルを連続した**ステップ**とし、それを**トランジション**によって区切った構成になっています。詳細は ISaGRAF 言語リファレンスマニュアルのSFC言語を参照して下さい。

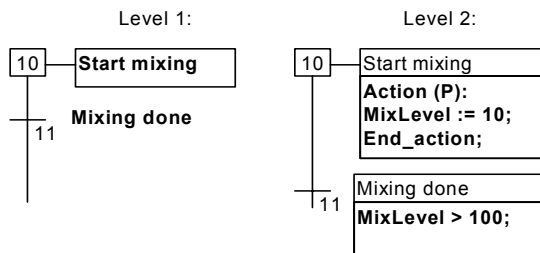
SFCのコンポーネント間は**方向を持ったライン**で接続されます。デフォルトのラインの方向は**上から下へ(↓)**です。下記のシンボルが SFC の基本的な構成要素です：



SFC言語によるプログラムは2つのレベルに分かれています。

レベル1 ではグラフィックチャートを表示します。ステップやトランジションにはリファレンスNo. とコメントがアタッチされています。

レベル2では**ST言語**(トランジション条件の記述には**LD言語**も可能)又は**IL言語**によるプログラミングでステップ内のアクションあるいはトランジションにアタッチされている条件の記述を行ないます。アクションやトランジションは他の言語(LD、FBD、ST、IL)で書かれた**サブプログラム**を参照することもできます。レベル1、レベル2のプログラミングの例を下に示します。

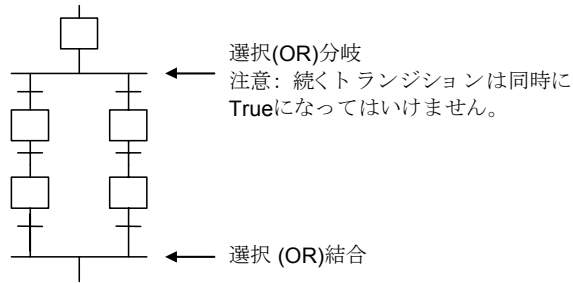


レベル2のプログラムはテキストエディタで入力します。ステップ内のアクションのプログラミングにはST、IL言語を使います。トランジション条件の記述にはST、ILの他にLD言語(QuickLD エディタを使用)が使えます。

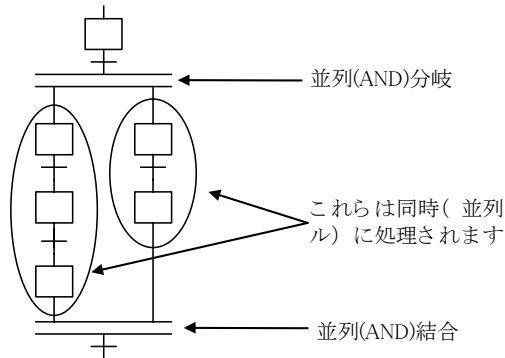
分岐と結合

分岐と結合は複数のステップやトランジションをつなぐマルチプルリンクです。

選択(OR)分岐ではいくつかのパスの中からどれかひとつだけが選択されて実行されます。

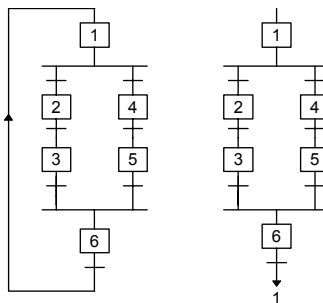


並列(AND)分岐は並列処理を意味します。



ステップへのジャンプ

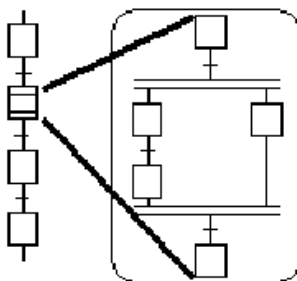
SFCエディタではコンポーネント間のリンクは必ず上から下(↓)方向になります。ただし、ステップへのジャンプのを使って上方向(↑)へリンクすることができます。下の図は同じ結果となります。



ステップからトランジションへのジャンプの記述は禁止されています。また、複数のステップから1つのトランジションへのジャンプは、並列 (AND) 結合によって明示的に表現することができます。

マクロステップ

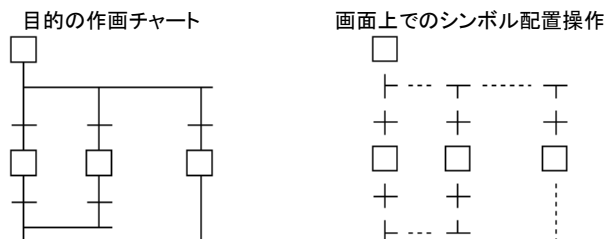
マクロステップは**独立に存在しているステップとトランジションのグループ**を代表する**ユニークなシンボル**です。マクロステップは開始ステップで始まり、終了ステップで終わります。



マクロステップの詳細は同一のSFCチャート内に書かれている必要があります。マクロステップの開始ステップとマクロステップシンボルは同じ**リファレンス番号**でなければいけません。マクロステップは更に別のマクロステップシンボルを含むこともできます。

A.4.2 SFC チャートの入力

SFCチャートを描くときはキーコンポーネントのみを配置するだけでよく、水平・垂直の接続ラインは自動的に引かれます。



SFC コンポーネントをチャート上に配置するためには、まずカーソルを目的の場所に移動してからエディタのツールバーより目的のコンポーネントを選択します。シンボルはカーソルのある位置に挿入されます。以下のファンクションキー入力を使うことも可能です。



イニシャルステップの挿入



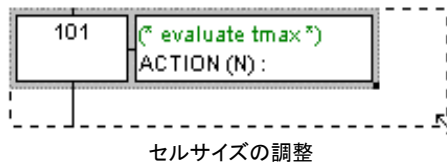
ステップの挿入

F4:	トランジションの挿入
F5:	ステップへのジャンプの挿入
F6: F7:	選択分岐と結合の挿入／選択ブランチの追加
*F6: *F7:	並列分岐と結合の挿入／並列 ブランチの追加
F8:	マクロステップの挿入
F9: *F9:	マクロステップの開始・終了ステップの挿入

(シンボルは SHIFT キーとの組み合わせを意味します)

編集用のグリッドは配置されるシンボルの場所を示します。グリッド表示があると最初のSFCチャートやサブパーツのレイアウト構成が容易です。「オプション」-「レイアウト」コマンドによりグリッドの表示／非表示が可能です。

SFC エディタは常にカレントのセルの位置が座標の形で表示されます。選択されているセルの場所はグレー表示されます。選択されているボックスの右のコーナーをマウスでドラッグすることによりセルサイズを自由に変更することができます。X/Y 比率を変更することも可能です。



分岐、結合の挿入

分岐と結合は必ず左から右へとコンポーネントを配置していきます。必ず左端の分岐から最初に配置していきます。シンボルの配置は、まず、チャート内に配置する場所を選択してから、ツールバーより配置するシンボルを選択します。

F6: F7: 選択分岐と結合の挿入／選択ブランチの追加
*F6: *F7: 並列分岐と結合の挿入／並列ブランチの追加

分岐、結合ブランチの挿入

各分岐／結合のブランチを追加するには、以下のシンボルをツールバーから選択します。左側のブランチのスタイル（選択、並列）にあわせて順番に右側に同じスタイルの新しいコンポーネントを配置します。左側にコンポーネントが配置されていないと、右側に新しいコンポーネントを追加することはできません。

F6: F7: 選択分岐と結合の挿入／選択ブランチの追加
*F6: *F7: 並列分岐と結合の挿入／並列ブランチの追加



マクロステップの挿入

このツールバーアイコンにより、メインチャートにマクロステップを挿入することができます。マクロステップの詳細はメインチャートと同じSFCチャート内に記述しなければなりません。



マクロステップの内容

マクロステップは親チャートと同一のSFCチャート内に書かれる必要があります。マクロステップは**開始ステップ**ではじまり、**終了ステップ**で終わります。マクロの開始ステップと親のマクロステップは**同じリファレンス番号**でなければなりません。マクロステップを書くためのツールバーボタンを以下に示します。



マクロステップの開始ステップの挿入



マクロステップの終了ステップの挿入

A.4.3 SFC チャートの編集

マウスあるいはカーソルキーにより、チャート内のセルを選択します。選択された場所はグレー色となります。選択された場所に対しては「**編集**」メニューのコマンド（カット、ペーストなど）が使えます。

編集(E)	ツール(T)	オプション(O)	ヘルプ(H)
元に戻す(U)		Ctrl+Z	
切り取り(I)		Ctrl+X	
コピー(C)		Ctrl+C	
貼り付け(P)		Ctrl+V	
クリア(U)			
検索(F)			
置換(R)			
ジャンプ(G)		Ctrl+G	
番号取り直し(n)			
変数選択(U)		Ctrl+D	
レベル 2 の編集(E)		Return	
セパレートウィンドウでレベル2の編集(w)		Ctrl+Return	
プログラムをメモファイルとしてコピー(d)			



切り取り／コピー／貼り付け／削除コマンド



シンボルの移動



ステップとトランジションの番号取り直し



指定ステップ、トランジションへのジャンプ



テキストの検索／置換



切り取り／コピー／貼り付け／削除コマンド

以下の編集コマンドが選択されたシンボルに使えます。

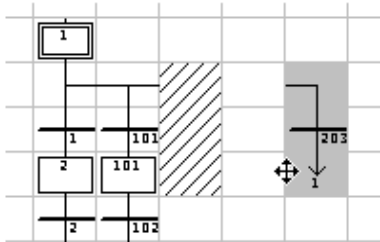
- **切り取り** 選択された長方形領域のクリップボードへの移動
- **コピー** 選択された長方形領域のクリップボードへのコピー
- **貼り付け** 選択された長方形領域にクリップボードの内容を貼り付ける
- **削除** 選択された長方形領域の削除

編集／貼り付けコマンドは、クリップボードの内容を画面にコピーしますが、このとき、ステップ/トランジションと一緒にレベル 2 プログラムの内容もコピーされます。また、コピーしたチャートを他の SFC プログラムへペーストすることもできます。コピーした要素は、選択されたセルの位置に挿入されます。



シンボルの移動

SFC チャートで選択されているシンボル／エレメントはマウスのドラッグ操作で別の場所へ移動させることができます。ドラッグ中は最初の場所は斜線表示されます。



シンボルの移動先は空白でなければなりません。SFC シンボルを移動中での挿入は行えません。



ステップとトランジションの番号取り直し

各ステップ、トランジションにはリファレンス番号が付けられます。この番号は「**編集**」-「**番号取り直し**」コマンドにより、編集中の SFC チャートのリファレンス番号を新しく自動で連続した番号で取り直すことができます。この際にステップへのジャンプのリファレンス番号やマクロステップへのジャンプなどの番号も自動で更新されます。



指定ステップ、トランジションへのジャンプ

SFC チャート編集中に「**編集**」-「**ジャンプ**」コマンドにより、編集中の SFC 内の指定されたステップやトランジションへジャンプできます。この際、ジャンプ先のステップやトランジションが見えるようにウィンドウのスクロールが自動で行われます。



テキストの検索／置換

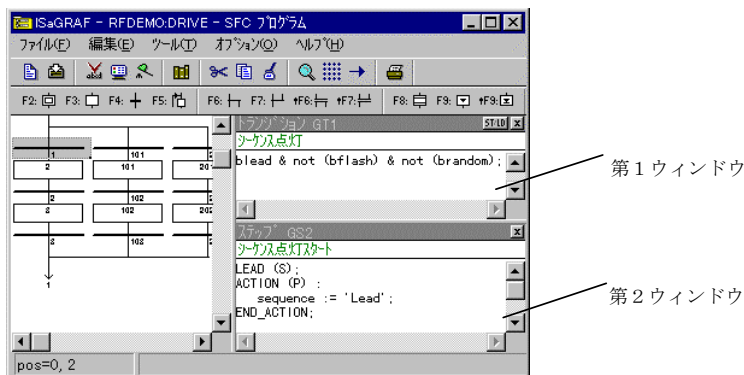
「**編集**」-「**検索／置換**」コマンドにより、編集中の SFC チャート内のステップ、トランジション内に記述されている全プログラムを対象にテキストの検索と置換が行えます。検索／置換のダイアログボックスで検索するテキストを入力します。もし、検索テキストが見つかった場合は、レベル 2 プログラムも同時に開かれます。

A.4.4 レベル2プログラムの入力

レベル2でのプログラムを入力する際にはSFCのステップ、トランジションをダブルクリックします。レベル2プログラミングウィンドウが SFC の右側に開きます。SFC プログラムとレベル2プログラムウィンドウ間にはセパレーションラインがあり、自由にずらすことができます。

レベル2プログラムを最大2つまで同時に開くことができます。以下のキーボード入力コマンドやマウス操作に従います。

	キーボード	マウス操作	「編集」メニューコマンド
第1 ウィンドウ	Enter	ダブルクリック	レベル2の編集
第2 ウィンドウ	Ctrl+Enter	Ctrl + ダブルクリッ ク	セパレートウィンドウでレベル2の編 集



2つのレベル2ウィンドウが開いているときは、これらの間のセパレーションの位置は調整できます。レベル2のタイトルバーの右上のボタンでレベル2ウィンドウを閉じることができます。

レベル2のプログラムにはデフォルトでST言語が使われます。ただし、トランジションのレベル2プログラムに関してはLD言語 (Quick LD エディタを使用) が使えます。レベル2ウィンドウのタイトルバーにある「ST/LD」切り替えボタンによりアクティブな言語を切り替えることができます。この切り替えはレベル2ウィンドウが空白状態のときのみ有効となります。



レベル2ウィンドウの上部には一行の編集可能なテキスト行があります。このテキストがステップやトランジションのコメントとなります。このコメントは「ジャンプ」コマンドや SFC ドキュメントの印刷の時のコメントとして使われます。



「オプション」-「リフレッシュ」メニューコマンドにより、レベル2ウィンドウが開いている時にレベル2での修正内容をメインの SFC チャートに反映させることができます。



変数名の挿入

テキストエディタでプログラミングを行っている場合にこのボタンを選択すると、プロジェクトで宣言されている変数辞書が開かれ、変数を選択すると、現在のカーソル位置に変数名を挿入できます。Quick LDでプログラムを行っている場合は、選択されているシンボルに対して変数名、ファンクションブロック名、パラメータを選択できます。



ステップにパルス(P)アクションの挿入

ステップのレベル2プログラミングを行っている場合に、このボタンを選択することで現在のカーソル位置にパルスアクションブロックを挿入することができます。以下にフォーマットを示します。

```

Action (P) :
    ST statement;
    ...
End_Action;

```

パルスアクションはステップが活性化状態になったときのみ実行されます。詳細は言語リファレンスを参照願います。



ステップにノンストアード(N)アクションの挿入

ステップのレベル2プログラミングを行っている場合に、このボタンを選択すると、カーソルの位置にノンストアードアクションブロックを挿入することができます。以下にフォーマットを示します。

```

Action (N) :
    ST statement;
    ...
End_Action;

```

ノンストアードアクションはステップが活性化状態の時、毎サイクル実行されます。詳細は言語リファレンスを参照願います。



P0、P1 アクションクオリファイア

ISaGRAF は新しく、**P0**、**P1** アクションクオリファイアをサポートしています。ステップのレベル2プログラミングを行っている場合に、このボタンを選択することで現在のカーソル位置に **P0**、**P1** アクションブロックを挿入できます。以下にこれらのフォーマットを示します。

Action (P0) : ST statement; ... End_Action;	Action (P1) : ST statement; ... End_Action;
--	--

P1 アクションはステップが活性化状態になったときのみ（パルスアクションと同様に）、P0 アクションはステップが非活性化状態になったときのみの実行されます。詳細は言語リファレンスを参照願います。

■ ブールアクション

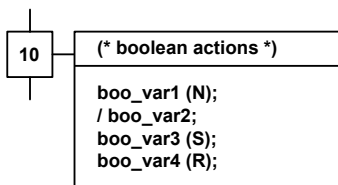
ブールアクションは現在の**ステップの状態**を内部・外部のブール型変数に割り当てます。以下に基本的なブールアクションの文法を示します。（< >は入力しません）

<ブール型変数名> (N) ; ステップのアクティビティ状態を変数に割り当て
 <ブール型変数名> ; 同上 (N 属性はオプション)
 / <ブール型変数名> ; ステップのアクティビティ状態の反転を変数に割り当て

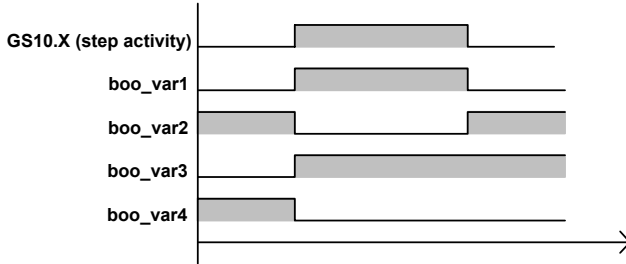
これらの他にも、ステップ状態がアクティブになったとき変数の値をセットあるいはリセットする場合があります。以下にこれらの文法を示します。

<ブール型変数名> (S) ; ステップの状態が TRUE になったとき変数の値を TRUE にする。
 <ブール型変数名> (R) ; ステップの状態が TRUE になったとき変数の値を FALSE にする。

例を示します。



以下の振る舞いを示します。



SFC アクション

SFCアクションではチャイルドSFCのシーケンスをコントロールします。即ち、ステップの状態に応じて起動、停止、Killを行ないます。SFCアクションには**N**(Non store)、**S**(Set)、**R**(Reset)識別子がつきます。以下に文法を示します。

- <<チャイルドプログラム> (N) ; ステップがアクティブになったときにチャイルドシーケンスを起動し、非アクティブになったときにチャイルドシーケンスを停止(Kill)させます。
- <チャイルドプログラム> ; 同上 (N 識別子はオプション)
- <チャイルドプログラム> (S) ; ステップがアクティブになったときにチャイルドシーケンスを起動し、非アクティブになったときにはチャイルドシーケンスに対して何もしません。
- <チャイルドプログラム> (R) ; ステップがアクティブになったときにチャイルドシーケンスを停止(kills)し、非アクティブになったときは何もしません。

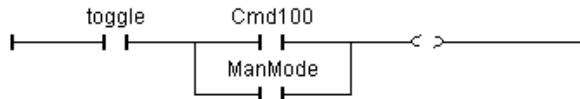
SFCアクションで指定されるプログラムは、実際に存在している、現在編集中のプログラムの**チャイルドSFCプログラム**である必要があります。

STでプログラムされたトランジション

トランジションのレベル2プログラムはブール表現を使います。これはブール型の条件をSTの文法に従って入力することです。最後に`;`を付けます。

Quick LDでプログラムされたトランジション

トランジションのレベル2プログラムにはQuick LDを使うこともできます。この場合、プログラムのラングは1行のみで、トランジションの条件判断の結果を表す1つのコイルのみ記述できます。トランジションの状態を表すコイルには変数名を割り付ける必要はありません。以下に例を示します。.



Quick LDエディタでプログラムする場合、キーボードの入力のみで行うことができます。カーソルキーでカーソルを移動させ、シンボルを挿入する際には以下のファンクションキーを選択します。

F2: 接点を後に挿入

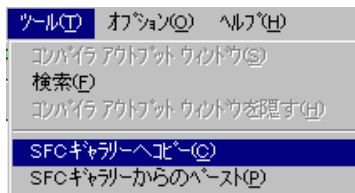
F3: 接点を前に挿入
F4: 接点を並列に挿入
F6: ファンクションブロックを後に挿入
F7: ファンクションブロックを前に挿入
F8: ファンクションブロックを並列に挿入
(マウスを使って画面上をクリックすることも可能です。)

接点やファンクションブロック、入出力パラメータをカーソルで選択して ENTER キーを選択すると、変数を選択するダイアログボックスが開きます。目的の変数名、ファンクションブロック名にカーソルを合わせて ENTER キーを選択します。ダブルクリックによっても選択ができます。

カーソルを接点やコイルに合わせて、Ctrl キー + スペースキーを選択することで接点、コイルのタイプ(a接点、b接点)を変更することができます。詳細は、「Quick LDの使い方」を参照願います。

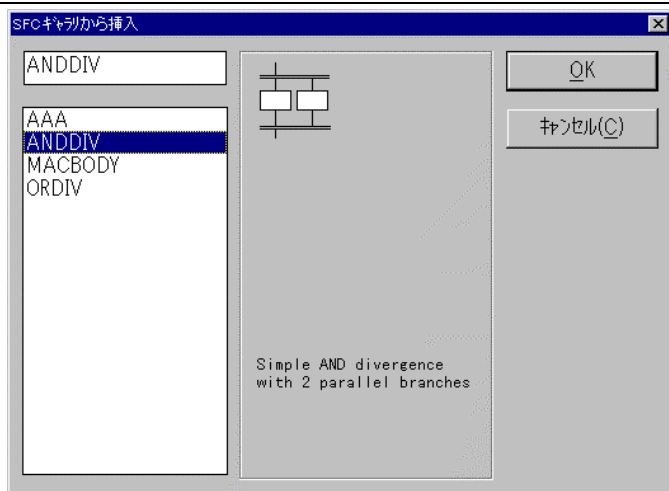
A.4.5 SFC ギャラリーの使い方

SFC エディタでは SFC ギャラリーを扱うことができます。SFC ギャラリーとは各種 SFC の構造を集めたもので、SFC チャート中に挿入することができます。SFC ギャラリーにはオプションでレベル2プログラムの内容を含めることができます。以下に「ツール」メニューコマンドを示します。



SFC ギャラリーへコピー
SFC ギャラリーからのペースト

選択されたエレメントを SFC ギャラリーへコピー
SFC ギャラリー中のエレメントを SFC チャート上にペースト



SFC ギャラリーへエレメントをコピーする際(即ち、SFC ギャラリーエレメントの新規作成の際)に、レベル2プログラムを含めるかどうかの選択が行えます。

A.4.6 ツールバーアイコン

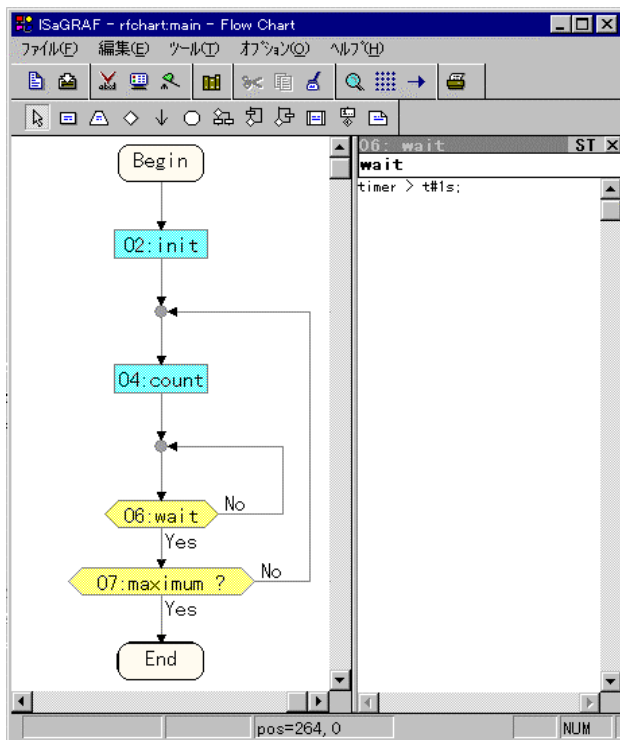
以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.5 フローチャートエディタの使い方



ISaGRAF フローチャートエディタはアクション、テストブロックを ST、IL、QuickLD 言語でプログラムができます。フローチャートは判断を記述するダイアグラムですが、シーケンシャル操作も記述することができます。LD や IL では無限ループとなるような場合でも、フローチャートではターゲットでの処理をブロックすることなく戻りのジャンプの記述も可能となります。



A.5.1 フローチャート(FC)言語の基本要素

フローチャート(FC)はシーケンシャルな処理を記述するグラフィカル言語です。フローチャートプログラムは**アクション**と**テスト**から成り立っています。アクションとテストをデータの流れを示す**リンク**で結合します。以下にフローチャート言語の基本要素を示します。



FC チャートの開始: "begin" シンボルはフローチャートプログラムの最初に必ず存在します。このシンボルはチャート内に1つだけ存在して、削除することはできません。このシンボルはフローチャートが実行される時の最初の状態を持ちます。



FC チャートの終了: "end" シンボルはフローチャートプログラムの最後に必ず存在します。このシンボルはチャート内に1つだけ存在して、削除することはできません。"End"シンボルに何も接続されていないような状態(ループ状態)が可能です。このシンボルが実行された時は、フローチャートの最後の状態となります。



FC フローリンク(リンク): フローリンクはチャート内の2つのポイントを接続するラインです。リンクには流れの方向性を示す矢印がついています。2つのリンクは同一のソースとなるポイントから引かれることはありません。



FC アクション: アクションシンボルは実行されるアクションを記述します。アクションシンボルは番号と名前によって識別されます。チャート内の2つのアクションシンボルは同一の番号あるいは名前を持つことができません。アクションは ST、LD、IL のいずれかの言語でプログラムされます。アクションはリンクで接続され、一つのリンクがアクションに入り込み、一つのリンクがアクションから出て行きます。



FC テスト: テストシンボルはブール型の条件式を表わします。テストシンボルは番号と名前によって識別されます。ST、LD、IL 言語で記述された表現の評価結果により"YES"、"NO"のいずれかのパスへフローが導かれます。ST 言語でプログラムされている時は表現の最後にセミコロン(;)をつけることがあります。LD 言語でプログラムされている時は、変数名が割り振られない一つのコイルが評価結果を持ちます。



FC サブプログラム: フローチャートプログラムでは階層構造のプログラムを行うことができます。一つのフローチャートは別のフローチャートを読み出すことができます。呼び出されるプログラムは FC サブプログラム(チャイルド FC プログラム)と呼ばれます。FC サブプログラムを読み出すプログラムは親 FC プログラムとも呼ばれます。フローチャートにおいては FC サブプログラムの処理が完了するまでの間は親 FC プログラムの処理はサスペンドされています。SFC プログラムでは同時実行が可能です。



FC I/O操作ブロック: I/O操作ブロックシンボルはアクションを実行するためのブロックです。他のアクションと同様にI/O操作ブロックは番号と名前前で識別されます。他のアクションブロックと同じ意味を持ちますが、I/O操作ブロックではターゲットのハードウェアに依存しているI/O操作などを区別して表現することによりプログラムをわかりやすくするためのものになっています。動作そのものは他のアクションと同じです。



FC コネクタ: コネクタはチャート上での2つのポイントを直接に線を引くことなくリンクさせるために使います。コネクタは円で表現され接続ソースから接続されます。接続先となるフローチャート上のポイント(通常は接続先の名前や番号)を指定することによりコネクタの設定が完了します。



FC コメント: コメントはフローチャートの実行とは関係ありませんが、フローチャート上の任意の場所に挿入することができます。コメントによりプログラムを読みやすくドキュメントすることができます。

A.5.2 フローチャートの入力







フローチャートを入力するためには以下のエレメント(アクション、テスト、コネクタなど)をグラフィックエリアに配置させてこれらをリンクで接続することになります。






FC オブジェクトの挿入

チャートにオブジェクトを挿入するためには、ツールバーから目的のボタンを選択して、グラフィックエリアをマウスクリックします。空白エリアに配置することも、フローリンクを選択することでリンク間に挿入させることも可能です。なお、リンク間の挿入は垂直フローリンク上でのみ可能です。

以下の基本要素を挿入することができます。


 アクション (ST,IL,QuickLD のいずれかでプログラム)
 IO 操作ブロック
 テスト (ST,IL,QuickLD のいずれかでプログラム)
 コネクタ
 FC サブプログラムの読み出し
 コメント

ISaGRAF のフローチャートでは以下の典型的なブロックを準備しています。これらは既に引かれてあるフローリンクに挿入することができます。空白エリアでの配置はできません。

 If / Then / Else
 Repeat until
 While



オブジェクトの選択

グラフィックオブジェクトの選択はプログラムの編集作業で必要になります。チャート内の一つあるいは複数のオブジェクトを選択するためには、ツールバー上の矢印のボタンを選択して、チャート内のオブジェクトをマウスクリックします。複数のオブジェクトの選択の場合はマウスをドラッグして選択したい領域を指定します。選択されたオブジェクトは青色で、小さい四角によって囲まれることで表示されます。Shift や Ctrl キーを押しながらマウスクリックすることにより特定のオブジェクトを含めたり、はずしたりすることができます。

新しいオブジェクトを選択することで、前に選択されていたオブジェクトは選択からはずれます。すべての選択をはずしたい場合は、チャート内の空白エリアをマウスクリックします。

単一のオブジェクトの選択の場合はキーボードでの矢印キーにより選択されているオブジェクトを切り替えることができます。フローリンクも選択することができます。



コメントの挿入

チャート内の空白エリアにコメントテキストを挿入することができます。コメントはフローチャートの処理には影響を与えません。プログラムを分かり易くするために使われます。コメントを挿入するためにはツールバーからコメントボタンを選択してコメントを配置したい場所をマウスクリックします。コメントテキストを書き込む際にはコメントオブジェクトをダブルクリックします。"(*" や "*)" などの特殊な記号は挿入する必要はありません。選択されたコメントブロックはコーナー部分をドラッグすることにより、自由にサイズの変更が可能です。



フローの挿入

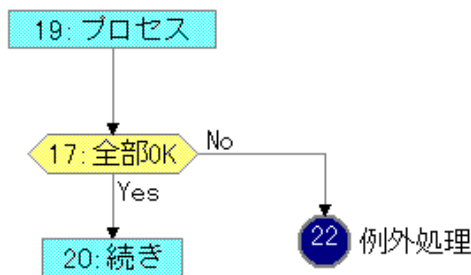
オブジェクト間をフローで接続するためにツールバー上のフローの挿入ボタンを選択します。フローの方向性にしたがってマウスをドラッグしてオブジェクト間をフローで接続します。まず、接続されていないオブジェクトを選択して、目的のポイントまでマウスをドラッグします。目的のポイントは未接続状態のオブジェクトの上部、もしくは、存在しているリンク上の場所となります。リンク上での結合部分は小さなグレーの円として表現されます。この結合ポイントはダイアグラムをアレンジするために移動させることができます。



コネクタの挿入

ISaGRAF フローチャートエディタではリンクの代わりにコネクタを使うこともできます。これはフローリンクと同様に2つのオブジェクト間の接続を表現するものです。コネクタを用いることで長いフローリンクを使わなくてすむために、チャートが分かり易いものになるメリットがあります。コネクタで別のフローチャートとの接続を行うことはできません。

コネクタはフローチャート上の一つのオブジェクトとして存在します。ツールバー上からコネクタのボタンを選択して、チャート上の配置する場所をマウスクリックします。この時接続先のリストボックスが表示され、これらから選択します。コネクタは円で表現され接続先のオブジェクトの番号と名前を持つことになります。フローリンクによりコネクタが別のオブジェクトから接続されます。



オブジェクトの移動

チャート内のオブジェクトの移動は、選択ボタンで選択モードにしてから、移動するオブジェクトをマウスでクリックしてあらかじめ選択しておきます。一つあるいは複数のオブジェクトを選択しておくことができます。これらをマウスでドラッグすることにより移動させることができます。オブジェクトは他のオブジェクトと重なり

合うことはできません。また、移動したオブジェクトは、既に張られているリンクに接続することはできません。

オブジェクト(アクション、テストなど)を選択して移動すると、下に接続されている残りのオブジェクトも自動的に移動します。



オブジェクトのサイズ変更

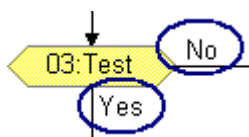
""Begin"と"End"のシンボル以外のすべてのグラフィックオブジェクトは自由にサイズの変更が可能です。オブジェクトのサイズを変更するためには、オブジェクトを選択してオブジェクトの境界線の小さな四角部分をマウスでドラッグします。ただし、この際オブジェクトが他のオブジェクトとの重なることができない(あるいはウィンドウの境界との干渉)ために、あらかじめ重なりそうなオブジェクトを移動させておく必要があるかもしれません。

垂直なフローリンクがオブジェクトと接続されている場合は、自動的に水平方向のサイズと左右の境界が調整され、フローリンクがオブジェクトの中心に来るようになります。



テストブロックの出力の入れ替え

テストブロックから出力される YES/NO の位置を入れ替えることができます。入れ替えるためには、YES、NO のラインのいずれかをダブルクリックします。



A.5.3 フローチャートの編集

フローチャートの「編集」メニューコマンドにより、選択されているオブジェクトに対しての編集が可能です。

編集(E)	ツール(T)	オプション(O)	ヘルプ(H)
元に戻す(U)			
切り取り(Q)			Ctrl+X
コピー(C)			Ctrl+C
貼り付け(P)			Ctrl+V
削除(D)			
検索(E)			
置換(R)			
ジャンプ(G)			
番号取り直し(N)			
変数選択(L)			Ctrl+D
レベル2の編集(E)			
セパレートウィンドウでレベル2の編集(W)			Return Ctrl+Return


☐ チャートの修正

DEL キーは選択したオブジェクトの削除に使えます。選択されたオブジェクトから接続されているフローリンクも同時に削除されます。「編集」-「元に戻す」メニューコマンドにより、削除されたオブジェクトを復元することができます。DEL キーは複数の選択されたオブジェクトに対しても操作ができます。「切り取り」、「コピー」、「貼り付け」コマンドにより選択されたオブジェクトを移動やコピーすることができます。

☐ 検索と置換

「編集」-「検索」、「置換」メニューコマンドにより、プログラム全体(アクション、テストブロックが ST,IL,QuickLD で記述されている)に対してテキストの検索、置換ができます。検索、置換は開かれているエディタウィンドウでテキストが検索され、テキストが検索されるとそのレベル2プログラムが開かれます。

➡ オブジェクトへの直接ジャンプ

「編集」-「ジャンプ」メニューコマンドにより、あるいはツールバー上の  ボタンの選択により、開いているフローチャート内で指定されたグラフィックオブジェクトへの直接ジャンプが行えます。ウィンドウのスクロールは自動的に行われ、オブジェクトが見えるところに表示されます。

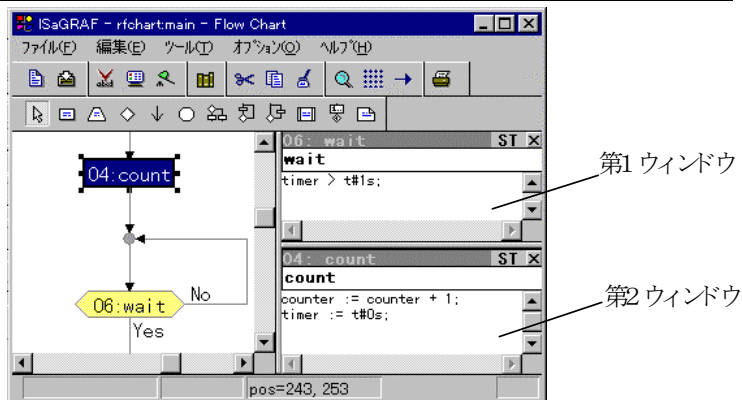
番号の取り直し

「編集」-「番号取り直し」メニューコマンドにより、フローチャート内でのオブジェクトに割り当てられている番号の取り直しを行います。フローチャート上でのオブジェクトはユニークな参照番号が取られます。新しいオブジェクトが挿入される毎に新しい番号が割り当てられます。「番号の取り直し」によって、チャート上での場所に応じてオブジェクトの番号が取り直されます。番号は上から下へ、左から右へ番号が増加します。

A.5.4 レベル2プログラムの入力

アクションやテストブロックにレベル2プログラムを記述する際には、アクションやテストブロックを選択してダブルクリックします。レベル2プログラムウィンドウがフローチャートウィンドウの右側に開きます。フローチャートウィンドウとレベル2プログラムウィンドウの間の境界はマウスで自由に移動することができます。最大で2つまでのレベル2ウィンドウを同時に開くことができます。以下にレベル2ウィンドウを開くためのキーボード、マウス、「編集」メニュー操作を示します。

	キーボード	マウス操作	「編集」メニューコマンド
第1ウィンドウ	Enter	ダブルクリック	レベル2の編集
第2ウィンドウ	Ctrl+Enter	Ctrl + ダブルクリック	セパレートウィンドウでレベル2の編集

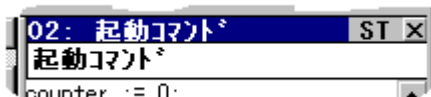


2つのレベル2ウィンドウが開いている時はこれらの境界はマウスで自由に移動できます。レベル2ウィンドウの右上のボタンでウィンドウを閉じることができます。

レベル2プログラミング言語のデフォルトは ST 言語です。なお、プログラム言語は IL、QuickLD 言語を用いることも可能です。選択されているプログラミング言語はレベル2ウィンドウのタイトルバーの右側に表示されます。プログラミング言語を変更する場合はこの部分をマウスクリックして変更します。「オプション」-「レベル2言語の設定」メニューコマンドによっても言語を切り替えることができます。ただし、プログラミング言語の切り替えはレベル2プログラムが空白の状態の時にのみ有効です。一度、プログラムを行った場合はこれを完全に削除することで再度選択が可能となります。



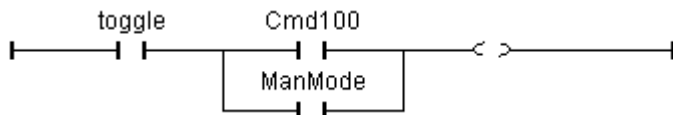
レベル2ウィンドウのタイトルバー下部に1行のテキスト編集ボックスが開きます。ここで入力されたテキストはテストやアクションブロックのコメントとなり、「ジャンプ」コマンドなどでジャンプ先を選択する場合に有効に使えます。また、フローチャートの印刷の際にも表示されます。



「オプション」-「リフレッシュ」メニューコマンドにより、レベル2プログラムの内容をメインフローチャートプログラムに反映させることができます。

A.5.5 Quick LD によるプログラミング

レベル2プログラミングに Quick LD エディタを使うことができます。テストブロックの LD プログラムのラングは1行のみとし、条件判断の結果を示す1つのコイルのみ記述できます。コイルへの変数割り当ては必要がありません。以下に例を示します。



Quick LD によるプログラミングでは、カーソルキーでセルを移動し、以下のショートカットキーを用いてラダープログラム用のシンボルを挿入します。

- F2: 接点を後に挿入／ラングの開始に使用
- F3: 接点を前に挿入
- F4: 接点を並列に挿入
- F5: コイルを並列に挿入(テストブロックでは使えません)
- F6: ブロックを後に挿入
- F7: ブロックを前に挿入
- F8: ブロックを並列に挿入
- F9: 選択されたコイルと並列でジャンプの挿入(テストブロックでは使えません)

ジャンプの飛び先はラング名(ラベル)となります。ラベルを設定する際には、ラングの先頭(左端)にカーソルをセットして ENTER キーを選択します。また、ジャンプ先を入力する「ラベルの入力」ダイアログボックスではジャンプ先のラベル名を設定します。

また、ファンクションキーの代わりに、ツールバーの該当するシンボルをクリックして挿入することもできます。

接点、コイル、ブロックの入出力パラメータにカーソルを置いて、ENTER キーで変数や定数を割り当てることができます。ブロックを選択して ENTER キーでファンクションブロックのタイプを選択することができます。ダブルクリックによっても同様のことができます。

CTRL+スペースキーもしくはツールバーで、選択されている接点のタイプ(a接点、b接点)を変更することができます。詳細は Quick LD エディタの使い方を参照願います。

A.5.6 表示オプション

「オプション」-「レイアウト」メニューコマンドにより、エディタのワークスペースや編集に関するパラメータがまとめられたダイアログボックスが開きます。「ワークスペース」グループボックスではツールバーやステータスバーの表示／非表示が行えます。「ドキュメント」グループボックスでは編集グリッドの表示／非表示やカラーモードの選択を行えます。



ツールバー上の「ズーム」ボタンにより、表示のズーム率を変更することができます。このボタンはテストやアクションを記述するレベル2の Quick LD プログラムでも有効となります。(選択されているウィンドウがズームの対象となります。)



ツールバー上の「グリッド」ボタンにより、編集グリッドの表示／非表示を行うことができます。このボタンはテストやアクションを記述するレベル2の Quick LD プログラムでも有効となります。(選択されているウィンドウがズームの対象となります。)

「オプション」―「フォント」メニューコマンドにより、チャート内のテキストフォントのタイプを選択することができます。ST,IL プログラムではフォントのサイズも有効に設定できます。FC, QuickLD ではフォントサイズは画面サイズに対応して自動的に変更されますのでサイズの設定は無効となります。

A.5.7 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。

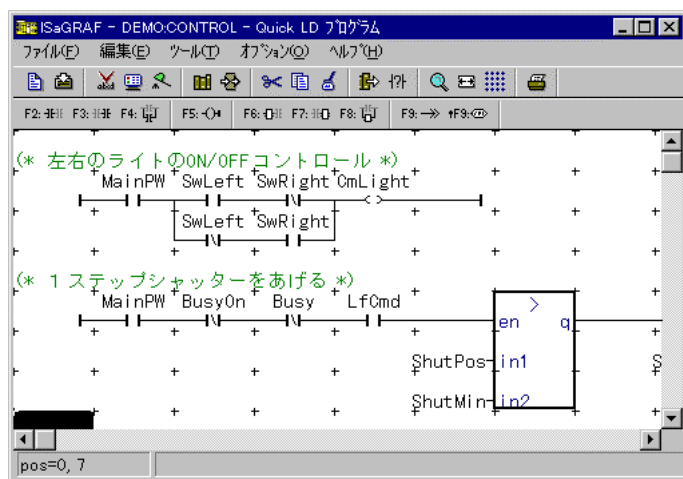


A.6 Quick LDエディタの使い方



この章ではQuick LDエディタに関してコマンドの特徴を説明します。他のエディタとの共通点に関しては「プログラムエディタ共通項目について」を参照願います。

LD言語はブール型のグラフィック表現です。ブール演算子 AND, OR, NOT がダイアグラムに表現されます。ブール入力は接点に割り付けられ、ブール出力はコイルに割り付けられます。Quick LDエディタはキーボードやマウスを使って簡単にLDダイアグラムが入力できるように作られたエディタです。このエディタでは各シンボルは自動的にアレンジされ、接続されます。ユーザによる手動でのライン接続は必要ありません。また、シンボル間の無駄な空白がありません。



A.6.1 LD言語の基本事項

LDダイアグラムは接点とコイルから表現されるラングのリストです。以下に、LDダイアグラムでの基本コンポーネントを示します。

└ 左母線

各ラングは左母線から始まります。これは常にTRUE状態です。QuickLDエディタでは最初の接点が配置されたときに自動で左母線が引かれます。各ラングは名前を持つことができます。この名前はジャンプインストラクションでの飛び先にも使われます。

接点

接点に割り付けられブール型変数の状態(TRUE、FALSE)によって、ブールデータの流れを変えます。即ち、接点の左側の状態を右側にどのように伝えるかが決まります。変数名は接点シンボルの上に表示されます。以下の接点タイプがQuickLDエディタでサポートされています。

┌┐ a接点

┌┐ b接点

┌┐ 立ち上がり接点

┌┐ 立ち下がり接点

コイル

コイルはラングの状態を受けてのアクションを表現します。即ち、コイルの左側のブール状態がコイルに割り付けられているブール型変数に設定されます。コイルの変数名はコイルのシンボルの上に表示されます。以下のタイプのコイルがQuickLDエディタでサポートされています。

{ } コイル

{ } 反転コイル

{S} セットコイル

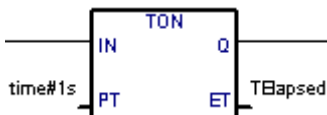
{R} リセットコイル

{P} 立ち上がり検出コイル

{N} 立ち下がり検出コイル

ファンクションブロック

LDダイアグラム中のブロックはファンクション、ファンクションブロック、サブプログラム、あるいは、オペレータ(演算子)のいずれかになります。最初の入力と出力は常にラングに接続されています。これ以外の入出力パラメータはブロックの外に表示されます。



“In Line FB”の定義方法:

と言うメニューコマンドが QuickLD エディタの“ツール”メニューに追加されました。このメニューコマンドで編集したファンクションブロックが“In Line FB”なのかどうかを決定します。このコマンドは、“In Line FB”の属性の有効/無効を交互に切り替えます。もし、“In Line FB”として定義されているならば、“In Line”と言う文字のタイトルバーがツールバーと共に表示されます。



右母線

ラングは右母線で終わります。QuickLDエディタでは右母線はコイルが配置されたときに自動的に引かれます。



ジャンプシンボル

ジャンプシンボルはラングラベル名へのジャンプができます。ジャンプ先のラングは同一のLDダイアグラムになければなりません。ジャンプシンボルはラングの最後に配置されます。ラング状態がTRUEの時に目的のラングへのジャンプが行われます。後ろ向きのジャンプは処理がループする可能性がありますので注意を要します。



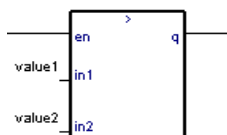
リターンシンボル

リターンシンボルはラングの最後に配置されます。もし、このラングの状態がTRUEの時、このプログラムの実行を停止して次のプログラムに移ります。



EN 入力

ファンクション、ファンクションブロック、演算子の中には最初の入力パラメータがブール型でないものがあります。この場合でも最初のパラメータは必ずラングに接続される必要があるため、このような状況ではブロックに自動的に "EN" と呼ばれる新たな入力パラメータが生成されます。"EN"入力がTRUE状態の時のみこのブロックが実行されます。以下に "EN"入力を含むブロックの例とこれと等価なSTプログラムコードを示します。

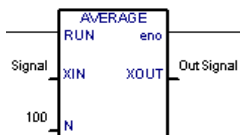


```
IF rung_state THEN
  q := (value1 > value 2);
ELSE
  q := FALSE;
END_IF;
(* continue rung with q state *)
```



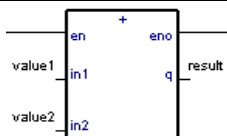
ENO 出力

ファンクション、ファンクションブロック、演算子の中には最初の出力パラメータがブール型でないものがあります。この場合でも最初のパラメータは必ずラングに接続される必要があるため、このような状況ではブロックに自動的に "ENO" と呼ばれる新たな出力パラメータが生成されます。"ENO"出力は常にブロックの最初の入力パラメータの状態と同じ状態となります。以下に "ENO"出力を含む AVERAGE ブロックの例とこれと等価なSTプログラムコードを示します。



```
AVERAGE(rung_state, Signal, 100);
OutSignal := AVERAGE.XOUT;
eno := rung_state;
(* continue rung with eno state *)
```

ある場合には、EN と ENO の両方が必要なブロックもあります。以下に算術演算ブロックの例を示し、等価なST表現も示します。



```
IF rung_state THEN
    result := (value1 + value2);
END_IF;
eno := rung_state;
(* continue rung with eno state *)
```

Quick LDエディタの制限

QuickLDエディタではコイルの右側にシンボルを挿入することはできません。もし、複数のコイルが必要な場合はコイルは並列に配置されます。一つのラングで複数のコイルを異なる条件で配置することはできません。

A.6.2 LDダイアグラムの入力

Quick LDエディタでの編集操作はキーボードやマウスを使って行うことができます。



編集グリッド

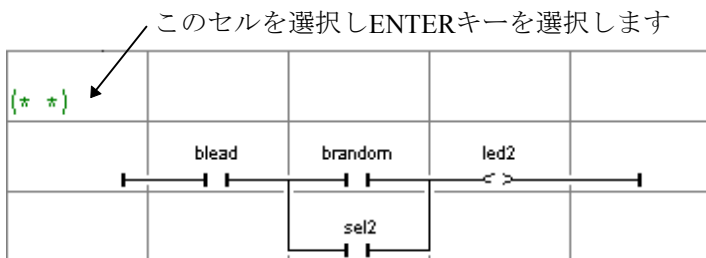
LDダイアグラムの各シンボルはグリッドで区切られたセル上に配置されます。1セルに1シンボルが配置されます。現カーソル位置を移動するためには、カーソルキーで移動するか、直接マウスで指定します。選択中のセルは反転状態となります。コピー／貼り付けなどの操作のために複数のセルを選択する場合はマウスでドラッグするか、SHIFT キーを選択しながらカーソルキーを動かします。

新しいラングの作成

新しいラングを作成する場合は、最終ラングの次の行に選択セルを移動します。そこで接点を挿入するために**F2**キーを選択します。接点1つにコイル1つの新しいラングが作成されます。

ラングコメントの入力

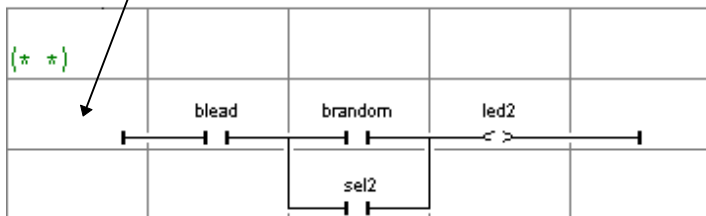
各ラングは最大2行までのテキストコメントを持つことができます。ラングコメントの入力のためには、ラングの左上の(* *)のセルを選択して ENTER キーを選択します。あるいはダブルクリックします。



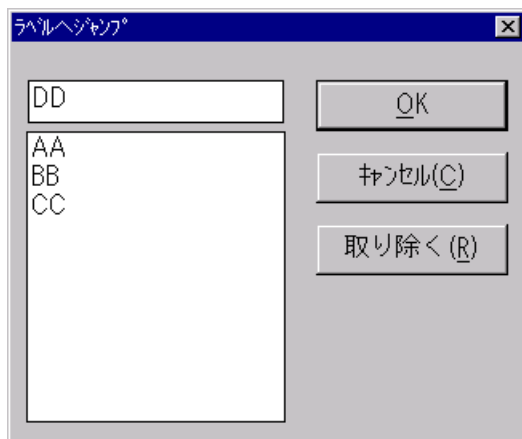
■ ラングラベルの入力

各ラングは名前を持つことができます。この名前(ラベル)はジャンプ操作のときのジャンプ先に使われます。ラングのラベルを入力するには、選択セルをラングの左端に移動して、ENTER キーを選択します。あるいはダブルクリックします。

このセルを選択しENTERキーを選択します



Quick LDエディタは入力済みのラングラベルを保存しています。「ラベルへジャンプ」ダイアログボックスでジャンプ先の選択を行うか、新しくラングラベルの入力を行います。



新しいラングラベル名を入力すると、これはリストに追加されます。“取り除く”ボタンが選択されると選択中のラベル名がリストから削除されます。ただし、これではラベル自体は削除されません。なお、選択中のラングのラベル名を消したい場合は単に、このダイアログボックスの入力フィールド内を空にして“OK”ボタンを選択します。

■ ラングへのシンボル挿入

既存のラングにシンボル(接点、コイル、ブロック)を挿入する場合は、まず、挿入位置のセルを選択して(複数セルの同時選択も可能)から以下のファンクションキーのいずれかを選択します。

F2..... 左側に接点を挿入

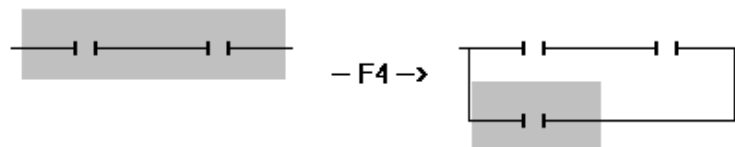
F3..... 右側に接点を挿入

- F4 選択シンボルと並列に接点を挿入
- F6 左側にブロックを挿入
- F7 右側にブロックを挿入
- F8 選択シンボルと並列にブロックを挿入

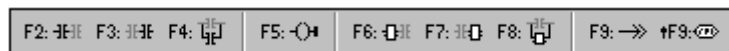
以下のコマンドは選択セルがコイルの場所の時のみ有効となります。

- F5 並列にコイルの追加
- F9 並列にジャンプシンボルの追加
- Shift + F9 並列にリターンシンボルの追加

F4、F8の並列挿入に関しては、複数の接点がまとめて選択されているような場合は、シンボルの挿入は選択されているシンボルグループに対して並列に行われます。以下に例を示します。



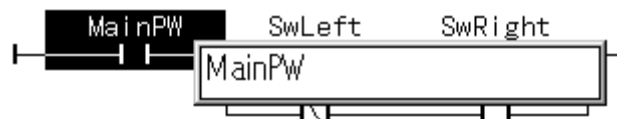
ダイアグラムへシンボルを挿入する場合、マウスで LD ツールバー上のボタンをクリックすることもできます。



変数名の入力

選択中の接点、コイル、ブロックに変数名やブロック名を割り当てるためには、ENTER キーを選択します。あるいは、ダブルクリックします。変数選択ダイアログボックスが表示されます。このダイアログボックスの使い方の詳細は「プログラムエディタ共通項目」を参照願います。ファンクション、ファンクションブロック、演算子の種類の設定の場合は、ボックスの内部を選択して ENTER キーを選択します。ブロックへの入出力パラメータの割り当ての際には、ボックスの**外側**を選択して ENTER キーを選択します。

「オプション」-「マニュアル入力」メニューコマンドが選択されている時は、変数名やファンクションブロック名の入力のために以下のような1行のテキストボックスが表示され、直接名前を入力することができます。変数名やファンクション名を入力した後で ENTER キーで入力が完了します。ESC キーで入力をキャンセルし、テキストボックスを閉じることができます。このテキストボックスはマウス操作では閉じることができません。





接点、コイルのタイプ変更

「編集」-「コイル/接点タイプの変更」コマンドにより選択中の接点やコイルのタイプを変更できます。この操作はスペースキー、あるいは、CTL+スペースキーを選択することでも行うことができます。接点、コイルのタイプは以下の通りです。

	a接点		コイル
	b接点		反転コイル
	立ち上がり接点		セットコイル
	立ち下がり接点		リセットコイル
			立ち上がり検出コイル
			立ち下がり検出コイル



ダイアグラムにラングの挿入

「編集」-「ラング挿入」コマンドにより、選択中のラングの直前にラングを挿入できます。挿入されるラングは接点1つ、コイル1つの基本ラングです。

A.6.3 LDダイアグラムの修正

「編集」メニューでダイアグラムの修正を行い、LDダイアグラムを完成させます。多くのコマンドは選択セルに対してのアクションとなります。

編集(E)	ツール(T)	オプション(O)	ヘルプ(H)
元に戻す(U)			
切り取り(C)		Ctrl+X	
コピー(C)		Ctrl+C	
貼り付け(P)		Ctrl+V	
形式を指定して貼り付け(O)			
クリア(Q)			
ラング挿入(I)			
シンボル/テキストのセット(S)		Enter	
コイル/接点タイプの変更(E)		Space	
検索(F)			
置換(R)			
対応変数名の検索(N)		Alt+F2	
対応コイルの検索(Q)		Alt+F5	
ダイアグラムをファイルでコピー(W)			



削除

DEL キーで選択されたシンボルを削除します。ただし、コイル、ジャンプ、リターンシンボルの場合、これが該ラングの唯一の出力の時は、削除することはできません。「編集」-「元に戻す」コマンドにより、削除されたシンボルを戻すことができます。削除コマンドは、複数のシンボルやラングが選択されている場合も実行できます。また、ラングコメントを選択して、DEL キーを押すと、コメントをクリアでき

ます。DEL キーをラングの左端で使うと対象のラングが削除されますので注意して下さい。

シンボルのコピー

「編集」-「切り取り」、「コピー」、「貼り付け」コマンドにより選択中のシンボルを移動したり、コピーしたりできます。「編集」-「形式を指定して貼り付け」コマンドにより、挿入位置が選択ができます。

- 左側に貼り付け
- 右側に貼り付け
- 並列に貼り付け

ラングの操作

選択セルがラングの左端の場合は、すべての編集コマンドがラング全体に対して行われます。これで、ダイアグラム内でのラングのアレンジが容易に行えます。更に、複数のラングをまとめて選択・操作する場合は、左端の列のセルを垂直方向に複数続けて選択します。

検索と置換

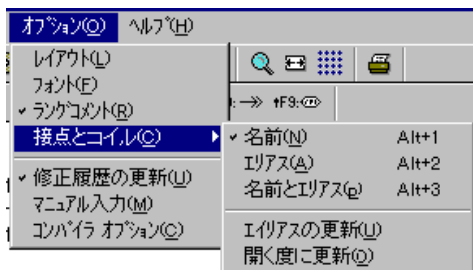
「編集」-「検索、置換」コマンドにより、ダイアグラム中のテキストを検索、置換することができます。検索は完全一致の場合のみサポートしています。検索は接点名、コイル名、ブロック名、ブロックパラメータ名、ラングラベル名に対して行われます。ラングコメント中のテキストの検索には使えません。置換コマンドによりブロックの置き換えはできません。検索方向は選択セルの前方、後方のどちらでも行えます。変数名の検索のために以下のショートカットが用意されています。

ALT + F2 選択変数名と同じ変数名を近い順に検索します。この検索はファンクションブロック名とラングラベル名に対しても行えます。

ALT + F5 選択変数名と同じコイルを近い順に検索します。この機能はデバッグ中に疑わしい接点が見つかった場合にコイルの検索をする場合に有効です。

A.6.4 表示オプション:「オプション」メニューコマンド

「オプション」メニューには、画面上でのLDダイアグラムの表示方法の変更、各種情報の表示・非表示の選択などのコマンドを含んでいます。



□ 接点とコメント

変数のコメントをダイアグラム上に表示したり非表示にしたりする時は**「オプション / 接点とコメント」**を使用して下さい。コメントはその変数上にカーソルを移動すると表示されます。このオプションはオフラインモードでもオンラインモードでも使用することが出来ます。

□ ラングコメント

「オプション」—**「ラングコメント」**コマンドにより、ラングコメントの表示・非表示がダイアグラム全体に対して設定できます。ラングコメントを非表示にすることで、より多くの行数のラングを画面に表示できます。このコマンドによりラングコメントが消えることはありません。いつでも表示・非表示の切り替えが行えます。

□ 変数名とエリアス

QuickLDエディタでは、接点・コイル・ブロックのパラメータを特定する場合、割り付けられた変数名の他に**「エリアス」**を使用することができます。各変数におけるエリアスとは変数のコメントの最初に出現した`:`より前の部分のテキスト、あるいは、左から16バイトの文字列となります。以下に例を示します。

変数のコメント:

short text
long text with no separator
short text: long description
日本語のコメントも使えます。

エリアス:

short text
long text with n
short text
日本語のコメント

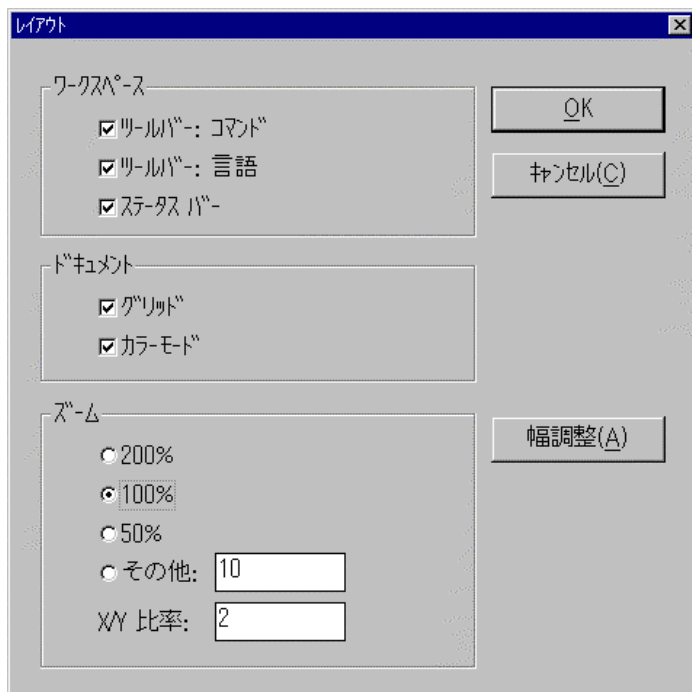
エリアスはLDダイアグラムの実行には何も影響を与えず、コメントと同じ位置づけになります。エリアスは変数を選択するときに自動的に変数コメントから抽出され、変更することはできません。ダイアグラムに表示する変数名やエリアスは3通りの表示方法があり、**「オプション」**—**「接点とコイル」**コマンドにより切り替えられます。

- 名前(変数名)のみ
- エリアスのみ
- 名前(変数名)とエリアス

Quick LD エディタでは、変数辞書が更新された時に LD プログラムの変数エリアス名を画面で自動更新は行いません。但し、**「オプション」**—**「接点とコイル」**—**「エリアスの更新」**メニューコマンドにより、LD プログラム中のエリアスが更新されます。また、**「オプション」**—**「接点とコイル」**—**「開く度に更新」**コマンドを設定すると、LD プログラムが開かれる時に変数エリアス名の自動更新を必ず行います。但し、この設定によりプログラムが開く際に要する時間が長くなることがあります。

表示オプション

「オプション」-「レイアウト」コマンドにより、エディタのワークスペースとLDダイアグラム表示方法に関するオプションパラメータを選択できます。



ワークスペース グループボックス内の設定にはツールバー、ステータスバー、LD ツールバーの表示・非表示選択があります。ドキュメント グループボックス内の設定にはグリッドの表示・非表示、カラーモードの選択があります。



更に、ズーム グループボックスの内の設定で、全体のズーム率を手動で設定できます。尚、ツールバーのズームボタンで規定の表示比率を設定することもできます。



又、セルの X/Y 比率のカスタマイズにより効率よくダイアグラムの内容をウィンドウに表示できます。X/Y 比率(セル幅)ボタンはツールバーアイコンにも含まれています。

ツールバーの“セル幅”ボタンにより、レイアウトコマンドを使わずに、自動的に現状のQuickLDエディタのウィンドウサイズにフィットするように X/Y 比率を設定できます。

「オプション」―「フォント」メニューコマンドにより、フォントのタイプを変更することができますが、フォントサイズに関してはグラフィック画面のズーム設定率により自動的に調整されますので、ここでの設定は有効にはなりません。

A.6.5 オンラインヘルプ

Quick LD エディタからファンクションブロックについてのヘルプを参照したい時

- LD ダイアグラム上でファンクションブロックを選択します。
- F1 を押してください。

ファンクションブロックについてのヘルプが表示されます。カスタマイズされた C や IEC のファンクション、ファンクションブロックでは、ヘルプはライブラリエディタで入力された“technical note”にあります。(テキストのみ)

A.6.6 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。

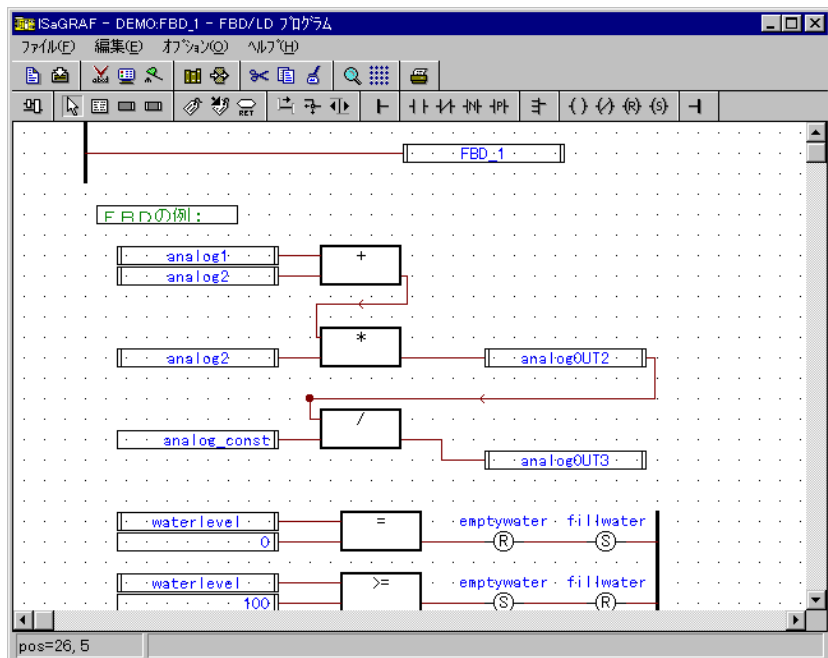
	他のプログラムを開く
	保存
	プログラムのバリファイ
	シミュレート
	デバッグ
	辞書
	サブプログラムのパラメータ
	切り取り
	コピー
	貼り付け
	シンボルの入力
	コイル/接点タイプ
	ズーム
	ズーム
	セル幅
	グリッド
	印刷

A.7 FBD/LDエディタの使い方



この章では FBD/LD エディタに関してコマンドの特徴を説明します。他のエディタとの共通点に関しては「プログラムエディタ共通項目について」を参照願います。

ISaGRAF FBD/LD グラフィックエディタはFBDプログラム用のエディタですが、LDエディタを含んでいます。ただ、純粋なLDプログラムの場合はQuick LDエディタを推奨します。



A.7.1 FBD/LD 言語の基本事項

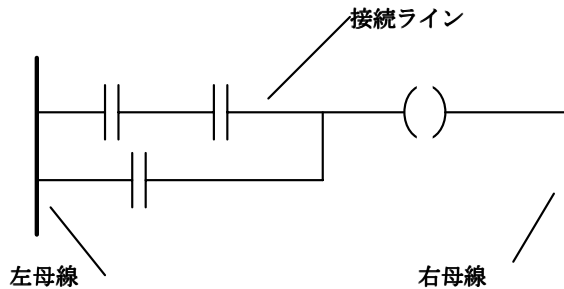
FBD言語は各種タイプの長方形(箱)で表わされるファンクションブロックが接続されたグラフィック表現方法の一つです。演算子もブロックの形で表現されます。箱の左側にはファンクションの入力が、箱の右側にはファンクションの出力が接続されます。変数は論理的なリンクをもって箱に接続されます。出力は別のファンクションの入力に接続されることもあります。



LD言語はブール表現のグラフィックな表現方法です。ブール演算の**AND**、**OR**、**NOT**がグラフィックに表現されます。ブール型入力とは接点に、ブール型出力はコイルに割り当てられます。

LDダイアグラムは左右の垂直のラインに挟まれています。これらは**左母線**、**右母線**と呼ばれています。

接点とコイルは左右の母線に**水平ライン**で接続されています。各々のライン上のセグメントは **FALSE** または **TRUE** の値を持ちます。**左母線**に接続されているポイントは全て TRUE 値となります。



LDとFBDダイアグラムは必ず左から右へ、さらに上から下へと処理されます。

LD言語とFBD言語の詳細については、言語リファレンスをご覧ください。

FBD/LDダイアグラムでサポートされている基本のコンポーネントを以下に示します。

ト

左母線

各 LD ラングは**左母線**に接続されなければなりません。これは常に TRUE 状態です。FBDエディタではブール型のシンボルを接続することもできます。

リ

右母線

コイルの右側は**右母線**に接続されなければなりません。ただし、FBDエディタでは必ずしも右母線が必要とは限りません。もしコイルの右側に別のシンボルが配置され、コイルからリンクされていない＝コイルが行末に配置されている場合は、右母線はコイルの一部として表現されます。

キ





垂直 ("OR") 接続ライン

LDエディタでの垂直接続ラインは左側の複数の接続をまとめて右側に接続します。これは左側に接続されている複数ラインの論理和 (OR) 演算となります。

ト

接点





接点に割り付けられブール型変数の状態(TRUE、FALSE)によって、ブールデータの流れを変えます。即ち、接点の左側の状態を右側にどのように伝えるかが決まります。変数名は接点シンボルの上に表示されます。以下の接点タイプがFBD/LDエディタでサポートされています。

 a接点
 b接点
 立ち上がり接点
 立ち下がり接点



コイル

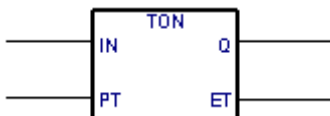
コイルはラングの状態を受けたアクションを表現します。即ち、コイルの左側のブール状態がコイルに割り付けられているブール型変数に設定されます。コイルの変数名はコイルのシンボルの上に表示されます。以下のタイプのコイルがFBD/LDエディタでサポートされています。

 コイル
 反転コイル
 セットコイル
 リセットコイル



ファンクションブロック

FBDダイアグラム中のブロックはファンクション、ファンクションブロック、サブプログラム、あるいは、オペレータ(演算子)のいずれかになります。ブロックの入力/出力パラメータは必ず変数、接点/コイル、別のブロックの入力等に接続されていなければなりません。パラメータの名前はブロックの内側に表示されます。



ラベル

ラベルはダイアグラム中のどこにでも配置することができます。ラベルは処理の実行順を変更するためのジャンプ命令の飛び先として使われます。ラベルは他のシンボルとは接続されません。ダイアグラムを読みやすくするために通常はラベルは左端に配置されます。



ラベルへのジャンプ

ジャンプ命令はダイアグラム中に配置された指定ラベルへ実行を移します。このシンボルの左側はブール変数(状態)に接続される必要があります。左の接続ラインがTRUEの時ジャンプが実行されます。後ろ向きのジャンプは処理がループする可能性がありますので注意を要します。



リターンシンボル

リターンシンボルはラングの最後に配置され、その左側はブール変数(状態)に接続されます。このラングの状態がTRUEの時、プログラムの実行を停止して次のプログラムに移ります。



変数

ダイアグラム中の変数は小さな長方形で表現されます。左右からの接続ラインが別のシンボルと接続されます。



接続ライン

シンボル間を接続する場合に使用します。接続ライン(リンク)は出力ポイントから入力ポイントへ接続されます。



論理反転接続ライン

論理接続(ブールリンク)を行う際に、論理を反転して接続する場合があります。この場合に使用します。



ユーザ定義のコーナ

接続ライン引く際に、ユーザが定義した任意の場所に、接続コーナポイントを配置することができます。配置されたコーナは接続ラインが通るポイントとなります。このコーナを配置することによって、ダイアグラムの接続ラインを整理して読みやすくすることができます。コーナがない場合は、FBDエディタはデフォルトのルールで接続ラインが引かれます。

A.7.2 FBDダイアグラムの入力

ダイアグラムの入力はシンボル(ブロック、変数、接点、コイルなど)を配置してこれらを接続ラインで接続することになります。



シンボルの配置

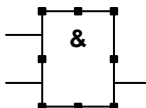
ダイアグラムにシンボルを配置する場合は、ツールバーから配置するシンボルに該当するボタンをまず選択してから、エリア上の配置したい場所をクリックします。



シンボルの選択

編集操作では、まずシンボルを選択します。LD/FBDエディタでは1つ以上のシンボルの選択が可能です。シンボルを選択する場合はツールバーで**選択ボタン**(矢印ボタン)をチェックしておきます。一つのシンボルを選択する場合はそのシンボルをクリックします。複数のシンボルを選択する場合はマウスで長方形領域をドラッグしながら必要とするシンボルを全て選択します。

選択された領域内(境界と交差してもOK)の**選択された**シンボルは小さな黒い点で囲まれます。



新しいシンボルの選択を行なうと以前に選択されていたシンボルは選択からはずれます。選択されているシンボルを選択からはずす場合は単に選択されているシンボルの外側でシンボルがない領域をクリックします。



コメントの挿入

コメントはダイアグラムの任意の場所で記入できます。コメントはプログラムの実行には影響を持ちません。プログラムを読みやすくするものです。コメントを挿入するためには、ツールバーよりこのボタンを選択します。次に、コメントを挿入したい場所で適当な長方形をドラッグして領域を決めます。コメントの入力には (* *) などの記号は必要ありません。コメント領域のサイズは境界線をドラッグすることで変更できます。



シンボルの移動

ダイアグラムの中のシンボルを移動する場合は、まず、移動したいシンボルを選択します。次に、選択されているシンボルの領域内をマウスでドラッグして移動させます。

シンボルに接続されているラインなどのオブジェクトはシンボルの移動により自動的に再描画されて新しいシンボルの場所に合わせてラインは接続されます。



接続ラインを引く

シンボル間の接続ポイント間にラインを引くには、これらのボタンをツールバーアイコンから選択します。接続ポイントから何もない領域に接続ラインを引くと、自動的にユーザ定義のコーナーに接続されて引き続き別のシンボルに接続しやすくなっています。



接続ラインの移動

「ツール」-「ラインの移動」コマンドにより、選択された接続ラインのルーティングを自動的に変更できます。このコマンドはユーザ定義のコーナーを経由している場合は使えません。接続ラインが3つのライン(縦、横、縦)から成り立っている場合は2番目のラインの移動が行われます。以下に例を示します。



接続ラインのタイプ変更

接続ラインの終端をダブルクリックすることで、論理反転(あるいはその逆)が簡単に変更できます。



LDラングの描き方

新しいLDラングを作成する場合は以下のように行います。

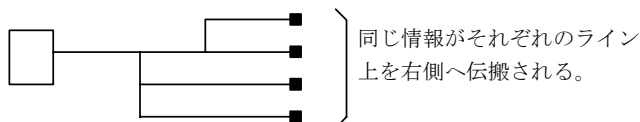
- ・ 左母線を挿入
- ・ コイルの挿入(このとき、自動的に右母線が引かれます)
- ・ 他の接点や垂直OR接続ラインの挿入(直接ラングに挿入できます)
- ・ ラングの外に接点やコイルを挿入すると水平ラインが自動で左右の母線へ引かれます

- ・ ただし、左右の母線の間に接点やコイルが配置されないと水平ラインは引かれません
- ・ ラング上に挿入された接点、コイルはラング上を移動できます
- ・ 接点、コイルを挿入したときに引かれた水平ラインは選択して削除もできます
- ・ 新しく接点やコイルを既存のラングのライン上に挿入できます

E

マルチ接続ライン

マルチ接続ラインはある出力ポイントの右側で作られます。即ち、ダイアグラムの中では同一の情報が複数の出力ポイントへ伝搬されます。



出力ポイントの右側のラインの数には制限はありません。一方、2本の異なる接続ラインが同一の入力ポイントに接続されることは、次のLDシンボルにおける例外を除いて存在しません。

┌ 右母線

└ 垂直 (OR) 接続ライン

(これらの2つのシンボルの左側からの入力数には制限がありません。)

A.7.3 FBDダイアグラムの編集:「編集」メニューコマンド

「編集」メニューコマンドには、ダイアグラムを編集して完成させるためのコマンドを含んでいます。たいいては選択されたシンボルに対して操作します。



E ダイアグラムの修正

DEL キーは選択されたシンボルの削除に使われます。シンボルを削除すると、接続したリンクも一緒に削除されます。「編集」-「元に戻す」コマンドにより、DEL

キーコマンドで削除されたシンボルを戻すことができます。グループ選択された複数のシンボルに対しても DEL キーを使うことができます。「編集」―「切り取り」、**「コピー」**、**「貼り付け」**コマンドにより選択中のシンボルを移動したり、コピーしたりできます。

検索と置換

「編集」―**「検索、置換」**コマンドにより、ダイアグラム中のテキストを検索、置換することができます。検索は完全一致の場合のみサポートしています。検索は接点名、コイル名、ブロック名、ブロックパラメータ名、ラングラベル名に対して行われます。ラックコメント中のテキストの検索には使えません。置換コマンドによりブロックの置き換えはできません。検索方向は選択セルの前方、後方のどちらでも行えます。

実行順表示

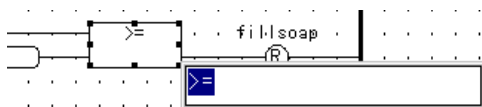
FBDダイアグラムではループを使用した場合など、各シンボルの実行順(基本は左から右へ、上から下へ)をダイアグラムから正しく判断しにくい場合があります。混乱を避けるために**「ツール」―「実行順の表示」**コマンドまたは CTRL+F1 キーにより、コンパイル時に決定されるシンボルの実行順を表示できます。実行順はコイル、ファンクションブロックの出力変数などのシンボルの近くに(1-n)で表示されます。



シンボルとテキストの入力

対象となるシンボルをダブルクリックして、わりあてられている変数やテキストを変更します。変数の場合は、変数選択用のダイアログボックスが開きます。シンボルが接点やコイルの場合は、タイプの変更(a接点、b接点、立ち上がり検出など)も可能です。

「オプション」―**「マニュアル入力」**メニューコマンドが選択されている時は、変数名やファンクションブロック名の入力のために以下のような1行のテキストボックスが表示され、直接名前を入力することができます。変数名やファンクション名を入力した後で ENTER キーで入力が完了します。ESC キーで入力した内容をキャンセルして、テキストボックスを閉じることができます。このテキストボックスはマウス操作では閉じることができません。



「オプション」―**「入力プロンプト」**をチェックすると、接点・コイルを配置した直後に、変数選択ウィンドウが表示され、変数の割付が促されます。変数やラベルシンボルの場合、必ず配置直に変数名やラベル名の設定を行わなければなりません。



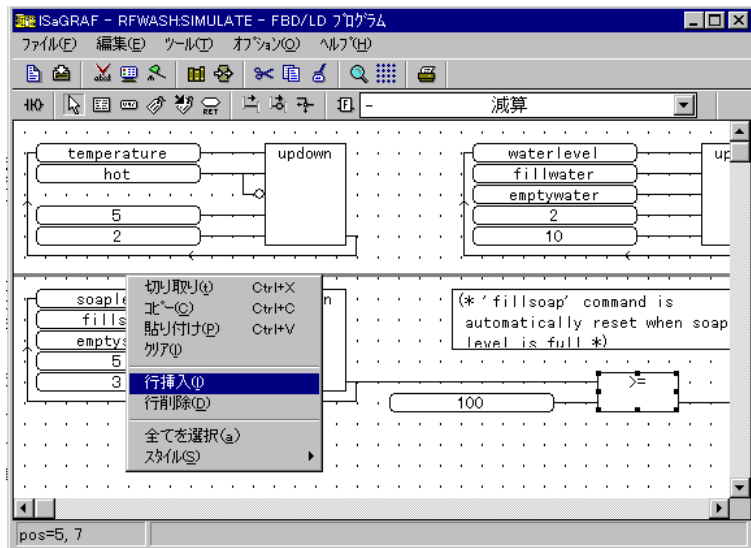
ファンクションブロックタイプの選択

ブロックのタイプを変更する場合は、ブロックをダブルクリックして新しいタイプを選択します。このとき、入力パラメータ数の変更ができます。(例えば、AND、

OR,ADD, MUL 演算は入力パラメータ数がデフォルトでは2ですがこれを変更することができます)

空間の挿入

FBD ダイアグラムでマウスの右ボタンをクリックすることにより、ポップアップメニューが表示されます。このポップアップメニューにはカーソルのある場所に空間を挿入／削除する以下のコマンドが含まれています。



行の挿入:..... このコマンドによりカーソルのある場所に編集グリッドで4行分の空間が挿入されます。

行の削除:..... このコマンドによりカーソルのある場所で未使用の行が削除されます。このコマンドで FBD エレメントが削除されることはありません。

右マウスクリックした時にポップアップメニューが開くと同時に水平のラインが表示されます。この場所で空白行が挿入されたり、削除されたりします。

A.7.4 表示オプション

「オプション」メニューコマンドにより、FBDダイアグラム表示方法に関するオプションパラメータを選択できます。

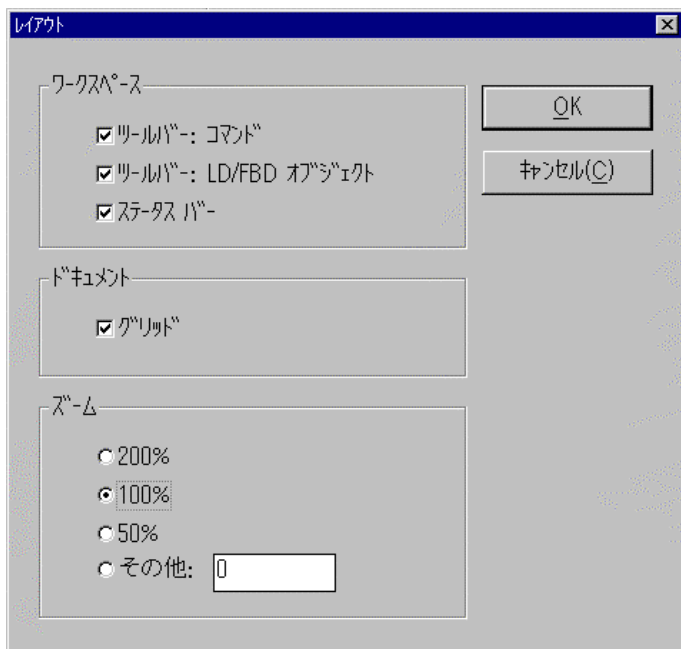
接点とコメント

変数のコメントをダイアグラム上に表示したり非表示にしたりする時は**「オプション / 接点とコメント」** を使用して下さい。コメントはその変数上にカーソルを移動する

と表示されます。このオプションはオフラインモードでもオンラインモードでも使用することが出来ます。

表示レイアウトのカスタマイズ

「オプション」-「レイアウト」コマンドにより、FBDダイアグラム表示方法に関するオプションパラメータを選択できます。以下のダイアログボックスが開きます。

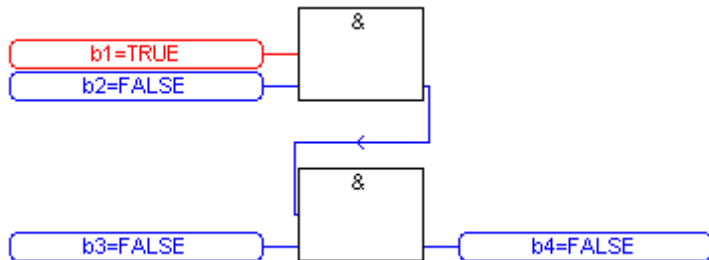


- ・ ワークスペース グループボックス内の設定にはツールバー(コマンド、FBD/LD コマンド)、ステータスバーの表示・非表示選択があります。
- ・ ドキュメント グループボックス内の設定にはグリッド表示・非表示選択があります。

Power flow デバッグ

LD/FBD のシミュレーションモードもしくはオンラインデバッグモードでは、論理フローを容易に確認するために赤と青に着色された power flows の表示が可能です。しかし、power flow デバッグを行うとターゲット上でメモリの確保が行われることになります。FBD では、オプションメニューの **Power flow デバッグの使用** を選択することでオフラインモードの間でも power flow デバッグが可能です。この機能はワークベンチをインストールした時デフォルトで使用可能となっています。

FBD エディタはその値に従って描画されます。"0" や "FALSE"状態は青色で表示されます。"0" 以外や "TRUE" 状態は赤で表示されます。



ページ境界の表示

印刷された時、FBD ダイアグラムは選択されたプリンタや用紙によって2つ折りに分離されます。FBD エディタではそれぞれの2つ折りの境界線を事前に表示させることが出来ます。ダイアグラムを作成している段階で、2ページにまたがる部分へのシンボル配置を避けることが出来ます。

エディタがグリッド表示されている場合、このページ境界は灰色のラインで表示されます。

FBD エディタはページサイズを表示する為に、最後に選択されたプリンタの設定環境を使用します。FBD でページ境界を見る前に使用するプリンタや用紙の確認をする必要がある。従ってダイアグラムを表示する前に、しなければならない事は次の通りです。

1. ドキュメントジェネレータの起動
2. プリンタや用紙サイズなどの選択
3. 印刷しないでドキュメントジェネレータを終了する。
4. FBD プログラムを開く
5. グリッドを表示する。



更に、ズーム グループボックス内の設定には全体のズーム率の選択があります。尚、ズームボタンはツールバーアイコンにも含まれています。

「オプション」―「フォント」メニューコマンドにより、フォントのタイプを変更することができますが、フォントサイズに関してはグラフィック画面のズーム設定率により自動的に調整されますので、ここでの設定は有効にはなりません。

A.7.5 スタイルと編集箇所の記録

FBD/LD エディタでは FBD/LD ダイアグラム上での全てのエレメントに対してグラフィックスタイルを割り当てることができます。スタイルコマンドによりエレメントに色をつけることができます。さらに、プログラムのバージョンの管理の目的で、編集箇所に跡を残すためのスタイルが準備されています。

なお、シミュレーションやデバッグ時には表示されない色があります。例えば、赤や青色は変数の TRUE/FALSE を表示するために使われます。

設定スタイル

ISaGRAF の FBD/LD エディタで設定できるスタイルを以下に示します。

- | | |
|-------------|--|
| 標準 | デフォルトのスタイルは黒色で表示されます。編集トラッキングを行う場合、標準スタイルはオリジナルのダイアグラムを表わします。プログラム実行時はこの部分がスキャンされます。 |
| 修正済み | 修正済みのスタイルはピンク色で表示されます。編集トラッキングでは、修正済みスタイルはオリジナルのダイアグラムから追加、修正が加わったことを示すことになります。プログラム実行時はこの部分もスキャンされます。 |
| 削除済み | 削除済みのスタイルは灰色の点線で表示されます。編集トラッキングでは、削除済みスタイルはオリジナルのダイアグラムから削除されたことを示します。プログラム実行時はこの部分はスキャンされません。 |
| カスタム | 既存のスタイルに加えて、カスタムスタイルを追加することができます。この場合、選択されたエレメントの色をカラー選択のダイアログボックスを使って任意に設定することができます。カスタムスタイルの設定による、プログラム実行時の影響はありません。 |

手でエレメントのスタイルを変更させる場合は、エレメントを選択した後で、「編集」→「スタイル」サブメニュー、あるいは、右マウスクリックによるポップアップメニューの「スタイル」サブメニューを使います。

編集トラッキング

「編集」→「スタイル」→「編集個所にマーク」メニューコマンドにより、編集のトラッキングの有効／無効を切り替えることができます。デフォルトではこの設定はセットされていますので、自動的に編集トラッキングが行われます。

「編集個所にマーク」コマンドがセットされている時は、変更が行われたエレメントはすべて修正済みのスタイルとなります。DEL キーやカット操作で削除された時はダイアグラム上から完全に削除されることなく、削除済みスタイル（点線での灰色表示）が適用されます。この機能により、プログラムの編集をすべて残しておくことが可能となります。

「編集」→「スタイル」→「削除済みアイテムのクリア」メニューコマンドにより、削除済みスタイルとなっているエレメントがダイアグラム上から完全に削除されます。このコマンドは常にダイアグラム上の全ての削除済みスタイルに適用されます。

削除済みスタイルのエレメントを標準スタイルなどに復元するためには、削除済みスタイルのエレメントを選択して「編集」→「スタイル」→「標準」などを選択します。この操作によって、オブジェクトの接続上の文法エラーなどが生ずる可能性

(例えば、一つの接続ポイントに2つ以上のオブジェクトが接続される等)があります。このエラーの検出は次のプログラムのベリファイで行われます。

A.7.6 オンラインヘルプ

LD/FBD エディタでファンクションブロックについてのヘルプを参照する時

- LD/FBD ダイアグラム上でファンクションブロックを選択します。
- F1 を押してください。

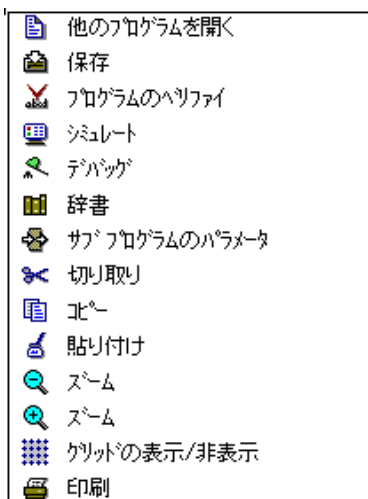
ファンクションブロックについてのヘルプが表示されます。カスタマイズされた C や IEC のファンクション、ファンクションブロックでは、ヘルプはライブラリエディタで入力された“technical note”にあります。(テキストのみ)

A.7.7 FBD ダイアグラムの印刷

"ファイル / プリント" コマンドはプリンタに FBD ダイアグラムを出力します。このコマンドは FBD ダイアグラムをプリントアウトする為に必要なドキュメントジェネレータを自動的に起動します。

A.7.8 ツールバーアイコン

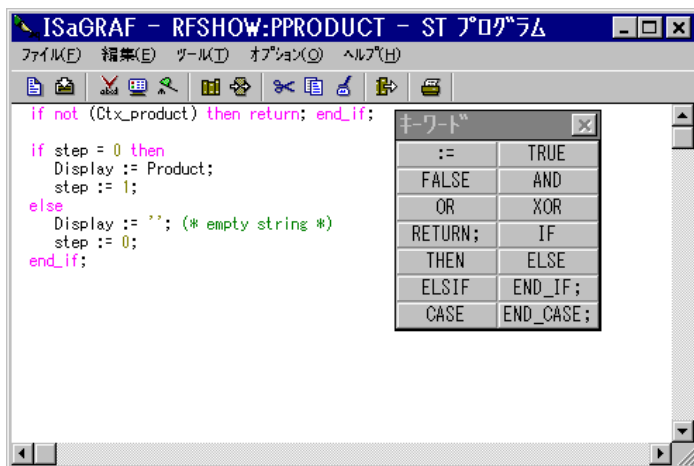
以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.8 テキストエディタの使い方



この章では、主にST、ILプログラムエディタとして使われるテキストエディタに関しての特徴について記述します。他のプログラムエディタと共通している項目に関しては、「プログラムエディタ共通項目」を参照願います。
ISaGRAFのテキストエディタは全てのテキスト入力に使われます。例えば、プロジェクト記述、ST、ILプログラムエディタ、修正履歴ファイルエディタ、ライブラリの技術メモ、Cソースエディタなどが含まれます。



A.8.1 「編集」メニューコマンド

「編集」メニューコマンドには、エディタで選択中のキャラクタに対して使われたり、現在のカーソル位置に対してのアクションを指定するコマンドが含まれています。





切り取り、貼り付け

DEL キーで削除されたテキストは「編集」-「元に戻す」コマンドで復元されます。「編集」-「切り取り」、「コピー」、「貼り付け」コマンドはプログラム中のテキストを移動させたり、コピーしたり、挿入したりします。また、他のアプリケーションとの間でもクリップボードを通してテキストをコピーなどができます。



検索と置換

「編集」-「検索」、「置換」コマンドにより、編集中のテキストウィンドウから指定のテキストを検索、置換することができます。検索方向は現在位置から↑方向と↓方向があります。エラー発生時などの際に対象の変数名などの検索に有効です。



ジャンプ

「編集」-「ジャンプ」コマンドにより、編集中のテキストの指定された番号の行へジャンプします。ISaGRAF のコード生成中のエラー表示には、ST・ILプログラムの行番号が含まれるため、エラー箇所の検出に有効です。



変数辞書からの変数(シンボル)挿入

「編集」-「変数の挿入」コマンドまたは CTRL+D により、現カーソル場所にプロジェクトで宣言されている変数辞書から変数(シンボル)を選択して挿入することができます。この変数選択には専用のダイアログボックスが表示されます。詳細は「プログラムエディタ共通項目」を参照願います。



ファイルの挿入

「編集」-「ファイルの挿入」コマンドにより、現カーソル場所にテキストファイルを挿入することができます。ただし、このコマンドでは ASCII 形式のテキストファイル以外のファイルを扱うことはできません。

A.8.2 強調表示機能

このエディタには、変数識別子、定数表現、言語キーワード等を色を変えて表示する機能が追加されました。

[注意]

キーワードのカラーリング表示の設定のカスタマイズはサポートしておりません。

A.8.3 構文の着色

テキストエディタでは言語のキーワードには数種類の色を使用しています。変数定義、定数等。ただし、これら使用されている色の設定は変更することは出来ません。

A.8.4 オプション

「オプション」メニューコマンドには、ツールバーの表示・非表示選択、キーワードの表示・非表示選択、フォントの指定が含まれています。個々で選択されたフォントはワークベンチ上の全てのテキストエディタで有効となります。

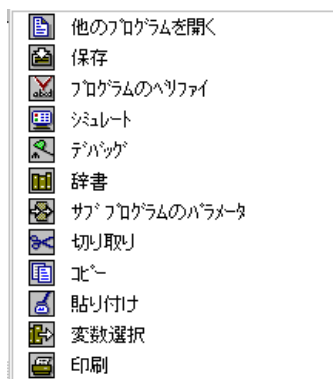
「オプション」-「キーワードの表示」コマンドにより、ST/ILプログラムソースコードの入力の際に、ST、ILプログラムにそれぞれに特有のキーワードツールバーが表示されます。ツールバー上のボタンをクリックすることで現カーソル位置にキーワードを挿入することができます。

ST	
:=	TRUE
FALSE	AND
OR	XOR
RETURN;	IF
THEN	ELSE
ELSIF	END_IF;
CASE	END_CASE;

IL			
TRUE	FALSE	LD	ST
AND	OR	XOR	ADD
SUB	MUL	DIV	LT
LE	EQ	NE	GE
GT	CAL	JMP	RET

A.8.5 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.9 プログラムエディタ共通項目について

この章では、各種 ISaGRAF プログラムエディタに共通の項目を取り上げてまとめて解説します。主に、他の ISaGRAF ツールとの連携や ISaGRAF の持つ共通のダイアログボックスに関する内容が多く含まれています。

A.9.1 他の ISaGRAF ツールのコール



ベリファイ(コンパイル)プログラム

「ファイル」-「ベリファイ」コマンドにより、ISaGRAF コード生成が実行されて、編集集中のプログラム文法チェックを行います。SFCプログラムではレベル1、2の両方がチェックされます。文法チェックが完了したら、コード生成ウィンドウを終了して次の操作に移ります。もし、プロジェクトが現在編集集中の1つのプログラムのみから構成されている場合は、エラーが存在しなければ、アプリケーションコードも生成されます。

「オプション」-「コンパイラオプション」コマンドにより、コンパイラと最適化のオプションを設定できます。なお、これらの詳細は「コード生成の使い方」を参照願います。



アプリケーションのシミュレートとデバッグ

「ファイル」-「シミュレート」、「デバッグ」コマンドにより、ISaGRAF グラフィックデバッガーがシミュレーションモードあるいはオンラインモードで起動します。この際、各プログラムエディタは全てデバッグモードで再オープンされます。このデバッグモードではプログラムの直接の変更はできません。(一度、デバッグモードから抜け出して、プログラムエディタを通常モードで開く必要があります。)詳細は、「グラフィックデバッガの使い方」を参照願います。



変数辞書の編集

「ファイル」-「辞書」コマンドにより、現在編集のプロジェクトの変数辞書エディタを起動します。更に、この辞書エディタからユーザ定義ワードの編集も行うことができます。詳細は、「辞書エディタの使い方」を参照願います。

A.9.2 プログラムのパラメータ定義

編集集中のプログラムがファンクション、ファンクションブロック、サブプログラムの時、「ファイル」-「パラメータ」コマンドにより、プログラムの入出力パラメータを定義することができます。もし、編集集中のプログラムがSFCプログラムや Begin, End セクションのトップレベルのプログラムの時はこのコマンドは無効となります。

ファンクション、ファンクションブロック、サブプログラムは最大32個の入出力パラメータ(合計)を持つことができます。ファンクションやサブプログラムは常に1つの出力パラメータしか持つことができません。この出力パラメータ名は必ずそのプログラム名と一致している必要があります。パラメータ定義のために以下のダイアログボックスが使われます。



パラメータウィンドウの上部にはサブプログラムのパラメータが上から入力パラメータ最後に出力パラメータの順で示されます。下部には現在選択されているパラメータの解説がなされます。ISaGRAF で使用できる変数の型は全てパラメータのデータ型として使用できます。パラメータ名の付け方には以下のルールがあります。

- ・ パラメータ名は半角16文字以内であること。
- ・ 最初の文字は英文字(a-z)であること。
- ・ 以降の文字は英文字、数字、_ であること。
- ・ パラメータ名には大文字小文字の区別はありません。

「挿入」ボタンにより、選択されているパラメータの前に新しいパラメータを挿入します。「削除」ボタンにより、選択されているパラメータを削除します。「アレンジ」ボタンにより、パラメータを自動的にソーティングしてリターンパラメータが最後にくるように並び替えることができます。「OK」ボタンによりサブプログラムのインタフェース定義(即ち、パラメータ構成)を保管してウィンドウを閉じます。「キャンセル」ボタンにより修正内容は保管されずにウィンドウを閉じます。

A.9.3 その他の「ファイル」メニューコマンド

以下に、プログラムエディタの「ファイル」メニューコマンドでその他の共通的な項目をリストアップします。



別のプログラムを開く

「ファイル」-「開く」コマンドにより、現在編集中のプログラムを閉じて、新しく同じ言語の別のプログラムを開くことができます。別の言語で記述されたプログラムを開くことはできません。



プログラムの印刷

「ファイル」－「印刷」コマンドにより、ドキュメントジェネレータが起動して、編集中のプログラムを印刷することができます。この「印刷」コマンドでは、プログラムのリストと内部変数が印刷されます。

SFC、FBD、Quick LDエディタでは、「編集」－「メタファイルとしてコピー」コマンドにより、編集中のダイアグラムをWindowsのメタファイルとしてクリップボードにコピーすることができます。この内容をエディタなどの他のアプリケーションに貼り付けることができます。SFCプログラムにおいてはレベル1（チャート、リファレンス番号、コメント）がその対象になります。

A.9.4 プログラムの修正履歴の更新

「ファイル」－「修正履歴」コマンドにより、修正履歴ファイルの編集を行うことができます。通常、このファイルはコード生成などが行われる際に文法チェックの内容やコード生成された日時などが自動的に追加されるようになっています。

もし、「オプション」－「修正履歴の更新」オプションがセットされていると、変更内容をディスクに保存するたびに、以下のダイアログボックスが開きます。



「OK」ボタンが選択されると、入力されたテキストは修正履歴ファイルに日付・時間スタンプ付きで追加保存されます。この機能はプログラムのメンテナンス上有効に活用できます。

A.9.5 変数辞書から変数選択







ST、ILプログラムを編集中には「編集」－「変数の挿入」コマンドにより、編集中のプログラムの現在のカーソル位置に、辞書に登録されている変数名を挿入できます。Quick LD や FBD プログラムでは、まず、ダイアグラム中の接点、コイルやブロックパラメータなどを配置してから、変数を選択します。

Quick LD では、「編集」－「シンボル／テキストのセット」コマンド（あるいは、シンボルや領域をダブルクリック）により、変数を割り当てます。更に、FBDエディタでは、シンボルをダブルクリックすることにより、ダイアグラム中のシンボルに変数

名を割り当てることができます。いずれの場合も、以下の変数名選択用のダイアログボックスが表示されます。



“スコープ”用の選択コンボボックスでは変数の範囲をグローバル、ローカルから選択できます。右上のコンボボックスでは、変数の型の選択ができます。横の4つのアイコンはよく使う変数の型の選択を行うためのショートカットとして使うことができます。

-  ブール型
-  整数／実数型
-  タイマ型
-  可変長文字列型

変数の選択をするためには、リストの中から変数名を選択(クリック)して、選択された変数名とそのコメントが選択ボックス(リストの上部)に表示されることを確認します。次に“OK”ボタンで、選択した変数が確定されます。尚、変数名をダブルクリックしたり、選択ボックスに直接変数名を入力しても同じ操作が行えます。

A.9.6 アウトプットウィンドウ

全ての言語プログラムエディタのツールメニューから、下記のコマンドが使用できます。これらのコマンドで、各エディタウィンドウ下部のアウトプットウィンドウに情報を表示することが可能です。

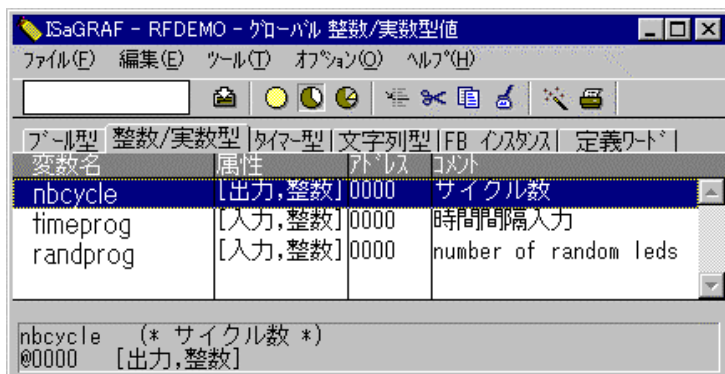
「コンパイラアウトプット」	前回のコード生成時のエラーメッセージをアウトプットウィンドウに表示。
「検索」	編集中のプログラム中の文字列検索を行い、結果のリストを表示します。SFC と FC プログラムに関しては、レベル2のプログラムが検索の対象となります。
「コンパイラアウトプットウィンドウを隠す」	アウトプットウィンドウを終了。

エラーメッセージや検索結果が表示されているときに、その表示されている行をダブルクリックすると、関連する個所にジャンプします。SFC と FC 言語の場合は、関連するレベル2のプログラムウィンドウがオープンします。

A.10 辞書エディタの使い方



辞書エディタにより内部変数、I/O変数、ファンクションブロックのインスタンス、更にアプリケーションの定義ワードを宣言、編集することができます。変数辞書では、通常の変数もファンクションブロックのインスタンスも同じレベルで扱い、定義ワードに関しては、定数として扱います。



変数、ファンクションブロックインスタンス、定義ワードはソースコードなどで使われる前に宣言（登録）されていなければなりません。FBD、Quick LDダイアグラムの中で使われるファンクションブロックは、特にインスタンスを宣言せずに使用できます。（プログラムエディタが自動的にインスタンスを宣言します。）

変数

変数は適用範囲とタイプ毎に分類されます。
変数の範囲は2つあります。



グローバル プロジェクト内のどのプログラムからも使える



ローカル 一つのプログラムからのみ使える

変数のタイプには以下のものがあります。



ブール型 TRUE/FALSE の2つを持つビット値



整数/実数型 整数、実数値



タイマ型 タイマ値



可変長文字列型 可変長文字列

変数は名前、コメント、属性、ネットワークアドレスなどのフィールドがあります。属性には以下の4種類が存在します。




内部	メモリ上に配置される変数
入力	入力デバイスにリンクされたI/O変数
出力	出力デバイスにリンクされたI/O変数
定数	リードオンリーの内部変数(初期値付き)

注意: タイマーは常に内部変数であり、**入力、出力変数**は常にグローバル変数となります。



定義ワード

定義ワードは、プログラム中に配置すると、定義されたテキストに置き換えられる「エリアス」です。全ての言語内で使用することができます。置き換えられるテキストには、変数名、定数、多項式などが定義できます。定義ワードは対象範囲によって分類されています。基本の範囲は以下の3つです。

-  **共通** どのプロジェクトからでも扱えます
-  **グローバル** プロジェクト内のどのプログラムからでも扱えます
-  **ローカル** 一つのプログラムからのみ扱えます

定義ワードは名前と、ST の文法に乗っ取った定義文字列(値)、フリーコメントから成り立っています。



ファンクションブロック インスタンス

テキスト言語であるSTと IL プログラムの中でファンクションブロックを使う場合は、ファンクションブロックのインスタンス(=コピー)を必ず辞書の中で宣言しておかなければなりません。これは、ファンクションブロックは内部にスタティックデータ("hidden"データ)を持っているため、ファンクションブロックのコピーはそれぞれを個別に認識して使用しなければならないからです。

以下にライブラリファンクションブロックの"R_TRIG" (立ち上がり検出)のインスタンスを登録し、それぞれ別の変数のエッジ検出に使われている例を示します。各々のファンクションブロックのインスタンスはユニークな名前で区別される必要があります。ファンクションブロックはライブラリユーティリティで入出力パラメータを宣言しておく必要があります。

ライブラリでの定義:

FB名:	R_TRIG
パラメータ:	Input=CLK
	Output=Q

ファンクションブロックのインスタンスの名前は辞書エディタで定義されます。

辞書でのインスタンスの宣言:

インスタンス名:	TRIG_B1	FB名:	R_TRIG
インスタンス名:	TRIG_B2	FB名:	R_TRIG

辞書エディタの使い方

インスタンスはSTプログラムの中で次のように使われます。

ST言語での使用:

```
TRIG_B1 (b1);  
edge_b1 := TRIG_B1.Q;    (* b1 変数立ち上がり検出 *)  
TRIG_B2 (b2);  
edge_b2 := TRIG_B2.Q;    (* b2 変数立ち上がり検出 *)
```

ファンクションブロックインスタンスは**グローバル**又は**ローカル**として定義されます。FBDや LD 言語においてファンクションブロックを使用する場合は改めて宣言する必要はありません。これは、ISaGRAF のグラフィックエディタが内部で自動的にインスタンスを宣言しているからです。

FBD言語での使用例:



(* ファンクションブロック名は常にライブラリの名称と同じになります。FBDエディタではファンクションブロックがダイアグラムに挿入されるたびにインスタンスを自動的に宣言しています。*)

FBDと Quick LD エディタで自動的に宣言されるファンクションブロックのインスタンスは、常に**ローカル**の定義になります。

ネットワークアドレス

ネットワークアドレスはオプションでつけることができます。変数のネットワークアドレスに0以外の数字を割り当てると、アプリケーション実行中に外部システムから ISaGRAF で使われている変数の状態をネットワークアドレス経由でモニタすることができます。例えば、MODBUSプロトコル経由でSCADAシステムなどから変数の状態をモニタすることが出来ます。つまり、ネットワークアドレスを使用すれば、ISaGRAF の変数名を使用できない外部コミュニケーションシステムからアプリケーションの実行状態をモニタすることが可能になります。ネットワークアドレスは変数辞書で変数の登録／登録内容修正時に設定できます。

A.10.1 辞書エディタメインウィンドウ

辞書エディタのメインウィンドウでは、同一タイプで同一範囲の変数のリストが表示されています。変数のタイプと範囲は常にウィンドウのタイトルバーに表示されています。以下にウィンドウを示します。

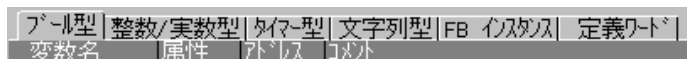


このウィンドウのワークスペースには、変数に関する主なフィールド、即ち、名前、属性、ネットワークアドレス、テキストコメントが表示されます。また、現在選択されている変数のこれらの情報は、ウィンドウ下部のステータスバーにも常に表示されます。また、各フィールドの幅はフィールド間をマウスでドラッグ(✂マークが表示)することで幅変更ができます。

以下のツールバーアイコンによって、変数の範囲(スコープ)を選択できます。

- 共通**ワークベンチ内のどのプロジェクトからでも扱えます
- グローバル**現在のプロジェクト内のどのプログラムからでも扱えます
- ローカル**一つのプログラムからのみ扱えます

下記のタブコントロールによって、変数のタイプを選択できます。



ツールバーアイコン左側のテキスト入力フィールドにテキストを入力することで入力されたテキストから始まる変数名の検索が行われます。検索の対象は表示中の変数全てです。「編集」-「検索」コマンドにより、変数名またはコメントに含まれるテキストを検索し、その変数にフォーカスを移します。検索には**大文字、小文字の区別はありません**。

A.10.2 変数の管理



「ファイル」メニューのコマンドは選択中のグループの変数全てに対して機能します。「ファイル」-「その他」コマンドでは、選択中の変数の範囲やタイプを変更することができます。

**変数の印刷**

「ファイル」-「印刷」コマンドにより、現在表示されている範囲(スコープ)の変数(又は定義ワード)のリストが印刷できます。印刷はISaGRAFのドキュメントジェネレータ経由で行われます。印刷項目には変数に定義した内容全てが含まれます。

**変数の新規作成**

「編集」-「新規作成」コマンドにより、新しく変数、ファンクションブロックのインスタンス、定義ワードを作成することができます。新しい変数はリスト上では現在選択中の変数の前に挿入されます。このコマンドが選択されると、変数定義のダイアログボックスが開きます。必要な事項を記述、選択の後、「保存」ボタンでリストに挿入されます。「キャンセル」ボタンで変数宣言をキャンセルして戻ります。



変数の編集

「編集」―「編集」コマンドまたは ENTER キー、ダブルクリックにより、リストの中で選択中の変数の記述の修正を行なうことが出来ます。「新規作成」コマンドと同様に変数の記述の入力を行ないます。“保存”、“キャンセル”ボタン以外に“次へ”及び“前へ”ボタンがあり、引き続き変数の修正を行なうことが出来ます。



「切り取り」、「貼り付け」

ISaGRAF 辞書エディタでは**複数行の同時選択**が可能であり、選択された内容に対して以下のコマンドも扱うことが出来ます。

コピー.....選択された複数の変数を辞書のクリップボードへコピーします。
 切り取り.....選択された複数の変数を辞書のクリップボードへコピーし、編集リストから削除します。
 削除.....選択された複数の変数を編集リストから削除します。
 貼り付け.....辞書のクリップボードの内容をリストに挿入します。

コピー／切り取り／貼り付けコマンドは同一タイプ、属性の変数間での操作に限られます。



変数の並び替え

「ツール」―「並べ替え」コマンドにより、編集中的変数や定義ワードを並び替えることが出来ます。並び替えのキーは変数の属性となります

- 最初は内部変数
- 次は入力変数
- 最後に出力変数

なお、同一属性内の変数はアルファベット順になります。ただ、定義ワードは常にアルファベット順となります。



ネットワークアドレスの設定

ネットワークアドレスは、オプションの設定です。0以外のアドレスを設定すると外部のシステム(HMI のシステム等)から、この変数を監視することが出来ます。ネットワークアドレスは、変数登録時にそれぞれに設定しなければなりません。「ツール」―「アドレスの取り直し」コマンドにより、ネットワークアドレスの番号付けを自動で行えます。このコマンド実行の前に番号を付けたい変数をリスト中から事前に選択しておく必要があります。16進数で**ベースアドレス**(グループの先頭アドレス)を入力することで、選択された変数が**連続番号**で割り付けられます。ベースアドレスに0を入力することで選択中の変数のネットワークアドレスは全て0にセットされます。



"true/false" 定義のインポート

定義ワードの編集時に「ツール」-「TRUE/FALSE 定義のインポート」コマンドにより、ブール型変数に割り付けられた TRUE, FALSE 状態の「値」の設定を定義ワードとして置き換えることができます。これらの設定は(TRUE, FALSE と定義された)は主にデバッグ時に使用されます。プログラム中にこれらの値が直接使われるのであれば定義ワードとして用意しておく必要があります。このコマンドは、同じ範囲(スコープ)に属する宣言済みのブール型変数の中で「値」に設定された TRUE/FALSE 文字列から TRUE/FALSE 定義ワードを作成します。

A.10.3 オブジェクトの記述

辞書オブジェクト(変数、ファンクションブロックインスタンス、定義ワード)の記述はもれなく行なわれる必要があります。記述のフィールドは個々のオブジェクトのタイプによって異なります。

以下に各変数型で共通のフィールドを示します。

名前 変数名のことで、最初の文字は半角英字(a-z)でのこりは半角英数字、_ のいずれかである必要があります。

ネットワークアドレス

..... 16進数のネットワークアドレスはオプション扱いです。ゼロ以外の数字が入力された変数はターゲットが実行時に外部のSCADAシステムからのモニタが可能です。

コメント 変数のフリーコメント領域

保持 保持指定はバッテリーバックアップされたメモリに対してスキャン毎に、変数を保管することを意味します。(入力、出力変数にはこの保持指定は出来ません。)



ブール型変数に必要なフィールド

属性 内部、入力、出力、定数のいずれかを設定

"False" 文字列 デバッグ時に FALSE 状態の時に表示される文字列

"True"文字列 デバッグ時に TRUE 状態の時に表示される文字列

初期化時に True このオプションをチェックすると、初期値を TRUE に設定できます。設定をしないと、初期値は FALSE になります。



整数型／実数型変数に必要なフィールド

名前: ネットワークアドレス:

コメント:

単位: 変換:

属性

- ☐ 内部(i)
- ☐ 入力(I)
- ☒ 出力(O)
- ☐ 定数(a)

フォーマット

- ☒ 整数(g)
- ☐ 実数型(r)

初期値:

☐ 保持(e)

保存(S) キャンセル(C) 次へ(N) 前へ(P)

属性 内部、入力、出力、定数のいずれかを設定

フォーマット 整数または実数の選択。このフォーマットはデバッグ時にも活用できます

ユニット デバッグ時に使用する値の「単位」を示す文字列

変換 入力・出力変数に設定する変換関数、変換テーブルの名前

初期値 変数の初期値（整数、実数のフォーマットにあわせる必要があります） 例：整数→0 実数→0.0。特に定義をしないと、0 になります。



タイマ型変数に必要なフィールド

名前: ネットワークアドレス:

コメント:

属性

- ☒ 内部(i)
- ☐ 定数(a)

初期値:

☐ 保持(e)

保存(S) キャンセル(C) 次へ(N) 前へ(P)

属性 内部、定数の設定

初期値 変数の初期値 特に設定しないと time#0s となります。



可変長文字列型変数に必要なフィールド

属性 内部、入力、出力、定数のいずれかを設定。

最大長 最大文字列長を指定します

初期値 変数の初期値



定義ワードに必要なフィールド

名前 STステートメントに使われる名称、最初の文字は半角英字 (a-z) でのこりは半角英数字、_ のいずれかである必要があります。

定義 ST言語の文法にあった文字列で定義ワードをコンパイル前にこの文字列で置き換えます。(例: 名前 = PI, 定義 = 3.14159)

コメント 自由なコメント



ファンクションブロックインスタンスに必要なフィールド

- 名前** インスタンス名、STステートメントで使われます。最初の文字は半角英字(a-z)でのこりは半角英数字、_のいずれかである必要があります。
- タイプ** 対応するライブラリ内のファンクションブロック名
- コメント** 自由なコメント

A.10.4 クイック変数登録

「ツール」-「クイック変数登録」で、複数の変数を一度に定義できます。クイック変数登録で作成された変数には通し番号がつけられます。以下の項目を設定してください。

- 変数につける最初と最後の番号
- 変数名の頭と終りに付ける文字列
- 変数名に使用される数字の桁数

さらに、変数の基本属性(内部、入力、出力)と変数タイプによる固有の情報(整数・実数型フォーマットや変数の保持、可変文字列の文字列最大長など)も設定できます。

変数名は数字で始めることはできないので、必ず変数名の頭に文字列を設定しなければいけません。「桁数」(数字の文字数)が"auto"になっていたら、自動的に最小の文字数が割り当てられます。「桁数」が設定されると、頭から"0"を付加して桁数があわせられます。このように、変数名の文字数が固定になるようにしておくと、ソート時に便利です。以下に使用例を上げます：

例: 文字数が変わる例：

ナンバリング: —

From: To:

桁数:

シンボル: —

名前: ##

この設定では、以下の3つの変数ができます：

Var9xx Var10xx var11xx

例: 文字数が固定になる例

ナンバリング: —

From: To:

桁数:

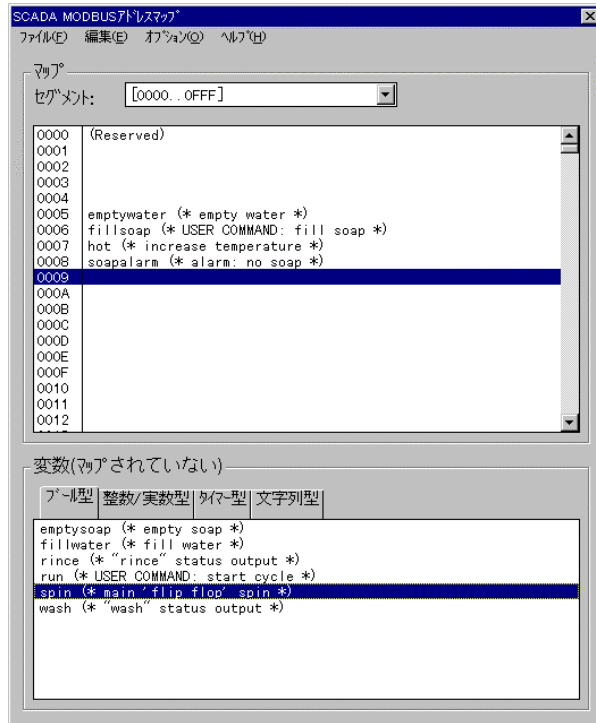
シンボル: —

名前: ##

この例では、**MyVar001** から **MyVar100** までの100個の変数が作成されます。

A.10.5 SCADA Modbus アドレスマップ

「ネットワークアドレス」は、ISaGRAF と SCADA ソフトとの間などで、Modbus プロトコルを使用して通信する場合に使われます。この場合、SCADA ソフトの方がマスタになり、ISaGRAF ターゲットがスレーブになります。ネットワークアドレスは、SCADA 側からコントロールされる変数の仮想マッピングとして使用されます。「ツール」—「SCADA Modbus アドレスマップ」を使うと、各変数を簡単にマッピングできます。



SCADA Modbus アドレスマップダイアログボックスには2つのリストがあります。上のリストは1セグメント(0~*FFF=4096 個)分のマッピング情報で、ネットワークアドレスが割り付けられている変数が表示されます。下のリストには、マッピングされていない変数が表示されます。ネットワークアドレス"0"には変数をマッピングできません。

「編集」-「選択変数のマップ」と「マップから削除」で変数をリスト間で移動して、マッピングを行います。リストされている変数をダブルクリックしても同様の処理が行えます。「セグメント」ドロップダウンリストボックスから他のセグメントのリストに切り替えることができます。

「オプション」メニューで、アドレスの10進/16進表示を切り替えます。

「編集」-「検索」メニューで、変数がマッピングされているかどうか、検索することができます。

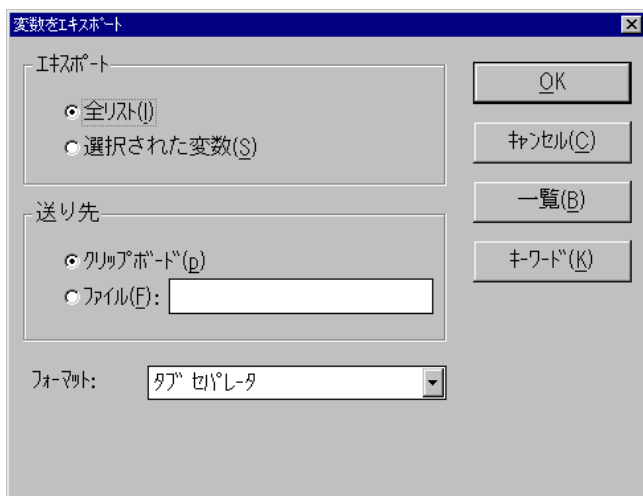
A.10.6 他のアプリケーションとのデータ交換

ISaGRAF 辞書エディタには他のアプリケーション(ワープロ、スプレッドシート、データベースなど)との情報交換の目的でインポート/エクスポートの機能を備えています。「ツール」-「テキストのエクスポート」コマンドにより、変数リストをASCII テキスト記述に変換しクリップボードあるいはファイル形式で保管します。「ツール」-「テキストのインポート」コマンドにより、アスキーテキストフォーマット

でクリップボードあるいはファイルに保管された内容から変数宣言内容のフィールドを取り出します。

「テキストのエクスポート」コマンド

「テキストのエクスポート」コマンドによりエクスポートテキストダイアログボックスが開かれます。ここでエクスポートの手段を設定します。



“全リスト”をオンにすることで現在選択されている変数は無視され、変数リスト全部がエクスポートされます。“選択された変数”をONにすることで選択された(ハイライト部分)変数のみがエクスポートされます。

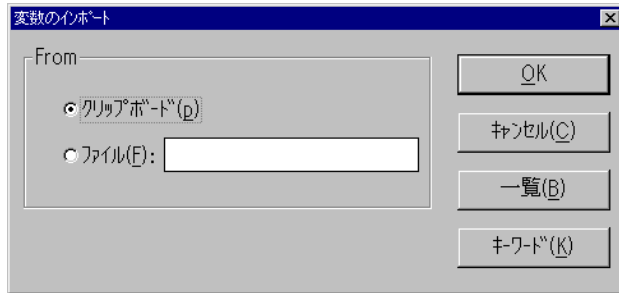
“クリップボード”をオンにすることでエクスポートされる情報はアスキーテキストフォーマットで Windows のクリップボードに保管されます。このあと“ペースト”コマンドで他のアプリケーションへ取り出すことが出来ます。“ファイル”をオンにすることでエクスポートされる情報はテキストファイルとして保管されます。この際、フルパスでファイル名を指定する必要があります。“一覧”ボタンで既存のファイルを指定することもできます。

更に、エクスポートするアスキーテキストのフォーマットを選択します。フォーマットの種類については後述します。“OK”ボタンでエクスポートされ、“キャンセル”ボタンでエクスポートせずにダイアログボックスを閉じます。

選択された全ての変数の全てのフィールドが、登録時の順番で、テキストにエクスポートされます。データの最初の行はフィールド名を表わしています。個々の変数オブジェクトデータは1行のテキストで表現されています。行の終わりはMS-DOS標準の“0d-0a”となります。最初の行のフィールド名と呼ばれる項目は“キーワード”ボタンを選択することで変更が可能です。“キーワード”に関しては後で述べます。何もしない場合はデフォルトのフィールド名が付いています。

「テキストのインポート」コマンド

「テキストのインポート」コマンドによりインポートテキストダイアログボックスが開きます。ここでインポート手段を設定することが出来ます。



“クリップボード”をオンにすると、インポートされる情報はアスキーテキストフォーマットの状態で Windows のクリップボードから取り出されます。“ファイル”をオンにすることでインポートされる情報は ASCII ファイルから読み出されます。この際、フルパスでファイル名を指定する必要があります。“一覧”ボタンによってブラウザから直接ファイルを指定することもできます。

更に、インポートアスキーテキストで使われているセパレータは自動的に認識されて取り出されます。フォーマットの種類については後で述べます。“OK”ボタンでインポートがなされ、“キャンセル”ボタンでインポートはなされずにダイアログボックスを閉じます。

インポートされるデータの最初の行にフィールド名が順に設定されていなければなりません。また、個々の変数オブジェクトデータは1行のテキストで表現されています。行の終わりはMS-DOS標準の“0d-0a”となります。フィールド名の並んでいる順は特に指定はありません。もし、フィールドが抜けていた場合はそのフィールドのデータにはデフォルト値が入ります。もし、インポートされるデータが既に変数辞書のリストの中に存在している場合は、上書きするかどうかをチェックする必要があります。この際に抜けているフィールドがあればそのフィールドのデータは更新されません。最初の行のフィールド名は“キーワード”ボタンを選択して変更が可能です。“キーワード”に関しては後で述べます。何もしない場合はデフォルトのフィールド名が付いています。

テキストフォーマットの例

以下に、エクスポートコマンドにより作られるテキストファイルの例を示します。インポートコマンドではフォーマットは自動認識されます。

• タブ セパレータ

説明: フィールド間がタブで区切られます。

例:	Name	Attribute	Comment
	level	internal	internal calculated water level
	alarm1	output	main alarm output

• コンマ セパレータ

説明: フィールド間がコンマで区切られます。

例:	Name,Attribute,Comment
	level,internal,internal calculated water level

alarm1,output,main alarm output

- セミコロン セパレータ

説明: フィールド間がセミコロンで区切られます。

例: Name;Attribute;Comment
level;internal;internal calculated water level
alarm1;output;main alarm output

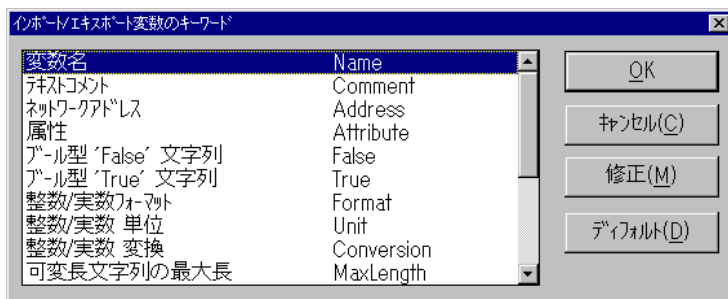
- コンマとダブルクォーテーション

説明: フィールド間がコンマで区切られ、
各フィールドはダブルクォーテーションで囲まれます。

例: "Name","Attribute","Comment"
"level","internal","internal calculated water level"
"alarm1","output","main alarm output"

キーワード

インポート、エクスポートデータの最初の行はフィールド名と呼ばれ、変更することが可能です。フィールド名は、キーワードに登録されています。なお、“**キーワード**”ボタンで“キーワード”ダイアログボックスが開き、ここで変更することが出来ます。



キーワードダイアログボックスではオブジェクトとキーワードが関連づけられています。キーワードを変更する場合は変更したいフィールドを選択して“**修正**”ボタンを選択します。ここで新しいキーワードを入力します。“**デフォルト**”ボタンは元のキーワードに変更します。このキーワードの付け方には以下のルールがあります。

- 半角16文字以内
- 最初の文字は半角英文字(a-z)
- 続く文字は英文字、数字、_ のいずれか
- 同一のキーワードは異なるフィールドには適用出来ません。

以下に ISaGRAF のデフォルトのキーワード(太字体)を示します。

変数名	Name
テキストコメント	Comment
ネットワークアドレス	Address
属性(内部、入力、出力)	Attribute
ブール型 'False' 文字列	False
ブール型 'True' 文字列	True
整数／実数フォーマット	Format
整数／実数単位	Unit
数値変換名	Conversion
文字列最大長	MaxLength
FBライブラリタイプ	Library
公式の定義	Equivalence
内部属性	Internal
入力属性	Input
出力属性	Output
定数属性	Constant
実数型フォーマット	Real
整数型フォーマット	Integer

A.10.7 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.11 I/O接続エディタの使い方



I/O接続エディタにより、アプリケーション内のI/O変数(論理)とターゲットマシン上のI/Oボードチャンネル(物理)をリンクすることができます。このためにはユーザーターゲット側のI/Oボードの選択・設定をし、チャンネルにI/O変数を割り当てていきます。

I/O接続エディタのウィンドウを以下に示します。






I/O接続エディタの左側にはターゲットのボードスロットがあるラックを示しています。スロットは空きスロットでもよいし、単一のI/OボードあるいはI/O装置機器を設定することができます。各スロットには番号が付いています。ラックにはシングルI/Oボード、I/O装置機器に含まれるボードを含めて、**最大255個**のボードを設定することができます。右側にはボードのパラメータやボードのチャンネルに接続されている変数を表わしています。一枚のボードには**最大128個**のI/Oチャンネルを割り付けることができます。

アイコン




ボードの右側に表示されているアイコンはボードのチャンネルに接続される変数のタイプと属性を示します。ISaGRAF では一つのボードに異なったタイプの変数を接続することは出来ません。以下にアイコンとその意味を示します。

- ⬇ ブール型タイプ
- ⬆ 整数型/実数型タイプ(両方のフォーマットが使用できます。)
- xyz 可変長文字列型タイプ
- ⇐ 入力-全チャンネルが接続されていない状態
- ⇒ 出力-全チャンネルが接続されていない状態
- ⇨ 入力-1つ以上のチャンネルが接続されている状態
- ⇩ 出力-1つ以上のチャンネルが接続されている状態

以下にスロットに入るI/Oボード／デバイスのタイプを表わすアイコンを示します。

I/O装置機器 (複数のI/Oボードがまとまっている)
実I/Oボード
バーチャルI/Oボード

以下にボードパラメータやチャンネルを表わすアイコンを示します。

ボードパラメータ
開放チャンネル
接続チャンネル



I/Oボードの移動

「編集」－「ボードを上に移動」、「ボードを下に移動」コマンドにより、選択中のI/Oボードを上、下に移動することができます。ツールバーアイコンからも操作が可能です。「編集」－「スロットの挿入」コマンドにより、選択中のI/Oボードの直前に空のI/Oスロットを挿入できます。

A.11.1 I/Oボードの定義

「編集」メニューには選択されたボードを定義したり、ボードのチャンネルにI/O変数を割り当てたりするコマンドが含まれています。



I/Oボードタイプの選択

ボードにI/O変数を割り付ける前に、ボードの選択を行います。I/O接続でボードを設定するためには、前もってワークベンチのライブラリにI/Oボードを登録しておく必要があります。このI/Oボードドライバは通常、デバイスのサプライヤより提供されます。「編集」－「ボード／装置機器セット」コマンドにより、使用するボードを選択することが出来ます。ここでは ISaGRAF ライブラリから単一ボードあるいは複数のボードがまとまった装置機器を選択することが出来ます。目的のスロットをダブルクリックすることでも同様にボード選択が可能です。

一つのボードの全てのチャンネルは同一タイプ（プル型、整数／実数型、可変長文字列型）と同一方向性（入力、出力）を持ちます。実数アナログ、整数アナログの区別はI/O接続時には考慮されません。I/O装置機器の場合は異なるタイプと方向性をもつチャンネルを含むことになります。表現方法としては複数のシングルI/Oボードの組み合わせとなります。ただし、ラックの占有スロット数は1つとなります。



スロットのクリア

「編集」－「スロットのクリア」コマンドにより、選択されているスロットに割り当てられているボードを削除することが出来ます。既にそのボードのチャンネルにI/O変数が接続されている場合はそれらは全て自動的に解放（切り離し）されます。



実ボードとバーチャルボード

「編集」メニューの「実<→>バーチャルボードのスイッチ」コマンドにより選択されたボードのモードを切り替えることができます。



..... 実I/Oボード



..... バーチャルI/Oボード

実モードではI/O変数はダイレクトに接続されているI/Oデバイスにリンクされています。即ち、アプリケーションプログラムでの入/出力操作は、実I/Oデバイス状態に対応しています。

バーチャルモードではI/O変数はまさに内部変数として振る舞います。デバッガーによりモニタしたり更新したりすることでI/O処理のシミュレーションを行ないます。実I/Oデバイスへの対応はしません。



技術メモ

「ツール」-「技術メモ」コマンドにより、選択されたI/Oボード・装置機器の技術メモが開きます。技術メモは、通常I/Oボード(ドライバ)のサプライヤによって記述されます。ここにはパラメータの意味をはじめI/Oボード管理に関する必要な情報が全て含まれています。



ボードチャネルの解放

「ツール」-「ボードチャネルの解放」コマンドにより、選択されているスロットのI/Oボードに接続されている全I/O変数を解放します。



I/Oチャネルのコメント

I/O変数のコメントは変数名の後に(* *)のフォーマットでI/Oチャネルに表示されます。ISaGRAF は%?.?.?フォーマットで表現する、直接表現変数(D.R.V)も扱うことができます。このため、変数名を割り付けてないI/Oチャネルもプログラム中で扱うことができます。この場合のコメントはI/O接続エディタで入力することができます。

A.11.2 ボードパラメータの設定

ボードパラメータをセットするには、ウィンドウ右側上部のボードパラメータアイコン(I/Oボードによっては存在しない場合もあります。)をダブルクリックします。あるいは「編集」-「ボードパラメータ/チャネルのセット」コマンドにより、行なうこともできます。ボードパラメータのアイコンを以下に示します。





..... ボードパラメータ

このパラメータアイコンはI/Oボードのドライバを開発時に必要なデータ(例えばボードのI/Oアドレスなど)をユーザが設定するためのものであり、全I/Oボードに対して存在しているものではありません。(詳細はI/Oボードの技術メモまたはハードウェアのドキュメントを参照願います。)

A.11.3 I/O変数とI/Oチャンネルの接続

I/Oチャンネルの接続を行う為には、接続するチャンネルをダブルクリックするか、チャンネルを選択してから「編集」-「ボードパラメータ/チャンネルのセット」コマンドを実行します。以下のアイコンは、チャンネルの状態を表しています

-  開放チャンネル
-  接続済みチャンネル

I/Oチャンネル接続ダイアログボックスではそのチャンネルのタイプと方向性が一致したI/O変数でまだ接続されていないものが選択肢としてリストされます。“接続”ボタンにより選択されている変数をそのチャンネルに接続することが出来ます。“解放”ボタンにより既に接続されている変数をチャンネルから解放(切り離す)することが出来ます。“次へ”、“前へ”ボタンは同一ボードの中で別のチャンネルを選択する場合に使います。ダイアログボックスのタイトルには常に選択されているチャンネルNo. が表示されています。

A.11.4 直接表現変数 (Directly Represented Variables)

I/O変数が接続されていないI/Oチャンネルはフリーチャンネルと呼ばれます。ISaGRAF の **直接表現変数**により、このフリーチャンネルをプログラムのソースコードの中で直接扱うことができます。直接表現変数の識別子は必ず“%”キャラクタで始まります。

以下に、単一のI/Oボードに対する直接表現変数の名前の付け方について示します。ここで、“s”はボードのスロット番号を示します。“c”はI/Oチャンネル番号を示します。

```
%IXs.c ..... ブール型入力ボードのフリーチャンネル
%IDS.c ..... 整数型入力ボードのフリーチャンネル
%ISS.c ..... 可変長文字列型入力ボードのフリーチャンネル
%QXS.c ..... ブール型出力ボードのフリーチャンネル
%QDS.c ..... 整数型出力ボードのフリーチャンネル
%QSS.c ..... 可変長文字列型出力ボードのフリーチャンネル
```

以下に、複数のI/Oボードから成り立っているI/O装置機器に対する直接表現変数の名前の付け方について示します。ここで、“s”はボードのスロット番号を示します。“b”はI/O装置機器の中でシングルボードのインデックス番号を示します。“c”はI/Oチャンネル番号を示します。

```
%IXs.b.c ..... ブール型入力ボードのフリーチャンネル
%IDS.b.c ..... 整数型入力ボードのフリーチャンネル
%ISS.b.c ..... 可変長文字列型入力ボードのフリーチャンネル
%QXS.b.c ..... ブール型出力ボードのフリーチャンネル
%QDS.b.c ..... 整数型出力ボードのフリーチャンネル
%QSS.b.c ..... 可変長文字列型出力ボードのフリーチャンネル
```

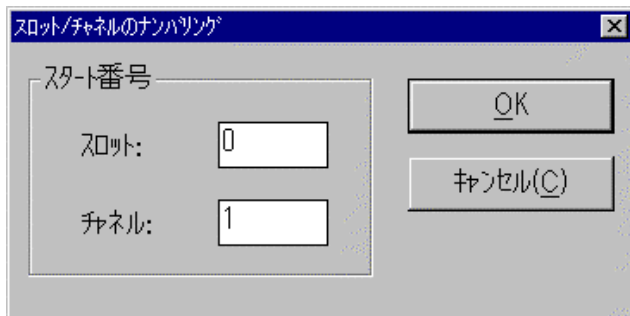
以下にこれらの例を示します。

- %QX1.6 ボードスロット番号1、I/Oチャンネル番号6のプール型出力カード
%ID2.1.7 ボードスロット番号2のI/O装置機器で、ボード番号が1、I/Oチャンネル番号7の整数型入力ボード

直接表現変数は**実数型**のデータタイプを持つことはできません。

A.11.5 ナンバリング

メニューの「オプション」-「スロット／チャンネルのナンバリング」で番号の振り方を設定します。下記のダイアログボックスで、スロットの開始番号と各ボードごとのチャンネルの開始番号を設定できます。



デフォルト値としてスロット番号には“0”、チャンネル番号には“1”が設定されています。

注意： 直接表現変数に使用されているシンボルに影響を与え、直接表現変数を使用している既存のプログラムでコンパイルエラーを起こす場合があるので、番号の変更には十分に注意してください。

A.11.6 チャンネル単位のプロテクション設定

ISaGRAF のワークベンチでは、階層化パスワードによるプロテクション機能を提供しています。I/O接続に関しては、パスワードによってグローバルにプロテクトをかけることができます。それに加えて、ISaGRAF では、I/Oボードのチャンネル単位にもプロテクトをかけることができます。

チャンネル単位でプロテクションをかける場合、プロジェクト管理エディタの「プロジェクト」-「パスワードセット」ですでにパスワードが設定されていることが前提となります。また、グローバルなI/Oプロテクションよりもチャンネル単位の方に高いレベルのパスワードを設定する必要があります。

I/Oチャンネル単位のプロテクションをかけている場合は、I/O接続エディタで、そのチャンネル名の前に小さなアイコンが表示されます。

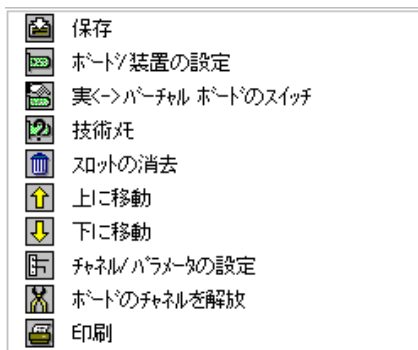


「編集」-「チャンネルのプロテクション設定」と「チャンネルのプロテクションの削除」で個別チャンネルにプロテクションの設定と削除ができます。どちらのコマンドの場合も、チャンネルにあるレベルのプロテクションをかけるために、それにあった適切なパスワードを入力します。従って、個別プロテクションがかけられているチャンネルの接続を変更する都度、適切なプロテクションレベルのパスワードを入力しなくてはなりません。

注意: チャンネルがプロテクションをかけられていて、対応するパスワードが削除された場合、それより高いレベルのパスワードが定義されていないと、そのチャンネルは、十分に高いレベルのパスワードが新たに定義されない限り変更することができなくなります。

A.11.7 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.12 数値変換テーブルの作成



ユーザは数値変換テーブルを作成することができます。変換テーブルはアナログ値を変換する点の集合です。このテーブルは整数・実数型入力及び出力変数に割り付けることができます。電気的な信号（センサから情報）と論理的な信号（アプリケーションで使用する値）間の変換を定義します。変換値は単調増加あるいは単調減少のいずれかかである必要があります。

変換テーブルは辞書ウィンドウの「ツール」-「変換テーブル」メニューで開くダイアログボックスから編集することができます。



定義済み変換テーブルは、整数/実数型の入出力属性の変数に割り付けられ、入出力データのフィルタとして使用することができます。変数に変換テーブルの属性を付けるには変数辞書エディタのダイアログを使用します。

A.12.1 変換テーブルのメインコマンド

辞書エディタの「ツール」-「変換テーブル」ダイアログボックスでは、変換テーブルのリストと、既存の変換テーブルを編集したり、新しいテーブルの作成、テーブル名を変更するためのメインコマンドボタンがあります。変換テーブルダイアログボックスを終了して、編集内容をセーブするためには、「OK」ボタンを押してください。



テーブルの新規作成

「**新規作成**」コマンドにより、新しいテーブルを作成することが出来ます。1つのプロジェクトあたり最大**127個**の変換テーブルを作成することが出来ます。作成された変換テーブルの中から実際に整数・実数型の入出力変数にアタッチされたテーブルのみがアプリケーションコードの中に組み込まれます。変換テーブルの名前の付け方には以下のルールがあります。

- 最大16半角文字
- 最初の文字は英文字(a-Z)
- 2番目以降は英文字、数字、_ のいずれか
- 大文字小文字の区別なし



テーブルの修正

「**編集**」コマンドにより、変換テーブルリスト中の選択されたテーブルの内容を修正できます。テーブル名をダブルクリックして起動することもできます。「編集」コマンドは新規テーブルが作られた時は自動的に起動されます。ここでは変換テーブルに最低でも2点以上の変換ポイントを入力します。

A.12.2 テーブルへの変換ポイント入力

「**編集**」コマンドにより開かれたダイアログボックスで、新たに変換テーブルのポイントを入力します。左側のボックス中のポイントリストは既に入力済みのポイント、右下のボックスには定義中のテーブルを折れ線グラフで表示します。ポイントは数値入力用ボックスに数値を記入することで定義されます。数値入力にはルールがありますがこの節の後半で説明します。

変換テーブル 'conv1'

ポイント:

10	10
15	15
20	30
30	35
40	70
50	75
...	

Electrical:

Physical:

OK

キャンセル(C)

保存(S)

削除(I)

数値変換テーブルの作成

左側のボックスのポイントリストは既に入力済みのポイントを示しますが、左の列は電気信号（外部）数値を、右の列はアプリケーション（内部）数値を示します。入力済みのポイントの修正や削除の場合はポイントを選択する必要があります。右下のボックスには、現在編集中的のテーブルを折れ線グラフ表示しています。軸には単位も座標もありません。ただ、定義したテーブルがこのグラフによって正しく入力されているか簡単にチェックすることが出来ます。

新しいポイントの入力

新しいポイントを入力する場合はポイントリストの最後 ("... ...")を選択します。ここで、外部データと内部データの対応を入力します。数字は単精度の浮動小数点で入力されます。**最低でも2点ポイント**の入力が必要です。外部・内部データの入力の後に**“保存”**ボタンを選択してテーブルにポイントの追加を行います。**最大32点ポイント**がそれぞれの変換テーブルに入力が可能です。

ポイントの修正

入力済みのポイントを修正する場合はまず変更するポイントをリストから選択します。そして、新しく内部・外部データ対を入力して**“保存”**ボタンを選択します。

ポイントの削除

削除したいポイントをリストから選択して**“クリア”**ボタンを選択します。ただし、削除した後でも、**最低でも2点ポイント**が残っている必要があります。

A.12.3 ルールと制限

変換テーブルを定義する際のルールを以下に示します。あくまでも変換テーブルは整数・実数型入出力変数の変換に使われます。

- 一つの外部（電気信号）データに対して2点以上の内部データを定義は出来ません。
- テーブルから作られるグラフは単調増加又は単調減少のいずれかである必要があります。
- 一つの内部データに対して2点以上の外部（電気信号）データを定義は出来ません。

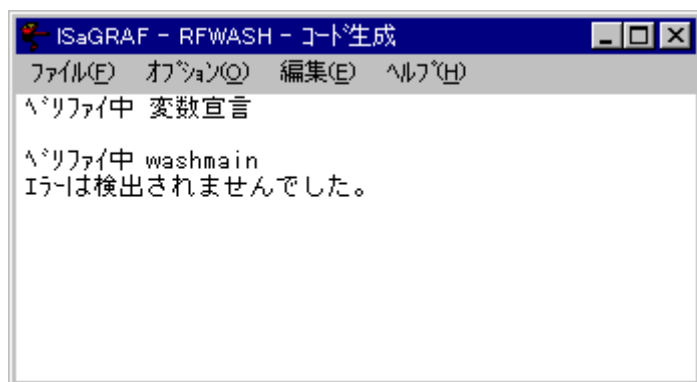
一つのプロジェクトに対して変換テーブルには以下の制限があります。

- 最大の変換テーブル数は1プロジェクトあたり**127個**です。
- 一つの変換テーブルの定義には最大**32点**ポイント入力が行えます。

A.13 コードジェネレータの使い方

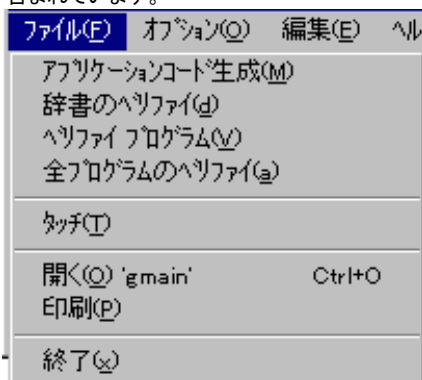


各種 ISaGRAF ワークベンチのウィンドウ「アプリケーションコード生成」あるいは「ベリファイ」コマンドを選択すると、アプリケーションコード生成ウィンドウが自動で開きます。コードが生成されたあとはこのウィンドウは自動では閉じません。その間ユーザは以下のコードジェネレーション（生成）のメニューにアクセスすることができます。



A.13.1 「ファイル」コマンドメニュー

「ファイル」メニューにはプログラムの文法チェックやアプリケーションコード生成のためのコマンドが含まれています。



アプリケーションコードの生成

「ファイル」-「アプリケーションコード生成」コマンドにより、アプリケーションコード（中間コード、TICコード）の生成を行ないます。プロジェクトコード生成の前に、単体のプログラムのコンパイルでは検出できない辞書や宣言の文法チェック（ベ

リファイ)を行ないます。内容としては、数値変換テーブル、I/O接続の処理、更にライブラリとのリンクを行ないます。これらの途中にエラーが発生した場合はエラーを修正しない限り次のステップには進めません。

既にプログラムのベリファイがエラーなしで行なわれていて、その後プログラムの修正や辞書修正が行なわれていない場合には、プログラム単体のベリファイのステップは省略されます。変数宣言及び、プログラムのコヒーレントチェックは毎回行なわれます。「アプリケーションコード生成」コマンド実行中に処理を中断したい場合は**ESCキー**を使います。

注意: プログラムのローカル変数が変更された場合はそのプログラムはベリファイされます。グローバル変数が変更された場合は全プログラムがベリファイされます。

■ プログラムの文法チェック

「ファイル」-「プログラムのベリファイ」コマンドにより、選択されたプログラムのみがベリファイされます。たとえ、プログラムが最後の修正時から変更が加わっていない場合でもベリファイされます。

「ファイル」-「辞書のベリファイ」コマンドにより、プロジェクトの全辞書の宣言内容の文法チェックが行われます。

「ファイル」-「全プログラムのベリファイ」コマンドにより、プロジェクト内の全プログラムがベリファイされます。たとえ、最後の修正時から変更が加わっていない場合でも全プログラムがベリファイされます。このベリファイはエラーが発生しても途中で中止されません。全プログラムでのベリファイを行ないエラーリストを作成することができます。ただし、ベリファイを中断する場合は**ESCキー**を使います。

■ 完全な再コンパイル(再コード生成)

「ファイル」-「タッチ」コマンドにより、次の「アプリケーションコード生成」コマンドの時に全プログラムがベリファイされるように擬似的に全てのプログラムの修正が加わったことにします。

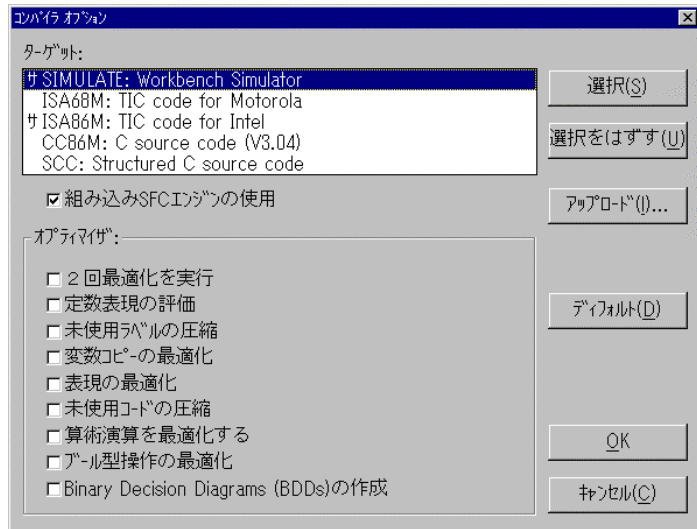
「ファイル」-「開く」コマンドにより、最後に「ベリファイ」されていたプログラムを開きます。

エラー発生時にはエラーメッセージをダブルクリックすることにより、エラーのプログラムを開き、エラー箇所までカーソルを移動させます。

A.13.2 コンパイラオプション

アプリケーションコード生成ウィンドウの「オプション」-「コンパイラオプション」コマンドまたはプログラム管理の「コード生成」-「コンパイラオプション」により、作成するアプリケーションコードや、ターゲットコードを最適化するためのパラメータ設定ができます。生成されるターゲットコードのタイプの選択、最適化パラメータの設定があります。

「アップロード」ボタンでダイアログボックスが開き、アップロード用にアプリケーションコードにソースコードを圧縮してダウンロード時に添付できます。アップロード機能の詳細については、後程説明します。



ISaGRAF ターゲットの選択

ダイアログボックス上部のリストはターゲットコードの種類を示します。(サ)マークは選択されたターゲットコードを示します。ISaGRAF コードジェネレータは一回のコンパイルで最大3種類のコードを生成することができます。“**選択**”、“**選択をはずす**”ボタンで生成したいターゲットコードを選択します。以下に標準で選択可能なターゲットコードを示します。

- SIMULATE:** このコードはワークベンチ上での ISaGRAF シミュレーション専用コードです。シミュレーション前にこの選択がされてシミュレーション用コードを生成しておく必要があります。
- ISA86M:** このコードは**Intel**ベースのCPUプロセッサ上で実行されるターゲット用の**TIC** (Target Independent Code)コードです。プロセッサタイプはバイトの並びのみを考慮しています。
- ISA68M:** このコードは**Motorola**ベースのCPUプロセッサ上で実行されるターゲット用の**TIC** (Target Independent Code)コードです。プロセッサタイプはバイトの並びのみを考慮しています。
- SCC:** このコードを選択すると、ISaGRAF コンパイラはプロジェクトのプログラムごとに構造化Cソースコードを生成し (appli.c, appli.h と個別プログラムの xxx.c, xxx.h ファイル)、ターゲットのライブラリとコンパイル/リンクすると、コンパイルモードの ISaGRAF ターゲットの実行モジュールを生成することができます。構造化Cとは、人が見やすい形式で記述されたCソースのことです。(IL 言語には使用できません。)
- CC86M:** このコードを選択すると、ISaGRAF コンパイラはプロジェクトで1本のCソースコードを生成します (appli.c と appli.h ファイル)。このファイルをターゲットのライブラリとコンパイル/リンクすると、コンパイルモードの ISaGRAF ターゲットの実行モジ

ユーロルを生成することができます。この機能は、構造化Cソースコード生成機能がサポートされていなかった ISaGRAF V3.23 より前のバージョンとの互換性のために残されています。CC86MはSCCと異なり、べた書きのCソースコードとなります。

注意: IL言語のプログラムに関しては、**SCC**によってCソースが生成されないの、**CC 86M**を使用してください。

☐ **SFCプログラムの処理**

“**組み込みSFCエンジンの使用**”のチェックボックスをONすることにより、ISaGRAF ターゲットにSFCエンジンが組み込まれていることを前提にアプリケーションコードを生成します。**通常は、このチェックボックスをONとしておきます。ターゲット実行時のパフォーマンスが高いためです。**しかし、ISaGRAF のターゲットの実装により、SFCエンジンを内蔵していないものがあります。この場合はこのチェックボックスをOFFすることにより、SFCプログラム記述が全てローレベルの言語に翻訳されます。このようなコードはカスタマイズされた ISaGRAF ターゲットで必要とされる場合があります。

☐ **コード生成の最適化オプション**

ターゲットコードの最適化のために ISaGRAF コードジェネレーションが扱うパラメータは「**コンパイラオプション**」コマンドで選択できます。デフォルトではコンパイル時間の短縮化のため全ての最適化オプションははずされています。

◆ “**2 回最適化を実行**”オプションはコード最適化を2回行ないます。ただし、通常2回目の最適化は1回目ほどの重要な意味を持ちません。

◆ “**定数表現の評価**”オプションは定数表現がコンパイラによって処理されます。例えば2+3は5としてターゲットでは置き換えられます。もし、オプションがセットされていない場合はランタイム中に2+3を計算します。

◆ “**未使用ラベルの圧縮**”オプションはプログラム中の未使用のジャンプラベル、ラベルを無視します。

◆ “**変数コピーの最適化**”オプションは中間結果などをストアするテンポラリ的な変数を最適化します。“**表現の最適化**”オプションと一緒にセットされます。例えば、プログラム中に2回以上使われた Expression の結果などを再利用します。

◆ “**未使用コードの圧縮**”オプションは無意味なコードをなくします。例えば、STプログラムの中で “var := 1; var := X;” があった場合に生成されるコードは “var := X;” のみとなります。

◆ “**算術演算の最適化**”オプションは算術演算を最適化します。例えば、計算式の “A + 0” は “A” に置き換えられます。同様に、“**論理値演算の最適**

化”オプションは論理値演算を最適化します。例えば、“A & A” は “A” に置き換えられます。

- ◆ “Binary Decision Diagram 作成”オプションは論理式 (AND, OR, XOR, NOT 演算子の組み合わせ)を条件付きジャンプで置き換えます。ただし、ジャンプ先でのシーケンスの処理が元の処理よりも短くなる場合にのみ適用されます。

以下のテーブルに最適化オプションのそれぞれのオプションに対するパフォーマンス効果と要求されるコンパイル時間をまとめます。

	速度向上	コンパイル時間
2 回最適化を実行	XXXX	(*)
定数表現の最適化	XXXXXXXX	XXXX
未使用ラベル圧縮	XXXX	XXXXXXXX
変数コピー最適化	XXXX	XXXXXXXX
表現の最適化	XXXX	XXXXXXXX
未使用コード圧縮	XXXX	XXXXXXXX
算術演算最適化	XXXXXXXX	XXXX
論理値操作最適化	XXXXXXXX	XXXX
Binary Decision Diagrams 作成	XXXXXXXXXX	XXXXXXXXXX

(*) コンパイル時間は2倍かかります。

A.13.3 Cソースコード生成

ISaGRAF ワークベンチはアプリケーションコードをCソースコードとして生成することができます。この場合、SFCチャート、データベースを含むプログラムの全てがCソースコードに含まれます。生成されるコードのタイプには、以下の2つがあります。

CC86M(Cソースコード V3.04) 非構造化Cソースコードを生成します。この形式はターゲットが ISaGRAF V3.23 より前のバージョンの場合使用します。

SCC(構造化Cソースコード) 構造化Cソースコードを生成します。ターゲットシステムが V3.23 以降の場合、こちらの形式を選択してください。

※プログラムにIL言語を使用している場合、SCCではCソースコード生成が出来ません。CC86Mでコード生成を行ってください。

CC86Mの場合、以下の2つのファイルが編集集中のプロジェクトのサブディレクトリに作られます。

APPLI.c..... 共通のアプリケーションソースコードファイル

APPLI.h..... 共通のCヘッダーファイル

構造化Cソースの場合、**APPLI.c** と **APPLI.h** ファイルに加え、各プログラムに1つずつ拡張子“.c”のソースファイルと“.h”のヘッダーファイルが作られます。

これらのファイルは ISaGRAF ターゲット用の実行モジュールを生成するために、ISaGRAF ターゲットライブラリと共にコンパイル／リンクされる必要があります。標準の ISaGRAF ターゲットは中間コードをソフトウェア処理(インタプリタ)して実行するため、**インタプリタ型**と呼ばれるのに対して、このCソースコードからコンパイルされる場合は**コンパイル型**と呼ばれます。

注意: アプリケーションのCコンパイル版ではいくつかのデバッグ機能が使えなくなります。例えば、アプリケーションのダウンロード、オンライン修正、ブレークポイントの設定などです。

A.13.4 コード生成情報のモニタ:「編集」メニューコマンド

「アプリケーションコード生成」-「編集」メニューコマンドにより、コード生成中、あるいはベリファイ中に作られたテキストファイルを開くことができます。コード生成ウィンドウはコード生成やベリファイ中のメッセージを表示します。画面表示のメッセージは後でモニタできるようにハードディスクに保管されます。

□ **編集コマンド**

「編集」-「クリア」コマンドにより、コード生成ウィンドウに表示中のテキストをクリアします。各コード生成、ベリファイの前にはウィンドウは自動的にクリアされます。「編集」-「コピー」コマンドにより、表示されているテキストをクリップボードへコピーします。この内容は他のテキストエディタなどで使うことができます。

□ **コンパイラ出力メッセージのモニタ**

「編集」-「メッセージ実行」コマンドにより、最後の「アプリケーションコード生成」、「ベリファイ」コマンドで表示されたエラーメッセージを含む全メッセージを再表示します。

その他の「編集」メニューコマンドによりコード生成やベリファイ中に生成されたファイルをモニタすることができますが、通常の ISaGRAF 使用においては使われません。

A.13.5 リソースファイルの定義

「オプション」-「リソース定義」コマンドにより、リソースを定義することができます。ここで言うリソースとは、ファイルやリストなどの形式のユーザ定義データ(ネットワーク構成、ハードウェア設定など。)のことで、ターゲットコードとマージしてダウンロードされます。このデータは ISaGRAF のカーネルが直接操作することではなく、ターゲット側にインストールされた他のプログラムが参照するためのものです。



リソース定義ファイル

リソースの定義は ISaGRAF のプロジェクトファイルと一緒に“**リソース定義ファイル**”として定義されます。このファイルはASCIIテキストファイルで、ISaGRAF のリソースコンパイラによって処理されます。リソースコンパイラはアプリケーションコード生成時に自動的に起動されます。ファイル内の記述はST言語の文法に従って書かれます。例えばコメントは(* *)で、文字列は ' ' で囲まれます。詳細の文法は該当の項を参照願います。

リソースファイル言語リファレンス

以下に、リソース定義ファイルの中で使われるキーワードとその記述方法を示します

ULONGDATA

意味: 整数値のリスト。ターゲットコードに**符号なし 32 ビット整数**で追加されます。整数はリソース定義ファイルで定義された順番に並びます。整数はコンマによって区切られます。リソースの名称は最大15文字です。

文法:

```

ULONGDATA '<resource_name>'
BEGIN
    ...target_selection...
    ...list of values...
END

```

例:

```

ULongData 'MYDATA'
Begin
    ...
    0, -1, 100_000,          (* decimal *)
    16#A0B1, 2#1011_0101    (* hexadecimal, binary *)
End

```

VARLIST

意味: 変数アドレス(VA)を示すリストです。変数はリソース定義ファイル内の変数名で識別されます。変数アドレスはターゲットコードでは**符号なし 16 ビット整数**で表現されます。アドレスはリソース定義ファイル内で指定された定義順になります。変数はコンマで区切られ、リソース名称は最大15文字です。

文法:

```
VARLIST '<resource_name>'
BEGIN
    ...target_selection...
    ...list of variable names...
END
```

例:

```
VarList 'LIST'
Begin
    ...
    Var100, MyParameter, Command, Alarm
End
```

BINARYFILE

意味: バイナリファイルリソース。MS-DOSファイル形式で保管されます。パス名から成り立っています。End of line キャラクタはリソースコンパイラで変換されません。リソース名称は最大15文字です。

文法:

```
BINARYFILE '<resource_name>'
BEGIN
    ...target_selection...
    FROM '<source_pathname>'
    TO '<destination_pathname>'
END
```

例:

```
BinaryFile 'MYFILE'
Begin
    ...
    From 'c:\user\config.bin'
    To '/dd/user/appl/config.dat'
End
```

TEXTFILE

意味: テキストファイルリソース。ASCII ファイルとして保管されます。ターゲットのリソース定義にはパス名が含まれます。End of line キャラクタはターゲットシステムに合わせたやり方でリソースコンパイラで変換されます。リソース名称は最大15文字です。

文法:

```
TEXTFILE '<resource_name>'
BEGIN
    ...target_selection...
    FROM '<source_pathname>'
    TO '<destination_pathname>'
END
```


例: `TextFile 'MYFILE'`
 `Begin`
 `...`
 `From 'c:\user\config.bin'`
 `To 'dd/user/appl/config.dat'`
 `End`

TARGET

意味: 指定するリソースを含むターゲットコードの名称。実際のターゲット名に関しては、前節（コンパイラオプション）をご覧ください。
"Target"ステートメントは複数のターゲットを選択可能になるべく、同一のリソースブロック内に何回も使われます。**"AnyTarget"**ステートメントが使われているところでは扱えません。

文法: **TARGET** '<target_name>'

例: `BinaryFile 'MYFILE'`
 `Begin`
 `Target 'ISA86M'`
 `Target 'ISA68M'`
 `...`
 `End`

ANYTARGET

意味: 指定リソースがコード生成時に全てのターゲットコードにマージされされることを意味します。「**アプリケーションコード生成**」コマンドで複数のターゲットコードを生成することができますが、これら全てのターゲットコードが対象になります。**"Target"** ステートメントが指定されている場合は扱えません。

文法: **ANYTARGET**

例: `ULongData 'MYDATA'`
 `Begin`
 `AnyTarget`
 `...`
 `End`

FROM

意味: ソースファイル名をパス名付きで指定します。ISaGRAF ワークベンチがインストールされているパソコン内部のバイナリファイル又はテキストファイルのファイル名を指します。パス名記述の際はMS-DOSシステムの指定に従う必要があります。

文法: **FROM** '<target pathname>'

例: `BinaryFile 'MYFILE'`
 `Begin`
 `...`
 `From 'c:\user\config.dat'`

```
To '/dd/user/appl/config.dat'  
End
```

TO

意味: デスティネーション(ターゲットシステム)における**バイナリファイル**または**テキストファイル**名を指します。ファイル名の記述はターゲットシステムの指定に従う必要があります。

文法: TO '<target pathname>'

例:

```
TextFile 'MYFILE'  
Begin  
...  
From 'c:\user\config.dat'  
To '/dd/user/appl/config.dat'  
End
```

リソース定義ファイルの例

以下に、一つのリソース定義ファイル例を示します。

(* リソース定義ファイル *)

```
ULongData 'DATA1'          (* データリスト *)  
Begin  
    Target 'ISA86M'         (* 指定されたターゲットコードに限定 *)  
    1, 0, 16#1A2B3C4D, +1, -1 (* 数値 *)  
End
```

```
VarList 'VLIST1'           (* 変数リスト *)  
Begin  
    Target 'ISA86M'         (* このターゲットコードに限定 *)  
    Valve1, StateX, Command, Alrm1 (* 変数名 *)  
End
```

```
BinaryFile 'FILE1'         (* バイナリファイルリソース *)  
Begin  
    AnyTarget              (* 全ターゲットコードを対象 *)  
    From 'c:\user\updatef.bin' (* ソースファイル名 *)  
    To 'updatef.cfg'        (* ターゲットでのファイル名 *)  
End
```

```
TextFile 'FILE2'           (* テキストファイルリソース *)  
Begin  
    Target 'ISA68M'  
    From 'c:\nw\nwbd.txt'   (* ソースファイル名 *)  
    To '/nw/dat/nwbd'      (* ターゲットでのファイル名 *)  
End
```

リソースコンパイル

リソースがリソース定義ファイルで定義されている場合は、コード生成が終了後に以下のダイアログボックスが開きます。



"Start compile" ボタンを選択することにより、リソースコンパイルが開始されます。出力メッセージやエラーメッセージはこのウィンドウに表示されます。"Exit" ボタンを選択することにより、リソースコンパイラを抜けます。この場合はリソースはターゲットコードに付加されません。



リソースのターゲットコードへのマージ

リソースの数、行数、ファイル数は ISaGRAF には制限はありません。リソースはターゲットコードの最後にリソースディレクトリ付きで付加されます。以下にリソースディレクトリフォーマットをC言語の文法で示します。

```
RESOURCE:
{
    long nbres;                /* 定義されているリソースの数 */
    {
        char name[16];        /* リソース名 */
        long type;            /* リソースデータタイプ */
        long size;            /* データブロック長 */
        void *data;
        char *path_offset;    /* 文字列へのポインタ */
    } /*nb of records */
}
```

上記の "type" フィールドには以下のいずれかが指定されます。

- 1 = バイナリファイル
- 2 = テキストファイル
- 3 = データリスト:ulong data (この場合 path_offset フィールドは使われません)
- 4 = 変数リスト:varlist (この場合 path_offset フィールドは使われません)

テキストファイルに関して、end of line キャラクタはターゲットシステム合わせてリソースコンパイラが翻訳します。全てのポインタは32ビットオフセットとなります。全てのリソース名やパス名は NULL で終了します。パス名やデータが上記のリソースディレクトリの後に続きます。

A.14 クロスリファレンスの使い方



ISaGRAF ワークベンチには、プロジェクトのプログラムに宣言されている変数をまとめて表示し、プログラム中のどこでつかわれているかという情報を提供する**クロスリファレンスエディタ**があります。クロスリファレンスにより、プロジェクトで宣言されるすべての変数をリストアウトし、プログラムのソースコードのどの部分に変数が使用されるかの場所を特定することができます。クロスリファレンスは、1つの変数のグローバルな見方ができるため、プロジェクトのメンテナンスの効率化に非常に有用です。クロスリファレンスはI/O変数とI/Oボードの割り付けや、プロジェクトの保守、ドキュメント作成にも役立ちます。また、クロスリファレンスはプロジェクトの辞書としても使用されます。未使用の変数も容易に見つけることができます。



リストの左には、プロジェクトで宣言されてオブジェクト(プログラム、変数及び定義された単語)、及びプロジェクトで参照されるライブラリエレメント(ファンクションとファンクションブロック)を示します。右のリストは左のリストで選択されたオブジェクトがプログラムで使われている場所を示します。

使用場所の記述には、プログラム名とフローチャートや SFC のステップ番号、トランジションあるいはテスト番号、テキスト言語に対しては行番号、LD、FBD 言語に対しては座標などの情報が含まれています。QuickLD の場合は、ラングの番号が表示されます。もし、変数が出力(コイル)として使用されているならば、ラングの番号の前に"*"の文字が付加されます。

クロスリファレンスを起動すると、「**オブジェクト検索**」ダイアログが表示されます。ここで、検索したいオブジェクト名だけを指定して表示することができます。これでクロスリファレンスの計算時間を短縮することができます。「全部」を選択すると、全オブジェクトが表示されます。

「**オプション**」メニューの「**未使用変数表示**」オプションを設定すると、アプリケーション中で使用されていない変数も表示することが可能です。

変数の検索をクロスリファレンスを使用して行う場合、変数名称を入力し **OK** をクリックすることによるある特定の変数の検索、もしくは **All** をクリックすることにより全ての変数の検索を実行できます。

オブジェクトタイプの選択

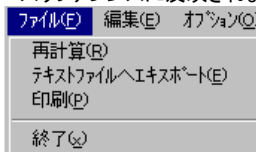
ISaGRAF 内では、非常に多数のオブジェクトを扱えるため、ツールバーのコンボボックスからウィンドウにリストされているオブジェクトのタイプを選択することができます。このオブジェクトタイプ選択により指定されたオブジェクトのみを参照することができます。

範囲を選択するたびに、クロスリファレンスは再計算されます。「**全部**」を選択すると、これを解除し、全てのオブジェクトを表示します。



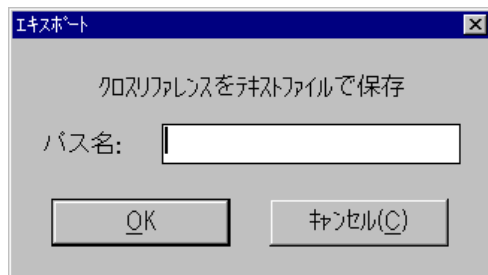
クロスリファレンスの再計算

「ファイル」-「再計算」コマンドにより、他の ISaGRAF の編集ウィンドウで修正された内容の最新情報がクロスリファレンスに反映されます。



クロスリファレンスのエクスポート

「ツール」-「テキストファイルへエクスポート」コマンドにより、クロスリファレンスの完全なテキストリストが ASCII テキストファイルに書き出されます。このファイルはWindowsメモ帳やワードプロセッサなどの他のアプリケーションでこのファイルを開くことができます。



辞書エラー

「編集」-「辞書エラー」コマンドにより、ダイアログボックスの中にプロジェクト辞書がロードされたとき、検出されるエラーのリストを表示します。



変数の統計

「ツール」-「変数統計」コマンドにより、プロジェクトで宣言されている変数に関して、変数タイプと属性に従って変数の数の統計を表示します。このコマンドにより、プロジェクトで宣言されるI/O変数、内部変数の総数を把握することができます。この機能を使うことで、I/O点数が限定された ISaGRAF ワークベンチが使用される時、コンパイルできるかどうかの判断に使うことができます。

変数統計						
	ブール型	整数型	実数型	タイ型	文字列	合計
内部	0	3	0	1	0	4
入力	4	2	0	---	0	6
出力	8	1	0	---	2	11
合計	12	6	0	1	2	21

I/Os
17



オブジェクトリストからの検索

「編集」-「検索」コマンドにより、リストからオブジェクトを検索できます。リストに現在表示されていないものに関しては、オブジェクトを見つけることはできないのでオブジェクトを検索する前にツールバーで「全部」を選択しておくことが望ましいです。

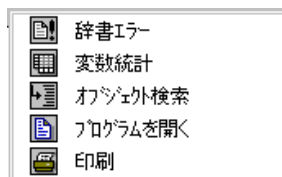


オブジェクトが使われているプログラムを開く

右のリストには、現在オープンされているプロジェクトの、選択されたオブジェクトを含むソースファイルや I/O 接続が表示されます。「編集」-「プログラムを開く」コマンドにより、選択されているオブジェクトが含まれているプログラムを直接開くことができます。また、右のリストのオブジェクトをダブルクリックすることによりプログラムを開くこともできます。

A.14.1 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.15 グラフィックデバッグの使い方



ISaGRAF にはグラフィック又はシンボリックデバッガーの機能があります。プログラム管理の「**デバッグ**」コマンドを選択することでターゲット側にダウンロードされたアプリケーションの制御が行なえます。この状態ではデバッガーはハードウェアリンク(ネットワーク)を通してターゲットと通信を行なっています。「**シミュレーション**」コマンドはデバッグ画面表示と同時にターゲットシステムをシミュレーションします。このシミュレーションによりユーザはターゲットI/Oがまだ完成していなくてもアプリケーションをテストすることができます。デバッガーウィンドウからアプリケーションを操作することができます。

デバッグが起動され、ターゲット側のアプリケーションがワークベンチ側のアプリケーションと同一の場合は、自動的に**プログラム管理ウィンドウがデバッグモード**で開かれます。(ターゲットとワークベンチ側でアプリケーションが一致していない場合は、プログラム管理ウィンドウは開きません。) また、プログラム管理ウィンドウのメニューから開かれる他のウィンドウ(グラフィックエディタ、テキストエディタ、辞書エディタ、変数リスト、I/O接続など)は全てデバッグモードで開かれます。即ち、ここではプログラムの変更、編集コマンドは無効になります。表示されたプログラムのコンポーネント(ステップ、トランジション、変数...)は現在の状態を表示します。コンポーネントをダブルクリックすることでターゲットの状態や変数の値のみを一時的に変更することができます。

デバッグが**シミュレーションモード**で実行されているとき、ワークベンチとターゲット間の通信は行なわれていません。デバッグはI/Oシミュレータウィンドウと通信を行っています。ターゲット自体が存在しませんのでデバッグメニューの「**ダウンロード**」、「**アプリケーションのスタート**」、「**アプリケーションのストップ**」コマンドなどは使えません。

A.15.1 デバッグウィンドウ

デバッグウィンドウにはアプリケーションの実行状態だけが表示され、他のISaGRAF のウィンドウを組み合わせる使用することによって、インタラクティブなデバッグが可能となります。ランタイムエラー検出時には、デバッグウィンドウの下部に表示されます。



ランタイムエラーの表示は、「**オプション**」-「**エラー表示**」により、エラーメッセージ表示の表示・非表示の切替えが可能です

デバッガーウィンドウメニュー（ツールバー）の下の状態表示欄にはターゲットのアプリケーションの状態とサイクルタイム情報が表示されます。以下にターゲットの状態例を示します。

- ロギング中:**..... デバッガーがターゲットと通信をしようとしている状態
- 遮断状態:**..... デバッガーがターゲットと通信できません。ケーブル接続又は通信パラメータの設定が正しいかを確認して下さい。
- アプリケーションがありません:**..... 通信は正常ですが、ターゲットには実行中の ISaGRAF アプリケーションが存在していません。アプリケーションを**ダウンロード**、あるいは、**更新**して下さい。
- アプリケーション活性:**..... 通信は正常で、ターゲットにアプリケーションがあり、デバッガとターゲットアプリケーションが通信しようとしています。ワークベンチとターゲット側のアプリケーションが同じであればプログラム管理ウィンドウが開きます。
- RUN:** ターゲットが**リアルタイムモード**で実行中です。
- STOP:**..... ターゲットが**サイクルモード**で実行中です。
- ブレイクポイント:**..... アプリケーションがブレイクポイントにさしかかったのでターゲットが**サイクルモード**で実行中です。
- 致命的なエラー:**..... ターゲットアプリケーションが致命的または意味不明のエラーにより実行できません。

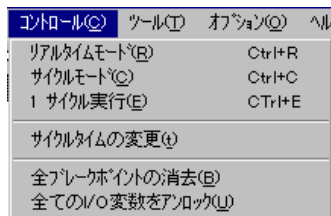
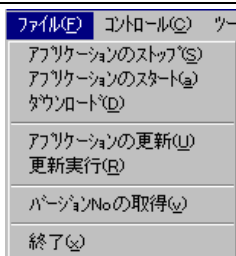
サイクルタイムに関する情報を以下に示します。

- 許容値:**..... プログラムされたサイクルタイム
- 現在値:**..... 最終のサイクルタイム測定値
- 最大値:**..... アプリケーションがRUNしてからのサイクルタイムの最大値
- オーバーフロー:**..... サイクルタイムの現在値が許容値を越えたサイクル数

全ての値はms表示です。サイクルタイムの表示はターゲットシステムにより表示の精度が変わります。詳しくはターゲットユーザガイドを参照願います。デバッガがシミュレーションモードで使われているときにこれらの情報は表示されません。

A.15.2 アプリケーションの制御

「ファイル」、「コントロール」メニューコマンドにより、ターゲット上での ISaGRAF アプリケーションを制御できます。



注意: 以下のいくつかのコマンドはシミュレーションモードでは使えません。なぜなら、シミュレーションモードでのアプリケーションはワークベンチ内部に作られるからです。



ターゲットアプリケーションのストップ

「ファイル」-「アプリケーションのストップ」コマンドにより、現在実行中のターゲット上のアプリケーションをストップさせることができます。



ターゲットアプリケーションの実行

「ファイル」-「アプリケーションのスタート」コマンドにより、ターゲット上のアプリケーションをスタートさせることができます。アプリケーションがダウンロードされたときは自動的にアプリケーションはスタートします。よって、「**アプリケーションのスタート**」コマンドは、通常「**アプリケーションのストップ**」コマンドの後に実行されることになります。

注意: 新しいアプリケーションをターゲットへダウンロードされる前に、ターゲット上のアプリケーションはストップさせておく必要があります。



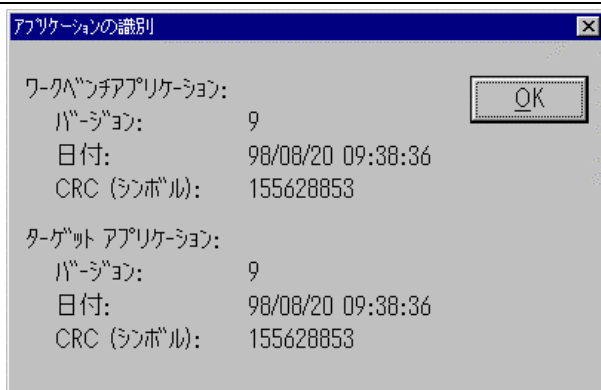
アプリケーションのダウンロード

「ファイル」-「ダウンロード」コマンドにより、ワークベンチ側で生成されたアプリケーションコードをターゲット側へダウンロードすることができます。ターゲットタイプに応じて、ダウンロードすべきコードのタイプ(実行TICコード、アプリケーションシンボルテーブル)を選択して下さい。



バージョンNo. の表示

「ファイル」-「バージョン No.の取得」コマンドにより、ワークベンチ上に開かれているアプリケーションとターゲット上に存在しているアプリケーションのバージョンNo.を確認することが出来ます。以下の項目が表示されます。



バージョン:.....この番号はコードジェネレータがカウントしている過去からのコード生成の回数です。

日付:.....アプリケーションコードが生成された日付と時刻を表示します。

CRC:.....シンボルテーブルから計算されたチェックサムを表示します。この数字はコードジェネレータで計算された値で、変数辞書の内容で決まります。(辞書テーブルを変更するとCRC値は変わってしまいます。)

注意: 「バージョン No.の取得」コマンドはシミュレーションモードでも使用できます。ただ、実際のデバッグモードではターゲットとの接続がなされていない場合は使用できません。



アプリケーションのオンライン修正

「ファイル」-「アプリケーションの更新」コマンドにより、実行中のターゲットアプリケーションに対してオンライン編集が可能になります。後の節で詳しく説明を行います。シミュレーションモードでは使えません。



リアルタイムモード

「コントロール」-「リアルタイムモード」コマンドにより、通常モードで、サイクル実行は指定されたサイクルタイム毎にトリガーがかかり起動されます。リアルタイムモードは、アプリケーションが実行可能状態になっていないと、設定できません。



サイクルモード

「コントロール」-「サイクルモード」コマンドにより、ターゲットアプリケーションをサイクル実行モードにします。このモードでは「コントロール」-「1サイクル実行」コマンドにより、1サイクルを実行して停止します。このコマンドもアプリケーションが実行可能状態になっていないと、設定できません。



1サイクル実行

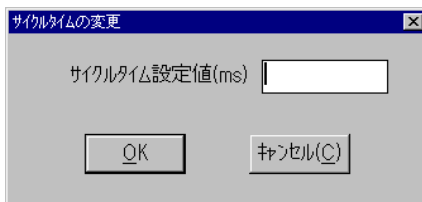
ターゲットアプリケーションがサイクルモードの時、「コントロール」-「1サイクル実行」コマンドにより、1サイクルを実行して停止します。

尚、アプリケーションスタート時の実行モードは「**アプリケーションのランタイムオプション**」内のパラメータとしてセットされています。



サイクルタイムの変更

「コントロール」-「**サイクルタイムの変更**」コマンドにより、プログラムされたサイクルタイムをターゲットアプリケーションが実行中に変更することができます。この変更値は「**許容値**」としてデバッガーのコントロールバーに表示されます。サイクルモードへの切り替えが必要な場合は、サイクルタイムを変更する前にセットしておく必要があります。サイクルタイムは整数で入力しますが単位はmsです。



全ブレークポイントの消去

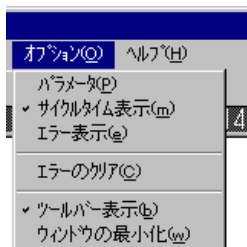
「コントロール」-「**全ブレークポイントの消去**」コマンドにより、アプリケーション全体にわたって設定されているブレークポイントを「全て」クリアします。ブレークポイントは、このコマンドを使わないとデバッガーウィンドウを閉じても自動的にクリアされることはありません。

全I/O変数のアンロック

「コントロール」-「**全てのI/O変数をアンロック**」コマンドにより、デバッグ中にアプリケーション内でロックされているI/O変数の全てをアンロックします。I/Oがロックされている時は、デバッガコマンドやアプリケーションによってI/O変数の状態は変更が可能ですが、実I/O状態は変化しなくなります。ロックされたI/O変数は、このコマンドを使わないと、デバッガーを閉じても自動的にアンロックされません。

A.15.3 オプション

「**オプション**」メニューによりデバッガーウィンドウに表示される情報をコントロールすることが出来ます。



通信パラメータ

「オプション」—「パラメータ」コマンドにより、通信パラメータを設定できます。ただし、設定できるのは**通信タイムアウト**パラメータだけです。その他の通信パラメータ（ポーレート、パリティ...）はプログラム管理ウィンドウの「デバッグ」—「通信リンク設定」コマンドで設定します。

通信タイムアウトとはワークベンチからの要求に対してターゲットが通信を返し始めるまでにターゲットシステムに与えられた時間のことを指します。**サイクルリフレッシュ間隔**とは、デバッガーでオープンされているウィンドウ表示のリフレッシュのために、デバッガーから送られる読み込み要求の時間間隔のことを指します。表示されている数値は全てミリ秒です。デバッガーがシミュレーションモードの時は通信パラメータは設定できません。



表示オプション

「オプション」—「サイクルタイム表示」コマンドにより、デバッグウィンドウへのサイクルタイム表示の有効・無効を切り替えます。「サイクルタイム表示」がセットされているときはサイクルタイム（許容値、現在値、最大値、オーバーフロー）が表示されます。セットされていないときは表示されず、ターゲットとの通信負荷が軽減されます。

「オプション」—「エラー表示」オプションがセットされているときはランタイムエラーがデバッグウィンドウに表示されます。セットされていないときはエラーリストは閉じられていて、ターゲットとの通信負荷が軽減されます。「オプション」—「エラーのクリア」コマンドにより、現在デバッグウィンドウに表示されているエラー表示がクリアされます。

「オプション」—「ウィンドウの最小化」コマンドにより、デバッグウィンドウのサイズを最小にして、更に常にトップに表示し、最小限必要な操作のみできるパネル表示に切り替えられます。



A.15.4 書き込みコマンド

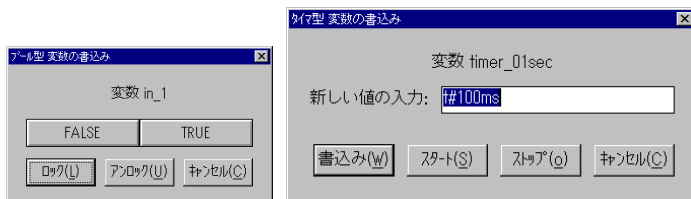
ISaGRAF シンボリックデバッガーによりアプリケーション内の変数の**値や状態**を変更することが出来ます。デバッガーウィンドウが開いている状態でプログラム編集ウィンドウなどでシンボル名称やエレメント部分を**ダブルクリック**することで変更が可能状態になります。

変数の操作

以下のウィンドウで変数をダブルクリックすると、変数の値を変更することができます。

- 辞書エディタ
- 変数リスト
- LD あるいは FBD プログラム
- I/O 接続エディタ

以下のコマンドがデバッグダイアログボックスで使われます。



- 新しい値の“書き込み”
- 変数の“ロック”(I/O変数に限ります。)
- 変数の“アンロック”(I/O変数に限ります。)
- タイマー変数のスタート あるいは、ストップ (自動リフレッシュモードをセット)

ブール型値で FALSE、TRUE 値を表すのにシンボリック変数が使えます。シンボリック変数とは、変数辞書で特定のブール型値変数に定義された文字列です。アナログ変数に対して“書き込み”を行う場合は変数辞書の定義に合わせて整数あるいは実数型で入力する必要があります。文字列型変数に対して“書き込み”を行う場合は変数に与えられた最大文字数を越えることは出来ません。

SFCオブジェクトの操作

デバッグ時に**SFCプログラム**を操作するには、プログラム管理ウィンドウの「**ファイル**」メニューのコマンドを使います。コマンドの実行の前に、対象となるプログラムを選択しておきます。以下に使えるコマンドを示します。



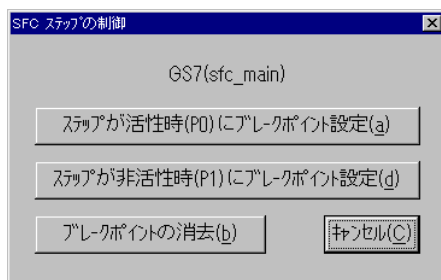
スタート: 選択されたSFCプログラムのイニシャルステップにトークンをセットして、選択されたプログラムを実行モードにします。

停止 (Kill) 存在している全トークンを取り除くことで選択されたプログラムを停止させます。

- 凍結(Freeze)** 存在している全トークンを選択されたプログラムから取り除くがそれらの場所を記憶させておきます。再起動コマンドと対になって操作されます。
- 再起動(Restart)** 凍結されているプログラムに対して一度取り除かれたトークンを元の場所に戻します。凍結コマンドと対になって操作されます。

チャイルドプログラムにおいては、これらのコマンドはそれぞれ"**GSTART**", "**GKILL**", "**GFREEZE**" さらに"**GRST**" ステートメントに対応します。

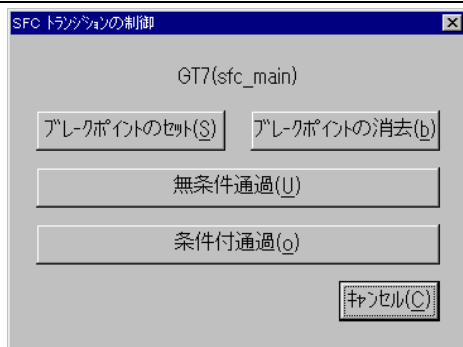
SFCチャート中の**SFCステップ**からも制御の状態をコントロールすることが出来ます。これを**ブレイクポイント**といいます。ステップを選択し、右クリックのショートカットメニューまたはエディタの「**編集**」-「**ブレイクポイント**」コマンドにより、以下のダイアログボックスに現れます。



- ステップが**活性(状態)時**にブレイクポイントの設定
- ステップが**非活性状態時**にブレイクポイントの設定
- ステップに設定されているブレイクポイントを**消去**

注意: 活性状態時ブレイクポイントと非活性時ブレイクポイントは同一ステップには設定できません。

SFCTランジョンからもブレイクポイントを設定して、制御の状態をコントロールすることが出来ます。ステップの場合と同様に、右クリックのショートカットメニューまたはエディタの「**編集**」-「**ブレイクポイント**」コマンドにより、以下のダイアログボックスが現れます。



- トランジション通過時にブレイクポイントの設定
- 設定されたブレイクポイントの消去
- 手動でのトランジション通過(トークンの移動又は追加)

無条件通過: トランジションの次のステップにトークンを移動(追加)します。このとき前のステップにあるトークンを取り除きません。

条件付き通過: トランジションの次のステップにトークンを移動します。このとき前のステップにあるトークンを自動的に取り除きます。

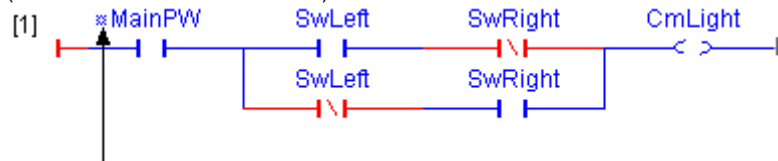
A.15.5 ロック状態とデバイス値の表示

ブール型や数値型 IO がロックされている時、ワークベンチはロック状態や現在のデバイス値を読み込みます。現在のデバイス値は強制出力値も含んでいます。IO のロック状態は以下のエディタ上で表示されます。

- FBD エディタ
- LD エディタ (Quick LD)
- 辞書
- 変数リスト (スパイリスト)

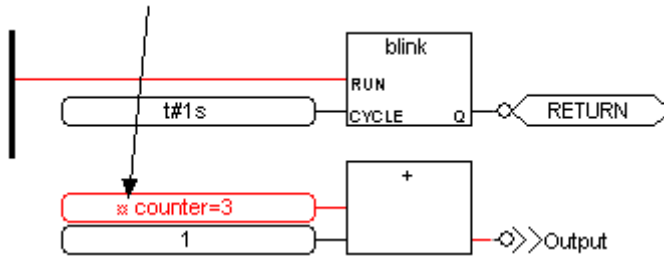
FBD and LD エディタでは、IO がロックされている場合その名称には"■" マークが表示されます。以下の LD ダイアグラムの例ではロックシンボルでマークされた MainPW I/O が示されています。

(*左右両スイッチによるライト制御 *)



MainPW はロックされています。

以下の FBD の例ではロックシンボルでマークされた counter が示されています。
counter はロックされています。



辞書や、変数リストではロック状態やアプリケーション値に加えて実際のデバイス値も表示します。この情報は以下の式を使用して表示されます。

AppValue (≠ Device value).

辞書での以下の例では、アプリケーション値=3、強制値=127のロックされた変数 counter を表示しています。

ISaGRAF - BLINKALL:[untitled] - List of variables		
Name	Value	Comment
FBDO	FALSE	first FBD variable
counter (ProgFBD)	3 (*127)	
FBDO3	TRUE	
<end of list>		

counter はロックされ強制値は 127

A.15.6 オンライン修正の詳細

ISaGRAF ではアプリケーションプログラムを停止することなく、オンラインでプログラムを修正する機能があります。これを“オンライン修正”と呼びます。この機能は**細心の注意を払って**行う必要があります。なぜなら、オンライン修正機能においては、ISaGRAF はユーザの変更による全ての問題点を検出することが出来ないかもしれないからです。オンライン修正を行うときは、デバッグモードで開かれているプログラム管理ウィンドウを一旦閉じてから改めて開き直して必要なプログラムを修正し、再度「**アプリケーションコード生成**」コマンドでコードを生成します。さらに、生成されたコードは「**アプリケーションの更新**」コマンドでダウンロードを行い、「**更新実行**」で、アプリケーションの実行を実際に切り替えます。ただし、変数辞書が変更された場合はCRCコードが一致しませんのでオンライン

修正は**無効**となります。この場合は一旦アプリケーションを停止してから新アプリケーションをダウンロード、スタートする必要があります。

□ **コードシーケンス**

ISaGRAFではデバッグの機能により、I/O接続や変数へいろいろな方法でアクセスすることができますが、オンライン修正における適用範囲は、**コードシーケンスの変更**に限られます。このコードシーケンスとは“サイクルの最初:Begin”又は“サイクルの最後:End”セクションに書かれているST、IL、LD、FBDプログラムの全リストと、SFC・FCのレベル2で記述されているプログラミングのことを指します。オンライン修正とは、ターゲットアプリケーションの実行を停止させることなく複数のコードシーケンスを変更することを意味しますが、**SFCにおいてはトークンのコントロールが大変クリティカルであるため、SFCの構造の変更、即ちステップやトランジションの追加、削除は行うことが出来ません。**

□ **内部変数の変更**

変数値に関してはアプリケーションに大変重要な位置づけでもありますがデバッガーからの変数の値の変更はいつでも行えます。しかし、**オンライン修正においては変数を新たに追加したり、削除したりすることは出来ません。**ただし、使わない変数(内部あるいはI/O)でも宣言をしておき(辞書にあらかじめ変数を定義しておき)はじめのプログラムでは使わなくても後で使用するということであればオンライン修正で行うことが出来ます。

- 辞書登録された変数

これらは、オンライン修正では追加、削除、名前の変更などはできません。オンライン修正の為に、今現在は使用しないかもしれませんが、いくつかの予備の変数をあらかじめ登録しておくことが考えられます。この様な予備の変数は、予期せぬアプリケーションの変更に際しても、アプリケーション CRC チェックサムが変わらないので、オンライン修正が可能になります。

- ファンクションブロックのインスタンス

C 言語や ICE 言語で記述されたファンクションブロックのインスタンスも、ISaGRAF のターゲットデータベースに登録されています。従って、ファンクションブロックのインスタンスが追加されたり削除されたりするとオンライン修正はできません。ですから、ブロックを挿入すると自動的にファンクションブロックのインスタンスが自動的に生成される QuickLD や FBD エディタでのプログラミングよりも、ファンクションブロックのインスタンスを明示的に辞書に登録して、ST 言語でプログラムを記述するほうがよいでしょう。ライブラリに登録されているファンクションブロック自体の編集も、オンライン修正ができなくなる原因になります。

- SFC のステップ

SFC のステップに関しても、そのダイナミック属性(活性・非活性状態)の情報がターゲット側に登録されています。よって、SFC のステップの追加や削除のような編集は、アプリケーションデータベースの変更につながり、オンライン修正を不可能にします。

- コードジェネレータが生成する内部変数

ISaGRAF のコードジェネレータは、複雑なプログラムをコード化する為に、隠れたテンポラリ変数を生成します。時には、プログラムの表現方法の変更が、このテンポラリ変数の変化につながる場合があり、その結果、オンライン修正が不可能になります。この様な状況を防ぐ為に、下記に示すような設定を **ISA.INI** ファイルに行うことで、強制的に、各プログラム毎に最低限のテンポラリ変数を割り振ることができます。

```
[DEBUG]
MNTVboo=8 ; for booleans
MNTVana=4 ; for integers and reals
MNTVtmr=4 ; for timers
MNTVmsg=2 ; for messages
```

ISA.INI ファイルにこの設定がされたときには、コードジェネレータは、アプリケーションのコード生成の結果、指定以上のテンポラリ変数が必要であった場合には、ワーニングメッセージを出力します。

□ 入力、出力変数の変更

ISaGRAF でのI/Oシステムは、ハードウェア仕様を考慮に入れてI/O開発キットによって変更することが出来ます。**オンライン修正では、I/Oボードの記述を変更したり、I/Oボードチャネルの割付を変更(追加、削除)したりすることは出来ません。** ボードパラメータの変更や、I/O チャネルのロックは、通常のデバッグ機能や、"OPERATE"ファンクションにより実行できます。

□ オンライン修正の操作手順

オンライン修正の手順を以下に説明します。

- ターゲットでのアプリケーションが実行中に一度デバッグウィンドウを閉じます。
- プログラムエディタでアプリケーションのソースコードを変更します。
- 「アプリケーションコード生成」を行います。
- アプリケーションコードを「**アプリケーションの更新**」コマンドでダウンロードします。(「ダウンロード」コマンドは使いません)
- 「**更新実行**」コマンドにより古いアプリケーションから新アプリケーションへの更新をサイクルタイムの間に行います。

以上の手続きにより、ターゲット内部では、常に各サイクルにおいて完全な信頼性あるアプリケーションが実行されていることとなります。オンライン修正の頻度に関しての制限はありません。

オンライン修正は結局「**アプリケーションのストップ**」、「**ダウンロード**」、「**アプリケーションのスタート**」をマニュアルで行うことと本質的には変わりありませんが、違いは**変数状態が失われることなくまた、アプリケーションの切り替え時間が大変短い(通常1～2サイクル)**ことにあります。

アプリケーションが切り替わる(更新される)間は**全変数(内部、I/O)の状態は同じ値を保持**していて、SFCにおいては**いかなるアクションも行われずSFCTークンは動きません。**

オンライン修正に必要なメモリ要求

オンライン修正実現の要件として、ターゲット内部に修正されたアプリケーションがダウンロードされるのに十分なメモリが空いている必要があります。アプリケーションが更新時には、旧、新2つのバージョンのコードが同時にメモリ上に存在することになります。

オンライン修正における制限

前にも述べましたが、オンライン修正ではコードシーケンスの変更が許されていますが、変数の定義、関数パラメータの変更、I/O接続変更などは行うことが出来ません。

修正版のアプリケーションがダウンロードされる時に ISaGRAF では危険な変更を未然にチェックする目的からエラー検出を行っています。エラーが検出された場合は、メッセージを表示してアプリケーションの更新は中止されます。チェックの内容にはシンボルテーブルのチェックサムチェック(CRCチェック)があります。ここでは、変数の定義、プログラム名、SFCのエレメント名の修正が無かったかどうかチェックしています。

SFCにおいて更新時にステップが活性状態の時、ノンストアード(N)アクションに関しては失われます。また、新しいステップのバルス(ステップの立ち上がり)アクションは実行されません。新しいステップのP1(ステップの立ち下がり)アクションは実行されます。また、更新時に有効だったトランジションは、条件式が新しいアプリケーションコードで再度チェックされます。

新しくダウンロードされたアプリケーションは**PLC内部でのバックアップ(ハードディスクへの保管)はなされません**。通常のダウンロードコマンドによってダウンロードされたアプリケーションのみハードディスクにバックアップされます。

注意: アプリケーションのオンライン修正を行った後に、「アプリケーションのストップ」コマンドを実行あるいはターゲット側を終了した場合には、次回アプリケーションがスタートされたときは、オンライン修正以前の旧アプリケーションがスタートします。これは、アプリケーションのオンライン修正の場合、アプリケーションコードはハードディスクに保存されないでメモリ上に展開されているためです。従って、次回からも修正されたアプリケーションで実行したい場合は、オンライン修正後に「ダウンロード」を必ず1回実行することが必要となります。(詳細は、ターゲットマニュアルを参照願います。)

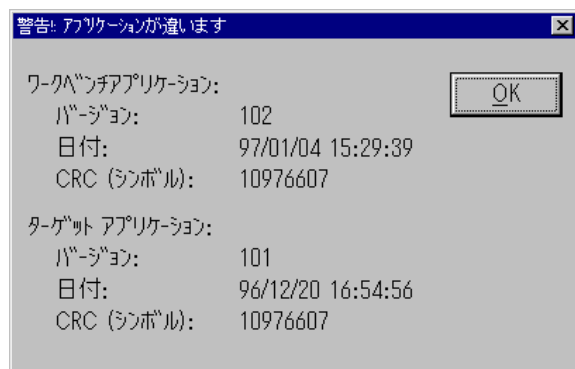


オンライン修正操作の詳細

アプリケーションの更新時の操作を一般的な注意事項を含めて以下に解説します。

- アプリケーションの修正を行う場合は対象のプロジェクトを前もって別名でバックアップしておくことが重要です。修正はそのコピーで行うのが賢明です。
- プログラムの修正時にはエディタウィンドウで「**修正履歴の更新**」をセットしておく方が好ましいです。後でのプログラムの保身に役立ちます。
- コードシーケンスに変更をした場合は必ず新たに「アプリケーションコード生成」コマンドでコードを再生成する必要があります。

- 「デバッグ」コマンドにより、まずターゲット側(旧アプリケーションが実行中)とワークベンチ間を接続しておきます。このとき、ワークベンチ側とターゲット側のアプリケーションコードが異なるために、メッセージボックスが開くのでより安全です。



- デバッグ「ファイル」-「アプリケーションの更新」コマンドを実行します。
- 新しい名前のプロジェクトからデバッガーを起動すると、シンボル情報が異なるため、ターゲットデータベースにアクセスすることができません。また、「アプリケーションの更新」も実行することができません。
- 修正されたアプリケーションを更新用にダウンロードします。この時、ダウンロードダイアログボックスでは「**後で更新**」を選択します。(この選択によりデータのダウンロードスピードが少し遅くなります。)



- ダウンロードが完了したら、「ファイル」-「更新実行」コマンドによりアプリケーションを更新します。更新には1~2サイクル要します。
- うまくアプリケーションが更新された場合は新しいアプリケーションプログラムがデバッグモードで再表示されます。もし、アプリケーションの更新が成功しなかった場合は古いアプリケーションが表示されたままになっています。

A.15.7 DDE通信

ISaGRAF ワークベンチデバッガーはDDE(Dynamic Data Exchange)サーバ機能を含んでいます。DDEの中で**Advise**と**Poke**ループをサポートしています。これにより、ISaGRAF 以外のDDEをサポートしているアプリケーションにデータを引き渡すことが出来ます。それ以外の Execute 等の DDE サービスは使用できません。

Adviseループでは ISaGRAF の変数に変化が生じたときのみクライアントタスクにデータが渡されます。Adviseループで変数をモニタリング中の場合は、その変数に対して**Request**ループも扱えます。全タイプの変数をモニタすることが出来ます。DDEでのパラメータを以下に示します。

アプリケーション名 "ISaGRAF"
トピック名 ISaGRAF のプロジェクト名
アイテム名 変数名

もし、変数がプログラムのローカル変数である場合は以下のように括弧で括って、プログラム名を明記する必要があります。

変数名(プログラム名)

ISaGRAF デバッガの DDE サーバは、現在実行中のアプリケーションのみ有効です。DDE Adviseループにより最大256変数の同時モニタが行なえます。DDEはデバッグモード、シミュレーションモードいずれの場合からでも使用することが出来ます。リフレッシュ時間間隔はワークベンチ側ターゲット側との通信の時間となります。

A.15.8 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.16 変数のスパイ



デバッグウィンドウの「ツール」―「スパイリスト」コマンドにより、非連続的な変数をモニタすることができます。

また、モニタされている変数の値を強制的に変更することも可能です。変数リストの定義では**最大32個の変数**を登録できます。異なるタイプの変数を組み合わせることも自由です。また、グローバル、ローカル変数を組み合わせることも可能です。登録されたリストの定義はハードディスクに保管することができます。登録されたリストは編集集中のプロジェクト専用で他のプロジェクトからは扱うことができません。

変数リストはアプリケーションの部分的な機能のテストに有効です。アプリケーションのソースコードとは別に特定部分のプロセスの変化を見ることができます。

スパイリストはデバッグモードのST、IL言語のエディタからも使用できます。変数のグループ分けと、プログラム実行の制御及びモニタリングもできます。



変数リストの保存

「ファイル」―「開く」、「上書き保存」、「名前を付けて保存」コマンドにより、スパイリストをハードディスクから開いたり、ハードディスクへ保存することができます。1プロジェクト中のスパイリストの数には制限はありません。ファイル名(スパイリスト名)の付け方には以下のルールがあります。

- 変数リスト名の最大長は半角8文字
- 最初の文字は英文字(a-z)
- 以降の文字は英文字、数字、_ のいずれか
- 大文字、小文字の区別なし

1ウィンドウで同じに表示できるリストは1つだけですが、スパイリストエディタは同じに複数オープンすることができます。



変数のリストへの挿入

「編集」―「挿入」コマンドにより、リストに別の変数を追加します。変数名はプロジェクトの辞書エディタから選択できます。新規の変数は選択中の変数の前に挿入されます。最大32個の変数が登録できます。同一変数は2度定義できません。一度に32個以上の変数リストを表示する場合は、2つのリストウィンドウを開くことになります。



選択された変数の変更

「編集」―「修正」コマンドにより、リスト中の選択された変数名を変更することができます。また、「編集」―「切り取り」コマンドにより、選択中の変数を削除できます。



ダンプ表示

「ズーム」ボタン、あるいは「オプション」-「ダンプリスト」コマンドで変数表示をリスト形式とダンプ表示形式に切り替えることが可能です。

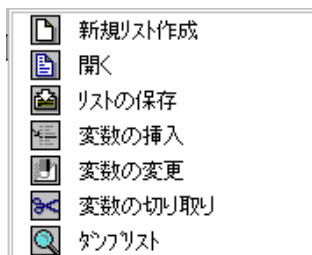
ダンプ表示モード時には、1つの変数の値のみ表示されます。ウィンドウの上部にその値を数値あるいは文字列で表示し、その下には、バイナリ形式でダンプしたデータが表示されます。この表示モードを使用することで変数の16進値のスパイが可能になります。



ダンプ表示は、印刷不可能なキャラクタを含む文字列変数をスパイしたり解析するのに非常に便利です。

A.16.1 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.17 ST、IL プログラムのデバッグ

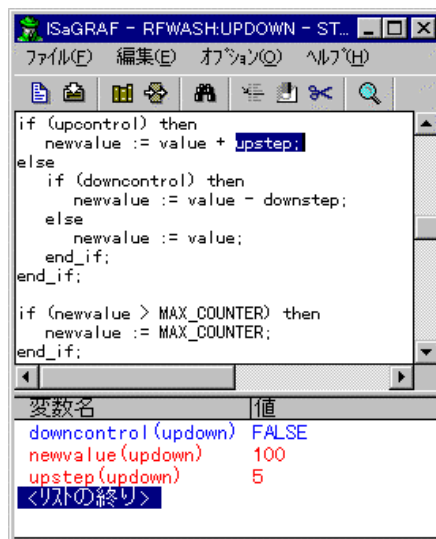
ST、IL プログラムのシミュレーション、デバッグ中は直接プログラムへの変更を加えることはできません。

- IL** IL プログラムではインストラクションがリストの形式になり、変数の現在値が同一行に表示されます。インストラクションをダブルクリックすることにより、対応する変数の値を変更することができます。
- また、「編集」-「スパイオバレント」により、現在のプログラムのスパイリストが開き、変数をモニタすることができます。



行	ラベル	Inst.	Op.	値	コメント
0000		LD	led1	=off	
0001		ST	led5	=off	
0002		LD	led2	=off	
0003		AND	bflash	=on	
0004		ST	led6	=off	

- ST** ST プログラムではエディタウィンドウにスパイリストウィンドウが組み込まれます。スパイリストと同じ要領で、モニタしたい変数をリストに追加していくことができます。2つのウィンドウ間のセパレーションをマウスでドラッグすることにより表示サイズの変更を行うことができます。



```

if (upcontrol) then
  newvalue := value + upstep;
else
  if (downcontrol) then
    newvalue := value - downstep;
  else
    newvalue := value;
  end_if;
end_if;

if (newvalue > MAX_COUNTER) then
  newvalue := MAX_COUNTER;
end_if;
  
```

変数名	値
downcontrol (updown)	FALSE
newvalue (updown)	100
upstep (updown)	5
<リストの終り>	

変数スパイのリストには変数名、現在値、コメントが表示され、それぞれのフィールドの幅はマウスドラッグにより変更することができます。



スパイリストの保存

「ファイル」-「保存」メニューコマンドにより、デバッグ中のプログラム名と同一名でスパイ変数リストが保存されます。このリストは次の ST、IL プログラムがデバッグ状態となる時に、自動的に開かれます。

また、デバッグウィンドウの「ツール」-「スパイリスト」メニューコマンドにより、このプログラム名のスパイリストを表示して修正、保存を行うことも可能です。



スパイリストへの変数追加

「編集」-「変数挿入」メニューコマンドにより、スパイリストに変数を挿入することができます。プロジェクト変数辞書の変数選択用のダイアログボックスから挿入する変数を選択するので、いちいち変数をマニュアル入力する必要がありません。このリストには最大32個までの変数を含めることができます。同一の変数名がリスト上に表示されることはありません。



ST テキストウィンドウで変数が選択(ハイライト)されている時は、「編集」-「スパイリストへ追加」あるいは、「スパイリストへの追加」ツールバーアイコンにより、スパイリストに変数を挿入することができます。変数を選択して右マウスクリックしてオープンされるポップアップメニューからの選択も可能です。



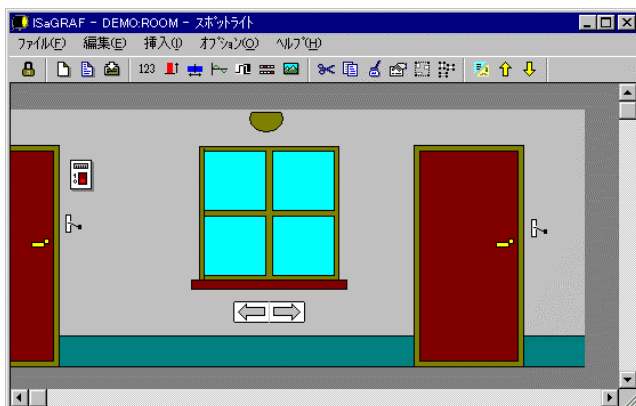
選択された変数の変更

「編集」-「変更」メニューコマンドにより、スパイリスト上で選択されている変数の変更を行うことができます。「編集」-「変数の切り取り」メニューコマンドにより、スパイリスト上から選択されている変数の削除を行うことができます。

A.18 SpotLight の使い方



ISaGRAF の SpotLight ツールはデバッグ中にモニタしたい情報をグラフィックやリストのスタイルで定義してモニタリングできます。グラフィック項目は ISaGRAF プロジェクトの変数にリンクされなければなりません。項目はオンライン中(デバッグ中)に新規に定義したり、アニメーションすることができます。



選択された項目の変数を書き込みする際には、グラフィックやレイアウト上の項目をダブルクリックするか、選択されているときにEnterキーの選択を行います。「ファイル」-「ロック」コマンドにより、作成した Spotlight ドキュメントの更新をロックすることができます。編集ロックがかかっているにもかかわらず、変数の値の更新はシンボルをダブルクリックすることで実現できます。

A.18.1 グラフィックレイアウトの構築

グラフィックレイアウト(画面)はバックグラウンド(背景)のグラフィックスやメタファイル上にグラフィックな変数項目を配置させることで作られます。この変数項目はデバッグ中にアニメーションされます。画面を構成する際には、まず、グラフィックスファイルを挿入、グラフィック変数項目を挿入、変数項目とプロジェクト内の変数とのリンクを行います。



バックグラウンドピクチャ

ビットマップ(BMP)やメタファイル(WMF)をバックグラウンド上に配置します。グラフィックレイアウトに含まれるピクチャの数に制約はありません。配置されたグラフィックスは移動、サイズ変更が可能です。これらのファイルはリストレイアウトに切り替えたときには表示されません。SpotLight はペイントツールではないので、これらのファイルの修正は行えません。

「オプション」-「背景色」コマンドでレイアウトの背景を指定された単一色にすることができます。

注意: ビットマップファイルは大きなメモリを占有するために、最適化されたファイルサイズにしておくことをお勧めします。

123

テキスト表示

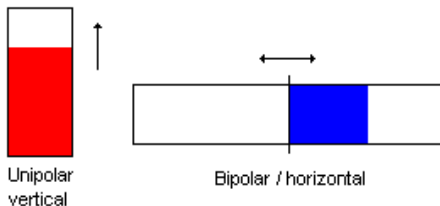
テキストは長方形の中にかかれる文字列です。このテキストには割り付けられた変数の値を表示します。この項目にはブール型、整数／実数型、可変長文字列型、タイマ型すべての変数を割り付けることができます。

長方形内の色やテキストの色の変更が行えます。更に、長方形サイズにあわせて文字のフォント指定を行うことができます。



棒グラフ、バイポーラ棒グラフ

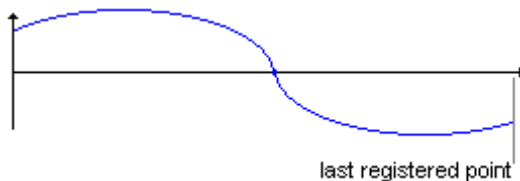
棒グラフでは割り付けられた整数／実数型変数の値が指定された色の長方形の形状で表示されます。垂直、水平の方向を設定できます。背景色も設定できます。通常の棒グラフは上下左右のいずれの方向でも設定可能です。バイポーラ型棒グラフでは2方向（正方向、負方向）を持ちます。バイポーラ棒グラフではスケールの最大値は2方向共に同じ値となります。



トレンドグラフ

レイアウト上に**トレンドグラフ**を挿入することが可能です。トレンドグラフは割り付けられた変数値の履歴を示します。正確な測定ツールではありませんが、複数の変数を配置することで変数間のタイミングも見ることができます。

トレンドグラフは最新の200点の値を蓄積、表示します。トレンドグラフのサイズを変更してもこの点数の変更はできません。



ブール型アイコン

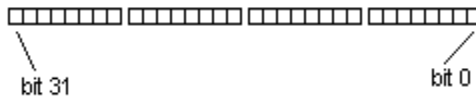
レイアウト上に**ブール型アイコン**が挿入できます。ブール型アイコンはブール型変数の状態をアイコンファイルの形状によって表現します。変数の値が FALSE (= 0) 時のアイコンファイル(.ICO)と TRUE (≠ 0) 時のアイコンファイルを割り付けることとなります。SpotLight はアイコンエディタではありませんので、アイコンファイ

ルの修正は行えません。これらのアイコンファイルは別のツールで準備しておく必要があります。



ビット展開

ビット展開は整数型変数をビット列(32ビットバイナリ)のグラフィック表示であらわすことができます。LSBは常に右側となります。整数型以外の変数を表示することも可能ですが、直接意味のある表示にはなりません。ビット列が正しく表示されるためには項目表示サイズの左右に十分な大きさが必要です。



選択、移動、サイズの変更

レイアウト上の項目を選択するためにはマウスで直接シンボルをクリックします。複数のシンボルをまとめて選択する場合はマウスで選択する領域を選びます。項目が選択されたときは、境界領域が小さな■によって囲まれます。

新しいシンボルを選択すると、その前に選択されていたシンボルの選択は解除されます。選択された項目を非選択とするためには、項目以外のバックグラウンドをマウスクリックします。

項目を移動する場合は、選択された領域の境界付近にマウスを近づけることにより、マウスカーソルの形状が移動用に変わります。ここで、マウスドラッグすることにより項目を移動させることができます。

シンボルのサイズの変更を行う際には、マウスを選択領域の■上に移動させ、マウスカーソルの形状が変化したところで、マウスドラッグします。この場合、ビットマップファイル、メタファイルも同様にリサイズされます。



グループ化／グループ解除

レイアウト上の複数の項目をグループ化して一つの項目にすることができます。「編集」-「グループ化」コマンドによりグループ化することができます。グループ化により、項目をまとめて移動することができますがサイズの変更はできません。グループ化された項目を元に戻すためにはグループ解除します。「編集」-「グループ解除」コマンドにより選択されたグループを解除することができます。

グループ内に別のグループやグラフィックスを含めることも可能です。グループ化された項目に関するスタイルの変更はできません。グループ化されてシンボルは、表示はされていますが、ダブルクリックによる変数値の変更はできません。グループ化された項目はリストレイアウトに切り替えると1つの項目にまとめられて表示されます。

A.18.2 リストレイアウト



「オプション」-「リスト／グラフ レイアウト」メニューコマンドか左のアイコンにより、グラフィックレイアウトとリストレイアウトモードの切り替えを行うことができます。

リストレイアウトモードでは、すべての項目がリスト上にならべられます。ひとつの項目の高さはスタイルによってあらかじめ決められています。ビットマップやメタファイルなどはこのモードでは表示されません。このモードでも項目の選択、項目のスタイル設定は可能です。ただし、複数の項目の選択、グループ化などは行うことができません。



「編集」-「リストの上／下へ移動」メニューコマンドにより選択された項目をリスト内で上下方向に移動させることができます。

A.18.3 スタイル設定

レイアウト上の項目のスタイルの変更、定義が行えます。シンボルを選択して、「編集」-「項目スタイル設定」コマンドを選択すると、項目のスタイルの設定が行えます。シンボルの選択はグラフィックレイアウト、リストレイアウトのいずれの場合も可能です。新規に項目を挿入する場合も、項目スタイルの設定ダイアログボックスが開きます。

設定ダイアログボックスでは以下のことが行えます。

■ スタイル設定:

項目の表示スタイル(単一テキスト、棒グラフ、トレンドグラフ…)をダイナミックに変更できます。前面色、背景色等も設定できます。ブール型アイコンが選択された場合は、指定するアイコンファイルはパス名付きで指定する必要があります。「...」ボタンを選択することでアイコンファイルを一覧して選択が可能です。

■ スケール:

棒グラフやトレンドグラフでの最大値を示します。パイボア棒グラフでは2方向共、同じ最大値となります。

■ 名前:

名前フィールドが選択されているときに「...」ボタンの選択により、プロジェクト内で宣言されている変数の選択が行えます。

☐ キャプション:

キャプションはグラフィック項目の近くに表示される変数名 + 値のテキスト表示です。キャプションの場所 (左右上下) や表示形式を選択できます。キャプションのカスタマイズはリストレイアウトモードに切り替えた場合には影響を受けません。

☐ 変数書き込み:

「変数書き込み」オプションをチェックすると、デバッグ中にグラフィックオブジェクトをダブルクリックすると、そのオブジェクトの変数の値を変更可能になります。

A.18.4 「ファイル」メニューコマンド

「ファイル」メニューコマンドにはレイアウトドキュメントの管理用に以下のコマンドが含まれます。



「ファイル」-「新規作成」メニューコマンドにより新規のレイアウトを作成できます。一つのプロジェクト内で作成できるレイアウトの数の制限はありません。SpotLight では1つのウィンドウで同時に2つ以上のレイアウトを開くことはできません。ただし、Spotlight ウィンドウを複数開くことにより、同時に数のレイアウトを使用することができます。



「ファイル」-「開く」メニューコマンドにより現在開かれているレイアウトを閉じて、選択されたレイアウトを開くことができます。また、レイアウトを選択するダイアログボックスでは「削除」ボタンにより登録されているレイアウトを削除することができます。ただし、この操作によって関連付けられているアイコン、ビットマップファイルなどは削除されることはありません。



「ファイル」-「上書き保存」メニューコマンドにより開かれているレイアウトをディスクに保存することができます。もし、レイアウトに名前(タイトル)が与えられていない場合は、これを入力する必要があります。レイアウト名は以下のルールに従う必要があります。

- 名前は最大8文字
- 最初の文字アルファベット(a-z)
- 以降は文字、数字、アンダースコア
- 大文字、小文字の区別なし

「名前をつけて保存」で既存のリストの名前を変更して保存することができます。

「ファイル」-「ロック」メニューコマンドにより開かれているレイアウトへの新規項目の追加、背景色の変更、項目のスタイル設定などをできなくすることができます。メニューコマンドの多くが選択不可となります。ただし、この状態でも既にレイアウト上にある項目への変数書き込みは可能です。間違ってレイアウトを変更させなくするためには有効な手段です。

A.18.5 バージョン 3.2 ユーザへの注意事項

SpotLight は、バージョン 3.0 あるいは 3.2 で作成されたグラフィックデータや変数のトレースのデータを読み込むことが可能です。これらのファイルは、「ファイ

「ル」-「開く」のダイアログに、作成されたバージョンのコメント付でリストアップされます。読み込んだファイルは、SpotLight で自由に編集することが可能です。

バージョン 3.2 で作成されたグラフィックデータがロック状態になっていた場合、編集する前に「ファイル」-「ロック」コマンドでロックを解除してください。

バージョン 3.2 で作成されたグラフィックデータや変数のトレースデータは、一旦オープンすると、終了時に SpotLight のデータフォーマットで保存することを促す為のメッセージボックスが自動的にオープンします。

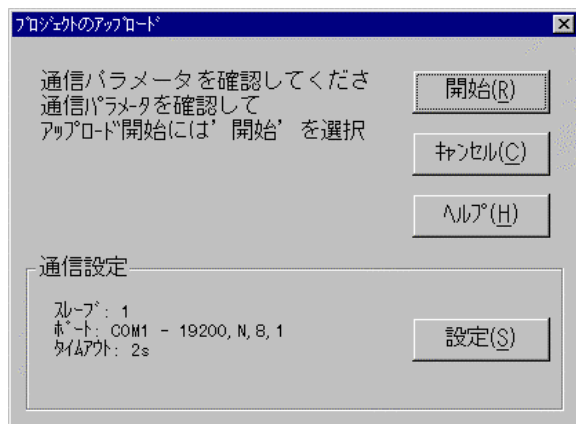
A.18.6 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.19 アプリケーションのアップロード

ISaGRAF はターゲットに保管されているアプリケーションコードをアップロードする機能があります。アップロード機能はターゲット内に保管されている圧縮されたアプリケーションのソース(EZS: Embedded Zipped Source code)をワークベンチに取り入れて、これを解凍することにより達成されます。



このアップロード機能を使用するためには、接続するターゲットバージョンが V3.23 以降であることと、前もってソースコードが添付されたアプリケーションコードがダウンロードされていることが前提となります。このアップロード用ソースコードの添付はオプション機能です。

A.19.1 プロジェクトのアップロード

“アップロード”ダイアログボックスはプロジェクト管理の“ファイル”メニューコマンドから開くことができます。ワークベンチ上で選択されているプロジェクトとアップロードされるプロジェクトとは関係がありません。ターゲットで実行中のアプリケーションをアップロードするためには以下のことを行う必要があります。

- 1- ターゲットとワークベンチが正しく接続されていること
- 2- 通信パラメータの設定（設定ボタンでリンク設定に従って行います）
- 3- “開始”ボタンを押す

圧縮ソース(EZS)をアップロードしてこれを解凍するためにはしばらく時間がかかります。ダイアログボックスのメッセージ表示によってアップロードの完了を知ることができます。さもなければエラーとなります。

アップロードされる ISaGRAF プロジェクト名はターゲットとの通信を通して読み出されます。もし、既にワークベンチ内に同じプロジェクト名が存在する場合は、上書きするか、名前の変更をすることができます。アップロードが完了した後でのプ

プロジェクトの登録をキャンセルすることはできません。これで アップロードされたプロジェクトを開くことができます。

❏ 発生する可能性のあるエラー

プロジェクトのアップロード時には以下のエラーが考えられます。エラーに関しては“アップロード”ダイアログボックスに表示されます。

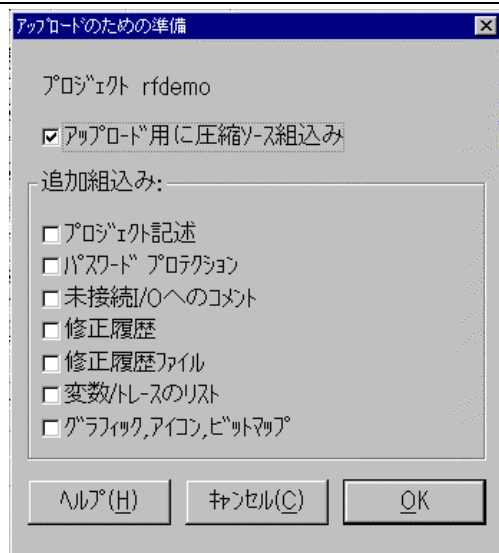
- ターゲットとの通信が確立しなかった
- ターゲットが Ver. 3.23 より以前のバージョンである
- ターゲットでアプリケーションが実行されていない
- ターゲットに圧縮ソース(EZS)がない

A.19.2 通信設定

“設定”ボタンによりワークベンチとターゲット間の通信設定を行うダイアログボックスを開くことができます。ここでの通信設定内容がターゲット側の通信プロトコルと一致している必要があります。

A.19.3 アップロードに備えた準備

ターゲット側からワークベンチ側へのアップロードが必要となる場合は、あらかじめアプリケーションコード生成時に圧縮ソース(EZS)の組み込みを設定しておく必要があります。このためには、“コード生成オプション”ダイアログボックスで“アップロード”ボタンを選択します。ここで表示されるチェックボックスで“アップロード用に圧縮ソース組み込み”をチェックすると、プログラムのデバッグ、アプリケーションコード生成のために必要な最小限のソースコードが組み込まれます。これ以外に組み込むことのできる個別の情報も、チェックボックスにより選択が可能です。ダイアログボックスから、圧縮ソースの内容を選択します。



注意: 圧縮ソースにはライブラリは含まれません。ライブラリとは例えばファンクション、ファンクションブロック、I/Oボード、I/O装置機器が含まれます。

アップロードの機構の理解のためには以下の項目も参照願います。

添付ファイル(追加組み込みファイル)

圧縮ソースに含まれる最小限のファイルに加えて、以下のファイルも組み込むことが可能です。これらのファイルを追加していくと、ターゲット側で要求されるメモリ量が増えますので注意が必要です。

プロジェクト記述

もし、組み込まない場合は、プロジェクトの記述はアップロードされた日付のみを示します。

パスワードプロテクション

アップロードされたプロジェクトはパスワードプロテクションがされていないので、必要な場合はここで選択してパスワードプロテクションシステムを組み込む必要があります。

未接続I/Oチャンネルへのコメント

ISaGRAF には未接続のI/Oチャンネルに対してのコメントを記述できます。接続済みのI/Oチャンネルのみ使用する場合はチェックしないでください。

プロジェクト修正履歴

プロジェクトの修正履歴ファイルです。

修正履歴

個々のプログラムに対する修正履歴やユーザのメモ、コンパイラのメッセージ等が含まれるファイルです。このファイルはプロジェクトが大きくなるとターゲットに要求されるメモリも大容量となる可能性があります。

スパイリスト

デバッグ中に生成、保存されたスパイリストの設定ファイルです。

グラフィックス、アイコン、ビットマップファイル

ワークベンチ上のプロジェクトディレクトリ内に含まれている Spotlight 用アイコン、ビットマップファイルなどのグラフィックスファイルです。この選択によりターゲット側で大容量メモリが必要となる場合があります。

A.19.4 ターゲットにおける圧縮ソースの保管

圧縮ソース (Embedded zipped source: 以降は **EZS** と表現します。) はリソースとしてターゲット内にアプリケーションコード (中間コード) と共に保管されます。従って、圧縮ソースを選択する場合は、別のリソースの名前に EZS を指定することはできません。別の名称のリソースの付加への制限や既にユーザが組み込んだリソースに影響を与えることはありません。

リソースに関する詳細はワークベンチガイドの「コード生成」を参照願います。

A.19.5 ターゲットに必用なメモリ容量

圧縮ソース (EZS) にはターゲット内にメモリの余裕が必要となります。最小限の EZS サイズ (プロジェクトソースのみを選択した場合) はアプリケーションコードサイズの約 1.5 倍程度となります。従って、EZS を含めた全体のサイズはアプリケーションコードの約 2.5 倍となります。

EZS はアプリケーションコードと同一のデータセグメント上に保存されるため、セグメント化されたメモリ制約を持つターゲット (例えば、DOS では 64KB サイズ) の場合、この点も注意が必要です。

A.19.6 アップロードプロジェクトに関して

アップロードされたプロジェクトには再度コード生成するために必要なファイルは全て含まれています。ダウンロード前に選択されたオプションによっては、更にプロジェクト記述、修正履歴などのファイルも含まれています。

デバッグ前には、一度アップロードされたプロジェクトでアプリケーションコード生成を行う必要があります。

ワークベンチ上でのデバッガーを起動して、ターゲットとのバージョン (CRC チェック) が異なっていると表示されますが、必要であれば生成されたコードをターゲット側にダウンロードすることによりこの表示をなくすることも可能です。

注意: ライブラリは圧縮ソースと併にはダウンロードされません。プロジェクトをアップロードして再度コード生成される前には、必要なファンクションやファンクションブロックライブラリがワークベンチ上にインストールされていることが必要となります。

このアップロード機能はターゲット側で要求されるメモリをできる限り少なくするために、モニタ/デバッグする際に必要なワークベンチ側のファイルが圧縮ソースコードと併にはダウンロードされません。従って、プロジェクトのモニタ/デバッグのみを行いたい場合でも、アップロードが完了の後でアプリケーションコード生成を行う必要があります。

A.19.7 バージョン間の互換性

アップロード機能は ISaGRAF バージョン 3.23 以降でワークベンチとターゲット間の通信プロトコルの拡張によってサポートされるようになりました。

ターゲット側がバージョン 3.23 以前 (3.0x, 3.20) の場合でも、ワークベンチ側からターゲット側へ圧縮ソース (EVS) をダウンロードすることは可能です。EVS は単なるアプリケーションコードに追加されたリソースとして扱われます。ただし、このようなターゲットではアップロード機能のための通信プロトコルの拡張が含まれていないために、ワークベンチ側へのアップロードを行うことができません。

A.20 デバッグ(診断)ツールの使い方

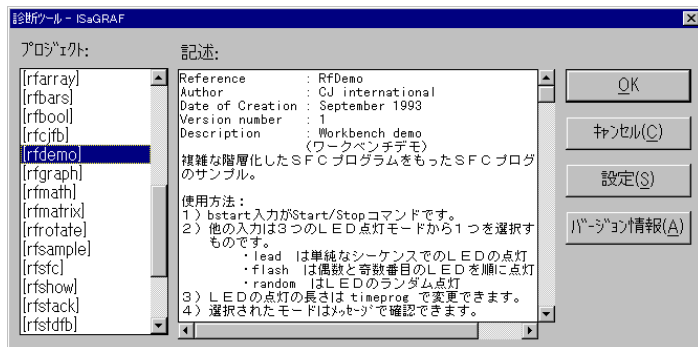
「診断ツール」は ISaGRAF デバッガーのサブセットで、デバッグの機能を限定して使用することができます。診断ツールによりプログラム内で使われている変数の操作を行なうことができます。

ISaGRAF ワークベンチに含まれているデバッガーと異なり、診断ツールは最終的なアプリケーションのメンテナンスのため使用します。診断ツールの起動は ISaGRAF グループまたはスタートメニューで以下のアイコンをダブルクリックすることで起動できます。

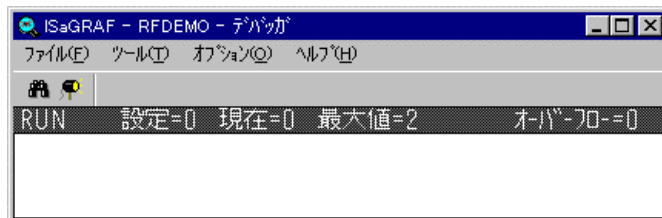


以下のダイアログボックスが表示されます。

ここで既存の、既にダウンロードされた ISaGRAF アプリケーションに対して機能制限されたデバッガーを起動することができます。



「OK」ボタンは選択されたプロジェクトでデバッガーをオープンします。「設定」ボタンは ISaGRAF ワークベンチとターゲット間の通信パラメータを設定します。詳細は「プログラム管理」の章を参照願います。



デバッグ(診断)ツールの使い方

注意: この診断ツールではプログラムのダウンロード、停止、更新は行なえません。もし、選択されて開かれたプロジェクトがターゲット内でのアプリケーションと異なっている場合は操作ができません。

診断ツールでのメニューは通常のデバッグ(シミュレーション)のメニューのサブセットとなっており、下記の機能が使用可能です。

- 変数のスパイリスト
- スポットライトによるグラフィックデバッグ

A.21 シミュレータの使い方

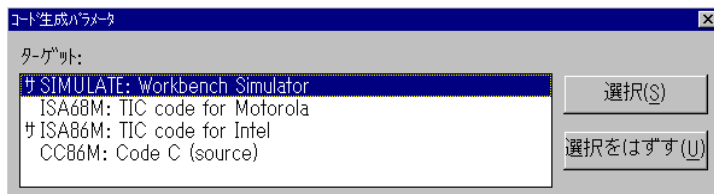


プログラム管理ウィンドウの「デバッグ」―「シミュレーション」コマンドにより、ISaGRAF シミュレータがデバッグとともに起動されます。シミュレーションはオフラインデバッグと言えます。このシミュレーションでは ISaGRAF のターゲットシステムを Windows 上で完全にシミュレーションします。標準で提供しているファンクション・ファンクションブロック・演算子に関しても使用可能です。ただし、C 言語ファンクションとファンクションブロックに関しては標準で提供しているもののみサポートしています。(ユーザ定義の C 言語関数をシミュレーションする場合は、ワークベンチ開発キット: オプションソフトによりカスタマイズする必要があります。) I/O ボードは仮想 I/O パネルとしてグラフィカルに Windows 上でシミュレーションされます。どの種類の I/O ボード(バーチャル I/O も含めて)に関してもグラフィカルシミュレーションされます。

A.21.1 デバッグとのリンク

シミュレーションは ISaGRAF デバッガ―との間の通信を完全にサポートしているので、オンラインデバッグで上と同じ操作が可能です。ただし、「ファイル」―「アプリケーションのスタート」、「アプリケーションのストップ」、「ダウンロード」、「アプリケーションの更新」コマンドに関しては無効となります。

シミュレーションを行うためには前もって、シミュレーション用のアプリケーションコードの作成が必要です。このためには、プログラム管理ウィンドウの「コード生成」―「コンパイラオプション」の設定でシミュレーション用のコードが選択されていることを確認した上で、「アプリケーションのコード生成」を実行する必要があります。

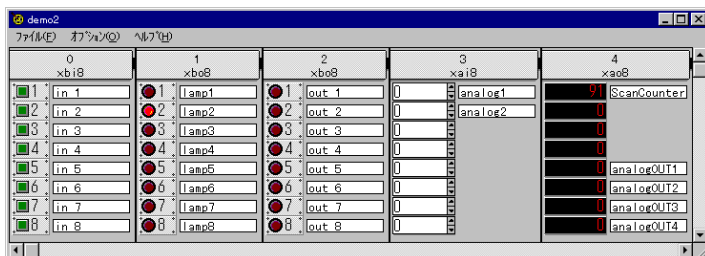


カーネルシミュレーションウィンドウ(仮想 I/O パネル)やデバッグウィンドウを開くことで、デバッグセッションで開かれているウィンドウは全て閉じます。

A.21.2 I/O シミュレーション

I/O ボードが仮想 I/O パネルウィンドウにスロット番号毎にグラフィカルにシミュレーションされます。ISaGRAF 標準の I/O ボード(ブール型、整数・実数型、可変長文字列型)を扱うことができます。ボード上にはスロット番号、I/O ボード名、チャンネル番号が表示されます。

入力ボードはボタンあるいはフィールドとともに表示され、出力ボードは LED あるいはフィールドとともに表示されます。I/O 接続エディタでバーチャル I/O ボードとして設定されている場合も、ボードはグラフィカルにシミュレーションされます。



Boolean型入力ボード: 入力ボードは小さなボタンと一緒に表示されます。チャンネル番号がボタン横に表示されます。ボタンが押されると該当する入力変数がTRUE状態となり、再度押されるとFALSE状態となります。右マウスをクリックすると、押されている間だけTRUE状態になります。

Boolean型出力ボード: Boolean型出力ボードはLED表示と一緒に表示されます。チャンネル番号がLED横に表示されます。値がTRUE状態の時LEDが点灯します。

Integer型入力ボード: Integer型入力ボードは入力用数値フィールドと一緒に表示されます。数値フィールドをクリックして数字を入力することで対応した変数名にアナログ値を代入することができます。**ENTER**キーの入力は必要ありません。あるいは、上下ボタンをクリックすることで入力することもできます。アナログ値は10進数、16進数いずれでも入力可能です。小さな▲の UP/DOWN ボタンによって、値をインクリメント/デクリメントすることができます。

Integer型出力ボード: Integer型出力ボードは出力用数値フィールドと一緒に表示されます。数値フィールドには変数の値が10進数あるいは16進数表示されます。この値の変更はできません。

可変長文字列型入力ボード: 文字列入力ボードはテキスト入力フィールドと一緒に表示されます。テキスト入力フィールドにカーソルを移動し、文字列入力を行うことで対応した変数への文字列入力ができます。**ENTER**キーの入力の必要はありません。

可変長文字列型出力ボード: 文字列出力ボードはテキスト出力フィールドと一緒に表示されます。この文字列の変更はできません。

A.21.3 ライブラリコンポーネント

ISaGRAF シミュレータは標準変換関数、関数、標準ファンクションブロックをサポートしています。以下にサポートされているオブジェクトのリストを行います。

変換関数:

DCB BCDコードのバイナリ変換 (Decimal Coded Binary)
SCALE LEDのメータ表示 (Volt meter led indicator)

□ ファンクション:

abs	絶対値 (absolute value)
acos	アークコサイン (arc cosine calculation)
ArCreate	整数配列の生成 (create array of integer)
ArRead	整数配列の読み込み (read array of integers)
ArWrite	整数配列の書き込み (write array of integers)
ascii	アスキーコードに変換 (character => ascii code)
asin	アークサイン (arc sine calculation)
atan	アークタンジェント (arc tangent calculation)
char	文字に変換 (ascii code => character)
cos	コサイン (cosine calculation)
delete	文字列の一部削除 (delete sub-string)
expt	指数 (exponent calculation)
find	文字列検索 (find sub-string)
insert	文字列挿入 (insert sub-string)
left	文字列左側抽出 (extract left sub-string)
limit	値制限フィルタ (bound value between limits)
log	常用対数 (logarithm (base 10))
max	最大値 (maximum of two integers)
mid	文字列抽出 (extract middle sub-string)
min	最小値 (minimum of two integers)
mten	文字列の長さ (length of a message)
mod	余り (modulo calculation)
mux4	4入力マルチプレクサ (multiplexer 4 entries)
mux8	8入力マルチプレクサ (multiplexer 8 entries)
odd	奇数パリティ (odd parity)
rand	ランダム整数値 (random integer value)
replace	文字列置き換え (replace sub-string)
right	文字列右側抽出 (extract right sub-string)
rol	ビット左回転 (rotate left)
ror	ビット右回転 (rotate right)
sel	セレクト (binary selector)
shl	ビット左シフト (shift left)
shr	ビット右シフト (shift right)
sin	サイン (sine calculation)
sqrt	平方根 (square root)
tan	タンジェント (tangent calculation)
trunc	切り捨て (truncate decimal part)

□ ファンクションブロック:

average	移動平均 (running average)
blink	ブリンク (blinking signal)
cmp	比較 (full comparison)
ctd	ダウンカウンタ (down counter)
ctu	アップカウンタ (up counter)
ctud	アップ・ダウンカウンタ (up/down counter)
derivate	微分器 (derivative regulation)
f_trig	立ち下がりがり検出 (falling edge detection)

hyster	ヒステリシス (hysteresis)
integral	積分器 (integral regulation)
lim_alm	リミットアラーム (alarm on limits)
r_trig	立ち上がり検出 (rising edge detection)
rs	dominant bistable のリセット (reset dominant bistable)
sema	セマフォ (semaphore)
sr	dominant bistable のセット (set dominant bistable)
stackint	整数スタック (stack of integer values)
tof	オフタイマー (off-delay timer)
ton	オンタイマー (on-delay timer)
tp	パルスタイマー (pulse timer)

ユーザ定義のC言語変換関数、ファンクション、ファンクションブロックはISaGRAF シミュレーションには標準では組み込まれません。通常、ユーザ定義関数はターゲットシステムに依存するため、Windows上で実現できない場合があります。

ISaGRAF シミュレータはユーザ定義の変換関数、関数、ファンクションブロックに対して以下の標準的な振る舞いを提供します。

- 新しい変換関数が実行されたとき、NULL変換となります。即ち、入力信号が全く変換されないで出力されます。
- C言語ファンクションが実行中は一切処理をしません。出力値は常にゼロ(あるいはNULL文字列)となります。
- C言語ファンクションブロックが実行中は一切の処理をしません。リターンパラメータは常にゼロ(あるいはNULL文字列)となります。

ただし、ワークベンチ開発ツールキット(オプションソフト)によりシミュレーションにユーザ定義の関数などの表示を組み込むことも可能です。

A.21.4 オプション

「オプション」メニューから仮想I/Oパネル表示の変更することができます。シミュレーション中にいつでもコマンドをセット、リセットできます。



- ⇒ 「カラーモード」コマンドにより、I/Oチャネル表示はカラービットマップ表示となります。モノクロディスプレイの場合は白黒表示にするためにこのオプションをはずして下さい。

- ⇒ 「**変数名**」コマンドにより、I/Oチャネル右横に変数名のラベルが表示されます。I/Oボードのシミュレーションパネルのサイズを小さくするためにはこのオプションをはずして下さい。
- ⇒ 「**16進数値**」コマンドにより、入力・出力アナログチャネルが16進数で表示、入力されます。
- ⇒ 「**常に前面**」コマンドにより、仮想I/Oパネルが常に画面の前面で表示されます。

A.21.5 入力信号状態の保存／読み込み

ISaGRAF シミュレータを使用して、I/Oシミュレーションパネルのトグルボタンやエディットコントロールボックスを使用した操作によって、入力チャネルの強制入力を行うことが可能です。これらの入力チャネル状態を保存したり、読み込んだりする為に下記のコマンドが、「**ツール**」メニューにあります。

入力パターンのロード ファイルに保存された入力チャネルの値を入力チャネルに設定します。

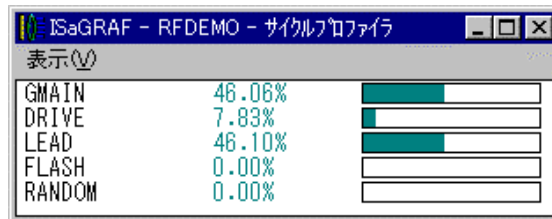
入力パターンのセーブ 入力チャネルの状態をファイルに保存します。この設定は「入力パターンのロード」によって呼び出すことができます。ファイルはプロジェクトディレクトリに保存されます。

メモ: 名前が付けられている入力チャネル(変数の割り付けが行われている入力)のみが保存されます

A.21.6 サイクルプロファイラ

サイクルプロファイラは、アプリケーション中のプログラム、ファンクション、ファンクションブロックのそれぞれの実行時間が1サイクル中どのように分布しているかを表示できる非常に便利な診断機能です。このツールには、アプリケーションのパフォーマンスのチェック機能や、どの部分のコードを最適化すべきかを調べるような時にも役立ちます。

サイクルプロファイラの起動は、シミュレータウィンドの「**ツール**」-「**サイクルプロファイラ**」のメニューコマンドで行います。そして、プログラム、関数、ファンクションブロックのそれぞれの実行にかかる時間をパーセンテージで表示します。



「**表示**」-「**平均**」のオプションをチェックすると、アプリケーションがスタートしてから、あるいは、最後に「**表示**」-「**リセット**」コマンドを実行してからの平均を計算して表示します。

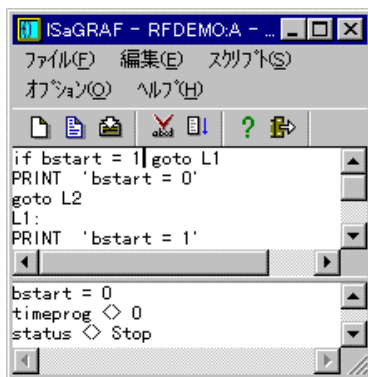
「表示」-「平均」のオプションがチェックされていない時は、直前のサイクル実行中に測定された情報を表示します。アプリケーションの実行モードを「**サイクルモード**」に設定してこの機能を使用すると、アプリケーションのそれぞれのコンテキストによる実行時間の測定をすることも可能です。

「表示」-「コピー」コマンドで、プログラム名とパーセンテージをクリップボードにテキスト形式でコピーすることができます。このデータを、ドキュメントやスプレッドシートに貼り付けることができます。

注意: この計測は精密なものではありません。パーセンテージの計算は TIC コードの命令数のカウントによって行われます。C言語ファンクション／ファンクションブロックの処理時間は含まれていません。
ファンクション、ファンクションブロックに表示される値は、1サイクル中に呼び出される全ての処理時間の合計値となっています。
また、Cソースジェネレータを使用して作成されたターゲットの場合は、信頼できるデータを表示することはできません。

A.21.7 シミュレーション スクリプト

ISaGRAF シミュレータには、シミュレーション用のスクリプトを作成し実行するツールがあります。スクリプトは、ST 言語に似たテキスト言語、シミュレータ上でのテストの自動化を可能にします。シミュレーションスクリプトエディタは、シミュレーションボードウィンドウ上で「**ツール**」-「**シミュレーションスクリプト**」を開きます。



上側のウィンドウは、スクリプトを入力するテキストエディタです。使用方法は、他の ISaGRAF テキストエディタと同様です。また、「**オプション**」メニューからは、タブ設定を行うことができます。

下部のウィンドウには、スクリプトが実行されたときの出力メッセージが表示されます。ウィンドウのセパレータは自由にドラッグできサイズ変更が可能です。スクリプトの編集時は、このウィンドウは隠れている場合がありますが、スクリプトが実行されると自動的に表示されます。



スクリプトの編集

スクリプトファイルに関する操作は「ファイル」メニューから行います。

新規作成..... 新規スクリプトファイルの作成

開く..... 既存のスクリプトファイルを開きます

上書き保存..... スクリプトテキストと出力ウィンドウの内容を保存します

名前を付けて保存... スクリプトテキストと出力ウィンドウの内容を別の名前で保存します

それぞれのスクリプトに対し、2つのファイルがプロジェクトディレクトリに作成されます。

<スクリプト名>.SCC スクリプト テキスト (命令)

<スクリプト名>.SCO 出力ウィンドウの内容

ここで、<スクリプト名>は、スクリプトの名前を指しています。これらのファイルは、通常のテキストファイルで、一般のテキストエディタでオープンすることができます。



スクリプトの編集で、「編集」-「シンボルの挿入」コマンドで宣言された変数名をカーソル位置に挿入することが可能です。



スクリプトの実行

スクリプトは実行する前に文法チェックとコンパイルする必要があります。「**スクリプト**」-「**スクリプト実行**」コマンドが選択されたとき必要に応じて文法チェックが自動的に行われます。下記の「**スクリプト**」メニューコマンドを使用します。



確認..... スクリプトの文法チェックとコンパイル



スクリプト実行..... スクリプトの実行

無題のスクリプトの場合、文法チェックを行う前に名前をつけて保存しなければなりません。名前が付けられたスクリプトの場合は、文法チェックの前に自動的にファイルへの保存が行われます。

スクリプトが実行されているときは、その内容は編集できません。スクリプトの実行が終了したときはメッセージが表示されます。また、下記の「**スクリプト**」コマンドで実行中のスクリプトを中断することも可能です。



スクリプト実行の中断..... 実行中スクリプトの中断

スクリプトの実行はターゲットサイクルの間で実行されます。サイクル中に無限ループがある場合は、シミュレータは、この無限ループを分割して実行し、サイクル実行が引き続き実行され、他の ISaGRAF アプリケーションがブロックされないようにしています。ISaGRAF のスクリプトインタプリタは、1サイクル中に同じ“ラベル”が2回カウントされたらスクリプトの実行を中断します。また、スクリプトの実行は“Cycle”や“Wait”のインストラクションによっても中断されます。

☐ スクリプト記述言語

スクリプト記述言語は、シンプルな ST 言語に似たテキスト言語です。しかし、各命令は1行ずつ別々に記述される点と、各行末のセミicolon(;)は不要な点がことなっています。ツールバー上の下記のボタンを使用して、使用可能なインストラクションを調べ、カーソル位置に挿入することが可能です。



インストラクションの挿入(キーワードとヘルプコメント)

下記の通り、いろいろなインストラクションがあります。

値を変数に代入するインストラクション

:=..... 代入

出力ウィンドウにメッセージを出力するインストラクション

Print..... テキストや変数の値の出力

PrintTime..... 現在時刻の出力

ISaGRAF のサイクルとの同期を取るためのインストラクション

Cycle..... 1サイクル実行

Wait..... 指定時間待ち

スクリプトの流れを制御する為のインストラクション:

ラベル スクリプト中のどこにでも配置できる

Goto ラベルへの無条件ジャンプ

If goto ラベルへの条件付ジャンプ

End スクリプトの終了

スクリプト言語は、大文字小文字を区別しません。コメントは ST 言語同様に、1行の行末に、"(*" と"*)"で囲むか、";" セミcolonを初めに付けて記述します。

":= " 代入

意味: 変数に値を強制的に代入します。内部変数、入力変数、出力変数に対して有効です。

文法: <varname> := <constant_expression>
<varname> = <constant_expression>

説明: <varname>は、変数辞書に登録された変数名、あるいは"%"の文字を接頭字として使用する直接表現変数です。
<constant_expression>は、指定した変数に対応する型の定数値です。ブール型変数に対しては TRUE/FALSE の代わりに1/0も使用できます。また、タイマー変数に対しては、"T#"や"TIME#"の接頭文字は省略可能です。

メモ: スクリプトによる入力変数に対する値の代入の場合は、変数のロックは不要です。対応する入力変数の表示の更新は、スクリプトから値が書きかえられた時に行われます。

注意: 変換関数の設定をしている入出力変数に対しての値の代入は行ってはいけません。スクリプトでは、変換関数や変換テーブルをサポートしていません。

例: MyBooVar := 1 (* same as TRUE *)
 MyIntVar := 1234
 MyRealVar := 1.2345
 MyMsgVar := 'Hello'
 MyTmrVar := t#12s

Print

意味: 出力ウィンドウへの文字列や変数の値の出力。出力ウィンドウの最後の文字から改行されて新しい行に表示が行われます。

文法: **Print** '<text>'
 Print <varname>

説明: <text>は、シングルクォーテーションで囲まれた文字列。
 <varname>は、変数辞書に登録された変数名、あるいは"%"の文字を接頭字として使用する直接表現変数です。

メモ: 変数の値の出力は、IEC 規格に応じたフォーマットで行われます。

例: Print 'Hello'
 Print MyBooVar

出力: Hello
 MyBoovar = TRUE

PrintTime

意味: 出力ウィンドウに、現在時刻を表示します。出力ウィンドの最後の文字から、改行されて新しい行に表示が行われます。

文法: **PrintTime**

メモ: 現在時刻の出力フォーマットは、Windows の現在のシステムの設定に依存しています。

例: Print 'Time now is:'
 PrintTime

出力: Time now is:
 15:45:22

Cycle

意味: 次の ISaGRAF のサイクルが実行されるまで、スクリプトの実行を一時停止します。

文法: **Cycle**

メモ: スクリプトの実行は、ISaGRAF のサイクルの最初に行われます。もしシミュレータが"サイクル"モードの場合、この"Cycle"命令はサイクル実行の直後に行われます。それ以降のスクリプトは、デバッガの「1サイクル実行」コマンドが実行された後、実行されることになります。

例: (* ISaGRAF のプログラムには、A を B にコピーするプログラムがありません*)
A := 0
Cycle
Print B
A := 1
Print B (* サイクルが実行されない / B に 1 がセットされない*)
Cycle
Print B

Output: B = 0
B = 0
B = 1

Wait

意味: 指定した時間が経過するまでスクリプトを一時停止します。

文法: **Wait <delay>**

説明: <delay> は、時間の定数を示す IEC 規格に応じた表現方法で記述します。"T#" や "TIME#" の接頭文字は省略可能です。この値は、10mS から1時間の間の値でなければなりません。

メモ: "Wait"インストラクションの精度は、ホストシステムの Windows に依存していますので、精密なものではありません。また、±1サイクル実行時間分の誤差を考慮する必要があります。

"Wait"インストラクションが実行されると、指定された時間が経過するまで ISaGRAF サイクルが実行され、その後スクリプトの実行が継続されます。

例: PrintTime
Wait 2s
PrintTime

Output: 15:45:27
15:45:29

ラベル

意味: ラベルは、スクリプト中のどこにでも配置できます。ラベルは"Goto"インストラクションのジャンプ先として使用し、スクリプトの流れを制御することが可能です。

文法: <labelname>:

説明: <labelname> は、英文字で始まり、英数字とアンダースコアなどの16文字以内のユニークな文字列です。宣言時には、コロン(":")を後に付けなければなりません。

メモ: ラベルを宣言した行に、他のインストラクションは記述できません。ラベル名は、変数辞書に宣言された変数名と同一のものが有ってはいけません。

例:

```
(* example of a script with an infinite loop *)
loop:
PrintTime
Wait 1s
Goto loop
```

Goto

意味: ラベルへの無条件ジャンプ

文法: **Goto** <labelname>

説明: <labelname> は、スクリプト中に宣言されたラベル名。

メモ: 逆方向ジャンプも可能です。無限ループの場合は、ISaGRAF のサイクル実行を維持する為に、スクリプトの実行はループ毎に分割されて実行されます。

例:

```
Print 'Before Jump'
Goto MyLabel
Print 'Within Jump' (* instruction never performed *)
MyLabel:
Print 'After Jump'
```

Output: Before Jump
After Jump

If Goto

意味: ラベルへの条件付きジャンプ。条件には、変数と変数の比較および、変数と定数の比較を使用します。

文法: **If** <var1> **test** <var2> **Goto** <labelname>
If <var1> **test** <constant_expression> **Goto** <labelname>

test に使用可能なものは、下記です。

```
= 両辺の値が同じ場合に真。  
<> 両辺の値が等しくない場合に真。  
< 左辺が右辺より小さい場合に真。  
<= 左辺が右辺以下の場合に真。  
> 左辺が右辺より大きい場合に真。  
>= 左辺が右辺以上の場合に真。
```

説明: <var1> <var2> は、変数辞書に登録された変数名、あるいは"%"の文字を接頭字として使用する直接表現変数です。

<constant_expression> は、指定した変数の型に対応した定数値です。ブール型変数に対しては TRUE/FALSE の代わりに1/0を使用することもできます。また、タイマー変数に対しては、"T#"や"TIME#"の接頭文字は省略可能です。

<labelname> は、スクリプト中に宣言されたラベル名です。

メモ: 逆方向ジャンプも可能です。無限ループの場合は、ISaGRAF のサイクル実行を維持する為に、スクリプトの実行はループ毎に分割されて実行されます。

例:

```
(* This script loops until MyVar is TRUE *)  
Loop:  
If MyVar = TRUE Goto TheEnd  
Print MyVar  
Goto Loop  
TheEnd:
```

End

意味: スクリプトの終了。

文法: **End**







メモ: スクリプトの最終行に"End"インストラクションを記述することは必須条件ではありません。

例:

```
(* This script loops until MyVar is TRUE *)  
Loop:  
If MyVar = FALSE Goto Continue  
End  
Continue:  
Print MyVar  
Goto Loop
```

A.21.8 ツールバーアイコン

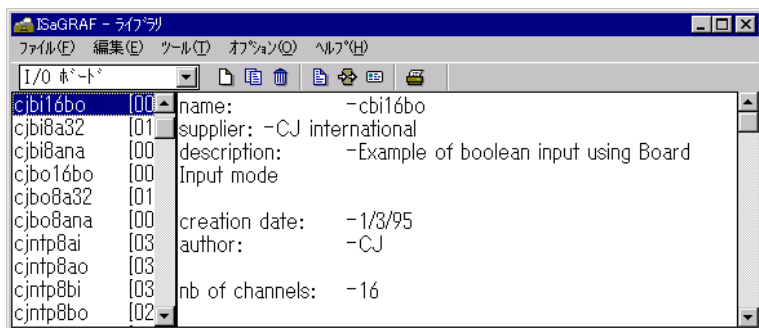
以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。

	新規スクリプト作成
	スクリプトを開く
	スクリプトの保存
	スクリプトのチェック
	Go
	インストラクションの挿入
	シンボルの挿入

A.22 ライブラリ管理ユーティリティの使い方

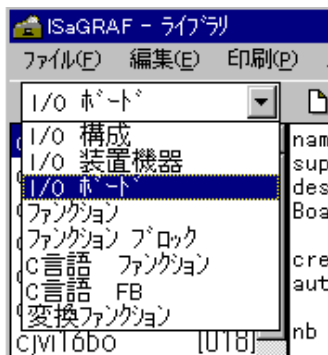


ISaGRAF ライブラリ管理ユーティリティは各種タイプのライブラリ群を管理しています。各タイプにつき1つのライブラリが用意されています。ライブラリ管理を使ってユーザが作成するドライバ、ファンクションなどのオブジェクトのパラメータ等を定義して登録しておくことができます。



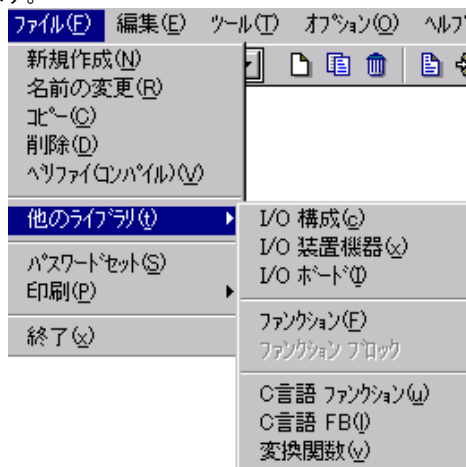
ISaGRAF ライブラリ管理の左側のウィンドウには選択されたライブラリの**エレメントリスト**を表示し、右側のウィンドウには選択されたエレメントの**技術メモ**を表示します。ライブラリ管理のメニューは選択されたライブラリのエレメントの作成、定義、変更のためのコマンドを含んでいます。「**ファイル**」-「**他のライブラリ**」または左上のコンボボックスより、ライブラリを選択することができます。

ISaGRAF ライブラリの選択は「**ファイル**」-「**他のライブラリ**」コマンド又はツールバーの左側のコンボボックスから行なうことができます。



A.22.1 ライブラリエレメントの管理:「ファイル」メニューコマンド

「ファイル」メニューにはオブジェクトとしてのライブラリを管理するコマンドが含まれています。



新規エレメントの作成

「ファイル」-「新規作成」コマンドにより、選択中のライブラリに新しいエレメントを追加します。以下のルールでエレメント名を入力します。

- 最大8半角文字
- 最初の文字は英文字(a-z)
- 以降の文字は英文字、数字、_ のいずれか
- ライブラリエレメント名には大文字小文字の区別はありません

テキストによるエレメントに対するコメント(技術メモ)が入力できます。以下の内容は必ず技術メモに含める必要があります。

- I/O構成の定義、
- I/Oボードに対するパラメータ、
- 関数やファンクションブロックに対するユーザインタフェース

C変換関数、Cファンクション、Cファンクションブロックが定義されるとソースコードのテンプレートが自動で作られます。



既存のエレメントの管理

「ファイル」-「名前の変更」コマンドにより、エレメントの名前を変更できます。「コピー」コマンドにより、同一ライブラリ内にコピーを作成します。既に同じ名前のエレメントが存在している場合は上書きされます。「削除」コマンドにより、エレメントが削除されます。「名前の変更」、「コピー」、「削除」コマンドにより、以下のコンポーネントも操作されます。

- 技術メモ
- I/O構成の定義
- I/Oボード、構成機器のパラメータ
- ファンクション、ファンクションブロックのインタフェース定義
- IEC言語で書かれたファンクション、ファンクションブロックのソースコード
- C変換関数、Cファンクション、Cファンクションブロックのソースコード



注意:「名前の変更」、「コピー」コマンドでは、C変換関数、C関数、Cファンクションブロックの技術メモ、ソースコード内部の名前は自動的に更新されません。



注意:「名前の変更」、「コピー」コマンドでは、IEC言語で記述されたファンクションの出力パラメータ(戻り値)の名前は自動的に更新されません。



パスワードプロテクションの設定

「ファイル」-「パスワードセット」コマンドにより、ライブラリ内の選択されたエレメントに対して各種レベルのパスワードプロテクションが設定できます。パスワードは選択されているエレメントのみに対して有効で、これは他のエレメントや他のライブラリに対しては無効です。パスワードプロテクションに関しては別章を参照願います。



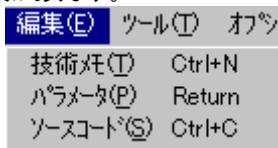
ファンクション、ファンクションブロックのベリファイ(コンパイル)

ライブラリでIEC言語によるファンクション、ファンクションブロックが選択されている場合、「ファイル」-「ベリファイ(コンパイル)」コマンドにより、選択された関数エレメントの文法チェックを行ない、対応したオブジェクトコードを生成します。これらは、ISaGRAF プロジェクト内で使われる前にエラーなしでコンパイルが完了している必要があります。IEC言語によるファンクション、ファンクションブロック以外のライブラリが選択されている場合はこのコマンドは使えません。



技術メモ

「編集」-「技術メモ」コマンドにより、選択されているエレメントの技術メモの入力、修正を行なえます。技術メモはオンラインヘルプ的に ISaGRAF プログラム作成時にエディタから“情報”として呼び出すことができます。技術メモには、エレメントの主な目的、プログラム入出力パラメータインタフェース、制限その他の必要な情報を含んでいる必要があります。



「ツール」-「標準技術メモフォーマット」コマンドにより、選択中ライブラリの全エレメントの作成に対して有効な標準技術メモフォーマットが作成できます。新規のエレメントを作成する場合にこのフォーマットが技術メモのテンプレートとして使われます。ユーザオリジナルの技術メモフォーマットが作成できます。



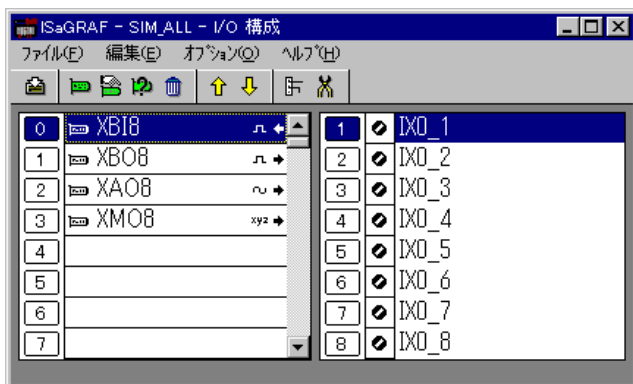
ライブラリエレメントのパラメータ

「編集」-「パラメータ」コマンドにより、現在選択中のライブラリエレメントのパラメータを修正できます。ライブラリエレメントのパラメータは ISaGRAF プログラム

とエレメントの間のインタフェース部分を定義しています。ライブラリ毎にパラメータの持つ意味が異なります。

● I/O構成のパラメータ

I/Oボードの集合体の構成を定義していて、I/Oチャンネルにはデフォルトで変数名が割付けられています。



● I/Oボード、I/O装置機器のパラメータ

ボードの物理的、論理的な構成を定義しています。



● ファンクション、ファンクションブロックのパラメータ

エレメントのST言語で記述した場合の引数に相当するインタフェース部を定義します。



● 変換関数のパラメータ

標準の設定済みのインタフェースを使うため存在しません。



ライブラリエレメントのソースコード

ライブラリ管理ユーティリティにより、変換関数、ファンクション、ファンクションブロックのライブラリエレメントのソースコードを管理できます。
 ファンクションのソースコードは IEC 言語 (LD, FBD, ST, IL) で書かれています。
 C言語ファンクション、C言語ファンクションブロック、C言語変換関数のソースコードは2つのファイルに分かれています。1つ目は**ソースヘッダー(*.h)**でエレメントのパラメータからくるインタフェース部分を定義しています。2つ目は本来の**ソースコード(*.c)**でエレメントの処理の実現部分を定義しています。
 Cファンクション、Cファンクションブロックのソースコードの場合、新しいライブラリエレメントが作成されるとライブラリ管理ユーティリティはソースコードのテンプレートファイルを自動生成します。パラメータ定義に基づいたソースヘッダーも自動生成します。ISaGRAF テキストエディタでソースコードを完成させることができます。
 C言語ソースファイルの定義に関しては、ターゲットマニュアル(Cプログラミングテクニック)を参照願います。

(注) :ライブラリ管理ユーティリティはI/Oボードに関するソースコードの管理を含んでいません。



ライブラリエレメントのアーカイブ

「ツール」-「アーカイブ」コマンドで、ISaGRAF のアーカイブマネージャを起動して、ライブラリエレメントのセーブ／復元を行う事ができます。まず、初めにライブ

ラリを選択してから、アーカイブコマンドを使用して下さい。この時、選択されたライブラリのリストが表示されます。

A.22.2 I/O 構成ライブラリ

I/O構成ライブラリにより新規作成するプロジェクトに定義済みのI/O構成を設定することができます。ここでは、以下の内容を定義します。

- I/Oボード設定
- I/Oボードパラメータ用のデフォルト値
- I/Oチャンネル用のデフォルト名

新規の ISaGRAF プロジェクトがライブラリのI/O構成を選択した場合、I/Oボードの選択、チャンネルに使用されているI/O変数名の辞書登録、変数のI/O接続が自動的に行なわれます。



I/O構成の定義

I/O構成の定義は ISaGRAF のI/O接続エディタ(詳細はI/O接続エディタの使用法参照)で行ないます。I/O構成に新しいI/Oボードを挿入する際にボードの全チャンネルにデフォルト変数名が割り当てられます。(プロジェクト作成時のI/O接続エディタにおいては各チャンネルに手動で変数を割り当てる必要があります。)標準のデフォルト変数名のフォーマットは以下のようになります。

<方向><タイプ><スロット番号>_<チャンネル番号>

<方向>

最初の文字はI/Oチャンネルの方向を示します。

"I" 入力チャンネル
 "O" 出力チャンネル

<タイプ>

2番目の文字はI/Oチャンネルのタイプを示します。

"X" ブール型
 "D" 整数型
 "M" 可変長文字列型

例として以下に標準的なI/O変数名を示します。

IX0_7 ブール型入力でスロット番号0、チャンネル番号7
 QD2_4 整数型出力でスロット番号2、チャンネル番号4

「編集」-「ボードパラメータ/チャンネルのセット」コマンド(あるいはチャンネル部分のダブルクリック)により、I/Oチャンネルに割り当てられているデフォルト変数名を変更できます。

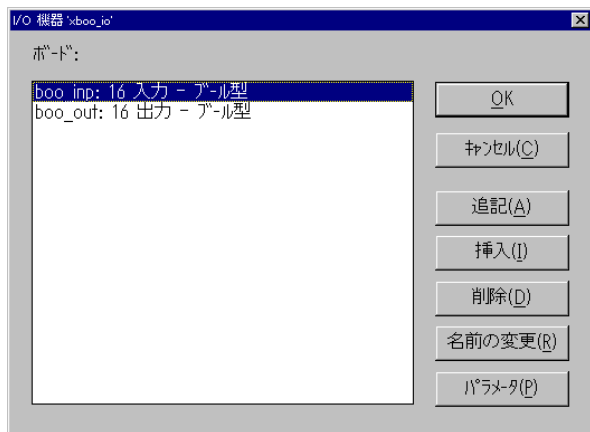
A.22.3 I/O 装置機器ライブラリ

I/O装置機器とは、異なったタイプや方向を持った複数のI/Oボードを組み合わせたI/Oデバイスです。個々のボードは同一タイプ(ブール型、整数型、可変長文字列型)でかつ同一方向(入力、出力)を持ちます。I/O装置機器は複数のI/Oボードを持ちますが、ラック内の1つのスロットしか占有しない装置です。



I/O装置機器の定義

I/O装置機器の定義をするにはまず、複数のI/Oボードのリストを定義する必要があります。更に、それぞれのI/Oボードに対してI/Oボードパラメータダイアログボックスを使って詳細なパラメータを入力する必要があります。



I/Oボードのリストを作成する場合はI/O機器ダイアログボックスで、“追記”ボタンによりI/Oボードリストの最後に新しいボードを追加することができます。“挿入”ボタンにより選択されているボードの前に新しいボードを挿入します。“削除”は選択されているボードの削除を、“名前の変更”は選択されているボードの名前の変更を行ないます。“パラメータ”ボタンにより、選択されたボードのパラメータ変更を行ないます。パラメータ設定は以下の項を参照願います。I/O装置機器は**最大16**のI/Oボードを持つことができます。I/O装置機器内のそれぞれのボードの名前は最大で**半角8文字**です。

A.22.4 I/Oボードライブラリ

I/Oボードライブラリはアプリケーションプログラムの変数とターゲットハードウェア(I/O)との間のインタフェースとなる部分です。プロジェクトアプリケーションを作成時に全入出力変数はターゲットのI/Oボードのチャンネルに割り当てられます。I/Oボードは**名前**とボードのサプライヤを示す**OEMキーコード**を持っています。この他、I/Oボードのチャンネル数、方向、タイプ等が記述されています。



I/Oボードパラメータ

I/Oボードパラメータには2つの部分があります。各ボードで共通な部分とボード固有な部分です。共通的な部分に関してはどのI/Oライブラリボードに対しても必要であり、I/Oボードパラメータ定義ボックスの上部で入力されます。ウィンドウの下部分に表示されるボード固有の部分は、ハードウェアサプライヤによって提供されるものです。

“**OEMキーコード**”はハードウェアサプライヤを定義する番号です。同じサプライヤは同一のOEMキーコードを持つ必要があります。OEMキーコードは**16ビット unsigned word**で、**16進数**で表現される必要があります。ICS Triplex ISaGRAF Inc. **インターナショナル社**で予約されている番号は1です。

“**チャネル数**”はボードで扱えるチャネル数です。“**タイプ**”はチャネルに接続される変数のタイプです。“**方向**”は接続される変数が入力か出力かを示します。

注意: 1枚のボードで異なったタイプ、方向をもつチャネルを持つことはできません。

OEMパラメータ

I/Oボードパラメータ定義ボックスの下部のOEMパラメータはボードのサプライヤにより定義されるものです。ボード固有のパラメータとなります。1枚のボードあ

たり最大**16個**のOEMパラメータ(割り込み番号、I/Oアドレス、メモリアドレス、...)がセットできます。ボードによってはOEMパラメータが不要な場合もあります。各々のパラメータのフォーマットや識別子はハードウェアサブライヤが定義する必要があります。

左側のグループボックスにはOEMパラメータのリストが表示されます。個々のパラメータは**パラメータ名**と論理番号(**0-15**)を持ちます。右側には選択されているパラメータの記述が表示されます。“**消去**”ボタンにより選択中のOEMパラメータを削除できます。

注意: 消去コマンドは取り消すことができません。

定義されたOEMパラメータは、アプリケーション開発者により、入出力変数とI/Oを接続する「I/O接続エディタ」上の設定で使用されます。パラメータ名は以下のルールに従う必要があります。

- パラメータ名の最大長16半角文字
- 最初の文字は英文字(a-z)
- 以降の文字は英文字、数字、_ のいずれか

「**フォーマット**」で、OEMパラメータの**内部フォーマット**を指定します。OEMパラメータの内部フォーマットとI/O接続エディタ上でのOEMパラメータの**入力フォーマット**は異なります。

以下に**内部フォーマット**例を示します。

word..... unsigned 16 bit word
long..... unsigned 32 bit word
word hexa..... unsigned 16 bit word
long hexa..... unsigned 32 bit word
ブール型 unsigned 16 bit word (最下位ビットのみが使われます)
文字 unsigned 16 bit word (最下位バイトのみが使われます)
可変長文字列..... 16バイトの文字列配列でNullでターミネイトされる
浮動小数..... single precision 32 bit floating value

I/O接続エディタ使用時の**入力フォーマット**は以下のようになります。

word..... unsigned decimal word
long..... decimal long word
word hexa..... unsigned hexadecimal word
long hexa..... unsigned hexadecimal long word
ブール型 "true" または "false"
文字 single character
可変長文字列..... ascii string (15 characters max)
浮動小数..... single precision floating value

「**アクセス**」ボックスでは、パラメータをエンドユーザからどのようにアクセスさせるかを定義します。

“**ユーザ定義**”..... チェックボックスにより、パラメータをユーザが定義するかどうか(I/O接続で表示するか)を決めることができます。これ

がチェックされるとI/O接続エディタを使用時にユーザが入力する必要があります。

“リードオンリー”... チェックボックスにより、ユーザはI/O接続エディタでパラメータを見ますが、変更はできません。パラメータの値は定数として扱われます。

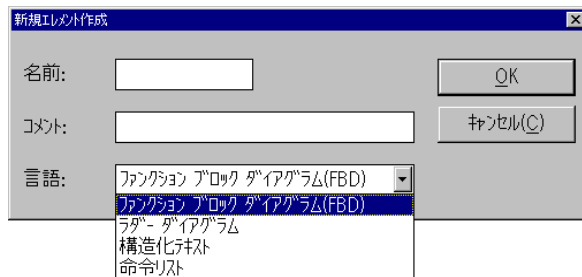
“隠れた”..... このチェックを行うと、パラメータは定数として扱われ、ユーザはI/O接続エディタ上でパラメータを見ることができません。

“デフォルト”..... 値はI/O接続エディタの入力フィールドに書かれる初期値を示します。「ユーザ定義」がチェックされていない場合は、パラメータ値は定数であり、かつI/O接続エディタに表示されません。チェックされていた場合は、パラメータのデフォルト値として表示されます。

デフォルト値として入力される値のタイプはフォーマットで選択したタイプと一致している必要があります。

A.22.5 IEC言語によるファンクション、ファンクションブロック

ISaGRAF はIEC言語で書かれたファンクションやファンクションブロックのライブラリを持つことができます。扱える言語は**FBD**、**LD**(ラダーダイアグラム)、**ST**(構造化テキスト)、**IL**(命令リスト)のいずれかです。FBDとLDは混在して扱うことができます。**SFC**言語と**FC**言語は関数として扱うことができません。なお、扱う言語はファンクションやファンクションブロックを作成時に選択しますが、途中で変更することはできません。



ファンクション、ファンクションブロックのコンパイル

IEC言語で書かれたファンクション、ファンクションブロックのライブラリはISaGRAF プロジェクトの中で使われる前にコンパイル(ペリファイ)されていなければなりません。登録された後は、自動的にLD/FBDエディタのファンクションを選択するボックス内に含まれるようになります。



ファンクションからライブラリ内の別のファンクションを呼び出すことはできますが、自分自身を呼び出すような**リカーシブ性はサポートされていません**。

また、IEC言語で記述されたファンクションブロックは別のファンクションブロック(CあるいはIEC言語で記述)を読み出すことができません。



ソースコードの入力

IEC言語のファンクションやファンクションブロックのソースコード、即ち、プログラムは言語にあわせてエディタで入力します。コード生成は直接エディタから呼び出してライブラリファンクションやファンクションブロックのコンパイルを行なうことができます。各エディタの使用法の詳細は、本マニュアルの該当する章をご覧ください。



ローカル変数の辞書

ライブラリファンクション、ファンクションライブラリはローカル変数、ローカル定義ワードを持つことができます。変数の宣言のためにはエディタウィンドウから「**ファイル**」-「**辞書**」メニューを使う必要があります。



ライブラリファンクション、ファンクションブロックはグローバル変数やグローバルファンクションブロックインスタンスにはアクセスできません。また、ファンクション内で使われるローカル変数はファンクションのボディーで初期化される必要があります。

IEC言語で記述されたファンクションブロックのローカル変数は、プロジェクト内でブロックが使われる毎に、インスタンスとしてコピーされます。よって、ブロックが次に読み出されるまでコピーされた値を保持します。



インタフェースの定義

新しいライブラリファンクション、ファンクションブロックが作成されると、呼び出しインタフェース、即ち、入出力パラメータを定義する必要があります。入力・出力パラメータの名前とタイプを定義するためにパラメータ定義ボックスが使われます。「**編集**」メニューの「**パラメータ**」コマンドにより、合計最大32の入力パラメータと出力パラメータを定義します。ファンクションの出力パラメータは1つのみで、かつファンクション名と同じでなければなりません。



パラメータダイアログボックスの上部のウィンドウには関数のパラメータ(入力、出力パラメータ)が示されています。パラメータの順番は、入力パラメーター>出力パラメータの順になります。

下部のウィンドウには現在選択されているパラメータに関する詳細情報を示しています。

- パラメータ名
- パラメータの方向(入力、出力)
- パラメータタイプ

次のいかなるタイプもパラメータとして扱うことができます。

- ブール型
- 整数型
- 実数型
- タイマ型
- 可変長文字列型

出力パラメータはパラメータリストの最後である必要があります。パラメータ名には以下のルールがあります。

- パラメータ名最大長は半角16文字
- 最初の文字は英文字(a-z)
- 以降の文字は英文字、数字、_ のいずれか
- 大文字、小文字の区別なし

一つのファンクションやファンクションブロックで同一のパラメータ名は許されません。同一のパラメータ名は異なるファンクションでは扱うことができます。

“挿入”ボタンは新しいパラメータを選択中のパラメータの前に挿入します。“削除”ボタンは選択中のパラメータを削除します。“アレレンジ”ボタンは自動でパラメータの順番を並び替えて出力パラメータが最後にくるようにします。“OK”ボタンはパラメータの定義を保管してダイアログボックスを終了します。“キャンセル”ボタンはパラメータ定義の変更を保管せずにダイアログボックスを閉じます。

A.22.6 C言語ファンクションとファンクションブロック

IEC言語によるファンクションインタフェースの定義と同様に、C言語で書かれたファンクション、ファンクションブロックはアプリケーションのFBDやST言語から呼び出されます。

ファンクションは同期プロセスなので、ISaGRAF のアプリケーションからこれらの関数が呼び出されると、処理が終わるまでアプリケーションは待機状態となります。処理が終了すれば再び ISaGRAF のアプリケーションが実行されます。ファンクションブロックは、プログラムコードとスタティクなデータを併せ持っています。例えば、カウンタの機能を持つファンクションブロックでは、カウントアップする処理の部分と、カウントの結果の両方を持ちます。C 言語ファンクションとファンクションブロックは、ISaGRAF プログラム中の処理も、システムリソースとのアクセスも、両方の機能を実現します。



新しいライブラリファンクション、ファンクションブロックが作成されると、呼び出しインタフェース、即ち、入出力パラメータを定義する必要があります。入力・出力パラ

ライブラリ管理ユーティリティの使い方

メータの名前とタイプを定義するためにパラメータ定義ボックスが使われます。「編集」-「パラメータ」コマンドにより、パラメータ定義を行います。ファンクションの場合は、入力パラメータは31個までで出力パラメータは1つのみです。ファンクションブロックでは合計最大32の入力パラメータと出力パラメータを定義できます。パラメータ定義はIEC言語によるファンクション、ファンクションブロックと同様に行いますが、以下にC言語による場合の相違部分を記述します。

C言語によるファンクションやファンクションブロックの場合、パラメータのタイプはISaGRAFでのタイプとC言語でのタイプが下記のように対応します。

ブール型	Unsigned long	unsigned 32 bit word: 1=true / 0=false
整数型	Long	signed integer 32 bit word
実数型	Float	単精度浮動小数値
タイマ型	Unsigned long	unsigned integer 32 bit word(単位は1ms)
可変長文字列型	char *	character string.

文字列型の場合、Nullキャラクタを含めることはできません。CコードではNullキャラクタはメッセージ(文字列)の終了を意味します。

C言語ファンクション、ファンクションブロックの詳細とターゲットへの実装方法については、ターゲットユーザガイドの「C言語プログラミング」を参照願います。

A.22.7 C言語数値変換関数

数値変換関数は、アナログ型の入出力変数に付加され、ISaGRAFのI/Oドライバから呼び出されます。

数値変換関数を入出力変数に割り当てるためには、まず変数辞書の定義を行います。

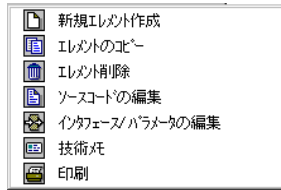
変換関数では**電氣的な値(外部)**と**アプリケーション値(内部)**との関係を定義し、値を変換します。ISaGRAFでは、ライブラリ上で、このメカニズムをC言語で定義することが出来ます。変換関数は、アナログ入力変換及びアナログ出力変換の2つに分けることができます。

変換関数では**整数アナログ**、**実数アナログ値**のどちらも扱えます。内部的には関数は浮動小数点処理をしています。どの変換関数でも関数のインタフェース部分は同じです。変換関数のC言語によるインタフェース部分は **GRCN0DEF.H** ヘッダーファイル内に記述されています。

マニュアルの「C言語プログラミング」にC言語関数のISaGRAF上での管理、組み込みに関して解説されています。

A.22.8 ツールバーアイコン

以下に、ツールバーアイコンと各アイコンボタンに割り付けられているコマンドを示します。



A.23 アーカイブユーティリティの使い方

ISaGRAF のアーカイブユーティリティを使うと、ISaGRAF のプロジェクトやライブラリをディスクにバックアップすることができます。アーカイブユーティリティは、ISaGRAF のプロジェクトマネージャや、ライブラリマネージャから呼び出されるダイアログボックスです。



信頼性のあるアーカイブを作成するために以下のガイドラインを守ることをお勧めします。

- ディスクのラベルに、保管されている内容を書きこんでおくこと。
- プロジェクトとライブラリのディスクは別々にすること。
- 異なるプロジェクトのディスクは別々にすること。

A.23.1 アーカイブユーティリティの呼び出し

アーカイブユーティリティのダイアログボックスはプロジェクトマネージャ・ウィンドウの「ツール」-「アーカイブ」メニューから起動します。プロジェクトデータ、共通データの保存、復元を行う事ができます。

アーカイブのダイアログボックスは、ISaGRAF ライブラリマネージャからも起動する事ができます。この場合は、ライブラリ管理エディタでの各種エレメントの保存・復元が可能です。

■ プロジェクトのアーカイブ

「ファイル」-「プロジェクト」によりプロジェクトのバックアップや復元が行えます。プロジェクトの全コンポーネント(プログラムのソース、オブジェクトコード、アプリケーションコード)が一つのファイル(*.PIA)ファイルにまとめられて保存されます。「オプション」-「圧縮」コマンドをセットすることでアーカイブファイルのサイズを小さくすることができます。

■ ライブラリエLEMENTのアーカイブ

ISaGRAF ライブラリのELEMENTはそれぞれ個別にバックアップすることができます。一つのライブラリエLEMENTを構成している全モジュール(技術メモ、定義、インタフェース、ソースコードなど)は、まとめて1つのアーカイブファイルになります。

■ 共通データのアーカイブ

「ツール」-「アーカイブ」-「共通データ」コマンドにより、ISaGRAF で使われる共通データをバックアップ、復元することが可能です。このコマンドは ISaGRAF のライブラリには影響を与えません。

以下にこのコマンドでコピーされるファイルを示します。

common.eqv 共通定義ワード

oem.bat..... ユーザ定義のMS-DOSコマンドファイル

これらのファイルは元のままでアーカイブファイルに保管されます。 また、アーカイブファイルは圧縮されません。

A.23.2 オプション

ISaGRAF アーカイブユーティリティが使用するディレクトリは、ダイアログボックスの下部に表示されます。「参照」ボタンを押してディレクトリの一覧を表示させ、他のディレクトリやドライブを選択する事ができます。



「オプション」-「圧縮モード」がセットされている場合は、バックアップの際に生成するファイルを圧縮して保存します。アーカイブファイルのサイズを小さくするには効果があります。

アーカイブファイルを復元する際にはアーカイブマネージャが圧縮モードを自動判別して復元しますので、**圧縮モード**がセットされているかはファイル復元処理には関係ありません。



A.23.3 バックアップと復元

「ワークベンチ」リスト(ダイアログの左側のリスト)は、ISaGRAF ワークベンチにあるオブジェクトを示しています。「アーカイブ」リスト(右側のリスト)は、指定のアーカイブディレクトリに存在するオブジェクトを示しています。

□ バックアップ

オブジェクトをアーカイブに保存するには、**左側**のリストボックス(ISaGRAF ワークベンチ内のファイル)からオブジェクトを選択して「**バックアップ**」ボタンを選択します。同時に複数のオブジェクトを選択することも可能です。**右側のリストボックス**(アーカイブの内容)のオブジェクトが選択されているときは「**バックアップ**」ボタンは選択できません。

□ 復元

オブジェクトをアーカイブからワークベンチに復元する場合には**右側**のリストボックス(アーカイブオブジェクト)からオブジェクトを選択して「**復元**」ボタンを選択します。同時に複数のオブジェクトを選択することも可能です。**左側のリストボックス**(ISaGRAF ワークベンチ)のオブジェクトが選択されているときは「**復元**」ボタンは選択できません。

A.23.4 アーカイブファイルの拡張子

アーカイブユーティリティはオブジェクト単位でアーカイブファイルを作ります。ファイルの名前はオブジェクトと同じですが、オブジェクトの種別によって下記のようなファイルの拡張子が付けられます。

.pia.....プロジェクト

.bia.....I/O ボード
.iaa.....IEC 言語ファンクション
.aia.....IEC 言語ファンクションブロック
.uia.....C 言語ファンクション
.fia.....C 言語ファンクションブロック
.cia.....C 言語変換関数
.ria.....I/O 構成
.xia.....I/O 装置機器

A.24 プロジェクトドキュメントの印刷

ISaGRAF のドキュメント印刷機能を使って、プロジェクト全体のドキュメントの印刷ができます。この機能を使うには、プロジェクト管理ウィンドウまたはプログラムのウィンドウの「プロジェクト」-「印刷」コマンドを実行します。プロジェクトの一部分のドキュメント印刷は、各エディタの「印刷」コマンドからでも行えます。いずれの場合もドキュメント印刷は同等の機能を持っています。

「編集」メニューのコマンドは印刷する項目の選択に使います。これは印刷するドキュメントの目次を作成することに相当します。プロジェクトに関するいかなる情報（プログラム、変数、オプション、I/O 接続...）もドキュメントにすることが出来ます。ただし、別のプロジェクトの情報や ISaGRAF ライブラリの情報は含みません。



「ファイル」-「印刷」コマンドで、指定した項目の内容をまとめてドキュメント印刷します。ドキュメントのフォーマットを生成するのに多少の時間がかかることがあります。印刷が完了するまでは、ワークベンチのほかの操作をしないことをお勧めします。ドキュメントの印刷にはハードディスクに充分な空き領域が必要です。もし、ハードディスク容量不足でエラーメッセージが出た場合は、不要なファイルを削除するか、ドキュメント内容を絞って印刷内容を減らす必要があります。「印刷」コマンドの実行時に、ドキュメント印刷に関するメモを記入することが出来ます。これは、印刷の履歴になります。つまり、ここで入力されたメモは次の印刷の時に次の印刷メモとあわせて履歴となって1ページ目に印刷されます。

A.24.1 ドキュメントの目次のカスタマイズ

「編集」メニューでドキュメントの「目次」を作成します。デフォルト（全コンポーネントを印刷します）または任意の内容（一部のコンポーネントのみ）を選択することができますし、目次内での順序を変更することもできます。



デフォルトリスト

「編集」-「デフォルトリスト」コマンドにより、ドキュメント目次の内容をデフォルトに戻します。デフォルト設定では全ての項目を含んでいます。以下に項目を示します。

- プロジェクト記述
- プログラムの階層（プログラム間のリンク関係）
- ソースコード（全プログラム）
- 修正履歴ファイル（全プログラム）
- 共通定義
- グローバル 定義
- ローカル 定義（全プログラム）
- グローバル 変数
- ローカル 変数（全プログラム）
- アプリケーションのオプション

- I/O 接続
- 変数のリスト
- 数値変換テーブル
- クロスリファレンスの圧縮
- クロスリファレンスの詳細
- 宣言のサマリー
- ネットワークアドレスのマップ
- 修正履歴(プロジェクト)

目次は「**ファイル**」-「**保存**」コマンドでディスクに保存できます。このコマンドは他のエディタウィンドウから印刷コマンドが実行されている時には、使用できません。



「切り取り」、「貼り付け」

「**編集**」-「**切り取り**」、「**貼り付け**」コマンドで項目リスト中のアイテムを移動させて、目次の内容をカスタマイズすることができます。また、一度に複数の項目の選択して切り取ることも可能です。



項目リストの削除

「**編集**」-「**削除**」コマンドにより、項目リストのアイテムを全て削除できます。これにより、項目を追加することで新しく目次を作成することになります。



目次項目の追加

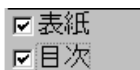
「**編集**」-「**挿入**」コマンドにより、「**アイテム追加**」のダイアログボックスが開きます。ここではプロジェクトの印刷項目(プロジェクトのコンポーネント)をの追加できます。

項目がプログラムに関する場合は、プログラム毎の選択になります。「**プログラム**」選択ボックスでプログラムを指定して、「**追加**」ボタンを選択して下さい。目次には同じコンポーネントは一度しか定義できません。

A.24.2 オプション

「**オプション**」メニューにより、ドキュメントの書式をカスタマイズすることが出来ます。

他のオプションは、ドキュメント生成ウィンドウのボタンで選択可能です。



「**表紙**」オプションは、プロジェクトのタイトルと印刷の履歴を記載した表紙をドキュメントの先頭に印刷するためのオプションです。このオプションが設定されていない場合は、ドキュメントの先頭アイテムが1ページ目に印刷されます。

「**目次**」オプションは、ドキュメントの最後に目次を印刷するためのオプションです。両方のオプションは、各エディタ(プログラム、辞書など)の「**印刷**」コマンドからドキュメント生成が起動したときにはデフォルトでは設定されていません。



SFCチャートのレベル分け

「オプション」-「SFCレベル分け」コマンドがセットされているときは、SFCプログラムの印刷の時にレベル1(チャートとコメント)が全て印刷されてからレベル2(プログラミングステートメント)が印刷されます。このオプションがセットされていないときはレベル1、2が一緒に印刷されます。



ページフォーマット


「オプション」-「ページフォーマット」コマンドにより、印刷ページのフォーマット用のパラメータを設定します。以下のパラメータを指定します。

- 左マージン: 1cm、2cm、なし から選択可能
- ページ枠: 印刷ページの印刷枠の有無の指定



ページタイトルのフォーマット

「オプション」-「ページタイトル」コマンドにより、各ページの下部分に印刷されるタイトルボックスの指定が行えます。タイトルボックス標準レイアウトを以下に示します。

		プロジェクト "XXXX"	(日付)
			(A°-J°)

メインタイトルの一行目 (ISaGRAF プロジェクトの名称)と日付、ページ番号に関してはドキュメントマネージャが自動で生成しますので変更は出来ません。

メインタイトルの2行目とメモ1～メモ3はユーザが設定できます。さらに、タイトルボックスの左のボックスのロゴ(シンボル)も変更が出来ます。このロゴを変更する場合は変更したいイメージファイル(**BMPファイル**)の場所をパス名付きで設定して下さい。イメージの色数に制限はありません。ビットマップイメージは決められたサイズに収まるように自動的に変形されます。ファイルの指定後、ロゴをクリックすれば新しいイメージが表示されます。印刷時に指定されたビットマップイメージが見つからない場合はブランクとなって印刷されます。



フォントの選択

「オプション」-「テキストフォント」、「タイトルフォント」コマンドにより、印刷されるドキュメントの本文(タイトルページやタイトルボックス部分以外)のフォントの種類やサイズ、スタイルを指定ができます。

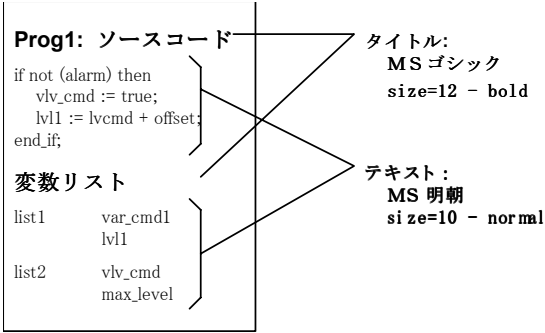
フォントの指定はWindows標準のフォント指定ダイアログボックスで行います。全てのテキスト(STやIL言語のプログラム、ダイアグラム中の名前など)が「テキストフォント」の設定にしたがって印刷されます。タイトルだけ、「タイトルフォント」で印刷されます。

もし、フォントが設定されていない場合は標準フォントが選択され、更にフォントスタイルとして、

- テキスト: 通常の字体
- タイトル: 太字

となります。

参考: タイトルとテキストの区別



A.25 パスワードプロテクション

ISaGRAF ワークベンチにはデータプロテクションシステムがあります。即ち、プロジェクトやライブラリのエレメントをパスワードで保護することができます。ライブラリのエレメントとはI/O構成、I/Oボード、I/O装置機器、IEC言語で書かれたファンクションやファンクションブロック、C言語ファンクション、C言語ファンクションブロック、変換関数です。パスワードプロテクションはプロジェクト単位、ライブラリエレメント単位で行います。複数のものに対してまとめてパスワードを設定することはできません。

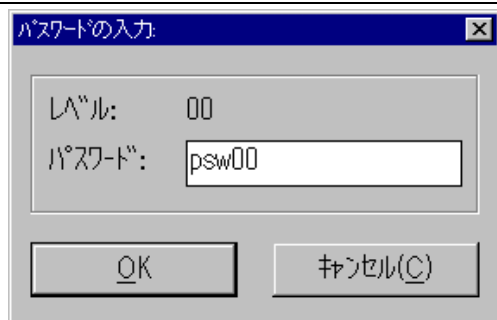
■ プロテクションレベル

1つのプロジェクトやライブラリエレメントに対して最大**16**個のアクセスレベルを設けることができます。各レベルには対応したパスワードが存在します。パスワードの番号は**0**から**15**であり、**0**が最も高いレベルとなります。ユーザがあるパスワードを知っていれば、そのレベルに対して登録されている操作はもちろん、それよりレベルの低いパスワードに登録されている操作も行うことができます。基本的なコマンドやプロジェクトのデータ、ライブラリのエレメントは1個のアクセスレベルにつきそれぞれ別々にプロテクトできます。

パスワードの使われ方の例としては新しくアプリケーションコード生成をする場合のメニュー「アプリケーションコード生成」コマンドに対してパスワードを設定することによりプロジェクトを保護することができます。他にもプログラム、アプリケーションのオプション、ライブラリのエレメントの技術メモなどを保護できます。

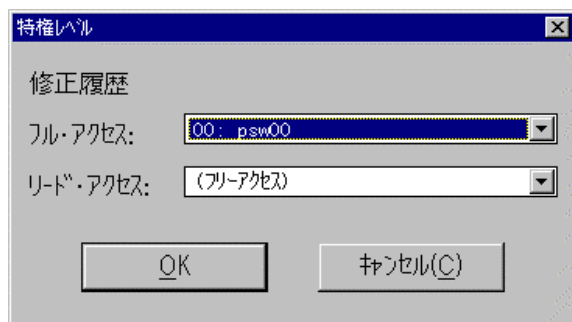
■ パスワードプロテクションの定義

ISaGRAF のメニューの「**パスワードセット**」コマンドで、プロジェクト単位、あるいはライブラリのエレメント単位でパスワードとそのアクセスレベルを定義できます。このコマンドは、ISaGRAF の「プロジェクト管理」ウィンドウ(プロジェクトの場合)、「ライブラリ管理」ウィンドウ(ライブラリエレメントの場合)のメニューから実行します。パスワードが未定義の時は、この「**パスワードセット**」コマンドはパスワードなしで使えます。いったんパスワードが設定されると、以降は本人が知っている最も高いレベルのパスワードを入力してからパスワード設定を行うこととなります。この場合、より高いパスワードおよびプロテクトされている項目は変更できません。「**パスワード設定**」コマンドによって、異なるアクセスレベルの設定や、定義されたレベルでのコマンドやデータのプロテクトを行うことができます。パスワードはダイアログボックス上部のテキストフィールドをダブルクリックして入力します。下図はその時の表示です。



ダイアログボックス下部には、プロテクト可能なアイテム(データや機能)が、現在のプロテクトレベルが表示されます。プロテクトレベルには、“リードアクセス”あるいは“フルアクセス”の特権も付加されています。リードの特権を設定すると、そのパスワードを知らない人が対応するデータを見ることも、印刷することもできないようになります。

ダイアログボックス下部の項目フィールドをダブルクリックすると、特権を設定するためのダイアログボックスが開きます。



これらの、特権は“フリーアクセス”か、定義済みのプロテクトレベルに設定可能です。フルアクセスの特権には、リードアクセスよりも低いレベルのプロテクトレベルは設定できません。

プロジェクト記述等のドキュメントの表示については、元々オープンなもののので、パスワードによるプロテクトは設定できません。

プロテクトされたデータへのアクセス

パスワードが設定されている場合でもワークベンチの起動時にはパスワードは不要です。ユーザがプロテクトされているデータや機能にアクセスする場合にのみダイアログボックスが表示され、パスワードの入力を要求してきます。

ユーザが要求されたパスワード(あるいはそれ以上のレベルに登録されているパスワード)を入力すれば、操作を継続することができます。一度、ユーザが入力したパスワードは記憶され、引き続きの操作でパスワードが必要な場合でも再度入力する必要がありません。ただし、ISaGRAF ワークベンチの全てのウィンドウが閉じてしまった場合は、再度パスワードを入力する必要があります。プロジェクト

管理、ライブラリ管理、およびアーカイブユーティリティで入力したパスワードは共有されません。

不正なパスワードが入力された場合は、プロテクトされている機能は使えません。

■ アーカイブユーティリティとのリンク

オブジェクト(プロジェクトやライブラリエlement)をアーカイブディスクへバックアップする場合は、プロテクション機能の“**アーカイブにバックアップ**”が有効となります。これはワークベンチ(パソコンのディスク)のオブジェクトのデータプロテクションシステムに相当するものです。データプロテクションシステムは、アーカイブディスク上に既にオブジェクトが存在するかどうかは確認しません。アーカイブユーティリティの「**バックアップ**」コマンドはオブジェクトのプロテクション情報も一緒に保存します。

オブジェクトを復元する時にすでにワークベンチに同一のオブジェクトが存在する場合は、プロテクション機能の“**アーカイブして上書き**”が有効となります。これもワークベンチのオブジェクトのデータプロテクションシステムに相当するものです。データプロテクションシステムは、アーカイブディスク上に既にオブジェクトが存在するかどうかは確認しません。

もしそのコマンドが有効なものなら、データのプロテクション情報はパソコンのディスク上のものに上書きされます。

■ 変数とI/O チャンネルに対するプロテクト設定

ISaGRAF ワークベンチは、階層化したパスワードを使用したデータのプロテクト機能を提供します。変数宣言やI/O 接続に対しては、データ全てをまとめて1個のパスワードによってプロテクトすることが可能ですが、更には、全ての変数やI/O 接続に対しそれぞれプロテクトを設定することも可能です。そのためには下記の条件が必要です；

- パスワードが「**パスワードセット**」コマンドによって設定済みであること。これで個別のプロテクションのレベルを用意しておきます。
- 変数辞書及びI/O 接続に対する一括のプロテクションレベルよりも高いプロテクトレベルを使うってプロジェクトを開くこと。

変数及びI/O 接続それぞれにプロテクションを設定すると、辞書エディタやI/O 接続エディタの変数名の左側に小さなアイコンが表示されます。

選択した変数やI/O チャンネルにプロテクトを設定するには、「**編集**」メニューの「**プロテクションの設定**」や「**プロテクションの削除**」コマンドを使用します。どちらのコマンドも付加するパスワードを入力するように要求しますので、その度に必要なレベル以上のパスワードを入力することになります。

注意: もし変数やI/O チャンネルが1レベルのパスワードだけでプロテクトされていた状態で、その関連するパスワードが削除されてしまった場合、なおかつそれより高いパスワードが設定されていない場合は、変数やI/O チャンネルの変更ができなくなります。これはより高いレベルのパスワードを設定されるまで続きます。

A.26 アドバンスト プログラミングテクニック

本章では ISaGRAF ワークベンチとターゲットシステムについてのより詳しい情報について解説します。本章を読まれる前には ISaGRAF の基本的な知識を備えておく必要があります。

A.26.1 ISaGRAF ツールの詳細情報

ISaGRAF の編集ツールを使っている際には、**マウスの右ボタン** をクリックすることによりポップアップメニューを開くことができますこのメニューで編集用の主なコマンドを使えます。メニューはカーソルの現在位置で開かれるためマウスの動きを最小限にとどめることができるため有効です。

ISaGRAF のツールは**複数起動**が可能です。ワークベンチのエディタで、同時にウィンドウを開くことで複数のソースファイルを並行して編集することができます。ただし、同一のソースファイルを2つ以上は開くことはできません。

ツールバーのグラフィックボタンの情報を見るための他のコマンドがあります。ツールバーの空きエリアをマウスでダブルクリックすると、ツールバーの内容をポップアップメニューとして表示します。また、マウスカーソルがツールバーのコマンドアイコン上にあるときはコマンドの説明を表示します。

A.26.2 ロックされたI/OとバーチャルI/O

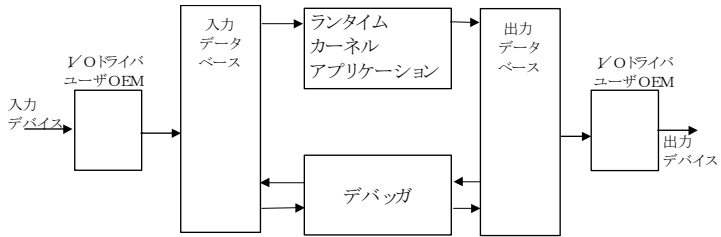
I/O接続エディタで、I/Oボードを**バーチャル**ボードに設定することで、物理I/Oチャンネルの処理を行わなくなります。I/Oボードがバーチャルと定義されていても ISaGRAF のターゲットでの処理には変化はありません。唯一の違いは入力デバイスからの読み込みがないこと、出力デバイスへの書き込みがないことです。このモードでは、ISaGRAF デバッガーを使って入力変数の値を変更できます。**バーチャル**の設定はボード単位で行うことになります(個々のI/Oチャンネルに対してはできません)。バーチャルボードの設定はアプリケーションコード生成を行う前に、I/O接続エディタで定義する必要があります。**バーチャル**という属性は**静的**にアプリケーションコードに含まれますので、アプリケーションがスタートするときにバーチャルとなります。

これとは別に、I/O変数を**ロック**することで物理I/Oと分離することも可能です。I/O変数のロック・アンロックは ISaGRAF のデバッガから行います。変数のロックは**動的**な操作であり、アプリケーションコードには保存されません。変数のロック操作は、一度に一つの変数(I/Oチャンネル)に対して行います。

以下に主なI/Oコントロール機能についてまとめます。

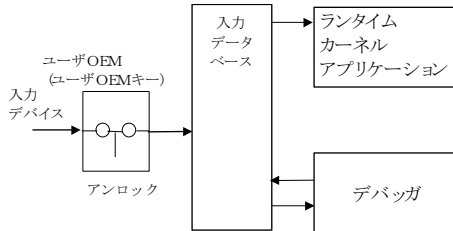
	バーチャル属性	I/O変数ロック
設定ツール	I/O接続エディタ	デバッガ
定義	静的	動的
選択モード	ボード単位	変数単位
適用例	検証、テスト	メンテナンス

以下の図に ISaGRAF のターゲットでのI/Oデータフローを示します。



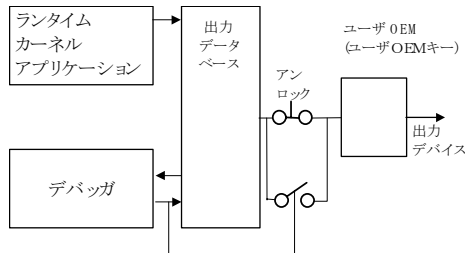
入力変数のロック

入力変数がロックされているときは、入力データベースへの様々なアクセスに変更はありませんが、データベースと入力デバイスは切り離されています。デバッガから入力データベースへのデータ書き込みは可能で、ISaGRAF ランタイムカーネルによって通常通り処理されます。



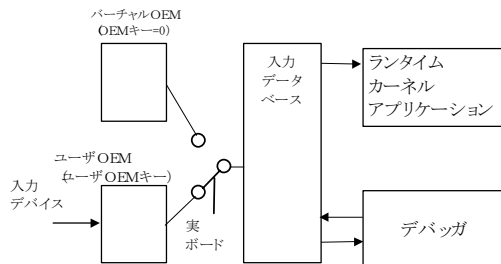
出力変数のロック

出力変数がロックされているとき、ランタイムカーネルと出力データベース間は切り離されています。この場合でも、デバッガにより出力ドライバを通して出力デバイスへのアクセスは可能です。



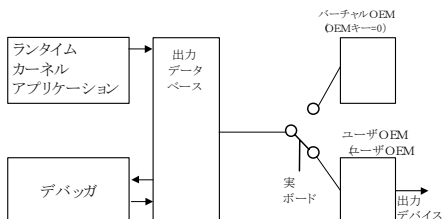
バーチャル入力ボード

入力ボードがバーチャル設定のときは、入力デバイスと入力データベースとの間が切り離されていることになります。バーチャルI/Oドライバが実I/Oドライバの代わりになります。

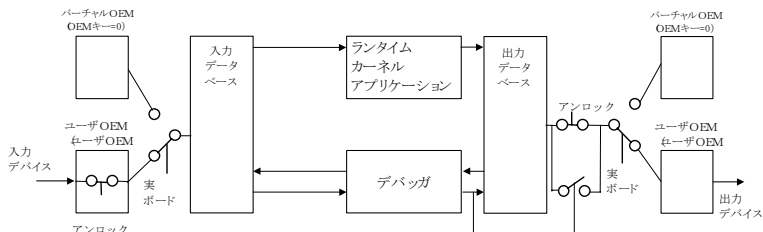


バーチャル出力ボード

出力ボードのバーチャル設定は、入力ボードのバーチャル設定と同様に、出力デバイスと出力データベースとの間が切り離されていることになります。カーネルは出力データベースを更新します。バーチャルI/Oドライバが実I/Oドライバの代わりになります。



全部の組み合わせを下図に示します。



A.26.3 通信リンク設定(ワークベンチターゲット)の動作確認

デバッガーウィンドウのステータスメッセージが“**遮断状態**”になっている場合は、ワークベンチとターゲット間の通信エラーによることがほとんどです。何か具体的なプログラムのターゲット上での動作確認をする前に、必ずターゲット上で**実行中のアプリケーションがない状態**でワークベンチとターゲット間の通信が正常であることを確認しておく必要があります。この手法なら、アプリケーションの動作による影響を切り離して通信リンクの動作を確認できます。

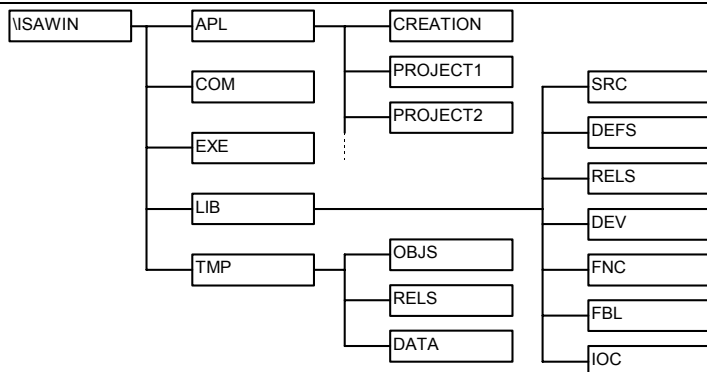
変換関数やC言語ファンクションなど、ユーザプログラムでC言語を使う場合は、ターゲットシステムに直接アクセスすることが可能になっています。このようなコンポーネントのプログラムに問題があると、システムエラーを起こしたり、ISaGRAFのシステムの動作に異常が発生すること考えられます。同じことが ISaGRAFのI/O開発ツールキットでI/Oドライバを開発する場合にもいえます。例えばボードの設定アドレスを間違えるなど、です。

下の表にデバッガーのメッセージから判断できる現象の原因例を簡単に示します。

ステータス	起動モード	原因(例)
“ 遮断状態 ” (ダウンロード前)		<ul style="list-style-type: none"> — ターゲットが立ち上がっていない — 通信ケーブルがない、正常でない — 通信のパラメータが不正 — ターゲットが正しくインストールされていない
“ 遮断状態 ” (ダウンロード後)	サイクル モード	<ul style="list-style-type: none"> — I/Oの構成が正しくない — ターゲットシステムが正常でない
	リアルタイム モード	<ul style="list-style-type: none"> — I/Oの構成が正しくない — ターゲットシステムが正常でない (Cプログラムエラー)
“ アプリケーション がありません ”		<ul style="list-style-type: none"> — アプリケーションがダウンロードされていない — アプリケーションがスタートしていない (Cプログラムエラー) — Intel/Motorola のTICコードが不一致 — ターゲットアプリケーションのバージョンが不一致

A.26.4 ISaGRAF のディレクトリ構成

ISaGRAF ワークベンチは特定のディスクディレクトリ構成のもとで動作します。ルートディレクトリ名はワークベンチのインストール時に指定することができます。デフォルトでは **ISAWIN** ディレクトリです。以下にインストールプログラムが作る標準のディスクディレクトリ構成を示します。



以下に ISaGRAF のサブディレクトリについて解説をします。

ディレクトリ	内容
APL	ISaGRAF プロジェクトのためのルートディレクトリ。 各プロジェクトごとに対応した一つのディレクトリを持ち、プロジェクトに必要なデータを全て含んでいます。 他のプロジェクトグループを作成し、別のディレクトリにプロジェクトディレクトリを作成することも可能です。インストール時には、サンプルのアプリケーションが入った "SMP" というディレクトリを作成します。
COM	共通データ。 プロジェクト間で共通に扱えるデータ。
EXE	ISaGRAF のプログラムとヘルプファイル。
LIB	ISaGRAF ライブラリデータ。 - ライブラリの要素 - 各要素に対するパラメータとインタフェース - 技術メモ
LIB\IOC	I/O構成のソースコード
LIB\FNC	IEC言語で記述されたファンクションのソースコード
LIB\FBL	IEC言語で記述されたファンクションブロックのソースコード
LIB\SRC	変換関数、C言語ファンクションのソースコード
LIB\DEFS	変換関数、C言語ファンクションのヘッダーファイル
LIB\RELS	変換関数、C言語ファンクションのオブジェクトコード
LIB\DEV	C言語のライブラリを作るためのコマンドファイル (メイクファイル、リンクリストなど)
TMP	テンポラリファイル。ISaGRAF のコード生成用として予約されています。削除しないようにしてください。

これらのサブディレクトリは別のディスク(あるいはサブディレクトリ)へ移動することができます。もし、標準以外のディスクディレクトリ構成を行う場合は、サブディレクトリのパス名はEXEサブディレクトリ内の **ISA.ini** ファイルの **WS001** セクションに記述しなければなりません。

以下のテーブルに **WS001** セクションのエントリーを示します。

Isa	ISaGRAF ワークベンチのルートディレクトリ名
IsaExe	ISaGRAF プログラムとヘルプファイル用のルートディレクトリ名
IsaApl	ISaGRAF プロジェクト用のルートディレクトリ名
IsaTmp	テンポラリファイル用のディレクトリ名
IsaSrc	ライブラリのソースコード用のディレクトリ名
IsaDefs	ライブラリのソースヘッダー用のディレクトリ名

注意: IsaTmp ディレクトリを別のディレクトリに変更する場合は、必ず、その中にサブディレクトリ OBJ、RELS、DATA を作る必要があります。以下に **WS001** セクションを変更したカスタムディレクトリ構成の場合の設定例を示します。

```
;file c:\ISAWIN\EXE\ISA.ini

[WS001]
Isa=c:\isawin
IsaExe=c:\isawin\exe
IsaApl=c:\isawin\apl
IsaTmp=c:\isawin\tmp
IsaSrc=c:\isawin\lib\src
IsaDefs=c:\isawin\lib\defs
```

ターゲットにC言語ファンクションやファンクションブロックを追加する場合は、**ISAWIN\LIB\DEV** ディレクトリにコマンドファイル、メイクファイル、マップファイルといった開発用のファイルを格納します。また、**ISAWIN\LIB\RELS** ディレクトリはCコンパイラで作ったオブジェクトやリンクに必要なCライブラリを保存するために使います。

A.26.5 アプリケーションシンボル

ISaGRAF アプリケーションの各オブジェクトは、名前(変数名)や**バーチャルアドレス**(コード生成時に計算されます)によって参照されます。バーチャルアドレスは、変数宣言の時に入力する**ネットワークアドレス**とは異なります。バーチャルアドレスは通信およびI/Oドライバ開発キットで作ったC言語アプリケーションからアクセスするとき等に使います。アプリケーションコード生成時に、プロジェクトを構成しているオブジェクト(変数、プログラム、ステップ、...)の名前とバーチャルアドレスとの対応を記述したテキストファイルが作られます。このファイルにはユーザアプリケーションが ISaGRAF のスタティックデータにアクセスするための情報が含まれています。このファイルの名前は **"APPLI.TST"** で **"ISAWIN\APL\prname"**(prname はプロジェクト名)ディレクトリに作られます。以下に**"APPLI.TST"**の記述フォーマットについて示します。

主な用語として以下の3つがあります。

VA: バーチャルアドレス
ATTR: 変数属性
USP: C言語ファンクション

変数属性には以下のものがあります。これらは**"変数属性"**フィールドに書かれます。

+X..... 内部変数
+C..... リードオンリーの内部変数(定数)

+I..... 入力変数
+O..... 出力変数

バーチャルアドレス以外の全ての番号は10進数で表現されます。バーチャルアドレス(VA)は "I"が先頭で16進の4桁で表現されます。

例えば、

123..... 10進数
!A003..... 16進数のバーチャルアドレス

"APPLI.TST"のファイル構造を以下に示します。ファイルは複数の**ブロック**から成り立っています。ブロックは複数の**レコード**から成り立っています。各レコードは1行のテキストで記述されます。各ブロックは1行のヘッダーから開始されます。

Start block
Description blocks
End block

1つのブロックの一般的な文法を以下に示します。

@ <ブロック名> <引数>
#レコード...
#レコード...
...

1番目のブロックにはアプリケーションの基本情報が含まれています。以下に例を示します。

```
@ISA_SYMBOLS, <appli_crc>
#NAME, <appli_name>, <version>
#DATE, <creation_date>
#SIZE, G=<nbprg>, S=<nbstep>, T=<nbtra>, L=0, P=<nbpro>, V=<nbvar>
#COMMENT, ICS Triplex ISaGRAF Inc.
```

ここで、

appli_crc..... アプリケーションシンボルのチェックサム
appli_name..... アプリケーション名
version..... ISaGRAF ワークベンチのバージョン
creation_date アプリケーション作成日時
nbprg プログラム数
nbstep..... SFCステップ数
nbtra SFCトランジション数
nbpro C言語ファンクションの数
nbvar..... 全変数の数

最後のブロックにはファイルの最後を表わすためのものです。以下のようになります。

@END_SYMBOLS

アプリケーションプログラムの記述のためのブロックは以下のような例になります。

```
@PROGRAMS, <nbprg>
#<va>, <name>
#...
```

nbprg このブロックに定義されているプログラム数
va プログラムのバーチャルアドレス
name プログラム名

SFCのステップを記述するブロックを以下に示します。SFC以外のプログラムには仮想的なステップが定義されます。

```
@STEPS, <nbsteps>
#<va>, <name>, <father>
#...
```

nbsteps このブロックに含まれるステップ数
va ステップのバーチャルアドレス
name ステップ名
father 親プログラムのバーチャルアドレス

SFCのトランジションを記述するブロックを以下に示します。

```
@TRANSITIONS, <nbtrans>
#<va>, <name>, <father>
#...
```

nbtrans このブロックに含まれるトランジション数
va トランジションのバーチャルアドレス
name トランジション名
father 親プログラムのバーチャルアドレス

ブール型変数を記述するブロックを以下に示します。

```
@BOOLEANS, <nb_boo>
#<va>, <name>, <attr>, <program>, <eq_false>, <eq_true>
#...
```

変数の数が 4095 を超える場合は、

```
X#(1.<varno>), <name>, <attr>, <program>, <eq_false>, <eq_true>
```

nb_boo このブロックに含まれる変数の数
va 変数のバーチャルアドレス
varno ブール型変数内でのアドレス
name 変数名

attr 変数属性
program 親プログラムのバーチャルアドレス。グローバル変数のときは
"I0000"
eq_false FALSE の時の文字列
eq_true TRUE の時の文字列

整数・実数型変数を記述するブロックを以下に示します。

```
@ANALOGS, <nb_ana>  
#<va>, <name>, <attr>, <program>, <format>, <unit>  
#...
```

変数の数が 4095 を超える場合は、

```
X#(2.<varno>), <name>, <attr>, <program>, <format>, <unit>
```

nb_ana このブロックに含まれる変数の数
va 変数のバーチャルアドレス
varno 整数・実数型変数内でのアドレス
name 変数名
attr 変数属性
program 親プログラムのバーチャルアドレス。グローバル変数の時は
"I0000"
format = "I" (整数型変数)
..... = "F" (実数型変数)
unit 単位を表す文字

タイマ型変数を記述するブロックを以下に示します。

```
@TIMERS, <nb_tmr>  
#<va>, <name>, <attr>, <program>  
#...
```

変数の数が 4095 を超える場合は、

```
X#(3.<varno>), <name>, <attr>, <program>
```

nb_tmr このブロックに含まれる変数の数
va 変数のバーチャルアドレス
varno タイマ型変数内でのアドレス
name 変数名
attr 変数属性 (かならず +X: 内部変数になります)
program 親プログラムのバーチャルアドレス。グローバル変数の時は
"I0000"

可変長文字列型変数を記述するブロックを以下に示します。

```
@MESSAGES, <nb_msg>  
#<va>, <name>, <attr>, <program>, < max_len>
```

#...

変数の数が 4095 を超えた場合は、

X#(4.<varno>),<name>,<attr>,<program>,<max_len>

nb_msg..... このブロックに含まれる変数の数

va 変数のバーチャルアドレス

varno..... 可変長文字列型変数内でのアドレス

name 変数名

attr 変数属性

program..... 親プログラムのバーチャルアドレス。グローバル変数の時は
"!0000"

lg_max..... 文字列の最大長

C言語ファンクションを記述するブロックを以下に示します。

@USP,<nb_usp>

#<va>,<name>

#...

nb_usp..... このブロックに含まれるC言語ファンクションの数

va C言語ファンクションのバーチャルアドレス

name C言語ファンクション名

C言語ファンクションブロックのインスタンスを記述するブロックを以下に示します。

@FBINSTANCES,<nb_fb>

#<va>,<inst_name>,<fb_name>

#...

nb_fb..... このブロックに含まれるC言語ファンクションブロックのイン
タンスの数

va C言語ファンクションブロックのインスタンスのバーチャルアド
レス

inst_name..... C言語ファンクションブロックのインスタンス名前

fb_name..... C言語ファンクションブロックの名前(タイプ)

A.26.6 ISaGRAF ワークベンチI/O無制限版(WDL)の制限

ISaGRAF ワークベンチ(I/O無制限版)における制限を以下の表に示します。
もちろん、これ以外にはワークベンチに使用しているコンピュータの構成(空きメモ
リやディスク容量)や、ターゲットシステム(空きメモリや各種リソースなど)によ
る制限もあります。

プロジェクト:

オブジェクト	最大値	メモ
--------	-----	----

アドバンスト プログラミングテクニック

プログラム数	255	メイン、ファンクションやサブプログラム類、チャイルド SFC プログラムをまとめたもの
階層レベル数	20	

プロジェクト数は使用中のコンピュータのハードディスクの容量にのみ制限を受けます。

オブジェクトの名前:

オブジェクト名	最大値	メモ
プロジェクト	8 文字	
プログラム	8 文字	
変数	32 文字	コメントは半角換算で 60 文字
定義ワード	16 文字	
定義ワードの内容	255 文字	コメントは半角換算で 60 文字
変換テーブル	16 文字	
変数リスト	16 文字	
ファンクション、ファンクションブロック (ライブラリ)	8 文字	C言語ファンクション／ファンクションブロック、IEC言語ファンクション／ファンクションブロック
ファンクションパラメータ (ライブラリ)	16 文字	C言語ファンクション／ファンクションブロック、IEC言語ファンクション／ファンクションブロック
I/Oボード	8 文字	
I/O構成	8 文字	
OEMパラメータ	16 文字	
変換関数	8 文字	

編集(一つのプログラム内で):

オブジェクト	最大値	メモ
SFC行	600	
SFC列	20	
SFCステップ	4095	1つのプロジェクト全体で(イニシャルステップ、開始ステップ、終了ステップを含む)
SFCTランジション	4095	1つのプロジェクト全体で
LD/FBDエディタ	200 列 2000 行	編集領域のセル単位
Quick LD エディタ	—	パソコンの性能によります。
ILラベル	251	同一のILプログラム内
テキスト	—	パソコンの性能によります。

辞書(一つのプロジェクト内で):

オブジェクト	最大値	メモ
ブール型変数	65535	
整数・実数型変数	65535	整数型と実数型の合計

タイマ型変数	65535	
可変長文字列型変数	65535	
定義ワード	4095	同一範囲(ローカル、グローバル、共通)
定義ワード	255	1つのプログラム内
変換テーブル	127	1つのプロジェクト内
テーブル内のポイント数	32	1つの変換テーブル内

ブール型変数と整数・実数型変数の最大値は、各属性(内部、入力、出力、定数)の合計を示します。また、コンパイラが自動的に生成する内部的な変数やテンポラリ変数も含まれます。辞書エディタで一度に編集できる変数は、各変数タイプ、スコープ毎に 16000 個を超えられません。パソコンの空きメモリの量によっては 16000 未満になる場合もあります。変数の数が 4095 を超えるアプリケーションは、V3.21 以前のバージョンのターゲットでは実行できません。また、ネットワークアドレスを使用した標準の MODBUS 通信は、4095 を超える変数に対しては行えません。

I/O接続:

オブジェクト	最大値	メモ
I/Oボード	256	1つのプロジェクト内(ボードあるいは装置機器の合計)
I/Oチャネル	128	同一のボード

I/Oボードの数、装置機器のアイテムは 256 をを超えられません。

ライブラリ:

オブジェクト	最大値	メモ
IEC言語ファンクション	255	ライブラリに登録可能な最大数
IEC言語ファンクションブロック	255	ライブラリに登録可能な最大数
C言語ファンクション	255	ライブラリに登録可能な最大数
C言語ファンクションブロック	255	ライブラリに登録可能な最大数
C言語ファンクションブロックインスタンス	4095	1つのプロジェクト内で同一のタイプのファンクションブロックに対して
ファンクション入力パラメータ	31	C言語ファンクション、IEC言語ファンクション
ファンクションブロックのパラメータ総数	32	入力、出力への割り振りが自由。少なくとも1つの出力パラメータが必要です。
変換関数	128	ライブラリに登録可能な最大数
I/O構成	255	ライブラリに登録可能な最大数
I/Oボード	255	ライブラリに登録可能な最大数
I/O装置機器	255	ライブラリに登録可能な最大数
I/OボードのOEMパラメータ	16	

B. 言語リファレンス

B.1 プロジェクトの構成

ISaGRAF のプロジェクトは複数の**プログラム**から成立っています。個々のプログラムがツリー構造でリンクされて一つのプロジェクトを形成しています。プログラムは **SFC**、**FC (Flow Chart)**、**FBD**、**LD**、**ST**、**IL** の6種類の言語から作られます。

B.1.1 プログラム

プログラムとはプログラミングの一つのユニットのことであり、プロセス中の**変数**間の操作を記述します。プログラムには**シーケンシャル**な部分と**サイクリック**な部分があります。サイクリックなプログラムとはターゲットのスキャン毎に実行されるプログラムであり、シーケンシャルなプログラムは**SFC**言語、あるいは、**FC**言語のルールに基づいて実行されます。

各プログラムは互いに階層的にリンクされています。トップレベルのプログラムはシステムによって起動されます。サブプログラム(下位の階層)はそれらの親プログラムによって起動されます。各プログラムは6種類の言語のその場に応じていずれかの言語で記述されます。

- ・シーケンシャルファンクションチャート(SFC): 高級レベルプログラミング
- ・フローチャート(FC): 高級レベルプログラミング
- ・ファンクションブロックダイアグラム(FBD): サイクリックオペレーション
- ・ラダーダイアグラム(LD): プールオペレーション
- ・構造化テキスト(ST): サイクリックオペレーション
- ・命令リスト(IL): ローレベルオペレーション

通常は1個のプログラムに複数の言語を混在させることはできません。ただし、LDとFBDは同一プログラムの中にミックスできます。

B.1.2 サイクリック及びシーケンシャルなセクション

プログラムの階層は5つのメイン**セクション**(グループ)に分けられます。

Begin 毎サイクルのはじめに実行されるプログラム
シーケンシャル SFC、FCダイナミックルールに従ったプログラム
End 毎サイクルの最後に実行されるプログラム
ファンクション類 ... どのプログラムからもコールできるサブプログラムの集合

'Begin' や 'End' セクションはサイクリックな動作を記述し、時間依存しないプログラムです。'シーケンシャル' セクションのプログラムはシーケンシャルな動作を記述し、同期などの時間依存型のプログラムです。'Begin' セクションのメインプログラムはランタイムサイクル毎のはじめに実行されます。'End' セクションのメインプログラムはランタイムサイクル毎の終りに実行されます。'シーケンシャル' セクションのメインプログラムはSFCやFCのダイナミックルールに基づいて実行されます。

プロジェクトの構成

"ファンクション" セクションのプログラムはプロジェクト内の他のプログラムから呼出されるサブプログラムのことを表します。"ファンクション" セクションのプログラムは同じセクションのプログラムを更に呼出すことができます。

シーケンシャルセクションでのメイン、チャイルドプログラムは必ず **SFC**、あるいは、**FC** 言語で記述されなければなりません。サイクリックセクションの **begin** や **end** は、**SFC** や **FC** 言語では記述できません。どのプログラムも一つ以上のサブプログラムを持つことができます。シーケンシャルセクションのプログラムは一つ以上の **SFC**、**FC** チャイルドプログラムを持つことができます (ただし、同じ言語で)。サブプログラムは **SFC** や **FC** 言語で記述することはありません。

一般的に **Begin** セクションのプログラムは入力デバイスのフィルタリングして取り込むような初期的な処理 (変換関数など) に使われます。取り込んだ変数は **シーケンシャルセクション** で使用されます。 **End** セクションのプログラムは **シーケンシャルセクション** で使われた変数の後処理 (内部変数を外部出力デバイスへ送り出すような処理) を行うときに使われます。

B.1.3 チャイルドSFC、FCプログラム

シーケンシャルセクションのどの **SFC**、**FC** プログラムも他の **SFC**、**FC** プログラムをコントロールすることができます。このような下位のプログラムを **チャイルドSFCプログラム** といいます。 **チャイルドSFCプログラム** はパラレル処理のプログラムとして扱われます。即ち、親プログラムから起動、停止 (kill)、凍結 (frozen)、再起動させることができます。ただし、親プログラムもチャイルドプログラムも **SFC** 言語で書かれていなければなりません。 **チャイルドSFCプログラム** はローカル変数、ローカル定義を持つことができます。

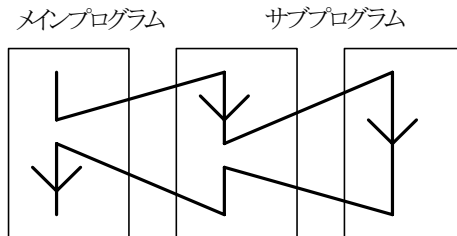
親プログラムがチャイルド **SFC** プログラムを起動するとき、チャイルド **SFC** プログラムの **インシャルステップ** に **SFCTOKEN(token)** をセットします (活性化します)。この起動コマンドは **GSTART** ステートメントで記述されます。もし、親プログラムがチャイルド **SFC** プログラムを停止 (kill) させると、チャイルドプログラムの **ステップ** に存在していた全トークンをクリアさせます。この停止コマンドは **GKILL** ステートメントで記述されます。

親プログラムがチャイルド **SFC** プログラムを凍結 (frozen) させると、チャイルドプログラムの **ステップ** に存在していたトークンを全てクリアしますが、これらの状態はメモリに保存されます。この凍結コマンドは **GFREEZE** ステートメントで記述されます。親プログラムが凍結したチャイルド **SFC** プログラムを再起動すると、凍結の時にクリアされたトークンが復元されます。この再起動コマンドは **GRST** ステートメントで記述されます。

シーケンシャルセクションの **FC** プログラムは別の **FCサブプログラム (チャイルドFCプログラム)** を制御することができます。 **SFC** と異なり、親 **FC** プログラムは **FCサブプログラム** が実行中は処理が待たされます。 **FC** プログラムでは親プログラムと **FCサブプログラム** が同時に実行することはできません。

B.1.4 ファンクションとサブプログラム

サブプログラムやファンクションは親プログラムから起動されます。親プログラムの処理はサブプログラムやファンクションの処理が終了するまで待たされます。



どのセクションのプログラムも一つ以上のサブプログラムを持つことができます。サブプログラムは一個の親プログラムにのみ所有されます。サブプログラムはローカル変数やローカル定義を持つことができます。**SFC**、**FC**以外のプログラムはサブプログラムとして記述することができます。**"ファンクション"** セクションのプログラムはプロジェクト内の他のプログラムから呼び出されるサブプログラムとして使われます。**"ファンクション"** セクションのプログラムは同じセクションの他のプログラムを呼び出すことができます。ファンクションはライブラリに格納されている場合もあります。

注意: ISaGRAFは**リカーシブ** (再入可能) な処理をサポートしていません。もし、**"ファンクション"** セクションのプログラムが自分自身や自分のサブプログラムから呼び出されるような場合はランタイムエラーが発生します。

注意: ファンクションとサブプログラムはローカルな変数のローカルな値 (親プログラム毎の値) をストアしません。ファンクションとサブプログラムにはインスタンスが生成されないため、ファンクションブロックをコールすることができません。

サブプログラムのインタフェースは**タイプ**と**ユニークな名前**をもった入出力パラメータをもって明示的に定義されなければなりません。**ST**言語のルールに基づいて出力パラメータ名はサブプログラム名と同じである必要があります。

以下に、サブプログラム内で出力パラメータを書き込む方法を言語別にまとめます。

ST: サブプログラム名と同じ名前の出力パラメータ名に代入します。

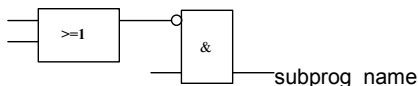
```
subprog_name := <式>;
```

IL: シーケンスの最後での現在結果の値 (IL レジスタ) が出力パラメータに格納されます。

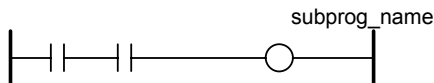
```
LD 10
ADD 20 (* 出力パラメータの値 = 30 *)
```

プロジェクトの構成

FBD: 出力パラメータとサブプログラム名を同じにします。

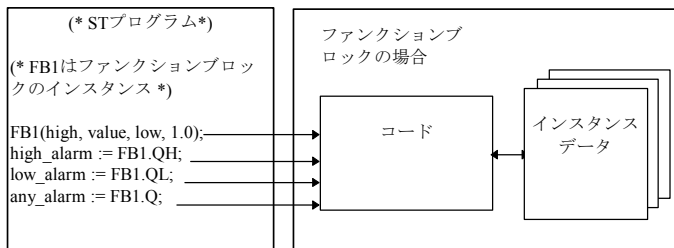


LD: 出力パラメータ名のコイルを使います。



B.1.5 ファンクションブロック

ファンクションブロックはLD、FBD、ST、IL言語で記述できます。ファンクションブロックにはインスタンスが作られます。即ち、個々のファンクションブロックインスタンスはローカルなスタティックデータを持ちます。ファンクションブロックをコールする場合は、実際はファンクションブロックのインスタンスをコールすることになります。プログラムのコードは同一のものですが、データはここに確保されたものになります。インスタンスの変数に保存される値はサイクル毎に保存されます。



注意:

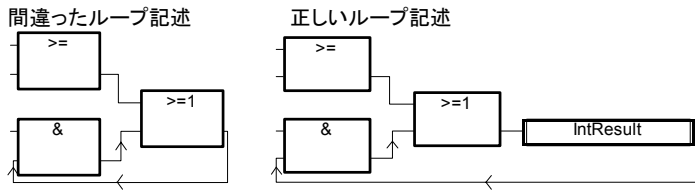
- ・IEC言語でかかれたファンクションブロックは別のファンクションブロックを読み出すことはできません。インスタンスの生成はファンクションブロック自身にのみ有効です。以下に、IEC言語のファンクションブロックから読み出すことができないファンクションブロックを示します。

SR, RS, R_Trig, F_Trig, SEMA, CTU, CTD, CTUD, TON, TOF, TP, CMP, StackInt, AVERAGE, HYSTER, LIM_ALARM, INTEGRAL, DERIVATE, BLINK, SIG_GEN

- ・同様に、立ち上がり接点、立ち下がり接点、セットコイル、リセットコイルも使えません。

- ・タイマ制御用の TSTART、TSTOP ファンクションはターゲットのバージョン 3.0x のファンクションブロック内では使えません。3.20 以降では扱えます。

- ・ファンクションブロックの中でのループを作成する場合は、ループの前にローカル変数を定義する必要があります。以下に例を示します。



B.1.6 言語の説明

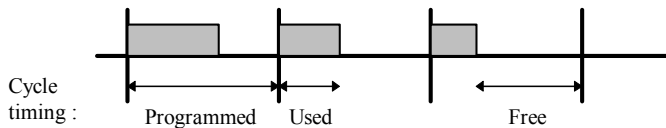
プログラムは以下のグラフィック／文字型の言語で記述されます。

- ・シーケンシャルファンクションチャート(SFC) : 高級レベルプログラミング
- ・フローチャート(FC) : 高級レベルプログラミング
- ・ファンクションブロックダイアグラム(FBD) : サイクリックオペレーション
- ・ラダー ダイアグラム(LD) : ブールオペレーション
- ・構造化テキスト(ST) : サイクリックオペレーション
- ・命令リスト(IL) : ローレベルオペレーション

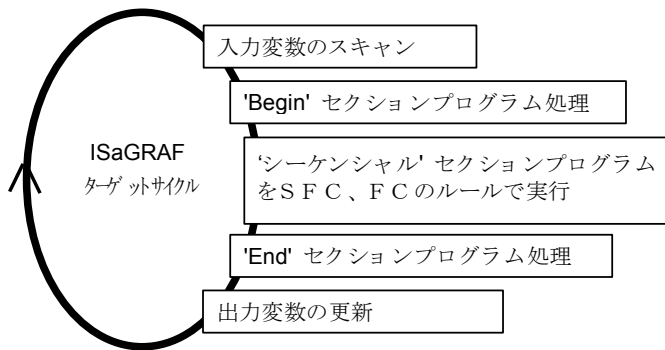
通常は一つのプログラムは一つの言語で記述します。使用するプログラム言語はプログラムを新規作成するときに決定し、その後は変更できません。なお例外としFBDとLDプログラムはミックスしてプログラムできます。

B.1.7 実行ルール

ISaGRAFは**同期型**システムです。全処理はあるクロックによって起動されます。このクロックの周期をサイクルタイムといいます(一般的なPLCではスキャンタイムと呼ばれる場合もありますが、ISaGRAFではスキャンに使用しないFreeな時間を含めた期間を指します)。



ターゲットのサイクルタイムの間での基本的な処理は以下のようになります。



このシステムで以下のことが可能になります。

- 1回のサイクル中は入力変数は同じ値であることが保証されます。
- 出力変数はサイクル中は1回のみ更新されます。
- 異なるプログラムからグローバル変数を安全に扱うことができます。
- アプリケーション全体の応答時間を予測あるいは制御できます。

B.2 共通オブジェクト

ここではISaGRAFのプログラミングデータベースの**共通オブジェクト**に関して解説します。
この共通オブジェクトは **SFC**, **FBD**, **LD**, **ST** あるいは **IL** 言語のいずれでも使えます。

B.2.1 基本タイプ

プログラムで使われる定数、変数には以下の基本のタイプがあります。グラフィックなプログラミングでも言語ベースのプログラミングでもこのタイプは同じように扱われます。

- | | |
|----------------------------|--------------------------|
| • ブール型(BOOLEAN): | True(真), Flase(偽)を持つブール値 |
| • 整数・実数型(ANALOG): | 整数又は実数(浮動小数値)のアナログ値 |
| • タイマ型(TIMER): | タイマ型値 |
| • 可変長文字列型(MESSAGE): | 文字列 |

注意: タイマ型値は最大が1日の長さであり、日時を保存するためには使えません。

B.2.2 定数表現

定数は一つのタイプしか持つことができません。異なるタイプで同じ表現の定数は定義できません。

B.2.2.1 ブール型定数

ブール型定数には以下の2つがあります。

- **TRUE** 整数値の1に等しい
- **FALSE** 整数値の0に等しい

"True" と "False" はキーワードに登録されていて大文字、小文字の区別はありません。

B.2.2.2 整数型定数

整数型定数値は符号付きの32ビット値で **-2147483647** から **+2147483647** の値を持ちます。以下の表記方法で表現されます。プレフィックスによってベースを決めています。

ベース	プレフィックス	例
10進数	(なし)	-908
16進数	"16#"	16#1A2B3C4D
8進数	"8#"	8#1756402
2進数	"2#"	2#1101_0001_0101_1101

2進数表現で使われているアンダースコア('_')は区切りの目的であり、意味はありません。2進数を読みやすくするためのものです。

B.2.2.3 実数型定数

実数型定数表現は**10進数**、及び、**浮動小数点**表示されます。整数型値との違いを表現するために小数点('.')を使います。浮動小数点表現は**E**あるいは**F**を使って**仮数部分**と**指数部分**を区切ります。指数部は-37 から +37 の値を取ります。以下に実数型定数の表現例を示します。

3.14159、	-1.0E+12、
+1.0、	1.0F-15、
-789.56、	+1.0E-37

"123"は実数型定数にはなりません。正しい表現方法は "123.0"となります。

B.2.2.4 タイマ型定数

タイマ型定数値は**0秒**から**23時間59分59秒999ミリ秒**(23h59m59s999ms)まで定義できます。最小単位はミリ秒です。

時間....."h" が時間の後に付きます。
分....."m" が分の後に付きます。
秒....."s"が秒の後に付きます。
ミリ秒....."ms"がミリ秒の後に付きます。

タイマ型定数を表記する際には必ず、"T#" あるいは "TIME#" のプレフィックスをつける必要があります。大文字、小文字の区別はありません。下記は記述例です。

T#1H450MS 1 時間と 450 ミリ秒
time#1H3M 1 時間 3 分

例えば"0"はタイマ型定数ではなく、単なる整数型定数値となります。

B.2.2.5 可変長文字列型定数

文字列あるいは可変長文字列型定数は必ず ' ' (これらは全て半角文字)で囲まれる必要があります。例えば、

‘これは文字列です。’

注意: アポストロフィ(')そのものを可変長文字列型定数の中を含めることはできません。また可変長文字列型定数はソースコード内に一行に収まっていなければなりません。最大長は255半角文字(スペースも含む)です。

空の可変長文字列型定数は2つのアポストロフィ('')で表現します。タブやスペースは含めません。

" (* 空の可変長文字列型定数 *)

注意: 辞書エディタで可変長文字列型変数の初期値を与えるときは最大長を指定して、(')を付けないで直接可変長文字列型を書きます。

印刷不可能な文字を表現する場合に ('\$')シンボルを使います。以下にいくつかの例を示します。

文字表現	意味	ASCII (16 進数)	例
\$\$	'\$' character	16#24	'I paid \$\$5 for this'
\$'	アポストロフィ	16#27	'Enter '\$Y\$' for YES'
\$L	改行	16#0a	'next \$L line'
\$R	キャリッジリターン	16#0d	' Ilo \$R He'
\$N	新しい行	16#0d0a	'This is a line\$N'
\$P	新しいページ	16#0c	'lastline \$P first line'
\$T	タブ	16#09	'name\$Tsize\$Tdate'
\$hh (*)	任意の文字	16#hh	'ABCD = \$41\$42\$43\$44'

(*) "hh" は表現したい文字の16進数アスキーコードを示します。

B.2.3 変数

変数はプログラムに対してローカルとグローバルの区別があります。ローカルは編集中のプログラムのみから参照可能です。グローバルは同一プロジェクトのどのプログラムからでも参照可能です。変数名は以下のルールに従う必要があります。

最大16半角文字

最初の文字は英字(a-z)

以降の文字は英字、数字、_(アンダースコア)のいずれか

B.2.3.1 予約語

予約語としては以下のものがあります。これらの予約語はプログラム、変数名、C言語ファンクション、C言語ファンクションブロックなどの名前には使えません。

- A ANA, ABS, ACOS, ADD, ANA, AND, AND_MASK, ANDN, ARRAY, ASIN, AT, ATAN,
- B BCD_TO_BOOL, BCD_TO_INT, BCD_TO_REAL, BCD_TO_STRING, BCD_TO_TIME, BOO, BOOL, BOOL_TO_BCD, BOOL_TO_INT, BOOL_TO_REAL, BOOL_TO_STRING, BOOL_TO_TIME, BY, BYTE,
- C CAL, CALC, CALCN, CALN, CALNC, CASE, CONCAT, CONSTANT, COS,
- D DATE, DATE_AND_TIME, DELETE, DINT, DIV, DO, DT, DWORD,
- E ELSE, ELSIF, EN, END_CASE, END_FOR, END_FUNCTION, END_IF, END_PROGRAM, END_REPEAT, END_RESSOURCE, END_STRUCT, END_TYPE, END_VAR, END_WHILE, ENO, EQ, EXIT, EXP, EXPT,
- F FALSE, FEDGE, FIND, FOR, FUNCTION,
- G GE, GFFREEZE, GKILL, GRST, GSTART, GSTATUS, GT,
- I IF, INSERT, INT, INT_TO_BCD, INT_TO_BOOL, INT_TO_REAL, INT_TO_STRING, INT_TO_TIME,
- J JMP, JMPC, JMPCN, JMPN, JMPNC,
- L LD, LDN, LE, LEFT, LEN, LIMIT, LINT, LN, LOG, LREAL, LT, LWORD,

共通オブジェクト

M	MAX, MID, MIN, MOD, MOVE, MSG, MUL, MUX,
N	NE, NOT,
O	OF, ON, OPERATE, OR, OR_MASK, ORN,
P	PROGRAM
R	R, REDGE, READ_ONLY, READ_WRITE, REAL, REAL_TO_BCD, REAL_TO_BOOL, REAL_TO_INT, REAL_TO_STRING, REAL_TO_TIME, REDGE, REPEAT, REPLACE, RESSOURCE, RET, RETAIN, RETC, RETCN, RETN, RETNC, RETURN, RIGHT, ROL, ROR,
S	S, SEL, SHL, SHR, SIN, SINT, SQRT, ST, STN, STRING, STRING_TO_BCD, STRING_TO_BOOL, STRING_TO_INT, STRING_TO_REAL, STRING_TO_TIME, STRUCT, SUB, SYS_ERR_READ, SYS_ERR_TEST, SYS_INITALL, SYS_INITANA, SYS_INITBOO, SYS_INITTMR, SYS_RESTALL, SYS_RESTANA, SYS_RESTBOO, SYS_RESTTMR, SYS_SAVALL, SYS_SAVANA, SYS_SAVBOO, SYS_SAVTMR, SYS_TALLOWED, SYS_TCURRENT, SYS_TMAXIMUM, SYS_TOVERFLOW, SYS_TRESET, SYS_TWRITE, SYSTEM,
T	TAN, TASK, THEN, TIME, TIME_OF_DAY, TIME_TO_BCD, TIME_TO_BOOL, TIME_TO_INT, TIME_TO_REAL, TIME_TO_STRING, TMR, TO, TOD, TRUE, TSTART, TSTOP, TYPE,
U	UDINT, UINT, ULINT, UNTIL, USINT,
V	VAR, VAR_ACCESS, VAR_EXTERNAL, VAR_GLOBAL, VAR_IN_OUT, VAR_INPUT, VAR_OUTPUT,
W	WHILE, WITH, WORD,
X	XOR, XOR_MASK, XORN

アンダースコア ('_')で始まるキーワードは内部で使用するキーワードなのでプログラム記述には使うことができません。

B.2.3.2 直接表現変数(Directly Represented Variables)

I/O変数が接続されていないI/Oチャンネルはフリーチャンネルと呼ばれます。ISaGRAF の **直接表現変数**はこのフリーチャンネルをプログラムのソースコードの中で直接扱うことができます。直接表現変数の識別子として必ず"%文字"で始まる変数名となります。

以下に、単一のI/Oボードに対する直接表現変数の名前の付け方について示します。ここで、"s" はボードのスロット番号を示します。"c" はI/Oチャンネル番号を示します。

%IXs.c ブール型入力ボードのフリーチャンネル
%IDS.c 整数型入力ボードのフリーチャンネル
%ISS.c 可変長文字列型入力ボードのフリーチャンネル
%QXS.c ブール型出力ボードのフリーチャンネル
%QDS.c 整数型出力ボードのフリーチャンネル
%QSS.c 可変長文字列型出力ボードのフリーチャンネル

以下に、複数のI/Oボードから成り立っているI/O装置機器に対する直接表現変数の名前の付け方について示します。ここで、"**s**" はボードのスロット番号を示します。"**b**" はI/O装置機器の中でのシングルボードのインデックス番号を示します。"**c**" はI/Oチャンネル番号を示します。

%IX**s.b.c** ブール型入力ボードのフリーチャンネル
 %ID**s.b.c** 整数型入力ボードのフリーチャンネル
 %IS**s.b.c** 可変長文字列型入力ボードのフリーチャンネル
 %QX**s.b.c** ブール型出力ボードのフリーチャンネル
 %QD**s.b.c** 整数型出力ボードのフリーチャンネル
 %QS**s.b.c** 可変長文字列型出力ボードのフリーチャンネル

以下にこれらの例を示します。

%QX1.6 ボードスロット番号1、I/Oチャンネル番号6のブール型出力カード
 %ID2.1.7 ボードスロット番号2のI/O装置機器で、ボード番号が1、I/Oチャンネル番号7の整数型入力ボード

直接表現変数は**実数型**データを持つことはできません。

B.2.3.3 ブール型変数

論理(ブール型)値変数は**TRUE**、**FALSE**のいずれかのブール型値を持つことができます。ブール型変数は以下の**属性**を持つことができます。

内部: プログラムによって更新されるメモリ上の変数
入力: 入力デバイスと接続される変数(システムが更新)
出力: 出力デバイスと接続される変数
定数: リードオンリーの内部変数

注意: ブール型変数を宣言するときはデバッグ時のTRUE、FALSEの表示を別の文字列で置き換えることが可能です。ただし、この置き換えられる文字列はプログラムでは扱えません。**定義ワード**で宣言されている場合は例外的に扱うことができます。

B.2.3.4 アナログ変数

連続量を扱うアナログ変数には符号付き整数と実数(浮動小数点)があります。それぞれのフォーマットを以下に示します。

整数	32 bit 符号付き整数	-2147483647 から +2147483647
実数	IEEE 32 ビット浮動小数値(単精度)	1 符号ビット + 23 仮数部ビット + 8 指数部ビット (実数の指数部は -37 ~ +37 の間になります。)

アナログ変数は以下の**属性**を持ちます。

内部:..... プログラムによって更新されるメモリ上の変数
入力:..... 入力デバイスと接続される変数(システムが更新)
出力:..... 出力デバイスと接続される変数
定数:..... リードオンリーの内部変数

注意: 実数型変数が外部のI/Oデバイスに接続される場合は、I/Oドライバは整数型変数として扱います。

注意: 整数型と実数型値は同一の整数型演算式でミックスさせて使用できません。

B.2.3.5 タイマ型変数

タイマ型変数はタイマやカウンタに使われます。タイマ型変数は正数で最大**23時間59分59秒999ms**まで定義可能です。タイマ型変数は32ビットを使用し、内部的にはミリ秒単位の正数になります。

注意: **タイマ型変数の属性は内部と定数のみ**で、入力、出力属性を指定することはできません。

タイマの値はISaGRAFのシステムが自動的に更新します。タイマが**アクティブ**の時はターゲットのリアルタイムクロックから読み出された信号で自動的に更新されています。ST言語では以下のステートメントを使用できます。

TSTART	タイマの自動リフレッシュを起動
TSTOP	タイマの自動リフレッシュを停止

B.2.3.6 可変長文字列型変数

可変長文字列型は文字列を含んでいます。文字列変数の長さは可変長です。ただし、可変長文字列型変数を辞書エディタで宣言した際の最大長を越えることはできません。最大でも255半角文字までとなります。可変長文字列型変数は以下の**属性**を持つことができます。

内部:..... プログラムによって更新されるメモリ上の変数
入力:..... 入力デバイスと接続される変数(システムが更新)
出力:..... 出力デバイスと接続される変数
定数:..... リードオンリーの内部変数

可変長文字列型変数はアスキーコードの**0~255**までのいずれのコードを含むことができます。NULL(**0**)キャラクタも文字列に含むことも可能です。ただし、CファンクションによってはNullを含む文字列を処理できない場合もあります。

B.2.4 コメント

STプログラム、ILプログラムでは**コメント**を挿入することができます。"(*"ではじまり、"*)"で終了する必要があります。STプログラムのどこにでもコメントを挿入可能です。また、複数行にわたるコメントも可能です。以下に例を示します。

```
counter := ivalue; (* assigns the main counter *)
(* 2行にまたがる
コメント行 *)
c := counter (* コメントはどこにでも挿入可能です *) + base_value +
1;
```

コメント内部には "(*"文字は含むことができません。

注意: ILプログラムではコメントの挿入場所は行の最後に限られています。

B.2.5 定義ワード

ISaGRAFでは定数表現、ブール型変数のTRUE、FALSE表現、キーワード、**ST**で使われる複雑な式などを別名で再定義することができます。定義ワードによりこの置き換えを可能にします。例えば、

定義ワード	定義内容
YES	TRUE
PI	3.14159
OK	(auto_mode AND NOT (alarm))

YES、PI、OKといった**識別子**は例えばSTプログラムのどこでも扱って複雑な表現を置き換えることができます。以下にSTプログラムに定義ワードを用いた例を示します。

```
If OK Then
  angle := PI / 2.0;
  isdone := YES;
End_if;
```

定義ワードはそのプログラム内からのみ参照できる**ローカル定義**、同一プロジェクト内のどのプログラムからでも参照可能な**グローバル定義**、どのプロジェクトのどのプログラムからでも参照可能な**共通定義**に分かれます。共通定義だけはアーカイブマネージャで保存します。

注意: もし、同一の定義ワードに対して2回以上定義がなされた場合は最後に書かれた定義が使われます。例えば、

定義付け:	(定義ワード)	(定義内容)
	OPEN	FALSE
	OPEN	TRUE

と2回定義すると、結局意味:
となります。

OPEN	TRUE
------	------

共通オブジェクト

定義ワード名は以下のルールに従う必要があります。

- 最大16半角文字
- 最初の文字は英字(a-z)
- 以降の文字は英字、数字、_(アンダースコア) のいずれか

注意: 定義ワード内で別の定義ワードは使用できません。例えば、以下の例は無効です。

PI 3.14159

PI2 PI*2 :無効

定数、変数、または演算を使うようにします。

PI2 6.28318

B.3 SFC 言語

SFC言語はシーケンシャルな手続きを記述するときに使うグラフィック言語です。プロセスのアクションを表現する部分(ステップ)と、ステップ間の遷移条件を表現する部分(トランジション)との2つのグラフィック記号から成り立っています。各トランジションには**布尔型の条件**を記述します。ステップの中のアクションは他の言語(ST、IL、LD、FBD)で記述します。

B.3.1 SFC チャートメインフォーマット

SFCプログラムは、**方向を持ったリンク**によって接続された**ステップ**と**トランジション**の組み合わせによって表されます。マルチプルリンクは、分岐と結合を表すのに使用されます。メインチャートのプログラムの一部分を**マクロステップ**という記号で置き換える事により、別の場所に分離して記述する事ができます。SFC の基本的な規則は次の通りです。

- ステップのあとに別のステップを続けることはできません。
- トランジションのあとに別のトランジションを続けることはできません。

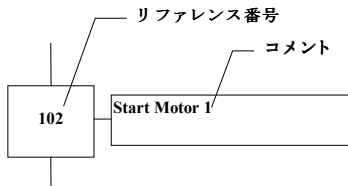
B.3.2 SFC 基本コンポーネント

SFC 言語の基本的なコンポーネント(グラフィックシンボル)は次の通りです。

- ステップ と イニシャルステップ
- トランジション
- 方向を持ったリンク
- ステップへのジャンプ

B.3.2.1 ステップとイニシャルステップ

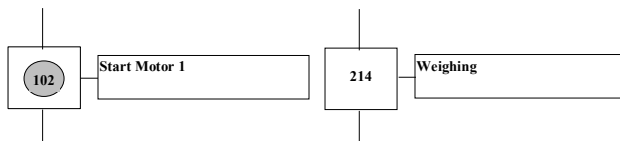
ステップはひとつの**正方形**によって表現され、各ステップは正方形の中に書かれた**リファレンス番号**によって参照されます。ステップの主な内容を表すコメントが、ステップ記号にリンクされた長方形の中に記述されます。このコメントはプログラミング言語の一部分ではありませんので自由な記述が可能です。このような表現方式をステップの**レベル1**表現と呼びます。



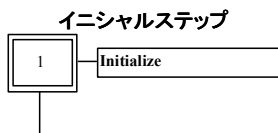
実行時には、**活性状態**のステップには**トークン**が示されます：

活性化ステップ

非活性化ステップ



SFCプログラムの初期状態は、二重枠で囲まれた**イニシャルステップ**記号で表されます。プログラムは自動的に各イニシャルステップから実行開始されます。



SFCプログラムは、**少なくとも1つ**のイニシャルステップを含んでいなければなりません。

以下はステップがもつ属性です。他のプログラム言語でもこれらのデータを使用する事ができます:

GSnnn. x ステップの活性状態(ブール値)

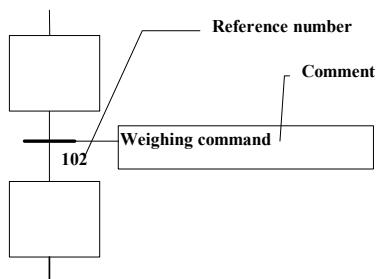
GSnnn. t ステップの活性持続時間(タイマ値)

(ここで、nnnはステップのリファレンス番号)

B.3.2.2 トランジション

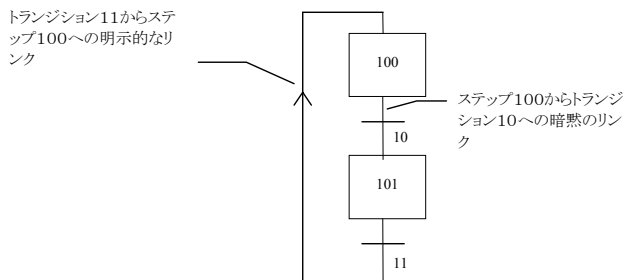
トランジションは、接続リンクに交差する小さい水平のバーによって表されます。各トランジションはトランジションの記号の下に書かれた**リファレンス番号**で参照されます。

トランジションの主な内容はトランジション記号の右側にコメントとして記述できます。このコメントはプログラミング言語の一部ではなく自由に記述できます。この様な記述のしかたをトランジションの**レベル1**表現と呼びます:



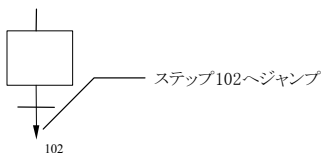
B.3.2.3 リンク

ステップとトランジションを接続する為に、方向を持ったリンクと呼ばれる一本のラインが用いられます。リンクの方向が明示的に与えられていないときは、上から下の向きにリンクされます。



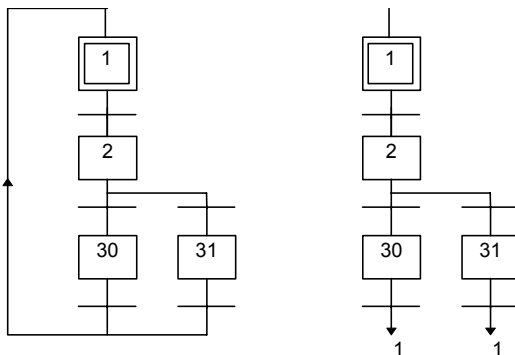
B.3.2.4 ステップへのジャンプ

トランジションからステップにリンクを接続する際に常に接続ラインを描く必要はなく、ジャンプ記号を用いる事ができます。ジャンプ先として目的のステップ番号を記述する必要があります。



ステップからトランジションへのジャンプ記号を用いる事はできません。

ジャンプの例ー 以下の2つのチャートは等価となります。



B.3.3 分岐と結合

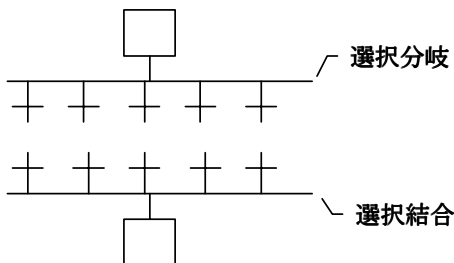
分岐とは、1 つのSFC記号(ステップかトランジション)からその他の複数のSFC記号に分岐する多重接続リンクを意味します。

結合とは、複数のSFC記号から1つのSFC記号に収束する多重接続リンクを意味します。

分岐と結合には、それぞれ選択(OR)と並列(AND)の2種類があります。

B.3.3.1 選択(OR)分岐

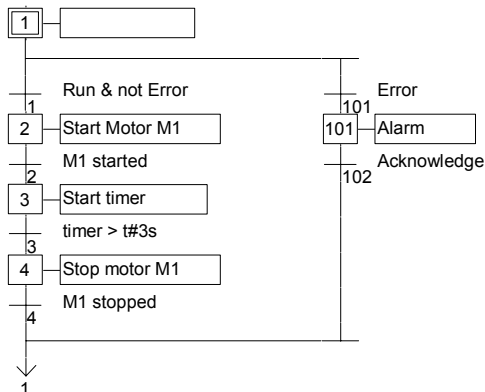
選択分岐とは1つのステップから複数のトランジションへの分岐で、複数の分岐の内の1つのみ選択する事を許します。また選択結合は複数のトランジションからひとつのステップへの結合で、通常、選択分岐によって始められたSFC分岐をまとめる(結合させる)のに使用されます。選択分岐と選択結合は、一本の水平の線によって表されます。



注意: 選択分岐の直後の複数のトランジションは**暗示的な排他条件であってはなりません**。排他性は、実行時に複数の分岐の内一つだけ選ばれる事を保証する様に、トランジションの条件中に明示的に記述する必要があります。

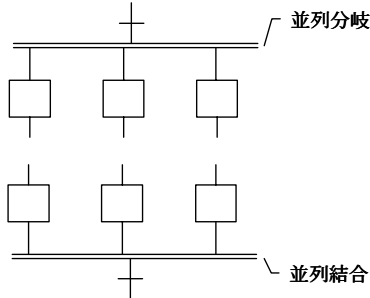
以下に、選択分岐と選択結合の例を示します:

(* 選択分岐と選択結合SFCプログラム *)



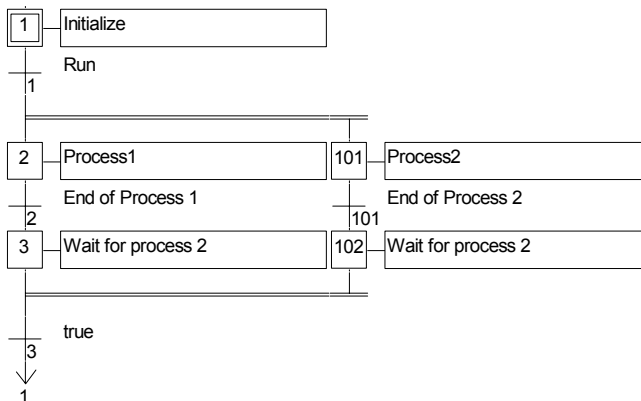
B.3.3.2 並列 (AND) 分岐

並列分岐は 1 つのトランジションから複数のステップへの分岐であり、プロセスの並行実行を意味します。また並列結合は複数のステップからひとつのトランジションへの結合です。一般に並列結合は並列分岐によって始められたSFC分岐をまとめる(結合させる)のに使用されます。並列分岐と並列結合は、二重の水平線によって表されます。



並列分岐と並列結合の例:

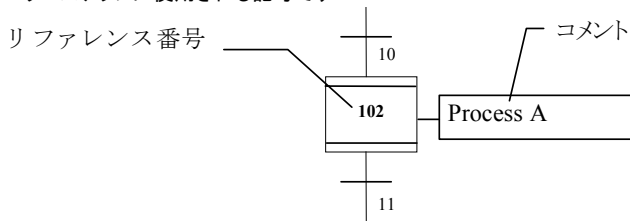
(* 並列分岐と並列結合をもったSFCプログラム *)



B.3.4 マクロステップ

マクロステップは、複数のステップとトランジションの重複しない組み合わせからなります。マクロステップのボディ(内容)は、同じSFCプログラム中のほかの場所に分けて記述します。マクロステップはメインSFCチャートに一個の記号で表されます。

下図がマクロステップに使用される記号です：



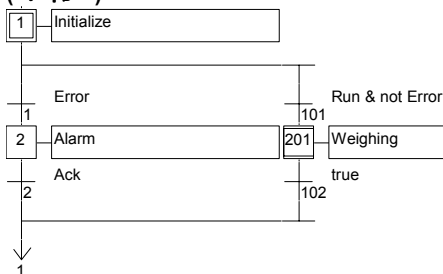
マクロステップ記号の中に書かれるリファレンス番号は、マクロステップ本体の先頭ステップのリファレンス番号です。マクロステップボディは**開始ステップ**で始まって**終了ステップ**で終わらなければならない、自己完結的でなければなりません。開始ステップはいかなる上側のリンク(直前のトランジション)も持たず、また終了ステップはいかなる下側のリンク(直後のトランジション)も持ちません。マクロステップ記号を別のマクロステップの本体の中で用いることもできます。

注意： マクロステップは複数のステップとトランジションのユニークな組み合わせであり、ひとつのSFCプログラムにおいて同じマクロステップを2回以上読み出すことはできません。

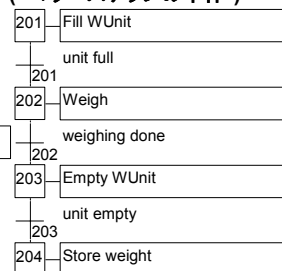
マクロステップの例：

(* マクロステップを持つSFCプログラム *)

(* メイン *)



(* マクロステップの本体*)



B.3.5 ステップの中でのアクション

SFCステップのレベル2表現は、そのステップが活性状態の間に実行すべきアクションの詳細な記述方式です。この記述は**SFCの書式**や**ST**の様な他の言語を用いて表現します。アクションの基本的な型は：

- ブールアクション
- ST言語でプログラムされたパルスアクション
- ST言語でプログラムされたノンストアードアクション
- SFCアクション

一つのステップの中に複数のアクション(同じ種類でも、異なっても構いません)を記述することができます。他の言語の使用を可能にする方法は以下の2通りがあります。

- サブプログラムをコール
- IL言語による表現

B.3.5.1 ブールアクション

ブールアクションは現在のステップの状態をブール型変数に代入します。ブール型の変数の属性は内部、もしくは出力変数です。値の代入はステップが活性化したときと非活性になるときに行われます。

以下に基本的なブールアクションの文法を示します。(＜>は入力しません)

<ブール型変数名> (N); ... ステップの活性化状態を変数に代入

<ブール型変数名>; 同上 (N 属性はオプション)

! <ブール型変数名>; ステップの活性化状態の反転を変数に代入

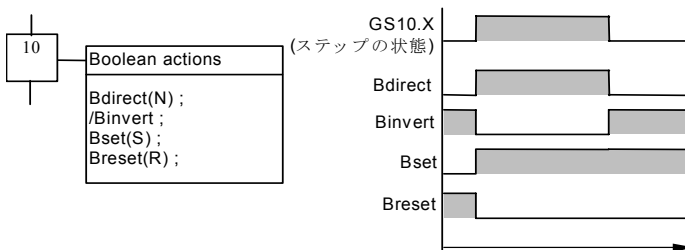
これらの他にも、ステップ状態が活性化になったとき、変数の値をセットあるいはリセットすることもできます。以下にこれらの文法を示します。

<ブール型変数名> (S); ステップの状態が TRUE になったとき変数の値を TRUE にする。

<ブール型変数名> (R); ... ステップの状態が TRUE になったとき変数の値を FALSE にする。

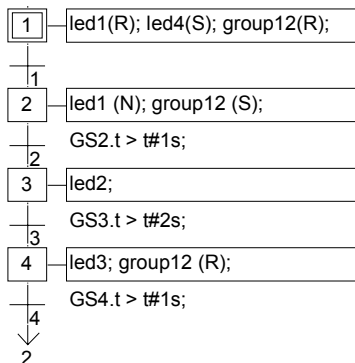
ブール型変数は内部あるいは出力変数でなければなりません。

以下に示すプログラム例とその振る舞いを示します。



ブールアクションの例：

(* ブールアクションを使った SFC プログラム例 *)

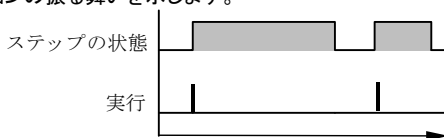


B.3.5.2 パルスアクション

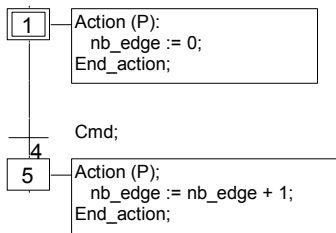
パルスアクションはステップの状態が**活性状態になったとき一回のみ**実行されるST又はILで書かれた命令です。次のような構文で記述します。

ACTION (P) :
 (* STの記述*)
END_ACTION ;

以下にパルスアクションの振る舞いを示します。



パルスアクションの例:

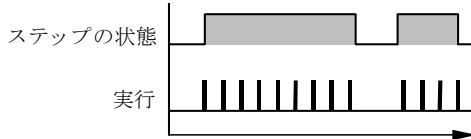


B.3.5.3 ノンストアードアクション

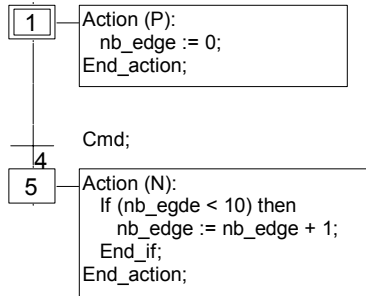
ノンストアード（ノーマル）アクションはステップの状態が**活性の間は毎ターゲットサイクル**実行される、ST又はILで書かれた命令です。次のような文法で記述します。

```
ACTION (N) :
    (* ST ステートメント*)
END_ACTION ;
```

以下にノンストアードアクションの振る舞いを示します：



ノンストアードアクションの例：



B.3.5.4 SFC アクション

SFCアクションではチャイルドSFCのシーケンスをコントロールします。即ち、ステップの状態に応じて起動、停止を行ないます。SFCアクションには**N**(Non stored)、**S**(Set)、**R**(Reset)識別子が付きます。以下に文法を示します。

<チャイルドプログラム> (N); ステップが活性状態になったときにチャイルドシーケンスを起動し、非活性状態になったときにチャイルドシーケンスを停止(Kill)させる。

<チャイルドプログラム> ; 同上 (N 識別子はオプション)

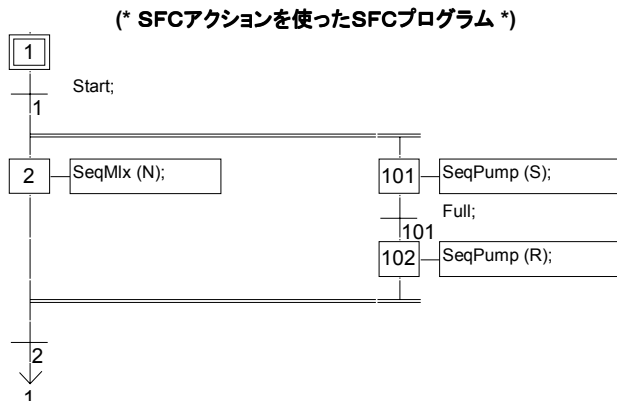
<チャイルドプログラム> (S); ステップが活性状態になったときにチャイルドシーケンスを起動し、非活性状態になったときにはチャイルドシーケンスに対して何もしない。

<チャイルドプログラム> (R); ステップが活性状態になったときにチャイルドシーケンスを停止(Kill)し、非活性状態になったときは何もしない。

アクションで指定されているSFCシーケンスは現在編集中のプログラムの**チャイルドSFCプログラム**である必要があります。

注意: SFCアクションで使用される**S**(Set)、**R**(Reset)識別子はST言語によるパルスアクションとしてプログラムされた **GSTART**, **GKILL** ステートメントと同様の意味を持ちます。

下記はSFCアクションの一例です。SFCプログラムは、**Father** という名前です。**SeqMix** と **SeqPump** という二つの子SFCを持っています。



B.3.5.5 アクションからのファンクションとファンクションブロックの呼び出し

ST、IL、FBDあるいはC言語で記述されたサブプログラム、ファンクション、ファンクションブロックは以下のような構文によりSFCアクションブロックから直接コールすることができます。

サブプログラム、ファンクション、C言語ファンクションの場合:

```

ACTION (P) :
    result := sub_program ( ) ;
END_ACTION;
    
```

または

```

ACTION (N) :
    result := sub_program ( ) ;
END_ACTION;
    
```

C言語または ST、IL、LD、FBD 言語で記述されたファンクションブロックの場合:

```

ACTION (P) :
    Fbinst(in1, in2);
    result1 := Fbinst.out1;
    result2 := Fbinst.out2;
END_ACTION;
    
```

または

```

ACTION (N) :
    Fbinst(in1, in2);
    result1 := Fbinst.out1;
    
```

```

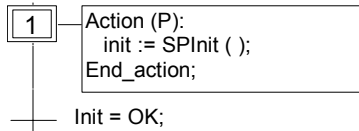
    result2 := Fbinst.out2;
END_ACTION;

```

詳細の文法に関してはST言語のセクションを参照ください。

以下に、アクションブロックからサブプログラムコールを行った例を示します。

(* アクションブロック内からサブプログラムを読み出したSFCプログラム *)



B.3.5.6 IL の文法

下記の構文を用いることによりSFCアクションブロックの中にIL言語で記述されたプログラムを直接埋め込むことができます。

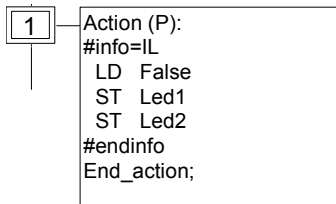
```

ACTION (P) :          (* または N *)
#info=IL
  <命令>
  <命令>
  ....
#endinfo
END_ACTION;

```

"#info=IL" と "#endinfo"という特別なキーワードを使用する際には、正確に上記構文を守るようにして下さい。**大文字と小文字は区別されます**。キーワードの前後にスペースやタブを挿入することは許されません。以下にアクションブロック中のILプログラム例を示します。

(* アクションブロックにILプログラムを記述したSFCプログラム *)



B.3.6 トランジションの条件

各トランジションには、トランジションをクリア（通過）するための条件を規定する**論理式**を記述します。通常、この条件記述にはST言語またはLD言語（Quick LD エディタ）が用いられます。これをトランジションの**レベル2**表記と呼びます。他の言語を使用することも可能です。

- ST言語の文法
- LD言語の文法
- IL言語の文法
- トランジションからのファンクションのコール

注意: トランジションに条件記述がない場合、デフォルト条件として**TRUE**が設定されます。

B.3.6.1 ST の文法

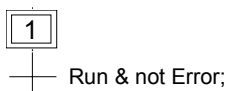
トランジションの**条件**を記述するために**ST言語**を使用することが出来ます。条件式はブール (boolean) でなければならず、下記の構文規定に示すようにセミコロンで終わらねばなりません。

< ブール型の式 > ;

条件記述は、TRUE/FALSE の定数、一つのブール型の入力／内部変数、あるいはブールを導出する変数の組合せで表されます。

以下にトランジションの条件を記述するSTプログラミング例を示します。

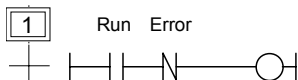
(* トランジションにSTプログラムを使ったSFCプログラム例 *)



B.3.6.2 LD の文法

トランジションの**条件**を記述するため **LD 言語**を使用することが出来ます。LDプログラムは1つのラングのみとなります。コイルの値がトランジションの条件に反映されます。

以下にトランジションの条件を記述するLD プログラミング例を示します:



B.3.6.3 IL の文法

下記の構文により、トランジションをIL言語で直接記述することができます。

```
#info=IL
    <命令>
    <命令>
    ....
#endinfo
```

ILシーケンスの**現在結果**(ILレジスタ)に保持される値によってトランジション条件の判定結果が決定されます。

```
現在結果 = 0  → FALSE
現在結果 <> 0 → TRUE
```

"#info=IL" と "#endinfo"という特別なキーワードを使用する際には、正確に上記構文を守るようにして下さい。**大文字と小文字は区別されます**。キーワードの前後にスペースやタブを挿入することは許されません。以下にアクションブロック中のILプログラム例を示します。

(* トランジションにILプログラムを使った SFC プログラム例 *)

```

1
├──
  #info=IL
  LD  Run
  &N Error
  #endinfo
```

B.3.6.4 トランジションからのファンクションのコール

トランジション条件を判定するために、FBD、LD、ST、ILまたはC言語で記述されたサブプログラムやファンクションをコールすることができます。

構文は以下の通りです。

```
< サブプログラム名 > ( );
```

サブプログラムの戻り値により条件判断が決定されます。

```
戻り値 = 0  → FALSE
戻り値 <> 0 → TRUE
```

トランジション中のサブプログラム例

(* トランジションでサブプログラムを読み出したSFCプログラム例 *)

```

1
├── EvalCond ( );
```

B.3.7 SFCの動作の原則

SFC言語の**5つ**の動作の原則を以下に示します。

初期状態

初期状態は**イニシャルステップ**で表現されます。処理の開始時はこのステップが活性状態になります。各SFCプログラムには**少なくとも一つ**のイニシャルステップが必要です。

トランジションの通過

トランジションには **enable** 状態と **disable** 状態があります。ここで、enable 状態とは、直前のステップが**活性**(トークンが存在)な状態の時をことを言います。直前のステップが活性でないときは disable 状態と言います。トランジションは以下の条件が整っているときにトークンが**通過**します。

- enable 状態になっていて、かつ、
- トランジションの条件がTRUEであること。

活性化ステップの状態変化

トランジションをトークンが通過することは同時に次のステップにトークンが移り、活性状態になります。この結果、トランジション直前のステップは非活性状態となります。

トランジションの同時通過

並列分岐は複数のトランジションでのトークンの遷移が同時に行なわれることを示します。このようなトランジションが分けて記述されているときは、直前のステップの活性状態が条件として使われます。

ステップの同時活性化と非活性化

処理中にステップが同時に活性、非活性になるような場合は優先順位は活性化にあります。

B.3.8 SFC プログラム階層化

ISaGRAFではSFCプログラムの**階層的な構造**を構築することができます。各SFCプログラムは別のSFCプログラムに対して起動、停止などのコントロールを行なうことができます。コントロールされるプログラムは**チャイルドSFC**プログラムと呼ばれます。SFCプログラムは“**親—子**”の関係で階層的にリンクされています。



階層的な構成には以下の基本的なルールがあります。

- 親SFCプログラムを持たないSFCプログラムは**メインSFC**プログラムと呼ばれます。
- メインSFCプログラムの起動は ISaGRAF のシステムがアプリケーションのスタート時に行います。

- 一つのSFCプログラムは複数のチャイルドSFCプログラムを持つことができます。
- チャイルドSFCプログラムは2つ以上の親SFCプログラムを持つことはできません。(1つの親プログラムのみ)
- チャイルドSFCプログラムは親SFCプログラムからのみコントロールされます。
- 親SFCプログラムは自分のチャイルドSFCプログラムのチャイルドSFCプログラム(孫SFCプログラム)をコントロールすることはできません。

親SFCプログラムがそのチャイルドSFCプログラムに対してコントロールできるアクションを以下に示します。

起動	(GSTART) チャイルドSFCプログラムの起動: チャイルドプログラムの各初期ステップを活性状態にします。孫のプログラムは自動的に起動されません。
停止	(GKILL) チャイルドSFCプログラムの停止: チャイルドプログラムの活性状態ステップを全て非活性状態にします。この際、孫プログラムも同時に停止されます。
凍結	(GFREEZE) チャイルドSFCプログラムの凍結: チャイルドプログラムの活性なステップとトランジションの演算を凍結状態にします。ただし、この状態をメモリに保管しておき後ほど再スタートが可能な状況にしておきます。孫プログラムも同時に凍結されます。
再起動	(GRST) 凍結したSFCプログラムの再起動: 凍結されていたステップを全て活性化することによりプログラムを再起動させます。孫プログラムは自動的に再起動はされません。
状況の取得	(GSTATUS) チャイルドSFCプログラムの現在の状況を取得します。(活性状態、非活性状態、凍結状態のいずれか)

B.4 フローチャート言語

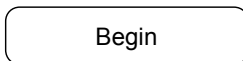
フローチャート(FC) はシーケンシャルな手続きを記述するグラフィカル言語です。フローチャートはアクションとテストと呼ばれるブロックから成り立っていて、これらのブロック間を方向のあるリンクで接続しています。コネクタは分岐や結合を表わすために使います。アクションとテストの内容は ST, IL, LD のいずれかの言語で記述することができます。アクションやテストの内部からは、他の言語(SFC をのぞく)で書いたファンクションやファンクションブロックをコールすることも可能です。フローチャートは別のフローチャートをコールすることもできます。この際、読み出されるフローチャートを **FC サブプログラム**と呼びます。

B.4.1 FC の構成要素

フローチャート言語は以下のグラフィックコンポーネントから成り立っています。

☐ フローチャートの開始

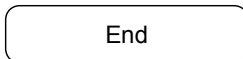
"Begin"シンボルはフローチャートプログラムの開始位置として必ず必要です。フローチャートが起動された時の最初の状態(1 回目のサイクル)になります。以下に"Begin"シンボルを示します。



"Begin"シンボルは必ず下側からチャート内の別のオブジェクトに接続されます。この接続が行われていないようなフローチャートは実行されません。

☐ フローチャートの終了

"End"シンボルはフローチャートの終了を示します。"End"シンボルはプログラム内に必ず1つは必要で、省略することは出来ません。。このシンボルはプログラム実行が終了した時、そのチャートの終了した状態を示します。以下に"End"シンボルを示します。



通常、"End"シンボルの上部には、チャート内の別のシンボルからの接続がありますが、他のオブジェクトから接続されない場合もあります。この場合はこのフローチャートは常にループの処理を行っていることになります。ただし、このような場合でも"End"シンボルはチャートの下部に表現されます。

☐ フローリンク

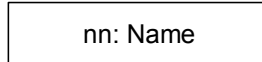
フローリンクはチャート内の2つのポイント間の処理の流れを意味します。フローリンクは必ず矢印を持ちます。以下にフローリンクを示します。



1 個のオブジェクトから複数のフローリンクは引き出すことはできません。

□ アクション

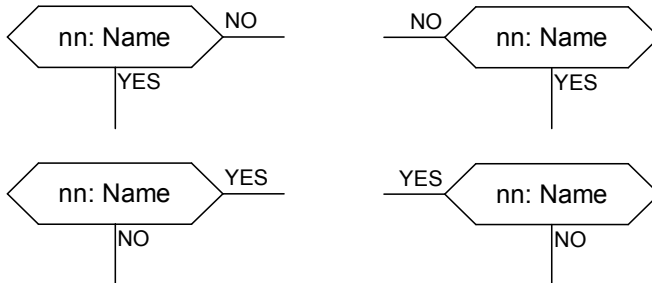
アクションシンボルは実行されるアクションを示します。アクションは番号と名前で識別されます。以下にアクションシンボルを示します。



チャート内で2つの異なるアクションシンボルが同一の名前または番号を持つことはできません。アクションのプログラム記述には ST、LD、IL のいずれかの言語が使えます。アクションシンボルは必ず1本のリンクが入り、1本のリンクが出て行きます。

□ テスト

テスト(条件)シンボルはブール型のテストを行います。テストは番号と名前によって識別されます。テストのプログラムは ST、LD、IL のいずれかの言語で記述されます。テストの評価内容によって、フローが YES あるいは NO の経路へ流れます。以下にテストのシンボルの例を示します。



2つの異なるテストシンボルが同一の番号または名前を持つことはできません。テストの内容は以下のいずれかの言語で記述されます。

- ST 言語による式
- 一行のラングで表現される LD(この場合、コイルには変数名を割り当てる必要はありません)
- 複数行からなる IL 言語。IL レジスタ(現在結果)が条件の結果に使われます。

ST 言語でプログラムされた時は、式の最後にセミコロン(;)をつけることもできますが、つけなくてもエラーにはなりません。LD 言語では、コイルが評価の値となります。評価の結果と流れは以下になります。

- 0 あるいは FALSE : NO
- 1 あるいは TRUE : YES

テストシンボルは必ず一つのリンクが入り、YES、NO の2つのリンクが出て行きます。

FC サブプログラム

フローチャートプログラムでは**階層的**にプログラムを構築することができます。各フローチャートプログラムは別のフローチャートを読み出すことができます。読み出されるプログラムは**チャイルドプログラム**、あるいは、**FC サブプログラム**と呼ばれます。FC サブプログラムを読み出すプログラムは**親プログラム**と呼ばれます。FC プログラムは以下のように親子関係の階層構造を持つことができます。



フローチャート内での**サブプログラム**シンボルは FC サブプログラムをコールします。FC サブプログラムが実行されている時はこの処理が終了するまでの間、親プログラムの実行は待たされます。FC サブプログラムは番号と名前（ファンクションやファンクションブロック同様）によって識別されます。以下に FC サブプログラムを読み出すサブプログラムシンボルを示します。



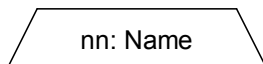
同一のチャート内で2つの異なるサブプログラムは同一の番号を持つことはできません。FC プログラムの階層構造では以下のルールが用いられます。

- FC プログラムで親プログラムを持たない FC プログラムはメイン FC プログラムと呼ばれます。
- メイン FC プログラムはアプリケーションがスタートした時に ISaGRAF のシステムが起動します。
- プログラムは複数の FC サブプログラムを持つことができます。
- FC サブプログラムは1つの親プログラムしか持つことができません。
- FC サブプログラムは親プログラムからのみコールされます。
- FC サブプログラム間同士でのコールはできません。

同一の FC サブプログラムが親プログラムのチャートの中から複数回にわたり読み出されることもあります。FC サブプログラムのコールは、サブチャート全体の実行を意味します。その間は親プログラムは待たされます。サブプログラムシンボルはチャート内のアクションシンボルと同様にフローリンクの接続ルールが適用されます。

I/O操作アクション

I/O操作アクションは、通常のアクションシンボルと同様に扱われます。識別も同様に番号と名前で行います。I/O操作アクションの目的は、プログラム内での処理を後で読みやすくするためと、チャート内で実行プラットフォームに依存する部分を明確にするためにあります。I/O操作アクションを使うかどうかは任意です。以下に FC I/O操作ブロックの例を示します。



I/O操作アクションは通常のアクションと同じふるまいをします。その属性、プログラム言語、フローの接続についても同じです。

注意: I/O操作アクションでは実行プラットフォームに依存する記述をするケースがあるために、このような名称がつけられています。特に、このブロックでI/O処理を行うという意味ではありません。

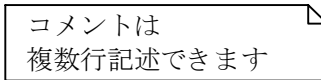
コネクタ

コネクタはチャート内の2つのポイントを直接フローリンクで接続する代わりに扱えるリンクです。コネクタは円形で表示されフローの始点から接続されます。コネクタには、横(場所はフローの方向によって変わりますが)にリンク先の識別名を記入します。リンクの先はチャート内のオブジェクトの番号と名前によって指定します。以下にコネクタの例を示します。



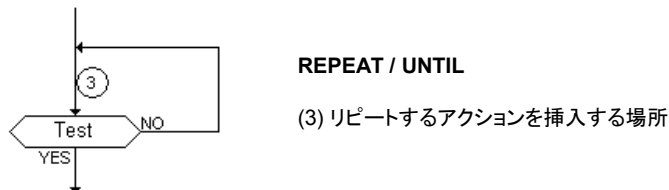
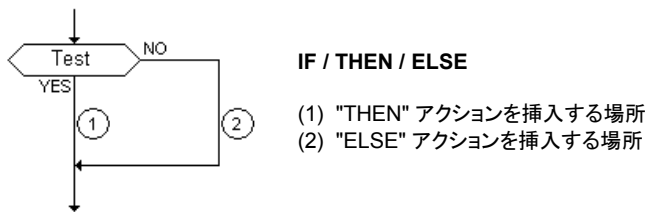
コメント

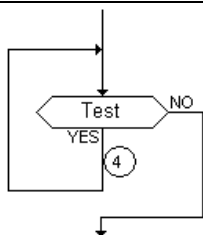
コメントシンボルはフローチャートの実行とはまったく関係がありません。チャート内の空白の場所に配置することができます。プログラムの解説に使うことができます。以下にコメントシンボルの例を示します。



B.4.2 FC 組み合わせ構造

ここではフローチャートの**組み合わせ構造**の例を示します。これらは基本オブジェクト(テスト、アクション、リンク)の組み合わせから成り立っています。





WHILE / DO

(4) リピートするアクションを挿入する場所

B.4.3 FC の実行時の原則

フローチャートのターゲット上での**実行時の原則**は以下のように説明できます。

- Begin シンボルではターゲットの1サイクルを使います。
- End シンボルではターゲットの1サイクルを使い、チャートの実行が終了します。このシンボルに到達した後はいかなるアクションも実行されません。
- 同一サイクル中に既に処理したアクションやテストに到達すると、フローは停止し、続きは次のサイクルで処理されます。

注意: SFC が状態遷移を表現するのに対して、FC では状態が一定とは限りません。デバッグでアクションが強調表示されている間は、繰り返しの処理は行われていません。

B.4.4 FC の文法チェック

レベル2プログラムを記述している ST,LD、IL プログラムの文法チェックとは別に、以下のようなフローチャート自体の**文法チェック**が存在します。

主な文法チェック:

- 全てのシンボルにおける接続ポイントは必ずフローリンクで接続されていなければならない。(但し、“End”シンボルのみ接続を省略することができます。)
- 他のシンボルと独立したシンボルやチャートは存在することはできません。
- コネクタシンボルでの接続先に存在中の正しいオブジェクトが指定されていなければならない。

その他のルール:

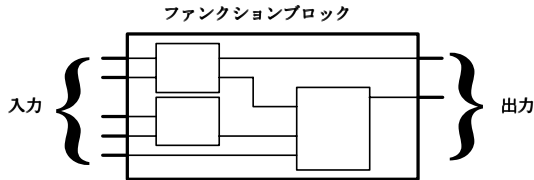
- 空のアクション(レベル2プログラムが存在しないもの)は実行時には何も処理が行われません。
- 空のテスト(レベル2プログラムが存在しないもの)は実行時には常に TRUE とみなされます。

B.5 FBD言語

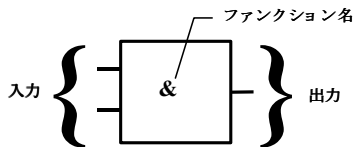
ファンクションブロックダイアグラム(FBD)はグラフィカルな言語です。ISaGRAFライブラリに定義済みのファンクション同士を**接続**して複雑な手続きをグラフィカルなエディタ上でプログラムすることができます。

B.5.1 FBDダイアグラムのフォーマット

FBDダイアグラムは**入力変数**と**出力変数**との間のファンクションといえます。このファンクションは**基本のファンクションブロック**が複数集まったセットのような形となります。入力や出力はブロックに**接続線**で接続されます。ブロックからの出力は更に別の入力の入力に接続されることがあります。



ファンクションブロックダイアグラム全体はISaGRAFライブラリに登録済みの**基本ファンクションブロック**を組み合わせて作られています。各ファンクションブロックは固定数の**入力接続ポイント**及び**出力接続ポイント**を持っています。ファンクションブロックは一つの長方形で表現されます。入力は**左側**の境界線に接続され、出力は**右側**の境界線に接続されます。一つの基本ファンクションブロックはこのブロックに対する入力、出力間で一つのファンクションを実現します。基本のファンクションブロックの処理(ファンクションブロック名)は長方形の中にシンボルとして書かれます。各入力、出力の**変数タイプ**は別に定義されています。



FBDプログラムへの入力はファンクションブロックの入力接続ポイントに接続されなければなりません。各入力変数のタイプは各入力で定義されたタイプと一致していなければなりません。FBDダイアグラムの入力は**定数**、**内部変数**、**外部入力変数**、**外部出力変数**のいずれでも構いません。

FBDプログラムの出力はファンクションブロックの出力接続ポイントに接続されなければなりません。各出力変数のタイプは各出力で定義されたタイプと一致していなければなりません。FBDダイアグラムの出力は**内部変数**、**外部出力変数**、プログラム名(編集集中のプログラムが**サブプログラム**である場合のみ)のいずれでも構いません。出力が現在編集集中であるサブプログラム名である場合は、プログラムの戻り値(出力パラメータ)への割り付けとなり、この出力値が親プログラムに戻されます。

入力変数とファンクションブロックの入力接続ポイント、出力変数とファンクションブロックの出力接続ポイントは**接続ライン**で接続されます。

接続には以下の3通りがあります。

- 入力変数からファンクションブロックの入力ポイントに接続
- ファンクションブロックの出力ポイントから別のファンクションブロックの入力ポイントに接続
- ファンクションブロックの出力ポイントから出力変数に接続

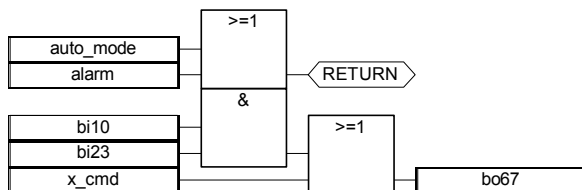
接続には**方向性**があります。即ち、データを左から右へ転送します。接続されるポイントの変数タイプは同じである必要があります。

右側から複数の接続線を引き出し、同じデータを複数のオブジェクトに転送することもできます。

B.5.2 RETURN ステートメント

“<RETURN>”というキーワードをダイアグラムの出力として使うことがあります。RETURNはファンクションブロックの**ブール型の出力**に接続されなければなりません。RETURNステートメントはプログラムの**条件による終了**を表わし、ブロックのブール型出力が**TRUE**の時はRETURN以降の部分の実行を行わずにFBDダイアグラムを抜け出します。

(* RETURN ステートメント を使ったFBDプログラムの例 *)



(* 等価なST言語: *)

```
If auto_mode OR alarm Then
    Return;
End_if;
bo67 := (bi10 AND bi23) OR x_cmd;
```

B.5.3 ジャンプとラベル

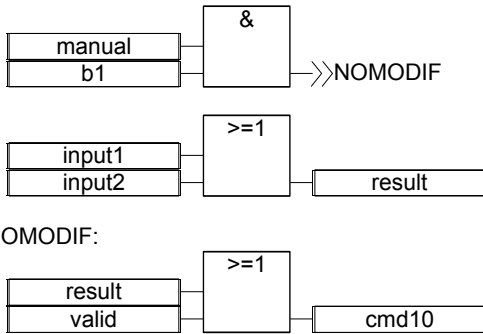
ダイアグラムの実行をコントロールするためにはラベルとジャンプを使います。ジャンプやラベルのシンボルの右側には何も接続されません。以下の表記方法が使われます。

>>LAB ラベルへのジャンプ (ラベル名は "LAB")

LAB: ラベル (ラベル名は "LAB")

もし、ジャンプシンボルの**左側**の接続ラインのブール状態が**TRUE**の時、対応したラベルにジャンプします。

(* ラベルとジャンプを使ったFBDプログラムの例 *)



(* 等価なIL言語: *)

```

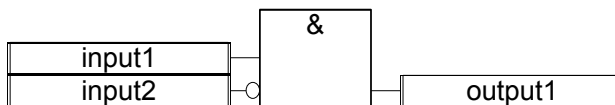
ld      manual
and     b1
jmpc    NOMODIF
ld      input1
or      input2
st      result
NOMODIF:
ld      result
and     valid
st      cmd10

```

B.5.4 論理の反転

ファンクションブロックの入力ポイントに接続されるときに**論理を反転**して接続することができます。この場合のシンボルとして入力ポイントに小さい円が表示されます。論理の反転(Boolean Negation)が使われるときは入力ポイントのタイプは**ブール型**である必要があります。

(* FBDにおける論理反転の例 *)



(* 等価なST言語: *)

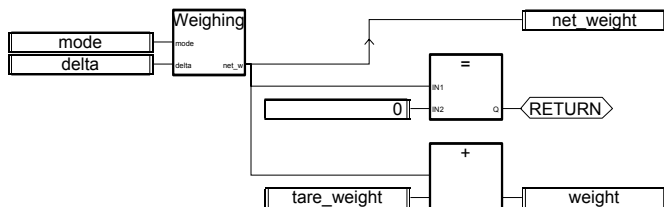
```
output1 := input1 AND NOT (input2);
```

B.5.5 FBDからのファンクション、ファンクションブロックのコール

FBDダイアグラムからサブプログラム、ファンクションやファンクションブロックをコールすることができます。サブプログラム、ファンクション及びファンクションブロックはファンクションボックスとして表記されます。ボックス内の名前はサブプログラム、ファンクション及びファンクションブロック名となります。サブプログラムとファンクションの場合、ファンクションの戻り値(出

カパラメータ)はファンクションボックスの1個のみの出力となります。ファンクションブロックは、複数の出力を持つことができます。

(* サブプログラムのブロックをコールする例 *)



(* 等価なST言語 *)

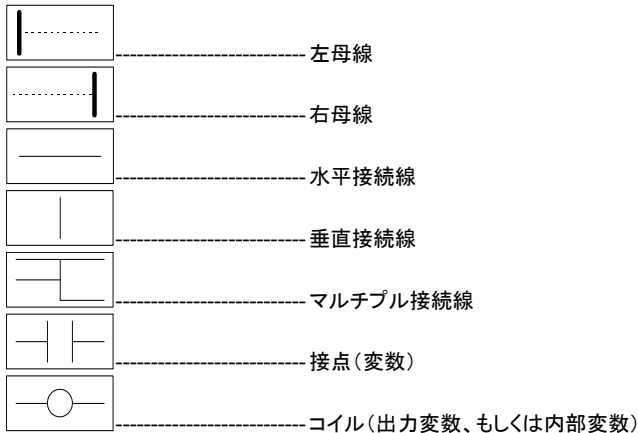
net_weight := Weighing (mode, delta); (* call sub-program *)

If (net_weight = 0) Then Return; End_if;

weight := net_weight + tare_weight;

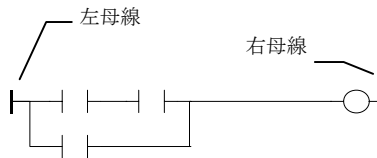
B.6 LD言語

ラダーダイアグラム(LD)はブール演算をグラフィックで表現します。入力(接点、あるいはコンタクト)と出力(コイル)の組み合わせから成り立っています。LD言語ではプログラムチャート上にグラフィックシンボルを配置することブール型データの評価や書き換えが可能です。LD言語は電気リレー回路と同様に扱うことができます。LDダイアグラムは左右の垂直な母線に接続する必要があります。下図はLDダイアグラムの基本的なコンポーネントです。

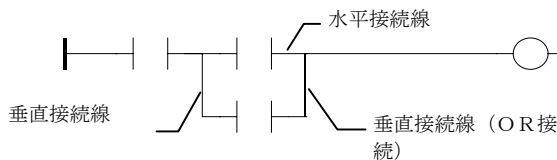


B.6.1 母線と接続線

LDダイアグラムは左右の垂直なライン、即ち左右の母線に接続する必要があります。



LDダイアグラムでのグラフィックシンボルは接続線で母線または別のシンボルに接続されます。接続線は垂直又は水平なラインです。



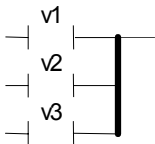
各々のラインセグメントはブール状態の **FALSE** または **TRUE** の値を持ちます。この状態は直接接続されているセグメントで同じ値になります。垂直な**左母線**に接続された水平なラインは **TRUE** 状態を持ちます。

B.6.2 マルチプル接続(並列接続)

1本の水平接続線の左右の端は同じブール状態になります。水平接続線と垂直接続線を組み合わせることで複数のラインをまとめることで**マルチプル接続**が可能になります。マルチ接続の端点のブール状態は以下のルールに従います。

複数の水平接続線が垂直接続線の**左側にマルチプル接続**されていて、**右側に1本の**水平接続線が接続されている場合は、右側のブール状態は左側の複数のラインの**論理和**になります。

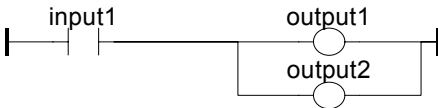
(* 左側にマルチプル接続する例 *)



(* 右側の状態は (v1 OR v2 OR v3) *)

右側でマルチプル接続する場合は、即ち、**左側の1本の**水平接続線が**複数の**接続線になって**右側に**接続される場合は左側のブール状態が右側に伝達されます。

(* 右側マルチプルライン *)

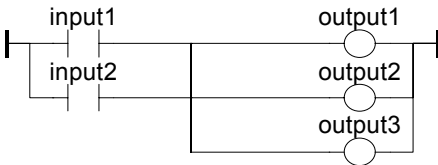


(* 等価なST言語 *)

```
output1 := input1;
output2 := input1;
```

左右のマルチプル接続線の場合、即ち、左右とも複数の水平接続線が接続される場合は右側のブール状態は左側の複数のラインの**論理和**となります。

(* 左右マルチプルラインの例 *)



(* 等価なST言語 *)

```
output1 := input1 OR input2;
output2 := input1 OR input2;
output3 := input1 OR input2;
```

B.6.3 LDの基本的な接点とコイル

入力接点(コンタクト)にはいくつかの種類があります。

- a接点
- b接点
- 立ち上がり接点、立ち下がり接点

出力コイルにもいくつかの種類があります。

- コイル
- 反転コイル
- セットコイル
- リセットコイル
- 立ち上がり検出コイル、立ち下がり検出コイル

変数名はこれらのグラフィックシンボルの上に書かれます。

□ a接点

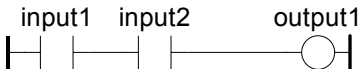
a接点は**ブール型変数**と**接続線**との間での**ブール演算**を行ないます。

ブール型変数



接点の右側の接続線の状態は、左側の**接続線**と接点に割り当てられた**変数**との**論理積**になります。

(* a接点の例 *)



(* 等価なST言語 *)

```
output1 := input1 AND input2;
```

□ b接点

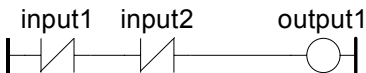
b接点は**接続線**と**ブール型変数の反転**とのブール演算を行ないます。

ブール型変数



コンタクトの右側の接続線の状態はコンタクトの左側の接続線とコンタクトに割り当てられた変数の**反転**との**論理積**になります。

(* b接点の例 *)

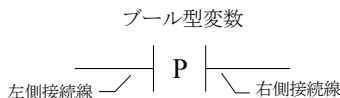


(* 等価なST言語 *)

output1 := NOT (input1) AND NOT (input2);

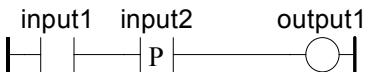
□ 立ち上がり接点

立ち上がり接点は**接続線**と、ブール型**変数**の立ち上がりとの**ブール演算**を行ないます。



接点の右側接続線の状態は、左側接続線がTRUEで、かつブール型変数がFALSEからTRUEに**立ち上がったときのみTRUE**になります。これ以外では**FALSE**となります。

(* 立ち上がり接点の例 *)

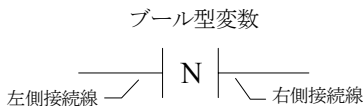


(* 等価なST言語 *)

output1 := input1 AND (input2 AND NOT (input2prev));
(* input2prev is the value of input2 at the previous cycle *)

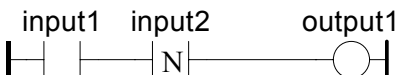
□ 立ち下がり接点

立ち下がり接点は**接続線**と、ブール型**変数**の立ち下がりとの**ブール演算**を行ないます。



接点の右側接続線の状態は、左側接続線が**TRUE**でかつブール型変数がTRUEからFALSEに**立ち下がったときのみTRUE**になります。それ以外では**FALSE**となります。

(* 立ち下がり接点の例 *)



(* 等価なST言語 *)

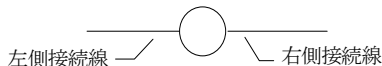
```
output1 := input1 AND (NOT (input2) AND input2prev);
```

(* input2prev is the value of input2 at the previous cycle *)

コイル

コイルは**接続線**のブール状態を**ブール出力**します。

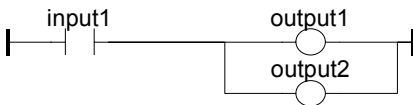
ブール型変数



左側接続線の状態がコイルに割り当てられた変数に代入されます。左側接続線の状態はそのまま右側接続線に伝えられます。右側接続線は通常右母線に接続されます。

コイルに割り当てられるブール変数は**内部変数**か外部**出力変数**である必要があります。変数名はプログラム名(ただし、**サブプログラムのみ**)も使えます。この場合は変数名はサブプログラムの出力パラメータに相当します。

(* コイルの例 *)



(* 等価なST言語 *)

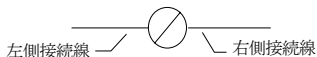
```
output1 := input1;
```

```
output2 := input1;
```

反転コイル

反転コイルは**接続線**のブール状態の**反転**を**ブール出力**します。

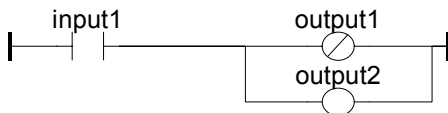
ブール型変数



左側接続線の反転状態がコイルに割り当てられた変数に代入されます。左側接続線の反転状態はそのまま右側接続線に伝えられます。右側接続線は通常右母線に接続されます。

コイルに割り当てられるブール変数は**内部変数**か外部**出力変数**である必要があります。変数名はプログラム名(ただし、**サブプログラムのみ**)も使えます。この場合は変数名はサブプログラムの出力パラメータに相当します。

(* 反転コイルの例 *)



(* 等価なST言語 *)

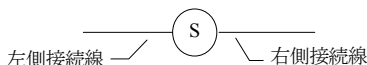
output1 := NOT (input1);

output2 := input1;

セットコイル

セットコイルは接続線のブール状態を**ブール出力**します。

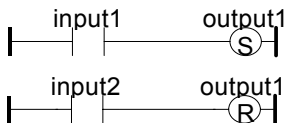
ブール型変数



割り当てられた変数の値は**左側接続線の状態**がTRUEになったときに**TRUEにセット**されます。出力変数はこの状態がリセットコイルで反転されるまで継続します。左側接続線の状態はそのまま右側接続線に伝えられます。右側接続線は通常、右母線に接続されます。

コイルに割り当てられるブール変数は**内部変数**か外部**出力変数**である必要があります。

(* セット、リセットコイルの例 *)



(* 等価なST言語 *)

IF input1 THEN

output1 := TRUE;

END_IF;

IF input2 THEN

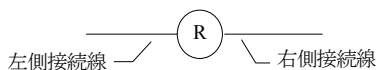
output1 := FALSE;

END_IF;

リセットコイル

リセットコイルは接続線のブール状態をブール出力します。

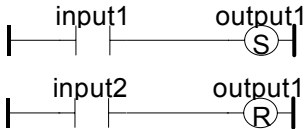
ブール型変数



割り当てられた変数の値は**左側接続線の状態**がTRUEになったときに**FALSEにリセット**されます。出力変数はこの状態がセットコイルで反転されるまで継続します。左側接続線の状態はそのまま右側接続線に伝えられます。右側接続線は通常、右母線に接続されます。

コイルに割り当てられるブール変数は**内部変数**か外部**出力変数**である必要があります。

(* セット、リセットコイルの例 *)

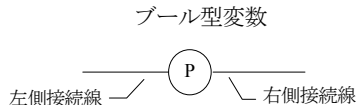


(* 等価なST言語 *)

```
IF input1 THEN
  output1 := TRUE;
END_IF;
IF input2 THEN
  output1 := FALSE;
END_IF;
```

⇨ 立ち上がり検出コイル

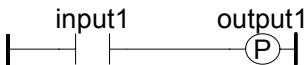
立ち上がり検出コイルは**接続線のブール出力**を行います。このコイルは Quick LD エディタを使用した場合のみ有効です。



割り当てられた変数は**左側接続線の状態**が FALSE からTRUEに立ち上がったときに**TRUEにセット**されます。それ以外の場合はすべて FALSE にリセットされます。左側接続線の状態はそのまま右側接続線に伝えられます。右側接続線は通常、右母線に接続されます。

コイルに割り当てられるブール変数は**内部変数**か外部**出力変数**である必要があります。

(* 立ち上がり検出コイルの例 *)



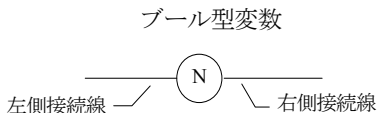
(* 等価なST言語 *)

```
IF (input1 and NOT(input1prev)) THEN
  output1 := TRUE;
ELSE
  output1 := FALSE;
END_IF;
```

(* input1prev は input1 の前回のサイクルの時の状態です *)

□ 立ち下がり検出コイル

立ち下がり検出コイルは**左側接続線のブール出力**を行います。このコイルは Quick LD エディタを使用した場合のみ有効です。



割り当てられた変数は**左側接続線の状態**が TRUE から FALSE に立ち下がったときに **TRUE** にセットされます。それ以外の場合はすべて FALSE にリセットされます。左側接続線の状態はそのまま右側接続線に伝えられます。右側接続線は通常、右母線に接続されます。

コイルに割り当てられるブール変数は**内部変数**か外部**出力変数**である必要があります。

(* 立ち下がり検出コイル*)



(* 等価なST言語 *)

```
IF (NOT(input1) and input1prev) THEN
```

```
    output1 := TRUE;
```

```
ELSE
```

```
    output1 := FALSE;
```

```
END_IF;
```

(* input1prev は input1 の前回のサイクルの状態です。*)

B.6.4 RETURN ステートメント

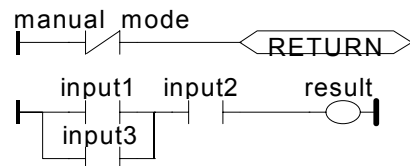
RETURN ラベルはプログラムの条件終了に使われます。RETURN の右側には何も接続されません。



RETURN ラベルの**左側接続線**のブール状態が **TRUE** の時、プログラムは以降のプログラムを実行しないでリターンします。

注意: LDプログラムがサブプログラムの時はプログラム名と同名のコイルに戻り値をセットして親プログラムにリターンする必要があります。

(* RETURN シンボルの例 *)



(* 等価なST言語 *)
 If Not (manual_mode) Then RETURN; End_if;
 result := (input1 OR input3) AND input2;

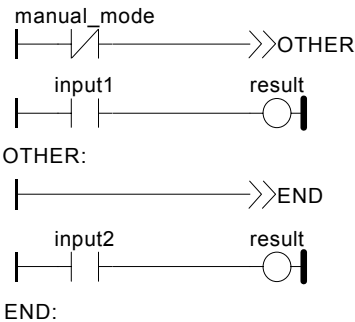
B.6.5 ジャンプ とラベル

ラベルや条件付き(条件無し)ジャンプを使ってダイアグラムの実行を制御できます。ラベルやジャンプシンボルには右側の接続線はありません。

>>LAB ラベル名"LAB"へジャンプ
 LAB: ラベル"LAB"

ジャンプシンボルの左側接続線のブール状態が **TRUE** の時、プログラムは対応したラベルシンボルへジャンプして、その後から実行されます。

(* ジャンプとラベルの例*)



(* 等価なIL言語 *)

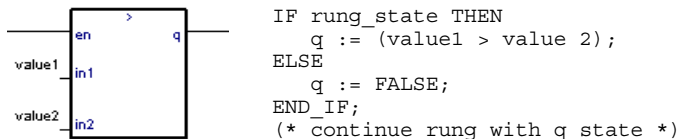
ldn	manual_mode
jmpc	other
ld	input1
st	result
jmp	END
OTHER:	ld input2
	st result
END:	(* end of program *)

B.6.6 LDの中でのブロック

QuickLDエディタを使うことにより、ブロックをラインで接続することができます。ここでのブロックとは、演算子、ファンクションブロック、ファンクションのことを指します。全てのブロックが必ずしもブール型入力、出力を持っているとは限らないために、新たにEN、ENOパラメータを入出力インタフェースに追加します。ただし、FBD/LDエディタでは必要な型の変数を直接接続できるため、EN、ENOは追加されません。

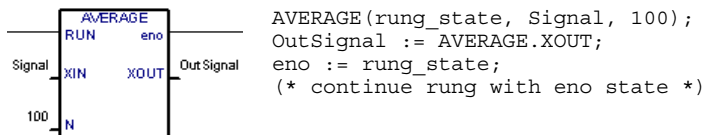
□ "EN" 入力

1番目の入力パラメータのタイプがブール型でない演算子、ファンクションブロック、ファンクションがあります。最初のものは必ずラングに接続する必要があるため、自動的に**EN**という入力パラメータが挿入されます。この**EN**が TRUE の時のみ、ブロックが実行されます。以下に比較命令の例とこれに等価なST言語表現を示します。



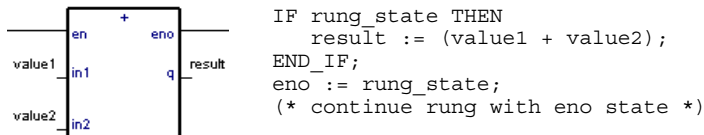
□ "ENO" 出力

1番目の出力パラメータのタイプがブール型でない演算子、ファンクションブロック、ファンクションがあります。最初のものは必ずラングに接続する必要があるため、自動的に**ENO**という出力パラメータが挿入されます。この**ENO**は常に1番目の入力パラメータの値を保持します。以下に AVERAGE ファンクションブロックの例とこれに等価なST言語表現を示します。



□ "EN"、"ENO" パラメータ

場合によっては、**EN**、**ENO**の両方が必要な場合があります。以下に、算術演算の例とこれに等価なST言語表現を示します。



B.6.7 ラダーにおける"In Line"ファンクションブロック

"In Line" ファンクションブロックは他のファンクションブロックと同様に IEC や ISaGRAF での実現方法によって定義されます。:

- "In Line" ファンクションブロックには入力と出力パラメータがあります。それぞれのパラメータは1つのデータ型を持ちます。入力と出力パラメータの合計数は最大32まで可能です。
- "In Line" ファンクションブロックはローカル変数(辞書で定義されたもの)を持ち、ダイアグラムで使用されるとその都度インスタンス化されます。

"In Line" ファンクションブロックの主な特徴はアプリケーションで使用される毎にその実態(コード)はインスタンス化されるということです。インスタンスの呼び出しは実際のコードに置き換えられます。この置き換えは Quick LD コンパイラによって行われます。インスタンス化のメカニズムはLDコンパイラによって行われる為、"In Line"ファンクションブロックのネストには制限はありません。従って、"In Line"ファンクションブロックは他の"In Line"ファンクションブロックや標準的なCファンクションブロックを呼び出すことが出来ます。

仕組み:

入力変数、出力変数、内部変数を各1つずつ持ったファンクションブロックを作成するとします。この例は edge detection ファンクションブロックです。:

名称:

FB1

入力:

IN (boolean) = 入力信号

出力:

Q (boolean) = IN が FALSE から TRUE に変化した時のみ TRUE にセットされる

内部インスタンスデータ(呼び出される毎にインスタンス化される):

PREV (boolean) = 前の周期での IN の状態。

FB1 ブロックのLDプログラム:

```

      IN      PREV      Q
|----] [-----] \ [----- ( ) -|
      IN      PREV
|----] [----- ( ) -|

```

ST 言語での表現:

```

Q := IN and not PREV;
PREV := IN;

```

以下はこのファンクションブロックをコールするクイック LD プログラムです。:

```

      B1      +-----+      B2
|----] [--|  FB1  |----- ( ) -|
      +-----+
      B3      +-----+      B4
|----] [--|  FB1  |----- ( ) -|
      +-----+

```

"FB1" は入力や出力への接続変数によってインスタンス化されます。以下のST言語は上記のプログラムをコンパイルした時に生成されるコードと同等のものです。:

```
(* code of the first call *)  
B2 := B1 and not PREV1;  
PREV1 := B1;  
(* code of the second call *)  
B4 := B3 and not PREV2;  
PREV2 := B3;
```

この例では、コンパイラはそれぞれのFBのコールに関して、内部変数を生成しなければならないことを示しました。(この例では PREV1 や PREV2)。この名前の割り当ては自動的に行われるので、"In Line"ファンクションブロックのネストには制限はありません。

標準のファンクションブロックのかわりにこの"In Line" ファンクションブロックを使用するとアプリケーションコードのサイズ(TICコードのサイズ)は増加します。また関数呼び出しやパラメータ渡しが必要でないので、実行時間の短縮につながります。

"In Line" ファンクションブロックはまた edge detection contacts やコイル(P and N)、Cや標準ファンクションブロック、ローカルジャンプ、ラベルもサポートしています。.

Limitations:

"In Line" ファンクションブロックは Quick LD でのみ利用可能です。:

- "In Line" ファンクションブロックは Quick LD で記述されなければならない。
- "In Line" ファンクションブロックをコールする全てのプログラムは Quick LD で記述されなければならない。
- SFCやFC上での Quick LD は "In Line" ファンクションブロックをコールすることが出来る。

以下はFBをコールする時の制限です。:

- "In Line" ファンクションブロックは 他の"In Line" ファンクションブロックをコールすることが出来る。
 - "normal" ファンクションブロックは "In Line" ファンクションブロックをコールすることが出来ない。
 - "In Line" ファンクションブロックは ファンクションブロックをコールすることが出来ない。
 - 再帰呼び出しは出来ません。
- ("normal" とは ISaGRAF 3.3 でインプリメントされているようなFBを指す。)

誤りがあった場合はコンパイル時に発見されます。またコンパイラは、"In Line"ファンクションブロックが"Verify" や "Make"コマンドの起動によりコンパイルされることを自動的に保証します。

"In Line" ファンクションブロックは1つのプロジェクト内でのみ定義されます。ISaGRAF のライブラリはこのファンクションブロックをサポートしていません。

B.7 ST言語

構造化テキスト(ST)はオートメーションプロセス用に設計されたハイレベルの構造化言語です。この言語は主にグラフィック言語で十分に表現しにくい手続きを表わすために使われます。STはSFC言語のトランジションやステップ内のアクションを記述するデフォルトの言語です。

B.7.1 STの主な文法

STプログラムはSTステートメントのリストになります。各々のステートメントの最後にはセミコロンの(:)による区切りが必要です。ソースコード中の変数は**処理を伴わないセパレータ**(スペース、タブなど)や**処理を伴うセパレータ**(:=、<、>など)で区切られています。コメントはテキスト中の任意の位置に(*, *)で囲んで挿入できます。

基本のSTステートメントは以下のものがあります。

- **代入**(変数:= 式;)
- **サブプログラム**や**ファンクション**のコール
- **ファンクションブロック**のコール
- **選択**ステートメント (IF, THEN, ELSE, CASE...)
- **繰り返し**ステートメント (FOR, WHILE, REPEAT...)
- **コントロール**ステートメント (RETURN, EXIT...)
- **SFC**等の他の言語とのリンク用の特殊ステートメント

処理を伴わないセパレータは、処理を伴うセパレータ、定数、識別子の間であれば自由に挿入できます。処理を伴わないセパレータとは**スペース**(空白)、**タブ**、**改行**です。ILのような行単位の書式の言語とは異なり、プログラムの任意の位置で改行できます。以下のルールを守ってこれらのセパレータを使用することで読みやすいSTプログラムを作ることができます。

- 1行に複数のステートメントを書かないようにします。
- 複雑なステートメントの集まりはタブを使ってインデント(字下げ)します。
- コメントをできる限り挿入します。

ソースコードの読みやすさの例

読みにくい	読みやすい
<pre>imax := max_ite; cond := X12; if not(cond (* alarm *)) then return; end_if; for i (* index *) := 1 to max_ite do if i <> 2 then Spcall(); end_if; end_for; (* no effect if alarm *)</pre>	<pre>(* imax : number of iterations *) (* i: FOR statement index *) (* cond: process validity *) imax := max_ite; cond := X12; if not (cond) then return; end_if; (* process loop *) for i := 1 to max_ite do if i <> 2 then Spcall (); end_if; end_for;</pre>

B.7.2 式と括弧

STの式はST演算子と、変数や定数のオペランドの組み合わせになります。一つの式(1個のST演算子と複数のオペランドの組)に対して、オペランドの**タイプ**は同じである必要があります。式とそのオペランドは同じタイプを持ち、式同士を組み合わせるとより複雑な式の記述を行ないます。例えば、

(boo_var1 AND boo_var2)	ブール型
not (boo_var1)	ブール型
(sin (3.14) + 0.72)	実数型
(t#1s23 + 1.78)	これは不正な式です。

括弧は式の一部分を分離させ、演算の優先順位を高くします。括弧が存在しない式では演算順序はST演算子間での既定の**優先順位**に従います。例えば、

2 + 3 * 6	これは 2+18=20	乗算は加算よりも優先順位が高い。
(2+3) * 6	これは 5*6=30	括弧が優先されます

注意: 一つのST式内に最大**8**レベルまでの括弧が認められています。

B.7.3 ファンクションやファンクションブロックのコール

標準的なSTによるファンクションのコールで以下のものを呼び出すことができます。

- ・ サブプログラム
- ・ ライブラリのIEC言語で書かれたファンクションやファンクションブロック
- ・ C言語ファンクション、C言語ファンクションブロック
- ・ 変換関数

≡ サブプログラムやファンクションのコール

名前: サブプログラム名
またはIEC言語やC言語で書かれたライブラリのファンクション

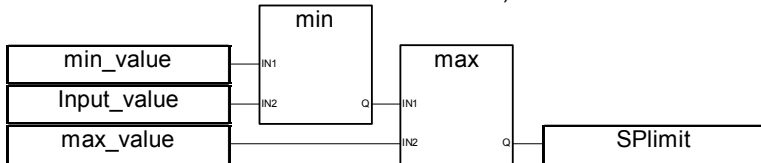
意味:	ST, IL, LD, FBDで書かれたサブプログラム、ファンクションやC言語を呼び出し、戻り値を得る
文法:	<変数> := <サブプログラム> (<パラメータ 1>, ... <パラメータ N>);
オペランド:	戻り値や引数のタイプはサブプログラムのパラメータの定義に従う必要があります。
戻り値:	サブプログラムの戻り値

サブプログラムコールはどのような式の内部でもコールできます。SFCプログラムのトランジション内部でも使えます。

例1: サブプログラムのコール

```
(* メインの ST プログラム *)
(* サブプログラムから整数型値を得てタイマ型値に変換 *)
ana_timeprog := SPLimit ( tprog_cmd );
appl_timer := tmr (ana_timeprog * 100);

(* 呼び出されるFBDサブプログラム、名前は 'SPLimit' *)
```



例2: ファンクションのコール

```
(* 複数の標準Cファンクションを式の内部で使う例: min, max, right, mlen left *)
limited_value := min (16, max (0, input_value) );
rol_msg := right (message, mlen (message) - 1) + left (message, 1);
```

ファンクションブロックのコール

名前:	ファンクションブロックインスタンス名
意味:	ISaGRAFのライブラリに登録されているファンクションブロックをコールして出力パラメータを使う。
文法:	(* ファンクションブロックのコール*) <ファンクションブロック名> (<p1>, <p2> ...); (* 出力パラメータの取得*) <結果> := <ファンクションブロック名>. <出力パラメータ 1>; ... <結果> := <ファンクションブロック名>. <出力パラメータ N>;
オペランド:	パラメータのタイプはファンクションブロックで定義されているタイプに一致しなければなりません。
戻り値:	文法を参照

ファンクションの意味、パラメータの型、戻り値の型などはISaGRAFライブラリに登録されているものを参照してください。

ファンクションブロックのインスタンス(コピーの名前)は必ず辞書に登録しておく必要があります。

(* ファンクションブロックをコールするSTプログラム例 *)

(* triqb1 : ファンクションブロック R TRIG - 立ち上がり検出*)

If (trigb1.Q) Then nb edge := nb edge + 1; End if;

次のブール演算子は ST 言語特有のものです。

- 他の標準ブール演算子として下記のものが使えます。

- これらの詳細については「標準演算子、ファンクションブロック、ファンクション」のセクションを参照してください。

名前:	REDGE	
意味:	ブール型の式全体の立ち上がり検出	
文法:	<edge> := REDGE (<boo_expression>, <memo_variable>);	
オペランド:	<boo_expression>立ち上がり検出するブール型変数または式。 <memo_variable>直前の変数状態をストアするための内部変数	
戻り値:	FALSE から TRUE へ変化したとき、TRUE それ以外の場合、FALSE	

REDGEを使っての立ち上がり検出は一回のサイクル内に2回以上は検出されません。立ち上がり検出はSFCのトランジション条件に使うことがあります。

注意: <memo_variable>変数は式や変数の直前の状態の保管のために使われるので、他の変数の立ち上がり検出などの目的で同時には使えません。通常、立ち上がり検出をしたい変数名が "xxx" のとき、<memo_variable>としては "EDGE_xxx" を使います。こうすることにより、他の REDGE の操作によって書き換えられることはありません。

例:

```
(* REDGE 演算子を使う ST プログラム *)

(* このプログラムはブール型入力変数の立ちあがりをカウントします *)
(* Bi120 が入力変数です *)
(* Edge_Bi120 が Bi120 の状態を保管します *)
```

```
If REDGE (Bi120, Edge_Bi120) Then
    Counter := Counter + 1;
End_if;
```

注意: この演算子は IEC61131-3 に含まれていません。互換性のためには R_TRIG 標準ブロックのご使用をお勧めします。

☐ **FEDGE 命令**

意味: ブール型変数の立ち下がり検出

文法: `<edge> := FEDGE (<boo_expression>, <memo_variable>);`

オペランド: <boo_expression> 立ち下がり検出するブール型変数
<memo_variable> 直前の変数状態をストアするための内部変数

戻り値: TRUE TRUE から FALSE へ変化したとき
FALSE 上記以外の場合

FEDGE() を使った立ち上がり検出は一回の実行スキャン内に2回以上は検出されません。立ち上がり検出はSFCのトランジション条件に使われる場合があります。

注意: <memo_variable> 変数は式や変数の直前の状態の保管のために使われるので、他の変数の立ち下がり検出などの目的で同時には使えません。
通常、立ち下がり検出をしたい変数名が "xxx" のとき、<memo_variable> としては "EDGE_xxx" を使います。こうすることにより、他の FEDGE 操作によって上書きされたりされることはありません。

例:

```
(* ST program using FEDGE operator *)

(* this program counts the falling edges of a boolean input *)
(* Bi120 is an input boolean variable *)
(* Edge_Bi120 is the memory of the Bi120 variable state *)
```

```
If FEDGE (Bi120, Edge_Bi120) Then
    Counter := Counter + 1;
End_if;
```

注意: この命令は IEC61131-3 に含まれていません。互換性のためには F_TRIG 標準ブロックのご使用をお勧めします。

B.7.5 ST基本ステートメント

ST言語の基本ステートメントには以下のものがあります。

- 代入
- RETURN ステートメント
- IF-THEN-ELSIF-ELSE 構造
- CASE ステートメント
- WHILE 繰り返しステートメント
- REPEAT 繰り返しステートメント
- FOR 繰り返しステートメント
- EXIT ステートメント

⇐ 代入

名前: :=
意味: 式を変数に代入
文法: <変数> := <任意の式>;
オペランド: 変数は内部、あるいは出力変数である必要があります。
 変数と式のタイプが一致している必要があります。

<任意の式>はサブプログラムやISaGRAFライブラリのファンクションへのコールでも構いません。

例:

(* 代入を使ったSTプログラム *)

(* 変数 ← 変数*)

```
bo23 := bo10;
```

(* 変数 ← 式*)

```
bo56 := bx34 OR alrm100 & (level >= over_value);
```

```
result := (100 * input_value) / scale;
```

(* サブプログラムの戻り値を代入 *)

```
rc := PSelect ( );
```

(* ファンクションコールの結果を代入 *)

```
limited_value := min (16, max (0, input_value) );
```

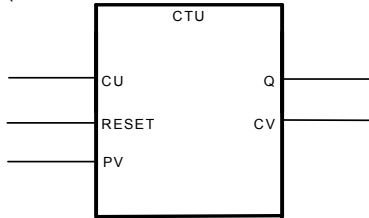
⇐ RETURN ステートメント

名前: RETURN
意味: 現在実行のプログラムを終了する
文法: RETURN ;
オペランド: (なし)

SFCのアクションブロックでは、RETURN ステートメント はそのアクションブロックのみの終了を意味します。

例:

(* FBD で記述したプログラマブルカウンタの仕様*)



(* このプログラムの ST 言語での実装。RETURN ステートメントを使用。*)

```

If not (CU) then
    Q := false;
    CV := 0;
    RETURN; (* プログラムの終了 *)
end_if;

if R then
    CV := 0;
else
    if (CV < PV) then
        CV := CV + 1;
    end_if;
end_if;
Q := (CV >= PV);

```

IF-THEN-ELSE ステートメント

名前: IF ... THEN ... ELSIF ... THEN ... ELSE ... END_IF

意味: 2つのSTステートメントの一方を実行します。ブール式の値によって選択し実行します。

文法:

```

IF <ブール式> THEN
    <ステートメント>;
    <ステートメント>;
    ...
ELSIF <ブール式> THEN
    <ステートメント>;
    <ステートメント>;
    ...
ELSE
    <ステートメント>;
    <ステートメント>;
    ...
END_IF;

```

ELSE や ELSIF ステートメントはオプションです。もし、ELSE ステートメントが書かれていない場合は、条件が FALSE の時は何も実行されません。

例:

(* IF ステートメントを使ったSTプログラム *)

```
IF manual AND not (alarm) THEN
    level := manual_level;
    bx126 := bi12 OR bi45;
ELSIF over_mode THEN
    level := max_level;
ELSE
    level := (lv16 * 100) / scale;
END_IF;
```

(* ELSE のない IF 構文*)

```
If overflow THEN
    alarm_level := true;
END_IF;
```

⇨ **CASE ステートメント**

名前: CASE ... OF ... ELSE ... END_CASE

意味: 複数のSTステートメントの中から整数式の内容によってステートメントを選択し実行します

文法: CASE <製数式> OF

<値> : <ステートメント> ;

<値> , <値> : <ステートメント> ;

...

ELSE

<ステートメント> ;

END_CASE;

<値>は必ず整数定数でなければなりません。複数の<値>をコンマで区切って並べている場合は同じステートメントを実行することを意味します。ELSE ステートメントはオプションです。

例:

(* CASE ステートメントを使ったSTプログラム *)

```
CASE error_code OF
    255:   err_msg := 'Division by zero';
          fatal_error := TRUE;
    1:    err_msg := 'Overflow';
    2, 3: err_msg := 'Bad sign';
ELSE
    err_msg := 'Unknown error';
END_CASE;
```

⇨ **WHILE ステートメント**

名前: WHILE ... DO ... END_WHILE

意味: STステートメントのグループの繰り返し実行を行いません。繰り返しの前に繰り返しの条件の評価を行いません

文法: WHILE <ブール式> DO

```

<ステートメント>;
<ステートメント>;
...
END_WHILE ;

```

注意: ISaGRAFはサイクルタイムに**同期**するシステムなので、入力変数の値はWHILEの繰り返し実行中には変化しません。即ち、定数や入力変数を直接、WHILE繰り返しの条件に使うことはできません。(無限ループができてしまいます。)

例:

(* WHILE ステートメントを使ったSTプログラム *)
 (* C言語ファンクションをシリアルポートからの文字の読み出しに使っています *)

```

string := ""; (* 空の文字列 *)
nbchar := 0;

WHILE ((nbchar < 16) & ComIsReady ( )) DO
    string := string + ComGetChar ( );
    nbchar := nbchar + 1;
END_WHILE;

```

⇐ **REPEAT ステートメント**

名前: REPEAT ... UNTIL ... **END_REPEAT**
意味: STステートメントのグループの繰り返し実行を行ないます。繰り返しの**後**に繰り返しの条件の評価を行ないます
文法: **REPEAT**
 <ステートメント>;
 <ステートメント>;
 ...
 UNTIL <ブール式>
 END_REPEAT ;

注意: ISaGRAFはサイクルタイムに**同期**するシステムなので、入力変数の値はREPEATの繰り返し実行中には変化しません。即ち、定数や入力変数を直接、REPEAT繰り返しの条件に使うことはできません。(無限ループができてしまいます。)

例:

(* REPEAT ステートメントを使ったSTプログラム *)
 (* Cファンクションをシリアルポートからの文字の読み出しに使っています *)

```

string := ""; (* 空の文字列 *)
nbchar := 0;
IF ComIsReady ( ) THEN
    REPEAT
        string := string + ComGetChar ( );
        nbchar := nbchar + 1;
    UNTIL ( (nbchar >= 16) OR NOT (ComIsReady ( )) )
    END_REPEAT;
END_IF;

```

FOR ステートメント

名前:	FOR ... TO ... BY ... DO ... END_FOR
意味:	整数型変数をインデックスとして有限回数分の繰り返し操作を行ないます。
文法:	FOR <index> := <mini> TO <maxi> BY <step> DO <ステートメント>; <ステートメント>; END_FOR;
オペランド:	index: 整数型の内部変数で、毎ループ増加します。 mini: index の初期値 maxi: index の最大値 step: index の毎ループの増加幅

[BY step] ステートメントはオプションです。何も指定が無ければ、STEP は1です。

注意: ISaGRAFはサイクルタイムに同期するシステムなので、入力変数の値はFOR の繰り返し実行中には変化しません。

以下に FOR ステートメントと等価な WHILE ステートメントを示します。

```

index := mini;
while (step <= maxi) do
  <ステートメント>;
  <ステートメント>;
  index := index + step;
end_while;

```

例:

```

(* FOR ステートメントを使ったSTプログラム *)
(* 文字列から数字だけを取り出します *)

```

```

length := mlen (message);
target := ""; (* 空の文字列 *)
FOR index := 1 TO length BY 1 DO
  code := ascii (message, index);
  IF (code >= 48) & (code <= 57) THEN
    target := target + char (code);
  END_IF;
END_FOR;

```

EXIT ステートメント

名前:	EXIT
意味:	FOR, WHILE, REPEAT の繰り返しステートメントからの抜け出し。
文法:	EXIT;
オペランド:	(なし)

EXIT は通常、IF ステートメントと組み合わせて、FOR、WHILE、REPEAT などのブロック内で使います。

例:

(* EXIT ステートメントを使ったSTプログラム *)
 (* 文字列から指定のキャラクタを検索します *)

```
length := mlen (message);
found := NO;
FOR index := 1 TO length BY 1 DO
  code := ascii (message, index);
  IF (code = searched_char) THEN
    found := YES;
    EXIT;
  END_IF;
END_FOR;
```

B.7.6 STの拡張

以下にST言語の拡張部分について説明します。

● TSTART - TSTOP: タイマコントロール

以下のステートメントやファンクションはチャイルドSFCプログラムをコントロールするために使われます。SFCステップ内のアクションブロック ACTION(): ... END_ACTION;で使えます。

- GSTART SFCプログラムの起動
- GKILL SFCプログラムの停止
- GFREEZE SFCプログラムの凍結
- GRST 凍結されたSFCプログラムの再起動
- GSTATUS SFCプログラムの現在の状態を取得

注意: このファンクションは IEC 61131-3 には含まれていません。IEC61131-3 に準拠で行う場合は GSTART と GKILL の代わりに、次の文法になります。
 child_name(S); (*GSTART(child_name);と等価 *)
 child_name(R); (*GKILL(child_name);と等価 *)

SFCのステップの状態を取得するには以下のものが使えます。

GSnnn.x ステップの活性／非活性状態を表すブール値
GSnnn.t ステップが活性状態になってからの経過時間
 ("nnn" はSFCステップのリファレンス番号です。)

別のSFCプログラムのステップの活性状態は次の表現で得ることができます。

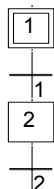
GSnnn(プログラム名).x

注意: 別のプログラムの参照に対しては、この表現は IEC61131-3 には含まれていません。IEC61131-3 に準拠で行う場合はグローバル変数(例:ref_step_X)を辞書で宣言し、これに参照したいステップ状態を代入します。即ち、ステップの中でNアクションクオリファイアを使ってアクションを記述します (ref_step_X(N);)。このような状況でステップの状態をテスト(参照)したい場所でこのグローバル変数(ref_step_X)を使います。

ST言語

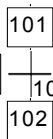
以下に、プログラムを示します。

Prog プログラム



ref_step_X(N);

Prog プログラムのステップ状態を使いたい 別プログラム



ref_step_X; (* = GS2(prog).X *)

TSTART ステートメント

名前: TSTART

意味: タイマ型変数のカウントを開始します。カウントは現在値から開始し、このコマンドで変数値が書き換わることはありません。

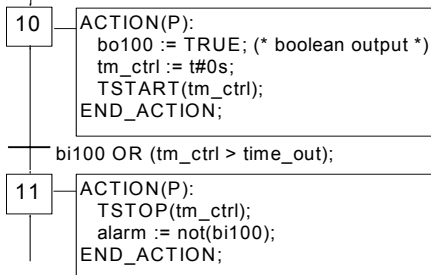
文法: TSTART (<timer_variable>);

オペランド: 非活性状態のタイマ変数

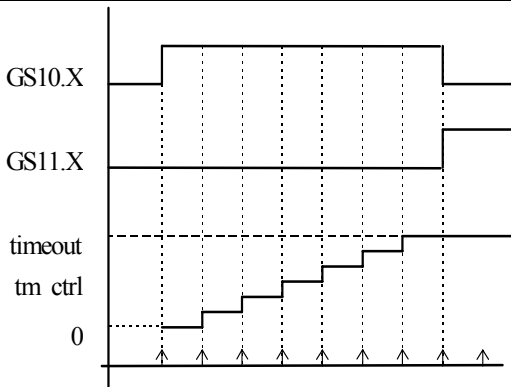
戻り値: (なし)

例:

(* TSTART、TSTOP ステートメントを使ったSTプログラム *)



bi100 が常時 FALSE の時のタイムチャート:



タイマ値は1サイクル中は同じ値を保ちます。

□ **TSTOP ステートメント**

名前: TSTOP
意味: タイマ型変数の更新を停止します。
 このコマンドで変数値が書き換わることはありません。
文法: TSTOP (<timer_variable>);
オペランド: 活性状態のタイマ変数
戻り値: (なし)

例: 上記の TSTART ステートメントを参照願います。

□ **GSTART ステートメント**

名前: GSTART
意味: チャイルドSFCプログラムを各初期ステップにトークンをセットすることにより起動させる
文法: GSTART (<child_program>);
オペランド: 指定するSFCプログラムは、GSTART ステートメントが書かれているSFCプログラムのチャイルドSFCプログラムである必要があります。
戻り値: (なし)

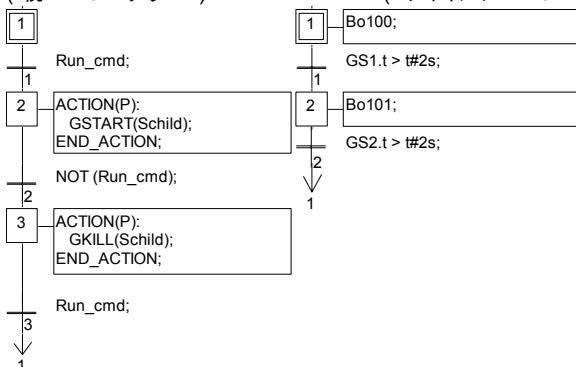
チャイルドSFCプログラムのチャイルド(孫SFCプログラム)は自動的に起動されません。

注意: GSTART は IEC 61131-3 には含まれていません。以下のようにSクオリファイアを使うことでチャイルドプログラムを起動します。
 Child_name(S);

例: GSTART、GKILL ステートメントの例

(*親SFCプログラム *)

(* チャイルドSFCプログラム 'Schild' *)



□ GKILL ステートメント

名前: GKILL

意味: チャイルドSFCプログラムにある実行トークンを取り除くことによりチャイルドSFCプログラムを停止します。

文法: GKILL (<child_program>);

オペランド: 指定するSFCプログラムは、GKILL ステートメントが書かれているSFCプログラムのチャイルドSFCプログラムである必要があります。

戻り値: (なし)

チャイルドSFCプログラムのチャイルド(孫SFCプログラム)も自動的に停止されます。

注意: GKILL は IEC 61131-3 norm に含まれていません。以下のようにRクオリファイアを使ってチャイルド SFC プログラムを停止させます。
Child_name(R);

例: 上記 GSTART ステートメントを参照ねがいます。

□ GFREEZE ステートメント

名前: GFREEZE

意味: チャイルドSFCプログラムを凍結します。後で GRST ステートメントで再起動が可能です。

文法: GFREEZE (<child_program>);

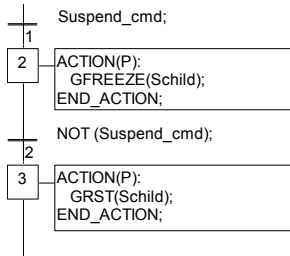
オペランド: 指定するSFCプログラムは、GFREEZE ステートメントが書かれているSFCプログラムのチャイルドSFCプログラムである必要があります。

戻り値: (なし)

チャイルドSFCプログラムのチャイルド(孫SFCプログラム)も自動的に凍結されます。

注意: GFREEZE は IEC 61131-3 には含まれません。

例:



GRST ステートメント

名前: GRST

意味: GFREEZE ステートメントで凍結されていたチャイルドSFCプログラムを再起動します。GFREEZE で取り除かれた実行トークンが戻されます

文法: GRST (<child_program>);

オペランド: 指定するSFCプログラムは、GRST ステートメントが書かれているSFCプログラムのチャイルドSFCプログラムである必要があります。

戻り値: (なし)

凍結されていたチャイルドSFCプログラムのチャイルド(孫SFCプログラム)も自動的に再起動されます。

注意: GRST は IEC 61131-3 には含まれません。

例: 上記 GFREEZE ステートメントを参照ねがいます。

GSTATUS ステートメント

名前: GSTATUS

意味: チャイルドSFCプログラムの現在の状態を戻します

文法: <ana_var> := GSTATUS (<child_program>);

オペランド: 指定するチャイルドSFCプログラムは、GSTATUS ステートメントが書かれているSFCプログラムのチャイルドSFCプログラムである必要があります。

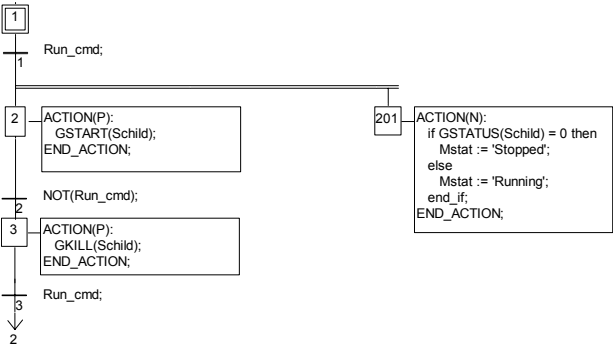
戻り値: 0 = プログラムが非活性状態 (killed)

1 = プログラムが活性状態 (started)

2 = プログラムが凍結状態 (frozen)

注意: GSTATUS は IEC 61131-3 には含まれません。

例:



B.8 IL言語

命令リスト(IL)はローレベルの言語です。小さなアプリケーションやアプリケーションの特定部分の最適化に適しています。命令は常に**現在結果(ILレジスタともいいます)**に関連しています。オペレーションは必ず現在結果の値とオペランドとの間で行なわれます。オペレーションの結果は再び現在結果に格納されます。

B.8.1 ILの主な文法

ILプログラムは**インストラクション**のリストです。各々のインストラクションは1行単位で書き、**オペレータ(命令)**と**修飾子**、さらに必要に応じて一つ以上の**オペランド**から成り立っています。これらの要素はカンマ(,)で区切られています。**ラベル**をインストラクション前に置くこともできます。ラベルにはコロン(:)を付けます。**コメント**を記入する場合は行の最後にかきます。コメントは「(*)で始まり「(*)で終わる必要があります。空白行やコメントだけの行も認められています。以下に、インストラクションの例を示します。

ラベル	オペレータ	オペランド	コメント
Start:	LD	IX1	(* 押しボタン *)
	ANDN	MX5	(* 禁止されていない場合は *)
	ST	QX2	(* モータースタート *)

ラベル

ラベル名の後にはコロン(:)がつきます。インストラクションはこのラベルの後から書かれます。ラベルは空白行に書いてもかまいません。ラベル名はジャンプなどの他のインストラクションでオペランドに使われることがあります。ラベル名は以下のルールに従う必要があります。

- 名前は16文字以内
- 最初の文字は**英字**(a-z)
- それ以降は**英字**、**数字**、**'_'**(アンダースコア)

ラベル名は同一のILプログラムの中では重複して使うことができません。ただし、変数名と同一のラベル名を定義することはできます。

命令修飾子

利用可能な命令修飾子を以下に示します。修飾子文字はオペレータと組み合わせます。修飾子間でスペースは含まれてはなりません。

N	オペランドのブール反転
(遅延操作
C	条件操作

'N' 修飾子 はオペランドのブール反転を意味します。例えば、

ORN IX12

はSTでは以下ようになります。

result := result OR NOT (IX12)

'修飾子 は 'が見つかるまでインストラクションの実行を先送りします。

'C' 修飾子 は現在結果がTRUE(ブール値以外の場合は0でない値)の時のみインストラクションを実行することを意味します。**'C'** 修飾子 は **'N'** 修飾子と合わせて使うこともできます。この場合は現在結果がFALSE(ブール値以外の場合は0)の時のみインストラクションを実行します。

一、延迟操作

ILレジスタ(現在結果)が1つしか無いため、オペレーションを遅延(先送り)しないといけない場合があります。このときインストラクションの実行順序が変更されることになります。

'\	これは修飾子	命令が先送りされること意味します。
'\	これはオペレータ	先送りされていた処理を実行します。

' ' 修飾子を付けたオペレータは、' ' が現れるまで実行を先送りされます。
例えば、

AND(IX12
OR IX35
)

は以下のように解釈されます。

```
result := result AND ( IX12 OR IX35 )
```

B.8.2 ILオペレータ(命令)

以下のテーブルはIL言語の標準命令をまとめたものです。

命令	修飾子	オペランド	説明
LD	N	変数、定数	オペランドをロード
ST	N	変数	現在結果のストア
S R		ブール型変数	TRUE にセット FALSE にリセット
AND & OR XOR	N (N (N (N (ブール型	ブール型 AND ブール型 AND ブール型 OR 排他的 OR
ADD SUB MUL DIV	((((変数、定数	加算 減算 乗算 除算
GT GE EQ LE LT	(((((変数、定数	比較: > 比較: >= 比較: = 比較: <= 比較: <

CAL	C N	ファンクションブロックのインスタンス名	ファンクションブロックのコール
JMP	C N	ラベル	ラベルへのジャンプ
RET	C N		サブプログラムからのリターン
)			遅延(先送り)処理の実行

以下に、IL固有の命令のみを示します。その他の標準的な命令は「標準命令、ファンクションブロック、ファンクション」で解説します。

LD 命令

オペレーション: 現在結果に値をロードします。

扱える修飾子: N

オペランド: 定数あるいは、
内部、入力、出力変数

例:

(* LD 命令の例 *)

```
LDex: LD false (* result := ブール定数 FALSE *)
      LD true (* result := ブール定数 TRUE *)
      LD 123 (* result := 整数定数 *)
      LD 123.1 (* result := 実数定数 *)
      LD t#3ms (* result := タイマ定数 *)
      LD boo_var1 (* result := ブール変数 *)
      LD ana_var1 (* result := アナログ変数 *)
      LD tmr_var1 (* result := タイマ変数 *)
      LDN boo_var2 (* result := NOT (ブール変数) *)
```

ST 命令

オペレーション: 現在結果を変数にストアします。現在結果はこの操作で変化しません。

扱える修飾子: N

オペランド: 内部、出力変数

例:

(* ST 命令の例 *)

```
STboo: LD false
      ST boo_var1 (* boo_var1 := FALSE *)
      STN boo_var2 (* boo_var2 := TRUE *)
STana: LD 123
      ST ana_var1 (* ana_var1 := 123 *)
STtmr: LD t#12s
      ST tmr_var1 (* tmr_var1 := t#12s *)
```

S 命令

オペレーション: 現在結果がTRUEの時にブール型変数にTRUEをストアします。もし、現在結果がFALSEの時は何の処理も行ないません。この操作によって現在結果は変化しません。

扱える修飾子: (なし)

オペランド: ブール型の出力、内部変数

IL言語

例:

```
(* S 命令の例 *)
SETex:    LD      true          (* 現在結果:= TRUE *)
          S       boo_var1      (* boo_var1 := TRUE *)
                                (* 現在結果は変化しない*)
          LD      false         (* 現在結果 := FALSE *)
          S       boo_var1      (* なにもしない。boo_var1 は変化しない*)
```

＝ R 命令

オペレーション: 現在結果がTRUEの時にブール型変数にFALSEをストアします。
もし、現在結果がFALSEの時は何の処理も行ないません。この
操作によって現在結果は変化しません。

扱える修飾子: (なし)

オペランド: ブール型の出力、内部変数

例:

```
(* R 命令の例 *)
RESETex: LD      true          (* 現在結果 := TRUE *)
          R       boo_var1      (* boo_var1 := FALSE *)
                                (* 現在結果は変化しない*)
          ST      boo_var2      (* boo_var2 := TRUE *)
          LD      false         (* 現在結果 := FALSE *)
          R       boo_var1      (* なにもしない。boo_var1 は変化しない*)
```

＝ JMP 命令

オペレーション: 指定のラベルへジャンプ

扱える修飾子: C N

オペランド: 同一のILプログラム内で定義されたラベル

例:

(* 以下の例はアナログセクタの値(0、1、2 のいずれか) をテストします。*)
(* 3つの出力ブール型に1をセット。セクタの内容が0の時ジャンプします *)

```
JMPex:    LD      selector      (* セクタの値は 0、1、2 のどれか *)
          BOO                      (* ブール型に変換*)
          JMPC    test1          (* selector = 0 の場合*)
          LD      true
          ST      bo0            (* bo0 := true *)
          JMP     JMPend         (* 終わりへ *)
test1:     LD      selector
          SUB     1              (* セクタの値を減らす。0 か 1 になる *)
          BOO                      (* ブール型に変換*)
          JMPC    test2          (* selector = 0 の場合*)
          LD      true
          ST      bo1            (* bo1 := true *)
          JMP     JMPend         (* 終わりへ *)
test2:     LD      true          (* 上記以外の場合*)
          ST      bo2            (* bo2 := true *)
JMPend:                                (* 終わり*)
```


RET 命令

オペレーション: 現在のILプログラムを終了します。もし、ILシーケンスがサブプログラム有的时候はILレジスタ(現在結果)が親プログラムに戻り値として戻されます。

扱える修飾子: C N

オペランド: (なし)

例:

(* 以下の例はアナログセレクタの値をテストします。(0 or 1 or 2) *)

(* 3つの出力ブール型に1をセット。セレクタの内容が0の時ジャンプします *)

```
JMPex:  LD      selector  (* セレクタの値は 0、1、2 のどれか *)
        BOO          (* ブール型に変換 *)
        JMPC    test1  (* selector = 0 の場合 *)
        LD      true
        ST      bo0    (* bo0 := true *)
        RET          (* 終わり。0 を返す。 *)
                (* セレクタ値の減算 *)

test1:   LD      selector
        SUB      1      (* 0 か 1 になる *)
        BOO          (* ブール型に変換 *)
        JMPC    test2  (* selector = 0 の場合 *)
        LD      true
        ST      bo1    (* bo1 := true *)
        LD      1      (* 実際のセレクタの値をロード *)
        RET          (* 終わり。1 を返す。 *)
                (* 上記以外の場合 *)

test2:   RETNC        (* セレクタの値が不正な場合 *)
        LD      true
        ST      bo2    (* bo2 := true *)
        LD      2      (* 実際のセレクタの値をロード *)
                (* 終わり。2 を返す。 *)
```

)" 命令

オペレーション: 遅延(先送り)されていた処理を実行します。先送りされていた処理は'('で修飾されています。

扱える修飾子: (なし)

オペランド: (なし)

例:

(* 遅延操作を含む例 *)

(* res := a1 + (a2 * (a3 - a4) * a5) + a6; *)

```
Delayed: LD      a1      (* 現在結果:= a1; *)
        ADD(    a2      (* 遅延する ADD - 現在結果 := a2; *)
        MUL(    a3      (* 遅延する MUL - 現在結果:= a3; *)
        SUB      a4      (* 現在結果 := a3 - a4; *)
        )          (* 遅延した MUL を実行 *)
                (* 現在結果:= a2 * (a3-a4); *)
```

MUL	a5	(* 現在結果:= a2 * (a3 - a4) * a5; *)
)		(* 遅延した ADD を実行 *)
		(* 現在結果 := a1 + (a2 * (a3 - a4) * a5); *)
ADD	a6	(* 現在結果 := a1 + (a2 * (a3 - a4) * a5) + a6; *)
ST	res	(* 変数 res に現在結果を格納*)

サブプログラムやファンクションのコール

IL、ST、LD、FBD、C言語で書かれたサブプログラムやファンクションをILから呼び出すことができます。サブプログラム名を命令として割り付けます。

オペレーション: サブプログラムやファンクションを実行します。サブプログラムからの戻り値はILレジスタ(現在結果)にストアされます。

扱える修飾子: (なし)

オペランド: コールする前に1番目のパラメータをILレジスタに格納しておく必要があります。残りのパラメータはコンマで区切ってオペランドのフィールドに書きます。

例:

(* プログラムのコール: アナログ値をタイマ値に変換 *)

Main:	LD	bi0	
	SUBPRO	bi1,bi2	(* サブプログラムをコールしてアナログ値を得る*)
	ST	result	(* 現在結果 := サブプログラムの戻り値 *)
	GT	vmax	(* オーバーフローしたか? *)
	RETC		(* オーバーフローした*)
	LD	result	
	MUL	1000	(* 秒をミリ秒に変換*)
	TMR		(* タイマ型に変換*)
	ST	tmval	(* 変換結果を変数タイマ変数に格納 *)

(* コールされるサブプログラム 'SUBPRO' : アナログ値の比較*)

(* バイナリ値を3つのブール in0, in1, in2 でサブプログラムの入力パラメータとして渡す *)

	LD	in2	
	ANA		(* 現在結果= ana (in2); *)
	MUL	2	(* 現在結果 := 2*ana (in2); *)
	ST	temporary	(* temporary := 現在結果 *)
	LD	in1	
	ANA		
	ADD	temporary	(* 現在結果 := 2*ana (in2) + ana (in1); *)
	MUL	2	(* 現在結果 := 4*ana (in2) + 2*ana (in1); *)
	ST	temporary	(* temporary := 現在結果 *)
	LD	in0	
	ANA		
	ADD	temporary	(* 現在結果= 4*ana (in2) + 2*ana (in1)+ana (in0);
*)			
	ST	SUBPRO	(*現在結果を親プログラムに返す*)

■ ファンクションブロックのコール: CAL 命令

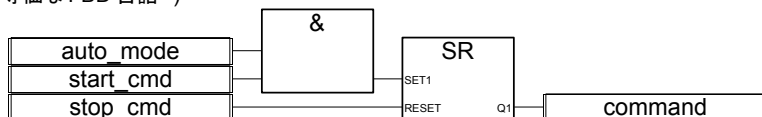
オペレーション: ファンクションブロックのコール
扱える修飾子: C N
オペランド: ファンクションブロックのインスタンス名。
 ファンクションブロックの入力パラメータはコールされる前に LD/ST 命令で代入しておく必要があります。
 出力パラメータがある場合は宣言済みの変数として扱われます。

例1:

(* ファンクションブロック SR のコール : SR1 は SR のインスタンス *)

```
LD      auto_mode
AND     start_cmd
ST      SR1.set1
LD      stop_cmd
ST      SR1.reset
CAL     SR1
LD      SR1.Q1
ST      command
```

(* 等価な FBD 言語 *)

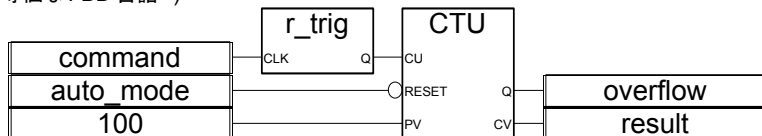


例2:

(*R_TRIG1 は R_TRIG ブロックのインスタンスで、CTU1 は CTU ブロックのインスタンス*)

```
LD      command
ST      R_TRIG1.clk
CAL     R_TRIG1
LD      R_TRIG1.Q
ST      CTU1.cu
LDN     auto_mode
ST      CTU1.reset
LD      100
ST      CTU1.pv
CAL     CTU1
LD      CTU1.Q
ST      overflow
LD      CTU1.cv
ST      result
```

(* 等価な FBD 言語 *)

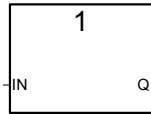


B.9 標準命令、ファンクションブロックとファンクション

B.9.1 標準命令

以下に IEC 61131-3 で定義されている標準命令を示します。

データ操作:	代入, 整数・実数の符号反転
ブール操作:	AND(論理積) OR(論理和) XOR(排他的論理和)
算術演算:	+(加算) -(減算) *(乗算) /(除算)
論理操作:	AND_MASK(ビット毎の AND マスク) OR_MASK(ビット毎の OR マスク) XOR_MASK(ビット毎の XOR マスク) NOT_MASK(ビット毎の反転)
比較テスト:	< (より小さい) <= (等しいかより小さい) > (より大きい) >= (等しいかより大きい) =(等しい) <>(等しくない)
データ変換:	BOO(ブール型変換) ANA(整数型変換) REAL(実数型変換) TMR(タイマ型変換) MSG(可変長文字列型変換)
その他:	CAT(文字列の結合) SYSTEM(システムへのアクセス) OPERATE(I/Oチャンネルの操作)

1 gain (代入)

引数:

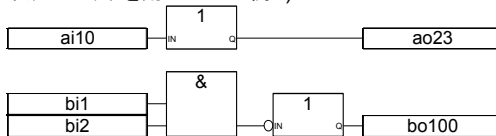
IN	あらゆるタイプ
Q	あらゆるタイプ

説明:

ある変数を別の変数に代入します。

このブロックは入力と出力を直接接続するブロックです。論理の反転(○シンボル)もあわせて使うことができます。

(* 代入ブロックを用いたFBD例 *)



(* 等価なST言語: *)

```
ao23 := ai10;
```

```
bo100 := NOT (bi1 AND bi2);
```

(*等価な IL 言語: *)

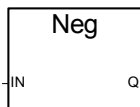
```
LD      ai10
```

```
ST      ao23
```

```
LD      bi1
```

```
AND     bi2
```

```
LDN     bo100
```

NEG (符号反転)

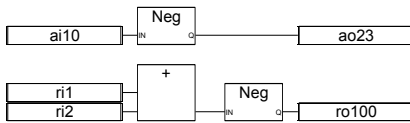
引数:

IN	整数・実数型	入力、出力パラメータは同じタイプ
Q	整数・実数型	

説明:

変数の反転(符号反転)の代入

(*Negation をブロックを用いたFBD例 *)



(* 等価なST言語: *)

ao23 := - (ai10);

ro100 := - (ri1 + ri2);

(*等価な IL 言語: *)

LD ai10

MUL -1

ST ao23

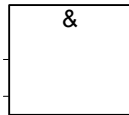
LD ri1

ADD ri2

MUL -1.0

LD ro100

& AND (論理積)



注意: この命令の入力数は3個以上に変更可能です。

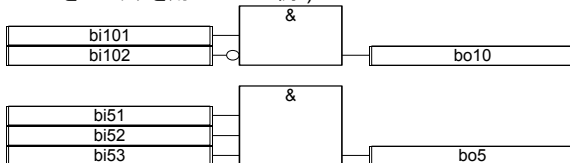
引数:

(入力)	ブール型	
出力	ブール型	入力パラメータの論理積

説明:

2つ以上の入力パラメータの論理積

(*"AND" をブロックを用いたFBD例*)



(* 等価なST言語: *)

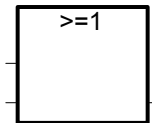
bo10 := bi101 AND NOT (bi102);

bo5 := (bi51 AND bi52) AND bi53;

(* 等価なIL言語: *)

LD	bi101	(* 現在結果 := bi101 *)
ANDN	bi102	(* 現在結果 := bi101 AND not(bi102) *)
ST	bo10	(* bo := 現在結果 *)
LD	bi51	(* 現在結果 := bi51;
&	bi52	(* 現在結果 := bi51 AND bi52 *)
&	bi53	(* 現在結果 := (bi51 AND bi52) AND bi53
*)		
ST	bo5	(* bo := 現在結果 *)

>=1 OR (論理和)



注意: この命令の入力数は3個以上に変更可能です。

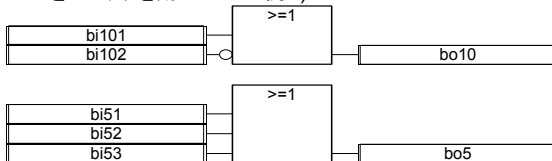
引数:

(入力)	ブール型	
出力	ブール型	入力パラメータの論理和

説明:

2つ以上の入力パラメータの論理和

(* "OR" をブロックを用いたFBD例 *)



(* 等価なST言語: *)

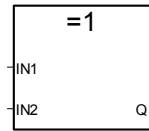
bo10 := bi101 OR NOT (bi102);

bo5 := (bi51 OR bi52) OR bi53;

(* 等価なIL言語: *)

LD	bi101
ORN	bi102
ST	bo10
LD	bi51
OR	bi52
OR	bi53
ST	bo5

=1 XOR（排他的論理和）



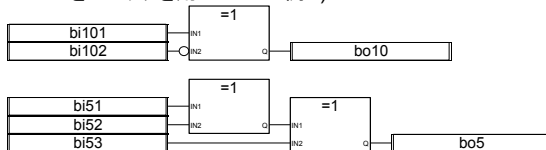
引数:

IN1	ブール型	
IN2	ブール型	
Q	ブール型	2つのブール入力の排他的論理和

説明:

2つのブール値の排他的論理和

(* "XOR" をブロックを用いたFBD例 *)



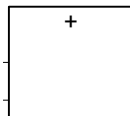
(* 等価なST言語: *)

bo10 := bi101 XOR NOT (bi102);
bo5 := (bi51 XOR bi52) XOR bi53;

(* 等価なIL言語: *)

```
LD      bi101
XORN    bi102
ST      bo10
LD      bi51
XOR     bi52
XOR     bi53
ST      bo5
```

+（加算）



注意: この命令の入力数は3個以上に変更可能です。

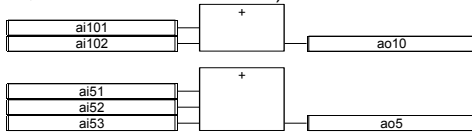
引数:

(入力)	整数・実数型 (全入力は同一タイプ)
出力	整数・実数型 全入力の符号付き加算

説明:

複数の整数・実数型変数の符号付き加算

(* 加算ブロックを用いたFBD例*)



(* 等価なST言語: *)

ao10 := ai101 + ai102;

ao5 := (ai51 + ai52) + ai53;

(* 等価なIL言語: *)

LD ai101

ADD ai102

ST ao10

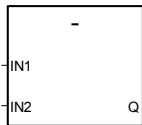
LD ai51

ADD ai52

ADD ai53

ST ao5

- (減算)



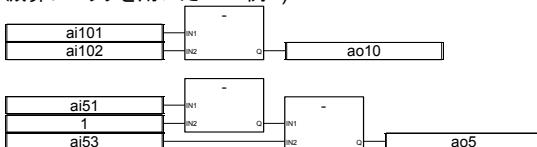
引数:

IN1	整数・実数型
IN2	整数・実数型 (IN1 と IN2 は同じタイプ)
Q	整数・実数型 減算結果 (IN1 - IN2)

説明:

2つの整数・実数型変数の減算(1番目－2番目)。

(* 減算ブロックを用いたFBD例 *)



(* 等価なST言語: *)

ao10 := ai101 - ai102;

ao5 := (ai51 - 1) - ai53;

(* 等価なIL言語: *)

LD ai101

標準命令、ファンクションブロックとファンクション

SUB	ai102
ST	ao10
LD	ai51
SUB	1
SUB	ai53
ST	ao5

* (乗算)



注意: この命令の入力数は3個以上に変更可能です。

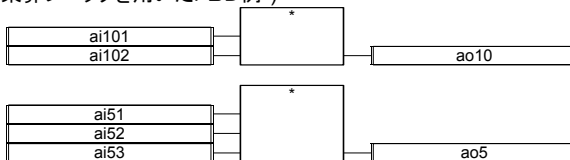
引数:

(入力)	整数・実数型 (全入力と同じタイプ)
出力	整数・実数型 入力の符号付き乗算

説明:

複数の整数・実数型変数の乗算

(* 乗算ブロックを用いたFBD例*)



(* 等価なST言語: *)

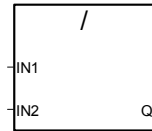
ao10 := ai101 * ai102;

ao5 := (ai51 * ai52) * ai53;

(* 等価なIL言語: *)

LD	ai101
MUL	ai102
ST	ao10
LD	ai51
MUL	ai52
MUL	ai53
ST	ao5

/ (除算)



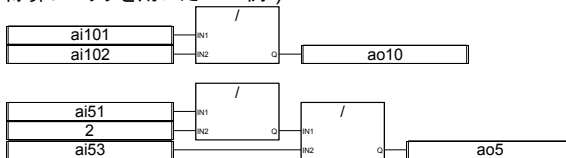
引数:

IN1	整数・実数型	
IN2	整数・実数型	除算数は0でないこと。 (IN1 と IN2 は同じタイプ)
Q	整数・実数型	符号付き除算 (IN1 / IN2)

説明:

2つの整数・実数型変数の除算(1番目／2番目)

(* 除算ブロックを用いたFBD例*)



(* 等価なST言語: *)

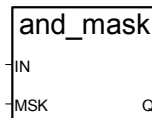
ao10 := ai101 / ai102;

ao5 := (ai5 / 2) / ai53;

(* 等価なIL言語: *)

```
LD      ai101
DIV     ai102
ST      ao10
LD      ai51
DIV     2
DIV     ai53
ST      ao5
```

AND_MASK



引数:

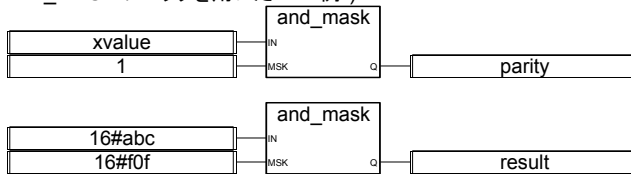
IN	整数型	
MSK	整数型	
Q	整数型	IN と MSK のビット毎の論理積

標準命令、ファンクションブロックとファンクション

説明:

整数型変数の論理積で IN の各ビットと MSK の各ビットとの論理積

(* AND_MASK ブロックを用いたFBD例*)



(* 等価なST言語: *)

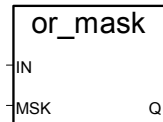
parity := AND_MASK (xvalue, 1); (* xvalue が奇数なら 1 になる*)

result := AND_MASK (16#abc, 16#0f); (* 16#a0c になる*)

(* 等価なIL言語: *)

```
LD      xvalue
AND_MASK 1
ST      parity
LD      16#abc
AND_MASK 16#0f
ST      result
```

OR_MASK



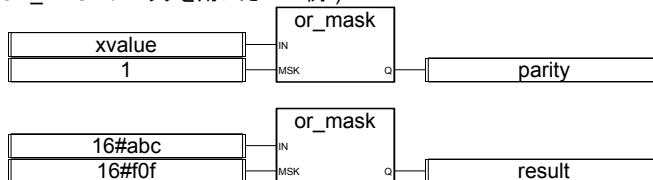
引数:

IN	整数型	
MSK	整数型	
Q	整数型	IN と MSK のビット毎の論理和

説明:

整数型変数の論理和で IN の各ビットと MSK の各ビットとの論理和

(* OR_MASK ブロックを用いたFBD例*)



(* 等価なST言語: *)

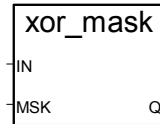
is_odd := OR_MASK (xvalue, 1); (* 必ず奇数になる*)

result := OR_MASK (16#abc, 16#0f); (* 16#bf になる*)

(* 等価なIL言語: *)

```
LD      xvalue
OR_MASK 1
ST      is_odd
LD      16#abc
OR_MASK 16#f0f
ST      result
```

XOR_MASK



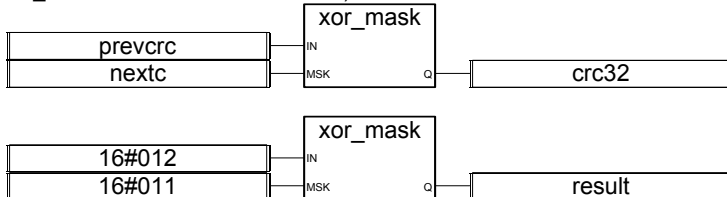
引数:

IN	整数型	
MSK	整数型	
Q	整数型	IN と MSK のビット毎の排他的論理和

説明:

整数型変数の排他的論理和で IN の各ビットと MSK の各ビットとの排他的論理和

(*XOR_MASK ブロックを用いたFBD例 *)



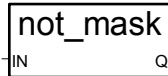
(* 等価なST言語: *)

```
crc32 := XOR_MASK (prevcrc, nextc);
result := XOR_MASK (16#012, 16#011); (* 16#003 になる *)
```

(* 等価なIL言語: *)

```
LD      prevcrc
XOR_MASK nextc
ST      crc32
LD      16#012
XOR_MASK 16#011
ST      result
```

NOT_MASK



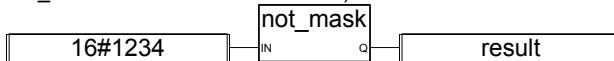
引数:

IN	整数型	
Q	整数型	32ビット表現の IN の各ビットの反転

説明:

整数型変数の各ビットの反転

(*NOT_MASK ブロックを用いたFBD例*)



(* 等価なST言語: *)

result := NOT_MASK (16#1234);

(* result は 16#FFFF_EDCB *)

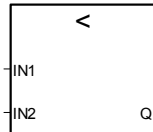
(* 等価なIL言語: *)

LD 16#1234

NOT_MASK

ST result

< (より小さい)



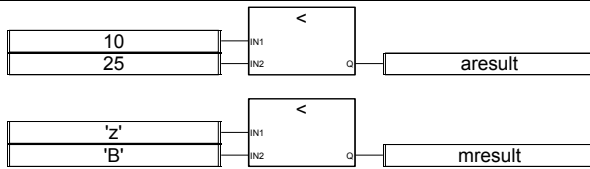
引数:

IN1	整数型、実数型、 タイマ型、文字列型	
IN2	整数型、実数型、 タイマ型、文字列型	
Q	ブール型	2つの入力は同一タイプ もし、IN1 < IN2 なら TRUE

説明:

ある変数が別のものより小さいか比較します。

(*"<" ブロックを用いたFBD例*)



(* 等価なST言語: *)

areresult := (10 < 25); (* areresult は TRUE *)

mresult := ('z' < 'B'); (* mresult は FALSE *)

(* 等価なIL言語: *)

LD 10

LT 25

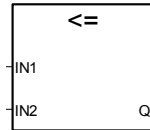
ST areresult

LD 'z'

LT 'B'

ST mresult

<= (等しいかより小さい)



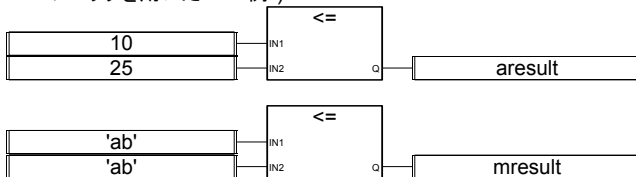
引数:

IN1	整数型、実数型、文字列型	
IN2	整数型、実数型、文字列型	2つの入力は同じタイプ
Q	ブール型	もし、IN1 <= IN2 なら TRUE

説明:

ある変数が別のものと等しいかより小さいかを比較します。

(* "<=" ブロックを用いたFBD例*)



(* 等価なST言語: *)

areresult := (10 <= 25); (* areresult は TRUE *)

mresult := ('ab' <= 'ab'); (* mresult は TRUE *)

(* 等価なIL言語: *)

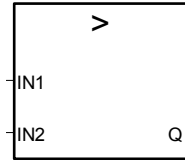
LD 10

LE 25

標準命令、ファンクションブロックとファンクション

ST	areresult
LD	'ab'
LE	'ab'
ST	mresult

> (より大きい)



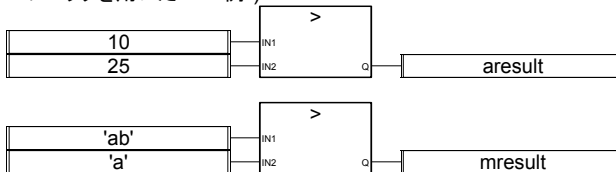
引数:

IN1	整数型、実数型、 タイマ型、文字列型	
IN2	整数型、実数型、 タイマ型、文字列型	2つは同じタイプ
Q	ブール型	もし IN1 > IN2 なら TRUE

説明:

ある変数が別のものより大きいかが比較します。

(* ">" ブロックを用いたFBD例 *)

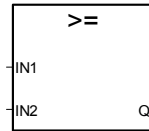


(* 等価なST言語: *)

areresult := (10 > 25); (* areresult は FALSE *)
mresult := ('ab' > 'a'); (* mresult は TRUE *)

(* 等価なIL言語: *)

LD	10
GT	25
ST	areresult
LD	'ab'
GT	'a'
ST	mresult

>= (等しいかより大きい)

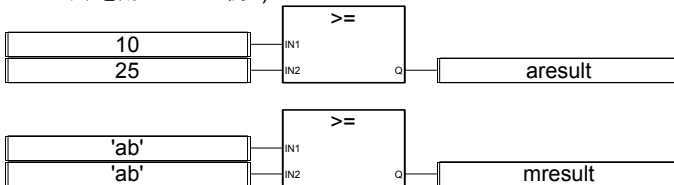
引数:

IN1	整数型、実数型、文字列型	
IN2	整数型、実数型、文字列型	2つの入力は同じタイプ
Q	ブール型	もし IN1 >= IN2 なら TRUE

説明:

ある変数が別のものと等しいかより大きいかが比較します。

(* ">="ブロックを用いたFBD例 *)



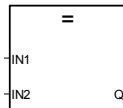
(* 等価なST言語: *)

aresult := (10 >= 25); (* aresult は FALSE *)

mresult := ('ab' >= 'ab'); (* mresult は TRUE *)

(* 等価なIL言語: *)

```
LD      10
GE      25
ST      aresult
LD      'ab'
GE      'ab'
ST      mresult
```

= (等しい)

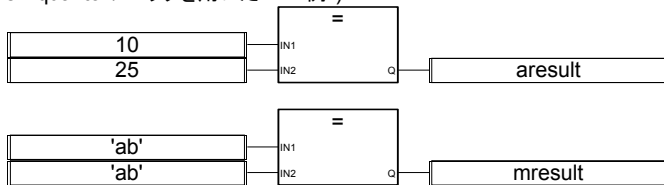
引数:

IN1	整数型、実数型、文字列型	
IN2	整数型、実数型、文字列型	2つの入力は同じタイプ
Q	ブール型	もし IN1 = IN2 なら TRUE

説明:

ある変数が別のものと等しいかが比較します。

(*Is Equal to"ブロックを用いたFBD例*)



(* 等価なST言語: *)

aresult := (10 = 25); (* aresult は FALSE *)

mresult := ('ab' = 'ab'); (* mresult は TRUE *)

(* 等価なIL言語: *)

```
LD      10
EQ      25
ST      aresult
LD      'ab'
EQ      'ab'
ST      mresult
```

<> (等しくない)



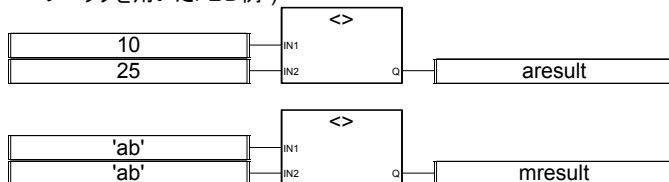
引数:

IN1	整数型、実数型、文字列型	
IN2	整数型、実数型、文字列型	2つの入力と同じタイプ
Q	ブール型	もし IN1 <> IN2 なら TRUE

説明:

ある変数が別のものと等しくないか比較します。

(*"<>" ブロックを用いたFBD例*)



(* 等価なST言語: *)

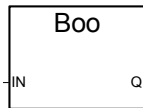
aresult := (10 <> 25); (* aresult は TRUE *)

mresult := ('ab' <> 'ab'); (* mresult は FALSE *)

(* 等価なIL言語: *)

```
LD      10
NE      25
ST      aresult
LD      'ab'
NE      'ab'
ST      mresult
```

BOO (ブール型変換)



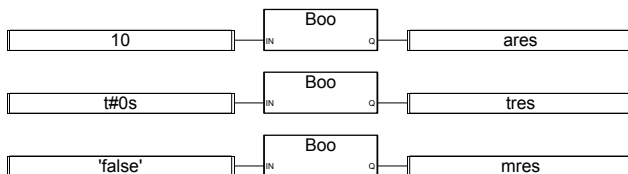
引数:

IN	あらゆるタイプ (ブール型以外)	
Q	ブール型	TRUE: 0以外の数値 FALSE: 0 TRUE: 'TRUE' 文字列 FALSE: 'FALSE' 文字列

説明:

変数をブール型に変換します。

(*ブール型変換ブロックを用いたFBD例*)



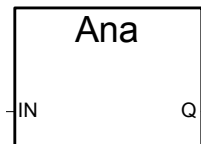
(* 等価なST言語: *)

ares := BOO (10);	(* ares は TRUE *)
tres := BOO (t#0s);	(* tres は FALSE *)
mres := BOO ('false');	(* mres は FALSE *)

(* 等価なIL言語: *)

```
LD      10
BOO
ST      ares
LD      t#0s
BOO
ST      tres
LD      'false'
BOO
ST      mres
```

ANA（整数型変換）



引数:

IN

あらゆるタイプ（整数型以外）

Q

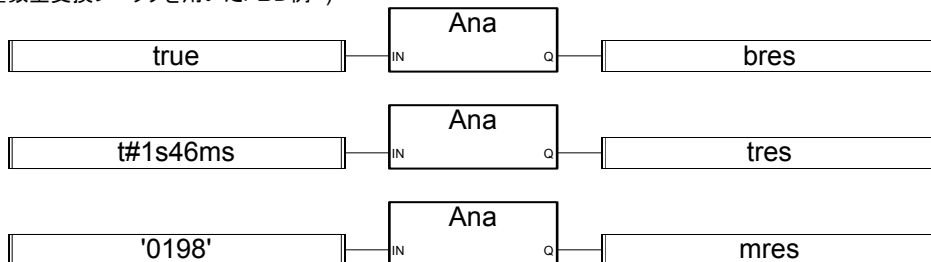
整数型

0 : IN が FALSE の時、
 1 : IN が TRUE の時
 タイマのms数値
 実数変数の整数部分
 文字列の10進数表現

説明:

変数を整数型へ変換します。

(* 整数型変換ブロックを用いたFBD例 *)



(* 等価なST言語: *)

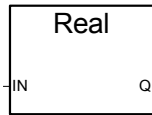
```
bres := ANA (true);
tres := ANA (t#1s46ms);
mres := ANA ('0198');
```

```
(* bres は 1 *)
(* tres は 1046 *)
(* mres は 198 *)
```

(* 等価なIL言語: *)

```
LD      true
ANA
ST      bres
LD      t#1s46ms
ANA
ST      tres
LD      '0198'
ANA
ST      mres
```

REAL (実数型変換)



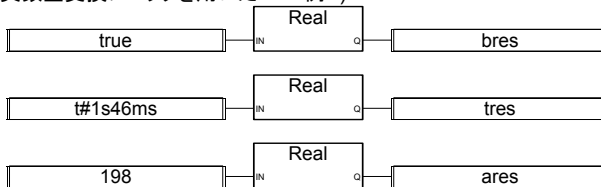
引数:

IN Q	ブール型、整数型、 タイマ型 実数型	実数型、文字列型以外 0.0 : IN が FALSE 1.0 : IN が TRUE タイマのミリ秒単位での数値 整数型と同じ数値
---	------------------------------	--

説明:

変数を実数型へ変換します。

(* 実数型変換ブロックを用いたFBD例 *)



(* 等価なST言語: *)

bres := REAL (true);

(* bres は 1.0 *)

tres := REAL (t#1s46ms);

(* tres は 1046.0 *)

ares := REAL (198);

(* ares は 198.0 *)

(* 等価なIL言語: *)

LD true

REAL

ST bres

LD t#1s46ms

REAL

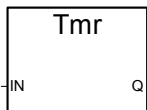
ST tres

LD 198

REAL

ST ares

TMR (タイマ型変換)



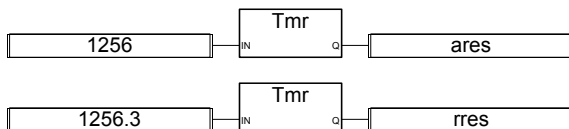
引数:

IN	整数型、実数型 (タイマ以外)
	ミリ秒単位の数値
Q	タイマ型
	IN で表現されるタイマ値
	IN が実数の時はその整数部分

説明:

整数型、実数型変数をタイマ値へ変換します。

(* タイマ型変換ブロックを用いたFBD例*)



(* 等価なST言語: *)

ares := TMR (1256);

rres := TMR (1256.3);

(* ares := t#1s256ms *)

(*rres := t#1s256ms *)

(* 等価なIL言語: *)

LD 1256

TMR

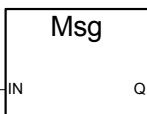
ST ares

LD 1256.3

TMR

ST rres

MSG (可変長文字列型変換)



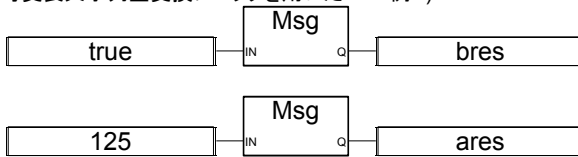
引数:

IN	ブール型、 整数型、実数型 (文字列型以外)
Q	文字列型
	IN がブール型の時 'false' 又は 'true'
	IN が整数、実数型の時 10 進数表現

説明:

変数を文字列型へ変換します。

(* 可変長文字列型変換ブロックを用いたFBD例 *)



(* 等価なST言語: *)

bres := MSG (true); (* bres は 'TRUE' *)

ares := MSG (125); (* ares は '125' *)

(* 等価なIL言語: *)

LD true

MSG

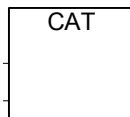
ST bres

LD 125

MSG

ST ares

CAT (可変長文字列結合)



注意: この命令の入力数は3個以上に変更可能です。

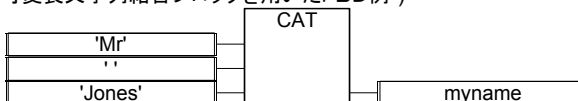
引数:

(入力)	文字列型	(文字列の結合の結果は、出力の文字列の最大長を 越えられません)
出力	文字列型	入力文字列の結合結果

説明:

複数の文字列を1つに結合します。

(* 可変長文字列結合ブロックを用いたFBD例*)



(*等価なST言語: +演算子を使います。*)

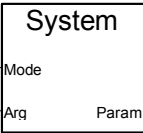
myname := ('Mr' + ' ') + 'Jones';

(* 意味: myname := 'Mr Jones' *)

標準命令、ファンクションブロックとファンクション

```
(* 等価なIL言語: *)
LD      'Mr'
ADD     ''
ADD     'Jones'
ST      myname
```

SYSTEM



引数:

Mode	整数型	システムパラメータの識別子とアクセスモード
Arg	整数型-タイマ型	書き込みモード時の書き込む値
Param	整数型	アクセスしたシステムパラメータの値

説明:
システムパラメータへアクセスします。

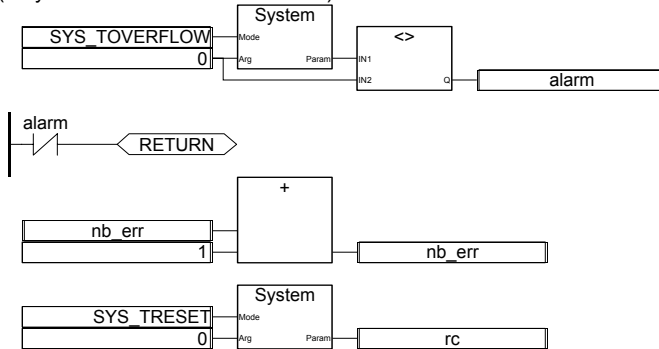
以下に扱えるSYSTEMファンクションのコマンド(モード)例を示します。これらのコマンドは定義済みのワードです。

コマンド	意味
SYS_TALLOWED	許容サイクルタイムの読み出し
SYS_TCURRENT	現在のサイクルタイムの読み出し
SYS_TMAXIMUM	最大サイクルタイムの読み出し
SYS_TOVERFLOW	サイクルタイムのオーバーフロー回数の読み出し
SYS_TRESET	タイミングカウンタのリセット
SYS_TWRITE	サイクルタイムの変更
SYS_ERR_TEST	ランタイムエラーのチェック
SYS_ERR_READ	最古のランタイムエラーの読み出し

以下にSYSTEMファンクションのそれぞれのコマンドに対する引数を示します。

コマンド	引数	戻り値
SYS_TALLOWED	0	許容サイクルタイム
SYS_TCURRENT	0	現在のサイクルタイム
SYS_TMAXIMUM	0	最大サイクルタイム
SYS_TOVERFLOW	0	オーバーフロー回数
SYS_TRESET	0	0
SYS_TWRITE	新サイクルタイム	書き込まれたサイクルタイム
SYS_ERR_TEST	0	エラーがなければ0
SYS_ERR_READ	0	最古のエラーのコード

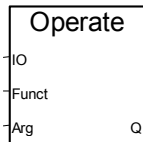
(* "System"ブロックを用いたFBD例*)



(* 等価なST言語: *)

```
alarm := (SYSTEM (SYS_TOVERFLOW, 0) <> 0);
If (alarm) Then
    nb_err := nb_err + 1;
    rc := SYSTEM (SYS_TRESET, 0);
End_If;
```

OPERATE



引数:

IO	全タイプ	入出力変数
Funct	整数型	指定アクション
Arg	整数型	I/Oアクションに対する引数
Q	整数型	戻り値

説明:

I/Oチャンネルへアクセスします。

OPERATE の処理内容はI/Oドライバに依存します。詳しくは、対応するI/Oドライバの技術メモなどを参照願います。

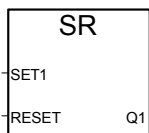
B.9.2 標準ファンクションブロック

ISaGRAFによりサポートされている標準のファンクションブロックを以下に示します。これらは、標準で備わっているためにISaGRAFライブラリに登録する必要がありません。

ブール操作:	SR: (セット優先双安定) RS: (リセット優先双安定) R_Trig: (立ち上がり検出) F_Trig: (立ち下がり検出) SEMA: (セマフォ)
カウンタ:	CTU: (アップカウンタ) CTD: (ダウンカウンタ) CTUD: (アップダウンカウンタ)
タイマ:	TON: (オンディレイタイマ) TOF: (オフディレイタイマ) TP: (パルスタイマ)
整数アナログ:	CMP: (完全比較) StackInt: (整数値のスタック)
実数アナログ:	AVERAGE: (移動平均) HYSTER: (ヒステリシス) LIM_ALARM: (ヒステリシスを持った上限／下限リミット) INTEGRAL: (時間積分) DERIVATE: (時間微分)
信号発信:	BLINK: (ブール型信号のブリンク) SIG_GEN: (信号発生器)

注意: 新しいC言語ファンクションブロックが作られた場合でもFBDダイアグラムからコールすることができます。

SR (セット優先双安定)



引数:

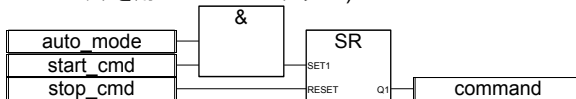
SET1	ブール型	TRUE なら Q1 を TRUE にセットします (優先)
RESET	ブール型	TRUE なら Q1 を FALSE にリセットします。
Q1	ブール型	ブール型のメモリ状態

説明:

セット優先双安定: 以下の表を参照

Set1	Reset	Q1	result Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

(*"SR"ブロックを用いたFBDプログラム*)



(*等価なST言語: SR1 は SR ブロックのインスタンス *)

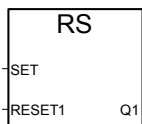
SR1((auto_mode & start_cmd), stop_cmd);

command := SR1.Q1;

(* 等価なIL言語: *)

```

LD      auto_mode
AND     start_cmd
ST      SR1.set1
LD      stop_cmd
ST      SR1.reset
CAL     SR1
LD      SR1.Q1
ST      command
  
```

RS (リセット優先双安定)

引数:

SET	ブール型	TRUE なら Q1 を TRUE にセットします。
RESET1	ブール型	TRUE で Q1 を FALSE にリセットします (優先)
Q1	ブール型	ブール型メモリ状態

説明:

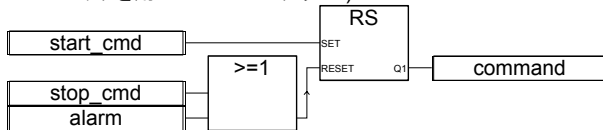
リセット優先双安定: 以下の表を参照

Set	Reset1	Q1	result Q1
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1

標準命令、ファンクションブロックとファンクション

1	0	1	1
1	1	0	0
1	1	1	0

(*RS"ブロックを用いたFBDプログラム*)



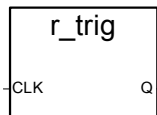
(*等価なST言語: RS1 は RS ブロックのインスタンス *)

```
RS1(start_cmd, (stop_cmd OR alarm));
command := RS1.Q1;
```

(* 等価なIL言語: *)

```
LD      start_cmd
ST      RS1.set
LD      stop_cmd
OR      alarm
ST      RS1.reset1
CAL     RS1
LD      RS1.Q1
ST      command
```

R_TRIG (立ち上がり検出)



引数:

CLK
Q

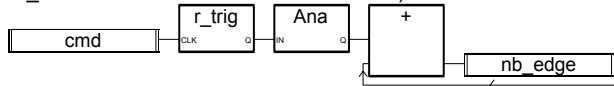
ブール型
ブール型

TRUE : CLK が FALSE が TRUE へ立ち上がったとき
FALSE: それ以外の場合

説明:

ブール型変数の立ち上がりを検出します(1サイクル前の値との比較を行います)。

(*R_TRIG"ブロックを用いたFBDプログラム*)



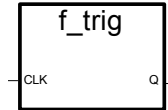
(*等価なST言語: R_TRIG1 は R_TRIG ブロックのインスタンス *)

```
R_TRIG1(cmd);
nb_edge := ANA(R_TRIG1.Q) + nb_edge;
```

(* 等価なIL言語: *)

```
LD      cmd
```

ST	R_TRIG1.clk
CAL	R_TRIG1
LD	R_TRIG1.Q
ANA	
ADD	nb_edge
ST	nb_edge

F_TRIG (立ち下がり検出)

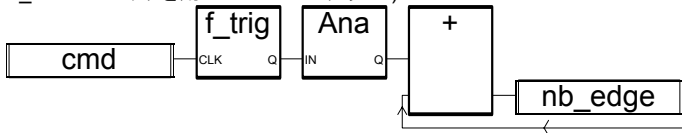
引数:

CLK	ブール型	
Q	ブール型	TRUE: CLK が TRUE から FALSE へ立ち下がったとき FALSE: それ以外の場合

説明:

ブール型変数の立ち下がりを検出します(1サイクル前の値との比較を行います)。

(*"F_TRIG"ブロックを用いたFBDプログラム*)



(*等価なST言語: F_TRIG1 は F_TRIG ブロックのインスタンス *)

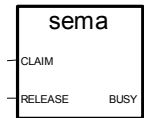
F_TRIG1(cmd);

nb_edge := ANA(F_TRIG1.Q) + nb_edge;

(* 等価なIL言語: *)

LD	cmd
ST	F_TRIG1.clk
CAL	F_TRIG1
LD	F_TRIG1.Q
ANA	
ADD	nb_edge
ST	nb_edge

SEMA (セマフォ)



引数:

CLAIM	ブール型	"テストと 取得" コマンド
RELEASE	ブール型	セマフォの開放
BUSY	ブール型	セマフォの状態

説明:

(* "x" はブール型 変数で初期値は FALSE です。*)

busy := x;

If claim Then

x := True;

Else

If release Then

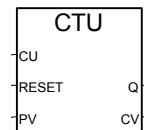
busy := False;

x := False;

End_if;

End_if;

CTU (アップカウンタ)



引数:

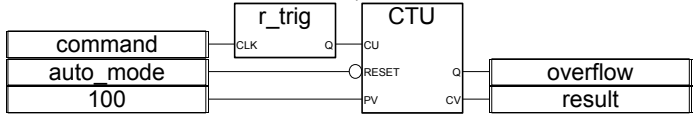
CU	ブール型	カウント入力 (CU が TRUE でカウント)
RESET	ブール型	リセットコマンド(優先)
PV	整数型	最大カウント値
Q	ブール型	オーバーフロー: CV = PV でTRUE
CV	整数型	現在のカウント結果

注意: CTUは入力CUの立ち上がり、立ち下がりを検出しません。必ず "R_TRIG" あるいは "F_TRIG" を使ってパルスカウンタを作る必要があります。

説明:

カウント数(整数型)は0から1ずつ増加

(*CTU"ブロックを用いたFBDプログラム*)



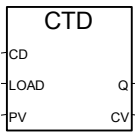
(*等価なST言語: R_TRIG1 は R_TRIG ブロックのインスタンス CTU1 は CTU ブロックのインスタンス*)

```
CTU1(R_TRIG1(command),NOT(auto_mode),100);
overflow := CTU1.Q;
result := CTU1.CV;
```

(* 等価なIL言語: *)

```
LD      command
ST      R_TRIG1.clk
CAL     R_TRIG1
LD      R_TRIG1.Q
ST      CTU1.cu
LDN     auto_mode
ST      CTU1.reset
LD      100
ST      CTU1.pv
CAL     CTU1
LD      CTU1.Q
ST      overflow
LD      CTU1.cv
ST      result
```

CTD (ダウンカウンタ)



引数:

CD	ブール型	カウント入力 (CD が TRUE の時カウントダウン)
LOAD	ブール型	ロードコマンド (優先) (LOAD が TRUE のとき CV = PV になります)
PV	整数型	カウンタの初期値
Q	ブール型	アンダーフロー: CV = 0 で TRUE
CV	整数型	カウンタ結果

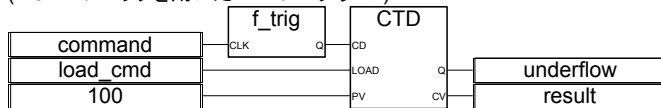
注意: CTDは入力CDの立ち上がり、立ち下がりを検出しません。必ず "R_TRIG" あるいは "F_TRIG" を使ってパルスカウンタを作る必要があります。

標準命令、ファンクションブロックとファンクション

説明:

カウント数(整数型)は PV 値から1ずつ0まで減少。

(*CTD"ブロックを用いたFBDプログラム*)



(*等価なST言語: F_TRIG1 は F_TRIG ブロックのインスタンス CTD1 は CTD ブロックのインスタンス*)

```
CTD1(F_TRIG1(command),load_cmd,100);
```

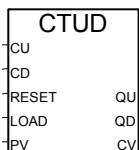
```
underflow := CTD1.Q;
```

```
result := CTD1.CV;
```

(* 等価なIL言語: *)

```
LD      command
ST      F_TRIG1.clk
CAL     F_TRIG1
LD      F_TRIG1.Q
ST      CTD1.cd
LD      load_cmd
ST      CTD1.load
LD      100
ST      CTD1.pv
CAL     CTD1
LD      CTD1.Q
ST      underflow
LD      CTD1.cv
ST      result
```

CTUD (アップダウンカウンタ)



引数:

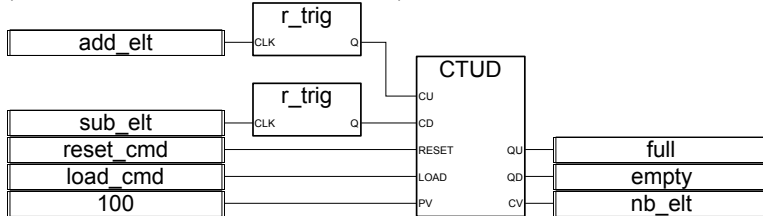
CU	ブール型	アップカウンタ入力 (CU が TRUE でカウント)
CD	ブール型	ダウンカウンタ入力 (CD が TRUE でカウント)
RESET	ブール型	リセットコマンド (優先) (RESET が TRUE の時 CV = 0)
LOAD	ブール型	ロードコマンド (LOAD が TRUE の時 CV = PV)
PV	整数型	最大カウント値
QU	ブール型	オーバーフロー: CV = PV のとき TRUE
QD	ブール型	アンダーフロー: CV = 0 のとき TRUE
CV	整数型	カウント結果

注意: CTUDは入力CU、CDの立ち上がり、立ち下がりを検出しません。必ず
"R_TRIG" あるいは "F_TRIG" を使ってパルスカウンタを作る必要があります。

説明:

カウント数(整数型)は0からPV 値まで1ずつ増加、あるいは、
現在値から1ずつ0まで減少。

(*CTUD"ブロックを用いたFBDプログラム*)



(*等価なST言語: R_TRIG1と R_TRIG2は R_TRIG ブロックのインスタンス、CTUD1は CTUD ブロックのインスタンス*)

```
CTUD1(R_TRIG1(add_elt), R_TRIG2(sub_elt), reset_cmd, load_cmd, 100);
```

```
full := CTUD1.QU;
```

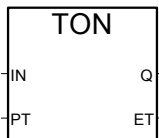
```
empty := CTUD1.QD;
```

```
nb_elt := CTUD1.CV;
```

(* 等価なIL言語: *)

```
LD      add_elt
ST      R_TRIG1.clk
CAL     R_TRIG1
LD      R_TRIG1.Q
ST      CTUD1.cu
LD      sub_elt
ST      R_TRIG2.clk
CAL     R_TRIG2
LD      R_TRIG2.Q
ST      CTUD1.cd
LD      load_cmd
ST      CTUD1.load
LD      100
ST      CTUD1.pv
CAL     CTUD1
LD      CTUD1.QU
ST      full
LD      CTUD1.QD
ST      empty
LD      CTUD1.CV
ST      nb_elt
```

TON (オンディレータイマ)



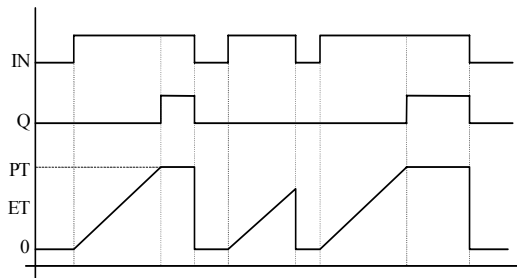
引数:

IN	ブール型	立ち上がり検出でタイマ値のインクリメントを開始 立ち下がり検出でタイマ値停止、リセット
PT	タイマ型	タイムアップ設定値
Q	ブール型	TRUE のとき、タイムアップ設定値が経過
ET	タイマ型	現在経過タイマ値

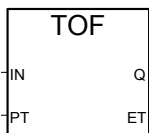
説明:

内部タイマ値を指定された値 (PT) までインクリメント

タイムチャート:



TOF (オフディレータイマ)



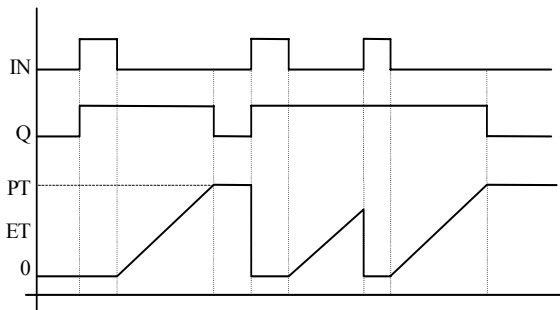
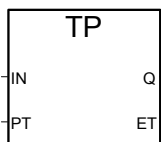
引数:

IN	ブール型	立ち下がり検出でタイマ値インクリメントを開始 立ち上がり検出でタイマ値停止、リセット
PT	タイマ型	設定タイマ値
Q	ブール型	TRUE のとき、設定タイマ値まで経過していない
ET	タイマ型	現在経過タイマ値

説明:

内部タイマ値を指定された値 (PT) までインクリメント

タイムチャート:

**TP (パルスタイマ)**

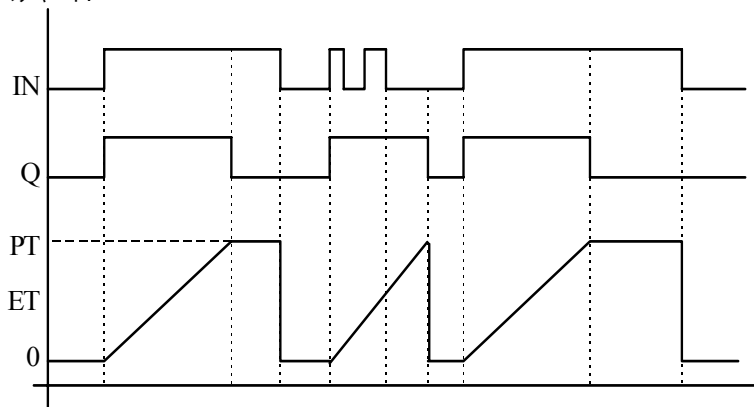
引数:

IN	ブール型	立ち上がり検出でタイマ値のインクリメントを開始(インクリメント中でなければ)。もし、FALSE 状態でタイマ値が設定タイマ値が経過したときはタイマ値をリセット。タイマがインクリメント中は IN の変化は無視されます。
PT	タイマ型	設定タイマ値
Q	ブール型	TRUE のとき、タイマがインクリメント中
ET	タイマ型	現在経過タイマ値

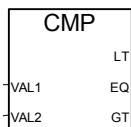
説明:

内部タイマ値を指定された値(PT)までインクリメント

タイムチャート:



CMP (比較)



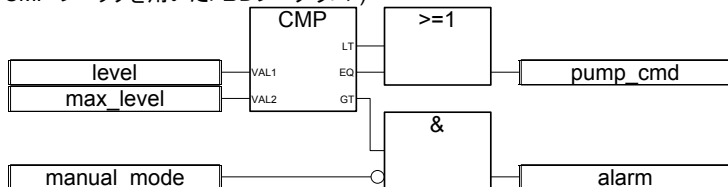
引数:

VAL1	整数型	符号付き整数
VAL2	整数型	符号付き整数
LT	ブール型	VAL1 < VAL2 のとき TRUE
EQ	ブール型	VAL1 = VAL2 のとき TRUE
GT	ブール型	VAL1 > VAL2 のとき TRUE

説明:

2つの整数型の値を比較します。3つの結果のいずれかをとります。(< , = , >)

(* "CMP" ブロックを用いたFBDプログラム *)



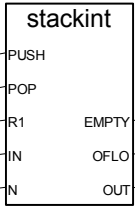
(* 等価なST言語: CMP1 は CMP ブロックのインスタンス名 *)

```
CMP1(level, max_level);
pump_cmd := CMP1.LT OR CMP1.EQ;
alarm := CMP1.GT AND NOT(manual_mode);
```

(* 等価なIL言語: *)

LD level
ST CMP1.val1
LD max_level
ST CMP1.val2
CAL CMP1
LD CMP1.LT
OR CMP1.EQ
ST pump_cmd
LD CMP1.GT
ANDN manual_mode
ST alarm

STACKINT (整数値のスタック)



引数:

PUSH	ブール型	プッシュコマンド: 立ち上がり検出により、 IN 値をスタックのトップに追加
POP	ブール型	ポップコマンド: 立ち上がり検出により、スタックの先頭(最後にプッシュされた値)を削除
R1	ブール型	スタックを空の状態にリセットする
IN	整数型	プッシュする値
N	整数型	スタックサイズ
EMPTY	ブール型	スタックが空の時 TRUE
OFLO	ブール型	オーバーフロー: スタックが一杯のとき TRUE
OUT	整数型	スタックの先頭にある値

説明:

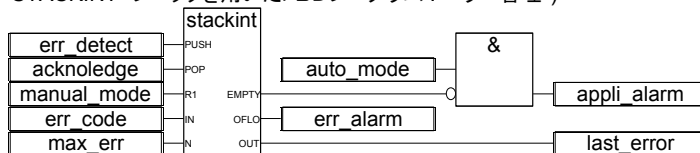
整数型値のスタックを管理します。

STACKINTファンクションブロックはPUSH、POP両入力に対して立ち上がりを検出します。**最大のスタックサイズは128**です。スタックサイズ**N**は**1から128**の値でなければなりません。

OFLO 値は一旦リセットされた後にのみ有効です(即ち、R1 は一度は TRUE にセットされた後に FALSE に戻っている必要があります)。

標準命令、ファンクションブロックとファンクション

(* "STACKINT" ブロックを用いたFBDプログラム：エラー管理*)



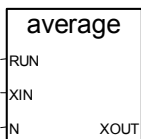
(*等価なST言語: STACKINT1 は STACKINT ブロックのインスタンス *)

```
STACKINT1(err_detect, acknowledge, manual_mode, err_code, max_err);
appli_alarm := auto_mode AND NOT(STACKINT1.EMPTY);
err_alarm := STACKINT1.OFLO;
last_error := STACKINT1.OUT;
```

(* 等価なIL言語: *)

```
LD      err_detect
ST      STACKINT1.push
LD      acknowledge
ST      STACKINT1.pop
LD      manual_mode
ST      STACKINT1.r1
LD      err_code
ST      STACKINT1.IN
LD      max_err
ST      STACKINT1.N
CAL     STACKINT1
LD      auto_mode
ANDN    STACKINT1.empty
ST      appli_alarm
LD      STACKINT1.OFLO
ST      err_alarm
LD      STACKINT1.OUT
ST      last_error
```

AVERAGE (移動平均)



引数:

RUN	ブール型	TRUE: 実行 / FALSE: リセット
XIN	実数型	実数値入力
N	整数型	平均を取るべきサンプル数(毎サイクル1入力)
XOUT	実数型	XIN 値のサンプル個数(N 個)の移動平均値

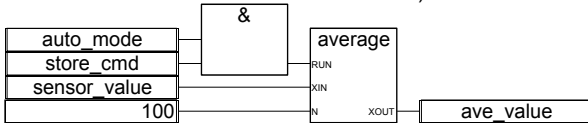
説明:

毎サイクル XIN を取り込み、過去のサンプル値(サンプル数 N)の平均値を求めます。
最新の N 個データが蓄えられています。

サンプル数**N**は**最大128**までです。"RUN"コマンドが **FALSE** の時(リセットモード)、出力値は入力値と同じ値になります。

蓄えられたサンプル数が N になると、最初に蓄えられた値は削除されます。

(* "AVERAGE"ブロックを用いたFBDプログラム*)



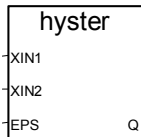
(*等価なST言語:AVERAGE1 は AVERAGE ブロックのインスタンス *)

```
AVERAGE1((auto_mode & store_cmd), sensor_value, 100);
ave_value := AVERAGE1.XOUT;
```

(* 等価なIL言語: *)

```
LD      auto_mode
AND     store_cmd
ST      AVERAGE1.run
LD      sensor_value
ST      AVERAGE1.xin
LD      100
ST      AVERAGE1.N
CAL     AVERAGE1
LD      AVERAGE1.XOUT
ST      ave_value
```

HYSTER (ヒステリシス)



引数:

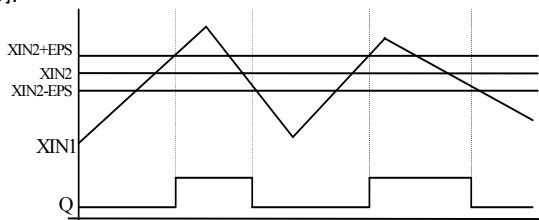
XIN1	実数型	実数
XIN2	実数型	実数 (XIN1 が XIN2+EPS を越えたかテストされる)
EPS	実数型	ヒステリシス値(>0)
Q	ブール型	XIN1 が XIN2+EPS を越えて、まだ、XIN2-EPS を下回っていない時、TRUE

説明:

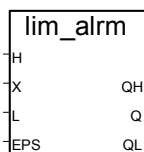
実数値の上限値のためのヒステリシス

標準命令、ファンクションブロックとファンクション

タイムチャート例:



LIM_ALARM (リミットアラーム)



引数:

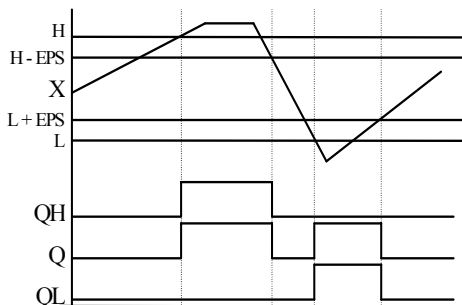
H	実数型	上限設定値
X	実数型	入力値
L	実数型	下限設定値
EPS	実数型	ヒステリシス値 (>0)
QH	ブール型	上限アラーム: X が上限値 H を越えている場合 TRUE
Q	ブール型	アラーム: 上下限を超えた場合、TRUE
QL	ブール型	下限アラーム: X が下限値 L を下回っている場合、 TRUE

説明:

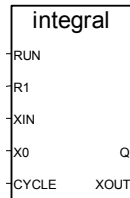
実数値に対する上下限に対するヒステリシス。

ヒステリシスが上限、下限リミットに与えられます。上限、下限に使われるヒステリシスデルタ量はEPSパラメータの半分になります。

タイムチャート:



INTEGRAL (積分)



引数:

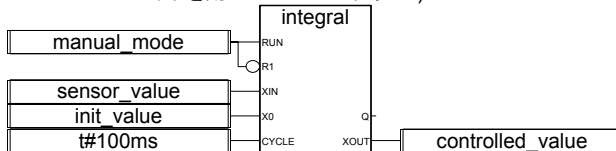
RUN	ブール型	モード、TRUE=積分 / FALSE=ホールド
R1	ブール型	上書きリセット
XIN	実数型	入力値
X0	実数型	初期値
CYCLE	タイマ型	サンプリング周期
Q	ブール型	R1 の反転
XOUT	実数型	積分結果

説明:

実数値を積分します。

指定したサンプリング周期毎に、入力 (XIN) の値に、前回サンプルしてから経過した時間 (ms 単位) をかけたものを前回の出力に加えて、XOUT から出力します。
 "CYCLE"パラメータ値がISaGRAFのサイクルタイムよりも短い場合はサンプリング間隔はサイクルタイムに合わせられます。

(*"INTEGRAL"ブロックを用いたFBDプログラム*)



(*等価なST言語: INTEGRAL1 は INTEGRAL ブロックのインスタンス *)

```

INTEGRAL1(manual_mode, NOT(manual_mode), sensor_value, init_value,
t#100ms);
controlled_value := INTEGRAL1.XOUT;

```

(* 等価なIL言語: *)

```

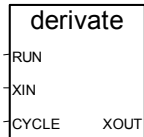
LD      manual_mode
ST      INTEGRAL1.run
STN     INTEGRAL1.R1
LD      sensor_value
ST      INTEGRAL1.XIN
LD      init_value
ST      INTEGRAL1.X0
LD      t#100ms
ST      INTEGRAL1.CYCLE
CAL     INTEGRAL1

```

標準命令、ファンクションブロックとファンクション

LD INTEGRAL1.XOUT
ST controlled_value

DERIVATE（微分）



引数:

RUN	ブール型	モード, TRUE=微分実行 / FALSE=リセット
XIN	実数型	入力値
CYCLE	タイマ型	サンプリング周期
XOUT	実数型	微分結果

説明:

実数値の微分を行います。

"CYCLE"パラメータ値がISaGRAFのサイクルタイムよりも短い場合はサンプリング間隔はサイクルタイムに合わせられます。

(*DERIVATE"ブロックを用いたFBDプログラム *)

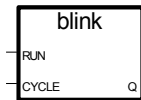


(*等価なST言語: DERIVATE1 は DERIVATE ブロックのインスタンス *)

```
DERIVATE1(manual_mode, sensor_value, t#100ms);  
derivated_value := DERIVATE1.XOUT;
```

(* 等価なIL言語: *)

```
LD        manual_mode  
ST        DERIVATE1.run  
LD        sensor_value  
ST        DERIVATE1.XIN  
LD        t#100ms  
ST        DERIVATE1.CYCLE  
CAL       DERIVATE1  
LD        DERIVATE1.XOUT  
ST        derivated_value
```

BLINK (ブリンク)

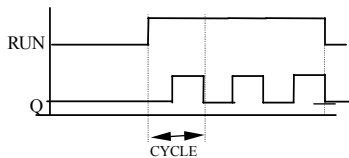
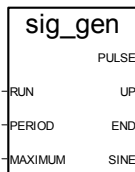
引数:

RUN	ブール型	モード、TRUE=ブリンク / FALSE=出力をリセット
CYCLE	タイマ型	ブリンク周期
Q	ブール型	ブリンクの出力信号

説明:

ブリンク信号を生成します。

タイムチャート:

**SIG_GEN (信号発生)**

引数:

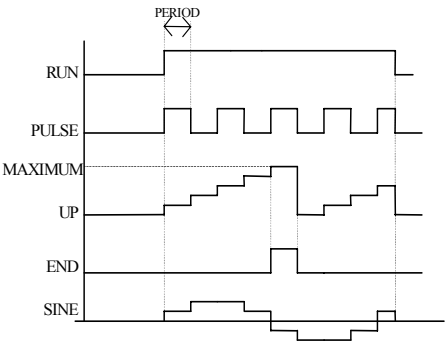
RUN	ブール型	モード TRUE=実行 / FALSE=FALSE にリセット
PERIOD	タイマ型	1 サンプルの時間 (パルスの幅)
MAXIMUM	整数型	最大カウント値
PULSE	ブール型	1 サンプル時間毎に反転します。
UP	整数型	1 サンプル時間毎のアップカウンタ
END	ブール型	アップカウンタの終了時、TRUE
SINE	実数型	正弦波形成信号 (波形半周期はアップカウント時間)

説明:

各種信号を同時発生します (ブール型矩形、整数型アップカウンタ、実数型正弦波)。

カウンタは最大値に達すると、0からリスタートします。END 信号は1PERIOD 分の時間だけ TRUE となります。

タイムチャート:



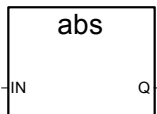
B.9.3 標準ファンクション

ISaGRAFでサポートしている標準のファンクションを以下に示します。これらは、標準で備わっているためにISaGRAFライブラリに登録する必要がありません。

数学計算:	ABS(絶対値) EXPT(指数関数), POW(べき乗) LOG(対数) SQRT(平方根) TRUNC(小数点以下切り捨て)
三角関数:	ACOS(アークコサイン), ASIN(アークサイン) ATAN(アークタンジェント) COS(コサイン), SIN(サイン) TAN(タンジェント)
レジスタコントロール:	ROL(左回転), ROR(右回転) SHL(左シフト), SHR(右シフト)
データ操作:	MIN(最小値), MAX(最大値) LIMIT(上下限) MOD(剰余) MUX4(マルチプレクサ(4 入力)) MUX8(マルチプレクサ(8 入力)) SEL(バイナリ選択) ODD(奇数パリティ) RAND(ランダム値)
データ変換:	ASCII(文字→アスキーコード変換) CHAR(アスキーコード→文字変換)
文字列操作:	MLen(文字列長) DELETE(文字列削除) INSERT(文字列挿入) FIND(文字列検索) REPLACE(文字列置換) LEFT(左側文字列抽出), MID(文字列抽出) RIGHT(右側文字列抽出) DAY_TIME(日時)
配列操作:	ARCREATE(整数配列の作成) ARREAD(配列要素読み出し) ARWRITE(配列要素書き出し)
バイナリファイル管理:	F_OPEN(読み出しモードでファイルオープン) F_WOPEN(書き込みモードでファイルオープン) F_CLOSE(ファイルのクローズ) F_EOF(ファイルの最終検出) FA_READ(ファイルからの整数値読み出し) FA_WRITE(ファイルへの整数値書き込み) FM_READ(ファイルからの文字列読み出し) FM_WRITE(ファイルへの文字列書き込み)

注意: これらのファンクションのうち、ターゲットによっては実装していないものもあります。

ABS（絶対値）



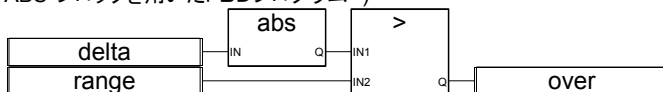
引数:

IN	実数型	実数値入力
Q	実数型	絶対値(常に0以上になります)

説明:

実数の絶対値を与えます。

(*ABS"ブロックを用いたFBDプログラム *)



(*等価なST言語: *)

over := (ABS (delta) > range);

(* 等価なIL言語: *)

```

LD      delta
ABS
GT      range
ST      over
  
```

EXPT（指数）



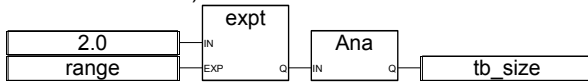
引数:

IN	実数型	符号付き実数
EXP	整数型	整数(指数部)
Q	実数型	(IN ^{EXP})

説明:

(base^{exponent})の演算結果を実数で与えます。'base' が最初の引数、'exponent' が2番目の引数で整数です。

(*EXPT"ブロックの例*)



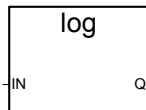
(*等価なST言語:*)

tb_size := ANA (EXPT (2.0, range));

(* 等価なIL言語: *)

```
LD      2.0
EXPT    range
ANA
ST      tb_size
```

LOG (常用対数)



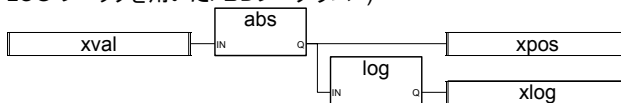
引数:

IN	実数型	0より大きい実数値
Q	実数型	入力値の常用対数(10を底とします)

説明:

実数入力の常用対数を与えます。

(*LOG"ブロックを用いたFBDプログラム *)



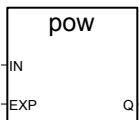
(*等価なST言語:*)

```
xpos := ABS (xval);
xlog := LOG (xpos);
```

(* 等価なIL言語: *)

```
LD      xval
ABS
ST      xpos
LOG
ST      xlog
```

POW (べき乗数)



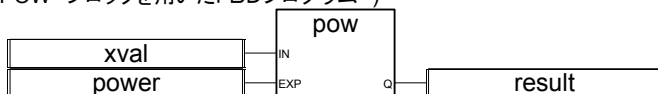
引数:

IN	実数型	実数値(非演算数)
EXP	実数型	実数値(べき数)
Q	実数型	(IN ^{EXP})
		1.0 : IN <> 0.0、EXP = 0.0 の場合
		0.0 : IN = 0.0、EXP < 0.0 の場合
		0.0 : IN = 0.0、EXP = 0.0 の場合
		0.0 : IN < 0 の場合

説明:

実数のべき乗(base^{exponent})を与えます。'base' が最初の引数、'exponent' が2番目の引数で実数です。

(* "POW" ブロックを用いたFBDプログラム *)



(*等価なST言語:*)

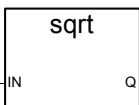
result := POW (xval, power);

(* 等価なIL言語: *)

```

LD    xval
POW   power
ST    result
  
```

SQRT (平方根)



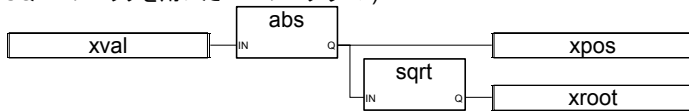
引数:

IN	実数型	実数値(>= 0)
Q	実数型	入力値の平方根

説明:

実数値の平方根を計算します。

(* "SQRT"ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

xpos := ABS (xval);

xroot := SQRT (xpos);

(* 等価なIL言語: *)

LD xval

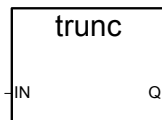
ABS

ST xpos

SQRT

ST xroot

TRUNC (小数点以下切り捨て)



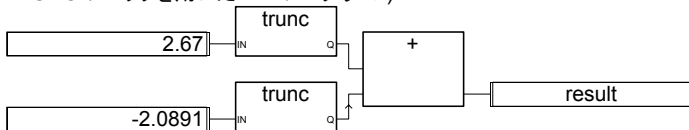
引数:

IN	実数型	実数値
Q	実数型	IN>0, 入力値以下の最大の整数値 IN<0, 入力値以上の最小の整数値

説明:

実数値の整数部分を与えます。

(* "TRUNC"ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

result := TRUNC (+2.67) + TRUNC (-2.0891);

(* 意味: result := 2.0 + (-2.0) := 0.0; *)

(* 等価なIL言語: *)

LD 2.67

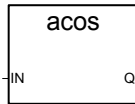
TRUNC

ST temporary (* 1個目の TRUNC の結果 *)

標準命令、ファンクションブロックとファンクション

LD	-2.0891
TRUNC	
ADD	temporary
ST	result

ACOS (アークコサイン)



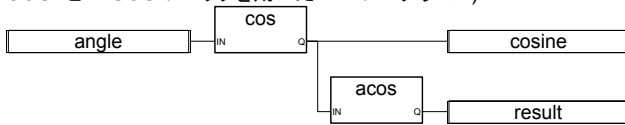
引数:

IN	実数型	-1.0 ~ +1.0 でなければなりません
Q	実数型	入力値のアークコサイン[0.0 ~ π] = 0.0 : 不正な入力値の時

説明:

実数値のアークコサインを計算します。

(* "COS" と "ACOS" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

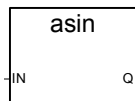
cosine := COS (angle);

result := ACOS (cosine); (* result は angle に等しい *)

(* 等価なIL言語: *)

```
LD      angle
COS
ST      cosine
ACOS
ST      result
```

ASIN (アークサイン)



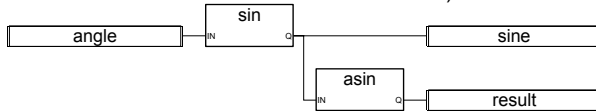
引数:

IN	実数型	-1.0 ~ +1.0 でなければなりません
Q	実数型	入力値のアークサイン[- $\pi/2$ ~ + $\pi/2$] = 0.0 : 不正な入力値の時

説明:

実数値のアークサインを計算します。

(* "SIN" と "ASIN" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

sine := SIN (angle);

result := ASIN (sine); (* result は angle に等しい *)

(* 等価なIL言語: *)

LD angle

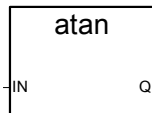
SIN

ST sine

ASIN

ST result

ATAN (アークタンジェント)



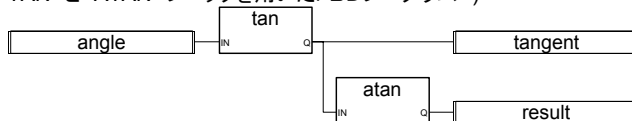
引数:

IN	実数型	実数値
Q	実数型	入力値のアークタンジェント[- $\pi/2 \sim +\pi/2$] = 0.0: 不正な入力値の時

説明:

実数値のアークタンジェントを計算します。

(* "TAN" と "ATAN" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

tangent := TAN (angle);

result := ATAN (tangent); (* result は angle に等しい *)

(* 等価なIL言語: *)

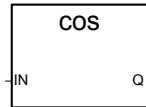
LD angle

TAN

標準命令、ファンクションブロックとファンクション

ST tangent
ATAN
ST result

COS（コサイン）



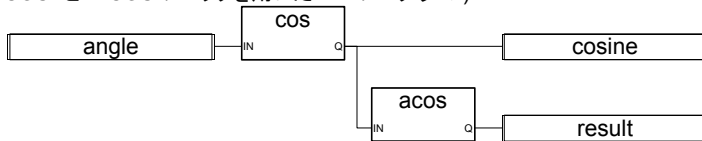
引数:

IN	実数型	実数値
Q	実数型	入力値のコサイン[-1.0 ~ +1.0]

説明:

実数値のコサインを計算します。

(* "COS" と "ACOS" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

cosine := COS (angle);

result := ACOS (cosine); (* result は angle と等しい *)

(* 等価なIL言語: *)

LD angle

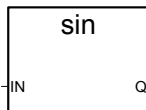
COS

ST cosine

ACOS

ST result

SIN（サイン）



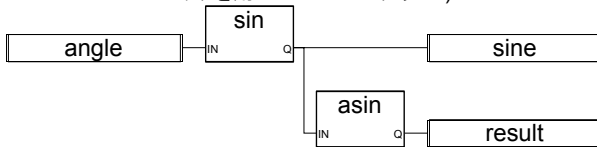
引数:

IN	実数型	実数値
Q	実数型	入力値のサイン[-1.0 ~ +1.0]

説明:

実数値のサインを計算します。

(* "SIN" と "ASIN" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

sine := SIN (angle);

result := ASIN (sine); (* result は angle と等しい *)

(* 等価なIL言語: *)

LD angle

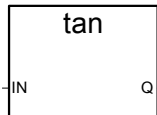
SIN

ST sine

ASIN

ST result

TAN (タンジェント)



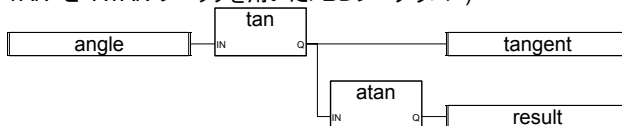
引数:

IN	実数型	π で割った余りが $\pi/2$ であってはなりません。
Q	実数型	入力値のタンジェント。 不正な入力に対しては、1E+38。

説明:

実数値のタンジェントを計算します。

(* "TAN" と "ATAN" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

tangent := TAN (angle);

result := ATAN (tangent); (* result は angle と等しい *)

(* 等価なIL言語: *)

LD angle

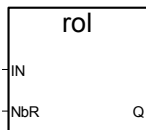
TAN

ST tangent

ATAN

ST result

ROL (左回転)

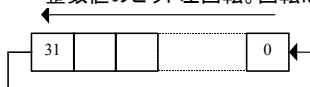


引数:

IN	整数型	整数値
NbR	整数型	回転させるビット数 [1~31]
Q	整数型	左回転した結果
nb_rotation <= 0 の時、変化無し		

説明:

整数値のビット左回転。回転は32ビットで行われます。



(* "ROL"ブロックを用いたFBDプログラム *)



(*等価なST言語:*)

result := ROL (register, 1);

(* register = 2#0100_1101_0011_0101*)

(* result = 2#1001_1010_0110_1010*)

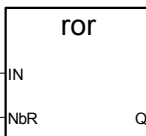
(* 等価なIL言語: *)

LD register

ROL 1

ST result

ROR (右回転)

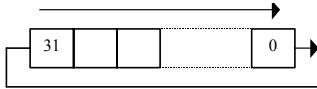


引数:

IN	整数型	整数値
NbR	整数型	回転させるビット数 [1~31]
Q	整数型	回転した結果
nb_rotation <= 0 の時、変化無し		

説明:

整数値のビット右回転。回転は32ビットで行われます。



(* "ROR" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

result := ROR (register, 1);

(* register = 2#0100_1101_0011_0101 *)

(* result = 2#1010_0110_1001_1010 *)

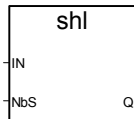
(* 等価なIL言語: *)

LD register

ROR 1

ST result

SHL (左シフト)

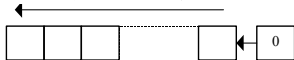


引数:

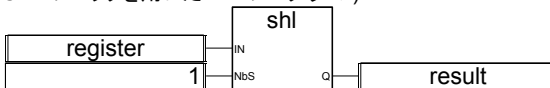
IN	整数型	整数値
NbS	整数型	ビットシフト数 [1~31]
Q	整数型	左シフトした結果
		nb_shift <= 0 の時、変化無し。
		最下位ビットには 0 が挿入されます。

説明:

整数値のビット表現で左シフトを行います。32ビット単位で行います。



(* "SHL" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

result := SHL (register, 1);

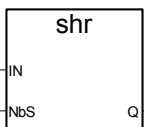
標準命令、ファンクションブロックとファンクション

```
(* register = 2#0100_1101_0011_0101 *)
(* result  = 2#1001_1010_0110_1010 *)
```

(* 等価なIL言語: *)

```
LD      register
SHL     1
ST      result
```

SHR (右シフト)



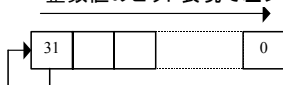
引数:

IN	整数型	整数値
NbS	整数型	ビットシフト数 [1~31]
Q	整数型	右シフトした結果

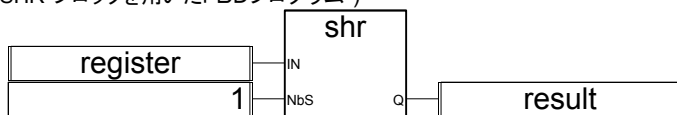
nb_shift <= 0 の時、変化無し。
最上位ビットは変化しません。

説明:

整数値のビット表現で左シフトを行います。32ビット単位で行います。



(* "SHR"ブロックを用いたFBDプログラム*)

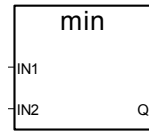


(*等価なST言語:*)

```
result := SHR (register,1);
(* register = 2#1100_1101_0011_0101 *)
(* result  = 2#1110_0110_1001_1010 *)
```

(* 等価なIL言語: *)

```
LD      register
SHR     1
ST      result
```


MIN (最小値)

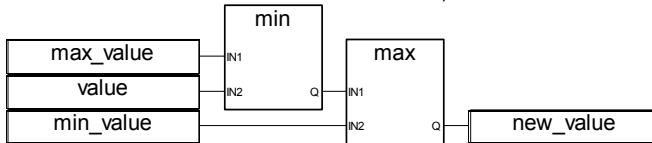
引数:

IN1	整数型	整数値
IN2	整数型	整数値
Q	整数型	2つの入力の最小値

説明:

2つの整数値の最小値を与えます。

(*"MIN" と "MAX"ブロックを用いたFBDプログラム*)



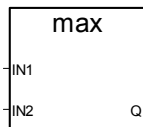
(*等価なST言語:*)

```
new_value := MAX (MIN (max_value, value), min_value);
```

(* 値を min_value～max_value の範囲に収める *)

(* 等価なIL言語: *)

```
LD      max_value
MIN     value
MAX     min_value
ST      new_value
```

MAX (最大値)

引数:

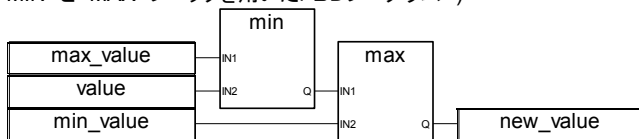
IN1	整数型	整数値
IN2	整数型	整数値
Q	整数型	2つの入力の最大値

説明:

2つの整数値の最大値

標準命令、ファンクションブロックとファンクション

(* "MIN" と "MAX" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

```
new_value := MAX (MIN (max_value, value), min_value);
```

(* 値を min_value ~ max_value の範囲に収める *)

(* 等価なIL言語: *)

```
LD      max_value
MIN      value
MAX      min_value
ST      new_value
```

LIMIT (上下限)



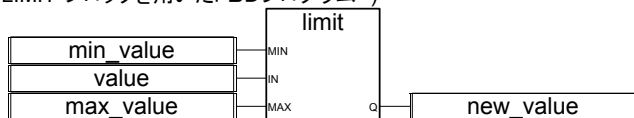
引数:

MIN	整数型	最小値設定値
IN	整数型	整数入力値
MAX	整数型	最大値設定値
Q	整数型	入力値を最大、最小値の範囲内に収めたもの

説明:

入力値に上下限を設けます。入力値が最大値と最小値の間の時は入力値のまま。最大値を超えたときは最大値に、最小値を下回った時は最小値になります。

(* "LIMIT" ブロックを用いたFBDプログラム *)



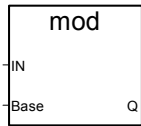
(* 等価なST言語: *)

```
new_value := LIMIT (min_value, value, max_value);
```

(* 値を min_value ~ max_value の範囲に収める *)

(* 等価なIL言語: *)

```
LD      min_value
LIMIT   value, max_value
ST      new_value
```

MOD (剰余)

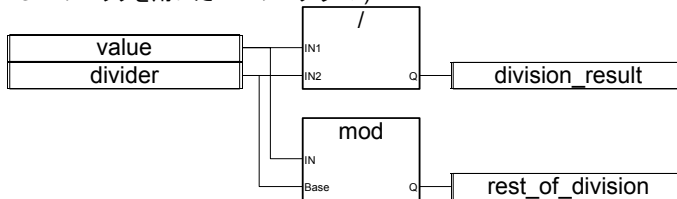
引数:

IN	整数型	入力値
Base	整数型	入力値 (0より大きい値)
Q	整数型	剰余算の計算結果
		Base <= 0 の時、-1

説明:

整数値の剰余を計算します。

(*"MOD" ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

division_result := (value / divider); (* 整数の除算*)

rest_of_division := MOD (value, divider); (* 余り*)

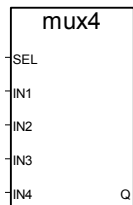
(* 等価なIL言語: *)

```

LD      value
DIV     divider
ST      division_result
LD      value
MOD     divider
ST      rest_of_division

```

MUX4 (マルチプレクサ)



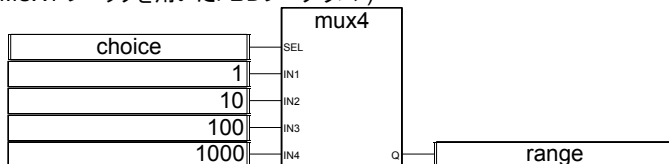
引数:

SEL	整数型	セレクト入力値 [0～3]
IN1..IN4	整数型	入力値
Q	整数型	= SEL = 0 のとき IN1 = SEL = 1 のとき IN2 = SEL = 2 のとき IN3 = SEL = 3 のとき IN4 = SEL が上記以外するとき 0

説明:

4点入力用マルチプレクサ: 4つの整数入力値の中から1つを選択します。

(* "MUX4" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

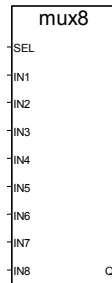
```
range := MUX4 (choice, 1, 10, 100, 1000);
```

(* 4つの入力から1つを選択する。choice が 1 のときは range は 10 *)

(* 等価なIL言語: *)

```
LD      choice
MUX4    1,10,100,1000
ST      range
```

MUX8 (マルチプレクサ)



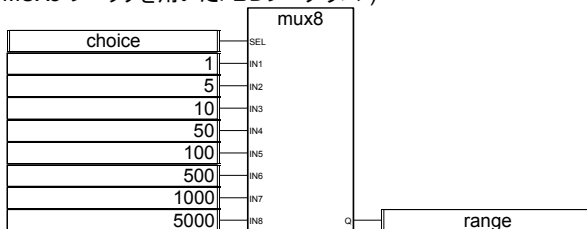
引数:

SEL	整数型	セレクト入力値 [0~7]
IN1..IN4	整数型	入力値
Q	整数型	= SEL = 0 のとき IN1 = SEL = 1 のとき IN2 = SEL = 2 のとき IN3 = SEL = 7 のとき IN8 = SEL が上記以外のとき 0

説明:

8点入力用マルチプレクサ: 8つの整数入力値の中から1つを選択します。

(* "MUX8"ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

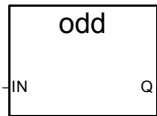
```
range := MUX8 (choice, 1, 5, 10, 50, 100, 500, 1000, 5000);
```

(* 8つの入力から1つを選択する。choice が 3 のときは range は 50 *)

(* 等価なIL言語: *)

```
LD      choice
MUX8    1,5,10,50,100,500,1000,5000
ST      range
```

ODD (奇数パリティ)



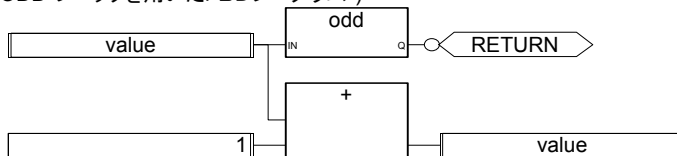
引数:

IN	整数型	整数入力
Q	ブール型	TRUE: 入力値が奇数の時 FALSE: 入力値が偶数の時

説明:

整数値のパリティチェック: 奇数か偶数かを判断します。

(* "ODD"ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

If Not (ODD (value)) Then Return; End_if;

value := value + 1;

(* value は常に偶数*)

(* 等価なIL言語: *)

LD value

ODD

RETNC

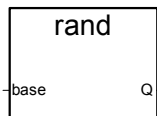
LD value

ADD

1

ST value

RAND (ランダム値)



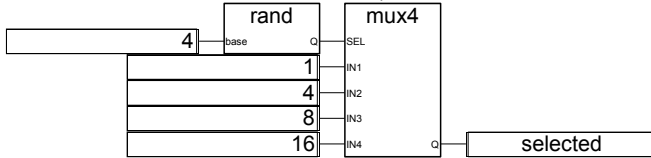
引数:

base	整数型	ランダム値の最大値
Q	整数型	[0~base-1] のランダム値

説明:

整数値で指定した範囲のランダム値を与えます。

(*RAND" ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

```
selected := MUX4 ( RAND (4), 1, 4, 8, 16 );
```

(*

発生するランダム値によって、4つの入力からの1つを任意に選択する。

RAND で発生される数値は [0~3]。結果として MUX4 の 'selected' には以下の出力がでます。

1: RAND 出力が 0,

4: RAND 出力が 1

8: RAND 出力が 2

16: RAND 出力が 3

*)

(* 等価なIL言語: *)

```
LD      4
RAND
MUX4    1,4,8,16
ST      selected
```

SEL (バイナリ選択)



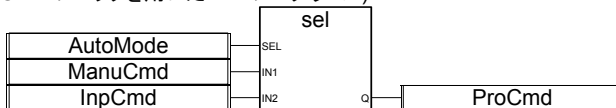
引数:

SEL	ブール型	選択用
IN1, IN2	整数型	整数値
Q	整数型	= IN1: SEL が FALSE のとき = IN2: SEL が TRUE のとき

説明:

バイナリ選択: 2つの整数値から1つを選択します。

(*SEL" ブロックを用いたFBDプログラム *)



(*等価なST言語:*)

```
ProcCmd := SEL (AutoMode, ManuCmd, InpCmd);
```

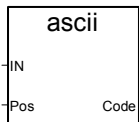
標準命令、ファンクションブロックとファンクション

(* 処理するコマンドを選択*)

(* 等価なIL言語: *)

LD	AutoMode
SEL	ManuCmd, InpCmd
ST	ProCmd

ASCII (文字→アスキーコード変換)



引数:

IN	文字列型	空でない文字列
Pos	整数型	変換する文字の文字列中の位置 [1 ~ len] (len は文字列 IN の長さ)
Code	整数型	選択した文字のアスキーコード [0 ~ 255] Pos が文字列長を越えていた場合、0

説明:

文字列中で指定の文字に対するアスキーコードを与えます。

(* "ASCII" ブロックを用いたFBDプログラム *)



(* 等価なST言語: *)

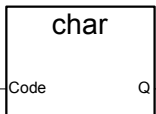
FirstChr := ASCII (message, 1);

(* FirstChr は文字列の1文字目のアスキーコード *)

(* 等価なIL言語: *)

LD	message
ASCII	1
ST	FirstChr

CHAR (アスキーコード→文字変換)



引数:

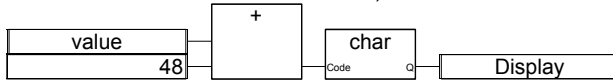
Code	整数型	アスキーコード [0 ~ 255]
Q	文字列型	1文字の文字列。

文字はアスキーコード (Code) に対応するものになります。アスキーコードは 256 で割った余りを使います。

説明:

与えられたアスキーコードに対応する文字を与えます。

(* "CHAR" ブロックを用いた FBD プログラム *)



(* 等価な ST 言語: *)

Display := CHAR (value + 48);

(* value は 0 ~ 9 *)

(* 48 は '0' のアスキーコード *)

(* 結果は '0' ~ '9' の文字になります *)

(* 等価な IL 言語: *)

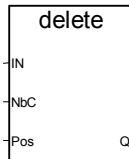
LD value

ADD 48

CHAR

ST Display

DELETE (文字列削除)



引数:

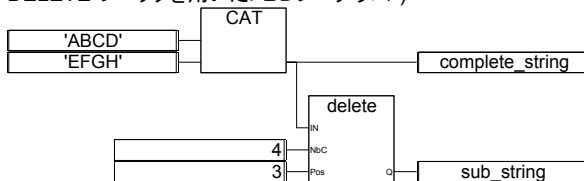
IN	文字列型	空でない文字列
NbC	整数型	文字列から削除する文字数
Pos	整数型	削除する最初の文字列の位置 (文字列の先頭の位置は 1)
Q	文字列型	指定文字列が削除された結果の文字列 Pos < 1 のとき、空文字列 Pos > IN の文字列のとき、最初の文字 NbC <= 0 のとき、最初の文字

説明:

文字列の一部を削除します。

標準命令、ファンクションブロックとファンクション

(* "DELETE" ブロックを用いた FBD プログラム *)



(* 等価な ST 言語: *)

complete_string := 'ABCD' + 'EFGH'; (* complete_string は 'ABCDEFGH' *)

sub_string := DELETE (complete_string, 4, 3); (* sub_string は 'ABGH' *)

(* 等価な IL 言語: *)

```
LD      'ABCD'
ADD     'EFGH'
ST      complete_string
DELETE  4,3
ST      sub_string
```

FIND (文字列検索)



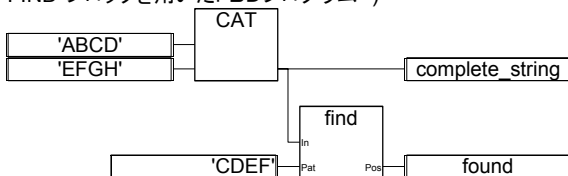
引数:

In	文字列型	入力文字列
Pat	文字列型	検索する文字列パターン。空でないこと。
Pos	整数型	= 0: 文字列パターンが見つからないとき = 検索する文字列が最初に見つかった位置 (1文字目であれば 1) 大文字・小文字の区別をします。

説明:

文字列の中から指定した文字列を検索して、見つかった場合はその場所を返します。

(* "FIND" ブロックを用いた FBD プログラム *)



(* 等価な ST 言語: *)

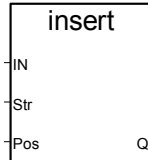
complete_string := 'ABCD' + 'EFGH'; (* complete_string は 'ABCDEFGH' *)

found := FIND (complete_string, 'CDEF'); (* found は 3 *)

(* 等価なIL言語: *)

```
LD      'ABCD'
ADD     'EFGH'
ST      complete_string
FIND    'CDEF'
ST      found
```

INSERT (文字列挿入)



引数:

IN	文字列型	元になる文字列
Str	文字列型	挿入する文字列
Pos	整数型	挿入位置。挿入文字列は番号の前になります。文字列の先頭は 1 です。
Q	文字列型	挿入した結果の文字列

Pos ≤ 0 のとき、空の文字列
Pos が文字列 IN の長さを越えるとき、2つの文字列を連結したことになります。

説明:

文字列の指定した位置に他の文字列を挿入します。

(* "INSERT"ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

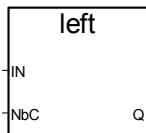
```
MyName := INSERT ('Mr JONES', 'Frank', 4);
```

```
(* MyName : 'Mr Frank JONES' *)
```

(* 等価なIL言語: *)

```
LD      'Mr JONES'
INSERT  'Frank',4
ST      MyName
```

LEFT (左側文字列抽出)



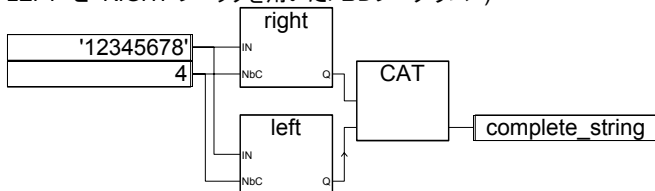
引数:

IN	文字列型	空でない文字列
NbC	整数型	取り出す文字数 (文字列 IN の長さ以下であること)
Q	文字列型	文字列 IN の左側文字列(文字列長は NbC) NbC <= 0 のとき、空の文字列。 NbC >= IN の長さのとき、文字列 IN 全体。

説明:

元の文字列の左側から指定された文字数だけ取り出します。

(*"LEFT" と "RIGHT"ブロックを用いたFBDプログラム *)



(*等価なST言語:*)

complete_string := RIGHT ('12345678', 4) + LEFT ('12345678', 4);

(* complete_string : '56781234'

RIGHT で取り出される文字列は '5678'

LEFT で取り出される文字列は '1234'

*)

(* 等価なIL言語: 先に LEFT をコールします。*)

LD '12345678'

LEFT 4

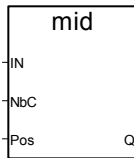
ST sub_string (* 中間結果 *)

LD '12345678'

RIGHT 4

ADD sub_string

ST complete_string

MID (文字列抽出)

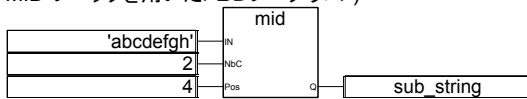
引数:

IN	文字列型	空でない文字列
NbC	整数型	取り出す文字数 (文字列 IN の長さ以下であること)
Pos	整数型	取り出す位置 (先頭は 1)
Q	文字列型	取り出した文字列 (文字列長は NbC)。 指定した引数が不正のとき、空の文字列。

説明:

指定した位置から指定した長さの文字列を取り出します。

(*MID"ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

```
sub_string := MID ('abcdefgh', 2, 4);
```

```
(* sub_string : 'de' *)
```

(*等価なIL言語: *)

```
LD      'abcdefgh'
```

```
MID     2,4
```

```
ST      sub_string
```

MLEN (文字列長)

引数:

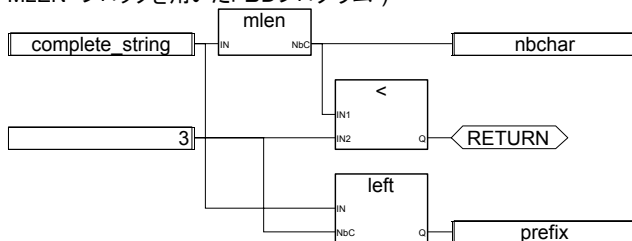
IN	文字列型	文字列
NbC	整数型	文字列 IN の文字列長

説明:

文字列の文字列長を計算します。

標準命令、ファンクションブロックとファンクション

(* "MLEN" ブロックを用いたFBDプログラム *)



(*等価なST言語: *)

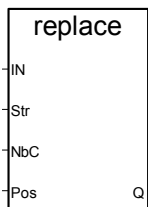
```
nbchar := MLEN (complete_string);
If (nbchar < 3) Then Return; End_if;
prefix := LEFT (complete_string, 3);
```

(* このプログラムは文字列から左側3文字を抽出して、prefix 文字列に入れます。もし、文字列長が3未満のときは何もしません。 *)

(*等価なIL言語: *)

```
LD      complete_string
MLEN
ST      nbchar
LT      3
RETC
LD      complete_string
LEFT   3
ST      prefix
```

REPLACE (文字列置換)



引数:

IN	文字列型	もとの文字列
Str	文字列型	挿入する文字列 (NbC 文字ぶんを置き換える)
NbC	整数型	削除される文字数
Pos	整数型	置換する最初の文字の位置 (先頭の位置は 1)
Q	文字列型	置換した結果の文字列 - まず、NbC 文字ぶんが Pos の場所から削除されます。 - 次に、文字列 Str がこの場所に挿入されます。 Pos <= 0 のとき、空の文字列になります。

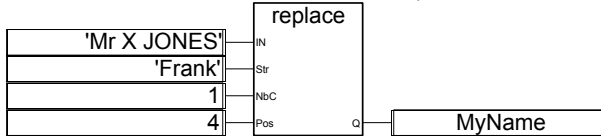
Pos > 文字列 IN の長さのとき、2つの文字列の結合
(IN + Str)したものになります。

NbC <= 0 のとき、元の文字列になります。

説明:

文字列の一部を別の文字列で置換します。

(*"REPLACE" ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

MyName := REPLACE ('Mr X JONES', 'Frank', 1, 4);

(* MyName は 'Mr Frank JONES' になります *)

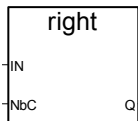
(*等価なIL言語: *)

LD 'Mr X JONES'

REPLACE 'Frank',1,4

ST MyName

RIGHT (右側文字列抽出)



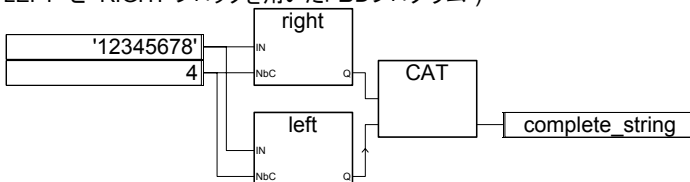
引数:

IN	文字列型	空でない文字列
NbC	整数型	取り出す文字数(文字列 IN の長さ以下)
Q	文字列型	文字列 IN の右側文字列(文字列長は NbC)
		NbC <= 0 のとき、空の文字列。
		NbC >= 文字列 IN の長さのとき、IN 全体。

説明:

文字列の右側から指定された文字数だけ抜き出します。

(*"LEFT" と "RIGHT"ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

```
complete_string := RIGHT ('12345678', 4) + LEFT ('12345678', 4);
```

(* complete_string は '56781234'

RIGHT で抽出される文字列 '5678'

LEFT で抽出される文字列 '1234'

*)

(*等価なIL言語: First done is call to LEFT *)

```
LD      '12345678'
```

```
LEFT    4
```

```
ST      sub_string (* 中間結果*)
```

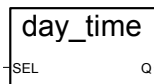
```
LD      '12345678'
```

```
RIGHT   4
```

```
ADD     sub_string
```

```
ST      complete_string
```

DAY_TIME (日付と時刻)



引数:

SEL

整数型

出力の選択

0= 現在の日付

1= 現在の時刻

2= 曜日

Q

文字列型

日時文字列

SEL = 0 のとき、'YYYY/MM/DD'

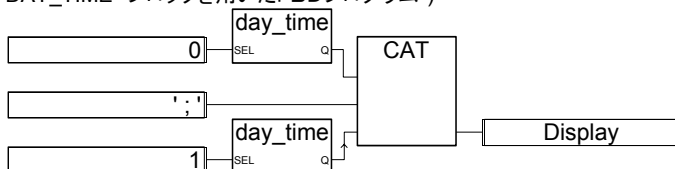
SEL = 1 のとき、'HH:MM:SS'

SEL = 2 のとき、曜日 (例: 'Monday')

説明:

日時、曜日を文字列で与えます。

(* "DAY_TIME" ブロックを用いたFBDプログラム*)



(*等価なST言語:*)

```
Display := Day_Time (0) + ' '; ' + Day_Time (1);
```

(* Display の書式は: 'YYYY/MM/DD ; HH:MM:SS'になります *)

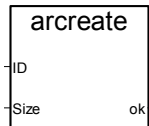
(*等価なIL言語: 先に day_time(1)をコールします *)


```

LD      1
DAY_TIME
ST      hour_str    (* 中間結果 *)
LD      0
DAY_TIME
ADD     ' ; '
ADD     hour_str
ST      Display

```

ARCREATE (配列作成)



引数:

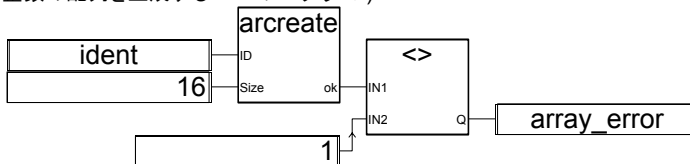
ID	整数型	配列識別子 (0 ~ 15)
Size	整数型	配列の要素数
ok	整数型	実行結果:
		=1 正常に配列が作成されたとき
		=2 不正な配列識別子、あるいは、既に配列作成済みのとき
		=3 不正な配列要素数
		=4 メモリ不足

説明:

整数型の配列を作成します。

注意: アプリケーションには**最大16**の配列(array)を定義できます。配列の要素は**整数値**をもちます。メモリの割り当ては動的に行われるため、配列サイズが利用可能なメモリ量に近い場合はシステムエラーを引き起こすことがあります。

(* 整数の配列を生成する FBD プログラム*)



(*等価なST言語:*)

```
array_error := (ARCREATE (ident, 16) <> 1));
```

(*等価なIL言語:*)

```

LD      ident
ARCREATE  16
NE      1
ST      array_error

```

ARREAD (配列要素の読み出し)



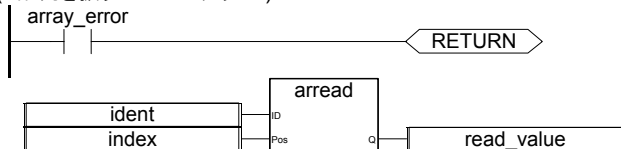
引数:

ID	整数型	配列識別子 (0 ~ 15)
Pos	整数型	配列内の要素の位置 (0 ~ size-1)
value	整数型	要素の値 =0 引数が不正

説明:

整数配列から指定した要素を読み出します。

(* 配列を扱うFBDプログラム*)



(*等価なST言語:*)

```
If (array_error) Then Return; End_if;
read_value := ARREAD (ident, index);
(* array_error は ARCREATE が返します*)
```

(*等価なIL言語:*)

```
LD      array_error
RETC
LD      ident
ARREAD  index
ST      read_value
```

ARWRITE (配列要素の書き込み)



引数:

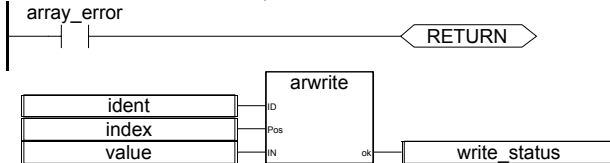
ID	整数型	配列識別子(0 ~ 15)
Pos	整数型	配列の要素の位置 (0 ~ size-1)

IN	整数型	要素に書き込む値
ok	整数型	実行結果 =1 書き込みが正常終了 =2 不正な配列識別子 =3 不正な位置指定

説明:

整数値の配列の要素に値の書き込み

(* 配列を扱うFBDプログラム*)



(*等価なST言語:*)

```
If (array_error) Then Return; End_if;
write_status := ARWRITE (Ident, Index, value);
(* array_error は ARCREATE が返します*)
```

(*等価なIL言語:*)

```
LD      array_error
RETC
LD      ident
ARWRITE index,value
ST      write_status
```

F_ROPEN (ファイルをリードモードでオープン)



引数::

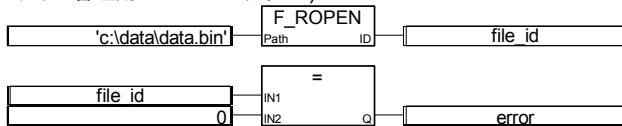
Path	文字列型	ファイル名 ファイル名にはディレクトリ名を示す記号 (/または\、 ¥)を含んでも構いません。移植性のため、/や \(¥)は同一のものと見なされます。
ID	整数型	ファイル番号 = 0 エラー発生時(ファイルが見つからなかったなど)

説明::

バイナリファイルをリードモードで開きます。FX_READ、F_CLOSE と組み合わせて使います。このファンクションは ISaGRAF のシミュレータ機能には含まれません。

標準命令、ファンクションブロックとファンクション

(* ファイル管理用のFBDプログラム*)



(* 等価なST言語:: *)

```
file_id := F_OPEN('c:\data \data.bin');
```

```
error := (file_id=0);
```

(* 等価なIL言語:: *)

```
LD      'c:\data\data.bin'
```

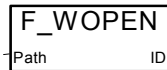
```
F_OPEN
```

```
ST      file_id
```

```
EQ      0
```

```
ST      error
```

F_WOPEN (ファイルをライトモードでオープン)



引数::

Path

文字列型

ファイル名

ファイル名にはディレクトリ名を示す記号(/または\、¥)を含んでいても構いません。移植性のため、/や\¥)は同一のものと見なされます。

ID

整数型

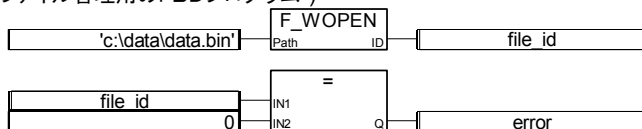
ファイル番号

= 0 エラー発生時(ファイルが見つからなかったなど)

説明::

バイナリファイルをライトモードで開きます。FX_READ、F_CLOSE と組み合わせて使います。このファンクションは ISaGRAF のシミュレータ機能には含まれません。

(* ファイル管理用のFBDプログラム*)



(* 等価なST言語:: *)

```
file_id := F_WOPEN('c:\data \data.bin');
```

```
error := (file_id=0);
```

(* 等価なIL言語:: *)

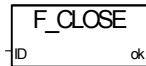
```
LD      'c:\data\data.bin'
```

```
F_WOPEN
```

```
ST      file_id
```

```
EQ      0
```

```
ST      error
```

F_CLOSE (ファイルのクローズ)

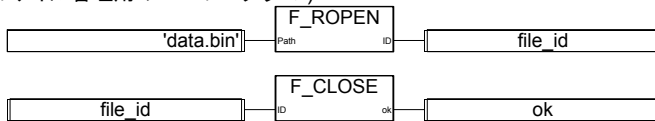
引数::

ID	整数型	ファイル番号 (F_ROPEN、F_WOPEN の戻り値)
ok	ブール型	戻り値 =TRUE: 正常にクローズ =FALSE: エラー発生

説明::

F_WOPEN、F_ROPEN で開いたバイナリファイルをクローズします。このファンクションは ISaGRAF のシミュレータ機能には含まれません。

(* ファイル管理用のFBDプログラム*)

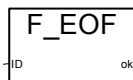


(* 等価なST言語:: *)

```
file_id := F_ROPEN('data.bin');
ok := F_CLOSE(file_id);
```

(* 等価なIL言語:: *)

```
LD      'data.bin'
F_ROPEN
ST      file_id
F_CLOSE      (* file_id は IL レジスタに格納済み*)
ST      ok
```

F_EOF (ファイル終端の検出)

引数::

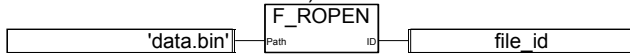
ID	整数型	ファイル番号 (F_ROPEN、F_WOPEN の戻り値)
ok	ブール型	ファイルの終端を示します。 =TRUE: ファイル読み出し、書き込みファンクションでファイルの最後に到達したとき。 なお、FM_READ で最後に読み出した文字が EOF でなければ、文字列は正しくない可能性があります。

説明::

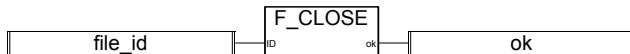
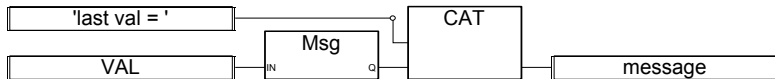
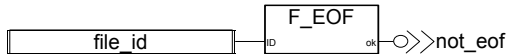
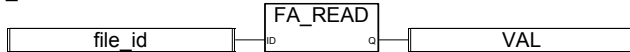
標準命令、ファンクションブロックとファンクション

ファイルの最後に到達したかチェックします。ISaGRAF のシミュレータには機能が含まれていません。

(* ファイル管理用のFBDプログラム*)



not_eof:



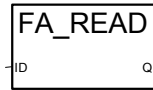
(* 等価なST言語:: *)

```
file_id := F_ROPEN('data.bin');
WHILE not(F_EOF(file_id))
    VAL := FA_READ(file_id);
END_WHILE;
MESSAGE := 'last val = ' + msg(VAL);
ok := F_CLOSE(file_id);
```

(* 等価なIL言語:: *)

```
LD      'data.bin'
F_ROPEN
ST      file_id
LD      file_id
F_EOF
JMPC    END_OF_FILE
NOT_EOF: LD      file_id
FA_READ
ST      VAL
LD      file_id
F_EOF
JMPNC   NOT_EOF (* EOF でなければ読み込む*)
END_OF_FILE: LD      VAL
MSG
ST      val_msg (* VAL を文字列に変換*)
LD      'last val = '
ADD     val_msg
ST      MESSAGE
LD      file_id
F_CLOSE
ST      ok
```

FA_READ (ファイルの読み込み)



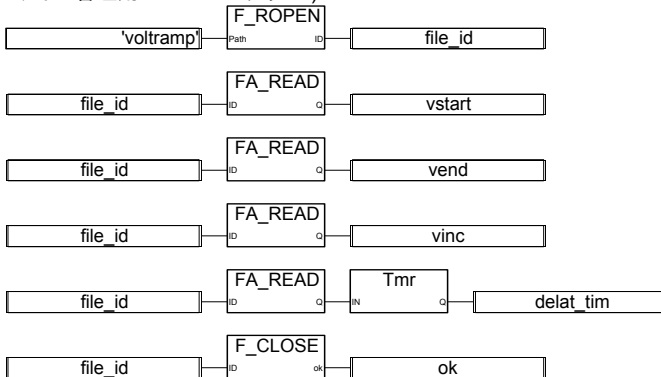
引数::

ID	整数型	ファイル番号 (F_ROPEN の戻り値).
Q	整数型	ファイルから読み出された整数値

説明::

バイナリファイルから整数値を読み出します。F_ROPEN、F_CLOSE と一緒に使います。前回アクセスした位置からシーケンシャルにファイルにアクセスします。F_ROPEN コールのための最初のコールではファイルの先頭の4バイトが読み出され、毎回のコールで読み出している場所をスタックにプッシュします。ファイルの最後に到達したかチェックするためには F_EOF を使います。このファンクションは ISaGRAF のシミュレータには含まれません。

(* ファイル管理用のFBDプログラム*)



(* 等価なST言語:: *)

```
file_id := F_ROPEN('voltramp.bin');
vstart := FA_READ(file_id);
vend := FA_READ(file_id);
vinc := FA_READ(file_id);
delta_tim := tmr(FA_READ(file_id));
ok := F_CLOSE(file_id);
```

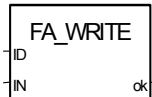
(* 等価なIL言語:: *)

```
LD      'voltramp.bin'
F_ROPEN
ST      file_id
FA_READ      (* vstart を読みこみ *)
ST      vstart
LD      file_id
FA_READ      (* vend を読みこみ *)
```

標準命令、ファンクションブロックとファンクション

ST	vend
LD	file_id
FA_READ	(* vinc を読みこみ *)
ST	vinc
LD	file_id
FA_READ	(* delta_tim を読みこみ *)
TMR	(* タイマ型に変換 *)
ST	delta_tim
LD	file_id
F_CLOSE	
ST	ok

FA_WRITE (ファイルの書き込み)



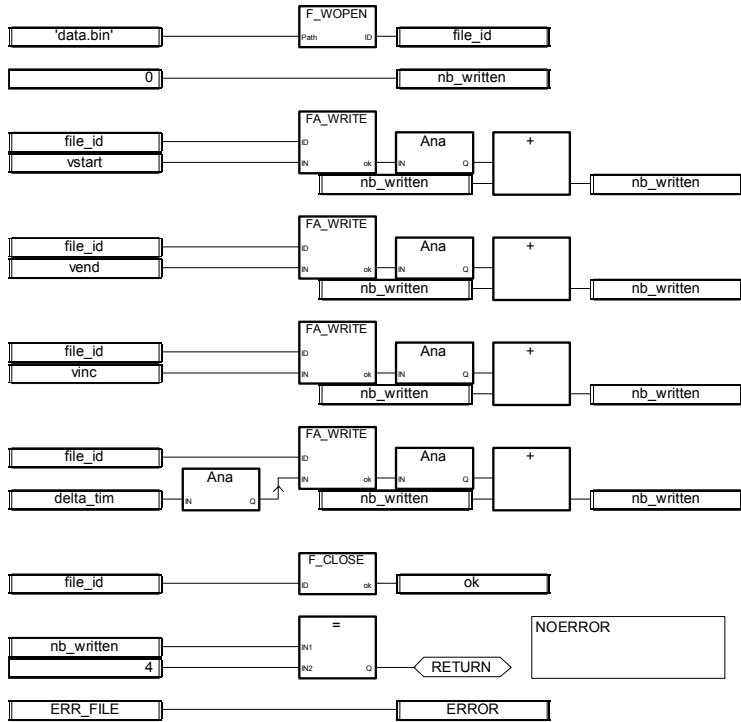
引数::

ID	整数型	ファイル番号 (F_WOPEN の戻り値)
IN	整数型	ファイルに書き込む整数値
OK	ブール型	実行結果 =TRUE: 正常終了

説明::

バイナリファイルに整数値を書き込みます。
前回アクセスした位置からシーケンシャル的にファイルにアクセスします。
F_WOPEN コールのための最初のコールではファイルの先頭の4バイトに書き込みます。
毎回のコールで読み出している場所をスタックにプッシュします。
ファイルの最後に到達したかチェックするためには F_EOF を使います。
このファンクションは ISaGRAF のシミュレータには含まれません。

(* FBDプログラム例 *)



(* 等価なST言語: *)

```

file_id := F_WOPEN('voltramp.bin');
nb_written := 0;
nb_written := nb_written + ana(FA_WRITE(file_id,vstart));
nb_written := nb_written + ana(FA_WRITE(file_id,vend));
nb_written := nb_written + ana(FA_WRITE(file_id,vinc));
nb_written := nb_written + ana(FA_WRITE(file_id,ana(delta_tim)));
ok := F_CLOSE(file_id);
IF ( nb_written <> 4) THEN
    ERROR := ERR_FILE;
END_IF;

```

(* 等価なIL言語: *)

```

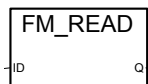
LD      'voltramp.bin'
F_WOPEN
ST      file_id
LD      0
ST      nb_written
LD      file_id      (* vstart を書きこみ *)
FA_WRITE vstart
ANA

```

標準命令、ファンクションブロックとファンクション

ADD	nb_written
ST	nb_written
LD	file_id (* vend を書きこみ *)
FA_WRITE	vend
ANA	
ADD	nb_written
ST	nb_written
LD	file_id (* vinc を書きこみ *)
FA_WRITE	vinc
ANA	
ADD	nb_written
LD	(* delta_tim を書きこみ *)
ANA	(* 整数型に変換*)
ST	ana_delta_tim
LD	file_id
FA_WRITE	ana_delta_tim
ANA	
ADD	nb_written
ST	nb_written
F_CLOSE	
ST	ok
LD	nb_written
EQ	4
RETC	(* 4 ならばリターン*)
LD	ERR_FILE (* それ以外はエラー*)
ST	ERROR

FM_READ (ファイルの読み込み)



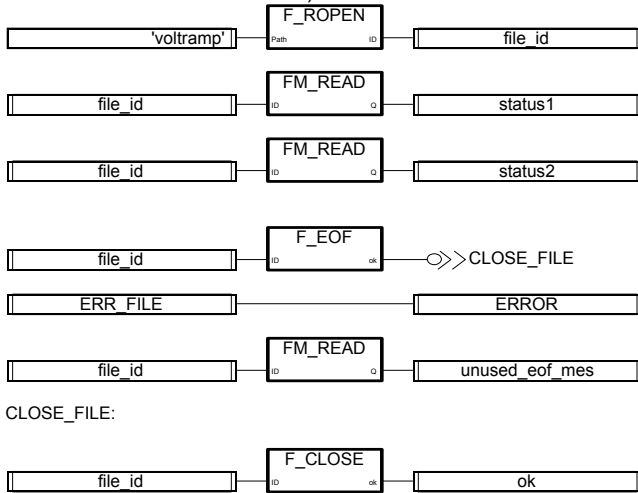
引数::

ID	整数型	ファイル番号 (F_ROPEN の戻り値).
Q	文字列型	ファイルから読み出された文字列

説明::

バイナリファイルからの文字列を読み出します。
F_ROPEN、F_CLOSE と一緒に使います。
前回アクセスした位置からシーケンシャルにファイルにアクセスします。
F_ROPEN コールのための最初のコールでは、ファイルの先頭の文字列を読み出します。
毎回のコールで読み出している位置をスタックにプッシュします。
文字列の最後には NULL(0)、エンドオブライン ("\\n") 又は、改行 ("\\r") が付いています。
ファイルの最後に到達したかチェックするためには F_EOF を使います。
このファンクションは ISaGRAF のシミュレータには含まれません。

(* ファイル管理用のFBDプログラム*)



(* 等価なST言語: *)

```

file_id := F_ROPEN('voltramp.bin');
status1 := FM_READ(file_id);
status2 := FM_READ(file_id);
IF (F_EOF(file_id)) THEN
    ERROR := ERR_FILE;
    unused_eof_mes := FM_READ(file_id);
END_IF;
ok := F_CLOSE(file_id);

```

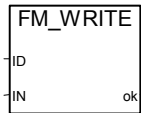
(* 等価なIL言語: *)

```

LD      'voltramp.bin'
F_ROPEN
ST      file_id
FM_READ      (* status1を読みこみ *)
ST      status1
LD      file_id
FM_READ      (* status2を読みこみ *)
ST      status2
LD      file_id
F_EOF
JMPNC    CLOSE_FILE (* ファイル終端ならジャンプしない*)
D      ERR_FILE
ST      ERROR
LD      file_id
FM_READ      (* unused_eof_mes 読みこみ *)
ST      unused_eof_mes
CLOSE_FILE: LD      file_id
F_CLOSE
ST      ok

```

FM_WRITE (ファイルの書き込み)



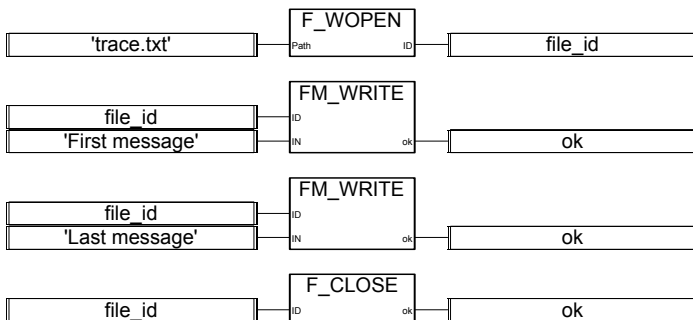
引数::

ID	整数型	ファイル番号 (F_WOPEN の戻り値)
IN	文字列型	ファイルに書き込む文字列
ok	ブール型	実行結果 =TRUE: 正常終了

説明::

バイナリファイルへ文字列を書き込みます。
 F_WOPEN、F_CLOSE と一緒に使います。
 文字列の最後は NULL(0)として書き込みます。
 前回アクセスした位置からシーケンシャル的にファイルにアクセスします。
 F_WOPEN コールの後の最初のコールでは、ファイルの先頭に文字列を書き込みます。
 毎回のコールで読み出している場所をスタックにプッシュします。
 このファンクションは ISaGRAF のシミュレータには含まれません。

(* ファイル管理用のFBDプログラム*)



(* 等価なST言語:: *)

```

file_id := F_WOPEN('trace.txt');
ok := FM_WRITE(file_id,'First message');
ok := FM_WRITE(file_id,'Last message');
ok := F_CLOSE(file_id);
  
```

(* 等価なIL言語:: *)

```

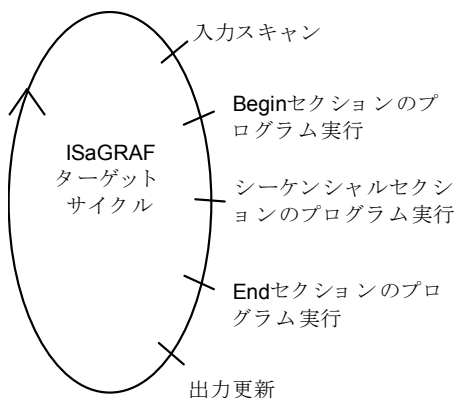
LD      'trace.txt'
F_WOPEN
ST      file_id
FM_WRITE 'First message' (* 1 番目の文字列を書きこむ *)
ST      ok
LD      file_id
  
```

FM_WRITE	'Last message' (* 2 番目の文字列を書きこむ *)
ST	ok
LD	file_id
F_CLOSE	
ST	ok

C. ターゲットユーザガイド

C.1 はじめに

ISaGRAF ターゲットはパソコンやボード上で ISaGRAF のアプリケーションをリアルタイムに実行するソフトウェアです。下図のような実行サイクルになります。



ターゲットサイクルは外界からの入力スキャン、ISaGRAF アプリケーションプログラムの実行、外界への出力更新から成り立っています。

– 第1部では、特定のシステム上でのターゲットについて起動方法などを解説します。

- DOSターゲット
- OS-9ターゲット
- VxWorksターゲット
- NTターゲット

が含まれています。各ターゲットで、はじめに起動方法、続いて、ターゲット固有の注意事項（電源オン時のスタートアップ処理、エラー処理、全体の振る舞いなど）についての情報をまとめます。

– 第2部では、ユーザ定義のC言語ファンクション、ファンクションブロック、変換関数の実装方法について解説を行います。

– 第3部では、ワークベンチとターゲット間の通信プロトコルであるMODBUSに関してファンクションコード毎に解説を行います。

– 第4部では、ターゲットの電源異常や再スタートの解説を行います。

C.2 ターゲットのインストール

ターゲットのインストールには約1MB のディスクの空き容量が必要です。ターゲットに付属する install.bat バッチファイルで必要なファイルをパソコンにコピーします。

例: a:\install a: c:\path

これはドライブ A :のファイルを C:\path にインストールします。

注意: ターゲットの種類によって、インストール手順は若干異なります。詳細はターゲット付属のドキュメントファイルをご覧ください。

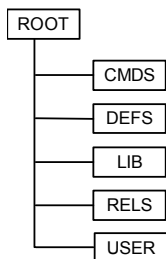
注意: 標準ターゲットのファイルは、下記のホームページからダウンロードして入手してください。URL は

<http://www.co-nss.co.jp>

または、ISaGRAF 開発ツールキットのCD-ROMに収録されているものをインストールしてください。

それ以外のターゲットの入手方法に付いては、サプライヤにお問合せ下さい。

ターゲットシステムには以下のようなディレクトリ構成ができます。



- | | |
|---------------------|---|
| ROOT ディレクトリ: | いくつかのツールと readme ファイルを含みます。 |
| CMDS ディレクトリ: | 実行モジュールを含みます。 |
| DEFS ディレクトリ: | ヘッダーファイルを含みます。 |
| LIB ディレクトリ: | ライブラリファイルを含みます。 |
| RELS ディレクトリ: | リロケートブルファイル(オブジェクトファイル)を含みます。 |
| USER ディレクトリ: | ユーザ定義C言語ファンクション、ファンクションブロック、変換関数のソースファイルとヘッダーファイルを含みます。 |
- ターゲットによっては、下記のディレクトリがあります。
- | | |
|--------------------|-------------------------------|
| ARK ディレクトリ: | ライブラリ用のI/Oボードのアーカイブファイルを含みます。 |
|--------------------|-------------------------------|

ターゲットシステムへのファイルのインストールが終われば、次の節に解説があるように各ターゲット毎の起動が可能になります。

C.3 DOSターゲットの使い方

C.3.1 ISaGRAF ターゲットの実行: ISA.EXE

MS-DOSターゲットではターゲットは1個のプログラム ISA.EXE として実行します。ISA.EXE の実行時のオプションは isa -? でみることができます。CMDSD ディレクトリから実行して下さい。

DOSターゲットではワークベンチターゲット間の通信がターゲットのパフォーマンスに影響を与えますので、ターゲット実行中にはこの通信にあまり負荷をかけないことが望めます。本ターゲットは割り込み処理ルーチンには影響を与えません。

通信リンクの設定: -t オプション

ISaGRAF ターゲットはワークベンチデバッガーとの通信にシリアル通信を使います。ポート名は -t オプションで指定します。通信ポートは COM1, 2, 3, 4 のの中から選択します。通信インタフェースの設計は COM1~COM3 が利用できるパソコンに互換なので、BIOSのバージョンに依存します。

デフォルト設定: なし もし、このオプションを指定しないと、ワークベンチ側との通信ができません。このとき、ターゲットにはエラー番号7が表示されます。

イーサネット通信はDOSターゲットではサポートされていません。

ISaGRAF を起動する前に**通信ポートのパラメータ(ボーレートなど)設定**が必要となります。この設定はワークベンチデバッガーのリンク設定の内容と一致している必要があります。(詳細はワークベンチユーザガイド セクションA参照。)

通信ポートの設定例

MODE COM1:9600,N,8,1

この設定により、通信パラメータは以下のように設定されます。

ボーレート	9600
パリティチェック	なし
データ長	8 bits
ストップビット	1 bit

ターゲットマシンのBIOSのバージョンによってはボーレート 19200 がサポートされていないものがありますので注意が必要です。同様の設定はICS Triplex ISaGRAF Inc. インターナショナル社のユーティリティソフト (ISAMOD.EXE) でも行うことができます。

例:

ISAMOD COM1

このコマンドは MODE COM1:19200,N,8,1 と同様の設定を行います。

スレーブ番号の設定: **-s** オプション

このオプションはターゲットのスレーブ番号を設定します。1～255(但し、13は除く)が使えます。スレーブ番号は通信プロトコルで使っています。スレーブ番号はおもに2つ以上のターゲットが存在する場合にこれらを区別するためのものです。ワークベンチのデバッガーを使用する際にはリンク設定でこのスレーブ番号を正しく設定する必要があります。

デフォルト値: **デフォルトスレーブ番号は1**
これは、ワークベンチ側のデフォルト値と同じです。

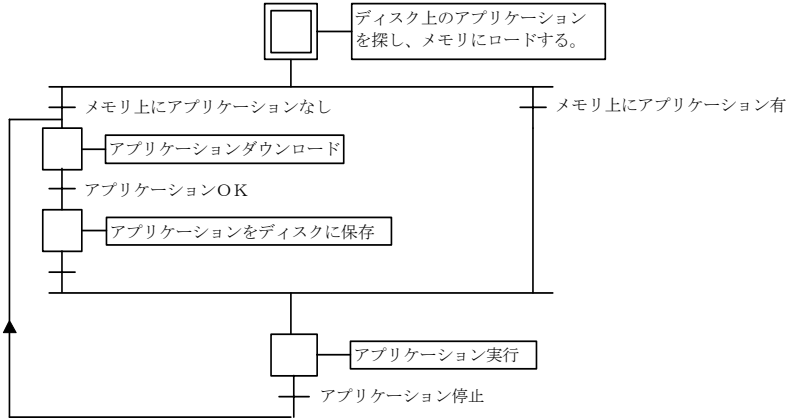
DOS ターゲットの起動例

isamod COM1	通信ポート COM1 を 19200 baud、パリティなし、8ビットデータ、ストップビット1に設定します。
isa -t=COM1	ISaGRAF ターゲットをデフォルトスレーブ番号1で通信ポート COM1 に設定して起動します。
isa -s=3 -t=COM1	ISaGRAF ターゲットをスレーブ番号3で通信ポート COM1 に設定して起動します。

C.3.2 DOSターゲットの特徴

ISaGRAF の起動

ターゲットが起動されるとき、以下のアルゴリズムで実行されます。



基本的な考え方

アプリケーションコード(中間コード)はワークベンチにより生成され、ダウンロードされたバイナリのデータベースです。これが、ISaGRAF ターゲットにより実行されます。シンボルテーブルが同時に使われる場合もあります。

アプリケーションのシンボルテーブルとはワークベンチにより生成され、ダウンロードされるテキスト形式のデータベースです。このファイルによってシンボル(変数名)とターゲット内部のデータとの関連付けを行います。ユーザーがシンボルを使用しない場合はこのファイルはダウンロードする必要はありません。シンボルテーブルについての詳細はセクションAのアドバンスドプログラミングに解説されています。

アプリケーションコードのバックアップ

新しいアプリケーションがワークベンチのデバッガーからダウンロードされると、アプリケーションコードはターゲットのカレントディレクトリに以下のファイル名で保存されます。

ISAx1 ISaGRAF アプリケーションコードのバックアップファイル(x はスレーブ番号)

さらに、アプリケーションのシンボルテーブルをあらかじめダウンロードしておくと、同じくカレントディレクトリに以下のファイル名で保存されます。

ISAx6 ISaGRAF アプリケーションシンボルテーブルのバックアップファイル(x はスレーブ番号)

ISaGRAF ターゲットが起動すると、アプリケーションコードとシンボルファイルがカレントディレクトリから検索され、メモリ上にロードされます。
シンボルファイルがない場合は、ターゲットはアプリケーションコードだけで実行されます。
アプリケーションコードがない場合は、ダウンロードされるのを待ちます。

ターゲットの電源ON時にデバッガーからのダウンロードなしで特定のアプリケーションをスタートさせるには、これらのファイルをターゲットのカレントディレクトリにコピーしておくようにします。ターゲットとワークベンチが同じパソコンならディスク間でコピーし、異なる場合はフロッピーディスクなどを使います。

もし、ISaGRAF ワークベンチが標準のディレクトリ(\\isa-win)にインストールされていれば、プロジェクト(MYPROJ)のアプリケーションコード(即ち、中間コード)は以下のようになります。

\\ISAWIN\\<プロジェクトグループ名>\\APL\\MYPROJ\\appli.x8m

同様に、アプリケーションシンボルファイルは以下のようになります。

\\ISAWIN\\<プロジェクトグループ名>\\MYPROJ\\appli.tst

例: アプリケーションコードのコピー

カレントディレクトリを ISA.EXE がインストールされているディレクトリ(c:\\target\\dos\\cmds)にして、下記のコマンドで MYPROJ プロジェクトのアプリケーションコードをコピーします。

copy \\ISAWIN\\<プロジェクトグループ名>\\MYPROJ\\appli.x8m isa11

この後、ISA.EXE を実行すると MYPROJ プロジェクトが起動されます。

このようなコマンドをバッチファイルにしてワークベンチの「ツール」メニューにユーザが追加することが可能です。(詳細は、セクションAを参照願います。)

エラー処理とエラー出力メッセージ

ISaGRAF ターゲットソフトにはエラー検出処理が含まれています。エラーメッセージとその説明は本マニュアルの後の章で解説を行います。

エラー検出の流れは以下になります。

- エラーとエラー番号は ISaGRAF のエラールーチンへ送られます。
- もし、ワークベンチ側のコード生成メニューでエラー処理のフラグがセットされていれば、エラー処理がなされますが、そうでない場合はエラー情報は失われ、なにもしません。

エラーが処理される場合:

- エラー番号(10進数)と引数(16進数)標準出力(stdout)に表示されます。
- このエラー番号と引数はリングFIFOバッファに登録されます。バッファサイズはワークベンチのコード生成メニューから設定可能です。もし、FIFOバッファが一杯であれば新しいエラーが発生すれば、最も古いエラー情報が失われます。
- エラー情報はワークベンチのデバッガーウィンドウや、アプリケーション中のSYSTEM ファンクションをコールして取得できます。

ワークベンチのデバッガーがエラーを検出した場合は、エラーウィンドウにメッセージが表示されます。ここでは、アプリケーションの実行状態（実行中、停止中）表示に加えて、エラーが発生したプログラム名や変数名、エラー番号、引数を[]内に表示します。

ターゲットの起動時の表示やエラー発生時には標準出力 `stdout` にメッセージが表示されますが、これを非表示にしたい場合は以下のようにリダイレクトを使って ISaGRAF を起動します。

```
isa -t=COM1 -s=1 >NUL
```

システムクロック

ISaGRAF DOS ターゲットはどのような DOS システムでも動作できるよう、サイクルタイムの同期やタイマ変数の更新周期は標準のタイマチックになります。この精度は55msとなります。

従って、55msより精度の高いタイマ変数を持つことはできません。同様の理由で55msより短いサイクルタイムを設定することはできません。もし、0より大きくて55ms未満のものを設定すると、サイクルタイムオーバーフローエラー（エラー番号62）が発生します。このときは、サイクルタイムのトリガーが無視されて、連続的に ISaGRAF ターゲットが実行されます。

標準のシステムチェックを変更しないメリットとしては、常駐型の ISaGRAF 以外のプログラムやC言語ファンクションが ISaGRAF の実行によって影響されることがないということです。

精度の高いシステムチェックが必要な場合はサプライヤに問い合わせ願います。

終了キー

デスクトップPC上で ISaGRAF を実行している場合は、以下のキーの組み合わせで ISaGRAF を終了させることができます。

shift + ctrl + alt

もちろん、産業用のアプリケーションなどでは不意なキー入力による ISaGRAF 終了を避けるべきで、これらのキー入力を無視するような ISaGRAF ターゲットもあります。

理由として、このような終了ではI/Oボードインタフェースが必ずしも出力をOFFして終了することを保証できないためです。正常に終了するには、

- デバッガーウィンドウからアプリケーションの停止を行います。I/Oのクローズが正しく行われます。
- その後、キーボードでターゲットを終了します。

☐ **アプリケーションコードのサイズの制限**

MS-DOSターゲットはインテルCPUのリアルモードのアプリケーションです。実行中はアプリケーションコードのサイズの最大値は64KBとなります。もし、これ以上の大きさのプログラムが作成され、ダウンロードされる場合はダウンロード後に ISaGRAF がこのアプリケーションを実行しようとしてシステムをクラッシュする可能性があります。

さらに、扱えるメモリも640KBのコンベンションメモリを越えることはできません。

C.4 OS-9ターゲットの使い方

まず、OS-9ターゲットに必要な実行モジュール(CMDS ディレクトリから)をファイル転送ツールで転送します。

次に、転送された ISaGRAF の実行をコマンドラインからのヘルプを参照して行います。

```
isa -?  
isaker -?  
isatst -?  
lsanet -?
```

C.4.1 シングルタスクモード ISaGRAF の実行: isa

ISaGRAF はシングルタスクとして実行することができます。しかし、この場合は通信の負荷が直接 ISaGRAF の処理パフォーマンスに影響するので注意が必要です。OS-9のマルチタスクシステム上では、スレーブ番号や通信ポートが重複しなければ同一CPU上で複数の ISaGRAF シングルタスクターゲットを実行できます。

通常、シングルタスクとしての実装はシングルタスクしか行えないOS、例えばボードマイコンやMS-DOSパソコンのためであったり、ISaGRAF ターゲットの移植作業を行う際のプロトタイプとして行う場合が多いです。このため、ISaGRAF の実装はマルチタスクモードで行うことが好ましいと言えます。

なお、ISaGRAF のシングルタスクモードでの実行で、バックグラウンドで実行中のタスクや割り込み処理などを妨げることはありません。

☞ **通信リンクの設定:-t オプション**

シングルタスクモードでの実行ではデバッガーとの通信にシリアル通信を使います。この通信ポートを **-t** オプションで設定します。

デフォルト値: なし もし、このオプションが省略されると、デバッガーとの通信は行えません。このとき、エラー番号7がコンソールに表示されます。

イーサネットリンクによる通信はシングルタスクモードでは使えません。

シリアルリンクデバイスはバイナリデータ転送モードでオープンします(コントロールキャラクタなし、XON/XOFF なし)。他の通信パラメータは ISaGRAF を起動する前に設定しておく必要があります。この設定内容は、ワークベンチ側での通信リンクの設定と一致させておく必要があります。

例:

xmode /t0 baud=19200

/t0 デバイスの通信ボーレートを 19200 に設定する

スレーブ番号の設定: -s オプション

このオプションはスレーブ番号を設定します。1～255 (但し、13(\$0D)は除く) が使えます。スレーブ番号は通信リンクプロトコル内部で使用されます。2つ以上のターゲットが実行している場合にこれらを識別するために使われます。ワークベンチ側でのリンク設定でデバッグするターゲットとのスレーブ番号を一致させる必要があります。

デフォルト値: スレーブ番号のデフォルト値は1です。(これは、ワークベンチ側のデフォルト設定と同じです。)

OS-9ターゲットの起動例:

isa -t=/t0	ISaGRAF をシングルタスクで実行。スレーブ番号1、通信ポート /t0 に設定。
isa -s=3 -t=/t1	ISaGRAF をシングルタスクで実行。スレーブ番号3、通信ポート /t1 に設定。
isa -t=/t0 & isa -s=3 -t=/t1	2つの ISaGRAF をシングルタスクで実行。1つはスレーブ番号1、通信ポート /t0 に設定、もう一つはスレーブ番号3、通信ポート /t1 に設定

C.4.2 マルチタスクモード ISaGRAF の実行: isaker, isatst, isanet

ISaGRAF ターゲットカーネルの通信レスポンスを改善するために、ターゲットの通信部分をアプリケーション実行部分から切り離すことにより2つのタスクに分離します(アプリケーションの実行タスクは isaker :カーネルタスク、通信タスクは isatst あるは isanet と呼ばれます)。

この構成は柔軟性があり、例えば1つのカーネルタスクに複数の通信タスクをリンクしたり、1つの通信タスクに最大4つのカーネルタスクをリンクさせたりできます。

このことで、プロセスモニタリングのようなアプリケーションとワークベンチデバッガが1つのアプリケーションの同じ通信ポートを共有したり、最大4つの異なるアプリケーションに1つの通信ポートでリンクする、といったことが簡単に構築できます。

カーネルタスクと通信タスクは独立しています。唯一、起動の条件として、カーネルタスクを最初に実行してから通信タスクを起動する必要があります。

マルチタスクモードでの ISaGRAF がバックグラウンドプロセスや割り込み処理を妨げることはありません。

C.4.2.1 カーネルタスクの実行: isaker

スレーブ番号: -s オプション

このオプションはスレーブ番号を設定します。1～255 (但し、13(\$0D)は除く) が使えます。スレーブ番号は通信リンクプロトコル内部で使用されます。複数のターゲットが実行している場合にこれらを識別するために使われます。ワークベンチ

側でのリンク設定でデバッグするターゲットとのスレーブ番号を一致させる必要があります。

デフォルト値: スレーブ番号のデフォルト値は1です。(これは、ワークベンチ側のデフォルト設定と同じです。)

C.4.2.2 シリアル通信タスクの実行: isatst

通信リンクの設定: -t オプション

ターゲットの通信タスク isatst はデバッガーとの通信にシリアル通信を使います。この通信ポートの名前を -t オプションで設定します。

デフォルト値: なし
もし、このオプションが省略されると、デバッガーとの通信は行えません。このとき、エラー番号7がコンソールに表示されます。

イーサネットによる通信は isatst 通信タスクでは使えません。

シリアル通信デバイスはバイナリデータ転送モードでオープンします。(コントロールキャラクタなし、XON/XOFF なし)他の通信パラメータは ISaGRAF を起動する前に設定しておく必要があります。この設定内容は、ワークベンチ側での通信リンクの設定と一致させておく必要があります。

例:

xmode /t0 baud=19200

/t0 デバイスの通信ボーレートを 19200 に設定する

スレーブ番号: -s オプション

このオプションはスレーブ番号を設定します。1~255(但し、13(\$0D)は除く)が使えます。このオプションは連続で最大4つの異なるカーネルスレーブに対して設定できます。スレーブ番号は通信リンクプロトコル内部で使用されます。複数のターゲットが実行している場合にこれらを識別するために使われます。ワークベンチ側でのリンク設定でデバッグするターゲットとのスレーブ番号を一致させる必要があります。

デフォルト値: スレーブ番号のデフォルト値は1です。(これは、ワークベンチ側のデフォルト設定と同じです。)

通信タスクの論理番号設定: -c オプション

このオプションは通信タスクの論理番号を設定します。同時に複数の通信タスクを管理する場合に使います。1~255の番号でそれぞれの通信タスクで重複しないようにします。

デフォルト値: スレーブ番号(-s オプションで指定)。これは、旧バージョン(3.0x)との互換性があります。

通信リンクの設定: -t オプション

ターゲット通信タスク isanet は標準イーサネット通信をデバッガー通信に使用します。ポート番号は -t オプションで設定します。

デフォルト値: なし

もし、このオプションが省略されると、デバッガーとの通信は行えません。このとき、エラー番号7がコンソールに表示されます。

ワークベンチ側のリンク設定をターゲット側の通信タスクと一致させておく必要があります。

ISaGRAF にとって、OS-9ターゲット側はサーバに相当し、デバッガー側は指定されたポート番号に接続するクライアントに相当します。

イーサネット通信によるデバッグを行う前に、OS-9のイーサネットデバイスを正しく設定しておく必要があります。例えば、イーサネット通信の確認は ping コマンドで行うことができます。

スレーブ番号: -s オプション

このオプションはスレーブ番号を設定します。1~255(但し、13(\$0D)は除く)が使えます。このオプションは連続で最大4つの異なるカーネルスレーブに対して設定できます。スレーブ番号は通信リンクプロトコル内部で使用されます。複数のターゲットを実行する場合にこれらを識別するために使われます。ワークベンチ側でのリンク設定でデバッグするターゲットとのスレーブ番号を一致させる必要があります。

デフォルト値: スレーブ番号のデフォルト値は1です。(これは、ワークベンチ側のデフォルト設定と同じです。)

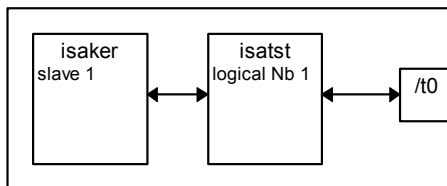
通信タスクの論理番号設定: -c オプション

このオプションは通信タスクの論理番号を設定します。同時に複数の通信タスクを管理する場合は使います。1~255の番号でそれぞれの通信タスクで重複しないようにします。

デフォルト値: スレーブ番号(-s オプションで指定)。これは、旧バージョン(3.0x)との互換性があります。

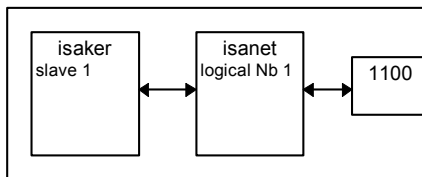
C.4.2.4 OS-9ターゲットの実行例:

**isaker &
isatst -t=/t0**



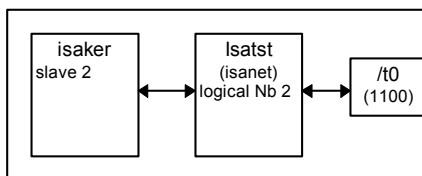
ISaGRAF カーネルタスクをデフォルトのスレーブ番号(1)で実行。
ISaGRAF シリアル通信タスクを、通信ポートは/t0、デフォルトのスレーブ番号1にリンク、デフォルトの論理番号1(=直前に指定したスレーブ番号はデフォルトの1)で実行。

**isaker &
isanet -t=1100**



ISaGRAF カーネルタスクをデフォルトスレーブ番号1で実行。
ISaGRAF イーサネット通信タスクを、ポート番号1100、デフォルトのスレーブ番号1にリンク、デフォルトの論理番号1(=直前に指定したスレーブ番号はデフォルトの1)で実行。

**isaker -s=2 &
isatst -t=/t0 -s=2** (あるいは isanet -t=1100 -s=2)



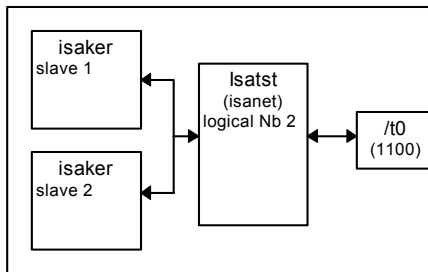
ISaGRAF カーネルタスクをスレーブ番号2で実行。
ISaGRAF シリアル(イーサネット)通信タスクを、通信ポート/t0(ポート番号1100)、スレーブ番号2にリンク、論理番号は2(直前に指定したスレーブ番号=2)

OS-9ターゲットの使い方

isaker -s=1 &

isaker -s=2 &

isatst -t=/t0 -s=1 -s=2 (あるいは **isanet -t=1100 -s=1 -s=2**)



ISaGRAF カーネルタスクをスレーブ番号1で実行。

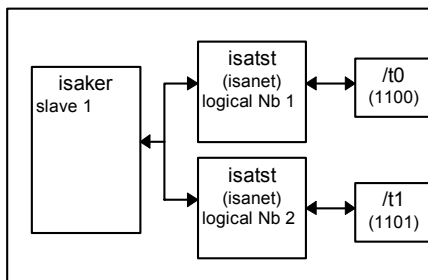
ISaGRAF カーネルタスクをスレーブ番号2で実行。

ISaGRAF シリアル(イーサネット)通信タスクを、通信ポート/t0(ポート番号1100)、スレーブ番号1、2両方にリンク、論理番号はデフォルトで2(最後に指定したスレーブ番号)。

isaker -s=1 &

isatst -t=/t0 -s=1 -c=1 & (あるいは、**isanet -t=1100 -s=1 -c=1 &**)

isatst -t=/t1 -s=1 -c=2 (あるいは、**isanet -t=1101 -s=1 -c=2**)



ISaGRAF カーネルタスクをスレーブ番号1で実行。

ISaGRAF シリアル(イーサネット)通信タスクを実行、通信ポート/t0(ポート番号1100)、スレーブ番号1にリンク、論理番号1。

ISaGRAF シリアル(イーサネット)通信タスクを実行、通信ポート/t1(ポート番号1101)、スレーブ番号1にリンク、論理番号2。

注意: シリアル、イーサネット通信タスクは共存できます。

C.4.3 OS9ターゲットの特徴

通信リンク

OS-9のシリアルキャラクタマネージャは柔軟性がありますので、マイクロウェア社がサポートしているほとんどの双方向デバイスが使えます。

具体例:

シリアルリンクはネットワーク経由で別のCPUにある通信ポートに対して行うことができます。

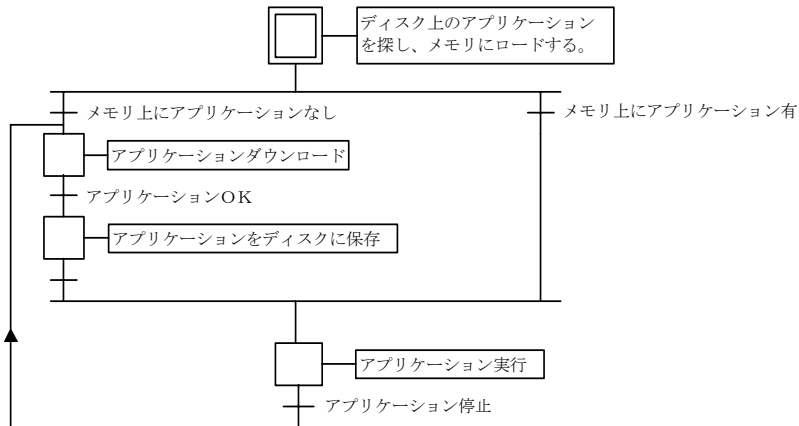
この場合の通信オプションの設定は以下のようになります。

`-t=/nr/MASTER/t0`

ここで、通信リンクは MASTER と呼ばれる別のCPUに対して ramnet ネットワーク経由で /t0 に接続されます。

ISaGRAF ターゲットの起動

ISaGRAF ターゲットは以下のアルゴリズムに従って実行されます。



基本的な考え方

アプリケーションコード(中間コード)はワークベンチにより生成され、ダウンロードされたバイナリのデータベースです。これが、ISaGRAF ターゲットにより実行されます。シンボルテーブルが同時に使われる場合もあります。

アプリケーションのシンボルテーブルとはワークベンチにより生成され、ダウンロードされるテキスト形式のデータベースです。このファイルによってシンボル(変数名)とターゲット内部のデータとの関連付けを行います。ユーザがシンボルを使用しない場合はこのファイルはダウンロードする必要はありません。シンボルテーブルについての詳細はセクションAのアドバンスドプログラミングに解説されています。

● OS-9用オブジェクトと複数ターゲット

すべての ISaGRAF ターゲットのアプリケーションコードは ISAxn のファイル名となります。ここで、x はカーネルのスレーブ番号、n はスペース番号と呼ばれて特別な意味を持っています。但し、ISAy3 だけは、例外です。y は通信タスクの論理番号です。

スレーブ番号や通信タスク論理番号が重複しなければ、同一CPU上で複数の ISaGRAF ターゲット(カーネルタスクと通信タスク)を実行可能です。異なるアプリケーションであっても、同一I/Oボードのアクセスが可能であるため、アプリケーション側で注意しておく必要があります。例えば、I/Oドライバにセマフォ管理を入れるということも一つの方法です。

OS-9ターゲットのオブジェクト名:

ファイル名:

ISAx1	ISaGRAF アプリケーションコードのバックアップファイル
ISAx6	ISaGRAF アプリケーションシンボルのバックアップファイル

メモリモジュール名(メモリスペース):

ISAx0	ISaGRAF カーネルシステムデータ
ISAx1	ISaGRAF アプリケーションコード
ISAx2	ISaGRAF カーネルのリアルタイムデータベース
ISAy3	ISaGRAF 通信用データバッファ(メールボックス)
ISAx4	ISaGRAF オンライン修正用アプリケーションコード1
ISAx5	ISaGRAF オンライン修正用アプリケーションコード2
ISAx6	ISaGRAF アプリケーションシンボル

ここで、x はスレーブ番号です。

ユーザはこれらと同じ名前のファイルを使わないようにしてください。

● アプリケーションコードのバックアップ

新しいアプリケーションがワークベンチのデバッガーからダウンロードされると、アプリケーションコードはターゲットのカレントディレクトリに以下のファイル名で保存されます。

ISAx1	ISaGRAF アプリケーションコードのバックアップファイル(x はスレーブ番号)
-------	--

さらに、アプリケーションのシンボルテーブルをあらかじめダウンロードしておくと、同じくカレントディレクトリに以下のファイル名で保存されます。

ISAx6	ISaGRAF アプリケーションシンボルテーブルのバックアップファイル(x はスレーブ番号)
-------	---

ISaGRAF ターゲットが起動すると、アプリケーションコードとシンボルファイルがカレントディレクトリから検索され、同名のメモリモジュール上にロードされます。シンボルファイルがない場合は、ターゲットはアプリケーションコードだけで実行されます。

アプリケーションコードがない場合は、ダウンロードされるのを待ちます。

ターゲットの電源ON時にデバッガーからのダウンロードなしで特定のアプリケーションをスタートさせるには、

- ・第一の方法として、ワークベンチをインストールしたパソコンからターゲットのカレントディレクトリにこれらのファイルをコピーしておくようにします。操作を簡単にするにはワークベンチのツールメニューを利用してください(詳細は、セクションAを参照願います)。
- ・第二の方法として、ワークベンチのパソコンからアプリケーションコード(必要ならシンボルテーブルも)を不揮発性メモリ(PROM や EPROM)に格納するようなツールを作ることです。

こうすれば、システムの電源オン時に必要であれば(例えば、高速アクセスが必要だとか、ブレイクポイントの管理などの理由で)アプリケーションコードを PROM からメモリデータモジュールの RAM である **ISAx1** に(必要ならシンボルテーブルも **ISAx6** にも)ロードすることが可能です。

注意: もしアプリケーションコードに書き込みが許されていない場合は(例えば、ROM化されているなど)、ISaGRAF のデバッグからのブレイクポイントの設定などは行えません。

もし、ISaGRAF ワークベンチが標準のディレクトリ(\isawin)にインストールされていれば、プロジェクト(MYPROJ)のアプリケーションコード(即ち、中間コード)は以下のようになります。

\ISAWIN\<プロジェクトグループ名>APL\MYPROJ\appli.x6m
(ターゲット上での ISAx1 に相当)

同様に、アプリケーションシンボルファイルは以下のようになります。

\ISAWIN\<プロジェクトグループ名>\MYPROJ\appli.tst
(ターゲット上の ISAx6 に相当)

エラー処理とエラー出力メッセージ

ISaGRAF ターゲットソフトにはエラー検出処理が含まれています。エラーメッセージとその説明は本マニュアルの後の章で解説を行います。

エラー検出の流れは以下のようになります。

- － エラーとエラー番号は ISaGRAF のエラールーチンへ送られます。
- － もし、ワークベンチ側のコード生成メニューでエラー処理のフラグがセットされていれば、エラー処理がなされますが、そうでない場合はエラー情報は失われ、なにもしません。

エラーが処理される場合:

- － エラー番号(10進数)と引数(16進数)標準出力(stdout)に表示されます。
- － このエラー番号と引数はリングFIFOバッファに登録されます。バッファサイズはワークベンチのコード生成メニューから設定可能です。もし、FIFOバッファが一杯であれば新しいエラーが発生すれば、最も古いエラー情報が失われます。
- － エラー情報はワークベンチのデバッガーウィンドウや、アプリケーション中の SYSTEM ファンクションをコールして取得できます。

ワークベンチのデバッガーがエラーを検出した場合は、エラーウィンドウにメッセージが表示されます。ここでは、アプリケーションの実行状態(実行中、停止中)表示に加えて、エラーが発生したプログラム名や変数名、エラー番号、引数を[] 内に表示します。

ターゲットの起動時の表示やエラー発生時には標準出力 `stdout` にメッセージが表示されますが、これを非表示にする場合は以下のように ISaGRAF を起動することになります。

```
prog_name [options] >>>/nil
```

＝ サイクルタイム、タスク、タスクのプライオリティ

－ ISaGRAF ターゲットサイクルの最後に(次のサイクルがスタートする直前)、以下のアルゴリズムで処理がされます。

アプリケーションにサイクルタイムが指定されている場合は、残った時間(指定サイクルタイム - 現在使用した時間)CPUを開放します。もし、この時間が負の値の時、サイクルオーバーフローとなり、CPUはスケジューリングのため1チック分だけ開放されます。

もし、サイクルタイムが指定されていない場合、あるいは、残り時間が1チック以下である場合は、CPUはスケジューリングのため1チック分だけ開放されます。

ターゲットのサイクルタイムの精度はOS-9システムのシステムチックの精度に相当します。

通常、サイクルタイムの指定は、サイクルのトリガをかけるため、またはCPUの空き時間を他のタスクに開放するためにおこないます。

－ 通信タスクは通信リンクからデータが来ないときはスリープ状態となっています。要求があれば、カーネルタスクとの Question/Answer プロトコルを使って実行中のアプリケーションの情報を取得します。通信タスクはカーネルに Question を投げかけ、カーネルのサイクルの最後に(これはアプリケーションデータの同期をとるため)カーネルは通信タスクに対して Answer を返します。

ISaGRAF のタスクは、自分のプライオリティを自らは変更しません。ユーザが全体の構成から判断して与えることになります。

例えば、優先度の低いタスクから処理を妨げられないようにするには、OS-9タスク管理パラメータ(`MIN_AGE` や `MAX_AGE`)を修正します。

＝ ターミナルモード

ターゲットのシリアル通信プロトコルはキャリッジリターン(\$0D)を3回受けると、OS-9のShellタスクがシリアルリンクデバイスに割り付けられていれば、これを起動します。

これにより、OS-9のShellプロンプトが汎用のターミナルに表示されます。これを使ってOS-9のShellを利用できます。

例:

ホストパソコン側から、

- － ISaGRAF のデバッガーを閉じます。
- － Windowsの「アクセサリ」のターミナルのセッションを正しい通信パラメータで開始します。
- － リターンキーを3回押します。

これで、OS-9のShellにログオンできたことになります。

- **logout** と入力すればターミナルモードを終了できます。

注意: 次回のワークベンチのデバッガーからの通信のためにも、必ずターミナルモードは **logout** で終了しておく必要があります。

C.5 VxWorksターゲットの使い方

ISaGRAF ターゲットを実行する前に、環境設定のためにいくつかのコマンドを実行する必要があります。これらのコマンドはスクリプトファイルから実行することができます。以下にこの設定について述べます。

C.5.1 システムリソースマネージャ: `isassr.o`

このモジュールは ISaGRAF ターゲットのいかなる構成の場合にも必要となります。必ず、最初にロードしなければなりません。このモジュールは、複数ターゲットの実行時のシステムリソースを管理します。

C.5.2 `isa.o`, `isakerse.o`, `isakeret.o` に共通の特徴

ISaGRAF を起動するには、以下のいずれかのモジュールをロードします。

<code>isa.o</code> :	ISaGRAF をシングルタスクモード(シリアル通信リンクのみ)で起動
<code>isakerse.o</code> :	ISaGRAF をマルチタスクモード(シリアル通信リンクのみ)で起動
<code>isakeret.o</code> :	ISaGRAF をマルチタスクモード(シリアル通信あるいは、イーサネット通信付き)で起動

これらのモジュールの詳細は後で解説します。

シリアル通信リンクの設定

ISaGRAF ターゲットは基本的にはシリアル通信をデバッガーとの間で行います。ISaGRAF ターゲットではこのシリアルリンクの設定は行いませんので、ユーザは別の手段で設定する必要があります。ただ、バイナリデータ転送モード(RAW モード)である必要があります。以下の `ISAMOD ()` サブルーチンが提供しており、これを使うこともできます。

```
uchar ISAMOD
(
    char *desc,      /* シリアルデバイス名*/
    uint32 baudrate /* ボーレート          */
)
```

解説:

指定されたシリアルリンクデバイスを指定されたボーレートでバイナリデータ転送モードに設定します。

戻り値:

0(正常時), BAD_RET (エラー時)

ワークベンチ側のリンク設定はターゲット側と一致させておく必要があります。

☐ システムクロックレート

グローバル変数 CLKRATE (uint32) をVxWorksのシステムクロックに初期化しておく必要があります。例えば、下記のようにして初期化できます。

```
CLKRATE = sysClkRateGet ()
CLKRATE のデフォルト値は 60Hz です。
```

C.5.3 シングルタスクモード ISaGRAF の実行: isa.o

ISaGRAF はシングルタスクとして実行することができます。しかし、この場合は通信の負荷が直接 ISaGRAF の処理パフォーマンスに影響するので注意が必要です。VxWorksのマルチタスクシステム上では、スレーブ番号や通信ポートが重複しなければ同一CPU上で複数の ISaGRAF シングルタスクターゲットを実行できます。

通常、シングルタスクとしての実装はシングルタスクしか行えないOS、例えばボードマイコンやMS-DOS/パソコンのためであったり、ISaGRAF ターゲットの移植作業を行う際のプロトタイプとして行う場合が多いです。このため、ISaGRAF の実装はマルチタスクモードで行うことが好ましいと言えます。

なお、ISaGRAF のシングルタスクモードでの実行で、バックグラウンドで実行中のタスクや割り込み処理などを妨げることはありません。

☐ スレーブ番号の登録

ISaGRAF ターゲットにはスレーブ番号を設定します。1～255(但し、13は除く)が使えます。スレーブ番号は通信リンクプロトコル内部で使用されます。2つ以上のターゲットが実行している場合にこれらを識別するために使われます。ワークベンチ側でのリンク設定でデバッグするターゲットとのスレーブ番号を一致させる必要があります。

スレーブ番号の登録のために以下の `isa_register_slave()` ルーチンを提供しています。

```
uchar isa_register_slave
(
    uchar slave      /* スレーブ番号 */
)
```

解説:

新しいスレーブ番号をマルチターゲット管理システムに登録します。

戻り値:

0 (正常時), BAD_RET (エラー時)

☐ アプリケーションのバックアップファイル

グローバル変数 TSK_FUNIT (char *) にアプリケーションバックアップ用のファイル名を設定しておくことができます。ISaGRAF ターゲットは標準のファイル管理ルーチン (`fopen`, `fread`, `fwrite`, `fclose`) をファイルのバックアップに使用します。デフォルト値は ("") で、バックアップしないという設定になっています。

例:

```
TSK_FUNIT = "host name :/C :/ISaGRAF/target/apl/"
```

この例ではバックアップ用として、`host_name` というパソコンのCドライブの `\\IsaGRAF\\target\\apl\\` ディレクトリを指定しています。最後の“\\”を忘れると、`IsaGRAF\\target\\` ディレクトリの `apl` ファイルに対してバックアップが行われてしまいますので注意して下さい。

複数のターゲットにそれぞれ異なるパス名を指定してから起動することもできます。詳細は後のアプリケーションバックアップのセクションを参照願います。

⇒ サイクルの最後での処理

`TSK_NBTCKSCHED` (uint 32) 変数が次のサイクルまでの待ち時間を表します。単位はチックです。デフォルト値は0 (即ち、同一優先度のタスクスケジューリング) です。

複数のターゲットにそれぞれ異なる値をセットしてから起動することもできます。詳細はサイクルタイム、タスクのプライオリティのセクションを参照願います。

⇒ ISaGRAF ターゲットの起動

環境の設定が終わったら、`isa_main` ルーチンで ISaGRAF ターゲットを起動します。

```
uchar isa_main
(
    uchar slave, /* スレーブ番号 */
    char *com     /* シリアル通信デバイス名 */
)
```

解説:

ISaGRAF ターゲットタスクの起動

戻り値:

0 (正常時)、それ以外 (エラー時)

スレーブ番号は前述のスレーブ番号の登録で行った番号です。

スレーブ番号や通信ポートが重複しなければ、複数のターゲットを起動することができます。

デバッグ時はワークベンチ側のリンク設定とターゲット側のリンク設定が一致していなければなりません。

⇒ VxWorksターゲットの起動例:

以下に、ISaGRAF をシングルタスクモード (スレーブ番号1、シリアルリンク設定 /`tyCo/1`) で実行するステップを示します。

カレントディレクトリはターゲットがインストールされているディレクトリです。

isassr.o モジュールのロード

ld < RELS/isassr.o

isa.o モジュールのロード

ld < CMDS/isa.o

シリアル通信リンク設定

ISAMOD ("tyCo/1", 19200)

システムクロックレート
CLKRATE = sysClkRateGet ()

スレーブ番号の登録
isa_register_slave (1)

バックアップファイル(デフォルト値なので省略可能)
TSK_FUNIT = ""

サイクルの最後の処理 (デフォルト値なので省略可能)
TSK_NBTCKSCHED = 0

ISaGRAF ターゲットの起動
sp (isa_main, 1, "tyCo/1")

C.5.4 マルチタスクモード ISaGRAF の起動: isakerse.o , isakeret.o

ISaGRAF ターゲットカーネルの通信レスポンスを改善するために、ターゲットの通信部分をアプリケーション実行部分から切り離すことにより2つのタスクに分離します(アプリケーションの実行タスクは isaker :カーネルタスク、通信タスクは isatst あるいは isanet と呼ばれます)。

この構成は柔軟性があり、例えば1つのカーネルタスクに複数の通信タスクをリンクしたり、1つの通信タスクに最大4つのカーネルタスクをリンクさせたりできます。

このことで、プロセスモニタリングのようなアプリケーションとワークベンチデバッガーが1つのアプリケーションの同じ通信ポートを共有したり、最大4つの異なるアプリケーションに1つの通信ポートでリンクする、といったことが簡単に構築できます。

カーネルタスクと通信タスクは独立しています。唯一、起動の条件として、カーネルタスクを最初に実行してから通信タスクを起動する必要があります。

マルチタスクモードでの ISaGRAF がバックグラウンドプロセスや割り込み処理を妨げることはありません。

通信用のハードウェアの性能にあわせて2つのモジュールが選択可能です。

- カーネルとシリアル通信リンク: isakerse.o

このモジュールは複数のカーネルタスクと複数のシリアル通信タスクを起動することができます。

- カーネルとシリアル/イーサネット通信リンク: isakeret.o

このモジュールは複数カーネルタスクと、複数のシリアルまたはイーサネット通信タスクを起動することができます。

ISaGRAF の起動手順は isakerse.o、isakeret.o モジュールのいずれも同じです。ただ、isakeret.o に関しては通信タスク(tst_main_ex)の起動時に、シリアルリンクデバイス名あるいはイーサネットリンク用のポート番号のどちらでも設定することができます。

VxWorks ターゲットはサーバになり、デバッガーは指定されたポート経由で接続するクライアントとして動作します。

カーネルのスレーブ番号の登録

ISaGRAF ターゲットにはスレーブ番号を設定します。1～255(但し、13は除く)が使えます。スレーブ番号は通信リンクプロトコル内部で使用されます。2つ以上のターゲットが実行している場合にこれらを識別するために使われます。ワークベンチ側でのリンク設定でデバッグするターゲットとのスレーブ番号を一致させる必要があります。

スレーブ番号の登録のために以下の `isa_register_slave()` ルーチンが提供されています。

```
uchar isa_register_slave
(
    uchar slave      /* スレーブ番号 */
)
```

解説:

新しいスレーブ番号をマルチターゲット管理システムに登録します。

戻り値:

0 (正常時), BAD_RET (エラー時)

通信タスクの登録

ISaGRAF 通信タスクには論理番号を設定します。2つ以上の通信タスクを同時に管理する場合に使用します。1～255の番号が使えます。ISaGRAF の通信タスクを起動する前にこの論理番号を登録しておく必要があります。

通信タスクの論理番号の登録のために以下の `isa_register_com()` ルーチンが提供されています。

```
uchar isa_register_com
(
    uchar com_id     /* 通信タスクの ID */
)
```

解説:

新しい通信タスク論理番号をマルチターゲット管理システムに登録します。

戻り値:

0 (正常時), BAD_RET (エラー時)

アプリケーションのバックアップファイル

グローバル変数 `TSK_FUNIT (char *)` にアプリケーションバックアップ用のファイル名を設定しておくことができます。ISaGRAF ターゲットは標準のファイル管理ルーチン (`fopen`, `fread`, `fwrite`, `fclose`) をファイルのバックアップに使用します。デフォルト値は ("") で、バックアップしないという設定になっています。

例:

```
TSK_FUNIT = "host name :/C :/ISaGRAF/target/apl/"
```

この例ではバックアップ用として、`host_name` というパソコンのCドライブの `\\SaGRAF\\target\\apl\\` ディレクトリを指定しています。最後の `" \"` を忘れると、

ISaGRAF\target\ ディレクトリの apl ファイルに対してバックアップが行われてしましますので注意して下さい。

複数のターゲットにそれぞれ異なるパス名を指定してから起動することもできます。詳細は後のアプリケーションバックアップのセクションを参照願います。

⇒ サイクルの最後での処理

TSK_NBTKSCHED (uint 32) 変数が次のサイクルまでの待ち時間を表します。単位はチックです。デフォルト値は0 (即ち、同一優先度のタスクスケジューリング) です。

複数のターゲットにそれぞれ異なる値をセットしてから起動することもできます。詳細はサイクルタイム、タスクのプライオリティのセクションを参照願います。

⇒ ISaGRAF カーネルタスクの起動

環境の設定が終わったら、isa_main ルーチンで ISaGRAF カーネルを起動します。

```
uchar isa_main
(
    uchar slave, /* スレーブ番号 */
    char *com     /* 未使用、空の文字列でも可 */
)
```

解説:

ISaGRAF カーネルタスクの起動

戻り値:

0 (正常時)、それ以外 (エラー時)

スレーブ番号は前述のスレーブ番号の登録で行った番号です。

スレーブ番号や通信ポートが重複しなければ、複数のカーネルを起動することができます。

⇒ ISaGRAF 通信タスクの起動

環境の設定が終わったら、ISaGRAF 通信タスク (tst_main_ex) を起動します。

```
uchar tst_main_ex
(
    char *com, /* 通信デバイス名 */
    uchar *slave, /* リンクするカーネルスレーブを指定する
                  4 バイトのフィールドの位置 */
    uchar com_id /* 通信タスクの ID */
)
```

解説:

ISaGRAF 通信タスクの起動

戻り値:

0 (正常時)、それ以外 (エラー時)

4バイトのフィールドで、通信リンクで接続されるカーネルタスクのスレーブ番号を指定します。もし、カーネルの数が4つ未満なら、フィールドの使わない部分をゼロクリアしておく必要があります。タスクが起動した後はこのフィールドは使われません。

通信デバイス名は通信リンクに設定されているシリアルデバイス名に相当します。論理番号が重複しなければ、複数の通信タスクが起動することもできます。

デバッガーとの通信時には、ワークベンチ側でのリンク設定はターゲット側と一致している必要があります。

☐ **VxWorksターゲットの起動例:**

以下に例を示します。

この例では、ISaGRAF カーネルタスクをスレーブ番号1で起動します。

ISaGRAF 通信タスクを起動します。(カーネルスレーブ番号1、通信タスク論理番号1、シリアルリンク通信デバイス名 /tyCo/1 で設定)

さらに、ISaGRAF 通信タスクを起動します。(カーネルスレーブ番号1、通信タスク論理番号2、ポート番号1100のイーサネット通信で設定)

カレントディレクトリはターゲットがインストールされているところです。

isassr.o モジュールのロード

ld < RELS/isassr.o

isakeret.o モジュールのロード(イーサネット通信が不要な場合は isakerse.o モジュールをロード)

ld < CMDS/isakeret.o

シリアル通信の設定

ISAMOD ("tyCo/1", 19200)

システムクロックレート設定

CLKRATE = sysClkRateGet ()

カーネルスレーブ番号登録

isa_register_slave (1)

通信論理番号の登録

isa_register_com (1)

isa_register_com (2)

バックアップファイル(デフォルト値なので省略可能)

TSK_FUNIT = ""

サイクルの最後の処理(デフォルト値なので省略可能)

TSK_NBTCKSCHED = 0

ISaGRAF カーネルの起動

sp (isa_main, 1, "")

通信タスクのスレーブリンク設定

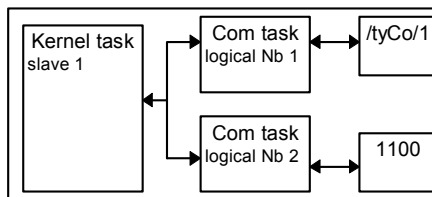
SlavesLink = 0x01000000

ISaGRAF 通信タスクの起動

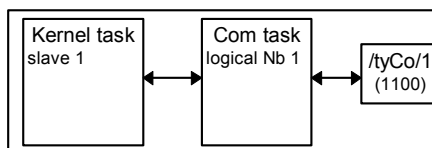
```
sp (tst_main_ex, "/tyCo/1", &SlavesLink, 1)
```

```
sp (tst_main_ex, "1100", &SlavesLink, 2)
```

以上の起動手順は下図のようになります。

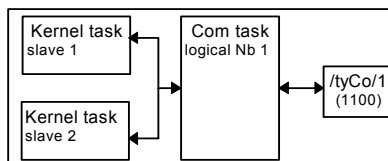


なお、以下のような構成をとることも可能です。



1 個のカーネルタスクに 1 個のシリアル(イーサネット)通信タスクがリンクした、最も基本的な構成です。

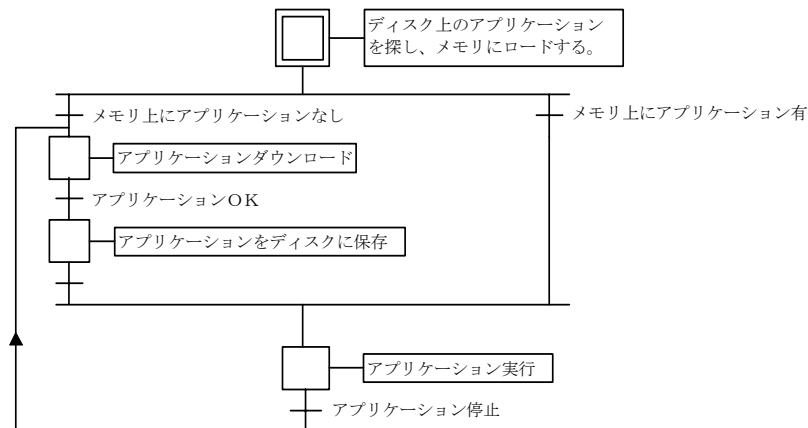
さらに、2つのカーネルタスクに対して1つの通信タスクを起動した場合の例を下図に示します。この場合のスレーブリンク設定は SlavesLink = 0x01020000 となります。



C.5.5 VxWorksターゲットの特徴

ISaGRAF ターゲットの起動

ISaGRAF ターゲットの起動は以下のアルゴリズムで行われます。



- 基本的な考え方

アプリケーションコード(中間コード)はワークベンチにより生成され、ダウンロードされたバイナリのデータベースです。これが、ISaGRAF ターゲットにより実行されます。シンボルテーブルが同時に使われる場合もあります。

アプリケーションのシンボルテーブルとはワークベンチにより生成され、ダウンロードされるテキスト形式のデータベースです。このファイルによってシンボル(変数名)とターゲット内部のデータとの関連付けを行います。ユーザがシンボルを使用しない場合はこのファイルはダウンロードする必要はありません。シンボルテーブルについての詳細はセクションAのアドバンスドプログラミングに解説されています。

- ISaGRAF の複数ターゲット

スレーブ番号や通信タスク論理番号が重複しなければ、同一CPU上で複数のISaGRAF ターゲット(カーネルタスクと通信タスク)を実行可能です。異なるアプリケーションであっても、同一I/Oボードのアクセスが可能であるため、アプリケーション側で注意しておく必要があります。例えば、I/Oドライバにセマフォ管理を入れるということも一つの方法です。

- アプリケーションのバックアップ

新しいアプリケーションがワークベンチ側からターゲットにダウンロードされると、アプリケーションコードは以下のファイル名で保存されます。(ターゲットでは標準的な fopen などのファイル管理ルーチンを使用します。)

pathISAx1 ISaGRAF アプリケーションコードのバックアップファイル(xはスレーブ番号)

さらに、アプリケーションのシンボルテーブルをあらかじめダウンロードしておく、以下のファイル名で保存されます。

pathISAx6 ISaGRAF アプリケーションシンボルテーブルのバックアップファイル(xはスレーブ番号)

path は ISaGRAF 起動時にグローバル変数 TSK_FUNIT で指定しす。デフォルトの(″)はディスクファイルユニットが存在しないことを意味します。ISaGRAF ターゲットの起動時にはカレントディレクトリでこれらのファイルを探して、見つければメモリ上にロードします。

シンボルファイルがない場合は、ターゲットはアプリケーションコードだけで実行されます。

アプリケーションコードがない場合は、ダウンロードされるのを待ちます。

ターゲットの電源ON時にデバッガーからのダウンロードなしで特定のアプリケーションをスタートさせるには、

- ・第一の方法として、ワークベンチをインストールしたパソコンからターゲットのカレントディレクトリにこれらのファイルをファイル転送ツールなどでコピーしておくようにします。操作を簡単にするにはワークベンチのツールメニューを利用してください(詳細は、セクションAを参照願います)。
- ・第二の方法として、ワークベンチのパソコンからアプリケーションコード(必要ならシンボルテーブルも)を不揮発性メモリ(PROM や EPROM)に格納するようなツールを作ることです。

こうすれば、システムの電源オン時に必要であれば(例えば、高速アクセスが必要だとか、ブレイクポイントの管理などの理由で)アプリケーションコード(必要ならシンボルテーブルも)を PROM から RAM にロードすることが可能です。

ISaGRAF タスクの起動前には、アプリケーションコードやアプリケーションシンボルテーブルのメモリ上でのアドレスを指定しておく必要があります。以下のようにして、SSRグローバル変数を初期化します。

SSR[x][1].space = アプリケーションコードのアドレス

必要に応じて、

SSR[x][6].space = アプリケーションシンボルのアドレス

この場合、短いプロシージャを作ることもできます。SSR グローバル変数は tasy0ssr.h ファイルに str_ssr 構造体としてを宣言されています。

注意: もしアプリケーションコードに書き込みが許されていない場合は(例えば、ROM化されているなど)、ISaGRAF のデバッガからのブレイクポイントの設定などは行えません。

もし、ISaGRAF ワークベンチが標準のディレクトリ(\\isawin)にインストールされていれば、プロジェクト(MYPROJ)のアプリケーションコード(即ち、中間コード)は以下ようになります。

\\ISAWIN\\<プロジェクトグループ名>\\APL\\MYPROJ\\appli.x6m

(ターゲット上での ISAx1 に相当)
同様に、アプリケーションシンボルファイルは以下のようになります。
`\\ISAWIN\\<プロジェクトグループ名>\\MYPROJ\\appli.tst`
(ターゲット上の ISAx6 に相当)

エラー処理とエラー出力メッセージ

ISaGRAF ターゲットソフトにはエラー検出処理が含まれています。エラーメッセージとその説明は本マニュアルの後の章で解説を行います。

エラー検出の流れは以下のようになります。

- エラーとエラー番号は ISaGRAF のエラールーチンへ送られます。
- もし、ワークベンチ側のコード生成メニューでエラー処理のフラグがセットされていれば、エラー処理がなされますが、そうでない場合はエラー情報は失われ、なにもしません。

エラーが処理される場合：

- エラー番号 (10進数) と引数 (16進数) 標準出力 (stdout) に表示されます。
- このエラー番号と引数はリングFIFOバッファに登録されます。バッファサイズはワークベンチのコード生成メニューから設定可能です。もし、FIFOバッファが一杯であれば新しいエラーが発生すれば、最も古いエラー情報が失われます。
- エラー情報はワークベンチのデバッガーウィンドウや、アプリケーション中の SYSTEM ファンクションをコールして取得できます。

ワークベンチのデバッガーがエラーを検出した場合は、エラーウィンドウにメッセージが表示されます。ここでは、アプリケーションの実行状態 (実行中、停止中) 表示に加えて、エラーが発生したプログラム名や変数名、エラー番号、引数を [] 内に表示します。

ターゲットでエラーが検出された場合、エラー番号はデフォルトの stdout 出力に表示されます。この表示のリダイレクトはVxWorksでは以下のルーチンを使います。

`ioGlobalStdSet()`

あるいは `ioTaskStdSet()`

後者の場合、カーネルや通信タスクはエラーを表示しません。

サイクルタイム、タスク、タスクのプライオリティ

- ISaGRAF ターゲットサイクルの最後に (次のサイクルがスタートする直前)、以下のアルゴリズムで処理がされます。

アプリケーションにサイクルタイムが指定されている場合は、残った時間 (指定サイクルタイム - 現在使用した時間) CPUを開放します。もし、この時間が負の値の時、サイクルオーバーフローとなり、CPUはスケジューリングのため、ISaGRAF 起動時にセットした TSK_NBTCKSCHED のチェック分だけ開放されます。

もし、サイクルタイムが指定されていない場合は、あるいは、残り時間が1チェック以下である場合は、CPUはスケジューリングのため ISaGRAF 起動時にセットした TSK_NBTCKSCHED のチェック分だけ開放されます。

ターゲットのサイクルタイムの精度はVxWorksのシステムチェックの精度に相当します。

通常、サイクルタイムの指定は、サイクルのトリガをかけるため、またはCPUの空き時間をVxWorks上の他のタスクに開放するためにおこないます。

ー 通信タスクは通信リンクからデータが来ないときはスリープ状態となっています。要求があれば、カーネルタスクとの Question/Answer プロトコルを使って実行中のアプリケーションの情報を取得します。通信タスクはカーネルに Question を投げかけ、カーネルのサイクルの最後に(これはアプリケーションデータの同期をとるため)カーネルは通信タスクに対して Answer を返します。

ISaGRAF のタスクは、自分のプライオリティを自らは変更しません。ユーザが全体の構成から判断して与えることになります。

C.6 NT ターゲットの使い方

C.6.1 ISaGRAF ターゲットの実行

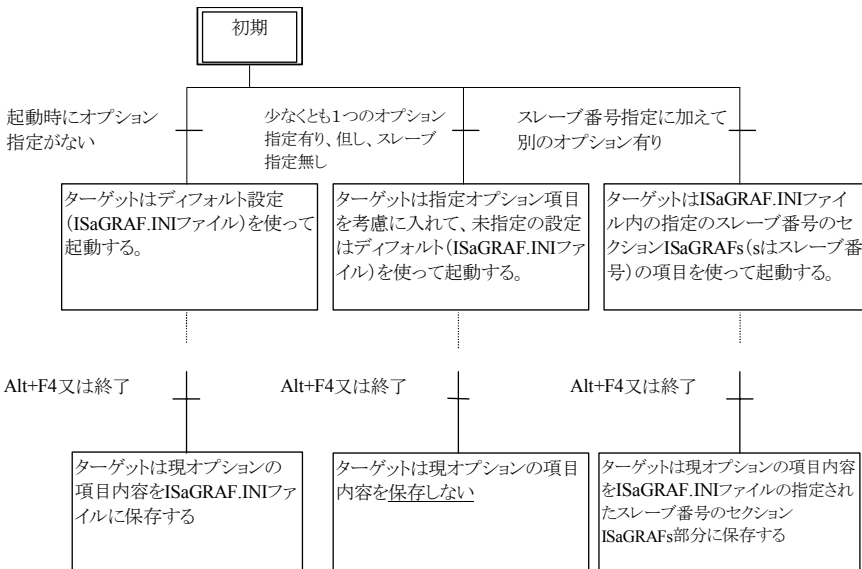
NTの環境では、ISaGRAF ターゲット(WISAKER.EXE)はカーネルスレッドと通信スレッドを持った1個のプログラムです。これを複数、同時に実行することもできます。ただし、WISAKER.EXE を起動する際にスレーブ番号が重ならないようにする必要があります。

本ターゲットの実行によって別の割り込み処理プログラムを妨げることはありません。

本ターゲットは、WindowsNT 3.51 以降 (NT4.0 が好ましい) で動作します。

C.6.2 各種オプションの概要

ターゲットのオプションの設定は下図のように保存／読み出しが行われます。



なお、ISaGRAF.ini ファイルは、カレントディレクトリに保存されます。

スレーブ番号の設定: -s オプション

このオプションはスレーブ番号を設定します。1～255(但し、13は除く)が使えます。スレーブ番号は通信リンクプロトコル内部で 사용됩니다。これはおもに複数のターゲットを1台のホストに接続する場合や、1台のパソコン上で複数のターゲットを実行する場合に、ターゲットを識別するためのものです。ワークベンチのデバッガを使う場合は、リンク設定でデバッグするターゲットとのスレーブ番号を一致させる必要があります。

デフォルト値: スレーブ番号のデフォルト値は1。または ISaGRAF.INI 内での設定値になります。

例:

NTターゲットをスレーブ番号2で起動します。

WISAKER.EXE -s=2

ユーザインタフェース: NTターゲットのメインウィンドウから「Options」-「Slave」コマンドで下記のウィンドウを表示します。



マウスや上下カーソルキーでスレーブ番号を変更できます。変更内容を有効にするにはNTターゲットの再スタートが必要になります。

通信リンクの設定: -t オプション

ISaGRAF ターゲットはデバッガー通信にシリアルリンク通信やイーサネット通信を使えます。ポートの指定に -t オプションを使います。シリアル通信には COM1, COM2, COM3, COM4 のいずれかの選択ができ、イーサネット通信にはポート番号 1100 以上のものが使えます。

デフォルト値: イーサネット通信ではポート番号は 1100 です。また、シリアル通信では COM1 となります。ただし、ISaGRAF.INI ファイルに設定値があればこれを優先します。

注意: デフォルトの通信リンクはイーサネット通信となっています。

例:

シリアル通信 COM2 を使ってターゲットを起動

WISAKER -t=COM2

イーサネット通信ポート番号 1101 を使ってターゲットを起動

WISAKER -t=1101

シリアル通信の設定:

-t=COMx オプションを付けている場合に限り以下のオプションが更に選択可能となります。

シリアル通信でのオプション設定項目

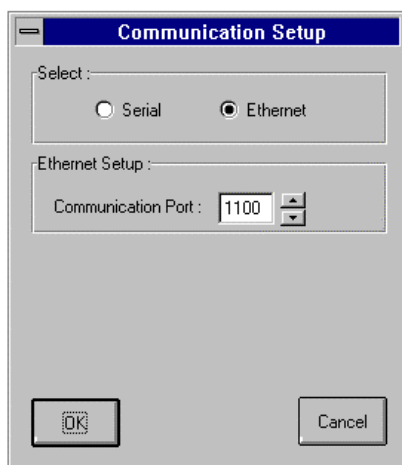
オプション	値	意味
Baud	600	ボーレート
	1200	
	2400	
	4800	
	9600	
	19200	
Parity	N	パリティなし
	E	偶数
	O	奇数
Data	7 または 8	データ長
Stop	1 または 2	ストップビット長さ
Flow	H	ハードウェア制御
	N	なし

上記の表でのデフォルト値は ボーレート 19200, パリティなし, 8 データ長, 1 ストップビット, フロー制御なし

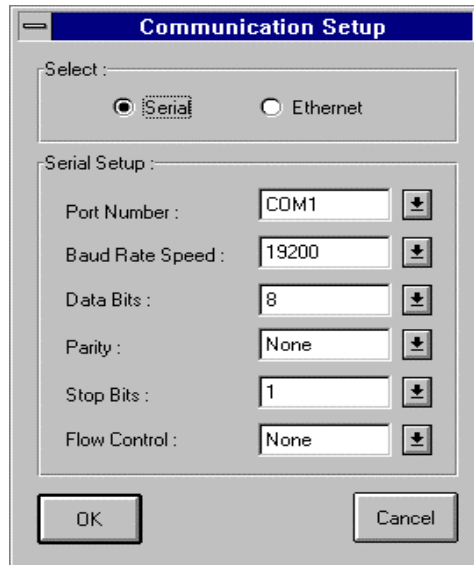
例:

WISAKER -t=COM1 baud=1200 data=8 parity=n stop=2

ユーザインタフェース: NTターゲットのメインウィンドウの「Options」-「Communication」コマンドで下記のウィンドウを表示します。



シリアル通信またはイーサネット通信を選択可能です。イーサネット通信を選択した場合はポート番号も変更可能です。番号はワークベンチ側のリンク設定内容と一致する必要があります。



シリアル通信を選択すると、シリアル用の設定画面が表示されます。設定内容はワークベンチ側のリンク設定内容と一致している必要があります。

■ バーチャルボードのグラフィックシミュレーション: -x オプション

もし、I/O接続エディタ(セクションA参照)でI/Oボードがバーチャルとして設定されていて、このオプションが指定されると、そのI/Oボードはシミュレーションされます。

設定可能な値は0または1です。1が指定されとグラフィックシミュレーションが行われます。

デフォルト値: デフォルト値は0です。または、ISaGRAF.ini ファイルの内容がデフォルト値となります。

例:

I/Oボードのシミュレーション付きでターゲットの起動

WISAKER -x=1

ユーザインタフェース: メニューのチェックで選択、非選択が選べます。

注意: グラフィックシミュレーションを行っているときは、リアルタイムターゲットのリアルタイム性は保証できませんのでご注意願います。サイクルタイムオーバーフローのエラーの頻度が高くなります。シミュレーションを非選択にすると正常な状態に戻ります。

ISaGRAF NT ターゲットのプライオリティの設定: -p オプション

ターゲットはNT上で実行されるので、プライオリティを指定することは有意義なことです。例えば、ある特定のクリティカルな ISaGRAF アプリケーションターゲットを高いプライオリティに設定して、その他のターゲットをより低いプライオリティに設定してバックグラウンドで実行させることができます。

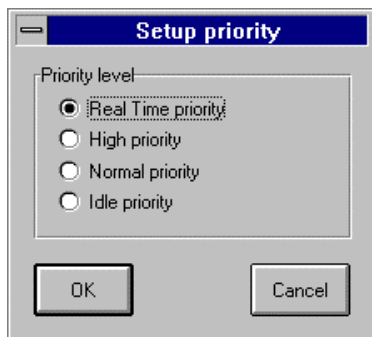
設定可能な値は0、1、2、3です。0が最も高く、3が最も低いプライオリティとなります。

例:

WISAKER -p=0

WISAKER -p=1

ユーザインタフェース: NTターゲットのメインウィンドウの「Options」-「Priority」コマンドで下記のウィンドウを表示します。



最も高いプライオリティは「Real Time」で最も低いものは「Idle」です。

0: Real time priority

1: High priority

2: Normal priority

3: Idle priority

注意: リアルタイムプライオリティは管理者の権限がないと設定できません。

例:

wisaker -t=COM1

スレブ番号をデフォルトの1、通信ポートを COM1 でターゲットを起動

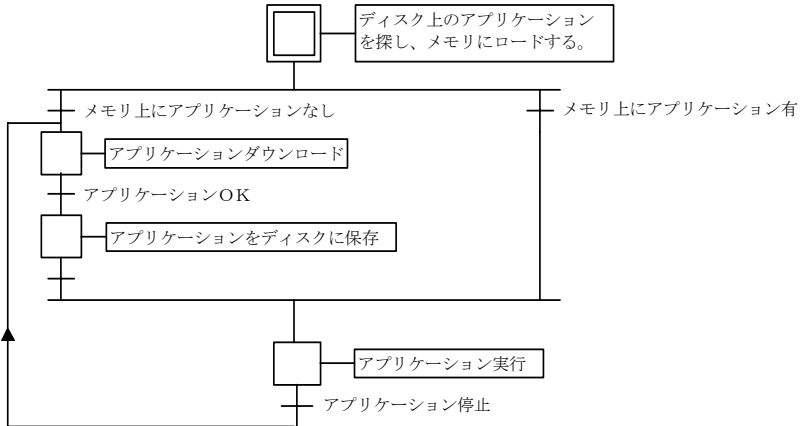
wisaker -s=3 -t=COM1

スレブ番号を3、通信ポートを COM1 でターゲットを起動

C.6.3 NTターゲットの特徴

ISaGRAF ターゲットの起動

ISaGRAF ターゲットは以下のアルゴリズムに従って実行されます。



● 基本的な考え方

アプリケーションコード(中間コード)はワークベンチにより生成され、ダウンロードされたバイナリのデータベースです。これが、ISaGRAF ターゲットにより実行されます。シンボルテーブルが同時に使われる場合もあります。

アプリケーションのシンボルテーブルとはワークベンチにより生成され、ダウンロードされるテキスト形式のデータベースです。このファイルによってシンボル(変数名)とターゲット内部のデータとの関連付けを行います。例えば、DDEインタフェースや変数名によるI/Oボードのシミュレーションにはこのシンボルテーブルが使われます。ユーザがこれらの機能を使用しない場合は不要です。シンボルテーブルについての詳細はセクションAのアドバンストプログラミングに解説されています。

● ISaGRAF 複数ターゲット

スレーブ番号や通信タスク論理番号が重複しなければ、同一CPU上で複数のISaGRAF ターゲット(カーネルタスクと通信タスク)を実行可能です。異なるアプリケーションであっても、同一I/Oボードのアクセスが可能であるため、アプリケーション側で注意しておく必要があります。例えば、I/Oドライバにセマフォ管理を入れるということも一つの方法です。

● アプリケーションのバックアップ

新しいアプリケーションがワークベンチ側からターゲットにダウンロードされると、アプリケーションコードは以下のファイル名でターゲットのカレントディレクトリに保存されます。

ISAx1 ISaGRAF アプリケーションコードのバックアップファイル(xはスレーブ番号)

さらに、アプリケーションのシンボルテーブルをあらかじめダウンロードしておくと、以下のファイル名で保存されます。

ISAx6 ISaGRAF アプリケーションシンボルテーブルのバックアップ
ファイル(xはスレーブ番号)

ISaGRAF ターゲットの起動時にはカレントディレクトリでこれらのファイルを探して、見つければメモリ上にロードします。

シンボルファイルがない場合は、ターゲットはアプリケーションコードだけで実行されます。

アプリケーションコードがない場合は、ダウンロードされるのを待ちます。

ターゲットの電源ON時にデバッガーからのダウンロードなしで特定のアプリケーションをスタートさせるには、これらのファイルをターゲットのカレントディレクトリにコピーしておくようにします。ターゲットとワークベンチが同じパソコンならディスク間でコピーし、異なる場合はフロッピーディスクなどを使います。

もし、ISaGRAF ワークベンチが標準のディレクトリ(\isawin)にインストールされていれば、プロジェクト(MYPROJ)のアプリケーションコード(即ち、中間コード)は以下ようになります。

\\SAWIN\\<プロジェクトグループ名>\\APL\\MYPROJ\\appli.x8m

同様に、アプリケーションシンボルファイルは以下のようになります。

\\SAWIN\\<プロジェクトグループ名>\\MYPROJ\\appli.tst

(ターゲット上での ISAx1 に相当)

例:

WISAKER.EXE がインストールされているディレクトリで以下のコマンドを実行すると WISAKER.EXE は `myproj` アプリケーションを見つけて実行します。

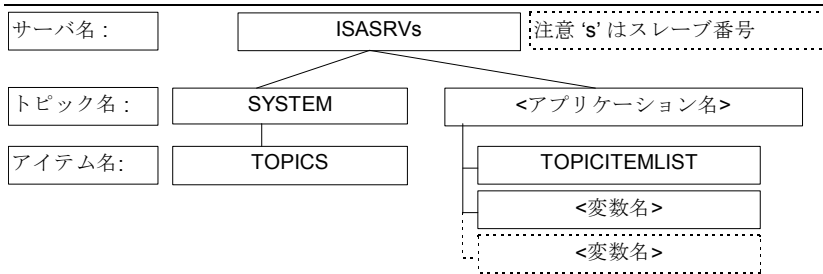
copy \\SAWIN\\<プロジェクトグループ名>\\MYPROJ\\appli.x8m isa11

このようなコマンドをバッチファイルにしてワークベンチの「ツール」メニューにユーザが追加することが可能です。(詳細は、セクションAを参照願います。)

□ **DDE の仕様**

NTターゲットはDDE(Dynamic Data Exchange)サーバでもあります。各種ソフトウェアがDDEクライアントとしてNTターゲットと接続して、変数のやりとりをすることが可能です。例えば、MS-ExcelではDDE通信を介してNTターゲットの整数型変数のグラフィックアニメーションを行うことができます。

DDEにはアプリケーションシンボルテーブルファイルが必要です。DDEに関係する項目は以下ようになります。



- 「 ISASRVs 」 DDEサーバ名で、's' はスレーブ番号です。
- 「 SYSTEM 」 標準トピック名です。「 TOPICS 」アイテムにアクセスするためのものです。
- 「 TOPICS 」 定義されているトピックのリストを与えます。これは実行中の ISaGRAF ターゲットのアプリケーション名を指します。
- 「 アプリケーション名 」 ISaGRAF アプリケーション名です。
- 「 TOPICITEMLIST 」 現在のトピックで利用可能なアイテムのリストで、DDE経由で選択可能な変数のリストを与えます。
- 「 変数名 」 アプリケーション変数名です。

例： EXCEL のセルで ISaGRAF アプリケーション"rfwash"の変数"waterlevel"の値を表示

= ISASRV1|rfwash!waterlevel

☐ DDEアドバイスループのレート: -d オプション

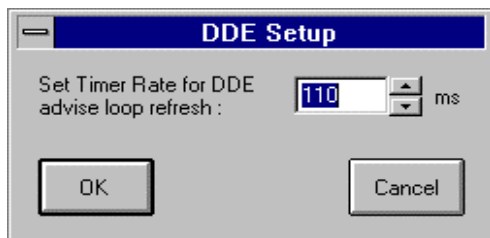
DDEクライアントからのDDEリクエスト(ポーリングによるリンク)は変数の数が増えるとポーリングのためのオーバーヘッドが大きくなります。これに対してアドバイスループはサーバ自身が変更があった変数のみをクライアントに渡します。この方法で通信オーバーヘッドは最小限に押さえることができます。このモードではサーバが定期的にアドバイスループに指定された変数に変化が生じていないかチェックします。このチェックの周期をDDEアドバイスループレートといいます。

このオプションを使うことで、DDEアドバイスループの設定時間をミリ秒単位で設定することができます。

デフォルト値： デフォルト値は 1000 ms 又は、ISaGRAF.INI ファイルに指定されている値です。

例：
WISAKER -d=100

ユーザインタフェース: NTターゲットのメインウィンドウの「Options」-「DDE」コマンドで下記のウィンドウを表示します。



エラー管理とエラー出力メッセージ

ISaGRAF ターゲットソフトにはエラー検出処理が含まれています。エラーメッセージとその説明は本マニュアルの後の章で解説を行います。

エラー検出の流れは以下のようになります。

- エラーとエラー番号は ISaGRAF のエラールーチンへ送られます。
- もし、ワークベンチ側のコード生成メニューでエラー処理のフラグがセットされていれば、エラー処理がなされますが、そうでない場合はエラー情報は失われ、なにもしません。

エラーが処理される場合：

- エラー番号(10進数)と引数(16進数)が WISAKER.EXE のウィンドウに表示されます。
- このエラー番号と引数はリングFIFOバッファに登録されます。バッファサイズはワークベンチのコード生成メニューから設定可能です。もし、FIFOバッファが一杯であれば新しいエラーが発生すれば、最も古いエラー情報が失われます。
- エラー情報はワークベンチのデバッガーウィンドウや、アプリケーション中の SYSTEM ファンクションをコールして取得できます。

ワークベンチのデバッガーがエラーを検出した場合は、エラーウィンドウにメッセージが表示されます。ここでは、アプリケーションの実行状態(実行中、停止中)表示に加えて、エラーが発生したプログラム名や変数名、エラー番号、引数を[]内に表示します。

ターゲットの起動時にスレーブ番号、通信ポートの設定、DDE サーバ名が表示されます。

システムクロック

NTターゲットはハードウェア固有の影響を受けないように、サイクルの同期やタイマ変数のリフレッシュには標準のタイマチックである10msを使います。

したがって、10msより精度の高いタイマ変数を定義できません。同様な理由で、10ms以下のサイクルタイムを設定した場合はエラー番号62のサイクルタイムオーバーフローが発生します。詳細は以下の節を参照願います。

- 註: 現状のNTターゲットではPCハードウェアシステムによって高精度なマルチメディアタイム(1ms分解能)を使用できるものに対応したシステムクロック処理部を取り入れてあります。この場合は1msの精度のタイム変数を定義できます。

サイクルタイムとタスクの動作

ISaGRAF ターゲットサイクルの最後に(次のサイクルがスタートする直前)、以下のアルゴリズムで処理がされます。

アプリケーションにサイクルタイムが指定されている場合は、残った時間(指定サイクルタイム - 現在使用した時間)CPUを開放します。もし、この時間が負の値の時、サイクルオーバーフローとなり、CPUはスケジューリングのため1チック分だけ開放されます。

もし、サイクルタイムが指定されていない場合、あるいは、残り時間が1チック以下である場合は、CPUはスケジューリングのため1チック分だけ開放されます。

従って、NTターゲットのタイミングの精度はWindowsNTのシステムチック(例えば10ms)に対応します。

通常、サイクルタイムの指定は、サイクルのトリガをかけるため、またはCPUの空き時間を他のタスクに開放するためにおこないます。

設定サイクルタイムがアプリケーションプログラムに対して十分余裕がないと、他のタスクが重くなりますので注意が必要です。また、CPUの性能やメモリ容量が十分でない場合にも同様の現象が起きます。

終了キー

デスクトップPCでNTターゲットをテストしている場合、ISaGRAF ターゲットの実行を止める場合があります。このときのキー入力は次のようになります。

alt + F4

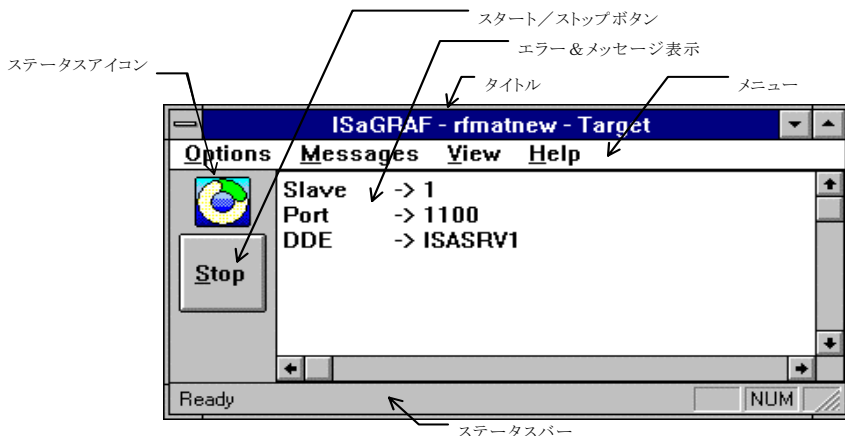
この停止方法ではI/Oボードの処理がクローズされないで終了する危険性もあります。従って、通常の ISaGRAF ターゲットの停止には以下の手順をお奨めします。

- ワークベンチデバッグから停止するか、ターゲット(WISAKER.EXE)ウィンドウの STOP ボタン/メニューを選択する。この時点でI/Oボードのクローズ処理が行われます。
- その後、システムメニューの「閉じる」で終了する。

尚、デバッグがオンライン接続状態中にターゲットウィンドウを閉じる場合に、デバッグが応答しなくなる場合もありますが、この場合は強制的にNT上から終了して下さい。

C.6.4 ユーザインタフェース

ISaGRAF のNTターゲットのGUIの解説を以下に行います。



主な構成要素は

- ウィンドウのタイトル
- メニューバー
- ステータスアイコン
- スタート/ストップボタン
- エラー & メッセージ表示
- ステータスバー

です。

ウィンドウのタイトルの内容は「 ISaGRAF - アプリケーション名 - target 」、となり、実行中のアプリケーション名が表示されます。実行中のアプリケーションがない場合は「 ISaGRAF - - Target 」の表示となります。

ISaGRAF NT ターゲットのメニューバー:

メニューバーには4つのメニューがあります。

- Options
- Messages
- View
- Help

• 「Options」メニュー

(セクション「各種オプションの概要」もご覧下さい)

「Options」メニューからは以下のオプションの設定が可能です。

Slave はスレーブ番号の変更を行います。変更したオプション値はターゲットの再スタート後に反映されます。このオプションはターゲット起動時にコマンドラインからオプション付きで起動された場合は使用できません。

Communication は通信の設定を行います。変更されたオプション値はターゲット再スタート後に反映されます。このオプションはターゲット起動時に -s オプション以外のオプションが設定された場合には使用できません。

DDE はDDEアドバイスループのレート設定を行います。変更されたオプション値はターゲットの再スタート後に反映されます。このオプションはターゲット起動時に -s オプション以外のオプションが設定された場合には使用できません。

Simulate I/O はチェックボックスをチェックしてあるか、そうでないかで設定されます。このオプション値はターゲットの再スタート後に反映されます。また、ワークベンチのI/O接続エディタでI/Oボードをバーチャルに設定しておく必要もあります。

Priority はプライオリティの変更を行います。変更されたオプション値は直ちに反映されます。

Default Options は以下のオプション設定のデフォルト値を読み出します。

- Communication
- DDE
- ウィンドウの表示場所とサイズ

変更されたオプション値はターゲットの再スタート後に反映されます。このオプションはターゲット起動時に -s オプション以外のオプションが設定された場合には使用できません。

• 「Messages」メニュー

「Messages」メニューは表示の管理を行います。

Acknowledge はエラー表示がなされている場合の確認のためのコマンドです。ステータスアイコンの赤の点灯が消えます。

Clear は表示されている全メッセージをウィンドウから消去します。

ISaGRAF NT ターゲットアイコン:

ステータスアイコンはNTターゲットの状態を表示します。

- アプリケーションの実行中はアイコンが回転します。
- アプリケーションがない場合（あるいはアプリケーションが停止中）はアイコンの回転は停止します。
- エラーメッセージが表示されているときはアイコンの中心が赤の点滅を行います。赤の点灯を止める場合は、「Messages」メニューの「Acknowledge」コマンドを選択するか、「Clear」コマンド（この場合、エラーメッセージはすべて消去されてしまいます）を選択します。エラーに関する情報は後述します。

以下に、アイコン状態を示します。

エラーなし

エラー表示有り
(中心部が赤の点灯)

アプリケーション実行中

アプリケーションなし

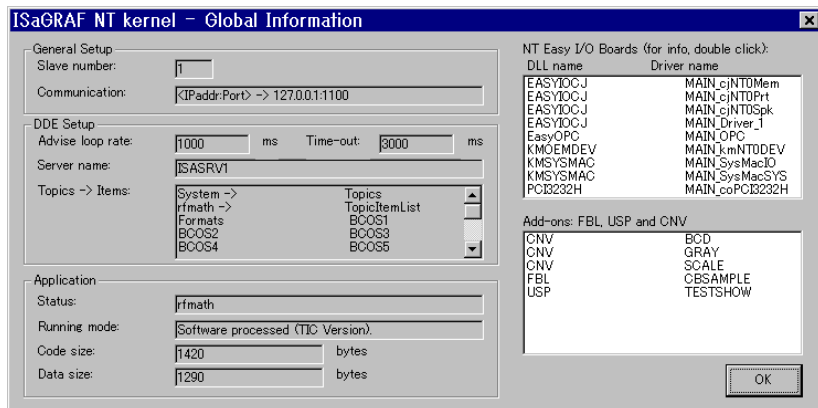


ISaGRAF NT ターゲットの Start/Stop ボタン:

NTターゲットウィンドウの Start/Stop ボタンはデバッガーのスタート/ストップコマンドと同等の機能を持ちます。ボタン上の文字はターゲットの状態を反映しています。すなわち、ターゲットが実行中はボタン上の文字 « Stop » と表示され、停止中は « Start » と表示されます。もし、アプリケーションが存在しない場合、« Start » 表示がなされているときにボタンを押しても再び « Start » 表示に戻ります。

ISaGRAF NT ターゲットの全体の情報

NTターゲットウィンドウの「View」-「Information」コマンドで、以下のダイアログボックスが表示されます。これにはターゲット構成や実行中のアプリケーションの構成情報が表示されます。



このダイアログボックスには3つのトピックが含まれています。

a) ターゲット設定:

- スレーブ番号
- 通信設定（通信リンクがイーサネットの場合、ポート番号と、NT上で使えるIPアドレス番号も表示されます）

b) DDE 設定:

- アドバンスループレート
- DDE サーバ名
- DDE トピック名とアイテム名。アプリケーション名や変数名などになります。

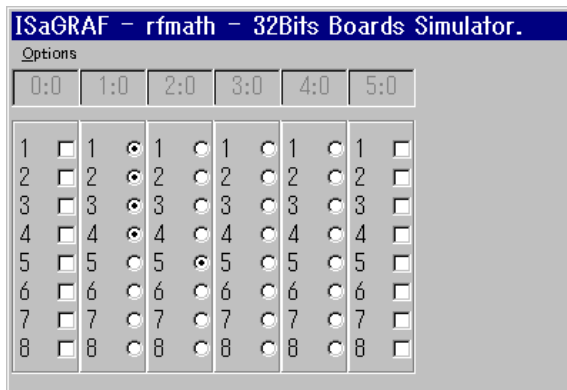
c) アプリケーション:

- アプリケーションが実行中はそのアプリケーション名、もしアプリケーションがない場合は 'No application' の文字が表示されます。

- アプリケーションの実行モードを表示します。具体的にはアプリケーションが中間コード(TICコード)をソフトウェア処理(インタプリト)されながら実行している場合は「Software processed (TIC Version)」と表示されます。また、Cソースコードを元にコンパイルされて実行している場合は、「C compiled」と表示されます。アプリケーションが存在しない場合は「No application.」と表示されます。
- アプリケーションコードのサイズをバイトで表示します。もし、実行モードが「C compiled」となっている場合はこのフィールドは0となります。
- データサイズをバイト表示します。このデータサイズはターゲット内のリアルタイムデータサイズと変数テーブルサイズの合計を指します。

ISaGRAF NT ターゲットのバーチャルボードのシミュレーション

NTターゲットウィンドウの「Option」-「Simulate I/O」コマンド が選択されている場合は、アプリケーションがスタートすると以下のシミュレーションウィンドウが開きます。



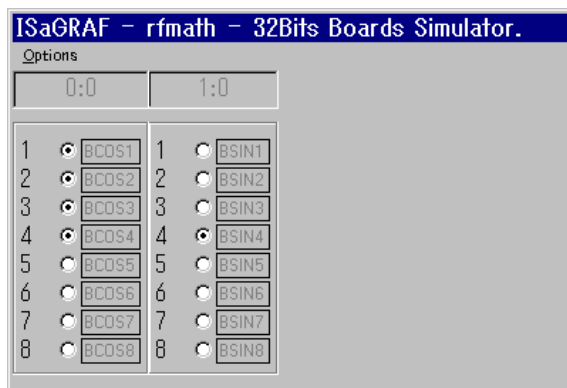
ウィンドウの内容はI/O接続エディタの構成(I/Oボードの数、変数名の数など)によって変わります。各ボードの上部の番号 < s : b > はスロット番号とボードの番号(識別子)を指します。この番号は0から始まり、変更することではできません。ボードがバーチャルあるいはシミュレーションボードが選択されていて、かつ、「Simulate I/O」がチェックされているときに、このシミュレータウィンドウはStart/Stop ボタンに合わせて開いたり、閉じたりします。このウィンドウはI/Oドライバルーチンのコールにより呼び出されます。

シミュレータウィンドウの「Options」メニューは2つの項目があります。

Variable names は変数名をボードの右横に表示します。ただし、アプリケーションシンボルテーブルがあらかじめ中間コードより前にダウンロードされているときに限ります。

Hexadecimal values は各整数型値を16進数表示することができます。デフォルト表示は10進数です。

例えば、変数名表示を行うと以下のようになります。



C.7 C言語プログラミング

C.7.1 概要

本マニュアルは既に ISaGRAF のコンセプトを理解し、ワークベンチの使用経験を持ったユーザを対象とするものです。ISaGRAF の標準ライブラリ中の**変換関数**、**C言語ファンクション**、**C言語ファンクションブロック**を使って自動化アプリケーションが開発できるようになれば、次は「ユーザ定義」の変換関数、C言語ファンクション、C言語ファンクションブロックを開発することができます。これは、ユーザが新しいライブラリを作成し ISaGRAF ターゲットの機能拡張が可能だということを意味します。

Cの開発環境と、ある程度のC言語プログラミングの経験があれば、マニュアルを使って制御に最適な ISaGRAF ターゲットのカスタマイズが可能になります。このことにより、ターゲット PLC の性能のみならず、ISaGRAF ワークベンチの利用環境の質的な向上も可能にします。

このドキュメントの中の情報は特定のターゲットシステムに限定したものではありません。しかしながら、シングルタスクシステムには適用できないようなものがあります。(マルチタスクキング機能を要求するようなもの等)

標準の ISaGRAF ワークベンチの特徴

ISaGRAF ワークベンチは多くのC言語ライブラリの管理機能を提供します。自動化アプリケーション側から見ればC言語変換関数、C言語ファンクション、C言語ファンクションブロックは“**ブラックボックス**”であり、そのインターフェースによってのみ定義されるものです。

ISaGRAF ライブラリ管理は、既存のライブラリに新たなコンポーネントを追加や新たな **ST/FBD** プログラミングで使用するためのインターフェースの定義のために使用します。さらに ISaGRAF ライブラリ管理はC言語変換関数、C言語ファンクション、C言語ファンクションブロックのC言語のソースコードのテンプレート生成と、編集機能も持っています。ライブラリ管理についての詳細は **ISaGRAF ユーザガイド(セクションA)**を参照してください。

C言語による開発

ISaGRAF ワークベンチにはC言語コンパイラやクロスコンパイラは含まれていません。ユーザが ISaGRAF のカーネルにC言語のコンポーネントを組み込むためには、ターゲットのシステム用のC言語コンパイラを別途用意しなければなりません。

クロスコンパイラをお使いの場合には、ユーザ定義の MS-DOS のバッチファイル (*.bat)を実行するために ISaGRAF ワークベンチの「ツール」メニューを使うことができます。この場合、使用するクロスコンパイラは DOS ウィンドウで実行できなければなりません。もしそうでない場合は Windows を終了し、純粋な DOS の上でコンパイル、リンクを実行して下さい。

☐ 技術メモ

ISaGRAF ライブラリマネージャでは、ユーザがそれぞれのライブラリのコンポーネント毎にテキスト形式の説明を記述できるようになっています。この**技術メモ**はアプリケーション開発者のための説明書であり、アプリケーションで正しく変換関数やファンクション、ファンクションブロックを記述するための有益な情報となります。

変換関数、C言語ファンクション、ファンクションブロックを、アプリケーション開発者がそれらを ISaGRAF の関数として使用するためには、技術メモの中で使い方を正確に定義しなければなりません。

C言語ファンクションの技術メモは、以下のように記述しなければなりません

- ☐ ファンクションの処理内容の詳細
- ☐ 引数(入力パラメータ)の説明
- ☐ 戻り値(出力パラメータ)の説明
- ☐ 引数、戻り値の詳細
- ☐ 使用法・条件など

C言語ファンクションブロックの技術メモは、以下のように記述しなければなりません。

- ☐ ブロックの処理内容の詳細
- ☐ 引数(入力パラメータ)の説明
- ☐ 戻り値(出力パラメータ)の説明
- ☐ 引数、戻り値の詳細
- ☐ 使用法・条件など

変換関数の技術メモは、以下のように記述しなければなりません。

- ☐ 入力変数に使用する場合の正確な意味
- ☐ 出力変数に使用する場合の正確な意味
- ☐ 処理可能な値の制限(限界)

さらに、技術メモには以下のような情報も必要となります。

- ☐ 変換関数、ファンクション、ファンクションブロック分別用の識別子(名前)
- ☐ メンテナンスや更新のための情報
- ☐ サポートするターゲットシステム
- ☐ マルチタスクシステムに特有の特徴
- ☐ 要求されるシステムのサービスやメモリ、ドライバなど

C.7.2 C言語変換関数

ISaGRAF ワークベンチには単純なアナログ入出力の変換をターゲット上で実行するための**線形変換テーブル**が用意されています。この変換テーブルはC言語による開発は必要ありませんが、関数は単調増加あるいは減少のみという制限があります。このツールに関する詳しい説明は ISaGRAF ユーザガイドを参照して下さい。

変換テーブルで対応できない場合は、C言語で特殊な操作を記述することで、ユーザはどんな複雑な変換でも利用できるようになります。基本的には、変換関数は**入力変数**及び**出力変数**に対して定義することができます。どちらか一方しか使わなくても、システムのクラッシュを防ぐために、実装やテストは ISaGRAF カーネルに変換関数を組み込む前に行われなければなりません。

変換関数はC言語で記述し、コンパイルし、ISaGRAF カーネルとリンクします。新たに機能拡張されたカーネルをターゲット上にインストールしてから、拡張機能をプロジェクトで使用するようにします。新しい変換関数の機能はワークベンチのシミュレータには実装されません。ISaGRAF アプリケーションのシミュレーションは、非標準の変換関数を使用する前に行う必要があります。

ICS Triplex ISaGRAF Inc.社が提供する標準の変換関数のC言語のソースコードは、ISaGRAF ワークベンチに組み込まれています。これらは新しい変換関数を作成するための例として使用できます。色々なアプリケーションでの互換性を保つため、この標準変換関数は**改造するべきではありません**。標準の変換関数は ISaGRAF シミュレータでもサポートされています。

注意: 変換関数は、アプリケーションの入力と出力の処理に**同期して** ISaGRAF I/Oマネージャによって実行されます。変換関数の処理時間は、ISaGRAF のアプリケーションの**サイクルタイム**に含まれます。サイクルタイムを必要以上に延ばさないためには、ユーザは変換関数の中でウェイトのオペレーションを使用しないことが必要です。

新しい変換関数の ISaGRAF ライブラリへの追加

ISaGRAF ライブラリマネージャを使って、ワークベンチ上の ISaGRAF ライブラリに新しい変換関数を追加します。変換関数ライブラリを選択しておき、「**ファイル**」→「**新規作成**」メニューを選びます。変換関数はあらかじめ定義されたインターフェースを使用するので、ワークベンチ上ではパラメータ無しで定義します。

新規作成の後、変換関数の**技術メモ**を記述します。新しい変換関数のC言語のテンプレートが自動的に作成されます。

ISaGRAF プロジェクトでの変換関数の使い方

定義された変換関数は、入力や出力のアナログ変数のフィルタとして使用できます。変数に変換関数を割り付けるには、辞書エディタを起動しアナログ入出力変数を選択しパラメータを入力します。「**変換**」のフィールドがその設定のためのものです。

変換:

(なし)	↓
(なし)	
bcd	
scale	

リストには変換関数と変換テーブルの両方がでてくるので、重複する名前を定義する事はできません。ISaGRAF カーネルに未実装、あるいは未定義の変換関数を変数に割り当てることはできません。

標準C言語インタフェース

変換関数のインタフェースは常に同じフォーマットになります。引数と戻り値は構造体によって受け渡しされます。この構造体は"**TACN0DEF.h**"ファイルで定義されています。

```
/*
   Name: tacn0def.h
   Target conversions definition file
*/

#define DIR_INPUT 0                /* direction = input conversion */
#define DIR_OUTPUT 1              /* direction = output conversion */

typedef int32  T_ANA;              /* integer ANA type */
typedef float  T_REAL;            /* real ANA type */

typedef struct {                  /* conversion structure */
    uint16 number;               /* conversion number (reserved) */
    uint16 direction;            /* conversion direction */
    T_REAL *before;              /* value before conversion */
    T_REAL *after;               /* value after conversion */
} str_cnv;

#define ARG_BEFORE (*(arg->before))
#define ARG_AFTER  (*(arg->after))
#define DIRECTION  (arg->direction)

/* eof */
```

"**str_cnv**"構造体がインタフェースを定義しています。変換関数のCの引数はこの構造体のポインタだけです。構造体のメンバ"**number**"は変換関数の番号 (ISaGRAF ライブラリの位置) であり、プログラム中で使用する必要のないものです。

構造体のメンバ"**direction**"は変換関数が入力変数に使うのか、出力変数に使うのかを示します。その値は、入力変換の時は **DIR_INPUT**、出力変換の時は **DIR_OUTPUT** となります。

構造体のメンバ"**before**"は変換前の値へのポインタです。このメンバ変数は入力変換か出力変換かで違った意味を持ちます。"**direction**"が **DIR_INPUT** の場合、入力変換関数に対しては、入力デバイスを読んだ値になります。"**direction**"が **DIR_OUTPUT** の場合、出力変換関数に対してはプログラムされた論理式に従った値になります。

構造体のメンバ"**after**"は変換後の値へのポインタです。このメンバ変数は入力の変換か出力の変換かで違った意味を持ちます。"**direction**"が **DIR_INPUT** の場合、入力変換関数に対しては、プログラムされた論理式に従った値になります。"**direction**"が **DIR_OUTPUT** の場合、出力変換関数に対しては、出力デバイスに出力する値になります。

プログラマは、**before** と **after** の構造体のメンバを直接アクセスするために "**ARG_BEFORE**" と "**ARG_AFTER**" のマクロを使用できます。加工された変数は**単精度浮動小数点型**変数です。変換が整数型変数に割り付けられた場合、変換結果は **long** 型の整数値になります。これは同じ変換関数を実数、整数両方のアナログ入出力値に対して使用できるということを意味します。



ソースコード

変換関数は、入力、出力変数の両方に使用さうので、Cソースコードは入力変換部と出力変換の二つの部分に分かれます。構造体のメンバ"**direction**"はこれらのいずれかを選択するために使います。ISaGRAF ライブラリマネージャが自動生成する関数のテンプレートには **IF** 文の構造もあります。以下のリストが変換関数のC言語ソースコードのテンプレートです。

```
/*
    conversion function
    name: sample
*/

#include <tasy0def.h>
#include <tacn0def.h>

void CNV_sample (str_cnv *arg)
{
    if (DIRECTION == DIR_INPUT) { /*INPUT CONV*/

    }
    else { /*OUTPUT CONV*/

    }
}

/* 以下の関数は変換関数の名前によって ISaGRAF のI/Oマネージャとのリンク
を定義しています。この関数は ISaGRAF ライブラリマネージャによって自動的に
生成されます。*/

UFP cnvdef_sample (char *name)
{
    sys_strcpy (name, "SAMPLE"); /* 変換関数の名前を与える */
    return (CNV_sample);        /* 関数のアドレスを返す*/
}
```

関数の機能を完成させる最も良い手段は、入力変換と出力変換を別々にローカル関数として記述することです。これらの関数は、上記のリストで示すようにメインルーチンの **IF** 文の構造によってそれぞれコールされます。

インクルードファイル"**TASY0DEF.H**"には、システムに依存する定義が含まれています。その中には、**UFP** 型も定義されています。これは、**void** 型の関数のポインターを指し、関数宣言のために使用されます。

プロジェクトとC言語変換関数との関連付け

変換関数の実装と ISaGRAF プロジェクト内で使う変換関数との関連付けは、変換関数の名前によって行われます。「宣言」用の関数部分が変換関数のCソースコードに付加されています。この宣言関数部はアプリケーションがスタートした時に1回だけコールされ、ISaGRAF I/Oマネージャに変換関数の名前を伝えます。以下に標準的な宣言関数のフォーマットを示します。

```
UFP cnvdef_XXX (char *name)
{
    strcpy (name, "XXX");           /* 変換関数名 */
    return (CNV_XXX);              /* 関数の戻り値 */
}
/* (XXX は変換関数名) */
```

strcpy 文の中で使用される関数の名前は、**大文字**でなければいけません。そして変換関数のコード中及び宣言関数中の名前は小文字でなければなりません。

"CNV_"や"cnvdef_"の接頭語を使用することによって、既存のC言語の予約語や ISaGRAF のライブラリの名前に干渉することなく、ユーザが関数に名前を付けることができます。

変換関数で特別な初期化を行うために、他のステートメントを宣言関数部に追加することもできます。ISaGRAF システムでは、この宣言関数部はアプリケーションがスタートする時**1回だけ**コールされることに注意して下さい。

宣言関数部は、たとえ ISaGRAF アプリケーション中で使用されていなくてもコールされます。ISaGRAF のアプリケーションで使用されている変換関数がカーネルに組み込まれていない場合は、致命的なエラーを起こしますので注意して下さい。

新しい変換関数をカーネルとリンクする前に、"**GRCN0LIB.C**"という名前のソースコードを編集し、リストに新しい変換関数を追加しなければなりません。"**GRCN0LIB.C**"にはただ宣言関数部の定義の配列があるだけです。この配列は、C言語で記述された変換関数と動的にリンクするために、アプリケーションが初期化される時に読みこまれます。以下にそのファイルの例を示します。

```
/* File "GRCN0LIB.c" - 変換関数の標準ライブラリ例 */

#include <tasy0def.h>                               /* required for types definition */

extern UFP cnvdef_scale (char *name);              /* declaration function for SCALE conv */
extern UFP cnvdef_bcd (char *name);                /* declaration function for BCD conv */

UFP_LIST CNVDEF[ ] = {                             /* array of declaration functions for */
    cnvdef_scale,                                   /* integrated conversion functions */
    cnvdef_bcd,
```

```
NULL };
```

```
/* end of file */
```

CNVDEF の配列は NULL ポインタで終わらなければなりません。さもないと実行時にクラッシュする可能性があります。**CNVDEF** の配列が未定義だと、新しい ISaGRAF カーネルをリンク生成する時に、「未解決の参照」といったエラーが起きます。

このファイルに変換関数名を追記することにより、新しいカーネルは、既存の変換関数に追加される形で作られることになります。また、**CNVDEF** 配列にプロジェクトで使用する変換関数のみ記述し、そのプロジェクトに特化したカーネルにカスタマイズすることも可能です。アプリケーションコードの生成時に、自動的に "**GRCNOLIB.C**" ファイルが生成されます。このファイルは、ISaGRAF のプロジェクトディレクトリに作られます。そしてプロジェクトで使用している変換関数のみをグループ化しています。

制限

ISaGRAF ライブラリは最大 **128** 個の変換関数を持つことができます。どんなタイプの処理でも変換関数の中で実行できます。しかし、変換関数は ISaGRAF のサイクルに**同期して**呼び出されるので、関数の実行時間は直接サイクルタイムに影響を与えるという点に注意してください。

C.7.3 C言語ファンクション

C言語ファンクションは標準の IEC61131-3 言語の拡張のために使用します。C言語ファンクションによって、システムコールをはじめ、特殊な演算、通信、ISaGRAF アプリケーションと他のタスクとのコミュニケーションのためのサービスを実現することができます。

C言語で記述したファンクションはコンパイルし、ISaGRAF カーネルとリンクします。新たに機能拡張されたカーネルをターゲット上にインストールしてから、拡張機能をプロジェクトで使用するようにします。

新しいファンクションの機能はワークベンチのシミュレータには実装されません。ISaGRAF アプリケーションのシミュレーションは、非標準のファンクションを使用する**前**に行う必要があります。

注意: C言語ファンクションは、アプリケーションのサイクルに**同期して** ISaGRAF カーネルによって実行されます。関数の処理時間は、ISaGRAF のアプリケーションの**サイクルタイム**に直接影響します。サイクルタイムを必要以上に延ばさないためには、ユーザは変換関数の中でウェイトのオペレーションを使用しないことが必要です。

C言語ファンクションの ISaGRAF ライブラリへの追加

ISaGRAF ライブラリマネージャを使って、ワークベンチ上の ISaGRAF ライブラリに新しい C言語ファンクションを追加します。C言語ファンクションライブラリを選択しておき、「**ファイル**」―「**新規作成**」メニューを選びます。新規作成の後、**技術メモ**

を記述して下さい。新しい関数のC言語のテンプレートが自動的に作成されます。

メニューの「**編集**」-「**パラメータ**」を選択し、関数の入力パラメータと出力パラメータを定義します。

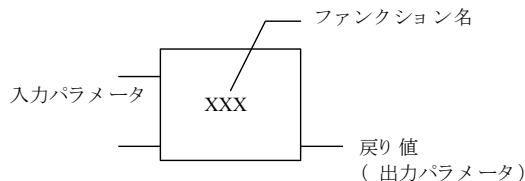
ISaGRAF プロジェクトでC言語関数の使い方

組み込まれたC言語関数は、ISaGRAF プロジェクトのプログラム中で標準関数と同様に使用することができます。C言語関数は **IEC 言語** や**フローチャート**からコールすることができます。

ST 言語でC言語関数をコールするには、関数のコールと同様の文法になります。入力パラメータは、関数名の後ろに括弧内に書きます。入力パラメータが複数ある場合はカンマで区切ります。そして、その式そのものが関数の戻り値となります。C言語関数は、代入文や式のどこに使っても構いません。以下は代入文にC言語関数を使った例です。

result := ProcName (par1, par2, ... parN);

FBD 言語でC言語関数をコールする場合も、標準関数と同様に使用できます。その入力パラメータはブロックの左側に接続され、出力パラメータは右側に接続されます。以下にブロックの標準的な形を示します。



C言語関数は、**SFC** のアクションブロックや、トランジションに付加される条件文からもコールできます。

C言語関数のインタフェースの定義

ISaGRAF ライブラリマネージャの「**編集**」-「**パラメータ**」メニューで新しいC言語関数の入力パラメータと出力パラメータを定義します。関数は最大 **31** の入力パラメータと常に **1** つの出力パラメータを持ちます。下のダイアログボックスはC言語関数のパラメータを記述するためのものです。



ダイアログボックスの上部のリストはC言語ファンクションのパラメータを示しています。ここで関数のパラメータの順番は、一番上から入力パラメータがあり、一番下が出力パラメータにならなければなりません。ダイアログボックスの下部は選択中のパラメータの詳細を記述するために使用します。

- パラメータの名前
- パラメータの方向（入力／出力）
- パラメータのデータ型

全ての ISaGRAF データ型がパラメータとして使用できます（ブール型値、整数型値、実数型値、タイマ型値、可変長文字列型値）。整数と実数の型は正確に区別して設定しなければなりません。下は ISaGRAF のデータ型とC言語の型の対応表です。

ブール型	unsigned long	符号なし 32 ビットワード。1=true / 0=false
整数型	long	符号付き 32 ビットワード。
実数型	float	符号付き単精度浮動小数点値。
タイマ型	unsigned long	符号なし 32 ビットワード。1単位が1ミリ秒。
可変長文字型	char *	文字列。

可変長文字列型値をC言語ファンクションに渡す場合、NULL キャラクタを含めることはできません。文字列は NULL キャラクタでターミネートされています。ただし、ターゲットのバージョンによっては文字列長の情報も渡されるのでこの限りではありません。最新の情報は下記の URL をご覧下さい。

<http://www.co-nss.co.jp>

出力パラメータはリスト中の最後である必要があります。パラメータの名前は以下に示すルールに従わなければなりません。

- 最大16文字
- 最初の文字は英字(a~z)でなければなりません
- 以降の文字は、英字、数字、アンダースコア(_)でなければなりません
- 大文字小文字の区別はありません

関数のパラメータ名は重複してはいけません。入力パラメータと出力パラメータに同じ名前を使用できません。別のファンクション内なら同名のパラメータが使用できます。デフォルトの出力パラメータの名前は"Q"です。この名前は自由に変更できます。パラメータ名はC言語ソースコード内での変数名に対応しています。

"挿入"ボタンは選択しているパラメータの前に新しいパラメータを挿入するのに使用します。"削除"ボタンは選択しているパラメータを削除するときに使用します。"アレンジ"ボタンは、出力パラメータがリストの一番最後になるように自動的にパラメータの並べ替えを行います。"OK"ボタンを押すとファンクションに関するインターフェース定義を保存し、ダイアログボックスを閉じます。"キャンセル"ボタンは、インターフェース定義を変更せずに、ダイアログボックスを閉じます。

☐ C言語ファンクションのCインターフェース

ファンクションのインターフェースはそのパラメータによって変わります。入力パラメータと出力パラメータは構造体として渡されます。この構造体は"GRUS0nnn.H"というファイルで定義されます。"nnn"は ISaGRAF のライブラリマネージャが管理する関数の番号です。以下に三角関数の"SIN"を計算するC言語ファンクションの例を示します。

```
/* File: GRUS0255.h - function "sample" */

typedef long          T_BOO;
typedef long          T_ANA;
typedef float         T_REAL;
typedef long          T_TMR;
typedef char          *T_MSG;

typedef struct {
    /* CALL */          T_REAL _param1;
    /* RETURN */        T_REAL _param2;
} str_arg;

#define PARAM1         (arg->_param1)
#define PARAM2         (arg->_param2)

/* end of file */
```

ISaGRAF のデータ型とC言語の型の対応を下表に示します。ISaGRAF のデータ型は、C言語の型としてファンクションのヘッダファイルで定義されています。

ブール型	T_BOO	long (32ビット)
整数型	T_ANA	long
実数型	T_REAL	float (32ビット、単精度)
タイマ型	T_TMR	long
可変長文字列型	T_MSG	char * (32ビットのポインタ)

"**str_arg**"構造体のそれぞれのメンバが、ファンクションの各パラメータに対応します。戻り値は構造体の最後のメンバです。引数は、関数パラメータ定義で設定した順序でリストにできます。パラメータ名を大文字にしたものは、ファンクションのCソースに渡される構造体内のパラメータに直接アクセスするためのマクロです。この名前は、ISaGRAF ライブラリマネージャを使用して定義したものになっています。

C言語のヘッダファイルは、ライブラリマネージャでファンクションのインターフェースを変更する度に更新されます。このことにより関数の実装と ISaGRAF のプログラミングとの間の完全な整合性が保証されます。

ソースコード

以下に、C言語ファンクションソースコードの標準的なテンプレートを示します。

```
/* Example of user function - Number is "255" - Name is "SAMPLE" */

#include "tasy0def.h"          /* ISaGRAF kernel common definitions */
#include "grus0255.h"          /* interface definition for function 255 */

void USP_sample (str_arg *arg)
{
    /* body of the function */
}

/* 以下の関数はC言語ファンクションの初期化とその宣言のために使われます。
この関数名でカーネルとリンクします。この関数は ISaGRAF ライブラリマネージャが自動生成します。*/

UFP uspdf_sample (char *name)
{
    strcpy (name, "SAMPLE");    /* ファンクション名を与える*/
    return (USP_sample);       /* ファンクションのアドレスを返す */
}

/* end of file */
```

"**TASY0DEF.H**"インクルードファイルはシステムに依存する定義のために必要です。また、void 型の関数ポインタを示す **UFP** 型の定義も含んでいます、これは関数の宣言のために使用されるものです。

プロジェクトとC言語ファンクションとの関連付け

C言語ファンクションの実装と ISaGRAF プロジェクト内で使うC言語ファンクションとの関連付けは、C言語ファンクションの名前によって行われます。「宣言」用の関数部分がC言語ファンクションのCソースコードに付加されています。この宣言関数部はアプリケーションがスタートした時に1回だけコールされ、ISaGRAF カーネルにC言語ファンクションの名前を伝えます。以下に標準的な宣言関数のフォーマットを示します。

```
UFP uspdf_xxx (char *name)
```

```
{
    strcpy(name, "XXX");          /* ファンクション名を与える*/
    return (USP_xxx);             /* ファンクションのアドレスを返す*/
}
/* (xxx はファンクション名) */
```

strcpy 文の中で使用される関数の名前は、**大文字**でなければいけません。そして関数の実装のための宣言関数中の名前は小文字でなければなりません。**"CSP_"**や**"uspdef_"**の接頭語を使用することによって、既存のC言語の予約語や ISaGRAF のライブラリの名前に干渉することなく、ユーザが関数に名前を付けることができます。

C言語ファンクションで特別な初期化を行うために、他のステートメントを宣言関数部に追加することもできます。ISaGRAF システムでは、この宣言関数部はアプリケーションがスタートする時**1回だけ**コールされることに注意して下さい。

宣言関数部は、たとえ ISaGRAF アプリケーション中で使用されていなくてもコールされます。ISaGRAF のアプリケーションで使用されているファンクションがカーネルに組み込まれていない場合は、致命的なエラーを起こしますので注意して下さい。

新しいファンクションをカーネルとリンクする前に、**"GRUS0LIB.C"**という名前のソースコードを編集し、リストに新しいファンクションを追加しなければなりません。**"GRUS0LIB.C"**にはただ宣言関数部の定義の配列があるだけです。この配列は、C言語で記述されたファンクションと動的にリンクするために、アプリケーションが初期化される時に読まれます。以下にそのファイルの例を示します。

```
/* File "GRUS0LIB.c" - Example using trigonometric functions */

#include <tasy0def.h>                      /* required for types definition */

extern UFP uspdef_fc1 (char *name);       /* declaration functions */
extern UFP uspdef_fc2 (char *name);
extern UFP uspdef_fc3 (char *name);
extern UFP uspdef_fc4 (char *name);

UFP_LIST USPDEF[ ] = {                   /* array of declaration functions */
                                     /* for integrated functions */
    uspdef_fc1,
    uspdef_fc2,
    uspdef_fc3,
    uspdef_fc4,

    NULL };

/* end of file */
```

USPDEF の配列は NULL ポインタで終わらなければなりません。さもないと実行時にクラッシュする可能性があります。**USPDEF** の配列が未定義だと、新しい ISaGRAF カーネルをリンク生成する時に、「未解決の参照」といったエラーが起きます。

このファイルにファンクション名を追加することにより、新しいカーネルは、既存のファンクションに追加される形で作られることになります。また、**USPDEF** 配列にプロジェクトで使用するファンクションのみ記述し、そのプロジェクトに特化したカーネルにカスタマイズすることも可能です。アプリケーションコードの生成時に、自動的に"**GRUSOLIB.C**"ファイルが生成されます。このファイルは、ISaGRAF のプロジェクトディレクトリに作られます。そしてプロジェクトで使用しているファンクションのみをグループ化しています。

制限

ISaGRAF ライブラリは最大 **255** 個のC言語ファンクションを持つことができます。どんなタイプの処理でもC言語ファンクションの中で実行できます。しかし、C言語ファンクションは ISaGRAF のサイクルに**同期して**呼び出されるので、C言語ファンクションの実行時間は直接サイクルタイムに影響を与えるという点に注意してください。

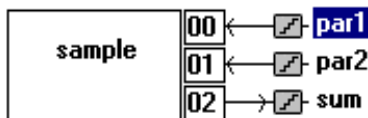
C言語ファンクションの例

以下に加算を実行する"**sample**"ファンクションを例にプログラミングの流れを紹介します。

以下のリストはファンクションの技術メモです。

名前:	SAMPLE
記述:	実数(アナログ変数)の加算
作成日:	1995 12 24
作者:	ICS Triplex ISaGRAF Inc.
入力パラメータ:	par1, par2: 実数アナログ
出力パラメータ:	sum 実数アナログ
プロトタイプ:	sum := sample (par1, par2);

下図は関数のインターフェースです。



次は、C言語ソースコードのヘッダーファイルです。

```
/* File: GRUS0255.h - C言語ファンクション定義 - Name: sample */
```

```
/* 標準 ISaGRAF データタイプの定義 */
```

```
typedef long T_BOO;
typedef long T_ANA;
typedef float T_REAL;
```

```
typedef long T_TMR;
typedef char *T_MSG;

/* 入力・出力パラメータ構造体の定義 */

typedef struct {
    T_ANA_par1; /* 入力パラメータ#1 */
    T_ANA_par2; /* 入力パラメータ#2 */
    T_ANA_sum; /* 出力パラメータ */
} str_arg;

/* 入力・出力パラメータをアクセスするための識別子 */

#define PAR1 (arg->_par1)
#define PAR2 (arg->_par2)
#define SUM (arg->_sum)

/* end of file */
```

以下のリストはC言語のソースコードです。太字の行だけがCプログラマが記述した部分です。

```
/* File: GRUS0255.c - C言語ファンクション - Name: SAMPLE */

#include "tasy0def.h" /* required for types definition */
#include "grus0255.h" /* C function source header */

/* Cメインルーチン: 加算の計算部分 */

void USP_sample (str_arg *arg)
{
    SUM = PAR1 + PAR2;
}

/* ISaGRAF カーネルとのダイナミックリンクの定義 */

UFP uspdf_sample (char *name)
{
    strcpy (name, "SAMPLE");
    return (USP_sample);
}

/* end of file */
```

C.7.4 C言語ファンクションブロック

C言語ファンクションブロックは、スタティックデータとその処理を組み合わせたもので、スタティックデータの処理ができるようにした複数のC言語の関数からなります。通常は、C言語ファンクションブロックは標準の IEC61131-3 言語の拡張のために使用します。データの加工のみに使われるC言語ファンクションとは異なり、ファンクションブロックはスタティックデータを処理することができます。従って、ファンクションブロックのアルゴリズムは時系列データの管理も可能になります。

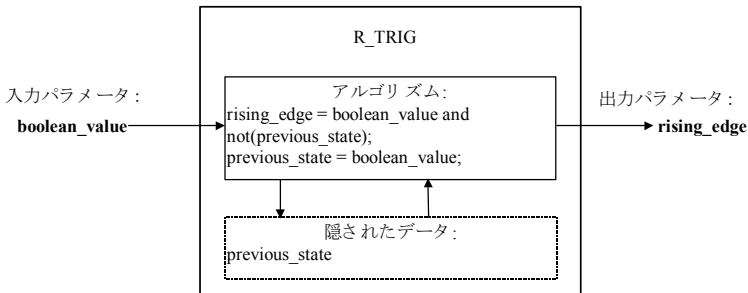
C言語で記述したファンクションブロックはコンパイルし、ISaGRAF カーネルとリンクします。新たに機能拡張されたカーネルをターゲット上にインストールしてから、拡張機能をプロジェクトで使用するようにします。

新しいファンクションブロックの機能はワークベンチのシミュレータには実装されません。ISaGRAF アプリケーションのシミュレーションは、非標準のファンクションブロックを使用する前に行う必要があります。

注意: ファンクションブロックは、アプリケーションのサイクルに同期して ISaGRAF カーネルによって実行されます。ファンクションブロックの処理時間や呼び出しのサービスは、ISaGRAF のアプリケーションサイクルタイムに直接影響を与えます。サイクルタイムを必要以上に延ばさないためには、ユーザはファンクションブロックの中でウェイトのオペレーションを使用しないことが必要です。

⇒ ファンクションブロックインスタンスの宣言

ファンクションブロックは、スタティックデータと処理とを組み合わせたオブジェクトです。以下の "R_TRIG"ファンクションブロックは、ブール型値変数の立ち上がりエッジを検出するファンクションブロックの例です。



上図で、隠されたスタティック変数"previous_state"はエッジを検出する為に必要なデータです。この変数はアプリケーション中で使用される複数の" R_TRIG"ファンクションブロックでそれぞれ違うデータでなければなりません。ST 言語で使うファンクションブロックのインスタンスは、辞書で宣言しなければなりません。なぜならば、ファンクションブロックは内部に隠されたデータを持っていて、ファンクションブロックの各コピー（インスタンス）は一つ一つ別の名前でなければならないからです。ファンクションブロックそのものの名前はライブラリマネージャで定義し、インスタンスの名前は、辞書エディタで設定します。

FBD 言語で使用するファンクションブロックは辞書宣言の必要はありません。なぜなら、ISaGRAF の FBD エディタは自動的に使用したファンクションブロックのインスタンスを宣言してくれるからです。FBD エディタによって自動的に宣言されたファンクションブロックのインスタンスは常に編集集中のプログラムにローカルなものとなります。

☐ C言語ファンクションブロックの ISaGRAF ライブラリへの追加

ISaGRAF ライブラリマネージャを使って、ワークベンチ上の ISaGRAF ライブラリに新しいC言語ファンクションブロックを追加します。C言語ファンクションブロックライブラリを選択しておき、「ファイル」→「新規作成」メニューを選びます。新規作成の後、**技術メモ**を記述して下さい。新しいファンクションブロックのC言語のテンプレートが自動的に作成されます。

メニューの「編集」→「パラメータ」を選択し、ファンクションブロックの入力パラメータと出力パラメータを定義します。

☐ ISaGRAF プロジェクトでC言語ファンクションブロックの使い方

組み込まれたC言語ファンクションブロックは ISaGRAF プロジェクトのプログラム中で標準ファンクションブロックと同様に使用することができます。C言語ファンクションブロックは **IEC 言語**や**フローチャート**からコールすることができます。

ST 言語中でC言語ファンクションブロックをコールするには、ファンクションブロックのコールと同様の文法になります。ファンクションブロックの入力パラメータ(引数)は、インスタンス名に続けて括弧内に書きます。複数ある場合はカンマで区切ります。出力パラメータ(戻り値)にアクセスするには、インスタンス名と出力パラメータ名から組み合わせられて作られるコンポーネント名を使って1個ずつ行います。名前はインスタンス名と出力パラメータ名をドット(.)で区切って表わします。例えば、

FBINSTNAME.parname

は、"**FBINSTNAME**"という名前のファンクションブロックのインスタンスの "**parname**"という名前の出力パラメータを表現します。

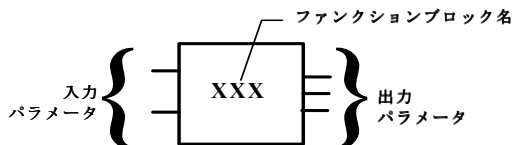
ST 言語中で使用するファンクションブロックのインスタンスは、辞書で宣言する必要があります。ファンクションブロックのそれぞれのコピー(インスタンス)は重複しない名前で識別されなければなりません。以下に ISaGRAF 辞書でのファンクションブロックインスタンス宣言の例を示します。

インスタンス:	TRIG1	タイプ:	R_TRIG
	TRIG2		R_TRIG

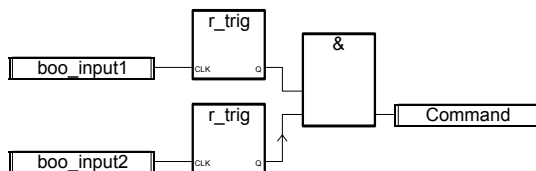
次に、これらの宣言されたインスタンスの ST 言語での使用例を示します。

```
TRIG1 (boo_input1);
TRIG2 (boo_input2);
Command := (TRIG1.Q & TRIG2.Q);
```

FBD プログラムはどんなC言語ファンクションブロックでもコールできます。ファンクションブロックは、標準のブロックと同様に扱えます。その入力パラメータは、左側に接続され、出力パラメータは右側に接続されます。標準的なブロックを次に示します。



FBD 言語で使用するファンクションブロックは宣言の必要はありません。ISaGRAF の FBD エディタが自動的に使用されたファンクションブロックのインスタンスを宣言しているからです。自動的に宣言されたファンクションブロックのインスタンスは、常に編集集中のプログラムに**ローカル**となります。以下に、前の ST 言語の例を FBD 言語でプログラムした例を示します。



☞ C言語FBのインタフェースの定義

ISaGRAF ライブラリマネージャのメニューの「**編集**」-「**パラメータ**」コマンドで新しいC言語ファンクションブロックの入力パラメータと出力パラメータを定義します。ファンクションブロックは最大 **32** のパラメータ(入力パラメータと出力パラメータの合計)を持てます。C言語ファンクションと違って、ファンクションブロックは複数の出力パラメータを持つことができます。下のダイアログボックスはC言語ファンクションブロックのパラメータを記述するためのものです。



ダイアログボックスの上部のリストはC言語ファンクションブロックのパラメータを示しています。ここでパラメータの順番はファンクションブロックのコール時のプロ

トタイプの順番を基本にして、一番上に入力パラメータがあり、その次に出力パラメータとなります。ダイアログボックスの下部は選択中のパラメータの詳細を記述するために使用します。

- パラメータの名前
- パラメータの方向(入力パラメータ／出力パラメータの選択)
- パラメータの変数型

全ての ISaGRAF データ型がパラメータとして使用できます(ブール型、整数型、実数型、タイマ型、可変長文字列型)。整数と実数の型は区別する必要があります。下表に ISaGRAF のデータ型とC言語の型の対応を示します。

ブール型	unsigned long	符号なし 32 ビットワード: 1=true / 0=false
整数型	long	符号付き 32ビット整数
実数型	float	単精度浮動小数点値
タイマ型	unsigned long	符号なし 32ビットワード (単位は 1 ms)
可変長文字列型	char *	文字列

可変長文字列型の値をC言語ファンクションブロックに渡す場合は、文字列に NULL キャラクタを含めることはできません。なぜなら、Cソースコードに渡される文字列は NULL キャラクタでターミネートされているからです。ただし、ターゲットのバージョンによっては文字列長の情報も渡されるのでこの限りではありません。最新の情報は下記の URL をご覧下さい。

<http://www.co-nss.co.jp>

出力パラメータはリスト中の最後である必要があります。パラメータの名前は以下に示すルールに従わなければなりません。

- 最大16文字
- 最初の文字は英字(a～z)でなければなりません
- 以降の文字は、英字、数字、アンダースコア(_)でなければなりません
- 大文字、小文字の区別はありません

ファンクションブロックのパラメータ名は重複してはいけません。入力パラメータに出力パラメータと同じ名前を使用できません。同じ名前のパラメータは別のファンクションブロック内であれば使用できます。パラメータ名はC言語ソースコード内での変数名に対応しています。

"挿入"ボタンは選択しているパラメータの前に新しいパラメータを挿入するのに使用します。"削除"ボタンは選択しているパラメータを削除するために使用します。"アレンジ"ボタンは、出力パラメータがリストの一番最後になるように自動的にパラメータの並べ替えを行います。"OK"ボタンを押すとファンクションブロックのインターフェース定義を保存し、ダイアログボックスを閉じます。"キャンセル"ボタンは、ファンクションブロックのインターフェース定義を変更せずに、ダイアログボックスを閉じます。

☐ C言語ファンクションブロックのCインタフェース

ファンクションブロックのインターフェースはそのパラメータ定義に依存します。引数は構造体として渡されます。この構造体は"GRFB0nnn.H"というファイルで定義されます。"nnn"は ISaGRAF のライブラリマネージャが管理するファンクションブロックの番号です。戻り値は、同じく"GRFB0nnn.H"というファイルで定義された番号によって表現されます。

以下に、" LIM_ALARM "ファンクションブロックのC言語インターフェースの例を示します。

```
/* function block interface - name: sample */

/* 標準 ISaGRAF のデータタイプ */

typedef      long           T_BOO;
typedef      long           T_ANA;
typedef      float          T_REAL;
typedef      long           T_TMR;
typedef      char           *T_MSG;

/* 入力パラメータの構造体 */

typedef struct {
    /* CALL */      T_BOO   _par1;
    /* CALL */      T_BOO   _par2;
} str_arg;

/* str_arg 構造体へのアクセス */

#define      PAR1          (arg->_par1)
#define      PAR2          (arg->_par2)

/* 出力パラメータの論理番号 */

#define      FBLPNO_Q1     0
#define      FBLPNO_Q2     1

/* end of file */
```

ISaGRAF のデータ型とC言語の型の対応を下表に示します。ISaGRAF のデータ型は、C言語の型として定義ファイルで定義されています。

ブール型	T_BOO	long (32 bits)
整数型	T_ANA	long
実数型	T_REAL	float (32 bits - 単精度)
タイマ型	T_TMR	long
可変長文字列型	T_MSG	char * (32 bits – char のポインタ)

"str_arg"構造体のそれぞれのメンバは、ファンクションブロックのパラメータそれぞれに対応します。戻り値は構造体の最後のメンバです。引数の順序は、ファン

クシオンブロックのパラメータ定義で設定したものと同じです。大文字の識別子は、ファンクションブロック実行サービス部分で直接パラメータにアクセスするためのマクロです。この名前は、ISaGRAF ライブラリマネージャでファンクションブロックを定義したときのものになっています。

出力パラメータ(戻り値)に適用された番号の順序は、ファンクションブロックのパラメータ定義で設定したものと同じです。1番目の出力パラメータの論理番号は常に0です。

C言語ソースのプログラム時には、出力パラメータの記述には論理番号ではなく、定義された識別子を使用するようにします。このことにより、インターフェースの定義を変更しても、ソースファイルの再コンパイルが容易に行なえるようになります。

C言語の定義ファイルは、ISaGRAF のライブラリマネージャでファンクションブロックのインターフェースを変更するたびに更新されます。これはファンクションブロックの実装とISaGRAF のプログラミングとの間の完全な整合性を保証します。

■ ソースコード

ファンクションブロックのC言語での実装は3つのエントリーポイントに分かれます。

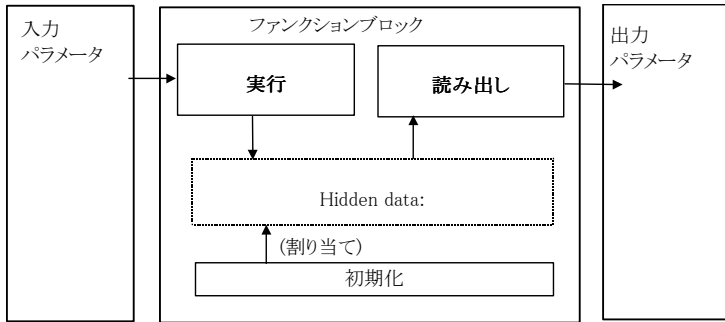
- 初期化サービス
- 実行サービス — 入力パラメータに対する処理
- 出力パラメータの読み出しサービス

同じファンクションブロックの各々のインスタンスには同一のプログラムコードが使用されます。このコードはコピーされて使われるわけではありません。スタティックデータは各々のインスタンスに割り当てられます。このスタティックデータはISaGRAF のプログラミングによって直接アクセスできません。これはファンクションブロックのインスタンスの“ローカル変数、即ち隠れた変数(hidden variable)”として確保されています。

“実行サービス”は、ターゲットサイクル毎にインスタンスが使用されるたびにコールされます。そして、入力パラメータを処理し割り当てられたデータを更新します。このサービスはファンクションブロックのメイン処理になります。

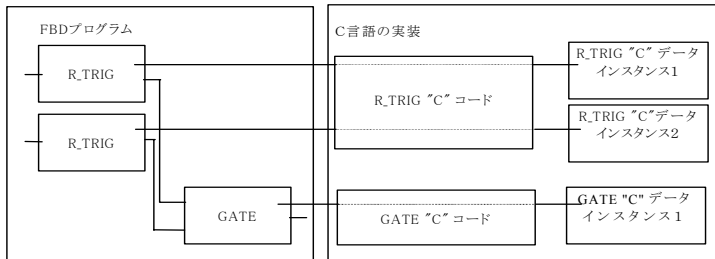
出力パラメータの“読み出しサービス”は、ISaGRAF カーネルによって、あるインスタンスの戻り値の現在値を読むために呼ばれます。このサービスでは特別な計算をする必要はありません。ただ、隠されたデータを ISaGRAF のアプリケーションに転送するだけの処理を行ないます。

ファンクションブロックダイアグラム:



● ファンクションブロックのスタティックデータ

ファンクションブロックは処理とスタティックデータをまとめたものです。データの構造体がファンクションブロックの各々のインスタンスに割り振られます。ST 言語や FBD の中で使用されるたびに、1つのインスタンスとデータ構造体が割り当てられます。次の例では、C言語のデータ構造と FBD で使用されたファンクションブロックのインスタンスとの関連を示します。



各々のインスタンスのデータ構造に必要なメモリは ISaGRAF システムがアプリケーションのスタート時に確保します。インスタンスのデータ構造体を指すポイントは「実行」や「読み出し」のサービス時に渡されます。

ISaGRAF ライブラリマネージャは自動的にデータの構造体の定義に必要なC言語ソースコードのテンプレートを生成します。データの構造体は常に「str_data」という名前になります。プログラマは、インターフェースの互換性を保証するためにも、この名前を変えない方がよいでしょう。隠されたデータ(hidden data)は一般に戻り値のイメージと一緒に内部変数としてグルーブ化されています。ファンクションブロックの「読み出し」のサービスは、戻り値にアクセスするためだけに使用されます。そして、他の処理を行うために使用はできません。

● 初期化サービス

ファンクションブロックの“初期化”サービスは、ISaGRAF カーネルによってアプリケーションがスタートした時に1度コールされます。C言語プログラマはこの時システムに対して1つのインスタンスに必要なメモリのサイズを指定できます。以下に標準的な“初期化”サービスを示します:

```
uint16 FBINIT_xxx (uint16 hinstance)
/* "xxx" はファンクションブロック名 */
{
    return (sizeof (str_data));
}
```

“hinstance”引数はインスタンスの番号です。これは ISaGRAF の内部処理で使われ、プログラミングには使用できません。初期化サービスは、**1つの**インスタンスに必要なデータのバイト数を返します。要求するメモリの量は **64K** バイトを越えてはいけません。初期化サービス内では他の処理を行うことはできません。このサービスのCソースコードは、ファンクションブロックが作られた時に、自動的に ISaGRAF ライブラリマネージャが作成します。

● 実行サービス

“実行”サービスは、アプリケーションで使用されるファンクションブロックの各々のインスタンスそれぞれに対し、毎回ターゲットサイクルの中でコールされます。このサービスは、入力パラメータのデータの処理を行い、隠れたスタティックデータ (hidden data) や出力パラメータを更新するために、ファンクションブロックのメイン処理を実行します。以下に標準的な“実行”サービスの内容を示します:

```
void FBACT_xxx (
uint16 hinstance,          /* "xxx" はファンクションブロック名 */
                           /* インスタンスの論理番号 */
str_data *data,           /* data: インスタンスのデータ構造体
                           へのポインタ */
str_arg *arg               /* 入力パラメータ構造体へのポインタ */
)
{
}
```

“hinstance”引数はインスタンスの番号です。これは ISaGRAF の内部処理で使われ、プログラミングには使用できません。“data”引数はインスタンスの構造体の far 型ポインタです。“arg”引数は、入力パラメータの値を持った構造体の far 型ポインタです。プログラマは、“arg”構造体のメンバにアクセスするためにはヘッダファイルで定義されている識別子を使用するようにします。

“実行”サービスのアルゴリズムは入力パラメータ (“arg”構造体のメンバ) の処理と、“data”構造体のデータの更新を行うことです。以下に、**R_TRIG** ファンクションブロック (立ち上がりエッジ検出) の“実行”サービスを示します。

/* ファンクションブロックヘッダファイルに登録されている定義 */

```
typedef struct {
    T_BOO _clk;          /* 入力パラメータ */
                           /* トリガー入力 */
}
```

```

} str_arg;

#define CLK    (arg->_clk)

/* ファンクションブロックインスタンスのデータ構造体 */

typedef struct {
    T_BOO prev_state;           /* トリガ入力の前回の状態 */
    T_BOO edge_detect;         /* 立ち上がり検出信号: 出力パ
ラメータの写し */
} str_data;

/* 実行サービス */

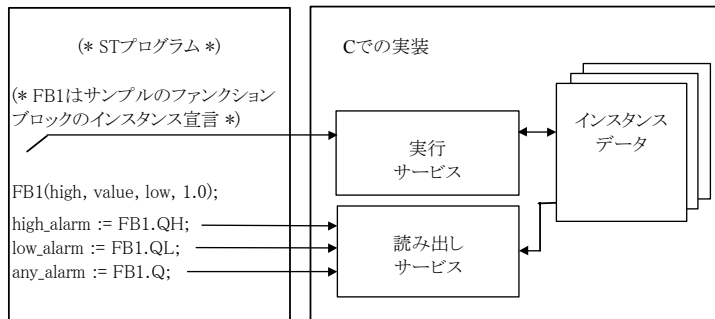
void FBACT_trig (uint16 hinstance, str_data *data, str_arg *arg)
{
    data->edge_detect = (T_BOO)(CLK && !data->prev_state);
    data->prev_state = CLK; /* calling parameter */
}

```

C言語ソースコードのテンプレートは、ファンクションブロックが作られた時に、自動的に ISaGRAF ライブラリマネージャが作成します。

● 出力パラメータの読み出しサービス

“読み出し”サービスは、ST 言語や FBD でファンクションブロックの戻り値が参照されるたびにコールされます。このサービスは出力パラメータの1つだけをを得るために使用されます。下記の例では“読み出し”サービスが ST プログラムの実行中に実行されている様子を表現しています。



同じ出力パラメータに対する“読み出し”サービスは、同一のサイクルで2回以上コールされるかもしれないので、このサービス中に特別な処理は行うことはできません。隠れたデータを ISaGRAF のアプリケーションに送る処理だけを行ないます。以下に標準的な“読み出し”サービスのテンプレートを示します：

```

/* 出力パラメータ値をキャスト(型変換)してコピーする処理 */

#define BOO_VALUE    ((T_BOO *)value)
#define ANA_VALUE    ((T_ANA *)value)

```

```
#define REAL_VALUE ((T_REAL *)value)
#define TMR_VALUE ((T_TMR *)value)
#define MSG_VALUE ((T_MSG *)value)

/* 出力パラメータ読み出しサービス: 各出力パラメータ毎にコールされる */

void FBREAD_xxx ( /* "xxx" はファンクションブロック名 */
uint16 hinstance, /* インスタンスの論理番号 */
str_data *data, /* インスタンスのデータ構造体へのポインタ*/
uint16 parno, /* 出力パラメータの論理番号 */
void *value) /* パラメータ値をコピーするバッファ */
{
    switch (parno) {
        case FBLPNO_XX: /* ... */ break;
        case FBLPNO_YY: /* ... */ break;
        /* .... */
    }
}
```

"hinstance"引数はインスタンスの番号です。これは ISaGRAF の内部処理で使われ、プログラミングには使用できません。"data"引数はインスタンスの構造体の far 型ポインタです。

"parno"引数は、要求されている出力パラメータの論理番号です。出力パラメータを識別するには、C言語ヘッダーで定義されている識別子を使用して下さい。この様な識別子は "FBLPNO_"という文字が先頭につきます。"value"引数は読み出す出力パラメータの値をコピーするバッファを指す far ポインタです。この引数で示されるデータ型は出力パラメータの ISaGRAF でのデータ型に一致します。以下の表に、ISaGRAF のデータ型とC言語のデータ型の対応を示します。

ブール型	long	符号なし 32ビットワード: 1=true / 0=false
整数型	long	符号付き 32ビット整数
実数型	float	単精度浮動小数点値
タイマ型	long	符号なし 32ビットワード (単位は 1 ms)
可変長文字列型	char *	文字列

次のマクロは、読み出す出力パラメータのデータ型に対応したバッファにアクセスするために使います。

```
#define BOO_VALUE ((T_BOO *)value)
#define ANA_VALUE ((T_ANA *)value)
#define REAL_VALUE ((T_REAL *)value)
#define TMR_VALUE ((T_TMR *)value)
#define MSG_VALUE ((T_MSG *)value)
```

これらは、値やパラメータを ISaGRAF のバッファにコピーするために、共通的に使用される処理です。

```
/* ブール型パラメータに対して */
*BOO_VALUE = parameter_value;
/* 整数型パラメータに対して */
```

```

    *ANA_VALUE = parameter_value;
/* 実数型パラメータに対して */
    *REAL_VALUE = parameter_value;
/* タイマ型パラメータに対して */
    *TMR_VALUE = parameter_value;
/* 可変長型文字列型パラメータに対して */
    strcpy (*MSG_VALUE, parameter_value);

```

C言語ソースコードのテンプレートは、ファンクションブロックが作られた時に、自動的に ISaGRAF ライブラリマネージャが作成します。

● C言語ファンクションブロックのソースファイル例

以下に、C言語ファンクションブロックの標準的な構成を示します。

```

/* ファンクションブロック (xxx はファンクションブロック名) */

#include <tasy0def.h>
#include <grfb0nnn.h> /* nnn はライブラリ中でのファンクションブロックの番号 */

/* 各ファンクションブロックの隠されたデータ用の構造体 */
typedef struct {
    /* メンバーの定義 */
} str_data;

/* 初期化サービス: 隠されたデータに必要なサイズを戻す */
word FBINIT_xxx (uint16 hinstance)
{
    return (sizeof (str_data));
}

/* 実行サービス: 入力パラメータの処理 */
void FBACT_xxx (uint16 hinstance, str_data *data, str_arg *arg)
{
    /* ... */
}

/* 出力パラメータの値をキャストしてコピーするマクロ */
#define BOO_VALUE ((T_BOO *)value)
#define ANA_VALUE ((T_ANA *)value)
#define REAL_VALUE ((T_REAL *)value)
#define TMR_VALUE ((T_TMR *)value)
#define MSG_VALUE ((T_MSG *)value)

/* 出力パラメータ読み出しサービス: 各出力パラメータ毎に読み出し */
void FBREAD_xxx (uint16 hinstance, str_data *data, uint16 parno, void *value)
{
    switch(parno)
    {
        case FBLPNO_XX: *???_VALUE = ...; break;
        case FBLPNO_YY: *???_VALUE = ...; break;
        ....
    }
}

```

```
}
```

/* 以下の関数はファンクションブロックの初期化と関数のアドレス宣言に使われます。ファンクションブロックの名前を使って ISaGRAF カーネルとのリンクを行います。この部分はライブラリマネージャによって自動的に生成されます。*/

```
ABP fbldef_XXX (char *name, IBP *initproc, RBP *readproc)
{
    strcpy (name, "XXX");
    *initproc = (IBP)FBINIT_XXX;
    *readproc = (RBP)FBREAD_XXX;
    return ((ABP)FBACT_XXX);
}

/* end of file */
```

インクルードファイル"TASK0DEF.H"には、システムに依存する定義が必要です。それには実装されたサービスに対する far ポインタを表すデータ型の定義も含まれます。

プロジェクトとC言語ファンクションブロックとのリンク

C言語ファンクションブロックの実装側と ISaGRAF のプロジェクトとの関連付けは、その名前によって行われます。ある種の宣言サービスがC言語ファンクションブロックのソースコードに付加されます。この宣言サービスはアプリケーションがスタートした時一度だけ実行され、ISaGRAF カーネルにC言語ファンクションブロックの名前を伝えます。以下に標準的な宣言サービスのフォーマットを示します。

```
ABP fbldef_XXX (char *name, IBP *initproc, RBP *readproc)
{
    strcpy (name, "XXX");          /* ファンクションブロック名 */
    *initproc = (IBP)FBINIT_XXX;   /* 初期化サービス */
    *readproc = (RBP)FBREAD_XXX;   /* 読み出しサービス */
    return ((ABP)FBACT_XXX);       /* 実行サービス */
}

/* XXX はファンクションブロック名 */
```

strcpy 文の中で使用されるファンクションブロックの名前は、**大文字**でなければいけません。実装されている各サービスと宣言サービス名の中のものは小文字でなければなりません。

"FBACT_", "FBINIT_", "FBREAD_"、"fbldef_" の様な接頭語を使用することによって、既存のC言語の予約語や ISaGRAF のライブラリの名前に干渉することなく、ユーザが関数に名前を付けることができます。
宣言サービスには他の処理を追加しないようにしてください。

組み込まれたファンクションブロック内の宣言サービスは、たとえ ISaGRAF アプリケーション中で使用されていなくてもカーネルからコールされます。ISaGRAF のアプリケーションで使用しているファンクションブロックがカーネルに組み込まれていない場合は、致命的なエラーを起こしますので注意して下さい。

新しいファンクションブロックをカーネルとリンクする前に、"GRFB0LIB.C"という名前のソースコードを編集し、リストに新しいファンクションブロックを追加する必要があります。"GRFB0LIB.C"の内容は宣言サービスの配列があるだけです。この配列は、C言語ファンクションブロックを動的にリンクするために、アプリケーションが初期化される時に読みこまれます。以下にそのファイルの例を示します。

```
/* File: grfb0lib.c – ファンクションブロックの組み込み */

#include <tasy0def.h>

extern ABP fbldef_fb1(char *name, IBP *init, RBP *read);
extern ABP fbldef_fb2(char *name, IBP *init, RBP *read);

FBL_LIST FBLDEF[ ] = {
    fbldef_fb1,
    fbldef_fb2,

    NULL };

/* end of file */
```

FBLDEF の配列は NULL ポインタで終わらなければなりません。さもないと実行時にクラッシュする可能性があります。**FBLDEF** の配列が未定義だと、新しいISaGRAF カーネルをリンク生成する時に、「未解決の参照」といったエラーが起きます。

このファイルにファンクションブロック名を追加することにより、新しいカーネルは、既存のファンクションブロックに追加される形で作られることになります。また、**FBLDEF** 配列にプロジェクトで使用するファンクションブロックのみ記述し、そのプロジェクトに特化したカーネルにカスタマイズすることも可能です。アプリケーションコードの生成時に、自動的に"GRFB0LIB.C"ファイルが生成されます。このファイルは、ISaGRAF のプロジェクトディレクトリに作られます。そしてプロジェクトで使用しているファンクションブロックのみをグループ化しています。

制限

ISaGRAF ライブラリは最大 **255** 個のC言語ファンクションブロックを持つことができます。どんなタイプの処理でもファンクションブロックの中で実行できます。各々のタイプのファンクションブロックは1個のプロジェクト中に最大 **255** 個のインスタンスを持つことができます。

ファンクションブロックは ISaGRAF のサイクルに同期して呼び出されるので、C言語ファンクションの実行時間は直接サイクルタイムに影響を与えるという点に注意してください。

C言語ファンクションブロックの例

以下にアップカウンタを実行する"sample"ファンクションブロックを例にプログラミングの流れを紹介します。下記はファンクションブロックの技術メモです。

名前:	SAMPLE
-----	--------

記述: Up counter

作成日: 01 February 1994

作成者: **ICS Triplex** ISaGRAF Inc.

入力パラメータ:

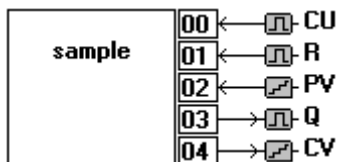
CU : counting input
R : reset command
PV : maximum programmed value

出力パラメータ:

Q : max detection
CV : counting result

プロトタイプ: SAMPLE (count, reset_command, maximum_value);
max_detect := SAMPLE.Q;
count_result := SAMPLE.CV;

次に、ファンクションブロックのインターフェースを示します。



次に、C言語のヘッダーを示します。

/* ファンクションブロックインタフェース – ファンクションブロック名: SAMPLE */

/* 標準 ISaGRAF データタイプ定義 */

```
typedef long T_BOO;
typedef long T_ANA;
typedef float T_REAL;
typedef long T_TMR;
typedef char *T_MSG;
```

/* 入力パラメータ構造体の定義 */

```
typedef struct {
    T_BOO _cu;
    T_BOO _r;
    T_ANA _pv;
} str_arg;
```

/* 入力パラメータのアクセス用の識別子 */

```
#define CU (arg->_cu)
#define R (arg->_r)
#define PV (arg->_pv)
```

```
/* 出力パラメータの論理番号 */
```

```
#define FBLPNO_Q          0
#define FBLPNO_CV        1
```

```
/* end of file */
```

以下のリストはC言語ファンクションブロックのソースコードです。**太字**の行だけがCプログラマによって記述された部分です。

```
/* ファンクションブロック - FB 名 SAMPLE */
```

```
#include <tasy0def.h>          /* データ型定義 */
#include <grfb0255.h>          /* ファンクションブロックのCソースヘッダ */
```

```
/* ファンクションブロックインスタンスのデータ用の構造体の定義*/
```

```
typedef struct {
    T_BOO overflow; /* TRUE: if 現在値 >= プログラム設定値 */
    T_ANA value;    /* カウント中の現在値 */
} str_data;
```

```
/* 初期化サービス: インスタンスデータ用のメモリ領域 */
```

```
word FBINIT_sample (uint16 hinstance)
{
    return (sizeof (str_data));
}
```

```
/* 実行サービス: カウントアップのアルゴリズム */
```

```
void FBACT_sample (uint16 hinstance, str_data *data, str_arg *arg)
{
    if (R) data->value = 0;
    else if (CU && data->value < PV) (data->value)++;
    data->overflow = (data->value >= PV) ? (T_BOO)1 : (T_BOO)0;
}
```

```
/* 出力パラメータをキャストして ISaGRAF バッファにコピーするマクロ*/
```

```
#define BOO_VALUE ((T_BOO *)value)
#define ANA_VALUE ((T_ANA *)value)
#define REAL_VALUE ((T_REAL *)value)
#define TMR_VALUE ((T_TMR *)value)
#define MSG_VALUE ((T_MSG *)value)
```

```
/* 読み出しサービス: 各出力パラメータ毎に読み出し */
```

```
void FBREAD_sample (uint16 hinstance, str_data *data, uint16 parno, void
*value)
{

```

```

switch (parno) {
    case FBLPNO_Q : *BOO_VALUE = data->overflow; break;
    case FBLPNO_CV : *ANA_VALUE = data->value; break;
}

}

/* ISaGRAF カーネルとのダイナミックリンクのための宣言サービス */

ABP fbldef_sample (char *name, IBP *initproc, RBP *readproc)
{
    strcpy (name, "SAMPLE");
    *initproc = (IBP)FBINIT_sample;
    *readproc = (RBP)FBREAD_sample;
    return ((ABP)FBACT_sample);
}

/* end of file */

```

C.7.5 コンパイルとリンクのテクニック

ISaGRAF ワークベンチび、ターゲットには各ターゲットに対応したCコンパイラやリンカは含まれていません。しかし、この章では、ISaGRAF ライブラリマネージャが生成するファイルを利用しやすくするように、かつコンパイラやリンカといったその他のツール上で、それらのファイルを使用するための主要なテクニックについて説明します。

☐ C言語ソースファイル

変換関数、ファンクション、ファンクションブロックのC言語のソースファイルはISaGRAF ライブラリマネージャが **ISAWIN\LIB\DEFS** と **ISAWIN\LIB\SRC** のディレクトリに作成します。このソースファイルの名前は ISaGRAF のライブラリに存在する変換関数、ファンクション、ファンクションブロックに対応する番号で生成されます。以下に使用するファイル名を示します。

\isawin\lib\defs\TACN0DEF.H	変換関数の定義ファイル
\isawin\lib\src\GRCN0nnn.H	変換関数のソースファイル
\isawin\lib\defs\GRUS0nnn.H	ファンクションの定義ファイル
\isawin\lib\src\GRUS0nnn.C	ファンクションのソースファイル
\isawin\lib\defs\GRFB0nnn.H	ファンクションブロックの定義ファイル
\isawin\lib\src\GRFB0nnn.C	ファンクションブロックのソースファイル

(nnn は変換関数、ファンクション、ファンクションブロックに対応する番号)

注意: ライブラリのエレメントの名前を変えたり、コピーした場合は、そのテキストやソースコードは ISaGRAF ライブラリマネージャによって更新されません。ソースコード中の名前や番号は手動で更新しなければなりません。

\ISAWIN\LIB\USPNUMS というファイルで、ISaGRAF ライブラリマネージャが管理するC言語ファンクションの番号と名前の関係がわかります。このファイルの例を以下に示します：

```

1  funct_A
10 funct_B
16 funct_C

```

`\ISAWIN\LIB\ FBLNUMS` というファイルで、ISaGRAF ライブラリマネージャが管理するC言語ファンクションブロックの番号と名前の関係がわかります。このファイルの例を以下に示します：

```

0  fbl_A
1  fbl_B
2  fbl_C

```

`\ISAWIN\LIB\ CNVNUMS` というファイルで、ISaGRAF ライブラリマネージャが管理する変換関数の番号と名前の関係がわかります。このファイルの例を以下に示します：

```

0  SCALE
1  BCD

```

これらのファイルは、変換関数、ファンクション、ファンクションブロックの作成、名前の変更、コピーや削除のたびに ISaGRAF ライブラリマネージャによって自動的に更新されます。ISaGRAF コードジェネレータは、アプリケーションコードが生成されるときに自動的に次のファイルを生成します。

<code>\isawin\apl\ppp\GRCN0LIB.C</code>	プロジェクト内で使用されている全ての 変換関数の配列宣言
<code>\isawin\apl\ppp\GRUS0LIB.C</code>	プロジェクト内で使用されている全ての ファンクションの配列宣言
<code>\isawin\apl\ppp\GRFB0LIB.C</code>	プロジェクト内で使用されている全ての ファンクションブロックの配列宣言。

(`ppp` は ISaGRAF プロジェクトの名前です)

これらのファイルは、あるプロジェクトで使用されている変換関数、ファンクション、ファンクションブロックだけの機能を持つ新しい ISaGRAF カーネルをビルドするために、リンク時に利用できます。

ターゲットシステムへのソースファイルのダウンロード

ターゲットシステムがコンパイラなどのツールをサポートしている場合は、ISaGRAF ライブラリマネージャによって作成されたC言語ソースファイル(*.C)と定義ファイル(*.H)をターゲットシステム側にダウンロードすることになります。Windows に付属する**ターミナル**等のソフトを利用できます。

ソースコードのメンテナンスをターゲットシステム側で行う場合は、ISaGRAF ライブラリマネージャでファンクションインタフェース部の変更を行うたびに、更新された定義ファイルをターゲット側にダウンロードしなおす必要があります。

ワークベンチのツールメニューをカスタム化してファイルのダウンロードなどの処理をバッチファイルとして登録しておくことができます。(詳細はワークベンチ プログラム管理を参照してください)

☐ クロスコンパイラの使用

ターゲットがIBM PC互換のパソコンの場合や、パソコンでターゲットシステム用のクロスコンパイラが利用できる場合、ソースファイルをパソコンで直接管理することが可能です。

この場合、ユーザは ISaGRAF ライブラリマネージャで変換関数、ファンクション、ファンクションブロックのソースファイルを書いたり、変更することができます。また、ワークベンチのツールメニューをカスタマイズして、コンパイラやリンクのためのコマンドをまとめてバッチファイルとして登録しておくこともできます。

変換関数、ファンクション、ファンクションブロックをパソコン上でコンパイル、リンクした場合、ユーザはこの新しくできた ISaGRAF カーネル(新しいファンクション類が組み込まれたもの)を、アプリケーションを実行する前に、ターゲットシステム上にダウンロードする必要があります。もしターゲットシステムが他のパソコンの場合、フロッピーディスクやネットワークを利用してターゲットシステム上へダウンロードします。

☐ ISaGRAF カーネルライブラリとのリンク

注意:

以下の内容は一般的な情報であり、使用するターゲットシステムによって異なることもあります。
各ターゲットについての詳細はターゲットディスク上のヘルプまたは README ファイルを参照願います。

ISaGRAF ターゲットソフトには、変換関数、C言語ファンクション、ファンクションブロックをコンパイル、リンクする為のたくさんのユーティリティが入っています。

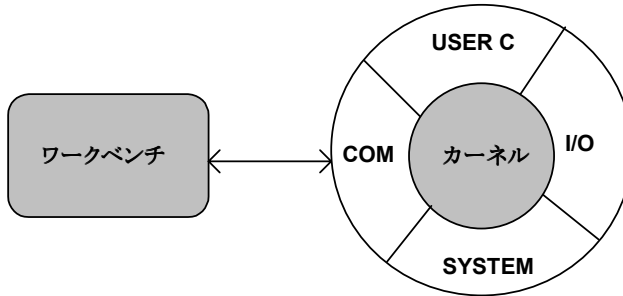
ISaGRAF ターゲットには以下の2つの実装の方法があります。

- シングルタスク ISaGRAF: 全ての機能が1つのプログラムで実行される
- マルチタスク ISaGRAF: 通信タスク(スレッド)が別に実行される

いずれの場合でも、C言語の部品は同じライブラリにグループ化されているので、シングルタスクでもマルチタスクでもCのプログラマにとっては何も違いはありません。ユーザのC言語ファンクションのライブラリはリンクされた結果、シングルタスクバージョンでは通常 **isa** と言うタスクになり、マルチタスクバージョンでは通常 **isaker** と言うタスクになります。

開発システム

ターゲットシステム

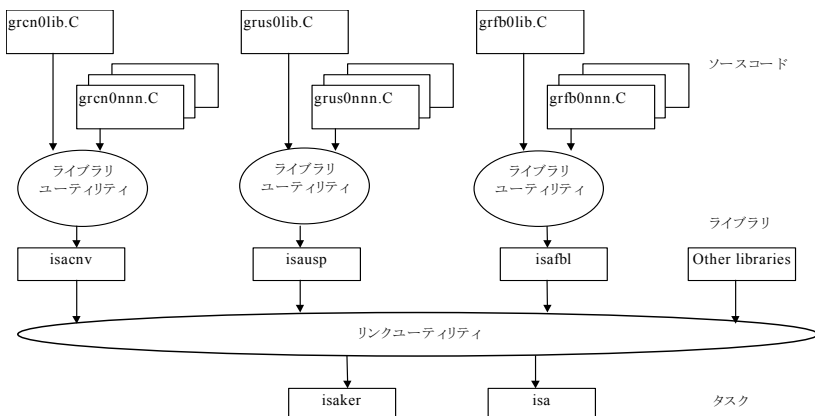


ISaGRAF ターゲットシステムの内部のカーネル部分はハードウェアに依存しないものとなっています。この部分では IEC の言語プログラムを実行し、その変数データベースも持っています。

カーネルとリンクを行うときの第一ステップは、そのプロジェクトに必要な全ての変換関数、C言語ファンクション、ファンクションブロックのライブラリを作成する事です。

ライブラリ	内容
ISAUSP	- GRUS0LIB のオブジェクトファイル (ファンクション宣言の配列) - 組み込む全てのファンクションのオブジェクトファイル
ISAFBL	- GRFB0LIB のオブジェクトファイル (ファンクションブロック宣言の配列) - 組み込む全てのファンクションブロックのオブジェクトファイル
ISACNV	- GRCN0LIB.obj(変換関数宣言の配列) - 組み込む全ての変換関数のオブジェクトファイル

次に、プログラマはこれらの新しいライブラリを、他のこれまで存在していた ISaGRAF カーネルのオブジェクトファイルやライブラリと一緒にリンクしなければなりません。コンパイル、リンクの様子を下図に示します。



以下に、リンク時に必要なオブジェクトとライブラリのリストを示します。

isaker をビルドするためのモジュール:

Object Module:	tast0mai	
Object Module:	tats0com	
Kernel library:	isaker	
Kernel library:	isaoem	
User library:	isausp	ユーザ定義ファンクション
User library:	isafbl	ユーザ定義ファンクションブロック
User library:	isacnv	ユーザ定義変換関数
Kernel library:	isasys	
System libraries:	(C コンパイラのマニュアルを参照してください)	

isa をビルドのためのモジュール:

Object Module:	tast0mai	
Object Module:	tast0com	
Kernel library:	isaker	
Kernel library:	isatst	
Kernel library:	isaoem	
User library:	isausp	ユーザ定義ファンクション
User library:	isafbl	ユーザ定義ファンクションブロック
User library:	isacnv	ユーザ定義変換関数
Kernel library:	isasys	
System libraries:	(C コンパイラのマニュアルを参照してください)	

プログラマはオブジェクトとライブラリの順番を、上記の図で示したような順番にしなければなりません。オブジェクトやライブラリはターゲットシステムにもよりますが、標準的な拡張子(".lib", ".obj", ".l", ".r"...)を持ちます。

≡ コンパイル、リンク時に必要なオプション

コンパイル、リンク時に便利なオプションを選択する事も可能です。オプションは変換関数、ファンクション、ファンクションブロックの処理内容によって異なりますが、ある処理ではその他のシステムライブラリ(数学演算、グラフィック等)もリンク時に必要になります。

ISaGRAF カーネルの全てのC言語ソースファイルはラージメモリモデルでコンパイルされています。プログラマは同じメモリモデルで変換関数、ファンクション、ファンクションブロックもコンパイルしなければなりません。

特別な定数がコンパイルのために定義されています。これらはターゲットシステムとCPUタイプとを示して、変換関数、ファンクション、ファンクションブロックのソースコードをシステムに依存しないものにするため使われます。以下に定数の例を示します。

DOS	DOS ベースのシステム(INTEL CPU)
ISAWNT	Windows-NT ベースのシステム(INTEL CPU)
OS9	OS9 システム (MOTOROLA CPU)
VxWorks	VxWorks システム (MOTOROLA CPU)

ISaGRAF ターゲットソフトと一緒に供給されるコマンドファイル (**ISACC.BAT** など)には、そのコンパイルコマンドの中でこの便利な定数がどのように設定されているかが示されています。

サポートされているコンパイラ

変換関数、ファンクション、ファンクションブロックを既存の ISaGRAF カーネルにリンクするために、以下のコンパイラをサポートしています。

Microsoft MSC 7.00 compiler	DOS ターゲット
Microsoft MSVC 4.2 compiler	Windows-NT ターゲット
Microware ULTRA-C compiler	OS-9 ターゲット
Tornado 1.0; GNU Toolkit 2.6	VxWorks ターゲット

これ以外の開発環境を使用する場合はご連絡下さい。

まとめ

以下に、新たにユーザが変換関数、ファンクション、ファンクションブロックを開発する時に、実施すべき作業をまとめます。

- ⇒1. ISaGRAF ライブラリマネージャで新しいエレメント(変換関数、ファンクション、ファンクションブロック)を作成します(名前、コメントの記入)。C言語のテンプレートが自動的に作成されます。
- ⇒2. エレメントがファンクションとファンクションブロックの場合は、ISaGRAF ライブラリマネージャでインターフェース(入力パラメータ、出力パラメータ)を定義します。C言語のヘッダーファイルが自動的に作成されます。
- ⇒3. ISaGRAF ライブラリマネージャで、エレメントの技術メモを記述します。
- ⇒4. ISaGRAF ライブラリマネージャで、変換関数、ファンクション、ファンクションブロックの処理アルゴリズムを記述したC言語のソースファイルを完成させます。他のエディタを使っても構いません。これで、ソースコードの準備は全て完了です。

- ⇒5. ISaGRAF ライブラリマネージャのメニューの「**オプション**」-「**論理番号表示**」を選択して、新しいエレメントに割り当てられた番号を確認します。この番号は".C" や ".H"のファイル名に関連付けされます。
- ⇒6. ".C" や ".H"ファイルを ISaGRAF ターゲットのライブラリやタスク、コンパイラがインストールされているターゲットシステムへダウンロードします。
- ⇒7. 新しいエレメントのソースコードをコンパイルし、エラーがあれば修正します。
- ⇒8. "GR??0LIB.C"のソースファイルに新しいエレメントの宣言サービス関数名を追記します。このファイルは、追加したエレメントの配列になります。??はエレメントの種別によって異なります。
- ⇒9. "GR??0LIB.C"のファイルをコンパイルします。
- ⇒10. ライブラリをビルドする時に使うオブジェクトファイル一覧のファイルに新しいエレメントのオブジェクトファイル名を追加します。
- ⇒11. ライブラリをビルドします。その後、新しいカーネルを作成するためにリンクを実行します。
- ⇒12. 新しくできたカーネルをターゲットシステムにインストールします。
- ⇒13. 新しいエレメントの動作とインターフェースをテストするための ISaGRAF のサンプルアプリケーションを作成します。

C.8 MODBUS リンク

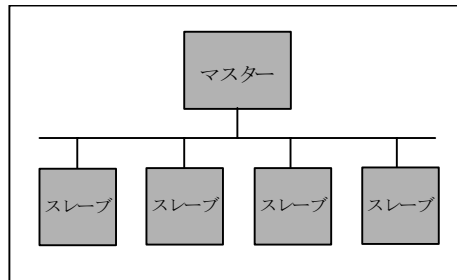
ISaGRAF アプリケーションが開発され動作確認がされれば、ISaGRAF ターゲットと、例えば、SCADAシステムなどと接続することができます。

ISaGRAF は各種のネットワークとの接続を可能にするオープンなシステムです。この項ではシンプルな産業用ネットワークとして MODBUS/MODICON 標準プロトコルを使って説明します。MODBUS にはRS232CやRS485のシリアルリンクが必要となります。

ISaGRAF のワークベンチとターゲットを接続している通信プロトコルは MODBUS に準拠しており、ISaGRAF ワークベンチに代わる MODBUS マスターから ISaGRAF の変数の読み書きが可能です。

C.8.1 MODBUS ネットワークとプロトコル

MODBUS ネットワークは一台のマスターステーションと、複数のスレーブステーションから構成されます。通常、マスターにはSCADAシステム、スレーブにはPLCが使われます。



マスターは1度に1個のスレーブに対して、スレーブ番号を使って1個のリクエストを送ります。そして、スレーブからの応答を待ってから、次のリクエストを送信します。関係のないスレーブは応答しません。

各通信フレームには、スレーブ番号、リクエスト番号とデータ、16ビットチェックサム(CRC)が含まれています。

リクエスト送信後、タイムアウト以内に応答が返らなかった場合は、指定された回数分リクエストの再送を行いません。それでも、応答がない場合は“遮断状態”のエラーを出します。タイムアウトの時間やリトライ回数の設定はスレーブ側の要件(アプリケーション内容などに依存します)を考慮してマスターステーション側で行う必要があります。

スレーブ側でのリクエストの処理でエラーが発生した場合は、本来の応答フレームの代わりにエラーメッセージを返します。

MODBUS はModicon社のプロトコルであり、国際標準規格ではありませんので、MODBUS 互換と言われる中でも多少の仕様の違いがあります。

例えば、以下のような項目が考えられます。

- 実装されているファンクションコードのリスト
- アドレスマッピング
- RTU (バイナリーコード) あるいは ASCII プロトコル
- その他...

C.8.2 ISaGRAF ターゲットとの MODBUS 通信

ISaGRAF アプリケーション変数へのアクセス

ISaGRAF ターゲット通信は以下の5種類の MODBUS ファンクションコードをサポートしています。

1	N ビット読み出し
3	N ワード読み出し
5	1 ビット書き込み
6	1 ワード書き込み
16	N ワード書き込み

ISaGRAF ターゲットのアプリケーション変数は「ネットワークアドレス」を通してアクセスされます。この「ネットワークアドレス」は、ワークベンチの変数辞書エディタであらかじめ定義しておく必要があります。変数は以下のものにアクセスできます。

- ブール型、整数型変数
- 入力、出力、内部変数
- ローカル、グローバル変数

ブール型値を書き込む場合はファンクションコードの5、6、16が使われます。FALSE 変数の書き込みは0、TRUE 変数の書き込みでは0以外の値を書き込みます。

ブール型値を読み出す場合はファンクションの1、3が使われます。ファンクションコード1ではビットフィールドが対応します。ファンクションコード3が使われると、バイトで行われ、TRUE 変数は16進数の 0xFFFF で読み出されます。

整数型変数値を書き込む場合はファンクション6、16が使われます。整数型値は16ビット整数で -32768 ~ +32767 の値を扱えます。ISaGRAF ターゲット内では 32 ビットデータです。

整数型値を読み出す場合はファンクションコード3が使われます。値は16ビット整数で -32768 ~ +32767 の値をとります。ISaGRAF ターゲット内では 32 ビットデータ表現なので、整数型値が最大値、最小値(-32768 ~ +32767)を越える場合は最大、最小値に丸められます。

実数型値は MODBUS のリクエストではアクセスできません。

注意:

- ISaGRAF のターゲットはエラーコード(指定のネットワークアドレスがない、など)の管理を行いません。従って、エラーが発生した場合でもマスターに対して応答を返しません。

略称:

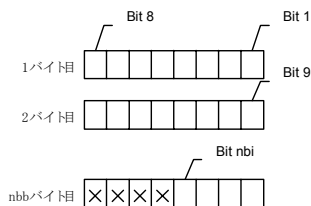
slv	スレーブ番号
nbw	ワード数
nbb	バイト数
nbi	ビット数
addH	ネットワークアドレス(上位バイト)
addL	ネットワークアドレス(下位バイト)
vH	値(上位バイト)
vL	値(下位バイト)
V	バイト値
bfd	ビットフィールド(nbb バイト分)
crcH	チェックサム(上位バイト)
crcL	チェックサム(下位バイト)

ファンクション1: N ビット 読み出し

ネットワークアドレス addH/addL から始まる nbi ビット(ブール型値)の読み出し

Question	Slv	01	addH	addL	00	nbi	crcH	crcL
Answer	Slv	01	nbb	bfd	...		crcH	crcL
			1 バイト目		nbb バイト目			

bfd は nbb バイト分の領域のビットフィールド番号で以下のフォーマットで示されます。



Bit 1 はネットワークアドレス addH/addL におけるブール値

Bit nbi はネットワークアドレス addH/addL + nbi - 1 におけるブール値

X は未定義の値

ファンクション3: N ワード読み出し

MODBUS リンク

ネットワークアドレス addH/addL から始まる nbw ワード分の読み出し

Question	slv	03	addH	addL	00	nbw	crcH	crcL
----------	-----	----	------	------	----	-----	------	------

Answer	slv	03	nbb	vH	vL	...	crcH	crcL
--------	-----	----	-----	----	----	-----	------	------

nbb は vH, vL バイト数の全合計バイト数です。

ファンクション5: 1 ビット書き込み

ネットワークアドレス addH/addL のブール変数(ビット)への書き込み

Question	slv	05	addH	addL	vH	00	crcH	crcL
----------	-----	----	------	------	----	----	------	------

Answer	slv	05	addH	addL	vH	00	crcH	crcL
--------	-----	----	------	------	----	----	------	------

ファンクション6: 1 ワード書き込み

ネットワークアドレス addH/addL の整数型値(ワード)書き込み

Question	slv	06	addH	addL	vH	vL	crcH	crcL
----------	-----	----	------	------	----	----	------	------

Answer	slv	06	addH	addL	vH	vL	crcH	crcL
--------	-----	----	------	------	----	----	------	------

ファンクション16: N ワード書き込み

ネットワークアドレス addH/addL から始まる nbw ワードの書き込み (nbb = 2nbw)

Question	Slv	10	addH	addL	00	nbw	nbb	vH	vL	...	crcH	crcL
----------	-----	----	------	------	----	-----	-----	----	----	-----	------	------

Answer	Slv	10	addH	addL	00	nbw	crcH	crcL
--------	-----	----	------	------	----	-----	------	------

例:

ファンクション1の例:

スレーブ番号 1 のネットワークアドレス 0x1020 から 15 ビット読み出しの場合

Question	01	01	10	20	00	0F	79	04
----------	----	----	----	----	----	----	----	----

Answer	01	01	02	00	12	39	F1
--------	----	----	----	----	----	----	----

読み出された値は 0x0012、2進数では 0000 0000 0001 0010 となります。

最も右側のビットがアドレス 0x1020 の論理値変数となります。

最も左側のビットがアドレス 0x102F の論理値変数となります。

この例では、アドレス 0x1021 と 0x1024 が TRUE となり、その他のアドレスは FALSE となります。

ファンクション16の例:

スレーブ番号1のネットワークアドレス 0x2100 から2words を書き込む。書き込まれる値は、0x1234 と 0x5678 です。

Question	01	10	21	00	00	02	04	12	34	56	78	1C	CA
----------	----	----	----	----	----	----	----	----	----	----	----	----	----

Answer	01	10	21	00	00	02	4B	F4
--------	----	----	----	----	----	----	----	----

ファイル転送

MODBUS プロトコルは基本的なサービスのみを規定していますので、メーカー独自の各種ファンクションコードによって拡張されている場合もあります。

ISaGRAFにおける MODBUS の場合は以下の制限があります。

- ISaGRAF の変数のみアクセスすることができます。
- 多量のデータを高速で転送することはできません。

このため、ISaGRAF は Modbus 風のファイル転送用のプロトコル(リモートファイル管理プロトコル)を設けています。以下のことができます。

- バイナリ、アスキーファイルのダウンロード
- バイナリ、アスキーファイルのアップロード
- 仮想あるいは実際のファイルを介しての動的なデータ交換

このため、ISaGRAF の通信リンクで他のアプリケーションと ISaGRAF ターゲットとが簡単にリモート通信できます。

このプロトコルは以下の考え方に基づいています。

- ISaGRAF ターゲット側のファイルは **リモートファイル**
- マスターコンピュータ側のファイルは **ローカルファイル**
- ファイル内のバイトデータには32ビットの**ベースアドレス+16ビットのバイトアドレス**を使ってアクセスします。

リモートファイル名を選択したり、ベースアドレスを選択したり、リモートファイルのデータを読み出したり、書き出したりするリクエストは16ビットのバイトアドレスを使って行います。

ファンクション 17: データ書き込み

nbb は vH, vL の合計バイト数になります。

Question	slv	11	addH	addL	00	nbb	nbb	vH	vL	...	crch	crcl
----------	-----	----	------	------	----	-----	-----	----	----	-----	------	------

Answer	slv	11	addH	addL	00	nbb	crch	crcl
--------	-----	----	------	------	----	-----	------	------

リクエストの内容は addH/addL アドレスの範囲によって変わります。

- 0xF000: リモートファイル名の初期化

nbb は vH, vL フィールドで示されるファイル名の文字数に相当します。文字列の最後には NULL (\0)を含みます。

MODBUS リンク

もし、ファイルが存在しない場合はファイルを作成します。(属性は読み込み可、書き込み可、実行可になります)

- **0xF002: ベースアドレスを指定値に変更**
nbb は4となります。最初の vH/vL バイトは指定値の上位ワードに、次が下位ワードに相当します。どのような 32 ビット値でも指定できます。
変更後の読み出し、書き込みにはこのベースアドレスが使われます。この変更リクエストがなければデフォルトのアドレスは0です。
- **0xF004: ファイルの削除**
nbb は0となります。
ファイルが存在していて削除可能であれば削除されます。
- **0xF004 を超える値: 予約**
- **0xF000 未満の値: バイト書き込み**
指定のバイト数 (vH、vL、... フィールドで指定する nbb バイト。ここでの上位、下位には意味がありません) を指定済みのリモートファイルに書き込みます。データの書き込みは渡された順 (上図の左から右へ) に行います。書きこむバイトアドレスは addH/addL で指定し、F000 未満でなければなりません。書きこむ先頭のアドレスはあらかじめ指定したベースアドレスに加算したものになります。計算結果のアドレスがファイルサイズを越えている場合は、ファイルサイズが大きくなります。このファンクションでファイルサイズを小さくすることはできません。

ファンクション 18: データ読み出し

Question	slv	12	addH	addL	00	nbb	crcH	crcL
----------	-----	----	------	------	----	-----	------	------

Answer	slv	12	nbb	V	V	...	crcH	crcL
--------	-----	----	-----	---	---	-----	------	------

指定のバイト数 (nbb バイト) を指定済みのリモートファイルから読み出します。データが読み出されるアドレスは addH/addL で指定します。この値は F000 より小さくなければなりません。読み出し位置は選択済みのベースアドレスに addH/addL のバイトアドレスを加えたものになります。

値はファイルから読み出した順に、V フィールドに左から右の順番で格納されます。

例:

リモートファイル名: 'target.fil' の選択。

Question	01	11	F0	00	00	0B	0B	74	...	00	25	9F
----------	----	----	----	----	----	----	----	----	-----	----	----	----

Answer	01	11	F0	00	00	0B	8F	0E
--------	----	----	----	----	----	----	----	----

ベースアドレス 0x10000 を選択。

Question	01	11	F0	02	00	04	04	00	01	00	00	76	11
----------	----	----	----	----	----	----	----	----	----	----	----	----	----

Answer	01	11	F0	02	00	04	6E	CA
--------	----	----	----	----	----	----	----	----

4バイト書き込み: 実際の開始アドレスは 0x107D0, 書き込み値は 01,02,03,04。

Question	01	11	07	D0	00	04	04	01	02	03	04	28	6F
----------	----	----	----	----	----	----	----	----	----	----	----	----	----

Answer	01	11	07	D0	00	04	FC	87
--------	----	----	----	----	----	----	----	----

4バイト読み込み: 実際の開始アドレスは 0x107D0。

Question	01	12	07	D0	00	04	B8	87
----------	----	----	----	----	----	----	----	----

Answer	01	12	04	01	02	03	04	58	7D
--------	----	----	----	----	----	----	----	----	----

C.9 電源異常時の管理

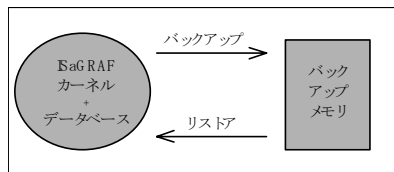
C.9.1 基本事項

電源異常時の管理は以下の3つの点で非常にターゲットに依存する問題です。

- プロセス仕様に依存
- ハードウェア能力に依存
- プログラミング手法に依存

ISaGRAF ターゲットにおける電源異常時の処理に関しては完全な手法が定まっているわけではありません。それぞれのアプリケーション内で対策を施すこと、少なくともハードウェア上で対策を施すことが基本的な考え方です。

電源異常時にシステムを正しく再スタートできるためには、3つの問題を解決しなければなりません。



- データのバックアップ
- 電源異常が発生したことを再スタート時に検出
- バックアップされたデータの復元

上記の2番目の問題点に対する解決方法は、標準的ターゲットソフトウェアにはありません。システム開発者はハードウェアの状況を ISaGRAF からアクセスできるようなC言語によるプログラムを記述して ISaGRAF に組み込むことを考慮する必要があります。

重要なこととしては、どのデータをバックアップすべきかを決めることです。以下に2種類のデータの種別を考えてみましょう。

- アプリケーション変数:
処理中のアイテムの数、電源異常の日時、アプリケーションのパラメータなど。
また、カウンタ、タイマ、中間データ、フラグなどのプログラムの変数。
- プログラム状態:
活性状態のステップ、各C言語プログラムの状態など。

これらの2つの種別について、どのような手法があるかを以下に示します。

C.9.2 アプリケーション変数のバックアップ

保持変数

ワークベンチで変数を登録するとき、内部変数に対しては“保持”属性を設定することが可能です。

ターゲットの1サイクルの最後に、“保持”属性を持った内部変数は指定されたメモリエリアにコピーされます。このメモリ領域は通常、バッテリバックアップされている RAM 領域となります。

ターゲットのスタート時に、1つでも“保持”属性の変数があると、ISaGRAF システムは“保持”変数内容を探しだし、

- 直前に同じアプリケーションを実行していた場合は、全ての保持変数に保管されていた値を代入します。
- 直前に実行していたアプリケーションが同一のアプリケーションでない場合、あるいは、アプリケーションが実行していなかった場合は、ISaGRAF は保管内容は無効とみなし、全ての保持変数の内容をNULLにリセットします。

保持変数値を保管するメモリ領域は、ISaGRAF ワークベンチで変数のタイプ別に定義します。プログラム管理ウィンドウの「コード生成」メニューの「アプリケーションランタイムオプション」コマンドで保持変数のメモリ領域設定を行なうことができます。

パラメータに使われる文字列の意味は以下のようなフォーマットで定義されています。

boo_add, boo_size, ana_add, ana_size, tmr_add, tmr_size, msg_add, msg_size

ここで、

boo_add: ブール型変数を保管するためのアドレス(16進数)で0以外。
boo_size: 指定されたアドレスから利用できるバイト数(16進数)。1つのブール型変数に対して1バイト占有。

ana_add: 整数型変数を保管するためのアドレス(16進数)で0以外。
ana_size: 指定されたアドレスから利用できるバイト数(16進数)。1つの整数型変数に対して4バイト占有。

tmr_add: タイマ型変数を保管するためのアドレス(16進数)で0以外。
tmr_size: 指定されたアドレスから利用できるバイト数(16進数)。1つのタイマ型変数に対して5バイト占有。

msg_add: 可変長文字列型変数を保管するためのアドレス(16進数)で0以外。
msg_size: 指定されたアドレスから利用できるバイト数(16進数)。1つの可変長文字列型変数に対して256バイト占有

要求事項:

- 全タイプの変数を保持する必要がない場合でも、必ず、全タイプのフィールドを指定する必要があります。保持する変数がない場合はサイズを0とします。(ただし、整数型変数の場合、サイズは4が最小値) アドレスはすべて0以外とします。

例:

バックアップ(保持)する変数を以下の数だけあるとします。

20	ブール型変数
0	整数型変数
0	タイマ型変数
3	可変長文字列型変数

バックアップメモリはアドレス 0xA2F200 にあるとします。

保管領域は以下のように設定します。

ブール型変数はアドレス 0xA2F200、サイズは20バイト

整数型変数はアドレス 0xA2F214、サイズは4バイト

タイマ型変数のためにダミーアドレス 0xA2F200 でサイズは0バイト

可変長文字列型変数はアドレス 0x A2F218 でサイズ 256×3 バイト

この場合のワークベンチ側(保持変数アドレス領域)に設定する値は以下のようになります。

A2F200,14,A2F214,4,A2F200,0,A2F218,300

SYSTEM ファンクションコール

もし、アプリケーション変数のほとんどを保持する必要がある場合は、SYSTEM ファンクションを使って変数のセットをまとめて扱うことも可能です。ただし、この場合は変数のバックアップや復元はアプリケーションレベルでプログラマが記述する必要があります。

まず、指定された、あるいは、全タイプの変数に対して以下のようにバックアップ用のメモリエリアを定義する必要があります。

```
<new_address> := SYSTEM(SYS_INITxxx,<address>);
```

ここで、

- <address> はバックアップメモリアドレス (16# で16進数フォーマット)を指定。これは必ず偶数アドレスである必要があります。奇数アドレスではエラーが発生します。
- SYS_INITxxx は以下のようになります。
 - * SYS_INITBOO ブール型変数に対するバックアップアドレスの指定
 - * SYS_INITANA 整数型変数に対するバックアップアドレスの指定
 - * SYS_INITTMR タイマ型変数に対するバックアップアドレスの指定
 - * SYS_INITALL ブール、整数、タイマ型全部の変数に対するバックアップアドレスの指定
- <new_address> は次の空きアドレスが戻されます。即ち、<address> + バックアップされた変数のサイズ(バイト数 = SYS_INITxxx で指定)です。これで必要なメモリバックアップ用サイズを確認できます。エラーが発生していると<new_address> は0となります。

次に、実際に変数のバックアップを行います。この手続きはアプリケーションのどの時点でも行えますが、バックアップそのものは現在のターゲットサイクルの最後に1回だけ行われます。

ハードウェアからのブール型入力やC言語ファンクションで電源異常を検出すると想定します。この場合、電源異常を検出してから少なくとも1サイクル遅れで ISaGRAF はバックアップ処理を行うことになります。

```
<error> :=SYSTEM(SYS_SAVxxx,0);
```

ここで、

- SYS_SAVxxx は以下のようになります。
 - * SYS_SAVBOO ブール型変数のバックアップ操作
 - * SYS_SAVANA 整数型変数のバックアップ操作
 - * SYS_SAVTMR タイマ型変数のバックアップ操作
 - * SYS_SAVALL ブール、整数、タイマ型全変数のバックアップ操作
- <error> エラー発生時(例えば、以前に SYS_INITXXX が呼ばれていない場合)は0以外の値が戻ります。

バックアップした変数の復元が必要なときは、アプリケーションのどの時点でも手続きを行えますが、復元そのものは現在のターゲットサイクルの最後に1回だけ行われます。バックアップされているデータの整合性を保証するために、適当な整数に印となるような定数を代入する方法もあります。

```
<error> := SYSTEM(SYS_RESTxxx,0);
```

ここで、

- SYS_RESTxxx は以下のようになります。
 - * SYS_RESTBOO ブール型変数の復元
 - * SYS_RESTANA 整数型変数の復元
 - * SYS_RESTTMR タイマ型変数の復元
 - * SYS_RESTALL ブール、整数、タイマ型全変数の復元
- <error> エラー発生時(例えば、以前に SYS_INITXXX が呼ばれていない場合)は0以外の値が戻ります。

以下に、変数のバックアップを行うための SYSTEM ファンクションコマンドをまとめます。

コマンド (予約キーワード)	値	意味
SYS_INITBOO	16#20	ブール型変数のバックアップの初期化
SYS_SAVBOO	16#21	ブール型変数の保存
SYS_RESTBOO	16#22	ブール型変数の復元
SYS_INITANA	16#24	整数型変数のバックアップの初期化
SYS_SAVANA	16#25	整数型変数の保存
SYS_RESTANA	16#26	整数型変数の復元
SYS_INITTMR	16#28	タイマ型変数のバックアップの初期化
SYS_SAVTMR	16#29	タイマ型変数の保存
SYS_RESTTMR	16#2A	タイマ型変数の復元
SYS_INITALL	16#2C	全変数のバックアップの初期化
SYS_SAVALL	16#2D	全変数の保存
SYS_RESTALL	16#2E	全変数の復元

コマンド (予約キーワード)	引数	戻り値
SYS_INITxxx	メモリアドレス	次の空きアドレス
SYS_SAVxxx	0	正常時は0
SYS_RESTxxx	0	正常時は0

カスタマイズプログラム

C言語ファンクション、ファンクションブロックを使用して、バッテリーバックアップメモリ領域の内容にアクセスすることも可能です。この場合、アプリケーションの任意のタイミングで変数の保存、復元が可能になります。

例:

1) アプリケーション固有の手続き:

backup, restore_temp, restore_date, restore_cpt はユーザC言語ファンクションになります。

backup(temperature, date, cnt); 3 個の重要なデータを保存

temperature := **restore_temp**(); 温度を復元

date := **restore_date**(); 日付を復元

cnt := **restore_cnt**(); カウンタを復元

2) 一般的な手続き:

backup_init, backup, backup_link, restore はユーザC言語ファンクションになります。

save_id := **backup_init**(address, size); バックアップする配列のメモリ確保

backup(save_id, cpt1, 3); cpt1 を3番目の要素として保管。

rest_id := **backup_link**(address, size); バックアップメモリをリンク

cpt1 := **restore**(rest_id, 3); バックアップ値 cpt1 の復元。

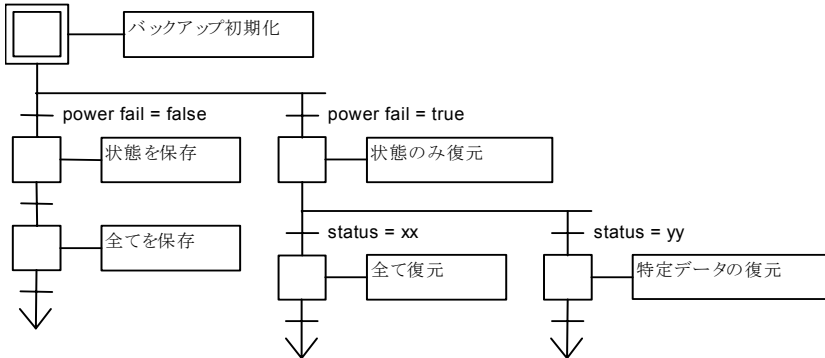
C.9.3 プログラム状態のバックアップ

全プログラムの全状態をバックアップすることは可能かもしれませんが、以下の3つの理由からおすすめできません。

- プロセスによっては再スタート前に特定の手続きを行う必要があるため。
- 全アプリケーションの全状態について扱うことはあまり意味がないため。
- ある外部のリソース、すなわちCプログラムや周辺機器の中には自動的に再スタートできないものがあるため。

最も良い方法は、再スタートに必要と思われるプロセスの状態を整数型、ブール型変数にバックアップしておくことかもしれません。それで完全ではありませんが、最低限のプロセスのイメージからの再スタートやアプリケーションの初期化などが可能になります。ただし、完全自動の再スタートを追求することには無理があるかもしれません。

例:



C.10 付録: エラーリストと説明

エラーリスト:

コード	メッセージ	種別
1	ランタイム データベース用にメモリをアロケートできません。	system
2	不正なアプリケーションデータベースが指定されました。 (Motorola/Intel)	application
3	通信メールボックスをアロケートできません。	system
4	カーネルデータベースをリンクできません。	system
5	カーネルへのリクエスト送信がタイムアウトしました。	system
6	カーネルからのアンサー待ちでタイムアウトしました。	system
7	通信を初期化できません。	system
8	保持変数用のメモリをアロケートできません。	application
9	アプリケーションが停止しました。	application
10	同時に発生する N 又は P アクションが多すぎます。	application
11	同時に発生するセットアクションが多すぎます。	application
12	同時に発生するリセットアクションが多すぎます。	application
13	未知の TIC 命令です。	application
16	データ読み出しリクエストに応答できません。	system
17	データ書き込みリクエストに応答できません。	system
18	デバッカーのツッシュンリクエストに応答できません。	system
19	MODBUS リクエストに応答できません。	system
20	デバッカーのプリケーションリクエストに応答できません。	system
21	デバッカーに応答できません。	system
23	未知のリクエストコードです。	system
24	イーサネット通信エラーです。	system
25	通信の同期エラーです。	system
28	アプリケーション用にメモリをアロケートできません。	application
29	アプリケーションの更新用にメモリをアロケートできません。	application
30	未知のOEMキーコードです。	application
31	ブール型入力ボードを初期化できません。	application
32	アナログ型入力ボードを初期化できません。	application
33	可変長文字列型入力ボードを初期化できません。	application
34	ブール型出力ボードを初期化できません。	application
35	アナログ型出力ボードを初期化できません。	application
36	可変長文字列型出力ボードを初期化できません。	application
37	ブール型ボードからの入力できません。	application
38	アナログ型ボードからの入力できません。	application
39	可変長文字列型ボードからの入力できません。	application
40	ブール型出力変数の出力できません。	application
41	アナログ出力変数の出力できません。	application
42	可変長文字列型出力変数の出力できません。	application
43	ブール型変数に operate 命令を実行できません。	application

44	アナログ型変数に operate 命令を実行できません。	application
45	可変長文字列型変数に operate 命令を実行できません。	application
46	ボードをオープンできません。	application
47	ボードをクローズできません。	application
50	ブール型出力変数の上書きができません。	program
51	アナログ型出力変数の上書きができません。	program
52	可変長文字列型出力変数の上書きができません。	program
61	未知のシステムリクエストコード	program
62	サンプリング周期がオーバーフローしました。	program
63	ファンクションがターゲットに実装されていません。	application
64	整数がゼロで除算されました。	program
65	変換関数がターゲットに実装されていません。	application
66	ファンクションブロックがターゲットに実装されていません。	application
67	標準ファンクションがターゲットに実装されていません。	application
68	実数がゼロで除算されました。	application
69	operate 命令のパラメータが不正です。	application
72	アプリケーションシンボルは修正できません。	application
73	更新できません。： 異なったブール型変数のセットです。	application
74	更新できません。： 異なった整数型変数のセットです。	application
75	更新できません。： 異なったタイマー変数のセットです。	application
76	更新できません。： 異なった可変長文字列型変数のセットです。	application
77	更新できません。： 新しいアプリケーションが見つかりません。	application
> 100	OEMエラーコード。 100 を超えるものはOEMキーごとのエラーメッセージです。詳細はI ／Oボードの技術メモ、あるいは、供給元に問い合わせください。	

エラーには以下の3つの種類があります。

– System エラー:

ターゲットソフト、あるいはハードウェアに起因するエラーです。

ターゲット (PLC) のハードリセットあるいは別のアプリケーションをRUNさせて同じ現象が現われるか確認します。

これらのエラーが発生した場合はターゲットソフトのベンダーの ISaGRAF のサポートへ連絡願います。

– Application エラー:

アプリケーションのパラメータやサイズ及び内容に起因するエラーです。

この場合は別のアプリケーションをRUNさせればエラーが消えることを確認して下さい。もし、引き続きエラーが出る場合は system エラーの可能性もあります。

– Program エラー:

プログラムのシーケンスに起因するエラーです。

このエラーはアプリケーションをサイクルモードで実行したり、ある特定のプログラムを停止させることで消える場合があります。

エラーの説明:

1	ランタイム データベース用にメモリをアロケートできません。	system
---	-------------------------------	--------

メモリが不足しています。ハードウェアをチェックして下さい。

2	不正なアプリケーションデータベースが指定されました。 (Motorola/Intel)	application
---	--	-------------

アプリケーションコードファイルが正しくありません。例えば、INTEL プラットフォームにMOTOROLAコードがダウンロードされた時や、ファイルが変更された場合に発生します。

3	通信メールボックスをアロケートできません。	system
---	-----------------------	--------

内部タスク通信用のメモリスペース3を確保できなかった場合に発生します。通信タスクがエラーを発生します。

4	カーネルデータベースをリンクできません。	system
---	----------------------	--------

指定されたスレーブ番号のカーネルタスクが無かった場合に発生します。通信タスクがエラーを発生します。

5	カーネルへのリクエスト送信がタイムアウトしました。	system
---	---------------------------	--------

通信タスクがカーネルに対してリクエストを送れません。カーネルタスクが実行していないか、ビジー状態の可能性があります。

6	カーネルからのアンサー待ちでタイムアウトしました。	system
---	---------------------------	--------

通信タスクがカーネルから応答を受信できません。カーネルタスクが実行していないか、ビジー状態の可能性があります。

7	通信を初期化できません。	system
---	--------------	--------

通信レイヤーが物理層を初期化できなかった時に発生します。また、通信ポートを指定しない場合も発生します。このエラーによりカーネルタスクの実行が停止することはありませんが、通信を行なうことはできません。

8	保持変数用のメモリをアロケートできません。	application
---	-----------------------	-------------

保持変数を管理できません。2つの理由が考えられます。

- パラメータとして渡された文字列が文法的に正しくなかった場合。
- 各ブロックで指定されたメモリサイズが不足している場合。

保持変数パラメータの文法を確認し、また、保持変数の数を減らすことを考えて下さい。

9	アプリケーションが停止しました。	application
---	------------------	-------------

アプリケーションがデバッガーから停止されたときに必ずでる警告です。

10	同時に発生する N 又は P アクションが多すぎます。	application
----	-----------------------------	-------------

1回のサイクル中にノーマル(N)、パルス(P)アクションが多すぎる場合に発生します。問題の発生箇所を見つける場合にはサイクルモードへの切り替えが有効です。同時に行なえるアクションの数は(2+SFCプログラム数×4)個程度です。

11	同時に発生するセットアクションが多すぎます。	application
----	------------------------	-------------

1回のサイクル中にセット(S)アクション(ステップがアクティブになったときに実行される)が多すぎる場合に発生します。上記のエラーを参考にしてください。

12	同時に発生するリセットアクションが多すぎます。	application
----	-------------------------	-------------

1回の PLC サイクル中にリセット(R)アクション(ステップが非アクティブになったときに実行される)が多すぎる場合に発生します。上記のエラーを参考にしてください。

13	未知の TIC 命令です。	application
----	---------------	-------------

アプリケーションコード(TIC: Target Independent Code)にエラーが検出されました。2つの場合が考えられます。

- 外部プログラムがアプリケーションコードに書き込みをしていることが考えられます。サイクルモードに切り替えてエラーが発生する箇所を特定して下さい。また、I/Oドライバのパラメータが間違っていないか確認して下さい。
- ターゲットソフトによっては一部の命令しか含んでいないものもあります。アプリケーションが含まれていない命令や変数型を指定したことが考えられます。

16	データ読み出しリクエストに応答できません。	system
----	-----------------------	--------

ISaGRAF の MODBUS リクエストのファンクション18(ファイルの読み出し)においてエラーが発生しました。通信の接続および、ターゲットとマスターの設定を確認して下さい。

17	データ書き込みリクエストに応答できません。	system
----	-----------------------	--------

ISaGRAF の MODBUS リクエストのファンクション17(ファイルの書き込み)においてエラーが発生しました。通信の接続および、ターゲットとマスターの設定を確認して下さい。

18	デバッガーのツェションリクエストに応答できません。	system
----	---------------------------	--------

デバッガーのリクエストに対する応答時にエラーが発生しました。通信の接続および、ターゲットとマスターの設定を確認して下さい。

19	MODBUS リクエストに応答できません。	system
----	-----------------------	--------

MODBUS リクエストに対する応答時にエラーが発生しました。通信の接続および、ターゲットとマスターの設定を確認して下さい。

20	デバッカーのブリークセッションリクエストに応答できません。	system
----	-------------------------------	--------

デバッカーのリクエストに対して応答時にエラーが発生しました。通信の接続および、ターゲットとマスターの設定を確認して下さい。

21	デバッカーに応答できません。	system
----	----------------	--------

デバッカーのリクエストに対して応答時にエラーが発生しました。通信の接続および、ターゲットとマスターの設定を確認して下さい。

23	未知のリクエストコードです。	system
----	----------------	--------

デバッカーからのリクエストが未知の内容です。

24	イーサネット通信エラーです。	system
----	----------------	--------

このエラーはデバッカーを閉じて通信を閉じるたびに発生します。この場合はシステムは正常に動作しています。それ以外で発生する場合は、イーサネット通信でエラーが検出されています。接続を確認し、システム設定内容をターゲットとワークベンチ側でチェックします。もし、2番目のエラーフィールドが表示されていれば、以下のような状況がわかります。

- 1: 送信及び受信中のエラー
- 2: ソケットを生成中のエラー
- 3: ソケットのバインディングあるいはリスニング中のエラー
- 4: 新クライアントを受付中のエラー

25	通信の同期エラーです。	System
----	-------------	--------

ターゲットとワークベンチ側の通信の同期エラーです。ターゲットとワークベンチの通信接続状態やシステム設定パラメータなどを確認してください。

28	アプリケーション用にメモリをアロケートできません。	application
----	---------------------------	-------------

メモリ不足です。アプリケーションサイズに対するハードウェアメモリサイズを確認して下さい。

29	アプリケーションの更新用にメモリをアロケートできません。	application
----	------------------------------	-------------

メモリ不足です。アプリケーションサイズに対するハードウェアメモリサイズを確認して下さい。

30	未知のOEMキーコードです。	application
----	----------------	-------------

アプリケーションがターゲットソフトで認識できないOEMコードのI/Oボードを使っています。ワークベンチのI/O接続エディタでボードを“バーチャル”に設定して問題のI/Oボードを特定して下さい。ワークベンチのライブラリがターゲットのバージョンと合っていない可能性があります。

31	ブール型入力ボードを初期化できません。	application
----	---------------------	-------------

ブール型入力ボードの初期化に失敗しました。ワークベンチのI/O接続エディタでブール型入力ボードのパラメータ設定を確認して下さい。

32	アナログ型入力ボードを初期化できません。	application
----	----------------------	-------------

アナログ型入力ボードの初期化に失敗しました。ワークベンチのI/O接続エディタでアナログ型入力ボードのパラメータ設定を確認して下さい。

33	可変長文字列型入力ボードを初期化できません。	application
----	------------------------	-------------

可変長文字列型入力ボードの初期化に失敗しました。ワークベンチのI/O接続エディタで可変長文字列型入力ボードのパラメータ設定を確認して下さい。

34	ブール型出力ボードを初期化できません。	application
----	---------------------	-------------

ブール型出力ボードの初期化に失敗しました。ワークベンチのI/O接続エディタでブール型出力ボードのパラメータ設定を確認して下さい。

35	アナログ型出力ボードを初期化できません。	application
----	----------------------	-------------

アナログ型出力ボードの初期化に失敗しました。ワークベンチのI/O接続エディタでアナログ型出力ボードのパラメータ設定を確認して下さい。

36	可変長文字列型出力ボードを初期化できません。	application
----	------------------------	-------------

可変長文字列型出力ボードの初期化に失敗しました。ワークベンチのI/O接続エディタで可変長文字列型出力ボードのパラメータ設定を確認して下さい。

37	ブール型ボードからの入力ができません。	application
----	---------------------	-------------

ブール型論理値入力ボードのリフレッシュ中にエラーが発生しました。ワークベンチのI/O接続エディタでボードのパラメータ設定を確認して下さい。

38	アナログ型ボードからの入力ができません。	application
----	----------------------	-------------

アナログ型入力ボードのリフレッシュ中にエラーが発生しました。ワークベンチのI/O接続エディタでボードのパラメータ設定を確認して下さい。

39	可変長文字列型ボードからの入力ができません。	application
----	------------------------	-------------

可変長文字列型入力ボードのリフレッシュ中にエラーが発生しました。ワークベンチのI/O接続エディタでボードのパラメータ設定を確認して下さい。

40	ブール型出力変数の出力ができません。	application
----	--------------------	-------------

ブール型出力ボードの更新中にエラーが発生しました。ワークベンチのI/O接続エディタでボードのパラメータ設定を確認して下さい。

付録: エラーリストと説明

41	アナログ型出力変数の出力ができません。	application
----	---------------------	-------------

アナログ型出力ボードの更新中にエラーが発生しました。ワークベンチのI/O接続エディタでボードのパラメータ設定を確認して下さい。

42	可変長文字列型出力変数の出力ができません。	application
----	-----------------------	-------------

可変長文字列型出力ボードの更新中にエラーが発生しました。ワークベンチのI/O接続エディタでボードのパラメータ設定を確認して下さい。

43	ブール型変数に <code>operate</code> 命令を実行できません。	application
----	--	-------------

ブール型変数に対しての **OPERATE** コール中にエラーが発生しました。**OPERATE** のパラメータ及びボードの技術メモを参照してください。

44	アナログ型変数に <code>operate</code> 命令を実行できません。	application
----	---	-------------

アナログ型変数に対しての **OPERATE** コール中にエラーが発生しました。**OPERATE** のパラメータ及びボードの技術メモを参照してください。

45	可変長文字列型変数に <code>operate</code> 命令を実行できません。	application
----	---	-------------

可変長文字列型変数に対しての **OPERATE** コール中にエラーが発生しました。**OPERATE** のパラメータ及びボードの技術メモを参照してください。

46	ボードをオープンできません。	application
----	----------------	-------------

アプリケーションがターゲットが認識できないI/Oボードを使用しています。ワークベンチでI/O接続を確認してください。ワークベンチのライブラリがターゲットのバージョンと合っていない可能性があります。

47	ボードをクローズできません。	application
----	----------------	-------------

アプリケーションがターゲットが認識できないI/Oボードを使用しています。ワークベンチでI/O接続を確認してください。

50	ブール型出力変数の上書きができません。	program
----	---------------------	---------

2つのSFCシーケンスが同一のブール型出力変数に対して同じサイクル内で書き込みを行なっています。予測できない結果を防ぐためにも、このようなことは避けるべきです。この場合は、プログラムの階層で上位にあるものが優先度が高くなります。2つのSFCプログラムが同一レベルであれば結果は予測できません。

51	アナログ型出力変数の上書きができません。	program
----	----------------------	---------

2つのSFCシーケンスが同一のアナログ型出力変数に対して同じサイクル内で書き込みを行なっています。上記の説明を参照してください。

52	可変長文字列型出力変数の上書きができません。	program
----	------------------------	---------

2つのSFCシーケンスが同一の可変長文字列型出力変数に対して同じサイクル内で書き込みを行なっています。上記の説明を参照してください。

61	未知のシステムリクエストコード	program
----	-----------------	---------

プログラムが **SYSTEM** コールで不正なコードを使用しています。

62	サンプリング周期がオーバーフローしました。	program
----	-----------------------	---------

ターゲットのサイクルタイムがワークベンチで指定したものを越えました。
マルチタスクシステムでは、このことはカーネルタスクに十分なCPU時間が与えられていないことを意味します。たとえ「現在のサイクル」の表示が指定のものより小さくてもです。
シングルタスクシステムでは、1サイクル中の処理が多すぎることを意味します。

本メッセージを消去するために、以下のような対策が考えられます。

- エラーが検出される場所での処理の数を減らします。
- アクティブなトークンの数、TRUEとなるトランジションの数を減らしたり、複雑な処理を最適化したりします。
- 他のタスクの CPU 負荷を減らし、ISaGRAF に処理時間を与えます。
- 通信量を減らしてカーネルタスクに時間を与えます。
- ダイナミックにサイクルタイムを変更して、処理段階ごとに合ったサイクルタイムになるように設定します。
- サイクルタイムを0に設定してオーバーフローチェックを無視します。

63	ファンクションがターゲットに実装されていません。	application
----	--------------------------	-------------

アプリケーションがターゲットにないC言語ファンクションを使っています。ワークベンチライブラリのバージョンがターゲットと合っていない可能性があります。

64	整数がゼロで除算されました。	program
----	----------------	---------

アプリケーションで整数型を0で除算しようとした。予測できない結果を招かぬよう、このようなことは避けるべきです。
この場合、演算結果は整数型の最大の値になります。
整数が負の時は、演算結果も負になります。

65	変換関数がターゲットに実装されていません。	application
----	-----------------------	-------------

アプリケーションがターゲットにないC言語変換関数を使っています。ワークベンチライブラリのバージョンがターゲットと合っていない可能性があります。
この場合、変換は行なわれません。

66	ファンクションブロックがターゲットに実装されていません。	application
----	------------------------------	-------------

アプリケーションがターゲットにないC言語ファンクションブロックを使っています。ワークベンチライブラリのバージョンがターゲットと合っていない可能性があります。

付録: エラーリストと説明

67	標準ファンクションがターゲットに実装されていません。	application
----	----------------------------	-------------

アプリケーションがターゲットに実装されていないファンクションブロックを使っています。ただし、これは標準ファンクションブロックです。ターゲットソフトのベンダーにお問い合わせ下さい。

68	実数がゼロで除算されました。	application
----	----------------	-------------

アプリケーションで実数型を0で除算しようとしました。予測できない結果を招かぬよう、このようなことは避けるべきです。
この場合、演算結果は実数型の最大の値になります。
実数が負の時は、演算結果も負になります。

69	operate 命令のパラメータが不正です。	application
----	------------------------	-------------

OPERATE 命令のパラメータが不正です。通常はコード生成の段階で見つかるエラーです。パラメータにはタイマー型か、入力・出力変数以外の変数が使えます。

72	アプリケーションシンボルは修正できません。	application
----	-----------------------	-------------

アプリケーションを更新しようとしたのですが、シンボルテーブルが異なっているために更新できませんでした。変数、ファンクションブロックインスタンスなどが変更、追加、削除されている可能性があります。

73	更新できません。: 異なった ブール型変数のセットです。	application
----	------------------------------	-------------

アプリケーションを更新しようとしたのですが、できませんでした。ブール型変数が変更、追加、削除されています。

74	更新できません。: 異なった整数型変数のセットです。	application
----	----------------------------	-------------

アプリケーションを更新しようとしたのですが、できませんでした。アナログ型変数が変更、追加、削除されています。

75	更新できません。: 異なったタイマー変数のセットです。	application
----	-----------------------------	-------------

アプリケーションを更新しようとしたのですが、できませんでした。タイマ型変数が変更、追加、削除されています。

76	更新できません。: 異なった可変長文字列型変数のセットです。	application
----	--------------------------------	-------------

アプリケーションを更新しようとしたのですが、できませんでした。可変長文字列型変数が変更、追加、削除されています。

77	更新できません。: 新しいアプリケーションが見つかりません。	application
----	--------------------------------	-------------

更新されたアプリケーションがメモリ上に見つけれられません。ダウンロード中に何か異常が発生しました。

D. 用語集

Action アクション	SFCプログラムのステップがアクティブの時に実行される一連のステートメントや代入操作。
Action (FC) FC アクション	フローチャートのシンボル。フローチャート実行時の命令が記述されるオブジェクト。
Activity of a step ステップの活性化状態	SFCのトークンがあるステップの状態。ステップに割り当てられているアクションはこの状態に基づいて実行されます。
Alias エイラス	変数のコメントの一部(先頭16バイト)で、ラダーエディタで変数のコメントの表示として使われます。
Analog アナログ	変数のタイプ。整数または実数の連続量です。
Attribute 属性	変数の属性で内部、入力、出力、定数があります。
Begin section Begin セクション	サイクリックプログラムの集まり。ターゲットサイクルの最初に実行されるセクションです。
Beginning step 開始ステップ	SFC(あるいはマクロステップ)の最初のステップ。直前のトランジションを持ちません。
Boolean ブール型	変数のタイプ。 TRUEかFALSEの状態をもちます。
Boolean action ブールアクション	SFCアクション。ステップのアクティブ状態をブール型変数に代入します。
Breakpoint ブレイクポイント	デバッグ時にユーザがSFCのステップやトランジションに設定できる一時停止ポイント。ターゲットシステムはSFCのトークンが設定されたポイントに移動すると停止します。
C function C言語ファンクション	C言語で記述されたファンクション。ISaGRAFプログラムから同期的に呼び出されます。標準のC言語ファンクションの他、ユーザが独自に開発することも可能です。
C language C言語	高級言語の一つ。 C言語ファンクション類を開発してISaGRAFとリンクすることが可能です。
C source code Cソースコード	C言語ファンクション類のプログラムソースコードとなるテキストファイル。

C source header Cソースヘッダ	C言語ファンクション類の定義やタイプを記述したヘッダファイル(*.Hファイル)。
Cell セル	SFC、FBD、LDといったグラフィック言語エディタでのシンボル配置における最小単位エリア。
Child SFC program チャイルドSFCプログラム	親SFCプログラムからコントロールされる子SFCプログラム。
Clearing a transition トランジションの通過	プログラム実行時にSFCの直前のステップのトークンが取り除かれて直後のステップにトークンがセットされること。
Coil コイル	LDプログラムで出力変数への代入を表わすシンボル。
Comment コメント	プログラム中に含めることができるテキスト。プログラムの実行に影響を与えません。
Comment (SFC) コメント(SFC)	SFCのステップやトランジションに付属するテキスト。プログラムの実行に影響を与えません。
Common 共通	定義ワードの範囲の一つ。どのプロジェクトのどのプログラムからも扱えます。
Condition (for a transition) トランジションの条件	SFCTランジションに割り付けられた論理式。この条件がFALSEの時にはトランジションは通過できません。
Connector (FC) FC コネクタ	フローチャートのアクションやテストへのリンクを意味するグラフィックシンボル。リンク先のリファレンス番号がついた小さな円で表示されます。
Constant expression 定数式	定数値で記述したリテラル式。定数式は1種類の型にのみ適用できます。
Contact 接点	LDプログラムでの入力変数の状態を表わすシンボル。
Conversion 数値変換	アナログ型の入力／出力変数に付けることができるフィルタ。入力や出力が更新されるときに自動的に使われます。
Conversion function 変換関数	C言語で記述された数値変換関数。アナログ入力、出力変数に付けることができます。
Conversion table 数値変換テーブル	アナログ入力、出力変数に付けることができる線形な変換を表現するポイント(X、Y)集合。
Cross references クロスリファレンス	ISaGRAFワークベンチによって計算される、辞書に宣言されている変数とその使用個所の情報。

用語集

Current result (IL) 現在結果(IL)	ILプログラムでの 1 個の命令の計算結果(現在結果)。現在結果は次の命令で変更できます。また変数への代入にも使われます。
Cycle timing サイクルタイム	ターゲットでのプログラム実行サイクル(周期)
Cycle to cycle mode サイクルモード	プログラム実行モードの一つ。このモードではデバッガーからの指示で1サイクル毎の実行が可能です。
Cyclic サイクリック	プログラムの属性の一つ。周期的に実行されます。
Decision (FC) デシジョン/テスト(FC)	テストともいいます。フローチャートの条件判断を意味するシンボルで、論理式を割り付けます。フローは YES/NO の出力から接続されます。
Defined word 定義ワード	プログラムで使用される表現を置き換える重複しない識別子。
Delayed operation (IL) 遅延操作(IL)	ILプログラムの命令の一つ。")"がプログラムの後で現れた時点で実行される命令です。
Diary 修正履歴	プログラムの修正時に残した修正履歴テキストファイル。1 回の修正ごとにその編集日付が追記されます。
Dictionary 辞書	1 個のプロジェクトでの各種タイプの変数や定義ワードを登録したもの。
Directly Variable 直接表現変数	Represented I/O変数を辞書に宣言せずに、I/OボードのI/Oチャネルを直接%表現で指定すること。 例: %Ixs.c (s:スロット番号、c:チャネル番号)
Edge エッジ	ブール型変数の変化。立ち上がりとは FALSE 状態から TRUE 状態になること。立ち下がりとは TRUE 状態から FALSE 状態になることを指します。
End section End セクション	サイクリックプログラムの集まり。ターゲットサイクルの最後に実行されるセクションです。
Ending step 終了ステップ	SFC(あるいはマクロステップ)の最後のステップ。直後にはトランジションはありません。
Expression 式	値を評価するための演算子や識別子をまとめた表現。
Father SFC program 親SFCプログラム	他のSFCプログラム(チャイルドSFCプログラム)をコントロールできるSFCプログラム。

FBD	ファンクションブロックダイアグラム言語
FC	フローチャート。
Flow Chart フローチャート	処理フローを表現するグラフィックプログラミング言語。プログラムの内容を記述するアクションと条件を記述し処理のフローを決定するテスト、そしてそれらを結ぶフローから構成されます。
Function block ファンクションブロック	FBD言語でのグラフィックコンポーネント。ISaGRAF のライブラリの基本的なファンクションを表現します。
Functional Block Diagram ファンクションブロックダイアグラム	グラフィック言語の一つ。基本的なブロックを接続したダイアグラム。
Global グローバル	変数や定義ワードの範囲。プロジェクト内の全てのプログラムから扱えます。
Hierarchy 階層	プロジェクト構成しているプログラムの階層。階層ツリーはプログラム間の親子関係を表わします。
I/O board I/Oボード	ハードウェアリソース。I/Oボードは1つの方向（入力あるいは出力）と1種類の型を持ちます。I/OボードのパラメータはISaGRAFライブラリに記述されています。
I/O channel I/Oチャンネル	I/Oボードの単一の接続ポイント。各I/Oチャンネルに一つのI/O変数を接続できます。
I/O connection I/O接続	プログラムの変数と、ターゲットシステム上のI/Oボードのチャンネル間のリンクの定義。
I/O variable I/O変数	入力・出力デバイスに割り当てられた変数。I/O変数は必ずどこかのI/Oボードのチャンネルに接続されなければなりません。
Identifier 識別子	プログラムでの変数や定数を表わす重複しないワード。
IL	インストラクションリスト言語
Initial situation 初期状態	1個のSFCプログラムのイニシャルステップの集まり。プログラムがスタートする前の状態。
Initial step イニシャルステップ	SFCプログラムのステップのうち、プログラムがスタートしたときに最初にアクティブになる特別なステップ。
Input 入力	変数の属性。入力デバイスに割り当てられます。

用語集

Instruction インストラクション	ILプログラムの基本命令。 1行で書きます。
Instruction List インストラクションリスト	低レベルのリテラル言語。基本的な操作のシーケンシャルなリストで記述します。
Integer 整数	アナログ変数のクラス。符号付きの32ビットフォーマット。
Internal 内部	変数の属性。入力、出力デバイスに割り当てられない内部の変数。
Jump to a step ステップへのジャンプ	SFCで指定されたステップへのリンクを表わす矢印(↓)シンボル。ジャンプ先のリファレンス番号が付きます。
Keyword キーワード	言語の予約語。
Label (IL) ラベル(IL)	IL言語の命令行の先頭に付けて行を識別します。ジャンプ(JMP)命令のオペランドにも使えます。
Ladder Diagram ラダーダイアグラム	論理式をプログラムするための、コンタクトとコイルを使うグラフィック言語。
LD	ラダーダイアグラム言語。
Level 1 of the SFC SFCのレベル1	SFCプログラムのおおまかな記述。レベル1ではステップとトランジションの接続とそれぞれのコメントを示します。
Level 2 of the SFC SFCのレベル2	SFCプログラムの詳細な記述。SFCステップのアクション記述やトランジションのブール条件の記述がされます。
Library ライブラリ	ハードウェアやソフトウェアのリソースの集合。これらはプロジェクト内で自由に利用できます。
Local ローカル	変数や定義ワードの範囲。プロジェクト中の1個のプログラム内でしか使用できません。
Locked I/O ロック	入力、出力変数が対応するI/Oデバイスから論理的に切り離されている状態。デバッガーからロックコマンドを送ることができます。
Macro step マクロステップ	メインSFCチャートとは別に記述されるステップとトランジションのグループ。メインSFCから呼び出されます。重複して呼び出すことはできません。
Matrix マトリックス	グラフィカルプログラムエディタにおける編集エリアを長方形のセルに分割した区域。

Message 可変長文字型	変数の型。可変長の文字列をもつ変数。文字列、ストリングとも呼ばれます。
Modbus	マスタースレーブ通信プロトコルの一種。ISaGRAFはModbusのスレーブになります。この場合、マスターはISaGRAF以外の外部のシステム（スーパーバイザリシステムなど）です。
Modifier (IL) 修飾子(IL)	IL命令の後ろに付く1文字。この文字で命令の意味を修飾します。
Network address ネットワークアドレス	各変数に任意で付けられる16進数アドレス。このアドレスはターゲットシステムがModbusプロトコルで外部システム（マスター）と接続する場合に使います。
Non-stored action ノンストアードアクション (ノーマルアクション)	SFCのアクション。対応しているステップがアクティブ状態のときに毎回ターゲットサイクル時に実行される一連のアクション。
OEM key code (I/O board) I/Oボードの OEM キーコード	ISaGRAFのライブラリに登録されるI/Oボードを認識するコード。16進数の16ビットで表現。このコードでボードのサプライヤを認識します。
OEM parameter (I/O board) I/Oボードの OEM パラメータ	I/Oボードのパラメータ。I/Oボードのドライバに必要な情報で、I/O接続時に設定する必要があります。定数や変数などが割り当てられます。
Operand (IL) オペランド(IL)	IL命令で処理される変数や定数式。
Operation (IL) 命令(IL)	IL言語の基本インストラクション。通常、1つの命令と1つのオペランドで1組になります。
Output 出力	変数の属性。ターゲットシステムの出力変数に割り当てられます。
Parameter (C function) C言語ファンクションのパラメータ	C言語ファンクションに与えられる入力値。各パラメータには型があります。
Parameter (I/O board) I/Oボードのパラメータ	I/Oボードのユーザ定義あるいは定数のパラメータ。ユーザ定義パラメータはI/O接続時に定義します。
Parent program 親プログラム	SFC以外のプログラム（サブプログラム）をコールするプログラム。
Power rail 母線	ラダープログラムでの左右の端にある母線。

用語集

Program プログラム	プロジェクト内のプログラムの基本単位。1個のプログラムは1つの言語で記述され、プロジェクトの階層構造の中に配置されます。
Project プロジェクト	1個のISaGRAFアプリケーションで、必要な全情報(プログラム、変数、ターゲットコードなど)を含んでいます。
Pulse action パルスアクション	SFCのアクション。対応するSFCステップがアクティブ状態になったときのみ1回だけ実行される一連のアクション。
Range 範囲	オブジェクトの通用範囲。ISaGRAFでの定義済みの範囲には共通、グローバル、ローカルの3つがあります。
Real 実数型	アナログ変数のクラス。浮動小数点で表現される IEEE 単精度32ビットフォーマットを持ちます。
Real board 実ボード	ターゲットシステムでI/Oデバイスに物理的に接続されたI/Oボード。
Real time mode リアルタイムモード	ターゲットシステムの通常実行モード。ターゲットのサイクルは指定のサイクルタイム毎に実行されます。
Reference number (SFC) リファレンス番号(SFC)	SFCステップやトランジションを識別する番号。1～65535が扱えます。
Register (IL) レジスタ(IL)	ILシーケンスにおけるILレジスタ内の現在結果。
Return value of a sub-program サブプログラムの戻り値	サブプログラムの実行の終わりに戻される値。戻り値は親プログラムで使われます。
Run time error ランタイムエラー	ISaGRAFターゲットシステムが実行中に検出するアプリケーションエラー。
Section セクション	プログラムのグループ。同じルールで実行されるプログラム群。
Separator セパレータ	テキスト言語内で使われる特別な文字で、識別子どうしを区切るためのものです。演算子を表わすものもあります。
Sequential Function Chart シーケンシャルファンクション チャート	ステップとトランジションをリンクしてでプロセスを記述するグラフィック言語。ステップにはアクションが割り当てられ、トランジションにはブール条件が割り当てられます。
Sequential section シーケンシャルセクション	プロジェクトを構成するプログラムのグループ。プログラムの実行方法はSFC言語やFC言語のダイナミックルールに従います。

SFC	シーケンシャルファンクションチャート(Sequential Function Chart)言語
ST	構造化テキスト(Structured Text)言語
Statement ステートメント	基本的なST言語のオペレーション
Step ステップ	SFCの基本コンポーネント。ステップは対象プロセスの1つの状態を示します。正方形で表わされます。リファレンス番号で参照されます。ステップのアクティブ状態によって対応するアクションの実行がコントロールされます。
String 文字列	メッセージ変数を構成する文字の集まり。
Structured Text 構造化テキスト	構造化テキスト言語。変数への代入、If/Then/Else などの高度な構造化、ファンクションの数コールなどをサポートしています。
Sub-program サブプログラム	SFC以外で記述されたプログラムで別の親プログラムから呼び出されます。
Target ターゲット	ISaGRAFワークベンチで作られたアプリケーションの実行環境。ISaGRAF制御カーネルソフトがワークベンチからのアプリケーションコード(中間コード)を解釈、実行しています。コンパイル型ターゲットもあります。
Target cycle ターゲットサイクル	サイクルタイム毎に実行するターゲットの処理サイクル。指定のサイクルタイムごとにトリガがかかります。
Technical note 技術メモ	ISaGRAFライブラリの各エレメント(C言語ファンクション、ファンクションブロック、変換関数、I/Oボード)に対するユーザガイド。エレメントの設計者が記述します。
Test(FC) テスト(FC)	デジジョンともいいます。フローチャートの条件判断を意味するシンボルで、論理式を割り付けます。フローはYES/NOの出力から接続されます。
Timer タイマ型	変数の型。時間の値を持っていて、ランタイム時に自動的に更新されます。
Token (SFC) トークン(SFC)	SFCプログラムのアクティブステップを示す印。灰色の円で表現されます。
Toolbox ツールボックス	グラフィカルエディタのウィンドウで使われる各種シンボルの選択をアイコン化してまとめたもの。

用語集

Top level program トップレベルプログラム	プロジェクトを構成するプログラム階層の最上位にあるプログラム。トップレベルプログラムはシステムが起動します。
Transition トランジション	SFCの基本コンポーネント。SFCステップ間の遷移条件になります。リファレンス番号で参照されます。各トランジションにはブール条件が割り当てられます。
Type 型／タイプ	同じフォーマットを持つ変数のクラス。4つの基本タイプ(ブール型、整数／実数型、タイマ型、可変長文字列型)があります。
Validity of a transition トランジションの評価	SFCトランジションの属性。直前のステップがアクティブの時、トランジションは毎サイクル、評価されます。
Variable 変数	プロジェクト内のプログラムで扱われる基本データ。
Virtual board バーチャルボード	ターゲット上でI/Oデバイスと物理的に接続されていないI/Oボード。(動作確認の時にリアルボードからバーチャルボードへ切り替えることができます。)

参考：IEC 61131-3 関連用語集（日本語－英語対応）

分類	日本語訳	英語
SFC要素		
	イニシャルステップ	initial Step
	ステップ	Step
	トランジション	Transition
	リンク	Link
	ジャンプ	Jump
	アクション	Action
	コメント	Comment
	選択分岐	divergence of sequence selection
	並列分岐	simultaneous sequence-divergence
	選択結合	convergence sequence selection
	並列結合	simultaneous sequence-convergence
	単一シーケンス	single sequence
	マクロステップ	macro step
	アクションクオリファイア	action qualifier
	P(パルス)	P
	N(ノーマル)	N
	S(セット)	S
	R(リセット)	R
	活性ステップ	active step
	非活性ステップ	inactive step
	活性	Active
	非活性	Inactive
ラダー		
	a接点	normally open contact
	b接点	normally close contact
	立ち上がり接点	positive transition sequence contacts
	立ち下がり接点	negative transition sequence contacts
	コイル	Coil
	反転コイル	negated coil
	セット(ラッチ)コイル	SET(latch) coil
	リセット(ラッチ解除)コイル	RESET(unlatch) coil
	左母線	Left power rail
	右母線	Right Power rail

用語集

	無条件ジャンプ	Unconditional Jump
	条件ジャンプ	conditional Jump
	立ち上がりコイル	Positive transition sensing coil
	立ち下がりコイル	Negative transition sensing coil
	ラベル	LABEL , label
	回路	Network
	ファンクションブロック	Function Block
FB		
	MOVE(転送)	MOVE
	ABS(絶対値)	ABS
	ACOS(アークコサイン)	ACOS
	ADD(加算)	ADD
	AND(論理積)	AND
	ARRAY(配列)	ARRAY
	ASIN(アークサイン)	ASIN
	ATAN(アークタンジェント)	ATAN
	TO(型変換ファンクション)	*_TO_*
	比較	Compare
	COS(コサイン)	COS
	CTD(ダウンカウンタ)	CTD
	CTU(アップカウンタ)	CTU
	CTUD(アップダウンカウンタ)	CTUD
	DELETE(文字列削除)	DELETE
	微分	
	DIW(除算)	DIV
	EXP(自然指数)	EXP
	EXPT(指数関数)	EXPT
	立ち下がりエッジ検出	F_TRIG
	立ち上がりエッジ検出	R_TRIG
	FIND(文字列検索)	FIND
	GE	GE
	GT	GT
	INSERT(文字列挿入)	INSERT
	積分	
	EQ	EQ
	NE	NE
	LEFT(左側文字列抽出)	LEFT
	LE	LE

	LT	LT
	LIMIT(上下限)	LIMIT
	LOG(常用対数)	LOG
	LN(自然対数)	LN
	MAX(最大値)	MAX
	MID(文字列抽出)	MID
	MIN(最小値)	MIN
	LEN(文字列長)	LEN
	MOD(剰余算)	MOD
	MUL(乗算)	MUL
	MUX(マルチプレクサ)	MUX
	NOT(論理否定)	NOT
	OR(論理和)	OR
	PID	PID
	REPLACE(文字列置換)	REPLACE
	RIGHT(右側文字列抽出)	RIGHT
	ROL(左回転)	ROL
	ROR(右回転)	ROR
	RS(リセット優先双安定)	RS
	RTC(リアルタイムクロック)	RTC
	SR(セット優先双安定)	SR
	SEL(バイナリ選択)	SEL
	セマフォ	SEMA
	SHL(左シフト)	SHL
	SHR(右シフト)	SHR
	SIN(サイン)	SIN
	SQRT(平方根)	SQRT
	SUB(減算)	SUB
	TAN(タンジェント)	TAN
	TOF(オフディレータイマ)	TOF
	TON(オンディレータイマ)	TON
	TP(パルスタイマ)	TP
	TRUNC(小数点以下切り捨て)	TRUNC
	XOR(排他的論理和)	XOR
	XORN(排他的論理和否定)	XORN
プログラム言語 & プログラム用語		
	命令リスト[言語]	instruction list language
	IL	IL

用語集

	構造化テキスト[言語]	structured text language
	ST	ST
	ファンクションブロック図[言語]	function block diagram language
	FBD	FBD
	シーケンシャルファンクションチャート	sequential function chart
	SFC	SFC
	ラダー図[言語]	ladder diagram language
	LD	LD
	CAL(コール)	CAL
	型	Type
	TYPE(型)	TYPE
	タスク	Task
	TASK(タスク)	TASK
	変数	Variable
	識別子	Identifier
	宣言	Declaration
	初期値	initial value
	コンフィグレーション	Configuraion
	入力パラメータ	
	出力パラメータ	
	POU(プログラム構成単位)	
	プログラム構成単位	program organization unit
	プログラム型	program type
	プログラム	Program
	ファンクションの型	function type
	ファンクション	Function
	ユニット、単位	Unit
	ファンクションブロック	function block
	FB(ファンクションブロック)	FB
	呼び出し	
	ファンクションブロックのインスタンス	function block instance
	ファンクションブロックダイアグラム	function block diagram
変数型		
	接点	Contact
	基本データ型	elementary data types
	データ型	deta type
	ARRAY(配列)	ARRAY
	BOOL(ブール型)	BOOL

	INT(整数型)	INT
	DINT(倍精度整数型)	DINT
	REAL(実数型)	REAL
	TIME(タイマ型)	TIME
	DT(日付時刻型)	DATE AND TIME,DT
	WORD(ワード型)	WORD
	DWORD(ダブルワード型)	DWORD
	STRING(可変長文字列型)	STRING
	DATE(日付型)	DATE
	直接表現変数	directly represented variables
	変数宣言	variable declaration
	VAR(ローカル変数宣言)	VAR
	VAR_INPUT(入力変数宣言)	
	VAR_OUTPUT(出力変数宣言)	
	VAR_EXTERNAL(外部参照変数宣言)	
	VAR_RETAIN(保持変数宣言)	
	VAR_GLOBAL(グローバル変数宣言)	
	ボディ	Body

E. 索引

-, B-305
 \$ シーケンス, B-235
 %, A-123, B-236
 &, B-302
 *, B-306
 /, B-307
:=, A-184
 := (ST), B-282
 +, B-304
 <, B-310
 <=, B-311
 <>, B-314
 =, B-313
 =1, B-304
 >, B-312
 >=, B-313
 >=1, B-303
 1 gain, B-301
1 gain (代入), B-301
 10進数, B-234
 16進数表示, C-427
 1サイクル実行, A-147
 ABS, B-342
 Acknowledge, C-425
 ACOS, B-346
 Adding licensing, A-17
 ANA, B-316
 AND, B-302
 AND_MASK, B-307
ANYTARGET, A-137
 Application エラー, C-479
 ARCREATE, B-369
 ARREAD, B-370
 ARWRITE, B-370
 ASCII, B-360
 ATAN, B-347
 AVERAGE, B-334
 a接点, B-267
 Begin, B-256
 Begin セクション, B-227, D-488
BINARYFILE, A-136
 BLINK, B-339
 BOO, B-315

BY, B-286
 b接点, B-267
 CAL 命令 (IL), B-299
 CASE, B-284
 CAT, B-319
 CHAR, B-360
 CMP, B-332
 COS, B-348
 CTD, B-327
 CTU, B-326
 CTUD, B-328
Cycle, A-186
 Cソースコード, A-133, A-194, C-433,
 C-458, D-488
 Cソースヘッダ, A-194, C-432, C-438,
 C-447, C-458, D-489
 C言語, A-217, C-432, C-433, C-438,
 C-447, C-448, C-458, D-488
 C言語コンパイラ, C-429, C-458
 C言語コンパイラの例, C-463
 C言語ファンクション, A-201, A-206, C-
 435, D-488
 C言語ファンクションインタフェース, C-
 438
 C言語ファンクションのソースコード, C-
 439
 C言語ファンクションの制限, C-441, C-
 455
 C言語ファンクションブロック, A-201, A-
 206, C-442
 C言語ファンクションブロックインタフェー
 ス, C-445, C-447
 C言語ファンクションブロックソースコード
 , C-448
 C言語ファンクションブロックのコール,
 C-444, C-454
 C言語プログラミング, C-429
 DAY_TIME, B-368
 DDE, A-158, C-420
 DDE (NT ターゲット), C-426
 DDEアドバイスループ, C-421, C-425
 DDEリクエスト, C-421
 DELETE, B-361

-
- DERIVATE, B-338
 - disable 状態のトランジション, B-254
 - DO, B-284, B-286
 - DOS ターゲット, C-385
 - ELSE, B-283, B-284
 - ELSIF, B-283
 - enable 状態のトランジション, B-254
 - End, A-188**
 - END_CASE, B-284
 - END_FOR, B-286
 - END_IF, B-283
 - END_REPEAT, B-285
 - END_WHILE, B-284
 - End セクション, B-227, D-490
 - ENO 出力, A-75
 - EN 入力, A-75
 - EQ, B-313
 - EXIT, B-286
 - EXPT, B-342
 - EZS, A-171
 - F_CLOSE, B-373
 - F_EOF, B-373
 - F_OPEN, B-371
 - F_TRIG, B-325
 - F_WOPEN, B-372
 - FA_READ, B-375
 - FA_WRITE, B-376
 - FALSE, A-110, A-150
 - FBD, A-36, A-84, A-106, B-227, B-231, B-261, C-436, C-444, D-491
 - FBD/LD エディタ, A-84
 - FBD エディタ, A-99
 - FBD シンボルの選択, A-87
 - FBD 実行順, A-90
 - FC, A-36, A-64, B-227, B-231, D-491
 - FC I/O 操作ブロック, B-258
 - FC エディタ, A-64
 - FC エLEMENTの選択, A-66
 - FC オブジェクトの移動, A-67
 - FC オブジェクトの挿入, A-66
 - FC コネクタ, A-67
 - FC コメント, A-67
 - FC サブプログラム, A-35, B-258
 - FC のサイズ変更, A-68
 - FC リンク, A-67
 - FIND, B-362
 - FM_READ, B-378
 - FM_WRITE, B-380
 - FOR, B-286
 - FROM, A-137**
 - GE, B-313
 - GFREEZE, B-228, B-255, B-290
 - GKILL, B-228, B-255, B-290
 - Goto, A-187**
 - GRST, B-228, B-255, B-291
 - GSTART, B-228, B-255, B-289
 - GSTATUS, B-255, B-291
 - GT, B-312
 - Hardware key, A-16
 - Hexadecimal values, C-427
 - HYSTER, B-335
 - I/O, A-45, A-121, A-123, A-142
 - I/Oチャンネル, A-120, **A-177**, D-491
 - I/Oチャンネルの OPERATE, B-321
 - I/Oチャンネルのコメント, A-122
 - I/Oのロック, A-148
 - I/Oボード, A-120, A-121, **A-177**, A-196, A-206, A-214, D-491
 - I/Oボードタイプ, A-121
 - I/Oボードの移動, A-121
 - I/O構成, A-27, A-195, A-206
 - I/O 接続, A-120
 - I/O接続, A-45, D-491
 - I/O装置機器, A-196, A-206
 - I/O変数, A-120, A-148, **A-177**, A-214, C-430, C-431, D-491
 - If, A-187**
 - IF, B-259, B-283
 - IL, A-36, A-96, A-161, B-227, B-231, B-251, B-253, B-293, D-491
 - IL エディタ, A-99
 - ILレジスタ, B-293
 - INSERT, B-363
 - INTEGRAL, B-337
 - ISAGRAF.INI(NT ターゲット), C-414
 - JMP, B-296
 - LD, A-36, A-61, A-71, A-73, A-84, B-227, B-231, B-265, B-295, D-492
 - LE, B-311
 - LEFT, B-364
 - Licensing, A-16
 - LIM_ALARM, B-336
 - LIMIT, B-354
 - LOG, B-343
 - LT, B-310
 - MAX, B-353
-

MID, B-365
MIN, B-353
MLen, B-365
MOD, B-355
Modbus, D-493
MODBUS, A-114
MODBUS 1 ビット書き込み, C-468
MODBUS 1 ワード書き込み, C-468
MODBUS N ビット読み出し, C-467
MODBUS N ワード書き込み, C-468
MODBUS N ワード読み出し, C-468
MODBUS スレーブ, C-465
MODBUS データ書き込み, C-469
MODBUS データ読み出し, C-470
MODBUS ファイル転送, C-469
MODBUS ファンクションコード, C-466
MODBUS プロトコル, C-465
MODBUS マスタ, C-465
MSG, B-318
MUX4, B-356
MUX8, B-357
NE, B-314
NEG, B-301
NOT, A-87, A-88
NOT_MASK, B-310
NT ターゲット, C-414
NT ターゲットアイコン, C-425
NT ターゲットの Message メニュー, C-425
NT ターゲットの Option メニュー, C-424
NT ターゲットの Start/Stop ボタン, C-426
NT ターゲットの View/Information コマンド, C-426
NT ターゲットのメインウィンドウ, C-424
ODD, B-358
OEMキーコード, A-196, A-197, D-493
OEM/パラメータ(I/Oボード), A-197, D-493
OF, B-284
OPERATE, B-321
OR, B-303
OR_MASK, B-308
OR 接続, A-85
OS-9 ターゲット, C-391
P0 アクションクオリファイア, A-60
P1 アクションクオリファイア, A-60
POW, B-344

Print, A-185**PrintTime, A-185**

Priority, C-425
Program エラー, C-479
Quick LD, A-61, A-71, A-73
Quick LD エディタ, A-99
R (リセット), B-296
R_TRIG, B-324
RAND, B-358
REAL, B-317
REDGE, B-280
REPEAT, B-259, B-285
REPLACE, B-366
RET, B-297
RETURN, B-272, B-282
RIGHT, B-367
ROL, B-350
ROR, B-350
RS, B-323
RS232C, A-48
Rung comment, A-81, A-91
S(セット), B-295
SEL, B-359
SEMA, B-326
SFC, A-36, A-51, A-132, A-150, A-209, B-227, B-231, B-241, C-436, D-495
SFCアクション, A-61, B-249
SFC エディタ, A-99
SFC ギャラリー, A-62
SFC シンボルの移動, A-57
SFCTークン, B-241
SFCの動作の原則, B-254
SFCレベル1, A-52, B-241, B-242, D-492
SFCレベル2, A-52, A-58, B-247, D-492
SFC親プログラム, B-254
Shellプロンプト, C-400
SHL, B-351
SHR, B-352
SIG_GEN, B-339
Simulate I/O, C-425, C-427
SIN, B-348
SlavesLink, C-408
Software key, A-16
Spawn, C-407
SpotLight, A-163

- SQRT, B-344
 SR, B-322
 SSR グローバル変数, C-411
 ST, A-36, A-61, A-96, A-161, B-227,
 B-231, B-277, B-295, C-436, C-444,
 D-495
 STACKINT, B-333
 Step, D-495
 ST エディタ, A-99
 SYSTEM, B-320
 System エラー, C-479
 SYSTEM ファンクション, C-474
 TAN, B-349
TARGET, A-137
TEXTFILE, A-136
 THEN, B-283
 TICコード, A-129, A-130
 TMR, B-318
TO, A-138, B-286
 TOF, B-330
 TON, B-330
 TP, B-331
 TRUE, A-110, A-150
 TRUNC, B-345
 tst_main_ex, C-407
 TSTART, B-288
 TSTOP, B-289
ULONGDATA, A-135
 UNTIL, B-285
VARLIST, A-136
 VxWorks ターゲット, C-402
Wait, A-186
 WHILE, B-259, B-284
 XOR, B-304
 XOR_MASK, B-309
 アーカイブ, A-31
 アーカイブディレクトリ, A-205
 アークコサイン, B-346
 アークタンジェント, B-347
 アイコン, A-15, A-164
 アクション, A-64, A-69, B-247, B-251,
 B-257, B-258, D-488
 アクション(FC), D-488
 アクセス権, A-211
 アスキーコード→文字変換, B-360
 アスキー変換, B-360
 アップカウンタ, B-326
 アップダウンカウンタ, B-328
 アップロード, A-170
 アップロード(オプション), A-172
 アップロードの準備, A-171
 アナログ, A-111, **A-178**, B-233, B-234,
 C-430, C-431, D-488
 アプリケーションコード, C-387, C-397,
 C-410, C-419
 アプリケーションコードサイズ, C-390
 アプリケーションコードの生成, A-42, A-
 129
 アプリケーションのスタート, A-146, **A-
 177**
 アプリケーションのストップ, A-146, **A-
 177**
 イーサネット, A-49
 イーサネット通信, C-394, C-415
 イニシャルステップ, B-242, B-254, D-
 491
 インスタンス, A-110
 インストール, A-12
 インストラクション, B-293, D-492
 インストラクションリスト, B-293, D-492
 インターフェース, A-38
 インポート, A-41, A-115
エクスポート, A-42, A-115
 エッジ, D-490
 エラー, A-134
 エラーの説明, C-480
エラーリスト, C-478
 エラー処理, C-388, C-399, C-412, C-
 422
 エリアス, A-81, D-488
 オプション(シミュレーション), A-180
 オプション(コード生成), A-130
 オフディレイタイム, B-330
 オペランド(IL), B-293, B-294, D-493
 オペレーション(IL), B-294
 オペレータ(IL), B-293, B-294
 オンディレイタイム, B-330
 オンライン, A-47, A-144
 オンライン修正, A-147, **A-153**
 カーネルスレッド, C-414
 キーワード, B-235, D-492
 クイック変数登録, A-113
 グラフィック, A-163, A-168
 グラフィックスのサイズ変更, A-165
 グラフィックス移動, A-165
 グラフィックス選択, A-165

- グリッド, A-76
クリップボード, A-96, A-109
グループ化, A-165
グループ解除, A-165
グローバル, A-104, A-105, **A-140**, B-232, B-235, D-491
クロスリファレンス, A-45, **A-140**, D-489
コイル, A-74, A-86, B-267, D-489
コイルタイプ, A-79
コイルの挿入, A-77
コード, A-130
コード生成, A-129
コーナ, A-87
コサイン, B-348
コネクタ, A-67, B-259, D-489
コピー(FBD), A-89
コピー(FC), A-69
コピー(LD), A-79, A-80
コピー(SFC), A-57
コピー(テキストエディタ), A-96
コピー(ライブラリエレメント), A-191
コピー(変数), A-109
コメント, D-489
コメント(FBD), A-88
コメント(FC), B-259
コメント(IL), B-293
コメント(SFC), B-241, B-242, D-489
コメント(ST), B-277
コンパイラ, A-129
コンパイラオプション, A-44, A-171
コンパイルメッセージ, A-134
サイクリック, B-227, D-490
サイクル, B-231
サイクルタイム, A-43, A-148, A-149, A-181, B-320, C-400, C-404, C-407, C-412, C-423, C-431, C-435, C-443, D-490
サイクルのトリガ, A-43
サイクルプロファイラ, A-181
サイクルモード, A-43, A-44, A-145, A-147, D-490
サイン, B-348
サブプログラム, A-34, B-227, B-229, B-250, B-253, B-258, B-263, B-278, B-298, D-495
シーケンシャル, A-51, B-227, B-241
シーケンシャルセクション, B-227, D-494
シーケンシャルファンクションチャート, B-241, D-494
システムクロック, C-389, C-403, C-422
システムリソースマネージャ, C-402
シミュレーション, A-46, A-144
シミュレーションI/O, C-417
シミュレータ, A-130, **A-177**, A-181, A-182, C-431, C-435, C-443
シミュレータウィンドウ, C-427
ジャンプ, A-57, A-69, A-75, A-86, A-97, B-262, B-273
シリアルリンクデバイス, C-400
シリアル通信, A-48
シリアル通信の設定, C-416
シングルタスク, C-391, C-403
シンボル, A-219
シンボルテーブル, A-219, C-388, C-398
ズーム, A-71, A-80, A-91
スクリプト, A-182, A-184
スタートモード, A-44
スタイル, A-93, A-166
スタイル/セクションの選択, A-37
ステートメント, B-277, D-495
ステップ, A-58, A-150, A-151, B-241, D-495
ステップの活性化状態, B-241, B-242, B-254, B-287, D-488
ステップの活性状態持続時間, B-242, B-287
ステップへのジャンプ, A-53, B-243, D-492
ストラクチャードテキスト, B-277
ストリング, B-234
スパイ, A-159, A-161, A-163
スレーブ番号, A-47, C-386, C-392, C-393, C-394, C-403, C-406, C-415, C-424, C-426
スレッド起動, C-407
スロット, A-121, A-124
スロットのクリア, A-121
スロットの挿入, A-121
セクション, A-33, B-227, D-494
セットコイル, B-270
セット優先双安定, B-322
セパレータ, B-277, D-494
セパレータ(プロジェクト), A-26
セマフォ, B-326

- セル, A-55, D-489
- ターゲット, A-130, D-495
- ターゲットサイクル, D-495
- ターミナルモード, C-400**
- タイプ, A-104, A-120, **A-141**, B-233, D-496
- タイマ, B-234, B-238
- タイマ型, A-110, A-111, D-495
- タイマ型変換, B-318
- タイムアウト, A-47
- ダウンカウンタ, B-327
- ダウンロード, A-146, **A-177**
- タスクの起動, C-411
- タッチ, A-42
- タンジェント, B-349
- ダンブ, A-160
- チャイルドSFCプログラム, A-35, B-228, D-489
- チャネル, A-122, A-123, A-124
- ツールボックス, D-495
- ツールメニュー, A-45
- ディスク, A-13
- ディレクトリ構成, A-217
- テキストエディタ, A-96
- テキスト表示, A-164
- デシジョン, A-68, A-69, B-257, D-490
- テスト, A-64, A-68, A-69, B-257
- デバッグ, A-144, A-175, **A-177**
- デバッグ オプション, A-148
- デバッグ, A-47
- デバッグ用ワークスペース, A-47
- トークン (SFC), D-495
- ドキュメント印刷, A-28, A-45
- トップレベルプログラム, A-33, D-496
- トランジション, A-58, A-150, A-151, B-242, D-496
- トランジションの通過, A-152**, B-254, D-489
- トランジションの評価, D-496
- トランジション条件, B-252
- トランジション条件, D-489
- トレンドグラフ, A-163, A-164, A-168
- ネットワークアドレス, A-106, A-109, A-114, D-493
- ノーマルアクション, A-59, B-249, D-493
- ノンストアード, A-59
- ノンストアードアクション, B-249
- バージョンNo. 取得, A-146
- バーチャルI/Oボード, C-417
- バーチャルアドレス, A-219
- バーチャルボード, A-122, **A-177**, A-214, D-496
- バイナリ選択, B-359
- パスワード, A-29, A-124, A-192
- バックアップ, A-31, A-205
- バックグラウンドピクチャ, A-163
- パラメータ(C言語ファンクション), A-192, A-201, D-493
- パラメータ(I/Oボード), A-192, A-197, D-493
- パラメータ(ファンクションブロック), A-192, A-201
- パリティ, A-48
- パルスアクション, A-59, B-248, D-494
- パルスタイマ, B-331
- ヒステリシス, B-335, B-336
- ビットマップ, A-163
- ビット展開, A-165
- ビット毎の排他的論理和, B-309
- ビット毎の論理積, B-307
- ビット毎の論理和, B-308
- ファイルのオープン, B-371, B-372
- ファイルのクローズ, B-373
- ファイルの書き込み, B-376, B-380
- ファイルの読み込み, B-375, B-378
- ファイル終端の検出, B-373
- ファンクション, A-41, A-199, A-206
- ファンクションセクション, A-34, B-227
- ファンクションのパラメータ, A-38
- ファンクションブロック, A-41, A-74, A-86, A-90, A-105, A-110, A-199, A-206, B-230, B-261, B-279, D-491
- ファンクションブロックインスタンス, A-113, C-443
- ファンクションブロックセクション, A-34, B-227
- ファンクションブロックダイアグラム, B-261, D-491
- ファンクションブロックのコール(IL), B-299
- ファンクションブロックのスタティックデータ, C-449
- ファンクションブロックのパラメータ, A-38
- ブールアクション, A-60, B-247, D-488
- ブール型, A-110, B-237, D-488
- ブール型変換, B-315

- フオント, A-209
プライオリティ, C-425
プライオリティ設定, C-418
ブリンク, B-339
ブレイクポイント, A-145, A-148, A-150,
D-488
フロー, A-67
フローチャート, A-64, D-491
フローチャートエディタ, A-64
フローリンク, B-256, B-259, B-260
プログラミング言語, A-36
プログラム, A-33, A-99, A-181, B-227,
D-494
プログラムのコピー, A-40
プログラムのコメント, A-37
プログラムの移動, A-39
プログラムの削除, A-41
プログラムの文法, A-99
プログラムの名前変更, A-39
プログラムユニット, A-33
プログラムを開く, A-38, A-143
プログラム管理, A-33
プログラム文法, A-130
プロジェクト, A-204, B-227, D-494
プロジェクトグループ, A-30
プロジェクトの移動, A-27
プロジェクトの記述, A-26
プロジェクトの編集, A-27
プロジェクトリスト, A-26
プロジェクト管理, A-26
プロジェクト記述, A-27, A-45
ブロック, A-220
プロテクション, A-124
プロテクションレベル, A-211
ページ, A-209
ベース, B-233
べき乗数, B-344
ベリファイ, A-42
ボード, A-120
ボードの検証, A-122
ボードパラメータ, A-122
ボーレート, A-48
マウスの右ボタン, A-214
マクロステップ, A-54, A-56, B-246, D-
492
マクロステップボディ, B-246
マトリックス, D-492
マルチタスク, C-392, C-405
マルチプレクサ, B-356, B-357
メタファイル, A-163
メッセージ, A-178, B-234
メモリ, A-12
メモリスペース, C-398
メモリモジュール, C-398
より小さい, B-310
より大きい, B-312
ライブラリ, A-31, A-41, A-121, A-122,
A-141, A-178, **A-190**, A-204, D-492
ライブラリ管理, A-190, C-429, C-431,
C-435, C-444
ラダーダイアグラム, B-265, D-492
ラベル, A-86, B-262, B-273
ラベル (IL), B-293, D-492
**ラベル (シミュレーションスクリプト), A-
187**
ラング, A-73, A-75, A-80, A-88
ラングコメント, A-76, A-81
ラングの挿入, A-79
ラングラベル, A-77
ランタイム, A-43
ランタイムエラー, A-43, A-44, A-144,
A-149, B-320, D-494
ランダム値, B-358
リアルタイムモード, A-43, A-44, A-145,
A-147, D-494
リセットコイル, B-270
リセット優先双安定, B-323
リソース, A-44, A-134
リソース定義ファイル, A-135
リターン, A-75, A-87
リファレンス番号, B-241, B-242, B-246,
D-494
リミットアラーム, B-336
リンク, A-47, A-87, A-88, A-89, A-171
リンク (SFC), B-243
リンク設定, A-47
レコード, A-220
レジスタ (IL), D-494
レベル2, A-58, A-69
ローカル, A-104, A-105, **A-140**, B-235,
D-492
ロック, D-492
ロックされたI/O, A-150, A-214
ワークベンチの制限, A-223
圧縮, A-205
圧縮ソース (EVS), A-171

- 移動(FBD), A-88
 移動(LD), A-88
 移動平均, B-334
 印刷, A-28, A-45, A-209
 印刷(プログラム), A-101
 印刷履歴, A-207
 隠れた変数, C-448
 右シフト, B-352
 右回転, B-350
 右側文字列抽出, B-367
 右母線, A-75
 加算, B-304
 可変引数の演算子(*), B-306
 可変引数の演算子(+), B-304
 可変引数の演算子(OR), B-303
 可変長文字型, D-493
 可変長文字列型, A-110, A-112
 可変長文字列型変換, B-318
 可変長文字列結合, B-319
 開始ステップ, A-54, A-56, B-246, D-488
 階層, A-33, A-39, B-227, B-254, D-491
 各ビットの反転, B-310
 括弧, B-278
 環境設定, C-402
 奇数パリティ, B-358
 技術メモ, A-122, A-192, C-430, C-431, C-436, C-444, D-495
 共通, A-105, A-204, D-489
 結合, A-52, A-55, B-244
 検索, A-57, A-69, A-80, A-90
 検索(テキストエディタ), A-97
 減算, B-305
 現在結果(IL), B-293, B-294, D-490
 言語, A-199, B-231
 更新, A-147, **A-153**
 構造化テキスト, B-277, D-495
 行の挿入／削除(FBD), A-91
 左シフト, B-351
 左回転, B-350
 左側文字列抽出, B-364
 左母線, A-73
 再計算, A-141
 最小値, B-353
 最大値, B-353
 最適化, A-130, A-132
 削除(FBD), A-89
 削除(FC), A-69
 削除(LD), A-79
 削除(SFC), A-57
 削除(テキストエディタ), A-96
 削除(ライブラリエレメント), A-191
 削除(変数), A-109
 削除済みスタイル, A-94
 指数, B-342
 時間の単位, B-234
 辞書, A-99, A-104, **A-140**, C-431, C-443, D-490
 式, D-490
 識別子, D-491
 実ボード, A-122, **A-177**, A-214, D-494
 実行サービス, C-450
 実数, B-234
 実数型, A-110, A-111, D-494
 実数型変換, B-317
 修飾子(IL), B-293, B-294, D-493
 修正済みスタイル, A-94
 修正履歴, A-27, A-38, A-45, D-490
 終了キー, C-389, C-423
 終了ステップ, A-54, A-56, B-246, D-490
 出力, A-120, A-142, B-231, B-232, D-493
 初期化サービス, C-450
 初期状態, B-242, B-254, D-491
 除算, B-307
 小さいか等しい, B-311
 小数点以下切り捨て, B-345
 上下限, B-354
 乗算, B-306
 剰余, B-355
 常用対数, B-343
 条件, B-257
 信号発生, B-339
 新しいラング, A-76
 新規ファクションの作成, A-36
 新規ファンクションブロックの作成, A-36
 新規プログラムの作成, A-36
 新規プロジェクト, A-27
 新規ライブラリエレメント, A-191
 親SFCプログラム, D-490
 親プログラム, B-229, D-493
診断, A-175
 垂直接続, A-85
 数値変換, D-489

- 数値変換テーブル, A-126, D-489
整数, B-233, D-492
整数型, A-110, A-111
整数型変換, B-316
整数値のスタック, B-333
積分, B-337
切り取り(FBD), A-89
切り取り(FC), A-69
切り取り(LD), A-79
切り取り(SFC), A-57
切り取り(テキストエディタ), A-96
切り取り(変数), A-109
接続, A-89
接続ライン, A-87, A-88
接続ライン(FBD), B-262
接続線(LD), B-265
接点, A-74, A-86, B-267, D-489
接点タイプ, A-79
接点の挿入, A-77
接点の立ちあがり, B-268
接点の立下り, B-268
絶対値, B-342
遷移, A-151
属性, D-488
他のライブラリ, A-190
代入, A-184, B-282, B-301
置換, A-57, A-69, A-80, A-90
置換(テキストエディタ), A-97
遅延オペレーション(IL), B-297
遅延操作(IL), B-294, D-490
中間コード, A-129, C-387, C-397, C-410, C-419
中間コードサイズ, C-427
直接表現変数, A-123, B-236, D-490
通信, A-47, A-145, A-149, A-217
通信スレッド, C-414
通信タスク, C-407
通信タスク論理番号, C-393, C-394, C-406
通信モジュール, C-402
通信リンクの設定, C-385, C-391, C-393, C-402, C-415, C-425, C-426
通信設定, A-171
定義ワード, A-38, A-105, A-110, A-112, A-204, B-239, D-490
定数, B-233
定数式, D-489
貼り付け(FBD), A-89
貼り付け(FC), A-69
貼り付け(LD), A-79
貼り付け(SFC), A-57
貼り付け(テキストエディタ), A-96
貼り付け(変数), A-109
電源異常時の管理, C-472
等しい, B-313
等しいかより大きい, B-313
等しくない, B-314
読み出しサービス, C-451
内部, D-492
日付と時刻, B-368
入力, A-120, A-142, A-181, B-231, B-232, D-491
排他的論理和, B-304
配列作成, B-369
配列要素の書き込み, B-370
配列要素の読み出し, B-370
反転コイル, B-269
範囲, A-104, A-105, A-106, D-494
番号取り直し, A-57, A-69
比較, B-332
微分, B-338
標準スタイル, A-94
浮動小数点, B-234
符号反転, B-301
復元, A-31, A-205
幅調整, A-82
分岐, A-52, A-55, B-244
文字列, A-112, A-160, D-495
文字列検索, B-362
文字列削除, B-361
文字列挿入, B-363
文字列置換, B-366
文字列抽出, B-365
文字列長, B-365
平方根, B-344
別のプログラム, A-100
変換, A-127
変換テーブル, A-128
変換ポイント, A-127, A-128
変換関数, A-202, A-206, C-430, D-489
変換関数の制限, C-435
変数, A-38, A-59, A-78, A-87, A-90, A-97, A-104, **A-140**, **A-141**, A-142, A-150, B-227, B-235, B-261, D-496
変数タイプ, A-106
変数のスパイ, A-159

-
- 変数のソート, A-109
 - 変数のリスト, A-161, A-166
 - 変数の印刷, A-108
 - 変数の新規作成, A-108
 - 変数の並び替え, A-109
 - 変数の編集, A-109
 - 変数リスト, A-159
 - 変数リストの保存, A-159
 - 変数辞書, A-38, A-142
 - 変数宣言, A-38, A-104
 - 変数統計, A-142
 - 変数名, B-235, C-427
 - 変数名の挿入, A-59
 - 編集箇所の記録, A-93
 - 保持変数, A-44, C-473
 - 母線, A-85, B-265, D-493
 - 棒グラフ, A-164
 - 名前の変更(ライブラリエレメント), A-191
 - 命令(IL), B-293
 - 命令リスト, B-293
 - 目次, A-207, A-208
 - 戻り値, D-494
 - 予約語, B-235, B-294
 - 立ち下がり検出, B-325
 - 立ち下がり接点, B-268
 - 立ち上がり検出, B-324
 - 立ち上がり検出コイル, B-271
 - 立ち上がり接点, B-268
 - 論理積, B-302
 - 論理値, B-237
 - 論理反転(FBD), B-263
 - 論理反転接続線, A-87, A-88
 - 論理和, B-303