

CENTUM VP

APCS ユーザカスタムブロック プログラミングガイド

IM 33J15U21-01JA

IM 33J15U21-01JA
2 版

はじめに

APCS (Advanced Process Control Station) は、制御バスを接続したコンピュータで制御機能を実行する CENTUM VP のステーションです。APCS は、高度制御／効率改善を主用途としています。APCS では、ユーザカスタムブロックと呼ばれる機能ブロックで、ユーザが C 言語で記述したアルゴリズムを実行することができます。このユーザが定義するアルゴリズムをユーザカスタムアルゴリズムと呼びます。

ユーザカスタムアルゴリズムは、Microsoft 社の Visual C++ で作成した DLL として実装します。ユーザは、Visual C++ の機能を利用して、制御アルゴリズムを C 言語で開発します。ユーザカスタムアルゴリズムを作成するため、システムが提供しているライブラリをユーザカスタムアルゴリズム作成用ライブラリと呼びます。ユーザカスタムアルゴリズム作成用ライブラリには、入出力結合端子アクセスや機能ブロックデータアクセスなどの関数が用意されています。ユーザは、これらの関数を組み合わせてプログラミングすることにより、ユーザ独自のアルゴリズムを開発することができます。

本書では、ユーザカスタムアルゴリズムのプログラミングを説明します。ユーザカスタムアルゴリズムは、C 言語で記述しますが、本書では、C 言語については説明しません。プログラム例をもとに、ユーザカスタムアルゴリズム作成用ライブラリをどのように使ってアルゴリズムを作成するかを説明します。本書で説明するユーザカスタムアルゴリズムは、サンプルプログラムとしてユーザカスタムアルゴリズム開発環境パッケージに含まれています。これらのプログラムはバーチャルテスト機能で実際に動かせます。

なお、本書に関連するドキュメントとして次のものがあります。

- APCS (IM 33J15U10-01JA)
- APCS ユーザカスタムブロック (IM 33J15U20-01JA)
- APCS ユーザカスタムブロック ライブラリ (IM 33J15U22-01JA)
- APCS ユーザカスタムブロック開発環境 (IM 33J15U23-01JA)

■ ユーザカスタムアルゴリズム作成の注意事項

ユーザカスタムアルゴリズムの作成には、C 言語に関する知識や機能ブロックの詳細機能についての知識を必要とします。

ユーザカスタムアルゴリズムは、ユーザが C 言語で記述します。ユーザプログラムの誤りに対し、システムは、つぎのような対処をしています。

- ユーザプログラムの連続した処理時間が 1 秒を超えると、システムにより当該ユーザカスタムブロックの処理が打ち切られます。
- ユーザプログラム内でアドレスエラーなどを検出すると、システムにより当該ユーザカスタムブロックのみ停止されます。

システムは以上のような処理をしていますので、通常ユーザプログラムの誤りは当該ユーザカスタムブロックのみに影響しますが、それでもユーザプログラムの誤りによっては APCS 制御機能（システム）に影響を与える可能性があります。そのため、実機の APCS でユーザカスタムアルゴリズムを稼動する前に、バーチャルテスト機能を使い、ユーザプログラムを十分にデバッグしてください。ユーザカスタムアルゴリズム作成の注意事項は以下のとおりです。

- 実機の APCS にユーザカスタムアルゴリズムをダウンロードする前に、バーチャルテスト機能でテストしてください。
- ユーザカスタムアルゴリズムで使用できる Windows の関数には制限があります。制限に従わないプログラムを稼動すると APCS 制御機能（システム）に影響を与える可能性があります。Windows の関数は、制限の範囲内でご使用ください。

安全に使用するための注意事項

■ 本製品の保護・安全および改造に関する注意

- ・本製品によって制御されるシステムおよび本製品自体を保護し、安全に操作するために、本書に記載されている安全に使用するための注意事項に従ってください。指示事項に反する扱いをされた場合、横河電機株式会社（以下、当社といいます）は安全性の保証をいたしかねます。
- ・ユーザーズマニュアルで指定していない方法で製品を使用した場合は、本製品で提供される保護機能が損なわれる可能性があります。
- ・本製品によって制御されるシステムおよび本製品そのものに保護または安全回路が必要な場合は、本製品外部に別途ご用意ください。
- ・本製品と組み合わせて使用する機器の仕様と設定については、必ず、機器の取扱説明書などで確認してください。
- ・本製品の部品または消耗品を交換する場合は、当社が指定する部品のみを使用してください。
- ・本製品および本製品の電源コードセットなどの付属品を、当社が指定する機器や用途以外に使用しないでください。
- ・本製品を改造することは、固くお断りいたします。
- ・本製品およびユーザーズマニュアルでは、安全に関する次の記号を使用しています。



「注意」を示します。本製品においては、感電など、人体への危険や機器損傷の恐れがあることを示すとともに、ユーザーズマニュアルを参照する必要があることを示します。また、ユーザーズマニュアルにおいては、人体への危険や機器損傷を避けるための注意事項が記載されている箇所に、本記号を「注意」「警告」の用語と一緒に使用しています。



「注意、高温表面」を示します。このマークの付いた機器は熱くなりますのでご注意ください。接触するとやけどなどの危険があります。



「保護導体端子」を示しています。感電防止のため、本製品を使用する前に、保護導体端子を必ず接地してください。



「機能接地端子」を示しています。「FG」と表示された端子も同じ機能を備えています。保護接地以外を目的とした接地端子です。本製品を使用する前に、機能接地端子を必ず接地してください。



「AC 電源」を示します。



「DC 電源」を示します。



「オン」を示します。電源スイッチなどの状態を示します。



「オフ」を示します。電源スイッチなどの状態を示します。

■ ユーザーズマニュアルに対する注意

- ・ ユーザーズマニュアルは、最終ユーザまでお届けいただき、最終ユーザがお手元に保管して隨時参照できるようにしてください。
- ・ ユーザーズマニュアルをよく読んで、内容を理解したのちに本製品を操作してください。
- ・ ユーザーズマニュアルは、本製品に含まれる機能詳細を説明するものであり、お客様の特定目的に適合することを保証するものではありません。
- ・ ユーザーズマニュアルの内容については、将来予告なしに変更することがあります。
- ・ ユーザーズマニュアルの内容について万全を期していますが、もしご不審な点や誤り、記載もれなどお気付きのことがありましたら、当社またはお買い求め先代理店までご連絡ください。乱丁、落丁はお取り替えいたします。

■ 本製品の免責について

- ・ 当社は、保証条項に定める場合を除き、本製品に関するいかなる保証も行いません。
- ・ 本製品のご使用または使用不能から生じる間接損害については、当社は一切責任を負いかねますのでご了承ください。

■ ソフトウェア製品について

- ・ 当社は、保証条項に定める場合を除き、本ソフトウェアに関するいかなる保証も行いません。
- ・ 本製品の各ソフトウェアに対するライセンスは、ご使用になるコンピュータの台数に応じて適正にご購入ください。
- ・ バックアップ以外の目的で本ソフトウェアを複製することは、当社の知的所有権を侵害する行為であり、固くお断りいたします。
- ・ 本ソフトウェアが収められているソフトウェアメディアは、大切に保管してください。
- ・ 本ソフトウェアをリバースコンパイル、リバースアセンブリ、リバースエンジニアリング、その他の方法により人間が読み取り可能な形にすることは、固くお断りします。
- ・ 当社から事前の書面による承認を得ることなく、本ソフトウェアの全部または一部を譲渡、交換、転貸などによって第三者に使用させることは、固くお断りいたします。

ユーザーズマニュアル中の凡例

■ ユーザーズマニュアル中のシンボルマーク

ユーザーズマニュアルの本文中では、次の各種記号が使用されています。



警告

死亡または重傷を招く可能性がある危険な状況を避けるための注意事項を記載しています。



注意

軽傷または物的損害を招く可能性がある危険な状況を避けるための注意事項を記載しています。

重要

操作や機能を知る上で、注意すべき事柄を記載しています。

補足

説明を補足するための事柄を記載しています。

参照

参照先を示します。

オンラインマニュアルでは、緑色の参照先をクリックすると、該当箇所が表示されます。黒色の参照先は、該当箇所が表示されません。

■ ユーザーズマニュアル中の表記

ユーザーズマニュアル中の表記は、次の内容を示します。

● ユーザーズマニュアル全体を通して共通に使用されている表記

入力文字列

次の書体の文字列は、ユーザが実際の操作において入力する内容を示します。

例：

FIC100.SV=50.0

▼記号

本製品のエンジニアリングを行うウィンドウの定義項目に関する説明箇所であることを示します。

本製品のエンジニアリングを行うウィンドウのヘルプメニューから「ビルダ定義項目一覧」を選択したときに開くウィンドウを経由して、選択した項目の説明を表示できます。なお、複数の定義項目が併記されている場合には、複数の定義項目に関する説明箇所であることを示します。

例：

▼ タグ名、ステーション名

△ 記号

ユーザが入力する文字列で、空白文字（スペース）を示します。

例：

.AL △ PIC010 △ -SC

{ } で囲った文字

ユーザが入力する文字列で、省略可能な文字列を示します。

例：

.PR △ TAG {△ .シート名}

● キーまたはボタン操作を示すために使用されている表記

[] で囲った文字

キーまたはボタンの操作説明において [] で囲まれている文字は、キーボードのキー、オペレーションキーボードのキー、ウィンドウに表示されるボタン名、またはウィンドウに表示されるリストボックスの選択項目のいずれかを示します。

例：

機能を切り替えるには [ESC] キーを押します。

● コマンド文やプログラム文などの書式説明の中で使用されている表記

コマンド文やプログラム文などの書式説明の中で使用されている表記は、次の内容を示します。

<>で囲った文字

ユーザが一定の規則に沿って任意に指定できる文字列を示します。

例：

```
#define <識別子> <文字列>
```

…記号

直前のコマンドや引数が繰り返し可能であることを示します。

例：

```
lmax (arg1, arg2, …)
```

[] で囲った文字

省略可能な文字列を示します。

例：

```
sysalarm <フォーマット文字列> [<出力値>…]
```

|| で囲った文字

ユーザが複数候補から任意に選択できる文字列を示します。

例：

opeguide	<フォーマット文字列> [<出力値>…]
	OG,<素子番号>

■ 図の表記

ユーザーズマニュアルに記載されている図は、説明の都合上、部分的に強調、簡略化、または省略されていることがあります。

ウィンドウの図では、機能理解や操作監視に支障を与えない範囲で、実際の表示と部品の表示位置や、大文字小文字など文字の種類が異なっている場合があります。

■ 入力文字

Windows では半角カタカナを使用できますが、本製品のソフトウェアへ入力する文字列には、半角カタカナを使用しないでください。

著作権および商標

■ 著作権

ソフトウェアメディアなどで提供されるプログラムおよびオンラインマニュアルなどの著作権は、当社に帰属します。

本製品を利用する目的でオンラインマニュアルの必要箇所をプリンタに出力することは可能ですが、全体の複製、または転載は著作権法で禁止されています。

したがって、オンラインマニュアルを電子的または上記出力を除く書面で複製したり、第三者に譲渡、販売、頒布（紙媒体、電子媒体、ネットワーク経由の配布など一切の方法を含みます）することを禁止します。また、無断でビデオ機器その他に登録、録画することも禁止します。

■ 商標

- CENTUM、ProSafe、Vnet/IP、PRM、Exaopc、Exaplog、Exapilot、Exaquantum、Exasmoc、Exarqe、Multivariable Optimizing Control/R robust Quality Estimation、StoryVIEW および FieldMate Validator は、横河電機株式会社の登録商標または商標です。
- 本製品で使用されている会社名、団体名、商品名およびロゴ等は、横河電機株式会社、各社または各団体の登録商標または商標です。

Blank Page

APCS ユーザカスタムブロック プログラミングガイド

IM 33J15U21-01JA 2 版

目 次

1.	ユーザカスタムアルゴリズム入門.....	1-1
1.1	ユーザカスタムブロックを動かす	1-2
1.2	機能ブロック定周期処理.....	1-15
1.3	機能ブロック終了処理	1-23
1.4	機能ブロック初期化処理.....	1-25
1.5	機能ブロックデータ設定時特殊処理.....	1-26
1.6	機能ブロックワンショット起動処理.....	1-31
2.	ユーザカスタムアルゴリズムと C 言語.....	2-1
2.1	データ型	2-3
2.2	システム定義インクルードファイル.....	2-10
3.	ユーザカスタムアルゴリズム	3-1
3.1	ユーザ定義関数の一般形.....	3-4
3.2	機能ブロック初期化処理.....	3-5
3.3	機能ブロック終了処理	3-11
3.3.1	オンラインの修正に伴う機能ブロック初期化処理／終了処理	3-13
3.3.2	機能ブロック初期化処理／終了処理のプログラミング	3-18
3.3.3	機能ブロック初期化処理／終了処理のデバッグ	3-19
3.4	機能ブロック定周期処理.....	3-21
3.5	機能ブロックワンショット起動処理.....	3-23
3.5.1	ビルダ定義項目「起動タイミング」.....	3-28
3.6	機能ブロックデータ設定時特殊処理.....	3-30
3.6.1	CSTM-C のブロックモードを AUT と O/S に限定	3-34
3.6.2	CSTM-C での入力上限／下限アラームの検出	3-40
3.7	ユーザ定義関数の戻り値.....	3-43
3.8	ユーザカスタムアルゴリズム作成用ライブラリー一覧	3-45
3.8.1	ユーザカスタムアルゴリズム作成用ライブラリのエラーコード	3-55
4.	ユーザカスタムアルゴリズムの開発	4-1
4.1	ユーザカスタムアルゴリズム開発のフォルダ構成.....	4-2
4.2	ユーザカスタムアルゴリズムの命名規則.....	4-4
4.3	ブロックステータスとアラームステータスの決定	4-5
4.3.1	ブロックステータス	4-6
4.3.2	アラームステータス	4-9
4.3.3	ユーザ定義インクルードファイル usrstatus.h	4-16
4.3.4	ブロックステータスの操作	4-18
4.4	ブロック形の決定	4-25
4.5	ビルダ定義項目の決定	4-27
4.6	ひな形にするユーザカスタムアルゴリズムの作成	4-34
4.7	ユーザカスタムアルゴリズムの共用	4-37
4.8	Windows のライブラリ	4-42

5.	汎用演算形ユーザカスタムブロックのプログラミング	5-1
5.1	多入力 CSTM-A	5-2
5.2	多入力多出力 CSTM-A	5-16
5.2.1	入力処理と UcaFpuExpCheck による演算エラーの検出	5-25
5.2.2	演算異常 ERRC アラームの発生と復帰	5-30
5.2.3	出力端子からのデータ出力	5-34
5.2.4	チューニングパラメータより弱い初期値の設定方法	5-36
6.	連続制御形ユーザカスタムブロックのプログラミング	6-1
6.1	多入力 CSTM-C (ブロックモードは AUT と O/S に限定)	6-3
6.1.1	UcaRWSetPv による PV のデータステータス作成	6-15
6.1.2	ユーザ定義アラームの発生と復帰	6-19
6.1.3	J01 端子と J02 端子からのデータ出力	6-24
6.1.4	機能ブロック初期化処理 (データアイテム P01 と P02 の初期化)	6-25
6.1.5	機能ブロックデータ設定時特殊処理 (AUT と O/S に限定)	6-27
6.2	CSTM-C のブロックモード遷移	6-29
6.2.1	ブロックモード遷移の UcaCtrlHandler による処理	6-41
6.2.2	前処理 (UcaCtrlHandler の内部で処理)	6-45
6.2.3	入力処理 (ユーザ記述)	6-46
6.2.4	ブロックモード遷移と設定値の作成 (UcaCtrlHandler の内部で処理)	6-48
6.2.5	制御演算処理 (ユーザ記述)	6-54
6.2.6	出力値作成処理 (UcaCtrlHandler の内部で処理)	6-57
6.2.7	出力処理 (ユーザ記述)	6-59
6.2.8	ブロックモード遷移付きの CSTM-C におけるデータの流れ	6-62
6.3	制御演算関数を使用した CSTM-C	6-63
6.3.1	制御ホールド	6-76
6.3.2	入力補償と出力補償	6-77
6.3.3	制御初期化	6-80
6.3.4	PI 制御演算 (ユーザ記述の制御演算) とレンジ変換	6-81
6.3.5	制御動作方向	6-83
6.3.6	制御出力動作	6-84
6.3.7	リセットリミット	6-87
6.3.8	不感帯動作	6-88
6.3.9	ユーザによる制御演算関数のパラメータ指定改造	6-90
6.3.10	制御演算関数呼び出しの省略	6-93
6.4	PID 調節ブロックと同じ動作をする CSTM-C	6-101
6.4.1	UcaCtrlHandler の演算処理実行ブロックモード	6-110
6.4.2	実効スキャン周期と制御周期	6-111
6.4.3	UcaCtrlPidInit による制御初期化	6-124
6.4.4	PID 制御演算	6-125
6.5	データアイテム PV とデータアイテム SV を使用しない場合	6-128
6.6	カスケード結合と実効スキャン周期	6-133
6.7	カスケードクローズとワンショット起動	6-143
6.8	制御演算の初期化と出力処理	6-165
7.	タグ名を指定したデータ入力	7-1
7.1	タグ名と入出力端子によるデータ入出力の違い	7-3
7.2	他ステーションデータ (副入力 RVnn を使用)	7-4
7.3	自ステーションデータ (副入力 RVnn を使用)	7-17
Appendix.	機能ごとの索引	App.-1

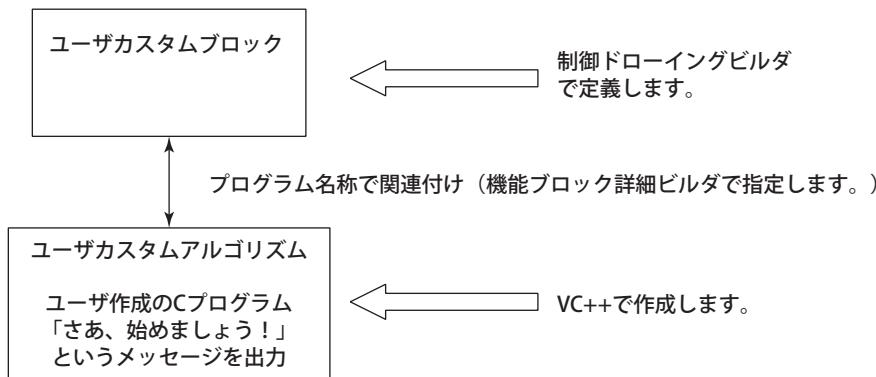
1. ユーザカスタムアルゴリズム入門

ユーザカスタムブロックは、ユーザがC言語で記述したアルゴリズムを実行する機能ブロックです。ユーザは、ユーザ独自のアルゴリズムをVisual C++（以下VC++と記述します）のDLLとして開発します。このユーザ定義のプログラムをユーザカスタムアルゴリズムと呼びます。

この章では、簡単なユーザカスタムアルゴリズムを指定したユーザカスタムブロックの動作を説明しながら、ユーザカスタムアルゴリズムをプログラミングする基本を説明します。バーチャルテスト機能を使用しユーザカスタムブロックを実行する手順を、サンプルプログラムを使用しながら説明します。

1.1 ユーザカスタムブロックを動かす

「さあ、始めましょう！」というシステムアラーム復帰メッセージを出力するユーザカスタムブロックを動かすことから始めます。ユーザカスタムブロックは、制御ドローイングビルダで定義します。また、「さあ、始めましょう！」というメッセージを出力するプログラムは、Visual C++でC言語で作成します。ユーザがC言語で作成するプログラムをユーザカスタムアルゴリズムと呼びます。機能ブロックとユーザカスタムアルゴリズムは、ユーザカスタムブロックのビルダ定義項目「プログラム名称」で関連付けをします。



010101J.ai

図 ユーザカスタムブロック

■ ユーザカスタムブロックを動かすための準備作業

最初にユーザカスタムブロックを動かすために、次の4つの準備作業が必要です。

● CENTUM VPプロジェクトの作成

システムビューでCENTUM VPプロジェクトを作成します。ここでは「LEARNUCA」という名前にします(*1)。そして、FCS0101にFCSステーション、FCS0121にAPCSステーションを定義します。本書の記述とサンプルアプリケーションは、「表 サンプルに合わせたCENTUM VPプロジェクトのステーション構成」で示すステーション構成を前提としていますので、必ず同じ構成にしてください。なお、本書ではシステムビューで操作するユーザアプリケーションをCENTUM VPプロジェクトと記述し、VC++のプロジェクトと区別します。

*1：プロジェクト名LEARNUCAは、Learn User Custom Algorithmの省略です。ユーザカスタムアルゴリズム作成を学習するためのCENTUM VPプロジェクトです。ユーザはプロジェクト名を自由に決めてください。本書のサンプルはLERANUCA以外のプロジェクト名でも動作可能です。

表 サンプルに合わせたCENTUM VPプロジェクトのステーション構成

ステーション名	機種	ステーションタイプ	データベースタイプ
FCS0101	LFCS	AFS20D 二重化フィールドコントロールユニット (キャビネット付き)	汎用
FCS0121	APCS	APCS アドバンスドプロセスコントロールステーション	汎用
HIS0164	HIS	PC 操作監視機能搭載汎用 PC	なし

LEARNUCA CENTUM VP プロジェクトに、ステーションを定義した結果を以下に示します。



010102J.ai

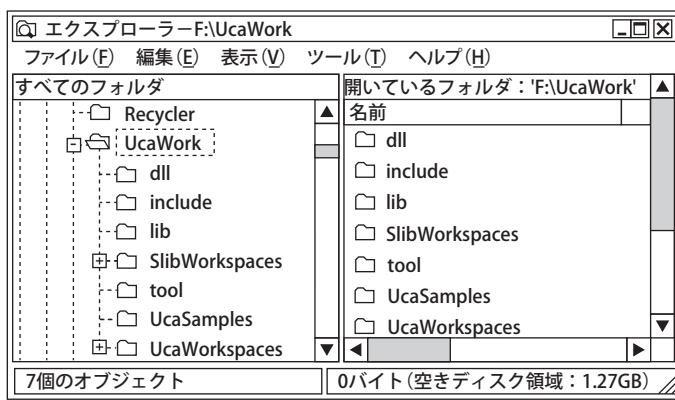
図 本書が前提とするCENTUM VPプロジェクトのステーション構成

● ユーザカスタムアルゴリズムを作成するためのフォルダ構成

ユーザカスタムアルゴリズム開発を行うコンピュータの Windows ファイルシステムに、ユーザカスタムアルゴリズム開発のためのフォルダを作成します。作業に使用するハードディスクドライブを決めます。そして、インストールされているユーザカスタムアルゴリズム開発のための以下のフォルダ構成をコピーします。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥UcaWork

CENTUM VP インストール先より、UcaWork フォルダを含めて、作業する Windows のファイルシステムに、<ドライブ名>:¥UcaWork となるようにコピーします（以下、<ドライブ名>:¥UcaWork を単に UcaWork と記述します）。



010103J.ai

図 ユーザカスタムアルゴリズム開発のためのフォルダ構成

● Visual C++のディレクトリパス

ディレクトリパスの設定に従い、ユーザカスタムブロック開発環境パッケージよりインストールされた<CENTUM VP インストール先> ¥UcaEnv¥include と作業フォルダの UcaWork¥include を「インクルードファイル」パス、<CENTUM VP インストール先> ¥UcaEnv¥dll と作業フォルダ UcaWork¥dll および UcaWork¥lib を「ライブラリファイル」パスにそれぞれ設定します。

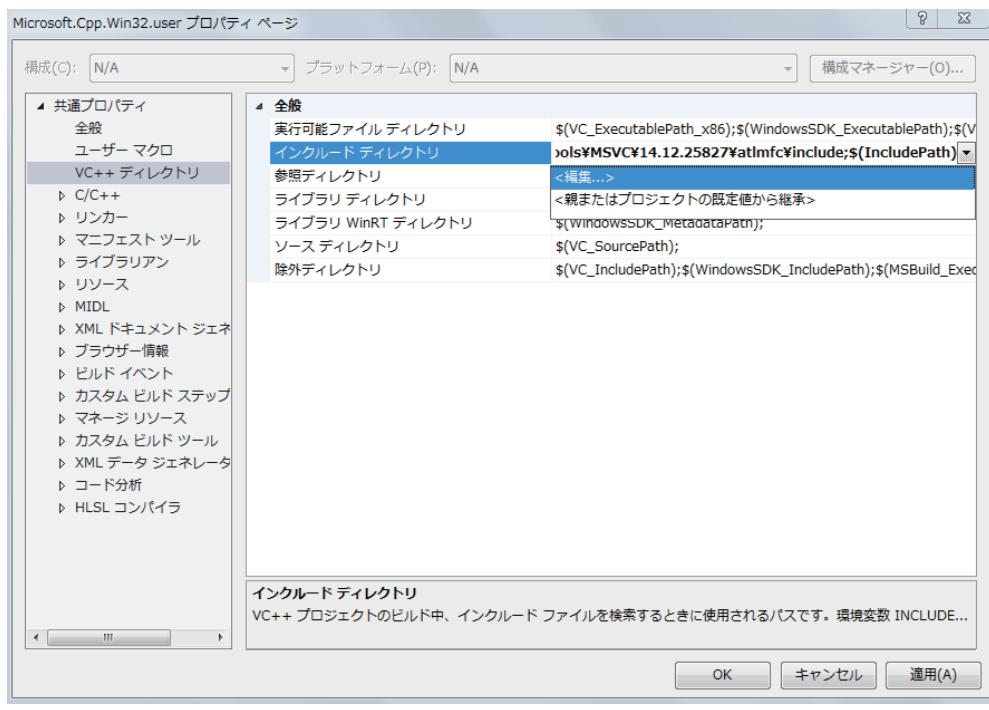
CENTUM VP を「C:¥CENTUMVP」にインストールし、UcaWork を「C:¥UcaWork」に配置したときの設定例を次に示します。

1. Visual Studio で、新規プロジェクトを作成し、[表示] – [プロパティマネージャー] を選択してください。
プロパティマネージャーウィザードが表示されます。
2. [Release | Win32] のユーザカスタムアルゴリズム開発で使用しているユーザプロパティシートを右クリックし [プロパティ] を選択してください。
「プロパティページ」ダイアログが表示されます。

参照 ディレクトリパスの設定の詳細およびユーザカスタムブロック開発環境パッケージのフォルダ構成の詳細については、以下を参照してください。

[APCS ユーザカスタムブロック開発環境 \(IM 33J15U23-01JA\)](#)

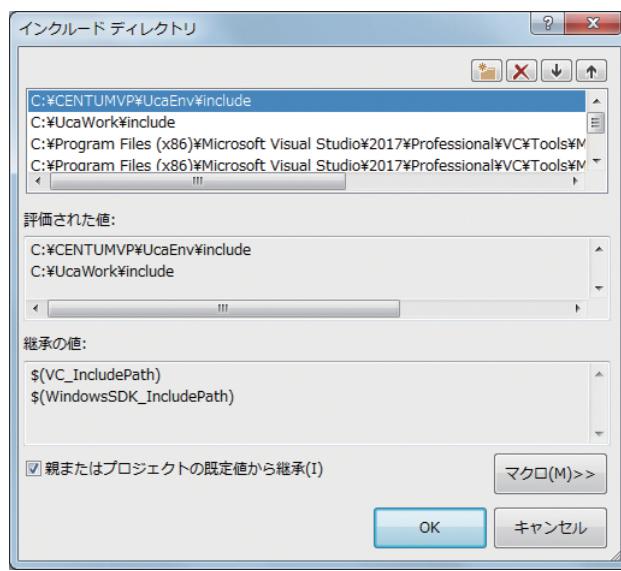
インクルードファイルのパス設定の例です。



010104J.ai

図 インクルードファイルのパス設定

1. ダイアログの左ペインの、[共通プロパティ] – [VC++ ディレクトリ] を選択してください。
 2. 右ペインの [全般] – [インクルードディレクトリ] を選択し、[編集] をクリックしてください。
- インクルードディレクトリダイアログが表示されます。

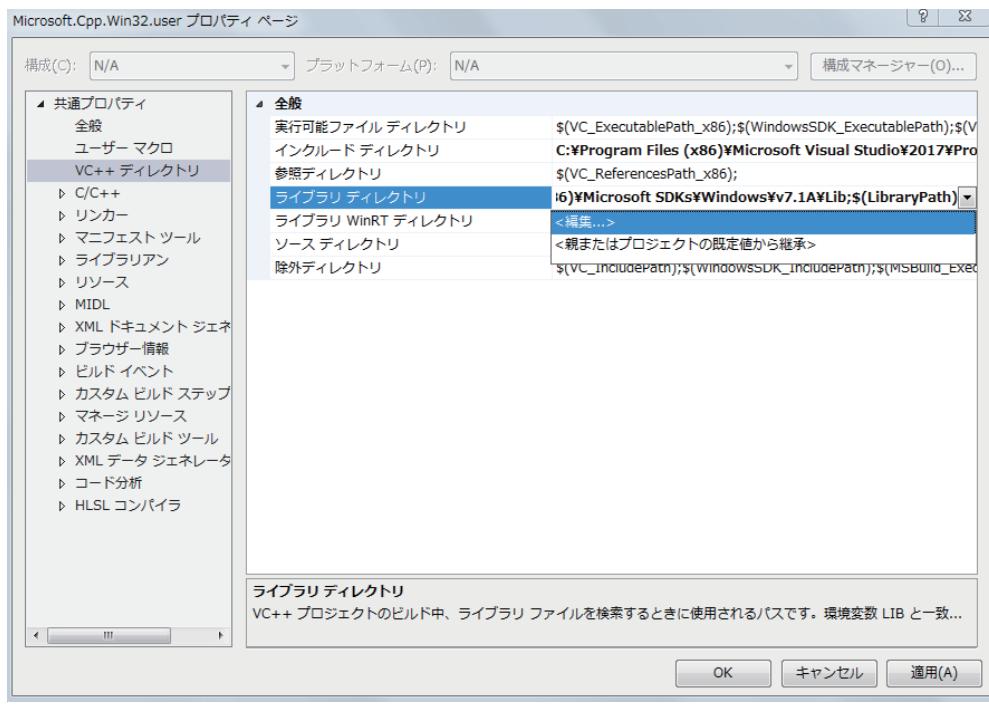


010105J.ai

図 インクルードディレクトリダイアログ

3. 一番上に、[C:\CENTUMVP\UcaEnv\include]、[C:\UcaWork\include] の順にパスを設定してください。
4. [OK] ボタンをクリックしてください。

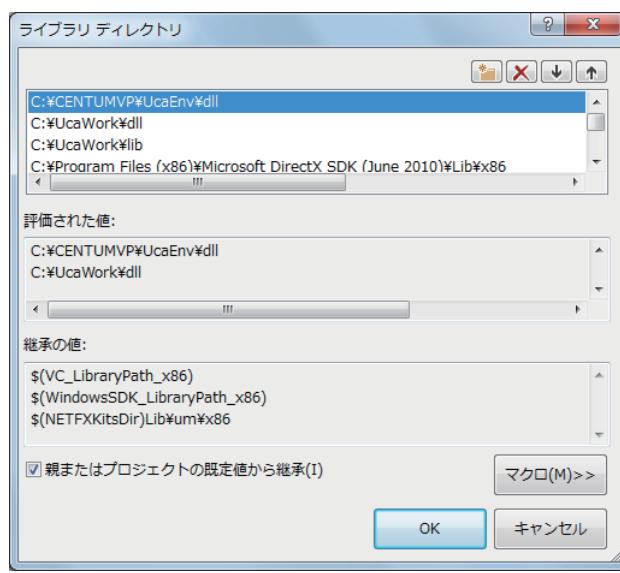
ライブラリファイルのパス設定の例です。



010106J.ai

図 ライブラリファイルのパス設定

1. ダイアログの左ペインの、[共通プロパティ] – [VC++ ディレクトリ] を選択してください。
 2. 右ペインの [全般] – [ライブラリディレクトリ] を選択し、[編集] をクリックしてください。
- ライブラリディレクトリダイアログが表示されます。



010107J.ai

図 ライブラリディレクトリダイアログ

3. 一番上に、[C:\CENTUMVP\UcaEnv\ dll]、[C:\UcaWork\ dll]、[C:\UcaWork\ lib] の順にパスを設定してください。
4. [OK] ボタンをクリックしてください。
5. プロパティページダイアログで [OK] ボタンをクリックしてください。

● アラームステータスの定義

ユーザカスタムブロックのアラームステータスの初期値を定義します。システムビューで、CENTUM VP プロジェクトの COMMON にある AlmStsLabel をダブルクリックすると、ユーザ定義ステータス文字列ビルダが表示されます。「アラームステータス定義」の USER16 がユーザカスタムブロックのアラームステータス定義です。[編集] メニューの「ユーザカスタムブロックデフォルトの取り込み」操作で、USER16 にアラームステータスのデフォルトが設定されますので、[ファイル] – [上書き保存] で書き込みます。



図 ユーザ定義ステータス文字列ビルダ

アラームステータスの定義を反映します。システムビューの [プロジェクト] – [無効素子の解決] で以下のダイアログを呼び出し、[開始] ボタンをクリックします。

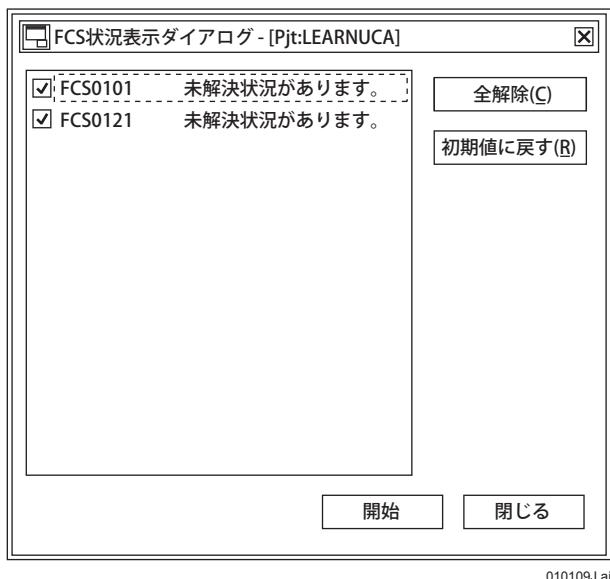


図 FCS状況表示ダイアログ

以下のダイアログでブロックステータス定義の変更が FCS0121 (APCS) に反映されたことがわかります。[閉じる] ボタンでダイアログを閉じます。

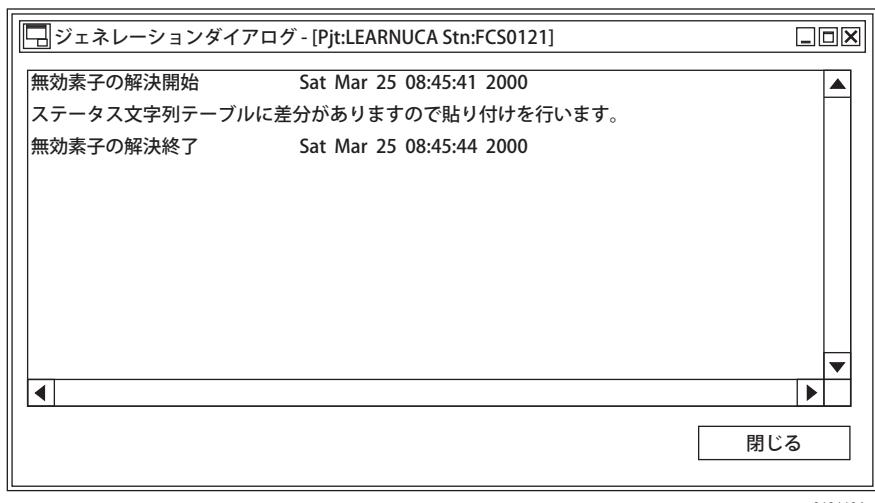


図 ジェネレーションダイアログ

これで準備ができました。

■ ユーザカスタムアルゴリズムのリビルドと登録

ユーザカスタムブロックを実際に動かしてみます。サンプルプログラムは、準備作業で以下のフォルダにコピーされています。

<ドライブ名>¥UcaWork¥UcaSamples¥_SMPL_LETSSTART

Visual Studio を起動し、_SMPL_LETSSTART¥_SMPL_LETSSTART.slnを開きます。スクロールバーを動かして、ソースコードから下図の部分を見つけてください。このユーザカスタムアルゴリズムは、機能ブロック定周期処理で、「さあ、始めましょう！」というシステムアラーム復帰メッセージを出力します。つまり、定周期で「さあ、始めましょう！」というメッセージを出力します。

Visual Studio の [ビルド] メニューの [リビルド] により、Release 版をリビルドします。

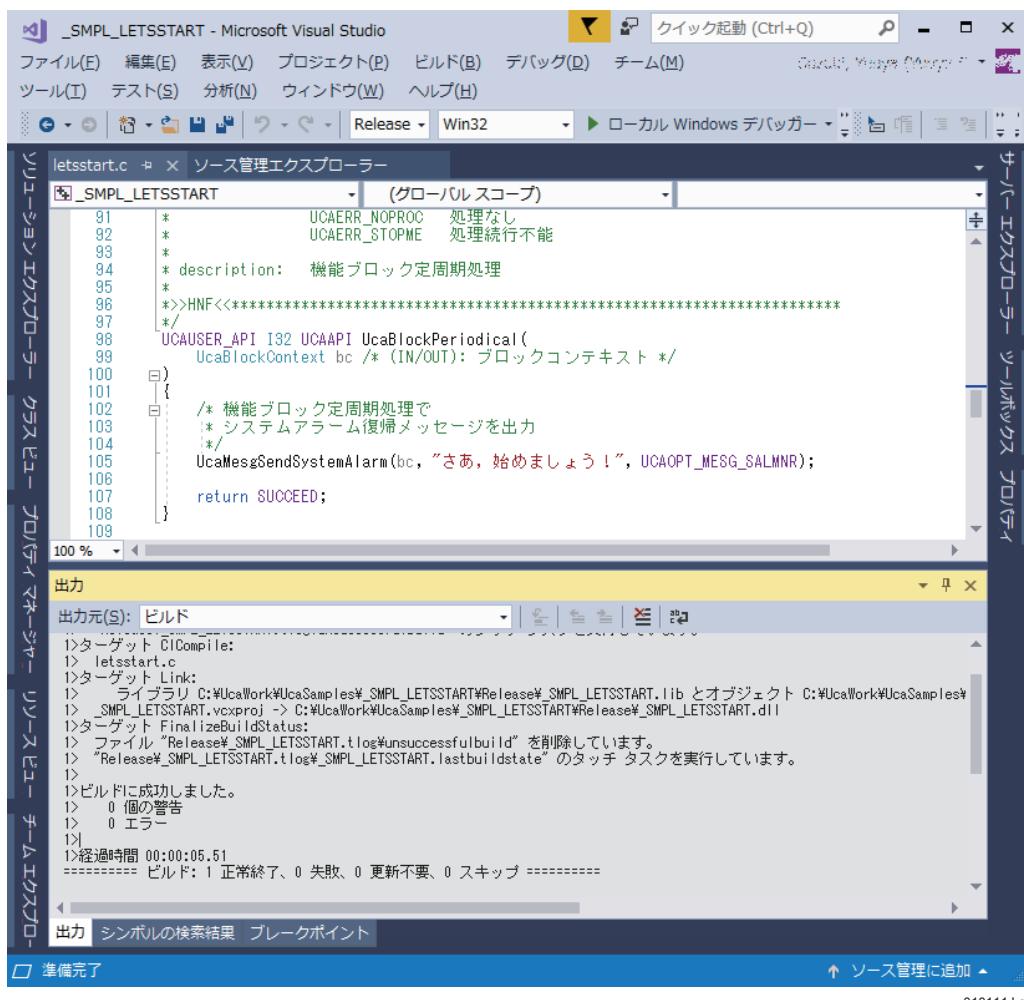


図 サンプルプログラム

システムビューより、ユーザカスタムアルゴリズムを CENTUM VP プロジェクトに登録します。

重要

ユーザカスタムアルゴリズムを登録するときは、Visual Studio より該当のソリューション(ユーザカスタムアルゴリズム)を閉じてください。Visual Studio が開いているソリューションを登録することはできません。登録時にエラーになります。

システムビューで、LEARNUCA プロジェクト、FCS0121 (APCS) のUSERCUSTOM フォルダを選択します。[ファイル] の [新規作成] – [ユーザカスタムアルゴリズム] をクリックすると、以下のユーザカスタムアルゴリズム登録ダイアログが表示されます。

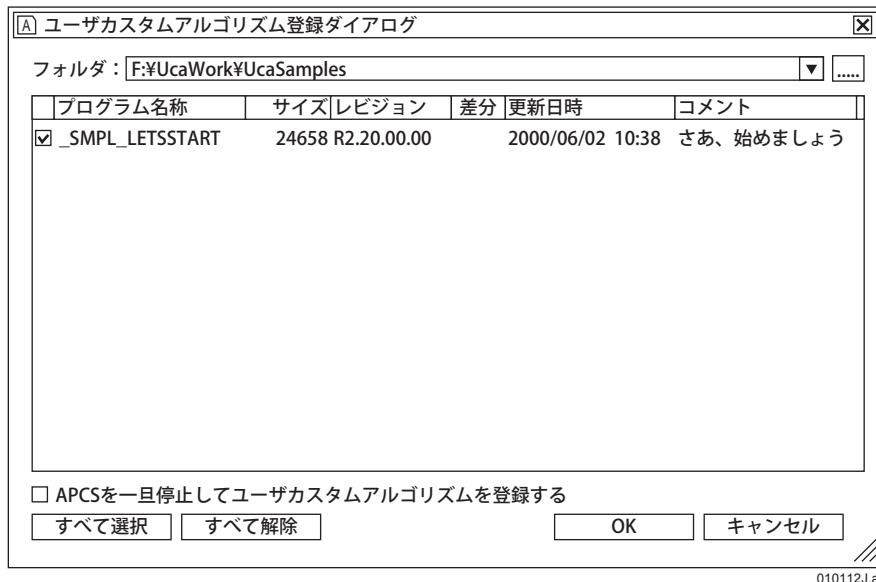


図 ユーザカスタムアルゴリズム登録ダイアログ

ダイアログ上部の「フォルダ」の右側にある [...] ボタンをクリックすると、以下のフォルダの参照ダイアログが表示されますので、<ドライブ名>¥UcaWork¥UcaSamples を指定し、[OK] ボタンをクリックします。ここでは、Visual Studio のソリューションのフォルダ (_SMPL_LETSSTART) の一段上のフォルダ (UcaSamples) を指定します。

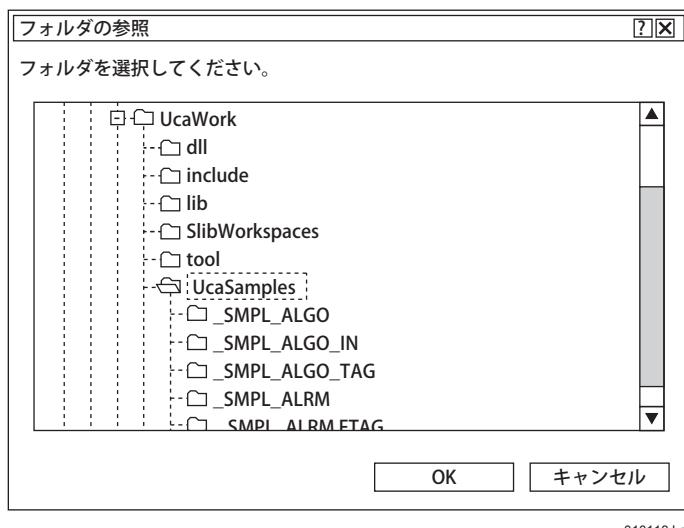


図 フォルダの参照ダイアログ

ユーザカスタムアルゴリズム登録ダイアログに戻るため、SMPL_LETSSTART の左にチェック印 (V) をつけて [OK] ボタンをクリックします。

ユーザカスタムアルゴリズム SMPL_LETSSTART が、CENTUM VP プロジェクト LEARNUCA の USERCUSTOM に登録されます。以下は、システムビューで _SMPL_LETSSTART を表示している場合です。

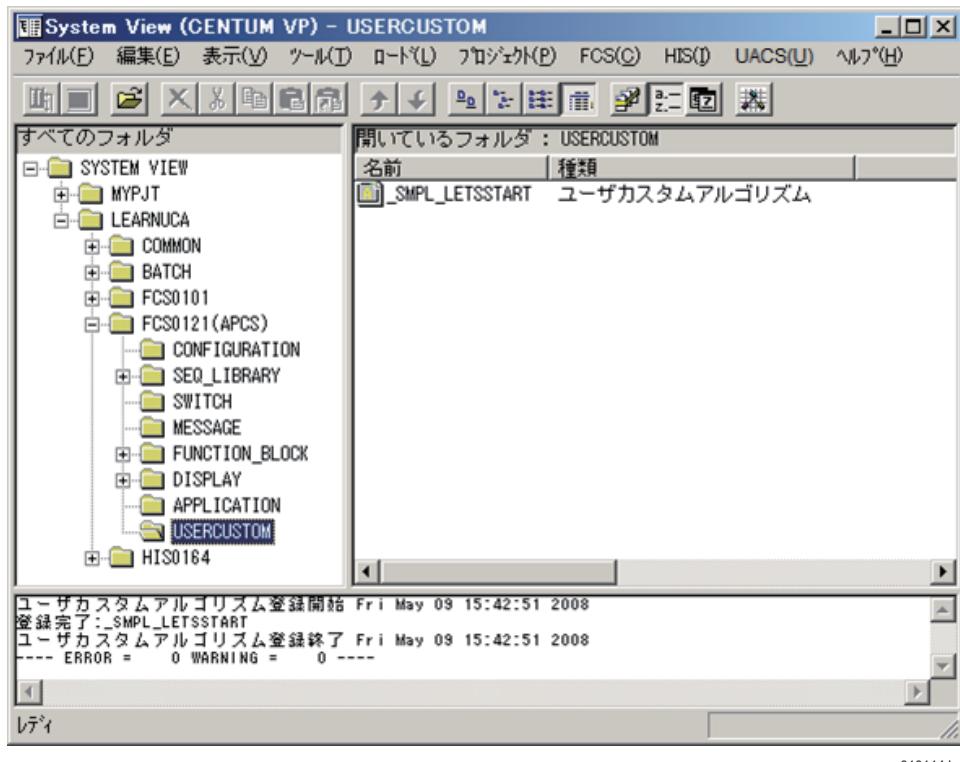
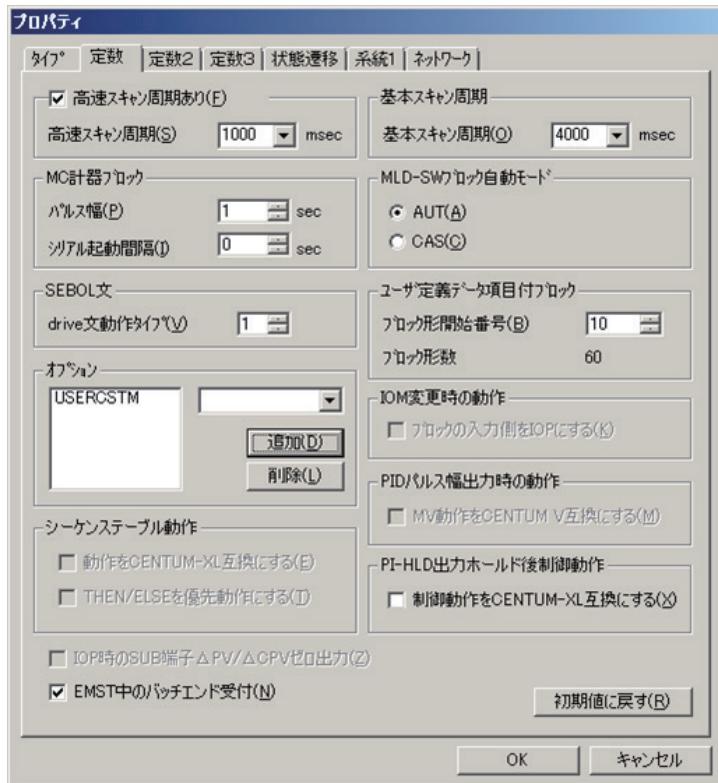


図 システムビューでの _SMPL_LETSSTART の表示

これで、ユーザカスタムアルゴリズム _SMPL_LETSSTART が登録されました。

■ ユーザカスタムブロックを制御ドローイングビルダで定義する

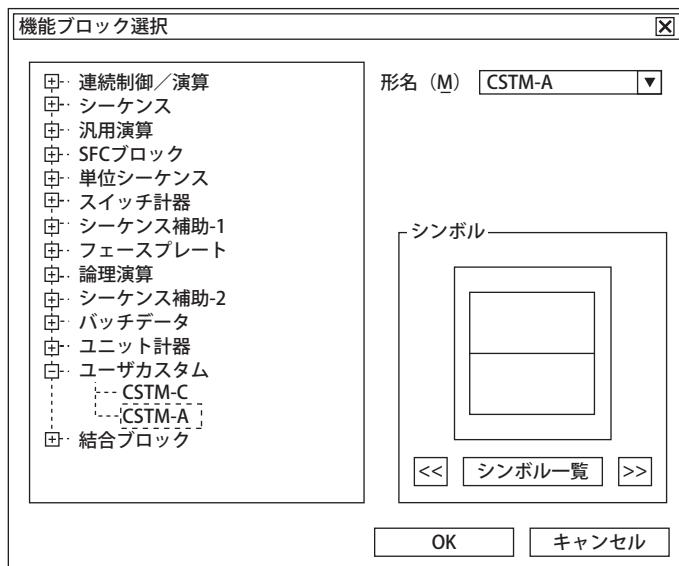
ユーザカスタムアルゴリズム _SMPL_LETSSTART を使用するユーザカスタムブロックを定義し、実行するには、まず、ユーザカスタムブロックを使用できるようにオプションを設定します。システムビューを起動し、FCS0121(APCS) のプロパティを開きます。定数タブを選択後、オプションの選択エリアで USERCSTM を選択し、[追加] ボタンをクリックします。



010119J.ai

図 ユーザカスタムブロックの定義

続いて、FCS0121 (APCS) の制御ドローイング DR0010 を開いてください（空いている制御ドローイングならどれでも構いません）。機能ブロック選択ダイアログで CSTM-A を選択します。



010115J.ai

図 機能ブロック選択ダイアログ

CMA01_START というタグ名で、汎用演算形ユーザカスタムブロックを定義します。本書では、以下のようにタグ名を命名します。

- 汎用演算形ユーザカスタムブロック (CSTM-A) のタグ名は、CMA で始まります。
- 連続制御形ユーザカスタムブロック (CSTM-C) のタグ名は、CMC で始まります。

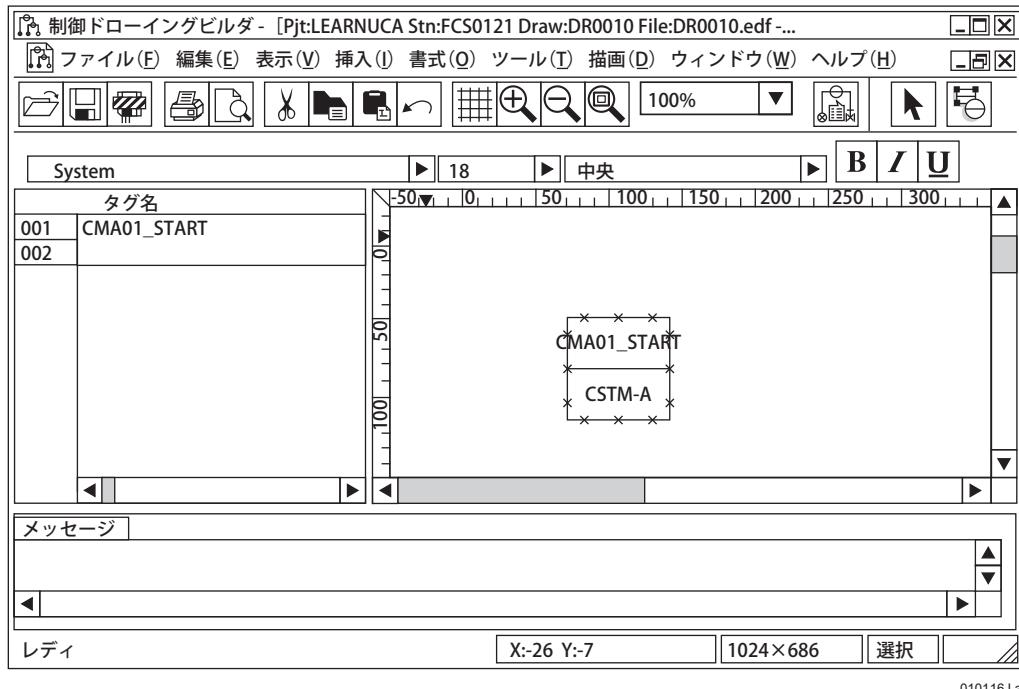


図 汎用演算形カスタムブロックCMA01_START

CMA01_START を指定して機能ブロック詳細ビルダを起動します。そして、[プログラム名称] に「_SMPL_LETSSTART」と指定します。機能ブロック詳細定義ビルダの [ファイル] - [更新] で修正を書き込みます。

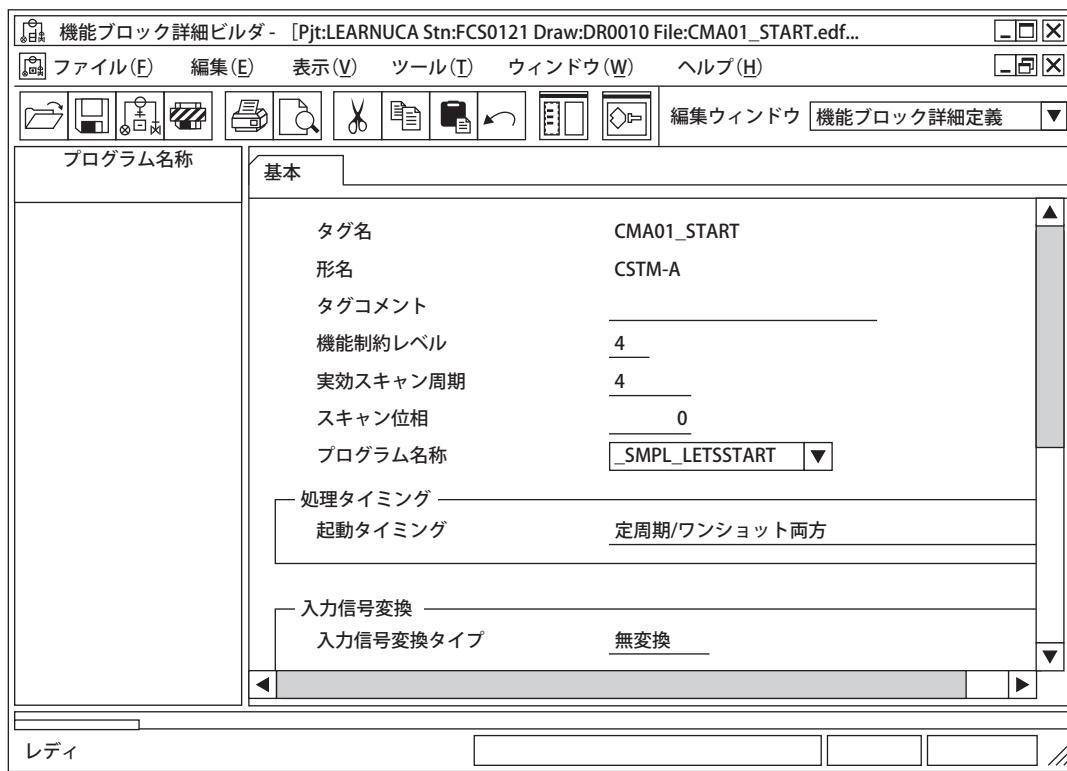


図 プログラム名称「_SMPL_LETSSTART」の指定

制御ドローイングビルダで [ファイル] – [上書き保存] により制御ドローイングをジェネレーションします。ジェネレーションが終了したら、システムビューから FCS0121 (APCS) を指定してテスト機能を起動します。FCS0121 がスタートしたら、システムアラームウィンドウで「さあ、始めましょう！」というシステムアラーム復帰メッセージが出力されていることを確認してください。

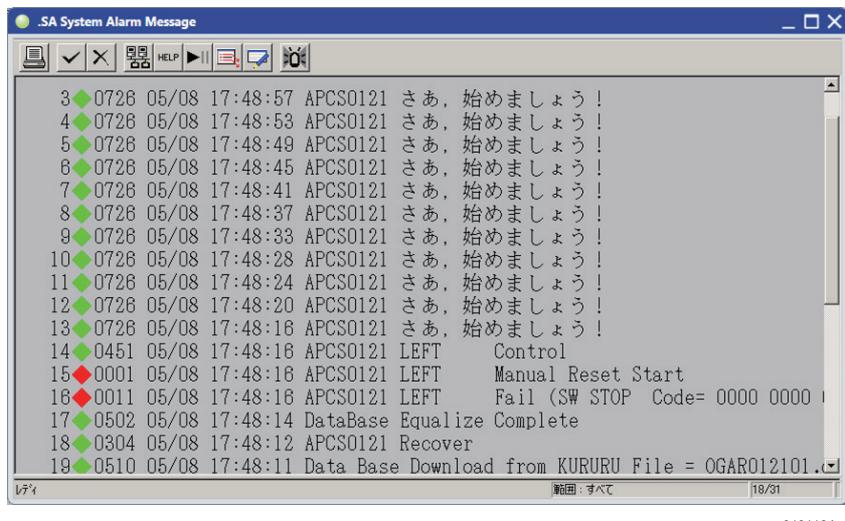


図 システムアラームウィンドウ

● ブロックモードをO/Sにする

汎用演算形ユーザカスタムブロック CMA01_START は、システムアラームメッセージ「さあ、始めましょう！」を出力し続けていますので、CMA01_START のブロックモードを AUT から O/S に変更してください。CMA01_START は、システムアラームメッセージの出力を停止します。ブロックモードを O/S(Out Of Service) にするとユーザカスタムブロックの機能はすべて停止します。

1.2 機能ブロック定周期処理

1つのユーザカスタムアルゴリズムには、次の5つの動作タイミングで起動される関数を定義します。

- ・機能ブロック初期化処理
- ・機能ブロック終了処理
- ・機能ブロック定周期処理
- ・機能ブロックワンショット起動処理
- ・機能ブロックデータ設定時特殊処理

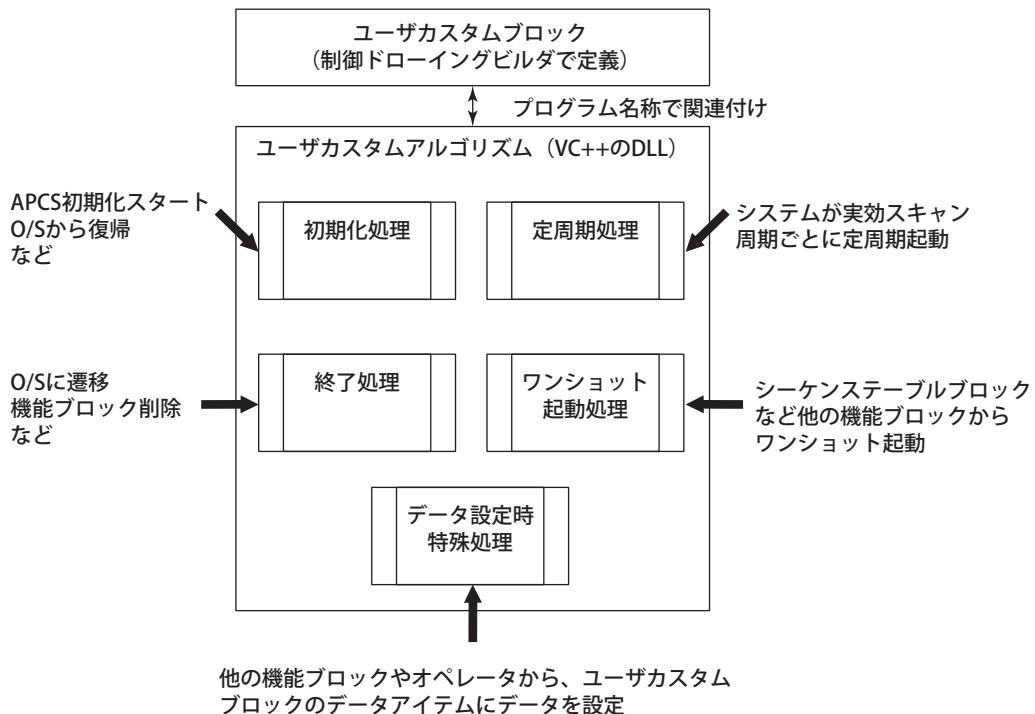


図 ユーザカスタムアルゴリズムの5つの処理

010201J.ai

これから、サンプルプログラムを使用して、これら5つの処理の動作を説明します。機能ブロック定周期処理の動作から説明します。

まず、サンプルソリューション _SMPL_SAMPLE1 をリビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは準備作業により以下の作業フォルダにコピーされています。

<ドライブ名>:\UcaWork\UcaSamples_\SMPL_SAMPLE1

Visual Studio を起動し、_\SMPL_SAMPLE1_\SMPL_SAMPLE1.sln を開きます。[ビルド] メニューの [リビルド] により、_\SMPL_SAMPLE1 の Release 版をリビルドします。ビルドの構成には、Release 版と Debug 版がありますが、ユーザカスタムブロックの開発は、最初から最後まで Release 版のみを使用します。リビルドが完了したら、システムビューでユーザカスタムアルゴリズムを登録します。

重要

ユーザカスタムアルゴリズムを登録するときは、Visual Studio より該当のソリューション(ユーザカスタムアルゴリズム)を閉じてください。Visual Studio が開いているソリューションを登録することはできません。登録時にエラーになります。

システムビューで、LEARNUCA プロジェクト、FCS0121 (APCS) の USERCUSTOM フォルダを選択します。[ファイル] の [新規作成] – [ユーザカスタムアルゴリズム] をクリックすると、以下のユーザカスタムアルゴリズム登録ダイアログが表示されます。

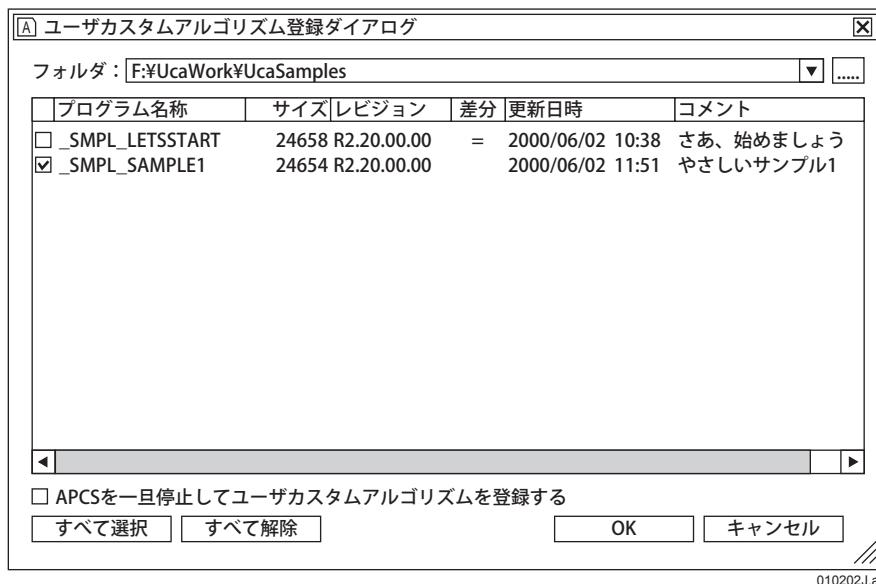


図 ユーザカスタムアルゴリズム登録ダイアログ

[フォルダ] に<ドライブ名>:\UcaWork\UcaSamples を指定します（右側の [...] ボタンをクリックすると、フォルダの選択ダイアログが表示されます）。_\SMPL_SAMPLE1 の右にチェック印（V）をつけて、[OK] ボタンをクリックすると、ユーザカスタムアルゴリズムが CENTUM VP プロジェクト LEARNUCA に登録されます。

補足

ユーザカスタムアルゴリズム登録ダイアログの「差分」欄に、「=」マークがありますが、これはCENTUM VPプロジェクトに登録されているファイル(DLL)と、ソリューション内のファイル(DLL)の更新日時が一致していることを示します。[差分]の意味を次表に示します。

表 ユーザカスタムアルゴリズム登録ダイアログの「差分」の意味

差分の表示	説明
=	CENTUM VPプロジェクトに登録してあるファイル(DLL)と、ソリューション内のファイル(DLL)の更新日時が一致しています。
*	CENTUM VPプロジェクトに登録してあるファイル(DLL)と、ソリューション内のファイル(DLL)の更新日時は、一致していません。通常は、ソリューションのプログラムを修正し、ファイル(DLL)を再ビルトしていることを意味します。
空欄	ソリューションはCENTUM VPプロジェクトに登録されていません。

ユーザカスタムアルゴリズムを登録すると、システムビューでFCS0121(APCS)のUSERCUSTOMフォルダに登録されているユーザカスタムアルゴリズムの一覧が表示されます。

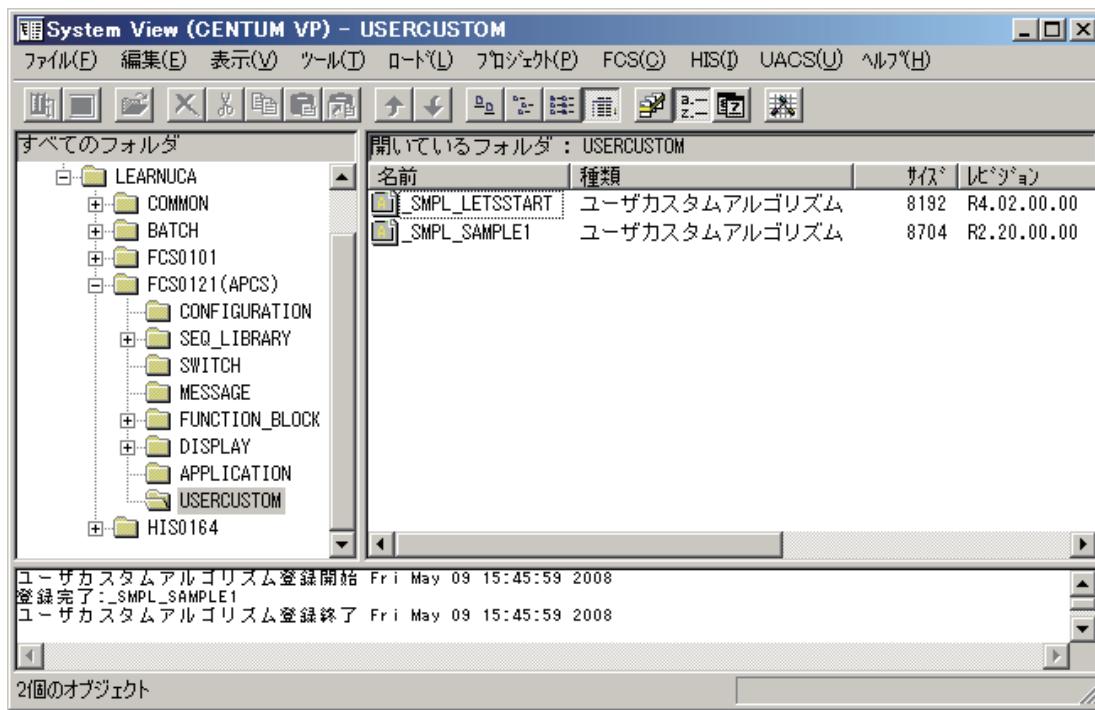


図 USERCUSTOMフォルダに登録されているユーザカスタムアルゴリズム

次に、空いている制御ドローイングにサンプルとして用意されている制御ドローイングをインポートします。ここでは、DR0010にインポートします。DR0010には、前節で作成したCMA01_STARTがありますが、定周期で「さあ始めましょう！」メッセージを出力し続けるため、上書きしてしまいます。サンプルの制御ドローイングを定義したテキストファイルがCENTUM VPインストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥drawings¥SAMPLE1.txt

FCS0121(APCS)の制御ドローイングのDR0010を指定して制御ドローイングビルダを起動します。[ファイル]メニューの[外部ファイル] – [インポート]を指定します。インポートダイアログで上記のSAMPLE1.txtを指定し、[開く]ボタンをクリックすると、以下の制御ドローイングが取り込まれます。[ファイル] – [上書き保存]で制御ドローイングを書き込みます。

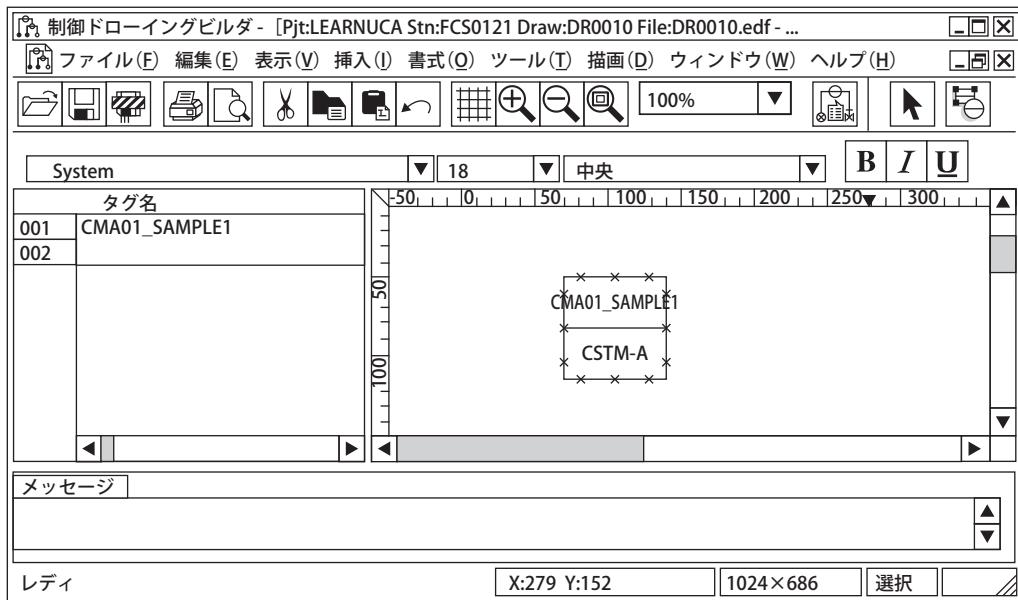


図 制御ドローイングDR0010へのSAMPLE1.txtのインポート

FCS0121 (APCS) を指定してテスト機能を起動します。そして、操作監視機能で CMA01_SMPL1 のチューニングウィンドウを表示します。ウィンドウに CMA01_SMPL1 のデータアイテム P01 と P02 が表示されています。データアイテム P01 と P02 が 4 秒周期で 1 ずつ増加しているのを確認してください。

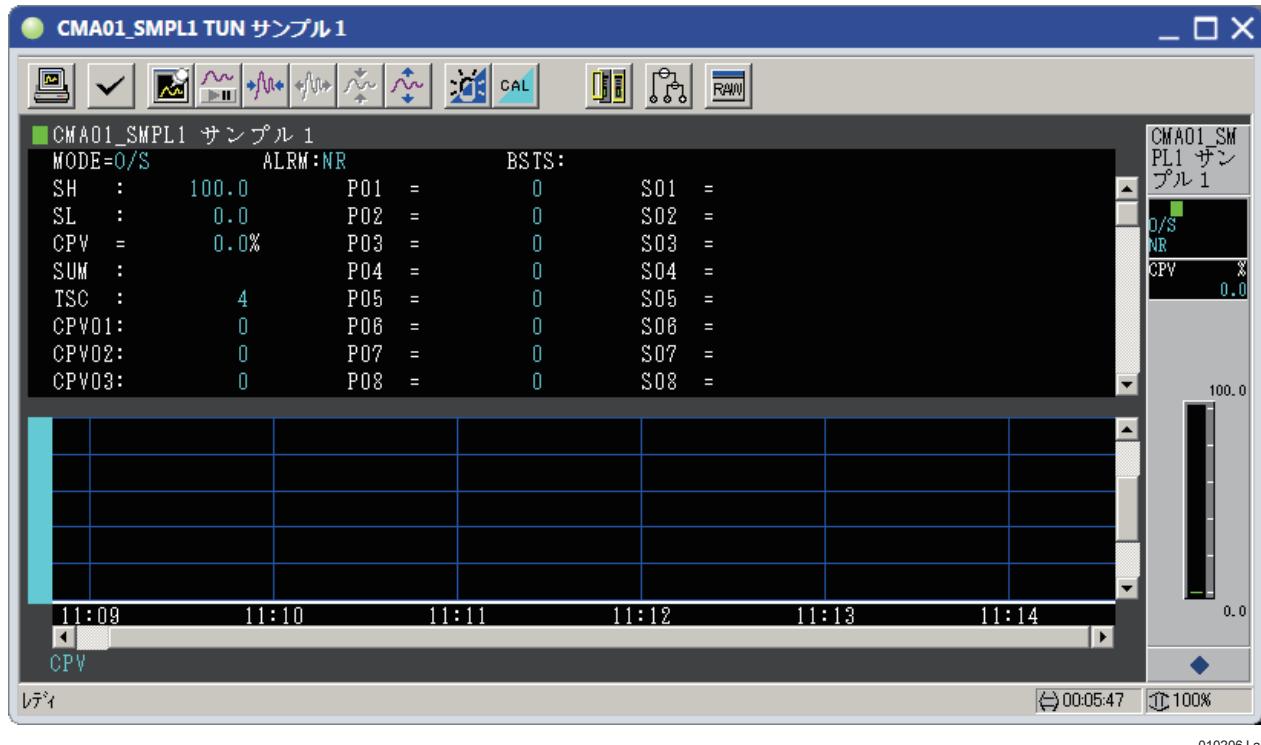
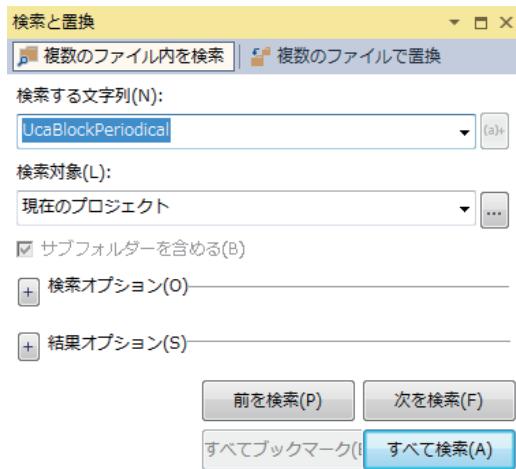


図 汎用演算形カスタムブロックCMA01_SMPL1のチューニングウィンドウ

■サンプルプログラムの機能ブロック定周期処理

Visual Studio でソリューション _SMPL_SAMPLE1 を開き、UcaBlockPeriodical という文字列で検索します。Visual Studio の [編集] – [検索と置換] – [フォルダーを指定して検索] を選択すると、検索と置換ダイアログが表示されます。



010207J.ai

図 ファイルから検索ダイアログ

検索文字列に「UcaBlockPeriodical」を入力します。[すべて検索] ボタンをクリックすると検索結果が Visual Studio の下側のペインに出力されます。表示したい行をダブルクリックすると、右上のペインに該当行を含むソースファイルが表示されます。

```

102
103 /*<<FNH>>*****
104 *
105 * Function name: UcaBlockPeriodical
106 * Return value: SUCCEED 正常終了
107 *                 UCAERR_NOPROC 処理なし
108 *                 UCAERR_STOPME 処理続行不能
109 *
110 * description: 機能ブロック定周期処理
111 *
112 *>>HNF<<*****
113 */
114 UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
115     UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
116 )
117 {
118     F64S p01;
119     F64S p02;
120     I32 rtnCode;
121
122     /* P01, P02を 1 増加 */
123     rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);
124     rtnCode = UcaDataGetPn(bc, &p02, 2, 1, NOOPTION);
125     p01.value += 1;
126     p02.value += 1;
127     rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);
128 }
```

検索結果 1

- すべて検索 "UcaBlockPeriodical", サブ フォルダー, 検索結果 1, 現在のプロジェクト: _SMPL_SAMPLE1.vcxproj, ""

C:\UcaWork\UcaSamples_\SMPL_SAMPLE1 - コピー\sample1.c(106):* Function name: UcaBlockPeriodical

C:\UcaWork\UcaSamples_\SMPL_SAMPLE1 - コピー\sample1.c(115):UCAUSER_API I32 UCAAPI UcaBlockPeriodical()

010208J.ai

図 検索結果

```

/*
* <<FNH>>*****
*
* Function name:      UcaBlockPeriodical
* Return value: SUCCEED    正常終了
*                   UCAERR_NOPROC   処理なし
*                   UCAERR_STOPME   処理続行不能
*
* description: 機能ブロック定期処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    F64S p01;
    F64S p02;
    I32 rtnCode;

    /* P01,P02を1増加 */
    rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);
    rtnCode = UcaDataGetPn(bc, &p02, 2, 1, NOOPTION);
    p01.value += 1;
    p02.value += 1;
    rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);
    rtnCode = UcaDataStorePn(bc, &p02, 2, 1, NOOPTION);

    return SUCCEED;
}

```

UcaBlockPeriodical は、システムがユーザカスタムブロックに定期周期で処理タイミングを与えるごとに呼び出される、ユーザ定義の関数です。処理の先頭行は、自ブロックのデータアイテム P01、P02 の値を読み出す関数 UcaDataGetPn を呼び出しています。

```

rtncode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);
rtncode = UcaDataGetPn(bc, &p02, 2, 1, NOOPTION);

```

UcaDataGetPn は、システムが提供するユーザカスタムアルゴリズム作成用ライブラリです。ユーザカスタムアルゴリズム作成用ライブラリの関数名は、Uca で始まります。自ブロックのデータアクセス、入出力結合端子アクセス、他の機能ブロックデータのアクセス、制御演算など種々のライブラリが用意されています。

UcaDataGetPn の引数並びの「&p02, 2, 1」の部分は、「データアイテム P01～P32 の 2 番目から 1 つ分のデータを変数 p02 に格納する」つまり変数 p02 にデータアイテム P02 の値を読み込みます。機能ブロック定期処理 UcaBlockPeriodical は、自ブロックのデータアイテム P01、P02 の値を変数 p01、p02 に読み込み、それぞれに 1 を加え、UcaDataStorePn でデータアイテム P01 と P02 に設定しています。

ユーザカスタムブロック CMA01_SMPL1 の実効スキャン周期を変更して、データアイテム P01 と P02 が増加する周期が変わることを確認します。CMA01_SMPL1 を指定して機能ブロック詳細ビルダを起動します。プログラム名称が _SMPL_SAMPLE1 になっているのを確認してください。実効スキャン周期は、デフォルトの 4 秒になっていますので、8 秒（メニューの 4:8）に指定を変更します。

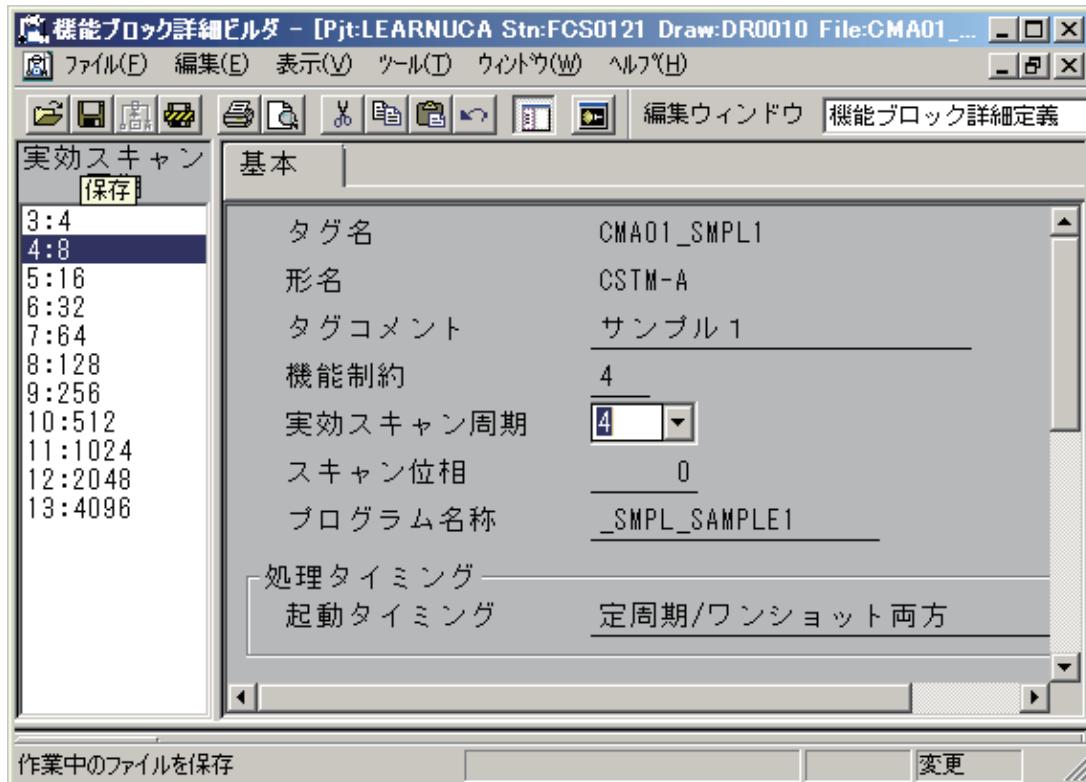


図 機能ブロック詳細ビルダで実効スキャン周期に8秒(4:8)を指定

010209J.ai

機能ブロック詳細ビルダから [ファイル] – [更新] で書き込みます。そして、制御ドローリングビルダから修正を書き込み ([ファイル] – [上書き保存]) ます。CMA01_SMPL1 のチューニングウィンドウで、データアイテム P01 と P02 が 8 秒ごとに 1 ずつ増加することを確認してください。

1.3 機能ブロック終了処理

この節では、機能ブロック終了処理について説明します。

ユーザカスタムブロック CMA01_SMPL1 のブロックモード (MODE) を AUT から O/S に変更します。チューニングウィンドウで次の 2 つの変化が確認できます。

1. データアイテム P01 と P02 の増加が停止します。これは、ブロックモードが O/S (Out of Service) になったため、機能ブロック定期処理 UcaBlockPeriodical が呼び出されなくなつたためです。
2. データアイテム S01 に「停止中」という文字列が設定されています。ユーザカスタムアルゴリズム _SMPL_SAMPLE1 の機能ブロック終了処理で設定しています。

■サンプルプログラムの機能ブロック終了処理

ソースコードを確認します。sample1.c を UcaBlockFinish という文字列で検索し、以下の部分を見つけてください。

```
/*
* <>FNH>*****
*
* Function name:      UcaBlockFinish
* Return value:       SUCCEED      正常終了
*                      UCAERR_NOPROC   処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description:        機能ブロック終了処理
*
*>>HNF<>*****
*/
UCAUSER_API I32 UCAAPI UcaBlockFinish(
    UcaBlockContext bc,           /* (IN/OUT): ブロックコンテキスト */
    I32 reason                  /* (IN):呼び出し理由 */
)
{
    I32 rtnCode;

    /* S01に「停止中」を設定 */
    rtnCode = UcaDataStoreEachSn(bc, "停止中", 1, NOOPTION);

    return SUCCEED;
}
```

UcaBlockFinish は、ユーザカスタムブロックが実行を停止するときに、システムから呼び出される、ユーザ定義の関数です。システムは、ユーザカスタムブロックのブロックモードが O/S に変わる直前に、機能ブロック終了処理 UcaBlockFinish を呼び出します。この例では、UcaDataStoreEachSn により自ブロックのデータアイテム S01 に「停止中」という文字列を設定します。

参照 機能ブロック終了処理の詳細については、以下を参照してください。

[「3.3 機能ブロック終了処理」](#)

1.4 機能ブロック初期化処理

この節では、機能ブロック初期化処理について説明します。

ユーザカスタムブロック CMA01_SMPL1 のブロックモード (MODE) を O/S から AUT に変更します。チューニングウィンドウで次の 2 つの変化が確認できます。

1. データアイテム P01 と P02 の増加が 0 に初期化されます。その後定周期で 1 ずつ増加します。
2. データアイテム S01 の「停止中」という文字列が消えます。

■ サンプルプログラムの機能ブロック初期化処理

ソースコードを確認します。sample1.c を UcaBlockInit という文字列で検索し、以下の部分を見つけてください。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockInit
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME    処理続行不能
*
* description:            機能ブロック初期化処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockInit(
    UcaBlockContext bc,           /* (IN/OUT): ブロックコンテキスト */
    I32 reason                  /* (IN):呼び出し理由 */
)
{
    F64S zero;
    I32 rtnCode;

    /* P01,P02を0に初期化 */
    zero.value = 0;
    zero.status = 0;
    rtnCode = UcaDataStorePn(bc, &zero, 1, 1, NOOPTION);
    rtnCode = UcaDataStorePn(bc, &zero, 2, 1, NOOPTION);

    /* S01をNULL文字列に初期化 */
    rtnCode = UcaDataStoreEachSn(bc, "", 1, NOOPTION);

    return SUCCEED;
}
```

UcaBlockInit は、ユーザカスタムブロックが実行を開始するときに、システムから呼び出されるユーザ定義の関数です。ユーザカスタムブロックのブロックモードが O/S から AUT に復帰した直後に、システムは機能ブロック初期化処理 UcaBlockInit を呼び出します。この例では、データアイテム P01 と P02 に 0, S01 に NULL 文字列をそれぞれ設定しています。

参照 機能ブロック初期化処理の詳細については、以下を参照してください。
[「3.2 機能ブロック初期化処理」](#)

1.5 機能ブロックデータ設定時特殊処理

機能ブロックデータ設定時特殊処理は、自ブロックのデータアイテムに外部からデータを設定されるときに実行されます。「外部」とはHISからのオペレータ入力や、他の機能ブロックからのデータ設定を意味します。

ユーザカスタムブロック CMA01_SMPL1 のデータアイテム P03 に HIS から「3」を入力してください。また、データアイテム S02 に「ABC」（半角）を入力してください。チューニングウィンドウでは P03 に「3」、S02 に「ABC」がそれぞれ設定されています。

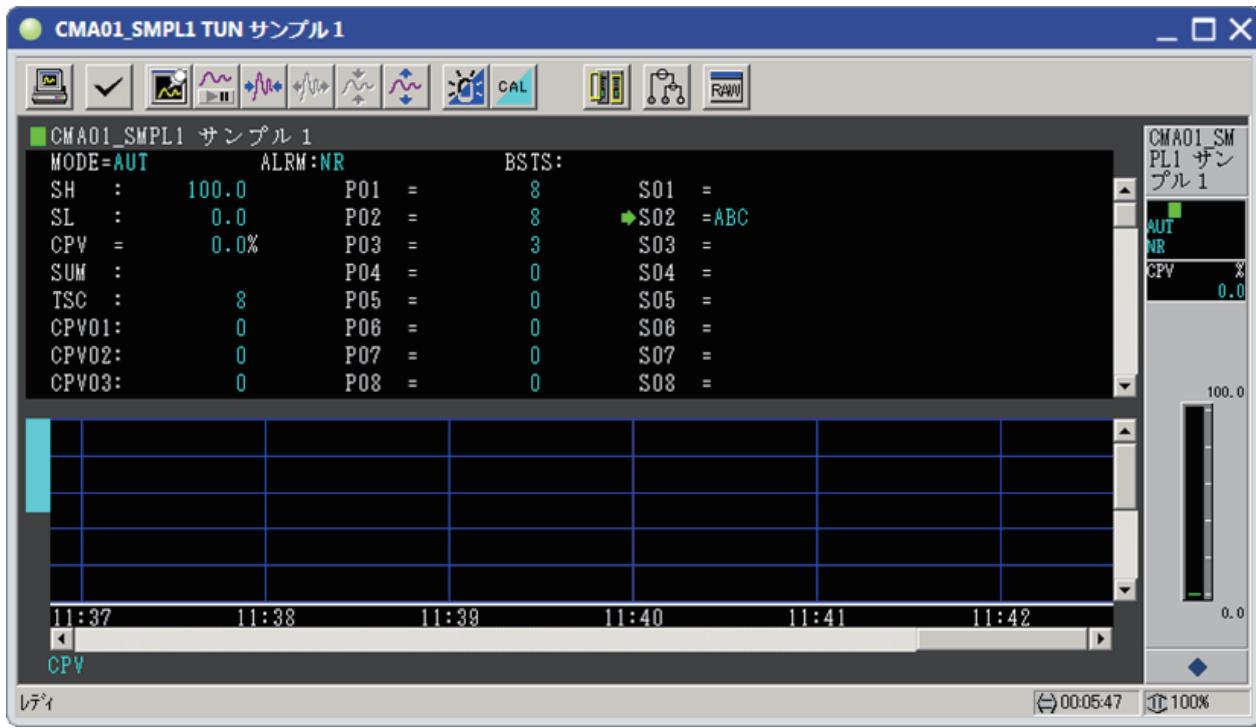
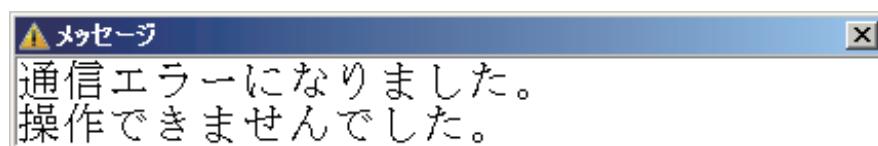


図 CMA01_SMPL1のP03とS02にデータを設定

次に、P03 に 12 (10 より大きい数) を入力します。入力はエラーとなり HIS で以下のダイアログが表示されます。



010502J.ai

図 メッセージダイアログ

S02 に「XYZ」を入力すると同様にエラーとなります。これはユーザカスタムアルゴリズム _SMPL_SAMPLE1 に定義されている機能ブロックデータ設定時特殊処理で、入力できる値を特定の範囲に制限しているためです。

■サンプルプログラムの機能ブロックデータ設定時特殊処理

ソースコードを確認します。sample1.c を UcaBlockDset という文字列で検索し、以下の部分を見つけてください。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockDset
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME   処理続行不能
*
* description:            機能ブロックデータ設定時特殊処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockDset (
    UcaBlockContext bc,             /* (IN/OUT): ブロックコンテキスト */
    DITMN itemName,                /* (IN): データアイテム名 */
    UcaUnivType *data,             /* (IN): 設定データ */
    UcaDataOrStatus dataOrStatus, /* (IN): 設定種別 */
    UcaPreOrPost preOrPost,        /* (IN): 呼び出し種別 */
    BOOL *result                  /* (OUT): 設定可否 */
)
{
#define HIGH_LIMIT 10.0           /* P03 設定値上限 */
#define LOW_LIMIT 0.0             /* P03 設定値下限 */

    F64S p03;
    I32 rtnCode;

    /* データ設定前処理でないなら何もしない */
    if (preOrPost != UCADSET_PRE) {
        return UCAERR_NOPROC;
    }

    /* データ設定前処理なら入力データの制限処理をする */
    if (strcmp(itemName, "P03", sizeof(DITMN)) == 0) {
        /* UcaUnivType として得られたデータを
         * F64S 型に変換しながら自動変数 s02 に設定する
         */
        rtnCode = UcaDataConvertType(bc, &data->dataValue,
                                     data->dType, &p03, UCA_DATATYPE_F64S, NOOPTION);

        /* 上下限チェック */
        if (LOW_LIMIT <= p03.value && p03.value <= HIGH_LIMIT) {
            *result = UCADSET_ALLOW;
        } else {
            *result = UCADSET_DENY;
        }
        return SUCCEED;
    } else if (strcmp(itemName, "S02", sizeof(DITMN)) == 0) {
        /* 先頭1文字が'A'なら入力できる */
        if (data->dataValue.string[0] == 'A') {
            *result = UCADSET_ALLOW;
        } else {
            *result = UCADSET_DENY;
        }
        return SUCCEED;
    }

    /* P03, S02 以外のデータアイテムは、処理なし */
}
```

```

    return UCAERR_NOPROC;
}

```

UcaBlockDset は、ユーザカスタムブロックのデータアイテムに外部からデータが設定されるとときに、システムから呼び出されるユーザ定義の関数です。システムは、データ設定の前後に、機能ブロックデータ設定時特殊処理を呼び出します。システムの処理は以下の図のようになります。

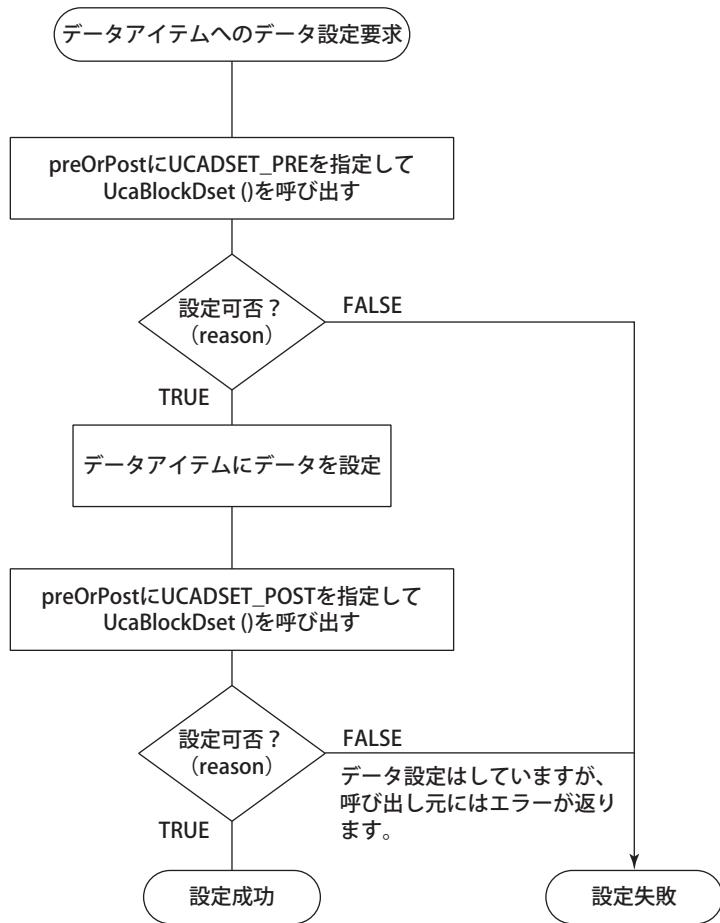


図 システムによる機能ブロックデータ設定時特殊処理の呼び出し

010503J.ai

_SMPL_SAMPLE1 の機能ブロックデータ設定時特殊処理 UcaBlockDset は、データ設定前処理でない（データ設定後処理）なら何もしないで処理を終了します。これがプログラム最初の以下の部分です。

```
/* データ設定前処理でないなら何もしない */
if (preOrPost != UCADSET_PRE) {
    return UCAERR_NOPROC;
}
```

データ設定前処理なら、データアイテム P03 と S02 への設定値に対して次の制限範囲内の検査をします。

- P03 は、0.0～10.0 の範囲内の値のみ入力可能とします。
- S02 は、先頭 1 文字が「A」の場合のみ入力可能とします。

P03, S02 どちらの検査でも、設定可能と判定すれば *result に UCADSET_ALLOW、設定不可と判定すれば UCADSET_DENY を指定し、SUCCEED でリターンされています。

P03 の場合は、システムより引数 data に指定された値を F64S 型に型変換しています。

```
rtncode = UcaDataConvertType(bc, &data->dataValue, data->dType, &p03,
                           UCA_DATATYPE_F64S, NOOPTION);
```

UcaDataConvertType は、data->dataValue に格納された data->dType 型の変数を、UCA_DATATYPE_F64S (F64S 型を示すラベル) に変換して p03 に格納します。次の部分で、P03 が 0.0(LOW_LIMIT) から 10.0(HIGH_LIMIT) の範囲内か検査を行い、*result (設定可否) を決定します。

```
/* 上下限チェック */
if (LOW_LIMIT <= p03.value && p03.value <= HIGH_LIMIT) {
    *result = UCADSET_ALLOW;
} else {
    *result = UCADSET_DENY;
}
return SUCCEED;
```

参照 機能ブロックデータ設定時特殊処理の詳細については、以下を参照してください。
[「3.6 機能ブロックデータ設定時特殊処理」](#)

補足 このプログラムでは、文字列の比較に Windows のライブラリ `strcmp` を使用しています。このプログラムは、`strcmp` を使用するために Windows が提供するインクルードファイル `string.h` を `#include` 行で取り込んでいます。

```
.....
/* **** */
* Windows インクルードファイル
*
* ユーザカスタムアルゴリズムで使用できる Windows のインクルードファイルは
* 以下の 9 個だけです。また、インクルードファイルで宣言されている関数を
* すべて使用できるわけではありません。
* 使用可能な関数は、ユーザカスタムアルゴリズムプログラミング説明書に
* 記載されています。
*/
/* #include <stdlib.h> */           /* Windows の文字列操作ライブラリ */
#include <string.h>
/* #include <math.h> */
/* #include <ctype.h> */
/* #include <stddef.h> */
/* #include <time.h> */
/* #include <float.h> */
/* #include <limits.h> */
/* #include <stdio.h> */
.....
```

参照 Windows のライブラリの詳細については、以下を参照してください。
「4.8 Windows のライブラリ」

1.6 機能ブロックワンショット起動処理

機能ブロックワンショット起動処理は、他の機能ブロックからユーザカスタムブロックがワンショット起動されたときに実行されます。ユーザカスタムブロックがシーケンステーブルからワンショット起動されるアプリケーションを例に説明します。

サンプルソリューション _SMPL_COND と _SMPL_OPRT の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは準備作業により以下の作業フォルダにコピーされています。

<ドライブ名> ¥UcaWork¥UcaSamples¥_SMPL_COND

_SMPL_COMD¥_SMPL_COND.sln を開きます。[ビルド] メニューの [リビルド] で _SMPL_COMD の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。_SMPL_OPRT も同じ手順で登録します。

次にサンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥drawings¥ONESHOT.txt

FCS0121 (APCS) の制御ドローイングの DR0011 (空いている制御ドローイングならどれでも構いません) を指定して制御ドローイングビルダを起動します。[ファイル] メニューの [外部ファイル] – [インポート] を指定します。インポートダイアログで上記の ONESHOT.txt を指定し、[開く] ボタンをクリックすると以下の制御ドローイングが取り込まれます。[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

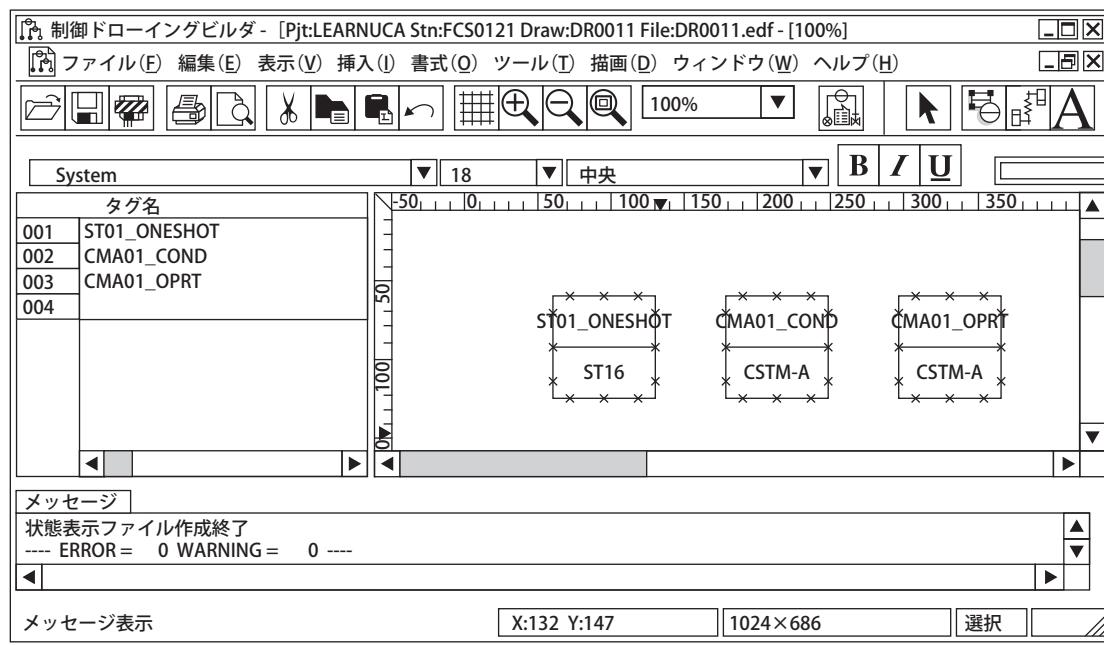


図 制御ドローイングのインポート

シーケンステーブルbrook ST01_ONESHOTを機能brook詳細ビルダで開きます。



010602J.ai

図 シーケンステーブルbrook ST01_ONESHOT

シーケンステーブル ST01_ONESHOT は、条件判定でユーザカスタムブロック CMA01_COND をワンショット起動します。CMA01_COND が真と判定すれば、1番のオペガイドメッセージ (%OG0001) が出力されます。

シーケンステーブル ST01_ONESHOT の状態操作として、ユーザカスタムブロック CMA01_OPRT をワンショット起動します。条件判定は、%SW0201～3 の PV で判定します。

- %SW0201.PV が 0 から 1 に変化するとデータに 10 を指定して CMA01_OPRT をワンショット起動します。
- %SW0202.PV が 0 から 1 に変化するとデータに 20 を指定して CMA01_OPRT をワンショット起動します。
- %SW0203.PV が 0 から 1 に変化するとデータに 30 を指定して CMA01_OPRT をワンショット起動します。

なお、ST01_ONESHOT の起動タイミングを次のように指定してあり、APCS の高速スキャン周期は 1 秒です。条件の変化を 1 秒周期で検査するために、「高速スキャン」にします。

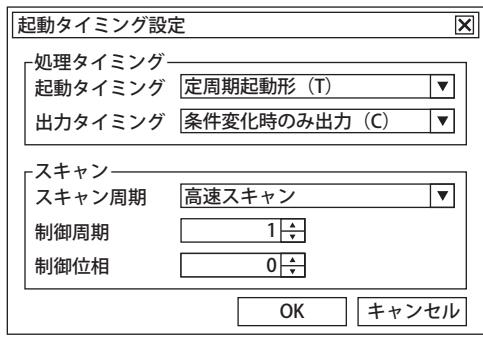


図 起動タイミングの設定

■ 条件判定の機能ブロックワンショット起動処理

バーチャルテスト機能で FCS0121 を起動して、シーケンステーブルを動かします。まず、シーケンステーブル ST01_ONESHOT のブロックモードを AUT にします。

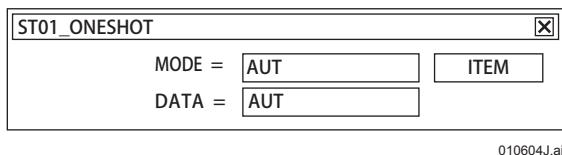


図 シーケンステーブルのブロックモードの変更

条件判定でワンショット起動される汎用演算形ユーザカスタムブロック CMA01_COND の動作から確認します。CMA01_COND のデータアイテム I01 が 0 であることを確認します（0 でなければ 0 を設定します）。

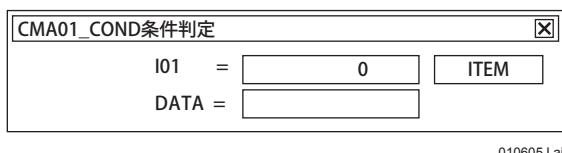


図 条件判定の確認

そして、I01 に「1」を設定します。

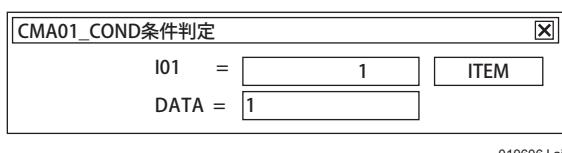


図 条件判定の設定

オペガイドメッセージ %OG0001 が出力されているのを確認してください。

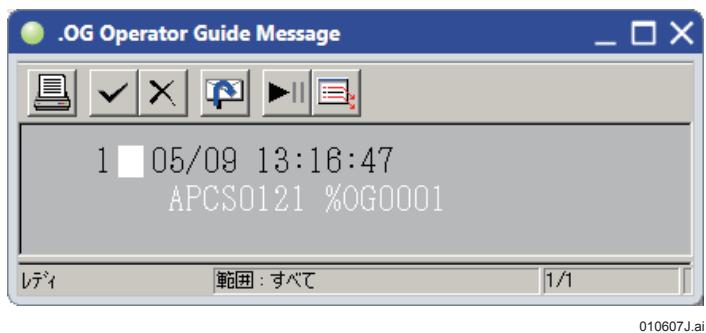


図 オペガイドウィンドウ

汎用演算形ユーザカスタムブロック CMA01_COND のデータアイテム I01 を何度か 0 から 1 に変化させて、オペガイドメッセージが出力されることを確認してください。

● サンプルプログラムの機能ブロックワンショット起動処理

ユーザカスタムアルゴリズム _SMPL_COND の cond.c を確認します。UcaBlockOneshot という文字列で検索し、ソースファイルから以下の部分を見つけてください。

```

/*
* <<FNH>>*****
*
* Function name:          UcaBlockOneshot
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC 处理なし
*                         UCAERR_STOPME 处理続行不能
*
* description:            機能ブロックワンショット起動処理
*
* >>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc,           /* (IN/OUT) : ブロックコンテキスト */
    I32 code,                    /* (IN) : 種別コード */
    I32 parameter,              /* (IN) : パラメータ */
    BOOL *result                /* (OUT) : 実行結果 */
)
{
    I32 i01;
    I32 rtnCode;

    /* 条件判定でなければ、何もしません */
    if (code != UCAONESHOT_CODECOND) {
        return UCAERR_NOPROC;
    }

    /* 条件判定 */
    /* データアイテム I01 が 0 以外なら真、0 なら偽 */
    rtnCode = UcaDataGetIn(bc, &i01, 1, 1, NOOPTION);
    if (i01 != 0) {
        *result = TRUE; /* 真 */
    } else {
        *result = FALSE; /* 偽 */
    }

    return SUCCEED;
}

```

ユーザカスタムブロック実行管理部（システム）は、引数 code に UCAONESHOT_CODECOND（条件判定）または UCAONESHOT_CODEOPRT（状態操作）を指定して、ユーザ定義関数 UcaBlockOneshot を呼び出します。ここでは、条件判定の場合のみ処理を行いますので、最初に code を検査して条件判定でなければ、何もしないで UCAERR_NOPROC でリターンします。

条件判定で呼び出された場合にはデータアイテム I01 の値を取り出し、0 以外なら真 (TRUE)、0 なら偽 (FALSE) を、引数 result が差す実行結果に返します。シーケンステーブル ST01_ONESHOT は、CMA01_COND が真を返すと条件成立時の動作、つまり %OG0001 のオペガイドメッセージを出力します。

■ 状態操作の機能ブロックワンショット起動処理

状態操作の動作を確認します。ウィンドウ名に %SW0201S0121 (S0121 は、ドメイン 1 のステーション 21 を表します) を指定して %SW0201 の計器図を呼び出します。



図 名前入力ダイアログ



図 計器図

計器図で %SW0201.PV を OFF(0)から ON(1)に変更してください。%SW0201.PV を OFF(0)から ON(1)にするごとに、データアイテム P01 が 1 増え、データアイテム P02 は 10 増えることを、汎用演算形ユーザカスタムブロック CMA01_OPRT のチューニングウィンドウで確認してください。

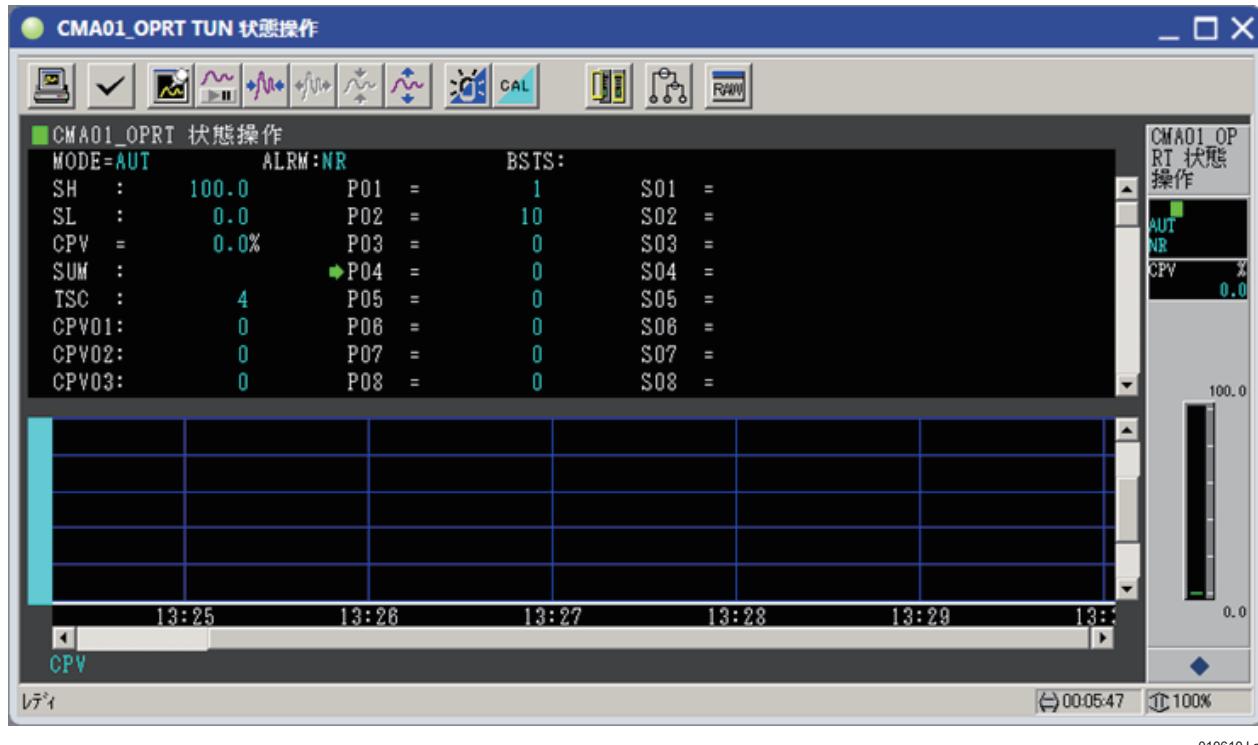


図 CMA01_OPRTのチューニングウィンドウ

同様の手順で、%SW0202.PV を OFF から ON に変更してください。CMA01_OPRT の P01 は 1 ずつ増加し、P02 は 20 ずつ増加します。

● サンプルプログラムの機能ブロックワンショット起動処理

ユーザカスタムアルゴリズム _SMPL_OPRT の oprt.c を確認します。UcaBlockOneshot という文字列で検索し、機能ブロックワンショット起動処理の部分を見つけてください。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockOneshot
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME   処理続行不能
*
* description:  機能ブロックワンショット起動処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc,             /* (IN/OUT): ブロックコンテキスト */
    I32 code,                      /* (IN): 種別コード */
    I32 parameter,                /* (IN): パラメータ */
    BOOL *result                  /* (OUT): 実行結果 */
)
{
    F64S p01;
    F64S p02;
    I32 rtnCode;

    /* 状態操作でなければ何もしません */
    if (code != UCAONESHOT_CODEOPRT) {
        return UCAERR_NOPROC;     /* 処理なし */
    }

    /* 状態操作 */
    /* データアイテム P01,P02 を 1 増加 */
    rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);
    rtnCode = UcaDataGetPn(bc, &p02, 2, 1, NOOPTION);
    p01.value += 1;                /* 1を加えます */
    p02.value += parameter;       /* パラメータ値を加えます */
    rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);
    rtnCode = UcaDataStorePn(bc, &p02, 2, 1, NOOPTION);

    *result = TRUE;               /* いつも真 */
    return SUCCEED;
}
```

ここでは状態操作のみ処理を行いますので、最初に引数 code が UCAONESHOT_CODEOPRT (状態操作) でなければ何もしないでリターンします。

状態操作の場合は、データアイテム P01 に 1 を加えます。また、データアイテム P02 に引数 parameter の値を加えます。引数 parameter には、シーケンステーブル ST01_ONESHOT の [データ] 欄に指定した値が渡されます。たとえば、%SW0201.PV が 0 から 1 に変化したときには、parameter には 10 が渡されます。また、%SW0202.PV が 0 から 1 に変化したときには、parameter には 20 が渡されます。

_SMPL_OPRT は状態操作の処理だけするので、処理が終了すると引数 result が差す実行結果に真 (TRUE) を設定し、SUCCEED でリターンします。状態操作のみの処理であれば、いつもこのように真を設定し SUCCEED でリターンしてください。

2. ユーザカスタムアルゴリズムとC言語

ユーザカスタムアルゴリズムは、ユーザがC言語で作成します。この章では、C言語の特徴と、ユーザカスタムアルゴリズムを記述するときに使用するデータ型と関数について説明します。

■ C言語

C言語はさまざまな分野で使用されている汎用のプログラム言語です。本書に記載されているプログラムとユーザカスタムアルゴリズムのサンプルは、C言語のANSI規格に従って書かれています。

■ データ型と関数

C言語のプログラムは関数と変数から構成されます。C言語の変数は、データ型を持ちます。ユーザカスタムアルゴリズムを記述するプログラマは、変数のデータ型を意識する必要があります。たとえば、F64S型（データステータス付き64ビット浮動小数）のデータを処理する関数には、F64S型の変数を指定する必要があります。また、I32型（32ビット整数）のデータを処理する関数には、I32型の変数を指定する必要があります。CENTUM VP の汎用演算ブロック (CALCU) の演算式や、プロセス制御用プログラム言語SEBOLでは、プログラマはデータ型を強く意識する必要はありませんでした。ユーザカスタムアルゴリズムをC言語で記述する場合、「データ型を強く意識する」ということが、演算式やSEBOLのプログラミングと異なる部分です。

C言語のプログラムは関数から構成されます。システムは、ユーザがユーザカスタムアルゴリズムを記述するために、2つの支援をしています。1つはユーザが記述する関数（ユーザカスタムアルゴリズム）のサンプルを提供していることです。ユーザカスタムブロック開発環境パッケージには、ユーザカスタムアルゴリズムのサンプルが含まれています。ユーザはこれらのサンプルのコピーに独自の修正を追加することで、ユーザ固有のアルゴリズムを記述することができます。サンプルの内容は、本書で説明しています。

もう1つの支援は、ユーザカスタムアルゴリズム作成用ライブラリです。これは入出力端子アクセス、自ブロックデータアクセスなどの関数群です。ユーザはユーザカスタムアルゴリズム作成用ライブラリの関数を組み合わせることにより、ユーザカスタムアルゴリズムを作成することができます。

■ Visual C++

ユーザカスタムアルゴリズムは、Microsoft 社の Visual Studio の C/C++ プログラム開発環境である Visual C++ で開発します。

ユーザカスタムアルゴリズムは、C 言語の機能のみを使用し、C++ の機能は使用しないで記述します。クラスなどの C++ の機能は使用できません。つまり、ユーザカスタムアルゴリズムの開発では、Visual C++ を「C 言語の開発環境」としてのみ使用します。Visual C++ では、C のソースファイルは file.c (サフィックスが .c)、C++ のソースファイルは file.cpp(サフィックスが .cpp) としますが、ユーザカスタムアルゴリズムのソースファイルは file.c とサフィックスを「.c」で作成します。

● Visual C++のバージョン

使用する CENTUM VP の レビジョンでサポートしていないバージョンの Visual C++ でコンパイルしたユーザカスタムアルゴリズムを使用できません。

たとえば、Visual Studio 2008 でビルドしたユーザカスタムアルゴリズムを CENTUM VP R6.06 では使用できません。使用したい場合は CENTUM VP R6.06 の開発環境を構築し、Visual Studio 2017 で再ビルドしたユーザカスタムアルゴリズムを再登録してください。CENTUM VP R6.06.00 以降では、Visual Studio 2017 でビルドしたユーザカスタムアルゴリズムを使用してください。

● Visual Studioの操作方法

Visual Studio の操作方法が MSDN オンラインで参照できます。ユーザカスタムアルゴリズムを開発するときはこれらのドキュメントを参照できるようにしておくことを推奨します。

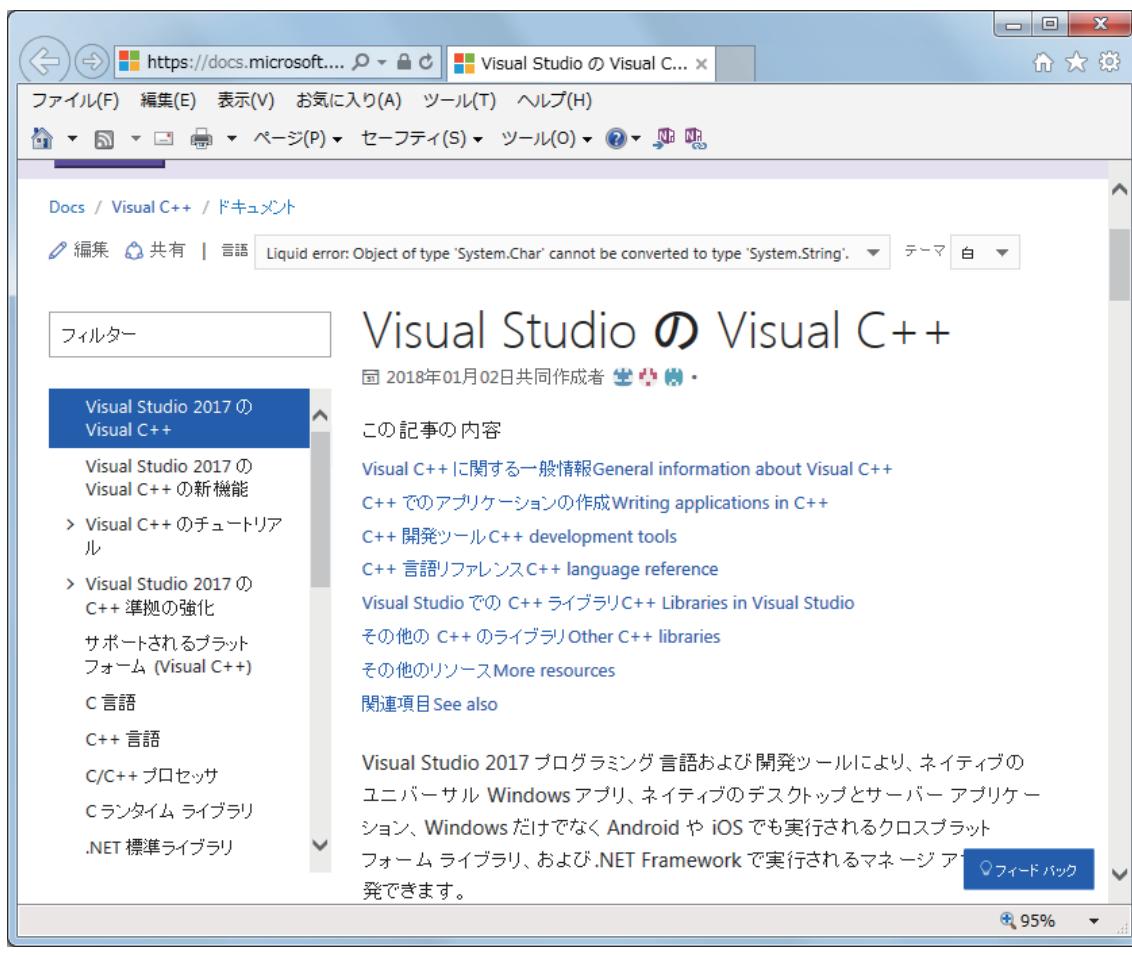


図 Visual Studio 2017 ドキュメント (MSDNオンライン)

2.1 データ型

ユーザカスタムアルゴリズムで機能ブロックデータ（自ブロックを含む）を扱うためのデータ型が用意されています。連続制御形ユーザカスタムブロックのデータアイテムで例を示します。

- データアイテムPVのデータ型はF64Sです。F64Sは、データステータス付き64ビット浮動小数を表します。
- データアイテムI01のデータ型はI32です。I32は、符号付き32ビット整数を表します。

ユーザカスタムアルゴリズムで、機能ブロックデータを扱う場合には、long や double などの C 言語の形宣言子は使用しないでください。代わりに、「表 ユーザカスタムアルゴリズムのプログラミングに使用するデータ型」のデータ型でプログラミングします。自ブロックのデータアイテム P01, P02, I01 を格納するための変数（C 言語の自動変数）を宣言する例を示します。

```

UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc           /* (IN/OUT): 実行コンテキスト */
)
{
    F64S p01;      /* データアイテム P01 の値を格納する、F64S 型の変数 p01 を宣言 */
    F64S p02;      /* データアイテム P02 の値を格納する、F64S 型の変数 p02 を宣言 */
    I32 i01;       /* データアイテム I01 の値を格納する、I32S 型の変数 i01 を宣言 */

    /* データアイテム P01, P02, I01 を自動変数 p01, p02, i01 に読みこむ */
    rtnode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);
    rtnode = UcaDataGetPn(bc, &p02, 2, 1, NOOPTION);
    rtnode = UcaDataGetIn(bc, &i01, 1, 1, NOOPTION);

    .....
    return SUCCEED;
}

```

次表にユーザカスタムアルゴリズムのプログラミングに使用するデータ型の一覧を示します。これらのデータ型は、システム定義インクルードファイル libucadatatype.h で定義されています。

参照

- ユーザカスタムブロックのデータアイテムのデータ型の詳細については、以下を参照してください。
[APCS ユーザカスタムブロック \(IM 33J15U20-01JA\)](#)
- システム定義インクルードファイルの詳細については、以下を参照してください。
[「2.2 システム定義インクルードファイル」](#)

表 ユーザカスタムアルゴリズムのプログラミングに使用するデータ型（1/2）

データ型	VC++のC言語における、データ型に対応する形宣言	サイズ	内容
BYTE	signed char	1バイト	符号付き8ビット整数
I8	signed char	1バイト	符号付き8ビット整数
U8	unsigned char	1バイト	符号なし8ビット整数
I16	signed short	2バイト	符号付き16ビット整数
I32	signed long	4バイト	符号付き32ビット整数
U16	unsigned short	2バイト	符号なし16ビット整数
U32	unsigned long	4バイト	符号なし32ビット整数
F32	float	4バイト	32ビット浮動小数点数
F64	double	8バイト	64ビット浮動小数点数
I16S	struct { I16 value; BYTE dummy[2]; U32 status; };	8バイト	データステータス付き 符号付き16ビット整数
I32S	struct { I32 value; U32 status; };	8バイト	データステータス付き 符号付き32ビット整数
U16S	struct { U16 value; BYTE dummy[2]; U32 status; };	8バイト	データステータス付き 符号なし16ビット整数
U32S	struct { U32 value; U32 status; };	8バイト	データステータス付き 符号なし32ビット整数
F32S	struct { F32 value; U32 status; };	8バイト	データステータス付き 32ビット浮動小数点数

表 ユーザカスタムアルゴリズムのプログラミングに使用するデータ型（2/2）

データ型	VC++のC言語における、データ型に対応する形宣言	サイズ	内容
F64S	struct { F64 value; U32 status; BYTE dummy[4]; };	16 バイト	データステータス付き 64 ビット浮動小数点数
F32SR	struct { F32 value; U32 status; F32 scaleHi; F32 scaleLow; };	16 バイト	データステータス／レンジ付き 32 ビット浮動小数点数
UcaUnivType	struct { U16 indOpt; I16 rqstCode; U16 iReturn; U8 dType; U8 dSize; union { I16 sint16; U16 uint16; I32 sint32; U32 uint32; F32 float32; F64 float64; BYTE string[16]; I16S sint16s; U16S uint16s; I32S sint32s; U32S uint32s; F32S float32s; F64S float64s; F32SR float32sr; } dataValue; };	24 バイト	任意のデータ型のデータを格納する構造体です。共用体 dataValue に格納されているデータのデータ型は、メンバ dType に保持します。メンバ dSize は、共用体 dataValue に格納されているデータのサイズで dType に対応します。indOpt,rqstCode,iReturn は、システム専用のメンバです。ユーザプログラムでは使用しないでください。
TAGN	BYTE TAGN[16];	16 バイト	データアイテム名
DITMN	BYTE DITMN[8];	8 バイト	データアイテム名

「表 ユーザカスタムアルゴリズムのプログラミングに使用するデータ型」のデータ型 I16S、F64S などに dummy という構造体メンバがありますが、これは各構造体のメンバの整合を取るために入れてあります。ユーザプログラムでは、メンバ dummy を使用しないでください。

これまで説明してきた F64S や I32 などのデータ型は、変数を宣言するときに使用します。これとは別に、形を表わすラベルが用意されています。任意のデータ型 UcaUnivType の共用体メンバ dataType のデータ型を示すコードは、構造体メンバ dType にこのラベルで定義されたコードで格納されます。また、このラベルはデータ型を変換するユーザカスタムアルゴリズム作成用ライブラリ UcaDataConvertType に指定します。

参照 使用例の詳細については、以下を参照してください。

[「1.5 機能ブロックデータ設定時特殊処理」](#)

データ型ごとのラベルを以下に示します。

表 データ型とラベル

データ型	ラベル	説明
I16	UCA_DATATYPE_I16	2 バイト符号付き整数
U16	UCA_DATATYPE_U16	2 バイト符号なし整数
I32	UCA_DATATYPE_I32	4 バイト符号付き整数
U32	UCA_DATATYPE_U32	4 バイト符号なし整数
F32	UCA_DATATYPE_F32	単精度浮動小数
F64	UCA_DATATYPE_F64	倍精度浮動小数
BYTE string[16]	UCA_DATATYPE_CHR	文字列
I16S	UCA_DATATYPE_I16S	データステータス付 2 バイト符号付き整数
I32S	UCA_DATATYPE_I32S	データステータス付 4 バイト符号付き整数
U32S	UCA_DATATYPE_U32S	データステータス付 4 バイト符号なし整数
F32S	UCA_DATATYPE_F32S	データステータス付 単精度浮動小数
F64S	UCA_DATATYPE_F64S	データステータス付 倍精度浮動小数
F32SR	UCA_DATATYPE_F32SR	データステータス／レンジ付 単精度浮動小数
DITMN	UCA_DATATYPE_DITM	データアイテム名

これらのラベルは、システム定義インクルードファイル libucadef.h で定義されています。

参照 システム定義インクルードファイルの詳細については、以下を参照してください。

[「2.2 システム定義インクルードファイル」](#)

■ 変数の宣言とVisual Studioの警告（warning）

ユーザカスタムアルゴリズム作成用ライブラリの関数呼び出しに指定する変数のデータ型が関数の呼び出し形式と合わない場合、Visual Studioはリビルド時に警告（warning）を検出します。

サンプルを使用して実際に警告を出してみます。作業フォルダの以下の位置に警告を出すソリューション _SMPL_WARNING があります。

<ドライブ名>:\UcaWork\UcaSamples_\SMPL_WARNING

Visual Studio を起動し、_SMPL_WARNING_\SMPL_WARNING.sln を開きます。

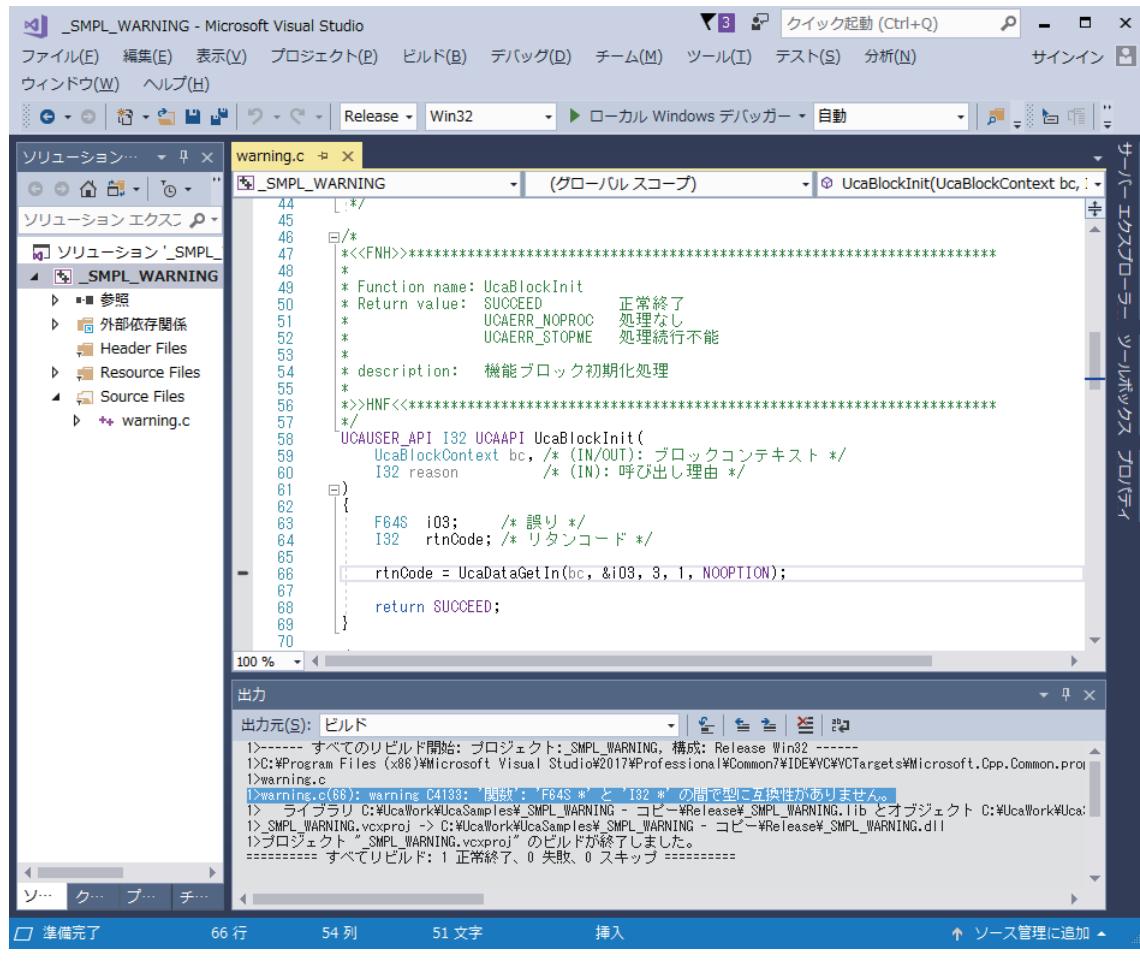


図 警告メッセージの出力

Visual Studio の [ビルド] メニューの [リビルド] で Release 版をリビルドします。出力ウィンドウに警告（warning C4133）メッセージが出力されます。「warning C4133」の部分をダブルクリックすると、右上のペインに該当のソースコードが表示されます。

「warning C4133」は、以下の行で検出されています。

```

.....
F64S i03;      /* 誤り */
I32 rtnCode;   /* リターンコード */

rtnCode = UcaDataGetIn(bc, &i03, 3, 1, NOOPTION); ←———— 警告が検出される行
.....

```

ユーザカスタムブロックのデータアイテム I03 のデータ型は、I32 (32ビット整数) です。しかし、このプログラムでは C 言語の変数 i03 を F64S 型で宣言しています (*誤り*) のコメントの部分)。該当行のユーザカスタムアルゴリズム作成用ライブラリ UcaDataGetIn の「&i03, 3, 1」の部分は、「データアイテム I01 ~ I32 の 3 番目から 1 つ分を C 言語の変数 i03 に格納する」という意味です。しかし、データ I01 ~ I32 は I32 型であり、変数 i03 は F64S 型で宣言されていてデータ型が合わないため、警告 (warinign C4133) が検出されました。

警告を取り除いてみます。i03 の宣言を以下のように「F64S から I32」に修正してください。

```
F64S i03; /* 誤り */
```

↓
修正

```
I32 i03; /* 正解 */
```

修正したら、もう一度リビルドしてください。今度は、警告が出ず Visual Studio のメッセージの最下行が「エラー 0、警告 0」となります。

警告を検出しても、Visual Studio はリビルドを継続し実行形式 (DLL) を作成します (エラーなら実行形式は作成しません)。しかし、警告を含んだプログラムは、誤りの可能性が大きいので「警告」されているのです。「警告」は決して無視しないでください。ユーザカスタムアルゴリズムを作成するときには、すべての警告を取り除いてください。

重要

ユーザカスタムアルゴリズムを作成する場合は、Visual Studio のデフォルトの警告レベル 3 (*1)において、すべての警告を無くしてください。

*1： 警告レベルは、[プロジェクト] メニューの [<プロジェクト名> のプロパティ] であらわれる「プロパティ ページ」ダイアログで指定します。「警告レベル」のデフォルトは「レベル 3」です。ユーザカスタムアルゴリズムを作成する場合には、デフォルトの「レベル 3」で全ての警告を取り除いてください。

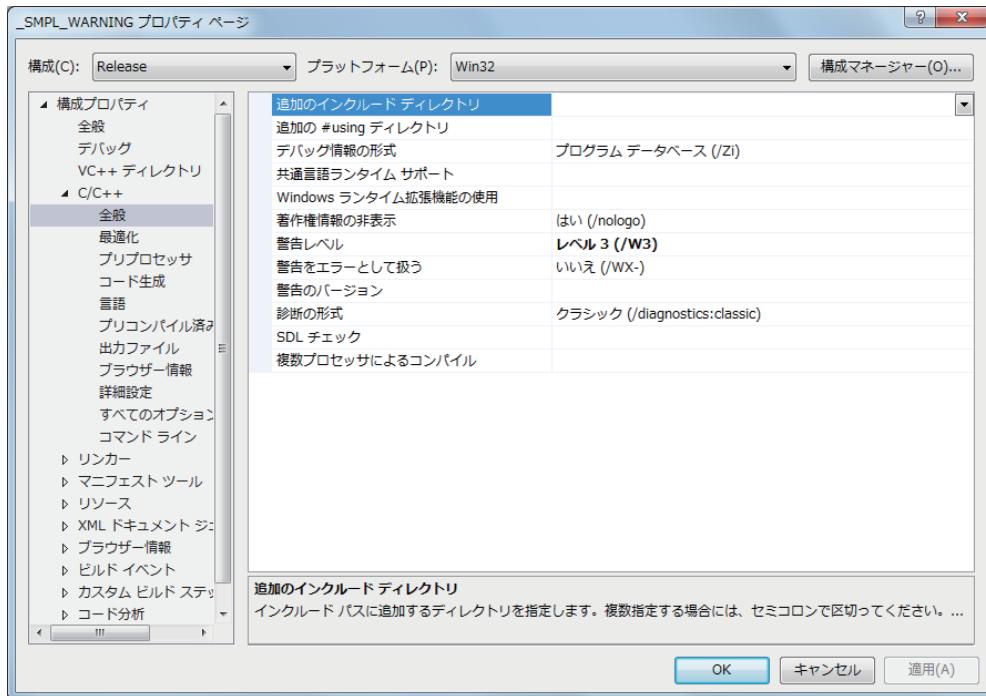


図 プロパティページダイアログ

020107.ai

参照 ユーザカスタムアルゴリズムのプロジェクトの設定の詳細については、以下を参照してください。
APCS ユーザカスタムブロック開発環境 (IM 33J15U23-01JA)

2.2 システム定義インクルードファイル

システム定義インクルードファイル (*1) は、ユーザカスタムアルゴリズムをC言語で記述するときに使用するデータ型 (F64SやI32など) やラベル (UCAERR_NOPROCのような名前)、およびユーザカスタムアルゴリズム作成用ライブラリの関数の呼び出し形式を定義しています。ユーザプログラムで、システム定義インクルードファイルlibuca.hを#include行で取り込むことにより、F64SやI32などのデータ型を使用できるようになります。

この節では、システム定義インクルードファイルの使い方と、それぞれのインクルードファイルの記述内容について説明します。

*1: 「インクルードファイル」を「ヘッダファイル」と呼ぶこともあります。本書では、VC++ の用語に合わせて「インクルードファイル」と記述しています。

システム定義インクルードファイルは、ユーザカスタムブロック開発環境パッケージに含まれていて、以下のフォルダにインストールされます。

<CENTUM VP インストール先> ¥UcaEnv¥include¥ システム定義インクルードファイル

「1.1 ユーザカスタムブロックを動かすための準備、● Visual C++ のディレクトリパス」で上記のフォルダを指定しました。VC++ で次のように設定しました。

パスの先頭に C:¥CENTUMVP¥UcaEnv¥include を設定しています。

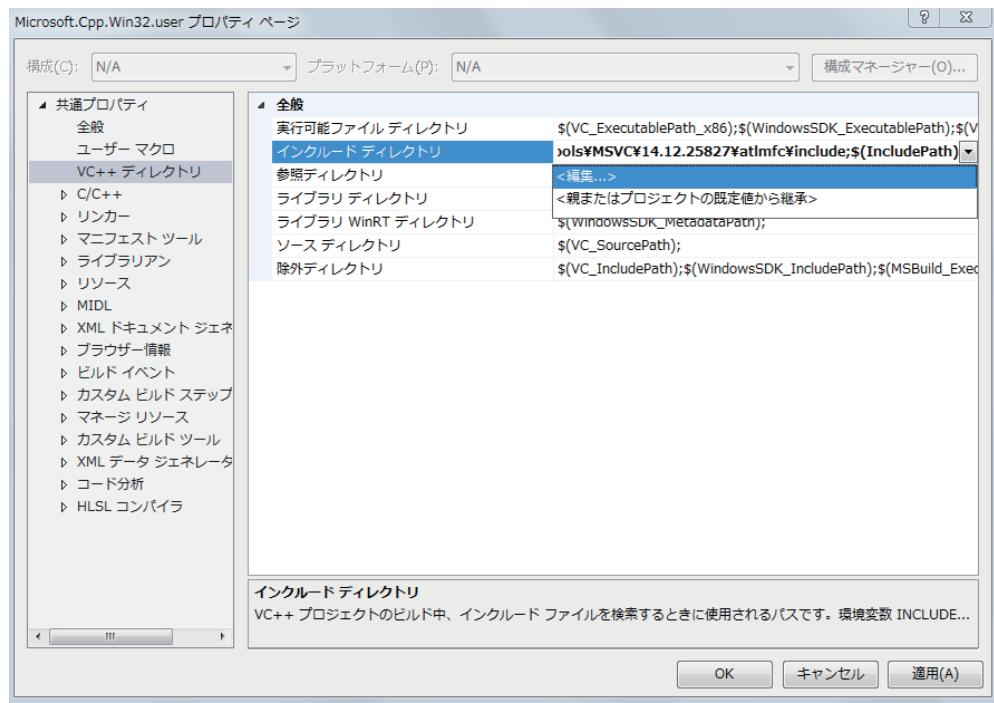


図 プロパティページダイアログ

1. ダイアログの左ペインの、[共通プロパティ] – [VC++ ディレクトリ] を選択してください。
2. 右ペインの [全般] – [インクルードディレクトリ] を選択し、[編集] をクリックしてください。
インクルードディレクトリダイアログが表示されます。

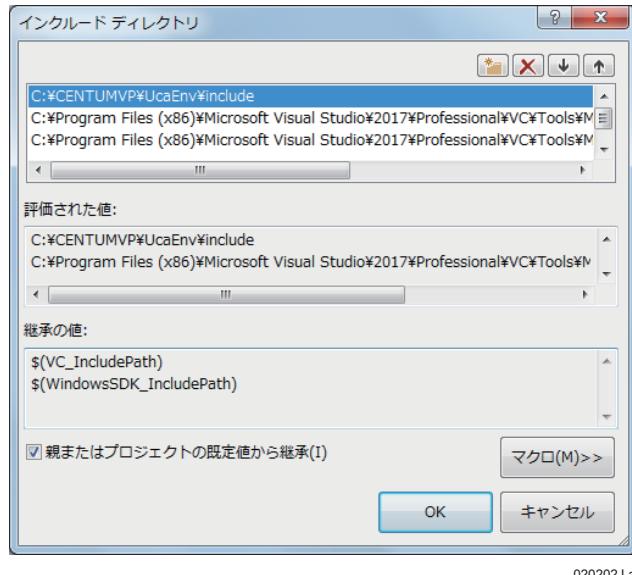


図 インクルードディレクトリダイアログ

システム定義インクルードファイルの一覧を以下に示します。

表 システム定義インクルードファイル

ファイル名	内容
libuca.h	システム定義インクルードファイルを代表します。ユーザプログラムで直接 #include <libuca.h> と記述し、ファイルを取り込みます。
libucadef.h	ロックステータス、アラームステータス、データステータスのマスクパターンなどを定義しています。
libucadatatype.h	F64S、I32Sなどのデータ型を宣言しています。
libucaproto.h	ユーザカスタムアルゴリズム作成用ライブラリ（ソース非公開の Uca***）の関数プロトタイプ宣言です。
libucaerr.h	エラーコードを定義しています。
libucauserfunc.h	UcaBlockInit()、UcaBlockPeriodical() など、ユーザ定義関数に関する定義です。

libuca.h はシステム定義インクルードファイルを代表します。以下のように、すべてのユーザ定義インクルードファイルを #include しています。

libuca.h の内容

```
.....
#include <libucadef.h>
#include <libucadatatype.h>
#include <libucaproto.h>
#include <libucaerr.h>
#include <libucauserfunc.h>
.....
```

上記のように libuca.h が他のシステム定義インクルードファイルと取り込んでいますので、ユーザが作成するユーザカスタムアルゴリズムでは、libuca.h のみを #include でインクルードします。

ユーザプログラムにおけるシステム定義インクルードファイルの #include

```
.....
/*********************************************
 * CENTUM VP システムインクルードファイル
 */
#include <libuca.h>      /* ユーザカスタムアルゴリズム作成用ライブラリ */

/*********************************************
.....
```

libuca.h を #include 行で取り込むことにより、F64S や I32 などの型名や、UcaDataStorePn などのユーザカスタムアルゴリズム作成用ライブラリの関数呼び出しをユーザプログラムに記述できるようになります。

3. ユーザカスタムアルゴリズム

ユーザカスタムアルゴリズムは、ユーザがC言語でプログラミングします。ユーザカスタムアルゴリズムを作成するためのライブラリをシステムが用意しています。このライブラリをユーザカスタムアルゴリズム作成用ライブラリと呼びます。入出力端子アクセス、自ブロックデータアクセス、機能ブロックデータアクセス、アラーム処理などを行う関数が用意されています。ユーザは、これらの関数を使用し、独自のアルゴリズムをユーザカスタムアルゴリズムとして作成します。

■ ユーザカスタムブロックとユーザカスタムアルゴリズム

ユーザカスタムブロックとユーザカスタムアルゴリズムの関係を以下に示します。ユーザカスタムブロックは、基本制御機能から標準ブロック（シーケンステーブルブロック、PID調節ブロックなど）と同じように処理タイミングを与えられます。ユーザカスタムブロックに処理タイミングが与えられると、ユーザカスタムブロック実行管理部は、ユーザが作成したユーザカスタムアルゴリズムの関数を呼び出します。ユーザ定義の関数は、システムにより提供されるユーザカスタムアルゴリズム作成用ライブラリを組み合わせて実現されています。

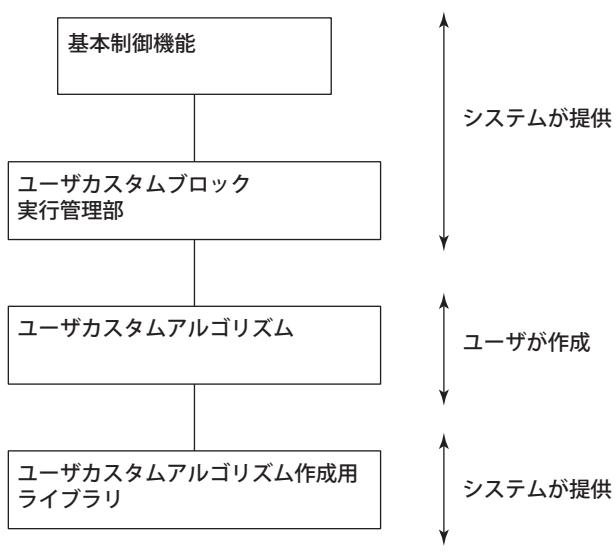


図 ユーザカスタムブロックとユーザカスタムアルゴリズムの関係

■ ユーザカスタムアルゴリズムの5つの入り口

ユーザカスタムアルゴリズムは、機能ブロック初期化処理 (UcaBlockInit)、機能ブロック終了処理 (UcaBlockFinish)、機能ブロック定周期処理 (UcaBlockPeriodical)、機能ブロックワンショット起動処理 (UcaBlockOneshot)、機能ブロックデータ設定時特殊処理 (UcaBlockDset) の5つの入り口をもつCプログラムです。ユーザカスタムアルゴリズムのプログラム名称をユーザカスタムブロックのビルダ定義項目に指定して、ユーザ記述のCプログラムを実行します。

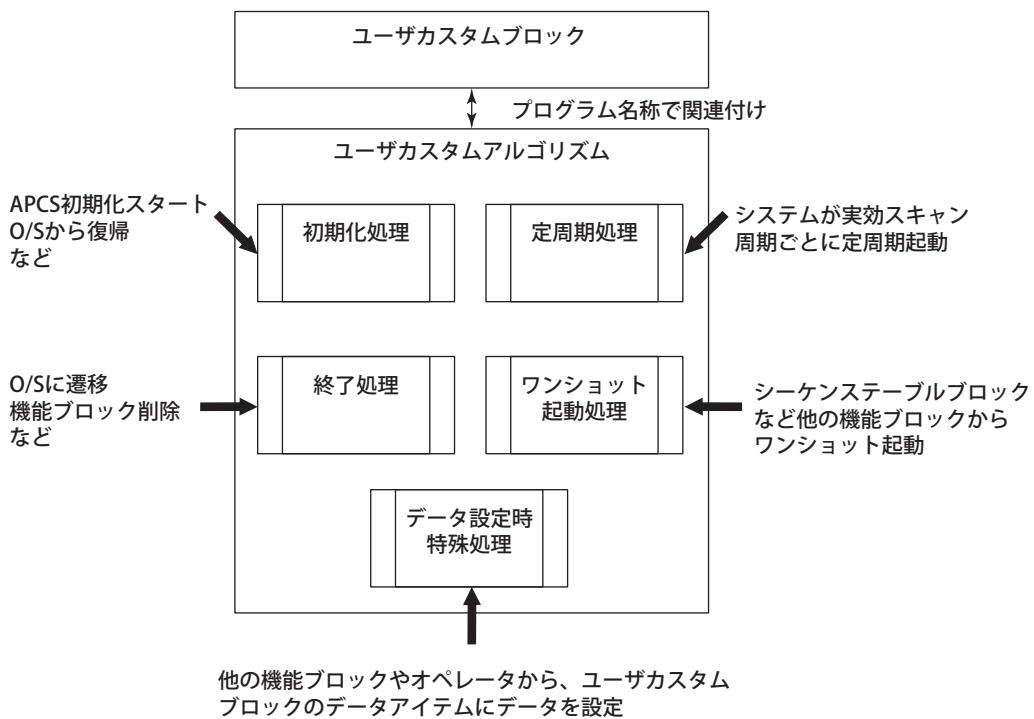


図 ユーザカスタムアルゴリズムの5つの処理

030002J.ai

ユーザカスタムアルゴリズムは、次の形式の C プログラムです。

```
#include <libuca.h>           /* ユーザカスタムアルゴリズム作成用ライブラリ */

UCAUSER_API I32 UCAAPI UcaBlockInit(
    UcaBlockContext bc,      /* (IN/OUT): ブロックコンテキスト */
    I32 reason              /* (IN): 呼び出し理由 */
)
{
    /* 機能ブロック初期化処理を記述 */
    return SUCCEED;
}

UCAUSER_API I32 UCAAPI UcaBlockFinish(
    UcaBlockContext bc,      /* (IN/OUT): ブロックコンテキスト */
    I32 reason              /* (IN): 呼び出し理由 */
)
{
    /* 機能ブロック終了処理を記述 */
    return SUCCEED;
}

UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc       /* (IN/OUT): ブロックコンテキスト */
)
{
    /* 機能ブロック定周期処理を記述 */
    return SUCCEED;
}

UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc,      /* (IN/OUT): ブロックコンテキスト */
    I32 code,                /* (IN): 種別コード */
    I32 parameter,           /* (IN): パラメータ */
    BOOL *result             /* (OUT): 実行結果 */
)
{
    /* 機能ブロックワンショット起動処理を記述 */
    return SUCCEED;
}

UCAUSER_API I32 UCAAPI UcaBlockDset(
    UcaBlockContext bc,      /* (IN/OUT): ブロックコンテキスト */
    DITMN itemName,          /* (IN): データアイテム名 */
    UcaUnivType *data,       /* (IN): 設定データ */
    UcaDataOrStatus dataOrStatus, /* (IN): 設定種別 */
    UcaPreOrPost preOrPost,   /* (IN): 呼び出し種別 */
    BOOL *result             /* (OUT): 設定可否 */
)
{
    /* 機能ブロックデータ設定時特殊処理を記述 */
    return SUCCEED;
}
```

ユーザカスタムアルゴリズムには、5つの関数すべての定義が必要です。たとえば、機能ブロック終了処理が必要ない場合でも、UcaBlockFinish 関数が必要です。処理が必要ない関数は、UCAERR_NOPROC でリターンします。

3.1 ユーザ定義関数の一般形

機能ブロック定周期処理を例に、ユーザ定義関数の一般形を示します。

■ 何らかの処理をする場合

```
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc           /* (IN/OUT): ブロックコンテキスト */
)
{
    /* 機能ブロック定周期処理を記述 */
    .....
    return SUCCEED;
}
```

■ 処理をしない場合

```
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc           /* (IN/OUT): ブロックコンテキスト */
)
{
    return UCAERR_NOPROC;       /* 処理なし */
}
```

ユーザ定義関数で何らかの処理をする場合には、SUCCEED でリターンし、ユーザカスタムブロック実行管理部（システム）に正常に処理したことを通知します。また、何も処理しない場合には、UCAERR_NOPROC でリターンし、処理をしていないことを通知します。ユーザ定義関数が UCAERR_NOPROC でリターンする場合でも、何か処理を記述すると、その処理自体は有効になります。

```
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc           /* (IN/OUT): ブロックコンテキスト */
)
{
    /* 誤り：ここに処理を書けば、その処理は有効になります */
    .....
    return UCAERR_NOPROC;       /* 処理なし */
}
```

しかし、システムは UCAERR_NOPROC でリターンされると「何もしないで戻ってきた」と判断してしまいますので、処理をした場合は SUCCEED、処理をしない場合は UCAERR_NOPROC でリターンしてください。

参照 ユーザ定義関数の戻り値のシステムによる扱いの詳細については、以下を参照してください。
[「3.7 ユーザ定義関数の戻り値」](#)

3.2 機能ブロック初期化処理

機能ブロック初期化処理は、ユーザカスタムブロックが動き始めるときに、ブロック自身を初期化するための処理です。システムは、APCSの初期化スタートや、ユーザカスタムブロックのブロックモードがO/Sから復帰するときに、機能ブロック初期化処理を呼び出します。

機能ブロック初期化処理の基本的な記述を以下に示します。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockInit
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME    処理続行不能
*
* description:  機能ブロック初期化処理
*
* >>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockInit(
    UcaBlockContext bc,          /* (IN/OUT): ブロックコンテキスト */
    I32 reason                 /* (IN): 呼び出し理由 */
)
{
    /* 機能ブロック初期化処理を記述 */
    return SUCCEED;
}
```

1番目の引数 `bc` に指定されるブロックコンテキストは、システムがユーザカスタムブロックの管理情報を保持する領域を差すポインタです。ユーザのプログラムが直接ブロックコンテキストを参照することはありません。ユーザは、ユーザカスタムブロック作成用ライブラリを呼び出すときに、引数にブロックコンテキスト（変数 `bc`）をそのまま指定します。

2番目の引数 `reason` は、呼び出し理由です。つまり、APCS 初期化スタートやブロックモード O/S からの復帰など、機能ブロック初期化処理が呼び出されたときの状況がシステムから渡されます。呼び出し理由の一覧を次表に示します。

表 機能ブロック初期化処理の呼び出し理由

ラベル	呼び出し理由
UCAINIT_START	APCS 初期化スタート
UCAINIT_NEW	制御ドローイングビルダより、オンラインでユーザカスタムブロックを追加
UCAINIT_PRG	機能ブロック詳細ビルダより、オンラインで「プログラム名称」を変更 システムビューより、オンラインでユーザカスタムアルゴリズムをダウンロード
UCAINIT_MAINT	機能ブロック詳細ビルダより、オンラインで「プログラム名称」以外を変更
UCAINIT_AWAKEN	外部からのデータ設定によりブロックモード（MODE）がO/Sから復帰

補足

- ・ ユーザカスタムブロックのブロックモードがO/SのときにAPCS制御機能を停止・再起動した場合には、APCS初期化スタートで機能ブロック初期化処理は呼び出されません。つまり、APCS初期化スタート時にブロックモードがO/Sであれば、機能ブロック初期化処理は呼び出されません。
- ・ ユーザカスタムブロックのブロックモードがO/Sのとき、ユーザカスタムブロックをオンラインで修正しても、機能ブロック初期化処理は呼び出されません。
- ・ ユーザカスタムブロックのブロックモードがO/Sのとき、ユーザカスタムブロックのプログラム名称に指定されているユーザカスタムアルゴリズムをオンラインで修正しても、機能ブロック初期化処理は呼び出されません。

機能ブロック初期化処理は、ユーザカスタムブロックが動き始めるときに、ブロック自身を初期化するための処理です。機能ブロック初期化処理では自ブロックのデータにのみアクセス可能であり、他の機能ブロックや入出力端子へのアクセスはできません。つまり、自ブロックデータにアクセスするユーザカスタムアルゴリズム作成用ライブラリのみ使用可能であり、他の機能ブロックデータや入出力端子にアクセスするユーザカスタムアルゴリズム作成用ライブラリは使用できません。

参照

機能ブロック初期化処理で使用可能なユーザカスタムアルゴリズム作成用ライブラリの詳細については、以下を参照してください。

「[3.8 ユーザカスタムアルゴリズム作成用ライブラリー一覧](#)」

● ブロックモードがO/Sから復帰するときの動作

ユーザカスタムブロックのブロックモードがO/Sから復帰（O/SからO/S以外に遷移）するときの動作詳細について説明します。この場合、ブロックモード変更指令に対する機能ブロックデータ設定時特殊処理と機能ブロック初期化処理の両方が実行されます。ブロックモードがO/Sから復帰するときのシステムの動作を以下に示します。

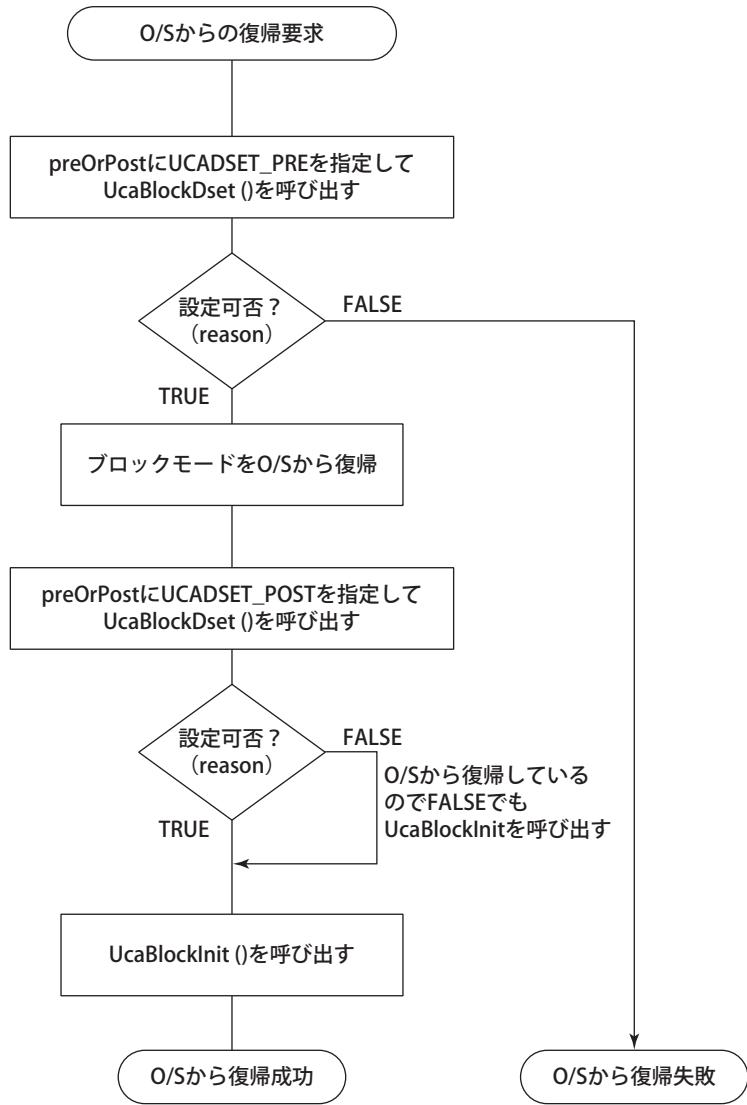


図 ブロックモードがO/Sから復帰するときのシステムの動作

030202J.ai

APCS 初期化スタートとオンラインでのユーザカスタムブロック追加時には、ブロックモードの O/S からの復帰に対する機能ブロックデータ設定時特殊処理は実行されません。機能ブロック初期化処理のみが実行されます。

表 機能ブロック初期化処理

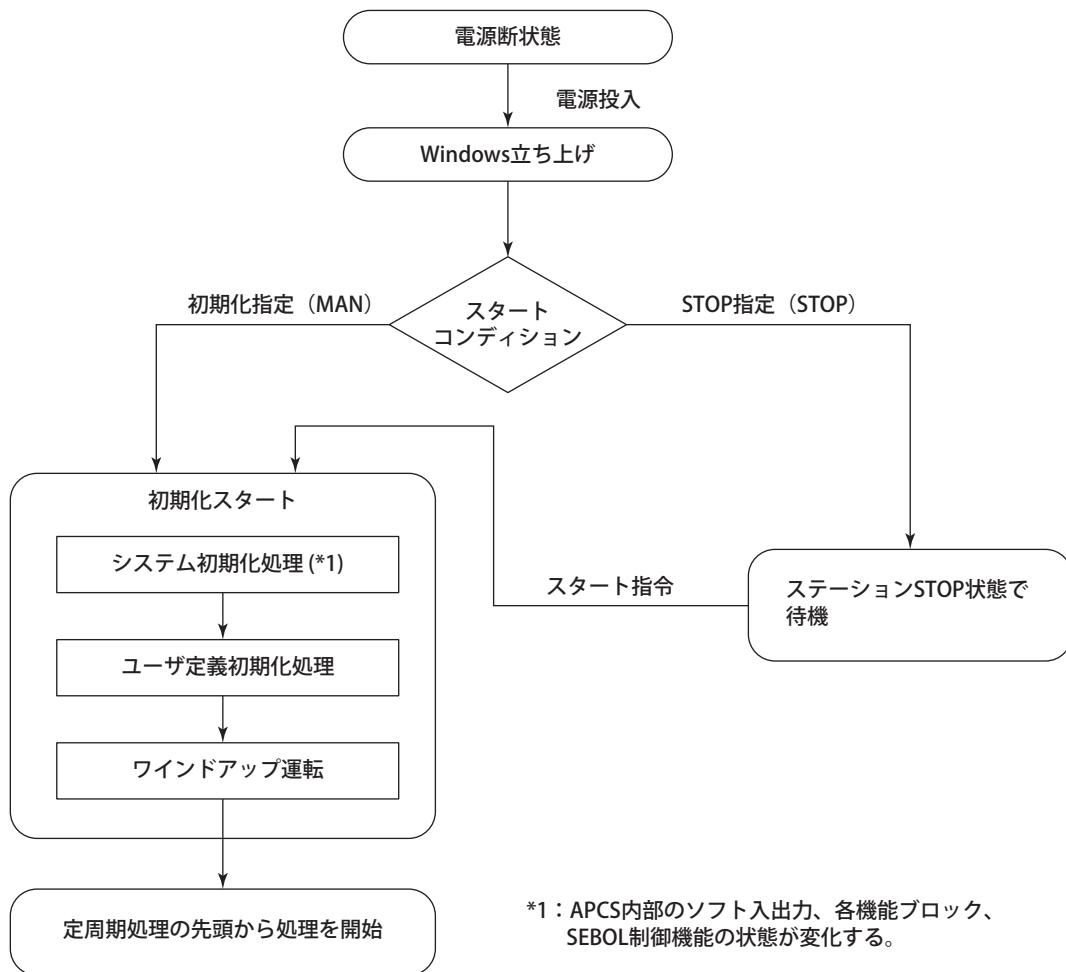
ラベル	呼び出し理由	MODEがO/Sから復帰することに対する 機能ブロックデータ設定時特殊処理
UCAINIT_START	APCS 初期化スタート	実行しない
UCAINIT_NEW	制御ドローイングビルダより、オンラインでユーザカスタムブロックを追加	実行しない
UCAINIT_PRG	機能ブロック詳細ビルダより、オンラインで「プログラム名称」を変更	実行する
	システムビューより、オンラインでユーザカスタムアルゴリズムをダウンロード	実行する
UCAINIT_MAINT	機能ブロック詳細ビルダより、オンラインで「プログラム名称」以外を変更	実行する
UCAINIT_AWAKEN	外部からのデータ設定によりブロックモード(MODE) が O/S から復帰	実行する

参照

- オンラインによる修正に伴う機能ブロック初期化処理の実行タイミングの詳細については、以下を参照してください。
[「3.3.1 オンラインの修正に伴う機能ブロック初期化処理／終了処理」](#)
- ブロックモード変更指令に対する機能ブロックデータ設定時特殊処理の詳細については、以下を参照してください。
[「3.6.1 CSTM-C のブロックモードを AUT と O/S に限定」](#)

■ APCS初期化スタートに伴う処理

APCSの初期化スタート動作を以下に示します。ユーザカスタムブロックは、初期化スタートの「システム初期化処理」の一部として呼び出されます。



030204J.ai

図 APCSのスタート動作の流れ

参照 APCS 初期化スタートの詳細については、以下を参照してください。

[APCS \(IM 33J15U10-01JA\)](#)

システムは、APCS 初期化スタート時、APCS 内のすべての機能ブロックを初期化します。これがシステム初期化処理です。ユーザカスタムブロックの機能ブロック初期化処理は、このシステム初期化処理の一部として呼び出されます。システムは、APCS 内のすべてのユーザカスタムブロックに対し、機能ブロック初期化処理を呼び出します。

機能ブロック初期化処理では、自ユーザカスタムブロックのデータにのみアクセス可能です。他の機能ブロックデータにアクセスすることは禁止されています。

参照 機能ブロック初期化処理で使用可能なユーザカスタムアルゴリズム作成用ライブラリの詳細については、以下を参照してください。

[「3.8 ユーザカスタムアルゴリズム作成用ライブラリー一覧」](#)

APCS 初期化スタート時に、ユーザカスタムアルゴリズムに記述したプログラムで他の機能ブロックのデータを初期化したい場合には、以下の手順でアプリケーションを作成します。

- ・ 実行したい処理を機能ブロックワンショット起動処理に記述します。
- ・ プログラムを記述したユーザカスタムアルゴリズムをユーザカスタムブロックのプログラム名称に指定します。
- ・ ユーザカスタムブロックを、I 形または B 形のユーザ定義初期化処理用シーケンステーブルからワンショット起動します。つまり、ユーザカスタムブロックを、ユーザ定義初期化処理の一部としてシーケンステーブルからワンショット起動します。ユーザ定義初期化処理内で実行するユーザカスタムブロックの機能ブロックワンショット起動処理 (UcaBlockOneshot) では、他の機能ブロックのデータにアクセス可能です。
- ・ 「ユーザ定義初期化処理」をするユーザカスタムアルゴリズムの機能ブロック定周期処理で、誤って「ユーザ定義初期化処理」と同じ処理をしないように注意してください。安全のために、機能ブロック定周期処理に何も処理を記述しないか、またはビルダ定義項目の起動タイミングは「ワンショットのみ」としてください。

参照 起動タイミングの詳細については、以下を参照してください。

[「3.5.1 ビルダ定義項目「起動タイミング」」](#)

- ・ 「ユーザ定義初期化処理」でワンショット起動するユーザカスタムブロックとユーザカスタムアルゴリズムは、「ユーザ定義初期化処理」専用とすることを推奨します。「ユーザ定義初期化処理」をするユーザカスタムブロックを「ユーザ定義初期化処理」専用としないで、(初期化処理が終わった後の) 通常のワンショット起動処理でも使用する場合には、UcaBlockOneshot の引数に指定されるパラメータ(シーケンステーブルのデータとして指定)により初期化処理と通常の処理を区別するようにしてください。

3.3 機能ブロック終了処理

機能ブロック終了処理は、ユーザカスタムブロックが実行を終了する直前に、自ブロック自身の後始末をするための処理です。システムは、ユーザカスタムブロックのブロックモードがO/Sに遷移するときや、オンラインの修正によりユーザカスタムブロックが削除されるときに、機能ブロック終了処理を呼び出します。

機能ブロック終了処理の基本的な記述を以下に示します。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockFinish
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME    処理続行不能
*
* description:            機能ブロック終了処理
*
* >>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockFinish(
    UcaBlockContext bc,          /* (IN/OUT): ブロックコンテキスト */
    I32 reason                 /* (IN): 呼び出し理由 */
)
{
    /* 機能ブロック終了処理を記述 */
    return SUCCEED;
}
```

1番目の引数bcに指定されるブロックコンテキストは、システムがユーザカスタムブロックの管理情報を保持する領域を指すポインタです。ユーザのプログラムが直接ブロックコンテキストを参照することはありません。ユーザはユーザカスタムブロック作成用ライブラリを呼び出すときに、引数にブロックコンテキスト（変数bc）をそのまま指定します。

2番目の引数reasonは、呼び出し理由です。つまり、ブロックモードがO/Sに遷移したとか、機能ブロックがオンラインで削除されたなど、機能ブロック終了処理が呼び出されたときの状況がシステムから渡されます。呼び出し理由の一覧を以下に示します。

表 機能ブロック終了処理の呼び出し理由

ラベル	呼び出し理由
UCAFINISH_DELETE	制御ドローイングビルダより、オンラインでユーザカスタムブロックを削除
UCAFINISH_PRG	機能ブロック詳細ビルダより、オンラインで「プログラム名称」を変更
	システムビューより、オンラインでユーザカスタムアルゴリズムを削除
UCAFINISH_MAINT	機能ブロック詳細ビルダより、オンラインで「プログラム名称」以外を変更
UCAFINISH_SLEEP	外部からのデータ設定によりブロックモード(MODE)がO/Sに遷移

補足

- ユーザカスタムブロックのブロックモードがO/Sのとき、ユーザカスタムブロックをオンラインで修正または削除しても、機能ブロック終了処理は呼び出されません。
- ユーザカスタムブロックのブロックモードがO/Sのとき、ユーザカスタムブロックのプログラム名称に指定されているユーザカスタムアルゴリズムをオンラインで修正しても、機能ブロック終了処理は呼び出されません。

機能ブロック終了処理は、ユーザカスタムブロックが終了するときに、ブロック自身のデータを変更するための処理です。機能ブロック終了処理では、自ブロックのデータにのみアクセス可能であり、他の機能ブロックや入出力端子へのアクセスはできません。つまり、自ブロックデータにアクセスするユーザカスタムアルゴリズム作成用ライブラリのみ使用可能であり、他の機能ブロックデータや入出力端子にアクセスするユーザカスタムアルゴリズム作成用ライブラリは使用できません。

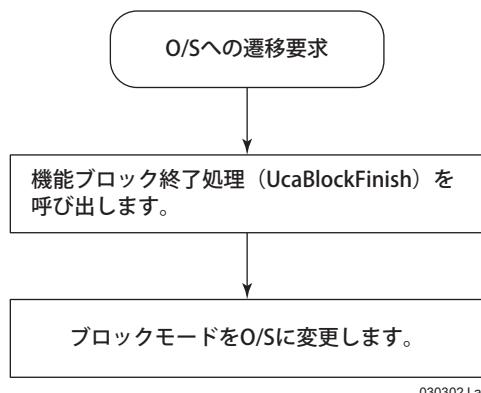
参照

機能ブロック初期化処理で使用可能なユーザカスタムアルゴリズム作成用ライブラリの詳細については、以下を参照してください。

[「3.8 ユーザカスタムアルゴリズム作成用ライブラリ一覧」](#)

■ ブロックモードがO/Sに遷移するときの動作

ブロックモードがO/Sに遷移するときの動作を説明します。ブロックモードO/Sへの遷移は、機能ブロックの実行を停止する特別なデータ変更です。このためシステムは、機能ブロックデータ設定時特殊処理を呼び出しません。ブロックモードがO/S以外からO/Sに遷移するときのシステムの処理を以下に示します。



030302J.ai

図 ブロックモードがO/S以外からO/Sに遷移するときのシステムの処理

3.3.1 オンラインの修正に伴う機能ロック初期化処理／終了処理

オンラインのユーザアプリケーション修正に伴うユーザカスタムブロックの動作について説明します。ユーザカスタムブロックに関するオンラインの修正は、次の4つに大別することができます。

- ・制御ドローイングビルダからのユーザカスタムブロックの追加と削除
- ・機能ロック詳細ビルダからのユーザカスタムブロックのビルダ定義項目の変更（「プログラム名称」以外）
- ・機能ロック詳細ビルダからのユーザカスタムブロックのビルダ定義項目「プログラム名称」の変更
- ・システムビューからのユーザカスタムアルゴリズムの追加・変更・削除

これらのオンラインの修正に伴い、ユーザカスタムブロック実行管理部（システム）は、ユーザカスタムアルゴリズムに記述されている機能ロック初期化処理、機能ロック終了処理、および機能ロックデータ設定時特殊処理を呼び出します。ここでは、それぞれのオンラインの修正に伴うシステムの動作を説明します。なお、この項の以下の説明で「機能ロックデータ設定時特殊処理」と記述してあるのは、「ロックモードをO/Sから復帰するためのロックモード変更指令に対する機能ロックデータ設定時特殊処理」を意味します。

■ 制御ドローイングビルダからユーザカスタムブロックを追加したときの動作

- ・ユーザは制御ドローイングビルダでユーザカスタムブロックを追加しオンラインでダウンロードします。
- ・システムはユーザカスタムブロックをAPCSの主記憶にダウンロードします。
- ・システムはユーザカスタムブロックに最初に処理タイミングが与えられたときに(*1)、CSTM-Cのときはロックモードを初期値のMAN(O/S)からMANにします。CSTM-Aのときはロックモードを初期値のAUT(O/S)からAUTにします。
- ・システムはプログラム名称に指定されたユーザカスタムアルゴリズムの機能ロック初期化処理を、呼び出し理由にUCAINIT_NEWを指定して呼び出します。

*1：ユーザカスタムブロックを追加してから、ユーザカスタムブロックに最初に処理タイミングが与えられるまでの間に外部からO/S以外のロックモードの設定された場合には、「オンラインでユーザカスタムブロックを追加」ではなく、「外部からのデータ設定によりロックモードがO/Sから復帰」として扱われます。

参照 システムの動作の詳細については、以下を参照してください。

「[3.2 機能ロック初期化処理](#)」

■ 制御ドローイングビルダからユーザカスタムブロックを削除したときの動作

- ・ ユーザは制御ドローイングビルダでユーザカスタムブロックの削除をオンラインで指示します。
- ・ システムはプログラム名称に指定されたユーザカスタムアルゴリズムの機能ブロック終了処理を、呼び出し理由に UCAFINISH_DELETE を指定して呼び出します。
- ・ システムはユーザカスタムブロックのブロックモードを O/S にします。
- ・ システムはユーザカスタムブロックを APCS の主記憶から削除します。

■ 機能ブロック詳細ビルダでユーザカスタムブロックの「プログラム名称」以外を変更

- ・ ユーザはビルダ定義項目の変更をオンラインでダウンロードします。
- ・ システムはプログラム名称に指定されたユーザカスタムアルゴリズムの機能ブロック終了処理を、呼び出し理由に UCAFINISH_MAINT を指定して呼び出します。
- ・ システムはユーザカスタムブロックのブロックモードを O/S にします。
- ・ システムはビルダ定義項目の変更を APCS 主記憶ユーザカスタムブロックの領域に反映します。
- ・ システムは機能ブロックデータ設定時特殊処理を変更前処理として呼び出します。
- ・ システムはユーザカスタムブロックのブロックモードを O/S から復帰します。
- ・ システムは機能ブロックデータ設定時特殊処理を変更後処理として呼び出します。
- ・ システムはプログラム名称に指定されたユーザカスタムアルゴリズムの機能ブロック初期化処理を、呼び出し理由に UCAINIT_MAINT を指定して呼び出します。

■ 機能ブロック詳細ビルダでユーザカスタムブロックの「プログラム名称」を変更

- ・ ユーザはビルダ定義項目の変更をオンラインでダウンロードします。
- ・ システムは変更前の「プログラム名称」に指定されたユーザカスタムアルゴリズムの機能ブロック終了処理を、呼び出し理由に UCAFINISH_PRG を指定して呼び出します。
- ・ システムはユーザカスタムブロックのブロックモードを O/S にします。
- ・ システムは新しい「プログラム名称」を APCS 主記憶のユーザカスタムブロックの領域に反映します。
- ・ システムは機能ブロックデータ設定時特殊処理を変更前処理として呼び出します。
- ・ システムはユーザカスタムブロックのブロックモードを O/S から復帰します。
- ・ システムは機能ブロックデータ設定時特殊処理を変更後処理として呼び出します。
- ・ システムは新しい「プログラム名称」に指定されたユーザカスタムアルゴリズムの機能ブロック初期化処理を、呼び出し理由に UCAINIT_PRG を指定して呼び出します。

次にシステムビューからのユーザカスタムアルゴリズムの追加・変更・削除について説明します。システムは処理対象のユーザカスタムアルゴリズムを使用しているすべてのユーザカスタムブロックに対して処理を行います。

■ システムビューからのユーザカスタムアルゴリズムの追加

- ・ ユーザは今まで登録されていないユーザカスタムアルゴリズムを登録し、オンラインのダウンロードを指示します。
- ・ システムはユーザカスタムアルゴリズムを APCS にダウンロードします。
- ・ システムはユーザカスタムアルゴリズムを APCS の主記憶にロードします。
- ・ システムはユーザカスタムアルゴリズムの「プログラム名称」が指定されているすべてのユーザカスタムブロックに対し、次の処理を行います。

```
while (<プログラム名称を指定したユーザカスタムブロックがある>) {  
    ユーザカスタムブロックのブロックモードを O/S にします。;  
    システムは、ユーザカスタムアルゴリズムに記述されている機能ブロックデータ設定時  
    特殊処理を変更前処理として呼び出します。;  
    ユーザカスタムブロックのブロックモードを O/S から復帰します。;  
    システムは、ユーザカスタムアルゴリズムに記述されている機能ブロックデータ設定時特  
    殊処理を変更後処理として呼び出します。;  
    システムは、ユーザカスタムアルゴリズムに記述されている機能ブロック初期化処理を、呼び出し理由に UCAINIT_PRG を指定して呼  
    び出します。;  
}
```

重要

ユーザカスタムブロックに指定された「プログラム名称」を持つユーザカスタムアルゴリズムが APCS にダウンロードされていない場合は、ユーザカスタムブロック実行管理部は「DLL 未ロード（異常レベル 2）」として扱います。この場合、システムはユーザカスタムブロックのブロックモードは O/S にしていません。したがって、システムビューからのユーザカスタムアルゴリズムの追加では、ブロックモードを一度 O/S にした上でユーザカスタムアルゴリズムとの関連付けをし、その後 O/S から復帰しています。

参照

DLL 未ロード（異常レベル 2）の詳細については、以下を参照してください。

APCS ユーザカスタムブロック (IM 33J15U20-01JA)

■ システムビューからのユーザカスタムアルゴリズムの変更

- ユーザは今まで登録されていたユーザカスタムアルゴリズムの更新、つまり修正されたユーザカスタムアルゴリズムのオンラインのダウンロードを指示します。
- システムはユーザカスタムアルゴリズムを APCS にダウンロードします。
- システムはダウンロードされた「プログラム名称」が指定されているすべてのユーザカスタムブロックに対し、次の処理を行います。

```
while (<プログラム名称を指定したユーザカスタムブロックがある>) {
    変更前のユーザカスタムアルゴリズムに記述されている機能ブロック終了処理を、呼び出し理由に UCAFINISH_PRG を指定して呼び出します。;
    ユーザカスタムブロックのブロックモードを O/S にします。;
}
```

- システムは古いユーザカスタムアルゴリズムを APCS の主記憶から削除します。
- システムは新しいユーザカスタムアルゴリズムを APCS の主記憶にロードします。
- システムはダウンロードされた「プログラム名称」が指定されているすべてのユーザカスタムブロックに対し、次の処理を行います。

```
while (<プログラム名称を指定したユーザカスタムブロックがある>) {
    システムは、新しいユーザカスタムアルゴリズムに記述されている機能ブロックデータ設定時特殊処理を変更前処理として呼び出します。;
    ユーザカスタムブロックのブロックモードを O/S から復帰します。;
    システムは、新しいユーザカスタムアルゴリズムに記述されている機能ブロックデータ設定時特殊処理を変更後処理として呼び出します。;
    新しいユーザカスタムアルゴリズムに記述されている機能ブロック初期化処理を、呼び出し理由に UCAINIT_PRG を指定して呼び出します。;
}
```

■ システムビューからのユーザカスタムアルゴリズムの削除

- ユーザはシステムビューから今まで登録されていたユーザカスタムアルゴリズムの削除を、オンラインで指示します。
- システムはユーザカスタムアルゴリズムの削除指令を APCS に通知します。
- システムは削除対象のユーザカスタムアルゴリズムの「プログラム名称」が指定されているすべてのユーザカスタムブロックに対し、次の処理を行います。

```
while (<プログラム名称を指定したユーザカスタムブロックがある>) {
    変更前のユーザカスタムアルゴリズムに記述されている機能ブロック終了処理を呼び出し理由に UCAFINISH_PRG を指定して呼び出します。;
    ユーザカスタムブロックのブロックモードを O/S にします。;
    ユーザカスタムブロックのブロックモードを O/S から復帰します。;
}
```

-
- システムはユーザカスタムアルゴリズムをAPCSの主記憶から削除します。

重要

ユーザカスタムブロックに指定された「プログラム名称」を持つユーザカスタムアルゴリズムがAPCSにダウンロードされていない場合は、ユーザカスタムブロック実行管理部は「DLL未ロード（異常レベル2）」として扱います。この場合、システムはユーザカスタムブロックのブロックモードはO/Sにしていません。したがって、「プログラム名称」に指定されているユーザカスタムアルゴリズムを削除するときには、一度O/Sにしたあと、O/Sから復帰しています。

参照

DLL未ロード（異常レベル2）の詳細については、以下を参照してください。

APCSユーザカスタムブロック (IM 33J15U20-01JA)

3.3.2 機能ブロック初期化処理／終了処理のプログラミング

機能ブロック初期化処理と機能ブロック終了処理はさまざまな機会に呼び出されますが、呼び出し理由ごとに処理を分けようとすればプログラムが難しくなります。

機能ブロック初期化処理と機能ブロック終了処理のプログラミングについて整理します。

- ・ ユーザは基本的には「機能ブロック初期化処理で自ブロックに必要な初期化をする」ようにプログラミングしてください。
- ・ ユーザは機能ブロック初期化処理と機能ブロック終了処理がどの呼び出し理由でも同じ処理をできるように、ユーザカスタムアルゴリズムを設計してください。
- ・ 同じ処理ができない場合に限り、引数に渡される呼び出し理由を使い処理を分けてください。
- ・ 機能ブロック初期化処理と機能ブロック終了処理では簡単な処理のみを行ってください。
- ・ 機能ブロック初期化処理や機能ブロック終了処理で難しい処理をしないでください。解決の困難なユーザアプリケーションミスのもとになります。

機能ブロック初期化処理と機能ブロック終了処理は、ユーザカスタムブロックが実行を開始・終了するときにブロック自身のデータを変更するための処理です。これらの処理では自ブロックのデータにのみアクセス可能であり、他の機能ブロックや入出力端子へのアクセスはできません。つまり、自ブロックデータにアクセスするユーザカスタムアルゴリズム作成用ライブラリのみ使用可能であり、入出力端子や他の機能ブロックデータにアクセスするユーザカスタムアルゴリズム作成用ライブラリは使用できません。

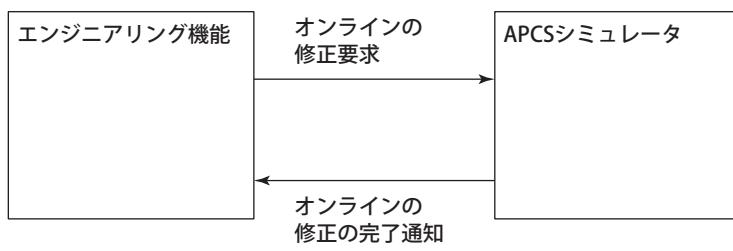
参照 機能ブロック初期化処理と機能ブロック終了処理で使用可能なユーザカスタムアルゴリズム作成用ライブラリの詳細については、以下を参照してください。

「3.8 ユーザカスタムアルゴリズム作成用ライブラリ一覧」

3.3.3 機能ブロック初期化処理／終了処理のデバッグ

バーチャルテスト機能で稼動しているユーザカスタムブロックの機能ブロック初期化処理と機能ブロック終了処理をVisual C++でデバッグする場合は、ブロックモードをO/Sに遷移（機能ブロック終了処理が呼び出されます）するか、ブロックモードをO/Sから復帰（機能ブロック初期化処理が呼び出されます）することによりデバッグします。オンラインの修正に伴う機能ブロック初期化処理／終了処理では、VC++のブレークポイントを使用したデバッグはしないでください。

オンラインの修正は、エンジニアリング機能からAPCSシミュレータにダウンロードされます。エンジニアリング機能は、オンライン修正の要求を出力し、APCSシミュレータからのオンライン修正完了の通知を待ちます。



030303J.ai

図 オンラインの修正

オンラインの修正に伴う機能ブロック初期化処理や機能ブロック終了処理の中にVisual C++でブレークポイントを設定すると、APCSシミュレータの実行はそこで停止てしまいます。その結果、APCSシミュレータは、エンジニアリング機能に「オンラインの修正の完了通知」を返せなくなります。エンジニアリング機能は、完了通知が戻ってこないでタイムアウトすると、「オンラインの修正は失敗した」と判断します。

機能ブロック初期化処理／機能ブロック終了処理の呼び出し理由によりユーザプログラムの処理が異なる場合には、Visual C++の機能で呼び出し理由の引数reasonの値を変更してデバッグしてください。呼び出し理由の値を以下に示します。

表 機能ブロック初期化処理の呼び出し理由の値

ラベル	呼び出し理由
UCAINIT_START	1
UCAINIT_NEW	2
UCAINIT_PRG	3
UCAINIT_MAINT	4
UCAINIT_AWAKEN	5

表 機能ブロック終了処理の呼び出し理由の値

ラベル	呼び出し理由
UCAFINISH_DELETE	2
UCAFINISH_PRG	3
UCAFINISH_MAINT	4
UCAFINISH_SLEEP	5

機能ブロック初期化処理／機能ブロック終了処理のデバッグについて整理します。

- ・機能ブロック初期化処理は、ユーザカスタムブロックのブロックモードをO/Sから復帰（HISなどからO/SからO/S以外に変更）することでデバッグしてください。
- ・機能ブロック終了処理は、ユーザカスタムブロックのブロックモードをO/Sに遷移（HISなどからO/S以外からO/Sに変更）することでデバッグしてください。
- ・機能ブロック初期化処理／機能ブロック終了処理のデバッグが終わったら、ブレークポイントの設定を削除してください。ブレークポイントの設定を残しておくと、以後のオンライン修正時にブレークポイントで停止してしまいます。
- ・機能ブロック初期化処理／機能ブロック終了処理をオンラインの修正に伴いVC++のブレークポイントを使用してデバッグしないでください。

補足

機能ブロック初期化処理／機能ブロック終了処理中のブレークポイントが原因でオンラインの修正に失敗したときの復旧手段について説明します。

- ・エンジニアリング機能が持つAPCSのユーザアプリケーションには問題は発生しません。オンラインの修正が失敗するだけで、データベースには影響はありません（修正は反映されません）。
- ・APCSシミュレータが持つAPCSのユーザアプリケーション（メモリイメージファイル）は使用できなくなります。復旧するにはテスト機能を終了するときに「削除」を指定して、APCSシミュレータが持つデータベース（ユーザアプリケーション）を削除してください。

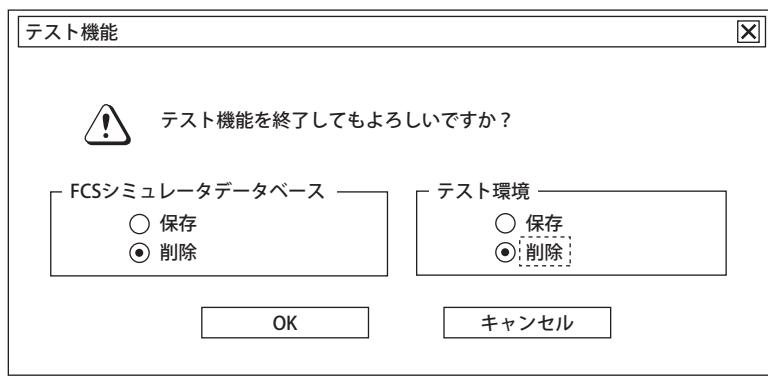


図 削除指定

このとき、APCSシミュレータが持っているチューニングパラメータ（変更してからチューニングパラメータセーブしていないパラメータ）は削除されます。

- ・再度テスト機能を起動すると、エンジニアリング機能が持つAPCSのユーザアプリケーションがAPCSシミュレータにダウンロードされます。チューニングパラメータは、最後にチューニングパラメータセーブしたときのデータとなります。

3.4 機能ブロック定周期処理

機能ブロック定周期処理は、ユーザカスタムブロックのビルダ定義項目「実効スキャン周期」に指定した周期ごとに繰り返し呼び出される定周期処理です。ユーザカスタムブロック実行管理部（システム）は、「実効スキャン周期」と「スキャン位相」によりユーザカスタムブロックの処理タイミングを決定し、機能ブロック定周期処理を呼び出します。

機能ブロック定周期処理の基本的な記述を以下に示します。

```
/*
* <<FNH>>*****
*
* Function name:      UcaBlockPeriodical
* Return value:       SUCCEED      正常終了
*                      UCAERR_NOPROC   処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description:        機能ブロック定周期処理
*
* >>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT):ブロックコンテキスト */
)
{
    /* 機能ブロック定周期処理を記述 */
    return SUCCEED;
}
```

引数 bc に指定されるブロックコンテキストは、システムがユーザカスタムブロックの管理情報を保持する領域を指すポインタです。ユーザのプログラムが直接ブロックコンテキストを参照することはありません。ユーザはユーザカスタムブロック作成用ライブラリを呼び出すときに、引数にブロックコンテキスト（変数 bc）をそのまま指定します。機能ブロック定周期処理が呼び出されるタイミングについて説明します。ユーザカスタムブロックの処理タイミングは、PID 調節ブロック（PID）や汎用演算ブロック（CALCU）などの標準ブロックと同じです。システムは同一スキャンに機能ブロックを、制御ドローリングの番号順に実行します。また、同一制御ドローリング内では、ブロック実行順序の設定（定義した機能ブロックの番号順）により機能ブロックの実行順を決定します。ユーザカスタムブロックの実行順もこの規則に従います。

参照 機能ブロックの実行順の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「7.1.2 処理の実行順序」](#)

ユーザカスタムブロックに処理タイミングが与えられると、ユーザカスタムブロック実行管理部（システム）は実効スキャン周期とスキャン位相を計算し、今回が実効タイミングであればユーザ定義関数 UcaBlockPeriodical を呼び出します。UcaBlockPeriodical が呼び出されると、ユーザプログラムがリターン（処理があるなら return SUCCEED、ないなら return UCAERR_NOPROC）するまでの C プログラムが実行されます。つまり、ユーザ定義関数 UcaBlockPeriodical は 1 回分の処理を途中で中断されることなく連続して実行します。

参照 ユーザカスタムブロックの「実効スキャン周期」と「スキャン位相」の詳細については、以下を参照してください。

[APCS ユーザカスタムブロック \(IM 33J15U20-01JA\)](#)

補足 ユーザカスタムアルゴリズムのユーザプログラムが連続して 1 秒を越えて実行し続けると、システムはユーザカスタムブロックの実行を打ち切り、ユーザカスタムブロックのブロックモードを O/S にします。システムが処理を打ち切るのは、機能ブロック定周期処理だけでなく、機能ブロックワンショット起動処理など 5 つのユーザ定義の処理すべてで同じです。

参照 処理の打ち切りの詳細については、以下を参照してください。

[APCS ユーザカスタムブロック \(IM 33J15U20-01JA\)](#)

3.5 機能ブロックワンショット起動処理

機能ブロックワンショット起動処理は、他の機能ブロックからユーザカスタムブロックがワンショット起動されたときに実行されます。機能ブロックワンショット起動処理は、条件判定または状態操作として呼び出されます。

条件判定として呼び出された場合の基本的な記述を以下に示します。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockOneshot
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC 处理なし
*                         UCAERR_STOPME 处理続行不能
*
* description:            機能ブロックワンショット起動処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc,           /* (IN/OUT): ブロックコンテキスト */
    I32 code,                    /* (IN): 種別コード */
    I32 parameter,              /* (IN): パラメータ */
    BOOL *result                /* (OUT): 実行結果 */
)
{
    .....
    /* 条件判定でなければ、何もしない */
    if (code != UCAONESHOT_CODECOND) {
        return UCAERR_NOPROC;
    }

    /* 条件判定 */
    if (<条件が真>) {
        *result = TRUE; /* 真 */
    } else {
        *result = FALSE; /* 偽 */
    }

    return SUCCEED;
}
```

状態操作の処理をする場合の基本的な記述を示します。

```
/*
* <<FNH>>*****
*
* Function name:      UcaBlockOneshot
* Return value:       SUCCEED      正常終了
*                      UCAERR_NOPROC   処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description:        機能ブロックワンショット起動処理
*
* >>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc,           /* (IN/OUT): ブロックコンテキスト */
    I32 code,                    /* (IN): 種別コード */
    I32 parameter,               /* (IN): パラメータ */
    BOOL *result                 /* (OUT): 実行結果 */
)
{
.....
/* 状態操作でなければ何もしない */
if (code != UCAONESHOT_CODEOPRT) {
    return UCAERR_NOPROC;      /* 処理なし */
}

/* 状態操作の処理を記述 */
.....
*result = TRUE;                /* いつも真 */
return SUCCEED;
}
```

1つのユーザカスタムアルゴリズムで条件判定と状態操作の両方を処理する場合は、次のようにになります。

```

UCAUSER_API I32 UCAAPI UcaBlockOneshot (
    UcaBlockContext bc,           /* (IN/OUT): ブロックコンテキスト */
    I32 code,                   /* (IN): 種別コード */
    I32 parameter,             /* (IN): パラメータ */
    BOOL *result                /* (OUT): 実行結果 */
)
{
    .....
    if (code == UCAONESHOT_CODECOND) {
        /* 条件判定 */
        .....
        if (<条件が真>) {
            *result = TRUE;          /* 真 */
        } else {
            *result = FALSE;         /* 偽 */
        }
    } else if (code == UCAONESHOT_CODEOPRT) {
        /* 状態操作 */
        .....
        *result = TRUE;          /* いつも真 */
    }
    return SUCCEED;
}

```

機能ブロックワンショット起動処理 UcaBlockOneshot の引数について説明します。1番目の引数 bc に指定されるブロックコンテキストは、システムがユーザカスタムブロックの管理情報を保持する領域を指すポインタです。ユーザのプログラムが直接ブロックコンテキストを参照することはありません。ユーザはユーザカスタムブロック作成用ライブラリを呼び出すときに、引数にブロックコンテキスト（変数 bc）をそのまま指定します。2番目の引数 code は、ワンショット起動の種別が渡されます。

表 機能ブロックワンショット起動処理の種別

ラベル	種別
UCAONESHOT_CODECOND	条件判定
UCAONESHOT_CODEOPRT	状態操作

3番目の引数 parameter はパラメータです。パラメータは、ユーザカスタムブロックを起動する機能ブロックが、ユーザカスタムブロックに渡す引数として指定します。シーケンステーブルブロックからユーザカスタムブロックをワンショット起動する場合には、データ欄にパラメータを指定します。以下ではユーザカスタムブロック CMA01_OPRT をワンショット起動するパラメータとして、10、20 または 30 が指定されています。



図 ユーザカスタムブロックのワンショット起動

パラメータを指定できるのは、状態操作としてユーザカスタムブロックをワンショット起動する場合だけです。条件判定でワンショット起動する場合には指定できません。

表 パラメータの指定

	指定の可否	シーケンステーブルの「データ」の記述
条件判定	不可	常に「ON」と記述します。
状態操作	0～32767 の整数を指定できます。	0～32767 の整数を指定します。

最後の引数 result は、条件判定の結果を格納する実行結果を指すポインタです。

表 ユーザプログラムにおける実行結果resultの設定

	判定結果が真	判定結果が偽
条件判定	*result = TRUE;	*result = FALSE
状態操作	常に *result = TRUE; と記述します。	

実行結果に設定した真 (TRUE)、偽 (FALSE) が有効になるのは、UcaBlockOneshot が SUCCEED でリターンした場合のみです。UCAERR_NOPROC でリターンすると実行結果に設定した真偽は無視されます。

参照 ユーザ定義関数の戻り値の詳細については、以下を参照してください。

[「3.7 ユーザ定義関数の戻り値」](#)

3.5.1 ビルダ定義項目「起動タイミング」

ユーザカスタムブロックのビルダ定義項目「起動タイミング」とユーザカスタムアルゴリズムの関係について説明します。ビルダ定義項目「起動タイミング」を以下に示します。

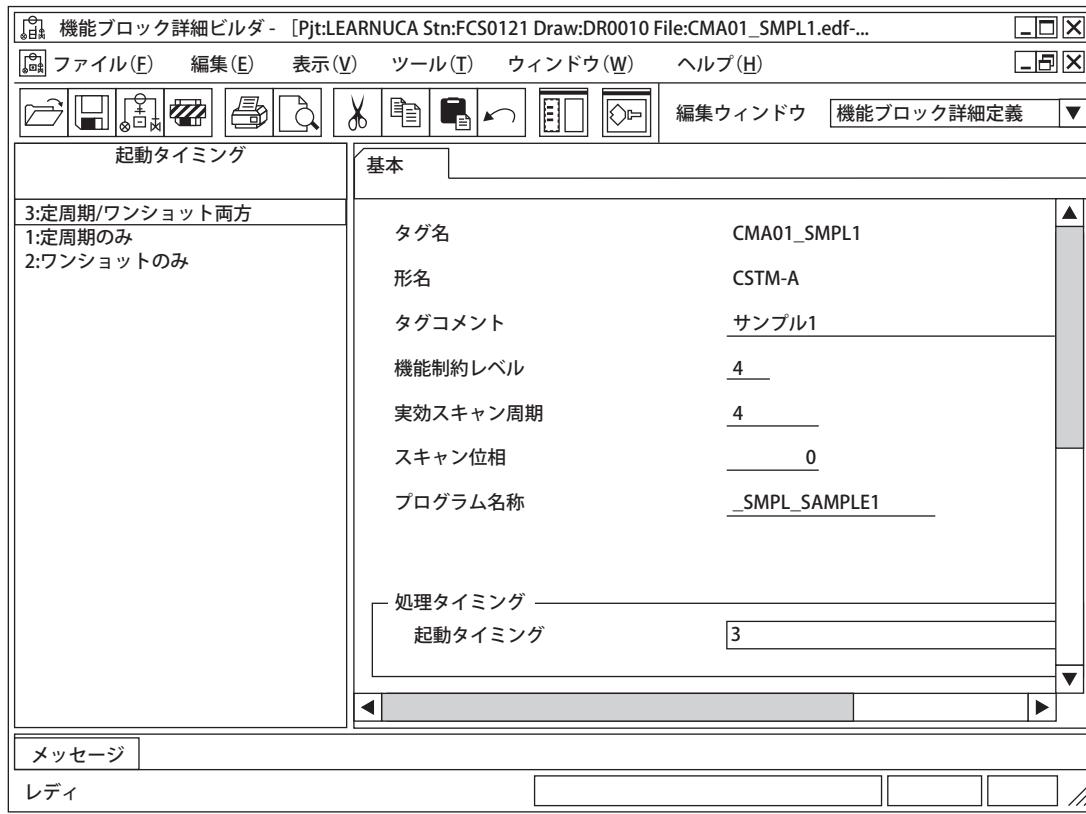


図 起動タイミング

ユーザカスタムアルゴリズムの機能ブロック定周期処理は UcaBlockPeriodical です。また、機能ブロックワンショット起動処理は UcaBlockOneshot です。たとえば、機能ブロック定周期処理のあり／なしにより、ユーザカスタムアルゴリズムのプログラムは次のように記述します。

● 機能ブロック定周期処理をする場合の記述

```
UcaBlockPeriodical(
    <引数並び>
)
{
    /* 機能ブロック定周期処理 */
    .....
    return SUCCEEDED;
}
```

● 機能ブロック定周期処理をしない場合の記述

```

UcaBlockPeriodical(
    <引数並び>
)
{
    return UCAERR_NOPREC; /* 処理なし */
}

```

ビルダ定義項目「起動タイミング」とユーザカスタムアルゴリズムの記述の関係を以下に示します。

表 機能ブロック定周期処理の実行

ビルダ定義項目 「起動タイミング」	ユーザカスタムアルゴリズムの記述		
	UcaBlockPeriodical処理あり UcaBlockoneshot処理あり	UcaBlockPeriodical処理あり UcaBlockoneshot処理なし	UcaBlockPeriodical処理なし UcaBlockoneshot処理あり
定周期、ワンショット両方	○	○	×
定周期のみ	○	○	×
ワンショットのみ	×	×	×

○：実行する

×：実行しない

表 機能ブロックワンショット起動処理の実行

ビルダ定義項目 「起動タイミング」	ユーザカスタムアルゴリズムの記述		
	UcaBlockPeriodical処理あり UcaBlockoneshot処理あり	UcaBlockPeriodical処理あり UcaBlockoneshot処理なし	UcaBlockPeriodical処理なし UcaBlockoneshot処理あり
定周期、ワンショット両方	○	×	○
定周期のみ	×	×	×
ワンショットのみ	○	×	○

○：実行する

×：実行しない

3.6 機能ブロックデータ設定時特殊処理

機能ブロックデータ設定時特殊処理は、自ブロックのデータアイテムに外部からデータを設定されるときに実行されます。「外部」とはHISからのオペレータ入力や、他の機能ブロックからのデータ設定を意味します。ユーザカスタムブロック実行管理部は、ユーザカスタムブロックのデータアイテムにデータを設定する前後に、機能ブロックデータ設定時特殊処理を呼び出します。

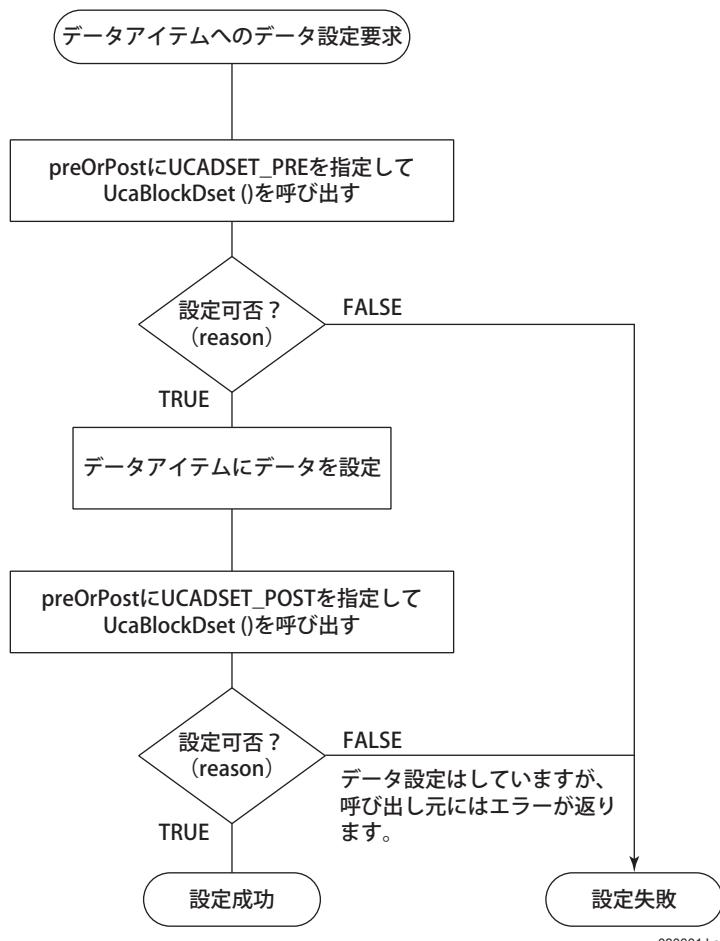


図 システムの機能データ設定時特殊処理の呼び出し

データ設定時に、機能ブロックデータ設定時特殊処理が呼び出されるデータアイテムと、呼び出されないデータアイテムがあります。

参照 ユーザカスタムブロックのデータ設定時特殊処理の対象データアイテムの詳細については、以下を参照してください。

[APCS ユーザカスタムブロック \(IM 33J15U20-01JA\)](#)

ロックモードが O/S 以外から O/S になるときは、機能ロックデータ設定時特殊処理ではなく機能ロック終了処理が呼び出されます。

参照 機能ロック終了処理の詳細については、以下を参照してください。
[「3.3 機能ロック終了処理」](#)

ロックモードが O/S から O/S 以外になるときには、機能ロックデータ設定時特殊処理と機能ロック初期化処理が呼び出されます。

参照 機能ロック初期化処理の詳細については、以下を参照してください。
[「3.2 機能ロック初期化処理」](#)

機能ロックデータ設定時特殊処理には、自ブロックのデータアイテムへのデータ設定時にデータ設定の可否を判定する処理（データ設定前処理）、またはデータ設定に伴う処理（データ設定後処理）を記述します（データ設定前処理と後処理を両方記述することもできます）。機能ロックデータ設定時特殊処理では自ブロックのデータにのみアクセス可能であり、他の機能ロックや入出力端子へのアクセスはできません。つまり、自ブロックデータにアクセスするユーザカスタムアルゴリズム作成用ライブラリのみ使用であり、他の機能ロックデータや入出力端子にアクセスするユーザカスタムアルゴリズム作成用ライブラリは使用できません。

参照 機能ロックデータ設定時特殊処理で使用可能なユーザカスタムアルゴリズム作成用ライブラリの詳細については、以下を参照してください。
[「3.8 ユーザカスタムアルゴリズム作成用ライブラリ一覧」](#)

機能ロックデータ設定時特殊処理は、以下の 3 種類に大別することができます。

表 機能ロックデータ設定時特殊処理の分類

分類	説明
数値形データアイテムに対する処理	データアイテム P01 のような数値形データアイテムに対し、外部からデータ設定されたときの処理を記述します。
文字列型データアイテムに対する処理	データアイテム S01 のような文字列型データアイテムに対し、外部からデータ設定されたときの処理を記述します。
ロックモード変更指令に対する処理	外部からロックモードを変更されたときの処理を記述します。これはデータアイテム MODE に対するロックモード変更指令に対する処理となります。

- 参照**
- 数値形データアイテムに対する処理および文字列型データアイテムに対する処理の詳細については、以下を参照してください。
[「1.5 機能ロックデータ設定時特殊処理」](#)
 - ロックモード変更指令に対する処理の詳細については、以下を参照してください。
[「3.6.1 CSTM-C のロックモードを AUT と O/S に限定」](#)

機能ブロックデータ設定時特殊処理の基本的な記述を以下に示します（以下の例は、数値形データアイテム P01 に対するデータ設定時特殊処理です）。

```
/*
* <<FNH>>*****
*
* Function name:      UcaBlockDset
* Return value:       SUCCEED      正常終了
*                      UCAERR_NOPROC   処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description: 機能ブロックデータ設定時特殊処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockDset(
    UcaBlockContext bc,           /* (IN/OUT): ブロックコンテキスト */
    DITMN itemName,              /* (IN): データアイテム名 */
    UcaUnivType *data,           /* (IN): 設定データ */
    UcaDataOrStatus dataOrStatus, /* (IN): 設定種別 */
    UcaPreOrPost preOrPost,      /* (IN): 呼び出し種別 */
    BOOL *result                 /* (OUT): 設定可否 */
)
{
    /* データ設定前処理でないなら何もしない */
    if (preOrPost != UCADSET_PRE) {
        return UCAERR_NOPROC;
    }

    /* データアイテム P01 ならデータ可否を判定する */
    if (strcmp(itemName, "P01", sizeof(DITMN)) == 0) {
        .....
        /* データの設定可否を判定 */
        if (<設定可能か判定>) {
            *result = UCADSET_ALLOW;    /* 設定可 */
        } else {
            *result = UCADSET_DENY;   /* 設定不可 */
        }
        return SUCCEED;
    }

    /* P01 のデータアイテムは、処理なし */
    return UCAERR_NOPROC;
}
```

機能ブロックデータ設定時特殊処理 UcaBlockDset の引数について説明します。1番目の引数 bc に指定されるブロックコンテキストは、システムがユーザカスタムブロックの管理情報を保持する領域を指すポインタです。ユーザのプログラムが直接ブロックコンテキストを参照することはありません。ユーザはユーザカスタムブロック作成用ライブラリを呼び出すときに、引数にブロックコンテキスト（変数 bc）をそのまま指定します。

2番目の itemName にはデータ設定の対象となるデータアイテム名が渡されます。英字は大文字です。

3番目の引数 data はこれから設定しようとしているデータのデータとデータ型です。データ設定前処理（5番目の引数 preOrPost が UCADSET_PRE）でユーザプログラムが引数 data のデータを書きかえると、システムは書きかえられたデータをデータアイテムに設定します。データ設定後処理（5番目の引数 preOrPost が UCADSET_POST）でユーザプログラムが引数 data のデータを書きかえても、データアイテムの値は変化しません。

4番目の引数 dataOrStatus はデータ設定の種別です。

表 機能ブロックデータ設定時特殊処理の設定種別

ラベル	種別
UCADSET_DATA	通常のデータ設定
UCADSET_STATUS	ブロックモード（MODE）に対するブロックモード変更指令

5番目の引数 preOrPost はデータ設定前処理かデータ設定後処理かを示します。

表 機能ブロックデータ設定時特殊処理の呼び出し種別

ラベル	種別
UCADSET_PRE	データ設定前処理
UCADSET_POST	データ設定後処理

6番目の引数 result が指す設定可否には、ユーザプログラムがデータ設定の可否を設定します。

表 機能ブロックデータ設定時特殊処理の設定可否

意味	記述
設定可	*result = UCADSET_ALLOW
設定不可	*result = UCADSET_DEN

なお、設定可否が有効になるのは UcaBlockDset が SUCCEED でリターンした場合です。UCAERR_NOPROC でリターンすると、設定可否への設定値には関係なく、ユーザカスタムブロック実行管理部は常に設定可能と判断します。

3.6.1 CSTM-CのブロックモードをAUTとO/Sに限定

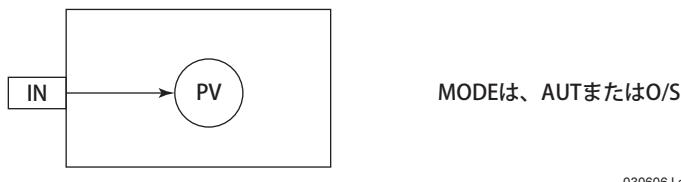
機能ブロックデータ設定時特殊処理の4番目の引数dataOrStatusには、UCADSET_DATA（通常のデータ設定）か、またはUCADSET_STATUS（データアイテムMODEに対するブロックモード変更指令）が渡されます。ここでは、ブロックモード変更指令に対する機能ブロックデータ設定時特殊処理の書き方を説明します。

参照 通常のデータ設定の詳細については、以下を参照してください。

「1.5 機能ブロックデータ設定時特殊処理」

これから説明する連続制御形ユーザカスタムブロックは、ブロックモードとしてAUTとO/Sのみを持つことができるようになります(*1)。機能ブロック初期化処理でブロックモードをAUTに初期化します。また、データ設定時特殊処理でブロックモードはAUTとO/Sのみに変更可能になるよう制限します。

機能ブロック定周期処理の動作としては、標準の指示ブロック(PVI)のように、IN端子からの入力信号を測定値(PV)として表示する連続制御形ユーザカスタムブロックを作ります。このユーザカスタムブロックは、入力上限／下限アラーム(HI, LO)と入力上上限／下下限アラーム(HH, LL)を検出します(*2)。ブロック図を示します。



030606J.ai

図 連続制御形ユーザカスタムブロック

手動操作ブロック(MLD)とこのユーザカスタムブロックと結合し、MLDが出力するMV値をユーザカスタムブロックのIN端子から入力してみます。

*1：何も制限しない場合、連続制御形ユーザカスタムアルゴリズム(CSTM-C)は、AUT, CAS, RCASなどPIDブロックと同じブロックモードを持ちます。

参照 ユーザカスタムブロックのブロックモードの詳細については、以下を参照してください。

APCS ユーザカスタムブロック (IM 33J15U20-01JA)

*2：アラームを検出するためには、ユーザ定義ステータス文字列ビルダでアラームステータスの初期値設定が必要です。

参照 初期値を設定する手順の詳細については、以下を参照してください。

「4.3.2 アラームステータス」

サンプルプログラムを動かしてみます。サンプルソリューション _SMPL_PVI_NOOUT の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは1章の準備作業により以下の作業フォルダにコピーされています。

<フォルダ名> : ¥UcaWork¥UcaSamples¥_SMPL_PVI_NOOUT

Visual Studio を起動し、_SMPL_PVI_NOOUT¥_SMPL_PVI_NOOUT.sln を開きます。[ビルド] メニューの [リビルド] で _SMPL_PVI_NOOUT の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。

次に、サンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥drawings¥PVI_NOOUT.txt

FCS0121(APCS) の制御ドローイングの DR0030（空いている制御ドローイングならどれでも構いません）を指定して制御ドローイングビルダを起動します。[ファイル] メニューの [外部ファイル] – [インポート] を指定します。インポートダイアログで上記の PVI_NOOUT.txt を指定し [開く] ボタンをクリックすると、以下の制御ドローイングが取り込まれます。[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

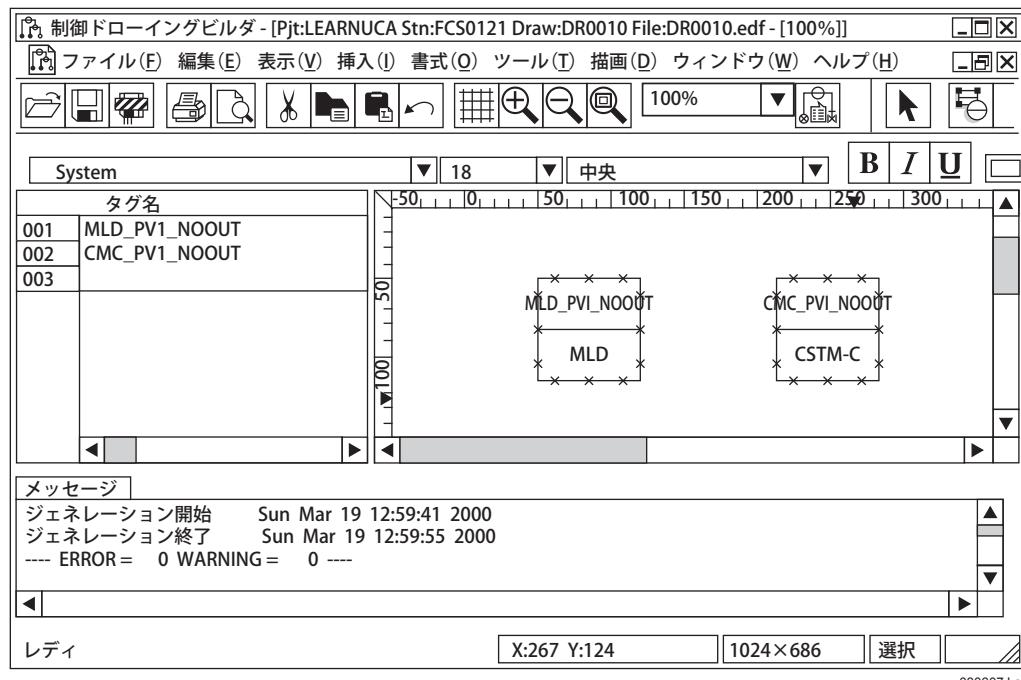


図 制御ドローイングのインポート

手動操作ブロック MLD_PVI_NOOUT の MV への出力を、連続制御用ユーザカスタムブロック CMC_PVI_NOOUT の IN 端子から入力しています。次に FCS0121 (APCS) をバーチャルテスト機能で起動してください。起動が完了したら、MLD_PVI_NOOUT の計器図と CMC_PVI_NOOUT のチューニングウィンドウを表示してください。そして、次表に従いデータを手入力してください。

表 MLD_PVI_NOOUTとCMC_PVI_NOOUTに手入力で設定するデータ

タグ名	データアイテム	データ
MLD_PVI_NOOUT	MV	50
	HH	70
	PH	60
	PL	40
	LL	30

以下は、CMC_PVI_NOOUT のチューニングウィンドウです。ブロックモードは AUT になっています。

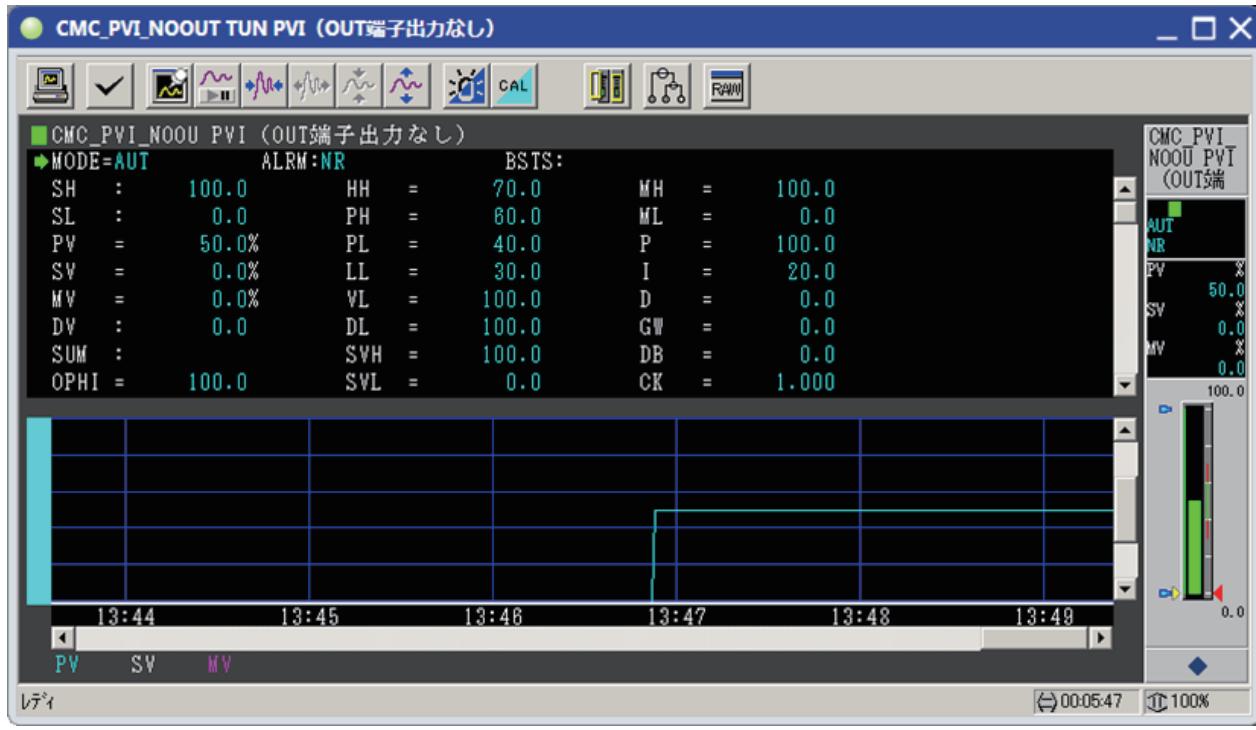
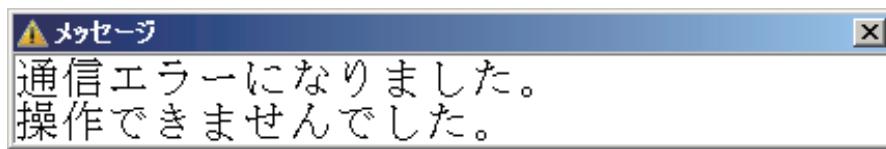


図 CMC_PVI_NOOUTのチューニングウィンドウ

連続制御形ユーザカスタムブロック CMC_PVI_NOOUT の動作について説明します。ブロックモードに MAN を設定してください。以下のエラーになります。



030610J.ai

図 エラーメッセージ

次に、ブロックモードを O/S にしてください。O/S になったことを確認してから、ブロックモードを AUT に戻してください。これでブロックモードが AUT と O/S のみに遷移可能になっていることを確認できました。

■ ブロックモードをAUTに初期化する

ユーザカスタムアルゴリズムのCプログラムについて説明します。ソリューション _SMPL_PVI_NOOUT の pvi_noout.c から、機能ブロック初期化処理 (UcaBlockInit で検索) を見つけます。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockInit
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC 处理なし
*                         UCAERR_STOPME 处理続行不能
*
* description:            機能ブロック初期化処理
*
* >>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockInit(
    UcaBlockContext bc,          /* (IN/OUT): ブロックコンテキスト */
    I32 reason                 /* (IN): 呼び出し理由 */
)
{
    I32 rtnCode;                /* リターンコード */

    /* ブロックモードを AUT に初期化 */
    rtnCode = UcaModeSet(bc, UCAMASK_MODE_AUT);

    return SUCCEED;
}
```

ユーザカスタムブロックが実行を開始するときに呼び出される機能ブロック初期化処理で、ユーザカスタムアルゴリズム作成用ライブラリ UcaModeSet によりブロックモードにAUTを設定しています。なお、ブロックモードがO/Sの状態でAPCSの制御機能を停止・起動した場合には、ブロックモードはO/Sのままになります。したがって、(APCS停止時のブロックモードがO/Sだったユーザカスタムブロックに対しては) 機能ブロック初期化処理は呼び出されません。

■ ブロックモードをAUTとO/Sに限定する

次に、機能ブロックデータ設定時特殊処理(UcaBlockDsetで検索)について説明します。

```

/*
* <<FNH>>*****
*
* Function name:          UcaBlockDset
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC 处理なし
*                         UCAERR_STOPME 处理続行不能
*
* description: 機能ブロックデータ設定時特殊処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockDset(
    UcaBlockContext bc,                      /* (IN/OUT): ブロックコンテキスト */
    DITMN itemName,                          /* (IN): データアイテム名 */
    UcaUnivType *data,                      /* (IN): 設定データ */
    UcaDataOrStatus dataOrStatus,           /* (IN): 設定種別 */
    UcaPreOrPost preOrPost,                 /* (IN): 呼び出し種別 */
    BOOL *result                           /* (OUT): 設定可否 */
)
{
    /* データ設定前処理でなければ何もしません */
    if (preOrPost != UCADSET_PRE) {
        return UCAERR_NOPROC; /* 处理なし */
    }

    /* ブロックモードに対するモード変更指令なら設定可否を判定します */
    if ((strncmp(itemName, "MODE", sizeof(DITMN))) == 0
        && (dataOrStatus == UCADSET_STATUS)) {

        /* AUTとO/Sは設定可能 */
        if ((strncmp(data->dataValue.string, "AUT", sizeof(data->dataValue.string)) ==
            0)
            || (strncmp(data->dataValue.string, "O/S", sizeof(data->dataValue.string)) ==
            0)) {
            *result = UCADSET_ALLOW; /* データ設定可能 */
        } else {
            *result = UCADSET_DENY; /* データ設定不可 */
        }
    }

    return SUCCEED;
}

```

このプログラムは、データ設定前処理の場合のみブロックモード変更指令を検査し、AUT または O/S への遷移を許可します。MODE に対するブロックモード変更指令か否かを次の部分で判定しています。

```
/* ブロックモードに対するモード変更指令なら設定可否を判定します */
if ((strncmp(itemName, "MODE", sizeof(DITMN))) == 0
    && (dataOrStatus == UCADSET_STATUS)) {
```

引数 itemName に "MODE" が設定されていて、かつ引数 dataOrStatus が UCADSET_STATUS なら、ブロックモード変更指令です。ブロックモード変更指令を受信すると、引数 data の値が AUT または O/S であれば result が差す設定可否に UCADSET_ALLOW を設定し、AUT または O/S 以外なら UCADSET_DENY を設定しています。なお、ブロックモード変更指令(引数 dataOrStatus が UCADSET_STATUS)の場合には、data の共用体メンバ dataValue は必ず文字列型ですので、このプログラムではデータ型(構造体メンバ dType)の検査は省略しています。

補足

このプログラムは、文字列の比較に Windows のライブラリ `strcmp` を使用しています。また、このプログラムでは、`strcmp` を使用するために Windows が提供するインクルードファイル `string.h` を `#include` 行で取り込んでいます。

```
/*
 * Windows インクルードファイル
 *
 * ユーザカスタムアルゴリズムで使用できる Windows のインクルードファイルは
 * 以下の 9 個だけです。また、インクルードファイルで宣言されている関数を
 * すべて使用できるわけではありません。
 * 使用可能な関数は、ユーザカスタムアルゴリズムプログラミング説明書に
 * 記載されています。
 */
/* #include <stdlib.h> */
#include <string.h> /* Windows の文字列操作ライブラリ */
/* #include <math.h> */
/* #include <ctype.h> */
/* #include <stddef.h> */
/* #include <time.h> */
/* #include <float.h> */
/* #include <limits.h> */
/* #include <stdio.h> */
....
```

参照

Windows のライブラリの詳細については、以下を参照してください。

[「4.8 Windows のライブラリ」](#)

3.6.2 CSTM-Cでの入力上限／下限アラームの検出

手動操作ブロックMLD_PVI_NOOUTがMVに出力するデータを連続制御形ユーザカスタムブロックCMA_PVI_NOOUTが入力信号として取り込み、測定値PVに表示する処理について説明します。

CMA_PVI_NOOUTのチューニングウィンドウを表示して、PVが50になっていることを確認してください。これはMLD_PVI_NOOUTのMV値を入力信号として取り込んだ結果です。この状態で計器図から以下のダイアログを呼び出して、MLD_PVI_NOOUTのMVを65まで少しづつ増加してください。

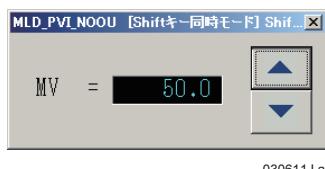


図 MLD_PVI_NOOUTのMVの操作

MLD_PVI_NOOUTのMVが増加するに従い、CMA_PVI_NOOUTのPVが追随することがわかります。CMA_PVI_NOOUTは実効スキャン周期（デフォルトの4秒に指定してあります）ごとに、IN端子からMVを入力しPVに反映します。CMA_PVI_NOOUTのPVがPH(60)を越えると、CMA_PVI_NOOUTはHI(入力上限アラーム)を発生します。

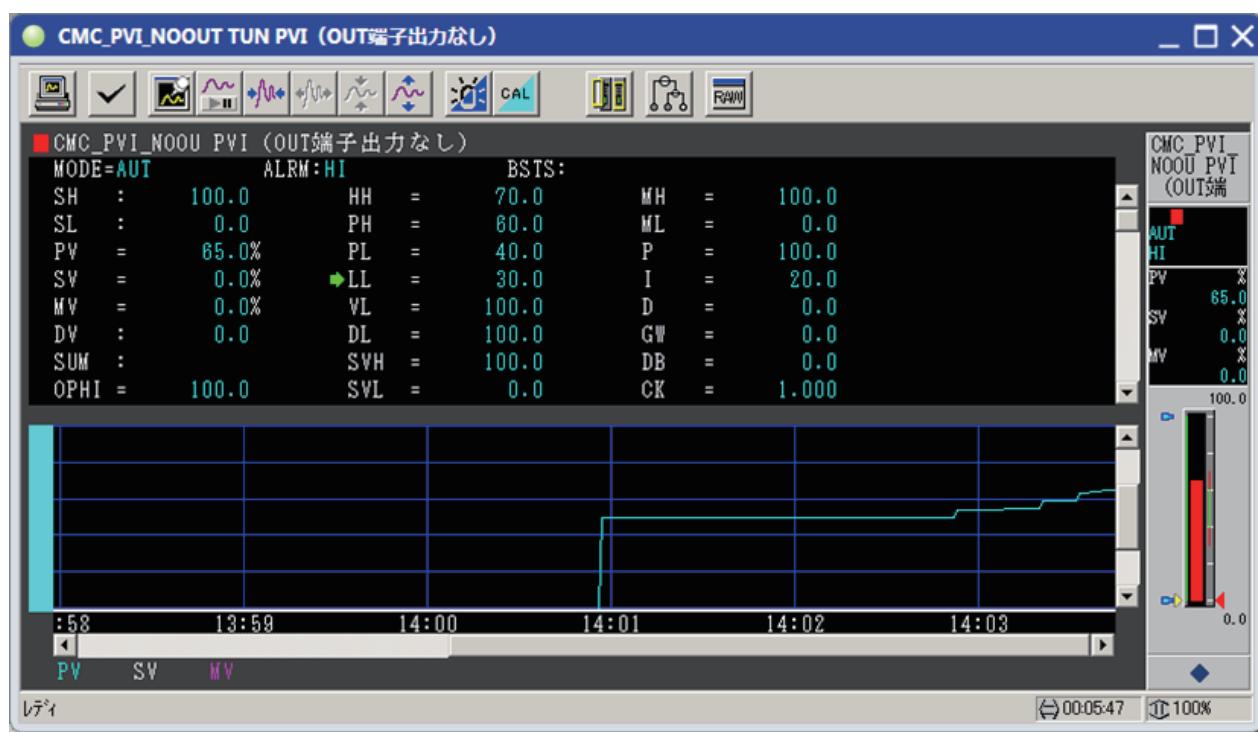


図 CMC_PVI_NOOUTのチューニングウィンドウ

さらに、MLD_PVI_NOOUT の MV 値を増加してください。MV 値が 70 を越えると、CMC_PVI_NOOUT の PV も 70（データアイテム HH の値）を越え、その結果 CMC_PVI_NOOUT は HH（入力上上限アラーム）を発生します。

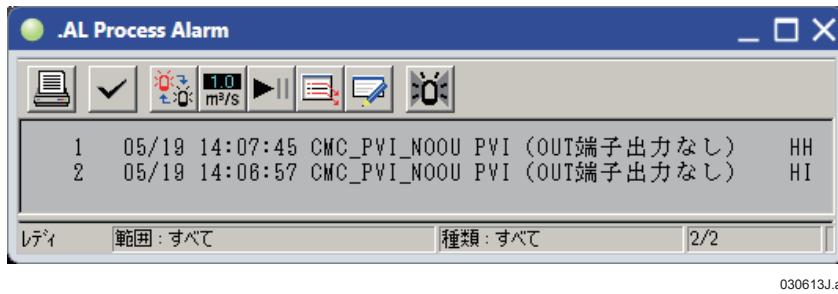


図 入力上上限アラームの発生

MLD_PVI_NOOUT の MV を増加したり減少したりして、CMC_PVI_NOOUT の PV 値が MLD_PVI_NOOUT の MV に追随するのを確認してください。また、CMC_PVI_NOOUT の PV がデータアイテム HH, PH, PL, LL の値を越えると、HH、HI、LO、LL アラームが発生したり復帰したりすることを確認してください。

ユーザカスタムアルゴリズムの C プログラムについて説明します。ソリューション _SMPL_PVI_NOOUT の pvi_noout.c から、機能ブロック定周期処理 (UcaBlockPeriodical で検索) を見つけます。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockPeriodical
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME   処理続行不能
*
* description:            機能ブロック定周期処理
*
* >>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnCode;          /* リターンコード */
    I32 rtnReadIn;        /* IN 端子入力リターンコード */

    /* IN 端子から入力信号を読み込みます */
    rtnReadIn = UcaRWReadIn(bc, NOOPTION);

    /* 入力信号を測定値 (PV) に設定します */
    rtnCode = UcaRWSetPv(bc, NULL, rtnReadIn, NOOPTION);

    return SUCCEED;
}
```

UcaBlockPeriodical は、ユーザカスタムブロックの実効スキャン周期ごとに定周期で呼び出されます。このプログラムは、ユーザカスタムアルゴリズム作成用ライブラリ UcaRWReadIn で IN 端子からデータを入力しています。UcaRWReadIn は、入力したデータを自ブロックのデータアイテム RV (入力値) に設定します。

次にユーザカスタムアルゴリズム作成用ライブラリ UcaRWSetPv を呼び出し、RV に保持されている入力値をデータアイテム PV (測定値) に設定しています。UcaRWSetPv は、PV に設定した測定値とデータアイテム HH, PH, LH, LL のデータを比較し、入力上限／下限アラーム (HI と LO) と入力上上限／下下限アラーム (HH と LL) を検出します。

参照

- UcaRWReadRv のように、アラームを検出するユーザカスタムアルゴリズム作成用ライブラリの詳細については、以下を参照してください。
[「4.3.2 アラームステータス」](#)
- UcaRWReadIn と UcaRWSetPv の動作の詳細については、以下を参照してください。
[「6. 連続制御形ユーザカスタムブロックのプログラミング」](#)

3.7 ユーザ定義関数の戻り値

機能ブロック定周期処理（UcaBlockPeriodical）などのユーザ定義関数は、ユーザカスタムブロック実行管理部から呼び出されます。ユーザ定義関数の戻り値（returnに指定するSUCCEEDやUCAERR_NOPROC）によるユーザカスタムブロック実行管理部の動作について説明します。

先に、ユーザ定義関数は、SUCCEED（処理成功）またはUCAERR_NOPROC（処理なしでリターンすることを説明しました。これらに加え、ユーザ定義関数の処理中に何らかの異常を検出し処理を継続できない場合に返す戻り値として、UCAERR_STOPMEがあります。機能ブロック定周期処理 UcaBlockPeriodical で何らかの重大な異常を検出し、UCAERR_STOPME でリターンする記述例を示します。

```
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc           /* (IN/OUT): ブロックコンテキスト */
)
{
    /* 機能ブロック定周期処理を記述 */
    .....

    if (<何らかの重大な異常を検出して続行できない>) {
        .....
        return UCAERR_STOPME;      /* 処理続行不可能 */
    }
    .....
    return SUCCEED;      /* 処理成功 */
}
```

ユーザ定義関数が UCAERR_STOPME でリターンすると、ユーザカスタムブロック実行管理部（システム）は戻り値 UCAERR_STOPME を「ユーザカスタムブロック異常レベル3」として扱い、ユーザカスタムブロックのブロックモードを O/S にし実行を停止します。

参照 異常発生時の処理の詳細については、以下を参照してください。

APCS ユーザカスタムブロック (IM 33J15U20-01JA)

機能ブロックワンショット起動処理 (UcaBlockOneshot) は引数のポインタ result が指す実行結果に条件判定の真偽を返します。また、機能ブロックデータ設定値特殊処理 (UcaBlockDset) は、引数のポインタ result が指す設定可否にデータ設定の可否を返します。これらのユーザ定義関数内で引数に設定する値と、ユーザ定義関数の戻り値によるユーザカスタムブロック実行管理部（システム）の処理を以下に示します。

表 ユーザ定義関数の戻り値によるユーザカスタムブロック実行管理部の動作

戻り値	ユーザカスタムブロック実行管理部（システム）の動作	
	機能ブロックワンショット起動処理の後	機能ブロックデータ設定時特殊処理の後
SUCCEED (処理成功)	UcaBlockOneshot が *result に TRUE を設定すれば真、FALSE を設定すれば偽と判定します。実行管理部は、判定結果の真偽を、ユーザカスタムブロックをワンショット起動した呼び出し元（シーケンステーブル）に返します。	UcaBlockDset が *result に UCADSET_ALLOW を設定すれば設定可能、UCADSET_DENY を設定すれば設定不可能と判定
UCAERR_NOPROC (処理なし)	ユーザカスタムブロック実行管理部は、ユーザカスタムブロックをワンショット起動した呼び出し元（シーケンステーブル）にエラーを返します。	常にデータ設定可能と判定
UCAERR_STOPME (処理続行不可能)	シーケンステーブルは、エラーを返されると前回の条件判定結果をそのまま使用します。	常にデータ設定可能と判定
その他		常にデータ設定可能と判定

3.8 ユーザカスタムアルゴリズム作成用ライブラリ一覧

ユーザカスタムアルゴリズム作成用ライブラリは、システムが提供する関数群です。入出力端子アクセス、制御演算、機能ブロックデータアクセスなどの関数が用意されています。

参照 各々の関数インターフェースの詳細については、以下を参照してください。
APCS ユーザカスタムブロックライブラリ (IM 33J15U22-01JA)

ユーザカスタムアルゴリズム作成用ライブラリには、ソースコードが公開される関数とソースコードが公開されない関数があります。ソースを公開しない関数は、システム内部のデータにアクセスするユーザカスタムブロック作成用ライブラリの核となる関数群です。

これに対しソースコードが公開される関数は、ユーザが汎用的に使用できる処理をソースコード非公開の関数を使用して実現した関数です。これらはユーザ自身のプログラミングの参考とするために、ユーザカスタムブロック開発環境パッケージでソースコードを公開しています。CENTUM VP インストール先の以下フォルダにソースコードが配置されています。

<CENTUM VP インストール先>¥UcaEnv¥Sample¥source¥libuca¥ <関数のソースファイル>

フォルダ libuca には 1 つ 1 つの関数ごとに関数と同じ名前のファイルにプログラムが記述されていますので、ソースコードを簡単に参照することができます。

ユーザカスタムアルゴリズムには、連続制御形ユーザカスタムブロックでのみ使用可能な関数や、汎用演算形ユーザカスタムブロックでのみ使用可能な関数があります。また、機能ブロック初期化処理、機能ブロック終了処理、および機能ブロックデータ設定時特殊処理では自ブロックのデータにはアクセスできますが、他の機能ブロックのデータにアクセスすることはできません。

参照 機能ブロック初期化処理、機能ブロック終了処理、および機能ブロックデータ設定時特殊処理の詳細については、以下を参照してください。
「3.2 機能ブロック初期化処理」
「3.3 機能ブロック終了処理」

ここではこのような「使用可否」の分類とともに、ユーザカスタムアルゴリズム作成用ライブラリの一覧を示します。

表 ブロックモードおよびステータス

分類	関数名	説明	CSTM-C		CSTM-A		ソース公開(*3)
			初期化終了デ特(*1)	定周期ショット(*2)	初期化終了デ特(*1)	定周期ショット(*2)	
ブロックモード	UcaModeSet	ブロックモード設定	○	○	○	○	
	UcaModeGet	ブロックモード取得	○	○	○	○	
	UcaModeGetPrev	前回モード取得	○	○	○	○	
ブロックステータス	UcaBstsSetExclusive	BSTS を排他的に設定	○	○	○	○	○
	UcaBstsSet	BSTS を設定	○	○	○	○	
	UcaBstsClear	BSTS をクリア	○	○	○	○	
	UcaBstsGet	BSTS を取得	○	○	○	○	
プロセスアラーム	UcaAlrmSet	アラームを設定		○		○	
	UcaAlrmClear	アラームをクリア		○		○	
	UcaAlrmGet	アラームを取得		○		○	
データステータス	UcaDstsIsGood	正常か判定	○	○	○	○	
	UcaDstsIsBad	BAD か判定	○	○	○	○	
	UcaDstsIsQst	QST か判定	○	○	○	○	
	UcaDstsIsPtpf	出力先異常か判定	○	○	○	○	
	UcaDstsIsCnd	コンディショナルか判定	○	○	○	○	
	UcaDstsIsCal	CAL か判定	○	○	○	○	
	UcaDstsSetBad	BAD を設定	○	○	○	○	
	UcaDstsClearBad	BAD を解除	○	○	○	○	
	UcaDstsSetQst	QST を設定	○	○	○	○	
	UcaDstsClearQst	QST を解除	○	○	○	○	
	UcaDstsSetCnd	コンディショナルを設定	○	○	○	○	
	UcaDstsClearCnd	コンディショナルを解除	○	○	○	○	
	UcaDstsChooseBadOrQst	BAD または QST を作成	○	○	○	○	○

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>¥UcaEnv¥sample¥source¥libuca にソースファイルが公開されている関数です。○印がソース公開を表します。

表 入出力結合

分類	関数名	説明	CSTM-C		CSTM-A		ソース公開(*3)
			初期化終了デ特(*1)	定周期ショット(*2)	初期化終了デ特(*1)	定周期ショット(*2)	
低水準入出力	UcaRWRead	結合端子データ参照	○		○		
	UcaRWWrite	結合端子データ設定	○		○		
	UcaRWReadback	出力読み返し	○		○		
	UcaRWReadString	文字列データ参照	○		○		
	UcaRWWriteString	文字列データ設定	○		○		
	UcaWRWWritePvToJnSub	PV を Jnn 端子に設定	○				○
高水準入出力	UcaRWReadIn	IN 端子データ参照	○		○		
	UcaRWSetPv	PV 作成	○				
	UcaRWSetCPv	CPV 作成			○		
	UcaRWWriteMv	OUT 端子データ設定	○				
	UcaRWReadbackMv	OUT 端子読み返し	○				
	UcaRWWriteCpvToOutSub	CPV を OUT 端子に設定			○	○	
	UcaRWWriteCpv	OUT 端子データ設定			○		
	UcaRWReadbackCpv	OUT 端子読み返し			○		
	UcaRWCheckOutputCondition	出力トラッキング状態の検査				○	
	UcaRWWriteSub	SUB 端子データ設定	○		○		
	UcaRWReadTrack	TSI、TIN 端子入力	○				
	UcaRWReadBin	BIN 端子入力	○				
	UcaRWReadRI	RL1、RL2 端子入力	○				
シーケンス入出力	UcaRWSqCond	シーケンス条件判定	○		○		
	UcaRWSqOprt	シーケンス状態操作	○		○		
	UcaRWSqCon	シーケンス結合可	○		○		

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>¥UcaEnv¥sample¥source¥libuca にソースファイルが公開されている関数です。○印がソース公開を表します。

表 自ブロックデータ

分類	関数名	説明	CSTM-C		CSTM-A		ソース公開(*3)
			初期化終了デ特(*1)	定周期ショット(*2)	初期化終了デ特(*1)	定周期ショット(*2)	
プロセスデータ	UcaDataCheckDv	偏差警報チェック		○			
	UcaDataConvertRange	制御出力レンジへの変換		○			
	UcaDataConvertRange_p	制御出力レンジへの変換		○		○	
	UcaDataSvPushback	設定値プッシュバック		○			
	UcaDataGetReadback	OUT 端子読み返し値を取得		○		○	
	UcaDataGetTrack	TIN 端子入力値を取得		○			
	UcaDataGetMvbb	前回出力データ（出力保証前）を取得		○			
	UcaDataGetMvbl	前回出力データ（出力リミット前）を取得		○			
自ブロックデータアイテム	UcaDataGet?? (?の部分をデータアイテム名に置き換えます。各データアイテムごとに関数があります)	データを取得	○	○	○	○	
	UcaDataStore?? (?の部分をデータアイテム名に置き換えます。各データアイテムごとに関数があります)	データを保存	○	○	○	○	
	UcaDataGetEachSn	Sn から取得	○	○	○	○	
	UcaDataStoreEachSn	Sn へ保存	○	○	○	○	
	UcaDataStoreF64SToMvn	MVn へ型変換して保存			○	○	○
その他	UcaDataGetTagName	タグ名を取得	○	○	○	○	
	UcaDataStoreErrorNumber	ERRC、ERRL を設定	○	○	○	○	

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>¥UcaEnv¥sample¥source¥libuca にソースファイルが公開されている関数です。○印がソース公開を表します。

表 ビルダ定義項目

分類	関数名	説明	CSTM-C		CSTM-A		ソース公開(*3)
			初期化終了デ特(*1)	定周期ショット(*2)	初期化終了デ特(*1)	定周期ショット(*2)	
ビルダ定義項目	UcaConfigNonlinearGain	非線形ゲイン指定	○	○			
	UcaConfigCompensation	入出力補償指定	○	○			
	UcaConfigDirection	制御動作方向指定	○	○			
	UcaConfigOutput	制御動作指定	○	○			
	UcaConfigDeadband	不感帯指定	○	○			
	UcaConfigDeadbandHys	ヒステリシス指定	○	○			
	UcaConfigGeneral	汎用指定項目の取得	○	○	○	○	

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>¥UcaEnv¥sample¥source¥libucaにソースファイルが公開されている関数です。○印がソース公開を表します。

表 制御演算

分類	関数名	説明	CSTM-C		CSTM-A		ソース公開(*3)
			初期化終了デ特(*1)	定周期ショット(*2)	初期化終了デ特(*1)	定周期ショット(*2)	
制御演算ハンドラ	UcaCtrlHandler	制御演算をする CSTM-C の主関数		○			
制御演算初期化	UcaCtrlInitSet	制御演算初期化指示	○	○	○	○	
	UcaCtrlInitClear	制御演算初期化解除	○	○	○	○	
	UcaCtrlInitCheck	制御演算初期化確認	○	○	○	○	
PID 演算	UcaCtrlPidInit	PID 演算初期化		○			
	UcaCtrlPid	PID 演算（非線形ゲインを含む）		○			
	UcaCtrlPidTiming	制御周期の計算		○			
	UcaCtrlPidGetTime	制御周期カウンタの取得		○			
	UcaCtrlPidPutTime	制御周期カウンタの保存		○			
制御ホールド	UcaCtrlHoldSet	制御ホールド指示		○			
	UcaCtrlHoldClear	制御ホールド解除		○			
	UcaCtrlHoldCheck	制御ホールド確認		○			
非線形ゲイン	UcaCtrlGetGapGainCoef_p	ギャップゲイン係数の取得		○			
	UcaCtrlGapGain_p	ギャップゲインの計算		○			
	UcaCtrlDvSquareGain_p	偏差二乗ゲインの計算		○			
入出力補償	UcaCtrlCompensation	入出力補償		○			
	UcaCtrlCompensation_p	入出力補償の計算		○			
不感帯動作	UcaCtrlDeadband	不感帯		○			
	UcaCtrlDeadband_p	不感帯の計算		○			
リセットリミット	UcaCtrlResetLimitOprt	リセットリミッタ		○			
	UcaCtrlResetLimitOprt_p	リセットリミッタ		○			

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>¥UcaEnv¥sample¥source¥libucaにソースファイルが公開されている関数です。○印がソース公開を表します。

表 機能ブロックデータ (1/2)

分類	関数名	説明	CSTM-C		CSTM-A		ソース 公開 (*3)
			初期化 終了 デ特(*1)	定周期 ショット (*2)	初期化 終了 デ特(*1)	定周期 ショット (*2)	
自ステーションタグデータ	UcaTagReadF64S	自ステーションタグデータ参照 (数値データ)		○		○	
	UcaTagReadI32S	自ステーションタグデータ参照 (整数データ)		○		○	
	UcaTagReadString	自ステーションタグデータ参照 (文字列データ)		○		○	
	UcaTagRead	自ステーションタグデータ参照 (任意のデータ型)		○		○	
	UcaTagWriteF64S	自ステーションタグデータ設定 (数値データ)		○		○	
	UcaTagWriteI32S	自ステーションタグデータ設定 (整数データ)		○		○	
	UcaTagWriteString	自ステーションタグデータ設定 (文字列データ)		○		○	
ワンショット起動	UcaTagWrite	自ステーションタグデータ設定 (任意のデータ型)		○		○	
	UcaTagOneshot	自ステーション機能ブロックワンショット起動		○		○	
他ステーションタグデータ	UcaOtherTagReadF64S	他ステーションタグデータ参照 (数値データ)		○		○	
	UcaOtherTagReadI32S	他ステーションタグデータ参照 (整数データ)		○		○	
	UcaOtherTagRead	他ステーションタグデータ参照 (任意のデータ型)		○		○	
	UcaOtherTagWriteF64S	他ステーションタグデータ設定 (数値データ)		○		○	
	UcaOtherTagWriteI32S	他ステーションタグデータ設定 (整数データ)		○		○	
	UcaOtherTagWrite	他ステーションタグデータ設定 (任意のデータ型)		○		○	
タグデータ 参照 (RVnn 設定 付き)	UcaTagReadToRvnF64S	自ステーションタグデータ参照 (RVnn 設定付き)		○		○	○
	UcaOtherTagReadToRvnF64S	他ステーションタグデータ参照 (RVnn 設定付き)		○		○	○

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>\UcaEnv\sample\source\libucaにソースファイルが公開されている関数です。○印がソース公開を表します。

表 機能ブロックデータ (2/2)

分類	関数名	説明	CSTM-C		CSTM-A		ソース 公開 (*3)
			初期化 終了 デ特(*1)	定周期 ショット (*2)	初期化 終了 デ特(*1)	定周期 ショット (*2)	
自ステーション タグデータ 一次元配列一括 アクセス	UcaTagArray1ReadF64S	機能ブロック一次元配列 一括参照 (数値データ)		○		○	○
	UcaTagArray1ReadI32S	機能ブロック一次元配列 一括参照 (整数データ)		○		○	○
	UcaTagArray1WriteF64S	機能ブロック一次元配列 一括設定 (数値データ)		○		○	○
	UcaTagArray1Writel32S	機能ブロック一次元配列 一括設定 (整数データ)		○		○	○
他ステーション タグデータ 一次元配列一括 アクセス	UcaOtherTagArray1ReadF64S	機能ブロック一次元配列 一括参照 (数値データ)		○		○	○
	UcaOtherTagArray1ReadI32S	機能ブロック一次元配列 一括参照 (整数データ)		○		○	○
	UcaOtherTagArray1WriteF64S	機能ブロック一次元配列 一括設定 (数値データ)		○		○	○
	UcaOtherTagArray1Writel32S	機能ブロック一次元配列 一括設定 (整数データ)		○		○	○

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>¥UcaEnv¥sample¥source¥libucaにソースファイルが公開されている関数です。○印がソース公開を表します。

表 溫圧補正演算

分類	関数名	説明	CSTM-C		CSTM-A		ソース 公開 (*3)
			初期化 終了 デ特(*1)	定周期 ショット (*2)	初期化 終了 デ特(*1)	定周期 ショット (*2)	
温圧補正	UcaCalcTc	温度補正 (温度単位：摂氏)	○	○	○	○	
	UcaCalcPcp	圧力補正 (圧力単位：Pa)	○	○	○	○	
	UcaCalcPckp	圧力補正 (圧力単位：kPa)	○	○	○	○	
	UcaCalcPcmp	圧力補正 (圧力単位：MPa)	○	○	○	○	
	UcaCalcTpccp	温圧補正（摂氏；Pa）	○	○	○	○	
	UcaCalcTpckp	温圧補正（摂氏；kPa）	○	○	○	○	
	UcaCalcTpcmp	温圧補正（摂氏；MPa）	○	○	○	○	
ASTM 補正	UcaCalcAstm1	ASTM 補正 1 (旧 JIS)	○	○	○	○	
	UcaCalcAstm2	ASTM 補正 2 (新 JIS (原油))	○	○	○	○	
	UcaCalcAstm3	ASTM 補正 3 (新 JIS (燃料油))	○	○	○	○	
	UcaCalcAstm4	ASTM 補正 4 (新 JIS (潤滑油))	○	○	○	○	

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>¥UcaEnv¥sample¥source¥libucaにソースファイルが公開されている関数です。○印がソース公開を表します。

表 その他

分類	関数名	説明	CSTM-C		CSTM-A		ソース公開(*3)
			初期化終了デ特(*1)	定周期ショット(*2)	初期化終了デ特(*1)	定周期ショット(*2)	
データ型変換	UcaDataConvertType	データ型の変換	○	○	○	○	
時間管理	UcaTimeResetLocalCounter	計時開始	○	○	○	○	
	UcaTimeGetLocalCounter	経過時間	○	○	○	○	
メッセージ	UcaMesgSendSystemAlarm	システムアラームメッセージの送信	○	○	○	○	
	UcaMesgSendPrintMessage	印字メッセージの送信	○	○	○	○	
	UcaMesgSendHistoricalMessage	ヒストリカルメッセージの送信	○	○	○	○	
FPU例外	UcaFpuExpClear	FPU例外発生フラグのクリア	○	○	○	○	
	UcaFpuExpCheck	FPU例外発生フラグの確認	○	○	○	○	
時間	UcaTime	システム時刻取得	○	○	○	○	○
	UcaLocaltime	現地時間で変換	○	○	○	○	○
	UcaGmtime	万国標準時で変換	○	○	○	○	○
	UcaMktime	通算時間に変換	○	○	○	○	○

*1：表の「初期化 終了 デ特」欄は、機能ブロック初期化処理、機能ブロック終了処理と機能ブロックデータ設定時特殊処理を表します。○印が使用可能を表します。

*2：「定周期 ショット」は、機能ブロック定周期起動処理、機能ブロックワンショット起動処理を表します。○印が使用可能を表します。

*3：「ソース公開」は、ユーザカスタムブロック開発環境パッケージの<CENTUM VP インストール先>¥UcaEnv¥sample¥source¥libucaにソースファイルが公開されている関数です。○印がソース公開を表します。

3.8.1 ユーザカスタムアルゴリズム作成用ライブラリのエラーコード

ユーザカスタムアルゴリズム作成用ライブラリの関数（以下、ライブラリ関数）が返す戻り値を、ユーザが記述するCプログラムでどのように扱うかを説明します。ライブラリ関数は、I32型でエラーコードを返します。正常時には値SUCCEED（ゼロ）を、エラー時にはエラーコード（ゼロ以外）を返します。ユーザは戻り値がSUCCEEDか否かを判定することにより、正常かエラーかがわかります。

通常ユーザカスタムアルゴリズムのプログラムでは、ライブラリ関数が返す戻り値により処理を分ける必要はありません。ユーザカスタムアルゴリズムで扱うデータにはデータステータスが付加されています。ライブラリ関数は内部でデータステータスを判定して処理を分けていますので、ユーザプログラムでライブラリ関数の戻り値により処理を分けなくても大丈夫です。

ライブラリ関数が返す戻り値により処理を分けない場合でも、変数rtnCodeに戻り値を格納する記述をすると、Visual C++でデバッグするときに変数rtnCodeの値でライブラリ関数の返す値を見るることができます。本書では、常に以下のようにライブラリ関数の戻り値をrtnCode変数に格納することを推奨します。また本書のサンプルはすべてそのようにコーディングされています。

```
/* データアイテム P01 の値を変数 p01 に読みこむ */
rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);
```

ユーザカスタムブロックには、システムがエラーコードを格納するデータアイテムERRAがあります。ライブラリ関数がエラーリターンするときにはユーザカスタムブロックのデータアイテムERRAに戻り値で戻すのと同じエラーコードがシステムにより設定されます。つまり、データアイテムERRAには最後にライブラリ関数が検出したエラーコードが保持されます。

参照 エラーコードの詳細については、以下を参照してください。
APCS ユーザカスタムブロックライブラリ (IM 33J15U22-01JA)

ユーザカスタムアルゴリズム作成用ライブラリの関数が返すエラーコードについて説明します。

- ライブラリ関数自身が検出するエラーコードは、0xE500～0xE5FFの範囲の整数です。
- ライブラリ関数が0xE5**以外のエラーコードを返す場合があります。これはライブラリ関数が使用しているシステム内部処理のエラーコードです。0xE5**以外のエラーコードは、CENTUM VPのプロセス制御用プログラム言語SEBOLの詳細エラーコードと同じです。

参照 詳細エラーコードの詳細については、以下を参照してください。
SEBOL リファレンス (IM 33J05L20-01JA) 「13.4 詳細エラーコード」

Blank Page

4. ユーザカスタムアルゴリズムの開発

この章では、ユーザカスタムアルゴリズム開発の流れについて説明します。

ユーザカスタムアルゴリズム開発の流れを以下に示します。

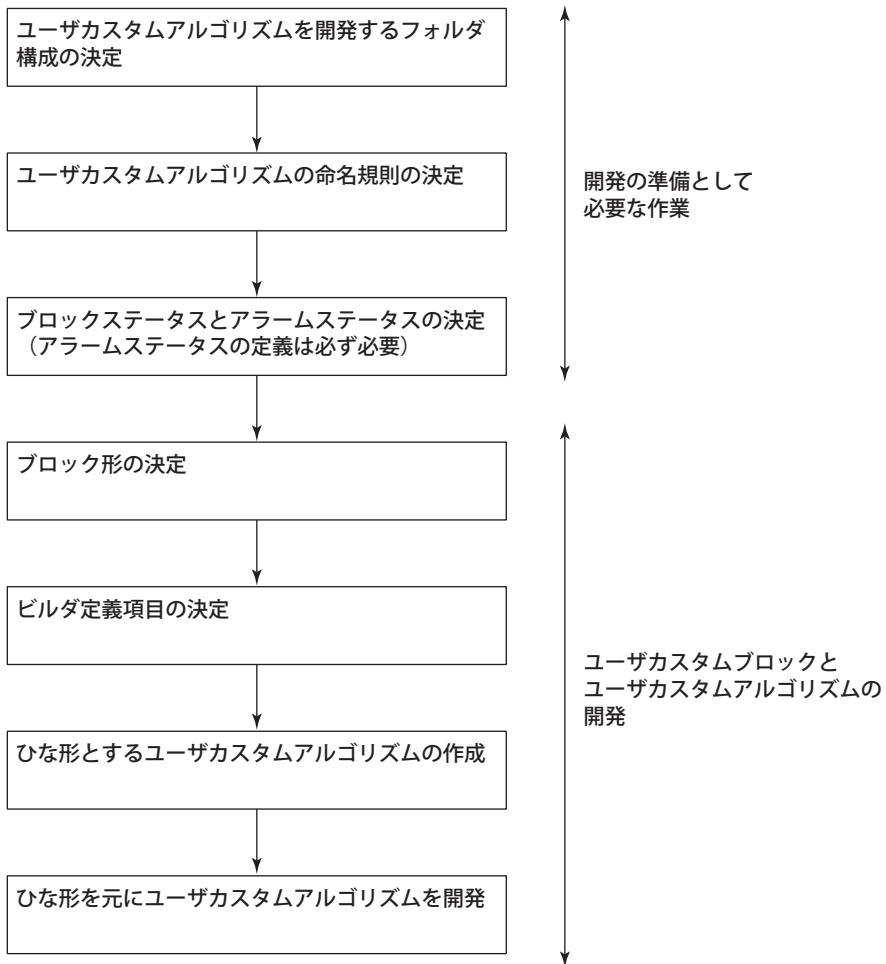


図 ユーザカスタムアルゴリズム開発の流れ

4.1 ユーザカスタムアルゴリズム開発のフォルダ構成

ユーザカスタムアルゴリズム開発を行うコンピュータのWindowsファイルシステムに、ユーザカスタムアルゴリズム開発のためのフォルダを作成します。これはVisual C++で開発するCプログラムを配置するフォルダとなります。ここでは、ユーザカスタムブロック開発環境パッケージに用意されている作業フォルダの構成を使用します。

作業に使用するハードディスクドライブを決めて、インストールされているユーザカスタムアルゴリズム開発のためのフォルダ構成をコピーします。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥UcaWork

このCENTUM VP インストール先より、UcaWork フォルダを含めて、作業する Windows のファイルシステムに<ドライブ名> ¥UcaWork となるようにコピーします。

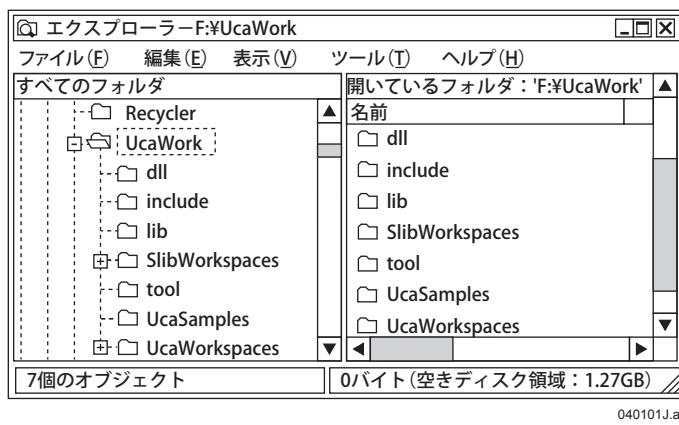


図 ユーザカスタムアルゴリズム開発のためのフォルダ構成

この構成とフォルダ名はシステムが規定するものではなく、ユーザ自身が決めることも可能です。ただし、本書のサンプルプログラムは「図 ユーザカスタムアルゴリズム開発のためのフォルダ構成の構成」を前提としていますので、本書のサンプルを使用して学習する場合には同じフォルダ構成で作業してください。

ユーザカスタムアルゴリズムの開発を始める前に、開発作業のフォルダ構成を決定してください。

- VC++ のパス設定などに精通しているプログラマは、「図 ユーザカスタムアルゴリズム開発のためのフォルダ構成」のフォルダの名前や配置を自由に変更しても構いません。この場合は、VC++ のパスなど必要な設定を自分で変更してください。
- VC++ のパス設定などに精通していない場合には、「図 ユーザカスタムアルゴリズム開発のためのフォルダ構成」の構成をそのまま使用し、VC++ の設定を本書の説明と同一にしてください。

「図 ユーザカスタムアルゴリズム開発のためのフォルダ構成」の各フォルダの用途を以下に示します。

表 ユーザカスタムアルゴリズム開発のフォルダ

フォルダ名	用途
dll	ユーザ定義ダイナミックリンクライブラリに対応する LIB ファイルを配置します。
include	ユーザ定義の共通インクルードファイルを配置します。
lib	ユーザ定義静态ライブラリを配置します。
SlibWorkspaces	ユーザ定義静态ライブラリのソリューションを配置します。
Tool	ユーザが作成したツールを配置します。
UcaSamples	ユーザカスタムプロック開発環境パッケージ内のサンプルのソリューションをコピーするフォルダです。
UcaWorkspaces	ユーザ自身が作成するユーザカスタムアルゴリズムのソリューションを配置します。また、ユーザ定義ダイナミックリンクライブラリは、このフォルダに配置します。

補足

VC++ では 1 つのソリューションで複数プロジェクトを構成可能ですが、ユーザカスタムアルゴリズムは 1 つのソリューションで 1 つのプロジェクトを構成して開発します。

4.2 ユーザカスタムアルゴリズムの命名規則

ユーザカスタムアルゴリズムに関する名前の規則について説明します。

システムが規定している名前は、次の2種類あります。

● ユーザカスタムアルゴリズム名

16文字以内の半角英数字および_（アンダースコア）で命名します。ただし、先頭文字に数字は使用できません。

- ・大文字／小文字の区別はありません。ユーザカスタムアルゴリズム名に小文字を含めても、CENTUM VP に取り込むとすべて大文字で扱われます。
- ・ユーザカスタムアルゴリズム名が重複しないようにしてください。
- ・以下のユーザカスタムアルゴリズム名はシステムが使用するライブラリ名と重複するので、ユーザは使用しないでください。

表 ユーザが使用できないユーザカスタムアルゴリズム名

名前	説明	理由
LIBUCA***	Libuca ではじまる名称	ユーザカスタムアルゴリズムで使用
LIBB***	Libb ではじまる名称	CENTUM VP システムで使用
YUC***	YUC ではじまる名称	CENTUM VP システムで使用
Z***	Z ではじまる名称	特注プログラムで使用
_S***から_Z***	_S、_T、..._Z で始まる名称	システムが予約

● C言語プログラムにおける名前

以下の関数名やラベル名はシステムが使用するのでユーザは使用しないでください。

表 ユーザカスタムアルゴリズム（C言語プログラム）でユーザが使用できない名前

名前	説明	理由
Uca****	Uca で始まる名前	ユーザカスタムアルゴリズム作成用 ライブラリで使用
UCA***	UCA で始まる名前	システム定義の #define 名などで使用
大文字小文字に関係なく UCA で始まる名前		システムが予約
大文字小文字に関係なく YUC で始まる名前		システムが予約

以上が、システムが規定する名前の規則です。ユーザはユーザカスタムアルゴリズムの開発を始める前に、名前の規則の範囲内でユーザが作成するユーザカスタムアルゴリズムと関数名など命名規則を決定してください。

4.3 ブロックステータスとアラームステータスの決定

ユーザカスタムブロックのブロックステータスとアラームステータスは、ユーザ定義可能です。ブロックステータスはCENTUM VPプロジェクトのCOMMONにあるBlkStsLabelのUSER8、アラームステータスはAlmStsLabelのUSER16に定義します。

重要

- アラームステータスの定義は、ユーザカスタムブロックを使用するために必ず必要です。
- ブロックステータスとアラームステータスの定義は、オフラインでのみ変更可能で
す。オンラインで変更することはできません。

4.3.1 ブロックステータス

ブロックステータスを定義する手順を説明します。

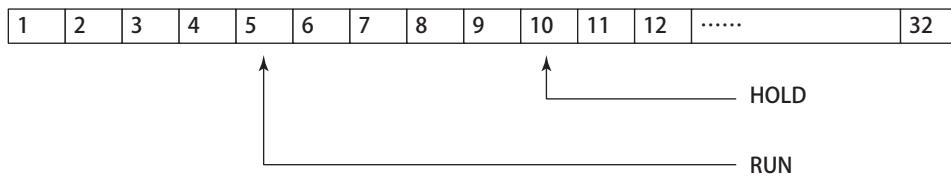
システムビューでCENTUM VPプロジェクトのCOMMONにあるBlkStsLabelをダブルクリックすると、ユーザ定義ステータス文字列ビルダが表示されます。ブロックステータス定義のUSER8がユーザカスタムブロックのブロックステータスの定義です。ブロックステータスにはシステムが与える初期値はありません。ユーザが自由に定義します。USER8にブロックステータスを定義し、[ファイル] - [上書き保存]で書き込みます。



図 ユーザ定義ステータス文字列ビルダ

040302J.ai

ここでは、ビット5にRUN、ビット10にHOLDというブロックステータスを定義しています。



040303J.ai

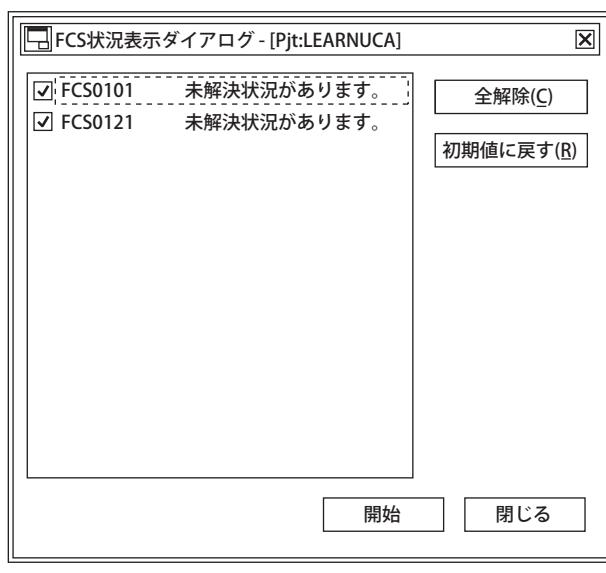
図 ブロックステータスの定義

ユーザカスタムブロックのブロックステータス（データアイテム BSTS）は、内部的には32ビットの整数です。ここではビット番号5にRUN、ビット番号10にHOLDを定義しました。BSTSの5ビットを1にするとBSTSが「RUN」になります。また、BSTSの10ビットを1にするとBSTSが「HOLD」になります。5ビットと10ビットの両方が1になると、BSTSは「RUN」となります。これはビット番号の小さい方が優先されるからです。なお、ブロックステータスの設定はユーザカスタムアルゴリズム作成用ライブラリUcaBstsSetExclusive()またはUcaBstsSet()、解除はUcsBstsClear()で行います。

参照 ブロックステータスの設定の詳細については、以下を参照してください。

[「4.3.4 ブロックステータスの操作」](#)

ブロックステータスの定義を反映します。システムビューの【プロジェクト】－【無効素子の解決】で以下のダイアログを呼び出し、【開始】ボタンをクリックします。



040304J.ai

図 FCS状況表示ダイアログ

以下のダイアログでブロックステータス定義の変更が FCS0121 (APCS) に反映されたことがわかります。[閉じる] ボタンでダイアログを閉じます。



040305J.ai

図 ブロックステータス定義変更の確認

ユーザカスタムブロックには CSTM-C (連続制御形ユーザカスタムブロック) と CSTM-A (汎用演算形ユーザカスタムブロック) の 2 種類ありますが、両方とも USER8 の定義を使用します。つまり、この例では CENTUM VP プロジェクト内のすべてのユーザカスタムブロックは、RUN または HOLD というブロックステータスを持つことができます。

- ・ ブロックステータスは、CENTUM VP プロジェクトの COMMON の BlkStsLabel に定義します。
- ・ ユーザカスタムブロックのブロックステータスは USER8 に定義します。USER8 以外に定義することはできません。
- ・ ブロックステータスは、8 文字 (8 バイト) 以下の英字で始まる半角英数字または、_ (アンダースコア) です。英字は大文字で指定します。計器図に表示されるのは先頭 4 文字のみです。チューニングウィンドウには 8 文字すべて表示されます。
- ・ USER8 の中に同じ文字列を複数定義することはできません。
- ・ ブロックステータスのビット番号が小さいほど優先度が高くなります。複数のビットが 1 になった場合、CENTUM VP のプログラム言語 SEBOL によるデータアイテム BSTS の取得や操作監視機能の計器図の表示における BSTS の値は、もっとも優先度の高いブロックステータスとなります。BSTS は内部的には 32 ビットの整数ですが、SEBOL や計器図の表示では文字列として扱われます。
- ・ 将来の追加に備え、適当にビット番号に空きを設けておきます。例では 5 ビットに RUN、10 ビットに HOLD を定義しています。RUN より優先度の高いブロックステータスをビット 1 ~ 4 に追加可能ですし、RUN と HOLD の間にブロックステータスを追加することもできます。

ブロックステータスとビット番号の対応を変更すること（たとえば RUN のビット番号を 5 から 7 にする）は可能です。ただし、エンジニアリングの最初によく検討し、後で変更が発生しないようにしてください。

参照 ブロックステータスとビット番号の対応の変更の詳細については、以下を参照してください。
[「4.3.3 ユーザ定義インクルードファイル usrstatus.h」](#)

4.3.2 アラームステータス

アラームステータスを定義する手順を説明します。

システムビューでCENTUM VP プロジェクトのCOMMONにあるAlmStsLabelをダブルクリックすると、ユーザ定義ステータス文字列ビルダが表示されます。アラームステータス定義のUSER16がユーザカスタムブロックのアラームステータス定義です。[編集] – [ユーザカスタムブロックデフォルトの取り込み] の操作でUSER16にアラームステータスのデフォルトが設定されますので、[ファイル] – [上書き保存] で書き込みます。

重要

USER16にデフォルトのアラームステータスを定義しAPCSに反映する操作は、ユーザカスタムブロックを使用するために必ず必要です。



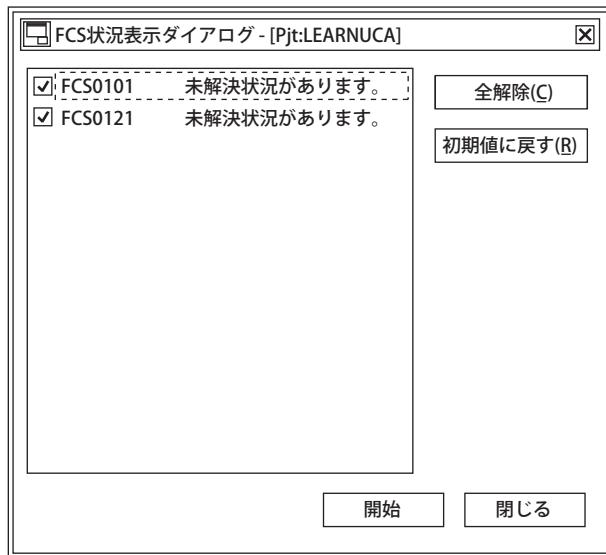
図 ユーザ定義ステータス文字列ビルダ

040306J.ai

参照 アラームの定義については、以下を参照してください。

[操作監視リファレンス Vol.2 \(IM 33J05A11-01JA\) 「7.4 アラームステータスの文字列とアラーム処理」](#)

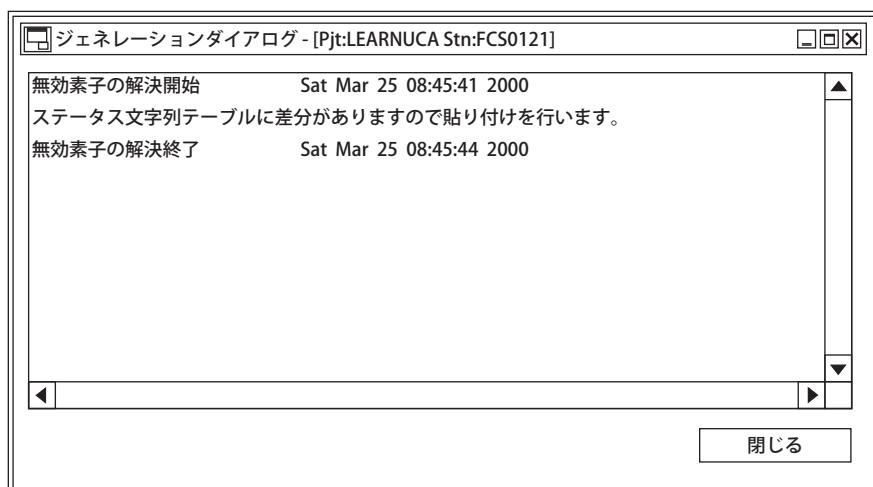
アラームステータスの定義を反映します。システムビューの [プロジェクト] – [無効素子の解決] で以下のダイアログを呼び出し、[開始] ボタンをクリックします。



040307J.ai

図 FCS状況表示ダイアログ

以下のダイアログでブロックステータス定義の変更が FCS0121 (APCS) に反映されたことがわかります。[閉じる] ボタンでダイアログを閉じます。



040308J.ai

図 ブロックステータス定義変更の確認

ユーザカスタムブロックデフォルトの取り込み操作で設定されるアラームステータスのうち、ERRC はユーザカスタムブロック固有です。ERRC 以外のアラームステータスの意味は、PID 調節ブロックなどの標準ブロックと同じです。

参照 ERRC アラームの詳細については、以下を参照してください。
[「5.2.2 演算異常ERRCアラームの発生と復帰」](#)

■ デフォルトのアラームの発生と復帰

デフォルトのアラームは、ユーザカスタムアルゴリズム作成用ライブラリの関数が発生・復帰します。デフォルトのアラームを以下の表に示します。表の「ユーザカスタムアルゴリズム作成用ライブラリ」欄の関数が該当するアラームを発生・復帰します。「制御」は、連続制御形ユーザカスタムブロック (CSTM-C)、「演算」は汎用演算形ユーザカスタムブロック (CSTM-A) を示し、○印はアラーム検出することを示します。

表 ユーザカスタムブロックのアラームステータスのデフォルト (1/2)

ビット番号	シンボル	名称	ユーザカスタムアルゴリズム作成用ライブラリ	制御	演算	説明
1～8		(使用不可)				システムが予約しています。
9	NR	正常状態				アラームがまったく発生していない状態です。
10	OOP	出力オープンアラーム	UcaCtrlHandler UcaRWWriteMvToOutSub UcaRWReadbackMv UcaRWWriteMv	○		操作端、プロセス入出力機器の故障／断線、あるいは出力先データの異常を原因として、出力のデータステータスが PTPF (出力フェイル) となる状態です。通常、出力機能が停止します。
			UcaRWWriteCpvToOutSub (*1) UcaRWReadbackCpv (*1) UcaRWWriteCpv (*1)		○	
			UcaRWWrite (*1) UcaRWReadback (*1)	○	○	
11	IOP	上限入力オープンアラーム	UcaRWReadIn	○	○	検出端、プロセス入出力機器の故障／断線、あるいは入力先データの異常を原因として、入力のデータステータスが BAD となる状態です。通常、入力信号を使う処理が停止します。要因が断線などによる入力振り切れの場合は、上限方向への振り切れとなった状態です。
12	IOP-	下限入力オープンアラーム				入力信号の断線などにより下限方向へ入力振り切れとなった状態です。入力のデータステータスが BAD となり、通常、入力信号を使う処理が停止します。
13	HH	上上限アラーム	UcaRWSetPv	○		測定値が上上限アラーム設定値を超える状態です。
14	LL	下下限アラーム				測定値が下下限アラーム設定値を下まわる状態です。
15～16		(ユーザ定義)				ユーザが決定します。
17	HI	上限アラーム	UcaRWSetPv	○		測定値が上限アラーム設定値を超える状態です。
18	LO	下限アラーム				測定値が下限アラーム設定値を下まわる状態です。
19～20		(ユーザ定義)				ユーザが決定します。
23～24		(ユーザ定義)				ユーザが決定します。

*1：オプション指定時にのみアラームを検出します。

表 ユーザカスタムブロックのアラームステータスのデフォルト (2/2)

ビット番号	シンボル	名称	ユーザカスタムアルゴリズム作成用ライブラリ	制御	演算	説明
21	DV+	正方向偏差アラーム	UcaCtrlHandler UcaDataCheckDv	○		測定値と設定値の偏差が正方向に偏差アラーム設定値を超える状態です。
22	DV-	負方向偏差アラーム				測定値と設定値の偏差が負方向に偏差アラーム設定値を超える状態です。
23～24		(ユーザ定義)				ユーザが決定します。
25	VEL+	正方向変化率アラーム	UcaRWSetPv	○		入力信号の指定時間内の変化量が正方向に変化率アラーム設定値を超える状態です。
26	VEL-	負方向変化率アラーム				入力信号の指定時間内の変化量が負方向に変化率アラーム設定値を超える状態です。
27	MHI	出力上限アラーム	UcaRWWWriteMvToOutSub UcaRWWWriteMv	○		出力信号が出力上限値を超えた状態です。実際の出力は出力上限値に制限されます。
28	MLO	出力下限アラーム				出力信号が出力下限値を下まわった状態です。実際の出力は出力下限値に制限されます。
29～30		(ユーザ定義)				ユーザが決定します。
31	ERRC	演算異常				演算に異常が発生したことを示します。
32	CNF	結合状態不良アラーム	UcaRWReadIn UcaRWRead (*1) UcaCtrlHandler UcaRWReadbackMv UcaRWWWriteCpvToOutSub UcaRWReadbackCpv UcaRWWWrite (*1) UcaRWReadback (*1)	○ ○ ○ ○	○ ○ ○ ○	入出力結合先のブロックがO/Sモードの状態です。保守のため一時的にサービスオフとした制御系で、まだ動作中の機能ブロックを明示するためのアラームです。通常は、IOP/OOPが同時に発生します。

*1：オプション指定時にのみアラームを検出します。

■ アラーム検出の分類

ユーザカスタムブロックにおけるアラームの検出は、次の3種類に大別することができます。

● ユーザカスタムアルゴリズム作成用ライブラリが検出

ユーザカスタムアルゴリズム作成用ライブラリが、アラームを検出します。たとえば、連続制御形カスタムブロックでユーザカスタムアルゴリズム作成用ライブラリ UcaRWSetPv によりデータアイテム PV にデータ設定をすると、UcaRWReadPv の内部で、HI、LO、HH、LL および VEL+、VEL- アラームが検出されます。

参照 アラームの詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「5. アラーム処理—FCS」](#)

● ユーザプログラムがユーザ定義のアラームを検出

ユーザが作成するプログラムで、ユーザ定義アラームをユーザカスタムアルゴリズム作成用ライブラリ UcaSetAlrm により発生し、UcaClearAlrm を復帰します。発生・復帰するアラームは、ERRC（演算異常）か、または（HI や LO などのデフォルトの標準ブロックと同じアラームではなく）ユーザ自身が定義したアラームです。

● ユーザプログラムがデフォルトに含まれるアラームを、標準ブロックと別の意味で検出

ユーザが作成するプログラムで、デフォルトのアラームをユーザカスタムアルゴリズム作成用ライブラリ UcaSetAlrm により発生し、UcaClearAlrm で復帰します。発生・復帰するアラームは、HI や LO などのデフォルトのアラームです。ユーザはユーザカスタムアルゴリズム作成用ライブラリに頼らず、ユーザプログラムが独自の判断でアラームの発生・復帰します。つまり、ユーザカスタムアルゴリズム作成用ライブラリを使わないで、ユーザ自身が処理プログラムを記述する場合にはユーザプログラムでアラームを検出します。また、デフォルトのアラームの意味をユーザ自身が再定義することが可能です。つまり、デフォルトのアラームに PID 調節ブロックなどの標準ブロックとユーザカスタムブロックで異なる意味を持たせることもできますが、アラームの意味を変更しないことを推奨します。

■ ユーザ定義アラームの定義

ユーザ定義アラームを定義する手順を説明します。ユーザ定義のアラームは空いているビット番号、つまりビット番号 15、16、19、20、23、24、29、30 に定義します。ここでは USER16 のビット 19 に「UHIG」、ビット 20 に「ULOW」を入力し、[ファイル] – [ファイルの上書き] で定義内容を書き込みます。

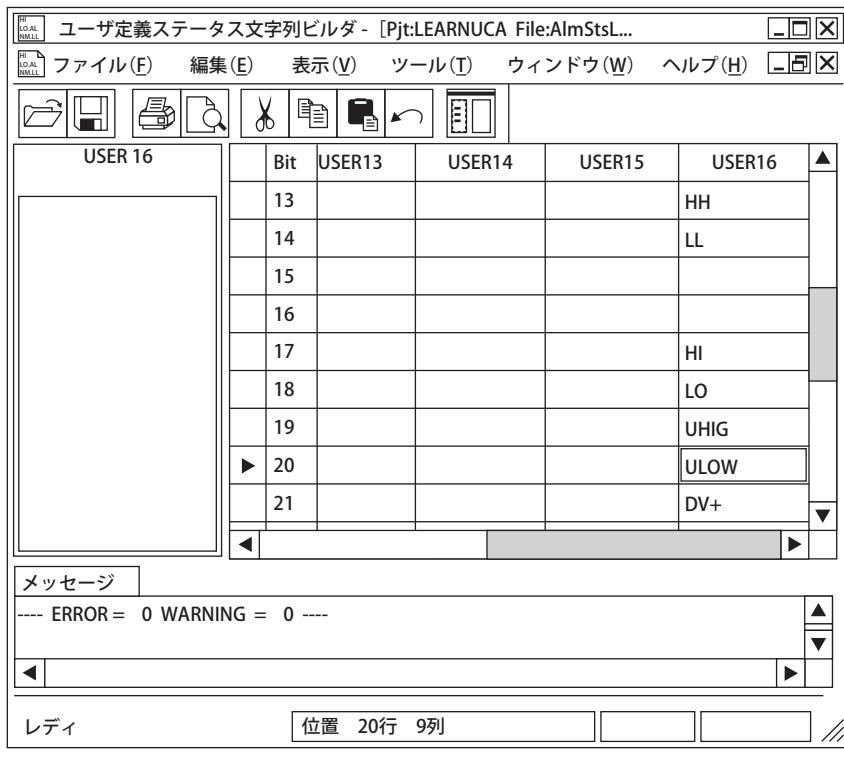
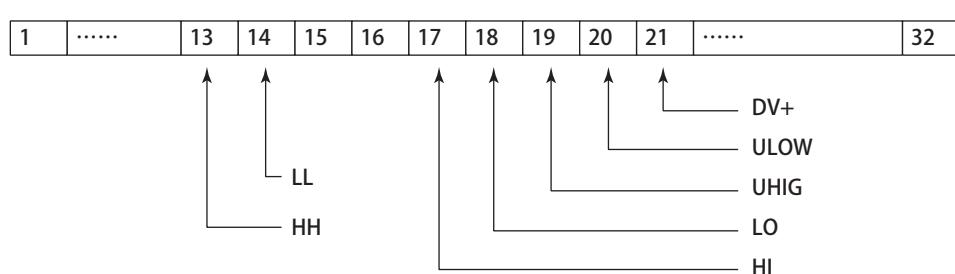


図 ユーザ定義ステータス文字列ビルダ

システムビューの [プロジェクト] – [無効素子の解決] でアラームステータス定義の変更を APCS に反映してください。

ビット 19 に UHIG、ビット 20 に ULOW を定義した結果を以下に示します。



040312J.ai

図 ユーザ定義アラームの定義

ユーザカスタムブロックのアラームステータス（データアイテム ALRM）は、内部的には 32 ビットの整数です。ビット番号 19 に UHIG を定義することは、ALRM の 19 ビットを 0 から 1 にすると UHIG アラームが発生し、ALRM の 19 ビットを 1 から 0 にすると UHIG アラームが復帰することを意味します。アラームステータスの発生はユーザカスタムアルゴリズム作成用ライブラリ UcaAlrmSet()、復帰は UcsAlrmClear() で行います。

ユーザカスタムブロックには CSTM-C(連続制御形ユーザカスタムブロック) と CSTM-A(汎用演算形ユーザカスタムブロック) の 2 種類ありますが、両方とも USER16 の定義を使用します。つまり、この例では CENTUM VP プロジェクト内のすべてのユーザカスタムブロックは、デフォルトに加え UHIG と ULOW のアラームステータスを持つことができます。

- アラームステータスは、CENTUM VP プロジェクトの COMMON の AlmStsLabel に定義します。
- HI、LO や HH、LL などデフォルトのアラームとビット番号の対応を変更することはできません。
- ユーザカスタムブロックのアラームステータスは USER16 に定義します。USER16 以外に定義することはできません。
- アラームステータスは 8 文字(8 バイト)以下の英字で始まる半角英数字、または_ (アンダースコア)、+ (プラス)、- (マイナス) です。英字は大文字で指定します。計器図に表示されるのは先頭 4 文字のみです。チューニングウィンドウには 8 文字すべて表示されます。
- USER16 の中に同じ文字列を複数定義することはできません。
- 他の文字列テーブルに存在する文字列を USER16 に定義する場合は、同じビット番号にのみ指定可能です。たとえば、USER10 のビット 10 に「ALARM01」という文字が定義されていれば、USER16 のビット 10 にのみ「ALARM01」を定義できます。ビット 10 以外に「ALARM01」を定義することはできません。
- アラームステータスのビット番号が小さいほどが優先度が高くなります。複数のアラームが発生(複数のビットが 1 になった)した場合、CENTUM VP のプログラム言語 SEBOL によるデータアイテム ALRM の取得や操作監視機能の計器図の表示における ALRM の値は、もっとも優先度の高いアラームステータスとなります。ALRM は内部的には 32 ビットの整数ですが、SEBOL や計器図の表示では文字列として扱われます。

HI、LO や HH、LL などデフォルトのアラームとビット番号の対応を変更することはできませんが、ユーザ定義のアラームのビット番号を変更すること(たとえば UHIG のビット番号を 19 から 29 にする)は可能です。ただし、エンジニアリングの最初によく検討し、後で変更が発生しないようにしてください。

参照 ユーザ定義のアラームとビット番号の対応の変更の詳細については、以下を参照してください。
「[4.3.3 ユーザ定義インクルードファイル usrstatus.h](#)」

4.3.3 ユーザ定義インクルードファイルusrstatus.h

ブロックステータスとアラームステータスの定義を、ユーザカスタムアルゴリズムのCプログラムから参照する手順を説明します。

最初に、以下の内容を定義したユーザ定義のインクルードファイルを作成します。

サンプルとして、<ドライブ名>\UcaWork\include\usrstatus.h に以下と同じ内容を定義したインクルードファイルが用意されています。

```
.....
/* ブロックステータス */
#define USR_BSTS_RUN UCAMASK_BSTS_05 /* RUN */
#define USR_BSTS_ROLD UCAMASK_BSTS_10 /* HOLD */

/* アラームステータス */
#define USR_ALRM_UHIG UCAMASK_ALRM_19 /* UHIG */
#define USR_ALRM_ULOW UCAMASK_ALRM_20 /* ULOW */
.....
```

ラベル USR_BSTS_RUN を定義している、次の #define 行の意味を説明します。

```
#define USR_BSTS_RUN UCAMASK_BSTS_05 /* RUN */
```

これは、USR_BSTS_RUN というラベルが BSTS のビット番号 5 に対応していることを示します。UCAMASK_BSTS_05 は、システム定義インクルードファイル libucadef.h に定義されています。

参照 システム定義インクルードファイルの詳細については、以下を参照してください。

[「2.2 システム定義インクルードファイル」](#)

```
.....
/* ブロックステータス */
#define UCAMASK_BSTS_01 0x80000000 /* BSTS01 */
#define UCAMASK_BSTS_02 0x40000000 /* BSTS02 */
#define UCAMASK_BSTS_03 0x20000000 /* BSTS03 */
#define UCAMASK_BSTS_04 0x10000000 /* BSTS04 */
#define UCAMASK_BSTS_05 0x08000000 /* BSTS05 */
#define UCAMASK_BSTS_06 0x04000000 /* BSTS06 */
#define UCAMASK_BSTS_07 0x02000000 /* BSTS07 */
.....
```

UCAMASK_BSTS_05 は 0x08000000、つまりビット 5 が 1 のマスクパターンです。

次に、ラベル USR_ALRM_UHIG を定義している #define 行の意味を説明します。

```
#define USR_ALRM_UHIG UCAMASK_ALRM_19 /* UHIG */
```

これは、USR_ALRM_UHIG というラベルが ALRM のビット番号 19 に対応していることを示します。UCAMASK_ALRM_19 は、システム定義インクルードファイル libucadef.h に定義されています。

参照 システム定義インクルードファイルの詳細については、以下を参照してください。
[「2.2 システム定義インクルードファイル」](#)

```
.....
/* アラームステータス */
.....
#define UCAMASK_ALRM_HI      0x00008000 /* HI */
#define UCAMASK_ALRM_LO      0x00004000 /* LO */
#define UCAMASK_ALRM_19      0x00002000 /* ALRM19 */
#define UCAMASK_ALRM_20      0x00001000 /* ALRM20 */
#define UCAMASK_ALRM_DVP     0x00000800 /* DV+ */
#define UCAMASK_ALRM_DVM     0x00000400 /* DV- */
.....
```

UCAMASK_ALRM_19 は 0x00002000、つまりビット 19 が 1 のマスクパターンです。libucadef.h には、UCAMASK_ALRM_HI、UCAMASK_ALRM_LO などデフォルトのアラームのラベルも定義されています。

ユーザ定義インクルードファイルは、<ドライブ名>\UcaWork\include に配置します。1章の準備作業で VC++ のインクルードファイルのパスに <ドライブ名>:\UcaWork\include を設定しました。

ユーザ定義ブロックステータスまたはユーザ定義アラームステータスのビット位置を変更した場合は、usrstatus.h を変更し、usrstatus.h を参照しているユーザカスタムアルゴリズムをすべてリビルドします。そして、ユーザカスタムアルゴリズムをシステムビューにより登録し直します。

4.3.4 ブロックステータスの操作

ユーザ定義インクルードファイルusrstatus.hを使ってブロックステータスRUNとHOLDを操作するプログラムを動かしてみます。

サンプルソリューション_SMPL_BSTS をリビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは、1章の準備作業により以下の作業フォルダにコピーされています。

<ドライブ名>:\UcaWork\UcaSamples\SMPL_BSTS

Visual Studio を起動し、_SMPL_BSTS\SMPL_BSTS.slnを開きます。[ビルド] メニューの [リビルド] により、_SMPL_BSTS の Release 版をリビルドします。ビルド構成には、Release 版と Debug 版がありますが、ユーザカスタムブロックの開発は最初から最後まで Release 版のみを使用します。リビルドが完了したら、システムビューでユーザカスタムアルゴリズムを登録します。

次に、空いている制御ドローイングにサンプルとして用意されている制御ドローイングをインポートします（ここでは、DR0040 にインポートします）。サンプルの制御ドローイングを定義したテキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> \UcaEnv\Sample\LearnUca\drawings\BSTS.txt

FCS0121 (APCS) の制御ドローイングの DR0040 を指定して制御ドローイングビルダを起動します。[ファイル] メニューの [外部ファイル] – [インポート] を指定します。インポートダイアログで上記の BSTS.txt を指定し [開く] ボタンをクリックすると、以下の制御ドローイングが取り込まれます。[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

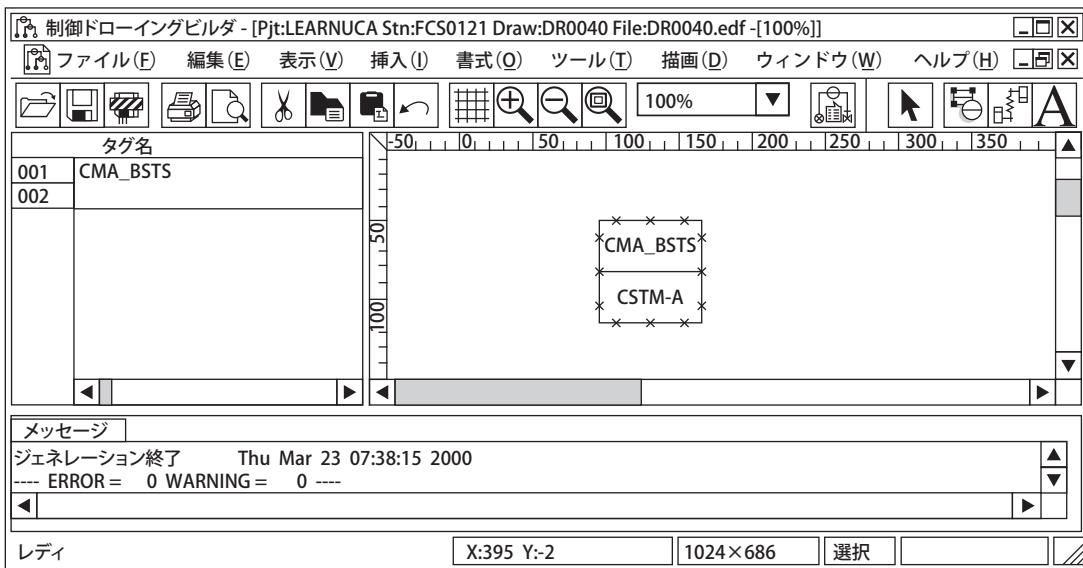


図 制御ドローイングのインポート

FCS0121 (APCS) を指定してテスト機能を起動し、操作監視機能で汎用演算形ユーザカスタムブロック CMA_BSTS のチューニングウィンドウを表示します。CMA_BSTS のプロックステータス (BSTS) は HOLD になっています。

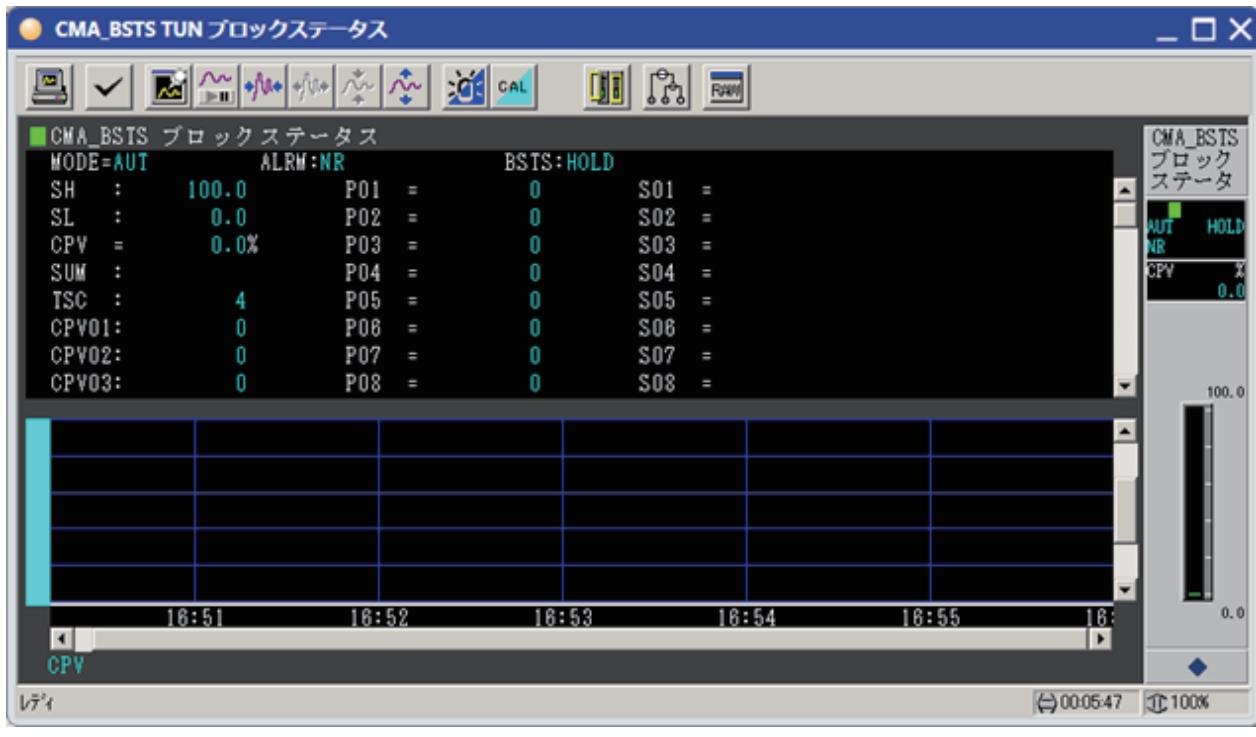
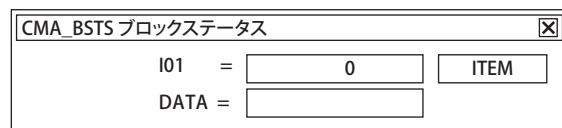


図 CMA_BSTSのチューニングウィンドウ

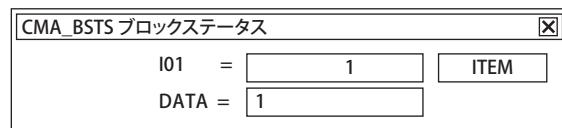
データアイテム I01 は 0 になっています。



040316J.ai

図 CMA_BSTSのブロックステータス

データアイテム I01 を 0 から 1 に変更します。



040317J.ai

図 CMA_BSTSのブロックステータスの変更

データアイテム I01 を 1 にすると、CMA_BSTS のブロックステータスは HOLD から RUN に変化します。そしてデータアイテム P01 の値が、実効スキャン周期（デフォルトの 4 秒です）ごとに 1 ずつ増加します。以下は BSTS が RUN になり、P01 は 7 まで増加したところです。

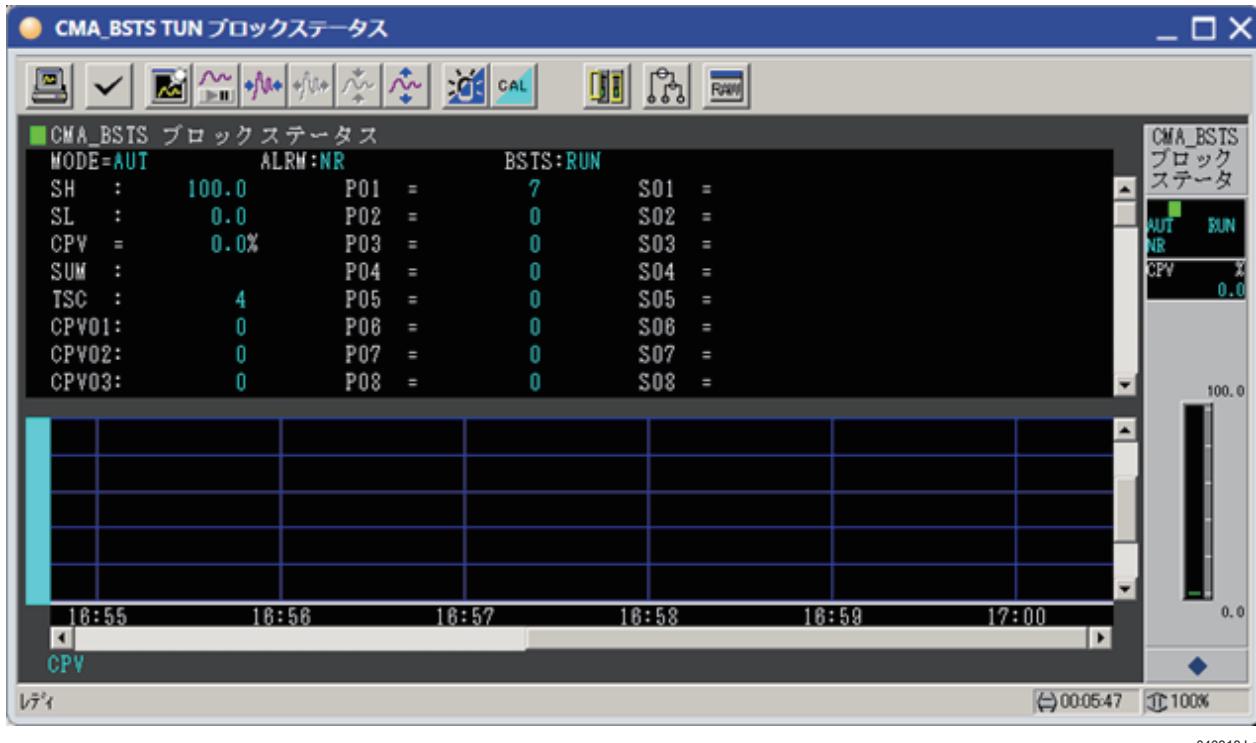
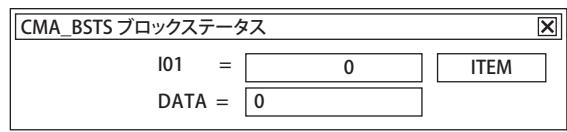


図 CMA_BSTSのブロックステータスの変化

データアイテム I01 を 1 から 0 に戻します。



040319J.ai

図 CMA_BSTSのブロックステータス

ブロックステータス (BSTS) は HOLD になり、データアイテム P01 の増加が停止します。何回か I01 に 1 と 0 を設定して、ブロックステータスと P01 の変化を確認してください。

■ UcaBstsSetExclusive関数でブロックステータスを操作する

ユーザカスタムアルゴリズムのCプログラムについて説明します。ソリューション _SMPL_BSTS のbsts.c をusrstatus.h という文字で検索し、次の部分を見つけてください。

```
.....
/*****
 * ユーザ定義インクルードファイル
 */
#include <usrstatus.h> /* ユーザ定義ブロックステータス・アラームステータス */

/*
* <<FNH>>*****
*
* Function name:      UcaBlockInit
* Return value:        SUCCEED      正常終了
*                      UCAERR_NOPROC 处理なし
*                      UCAERR_STOPME 处理続行不能
*
* description: 機能ブロック初期化処理
*
* >>HNF<<*****
*/
{
    I32 zero;
    I32 rtnCode;

    /* I01を0に初期化 */
    zero = 0;
    rtnCode = UcaDataStoreIn(bc, &zero, 1, 1, NOOPTION);

    /* ブロックステータスを HOLD に初期化 */
    rtnCode = UcaBstsSetExclusive(bc, USR_BSTS_HOLD);

    return SUCCEED;
}
```

このプログラムでは、usrstatus.h を #include 行で取り込んでいます。これで、usrstatus.h で #define 定義してある USR_BSTS_HOLD と USR_BSTS_RUN のラベルを使用してプログラミングすることができます。

参照 usrstatus.h の詳細については、以下を参照してください。

「4.3.3 ユーザ定義インクルードファイル usrstatus.h」

機能ブロック初期化処理 UcaBlockInit は、データアイテム I01 を 0 にし、ブロックステータスを HOLD に初期化しています。このユーザカスタムアルゴリズムは、一度に 1 つのブロックステータスのみ成立（対応するビットが 1）し、他のブロックステータスは不成立（対応するビットが 0）にしています。このようにブロックステータスを排他的に使用する場合には、UcaBstsSetExclusive を使用します。ブロックモード HOLD に対応するビットのみ 1 にするのは以下の部分です。

```
...  
/* ブロックステータスを HOLD に初期化 */  
rtnCode = UcaBstsSetExclusive(bc, USR_BSTS_HOLD);  
...
```

UcaBstsSetExclusive は USR_BSTS_HOLD のビットに 1 を設定し、USR_BSTS_HOLD 以外のビットには 0 を設定します。つまり、「UcaBstsSetExclusive(bc, USR_BSTS_HOLD);」の行を実行すると、データステータスは以下のように USR_BSTS_HOLD のビットのみ 1 となります。

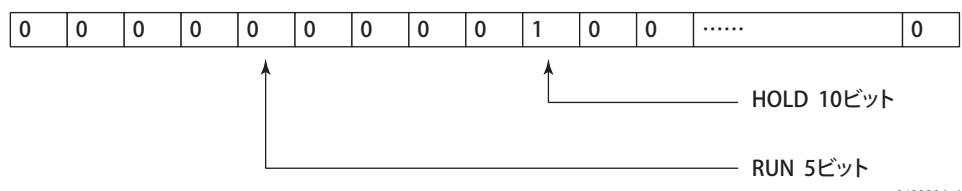


図 UcaBstsSetExclusive によりUSR_BSTS_HOLDの第10ビットのみ1にした状態

040320J.ai

次に、機能ブロック定周期処理について説明します。UcaBlockPeriodical という文字で検索し、次の部分を見つけてください。

```
/*
*<<FNH>>*****
*
* Function name: UcaBlockPeriodical
* Return value:  SUCCEED      正常終了
*                 UCAERR_NOPROC   処理なし
*                 UCAERR_STOPME    処理続行不能
*
* description:  機能ブロック定周期処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc      /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 i01;
    F64S p01;
    I32 rtnCode;

    /* i01 の値を取得 */
    rtnCode = UcaDataGetIn(bc, &i01, 1, 1, NOOPTION);

    if (i01 == 0) {
        /* ブロックステータス HOLD を設定 */
        rtnCode = UcaBstsSetExclusive(bc, USR_BSTS_HOLD);           ←———— HOLDを設定
    } else {
        /* ブロックステータス RUN を設定 */
        rtnCode = UcaBstsSetExclusive(bc, USR_BSTS_RUN);           ←———— RUNを設定
    }

    /* データアイテムP01を1増加 */
    rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);
    p01.value += 1;
    rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);
}

return SUCCEED;
}
```

機能ブロック初期化処理では、データアイテム I01 が 0 のときはブロックスステータス HOLD の処理、I01 が 0 以外のときはブロックスステータス RUN の処理をしています。

- I01 が 0 のときは BSTS に HOLD を設定し、あとは何もしません。
- I01 が 0 以外のときは BSTS に RUN を設定し、データアイテム P01 を 1 増加しています。

このプログラムでは、データアイテム I01 の値が変わらない限り実効スキャン周期ごとにブロックスステータスが HOLD なら HOLD、RUN なら RUN を上書きすることになりますが、ブロックスステータスに関する処理を単純にできるのでこのようにしてあります。

ブロックスステータスの操作には、これまで説明してきた UcaBstsSetExclusive の他に、UcaBstsSet と UcaBstsClear の 2 つの関数があります。これらの関数の動作を以下に示します。

表 ブロックスステータスの操作をする関数

関数名	機能	用途
UcaBstsSetExclusive	引数に指定されたブロックスステータスをセット(1に)します。指定された以外のブロックスステータスをクリア(0に)します。	ブロックスステータスを排他的に使う場合に使用します。
UcaBstsSet	引数に指定されたブロックスステータスをセット(1に)します。指定された以外のブロックスステータスは変化しません。	ブロックスステータスを各ビットごとに管理する場合に使用します。
UcaBstsClear	引数に指定されたブロックスステータスをクリア(0に)します。指定された以外のブロックスステータスは変化しません。	

同時にはひとつのブロックスステータスのみ成立するように、ブロックスステータスを使用する場合には UcaBstsSetExclusive を使用します。この節の例でも、UcaBstsSetExclusive を使用しています。

ブロックスステータスの各ビットを独立して管理する場合には、UcaBstsSet と UcaBstsClear を使用します。これらの関数は引数に指定されたブロックスステータスのビットのみを操作します。このような使い方は高度なブロックスステータスの管理方法となりプログラムが難しくなります。特に理由のない限り、ブロックスステータスは UcaBstsSetExclusive を使った排他的な管理としてください。

4.4 ブロック形の決定

ユーザカスタムブロックには、次の2種類があります。

表 ユーザカスタムブロックのブロック形

ブロック形名	名称	説明
CSTM-C	連続制御形ユーザカスタムブロック	連続制御を実現するためのユーザカスタムブロックです。主に、制御演算結果を FCS の連続制御ブロックの設定値として出力する、セットポイント制御に使用します。
CSTM-A	汎用演算形ユーザカスタムブロック	汎用演算を実現するためのユーザカスタムブロックです。主に、pct 入力信号の特殊な変換処理などに使用します。

連続制御形ユーザカスタムブロックは PID 調節ブロック (PID) のデータアイテムと入出力端子に加えて、32 入力端子、16 出力端子、ユーザカスタムブロック固有データアイテムをもちます。また、アルゴリズムはユーザが C プログラムで記述するユーザカスタムアルゴリズムで定義します。連続制御形ユーザカスタムブロックの構成を以下に示します。

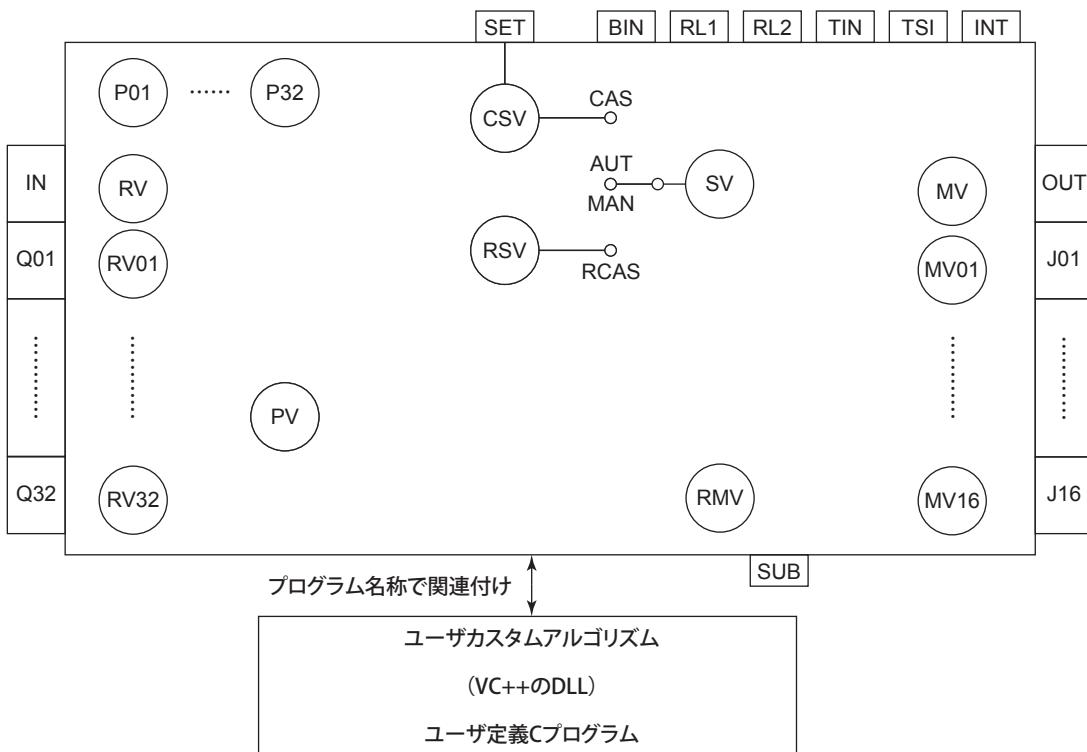
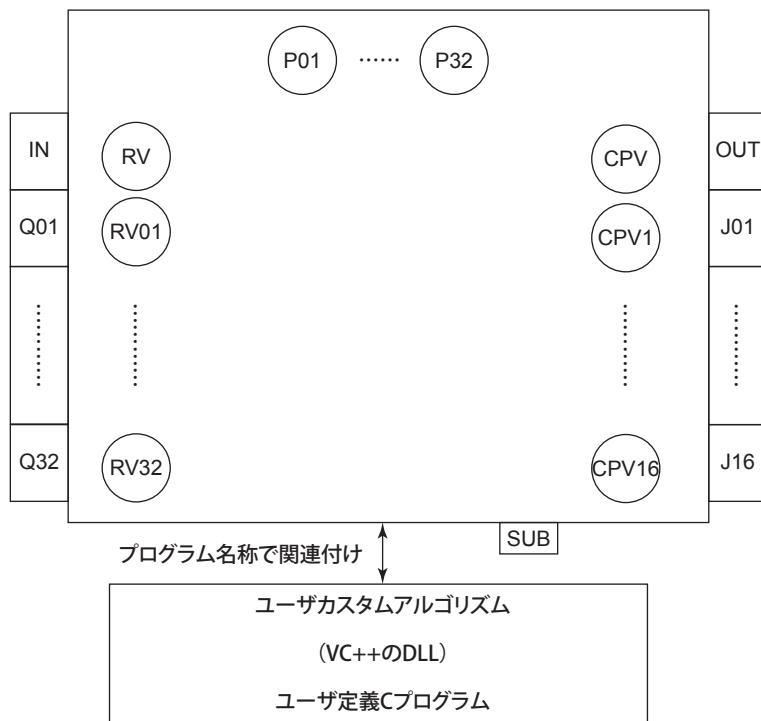


図 連続制御形ユーザカスタムブロックの構成

040402J.ai

汎用演算形ユーザカスタムブロックは汎用演算ブロック (CALCU) のデータアイテムと入出力端子に加えて、32 入力端子、16 出力端子、ユーザカスタムブロック固有データアイテムをもちます。また、アルゴリズムはユーザが C プログラムで記述するユーザカスタムアルゴリズムで定義します。汎用演算形ユーザカスタムブロックの構成を以下に示します。



040403J.ai

図 汎用演算形ユーザカスタムブロックの構成

1台のAPCSでいくつのユーザカスタムブロックを実行できるかは、データベースタイプで決まります。たとえば、APCSの「汎用」データベースタイプでは最大500個のユーザカスタムブロックを実行することができます。つまり、APCSの汎用データベースタイプではCSTM-CとCSTM-Aの合計で最大500個を実行することができます。

参照 APCSのデータベースタイプの詳細については、以下を参照してください。

[APCS \(IM 33J15U10-01JA\)](#)

エンジニアリングの最初に、APCSでCSTM-CとCSTM-Aをそれぞれいくつずつ使用するかを、CSTM-CとCSTM-Aの合計がデータベースタイプのユーザカスタムブロック最大個数の範囲内になるように決めてください。

4.5 ビルダ定義項目の決定

ユーザカスタムブロックのビルダ定義項目と、ユーザカスタムアルゴリズムの関係について説明します。ビルダ定義項目の指定がユーザカスタムブロックで有効になるか否は、次の3種類に分類できます。

● システムが処理

指定内容をシステムが処理するビルダ定義項目です。ユーザカスタムアルゴリズムの内容によらず、ビルダ定義項目の指定が常に有効になります。タグコメント、機能制約レベルなどが該当します。

● ユーザカスタムアルゴリズム作成用ライブラリが内部で処理

指定内容をユーザカスタムアルゴリズム作成用ライブラリが処理するビルダ定義項目です。たとえば、ビルダ定義項目「入力信号変換タイプ」の指定値は「無変換」または「通信入力」ですが、この指定はユーザカスタムアルゴリズム作成用ライブラリ UcaRWReadin が処理します。つまり、IN 端子のデータを UcaRWReadin で取得しているユーザカスタムアルゴリズムでは、ビルダ定義項目「入力信号変換タイプ」の指定が有効になります。この分類は、ユーザカスタムアルゴリズムで、該当のビルダ定義項目を処理するユーザカスタムアルゴリズム作成用ライブラリ関数を使用しているか否かにより、ビルダ定義項目の有効・無効が決まります。

● ユーザカスタムアルゴリズム作成用ライブラリで指定値を取り出しユーザが処理

指定内容をユーザカスタムアルゴリズム作成用ライブラリで取り出し、指定内容に従いユーザプログラムが処理を分けます。

ユーザは有効にしたいビルダ定義項目を決定し、ユーザカスタムアルゴリズムの中でユーザカスタムアルゴリズム作成ライブラリを使用し、ビルダ定義項目の指定が有効になるようにプログラムを作成します。

それぞれのビルダ定義項目が、前述3つの分類のどれに対応するかを以下に示します。

参照 各ビルダ定義項目の意味と指定値の詳細については、以下を参照してください。
APCS ユーザカスタムブロック (IM 33J15U20-01JA)

表 ビルダ定義項目の分類 (1/5)

定義項目	指定の有無		分類(*1)		
	CSTM-C	CSTM-A	システム	ライブラリ	ユーザ
基本設定	タグ名	○	○	○	UcaDataGetTagName
	タグコメント	○	○	○	
	機能制約レベル	○	○	○	
	実効スキャン周期	○	○	○	UcaDataGetTsc
	スキャン位相	○	○	○	
	プログラム名称	○	○	○	UcaDataGetPrgn
	起動タイミング	○	○	○	
	開閉マーク	○		○	
	入力信号変換タイプ	○	○		UcaRWReadIn
	データ変換ゲイン	○	○		
	データ変換/バイアス	○	○		
	入力上限検出設定値	○	○		
	入力下限検出設定値	○	○		
	積算時間単位	○	○		UcaRWSetPv
	積算低入力カット値	○	○		
	メジャートラッキング MAN 時	○			UcaCtrlHandler
	メジャートラッキング AUT かつ CND 時	○			
	メジャートラッキング CAS かつ CND 時	○			
	出力信号変換タイプ	○			UcaRWWWriteMvToOutSub UcaRWWWriteMv
	データ変換ゲイン	○			
	データ変換/バイアス	○			

*1： システム：ビルダ定義項目をシステムが処理します（前述の「●システムが処理」）。

ライブラリ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリが処理します（前述の「●ユーザカスタムアルゴリズム作成用ライブラリが内部で処理」）。

ユーザ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリで取り出し、ユーザプログラムで利用することができます（前述の「●ユーザカスタムアルゴリズム作成用ライブラリで指定値を取り出しユーザが処理」）。

表 ビルダ定義項目の分類 (2/5)

定義項目	指定の有無		分類(*1)		
	CSTM-C	CSTM-A	システム	ライブラリ	ユーザ
機能 プロック 情報	タグマークの種類	○	○	○	
	ステータス変更メッセージバイパス	○	○	○	
	上位ウィンドウ名	○	○	○	
	ヘルプメッセージ番号	○	○	○	
	MV 計器図表示	○		○	
	CAS マーク	○		○	
	CAS マークの種類	○		○	
	CMP マーク	○		○	
	スケールの逆表示	○	○	○	
	MV の逆表示	○		○	
	置針	○		○	
	目盛り分割数	○	○	○	
入力 処理	上位設備名	○	○	○	
	測定値レンジ上限値	○	○	UcaRWReadIn	UcaDataGetPh
	測定値レンジ下限値	○	○		UcaDataGetPl
	工業単位記号	○	○	○	
	入力フィルタ	○	○	UcaRWSetPv(CSTM-C) UcaRWSetCpv(CSTM-A)	
PV/CPV 振り切り	PV/CPV 振り切り	○	○		

*1 : システム：ビルダ定義項目をシステムが処理します（前述の「●システムが処理」）。

ライブラリ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリが処理します（前述の「●ユーザカスタムアルゴリズム作成用ライブラリが内部で処理」）。

ユーザ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリで取り出し、ユーザプログラムで利用することができます（前述の「●ユーザカスタムアルゴリズム作成用ライブラリで指定値を取り出しユーザが処理」）。

表 ビルダ定義項目の分類 (3/5)

定義項目	指定の有無			分類(*1)	
	CSTM-C	CSTM-A	システム	ライブラリ	ユーザ
アラーム処理					
アラーム処理レベル	○	○	○		
入力オーブンアラーム	○	○		UcaRWReadIn	
演算入力値異常検出	○	○			
上下限アラーム 入力上上限／入力下下限アラーム	○				
上下限アラーム入力上下限アラーム	○				
上下限アラームヒステリシス	○			UcaRWSetPv	
入力変化率アラーム	○				
サンプリング数	○				
サンプリング間隔	○				
ヒステリシス	○				
偏差アラーム	○				
DV チェックフィルタゲイン	○			UcaCtrlHandler UcaDataCheckDv	
DV チェックフィルタ時定数	○				
ヒステリシス	○				
出力オーブンアラーム	○	○		UcaRWWwritePvToJnSub (*2) UcaRWWwrite (*2) UcaRWReadback (*2)	
	○			UcaCtrlHandler UcaRWReadbackMv	
		○		UcaRWWwriteCpvToOutSub (*2) UcaRWWwriteCpv (*2) UcaRWReadbackCpv (*2)	
出力上下限アラーム	○			UcaRWWwriteMvToOutSub UcaRWWwriteMv	
ヒステリシス	○				
結合状態不良アラーム	○	○		UcaRWReadIn UcaRWRead UcaRWWwritePvToJnSub (*2) UcaRWWrite (*2) UcaRWReadback (*2)	
	○			UcaCtrlHandler UcaRWReadbackMv	
		○		UcaRWWwriteCpvToOutSub UcaRWWwriteCpv UcaRWReadbackCpv	

*1: システム: ビルダ定義項目をシステムが処理します(前述の「●システムが処理」)。

ライブラリ:ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリが処理します（前述の「●ユーザカスタムアルゴリズム作成用ライブラリが内部で処理」）。

ユーザ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリで取り出し、ユーザプログラムで利用することができます（前述の「●ユーザカスタムアルゴリズム作成用ライブラリで指定値を取り出しうるユーザが処理」）。

*2：オプション指定時のみ処理

表 ビルダ定義項目の分類 (4/5)

定義項目	指定の有無			分類(*1)	
	CSTM-C	CSTM-A	システム	ライブラリ	ユーザ
制御演算処理	PID 制御アルゴリズム	○		UcaCtrlPid	
	制御周期	○		UcaCtrlPidTiming	
	制御動作方向 (*2)	○			UcaConfigDirection
	入出力補償 (*2)	○		UcaCtrlCompensation	UcaConfigCompensation UcaCtrlCompensation_p
	非線形ゲイン	○		UcaCtrlPid	UcaConfigNonLinearGain
	ギャップゲイン	○			UcaCtrlGetGapGainCoef_p
	不感帯動作	○			UcaConfigDeadband
	ヒステリシス	○		UcaCtrlDeadband	
	AUT フォールバック	○		UcaCtrlHandler	
	コンピュータバックアップモード	○			
出力処理	出力変化率リミッタ	○		UcaRWWWriteMvToOutSub UcaRWWWriteMv	
	MAN 時 出力変化率リミッタバイパス	○			
	出力値トラッキング	○		UcaCtrlHandler UcaRWReadbackMv UcaRWWWriteMvToOutSub UcaRWWWriteMv	
		○		UcaRWWWriteCpvToOutSub UcaRWReadbackCpv UcaRWWWriteCpv UcaRWCheckOutputCondition	
	出力信号変換タイプ		○	UcaRWWWriteCpvToOutSub UcaRWReadbackCpv UcaRWWWriteCpv	
	データ変換ゲイン		○		
	データ変換バイアス		○		
	補助出力 出力値	○	○	UcaRWWWriteCpvToOutSub UcaRWWWritePvToJnSub UcaRWWWriteSub	
	補助出力 出力動作	○	○		
	MV データの表示形式	○		○	
	操作出力レンジ上限値	○		○	UcaRWWWriteMvToOutSub UcaRWWWriteMv
	操作出力レンジ下限値	○		○	
	操作出力工業単位記号	○		○	
	クランプ時出力方向制限	○		○	UcaRWWWriteMvToOutSub UcaRWWWriteMv
	制御／演算出力動作 (*2)	○		○	

*1： システム：ビルダ定義項目をシステムが処理します（前述の「●システムが処理」）。

ライブラリ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリが処理します（前述の「●ユーザカスタムアルゴリズム作成用ライブラリが内部で処理」）。

ユーザ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリで取り出し、ユーザプログラムで利用することができます（前述の「●ユーザカスタムアルゴリズム作成用ライブラリで指定値を取り出しユーザが処理」）。

参照 表中の (*2) について、制御動作方向、入出力補償、制御／演算出力動作におけるビルダ定義項目の指定とユーザカスタムアルゴリズムの C プログラムの関係については、以下を参照してください。

「6.3.10 制御演算関数呼び出しの省略」

表 ビルダ定義項目の分類 (5/5)

定義項目	指定の有無		分類(*1)		
	CSTM-C	CSTM-A	システム	ライブラリ	ユーザ
結合情報	測定入力 (IN)	○	○	UcaRWReadIn	
	操作出力 (OUT)	○		UcaCtrlHandler UcaRWReadbackMv UcaRWWriteMvToOutSub UcaRWWriteMv	
			○	UcaRWWriteCpvToOutSub UcaRWReadbackCpv UcaRWWriteCpv	
	設定入力 (SET)	○			
	補助出力 (SUB)	○	○	UcaRWWriteCpvToOutSub UcaRWWritePvToJnSub UcaRWWriteSub	
	リセット信号 1 入力 (RL1)	○		UcaCtrlHandler UcaRWReadRI	
	リセット信号 2 入力 (RL2)	○			
	補償入力 (BIN)	○		UcaCtrlHandler UcaRWReadBin	
	トラッキング信号入力 (TIN)	○		UcaCtrlHandler UcaRWReadTrack	
	トラッキング SW 入力 (TSI)	○			
その他	インタロック SW 入力 (INT)	○		UcaCtrlHandler UcaRWReadInt	
	第 n 演算入力 (Q01 ~ Q32)	○	○	UcaRWRead UcaRWReadString UcaRWSeqCond	
	第 n 演算出力 (J01 ~ J16)	○	○	UcaRWWritePvToJnSub UcaRWReadback UcaRWWrite UcaRWWriteString UcaRWSeqOprt	
	固定定数	○	○	○	
	汎用指定項目 1 ~ 8	○	○		UcaConfigGeneral

*1：システム：ビルダ定義項目をシステムが処理します（前述の「●システムが処理」）。

ライブラリ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリが処理します（前述の「●ユーザカスタムアルゴリズム作成用ライブラリが内部で処理」）。

ユーザ：ビルダ定義項目をユーザカスタムアルゴリズム作成用ライブラリで取り出し、ユーザプログラムで利用することができます（前述の「●ユーザカスタムアルゴリズム作成用ライブラリで指定値を取り出しユーザが処理」）。

■ ビルダ定義項目のデフォルトファイル作成

ユーザカスタムブロックのビルダ定義項目のデフォルトは決まっています。システムが規定しているデフォルトを、デフォルトファイル作成機能によりユーザが変更することができます。システムが規定するデフォルトと異なる指定をする機能ブロックを多数作成する場合には、デフォルトファイル作成が便利です。以下は、連続制御形ユーザカスタムブロック (CSTM-C) の機能ブロック詳細ビルダでのデフォルトファイル作成です。

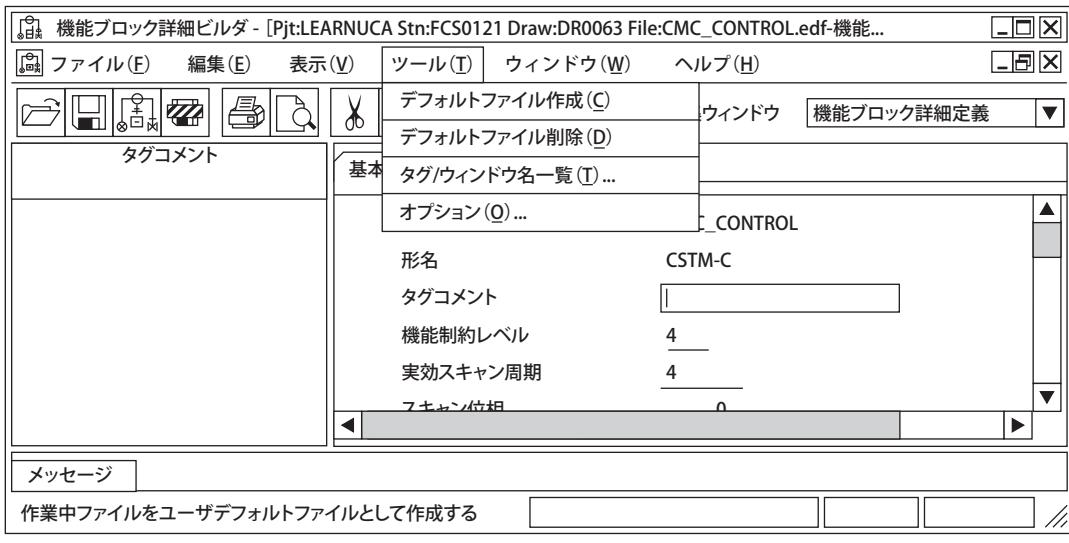


図 デフォルトファイル作成

- ビルダ定義項目の「[デフォルトファイル作成]」は、機能ブロック形ごとに指定します。たとえば、CSTM-C に対してデフォルトファイル作成処理をすると、CSTM-C に対して有効です。
- デフォルトと指定をしてから「[デフォルトファイル作成]」操作をします。「[デフォルトファイル作成]」すると、以後、同じブロック形の機能ブロックを作成すると、「[デフォルトファイル作成]」をしたときのビルダ定義項目の指定がデフォルトとなります。「[デフォルトファイル作成]」以前に定義した（同じブロック形の）機能ブロックには、何も影響はありません。
- デフォルトの設定は、「[デフォルトファイル作成]」を実行したコンピュータ内で有効です。同じ CENTUM VP プロジェクトに対し複数のコンピュータでエンジニアリングする場合には、それぞれのコンピュータで「[デフォルトファイル作成]」をしてください。
- 「[デフォルトファイルの作成]」をすると、該当のコンピュータで任意の CENTUM VP プロジェクトをエンジニアリングする場合に有効です。標準ブロック (PID 調節ブロックなど) に対し「[デフォルトファイルの作成]」をすると、FCS と APCS の両方で有効になります。
- 「[デフォルトファイル削除]」をすると、該当の機能ブロック形のデフォルトはシステムが規定するデフォルトに戻ります。

4.6 ひな形にするユーザカスタムアルゴリズムの作成

APCSで多数のユーザカスタムブロックを稼動する場合には、どのようなユーザカスタムアルゴリズムをいくつずつ作成する必要があるのか、開発の初期に計画を立てます。そして、APCSで稼動するユーザカスタムアルゴリズムをいくつかのパターンに分類してください。その分類ごとにユーザカスタムアルゴリズムを1つずつ作成し、内容を十分に吟味します。そして完成したユーザカスタムアルゴリズムは、同じパターンのユーザカスタムアルゴリズムの「ひな形」にします。

ひな形ができれば、同じ分類のユーザカスタムアルゴリズムはひな形のコピーを修正することで作成することができます。以下はひな形作成の留意点です。

- ・ひな形に不具合がある場合には、そのひな形を元に作成したすべてのユーザカスタムアルゴリズムに「同じ不具合吸収の修正」が必要になります。したがって、開発の初期でひな形に対して十分にテストを行い、不具合をなくすようにしてください。
- ・ひな形を作成するときに、ユーザカスタムアルゴリズムを複数のユーザカスタムブロックで共用するかしないかを検討してください。

参照 ユーザカスタムアルゴリズムの共用の詳細については、以下を参照してください。
「4.7 ユーザカスタムアルゴリズムの共用」

■ ユーザカスタムアルゴリズムのサンプル一覧

本書で説明しているユーザカスタムアルゴリズムのサンプル一覧を以下に示します。

表 ユーザカスタムアルゴリズムのサンプル (1/2)

プログラム名称 (ソリューション名)	説明している章	内容
_SMPL_LETSSTART	1.1	「さあ、始めましょう！」というメッセージを定期周期で出力
_SMPL_SAMPLE1	1.2～1.5	以下処理の簡単なプログラム例 機能ブロック定期周期処理 機能ブロック終了処理 機能ブロック初期化処理 機能ブロックデータ設定時特殊処理
_SMPL_COND	1.6	機能ブロックワンショット起動処理（条件判定）
_SMPL_OPRT	1.6	機能ブロックワンショット起動処理（状態操作）
_SMPL_PVI_NOOUT	3.6.1～3.6.2	指示ブロック (PVI) と同じ動作をするプログラム例 (OUT 端子と SUB 端子への出力はしない) ブロックモード変更指令に対する機能ブロックデータ設定時特殊処理 (ブロックモードを AUT と O/S に限定) 入力上限／下限アラームと入力上上限／下下限アラームの検出
_SMPL_BSTS	4.3.4	ブロックステータスの操作
_SMPL_ALGO_IN	5.1	多入力 CSTM-A のサンプル (*1) 入力端子からデータを入力し演算結果 CPV を作成
_SMPL_ALGO	5.2	多入力多出力 CSTM-A のサンプル 入力端子からデータを入力し演算結果 CPV を作成 出力端子からデータを出力 演算エラーの検出 演算異常 ERRC アラームの発生と復帰 チューニングパラメータより弱い初期値の設定
_SMPL_ALRM	6.1	多入力 CSTM-C のサンプル 入力端子からデータを入力し演算結果 PV を作成 入力上限／下限アラームと入力上上限／下下限アラームの検出 ブロックモードを AUT と O/S に限定 Jnn 出力端子からデータを出力 ユーザ定義アラームの発生と復帰

*1： ひな形については、「5.2 多入力多出力 CSTM-A」を参考にしてください。

表 ユーザカスタムアルゴリズムのサンプル (2/2)

プログラム名称 (ソリューション名)	説明している章	内容
_SMPL_MODE	6.2	CSTM-C のブロックモード遷移 AUT、CAS、TRK などブロックモードに従い動作する CSTM-C
_SMPL_CONTROL	6.3	制御演算関数の使い方 入力補償、出力補償、不感帯動作、制御ホールドなどを処理する関数の使い方 実効スキャン周期ごとに制御演算を実効します
_SMPL_PID	6.4	標準の PID 調節ブロックと同じ動作をする CSTM-C サンプルを改造することにより PID 調節ブロックにユーザ独自の動作を追加できます 制御周期を有効にするプログラムの書き方
_SMPL_CONTROL2	6.6	制御周期を有効にするプログラムの書き方 それ以外は、6.3 節の _SMPL_CONTROL と同じ
_SMPL_ALRM_FTAG	7.2	多入力 CSTM-C のサンプル タグ名を指定して他ステーションデータを入力し測定値 PV を作成 それ以外は、6.1 節の _SMPL_ALRM と同じ
_SMPL_ALGO_TAG	7.3	多入力多出力 CSTM-A のサンプル タグ名を指定して自ステーション内データを入力し演算結果 CPV を作成 それ以外は、5.2 節の _SMPL_ALGO と同じ

サンプルは、ユーザカスタムブロック開発環境パッケージの以下のフォルダにあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥UcaWork¥UcaSamples¥ <サンプルのソリューション>

4.7 ユーザカスタムアルゴリズムの共用

ユーザカスタムアルゴリズムの共用について説明します。複数のユーザカスタムブロックで、同じユーザカスタムアルゴリズムを使用することができます。これは複数のユーザカスタムブロックに対し、機能ブロック詳細ビルダで同じ「プログラム名称」を指定することを意味します。



040701.ai

図 機能ブロック詳細ビルダで [プログラム名] に「_SMPL_PVI_NOOUT」を指定

ユーザカスタムアルゴリズムを共用できれば、ユーザカスタムアルゴリズムのプログラム数を少なくすることができます。また、開発するプログラムが少なければ、不具合の数も少なくなるはずです。

ユーザカスタムアルゴリズムを共用する場合には、データの入力と出力は入出力端子から行うようにプログラミングします。ユーザカスタムブロックは、32個の入力端子と16個の出力端子を持っていますので、この端子数で余裕（機能追加の改造で必要な端子の数が増えるかもしれません）をもってユーザカスタムアルゴリズムを作成できる場合には、入出力端子を使ってプログラミングをし、ユーザカスタムアルゴリズムを共用することを検討してください。

一般にタグ名を使用してプログラムを作成すると、1つのユーザカスタムアルゴリズムは1つのユーザカスタムブロックでのみ使用可能となります。つまり、ユーザカスタムアルゴリズムを複数のユーザカスタムブロックで共用することはできなくなります。タグ名を使用して記述したユーザカスタムアルゴリズムを複数のユーザカスタムブロックのプログラム名に指定することはできますが、同じ機能ブロックに対して複数のユーザカスタムブロックから同じ処理をすることになるので、ユーザのアプリケーションとしては（特別な意図がある場合を除いては）誤りとなります。

● ビルダ定義項目「汎用指定項目1～8」

ユーザカスタムアルゴリズムを共用する場合に、「プログラムの大部分は同じなのだけれど、少しだけ処理が違う動作」が必要になる場合があります。このようなときは、ビルダ定義項目「汎用指定項目1～8」を使用します。「汎用指定項目1～8」の指定値は、UcaConfigGeneralで取り出すことができます。汎用指定項目1の指定値により、ユーザプログラムが処理を分ける例を以下に示します。

```
.....
I32 rtnCode;
I32 configGen1;1 /* 汎用指定項目1 */

.....
rtnCode = UcaConfigGeneral(bc, 1, &configCen1); /* 汎用指定項目1を取り出す */
switch (configGen1) {
    case 0:
        /* 指定値が0の処理（デフォルト）*/
        .....
        break;
    case 1:
        /* 指定値が1の処理 */
        .....
        break;
    case 2:
        /* 指定値が2の処理 */
        .....
        break;
    default:
        /* その他 */
        .....
        break;
}
.....
```

汎用指定項目1～8のデフォルトは0です。したがって、ユーザカスタムアルゴリズムの処理も汎用指定項目の指定が0のときにもっともよく使用する処理をするようにしてください。汎用指定項目は16ビットの整数です。それぞれに-32768～32767の値を指定することができます。

汎用指定項目を使う場合、指定が複雑になりすぎないように注意してください。多数の汎用指定項目を使用し（たとえば、汎用指定項目1～8を全部使うなど）、ユーザカスタムアルゴリズムを無理に共用するのは、不具合の原因になります。ユーザカスタムアルゴリズムのプログラムが複雑になりますし、また、機能ブロック詳細ビルダで汎用指定項目のデータ入力を誤る可能性が大きくなります。

● ユーザカスタムアルゴリズムのオンラインの修正

ユーザカスタムアルゴリズムを共用すると、ユーザカスタムアルゴリズムをオンラインの修正をしている間は該当のユーザカスタムアルゴリズムを使用しているすべてのユーザカスタムブロックのロックモードがO/Sになります。システムはユーザアルゴリズムアルゴリズムのロードが完了すると、ロックモードをO/Sから復帰します。

参照 ユーザカスタムアルゴリズムをオンラインで追加、変更、削除したときのシステムの動作の詳細については、以下を参照してください。

「3.3.1 オンラインの修正に伴う機能ブロック初期化処理／終了処理」

● ユーザカスタムアルゴリズム共用のポイント

- 複数のユーザカスタムブロックでユーザカスタムアルゴリズムを共用することが可能です。
- ユーザカスタムアルゴリズムを共用する場合には、入出力端子からデータを入出力します。
- ユーザカスタムアルゴリズムを共用する場合には、(特別な意図のない限り) プログラム中に「タグ名」を記述しないでください。
- 少しだけ処理の異なるユーザカスタムアルゴリズムを作成したい場合には、ビルダ定義項目「汎用指定項目1～8」を利用します。ただし、汎用指定項目の指定は単純な指定に留めてください。複雑な指定が必要になるような無理なユーザカスタムアルゴリズムの共用はしないでください。
- ユーザカスタムアルゴリズムをオンラインで追加、変更、削除すると、そのユーザカスタムアルゴリズムを使用しているすべてのユーザカスタムブロックのロックモードがO/Sになります。オンラインロードが完了すると、ロックモードはO/Sから復帰します。

■ UcaDataGetTagNameでのタグ名の取得

ユーザカスタムアルゴリズムを複数のユーザカスタムブロックで共用した場合、Visual C++ のデバッグ中に「どのユーザカスタムブロックの処理を実行しているか」をわかるようにしておく方法について説明します。ユーザ定義関数の最初で、UcaDataGetTagname によりタグ名を取得しておきます。機能ブロック定周期処理の先頭で、変数 tagName にタグ名を取得する記述は以下のようになります。

VC++ のデバッグ機能により、UcaDataGetTagname 呼び出しの次の行にブレークポイントを張り一時停止した状態を以下に示します。ウォッチウィンドウ (VC++ の [デバッグ] - [ウィンドウ] - [ウォッチ] - [ウォッチ 1] でウォッチウィンドウが表示されます) のシンボル名にタグ名を格納している変数「tagName」を指定すると、現在処理タイミングを与えられているユーザカスタムブロックのタグ名は CMA_BSTS であることがわかります。

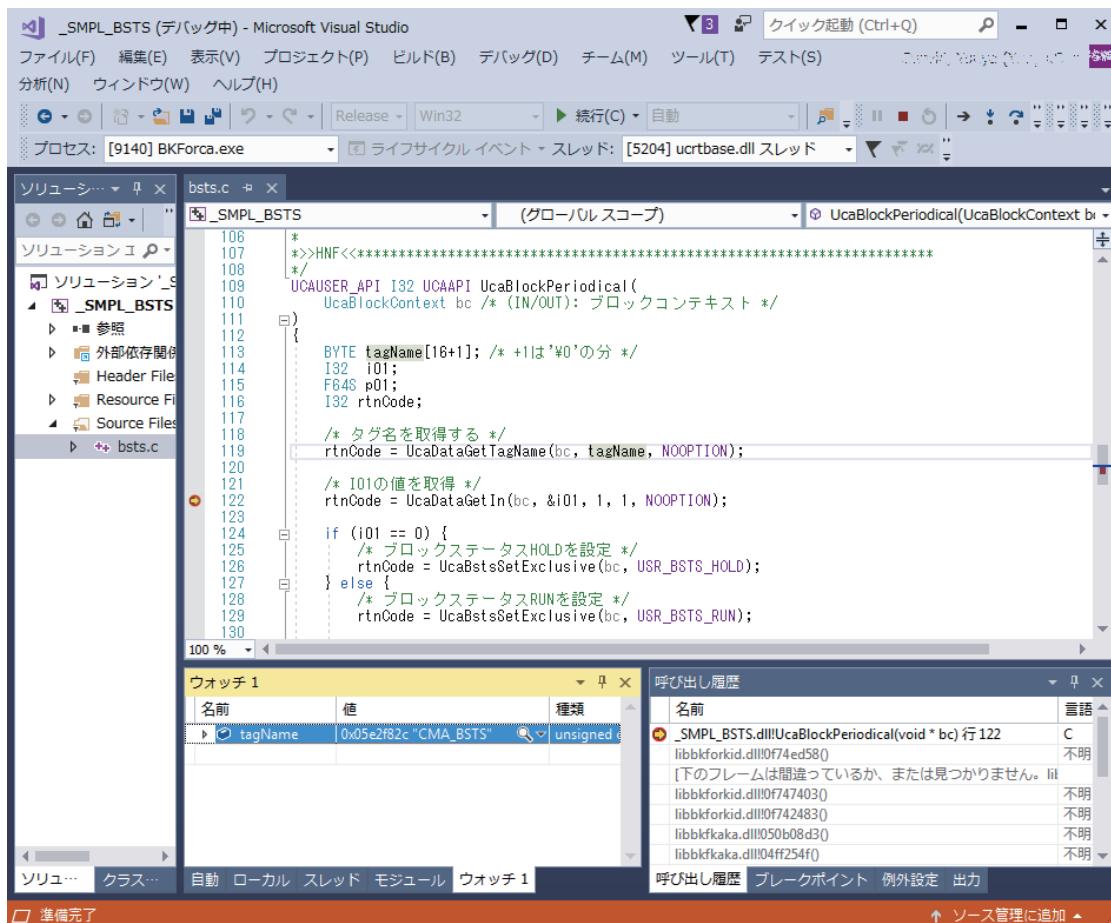


図 一時停止した状態

● 文字列の末尾に“¥0”を追加

ここでC言語の文字列に関する注意事項について説明します。以下はタグ名を取得するプログラムです。

```
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc      /* (IN/OUT): ブロックコンテキスト */
)
{
    BYTE tagName[16+1];    /* +1は'¥0'の分 */   ←———— 必ず1バイト余分に宣言する
    I32 i01;
    I32 rtnCode;

    /* タグ名を取得する */
    rtnCode = UcaDataGetTagName(bc, tagName, NOOPTION);
    .....
}
```

タグ名を格納する変数 tagName は、17 バイト（16 + 1）の文字列配列です。CENTUM VP のタグ名は最大 16 文字ですから 1 バイト余分に配列を宣言しています。この 1 バイトの余分は、文字列の終端に付加する '¥0'（ヌル文字列、値は 0）の分です。

C 言語では、文字列は文字の並びの末尾に “¥0” を付加したものです。自ブロックのタグ名が「TAGNAME890123456」（16 文字）であれば、上記の変数 tagName には、以下のデータが返されます。

T	A	G	N	A	M	E	8	9	0	1	2	3	4	5	6	'¥0'
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	------

040705.ai

図 変数tagNameに返されるデータ

最後の 17 バイト目に、システムは文字列の終端を示す “¥0” を付加します。文字列を返すユーザカスタムアルゴリズム作成用ライブラリを使用する場合には、文字列を格納する配列はいつも “¥0” の分 1 バイトを余分に確保してください。引数に指定する文字配列を 1 バイト余分に確保する必要のあるユーザカスタムアルゴリズム作成用ライブラリを以下に示します。

表 文字列を返すユーザカスタムアルゴリズム作成用ライブラリ

関数名	機能	説明
UcaDataGetTagName	自ブロックのタグ名を取得します。	タグ名は最大 16 バイトなので配列は 17 バイト必要です。
UcaDataGetEachSn	データアイテム S01 ~ S16 から文字列を取得します。	S01 ~ S16 は最大 16 バイトなので配列は 17 バイト必要です。
UcaDataGetPrgn	データアイテム PRGN からユーザカスタムアルゴリズムのプログラム名称を取得します。	PRGN は最大 16 バイトなので配列は 17 バイト必要です。
UcaRWReadString	Q01 ~ Q32 端子から文字列を取得します。	Q01 ~ Q32 端子から入力する文字列最大 16 バイトなので配列は 17 バイト必要です。
UcaTagReadString	タグ名とデータアイテム名を指定して文字列データを取得します。	文字列データは最大 16 バイトなので配列は 17 バイト必要です。

4.8 Windowsのライブラリ

ユーザカスタムアルゴリズムのプログラムでユーザが使用できるWindows提供のライブラリについて説明します。ユーザはWindows提供の関数を次の規則に従って使用してください。

- Windows提供のインクルードファイルのうち、ユーザカスタムアルゴリズムで使用できるのは以下のインクルードファイルのみです。これら以外のWindows提供インクルードファイルを#includeで取り込まないでください。

表 ユーザカスタムアルゴリズムで使用できるWindows提供インクルードファイル

ファイル名	説明
string.h	strcpy()、strcmp() など文字列操作関数
math.h	sin()、cos()、exp() などの算術関数
stdlib.h	strtod()、div() などの標準関数
ctype.h	toupper()、isupper() などの文字操作関数
stddef.h	offsetof() 関数
time.h	ユーザカスタムアルゴリズム作成用ライブラリの UcaLocaltime() 関数などが使用
float.h	_isnan() 関数、および、浮動小数演算に関する定数
limits.h	整数の最大値、最小値を表す定数
stdio.h	sprintf() 関数

「表 ユーザカスタムアルゴリズムで使用できるWindows 提供インクルードファイル」のインクルードファイルで宣言されているすべての関数を使用できるわけではありません。ユーザが使用できる関数は限られています。ユーザカスタムアルゴリズムで使用可能な関数を以下に示します。

表 ユーザカスタムアルゴリズムで使用可能なWindows提供ライブラリー覧（1/2）

分類	関数名	機能	推奨されない関数(*1)	使用インクルードファイル
abs	labs()	整数の絶対値を求める	—	math.h/stdlib.h
bsearch	bsearch_s()	2分検索を行なう	bsearch()	stdlib.h
conv	toupper()	大文字に変換	—	stdlib.h/ctype.h
	tolower()	小文字に変換	—	stdlib.h/ctype.h
ctype	isupper()	大文字かどうかのチェック	—	ctype.h
	islower()	小文字かどうかのチェック	—	ctype.h
	isdigit()	0～9の文字かのチェック	—	ctype.h
	isxdigit()	0～9、A～Fかのチェック	—	ctype.h
	isalnum()	アルファニューメリックかどうかのチェック	—	ctype.h
	isprint()	印字文字かどうかのチェック	—	ctype.h
div	ldiv()	整数の割算の答えと余りを求める	—	stdlib.h
exp	exp()	exp(x) を求める	—	math.h
	log()	log(x) を求める	—	math.h
	pow()	x の y 乗を求める	—	math.h
	sqrt()	x の平方根を求める	—	math.h
floor	floor()	x を越えない最大の整数を返す	—	math.h
	ceil()	x より大きくて最小の整数を返す	—	math.h
	fmod()	浮動小数点割算の余りを求める	—	math.h
	fabs()	x の絶対値を求める	—	math.h
_isnan	_isnan()	NAN (Not A Number) のチェック	—	float.h

表 ユーザカスタムアルゴリズムで使用可能なWindows提供ライブラリ一覧 (2/2)

分類	関数名	機能	推奨されない 関数 (*1)	使用インクルードファイル
memory	memccpy()	ターミネタつきメモリー領域コピー	—	string.h
	memchr()	メモリー領域から char データを探す	—	string.h
	memcmp()	メモリー領域の比較	—	string.h
	memcpy_s()	メモリー領域のコピー	memcpy()	string.h
	memmove_s()	重なりがあるメモリー領域のコピー	memmove()	string.h
	memset()	メモリー領域への特定データセット	—	string.h
offsetof	-offsetof()	構造体メンバーのオフセット値を求める	—	stddef.h
string	strcat_s()	文字列の結合	strcat()	string.h
	strncat_s()	長さ指定の文字列の結合	strncat()	string.h
	strcmp()	文字列の比較	—	string.h
	strncmp()	長さ指定の文字列の比較	—	string.h
	strcpy_s()	文字列のコピー	strcpy()	string.h
	strncpy_s()	長さ指定の文字列のコピー	strncpy()	string.h
	strlen()	文字列長を求める	—	string.h
	strchr()	文字列から最初に現れた特定文字の抽出	—	string.h
	strrchr()	文字列の最後に現れた特定文字の抽出	—	string.h
	strpbrk()	文字パターンのサーチ	—	string.h
strtod	strtod()	文字列の倍精度浮動小数点数への変換	—	stdlib.h
	strtol()	文字列の整数への変換	—	stdlib.h
strtoul	strtoul()	文字列の符号なし整数への変換	—	stdlib.h
swab	_swab()	偶数バイトと奇数バイトの入れ換え	swab()	stdlib.h
trig	sin()	sin(x) を求める	—	math.h
	cos()	cos(x) を求める	—	math.h
	tan()	tan(x) を求める	—	math.h
	asin()	arcsin(x)	—	math.h
	acos()	arccos(x)	—	math.h
	atan()	arctan(x)	—	math.h
sprintf	sprintf_s()	フォーマッタ	sprintf()	stdio.h

補足 上記の表にあげた関数の説明の詳細については、MSDN ライブラリの C ランタイムライブラリを参照してください。

*1 : CS 3000 のユーザカスタムアルゴリズムをリビルドして使用する場合などで、推奨されていない関数を使用するとコンパイル時に警告が表示されることがあります。新しい _s 付きの関数を使用する事を推奨します。

5. 汎用演算形ユーザカスタムブロックのプログラミング

汎用演算形ユーザカスタムブロック（CSTM-A）は、入力したデータに対して汎用的な演算を実行することを主用途とするユーザカスタムブロックです。この章ではサンプルプログラムをもとに、CSTM-Aのプログラミングをデータの入力と出力を中心に説明します。

■ CSTM-Aのデータアイテムとプログラミング

汎用演算形ユーザカスタムブロックでは、演算結果はデータアイテム CPV に表示します。また、CSTM-A は演算結果を出力端子より出力することができます。CSTM-A は、32 の入力端子と 16 の出力端子を持ちます。さらに、演算入力値 RV、RV01～RV32 と演算出力値 CPV、CPV01～CPV16 などのデータアイテムを持ちます。CSTM-A の構成を以下に示します。

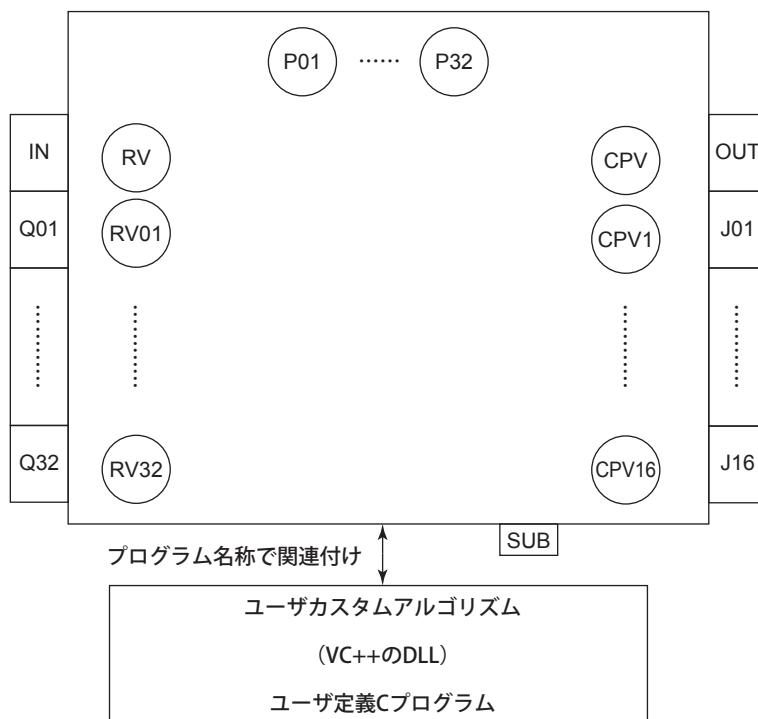


図 CSTM-Aの構成

この章では、データ入力と出力に入出力端子を使用したプログラミングを説明します。タグ名を指定したデータの入出力は、入出力端子用のユーザカスタムアルゴリズム作成用ライブラリを呼び出している部分を、タグ名を指定するユーザカスタムアルゴリズム作成用ライブラリの呼び出しに置きかえることで実現できます。

参照 タグ名を指定したプログラムのデータ入力と出力の記述は、CSTM-A と CSTM-C で同様です。詳細については、以下の参照してください。

[「7. タグ名を指定したデータ入力」](#)

5.1 多入力CSTM-A

汎用演算形カスタムブロック（CSTM-A）の入力処理と演算出力値CPVへの出力処理のプログラミングを説明します。ここでは、3つの入力データをIN端子、Q01端子、Q02端子から読み込み、演算結果をデータアイテムCPVに格納するユーザカスタムアルゴリズムを説明します。

サンプルプログラムが用意してあります。サンプルソリューション _SMPL_ALGO_IN の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは 1 章の準備作業により以下の作業フォルダにコピーされています。

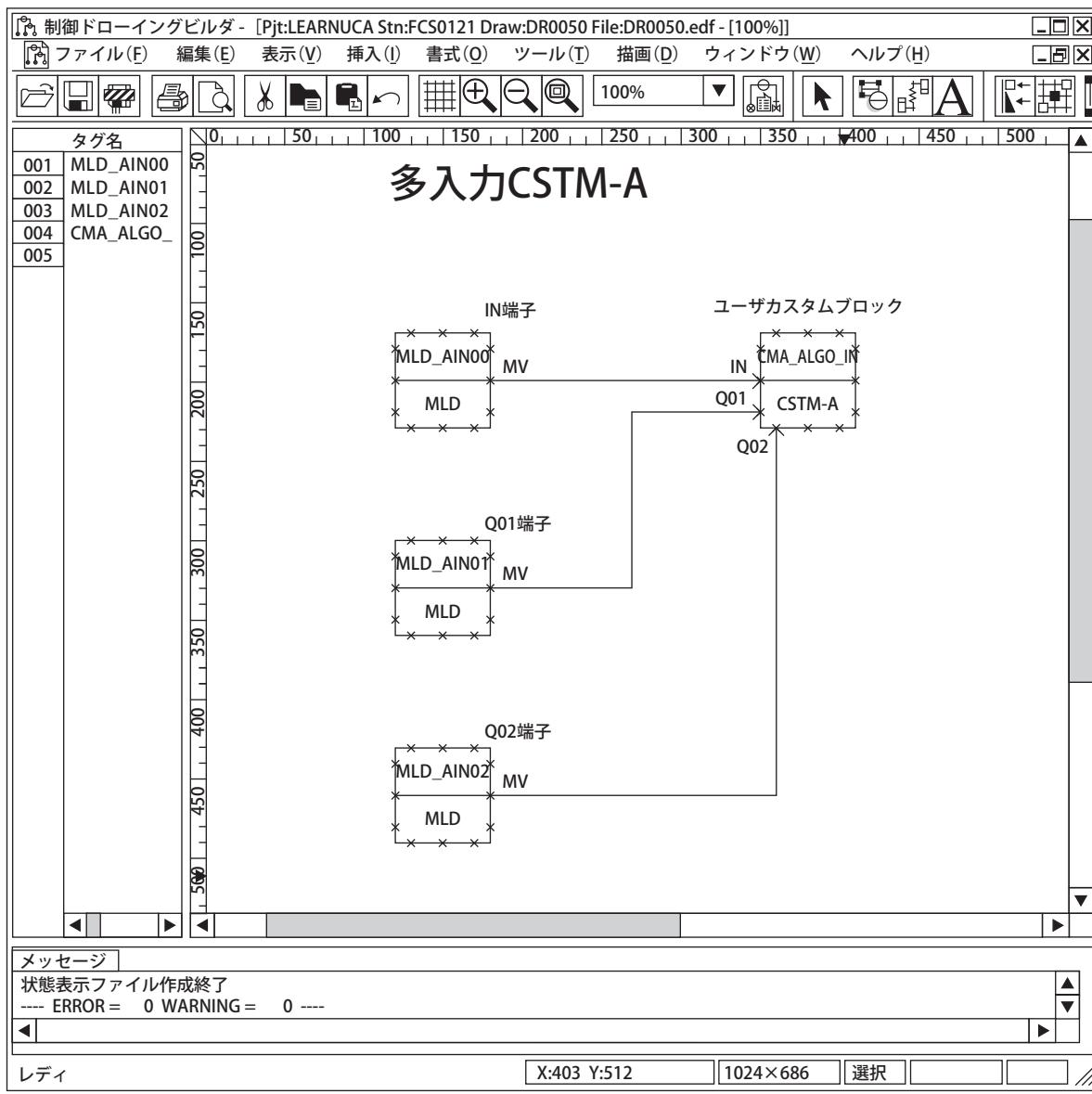
<ドライブ名>:\UcaWork\UcaSamples\SMPL_ALGO_IN

Visual Studio を起動し、_SMPL_ALGO_IN\SMPL_ALGO_IN.sln を開きます。[ビルド] メニューの [リビルド] で _SMPL_ALGO_IN の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。

次に、サンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> \UcaEnv\Sample\LearnUca\drawings\ALGO_IN.txt

FCS0121 (APCS) の制御ドローイングの DR0050 (空いている制御ドローイングならどれでも構いません) を指定して制御ドローイングビルダを起動します。[ファイル] メニューの [外部ファイル] – [インポート] を指定します。インポートダイアログで上記の ALGO_IN.txt を指定し [開く] ボタンをクリックすると、次図の制御ドローイングが取り込まれます。[ファイル] – [上書き保存] で制御ドローイングを書き込みます。



050101J.ai

図 制御ドローイングのインポート

この制御ドローイングでは、3つの手動操作ブロック（MLD）がデータアイテム MV へ出力するデータを汎用演算形カスタムブロック（CSTM-A）が IN 端子、Q01 端子、Q02 端子から入力しています。CSTM-A の演算出力値を保持するデータアイテム CPV には、次の式の計算結果を格納します。

$$CPV = IN \text{ 端子入力データ} + Q01 \text{ 端子入力データ} + Q02 \text{ 端子入力データ}$$

動作を確認するためにコントロール（8ループ）のウィンドウが用意してありますので、HIS0164 の CG0050 に取り込んでおきます。テキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥graphics¥ALGO_IN_CG.xaml

システムビューで、HIS0164 の WINDOW にウィンドウ種「コントロール（8ループ）」、ウィンドウ名「CG0050」を作成します（ウィンドウ名は自由に命名してください）。

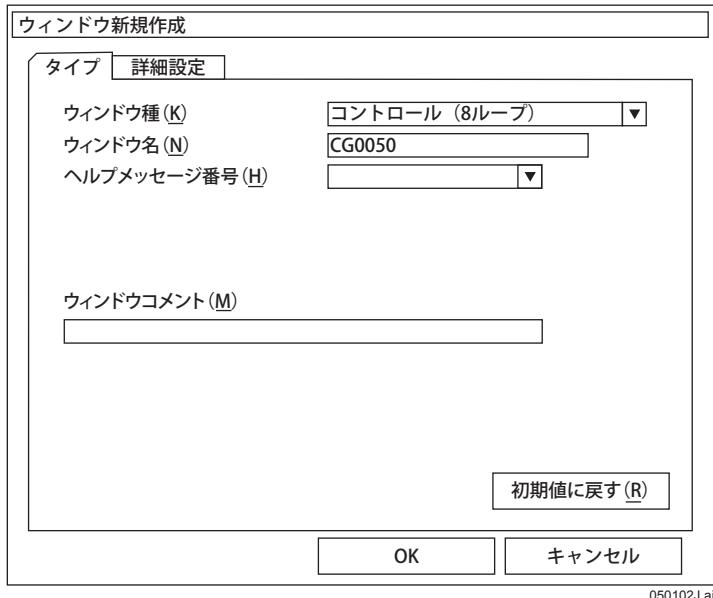
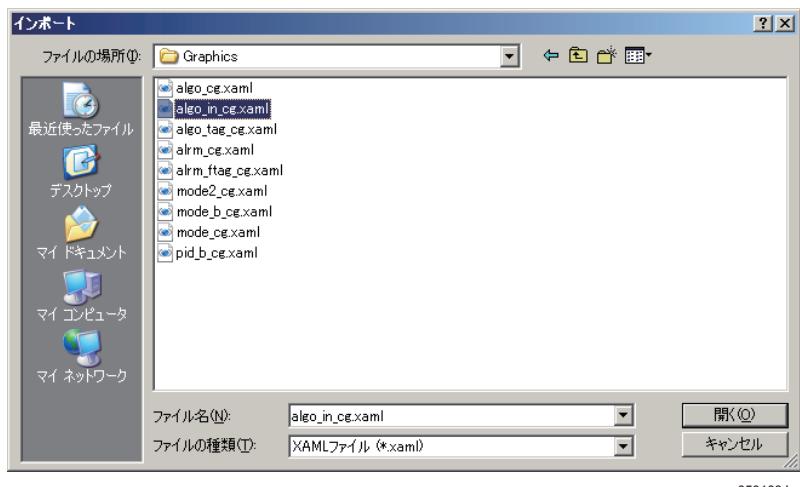


図 コントロールウィンドウの作成

HIS0164 の WINDOW に CG0050 ができますので、システムビューより CG0050 をダブルクリックします。

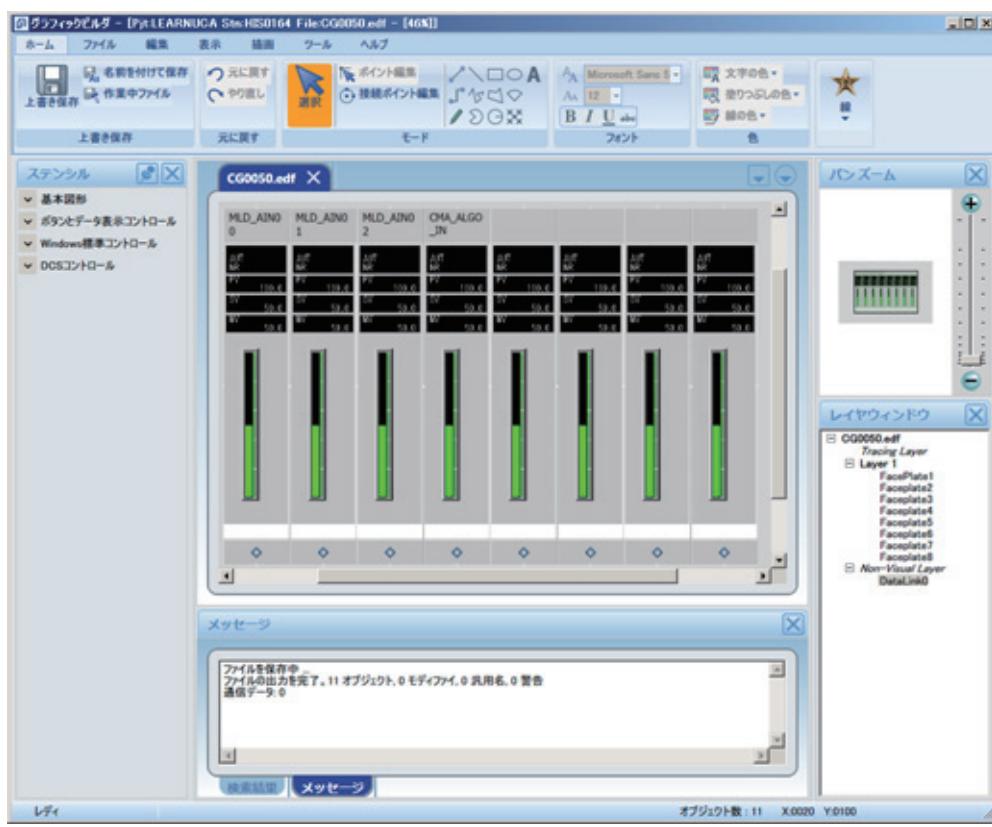
グラフィックビルダの [ファイル] – [外部ファイル] – [インポート] で以下のダイアログを表示し、algo_in_cg.xaml を指定し、[開く] ボタンをクリックします。



050103J.ai

図 ファイルを開くダイアログ

グラフィックビルダの [ファイル] – [上書き保存] でファイルを書き込みます。



050104J.ai

図 ファイルの保存

プログラムを動かしてみます。FCS0121 (APCS) を指定してバーチャルテスト機能を起動します。起動が完了したらウィンドウ CG0050 を表示します。



図 ウィンドウの呼び出し

3つの手動操作ブロック (MLD_AIN00, MLD_AIN01, MLD_AIN02) と 1 つの汎用演算形ユーザカスタムブロック (CMA_ALGO_IN) が並んでいます。CMA_ALGO_IN のデータアイテム CPV には、3つの手動操作ブロックの MV 値の合計が表示されます。手動操作ブロックの MV 値を変更し、CMA_ALGO_IN のデータアイテム CPV が実効スキャン周期 (4 秒) ごとに 3 つの MV 値の合計を表示するのを確認してください。以下は MV 値にそれぞれ 10.0, 20.0, 30.0 を指定し、その結果 CMA_ALGO_IN のデータアイテム CPV が 60.0 になっています。

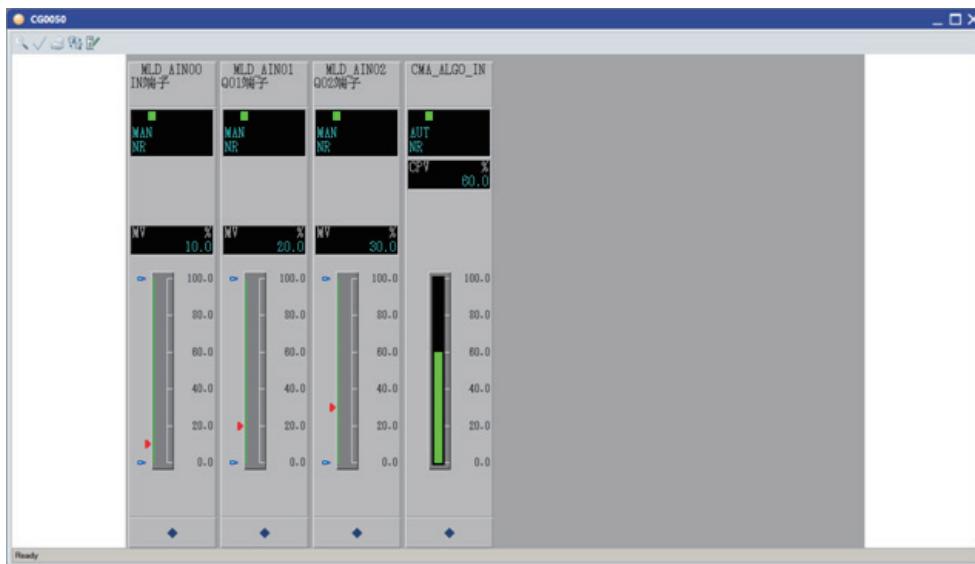


図 CMA_ALGO_INのCPV

CPV の値は 3 つの MV 値の合計です。手動操作ブロックの MV 値を変更すると、CMA_ALGO_IN の実効スキャン周期 (4 秒) ごとに CPV が追隨することを確認してください。

■ 多入力のサンプルプログラム

ユーザカスタムアルゴリズム _SMPL_ALGO_IN の algo_in.c について説明します。UcaBlockPeriodical という名前で検索して、機能ブロック定周期処理を見つけてください。

```
/*
*<<FNH>>*****
*
* Function name:      UcaBlockPeriodical
* Return value:       SUCCEED      正常終了
*                      UCAERR_NOPROC    処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description:        機能ブロック定周期処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnReadIn;          /* IN 端子入力リターンコード */
    F64S rv;                /* RV */
    F64S rv01;              /* RV01 */
    F64S rv02;              /* RV02 */
    F64S cpv;               /* CPV */
    I32 rtnCode;             /* リターンコード */

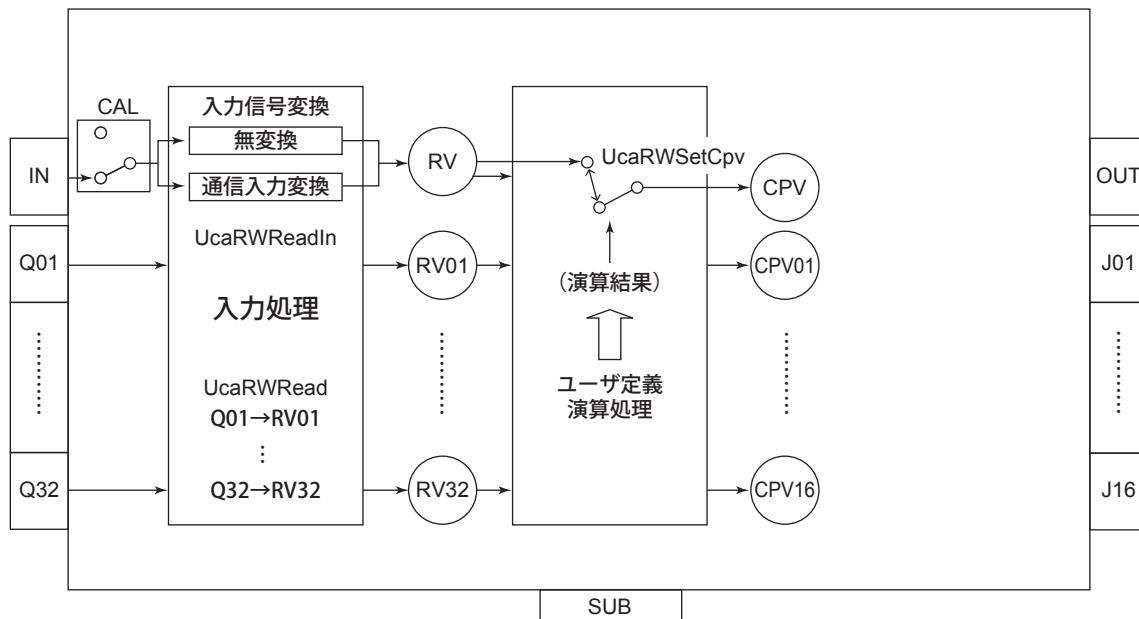
    /* ===== 入力処理 ===== */
    rtnReadIn = UcaRWReadIn(bc, NOOPTION);           /* IN 端子から RVへ読み込み */
    rtnCode = UcaRWRead(bc, 1, NOOPTION);             /* Q01 端子から RV01へ読み込み */
    rtnCode = UcaRWRead(bc, 2, NOOPTION);             /* Q02 端子から RV02へ読み込み */

    /* ===== 演算処理 ===== */
    /* 入力したデータをデータアイテムから変数に読み込み */
    rtnCode = UcaDataGetRv(bc, &rv, NOOPTION);         /* RV */
    rtnCode = UcaDataGetRvn(bc, &rv01, 1, 1, NOOPTION); /* RV01 */
    rtnCode = UcaDataGetRvn(bc, &rv02, 2, 1, NOOPTION); /* RV02 */
    /* 演算を実行： CPV = RV + RV01 + RV02
     * (演算エラーの検出は省略しデータステータスは常に正常)
    */
    cpv.value = rv.value + rv01.value + rv02.value;
    cpv.status = 0; /* データステータス正常 */

    /* 演算結果をデータアイテム CPV に出力 */
    /* 引数 rtnReadIn は、IN 端子読み込みのリターンコード */
    rtnCode = UcaRWSetCpv(bc, &cpv, rtnReadIn, NOOPTION);

    return SUCCEED;
}
```

CSTM-A における入力データの流れを以下に示します。C プログラムと合わせてみてください。



050107J.ai

図 CSTM-Aにおける入力データの流れ

IN 端子から読み込んだデータは、データアイテム RV に格納されます。また、入力端子 Q01 と Q02 から読み込んだデータは、それぞれデータアイテム RV01 と RV02 に格納されます。プログラムでは以下の部分です。

```
.....
/* ====== 入力処理 ===== */
rtnReadIn = UcaRWReadIn(bc, NOOPTION); /* IN 端子から RVへ読み込み */
rtnCode = UcaRWRead(bc, 1, NOOPTION); /* Q01 端子から RV01 へ読み込み */
rtnCode = UcaRWRead(bc, 2, NOOPTION); /* Q02 端子から RV02 へ読み込み */
....
```

● IN端子からの入力処理

UcaRWReadIn は IN 端子からデータを入力し、データアイテム RV に格納します。UcaRWReadIn は CSTM-A のビルダ定義項目「入力信号変換タイプ」に「通信入力」を指定すれば、入力データを「通信入力変換」します。

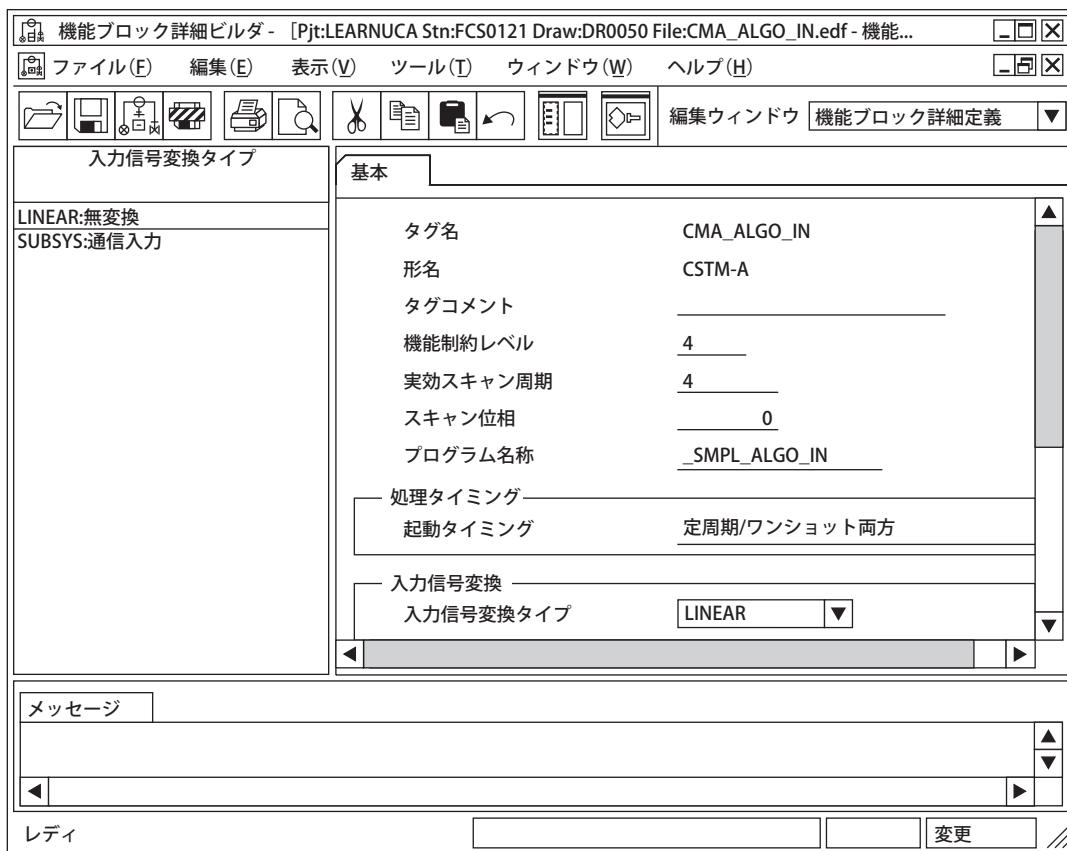


図 IN端子からの入力処理

050108J.ai

参照

入力信号変換の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「3.1.1 連続制御ブロックと演算ブロックに共通の入力信号変換」](#)

● Qnn端子からの入力処理

UcaRWRead は Qnn 端子からデータを入力し、データアイテム RVnn に格納します。UcaRWRead(bc, 2, NOOPTION) という行では第 2 引数に「2」が指定されているので、Q02 端子からデータを入力しデータアイテム RV02 に格納します。UcaRWRead は、Qnn 端子からデータアイテム RVnn へデータ入力をするだけで、「入力信号変換タイプ」の指定は UcaRWRead の動作には何も関係ありません（常に無変換でデータ入力されます）。

● 演算処理

これで入力端子からのデータ入力が終了しました。次に、データアイテムから C 言語の変数にデータを取得し、演算処理を実行します。

```
.....
/* ===== 演算処理 ===== */
/* 入力したデータをデータアイテムから変数に読み込み */
rtnCode = UcaDataGetRv(bc, &rv, NOOPTION); /* RV */
rtnCode = UcaDataGetRvn(bc, &rv01, 1, 1, NOOPTION); /* RV01 */
rtnCode = UcaDataGetRvn(bc, &rv02, 2, 1, NOOPTION); /* RV02 */

/* 演算を実行:CPV=RV+RV01+RV02
 * (演算エラーの検出は省略しデータステータスは常に正常)
 */
cpv.value = rv.value + rv01.value + rv02.value;
cpv.status = 0; /* データステータス正常 */
.....
```

データアイテムから取得したデータを格納する変数は、ユーザ定義関数 UcaBlockPeriodical の最初で以下のように宣言されています。

```
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
.....
    F64S rv;      /* RV */
    F64S rv01;   /* RV01 */
    F64S rv02;   /* RV02 */
    F64S cpv;    /* CPV */
    F64S p01;    /* P01 */
.....
```

UcaDataGetRv はデータアイテム RV からデータを取得し、引数に指定された変数 rv に格納しています。また UcaDataGetRvn は、データアイテム RV01 と RV02 からデータを取得し、引数に指定された変数 rv01 と rv02 に格納しています。

データの取得が済むと $RV + RV01 + RV02$ の演算を行い、計算結果を cpv.value (value はデータ値を意味するメンバです) に格納しています。このプログラムではオーバフローなど浮動小数演算エラーに検出はしないで、データステータス (cpv.status) には 0 (データステータス正常) を格納しています。

参照 浮動小数演算エラーの検出の詳細については、以下を参照してください。
[「5.2 多入力多出力 CSTM-A」](#)

変数 cpv に設定された演算結果は、UcaRWSetCpv でデータアイテム CPV に設定します。UcaRWSetCpv の第 3 引数には、UcaRWReadIn のリターンコード（変数 rtnReadIn）を指定します。

```
.....  
/* 演算結果をデータアイテム CPV に出力 */  
/* 引数 rtnReadIn は、IN 端子読み込みのリターンコード */  
rtnCode = UcaRWSetCpv(bc, &cpv, rtnReadIn, NOOPTION);  
.....
```

UcaRWSetCpv はデータアイテム CPV にデータを格納するだけでなく、CPV のデータステータス作成やデジタルフィルタ処理などを行います。

■ UcaRWSetCpv によるCPVのデータステータス作成

UcaRWSetCpv は、演算結果をデータアイテム CPV に格納します。UcaRWSetCpv の呼び出し部分を見てみます。

```
.....
/* 演算結果をデータアイテム CPV に出力 */
/* 引数 rtnReadIn は、IN 端子読み込みのリターンコード */
rtnCode = UcaRWSetCpv(bc, &cpv, rtnReadIn, NOOPTION);
.....
```

UcaRWSetCpv は、次の処理をします。

- CPV のデータステータス作成
- デジタルフィルタ処理 (UcaRWSetCpv にオプション UCAOPT_NOFILTER を指定すると、ビルダ定義項目「入力フィルタ」の定義に関係なくデジタルフィルタ処理をしません)
- 積算処理 (UcaRWSetCpv にオプション UCAOPT_NOSUM を指定すると、ビルダ定義項目「積算単位時間」の定義に関係なく積算処理をしません)
- CPV へのデータ設定

参照 デジタルフィルタ、積算の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「3.2 デジタルフィルタ」](#)

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「3.3 積算」](#)

CPV のデータステータス作成について説明します。データステータス作成方法は、ビルダ定義項目「演算入力値異常検出」の指定により異なります。汎用演算形ユーザカスタムブロック CMA_ALGO_IN では、「全検出形」を指定しています (CSTM-A のデフォルトは非検出形です)。

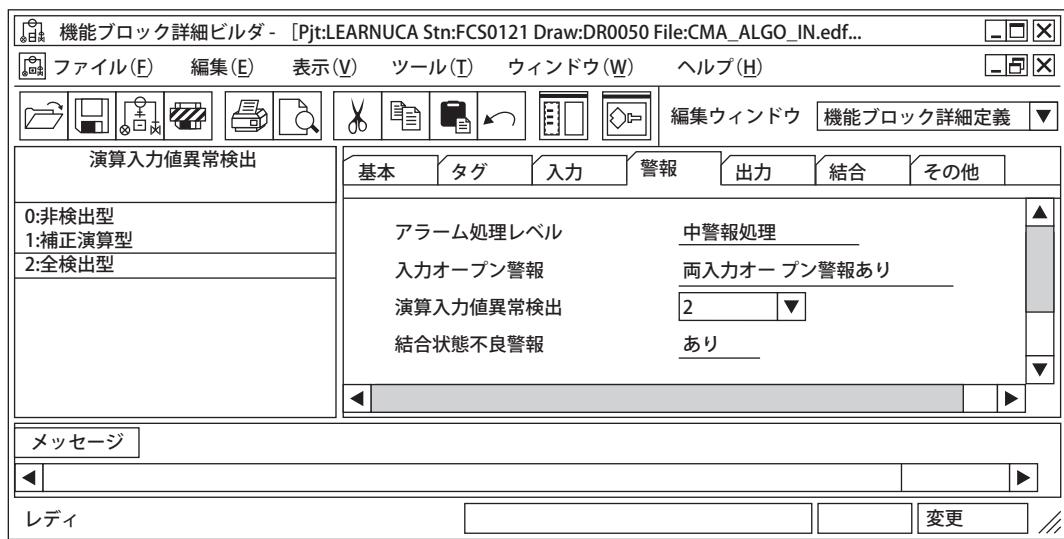


図 演算入力値異常検出の指定

050109J.ai

UcaRWSetCpv の動作を以下に示します。

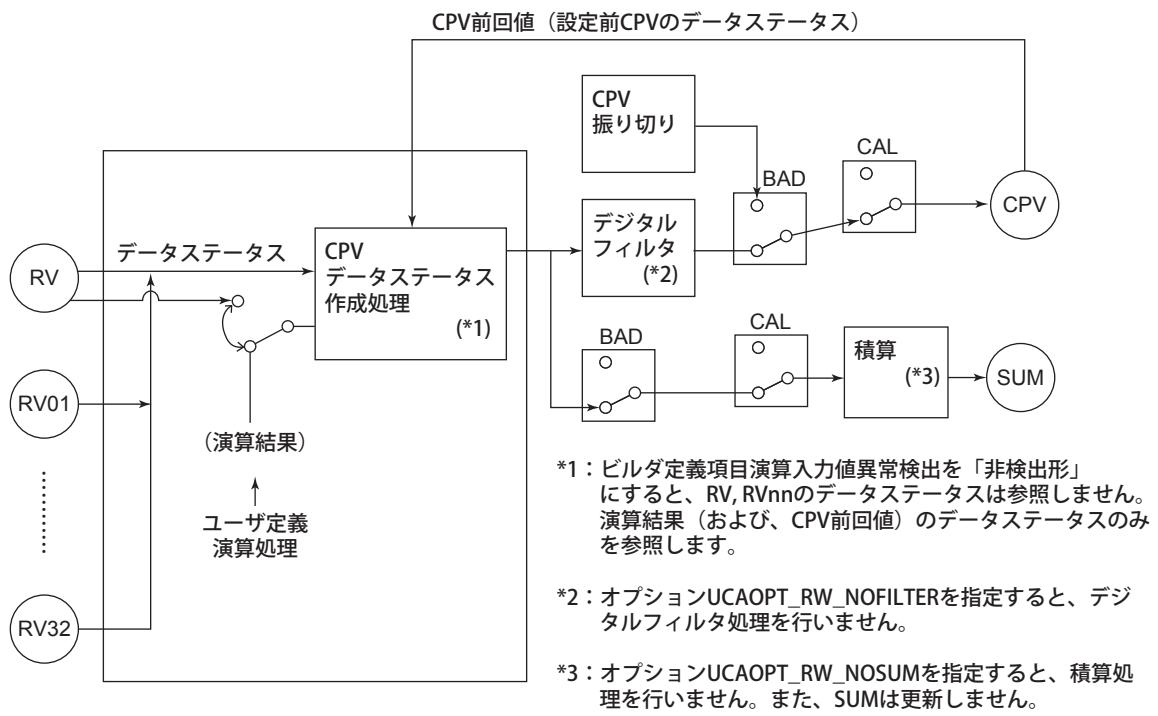


図 UcaRWSetCpvの動作

CPV のデータステータス BAD と QST は、次の 3 つのデータから作成されます。

- UcaRWSetCpv の引数に指定する演算結果のデータステータス BAD と QST。
ただし、「UcaRWSetCpv(bc, NULL, rtnReadIn, NOOPTION);」のように UcaRWSetCpv の第 2 引数（演算結果を指定）に「NULL」を指定すると、演算結果のデータステータスは正常（BAD でも QST でもない）として扱われます。
- データアイテム **RV** に入力したデータのデータステータス BAD と QST。
- データアイテム **RV01** ~ **RV32** に入力したデータのデータステータス BAD と QST。

UcaRWSetCpv はこの 3 つのデータのデータステータス (BAD と QST) をビルダ定義項目「演算入力値異常検出」の指定により、以下のように使用してデータアイテム CPV に設定するデータステータス (BAD と QST) を決定します。なお、UcaRWSetCpv に UCAOPT_RW_NOPVSTS オプションを指定すると、ビルダ定義項目「演算入力値異常検出」の指定とは無関係に常に「非検出形」指定と同じ動作になります。

表 CPV データステータス作成

ビルダ定義項目 「演算入力値異常検出」 の指定	(1) UcaRWSetCpv の引 数のデータステータス	(2) RV のデータ ステータス	(3) RVnn のデータ ステータス	作成される CPV の データステータス
全検出形	BAD	任意	任意	BAD
	任意	BAD	任意	BAD
	任意	任意	BAD	BAD(*1)
	QST	not BAD	not BAD	QST
	not BAD	QST	not BAD	QST
	not BAD	not BAD	QST	QST
	not BAD nor QST	not BAD nor QST	not BAD nor QST	0 (正常)
補正演算形	BAD	任意	任意	BAD
	任意	BAD	任意	BAD
	任意	任意	BAD	QST(*1)
	QST	not BAD	not BAD	QST
	not BAD	QST	BAD	QST
	not BAD	not BAD	QST	QST
	not BAD nor QST	not BAD nor QST	not BAD nor QST	0 (正常)
非検出形	BAD	任意	任意	BAD
	QST	任意	任意	QST
	not BAD nor QST	任意	任意	0 (正常)
UCAOPT_RW_NOPVSTS オプションを指定 （「演算入力値異常検出」 の指定は無視されます）	BAD	任意	任意	BAD
	QST	任意	任意	QST
	not BAD nor QST	任意	任意	0 (正常)

任意： 任意のデータステータス

*1： 全検出形と補正演算形では、RVnn が BAD の場合の扱いのみが異なります。

ビルダ定義項目と UcaRWSetCpv のオプション UCAOPT_RW_NOPVSTS の指定により、データステータスの作成方法が決まります。データステータスの作成方法が決まると、上表の各行の条件を上から下に評価して最初に該当する行に従って CPV のデータステータスの BAD と QST が決定されます。

データステータス作成処理は、ユーザ定義関数の入り口（たとえば、機能ブロック定期処理なら UcaBlockPeriodical の先頭行）から UcaRWSetCpv を呼び出すまでに、以下のユーザカスタムアルゴリズム作成用ライブラリによりデータが設定されているデータアイテム (RV01 ~ RV32) のみを対象とします。

表 データステータス作成の対象となるRVnnを決める関数

関数名	処理内容
UcaRWRead	入力端子 Qnn からデータアイテム RVnn へのデータ入力
UcaDataStoreRvn	データアイテム RVnn へのデータ設定
UcaTagReadToRvnF64S	タグ名を指定してデータを入力し、入力データを RVnn に設定 (自ステーションデータ)
UcaOtherTagReadToRvnF64S	タグ名を指定してデータを入力し、入力データを RVnn に設定 (他ステーションデータ)

重要

- 結合を定義してある入力端子からのみデータ入力処理をしてください。入力結合を定義していない入力端子 Q01 ~ Q32 に対して UcaRWRead で入力端子からデータを読み込むと、対応するデータアイテム RV01 ~ RV32 のデータステータスは QST になります (RV01 ~ RV32 のデータ値は不变です)。

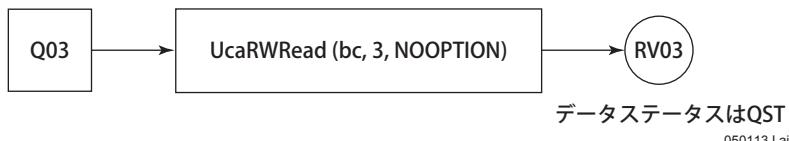


図 入力結合未定義のQnn端子からの読み込み

- 1回のユーザ定義関数の処理内では、UcaRWSetCpv は一度だけ呼び出してください。機能ブロック定期処理の場合、UcaBlockPeriodical の入り口から return するまでの間、UcaRWSetCpv を一度だけ呼び出してください。
1回のユーザ定義関数の処理内では、UcaRWSetCpv を複数回呼び出さないでください。複数回呼び出すと、たとえば SUB 端子から出力可能な ΔCPV (前回 CPV と今回 CPV の差分) が正常に作成できなくなります。 ΔCPV は、UcaRWSetCpv が呼び出された間の CPV の差分です。UcaRWSetCpv を1回のユーザ定義関数の処理内で2回呼び出すと、 ΔCPV は同じ処理内の2回の UcaRWSetCpv 呼び出しに指定した CPV 値の差分となってしまいます。

5.2 多入力多出力CSTM-A

この節では、汎用演算形ユーザカスタムブロックの出力処理について説明します。また、演算異常アラームERRCの検出についても説明します。前節の多入力CSTM-Aのサンプルに対して、次の機能を追加し、プログラムを完成します。

- OUT端子、J01端子、J02端子からのデータ出力
- 演算異常アラームERRCの発生と復帰
- データアイテムERRCとERRLへのエラー情報の保存
- チューニングパラメータより弱い初期値の設定方法

実際の運転に使用するユーザカスタムアルゴリズムは、（前節のプログラムではなく）機能追加されたこの節のプログラムを参考にしてください。

サンプルプログラムが用意してありますので、動かしてみます。サンプルソリューション _SMPL_ALGO の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは 1 章の準備作業により以下の作業フォルダにコピーされています。

<ドライブ名> ¥UcaWork¥UcaSamples¥_SMPL_ALGO

Visual Studio を起動し、_SMPL_ALGO¥_SMPL_ALGO.sln を開きます。[ビルド] メニューの [リビルド] で _SMPL_ALGO の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。

次に、サンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥drawings¥ALGO.txt

FCS0121 (APCS) の制御ドローイングの DR0051 (空いている制御ドローイングならどれでも構いません) を指定して制御ドローイングビルダを起動します。[ファイル] メニューの [外部ファイル] – [インポート] を指定します。インポートダイアログで上記の ALGO.txt を指定し、[開く] ボタンをクリックすると、次図の制御ドローイングが取り込まれます。[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

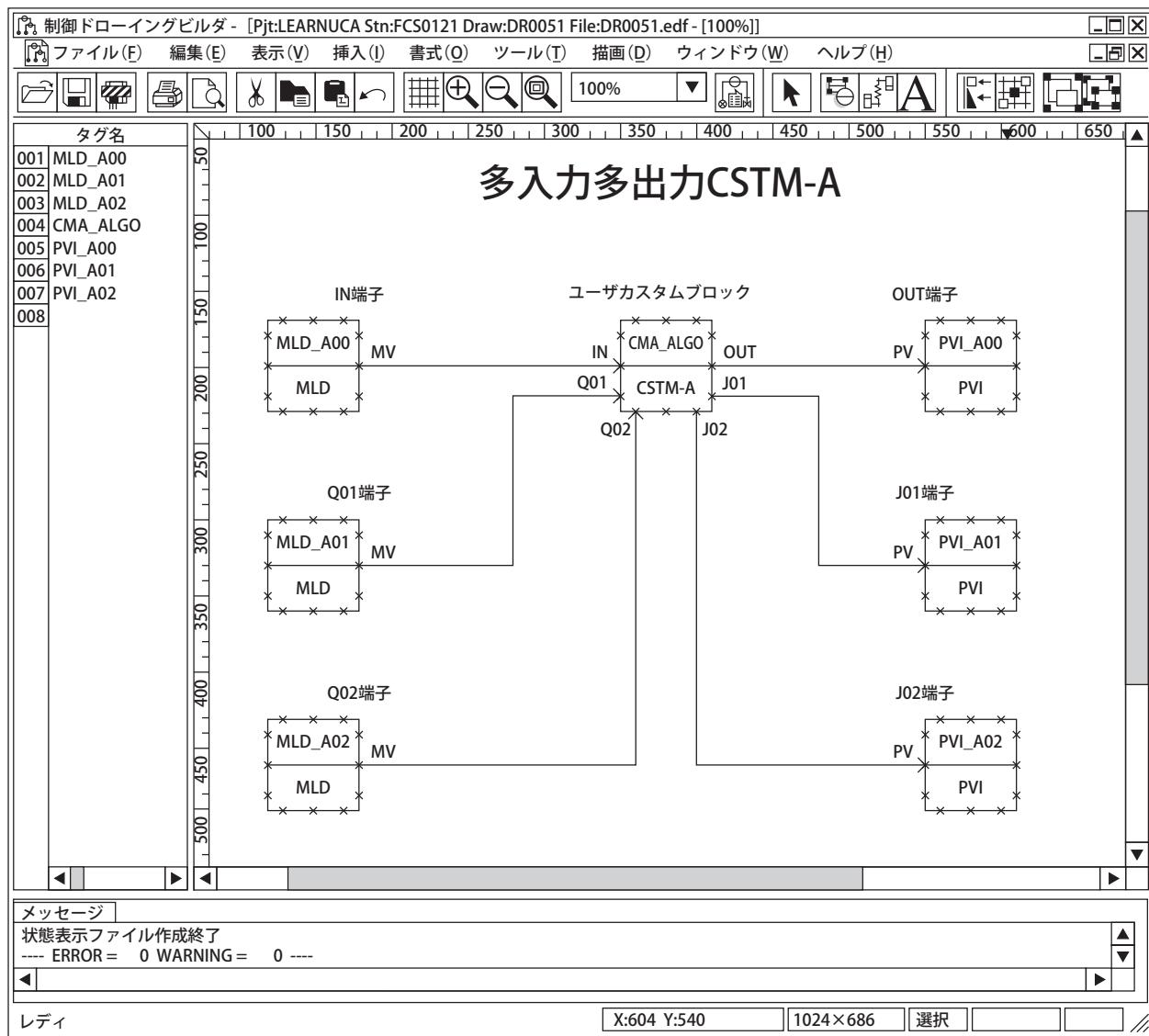


図 制御ドローイングのインポート

この制御ドローイングでは、3つの手動操作ブロック（MLD）がデータアイテム MV へ出力するデータを汎用演算形カスタムブロック（CSTM-A）が IN 端子、Q01 端子、Q02 端子から入力しています。また、ユーザカスタムブロックが OUT 端子、J01 端子、J02 端子から出力するデータを、3つの指示ブロック（PVI）のデータアイテム PV に設定しています。CSTM-A の演算出力値を保持するデータアイテム CPV、CPV01、CPV02 には、それぞれ次のような計算結果を格納します。このプログラムでは、データアイテム P01 に演算のパラメータとして倍率を格納しています。

$$CPV = P01 \times (IN \text{ 端子入力データ} + Q01 \text{ 端子入力データ} + Q02 \text{ 端子入力データ})$$

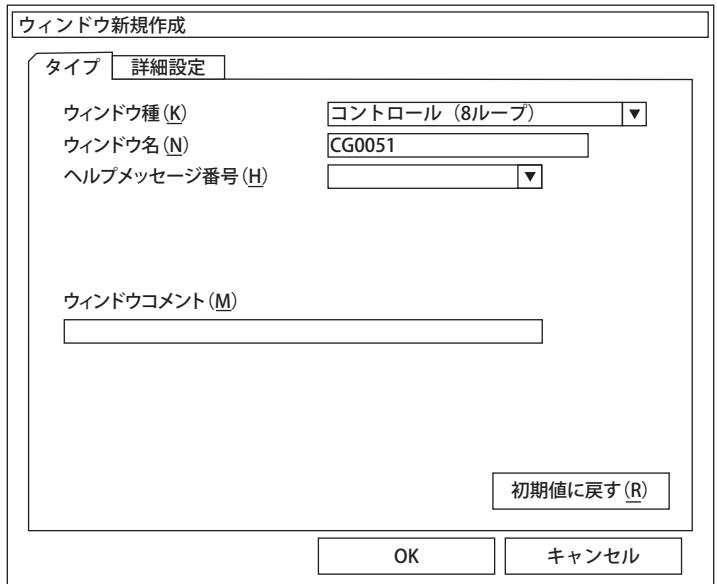
CPV01= IN 端子入力データ、Q01 端子入力データ、Q02 端子入力データの最大値

CPV02= IN 端子入力データ、Q01 端子入力データ、Q02 端子入力データの最小値

動作を確認するためにコントロール（8ループ）のウィンドウが用意してありますので、HIS0164 の CG0051 に取り込んでおきます。テキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥graphics¥ALGO_CG.xaml

システムビューで、HIS0164 の WINDOW にウィンドウ種「コントロール（8ループ）」、ウィンドウ名「CG0051」を作成します（ウィンドウ名は自由に命名してください）。

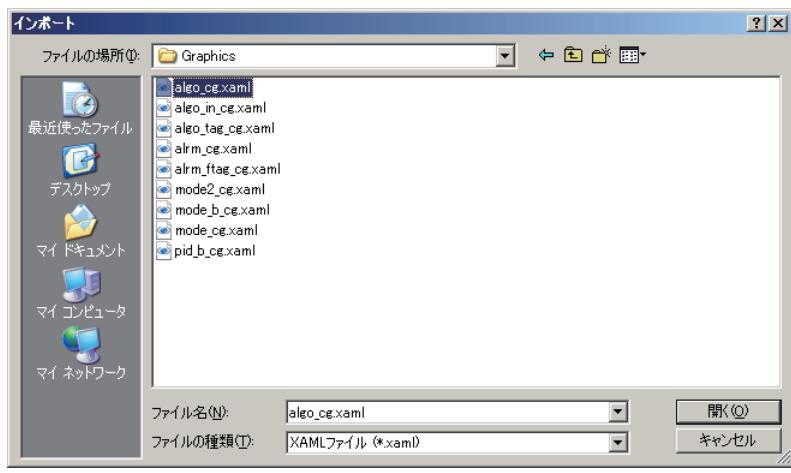


050202J.ai

図 コントロールウィンドウの作成

HIS0164 の WINDOW に CG0051 ができますので、システムビューより CG0051 をダブルクリックします。

グラフィックビルダの [ファイル] – [外部ファイル] – [インポート] で以下のダイアログを呼び出し、algo_cg.xaml を指定し、[開く] ボタンをクリックします。



050203J.ai

図 ファイルを開くダイアログ

グラフィックビルダの [ファイル] – [上書き保存] でファイルを書き込みます。

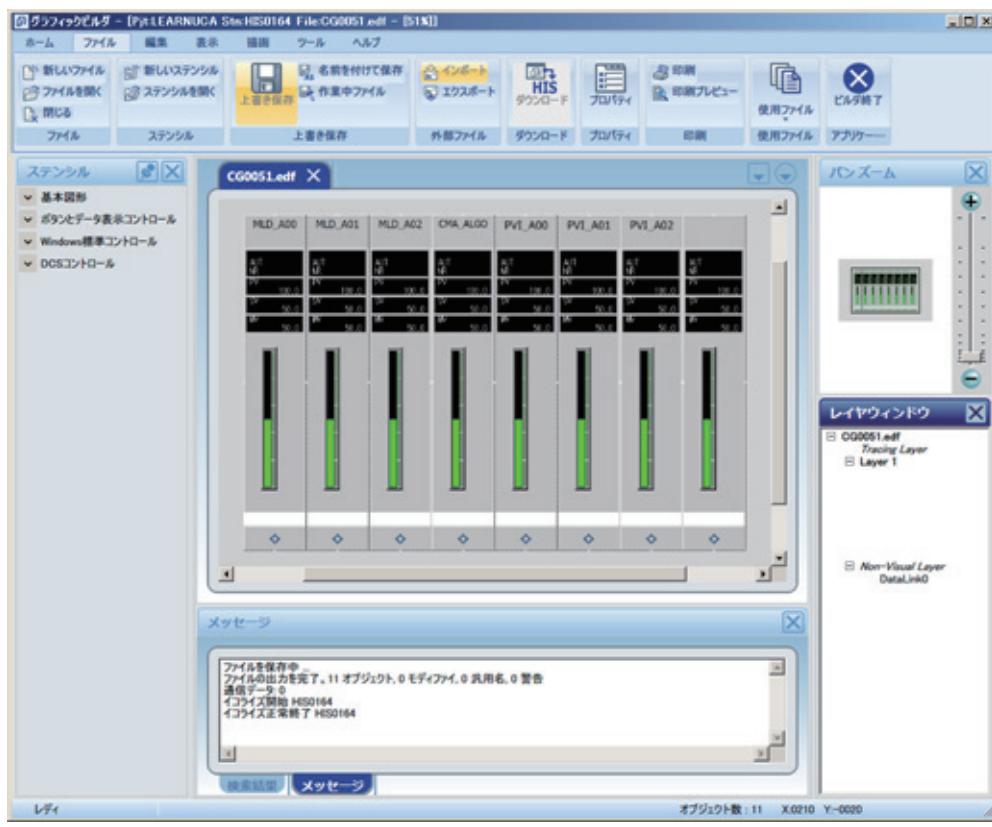


図 ファイルの保存

次に、FIC0121 (APCS) を指定してバーチャルテストを起動します。起動が完了したらウィンドウ CG0051 を表示します。



図 ウィンドウの呼び出し

3つの手動操作ブロック (MLD_A00, MLD_A01, MLD_A02) と1つの汎用演算形ユーザカスタムブロック (CMA_ALGO) が並んでいます。CMA_ALGO のデータアイテム CPV には、3つの手動操作ブロックの MV 値の合計が表示されます。手動操作ブロックの MV 値を変更し、CMA_ALGO のデータアイテム CPV が実効スキャン周期 (4秒) ごとに3つの MV 値の合計を表示するのを確認してください。以下は、MV 値にそれぞれ 10.0、20.0、30.0 を指定し、その結果 CMA_ALGO のデータアイテム CPV が 60.0 になっている様子です。

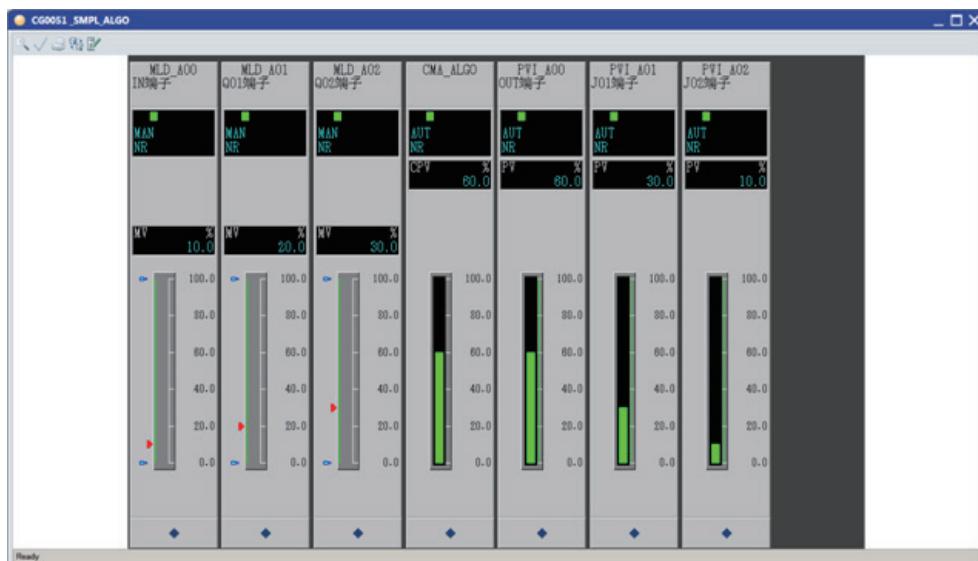


図 CMA_ALGOのCPV

CMA_ALGO の OUT 端子からはデータアイテム CPV (3つの入力の合計) のデータを出力します。J01 端子からは3つの入力の最大値を、J02 端子からは3つの入力の最小値をそれぞれ出力します。OUT 端子、J01 端子、J02 端子は、それぞれ指示ブロック PVI_A00、PVI_A01、PVI_A02 の PV と結合しています。手動操作ブロックの MV 値を変更すると、指示ブロックの PV が変化することを確認してください。

■ 多入力多出力のサンプルプログラム

ユーザカスタムアルゴリズム _SMPL_ALGO の algo.c について説明します。
UcaBlockPeriodical という名前で検索して機能ブロック定周期処理を見つけてください。

```
/*
* <<FNH>>*****
*
* Function name:      UcaBlockPeriodical
* Return value:       SUCCEED      正常終了
*                      UCAERR_NOPROC   処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description:        機能ブロック定周期処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnCode; /* リターンコード */

    /* 演算処理本体 */
    rtnCode = algo_calc(bc);

    return SUCCEED;
}
```

このプログラムは演算処理本体の関数 algo_calc を呼び出しています。algo_calc は同じソースファイル (algo.c) の下の方にあります。機能ブロック定周期処理のすぐあとに、機能ブロックワンショット起動処理 (UcaBlockOneshot) があるので確認してください。

```
/*
* <<FNH>>*****
*
* Function name: UcaBlockOneshot
* Return value: SUCCEED 正常終了
*                 UCAERR_NOPROC   処理なし
*                 UCAERR_STOPME   処理続行不能
*
* description: 機能ブロックワンショット起動処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc, /* (IN/OUT): ブロックコンテキスト */
    I32 code,           /* (IN): 種別コード */
    I32 parameter,      /* (IN): パラメータ */
    BOOL *result        /* (OUT): 実行結果 */
)
{
    I32 rtnCode;          /* リターンコード */

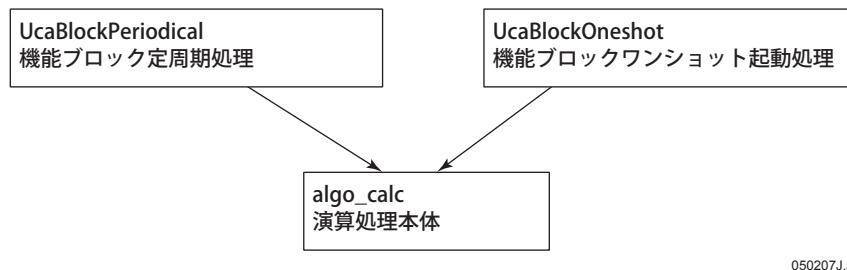
    /* 状態操作でなければ何もしない */
    if (code != UCAONESHOT_CODEOPRT) {
        return UCAERR_NOPROC;
    }

    /* 演算処理本体 */
    rtnCode = algo_calc(bc);

    *result = TRUE;

    return SUCCEED;
}
```

機能ブロックワンショット起動処理でも、algo_calc を呼び出しています。このユーザカスタムアルゴリズムは、実効スキャン周期による機能ブロック定周期処理と機能ブロックワンショット起動処理の両方で algo_calc 関数に記述してある同じ処理をします。

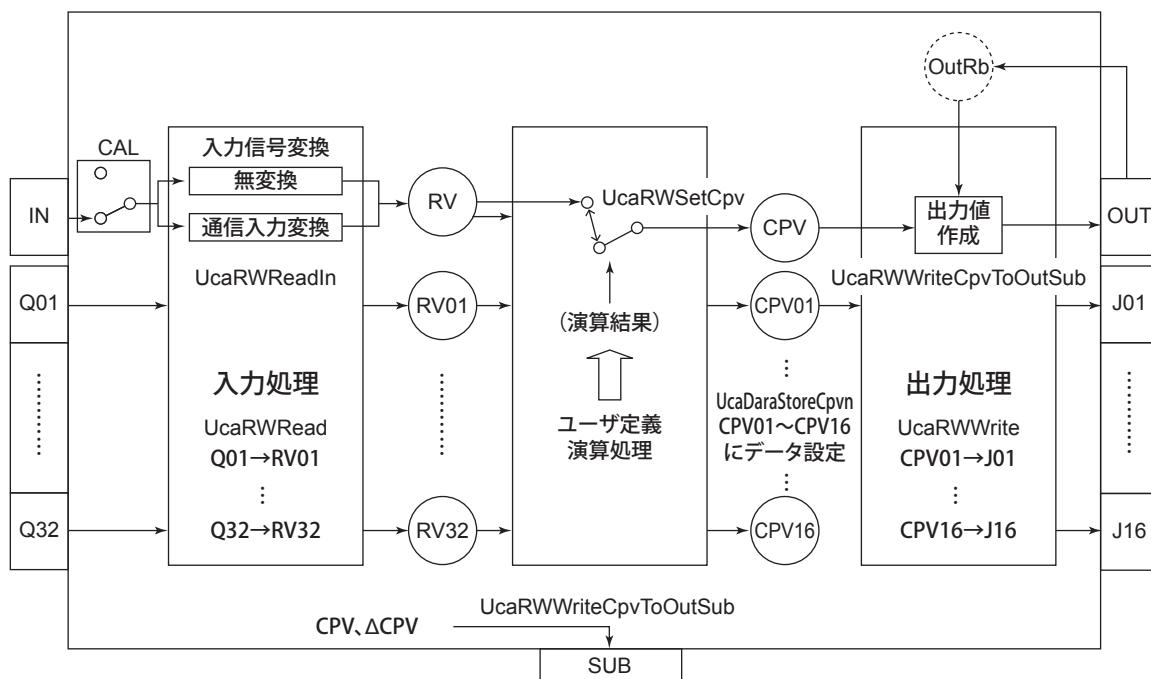


050207J.ai

参照 機能ブロック定周期処理と機能ブロックワンショット起動処理の使い分けの詳細については、以下を参照してください。

[「3.5.1 ビルダ定義項目「起動タイミング」」](#)

algo_calc のソースコードについて説明する前に、汎用演算形ユーザカスタムブロックのデータの流れについて説明します。



050208J.ai

図 汎用演算形ユーザカスタムブロックのデータの流れ

UcaRWReadIn は、IN 端子からデータを入力しデータアイテム RV に設定します。

UcaRWReadIn の入力動作として、ビルダ定義項目「入力信号変換タイプ」により「無変換」または「通信入力変換」を指定することができます。UcaRWRead は、入力端子 Q01 ~ Q32 よりデータを読み込み、データアイテム RV01 ~ RV32 に設定します。

ユーザ定義の演算処理は、データアイテム RV と RV01 ~ RV32 に入力したデータに対し演算を行います。そして演算結果を UcaRWSetCpv によりデータアイテム CPV に設定します。CPV 以外に出力するデータがあれば、UcaDataStoreCpvn により、データアイテム CPV01 ~ CPV16 に設定します。

参照 ユーザ定義の演算処理の詳細については、以下を参照してください。

[「5.1 多入力 CSTM-A」](#)

出力処理は、データアイテム CPV に設定された演算結果を UcaRWWWriteCpvToOutSub により OUT 端子から出力します。また、データアイテム CPV01 ~ CPV16 のデータを UcaRWWWrite により、出力端子 J01 ~ J16 から出力します。

5.2.1 入力処理とUcaFpuExpCheckによる演算エラーの検出

演算処理本体の関数algo_calcについて説明します。

ユーザカスタムアルゴリズム _SMPL_ALGO の algo.c から algo_calc という名前で検索し、以下の部分を見てください。

```
I32 algo_calc(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    F64S rv;           /* RV */
    F64S rv01;         /* RV01 */
    F64S rv02;         /* RV02 */
    F64S cpv;          /* CPV */
    F64S cpv01;        /* CPV01 */
    F64S cpv02;        /* CPV02 */
    F64 max;           /* 最大値 */
    F64 min;           /* 最小値 */
    F64S p01;          /* P01 */
    I32 rtnReadIn;     /* IN 端子入力リターンコード */
    U32 expFlag;       /* 浮動小数演算エラー検出フラグ */
    I32 rtnCode;        /* リターンコード */

    /*
     * アラームを検出する場合は、
     * 処理の最初で対象アラーム (ERRC) をクリアしておきます。
     */
    rtnCode = UcaAlrmClear(bc, UCAMASK_ALRM_ERRC);

    /* ====== 入力処理 ====== */
    rtnReadIn = UcaRWReadIn(bc, NOOPTION);           /* IN 端子から RVへ読み込み */
    rtnCode = UcaRWRead(bc, 1, NOOPTION);             /* Q01 端子から RV01へ読み込み */
    rtnCode = UcaRWRead(bc, 2, NOOPTION);             /* Q02 端子から RV02へ読み込み */
}
```

(続く)

algo_calc は、最初に演算異常を示す ERRC アラームを UcaAlarmClear でクリアしています。 algo_calc は、IN 端子、Q01 端子、Q02 端子からデータを入力しています。データ入力の部分は、前節のユーザカスタム _SMPL_ALGO_IN と同様です。

algo_calc はデータアイテム RV, RV01, RV02, P01 からデータを取得し、演算を実行します。

(続き)

```
/* ===== 演算処理 ===== */
/* 入力したデータをデータアイテムから変数に読み込み */
rtnCode = UcaDataGetRv(bc, &rv, NOOPTION); /* RV */
rtnCode = UcaDataGetRvn(bc, &rv01, 1, 1, NOOPTION); /* RV01 */
rtnCode = UcaDataGetRvn(bc, &rv02, 2, 1, NOOPTION); /* RV02 */

/* データアイテム P01 の倍率を読み込み */
rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION); /* P01 */

/*
 * 演算を実行 : CPV = P01 * (RV + RV01 + RV02)
 *
 * 浮動小数演算エラー（オーバーフロー等）を検出します
 */
rtnCode = UcaFpuExpClear(bc); /* 演算エラーフラグをクリア */

cpv.value = p01.value * (rv.value + rv01.value + rv02.value);

rtnCode = UcaFpuExpCheck(bc, &expFlag);
if (expFlag == 0) {
    /* 演算エラーなし */
    cpv.status = 0; /* データステータス正常 */
} else {
    /* 演算エラー検出 */
    cpv.status = UCAMASK_DSTS_BAD; /* データステータス BAD */
}

/* ERRC アラーム発生を設定 */
rtnCode = UcaAlrmSet(bc, UCAMASK_ALRM_ERRC);

/* データアイテム ERRC,ERRL にエラーコードとエラー位置を設定 */
rtnCode = UcaDataStoreErrorNumber(bc, expFlag,
                                  ERRL_ALGO_CALCCPV, NOOPTION);
}
```

(続く)

プログラムの以下の部分で演算エラーを検出しています。演算の前で UcaFpuExpClear を呼び出し演算エラーフラグをクリアし、演算のあとで UcaFpuExpCheck により演算エラーフラグを検査しています。

```
.....
rtnCode = UcaFpuExpClear(bc); /* 演算エラーフラグをクリア */
cpv.value = p01.value * (rv.value + rv01.value + rv02.value);
rtnCode = UcaFpuExpCheck(bc, &expFlag);
.....
```

UcaFpcExpCheck の引数 expFlag には、次表の値が返されます。各演算エラーごとに 1 ビットが割り当てられています。UcaFpuExpClear を呼び出してから UcaFpcExpCheck を呼び出すまでの間に複数の種類の演算エラーが発生すると、expFlag にはそれぞれの演算エラーの値を OR した値が返ります。

表 UcaFpuExpCheckが返す演算エラー

意味	ラベル	値（10進）	値（16進）
正常（例外発生なし）	なし	0	0x00000000
オーバフロー	UCAERR_FPU_OVERFLOW	4	0x00000100
ゼロ割り算	UCAERR_FPU_ZERODIV	8	0x00001000
無効演算	UCAERR_FPU_INVALID	16	0x00010000
演算フラグクリア抜け	UCAERR_FPU_EXNOCLR	32	0x00100000

注：ラベルは、システム定義インクルードファイル libucadef.h で定義されています。

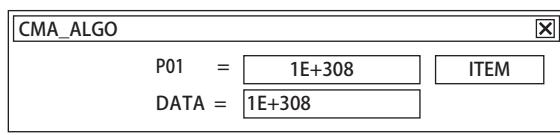
実際にオーバフローを起こしてみます。このプログラムの演算結果は以下の式で計算します。

$$CPV = P01 \times (IN \text{ 端子入力データ} + Q01 \text{ 端子入力データ} + Q02 \text{ 端子入力データ})$$

データアイテム P01 に、オーバフローが発生するような大きな値を入力してみます。次の値を確認してください。

CMA_ALGO の P01 1.0
 MLD_A00 の MV 10.0
 MLD_A01 の MV 20.0
 MLD_A02 の MV 30.0

この結果、CMA_ALGO の CPV は $1.0 \times (10.0 + 20.0 + 30.0) = 60.0$ になっています。
 CMA_ALGO のチューニングウィンドウを表示し、データアイテム P01 に「1E+308」を入力します（1E+308 は、「 1×10 の 308 乗」の意味です）。



050210J.ai

図 P01への入力

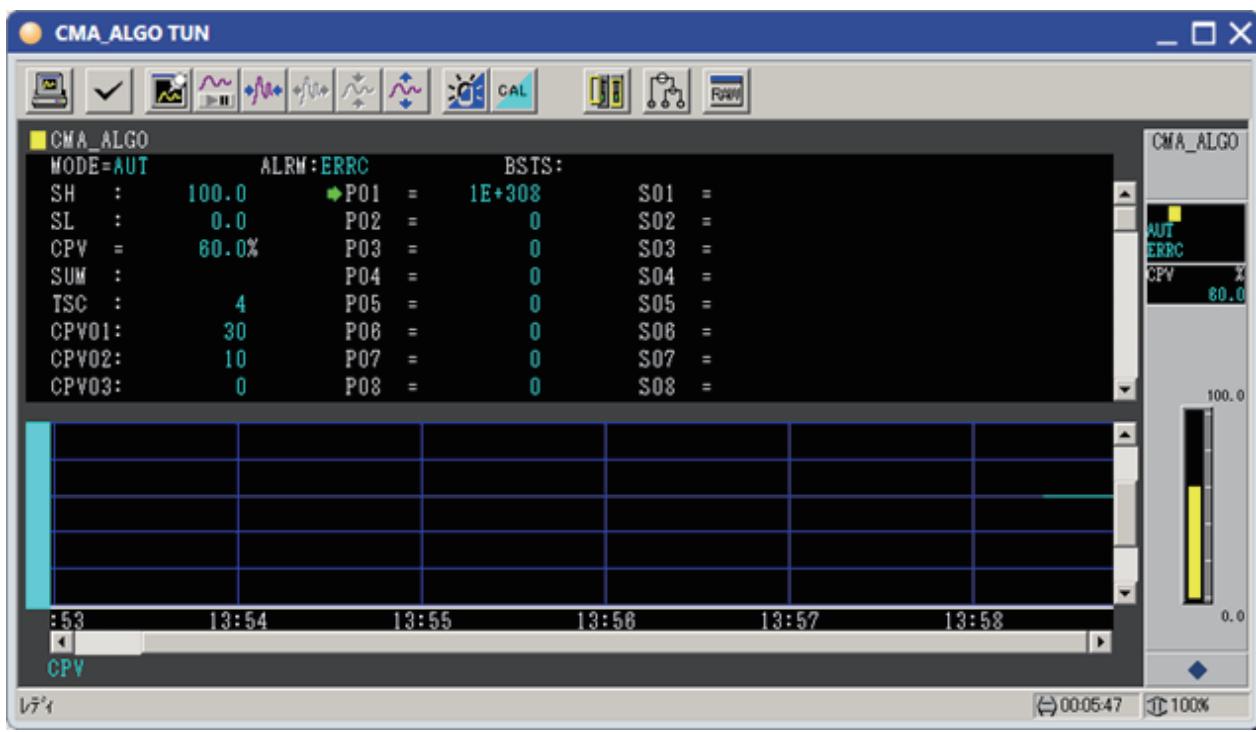


図 CMA_ALGOのチューニングウィンドウ

データアイテム P01 に 1E+308 が設定されています。データアイテム CPV の値は 60.0 のまま変化していません。これは、UcaFpuExpCheck が演算エラーを検出するとプログラムの以下の部分で CPV のデータステータスに BAD を設定しているからです。引数に指定する変数 cpv のデータステータスが BAD の場合、UcaRWSetCpv はデータアイテム CPV のデータを前回値（ここでは 60.0）のまま保持します。

```
.....
/* 演算エラー検出 */
cpv.status = UCAMASK_DSTS_BAD; /* データステータス BAD */
.....
```

参照 本プログラムの UcaRWSetCpv 呼び出し処理の詳細については、以下を参照してください。
[「5.2.3 出力端子からのデータ出力」](#)

次に、CMA_ALGO のデータアイテム ERRC と ERRL を確認してください。データアイテム ERRC と ERRL は、プログラマが自身のプログラムの障害解析をするための情報を残すためのデータアイテムです。データアイテム ERRC と ERRL へのデータ設定をする／しないは、ユーザカスタムブロックの動作には何も影響は与えません。

このプログラムでは、データアイテム ERRC には UcaFpcExpCheck が引数 expFlg に返した値を表示しています。ERRC が 0x4 なので、オーバフローが検出されたことがわかります。

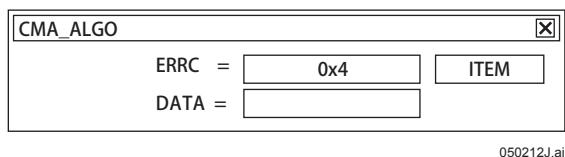


図 ERRC

データアイテム ERRL には 50 が設定されています。ERRL が 50 なので、この演算エラーは CPV の計算で検出されたことがわかります（このプログラムは ERRL を 1箇所でのみ設定していますが、複数な演算をする場合にいくつかの箇所で ERRL に異なる値を設定しておけば、エラーの箇所を特定することができます）。

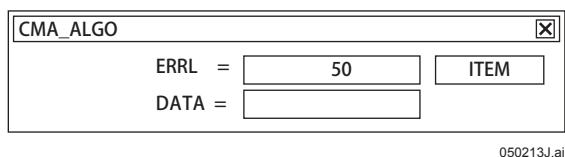
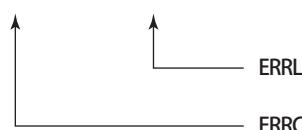


図 ERRL

データアイテム ERRC と ERRL は、プログラムの次の部分で設定しています。

```
.....
/* データアイテムERRC,ERRLにエラーコードとエラー位置を設定 */
rtnCode = UcaDataStoreErrorNumber(bc, expFlag, ERRL_ALGO_CALCCPV, NOOPTION);
.....
```



データアイテム ERRL に指定しているラベル ERRL_ALGO_CALCCPV は、プログラムの先頭の方で #define 定義されています。

```
.....
/* データアイテム ERRL に設定するエラー発生位置 */
#define ERRL_ALGO_CALCCPV      50      /* CPV 計算処理 */
.....
```

5.2.2 演算異常ERRCアラームの発生と復帰

演算異常を知らせるERRCアラームの発生と復帰について説明します。

汎用演算形ユーザカスタムブロック CMA_ALGO のチューニングウィンドウを確認してください。ERRC アラームが発生しています。

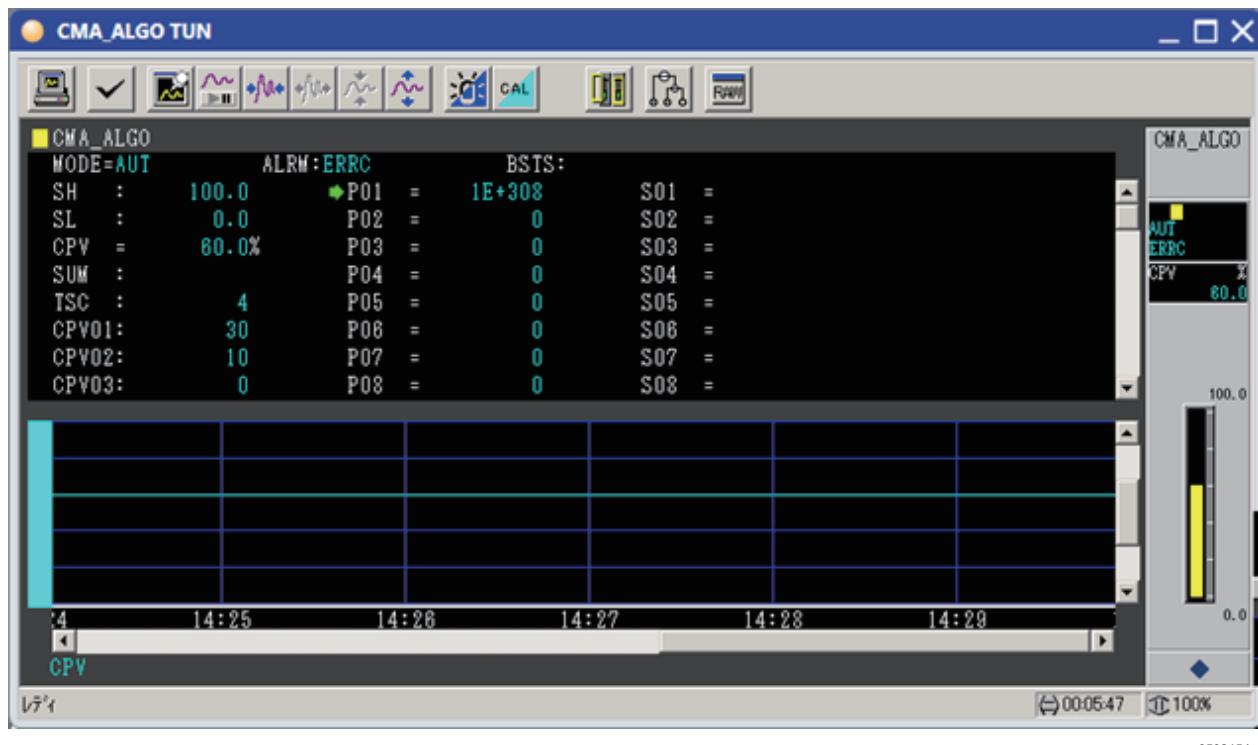


図 CMA_ALGOのチューニングウィンドウ

050215J.ai

この状態でデータアイテム P01 に 1 を設定すると、ERRC アラームが復帰します。プロセスアラームウィンドウで ERRC アラームの発生と復帰を確認してください。



図 アラームの発生と復帰

050216J.ai

データアイテム P01 に「1E+308」と「1」を交互に設定して、ERRC アラームを発生・復帰してください。CMA_ALGO の実効スキャン周期は 4 秒にしてありますので、P01 を変更してから ERRC アラームの状態が変化するまでに最大 4 秒かかります。

アラーム発生の設定は UcaAlrmSet、アラーム復帰の設定は UcaAlrmClear で行います。これらの関数には、libucadef.h に定義されているアラームのラベルを指定します。たとえば ERRC アラームを発生・復帰するには、ラベル UCAMASK_ALRM_ERRC を指定します。

参照

libucadef.h に定義されているアラームのラベルの詳細については、以下を参照してください。

[「4.3.3 ユーザ定義インクルードファイル usrstatus.h」](#)

UcaAlrmSet と UcaAlrmClear による設定は、各アラームステータスごとに後優先となります。つまり、ユーザ定義関数（たとえば機能ブロック定期処理なら UcaBlockPeriodical）の先頭から return するまでの間で、最後に呼び出された UcaAlrmSet または UcaAlrmClear による設定が有効となります。

実際にアラームの発生・復帰メッセージを出力するのはユーザカスタムブロック実行管理部です。ユーザ定義関数がリターンしたあとに、ユーザカスタムブロック実行管理部は最後に呼び出された UcaAlrmSet または UcaAlrmClear の設定に従い、アラーム発生・復帰処理を行います。

ユーザカスタムブロック実行管理部の動作を以下に示します。

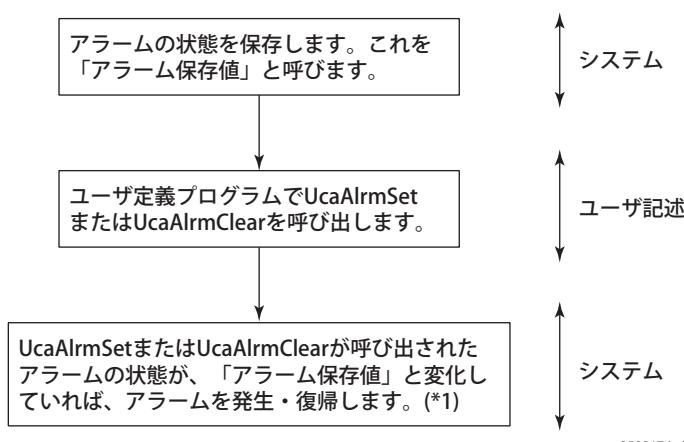


図 ユーザカスタムブロック実行管理部のアラーム発生・復帰処理

- *1：アラーム保存値が、「復帰」状態で、UcaAlrmSet により「発生」が設定されていれば、アラームを発生しプロセスアラームメッセージを出力します。アラーム保存値が「発生」状態で、UcaAlrmClear により「復帰」が設定されていれば、アラームを復帰しプロセスアラーム復帰メッセージを出力します。
- UcaAlrmSet が呼び出されていてもアラーム保存値が「発生」状態であれば、プロセスアラームメッセージは出力されません。同様に UcaAlrmClear が呼び出されていてもアラーム保存値が「復帰」状態であれば、プロセスアラーム復帰メッセージは出力されません。

重要

- ユーザカスタムブロック実行管理部がアラーム処理を行うのは、機能ブロック定期処理と機能ブロックワンショット起動処理の前後だけです。
- ユーザ定義関数の戻り値が UCAERR_STOPME 以外 (SUCCEED または UCAERR_NOPROC) であれば、ユーザカスタムブロック実行管理部はアラーム発生・復帰処理を行います。

アラーム検出のプログラミングについて説明します。次のプログラムは誤りです。一つのプログラムの2箇所でERRCアラームの検出処理をしていますが、いつも2回目(下)の検出結果が優先されますので1回目(上)の検出は無意味になります。たとえば、1回目(上)の検査でアラーム発生条件が成立し UcaAlrmSet でERRCアラーム発生を設定しても2回目(下)の検査でアラーム発生条件が成立しないため、UcaAlrmClear でERRCアラーム復帰を設定すれば2回目の復帰が優先されるのでERRCアラームは発生しません。

```
...
/* 悪い例：アラーム発生・復帰の誤り例 */

if (アラームを発生する条件が成立) {
    rtnCode = UcaAlrmSet(bc, UCAMASK_ALRM_ERRC);
} else {
    rtnCode = UcaAlrmClear(bc, UCAMASK_ALRM_ERRC);
}
...

...
if (アラームを発生する条件が成立) {
    rtnCode = UcaAlrmSet(bc, UCAMASK_ALRM_ERRC);
} else {
    rtnCode = UcaAlrmClear(bc, UCAMASK_ALRM_ERRC);
}
...
```

アラームの検出は、次のプログラムのように記述します。処理の最初で UcaAlrmClear によりERRCアラームの復帰を設定しておきます。そして、アラーム検出処理ではアラームを発生したい場合のみ UcaAlrmSet を呼び出します。

```
/* 正解：アラーム発生・復帰の記述例 */

/* 処理の最初でアラームの復帰を設定します */
rtnCode = UcaAlrmClear(bc, UCAMASK_ALRM_ERRC);

if (アラームを発生する条件が成立) {
    rtnCode = UcaAlrmSet(bc, UCAMASK_ALRM_ERRC);
}
...

...
if (アラームを発生する条件が成立) {
    rtnCode = UcaAlrmSet(bc, UCAMASK_ALRM_ERRC);
}
...
```

このように記述すれば、アラーム発生を検出したときには後から呼び出された UcaAlrmSet が有効となり、ERRC アラームが発生します。アラーム発生を検出しなければ、最初に呼び出した UcaAlrmClear が有効となり、ERRC アラームが復帰します。

ユーザカスタムアルゴリズム _SMPL_ALGO の algo.c では、演算処理本体を行う algo_calc 関数の最初で UcaAlrmClear を呼び出しています。

```
...
/*
 * アラームを検出する場合は、
 * 処理の最初で対象アラーム(ERRC)をクリアしておきます。
 */
rtnCode = UcaAlrmClear(bc, UCAMASK_ALRM_ERRC); ←———— ERRC復帰を設定
...
```

また、CPVへの設定値を計算したあとで演算エラーを検出すると、UcaAlrmSet を呼び出しています。

```
.....
rtnCode = UcaFpuExpClear(bc); /* 演算エラーフラグをクリア */
cpv.value = p01.value * (rv.value + rv01.value + rv02.value);
rtnCode = UcaFpuExpCheck(bc, &expFlag);
if (expFlag == 0) {
    /* 演算エラーなし */
    cpv.status = 0; /* データステータス正常 */
} else {
    /* 演算エラー検出 */
    cpv.status = UCAMASK_DSTS_BAD; /* データステータスBAD */
}

/* ERRCアラーム発生を設定 */
rtnCode = UcaAlrmSet(bc, UCAMASK_ALRM_ERRC); ←———— ERRC発生を設定

/* データアイテムERRC,ERRLにエラーコードとエラー位置を設定 */
rtnCode = UcaDataStoreErrorNumber(bc, expFlag, ERRL_ALGO_CALCCPV, NOOPTION);
}
.....
```

このプログラムは ERRC アラームの検出を 1箇所でのみ行っていますが、本書で説明しているサンプルはいつも「最初で UcaAlrmClear を呼び出し、アラームの発生を検出すると UcaAlrmSet を呼び出す」という記述でプログラミングしています。

5.2.3 出力端子からのデータ出力

出力端子からのデータ出力について説明します。

algo_calc は、UcsRWSetCpv により演算結果をデータアイテム CPV に格納します。汎用演算形ユーザカスタムブロック CMA_ALGO は、ビルダ定義項目「演算入力値異常検出」に「全検出形」を指定しています。

参照 演算結果の CPV への格納の詳細については、以下を参照してください。

「5.1 多入力 CSTM-A」の「■ UcaRWSetCpv による CPV のデータステータス作成」

また、algo_calc は 3 つの入力の最大値をデータアイテム CPV01 に格納し、3 つの入力の最小値をデータアイテム CPV02 に格納します。

(続き)

```
/* 演算結果をデータアイテム CPV に出力 */
/* 引数 rtnReadIn は、IN 端子読み込みのリターンコード */
rtnCode = UcaRWSetCpv(bc, &cpv, rtnReadIn, NOOPTION);

/* CPV01 に RV,RV01,RV02 の最大値を保存 */
max = rv.value;
if (max < rv01.value) {
    max = rv01.value;
}
if (max < rv02.value) {
    max = rv02.value;
}
cpv01.value = max;
cpv01.status = 0;
rtnCode = UcaDataStoreCpvn(bc, &cpv01, 1, 1, NOOPTION);

/* CPV02 に RV,RV01,RV02 の最小値を保存 */
min = rv.value;
if (min > rv01.value) {
    min = rv01.value;
}
if (min > rv02.value) {
    min = rv02.value;
}
cpv02.value = min;
cpv02.status = 0;
rtnCode = UcaDataStoreCpvn(bc, &cpv02, 2, 1, NOOPTION);
```

(続く)

最後に、UcaRWWriteCpvToSOutSub でデータアイテム CPV に格納された演算結果を OUT 端子から出力します。また、UcaRWWrite により CPV01 のデータを J01 端子から出力し、CPV02 のデータを J02 端子から出力します。

(続き)

```
/* ===== 出力処理 ===== */
rtnCode = UcaRWWriteCpvToSOutSub(bc, NOOPTION);           /* CPV を OUT 端子から出力 */

rtnCode = UcaRWWrite(bc, 1, NOOPTION);                      /* CPV01 を J01 端子から出力 */
rtnCode = UcaRWWrite(bc, 2, NOOPTION);                      /* CPV02 を J02 端子から出力 */

return SUCCEED;
}
```

UcaRWWriteCpvToSOutSub は SUB 端子への出力も行いますが、出力内容はビルダ定義項目「補助出力」の指定で決まります。



図 補助出力（SUB端子）の指定

050220J.ai

5.2.4 チューニングパラメータより弱い初期値の設定方法

ユーザカスタムアルゴリズム_SMPL_ALGOでは、データアイテムP01をCPVの計算に使用する倍率として使用しています。このため、機能ブロック初期化処理でデータアイテムP01を1.0に初期化しています。機能ブロック初期化処理で単純に1.0を設定するだけでは、HISなどからP01に倍率を設定したあとに機能ブロック初期化処理を実行すると(APCS初期化スタートやブロックモードO/Sから復帰などで機能ブロック初期化処理が実行されます)、P01は再び1.0に戻ってしまいます。

参照 機能ブロック初期化処理の詳細については、以下を参照してください。
[「3.2 機能ブロック初期化処理」](#)

ここでは、一回だけ自ブロックのデータアイテム(例ではP01)を初期化し、以後、外部からデータが設定された場合には、設定された(新しい)データを有効にするプログラムの記述方法を説明します。この方法で記述すれば、データを設定したあとにチューニングパラメータセーブで保存したデータが機能ブロック初期化処理で設定する初期値より強くなります。

チューニングパラメータより弱い初期化を実現するために、初期値を設定済みか判定するためのフラグとしてデータアイテムI08を使用しています。

- ユーザカスタムブロックを定義したときのI08の初期値は0です。
- I08が0ならP01を1.0に初期化します。同時にI08を-1にし初期化済みとします。
- I08が0以外なら何もしません。
- I08自身もチューニングパラメータセーブの対象となります。つまり、チューニングパラメータセーブしたデータを復元するとI08のデータも復元されます。

algo.c を UcaBlockInit で検索し、機能ブロック初期化処理を確認してください。データアイテム I08 が 0 の場合に限りデータアイテム P01 を初期化し、データアイテム I08 にも「初期化済み」を示す
– 1 を設定しています。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockInit
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME   処理続行不能
*
* description:  機能ブロック初期化処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockInit(
    UcaBlockContext bc, /* (IN/OUT): ブロックコンテキスト */
    I32 reason        /* (IN): 呼び出し理由 */
)
{
    F64S p01; /* P01 */
    I32 i08; /* I08 */
    I32 rtnCode; /* リターンコード */

    /* I08 が 0なら P01 を初期化します */
    rtnCode = UcaDataGetIn(bc, &i08, 8, 1, NOOPTION);
    if (i08 == 0) {
        /* 倍率 P01 の初期値に 1.0 を設定 */
        p01.value = 1.0;
        p01.status = 0;
        rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);

        /* i08 に初期化済み (-1) を設定 */
        i08 = -1;
        rtnCode = UcaDataStoreIn(bc, &i08, 8, 1, NOOPTION);
    }

    return SUCCEED;
}
```

このプログラムでは、データ設定時特殊処理で、初期化済みフラグとして使用しているデータアイテム I08へのデータ設定を禁止しています。

```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockDset
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME   処理続行不能
*
* description:  機能ブロックデータ設定時特殊処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockDset(
    UcaBlockContext bc,           /* (IN/OUT): ブロックコンテキスト */
    DITMN itemName,              /* (IN): データアイテム名 */
    UcaUnivType *data,           /* (IN): 設定データ */
    UcaDataOrStatus dataOrStatus, /* (IN): 設定種別 */
    UcaPreOrPost preOrPost,       /* (IN): 呼び出し種別 */
    BOOL *result                 /* (OUT): 設定可否 */
)
{
    if (strncmp(itemName, "I08", sizeof(DITMN)) == 0) {
        /* I08はデータ設定を禁止 */
        *result = UCADSET_DENY;
    } else {
        *result = UCADSET_ALLOW;
    }

    return SUCCEED;
}
```

6. 連続制御形ユーザカスタムブロックのプログラミング

連続制御形ユーザカスタムブロック (CSTM-C) は、連続制御を行うためのユーザカスタムブロックです。CSTM-CはFCSから入力したデータに対し制御演算を行い、演算の結果をFCSの連続制御ブロックの設定値として出力するセットポイント制御に使用します。また、入力したデータより決定した測定値 (PV) に対し、入力上限／下限アラーム (HIとLO) や入力上上限／下下限アラーム (HHとLL) を検出することができます。

■ CSTM-Cのデータアイテムとプログラミング

CSTM-C は標準ブロック PID 調節ブロックの入出力端子とデータアイテムに加え、32 の入力端子、16 の出力端子などを持ちます。CSTM-C の構成を以下に示します。

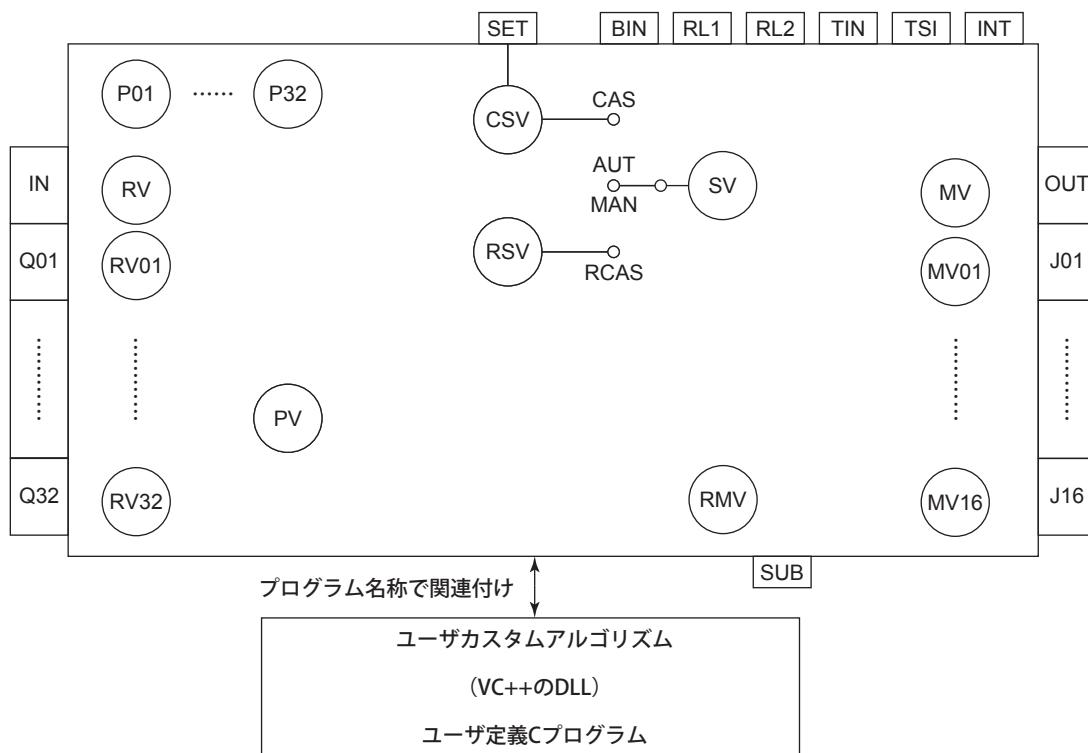


図 CSTM-Cの構成

060001J.ai

CSTM-C は PID 調節ブロックと同じブロックモードを持ちます。ブロックモードを切り換えることにより、自動 (AUT)、手動 (MAN) やカスケード (CAS) などの動作を CSTM-C で実現することができます。また、CSTM-C では制御演算用のユーザカスタムアルゴリズム作成用ライブラリを呼び出すことにより、入出力補償やリセットリミッタなどを使った制御動作を実現することができます。

PID 演算のユーザカスタムアルゴリズム作成用ライブラリを使用すれば、PID 調節ブロックと同じ動作をする CSTM-C を実現できます。PID 調節ブロックと同じ動作をするユーザカスタムアルゴリズムのサンプルが用意されていますので、このサンプルプログラムを修正することにより、PID 調節ブロックの動作を改造した CSTM-C を作成することが可能です。

この章では、まず 3 つの入力の合計をデータアイテム PV に表示するユーザカスタムアルゴリズムを説明します。このプログラムは、決定した測定値 (PV) に対し入力上限／下限アラーム (HI と LO) や入力上上限／下下限アラーム (HH と LL) を検出します。また、入力データの計算結果を測定値 PV に格納するときに、PV のデータステータスがどのように作成されるかについても説明します。この CSTM-C は、ブロックモードを AUT と O/S に限定します。また、ユーザ定義アラームの発生・復帰のプログラム例を示します。次に、多入力、多出力の CSTM-C のサンプルをもとに、ブロックモードに従い自動 (AUT)、手動 (MAN) やカスケード (CAS) のなどの動作をするユーザカスタムアルゴリズムをどのようにプログラミングするかを説明します。

さらに、入出力補償やリセットリミッタなどを使用したユーザカスタムアルゴリズム作成用ライブラリの使い方を説明します。最後に、PID 調節ブロックと同じ動作をする CSTM-C について説明します。

この章では、データ入力と出力に入出力端子を使用したプログラミングを説明します。タグ名を指定したデータの入出力は、入出力端子用のユーザカスタムアルゴリズム作成用ライブラリを呼び出している部分をタグ名を指定するユーザカスタムアルゴリズム作成用ライブラリの呼び出しに置きかえることで実現できます。

参照

タグ名を指定したプログラムのデータ入力と出力の記述は、CSTM-A と CSTM-C で同様です。詳細については、以下を参照してください。

「[7. タグ名を指定したデータ入力](#)」

6.1 多入力CSTM-C（ブロックモードはAUTとO/Sに限定）

連続制御形カスタムブロック（CSTM-C）の入力処理と測定値PVへのデータ設定について説明します。ここでは3つの入力データをIN端子、Q01端子、Q02端子から読み込み、3つのデータの合計をデータアイテムPVに格納するユーザカスタムアルゴリズムを説明します。このCSTM-Cはユーザカスタムアルゴリズムのプログラムにより、ブロックモードをAUTとO/Sに限定します。また、PVのデータ（3つの入力の合計）を出力端子J01から出力し、3つの入力データの平均を出力端子J02から出力します。

サンプルプログラムが用意してありますので、動かしてみます。サンプルソリューション _SMPL_ALGO_IN の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは1章の準備作業により以下の作業フォルダにコピーされています。

<ドライブ名>:\UcaWork\UcaSamples\SMPL_ALRM

Visual Studio を起動し、_SMPL_ALRM\SMPL_ALRM.sln を開きます。[ビルド] メニューの [リビルド] で _SMPL_ALRM の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。

次に、サンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイル ALRM.txt と FALRM.txt が CENTUM VP インストール先の以下にあります。ALRM.txt を FCS0121 (APCS) に、FALRM.txt を FCS0101 (FCS) にそれぞれインポートします。

<CENTUM VP インストール先> \UcaEnv\Sample\LearnUca\drawings\ALRM.txt と FALRM.txt

FCS0101 (FCS) の制御ドローイングの DR0060 (空いている制御ドローイングならどれでも構いません) を指定して制御ドローイングビルダを起動します。[ファイル] メニューの [外部ファイル] – [インポート] を指定します。インポートダイアログで上記の FALRM.txt を指定し、[開く] ボタンをクリックすると3つの手動操作ブロックを定義した制御ドローイングが取り込まれます。[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

同じ手順で、FCS0121 (APCS) の制御ドローイングの DR0060 (空いている制御ドローイングならどれでも構いません) に ALRM.txt をインポートします。次図の制御ドローイングが取り込まれますので、[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

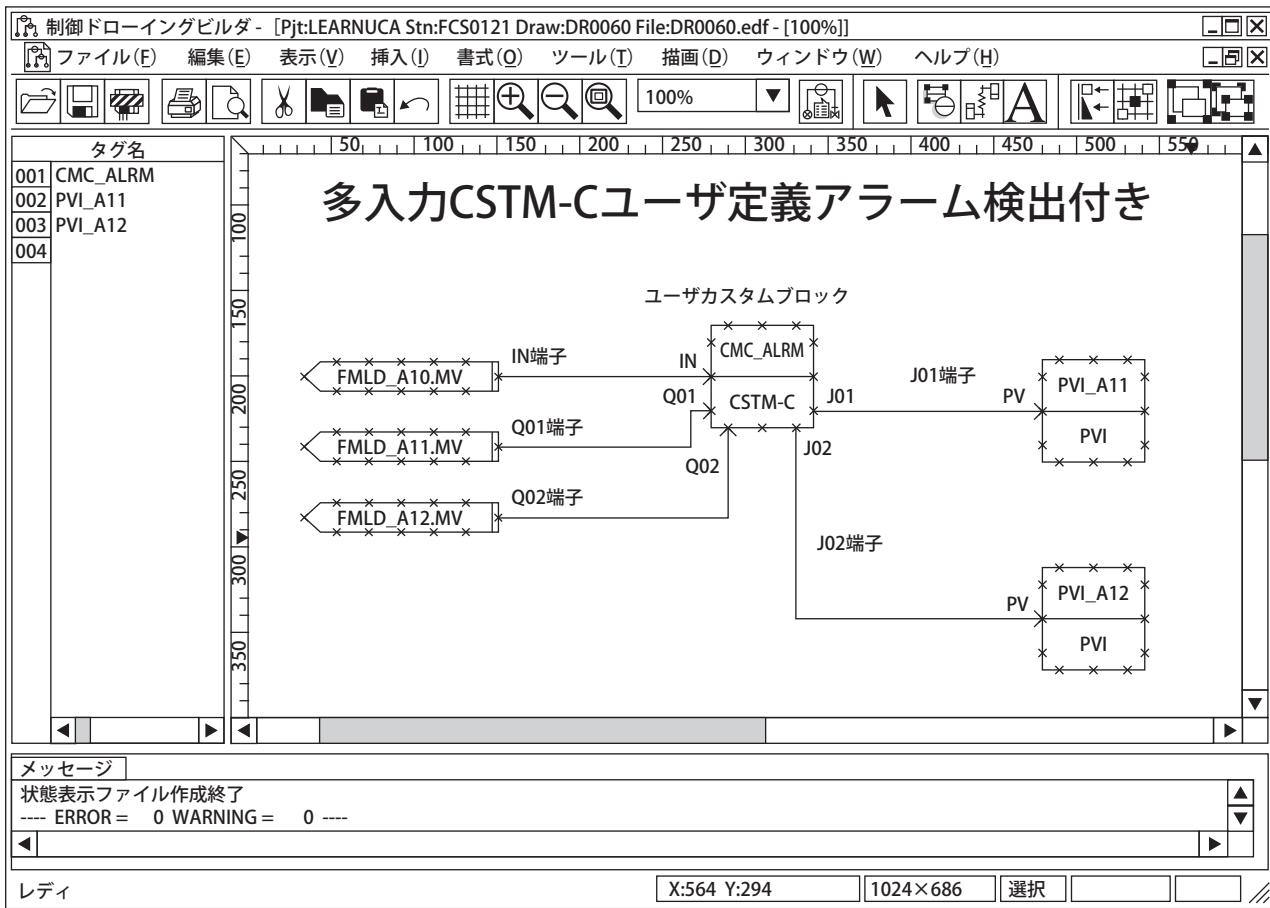


図 制御ドローイングのインポート

このプログラムは、FCS (FCS0101) の3つの手動操作ブロックが MV に出力するデータを、APCS (FCS0121) のユーザカスタムブロック CMC_ALRM が IN 端子、Q01 端子、Q02 端子から入力しています。前図の APCS の制御ドローイングでは、FCS のデータは「AREAOUT」で結合しています。

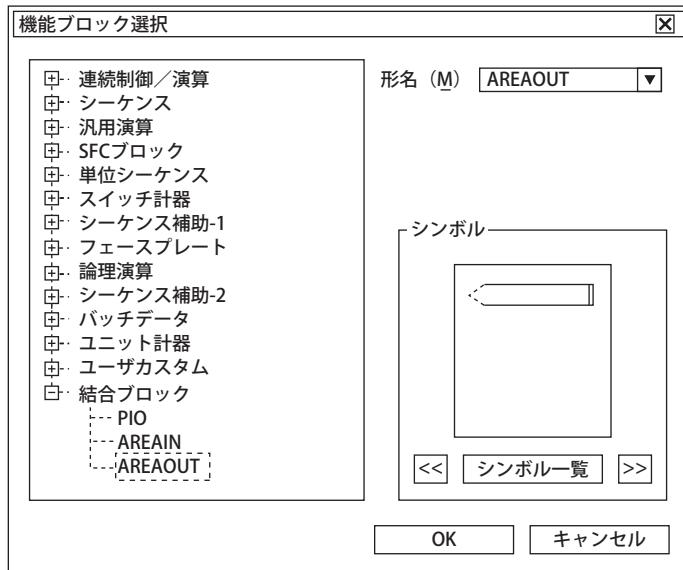


図 データ結合

STM-C がデータを入力する端子の数が多い場合には、「AREAOUT」の図を描画しないで、機能ブロック詳細ビルダで入力するデータを直接記述することもできます。

次図は、連続制御形ユーザカスタムブロック CMC_ALRM の機能ブロック詳細ビルダです。「AREAOUT」を使って制御ドローイングに結線を描画すると、システムは次図のように結合するデータを設定します (>FMLD_A10.MV、>FMLD_A11.MV と >FMLD_A12.MV の指定の部分)。「AREAOUT」で描画しないで、この部分に「>FMLD_A10.MV」などを直接入力することも可能です。先頭の「>」は、ステーション外のデータであることを示すために必要です(直接手入力する場合には、「>」とタグ名の間には空白を入れないで詰めて入力してください)。



060103J.ai

図 機能ブロック詳細ビルダでの入力

ユーザカスタムアルゴリズムからの他ステーションデータアクセスは、制御ステーション間結合を利用します。制御ステーション間結合とは、自ステーション（APCS）の機能ブロックと他ステーション（FCS）の機能ブロックとの間で、データ結合または端子間結合を行う入出力結合方式です。

制御ドローイングビルダで他の制御ステーションの機能ブロックに対する入出力結合情報（「AREAOUT」、または先頭に「>」を指定したデータ結合）を指定すれば、システムによりステーション間結合ブロック（ADL）が生成され、他ステーションの機能ブロックとのデータのやり取りが ADL ブロックを介して行われます。

参照 制御ステーション間結合およびステーション間結合ブロックの詳細については、以下を参照してください。

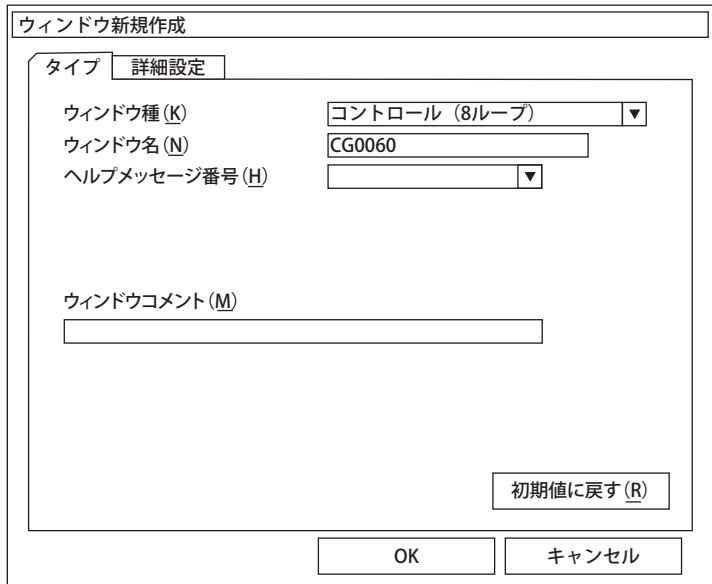
[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「2.4 制御ステーション間結合」](#)

[機能ブロックリファレンス Vol.2 \(IM 33J15A31-01JA\) 「1.46 ステーション間結合ブロック \(ADL\)」](#)

動作を確認するために、コントロール（8ループ）のウィンドウが用意してありますので、HIS0164 の CG0060 に取り込んでおきます。テキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥graphics¥ALRM_CG.sva

システムビューで HIS0164 の WINDOW にウィンドウ種「コントロール（8ループ）」、ウィンドウ名「CG0060」を作成します（ウィンドウ名は自由に命名してください）。

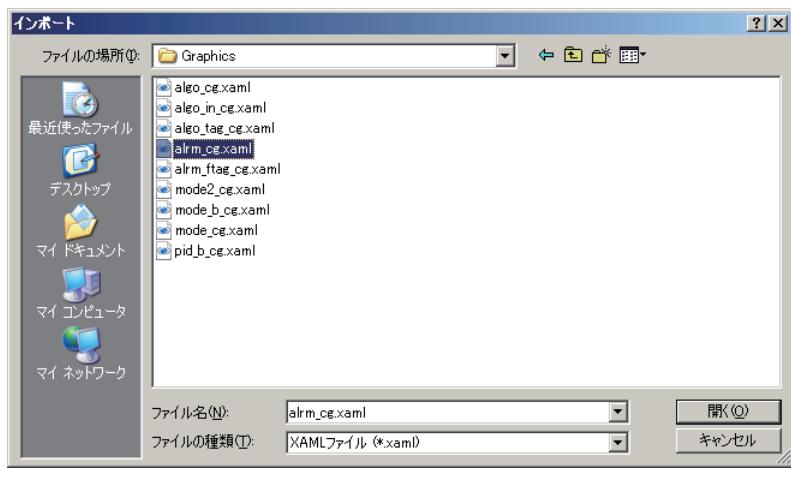


060104J.ai

図 コントロールウィンドウの作成

HIS0164 の WINDOW に CG0060 ができますので、システムビューより CG0060 をダブルクリックします。

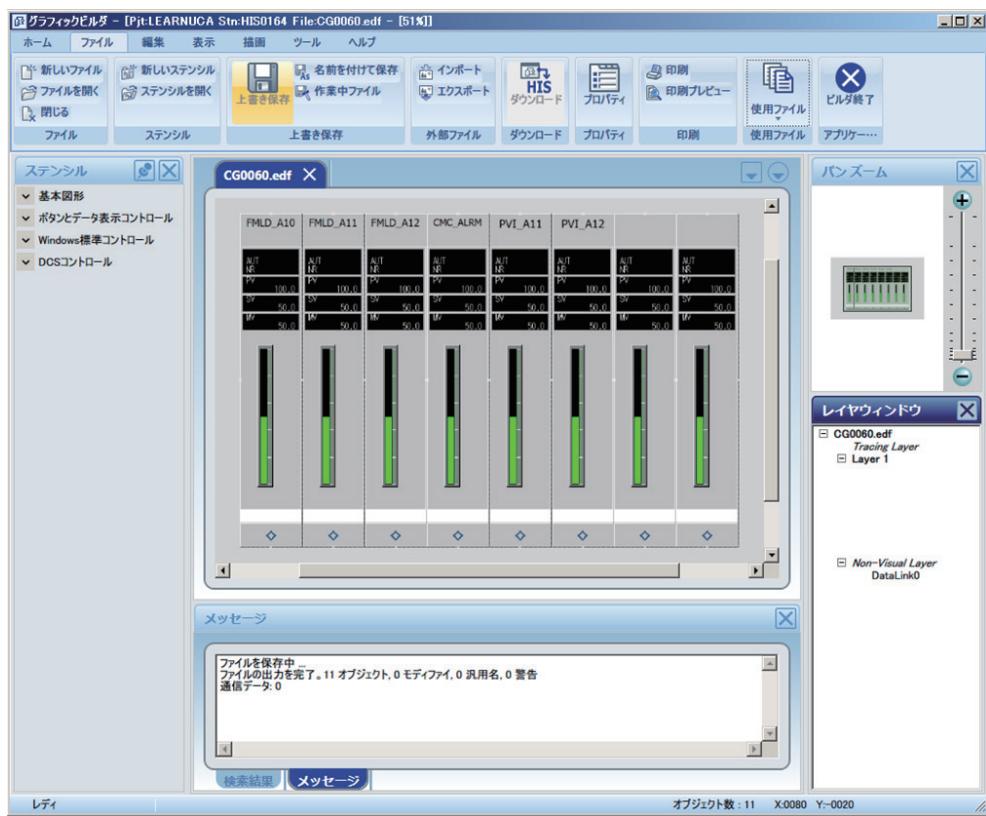
グラフィックビルダの [ファイル] – [外部ファイル] – [インポート] で以下のダイアログを呼び出し、alarm_cg.xaml を指定し、[開く] ボタンをクリックします。



060105J.ai

図 ファイルを開くダイアログ

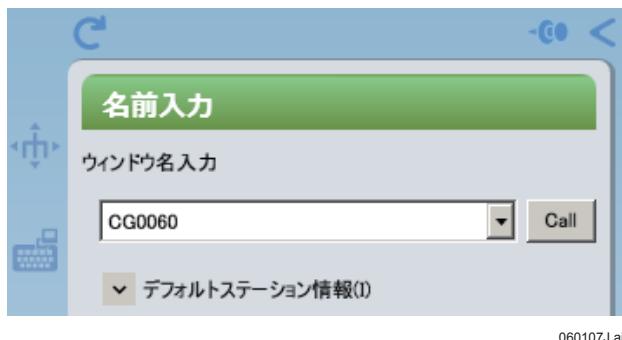
グラフィックビルダの [ファイル] – [上書き保存] でファイルを書き込みます。



060106J.ai

図 ファイルの保存

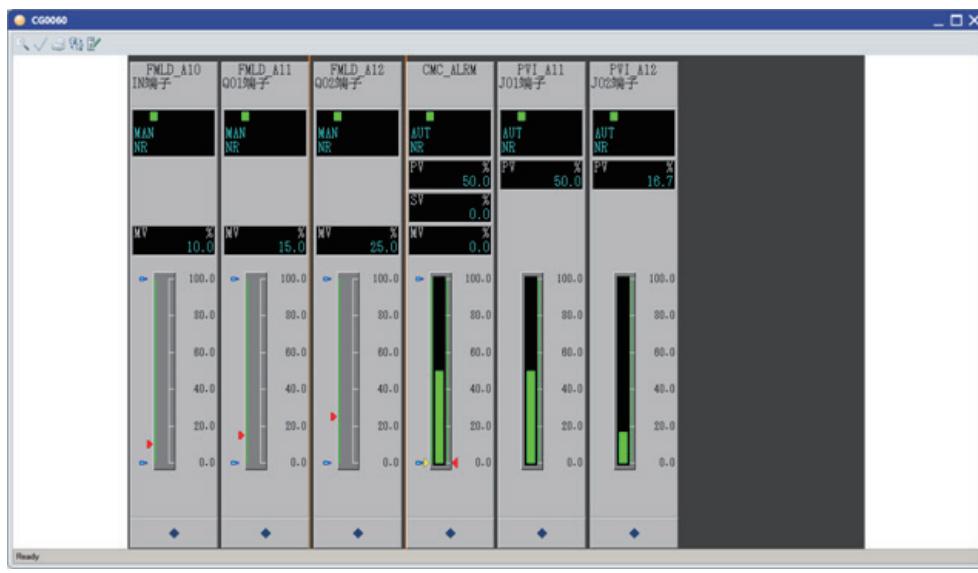
ここでプログラムを動かしてみます。まず、FCS0101 を指定してバーチャルテスト機能を起動します。起動が完了したら、FCS0121 (APCS) を指定してバーチャルテスト機能を起動します。FCS0101 と FCS0121 の両方の起動が完了したらウィンドウ CG0060 を表示します。



060107J.ai

図 ウィンドウの呼び出し

左 3 つの計器図は FCS の手動操作ブロック (FMLD_A10、FMLD_A11、FMLD_A12) です。その右には、FCS0121 (APCS) の 1 つの汎用演算形ユーザカスタムブロック (CMC_ALRM) と 2 つの指示ブロック (PVI_A11、PVI_A12) が並んでいます。CMC_ALRM のデータアイテム PV には、3 つの手動操作ブロックの MV 値の合計が表示されます。手動操作ブロックの MV 値を変更し、CMC_ALRM のデータアイテム PV が実効スキャン周期 (4 秒) ごとに 3 つの MV 値の合計を表示するのを確認してください。以下は、MV 値にそれぞれ 10.0、15.0、25.0 を指定し、その結果 CMC_ALRM のデータアイテム PV が 50.0 になっている状態です。



060108J.ai

図 CMC_ALRM の PV

CMC_ALRMは、J01端子からデータアイテムPV(3つの入力の合計)のデータを出力します。J02端子からは3つの入力の平均を出力します。J01端子、J02端子は、それぞれ指示ブロックPVI_A11、PVI_A12のPVと結合しています。手動操作ブロックのMV値を変更すると指示ブロックのPVが変化することを確認してください。

■ 多入力のサンプルプログラム

ユーザカスタムアルゴリズム _SMPL_ALRM の alrm.c について説明します。
UcaBlockPeriodical という名前で検索して機能ブロック定周期処理を見つけてください。

```
/*
* <<FNH>>*****
*
* Function name:      UcaBlockPeriodical
* Return value:       SUCCEED      正常終了
*                      UCAERR_NOPROC   処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description:        機能ブロック定周期処理
*
* >>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode;           /* リターンコード */

    /* 測定値作成処理本体 */
    rtnCode = alrm_calc(bc);

    return SUCCEED;
}
```

このプログラムは、測定値作成処理の関数 alrm_calc を呼び出しています。alrm_calc は、同じソースファイル(alrm.c)の下の方にあります。機能ブロック定周期処理のすぐあとに、機能ブロックワンショット起動処理 (UcaBlockOneshot) があるので確認してください。

```

/*
* <<FNH>>*****
*
* Function name:      UcaBlockOneshot
* Return value:       SUCCEED      正常終了
*                      UCAERR_NOPROC   処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description:        機能ブロックワンショット起動処理
*
* >>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc, /* (IN/OUT) : ブロックコンテキスト */
    I32 code,           /* (IN) : 種別コード */
    I32 parameter,      /* (IN) : パラメータ */
    BOOL *result         /* (OUT) : 実行結果 */
)
{
    I32 rtnCode;          /* リターンコード */

    /* 状態操作でなければ何もしない */
    if (code != UCAONESHOT_CODEOPRT) {
        return UCAERR_NOPROC;
    }

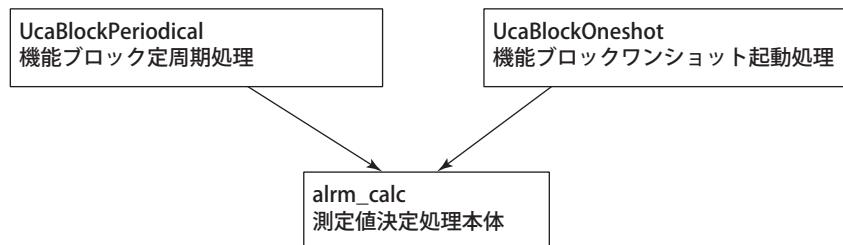
    /* 測定値作成処理本体 */
    rtnCode = alrm_calc(bc);

    *result = TRUE;

    return SUCCEED;
}

```

機能ブロックワンショット起動処理でも、alrm_calc を呼び出しています。このユーザカスタムアルゴリズムは、実効スキャン周期による機能ブロック定周期処理と機能ブロックワンショット起動処理の両方で alrm_calc 関数に記述してある同じ処理を行います。



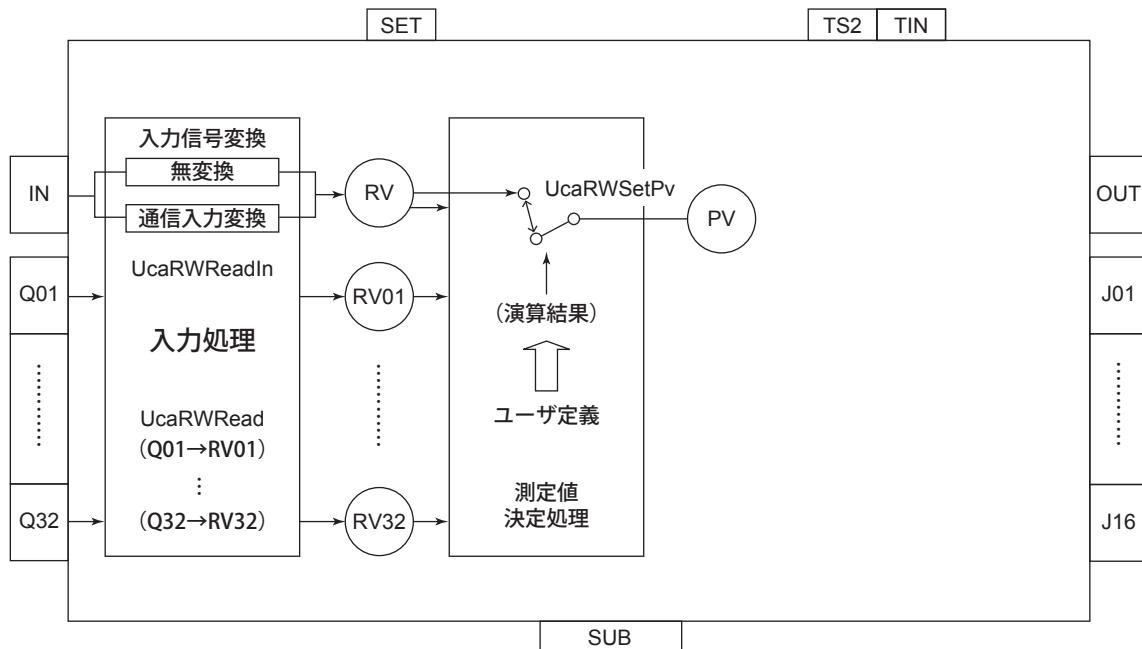
060109J.ai

図 alrm_calc

参照 機能ブロック定周期処理と機能ブロックワンショット起動処理の使い分けの詳細については、以下を参照してください。

[「3.5.1 ビルダ定義項目「起動タイミング」」](#)

alrm_calc のソースコードを説明する前に、連続制御形ユーザカスタムブロックの入力データの流れについて説明します。



060110J.ai

図 CSTM-Cの入力データの流れ

UcaRWReadIn は、IN 端子からデータを入力しデータアイテム RV に設定します。UcaRWReadIn の入力動作として、ビルダ定義項目「入力信号変換タイプ」により「無変換」または「通信入力変換」を指定することができます。UcaRWRead は入力端子 Q01～Q32 よりデータを読み込み、データアイテム RV01～RV32 に設定します。ユーザ定義の測定値 (PV) 作成処理は、データアイテム RV と RV01～RV32 に入力したデータに対し計算を行います。そして計算結果を UcaRWSetPv によりデータアイテム PV に設定します。

それでは、プログラムを確認します。ユーザカスタムアルゴリズム _SMPL_ALRM の alrm.c から alrm_calc という名前で検索し、以下の部分を見つけてください。

```
I32 alrm_calc(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnReadIn; /* IN 端子入力リターンコード */
    F64S pv; /* PV */
    F64S rv; /* RV */
    F64S rv01; /* RV01 */
    F64S rv02; /* RV02 */
    F64S p01; /* P01 */
    F64S p02; /* P02 */
    F64S ave; /* 平均 */
    I32 rtnCode; /* リターンコード */
}
```

(続く)

(続き)

```

/*
 * アラームを検出する場合は、
 * 処理の最初で対象アラームをクリアしておきます。
 */
rtnCode = UcaAlrmClear(bc, USR_ALRM_UHIG);
rtnCode = UcaAlrmClear(bc, USR_ALRM_ULOW);

/* ====== 入力処理 ======
rtnReadIn = UcaRWReadIn(bc, NOOPTION); /* IN 端子から RVへ読み込み */

rtnCode = UcaRWRead(bc, 1, NOOPTION); /* Q01 端子から RV01へ読み込み */
rtnCode = UcaRWRead(bc, 2, NOOPTION); /* Q02 端子から RV02へ読み込み */

/* ====== 演算処理 ======
/* 入力したデータをデータアイテムから変数に読み込み */
rtnCode = UcaDataGetRv(bc, &rv, NOOPTION); /* RV */
rtnCode = UcaDataGetRvn(bc, &rv01, 1, 1, NOOPTION); /* RV01 */
rtnCode = UcaDataGetRvn(bc, &rv02, 2, 1, NOOPTION); /* RV02 */

/*
 * 演算を実行: PV=RV+RV01+RV02
 * ave=RV,RV01,RV02 の平均
 *
 * (演算エラーの検出は省略しデータステータスは常に正常)
 */
/* 3入力の合計 */
pv.value = rv.value + rv01.value + rv02.value;
pv.status = 0;

/* 合計をデータアイテム PV に設定 */
/* 引数rtnReadInは、IN 端子読み込みのリターンコード */
rtnCode = UcaRWSSetPv(bc, &pv, rtnReadIn, NOOPTION);

```

(続く)

alrm_calc は、最初にユーザ定義アラーム UHIG と HLOW を UcaAlarmClear でクリアしています。

参照

このプログラムでは演算エラーの検出は省略しています。演算エラーを検出し演算異常 ERRC アラームを検出する方法の詳細については、以下を参照してください。

[「5.2.1 入力処理と UcaFpuExpCheck による演算エラーの検出」](#)

[「5.2.2 演算異常 ERRC アラームの発生と復帰」](#)

● IN端子からの入力処理

alrm_calc は、UcaRWReadIn で IN 端子からデータアイテム RV にデータを入力します。UcaRWReadIn は、CSTM-A のビルダ定義項目「入力信号変換タイプ」に「通信入力」を指定すれば、入力データを「通信入力変換」します。

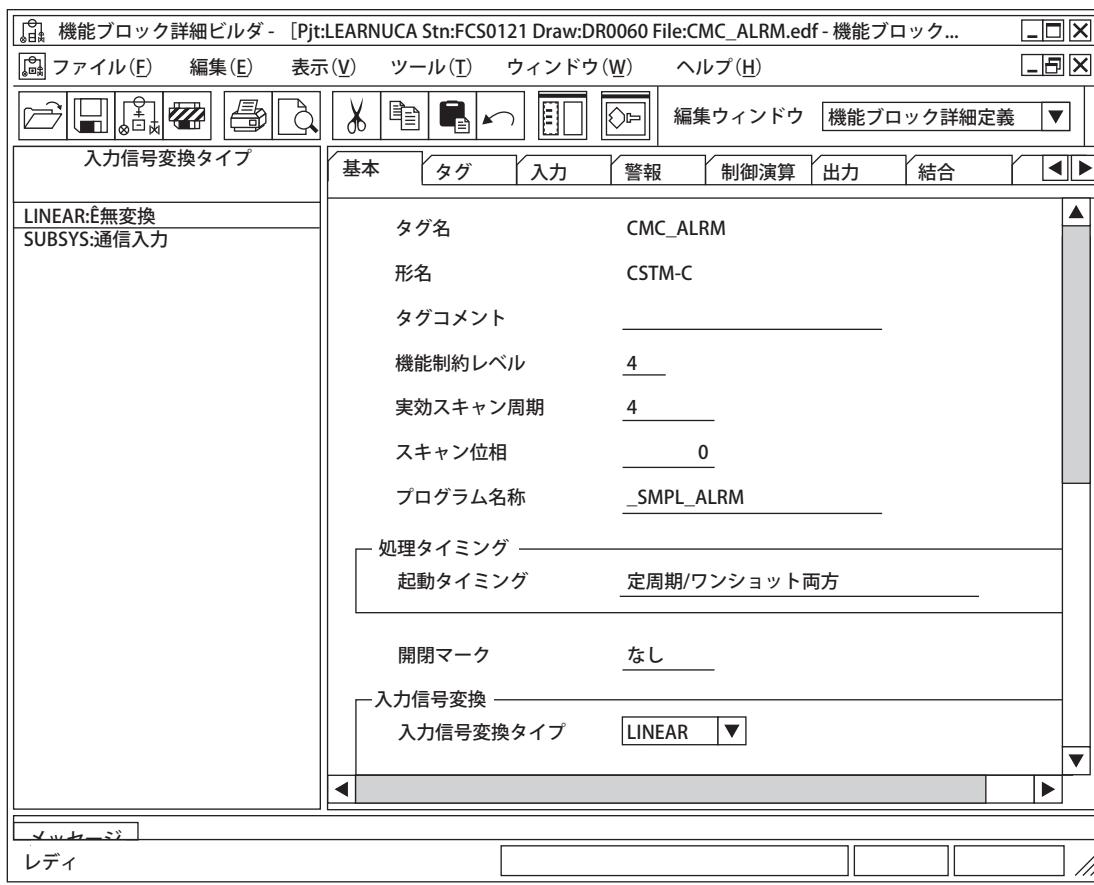


図 入力信号変換タイプの指定

参照

入力信号変換の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「3.1.1 連続制御ブロックと演算ブロックに共通の入力信号変換」](#)

● Qnn端子からの入力処理

UcaRWRead は Qnn 端子からデータを入力し、データアイテム RVnn に格納します。UcaRWRead(bc, 2, NOOPTION) という行では第 2 引数に「2」が指定されているので、Q02 端子からデータを入力しデータアイテム RV02 に格納します。UcaRWRead は Qnn 端子からデータアイテム RVnn へデータ入力をするだけで、「入力信号変換タイプ」の指定は UcaRWRead の動作には何も関係ありません。

データ入力が終了するとデータアイテム RV、RV01、RV02 からデータを取得し、測定値として 3 つの入力の合計を計算します。測定値の計算結果（変数 pv）を、UcaRWSetPv でデータアイテム PV に設定します。

6.1.1 UcaRWSetPvによるPVのデータステータス作成

UcaRWSetPvは、演算結果をデータアイテムPVに格納します。

UcaRWSetPv の呼び出し部分を確認します。

```
.....
/* 合計をデータアイテム PV に設定 */
/* 引数 rtnReadIn は、IN 端子読み込みのリターンコード */
rtnCode = UcaRWSetPv(bc, &Pv, rtnReadIn, NOOPTION);
.....
```

UcaRWSetPv は、次の処理を行います。

- PV のデータステータス作成
- デジタルフィルタ処理 (UcaRWSetPv にオプション UCAOPT_NOFILTER を指定するとビルダ定義項目「入力フィルタ」の定義に関係なくデジタルフィルタ処理を行いません)
- 積算処理 (UcaRWSetPv にオプション UCAOPT_NOSUM を指定するとビルダ定義項目「積算単位時間」の定義に関係なく積算処理を行いません)
- PV へのデータ設定

参照 デジタルフィルタおよび積算の詳細については、以下を参照してください。

機能ブロック共通機能リファレンス (IM 33J15A20-01JA) 「3.2 デジタルフィルタ」

機能ブロック共通機能リファレンス (IM 33J15A20-01JA) 「3.3 積算」

PV のデータステータス作成について説明します。データステータス作成方法は、ビルダ定義項目「演算入力値異常検出」の指定により異なります。連続制御形ユーザカスタムブロック CMC_ALRM には、「全検出形」を指定しています (CSTM-C のデフォルトは、補正演算形です)。

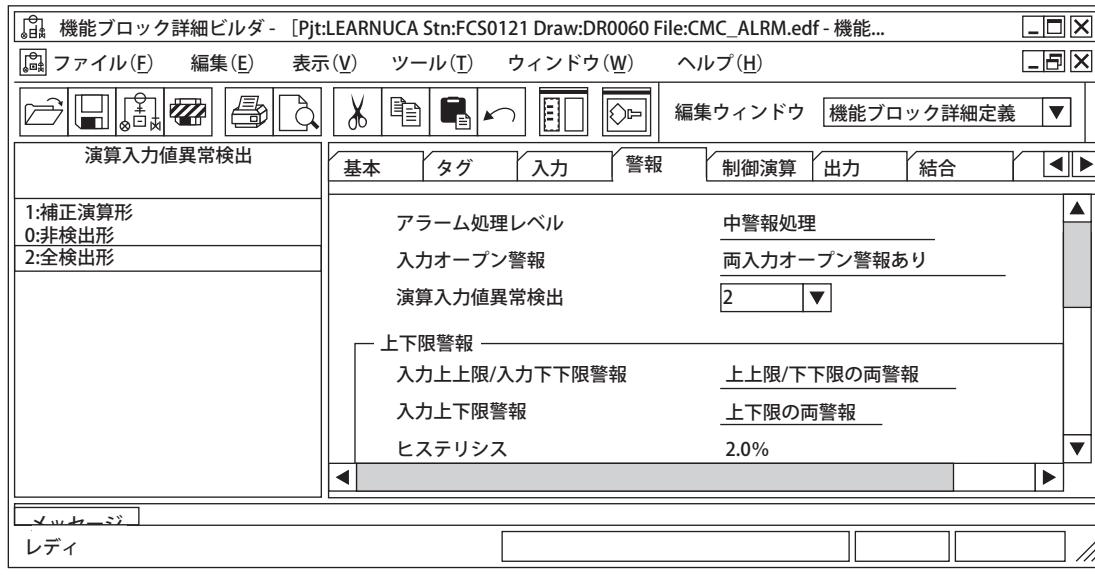


図 演算入力値異常検出の指定

UcaRWSetPv の動作を以下に示します。

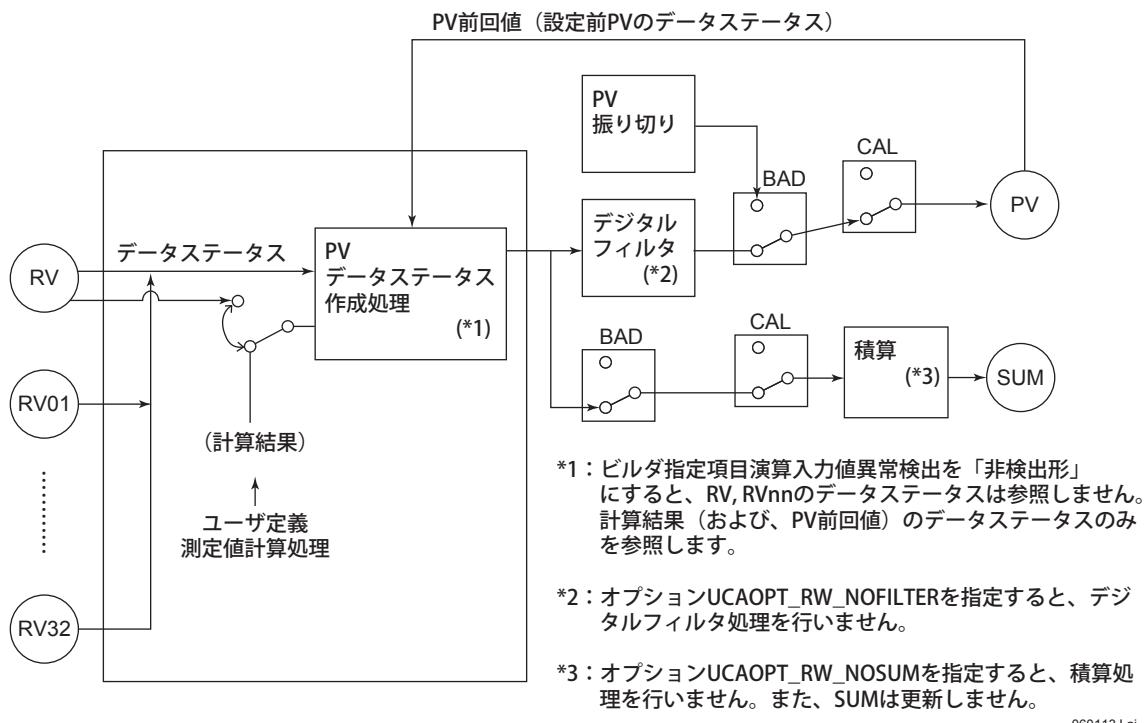


図 UcaRWSetPvの動作

060113J.ai

PV のデータステータス BAD と QST は、次の 3 つのデータから作成されます。

- UcaRWSetPv の引数に指定する演算結果のデータステータス BAD と QST
「UcaRWSetPv(bc, NULL, rtnReadIn, NOOPTION);」のように UcaRWSetPv の第 2 引数（演算結果を指定）に「NULL」を指定すると、演算結果のデータステータスは正常（BAD でも QST でもない）として扱われます。
- データアイテム RV に入力したデータのデータステータス BAD と QST
- データアイテム RV01 ~ RV32 に入力したデータのデータステータス BAD と QST

UcaRWSetPv は、この 3 つのデータのデータステータス (BAD と QST) を、ビルダ定義項目「演算入力値異常検出」の指定により、以下のように使用してデータアイテム PV に設定するデータステータス (BAD と QST) を決定します。なお、UcaRWSetPv に UCAOPT_RW_NOPVSTS オプションを指定すると、ビルダ定義項目「演算入力値異常検出」の指定とは無関係に、常に「非検出形」指定と同じ動作になります。

表 PV データステータス作成

ビルダ定義項目 「演算入力値異常検出」 の指定	(1) UcaRWSetPv の引 数のデータステータス	(2) RV のデータ ステータス	(3) RVnn のデータ ステータス	作成される PV の データステータス
全検出形	BAD	任意	任意	BAD
	任意	BAD	任意	BAD
	任意	任意	BAD	BAD (*1)
	QST	not BAD	not BAD	QST
	not BAD	QST	not BAD	QST
	not BAD	not BAD	QST	QST
	not BAD nor QST	not BAD nor QST	not BAD nor QST	0 (正常)
補正演算形	BAD	任意	任意	BAD
	任意	BAD	任意	BAD
	任意	任意	BAD	QST (*1)
	QST	not BAD	not BAD	QST
	not BAD	QST	not BAD	QST
	not BAD	not BAD	QST	QST
	not BAD nor QST	not BAD nor QST	not BAD nor QST	0 (正常)
非検出形	BAD	任意	任意	BAD
	QST	任意	任意	QST
	not BAD nor QST	任意	任意	0 (正常)
UCAOPT_RW_NOPVSTS オプションを指定 （「演算入力値異常検出」 の指定は無視されます）	BAD	任意	任意	BAD
	QST	任意	任意	QST
	not BAD nor QST	任意	任意	0 (正常)

任意： 任意のデータステータス

*1： 全検出形と補正演算形では、RVnn が BAD の場合の扱いのみが異なります。

ビルダ定義項目と UcaRWSetPv のオプション UCAOPT_RW_NOPVSTS の指定により、データステータスの作成方法が決まります。データステータスの作成方法が決まると、上表の各行の条件を上から下に評価して最初に該当する行に従って PV のデータステータスの BAD と QST が決定されます。

データステータス作成処理は、ユーザ定義関数の入り口（たとえば、機能ブロック定期処理の場合 UcaBlockPeriodical の先頭行）から UcaRWSetPv を呼び出すまでに、以下のユーザカスタムアルゴリズム作成用ライブラリによりデータが設定されているデータアイテム (RV01 ~ RV32) のみを対象とします。

表 データステータス作成の対象となるRVnnを決める関数

関数名	処理内容
UcaRWRead	入力端子 Qnn からデータアイテム RVnn へのデータ入力
UcaDataStoreRvn	データアイテム RVnn へのデータ設定
UcaTagReadToRvnF64S	タグ名を指定してデータを入力し、入力データを RVnn に設定 (自ステーションデータ)
UcaOtherTagReadToRvnF64S	タグ名を指定してデータを入力し、入力データを RVnn に設定 (他ステーションデータ)

重要

結合を定義してある入力端子からのみデータ入力処理を行ってください。入力結合を定義していない入力端子 Q01 ~ Q32 に対して UcaRWRead で入力端子からデータを読み込むと、対応するデータアイテム RV01 ~ RV32 のデータステータスは QST になります (RV01 ~ RV32 のデータ値は不变です)。

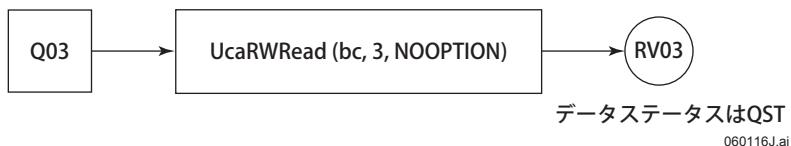


図 入力結合未定義のQnn端子からの読み込み

重要

1回のユーザ定義関数の処理内では、UcaRWSetPv は一度だけ呼び出してください。機能ブロック定期処理の場合、UcaBlockPeriodical の入り口から return するまでの間で、UcaRWSetPv を一度だけ呼び出してください。

1回のユーザ定義関数の処理内では、UcaRWSetPv を複数回呼び出さないでください。複数回呼び出すと、たとえば SUB 端子から出力可能な ΔPV (前回 PV と今回 PV の差分) が正常に作成できなくなります。 ΔPV は UcaRWSetPv が呼び出された間の PV の差分です。UcaRWSetPv を1回のユーザ定義関数の処理内で2回呼び出すと、 ΔPV は同じ処理内の2回の UcaRWSetPv 呼び出しに指定した PV 値の差分となってしまいます。

6.1.2 ユーザ定義アラームの発生と復帰

alrm_calcは計算した測定値をUcaRWSetPvでデータアイテムPVに設定が完了すると、次にユーザ定義アラームUHIGとULOWの検出をします。まず、実際にUHIGアラームとULOWアラームを発生・復帰してみます。

以下の表に従って連続制御形ユーザカスタムブロック CMC_ALRM のチューニングウィンドウを表示し、データを手入力してください。

表 CMC_ALRMに手入力で設定するデータ

データアイテム	データ	意味
PH	70	HI アラーム上限値
PL	30	LO アラーム下限値
P01	60	UHIG アラーム上限値
P02	40	UHIG アラーム下限

以下の図は、CMC_ALRM のチューニングウィンドウのデータアイテム P01 と P02 表示部分です。データアイテム S01 と S02 には、機能ブロック初期化処理で「UHIG 上限値」と「UHIG 下限値」をコメントとして設定しています。

参照 機能ブロック初期化処理の詳細については、以下を参照してください。
[「6.1.4 機能ブロック初期化処理（データアイテム P01 と P02 の初期化）」](#)

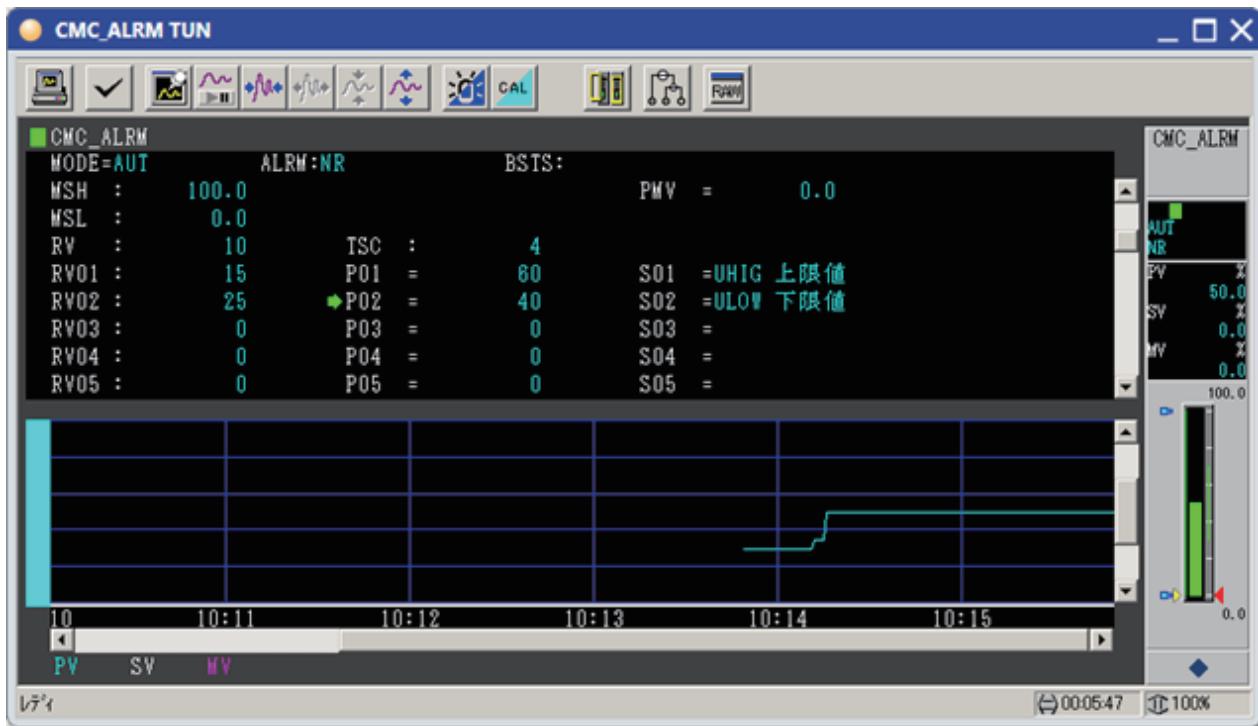
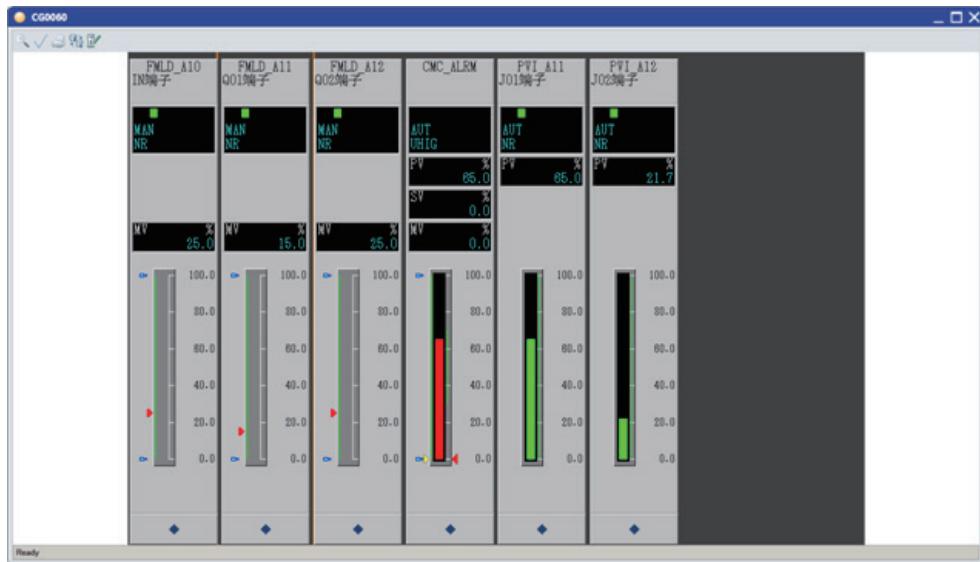


図 CMC_ALRMのチューニングウィンドウ

060118J.ai

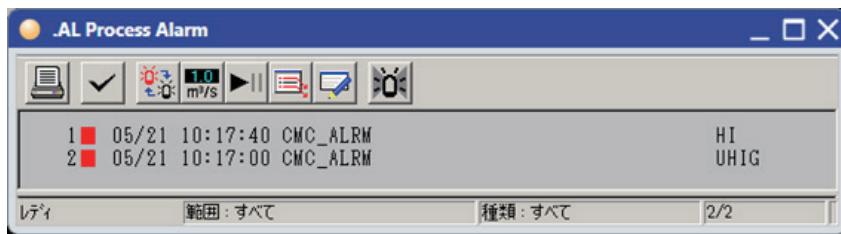
CMC_ALRM にデータを出力している手動操作ブロックの MV を増加します。CMC_ALRM の PV 値 (3 つの MV 値の合計) が 60.0 を越えると、UHIG アラームが発生します。以下は、CMC_ALRM の PV 値が 65.0 になり、UHIG アラームを発生している状態です。



060119J.ai

図 UHIGアラームの発生

続いて、いずれかの MV 値をさらに 10.0 増加してください。CMC_ALRM の PV 値が 75.0 になり、HI アラームが発生します。



060120J.ai

図 HIアラームの発生

CMC_ALRM の計器図から「UHIG」の表示が消え、代わりに「HI」が表示されていることを確認してください。アラームステータスのユーザ定義アラームの定義をもう一度見てみます。ユーザ定義アラームは、システムビューの COMMON の AlmStsLabel で定義します。AlmStsLabel をダブルクリックして以下のウィンドウを呼び出してください。



図 ユーザ定義アラームの定義

ユーザ定義アラーム UHIG と ULLOW を、ビット番号 19 と 20 に定義しています。入力上限／下限アラーム HI と LO は、ビット番号 17 と 18 に定義されています。ビット番号の小さい方が優先順位が高く、HI は UHIG より優先度が高くなります。このため、UHIG アラームと HI アラームが両方発生しているときには、計器図には優先度の高い HI が表示されます。

重要 HI や LO のようなデフォルトのアラームのビット番号を変更することはできません。

手動操作ブロックの MV 値を変更し、CMC_ALRM の HI、LO、UHIG、ULOW アラームを発生・復帰してください。

■ ユーザ定義アラーム検出のプログラミング

プログラムについて説明します。alrm.c の alrm_calc 関数の以下の部分を見てください。最初に UcaDataGetPv で、データアイテム PV から変数 pv にデータを取り出しています。前節で説明したように、データアイテム PV にデータを設定するときに UcaRWSetPv がデータステータスを作り出していますので、この「PV の取り直し」が必要です。このプログラムでは、データアイテム PV のデータステータスが BAD でなければ、UHIG と ULOW アラームを検出しています。

(続き)

```
/* ===== ユーザ定義アラーム UHIG と ULOW の検出 ===== */
/* UcaRWSetPv が作成したデータステータスを反映した PV 値を取得 */
rtnCode = UcaDataGetPv(bc, &pv, NOOPTION);
if (!UcaDstsIsBad(bc, pv.status)){
    /* データステータスが BAD でないときに限りアラームを検出します */
    rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION); /* P01 は UHIG 上限値 */
    rtnCode = UcaDataGetPn(bc, &p02, 2, 1, NOOPTION); /* P02 は ULOW 下限値 */

    /* UHIG アラームの検出 */
    if (pv.value > p01.value){
        rtnCode = UcaAlrmSet(bc, USR_ALRM_UHIG); /* UHIG 発生を設定 */
    }

    /* ULOW アラームの検出 */
    if (pv.value < p02.value){
        rtnCode = UcaAlrmSet(bc, USR_ALRM_ULOW); /* ULOW 発生を設定 */
    }
}
```

連続制御形ユーザカスタムブロック CMC_ALRM のデータアイテム P01 に UHIG 上限値、P02 に ULOW アラーム下限値を保持しています。PV が P01 より大きくなると UcaSetAlrm で UHIG アラームの発生を設定し、PV が P02 より小さくなると UcaSetAlrm で ULOW アラームの発生を設定しています。

UcaSetAlrm の引数に設定してあるラベル USR_ALRM_UHIG と USR_ALRM_OLOW は、ユーザ定義インクルードファイル usrstatus.h に定義しています。

参照 ユーザ定義インクルードファイル usrstatus.h の詳細については、以下を参照してください。
[「4.3.3 ユーザ定義インクルードファイル usrstatus.h」](#)

alrm.c の最初の方で、usrstatus.h を #include で取り込んでいます。

```
...
/***********************/
* ユーザ定義インクルードファイル
*/
#include <usrstatus.h> /* ユーザ定義ステータス */
...
```

usrstatus.h では、ラベル USR_ALRM_UHIG と USR_ALRM_ULOW を #define で定義しています。

```
.....
/* アラームステータス */
#define USR_ALRM_UHIG UCAMASK_ALRM_19 /* UHIG */
#define USR_ALRM_ULOW UCAMASK_ALRM_20 /* ULOW */
.....
```

alarm_calc 関数は、処理の最初で USR_ALRM_UHIG と USR_ALRM_ULOG を指定して UcaAlrmClear を呼び出して、アラームの復帰を設定しています。

```
.....
/*
 * アラームを検出する場合は、
 * 処理の最初で対象アラームをクリアしておきます。
 */
rtnCode = UcaAlrmClear(bc, USR_ALRM_UHIG);
rtnCode = UcaAlrmClear(bc, USR_ALRM_ULOW);
.....
```

アラーム復帰を設定する UcaAlrmClear とアラーム発生を設定する UcaAlrmSet は、両者の最後の設定が有効（後優先）となります。

参照 UcaAlrmClear と UcaAlrmSet の使い方の詳細については、以下を参照してください。
「[5.2.2 演算異常 ERRC アラームの発生と復帰](#)」

6.1.3 J01端子とJ02端子からのデータ出力

J01端子とJ02端子からのデータ出力について説明します。

測定値作成処理をする alrm_calc 関数は、最後に J01 端子と J02 端子からデータを出力します。alrm_calc は、3 つの入力の平均をデータアイテム MV02 に設定します。

続いて、UcaRWWritePvToJnSub で PV のデータを J01 端子から出力します。また、UcaRWWrite で MV02 のデータを J02 端子から出力します。

```
/* 3入力の平均を計算しデータアイテム MV02 に設定 */
ave.value = (rv.value + rv01.value + rv02.value) / 3.0;
ave.status = 0;
rtnCode = UcaDataStoreF64SToMvn(bc, &ave, 2, NOOPTION);

/* ===== 出力処理 ===== */
/* PVに設定された合計を J01 端子から出力 */
/* MVを使っていないので、SUB 端子出力は PV または ΔPV のみ有効 */
rtnCode = UcaRWWritePvToJnSub(bc, 1, UCAOPT_RW_SETCNF);

/* MV02 に設定された平均を J02 端子から出力 */
rtnCode = UcaRWWrite(bc, 2, UCAOPT_RW_SETCNF);

return SUCCEED;
}
```

UcaRWWritePvToJnSub は SUB 端子への出力も行いますが、出力内容はビルダ定義項目「補助出力」の指定で決まります。このプログラムでは、データアイテム MV は使用しませんので、「補助出力」の指定には「PV」または「 ΔPV 」を指定します。



図 補助出力の指定

6.1.4 機能ブロック初期化処理（データアイテムP01とP02の初期化）

_SMPL_ALRMの機能ブロック初期化処理では、次の2つの処理を行っています。

- ・ブロックモードをAUTに初期化します。
- ・データアイテムP01、P02、S01、S02を初期化します。

alm.c を UcaBlockInit で検索し、以下の部分を見てください。

```
/*
*<<FNH>>*****
*
* Function name:          UcaBlockInit
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC 处理なし
*                         UCAERR_STOPME 处理続行不能
*
* description:            機能ブロック初期化処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockInit(
    UcaBlockContext bc, /* (IN/OUT) : ブロックコンテキスト */
    I32 reason        /* (IN) : 呼び出し理由 */
)
{
    I32 i08;           /* I08 */
    F64S p01;          /* P01 */
    F64S p02;          /* P02 */
    F32 sh;            /* SH */
    F32 sl;            /* SL */
    I32 rtnCode;       /* リターンコード */

    /* ブロックモードを AUT に初期化 */
    rtnCode = UcaModeSet(bc, UCAMASK_MODE_AUT);

    /* I08 が 0 なら P01 と P02 を初期化します */
    rtnCode = UcaDataGetIn(bc, &i08, 8, 1, NOOPTION);
    if (i08 == 0){
        /* ===== P01 に初期値として SH を設定 ===== */
        rtnCode = UcaDataGetSh(bc, &sh, NOOPTION);
        p01.value = sh;           /* P01 は UHIG アラーム上限値 */
        p01.status = 0;
        rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);
        /* S01 は P01 を説明 */
        rtnCode = UcaDataStoreEachSn(bc, "UHIG 上限値", 1, NOOPTION);

        /* ===== P02 に初期値として SL を設定 ===== */
        rtnCode = UcaDataGetSl(bc, &sl, NOOPTION);
        p02.value = sl;           /* P02 は ULOW アラーム下限値 */
        p02.status = 0;
        rtnCode = UcaDataStorePn(bc, &p02, 2, 1, NOOPTION);
        /* S02 は P02 を説明 */
        rtnCode = UcaDataStoreEachSn(bc, "ULOW 下限値", 2, NOOPTION);
    }
}
```

(続く)

(続き)

```

/* ===== i08 に初期化済み (-1) を設定 ===== */
i08 = -1;
rtnCode = UcaDataStoreIn(bc, &i08, 8, 1, NOOPTION);
}

return SUCCEED;
}

```

機能ブロック初期化処理の最初で、UcaModeSet によりブロックモードを AUT に初期化しています。また、このプログラムの機能ブロックデータ設定時特殊処理で、ブロックモードを AUT と O/S に限定しています。

参照 ブロックモードの限定の詳細については、以下を参照してください。

[「3.6.1 CSTM-C のブロックモードを AUT と O/S に限定」](#)

次に、データアイテム I08 が 0 であれば、データアイテム P01、P02、S01、S02 を以下に従い初期化しています。データアイテム S01 と S02 に設定している文字列はチューニングウィンドウを表示したときのための「コメント」です。プログラムの動作には何の影響もありません。

表 初期値の設定

データアイテム	初期値	用途
P01	データアイテム SH のデータ	UHIG の上限値
P02	データアイテム SL のデータ	ULOW の下限値
S01	UHIG 上限	コメント
S02	ULOW 下限	コメント

本プログラムのデータアイテム P01 と P02 の初期化は、HIS など外部からの設定したデータをチューニングパラメータセーブしたものより「弱い」初期化となります。

参照 「弱い」初期化の詳細については、以下を参照してください。

[「5.2.4 チューニングパラメータより弱い初期値の設定方法」](#)

6.1.5 機能ブロックデータ設定時特殊処理 (AUTとO/Sに限定)

_SMPL_ALRMの機能ブロックデータ設定時特殊処理では、次の2つの処理を行っています。

- ・ ブロックモードをAUTとO/S以外の変更を禁止します。
- ・ データアイテムS01、S02とI08への設定を禁止します。

alrm.c を UcaBlockDset で検索し、以下の部分を見つけてください。

```

/*
*<<FNH>>*****
*
* Function name:          UcaBlockDset
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME   処理続行不能
*
* description:            機能ブロックデータ設定時特殊処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockDset(
    UcaBlockContext bc,           /* (IN/OUT) : ブロックコンテキスト */
    DITMN itemName,              /* (IN) : データアイテム名 */
    UcaUnivType *data,           /* (IN) : 設定データ */
    UcaDataOrStatus dataOrStatus, /* (IN) : 設定種別 */
    UcaPreOrPost preOrPost,       /* (IN) : 呼び出し種別 */
    BOOL *result                 /* (OUT) : 設定可否 */
)
{
    /* データ設定前処理でないなら何もしない */
    if (preOrPost != UCADSET_PRE) {
        return UCAERR_NOPROC;
    }
    /* ===== データ設定の可否を検査 ===== */
    if (strncmp(itemName, "MODE", sizeof(DITMN)) == 0
        && dataOrStatus == UCADSET_STATUS) {
        /* ブロックモードに対するモード変更指令 */
        /* AUTとO/Sは設定可能 */
        if (strncmp(data->dataValue.string, "AUT", sizeof(data->dataValue.string)) == 0
            || strncmp(data->dataValue.string, "O/S", sizeof(data->dataValue.string)) == 0) {
            *result = UCADSET_ALLOW; /* データ設定可能 */
        } else {
            *result = UCADSET_DENY; /* データ設定不可 */
        }
    } else if (strncmp(itemName, "S01", sizeof(DITMN)) == 0
        || strncmp(itemName, "S02", sizeof(DITMN)) == 0
        || strncmp(itemName, "I08", sizeof(DITMN)) == 0) {
        /* S01,S02とI08はデータ変更不可 */
        /* S01はP01の説明 */
    }
}

```

(続く)

(続き)

```
/* S02 は P02 の説明 */
/* I08 は初期値設定フラグ */
*result = UCADSET_DENY; /* データ設定不可 */
} else {
    *result = UCADSET_ALLOW; /* データ設定可能 */
}

return SUCCEED;
}
```

機能ブロックデータ設定時特殊処理で、ブロックモードを AUT と O/S に限定しています。

参照 ブロックモードの限定の詳細については、以下を参照してください。

[「3.6.1 CSTM-C のブロックモードを AUT と O/S に限定」](#)

また、データアイテム S01、S02 と I08 へのデータ設定を禁止しています。データアイテム S01 と S02 には「UHIG 上限値」と「ULOW 下限値」が設定されていますが、これらはコメントなので外部からのデータ変更を禁止しています。また、データアイテム I08 は初期化済みか否かを示すフラグなので、外部からのデータ変更を禁止しています。

6.2 CSTM-Cのブロックモード遷移

連続制御形ユーザカスタムブロック（CSTM-C）は、PID調節ブロックと同じブロックモードを持ちます。ブロックモードを切り換えることにより、自動（AUT）、手動（MAN）やカスケード（CAS）などの動作をCSTM-Cで実現することができます。

CSTM-C の基本ブロックモードを以下に示します。

表 連続制御形ユーザカスタムブロックの基本ブロックモード

シンボル	名称	説明
O/S	サービスオフ (Out of Service)	機能ブロックの機能がすべて停止している状態
IMAN	初期化手動 (Initialization MANual)	演算処理と出力処理が停止している状態
TRK	トラッキング (TRacking)	演算処理が停止していて、指定値を強制出力している状態
MAN	手動 (MANual)	演算処理が停止していて、手動で設定された操作出力値を出力している状態
AUT	自動 (AUTomatic)	演算処理を実行していて、演算結果を出力している状態
CAS	カスケード (CAScade)	カスケード上流ブロックから設定された設定値（CSV）を用いて演算処理を実行し、演算結果を出力している状態
PRD	プライマリダイレクト (PRimary Direct)	演算処理が停止していて、カスケード上流ブロックからの設定された設定値（CSV）を直接出力している状態
RCAS	リモートカスケード (Remote CAScade)	上位システムのコンピュータからリモート設定された設定値（RSV）を用いて制御演算処理を実行し、演算結果を出力している状態
ROUT	リモート出力 (Remote OUTput)	演算処理が停止していて、上位システムのコンピュータからリモート設定された操作出力値（RMV）を直接出力している状態

参照 ブロックモードの詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「6.1 ブロックモード」](#)

この節では、ブロックモードにより動作を変えるCSTM-Cのプログラミングについて説明します。ブロックモード遷移の動作を説明するためのごく単純な演算をするサンプルプログラムが用意してありますので、動かしてみます。サンプルソリューション _SMPL_MODE の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは 1 章の準備作業により以下の作業フォルダにコピーされています。

<ドライブ名>¥UcaWork¥UcaSamples¥_SMPL_MODE

Visual Studio を起動し、_SMPL_MODE¥_SMPL_MODE.sln を開きます。[ビルド] メニューの [リビルド] で _SMPL_MODE の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。

トラッキングスイッチとして使用するコモンスイッチにタグ名を定義します。FCS0121 (APCS) のコモンスイッチ %SW0201 (空いていれば %SW0201 以外でも構いません) に、タグ名 SW_C20 を付けます。システムビューの SWITCH フォルダの SwitchDef をダブルクリックしてください。%SW0201 にタグ名とタグコメントを指定して [ファイル] – [上書き保存] を行います。

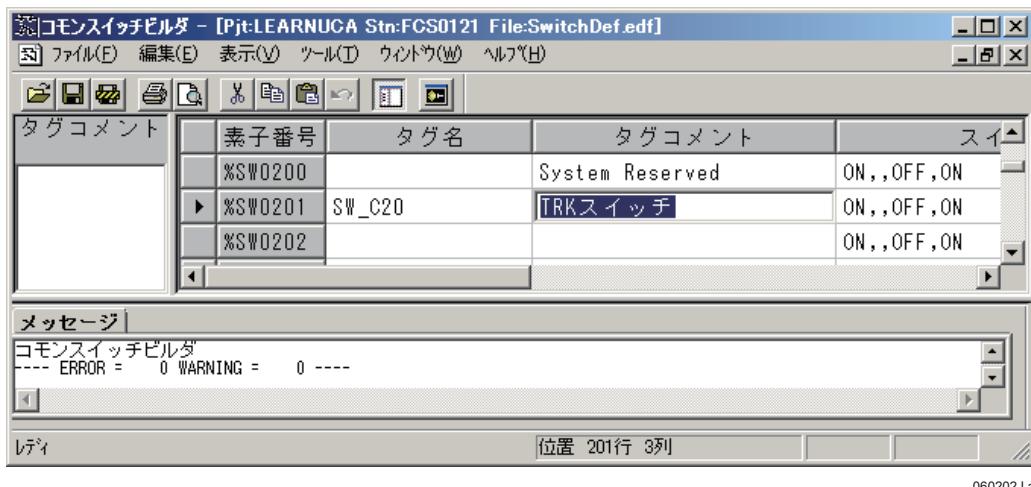


図 タグ名とタグコメントの指定

次にサンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイル MODE.txt が CENTUM VP インストール先の以下にあります。MODE.txt を FCS0121 (APCS) の制御ドローイング DR0061 (空いていればどの制御ドローイングでも構いません) にインポートします。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥drawings¥MODE.txt

動作を確認するためにコントロール (8 ループ) のウィンドウが 2 つ用意していますので、HIS0164 の CG0061 と CG0062 (他の名前でも構いません) に取り込んでおきます。テキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥graphics¥MODE_CG.xaml と MODE2_CG.xaml

制御ドローイングを見てください。連続制御形ユーザカスタムブロック CMC_MODE は、3つの入力端子からデータを入力します。IN 端子は手動操作ブロック MLD_C20.MV、Q01 端子は MLD_C21.MV、Q02 端子は MLD_C22.MV と接続してあります。また、CMC_MODE は、OUT 端子と J01 端子からデータを出力します。OUT 端子は指示ブロック PVI_C20.PV、J01 端子は PVI_C21.PV と接続してあります。

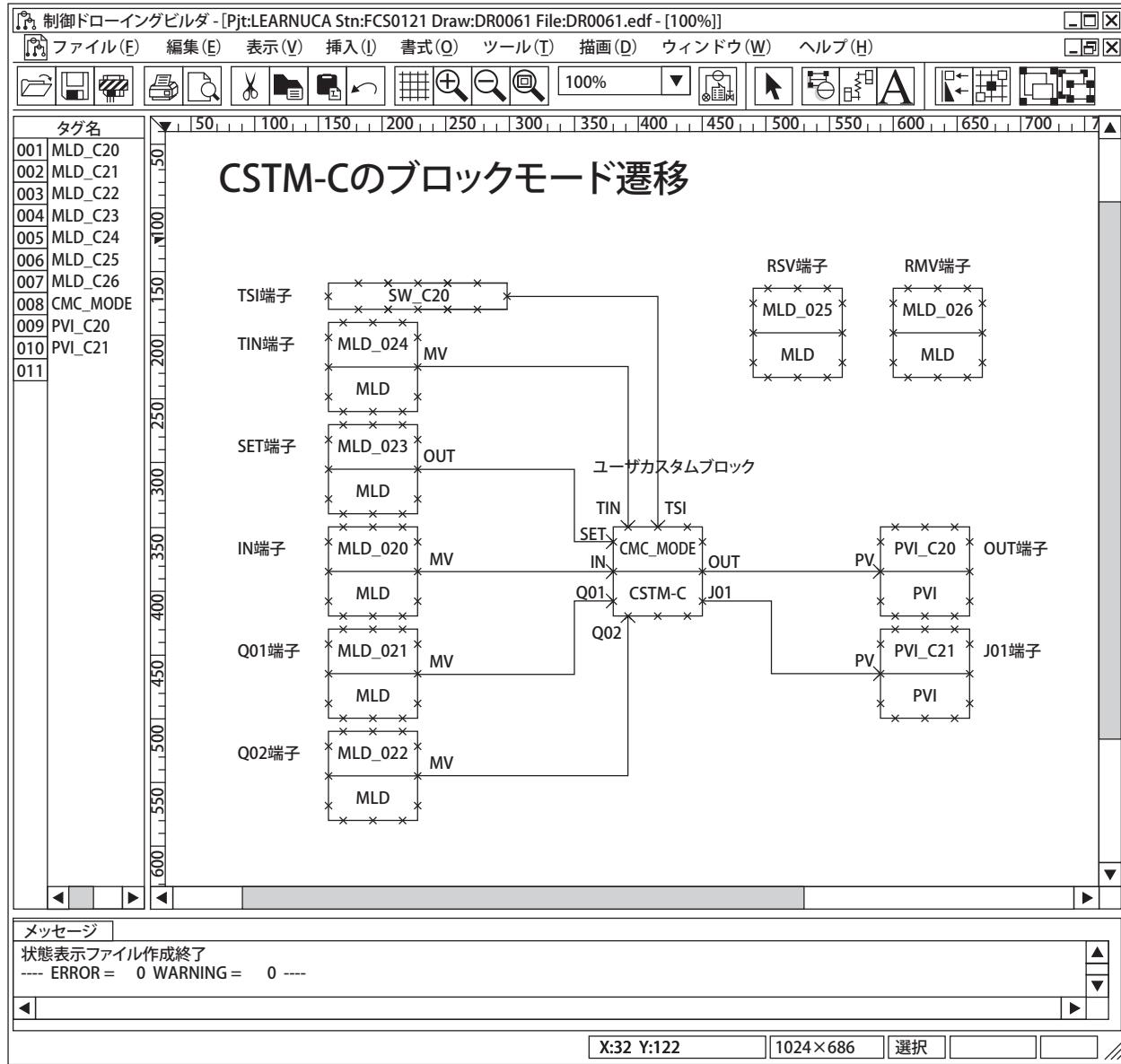


図 CMC_MODE の制御ドローイング

CMC_MODE の PV 値とデータアイテム P01 は、次の計算式で作成されます。

- PV 値は、3つの入力の合計です。
 $PV = IN \text{ 端子入力値} + Q01 \text{ 端子入力値} + Q02 \text{ 端子入力値}$
- P01 は、3つの入力の平均です。
 $P01 = (IN \text{ 端子入力値} + Q01 \text{ 端子入力値} + Q02 \text{ 端子入力値}) / 3.0$

この制御ドローイングではブロックモードごとの動作を確認できるように、CMC_MODE の SET 端子やデータアイテム RSV にデータを結合してあります。それでは、プログラムを動かしてみます。バーチャルテスト機能で FCS0121 (APCS) を起動します。起動が完了したらウィンドウ CG0061 を表示します。



図 ウィンドウの呼び出し

3つの手動操作ブロック (MLD_C20、MLD_C21、MLD_C22)、1つの連続制御形ユーザカスタムブロック (CMC_MODE) と2つの指示ブロック (PVI_C20、PVI_C21) が並んでいます。

■ ブロックモードAUTの動作

CMC_MODE のブロックモードを AUT にしてください。また、CMC_MODE のデータアイテム SV に 2.0 を設定してください。手動操作ブロックの MV 値を変更すると、3つの MV 値の合計が CMC_MODE の PV 値となります。また、データアイテム MV は PV の 2.0 倍 ($PV \times SV$) となり、同じ値が OUT 端子から出力され、PVI_C20 の PV 値となります。PVI_C21 には CMC_MODE がデータアイテム MV01 を経由して J01 端子から出力する値、つまり 3つの MV 値の平均に SV (2.0) を掛けた値が表示されます。3つの MV 値をそれぞれ 8.0、10.0、12.0 を設定した結果、CMC_MODE の PV が 30.0、CMC_MODE の MV と PVI_C20 の PV が 60.0 になった状態を次図に示します。

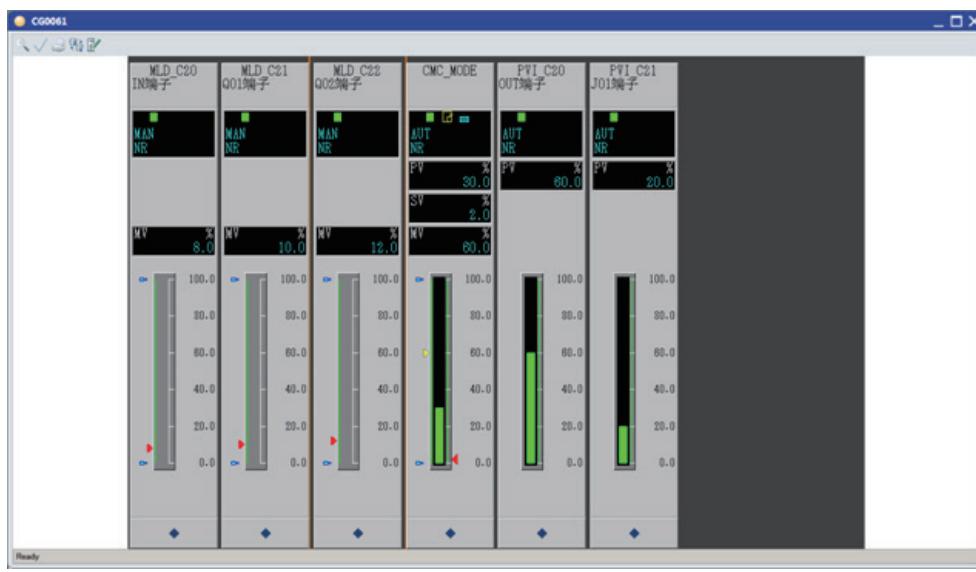


図 ブロックモードAUTの動作

手動操作ブロックの MV 値を変更して、CMC_MODE の PV と MV が変化するのを確認してください。また、CMC_MODE の SV 値を変更して CMC_MODE の MV 値が「PV × SV」で計算されること、およびデータアイテム MV01 を経由して J01 端子からは「データアイテム P01 に設定される 3 つの入力の平均× SV」が出力されること (PVI_C21.PV で表示) を確認してください。上図の状態におけるデータアイテム P01 と MV01 の値を以下に示します。

P01 には 3 つの入力の平均が設定されています ($(8.0 + 10.0 + 12.0) \div 3 = 10$)。

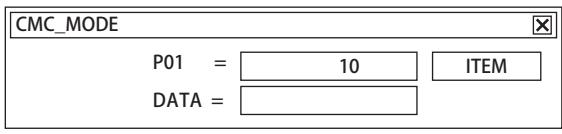


図 P01

MV01 には、P01 × SV が設定されています ($10 \times 2.0 = 20$)。MV01 の値は J01 端子から出力されます (指示ブロック PVI_C21 の PV として表示されています)。

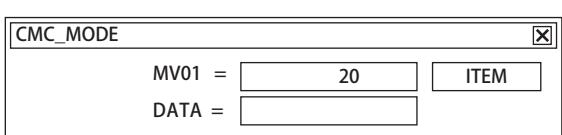
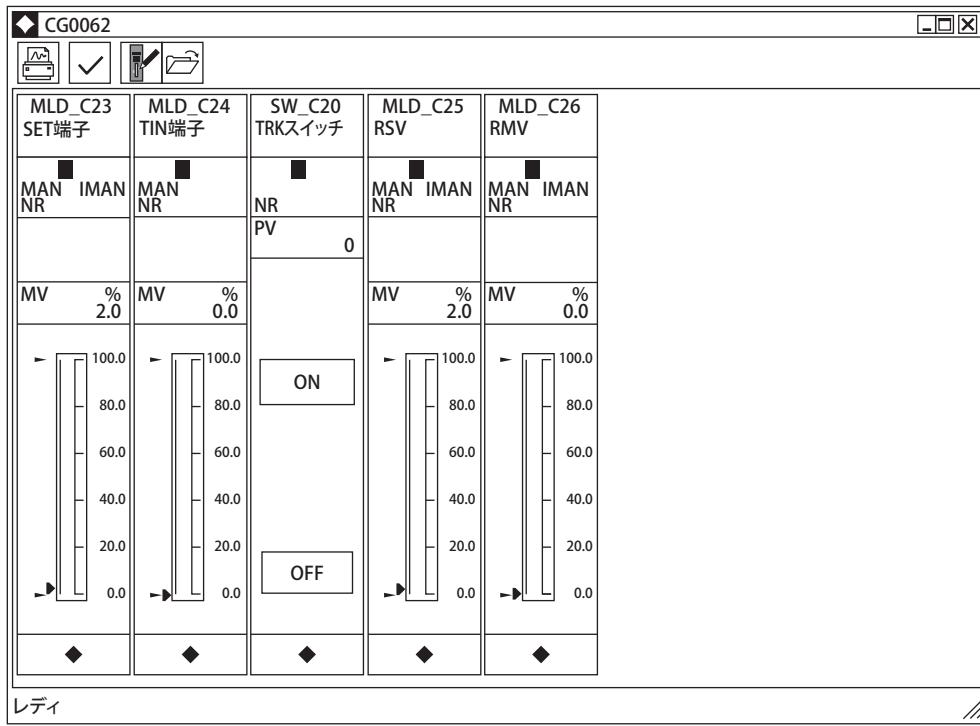


図 MV01

■ ブロックモードCASの動作

次にブロックモード CAS の動作を確認します。CMC_MODE の SET 端子には、手動操作ブロック MLD_C23 の MV を結合しています。ウィンドウ CG0062 を表示してください。



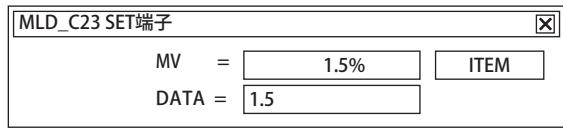
060208J.ai

図 手動操作ブロックのコントロールウィンドウ

この 8 ループのウィンドウには、連続制御形ユーザカスタムブロック CMC_MODE へデータ出力をする手動操作ブロックが並べてあります。また、ソフトスイッチ SW_C20 (%SW0201) の PV を CMC_MODE の TSI 端子に接続してあります。SW_C20 の PV をトラッキングスイッチとして使用しています。SW_C20 の PV を 1 (ON) にすると CMC_MODE は、トラッキング動作をします（ここでは 0 (OFF) のままにしておきます）。

手動操作ブロック MLD_C23 と MLD_C25 は、ビルダ定義項目「出力トラッキング」を「あり」にしてあります。このため MLD_C23 の MV は、CMC_MODE の SV と同じ 2.0 になっています。また、MLD_C25 の MV は、CMC_MODE の RSV と同じ 2.0 になっています。

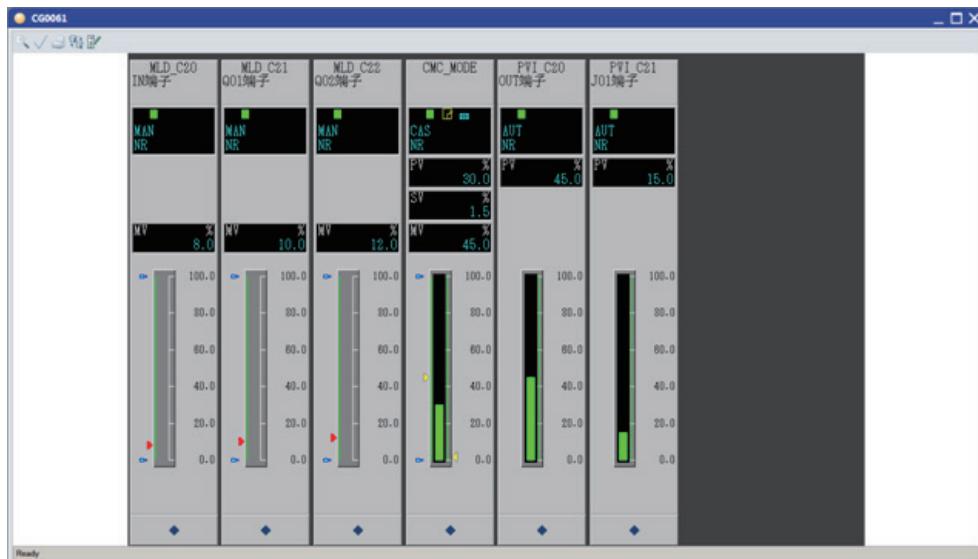
MLD_C23 のブロックモードは結合先の CMC_MODE のブロックモードが AUT の (CAS ではない) ため、MAN (IMAN) です。CMC_MODE のブロックモードを AUT から CAS に変更してください。カスケード結合が成立し、MLD_C23 のブロックモードが MAN (IMAN) から MAN になります。MLD_C23 の MV に「1.5」を入力してください。



060209J.ai

図 MVの入力

CMC_MODE のブロックモードを CAS にすると、(CMC_MODE の SET 端子と MLD_C23 の MV 値を結合してあるので) CMC_MODE のデータアイテム CSV は MLD_C23 の MV 値と同じ値になります。そして、CMC_MODE の CSV がデータアイテム SV に反映されます。つまり、MLD_C23 の MV 値が、CMC_MODE の SV 値となります (以下では、SV 値が MLD_C23 の MV 值と同じ 1.5 になっています)。CMC_MODE の MV 値は「SV × PV」、つまり「MLD_C23 の MV 値 × PV」です。MLD_C23 の MV 値を変更して、CMC_MODE の SV と MV が変化するのを確認してください。



060210J.ai

図 CMC_MODEのデータアイテムの変化

■ ブロックモードRCASの動作

ブロックモード RCAS の動作を確認します。手動操作ブロック MLD_C25 の OUT 端子と CMC_MODE のデータアイテム RSV を結合してあります。以下は、MLD_C25 の OUT 端子から CMC_MODE の RSV にデータを出力するように、機能ブロック詳細ビルダでデータ結合を定義したウィンドウです（制御ドローイングで MLD_C25 の OUT 端子と CMC_MODE の RSV を結線することが可能ですが。このサンプルでは、制御ドローイングの結線が複雑になりすぎるので、結線しないでデータ結合を以下の機能ブロック詳細ビルダで定義しています）。

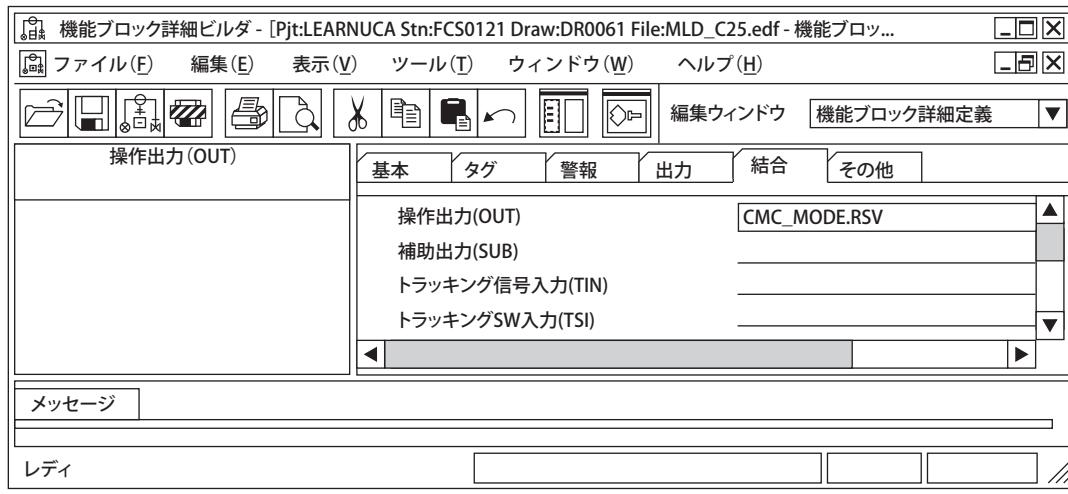


図 MLD_C25のデータ結合の定義

ウィンドウ CG0062 を見てください。MLD_C25 のブロックモードは、結合先の CMC_MODE のブロックモードが CAS の（RCAS ではない）ため、MAN (IMAN) です。

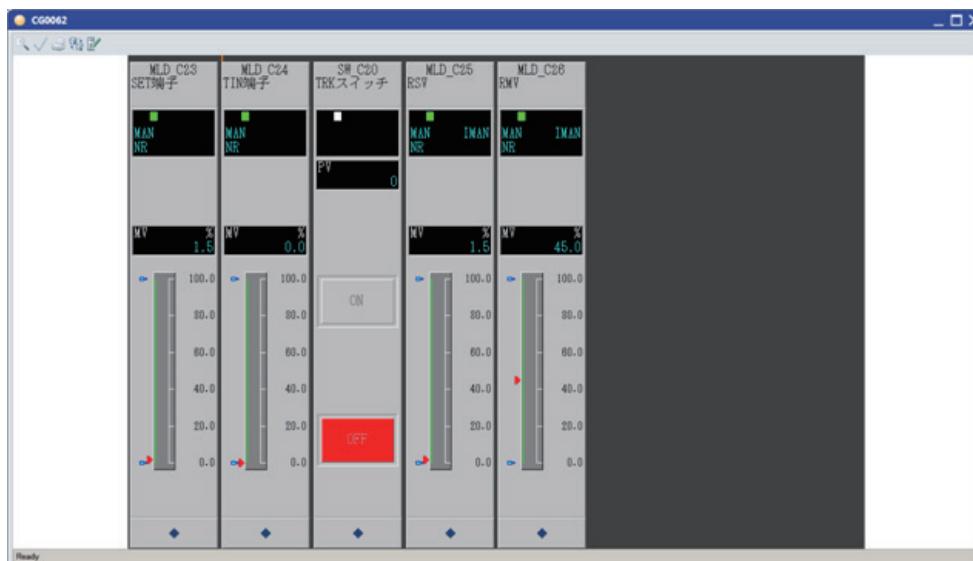
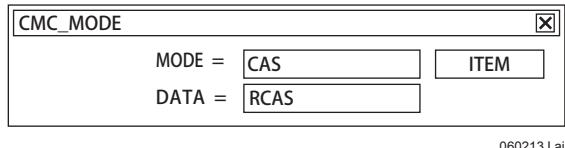


図 MLD_C25のブロックモード

CMC_MODE のブロックモードを CAS から RCAS に変更してください。

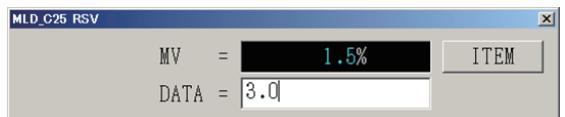


060213J.ai

図 CMC_MODEのブロックモードの変更

CMC_MODE のデータアイテム RSV と結合してある MLD_C25 のブロックモードは、(ブロックモードが RCAS になったため) MAN (IMAN) から MAN になります。また、CMC_MODE の SET 端子と結合してある MLD_C23 のブロックモードは、(ブロックモードが CAS 以外になったため) MAN から MAN (IMAN) になります。

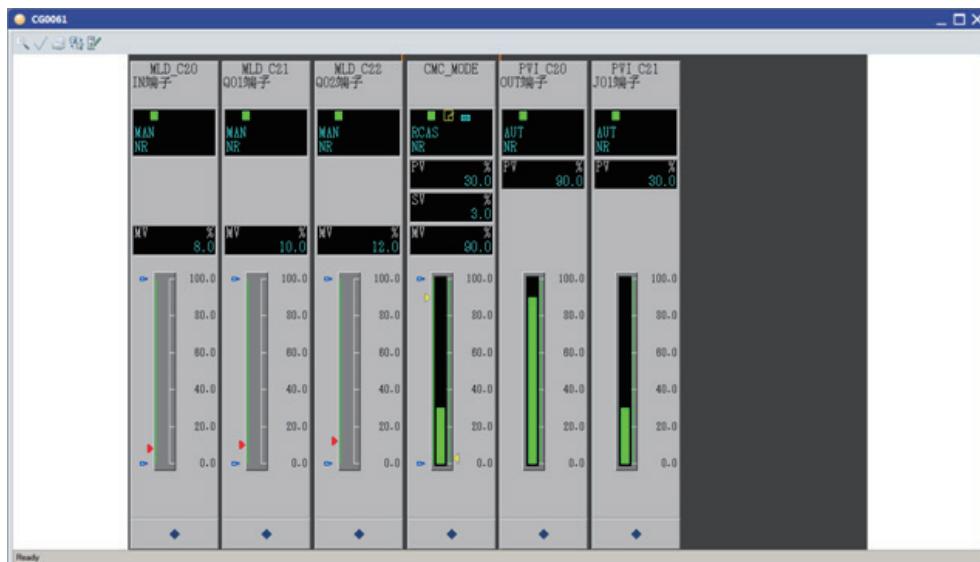
MLD_C25 の MV に「3.0」を設定してください。



060214J.ai

図 MLD_C25のMVの設定

CMC_MODE のブロックモードを RCAS にすると、MLD_C25 の MV 値は MLD_C25 の OUT 端子から CMC_MODE のデータアイテム RSV に出力されるようになります。CMC_MODE は、データアイテム RSV に設定されたデータをデータアイテム SV に反映します。つまり、MLD_C25 の MV 値が、CMC_MODE の SV 値となります (以下では、SV 値が MLD_C25 の MV 値と同じ 3.0 になっています)。CMC_MODE の MV 値は「SV × PV」、つまり「MLD_C25 の MV 値 × PV」です。MLD_C25 の MV 値を変更して、CMC_MODE の SV と MV が変化するのを確認してください。



060215J.ai

図 CMC_MODEのデータアイテムの変化

これまで、ブロックモードが AUT、CAS、RCAS の動作を見てきました。3つのブロックモードでは、CMC_MODE の SV の作り方が違うことに注意してください。

- ・ ブロックモード AUT の場合は、データアイテム SV のデータを使用します。
- ・ ブロックモード CAS の場合は、SET 端子結合先のデータがデータアイテム CSV を経由してデータアイテム SV に反映されます。
- ・ ブロックモード RCAS の場合は、データアイテム RSV のデータがデータアイテム SV に反映されます。

参照 CAS と RCAS の動作の相違の詳細については、以下を参照してください。
APCS (IM 33J15U10-01JA)

■ トラッキング動作

トラッキング動作を確認します。CMC_MODE の TSI 端子はコモンスイッチ SW_C20.PV と接続してあります。また、TIN 端子は、手動操作ブロック MLD_C24 の MV と接続してあります。MLD_C24 の MV に「35.0」を設定してください。そして、SW_C20 の PV を 0 (OFF) から 1 (ON) に変化させてください。

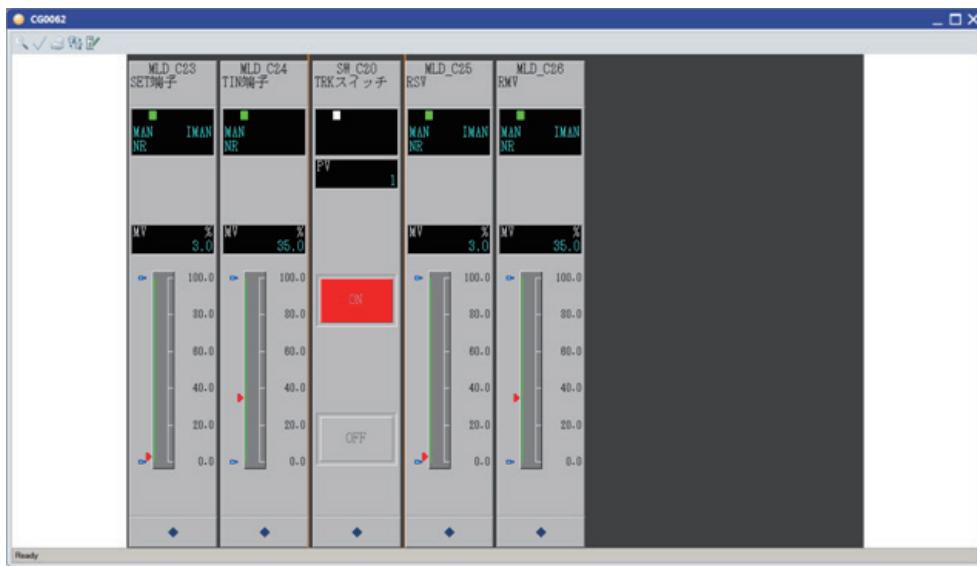


図 トランクの設定

トラッキングスイッチ (SW_C20 の PV) が 1 (ON) になると、CMC_MODE のブロックモードは RCAS から RCAS (TRK) になりトラッキング動作を開始します。つまり、CMC_MODE の OUT 端子から TIN 端子に接続してある MLD_C24 の MV 値がそのまま出力されます。以下は、CMC_MODE の MV 値と CMC_MODE の OUT 端子と結合してある PVI_C20 の PV 値が、MLD_C24 の MV 値と同じ 35.0 になった場合です。

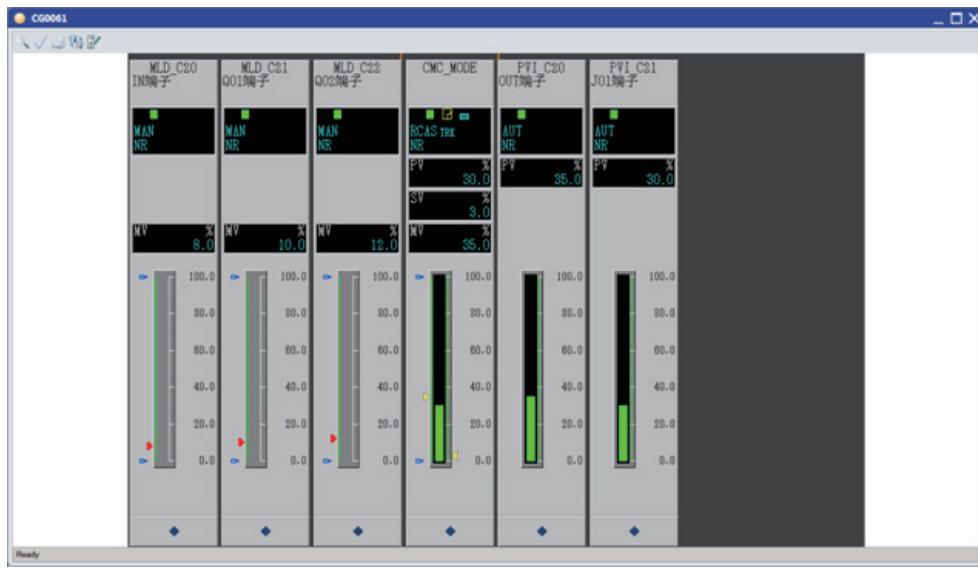


図 ト ラッキ ン グ動 作

MLD_C24 の MV を変更して、CMC_MODE の MV と PVI_C20 の PV が追従するのを確認してください。また、トラッキングスイッチ SW_C20 の PV を変更して動作を確認してください。

■ ブロックモードごとの動作

これまで、ブロックモードが AUT、CAS と RCAS の動作、およびトラッキング動作について説明してきました。この他のブロックモードにおける動作を以下に示します。

表 CMC_MODEのブロックモードごとの動作

ブロックモード	PV値とP01の作成	制御演算	OUT端子出力値	J01端子出力値
O/S	しない	しない	出力しない	出力しない
IMAN	作成する	しない	OUT 端子から読み返したデータをそのまま出力 (*1)	MV01 に保持されている(前回と同じ)値を出力
TRK	作成する	しない	TIN 端子 (MLD_C24.MV を結合) から入力したデータを出力	MV01 に保持されている(前回と同じ)値を出力
MAN	作成する	しない	MV 値をそのまま出力	MV01 に保持されている(前回と同じ)値を出力
AUT	作成する	演算する	「SV 値 × PV」を出力	「SV 値 × P01」を出力
CAS	作成する	演算する	「SET 端子から設定される値 × PV」を出力 (SET 端子には MLD_C23.MV を結合)	「SET 端子から設定される値 × P01」を出力 (SET 端子には MLD_C23.MV を結合)
PRD	作成する	しない	SET 端子から設定される値をそのまま出力 (SET 端子には MLD_C23.MV を結合)	MV01 に保持されている(前回と同じ)値を出力
RCAS	作成する	演算する	「RSV 値 × PV」を出力 (RSV には MLD_C25.MV を結合)	「RSV 指定値 × P01」を出力 (RSV には MLD_C25.MV を結合)
ROUT	作成する	しない	RMV 値をそのまま出力 (RMV には MLD_C26.MV を結合)	MV01 に保持されている(前回と同じ)値を出力

*1：ビルダ定義項目「出力トラッキング」の指定により、MV 値を出力することもできます。

参照 出力トラッキングの指定の詳細については、以下を参照してください。
[「6.2.7 出力処理（ユーザ記述）」](#)

各ブロックモードの動作を確認できるように CMC_MODE の端子やデータアイテムに手動操作ブロックからデータを設定できるようにしてありますので、動作を確認してください。

6.2.1 ブロックモード遷移のUcaCtrlHandlerによる処理

連続制御形ユーザカスタムブロックのブロックモード遷移に伴う処理は、ユーザカスタムアルゴリズム作成用ライブラリのUcaCtrlHandlerが行います。ユーザは入力処理、制御演算処理、出力処理を3つの関数に記述します。それ以外の処理は、システムのUcaCtrlHandlerがすべて行います。ユーザが記述する3つの関数の主な処理内容を示します。

- ・入力処理の関数
データを入力し PV 値を作成します。
- ・制御演算処理の関数
PV 値や SV 値などより制御演算を行い、出力値を計算します。
- ・出力処理の関数
OUT 端子からデータを出力します。

サンプルプログラム mode.c を UcaBlockPeriodical で検索して機能ブロック定周期処理を見つけてください。UcaBlockPeriodical は、ユーザカスタムアルゴリズム作成用ライブラリの関数 UcaCtrlHandler を呼び出しています。UcaCtrlHandler の引数に3つの関数、mode_input（入力処理）、mode_control（制御演算）、mode_output（出力処理）を指定しています。

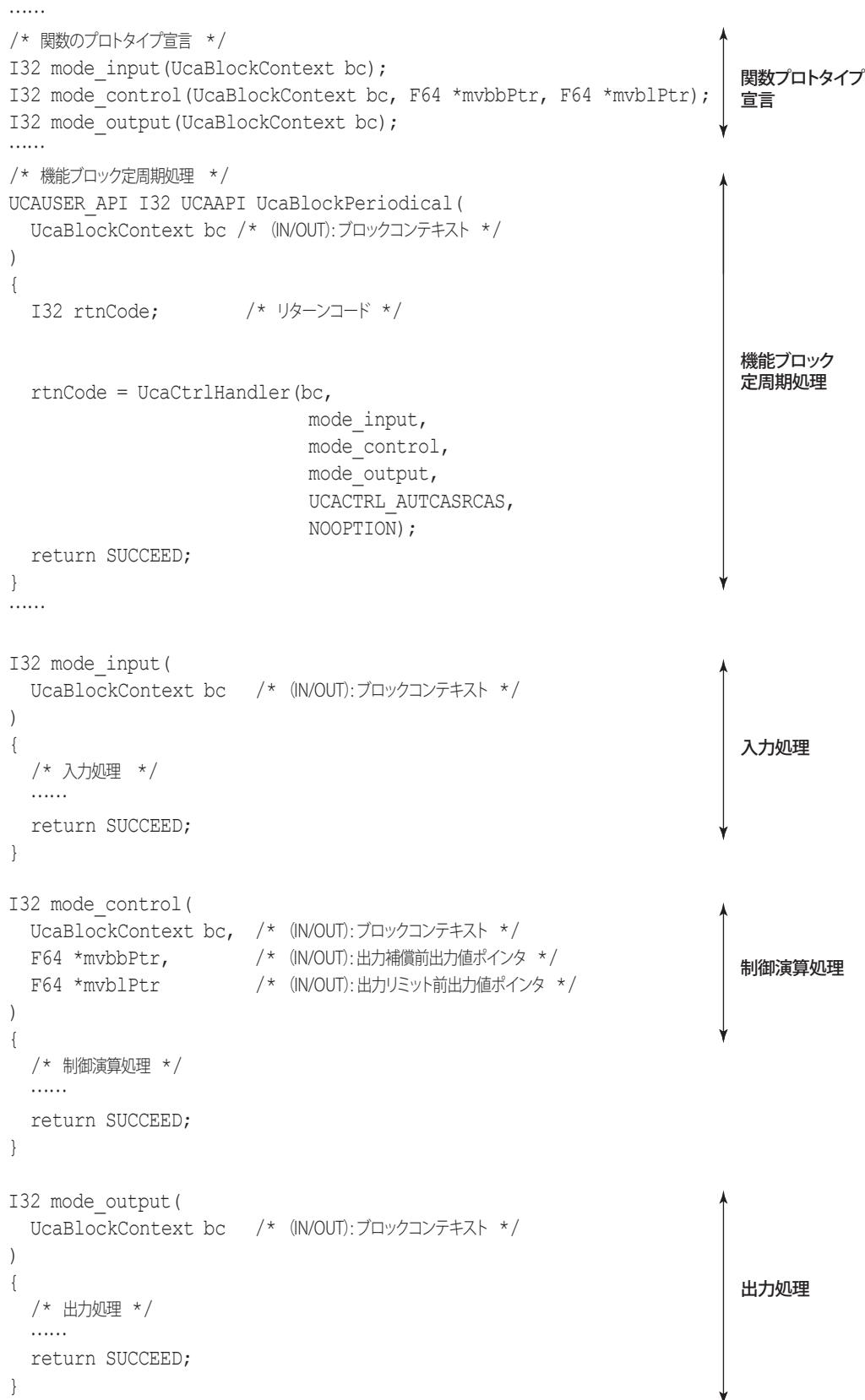
```
/*
* <<FNH>>*****
*
* Function name:          UcaBlockPeriodical
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME   処理続行不能
*
* description:            機能ブロック定周期処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode; /* リターンコード */

    rtnCode = UcaCtrlHandler(bc,
                           mode_input,
                           mode_control,
                           mode_output,
                           UCACTRL_AUTCASCAS,
                           NOOPTION);

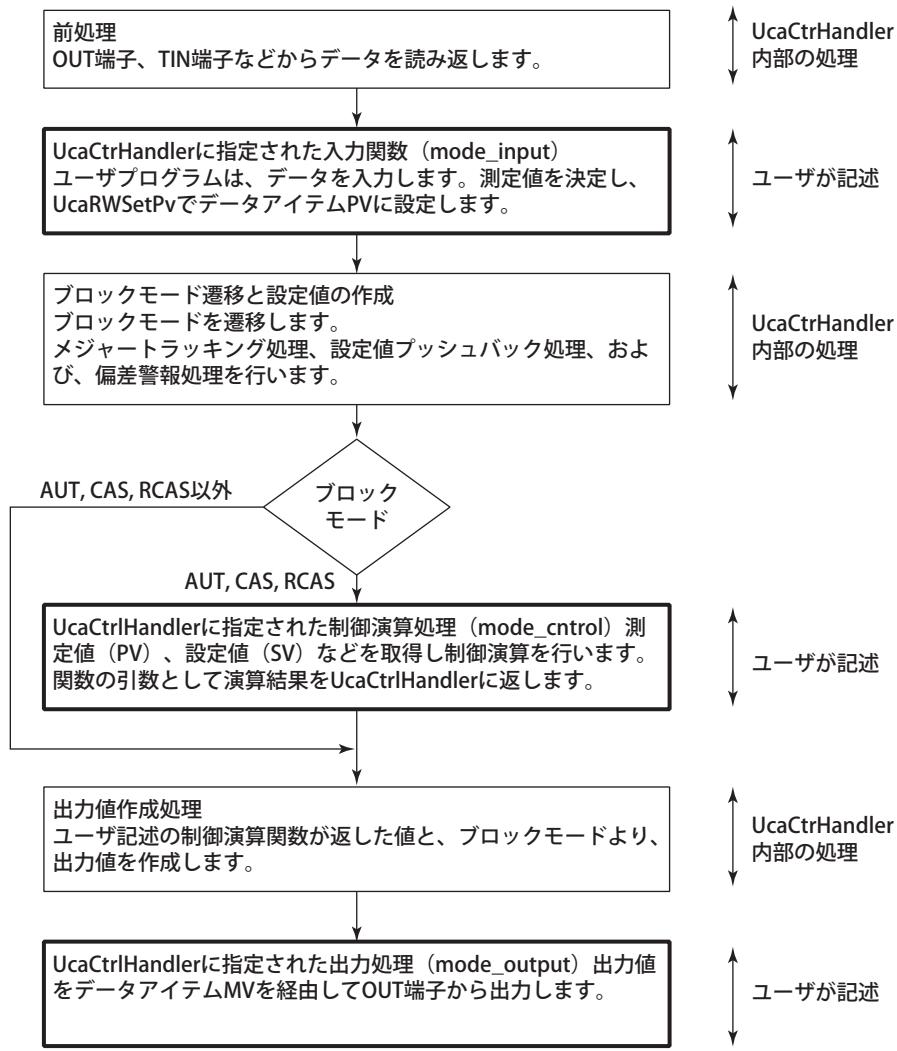
    return SUCCEED;
}
```

■ ソースファイルの構成とUcaCtrlHandlerの動作

次図は、ソースファイル mode.c の構成です。ソースファイルの下の方に、mode_input(入力処理)、mode_control(制御演算)、mode_output(出力処理) の 3 つの関数があります(コメントなどは省略してあります)。



UcaCtrlHandler には 3 つの関数 mode_input、mode_control、mode_output が指定されています。また 3 つの関数名の次に UCACTRL_AUTCASCAS を指定していますので、UcaCtrlHandler はブロックモードが AUT、CAS または RCAS のときに、制御演算関数 mode_control を呼び出します。UcaCtrlHandler は、次図のように動作します。



060221J.ai

図 UcaCtrlHandler処理の流れ

UcaCtrlHandler は、まず前処理として OUT 端子や TIN 端子など端子の結合先からデータを読み返します。前処理が完了すると、UcaCtrlHandler は引数に指定されたユーザ記述の入力処理関数 (mode_input) を呼び出します。入力処理関数は測定値 PV を作成します。つまり入力したデータより測定値を計算し、測定値を UcaRWSetPv でデータアイテム PV に設定します。

入力関数が return すると、UcaCtrlHandler はブロックモードを遷移します。このとき条件が成立すれば、UcaCtrlHandler は AUT フォールバックや MAN フォールバック処理を行います。ブロックモードの遷移が完了すると、UcaCtrlHandler は測定値トラッキング（メジャートラッキング）と設定値プッシュバック処理をします。これらの処理により設定値 (SV) が決まります。

次に、UcaCtrlHandler は引数に指定されたユーザ記述の制御演算関数 (mode_control) を呼び出します。制御演算関数は測定値 PV や設定値 SV を取得し、制御演算を実行します。ユーザ記述の制御演算関数は、引数に演算結果の出力値を返します。

UcaCtrlHandler は、引数に返された演算結果の出力値とブロックモードを元に出力値を決定します。UcaCtrlHandler は、引数に指定されたユーザ記述の出力処理関数 (mode_output) を呼び出します。出力処理関数は、出力値をデータアイテム MV を経由して OUT 端子から出力します。

UcaCtrlHandler によるこのような処理によりブロックモード遷移や出力値決定処理などの複雑な処理はシステムに任せ、ユーザは入力処理、制御演算処理、出力処理を自由に記述することができます。

6.2.2 前処理 (UcaCtrlHandlerの内部で処理)

UcaCtrlHandlerが行う前処理について説明します。

前処理では、CSTM-C の端子からデータを読み込みます。このように前処理で準備をしてあるので、入力処理、制御演算処理、出力処理のユーザプログラムは端子からのデータ読み返しを意識する必要がなくなります。通常ユーザは、前処理が実行されていることを特に意識する必要はありません。前処理が行う端子からの読み込み処理を以下に示します。

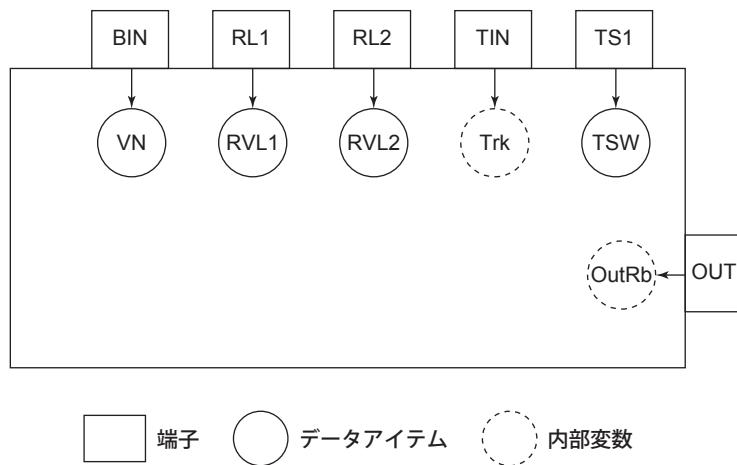


図 UcaCtrlHandlerの前処理

前処理の処理内容を以下に示します。

表 UcaCtrlHandlerの前処理

端子	データ格納先	内部で使用している関数	説明
TIN 端子	内部変数 Trk	UcaRWReadTrack	トラッキング信号入力とトラッキングスイッチを読み取ります。
TSI 端子	データアイテム TSW		
OUT 端子	内部変数 OutRb (OUT 端子読み返し値)	UcaRWReadbackMv	OUT 端子からデータを読み返します。
BIN 端子	データアイテム VN	UcaRWReadBin	BIN 端子から補償入力を読み取ります。
RL1 端子	データアイテム RVL1	UcaRWReadRI	リセット信号 1 入力とリセット信号 2 入力を読み取ります。
RL2 端子	データアイテム RVL2		

6.2.3 入力処理（ユーザ記述）

入力処理は、ユーザが記述します。ユーザプログラムは、データを入力しPV値を作成します。計算したPV値は、UcaRWSetPvでデータアイテムPVに設定します。

参照 入力処理の書き方の詳細については、以下を参照してください。
[「6.1 多入力 CSTM-C（ブロックモードは AUT と O/S に限定）」](#)

入力処理は CSTM-C のブロックモードによらず、いつも実行されます。つまり、CSTM-C に処理タイミングが与えられれば、ユーザプログラムの UcaRWSetPv が呼び出されてデータアイテム PV が更新されます。入力処理の関数 mode_input について説明します。

```
/*
*<<FNH>>*****
*
* Function name:      mode_input
* Return value:       SUCCEED 正常終了
*
* description:        入力処理
*
*>>HNF<<*****
*/
I32 mode_input(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    F64S pv;           /* PV */
    F64S rv;           /* RV */
    F64S rv01;         /* RV01 */
    F64S rv02;         /* RV02 */
    F64S p01;          /* P01 */
    I32 rtnCode;        /* リターンコード */
    I32 rtnReadIn;     /* IN 端子入力リターンコード */

    /* ===== 入力処理 ===== */
    rtnReadIn = UcaRWReadIn(bc, NOOPTION); /* IN 端子から RVへ読み込み */

    rtnCode = UcaRWRead(bc, 1, NOOPTION); /* Q01 端子から RV01へ読み込み */
    rtnCode = UcaRWRead(bc, 2, NOOPTION); /* Q02 端子から RV02へ読み込み */

    /* ===== PV 値作成 ===== */
    /* 入力したデータをデータアイテムから変数に読み込み */
    rtnCode = UcaDataGetRv(bc, &rv, NOOPTION); /* RV */
    rtnCode = UcaDataGetRvn(bc, &rv01, 1, 1, NOOPTION); /* RV01 */
    rtnCode = UcaDataGetRvn(bc, &rv02, 2, 1, NOOPTION); /* RV02 */
```

(続く)

(続き)

```

/*
 * 演算を実行: PV=RV,RV01,RV02 の合計
 *             P01=RV,RV01,RV02 の平均
 */

/* 3入力の合計を計算しデータアイテム PVに設定 */
pv.value = rv.value + rv01.value + rv02.value;
pv.status = 0;          /* データステータス正常 */

rtnCode = UcaRWSetPv(bc, &pv, rtnReadIn, NOOPTION);

/* 3入力の平均を計算しデータアイテム P01に設定 */
p01.value = (rv.value + rv01.value + rv02.value) / 3.0;
p01.status = 0;          /* データステータス正常 */

rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);

return SUCCEED;
}

```

この入力処理関数では、3つの入力の合計を UcaRWSetPv でデータアイテム PV に設定しています。また、3入力の平均をデータアイテム P01 に設定しています。データアイテム P01 に設定された平均値は、制御演算処理を行う mode_control で使用します。このように、入力処理で作成した中間データを演算処理で使用する場合には、データアイテム P01 ~ P32 を使用してください。

もう一度入力関数のインターフェースを確認してください。

```

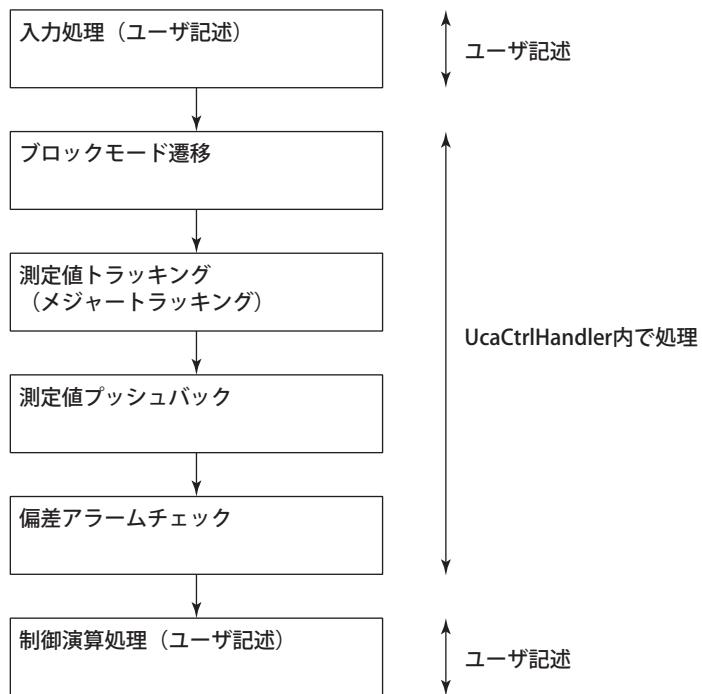
...
I32 mode_input(
    UcaBlockContext bc           /* (IN/OUT) : ブロックコンテキスト */
)
...

```

入力関数の引数は、ブロックコンテキスト (bc) のみです。このインターフェースは固定です。入力関数の引数を変えることはできません（インターフェースは固定ですが、関数名 mode_input と変数名 bc は別の名前に変更可能です）。

6.2.4 ブロックモード遷移と設定値の作成（UcaCtrlHandlerの内部で処理）

ユーザ記述の入力処理が終了すると、UcaCtrlHandlerはブロックモードを遷移し、設定値プッシュバック処理を行います。ユーザ記述の入力処理と制御演算処理の間に、UcaCtrlHandlerが処理する内容を以下に示します。



060224J.ai

図 UcaCtrlHandlerの処理（部分）

■ ブロックモード遷移

ブロックモード遷移処理では、初期化手動や MAN フォールバックなどの条件を検査し、条件が成立するとブロックモードの遷移を実行します。

参照 UcaCtrlHander のブロックモード遷移処理は、PID 調節ブロックなどの標準の調節ブロックと同じです。初期化手動条件や MAN フォールバック条件など、動作の詳細については、以下を参照してください。

[機能ブロッククリファレンス Vol.1 \(IM 33J15A30-01JA\) 「1.4 調節ブロックに共通の制御演算処理」](#)

● 初期化手動

初期化手動は、ブロックモードを初期化手動 (IMAN) モードに変更して一時的に制御動作を中断させる機能です。初期化手動は、初期化手動条件が成立したときに動作します。

● MANフォールバック

MAN フォールバックは、ブロックモードを手動 (MAN) モードにして制御を強制的に停止させる異常処理機能です。MAN フォールバック条件が成立したときに動作します。

● AUTフォールバック

AUT フォールバックは、AUT フォールバック条件成立時に、ブロックモードをカスケード (CAS) モードまたはプライマリダイレクト (PRD) モードから自動 (AUT) モードに変更して、オペレータ設定値を使用した制御動作に切り換える異常処理機能の一つです。AUT フォールバックの有無は、CSTM-C の機能ブロック詳細定義ビルダで指定します (デフォルトは「なし」)。

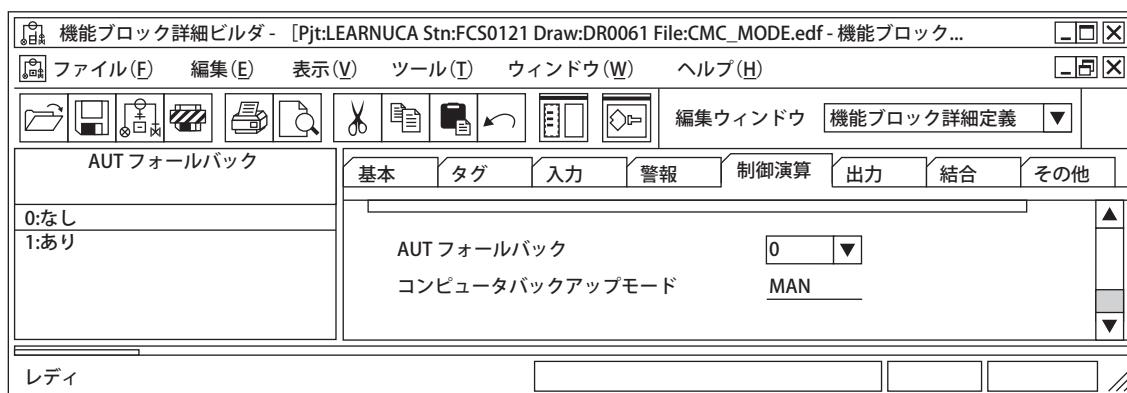
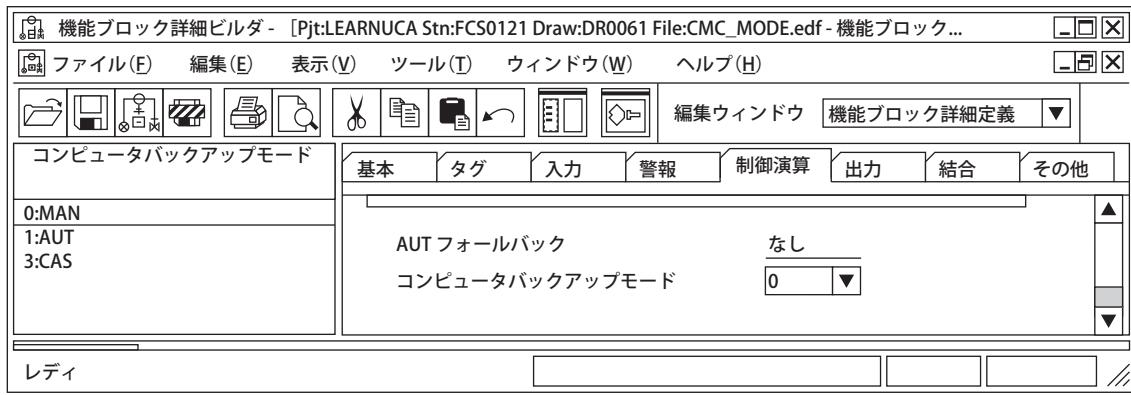


図 AUTフォールバックの指定

● コンピュータフェイル

コンピュータフェイルは、リモートカスケード (RCAS) モードまたはリモート出力 (ROUT) モードの動作を一時中断して、コンピュータバックアップモードに切り換える機能です。コンピュータフェイル条件が成立したときに動作します。コンピュータバックアップモードの設定は、CSTM-C の機能ブロック詳細ビルダで指定します（デフォルトは「MAN」）。



060226J.ai

図 コンピュータバックアップモードの設定

● ブロックモード変更インタロック

ブロックモード変更インタロックとは、自動運転中の機能ブロックの制御演算処理を停止させ、同時に現在停止中の機能ブロックに対してブロックモードを自動運転に変更されるのを禁止するための機能です。ブロックモード変更インタロック処理では INT 端子からデータを読み込みます。そして、ブロックモード変更インタロック条件が成立したとき、つまり INT 端子の結合先のスイッチが ON (1) になったときにブロックモード変更インタロック処理が動作します。

● トラッキング

トラッキングスイッチ TSW の状態に従ってトラッキング状態を設定します。TSW がオン (1) なら基本ブロックモード TRK を設定します。TSW がオフ (0) なら TRK をリセットします。

■ 測定値トラッキング（メジャートラッキング）

測定値トラッキングは、手動（MAN）モードから自動（AUT）モードへ切り換えたときの急激な操作出力値（MV）の変動を抑えるために、手動（MAN）モード時は設定値（SV）を測定値（PV）に一致させる機能です。また、自動モードかつ下位側とのカスケード結合がオープン（AUT（IMAN））、またはカスケードモードかつ下位側とのカスケード結合オープン（CAS（IMAN））のときに、設定値（SV）と測定値（PV）を一致させることもできます。

参照 UcaCtrlHandler の測定値トラッキング処理は、PID 調節ブロックなどの標準の調節ブロックと同じです。動作の詳細については、以下を参照してください。

[機能ブロッククリアレンス Vol.1 \(IM 33J15A30-01JA\) 「1.4 調節ブロックに共通の制御演算処理」の「■ 測定値トラッキング（メジャートラッキング）」](#)

測定値トラッキングの指定は、CSTM-C の機能ブロック詳細ビルダで指定します（以下はデフォルトです）。

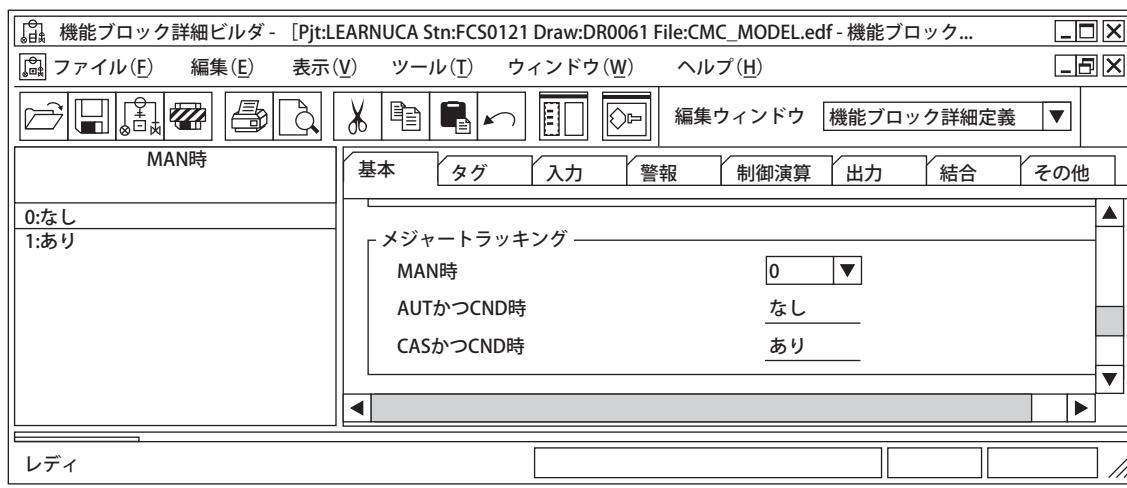
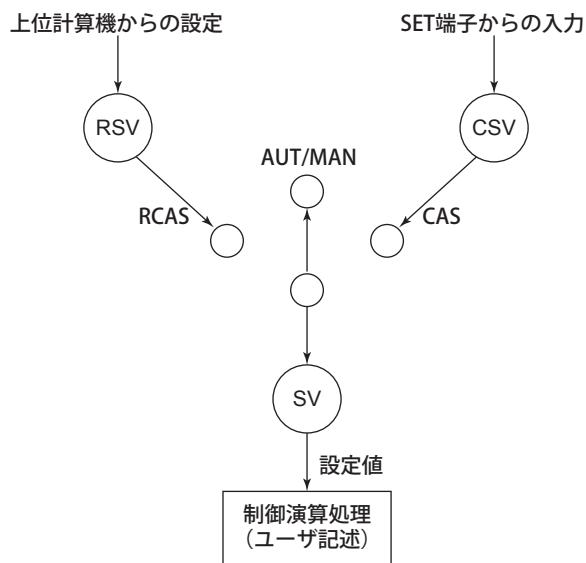


図 測定値トラッキングの指定

UcaCtrlHandler は、ブロックモードが PRD から AUT、CAS または RCAS に切り換わったときに、メジャートラッキングを行います。これはビルダ定義項目には関係なく、常時行われます。

■ 設定値プッシュバック

設定値プッシュバックは、3種の設定値を同じ値にする機能です。設定値 (SV)、カスケード設定値 (CSV)、およびリモート設定値 (RSV) の関係を以下に示します。



060228J.ai

図 設定値プッシュバック

● 自動 (AUT) モードまたは手動 (MAN) モードの場合の動作

カスケード設定値 (CSV) およびリモート設定値 (RSV) に、設定値 (SV) と同じ値を設定します。設定値 (SV) に対して機能ブロック外部からデータ設定が行われた場合でも、自動的にカスケード設定値 (CSV) およびリモート設定値 (RSV) に同じ値が設定されます。

● カスケード (CAS) モードの場合の動作

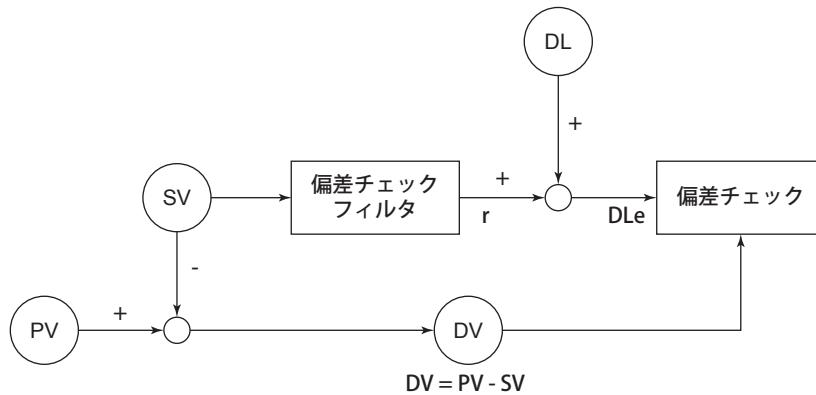
設定値 (SV) およびリモート設定値 (RSV) を、カスケード設定値 (CSV) に一致させます。

● リモートカスケード (RCAS) モードの場合の動作

設定値 (SV) およびカスケード設定値 (CSV) を、リモート設定値 (RSV) に一致させます。

■ 偏差アラームチェック

偏差アラームチェックは、測定値（PV）と設定値（SV）との偏差（ $DV = PV - SV$ ）の絶対値が偏差アラーム設定値（DL）の絶対値を超えているかどうかを判定する機能です。超えている場合には、偏差が正方向であれば正方向偏差アラーム（DV+）が発生します。また、偏差が負方向であれば負方向偏差アラーム（DV-）が発生します。



060229J.ai

図 偏差アラームチェック

参照 UcaCtrlHander の偏差アラーム処理は、標準ブロックと同じです。動作の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「5.6 偏差アラームチェック」](#)

6.2.5 制御演算処理（ユーザ記述）

ブロックモード遷移や測定値プッシュバック処理が完了すると、UcaCtrlHandlerはユーザ定義の制御演算処理関数を呼び出します。

mode.c の mode_control 関数について説明します。

```

/*
* <<FNH>>*****
*
* Function name: mode_control
* Return value: SUCCEED           正常終了
*
* description: 制御演算処理
*
*>>HNF<<*****
*/
I32 mode_control(
    UcaBlockContext bc,          /* (IN/OUT) : ブロックコンテキスト */
    F64 *mvbbPtr,               /* (IN/OUT) : 出力補償前出力値ポインタ */
    F64 *mvblPtr                /* (IN/OUT) : 出力リミット前出力値ポインタ */
)
{
    F64S pv;                   /* PV */
    F32S sv;                   /* SV */
    F64S p01;                  /* P01 */
    F64S mv01;                 /* MV01 */
    I32 rtnCode;                /* リターンコード */

    /* データアイテムからデータを取得 */
    rtnCode = UcaDataGetPv(bc, &pv, NOOPTION);           /* PVは入力の合計 */
    rtnCode = UcaDataGetSv(bc, &sv, NOOPTION);           /* SV */
    rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);   /* P01は入力の平均 */

    /*
     * 演算を実行： MVBB = SV * PV
     *             MVBL = SV * PV
     *             MV01 = SV * P01
     */
    /* 計算結果を MVBB,MVBL に保存 */
    *mvbbPtr = sv.value * pv.value;
    *mvblPtr = *mvbbPtr;

    /* 計算結果を MV01 に保存 */
    mv01.value = sv.value * p01.value;
    mv01.status = 0;           /* データステータス正常 */
    rtnCode = UcaDataStoreF64SToMvn(bc, &mv01, 1, NOOPTION);

    return SUCCEED;
}

```

制御演算処理関数のインタフェース部分を見てください。

```
.....
I32 mode_control(
    UcaBlockContext bc,           /* (IN/OUT) : ブロックコンテキスト */
    F64 *mvbbPtr,                /* (IN/OUT) : 出力補償前出力値ポインタ */
    F64 *mvblPtr                /* (IN/OUT) : 出力リミット前出力値ポインタ */
)
.....
```

制御演算関数には、「ブロックコンテキスト」「出力補償前出力値ポインタ」「出力リミット前出力値ポインタ」の3つの引数があります。この引数は固定であり、変更することはできません（インターフェースは固定ですが、関数名 mode_control と変数名 bc, mvbbPtr, mvblPtr は、別の名前に変更することができます）。

ユーザは、制御演算により操作出力値を決定し、演算結果を「出力補償前出力値ポインタ」「出力リミット前出力値ポインタ」が差す領域に設定します。出力リミット前出力値は、出力補償をした後の操作出力値です。出力リミット前出力値は、データアイテム MV を経由して OUT 端子から出力される出力値の元になるデータです。出力値の関係を以下に示します。

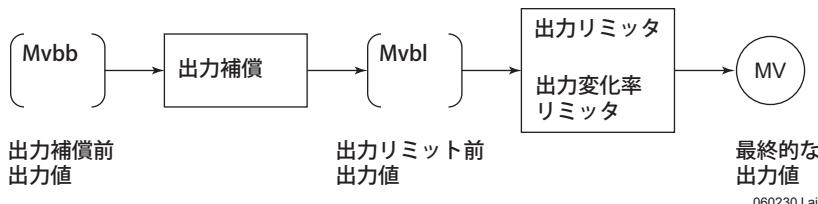


図 出力値の関係

■ 出力補償前出力値

出力補償前出力値は、出力補償をする前の操作出力値です。出力補償前出力値は、補助出力 (SUB 端子) から操作出力値 (MV) または操作出力変化量 (Δ MV) を出力するために使用されます。

参照 補助出力の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「4.9 補助出力」の「■ 連続制御ブロックにおける補助出力」](#)

■ 出力リミット前出力値

出力リミット前出力値は、出力補償をした後の操作出力値です。出力リミット前出力値は、最終的に OUT 端子から出力される出力値の元になります。ユーザ記述の出力処理で呼び出す UcaRWWriteMvToOutSub 関数は、出力リミット前出力値に対して出力リミッタや出力変化率リミッタの処理を行い、最終的な出力値を作成します。この最終的な出力値がデータアイテム MV に設定され、OUT 端子から出力されます。

補足 この節のサンプルは、ブロックモードごとの動作を説明することを目的としていますので、「出力補償」はしていません。「出力補償」をしないので、「出力補償前出力値」と「出力リミット前出力値」は同じ値です。

参照 出力補償の詳細については、以下を参照してください。

[「6.3 制御演算関数を使用した CSTM-C」](#)

制御演算関数 mode_control は、データアイテム PV、SV、P01 からデータを取り出し、次の計算をします。操作出力値は「SV × PV」です。出力補償はしませんので、「出力補償前出力値ポインタ (mvbbPtr)」と「出力リミット前出力値ポインタ (mvblPtr)」が指す領域に、同じ計算結果を設定します。

```
.....
/* 計算結果を MVBB,MVBL に保存 */
*mvbbPtr = sv.value * pv.value;
*mvblPtr = *mvbbPtr;
.....
```

J01 端子から出力するデータは、データアイテム MV01 に設定します。MV01 には、「データアイテム P01 × SV」を設定します。P01 には、入力処理 (mode_input) で 3 つの入力の平均値が設定されています。このように、入力処理と制御演算処理の間のデータのやり取りには、データアイテム P01～P32 を使用してください。

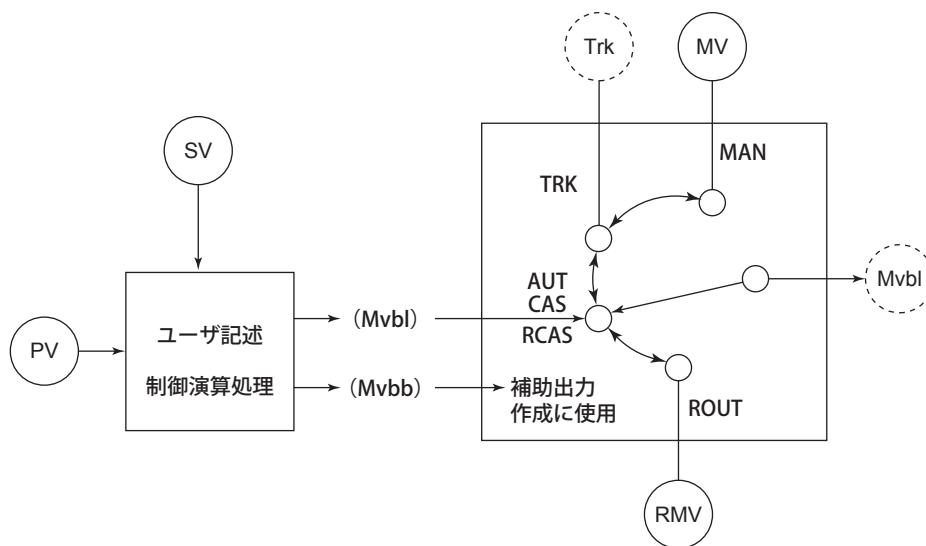
```
.....
/* 計算結果を MV01 に保存 */
mv01.value = sv.value * p01.value;
mv01.status = 0;           /* データステータス正常 */
rtnCode = UcaDataStoreF64SToMvn(bc, &mv01, 1, NOOPTION);
.....
```

ユーザ記述の制御演算関数が return すると、UcaCtrlHandler は「出力補償前出力値ポインタ」と「出力リミット前出力値ポインタ」が指す領域に設定されたデータより、出力値の作成処理をします。

6.2.6 出力値作成処理（UcaCtrlHandlerの内部で処理）

UcaCtrlHandlerの出力値作成処理は、ブロックモードに応じて出力リミット前出力値を決定し、内部変数Mvblに格納します。ブロックモードが、AUT、CAS、またはRCASであれば、ユーザ記述の制御演算処理が引数に返した出力リミット前出力値（Mvbl）として返した値が、内部変数Mvblに格納されます。

ユーザ記述の制御演算処理が引数に返した出力補償前出力値（Mvbb）は、補助出力（SUB端子）から、操作出力値（MV）または操作出力変化量（ΔMV）を出力するために使用されます。UcaCtrlHandlerは、出力補償前出力値を決定し内部変数Mvbbに格納します。



060231J.ai

図 UcaCtrlHandlerの出力リミット前出力値作成処理

ブロックモードによる出力リミット前出力値作成処理は以下のとおりです。

- ・ 基本ブロックモード TRK（トラッキング）が成立している場合には TIN 端子からの入力データ（内部変数 Trk）を出力リミット前出力値とし、内部変数 Mvbl に格納します。
- ・ ブロックモードが ROUT（リモート出力）のときはデータアイテム RMV に設定されているデータ（上位コンピュータから設定します）を出力リミット前出力値とし、内部変数 Mvbl に格納します。
- ・ ブロックモードが MAN（手動）のときはデータアイテム MV に設定されているデータを出力リミット前出力値とし、内部変数 Mvbl に格納します。
- ・ ブロックモードが PRD（プライマリダイレクト）のときはデータアイテム CSV に設定されているデータ（カスケード上流ブロックから設定します）を出力リミット前出力値とし、内部変数 Mvbl に格納します。CSV のデータは UcaCtrlHandler の測定値 プッシュバックによりデータアイテム SV に反映されていますので、(UcaCtrlHandler 内部の処理では) データアイテム SV のデータを使用します。
- ・ それ以外 (AUT、CAS、RCAS) のときはユーザ定義の制御演算処理で計算した Mvbl (出力リミット前出力値) を出力リミット前出力値とし、内部変数 Mvbl に格納します。

補足 基本ブロックモード IMAN（初期化手動）が成立している場合の出力値は、UcaCtrlHandler の出力値決定処理ではなく UcaRWWriteMvToOutSub 関数により行われます。

この処理で作成された出力リミット前出力値（内部変数 Mvbl）は、ユーザ記述の出力処理から呼び出される UcaRWWriteMvToOutSub によりデータアイテム MV を経由して OUT 端子から出力されます。UcaRWWriteMvToOutSub は、（本処理で決定した）出力リミット前出力値に対して出力リミットや出力変化率リミッタなどの処理した結果を OUT 端子から出力します。

6.2.7 出力処理（ユーザ記述）

ユーザ記述の出力処理について説明します。

出力処理は、mode.c の mode_output 関数です。

```
/*
*<<FNH>>*****
*
* Function name: mode_output
* Return value: SUCCEED 正常終了
*
* description: 出力処理
*
*>>HNF<<*****
*/
I32 mode_output(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode; /* リターンコード */

    /* OUT 端子と SUB 端子出力処理 */
    rtnCode = UcaRWWriteMvToOutSub(bc, NOOPTION);

    /* Jnn 端子出力処理 */
    rtnCode = UcaRWWrite(bc, 1, NOOPTION); /* M01 から J01 端子へ書き込み */

    return SUCCEED;
}
```

出力関数のインターフェースは以下の部分です。

```
.....
I32 mode_output(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
.....
```

出力関数の引数は、ブロックコンテキスト (bc) のみです。このインターフェースは固定です。出力関数の引数を変えることはできません（インターフェースは固定ですが、関数名 mode_output と変数名 bc は別の名前に変更可能です）。

出力処理の最初で、UcaCtrlHandler が出力作成処理で内部変数 Mvbl に格納した出力リミット前出力値を、UcaRWWWriteMvToOutSub によりデータアイテム MV を経由して OUT 端子から出力します。

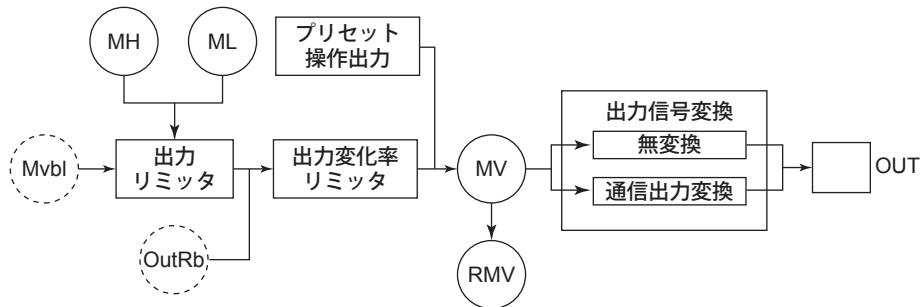
```
.....
/* OUT 端子と SUB 端子出力処理 */
rtnCode = UcaRWWWriteMvToOutSub(bc, NOOPTION);
.....
```

UcaRWWWriteMvToOutSub は、以下の処理をします。

- ・ 出力値リミッタ
- ・ 出力変化率リミッタ
- ・ プリセット操作出力
- ・ 出力信号変換
- ・ 出力クランプ
- ・ 出力値トラッキング
- ・ 補助出力

参照 出力処理の動作の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「4. 出力処理」の「■ 連続制御ブロックに共通の出力処理」](#)



060232J.ai

図 UcaRWWWriteMvToOutSubの出力処理

出力値トラッキングについて説明します。ユーザカスタムブロックのビルダ定義項目「出力値トラッキング」に「あり」または「なし」を指定します（デフォルトは「あり」です）。出力値トラッキングの指定により、基本ブロックモード IMAN が成立しているときの出力値が決まります。

「あり」： 出力値を OUT 端子からの読み返し値（つまり OUT 端子結合先のデータ）と一致させます。

「なし」： 出力値を MV 値と一致させます。

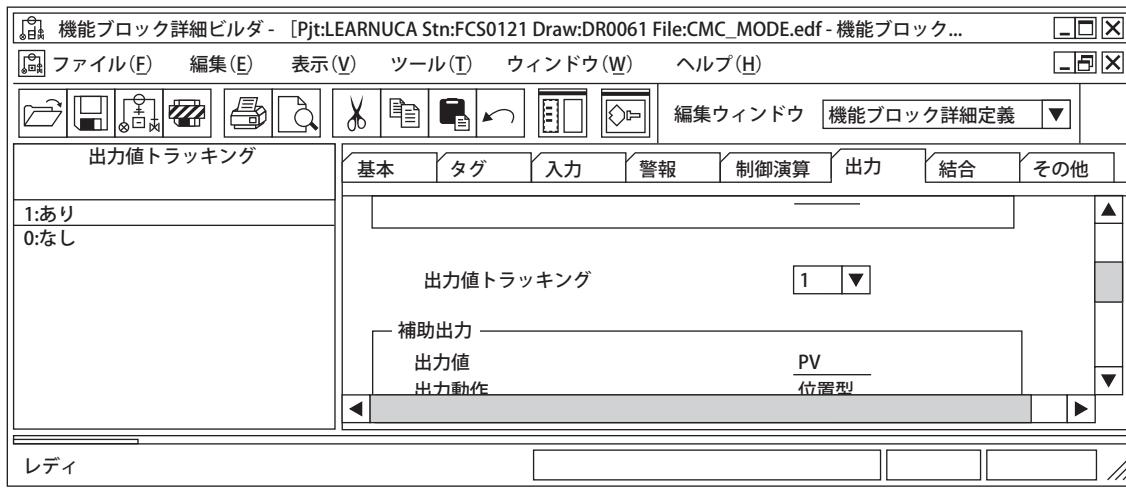


図 出力値トラッキングの指定

UcaRWWriteMvToOutSub は、UcaCtrlHandler が内部変数 Mvbb に設定した出力補償前出力値を使用して、SUB 端子からのデータ出力も行います。出力する内容は、CSTM-C に対して機能ブロック詳細ビルダで指定します。



図 出力内容の指定

mode_output は、制御演算処理 mode_control がデータアイテム MV01 に設定したデータを、UcaRWWrite により J01 端子から出力します。

```
.....
/* Jnn 端子出力処理 */
rtnCode = UcaRWWrite(bc, 1, NOOPTION); /* M01 から J01 端子へ書き込み */
.....
```

6.2.8 ブロックモード遷移付きのCSTM-Cにおけるデータの流れ

CSTM-Cのブロックモード遷移について説明します。

以下に CSTM-C におけるデータの流れを示します。

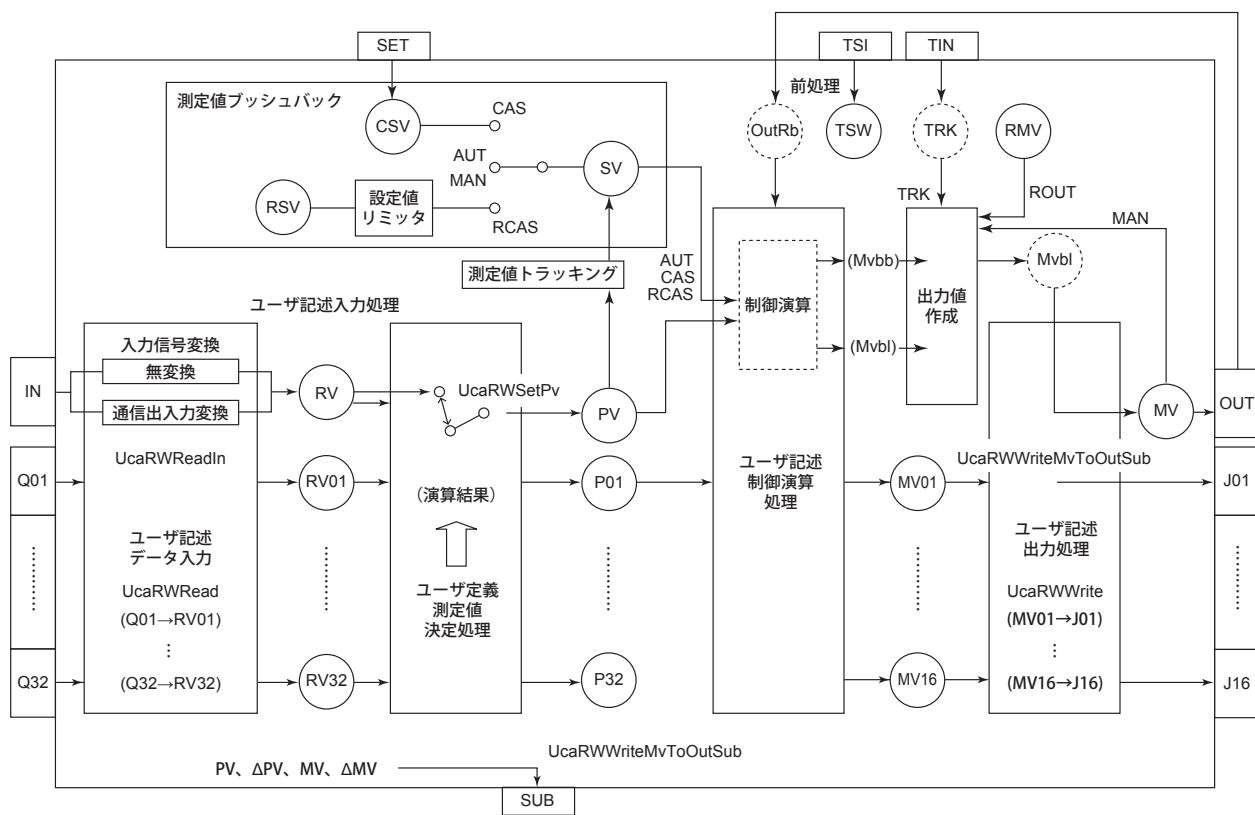


図 CSTM-Cにおけるデータの流れ

060235J.ai

UcaCtrlHandler を使ったブロックモード遷移を伴う CSTM-C の動作は、次のようになります。

1. UcaCtrlHandler は、前処理として OUT 端子や TIN 端子の結合先からデータを読み取ります。
2. ユーザ記述の「入力処理」はデータを入力し、測定値を計算し計算結果を UcaRWSetPv によりデータアイテム PV に設定します。制御演算処理に渡したいデータがあれば、データアイテム P01 ~ P32 に設定しておきます。
3. UcaCtrlHandler は測定値トラッキングや設定値プッシュバック処理を行い、測定値 (PV) と設定値 (SV) を決定します。
4. ユーザ記述の「制御演算処理」は制御演算を行い、出力補償前出力値 (Mvbb) と出力リミット前出力値 (Mvbl) を作成します。J01 ~ J16 端子から出力するデータがあれば、データアイテム MV01 ~ MV16 に設定しておきます。
5. UcaCtrlHandler は、ブロックモードに従い出力値を決定します。
6. ユーザ記述の「出力処理」は、UcaRWWriteMvToOutSub により出力値をデータアイテム MV を経由して OUT 端子から出力します。また、必要に応じて MV01 ~ MV16 のデータを J01 ~ J16 端子より出力します。

6.3 制御演算関数を使用したCSTM-C

この節ではPI制御を行う連続制御形ユーザカスタムブロックをサンプルとし、制御演算関数の使い方を説明します。制御演算関数の使い方を説明するためにPI制御を例にしています。

ユーザカスタムアルゴリズム作成用ライブラリには、制御演算処理を行う関数が用意されています。

表 制御演算関数

機能	関数名	説明
制御ホールド	UcaCtrlHoldCheck	制御ホールド要求が設定されているか検査します。
リセットリミット	UcaCtrlResetLimitOprt	リセットリミット機能を実行します。
不感帯動作	UcaCtrlDeadband	不感帯動作を実行します。
入出力補償	UcaCtrlCompensation	入力補償または出力補償を実行します。
制御演算初期化	UcaCtrlInitCheck	制御演算の初期化要求が設定されているか検査します。
	UcaCtrlInitClear	制御演算の初期化要求を解除します。
	UcaCtrlFuncInit	制御演算関数が保持する内部パラメータを初期化します。

また、制御演算に関するビルダ定義項目の指定値を取得する関数が用意されています。

表 制御演算に関するビルダ項目を取得する関数

ビルダ定義項目	関数名	説明（（*）印はデフォルト）
制御動作方向	UcaConfigDirection	「逆動作（*）」「正動作」から指定値を取得
制御／演算出力動作	UcaConfigOutput	「位置形（*）」「速度形」から指定値を取得
入出力補償	UcaConfigCompensation	「なし（*）」「入力補償」「出力補償」から指定値を取得
不感帯動作	UcaConfigDeadband	「なし（*）」「あり」から取得

ユーザ記述の制御演算処理では、これらの関数を使用することにより、ユーザカスタムアルゴリズムに入出力補償や不感帯動作を伴う制御演算を記述することができます。この節では、PI制御を行うユーザカスタムアルゴリズムを例に、これらの関数の使い方を説明します。

このサンプルは、制御の周期にユーザカスタムブロックの実効スキャン周期を使用します。ビルダ定義項目の「制御周期」の指定は無視します。つまり、ユーザカスタムブロックは実効スキャン周期ごとにOUT端子から操作出力値を出力します（制御周期を使用する場合は実効スキャン周期ごとにPV値を作成しますが、OUT端子からの出力は制御周期でのみ行います）。

参照 実効スキャン周期と制御周期の詳細については、以下を参照してください。

[「6.4.2 実効スキャン周期と制御周期」](#)

PI 制御アルゴリズムは、比例 (P) 積分 (I) の制御動作が組み合わされた簡単な制御動作です。PI 制御演算式を示します。

$$MV(t) = \frac{100}{PB} \left\{ ev(t) + \frac{1}{TI} \int ev(t) dt \right\}$$

060338J.ai

MV (t)	:	操作出力
ev (t)	:	実効偏差 (入力補償後 PV (t) - 設定値 SV (t))
PV (t)	:	測定値
SV (t)	:	設定値
PB	:	比例帯 (データアイテム P に保持)
TI	:	積分時間 (データアイテム I に保持)

上記の PI 制御演算式を実効スキャン周期ごとのサンプリング値を使用した式に変形し、差分演算式で表すと以下の式になります。

$$\Delta MV = \frac{100}{PB} * \left\{ \Delta ev + \frac{\Delta T}{TI} * ev \right\}$$

060339J.ai

deltaMv :	操作出力変更量
PB	: 比例帯 (データアイテム P に保持)
ev	: 実効偏差 (入力補償後 PV - 設定値 SV)
Δ ev	: 前回 ev と今回 ev の変化量
Δ T	: 実効スキャン周期
TI	: 積分時間 (データアイテム I に保持)

この演算式では操作出力変化量 (deltaMv) を計算します。

それではサンプルプログラムを動かしてみます。サンプルソリューション _SMPL_CONTROL の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは 1 章の準備作業により以下の作業フォルダにコピーされています。

<ドライブ名>¥UcaWork¥UcaSamples¥_SMPL_CONTROL

Visual Studio を起動し、_SMPL_CONTROL¥_SMPL_CONTROL.sln を開きます。[ビルド] メニューの [リビルド] で _SMPL_CONTROL の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。

次にサンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイル CONTROL.txt が CENTUM VP インストール先の以下にあります。CONTROL.txt を FCS0121 (APCS) の制御ドローイング DR0063 (空いていればどの制御ドローイングでも構いません) にインポートします。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥drawings¥CONTROL.txt

この制御ドローイングには、PI 制御を行う連続制御形ユーザカスタムブロック CMC_CONTROL を中心とするフィードバックループがあります。つまり、CMC_CONTROL の OUT 端子からの出力値は、1 次遅れブロック LAG_C00 を経由して CMC_CONTROL の IN 端子にフィードバックします。CMC_CONTROL の IN 端子から入力するデータには加算ブロック ADD_C00 を使い、手動操作ブロック MLD_C00 の MV 値を合計しています。この手動操作ブロックは、プラントに加わる外乱を模擬しています。また、1 次遅れブロック LAG_C00 はプラントの遅れ時間を模擬していて、1 次遅れ時間設定値 (データアイテム 1) には基本スキャン周期の 2 倍の 8 秒を指定しています。

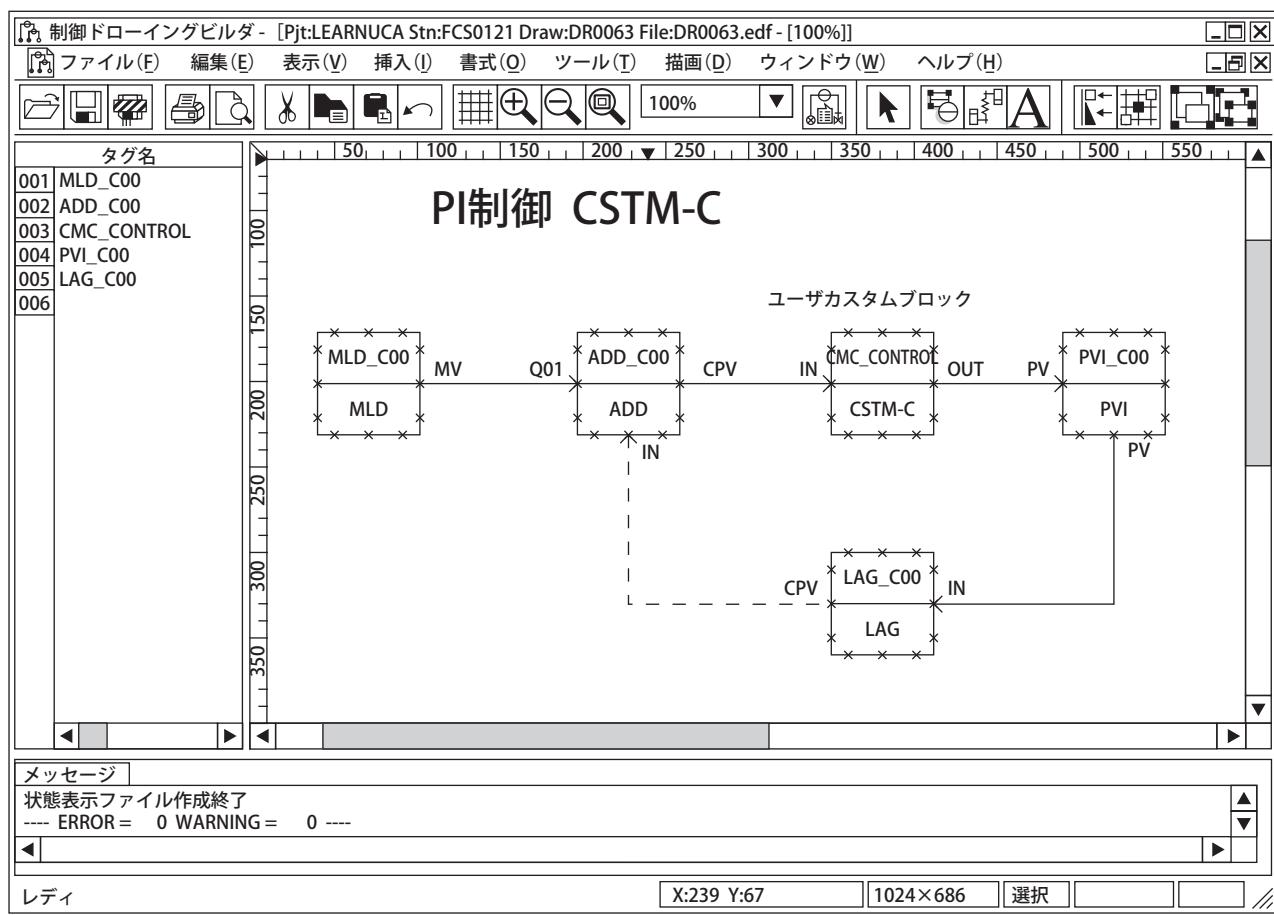


図 CMC_CONTROLのフィードバックループ

それではプログラムを動かしてみます。FCS0121 (APCS) を指定してバーチャルテストを起動します。起動が完了したら、手動操作ブロック MLD_C00 の MV に「5.0」を設定してください。



060304J.ai

図 MVの設定

連続制御形ユーザカスタムブロック CMC_CONTROL のチューニングウィンドウを表示してください。CMC_CONTROL のブロックモードを MAN にしてから MV に「45.0」を設定します。次に、SV に「50.0」を設定します。CMC_CONTROL はビルダ定義項目で「MAN 時にメジャートラッキング」するようにしてありますので、ブロックモードが MAN であれば PV 値が SV 値に設定（トラッキング）されます。しばらくすると、PV と SV が 50.0 になりますので、ブロックモードを AUT にしてください。

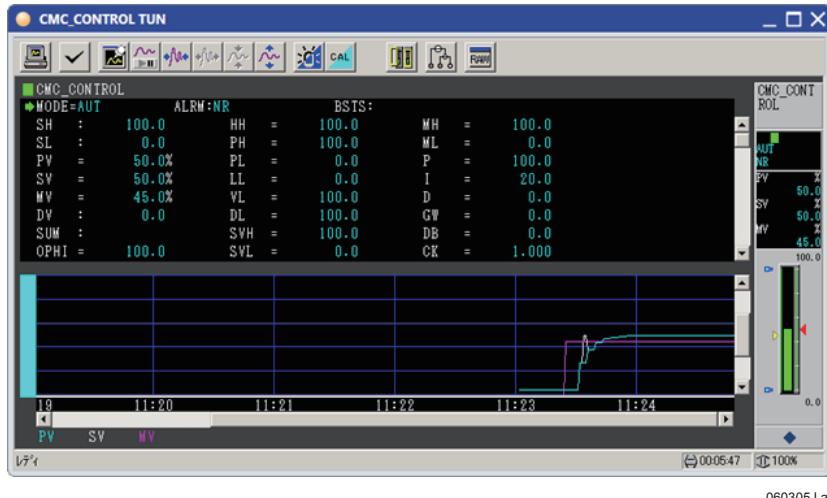


図 CMC_CONTROLのトラッキング

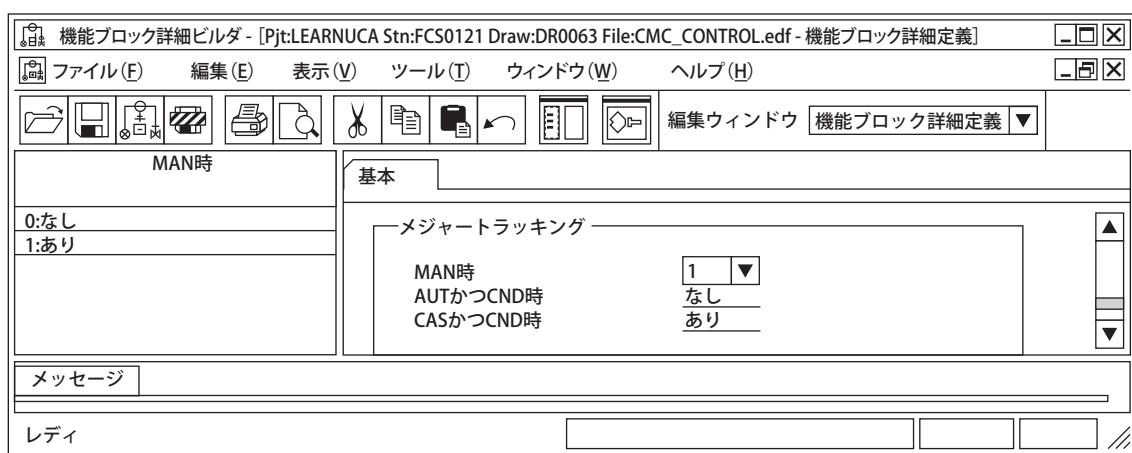


図 メジャートラッキングの指定

この状態(ブロックモードが AUT のまま)から、手入力で SV を「70.0」に変更してください。PI 制御により PV が SV に近づいていきます。以下は、PI 制御の結果 V と SV が一致した状態です。手動操作ブロック MLD_C00 の MV が「5.0」なので、CMC_CONTROL の操作出力値 MV は 65.0 ($5.0 + 65.0 = 70.0$) になります。

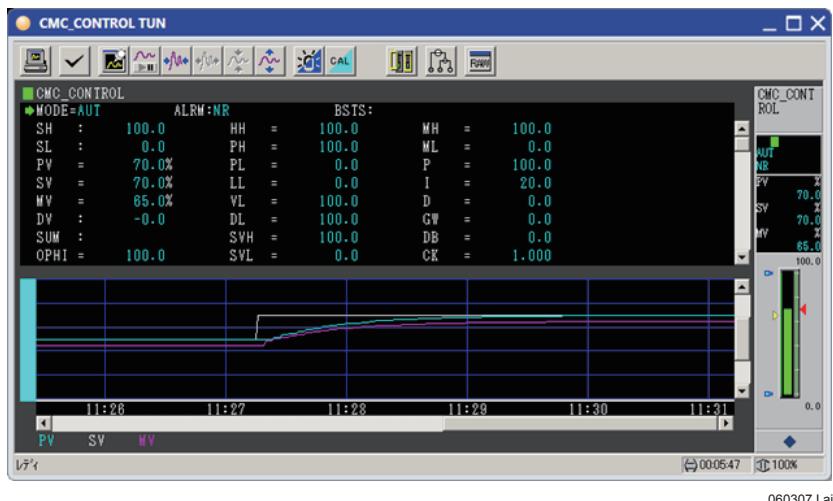


図 PI制御

■ PI制御演算を行うサンプルプログラム

ユーザカスタムアルゴリズムの _SMPL_CONTROL の control.c について説明します。このプログラムは、機能ブロック定周期処理 (UcaBlockPeriodical) のみ処理をします。機能ブロック定周期処理は、入力処理 (input)、制御演算処理 (control)、出力処理 (output) の 3 つのユーザ定義関数を指定して UcaCtrlHandler を呼び出します。

```
.....
/*********************************************
 * プロトタイプ宣言
 */
I32 input(UcaBlockContext bc);
I32 control(UcaBlockContext bc, F64 *mvbbPtr, F64 *mvblPtr);
I32 output(UcaBlockContext bc);

.....
/*
*<<FNH>>*****
*
* Function name:          UcaBlockPeriodical
* Return value:           SUCCEED      正常終了
*                         UCAERR_NOPROC   処理なし
*                         UCAERR_STOPME   処理続行不能
*
* description:            機能ブロック定周期処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode;           /* リターンコード */

    rtnCode = UcaCtrlHandler( bc,
        input,
        control,
        output,
        UCACTRL_AUTCASRCAS,
        NOOPTION);

    return SUCCEED;
}
```

● ユーザ定義の入力処理

ユーザ定義の入力処理（関数 input）は、UcaRWReadIn で IN 端子からデータアイテム RV にデータを読み込み、読み込んだデータを UcaRWSetPv でデータアイテム PV に設定します。

```
/*
* <<FNH>>*****
*
* Function name: input
* Return value: SUCCEED      正常終了
*
* description: 入力処理
*
*>>HNF<<*****
*/
I32 input(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode;          /* リターンコード */
    I32 rtnReadIn;        /* IN 端子入力リターンコード */

    /* IN 端子から RV にデータを読み込み */
    rtnReadIn = UcaRWReadIn(bc, NOOPTION);

    /* RV から PV にデータを設定 */
    rtnCode = UcaRWSetPv(bc, NULL, rtnReadIn, NOOPTION);

    return SUCCEED;
}
```

● ユーザ定義の出力処理

ユーザ定義の出力処理（output 関数）は、UcaRWWriteMvToOutSub で OUT 端子と SUB 端子からの出力処理を行います。

```
/*
* <<FNH>>*****
*
* Function name: output
* Return value: SUCCEED      正常終了
*
* description: 出力処理
*
*>>HNF<<*****
*/
I32 output(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode;          /* リターンコード */

    /* OUT 端子と SUB 端子からデータを出力 */
    rtnCode = UcaRWWriteMvToOutSub(bc, NOOPTION);

    return SUCCEED;
}
```

● ユーザ定義の制御演算処理

制御演算を行うユーザ定義関数 controlについて説明します。最初にソースコード全体を示します。

```

/*
* <<FNH>>*****
*
* Function name: control
* Return value: SUCCEED      正常終了
*
* description:             制御演算処理
*
*>>>HNF<<*****
*/
I32 control(
    UcaBlockContext bc, /* (IN/OUT) : ブロックコンテキスト */
    F64 *mvbbPtr,        /* (IN/OUT) : 出力補償前出力値ポインタ */
    F64 *mvblPtr         /* (IN/OUT) : 出力リミット前出力値ポインタ */
)
{
    F64S pv;           /* PV */
    F32S sv;           /* SV */
    F64 ev;            /* 実効偏差 */
    F64 cv;            /* CV */
    F64 Pin;           /* 比例項入力値 */
    F64 Iin;           /* 積分項入力値 */
    F32 pb;             /* P */
    F32 itime;          /* I */
    F64S p01;           /* P01 */
    F64 calc;           /* PID演算結果 */
    F64 deltaMv;        /* 出力変化値 */
    F64S outRb;          /* 出力読み返し値 */
    F64 mvbbVal;        /* 出力補償前出力値 */
    F64 mvblVal;        /* 出力リミット前出力値 */
    F64 lastMvblVal;    /* 前回出力リミット前出力値 */
    I16 configDir;       /* ビルダ定義項目動作方向 */
    I16 configOut;       /* ビルダ定義項目出力動作 */
    I16 configComp;       /* ビルダ定義項目入出力補償 */
    I16 configDeadband;  /* ビルダ定義項目不感帯動作 */
    BOOL isCtrlHold;     /* 制御ホールドかどうか */
    BOOL isCtrlInit;     /* 制御初期化要求かどうか */
    I32 time;            /* 制御周期時間 */
    I32 rtnCode;          /* リターンコード */
}

```

(続く)

(続き)

```

/* 制御ホールド確認 */
rtnCode = UcaCtrlHoldCheck(bc, &isCtrlHold);

/* 制御演算処理 */
/*****************************************/
/* 制御ホールド時には、制御演算を行いません。 */
/*****************************************/
if (!isCtrlHold) { /* 制御演算状態(制御ホールドでない場合) */

    /* データアイテムを取得 */
    rtnCode = UcaDataGetPv(bc, &pv, NOOPTION); /* PV */
    rtnCode = UcaDataGetSv(bc, &sv, NOOPTION); /* SV */
    rtnCode = UcaDataGetP(bc, &pb, NOOPTION); /* P */
    rtnCode = UcaDataGetI(bc, &itime, NOOPTION); /* I */
    rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION); /* P01(前回CV値) */
    rtnCode = UcaDataGetTsc(bc, &time, NOOPTION); /* TSC */

    /* 実効スキャン周期を取得 */
}

/* 入力補償 */
cv = pv.value;
rtnCode = UcaConfigCompensation(bc, &configComp);
if (configComp == UCACONFIG_COMP_INPUT) {
    rtnCode = UcaCtrlCompensation(bc, &cv, NOOPTION);
}

/* 制御初期化 */
rtnCode = UcaCtrlInitCheck(bc, &isCtrlInit);
if (isCtrlInit){
    rtnCode = UcaCtrlFuncInit(bc, NOOPTION); /* 制御初期化 */
    time = 0; /* 制御周期時間をクリア */
    p01.value = cv; /* 前回CV値を初期化 */
    p01.status = 0; /* データステータス正常 */
    rtnCode = UcaCtrlInitClear(bc); /* 初期化要求をクリア */
}

```

(続く)

(続き)

```

/* 入力項を設定 */
ev = cv - sv.value;                                /* 実効偏差 */
Pin = cv - p01.value;
Iin = ev;

/* PI制御演算 */
if (itime <= 0.01){
    calc = 0.0;                                     /* 積分時間異常 */
} else if (pb >= 0.01){
    calc = 100.0 / pb * (Pin + time * Iin / itime);
} else {
    calc = time * Iin / itime; /* 比例動作バイパス */
}

/* CV前回値をP01に保存(データステータスは正常) */
p01.value = cv;
p01.status = 0;
rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);

/* レンジ変換 */
rtnCode = UcaDataConvertRange(bc, calc, &deltaMv, UCAOPT_RANGE_DIFFVALUE);

/* 制御動作方向 */
rtnCode = UcaConfigDirection(bc, &configDir);
if (configDir == UCACONFIG_DIR_REV){                /* 逆動作 */
    deltaMv = -deltaMv;
}

/* 制御出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS){                /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimitOpnt(bc, &deltaMv, UCAOPT_CTRL_TSCTIME);

    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE){
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }
}

```

PI制御演算
(ユーザ定義の
制御演算)

レンジ変換

制御動作方向

制御出力動作を
取得
リセットリミッタ

不感帯動作

(続く)

(続き)

```

/* MVBB設定 */
rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
mvbbVal = deltaMv + lastMvblVal;

} else if (configOut == UCACONFIG_ACT_VEL) { /* 速度形 */
    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE) {
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = deltaMv + outRb.value;
}

/* MVBL設定 */
/* 出力補償(入力補償時にビルダ定義項目取得済み) */
mvblVal = mvbbVal;
if (configComp == UCACONFIG_COMP_OUTPUT){
    rtnCode = UcaCtrlCompensation(bc, &mvblVal, NOOPTION);
}

} else /* 非制御状態(制御ホールドの場合) */
/* 出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS){ /* 位置形 */
    rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
    mvbbVal = lastMvblVal;

} else if (configOut == UCACONFIG_ACT_VEL){ /* 速度形 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = outRb.value;

}

/* MVBL には MVBB 値を設定します。 */
mvblVal = mvbbVal;
}

/* MVBB、MVBL を保存 */
*mvbbPtr = mvbbVal;
*mvblPtr = mvblVal;

return SUCCEED;
}

```

この制御演算処理におけるデータの流れを以下に示します。

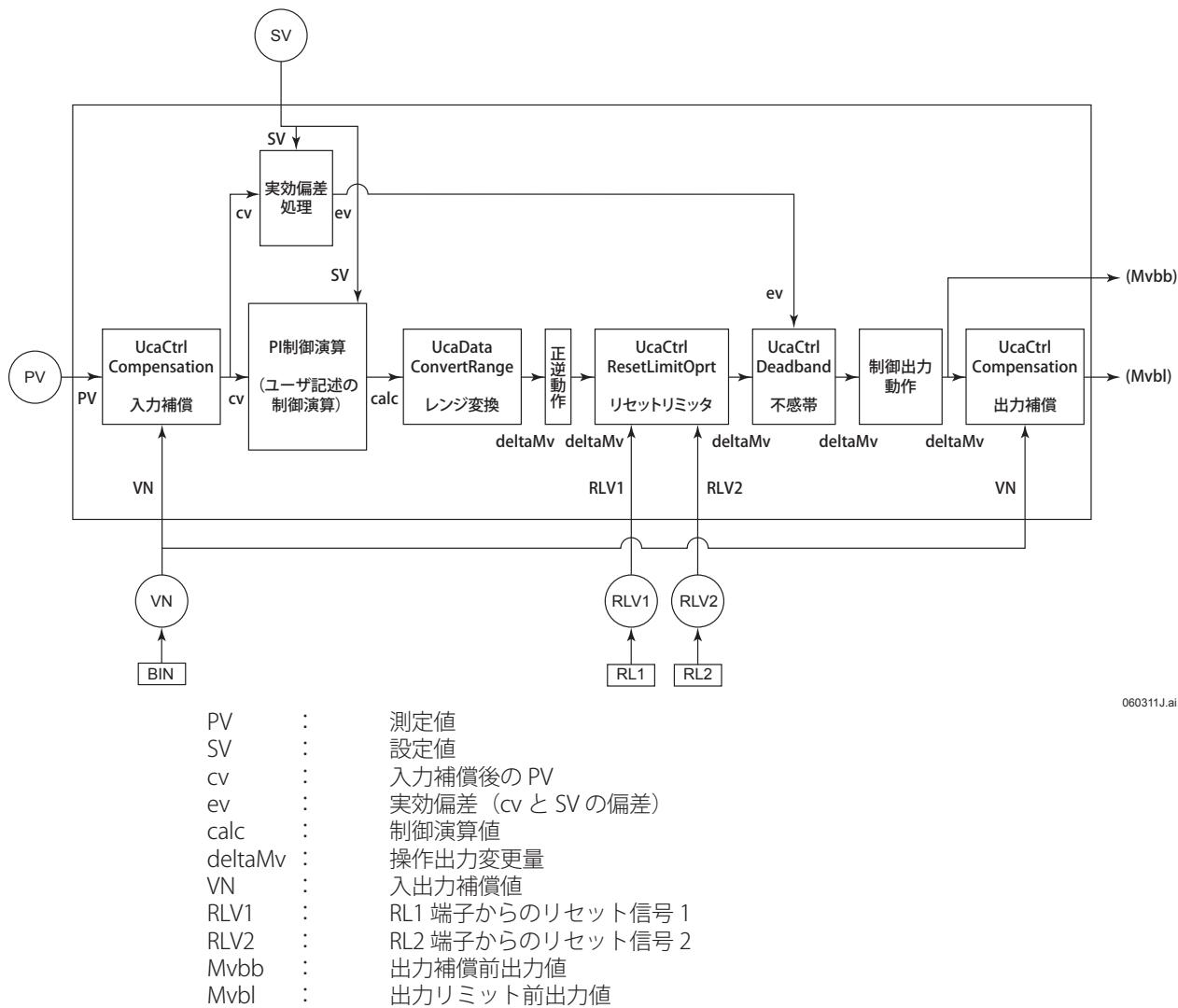


図 制御演算関数を使用した制御演算のデータの流れ

6.3.1 制御ホールド

制御ホールドは、ブロックモードを保持したまま一時的に制御動作を中断させる機能です。制御ホールド時には制御演算はしませんが、制御出力動作は通常どおりに行います。

IN 端子からの入力処理をする UcaRWReadIn は、次に示す条件が成立すると制御ホールドの要求を設定します。

- IN 端子の結合先がオープンの（切り替えスイッチなどで選択されていない）とき

制御演算処理の最初で UcaCtrlHoldCheck を呼び出し、UcaRWReadIn が制御ホールド要求を設定しているか否かを検査しています。

```
.....
/* 制御ホールド確認 */
rtnCode = UcaCtrlHoldCheck(bc, &isCtrlHold);

/* 制御演算処理 */
/*********************************************
 * 制御ホールド時には、制御演算を行いません。
 *****************************************/
/*********************************************
 * 制御ホールドでない場合には実行する、制御演算処理が記述されています
 */
if (!isCtrlHold) {           /* 制御演算状態（制御ホールドでない場合） */
    .... (制御ホールドでない場合に実行する、制御演算処理が記述されています)
} else { /* 非制御状態（制御ホールドの場合） */
   /*********************************************
     * 出力動作が位置形の場合、MVBB には前回 MVBL 値を設定します。
     */
    /* 速度形の場合、MVBB には出力読み返し値を設定します。
     */
    /*****
     */
/* 出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS) {           /* 位置形 */
    rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
    mvbbVal = lastMvblVal;
} else if (configOut == UCACONFIG_ACT_VEL) {      /* 速度形 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = outRb.value;
}
/*****
 */
/* MVBL には MVBB 値を設定します。
 */
/*****
 */
mvblVal = mvbbVal;
}

....
```

UcaCtrlHoldCheck で、制御ホールドが要求されているかを引数 isCtrlHold に取得します。制御ホールドが要求されていなければ、制御演算処理を実行します。if (!isCtrlHold) の「！」は C 言語の否定演算子です。「if (!isCtrlHold)」は "if not isCtrlHold"、つまり「制御ホールドが要求されていなければ」と読んでください。

制御ホールドが要求されている場合には（{} else {以下の部分}）、UcaConfigOutput でビルダ定義項目「制御出力動作」を取り出し、位置形または速度形の出力補償前出力値を作成しています。

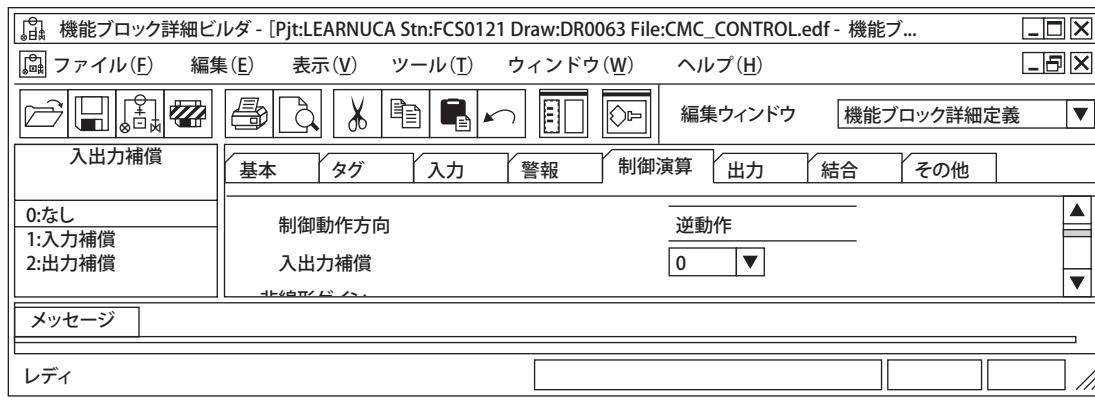
参照 制御出力動作と出力補償前出力値の作り方の詳細については、以下を参照してください。
[「6.3.6 制御出力動作」](#)

6.3.2 入力補償と出力補償

入力補償とは、制御演算の入力信号（測定値PV）に外部からBIN端子を経由して指定する入力補償値を加算する制御動作です。また、出力補償とは制御演算の出力信号（制御演算で計算する出力補償前出力値Mvbb）に外部からBIN端子を経由して指定する出力補償値を加算する制御動作です。

UcaCtrlHandlerは前処理でBIN端子の結合先からデータを入力し、入出力補償値を保持するデータアイテムVNに設定しています。つまり、ユーザ記述の制御演算処理をするときには、入出力補償値はデータアイテムVNに設定済みです。

入出力補償は、ビルダ定義項目で「なし」「入力補償」「出力補償」から指定します。



060312J.ai

図 入出力補償の指定

- デフォルトは「なし」です。
- 「入力補償」または「出力補償」のどちらかを指定できます。「入力補償」と「出力補償」の両方を指定することはできません。

参照 入力補償と出力補償の詳細については、以下を参照してください。

機能ブロッククリアレンス Vol.1 (IM 33J15A30-01JA) 「1.4 調節ブロックに共通の制御演算処理」の「■入出力補償」

機能ブロッククリアレンス Vol.1 (IM 33J15A30-01JA) 「1.4 調節ブロックに共通の制御演算処理」の「■入力補償」

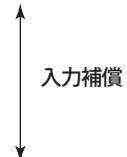
機能ブロッククリアレンス Vol.1 (IM 33J15A30-01JA) 「1.4 調節ブロックに共通の制御演算処理」の「■出力補償」

入力補償の処理をしているプログラムは、C 言語の変数にデータアイテムのデータを取得した後の次の部分です。

```
.....
/* データアイテムを取得 */
rtnCode = UcaDataGetPv(bc, &pv, NOOPTION);           /* PV */
rtnCode = UcaDataGetSv(bc, &sv, NOOPTION);           /* SV */
rtnCode = UcaDataGetP(bc, &pb, NOOPTION);           /* P */
rtnCode = UcaDataGetI(bc, &itime, NOOPTION);         /* I */
rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);    /* P01(前回CV値) */
rtnCode = UcaDataGetTsc(bc, &time, NOOPTION);        /* TSC */

/* 入力補償 */
cv = pv.value;
rtnCode = UcaConfigCompensation(bc, &configComp);
if (configComp == UCACONFIG_COMP_INPUT) {
    rtnCode = UcaCtrlCompensation(bc, &cv, NOOPTION);
}
.....

```



UcaConfigCompensation は、ビルダ定義項目「入出力補償」の指定値を取り出す関数です。ビルダ指定値を変数 configComp が UCACONFIG_COMP_INPUT (ビルダ定義項目「入出力補償」の指定が「入力補償」) であれば、UcaCtrlCompensation を呼び出し入力補償を実行します。UcaCtrlCompensation は、2 番目の引数 cv に測定値 PV (2 行上の cv = pv.value で設定済み) を入力し、入力補償した結果を同じ変数 cv に上書きします。

重要

UcaCtrlCompensation は、入力補償を実行した後、データアイテム VN に「0.0」を設定します。

入力補償の演算式を示します。

$$cv = PV + CK(VN + CB)$$

cv	:	入力補償後の PV
PV	:	測定値
CK	:	入出力補償ゲイン (データアイテム CK)
CB	:	入出力バイアス (内部バイアス、データアイテム CB)
VN	:	入出力補償値 (バイアス信号、データアイテム VN)

入力補償のデータの流れを以下に示します。

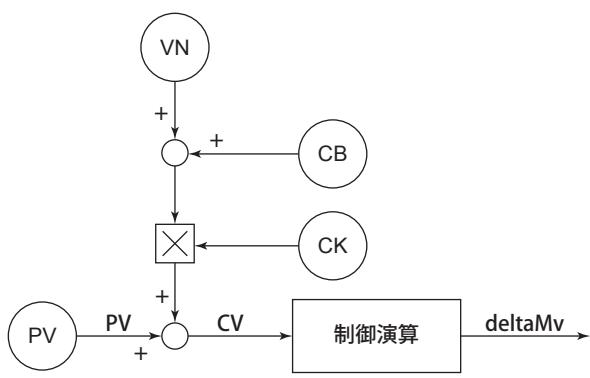


図 入力補償

次に出力補償について説明します。出力補償をするのは、制御演算が終わった後の次の部分です。

```
.....
/* 出力補償 */
mvblVal = mvbbVal;
rtnCode = UcaConfigCompensation(bc, &configComp);
if (configComp == UCACONFIG_COMP_OUTPUT) {
    rtnCode = UcaCtrlCompensation(bc, &mvblVal, NOOPTION);
}
.....
```

UcaConfigCompensation で変数 configComp に取り出したビルダ定義項目「入出力補償」の指定値を検査します。ビルダ指定値が UCACONFIG_COMP_OUTPUT(ビルダ定義項目「入出力補償」の指定が「出力補償」) であれば、UcaCtrlCompensation を呼び出し出力補償を実行します。UcaCtrlCompensation は、2 番目の引数 mvblVal に出力補償前出力値(2 行上の mvblVal = mvbbVal で設定済み) を入力し、出力補償した結果同じ変数 mvblVal に上書きします。

重要

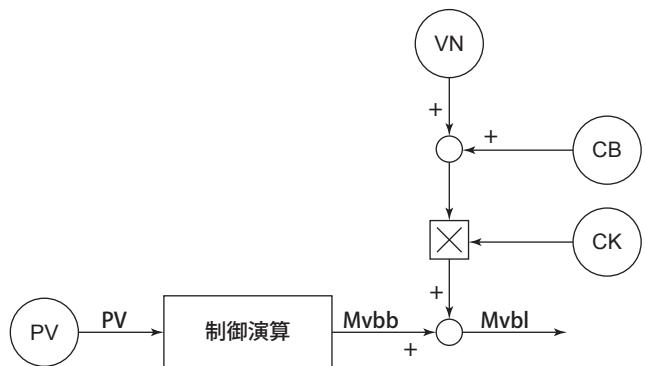
UcaCtrlCompensation は、出力補償を実行した後、データアイテム VN に「0.0」を設定します。

出力補償の演算式を示します。

$$Mvbl = Mvbb + CK(VN + CB)$$

Mvbb :	出力補償前出力値
Mvbl :	出力リミット前出力値(出力補償後出力値)
CK :	入出力補償ゲイン(データアイテム CK)
CB :	入出力バイアス(内部バイアス、データアイテム CB)
VN :	入出力補償値(バイアス信号、データアイテム VN)

出力補償のデータの流れを以下に示します。



060315J.ai

図 出力補償

6.3.3 制御初期化

自動運転をしないブロックモード（たとえばMAN）から自動運転を行うブロックモード（たとえばAUT）にブロックモードが遷移した場合、最初の制御演算の前に制御の初期化をする必要があります。

UcaCtrlHanlder（システム）は、制御の初期化が必要となる条件が成立すると「制御初期化の要求」を設定します。ユーザプログラムは、UcaCtrlInitCheck 関数で初期化要求が設定されているか検査し、初期化が必要であれば初期化を実行し、制御初期化の要求を UcaCtrlInitClear で解除します。この処理をしているのは、次の部分です。

```
.....
/* 制御初期化 */
rtnCode = UcaCtrlInitCheck(bc, &isCtrlInit);
if (isCtrlInit){
    rtnCode = UcaCtrlFuncInit(bc, NOOPTION);      /* 制御初期化 */
    time = 0;                                     /* 制御周期時間をクリア */
    p01.value = cv;                               /* 前回 CV 値を初期化 */
    p01.status = 0;                               /* データステータス正常 */
    rtnCode = UcaCtrlInitClear(bc);               /* 初期化要求をクリア */
}
.....
```

UcaCtrlInitCheck は、引数 isCtrlInit に制御初期化の要求が設定されているか否かを返します。制御初期化の要求が設定されていれば（isCtrlInit 変数が TRUE、つまり 0 以外であれば）、ユーザプログラムで制御初期化処理を実行します。制御初期化処理のポイントを示します。

- UcaCtrlInitCheck で制御初期化が必要か判定します。
- 制御初期化処理では、UcaCtrlFuncInit 関数でユーザカスタムアルゴリズム作成用ライブラリの制御演算関数の内部パラメータを初期化します。
- ユーザ記述の制御演算で使用する変数を初期化します。
- 制御初期化を実行したら、UcaCtrlInitClear を呼び出し制御初期化の要求を解除しておきます。

重要

- UcaCtrlInitCheck、UcaCtrlFuncInit、UcaCtrlInitClear は、必ず呼び出してください。
- ユーザ記述の制御演算で使用する変数の初期化（上例では、time、p01.value、p01.status の初期化）が必要か否かは、ユーザプログラムに依存します。

6.3.4 PI制御演算（ユーザ記述の制御演算）とレンジ変換

入力補償と制御初期化のあとには、制御演算を実行します。

このサンプルは PI 制御を行いますが、次の計算式により操作出力変化量（プログラムでは変数 deltaMv）を計算します。

$$\text{deltaMv} = \frac{100}{\text{PB}} * \left\{ \Delta \text{ev} + \frac{\Delta T}{\text{TI}} * \text{ev} \right\}$$

060340J.ai

上の式とプログラムの変数名の対応を以下に示します。

表 式とプログラムの変数名の対応

内容	式における記述	プログラムの変数名	補足
操作出力変更量	deltaMv	deltaMv	
比例帯	PB	pb	データアイテム P
実効偏差（入力補償後 PV - 設定値 SV）	ev	ev = cv - sv.value	cv は入力補償後の PV
前回 ev と今回 ev の変化量	Δev	Pin = cv - p01.value	cv 前回値をデータアイテム P01 に保持
実効スキャン周期	ΔT	time	データアイテム TSC
積分時間	TI	itime	データアイテム

この式の計算をするのは、プログラムの次の部分です。

```

.....
rtnCode = UcaDataGetTsc(bc, &time, NOOPTION);      /* TSC */

.....
/* 入力項を設定 */
ev = cv - sv.value;                                /* 実効偏差 */
Pin = cv - p01.value;
Iin = ev;

/* PI 制御演算 */
if (itime <= 0.01){
    calc = 0.0;                                     /* 積分時間異常 */
} else if (pb >= 0.01){
    calc = 100.0 / pb * (Pin + time * Iin / itime) ;
} else {
    calc = time * Iin / itime ;                   /* 比例動作バイパス */
}

/* CV 前回値を P01 に保存 (データステータスは正常) */
p01.value = cv;
p01.status = 0;
rtnCode = UcaDataStorePn(bc, &p01, 1, 1, NOOPTION);

/* レンジ変換 */
rtnCode = UcaDataConvertRange(bc, calc, &deltaMv, UCAOPT_RANGE_DIFFVALUE);
.....

```

このプログラムは、cv（入力補償後の PV）と設定値 SV を元に制御演算を行い、演算結果を変数 calc に作成します。また、変数 cv のデータをデータアイテム P01 に保持し、次の制御演算で cv 前回値として使用します。

制御演算の計算結果 calc を UcaDataConvertRange により測定値 PV のレンジから操作出力値 MV のレンジへレンジ変換することにより、操作出力変更量 deltaMv を決定します。

ユーザ記述の制御演算のポイントです。

- ・ 入力補償後の PV（変数 cv）と設定値 SV を使って制御演算を行い、演算結果を UcaDataConvertRange により測定値（PV）のレンジから操作出力値（MV）のレンジにレンジ変換します。
- ・ 制御演算に必要な前回値はデータアイテム P01～P32 に保持するようにしてください。サンプルでは、cv（入力補償後の PV）をデータアイテム P01 に保持しています。
- ・ 制御周期に「実効スキャン周期」を使用する場合には、実効スキャン周期を保持するデータアイテム TCS から周期時間を取り出すことができます。サンプルでは、データアイテム TCS から変数 time に周期時間を取得しています。

本サンプルの PI 制御は、エラー処理などを省略しています（たとえば、積分時間 itime が 0.01 以下なら計算結果 calc を 0.0 にするだけです）。本サンプルの目的は制御演算関数の使い方を説明することです。PI 制御演算の説明が目的ではありませんので、PI 制御に関する詳細な処理は省略しています。

補足 ユーザカスタムアルゴリズム作成用ライブラリには、PID 制御演算をする UcaCtrlPid 関数が用意されています。PI 制御演算を自作する場合には、（その前に）PID 制御演算の関数 UcaCtrlPid を使用することを検討してください。システムが用意している UcaCtrlPid 関数を使う方がプログラムの作成が簡単です。

参照 UcaCtrlPid の使い方の詳細については、以下を参照してください。

「[6.4 PID 調節ブロックと同じ動作をする CSTM-C](#)」

6.3.5 制御動作方向

制御動作方向は、操作出力変更量（プログラムでは変数deltaMv）の増減方向を示す正動作と逆動作を切り換える機能です。制御動作方向はビルダ定義項目として「逆動作」または「正動作」のどちらか一方を指定します。デフォルトは「逆動作」です。

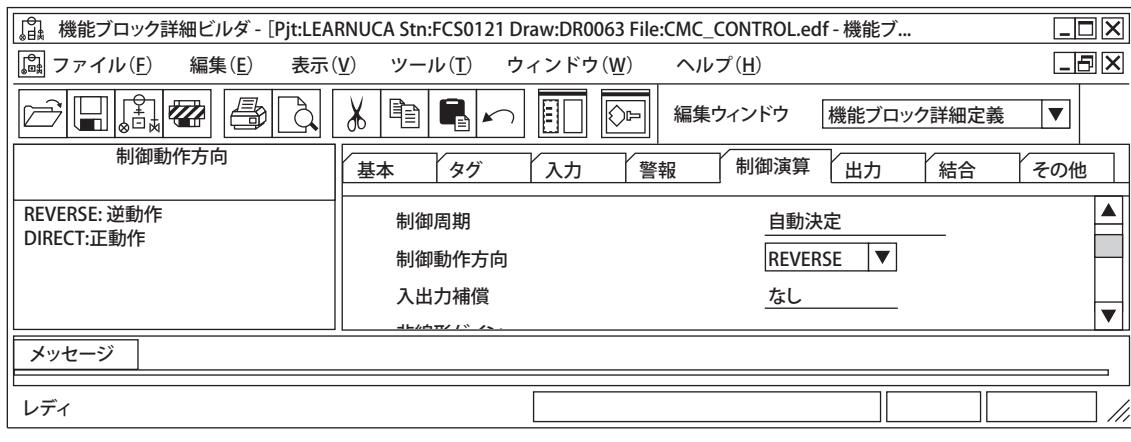


図 制御動作方向の指定

正動作と逆動作の説明を以下に示します。

参照 制御動作方向の詳細については、以下を参照してください。

[機能ブロッククリアレンス Vol.1 \(IM 33J15A30-01JA\)](#) 「1.4 調節ブロックに共通の制御演算処理」の「■ 制御動作方向」

表 正動作と逆動作

制御動作方向	説明
正動作	測定値 PV が増加すると操作出力値 MV も増加し、測定値 PV が減少すると操作出力値 MV も減少する制御動作です。
逆動作	測定値 PV が増加すると操作出力値 MV が減少し、測定値 PV が減少すると操作出力値 MV が増加する制御動作です。

制御動作方向の処理をするプログラムは以下の部分です。

```
.....
/* 制御動作方向 */
rtnCode = UcaConfigDirection(bc, &configDir);
if (configDir == UCACONFIG_DIR_REV) {           /* 逆動作 */
    deltaMv = -deltaMv;
}
.....
```

UcaConfigDirection で、ビルダ定義項目「制御動作方向」の指定を変数 configDir に取得します。指定が逆動作 (UCACONFIG_DIR_REV) であれば、「deltaMv = -deltaMv」という処理で操作出力変更量の正負を逆転し、測定値 PV の変化と操作出力変更量の正負を逆にしています。

6.3.6 制御出力動作

制御出力動作は、周期ごとの操作出力変更量（プログラムの変数deltaMv）を実際の操作出力量（プログラムの出力補償前出力値mvbbVal）に変換する処理です。制御出力動作はビルダ定義項目として「速度形」または「位置形」のどちらか一方を指定します。デフォルトは「位置形」です。

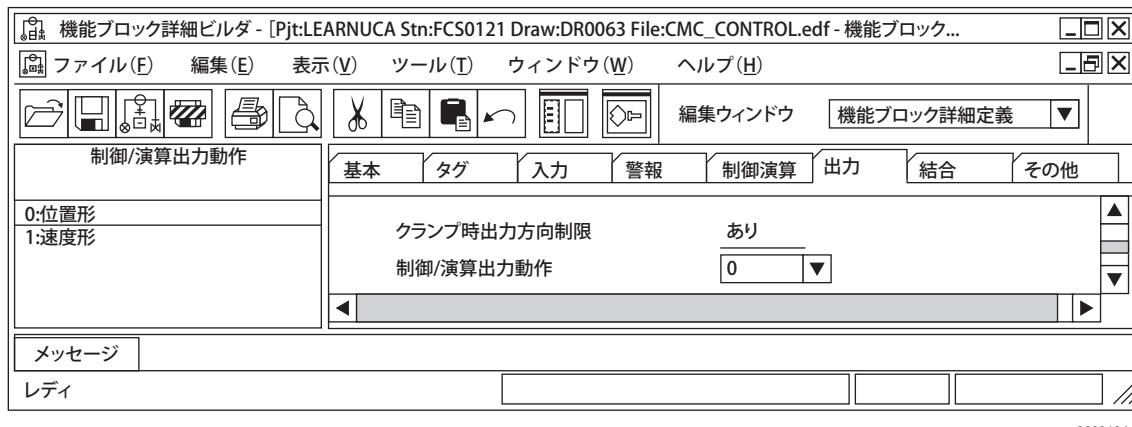


図 制御出力動作の指定

速度形と位置形の説明を以下に示します

表 速度形と位置形

制御出力動作	説明
速度形	出力先から読み返した値（内部変数 OutRb）に、今回の操作出力変更量（変数 deltaMv）を加算して、出力補償前出力値（変数 mvbbVal）を決定します。
位置形	前回の出力リミット前出力値（内部変数 Mvbl）に、今回の操作出力変更量（変数 deltaMv）を加算して、出力補償前出力値（変数 mvbbVal）を決定します。

重要

APCS の CSTM-C を上流として、FCS の機能ブロックとカスケード結合（APCS の CSTM-C の OUT 端子からの出力を FCS の機能ブロックの SET 端子に接続）する場合は、以下のようにしてください。

- CSTM-C の制御出力動作は位置形にしてください。
- CSTM-C の制御周期は、制御ステーション間結合（ステーション間結合ブロック）の通信周期より長くしてください。

参照

制御出力動作の詳細とステーション間結合ブロックの詳細については、以下を参照してください。

機能ブロッククリファレンス Vol.2 (IM 33J15A31-01JA) 「1.46 ステーション間結合ブロック (ADL)」

機能ブロッククリファレンス Vol.1 (IM 33J15A30-01JA) 「1.4 調節ブロックに共通の制御演算処理」の「■ 制御出力動作」

制御出力動作を行う部分のプログラムについて説明します。処理の最初に、UcaConfigOutputによりビルダ定義項目「制御出力動作」の指定を取り出し、位置形と速度形の処理を分けています。

```
.....
/* 制御出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS){           /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimitOprt(bc, &deltaMv, UCAOPT_CTRL_TSCTIME);

    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE){
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
    mvbbVal = deltaMv + lastMvblVal;
} else if (configOut == UCACONFIG_ACT_VEL){      /* 速度形 */
    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE){
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = deltaMv + outRb.value;
}

/* MVBL設定 */
/* 出力補償 */
mvblVal = mvbbVal;
rtnCode = UcaConfigCompensation(bc, &configComp);
if (configComp == UCACONFIG_COMP_OUTPUT){
    rtnCode = UcaCtrlCompensation(bc, &mvblVal, NOOPTION);
}
.....
```

位置形の場合には、操作出力変更量 deltaMv にリセットリミット処理と不感帯動作処理を実行したあとに、出力補償前出力値を決定します。つまり、前回の出力リミット前出力値（内部変数 Mvbl）を UcaDataGetMvbl で変数 lastMvblVal に取得し、これと deltaMv を加えた値を、出力補償前出力値 mvbbVal に代入します。

速度形の場合には、操作出力変更量 deltaMv に不感帯動作処理を実行したあとに、出力補償前出力値を決定します。つまり、出力値読み返し値（内部変数 OutRb）を UcaDataReadback で変数 outRb に取得し、これと deltaMv を加えた値を出力補償前出力値 mvbbVal に代入します。なお、UcaCtrlHandler の前処理で OUT 端子から読み返したデータを内部変数 OutRb に格納しています。

出力補償前出力値 (mvbbVal) が決定すると、出力補償を行い出力リミット前出力値 (mvblVal) を作成します。

参照 入出力補償の詳細については、以下を参照してください。
「[6.3.2 入力補償と出力補償](#)」

重要 位置形で、かつリセットリミットを使用しない場合（積分項のある制御演算をしない場合）、前回の出力リミット前出力値（UcaDataGetMvbl で取得）ではなく、前回 MV 値を UcaDataGetMv で取得し以下のようにしてください。リセットリミットをしない場合には、出力リミットされている MV 値を使用するのが適切です。

- 「制御演算状態（制御ホールドでない場合）」の処理は、UcaDataGetMvbl から UcaDataGetMv に変更します。

```
F32S lastMv;  
F64 lastMvblVal;  
.....  
rtnCode = UcaDataGetMv(bc, &lastMv, NOOPTION);           ← UcaDataGetMvにします  
mvbbVal = deltaMv + lastMv.value;  
.....  
.....  
rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION); ← UcaDataGetMvblの  
mvbbVal = deltaMv + lastMvblVal;  
.....
```

6.3.7 リセットリミット

リセットリミット機能は、制御演算の積分項に制限値を設けることでリセットワインドアップ（積分飽和）の発生を防ぐ機能です。リセットリミット機能は、制御出力動作が位置形の場合にのみ適用します。リセットリミット機能は、入力端子RL1とRL2の結合先から読み込んだ値を使って操作出力変更量の補正演算を行います。

UcaCtrlHanlder の前処理で、入力端子 RV1 の結合先データをデータアイテム RLV1 に、入力端子 RV2 の結合先データをデータアイテム RLV2 に、それぞれ読み込んでいます。したがって、ユーザ記述の制御演算処理をするときには、データアイテム RLV1 と RLV2 にリセット信号値が読み込まれています。

参照 リセットリミットの動作の詳細については、以下を参照してください。

[機能ブロッククリアレンス Vol.1 \(IM 33J15A30-01JA\) 「1.4 調節ブロックに共通の制御演算処理」の「■ リセットリミット機能」](#)

リセットリミット処理は、制御出力動作が位置形の場合のみ適用します。プログラムの以下の部分で、リセットリミット処理をする UcaCtrlResetLimitOprt を呼び出し、操作出力変更量 deltaMv を補正しています。

```
.....
/* 制御出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if ( configOut == UCACONFIG_ACT_POS) {           /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimitOprt( bc, &deltaMv, UCAOPT_CTRL_TSCTIME); ↑ リセット
                                                    ↓ リミット

    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if ( configDeadband == UCACONFIG_DEADBAND_TRUE) {
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }
}
.....
```

補足 このサンプルは制御の周期として実効スキャン周期を使用しているので、UcaCtrlResetLimitOprt のオプションに UCAOPT_CTRL_TSCTIME を指定し「制御時間に実効スキャン周期を使用すること」を指示しています。

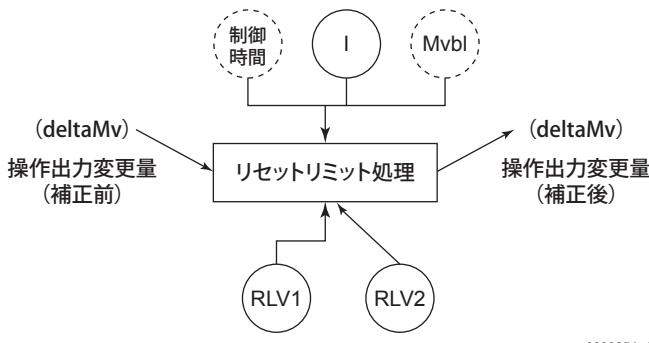


図 リセットリミット処理

060325J.ai

6.3.8 不感帯動作

不感帯動作は、測定値PVと設定値SVの偏差（データアイテムDV = PV - SV）があらかじめ設定しておいた不感帯幅（データアイテムDB）の範囲内にある場合には、操作出力変更量（変数deltaMv）を0にする機能です。不感帯動作はビルダ定義項目として「あり」または「なし」のどちらか一方を指定します。デフォルトは「なし」です。

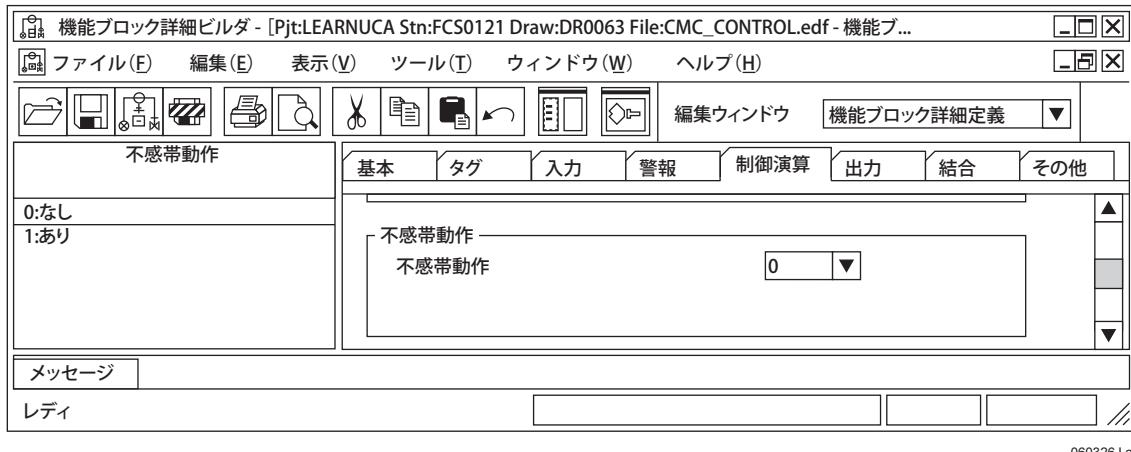


図 不感帯動作の指定

また、不感帯動作に「あり」を指定する場合には、「ヒステリシス」を指定します。デフォルトは、PVスケールスパンの1.0%相当量です。



図 ヒステリシスの指定

不感帯動作は、位置形と速度形のそれぞれで UcaConfigDeadband と UcaCtrlDeadband を呼び出して実現します。位置形ではリセットリミット処理の後に不感帯処理をします。プログラムの以下の部分です。

```
/* 制御出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS){           /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimitOprt(bc, &deltaMv, UCAOPT_CTRL_TSCTIME);

    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE) {
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
    mvbbVal = deltaMv + lastMvblVal;

} else if (configOut == UCACONFIG_ACT_VEL){      /* 速度形 */
    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE) {
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = deltaMv + outRb.value;
}

....
```



UcaConfigDeadband で、ビルダ定義項目「不感帯動作」の指定を取得します。指定が「あり」（変数 configDeadband に取得した値が UCACONFIG_DEADBAND_TRUE）であれば、UcaCtrlDeadband を呼び出し不感帯動作を実行します。UcaCtrlDeadband は内部でビルダ定義項目「ヒステリシス」の指定値を取得し、不感帯動作を実行します。

参照 不感帯動作の動作の詳細については、以下を参照してください。

[機能ブロッククリアレンス Vol.1 \(IM 33J15A30-01JA\)](#) 「1.4 調節ブロックに共通の制御演算処理」の「■ 不感帯動作」

6.3.9 ユーザによる制御演算関数のパラメータ指定改造

制御演算関数は、決められたデータアイテムからデータを入力します。ここでは、制御演算関数のパラメータを決められたデータアイテム以外にすることによりユーザ独自の動作を実現する方法を、出力補償を例に説明します。

■ 制御演算関数を呼び出す前に、データアイテムの値を変更

出力補償におけるデータの流れと演算式を示します。

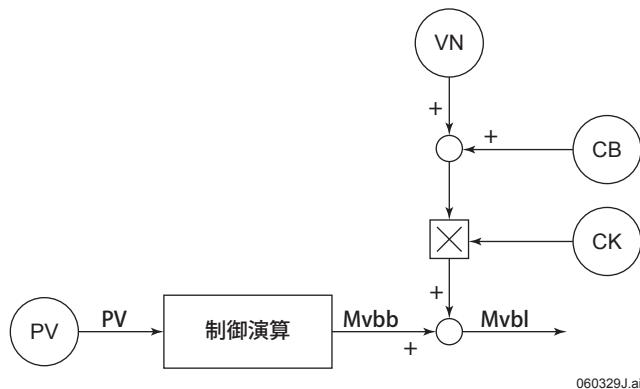


図 出力補償のデータの流れ

出力補償の演算式を示します。

$$Mvbl = Mvbb + CK(VN + CB)$$

Mvbb	:	出力補償前出力値
Mvbl	:	出力リミット前出力値（出力補償後出力値）
CK	:	入出力補償ゲイン（データアイテム CK）
CB	:	入出力バイアス（内部バイアス）
VN	:	入出力補償値（バイアス信号）

サンプルプログラムで出力補償をするのは、UcaCtrlCompensation を呼び出している次の部分です。

```
.....
/* 出力補償 */
mvblVal = mvbbVal;
rtnCode = UcaConfigCompensation(bc, &configComp);
if (configComp == UCACONFIG_COMP_OUTPUT) {
    rtnCode = UcaCtrlCompensation(bc, &mvblVal, NOOPTION);
}
.....
```

UcaCtrlCompensation は、2 番目の引数 mvblVal に出力補償前出力値（2 行上の mvblVal = mvbbVal で設定済み）を入力し、出力補償した結果同じ変数 mvblVal に上書きします。それ以外の入力パラメータは 3 つのデータアイテム、CK（入出力補償ゲイン）、CB（入出力バイアス）、VN（入出力補償値）から入力しています。

したがって、UcaCtrlCompensation を呼び出す前にデータアイテム CK、CB、または VN のデータを変更しておけば、出力補償処理をする関数 UcaCtrlCompensation の入力パラメータを変更することができます。

補足

- UcaCtrlHandler の前処理は、BIN 端子の結合先から入出力補償値を読み込みデータアイテム VN に設定します。
- UcaCtrlCompensation は、出力補償を実行したあと、データアイテム VN に 0.0 を設定します。

制御演算関数がデータを入力するデータアイテムを確認し、関数を呼び出す前にデータアイテムの値を変更すれば、制御演算関数の入力するパラメータを変更することができます。

■ パラメータをすべて引数で指定する制御演算関数

これまで説明してきた制御演算関数は、それぞれの機能に対応したデータアイテムからデータを入力しています。これに対し、すべてのパラメータを関数の引数に指定する制御演算関数が用意されています。これらの関数は、関数名の末尾に「_p」が付いています。全パラメータを引数に指定する制御演算関数を以下に示します。

表 全パラメータを引数に指定する制御演算関数

機能	関数名	説明
リセットリミット	UcaCtrlResetLimitOpt_p	リセットリミット機能を実行します。
不感帯動作	UcaCtrlDeadband_p	不感帯動作を実行します。
入出力補償	UcaCtrlCompensation_p	入力補償または出力補償を実行します。

出力補償を例に、全パラメータを引数で指定する制御演算関数の使い方を説明します。UcaCtrlCompensation を UcaCtrlCompensation_p に置き換えたプログラムを示します。

```
.....
F32S vn;          /* 入出力補償値 (VN) */
F32 ck;          /* 入出力補償ゲイン (CK) */
F32 cb;          /* 入出力補償バイアス (CB) */

.....
/* 出力補償 */
mvblVal = mvbbVal;
rtnCode = UcaConfigCompensation(bc, &configComp);
if (configComp == UCACONFIG_COMP_OUTPUT) {
    rtnCode = UcaDataGetVn(bc, &vn, NOOPTION);           /* VN */
    rtnCode = UcaDataGetCk(bc, &ck, NOOPTION);           /* CK */
    rtnCode = UcaDataGetCb(bc, &cb, NOOPTION);           /* CB */
    rtnCode = UcaCtrlCompensation_p(bc, &vn, ck, cb,
                                    &mvblVal, NOOPTION);
    rtnCode = UcaDataStoreVn(bc, &vn, NOOPTION);         /* VN (=0.0) を保存 */
}
.....
```

UcaCtrlCompensation_p は、UcaCtrlCompensation がデータアイテムから入力していた入出力補償値 (VN)、入出力補償ゲイン (CK)、入出力補償バイアス (CB) を引数から入力します。また、出力補償を実行したあとに、引数 vn が差す領域 (入出力補償値) に 0.0 を設定します。上のプログラムは、(もとの) サンプルの UcaCtrlCompensation を使用したプログラムと完全に等価です。

上の例では、データアイテム VN、CK、CB のデータを取得して、引数のデータを作り出しています。この部分 (UcaDataGet <データアイテム名> でデータを取得する部分) を置きかえることにより、ユーザが独自に作成したパラメータによる出力補償を実現することができます。

ユーザプログラムで、制御演算関数のパラメータを変更する方法について説明してきました。

- 制御演算関数は決められたデータアイテムからデータを入力します。何らかの小さい改造をする場合には、制御演算関数を呼び出す前にデータアイテムの値を変更するようにしてください。
- 制御演算関数には、全パラメータを引数から取得する「_p 付き」の関数が用意されています。制御演算に渡すパラメータをユーザ独自の計算式で計算する場合には、「_p 付き」の関数を使い引数に明示的にパラメータを渡すようにしてください。
- 決められたデータアイテム（出力補償ではデータアイテム VN、CK、CB）を使わないので機能を実現したい場合には、全パラメータを引数で渡す「_p 付き」の制御演算関数を使用します。

6.3.10 制御演算関数呼び出しの省略

制御演算関数の機能が不要なユーザプログラムでは、不要な機能の処理をする制御演算関数の呼び出しを省略することができます。ただし、制御演算初期化の関数群は呼び出しを省略できません。省略の可否を以下に示します。

重要

入出力補償、制御動作方向、制御出力動作はプログラムの記述を省略可能ですが、プログラムを省略した場合でも対応するビルダ定義項目の指定が必要です。詳細は個々の項目で説明します。

表 制御演算関数呼び出しの省略可否

機能	関数名	プログラムの省略	補足
制御ホールド	UcaCtrlHoldCheck	可能	
入出力補償	UcaConfigCompensation UcaCtrlCompensation UcaCtrlCompensation_p	可能	プログラムを省略した場合でも、プログラムの記述に合わせたビルダ定義項目の指定が必要
レンジ変換	UcaDataConvertRange	→	UcaDataConvertRange の呼び出しは、測定値 PV と操作出力値 MV のレンジが異なる場合には、省略できません。レンジが同じなら省略できます。
リセットリミット	UcaCtrlResetLimitOprt UcaCtrlResetLimitOprt_p	可能	
不感帯動作	UcaConfigDeadband UcaCtrlDeadband UcaCtrlDeadband_p	可能	
制御演算初期化	UcaCtrlInitCheck	不可	
	UcaCtrlInitClear	不可	
	UcaCtrlFuncInit	不可	
制御動作方向	UcaConfigDirection	可能	プログラムを省略した場合でも、プログラムの記述に合わせたビルダ定義項目の指定が必要
制御出力動作	UcaConfigOutput	可能	プログラムを省略した場合でも、プログラムの記述に合わせたビルダ定義項目の指定が必要

使用しない機能の制御演算関数の呼び出しを省略する必要はありません。機能を使用しない場合でも、該当機能の制御演算関数の呼び出しを省略しないで残したままで実害はありません。

■ 制御ホールド

制御ホールド機能を使用しない場合には、UcaCtrlHoldCheck の呼び出しと制御ホールド時の処理を省略できます。省略可能な行を、先頭の●印で示します。

```

●      /* 制御ホールド確認 */
●      rtnCode = UcaCtrlHoldCheck(bc, &isCtrlHold);

●      /* 制御演算処理 */
●      /***** */
●      /* 制御ホールド時には、制御演算を行いません。 */          */
●      /***** */
●      if (!isCtrlHold) {           /* 制御演算状態 (制御ホールドでない場合) */

...
... (制御演算の処理を記述。ここは、省略しない。)
...

●      } else { /* 非制御状態 (制御ホールドの場合) */
●      /***** */
●      /* 出力動作が位置形の場合、MVBB には前回 MVBL 値を設定します。 */      */
●      /* 速度形の場合、MVBB には出力読み返し値を設定します。 */      */
●      /***** */

●      /* 出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS){           /* 位置形 */
    rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
    mvbbVal = lastMvblVal;

} else if (configOut == UCACONFIG_ACT_VEL){      /* 速度形 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = outRb.value;

}
●      /***** */
●      /* MVBL には MVBB 値を設定します。 */          */
●      /***** */
mvblVal = mvbbVal;
●  }

/* MVBB、MVBL を保存 */
*mvbbPtr = mvbbVal;
*mvblPtr = mvblVal;

return SUCCEED;
}

```

■ 入出力補償

入力補償と出力補償が不要な場合は、UcaConfigCompensation と UcaCtrlCompensation の呼び出しを省略できます。入力補償か出力補償のどちらか一方の処理を省略することができます（この場合でも、ビルダ定義項目の指定は必要です）。また、両方を省略することもできます。省略可能な行を、先頭の●印で示します。

- ・ 入力補償

```
●          /* 入力補償 */
cv = pv.value;
●          rtnCode = UcaConfigCompensation(bc, &configComp);
●          if (configComp == UCACONFIG_COMP_INPUT) {
●              rtnCode = UcaCtrlCompensation(bc, &cv, NOOPTION);
●          }
```

- ・ 出力補償

```
/* MVBL設定 */
●          /* 出力補償 */
mvblVal = mvbbVal;
●          rtnCode = UcaConfigCompensation(bc, &configComp);
●          if (configComp == UCACONFIG_COMP_OUTPUT) {
●              rtnCode = UcaCtrlCompensation(bc, &mvblVal, NOOPTION);
●          }
```

重要

プログラムでは、入力補償または出力補償のどちらかのみを記述する場合でも、CSTM-C のビルダ定義項目「入出力補償」はプログラムに合わせた指定が必要です。たとえば、プログラムに出力補償の処理のみを記述する場合には、ビルダ定義項目「入出力補償」に「出力補償」を指定してください（ビルダ定義項目「入出力補償」のデフォルトは「なし」です）。指定をしないとデータアイテム VN に関する MAN フォールバック処理が正常動作しません。ビルダ定義項目が「なし」の場合、入出力補償値を保持するデータアイテム VN のデータステータスが BAD でも MAN フォールバックしなくなります。ブロックモード遷移処理では、データアイテム VN のデータステータスが BAD で、かつ「入出力補償」の指定が「なし」以外の場合に MAN フォールバックを行います。

参照 ブロックモード遷移処理の詳細については、以下を参照してください。

『6.2.4 ブロックモード遷移と設定値の作成 (UcaCtrlHandler の内部で処理)』



図 入出力補償の指定

060332J.ai

■ レンジ変換

レンジ変換をする関数 UcaDataConvertRange は、指定されたデータを測定値 PV のレンジから操作出力値 MV のレンジに変換します。以下の行では、引数 calc のデータをレンジ変換し、結果を deltaMv に設定しています。測定値 PV と操作出力値 MV のレンジが異なる場合には、UcaDataConvertRange の呼び出しは省略できません。

測定値 PV と操作出力値 MV のレンジが同じで、レンジ変換関数 UcaDataConvertRange の呼び出しを省略する場合には、次のように置き換えます。置き換えの対象となる行を、先頭の●印で示します。

```
● /* レンジ変換 */
● rtnCode = UcaDataConvertRange(bc, calc, &deltaMv, UCAOPT_RANGE_DIFFVALUE);
    ↓
deltaMv = calc;
```

■ 制御動作方向

サンプルプログラムは、ビルダ定義項目により制御動作方向を「正動作」「逆動作」のどちらかに指定できるようにしています。どちらかの動作に固定で構わない場合、ビルダ指定値を取り出す UcaConfigDirection の呼び出しを省略できます（省略した場合でもビルダ定義項目の指定は必要です）。次のように置き換えます。置き換えの対象となる行を、先頭の●印で示します。

```
● /* 制御動作方向 */
● rtnCode = UcaConfigDirection(bc, &configDir);
● if (configDir == UCACONFIG_DIR_REV) {           /* 逆動作 */
●     deltaMv = -deltaMv;
● }
```

正動作固定 処理不要 (deltaMv = deltaMvの意味です)。
逆動作固定 deltaMv = -deltaMv;

重要

プログラムでは、正動作または逆動作のどちらかのみを記述する場合でも、CSTM-C のビルダ定義項目「制御動作方向」はプログラムに合わせた指定が必要です。たとえば、プログラムを逆動作固定で記述する場合には、ビルダ定義項目「制御動作方向」に「逆動作」を指定してください（ビルダ定義項目「制御動作方向」のデフォルトは「逆動作」です）。システム内部にビルダ定義項目「制御動作方向」の指定に従い動作する部分があるので、プログラムの記述とビルダ定義項目が合っていないと、制御動作方向が正しく動作しません。システムは、次の処理でビルダ定義項目「制御動作方向」の指定値を参照しています。

- 出力クランプステータスのトラッキング

操作出力値 MV のデータステータスである CLP+ または CPL- を、設定値 (SV、CSV、RSV) にコピーする処理です。制御動作方向が「正動作」であれば、MV の CLP+ は設定値(CV、SCV、RSV)の CLP-、MV の CLP- は設定値の CLP+ としてコピーされます。「逆動作」の場合は、MV の CLP+ は設定値の CLP+、MV の CPL- は設定値の CLP- としてコピーされます。

- PRD (プライマリダイレクト) 時の操作出力値

プライマリダイレクト (PRD) モード動作は、カスケード上流ブロックからの設定値 (CSV) を操作出力値 (MV) に変換して出力する機能です。カスケード設定値 (CSV) から操作出力値 (MV) への変換の方法は、制御動作方向の「正動作」または「逆動作」の指定に応じて異なります。

参照

出力クランプステータスのトラッキングおよび PRD (プライマリダイレクト) 時の操作出力値の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「4.3 出力クランプ」](#)

[機能ブロッククリアレンス Vol.1 \(IM 33J15A30-01JA\) 「1.4 調節ブロックに共通の制御演算処理」の「■ プライマリダイレクト \(PRD\) モード動作」](#)

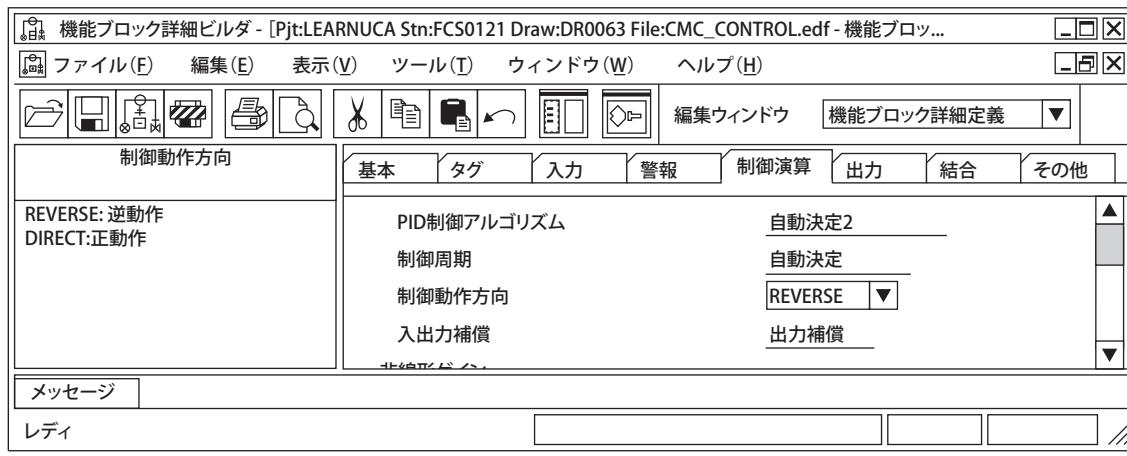


図 制御動作方向の指定

■ 制御出力動作

サンプルプログラムは、ビルダ定義項目により制御出力動作を「位置形」「速度形」のどちらかに指定できるようにしています。どちらかの動作に固定で構わない場合、ビルダ指定値を取り出す UcaConfigOutput の呼び出しを省略できます（省略した場合でもビルダ定義項目の指定は必要です）。この場合は先頭に●印をしてある行を省略し、「位置形」または「速度形」のどちらかの処理のみを記述します。

重要 APCS の CSTM-C を上流として、FCS の機能ブロックとカスケード結合（APCS の CSTM-C の OUT 端子からの出力を FCS の機能ブロックの SET 端子にステーション間結合ブロックを経由して接続）する場合は、CSTM-C の制御出力動作は位置形にしてください。

参照 ステーション間結合ブロックの詳細については、以下を参照してください。

[機能ブロッククリアレンス Vol.2 \(IM 33J15A31-01JA\) 「1.46 ステーション間結合ブロック \(ADL\)」](#)

```

● /* 制御出力動作 */
● rtnCode = UcaConfigOutput(bc, &configOut);
● if (configOut == UCACONFIG_ACT_POS){           /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimitOprt(bc, &deltaMv, UCAOPT_CTRL_TSCTIME);

    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE){
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
    mvbbVal = deltaMv + lastMvblVal;
}

● } else if (configOut == UCACONFIG_ACT_VEL){ /* 速度形 */
    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE){
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = deltaMv + outRb.value;
● }
```

重要

プログラムでは位置形または速度形のどちらかの処理だけを記述する場合でも、CSTM-C のビルダ定義項目「制御出力動作」は、プログラムに合わせた指定が必要です。たとえば、プログラムを位置形で記述する場合には、ビルダ定義項目「制御出力動作」に「位置形」を指定してください（ビルダ定義項目「制御出力動作」のデフォルトは「位置形」です）。システム内部にビルダ定義項目「制御出力動作」の指定に従い動作する部分があるので、プログラムの記述とビルダ定義項目が合っていないと制御出力動作が正しく動作しません。システムは UcaRWWriteMvToOutSub 内部の出力値に関する処理などでビルダ定義項目が「位置形」か「速度形」かを参照し、指定により動作を分けています。

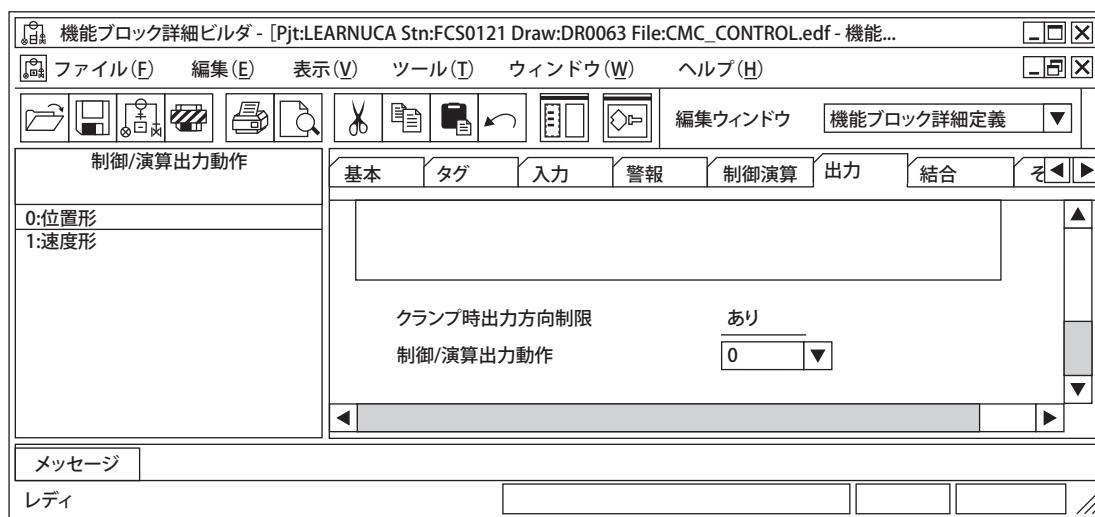


図 制御出力動作の指定

060337J.ai

■ リセットリミット

リセットリミット機能を使用しない場合は、UcaCtrlResetLimitOprt 関数の呼び出しを省略することができます。以下の先頭に●印のある行を省略できます。

```
●          /* リセットリミット */
●          rtnCode = UcaCtrlResetLimitOprt(bc, &deltaMv,
UCAOPT_CTRL_TSCTIME);
```

重要

制御出力動作が位置形で PID 演算や PI 演算のように積分項のある制御演算をする場合には、リセットリミット機能を省略できません。

■ 不感帶動作

不感帯動作機能を使用しない場合は、UcaConfigDeadband 関数と UcaCtrlDeadband 関数の呼び出しを省略することができます。以下の先頭に●印のある行を省略できます。

```
/* 制御出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS) { /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimit0prt(bc, &deltaMv, UCAOPT_CTRL_TSCTIME);

    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE) {
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }
}

/* MVBB 設定 */
rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
mvbbVal = deltaMv + lastMvblVal;

} else if (configOut == UCACONFIG_ACT_VEL) { /* 速度形 */
    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE) {
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }
}

/* MVBB 設定 */
rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
mvbbVal = deltaMv + outRb.value;
}
```

補足 ビルダ定義項目のデフォルトファイル作成機能により、システムが規定しているビルダ定義項目のデフォルトをユーザが変更することができます。この機能は、システムのデフォルトと異なる指定の機能ブロックを多数作成するときに便利です。

参照 デフォルトと異なる指定の機能ブロックの作成の詳細については、以下を参照してください。
「4.5 ビルダ定義項目の決定」の「■ ビルダ定義項目のデフォルトファイル作成」

6.4 PID調節ブロックと同じ動作をするCSTM-C

ユーザカスタムアルゴリズム作成用ライブラリには、PID制御演算を行う関数UcaCtrlPidが用意されています。また、標準の調節ブロックと同様の制御周期動作を実現するための関数UcaCtrlPidTimingがあります。これらの関数を使用することにより、標準のPID調節ブロックと同じ動作をするユーザカスタムアルゴリズムを記述することができます。

この節では、PID調節ブロックと同じ動作をするユーザカスタムアルゴリズムをサンプルとして、PID制御演算をし、制御周期に従い出力処理を行うCSTM-Cを説明します。ユーザはこのサンプルを改造することにより、PID調節ブロックにユーザ独自の動作を追加したCSTM-Cを実装することができます。

ここで説明するPID調節ブロックと同じ動作をするサンプルは、入出力補償や不感帯動作を行います。

参照 入出力補償、不感帯動作のプログラミングの詳細については、以下を参照してください。

「[6.3 制御演算関数を使用した CSTM-C](#)」

表 制御演算関数

機能	関数名	説明	使用／不使用	
			6.3節	6.4節
制御周期処理	UcaCtrlPidTiming	制御周期のカウンタ操作をします。		○
PID 制御演算	UcaCtrlPid	PID 制御演算を実行します。		○
制御ホールド	UcaCtrlHoldCheck	制御ホールド要求が設定されているか検査します。	○	○
入出力補償	UcaCtrlCompensation	入力補償または出力補償を実行します。	○	○
リセットリミット	UcaCtrlResetLimitOprt	リセットリミット機能を実行します。	○	○
不感帯動作	UcaCtrlDeadband	不感帯動作を実行します。	○	○
制御演算初期化	UcaCtrlInitCheck	制御演算の初期化要求が設定されているか検査します。	○	○
	UcaCtrlInitClear	制御演算の初期化要求を解除します。	○	○
	UcaCtrlFuncInit	制御演算関数が保持する内部パラメータを初期化します。	○	
	UcaCtrlPidInit	UcaCtrlPid と制御演算関数が保持する内部パラメータを初期化します。		○

○： 使用する

表 制御演算に関するビルダ項目を取得する関数

ビルダ定義項目	関数名	説明	使用／不使用	
			6.3節 (*1)	6.4節 (*1)
制御動作方向	UcaConfigDirection	「逆動作 (*2)」「正動作」から指定値を取得	○	○
制御／演算出力動作	UcaConfigOutput	「位置形 (*2)」「速度形」から指定値を取得	○	○
入出力補償	UcaConfigCompensation	「なし (*2)」「入力補償」「出力補償」から指定値を取得	○	○
不感帯動作	UcaConfigDeadband	「なし (*2)」「あり」から取得	○	○

○： 使用する

*1：・ 6.3 節のサンプルは、PID 制御演算の関数 UcaCtrlPid を使用しないでユーザ定義の制御演算を行い、実効スキャン周期に出力をする（制御周期は使わない）プログラムです。

・ 6.4 節のサンプルは、PID 制御演算の関数 UcaCtrlPid を使用し、制御周期に出力をするプログラムです。

*2： デフォルト

■ PID調節ブロックと同じ動作をするサンプルプログラム

PID 調節ブロックと同じ動作をするサンプルプログラムについて説明します。サンプルソリューションは 1 章の準備作業により以下の作業フォルダにコピーされています。

<ドライブ名>¥UcaWork¥UcaSamples¥_SMPL_PID

ユーザカスタムアルゴリズム _SMPL_PID の pid.c を見てください。このプログラムは、機能ブロック定周期処理 (UcaBlockPeriodical) のみ処理をします。機能ブロック定周期処理は、入力処理 (pid_input)、制御演算処理 (pid_control)、出力処理 (pid_output) の 3 つのユーザ定義関数を指定して UcaCtrlHandler を呼び出します。

```
/*
 * プロトタイプ宣言
 */
I32 pid_input(UcaBlockContext bc);
I32 pid_control(UcaBlockContext bc, F64 *mvbbPtr, F64 *mvblPtr);
I32 pid_output(UcaBlockContext bc);
.....
/*
* <<FNH>>*****
*
* Function name:      UcaBlockPeriodical
* Return value:       SUCCEED          正常終了
*                      UCAERR_NOPROC    処理なし
*                      UCAERR_STOPME    処理続行不能
*
* description:        機能ブロック定周期処理
*
*>>HNF<<*****
*/
UCAUSER_API I32 UCAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnCode;           /* リターンコード */

    rtnCode = UcaCtrlHandler( bc,
                            pid_input,
                            pid_control,
                            pid_output,
                            UCACTRL_ALLMODE,    ↑ 任意のブロックモードで制御演算関数を
                            NOOPTION);           ↓ 呼び出す
    return SUCCEED;
}
```

UcaCtrlHandler の 5 番目の引数には、UCACTRL_ALLMODE を指定しています。

● ユーザ定義の入力処理

ユーザ定義の入力処理（関数 pid_input）は、UcaRWReadIn で IN 端子からデータアイテム RV にデータを読み込み、読み込んだデータを UcaRWSetPv でデータアイテム PV に設定します。入力処理は、6.3 節のサンプルと同じです。

```
/*
* <<FNH>>*****
*
* Function name: pid_input
* Return value: SUCCEED正常終了
*
* description: 入力処理
*
*>>HNF<<*****
*/
I32 pid_input(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode; /* リターンコード */
    I32 rtnReadIn; /* IN 端子入力リターンコード */

    /* IN 端子から RV にデータを読み込み */
    rtnReadIn = UcaRWReadIn(bc, NOOPTION);

    /* RV から PV にデータを設定 */
    rtnCode = UcaRWSetPv(bc, NULL, rtnReadIn, NOOPTION);

    return SUCCEED;
}
```

● ユーザ定義の出力処理

ユーザ定義の出力処理 (pid_output 関数) は、UcaRWWWriteMvToOutSub で OUT 端子と SUB 端子への出力処理を行います。制御周期を使用するため、オプションに UCAOPT_RW_CTRLTIMING を指定しています。

```
/*
* <<FNH>>*****
*
* Function name: pid_output
* Return value: SUCCEED 正常終了
*
* description: 出力処理
*
*>>HNF<<*****
*/
I32 pid_output(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnCode; /* リターンコード */

    /* OUT端子とSUB端子からデータを出力 */
    /*
     * UCAOPT_RW_CTRLTIMINGオプションを指定し
     * AUT,CAS,RCASモードで制御周期でない場合には、出力をホールド
     */
    rtnCode = UcaRWWWriteMvToOutSub(bc, UCAOPT_RW_CTRLTIMING); ↑ AUT、CAS、RCAS時は
                                                                ↓ 制御周期のみ出力

    return SUCCEED;
}
```

● ユーザ定義の制御演算処理

PID 制御演算を行うユーザ定義関数 pid_control について説明します。ここでは、ソースコード全体を示します。

```

/*
* <<FNH>>*****
*
* Function name: pid_control
* Return value:  SUCCEED 正常終了
*
* description:  PID演算処理
*
*>>HNF<<*****
*/
I32 pid_control(
    UcaBlockContext bc,          /* (IN/OUT): ブロックコンテキスト */
    F64 *mvbbPtr,               /* (IN/OUT): 出力補償前出力値ポインタ */
    F64 *mvblPtr                /* (IN/OUT): 出力リミット前出力値ポインタ */
)
{
    F64S pv;                   /* PV */
    F32S sv;                   /* SV */
    F64 ev;                   /* 実効偏差 */
    F64 cv;                   /* CV */
    F64 calc;                 /* PID演算結果 */
    F64 deltaMv;              /* 出力変化値 */
    F64S outRb;               /* 出力読み返し値 */
    F64 mvbbVal;              /* 出力補償前出力値 */
    F64 mvblVal;              /* 出力リミット前出力値 */
    F64 lastMvblVal;          /* 前回出力リミット前出力値 */
    I16 configDir;             /* ビルダ定義項目動作方向 */
    I16 configOut;              /* ビルダ定義項目出力動作 */
    I16 configComp;             /* ビルダ定義項目入出力補償 */
    I16 configDeadband;         /* ビルダ定義項目不感帯動作 */
    BOOL isCtrlTime;            /* 制御周期かどうか */
    BOOL isCtrlHold;             /* 制御ホールドかどうか */
    BOOL isCtrlInit;             /* 制御初期化要求かどうか */
    I32 rtnCode;                /* リターンコード */

    /* 制御周期かどうかを確認 */
    rtnCode = UcaCtrlPidTiming(bc, &isCtrlTime, NOOPTION);
    ↑   ↓   制御周期カウンタ処理

    /* 制御ホールド確認 */
    rtnCode = UcaCtrlHoldCheck(bc, &isCtrlHold);
}

```

(続く)

(続き)

```

/* 制御演算処理 */
/*****************************************/
/* 制御周期の場合に制御演算を行います。 */
/* 非制御周期の場合や制御ホールド時には、制御演算を行いません。 */
/*****************************************/

if (isCtrlTime && !isCtrlHold){ /* 制御演算状態(制御周期かつ制御ホールドでない場合) */

    /* 入力補償 */
    rtnCode = UcaDataGetPv(bc, &pv, NOOPTION);
    cv = pv.value;
    rtnCode = UcaConfigCompensation(bc, &configComp);
    if (configComp == UCACONFIG_COMP_INPUT){
        rtnCode = UcaCtrlCompensation(bc, &cv, NOOPTION);
    }

    /* 制御初期化 */
    rtnCode = UcaCtrlInitCheck(bc, &isCtrlInit);
    if (isCtrlInit){
        rtnCode = UcaCtrlPidInit(bc, cv, NOOPTION);
        rtnCode = UcaCtrlInitClear(bc);
    }

    /* 制御演算 */
    rtnCode = UcaCtrlPid(bc, cv, &calc, NOOPTION);

    /* レンジ変換 */
    rtnCode = UcaDataConvertRange(bc, calc, &deltaMv, UCAOPT_RANGE_DIFFVALUE);

    /* 制御動作方向 */
    rtnCode = UcaConfigDirection(bc, &configDir);
    if (configDir == UCACONFIG_DIR_REV){ /* 逆動作 */
        deltaMv = -deltaMv;
    }

    /* 実効偏差計算 */
   /*****************************************/
    /* 不感帯動作UcaCtrlDeadband()で使用します */
   /*****************************************/
    rtnCode = UcaDataGetSv(bc, &sv, NOOPTION);
    ev = cv - sv.value;
}

```

(続く)

(続き)

```

/* 制御出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS) {           /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimit0prt(bc, &deltaMv, NOOPTION); ↑オプションNOOPTION指定
                                                    ↓により、制御周期で処理

    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE){
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
    mvbbVal = deltaMv + lastMvblVal;

} else if (configOut == UCACONFIG_ACT_VEL) { /* 速度形 */
    /* 不感帯動作 */
    rtnCode = UcaConfigDeadband(bc, &configDeadband);
    if (configDeadband == UCACONFIG_DEADBAND_TRUE){
        rtnCode = UcaCtrlDeadband(bc, ev, &deltaMv, NOOPTION);
    }

    /* MVBB設定 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = deltaMv + outRb.value;
}

/* MVBL設定 */
/* 出力補償 */
mvblVal = mvbbVal;
rtnCode = UcaConfigCompensation(bc, &configComp);
if (configComp == UCACONFIG_COMP_OUTPUT) {
    rtnCode = UcaCtrlCompensation(bc, &mvblVal, NOOPTION);
}

} else {/* 非制御状態(非制御周期または制御ホールドの場合) */

    ****
    /* 出力動作が位置形の場合、MVBLには前回MVBL値を設定します。          */
    /* 速度形の場合、MVBLには出力読み返し値を設定します。                      */
    ****
}

```

(続く)

(続き)

```
/* 出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS){ /* 位置形 */
    rtnCode = UcaDataGetMvbl(bc, &lastMvblVal, NOOPTION);
    mvbbVal = lastMvblVal;

} else if (configOut == UCACONFIG_ACT_VEL){ /* 速度形 */
    rtnCode = UcaDataGetReadback(bc, &outRb, NOOPTION);
    mvbbVal = outRb.value;

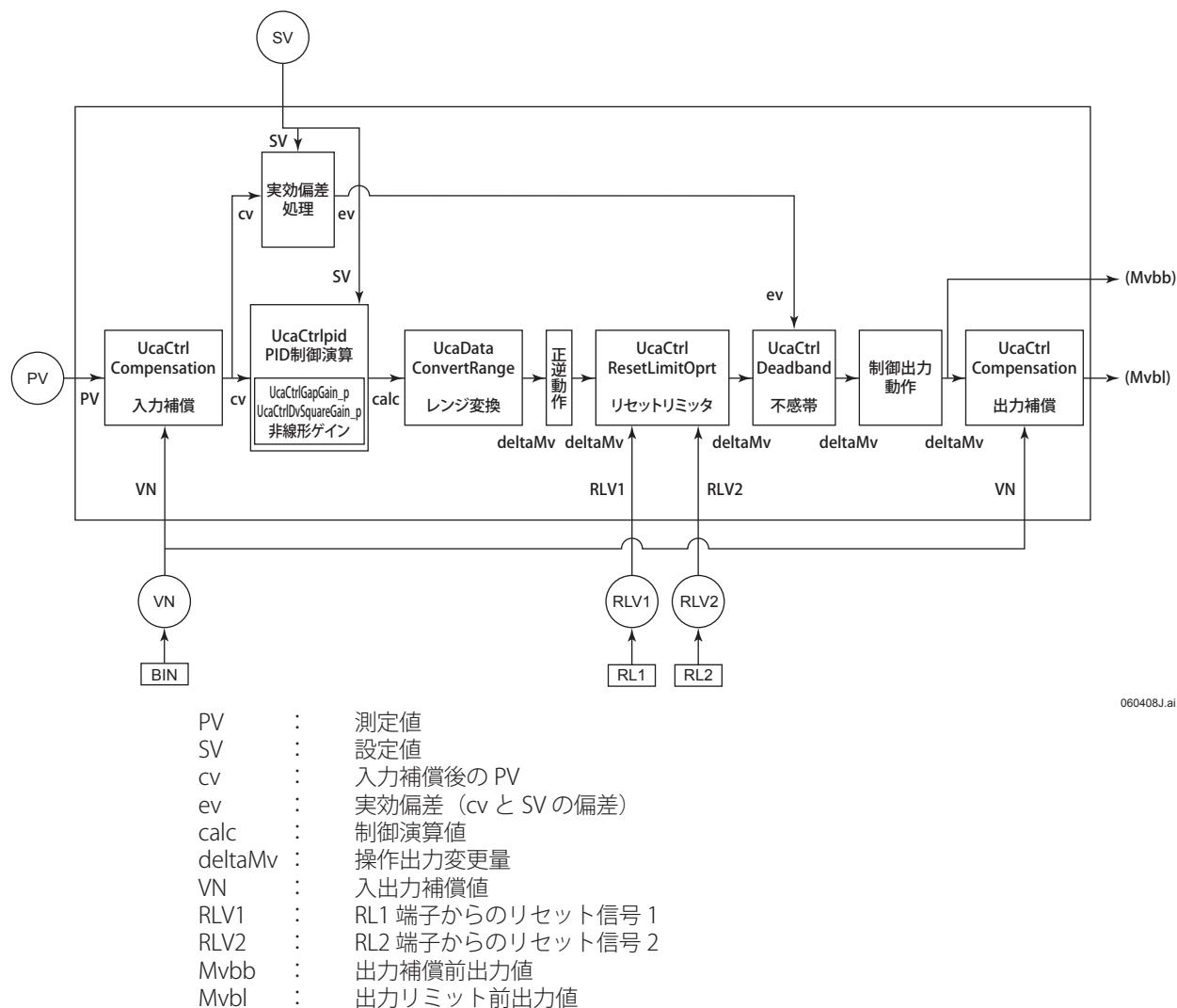
}

/*****************************************/
/* MVBL には MVBB 値を設定します。 */
/*****************************************/
mvblVal = mvbbVal;
}

/* MVBB、MVBL を保存 */
*mvbbPtr = mvbbVal;
*mvblPtr = mvblVal;

return SUCCEED;
}
```

この制御演算処理におけるデータの流れを以下に示します。



060408J.ai

図 PID制御演算を使った制御演算のデータの流れ

6.4.1 UcaCtrlHandlerの演算処理実行ブロックモード

UcaCtrlHandlerの演算処理実行ブロックモードについて説明します。

UcaCtrlHandler の第 5 引数には、演算処理実行ブロックモードを指定します。このプログラムは、UCACTRL_ALLMODE を指定しています。

```
.....
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnCode;           /* リターンコード */

    rtnCode = UcaCtrlHandler( bc,
        pid_input,
        pid_control,
        pid_output,
        UCACTRL_ALLMODE,   ↑ 演算処理実行ブロックモードの指定
        NOOPTION );
}

return SUCCEED;
}
```

演算処理実行モードには、以下の 2 つの指定があります。

表 UcaCtrlHandlerの演算処理実行モード

	UCACTRL_ALLMODE	UCACTRL_AUTCASRCAS
動作	UcaCtrlHanlder は、すべてのブロックモードにおいてユーザ記述の制御演算関数を呼び出します。	UcaCtrlHanlder は、ブロックモード AUT, CAS, RCAS の場合にのみユーザ記述の制御演算関数を呼び出します。
使い分け	UcaCtrlPid 関数を使用する場合に指定します。	UcaCtrlPid 関数を使用しない場合に指定します。

■ UcaCtrlPidを使用する場合は、UCACTRL_ALLMODEを指定する

標準の PID 調節ブロックは、自動運転動作の立ち上がり（ブロックモードが MAN から AUT に遷移したあとなど）を早くするために、ブロックモードが AUT、CAS、RCAS 以外の場合に特殊処理をしています。PID 制御演算を行う UcaCtrlPid 関数には、これと同じ特殊処理が組み込まれています。このため、ユーザ記述の制御演算関数で UcaCtrlPid 関数を呼び出す場合には、UcaCtrlHanlder の演算処理実行モードに UCACTRL_ALLMODE を指定して、どのブロックモードでも UcaCtrlPid が実行されるようにしてください。

重要

演算処理実行モードに UCACTRL_ALLMODE を指定した場合でも、ユーザ記述の制御演算関数が引数に返す出力補正前出力値と出力リミット前出力値が有効なのは、AUT、CAS、RCAS 動作をする場合だけです。それ以外の場合は、UcaCtrlHandler の出力値作成処理は、ユーザ記述の制御演算関数の引数に返された出力補正前出力値と出力リミット前出力値を無視します。つまり、出力値をユーザ定義のプログラムで決定できるのは、AUT、CAS、RCAS の場合だけであり、それ以外のときは UcaCtrlHandler が出力値を決定します。

6.4.2 実効スキャン周期と制御周期

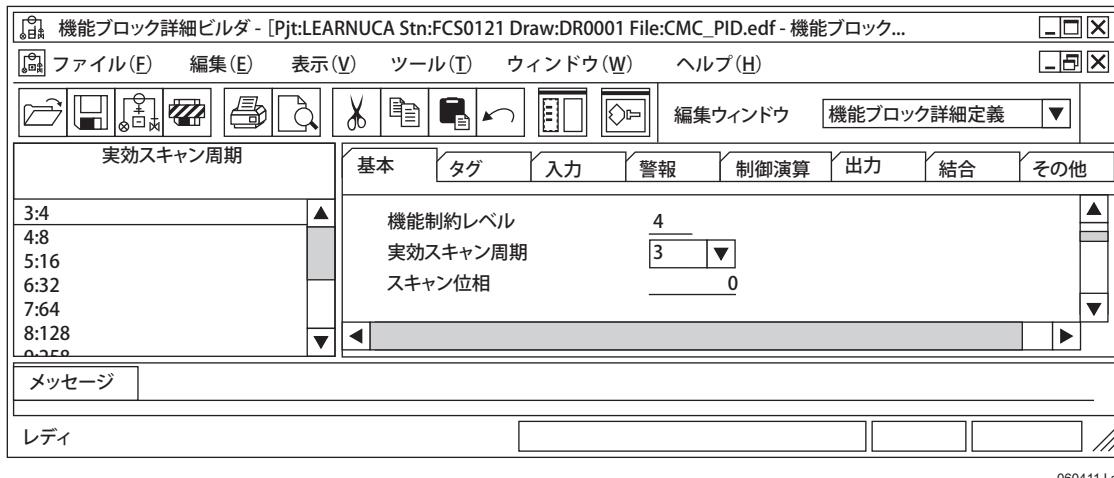
連続制御形ユーザカスタムブロック（CSTM-C）を定周期起動する周期は、実効スキャン周期で指定します。制御周期は、CSTM-Cが自動運転（AUT、CAS、RCAS）のときの制御演算処理および出力処理を実行する周期です。CSTM-Cの制御周期は、常に実効スキャン周期の整数倍になります。

参照 制御周期の詳細については、以下を参照してください。

[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\) 「7.1.4 調節ブロックの制御周期」](#)

■ 実効スキャン周期と制御周期の指定

CSTM-C の実効スキャン周期は、機能ブロック詳細ビルダの基本タブシートで指定します。



060411J.ai

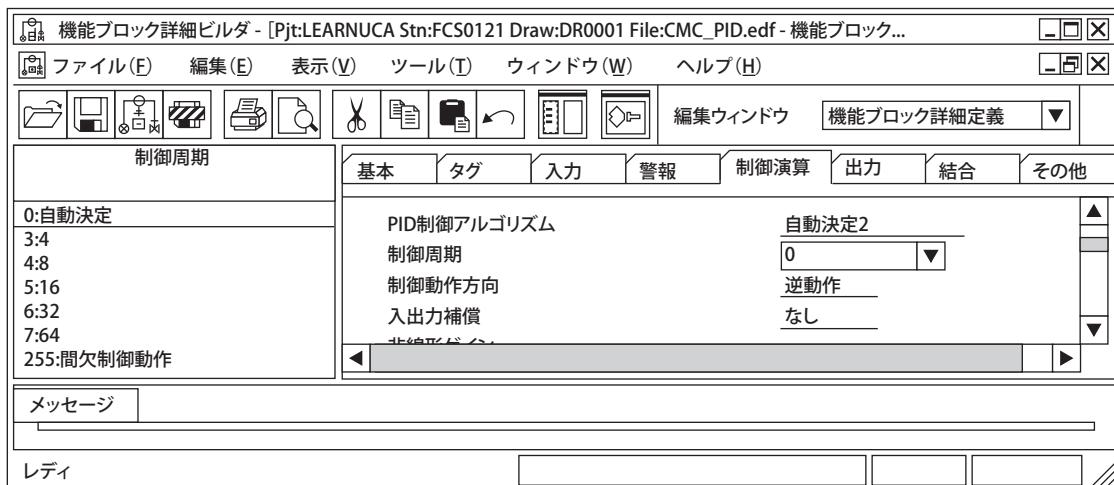
図 実効スキャン周期の指定

* : デフォルトは 4 秒（指定は「3：4」）です。

参照 実効スキャン周期とスキャン位相の詳細については、以下を参照してください。

APCS ユーザカスタムブロック (IM 33J15U20-01JA)

制御周期は、機能ブロック詳細ビルダの制御演算タブシートで指定します。



060412J.ai

図 制御周期の指定

* : • デフォルトは「0：自動決定」です。
• 「自動決定」を指定したときは、次表のように積分時間(データアイテム I)により制御周期が決まります。

表 自動決定にした場合の制御周期 (秒)

積分時間 (秒) (データアイテム I)	基本周期 (秒)		
	4	8	16
1 ~ 255	4	8	16
256 ~ 1023	8	8	16
1024 ~ 2047	16	16	16
2048 ~ 10000	32	32	32

* : 「実効スキャン周期」が「制御周期」より大きい場合には、制御周期は実効スキャン周期に一致します。

■ 実効スキャン周期と制御周期

ユーザカスタムブロック実行管理部は、実効スキャン周期ごとにユーザカスタムアルゴリズムの機能ブロック定周期処理をするユーザ定義関数 UcaBlockPeriodical を呼び出します。「自動運転のときは制御周期の場合だけ制御演算と出力処理をする」という動作は、ユーザプログラムが UcaCtrlPidTiming を使用して実現します。

UcaCtrlPidTiming は、制御周期に関する次の処理をします。

- ・システムが CSTM-C ごとに内部に保持している制御周期カウンタの操作をします。システムは制御周期カウンタを今回の処理タイミングが制御周期に該当するか否かを判定するために使用します。
- ・UcaCtrlPidTiming は、CSTM-C のビルダ定義項目「制御周期」の指定値を内部的に参照します。
- ・UcaCtrlPidTiming は、今回の処理タイミングが制御周期か否かを引数に返します。

重要

- ・PID 制御演算をする UcaCtrlPid 関数は、UcaCtrlPidTiming の制御周期カウンタが管理する時間を利用しています。UcaCtrlPid を使用するプログラムは、UcaCtrlPidTiming も呼び出してください。
- ・リセットトリミットをする UcaCtrlResetLimitOpt 関数に、NOOPTION を指定する（制御時間に実効スキャン周期を使用する UCAOPT_CTRL_TSCTIME オプションを指定しない）場合には、UcaCtrlPidTiming を呼び出す必要があります。UcaCtrlResetLimitOpt は、UcaCtrlPidTiming の制御周期カウンタが管理する時間を利用しています。

補足

UcaCtrlPidTiming の名前に「Pid」という文字が含まれているのは、PID 制御演算（UcaCtrlPid 関数）を使用するときに UcaCtrlPidTiming を使用するからです。ただし、UcaCtrlPid を使用しない場合でも UcaCtrlPidTiming を使用して「制御周期」動作を実現することができます。

制御周期の動作を実現するために、サンプルプログラムでどのように UcaCtrlPidTiming を使っているか説明します。システムは実効スキャン周期ごとに（ビルダ定義項目「制御周期」の指定とはまったく無関係に）、機能ブロック定周期処理を呼び出します。

```

...
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode; /* リターンコード */

    rtnCode = UcaCtrlHandler(bc,
        pid_input,
        pid_control,
        pid_output,
        UCACTRL_ALLMODE,
        NOOPTION);

    return SUCCEED;
}

```

UcaBlockPeriodical から呼び出される UcaCtrlHandler は、3 つのユーザ定義関数 pid_input（入力処理）、pid_control（制御演算）、pid_output（出力処理）を毎回呼び出します。

● 入力処理

入力処理の関数 pid_input が呼び出されると UcaRWReadIn で IN 端子からデータアイテム RV にデータを取得し、UcaRWSetPv でデータアイテム RV からデータアイテム PV にデータを設定します。

```
.....
I32 pid_input(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    I32 rtnCode;           /* リターンコード */
    I32 rtnReadIn;         /* IN 端子入力リターンコード */

    /* IN 端子から RV にデータを読み込み */
    rtnReadIn = UcaRWReadIn(bc, NOOPTION);

    /* RV から PV にデータを設定 */
    rtnCode = UcaRWSetPv(bc, NULL, rtnReadIn, NOOPTION);

    return SUCCEED;
}
```

したがって、CSTM-C の PV は「制御周期」の指定には関係なく、「実効スキャン周期」ごとに更新されます。また、UcaRWSetPv により検出される入力上限アラーム (HI アラーム) や入力下限アラーム (LO アラーム) などは、実効スキャン周期ごとに検査されます。

● 制御演算処理

制御演算の関数 pid_control は、制御周期にのみ制御演算を実行するために、次の部分で UcaCtrlPidTiming を呼び出して処理を分けています。

```
.....
/* 制御周期かどうかを確認 */
rtnCode = UcaCtrlPidTiming(bc, &isCtrlTime, NOOPTION); ↑ 制御周期カウンタ処理

/* 制御ホールド確認 */
rtnCode = UcaCtrlHoldCheck(bc, &isCtrlHold);

/* 制御演算処理 */
/*****************/
/* 制御周期の場合に制御演算を行います。 */
/* 非制御周期の場合や制御ホールド時には、制御演算を行いません。 */
/*****************/

if (isCtrlTime && !isCtrlHold){ /* 制御演算状態(制御周期かつ制御ホールドでない場合) */

    /* 入力補償 */
    rtnCode = UcaDataGetPv(bc, &pv, NOOPTION); ← 制御周期か判定
    cv = pv.value;
    rtnCode = UcaConfigCompensation(bc, &configComp);
    if (configComp == UCACONFIG_COMP_INPUT) {

        rtnCode = UcaCtrlCompensation(bc, &cv, NOOPTION);
    }
}
.....
```

制御演算処理をするユーザ定義関数 pid_control は、処理の最初で UcaCtrlPidTiming を呼び出し制御周期カウンタの処理を実行し、また引数 isCtrlTime に制御周期か否かを取得します。引数 isCtrlTime には、今回の処理タイミングが制御周期ならば TRUE、制御周期でなければ FALSE が設定されます。

UcaCtrlTiming を呼び出した少しあとの if 文で「制御周期かつ制御ホールドでない」ことを判定し、条件が成立したときにのみ、以後の制御演算を実行しています。

参照 制御ホールドに関するプログラミングの詳細については、以下を参照してください。
[「6.3.1 制御ホールド」](#)

この他に、制御演算処理ではリセットリミットを行う UcaCtrlResetLimitOprt のオプションを NOOPTION にし、リセットリミットの制御時間に制御周期を使用（デフォルト）するようにします。

```
/* 制御出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS){ /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimitOprt(bc, &deltaMv, NOOPTION); ← NOOPTIONを指定
}
.....
```

● 出力処理

出力処理の関数 pid_output について説明します。自動運転時（ブロックモード AUT、CAS または RCAS）には制御周期にのみ出力処理をするために、ユーザ記述の出力処理関数 pid_output は UcaRWWWriteMvToOutSub に UCAOPT_RW_CTRLTIMING オプションを指定して呼び出しています。

```
.....  
I32 pid_output(  
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */  
)  
{  
    I32 rtnCode; /* リターンコード */  
  
    /* OUT端子とSUB端子からデータを出力 */  
    /*  
     * UCAOPT_RW_CTRLTIMINGオプションを指定し  
     * AUT,CAS,RCASモードで制御周期でない場合には、出力をホールド  
     */  
    rtnCode = UcaRWWWriteMvToOutSub(bc, UCAOPT_RW_CTRLTIMING);  
  
    return SUCCEED;  
}
```

← 自動運転時に制御周期のみ出力処理をするように指定

UcaRWWWriteMvToOutSub は、UCAOPT_RW_CTRLTIMING オプションを指定されると内部で UcaCtrlPidTiming を呼び出して（今回が）制御周期か否かを判定し、自動運転中（AUT、CAS、または RCAS 動作）の場合は、制御周期のときだけ出力処理を行います。

■ 制御周期の使用・不使用によるプログラムの相違点

ビルダ定義項目「制御周期」に従い動作する CSTM-C のプログラムについて説明してきました。6.3 節の PI 制御をするサンプルは、制御周期の指定は無視しています。この 6.4 節のプログラムと 6.3 節のプログラムを比較して、制御周期に関するプログラムの書き方を整理します。

表 制御周期の使用・不使用によるプログラムの相違点

	制御周期を使用	制御周期を不使用
ビルダ項目「制御周期」の指定	ビルダの指定が有効になり、指定に従い動作します	無視します
サンプル	6.4 節 PID 調節ブロックと同じ動作をするサンプル 6.6 節 PI 制御をするサンプル（制御周期を使用）	6.3 節 PI 制御をするサンプル
ビルダ項目「制御周期の指定」	有効(UcaCtrlPidTiming の内部で参照)	無視
UcaCtrlPidTiming	呼び出す	呼び出さない
入力処理	実効スキャン周期ごとに呼び出され、測定値 PV も毎実効スキャン周期ごとに更新する	
出力処理	UcaRWWWriteMvToOutSub のオプションに UCAOPT_RW_CTRLTIMING を指定することにより、自動運転では制御周期にのみ出力処理を実行する	UcaRWWWriteMvToOutSub のオプションに NOOPTION を指定することにより、自動運転でも毎実効スキャン周期に出力処理を実行する
制御演算処理	UcaCtrlPidTiming で制御周期か否かを判定し、制御周期の場合のみ制御演算を実行	制御周期か否かで処理を分けない（UcaCtrlPidTiming は呼び出さない）
前回制御演算実行からの時間差	UcaCtrlPidGetTime で前回制御周期からの時間差を取得	UcaDataGetTsc でデータアイテム TCS に保持されている実効スキャン周期を取得
UcaCtrlResetLimitOpt のオプション	制御時間に制御周期を使用するので、NOOPTION を指定 (*1)	制御時間に実効スキャン周期を使用するので、UCAOPT_CTRL_TSCTIME を指定

参照

前回制御周期からの時間差取得の詳細については、以下を参照してください。

[「6.6 カスケード結合と実効スキャン周期」の「■ 制御周期を有効にする」のサンプル _SMPL_CONTROL2](#)

*1： UcaCtrlResetLimitOpt に NOOPTION を指定した場合は、UcaCtrlPidTiming が作り出した制御周期を使用します。つまり、UcaCtrlResetLimitOpt に NOOPTION を指定する場合には（制御時間に実効スキャン周期を使用する UCAOPT_CTRL_TSCTIME を指定しない場合には）、UcaCtrlPidTiming を使用する必要があります。

補足

PID 制御演算をする関数 UcaCtrlPid には、「制御時間に実効スキャン周期を使用する」 UCAOPT_CTRL_TCSTIME オプションがあります。UCAOPT_CTRL_TSCTIME オプションを指定する場合には、上の表の「制御周期を不使用」に従ってプログラムを作成し、制御演算処理は UcaCtrlPid 関数を使用することになります。しかし、本書では、CSTM-C は、PID 制御演算をする場合には、制御周期とセット使用することを推奨します（UcaCtrlPid 関数のオプション指定は NOOPTION です）。PID 制御演算と制御周期をセットにした CSTM-C では、標準の PID 調節ブロックと同じ動作を土台にユーザ独自の動作を追加することができます。このため、本節では PID 制御演算と制御周期機能と合わせて使用する場合のプログラミングを説明しています。

■ 間欠制御

間欠制御は、自動運転中（AUT、CAS、RCAS）に制御スイッチ（データアイテム CSW）が 1 になっている実効スキャン周期のみ制御演算処理と出力処理を実行する制御動作です。制御動作を実行すると、制御スイッチ（データアイテム CSW）はシステムが 1 から 0 にします。

CSTM-C のビルダ定義項目「制御周期」に「間欠制御」を指定する場合は、UcaCtrlPidTiming を使用してください。この節で説明した PID 調節ブロックと同じ動作をするサンプルは、間欠制御を指定してもまったく修正なしで動作可能です。つまり、間欠制御を使用する場合のプログラミングは「制御周期」を使用する場合のプログラミングと同じであり、UcaCtrlPidTiming の使い方には「間欠制御だから特別な記述」は不要です。

- ・ 間欠制御を使用する場合には、機能ブロック定周期処理（UcaBlockPeriodical）にユーザプログラムを記述してください。UcaBlockPeriodical は、実効スキャン周期ごとに呼び出されます。
- ・ 実効スキャン周期（機能ブロック定周期処理 UcaBlockPeriodical が呼び出されてから返るまでの間）に一回以上 UcaCtrlPidTiming を呼び出してください。一度の処理タイミングで UcaCtrlPidTiming を複数回呼び出しても問題ありません。
- ・ UcaCtrlPidTiming は、制御スイッチ（データアイテム CSW）が 1 になっていると今回の処理タイミングを制御周期と判定し、UcaCtrlPidTiming は制御スイッチ（データアイテム CSW）を 1 から 0 にします。ユーザプログラムでデータアイテム CSW を 0 にする必要はありません。同じ処理タイミングで複数回 UcaCtrlPidTiming を呼び出した場合、データアイテム CSW を 1 から 0 に戻すのは 1 回目の呼び出し時ですが、UcaCtrlPidTiming の 2 回目以後の呼び出しでも「制御周期である」と正しく判定できます。
- ・ 間欠制御を指定した場合、実効スキャン周期の処理タイミングで制御スイッチ（データアイテム CSW）が 1 であれば UcaCtrlPidTiming が「今回は制御周期である」と判定し、制御周期の動作が行われます（CSTM-C のデータアイテムを 0 から 1 に変更したタイミングで制御動作が実行されるわけではありません）。このときに UcaCtrlPid を呼び出すと、UcaCtrlPid は実効偏差 ev (= 入力補償後の PV - SV) など、PID 制御演算に必要なデータをユーザカスタムブロック内部の領域に保持します。このデータは、次回の UcaCtrlPid 呼び出し時（次に CSW が 0 から 1 に変化したとき）に「前回値」として使用されます。
- ・ CSTM-C のデータアイテムを 0 から 1 に変更したタイミングではデータアイテム CSW に対する機能ブロックデータ設定時特殊処理が実行されます。しかし、機能データ設定時特殊処理で制御演算の処理は実行しないでください（機能ブロックデータ設定時特殊処理では自ブロックのデータアクセスしかできませんので、制御演算は実行できません）。

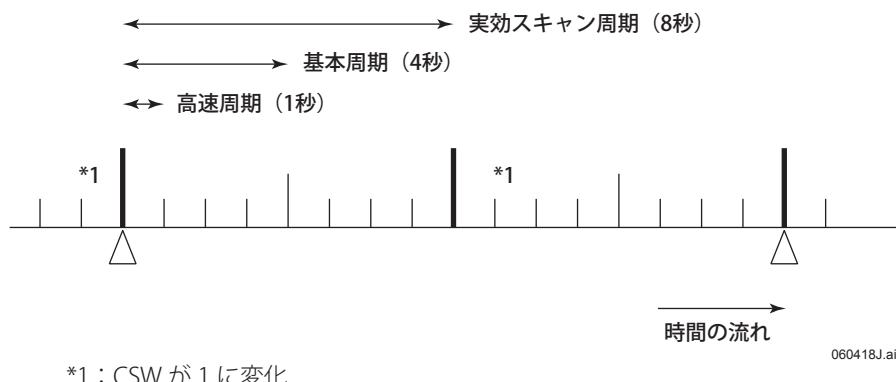


図 高速周期のシーケンステーブルなどでCSWを0から1に変化するときのCSTM-Cの間欠制御動作

- ・ 実効スキャン周期（太字の縦棒）で、CSWが1（左側に *1印）のとき制御周期となります。
- * : 「CSWが0から1に変化」するのと「CSTM-Cの実効スキャン周期」が同じ周期の場合には、「CSWを0から1に変化」する機能ブロックの処理タイミングがCSTM-Cの処理タイミングより前であれば、CSTM-Cは制御周期と判定します。つまり、機能ブロックの実行順でCSTM-Cの制御周期となるかが決まります。

■ UcaCtrlPidTimingを使用するポイント

制御周期を使用する場合の UcaCtrlPidTiming を使用するポイントを整理します。

- ・定周期で処理をする場合には、実効スキャン周期（機能ブロック定周期処理 UcaBlockPeriodical が呼び出されてからリターンするまでの間）に一回以上 UcaCtrlPidTiming を呼び出してください。
- ・一度の処理タイミングで UcaCtrlPidTiming を複数回呼び出しても問題ありません。UcaCtrlPidTiming が制御周期カウンタ処理を行うのは、処理タイミングにおける最初の 1 回だけです。2 回目以後の呼び出しでは、今回が制御周期か否かを判定するだけです。
- ・前回の制御周期から今回の制御周期までの時間の差分 (ΔT) を取得するには、UcaCtrlPidTiming を呼び出した後に（同じ処理タイミング内で）UcaCtrlPidGetTime を呼び出して、制御カウンタ値を取得します。UcaCtrlPidGetTime が引数に返す制御カウンタ値は、前回の制御周期からの時間の差分です。時間の差分の単位は秒で、分解能は基本周期です。
- ・間欠制御の場合でも、同じ方法で前回の制御周期から今回の制御周期までの時間の差分を取得できます。

■機能ブロックワンショット起動処理

機能ブロックワンショット起動処理で、前回の制御実行タイミングから今回の制御実行タイミングまでの時間の差分 (ΔT) を必要とするような制御演算をするユーザプログラムを記述する場合の注意事項について説明します。

重要

前回の制御実行タイミングから今回の制御実行タイミングまでの時間の差分 (ΔT) を必要とするような制御演算をする CSTM-C の処理タイミングをシーケンスブロックなど外部の機能ブロックで決定したい場合には、ワンショット起動ではなく、制御周期の「間欠制御」動作を使用することを推奨します。

処理の内部で、時間の差分が使用している制御演算関数を以下に示します。時間の差分欄に「使用」と書いてある関数を呼び出しているプログラムをワンショット起動する場合は、注意が必要です。

表 制御演算関数における時間の差分の使用

機能	関数名	説明	時間の差分	注意
制御周期処理	UcaCtrlPidTiming	制御周期のカウンタ操作をします。	使用	(*)1)
PID 制御演算	UcaCtrlPid	PID 制御演算を実行します。	使用	(*)1)
制御ホールド	UcaCtrlHoldCheck	制御ホールド要求が設定されているか検査します。		
入出力補償	UcaCtrlCompensation UcaCtrlCompensation_p	入力補償または出力補償を実行します。		
リセットリミット	UcaCtrlResetLimitOprt	リセットリミット機能を実行します。	使用	(*)1)
	UcaCtrlResetLimitOprt_p		使用	(*)1)
不感帯動作	UcaCtrlDeadband UcaCtrlDeadband_p	不感帯動作を実行します。		
制御演算初期化	UcaCtrlInitCheck	制御演算の初期化要求が設定されているか検査します。		
	UcaCtrlInitClear	制御演算の初期化要求を解除します。		
	UcaCtrlFuncInit	制御演算関数が保持する内部パラメータを初期化します。		
	UcaCtrlPidInit	UcaCtrlPid と制御演算関数が保持する内部パラメータを初期化します。		

*1 : UcaCtrlPidTiming が管理する制御周期カウンタを利用します。

補足

時間の差分を使用しない制御演算関数のみを使用し、かつユーザ記述のプログラムでも時間の差分を必要としない場合、ワンショット起動に関する特別な制限はありません。

時間の差分を使用する制御演算関数を使用している CSTM-C をワンショット起動する場合には、次の制限があります。

- CSTM-C のビルダ定義項目「処理タイミング」に「ワンショットのみ」を指定してください。または、機能ブロック定周期処理 (UcaBlockPeriodical) には処理を記述しないでください。つまり、機能ブロック定周期処理と機能ブロックワンショット起動処理の両方が起動されないようにしてください。
- UcaCtrlPidTiming を使用する場合、制御周期カウンタの分解能は基本周期です。たとえば基本周期が 4 秒であれば、制御周期カウンタは 4 の整数倍の値を取ります。前回の処理タイミングから今回の処理タイミングの時間差は、基本周期の整数倍になります。たとえば基本周期が 4 秒であると、10 秒や 30 秒などの 4 (基本周期) で割り切れない時間差を作り出すことはできません。

● UcaCtrlPidTimingを使用するプログラムをワンショット起動する

制御周期に従い動作するように作成したプログラムを機能ブロックワンショット起動処理から呼び出せば、CSTM-C は正常動作します。つまり、UcaCtrlHandler を (UcaBlockPeriodical からではなく) UcaBlockOneshot から呼び出します。この場合、機能ブロックワンショット起動処理 UcaBlockOneshot が呼び出されてからリターンするまでの間に一回以上 UcaCtrlPidTiming を呼び出してください。

ワンショット起動する場合は、次の制限事項があります。

- UcaCtrlPidTiming が管理する制御周期カウンタの分解能は基本周期です。
- UcaCtrlResetLimitOpt (UCAOPT_CTRL_TSCTIMING オプションを指定しない場合) が、内部で使用する制御時間の分解能は基本周期です。なお、UCAOPT_CTRL_TSCTIMING は制御の周期に実効スキャン周期を使用するというオプションなので指定できません。
- 前回の制御周期から今回の制御周期までの時間の差分 (ΔT) を取得するには、UcaCtrlPidTiming を呼び出した後に、(同じ処理タイミング内で) UcaCtrlPidGetTime を呼び出して、制御カウンタ値を取得します。UcaCtrlPidGetTime が引数に返す制御カウンタ値は、前回の制御周期からの時間の差分 (単位は秒) です。ただし、時間の差分は基本周期の整数倍になります。たとえば、基本周期が 4 秒の APCS で、高速周期 (1 秒周期) で動作するシーケンステーブルから (高速周期で動作するタイマブロックを使い 30 秒間隔を作り出し)、30 秒周期で CSTM-C をワンショット起動した場合には、UcaCtrlPidGetTiming で取得できる制御カウンタ値は 28 秒と 32 秒を交互に繰り返します。
- UcaCtrlPid が内部で管理している時間の差分の分解能も基本周期となります。つまり、内部の時間の差分は基本周期の整数倍となります。

● UcaCtrlPidTimingを使用しないプログラムのワンショット起動

前回の処理タイミングと今回の処理タイミングの時間差 (ΔT) が必要となる制御演算を行っていて、UcaCtrlPidTiming は使用しないプログラムをワンショット起動する場合、プログラムの難易度が高くなります。前回処理タイミングと今回処理タイミングの間の時間差を作り出す専用の機能がないので、ユーザアプリケーションで以下のような工夫が必要となるからです。

- ・ ワンショット起動をする周期が決まっている場合、CSTM-C のデータアイテム I01 ~ I08 に周期を設定し、ユーザプログラムで周期を取り出すことは可能です。
- ・ ユーザカスタムアルゴリズム作成用ライブラリのローカルタイマ機能を使用すれば、前回処理タイミングと今回の時間差を作ることは可能です。
- ・ 標準のタイマブロックを高速周期指定で使用し、ユーザカスタムアルゴリズムでタイマブロックのタイマ経過時間 PV を取得すれば、前回処理タイミングと今回の時間差を作ることが可能です(高速周期は 1 秒周期なので、時間差の分解能は秒になります)。シーケンステーブルでユーザカスタムブロックをワンショット起動し、ユーザカスタムアルゴリズムはタイマブロックのタイマ経過時間 PV を取得します。ユーザカスタムブロックをワンショット起動したあとに、シーケンステーブルよりタイマブロックをタイマ起動動作(タイマ経過時間 PV を 0 にリセットします)しておけば、ユーザカスタムアルゴリズムで取得する PV が前回処理からの時間差 (ΔT) となります。

前回の処理タイミングと今回の処理タイミングの時間差 (ΔT) が必要となる制御演算を行うプログラムを(定周期で起動しないで)ワンショット起動するようなユーザアプリケーションは、推奨しません(間欠制御を使用することを推奨します)。

6.4.3 UcaCtrlPidInitによる制御初期化

自動運転をしないブロックモード（たとえばMAN）から自動運転を行うブロックモード（たとえばAUT）にブロックモードが遷移した場合、最初の制御演算の前に制御の初期化をする必要があります。UcaCtrlPid関数を使用する（PID制御演算をする）場合の制御初期化用関数としてUcaCtrlPidInitが用意されています。

UcaCtrlHanlder（システム）は、制御の初期化が必要となる条件が成立すると「制御初期化の要求」を設定します。ユーザプログラムは、UcaCtrlInitCheck関数で初期化要求が設定されているか検査し、初期化が必要であればUcaCtrlPidInitで初期化を実行し、制御初期化の要求をUcaCtrlInitClearで解除します。この処理をしているのは次の部分です。

```
.....
/* 制御初期化 */
rtnCode = UcaCtrlInitCheck(bc, &isCtrlInit);
if (isCtrlInit) {
    rtnCode = UcaCtrlPidInit(bc, cv, NOOPTION);
    rtnCode = UcaCtrlInitClear(bc);
}
.....
```

UcaCtrlInitCheckは、引数isCtrlInitに制御初期化の要求が設定されているか否かを返します。制御初期化の要求が設定されていれば（isCtrlInit変数がTRUE、0以外であれば）、ユーザプログラムで制御初期化処理を実行します。制御初期化処理のポイントを示します。

- ・ UcaCtrlInitCheckで制御初期化が必要か判定します。
- ・ UcaCtrlPidでPID制御演算をする場合は、UcaCtrlPidInitを呼び出して制御初期化を行います。
- ・ 制御初期化を実行したらUcaCtrlInitClearを呼び出し、制御初期化の要求を解除しておきます。

補足 UcaCtrlPidInitはUcaCtrlFuncInitが行う制御演算関数のための初期化に加え、UcaCtrlPidのための初期化をします。つまり、UcaCtrlPidInitはUcaCtrlFuncInitの処理を包含しています。

参照 UcaCtrlFuncInitの詳細については、以下を参照してください。
[「6.3.3 制御初期化」](#)

6.4.4 PID制御演算

PID制御演算は、PID制御アルゴリズムを使用して操作出力変更量を算出します。PID制御動作は、比例（P）、積分（I）、微分（D）の3種類の動作が組み合わされた制御動作です。

サンプルプログラムでPID制御演算を行うのは、UcaCtrlPid関数を呼び出している次の部分です。

```
.....
/* 制御演算 */
rtnCode = UcaCtrlPid(bc, cv, &calc, NOOPTION);

/* レンジ変換 */
rtnCode = UcaDataConvertRange(bc, calc, &deltaMv, UCAOPT_RANGE_DIFFVALUE);
.....
```

UcaCtrlPidは、引数に指定されたcv（入力補償後のPV）を元に操作出力変更量calcを算出します。UcaCtrlPidにより算出されたcalcを、UcaDataConvertRangeにより測定値PVのレンジから出力値MVのレンジにレンジ変換し、操作出力変更量deltaMvを決定します。

■ PID制御演算アルゴリズム

UcaCtrlPid は、5 つの PID 制御アルゴリズムをサポートしています。どの PID 制御アルゴリズムを使用するかは、連続制御形ユーザカスタムブロックのビルダ定義項目「PID 制御アルゴリズム」で指定します。デフォルトは、「自動決定 2」です。

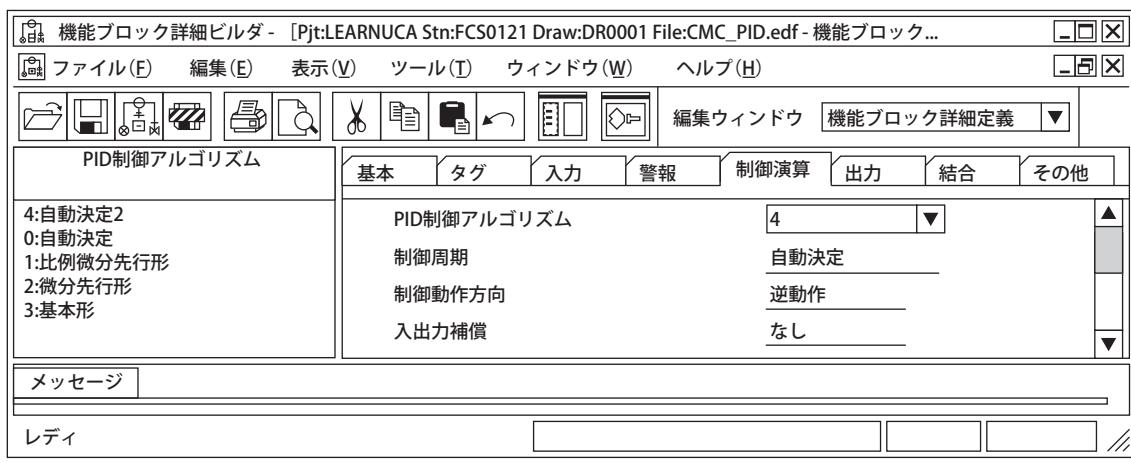


図 PID制御演算アルゴリズムの指定

UcaCtrlPid 関数は、その処理の中でビルダ項目「PID 制御アルゴリズム」の指定を取り出し、指定された PID 制御アルゴリズムにより操作出力変更量を算出します。

5 つの PID 制御アルゴリズムを以下に示します。

表 PID制御アルゴリズム一覧

PID制御アルゴリズム	補足
4: 自動決定 2	AUT モード、RCAS モード時は、I-PD と同じ CAS モード時は、PI-D と同じ
0: 自動決定	AUT モード時は、I-PD と同じ CAS モード、RCAS モード時は、PI-D と同じ
1: 比例微分先行形 (I-PD)	
2: 微分先行形 (PI-D)	
3: 基本形 (PID)	

参照 5 つの PID 制御アルゴリズムの詳細については、以下を参照してください。

機能ブロッククリアレンス Vol.1 (IM 33J15A30-01JA) 「1.5 PID 調節ブロック (PID)」

■ 非線形ゲイン

PID 制御演算を行う関数 UcaCtrlPid は、非線形ゲイン処理を行います。非線形ゲインは、制御演算を行うときに、比例ゲインを測定値 PV と設定値 SV の偏差の大きさに対応づけて変化させる機能です。非線形ゲインの動作指定は、連続制御形ユーザカスタムブロックのビルダ定義項目「非線形ゲイン」で行います。UcaCtrlPid はビルダ定義項目「非線形ゲイン」の指定値を内部的に取得し、指定に従って非線形ゲイン処理を実行します。非線形ゲインのデフォルトは「なし」です。



060422J.ai

図 非線形ゲインの指定

参照

非線形ゲインの動作の詳細については、以下を参照してください。

[機能ブロッククリアレンス Vol.1 \(IM 33J15A30-01JA\) 「1.4 調節ブロックに共通の制御演算処理」の「■ 非線形ゲイン」](#)

6.5 データアイテムPVとデータアイテムSVを使用しない場合

UcaCtrlHandlerは、測定値PVと設定値SVに関係する次の処理を行います。

表 UcaCtrlHandlerのPVとSVに関係する処理

処理	説明
測定値トラッキング (メジャートラッキング)	手動(MAN)モード時などに、設定値SVを測定値PVに一致させる機能です。
設定値プッシュバック	設定値 SV、カスケード設定値 CSV、およびリモート設定値 RSV の 3 つの設定値を一致させる機能です。
偏差アラームチェック	測定値 PV と設定値 SV の偏差 ($DV = PV - SV$) の絶対値が偏差アラーム設定値 DL の絶対値を越えているか検査する機能です。

参照 UcaCtrlHandler の処理の詳細については、以下を参照してください。

「6.2.4 ブロックモード遷移と設定値の作成 (UcaCtrlHandler の内部で処理)」

ユーザカスタムアルゴリズムのユーザ定義の入力処理で、測定値 PV を設定しない (UcaRWSetPv を呼び出さない) ような場合など、データアイテム PV とデータアイテム SV の扱いが標準の調節ブロックと異なるときに、上表の処理をしないようにするための対処について説明します。対処方法には、次の 2 つがあります。

- ・ビルダ定義項目などを利用する方法
- ・UcaCtrlHandler にオプション UCAOPT_CTRL_NOSVOPRT を指定する方法

■ ビルダ定義項目などを利用

測定値トラッキング（メジャートラッキング）、設定値プッシュバック、偏差アラームチェックのビルダ定義項目を、それぞれの処理が動作しないように指定することができます。

● 測定値トラッキング（メジャートラッキング）

連続制御形ユーザカスタムブロックのビルダ定義項目「メジャートラッキング」で、メジャートラッキングをしないように指定を変更してください。



図 メジャートラッキングの指定

- MAN 時のデフォルトは「なし」です。
- AUT かつ CND 時のデフォルトは「なし」です。
- CAS かつ CND 時のデフォルトは「あり」ですので、「なし」に変更します。

UcaCtrlHandler はブロックモードが PRD から AUT、CAS または RCAS に切り換わったときに、メジャートラッキングを行います。ビルダ定義項目には関係なく、常に行われます。

● 設定値プッシュバック

設定値プッシュバックは、設定値 SV、カスケード設定値 CSV、およびリモート設定値 RSV の 3 つの設定値を一致させる機能です。UcaCtrlHandler は、ブロックモードに従い等値化の方法を決定します。

参照 等値化の方法の詳細については、以下を参照してください。

「[6.2.4 ブロックモード遷移と設定値の作成 \(UcaCtrlHandler の内部で処理\)](#)」

その等値化の方法で都合が悪い場合には、ユーザ定義の制御演算関数内で（等値化の元になるデータを UcaDataStoreSv、UcaDataStoreCsv、または UcaDataStoreRsv で保存したあと）UcaDataSvPushback を使って、3 つの設定値の等値化をやり直してください。

UcaDataSvPushback には、オプション指定により等値化の方法を指定することができます。また、UcaCtrlHandler の設定値プッシュバックのあとで、UcaDataSvPushback により等値化のやり直しをしても何も問題はありません（ただし、UcaCtrlHandler により偏差アラームチェックは、UcaCtrlHandler による設定値プッシュバックで決定した SV に対して行われます）。

● 偏差アラームチェック

連続制御形ユーザカスタムブロックのビルダ定義項目「偏差アラーム」で、偏差アラームチェックをしないように指定を変更してください。



図 偏差アラームの指定

偏差アラームのデフォルトは「BOTH：両方向の検出」ですので、「NO：アラームなし」に変更します。

■ UcaCtrlHandlerにUCAOPT_CTRL_NOSVOPRTオプションを指定

UcaCtrlHandler に UCAOPT_CTRL_NOSVOPRT オプションを指定すると、UcaCtrlHandler は測定値トラッキング（メジャートラッキング）、設定値プッシュバック、偏差アラームチェックを行いません。UCAOPT_CTRL_NOSVOPRT オプション指定時の動作を以下に示します。

参照 各処理の詳細については、以下を参照してください。

「[6.2.4 ブロックモード遷移と設定値の作成（UcaCtrlHandler の内部で処理）](#)」

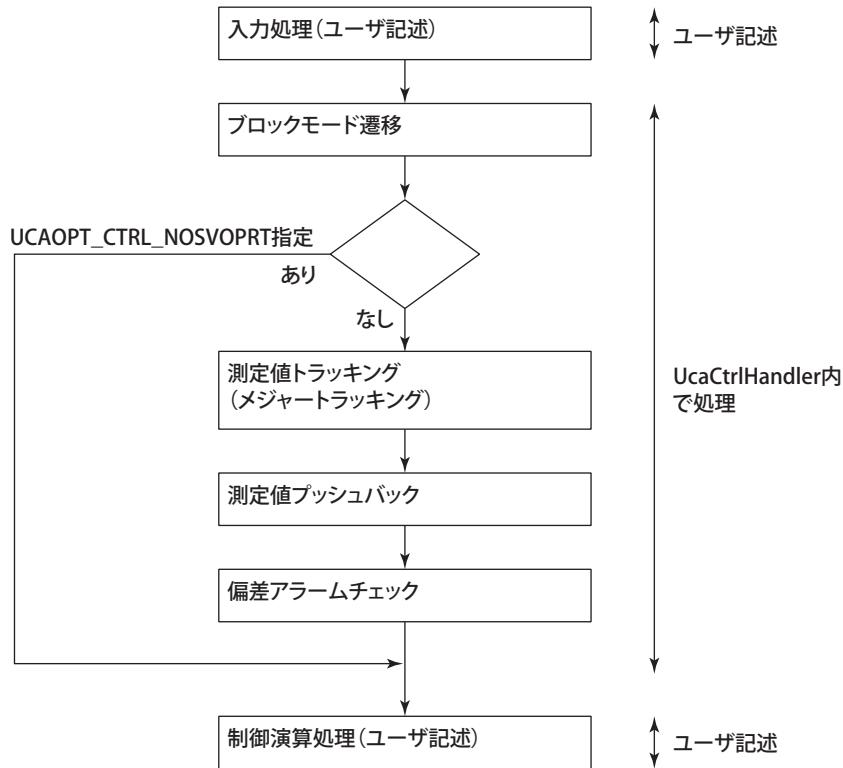


図 UcaCtrlHandlerの処理（部分）

060504J.ai

UCAOPT_CTRL_NOSVOPRT オプションを指定し、かつ測定値トラッキング（メジャートラッキング）、設定値プッシュバック、偏差アラームチェックのいずれかの処理を必要とする場合には、ユーザ記述の制御演算処理で以下の関数を呼び出してください。

表 UCAOPT_CTRL_NOSVOPRTで省略した処理をする関数

処理	関数	注意事項	呼び出し順
測定値トラッキング (メジャートラッキング)	UcaDataMeasureTracking	入力処理で UcaRWSetPv によりデータ項目 PV にデータを設定してください。	先
設定値プッシュバック	UcaDataSvPushback	データアイテム SV, RSV, CSV にアクセスします。	
偏差アラームチェック	UcaDataCheckDv	データアイテム PV と SV にアクセスします。	後

● 留意事項

- 上表の処理を制御演算関数で呼び出す場合には、表の上の関数から下の関数の順に呼び出してください。たとえば、メジャートラッキングと設定値プッシュバックをする場合、UcaDataMeasureTraking を先に呼び出し、UcaDataSvPushback を後にしてください。
- UCAOPT_CTRL_NOSVOPRT を指定し、かつ測定値トラッキング（メジャートラッキング）、設定値プッシュバック、偏差アラームチェックのどれかの処理をする場合には、UcaCtrlHandler の演算処理実行モードに UCACTRL_ALLMODE を指定してください。ユーザ記述の制御演算関数が任意のブロックモードで呼び出されますので、プログラムに設定値プッシュバックをする UcaDataSvPushback 呼び出しを記述すれば、任意のブロックモードで制御演算関数に記述した処理が行われます。この場合、必要に応じて、制御演算処理で UcaModeGet に UCAOPT_MODEGET_DOMINANT を指定してブロックモードを取得し、ブロックモードごとに処理を分けてください。
- 上記 2 点からもわかるように、UCA_CTRL_NOSVOPRT を使いこなすのは、少々難易度が高くなります。UCA_CTRL_NOSVOPRT はここでの説明を理解した上でご使用ください。

6.6 カスケード結合と実効スキャン周期

カスケード結合では、上流側の機能ブロックの出力端子（OUT）から、下流側の機能ブロックの設定入力端子（SET）を介して、最終的に下流側の機能ブロックのカスケード設定値（CSV）としてデータが設定されます。下流側の機能ブロックのブロックモードがカスケード（CAS）になるとカスケードクローズとなりカスケード制御が行われます。下流側の機能ブロックのブロックモードが非カスケードモード（CAS以外）であれば、カスケードオープンとなります。カスケードオープンの場合は、上流側機能ブロックの基本ブロックモードIMAN（初期化手動）が成立します。

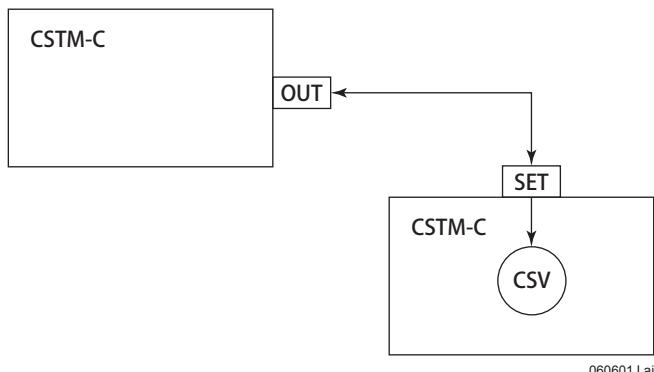


図 CSTM-CとCSTM-Cのカスケード結合

この節では、下流側の機能ブロックのブロックモードが非カスケードモードからカスケードに変化したときに、カスケードオープンからカスケードクローズとなるまでの動作について説明します。また、上流側機能ブロックのIMANの解除は実効スキャン周期に依存しますので、制御周期が長いユーザカスタムブロックであっても実効スキャン周期の時間を短くしておくと、IMANの解除に必要な時間が短くなることを説明します。

● データステータスCNDと基本ブロックモードIMAN

カスケード結合の動作に関するデータステータス CND（コンディショナル）と、基本ブロックモード IMAN（初期化手動）について説明します。CND はカスケードオープンであることを示すデータステータスです。下流側の機能ブロックが非カスケードモードになった、またはカスケード結合の経路がスイッチなどにより接続切れとなったとき、(下流側機能ブロックの) データアイテム CSV のデータステータスが CND になります。CND はカスケード設定値 CSV のようなカスケード結合の対象となるデータアイテムのみで成立します。

上流側の機能ブロックの出力端子（OUT）から下流側の機能ブロックの設定入力端子（SET）に出力する場合、上流側の機能ブロックは出力の前に下流側の SET 端子に対応するデータアイテム（CSV）のデータステータスを検査します。下流側機能ブロックのデータアイテム CSV のデータステータス CND が成立していると、上流側機能ブロックは「カスケードオープン」であることがわかりますので、SET 端子へのデータ出力をしません。このとき、上流側機能ブロックは自身のブロックモードの IMAN を成立させます。

● 実効スキャン周期と制御周期が同じ場合の動作

3つの連続制御形ユーザカスタムブロック C1、C2、C3 が、以下のようにカスケード結合されている場合の動作を説明します。

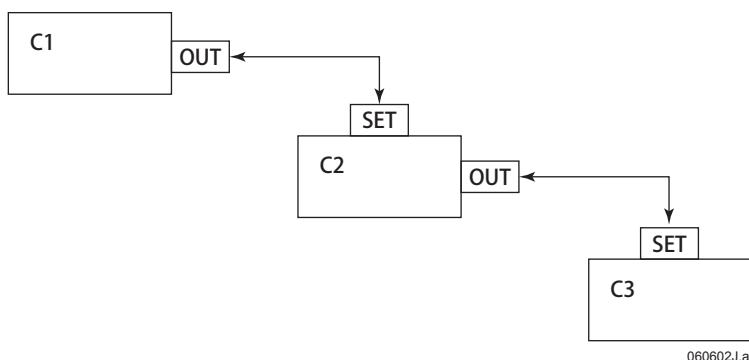


図 2段のカスケード結合

C1、C2、C3 の実効スキャン周期と制御周期は、すべて 4 秒とします。これは次の 2 つのどちらかの場合です。

- ・ ユーザプログラムで UcaCtrlPidTiming を呼び出して実効スキャン周期と制御周期の両方を有効にし、実効スキャン周期と制御周期にそれぞれ 4 秒を指定した場合
- ・ UcaCtrlPidTiming を呼び出さない（ビルダ定義項目「制御周期」の指定は無視）で実効スキャン周期のみ有効とし、実効スキャン周期（4 秒を指定）のたびに制御演算を行う場合

機能ブロックの実行順は、C1、C2、C3 の順です。2 つのカスケード結合はカスケードオープンとします。最初は C1 のブロックモードは IMAN (AUT)、C2 のブロックモードは IMAN (CAS)、C3 のブロックモードは AUT とします。この状態で C3 のブロックモードを AUT から CAS に変更すると、まず C2 と C3 がカスケードクローズとなり、その後 C1 と C2 がカスケードクローズとなります。大まかには、1 段のカスケード結合が成立するのに、2 実効スキャン周期が必要です。この動作の詳細を次表に示します。

表 カスケードオープンからカスケードクローズになるまで（その1）

実効スキャン 周期回数と 経過時間 (秒)	処理タイミング			ブロックモード			CSVのデータ ステータス (*2)	
	C1	C2	C3	C1	C2	C3	C2	C3
初期状態				IMAN (AUT)	IMAN (CAS)	AUT	CND	CND
C3 を CAS に				IMAN (AUT)	IMAN (CAS)	AUT → CAS	CND	CND
1回 (4秒)	○			IMAN (AUT)	IMAN (CAS)	CAS	CND	CND
		○		IMAN (AUT)	IMAN (CAS)	CAS	CND	CND
			●	IMAN (AUT)	IMAN (CAS)	CAS	CND	
2回 (8秒)	○			IMAN (AUT)	IMAN (CAS)	CAS	CND	
		○		IMAN (AUT)	CAS (*1)	CAS	CND	
			●	IMAN (AUT)	CAS	CAS	CND	
3回 (12秒)	○			IMAN (AUT)	CAS	CAS	CND	
		●		IMAN (AUT)	CAS	CAS		
			●	IMAN (AUT)	CAS	CAS		
4回 (16秒)	○			AUT (*1)	CAS	CAS		
		●		AUT	CAS	CAS		
			●	AUT	CAS	CAS		
5回 (20秒)	●			AUT	CAS	CAS		
		●		AUT	CAS	CAS		
			●	AUT	CAS	CAS		

処理タイミング：

○：処理タイミングが与えられますが制御演算はしません。

●：処理タイミングが与えられて制御演算も実行します。

*1： IMAN が消える処理タイミングでは、制御演算の初期化を行います。

*2： C1 のブロックステータスは AUT の (CAS ではない) ため、C1 のデータアイテム CSV のデータステータス CND は成立しています。

C3 のブロックモードを AUT から CAS に変更した後の個々の動作を表の上から順に説明します。1回目の実効スキャン周期の C1 の処理タイミングを「1-C1」のように記述します。

1-C1： C1 は C2 の CSV のデータステータス CND が成立しているので、IMAN が成立したまま変化しません。

1-C2： C2 は C3 の CSV のデータステータス CND が成立しているので、IMAN が成立したまま変化しません。

1-C3： C3 は自身のブロックモードが CAS なので、制御演算を実行します。また、自身のブロックモードが CAS になったので、データアイテム CSV のデータステータス CND を解除します。

2-C1： C1 は C2 の CSV のデータステータス CND が成立しているので、IMAN が成立したまま変化しません。

2-C2： C2 は C3 の CSV のデータステータス CND が解除されているので、IMAN (CAS) から CAS に変化します。制御演算の初期化が行われます。CSV のデータステータス CND は成立したままです。

2-C3： C3 は CAS のまま変化しません。

- 3-C1 : C1 は C2 の CSV のデータステータス CND が成立しているので、IMAN が成立したまま変化しません。
- 3-C2 : C2 は自身のブロックモードが CAS なので、制御演算を開始します。また、CSV のデータステータス CND を解除します。
- 3-C3 : C3 は CAS のまま変化しません。
- 4-C1 : C1 は C2 の CSV のデータステータス CND が解除されているので、IMAN (AUT) から AUT に変化します。制御演算の初期化が行われます。CSV のデータステータス CND は成立したままです。
- 4-C2 : C2 は CAS のまま変化しません。
- 4-C3 : C3 は CAS のまま変化しません。
- 5-C1 : C1 は自身のブロックモードが AUT なので、制御演算を開始します。CSV のデータステータス CND は成立したままです。
- 5-C2 : C2 は CAS のまま変化しません。
- 5-C3 : C3 は CAS のまま変化しません。

C3 のブロックモードを AUT から CAS に変更したあとに、2段のカスケード結合が両方ともカスケードクローズになり、C1、C2、C3 の3つとも制御演算を開始するまでに5実効スキャン周期が必要です。また、C1 のブロックモード IMAN が解除されるまでには、4実効スキャン周期かかります。

● 実効スキャン周期と制御周期が違う場合の動作

UcaCtrlPidTiming を使ってビルダ定義項目「制御周期」の指定を有効にし、実効スキャン周期に4秒、制御周期に16秒を指定した場合の動作を説明します。説明を簡単にするため、C1、C2、C3 の3つとも同じ時間指定とし、スキャン位相の指定も同じ（ゼロ）とします。機能ブロックの実行順は、C1、C2、C3 の順です。最初はカスケードオープンとします。最初は C1 のブロックモードは IMAN (AUT)、C2 のブロックモードは IMAN (CAS)、C3 のブロックモードは AUT とします。この状態で以下のタイミングで C3 のブロックモードを AUT から CAS に変更した場合の動作を、次表に示します。以下の 1、2、3 の数字は、表の実効スキャン周期回数と一致します。

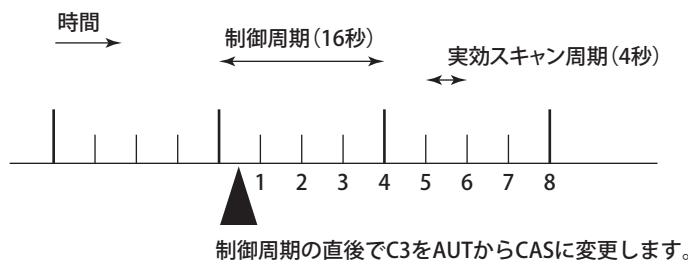


図 実効スキャン周期と制御周期

表 カスケードオープンからカスケードクローズになるまで（その2）

実効スキャン 周期回数 と経過時間 (秒)	処理タイミング			ブロックモード			CSVのデータ ステータス (*2)	
	C1	C2	C3	C1	C2	C3	C2	C3
初期状態				IMAN (AUT)	IMAN (CAS)	AUT	CND	CND
C3 を CAS に				IMAN (AUT)	IMAN (CAS)	AUT → CAS	CND	CND
1回 (4秒)	○			IMAN (AUT)	IMAN (CAS)	CAS	CND	CND
		○		IMAN (AUT)	IMAN (CAS)	CAS	CND	CND
			○	IMAN (AUT)	IMAN (CAS)	CAS	CND	
2回 (8秒)	○			IMAN (AUT)	IMAN (CAS)	CAS	CND	
		○		IMAN (AUT)	CAS	CAS	CND	
			○	IMAN (AUT)	CAS	CAS	CND	
3回 (12秒)	○			IMAN (AUT)	CAS	CAS	CND	
		○		IMAN (AUT)	CAS	CAS		
			○	IMAN (AUT)	CAS	CAS		
4回★ (16秒)	◎			AUT (*1)	CAS	CAS		
		◎		AUT	CAS (*1)	CAS		
			●	AUT	CAS	CAS		
5～7回 (20～28秒)	○			AUT	CAS	CAS		
		○		AUT	CAS	CAS		
			○	AUT	CAS	CAS		
8回★ (32秒)	●			AUT	CAS	CAS		
		●		AUT	CAS	CAS		
			●	AUT	CAS	CAS		

実効スキャン周期回数：

数字の右の★印は、制御周期を表します。

処理タイミング：

○： 実効スキャン周期の処理タイミングです。

◎： 制御周期の処理タイミングですが制御演算はしません。

●：制御周期の処理タイミングで制御演算を実行します。

*1： IMAN が消えたあの最初の制御周期では、制御演算の初期化を行います。

*2： C1 のブロックステータスは AUT の (CAS ではない) ため,C1 のデータアイテム CSV のデータステータス CND は成立しています。

C3 のブロックモードを AUT から CAS に変更したあの個々の動作を表の上から順に説明します。1回目の実効スキャン周期の C1 の処理タイミングを「1-C1」のように記述します。

1-C1：C1 は C2 の CSV のデータステータス CND が成立しているので、IMAN が成立したまま変化しません。

1-C2：C2 は C3 の CSV のデータステータス CND が成立しているので、IMAN が成立したまま変化しません。

1-C3：C3 は自身のブロックモードが CAS になったので、CSV のデータステータス CND を解除します。

2-C1：C1 は C2 の CSV のデータステータス CND が成立しているので、IMAN が成立したまま変化しません。

- 2-C2 : C2 は C3 の CSV のデータステータス CND が解除されているので、IMAN (CAS) から CAS に変化します。CSV のデータステータス CND は成立したままです。
- 2-C3 : C3 は CAS のまま変化しません。
- 3-C1 : C1 は C2 の CSV のデータステータス CND が成立しているので、IMAN が成立したまま変化しません。
- 3-C2 : C2 は自身のブロックモードが CAS になったので、CSV のデータステータス CND を解除します。
- 3-C3 : C3 は CAS のまま変化しません。
- 4-C1 : C1 は C2 の CSV のデータステータス CND が解除されているので、IMAN (AUT) から AUT に変化します。ブロックモードが AUT になった制御周期なので、制御演算の初期化をします。ブロックモードが AUT なので、CSV のデータステータス CND は成立したままです。
- 4-C2 : C2 はブロックモードが IMAN (CAS) から CAS になった後の最初の制御周期なので、制御演算の初期化をします。
- 4-C3 : C3 はブロックモードが CAS のままなので、制御演算を実行します。
- 5-C1 : C1 は AUT のまま変化しません。
- 5-C2 : C2 は CAS のまま変化しません。
- 5-C3 : C3 は CAS のまま変化しません。

6回目と7回目の実効スキャン周期は5回目と同じです。

- 8-C1 : C1 はブロックモードが AUT になった後の2回目の制御周期なので、制御演算の実行を開始します。
- 8-C2 : C2 はブロックモードが CAS になった後の2回目の制御周期なので、制御演算の実行を開始します。
- 8-C3 : C3 はブロックモードが CAS のままなので、制御演算を実行します。

一つのカスケード結合の下流側機能ブロックのデータステータス CND が解除されてから上流側機能ブロックのブロックモード IMAN が解除されるのに、実効スキャン周期が2回必要なことがわかります。また、IMAN が解除された直後の（実効スキャン周期と制御周期が重なっていれば、IMAN が解除された）制御周期に制御演算の初期化が行われ、その次の制御周期で機能ブロックは制御演算を開始します。

● 標準ブロックと連続制御形ユーザカスタムブロック

これまでの説明は、連続制御形ユーザカスタムブロックと標準の調節ブロックに適応できます。ただし、標準の機能ブロックは実効スキャン周期の指定はありませんので、基本周期が対応します。以下にカスケード結合下流側で機能ブロックのデータアイテム CSV のデータステータス CND が解除されてから、上流側の機能ブロックで IMAN が解除されるまでと、制御演算の実行が開始されるまでを整理しておきます。

表 IMAN、CNDの解除と制御演算の開始

	連続制御形ユーザカスタムブロック (CSTM-C)	標準の調節ブロック (PIDなど)
IMAN の解除	実効スキャン周期を 1 回	基本周期を 1 回
CND の解除	実効スキャン周期を 2 回	基本周期を 2 回
制御演算の開始	制御周期を 2 回	制御周期を 2 回

*：すべてカスケード結合の下流側の機能ブロックで、データステータス CND が解除された後の処理回数です。

CSTM-C のブロックモード IMAN およびデータステータス CND が解除されるまでは、実効スキャン周期の回数で決まります。したがって、制御周期を有効にして実効スキャン周期は短い時間を指定し、制御周期には実際に制御をしたい（長い）周期を指定すれば、IMAN が解除されるまでの時間を短くすることができます。

■ 制御周期を有効にする

実効スキャン周期ごとに制御演算を実行し、制御周期の指定は無視するようなユーザカスタムアルゴリズムを制御周期の指定に従い動作するように直してみます。実効スキャン周期を小さくすれば、IMAN が解除されるまでの時間が短くなります。

ここでは、6.3 節の PI 制御演算を行うユーザカスタムアルゴリズム (_SMPL_CONTROL) を改造します。6.3 節のプログラムは、実効スキャン周期に毎回制御演算を行うようにできています。このプログラムはビルダ定義項目「制御周期」の指定は無視しています。_SMPL_CONTROL を、制御周期を使用するように改造したユーザカスタムアルゴリズム _SMPL_CONTROL2 を見ていきます。

参照 制御周期の使用・不使用によるプログラムの違いの詳細については、以下を参照してください。
[「6.4.2 実効スキャン周期と制御周期」](#)

_SMPL_CONTROL2 のソースファイル control2.c の機能ブロック初期化処理 UcaBlockPeriodical は、UcaCtrlHandler の演算実行ブロックモードに UCACTRL_AUTCASCAS を指定しています (_SMPL_CONTROL も同じです)。この指定により、UcaCtrlHandler は、ブロックモード AUT、CAS、RCAS の場合のみユーザ定義の制御演算関数 control を呼び出します。

```
UCAUSER_API I32 UCAAPI UcaBlockPeriodical(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnCode; /* リターンコード */

    rtnCode = UcaCtrlHandler( bc,
        input,
        control,
        output,
        UCACTRL_AUTCASCAS, ← 元々AUT、CAS、RCAS時のみ
        NOOPTION );           ユーザ定義関数controlを呼び
                               出すように指定

    return SUCCEED;
}
```

制御周期を使用する場合は、制御周期のカウンタ処理を行う UcaCtrlPidTiming 関数を、実効スキャン周期のたびに 1 度以上呼び出す必要があります。ユーザ定義の制御演算関数は、ブロックモードが AUT、CAS、RCAS 以外だと呼び出されません。そこで実効スキャン周期のたびに必ず呼び出されるユーザ定義の入力処理 input の最後で、UcaCtrlPidTiming を呼び出しておきます (UcaCtrlPidTiming の呼び出しは、SMPL_CONTROL2 での追加です)。

参照 UcaCtrlPidTiming の詳細については、以下を参照してください。
[「6.4.2 実効スキャン周期と制御周期」の「■ UcaCtrlPidTiming を使用するポイント」](#)

```

I32 input(
    UcaBlockContext bc /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnCode;           /* リターンコード */
    I32 rtnReadIn;        /* IN端子入力リターンコード */
    BOOL isCtrlTime;      /* 制御周期かどうか */

    /* IN端子からRVにデータを読み込み */
    rtnReadIn = UcaRWReadIn(bc, NOOPTION);

    /* RVからPVにデータを設定 */
    rtnCode = UcaRWSetPv(bc, NULL, rtnReadIn, NOOPTION);

    /* 制御周期カウンタ処理を実行
     * 実効スキャン周期に1度呼び出す必要があるので
     * ここで呼び出させておきます。
     */
    rtnCode = UcaCtrlPidTiming(bc, &isCtrlTime, NOOPTION); ← UcaCtrlPidTimingを
                                                               呼び出す

    return SUCCEEDED;
}

```

ユーザ記述の制御演算関数 control には、UcaCtrlPidTiming を使い制御周期か否かを判定する処理を追加します。

```

.....
/* 制御周期かどうかを確認 */
rtnCode = UcaCtrlPidTiming(bc, &isCtrlTime, NOOPTION); ← UcaCtrlPidTimingを
                                                               呼び出す

/* 制御ホールド確認 */
rtnCode = UcaCtrlHoldCheck(bc, &isCtrlHold);

/* 制御演算処理 */
/*****************/
/* 制御周期の場合に制御演算を行います。          */
/* 制御ホールド時には、制御演算を行いません。      */
/*****************/
if (isCtrlTime && !isCtrlHold) { /* 制御演算状態(制御周期かつ制御ホールドでない場合) */
    /* データアイテムを取得 */
    rtnCode = UcaDataGetPv(bc, &pv, NOOPTION);          /* PV */

    rtnCode = UcaDataGetSv(bc, &sv, NOOPTION);          /* SV */
    rtnCode = UcaDataGetP(bc, &pb, NOOPTION);          /* P */
    rtnCode = UcaDataGetI(bc, &itime, NOOPTION);        /* I */
    rtnCode = UcaDataGetPn(bc, &p01, 1, 1, NOOPTION);   /* P01(前回CV値) */

    /* 前回制御周期からの経過時間(秒)を取得 */
    rtnCode = UcaCtrlPidGetTime(bc, &time); ← UcaCtrlPidGetTimeで前回制御
                                                周期からの経過時間を取得
.....

```

_SMPL_CONTROL2 は制御周期ごとに制御演算を実行するので、前回制御周期からの経過時間（単位は秒）を UcaCtrlPidGetTime で取得します。SMPL_CONTROL は実効スキャン周期ごとに制御演算を実行しているので、この部分で前回からの経過時間として UcaDataGetTsc によりデータアイテム TSC に保持されている実効スキャン周期（単位は秒）を取得しています。

_SMPL_CONTROL2 では、リセットリミットを行う UcaCtrlResetLimitOprt のオプションを NOOPTION にします（_SMPL_CONTROL は、制御時間に実効スキャン周期を使用する、UCAOPT_CTRL_TSCTSIME オプションを指定しています）。

```
/* 制御出力動作 */
rtnCode = UcaConfigOutput(bc, &configOut);
if (configOut == UCACONFIG_ACT_POS){      /* 位置形 */
    /* リセットリミット */
    rtnCode = UcaCtrlResetLimitOprt(bc, &deltaMv, NOOPTION); ← NOOPTIONを指定
....
```

ユーザ定義の出力処理では、UcaRWWriteMvToOutSub のオプションに UCAOPT_RW_CTRLTIMING を指定します（_SMPL_CONTROL は NOOPTION）。UCAOPT_RW_CTRLTIMING を指定すると自動運転中（CAS、AUT、RCAS 時）の場合は制御周期のときだけ出力処理を行います。

```
I32 output(
    UcaBlockContext bc           /* (IN/OUT): ブロックコンテキスト */
)
{
    I32 rtnCode;                /* リターンコード */

    /* OUT端子とSUB端子からデータを出力 */
    rtnCode = UcaRWWriteMvToOutSub(bc, UCAOPT_RW_CTRLTIMING); ← UCAOPT_RW_CTRLTIMINGを指定

    return SUCCEED;
}
```

6.7 カスケードクローズとワンショット起動

ここではカスケードオープンからカスケードクローズになったときに、ユーザカスタムブロックをワンショット起動することにより、ユーザカスタムブロックのIMANを消したり、制御演算の実行開始を早くする方法を解説します。要点は次の2点となります。

- ・カスケード結合の上流側がユーザカスタムブロックの場合に、カスケードオープンからカスケードクローズになったときに、上流側のユーザカスタムブロックを3回連続してワンショット起動します。
- ・このようなワンショット起動を使用する場合には、ワンショット起動時に出力値が急変しないようにするために、ユーザカスタムアルゴリズムを一定の規則に従い作成しておく必要があります。

■サンプルプログラム

カスケードクローズになったときに、ユーザカスタムブロックをワンショット起動することにより、IMAN消去と制御演算開始を実現します。この動作を実現しているサンプルプログラムが用意しておりますので、サンプルをもとに説明していきます。次の表に示すサンプルプログラムがあります。

表 サンプルプログラム

分類	説明	制御ドローイングのコメント(*1)	ユーザカスタムアルゴリズム(*2)	コントロール(8ループ)ウィンドウ(*3)
PIDと同じ動作をするユーザカスタムブロックを間欠制御で起動	ユーザカスタムアルゴリズム _SMPL_PID_B は、6.4節の _SMPL_PID を定周期でもワンショットでも呼び出し可能としたものです。 ユーザカスタムブロックは間欠制御で起動します	PID_B.txt PID タイプ間欠制御	_SMPL_PID_B	pid_b_cg.xaml
入力したデータのみから出力値を計算 (カスケードの最下流が他ステーション)	ユーザカスタムアルゴリズム _SMPL_MODE_B は、6.2節の _SMPL_MODE を定周期でもワンショットでも呼び出し可能としたものです。 カスケード結合の下流が他ステーションデータの場合に、MLD-SW ブロックを使って処理をする例です	MODE_B.txt (APCS 側) MODE_B_FCS.txt (FCS 側) MODE_B 実効スキャン周期 (ADL付き)	_SMPL_MODE_B	mode_b_cg.sva

*1： 制御ドローイングのサンプルは、下記のフォルダにありますので、制御ドローイングビルダよりインポートしてください。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥Drawings

*2： サンプルソリューションは、1章の準備作業により下記のフォルダにコピーされています。

<ドライブ名> ¥UcaWork¥UcaSamples

*3： コントロール(8ループ)ウィンドウのサンプルは、下記のフォルダにありますので、HISのウィンドウにインポートしてください。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥Graphics

■ PIDと同じ動作をするユーザカスタムブロックを間欠制御で起動

2段のカスケード結合のある制御ドローリングで、シーケンステーブルからユーザカスタムブロックをワンショット起動することにより、カスケードクローズ時のIMAN消去と制御開始を早くするプログラム例を説明します。下図に制御ドローリングを示します。

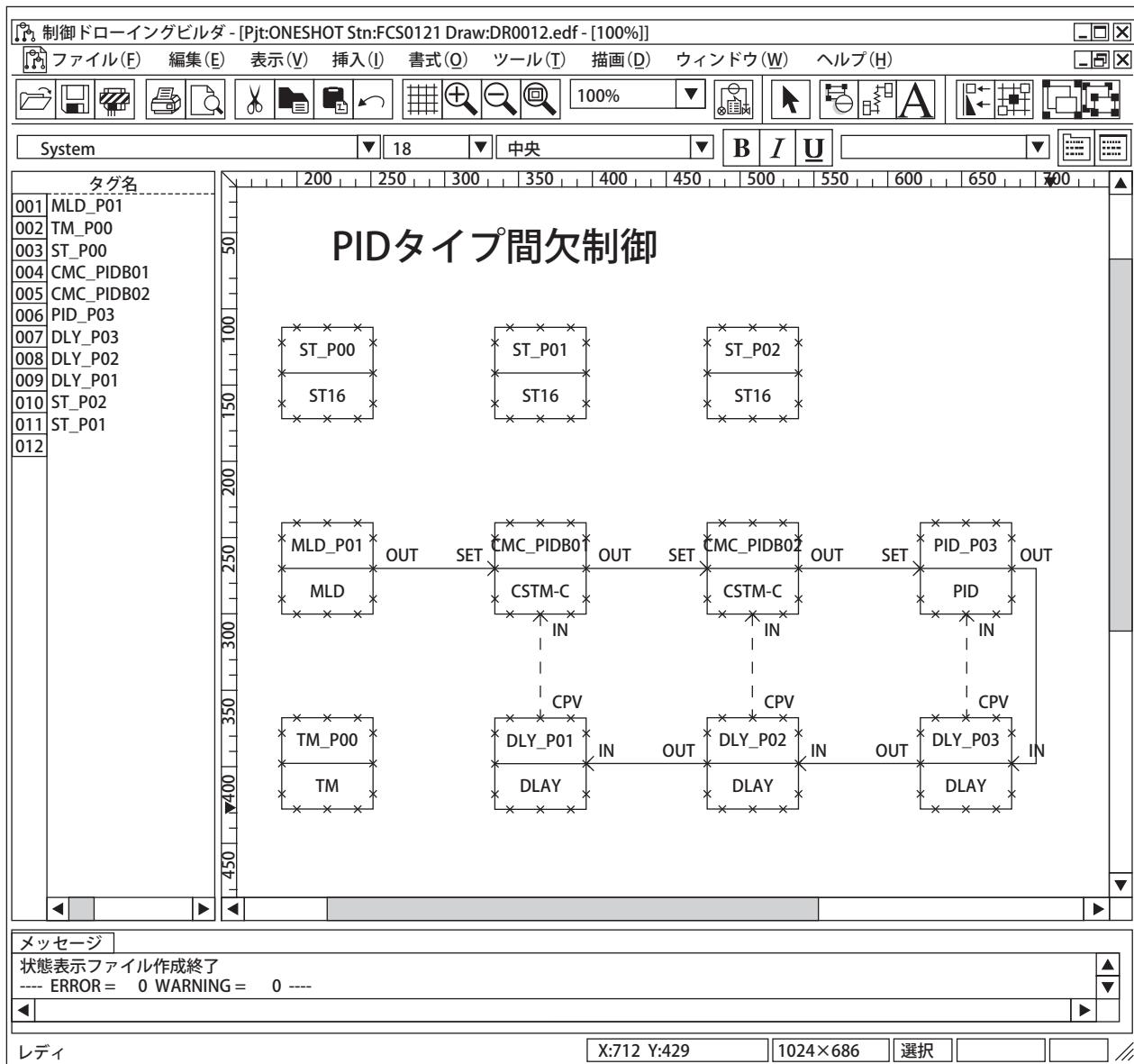


図 間欠制御用ドローイング

この制御ドローリングには、次の2つのカスケード結合があります。

- CMC_PIDB01 (CSTM-C) と CMC_PIDB02 (CSTM-C)
- CMC_PIDB02 (CSTM-C) と PID_P03 (PID)

ユーザカスタムブロック CMC_PIDB01 と CMC_PIDB02 には、実効スキャン周期に 4 秒、制御周期として間欠制御を指定しています。両方ともユーザカスタムアルゴリズム _SMPL_PID_B を使用します (PID 調節ブロックと同じ動作をするものを、一部改造したものです)。

CMC_PIDB01 と CMC_PIDB02 はシーケンステーブル ST_P00 より間欠制御として起動されます。次図はシーケンステーブル ST_P00 です。タイマブロック TM_P00 がカウントアップすると、CMC_PIDB01 と CMC_PIDB02 のデータアイテム CSW に 1 を設定しています。

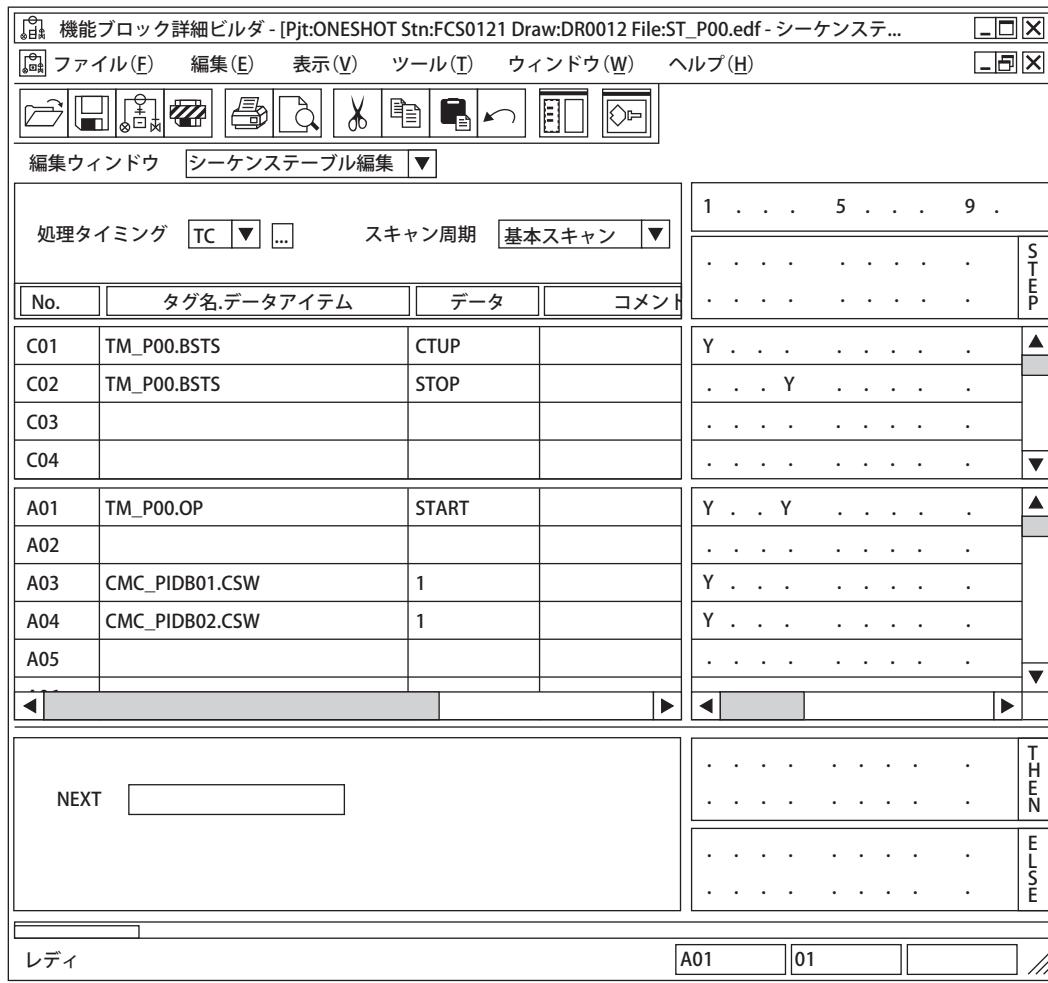
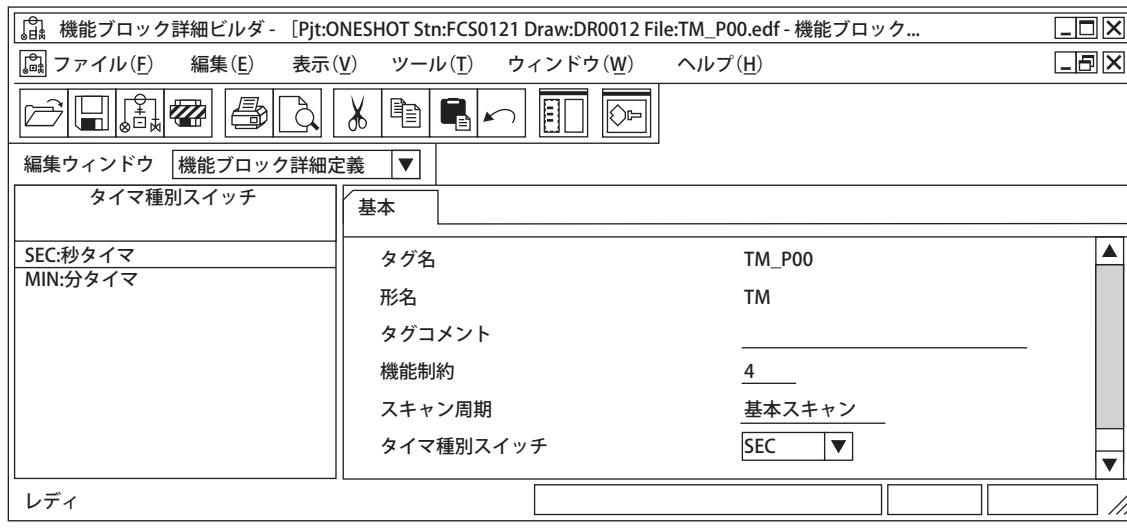


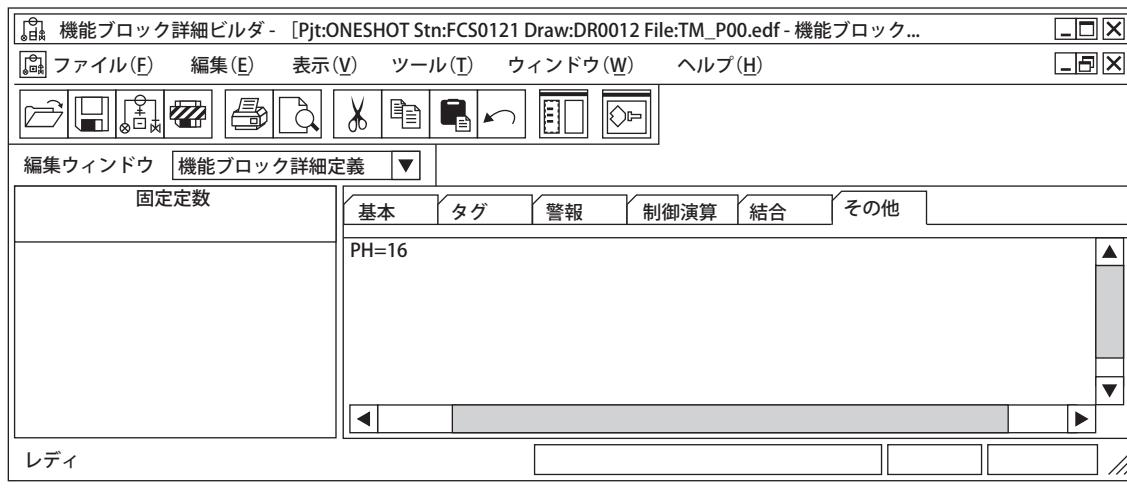
図 ST_P00の編集

タイマブロック TM_P00 は、下図のように 16 秒でカウントアップするように指定されています。



060704J.ai

図 タイマブロックの設定1



060705J.ai

図 タイマブロックの設定2

ユーザカスタムブロックをワンショット起動することによりカスケードクローズを早くするためのシーケンステーブルが 2 つあります。制御ドローイングビルダのシーケンステーブルの実行順は、下段のカスケード結合から上段のカスケード結合の順に処理するようにします。

- ・ シーケンステーブル ST_P02 は CMC_PIDB02 と PID_P03 のカスケード結合に関する処理をします。
- ・ シーケンステーブル ST_P01 は CMC_PIDB01 と CMC_PIDB02 のカスケード結合に関する処理をします。
- ・ 実行順は、下段の処理をする ST_P02 が先で、上段の処理をする ST_P01 が後です。

● カスケードクローズを検出しワンショット起動する

下段のカスケード結合に関する処理をするシーケンステーブル ST_P02 の場合、処理タイミングは「TC」つまり、条件変化時のみ出力となっています。

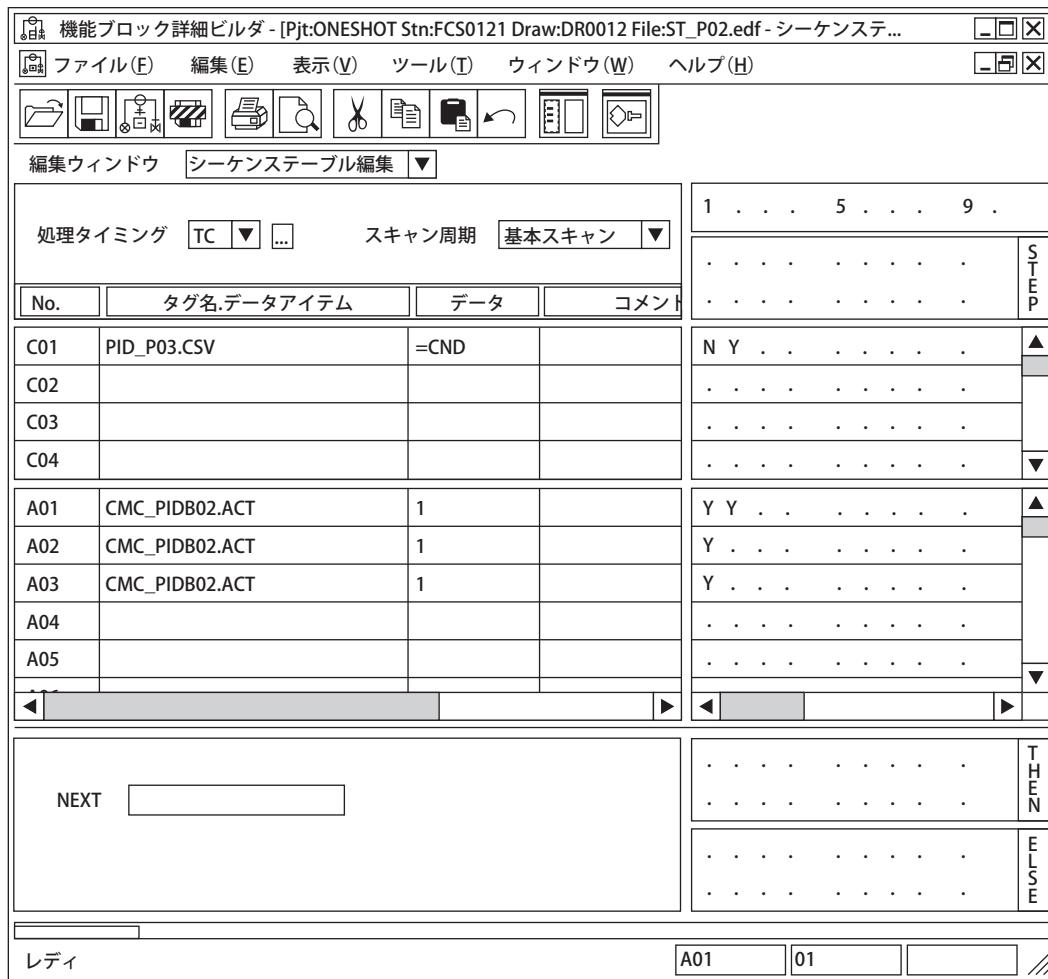


図 ST_P02

このシーケンステーブルは、CMC_PIDB02 と PID_P03 のカスケード結合に関する処理をします。カスケード下流の PID_P03 のデータアイテム CSV のデータステータス CND (コンディショナル) が、ON から OFF になったときに、上流の CMC_PIDB02 を 3 回ワンショット起動します。これはカスケードオープンからカスケードクローズへ移行するのを早める処理です。

参照 データステータス CND については、以下を参照してください。
「6.6 カスケード結合と実効スキャン周期」

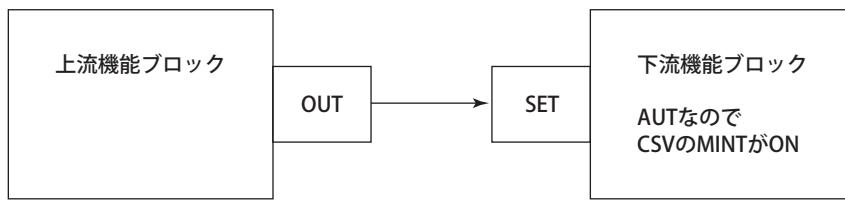
カスケード下流の PID_P03 のデータアイテム CSV のデータステータス CND (コンディショナル) が、OFF から ON になったときには、上流の CMC_PIDB02 を 1 回ワンショット起動します。これは、下流がカスケード結合できなくなったときに、すばやく上流のブロックモード IMAN を成立させるための処理です。

● 3回ワンショット起動する理由

シーケンステーブル ST_P02 は、ユーザカスタムブロックを 3 回連続してワンショット起動しています。3 回連続する必要があるのは、通常の機能ブロックの実行順とは逆順にワンショット起動するので、データステータス MINT（マスター初期化）を ON から OFF にする必要があるからです。

データステータス MINT は、カスケード下流の機能ブロックがカスケード上流の機能ブロックに、「下流のデータアイテム CSV の値は、前回上流側が出力した値とは違っているので、上流側は初期化動作をしてください」と通知するためのものです。

下図では、カスケード下流の機能ブロックが AUT でカスケードオープンです。



060707J.ai

図 上流の機能ブロックと下流の機能ブロックの関係

下流の機能ブロックは、ブロックモードが AUT なので、設定値プッシュバック処理は、データアイテム SV の値を、データアイテム CSV と RSV に設定します。設定値プッシュバックにより、データアイテム CSV の値が変更されると、システムは、CSV のデータステータス MINT を ON にします。

参照

設定値プッシュバックについては、以下を参照してください。

[「6.2.4 ブロックモード遷移と設定値の作成」の「■ 設定値プッシュバック」](#)

たとえば、オペレータが下流機能ブロックの SV 値を手入力で変更した直後の処理タイミングで、(変更された) SV 値が CSV に設定値プッシュバックされるときに MINT が ON になります。また、IMAN が解除された処理タイミング (IMAN (CAS) から CAS へ変化) の設定値プッシュバックでも、システムは CSV の MINT を ON にします。

上流側の機能ブロックは、SET 端子の先の CSV のデータステータス MINT が ON であれば、(上流から下流への) 出力値の出力はしないで、出力値をトラッキングします（出力値トラッキングを指示するビルダ定義項目「出力値トラッキング」を「あり」とする指定は必要です）。この処理により、上流側が保持する「前回出力値」と下流が CSV に保持する値が同じになり、下流側への出力値が急変することがなくなります。

データステータス MINT は、上流側の機能ブロックが下流側の SET 端子経由で CSV よりデータをトラッキングすると ON から OFF になります。通常、制御ドローイングの機能ブロック実行順は、カスケード結合の上流から下流の順に実行順を与えますので、下流側に処理タイミングが与えられるときには、上流側からのトラッキングは済んでおり、下流側機能ブロックの CSV の MINT は OFF になっています。したがって、下流機能ブロックが CAS になったあとに処理タイミングが与えられ、データアイテム CSV のデータステータス CND が ON から OFF になるときには、MINT は OFF になっています。つまり、通常は次の順に動作します。

- ・ 上流機能ブロックが下流機能ブロックをトラッキングして下流の MINT が OFF になる。
- ・ 下流機能ブロックに処理タイミングが与えられ、下流機能ブロックの CND が OFF になる。
- ・ 次の周期で上流機能ブロックは下流機能ブロックの CND が OFF になったことを検知する。

しかし、ここで説明する処理では、ユーザカスタムブロックをワンショット起動していますので、「ワンショット起動するユーザカスタムブロックの下流側機能ブロックの CSV の MINT を OFF にする」ために、ワンショット起動が（1回多く）3回必要です。下流側の MINT を OFF にしないと、上流側機能ブロックの IMAN は消えません。

補足

ワンショット起動した場合には、UcaCtrlPidTiming は常に「今回を制御周期」と判定します。

MINT が ON になるのは下流機能ブロックの設定値プッシュバックで CSV の値が変更された場合です。下流機能ブロックの MINT が ON と場合と OFF の場合のそれぞれで、上流機能ブロックを 3 回ワンショット起動したときの処理内容を示します。

表 上流機能ブロックを3回ワンショット起動したときの処理内容

	下流機能ブロックのMINTがON	下流機能ブロックのMINTがOFF
1回目	上流側をワンショット起動すると、下流側の CSV のデータステータス MINT が ON から OFF になります。ブロックモード IMAN が成立しているので IMAN 動作となります。	上流側をワンショット起動すると、上流側のブロックモードの IMAN が消えます（IMAN (CAS) から IMAN になります）。制御演算の初期化を実行します。
2回目	上流側をワンショット起動すると、上流側のブロックモードの IMAN が消えます（IMAN (CAS) から IMAN になります）。制御演算の初期化を実行します。	上流側をワンショット起動すると、上流側の CSV のデータステータス CND が ON から OFF になります。制御演算の初期化を実行します。
3回目	上流側をワンショット起動すると、上流側の CSV のデータステータス CND が ON から OFF になります。制御演算の初期化を実行します。	制御演算の初期化を実行します。

PID と同じ動作をするユーザカスタムブロックが制御初期化をするときには、出力値の変化 (ΔMV) は 0 となります。

参照

出力値の変化については、以下を参照してください。

[「6.8 制御演算の初期化と出力処理」](#)

● 制御演算の初期化を実行させる

前の表では、2回目と3回目のワンショット起動でも「制御演算の初期化を実行」しています。これは、ユーザカスタムアルゴリズムのワンショット起動処理を行うUcaBlockOneshot関数で、「パラメータが1の場合には制御演算の初期化をする」ようにユーザプログラムを作成しているからです。下記に _SMPL_PID_B (pid_b.c) のワンショット起動処理を示します。

```

UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc, /* (IN/OUT) : ブロックコンテキスト */
    I32 code,           /* (IN) : 種別コード */
    I32 parameter,      /* (IN) : パラメータ */
    BOOL *result        /* (OUT) : 実行結果 */
)
{
    I32 rtnCode;          /* リターンコード */

    /* 状態操作でなければ何もしない */
    if (code != UCAONESHOT_CODEOPRT) {
        return UCAERR_NOPROC;
    }

    /* パラメータが1なら強制的に制御初期化を指示する */
    if (parameter == 1) {
        rtnCode = UcaCtrlInitSet(bc);
    }

    /* 処理は、定期期と同じ */
    rtnCode = UcaCtrlHandler(bc,
        pid_input,
        pid_control,
        pid_output,
        UCACTRL_ALLMODE,
        NOOPTION);

    *result = TRUE;

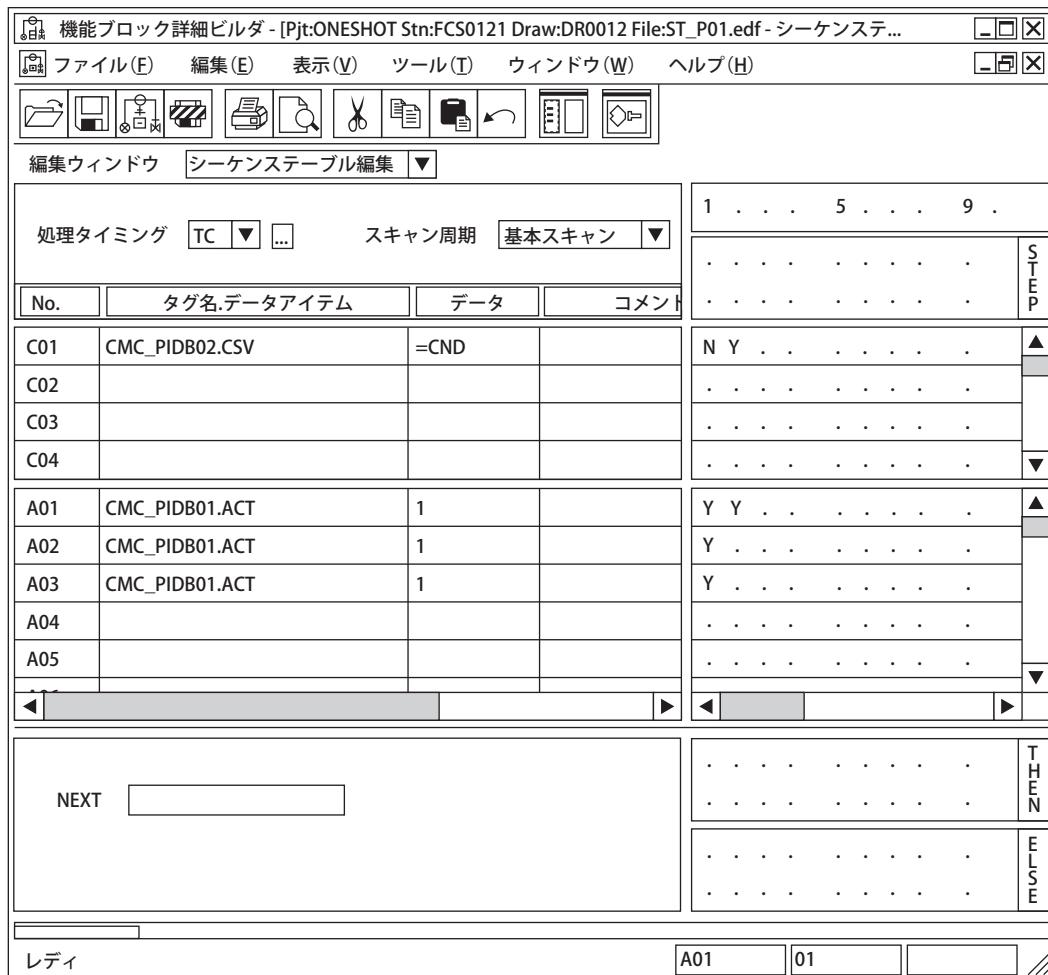
    return SUCCEED;
}

```

再度、シーケンステーブルST_P02を見ると、CMC_PIDB02をワンショット起動するときには、データに「1」を指定しています。この「1」が、UcaBlockOneshotの引数parameterにパラメータとして渡されます。

● カスケード結合に一つずつシーケンステーブルを使用する

カスケード結合 CMC_PIDB01 と CMC_PIDB02 の処理をするのは、シーケンステーブル ST_P01 です。



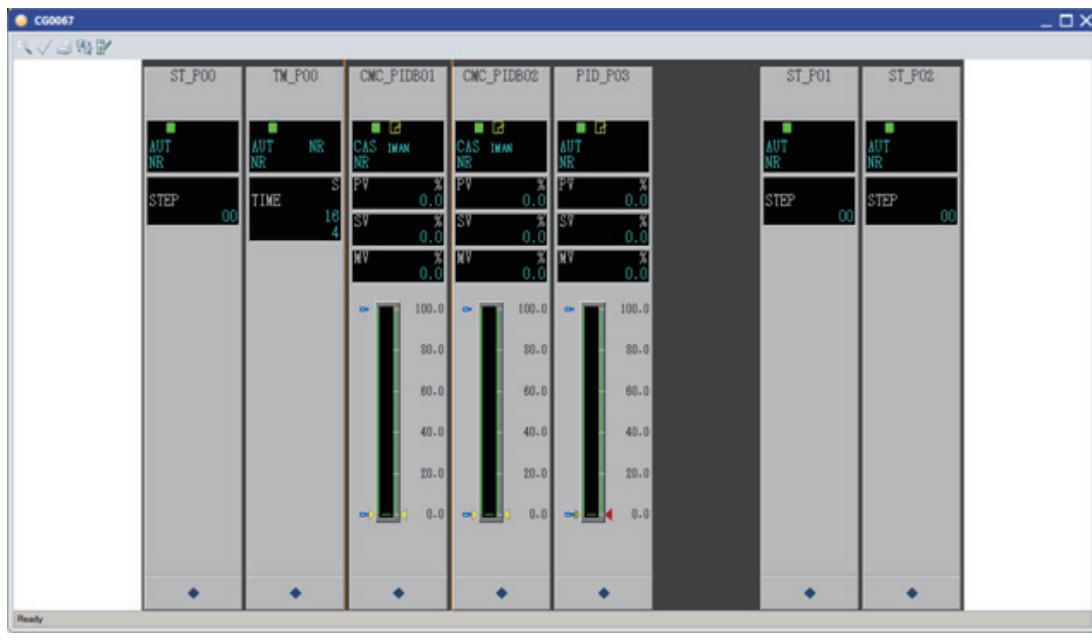
060709J.ai

図 シーケンステーブルST_P01

ST_P01 は、下流側の CMC_PIDB02 のデータアイテム CSV のデータステータス CND (コンディショナル) の変化に従い、上流側の CMC_PIDB01 をワンショット起動します。ST_P02 と ST_P01 の処理を分けて、別々のシーケンステーブルにしている理由を説明します。ST_P02 が CMC_PIDB02 をワンショット起動すると CMC_PIDB02 のデータアイテム CSV のデータステータス CND が変化します。この変化を捕まえるために、ST_P01 と ST_P02 を別々にし、ST_P01 には ST_P02 の処理が済んだあと (CMC_PIDB02 の CND が変化したあと) に処理タイミングを与えていました。

● カスケードクローズの動作

サンプルプログラムについて説明します。8 ループのビューが用意してあるので、開いてください。そして下図のようにブロックモードを設定してください。



060710J.ai

図 ブロックモードの設定例

カスケード結合の最下流の PID_P03 のブロックモードが AUT なので、CMC_PIDB01 と CMC_PIDB02 は、両方とも IMAN (CAS) となっています。

PID_P03 のブロックモードが AUT から CAS に変化すると、次の基本スキャン周期で、CMC_PIDB01 と CMCPID_B02 のブロックモードが、両方とも IMAN (CAS) から CAS に変化します。

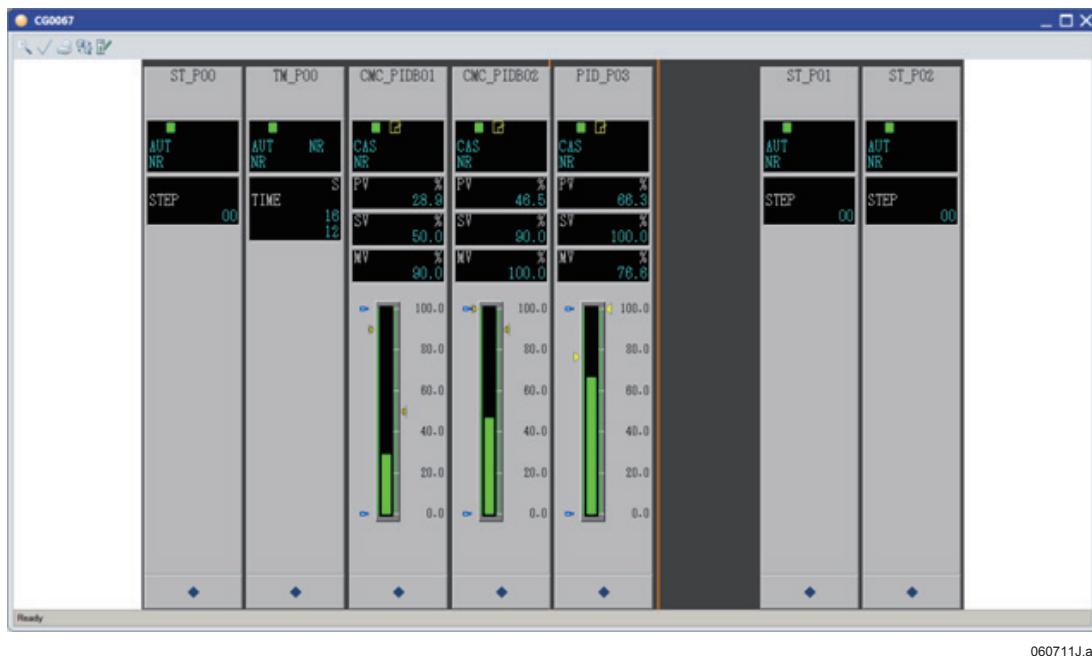


図 CMC_PIDB01、CMCPID_B02のブロックモードの変化

右側にある2つのシーケンステーブルをAUTからMANにするとカスケードクローズ処理、つまりIMANが落ちるまでの時間が長くなります。2つのシーケンステーブルのブロックモードをAUTとMANに変化させてから動作をさせると、ワンショット起動の効果を確認することができます。

■ 入力したデータのみから出力値を計算（カスケードの最下流が他ステーション）

入力したデータのみから出力値を計算するユーザカスタムブロックの例を示します。ユーザカスタムアルゴリズムは、「6.2 CSTM-C のブロックモード遷移」で説明した _SMPL_MODE を改造した _SMPL_MODE_B です。下図に制御ドローイングの一例を示します。

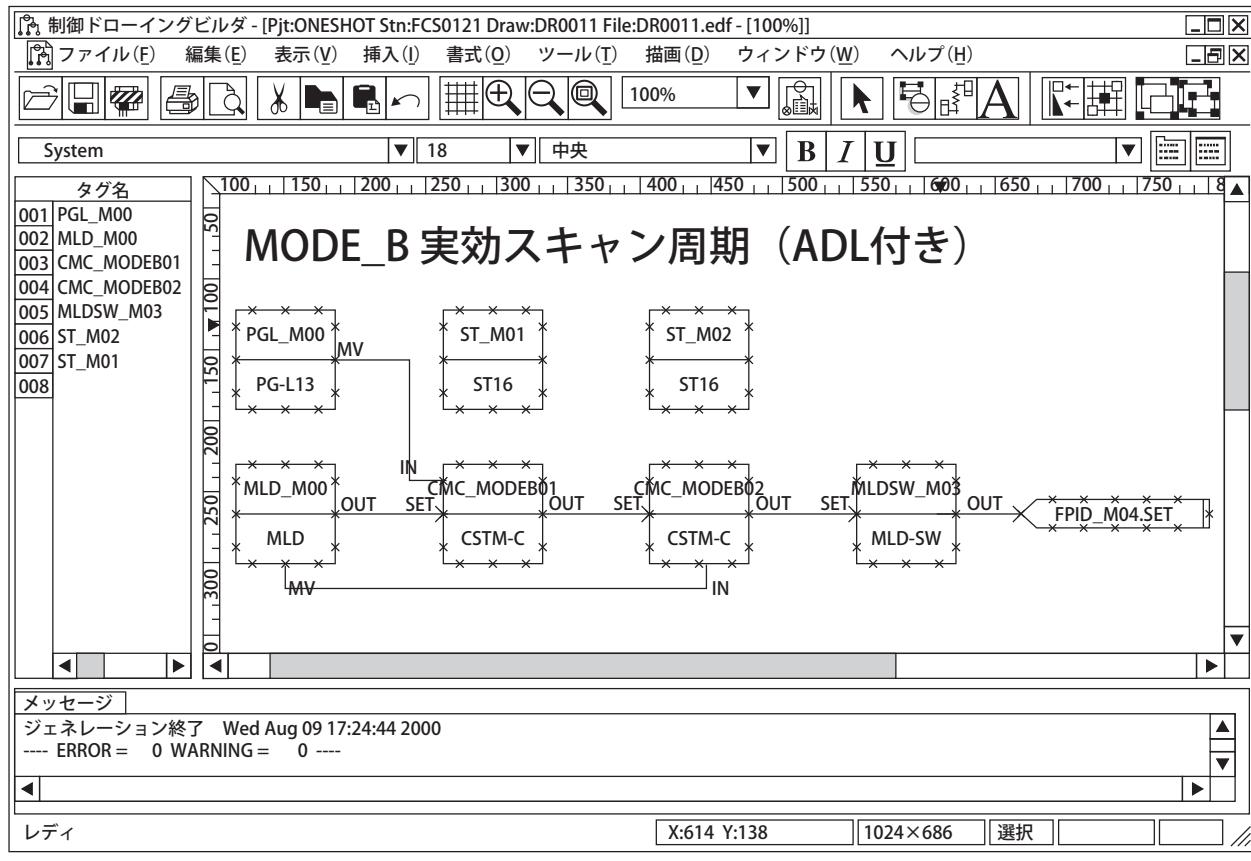


図 入力したデータのみから出力値を計算する制御ドローイング

この制御ドローイングには、CMC_MODEB01 と CMC_MODEB02 の 2 つのユーザカスタムブロックがあります。両方ともユーザカスタムアルゴリズムは _SMPL_MODE_B です。2 つとも実効スキャン周期に 16 秒を指定してあり、制御周期は使用しません（ユーザカスタムアルゴリズムで UcaCtrlPidTiming を使用していません）。

制御ドローイングには、2 つのカスケード結合があります。

- CMC_MODEB01 (CSTM-C) と CMC_MODEB02 (CSTM-C)
- CMC_MODEB02 (CSTM-C) と FCS0101 の FPID_M04 (PID)、間に MLDSW_M03 が存在します。

それぞれのカスケード結合に関する処理をするシーケンステーブル ST_M01 と ST_M02 があり、下段のカスケード結合の処理をする ST_M02 を先に実行し、ST_M01 を後に実行します。

FCS 側の制御ドローイングを下図に示します。

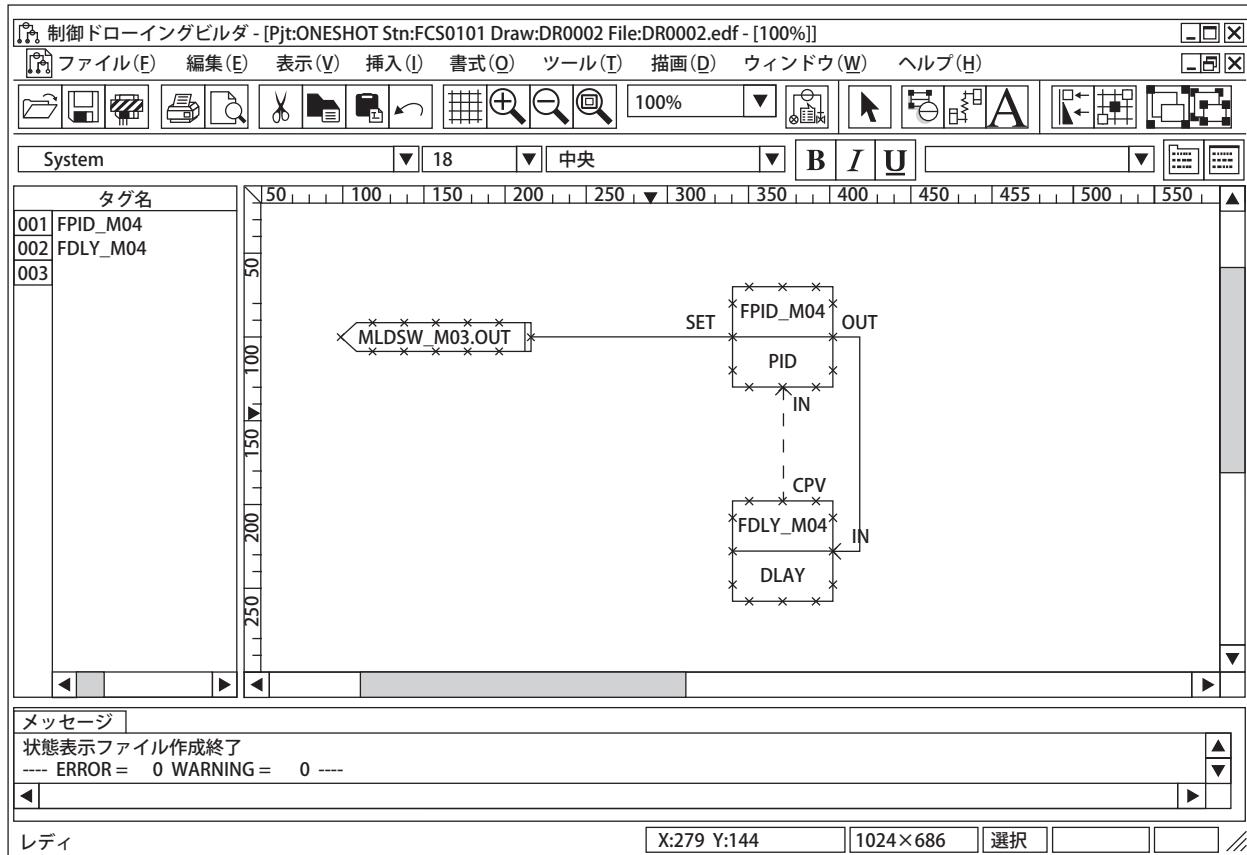
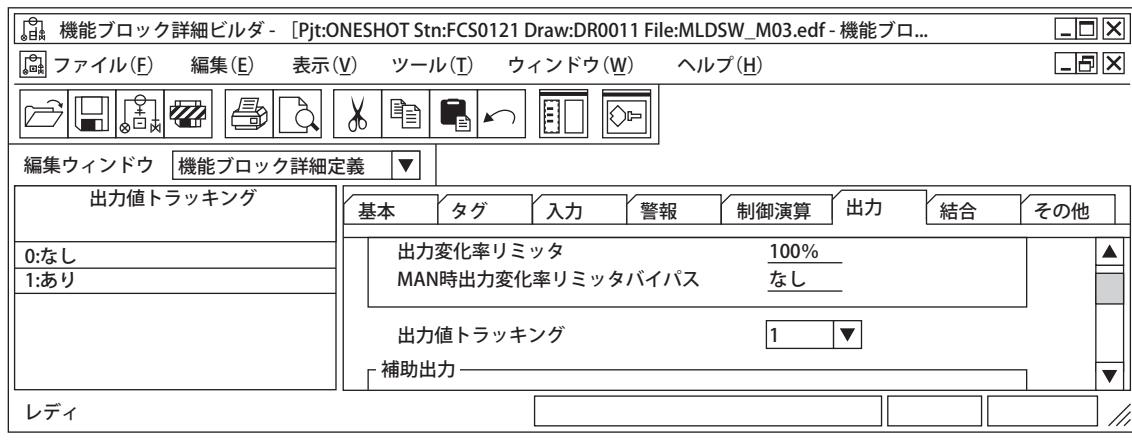


図 FCS側の制御ドローイング

● FCSとの間のカスケード結合にはMLD-SWブロックをはさむ

この制御ドローイングでは、FCS の FPID_M04 と APCS の CMC_MODEB02 の間に MLDSW_M03 (MLD-SW) をはさんでいます。CMC_MODEB02 の実効スキャン周期は 16 秒です。MLDSW_M03 は基本スキャン周期の 4 秒周期で動作します。このように、APCS 側の FCS に最も近いユーザカスタムブロック (ここでは CMC_MODEB02) の実効スキャン周期が長い場合には、基本スキャン周期ごとに処理タイミングを与えられる MLD-SW を (APCS 側に) はさみ、後述するシーケンステーブルを組み合わせるとカスケード結合のオープンとクローズを早くすることができます。MLDSW_M03 (MLD-SW) は 4 秒周期で、FCS の FPID_M04 のデータを参照します (ADL 経由です)。MLD-SW をはさまない場合、CMC_MODEB2 は、実効スキャン周期 (16 秒) ごとに FPID_M04 のデータを参照し、時間間隔が長くなります。

MLDSW_M03 は、出力トラッキングを「あり」(デフォルトは「なし」) に指定する必要があります。



060714J.ai

図 出力トラッキングの設定

補足 この出力トラッキングでは、MLD-SW が、制御ステーション間結合 (ADL) を経由して、FCS のデータを取得することになります。したがって、FCS のデータが APCS の MLD-SW に反映されるのにかかる時間は、制御ステーション間結合のデータ通信周期に依存します。

参照 制御ステーション間結合のデータ通信周期の詳細については、以下を参照してください。
APCS (IM 33J15U10-01JA) 「2.5 制御ステーション間結合」

● 下段のカスケード結合の処理をするシーケンステーブルST_M02

シーケンステーブルST_M02は、他ステーションデータにアクセスすることができません。このため、シーケンステーブルは MLDSW_M03 に反映された FPID_M04 のデータを参照しています。



060715J.ai

図 ST_M02の動作

MLDSW_M03 のデータアイテム CSV のデータステータス CND の変化に応じて、ユーザカスタムブロック CMC_MODEB02 をワンショット起動しています。

なお、FCS の FPID_M04 のブロックモードが AUT から CAS に変化して、MLDSW_M03 のデータアイテム CSV のデータステータス CND が ON から OFF になるには、MLDSW_M03 に 2 回処理タイミングが与えられる必要があります。つまり、2 基本スキャン周期必要です。CMC_MODEB02 の実効スキャン周期は 16 秒です。MLDSW_M03 は基本スキャン周期の 4 秒周期で動作します。このように、APCS 側の FCS に最も近いユーザカスタムブロック（ここでは CMC_MODEB02）の実効スキャン周期が長い場合には、基本スキャン周期ごとに処理タイミングを与えられる MLD-SW をはさむと有効です。上図のようにシーケンステーブルを使い、APCS と FCS に跨るカスケード結合のクローズとオープンの動作を速くすることができます。

● 上段のカスケード結合の処理をするシーケンステーブルST_M01

以下にシーケンステーブル ST_M01 を示します。



図 ST_M01

ルール番号 1 と 2 は、今までのシーケンステーブルと同じ処理をしています。このサンプルのユーザカスタムブロックは、実効スキャン周期が 16 秒なので、ルール番号 5 と 6 の処理を追加しています。カスケード下流のユーザカスタムブロック(CMC_MODEB02)の実効スキャン周期が 16 秒であり、データステータス CND の変化も 16 秒ごとに時間がかかるので、基本スキャン周期(4 秒)ごとにシーケンステーブルにルール番号 5 と 6 の追加処理を実行させます。

ルール番号 5 は、CMC_MODEB02 のブロックモードが CAS 以外(たとえば AUT)から CAS に変化したときに、CMC_MODEB02 を一度ワンショット起動してデータアイテム CSV のデータステータス CND を ON から OFF にし、(上流)の CMC_MODEB01 を 3 回ワンショット起動します。この処理をシーケンステーブルに入れないと、CMC_MODEB02 のデータステータス CND が変化するのに、最大実効スキャン周期の 16 秒かかります。

ルール番号 6 は、CMC_MODEB02 のブロックモードが CAS から CAS 以外（たとえば AUT）に変化したときに、上流の CMC_MODEB01 のブロックモード IMAN を成立させています。

この処理が必要になるのは、実効スキャン周期が 16 秒と長いためです。前述の PID_B のサンプルでは、ユーザカスタムブロックの実効スキャン周期が 4 秒(つまりシーケンステーブルが起動される基本スキャン周期と同じ) なので、ルール番号 5 と 6 の処理は必要ありませんでした。

● 制御初期化時は、前回の出力値を出力する

ユーザカスタムアルゴリズム _SMPL_MODE_B のワンショット起動処理を以下に示します。

```
UCAUSER_API I32 UCAAPI UcaBlockOneshot(
    UcaBlockContext bc, /* (IN/OUT) : ブロックコンテキスト */
    I32 code,           /* (IN) : 種別コード */
    I32 parameter,      /* (IN) : パラメータ */
    BOOL *result        /* (OUT) : 実行結果 */
)
{
    I32 rtnCode;          /* リターンコード */

    /* 状態操作でなければ何もしない */
    if (code != UCAONESHOT_CODEOPRT) {
        return UCAERR_NOPROC;
    }

    /* パラメータが 1 なら強制的に制御初期化を指示する */
    if (parameter == 1) {
        rtnCode = UcaCtrlInitSet(bc);
    }

    /* 処理は、定期周期と同じ */
    rtnCode = UcaCtrlHandler(bc,
        mode_input,
        mode_control,
        mode_output,
        UCACTRL_AUTCASRCAS,
        NOOPTION);

    *result = TRUE;

    return SUCCEED;
}
```

このプログラムも、パラメータ（引数 parameter）が 1 なら、UcaCtrlInitSet を呼び出し制御初期化を要求しています。

ユーザ記述の制御演算処理では、制御初期化を要求されると、前回の出力値をそのまま出力しています。

```
I32 mode_control(
    UcaBlockContext bc,           /* (IN/OUT) : ブロックコンテキスト */
    F64 *mvbbPtr,                /* (IN/OUT) : 出力補償前出力値ポインタ */
    F64 *mvblPtr                /* (IN/OUT) : 出力リミット前出力値ポインタ */
)
{
    F64S pv;                    /* PV */
    F32S sv;                    /* SV */
    F64S p01;                  /* P01 */
    F64S mv01;                  /* MV01 */
    I32 rtnCode;                /* リターンコード */
    BOOL isCtrlInit;             /* 制御初期化要求かどうか */

    /* 制御初期化（前回値をそのまま出力） */
    rtnCode = UcaCtrlInitCheck(bc, &isCtrlInit);
    if (isCtrlInit) {
        rtnCode = UcaDataGetMvbb(bc, mvbbPtr, NOOPTION);          /* MVBB 前回値 */
        *mvblPtr = *mvbbPtr;
        rtnCode = UcaCtrlInitClear(bc);                            /* 初期化要求をクリア */
        return SUCCEED;
    }
}
```

以下省略

まとめると次のようにになります。

- ここで説明するプログラム作成方法では、カスケード結合に関するワンショット起動をする場合には、ワンショット起動のパラメータに 1 を指定します。
- パラメータが 1 の場合、ワンショット起動処理は制御初期化の動作をするようにします。
- 制御初期化では、出力値の変化 (ΔMV) を 0 になるようにするか（前述の PID タイプの例）、前回値を出力するなどして、出力値が急変しないようにします。
- 制御初期化の処理自体は、ワンショット起動以外にも通常のモード変化（CAS 以外から CAS への変化など）で実行されることがありますので、本来の制御初期化の処理が、カスケード結合の処理のためのワンショット起動と共にできるように設計してください。

● カスケードクローズの動作

サンプルプログラムを動かします。8 ループのビューが用意してあるので、開いてください。そして下図のようにブロックモードを設定してください。

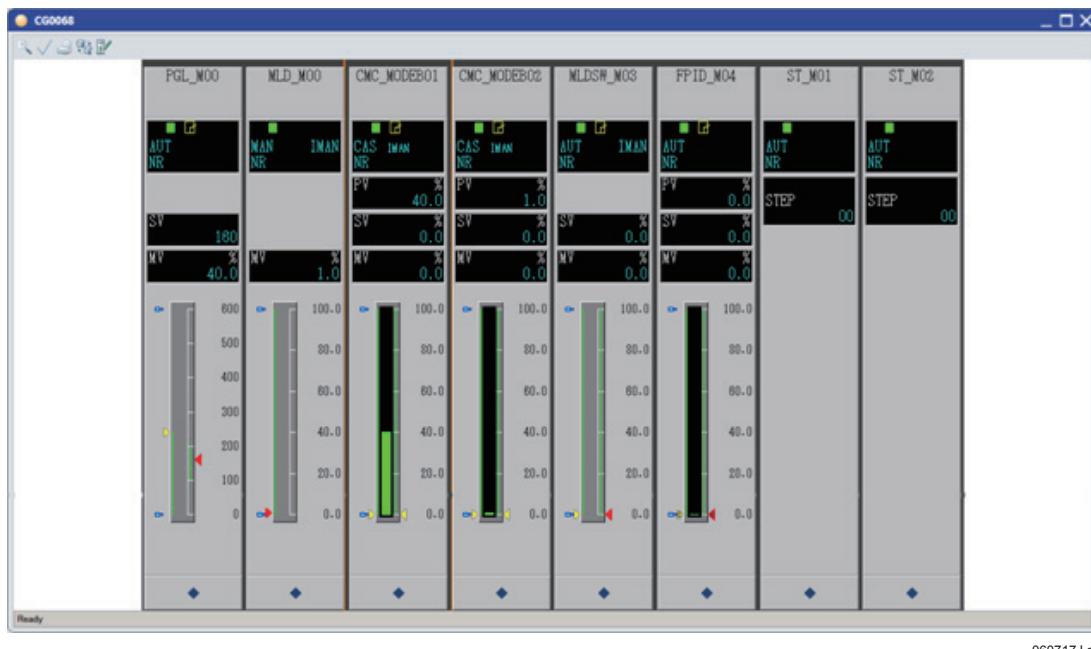


図 カスケードクローズの動作1

カスケード結合の最下流の FPID_M04 のブロックモードが AUT なので、それより上流の MLD_M00 から MLDSW_M03 は、すべて IMAN が成立しています。

FPID_M04 のブロックモードを AUT から CAS に変化すると、そのあとの 2 回目の基本スキャン周期で、MLDSW_M03 のブロックモードが、IMAN (AUT) から AUT に変化します。さらに 1 基本スキャン周期で CMC_MODEB01 と CMC_MODEB02 が IMAN (CAS) から CAS になります。その後 2 回の基本スキャン周期で MDL_M00 の IMAN が落ち、下図の状態となります。

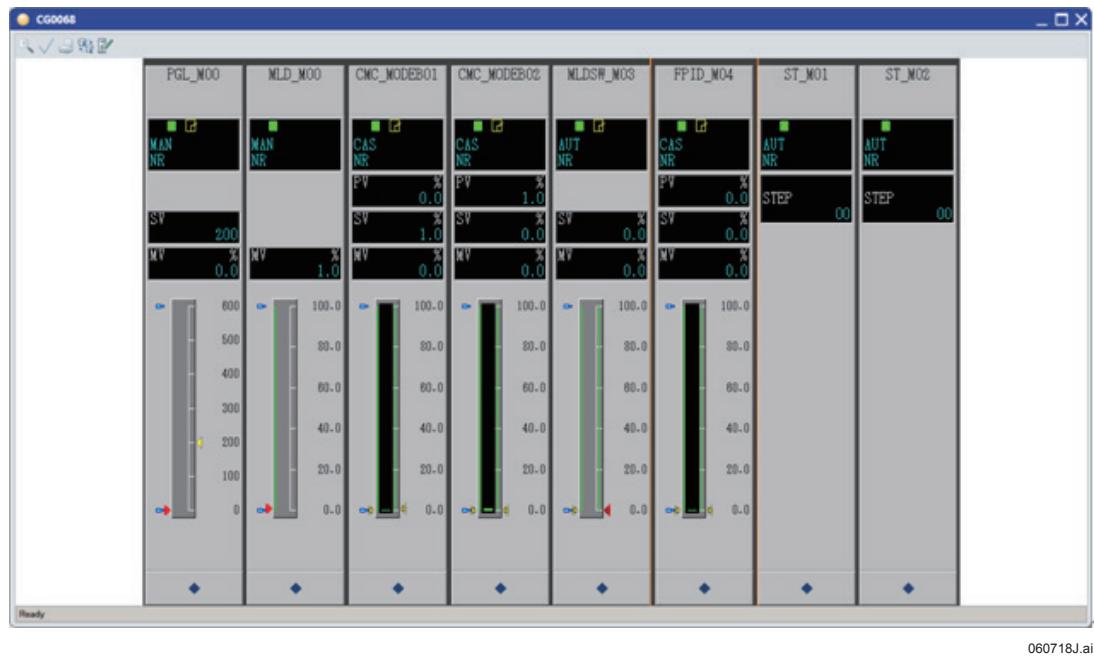


図 カスケードクローズの動作2

上図の状態から CMC_MODEB02 のブロックステータスを CAS から AUT に変化すると、次の基本スキャン周期で CMC_MODEB01 のブロックステータスが CAS から IMAN (CAS) になります。

また、CMC_MODEB02 のブロックステータスを AUT から CAS に変化すると、次の基本スキャン周期で CMC_MODEB01 のブロックステータスが IMAN (CAS) から CAS になります。

これらの処理が次の基本スキャン周期で行われるのは、シーケンステーブル ST_M01 のルール番号 5 と 6 の記述の効果です。

■ シーケンステーブルの処理タイミング

シーケンステーブルとユーザカスタムブロックの処理タイミングについて要点を説明します。

- 間欠制御を指示（ユーザカスタムブロックのCSWに1を設定）するシーケンステーブルは、実行順をユーザカスタムブロックより前にしてください。
- カスケードクローズのために、ユーザカスタムブロックをワンショット起動するシーケンステーブルは、実行順をユーザカスタムブロックより後にしてください。つまり、ユーザカスタムブロックに処理タイミングを与えたあとに「CSVのデータステータスCNDがONからOFFになった」かを検査します。ワンショット起動するユーザカスタムブロックのビルダ定義項目には「定周期とワンショット起動の両方」を指定してください。
- カスケードクローズのためのシーケンステーブルは、カスケードの下段から上段の順に実行してください。下図のように、ユーザカスタムブロックC1、C2、C3がある場合には、C2とC3のカスケード結合の処理をするシーケンステーブルが先で、C1とC2の処理をするシーケンステーブルがあとです。

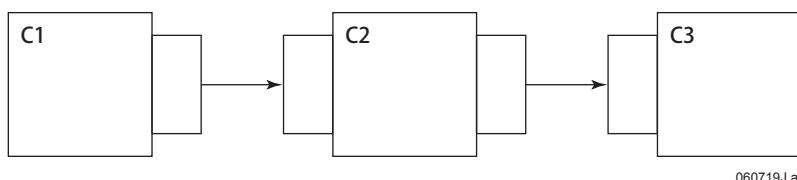


図 シーケンスケーブルの動作タイミング

- PID調節ブロックなど、標準の調節ブロックはワンショット起動できません。ワンショット起動でカスケードクローズを早くできるのはユーザカスタムブロックのみです。

■ 3回連続ワンショット起動が可能なユーザカスタムアルゴリズム

ここで説明している方法では、同一のユーザカスタムブロックを同じ処理タイミングで3回連続してワンショット起動しています。ユーザカスタムブロックが3回のワンショット起動で同じ出力値を出力するようにしておく必要があります（アプリケーション次第ですが、通常は同じ出力値を出力することを推奨します）。

- 入力したデータのみから、出力値を決定する場合、アプリケーションとして問題が無ければ、通常の出力処理をしてください。同一処理タイミングに連続して実行しても、入力データが同じなので、出力値も同じになります。
- カスケード結合のためのワンショット起動で出力値が変化すると問題がある場合の処理例として、SMPL_MODE_Bは、前回の出力補償前出力値を出力しています。
- 制御演算で前回出力値からの差分(ΔMV)を作り出す場合、 ΔMV が0になります。PID演算初期化とPID演算のライブラリ関数(UcaCtrlPidInitとUcaCtrlPid)を使用していれば ΔMV が0になります。 ΔMV を0にすることにより、出力値は、前回の出力値と変化がなくなります。これもアプリケーション次第ですが、最低限、前回出力からの変化が小さくなるようにする必要があります。

■ 制限事項

ここで説明した手法では、ワンショット起動を使用しています。以下に制限事項を説明します。

- ・ワンショット起動の場合、積算処理と入力変化率警報検出は行われません。

参照 この制限事項の詳細については、以下を参照してください。

APCS ユーザカスタムブロック ライブラリ (IM 33J15U22-01JA) 「3.2.2 UcaRWSetPv」

- ・同じ処理タイミングに複数回連続してワンショット起動すると、SUB 端子から出力する ΔPV と ΔMV は、0 になります。
- ・制御周期を（間欠制御ではなく）自動決定または4、8、16、32、64秒固定で指定しているユーザカスタムブロックに対して、ワンショット起動をすると、ワンショット起動あととの最初の制御周期における前回からの時間差は、（ワンショット起動したことに関係なく）制御周期の指定値で決まります。前回の PV 値などは、ワンショット起動したときに保存したものが使われる所以、前回時間と保存したデータに少々のズレが発生します。

6.8 制御演算の初期化と出力処理

制御演算の初期化と、IMANが消えた直後の制御周期の出力について説明します。

制御演算の初期化は、UcaCtrlPidInit関数またはUcaCtrlFuncInit関数により行います。UcaCtrlPidInitは、PID演算をする場合（つまりUcaCtrlPidを使用する場合）の初期化をします。UcaCtrlFuncInitは、UcaCtrlPidを使用しない場合に制御演算関数の初期化をするため使用します。

それぞれの初期化用関数の内部処理とその効果を下表に示します。

表 制御演算の初期化の効果

初期化される機能と 関連する関数	UcaCtrlPidInit関数の効果	UcaCtrlFuncInit関数の効果
制御周期カウンタの初期化 (UcaCtrlPidTiming)	前回からの時間差が 0 になります。その結果、今回の積分項が 0 になります。	同一処理タイミング内で UcaCtrlPidGetTime で取得できる前回からの時間差は 0 になります。
不感帯動作の初期化 (UcaCtrlDeadband) (UcaCtrlDeadband_p)	その処理タイミングの不感帯のヒスティリシスを無効化します。	その処理タイミングの不感帯のヒスティリシスを無効化します。
PID 演算の内部変数 (UcaCtrlPid)	比例項、微分項の演算を初期化します。 初期化の処理の一部として UcaCtrlPidInit の引数に指定される cv(入力補償あとの入力値) を保持します。 この結果その処理タイミングでは、前回と今回の cv が同じになりますので、比例項、微分項が 0 になります。	(PID 演算に関する初期化はしません。)

注： 内部的には、UcaCtrlPidInit の処理は、UcaCtrlFuncInit の処理に PID 演算の内部変数の初期化を加えたものです。

上の表より、制御演算の初期化をすると、PID 制御演算の比例項、微分項、積分項はすべて 0 になります。つまり、前回からの出力値の差分 (ΔMV) が 0 になります。

参照 PID 制御演算の計算式については、以下を参照してください。

[機能ブロッククリアレンス Vol.1 \(IM 33J15A30-01JA\) 「1.5 PID 調節ブロック \(PID\)」の「■ PID 制御演算」](#)

制御演算の初期化のタイミングはシステムが決定します。ユーザプログラムでは、ライブラリ関数 UcaCtrlInitCheck により（システムが決定した）初期化が必要か否かを検査します。下図は、PID 調節ブロックと同じ動作をするサンプルプログラム _SMPL_PID の制御初期化の部分です。

```
.....
/* 制御初期化 */
rtnCode = UcaCtrlInitCheck(bc, &isCtrlInit);
if (isCtrlInit) {
    rtnCode = UcaCtrlPidInit(bc, cv, NOOPTION);
    rtnCode = UcaCtrlInitClear(bc);
}
.....
```

図 サンプルプログラム内制御初期化の部分

ユーザカスタムブロックの IMAN が消えた直後の制御周期（たとえば IMAN (CAS) から CAS になった直後）には、UcaCtrlInitCheck は、「制御初期化が必要である」とし、引数 isCtrlInit に TRUE (真=制御初期化が必要) を返します。この結果、UcaCtrlPidInit が呼び出され制御初期化が実行されます。

実際には、IMAN (CAS) から CAS、AUT から CAS など、CAS 動作以外をするブロックモードから CAS ヘブロックモードが変化したときには、UcaCtrlInitCheck は「制御初期化が必要である」としています。

以上をまとめると、以下のようにになります。

- ・ CAS 動作以外をするブロックモードから CAS 動作をするブロックモードになったときには、UcaCtrlInitCheck は「制御初期化が必要」という判定をします。
- ・ 制御初期化処理で UcaCtrlPidInit を呼び出すと、PID 演算の比例項、微分項、積分項は 0 になり、この結果、出力値の差分 (ΔMV) が 0 になります。
- ・ この結果、制御初期化をした処理タイミングでは、3 回連續してワンショット起動をしても、 ΔMV は 0 であり、出力値は変化しません。
- ・ UcaCtrlFuncInit で制御演算関数の初期化をし、演算自体はユーザプログラムで独自に記述する場合には、UcaCtrlInitCheck で「制御初期化が必要」としたときの処理（つまり UcaCtrlFuncInit の前後）に、演算結果の出力値の差分 (ΔMV) が 0 となるような工夫を入れてください。あるいは、制御初期化をしたときには、前回の出力リミット前出力値か前回の MV 値を出力値として使用してください。

参照 上記内容については、以下を参照してください。

「6.3.3 制御初期化」

「6.3.6 制御出力動作」

■ 制御初期化をした処理タイミングにおける出力値

制御周期が有効なプログラムの場合、制御初期化が実行されるタイミングには、次の2種類があります。IMANが消える処理は、実効スキャン周期のたびに毎回行われることに注意してください。

- IMANが消えた実効スキャン周期が制御周期でない場合
この場合は、次回以後の最初の制御周期で制御初期化が行われます。
- IMANが消えた実効スキャン周期が制御周期の場合
この場合は、IMANが消えるのと同じ処理タイミングの一連の処理で制御初期化が行われます。制御周期を使用しないで実効スキャン周期のみで動作する場合には、この分類となります。また、UcaCtrlPidTimingを使用しているプログラムをワンショット起動する場合（定周期起動との混在は不可）もこの分類となります。

補足 IMANを消す処理は、UcaCtrlHandlerのブロックモード遷移処理で行われます。ブロックモード遷移処理は、実効スキャン周期ごとにUcaCtrlHandlerにより行われます。

参照 ブロックモード遷移処理については、以下を参照してください。
[「6.2.4 ブロックモード遷移と設定値の作成（UcaCtrlHandlerの内部で処理）」](#)

この2種類の場合のUcaCtrlHandlerの出力値作成処理の動作を下表に示します。

表 IMANが消えた直後の制御周期における出力値作成処理

	IMANが消えた実効スキャン周期が制御周期の場合の動作	IMANが消えた実効スキャン周期が制御周期ではない場合の、次の制御周期時の動作
例	6.6節「表カスケードオーブンからカスケードクローズになるまで（その1）」の2-C2と4-C1 6.6節「表カスケードオーブンからカスケードクローズになるまで（その2）」の4-C1	6.6節「表カスケードオーブンからカスケードクローズになるまで（その2）」の4-C2
出力値	出力値は、IMAN成立時と同じになります。 IMANが消えた処理タイミングが制御周期である場合、UcaCtrlHandlerの出力値作成処理は、IMAN成立時と同じ動作をします。	ユーザ記述の制御演算処理が引数に返すMvbl（出力リミット前出力値）が出力されます。 つまり、UcaCtrlHandlerの出力値作成処理は、CASやAUT時と同じ動作をします。

上表からわかるように、IMANが消えるのと同じ処理タイミングで制御初期化が行われた場合、その処理タイミングでは、システムが「IMAN成立時と同じ」出力動作をします。ユーザアプリケーションは、この動作を変更することはできません。

一方、IMANが消える実効スキャン周期が制御周期でない場合には、次回の制御周期の出力はユーザ記述の制御演算処理が作成し引数Mvblに返す値が出力されます。

Blank Page

7. タグ名を指定したデータ入力

この章では、タグ名を指定したデータ入力のプログラミングについて説明します。ユーザカスタムアルゴリズム作成用ライブラリには、APCS内の機能ブロックデータ入出力の関数と、APCS外（FCS）の機能ブロックデータ入出力の関数が用意されています。これらの関数の使い方は、連続制御形ユーザカスタムブロック（CSTM-C）と汎用演算形ユーザカスタムブロック（CSTM-A）で同じです。

表 タグ名を指定したデータ入出力の関数

	他ステーション (APCS外)	自ステーション (APCS内)
数値データ読み込み (副入力 RVnn 設定付き)	UcaOtherTagReadToRvnF64S	UcaTagReadToRvnF64S
数値データ読み込み	UcaOtherTagReadF64S	UcaTagReadF64S
	UcaOtherTagReadI32S	UcaTagReadI32S
	UcaOtherTagRead	UcaTagRead
数値データ書きこみ	UcaOtherTagWriteF64S	UcaTagWriteF64S
	UcaOtherTagWritel32S	UcaTagWritel32S
文字列データ読み込み	なし	UcaTagReadString
文字列データ書き込み	なし	UcaTagWriteString
汎用データ型読み込み	UcaOtherTagRead	UcaTagRead
汎用データ型書き込み	UcaOtherTagWrite	UcaTagWrite

■ F64S型の関数を使用

数値データにアクセスする場合、通常は F64S 型用の関数を使用します（関数名の末尾が F64S）。入力用の関数は、入力したデータを F64S 型に変換します。また、出力用の関数は引数に指定された F64S 型のデータを出力先のデータに合わせて型変換します。したがって、ユーザのプログラムは一貫してデータを F64S 型で扱うことができます。処理対象のデータを特に整数型として扱う必要がある場合（フラグなどとして使用する「==」で比較するデータなど (*1)）には、I32S 型の関数を使用してください（関数名の末尾が I32S）。

*1： 浮動小数データは有限な精度を持つ値であるため、演算を行ったり、F64 型から F32 型への型変換を行ったりすると、ほとんどの場合は演算結果に微小な誤差がでます。誤差を含んだ演算結果に対して、「==」による比較をすると、結果は不一致となります。したがって、浮動小数データの比較には、>=、<=、>、<を使用してください。また、フラグなど「==」による比較が必要なデータには、浮動小数データではなく整数データを使用してください。

■ 副入力RVnnへの設定付きデータ読み込み

UcaOtherTagReadToRvnF64S と UcaTagReadToRvnF64S は、データを入力すると同時に、ユーザカスタムブロックのデータアイテム RVnn（副入力）にデータを格納します。データアイテム RVnn は、F64S 型、つまりデータステータス付きのデータ型を持ちます。UcaOtherTagReadToRvnF64S または UcaTagReadToRvnF64S でデータを入力すると、CSTM-C では PV の作成時に、CSTM-A では CPV の作成時に、ビルダ定義項目「演算入力値異常検出」の指定により、入力値 RV と副入力 RVnn のデータステータスを PV または CPV に反映することができます。

参照 詳細については、以下を参照してください。

「5.1 多入力 CSTM-A」の「■ UcaRWSetCpv による CPV のデータステータス作成」

「6.1.1 UcsRWSetPv による PV のデータステータス作成」

■ タグ名を指定したデータ入力

7.2 節では、6.1 節の「多入力 CSTM-C（ブロックモードは AUT と O/S に限定）」プログラムの入力端子からのデータ入力を、タグ名を指定した入力に置き換えます。ここでは FCS の機能ブロックデータ（他ステーションデータ）を UcaOtherTagReadToRvnF64S により入力します。また、入力値 RV と副入力 RVnn を使用して測定値 PV のデータステータスを作成します。

7.3 節では、5.2 節の「多入力多出力 CSTM-A」プログラムの入力端子からのデータ入力を、タグ名を指定した入力に置き換えます。ここでは APCS 内の機能ブロックデータ（自ステーションデータ）を UcaTagReadToRvnF64S により入力します。また、入力値 RV と副入力 RVnn を使用して演算結果 CPV のデータステータスを作成します。

7.1 タグ名と入出力端子によるデータ入出力の違い

一般にタグ名を使用してプログラムを作成すると、1つのユーザカスタムアルゴリズムは1つのユーザカスタムブロックでのみ使用可能となります。つまり、ユーザカスタムアルゴリズムを複数のユーザカスタムブロックで共用することはできなくなります。タグ名を使用して記述したユーザカスタムアルゴリズムを複数のユーザカスタムブロックのプログラム名称に指定することはできますが、同じ機能ブロックに対して複数のユーザカスタムブロックから同じ処理をすることになるので、ユーザのアプリケーションとしては（特別な意図がある場合を除いては）誤りとなります。一方、タグ名を使用して記述すると、ユーザカスタムアルゴリズムを見るだけで処理対象のタグ名がわかります。

入力端子を使ったデータ入力では、アラーム処理が適応されます。タグ名を利用したデータ入力では、（UcaOtherTagReadF64Sなどの関数への）オプション指定により入力値不良アラームチェック（不良時はIOPアラームを発生）の検出のみが可能です。

FCSへのセットポイント制御を目的として連続制御形ユーザカスタムブロックを使用する場合、制御ステーションの連続制御ブロックの設定値に出力するデータは、ユーザカスタムブロックのOUT端子から出力してください。タグ名を指定した出力はしないでください。OUT端子による出力には、出力リミッタ処理やアラーム処理が適応されます。タグ名を指定した出力ではこれらの処理は適応されず、（UcaOtherTagWriteF64Sなどの関数の）引数に指定されたデータがそのままFCSの機能ブロックに出力されてしまい危険です。タグ名を指定したデータ出力は、FCSの制御に直接関係のない「単なるデータの設定」の場合にのみ使用してください。

要点を整理します。

- ・ 一般に、タグ名を使用したプログラミングをすると、ユーザカスタムアルゴリズムを複数のユーザカスタムブロックで共用できなくなります。

参照 詳細については、以下を参照してください。
「[4.7 ユーザカスタムアルゴリズムの共用](#)」

- ・ タグ名を指定したデータ入力では、入力値不良アラームチェックのみが可能です。
- ・ FCS のセットポイント制御を目的としたデータ出力は、OUT 端子から出力してください。タグ名を指定したデータ出力はしないでください。
- ・ FCS へのデータ出力は、通常 OUT 端子か J01～J16 端子より出力してください。特に FCS の制御に関連する大切なデータは、OUT 端子から出力してください。OUT 端子による出力には、出力リミッタや出力変化率リミッタが適応されます。

参照 詳細については、以下を参照してください。
「[6.2.7 出力処理（ユーザ記述）](#)」

- ・ タグ名を指定したデータ出力は、FCS の制御に直接関係のない「単なるデータ設定」にのみ使用してください。

7.2 他ステーションデータ（副入力RVnnを使用）

ユーザカスタムアルゴリズムからの他ステーションデータアクセスは、制御ステーション間結合を利用します。制御ステーション間結合とは、自ステーション（APCS）の機能ブロックと他ステーション（FCS）の機能ブロックとの間で、データ結合または端子間結合を行う入出力結合方式です。

制御ドローリングビルダで他の制御ステーションの機能ブロックに対する入出力結合情報を指定すれば、システムによりステーション間結合ブロック（ADL）が生成され、他ステーションの機能ブロックとのデータのやり取りがADLブロックを介して行われます。

ユーザカスタムアルゴリズムより、他ステーションの機能ブロックデータにアクセスするためにも、ADLブロックが必要です。このADLブロックは、ユーザが1バッチ形数値データ設定ブロック（BDSET-1L）を定義することにより生成します。BDSET-1Lにアクセス対象の機能ブロックデータを指定すると、その機能ブロックデータにアクセスするためのADLブロックがシステムにより自動生成されます。

重要

- この章で説明する他ステーションデータアクセス関数を使用する場合は、BDSET-1Lをステーション間結合ブロック（ADL）を自動生成するために使用します。BDSET-1L本来の機能は各種設定値や制御パラメータを他ステーションに設定することですが、同一のAPCS内ではBDSET-1Lをバッチデータ設定ブロックの機能としては使用しないでください。バッチデータ設定ブロックの機能が必要な場合には、2バッチ形数値バッチデータ設定ブロック BDSET-2L を使用してください。BDSET-2Lは、BDSET-1Lの機能を包含しています。
- 他ステーションデータアクセス関数のためのADLブロックは、BDSET-1Lで生成してください。BDSET-2LなどBDSET-1L以外のバッチデータ設定ブロックで生成したADLブロックは、他ステーションデータアクセス関数の動作に無関係です。

サンプルプログラムが用意しておりますので、動かしてみます。サンプルソリューション _SMPL_ALGO_FTAG の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは1章の準備作業により以下の作業フォルダにコピーされています。

<ドライブ名>:\UcaWork\UcaSamples_SMPL_ALRM_FTAG

Visual Studio を起動し、_SMPL_ALRM_FTAG__SMPL_ALRM_FTAG.slnを開きます。[ビルド] メニューの [リビルド] で _SMPL_ALRM_FTAG の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。

次にサンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイル ALRM_FTAG.txt と FALRM_FTAG.txt が CENTUM VP インストール先の以下にあります。ALRM_FTAG.txt は FCS0121 (APCS) に、FALRM_FTAG.txt は FCS0101 (FCS) にそれぞれインポートします。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥drawings¥ALRM_FTAG.txt と FALRM_FTAG.txt

FCS0101 (FCS) の制御ドローイングの DR0070 (空いている制御ドローイングならどれでも構いません) を指定して制御ドローイングビルダを起動します。[ファイル] メニューの [外部ファイル] – [インポート] を指定します。インポートダイアログで FALRM_FTAG.txt を指定し、[開く] ボタンをクリックすると 3 つの手動操作ブロック (FMLD_T10、FMLD_T11、FMLD_T12) を定義した制御ドローイングが取り込まれます。[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

同じ手順で、FCS0121 (APCS) の制御ドローイングの DR0070 (空いている制御ドローイングならどれでも構いません) に ALRM_FTAK.txt をインポートします。次図の制御ドローイングが取り込まれますので、[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

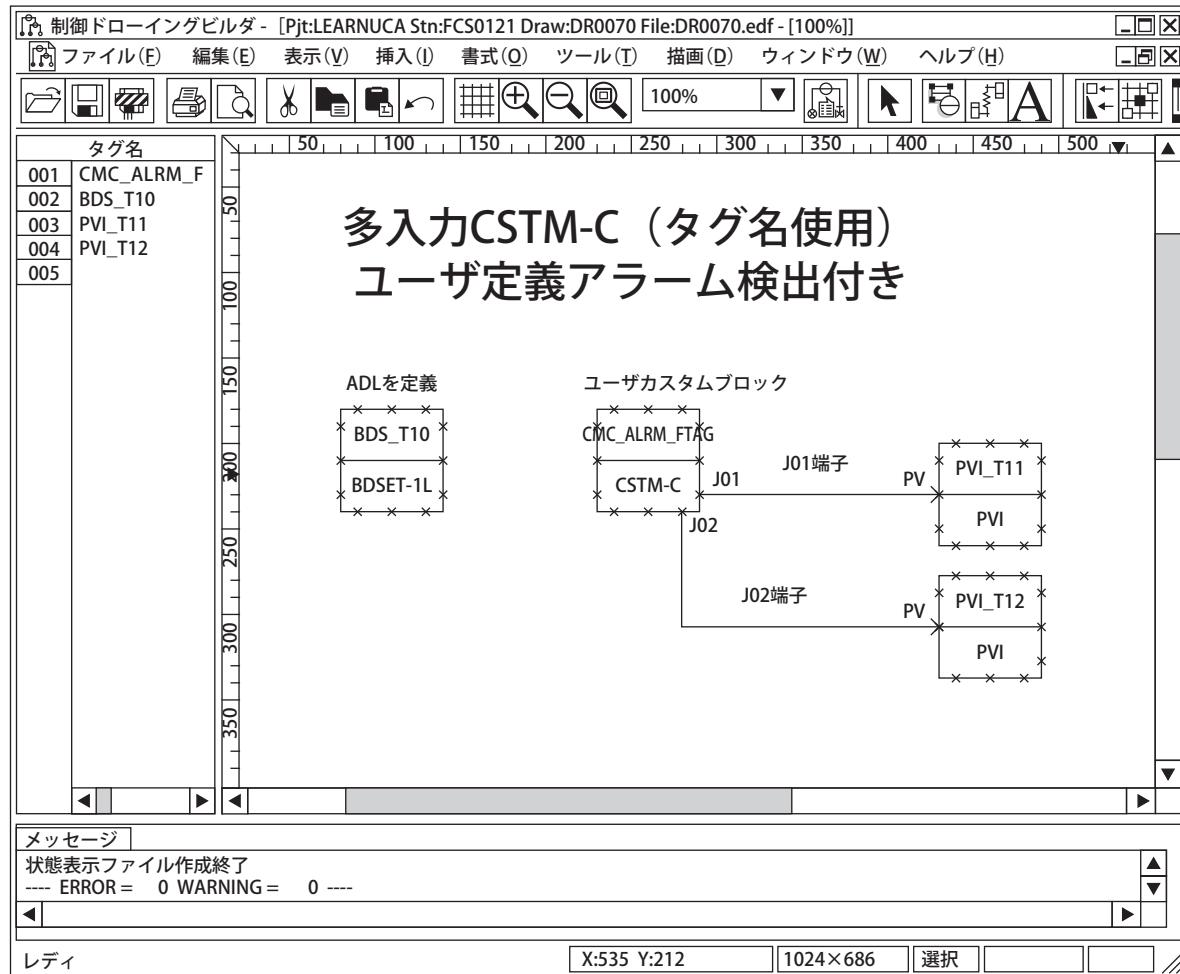


図 制御ドローイングのインポート

この制御ドローイングでは、連続制御形ユーザカスタムブロック (CSTM-C) が J01 端子と J02 端子から出力するデータを、2つの指示ブロック (PVI) が入力しています。CSTM-C は 3つの手動操作ブロックの MV 値 (FMLD_T10.MV、FMLD_T11.MV、FMLD_T12.MV) を入力していますが、ユーザカスタムアルゴリズムは「タグ名」を指定した入力方法で記述してあるので、制御ドローイング上にはデータ入力の配線はありません。1バッチ形数値データ設定ブロック (BDSET-1L) には、ステーション間結合ブロックを生成するために、CSTM-C が入力する 3つの MDL の MV 値 (FMLD_T10.MV、FMLD_T11.MV、FMLD_T12.MV) を定義してあります。CSTM-C の PV の計算式と J01 端子、J02 端子からの出力値を示します。

- PVは3つの入力の合計です。
 $PV = FMLD_T10.MV + FMLD_T11.MV + FMLD_T12.MV$
- J01端子からの出力値は、PV (FMLD_T10.MV、FMLD_T11.MV、FMLD_T12.MVの合計) です。
- J02端子からの出力値は、FMLD_T10.MV、FMLD_T11.MV、FMLD_T12.MVの平均です。

動作を確認するために、コントロール（8ループ）のウィンドウが用意してありますので、HIS0164のCG0070に取り込んでおきます。テキストファイルがCENTUM VPインストール先の以下にあります。

<CENTUM VPインストール先> ¥UcaEnv¥Sample¥LearnUca¥graphics¥ALRM_FTAC_CG.xaml

システムビューで、HIS0164のWINDOWにウィンドウ種「コントロール（8ループ）」、ウィンドウ名「CG0070」を作成します（ウィンドウ名は自由に命名してください）。

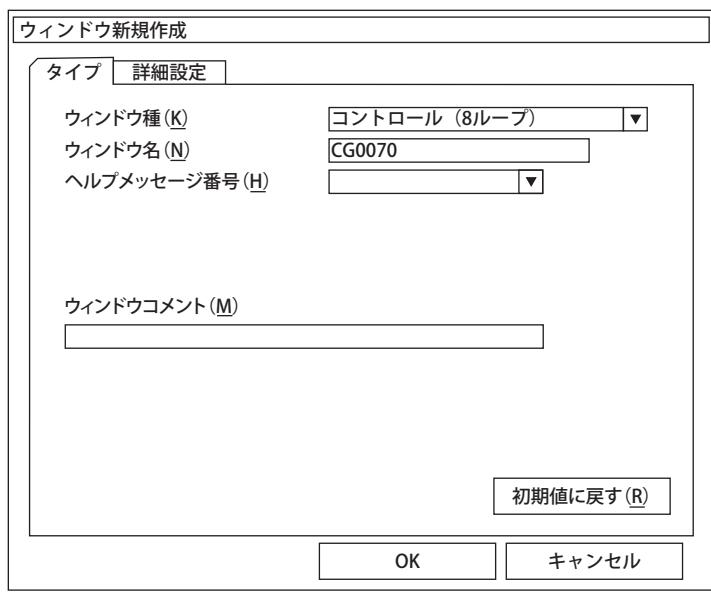
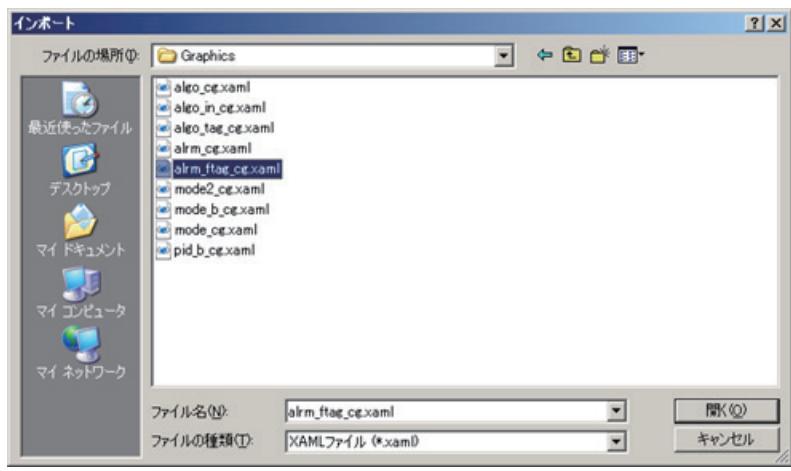


図 コントロールウィンドウの作成

HIS0164のWINDOWにCG0070ができますので、システムビューより CG0070をダブルクリックします。

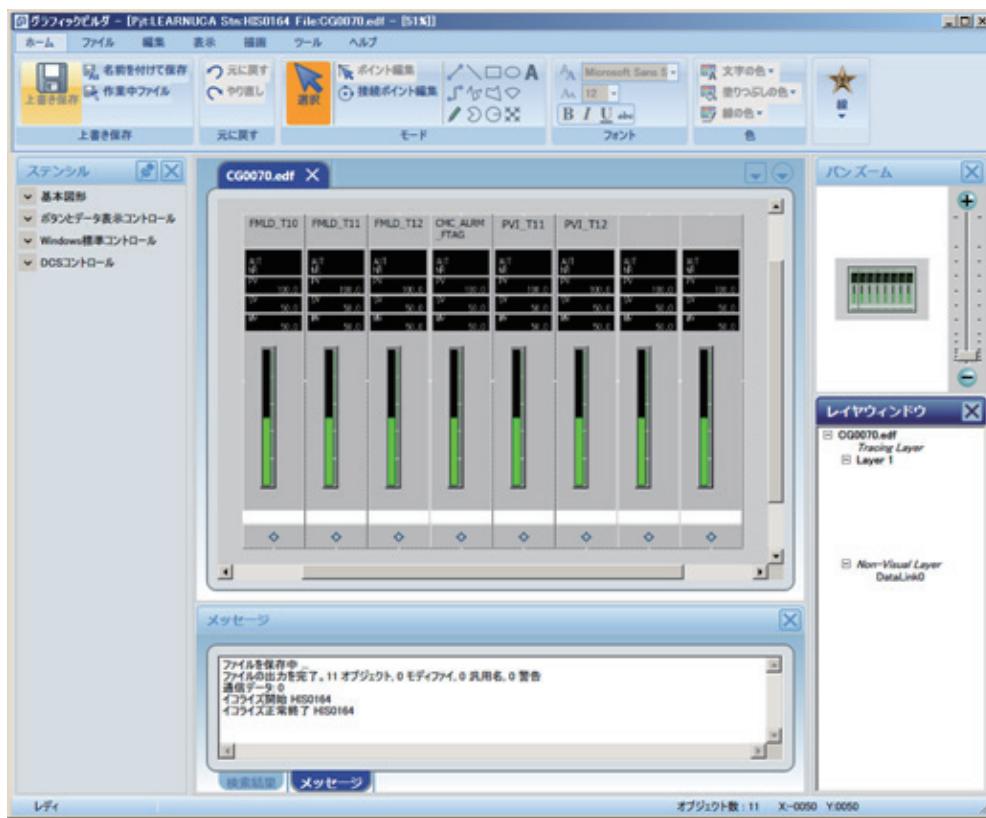
グラフィックビルダの [ファイル] – [外部ファイル] – [インポート] で以下のダイアログを呼び出し、alarm_ftag_cg.xaml を指定し、[開く] ボタンをクリックします。



070203J.ai

図 ファイルを開くダイアログ

グラフィックビルダの [ファイル] – [上書き保存] でファイルを書き込みます。



070204J.ai

図 ファイルの保存

それではプログラムを動かしてみます。まずFCS0101を指定してバーチャルテスト機能を起動します。起動が完了したら、FCS0121(APCS)を指定してバーチャルテスト機能を起動します。FCS0101とFCS0121の両方の起動が完了したらウィンドウCG0070を表示します。



図 ウィンドウの呼び出し

左3つの計器図は、FCS0101の3つの手動操作ブロック(FMLD_T10、FMLD_T11、FMLD_T12)です。その右にFCS0121(APCS)の、1つの汎用演算形ユーザカスタムブロック(CMC_ALRM_FTAK)と2つの指示ブロック(PVI_T11、PVI_T12)が並んでいます。CMC_ALRM_FTAKのデータアイテムPVには、3つの手動操作ブロックのMV値の合計が表示されます。手動操作ブロックのMV値を変更し、CMC_ALRM_FTAKのデータアイテムPVが実効スキャン周期(4秒)ごとに3つのMV値の合計を表示するのを確認してください。以下は、MV値にそれぞれ「10.0」、「25.0」、「35.0」を指定し、その結果CMC_ALRM_FTAKのデータアイテムPVが70.0になっている状態です。

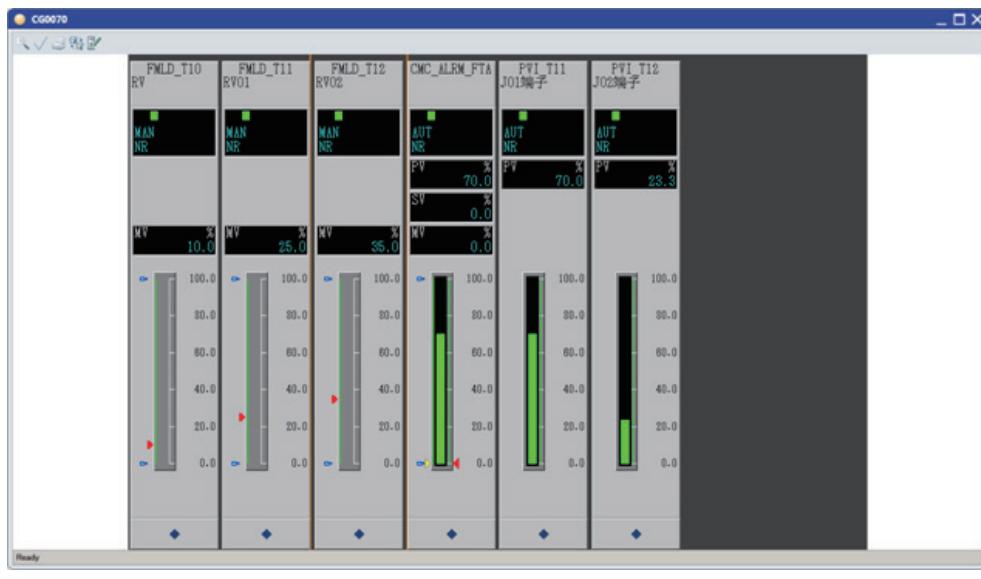


図 CMC_ALRM_FTAKのPV

CMC_ALRM_FTAG は、J01 端子からデータアイテム PV (3 つの入力の合計) のデータを出力します。J02 端子からは 3 つの入力の平均を出力します。J01 端子、J02 端子は、それぞれ指示ブロック PVI_T11、PVI_T12 の PV と結合しています。手動操作ブロックの MV 値を変更すると指示ブロックの PV が変化することを確認してください。

それでは、ユーザカスタムアルゴリズム _SMPL_ALRM_FTAG について説明します。このプログラムは、6.1 節で説明している _SMPL_ALRM の、入力端子からのデータ入力をタグ名を指定したデータ入力に置き換えたものです。ここでは、タグ名を指定した入力のみを説明します。

参照 タグ名を指定した入力以外の部分の詳細については、以下を参照してください。

- 「6.1 多入力 CSTM-C (ブロックモードは AUT と O/S に限定)」
 - 「6.1.1 UcaRWSetPv による PV のデータステータス作成」
 - 「6.1.2 ユーザ定義アラームの発生と復帰」
 - 「6.1.3 J01 端子と J02 端子からのデータ出力」
 - 「6.1.4 機能ブロック初期化処理 (データアイテム P01 と P02 の初期化)」
 - 「6.1.5 機能ブロックデータ設定時特殊処理 (AUT と O/S に限定)」
-

● タグ名を指定した他ステーションデータの入力

タグ名を指定したデータ入力について説明します。alm_ftag.c から alm_calc で検索して測定値決定処理をする alm_calc 関数を見てください。関数の最初に、データ入力処理があります。このプログラムは測定値 PV のデータステータス作成に副入力 RVnn を使用しますので、副入力 RVnn へのデータ設定付きの UcaOtherTagReadToRvnF64S を使用して機能ロックデータを取得します。

```
I32 alm_calc(
    UcaBlockContext bc /* (IN/OUT) : ブロックコンテキスト */
)
{
    F64S pv; /* PV */
    F64S FMLD_T10_mv; /* FMLD_T10.MV */
    F64S FMLD_T11_mv; /* FMLD_T11.MV */
    F64S FMLD_T12_mv; /* FMLD_T12.MV */
    F64S p01; /* P01 */
    F64S p02; /* P02 */
    F64S ave; /* 平均 */
    I32 rtnCode; /* リターンコード */

    /*
     * アラームを検出する場合は、
     * 処理の最初で対象アラームをクリアしておきます。
     */
    rtnCode = UcaAlrmClear(bc, USR_ALRM_UHIG);
    rtnCode = UcaAlrmClear(bc, USR_ALRM_ULOW);

    /* ===== 入力処理 ===== */
    rtnCode = UcaOtherTagReadToRvnF64S(bc, "FMLD_T10", "MV", 0, 0, &FMLD_T10_mv, 0,
    NOOPTION);
    rtnCode = UcaOtherTagReadToRvnF64S(bc, "FMLD_T11", "MV", 0, 0, &FMLD_T11_mv, 1,
    NOOPTION);
    rtnCode = UcaOtherTagReadToRvnF64S(bc, "FMLD_T12", "MV", 0, 0, &FMLD_T12_mv, 2,
    NOOPTION);

    /* ===== 演算処理 ===== */
    /*
     * 演算を実行: PV は、FMLD_T10、FMLD_T11、FMLD_T12 の MV の合計
     * ave は、FMLD_T10、FMLD_T11、FMLD_T12 の MV の平均
     *
     * (演算エラーの検出は省略しデータステータスは常に正常)
     */
    /* 3 入力の合計 */
    pv.value = FMLD_T10_mv.value + FMLD_T11_mv.value + FMLD_T12_mv.value;
    pv.status = 0;

    /* 合計をデータアイテム PV に設定 */
    /* UcaRWReadln を呼び出していくので第 3 引数には、SUCCEED を指定 */
    rtnCode = UcaRWSetPv(bc, &pv, SUCCEED, NOOPTION);
}
```

(続く)

プログラムの最初で、FCS から取得したデータを格納する変数を宣言しています。

```
.....
F64S FMLD_T10_mv;          /* FMLD_T10.MV */
F64S FMLD_T11_mv;          /* FMLD_T11.MV */
F64S FMLD_T12_mv;          /* FMLD_T12.MV */
.....
```

たとえば、F64S 型の変数 FMLD_T10_mv には、FCS の手動操作ブロック FMLD_T10 の MV 値を格納します。C 言語のプログラムでは変数を小文字で記述しますが、「タグ名」は大文字で記述し、データアイテムの部分のみ小文字で記述してあります（これはサンプルで採用した書き方です。タグ名を大文字で記述するか小文字で記述するかはプログラマの自由です。なお、C 言語では大文字と小文字を区別します）。

機能ブロックデータを取得するのは、以下の部分です。このプログラムは、測定値 PV のデータステータス作成に副入力 RVnn を使用しますので、副入力 RVnn へのデータ設定付きの UcaOtherTagReadToRvnF64S を使用して機能ブロックデータを取得します。

```
.....
/* ====== 入力処理 ===== */
rtnCode = UcaOtherTagReadToRvnF64S(bc, "FMLD_T10", "MV", 0, 0, &FMLD_T10_mv, 0,
NOOPTION);
rtnCode = UcaOtherTagReadToRvnF64S(bc, "FMLD_T11", "MV", 0, 0, &FMLD_T11_mv, 1,
NOOPTION);
rtnCode = UcaOtherTagReadToRvnF64S(bc, "FMLD_T12", "MV", 0, 0, &FMLD_T12_mv, 2,
NOOPTION);
.....
```

UcaOtherTagReadToRvnF64S は、引数に指定された機能ブロックデータをステーション間結合ブロック経由で取得し、引数に指定されたデータ格納先の変数とデータアイテム RVnn にデータを設定します。

機能ブロックデータを指定するのは次の部分です。

"FMLD_T10",	"MV",	0,	0,

070207.J.ai

取得したデータを格納する引数指定の記述は、「&FMLD_T10_mv」の部分です。その次の数字は、データを格納する副入力 RVnn の番号を指定します。0 を指定するとデータアイテム RV にデータが格納されます。1 なら RV01、2 なら RV02 です。

3 つの MV 値を取得すると合計を変数 pv.value に設定します。そして、計算結果の pv を引数に指定して UcaRWSetPv を呼び出し、データアイテム PV に測定値を設定します。UcaRWSetPv は引数に指定された測定値（変数 pv）、データアイテム RV と RVnn のデータステータスを元にデータアイテム PV に設定するデータステータスを決定します。

参照 詳細については、以下を参照してください。

[「6.1.1 UcaRWSetPv による PV のデータステータス作成」](#)

重要

UcaRWReadIn により IN 端子からデータを入力する場合には、UcaRWSetPv の第 3 引数に IN 端子の情報として UcaRWReadIn のリターンコードを渡します。このプログラムは IN 端子からのデータ入力をしないので、UcaRWSetPv の第 3 引数に SUCCEED を指定します。

PV のデータステータス作成方法は、ビルダ定義項目「演算入力値異常検出」の指定により異なります。連続制御形ユーザカスタムブロック CMC_ALRM_FTAG には、「全検出形」を指定しています（CSTM-C のデフォルトは補正演算形です）。全検出形を指定すると、UcaRWSetPv はデータアイテム RV と RVnn のデータステータスを PV に反映します。

参照 詳細については、以下を参照してください。

[「6.1.1 UcaRWSetPv による PV のデータステータス作成」](#)



図 演算入力値異常検出の指定

■ BDSET-1LによるADLブロック生成とデータアクセス

UcaOtherTagReadToRvnF64Sなど、ユーザカスタムアルゴリズム作成用ライブラリの他ステーションデータアクセス関数を使用するには、アクセス対象の機能ブロックデータに対応するステーション結合ブロック（ADL ブロック）が必要です。ADL ブロックは、ユーザが BDSET-1L で定義します。このプログラムでは、BDS_T10S に FMLD_T10.MV、FMLD_T11.MV、FMLD_T12.MV を定義しています。BDS_T10S を機能ブロック詳細ビルダで呼び出し、結合タブシートを表示します。

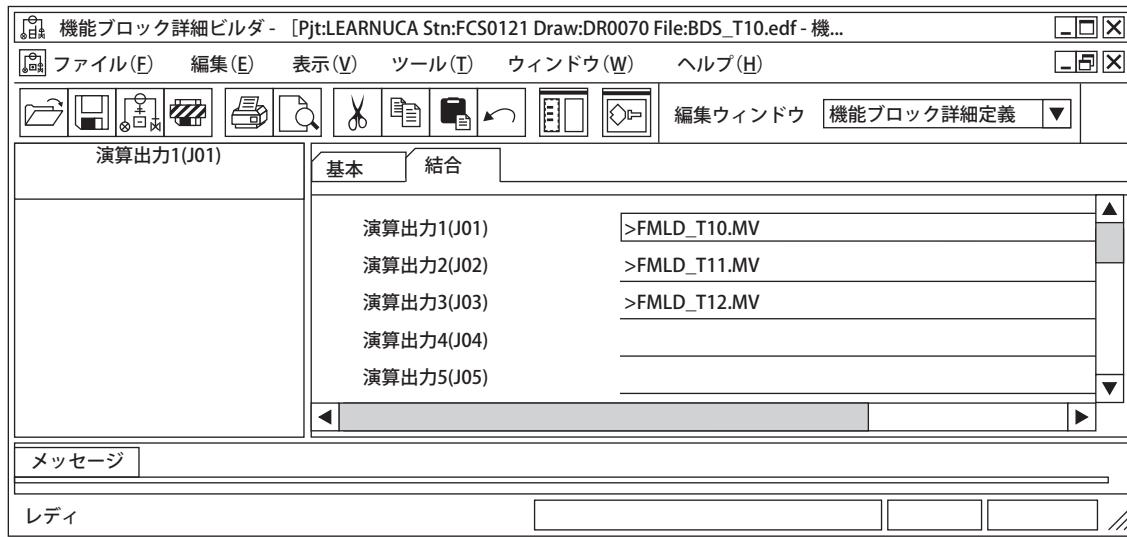


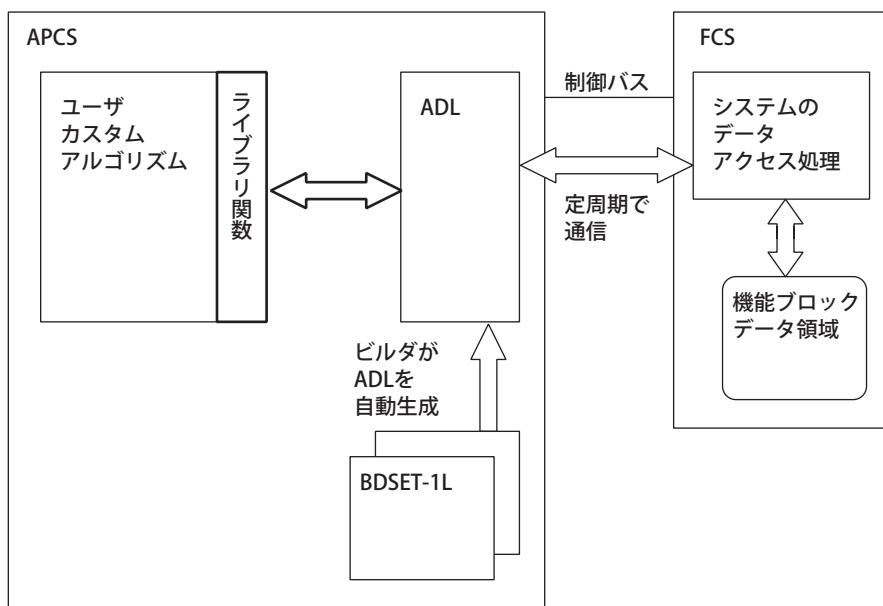
図 結合タブシート

各データの前に「>」が付けてあります。ステーション外のデータであることを示すために先頭の「>」が必要です。ユーザが上図の BDSET-1L を定義すると、システムは FMLDT10.MV、FMLDT11.MV、FMLDT12.MV の 3 つの ADL ブロックを自動的に生成します。

重要 BDSET-1L へのデータ結合の指定では、先頭の「>」とタグ名の間には空白を入れないで詰めて入力してください。

補足 このサンプルプログラムでは、BDSET-1L をユーザカスタムブロックと同じ制御ドローイングに定義しています。実際のアプリケーションでは、BDSET-1L を特定の制御ドローイングにまとめることを推奨します。なお、BDSET-1L はステーション間結合ブロックを生成するためのもので、BDSET-1L 本来の使い方をするのではありませんので、BDSET-1L の実行順を考慮する必要はありません。

以下に APCS 外部のデータに ADL ブロック経由でアクセスする構成を示します。

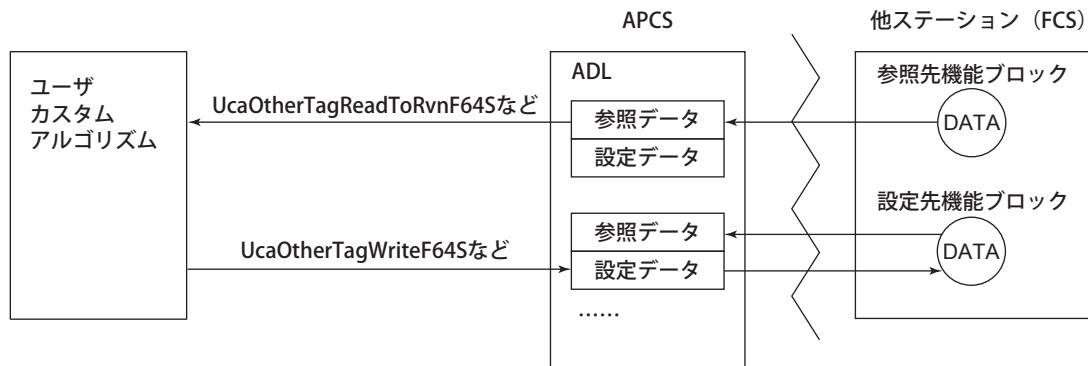


070210J.ai

図 ユーザカスタムアルゴリズムからのデータアクセス

- 制御ステーション間結合を行う ADL ブロックは、BDSET-1L を定義することで自動的に作成されます。
- 太枠の部分が UcaOtherTagReadToRvnF64S など、他ステーションデータアクセス用の関数に相当します。
- 読み込み用の関数 (UcaOtherTagRead***)、書き込み用の関数 (UcaOtherTagWrite***) がありますが、BDSET-1L で ADL ブロックを生成しておけばどの関数も使用可能となります。

他ステーションデータアクセスは、ADL ブロックを中継するデータアクセスとなります。関数 UcaOtherTagRead***/Write*** がアクセスするのは ADL ブロックです。他ステーションの機能ブロックデータにアクセスするのは ADL ブロックです。BDSET-1L の結合は「データ設定結合」ですから、自動生成される ADL ブロックは「データ参照およびデータ設定」となり、ADL ブロックはデータ結合ごとに「参照データ」と「設定データ」の領域を持ちます。



070211J.ai

図 他ステーションデータアクセス

ADL ブロックは、定周期でデータアクセス通信を行います。参照データには、定周期で機能ブロックデータが取り込まれます。また、設定データにデータが設定されると次の通信処理で（一度だけ）機能ブロックに出力されます。したがって、UcaOtherTagRead***/Write*** の動作は、次のようにになります。

- UcaOtherTagReadToRvnF64S などは、ADL ブロックの参照データ（もっとも最近に読み込んだデータ）を取得します。
- UcaOtherTagWriteF64S などの出力先は ADL ブロックの「設定データ」に保持されます。当該の ADL ブロックの次回通信時に、設定データに保持されている値が実際の機能ブロックデータに（一度だけ）出力されます。

したがって、UcaOtherTagReadToRvnF64S などで取得できるデータは関数を実行する「少し前」のデータであり、UcaOtherTagWriteF64S などによる出力が実際に機能ブロックに 出力されるのは関数を実行してから「少し後」となります。

参照

- 制御ステーション間結合の詳細については、以下を参照してください。
[機能ブロック共通機能リファレンス \(IM 33J15A20-01JA\)](#) 「2.4 制御ステーション間結合」
- ステーション間結合ブロックの詳細については、以下を参照してください。
[機能ブロッククリファレンス Vol.2 \(IM 33J15A31-01JA\)](#) 「1.46 ステーション間結合ブロック (ADL)」
- ステーション間結合ブロックの通信周期の詳細については、以下を参照してください。
[APCS ユーザカスタムブロック \(IM 33J15U20-01JA\)](#)

7.3 自ステーションデータ（副入力RVnnを使用）

自ステーション内の機能ブロックデータをタグ名を指定して入力する方法を説明します。この節では、5.2節の「多入力多出力CSTM-C」のデータ入力処理を、入力端子からタグ名指定に置き換えます。

タグ名を指定して他ステーションの機能ブロックデータにアクセスする場合は、BDSET-1Lにデータを定義しステーション間結合ブロックを生成する必要があります。タグ名を指定した自ステーションデータアクセスでは、BDSET-1Lのデータ定義は不要です。自ステーションデータアクセス用の関数を呼び出すだけでデータアクセスが可能であり、関数を呼び出す以外の追加作業はありません。

サンプルプログラムが用意してありますので、動かしてみます。サンプルソリューション _SMPL_ALGO_TAG の Release 版をビルドし、ユーザカスタムアルゴリズムを登録します。ソリューションは 1 章の準備作業により以下の作業フォルダにコピーされています。

<ドライブ名> ¥UcaWork¥UcaSamples¥_SMPL_ALGO_TAG

Visual Studio を起動し、_SMPL_ALGO_TAG¥_SMPL_ALGO_TAG.sln を開きます。[ビルド] メニューの [リビルド] で _SMPL_ALGO_TAG の Release 版をリビルドし、ユーザカスタムアルゴリズムを登録します。

次にサンプルの制御ドローイングをインポートします。サンプルの制御ドローイングを定義したテキストファイル ALGO_TAG.txt が CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥drawings¥ALGO_TAG.txt

FCS0121 (APCS) の制御ドローイングの DR0071 (空いている制御ドローイングならどれでも構いません) を指定して制御ドローイングビルダを起動します。[ファイル] メニューの [外部ファイル] – [インポート] を指定します。インポートダイアログで上記の ALGO_TAG.txt を指定し、[開く] ボタンをクリックすると次図の制御ドローイングが取り込まれますので、[ファイル] – [上書き保存] で制御ドローイングを書き込みます。

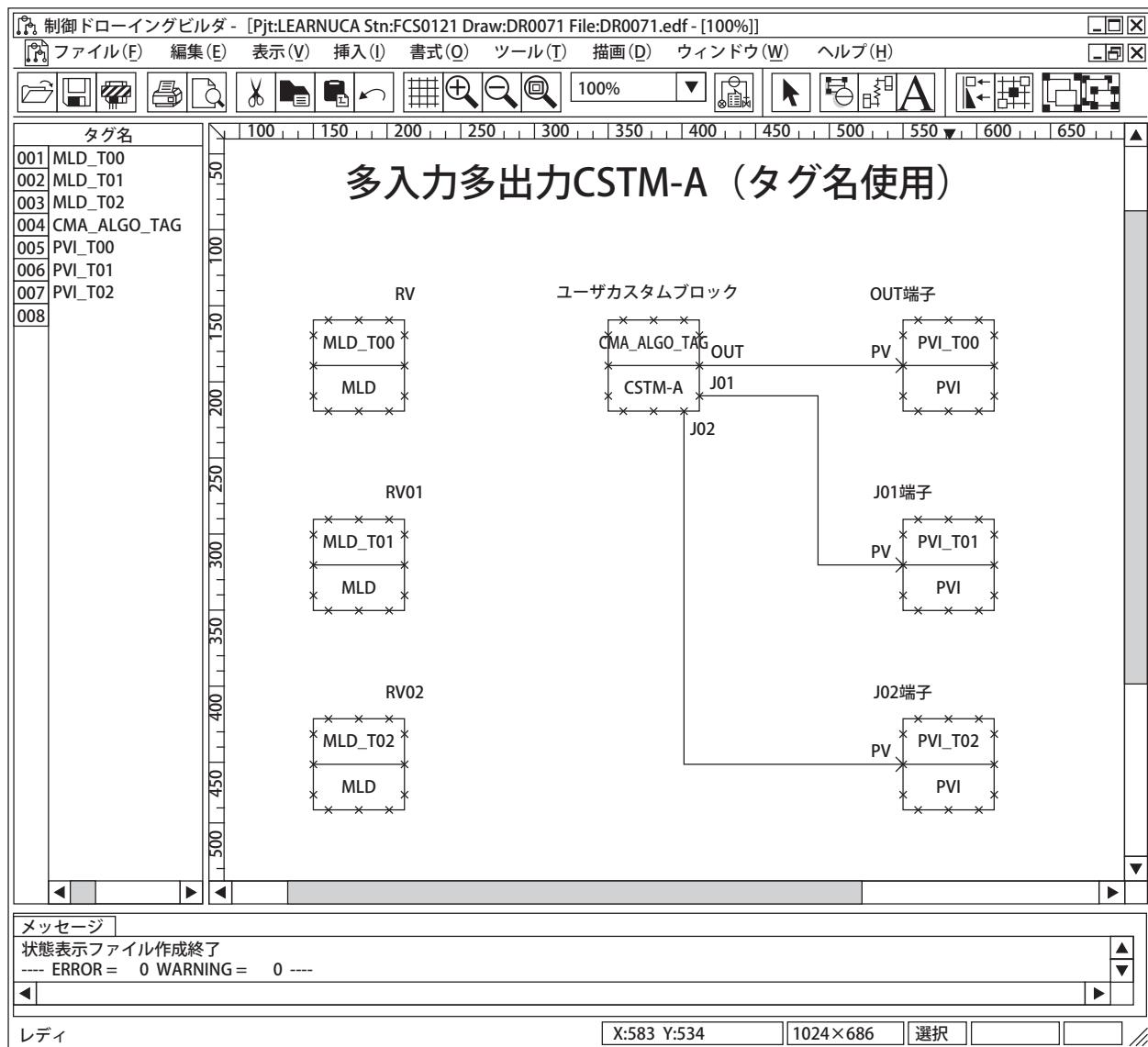


図 制御ドローイングのインポート

この制御ドローイングでは、汎用演算形カスタムブロック (CSTM-A) が OUT 端子、J01 端子、J02 端子から出力するデータを、3 つの指示ブロック (PVI) が入力しています。ユーザカスタムアルゴリズムは、3 つの手動操作ブロック (MLD) の MV 値をタグ名を指定して入力しています。このため、手動操作ブロックから汎用演算形ユーザカスタムブロックへの配線はありません。CSTM-A の演算出力値を保持するデータアイテム CPV、CPV01、CPV02 には、それぞれ次のような計算結果を格納します。このプログラムでは、データアイテム P01 に演算のパラメータとして倍率を格納しています。

$$CPV = P01 \times (IN \text{ 端子入力データ} + Q01 \text{ 端子入力データ} + Q02 \text{ 端子入力データ})$$

$$\begin{aligned} CPV01 &= IN \text{ 端子入力データ}、Q01 \text{ 端子入力データ}、Q02 \text{ 端子入力データ} の最大値 \\ CPV02 &= IN \text{ 端子入力データ}、Q01 \text{ 端子入力データ}、Q02 \text{ 端子入力データ} の最小値 \end{aligned}$$

動作を確認するためにコントロール（8ループ）のウィンドウが用意してありますので、HIS0164 の CG0071 に取り込んでおきます。テキストファイルが CENTUM VP インストール先の以下にあります。

<CENTUM VP インストール先> ¥UcaEnv¥Sample¥LearnUca¥graphics¥ALGO_TAG_CG.sva

システムビューで、HIS0164 の WINDOW にウィンドウ種「コントロール（8ループ）」、ウィンドウ名「CG0071」を作成します（ウィンドウ名は自由に命名してください）。

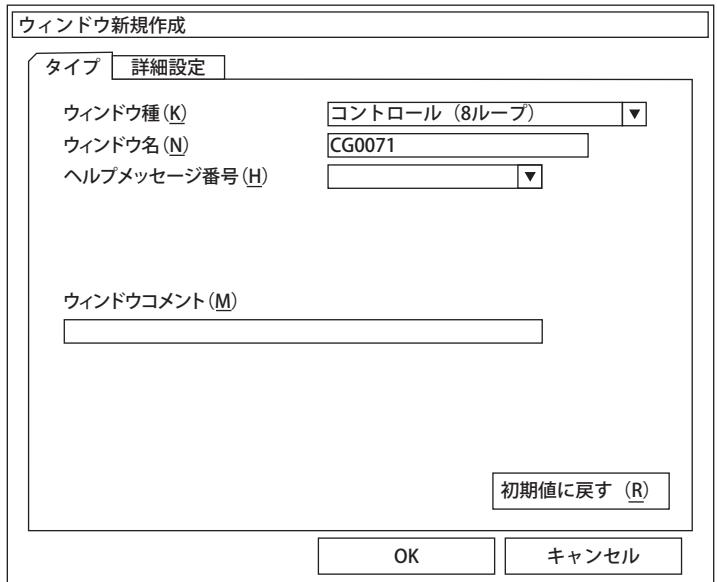
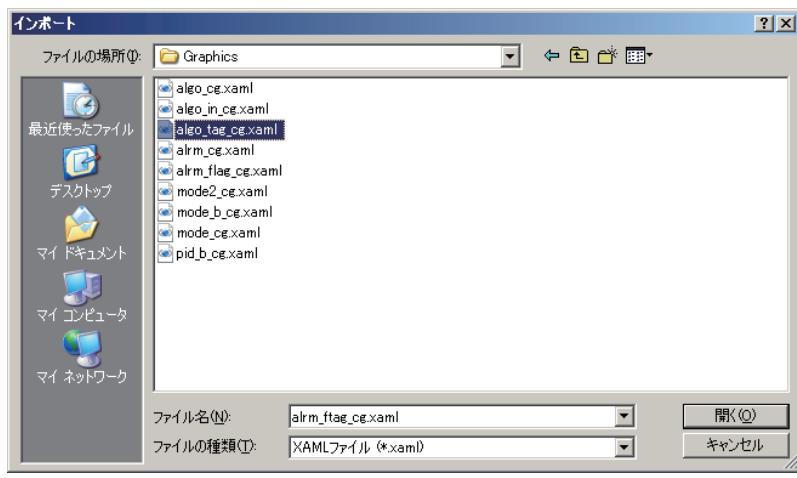


図 コントロールウィンドウの作成

HIS0164 の WINDOW に CG0071 ができますので、システムビューより CG0071 をダブルクリックします。

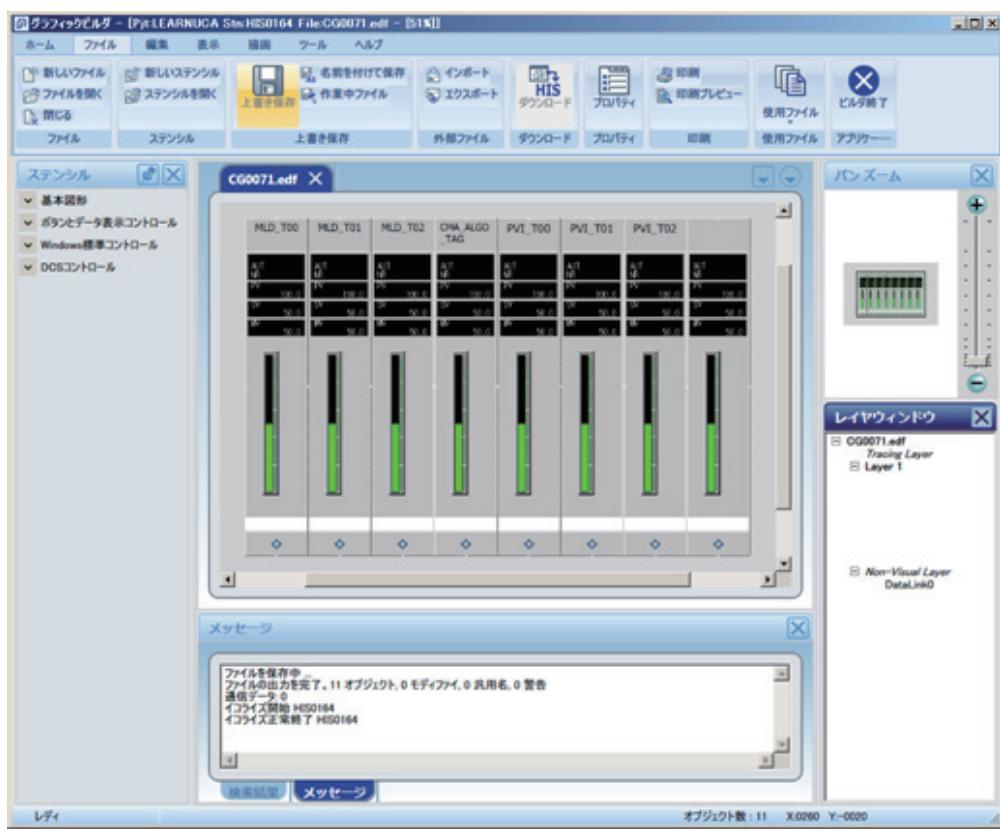
グラフィックビルダの [ファイル] – [外部ファイル] – [インポート] で以下のダイアログを呼び出し、algo_tag_cg.xaml を指定し、[開く] ボタンをクリックします。



070303J.ai

図 ファイルを開くダイアログ

グラフィックビルダの [ファイル] – [上書き保存] でファイルを書き込みます。



070304J.ai

図 ファイルの保存

それでは、プログラムを動かしてみます。FIC0121 (APCS) を指定してバーチャルテストを起動します。起動が完了したらウィンドウ CG0071 を表示します。



図 ウィンドウの呼び出し

3つの手動操作ブロック (MLD_T00、MLD_T01、MLD_T02) と1つの汎用演算形ユーザカスタムブロック (CMA_ALGO_TAG) が並んでいます。CMA_ALGO_TAG のデータアイテム CPV には、3つの手動操作ブロックの MV 値の合計が表示されます。手動操作ブロックの MV 値を変更し、CMA_ALGO_TAG のデータアイテム CPV が実効スキャン周期 (4 秒) ごとに3つのMV値の合計を表示するのを確認してください。以下は MV 値にそれぞれ 5.0、10.0、15.0 を指定し、その結果 CMA_ALGO_TAG のデータアイテム CPV が 30.0 になっている状態です。

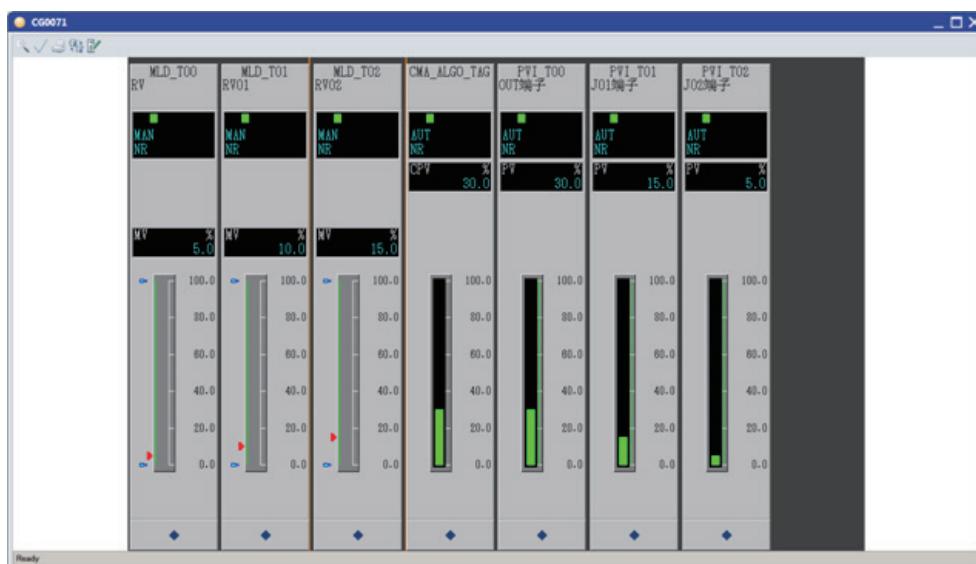


図 CMA_ALGO_TAGのCPV

CMA_ALGO_TAG の OUT 端子からはデータアイテム CPV (3 つの入力の合計) のデータを出力します。J01 端子からは 3 つの入力の最大値を、J02 端子からは 3 つの入力の最小値をそれぞれ出力します。OUT 端子、J01 端子、J02 端子は、それぞれ指示ブロック PVI_T00、PVI_T01、PVI_T02 の PV と結合しています。手動操作ブロックの MV 値を変更すると、指示ブロックの PV が変化することを確認してください。

それでは、ユーザカスタムアルゴリズム _SMPL_ALGO_TAG について説明します。このプログラムは、6.1 節で説明している _SMPL_ALGO の入力端子からのデータ入力を、タグ名を指定したデータ入力に置き換えたものです。ここでは、タグ名を指定した入力のみを説明します。

参照 タグ名を指定した入力以外の部分の詳細については、以下を参照してください。

「5.2 多入力多出力 CSTM-A」

「5.2.1 入力処理と UcaFpuExpCheck による演算エラーの検出」

「5.2.2 演算異常 ERRC アラームの発生と復帰」

「5.2.3 出力端子からのデータ出力」

「5.2.4 チューニングパラメータより弱い初期値の設定方法」

■ タグ名を指定した自ステーションデータの入力

タグ名を指定したデータ入力について説明します。algo_tag.c から algo_calc で検索して演算結果を計算する algo_calc 関数を見てください。関数の最初にデータ入力処理があります。このプログラムは演算結果 CPV のデータステータス作成に副入力 RVnn を使用しますので、副入力 RVnn へのデータ設定付きの UcaTagReadToRvnF64S を使用して機能ブロックデータを取得します。

```
I32 algo_calc(
    UcaBlockContext bc                /* (IN/OUT) : ブロックコンテキスト */
)
{
    F64S MLD_T00_mv;                 /* MLD_T00.MV */
    F64S MLD_T01_mv;                 /* MLD_T01.MV */
    F64S MLD_T02_mv;                 /* MLD_T02.MV */
    F64S cpv;                       /* CPV */
    F64S cpv01;                     /* CPV01 */
    F64S cpv02;                     /* CPV02 */
    F64 max;                        /* 最大値 */
    F64 min;                        /* 最小値 */
    F64S p01;                       /* P01 */
    U32 expFlag;                    /* 浮動小数演算エラー検出フラグ */
    I32 rtnCode;                     /* リターンコード */

    /*
     * アラームを検出する場合は、
     * 処理の最初で対象アラーム (ERRC) をクリアしておきます。
     */
    rtnCode = UcaAlrmClear(bc, UCAMASK_ALRM_ERRC);

    /* ===== 入力処理 ===== */
    rtnCode = UcaTagReadToRvnF64S(bc, "MLD_T00", "MV", 0, 0, &MLD_T00_mv, 0, NOOPTION);
    rtnCode = UcaTagReadToRvnF64S(bc, "MLD_T01", "MV", 0, 0, &MLD_T01_mv, 1, NOOPTION);
    rtnCode = UcaTagReadToRvnF64S(bc, "MLD_T02", "MV", 0, 0, &MLD_T02_mv, 2, NOOPTION);
}
```

(続く)

プログラムの最初で、FCS から取得したデータを格納する変数を宣言しています。

```
.....
F64S MLD_T00_mv;          /* MLD_T00.MV */
F64S MLD_T01_mv;          /* MLD_T01.MV */
F64S MLD_T02_mv;          /* MLD_T02.MV */
.....
```

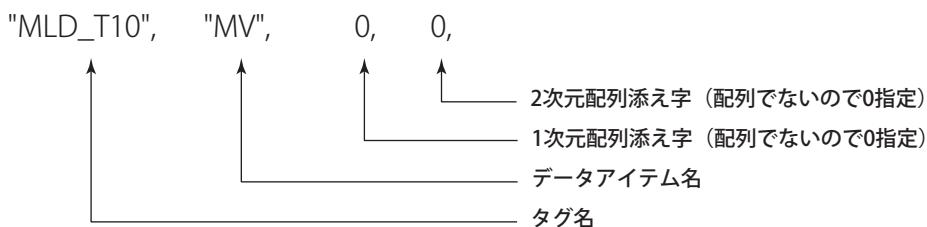
たとえば、F64S 型の変数 MLD_T10_mv には、FCS の手動操作ブロック MLD_T10 の MV 値を格納します。C 言語のプログラムでは、通常、変数を小文字で記述しますが、「タグ名」は大文字で記述し、データアイテムの部分のみ小文字で記述してあります（これはサンプルで採用した書き方です。タグ名を大文字で記述するか小文字で記述するかはプログラマの自由です。なお、C 言語では大文字と小文字を区別します）。

機能ブロックデータを取得するのは、以下の部分です。このプログラムは演算結果 CPV のデータステータス作成に副入力 RVnn を使用しますので、副入力 RVnn へのデータ設定付きの UcaOtherTagReadToRvnF64S を使用して機能ブロックデータを取得します。

```
.....
rtnCode = UcaTagReadToRvnF64S(bc, "MLD_T00", "MV", 0, 0, &MLD_T00_mv, 0, NOOPTION);
rtnCode = UcaTagReadToRvnF64S(bc, "MLD_T01", "MV", 0, 0, &MLD_T01_mv, 1, NOOPTION);
rtnCode = UcaTagReadToRvnF64S(bc, "MLD_T02", "MV", 0, 0, &MLD_T02_mv, 2, NOOPTION);
.....
```

UcaTagReadToRvnF64S は引数に指定された機能ブロックデータを取得し、引数に指定されたデータ格納先の変数とデータアイテム RVnn にデータを設定します。

機能ブロックデータを指定するのは次の部分です。



070307J.ai

取得したデータを格納する引数指定の記述は、「&MLD_T10_mv」の部分です。その次の数字は、データを格納する副入力 RVnn の番号を指定します。0 を指定するとデータアイテム RV にデータが格納されます。1 なら RV01、2 なら RV02 です。

algo_calc は、3 つの MV 値を取得すると、3 つの MV 値の合計にデータアイテム P01 に保持している倍率を掛けた数を変数 cpv.value に設定します。このとき、UcaFpuExpCheck で演算エラーの検出と演算異常を示す ERRC アラーム発生処理をしています。

参照

- ・ 演算エラーの検出の詳細については、以下を参照してください。
「[5.2.1 入力処理と UcaFpuExpCheck による演算エラーの検出](#)」
- ・ ERRC アラーム発生処理の詳細については、以下を参照してください。
「[5.2.2 演算異常 ERRC アラームの発生と復帰](#)」

演算結果 cpv ができると変数 cpv を引数に指定して UcaRWSetCpv を呼び出し、データアイテム CPV に演算結果を設定します。UcaRWSetCpv は引数に指定された演算結果(変数 cpv)、データアイテム RV と RVnn のデータステータスをもとに、データアイテム CPV に設定するデータステータスを決定します。

```
.....
/* 演算結果をデータアイテム CPV に出力 */
/* UcaRWReadIn を呼び出していないので第3引数に SUCCEED を指定 */
rtnCode = UcaRWSetCpv(bc, &cpv, SUCCEED, NOOPTION);
.....
```

重要

UcaRWReadIn により IN 端子からデータを入力する場合には、UcaRWSetCpv の第 3 引数に IN 端子の情報として UcaRWReadIn のリターンコードを渡します。このプログラムは IN 端子からのデータ入力をしないので、UcaRWSetCpv の第 3 引数に SUCCEED を指定します。

CPV のデータステータス作成方法は、ビルダ定義項目「演算入力値異常検出」の指定により異なります。連続制御形ユーザカスタムブロック CMC_ALGO_TAG には、「全検出形」を指定しています (CSTM-A のデフォルトは、非検出形です)。全検出形を指定すると、UcaRWSetCpv はデータアイテム RV と RVnn のデータステータスを CPV に反映します。

参照

CPV のデータステータス作成の詳細については、以下を参照してください。

[「5.1 多入力 CSTM-A」の「■ UcaRWSetCpv による CPV のデータステータス作成」](#)

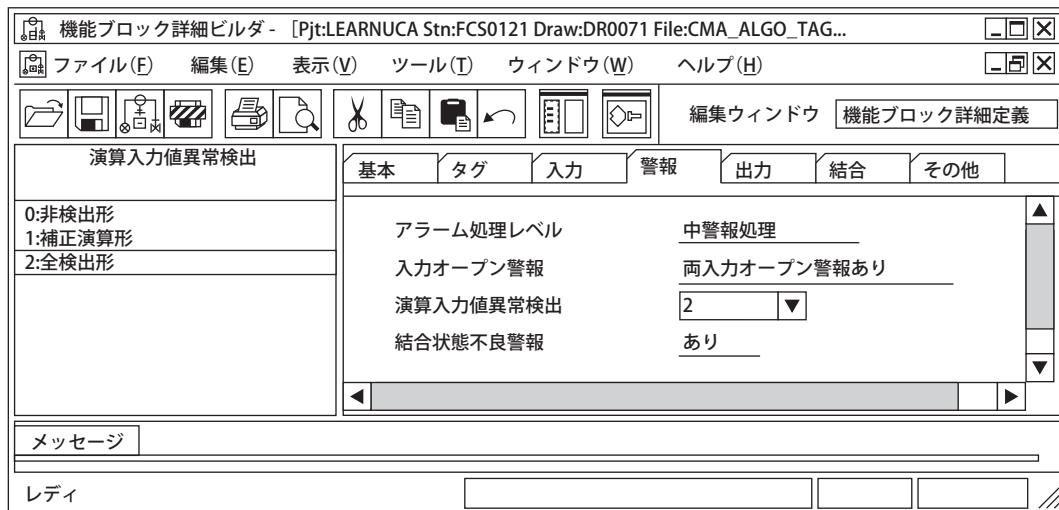


図 演算入力値異常検出の指定

070308J.ai

Appendix. 機能ごとの索引

ユーザカスタムアルゴリズムの機能ごとの索引を以下に示します。

表 機能別索引 (1/2)

機能	章	説明	機能の有無	
			CSTM-C	CSTM-A
データ アクセス	IN 端子	5.1 6.1.1	UcaRWRead による IN 端子からのデータ入力	○ ○
	Qnn 端子	5.1 6.1.1	UcaRWRead による Qnn 端子からのデータ入力	○ ○
	OUT 端子	5.2.3	UcaRWWWriteCpvToOutSub による OUT 端子からのデータ出力	○ ○
		6.2.7	UcaRWWWriteMvToOutSub による OUT 端子からのデータ出力	○ ○
	Jnn 端子	5.2.3 6.1.3	UcaRWWWrite による Jnn 端子からのデータ出力	○ ○
		6.1.3	UcaRWWWritePvToJnSub による PV 値の Jnn 端子からの出力	○ ○
	タグ名指定 (他ステーション)	7.2	タグ名を指定した他ステーションのデータアクセス	○ ○
	タグ名指定 (自ステーション)	7.1	タグ名を指定した自ステーション内データのアクセス	○ ○
測定値	PV のデータステータス作成	6.1.1	UcaRWSetPv による PV のデータステータス作成	○ ○
演算出力値	CPV のデータステータス作成	5.1	UcaRWSetCpv による CPV のデータステータス作成	○ ○
アラーム ステータス	ユーザ定義アラームの定義	4.3.2 4.3.3	ビルダによるアラームステータスの定義と、ユーザ定義インクルードファイルのラベル作成	○ ○
	ユーザ定義アラームを検出する関数	4.3.2	「表 ユーザカスタムブロックのアラームステータスのデフォルト」とアラームを検出する関数の一覧	○ ○
	HI、LO、HH、LL アラームの検出	3.6.2 6.1.2	UcaRWSetPV による HI、LO、HH、LL アラームの検出	○ ○
	ERRC アラームの検出	5.2.2	ユーザプログラムによる ERRC (演算異常) アラームの発生と復帰	○ ○
	ユーザ定義アラームの検出	6.1.2	ユーザプログラムによるユーザ定義アラームの発生と復帰	○ ○

○： 使用できる

空欄： 使用できない

表 機能別索引 (2/2)

機能		章	説明	機能の有無	
				CSTM-C	CSTM-A
ロックステータス	ユーザ定義 ブロックステータスの定義	4.3.1 4.3.3	ビルダによるブロックステータスの定義と、ユーザ定義インクルードファイルのラベル作成	○	○
	ブロックステータスの操作	4.3.4	UcaBstsSetExclusive を使ったブロックステータス操作	○	○
ロックモード	ブロックステータスを AUT に固定	3.6.1 6.1.5	機能ブロックデータ設定時特殊処理で、ブロックモードを AUT と O/S に限定	○	
	ブロックステータス遷移	6.2	UcaCtrlHandler を使った、ブロックモード遷移を伴うユーザカスタムアルゴリズムのプログラミング	○	
制御演算	制御ホールド	6.3.1	制御ホールドの処理	○	
	入出力補償	6.3.2	入力補償と出力補償	○	
	制御初期化	6.3.3	自動運転に入るときの初期化	○	
	制御動作方向	6.3.5	「逆動作」と「正動作」	○	
	制御出力動作	6.3.6	「速度形」と「位置形」	○	
	リセットリミット	6.3.7	リセットリミット機能の処理	○	
	不感帯動作	6.3.8	不感帯動作の処理	○	
	PID 制御演算	6.4	PID 調節ブロックと同じ動作をする CSTM-C	○	
制御周期	制御周期	6.4.2	実効スキャン周期と制御周期の処理	○	
	制御周期	6.6	制御周期の指定が有効な CSTM-C	○	
カスケード結合	カスケード結合	6.6	カスケード結合と実効スキャン周期、制御周期の関係	○	
ユーザカスタムアルゴリズム作成用ライブラリ	関数一覧	3.8	ユーザカスタムアルゴリズム作成用ライブラリの関数一覧。CSTM-C/CSTM-A における使用可否などを表にまとめてあります。	○	○
	検出できるアラーム一覧	4.3.2	ユーザカスタムアルゴリズム作成用ライブラリの関数が検出できるアラームを、「表ユーザカスタムブロックのアラームステータスのデフォルト」にまとめてあります。	○	
	ビルダ定義項目との関連一覧	4.5	ユーザカスタムアルゴリズム作成用ライブラリの関数とビルダ定義項目の関係	○	○
Windows	使用可能な Windows 関数一覧	4.8	ユーザカスタムアルゴリズムから呼び出し可能な、Windows のライブラリ関数の一覧		
サンプル	サンプル一覧	4.6	本書で説明しているユーザカスタムアルゴリズムのサンプル一覧	○	

○： 使用できる

空欄： 使用できない

APCS ユーザカスタムブロック プログラミングガイド

IM 33J15U21-01JA 2 版

索引

C

C 言語 2-1

P

PID 調節ブロック 6-101

R

RVnn 7-4, 7-17

W

Windows のライブラリ 4-42

ア

アラームステータス 4-5

カ

カスケードクローズ 6-143

カスケード結合 6-133

キ

共用 4-37

シ

実効スキャン周期 6-133

終了処理 1-23, 3-11

初期化処理 1-25, 3-5

セ

制御演算関数 6-63

テ

定期期処理 1-15, 3-21

データ設定時特殊処理 1-26, 3-30

ハ

汎用演算形ユーザカスタムブロック 5-1

ヒ

ビルダ定義項目 4-27

フ

フォルダ構成 4-2

ブロック形 4-25

ブロックステータス 4-5

ブロックモード遷移 6-29

メ

命名規則 4-4

ユ

ユーザカスタムアルゴリズム 3-1

ユーザ定義関数の戻り値 3-43

レ

連続制御形ユーザカスタムブロック 6-1

ワ

ワンショット起動 6-143

ワンショット起動処理 1-31, 3-23

Blank Page

改訂情報

資料名称 : APCS ユーザカスタムブロックプログラミングガイド

資料番号 : IM 33J15U21-01JA

2019年8月／2版／R6.07以降

前書き 「■商標」の記述変更

2018年8月／初版／R6.06

新規発行

■ お問い合わせについて

問い合わせ : <http://www.yokogawa.co.jp/dcs> より、お問い合わせ
フォームをご利用ください。

■ 著作者 横河電機株式会社

■ 発行者 横河電機株式会社

〒 180-8750 東京都武蔵野市中町 2-9-32

Blank Page
