Motortronics®

SV-NET CONTROLLER

プログラミングマニュアル

C言語編

TAMAGAWA SEIKI CO.,LTD

■ はじめに

このたびは、SV-NET コントローラをお買い上げいただき、ありがとうございます。
SV-NET コントローラは、多摩川精機モーションネットワーク【SV-NET】と【EtherCAT】に対応したモーションコントローラです。弊社の SV-NET 対応ドライバと組合せて使用する事により、柔軟なモーションコントロールシステムを構築可能です。

本マニュアルでは、Motion Designer（SV-NET コントローラ専用プログラミングツール）に実装された、C 言語の機能について説明します。C 言語モーションコントローラが、お客様のモーションプログラム作成を支援致します。C 言語モーションコントローラの使い方、機能、手法を十分理解いただき御社システムの制御にご活用下さい。

■ 略 称

本書では以下の略称を使用します。

略 称	内 容
SVC またはコントローラ	SV-NET コントローラ
SVD またはドライバ	SV-NET 対応ドライバ
サーボモータまたモータ	AC サーボモータ
SVCC	SV-NET Controller Compact
SVCE	SV-NET Controller Ether
TmasM	多摩川モーションアセンブラ言語
TMc	多摩川モーション C 言語
TMoS またはモーション OS	多摩川モーション OS
C 言語モーションコントローラ	C 言語対応 SV-NET コントローラ
プログラム	SV-NET Controller モーションプログラム

■ 改訂履歴

版	日付	内容
1.0	2013/05/23	新規作成
2.0	2013/08/07	予約語追加。（M_Code、NC_Call、Mov*WL 関連のコマンド、EtherCAT 関連のコマンド） 2.3 項演算子の注記追加。

■ 安全に関するご注意

- 保証内容
  - 保証期間

    出荷後 1 年以内に弊社へご連絡またはご返却頂いた場合、修理、または代品の納入を限度とさせて頂きます。
  - 保証範囲

    保障期間内であっても、本書記載事項から逸脱した使用、保存状況による品質低下につきまして、

    弊社はその責を負いかねますのであらかじめご了承願います。
  - 製品仕様書・マニュアル等に記載されている以外の条件・環境・取扱いでご使用になられた場合。
  - 弊社以外による改造・修理をされた場合。
  - 製品本来の使い方以外でご使用になられた場合。
  - 弊社出荷当時の技術の水準では予見出来なかった場合。
  - 保証の制限
  - 弊社製品に起因して生じた他への損害に関して、弊社では責任を負いません。
  - 弊社担当者以外の者が作成したプログラムにより生じた結果について、弊社は責任を負いません。
- 使用条件
  - 本製品は一般工業製品向けに設計・製作されております、人命に危険を及ぼすような状況下で使用される機器、システムに使用する目的ではご使用になれません。
  - 本製品は極端に高い信頼性を要求される様な分野への使用を前提としておりません。下記用途で使用される場合、仕様書・マニュアル等を良く確認のうえ、弊社営業担当者までご相談下さい。また万一故障があっても、危険を最小にする安全回路などの安全対策を必ず実施してください。
  - 原子力制御設備、宇宙、鉄道、航空、車両設備、医用装置、安全装置、焼却設備。
  - 人命や財産に危険が及ぶシステム・機械・装置。
  - ガス・水道・電力供給システムや 24 時間連続運転システムなどの高い信頼性が必要な設備。
  - 屋外での使用やマニュアル等に記載の無い条件での使用。
  - その他上記に準ずる、高度な信頼性が必要とされる用途。
  - 弊社は常に品質及び信頼性について向上させる様努めております。しかし一般的に本製品はある確率で故障致します。ご使用に当たっては、本製品の作動不良等で考えられる連鎖又は波及の状況を考慮されて、事故回避の為に多重の安全策を設ける様にして下さい。
  - マニュアル等に記載されているプログラム例やアプリケーション事例は参考用です。ご使用になられる場合には、対象のシステム、機器、装置等の機能や安全性をご確認の上ご使用下さい。

- **仕様の変更**

製品仕様書、マニュアル、カタログなどに記載の内容は、性能改善、仕様拡張、付属品追加等の理由により、必要に応じて変更する場合があります。最新の内容については、弊社営業担当者までご相談下さい。

- **バージョンアップ**

本製品は、性能改善、仕様拡張の為に、本体ソフトウェアをバージョンアップする場合があります。ご使用にあたっては、最新のバージョンを確認するようお願い致します。またバージョンアップする際には、弊社営業担当者までご相談下さい。

- **サービスの範囲**

本製品の価格には、技術派遣などのサービス費用は含まれておりません。必要であれば、弊社営業担当者までご相談下さい。

- **技術者の派遣サービス**

弊社ではお客様の装置立ち上げの為、若干の費用を頂く事により技術者の派遣サービスを提供しております。主な支援作業の内容としては下記のとおりです。

  - **サーボゲインの調整**
  - SV-NET **コントローラの運転プログラム作成**
  - **サーボゲイン調整方法説明**
  - Motion Designer **の取扱説明**

装置の初期立ち上げや、新規システムの導入には時間がかかります。特に新規にシステム導入または変更された場合には、弊社の技術者派遣サービスをご利用頂く事をおすすめ致します。作業費用および作業内容についてご不明な点がございましたら、弊社営業担当者までご連絡下さい。

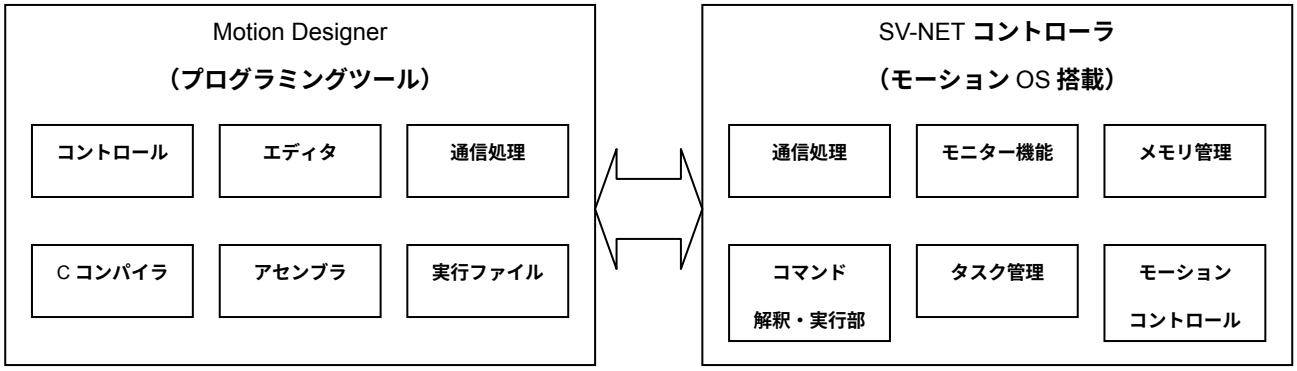
■ 目 次

1．概 要

1．1 C 言語モーションコントローラの概要

C 言語モーションコントローラは、SV-NET コントローラ（以降コントローラと記載）と専用プログラミングツール Motion Designer に実装された C 言語対応機能の総称です。Motion Designer に実装された専用のエディタを使用し、C 言語によりコントローラ専用のモーションプログラムを作成します。構文ルールは C 言語に準拠し、プログラムの作成ミスを防ぐ為に煩雑な仕様は制限しています。本マニュアルでは、C 言語モーションコントローラの機能と使い方、Motion Designer 上でのプログラム作成方法について説明します。

C 言語モーションコントローラの機能ブロック図を示します。



1．2 Motion Designer の特徴

Motion Designer を使用する事により、モーションプログラムの作成を効率よく行う事ができます。プログラムを作成する為に準備された強力なエディタや、コントローラ・ドライバの状態を管理する為の各種コントロール、パラメータの保存・転送機能等がプログラム作成を支援致します。

- 強力なエディタを搭載
  （UNDO／REDO 機能、検索・置換機能、定義ラベルジャンプ機能、自動候補表示、クイック HELP 機能…etc）
- 強力なデバッグ機能
  （グローバル変数監視、クイックモニター機能、タスク毎のステップ実行、ブレーク停止機能）
- プログラム作成を支援する専用コントロール
  （ジョグコントロール、サーボモニターコントロール、デジタルオシロ、タスクコントロール）
- プロジェクト・パラメータ管理
  （プロジェクトファイルの管理、パラメータの保存・転送機能、プログラムテキストの保存・転送・読み出し機能）

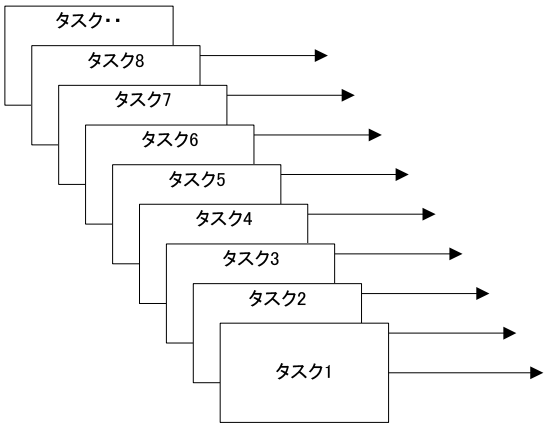
１．３　モーションコントロールの特徴

コントローラに搭載されたモーション OS は様々な機械の要求に応えるために、本体実行部に多くのコマンドを実装しています。C 言語から実行部のコマンド呼び出すことにより、様々な状況に対応する事が可能となります。

- ノーウェイト型の動作コマンド。
- フィードスロープコマンドやベル型コマンドによる細分化された動作コマンド。
- 細分化された動作コマンドによる、新しい加減速パターンの作成。
- サーボ情報等をプログラムで使用するモニター変数のサポート。
- 複数のプログラムを同時に実行可能なタスク管理機能。
- ネットワーク変数による外部機器とのメモリ共有。　　.....etc

１．４　プログラムとタスク

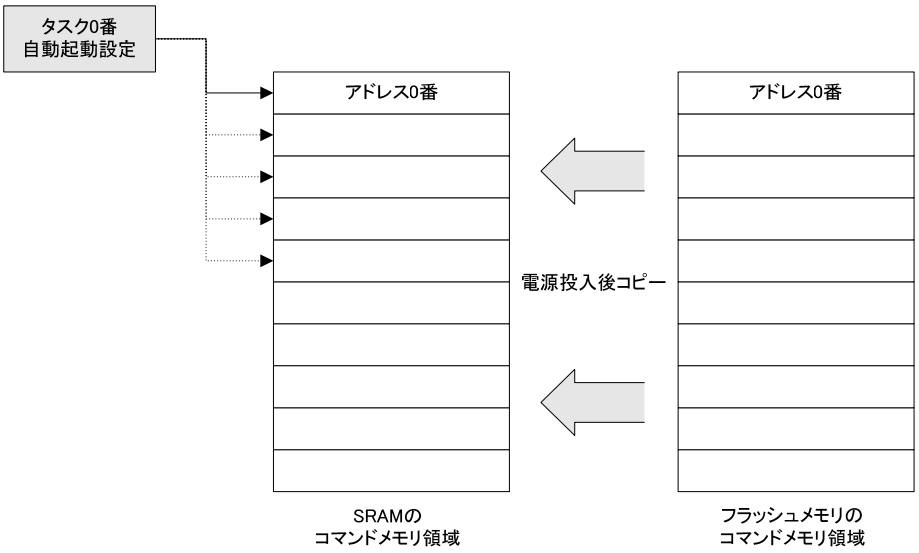
プログラムステップ数は対象のコントローラにより変化します。SVCC シリーズ（SV-NET Controller Compact）の場合、プログラムは 7800 ステップ、タスクは最大 8 本まで使用可能です。SVCE シリーズ（SV-NET Controller Ether）の場合、プログラムは 40000 ステップ、タスクは最大 8 本まで使用可能です。タスクとはプログラムを実行する内部のソフトウェアの事です。複数のタスクを起動する事で、複数のプログラムを並列に実行する事が可能です。



複数のプログラムを並列に実行

１．５ コマンドメモリ

プログラムを格納するエリアをコマンドメモリと呼びます。コマンドメモリ領域のデータは、電源投入後フラッシュメモリより SRAM にコピーされます。またコントローラのメモリスイッチ設定の【タスク 0 番自動起動】が ON になっていれば、自動的にタスク 0 番がコマンドメモリの先頭アドレスからプログラムを実行します。C 言語モーションコントローラでは、タスク 0 番がメインタスクに対応します。従いまして、C 言語のエントリポイントである main 関数はタスク 0 番に配置されます。



1．6 モーション制御仕様

(A) SVCC シリーズ

項 目	仕 様	備 考
制御軸数	MAX8 軸	SV-NET8 軸
伝送周期	2.0ms	SV-NET8 軸
補間周期	4.0ms	SV-NET8 軸
制御方式	位置制御、速度制御、トルク制御	
補間機能	直線補間（8 軸）、円弧補間（2 軸） ヘリカル補間（3 軸）	
補正機能	電子ギア	
指令単位	mm、deg	
最大指令値	-2147483648～2147483647	32bit 符号付き整数
速度指令単位	%、mm/sec、deg/sec、min^{-1}(rpm)	
加減速処理	S 字、台形制御方式	
無限長送り	有り	
原点復帰機能	原点近傍信号＋リミット信号	モータの零点を原点＋リミット指定可
	原点近傍信号 1	モータの零点を原点
	原点近傍信号 2	原点近傍信号入力即原点
	原点近傍信号 3	原点近傍信号解除後原点
	メカストッパ突き当て	メカストッパ突き当て式
速度オーバーライド機能	有り	0～100%
位置オーバーライド機能	有り	動作中に目標位置の上書きが可能
プログラムステップ	7800 ステップ	約 1MB
ユーザータスク	最大 8 本	
メモリバックアップ	有り	FLASH Memory に保存
整数型変数容量	128KB	32bit 符号付き整数
倍精度浮動少数点型変数容量	無し	未対応
算術演算	有り	C 言語対応
論理演算	有り	C 言語対応
ジャンプ命令	有り	C 言語対応
サブルーチン呼び出し	有り	C 言語関数呼び出し
スタックポインタ	512 個	

(B) SVCE シリーズ

項 目	仕 様	備 考
制御軸数	MAX16 軸	SV-NET8 軸＋EtherCAT8 軸
伝送周期	2.0ms（SV-NET）	SV-NET8 軸
	0.5ms（EtherCAT）	EtherCAT8 軸
補間周期	4.0ms（SV-NET）	SV-NET8 軸
	1.0ms（EtherCAT）	EtherCAT8 軸
制御方式	位置制御、速度制御、トルク制御	
補間機能	直線補間（8 軸）、円弧補間（2 軸）、 ヘリカル補間（3 軸）	
補正機能	電子ギア	
指令単位	mm、deg	
最大指令値	-2147483648～2147483647	32bit 符号付き整数
速度指令単位	%、mm/sec、mm/min、deg/sec、deg/min、 min^{-1}(rpm)	
加減速処理	S 字、台形制御方式	
無限長送り	有り	
原点復帰機能	原点近傍信号＋リミット信号	モータの零点を原点＋リミット指定可
	原点近傍信号 1	モータの零点を原点
	原点近傍信号 2	原点近傍信号入力即原点
	原点近傍信号 3	原点近傍信号解除後原点
	メカストッパ突き当て	メカストッパ突き当て式
速度オーバーライド機能	有り	0～100%
位置オーバーライド機能	有り	動作中に目標位置の上書きが可能
プログラムステップ	40000 ステップ	約 4MB
ユーザータスク	最大 8 本	
メモリバックアップ	有り	FLASH Memory に保存
整数型変数容量	1MB	32bit 符号付き整数
倍精度浮動少数点型変数容量	512KB	倍精度浮動少数点
算術演算	有り	C 言語対応
論理演算	有り	C 言語対応
ジャンプ命令	有り	C 言語対応
サブルーチン呼び出し	有り	C 言語関数呼び出し
スタックポインタ	512 個	

1

概要

２．C言語構文仕様

２．１　データ型

C言語モーションコントローラではint型（符号付き32ビット整数）とdouble型（倍精度浮動小数点）をサポートします。※double型はSVCCではサポートされません。

■　整数型

C言語モーションコントローラでサポートする整数型はint型のみです、int型は32ビット符号付き整数を表す型で、表現可能な範囲は-2,147,483,648 ～ 2,147,483,647となります。

使用例1）　`int num;`　　//変数名numを初期値無し“0”で使用

使用例2）　`int num = 10;`　//変数名numを初期値“10”で使用

■　浮動小数点型

C言語モーションコントローラでサポートする浮動小数点型はdouble型のみです、double型は倍精度浮動小数点を表す型で、表現可能な範囲は±4.94e－324（最小値） ～ ±1.79e308（最大値）となり、有効桁数は15桁です。

使用例1）　`double real = 3.14;`　　//変数名realを初期値“3.14”で使用

使用例2）　`double real = 3e10;`　　//変数名realを初期値“3e10”で使用

■　void型

void型は値が何も得られない事を示します。void型の使用される例としては、値を返却しない関数の型や引数の無い関数のパラメータとして使用されます。

使用例1）　`void func(int x, int y){・・・}`　//値を返却しない関数の例

使用例2）　`int func(void){・・・}`　　//引数の無い関数の例

２．２　定数

■　整数定数

整数定数は 10 進数と 16 進数で表すことができます。

・10 進定数　0 でない数字で始まります。(“0”は除く)

10 進定数の例）　1、10、1024

・16 進定数　0x か 0X の 2 文字で始まります。A～F までの 16 進数の数字は大文字・小文字を区別しません。

16 進定数の例）　0x7F、0Xff、OxABCD

■　浮動小数点定数

浮動小数点定数は、少数点を伴った 10 進数文字列か指数表記で表します。指数表記の E は小文字の e でもかまいません。

浮動少数点定数の例）　3.14　1.0E-3　//1.0×10^{-3}

２．３　式と演算子

式は演算子とオペランド（非演算数）の組合せです。式自身もオペランドになる為、演算子で結合し、より複雑な式を作り出すことが可能です。式に 2 つ以上の演算子がある場合、演算子とオペランドの結びつき方は、演算子の優先度により決定されます。算術演算子は+、-よりも*、/、%が優先されます。算術式は日常的な演算順位が適用されます。

■　演算子の優先度

下記に演算子の優先度を記載します。

優先度	演算子	結合規則
1	( ) [ ] -> .	左から右
2	! ~ ++ -- + - (type) * & sizeof	右から左
3	* / %	左から右
4	+ -	左から右
5	<< >>	左から右
6	< <= > =>	左から右
7	== !=	左から右
8	&	左から右
9	^	左から右
10	|	左から右
11	&&	左から右
12	||	左から右
13	?:	右から左
14	= += -= *= /= %= &= ^= |= <<= >>=	右から左
15	,	左から右

※灰色塗り潰し項目はサポートされていない演算子です。

■ 算術演算子

下記に算術演算子を記載します。

演算子	意味	例	結果
＊	乗算	x * y	x と y の積
/	除算	x / y	x と y の商
%	剰余	x % y	除算 x÷y の剰余
+	加算	x + y	x と y の和
-	減算	x - y	x と y の差
+（単項）	正の符号	+x	x の値
-（単項）	負の符号	-x	x の算術否定（符合反転）
++	インクメント	++x x++	x がインクリメントされる（x=x+1） 前置演算子（++x）は x の評価前にインクリメントする。 後置演算子（x++）は x の評価後にインクリメントする。
--	デクリメント	--x x—	x がデクリメントされる（x=x-1） 前置演算子（--x）は x の評価前にデクリメントする。 後置演算子（x--）は x の評価後にデクリメントする。

■ 代入演算子

下記に代入演算子を記載します。

演算子	意味	例	結果
=	単純代入	x = y	y の値を X に代入
+=	加算代入	x += y	x と y の値を加算した結果を x に代入
-=	減算代入	x -= y	x から y の値を減算した結果を x に代入
*=	乗算代入	x *= y	x と y の値を乗算した結果を x に代入
/=	除算代入	x /= y	x から y の値を除算した結果を x に代入
%=	剰余代入	x %= y	x と y の除算した余りを x に代入
&=	論理積代入	x &= y	x と y のビット単位の論理積を x に代入
^=	排他的論理和代入	x ^= y	x と y のビット単位の排他的論理和を x に代入
|=	論理和代入	x |= y	x と y のビット単位の論理和を x に代入
<<=	左シフト代入	x <<= y	x を y ビット左シフトした結果を x に代入
>>=	右シフト代入	x >>= y	x を y ビット右シフトした結果を x に代入

※代入式への代入式はサポートされていません。

例） x = y = 10; //コンパイルエラー

■ 関係演算子

下記に関係演算子を記載します。

演算子	意味	例	結果
<	より小さい	x < y	x が y よりも小さければ 1、そうでなければ 0
<=	より小さいか等しい	x <= y	x が y よりも小さいか等しければ 1、そうでなければ 0
>	より大きい	x > y	x が y よりも大きければ 1、そうでなければ 0
>=	より大きいか等しい	x >= y	x が y よりも大きいか等しければ 1、そうでなければ 0
==	等しい	x == y	x が y と等しければ 1、そうでなければ 0
!=	等しくない	x != y	x と y が等しくなければ 1、どうでなければ 0

■ 論理演算子

下記に論理演算子を記載します。

演算子	意味	例	結果
&&	論理積	x && y	x と y の両方が等しくなければ 1、そうでなければ 0
||	論理和	x || y	x と y の少なくとも一方が 0 と等しくなければ 1、そうでなければ 0
!	論理否定	!x	x が 0 に等しければ 1、そうでなければ 0

■ ビット演算子

下記にビット演算子を記載します。

演算子	意味	例	結果
&	ビット単位の論理積	x & y	x と y の両方が 1 なら 1、そうでなければ 0
|	ビット単位の論理和	x | y	x か y の片方あるいは両方が 1 なら 1、そうでなければ 0
^	ビット単位の排他的論理和	x ^ y	x か y の片方が 1 でもう一方が 0 なら 1、そうでなければ 0
~	ビット単位の論理否定	~x	x が 0 なら 1、1 なら 0
<<	左シフト	x << y	x の各ビットを y ビット分左にシフト
>>	右シフト	x >> y	x の各ビットを y ビット右にシフト

■ メモリアクセス演算子

下記にビット演算子を記載します。

演算子	意味	例	結果
&	アドレス	&x	x への定数ポインタ
*	ポインタ間接	*p	p によって指されるオブジェクト
[ ]	配列要素	x[i]	配列 x の添字 i の要素
.	構造体、共用体のメンバ	s.x	構造体あるいは共用体 s のメンバ x
->	構造体、共用体のメンバ	p->x	p で指される構造体あるいは共用体のメンバ x

※灰色塗り潰し項目はサポートされていない演算子です。

■ その他演算子

下記にその他演算子を記載します。

演算子	意味	例	結果
( )	関数呼び出し	func(x, y)	引数 x と y で関数を実行
(type)	キャスト	(long)x	指定された型での x の値
sizeof	バイト単位での大きさ	sizeof(x)	x が占めるバイト数
?:	条件の評価	x ? y : z	x が 0 でなければ y,そうでなければ z
,	逐次演算子	x, y	まず x、次に y を評価。for 文で使用される。

※灰色塗り潰し項目はサポートされていない演算子です。

2．4 文

C 言語における文は下記のように分類されます。

名前	構文	内容
名札付き文	識別子 : 文	goto 文のジャンプ先として使用される
	case 定数式 : 文	この名札は switc 文でのみ使用される
	default : 文	

複合文	{
	宣言
	文
	}
		複合文（ブロック文）は、複数の文を 1 つにまとめる役割を持つ。

式文	式;	式の後にセミコロンが続く文
空文	;	何の作用も及ぼさない
選択文	if（式）文	if 文の説明参照
	if（式）文 else 文	if 文の説明参照
	switch（式）文	switch 文の説明参照
繰り返し文	while（式）文	while 文の説明参照
	do 文 while（式）	do…while 文の説明参照
	for（式；式；式）文	for 文の説明参照
分岐文	goto 識別子:	goto 文の説明参照
	continue;	continue 文の説明参照
	break;	break 文の説明参照
	return 式;	return 文の説明参照

■ if 文

【構文】 if (式) 文 1 else 文 2

式の判定結果が 0 でない場合に文 1 が、式の判定結果が 0 の場合に文 1 が実行されます。下記に構文例を記載します。

if 文 例 1）

```
if ( x )
{
        x = y + 1;              //x の値が 0 以外の時に文 1（x = y + 1）が実行
}
else
{
        x = y - 1;              //x の値が 0 の時に文 1（x = y - 1）が実行
}
```

上記の構文において else 式内の処理が不要な場合は、else｛文 2｝は省略する事が可能です。また文 1、文 2 が 1 文のみの場合は、中括弧 “｛”“｝” を省略することが可能です。

if 文 例 2）

```
if ( x )
        x = x + 1;              //else 文と中括弧を省略した例
```

また else 文を else if 文に拡張する事により多くの条件に分岐する事が可能です。

if 文 例 3）

```
if ( x == 0 )       { x = y + 1; }      //x が 0 と等しい場合は y に 1 加算して x に代入
else if ( x == 1 )  { x = y + 2; }      //x が 1 と等しい場合は y に 2 加算して x に代入
else if ( x == 2 )  { x = y + 3; }      //x が 2 と等しい場合は y に 3 加算して x に代入
else                { x = 0; }          //x が 0～2 以外の場合は x に 0 を代入
```

■ switch 文

【構文】 switch (式) 文

switch 文の式の結果には int 型のみ許されます。式の値に一致する case ラベルまたは default ラベル内の処理が実行されます。下記に switch 文の構文例を記載します。

switch 文 例 1）

```
swicth ( x )
{
        case 0:
                x = y + 1;          //x が 0 と等しい場合は y に 1 加算して x に代入
                break;
        case 1:
                x = y + 2;          //x が 1 と等しい場合は y に 2 加算して x に代入
                break;
        default:
                x = 0;              //x が 0～1 以外の場合は x に 0 を代入
                break;
}
```

case ラベルに与えられる定数は互いに異なっていなければなりません。同じ値の定数を指定するとコンパイラがエラーを表示します。また case ラベルまたは default ラベル内の break 式を省略した場合には、次の case ラベルに処理が実行されます。

switch 文 例 2）

```
swicth ( x )
{
        case 0:
                x = y + 1;          //x が 0 と等しい場合は y に 1 加算して x に代入
        case 1:                     //break 式がない為、処理が case 1 ラベルに移る
                x = y + 2;          //x が 1 と等しい場合は y に 2 加算して x に代入
        default:                    //break 式がない為、処理が default ラベルに移る
                x = 0;              //x が 0～1 以外の場合は x に 0 を代入
                break;
}
```

■ while 文

【構文】 while (式) 文

while 文は上部駆動のループで、はじめにループ条件の式の判定結果が 0 以外であれば、ループ本体の文を実行し、0 であればループ本体の次の文から処理を継続します。下記に while 文の構文例を記載します。

while 文 例 1）

```
while ( x )
{
        y++;        //x の値が 0 以外なら y の値をインクリメントします。
}
```

while 文を使用し無限ループを作成するためにはループ式の評価結果が 1 となるようにプログラムを作成します。

while 文 例 2）

```
while ( 1 )         //条件式が常に 1 となり無限ループとなる。
{
        y++;
}
```

■ do…while 文

【構文】 do 文 while (式)

do…while 文は下部駆動のループで、はじめにループ本体の文を実行し次に式を評価します。ループ条件の式の判定結果が 0 以外であれば、ループ本体の文を実行し、0 であればループから処理を抜けます。while 文と大きく異なるのは、ループ本体の処理が少なくとも 1 回は実行される事です。下記に do…while 文の構文例を記載します。

do…while 文　例 1）

```
do
{
        y++;        //x の値が 0 以外なら y の値をインクリメントします。ループ式の評価前に必ず 1 度実行される。
} while ( x )
```

■ for文

【構文】 for (式1; 式2; 式3) 文

for 文は式 1 で一度だけ初期化を行い、式 2 のループ判定式の結果が 0 以外ならループ本体の文を実行します。ループ本体の文を実行後、式 3 の処理を行います。下記に for の構文例を記載します。

for 文 例 1）

```
for ( x = 0; x < y; x++ )        //for 文開始直後に一度だけ x = 0 の処理を行います。
{
        z++;                     //x が y より小さければ z の値をインクリメントします。
}                                //ループ本体の処理が終わったら、x の値をインクリメントします。
```

for 文を使用し無限ループを作成するためには全ての式を空文となるようにプログラムを作成します。

for 文 例 2）

```
for ( ; ; )                      //for 文の式全てを空文とし無限ループ
{
        z++;
}
```

■ goto 文

【構文】 goto ラベル名

goto 文を使うと、関数内の任意の位置にジャンプする事が可能です。ジャンプ先はラベルの名前で指定します。下記に goto 文の構文例を記載します。

goto 文 例 1）

```
while ( 1 )                    //無限ループ
{
        if ( x >= 100 )
        {
                goto LBL0;          //x が 100 より大きいか等しいなら、ラベル LBL0 へジャンプ
        }
}
LBL0:                          //ラベルはコロンの前にラベル名を指定する
```

ラベルは goto 文が実行された関数内に無くてはなりません。従って関数間をまたいでジャンプする事はできません。同一の関数内であれば goto 文の前にラベルが指定されていても動作します。

■ continue 文

【構文】 continue;

continue 文はループ本体の中でのみ使用可能です。continue 文が実行されるとループ本体の以降の処理が実行されず、ループ処理を継続します。ある条件の時だけループ本体の処理を実行したくない場合に使用します。下記に continue 文の構文例を記載します。continue 文　例 1） for (x = 0; x < y; x++) //for 文開始直後に一度だけ x = 0 の処理を行います。

```
{
        if( x == 0 ) { continue; }      //x の値が 0 と等しければ、以降のループ本体の処理をキャンセル
        z++;                            //x が y より小さければ z の値をインクリメントします。
}                                       //ループ本体の処理が終わったら、x の値をインクリメントします。
```

continue 文　例 2）

```
while ( x < y)                          //for 文開始直後に一度だけ x = 0 の処理を行います。
{
        if( x == 0 ) { continue; }      //x の値が 0 と等しければ、以降のループ本体の処理をキャンセル
        z++;                            //x が y より小さければ z の値をインクリメントします。
}                                       //ループ本体の処理が終わったら、x の値をインクリメントします。
```

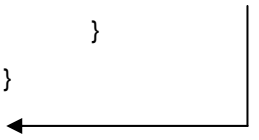
■ break 文

【構文】 break;

break 文を使うと、ループ文の任意の地点でループ実行を終了させる事がきます。下記に break 文の構文例を記載します。

break 文 例 1）

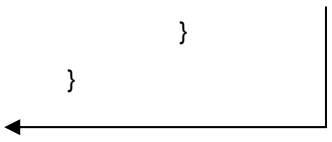
```
while ( 1 )                    //無限ループ
{
        if ( x >= 100 )
        {
                break;      //x が 100 より大きいか等しいなら、ループから脱出
        }
}
```



ループから脱出

多重のループにおいて break 文が実行された場合、一番内側のループのみ脱出します。下記に構文例を記載します。

```
while ( 1 )                    //無限ループ
{
        while ( 1 )
        {
                while ( 1 )
                {
                        if ( x >= 100 )
                        {
                                break;      //x が 100 より大きいか等しいなら、ループから脱出
                        }
                }
        }
}
```



一番内側のループのみ脱出

■ return 文

【構文】 return 式;

return 文は現在の関数の実行を終了し、制御を呼び出し側に返します。return 文の式に値を与えると、関数の返却値として呼び出し側に返却します。下記に return 文の構文例を記載します。

return 文 例 1）

```
vodi func ( int x, int y )                    //void 型の関数
{
        return;                               //void 型の関数に戻り値は無し
}
```

return 文 例 2）

```
int func2 ( int x, int y )                    //int 型の関数
{
        int z = x + y;
        return z;                             //int 型の値を返却
}
```

return 値で返却される値の型と関数の型が一致しないとコンパイラがエラーを表示します。

２．５　変数

■　変数の種類

コントローラはプログラムの実行部にマルチタスク構造を有する為、一般的なC言語で使用されるローカル変数（関数内で定義された変数）とグローバル変数（大域変数）のみではデータの信頼性に問題が発生する場合があります。例えばグローバル変数が異なるタスク間で同時またはそれに近いタイミングで変更・参照した場合にデータがプログラムの意図した内容であるかプログラムの実行タイミングに依存してしまう為です。そこでグローバル変数を各タスク毎で使用可能なグローバルタスク変数と全てのタスクから変更・参照可能なグローバルグローバル変数の2種類を新たに拡張しました。グローバルグローバル変数は全てのタスクから参照・変更できる為に便利ですが、複数のタスク間で使用する場合には他のタスクから変更される可能性がある事を前提に使用する必要があります。下記に変数の種類を記載します。

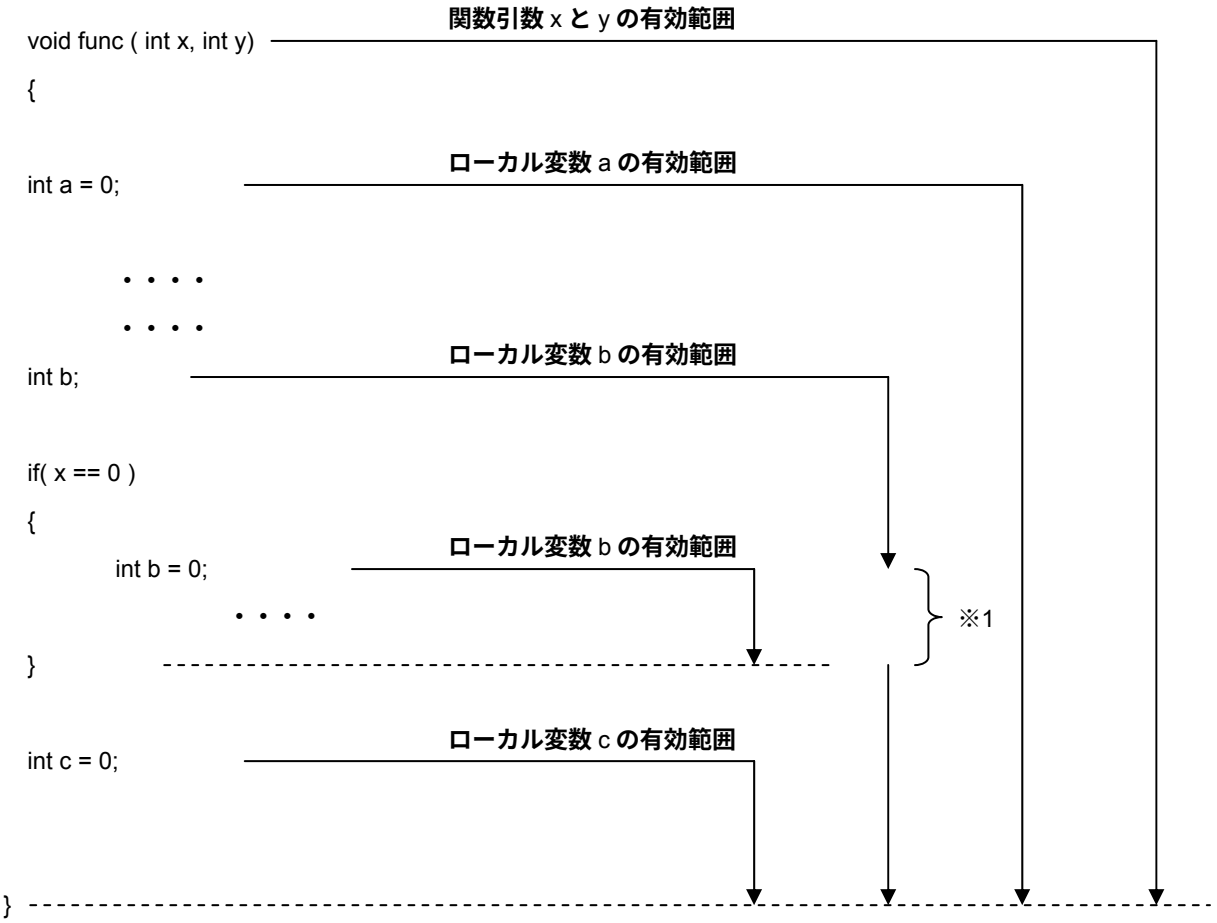
- 変数
  - ・ローカル変数
  - ・グローバル変数
    - ・グローバルグローバル変数
    - ・グローバルタスク変数

C言語モーションコントローラで使用可能な変数の種類

■ ローカル変数

ローカル変数は関数内で定義された一時的に使用可能な変数です。ローカル変数の有効範囲は関数内で定義された位置から関数を終了するまでの期間です。ローカル変数はブロックの先頭以外の任意の場所で定義可能です。但し for 文の初期化式の中で新たにローカル変数は定義する事は出来ません。下記にローカル変数の有効範囲を記載します。

■ ローカル変数の有効範囲



※1 の範囲はブロックレベルの深い方（if 文以下のブロック文）で定義された b のデータが有効。

■　グローバル変数

グローバル変数は前述したように、全てのタスクで使用可能なグローバルグローバル変数とタスク内でのみ使用可能なグローバルタスク変数に分類されます。それぞれの指定方法（グローバルグローバル変数かタスク＊変数なのか）は変数定義文の前に#pragma global、#pragma main、#pragma task＊、で定義する変数がどのタスクで使用されるか宣言し関数ブロックの外で宣言します。下記にグローバル変数の定義例を記載します。

■　グローバルグローバル変数の定義例

```
#pragma global
int x, y, z;                    //変数 x、y、z はグローバルグローバル変数
void func ( )                   //グローバルグローバル変数は関数ブロック外で定義
{
・・・
}
```

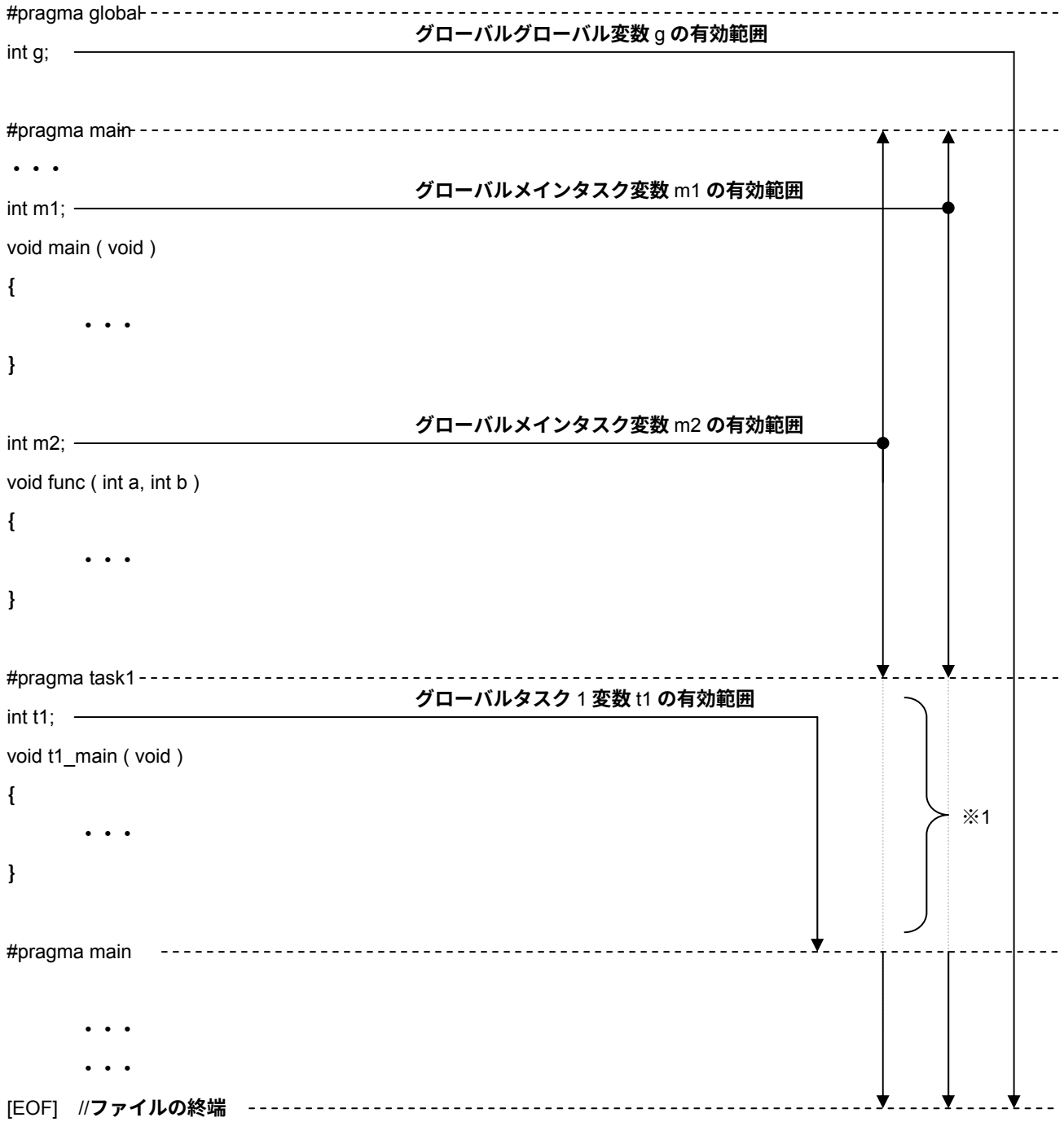
■　グローバルメインタスク変数の定義例

```
#pragma main
int u, v, w                     //変数 u、v、w、はグローバルメインタスク変数
void main ( )                   //グローバルメインタスク変数は関数ブロック外で定義
{
・・・
}
```

■　グローバルタスク 3 変数の定義例

```
#pragma task3
int a, b, c                     //変数 a、b、c、はグローバルタスク 3 変数
void task3_func ( )             //グローバルタスク 3 変数は関数ブロック外で定義
{
・・・
}
```

- グローバル変数の有効範囲（1 ファイル単位）



※1 の範囲はグローバルメインタスク変数 m1、m2 は無効。（使用できない）

■ グローバル変数の有効範囲（ファイル間）

グローバル変数はファイル間で有効範囲を保持します。ファイル 1 番でグローバルメインタスク変数として定義された変数はファイル 2 番のメインタスク領域（#pragma main 以下）で使用可能とです。下記に例を記載します。

```
#pragma global
int g;

#pragma main
int m1;                         //グローバルメインタスク変数 m1 はファイル 1 番で定義
void main ( void )
{
        ・・・
}
[EOF]   //ファイルの終端
```

ファイル 1 番

```
#pragma task1
int t1;

void t1_main ( void )
{
        ・・・
}

                                        ファイル 2 番でのグローバルメインタスク変数 m1 の有効範囲
#pragma main
void main_func ( void )
{
        m = (m << 4) + m;               //ファイル 1 番で定義したグローバルメインタスク変数 m1
}
[EOF]   //ファイルの終端
```

ファイル 2 番

■　記憶クラス

通常の C コンパイラでは static、extern、auto、extern 等の記録クラス指定子をサポートしていますが、C 言語モーションコントローラでは、グロバール変数の有効範囲を各タスク毎に仕様を拡張した為、記憶クラス指定子をサポートしていません。

2．6 予約語一覧

● C言語構文リスト

- auto
- break
- case
- char
- const
- continue
- default
- do
- double
- else
- enum
- extern
- float
- for
- goto
- if
- int
- long
- register
- return
- short
- signed
- sizeof
- static
- struct
- switch
- typedef
- union
- unsigned
- void
- volatile
- while
- string
- main

● アセンブラ構文リスト

- MAIN
- _main
- VARLIST
- VAREND
- DVARLIST
- DVAREND
- GVARLIST
- GVAREND
- NVARLIST
- NVAREND
- LVARLIST
- LVAREND
- VSTK※
- ASTK※
- LSTK※
- PSTK※
- RSTK※
- GSTK※
- DVSTK※
- DASTK※
- DLSTK※
- DPSTK※
- DRSTK※
- DGSTK※

※0～35 の数字

● モニター変数

- DI
- DO
- AI
- AO
- TIM
- HOME_LS
- TASK_STS
- SVD_CPLS
- SVD_FPLS
- SVD_FVEL
- SVD_FCUR
- SVD_STS
- SVD_ALM
- SVD_LOAD
- SVD_TEMP
- SVD_PWR
- IMU_WX
- IMU_WY
- IMU_WZ
- IMU_ACCX
- IMU_ACCY
- IMU_ACCZ
- IMU_ROLL
- IMU_PITCH
- IMU_YAW
- IMU_STS
- IMU_CNT
- IMU_ALM
- MCH_CPLS
- MCH_FPLS
- MCH_FCUR
- MCH_FVEL
- MCH_FSPD
- MCH_CPOS
- MCH_FPOS
- MCH_SVSTS
- MCH_SVALM
- MCH_JSTS
- MCH_STS
- MCH_ALM
- RS_STS
- RS_ERR
- RS_ECNT
- CC_STS

- CC_ERR
- CC_ECNT
- DV_STS
- DV_ERR
- DV_ECNT
- COM_AUTO
- COM_STS
- COM_OERR
- COM_FERR
- COM_PERR
- SV_TECREC
- SV_NERR
- SV_TERR
- SV_OERR
- SV_VERR

● ネットワーク変数

- NET
- RS
- CC
- DV
- PR
- CC_RX
- CC_RY
- CC_RWR
- CC_RWW
- DV_IN
- DV_OUT
- PR_IN
- PR_OUT
- TCP_IP
- ECAT_IN
- ECAT_OUT
- ETH_IN
- ETH_OUT
- M_CODE
- DP_RAM

● テーブル変数

- SVC_TBL
- POS_TBL
- VEL_TBL
- TIM_TBL
- CUR_TBL
- ACC_TBL
- MTN_TBL

● バックアップ変数

- FRAM

● API 関数

- Nop
- AlarmReset
- SmoothingSet
- ParameterSet
- ParameterGet
- ParameterSave
- MonitorGet
- NC_Call
- End

- copy
- abs
- swap
- swap2

- acos
- asin
- atan
- atan2
- cos
- sin
- tan
- cosh
- sinh
- tanh
- exp
- frexp
- ldexp
- log
- log10
- modf
- pow
- sqrt
- ceil
- fabs
- floor
- fmod

- TaskStart
- TaskId
- TaskStatus
- TaskReStart
- TaskStep
- TaskWait
- TaskEnd

- TimerSet
- Waitmsec

- BitOn
- BitOff
- BitIn
- DioOut
- DioIn
- AioOut
- AioIn

- PassM
- DecelM
- InposM
- OrgM
- PassA
- DecelA
- InposA
- OrgA

- ServoOn
- ServoOff
- ServoFree

- ServoMode
- ServoVelocity
- ServoCurrent
- ServoParameter

- RS_Start
- RS_Stop
- RS_Set
- RS_Get
- RS_Wait
- ComStart
- ComStop
- ComWrite
- ComRead
- EcatOperation
- EcatOperationN
- EcatMbxWrite
- EcatMbxRead
- EcatCycWrite
- EcatCycRead
- ModbusTcpRequest
- ModbusRtuRequest
- ModbusTcpVariableAllocate
- ModbusRtuVariableAllocate
- ModbusTcpBitMode
- ModbusRtuBitMode

- HomeStart
- HomeZero
- HomePosition
- HomeClear
- HomeBump
- HomeServo

- JogJ_Set
- MovaJ_Set
- MovaJT_Set
- MovaJBL_Set
- MovaJBLT_Set
- MovaJARC_Set
- MovaJARC1_Set
- MovaJARC2_Set
- MoviJ_Set
- MoviJT_Set
- MoviJBL_Set
- MoviJBLT_Set
- MoviJARC_Set
- MoviJARC1_Set
- MoviJARC2_Set

- MovaWL_Set
- MoviWL_Set
- MovaWA_Set
- MoviWA_Set
- MovaWL
- MoviWL
- MovaWA
- MoviWA

- MoveStart
- MoveStopAll
- MoveStopAxis
- MoveHold
- OVR_Set
- OVR_Get

- TableTeach
- TableSave
- TableMovaP
- TableMovaT
- TableMoviP
- TableMoviT

- UserFunc

● アセンブラコマンド

- NOP
- PRMSET
- PRMREF
- VARSET
- VARREF
- ALMRST
- ALMCLR
- ACCSET
- PRMSET2
- DVARSET
- DVARREF
- END
- SEMSET
- SEMCLR
- SEMRST
- PRMSAVE
- REBOOT

- CALC
- PUSH
- POP
- SALLOC
- SFREE
- SPSAV
- SPRST
- PRMGET
- MONGET
- ZPGET
- CALC2
- AFREE
- COPY

- ID
- NOT
- NEG
- ABS
- ADD
- SUB
- MUL
- DIV
- MOD
- AND
- OR
- XOR

- ROT
- SHIFT
- FIELD1
- FIELD2
- SCALE
- SIN
- COS
- MERGE
- SWAP
- SWAP2

- JUMP0
- JUMP1
- CALL
- RET
- JMP0
- JMP1
- JMP2
- JMPAND
- JMPEQ
- JMPNE
- JMPLT
- JMPGT
- JMPLE
- JMPGE
- JMPBIT
- JNPBIT
- JMPAXIS
- JMPMCH
- JMPDIO
- JNPDIO

- DJMP2
- DJMPEQ
- DJMPNE
- DJMPLT
- DJMPGT
- DJMPLE
- DJMPGE

- TASTART
- GETTID
- GETTST
- TRESTART
- TSTEP
- TWAIT
- TEND

- SETT
- WAITT
- TIME
- WAIT

- OUTPUT
- INPUT
- DAOUT
- ADIN
- DOUT
- DIN
- AOUT
- AIN
- BITON
- BITOFF
- BITIN

- PASSM
- DECELM
- INPOSM
- ORGM
- PASSA
- DECELA
- INPOSA
- ORGA

- DCALC
- DCALC2
- DID
- DNEG
- DADD
- DSUB

- DMUL
- DDIV
- ITOD
- DTOI
- DCOPY

- ACOS
- ASIN
- ATAN
- ATAN2
- COS
- SIN
- TAN
- COSH
- SINH
- TANH
- EXP
- FREXP
- LDEXP
- LOG
- LOG10
- MODF
- POW
- SQRT
- CEIL
- FABS
- FLOOR
- FMOD

- SVINIT
- SVON
- SVOFF
- SVFREE
- SVPRM
- SVMODE
- SVVEL
- SVCUR
- SVPRM2
- SVSCAN
- SVEND

- HOME
- HOMESET
- HOMESET2
- HOME2
- HOMESTART
- HOMEPOS
- HOMECLR
- HOMINGS
- HOMINGE
- HOMEBUMP
- HOMESV

- RUNRS
- STOPRS
- GETRS
- SETRS
- FINRS
- RUNRSN
- STOPRSN
- GETRSN
- SETRSN

- ECATOP
- ECATOPN
- ECATMBXWR
- ECATMBXRD
- ECATCYCWR
- ECATCYCRD
- ECATCANSET
- ECATPRMSET

- MODTREQ
- MODTALLOC
- MODTBITMD
- MODRREQ
- MODRALLOC
- MODRBITMD

- SETJOGJ
- JOGJ

- SETMOVAJ
- SETMOVAJT
- SETMOVAJFS
- SETMOVAJCU
- SETMOVAJTW
- SETMOVAJA1
- SETMOVAJA2
- SETMOVAJBL
- SETMOVAJBLT
- SETMOVAWL
- SETMOVAWA

- SETMOVIJ
- SETMOVIJT
- SETMOVIJFS
- SETMOVIJCU
- SETMOVIJTW
- SETMOVIJA1
- SETMOVIJA2
- SETMOVIJBL
- SETMOVIJBLT
- SETMOVIWL
- SETMOVIWA

- MOVAJ
- MOVAJT
- MOVAJFS
- MOVAJCU
- MOVAJTW
- MOVAJA1
- MOVAJA2
- MOVAJBL
- MOVAJBLT
- MOVAWL
- MOVAWA

- MOVIJ
- MOVIJT
- MOVIJFS
- MOVIJCU
- MOVIJTW
- MOVIJA1
- MOVIJA2
- MOVIJBL
- MOVIJBLT
- MOVIWL
- MOVIWA

- MOVE
- STOP
- SETOVR
- GETOVR
- SETBUF
- GETBUF
- SETWAIT
- GETWAIT
- STOPJ
- HOLDJ

- SETTBL
- FILLTBL
- SETPTBL
- FILLPTBL
- SETVTBL
- FILLVTBL
- SETATBL
- FILLATBL
- SETTTBL
- FILLTTBL
- TEACH
- TBLSAVE

- TBLMOVA
- TBLMOVA2
- TBLMOVAT
- TBLMOVAT2
- TBLMOVI
- TBLMOVI2

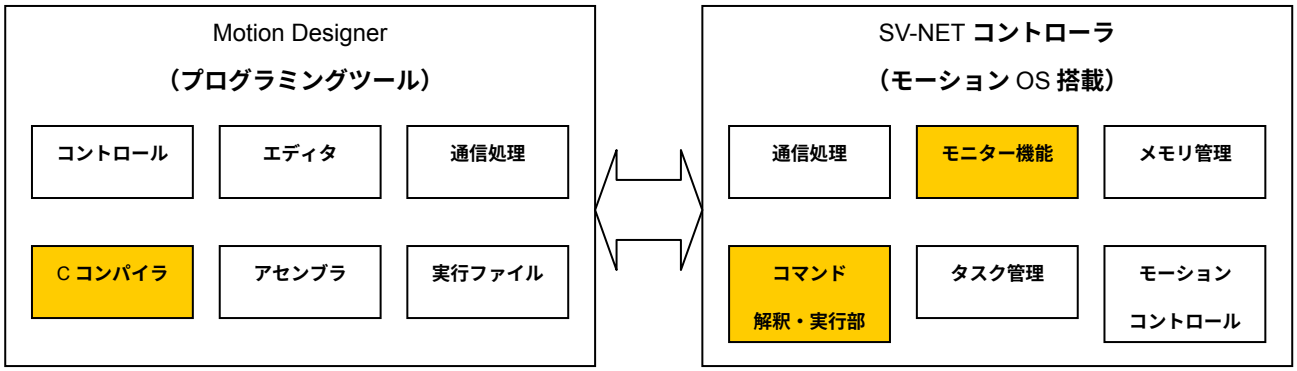
- TBLMOVIT
- TBLMOVIT2

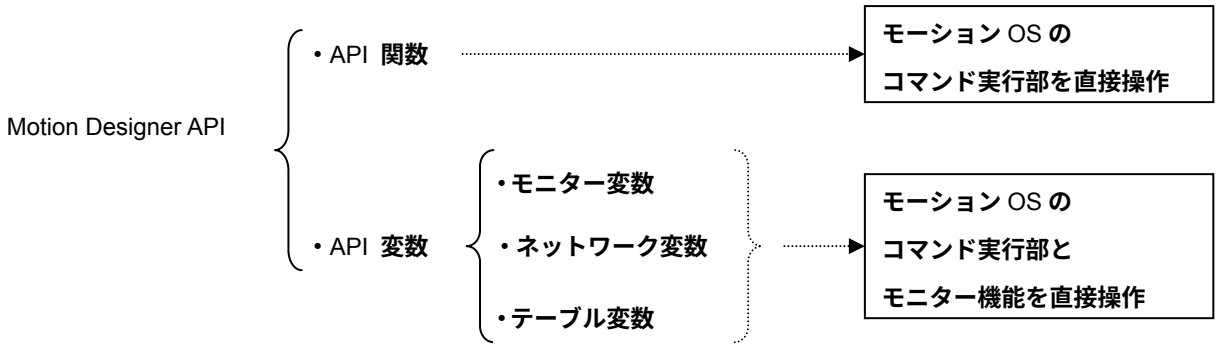
3．Motion Designer API 概要

3．1 Motion Designer API

Motion Designer API とは、モーション OS のコマンド実行部、モニター機能部に C 言語から直接アクセス可能な機能の総称です。API を分類すると、コマンド実行部に直接アクセス可能な Motion Designer API 関数と、機器の状態を監視するモニター変数やネットワーク機器のデータを監視するネットワーク変数等の Motion Designer API 変数に分類されます。これらの API を使用する事により、簡単に Motion Designer の開発環境上からモーション OS にアクセスする事が可能となり、様々な機械の状況に対応することが可能となります。

C 言語モーションコントローラの機能ブロック図と Motion Designer API の分類を図示します。





Motion Designer API の分類

３．２ 動作命令 API 関数

■ 動作設定・開始命令の説明

Motion Designer API からモーション OS の軸動作設定及び軸動作開始を直接実行する事が可能です。API で提供する動作設定命令では 1 軸づつ動作目標値を設定します。複数の軸を同期させて動作させる場合は、あらかじめ複数軸の動作目標値を設定し、動作開始命令で複数の軸を同時に開始します。API で提供される軸動作設定命令には下記の種類があります。

分類	関数名	説明
動作設定命令	JogJ_Set	一定速動作
	Mov＊J_Set	位置決め
	Mov＊JT_Set	時間指定位置決め
	Mov＊JBL_Set	ベル型加減速
	Mov＊JARC_Set	中心・角度指定円弧補間

関数名の＊には “ a ” または “ i ” を指定します。“ a ” が指定された場合は、引数に与えられた目標位置がコントローラの管理する座標の絶対位置に、“ i ” が指定された場合は、コントローラの管理する座標の現在位置からの相対位置に変換されます。また動作開始命令は全ての動作設定命令共通で、下記の関数となります。

分類	関数名	説明
動作開始命令	MoveStart	動作開始

動作開始命令で設定された軸は同時に軸動作を開始（同期）します。

■ 動作設定・開始命令の引数リスト

動作設定命令、動作開始命令ともに引数の個数は異なりますが、全ての関数において第 1 の引数と第 2 の引数は共通です。第 1 の引数は“ 機構番号 ”、第 2 の引数は“ 設定軸番号 ”となり、第 3 の引数以降は関数毎に意味が異なります。下記に MoveStart 関数と MovaJ_Set 関数の例を記載します。

・MoveStar (mch, setup)

引数の説明　第 1 の引数 mch：機構番号　第 2 の引数 setup：設定軸番号

・MovaJT_Set (mch, setup, pos, tim)

引数の説明　第 1 の引数 mch：機構番号　第 2 の引数 setup：設定軸番号

第 3 の引数 pos：目標位置　第 3 の引数 tim：到達時間

■ 機構番号

コントローラの制御対象は、いくつかの「軸」から構成されます。それらをまとめたグループを一括して「機構」または「グループ」と称します。既定の設定では存在する各軸が、一つの機構に各軸座標系として登録されています。SV-NET を 2 系統持つコントローラは、SV-NET 系統毎に異なる機構グループが存在します。SV-NET と EtherCAT の共存するコントローラでも同様です。動作設定・開始命令に与えられる第 1 の引数は、軸の集まりをまとめたグループ番号を意味します。本書では機構番号のことをグループ番号とも記載します。

■ 設定軸番号

設定軸番号引数の下位ビットから、優先的に立っているビットに対応したデータが「設定有り」となります。
設定軸番号引数のビット 0 から 31 が、機構に属する第 1 軸から第 32 軸までに対応します。

例）設定軸番号に与えられる引数の値が 0x0505 の場合　第 1 軸、第 3 軸、第 9 軸、第 11 軸が「設定有り」となります。
　　コントローラに最大軸数の制限がある場合は、最大軸数以上の設定は無効です。

動作設定命令は一度の命令で指定された 1 軸のみ設定可能です。従って、設定軸番号に複数軸を設定しても複数軸の目標値は設定されません。具体的には複数軸を設定軸番号に指定した場合、設定軸番号で指定された番号の一番若い軸の目標値が設定されます。

例）MovaJT_Set (0, 0x05, pos, tim)
　　上記の例では機構（グループ）番号 0 番目の第 1 軸目と第 3 軸が指定されていますが、実際に目標値が設定されるのは、第 1 軸のみとなります。

■ 同時到達

目標値有りの軸が複数存在している場合、次の動作開始命令で指定された設定軸は同時到達となります。設定軸番号の中で到達時間が最長の軸に合わせて、他の軸の動作速度を低下させます。（長軸基準）但し同時到達させる為には、各軸の加減速フィルタの設定値を同じ値にする必要があります。同時到達の必要はなく、各軸ともに目標速度で移動したい場合には、動作開始命令を分けて実行する必要があります。下記にその場合の例を示します。

```
MovaJ_Set ( 0, 0x01, 10000, 2000 );     //グループ 0 番　設定軸 1 軸目　目標位置 10000　目標速度 2000
MoveStart ( 0, 0x01 );                  //1 軸目動作開始
MovaJ_Set ( 0, 0x02, 15000, 4000 );     //グループ 0 番　設定軸 2 軸目　目標位置 15000　目標速度 4000
MoveStart ( 0, 0x02 );                  //2 軸目動作開始
InposM ( 0 );
```

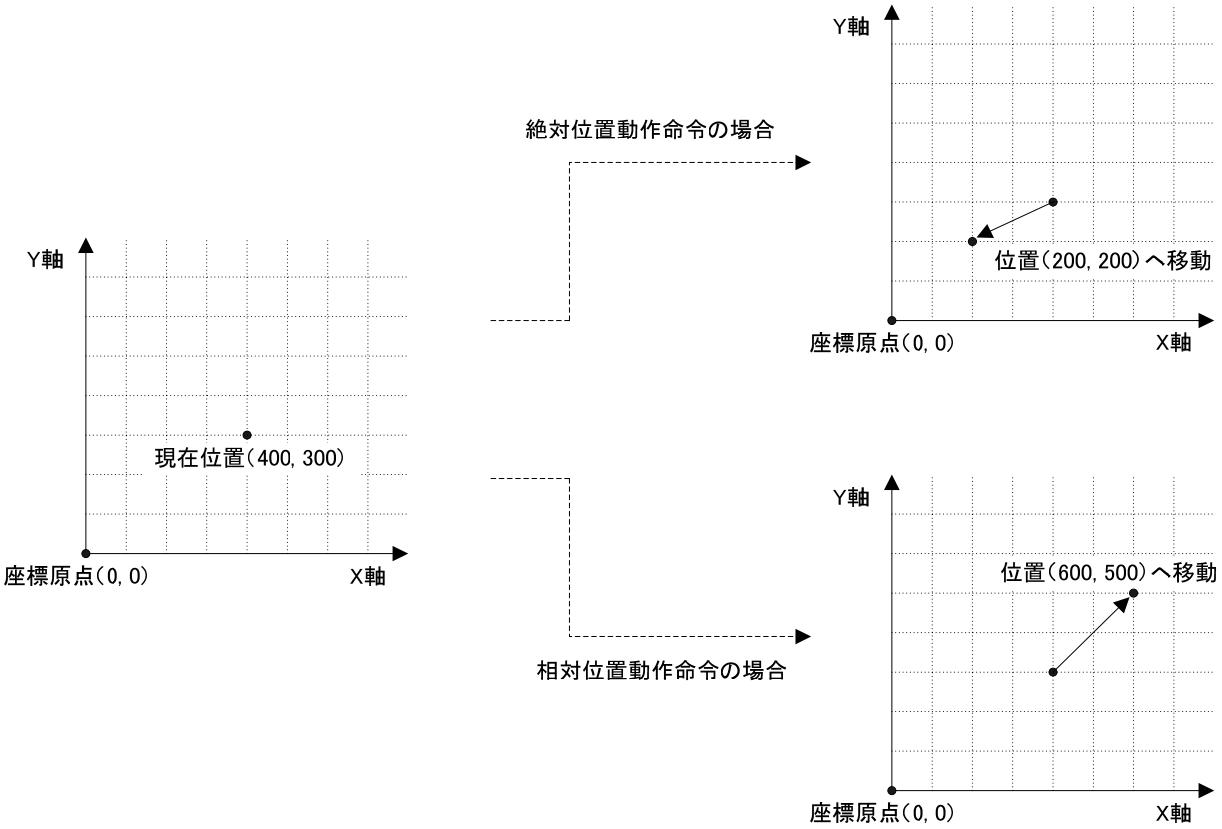
上記のプログラムを実行すれば、1 軸目、2 軸目ともに目標速度で目標位置まで移動します。ノーウェイト型の動作命令により、MoveStart 関数実行後、軸の動作完了を待たずに次の命令を実行可能です。InposM 関数は機構（グループ）に属する軸が、全てインポジションになるまで現在インデクスで待機する命令です。

■ 位置オーバーライド

既に動作中の軸に対して、新たに動作設定・開始命令を実行すれば、目標位置、目標速度の再設定が可能です。
すなわち目標位置のオーバーライドが可能です。

■ 絶対位置動作命令

コントローラの動作命令には、絶対位置動作命令と相対位置動作命令があります。絶対位置動作は、コントローラの座標原点を基準に目標位置の値をドライバに指令値として与えます。相対位置動作は、現在位置を基準に指令位置をドライバに与えます。仮に直交する 2 軸の現在位置が（400、300）の位置にある時、動作命令の引数として（200、200）を与えた場合の、絶対位置動作命令と相対位置動作命令の動作を下図に記載します。



■ 動作設定命令の移動方向

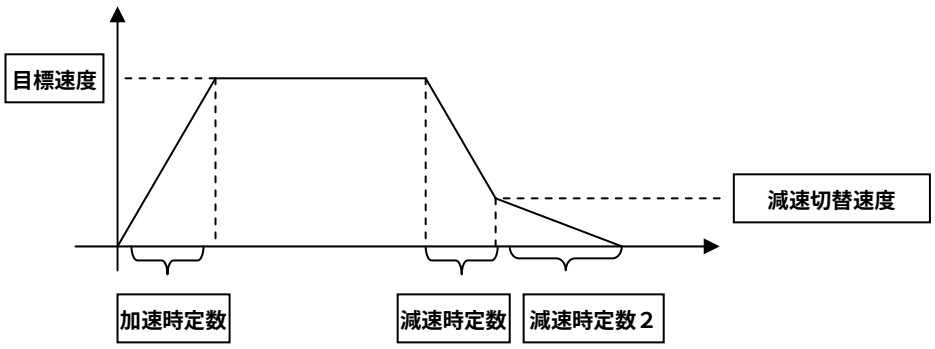
動作設定命令において移動方向は目標位置により決定します。例えば、相対位置動作命令の場合、目標位置の符号が正であれば正方向に移動し負であれば負方向に移動します。絶対位置動作の場合は、現在位置を基準に目標位置が大きければ正方向に移動し小さければ負方向に移動します。但し、JogJ_Set 関数や HomeStart 関数のように動作命令の種類によっては、目標位置の引数を持たない命令もあります。これらの命令で移動方向を変更するには、速度の引数に符号を与える事で実現します。通常の動作設定命令では、速度／時間の引数に符号を与えても内部で無効（符号無し）とし処理されます。

■ 複合動作コマンド

一般的な動作パターン（加速－等速－減速）を、幾つかの動作設定命令を組み合わせることで実現する動作を、複合動作コマンドと呼びます。複合動作コマンドを使用する場合には、速度 3 次タイプのベル型加減速動作設定命令 Mov※JBL_Set 関数を使用します。複合動作コマンドを使用する場合、加減速フィルタの値になるべく短い時間を設定してください。また複合コマンドを使用する際には、初速と終速の引数に正しい値を与える事が重要です。実際にプログラムした後の動作が滑らかでない場合は、初速と終速の引数の値を見直す必要があります。

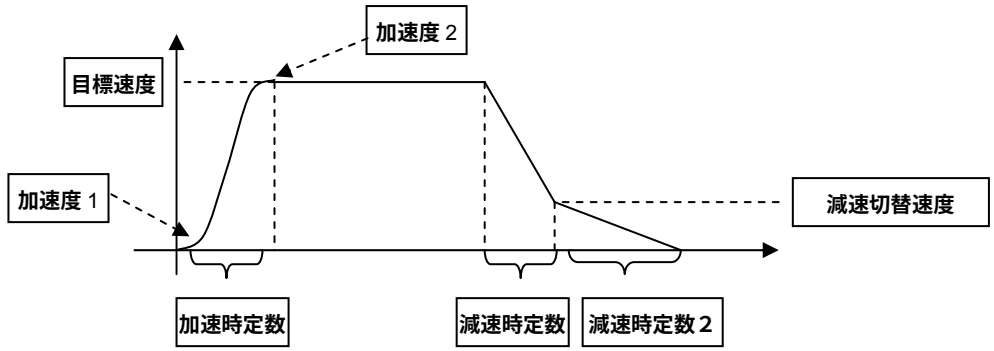
■ 速度 1 次タイプ

速度 1 次タイプの速度曲線の例です。速度 1 次タイプの速度曲線は、Mov※JBL_Set 関数の加速度係数 1、2 を 100%に指定する事で実現可能です。

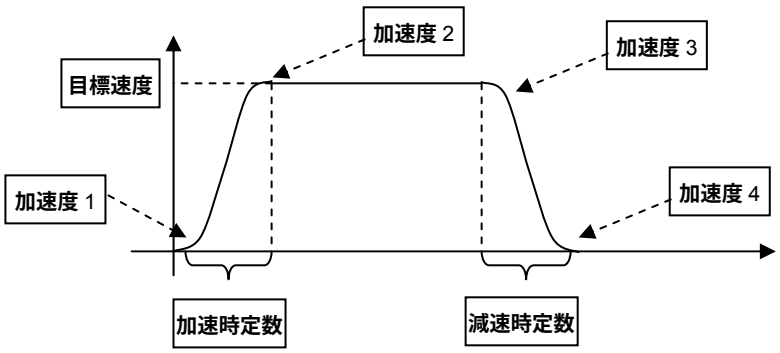


■ 速度 3 次タイプ

速度 3 次タイプの速度曲線の例 1 です。（下図は、減速側が 1 次の場合）



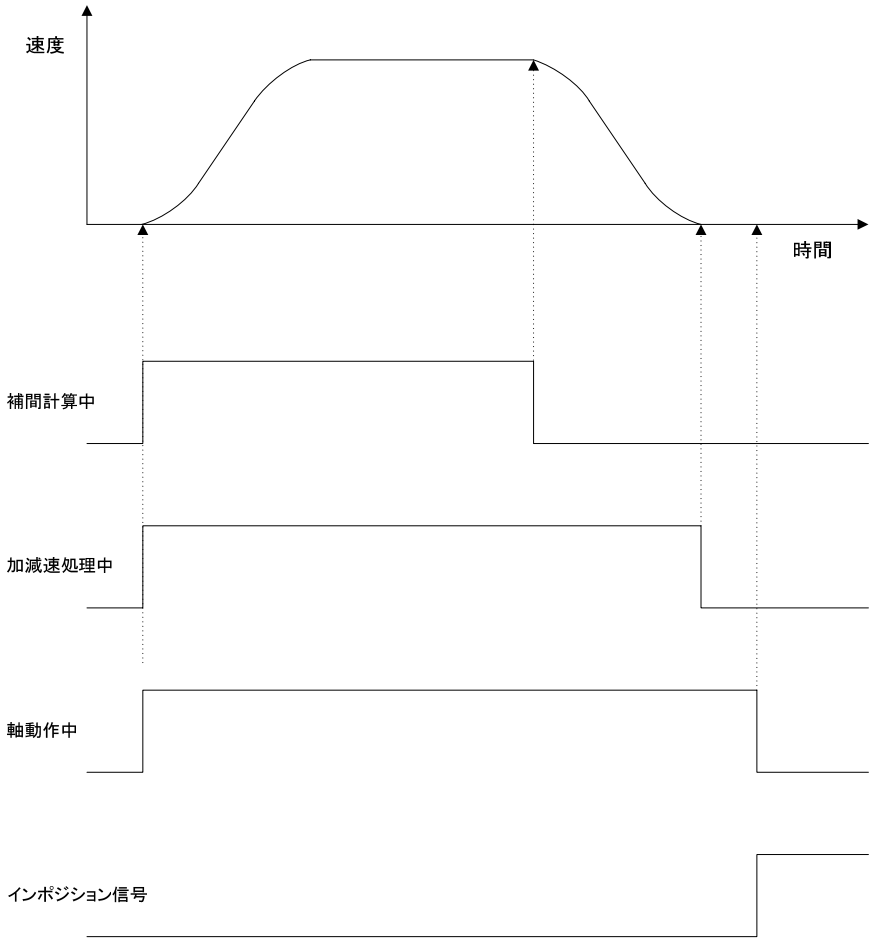
速度 3 次タイプの速度曲線の例 2 です。（下図は、減速側も 3 次の場合）



３．３ PASS 命令 API 関数

■ PASS ポイント

コントローラは動作命令実行後に、補間計算中／指令払い出し中（加減速処理中）／軸動作中／原点復帰中の 4 つの PASS ポイントを持っています。各ポイントがどのような意味を持つのか下図を用いて説明します。（原点復帰中は除く）



■ 補間計算中

コントローラ内部で位置指令データを計算中の状態です。補間計算が完了しても、加減速フィルタに値が設定されている場合、フィルタの払い出しが完了するまで軸は動作します。補間計算中の PASS ポイントは、主に複合動作コマンドを使用する場合に使用します。補間計算中は、指令払い出し中（加減速処理中）／軸動作中ステータスも ON の状態となります。

■ 指令払い出し中（加減速フィルタ処理中）

コントローラ内部で指令位置の払い出しが未完了の状態です。指令払い出し中の PASS ポイントは、インポジションを待たなくても良い動作の時に使用します。この PASS ポイントを使用する事により、装置の運転時間を短縮する事が可能となります。指令払い出し中は、軸動作中ステータスも ON の状態となります。

■ 軸動作中

指令払い出し完了後、ドライバからインポジション信号が OFF の状態です。
軸動作中の PASS ポイントは、インポジションを待ってから、次のプログラムを実行したい場合に使用します。

■ PASS 命令の種類

PASS 命令の種類としては、機構に属する全軸の PASS ポイントを指定する PassM／DecelM／InposM／OrgM 関数と、機構に属する各軸の PASS ポイントを指定する PassA／DecelA／InposA／OrgA 関数があります。それぞれの内容は下表の通りです。

機構内全軸	機構内各軸	PASS ポイント	内容
PassM 関数	PassA 関数	補間計算中	補間計算中なら現在インデクスで待機
DecelM 関数	DecelA 関数	加減速処理中	指令払い出し未完了なら現在インデクスで待機
InposM 関数	InposA 関数	軸動作中	軸動作中なら現在インデクスで待機
OrgM 関数	OrgA 関数	原点復帰動作中	原点復帰中なら現在インデクスで待機

３．４ サーボ命令 API 関数

■ サーボ命令について

サーボ命令は、ServoOn（サーボオン）／ServoOff（サーボオフ）／ServoFree（サーボフリー）関数の他に、ドライバの制御モードを設定する ServoMode 関数、速度制御時の速度を設定する ServoVelocity 関数、電流制御時の電流値を設定する ServoCurrent 関数、ドライバのパラメータを変更する ServoParameter 関数があります。

■ サーボオン／サーボオフ／サーボフリー命令について

ServoOn、ServoOff、ServoFree 関数は、いずれも 2 個の共通引数を持ちます。引数の内容は動作開始命令と同じく、第 1 の引数が“ 機構番号 ”、第 2 の引数が“ 設定軸番号 ”となります。設定軸番号に複数軸を設定し、同時にサーボオンする事が可能です。設定軸番号は動作命令と同様に、ビット 0～ビット 31 が第 1 軸～第 32 軸に対応します。

■ サーボパラメータ命令について

ServoParameter／ServoMode／ServoVelocity／ServoCurrent 関数は、ドライバにパラメータを転送するコマンドです。サーボオン命令と異なり一度に 1 軸のみ設定可能です。ServoParameter／ServoMode／ServoVelocity／ServoCurrent 関数は第 1 の引数、第 2 の引数が共通です。第 1 の引数は“ 機構番号 ”、第 2 の引数は“ 軸番号 ”となります。第 2 の引数がサーボオン命令や動作命令と異なり、“ 設定軸番号 ”ではなく、“ 軸番号 ”となっている点に注意が必要です。軸番号は機構（グループ）に属する軸番号が 1 軸目なら“ 1 ”を、2 軸目なら“ 2 ”を指定します。第 3 の引数以降は命令毎に異なります。

例）ServoMode (0, 2, 2)　　//グループ 0 番 2 軸目を速度制御モードに設定します。

サーボパラメータ命令は、機構（グループ）が接続されるネットワークの種類のより使用可否が決定します。詳しくは各関数の説明を参照します。

３．５ タイマ命令 API 関数

■ タイマ命令について

コントローラの既定の設定では、プログラムで使用出来るタイマが 32 個用意されています。タイマのカウンタ最大値は 2147483647 です。またタイマは常に 1msec でカウントアップされおり、最大値になるとそれ以上カウントアップされません。タイマ命令には TimerSet（タイマ初期化）命令と Waitmsec（単純時間待ち）命令があります。Waitmsec 命令は実行時に、現在タイマ値が “ 0 ” に初期化されます。複数の異なるタスクでタイマ命令を使用する場合は、タイマ番号が重複しないようにします。※単純時間待ち処理を実行したい場合は、Waitmsec 関数より TaskWait 関数を使用した方が簡単です。TaskWait 関数はタイマ番号の重複を考慮する必要はありません。処理を実行したタスクに対して有効です。

３．６ I／O 命令 API 関数

■ I／O 命令について

I／O 命令により、コントローラに搭載されているデジタル I／O やアナログ I／O のデータを参照・設定する事が可能です。指定されたデジタル I／O の全ビットを操作する、DioIn／DioOut 関数と、任意のビットを操作可能な BitOn／BitOff／BitIn 関数があります。DioIn／BitIn 関数のようにデジタル I／O の入力値をプログラムで使用する場合は、モニター変数 DI[※]を使用したほうが簡単です。

4．Motion Designer API 変数一覧

■ モニター変数（I/O 系）

分類	変数名	説明
モニター変数（I/O 系）	DI [※]	デジタル I/O の入力データ
	DO [※]	デジタル I/O の出力データ
	AI [※]	アナログ I/O の入力データ
	AO [※]	アナログ I/O の出力データ

■ モニター変数（ドライバ系）

分類	変数名	説明
モニタ変数（ドライバ系）	SVD_CPLS [※]	サーボドライバの現在指令パルス（単位：パルス）
	SVD_FPLS [※]	サーボドライバの現在実パルス（単位：パルス）
	SVD_FVEL [※]	サーボドライバの現在実速度（単位：rpm）
	SVD_FCUR [※]	サーボドライバの現在実電流（単位：0.01A ）
	SVD_STS [※]	サーボドライバの現在サーボステータス
	SVD_ALM [※]	サーボドライバのアラームコード
	SVD_LOAD [※]	サーボドライバの過負荷モニター（単位：0.1%） SV-NET Only
	SVD_TEMP [※]	サーボドライバの内部温度（単位：0.1℃） SV-NET Only
	SVD_PWR [※]	サーボドライバの駆動電源電圧（単位：0.1V） SV-NET Only

■ モニター変数（コントローラ系）

分類	変数名	説明
モニタ変数（コントローラ系）	MCH_CPLS [※] [※]	機構内（グループ）各軸の現在指令パルス（単位：パルス）
	MCH_FPLS [※] [※]	機構内（グループ）各軸の現在実パルス（単位：パルス）
	MCH_FVEL [※] [※]	機構内（グループ）各軸の現在実速度（単位：rpm）
	MCH_FCUR [※] [※]	機構内（グループ）各軸の現在実電流（単位：0.01A ）
	MCH_FSPD [※] [※]	機構内（グループ）各軸の現在実速度（単位：速度指令単位）
	MCH_CPOS [※] [※]	機構内（グループ）各軸の現在指令位置（単位：位置指令単位）
	MCH_FPOS [※] [※]	機構内（グループ）各軸の現在実位置（単位：位置指令単位）
	MCH_SVSTS [※] [※]	機構内（グループ）各軸の現在サーボステータス
	MCH_SVALM [※] [※]	機構内（グループ）各軸のアラームコード
	MCH_JSTS [※] [※]	機構内（グループ）各軸の現在機構ステータス
	MCH_ALM [※]	機構（グループ）のアラームコード
	MCH_STS [※]	機構（グループ）の現在機構ステータス

■ モニター変数（ステータス系）

分類	変数名	説明
モニタ変数（ステータス系）	TIM [※]	タイマの現在値（単位：msec）
	TASK_STS [※]	タスク現在ステータス
	CC_STS	CC Link　通信ステータス
	CC_ERR	CC Link　エラーコード
	CC_ECNT	CC Link　エラーカウンタ
	DV_STS	DeviceNet　通信ステータス
	DV_ERR	DeviceNet　エラーコード
	DV_ECNT	DeviceNet　エラーカウンタ
	RS_STS	RS232C　通信ステータス
	RS_ERR	RS232C　エラーコード
	RS_ECNT	RS232C　エラーカウンタ

分類	変数名	説明
モニタ変数（ステータス系）	COM_AUTO [※]	COM ポート　自動送受信の状態
	COM_STS [※]	COM ポート　自動送受信ステータス
	COM_OERR [※]	COM ポート　オーバーランエラーカウンタ
	COM_FERR [※]	COM ポート　フレーミングエラーカウンタ
	COM_PERR[※]	COM ポート　パリティエラーカウンタ
	SV_TECREC	SV-NET　CAN エラーカウンタ
	SV_NERR	SV-NET　受信未完了エラーカウンタ
	SV_TERR	SV-NET　送信未完了エラーカウンタ
	SV_OERR	SV-NET　受信オーバーランカウンタ
	SV_VERR	SV-NET　SV-NET エラーデータ受信カウンタ

■ ネットワーク変数

分類	変数名	説明
ネットワーク変数	RS [※]	RS232C 通信データ格納用ネットワーク変数
	CC_RX [※]	CC Link 接点データ格納用ネットワーク変数 （方向：マスタ←コントローラ）
	CC_RY [※]	CC Link 接点データ格納用ネットワーク変数 （方向：マスタ→コントローラ）
	CC_RWR [※]	CC Link レジスタ格納用ネットワーク変数 （方向：マスタ←コントローラ）
	CC_RWW [※]	CC Link レジスタ格納用ネットワーク変数 （方向：マスタ→コントローラ）
	DV_IN [※]	DeviceNet レジスタ格納用ネットワーク変数 （方向：マスタ→コントローラ）
	DV_OUT [※]	DeviceNet レジスタ格納用ネットワーク変数 （方向：マスタ←コントローラ）
	TCP_IP [※]	TCP/IP 用ネットワーク変数
	ECAT_IN [※]	EtherCAT レジスタ格納用ネットワーク変数 （方向：スレーブ→コントローラ）
	ECAT_OUT [※]	EtherCAT レジスタ格納用ネットワーク変数 （方向；スレーブ←コントローラ）
	ETH_IN [※]	Ethernet レジスタ格納用ネットワーク変数
	ETH_OUT [※]	Ethernet レジスタ格納用ネットワーク変数
	DP_RAM [※] [※]	外部拡張 RAM 格納用ネットワーク変数

■ テーブル変数

分類	変数名	説明
テーブル変数	SVC_TBL [※] [※] [※]	テーブル変数
	POS_TBL [※] [※]	位置テーブルデータ
	VEL_TBL [※] [※]	速度テーブルデータ
	TIM_TBL [※] [※]	時間テーブルデータ
	ACC_TBL [※] [※]	加減速テーブルデータ

５．Motion Designer API 変数詳細

モニター変数（I/O 系）　詳細

■ DI [※1]

【 説明 】 デジタル I/O の入力データを取得します。

【 配列 】 ※1 第 1 のインデクス : デジタル I/O 番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( DI[0] & 0x05 )          //デジタル I/O_0 番のビット 0 番、ビット 2 番が “ 1 ” か
        {
        ・・・
        }
        ・・・
}
```

【 備考 】 (A) モニター変数の配列インデクスに変数を指定する事は出来ません。

(B) デジタル I/O 番号はコントローラにより最大値が異なります。
存在しないデジタル I/O 番号を指定した場合、アラームが発生します。

(C) デジタル I/O 番号の割り当ては、デジタル I/O の 1 枚目が 0 番、2 枚目が 1 番・・・となります。

■ DO [※1]

【 説明 】 デジタル I/O の出力データを取得します。

【 配列 】 ※1 第 1 のインデクス : デジタル I/O 番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        while ( 1 )
        {
        ・・・
        RS [ 10 ] = DO [ 1 ];          //デジタル I/O_1 番のデータを RS [ 10 ]に設定
        ・・・
        }
}
```

【 備考 】 (A) モニター変数の配列インデクスに変数を指定する事は出来ません。

(B) デジタル I/O 番号はコントローラにより最大値が異なります。
存在しないデジタル I/O 番号を指定した場合、アラームが発生します。

(C) デジタル I/O 番号の割り当ては、デジタル I/O の 1 枚目が 0 番、2 枚目が 1 番・・・となります。

■ AI [※1]

【 説明 】 アナログ I/O の入力データを取得します。

【 配列 】 ※1 第 1 のインデクス : アナログ I/O チャンネル番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        if ( AI[0] >= 1024 )        //アナログ I/O チャンネル 0 番が " 1024 " 以上か
        {
        ・・・
        }
        ・・・
}
```

【 備考 】 (A) モニター変数の配列インデクスに変数を指定する事は出来ません。

(B) アナログ I/O チャンネル番号はコントローラにより最大値が異なります。
存在しないアナログ I/O チャンネル番号を指定した場合、アラームが発生します。

(C) アナログ I/O チャンネル番号の割り当ては、チャンネル 1 番目が 0 番、2 番目が 1 番・・・
となります。

■ AO [※1]

【 説明 】 アナログ I/O の出力データを取得します。

【 配列 】 ※1 第 1 のインデクス : アナログ I/O チャンネル番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        while ( 1 )
        {
        ・・・
        RS [ 10 ] = AO [ 1 ];          //アナログ I/O チャンネル 1 番のデータを RS [ 10 ]に設定
        ・・・
        }
}
```

【 備考 】 (A) モニター変数の配列インデクスに変数を指定する事は出来ません。

(B) アナログ I/O チャンネル番号はコントローラにより最大値が異なります。
存在しないアナログ I/O チャンネル番号を指定した場合、アラームが発生します。

(C) アナログ I/O チャンネル番号の割り当ては、チャンネル 1 番目が 0 番、2 番目が 1 番・・・
となります。

モニター変数（ドライバ系）　詳細

■ SVD_CPLS [※1]

【 説明 】 サーボドライバの現在指令パルスを取得します。（単位：パルス）
SVD_CPLS のデータにはコントローラが内部で保持している原点オフセットデータは含まれません。
サーボドライバの生データを取得します。

【 配列 】 ※1 第 1 のインデクス : コントローラが管理するサーボドライバ番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_CPLS [ 1 ];    //サーボドライバ 1 番の現在指令パルスを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。
（C）サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。
（D）複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の
割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。
・サーボドライバ番号 0 番 : 機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番 : 機構（グループ）1 台目の 2 軸目
:
・サーボドライバ番号 8 番 : 機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番 : 機構（グループ）2 台目の 2 軸目

■ SVD_FPLS [※1]

【 説明 】 サーボドライバの現在実パルスを取得します。（単位：パルス）
SVD_FPLS のデータにはコントローラが内部で保持している原点オフセットデータは含まれません。
サーボドライバの生データを取得します。

【 配列 】 ※1 第 1 のインデクス : コントローラが管理するサーボドライバ番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_FPLS [ 1 ];     //サーボドライバ 1 番の現在実パルスを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。
（C）サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。
（D）複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の
割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。
・サーボドライバ番号 0 番 : 機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番 : 機構（グループ）1 台目の 2 軸目
:
・サーボドライバ番号 8 番 : 機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番 : 機構（グループ）2 台目の 2 軸目

■ SVD_FVEL [※1]

【 説明 】 サーボドライバの現在実速度を取得します。（単位：rpm）

【 配列 】 ※1 第 1 のインデクス ： コントローラが管理するサーボドライバ番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_FVEL [ 1 ];     //サーボドライバ 1 番の現在実速度を RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。

（C）サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。

（D）複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。

・サーボドライバ番号 0 番 ： 機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番 ： 機構（グループ）1 台目の 2 軸目
：
・サーボドライバ番号 8 番 ： 機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番 ： 機構（グループ）2 台目の 2 軸目

■ SVD_FCUR [※1]

【 説明 】　サーボドライバの現在実電流を取得します。（単位：0.01A）

【 配列 】　※1 第 1 のインデクス　：　コントローラが管理するサーボドライバ番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_FCUR [ 1 ];    //サーボドライバ 1 番の現在実電流を RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】

（A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。

（C）サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。

（D）複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の
割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。

・サーボドライバ番号 0 番　：　機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番　：　機構（グループ）1 台目の 2 軸目
　：
・サーボドライバ番号 8 番　：　機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番　：　機構（グループ）2 台目の 2 軸目

（E）機構（グループ）のネットワークタイプにより、電流値ではなくトルク換算値の場合もあります。
詳細はサーボドライバのマニュアルを参照します。ネットワークタイプが SV-NET の場合は、
電流値（単位：0.01A）を取得します。

■ SVD_STS [※1]

【 説明 】 サーボドライバのステータスを取得します。

【 配列 】 ※1 第 1 のインデクス : コントローラが管理するサーボドライバ番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_STS [ 1 ];      //サーボドライバ 1 番の現在ステータスを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】
(A) モニター変数の配列インデクスに変数を指定する事は出来ません。
(B) サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。
(C) サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。
(D) 複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の
割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。
・サーボドライバ番号 0 番 : 機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番 : 機構（グループ）1 台目の 2 軸目
:
・サーボドライバ番号 8 番 : 機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番 : 機構（グループ）2 台目の 2 軸目
(E) 機構（グループ）のネットワークタイプにより、サーボステータスの割り当ては異なります。

（F）SV-NET サーボステータスは下記の内容となります。

BIT_0	：	サーボオン
BIT_1	：	プロファイル動作中
BIT_2	：	インポジション
BIT_3	：	アラーム検出
BIT_4	：	正方向ソフトリミット
BIT_5	：	負方向ソフトリミット
BIT_6	：	トルクリミット
BIT_7	：	速度リミット
BIT_8	：	位置偏差過大
BIT_9	：	予約
BIT_10	：	原点復帰中
BIT_11	：	ゲインセレクト
BIT_12	：	バックアップ電池電圧低下
BIT_13	：	予約
BIT_14	：	予約
BIT_15	：	予約

■ SVD_ALM [※1]

【 説明 】 サーボドライバのアラームコードを取得します。

【 配列 】 ※1 第 1 のインデクス : コントローラが管理するサーボドライバ番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_ALM [ 1 ];      //サーボドライバ 1 番のアラームコードを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 (A) モニター変数の配列インデクスに変数を指定する事は出来ません。

(B) サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。

(C) サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。

(D) 複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の
割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。

・サーボドライバ番号 0 番 : 機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番 : 機構（グループ）1 台目の 2 軸目
:
・サーボドライバ番号 8 番 : 機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番 : 機構（グループ）2 台目の 2 軸目

(E) アラームコードの詳細はサーボドライバのマニュアルを参照します。

■ **SVD_LOAD [※1]** **SV-NET Only**

【 説明 】 サーボドライバの過負荷モニターのデータを取得します。（単位：0.1%）

【 配列 】 ※1 第 1 のインデクス : コントローラが管理するサーボドライバ番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_LOAD [ 1 ];    //サーボドライバ 1 番の過負荷モニターを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。

（C）サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。

（D）複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の
割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。

・サーボドライバ番号 0 番 : 機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番 : 機構（グループ）1 台目の 2 軸目
:
・サーボドライバ番号 8 番 : 機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番 : 機構（グループ）2 台目の 2 軸目

（E）機構（グループ）のネットワークタイプが SV-NET の場合に有効です。

■ SVD_TEMP [※1]　　SV-NET Only

【 説明 】　サーボドライバの内部温度のデータを取得します。（単位：0.1℃）

【 配列 】　※1 第 1 のインデクス　　：　　コントローラが管理するサーボドライバ番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_TEMP [ 1 ];    //サーボドライバ 1 番の内部温度を RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】
（A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。
（C）サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。
（D）複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の
割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。
・サーボドライバ番号 0 番　：　機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番　：　機構（グループ）1 台目の 2 軸目
：
・サーボドライバ番号 8 番　：　機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番　：　機構（グループ）2 台目の 2 軸目
（E）機構（グループ）のネットワークタイプが SV-NET の場合に有効です。

■ SVD_PWR [※1]　　SV-NET Only

【 説明 】　サーボドライバの駆動電源電圧のデータを取得します。（単位：0.1V）

【 配列 】　※1 第 1 のインデクス　　：　　コントローラが管理するサーボドライバ番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SVD_PWR [ 1 ];      //サーボドライバ 1 番の駆動電源電圧を RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】

（A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）サーボドライバ番号はコントローラにより最大値が異なります。
存在しないサーボドライバ番号を指定した場合、アラームが発生します。

（C）サーボドライバ番号の既定の設定は、ネットワーク ID 番号が若い順に 1 軸目、2 軸目・・・
1 軸目がサーボドライバ番号 0 番、2 軸目がサーボドライバ番号 1 番・・・となります。

（D）複数の機構（グループ）を持つコントローラのサーボドライバ番号は、既定の設定で下記の
割り当てとなります。仮に機構（グループ）0 番の最大保有軸数が 8 軸の場合。

・サーボドライバ番号 0 番　：　機構（グループ）1 台目の 1 軸目
・サーボドライバ番号 1 番　：　機構（グループ）1 台目の 2 軸目
　　：
・サーボドライバ番号 8 番　：　機構（グループ）2 台目の 1 軸目
・サーボドライバ番号 9 番　：　機構（グループ）2 台目の 2 軸目

（E）機構（グループ）のネットワークタイプが SV-NET の場合に有効です。

モニター変数（コントローラ系）　詳細

■ MCH_CPLS [※1] [※2]

【 説明 】 機構（グループ）内各軸の現在指令パルスを取得します。（単位：パルス）
MCH_CPLS のデータにはコントローラが内部で保持している原点オフセットデータが含まれます。

【 配列 】 ※1 第 1 のインデクス : 機構（グループ）番号
※2 第 2 のインデクス : 機構（グループ）内軸番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_CPLS [ 0 ] [ 1 ];        //機構（グループ）1 台目 2 軸目の
        ・・・                                    //現在指令パルスを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_CPLS [0] [※]
機構 2 台目：MCH_CPLS [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_CPLS [0] [0]
機構 1 台目 2 軸目：MCH_CPLS [0] [1]

■ MCH_FPLS [※1] [※2]

【 説明 】　機構（グループ）内各軸の現在実パルスを取得します。（単位：パルス）
MCH_FPLS のデータにはコントローラが内部で保持している原点オフセットデータが含まれます。

【 配列 】　※1 第 1 のインデクス　：　機構（グループ）番号
※2 第 2 のインデクス　：　機構（グループ）内軸番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_FPLS [ 0 ] [ 1 ];        //機構（グループ）1 台目 2 軸目の
        ・・・                                    //現在実パルスを RS [ 20 ]に設定
        }
}
```

【 備考 】
（A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_FPLS [0] [※]
機構 2 台目：MCH_FPLS [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_FPLS [0] [0]
機構 1 台目 2 軸目：MCH_FPLS [0] [1]

■ MCH_FVEL [※1] [※2]

【 説明 】 機構（グループ）内各軸の現在実速度を取得します。（単位：rpm）
MCH_FVEL のデータは SVD_FVEL のデータと同じです。

【 配列 】 ※1 第 1 のインデクス : 機構（グループ）番号
※2 第 2 のインデクス : 機構（グループ）内軸番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_FVEL [ 0 ] [ 1 ];        //機構（グループ）1 台目 2 軸目の
        ・・・                                    //現在実速度を RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_FVEL [0] [※]
機構 2 台目：MCH_FVEL [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_FVEL [0] [0]
機構 1 台目 2 軸目：MCH_FVEL [0] [1]

■ MCH_FCUR [※1] [※2]

【 説明 】　機構（グループ）内各軸の現在実電流を取得します。（単位：0.01A）
MCH_FCUR のデータは SVD_FCUR のデータと同じです。

【 配列 】　※1 第 1 のインデクス　：　機構（グループ）番号
※2 第 2 のインデクス　：　機構（グループ）内軸番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_FCUR [ 0 ] [ 1 ];        //機構（グループ）1 台目 2 軸目の
        ・・・                                    //現在実電流を RS [ 20 ]に設定
        }
}
```

【 備考 】
（A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_FCUR [0] [※]
機構 2 台目：MCH_FCUR [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_FCUR [0] [0]
機構 1 台目 2 軸目：MCH_FCUR [0] [1]

■ MCH_FSPD [※1] [※2]

【 説明 】　機構（グループ）内各軸の現在実速度を取得します。（単位：速度指令単位）
MCH_FVEL とは異なり速度指令単位でデータを取得します。

【 配列 】　※1 第 1 のインデクス　：　機構（グループ）番号
※2 第 2 のインデクス　：　機構（グループ）内軸番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_FSPD [ 0 ] [ 1 ];        //機構（グループ）1 台目 2 軸目の
        ・・・                                    //現在実速度を RS [ 20 ]に設定
        }
}
```

【 備考 】
（A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_FSPD [0] [※]
機構 2 台目：MCH_FSPD [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_FSPD [0] [0]
機構 1 台目 2 軸目：MCH_FSPD [0] [1]

■ MCH_CPOS [※1] [※2]

【 説明 】　機構（グループ）内各軸の現在指令位置を取得します。（単位：位置指令単位）
MCH_CPLS とは異なり位置指令単位でデータを取得します。

【 配列 】　※1 第 1 のインデクス　：　機構（グループ）番号
※2 第 2 のインデクス　：　機構（グループ）内軸番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_CPOS [ 0 ] [ 1 ];        //機構（グループ）1 台目 2 軸目の
        ・・・                                    //現在指令位置を RS [ 20 ]に設定
        }
}
```

【 備考 】
（A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_CPOS [0] [※]
機構 2 台目：MCH_CPOS [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_CPOS [0] [0]
機構 1 台目 2 軸目：MCH_CPOS [0] [1]

■ MCH_FPOS [※1] [※2]

【 説明 】 機構（グループ）内各軸の現在実位置を取得します。（単位：位置指令単位）
MCH_FPLS とは異なり位置指令単位でデータを取得します。

【 配列 】 ※1 第 1 のインデクス : 機構（グループ）番号
※2 第 2 のインデクス : 機構（グループ）内軸番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_FPOS [ 0 ] [ 1 ];        //機構（グループ）1 台目 2 軸目の
        ・・・                                    //現在実位置を RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_FPOS [0] [※]
機構 2 台目：MCH_FPOS [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_FPOS [0] [0]
機構 1 台目 2 軸目：MCH_FPOS [0] [1]

■ MCH_SVSTS [※1] [※2]

【 説明 】　機構（グループ）内各軸の現在ステータスを取得します。
MCH_SVSTS のデータは SVD_STS のデータと同じです。

【 配列 】　※1 第 1 のインデクス　：　機構（グループ）番号
※2 第 2 のインデクス　：　機構（グループ）内軸番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_SVSTS [ 0 ] [ 1 ];      //機構（グループ）1 台目 2 軸目の
        ・・・                                  //現在ステータスを RS [ 20 ]に設定
        }
}
```

【 備考 】
（A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_SVSTS [0] [※]
機構 2 台目：MCH_SVSTS [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_SVSTS [0] [0]
機構 1 台目 2 軸目：MCH_SVSTS [0] [1]

■ MCH_SVALM [※1] [※2]

【 説明 】　機構（グループ）内各軸のアラームコードを取得します。
MCH_SVALM のデータは SVD_ALM のデータと同じです。

【 配列 】　※1 第 1 のインデクス　：　機構（グループ）番号
※2 第 2 のインデクス　：　機構（グループ）内軸番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_SVALM [ 0 ] [ 1 ];      //機構（グループ）1 台目 2 軸目の
        ・・・                                  //アラームコードを RS [ 20 ]に設定
        }
}
```

【 備考 】
（A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_SVALM [0] [※]
機構 2 台目：MCH_SVALM [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_SVALM [0] [0]
機構 1 台目 2 軸目：MCH_SVALM [0] [1]

■ MCH_JSTS [※1] [※2]

【 説明 】　機構（グループ）内各軸の現在機構ステータスを取得します。

【 配列 】　※1 第 1 のインデクス　：　機構（グループ）番号
※2 第 2 のインデクス　：　機構（グループ）内軸番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_JSTS [ 0 ] [ 1 ];        //機構（グループ）1 台目 2 軸目の
        ・・・                                    //現在機構ステータスを RS [ 20 ]に設定
        }
}
```

【 備考 】
（A）モニター変数の配列インデクスに変数を指定する事は出来ません。
（B）機構（グループ）番号、機構内軸番号はコントローラにより最大値が異なります。
存在しない機構番号、機構内軸番号を指定した場合、アラームが発生します。
（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_JSTS [0] [※]
機構 2 台目：MCH_JSTS [1] [※]
（D）機構（グループ）内軸番号の既定の設定は、機構（グループ）の属するネットワーク ID の
若い順に 1 軸目、2 軸目・・・となり、1 軸目が 0 番、2 軸目が 1 番となります。
従いまして、機構（グループ）内軸番号のインデクスは下記のようになります。
機構 1 台目 1 軸目：MCH_JSTS [0] [0]
機構 1 台目 2 軸目：MCH_JSTS [0] [1]

（E）各軸機構ステータスは下記の内容となります。

BIT_0	：	補間計算中
BIT_1	：	加減速処理中（指令払い出し未完了）
BIT_2	：	軸動作中（指令払い出し未完了またはインポジションでない）
BIT_3	：	原点復帰中
BIT_4	：	速度リミット検出
BIT_5	：	正方向ソフトリミット検出
BIT_6	：	負方向ソフトリミット検出
BIT_7	：	サーボオンステータス
BIT_8	：	アラーム発生中
BIT_9	：	予約
BIT_10	：	予約
BIT_11	：	予約
BIT_12	：	動作命令の途中で停止命令入力
BIT_13	：	正方向ストロークリミット検出
BIT_14	：	負方向ストロークリミット検出
BIT_15	：	原点復帰済み

■ MCH_ALM [※1]

【 説明 】 機構（グループ）のアラームコードを取得します。

【 配列 】 ※1 第 1 のインデクス : 機構（グループ）番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_ALM [ 0 ];          //機構（グループ）1 台目の
        ・・・                               //アラームコードを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）機構（グループ）番号はコントローラにより最大値が異なります。
存在しない機構番号を指定した場合、アラームが発生します。

（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_ALM [0]
機構 2 台目：MCH_ALM [1]

■ MCH_STS [※1]

【 説明 】 機構（グループ）の現在機構ステータスを取得します。

【 配列 】 ※1 第 1 のインデクス : 機構（グループ）番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = MCH_STS [ 0 ];          //機構（グループ）1 台目の
        ・・・                               //現在機構ステータスを RS [ 20 ]に設定
        }
}
```

【 補足 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）機構（グループ）番号はコントローラにより最大値が異なります。
存在しない機構番号を指定した場合、アラームが発生します。

（C）機構（グループ）番号の既定の設定は、機構の 1 台目が 0 番、2 台目が 1 番・・・となります。
従いまして、機構（グループ）番号のインデクスは下記のようになります。
機構 1 台目：MCH_STS [0]
機構 2 台目：MCH_STS [1]

（D）機構ステータスは下記の内容となります。

BIT_0	：	補間計算中
BIT_1	：	加減速処理中（指令払い出し未完了）
BIT_2	：	軸動作中（指令払い出し未完了またはインポジションでない）
BIT_3	：	原点復帰中
BIT_4	：	速度リミット検出
BIT_5	：	正方向ソフトリミット検出
BIT_6	：	負方向ソフトリミット検出
BIT_7	：	予約
BIT_8	：	アラーム発生中
BIT_9	：	ワーニング発生中
BIT_10	：	予約
BIT_11	：	予約
BIT_12	：	動作命令の途中で停止命令入力
BIT_13	：	正方向ストロークリミット検出
BIT_14	：	負方向ストロークリミット検出
BIT_15	：	原点復帰済み

モニター変数（ステータス系）　詳細

■ TIM [※1]

【 説明 】 タイマの現在値を取得します。

【 配列 】 ※1 第 1 のインデクス : タイマ番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = TIM [ 4 ];          //タイマ 4 番の値を RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 (A) モニター変数の配列インデクスに変数を指定する事は出来ません。

(B) タイマ番号はコントローラにより最大値が異なります。

存在しないタイマ番号を指定した場合、アラームが発生します。

例えば SVCC、SVCE は API 用に 32 個のタイマを保有します。

その場合のタイマ番号の割り当ては下記のようになります。

タイマ 0 番：TIM [0]

タイマ 1 番：TIM [1]

:

:

タイマ 31 番：TIM [31] //計 32 個

■ TASK_STS [※1]

【 説明 】　タイマの現在起動状態を取得します。
タスク起動中なら“1”、タスク停止中なら“0”を取得します。

【 配列 】　※1 第1のインデクス　：　タスク番号

【 属性 】　読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = TASK_STS [ 4 ];          //タスク4番の起動状態を RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】　(A) モニター変数の配列インデクスに変数を指定する事は出来ません。
(B) タスク番号はコントローラにより最大値が異なります。
存在しないタスク番号を指定した場合、アラームが発生します。
例えば SVCC、SVCE は API 用に 8 個のタスクを保有します。
その場合のタスク番号の割り当ては下記のようになります。
タスク 0 番：TASK_STS [0]　//メインタスク
タスク 1 番：TASK_STS [1]　//タスク 1 番
：
：
タスク 7 番：TASK_STS [7]　//計 8 個

■ RS_STS

【 説明 】 RS232C の現在通信ステータスを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        while( RS_STS & 0x30 ) { };          //RS232C 通信ステータスを判定
        ・・・                                //データ送受信中なら待ち
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）RS232C 通信ステータスは下記の内容となります。

BIT_0	：	自動送受信モード有効
BIT_1	：	RS コマンド実行中
BIT_2	：	予約
BIT_3	：	エラー発生中
BIT_4	：	データ送信中
BIT_5	：	データ受信中
BIT_6	：	予約
BIT_7	：	予約

■ RS_ERR

【 説明 】 RS232C 通信のエラーコードを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = RS_ERR;          //RS232C 通信エラーコードを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 (A) モニター変数の配列インデクスに変数を指定する事は出来ません。

(B) エラーコードの詳細はエラーコード一覧を参照します。

■ RS_ECNT

【 説明 】 RS232C 通信のエラーカウントを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = RS_ECNT;          //RS232C 通信エラーカウントを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ CC_STS

【 説明 】 CC Link の現在通信ステータスを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( CC_STS & 0x08 ) { ・・・ };        //CC Link 通信ステータスを判定
        {
        ・・・                                  //エラー発生中なら処理を実行
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）CC Link 通信ステータスは下記の内容となります。

BIT_0	：	送信可能
BIT_1	：	予約
BIT_2	：	予約
BIT_3	：	エラー発生中
BIT_4	：	予約
BIT_5	：	予約
BIT_6	：	予約
BIT_7	：	予約

■ CC_ERR

【 説明 】 CC Link 通信のエラーコードを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = CC_ERR;          //CC Link 通信エラーコードを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）エラーコードの詳細はエラーコード一覧を参照します。

■ CC_ECNT

【 説明 】 CC Link 通信のエラーカウントを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = CC_ECNT;          //CC Link 通信エラーカウントを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ DV_STS

【 説明 】 DeviceNet の現在通信ステータスを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( CC_STS & 0x08 ) { ・・・ };        //DeviceNet 通信ステータスを判定
        {
        ・・・                                  //エラー発生中なら処理を実行
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

（B）DeviceNet 通信ステータスは下記の内容となります。

BIT_0	：	正常通信中
BIT_1	：	予約
BIT_2	：	予約
BIT_3	：	エラー発生中
BIT_4	：	予約
BIT_5	：	予約
BIT_6	：	予約
BIT_7	：	予約

■ DV_ERR

【 説明 】 DeviceNet 通信のエラーコードを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = DV_ERR;          //DeviceNet 通信エラーコードを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 (A) モニター変数の配列インデクスに変数を指定する事は出来ません。

(B) エラーコードの詳細はエラーコード一覧を参照します。

■ DV_ECNT

【 説明 】 DeviceNet 通信のエラーカウントを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = DV_ECNT;          //DeviceNet 通信エラーカウントを RS [ 20 ]に設定
        ・・・
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ COM_AUTO [※1]

【 説明 】 COM ポートの現在自動送受信状態を取得します。
自動送受信中なら “ 1 ”、自動送受信停止中なら “ 0 ” を取得します。

【 配列 】 ※1 第 1 のインデクス : COM ポート番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = COM_AUTO [ 0 ];          //COM ポート 0 番の
        ・・・                                //自動送受信状態を RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ COM_STS [※1]

【 説明 】 COM ポートの現在自動送受信ステータスを取得します。
正常完了なら “ 1 ”、異常終了なら負の数を取得します。
異常コード：-10（オーバーランエラー）
異常コード：-1xxx（タイムアウトエラー、xxx には受信または送信済みデータ数）

【 配列 】 ※1 第 1 のインデクス : COM ポート番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = COM_STS [ 0 ];          //COM ポート 0 番の
        ・・・                               //自動送受信ステータスを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ COM_OERR [※1]

【 説明 】 COM ポートのオーバーランエラーカウンタを取得します。
電源投入からの累積カウンタです。

【 配列 】 ※1 第 1 のインデクス : COM ポート番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = COM_OERR [ 0 ];          //COM ポート 0 番の
        ・・・                                //オーバーランエラーカウンタを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ **COM_FERR [※1]**

【 説明 】 COM ポートのフレーミングエラーカウンタを取得します。
電源投入からの累積カウンタです。

【 配列 】 ※1 第 1 のインデクス : COM ポート番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = COM_FERR [ 0 ];          //COM ポート 0 番の
        ・・・                                //フレーミングエラーカウンタを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ COM_PERR [※1]

【 説明 】 COM ポートのパリティエラーカウンタを取得します。
電源投入からの累積カウンタです。

【 配列 】 ※1 第 1 のインデクス : COM ポート番号

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = COM_PERR [ 0 ];          //COM ポート 0 番の
        ・・・                                ///パリティエラーカウンタを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ SV_TECREC

【 説明 】 SV-NET 通信の CAN エラーカウンタを取得します。

【 配列 】 無し

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SV_TECREC;          //SV-NET 通信 CAN エラーカウンタを
        ・・・                          //RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ SV_NERR [※1]

【 説明 】 SV-NET 通信の受信未完了エラーカウンタを取得します。
電源投入からの累積カウンタです。

【 配列 】 ※1 第 1 のインデクス : 軸番号（SV-NET に接続された）

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SV_NERR[2];          //2 軸目の SV-NET 通信受信未完了
        ・・・                            //エラーカウンタを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ SV_TERR [※1]

【 説明 】 SV-NET 通信の送信未完了エラーカウンタを取得します。
電源投入からの累積カウンタです。

【 配列 】 ※1 第 1 のインデクス : 軸番号（SV-NET に接続された）

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SV_TERR[2];          //2 軸目の SV-NET 通信送信未完了
        ・・・                            //エラーカウンタを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ SV_OERR [※1]

【 説明 】 SV-NET 通信の受信オーバーランエラーカウンタを取得します。
電源投入からの累積カウンタです。

【 配列 】 ※1 第 1 のインデクス : 軸番号（SV-NET に接続された）

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SV_OERR[2];          //2 軸目の SV-NET 通信受信オーバーラン
        ・・・                            //エラーカウンタを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

■ SV_VERR [※1]

【 説明 】 SV-NET 通信の SV-NET エラーコード受信カウンタを取得します。
電源投入からの累積カウンタです。

【 配列 】 ※1 第 1 のインデクス : 軸番号（SV-NET に接続された）

【 属性 】 読み込み専用

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 20 ] = SV_VERR[2];          //2 軸目の SV-NET 通信エラーコード受信
        ・・・                            //カウンタを RS [ 20 ]に設定
        }
}
```

【 備考 】 （A）モニター変数の配列インデクスに変数を指定する事は出来ません。

ネットワーク変数　詳細

■ RS [※1]

【 説明 】 RS232C 通信ネットワーク変数の値を取得・設定します。

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 1 ] = SVD_STS [ 0 ];          //サーボドライバ 1 軸目の現在ステータスを
        ・・・                              //RS [ 1 ]に設定
        }
}
```

【 備考 】 （A）RS232C の通信設定はコントローラのパラメータ設定に従います。

■ CC_RY [※1]

【 説明 】 CC Link 接点データ格納用ネットワーク変数の値を取得・設定します。
（方向：マスタ→コントローラ）

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( CC_RY [ 1 ] & 0x02 )       //CC_RY [ 1 ]のビット 1 が“ 1 ”か
        {
        ・・・                          //“ 1 ”なら処理を実行
        }
}
```

【 備考 】 （A）既定の設定では、CC_RY の有効な範囲は CC_RY [0] 〜 CC_RY [7]までです。
（B）詳細は CC Link 通信設定マニュアルを参照します。

■ CC_RX [※1]

【 説明 】 CC Link 接点データ格納用ネットワーク変数の値を取得・設定します。
（方向：マスタ←コントローラ）

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        CC_RX [ 1 ] = DI [ 0 ];          //デジタル I/O_0 番目の入力状態を
        ・・・                            //CC_RX [ 1 ]に設定
        }
}
```

【 備考 】 （A）既定の設定では、CC_RX の有効な範囲は CC_RX [0] 〜 CC_RX [7]までです。
（B）詳細は CC Link 通信設定マニュアルを参照します。

■ CC_RWW [※1]

【 説明 】 CC Link レジスタ格納用ネットワーク変数の値を取得・設定します。
（方向：マスタ→コントローラ）

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( CC_RY [ 1 ] & 0x02 )          //CC_RY [ 1 ]のビット 1 が“ 1 ”か
        {
        ・・・                              //“ 1 ”なら処理を実行
        TableMovaP ( 0, 0x01, CC_RWW [ 5 ] );   //CC_RWW [ 5 ]で指定された
        InposA ( 0, 0x01 );                     //テーブル番号で位置決め
        }
}
```

【 備考 】 （A）既定の設定では、CC_RWW の有効な範囲は CC_RWW [0] 〜 CC_RWW [15]までです。
（B）詳細は CC Link 通信設定マニュアルを参照します。

■ CC_RWR [※1]

【 説明 】 CC Link レジスタ格納用ネットワーク変数の値を取得・設定します。
（方向：マスタ←コントローラ）

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        CC_RWR [ 10 ] = SVD_STS [ 0 ];        //サーボドライバ 1 軸目の現在ステータスを
        ・・・                                 //CC_RWR [ 10 ]に設定
        }
}
```

【 備考 】 （A）既定の設定では、CC_RWR の有効な範囲は CC_RWR [0] 〜 CC_RWR [15]までです。
（B）詳細は CC Link 通信設定マニュアルを参照します。

■ DV_IN [※1]

【 説明 】 DeviceNet レジスタ格納用ネットワーク変数の値を取得・設定します。
（方向：マスタ→コントローラ）

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( DV_IN [ 1 ] & 0x02 )          //DV_IN [ 1 ]のビット 1 が “ 1 ” か
        {
        ・・・                              // “ 1 ” なら処理を実行
        TableMovaP ( 0, 0x01, DV_IN [ 5 ] );      //DV_IN [ 5 ]で指定された
        InposA ( 0, 0x01 );                       //テーブル番号で位置決め
        }
}
```

【 備考 】 （A）既定の設定では、DV_IN の有効な範囲は DV_IN [0] 〜 DV_IN [24]までです。
（B）詳細は DeviceNet 通信設定マニュアルを参照します。

■ **DV_OUT [※1]**

【 説明 】 DeviceNet レジスタ格納用ネットワーク変数の値を取得・設定します。
（方向：マスタ←コントローラ）

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        DV_OUT [ 10 ] = SVD_STS [ 0 ];        //サーボドライバ 1 軸目の現在ステータスを
        ・・・                                 //DV_OUT [ 10 ]に設定
        }
}
```

【 備考 】 （A）既定の設定では、DV_OUT の有効な範囲は DV_OUT [0] 〜 DV_OUT [24]までです。
（B）詳細は DeviceNet 通信設定マニュアルを参照します。

■ TCP_IP [※1]

【 説明 】 TCP／IP レジスタ格納用ネットワーク変数の値を取得・設定します。

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( TCP_IP [ 1 ] & 0x02 )          //TCP_IP [ 1 ]のビット 1 が “ 1 ” か
        {
        ・・・                                          // “ 1 ” なら処理を実行
        TableMovaP ( 0, 0x01, TCP_IP [ 5 ] );   //TCP_IP [ 5 ]で指定された
        InposA ( 0, 0x01 );                             //テーブル番号で位置決め
        }
}
```

【 備考 】 （A）TCP_IP ネットワーク変数は TCP／IP 通信のデータを取得・設定します。

■ **ECAT_IN [※1]**

【 説明 】 EtherCAT レジスタ格納用ネットワーク変数の値を取得・設定します。
（方向：スレーブ→コントローラ）※EtherCAT はコントローラがマスタとなります。

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( ECAT_IN [ 1 ] & 0x02 )       //ECAT_IN [ 1 ]のビット 1 が “ 1 ” か
        {
        ・・・                             // “ 1 ” なら処理を実行
        TableMovaP ( 0, 0x01, ECAT_IN [ 5 ] );    //ECAT_IN [ 5 ]で指定された
        InposA ( 0, 0x01 );                       //テーブル番号で位置決め
        }
}
```

【 備考 】 （A）ECAT_IN ネットワーク変数は EtherCAT リアルタイムメッセージのデータを取得・設定します。
変数の割り付けや使用方法は EtherCAT 通信設定マニュアルを参照します。

■ ECAT_OUT [※1]

【 説明 】 EtherCAT レジスタ格納用ネットワーク変数の値を取得・設定します。
（方向：スレーブ←コントローラ)）※EtherCAT はコントローラがマスタとなります。

【 配列 】 ※1 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        ECAT_OUT [ 10 ] = DI [ 0 ];          //デジタル I/O_0 番の入力状態を
        ・・・                                //ECAT_OUT [ 10 ]に設定
        }
}
```

【 備考 】 (A)ECAT_OUT ネットワーク変数は EtherCAT リアルタイムメッセージのデータを取得･設定します。
変数の割り付けや使用方法は EtherCAT 通信設定マニュアルを参照します。

■ **DP_RAM [※1] [※2]**

【 説明 】 外部拡張 RAM 格納用ネットワーク変数の値を取得・設定します。

【 配列 】 ※1 第 1 のインデクス : 外部拡張 RAM のオフセット
※2 第 1 のインデクス : ネットワーク変数のオフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        RS [ 10 ] = DP_RAM [ 0 ] [ 100 ];        //DP_RAM [ 0 ] [ 100 ]の値を
        ・・・                                    //RS [ 10 ]に設定
        }
}
```

【 備考 】 (A) 外部拡張 RAM ネットワーク変数の割り付けや使用方法は、個別の設定マニュアルを参照します。

テーブル変数　詳細

■ SVC_TBL [※1] [※2] [※3]

【 説明 】 テーブル変数の値を取得・設定します。

【 配列 】 ※1 第 1 のインデクス : テーブル変数大分類（テーブル種別）
※2 第 2 のインデクス : テーブル変数中分類（サーボドライバ番号）
※3 第 3 のインデクス : テーブル変数オフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        SVC_TBL [ 0 ] [ 1 ] [ 0 ] = RS [ 0 ];        //RS [ 0 ]のデータを
        ・・・                                        //テーブル変数 SVC_TBL [ 0 ] [ 1 ] [ 0 ]に設定
        }
}
```

【 備考 】 （A）SVC_TBL 変数を使用する事により、コントローラが保有する全てのテーブル変数へ
アクセスする事が可能です。
（B）コントローラが保有するテーブル変数のサイズを超えてアクセスした場合、
アラームが発生します。
（C）テーブル変数大分類は下記のように割り当てられています。
SVC_TBL [0] [※] [※] ： 位置テーブル
SVC_TBL [1] [※] [※] ： 速度テーブル
SVC_TBL [2] [※] [※] ： 時間テーブル
SVC_TBL [3] [※] [※] ： 加減速テーブル //既定の設定では第 1 インデクスは最大 3
（D）テーブル変数中分類は下記のように割り当てられています。
SVC_TBL [※] [0] [※] ： サーボドライバ番号 1 軸目
SVC_TBL [※] [1] [※] ： サーボドライバ番号 2 軸目
SVC_TBL [※] [2] [※] ： サーボドライバ番号 3 軸目
： //第 2 インデクスの最大値はコントローラが保有可能な
//最大軸数まで設定可能

■ POS_TBL [※1] [※2]

【 説明 】 位置テーブル変数の値を取得・設定します。

POS_TBL 変数は SVC_TBL [0] [※] [※]と同じデータです。

【 配列 】 ※1 第 1 のインデクス : テーブル変数中分類（サーボドライバ番号）

※2 第 2 のインデクス : テーブル変数オフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        POS_TBL [ 0 ] [ 1 ] = RS [ 0 ];          //RS [ 0 ]のデータを
        ・・・                                    //テーブル変数 POS_TBL [ 0 ] [ 1 ]に設定
        }
}
```

【 備考 】 （A）コントローラが保有する位置テーブル変数のサイズを超えてアクセスした場合、
アラームが発生します。

（B）テーブル変数中分類は下記のように割り当てられています。

POS_TBL [0] [※] ： サーボドライバ番号 1 軸目

POS_TBL [1] [※] ： サーボドライバ番号 2 軸目

POS_TBL [2] [※] ： サーボドライバ番号 3 軸目

： //第 1 インデクスの最大値はコントローラが保有可能な

//最大軸数まで設定可能

■ VEL_TBL [※1] [※2]

【 説明 】 速度テーブル変数の値を取得・設定します。
VEL_TBL 変数は SVC_TBL [1] [※] [※]と同じデータです。

【 配列 】 ※1 第 1 のインデクス : テーブル変数中分類（サーボドライバ番号）
※2 第 2 のインデクス : テーブル変数オフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        VEL_TBL [ 0 ] [ 1 ] = RS [ 0 ];          //RS [ 0 ]のデータを
        ・・・                                    //テーブル変数 VEL_TBL [ 0 ] [ 1 ]に設定
        }
}
```

【 備考 】 （A）コントローラが保有する速度テーブル変数のサイズを超えてアクセスした場合、
アラームが発生します。
（B）テーブル変数中分類は下記のように割り当てられています。
VEL_TBL [0] [※] ： サーボドライバ番号 1 軸目
VEL_TBL [1] [※] ： サーボドライバ番号 2 軸目
VEL_TBL [2] [※] ： サーボドライバ番号 3 軸目
： //第 1 インデクスの最大値はコントローラが保有可能な
//最大軸数まで設定可能

■ TIM_TBL [※1] [※2]

【 説明 】　時間テーブル変数の値を取得・設定します。
TIM_TBL 変数は SVC_TBL [2] [※] [※]と同じデータです。

【 配列 】　※1 第 1 のインデクス　：　テーブル変数中分類（サーボドライバ番号）
※2 第 2 のインデクス　：　テーブル変数オフセット

【 属性 】　読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        TIM_TBL [ 0 ] [ 1 ] = RS [ 0 ];          //RS [ 0 ]のデータを
        ・・・                                    //テーブル変数 TIM_TBL [ 0 ] [ 1 ]に設定
        }
}
```

【 備考 】　（A）コントローラが保有する時間テーブル変数のサイズを超えてアクセスした場合、
アラームが発生します。
（B）テーブル変数中分類は下記のように割り当てられています。
TIM_TBL [0] [※]　：　サーボドライバ番号 1 軸目
TIM_TBL [1] [※]　：　サーボドライバ番号 2 軸目
TIM_TBL [2] [※]　：　サーボドライバ番号 3 軸目
：　//第 1 インデクスの最大値はコントローラが保有可能な
//最大軸数まで設定可能

■ ACC_TBL [※1] [※2]

【 説明 】 加減速テーブル変数の値を取得・設定します。
ACC_TBL 変数は SVC_TBL [3] [※] [※]と同じデータです。

【 配列 】 ※1 第 1 のインデクス : テーブル変数中分類（サーボドライバ番号）
※2 第 2 のインデクス : テーブル変数オフセット

【 属性 】 読み込み・書き込み可能

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        while ( 1 )
        {
        ・・・
        ACC_TBL [ 0 ] [ 1 ] = RS [ 0 ];          //RS [ 0 ]のデータを
        ・・・                                    //テーブル変数 ACC_TBL [ 0 ] [ 1 ]に設定
        }
}
```

【 備考 】 （A）コントローラが保有する加減速テーブル変数のサイズを超えてアクセスした場合、
アラームが発生します。
（B）テーブル変数中分類は下記のように割り当てられています。
ACC_TBL [0] [※] ： サーボドライバ番号 1 軸目
ACC_TBL [1] [※] ： サーボドライバ番号 2 軸目
ACC_TBL [2] [※] ： サーボドライバ番号 3 軸目
： //第 1 インデクスの最大値はコントローラが保有可能な
//最大軸数まで設定可能

6．Motion Designer API 関数一覧

Motoion Designer ではコントローラの専用コマンドを直接駆動可能な専用 API 関数を準備しています。

■ システム命令

分類	関数名	説明
システム命令	void Nop ( )	無効果
	void AlarmReset (int mch)	アラームリセット
	void SmoothingSet (int mch, int setup, int t1, int t2)	加減速時定数設定
	void ParameterSet (int cls_no, int grp_no, int id_no, int data)	パラメータ設定
	void ParameterGet (int cld_no, int grp_no, int id_no, int data_num, int &var_adr)	パラメータ取得
	void ParameterSave ( )	パラメータ保存
	void MonitorGet(int cls_no, int grp_no, int id_no, int data_num, int &var_adr)	モニターデータ取得
	End ( )	実行タスク終了

■ データ命令

分類	関数名	説明
データ命令	void copy (int &var_adr1, int &var_adr2, size)	データ転送
	int abs (int iop1)	絶対値
	int swap (int op1)	下位バイト、上位バイト入れ替え
	int swap2 (int op1, int op2)	下位バイト、上位バイト合成

■ 浮動小数点演算命令

分類	関数名	説明
浮動小数点演算命令	double acos (doube op1)	逆余弦
	double asin (doube op1)	逆正弦
	double atan (doube op1)	逆正接
	double atan2 (doube op1, double op2)	除算した結果の逆正接
	double cos (doube op1)	ラジアン値の余弦
	double sin (doube op1)	ラジアン値の正弦
	double tan (doube op1)	ラジアン値の正接
	double cosh (doube op1)	双曲線余弦
	double sinh (doube op1)	双曲線正弦
	double tanh (doube op1)	双曲線正接
	double exp (doube op1)	指数関数
	double frexp (doube op1, int &op2)	[0.5, 1.0]の値と 2 のべき乗の積
	double ldexp (doube op1, int op2)	2 のべき乗の乗算
	double log (doube op1)	自然対数
	double log10 (doube op1)	10 を底とする対数
	double modf (doube op1, double &op2)	整数部分と小数部分に分解
	double pow (doube op1, double op2)	べき乗
	double sqrt (doube op1)	正の平方根
	double ceil (doube op1)	小数点以下切り上げ
	double fabs (doube op1)	絶対値
	double floor (doube op1)	小数点以下切り捨て
	double fmod (doube op1, double op2)	剰余

■ タスク命令

分類	関数名	説明
タスク命令	void TaskStart (int func_name)	タスク起動
	int TaskId (void)	タスク番号取得
	int TaskStatus (int task_no)	タスクステータス取得
	void TaskReStart (int task_no)	タスク再起動
	void TaskStep (int task_no)	タスクステップ実行
	void TaskWait (int msec)	タスク実行待ち
	void TaskEnd (int task_no)	指定タスク終了

■ タイマー命令

分類	関数名	説明
タイマー命令	void TimerSet (int timer, int init)	タイマー初期化
	void Waitmsec (int timer, int msec)	ウェイト

■ Ｉ／Ｏ命令

分類	関数名	説明
Ｉ／Ｏ命令	void BitOn (int dio, int bit)	ビットセット
	void BitOff (int dio, int bit)	ビットクリア
	int BitIn (int dio, int mask)	ビット
	void DioOut (int dio, int data)	DIO 出力
	int DioIn (int dio)	DIO 入力
	void AioOut (int aio, int data)	アナログ出力
	int AioIn (int aio)	アナログ入力

■ PASS 命令

分類	関数名	説明
PASS 命令	void PassM (int mch)	機構内全軸補間計算待ち
	void DecelM (int mch)	機構内全軸払い出し完了待ち
	void InposM (int mch)	機構内全軸インポジション待ち
	void OrgM (int mch)	機構内全軸原点復帰完了待ち
	void PassA (int mch, int setup)	機構内指定軸補間計算待ち
	void DecelA (int mch, int setup)	機構内指定軸払い出し完了待ち
	void InposA (int mch, int setup)	機構内指定軸インポジション待ち
	void OrgA (int mch, int setup)	機構内指定軸原点復帰完了待ち

■ サーボ命令

分類	関数名	説明
サーボ命令	void ServoOn (int mch, int setup)	サーボオン
	void ServoOff (int mch, int setup)	サーボオフ
	void ServoFree (int mch, int setup)	サーボオフ＋ブレーキ開放
	void ServoMode (int mch, int axis_no, int mode)	制御モード設定
	void ServoVelocity (int mch, int axis_no, int vel)	速度設定
	void ServoCurrent (int mch, int axis_no, int cur)	電流設定
	void ServoParameter (int mch, int axis_no, int id_no, int data)	パラメータ設定

■ 原点復帰命令

分類	関数名	説明
サーボ命令	void HomeStart (int mch, int setup, int hs, int ms, int ls, int hm_io, int hm_ls, int lm_io, int lm_ls)	Z 相検出原点復帰
	void HomeZero (int mch, int setup)	その場原点
	void HomePosition (int mch, int setup, int setpos)	現在位置変更
	void HomeClear (int mch, int setup)	原点復帰済みステータスクリア
	void HomeServo (int mch, int setup, int typ, int pre, int dir, int vel, int crp, int hm_io, int hm_ls)	SV-NET 原点復帰
	void HomeBump (int mch, int setup, int pre, int dir, int vel, int tim, int trq)	SV-NET 突き当て式原点復帰

■ ネットワーク命令

分類	関数名	説明
ネットワーク命令 SVCC Only	void RS_Start (void)	RS232C 自動送受信を開始
	void RS_Stop (void)	RS232C 自動送受信を停止
	void RS_Set (int &var_adr, int rs_adr, int dev1, int dev2, int num)	RS232C データ設定
	void RS_Get (int &var_adr, int rs_adr, int dev1, int dev2, int num)	RS232C データ取得
	void RS_Wait (void)	RS232C 送受信完了待ち
ネットワーク命令	int ComStart (int com, int r/w, int num, int &var_adr, int interval)	COM ポート自動送受信を開始
	int ComStop (int com)	COM ポート自動送受信を停止
	int ComWrite (int com, int num, int &var_adr, int timeout)	COM ポートデータ設定
	int ComRead (int com, int num, int &var_adr, int timeout)	COM ポートデータ取得
	int ModbusTcpRequest (int port, int station, int code, int adr, int num, int &var_adr, int timeout)	ModbusTCP マスタからスレーブへリクエスト
	int ModbusRtuRequest (int com, int station, int code, int adr, int num, int &var_adr, int timeout)	ModbusRTU マスタからスレーブへリクエスト

■ 動作設定命令

分類	関数名	説明
動作設定命令	void JogJ_Set (int mch, int setup, int vel)	一定速動作
	void Mov＊J_Set (int mch, int setup, int pos, int vel)	位置決め
	void Mov＊JT_Set (int mch, int setup, int pos, int tim)	時間指定位置決め
	void Mov＊JBL_Set (int mch, int setup, int pos, int vel1, int vel2, int acc1, int acc2)	ベル型加減速
	void Mov＊JARC_Set (int mch, int setup, int pos, int vel, int mode, int opt)	中心・角度指定円弧補間

※は i または a、i なら相対位置動作設定命令、a なら絶対位置動作設定命令。

■ 動作制御命令

分類	関数名	説明
動作制御命令	void MoveStart (int mch, int setup)	動作開始
	void MoveStopAll (int mch int lvl, int aft)	機構内全軸停止
	void MoveStopAxis (int mch, int setup, int levl, int aft)	機構内軸指定停止
	void OVR_Set (int mch, int typ, int ovr_no, int data1, int data2)	オーバーライド設定
	void OVR_Get (int mch, int typ, int ovr_no, int &var_adr1, int &varadr2)	オーバーライド取得

■ テーブル命令

分類	関数名	説明
テーブル命令	void TableTeach (int mch, int setup, int tbl_no)	ティーチング
	void TableSave (void)	テーブルデータ保存
	void TableMovaP (int mch, int setup, int tbl_no)	絶対位置テーブル位置決め
	void TableMovaT (int mch, int setup, int tbl_no)	時間指定絶対位置テーブル位置決め
	void TableMoviP (int mch, int setup, int tbl_no)	相対位置テーブル位置決め
	void TableMoviT (int mch, int setup, int tbl_no)	時間指定相対位置テーブル位置決め

システム命令　詳細

■ **void Nop (void)**

【 説明 】 Nop 関数は、何も効果を及ぼしません。
主にコマンドメモリのクリアや時間待ちに使用します。

【 引数 】 無し

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        Nop ( );    //何の効果もありません。
        ・・・
}
```

■ **void AlarmReset (int mch)**

【 説明 】 mch で指定された機構（グループ）の異常を解除します。
ドライバがアラーム状態であれば、機構（グループ）に属するドライバの異常も解除します。

【 引数 】 mch 機構（グループ）番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( x & 0x01 )
        {
                AlarmReset ( 0 );     //x のビット 0 番が “ 1 ” ならば、
                                      //機構 0 番のアラームリセットを実行します。
        }
        ・・・
}
```

【 備考 】 （A）アラームリセット実行後、サーボオンする場合には 100msec 以上待つ必要があります。

■ **void SmoothingSet (int mch, int setup, int t1, int t2)**

【 説明 】 mch で指定された機構（グループ）の各軸加減速時定数を設定します。
SmoothinSet 関数で指定した時間は、加速、減速時間共通です。
いずれも t1＋t2 の値が設定されます。
setup で複数軸の指定を行うと、設定された軸全てに同じ値が設定されます。
軸毎に異なる設定にしたい場合は、SmoothingSet 関数を複数回実行する必要があります。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
t1	：	加減速時定数 1（単位：msec）
t2	：	加減速時定数 2（単位：msec）

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        SmoothigSet ( 0, 0x03, 100, 100 );      //機構 0 番の 1 軸目と 3 軸目の加減速時定数を
                                                //200msec（100＋100）に設定します。
        ・・・
}
```

【 備考 】 （A）軸動作中に加減速時定数を変更するは出来ません。
軸動作中に SmoothingSet 関数を実行するとアラームが発生します。
（B）引数 t1、t2 に設定できる最大値は 2000 です。
（C）加減速時定数 t1 および t2 の値は補間周期単位で割り切れる値を設定します。
端数の設定は無効となります。

■ **void ParameterSet (int cls_no, int grp_no, int id_no, int data)**

【 説明 】 **指定されたパラメータの値を設定します。**
クラス番号、グループ番号、ID 番号はパラメーター一覧を参照します。

【 引数 】
cls_no ： **クラス番号**
grp_no ： **グループ番号**
id_no ： ID **番号**
data ： **設定値**

【 戻値 】 **無し**

【 使用例 】
```
#pragma main

void main ( void )
{
        ・・・
        ParameterSet ( 0x2001, 1, 1, 1000 );    //機構 0 番 1 軸目のパルスレート分子を
                                                //1000 に設定します。
        ・・・
}
```

【 備考 】 **（A）**ParameterSet **関数でコントローラの全てのパラメータにアクセス可能ですが、**
不正なデータを設定してもアラームは発生しません。
ご使用になる場合にはパラメータ番号を良くご確認のうえお使いください。
（B）パラメータの変更は、軸動作中・サーボオン時に実行しないようにしてください。

■ **void ParameterGet (int cls_no, int grp_no, int id_no, int data_num, int &var_adr)**

【 説明 】 **指定されたパラメータの値を変数に格納します。**

var_adr **で指定された変数のアドレスに、**data_num **で指定された個数分のデータを格納します。**

クラス番号、グループ番号、ID **番号はパラメーター一覧を参照します。**

【 引数 】

cls_no	：	**クラス番号**
grp_no	：	**グループ番号**
id_no	：	ID **番号**
data_num	：	**パラメータ取得個数**
&var_adr	：	**パラメータを取得する変数のアドレス**

【 使用例 】

```
#pragma main
int prm_arr[10]

void main ( void )
{
        ・・・
        ParameterGet ( 0x2001, 1, 0, 10, prm_arr );
                                        //機構 0 番 1 軸目の ID0 番から 10 個分の
                                        //パラメータを配列 prm_arr に格納します。
        ・・・
}
```

【 備考 】 **（**A**）変数の指定と取得個数の指定を誤ると、他で参照している変数の値も書き換えてしまう為、注意が必要です。パラメータを格納する変数は配列型で定義します。**

■ **void MonitorGet (int cls_no, int grp_no, int id_no, int data_num, int &var_adr)**

【 説明 】 指定されたモニター項目の値を変数に格納します。
var_adr で指定された変数のアドレスに、data_num で指定された個数分のデータを格納します。
クラス番号、グループ番号、ID 番号はモニター項目一覧を参照します。

【 引数 】

cls_no	：	クラス番号
grp_no	：	グループ番号
id_no	：	ID 番号
data_num	：	モニター項目取得個数
&var_adr	：	モニター項目を取得する変数のアドレス

【 使用例 】

```
#pragma main
int mon_arr[10]

void main ( void )
{
        ・・・
        MonitorGet ( 0x1000, 3, 0, 10, mon_arr );
                                        //SVD 関連の各軸現在ステータス 10 個分の
                                        //モニター値を配列 mon_arr に格納します。
        ・・・
}
```

【 備考 】 （A）変数の指定と取得個数の指定を誤ると、他で参照している変数の値も書き換えてしまう為、
注意が必要です。モニター項目を格納する変数は配列型で定義します。

■ **void End (void)**

【 説明 】 End 関数は、プログラムの最後を示します。
End 関数を実行したタスク（番号）が停止します。

【 引数 】 無し

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main(void)
{
        ・・・
        End();      //メインタスクが停止します。
}

#pragma task3
void task3_main ( void )
{
        ・・・
        End ( );    //タスク 3 番が停止します。
}
```

データ命令　詳細

■ **void copy (int &var_adr, int &var_adr2, int size)**

【 説明 】 var_adr2 で指定された変数アドレスから、var_adr1 で指定された変数アドレスへデータ転送します。
転送するデータの個数は size で指定します。

【 引数 】

&var_adr1	：	データ転送先変数アドレス
&var_adr2	：	データ転送元変数アドレス
size	：	データ転送個数

【 戻値 】 無し

【 使用例 】

```
#pragma main
int dat1[200];
int dat2[50];

void main ( void )
{
        ・・・
        copy ( &dat1[100], &dat2[0], 50 );      //dat2[0]～dat2[49]の値を 50 個、
                                                //dat1[100]～dat1[149]へ転送します。
        ・・・
}
```

【 備考 】 （A）size の指定は最大 2000 個までです。

■ **int abs (int op1)**

【 説明 】 絶対値演算を行い、結果を変数に格納します。

【 引数 】 op1 ： 絶対値演算する元のデータ

【 戻値 】 絶対値演算の結果

【 使用例 】

```
#pragma main
int dest;
int op;

void main ( void )
{
        op = -55;
        dest = abs ( op );          //変数 dest には 55 が格納
        ・・・
}
```

■ **int swap (int op1)**

【 説明 】 swap 演算（上位 2 バイト、下位 2 バイト交換）を行い、結果を変数に格納します。

【 引数 】 op1 ： swap 演算する元のデータ

【 戻値 】 swap 演算の結果

【 使用例 】

```
#pragma main
int dest;
int op;

void main ( void )
{
        op = 0xAAAA5555;
        dest = swap ( op );          //変数 dest には 0x5555AAAA が格納
        ・・・
}
```

■ **int swap2 (int op1, int op2)**

【 説明 】 op1 の下位 2 バイトと op2 の下位 2 バイト合成を行い、結果を変数に格納します。

【 引数 】 op1 ： swap2 演算する元のデータ（下位 2 バイトを結果の上位 2 バイトへ）
op2 ： swap2 演算する元のデータ（下位 2 バイトを結果の下位 2 バイトへ）

【 戻値 】 swap2 演算の結果

【 使用例 】

```
#pragma main
int dest;
int op1;
int op2;

void main ( void )
{
        op1 = 0x11112222;
        op2 = 0x33334444;
        dest = swap ( op1, op2 );          //変数 dest には 0x22224444 が格納
        ・・・
}
```

浮動小数点演算命令　詳細

■ **double acos (double op1)**

【 説明 】 浮動小数点値の逆余弦を計算し、結果を変数に格納します。

【 引数 】 op1 ： 逆余弦を求める浮動小数点値

【 戻値 】 正常 ： op1 の逆余弦値
異常 ： 定義域エラーの時は非数を返します

【 使用例 】

```
#pragma main
double dest;
double op;

void main ( void )
{
        op = -1.0;
        dest = acos ( op );          //変数 dest には 3.141592…が格納
        ・・・
}
```

【 備考 】 （A）op1 の値が[-1.0, 1.0]の範囲を超えている時、定義域エラーになります。
（B）戻値の範囲は[0, π]になります。

■ **double asin (double op1)**

【 説明 】 浮動小数点値の逆正弦を計算し、結果を変数に格納します。

【 引数 】 op1 ： 逆正弦を求める浮動小数点値

【 戻値 】 正常 ： op1 の逆正弦値
異常 ： 定義域エラーの時は非数を返します

【 使用例 】
```
#pragma main
double dest;
double op;

void main ( void )
{
        op = -1.0;
        dest = asin ( op );          //変数 dest には-1.57079632…が格納
        ・・・
}
```

【 備考 】 （A）op1 の値が[-1.0, 1.0]の範囲を超えている時、定義域エラーになります。
（B）戻値の範囲は[-π/2，π/2]になります。

■ **double atan (double op1)**

【 説明 】 浮動小数点値の逆正接を計算し、結果を変数に格納します。

【 引数 】 op1 ： 逆正接を求める浮動小数点値

【 戻値 】 正常 ： op1 の逆正弦値
異常 ： 定義域エラーの時は非数を返します

【 使用例 】

```
#pragma main
double dest;
double op;

void main ( void )
{
        op = -1.0;
        dest = atan ( op );          //変数 dest には-0.78539816…が格納
        ・・・
}
```

【 備考 】 （A）戻値の範囲は[-π/2，π/2]になります。

■ **double atan2 (double op1, double op2)**

【 説明 】 除算した結果の逆正接を計算し、結果を変数に格納します。

【 引数 】 op1 ： 除数
op2 ： 被除数

【 戻値 】 正常 ： op1 を op2 で除算した結果の逆正弦値
異常 ： 定義域エラーの時は非数を返します

【 使用例 】

```
#pragma main
double dest;
double op1, op2;

void main ( void )
{
        op1 = -1.0;
        op2 = -0.5;
        dest = atan2 ( op1, op2 )          //変数 dest には-2.0344439…が格納
        ・・・
}
```

【 備考 】 （A）op1、op2 の値がともに 0.0 の時、定義域エラーになります。
（B）戻値の範囲は[-π，π]になります。

■ **double cos (double op1)**

【 説明 】 浮動小数点のラジアン値の余弦を計算し、結果を変数に格納します。

【 引数 】 op1 ： 余弦を求めるラジアン値

【 戻値 】 正常 ： op1 の余弦値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -1.0;
        dest = cos ( op1 )          //変数 dest には 0.54030230…が格納
        ・・・
}
```

■ **double sin (double op1)**

【 説明 】 浮動小数点のラジアン値の正弦を計算し、結果を変数に格納します。

【 引数 】 op1 ： 正弦を求めるラジアン値

【 戻値 】 正常 ： op1 の正弦値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -1.0;
        dest = sin ( op1 )          //変数 dest には-0.84147098…が格納
        ・・・
}
```

■ **double tan (double op1)**

【 説明 】 浮動小数点のラジアン値の正接を計算し、結果を変数に格納します。

【 引数 】 op1 ： 正接を求めるラジアン値

【 戻値 】 正常 ： op1 の正接値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -1.0;
        dest = tan ( op1 )          //変数 dest には-1.55740772…が格納
        ・・・
}
```

■ **double cosh (double op1)**

【 説明 】 浮動小数点値の双曲線余弦を計算し、結果を変数に格納します。

【 引数 】 op1 ： 双曲線余弦を求める浮動小数点値

【 戻値 】 正常 ： op1 の双曲線余弦値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -1.0;
        dest = cosh ( op1 )          //変数 dest には 1.54308063…が格納
        ・・・
}
```

■ **double sinh (double op1)**

【 説明 】 浮動小数点値の双曲線正弦を計算し、結果を変数に格納します。

【 引数 】 op1 ： 双曲線正弦を求める浮動小数点値

【 戻値 】 正常 ： op1 の双曲線正弦値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -1.0;
        dest = sinh ( op1 )          //変数 dest には-1.17520119…が格納
        ・・・
}
```

■ double tanh (double op1)

【 説明 】 浮動小数点値の双曲線正接を計算し、結果を変数に格納します。

【 引数 】 op1 ： 双曲線正接を求める浮動小数点値

【 戻値 】 正常 ： op1 の双曲線正接値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -1.0;
        dest = tanh ( op1 )          //変数 dest には-0.76159415…が格納
        ・・・
}
```

■ double exp(double op1)

【 説明 】　浮動小数点値の指数関数を計算し、結果を変数に格納します。

【 引数 】　op1　：　指数関数を求める浮動小数点値

【 戻値 】　正常　：　op1 の指数関数値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -1.0;
        dest = exp ( op1 )          //変数 dest には 0.36787944…が格納
        ・・・
}
```

■ **double frexp (double op1, int &op2)**

【 説明 】 浮動小数点値を[0.5, 1.0)の値と 2 のべき乗の積に分解します。

【 引数 】 op1 ： [0.5, 1.0)の値と 2 のべき乗の積に分解する浮動小数点値

&op2 ： 2 のべき乗値を格納する変数のアドレス

【 戻値 】 op1 が 0.0 の時 ： 0.0

op1 が 0.0 以外 ： dest×2^{exp} の指している領域の値 = value で定義される ret の値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -1.0;
        dest = frexp ( op1, &op2 )          //変数 dest には-0.76159415…が格納
        ・・・
}
```

【 備考 】 （A）戻値の範囲は[0.5, 1.0)または 0.0 になります。

■ **double ldexp (double op1, int op2)**

【 説明 】 浮動小数点値と 2 のべき乗の積を計算します。

【 引数 】 op1 ： 2 のべき乗を求める浮動小数点値
op2 ： 2 のべき乗値

【 戻値 】 op1×2^{op2} の演算結果の値

【 使用例 】

```
#pragma main
double dest;
double op1;
int op2;

void main ( void )
{
        op1 = -2.0;
        op2 = 2;
        dest = ldexp ( op1, op2 )          //変数 dest には-8.0…が格納
        ・・・
}
```

■ double log (double op1)

【 説明 】　浮動小数点値の自然対数を計算します。

【 引数 】　op1　：　自然対数を求める浮動小数点値

【 戻値 】　正常　：　op1 の自然対数の値
　　異常　：　定義域エラーの時は非数を返します

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = 2.0;
        dest = log ( op1 )          //変数 dest には 0.69314718…が格納
        ・・・
}
```

【 備考 】　（A）d の値が負の時、定義域エラーになります。
　　（B）d の値が 0.0 の時、範囲エラーになります。

■ **double log10 (double op1)**

【 説明 】 浮動小数点値の 10 を底とする対数を計算します。

【 引数 】 op1 ： 10 を底とする対数を求める浮動小数点値

【 戻値 】 正常 ： op1 は 10 を底とする対数値
異常 ： 定義域エラーの時は非数を返します

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = 2.0;
        dest = log10 ( op1 )          //変数 dest には 0.30102999…が格納
        ・・・
}
```

【 備考 】 （A）d の値が負の時、定義域エラーになります。
（B）d の値が 0.0 の時、範囲エラーになります。

■ **double modf (double op1, double &op2)**

【 説明 】 浮動小数点値を整数部分と小数部分に分解します。

【 引数 】 op1 ： 整数部分と小数部分を分解する浮動小数点値
&op2 ： 整数部分を格納する変数のアドレス

【 戻値 】 正常 ： op1 の小数部分

【 使用例 】

```
#pragma main
double dest;
double op1, op2;

void main ( void )
{
        op1 = 3.14;
        dest = modf ( op1, &op2 )          //変数 dest には 0.14 が格納
                                           //変数 op2 には 3.0 が格納
        ・・・
}
```

■ **double pow (double op1, double op2)**

【 説明 】 浮動小数点値のべき乗を計算します。

【 引数 】 op1 ： べき乗される値
op2 ： べき乗する値

【 戻値 】 正常 ： op1 の op2 乗の値
異常 ： 定義域エラーの時は非数を返します

【 使用例 】

```
#pragma main
double dest;
double op1, op2;

void main ( void )
{
        op1 = 2.0;
        op2 = 8.0;
        dest = pow ( op1, op2 )          //変数 dest には 256.0 が格納
        ・・・
}
```

■ **double sqrt (double op1)**

【 説明 】 浮動小数点値の正の平方根を計算します。

【 引数 】 op1 ： 正の平方根を求める浮動小数点値

【 戻値 】 正常 ： op1 の正の平方根の値
異常 ： 定義域エラーの時は非数を返します

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = 2.0;
        dest = sqrt ( op1 )          //変数 dest には 1.41421356…が格納
        ・・・
}
```

【 備考 】 （A）d の値が負の時、定義域エラーになります。

■ **double ceil (double op1)**

【 説明 】 浮動小数点値の小数点以下を切り上げた整数値を求めます。

【 引数 】 op1 ： 小数点以下を切り上げる浮動小数点値

【 戻値 】 op1 の小数点以下を切り上げた整数値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = 3.14;
        dest = ceil ( op1 )          //変数 dest には 4.0 が格納
        op1 = -3.14;
        dest = ceil ( op1 )          //変数 dest には-3.0 が格納
        ・・・
}
```

【 備考 】 （A）d の値が負の時は小数点以下を切り捨てた値を返します。

■ double fabs (double op1)

【 説明 】 浮動小数点値の絶対値を計算します。

【 引数 】 op1 ： 絶対値を求める浮動小数点値

【 戻値 】 op1 の絶対値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = -3.14;
        dest = fabs ( op1 )          //変数 dest には 3.14 が格納
        ・・・
}
```

■ **double floor (double op1)**

【 説明 】 浮動小数点値の小数点以下を切り捨てた整数値を求めます。

【 引数 】 op1 ： 小数点以下を切り捨てる浮動小数点値

【 戻値 】 op1 の小数点以下を切り捨てた整数値

【 使用例 】

```
#pragma main
double dest;
double op1;

void main ( void )
{
        op1 = 3.14;
        dest = floor ( op1 )          //変数 dest には 3.0 が格納
        op1 = -3.14;
        dest = floor ( op1 )          //変数 dest には-4.0 が格納
        ・・・
}
```

【 備考 】 （A）d の値が負の時は小数点以下を切り上げた値を返します。

■ **double fmod (double op1, double op2)**

【 説明 】 浮動小数点値どうしを除算した結果の余りを計算します。

【 引数 】 op1 ： 被除数
op2 ： 除数

【 戻値 】 op2 の値が 0.0 の時 ： 非数を返します
op2 の値が 0.0 以外 ： op1 を op2 で除算した結果の余り

【 使用例 】

```
#pragma main
double dest;
double op1, op2;

void main ( void )
{
        op1 = -3.14;
        op2 = 1.5;
        dest = fmod ( op1, op2 )            //変数 dest には-0.14 が格納
        ・・・
}
```

【 備考 】 （A）戻値の符号は op1 を同じ符号になります。
（B）op1÷op2 の商を表現できない場合、結果の値は保証されません。

タスク命令　詳細

■ **void TaskStart (int func_name)**

【 説明 】 タスクを起動し、引数 func_name で指定された関数から実行します。
起動されるタスク番号は、func_name が定義された関数が配置されたタスク番号となります。
タスク番号の指定は#pragma task＊で指定します。

【 引数 】 func_name ： タスクを起動後に実行される関数名

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        TaskStart ( task3_main );        //タスク 3 番を起動し、task3_main 関数から実行
        ・・・
}

#pramga task3

void task3_main ( void )
{
        ・・・
}
```

【 備考 】 （A）既に起動しているタスクに TaskStart 関数を実行した場合は無効です。（何もおきません）
（B）TaskStart 関数の引数にグローバル関数を指定する事は出来ません。

■ int TaskId (void)

【 説明 】 TaskId 関数を実行したタスク番号を取得し、結果を変数に格納します。

【 引数 】 無し

【 戻値 】 タスク番号

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        global_func ( );        //グローバル関数呼び出し
        ・・・
}

#pramga global
int task_id;

void global_func ( int x, int y )
{
        ・・・
        task_id = TaskId ( );           //main タスクが実行した場合、task_id には 0 を格納
        ・・・
}
```

■ **int TaskStatus (int task_no)**

【 説明 】 指定されたタスク番号のステータス（状態）を取得し、結果を変数に格納します。

【 引数 】 task_no : タスク番号

【 戻値 】 タスクステータス

【 使用例 】

```
#pragma main
int t_status;

void main ( void )
{
        ・・・
        t_status = TaskStatus( 1 );        //タスク 1 番のステータスを t_status に格納
        ・・・
}
```

■ **void TaskReStart (int task_no)**

【 説明 】 task_no で指定されたタスクを再起動します。
再起動されたタスクは現在のステップから実行されます。

【 引数 】 task_no ： タスク番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        TaskReStart ( 3 );     //タスク 3 番を再起動
        ・・・
}

#pramga task3

void task3_main ( void )
{
        ・・・
}
```

【 備考 】 （A）既に起動しているタスクに TaskReStart 関数を実行した場合は無効です。（何もおきません）

■ void TaskStep (int task_no)

【 説明 】 task_no で指定されたタスクをステップ実行します。ステップ実行後にタスクは停止します。

【 引数 】 task_no ： タスク番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        TaskStep ( 3 );        //タスク 3 番をステップ実行
        ・・・
}
```

【 備考 】 （A）既に起動しているタスクに TaskStep 関数を実行した場合は、現在ステップ実行後停止します。

■ **void TaskWait (int msec)**

【 説明 】 本コマンドを実行したタスクを指定した時間停止（単純待ち）します。

【 引数 】 msec ： 待ち時間（単位：msec）

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        TaskWait ( 300 );      //メインタスクを 300msec 停止（待ち）します。
        ・・・
}
```

【 備考 】 （A）タイマ命令の Waitmsec 関数と異なり、タイマ番号を指定する必要はありません。

■ **void TaskEnd (int task_no)**

【 説明 】 task_no で指定されたタスクを停止します。

【 引数 】 task_no ： タスク番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        TaskEnd ( 3 );          //タスク 3 番を停止
        ・・・
}
```

タイマ命令　詳細

■ **void TimerSet (int timer, int msec)**

【 説明 】 タイマを初期化します。
timer で指定されたタイマ番号を init で指定された値で初期化します。

【 引数 】 timer ： 初期化するタイマ番号
msec ： 初期化する値（単位：msec）

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        TimerSet ( 3, 0 );              //タスク 3 番を 0 に初期化
        while ( TIM[3] <= 1000 )        //タイマ 3 番が 1000msec 以下ならループ処理
        {
                ・・・
        }
        ・・・
}
```

【 備考 】 （A）コントローラ内部のタイマは 32 ビットの 1msec カウンタです。
タイマ値が 2147483647 未満ならば、常にカウントアップを継続します。
（B）複数のタスクで同じタイマ番号を使用される場合は、初期化するタイミングにご注意ください。

■ **void Waitmsec (int timer, int msec)**

【 説明 】 **単純時間待ちを実行します。**
timer **で指定されたタイマを** 0 **で初期化し、**msec **で指定された時間が経過するまで**
現在ステップで待機します。

【 引数 】 timer ： **初期化するタイマ番号**
wait ： **待ち時間（単位：**msec）

【 戻値 】 **無し**

【 使用例 】
```
#pragma main

void main ( void )
{
        ・・・
        Waitmsec ( 3, 500 );        //タスク 3 番を 0 に初期化し、500msec 待ち
        ・・・
}
```

【 備考 】 **(**A**) 複数のタスクで** Waitmsec **関数を使用する場合、タイマ番号が重複しないようにご注意ください。**

Ｉ／Ｏ命令　詳細

■ **void BitOn (int dio, int bit)**

【 説明 】 デジタル I/O の出力を ON します。
dio に出力するデジタル I/O 番号を、bit に出力するビット番号を指定します。
bit の 1 に対応するビットのみ ON します。
bit で 0 に指定されたビットには何もしません。

【 引数 】 dio : 出力するデジタル I/O 番号
bit : 出力するビット番号

【 戻値 】 無し

【 使用例 】
```
#pragma main

void main ( void )
{
        ・・・
        BitOn ( 0, 0x8001 );        //デジタル I/O_0 番のビット 0 番ビット 15 番を ON
        ・・・
}
```

【 備考 】 （A）存在しないデジタル I/O 番号を指定した場合、エラーとなります。

■ **void BitOff (int dio, int bit)**

【 説明 】 デジタル I/O の出力を OFF します。
dio に出力するデジタル I/O 番号を、bit に出力するビット番号を指定します。
bit の 1 に対応するビットのみ OFF します。
bit で 0 に指定されたビットには何もしません。

【 引数 】 dio : 出力するデジタル I/O 番号
bit : 出力するビット番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        BitOff ( 0, 0x8001 );          //デジタル I/O_0 番のビット 0 番ビット 15 番を OFF
        ・・・
}
```

【 備考 】 （A）存在しないデジタル I/O 番号を指定した場合、エラーとなります。

■ **void BitIn (int dio, int mask)**

【 説明 】 デジタル I/O の入力を取得します。
dio に取得するデジタル I/O 番号を、mask に取得するビット番号を指定します。
mask の 1 に対応するビットのみデータを取得します。
mask で 0 に指定されたビットは 0 に設定されます。

【 引数 】 dio : 入力を取得するデジタル I/O 番号
mask : 入力を取得するビット番号

【 戻値 】 無し

【 使用例 】
```
#pragma main

void main ( void )
{
        ・・・
        BitIn ( 0, 0x8001 );        //デジタル I/O_0 番のビット 0 番ビット 15 番の入力を取得
        ・・・
}
```

【 備考 】 （A）存在しないデジタル I/O 番号を指定した場合、エラーとなります。

■ **void DioOut (int dio, int data)**

【 説明 】 デジタル I/O にデータを出力します。
dio に出力するデジタル I/O 番号を、data に出力するデータを指定します。
data の 1 に対応するビットは ON、0 に対応するビットは OFF します。

【 引数 】 dio ： 出力するデジタル I/O 番号
data ： 出力するデータ

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        DioOut ( 0, 0xFFFF )        //デジタル I/O_0 番の全ビットを ON
        ・・・
}
```

【 備考 】 （A）存在しないデジタル I/O 番号を指定した場合、エラーとなります。

■ int DioIn (int dio)

【 説明 】 デジタル I/O の入力データを取得します。
dio にデータを取得するデジタル I/O 番号をを指定します。
取得したデータの 1 に対応するビットは ON、0 に対応するビットは OFF の状態を示します。

【 引数 】 dio : データを取得するデジタル I/O 番号

【 戻値 】 デジタル I/O の入力データ

【 使用例 】

```
#pragma main
int dio_data;

void main ( void )
{
        ・・・
        dio_data = DioIn ( 0 )          //デジタル I/O_0 番のデータを取得
        ・・・
}
```

【 備考 】 （A）存在しないデジタル I/O 番号を指定した場合、エラーとなります。

■ **void AioOut (int dio, int data)**

【 説明 】 アナログ I/O にデータを出力します。
aio に出力するアナログ I/O チャンネル番号を、data に出力するデータを指定します。
出力するアナログデータの分解能はハードウェアに依存します。

【 引数 】 aio ： 出力するアナログ I/O チャンネル番号
data ： 出力するデータ

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        AioOut ( 1, 2048 )          //アナログ I/O チャンネル 1 番に 2048 を出力
        ・・・
}
```

【 備考 】 （A）存在しないアナログ I/O チャンネル番号を指定した場合、エラーとなります。

■ **int AioIn (int dio)**

【 説明 】 アナログ I/O の入力データを取得します。
aio にデータを取得するアナログ I/O チャンネル番号をを指定します。
取得するアナログデータの分解能はハードウェアに依存します。

【 引数 】 aio : データを取得するアナログ I/O チャンネル番号

【 戻値 】 アナログ I/O の入力データ

【 使用例 】
```
#pragma main
int aio_data

void main ( void )
{
        ・・・
        aio_data = AioIn ( 1 )          //アナログ I/O チャンネル 1 番のデータを取得
        ・・・
}
```

【 備考 】 （A）存在しないアナログ I/O チャンネル番号を指定した場合、エラーとなります。

ＰＡＳＳ命令　詳細

■ **void PassM (int mch)**

【 説明 】 mch で指定された機構（グループ）内全軸の補間計算が完了するまで、
現在ステップで待機します。

【 引数 】 mch ： 機構（グループ）番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        MovaJ_Set ( 0, 0x01, 1000, 1000 );      //グループ 0 番 1 軸目　目標位置・速度設定
        MovaJ_Set ( 0, 0x02, 1000, 1000 );      //グループ 0 番 2 軸目　目標位置・速度設定
        MovaJ_Set ( 0, 0x04, 1000, 1000 );      //グループ 0 番 3 軸目　目標位置・速度設定
        MoveStart ( 0, 0x07 );                  //グループ 0 番 1～3 軸目　動作開始
        PassM ( 0 );                            //グループ 0 番全軸　補間計算完了待ち
        ・・・
}
```

【 備考 】 （A）補間計算とはコントローラ内部で指令位置を計算中の状態です。
（B）複数のタスクで同一グループ内の軸を別に制御する場合は、PassA 関数を使用します。
別タスクで同一グループ内の軸が動作中の場合に待機状態となる為。

■ **void DecelM (int mch)**

【 説明 】 mch で指定された機構（グループ）内全軸の指令払い出しが完了するまで、
現在ステップで待機します。

【 引数 】 mch ： 機構（グループ）番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        MovaJ_Set ( 0, 0x01, 1000, 1000 );      //グループ 0 番 1 軸目　目標位置・速度設定
        MovaJ_Set ( 0, 0x02, 1000, 1000 );      //グループ 0 番 2 軸目　目標位置・速度設定
        MovaJ_Set ( 0, 0x04, 1000, 1000 );      //グループ 0 番 3 軸目　目標位置・速度設定
        MoveStart ( 0, 0x07 );                  //グループ 0 番 1～3 軸目　動作開始
        DecelM ( 0 );                           //グループ 0 番全軸　指令払い出し完了待ち
        ・・・
}
```

【 備考 】 （A）指令払い出しとはコントローラ内部で指令位置を更新中の状態です。
（B）複数のタスクで同一グループ内の軸を別に制御する場合は、DecelA 関数を使用します。
別タスクで同一グループ内の軸が動作中の場合に待機状態となる為。

■ **void InposM (int mch)**

【 説明 】 mch で指定された機構（グループ）内全軸の位置決め（インポジション）が完了するまで、
現在ステップで待機します。

【 引数 】 mch ： 機構（グループ）番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        MovaJ_Set ( 0, 0x01, 1000, 1000 );      //グループ 0 番 1 軸目　目標位置・速度設定
        MovaJ_Set ( 0, 0x02, 1000, 1000 );      //グループ 0 番 2 軸目　目標位置・速度設定
        MovaJ_Set ( 0, 0x04, 1000, 1000 );      //グループ 0 番 3 軸目　目標位置・速度設定
        MoveStart ( 0, 0x07 );                  //グループ 0 番 1～3 軸目　動作開始
        InposM ( 0 );                           //グループ 0 番　全軸位置決め完了待ち
        ・・・
}
```

【 備考 】 （A）位置決め完了とはサーボドライバがインポジションの状態です。
（B）複数のタスクで同一グループ内の軸を別に制御する場合は、InposA 関数を使用します。
別タスクで同一グループ内の軸が動作中の場合に待機状態となる為。

■ **void OrgM (int mch)**

【 説明 】 mch で指定された機構（グループ）内全軸の原点復帰が完了するまで、
現在ステップで待機します。

【 引数 】 mch ： 機構（グループ）番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        HomeStart ( 0, 0x01, ・・・ );      //グループ 0 番 1 軸目　原点復帰開始
        HomeStart ( 0, 0x02, ・・・ );      //グループ 0 番 2 軸目　原点復帰開始
        HomeStart ( 0, 0x04, ・・・ );      //グループ 0 番 3 軸目　原点復帰開始
        OrgM ( 0 );                         //グループ 0 番全軸　　原点復帰完了待ち
        ・・・
}
```

【 備考 】 （A）複数のタスクで同一グループ内の軸を別に制御する場合は、OrgA 関数を使用します。
別タスクで同一グループ内の軸が動作中の場合に待機状態となる為。

■ **void PassA (int mch, int setup)**

【 説明 】 mch で指定された機構（グループ）内の setup で指定された軸の補間計算が完了するまで、
現在ステップで待機します。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        MovaJ_Set ( 0, 0x04, 1000, 1000 );      //グループ 0 番 3 軸目　目標位置・速度設定
        MoveStart ( 0, 0x04 );                  //グループ 0 番 3 軸目　動作開始
        PassA ( 0, 0x04 );                      //グループ 0 番 3 軸目　補間計算完了待ち
        ・・・
}
```

【 備考 】 （A）補間計算とはコントローラ内部で指令位置を計算中の状態です。

■ **void DecelA (int mch, int setup)**

【 説明 】 mch で指定された機構（グループ）内の setup で指定された軸の指令払い出しが完了するまで、
現在ステップで待機します。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        MovaJ_Set ( 0, 0x04, 1000, 1000 );      //グループ 0 番 3 軸目　目標位置・速度設定
        MoveStart ( 0, 0x04 );                  //グループ 0 番 3 軸目　動作開始
        DecelA ( 0, 0x04 );                     //グループ 0 番 3 軸目　指令払い出し完了待ち
        ・・・
}
```

【 備考 】 （A）指令払い出しとはコントローラ内部で指令位置を更新中の状態です。
（B）複数のタスクで同一グループ内の軸を別に制御する場合は、DecelA 関数を使用します。
別タスクで同一グループ内の軸が動作中の場合に待機状態となる為。

■ **void InposA (int mch, int setup)**

【 説明 】 mch で指定された機構（グループ）内の setup で指定された軸の位置決め（インポジション）が完了するまで、現在ステップで待機します。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        MovaJ_Set ( 0, 0x04, 1000, 1000 );      //グループ 0 番 3 軸目　目標位置・速度設定
        MoveStart ( 0, 0x04 );                  //グループ 0 番 3 軸目　動作開始
        InposA ( 0, 0x04 );                     //グループ 0 番 3 軸目　位置決め完了待ち
        ・・・
}
```

【 備考 】 （A）位置決め完了とはサーボドライバがインポジションの状態です。

■ void OrgA (int mch, int setup)

【 説明 】 mch で指定された機構（グループ）内の setup で指定された軸の原点復帰が完了するまで、
現在ステップで待機します。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        HomeStart ( 0, 0x04, ・・・ );          //グループ 0 番 3 軸目　原点復帰開始
        OrgM ( 0, 0x04 );                       //グループ 0 番 3 軸目　原点復帰完了待ち
        ・・・
}
```

サーボ命令　詳細

■ **void ServoOn (int mch, int setup)**

【 説明 】 mch で指定された機構（グループ）内の setup で指定された軸をサーボオンします。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoOn ( 0, 0x0C );        //グループ 0 番 3、4 軸目　サーボオン
        ・・・
}
```

■ void ServoOff (int mch, int setup)

【 説明 】 mch で指定された機構（グループ）内の setup で指定された軸をサーボオフします。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoOff ( 0, 0x0C );          //グループ 0 番 3、4 軸目　サーボオフ
        ・・・
}
```

■ **void ServoFree (int mch, int setup)**

【 説明 】 mch で指定された機構（グループ）内の setup で指定された軸をサーボオフ＋ブレーキを解除します。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoFree ( 0, 0x0C );          //グループ 0 番 3、4 軸目　サーボオフ＋ブレーキ解除
        ・・・
}
```

【 備考 】 （A）ServoFree 関数は機構（グループ）が SV-NET の場合のみ使用可能です。

■ void ServoMode (int mch, int axis_no, int mode)

【 説明 】 mch で指定された機構（グループ）内の axis_no で指定された軸の制御モードを変更します。
制御モードの設定は下記の通りです。

0	：	制御モード無し
1	：	位置制御モード（既定値）
2	：	速度制御モード
3	：	電流制御モード

【 引数 】

mch	：	機構（グループ）番号
axis_no	：	軸番号
mode	：	制御モード

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoOn ( 0, 0x04 );          //グループ 0 番 3 軸目　サーボオン
        ServoMode ( 0, 3, 2 );        //グループ 0 番 3 軸目　速度制御モード
        ・・・
}
```

【 備考 】
（A）ServoMode 関数は機構（グループ）が SV-NET の場合のみ使用可能です。
（B）ServoOn 関数後、コントローラは強制的にドライバを位置制御モードにします。
速度制御モード･電流制御モードで使用する場合、サーボオン後にモード再設定します。
（C）軸動作中のモード変更は、ドライバの仕様により制限（可否）があります。
（B）速度制御モード・電流制御モードから位置制御に変更し軸を動作させる場合、
再度 ServoOn 関数を実行する必要があります。

■ **void ServoVelocity (int mch, int axis_no, int vel)**

【 説明 】 mch で指定された機構（グループ）内の axis_no で指定された軸の速度を設定します。

【 引数 】

mch	：	機構（グループ）番号
axis_no	：	軸番号
vel	：	設定速度（単位：rpm）

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoOn ( 0, 0x04 );          //グループ 0 番 3 軸目　サーボオン
        ServoMode ( 0, 3, 2 );        //グループ 0 番 3 軸目　速度制御モード
        ServoVelocity ( 0, 3, 100 );  //グループ 0 番 3 軸目　設定速度 100rpm
        ・・・
}
```

【 備考 】 （A）制御モードが速度制御モードに設定されていない場合、軸は動作しません。

（B）速度制御モード時の加減速時定数はサーボドライバ内部のパラメータを使用します。

■ **void ServoCurrent (int mch, int axis_no, int cur)**

【 説明 】 mch で指定された機構（グループ）内の axis_no で指定された軸の速度を設定します。

【 引数 】

mch	：	機構（グループ）番号
axis_no	：	軸番号
cur	：	設定電流（単位：0.01A）

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoOn ( 0, 0x04 );          //グループ 0 番 3 軸目　サーボオン
        ServoMode ( 0, 3, 3 );        //グループ 0 番 3 軸目　電流制御モード
        ServoCurrent ( 0, 3, 100 );   //グループ 0 番 3 軸目　設定電流 1.0A
        ・・・
}
```

【 備考 】 （A）制御モードが電流制御モードに設定されていない場合、軸は動作しません。

■ **void ServoParameter (int mch, int axis_no, int id_no, int data)**

【 説明 】 mch で指定された機構（グループ）内の axis_no で指定された軸のパラメータを設定します。
id_no でパラメータ番号、data で設定するパラメータ値を設定します。

【 引数 】

mch	：	機構（グループ）番号
axis_no	：	軸番号
id_no	：	パラメータ ID 番号
data	：	設定パラメータ値

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoParameter ( 0, 3, 50, 120);        //グループ 0 番 3 軸目
        ・・・                                   // パラメータ ID50 番を 120 に設定
}
```

【 備考 】 （A）パラメータ ID 番号はサーボドライバの取扱説明書を参照します。

原点復帰命令　詳細

■ **void HomeStart (int mch, int setup, int hs, int ms, int ls, int hm_io, int hm_ls, int lm_io, int lm_ls)**

【 説明 】 Z 相検出原点復帰を実行します。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
hs	：	原点復帰開始速度（単位：速度指令単位）
ms	：	原点近傍 LS 移動速度（単位：速度指令単位）
ls	：	Z サーチ速度（単位：速度指令単位）
hm_io	：	原点近傍 LS デジタル I/O 番号
hm_ls	：	原点近傍 LS デジタル I/O ビット番号
lm_io	：	限界 LS デジタル I/O 番号
lm_ls	：	限界 LS デジタル I/O ビット番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ServoOn ( 0, 0x04 );          //グループ 0 番 3 軸目　サーボオン
        HomeStart ( 0, 0x04,          //グループ 0 番 3 軸目
                1000,                 //原点復帰開始速度          1000（指令速度単位）
                500,                  //原点近傍 LS 移動速度      500（指令速度単位）
                200,                  //Z サーチ速度              200（指令速度単位）
                0,                    //原点近傍 LS デジタル I/O 番号          0 番
                0x0001                //原点近傍 LS デジタル I/O ビット番号    ビット 0 番
                0,                    //限界 LS デジタル I/O 番号              0 番
                0x8000 );             //限界 LS デジタル I/O ビット番号        ビット 15 番
        OrgA ( 0, 0x04 );             //グループ 0 番 3 軸目　原点復帰完了待ち
        ・・・
}
```

【 備考 】 （A）サーボオン処理が完了していない軸に対して HomeStart 関数を実行した場合、
アラームが発生します。

■ **void HomeZero (int mch, int setup)**

【 説明 】 その場原点処理を実行します。現在位置が 0 となります。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        HomeZero ( 0, 0x07 );          //グループ 0 番 1~3 軸目　現在位置を 0
        ・・・
}
```

【 備考 】 （A）HomeZero 関数は必ず軸が停止した状態で実行してください。

■ void HomePosition (int mch, int setup, int setpos)

【 説明 】　現在位置変更を実行します。現在位置が pos となります。

【 引数 】　mch　：　機構（グループ）番号
　　setup　：　設定軸番号
　　pos　：　原点プリセット位置

【 戻値 】　無し

【 使用例 】

```
#pragma main

void main ( void )
{
        HomePosition ( 0, 0x01, 100 );      //グループ 0 番 1 軸目　現在位置を 100
        HomePosition ( 0, 0x02, 200 );      //グループ 0 番 2 軸目　現在位置を 200
        ・・・
}
```

【 備考 】　（A）HomePosition 関数は必ず軸が停止した状態で実行してください。
　　（B）HomePosition 関数は 1 軸づつ設定します。複数の軸を設定したい場合には複数回実行します。
　　仮に setup に複数軸が設定された場合、一番若い軸のみ実行されます。

■ **void HomeClear (int mch, int setup)**

【 説明 】 原点復帰済みステータスをクリアします。
設定軸番号により指定した軸のステータスだけをクリアする事も出来ます。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】
```
#pragma main

void main ( void )
{
        HomeClear ( 0, 0x05 );          //グループ 0 番の 1 軸目と 3 軸目の
                                        //原点復帰済みステータスをクリアします。
        ・・・
}
```

【 備考 】 （A）HomeClear 関数は必ず軸が停止した状態で実行してください。

■ **void HomeServo (int mch, int setup, int typ, int pre, int dir, int vel, int crp, int hm_io, int hm_ls)**

【 説明 】 サーボドライバに原点復帰を指示します。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
typ	：	原点復帰タイプ
		0：原点近傍 LS が ON した位置から一回転内の Z 相位置を原点
		2：原点近傍 LS が ON した位置を原点
		3：原点近傍 LS が ON→OFF した位置を原点
pre	：	原点復帰後のプリセット位置（単位：pulse）
dir	：	原点復帰方向
vel	：	原点復帰速度（単位：rpm）
crp	：	creep 速度（単位：rpm）
hm_io	：	原点近傍 LS デジタル I/O 番号
hm_ls	：	原点近傍 LS デジタル I/O ビット番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ServoOn ( 0, 0x04 );          //グループ 0 番 3 軸目　サーボオン
        HomeStart ( 0, 0x04,          //グループ 0 番 3 軸目
                0,                    //原点復帰タイプ          0
                0,                    //プリセット位置          0pulse
                0,                    //原点復帰方向            0（正方向）
                100,                  //原点復帰速度            100rpm
                50                    //creep 速度              50rpm
                0,                    //原点近傍 LS デジタル I/O 番号      0 番
                0x8000 );             //原点近傍 LS デジタル I/O ビット番号  ビット 15 番
        OrgA ( 0, 0x04 );             //グループ 0 番 3 軸目　原点復帰完了待ち
        ・・・
}
```

【 備考 】 （A）サーボオン処理が完了していない軸に対して HomeServo 関数を実行した場合、
アラームが発生します。
（B）HomeServo 関数の加減速時定数はサーボドライバ内部のパラメータを使用します。

■ **void HomeBump (int mch, int setup, int pre, int dir, int vel, int msec, int trq)**

【 説明 】 サーボドライバに突き当て原点復帰を指示します。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
pre	：	原点復帰後のプリセット位置（単位：pulse）
dir	：	原点復帰方向
vel	：	原点復帰速度（単位：rpm）
msec	：	突き当て時間（単位：msec）
trq	：	突き当てトルク（単位：0.01A）

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ServoOn ( 0, 0x04 );          //グループ 0 番 3 軸目　サーボオン
        HomeStart ( 0, 0x04,          //グループ 0 番 3 軸目
                0,                    //プリセット位置          0pulse
                0,                    //原点復帰方向            0（正方向）
                100,                  //原点復帰速度            100rpm
                500,                  //突き当て時間            500msec
                100 );                //突き当てトルク          1.0A
        OrgA ( 0, 0x04 );             //グループ 0 番 3 軸目　原点復帰完了待ち
        ・・・
}
```

【 備考 】 （A）サーボオン処理が完了していない軸に対して HomeBump 関数を実行した場合、
アラームが発生します。
（B）HomeBump 関数の加減速時定数はサーボドライバ内部のパラメータを使用します。
（C）HomeBump 関数完了後、自動的にサーボオフします。続けて軸を動作させる場合には、
再度サーボオン処理を実行する必要があります。

ネットワーク命令　詳細

■ **void RS_Start (void)**

【 説明 】 コントローラの RS232C ポートに接続された機器との自動送受信を開始します。

【 引数 】 無し

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        RS_Start ( );                   //RS232C の自動送受信を開始します。
        ・・・
}
```

【 備考 】 (A) 自動送受信の接続機器、先頭アドレス、取得個数等はコントローラのパラメータ設定に従います。

(B) RS_Start 関数はコントローラ機種タイプが SVCC のみ使用可能です。

■ **void RS_Stop (void)**

【 説明 】 コントローラの RS232C ポートに接続された機器との自動送受信を停止します。

【 引数 】 無し

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        RS_Start ( );           //RS232C の自動送受信を開始します。
        ・・・
        RS_Stop ( );            //RS232C の自動送受信を停止します。
}
```

【 備考 】 （A）RS_Stop 関数はコントローラ機種タイプが SVCC のみ使用可能です。

■ **void RS_Set (int &var_adr, int &rs_adr, int dev1, int dev2 int num)**

【 説明 】 コントローラの RS232C ポートに接続された機器にデータを設定します。

【 引数 】

&var_adr	：	設定（書き込み）するデータの変数のアドレス
&rs_adr	：	接続機器の先頭アドレス
dev1	：	接続機器のデバイス種別 1 文字目
dev2	：	接続機器のデバイス種別 2 文字目
num	：	設定（書き込み）個数

【 戻値 】 無し

【 使用例 】

```
#pragma main
int rs_temp[64];

void main ( void )
{
        RS_Set ( &rs_temp[0], 0x010, 0, 0, 10 );
        ・・・                          //変数 rs_temp[0]から 10 個分のデータを
        ・・・                          //接続機器のアドレス 0x10 番地に設定（書き込み）します。
        ・・・
}
```

【 備考 】 （A）一度に転送可能なデータ個数は 64 個です。

（B）接続機器のデバイス種別は計算機リンクでのみ使用します。他プロトコルでは使用しません。

（C）RS_Set 関数はコントローラ機種タイプが SVCC のみ使用可能です。

■ **void RS_Get (int &var_adr, int &rs_adr, int dev1, int dev2 int num)**

【 説明 】 コントローラの RS232C ポートに接続された機器のデータを取得します。

【 引数 】

&var_adr	：	取得（読み込み）するデータの変数のアドレス
&rs_adr	：	接続機器の先頭アドレス
dev1	：	接続機器のデバイス種別 1 文字目
dev2	：	接続機器のデバイス種別 2 文字目
num	：	取得（読み込み）個数

【 戻値 】 無し

【 使用例 】

```
#pragma main
int rs_temp[64];

void main ( void )
{
        RS_Get ( &rs_temp[0], 0x010, 0, 0, 64 );
        ・・・                          //接続機器のアドレス 0x10 番地から 64 個分のデータを
        ・・・                          //変数 rs_temp[0]～rs_temp[63]に取得（読み込み）します。
        ・・・
}
```

【 備考 】 （A）一度に転送可能なデータ個数は 64 個です。

（B）接続機器のデバイス種別は計算機リンクでのみ使用します。他プロトコルでは使用しません。

（C）RS_Get 関数はコントローラ機種タイプが SVCC のみ使用可能です。

■ **void RS_Wait (void)**

【 説明 】 RS_Set ( )、RS_Get ( )コマンドが完了するまで現在インデクスで待機します。

【 引数 】 無し

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        RS_Set ( ・・・ );          //RS232C データ設定
        RS_Wait ( );               //データ設定完了するまで待機
        RS_Get ( ・・・ );          //RS232C データ取得
        RS_Wait ( );               //データ取得完了するまで待機
        ・・・
}
```

【 備考 】 （A）RS_Wait 関数はコントローラ機種タイプが SVCC のみ使用可能です。

■ **int ComStart (int com, int r/w, int num, int &var_adr, int interval)**

【 説明 】 コントローラの COM ポートに接続された機器からデータを参照・設定します。
本コマンドは無手順方式の自動送受信に使用します。

【 引数 】

com	：	COM ポート番号
r/w	：	送信または受信設定（0：受信設定　0 以外：送信設定）
num	：	送信または受信するデータ数
&var_adr	：	送信または受信するデータを格納する変数の先頭アドレス
interval	：	送信または受信間隔（単位：msec）

【 戻値 】

1	：	ComStart 関数が正常に実行された
-1	：	無効な COM 番号を指定
-2	：	COM 設定が無手順方式でない
-20	：	コマンド無効、既に自動送受信中

【 使用例 】

```
#pragma main
int com_temp[64];

void main ( void )
{
        ComStart ( 0, 0, 20, &com_temp[0], 100 );
        ・・・                          //COM ポート 0 番の機器から
        ・・・                          //変数 com_temp[0]に 20 個分のデータを
        ・・・                          //100msec 周期で取得します。
        ・・・
}
```

【 備考 】 （A）一度に送信・受信可能なデータ個数は 255 個です。
255 以上の設定をした場合、内部で自動的に 255 とします。
（B）送受信間隔の最小値は 2（単位：msec）です。
2 以下の設定をした場合、内部で自動的に 2 とします。
送受信間隔の設定は、ボーレートや機器と通信するデータサイズを確認し決定します。

■ int ComStop (int com)

【 説明 】 コントローラの COM ポートに接続された機器との自動送受信を停止します。
本コマンドは無手順方式の自動送受信モードを停止します。

【 引数 】 com : COM ポート番号

【 戻値 】
1	:	ComStop 関数が正常に実行された
-1	:	無効な COM 番号を指定
-2	:	COM 設定が無手順方式でない
-20	:	コマンド無効、既に自動送受信停止中

【 使用例 】
```
#pragma main
int com_temp[64];

void main ( void )
{
        ComStop( 0 );           //COM ポート 0 番の
        ・・・                  //自動送受信を停止します。
        ・・・
}
```

■ **int ComWrite (int com, int num, int &var_adr, int timeout)**

【 説明 】 コントローラの COM ポートに接続された機器にデータを設定します。
本コマンドは無手順方式で使用します。

【 引数 】

com	：	COM ポート番号
num	：	データ設定個数
&var_adr	：	設定するデータを格納する変数の先頭アドレス
interval	：	タイムアウト時間（単位：msec）

【 戻値 】

1	：	ComWrite 関数が正常に実行された
-1	：	無効な COM 番号を指定
-2	：	COM 設定が無手順方式でない
-10	：	送信オーバーラン
-20	：	コマンド無効、既に自動送受信中
-1xxx	：	タイムアウト、下位 3 桁に送信済みデータ個数を格納

【 使用例 】

```
#pragma main
int com_temp[64];

void main ( void )
{
        ComWrite ( 0, 20, &com_temp[0], 1000 );
        ・・・                          //COM ポート 0 番の機器へ
        ・・・                          //変数 com_temp[0]の 20 個分のデータを設定します。
        ・・・
}
```

【 備考 】 （A）一度に送信可能なデータ個数は 255 個です。
255 以上の設定をした場合、内部で自動的に 255 とします。
（B）タイムアウト時間の最小値は 10（単位：msec）です。
10 以下の設定をした場合、内部で自動的に 10 とします。

■ **int ComRead (int com, int num, int &var_adr, int timeout)**

【 説明 】 コントローラの COM ポートに接続された機器からデータを取得します。
本コマンドは無手順方式で使用します。

【 引数 】

com	：	COM ポート番号
num	：	データ取得個数
&var_adr	：	取得したデータを格納する変数の先頭アドレス
interval	：	タイムアウト時間（単位：msec）

【 戻値 】

1	：	ComRead 関数が正常に実行された
-1	：	無効な COM 番号を指定
-2	：	COM 設定が無手順方式でない
-10	：	送信オーバーラン
-20	：	コマンド無効、既に自動送受信中
-1xxx	：	タイムアウト、下位 3 桁に受信済みデータ個数を格納

【 使用例 】

```
#pragma main
int com_temp[64];

void main ( void )
{
        ComRead ( 0, 20, &com_temp[0], 1000 );
        ・・・                          //COM ポート 0 番の機器から
        ・・・                          //変数 com_temp[0]へ 20 個分のデータを取得します。
        ・・・
}
```

【 備考 】 （A）一度に受信可能なデータ個数は 255 個です。
255 以上の設定をした場合、内部で自動的に 255 とします。
（B）タイムアウト時間の最小値は 10（単位：msec）です。
10 以下の設定をした場合、内部で自動的に 10 とします。

■ int ModbusRtuRequest (int com, int station, int code, int num, int &var_adr, int timeout)

【 説明 】 コントローラの COM ポートに接続された ModbusRTU スレーブ機器のデータを参照・設定します。
本コマンドはコントローラが ModbusRTU マスタモードの時に有効です。

【 引数 】

com	：	COM ポート番号
station	：	リクエストを発行するスレーブの局番号
code	：	Modbus ファンクションコード
adr	：	リクエストを要求するスレーブのデータアドレス
num	：	データ個数
&var_adr	：	設定または取得する変数先頭アドレス
interval	：	タイムアウト時間（単位：msec）

【 戻値 】

1	：	ModbusRtuRequest コマンドが正常に実行された
-1	：	無効なファンクションコードを指定
-2	：	無効な局番号を指定
-3	：	登録されていない局番号を指定
-4	：	Force Single Coil コマンドで不正なデータを指定
-5	：	不正な COM ポート番号を指定
-20	：	応答の局番号が不正
-21	：	応答のファンクションコードが不正
-22	：	Read コマンドの受信バイト数が不正
-23	：	Write コマンドの受信バイト数が不正
-200	：	指定した局のスレーブと接続が確立していない
-500	：	Modbus マスタの設定でない
-1xxx	：	タイムアウト、下位 3 桁に受信済みデータ個数を格納

【 ファンクションコード補足 】

01H	：	Read Coil Status
02H	：	Read Input Status
03H	：	Read Holding Register
04H	：	Read Input Register
05H	：	Force Single Register
06H	：	Preset Single Register
0FH	：	Force Multiple Coils
10H	：	Preset Multiple Registers

【 使用例 】

```
#pragma main
Int mon[50];
int ret;

void main ( void )
{
        ret = ModbusRtuRequest( 0, 2, 0x02, 0xC4, 14, &mon[0], 1000 )
                                        //COM ポート 0 番
                                        //局番号 2 のスレーブ
                                        //ファンクションコード（0x02）  ReadInputStatus
                                        //スレーブ内部アドレス  0xC4 番地から 14 個のデータ
                                        //データ取得先変数  mon[0]の先頭アドレス
                                        //タイムアウト時間  1000msec

        if( ret != 1 )
        {
                //エラー処理
        }
        ・・・
}
```

【 備考 】

（A）一度に参照・設定可能なデータ個数は 255 個です。（接点データも 255 個まで）
255 以上の設定をした場合、内部で自動的に 255 とします。

（B）タイムアウト時間の最小値は 10（単位：msec）です。
10 以下の設定をした場合、内部で自動的に 10 とします。

■ void ModbusRtuVariableAllocate (int com, int &coil_bit, int &input_bit, int &input_reg, int &holding_reg)

【 説明 】 ModbusRTU 通信で使用するデータの変数を割り当てます。
本コマンドはコントローラが ModbusRTU スレーブモードの時に有効です。

【 引数 】

com	：	COM ポート番号
&coil_bit	：	コイルビットに割り当てる変数の先頭アドレス
&input_bit	：	入力ビットに割り当てる変数の先頭アドレス
&input_reg	：	入力レジスタに割り当てる変数の先頭アドレス
&holding_reg	：	保持レジスタに割り当てる変数の先頭アドレス

【 戻値 】 無し

【 備考 】 （A）SVCE、SVCX シリーズでは、本コマンドは COM ポート 0 番のみ有効です。
無効な COM ポート番号を指定した場合には何もおきません。

■ **void ModbusRtuBitMode (int com, int bit_mode)**

【 説明 】 ModbusRTU のビットデバイス割り当てを設定します。
引数 bit_mode が 0 の時はビットパターンで、
引数 bit_mode が 1 の時は bool 型（ワードデータを真・偽）として割り当てられます。
ビットパターンの場合は 1 レジスタ辺り 16 ビット、
レジスタ割り当ての場合は 1 レジスタ辺り 1 ビットのデータを表現します。
本設定はマスタ及びスレーブともに共通の設定です。

【 引数 】

com	：	COM ポート番号
bit_mode	：	ビットの割り当てを設定します。（0：ビットパターン　1：レジスタ）

【 戻値 】 無し

【 備考 】 （A）SVCE、SVCX シリーズでは、本コマンドは COM ポート 0 番のみ有効です。
無効な COM ポート番号を指定した場合には何もおきません。

■ int ModbusTcpRequest (int com, int station, int code, int num, int &var_adr, int timeout)

【 説明 】 コントローラの Ethernet ポートに接続された ModbusTCP スレーブ機器のデータを参照・設定します。
本コマンドはコントローラが ModbusTCP マスタモードの時に有効です。

【 引数 】

port	：	Ethernet ポート番号
station	：	リクエストを発行するスレーブの局番号
code	：	Modbus ファンクションコード
adr	：	リクエストを要求するスレーブのデータアドレス
num	：	データ個数
&var_adr	：	設定または取得する変数先頭アドレス
interval	：	タイムアウト時間（単位：msec）

【 戻値 】

1	：	ModbusTcpRequest コマンドが正常に実行された
-1	：	無効なファンクションコードを指定
-2	：	無効な局番号を指定
-3	：	登録されていない局番号を指定
-4	：	Force Single Coil コマンドで不正なデータを指定
-5	：	不正な COM ポート番号を指定
-20	：	応答の局番号が不正
-21	：	応答のファンクションコードが不正
-22	：	Read コマンドの受信バイト数が不正
-23	：	Write コマンドの受信バイト数が不正
-200	：	指定した局のスレーブと接続が確立していない
-500	：	Modbus マスタの設定でない
-1xxx	：	タイムアウト、下位 3 桁に受信済みデータ個数を格納

【 ファンクションコード補足 】

01H	：	Read Coil Status
02H	：	Read Input Status
03H	：	Read Holding Register
04H	：	Read Input Register
05H	：	Force Single Register
06H	：	Preset Single Register
0FH	：	Force Multiple Coils
10H	：	Preset Multiple Registers

【 使用例 】

```
#pragma main
Int mon[50];
int ret;

void main ( void )
{
        ret = ModbusTcpRequest( 0, 2, 0x02, 0xC4, 14, &mon[0], 1000 )
                                        //Ethernet ポート 0 番
                                        //局番号 2 のスレーブ
                                        //ファンクションコード（0x02）  ReadInputStatus
                                        //スレーブ内部アドレス  0xC4 番地から 14 個のデータ
                                        //データ取得先変数  mon[0]の先頭アドレス
                                        //タイムアウト時間  1000msec

        if( ret != 1 )
        {
                //エラー処理
        }
        ・・・
}
```

【 備考 】

（A）一度に参照・設定可能なデータ個数は 255 個です。（接点データも 255 個まで）
255 以上の設定をした場合、内部で自動的に 255 とします。

（B）タイムアウト時間の最小値は 10（単位：msec）です。
10 以下の設定をした場合、内部で自動的に 10 とします。

■ void ModbusTcpVariableAllocate (int port, int &coil_bit, int &input_bit, int &input_reg, int &holding_reg)

【 説明 】 ModbusTCP 通信で使用するデータの変数を割り当てます。
本コマンドはコントローラが ModbusTCP スレーブモードの時に有効です。

【 引数 】

port	：	Ethernet ポート番号
&coil_bit	：	コイルビットに割り当てる変数の先頭アドレス
&input_bit	：	入力ビットに割り当てる変数の先頭アドレス
&input_reg	：	入力レジスタに割り当てる変数の先頭アドレス
&holding_reg	：	保持レジスタに割り当てる変数の先頭アドレス

【 戻値 】 無し

【 備考 】 （A）SVCE、SVCX シリーズでは、本コマンドは Ethernet ポート 0 番のみ有効です。
無効な Ethernet ポート番号を指定した場合には何もおきません。

■ **void ModbusTcpBitMode (int com, int bit_mode)**

【 説明 】 ModbusTCP のビットデバイス割り当てを設定します。
引数 bit_mode が 0 の時はビットパターンで、
引数 bit_mode が 1 の時は bool 型（ワードデータを真・偽）として割り当てられます。
ビットパターンの場合は 1 レジスタ辺り 16 ビット、
レジスタ割り当ての場合は 1 レジスタ辺り 1 ビットのデータを表現します。
本設定はマスタ及びスレーブともに共通の設定です。

【 引数 】

port	：	Ethernet ポート番号
bit_mode	：	ビットの割り当てを設定します。（0：ビットパターン　1：レジスタ）

【 戻値 】 無し

【 備考 】 （A）SVCE、SVCX シリーズでは、本コマンドは Ethernet ポート 0 番のみ有効です。
無効な Ethernet ポート番号を指定した場合には何もおきません。

動作設定命令　詳細

■ **void JogJ_Set (int mch, int setup, int vel)**

【 説明 】 一定速動作命令の目標設定を行います。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
vel	：	目標速度（単位：速度指令単位）

【 戻値 】 無し

【 使用例 】

```
#pragma main
int mch = 0;
int setup = 0x05;
int vel = 1000;

void main ( void )
{
        ・・・
        ServoOn ( mch, setup );        //グループ 0 番 1、3 軸目　サーボオン
        JogJ_Set ( mch, 0x01, vel );   //1 軸目の一定速動作設定
        JogJ_Set ( mch, 0x04, vel );   //3 軸目の一定速動作設定
        MoveStart( mch, setup );       //一定速度動作開始
        ・・・
        while ( DI[0] & 0x02 ) { }     //デジタル I/O_0 番のビット 1 番が“1”なら待機。
        JogJ_Set ( mch, 0x01, 0 );     //1 軸目の一定速動作 0 速度設定
        JogJ_Set ( mch, 0x04, 0 );     //3 軸目の一定速動作 0 速度設定
        MoveStart ( mch, setup );      //一定速動作停止（0 速度停止）
        ・・・
}
```

【 備考 】 （A）目標設定は 1 軸づつ設定します。

■ **void Mov※J_Set (int mch, int setup, int pos int vel)**

※a：絶対位置　i：相対位置

【 説明 】　絶対位置（MovaJ）・相対位置（Movij）位置決め動作の目標設定を行います。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
pos	：	目標位置（単位：位置指令単位）
vel	：	目標速度（単位：速度指令単位）

【 戻値 】　無し

【 使用例 】

```
#pragma main
int mch = 0;
int setup = 0x05;
int pos = 10000;
int vel = 1000;

void main ( void )
{
        ・・・
        ServoOn ( mch, setup );              //グループ 0 番 1、3 軸目　サーボオン
        MovaJ_Set ( mch, 0x01, vel, pos);    //1 軸目の位置決め動作設定
        MovaJ_Set ( mch, 0x04, vel, pos);    //3 軸目の位置決め動作設定
        MoveStart ( mch, setup );            //位置決め動作開始
        InposA ( mch, setup );               //1、3 軸目位置決め完了待ち
}
```

【 備考 】　（A）目標設定は 1 軸づつ設定します。

■ **void Mov※JT_Set (int mch, int setup, int pos int usec)**

※a：絶対位置　i：相対位置

【 説明 】　絶対位置（MovaJ）・相対位置（Movij）時間指定位置決め動作の目標設定を行います。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
pos	：	目標位置（単位：位置指令単位）
usec	：	目標時間（単位：usec）

【 戻値 】　無し

【 使用例 】

```
#pragma main
int mch = 0;
int setup = 0x05;
int pos = 10000;
int usec = 3000000;

void main ( void )
{
        ・・・
        ServoOn ( mch, setup );                 //グループ 0 番 1、3 軸目　サーボオン
        MovaJT_Set ( mch, 0x01, pos, usec );    //1 軸目の時間指定位置決め動作設定
        MovaJT_Set ( mch, 0x04, pos, usec );    //3 軸目の時間指定位置決め動作設定
        MoveStart ( mch, setup );               //時間指定位置決め動作開始
        InposA ( mch, setup );                  //1、3 軸目位置決め完了待ち
}
```

【 備考 】　（A）目標設定は 1 軸づつ設定します。

（B）目標時間は補間計算完了までの時間です。

加減速時定数が設定されている場合は、位置決め完了時間が時定数設定時間だけ長くなります。

■ **void Mov※JBL_Set (int mch, int setup, int pos int vel1, int vel2, int acc1, int acc2)**

※a：絶対位置　i：相対位置

【 説明 】　絶対位置（MovaJ）・相対位置（Movij）ベル型加減速位置決め動作の目標設定を行います。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
pos	：	目標位置（単位：位置指令単位）
vel1	：	目標初速（単位：速度指令単位）
vel2	：	目標終速（単位：速度指令単位）
acc1	：	加速度係数 1（単位：0.0001%）
acc2	：	加速度係数 2（単位：0.0001%）

【 戻値 】　無し

【 使用例 】

```
#pragma main
int mch = 0;
int setup = 0x01;
int pos = 10000;
int vel1 = 0, vel2 = 1000;
int acc1 = 0, acc2 = 0;

void main ( void )
{
        ・・・
        ServoOn ( mch, setup );                          //グループ 0 番 1 軸目　サーボオン
        MovaJBL_Set ( mch, setup, pos, vel1, vel2, acc1, acc2 );      //1 軸目加速領域設定
        MoveStart ( mch, setup );                        //ベル型加減速位置決め動作開始
        MovaJBL_Set ( mch, setup, pos*2, vel2, vel2, acc1, acc2 );    //1 軸目等速領域設定
        PassA ( mch, setup );                            //1 軸目補間計算完了待ち
        MoveStart ( mch, setup );                        //ベル型加減速位置決め動作開始
        MovaJBL_Set ( mch, setup, pos*3, vel2, vel1, acc1, acc2 );    //1 軸目減速領域設定
        PassA ( mch, setup );                            //1 軸目補間計算完了待ち
        MoveStart ( mch, setup );                        //ベル型加減速位置決め動作開始
        InposA ( mch, setup );                           //1 軸目位置決め完了待ち
}
```

【 備考 】 (A) 目標設定は 1 軸づつ設定します。

(B) Mov※JBL_Set 関数は複合動作コマンドです。複数のコマンドで加速―等速―減速を実現します。

(C) 複合動作コマンドを使用する場合は、初速と終速の値を正しく設定します。

■ **void Mov※JARC_Set (int mch, int setup, int pos int vel, int mode, int opt)**

※a：絶対位置　i：相対位置

【 説明 】　絶対位置（MovaJ）・相対位置（Movij）中心・角度指定円弧補間動作の目標設定を行います。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
pos	：	目標位置（単位：位置指令単位）
vel	：	目標速度（単位：速度指令単位）
mode	：	モード　3：x 軸（中心指定、CCW）
		2：x 軸（中心指定、CW）
		1：x 軸（始点/終点角度指定）
		0：同期軸（z、Θ軸）
		-1：y 軸
opt	：	補助データ（中心座標または始点・終点角度）
		中心座標　：x、y 軸の現在位置から相対位置を設定
		（単位：位置指令単位）
		始点・終点角度：x 軸設定時に始点角度、y 軸指定時に終点角度を設定
		（単位：0.001deg）

【 戻値 】　無し

【 使用例 】

```
#pragma main
int mch = 0;
int setup_x = 0x01, setup_y = 0x02;
int xpos = 0, ypos = 0;
int cx = 5000, cy = 0;
int vel = 1000;

void main ( void )
{
        ・・・
    ServoOn ( mch, setup_x | setup_y );          //グループ 0 番 1、2 軸目　サーボオン
    MovaJARC2_Set ( mch, setup_x, xpos, vel, 2, cx );  //1 軸目（x 軸）円弧動作設定
    MovaJARC2_Set ( mch, setup_y, ypos, vel, -1, cy );  //2 軸目（y 軸）円弧動作設定
    MoveStart ( mch, setup_x | setup_y );              /円弧補間動作開始
    InposA ( mch, setup_x | setup_y );           //90 度円弧補間動作完了待ち
}
```

【 備考 】 (A) 目標設定は 1 軸づつ設定します。

(B) 円弧の軌跡を真円にする為には、現在位置を基準に目標位置を正しく設定する必要があります。
コントローラは現在位置、目標位置から円弧中心座標を求めます。

動作制御命令　詳細

■ **void MoveStart (int mch, int setup)**

【 説明 】 軸動作を開始します。

【 引数 】 mch ： 機構（グループ）番号
setup ： 設定軸番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoOn ( 0, 0x0C );              //グループ 0 番 3、4 軸目　サーボオン
        MovaJ_Set ( 0, 0x04, 10000, 1000) ;      //3 軸目位置決め動作設定
        MovaJ_Set ( 0, 0x08, 10000, 1000) ;      //4 軸目位置決め動作設定
        MoveStart ( 0, 0x0C );            //3、4 軸目動作開始
        InposA ( 0, 0x0C );               //3、4 軸目位置決め完了待ち
        ・・・
}
```

【 備考 】 （A）MoveStart 関数の setup に 0 を指定された軸の動作設定データはクリアされます。
設定データがクリアされた軸を動作させるためには、動作設定を再設定する必要があります。

■ **void MoveStopAll (int mch, int lvl, int aft)**

【 説明 】 機構（グループ）内全軸の動作を停止します。

【 引数 】 mch ： 機構（グループ）番号
lvl ： 停止要求レベル（0：停止処理無し 1：減速停止 2：即停止）
aft ： 停止後処理（0：処理無し 1：サーボオフ）

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        ServoOn ( 0, 0x0C );          //グループ 0 番 3、4 軸目　サーボオン
        JogJ_Set ( 0, 0x04, 1000) ;   //3 軸目一定速動作設定
        JogJ_Set ( 0, 0x08, 1000) ;   //4 軸目一定速動作設定
        MoveStart ( 0, 0x0C );        //3、4 軸目動作開始
        ・・・
        while ( 1 )
        {
                if ( DI[0] & 0x80 )           //デジタル I/O_0 番のビット 7 番が " 1 " か
                {
                        MoveStopAll ( 0, 1, 0 );      //減速停止
                }
        }
}
```

【 備考 】 （A）速度制御・電流制御・原点復帰（HomeServo、HomeBump）モード時の減速停止処理は、
ドライバ内部の設定値（減速）に従います。

■ **void MoveStopAxis (int mch, int setup, int lvl, int aft)**

【 説明 】 機構（グループ）内全軸の動作を停止します。

【 引数 】
mch	：	機構（グループ）番号
setup	：	設定軸番号
lvl	：	停止要求レベル（0：停止処理無し　1：減速停止　2：即停止）
aft	：	停止後処理（0：処理無し　1：サーボオフ）

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        TaskStart ( axis_stop_check );  //タスク 4 番スタート
        ServoOn ( 0, 0x0C );            //グループ 0 番 3、4 軸目　サーボオン
        JogJ_Set ( 0, 0x04, 1000) ;     //3 軸目一定速動作設定
        JogJ_Set ( 0, 0x08, 1000) ;     //4 軸目一定速動作設定
        MoveStart ( 0, 0x0C );          //3、4 軸目動作開始
        ・・・
}

#pragma task4
void axis_stop_check ( void )
{
        while ( 1 )
        {
                if ( DI[0] & 0x80 )             //デジタル I/O_0 番のビット 7 番が “ 1 ” か
                {
                        MoveStopAxis ( 0, 0x0C, 1, 1 ); //減速停止＋サーボオフ
                }
        }
}
```

【 備考 】 （A）速度制御・電流制御・原点復帰（HomeServo、HomeBump）モード時の減速停止処理は、ドライバ内部の設定値（減速）に従います。

■ **void OVR_Set(int mch, int typ, int ovr_no, int data1, int data2)**

【 説明 】 指定された速度オーバーライド値を設定します。

【 引数 】

mch	：	機構（グループ）番号
typ	：	データタイプ（0：0.01%　1：分子分母）
ovr_no	：	オーバーライド番号
data1	：	データタイプ 0 の時：オーバーライド値　データタイプ 1 の時：分子
data2	：	データタイプ 0 の時：無効　データタイプ 1 の時：分子

【 戻値 】 無し

【 使用例 】

```
#pragma main
int ovr1, ovr2;
void main ( void )
{
        ・・・
        ovr1 = 30:                          //分子を 30 に設定
        ovr2 = 100;                         //分母を 100 に設定
        OVR_Set ( 0, 1, 0, ovr1, ovr2 )     //速度オーバーライド 0 番を 30.00%に設定
        ・・・
}
```

【 備考 】 （A）速度オーバーライドは機構毎に 4 個あり、最終的にすべてをじた値が使われます。

■ **void OVR_Get(int mch, int typ, int ovr_no, int &var_ad1r, int &var_adr2)**

【 説明 】 指定された速度オーバーライド値を取得します。

【 引数 】

mch	：	機構（グループ）番号
typ	：	データタイプ（0：0.01% 1：分子分母）
ovr_no	：	オーバーライド番号
&var_adr1	：	データタイプ 0 の時：オーバーライド値 データタイプ 1 の時：分子
&var_adr2	：	データタイプ 0 の時：無効 データタイプ 1 の時：分子

【 戻値 】 無し

【 使用例 】

```
#pragma main
int ovr1, ovr2;

void main ( void )
{
        ・・・
        OVR_Get ( 0, 1, 0, &ovr1, &ovr2 )        //速度オーバーライド 0 番の分子分母を取得
        ・・・
}
```

【 備考 】 （A）速度オーバーライドは機構毎に 4 個あり、最終的にすべてをじた値が使われます。

（B）データタイプを 0.01%で取得する場合、小数点の端数を考慮ください。

テーブル命令　詳細

■ void TableTeach (int mch, int setup, int tbl_no)

【 説明 】 ティーチングを実行します。
指定された軸の現在位置データを、指定された位置テーブルへ設定します。

【 引数 】
mch : 機構（グループ）番号
setup : 設定軸番号
tbl_no : テーブル番号

【 戻値 】 無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( RS[0] & 0x04 )              //ネットワーク変数 RS[0]のビット 2 番が " 1 " か
        {
                TableTeach ( 0, 0x0F, 10 );      //グループ 0 番 1～4 軸目の現在位置を
                                                 //各位置テーブルへ設定します。
        }
        ・・・
}
```

【 備考 】 （A）テーブルデータの範囲はコントローラの仕様により可変します。
詳細はコントローラの仕様書を参照ください。

■ void TableSave (void)

【 説明 】　テーブルデータをフラッシュメモリに保存します。

【 引数 】　無し

【 戻値 】　無し

【 使用例 】

```
#pragma main

void main ( void )
{
        ・・・
        if ( RS[0] & 0x08 )             //ネットワーク変数 RS[0]のビット 3 番が “ 1 ” か
        {
                TableSave ( );          //テーブルデータ保存
                Waitmsec ( 5000 );      //保存完了待ち
        }
        ・・・
}
```

【 備考 】　（A）テーブルデータはすべて保存されます。

（B）TableSave 関数実行後、保存完了まで数秒掛かります。
連続で保存されることを防止する為に TableSave 関数実行後に保存完了待ちを実行します。

■ **void TableMov※P (int mch, int setup, int tbl_no)**

※a：絶対位置　i：相対位置

【 説明 】　絶対位置（TableMovaP）・相対位置（TableMoviP）テーブル位置決め動作を行います。
SetMov※J_Set 関数と異なり、一度に複数軸の動作が可能です。
TableMov※P 関数の後に MoveStart 関数は不要です。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
tbl_no	：	テーブル番号（位置・速度テーブル）

【 戻値 】　無し

【 使用例 】

```
#pragma main
int mch = 0;
int setup = 0x0F;

void main ( void )
{
        ・・・
        int i = 0;
        while ( i < 10 )                        //i が 0～9 までループ
        {
                TableMovaP ( mch, setup, i );   //テーブル番号 0～9 まで位置決め開始
                PassA (mch, setup );
        }
}
```

【 備考 】　（A）TableMovaP 関数を使用する場合、指定するテーブル番号（位置・速度テーブル）に、
あらかじめデータが設定されている必要があります。
（B）位置・速度テーブルは、それぞれ位置指令単位と速度指令単位として動作命令の目標値に
設定されます。

■ **void TableMov※T (int mch, int setup, int tbl_no)**

※a：絶対位置　i：相対位置

【 説明 】　絶対位置（TableMovaP）・相対位置（TableMoviP）時間指定テーブル位置決め動作を行います。
SetMov※JT_Set 関数と異なり、一度に複数軸の動作が可能です。
TableMov※T 関数の後に MoveStart 関数は不要です。

【 引数 】

mch	：	機構（グループ）番号
setup	：	設定軸番号
tbl_no	：	テーブル番号（位置・時間テーブル）

【 戻値 】　無し

【 使用例 】

```
#pragma main
int mch = 0;
int setup = 0x0F;

void main ( void )
{
        ・・・
        int i = 0;
        while ( i < 10 )                        //i が 0～9 までループ
        {
                TableMovaT ( mch, setup, i );   //テーブル番号 0～9 まで位置決め開始
                PassA (mch, setup );
        }
}
```

【 備考 】　（A）TableMovaT 関数を使用する場合、指定するテーブル番号（位置・時間テーブル）に、
あらかじめデータが設定されている必要があります。
（B）位置・時間テーブルは、それぞれ位置指令単位と msec として動作命令の目標値に
設定されます。