

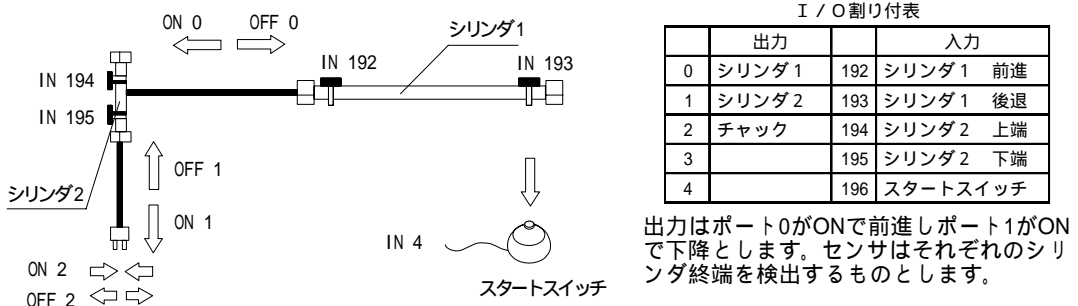
第2章 インタプリタ言語でのプログラミング

コントローラのいきさつについて

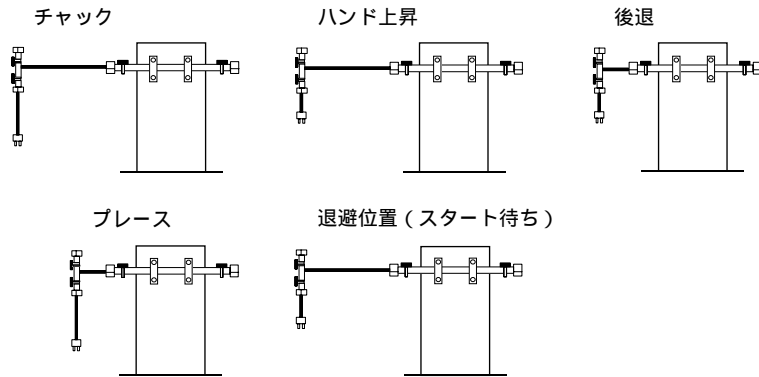
現在のコンピュータの姿が確立したのは1970年頃のことです。またソフトウェア技術についても同時期に現在の基礎はできあがっていました。しかし当時のコンピュータはまだ実用化の初期段階でとても高価でしたし、デリケートでもあったため現場の機械の制御をしたり、その大きさから装置に埋め込んで使用するなどということは不可能でした。コンピュータは主に政府大企業の中の膨大なデータを処理するためだけに用いられていました。それでも1970年ころのアメリカの自動車産業は最盛期を迎えており巨大な工場と多くの制御を必要とする装置がありました。当時の装置はリレー回路による論理制御と、カムによる制御がほとんどでした。機械設計者はカラクリ仕掛をつくるように綿密に装置とカムを設計し機械自体で動くものをつくりました。ただこの方法では装置に柔軟性をもたせることが難しく調整完了までに熟練工と時間を必要としていました。またリレー回路による制御では、プログラム変更は配線変更を意味し、ちょっとしたシーケンサ変更も多くの作業を必要としました。アメリカの自動車工場のなかでこうしたリレー回路制御をなんとかもう少し保守性が高く、作業のしやすいものがないかということで現在のシーケンサのルーツになるものが考案されたそうです。そしてこの1970年はマイクロプロセッサ黎明期でもありました。この時代は大変多くのことが同時進行していたわけです。やがて1980年になるとマイクロプロセッサは量産されるようになり次第に機器制御に用いられるようになってきました。そして、その中でシーケンサと呼ばれる専用のコントローラが市販されるようになってきました。シーケンサはリレー回路をマイコンで模するもので、リレー回路の欠点を補いコンパクトかつ保守性の高いものです。当初のものは制御点数も少なくプログラムの入力方法も機械語に近い状態のものでしたが、最近では1000点近い規模とリレー回路をそのまま書き込める回路設計のしやすいものになってきました。シーケンサはリレー回路を模するという初期の目的に忠実に発展してきました。現在でもシーケンサのプログラミング言語の主流はラダー言語と呼ばれるリレー回路を記述するものとなっています。しかし制御の実態は単純なI/O制御からNC制御へと主流が移ってきています。安価になったステップモータやサーボモータが専用機に組み込まれそれとともに通信も必要になってきています。制御に対する考え方(言語)はそのままですが対象は大きく変わってきています。これに対してマイクロプロセッサ技術はハード・ソフトとともに長足の進歩をとげ現在では、卓上型のパソコンですら1970年の先端コンピュータの性能をしのぎソフトウェアも昔とは比較にならない強力なマンマシンインターフェースを備えています。シーケンサがリレー回路の域を脱出ししないのに対して対照的です。私どものMPC-684はNC制御、通信を目的に企画されています。シーケンサのラダー言語では数値・通信を素直に扱うことができません。そのことが、装置の柔軟性・機能を損ねているのではないかと考えられます。ADVFSはBASICライクなインタプリタです。しかし、通常のBASICとは大きくことなり、I/O制御やパルス発生をももとの機能として含んでおり制御用インタプリタとして開発されています。当初BASICインタプリタはコンピュータのビギナのために考案された言語ですがその簡便性と実用性のため現在でも広く使用されている言語です。デバッグや昨今流行の構造的プログラミングという観点ではやや難がありますが一人でコーディングできる程度のアプリケーションにはその起動性は魅力的です。こうした環境を制御の分野に実現しようとしたのがMPC-684です。

インタプリタ言語でのプログラミング

ラダー言語とインタプリタ言語（MPC684ではADVFSC）ではプログラミングの考え方がまったく異なっています。（この章はラダー言語をご存知の方のためです。MPCのみ検討されている方は読み飛ばしてください）ラダー言語の基本は電子回路で物事を考えます。たとえばこのセンサとあのセンサがONしたらこのソレノイドを動作するというようにです。これに対してADVFSCでは装置を時間の流れの中でとらえます。一番に何々して二番目に何々してというふうにシーケンスを字義どおり考えます。制御の分野の人の中にはときどき機械の動作を考えた瞬間にリレー回路が頭の中に浮かぶ人も多いと思います。たまにある誤解はこんな方がADVFSCでこうしたリレー回路を表現することです。残念ながらADVFSCはリレー回路を表現することは得意ではありません。リレー回路の観点にたてばADVFSCは良い制御言語とは言えません。ここでは、単純なP&Pの制御例としてとりあげてみます。構成は次のようなものとします。



作業としては次の図のようなフローとします。パワーオン後の退避状態（OFF0、OFF1の状態）から前進下降しワークをつかみ、その後上昇し、後退し、さらに下降してワークを置きます。（プレス）そして上昇しさらに前進の上その状態でタイミング待ちとなります。



前記の動作をADVFSCで表現すると次のようになります。
P&Pサンプルプログラム

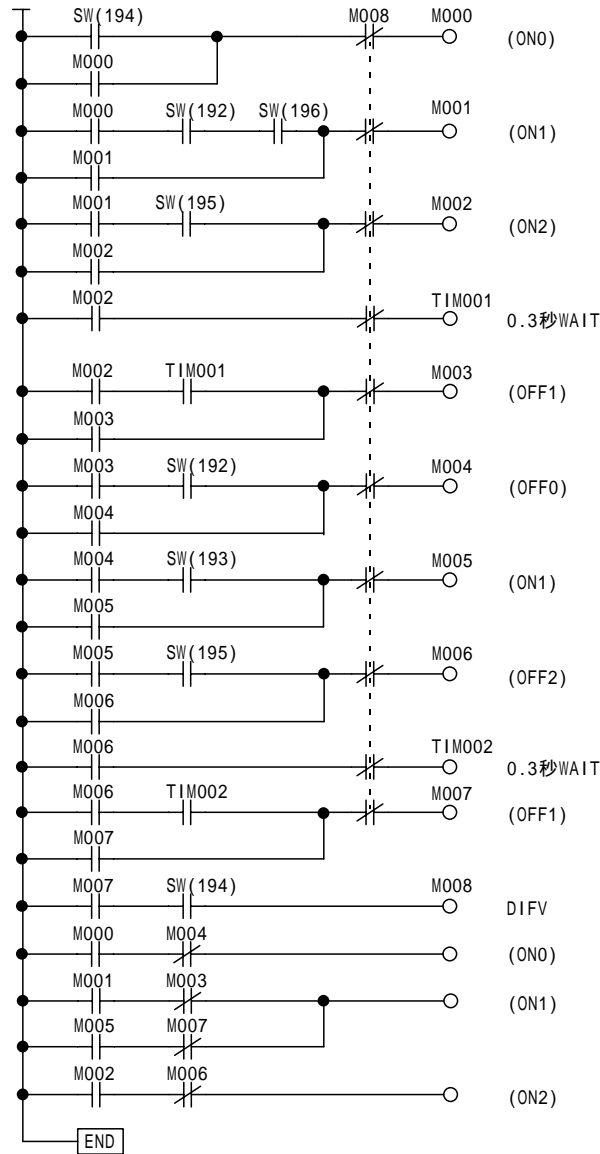
```

DO
ON 0
WAIT SW(192)==1
WAIT SW(196)==1
ON 1
WAIT SW(195)==1
ON 2
TIME 300
OFF 1
WAIT SW(194)==1
OFF 0
WAIT SW(193)==1
ON1
WAIT SW(195)==1
    
```

- ' ハンド前進
- ' 前進検出
- ' スタート待ち
- ' ハンド下降
- ' 下降検出
- ' チャック
- ' 0.3秒待ち
- ' ハンド上昇
- ' 上昇検出
- ' ハンド後退
- ' 後退検出
- ' ハンド下降
- ' 下降検出

| | |
|-----------------|----------|
| OFF 2 | ' チャック解放 |
| TIME 300 | ' 0.3秒待ち |
| OFF 1 | ' ハンド上昇 |
| WAIT SW(194)==1 | ' 上昇検出 |
| LOOP | |

このようにADVFSCでは、動作を時間を追って順に記述しますがシーケンサでは、全く異なった考え方でプログラムします。シーケンサのラダー図で同等の動作を表現すると次の図のようになります。

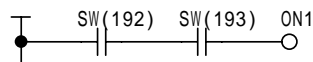


シーケンサでは、装置の動作を回路で実現するものとしてプログラムします。これは、コントローラのいきさつについてで述べたように自動機がリレー回路によって制御されており、シーケンサはこれを”ソフト化”するために開発されたものであることを意味しています。両者のプログラム方法の違いは根本的なもので、目的は同じ結果でも異質なものと言えます。

ここでは例として、P&PをとりあげたためにADVFSCの例が明解でラダー図が冗長なものとなっていますが、対象となる機器を替えると状況が変わることがあります。工場の現場で使用されている自動機の殆どは単純なプレイバックマシンですが、これらは簡単な論理から成立しています。こうした場合はラダー図の方がむしろシンプルです。例えば、コンベア制御の殆どが数個のセンサの論理から出力応答を得ます。

一定の大きさのワークが来たらストッパーをONするというロジックは次の図のようになります。これに対してインタブリタでは次のようにやや難しそうな雰囲気を持つプログラムになります。

```
100 IF SW(192)==1 AND SW(193)==1 THEN
110 ON 0
120 END_IF
```



ラダー図とインタブリタのプログラミングを比較すると一般に次のようなことが言えます。

ラダー言語でのプログラムの特徴

- ・ 動作を回路で表現するために論理表現は素直であるが、P&Pのように時間順序を持つ制御のプログラムは自己保持リレーのようなラダー独特の考え方を必要として冗長になりやすい。
- ・ プログラムの基本概念に数値操作が含まれていないために数を扱う制御(パルス発生等)を直接扱うことができない。
- ・ リレー回路によってプログラムを表現するので、回路そのもののデバッグはプロコンの発達に伴い非常に容易となっている。只プロコンは専用機であることが多く高価であることを免れない。
- ・ リレー回路であることから装置の仕様変更がプログラムの全体に波及することが多い。このことは保守性の困難さにつながってくる。規模の大きなプログラムでは自分の書いたものですら読み返せない。

ADVFSCの特徴

- ・ 動作をインタブリタで表現するため論理よりも物語のようにプログラムを記述するスタイルとなる。P&Pや軸制御のように時間順序を持つ制御は素直に記述できる。
- ・ プログラムの基本概念に数値制御が含まれており、パルス発生等のステップモータやサーボモータの制御を直接素直に扱うことができる。
- ・ デバッグの方法が、リレー回路のように具体的でなく変数や実行文番号のモニターを介して間接的に行う。このためプログラミングに対する基礎的な教養が身に付いていないと迷子になる。
- ・ 装置の仕様変更に対して柔軟性が高く部分の変更は全体に及ばない。又、プログラムの整理の仕方にもよるが一般に読み直し易い。これは、稼働後の変更の際に部分の変更が全体に及ばないことによる。

これらは、どちらが良いとか悪いのではなく性質の違いを表しています。制御すべき装置の動作がリレー回路で記述しやすいのであれば、ラダーは有効でADVFSCは記述可能ということにとどまります。もし、ステップモータやサーボモータが無く多品種少量という目的が無ければラダー図で全ての装置を効率よく記述できます。只、最近の装置は多軸制御をベースとした多品種少量の傾向が強まっており、ADVFSCが有用となる機会が増えているものと考えられます。

プログラムの基本テクニック

ADVFSCではプログラムを効率よく記述するための方法が用意されています。

I/Oのシンボル化

I/Oを番号のみで表現すると意味不明の場合があります。ADVFSCではCONSTコマンドで定数を定義することによって入出力をラベルで表現することができます。

```
      P&Pサンプルプログラム
CONST SOLO 0          シリンダ 1
CONST SOL1 1          シリンダ 2
CONST HAND 2          チャック
CONST CL1F 192        前進端センサ
CONST CL1B 193        後進端センサ
CONST CL2U 194        上端センサ
CONST CL2D 195        下端センサ
CONST START 196       スタートスイッチ
DO
  ON SOLO              ハンド前進
  WAIT SW(CL1F)==1     前進検出
  WAIT SW(START)==1   スタート待ち
  ON SOL1              ハンド下降
  WAIT SW(CL2D)==1     下降検出
  ON HAND              チャック
  TIME 300             0.3秒待ち
  OFF SOL1             ハンド上昇
  WAIT SW(CL2U)==1     上昇検出
  OFF SOLO             ハンド後退
  WAIT SW(CL1B)==1     後退検出
  ON SOL1              ハンド下降
  WAIT SW(CL2D)==1     下降検出
  OFF HAND             チャック解放
  TIME 300             0.3秒待ち
  OFF SOL1             ハンド上昇
  WAIT SW(CL2U)==1     上昇検出
LOOP
```

マルチタスク

インタプリタではプログラムが時間順序で書かれるためにひとつの動作しか記述できません。しかし、実際の装置ではいくつもの要素が同時に進行するために複数のプログラムを動作させる必要があります。ADVFSCではFORKコマンドにより複数のプログラムを動作させることができます。例としては、先のP&Pを2個同時にサポートするものを想定して次のサンプルプログラムを作りました。ここでは互いのP&P#1と#2は全く別の装置として独立して動作します。尚、ADVFSCのマルチタスクは32本です。

```
      P&Pサンプルプログラム#1
CONST SOLO 0          シリンダ 1
CONST SOL1 1          シリンダ 2
CONST HAND 2          チャック
CONST CL1F 192        前進端センサ
CONST CL1B 193        後進端センサ

CONST CL2U 194        上端センサ
CONST CL2D 195        下端センサ
CONST START 196       スタートスイッチ
,
CONST so10 4          シリンダ 1
CONST so11 5          シリンダ 2
CONST hand 6          チャック
CONST c11F 197        前進端センサ
CONST c11B 198        後進端センサ
CONST c12U 199        上端センサ
```

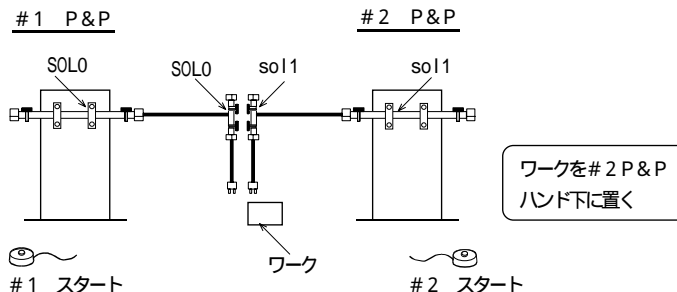
```

CONST c12D 200          ' 下端センサ
CONST start 201        ' スタートスイッチ
,
FORK 1 *P&P#2
*P&P#1
DO
  ON SOLO                ' ハンド前進
  WAIT SW(CL1F)==1      ' 前進検出
  WAIT SW(START)==1    ' スタート待ち
  ON SOL1                ' ハンド下降
  WAIT SW(CL2D)==1    ' 下降検出
  ON HAND                ' チャック
  TIME 300              ' 0.3秒待ち
  OFF SOL1              ' ハンド上昇
  WAIT SW(CL2U)==1    ' 上昇検出
  OFF SOLO              ' ハンド後退
  WAIT SW(CL1B)==1    ' 後退検出
  ON SOL1                ' ハンド下降
  WAIT SW(CL2D)==1    ' 下降検出
  OFF HAND              ' チャック解放
  TIME 300              ' 0.3秒待ち
  OFF SOL1              ' ハンド上昇
  WAIT SW(CL2U)==1    ' 上昇検出
LOOP
,
P&Pサンプルプログラム#2
*P&P#2
DO
  ON sol0                ' ハンド前進
  WAIT SW(c11F)==1      ' 前進検出
  WAIT SW(start)==1    ' スタート待ち
  ON sol1                ' ハンド下降
  WAIT SW(c12D)==1    ' 下降検出
  ON hand                ' チャック
  TIME 300              ' 0.3秒待ち
  OFF sol1              ' ハンド上昇
  WAIT SW(c12U)==1    ' 上昇検出
  OFF sol0              ' ハンド後退
  WAIT SW(c11B)==1    ' 後退検出
  ON sol1                ' ハンド下降
  WAIT SW(c12D)==1    ' 下降検出
  OFF hand              ' チャック解放
  TIME 300              ' 0.3秒待ち
  OFF sol1              ' ハンド上昇
  WAIT SW(c12U)==1    ' 上昇検出
LOOP
,

```

タスク間インタロック (セマフォ関数 RSV(), RLS())

複数のタスクによってサポートされた機器が一つの装置の上で動作する時には、干渉もしくは同期という問題があります。例えば、先の例でのP&Pが次の図のように配置されており早くボタンを押されたP&Pがワークをとるものとすると、インタロックが必要となります。このためにADVFSCLは、メモリI/Oとセマフォ関数が用意されています。セマフォとは、マルチタスクで広く用いられているインタロック用の関数で矛盾なくインタロックを実現します。RSV()、RLS()についてはコマンドリファレンスを参照して下さい。



```

' P&Pサンプルプログラム#1
CONST SOLO 0 ' シリンダ 1
CONST SOL1 1 ' シリンダ 2
CONST HAND 2 ' チャック
CONST CL1F 192 ' 前進端センサ
CONST CL1B 193 ' 後進端センサ
CONST CL2U 194 ' 上端センサ
CONST CL2D 195 ' 下端センサ
CONST START 196 ' スタートスイッチ
,
CONST so10 4 ' シリンダ 1
CONST so11 5 ' シリンダ 2
CONST hand 6 ' チャック
CONST c11F 197 ' 前進端センサ
CONST c11B 198 ' 後進端センサ
CONST c12U 199 ' 上端センサ
CONST c12D 200 ' 下端センサ
CONST start 201 ' スタートスイッチ
,
CONST memo1 -1 ' セマフォ用メモリ I/O
FORK 1 *P&P#2
*P&P#1
DO
  WAIT SW(START)==1 ' スタート待ち
  WAIT RSV(memo1)==0 ' セマフォ獲得
  ON SOLO ' ハンド前進
  WAIT SW(CL1F)==1 ' 前進検出
  ON SOL1 ' ハンド下降
  WAIT SW(CL2D)==1 ' 下降検出
  ON HAND ' チャック
  TIME 300 ' 0.3秒待ち
  OFF SOL1 ' ハンド上昇
  WAIT SW(CL2U)==1 ' 上昇検出
  OFF SOLO ' ハンド後退
  WAIT SW(CL1B)==1 ' 後退検出
  ON SOL1 ' ハンド下降
  WAIT SW(CL2D)==1 ' 下降検出
  OFF HAND ' チャック解放
  TIME 300 ' 0.3秒待ち
  OFF SOL1 ' ハンド上昇
  WAIT SW(CL2U)==1 ' 上昇検出
  a=RLS(memo1) ' セマフォ解放
LOOP
,
' P&Pサンプルプログラム#2
*P&P#2
DO
  WAIT SW(start)==1 ' スタート待ち
  WAIT RSV(memo1)==0 ' セマフォ獲得
  ON so10 ' ハンド前進
  WAIT SW(c11F)==1 ' 前進検出
  ON so11 ' ハンド下降
  WAIT SW(c12D)==1 ' 下降検出
  ON hand ' チャック
  TIME 300 ' 0.3秒待ち
  OFF so11 ' ハンド上昇
  WAIT SW(c12U)==1 ' 上昇検出
  OFF so10 ' ハンド後退
  WAIT SW(c11B)==1 ' 後退検出
  ON so11 ' ハンド下降
  WAIT SW(c12D)==1 ' 下降検出
  OFF hand ' チャック解放
  TIME 300 ' 0.3秒待ち
  OFF so11 ' ハンド上昇
  WAIT SW(c12U)==1 ' 上昇検出
  a=RLS(memo1) ' セマフォ解放
LOOP

```

非常停止・モード切り換え

装置制御の基本に一時停止やモード切り換えがあります。インタプリタでは順次制御の途中で入力を検出して動作を停止したり、変更することは本来難しいものです。しかし、マルチタスクを使用するとこれは比較的容易にできます。次に一時停止とモード切り換えの例を示します。モード切り換えはポート 197 のスイッチ一時停止は198としています。又、モード切り換えがオートからマニュアルに切り換わる場所では再スタートスイッチ199を通してポートをクリアすることとしています。こうした後始末に相当するプログラムは装置によって大きく異なります。この例で重要なことは動作のプログラムをタスク#1に移して主タスクでは制御スイッチの操作を監視し、動作タスク#1をFORK,QUIT,CONTで制御することにあります。又、マニュアルモード*MANU～GOTO*MAINまでのプログラムは入力ポートをパラレルで読み取って出力ポートにそのままパラレル出力しています。これはマニュアルモードでスイッチにより各出力を単動させる場合に有効なプログラムです。

```
' P&Pサンプルプログラム#1
CONST SOLO 0          ' シリンダ 1
CONST SOL1 1         ' シリンダ 2
CONST HAND 2         ' チャック
CONST CL1F 192       ' 前進端センサ
CONST CL1B 193       ' 後進端センサ
CONST CL2U 194       ' 上端センサ
CONST CL2D 195       ' 下端センサ
CONST START 196      ' スタートスイッチ
CONST MODE 197       ' モード・スイッチ
CONST PAUSE 198      ' 一時停止スイッチ
CONST RSTART 199    ' 再スタートスイッチ
'
*MAIN
IF SW(MODE)==1 THEN  ' オートマニュアル識別
  GOTO *AUTO
ELSE
  GOTO *MANU
END_IF
*AUTO
FORK 1 *P&P#1        ' オート時制御
DO WHILE SW(MODE)==1
  IF SW(PAUSE)==1 THEN
    PAUSE 1
    WAIT SW(PAUSE)==0
    CONT 1
  END_IF
  TIME 100
LOOP
QUIT 1
WAIT SW(RSTART)==1
OFF SOLO SOL1 HAND
GOTO *MAIN
*MANU
DO WHILE SW(MODE)==0  ' マニュアル時制御
  port=IN(25)
  TIME 100
  OUT port 0          ' パラレル入力パラレル出力
LOOP
GOTO *MAIN
*P&P#1
DO
  WAIT SW(START)==1  ' スタート待ち
  ON SOLO             ' ハンド前進
  WAIT SW(CL1F)==1   ' 前進検出
  ON SOL1             ' ハンド下降
  WAIT SW(CL2D)==1   ' 下降検出
  ON HAND            ' チャック
  TIME 300           ' 0.3秒待ち
  OFF SOL1           ' ハンド上昇
  WAIT SW(CL2U)==1   ' 上昇検出
  OFF SOLO           ' ハンド後退
  WAIT SW(CL1B)==1   ' 後退検出
```


| | |
|------------------|----------|
| ON SOL1 | ' ハンド下降 |
| WAIT SW(CL2D)==1 | ' 下降検出 |
| OFF HAND | ' チャック解放 |
| TIME 300 | ' 0.3秒待ち |
| OFF SOL1 | ' ハンド上昇 |
| WAIT SW(CL2U)==1 | ' 上昇検出 |
| LOOP | |

制御文

さて、ここまでの例題でも様々な制御文が例示されてきましたが、ADVFCでは次の制御文が用意されています。

- ・ IF ~ THEN ~ ELSE ~ END_IF
- ・ IF ~ THEN ~ END_IF
- ・ DO ~ LOOP
- ・ FOR ~ NEXT
- ・ SELECT ~ CASE ~ CASE_ELSE ~ END_SELECT
- ・ GOTO ~
- ・ GOSUB ~ RETURN
- ・ WHILE ~ WEND
- ・ WAIT ~ WS0(),WS1()
- ・ BREAK

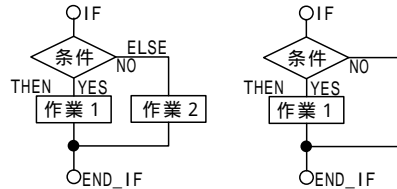
これらはいずれも求める条件判断や繰り返し作業にマッチした使い方をされるべきです。ここではそれぞれの制御文の意味を概説しておきます。詳細はコマンドリファレンスを参照して下さい。

・ IF ~ THEN ~ ELSE ~ END_IF

ひとつの条件から異なる動作を必要とする場合に有効です。

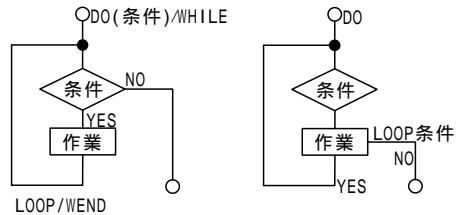
・ IF ~ THEN ~ END_IF

ある条件が成立した時にある動作する場合に用います。



・ DO ~ LOOP, WHILE ~ WEND

繰り返し動作で条件を付加することができます。DOもしくはLOOPのあとにWHILEあるいはUNTILという条件を与えることができます。WHILEは条件が成立している間UNTILは条件成立するまでという意味です。



・ FOR ~ NEXT

数の定まった繰り返し作業です。100回などという場合は次のように記述します。又、STEPを与えることによって数の増加を1だけでなくいろいろ数（負まで含めて）を与えることができます。

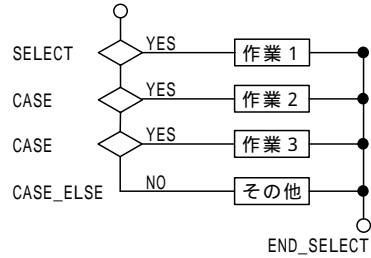
```
FOR   I=1 TO 100
      作業
NEXT  I
```

・ BREAK

IF ~ END_IF, DO ~ LOOP, WHILE ~ WEND, FOR ~ NEXTの制御フローを終了し、そのループ文の次の文から再開します。

・ SELECT ~ CASE ~ CASE_ELSE ~ END_SELECT

ひとつの入力によっていくつもの場合分けを必要とする場合に有効です。DSW (デジスイッチ) によって得た数にしたがって異なるプログラムを動作させる等が例となります。



・ GOTO ~ ,GOSUB ~ RETURN

GOTO ~ は単純にラベルで示された場所へ制御を移すことです。GOSUBはサブルーチンコールです。GOSUBで呼ばれたモジュールはRETURN文でGOSUBの下に戻ります。

・ WAIT ~ ,WS0(),WS1()

WAITは最も単純な条件待ちです。条件が成立するまでWAIT文から抜け出せません。これを関数化したものがWS0(),WS1()です。これにはタイムアウトが付加できます。

BASIC と比較して

ADVFSCは、BASICと良く似た言語です。これによりプログラムを親しみ易いものにしてユーザーの負担を軽くしています。只、装置制御を対象とした組み込み用ボードコンピュータという制限からその仕様は様々な点で簡略化されています。例えば、文字列変数は32個までに限定していますし、配列宣言も15までで次元に限定しています。関数は引き数を1つに限定しているため本来2つの引き数を必要とする関数もADVFSCに限った特別なラベルとなっています。(INP\$#0(),INP\$#2()) ももちろんADVFSCは装置制御を広くサポートするように設計されていますので、I/O 制御パルス発生等のアプリケーションには柔軟に効率良く対応します。ADVFSC/MPC-684が有用となるアプリケーションは主に次のようなものです。

- ・ パルス発生で数値制御を含むもの
- ・ 通信や演算を含む制御
- ・ 専用のプロコンを用いず汎用パソコンでプログラミングを必要とする場合
- ・ A/D、D/A等の汎用パソコンの周辺を用いる応用

又、BASICと大きく異なっているのは次のような事柄です。

- ・ 扱える変数が4byte長整数に限られていること
- ・ 配列は次元のみで15ラベルまで、又総数10000に限られていること
- ・ 文字列変数は32個までで長さが80文字以内
- ・ 一部関数が引き数1ヶの為変形されていることINP\$#0(),INP\$#2()