マルチ・マイクロコンピュータのシステム記述言語MPL

茅野昌明・浜田喬・安藤友久・山口剛*
（東京大学 生産技術研究所）

1、まえがき

近年のLSI技術の進歩によって，マイクロ・プロセッサが高性能でしかも低価格となったが，周辺機器については，機械的な部分も多いためそれほど安価ではない。従って，各ユーザがそれぞれマイクロ・コンピュータ・システムを用意し，それぞれに必要な周辺機器を手えたのでは負担が大きくなる。そこで，図1のように，個々のマイクロ・コンピュータ・システムを結合し，周辺機器を共有すると，性能価格比の良いシステムが構成できる。

さらに，このようなネットワークにI/O制御用プロセッサ，通信専用プロセッサなどの専用プロセッサを追加して，システム全体の性能を向上させることもできる。

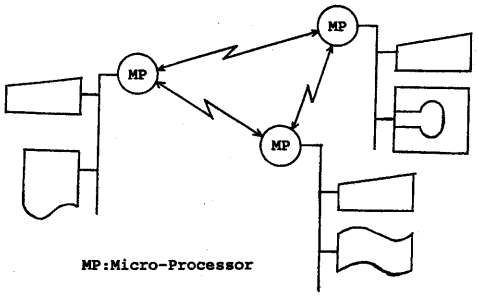

図1、Microcomputer Network

ところで，このマイクロ・コンピュータ・ネットワークのソフトウェアについて考えてみると，個々のノードごとにオペレーティング・システムを記述していたのでは，見通しが悪く，作成の能率も悪い。従って，ネットワークシステム全体を統一的に記述できるプログラミング言語が必要である。また，ユーザの利用目的に適合させるため，あるいは要求の増加によってシステムを拡張するためにシステムの構成を変えた場合にも，プログラムを書き直すのが容易な記述性の良い高水準言語が必要となる。

一方で，オペレーティング・システムなどの記述は，従来

i)速さ，サイズの両面で効率が良いこと
ii)アドレス操作，ビット演算が必要なこと
iii)I/Oデバイス操作や割込みを扱えること

などの理由からアセンブリ言語で行われて来たが，オペレーティング・システムが大規模で複雑になるにつれ，作成の労力が膨大なものとなり，保守・改良が困難になってきた。そこで，各方面でシステムを記述するための高水準言語の研究が進められている。特に，1973年にC.A.R. Hoareが提案したmonitorの概念をもとに，P. Brinch Hansenの設計したConcurrent Pascal（1975），N. Wirthの設計したModula（1977）は，並列に動作する複数のプロセスとそれらの共有変数を管理するモニタによって構造的にオペレーティング・システムを記述するものであった。

しかし，これらの言語はユニ・プロセッサを対象としており，プロセス間通信を共有変数によって行うため，共有空間をもたないマルチ・プロセッサ・システムを記述するのに適していない。

＊ 現在は富士通株式会社

ここでは，以上のようなシステム記述言語の研究成果をもとに，言語Modulaを母体として設計したマイクロ・コンピュータ・ネットワークのためのシステム記述言語ＭＰＬ（Multi-processors Programming Language）の基本仕様とその実装方法について述べる。

２．ＭＰＬの概要

ＭＰＬは，N. Wirthによって設計されたユニ・プロセッサ用システム記述言語Modulaを利用し，Modulaの特徴はそのまま残し，さらに次のようなマイクロ・コンピュータ・ネットワーク・システム記述言語に必要とされる機能を付加した言語である。

i)ネットワーク・システム全体を統一的にかつ効率よく記述できること
ii)プロセッサ間のメッセージ通信が記述できること

２．１　プログラムの構造

ＭＰＬプログラムの構造を図２に示す。processor module（ＰＭ）はネットワークの各ノード・プロセッサの動作を記述し，各プロセッサに共通な環境がglobal objectsである。ＰＭの中には，一般moduleとそれに含まれるprocess（Ｐ），入出力デバイスを扱うdevice module（ＤＭ）とそれに含まれるdevice process（ＤＰ），プロセッサ間の通信を扱うcommunication module（ＣＭ）とcommunication process（ＣＰ），および同一プロセッサ内の各プロセス（Ｐ，ＤＰ，ＣＰ）間の通信・同期を記述するinterface module（ＩＭ）がある。それぞれのmoduleは，機能的に密接な関係にある手続きの集合とそれらの手続きが取り扱うデータからなる。moduleは，その内部および外部で宣言された名前（identifier）の有効範囲を制御するための壁の役割をもち，module間のつながりを示すのが，define宣言とuse宣言である。前者は外部に対して使用を許す名前を定義するための宣言であり，後者は外部で宣言された名前でそのmodule内で使用が許される名前を定義する宣言である。

メインプログラムの本体（body）では，論理プロセッサ（プロセッサ名）と物理プロセッサ（プロセッサ番号）を対応づけている。

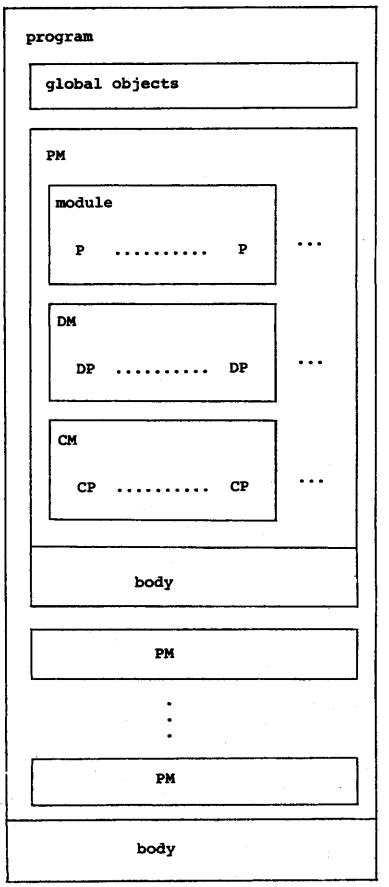

図２．ＭＰＬプログラムの構造

2．2　プロセッサ指定と並列文

ネットワーク・システムにおいては，各PMに共通な部分が多いのでそのような共通部分をまとめて記述するためにプロセッサ指定と並列文を導入している。

プロセッサ指定はプロセッサ名（たとえばP1，P2，P3）を"(:"と":)"で囲んだ部分を宣言の前に置いて，その宣言が有効なプロセッサを指定する。プロセッサ指定は入れ子にできるが，内側の指定は外側の指定に含まれなければならない。プロセッサ指定のない宣言は外側の指定によるプロセッサ名すべてに有効であるとみなされる。図3にその例を示すが，図3(a)のようにプロセッサ指定を用いたプログラムは，図3(b)のように別々のプロセッサモジュール内で宣言を行っても同じである。

```
(;P1,P2,P3:):processor module PM;

  (:P1,P2:):const A=1;
  (:P3:):   const A=2;
  (:P1:):var I:integer;
  (:P2,P3:):var C;char;
  var X,Y,Z:bits;

  module M;
    (:P1:):procedure PROC;
             begin ..... end PROC;
    (:P2,P3:):process P;
      (:P2:):var U:signal;
      (:P3:):var V:array 1:8 of char;
     begin ..... end P;
  begin ..... end M;

begin ..... end PM;
```

図3 (a) プロセッサ指定

```
processor module P1;
  const A=1;
  var I:integer;
  var X,Y,Z:bits;
  module M;
    procedure PROC;
    begin ..... end PROC;
  begin ..... end M;
begin ..... end P1;

processor module P2;
  const A=1;
  var C:char;
  var X,Y,Z:bits;
  module M;
    process P;
      var U:signal;
    begin .....end P;
  begin ..... end M;
begin ..... end P2;

processor module P3;
  const A=2;
  var C;char;
  var X,Y,Z:bits;
  module M;
    process P;
      var V:array 1:8 of char;
    begin ..... end P;
  begin ..... end M;
begin ..... end P3;
```

図3 (b) 各プロセッサモジュール

並列文は"parbegin"と"parend"で囲まれた部分であり，プロセッサごとに実行する文を指定する。並列文は入れ子にすることができ，図4(a)に示すような並列文を含む手続きPは各プロセッサにおいて図4(b)に示すような文を実行する。

```
(:P1,P2,P3,P4,P5:):procedure P;
begin
  S0;
  parbegin
    (:P1:): S1;
    (:P2,P3,P4:): begin
                    S2;
                    parbegin
                      (:P2:): S3;
                      (:P3:): S4;
                    parend
                  end;
  parend
end P;
```

図4 (a) 並列文

processor	statements
P1	S0;S1
P2	S0;S2;S3
P3	S0;S2;S4
P4	S0;S2
P5	S0;

図4 (b) 各プロセッサの実行文

2．3　プロセッサ間の通信

プロセッサ間の通信は，すべてCMによって記述される。CMの中のCPがポートを介したバイト単位の標準手続きinput，outputを呼び出し，それらの要求が一致した時に通信が行われる。（図5）

input，outputは互いに通信する相手をプロセッサ名で指定するが，相手プロセッサとそれに接続されているポートのアドレスを結びつけるのがportin，portout宣言である。また，CPおよび一般デバイス用のポートにアクセスするDPは，それぞれ個有の割込みレベルが指定され，レベル優先割込みが行われる。

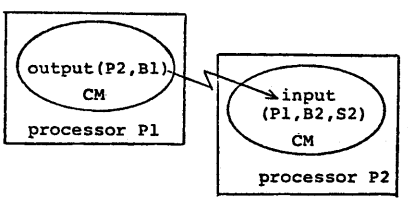

図5．プロセッサ間のメッセージ通信

3、MPLによるパケット通信システムの記述

MPLのプログラム例として，図6に示すような4つのプロセッサによる単純なループ結合におけるパケット通信システムを記述する。各論理プロセッサP1，P2，P3，P4は，物理プロセッサ#1，#2，#3，#4に割り当てられる。各プロセッサモジュールの構成を図7に，システム全体を記述したプログラムリストを図8に示す。

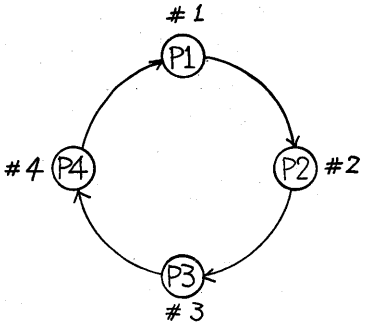

図6、単純なループ結合

各プロセッサ上には，パケットを生産し，宛名をつけて隣のプロセッサに送り出すプロセスPRODUCER(P)と，自分宛のパケットを消費し，その他はさらに隣のプロセッサへ送るプロセスCONSUMER(C)が存在する。また，隣のプロセッサからポートを介して1バイトずつデータを入力する通信プロセスとしてINPORT，隣のプロセッサへ1バイトずつデータを出力する通信プロセスとしてOUTPORTがある。入出力パケットのバッファとしてIB，OBを用意する。これらのバッファはCONSUMERとINPORTおよびPRODUCERとOUTPORTなどのプロセスによって共有されるので，インタフェースモジュール(IM)によって管理される。手続きPUTは，IBに1バイトずつつめ，GETPCKTはIBがいっぱいになったらそれをパケットとして取り出す手続きである。GETは，OBから1バイトずつ取り出す手続きで，PUTPCKTはOBにパケットをつめる手続きである。INITIB,INITOBはそれぞれIB,OBを初期化する手続きである。ここで，IB,OBは仮引数であり，それらの実体はIMの外から与えられる。以上の各手続きはIMの先頭においてdefine宣言されており，これらはIMの外から呼び出される。このIMはバッファの型とバッファに対する一連の手続きをもった抽象データ型である。また，IMは通信モジュール(CM)によって囲まれており，手続きINPCKTはIMの手続きGETPCKTを呼び出して，パケットを取り出し，手続きOUTPCKTはIMの手続きPUTPCKTを呼び出して，パケットを送出する。これらの手続きは，CMの先頭でdefine宣言されており，CMの外から呼び出される。通信プロセスINPORT,OUTPORTは，標準手続きinput,outputを呼び出して隣のプロセッサと通信を行う。通信する相手のプロセッサ名とそれに接続されているポー

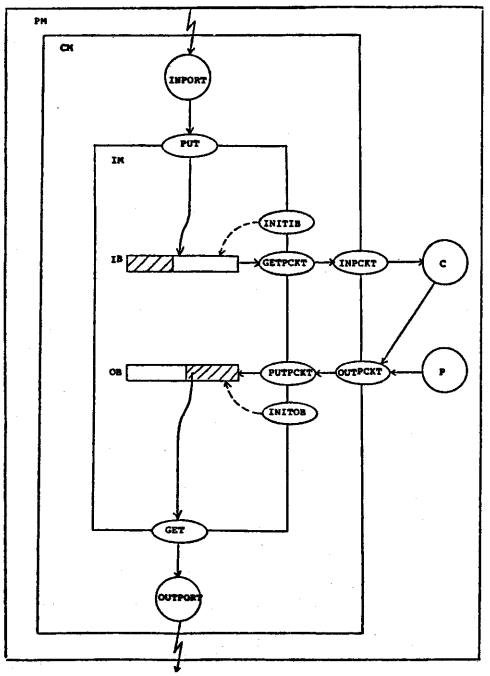

図7、プロセッサモジュールの構成

```
PROGRAM PACKET_COMM(P1,P2,P3,P4);
      CONST PCKTLNG=128;
(:P1,P2,P3,P4:):PROCESSOR MODULE PM;
   COMMUNICATION MODULE PORT_DRIVER;
      DEFINE INPCKT,OUTPCKT;
      USE PCKTLNG;
      (:P1:):PORTIN P4=0; (:P1:):PORTOUT P2=0;
      (:P2:):PORTIN P1=0; (:P2:):PORTOUT P3=0;
      (:P3:):PORTIN P2=0; (:P3:):PORTOUT P4=0;
      (:P4:):PORTIN P3=0; (:P4:):PORTOUT P1=0;

         INTERFACE MODULE PACKET_BUFFER;
            DEFINE PCKTBUF,PUT,GET,PUTPCKT,GETPCKT,INITIB,INITOB;
            USE PCKTLNG;
            TYPE
              PACKET=ARRAY 1:PCKTLNG OF CHAR;
              PCKTBUF=RECORD
                        BUF:PACKET;
                        PTR:INTEGER;
                        FULL,EMPTY:SIGNAL
                      END;
            PROCEDURE PUT(DATA:CHAR; VAR IB:PCKTBUF);
            BEGIN
              WITH IB DO
                IF PTR=PCKTLNG+1 THEN
                  SEND(FULL);
                  WAIT(EMPTY)
                FI;
                BUF(.PTR.):=DATA;
                PTR:=PTR+1;
              HTIW
            END PUT;

            PROCEDURE GETPCKT(VAR P:PACKET; VAR IB:PCKTBUF);
            BEGIN
              WITH IB DO
                IF PTR<PCKTLNG+1 THEN WAIT(FULL) FI;
                P:=BUF;
                PTR:=1;
                SEND(EMPTY)
              HTIW
            END GETPCKT;

            PROCEDURE GET(VAR DATA:CHAR; VAR OB:PCKTBUF);
            BEGIN
              WITH OB DO
                IF PTR=PCKTLNG+1 THEN
                  SEND(EMPTY);
                  WAIT(FULL)
                FI;
                DATA:=BUF(.PTR.);
                PTR:=PTR+1
              HTIW
            END GET;

            PROCEDURE PUTPCKT(P:PACKET; VAR OB:PCKTBUF);
            BEGIN
              WITH OB DO
                IF PTR<PCKTBUF+1 THEN WAIT(EMPTY) FI;
                BUF:=P;
                PTR:=1;
                SEND(FUL)
              HTIW
            END PUTPCKT;

            PROCEDURE INITIB(VAR IB:PCKTBUF);
            BEGIN
              WITH IB DO PTR:=1 HTIW;
            END INITIB;

            PROCEDURE INITOB(VAR OB:PCKTBUF);
            BEGIN
              WITH OB DO PTR:=PCKTLNG+1 HTIW;
            END INITOB;
         END PACKET_BUFFER;

         VAR INBUF,OUTBUF:PCKTBUF;

         PROCESS INPORT(.4.);
           VAR TEMP,STATUS:CHAR;
         BEGIN
           LOOP
             PARBEGIN
               (:P1:):INPUT(P4,TEMP,STATUS);
               (:P2:):INPUT(P1,TEMP,STATUS);
               (:P3:):INPUT(P2,TEMP,STATUS);
               (:P4:):INPUT(P3,TEMP,STATUS)
             PAREND
             PUT(TEMP,INBUF)
           POOL
         END INPORT;

         PROCEDURE INPCKT(VAR P:PACKET);
         BEGIN GETPCKT(P,INBUF) END INPCKT;

         PROCESS OUTPORT(.3.);
           VAR TEMP:CHAR;
         BEGIN
           LOOP
             GET(TEMP,OUTBUF);
             PARBEGIN
               (:P1:):OUTPUT(P2,TEMP);
               (:P2:):OUTPUT(P3,TEMP);
               (:P3:):OUTPUT(P4,TEMP);
               (:P4:):OUTPUT(P1,TEMP)
             PAREND
           POOL
         END OUTPORT;

         PROCEDURE OUTPCKT(P:PACKET);
         BEGIN PUTPCKT(P,OUTBUF) END OUTPCKT;

      BEGIN
        INITIB(INBUF); INITOB(OUTBUF);
        INPORT; OUTPORT
      END PORT_DRIVER;

         PROCESS CONSUMER;
           (:P1:):CONST SELF='P1      ';
           (:P2:):CONST SELF='P2      ';
           (:P3:):CONST SELF='P3      ';
           (:P4:):CONST SELF='P4      ';
           VAR P:PACKET; I:INTEGER;
               ADDR:ARRAY 1:8 OF CHAR;
         BEGIN
           LOOP
             INPCKT(P);
             I:=1;
             WHILE I<=8 DO
               ADDR(.I.):=P(.I.);
               I:=I+1
             ELIHW;
             IF ADDR=SELF THEN
               (* CONSUME PACKET DATA *)
             ELSE OUTPCKT(P) FI
           POOL
         END CONSUMER;

         PROCESS PRODUCER;
           VAR P:PACKET;
         BEGIN
           LOOP
             (* PRODUCE PACKET DATA *)
             OUTPCKT(P)
           POOL
         END PRODUCER;

      BEGIN
        CONSUMER;
        PRODUCER
      END PM;

      BEGIN
        ALLOC(P1,1);
        ALLOC(P2,2);
        ALLOC(P3,3);
        ALLOC(P4,4)
      END PACKET_COMM.
```

図8. パケット通信システム
MPL ソースプログラム

トのアドレスはportin, portout宣言によって対応づけられている。これらはプロセッサごとに異なるアドレスなので，プロセッサ指定されている。また，通信の相手プロセッサ名もプロセッサごとに異なるので，*input*, *output*は並列文の中で呼び出される。IB,OBの実体INBUF,OUTBUFはCMにおいて変数宣言されている。CMはプロセッサモジュールPMに囲まれている。プロセスCONSUMERはパケットが自分宛ならば消費し，そうでなければ再び隣のプロセッサへ送出する。各プロセッサは自分の名前をSELFという定数として，プロセッサ指定つきの定数宣言をしている。プロセスPRODUCERは次々とパケットを作り出し，隣のプロセッサ

へ送出する。4つのプロセッサはこのように唯一のPMによってまとめて記述されるので効率良く記述できる。さらに，PMは *global objects* と *Program* の *body* に囲まれている。*Program* の *body* においては，*alloc* 文によって，各論理プロセッサが物理プロセッサに割り当てられる。

4. MPLの実装

4.1 実装の概要

MPLの実装の概略を図9に示す。MPLソースプログラムはプリプロセッサによって各プロセッサに対して1つずつのModulaソースプログラムに変換される。変換されたModulaプログラムはModulaコンパイラによってM-Codeという仮想スタック・マシンコードに落とされる。このM-Codeは，次に示されるような変換が行われて，マイクロコンピュータ上で直接インタプリトされるM'-Codeに変換される。

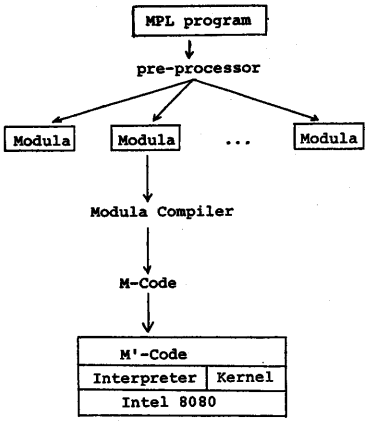

図9. MPLの実装

i) M-Codeの命令コードおよびオペランド部はワード(2バイト)単位であるため，これらをバイト単位に圧縮する。
ii) M-Code内に散在する定数を定数テーブルに集めそのアドレスをセットする。
iii) Procedure, Process呼び出しおよびJump命令等による参照アドレスをセットする。
iv) その他インタプリト時に必要な情報を付け加える。
v) マイクロコンピュータで直接ロード可能なフォーマットに変換する。

変換されたM'-Codeはマイクロコンピュータ上にロードされ，仮想計算機MPLマシンによって実行される。

4.2 MPLマシン

MPLマシンはインタプリタと核(カーネル)から成っており，今回は *Intel 8080* 上に作成した。MPLマシンのメモリマップを図10に示す。スタック領域は，プロセスが生成されるたびに，そのためのプロセス領域が積まれる。プロセス領域は，プロセス・ディスクリプタ(PD)，Static Stack (SS)，Display Vector (DV)，Dynamic Stack (DS)，With Stack (WS)から成る。各プロセスの変数は，SS上に取られDVを用いてアクセスされる。図11に示すように，手続きが呼ばれるたびにその手続きにローカルな変数とその手続きへのパラメータがActivation Area (AA)としてSSに積まれる。DVは手続きの動的な深さ(*lexical depth*)に対応し

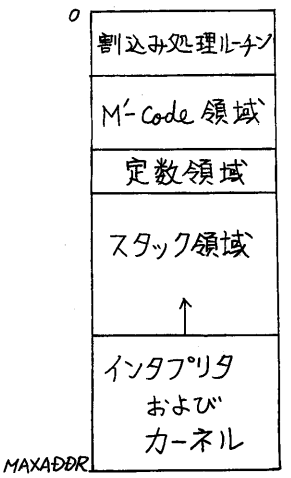

図10. MPLマシン

て，それぞれのAAのbaseアドレスを指している。M'-Codeにおける変数のアドレスは，このlexical depthとdisplacementを用いて表わされる。Dynamic Stackは2バイトのオペランドが積まれ，演算およびパラメータの引き渡し等に用いられるスタックである。With Stackは，レコード変数のフィールドのアクセスに用いられる。各フィールドにアクセスするには，はじめにそのフィールドの先頭アドレスをWith Stackに積み，そこからのdisplacementでアクセスされる。

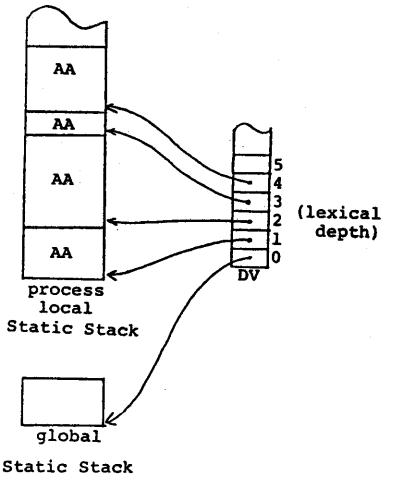

図11．Static Stack と Display Vector

また，MPLでは，複数のプロセスが並行に動作するが，その管理は，図12に示す各プロセスごとに存在するプロセス・ディスクリプタ（PD）を用いて，核が行う。核のもっている機能は，

i)プロセスの生成
ii)信号（signal）に対するsend，wait命令によるプロセスの切り換え
iii)デバイス・プロセス中のdoio手続きによる割込み待ち
iv)プロセスのスケジューリング

であり，プロセス管理に用いられるPDについて以下に述べる。

i)PNX（Process Chain）

一般プロセスは，図13に示すようなCyclic Process Chainにつながれている。Current Process Pointer（CP）は，現在実行中のプロセスのPDをさし，プロセスが新たに生成されると，CPのさすPDの次に新たに生成されたプロセスのPDがそう入される。プロセスが停止またはwait命令によって信号を待つときは，Process Chainをたどって次に実行可能なプロセスを見つける。

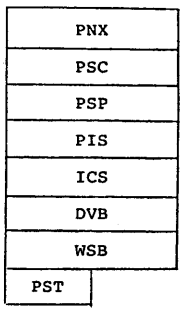

図12．PD

ii)PSC（Process Signal Chain）

プロセスがwait命令を実行すると，図14に示すようにそのPDがwaiting rankに従って，Signal Chainにそう入される。Signal Chainにつながれたプロセスは，他のプロ

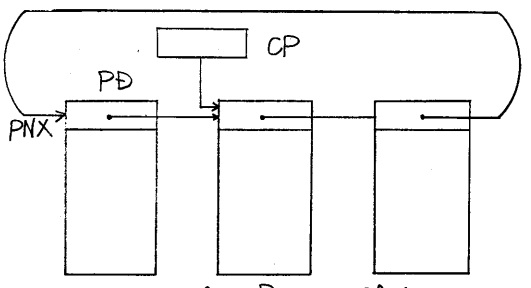

図13．Cyclic Process Chain

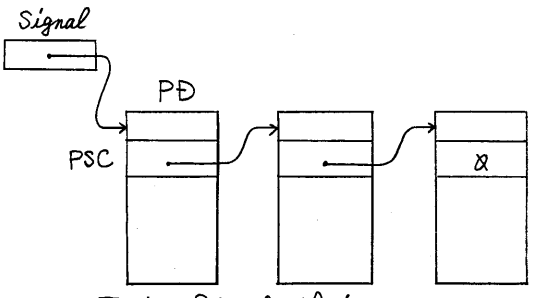

図14．Signal Chain

セスからのsend命令によって，Signal Chainの先頭につながれているプロセスがSignal Chainからはずされ，実行が再開される。

iii) ＰＳＰ（Process Stack Pointer）

プロセスが制御を取られた時に，スタックポインタ（ＳＰ）がsaveされる領域で，そのプロセスが実行を再開する時に，ＰＳＰの値がＳＰにreloadされる。

iv) ＰＩＳ（Interrupted Process Stack Pointer）

デバイス割込みによって割込まれたプロセスのＳＰがそのデバイスに対応するＤＰヌはＣＰのＰＩＳ領域にsaveされる。ＤＰヌはＣＰがdoio命令を実行して，割込み待ちになる時に，ＰＩＳの値がＳＰにreloadされ，割込まれたプロセスが再開される。

v) ＩＣＳ（Instruction Counter Save）

プロセスが制御を取られた時にInstruction Counter（ＩＣ）の値がsaveされる領域。

vi) ＤＶＢ（Display Vector Base）

プロセスのDisplay Vectorのbaseアドレスが格納されている領域。

vii) ＷＳＢ（With Stack Base）

プロセスのWith Stackのbaseアドレスが格納されている領域。

viii) ＰＳＴ（Process Status）

プロセスの状態（active，dead，runable，wait）を示す領域。

5．あとがき

マイクロ・コンピュータ・ネットワークのためのシステム記述言語ＭＰＬの基本仕様とその実装方法について述べた。現在，一応マイクロコンピュータ上へのインプリメントを行い，インタプリタおよびカーネルの動作テストを行っている段階であり，実際のネットワーク上で前述のパケット通信システムなどを動作させるまでには至っていない。従って，今後の課題としては，実際のネットワーク上でＭＰＬによって記述されたシステムを実際に動かして，ＭＰＬの評価・検討を行うことである。

また，実行効率およびメモリサイズについて考えると，「仮想コード＋インタプリタ」方式ではなく，直接マイクロ・コンピュータの機械語を生成するＭＰＬコンパイラへの変更も今後の課題の一つである。

参考文献

(1) N.Wirth, "Modula: A Language for modular multiprogramming" SOFTWARE-PRACTICE AND EXPERIENCE, Vol.7 No.1 pp.3-35(1977)

(2) I.C.Wand and J.Holden,"MCODE:A Description of the bootstrapping interface of the Univ. of York Modula Compiler", York Computer Science Report No.14 (1978)

(3) 山口 剛，"マイクロ・コンピュータ・ネットワークとそのシステム記述言語" 東大修論（1981）