マルチマイクロプロセッサ向き リアルタイム処理言語 RCC について

天満 尚二（広島大学 工学部）
阿江 忠（広島大学 工学部）

1. まえがき

マイクロプロセッサの進歩によって、専用目的ではあるが、処理能力の高い分散システムの実現が可能となってきた。このような専用目的のための分散システム（例えば、コントローラー）の開発において、システムプログラマは、入出力などのハードウェアを念頭においた設計が必要となる。これは、分散システムにおけるオペレーティングシステム（OS）の構築方法が完成していないことによる。したがって、システムプログラマは、システムを作成する際、各システムごとに、本来OSの提供する機能まで考慮し、設計を行なわなければならない。

このような分散システム上のソフトウェアを見ると、現状では、アセンブラを使用している場合が多く、次いで、PL/Mなど高級言語が使われている。しかし、これらの言語は、本来、分散システムの記述を目的とした言語ではなく、結局、各プロセッサ単位の記述になってしまう。

分散システムのための開発ツールを持つ開発システムとして期待されるのは、Adaを備えた開発システムであろう。しかし、Adaは、言語自体の処理系が大きく、小規模な分散システムの開発には問題があるように思われる。逆に、処理系のコンパクトさから見れば、マルチマイクロプロセッサのレベルには、C言語、FORTHなどが、そのベースとして有望であろう。

この観点から、我々はC言語を基本にした分散処理言語RCCの設計を行なった。RCC（Real-time C for Controller）は、プロセッサ間構造と密接に関係している言語で、各プロセス間の通信には、共有変数を持つメッセージ方式を用いている。

以下、2節ではコントローラーの開発言語について簡単に考察し、3節では、RCCの概要を述べる。

2. コントローラー開発言語

一般に、コントローラーは、図2.1のように表わすことができる。オブジェクト（object）は、自分の状態を示す状態データ（state data）をコントローラーに送出する。コントローラーは、この状態データに基づき、データに対応する処理を行ない、制御を行なうためのコマンド（command）をオブジェクトに対し送出する。これらの動作は、通常、決められた時間内に行なわなければならない。図2.1におけるコントローラーは、ハードウェアとソフトウェアの両方を含めた概念図である。このコントローラーのソフトウェアは、リアルタイム性、並列性を持つものであり、ソフトウェアを記述するプログラミング言語は、この両者の性質を備えていなければならない。

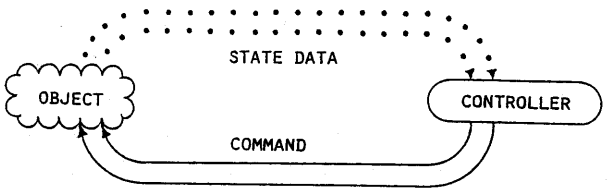


図2.1 コントローラーの概念図

このようなコントローラー開発用言語は、以下の性格を持つ必要があろう。

（1） 同時に、かつ/または、非同期に発生する事象を処理しなければならない。

(2)　物理的に分散したプロセスを実行させるため、本来、OSのサポートすべき機能をプログラミング言語自身が持つ必要がある。

(3)　プロセスの存在するプロセッサの指定ができること。

(4)　ハードウェアに依存する部分の記述が可能なこと。インライン機能や手続きの形式でのみ、アセンブラや機械語の使用が可能である方がよい。

次に、各プロセッサ上で並列に動作する各プロセス間の通信方法に関しては、メッセージパッシングと共有変数方式に大別できる。表2.1は、従来の並列処理言語の通信方法を示したものである [1]-[9]。メッセージパッシングと共有変数方式は、直接的には、通信回線と共有メモリという物理的媒体に対応するが、プログラミング言語上から見れば、通信媒体よりむしろ、通信に対する制御の流れの問題とみることができる [11]。

Concurrent Language	Message	Common Variable
Concurrent Pascal	X	O
Path Pascal	X	O
Modula	X	O
ILIAD	X	O
Concurrent C	O	O
RCC	O	O
Ada	O	X
PLITS	O	X
CSP	O	X
DP	O	X

プログラミング言語と通信方式の関係
表2.1

Rccでは、プロセス間通信の方法は、主に、メッセージによって行なっている。しかし、完全なメッセージパッシングではなく、共有変数を持つメッセージである。メッセージの利点は、

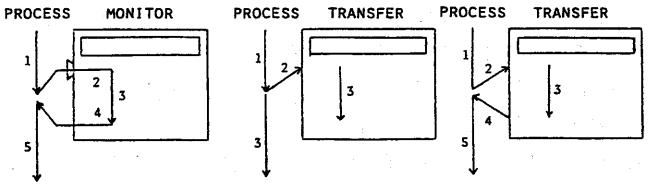
(1)　メッセージ上のデータは、局所的であり、したがって、サイド・エフェクトを被けない。

(2)　メッセージによる通信では、特殊なハードウェアを必要としない。

(3)　自由度の高い通信が可能である。

しかし、言語におけるプリミティブや同期問題は、複雑になる。RCCにおいても、この問題は同様に残された問題である。

一方、共有変数方式はどうであろうか。共有変数方式は、Concurrent Pascal, Path Pascal などに代表されるように良く検討されている。現状の共有変数へのアクセス方法は、プロセジャコールの形式を持ち、メッセージのような自由度の高い通信とはいえない。しかし、物理的に共有できる記憶媒体を持つ場合、通信時間はメッセージより短かくてすむ。したがって、我々は、これらの長短を考慮した結果、本稿では、新しく共有変数を使ったメッセージ方式を提案する。これはConcurrent Pascalにおけるモニタ内プロセジャを1つのプロセスとするようなものである（図2.2参照）。(なお、RCCにおいては、他のすべての機能単位もすべてプロセスと呼ぶ。)しかし、モニタはデータタイプであり、



(a) Access to Common Variable in Concurrent Pascal　　(b) Access to Common Variable in RCC

図2.2　通信概念の違い

RCCにおいてはあくまでプロセスである点が，本質的に異なる。

以上のことを考慮し，RCCの設計は次の方針で行なった。

(1) プロセス構造，プロセッサ構成を明確に記述する。

(2) プロセス間通信は，共有変数を持つメッセージ方式を使用する。しかし，プロセッサ間の通信媒体が通信回線の場合もあるため，純粋のメッセージもライブラリ関数の形式で使用可能であること。

(3) RCCは，C言語の上位互換性を持つものとする。これは，システム記述言語として著名であるC言語を母体とすることで，RCCも，おのずとシステム記述言語の性格を持つことができるからである。

3. RCCの設計

RCCは，前述したように，C言語の拡張である。したがって，基本単位（C言語における関数）は，C言語のシンタックスと同じである（詳しくは，文献10）。RCCのシンタックスは，BNFの拡張で記述されている。" と " で囲まれている単語は，予約語である。{X} はXが無くてもよいことを示し，$\{X\}^+$ はXが1回以上くりかえされることを，また，$\{X\}^*$ はXを0回以上くりかえすことを示す。

3.1 RCCの概観

図3.1は，RCCのプロセス構造の一例を示したものである。RCCのプログラムは，2つの部分で構成されている。1つは，performing，control，transfer と呼ぶプロセス（process）の定義部分であり，もう1つは，これらのプロセスのリンクとプロセッサの割当てを定義する部分である。

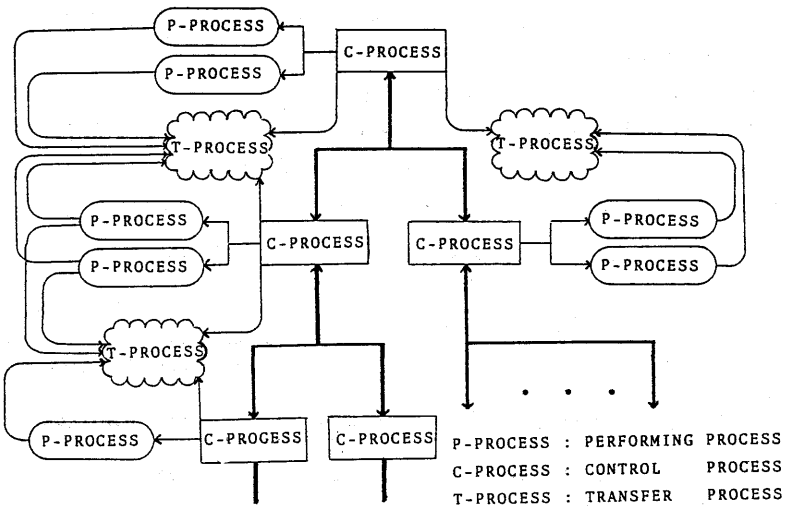


図 3.1 RCCのプロセス構造の例

- 構造定義部分
- プロセス定義部分

構造定義部分は，以下を定義する部分である。

(1) control processのプロセッサへの割当て。

(2) control processと直結するperforming processの定義。

(3) 直結するcontrol process／performing process間の通信を行なうtransfer processの定義と，このプロセッサに直結するプロセッサの定義。

この構造定義部分において，プロセッサ上に存在するプロセス数の上限は，静的に決定される。

プロセス定義部分は，次の3つのプロセスの定義を行なう。

control process

control processは、このcontrol processに直結するperforming processとcontrol processの起動、制御を行なう。

performing process

実際に処理を行なうプロセスである。

transfer process

performing process間、control process間、performing processとcontrol process間の通信を行なうためのプロセス。ただし、これらのプロセスは、構造定義部分において直結しているものに限る。transfer processは、前述の共有変数を持つメッセージ方式の通信を実際に行なうプロセスである。

3.2 プロセス定義部分

3.2.1 Control Process

control processのシンタックスを以下に示す。

```
<control_process>    ::= "#control" "{" <c_process> "}"
```

ここで〈c_process〉はC言語における1つのプログラムである。ただし、〈c_process〉は、次に示すactivate statementを含むことを許される。

activate statementは、以下に示す環境を持つ。

(1) control processにのみ含まれる。

(2) activate statementは、control processに直結するperforming processと他のcontrol processに対してのみ有効である。

activate statementのシンタックスを以下に示す。

```
<activate_statement> ::= "#active" <act_statement>

<act_statement>      ::= "(" <numeric_act>    ")" {<exec_times>}
                       | "(" <parallel_act>   ")" {<exec_times>}
                       | "(" <sequential_act> ")" {<exec_times>}

<numeric_act>        ::= <numeric_act_list> {<limit_times>}

<parallel_act>       ::= <parallel_act_list>

<sequential_act>     ::= <sequential_act_list>

<numeric_act_list>   ::= <act_proc_name> {">" <act_proc_name>}*
                       | <act_statement> {">" <act_statement>}*

<parallel_act_list>  ::= <act_proc_name> {"," <act_proc_name>}*
                       | <act_statement> {"," <act_statement>}*

<sequential_act_list>::= <act_proc_name> {";" <act_proc_name>}*
                       | <act_statement> {";" <act_statement>}*

<act_proc_name>      ::= <process_name>
                       | <processor_id> "." <process_name>

<exec_times>         ::= "^" <integer>

<limit_times>        ::= "=" <integer>
```

ここで、もし〈exec_times〉、〈limit_times〉が省略されている場合は、無限回を意味するものである。

Numeric Activate Statement

activate statement　　avtive(p1 > p2 > p3 > ... > pm = k) ^ n　　は、以下に示す状態を満足するプロセスPiをすべて起動する。

(1)　$n \geq \#(p1) \geq \#(p2) \geq \#(p3) \geq \ldots \geq \#(pm) \geq 0$

(2)　$k \geq \#(pi) - \#(pi+1) \geq 0$,　　i=1,2,3,...,m-1

ここで、#(Pi) は、Piの実行回数を評価する関数である。そして、#(Pi) の評価は、Piの実行が終了したつど行なわれる。

この numeric activate statementを使用した例として、パイプライン処理を以下に示す。図3.2は、この例で用いられるプロセスの構造を示したものである。このような構造を持つプロセスを次に示す numeric activate statement で起動する。

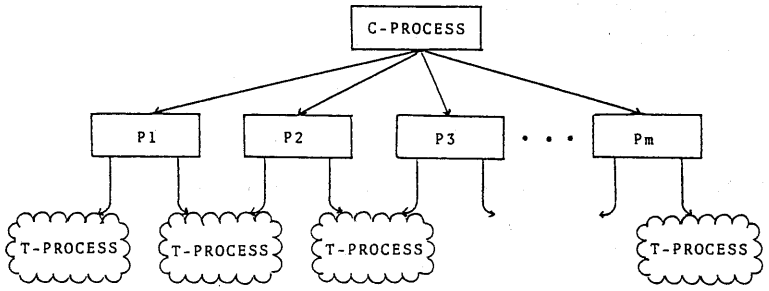


図3.2　　パイプライン処理の例

avtive(p1 > p2 > p3 > ... > pm = 1) ^ n

表3.1に示すようにトレースされる。

この numeric activate statement を使って Producer_Consumer問題を記述したのが Program segment 3.1 である。

```
#control {
   control()
   {
    active(producer>consumer=N)
   }
}
```

Program segment 3.1

(1)\(2)	#(P1)	#(P2)	#(P3)	...	#(Pn)
0	* 0	0	0		0
1	* 1	* 0	0		0
2	* 2	* 1	* 0		0
⋮					
n	* n	* n-1	* n-2		* n-m+1
n+1	n+1	* n	* n-1		* n-m
⋮					
n+m-1	n+1	n+1	n+1		* n+1
n+m	n+1	n+1	n+1		n+1

(1)：実行回数
(2)：各プロセスの実行回数
＊：実行可能状態

表3.1　　activate statementのトレース

Parallel Activate statement

parallel activate statementは、プロセスを単に並列に実行する場合に使用する。

activate statement　　active(p1 , p2 , p3 , ... , pm) ^ n　　は、以下の状態を満足するすべてのプロセスPiを起動する。

$n \geq \#(pi) \geq 0$,　　i=1,2,3,...,m

この場合も、前述の numeric activate statement と同様、#(Pi) の評価は、Piの終了の

つど行なわれる。

Program segment 3.2 は、parallel activate statement を Producer_Consumer 問題に使用した場合のものである。
Program segment 3.2 を使用した場合、transfer process 上でバッファ操作のための処理が必要である（Program segment 3.5 参照）。Program segment 3.1 と 3.2 を比較すると、transfer process 上の処理は、前者は Path Pascal 的であり、後者は Concurrent Pascal 的である。

```
#control {
   control()
   {
    active(producer,consumer)
   }
}
```

Program segment 3.2

Sequential Activate Statement

この activate statement は、プロセスの実行を逐次行なう場合に使用する。例えば、sequential activate statement `active(p1 ; p2 ; p3 ; ... ; pm) ^ n` は、次に示す条件を満足するプロセス Pi を起動する。

(1) $n \geq \#(pi) \geq 0$

(2) pi-1 is end , i=1,2,3, ... ,m+1

したがって、この sequential activate statement は、右の for 文と同じ意味を持つ。ただし、この for 文で、Pi() は、プロセス Pi を起動するものとする。

```
for ( i=1 ; i<=n ; i++ ) {
    P1() ;
    P2() ;
     .
     .
     .
    Pm() ;
}
```

3.2.2 Performing Process

Performing Process は、C 言語の 1 つのプログラムに相当し、実際に、さまざまな処理を行なうプロセスである。performing process のシンタックスを以下に示す。

```
<performing_process> ::= "#perform" "{" <p_process> "}"
```

Producer_Consumer 問題における performing process を Program segment 3.3 に示す。ここで、extern 宣言をしている関数 produce(), consume() は、transfer process である。

```
#perform {
    char  x='A' ;
    producer()
    {
       extern produce()

       produce(x%27+0x61) ; x++ ;
    }
}

#perform {
    consumer()
    {
       extern  consume() ;

       printf("%c",consume()) ;
    }
}
```

program segment 3.3

3.2.3 Transfer Process

Transfer Process は、プロセス間通信（performing process、control process の両者を含む）を行ない、この上の共有変数の管理を行なう。transfer process のシンタックスを以下に示す。

```
<transfer_process>   ::= "#transfer" "{"
                           <common_var_decl>+
                           <t_process>+ "}"
```

ここで、〈common_var_decl〉と〈t_process〉は、C 言語の変数宣言と関数定義の

シンタックスと同じである。

〈common_var_del〉で宣言されている共有変数は、常に、〈t_process〉によってアクセスされる。したがって、共有変数を必要とするプロセスは、〈t_process〉を起動し、間接的にアクセスを行なう。このとき、通信における制御の流れがメッセージに類似しており、自由度が従来の形式、プロセジャコールの形式よりも高い。〈t_process〉は、直接であろうと間接であろうと、再帰呼び出しは禁止されている。さらに、〈t_process〉の実行は、1つの transfer process 内では、常に、1つであり、排他的な実行が保証されている。

RCCでは、前述のように、〈t_process〉の排他的な実行の み保証するだけである。したがって、さらに高度なアクセス方法（例えば、優先順位のついたアクセス、キューを使ったアクセスなど）は、プログラマが、自ら、実現しなければならない。これは、最も単純な機能のみ提供し、コントローラーのおかれているさまざまな環境に対応できるようにしたことによる。

Producer_Consumer問題における transfer process を Program segment 3.4 、3.5 に示す。

```
#transfer {
    int   count=0 , lastpt=0 ;
    char  buffer[N] ;

    produce(x)
      char x ;
    {
      buffer[lastpt]=x ;
      lastpt=(lastpt++)%N ;
      ++count
    }

    consume()
    {
      char x ;

      x=buffer[(lastpt-count)%N] ;
      --count ;
      return x ;
    }

}
```

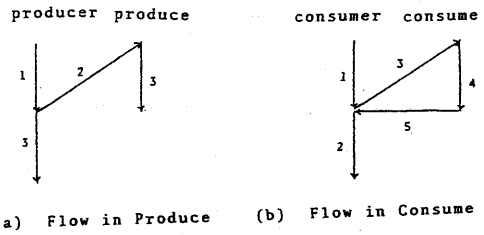
program segment 3.4

```
#transfer {
    int   count=0 , lastpt=0 ;
    char  buffer[N] ;

    produce(x)
      char x ;
    {
      if(count==N) return 0 ;
         else {
            buffer[lastpt]=x ;
            lastpt=(lastpt++)%N ;
            ++count
            return 1 ;
         }
    }

    consume()
    {

      char x ;

      if(count==0) return 0 ;
          else {
             x=buffer[(lastpt-count)%N] ;
             --count ;
             return x ;
          }
    }
}
```

program segment 3.5

Program segment 3.4 は、Program segment 3.1 に対応し、Program segment 3.5 は、Program segment 3.2 に対応する。

transfer process を用いた通信の制御の流れはメッセージと似ている。図3.3は、Program segment 3.4 の transfer process と performing process の実行順序を示したものである。従来の共有変数方式の通信では、常に、図3.3-(b)のようになるが、これは並列性を低下するものであり、RCCでは、producer を並行に動作させることが可能である。これは、アクノレッジを待つ方式と待たない方式のメッセージに対応する。



(a) Flow in Produce (b) Flow in Consume

図3.3 通信の制御の流れ

3.3 構造定義部分

構造定義部分は、control processのプロセッサへの割当て、transfer process と performing process のリンク情報を定める部分である。シンタックスを右に示す。ここで、〈processor_id_list〉が省略されている場合は、transfer processがそのプロセッサ内でのみ使われることを意味する。Producer_Consumer問題を単一プロセッサ内で行なうと、Program segment 3.6のようになる。

```
<structure_part>     ::= "#structure" "{" <structure>+ "}"
<structure>          ::= {<processor_id> "<-" <proc_decl> }+
<procdecl>           ::= <c_proc_decl>+
                         <p_proc_decl>*
                         <t_proc_decl>*
<c_proc_decl>        ::= "control" "-" <proc_name_list> ";"
<p_proc_decl>        ::= "perform" "-" <proc_name_list> ";"
<t_proc_decl>        ::= "connect"
                         { "->" <processor_id_list> }
                         "-" <process_name_list> ";"
<process_name_list>  ::= <process_name> {"," <process_name>}*
<processor_id_list>  ::= <processor_id> {"," <processor_id>}*
```

```
#structure {
  processor1 <- control - control  ;
                perform - producer ,
                          consumer ;
                connect - produce  ,
                          consume  ;
}
```

program segment 3.6

3.4 プログラム構成

RCCのプログラムは、右に示すシンタックスに従って記述される。

```
<program>          ::=<structure_part> <process_cluster>
<process_cluster>  ::= <control_process>+ *  <process_cluster>
                     | <performing_process>*  <process_cluster>
                     | <transfer_process>     <process_cluster>
```

RCCにおける変数のスコープは、ほとんどC言語と同じであるが、C言語のグローバル変数は、RCCでは、プロセス内のローカル変数となる。共有変数は、本質的にはグローバル変数であるがtransfer process 上のローカル変数の形式をとる。しかし、本来グローバル変数であるため、共有変数の値は常に保持されている。

4 おわりに

以上、Producer_Consumer問題を例にRCCの性質を説明した。現在、マルチマイクロプロセッサUNIP[12]上へのインプリメンテーションを行なっている。同時に、昨年、LAN (Local Area Network)のノードプロセッサのソフトウェアをC言語で記述を行なった[13]が、RCCによる書き換えを検討している。

構造定義部分は、将来、特に指定のない場合は、プロセッサへ各プロセスを自動的に割当てを行なうローダーに置き換わる方が望ましいであろう。また、このようなローダーを含めた、分散システムのための開発システムについても、現在検討を行なっている。

参考文献

(1) P.B.Hansen:"The architecture of Concurrent Programs", Prentice-Hall(1977).
(2) R.B.Kolstad et al.,SIGPLAN NOTICES, Vol.15,No.9,pp.15-24 (Sep. 1980).
(3) N.Wirth,Software-Practice and Experience,Vol.7,No.1,pp.3-35, (Jan.-Feb. 1977).
(4) H.A.Schutz,IEEE Trans. Software Engineering,Vol.SE-5,No.3, pp.248-255 (May 1879).
(5) Y.Tujino et al.,IECE of Japan, Techinical Report,EC81-13, (July 1981) in Japanese.
(6) DoD,United States Department of Defence (July 1980).
(7) J.A.Feldman,CACM,Vol.22,No.6, pp.353-368 (June 1979).
(8) C.A.R.Hoare,CACM,Vol.21,No.8, pp.666-677 (Aug. 1878).
(9) P.B.Hansen,CACM,Vol.21,No.11, pp.934-941, (Nov. 1978).
(10) B.W.Kernigham et al.:"The C Programming Language", Prentice-Hall , (1978).
(11) J.A.Stankovic,IEEE Computer, Vol.15,No.4,pp.19-25,(Apr. 1982).
(12) T.Ae et al.,IECE of Japan, Techinical Report,EC81-39, (Oct. 1981) in japanese.
(13) Y.Ozaki et al.,IECE of japan, Techinical Report,EC81-55, pp.63-74,(Dec.-1981)in Japanese.