機器組み込み用並行処理言語 coroutine C の設計とその処理系の実現†

中村八束†† 山崎靖夫††† 不破泰††

多様な実行環境が存在する機器組み込み用ボード・コンピュータにおいて並行処理 (concurrent processing) 機能を実現させるとき，通常ハードウェアに依存するようなリアルタイム・モニタ等や状態遷移法などの技法による記述が必要であるため，簡便に並行処理を行うことは難しい．そのため並行処理が記述しやすく，実行環境に依存せず並行処理を簡便に行えるようなシステム記述並行処理言語が必要である．本論文では，機器組み込み用並行処理言語 coroutine C を提案し，その基本的な処理系の作成と，そこにおける並行処理機能の実現方法を述べる．また，その応用例を紹介する．言語 coroutine C の並行処理方式の特徴は，(1)各タスクをコルーチン (coroutine) 化し，コンパイラが実行時の処理時間を積算しながら自動的かつ妥当な場所にタスク切り替えコードを挿入する．(2)ターゲット CPU の載ったボードであれば実行時に特別なハードウェアは不要である．(3)タスクの切り替え点が実行前に決定されていることやハードウェアに依存せずにタスクを実行できることからシミュレータなどでデバッグすることが容易である．などの点である．

1. まえがき

多様な実行環境が存在する機器組み込み用ボード・コンピュータの処理形態において，処理内容が独立しており各々の処理が並列的に実行されることを要求する場合や，処理内容に応じて処理を並列的に行うことによって処理効率がよくなる場合が多く見られる．このような場合，一つの CPU で処理を行おうとすると，単一な流れの処理では，すべての事象に対して同時にサービスを行うためにリアルタイム・モニタ等を用いたり状態遷移法[1]などの技法を用いなければならず，簡便に並行処理 (concurrent processing) を行うには記述性，保守性に難がある．これらを高級言語上で並行処理を実現する方法で処理することによって，記述性，保守性の向上が見込まれる．

そこで，多様な実行環境に柔軟に対応できるために並行処理のための特別なハードウェアを必要とせずに並行処理を簡便に記述し，実現できるように，システム記述言語Cに並行処理機能を付加することにより，言語 coroutine C を設計した．

タスクが非同期に動き並行処理を行うことのできる言語として，concurrent PASCAL, Ada, Modula-2[2)~6)]等が提案されている．並行処理を行う場合，タスク間の共有資源のような際どい資源で相互排除する必要があるが，concurrent PASCAL においてはモニタ (monitor) による共有資源のモジュール化と排他的な短期間スケジューリングによって相互排除と同期を実現し，Ada においてはランデブー (rendezvous) によって値の受け渡しを行って相互排除と同期を実現している．また Modula-2 においては TRANSFER によるコルーチン[7)] (coroutine) の記述で明示的にタスクの切り替えを行うことによって，相互排除，同期を行っている．

提案する言語 coroutine C においては，相互排除と同期を以下のように行う．

(1) コンパイラがタスクを一つのコルーチンとし，オブジェクト・コード生成時に，そのオブジェクト・コードを実行するのに要する CPU のクロック数 (以降，ステート数と言う) を積算しながら自動的にコルーチン移動コード (以降，タスク切り替えコードと言う) を生成することによって，タスク切り替えを行う．そのためタスクに割り当てる1実行当たりの最大実行ステート数 (以降，最大実行時間と言う) をプログラム上で指定する．

(2) タスクのスケジューリングは短期間スケジューリングで行う．

(3) タスク内関数を使わない代入文は不可分な処理とし，タスク切り替えによってタスク間の共有部分で相互実行が起きないように，コンパイラは構文解析の際に相互排除処理を行う．

この方式によって，coroutine C は実行に際してターゲット CPU の載ったボードであれば特別にタス

† Design and Implementation of a Concurrent Programming Language "coroutine C" for Embedded Computer System by YATSUKA NAKAMURA (Department of Information Engineering, Faculty of Engineering, Shinshu University), YASUO YAMAZAKI (NIPPON・DATA GENERAL Corp.) and YASUSHI FUWA (Department of Information Engineering, Faculty of Engineering, Shinshu University).

†† 信州大学工学部情報工学科
††† 日本・データゼネラル(株)

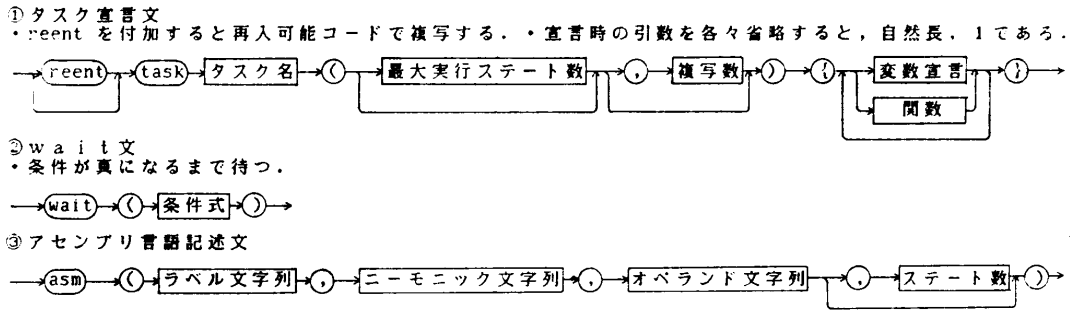

図 1 付加した構文
Fig. 1 Syntax added to language C.

ク管理のハードウェアを要求せず，割り込みは用いない．またタスクの切り替え点が実行前に決まっていることとハードウェアに依存せずに実行することから，シミュレータなどでのデバッグが容易にできる．

このような特徴を持つ言語 coroutine C の処理系を言語C[8]を用いて作成した．ターゲット CPU はザイログ Z80 であり，処理系はコンパイラとタスク管理を行うカーネル，演算処理を行うランタイム・パッケージから成る．

2. 言語 coroutine C の概要

2.1 言語 coroutine C の文法

言語 coroutine C の文法で基本的な構文は言語Cの構文をそのまま流用している．一部検討すべき箇所があるが，言語Cの構文に並行処理機能を加えることによって実現できる．しかし，現在処理系の作成上，言語Cの構文に制限を加えた文法であり変数，関数は宣言する前に参照はできない．付加した構文を**図 1** に，作成上制限した部分を**表 1** に示す．

2.2 プログラム構造およびタスク構造表現

プログラム構造は**図 2** のようになり，タスク宣言文で囲まれたプログラムは通常の言語Cのプログラムとなる．したがってタスクは，その内部の大域変数と関数によって記述される．実行が開始し終了するのは main( ) と定義された関数である．実際には複数のタスクが記述されるため，タスク間の同期を行わないと並行処理の実現は不可能になる．coroutine C においては，タスク記述の外に同期を支援するために二つの構造がある．それらは共有変数，共有関数である．タスクを並行実行される関数と見なすと，言語Cのプログラムの形をしており全体が二重構造になっている．

表 1 コンパイラ作成上の制限事項
Table 1 Restrictions on a compiler.

- 型の制限 char (unsigned 8 bit), int (16 bit).
- 再帰，構造体，ポインタに関する部分の削除，条件式の評価方法（すべて評価する）．
- 標準関数の制限 組み込まれるのは，inp ( ), outp ( ).
- 付加した関数 taskno ( ), copyno ( ).

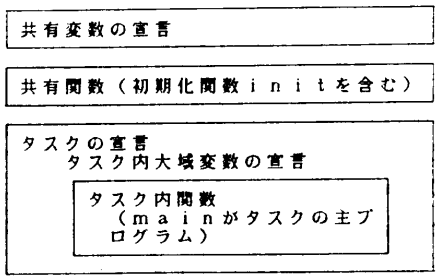

図 2 プログラムの構造
Fig. 2 Structure of programs.

2.3 共有部分と各タスクとの関係

前節のプログラム構造の場合の相互参照関係は**図 3** のようになる．タスク内で宣言された大域変数，関数は，その内部でのみ有効であり，他のタスクから見て全く不可視で参照，実行はできない．そのため共有部分でタスクの同期やデータ等のやりとりを行う．共有

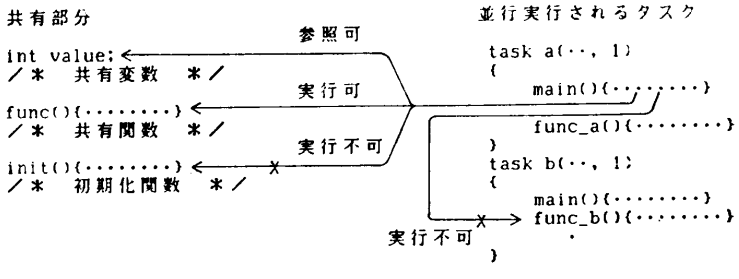

図 3 共有部分とタスクの関係
Fig. 3 Relation between shared resources and task modules.

関数中には，すべてのタスクの実行に先立って1回だけ実行され並行処理実行の環境（共有変数等の初期設定など）を整える初期化関数 init（ ）を記述する．この関数も一つのタスクと見なすことができる．各タスクからは共有変数および共有関数を初期化関数を除いて，すべて排他的に参照，実行ができる．

2.4 並行処理機能の実現

ここで，どのように並行処理機能を組み込むか，また，その記述上の考慮点について述べる．並行処理機能の実現方法は次の四つの項目にまとめられる．

（1） 基本的に，ターゲット CPU が載った組み込み用ボード・コンピュータで実行するプログラム上で，並行処理を実現することを目的としており，並行処理のための割り込み信号発生回路やタイマなど，特別なハードウェアを必要としない．このことは，コンパイラが生成したオブジェクト・コードのみで並行処理機能を実現するようにコード生成することを意味する．

（2） コンパイラによって自動的にコルーチン化されたタスク間を，カーネルを介して移動をすることによって並行実行を行う．Modula-2 などはコルーチンを記述することで並行処理機能を実現するが，coroutine C はコルーチンによるタスク切り替えをプログラム上に記述する方法はとらず，どのようにコルーチン化すればよいかを考慮しなくてよい．

（3） coroutine C において並行処理機能を実現させる場所はコンパイラによってオブジェクト・コードを生成する時であり，コンパイラはオブジェクト・コードに自動的かつ妥当な場所へタスク切り替えコードを挿入する．その際の処理は次章で述べる．

（4） タスク間における同期は共有変数と共有関数で行うため実行に際して共有部分の相互排除がコンパイル時に保障される．

2.5 coroutine C の並行処理形態について

各種提案されている並行処理言語において種々の実現方法が用いられているが coroutine C において，これまで述べた並行処理の形態を用いた理由は次の点からである．

（1） 目的とする分野を組み込み用ボード・コンピュータに絞ったことは，入出力処理に複雑さがなくコルーチン生成上問題が少ないからである．なぜならキーボード入力，ディスク入出力を伴った処理で待ちが生じると，一つのタスクの CPU 占有を防ぐためにカーネルで複雑な同期機構が必要になったり，ハードウェアに依存しなければならないためである．

（2） 通常，並行処理言語はカーネルでのタスクの切り替えを，タイマ割り込み等，外的要因によって行うためタスク切り替え点が実行時に全く不明である．そのため，何らかの同期機構が必要となり，カーネルのタスク管理で，かなりのオーバヘッドが伴う．そこで共有部分には基本的に不可分な操作としてコルーチン化することで複雑な同期機構が不要になる（この方法のために時間のかかる共有関数は書けない）．

（3） コンパイラが自動的にコルーチン化することは明らかに(1)，(2)の要件を満たすためであり，なおかつタスク切り替えがコルーチンの実行の移動による以上，タスク宣言文で宣言された1実行当たりの最大実行時間はコンパイラがオブジェクト・コード生成時に保障しなければならない．実際にタスク切り替えコードが挿入されうる場所は代入文の終り，ループ構成要素の終りまたは初め，wait 文中などである．

（4） タスク切り替え点が実行前に確定していることによって，シミュレータやデバッガなどでのデバッグが容易にできるようにするためである．

3. コンパイル時における並行処理機能の組み込み法

3.1 コンパイラの構造

コンパイラ本体は言語Cで記述され再帰下降型1パス・コンパイラ[9]であり，生成するオブジェクト・コードはアセンブラが受け付ける形式である．そのためアセンブラ時に手作業による最適化が可能である．

3.2 構文解析時における並行処理機能の組み込み

コンパイル作業は1パスで行われるため，構文解析時に並行処理機能を組み込む必要がある．したがって，タスクの1回当たりのCPU占有時間をタスク宣言文で宣言された最大実行時間を越えないように，コンパイラは構文解析とオブジェクト・コードの CPU ステート数の積算を行いながら，タスク切り替えコードをコルーチン化したタスクの的確な場所へ挿入する．やむをえず最大実行時間を越えたときは警告を出す．このことによって並行処理機能が実現する．次に実際の挿入の方法を順を追って述べる．

（1） 代 入 文

タスク内関数を使わない代入文は，本言語仕様において不可分な操作であるため，代入が完了するまでの全ステート数の積算が行われ，タスク切り替えコードの挿入は代入の終了時に考慮される．

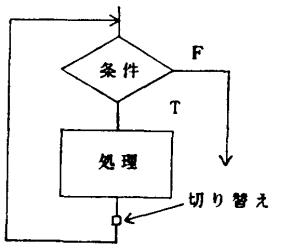

図 4 while 文でのタスク切り替え命令の挿入
Fig. 4 Insertion of task switching code into while statement.

(2) if 文

if 文では制御構造において真，偽の二つの通り方があるためにコンパイラは真の処理，偽の処理について別々にタスク切り替えコードの挿入を考慮する．しかし実行に際して真，偽いずれを通るかは条件式が自明でない限り不明であり，タスクにおける if 文の分岐の仕方によって最大実行時間を越えては困る．したがって次の文を積算する際に if 文の挿入後の残りステート数の多い方を用いる．このことで真，偽による占有時間に差が生じるが，宣言された最大実行時間はCPU 占有時間の最大値であるから不都合はない．

(3) ループを構成する文 (while 文，do 文，for 文，wait 文)

ループを構成する文は，そのループ内の処理が始まる点または終る点でタスク切り替えコードの挿入が必ず行われる．while 文の例を図 4 に示す．したがって実行時にタスク内のループを構成する文は1回実行するごとに他のタスクに制御が移るため，見掛け上，特に指定しない限りループは存在しない．ただし，共有関数中は一切タスク切り替えコードの挿入を行わないためにループ構造ができるので，プログラムを記述する際に実行時間，条件式に注意しなければならない．さもないと最悪の場合一つのタスクが CPU を永久占有し他のタスクが停止する．他のタスクとの同期をとるための wait 文の待ちは，ビジーウェイトで行われるため，条件式で共有部分が使われていない場合にコンパイラは警告する．

(4) 関数の呼び出し，戻り

カーネルではタスク切り替えでプログラム・カウンタとスタック・ポインタの入れ換えを行うので，関数呼び出し，戻りにスタック・フレームを使えば再帰呼び出しも可能であるが，現在は引数の渡しを代入によって，戻り値をレジスタによって行っている．関数の処理については，タスク内関数と共有関数とでは，タスク切り替えコードの挿入を行うか行わないかの違いがある．

(5) break, continue, goto 文

本来ならば break, continue, goto などの直接，制御の流れを変える構文は全経路の解析を行って未到達な文およびループ構造の有無を検出し，制御の流れを確定した後にステート数の積算およびタスク切り替えコードの挿入を行うことが望ましいが，コンパイラが再帰下降型1パス・コンパイラであるために流れを確定しにくく，これらの文については実行前に挿入を行い未到達文の削除は行わない．

3.3 共有変数，共有関数を伴う一連の処理

前節ではコンパイラが並行処理機能を組み込む基本的な方策を述べた．しかし共有部分と wait 文でタスク間の同期を行う場合に同期処理の途中，wait 文以外の所でタスク切り替えコードが挿入されると共有部分で相互実行が起きてしまう．そのため明示的に [*, *] で囲むことによって wait 文を除いてタスク切り替えコードの挿入を抑制できる．例として次のような場合 (sw は共有変数)．

```
[*
  wait (sw==0);
  a=inp (0x00);
  sw=1;
*]
```

また共有部分を操作したタスクを区別できるように，実行時に組み込み関数 taskno( ) によってタスクに付けられたタスク番号を知ることができる．

3.4 式の展開と最適化

式の展開は構文木を作りコードを生成するが，その際に最適化は現在は行っていない．また未到達な文，自明な条件式，条件式の自明点での評価打切などの処理も現在行っていない．例外として return 文について以降の文が実行されない時は以降のコードを生成しない．変数の型の変換は言語Cの仕様に準拠している．

3.5 メモリへの変数エリアの割り付け

すべての変数は静的に宣言された順序に割り付けられ，関数の引数も同様である．また変数値の参照はポインタによるインダイレクトな参照ではなく直接アドレスによる参照である．

3.6 アセンブリ言語のインライン展開

coroutine C は Z80 のアセンブリ言語の記述を受け付ける．この時，変数の参照は [a] のように書く

ことでコンパイラは自動的にaの変数のラベルと置換する．この処理以外は記述されたアセンブリ言語が，そのままオブジェクト・コードに出力され文法エラー等はアセンブラによって検出する．

3.7 タスクの複写と再入可能コード化

入出力ポートやバッファなど同一のプログラム構造になったタスクを複数個必要な場合，コンパイラはタスクごとに一つのプログラムから任意個のオブジェクト・コードを生成できる．また再入可能なコードの生成を指定することでオブジェクト・コードを小さくできる．この場合，変数領域は連続して静的に作られる．どちらの場合においても実行時に組み込み関数 copyno（ ）によって複写された番号を知ることができる．

4. 並行処理の実行

4.1 カーネル処理

カーネルには，すべてのタスクの実行に先立ってプログラムの実行環境を整える実行環境初期化ルーチンとタスクの切り替えを行うタスク切り替えルーチンが組み込まれている．

（1） 実行環境初期化ルーチン

通常カーネルは0番地から常駐して，起動されるとCPU を割り込み禁止モードにし，タスク実行のための情報が書かれたROMエリアのタスク・コントロール・ブロック (TCB) 表 (**表 2**) の初期値をRAMエリアへ複写する．その後，共有関数中に初期化関数 init（ ）があれば制御を移し，終了した時点で並行処理状態になる．

（2） タスク切り替えルーチン

カーネルから制御が移されたタスクは，コンパイラがタスク切り替えコードを挿入した場所の，

```
ld    a, SLEEP (または HALT)
call  @ kernel
```

の命令でカーネルに制御を戻す（**図 5** 参照）．この時，カーネルに渡されるのは次にこのタスクが実行可能状態になった時に実行するか否かの状態値と，実行を再開する時の再開アドレスである．

カーネルはタスクから制御を渡されると，そのタスクの状態値（待ち，終了），再開アドレスおよびスタック・ポインタを TCB に保存し，タスクの起動する順序が書かれたタスク系列表から次に起動すべきタスク番号を取り出して，TCB の状態値が待ちならばそのタスクを，終了ならば系列表の順に実行可能なタスクが見つかるまで捜し，TCB 表から再開アドレス，スタック・ポインタを設定し制御を移す．この時，タスクのオブジェクト・コードが再入可能コードで生成されているプログラムの場合は，データ領域を参照する際のオフセット値も設定される．タスク系列表とタスクの呼び出し順序の関係を**図 6** に示す．このルーチンは他のタスクへの制御の移動を援助するだけであるため，高速かつコンパクトになっている．

4.2 タスクにおける並行動作

各タスクにおける並行動作は，3章で述べたようにコンパイル時に自動的にタスクをコルーチン化するため，タスク宣言文で宣言された最大実行時間を越えないように動作する．

オブジェクト・コードには前節の切り替えコードが挿入されているので，タスクの切り替え場所は容易にわかる．

4.3 タスク間の同期について

coroutine C の並行処理機能の実現は，自動的にコルーチン化するコンパイラとタスク切り替えのみを行う低レベルなカーネルの機能によっている．このことは実行速度の点で大きく寄与しているが，これだけで

表 2 タスク・コントロール・ブロック (TCB) 表
Table 2 A task control block table.

状態値	開始番地	スタック ポインタ	再入コード オフセット	タスク 複写番号
1 byte	2 byte	2 byte	2 byte	1 byte

・表はタスク番号順に作られ，状態値，開始番地，スタック・ポインタは切り替え時に更新される．

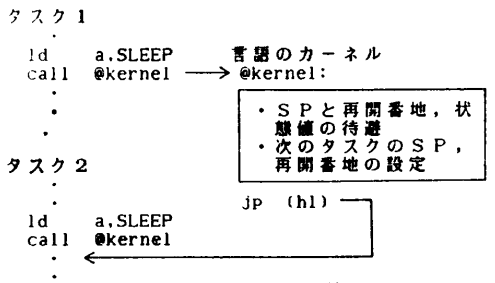

図 5 タスクの切り替え
Fig. 5 Switching of task modules.

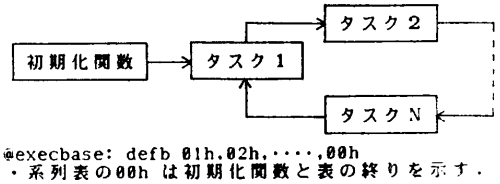

図 6 タスク系列表
Fig. 6 A task sequence table.

は各タスクが全く非同期に実行されることになりタスク間で同期を行う必要がある.

coroutine C では同期を行うために，2章で述べたようにタスク間で共有する変数および関数と，関数内にキュー等を作るためのタスク番号，複写時の複写番号を取り出す組み込み関数が用意してある．これらを用いることによって，さまざまな同期機構を作成することが可能である．作成するときは，各タスクの wait 文で同期が完了するまで繰り返し共有部分を参照し続けるようにし，共有関数はその中で留まらずに成功，不成功を返すように記述しなければならない．これはカーネルが共有部分の操作に関与していないことによる.

カーネルに同期機構を組み込まない利点は，過大な同期機構によってカーネルの CPU 占有時間が大きくなることもなく，プログラマが必要な同期機構を作ることができることである．ただし同期処理を各タスクが独自に行うため，プログラム記述する際に実行時の挙動に注意が必要である欠点がある.

4.4 タスクのスケジューリング

coroutine C においてタスクの数は任意個記述できるが，カーネルで256個に制限している．タスクはプログラムの起動と共にすべて起動され，各タスクは待ちと実行を繰り返しながらタスク内の main( ) の終りに達するとそのタスクは終了する．これはカーネルが TCB 表の状態値によってタスクに制御を渡すか渡さないかを決めているためである．タスクの優先度は，そのタスクに割り当てた最大実行時間により実行中に動的に変更されることはない．起動順序はプログラムに記述されたタスクの順序である．実行時にタスクの優先度，実行順序を変えたい場合はタスク系列表を実行中に再配置する必要があるであろう.

5. coroutine C の性能評価

coroutine C で作られたプログラムにおいて，並行処理実行状態になった時のカーネルの最早切り替え時間（タスクの切り替えを始める時から次のタスクの実行を開始するまでの時間）は**表3**のようになる．現在のコンパイラは1，2章で述べた概念を実現できるレベルにしか達していない.

次にオブジェクト・コードの動作速度について以下の応用例をもって評価した.

以前我々が開発した，ネットワークを構築するための簡易多チャネル通信制御装置(MCC : Multi-channel Communication Controller)[10] へ組み込まれたボード・コンピュータに，coroutine C を用いて制御プログラムを作成し評価した．**図7**に簡易多チャネル通信制御装置の概念図を示す．周辺7チャネルからのデータをパケット化してホスト・チャネルに出力し，またホスト・チャネルからのパケット化されたデータは該当周辺チャネルに振り分けて出力する．内部には各チャネルごとに入出力バッファが設けてある．制御プログラムは周辺の入力および出力，ホストの入力および出力を各々タスクで記述してあり，それぞれ並行動作する.

この制御プログラムを coroutine C で記述して

表3 タスク切り替え時間（Z80A 4MHz）
Table 3 Switching time of task modules (Z80A 4MHz).

実行タスク数		最早切り替え時間
非再入コード生成時	1～ 32	93.75 μs
	33～256	99.75 μs
再入コード生成時	1～ 32	105.25 μs
	33～256	111.25 μs

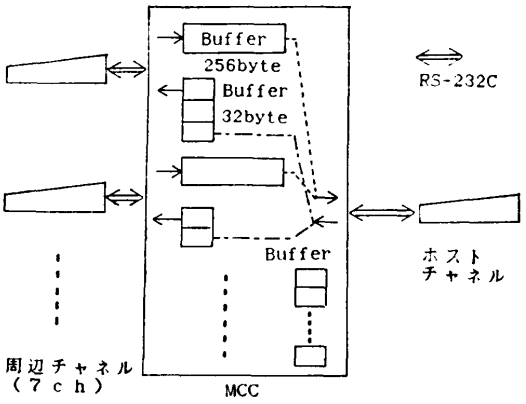

図7 簡易多チャネル通信制御装置 (MCC)
Fig. 7 Multi-channel communication controller (MCC).

表4 簡易多チャネル通信制御装置における比較（Z80A 4MHz）
Table 4 Comparison between coroutine C and assembly language on MCC (Z80A 4MHz).

制御プログラム作成方法	通信速度 (baud)		生成コード (kbyte)
	ホスト	周 辺	
非再入コード	2567	367	12.4
再入コード	2247	321	5.4
アセンブリ言語	9600	1200	6.3

・coroutine C は系列表をホスト・チャネル7回，周辺チャネル各1回の割合にした場合.

表 5 簡易多チャネル通信制御装置におけるタスクのステート数
Table 5 Execution steps of task modules on MCC.

タスク名	タスク切り替えに至るステート数
inputp	420, 2385
outputp	110, 186, 242, 328, 407, 580, 1873, 1881, 1979 2091, 2238, 3058, 3304, 3446
inputh	10, 61, 80, 90, 537, 725, 869, 892, 1580, 1954 2795, 2805, 3465, 3656
outputh	574, 753, 996, 1447, 1454, 1521, 1662, 1938 1998
timep	989
pcounter	208, 300

ROM 化し組み込んだ時と，アセンブリ言語で記述した時の各チャネルの通信速度は表 4 のようになった．また実際にプログラムを実行して，各チャネル入出力のタスクの並行実行とタスク間の同期がうまく行われていることを確認した．

アセンブリ言語での記述は状態遷移法[1]という並行処理プログラムを記述する技法で書かれたものである．coroutine C はアセンブリ言語で記述した場合に比べ速度比で 1/4 であり，機械語レベルでのコードの大きさでは 2 倍程度，周辺チャネルの入出力のタスクを再入可能コードにした場合で若干小さい程度であった．

また，コンパイルした結果のアセンブリ言語のプログラムから，各タスクごとのタスク切り替えに至るステート数の一覧を表 5 に示す．タスクはそれぞれの最大実行時間を 3,000 ステートと指定した．各タスクは，3 章で述べたように様々な条件により，10 ステートのものから，3,656 ステートのものまでに切り替えコードを挿入されて切断されている．指定した 3,000 ステートを越えるものが一部あるが，これは代入文中のタスク切り替えを行わない部分で越えたものである．MCC の周辺チャネル 300 baud，ホストチャネル 2,400 baud，周辺 7 チャネル中 3 チャネルで入出力を行い，ホストチャネルも入出力中という，MCC にとり平均的な使用状況においてこのプログラムを実行した場合，CPU の処理におけるタスク切り替えの占める割合（オーバヘッド）は約 22% であった．

図 8 に coroutine C で記述した MCC の制御プログラムの一部を，図 9 にコンパイルした結果の一部を示す．

```
task inputp(3000, 7)
{
    char    taskn, ipdata;

    main() {
        taskn = copyno();
        loop () {
            while (!(inp(ioc[taskn]) & 0x22) || (bufpc[taskn] == 0xff))
                tbufcomp(taskn, bufpc[taskn]);
            if (inp(ioc[taskn]) & 0x20) {
                bufpc[taskn] = 0xff;
                tbufcomp(taskn, 0xff);
            }
            else {
                ipdata = inp(iod[taskn]);
                inbufi(taskn, ipdata);
                if (ipdata == endcode[taskn]) {
                    bufpc[taskn] = pccheck1(bufpc[taskn] + 1);
                    bufcr[taskn] = bufcr[taskn] + 1;
                    tbufcomp(taskn, bufpc[taskn]);
                }
            }
        }
    }
}
```

図 8 MCC 制御プログラム（一部）
Fig. 8 A part of source program for MCC written in coroutine C.

記述性，保守性を考えると言語仕様をフルセットの言語 C なみにし，オブジェクト・コードの生成を最適化することで簡便に並行処理プログラムを記述でき，十分実用になると考えられる．

6. むすび

本論文では，組み込みシステム用記述言語 coroutine C を提案し，その並行処理機能の実現方法とその方式を検証するための基本的な処理系（コンパイラ，実行時カーネル）の作成について述べた．

組み込まれる環境に依存せずに並行処理を簡便に行うこと，および並行実行時のタスクの切り替え点の不確定さからの解放を重点に設計したが，初期の目標にほぼ達したと考えられる．実際に通信コントローラのソフト設計に適用し，アセンブリ言語で記述したものと比べて，ほぼ満足のゆく性能を得た．

今後さらに各種モデルにおける評価

```
        mnemonic code           source program
j00500::                    ; loop ()
        ld      a,(v001E0)  ;  {
        ld      c,a         ;     wait (inp(ioch) & 0x22);
        ld      b,00h
        in      a,(c)
        ld      e,34
        and     e
        or      a
        jp      nz,j00510
        ld      a,SLEEP
        rst     08h
        jp      j00500
j00510::
        ld      a,(v00200)  ;     idata = inp(iodh);
        ld      c,a
        ld      b,00h
        in      a,(c)
        ld      e,a
        ld      hl,v00320
        ld      (hl),e
        call    m000B0      ;     inbufo();
        ld      a,SLEEP
        rst     08h
        ld      a,(v00320)  ;     if (idata == endcodeh)
        ld      de,(v00170)
        sub     e
        ld      a,00h
        jr      nz,$+03h
        inc     a
        or      a           ;        break;
        jp      z,j00520
        jp      j00530
        ld      a,SLEEP
        rst     08h
j00520::
        ld      a,SLEEP     ;  }
        rst     08h
```

図 9 MCC 制御プログラムのコンパイル結果（一部）
Fig. 9 A part of object program for MCC written in coroutine C.

と，タスク切り替えコード挿入場所や方法の検討，コンパイラをフルセットの言語Cなみの言語仕様にすること，オブジェクト・コードの最適化を行うことが必要である．

参 考 文 献

1) Nakamura, Y. and Fuwa, Y.: A Simple Programming Method of State Transition Diagrams for Parallel Processings, *J. Inf. Process.*, Vol. 5, No. 3, pp. 148-154 (1982).
2) Stotts, P. D.: A Comparative Survey of Concurrent Programming Languages, *SIGPLAN Notices*, Vol. 17, No. 10, pp. 50-61 (1982).
3) Andrews, G. R. and Schneider, F. B.: Concepts and Notations for Concurrent Programming, *ACM Comput. Surv.*, Vol. 15, No. 1, pp. 3-43 (1983).
4) 土居範久：プロセスと同期，*bit*, Vol. 12 No. 8 (1980)-Vol. 14, No. 8 (1982).
5) Filman, R. E. and Friedman, D. P.: *Coordinated Computing*, McGraw-Hill, New York (1984).
6) Brinch, H. P.: *The Architecture of Concurrent Programs*, Prentice-Hall, Englewood (1977).（田中英彦(訳)：並行動作プログラムの構造，日本コンピュータ協会，東京 (1980)).
7) Knuth, D. E.: *The Art of Computer Programming*, Vol. 1, pp. 190-196, Addison Wesley, Reading, Massachusetts (1968).
8) Kernighan, B. W. and Richie, D. M.: *The C Programming Language*, Prentice-Hall, Englewood (1978).（石田晴久(訳)：プログラミング言語C，共立出版，東京 (1981)).
9) Berry, R. E.: *Programming Language Translation*, Eills Horwood, England (1981).（武市正人(訳)：プログラム言語の処理系，近代科学社，東京 (1983)).
10) 中村，中西，不破：パーソナルコンピュータの簡易ネットワーク装置とその応用について，信州大学工学部紀要，No. 53 (1982).

(昭和 61 年 6 月 2 日受付)
(昭和 61 年 10 月 8 日採録)

中村 八束（正会員）
昭和 18 年生．昭和 46 年，東京工業大学大学院理工学研究科数学専攻博士課程修了．同年信州大学工学部講師．現在，同大学工学部情報工学科教授．理学博士．情報理論，パターン認識，音声・画像処理，マイコン応用技術等の研究に従事．電子通信学会，日本数学会等会員．

山崎 靖夫（正会員）
昭和 36 年生．昭和 61 年，信州大学大学院工学研究科情報工学専攻修士課程修了．同年日本・データゼネラル(株)入社．論理回路の CAD，DA の研究開発に従事．電子通信学会，日本ソフトウェア科学会各会員．

不破 泰（正会員）
昭和 33 年生．昭和 58 年，信州大学大学院工学研究科情報工学専攻修士課程修了．現在，信州大学工学部情報工学科助手．並行処理，ネットワーク，マイコン応用技術等の研究に従事．電子通信学会，計測自動制御学会各会員．