ソフトウェア工学　56－6
（１９８７．９．９）

Ｖ６０リアルタイムＯＳにおける マルチタスクデバッガの実現

中本幸一　沓掛由美子　橋本一也　阿部隆
日本電気（株）マイクロコンピュータソフトウェア開発本部

Ｖ６０リアルタイムＯＳは，マルチタスク環境とランデブによるタスク間通信機能を特徴としたリアルタイムＯＳである．本稿では，Ｖ６０リアルタイムＯＳの提供するマルチタスク環境下で並行に動作する複数のタスクからなるアプリケーションプログラムをデバッグするためのデバッガの仕様と実現法について述べる．本デバッガは，プログラマの有する概念に近いレベルでのデバッグ環境を提供すべく，単一タスクに着目したデバッグ，タスクの生成・消滅やランデブなどのタスク間の同期・通信機能に着目したデバッグ・トレース，複数のタスクにおいて発生するイベントの生起順序に着目したデバッグが可能である．

AN IMPLEMENTAION OF A MULTITASK DEBUGGER UNDER V60 REALTIME OPERATING SYSTEM

Yukikazu NAKAMOTO, Yumiko KUTUKAKE, Kazuya HASHIMOTO and Takashi ABE

Microcomputer Software Development Laboratory
14-22, Shibaura, 4-chome, Minatoku, Tokyo 108, JAPAN

V60 Real Time Operating System(RTOS), designed for NEC's 32-bit microprocessor V60, provides multitask environment and rendezvous-based intertask communication facility. This paper describes a multitask debugger for concurrent tasks under V60 RTOS. The debugger provides intratask, intertask and concurrent execution debug facilities. Intratask debug facilities provide debug environment concentrated on task execution and states. With intertask debug facilities a user can debug multitask, concentrating on task synchronization and communication. Concurrent execution debug facilities are provided for concurrent task behavioral level debugging. Using these facilities, a user can debug tasks based on concurrent programming model and high level programming language model.

1．はじめに

組込み型システムは，その高度化・多様化が要請され，これに伴ってその機能を実現するアプリケーションソフトウェアは複雑になり，その開発規模も増大してきている．このような複雑なアプリケーションソフトウェアを開発するために，ＯＳの提供する機能の高度化が図られ，ユーザの欲するモデルでのアプリケーションソフトウェアの開発を可能にしてきている．又，大規模なアプリケーションソフトウェアを開発するために，従来から使用されてきた機械語に代って，高級言語が使用されるようになってきた．

一方，このような組込み型システム上のソフトウェアをデバッグする手段としては，インサーキットエミュレータ(ICE)のようなデバッグ装置，又はモニタ等が用いられてきた．しかし，従来からのＩＣＥ，モニタ等のデバッグ手段を用いて，大規模なアプリケーション・ソフトウェアをデバッグしようとすると，以下のような問題が生じると考えられる．

1)デバッグ手段として原始的である

ＯＳ上のアプリケーション・ソフトウェアを開発するユーザは，ＯＳで提供されているプログラミングモデルを使ってアプリケーション・ソフトウェアを開発している．ところが，従来からのデバッグ手段では，これらのプログラミングモデルが直接サポートされないため，アプリケーション・ソフトウェアをデバッグする時に，ユーザは，ＯＳの提供するプログラミングモデルがＯＳ内で，どのように実現されているかを知る必要がある．例えば，タスクのメモリ内容やタスク間通信データの参照・変更のために，デバッグするユーザはＯＳの内部構造を知らなければならない時がある．このように，ユーザのプログラミングモデルにおけるプログラム記述時とデバッグ時の間のセマンティックギャップは，デバッグ効率向上を疎外する．

2)高級言語によるデバッグ手段がない

従来からのデバッグ手段の多くは，機械語レベルのデバッグ手段しか提供していない．このために，アプリケーション・ソフトウェアのデバッグの時に，高級言語で記述されたプログラムの機械語への展開の知識が必要となる．このようなプログラム記述時とデバッグ時の間のセマンティックギャップも，デバッグ効率向上を疎外する要因となる．

３２ビットマイクロプロセッサＶ６０／７０用リアルタイムＯＳ（以下ＲＴＯＳと略す）[1] は，先に述べた高度で大規模なアプリケーションへの対応を目的としたＯＳである．このために，リアルタイム制御機能だけでなく，制御処理で発生するデータに対する処理，制御のためのデータベース，又は高度なマンマシンインタフェースのための機能を提供するために，ＵＮＩＸを融合したした統合型リアルタイムＯＳである．

ＲＴＯＳでは，先に述べたデバッグ上の問題点に対処するために，マルチタスクの環境下で並行に動作する複数のタスク群からなるアプリケーションソフトウェアをデバッグするためのデバッガ[2] を提供している．本稿では，本デバッガの機能仕様，並びに実現方式について述べる．第２節では，本デバッガの機能を明らかにするために，ＲＴＯＳの機能の概略を述べ，ＲＴＯＳの提供するプログラミングモデルを述べる．第３節では，本デバッガの機能仕様を述べる．第４節では，本デバッガの実現方式について述べる．

2．ＲＴＯＳの機能

本節では，ＲＴＯＳの機能を述べる．

(1) マルチタスク環境

ＲＴＯＳのタスクは，リアルタイム制御用のリアルタイムタスクとＵＮＩＸプロセスに対応するＴＳＳタスクとに分類される．タスクはシステム構成定義言語（以下ＳＧ言語）[3] と呼ばれる言語を用いて定義される．ＳＧ言語のタスク定義部では，タスクの優先度，次に述べるランデブのエントリ名，対応するオブジェクトファイル名等が定義される．このようにして定義されたものをタスクタイプと言う．タスクタイプはタスクの論理的な単位である．タスクは，タスクタイプから生成される実際に実行される単位である．タスク／タスクタイプは，ＳＧ言語で定義されたタスク名／タスクタイプ名，又は，タスク／タスクタイプ生成システムコール時に返却される識別子（それぞれタスクトークン，タスクタイプトークンと呼ぶ）を用いて，ユーザにアクセスされる．図１にタスク／タスクタイプの定義例を示す．

(2) ランデブ機能

ＲＴＯＳでは，タスク間通信の手段として，Ａｄａのランデブの機能を提供している．ランデブのエントリ

```
/* タスクタイプ哲学者の定義 */
task    philosopher {
        objfile "phil.o" ;
            /* タスクタイプの
               オブジェクトファイルの定義 */
        roundrobin      ;
}       phil[5] ;
/* タスクタイプ哲学者から
   5個のタスクが定義されている */

/* タスクタイプダイニングテーブルの定義 */
task    diningtable {
        entry   get ;
        entry   free ;
                /* ランデブのエントリの定義 */
        objfile "dining.o" ;
        roundrobin      ;
}       dining ;
/* タスクタイプダイニングテーブルから
   1個のタスクが定義されている */
```

図1．システム構成定義の例

名は(1)で述べたように，ＳＧ言語でタスクタイプ毎に定義される．アプリケーションプログラム中で，ランデブを用いる場合は，リアルタイムＣ言語[3]を用いて記述する．リアルタイムＣ言語によるランデブのエントリ呼出し／アクセプトの記述例を図2，図3に示す．図2，図3は，「5人の哲学者の問題」の1つの解である．これは，フォークを管理するダイニングテーブルタスクと5つの哲学者タスクからなる．哲学者タスクがダイニングテーブルタスクに対して，エントリコールにより，フォークを取りにいき，食事をして，エントリコールでフォークを返す．ダイニングテーブルタスクには，フォークを取るエントリとフォークを返すエントリがある．

(3) その他の機能

ＲＴＯＳでは，タスク間通信の手段として，ランデブ以外に従来よりタスク間通信機能として提供されているセマフォ・イベントフラグ・メールボックスの機能を提供している．又，一定周期毎にタスクを生成・起動することのできる周期タスク，一定周期毎にタスクのある関数を実行する周期ハンドラの機能がある．更に，割込み／例外発生時に，タスクのある関数を実行する割込み／例外ハンドラの機能がある．

3．外部仕様

3．1設計目標

単一のプログラムをデバッグするのに比べて，複数の相互に同期・通信をとりながら並行に実行するプログラ

```
#define FREE 0
#define OK   1
#define NG   0

struct  forks {
        int f1 ;
        int f2 ;
        int result ;
} ;

int     fflag[5]= {FREE,FREE,FREE,FREE,FREE} ;

ros_main()
{

    while(1){
        select{
            accept free(fork) int *fork ;{
              fflag[*fork]=FREE ;
            }
            or
            accept get(forks) struct forks *forks ;{
                if(fflag[forks->f1] == FREE &&
                    fflag[forks->f2] == FREE){
                    fflag[forks->f1] = ~FREE ;
                    fflag[forks->f2] = ~FREE ;
                    forks->result = OK ;
                }else
                    forks->result = NG ;
            }
        }
    }
}
```

図2．ダイニングテーブルタスクの記述

```
#define SEC 1000
#define EATTIME 30*SEC
#define OK 1

#define PHILPTNS        3
#define PHILMED         0       /* meditating */
#define PHILEAT         1

extern TASK dining ;

struct  {
        int     f1 ;
        int     f2 ;
        int     result ;
}       forks ;

ros_main(pnum)
int *pnum ;
{
    forks.f1 = *pnum ;
    forks.f2 = (*pnum+1)%5 ;

    for(;;) {
        do{
            meditating(*pnum,rand()%10) ;
            call dining.get(forks) ;
        } while(forks.result != OK) ;
        eating(*pnum, rand()%10) ;
        call dining.free(forks.f1) ;
        call dining.free(forks.f2) ;
    }
}
```

図3．哲学者タスクの記述

ムのデバッグには，様々の困難が伴う．これには，次のような原因が考えられる．

(1) あるプログラムの動作の正当性の保証が，そのプログラムの実行状態のみによって決定されるのではなく，これと通信をとりながら実行する他のプログラムの実行状態にも依存する．

(2) 相互に同期をとる複数のプログラムの実行動作の関連を把握する必要がある．

このように，並行実行するプログラムのデバッグ時には，把握しなければならない情報の量が飛躍的に増大する．上記のような並行動作するプログラムのデバッグを支援するデバッガが幾つか提案，又は実現されている[5],[6],[7]．[5],[6]は，プログラムの並行動作レベルでのデバッグを支援するデバッガである．即ち，並行，又は並列動作するプログラムの振舞いを記述する言語を提供し，これを用いてユーザの意図する振舞いを記述させ，この動作記述と実際のプログラムの動作と比較し，一致，又は不一致したことにより並列動作するプログラムのデバッグを行う．しかし，プログラム内をデバッグする機能はない．[7] では，デバッグを並行動作するプログラムの内部のデバッグとプログラム間のデバッグに分け，プログラム内部のデバッグでは，通常の逐次型プログラムのデバッグ手法を用い，プログラム間のデバッグではプログラム間の通信に着目してデバッグを行うものである．しかし，[5],[6] のような並行動作レベルでのデバッグ機能はない．

筆者らは，第2節で述べたＲＴＯＳの提供する並行動作のプログラミングモデルに適合した並行マルチタスクデバッガを開発している．これは，[7] のプログラム内デバッグとプログラム間デバッグの手法を基にして，並行動作レベルでのデバッグ機能を可能としている．本デバッガでは次のような設計目標を設定した．

1)第１節で述べたデバッグ時のセマンティックギャップを解消すべく，ユーザのプログラム記述レベルでのデバッグ、ユーザの持つ並行プログラムモデルに基くデバッグを可能とすること．

2)タスクの状態を知るには、タスク内の状態だけでなく，タスクを管理するＯＳの内部状態も必要である．このため，ＯＳ内部の管理データの表示を行えるようにする．

3)デバッガの実行結果の了解性，デバッガコマンドの実行容易性を高めるために，マンマシンインタフェースを良くする．

３．２マルチタスクデバッガの機能

本節では第３．１節で述べた設計方針に基いて設けられたマルチタスクデバッガの機能について述べる．

(1) デバッグ対象タスク

本デバッガでは，デバッガが扱うこのできるタスクをデバッグ対象タスクと呼び，デバッグ対象以外のタスクを通常タスクと呼ぶ．通常タスクからデバッグ対象タスクへ，又デバッグ対象タスクから通常タスクへの移行は，デバッガコマンドにより動的に行うことができる．デバッグ対象の指定はタスク／タスクタイプの両方に可能である．タスクを指定した場合は，指定されたタスクがデバッグ対象となる．タスクタイプを指定した場合は，指定されたタスクタイプから生成される全タスクがデバッグ対象となる．

(2) タスク内デバッグ機能

本デバッガでは，タスクに閉じたデバッグ機能として，タスク毎に独立した実行制御機能，データ参照・変更機能がある．

a)実行制御機能

タスクの実行を制御する方法として，ブレークポイント，アドレストラップ，実行ブレークエントリコールがある．

・ブレークポイント

タスクの実行アドレスが、デバッガのコマンドで指定されたアドレスに到達した時に，指定されたタス

表１．同期・通信関係のシステムコール

ランデブ関連	
rnd_cll	エントリコール
rnd_acc	アクセプト(ランデブの開始)
rnd_end	ランデブの終了
rnd_snd	メッセージ通信
rnd_sdr	応答付メッセージ通信
rnd_pnd	ランデブの中断
rnd_rls	中断されたランデブの再開
セマフォ関連	
wai_sem	Ｐ命令
sig_sem	Ｖ命令
イベントフラグ関連	
set_flg	イベントフラグにセット
wai_flg	イベントフラグにセットされるのを待つ
メールボックス関連	
snd_msg	メッセージ通信
rcv_msg	メッセージ受信

クの実行中断，デバッガコマンドの実行を行う．ブレークポイントはタスク毎に独立して設定が可能である．又，テキストを共有する複数のタスクに対しても，独立して設定が可能である．高級言語レベルでは，タスク，ファイル，関数，行番号を指定して，行番号に対応する仮想アドレスにブレークポイントを設定できる．高級言語レベルでブレークポイントが設定できるには，コンパイル時にデバッガオプションでソースファイルがコンパイルされている必要がある．機械語レベルでは，タスク，仮想アドレスを指定して，ブレークポイントを設定できる．

- アドレストラップ

  タスクの実行時に，デバッガのコマンドで指定されたアドレスにデータを読み／書きした時に指定されたタスクの実行中断，デバッガコマンドの実行を行うことができる．タスク毎に独立して設定が可能である．高級言語レベルでは，タスク，ファイル，関数，式(lvalue)を指定して，式から計算されるアドレスに，アドレストラップを設定する．高級言語レベルでアドレストラップが設定できるには，コンパイル時にデバッガオプションでソースファイルがコンパイルされている必要がある．機械語レベルでは，タスク，仮想アドレスを指定して，アドレストラップを設定できる．なお，アドレストラップはＶ６０のデバッグサポート機能[8] を利用している．

- 実行ブレークエントリコール

  第４節で述べるように，デバッガはターゲットマシン上ではデバッガタスクとして実現されている．ユーザタスクからデバッガタスクの特定エントリに単純エントリコールを行うことにより，ユーザタスクの実行を中断することができる．これを実行ブレークエントリコールと呼ぶ．この使用法として，a)ユーザ埋込み方式によるブレークポイントとして，b)ユーザタスクとデバッガタスクの同期をとる手段が考えられる．

b)デバッグ機能

タスクの内部状態を参照／変更やタスクの命令部に対応するソースファイルを表示する機能である．

- タスクの内部状態の参照／変更

  高級言語レベルでは，タスク，ファイル，関数，式(lvalue) を指定して，式から計算されるアドレスのデータを，式の型に従って表示／変更することができる．又，データをユーザ指定の型で表示することもできる．機械語レベルでは，タスク，仮想アドレスを指定して，指定されたアドレスの内容を2進/8進/10進/16進/ASCII 形式で表示／変更することができる．更に，タスクのスタックフレームを関数名，仮アーギュメント名，実アーギュメント値，関数呼出しの行番号／アドレスを伴って表示する．

- タスクの命令部の表示

  高級言語レベルでは，タスク，ファイル，関数，行番号を指定して，タスクの命令部に対応するソースプログラムを表示する．機械語レベルでは，タスク，仮想アドレス指定して，命令部を逆アセンブル表示する。

(3) タスク間デバッグ機能

ＲＴＯＳでは，タスク間の同期・通信機能はＯＳレベルで，システムコールの形で提供されている．このシステムコールの例を表１に示す．ユーザはこれらのシステムコールでＯＳの事象を認識しうることになる．従って，タスク間のデバッグでは，システムコールの実行前／後を契機として，タスク間の同期・通信が行なわれた時に，ユーザが発見しようとするタスク間同期・通信を見い出し，この時，着目しているタスク群の実行を中断，タスク群の内部状態の参照／変更タスク間の同期パラメタや通信内容の参照／変更することになる．ユーザが発見しようとするタスク間同期・通信を見い出すために，ＯＳシステムコール記述言語を導入する．ＯＳシステムコール記述言語は，単一のシステムコールについて，発生するイベント中から，ある条件を満足するイベントのみを抽出するためのものである．図４にシステムコール記述言語の記述例を示す．

又，ＲＴＯＳでは，Ｃ言語にＡｄａのランデブ記述を組込んだリアルタイムＣ言語でランデブ機能を利用する．このため，リアルタイムＣ言語レベルでランデブによるタスクの同期・通信をデバッグするための，ＯＳシステム記述言語に対応するものとして，ランデブ記述言語を導入する．，ランデブ記述言語の記述例を図５に示す．

ユーザはシステムコール記述言語・ランデブ記述言語で記述された条件を満足するシステムコール・ランデブが発生した時，着目しているタスク群の実行を中断し，通常のデバッグコマンドを実行して，バグの原因を追求する．

(4) ＯＳイベントトレース機能

ここでいうイベントとは，(3) で述べたタスクの同期・通信のためのシステムコールに加えて，周期事象，割込み／例外事象といった非同期事象を含む．

・システムコール

前節で述べたシステムコール記述言語／ランデブ記述言語により，トレースしたいシステムコール／ランデブ記述を指定する．トレース情報としては，システムコールパラメタ（これは，タスク間同期・通信パラメタに対応する），システムコール実行時刻，システムコール実行タスクトークンである．ランデブについては，エントリ呼出し，アクセプトの関連を示すチャートを生成する．その例を図６に示す．

・非同期事象

非同期事象には，周期タスクの起動，周期ハンドラの起動，割込みハンドラの起動，例外の発生がある．トレースしたい種類の非同期事象を指定する．トレース情報としては，非同期事象の発生により起動されるタスク／ハンドラ，発生時刻などである．

```
task1@wai_flg^[a_flg==FLAG1]

ータスクtask1でフラグFLAG1で待つシステムコールが
ー終了した時に，イベント発生

task2@^sig_sem[a_sem==SEM1]

ータスクtask2がセマフォSEM1にＶ命令を発行した時に
ーイベント発生

注: task1@,task2@ は指定するタスクを表す
    タスク名，システムコール名ともワイルカード文字
    が可
    ^システムコール はシステムコールの実行前
    システムコール^ はシステムコールの実行後を表す
    []内に欲するシステムコールの実行条件を記述する
```

図４．システムコール記述言語の記述例

```
task1@ call *@.E1

ータスクtask1から任意のタスクのエントリEにコール
ーした時に，イベント発生

task2@.E2 accept:ok task1@ [(struct E *)$1->x<10]

ータスクtask2がタスクtask1からエントリE2でランデ
ーブが成立し，且つランデブパラメタの内容がある条件
ーを満足した時に，イベント発生

task3@ call:delay task1@.*

ータスクtask3からタスクtask1の任意のエントリにコー
ールし，タイムアウトした時に，イベント発生


注: task1@,task2@ は指定するタスクを表す
    タスクの次の.E1，.E2はエントリ名である
    タスク名，エントリ名ともワイルカード文字が可
    call，acceptはランデブ動作をを表すキーワード
    キーワードの直後の:ok，:delayはランデブ動作の
    終了状態を表す．:okは正常終了，:delayはタイム
    アウト終了を表す．
    []内に欲するランデブ条件を記述する
```

図５．ランデブ記述言語の記述例

(5) ＯＳ資源表示機能

タスク関連では，タスクのプライオリティ，走行時間，状態，タスクの実行待ちキューが表示される．ランデブ関連では，ランデブ待ちキュー，ランデブのネストの状態が表示される．又，セマフォ・イベントフラグのようなタスク間同期・通信用の資源では，各資源を待つタスクのキュー，資源の状態が表示される．

(6) イベント監視機能

並行プログラミングレベルでのデバッグ機能として，ブレークポイント・アドレストラップの発生，ＯＳイベント実行というユーザの認識すべき動作の系列を順路式に似た形式で記述し，この記述に基き実行が行われているか監視する．

(7) ユーザインタフェース機能の分離

本デバッガでは，ユーザインタフェース機能をデバッガから分離している．これにより，既存のウィンドウシステムのマルチウィンドウ機能やコマンドインタプリタの機能をデバッグ時に享受することができる．これにより次のような効果が得られる．

a) マルチウィンドによるマルチタスクのデバッグ

本デバッガでマルチウィンドウ機能を利用することにより，

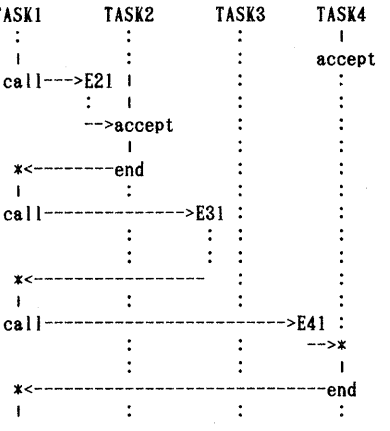

図６．ランデブトレーサの例

・タスク毎のデバッグ・セッションをマルチウィンドウのあるウィンドウに割りあてることにより，あるタスクのデバッグの際に他タスクのデバッグセッションが混在しないようにして，現在着目しているタスクのデバッグに集中できるようにする．尚，タスクのデバッグ・セッションのウィンドウへの割りあて方は任意である，

・あるウィンドウでデバッグ・セッションを開きながら，別のウィンドウでソースファイルを見る，

・あるウィンドウでデバッグセッションを行いながら，別のウィンドウでトレーサを実行するなど，ウィンウ毎に同時にコマンドを実行する，

などが可能である．

b)コマンドインタプリタの利用

本デバッガでは，通常ユーザが使用しているコマンドインタプリタでデバッガコマンドを実行できる．これにより，ユーザは新たなコマンドインタプリタの使用法を覚える必要がない．例えば，コマンドインタプリタとして，cshを利用していることにより，次のことが可能となる．

a)デバッガ・コマンドの実行結果をファイルに蓄積できる．

b)デバッガ・コマンドのコマンドファイルにすることができる．

c)デバッガ・コマンドの実行結果を他のＵＮＩＸコマンド(awk,grep etc)とパイプでの結合による加工が可能である．トレースコマンドの実行結果など対象となるタスク・ＯＳの資源が混在している時に，必要とする情報を抽出したい場合に，この機能は有用と思われる．

4．実現方式

4．1システム構成

本デバッガは，ハードウェア構成として，ＵＮＩＸをＯＳとしたマンマシンインタフェース機能を提供するホスト部とユーザタスクとデバッガタスクのあるターゲット部に分れる．

(1) ホスト部

第３節で述べたように，本デバッガは，既存のユーザインタフェースを利用するため，デバッガコマンドをデバッガのサブコマンドとはせずに，デバッガコマンドそれ自身をＵＮＩＸのコマンドとして実現している．このためデバッグの中間状態をバックグラウンドプロセスで記憶する必要がある．このようなバックグラウンドプロセスを，次に示す．

・シンボルテーブル管理プロセス

本プロセスは，タスクリンカ[3] の出力ファイルであるロードモジュールファイルのシンボルテーブル部を読込み，シンボルから内部表現値へ，又内部表現値からシンボルへの変換を行う．ここでシンボルとは，関数名，変数名，構造体／共用体のメンバ名などＣソースファイルで定義された名前である．更に，行番号からアドレス，アドレスから行番号への変換も行う．

・タスクライブラリ管理プロセス

システム構成ファイルからライブラリ生成ツール[3] タスクライブラリと呼ぶ．本プロセスはタスクライブラリを読込み，シンボルから内部表現値へ，又内部表現値からシンボルへの変換を行う．ここでシンボルとは，タスク名，ランデブのエントリ名，セマフォ名，イベントフラグ名などシステム構成ファイルで定義された名前である．更に，デバッグ対象タスクの管理も行う．

・ターゲットタスク管理プロセス

本プロセスは，デバッグ対象タスクに設定されたブレークポイントやアドレストラップ，ＯＳイベントの管理を，タスク単位で行う．即ち，ユーザの設定したイベントのイメージと次に述べるデバッガタスク内での管理情報の相互変換を行う．

デバッガコマンドは，上記バックグラウンドプロセスと通信しながら，ユーザレベルのデバッグコマンドイメージをデバッガの内部表現に変換して，ターゲット部にあるデバッガタスクに内部コマンドの形式で送信する．又，デバッガタスクから送信されてきたコマンドの実行結果を，ユーザレベルの表現に変換する．デバッガコマンドとバックグラウンドプロセス，及びバックグラウンドプロセス間のプロセス間通信は，ソケット(4.2bsd)，あるいはＦＩＦＯファイル(SYSTEM3) で実現している．

(2) ターゲット部

ターゲット部は，デバッガタスクとカーネル内デバッガ機能に分けられ，この間で機能分担を図っている．

a) デバッガタスク

・デバッガ内部コマンドの実行

・イベント条件成立の判定

• カーネル内管理データの参照

b) カーネル内デバッガ機能

• イベントの検出

• タスクの実行制御

• タスク空間(命令部／データ部)の参照／変更

システムコールの実行や非同期事象の検出は，カーネル内に，検出用のフラグを設け，カーネル内で検出し，検出したイベントをデバッガタスクに送る．デバッガタスクでは，送られてきたイベントの中から，ユーザの指定した条件を満足するイベントを見い出し，これをホスト部に送信する．イベントのトレース機能は，このような方法で実現している．

カーネル内デバッガ機能の中で，特に，実行制御に関係する機能について説明する．ＲＴＯＳのカーネルでは，タスクのブレークポイントやアドレストラップ，検出すべきシステムコールの実行などの情報も，タスクのコンテキストの一部と捕え，タスクのタスク管理ブロック（以下ＴＣＢ）の拡張エリアに，これらのデバッグ情報を管理している．そして，カーネルのディスパッチャがタスクに実行の制御を渡す時に，ＴＣＢ中にあるデバッグ情報をもとに，ブレークポイントやアドレストラップの設定を行う．ディスパッチャは，このようなデバッグ情報が設定されたタスクから実行の制御がなくなる時，設定されていたブレークポイントやアドレストラップが解除される．これにより，テキストを共有したタスクに対しても，タスク毎に独立したブレークポイントやアドレストラップの設定を可能としている．

5．あとがき

本デバッガは現在，ホスト部をPC9800/PC_UXで，ターゲット部をＶ６０ボードにより実現中である．

本稿では，主に，リアルタイムタスクをデバッグする機能について述べた．ＲＴＯＳでは，ＵＮＩＸプロセスもリアルタイムタスクと同時に走行するリアルタイムＵＮＩＸ(以下ＲＴＵＸ)の環境を提供している．ＲＴＵＸでは，ＵＮＩＸプロセスもリアルタイムタスクと同一のＴＣＢを使用しており[9]，本デバッガはそのまま複数のＵＮＩＸプロセスをも同時にデバッグできる環境を提供することになる．

又，ＲＴＵＸ上に，ホスト部の実行環境を移植することによって，ターゲットマシン上にデバッガのホスト部機能，デバッガタスク，デバッグ対象タスクを共存させることにより，ホストマシンなしでのデバッグ環境の実現も可能である．

機能面からは，タスク間同期・通信を疑似実行するスタブ機能の拡張が考えられる．

末筆ながら，本デバッガの開発に協力して頂いた方々に御礼申し上げます．

参考文献

[1] 古城,下島,南沢,永作,桑田,門田,小泉,寺本: "耐故障機構とランデブ機能を備えるＶ６０リアルタイムＯＳ",日経エレクトロニクス，3-23，No.417，1987.

[2] 中本,永江,古城:"Ｖ６０リアルタイムＯＳにおけるマルチタスクデバッガ"，第33回情処全大，３Ｖ-4，1986.

[3] 高橋,福岡,福田,森山,下島:"Ｖ６０リアルタイムＯＳのソフト開発サポート"，第34回情処全大，4Ｙ-7，1987.

[4] 土屋,高橋,北村,今村,古城:"Ｖ６０リアルタイムＯＳにおけるタスク間通信"，第33回情処全大，３Ｖ-2，1986.

[5] Bates,P.C. and Wileden,J.C.:"High-Level Debugging of Distributed Systems: The Behavioral Abstraction Approach", Journal of Systems and Software, No.3, 1983.

[6] Baiard,F., Francesco,N.D. and Valini,G.:"DEVELOPMENT OF A DEBUGGER FOR A CONCURRENT LANGUAGE", IEEE Trans. on SE, Vol.12, No.4, 1986.

[7] 野村,朴,高橋,天野,相磯:"並列計算機(SM)**2-II上のデバッグシステムの研究",第32回情処全大，３Ｇ-6，1986.

[8] Ｖ６０／Ｖ７０アーキテクチャマニュアル，NEC，1987.

[9] 下島,世良,水橋,古城,寺本:"Ｖ６０／Ｖ７０リアルタイムＵＮＩＸ－概要－"，第35回情処全大，３Ｄ-3，1987.