組み込みシステム向けリアルタイムOSの開発

堤　大祐，山本　寧，中野　隆司*
阿部　真澄*，菅原　博道**，川口　隆**

Development of Real-Time Operating System for Embedded Applications

Daisuke TSUTSUMI，Yasushi YAMAMOTO, Ryuji NAKANO *
Masumi ABE*, Hiromichi SUGAWARA**, Takashi KAWAGUCHI**

抄　録

近年，半導体技術の進歩により，CPUは高機能かつ高性能になり，メモリなどの記憶素子の容量も増大している。それに伴い，組み込みシステムの開発においても，システムの機能や性能が向上し，開発するソフトウェアの規模も大きくなった。このため，組み込みシステムの開発にはリアルタイムOSを用いてアプリケーション・ソフトウェアを開発することが一般化してきた。今回，日立製のH8Sシリーズの CPUをターゲットに，国内において事実上の標準となっているμITRON仕様のリアルタイムOSを民間企業と共同で開発し，実際の制御機器に組み込んで機能評価を行ったので報告する。

キーワード　リアルタイムOS，μITRON，自動給餌機

1．はじめに

CPUの高速化と高機能化により，組み込みシステムの機能や性能が向上し，システム全体の規模が大きくなってきた。また，組み込みシステムでは制御対象によって，システムの規模や要求仕様が異なるため，個別に開発を行うことが多く，ソフトウェア開発にかかる作業量が多くなっていた。

システムの規模が大きい場合，リアルタイムOSを用いてアプリケーション・ソフトウェアを開発することが一般化しており，汎用的に使用できるリアルタイムOSの必要性が高まっている。一方，市販されているリアルタイムOSは，リアルタイムOS自体の仕様やライセンス料などのコストの面から適用するシステムに合わないことがあり，自社開発したものを使用する場合も多かった。

今回，マイコンボードを主力製品として製造・販売している民間企業が，制御機器などに広く使用されている日立製のH8Sシリーズの CPUを搭載したマイコンボードを新規に開発した上，国内で業界標準となっているμITRON（Industrial The Real-time Operating system Nucleus）仕様でH8S用のリアルタイムOSを共同で開発した。本報ではリアルタイムOSの概要と実機での使用例について報告する。

2．リアルタイムOS

2.1　リアルタイムOSの有効性

組み込みシステムでは並行処理や時間経過処理，さらに，外部からの通信など非同期に生じる処理が求められることが多い。

CPUの性能が向上するに従い，組み込みシステムの規模が大きくなっており，制御の内容もより複雑になっている。リアルタイムOSを用いない従来の開発手法では，プログラミングの作業負荷が増大し，プログラムの維持・管理が非常に困難になる場合も多かった。

リアルタイムOSを用いると実行したい機能を「タスク」という単位で独立に記述することができ，プログラムを機能別にモジュール化できる。また，各タスクはそれぞれ独立に実行可能であり，システムコールを用いて必要に応じて同期をとりながら実行できる。

このように，リアルタイムOSを用いることにより，処理機能をタスクとして記述することによって，並列処理，時間管理処理，非同期処理が容易に実行でき，アプリケーション・

*株式会社北斗電子
**北原電牧株式会社

ソフトウェアの効率的な開発が可能となる。

った。このため，組み込みシステムの開発においても

ソフトウェアを開発すること

が一般化してきた。今回，日立製の H8S シリ

2.2 リアルタイム OS の仕様

今回開発したリアルタイム OS は μITRON3.0仕様に基づいている。

μITRON は組み込み向けリアルタイム OS であり，その仕様は公開されている。国内におけるリアルタイム OS の使用事例の中で，50% を越えるシェアがあり，組み込みシステムの分野では実質的な標準仕様となっている[1-3]。

μITRON3.0はその機能別にレベル R (Required)，レベル S (Standard)，レベル E (Expand)の 3 つに分類される。それぞれ，リアルタイム OS としての必須の機能，標準的な機能，拡張機能を示している。

個々の組み込みシステムでは，システムに必要な処理があらかじめ決まっている場合が多く，システム動作中に動的に新たな処理を追加することは少ない。そこで，開発するリアルタイム OS は必要な処理があらかじめ定義できるシステムを対象にする。

レベル E ではタスクやイベントフラグなどのオブジェクトを動的に生成するシステムコールをはじめ多くの拡張機能を含んでいる。今回のように，システムに必要なオブジェクトが実行前に定義可能な場合，レベル E の動的にオブジェクトを生成する機能は必要ないので，リアルタイム OS の開発はレベル S を基本に行った。レベル E の機能のうち，特に使用頻度が高いと思われるタイムアウト機能を追加した。

オブジェクトの生成を実行前に静的に行うソフトウェアであるコンフィグレータをあわせて開発した。

リアルタイム OS の開発は以下の手順で行った。

1．レベル S のリアルタイム OS にタイムアウト機能を追加したリアルタイムＯＳの開発
2．テストプログラムによるリアルタイム OS の機能評価
3．コンフィグレータの開発
4．テストプログラムによるコンフィグレータの機能評価
5．実際の制御システムにおけるリアルタイム OS の機能評価

μITRON3.0仕様（レベル S）ではタスクの状態を表 1 に示す 6 種の状態を定義している。タスクの状態はリアルタイム OS の機能を呼び出すシステムコールによって他の状態に遷移する。遷移の状態を図 1 に示す。

μITRON3.0仕様のシステムコールにはタスクの起動・終了などを行うタスク管理機能，タスクの強制待ち状態への移行・再開などを行うタスク付属同期機能，イベントフラグのセット・クリア，セマフォの獲得・返却，メイルボックスの送信・受信などを行う同期・通信機能，タスクの実行の遅延

開発したリアルタイム OS の仕様は表 2 に示す，合計 35 のシ

表 1　タスクの状態

タスクの状態		説明
RUN	実行状態	現在実行中のタスク
READY	実行可能状態	他のタスクが実行中のため、実行できない状態
WAIT	待ち状態	自タスクが発行したシステムコールによって実行が止まっている状態
SUSPEND	強制待ち状態	他タスクが発行したシステムコールによって実行が中断している状態
WAIT-SUSPEND	二重待ち状態	待ち状態と強制待ち状態が重なった状態
DORMANT	休止状態	タスクが起動される前、終了後の状態
		スタックポインタなどは起動時に初期化される

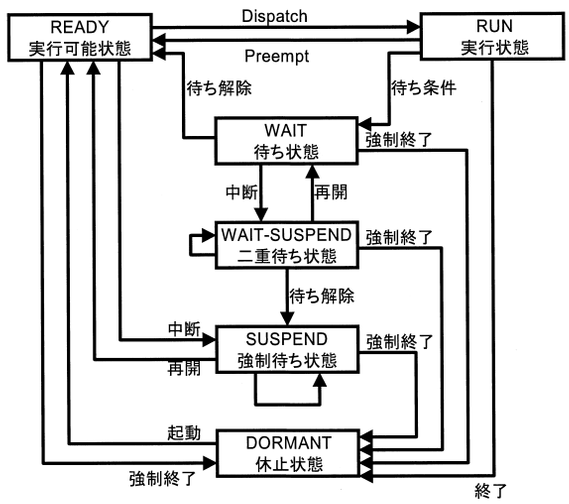

図 1　タスクの状態遷移図

表 2　システムコール一覧

	システムコール	機能
タスク管理	chg_pri	タスク優先度変更
	dis_dsp	ディスパッチ禁止
	ena_dsp	ディスパッチ許可
	ext_tsk	自タスク終了
	get_tid	自タスクのタスクIDの参照
	rel_wai	他タスクの待ち状態解除
	rot_rdq	タスクのレディーキュー回転
	sta_tsk	タスク起動
	ter_tsk	他タスク強制終了
タスク付属同期	can_wup	タスク起床要求を無効化
	rsm_tsk	強制待ち状態のタスクを再開
	slp_tsk	自タスクを起床待ち状態へ移行
	sus_tsk	他タスクを強制待ち状態へ移行
	tslp_tsk	自タスクを起床待ち状態へ移行(タイムアウトあり)
	wup_tsk	他タスクの起床
同期・通信	clr_flg	イベントフラグのクリア
	set_flg	イベントフラグセット
	sig_sem	セマフォ返却
	snd_msg	メイルボックスへ送信
	pol_flg	イベントフラグ待ち(ポーリング)
	prcv_msg	メイルボックスから受信(ポーリング)
	preq_sem	セマフォ獲得(ポーリング)
	rcv_msg	メイルボックスから受信
	trcv_msg	メイルボックスから受信(タイムアウトあり)
	twai_flg	イベントフラグ待ち(タイムアウトあり)
	twai_sem	セマフォ獲得(タイムアウトあり)
	wai_flg	イベントフラグ待ち
	wai_sem	セマフォ獲得
割込管理	loc_cpu	割込とディスパッチの禁止
	ret_int	割込処理からのリターン
	unl_cpu	割込とディスパッチの許可
時間管理	dly_tim	タスク遅延
	get_tim	システムクロック参照
	set_tim	システムクロック設定
システム管理	get_ver	バージョン参照

ステムコールが使用可能である。

2.3 リアルタイム OS の機能

2.3.1 スケジューリング

実行可能なタスクから次に実行するタスクを探す処理をスケジューリングという。μITRON3.0ではタスクの実行において優先度方式を採用している。そのため，優先度数だけのレディキューを持つ。実行可能なタスクはそれぞれの優先度のレディキューに接続している。実行するタスクの探索は高い優先度順にレディキューに接続しているタスクの有無を調べ，スケジューラはレディキューの先頭に接続しているタスクを実行する。

図２の場合，優先度１のレディキューにはタスク１，タスク３，タスク４の順に接続し，優先度２のレディキューには接続しているタスクはなく，優先度３のレディキューにはタスク２が接続している。このとき，リアルタイム OS は優先度がもっとも高い優先度１のレディキューの先頭に接続されているタスク１を実行する。

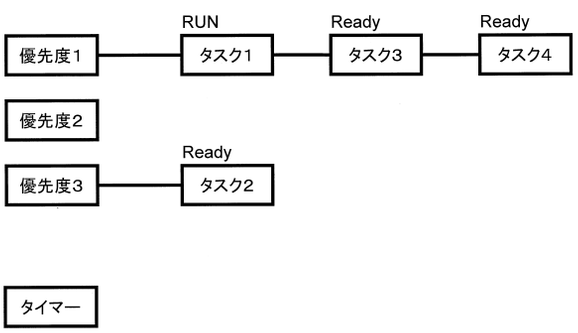

図２　レディキューの接続例

例えば，タスク１が実行中に時間待ち状態になった場合，リアルタイム OS はタスク１をレディキューから外し，タイマーキューに接続する。次に，優先度がもっとも高い優先度１のレディキューに接続している先頭のタスク３を実行する（図３）。

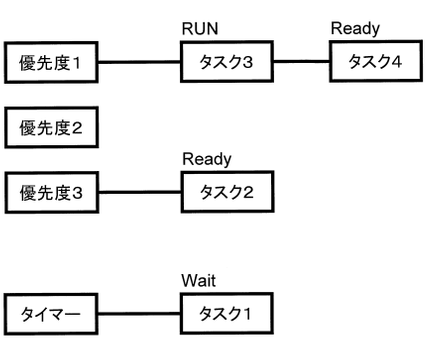

図３　実行中のタスクが待ち状態になったレディキューの接続例

キューはレディキュー，タイマーキューのほかにセマフォ待ちタスクを接続するセマフォキューなどがある。このようにリアルタイム OS はキューを操作してタスクをスケジューリングする。待ち状態を伴うシステムコールの発行や割り込み処理が発生したとき，リアルタイム OS は優先度の高いレディキューに接続しているタスクを探索し実行する。

タスクをキューから外したり，接続するキューの操作にはタスクの状態を管理するためのコントロールブロック TCB（Task Control Block）を用いて行っている。以下にキューの構造とコントロールブロックの構造について述べる。

2.3.2 キューの構造

レディキューは優先度数だけあり，キューに接続している先頭の TCB を指す。優先度の数はコンフィグレータにより指定する。

TCB には同じキューにつながっている次の TCB と前の TCB のポインタを持っていて，図４のような待ち行列を構成する。

新たにタスクがキューに接続する場合，末尾の TCB のあとに接続する。また，タイマーキューは待ち時間の少ない順に接続する。図５の場合，新たにタスク３が時間 T3の待ち状態になり，各タスクの待ち時間が T1＜ T3＜ T4の場合，タスク３はタスク１とタスク４の間に接続される。

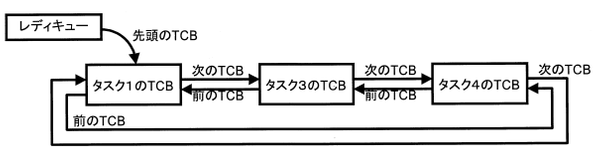

図４　レディキューの接続

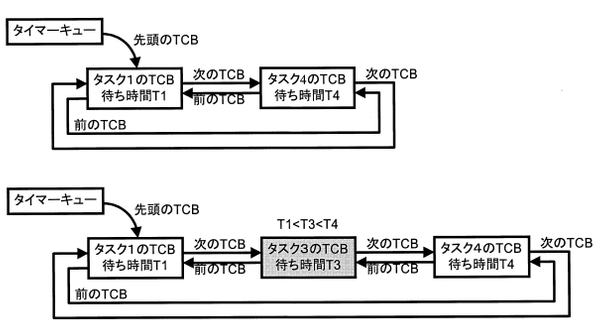

図５　タイマーキューの接続

2.3.3 コントロールブロックの構造

コントロールブロックにはタスクを管理する TCB のほか，フラグを管理する FCB（Flag Control Block），セマフォを管理する SCB（Semaphore Control Block），メイルボックスを

管理する MCB (Mailbox Control Block)があり，これらのコントロールブロックを用いてリアルタイム OS はシステムを管理する。

TCB にはタスクの番号，実行状態，スタックポインタ，優先度，待ち状態などタスクの情報を記述する。タスクの番号は他のタスクと区別するための識別子である。タスクの実行状態は実行状態(RUN)，実行可能状態(READY)，待ち状態(WAIT)，強制待ち状態(SUSPEND)，二重待ち状態(WAIT-SUSPEND)，休止状態(DORMANT)がある。タスクの待ち状態とは待ち状態(WAIT)の内容を示し，この待ちにはタイマー待ち，イベントフラグ待ち，セマフォ待ちなどがある。 TCB の構造を表 3 に示す。

表 3　TCB の構造

項目	説明
タスクID	タスクの番号(固定)
現在のスタックポインタ	レジスタなどを待避させる
タスクの状態	DORMANT、WAITなど
現在の優先度	接続されるレディキュー
待ち状態	時間待ち状態など
属しているキュー	レディキューなど
キューにおける次のTCB	
キューにおける前のTCB	
その他	

FCB， SCB， MCB はそれぞれ固有の番号と関連する情報が記述される。

3．コンフィグレータ

3.1　コンフィグレータの機能

今回開発した μITRON3.0仕様のリアルタイム OS では使用するタスク数やイベントフラグ数などをあらかじめ設定しておかなければならない。そのため，タスクの数，タスクの優先度，スタックサイズやイベントフラグの数などリアルタイム OS を用いた組み込みシステムの実行に必要な初期データを設定するコンフィグレータを開発した。

コンフィグレータはテキストファイルに一定のフォーマットに従って記述することにより， TCB などのコントロールブロックの初期状態を定義し，初期化を行うプログラムを自動的に生成するソフトウェアである。

タスクの設定にはタスク番号，実行するタスク（関数），スタックの大きさ，実行の優先度などを指定する。フラグ，セマフォ，メイルボックスの設定はそれぞれの番号や属性，初期状態を指定する。連想配列により，タスク番号などはラベルで指定することもできる[4-5)]。

システムクロック用タイマーの設定は任意の時間間隔でシステムクロックを定義できる。

3.2　コンフィグレータの動作

タスクを設定するにためには，タスク番号(tskid)と実行するタスク(task)は必須である。スタックの大きさ(stksz)，タスクの優先度(itskpri)はその設定を省略でき，省略した場合，初期値が設定される。この初期値は別に指定することが可能である。

これらの指定はテキストファイルに記述し（config.list)，コンフィグレータによって，初期化に必要なファイル（config.c， config.h）を生成する。 config.list， config.c， config.h の名称は自由に定義できる。

図 6 の場合，スタックの大きさ(stksz)は省略せず， word

```
taskcreat = {
    tskid = SHELLTASK;/* タスクＩＤ */
    task = ShellTask; /* 実行するタスク */
    stksz = 2048;     /* ユーザスタックサイズ */
    itskpri;          /* タスク起動時優先度 */
};
```

図 6　config.list の例（タスク管理テーブルの設定部分）

サイズ(16bit)で2048個分を確保することを示し，タスクの優先度(itskpri)はコンフィグレータが持っている初期値を用いる。また，この初期値は自由に変更することができる。

コンフィグレータは config.list の内容をより， config.c と

```
tcbtbl[0].tskid = SHELLTASK;
tcbtbl[0].task = (FP) ShellTask;
tcbtbl[0].itskpri = 1;
tcbtbl[0].isp = (VH*) &UsrStk1[2048];
```

図 7　config.c の例（図 6 に関する部分）

config.h を生成する。 config.c では図 7 に示すように，タスクの TCB を初期化する。指定したスタックサイズによりタスクのスタックポインタを自動的に設定する。

4．リアルタイム OS の動作例

4.1　タスクの構成

実機での動作例を示す。リアルタイム OS を自動給餌機に適用した。自動給餌機は濃厚飼料や粗飼料を各牛ごとに任意の割合で給餌できる省力化酪農機器である。自動給餌機はあらかじめ設定された時刻になると牛舎内を自走し，各牛の前で停止し，牛ごとに設定された量の飼料を与えるものである。自動給餌機は複数の種類の飼料を任意の割合で給餌できる。給餌量は給餌モータの動作時間で制御する。

自動給餌機の外観を図 8 に，自動給餌機の制御盤を図 9 にそれぞれ示す。図 9 の白丸で囲った部分がリアルタイム OS が動作するマイコンボードである。

自動給餌機で実行されるタスクの一部を図10に示す。図中

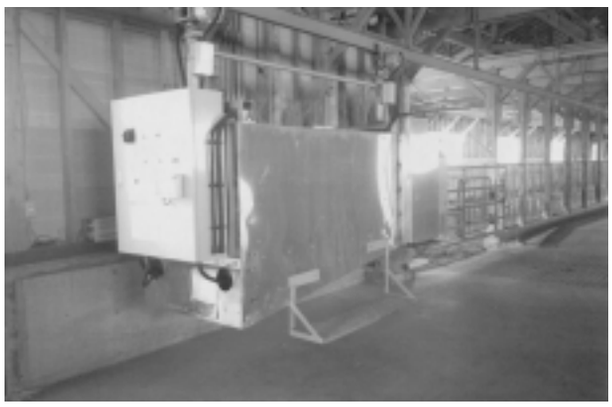

図8　自動給餌機

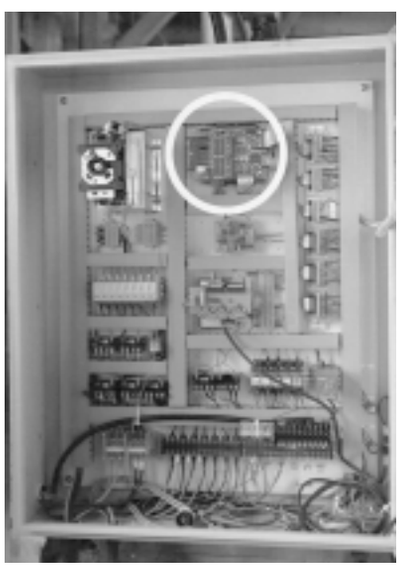

図9　自動給餌機の制御盤
（白丸がリアルタイムOSが動作するマイコンボード）

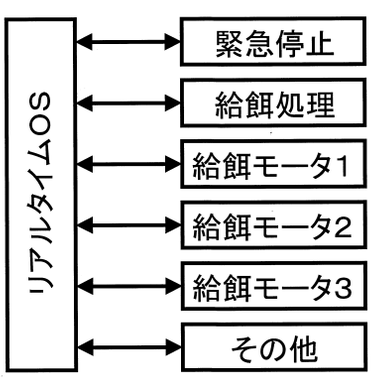

図10　自動給餌機におけるタスクの構成

の緊急停止タスクは緊急停止ボタンを定期的に監視し，ボタンが押下されたときは自動給餌機を緊急停止させるものである。給餌処理タスクは各牛に対して，設定された量を給餌するため，各給餌モータを起動する。給餌モータタスクは与えられた時間だけ各給餌モータを動作させるタスクである。また，各タスクのプログラムを図11，図12，図13にそれぞれ示す。

```
void
CheckButton() {
    while (1) {
      dly_tsk(interval);  /* 一定時間待つ */
       if (BUTTON) {       /* I/Oポートのチェック */
           Action();
       }
    }
}
```

図11　緊急停止タスクの例

```
void
Feeding() {
    while (1) {
        :
        wai_flg(NULL,FEEDINGPOINT,P,mode);
        /* 給餌地点に到着後，  */
        /* 給餌モータ制御タスクを起動する */
        sta_tsk(MOTOR1,FeedingTime1);
        dly_tsk(wait);
        sta_tsk(MOTOR2,FeedingTime2);
        dly_tsk(wait);
        sta_tsk(MOTOR3,FeedingTime3);
        /* イベントフラグを用いて */
        /* すべての給餌の終了を確認する */
        wai_flg(NULL,END_OF_FEEDING,X,mode);
        :
    }
}
```

図12　給餌処理タスクの例

```
void
MotorN(int time) {
    MotorN = ON;
    /* モータを始動し給餌時間だけ待つ。 */
    dly_tsk(time);
    MotorN = OFF;
    /* モータを停止し給餌終了を通知する。 */
    set_flg(END_OF_FEEDING,Bit(N));
    /* 自タスクを終了する。 */
    ext_tsk();
}
```

図13　給餌モータNタスクの例（N＝1，2，3）

4.2　制御プログラムの開発フロー

リアルタイムOSを用いた制御プログラムの開発フローを

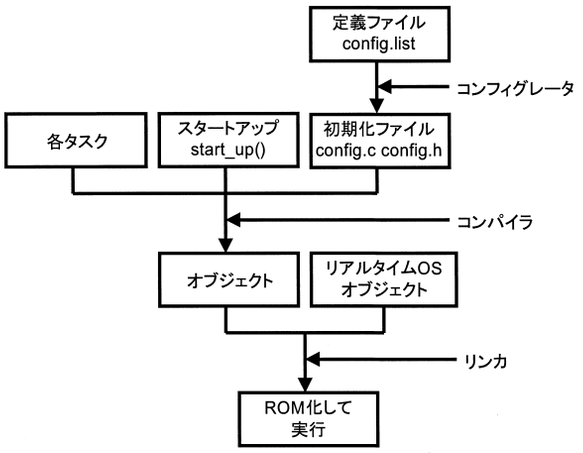

図14　リアルタイムOSを用いた制御プログラムの開発フロー

図14に示す。

リアルタイムOSで処理を行う場合，記急停止タスクなど必要なタスクを記述したファイルを作成する。次に，タスクやイベントフラグなど必要なオブジェクトを静的に生成する必要がある。これを支援するソフトウェアがコンフィグレータで，必要な情報をconfig.listに記述することで，指定したオブジェクトを生成する初期化ファイルconfig.c，config.hを生成する。

この適用例では緊急停止タスク,給餌処理タスク,給餌モータ制御タスク１，２，３の５つのタスクと給餌地点到着を示すイベントフラグと全給餌終了を示すイベントフラグを生成する。図15に定義例の一部を示す。

```
/* 緊急停止タスクの設定 */
taskcreat = {
    tskid = CHECKBUTTON;
    task = CheckButton;
    itskpri= 1;
};
/* 給餌処理タスクの設定 */
taskcreat = {
    tskid = FEEDING;
    task = Feeding;
    itskpri= 2;
};
/* 給餌地点到着フラグの設定 */
taskcreat = {
    flgid = FEEDINGPOINT;
};
```

図15　config.listの例

図15に示したconfig.listよりコンフィグレータはconfig.c，config.hを生成する。生成したconfig.cの一部を図16に，config.hの一部を図17にそれぞれ示す。図16ではタスクのID，実行する関数，実行する優先度を設定し，イベントフラグのIDとフラグの初期パタンを設定する。図17では最大優先度数，最大タスク数の設定，コンフィグレータで指定したタスク番号やフラグ番号のラベルの定義を行う。

```
tcbtbl[0].tskid = CHECKBUTTON;
tcbtbl[0].task = (FP) CheckButton;
tcbtbl[0].itskpri = 1;
tcbtbl[1].tskid = FEEDING;
tcbtbl[1].task = (FP) Feeding;
tcbtbl[1].itskpri = 2;
fcbtbl[0].flgid = FEEDINGPOINT;
fcbtbl[0].iflgptn = 0;
```

図16　生成したconfig.c（一部）

```
#define MAX_TSKPRI (8) /* 最大優先度数 */
#define MAX_TSK 10 /* 最大タスク数 */
#define CHECKBUTTON 1
#define FEEDINGPOINT1
```

図17　生成したconfig.h（一部）

次に，スタートアッププログラムを作成する。これは，システム初期化後，どのタスクを実行するかを指定する。指定はstart_up()関数で行う。この動作例では緊急停止タスクと給餌処理タスクをここで起動する。図18にスタートアップタスク指定の例を示す。

```
void
start_up() {
    sta_tsk(CHECKBUTTON, 0);
    sta_tsk(FEEDING, 1);
}
```

図18　スタートアップタスク指定

以上のように，各タスクを記述したファイル，初期化ファイルconfig.c，config.hとスタートアップstart__up()を記述したファイルをコンパイルし，リアルタイムOSのオブジェクトとリンクしたものをROM化するなどして実行する。

4.3　動作結果

これらのタスクの動作を図19に示す。図の横軸は時間の経過を示し，縦軸はタスクを示す。●点でタスクが動作したことを示す。給餌処理タスクは給餌地点に到着後，給餌モータタスク１，２，３を起動する。給餌モータタスクはそれぞれ

の給餌モータをONにしたあと，それぞれ指定された時間だけ待ち状態にはいる。その後，リアルタイムOSによって起床され，給餌モータをOFFにする。これで，それぞれ指定

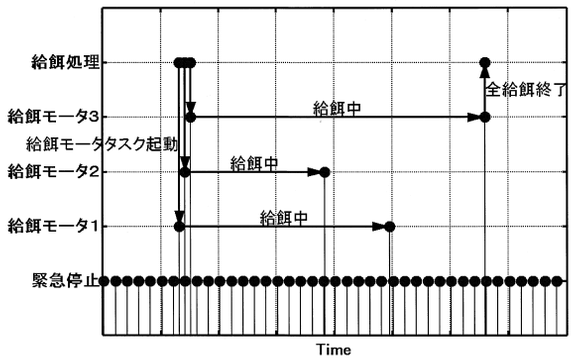

図19　自動給餌機による給餌タスクの動作

された時間だけ給餌モータをONにすることができる。また，緊急停止タスクはこれらのタスクとは独立に一定間隔ごとに緊急停止スイッチの状態を監視する。

以上のように，実機において，開発したリアルタイムOSが正常に動作することを確認した。また，リアルタイムOSを用いることにより，タスクを独立に記述でき，効率的なプログラム開発を行うことができた。

5．ま と め

H8S用のμITRON3.0仕様のリアルタイムOSを開発した。実際に自動給餌機にこのリアルタイムOSを適用してその動作を確認した。

リアルタイムOSを用いることにより，必要な機能をタスクとして独立に記述できた。それにより，複数のモータの同時制御などの並列性，イベントフラグを用いたタスク間の同期機能を容易に実現できた。また，パソコンなどとの通信に伴う非同期処理も容易に行えた。

今後はより高性能で高機能な日立SHシリーズをはじめ，種々のCPUへの対応と，少ないメモリで動作するワンチップ・モードへの対応を予定している。

引用文献

1）μITRON3.0標準ハンドブック，パーソナルメディア，1993
2）ITRON標準ガイドブック2，パーソナルメディア，1994
3）http://www.tron.org/tronproject/tp_bsc.html
4）プログラミングPerl改訂版，オライリー・ジャパン，1997
5）http://developer.gnome.org/doc/API/glib/glib-hash-tables.html # GHASHTABLE