お客様各位

カタログ等資料中の旧社名の扱いについて

2010年4月1日を以ってNECエレクトロニクス株式会社及び株式会社ルネサステクノロジが合併し、両社の全ての事業が当社に承継されております。従いまして、本資料中には旧社名での表記が残っておりますが、当社の資料として有効ですので、ご理解の程宜しくお願い申し上げます。

ルネサスエレクトロニクス ホームページ（http://www.renesas.com）

2010年4月1日
ルネサスエレクトロニクス株式会社

【発行】ルネサスエレクトロニクス株式会社（http://www.renesas.com）

【問い合わせ先】http://japan.renesas.com/inquiry

ご注意書き

1. 本資料に記載されている内容は本資料発行時点のものであり、予告なく変更することがあります。当社製品のご購入およびご使用にあたりましては、事前に当社営業窓口で最新の情報をご確認いただきますとともに、当社ホームページなどを通じて公開される情報に常にご注意ください。
2. 本資料に記載された当社製品および技術情報の使用に関連し発生した第三者の特許権、著作権その他の知的財産権の侵害等に関し、当社は、一切その責任を負いません。当社は、本資料に基づき当社または第三者の特許権、著作権その他の知的財産権を何ら許諾するものではありません。
3. 当社製品を改造、改変、複製等しないでください。
4. 本資料に記載された回路、ソフトウェアおよびこれらに関連する情報は、半導体製品の動作例、応用例を説明するものです。お客様の機器の設計において、回路、ソフトウェアおよびこれらに関連する情報を使用する場合には、お客様の責任において行ってください。これらの使用に起因しお客様または第三者に生じた損害に関し、当社は、一切その責任を負いません。
5. 輸出に際しては、「外国為替及び外国貿易法」その他輸出関連法令を遵守し、かかる法令の定めるところにより必要な手続を行ってください。本資料に記載されている当社製品および技術を大量破壊兵器の開発等の目的、軍事利用の目的その他軍事用途の目的で使用しないでください。また、当社製品および技術を国内外の法令および規則により製造・使用・販売を禁止されている機器に使用することができません。
6. 本資料に記載されている情報は、正確を期すため慎重に作成したものですが、誤りがないことを保証するものではありません。万一、本資料に記載されている情報の誤りに起因する損害がお客様に生じた場合においても、当社は、一切その責任を負いません。
7. 当社は、当社製品の品質水準を「標準水準」、「高品質水準」および「特定水準」に分類しております。また、各品質水準は、以下に示す用途に製品が使われることを意図しておりますので、当社製品の品質水準をご確認ください。お客様は、当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途に当社製品を使用することができません。また、お客様は、当社の文書による事前の承諾を得ることなく、意図されていない用途に当社製品を使用することができません。当社の文書による事前の承諾を得ることなく、「特定水準」に分類された用途または意図されていない用途に当社製品を使用したことによりお客様または第三者に生じた損害等に関し、当社は、一切その責任を負いません。なお、当社製品のデータ・シート、データ・ブック等の資料で特に品質水準の表示がない場合は、標準水準製品であることを表します。

   標準水準： コンピュータ、OA 機器、通信機器、計測機器、AV 機器、家電、工作機械、パーソナル機器、産業用ロボット

   高品質水準： 輸送機器（自動車、電車、船舶等）、交通用信号機器、防災・防犯装置、各種安全装置、生命維持を目的として設計されていない医療機器（厚生労働省定義の管理医療機器に相当）

   特定水準： 航空機器、航空宇宙機器、海底中継機器、原子力制御システム、生命維持のための医療機器（生命維持装置、人体に埋め込み使用するもの、治療行為（患部切り出し等）を行うもの、その他直接人命に影響を与えるもの）（厚生労働省定義の高度管理医療機器に相当）またはシステム等
8. 本資料に記載された当社製品のご使用につき、特に、最大定格、動作電源電圧範囲、放熱特性、実装条件その他諸条件につきましては、当社保証範囲内でご使用ください。当社保証範囲を超えて当社製品をご使用された場合の故障および事故につきましては、当社は、一切その責任を負いません。
9. 当社は、当社製品の品質および信頼性の向上に努めておりますが、半導体製品はある確率で故障が発生したり、使用条件によっては誤動作したりする場合があります。また、当社製品は耐放射線設計については行っておりません。当社製品の故障または誤動作が生じた場合も、人身事故、火災事故、社会的損害などを生じさせないようお客様の責任において冗長設計、延焼対策設計、誤動作防止設計等の安全設計およびエージング処理等、機器またはシステムとしての出荷保証をお願いいたします。特に、マイコンソフトウェアは、単独での検証は困難なため、お客様が製造された最終の機器・システムとしての安全検証をお願いいたします。
10. 当社製品の環境適合性等、詳細につきましては製品個別に必ず当社営業窓口までお問合せください。ご使用に際しては、特定の物質の含有・使用を規制する RoHS 指令等、適用される環境関連法令を十分調査のうえ、かかる法令に適合するようご使用ください。お客様がかかる法令を遵守しないことにより生じた損害に関して、当社は、一切その責任を負いません。
11. 本資料の全部または一部を当社の文書による事前の承諾を得ることなく転載または複製することを固くお断りいたします。
12. 本資料に関する詳細についてのお問い合わせその他お気付きの点等がございましたら当社営業窓口までご照会ください。

注 1. 本資料において使用されている「当社」とは、ルネサスエレクトロニクス株式会社およびルネサスエレクトロニクス株式会社がその総株主の議決権の過半数を直接または間接に保有する会社をいいます。

注 2. 本資料において使用されている「当社製品」とは、注 1 において定義された当社の開発、製造製品をいいます。

RENESAS

アプリケーション・ノート

78K0/Kx2

サンプル・プログラム

簡易OS編

対象デバイス

78K0/KB2マイクロコントローラ
78K0/KC2マイクロコントローラ
78K0/KD2マイクロコントローラ
78K0/KE2マイクロコントローラ
78K0/KF2マイクロコントローラ

資料番号　U19214JJ2V0AN00（第2版）
発行年月　March 2009 N

〔メ　モ〕

目次要約

CMOSデバイスの一般的注意事項

① 入力端子の印加波形

入力ノイズや反射波による波形歪みは誤動作の原因になりますので注意してください。

CMOSデバイスの入力がノイズなどに起因して，V_{IL}（MAX.）からV_{IH}（MIN.）までの領域にとどまるような場合は，誤動作を引き起こす恐れがあります。入力レベルが固定な場合はもちろん，V_{IL}（MAX.）からV_{IH}（MIN.）までの領域を通過する遷移期間中にチャタリングノイズ等が入らないようご使用ください。

② 未使用入力の処理

CMOSデバイスの未使用端子の入力レベルは固定してください。

未使用端子入力については，CMOSデバイスの入力に何も接続しない状態で動作させるのではなく，プルアップかプルダウンによって入力レベルを固定してください。また，未使用の入出力端子が出力となる可能性（タイミングは規定しません）を考慮すると，個別に抵抗を介してV_{DD}またはGNDに接続することが有効です。

資料中に「未使用端子の処理」について記載のある製品については，その内容を守ってください。

③ 静電気対策

MOSデバイス取り扱いの際は静電気防止を心がけてください。

MOSデバイスは強い静電気によってゲート絶縁破壊を生じることがあります。運搬や保存の際には，当社が出荷梱包に使用している導電性のトレーやマガジン・ケース，または導電性の緩衝材，金属ケースなどを利用し，組み立て工程にはアースを施してください。プラスチック板上に放置したり，端子を触ったりしないでください。

また，MOSデバイスを実装したボードについても同様の扱いをしてください。

④ 初期化以前の状態

電源投入時，MOSデバイスの初期状態は不定です。

電源投入時の端子の出力状態や入出力設定，レジスタ内容などは保証しておりません。ただし，リセット動作やモード設定で定義している項目については，これらの動作ののちに保証の対象となります。

リセット機能を持つデバイスの電源投入後は，まずリセット動作を実行してください。

⑤ 電源投入切断順序

内部動作および外部インタフェースで異なる電源を使用するデバイスの場合，原則として内部電源を投入した後に外部電源を投入してください。切断の際には，原則として外部電源を切断した後に内部電源を切断してください。逆の電源投入切断順により，内部素子に過電圧が印加され，誤動作を引き起こしたり，異常電流が流れ内部素子を劣化させたりする場合があります。

資料中に「電源投入切断シーケンス」についての記載のある製品については，その内容を守ってください。

⑥ 電源OFF時における入力信号

当該デバイスの電源がOFF状態の時に，入力信号や入出力プルアップ電源を入れないでください。入力信号や入出力プルアップ電源からの電流注入により，誤動作を引き起こしたり，異常電流が流れ内部素子を劣化させたりする場合があります。

資料中に「電源OFF時における入力信号」についての記載のある製品については，その内容を守ってください。

M8E0710J

はじめに

対 象 者　このマニュアルは，78K0マイクロコントローラの応用システムを設計・開発するユーザを対象とします。

目　　的　簡易OSの仕様，使い方についてユーザに理解していただくことを目的とします。

構　成　このマニュアルは，大きく分けて次の内容で構成しています。

- 概　　説
- 仕様
- 使い方

読 み 方　このマニュアルの読者には，マイクロコンピュータおよびC言語に関する一般知識を必要とします。

実際の応用例を理解しようとするとき

→漢字表示デモンストレーション用ボードのユーザーズ・マニュアルを参照してください。

命令機能の詳細を理解しようとするとき

- 78K0マイクロコントローラの場合

→78K0 マイクロコントローラ ユーザーズ・マニュアル 命令編を参照してください。

凡　　例　データ表記の重み：左が上位桁，右が下位桁

アクティブ・ローの表記：$\overline{xxx}$（端子，信号名称に上線）

メモリ・マップのアドレス：上部－上位，下部－下位

注　：本文中に付けた注の説明

注意：気を付けて読んでいただきたい内容

備考：本文の補足説明

数の表記：2進数　… xxxxまたはxxxxB

10進数 … xxxx

16進数 … xxxxH

2のべき数を示す接頭語（アドレス空間，メモリ容量）：

K（キロ）… $2^{10} = 1024$

M（メガ）… $2^{20} = 1024^2$

G（ギガ）… $2^{30} = 1024^3$

関連資料　関連資料は暫定版の場合がありますが，この資料では「暫定」の表示をしておりません。あらかじめご了承ください。

★　**漢字表示デモンストレーション・ボードの関連資料**

資　料　名	資料番号	
	和　　文	英　　文
漢字表示デモンストレーション用ベース・ボード　ユーザーズ・マニュアル	U19207J	未定
漢字表示デモンストレーション用78K0/KF2ボード　ユーザーズ・マニュアル	U19208J	未定
漢字表示デモンストレーション用78K0R/KG3ボード　ユーザーズ・マニュアル	U19209J	未定
漢字表示デモンストレーション用V850ES/JG3ボード　ユーザーズ・マニュアル	U19210J	未定
78K0/Kx2サンプル・プログラム（漢字フォント編）アプリケーション・ノート	U19211J	未定
78K0R/Kx3サンプル・プログラム（漢字フォント編）アプリケーション・ノート	U19212J	未定
V850ES/Jx3サンプル・プログラム（漢字フォント編）アプリケーション・ノート	U19213J	未定
78K0/Kx2サンプル・プログラム（簡易OS編）アプリケーション・ノート	このノート	未定
78K0R/Kx3サンプル・プログラム（簡易OS編）アプリケーション・ノート	U19215J	未定
V850ES/Jx3サンプル・プログラム（簡易OS編）アプリケーション・ノート	U19216J	未定
フォント・ユーティリティ　ユーザーズ・マニュアル	U19527J	未定
78K0/Kx2 サンプル・プログラム（フォント選択編）アプリケーション・ノート	U19528J	未定
78K0R/Kx3 サンプル・プログラム（フォント選択編）アプリケーション・ノート	U19529J	未定
V850ES/Jx3 サンプル・プログラム（フォント選択編）アプリケーション・ノート	U19530J	未定
漢字表示デモンストレーション用拡張ボード　ユーザーズ・マニュアル	U19526J[注1]	未定
78K0/Kx2 サンプル・プログラム（ドットＬＣＤ制御編）アプリケーション・ノート	U19531J[注1]	未定
78K0R/Kx3 サンプル・プログラム（ドットＬＣＤ制御編）アプリケーション・ノート	U19532J[注1]	未定
V850ES/Jx3 サンプル・プログラム（ドットＬＣＤ制御編）アプリケーション・ノート	U19533J[注1]	未定

注1.　2009年夏発行予定

78K0**マイクロコントローラ・デバイスの関連資料**

資　料　名	資料番号	
	和　　文	英　　文
78K0/Kx2　ユーザーズ・マニュアル	U18598J	U18598E
78K0 ユーザーズ・マニュアル 命令編	U12326J	U12326E

注意　上記関連資料は予告なしに内容を変更することがあります。設計などには，必ず最新の資料をご使用ください。

目　　次

第1章　概　　説

この章では，本OSの特徴，開発環境，諸元について説明します。

1.1　特　　徴

> **注意**　本OSは，リアルタイムOS（RTOS）の代わりになるものではありませんし，互換性もありません。また動作保証やサポートもありませんので，製品組み込みOSが必要な場合は，当社製リアルタイムOSあるいはサード・パーティ製OSのご購入を推奨します。
> 本マニュアルは，本OSが組み込まれたサンプル・プログラムやデモンストレーション・プログラムの動作を理解したり，動作確認のために改造する場合の一助として作成しています。

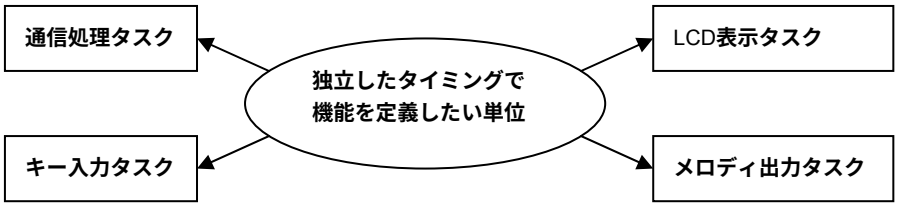
本OSは，簡単・軽量であるとともに，近年のソフトウエア開発の潮流であるモデリング設計を容易にコーディングに結びつけられることを狙ったマルチタスク簡易ＯＳです。

- 簡　単：ノンOSの代表的なプログラム構造であるラウンドロビン方式をベースとした簡単な構造のため，比較的容易に理解できて手軽に使えます。
- 軽　量：カーネル・サイズが1Kバイト以下と小さく，さらにシングル・スタック方式により小容量のRAMでも実用的なアプリケーション・プログラムを作れます。
- 容易化：状態遷移図（表）やデータ・フロー図と対比しやすいコーディングが容易にできるように，状態遷移システム・コールやパイプ機能を持っています。

補足．基本的な用語

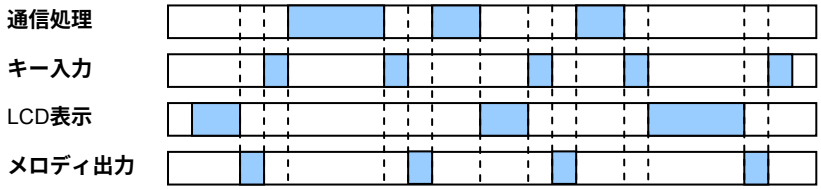
(1) タスク

独立したタイミングで機能を定義したい単位です。



(2) スケジューリング

マルチタスクOSでは，各タスクが巨視的には同時に動作しているように見えます。しかし，CPUは1つですから，実際には次々と切り替えながら処理を進めます。どのタスクを次に実行するかを決めることがスケジューリングです。

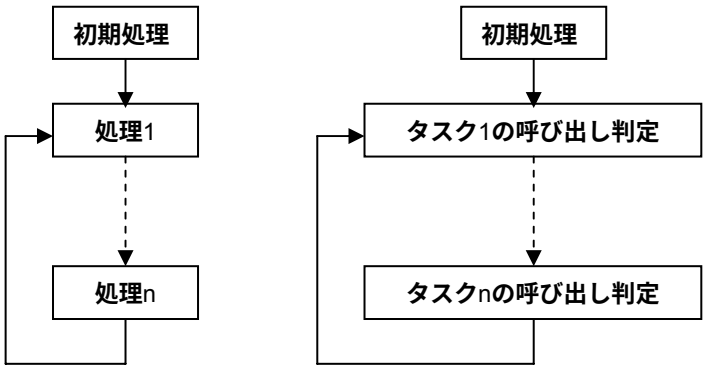


1. 1. 1　ラウンドロビン・スケジューリング

ノンOSの代表的なプログラム構造としてラウンドロビン処理があります。本OSはこれに準じるラウンドロビンによるスケジューリングを行います。タスクの状態を順番に調べて，実行可能な状態であればそのタスクを呼び出します。

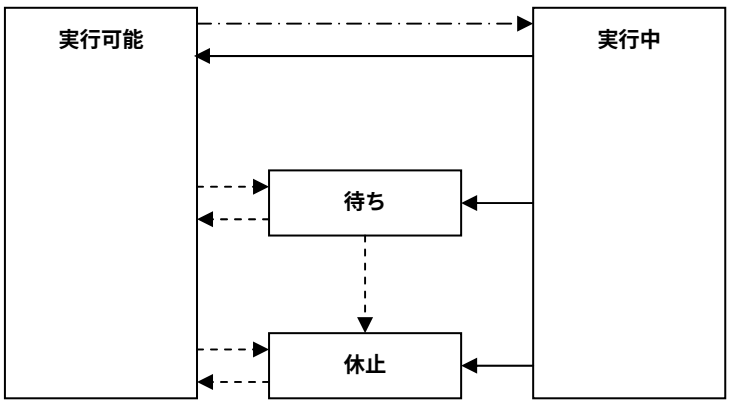
なお本OSでは，呼び出したタスクの中で所定時間（数ms）以内にスケジューラに戻るシステム・コールを発行する必要があります。後述（1. 1. 3項）の状態遷移設計を行うと，このシステム・コールを状態遷移のタイミングに合わせられるため，スケジューラへ戻ることをあまり意識せずにコーディングできます。

図1－1　ラウンドロビン方式



補足.　タスクの状態（OS管理上の状態）

OSはタスクの実行管理をするために次の図のように状態を定めています。基本的にはシステム・コール（OS提供の関数）によって状態を変えます（詳細は2章の仕様を参照）。



OS（スケジューラ）による呼び出し

実行中タスクが自分の状態を変えるシステム・コールを発行

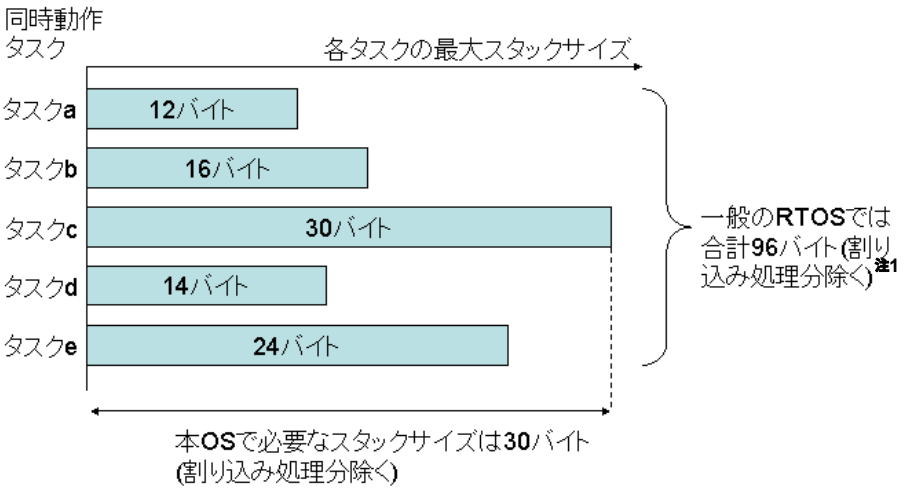
実行中タスクが他のタスクの状態を変えるシステム・コールを発行

1.1.2　シングル・スタック

一般のRTOS（リアルタイムOS）は，タスクごとにスタックを持つのでRAMサイズが肥大化しやすい傾向にあります。本OSは1本のスタックしか使いません。各タスクの中で一番深いサイズを要求するタスクのスタック・サイズと割込み用スタックがあれば足ります。

逆に，本OSではタスクからOSへ戻るときにスタックを開放しなければならないという制約があります。RAM容量の大きいMCUを使う場合はRTOSに比べると不自由ですが，タスクごとのスタック見積が荒くても他のタスクを破壊する危険性が少ないので，初心者でも比較的安心して本ＯＳを使うことができます。

図1－2　スタック・サイズ比較



★ **注1.　すべてのタスクのスタックが一番深い状態でウエイトがかかるようなワースト・ケース。**

1.1.3　状態遷移システム・コール

近年，ソフトウェア開発の上流設計においてモデリング技術が普及しつつあり，モデルの一つとして状態遷移図（表）を使った設計も珍しくありません。本OSでは，タスクの1状態を1関数に対応づけて，関数間を状態遷移システム・コールで移動することにより，状態遷移図と対比しやすいコーディングを行うことができます。

以下，図1－3を例に説明します。

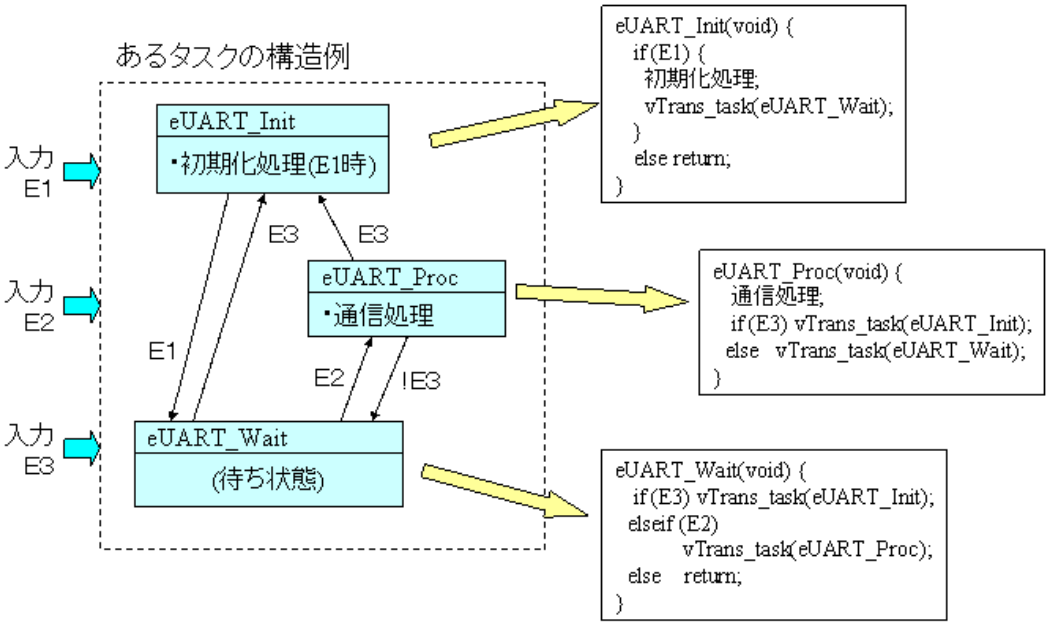
① 最初にOSから呼ばれるタスク・プログラムをeUART_Initとします。

② eUART_Initは入力E1が‘偽’であれば，return文を実行します。本OSではトップ階層のreturn文は自分自身へ戻るという遷移になります。従って，eUART_Initは入力E1が‘真’にならない限り何度でもOSから呼び出されます。

③ eUART_Initは入力E1が‘真’になるとvTrans_taskという状態遷移システムコールを発行します。これにより，次にOSがこのタスクを呼ぶとき，eUART_Waitを呼び出します。

④ 以下同様にreturnによる自分自身への遷移やvTrans_taskによる他の関数への移動を行いながら処理が進みます。

図1－3　状態遷移プログラミング



補足.　状態遷移とタスク切り替え

本OSはマルチ・タスク処理のためのタスク切り替えを自動では行いません。しかし，上記のように状態遷移設計を行うと状態遷移のタイミングでタスク切り替えが実施されるため，あまりタスク切り替えを意識せずにコーディングを行うことができます。

1.1.4　データ・フロー型タスク間通信

本OSでは，タスク間でデータやタイミングを受け渡す方法として次の方法が使えます。

・グローバル変数で受け渡し

・OS提供のイベント・フラグによるタイミング通知

・OS提供のパイプによるデータの非同期連続受け渡し

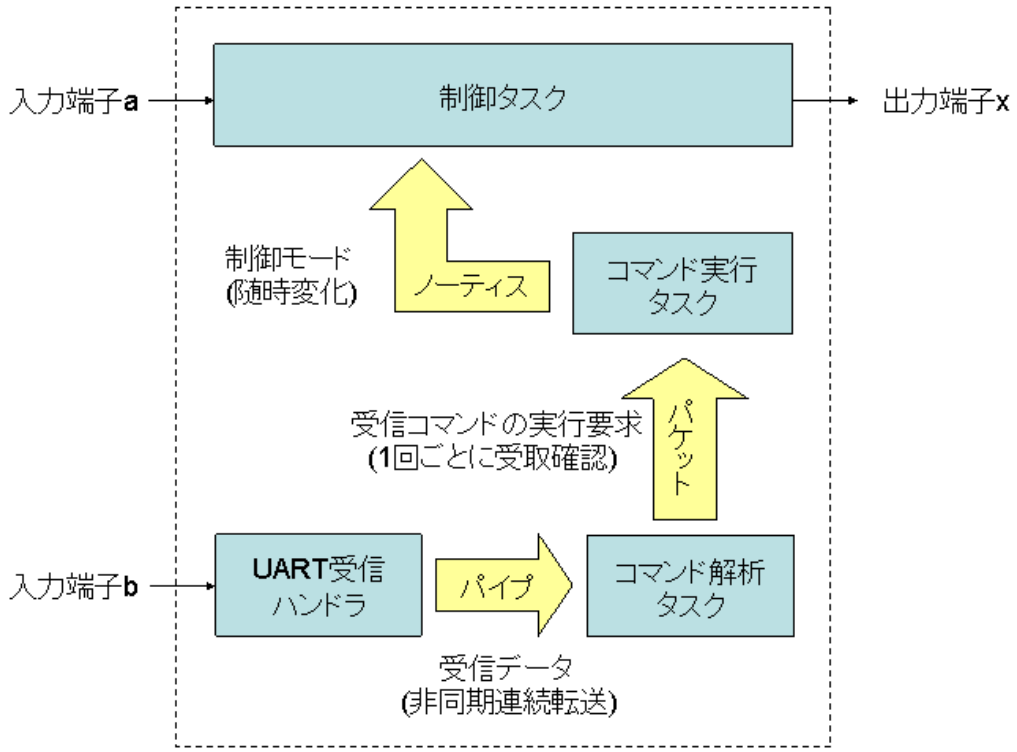
これをどのように使うかはユーザの自由ですが，データ・フロー設計を行いたい場合は，例えば次のように概念を定めて使い方を決めておくと良いでしょう。

・ノーティス：グローバル変数に随時値を書き込んで，相手タスクに一方的にデータを通知する方法です。

・パケット　：１回ごとに相手が受け取ったかどうか確認する方法です。グローバル変数を使ってユーザ・プログラムで受取管理（ハンドシェーク）することもできますが，イベント・フラグ機能を使うと，通知タイム・アウトの管理もできます。

・パイプ　　：送り側と受け側の間で，非同期で連続してデータを転送する方法です。OS提供のパイプ用システム・コールによりパイプへのデータ書き込みや読み出しを行います。

図1－4　データ・フロー設計の場合のデータ受渡概念例

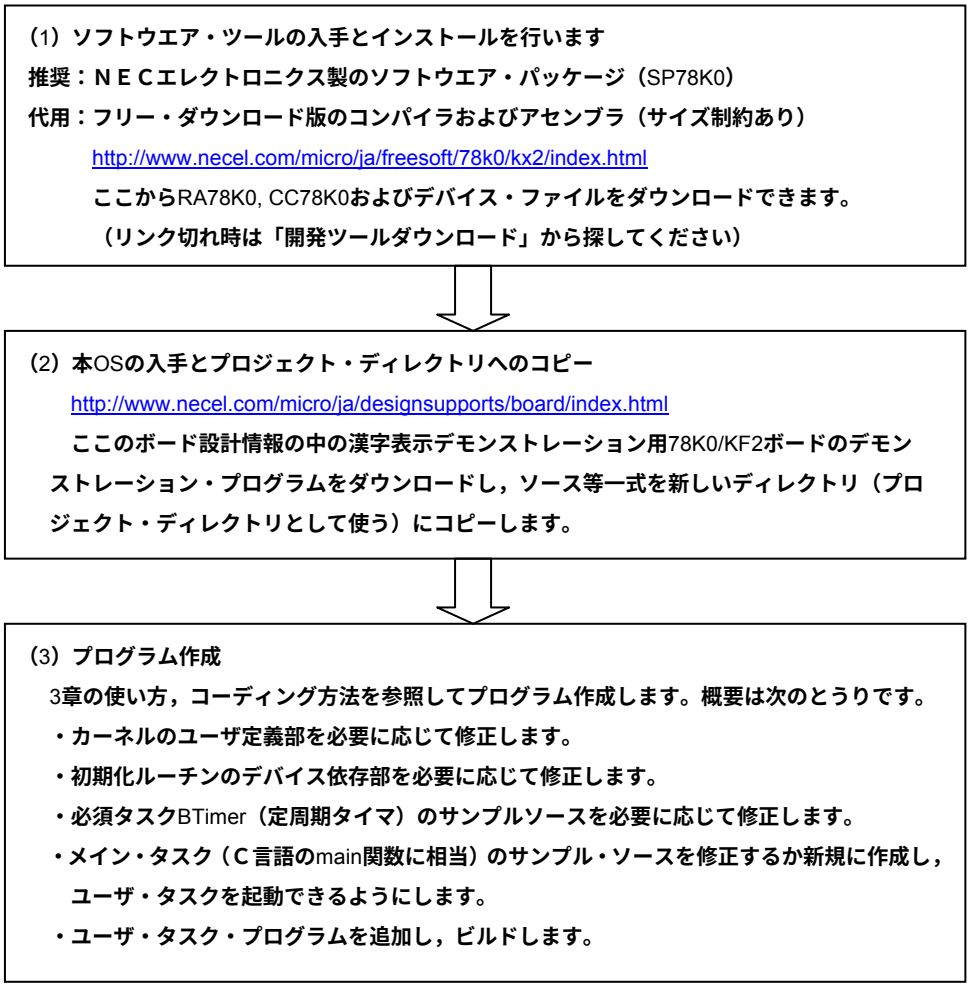


1. 2　開発環境

本OSを使用したアプリケーション・プログラムの開発に推奨するソフトウエア・ツールは，NECエレクトロニクス製のソフトウエア・パッケージ（SP78K0）です。フリー・ダウンロード版のコンパイラおよびアセンブラ（CC78K0, RA78K0）でも構いませんが生成できるオブジェクト・サイズに制限があります。

本OSのソース・プログラムは，漢字表示デモンストレーション・プログラムの一部として提供します。この中に含まれるユーザ定義部を書換えたり，ユーザプログラムを追加して最終的に1本のアプリケーション・プログラムとしてビルドします。

図1－5　開発環境構築・開発手順の概要



★

1.3 諸　元

本OSで定義可能な項目の上限値や消費する資源について次の表に示します。

★

表1－1　諸元一覧表

区分	項目	内容
カーネル種別		78K0（バンク非対応）版
スケジューラ	最大コード空間	64Kバイト
	同時動作タスク数（最大）	64個[注1]
	RAM消費量（固定部）	1バイト
	RAM消費量（タスク当り）	3バイト
	スタック消費量	2バイト
セマフォ	定義可能数	128個[注1]（1個で1資源の使用中／開放中を管理）
	RAM消費量	定義数÷8バイト（切り上げ）
イベント・フラグ	定義可能数	64個[注1]
	RAM消費量（フラグ当り）	2バイト
パイプ	定義可能数	64個[注1]
	パイプの長さ（可能範囲）	3, 7, 15, 31, 63, 127, 255バイトのいずれか
	RAM消費量（パイプ当り）	パイプの長さ＋3バイト
	定数用ROM消費量	定義数×4バイト
時間管理	最大カウント範囲	約497日（10 ms周期カウントの場合）
	RAM消費量	5バイト（周期カウント4バイト＋分周1バイト）
カーネル・サイズ	スケジューラ＋システムコール	1Kバイト以下[注2]
消費クロック数	スケジューラ	タスク数×136 + 75クロック（1ラウンドロビン当り）[注2]
備考		

注1.　実装するMCUの資源範囲を超える定義はできません。

注2.　測定機能（5章参照）が無い場合の暫定値です。

第2章　仕　　様

この章では，本OSの機能・構造およびシステム・コールの詳細について説明します。

2. 1　処理の優先順位

本OSではプログラムを以下のように区分して優先順序を定めています。

(1) 割り込みハンドラ

最優先で処理します。OSとしては多重割り込みは特にサポートしてないので，必要な場合はユーザが管理してコーディングします。

ハンドラ内でシステム・コールを発行する場合は，タスクから発行されたシステム・コールと干渉しないように注意が必要です。具体例は2.8節を参照してください。

(2) カーネル（スケジューラ，システム・コール）

処理の優先順位はタスクと同じです。従って，タスクが実行権を開放し無い限りスケジューラに処理が戻りません。ただし，2基本周期（10～20 ms）以上タスクが実行権を開放しない場合は暴走と見なしてタスクを強制終了し，スケジューラに処理を戻します。

(3) タスク

マクロ的に見たときに他のタスクと平行して同時に動作する処理単位です。ミクロ的には割込みハンドラによる中断や，システム・コールによるタスク切替えの中断が発生します。

なお，基本周期以内の時間でスケジューラに戻るシステム・コールを発行する必要があります。プログラム機能設計を状態遷移図に基づいて行う場合は，状態遷移システム・コールによって自動的にスケジューラに戻るため，スケジューラに戻ることをあまり意識しなくても済みます。

2. 2　タスク管理

2. 2. 1　タスクの状態管理

タスクの状態には，次の二つがあります。

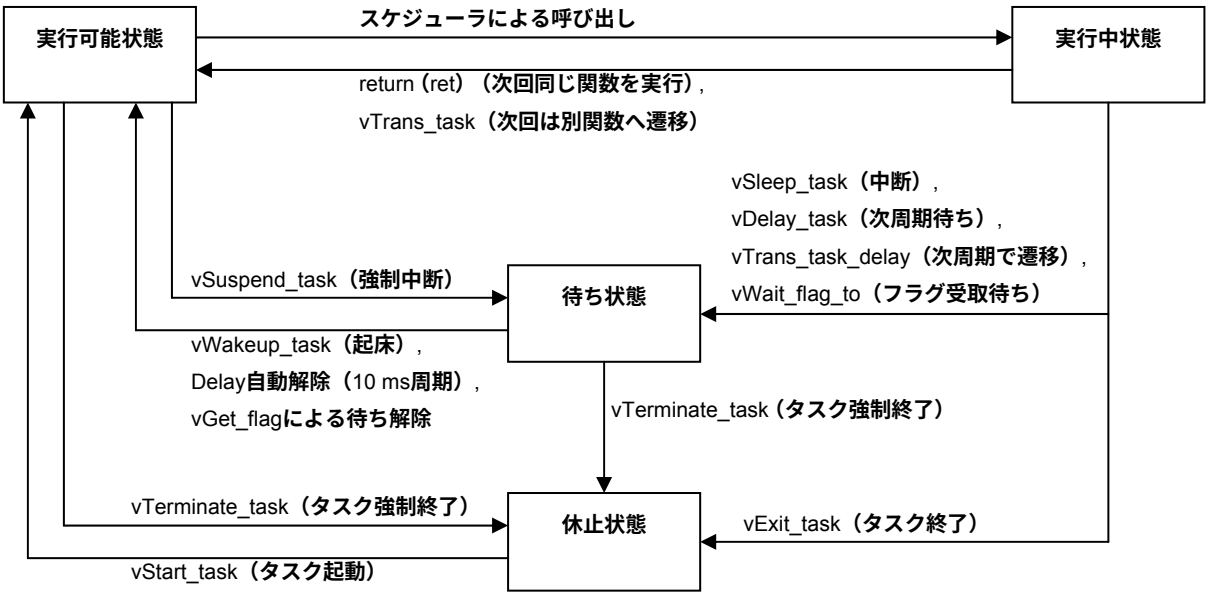
(1) 実行状態（ＯＳ管理上の状態）

スケジューラがタスク呼び出し制御を行うための状態です。実行中，実行可能，待ち，休止の4状態があります。これらの状態は，スケジューリングやシステム・コールによって変わります（図2－1参照）。

(2) 機能状態（ユーザ・プログラミング上の状態）

機能状態とは，アプリケーションから見たタスクの挙動を状態遷移で表した場合の状態を指します。機能状態はシステム・コールによって現在の状態を維持するか，次の状態へ遷移するかを指定できます（図2－2参照）。

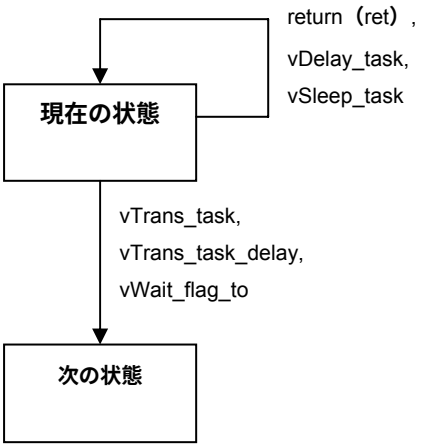
図2－1　OS管理上の実行状態の遷移



注　vで始まる名称はシステム・コール

return文もトップ階層関数ではシステム・コール扱い

図2－2　機能状態の遷移



システム・コール	遷移にかかる時間
return（ret），vTrans_task	すぐ（ラウンドロビン一巡後）
vDelay_task, vTrans_task_delay	次の基本周期（標準10 ms）
vSleep_task, vWait_flag_to	関連する他のタスクが所定の動作を行うまで待つ

2. 2. 2　タスク制御データ

タスクの状態はシステム・コールによって変更するので，通常は直接タスク制御データを操作する必要はありません。処理が間に合わない場合など，直接操作したい場合のために制御データの構造を説明します。操作する具体的な変数名については，後述の「システム定義定数・変数」を参照してください。

(1) TCB1

1タスク当り1バイトから構成するテーブルです。タスクの順序はユーザー定義ファイル（詳細3章）に記載したタスクの順序（ = タスクID番号）と同じです。

bit7：起動フラグ　　：タスクが起動状態（非休止状態）で‘1’になります。
bit6：実行可能フラグ　：実行可能状態または実行中は‘1’になります。
　　待ち状態では‘0’になります。
bit5：ディレー・フラグ：‘1’なら次の基本周期で自動的に実行可能状態にします。
　　システムコールvDelay_task，vTrans_task_delay用のフラグです。
bit4：補助フラグ　　：ユーザ定義フラグです。
bit3-0：上位アドレス　：タスクの実行アドレスの上位4bitを格納します。ただし64K版は0固定です。

(2) TCB2

1タスクあたり2バイトから構成するテーブルです。タスクの実行アドレスの下位16bitを格納します。

2. 2. 3　スケジューリング

スケジューラは，タスク登録順（ユーザ定義ファイルUDEF_a.hに記載した順）でTCBを調べ，実行可能ならタスクを呼び出します（TCBに格納してある実行アドレスをコールします）。

1タスクごとに基本周期（標準10 ms）のタイム・オーバがないかチェックし，オーバしている場合は最初のタスク（システム予約の基本周期タスク）のスケジュールに戻ります。

全てのタスクが実行可能で無い場合は，消費電力低減のためにHALTします。タイマやUARTなどの割り込みがかかるとスケジューリングを再開します。

2. 2. 4　タスクの定義方法

タスクの定義は、ユーザ定義ファイルUDEF_a.hの中に記述します。タスク名をvDefine_taskマクロで定義します。タスク・ハンドル名はタスク名の前にhを付けた名称になります。

- タスク・ハンドル名には登録順にID番号が付きます（0～63）。
- 最初に登録するタスク（ID = 0）は基本周期タスクに予約しています。タスク名はBTimer，ハンドル名はhBTimer，関数名はeBTimerに固定しています。
- ２番目以降にユーザ・タスクを登録しますが，通常２番目（ID = 1）にはメイン・タスクを割り当てます。タスク名はMain，ハンドル名はhMain，関数名はeMainに固定しています。
- メイン・タスクはMCUリセット後に最初に呼び出されるタスクです。このタスクの中で，他のタスクを順次起動します。通常はシステム予約の基本周期タスクを最初に起動します。

- UDEF_a.hでのタスク定義例

```
vDefine_task   BTimer ;基本周期タスク（システム予約タスク）
vDefine_task   Main   ;メイン・タスク。C言語のmain関数に相当します。
vDefine_task   User   ;ユーザ・タスク名の例。この場合の関数名はeUserになります。
```

・プログラム中でのハンドル名の使用方法

C言語ではvTASK_IDマクロでextern宣言してから使用します。

宣言文の例	extern　vTASK_ID(hUser);
システム・コールでの使用例	I = vRefer_task(hUser);

アセンブラでは単純にextrn宣言して使用します。

宣言文の例	extrn　_hUser_func
システム・コールでの使用例	vRefer_task　_hUser_func

（注意：アセンブラでは，ハンドル名，変数名，関数名の先頭に_が付きます）

2.3 セマフォ

セマフォは主に共有資源の管理に使用します。資源の名前（セマフォ・ハンドル名）はユーザ定義ファイルUDEF_a.hの中で定義します。1セマフォ当りの資源管理数は1個固定です。

・UDEF_a.hでの定義例

```
vDefine_sema    hsTimer_ch0     ;/* タイマch0の資源管理フラグ */
vDefine_sema    hsTimer_ch1     ;/* タイマch1の資源管理フラグ */
```

・プログラム中でのハンドル名の使用方法

C言語ではvSEMA_IDマクロでextern宣言してから使用します。

宣言文の例	extern　vSEMA_ID(hsTimer_ch0);
システム・コールでの使用例	vRefer_sema(hsTimer_ch0);

アセンブラでは単純にextrn宣言して使用します。

宣言文の例	extrn　_hsTimer_ch0
システム・コールでの使用例	vRefer_sema　_hsTimer_ch0

（注意：アセンブラでは，ハンドル名，変数名，関数名の先頭に_が付きます）

・資源の獲得

vPolling_semaにより資源の獲得をこころみます。獲得に成功した場合，成功ステータスが返ります。獲得に失敗した場合，失敗ステータスが返ります。

・資源の開放

vSignal_semaにより開放します。他のタスクが使用中であっても開放してしまいます。

・資源の状態参照

vRefer_semaにより確認できます。

2. 4　イベント・フラグ

本OSのイベント・フラグは1bitの情報を伝達するために使います。単なるフラグと異なり，受け取りを行うと送り側タスクを自動的にウエイクアップします。また指定時間経っても受け取られない場合にも送り側タスクを自動的にウエイクアップします。

・UDEF_a.hでの定義例

```
vDefine_flag        hftCSI_EEP                  ;/* EEPROMアクセス用 */
```

・プログラム中でのハンドル名の使用方法

C言語ではvFLAG_IDマクロでextern宣言してから使用します。

宣言文の例　　extern　vFLAG_ID(hftCSI_EEP);

システム・コールでの使用例　　vRefer_flag(hftCSI_EEP);

アセンブラでは単純にextrn宣言して使用します。

宣言文の例　　extrn　_hftCSI_EEP

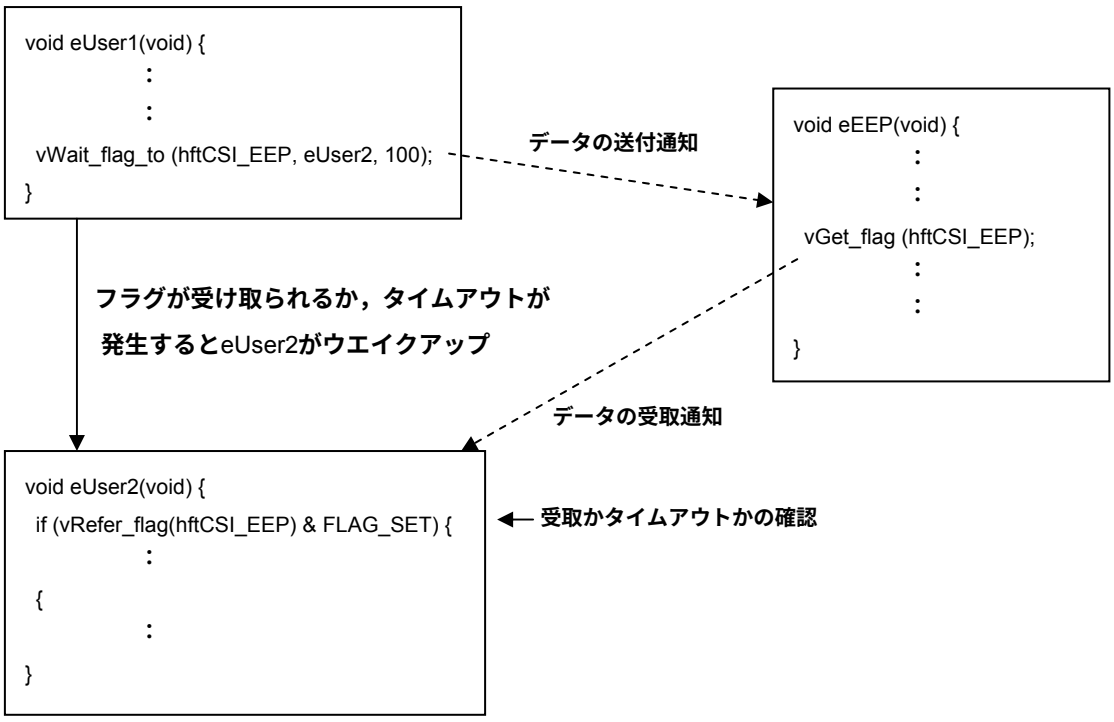
システム・コールでの使用例　　vRefer_flag　_hftCSI_EEP

（注意：アセンブラでは，ハンドル名，変数名，関数名の先頭に_が付きます）

・イベント・フラグの使用例（図2－3）

イベントフラグは1bitの情報伝達であるため、通常は付帯するデータをグローバル変数で定義します。送り側タスクで付帯情報を格納した後にvWait_flag_toを発行すると送り側タスクは待ち状態になります。受取側でvGet_flagを発行することにより送り側タスクを自動的に実行可能状態にします。

図2－3　イベント・フラグの使用例



・イベント・フラグの制御データ構造（参考）

イベント・フラグは，1定義あたり2バイトのRAMを使用します。具体的変数名は後述の「システム定義変数・定数」を参照としてください。

1バイト目bit7 ：送り側がvWait_flag_toを発行すると‘1’になり，受取側でvGet_flagを実行すると‘0’になります。タイムアウトではクリアされません。

1バイト目bit6 ：タイムアウト待ち状態で‘1’になります。タイムアウトにより‘0’になります。

1バイト目bit5-0 ：送り側タスクのＩＤ番号が格納されます。

２バイト目 ：タイムアウトカウンタです。基本周期ごとにデクリメントし，ゼロになったらタイムアウトと判定します。

2.5 パイプ

データを送り側タスクと受け側タスクの間で非同期で連続して転送したい場合にパイプを使用します。データの単位は1バイトです。

・UDEF_a.hでの定義方法

パイプ名（ハンドル名）とパイプの長さ（データ格納の最大数）をvDefine_pipeマクロで定義します。

記述方法がMCUの種類により異なりますので，定義例はUDEF_a.hのコメントを参照して下さい。

・プログラム中でのハンドル名の使用方法

C言語ではvPIPE_IDマクロでextern宣言してから使用します。

宣言文の例	extern　vPIPE_ID (hpUART1_tx);
システム・コールでの使用例	vRefer_pipe(hpUART1_tx);

アセンブラでは単純にextrn宣言して使用します。

宣言文の例	extrn　_hpUART1_tx
システム・コールでの使用例	vRefer_pipe　_hpUART1_tx

（注意：アセンブラでは，ハンドル名，変数名，関数名の先頭に_が付きます）

・データを送る側ではvPut_pipeを使用します。ただしvPut_pipeの返り値が‘偽’の場合はパイプ満杯のためデータが送れなかったことを示します。パイプが満杯にならない限りデータを連続で送り続けることが出来ます。

・データを受け取る側ではvGet_pipeを使用します。ただしデータがなかった場合は-1が返ってきます。パイプに格納されているデータを連続で受取り続けることが出来ます。

2.6 時間管理

本OSでは基本周期（標準10 ms）を単位として時間管理をしています。実際には次の二つのカウンタを用いています（具体的な変数名は「システム定義定数・変数」を参照してください。）。

(1) システム時刻カウンタ（32ビット）：基本周期を1単位とするカウンタです。リセット後（初期設定後）を0とします。システム・コールによって値の参照や設定を行うことができます。

(2) 分周カウンタ（8ビット）：基本周期の整数分の1を単位とするカウンタです。何分周するかはアプリケーションごとに決めます。

・システム時刻の参照，設定

参照はvGet_tme，設定はvSet_timeで行います。

・カウントアップ

本OSではシステム時刻カウンタおよび分周カウンタは自動的にはカウントアップしません。基本周期の整数分の1を周期とするインターバル・タイマによるハンドラを作成し，その中からカウントアップのためのシステム・コールvCount_timeを発行する必要があります。

・基本周期タスク（システム予約タスク）

本OSでは時間管理を全く行わないアプリケーションを除いて，基本周期ごとに1回だけ実行する基本周期タスクを必須としています。タスク名BTimer，ハンドル名hBTimer，関数名eBTimerとしています。この関数の中から時間にかかわるタスク制御を行うシステム・コールvControl_timeを発行する必要があります。

・タスクの暴走検出

本ＯＳでは，システム予約の基本周期タスクが実行可能になってから１基本周期が経過しても基本周期タスクが実行されない場合は，タスクが暴走していると判断して，その時実行中のタスクを強制終了します。

暴走検出はシステム・コールvCount_timeの中で行っています。

・ウォッチドッグ・タイマによるリセット

本ＯＳのスケジューラは，ラウンドロビンが一巡するごとにウォッチドッグ・タイマをクリアします。ウォッチドッグ・タイマのオーバフロー時間は，タスクの暴走検出よりも長くする必要があるため，基本周期の3倍以上に設定します。

なお，デフォルトではhalt, stop時はウォッチドッグ・タイマが停止する設定になっているため，スタンバイ状態の時間が長くてもウォッチドッグ・リセットはかかりません。

2. 7　システム・コール

システム・コールの一覧を表2－1に示します。以下，システム・コールの詳細説明の読み方について解説します。

(1) 記述形式

C言語：関数のプロトタイプを示します。

必須インクルード・ファイルKernel_c.hの中でマクロ定義およびプロトタイプ宣言を行っているため，ユーザがあらためて宣言する必要はありません。

引数としてハンドル名，関数名を記述する場合は型を明示していません。自動的に型キャストが行なわれます。

注意　複数ステートメントでマクロ定義されている関数もあるため，if文，while文の実行ステートメントとして記述する場合は必ず{}内に記述してください。

良い例　if (xxxx) { vSleep_task(); }

悪い例　if (xxxx) vSleep_task();
else　xxxxx;　←エラーになる。

ASM：アセンブラのマクロ記述形式を示します。

必須インクルード・ファイルKernel_a.hの中でマクロ定義してあります。引数としては多くの場合，ハンドル名かエントリ・アドレス名しか記述できません。レジスタ格納値を引数として使いたい場合は，カーネル・ヘッダ（Kernel_a.h）のマクロ定義を参照して前後処理を付加したサブルーチン・コール記述を行います。

レジスタの使用方法はC言語と混在できるように次のように定めてあります。

引　数：第一引数のみAX（1～2バイト時），BC（4バイト時上位）とし，第二引数以降はスタックに積みます（最後の引数から積む）。

返り値：CY（boolean型時），BC（1～2 バイト時），DE（4バイト時上位）。

保　存：HLレジスタのみ内容を保持します。他のレジスタは破壊される可能性があります。

(2) 機能

機能の詳細を記載します。

(3) 注意事項

システム・コール内部で割り込みの許可や禁止を行っているかなど，使用上の注意事項を記載してあります。

なお，割り込み許可を行うシステム・コールはvControl_timeのみです。それ以外で割り込み禁止区間を持つシステム・コールは割り込み許可／不許可状態を維持します（割り込み禁止状態で呼ばれた場合は割り込み禁止区間が終わっても割り込み禁止を維持します）。

表2－1　システム・コール一覧

区分	コール名	内容
タスク制御	vStart_task	他タスクの起動または強制的遷移。
	vTerminate_task	他タスクの強制終了。
	vExit_task[注1]	自タスクの終了。
	vSuspend_task	他タスクを待ち状態にする。
	vSleep_task[注1]	自タスクを中断して待ち状態にする。
	vWakeup_task	待ち状態の他タスクを実行可能にする（起床）。
	return[注1], ret[注1]	現在の状態関数へ再遷移（return：C言語，ret：アセンブラ）。
	vDelay_task[注1]	現在の状態関数へ再遷移（基本周期後）。
	vTrans_task[注1]	次の状態関数へ遷移。
	vTrans_task_delay[注1]	次の状態関数へ遷移（基本周期後）。
	vActive_task	ユーザ定義フラグをセットする。
	vRetire_task	ユーザ定義フラグをクリアする。
	vRefer_task	タスクの状態参照。
	vGet_task_id	タスクIDの取得。
資源管理	vPolling_sema	資源の獲得要求（返り値が要求結果）。
	vSignal_sema	資源の開放。
	vRefer_sema	資源の状態参照。
イベント制御	vWait_flag_to[注1]	イベント・フラグをセットし，待ち状態になる（タイムアウト機能付き）。
	vSet_flag	イベント・フラグをセットする。
	vClear_flag	イベント・フラグをクリアする。
	vRefer_flag	イベント・フラグの状態参照。
	vGet_flag	イベント・フラグがセットされている場合はクリアし，要求元タスクを実行可能にする。
パイプ制御	vInit_pipe	パイプの初期化。
	vRefer_pipe	パイプの空きバイト数を参照。
	vRefer_pipe_empty	パイプが空かどうかを参照。
	vPut_pipe	パイプへのデータ書き込み。
	vGet_pipe	パイプからのデータ読み出し。
時間管理	vSet_time	システム時刻の設定。
	vGet_time	システム時刻の取得。
	vCount_time	システム時刻のカウントアップ。
	vControl_time	基本周期タスクの必須処理。

注1.　トップ階層の関数でのみ有効

2. 7. 1　vStart_task（他タスクの起動，強制遷移）

(1) 記述形式

C言語：void vStart_task(hTask, eTask)

hTask：起動するタスク・ハンドル名またはタスクID番号

eTask：関数名

ASM：vStart_task　_hTask, _eTask

_hTask：起動するタスク・ハンドル名またはタスクID番号（定数）

_eTask：エントリ・アドレス名（ラベル名，定数）

(2) 機能

・他タスクを実行可能状態にします。

・他タスクの実行開始関数を強制的にehTaskにします。

(3) 注意事項

・内部に割り込み禁止期間があります。

・休止状態でないタスクに対して使うのは，強制終了のために資源開放の関数を指定する場合のみにしてください。それ以外の場合，そのタスクで確保していた資源が開放されずに重大な障害を引き起こす可能性があります。

2. 7. 2　vTerminate_task（他タスクの強制終了）

(1) 記述形式

C言語：void vTerminate_task (hTask)

hTask：強制終了したいタスク・ハンドル名またはタスクID番号

ASM：vTerminate_task　_hTask

_hTask：強制終了したいタスク・ハンドル名またはタスクID番号（定数）

(2) 機能

・他タスクを強制的に休止状態にします。

(3) 注意事項

・内部に割り込み禁止期間があります。

・指定したタスクで資源を確保していた場合，資源が開放されずに重大な障害を引き起こす可能性があります。資源を確保しているタスクを強制停止する場合は，vSignal_semaにより資源も強制的に開放する必要があります。

2. 7. 3　vExit_task（自タスクの終了）

(1) 記述形式

C言語：void vExit_task(void)

ASM：vStart_Exit

(2) 機能

・自タスクを休止状態にします。

(3) 注意事項

・本システム・コールはトップ階層の関数でのみ有効です。下位関数内から呼び出した時の動作は保証されません。

・内部に割り込み禁止期間があります。

2. 7. 4　vSuspend_task（他タスクを待ち状態に変更）

（1）記述形式

C言語：void vSuspend_task(hTask)

hTask：タスク・ハンドル名またはタスクID番号

ASM：vSuspend_task　_hTask

_hTask：タスク・ハンドル名またはタスクID番号（定数）

（2）機能

・他タスクを強制的に待ち状態にします。

（3）注意事項

・内部に割り込み禁止期間があります。

2. 7. 5　vSleep_task（自タスクを待ち状態に変更）

（1）記述形式

C言語：void vSleep_task(void)

ASM：vSleep_task

（2）機能

・自タスクを待ち状態にします。

・他タスクによりvWakeup_taskで待ち解除された場合は自関数（の最初）へ戻ります。

もし異なる関数から再開したい場合は、

vStart_task(自分のタスク・ハンドル名, 再開関数名);

vSleep_task();

の順に実行します。

（3）注意事項

・本システム・コールはトップ階層の関数でのみ有効です。下位関数内から呼び出した時の動作は保証されません。

・内部に割り込み禁止期間があります。

2. 7. 6　vWakeup_task（他タスクを実行可能に変更）

（1）記述形式

C言語：void vWakeup_task(hTask)

hTask：タスク・ハンドル名またはタスクID番号

ASM：vWakeup_task　_hTask

_hTask：タスク・ハンドル名またはタスクID番号（定数）

（2）機能

・他タスクを実行可能に変更します。起床するとも言います。

（3）注意事項

・内部に割り込み禁止期間があります。

2. 7. 7　return, ret（現在の状態関数へ再遷移）

（1）記述形式

C言語：return

ASM：ret

（2）機能

・現在の状態関数へ再遷移します。遷移にかかる時間はスケジューラでラウンドロビンが一巡する時間です。タスクの数が多くなるほど時間がかかります。

（3）注意事項

・本記述はトップ階層の関数でのみシステム・コールとして扱われます。下位関数内に記述した場合は，通常通りのreturn文，ret命令と解釈されます。

2. 7. 8　vDelay_task（現在の状態関数へ再遷移（基本周期後））

（1）記述形式

C言語：vDelay_task(void)

ASM：vDelay_task

（2）機能

・現在の状態関数へ再遷移します。遷移する前に，次の基本周期が始まるまで自タスクを待ち状態にします。消費電力を低減する機能のため，待ち状態時間は一定していません。例えば基本周期が10msの場合，待ち状態時間は通常は0～10 ms（最悪0～20 ms）の間で変動します。

（3）注意事項

・本システム・コールはトップ階層の関数でのみ有効です。下位関数内から呼び出した時の動作は保証されません。

・内部に割り込み禁止期間があります。

・本ＯＳのシステム予約タスクである基本周期タスクが起動されている必要があります。基本周期タスクが休止状態の場合は，vDelay_taskを発行したタスクは待ち状態の解除が自動的に行われません。

2. 7. 9　vTrans_task（次の状態関数へ遷移）

（1）記述形式

C言語：void vTrans_task_f(eEntry)

eEntry：関数名（通常はタスク内の遷移先関数名）

ASM：vTrans_task_f _ehEntry

_eEntry：遷移先アドレス（通常はタスク内のサブルーチンのエントリ・ アドレス（ラベル名、定数））

（2）機能

- タスク内の実行関数を切り替えます。本ＯＳではタスクの機能を状態別の関数（サブルーチン）群で構成することを想定しています。関数間を移動（遷移）するために本システム・コールを使用します。
- 遷移にかかる時間はスケジューラでラウンドロビンが一巡する時間です。タスクの数が多くなるほど時間がかかります。

（3）注意事項

- 本システム・コールはトップ階層の関数でのみ有効です。下位関数内から呼び出した時の動作は保証されません。
- 内部に割り込み禁止期間があります。

2. 7. 10　vTrans_task_delay（次の状態関数へ遷移（基本周期後））

（1）記述形式

C言語：void vTrans_task_delay(eEntry)

eEntry：関数名（通常はタスク内の遷移先関数名）

ASM：vTrans_task_delay _ehEntry

_eEntry：遷移先アドレス（通常はタスク内のサブルーチンのエントリ・ アドレス）（ラベル名，定数）

（2）機能

- タスク内の実行関数を切り替えます。vTrans_taskと異なり、次の基本周期が始まるまで待ち状態になります。

（3）注意事項

- 本システム・コールはトップ階層の関数でのみ有効です。下位関数内から呼び出した時の動作は保証されません。
- 内部に割り込み禁止期間があります。
- 本ＯＳのシステム予約タスクである基本周期タスクが起動されている必要があります。基本周期タスクが休止状態の場合は，vTrans_task_delay を呼び出したタスクは待ち状態の解除が自動的に行われません。

2. 7. 11　vActive_task（ユーザ定義フラグのセット）

（1）記述形式

C言語：void vActive_task(hTask)

hTask：タスク・ハンドル名またはタスクID番号

ASM：vActive_task _hTask

_hTask：タスク・ハンドル名またはタスクID番号（定数）

（2）機能

- タスク制御データTCB1のユーザ定義フラグを‘1’にします。
- この機能をどのように使うかはアプリケーション依存ですが，例えば他のタスクに対してデータ・インタフェースやコマンド受け付けなどの動作が可能になったことや，資源利用中を示すために用いることができます。

（3）注意事項

- 内部に割り込み禁止期間があります。

2. 7. 12　vRetire_task（ユーザ定義フラグのクリア）

（1）記述形式

C言語：void vRetire_task(hTask)

hTask：タスク・ハンドル名またはタスクID番号

ASM：vRetire_task _hTask

_hTask：タスク・ハンドル名またはタスクID番号（定数）

（2）機能

- タスク制御データTCB1のユーザ定義フラグを‘0’にします。
- この機能をどのように使うかはアプリケーション依存ですが，例えば他のタスクに対してデータ・インタフェースやコマンド受け付けなどの動作ができないことや，資源開放したことを示すために用いることができます。

（3）注意事項

- 内部に割り込み禁止期間があります。

2. 7. 13　vRefer_task（タスクの状態参照）

(1) 記述形式

C言語：unsigned char vRefer_task(hTask)

hTask：タスク・ハンドル名またはタスクID番号

ASM：vRefer_task _hTask

_hTask：タスク・ハンドル名またはタスクID番号（定数）

(2) 機能

・タスクがどのような状態かの情報を取得します。

（返り値＆TASK_START）== 0：タスクは休止状態です。

（返り値＆TASK_READY）!= 0：タスクは実行可能状態です。

（返り値＆TASK_DELAY）!= 0：タスクは次の基本周期で実行可能状態になります。

（返り値＆TASK_ACTIVE）!= 0：ユーザ定義フラグがセットされています。

（返り値＆TASK_ACTIVE）== 0：ユーザ定義フラグがクリアされています。

(3) 注意事項

・対象タスクの状態変更を行うハンドラがある場合は，vRefer_taskの実行結果が実際と一致しなくなる可能性があります。支障がある場合は割込み禁止状態で本システム・コールを使用します。

2. 7. 14　vGet_task_id（実行中タスクのID参照）

(1) 記述形式

C言語：unsigned char vGet_task_id(void)

ASM：vGet_task_id

(2) 機能

・実行中タスクのタスクID情報を取得します。

(3) 注意事項

・ハンドラ内で使用した場合，スケジューラで実行可能かどうか判定中のタスクIDが返ってきたり，最大タスクID＋1（halt可能か判定中）が返ってくる場合があります。必ずしもタスク実行中とは限りません。

2. 7. 15　vPolling_sema（資源獲得要求）

(1) 記述形式

C言語：boolean vPolling_sema(hsSema)

hsSema：セマフォ・ハンドル名またはID番号

ASM：vPolling_sema _hsSema

_hsSema：セマフォ・ハンドル名またはID番号（定数）

(2) 機能

・指定したセマフォで管理する資源の獲得を試みます。

返り値 == 偽　　：獲得できなかった

返り値 != 偽　　：獲得できた

(3) 注意事項

・内部に割り込み禁止期間があります。

2. 7. 16　vSignal_sema（資源解放）

(1) 記述形式

C言語：void vSignal_sema(hsSema)

hsSema：セマフォ・ハンドル名またはID番号

ASM：vSignal_sema _hsSema

_hsSema：セマフォ・ハンドル名またはID番号（定数）

(2) 機能

・指定したセマフォで管理する資源を開放します。

(3) 注意事項

・内部に割り込み禁止期間があります。

・他タスクが獲得した資源でも開放してしまいます。他タスクを強制停止させる時以外は他タスクで獲得した資源を開放しないでください。

2. 7. 17　vRefer_sema（資源状態参照）

(1) 記述形式

C言語：boolean vRefer_sema(hsSema)

hsSema：セマフォ・ハンドル名またはID番号

ASM：vRefer_sema _hsSema

_hsSema：セマフォ・ハンドル名またはID番号（定数）

(2) 機能

・指定したセマフォで管理する資源の情報を取得します。

返り値 == 偽　　：空き状態

返り値 != 偽　　：使用中

(3) 注意事項

・対象セマフォの状態変更を行うハンドラがある場合は，vRefer_semaの実行結果が実際と一致しなくなる可能性があります。支障がある場合は割込み禁止状態で本システム・コールを使用します。

2. 7. 18　vWait_flag_to（タイムアウト機能付きのイベント・フラグ解除待ち）

(1) 記述形式

C言語：void vWait_flag_to(hfFlag, ehNext, unsigned char nTime)

hfFlag：フラグ・ハンドル名またはID番号

ehEntry：関数名（通常はタスク内の遷移先関数名）

nTime：タイムアウト時間を1～255の数値で指定します。数値は基本周期単位です。

ASM：vWait_flag_to _hfFlag, _ehNext, nTime_rp

_hfFlag：フラグ・ハンドル名またはID番号（定数）

_ehEntry：遷移先アドレス（通常はタスク内のサブルーチンのエントリ・アドレス）
（ラベル名，定数）

_nTime：タイムアウト時間を格納したペア・レジスタ名を指定します。
格納値は基本周期を単位とする1～255の数値です。

(2) 機能

- イベント・フラグをセットして自タスクを待ち状態にします。他タスクがイベント・フラグを取得したら遷移先関数で起床します。
- 他タスクがイベント・フラグを取得しないうちにタイムアウトした場合も遷移先関数で起床します。
- 起床した要因がフラグ取得かタイムアウトかを判定するにはvRefer_flagを使用します。

(3) 注意事項

- 本システム・コールはトップ階層の関数でのみ有効です。下位関数内から呼び出した時の動作は保証されません。
- 内部に割り込み禁止期間があります。
- すでにフラグセット済であっても同じ動作をします。もしセット済の場合はフラグの解除待ちをしたくない場合は，事前にvRefer_flagでフラグの状態を確認します。

2. 7. 19　vSet_flag（イベント・フラグのセット）

(1) 記述形式

C言語：void vCancel_flag (hfFlag)

hfFlag：フラグ・ハンドル名またはID番号

ASM：vCancel_flag _hfFlag

_hfFlag：フラグ・ハンドル名またはID番号（定数）

(2) 機能

- イベント・フラグをセットします。

(3) 注意事項

- すでにイベント・フラグがセット済であっても同じ動作をします。もしvWait_flag_toによってイベント・フラグの解除待ちであった場合，タイムアウトの指定が取り消されてしまいます。

2. 7. 20　vClear_flag（イベント・フラグのクリア）

（1）記述形式

C言語：void vCancel_flag (hfFlag)

hfFlag：フラグ・ハンドル名またはID番号

ASM：vCancel_flag _hfFlag

_hfFlag：フラグ・ハンドル名またはID番号（定数）

（2）機能

・イベント・フラグ解除待ちを取消します。

・タイムアウト後に取消したい場合や，イベント・フラグを用いたハンドシェークの場合に使用します。

（3）注意事項

・他タスクが要求したイベント・フラグ解除待ちも取り消されます。他タスクを強制停止する場合以外は他タスクが要求したイベント・フラグ解除待ちを解除しないで下さい。

2. 7. 21　vRefer_flag（フラグの参照）

（1）記述形式

C言語：unsigned char vRefer_flag(hfFlag)

hfFlag：フラグ・ハンドル名またはID番号

ASM：vRefer_flag _hfFlag

_hfFlag：フラグ・ハンドル名またはID番号（定数）

（2）機能

・フラグの状態情報を取得します。

（返り値＆FLAG_SET）!= 0　　：セット状態。

（返り値＆FLAG_TO） != 0　　：タイムアウト待ち状態。

（3）注意事項

・対象イベント・フラグの状態変更を行うハンドラがある場合は、vRefer_flagの実行結果が実際と一致しなくなる可能性があります。支障がある場合は割込み禁止状態で本システム・コールを使用します。

2. 7. 22　vGet_flag（フラグ取得）

（1）記述形式

C言語：boolean vGet_flag(hfFlag)

hfFlag：フラグ・ハンドル名またはID番号

ASM：vGet_flag _hfFlag

_hfFlag：フラグ・ハンドル名またはID番号（定数）

（2）機能

・フラグがセットされている場合（解除待ち状態）、フラグをクリアした上で解除待ちのタスクを実行可能状態にします。

返り値 == 偽　　：クリア状態であった（解除待ちではなかった）

返り値 != 偽　　：セット状態であった（待ち状態の解除を実行した）

（3）注意事項

・内部に割り込み禁止期間があります。

2. 7. 23　vInit_pipe（パイプの初期化）

(1) 記述形式

C言語：void vInit_pipe(hpPipe)

hpPipe：パイプ・ハンドル名またはID番号

ASM：vInit_pipe _hpPipe

_hpPipe：パイプ・ハンドル名またはID番号（定数）

(2) 機能

・パイプの初期化をします。格納されていたデータは破棄されます。

(3) 注意事項

・実際にはポインタのみ初期化するため，RAM上には初期化前のデータが残ります。

・パイプはOS起動時に自動的に初期化されます。

2. 7. 24　vRefer_pipe（パイプの空きバイト数の参照）

(1) 記述形式

C言語：unsigned short vRefer_pipe(hpPipe)

hpPipe：パイプ・ハンドル名またはID番号

ASM：vRefer_pipe _hpPipe

_hpPipe：パイプ・ハンドル名またはID番号（定数）

(2) 機能

・後何バイト書き込みできるかの情報を指定パイプから取得します。

(3) 注意事項

・特にありません。

2. 7. 25　vRefer_pipe_empty（パイプが空かの判定）

(1) 記述形式

C言語：boolean vRefer_pipe_empty (hpPipe)

hpPipe：パイプ・ハンドル名またはID番号

ASM：vRefer_pipe_empty _hpPipe

_hpPipe：パイプ・ハンドル名またはID番号（定数）

(2) 機能

・パイプが空（データ無し）かどうかの情報を返します。返り値が‘真’なら空です。

(3) 注意事項

・特にありません。

2. 7. 26　vPut_pipe（パイプへのデータ書き込み）

（1）記述形式

C言語：boolean vPut_pipe(hpPipe, unsigned char data)

hpPipe：パイプ・ハンドル名またはID番号

data：書き込みデータ

ASM：vPut_pipe _hpPipe, data_rp

_hpPipe：パイプ・ハンドル名またはID番号（定数）

data_rp：書き込みデータを格納したペア・レジスタ名を指定します。

データはペア・レジスタの下位8bitに格納します。

（2）機能

・パイプへ1バイトのデータを書き込みます。

返り値 == 偽　　：パイプが満杯のため書き込めず。

返り値 != 偽　　：パイプに正常に書き込めた。

（3）注意事項

・特にありません

2. 7. 27　vGet_pipe（パイプからのデータ読み出し）

（1）記述形式

C言語：short vGet_pipe(hpPipe)

hpPipe：パイプ・ハンドル名またはID番号

ＡＳＭ：vGet_pipe _hpPipe

_hpPipe：パイプ・ハンドル名またはID番号（定数）

（2）機能

・パイプから1バイトのデータを読出します。

返り値 == -1：パイプが空のため読み出せず。

返り値 ＞ 0 ：下位8bitが読出しデータ。

（3）注意事項

・特にありません

2. 7. 28　vSet_time（システム時刻の設定）

（1）記述形式

C言語：void vSet_time(unsigned long nTime)

nTime：時刻データ。基本周期時間（標準10 ms）を1単位とします。

ASM：vSet_time

時刻データの下位16ビットをAX，上位16ビットをBCに格納してから呼び出します。

（2）機能

・システム時刻を設定します。

・設定値は基本周期時間（標準10ms）を1単位とします。

（3）注意事項

・内部に割り込み禁止期間があります。

・本OSではリセット後，スケジューラ起動直前に自動的に0にクリアします。

2. 7. 29　vGet_time（システム時刻の取得）

（1）記述形式

C言語：unsigned long vGet_time(void)

ASM：vGet_time

（2）機能

・システム時刻を取得します。返り値は，基本周期時間（標準10 ms）を1単位とする4バイトのカウント値です。アセンブラではBC（下位16ビット），DE（上位16ビット）に返り値が格納されます。

（3）注意事項

・内部に割り込み禁止期間があります。

2. 7. 30　vCount_time（システム時刻のカウント）

（1）記述形式

C言語：void vCount_time(int nCNT)

ASM：vCount_time nCNT

nCNT：分周数。基本周期を何分周しているかを指定します。例えば基本周期10 msに対してタイマ割り込み周期が1 msなら10を指定します。

（2）機能

・システム時刻のカウントおよび基本周期タスクの起床を行います。

・暴走タスクを検出した場合はそのタスクの強制停止も行います。

・タイマ割り込みハンドラ内から呼び出してください。

（3）注意事項

・暴走タスクを強制終了した場合は処理がスケジューラに戻ります。本関数を呼び出したハンドラには戻りません。

2. 7. 31　vControl_time（基本周期処理）

（1）記述形式

C言語：void vControl_time(void)

ASM：vControl_time

（2）機能

・基本周期の先頭でOS固有の処理（Delayタスクの起床，イベントフラグのタイムアウト・チェック）を行います。

・基本周期タスクから呼び出してください。

（3）注意事項

・内部で割り込み禁止および許可を行っています。割り込み許可状態で関数から戻ります。

2. 8　割り込みハンドラ

2. 8. 1　ハンドラからのシステム・コール

本ＯＳでは割り込みハンドラ専用のシステム・コールを定義していません。もしシステム・コールを使用する場合は，タスク処理との競合が起きないことを確認する必要があります。代表例を次に示します。

（1）ハンドラからのvWakeup_task

タスク側でvSleep_taskを実行する場合は，タスク動作中にvWakeup_taskが発行されてもタスク最後のvSleep_taskが実行されて待ち状態になってしまいます。これを避けるには次のように記述します。

```
void eTest(void) {
    vSuspend_task(eTest);     …  最初に自分を待ち状態へ変えます。
        :
        :
    return;                   …  関数実行中にハンドラでvWakeup_task(eTest);が発行されている場
}                                 合は自タスクへ再遷移します。そうでなければ待ち状態になります。
```

（2）パイプの書き込み，読み出し

ハンドラとタスクで読み書きが異なる場合は同じパイプをアクセスしても干渉しません。同じ動作を行う場合は競合が起きますので，この場合はタスクで割り込み禁止状態にしてからパイプをアクセスします。

2.9 システム定義定数・変数

システム定義定数・変数の一覧を表2－2に示します。これらは次のファイルをインクルードすると使えます。

Kernel_c.h（C言語用），Kernel_a.h（アセンブラ用）

定数名　　：C言語，アセンブラともそのままの名称で使います。

シンボル名：C言語ではそのままの名称で使います。大文字のみのシンボルは使用時にunsigned charなどにキャストしてください。gcはunsigned char型, giはunsigned short型, glはunsigned long型です。アセンブラでは名称の先頭に_（アンダスコア）を付けます。

表2－2　システム定義定数・変数一覧

区分		名称	内容
タスク	シンボル	TASKID_MAX	タスク登録数。
	シンボル	gcKernel_TID	実行中タスクのＩＤ。1バイト変数。
	シンボル	gcKernel_TCB1	タスク制御データ1。1バイト配列。
	シンボル	giKernel_TCB2	タスク制御データ2。2バイト配列。
	シンボル	gcKernel_TCB1_hBTimer	システム予約タスクBTimerの制御データ1アドレス。
	定数	TASK_START_BP	スタート・フラグのビット位置。
	定数	TASK_START	同上のマスク値。
	定数	TASK_READY_BP	実行可能フラグのビット位置。
	定数	TASK_READY	同上のマスク値。
	定数	TASK_DELAY_BP	ディレー・フラグのビット位置。
	定数	TASK_DELAY	同上のマスク値。
	定数	TASK_ACTIVE_BP	ユーザ定義フラグのビット位置。
	定数	TASK_ACTIVE	同上のマスク値。
	定数	TASK_ADDR_MASK	実行アドレス上位4ビットのマスク値。
	定数	TASK_CALL	タスク呼び出し可能条件。
	定数	TASK_SLEEP	待ち状態マスク値。
セマフォ	シンボル	SEMAID_MAX	セマフォ登録数。
	シンボル	SEMAID_BYTE_MAX	セマフォ管理バイト数。
	シンボル	gcKernel_SCB	セマフォ。1バイト配列（8セマフォ単位）。
イベント・フラグ	シンボル	FLAGID_MAX	イベント・フラグ登録数。
	シンボル	gcKernel_FCB	イベント・フラグ制御データ。1バイト配列（1イベント・フラグ2要素使用）。
	定数	FLAG_SET_BIT	受け取り待ちフラグのビット位置。
	定数	FLAG_SET	同上のマスク値。
	定数	FLAG_TO_BIT	タイムアウト・フラグのビット位置。
	定数	FLAG_TO	同上のマスク値。
	定数	FLAG_WAIT_TO	タイムアウト・カウント条件。
	定数	FLAG_WAKE_ID	要求元タスクIDのマスク値。
パイプ	シンボル	PIPEID_MAX	パイプ登録数。
	シンボル	gcKernel_PCB	パイプ制御データ。1バイト配列（1パイプ2要素使用）。
	シンボル	giKernel_PIB	パイプ属性データ。2バイト配列（1パイプ2要素使用，ROM）。
時間	シンボル	glKernel_time	システム時刻カウンタ。4バイト変数。
	シンボル	gcKernel_tsub	分周カウンタ。1バイト変数。

第3章　使 い 方

この章では，コーディング方法，効果的設計方法，デバッグのヒントについて説明します。

3. 1　コーディング方法

タスクをコーディングする上で必要な知識として，全体的ファイル構成，作成・変更対象ファィル，タスク・プログラムの記述例について説明します。

3. 1. 1　ファイル構成

本OSはサンプル・タスクとともに提供するためファイルの数が多くなっています。主要なファイルの代表的構成は次の通りです。なお，名称の_xの部分は，C言語用は_c，アセンブラ用は_aになります。

カーネル

- Kernel_data_a.asm　・カーネルのデータ定義部です。通常は変更不要です。
  - UDEF_a.h　・ユーザ定義ファイルです。目的に合わせて変更します。
- Kernel_a.asm　・スケジューラおよびシステム・コールです。通常は変更不要です。
- Kernel_x.h　・共通ヘッダです。通常は変更不要です。システム・コールなどOS機能が定義されているため，タスク・プログラムは必ずこれをインクルードします。
- Kernel_init_c.c　・初期化処理です。ハードウェア依存部であるため，使用するMCUやアプリケーションに合わせてクロック設定やポート設定を変更します。
- Kernel_OPT_a.asm　・MCU依存のオプション・バイトです。必要に応じて修正します。

サンプル・タスク

- UDEF_port_x.h　・ポート名をリネームする場合のサンプルです。
- BTimer_c.c　・システム予約タスク（基本周期タスク）のサンプル・ソースです。
- BTimer_INT_a.asm　・基本周期タイマ・ハンドラのサンプル・ソースです。処理速度を上げるためにアセンブラで記述する場合の例です。BTimer_c.cでC言語記述することも可能です。
- Main_c.c　・メイン・タスクのサンプル・ソースです。初期化後，最初に実行されるタスクです。
- User_c.c　・ユーザ・タスクの雛型です。スタディとしてまず何か記述したい場合には，これだけを変更してビルドしてみるのが手っ取り早いでしょう。
- その他Xxxx_c.c　・各サンプル・タスクのC言語ソースです。
- その他Xxxx_a.asm　・アセンブラ記述のサンプル・ソースです（ハンドラを高速化する場合の例）。

これら以外にフォント関係のライブラリやヘッダ，共通処理をまとめたUDEF_func_x.xなどが含まれる場合があります。

リリース・ノートは，*.TXT（例えばSM05FV2j.TXT）です。

3. 1. 2 プログラムの作成手順

サンプル・ソース（R3.01以降）をベースとしたプログラムの作成手順は次の通りです。

★ (1) 準備

(a) 添付プロジェクト・ファイルからの起動を試みる

① 新規に作成したディレクトリに本OSを含むサンプル・ファイル一式をコピーします。

② この中のトップ階層にあるプロジェクト・ファイル（*.prw）をクリックして開きます（フリー・ツール使用時は*_F.prwを開きます）。デバイス・ファイルは事前に登録が必要です。

③ インストールされているツールのリビジョンと合わないため，リビジョン選択を要求されます。ここでは，OKを押した後，いったんキャンセルを押します。

④ “プロジェクト→プロジェクト設定→ツールバージョン設定→詳細設定”により，ツールの選択を行います。これでビルドができるはずです。正常にビルドできない場合は，以下の手順でプロジェクト・ファイルを作り直します。

(b) プロジェクト・ファイルの作り直し

① PM＋（プロジェクト・マネージャ）で，“ファイル→ワークスペースの新規作成”を選び，サンプル・ファイルをコピーしたディレクトリをワークスペースとして指定します。

② ソース・ファイル選択を行います。ファイル名の最後が_c.c，_a.asmのものを指定します。ただし，_Fの付いたファイル名はフリー版コンパイラ用ですので，ボードのユーザーズ・マニュアルあるいはリリース・ノートの注意事項に従ってください。

③ サンプル・ファイルにライブラリ（.lib）やリンク・ディレクティブ（.dr）が含まれている場合は“プロジェクト関連ファイル”に登録します。

④ コンパイラのオプション設定を行います。

- プリプロセッサ→定義マクロ：通常動作時は指定無しまたはKOS_CODE_TYPE=0を指定します。測定機能（5章参照）を追加するときはKOS_CODE_TYPE=1を指定します。
- プリプロセッサ→インクルード・ファイル・パス：ヘッダが格納されているディレクトリを指定します。
- メモリ・モデル：ノーマル（“スタティック”にチェック無し）を選択します。
- 出力：“アセンブラ・ソース・モジュール・ファイルの出力”にチェックを付けます。

⑤ アセンブラのオプション設定を行います。

- その他→インクルード・ファイル・パス：ヘッダがあるディレクトリを指定します。
- その他→シンボル定義：通常動作時はKOS_CODE_TYPE=0を指定します（省略不可）。測定機能（5章参照）を追加するときはKOS_CODE_TYPE=1を指定します。

⑥ リンカのオプション設定（出力1）を行います。

- 出力ファイル名を必要に応じて付けます。
- セキュリティIDは，とりあえずはFFFFFFFFFFFFFFFFFFFFで良いでしょう。
- その他：ディレクティブ・ファイル（*.dr）が同梱されていれば，それを指定します。

⑦ ビルドしてみます。ビルドが正常にできない場合は，オプション設定やソース・ファイル選択を確認してください。

(2) Kernel_init_c.cおよびKernel_OPT_a.asmの修正

クロック選択やポート初期化など基本的な設定を実際のシステムに合わせて修正します。

コンパイル・エラーが出ないのであれば，ポート設定は後日の修正でも構わないでしょう。

(3) ユーザ定義ファイルUDEF_a.hの修正

最低限必要なのは，システム予約のBTimerタスクと最初に実行されるMainタスクの登録です。この2つだけは削除したり変更したりしないようにします。

その他のタスクやセマフォ，イベント・フラグ，パイプは順次修正でも構いません。RAM不足でコンパイル・エラーが出ない限りはとりあえずサンプル記述をそのまま残しておいても構わないでしょう。

(4) システム予約タスクBTimerの修正または新規作成

BTimerは基本周期（標準10ms）を作るためのタスクです。関数名はe BTimerです。

最低限必要なのは下記です。

- eBTimerの中でインターバル・タイマの初期設定を行って起動します。インターバル時間は基本周期の整数分の1とします。
- タイマ割り込みの中でvCount_time(nCNT);を実行します。nCNT = 基本周期／インターバル時間。
- eBTimerの中でvControl_time();を実行します。eBTimerの最後にはvSleep_task();を記述します。
- サンプルによってはLEDとキーのダイナミック・スキャン処理などが含まれていますが，とりあえず支障なければ残しておいて後で削除するのでもよいでしょう。

(5) メイン・タスクMainの修正または新規作成

メイン・タスクは初期化後，最初に実行されるタスクです。通常は最初にシステム予約タスクBTimerを起動する記述を行います。

```
vStart_task(hBTimer, eBTimer);
```

さらに必要に応じてユーザ・タスクを含む各種タスクを起動する記述を行います。

(6) ユーザ・タスクの雛型修正または新規追加

ユーザ・タスクを作るには，まずは雛型（User_c.c）を修正するのが手っ取り早いでしょう。雛型は次の3つの関数から構成されています。

① eUser：初期化，資源獲得用の関数ですが，雛型では単に次の関数へ遷移します。

② eUser_main：主処理用の関数ですが，雛型ではvDelay_taskだけ実行しています。これは10 ms間隔でeUser_mainがOSから呼ばれることを意味します。

③ eUser_end：資源開放用の関数です。①でタイマやシリアル・インタフェースなどの資源を獲得している場合は，ここで必ず開放します。eUser_mainの処理が終わった場合にここに遷移するようにシステム・コールを発行します。　また，この関数はスタンバイ機能を管理しているタスクから，ユーザ・タスクの強制終了のために呼び出されることがあります。

雛型を使わずに新規に作成する場合は，（3）でタスク名を登録し，（5）でタスクを起動する記述を追加します。

3. 1. 3　タスク・プログラムの記述例

本OSのタスク・プログラムは，共通ヘッダ（Kernel_x.h）をインクルードすることだけが必須であり、それ以外は一般の関数（サブルーチン）の書き方に準じます（若干の注意事項については後述）。

(1) 共通ヘッダのインクルード方法

共通ヘッダにはシステム・コールを使うための定義や定数が記載されています。

(a) C言語の場合

C言語用共通ヘッダKernel_c.hを，プラグマ指令とライブラリ関数ヘッダの間でインクルードします。

```
#pragma XXXXXXX        …各種プラグマ指令（必要に応じて）。ただしSFR, DI, EI, HALT, STOP,
                          NOPは，Kernel_c.hで定義してあります。
#include "Kernel_c.h"  …必須
#include <stdio.h>
      :                ライブラリ関数のヘッダ（必要に応じて）
#include <string.h>
```

(b) アセンブラの場合

アセンブラ用共通ヘッダKernel_a.hをHEADER_MODE定義の直後でインクルードします。

```
$set       (HEADER_MODE)        …必須
$include   (Kernel_a.h)         …必須
```

(2) プログラミングおよびコーディング上の注意事項

①C言語，アセンブラ共通

下記のシステムコールは，タスクのトップ階層の関数（OSから呼び出される関数，サブルーチン）でしか使えません。

vExit_task, vSleep_task, vDelay_task, vTrans_task, vTrans_task_delay, vWait_flag_to

②C言語

システム・コールをif文，while文の実行ステートメントとして記述する場合は必ず{}内に記述してください（マクロ定義が複数ステートメントの場合があるため）。

```
良い例  if (xxxx) { vSleep_task(); }
悪い例  if (xxxx) vSleep_task();
        else      xxxxx;                ←エラーになる。
```

またシステム定義シンボルやハンドルをシステム・コールの引数以外で使用する場合は適当な型にキャストして使用してください。

```
例      i = (unsigned char) hCOM1_tx;
        vWakeup_task(i);
```

③アセンブラ

タスクのトップ階層プログラムでスタックにワーク領域を確保している場合，下記システム・コールを発行する前にスタックを開放してください（ret命令を実行するため）。

vExit_task, vSleep_task, vDelay_task, vTrans_task, vTrans_task_delay, vWait_flag_to

(3) C言語のタスク間インタフェース用ヘッダ

以下の説明はOS必須事項ではなくサンプル・タスクでの方式説明です。

サンプル・タスクのC言語ソースでは，他のタスクとのインタフェース定義を楽にするために変数定義と外部参照宣言（extern宣言）を１つのヘッダ・ファイルで切り替えて使うようにしています。

(a) 切り替え方法

共通ヘッダで定義されているvGlovalマクロを使用して，

vGlobal　unsigned char　　gcADC_in_ch;

のように記述します。vGlovalは変数を定義するタスクでは単なるコメントとして無視され，参照するタスクではextern宣言に置き変わるようにしてあります。

(b) インタフェース用ヘッダの実際の構成

変数だけでなく，参照用となる他の情報も一緒に記述しています。

・タスク・ハンドル名の参照宣言

例　extern　　vTASK_ID (当該タスクのハンドル名);

・インタフェース用に定めた構造体宣言，typedefなど

・公開関数のプロトタイプ宣言（実際にはタスク内の全ての関数をまとめて記載）

・外部公開する変数定義（定義の先頭にvGlobalを記述します）

例　vGlobal　unsigned char　　gcADC_in_ch;

(c) 変数定義するタスクでのインクルード方法

共通ヘッダやライブラリ関数のヘッダのインクルードの後に当該タスクのインタフェース用ヘッダをインクルードします。

例

```
#pragma XXXXXXX
#include "Kernel_c.h"
#include <stdio.h>
#include " XXX_c.h"        …  当該タスクのインタフェース用ヘッダ
```

(d) 参照するタスクでのインクルード方法

共通ヘッダ，ライブラリ関数および当該タスクのインタフェース用ヘッダの次に以下のように記述します。

```
#undef   vGlobal          …  vGlobal マクロをキャンセル
#define  vGlobal extern   …  vGlobal マクロをextern に変更
#include " YYY_c.h"       …  参照するタスクのインタフェース用ヘッダ
```

3. 1. 4　関数・変数の命名方法

以下の説明はOS必須事項ではなくサンプル・タスクでの例です。

この例では複数人数による開発の時に，関数や変数を誰がどこで定義しているのかがある程度把握しやすくするために下記の命名方法を採用しています。

接頭辞１文字目：スコープ（影響が及ぶ範囲）。

接頭辞２文字目：変数の型（ただしコンパイラの型チェックで十分と考える場合は省略）

接頭辞の次の文字：大文字

また，Cとアセンブラの言語混在（関数名や変数名の相互参照）のためアセンブラでは名称の先頭に _（アンダスコア)をつけます。

(1) 関数名，エントリ・ラベル名の接頭辞

v：システム・コール用です。マクロ形式も含みます。Kernel_x.x内で定義しています。

e：各タスク内の状態関数名に使用します。vTrans_task等の引数として記述する関数名です。

u：状態関数以外のユーザ定義関数で使用します。

(2) 変数名

①1文字目（影響範囲）

g：グローバル変数。主にタスク間でのデータ受け渡し用です。

t：タスク内のスタティック変数。

z：端子名，特殊機能レジスタ（SFR）をリネームする場合に使用します。

②2文字目（型）

b：boolean型，c：unsigned char型，d：char型，i：unsigned short型，j：short型，

l：unsigned long型，m：long型，p：ポインタ，s：構造体

(3) マクロ定義定数（アセンブラではSET,EQUでの定義）

全大文字の名称か, kまたはn（主としてサイズ）の接頭辞をつけます。

(4) その他

UDEF内で定義するハンドル名称（h：タスク・ハンドル名，hs：セマフォ・ハンドル名，

hft：イベント・フラグ・ハンドル名（タイムアウト付き），hp：パイプ・ハンドル名）

3. 2　効果的設計方法

本OSの特徴を生かしたアプリケーション・プログラム設計方法について説明します。

3. 2. 1　設計手順

大きく分けて次の4つのフェーズがあります。

（1）タスク分割

（2）タスク間インタフェース設計

（3）タイミング設計

（4）タスク設計

3. 2. 2　タスク分割

タスクとは独立したタイミングで機能を規定したい（動作して欲しい）単位です。実際には1つのCPUが時分割で動作しますが，マクロ的にはそれぞれのタスクは独立に動作しているように考えることが出来ます。

アプリケーション・プログラムをタスクへ分割するには，まず必要な機能を全て洗い出し，整理して独立に動作する単位にまとめます。この時，割込みハンドラは関連の深いタスクに従属すると考えてまとめるのが良いでしょう。

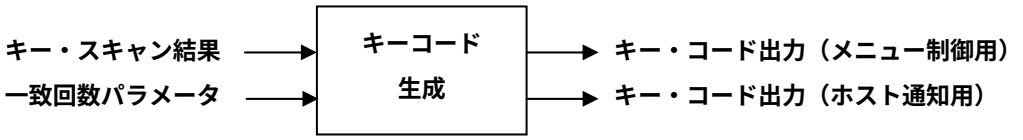
最終的にはタスク単位の機能リストを作るとよいでしょう。機能を箇条書きにしておくと，再分割や併合のカット＆ペーストが行いやすくなります。

3. 2. 3　タスク間インタフェース設計

このフェーズではタスク間のデータ受け渡し方法を定めます。1. 1. 4項で説明したノーティス，パケット，パイプのような受け渡し概念を定めて，タスク間でどのような種類のデータをどのような方法で受け渡すか決めていきます。

最初の段階では全体を俯瞰できる図を書いて，タスク間を結ぶ線の上に信号の種類・受け渡し方法を書いていきます。最終的にはタスク単位で，入力データ，出力データについて受け渡し方法と相手先タスク名を書いた図や表を作ります。受け渡しデータがコード化されている場合はコード表も作ります。

例：



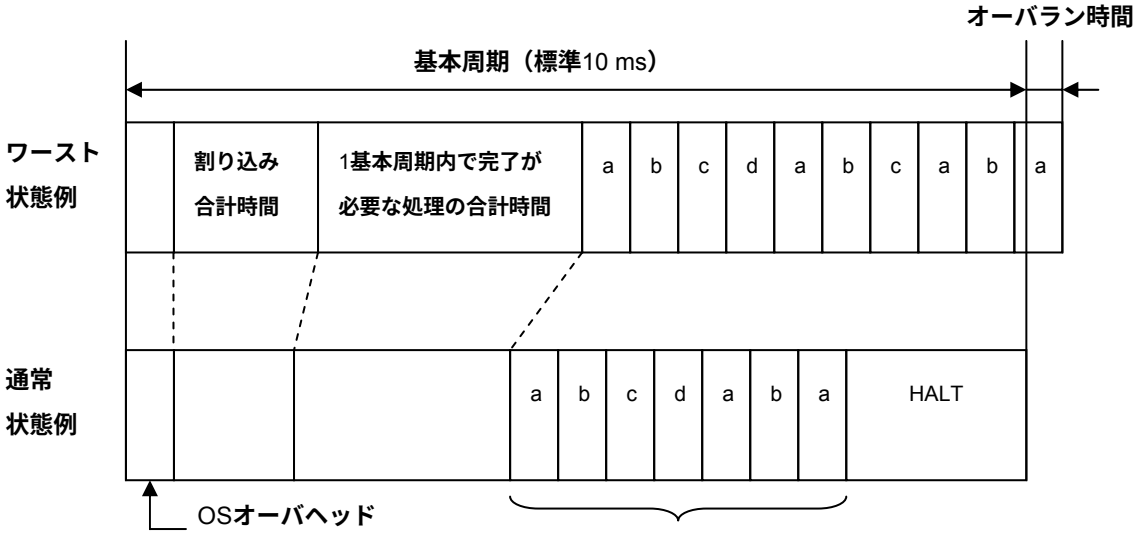
名称	種別，i／o	内容	関連タスク
キー・スキャン結果 gKey_scan	ノーティス (i)	チャタリング含むキー情報	スキャン
一致回数パラメータ gPara_key_cnt	ノーティス (i)	何回一致でキー情報を生成するかのパラメータ	システム制御
キー・コード出力（メニュー制御用） gKey_code	パケット (o)	チャタリング除去したキー情報をASCIIコードで出力	メニュー制御

3. 2. 4 タイミング設計

本OSは自動でタスク切り替えを行いません。従って適切なタイミング設計を行って，所定の時間内に状態遷移システム・コール（return, vTrans_task, vDelay_task, vTrans_task_delay, vSleep_task, vWait_flag_to）をタスクで発行する必要があります。このために以下のタイミングを決めたり確認したりします。

- 基本周期の選定（標準的には10 ms推奨）
- 1基本周期内で処理が完了しなければならないタスク，および割り込みのワースト合計値の確認
- 1基本周期内で処理が完了しなくて良いタスクの許容スライス時間の選定

図3－1 タイミング検討図



(1) 基本周期の選定

基本周期はユーザ・インタフェース（キー，ブリンク，ブザー等）の最小制御単位を考慮して10 msを標準としています。必ず10 msにしなければならないわけではありませんが，標準化されている方がソフトウエアの部品化，流通化をしやすくなります。

基本周期の時間精度はアプリケーションに依存します。精密な制御や時計機能を基本周期と連動したい場合は高精度が必要ですが，OSとしては特別精度を要求する条件はありません。

(2) 1基本周期内のワースト合計値確認

基本周期とCPUクロック（処理能力）が妥当であるかは，1基本周期内に完了しなければならない処理の合計時間が基本周期に対して余裕があるかどうかで判断します。手順は次の通りです。

① 1基本周期内に発生する割り込みの頻度と割り込み処理時間を見積もって，割り込みにかかるワースト合計値を算出します。

② 1基本周期内に必ず実行完了しなければならないタスク処理のワースト合計時間を見積もります。

③ OSオーバヘッドを表1－1あるいはリリース・ノートで示されたクロック数から算出します。

④ 上記3つの合計が基本周期に対して余裕がないようであれば，割り込み頻度の緩和，必須実行処理の分割，CPUクロック高速化あるいは基本周期の延長を検討します。

(3) 許容スライス時間

基本周期内で終わらなくて良い処理は，タイム・スライスを行って分割処理します。ワースト・ケースではスライス時間は図3－1のオーバラン時間となります。

オーバランが発生しても次の基本周期の開始が遅れるだけで，遅れが累積することはありません。ユーザ・インタフェース中心のアプリケーションでは，基本周期の開始が0.5～1 ms程度遅れても問題ない場合が多いでしょう。

スライス時間が決まったら，次のタスク設計（状態遷移設計）にその条件を反映します。

3. 2. 5　タスク設計

(1) 状態関数の割りふり

本OSでは，タスクを状態遷移図（表）にもとづいて設計することを推奨します（図1－3参照）。状態遷移図の1つの状態を1つの関数（アセンブラではサブルーチン）に割り当てるため，状態ごとに関数名を付けておくとコーディング時に対比しやすくなります。

(2) タスク内静的変数の割りふり

本OSでは状態遷移時（スケジューラに戻る時）にスタックを開放する必要があります。C言語では自動的に開放されます。従って，自分自身への遷移の場合でも自動変数の内容は引き継がれません。そこで，状態遷移時に引き継ぐ内容をローカルな静的変数として定義しておきます。

(3) 状態関数の詳細設計

機能，タイミングについて詳細を定めます。もし，タイミング設計で規定した時間内に処理が終わらない場合は，状態分割（関数分割）を行います。下の例での分割手順は次のとおりです。

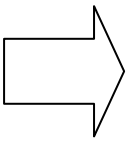
① 処理1と2の間に（A）部を追加

② 処理2と3の間に（B）部を追加

ただし，自動変数を使っている部分は値が引き継がれないので静的変数に変更する必要があります。

例：許容時間0.5 msの場合，処理を3分割する。

```
void eTask_a(void) {
  処理1;    …  0.4 ms
  処理2;    …  0.3 ms
  処理3;    …  0.4 ms
  return;
}
```



```
void eTask_a(void) {
  処理1;
  vTrans_task(eTask_a2);
}                                          (A)

void eTask_a2(void) {
  処理2;
  vTrans_task(eTask_a3);
}                                          (B)

void eTask_a3(void) {
  処理3;
  return;
}
```

3. 3 デバッグのヒント

本OSを使用したアプリケーション・プログラムのデバッグは，基本的には通常のデバッグ方法と変りありません。以下，本OS特有のケースについて説明します。

3. 3. 1 共用関数内や不正アドレス領域でプログラム停止した場合

この場合に，どのタスクの実行中だったかをスタック・ポインタから追いきれない場合は以下の手順で実行関数を特定します。

(1) タスクID取得

シンボル・リストでgcKernel_TIDのアドレスを調べ、そのアドレスのRAM内容を見ます。この変数は実行中タスクのID（0から始まるタスク登録順の番号）を示しています。

もしこの値が登録タスク数と同じであれば，スケジューラがHALTしている時に発生した割り込みの処理中に停止したことになります。それ以外の場合は次にすすみます。

(2) 状態関数のアドレス取得

シンボル・リストでgiKernel_TCB2のアドレスを調べ，giKernel_TCB2＋gcKernel_TID＊2のRAM内容を見ます。ここから始まる2バイトが実行中状態関数のエントリ・アドレスなので，シンボル・リストをアドレス順で並び換えて一致するアドレスを探せば状態関数名が分かります。

3. 3. 2 タスクをある状態関数から動作させる方法

本OSでは次の手順でタスク制御データを書き換えることにより，任意の状態関数からタスクを実行できます。

(1) 準備

リセット直後の場合はプログラム実行をスケジューラの所まで進めます。

(2) 状態関数のアドレス取得

シンボル・リストから動作させたい状態関数名を探してアドレスを調べます。

(3) 状態関数のアドレス格納

（giKernel_TCB2＋指定タスクID＊2）から始まる2バイトに動作させたい状態関数のアドレス（下位16bit）を入力します（下位、上位の順）。なおタスクIDは0から始まる登録順の番号です。

(4) タスクの実行状態の変更

（gcKernel_TCB1＋指定タスクID）の上位4bitに0xCまたは0xD（ユーザ定義フラグをセットしたい場合）を入力します。

(5) プログラム実行

スケジューラの実行を再開すれば1ラウンドロビンの間に指定タスクは指定の状態関数から実行します。もしすぐに指定タスクを動作させたい場合は，gcKernel_TIDに指定タスクのIDを書き込んでおきます。

なおスケジューラのループ開始位置main_loopにブレーク・ポイントを設定しておけば1回タスクが実行されるごとに停止するので状態変化を追いやすくなります。

第4章　タスク設計の標準化例

この章では，本OSの応用としてタスク間コマンド・インタフェースの標準化例について説明します。

4.1　タスク間インタフェースの標準化

通常の設計では1本のプログラムは複数のタスクから構成します。これら複数のタスク間で情報や信号のやり取りをしながらシステムとしての機能を果します。しかし，このやり取りが複雑に絡み合うと，タスクの独立性が損なわれ，独立した設計・コーディングが困難になったり，不要タスクの切り離しや新規タスクの追加のたびに関連タスクの修正が必要になります。

そこで，これらの手間を少しでも緩和するため，タスク間のインタフェースの標準化を考えます。以下，漢字表示デモンストレーション・プログラム（以下デモ・プログラム）での標準化例を説明します。

（1）コマンドと応答の標準化

デモ・プログラムでは，他のタスクに対するコマンド（動作指示）と結果の応答について次のように形式を定めています。

①　コマンド

1バイト目：発行元識別子（ = 発行元のタスクID ＋ 応答受付領域区分（後述））
2バイト目：コマンド種別
3バイト目：引数のバイト数（0以上）
4バイト目～：引数（形式はコマンド種別に依存）

②　応答

1バイト目：応答元識別子（ = 応答元のタスクID）
2バイト目：コマンド種別
3バイト目：応答データのバイト数（1以上）
4バイト目～：コマンドに対する動作結果。先頭1バイトはステータス情報（必須）。

コマンドを出すタスクは，①の内容を依頼先のタスクのコマンド受付領域に書き込みし，依頼先タスクをWakeupします。ただし，依頼先のコマンド領域にすでに書き込みがあればクリアされるまで待ちます。これは，自分または他のタスクが発行したコマンドをまだ実行中のためです。

コマンドを受け取ったタスクは，コマンドに基づく動作を行い，結果を発行元タスクの応答受付領域に書き込みます。応答先（発行元）情報がコマンドで渡されるため，コーディングの段階ではどのタスクからコマンドが来るかを知らなくても済みます。なお，応答受付領域にすでに書き込みがある場合はクリアされるまで待ちます。

複数のタスクからコマンドや応答を受け付ける場合は，上述のようにクリア待ちが発生する可能性があります。デモ・プログラムでは速度や効率の面でクリア待ちが望ましくない場合のために，コマンド／応答受付領域を4つまで持てるようにし，待ちの発生が無い専用の領域を指定できるようにしています。

（2）メモリ領域の標準化

デモ・プログラムの各タスクでは，前述のコマンド／応答受付領域を持つようにしています。この他にタスクの動作パラメータやデータ領域も他のタスクから標準化したアクセスができるように，下図のように標準的な配置を決めています。デモ・プログラムでは，この配置を構造体名“gタスク名”で定義しています。

図4－1　メモリ領域の標準配置（“gタスク名”）

アドレス	
a1	コマンド／応答受付領域1
b	パラメータ領域
c	データ領域
(a2)	コマンド／応答受付領域2
(a3)	コマンド／応答受付領域3
(a4)	コマンド／応答受付領域4
d	ローカル変数領域

デモ・プログラムの各タスクは，これらの領域の先頭アドレスを定数テーブル（const 配列）として定義します。この配列の名称は，“tタスク名”に固定しています。またこの配列にパラメータ領域のデフォルト値を持たせています。

また，流動的なタスクIDとは別にタスクを指名するためのユニット記号もタスク情報として持ちます。このため，デモ・プログラムにおいては，“タスク”を“ユニット”とも呼んでいます。

図4－2　タスク情報テーブル（“tタスク名”）

相対ワード位置	下位バイト	上位バイト
0	ユニット記号	受付領域数（0～4）
1	受付領域1アドレス（a1）	
2	パラメータ領域アドレス（b）	
3	データ領域アドレス（c）	
4	受付領域2アドレス（a2）（n≧2）	
5	受付領域3アドレス（a3）（n≧3）	
6	受付領域4アドレス（a4）（n = 4）	
4または3＋n（n≧1）	ローカル領域アドレス（d）	
5または4＋n（n≧1）	デフォルト・パラメータの並び（パラメータ領域分）	

各タスクのタスク情報テーブルの先頭アドレスを集めたテーブルは，UDEF_data_a.asmの中で定義しています。さらにUDEF_func_c.cの中で，これらのテーブルを使って，あるタスクの特定メモリ領域のアドレスを取得する関数を定義しています。

4.2　定義方法

デモ・プログラムにおけるメモリ領域およびタスク情報テーブルのコーディング例を以下に示します。

4.2.1　メモリ領域の定義

タスク・ヘッダ（Task_c.h）の中で，次の例のように記述します。タスク名が“ Task ”の例です。

```
struct Task_REG {
    /*----------コマンド/応答受付領域1（領域数0なら記述不要）----------------------------------*/
    unsigned char       req;                    /*  発行元／応答元識別子*/
    unsigned char       cmd;                    /*  コマンド種別  */
    unsigned char       num;                    /*  引数／応答データのバイト数  */
    unsigned char       cmdpara[サイズ];        /*引数／応答データの格納領域  */
    /*--------パラメータ領域（128バイト以内。パラメータがない場合は記述不要。）----------*/
    unsigned char       para1;
                :
    unsigned char       paran;
    /*--------データ領域（32Kバイト以内。データがない場合は記述不要。）----------------------*/
    unsigned char       data1[サイズ];
                :
    unsigned char       datan;
    /*----------コマンド/応答受付領域2～4（領域数に応じて記述）----------------------------------*/
    /*--------ローカル変数領域（ローカル変数がない場合は記述不要）----------------------*/
    unsigned char       wk;         /*  非標準のタスク間インタフェース変数もここで定義します  */
    unsigned char       tmp;
    /*-------------------------------------------------------*/
};

vGlobal struct Task_REG             gTask;
/***  領域が全く無い場合でも　vGlobal char gTask;  のように最低1バイトの定義が必要です。  ***/
```

4.2.2　メモリ領域の初期化

初期化の対象，タイミングは次のとおりです。

- gTask.cmd

  タスク起動後，コマンド受付を開始するまでに0クリアします。ただしタスク起動前に設定されたコマンドを有効にしたい場合はクリアしません。

- パラメータ領域

  タスクの起動時の挙動指定も含む可能性があるため，原則タスク起動前に他のタスクから設定します。デモ・プログラムではMainタスクで全タスクの初期化を一括で行っています。

- その他：

  必要に応じてタスク・プログラム内で初期化を行います。

4. 2. 3　タスク情報テーブルの定義

タスク・プログラム（Task_c.h）の中で，次の例のように記述します。タスク名が“ Task ”の例です。

```
const unsigned short tTask[] = { 0x141,        /* 上位1バイトは受付領域数，下位1バイトはユニット記号*/
    &gTask.req,                                /* 各領域の先頭メンバ名のアドレスを記載します。領域が無い
                                                  場合は次の領域のメンバ名を記載します。領域が全く無い場
                                                  合はgTask のアドレスを記載します。 */
    &gTask.para1,
    gTask.data1,
    &gTask.wk,
     /* para */ 0x0000, … , 0x0000};           /* パラメータ領域のデフォルト値 */
```

4. 3　アクセス方法

メモリ領域やタスク情報テーブルのアクセス方法を以下に示します。
関数を使用するにはヘッダUDEF_func_c.hをインクルードしておく必要があります。

4. 3. 1　アドレスの取得方法

あるタスクの特定メモリ領域のアドレス（ポインタ）の取得は次のようにします。

（1）タスクIDの取得

相手タスク名が不明でユニット記号しか分からない場合は，まず相手タスクIDを次の関数で取得します。

```
unsigned char uUDEF_get_TID(ユニット記号);
```

関数の返り値がUDEF_TID_ERRORの場合はエラーです（アクセスできません）。

（2）アドレス（ポインタ）の取得

次の関数で取得します。

```
unsigned char* uUDEF_get_addr(タスクID，領域名);
```

ここで，

・タスクIDはuUDEF_get_TIDの返り値を使用するか，unsigned charキャストした“ hタスク名 ”を使用します。

・領域名には次の定数名を使用します。

パラメータ領域：UDEF_ADDR_PARA
データ領域　　：UDEF_ADDR_DATA
ローカル領域　：UDEF_ADDR_LOCAL
パラメータ・デフォルト値領域　：UDEF_ADDR_IP
コマンド／応答受付領域については，「4. 3. 2　コマンド発行方法」を参照してください。

関数の返り値が0の場合はエラーです。

4.3.2　コマンド発行方法

(1) タスクIDの取得

相手タスク名が不明でユニット記号しか分からない場合は，まず相手タスクIDを次の関数で取得します。

unsigned char uUDEF_get_TID(**ユニット記号**);

関数の返り値が0xffの場合はエラーです。

(2) コマンド受取領域のアドレス取得

次の関数で取得します。

unsigned char* uUDEF_get_REG(**タスク**ID + **コマンド受付領域区分**);

ここで，

・タスクIDはuUDEF_get_TIDの返り値を使用するか，unsigned charキャストした“ hタスク名 ”を使用します。

・コマンド受付領域区分として，（受付領域番号 -1 ）<< 6　をタスクIDに加算します。

関数の返り値は次のように判断します。

== UDEF_REG_BUSY　：受付領域にすでに値がかかれています。待ちが必要です。

== UDEF_REQ_ERROR：エラーです。アドレスが取得できません。

== 上記以外　　：コマンド書き込みして下さい。相手タスクへWakeup依頼済です。

(3) コマンド書き込み

コマンドはアドレス取得後に必ずすぐ書き込みします。アドレスがポインタpに格納してあるとすると，

*p++ = 発行元識別子（発行元または返信先のタスクID + 応答受付領域区分）;

*p++ = コマンド種別;

*p++ = 引数バイト数（0以上）;

*p++ = 引数1バイト目（引数がある場合）

：

ここで，

・応答受付領域区分は，（応答受付領域番号 -1 ）<< 6　です。

4.3.3　応答方法

(1) 応答受取領域のアドレス取得

コマンド実行が完了した後，次の関数で取得します。

unsigned char* uUDEF_get_REG(**発行元識別子**);

ここで，

・発行元識別子は，コマンド書き込みされた値を使用します。

関数の返り値は次のように判断します。

== UDEF_REG_BUSY　：応答先受付領域にすでに値がかかれています。待ちが必要です。

== UDEF_REQ_ERROR：エラーです。アドレスが取得できません。

== 上記以外　　：応答書き込みして下さい。相手タスクへWakeup依頼済です。

(2) 応答書き込み

応答（コマンド処理結果）はアドレス取得後に必ずすぐ書き込みします。アドレスがポインタpに格納してあるとすると，

*p++ = 応答元のタスクID;
*p++ = コマンド種別;
*p++ = 応答データ・バイト数（1以上）;
*p++ = 1バイト目はステータス（必須）
*p++ = 2バイト目（応答データ・バイト数が2以上の場合）
：

デモ・プログラムにおいては，標準的なステータスの意味を表4－1のように定めています。これ以外の値は，タスクごとに個別に定義しています。特にデモ・プログラムのホスト・コマンド受信タスクでは，ホスト・マシンへ応答するためのステータスを数多く定義しています。

表4－1　ステータス

ステータス記号	意味
c	無効コマンドです（未定義のコマンド種別です）。
o	コマンドを正常に受け付けました。
p	コマンド引数で指定した値が不正か，個数が足りません。
r	リビジョンが異なるため要求された動作を行いません。
v	資源不足でコマンドを実行できません。
!	イベントが発生したときに自動的に送られるメッセージです。コマンドに対する応答ではありませんが，応答と同じ形式で通知する場合に使用します。

★

第5章　測定機能

この章では，リビジョン3.01で追加されたスタック消費量および実行時間の測定機能の使い方について説明します。

5. 1　測定内容

特定の状態関数について，実際の実行状態におけるスタックの消費量（最大値）と実行時間（最大値）を測定します。測定中も割り込み許可になっているため，割り込みで消費するスタックや実行時間も含まれる可能性があります。

(1) スタック消費量

デフォルトでは32バイト単位で状態関数のスタック消費量を測定します。測定は，状態関数実行前に未使用判定パターンをスタック領域に書き込み，状態関数終了後にそのパターンが保持されているか否かを判定することにより行われます。未使用判定パターンおよび判定単位バイト数はKernel_a.asmの冒頭部分を書き換えることにより変更できます。判定単位を小さくするほど，オーバヘッドが大きくなります。

(2) 実行時間

測定単位は，基本周期（標準10ms）を生成するためのインターバル・タイマのクロック単位となります。最大は標準で10～20ms（暴走判定によりタスクが打ち切られるまでの時間）です。

5. 2　測定手順

5. 2. 1　測定準備

測定機能を有効にするためのビルドを行います。また，測定対象となる状態関数のアドレスや測定条件を調べます。

(1) 測定のためのビルド

以下の設定を追加した上でビルドを行います。なお，基本周期生成用タイマはTM50固定です。

- ソース・ファイルとしてTools_c.cを追加します。これは測定用の変数と測定のオン／オフを制御するタスクです。測定オン／オフの信号入力端子として，zMS1を定義する必要があります。漢字表示デモ・ボードでは，この端子はDIP SW1-1に接続されています。
- コンパイラ・オプションの定義シンボルおよびアセンブラ・オプションのシンボル定義の欄にKOS_CODE_TYPE=1 を記述します。
- リンカ・オプションのリンク・リスト・ファイル出力でパブリック・シンボル・リストの出力にチェックを付けます。ただし，デバッガを使用する場合は，デバッガ機能でシンボル検索できますので，リンカのリストは無くても構いません。

(2) 測定条件調査

リンカの出力リストまたはデバッガ機能で測定対象の状態関数のアドレスを調べます。また，その状態関数がどのような条件で最大スタック消費，最大実行時間になるかを調べておきます。

5. 2. 2　測定の実施

次の手順で測定を実施します。

(1) 測定オフ確認

zMS1信号が‘ L ’であることを確認します。漢字表示デモ・ボードを使用する場合は，DIP SW1-1**がオフ（緑**LED**点灯）であることを確認します。**

(2) 測定変数の初期化

測定タスク（Tools**）で定義している測定用変数を初期化します。ホスト・マシンを使う方法とデバッガを使う方法があります。なお，下記で**aaaa**は状態関数のアドレスであり，リトル・エンディアンで指定します。**

① ホスト・マシンを使う方法

COM1_rx_c.cが組み込まれている場合は，ホスト・マシンから下記のライト・コマンドで設定します。

```
$WW'2 aaaa 00000000 0000
```

② デバッガを使う場合

シンボルgTools**のアドレスを調べ，メモリ・ウインドウで**gTools +6**の位置から下記を書き込みます。**

```
aaaa000000000000
```

(3) 測定開始

zMS1信号を‘ H ’にします。漢字表示デモ・ボードを使用する場合は，DIP SW1-1**をオン（赤**LED**点灯）にします。**

次に測定対象の状態関数が動くように各種操作を行います。測定時間に制限はありません。スタック，実行時間ともに最大値が保持されます。

(4) 測定終了

zMS1信号を‘ L ’にします。漢字表示デモ・ボードを使用する場合は，DIP SW1-1**をオフ（緑**LED**点灯）にします。次に測定結果を記録します。**

① ホスト・マシンを使う方法

COM1_rx_c.cが組み込まれている場合は，ホスト・マシンから下記のリード・コマンドで結果を記録します。

```
$WR'2 8
```

② デバッガを使う場合

メモリ・ウインドウでgTools **+6の位置から8バイト（16進数16桁）を記録します。**

5. 2. 3　測定結果の分析

測定結果から簡単にスタック消費量，実行時間を計算するエクセル・シート（SS_EC_78K0.xls）がダウンロード・ファイルのKERNELディレクトリに格納されています（SM05FVx_R301以降）。計測値欄に結果（ダミー５文字＋16進数16桁）を貼り付けるだけで，スタック消費量と実行クロック数，実行時間が自動的に計算されます。

この機能を有効にするには，エクセルの分析ツールが有効になっている必要があります。分析ツールを有効にするには，ツール→アドイン→分析ツール　を選択してチェックをつけます。

手作業で計算する場合は，16進数16桁に対して次のようにします。

① 状態関数アドレス

先頭4桁（2バイト）が状態関数アドレス（リトル・エンディアン）を示します。これは，結果の開始位置が正しいことの確認として調べます。

② 実行クロック数

（（（11～12桁目）*A+（9～10桁目））*B＋（7～8桁目））*C

ここで，A：システム時刻用の分周値（BTimer定義のBTIMER_CNT値），

B：基本周期用インターバル・タイマの周期（BTimer定義のkCR50値），

C：基本周期用インターバル・タイマの入力クロック分周数（f_{PRS}に対する分周数）

③ スタック消費量

最後４桁（2バイト）がスタック消費バイト数（リトル・エンディアン）を示します。

次に算出した結果に対して問題がないか，検討します。

(1) スタック消費量

スタックとして確保されたサイズに対してどの程度余裕があるか，割り込み（特に多重割り込み許可時）用のスタックが加算されても問題ないかを確認します。

なお，確保されているスタック・サイズは，リンク・リスト（パブリック・シンボル・リスト）から_@STBEG － _@STENDにより計算します。

(2) 実行時間

状態関数の実行時間が基本周期（標準10ms）に対して十分小さいか確認します。発生し得る割り込みも加算して判定します。例えば基本周期内で必須の処理であれば，図3－1（3. 2. 4項参照）の必須処理時間として割り当てる範囲で大小判定します。また基本周期内で完了しなくても良い処理は，図3－1のオーバラン許容時間を判定基準にします。大きい場合は，状態関数を分割したり，１回の処理量を減らしたりします。

★

付録A　改版履歴

A. 1　本版で改訂された主な箇所

箇　所	内　容
はじめに	
p.7	**関連資料**に追加
第1章　概　説	
p.12	**図**1－2　**スタック・サイズ比較**に注を追加
p.15	**図**1－5　**開発環境構築・開発手順の概要**の本OS入手先アドレスを変更
p.16	**表**1－1　**諸元一覧表**のパイプ定数用ROM消費量の内容を変更
第3章　使い方	
p.41	3. 1. 2 (1)**準備**　測定機能追加に伴い手順を変更
第5章　測定機能	
p.56～p.58	**第5章　測定機能**を追加
付録A　**改版履歴**	
p.59	**付録**A　**改版履歴**を追加

〔メ　モ〕

【発　行】

NECエレクトロニクス株式会社

〒211-8668　神奈川県川崎市中原区下沼部1753

電話（代表）：(044)435−5111

【ホームページ】

NECエレクトロニクスの情報がインターネットでご覧になれます。

URL（アドレス）　http://www.necel.co.jp/

【資料請求先】

NECエレクトロニクスのホームページよりダウンロードいただくか，NECエレクトロニクスの販売特約店へお申し付けください。

お問い合わせ先

【営業関係，デバイスの技術関係お問い合わせ先】

半導体ホットライン

（電話：午前 9:00〜12:00，午後 1:00〜5:00）

電　話　：(044)435−9494

E-mail　：info@necel.com

【マイコン開発ツールの技術関係お問い合わせ先】

開発ツールサポートセンター

E-mail　：toolsupport-micom@ml.necel.com

【漢字表示プログラム／ボードの技術関係お問い合わせ先】　E-mail　：kanji-demo@ml.necel.com