μITRON 導入支援プログラム

μITRONトレーナ Cmtoy

V2.00c

佐々木　芳

はじめに

本書では、Windows 上でμITRON 仕様 OS を理解するためのツール **Cmtoy** の説明と使用方法を解説します。Cmtoy は C 言語のプログラミングとμITRON 仕様 OS を理解するためのトレーニングツールです。
μITRON 上で動作するサンプルプログラムを Cmtoy 上で動作させタスク、割り込みハンドラの関連する動きを確認できます。また、Microsoft Visual Studio6.0、Microsoft Visual C++ 2005/2008 Express Edition、Borland C++などの DLL を作成できる開発環境があればサンプルプログラムを変更して動作の違いを確認できます。このように C プログラムの動きを確認しながら OS の役割や機能を理解するツールとして考えています。

1990 年代後半ごろ実際にμITRON3.0 仕様の実装、ITRON を使った通信機器ファームウェアの開発をする機会に恵まれました。その経験から時々ITRON の講習も頼まれてやっていました。そのとき感じたことは、簡単なプログラムの演習をしたいのに ITRON 仕様とは直接関係のないことで準備に時間がとられるということでした。具体的には開発用 PC、ハードウェア、デバッグツール（ICE）などの設定やシリアルケーブルの準備や動作確認です。また組込み CPU の面倒な初期設定を理解するのにも時間がとられました。そこで Windows 上のμITRON OS シミュレータあれば、効率がいいのではないかと考え始めました。それから２，３年かけて Windows の機能を調査し Cmtoy 基本形をつくり公開するにいたりました。公開するにあたり、以下の点も考慮しました。

- サンプルプログラムを用意して段階的にμITRON 仕様 OS を理解する。
- 仕事としてプログラム開発するときに必要となる文書の基本構成（システム仕様書または要求仕様書、システム分析、実装設計書など）を理解する。

このようにプログラム開発には単に C 言語などのコンピュータ言語を使えるだけでなく、日本語で書かれた文書の作成、読解が重要であることの雰囲気が伝わってくれればいいなあと思います。

V2.00 リリースにあたって(2019,4,1)
退職して時間ができたことに加えて、偶然にも VisualStudio6.0 が Windows10 で使えることを知りました。そこで再び CPU や周辺 IC のデータシート、ユーザーズマニュアルも読み直し、C-Machine のハードウェアのシミュレート方法の実装設計を見直すことにしました（2018 年初め）。実装設計を見直して全面的にコードを書き換えたので V1.07 ではなく V2.00 にすることにしました。それでも、μITRON アプリケーションのコード、バイナリには影響を与えないようにしたつもりです。
解説書も大幅に書き直しました。しかしチュートリアルの部分はほとんど同じです。チュートリアルの理解のためには以下の書籍でより詳しく解説しているので参考にしてください。
μITRON」入門―“組み込み系”「リアルタイム OS」の基礎（I・O BOOKS）　工学社

参考文献
[1] μITRON4.0 仕様 Ver4.01.00 （社）トロン協会 ITRON 部会
[2] マイクロソフト MSDN ライブラリ 2001 年 10 月リリース
[3] INTEL 8086-datasheet
[4] INTEL 80386-datasheet
[5] INTEL 80386 Hardware Reference Manual
[6] INTEL Pentium4-datasheet
[7] INTEL Pentium ファミリー　ユーザーズマニュアル
[8] INTEL 8259A PROGRAMMABLE INTERRUPT CONTROLLER
[9] MOTOLORA mc68040 32-BIT MICROPROCESSOR USER'S MANUAL
[10] National Semiconductor　PC16550D FIFO 内蔵・汎用非同期レシーバ／トランスミッタ

[11]富士通　マイクロコントローラ 16 ビットオリジナル CMOS MB90580C シリーズ DATA SHEET
[12]TMS320C54x, TMS320LC54x, TMS320VC54x FIXED-POINT DIGITAL SIGNAL PROCESSORS
[13] NEC V25 V35 16-/8- and 16-Bit Single-Chip Microcontrollers Users Manual
[14]トランジスタ技術　SPECIAL　No.8　特集　データ通信技術のすべて　CQ 出版社
[15]M系列とその応用　柏木濶著　昭晃堂発行
[16]AT 互換機　アーキテクチャハンドブック　ナツメ社
[17]Windows Internals sixth edition　Microsoft Press 発行
[18]Install Visual Studio 6.0 on Windows 10
(https://www.codeproject.com/Articles/1191047/Install-Visual-Studio-on-Windows)

ホームページ μITRON トレーナ Cmtoy

1 Cmtoy の概要

Cmtoy は、Windows の 32 ビットアプリケーションであり以下のモジュールから構成されています。

- Cmtoy.exe - GUI（Windows ダイアログアプリケーション）
- cm.dll - C-Machine（ターゲットハードウェアをシミュレート）
- kpdll.dll - μITRON カーネルをシミュレート（固定ファイル名）
- app.dll - μITRON 上のアプリケーション（ファイル名は任意）
- ActiveX コントロール - ボタン、LED、A/D コンバータなどのデバイスをシミュレート

これらは、Windows の同一プロセス内で実行されるプログラムです。ユーザはμITRON 上のアプリケーションを C 言語で記述して、32 ビット DLL(Dynamic-Link Library)形式の実行モジュールとして作成します。各モジュールの関係は下図のとおりです。

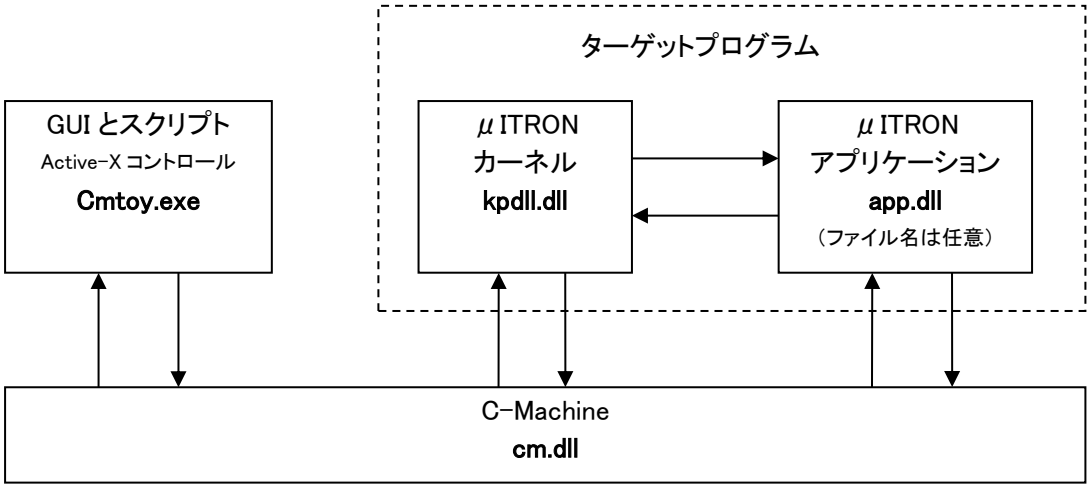


図 1-1 プログラム構成

μITRON カーネルは Cmtoy の一部として提供しています。皆さんはμITRON 上のアプリケーションを C 言語で記述して、実行形式は Windows の DLL として作成します。プログラムの作成、実行形式の DLL 作成には VisualStudio6.0 のような開発環境が必要です。Microsoft は無料の VisualStudio バージョンも配布しているのでそれを使うこともできます。Cmtoy を起動した後でμITRON 上のアプリケーションファイルを指定してをロードし、実行します。開発環境のデバッグ機能を使ってソースレベルのデバッグも可能です。
Cmtoy の目的はパソコンだけあれば、μITRON マルチタスクプログラミングの学習、体験ができるようにすることです。

※本書では言語 C の知識を前提に解説します。関数、変数、ポインタなどについては説明しません。その他ルーチン、サブルーチンなどの一般的に使用されているプログラミング用語についても説明せずに使用します。

ここで使用している用語（ターゲットハードウェア、ターゲットプログラムなど）はこれ以降の項で説明します。

1.1 GUI の概要

周辺装置を、GUI のダイアログ上にコントロールとして配置して、マウスで操作します。ハードウェア操作に加えて、デバッグ用に文字列を表示するための出力ウインドウもダイアログ内に持ちます。以下に Cmtoy のダイアログイメージを示します。

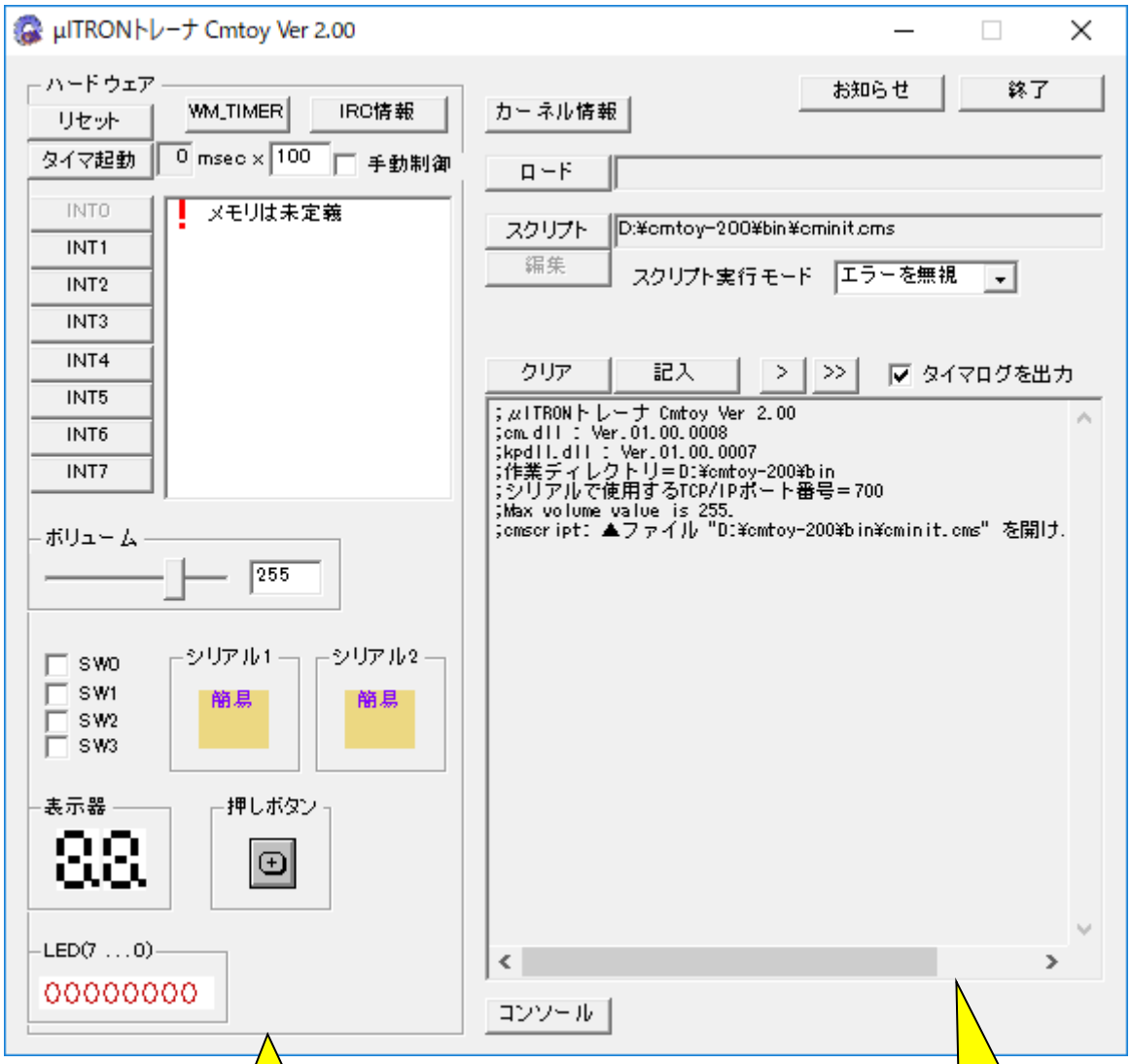


図 1-2 Cmtoy の GUI 構成

このダイアログ内の左側の「ハードウェア」の部分に配置されているコントロールがハードウェアをシミュレートする GUI となります。それ以外の右側の部分は Cmtoy を操作する GUI です。
GUI から以下の操作ができます。

- ファイル名を指定してµITRON アプリケーション（DLL 形式）を Cmtoy.exe のアドレス空間にロードする。
- µITRON カーネルを実行する。カーネルはユーザタスクを生成し、実行する。
- インターバルタイマを起動、停止。割込み周期の設定。
- 外部割込みを発生する。
- ボリューム値を変更。最大値を設定。

・ ボタン、スイッチの操作。
・ コンソールからコマンドラインによる操作。
・ スクリプトファイルによるコマンドライン操作のバッチ処理。
・ 割込みコントローラ（IRC）のレジスタを参照。
・ メモリ/IO 空間の参照。
・ ターゲットプログラム（μITRON カーネル）の情報を表示。

各ボタンの意味は、マウスポインタをその上に持っていくとツールチップが表示されて簡単な説明が表示されるので確認できます。

図 1-3「ロード」ボタンのツールチップ

1.2 ファイル構成

ホームページからダウンロードしたファイルを解凍すると以下のファイルが得られます。

```
Cmtoy-200¥
        doc¥
           Cmtoy_Vxxx.pdf          解説書
        bin¥
           Cmtoy.exe               Cmtoy の実行モジュール
           cm.dll                  C-Machine の実行モジュール
           kpdll.dll               μITRON カーネルの実行モジュール
           app1.dll                アプリケーション（以下の step1 ディレクトリ内にソースがある）
           app2.dll                アプリケーション（以下の step2 ディレクトリ内にソースがある）
           app3.dll                アプリケーション（以下の step3 ディレクトリ内にソースがある）
           app4.dll                アプリケーション（以下の step4 ディレクトリ内にソースがある）
           app5.dll                アプリケーション（以下の step5 ディレクトリ内にソースがある）
           app6.dll                アプリケーション（以下の step6 ディレクトリ内にソースがある）
           puch.ocx                ボタンをシミュレートする ActiveX コントロール
           segled.ocx              LED 表示器をシミュレートする ActiveX コントロール
           led.ocx                 LED ランプをシミュレートする ActiveX コントロール
           serial.ocx              シリアルポートをシミュレートする ActiveX コントロール
           install.bat             ActiveX コントロール登録用バッチファイル
           uninstall.bat           ActiveX コントロール登録解除用バッチファイル
           REGSVR32.EXE            ActiveX コントロール登録ユーティリティ
           script.txt              スクリプトのサンプルファイル
           sample¥                 タスク。割込みハンドラの例
           memory¥                 メモリ/IO 空間制御関係のスクリプトファイル、実行ファイル
           16550¥                  16550 制御関係のスクリプトファイル、実行ファイル
        include¥                   アプリケーションがインクルードするヘッダファイル
           hal.h                   C-Machine の機能を使うためのヘッダファイル
           hal_uart.h              16550 相当のシリアル機能を使うためのヘッダファイル
           itron.h                 μITRON の C 言語インタフェースを定義したヘッダファイル
           kernel_cfg.h            μITRON のコンフィグレーションで使用するヘッダファイル
```

```
      kernel_id.h          μITRON の機能に関する定義をしたヘッダファイル
 LIB¥                      アプリケーションがリンクするライブラリファイル
      cm.lib               VisualStudio6.0 が生成したインポートライブラリ
      kpdll.lib            VisualStudio6.0 が生成したインポートライブラリ
      2omf.bat             cm.lib,kpdll.lib を OMF 形式に変換するバッチファイル
tools¥
      cmtoy_700.ht         ハイパーターミナル定義ファイル（ポート 700 用）
      cmtoy_701.ht         ハイパーターミナル定義ファイル（ポート 701 用）
mITRON¥                    μITRON 用サンプルプログラム
      Sample¥
            Kernel_cfg.c         μITRON コンフィギュレーションファイル
            test.c               サンプルタスク
            sample.c             サンプルタスク、割込みハンドラなど
            debug.c              シリアルポートを使ったデバグ出力
            App¥
                  App.sln        app.dll を作成する Visual C++ 2008 Express Edition のソリューションファイル
                  App.dsw        app.dll を作成する VisualStudio6.0 のプロジェクトファイル
                  App.mke        VisualStudio6.0 がイクスポートしたメイクファイル
                  Makefile.bcc   Borland C++ Compiler 5.5 付属の make 用メイクファイル例
                  StdAfx.h       VisualStudio6.0 の AppWizard が生成したファイル
                  StdAfx.cpp     VisualStudio6.0 の AppWizard が生成したファイル
                  App.cpp        VisualStudio6.0 の AppWizard が生成したファイル
            Bccapp¥
                  bccapp.bdp     BCC Developper のプロジェクトファイル
      sample_16550¥              16550 制御のサンプル app.dll のソースプログラム
      sample_memory¥             メモリ/IO 空間制御のサンプル app.dll のソースプログラム
      Step1¥                     チュートリアルサンプル app1.dll のソースプログラム
      Step2¥                     チュートリアルサンプル app2.dll のソースプログラム
      Step3¥                     チュートリアルサンプル app3.dll のソースプログラム
      Step4¥                     チュートリアルサンプル app4.dll のソースプログラム
      Step5¥                     チュートリアルサンプル app5.dll のソースプログラム
      Step6¥                     チュートリアルサンプル app6.dll のソースプログラム
```

Cmtoy V2.00 では、以下のソースファイル、プロジェクトファイルを変更しています。

debug.c	バグの修正、機能追加を含む変更
hal.h	バグの修正、機能追加を含む変更
hal_uart.h	バグの修正、機能追加を含む変更
app.dsw	プリプロセッサ・マクロから”APP_EXPORTS”を削除 デバグバージョンのビルド後、*.dll を¥bin にコピーしない デバグバージョンの「デバッグセッションの実行可能ファイル」に d:¥cmtoy-200¥bin¥Cmtoy.exe を設定
app.sln	Visual C++ 2008 Express Edition のソリューションファイル（以前は 2005 Express）
serial.ocx	バグの修正(V1.6)

1.3 ターゲットハードウェアの概要

C-Machine では、仮想の 16～32 ビット CPU を搭載したスタータキット程度のハードウェアをシミュレートします。本書ではこの仮想的なハードウェアを「ターゲットハードウェア」と呼びます。ハードウェアの主な構成要素は以下のとおりです。以下、この仮想 CPU を「ターゲット CPU」と呼びます。また、ターゲット CPU 上で動くシステムプログラムを「ターゲットプログラム」と呼びます。

・CPU	ステータスレジスタに１ビットの割込み制御フラグを持つ x86 のような CPU をシミュレート
・外部割込み（16 個）	割込みコントローラ(IRC)として 8259 をシミュレート（カスケード接続のない１６レベルの割込み要求を制御）
・インターバルタイマ	レベル 0 の外部割込みを使い周期的な割り込みを発生する（μITRON の

	システムタイマとして使う）
・LED（8 個）	LED のランプ
・表示器	7 セグメント LED を 2 個配置、2 桁表示可能
・ボリューム（１個）	A/D コンバータをシミュレート（4 ビット～16 ビット）
・スイッチ（4 個）	ON/OFF 切り替えのディップスイッチをシミュレート
・ボタン（１個）	押している間だけ ON,放すと OFF となるボタンスイッチをシミュレート
・シリアルポート(2 個)	UART(RS232-C)をシミュレート（Windows のハイパーターミナルと通信できる）。16550 をシミュレート。
・メモリ/IO 空間	ターゲット CPU の物理アドレスに依存した RAM、EEPROM、フラッシュメモリ、メモリマップド IO、ポート IO などをシミュレート。
・リセットボタン	リセット信号をシミュレートして、ターゲット CPU を実行開始させる

次の項から、これらの構成要素についてプログラマおよびソフトウェアの観点から説明します。実際のハードウェアにおける各デバイスは並列に動作していますが、ソフトウェアで並列な動作をシミュレートする場合は逐次的にならざるを得ず、論理的な並列動作を追求することになります。C-Machine では実機のハードウェアを制御するプログラムを構造、制御フロー、アルゴリズムを変えずに動かすことを狙っています。

1.3.1基本構成

ターゲットハードウェアの CPU とメモリ、割込みコントローラなどの各種周辺装置（peripheral device）は１つのバス(bus)で接続されています。実機の CPU、デバイスは実際に並列動作していてバスを使う通信の信号はすべての周辺装置に届いています。しかし、バスマスタが常に通信を制御するバス通信の特性では必ず CPU とデバイスの１：１通信です。この特性を利用して C-Machine はハードウェアをシミュレートします。

C-Machine で行うハードウェアのシミュレートは、回路設計で行うハードウェアのシミュレートとは違います。回路設計では実時間（システムクロック）に沿った実時間の回路動作を検証しますが、C-Machine では回路レベルでの実時間タイミングは無視して、ターゲットプログラムの論理的な動き（制御フロー、アルゴリズム）だけをシミュレートするための機能を提供します。

(1) システムバス

ターゲットハードウェアのバスによる接続関係を一般的に以下のようなブロック図で表現します。

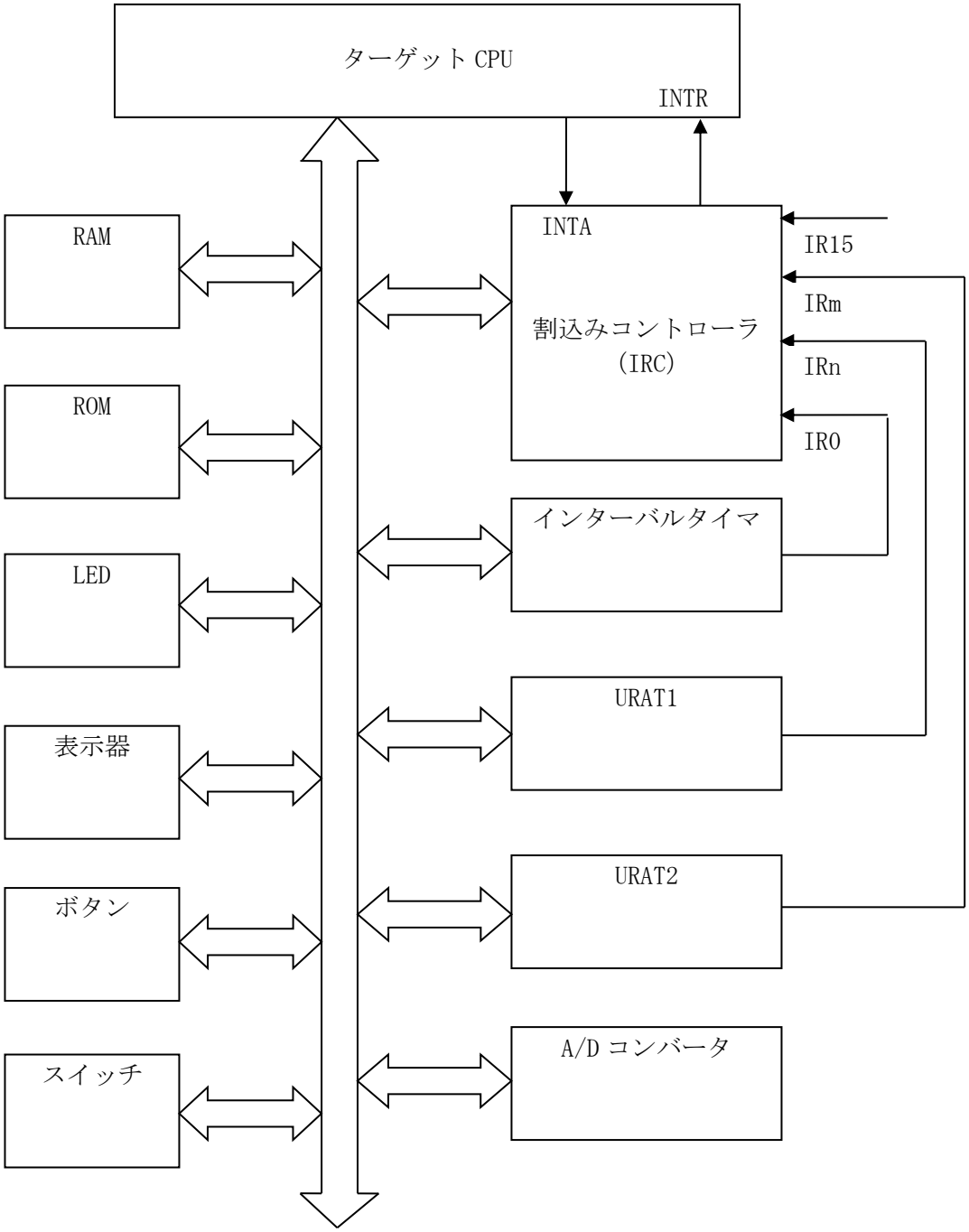


図 1-4 ターゲットハードウェアのブロック図

ここで矢印 ⟺ は、システムバス(system bus)を表します。

システムバスとは CPU と周辺装置間を結ぶ伝送路で
・アドレスバス（16～32 ラインを想定）
・データバス（8～16 ラインを想定）
・制御信号
で構成されています。CPU によってはアドレスバスとデータバスで同じ信号線を使うものもあるし、信号線を分けているものもあります。例えば 8086 CPU では 20 本のアドレス線があり、下位の 0～15 はデータバスと共通の信号線(multiplexed bus)となっています。

アドレス線の数により CPU の物理アドレスのサイズが決まります。また、ポート IO をサポートする CPU は、アドレスバスに出ているアドレスがメモリアドレスかポートアドレスかを区別する制御信号を持っています。ポートアドレスとしてはアドレス線のすべてまたは一部を使います。例えば 8086 CPU の場合 20 本のアドレス線を持っていて、メモリアドレスは 0～2^{20}-1(0xfffff)となり、ポートアドレスは 0～2^{16}-1(0xffff)の範囲です。このようにメモリ、ポートとも有限な整数値で指定される「アドレス空間（1 次元配列）」を構成します。

制御信号にはクロック信号（システムクロック）が含まれていて CPU とすべての周辺装置のタイミングの基準となる信号を提供します。CPU と周辺装置が通信する場合はこのクロック信号のタイミングに合わせて信号を制御します（クロック同期）。

CPU と各周辺装置はこの１つのシステムバスを使って通信します。そのためバスの使用権を持ったある装置がバスの制御を開始する必要があります。この使用権を持った装置をバスマスターと呼びます。複数の装置がバスの使用権を要求した場合にバスマスターが常に１つだけバス上に存在するように調停するバスアービトレーション（bus arbitration）という仕組みが組み込まれています。

ほとんどの場合 CPU がバスマスターになり CPU と各装置間で１：１のデータ送受信を行います。バスマスターがアドレスバスにアドレスを送出し、周辺装置は自分あてのアドレスを確認したらデータバスにデータを出力するか、データバスの内容を読み取ります。実際には各周辺装置がそれぞれアドレスバスを監視しるわけではなく、アドレスデコーダ(address decoder)という回路がアドレスバスを監視していて有効なアドレス信号を検出するとアドレス内容からどの周辺装置かを特定し、その周辺装置へ 1 ラインのチップセレクト信号を送出します。このように各周辺装置はチップセレクト信号だけを監視して自分が応答すべきか判断しています。データバスも各周辺装置にすべて接続しているわけではなく必要な数の信号線が接続しています。例えばデータバスが 16 ラインあっても周辺装置には 8 ラインしか接続していないこともあります。

CPU と周辺デバイスは CPU が定めるバスオペレーション(bus operation)またはバスサイクル(bus cycle)に従って通信します。制御信号にはバスオペレーションを指定する信号線が含まれています。例えば 8086 CPU には以下のバスサイクルがあります。

① 割込みアクノリッジサイクル（CPU が割込みコントローラから割込みベクタ番号を読みだす）
② I/O リードサイクル（CPU が IO ポートからデータを読みだす）
③ I/O ライトサイクル（CPU が IO ポートへデータを書きだす）
④ Halt サイクル（CPU はバス制御を停止）
⑤ 命令フェッチサイクル（CPU がメモリから命令コードを読みだす）
⑥ メモリリードサイクル（CPU がメモリからデータを読みだす）
⑦ メモリライトサイクル（CPU がメモリへデータを書きだす）

※CPU のバス構造には命令用とデータ用に物理的に別なバスをもつハーバード・アーキテクチャもありますが、通常この違いを C コンパイラが隠してくれるのでプログラマは意識しなくても問題とはなりません。

※システムバスは CPU と周辺装置を結ぶ伝送路です。小規模なコンピュータならすべての周辺装置が CPU と同じ基板上に実装されていますが、中にはオプションの周辺装置を別基盤に実装したい場合もあります。そのような場合は PCI（Peripheral Component Interconnect）という規格に基づいてシステムバスを別基盤に延長します。PCI により延長されたシステムバスに繋がった周辺装置はプログラムからは別基盤上にあることを意識せずに使うことができます。

(2) メモリシステム

メモリは 8 ビットまたは 16 ビットのメモリセルの配列です。アドレスバスはこのセルの配列要素を指定します。このメモリセルが CPU の読み書きする最小単位となります。ほとんどの CPU のメモリ

システムのメモリセルは 8 ビットです。16 ビットのメモリセルの場合、データバスは最低 16 ライン必要です。本書では、8 ビットメモリセルを使う CPU を「バイトアドレッシング」、16 ビットのメモリセルを使う CPU を「ワードアドレッシング」と呼ぶことにします。
8/16 ビットのメモリセルの違いにより C コンパイラのデータタイプは以下のようになることが考えられるので注意が必要です。（char のビットサイズが違います。）

メモリセルのタイプ / C 言語のデータタイプ	8bit メモリセル バイトアドレッシング	16bit メモリセル ワードアドレッシング
char, unsigned char	8bit	16bit
short, unsigned short	16bit	16bit
long, unsigned long	32bit	32bit

(3) CPU

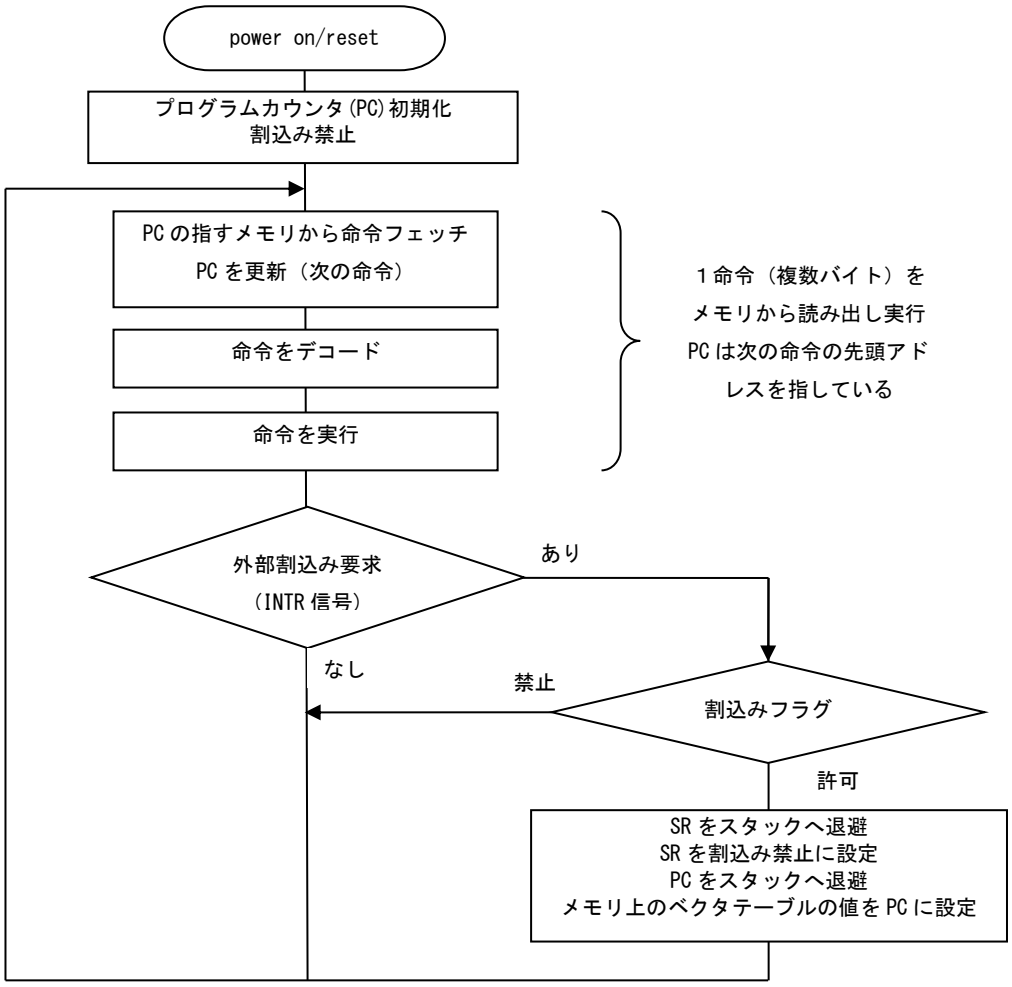
まずプログラマから見える CPU の動作、プログラミングモデルを確認します。ここではプログラム内蔵方式 CPU を前提にします。プログラム内蔵方式 CPU ではメモリを次のような用途に対応した領域に分けて使います。

- コード領域（命令コードを格納する）
- データ領域
- スタック領域（後入れ先出しのデータ構造として使う）
- 割込みベクタテーブル

これらの領域を前提に以下のようなレジスタ構成を持ちます。

- PC（プログラムカウンタ）　　コード領域内の次に実行する命令コードを指す
- SP（スタックポインタ）　　スタック領域内で後入れ先出し構造を扱う
- SR（ステータスレジスタ）　　CPU の内部状態を示す（割込み禁止／許可フラグを持つ）
- 汎用レジスタ（ｎ個）　　演算やメモリアドレッシングに使う

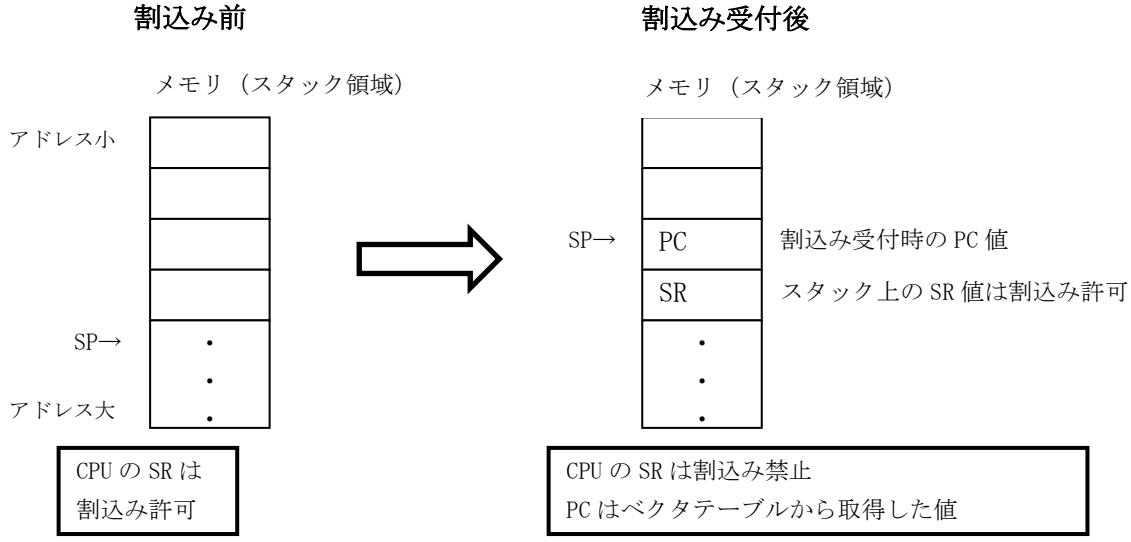
以下のフローチャートはプログラム内蔵式 CPU の働きのすべてです。最新の CPU はマイクロアーキテクチャ構造を持ち、パイプライン、命令先読み、分岐予測などを行うものもありますが、プログラマおよびソフトウェア（プログラム）から見える CPU の動作はこのフローチャートとなります。
本書では CPU 例外が発生した場合の動作については考えないことにします。



1-5 CPU の動作

これを見るとわかるように CPU は常にバスサイクルを使ってメモリにアクセスします。メモリから命令を読み込み、その命令に従った動作をします。命令の内容によってはメモリからデータを読み込んだり、メモリへデータを書き出します。割込みが発生すると割込みアクノリッジサイクルで割込み要求に対応するベクタ番号を読み出し、メモリ上のベクタテーブルからベクタ番号に対応する内容を読み出しプログラムカウンタ（PC）へ設定します。
このように CPU は 1 命令実行するために複数のバスサイクルを使用します。割込みハンドラの起動は命令終了時に行われ、ある命令を実行している最中のバスサイクルに割り込むことはありません。

割込み受付前後の CPU、スタックの様子を以下に示します。



1.3.2ターゲット CPU

(1) プログラミングモデル

ターゲット CPU のレジスタ構成は以下のように想定します。

PC（プログラムカウンタ）
SP（スタックポインタ）
SR（ステータスレジスタ）　　割込み禁止／許可フラッグを持つ
汎用レジスタ群　　8 ビット、16 ビット、32 ビットレジスタで構成される

ターゲットプログラムを C 言語で開発することを想定しているのでプログラムがこのレジスタ構成に依存することはありません。

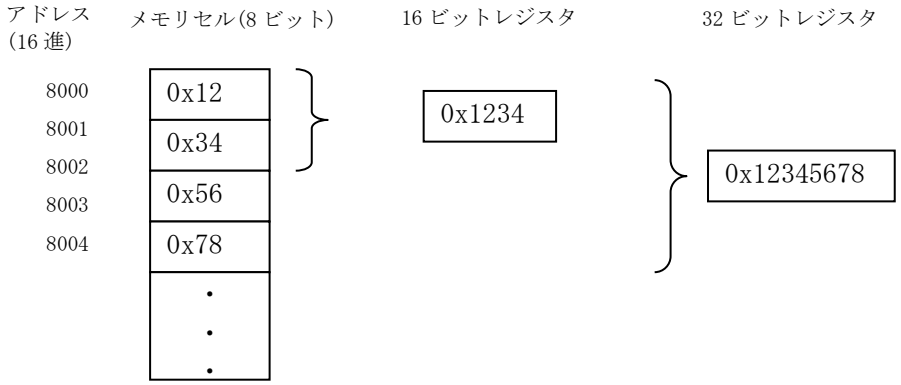
ターゲット CPU のアドレス空間は、論理アドレスと物理アドレスが一致しているものを想定しています。アドレス変換（ページ機構やセグメンテーション機構）を行わない CPU です。
物理アドレスとはアドレスバス（アドレス線）で指定されるメモリアドレスで、ハードウェアの実装で決まります。一方論理アドレスとは CPU の実行するプログラムで使用するメモリアドレスです。
例えば C 言語の以下のプログラムで 0xe000 は論理アドレスです。

```
char *p = (char*)0xe000;
*p = 0;
```

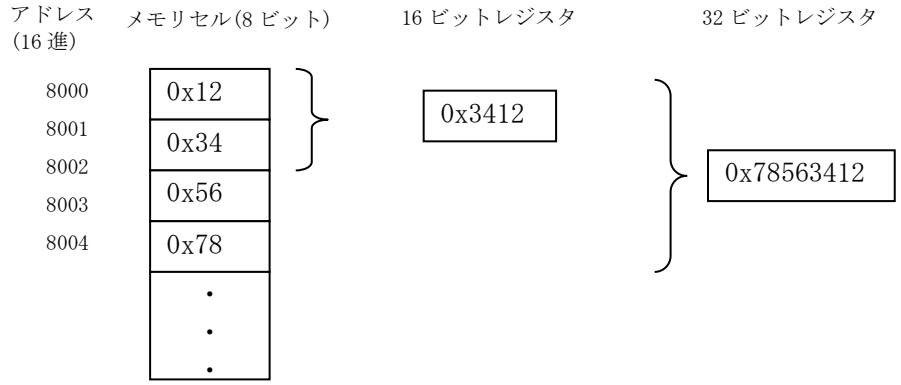
これは論理アドレス 0xe000 へ 8 ビットデータ 0 を書き込むという意味になります。論理アドレスと物理アドレスが一致しているということはアドレスバスへも 0xe000 が出力されるということです。

ターゲット CPU は 8 ビット、16 ビット、32 ビットの整数データを扱います。このデータをメモリに保存する方法を考察しましょう。ビッグエンディアン(big endian)とリトルエンディアン(little endian)という２つの保存方法を説明します。どちらもメモリセルのサイズより大きなデータは連続したメモリセルに保存します。
まず、メモリセルが 8 ビットのメモリと CPU のレジスタの間でデータを交換する場合を考えます。アドレス 0x8000 に 16 ビットデータを読み書きする場合と 32 ビットデータを読み書きする場合を以下に示します。

ビッグエンディアン



リトルエンディアン



8 ビットメモリセル上のデータの並び

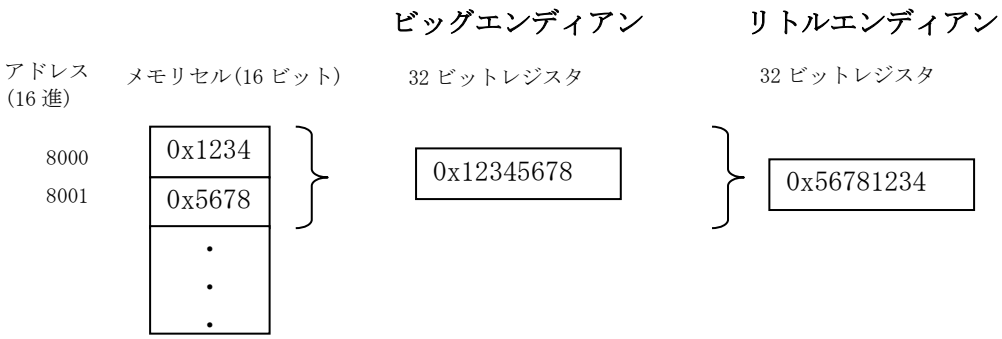
C 言語のプログラムでもこの違いを確認しましょう。

```
typedef unsigned char      BYTE;
typedef unsigned short     WORD;
typedef unsigned long      DWORD;
BYTE* p = (BYTE*)0x8000;
BYTE B1, B2;
WORD W1, W2;
DWORD D1;
*p = 0x12;
*(p+1) = 0x34;
*(p+2) = 0x56;
*(p+3) = 0x78;

W1 = *(WORD*)p;
D1 = *(DWORD*)p;
B1 = (BYTE)W1;
B2 = (BYTE)D1;
W2 = (WORD)D1;
if(B1 == B2)
   ;//little endian B1 = B2 = 0x12
else
   ;//big endian       B1= 0x34, B2= 0x78
```

```
if(W1 == W2)
    ;//little endian W1 = W2 = 0x3412
else
    ;//big endian W1= 0x1234, W2= 0x5678
```

次に、メモリセルが 16 ビットのメモリと CPU のレジスタの間でデータを交換する場合を考えます。アドレス 0x8000 に 32 ビットデータを読み書きする場合を以下に示します。



16 ビットメモリセル上のデータの並び

ちなみに、エンディアンが問題になるのは、ネットワークを通してバイナリデータを交換する場合やファイルを通して異なるシステムとバイナリデータを交換する場合です。参考までに、TCP/IP プロトコルスタックではビッグエンディアンに統一しているようです（ネットワークバイトオーダー）。

ところで Windows が動いている x86 CPU は、8 ビットメモリセルでリトルエンディアンです。そのため Cmtoy では μ ITRON アプリケーションを作成する C コンパイラは 8 ビットメモリセルでリトルエンディアンのシステムを前提にしています。ですから C コンパイラの作成するコードセクション、データセクション、スタックセクションはすべて 8 ビットメモリセルかつリトルエンディアン形式で、配置されるアドレスは Windows が決める 32 ビットアドレス内のどこかになります。このように C 言語で定義したデータはすべて 8 ビットメモリセルでリトルエンディアンとして扱われます。C 言語は CPU に依存せずプログラムを書くために開発されてきた経緯があるので、Windows で動作確認ができたプログラムは別な CPU に移植しても実行結果も同じになることが基本的に保証されています。CPU に依存するインラインアセンブラはマクロで記述するなどして工夫すれば、移植の工数を大幅に減らすことは可能となります。
しかし、ハードウェアの実装で決まる物理アドレスに依存するメモリ、IO ポートは、C-Machine がシミュレートします。C-Machine はエンディアン、メモリセルの組み合わせで決まる 4 通りのメモリシステムをシミュレートします。ハードウェアに依存するメモリを C 言語から取り扱うためには C-Machine の用意する関数を経由しなければ正しく扱えません。

ハーバード・アーキテクチャを採用した DSP(Digital Signal Processor)の中には特定の命令と命令の間に同期をとるためにプログラムで明示的にクロックを消費するためだけの命令を実行する必要があるものがあります。このような場合でも C 言語でプログラムを開発していると C コンパイラが命令の並びを判定して勝手に必要なクロック数を消費する命令を挿入するのでプログラマはそれを意識しなくても問題にはなりません。

(2) バスサイクル

ターゲット CPU は以下のバスサイクルを定義します。

① 割込みアクノリッジサイクル（CPU が割込みコントローラから割込みベクタ番号を読みだす）
② I/O リードサイクル（CPU が IO ポートからデータを読みだす）
③ I/O ライトサイクル（CPU が IO ポートへデータを書きだす）
④ Halt サイクル（CPU はバス制御を停止）
⑤ 命令フェッチサイクル（CPU がメモリから命令コードを読みだす）
⑥ メモリリードサイクル（CPU がメモリからデータを読みだす）
⑦ メモリライトサイクル（CPU がメモリへデータを書きだす）

(3) 制御信号

ターゲット CPU の主な制御信号をまとめます。これは、あくまでプログラマが CPU の動作を理解するための参考と考えてください。（アドレスバス、データバスは省略）

信号シンボル	入出力	信号名と説明
INTR	入力	INTERRUPT REQUEST:命令の最後のクロックサイクルで評価し、割込みアクノリッジサイクルを開始するかどうか決める。
INTA	出力	INTERRUPT ACKNOWLEDGE:CPU による割込みアクノリッジサイクルを示す
S0,S1,S2	出力	STATUS:バスサイクルを区別する
M/IO	出力	STATUS LINE:メモリアクセスと IO アクセスを区別する。
RD	出力	READ:メモリリードサイクルまたは IO リードサイクル
WR	出力	WRITE: メモリライトサイクルまたは IO ライトサイクル
LOCK	出力	LOCK:他のバスマスターがシステムバスの制御権を取れないようにする
HOLD/HLDA	入力／出力	HOLD/HOLD ACKNOWLEDGE:他のバスマスターがシステムバスの制御権を要求するときに使う。
CLK	入力	CLOCK:CPU およびバスコントローラーの基本タイミングに使うクロックを供給
READY	入力	READY:バスサイクルを終了していいか判断する。それまでアイドル状態（複数クロック）をバスサイクルに挿入する。

この表の「入出力」は CPU から見た場合の入力と出力です。出力とある信号は CPU が駆動し、入力とある信号は周辺装置が駆動します。アドレスバスは CPU が駆動し、データバスはバスサイクルにより CPU または周辺装置が駆動します。

LOCK 信号（出力）：CPU は複数の連続するバスサイクル間を他のバスマスターからの保護するために LOCK 信号（BUS LOCK）を使います。例えば 1 命令でメモリからリードしてデータを修正し同じアドレスに書き戻す（READ/MODIFY/WRITE 操作）場合、LOCK 信号を使うとこの途中で他の周辺装置が同じアドレスのメモリを書き換える（衝突）のを防ぎます。

HOLD 信号（入力）：CPU は周辺装置をバスマスターにするため HOLD 信号（BUS HOLD REQUEST）を監視します。CPU は一連のバスサイクルが終了して HOLD 信号を受信していた場合、HLDA（Hold acknowledge）を出力してバスマスターを譲ることを知らせます。LOCK 信号を出力している間は HLDA を出力しません。バスマスターを譲った後 CPU はシステムバスを制御しません。

CPU は Halt 命令を実行すると Halt サイクルを実行して、システムバスを使うのをやめ Halt 状態に入ります。PC は Halt 命令の次のアドレスを指しています。外部割込みが発生すると Halt 状態を終了し、通常の動作に戻ります。

※中にはREADY信号を操作しない遅い周辺装置もあります。そのような場合CPUが明示的に必要なクロックを待つ必要があります。例えば8086ではNOP命令をプログラムで実行して必要なクロック数を確保すします。C言語プログラムにNOP命令を挿入するには以下のようなマクロを定義して行います。（Microsoft C/C++ Compilerの場合）

```
#define NOP       _asm  nop
```

※実際のCPUのデータシートには、各バスオペレーションでこれらの信号がどのように関連しあって使われているかを示すタイミングチャート（Timing Waveforms）が載っています。

(4) 割込みモデル

ターゲットCPUの割込みモデルは、8086と8259A（割込みコントローラ）を参考にしています。周辺装置の割込み要求信号IRはIRC(割込みコントローラ)のIRn（n=0～15）ピンにつながっています。周辺装置から割込み要求が発生してからCPUが割込みハンドラプログラムを起動するまでの流れは以下のようになります。

① 周辺装置はIR信号をIRCへ送る。
② IRCはCPUへINTR信号を送る。
③ CPUはSRとPCをスタックへ保存。（ライトサイクル）
④ CPUはSRの割込みフラッグを割込み禁止にする。
⑤ CPUはIRCから割込み番号を読み取る。（割込みアクノリッジサイクル）
⑥ CPUは割込み番号からベクタテーブルの該当アドレスの内容を読み出して（リードサイクル）、PCへ設定する。

1.3.3メモリマップドIOとポートマップドIO

CPUが周辺装置と通信する方法を考察します。周辺装置のレジスタはハードウェアの実装設計で決めた特定のメモリアドレスまたはポートアドレスに配置されます。どちらのアドレスを使うかによってメモリマップドI/O（Memory-mapped I/O）とポートマップドI/O（Port-mapped I/O）と呼びます。ここでいうアドレスとは物理アドレスのことです。

CPUはバスマスターとなりこれらのレジスタを読み書きすることで周辺装置を制御します。メモリアドレスに配置されたレジスタはメモリリードサイクル／メモリライトサイクルを使って読み書きし、IOポートアドレスに配置されたレジスタはIOリードサイクル／IOライトサイクルを使って読み書きします。CPUがアドレスバスに出力したアドレスはアドレスデコーダ回路により設計時に決めたアドレスに従い対象周辺装置にチップセレクト信号を送ります。チップセレクト信号を受けた周辺装置はバスサイクルのタイミングに従ってデータバスにデータを出力したり、データバスからデータを入力します。遅い周辺装置はREADY信号を送出しデータがそろうまでCPUにバスサイクル終了を遅らせることができます。READY信号を使ってハードウェアがタイミングを遅らせるてもプログラムは影響を受けません。また、その実行中の命令の時間が延びるだけなのでプログラムからはわかりません。

メモリマップドIOの場合は、周辺装置のレジスタはROM,RAMと同じメモリアドレス内に存在します。したがって、C言語で周辺装置を操作できます。しかし、周辺装置のレジスタとROM,RAMには決定的な違いがあります。それぞれの特徴を列記してみましょう。

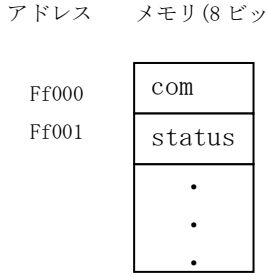
①ROM内のデータは常に変わりません、CPUはいつ読み出しても同じアドレスからはいつも同じ値が得られます。
②RAM内のデータはCPUが書き込んだ値です。CPUが書き換えない限りいつまでも同じ値を保持しています。電源ON直後はRAMのデータは不定です。したがって、通常電源ON直後にRAM領域全体

をプログラムで 0x0 に初期化します。パリティ付き RAM の場合はこの初期化は必須です。
③周辺装置のレジスタは CPU と関係なく書き換わります。CPU は読むたびに違う値が得られることを前提にしなければなりません。
④周辺装置のレジスタへ書き込んだ値は周辺装置へのコマンドとなることがあります。この場合は、書き込む順番が重要な意味を持ちます。

CPU から見ると特徴の③は、放っておくと消えてなくなってしまう揮発性物質を連想させます。このアドレスにはあたかも揮発性(volatile)データが格納されているといえます。この特徴は C 言語の型修飾子 volatile を使って扱います。

(1) C 言語でメモリマップド IO を使う

簡単な C プログラムを使って考察しましょう。
ここで、ある周辺装置は２つのレジスタ com, status を持っていて、com はアドレス 0xff000、status はアドレス 0xff001 に配置されているものとします。




```
typedef struct some_device_reg{
    char   com;
    char   status;
}DEV_REG;

DEV_REG* pDevReg = (DEV_REG*)0xff000;

void Clear()
{
    char temp = pDevReg->status;
}
```

この関数 Clear()内の変数 temp は一度も使われないので C コンパイラはこの代入文を機械語に変換する必要はないと判断してしまい、何もしない関数となってしまいます。関数 Clear()にメモリリード命令（１回のメモリリードサイクル）を実行させるには修飾子 volatile を使用し、構造体の宣言を以下のように変えます。

```
typedef volatile struct some_device_reg{
    char   com;
    char   status;
}DEV_REG;
```

こうすることにより、C コンパイラは構造体 some_device_reg のメンバに関する最適化を行いません。
また、周辺装置を設定するための以下のような関数を考えます。

```
void Init()
{
    pDevReg->com  = (char)0x80;
    pDevReg->com  = (char)0x01;
    pDevReg->com  = (char)0x40;
    pDevReg->com  = (char)0x55;
}
```

RAM 領域のあるアドレスにこのようにデータを書きこんだ場合、1 回 0x55 書き込めば同じことになりますが、メモリマップド IO の場合は違う意味になります。メモリマップド IO の場合は、周辺装置がこの順番に書かれたデータ内容に従って内部状態を変えるので、この順番に書かれたデータが重要な意味を持ちます。
ここで注意すべきは、以下の C 言語プログラムの 0xff000 は論理アドレスです。

```
DEV_REG* pDevReg = (DEV_REG*)0xff000;
```

周辺装置を制御するためにはアドレスバスにもこの 0xff000 が出力される必要があります。そのためメモリマップド IO として使われるアドレス領域は CPU がアドレス変換（ページ変換など）をしないことが重要です。同時にこの領域はメモリキャッシュの対象外にしておく必要があります。メモリキャッシュ領域では、CPU がリード命令、ライト命令を実行しても実際にメモリリードサイクル／ライトサイクルが実行されるのはいつになるのかプログラムでは感知できないので、周辺装置の制御が正しくできません。

Cmtoy では、エンディアンを含めメモリマップド IO とポートマップド IO 領域をシミュレートします。したがって、C-Machine の提供する関数を使ってこれらの領域にアクセスする必要があります。例えば、上記で示した関数は以下のようになります。

```
void Clear()
{
    char temp = PREAD_BYTE(pDevReg->status);//READ_BYTE(0xff001)と同じ
}
void Init()
{
    PWRITE_BYTE(pDevReg->com, 0x80);// WRITE_BYTE(0xff000, 0x80)と同じ
    PWRITE_BYTE(pDevReg->com, 0x01);
    PWRITE_BYTE(pDevReg->com, 0x40);
    PWRITE_BYTE(pDevReg->com, 0x55);
}
```

(2) C 言語でポートマップド IO を使う

ポート IO を制御する場合、C のプログラム内にはインラインアセンブラ機能を使って埋め込みます。インラインアセンブラ行は最適化の対象にならないので記述したとおりに実行されます。
INTEL x86 互換 CPU の場合、IO ポートのアクセスに以下のような関数を定義します。これはマイクロソフトの C/C++コンパイラを使った場合の例です。

```
//ＩＯポートから入力
DWORD input_dword( WORD port )
{
    DWORD data;
    _asm{
        mov dx,port
```

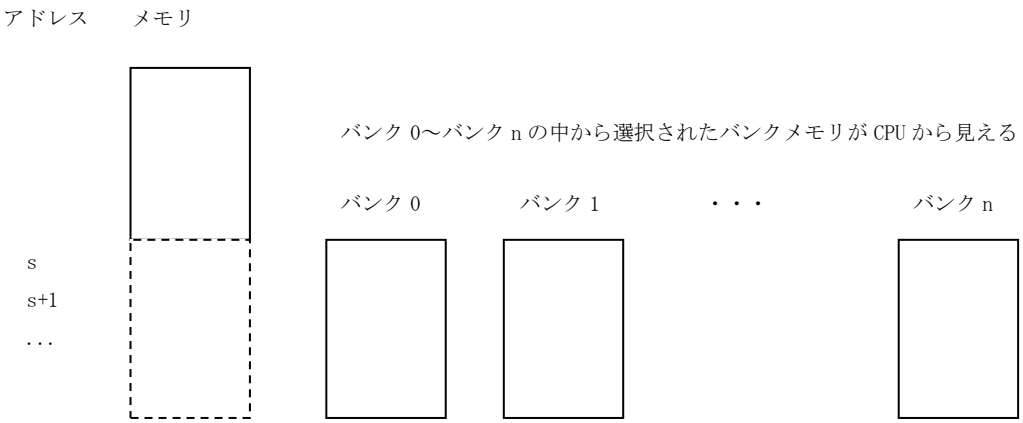
```
        in  eax,dx
        mov data,eax
    }
    return data;
}

//ＩＯポートへ出力
void output_dword( WORD port, DWORD data)
{
    _asm{
        mov eax,data
        mov dx,port
        out dx,eax
    }
}
```

※インラインアセンブラの記述方法はコンパイラ依存です。

1.3.4メモリバンク

メモリバンクというメモリ実装について考察しましょう。メモリバンクとはメモリアドレスのある連続領域に複数のメモリを実装する仕組みです。CPU からアクセスできるのはその中の 1 つです。したがって、ハードウェアでどのバンクのメモリがアクセスできるか（CPU から見えるか）を変更する仕組みを実装しておく必要があります。これを「バンクメモリを選択する」とか「切り替える」ということにします。



メモリバンクの構造

バンクメモリの選択はその目的によりプログラムから行うか、周辺装置から行うかが考えられます。フォントデータなど大量の ROM データを小さいメモリ領域で扱う場合などは、CPU が必要なバンクを選択してその中のデータにアクセスします。定期的に必ず受信する通信パケットを CPU に渡す場合に周辺装置がデータをバンクメモリに書き込み、書き込み完了後にバンクメモリを切り替えて CPU に割込みで通知するなどの使い方も考えられます。

1.3.5割込みコントローラ(IRC)

ターゲット CPU の割込みモデルは、8086 と 8259A（割込みコントローラ）を参考にしています。た

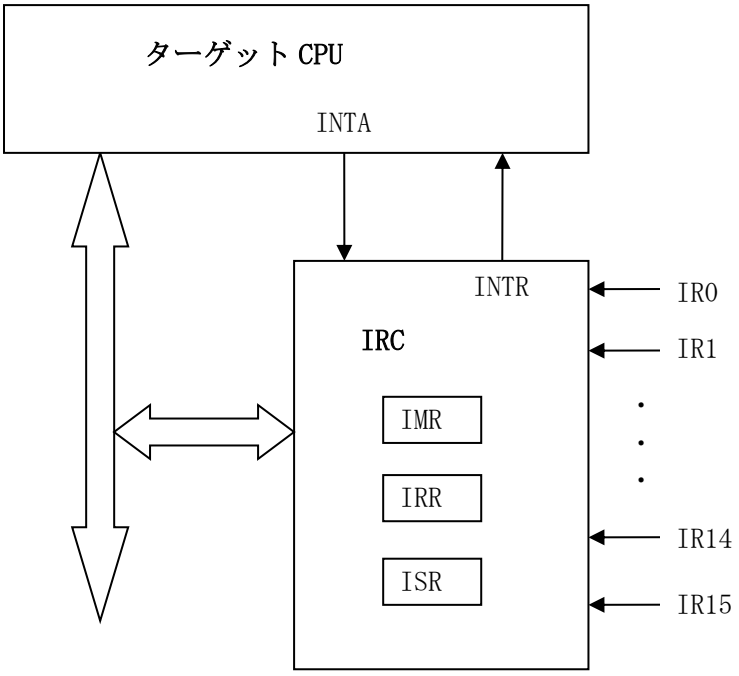
だし、IRC は 16 レベルを扱えて、8259A に即していえばエッジトリガモード、固定優先度、指定 EOI（EndOfInerrupt）コマンドモードと同等で、カスケード接続はないものとします。割込みレベル０が最高優先度とします。

IRC の主な制御信号をまとめます。これは、あくまでプログラマが IRC の動作を理解するための参考と考えてください。

信号シンボル	入出力	信号名と説明
INTR	出力	INTERRUPT:CPU へ割り込みを通知する
IR0 - IR15	入力	INTERRUPT REQUESTS:周辺装置からの割込み要求
INTA	入力	INTERRUPT ACKNOWLEDGE:CPU による割込みアクノリッジサイクルで割込みベクタ番号をデータバスへ出力するために使われる
CS	入力	CHIP SELECT:セットされたら CPU と IRC の通信が可能となる。INTA は CS と独立に評価する。

ここで、入出力は割込みコントローラから見た入力、出力です。

CPU との接続は以下のようになります。



IRC は次の 3 つの 16 ビットレジスタで割込み信号を制御します。各レジスタのビット０は IR0、ビット１は IR1、…に対応します。

- IMR（割込みマスクレジスタ）
- IRR（割込み要求レジスタ）
- ISR（割込みサービス中レジスタ）

次に割込み制御手順を整理します。

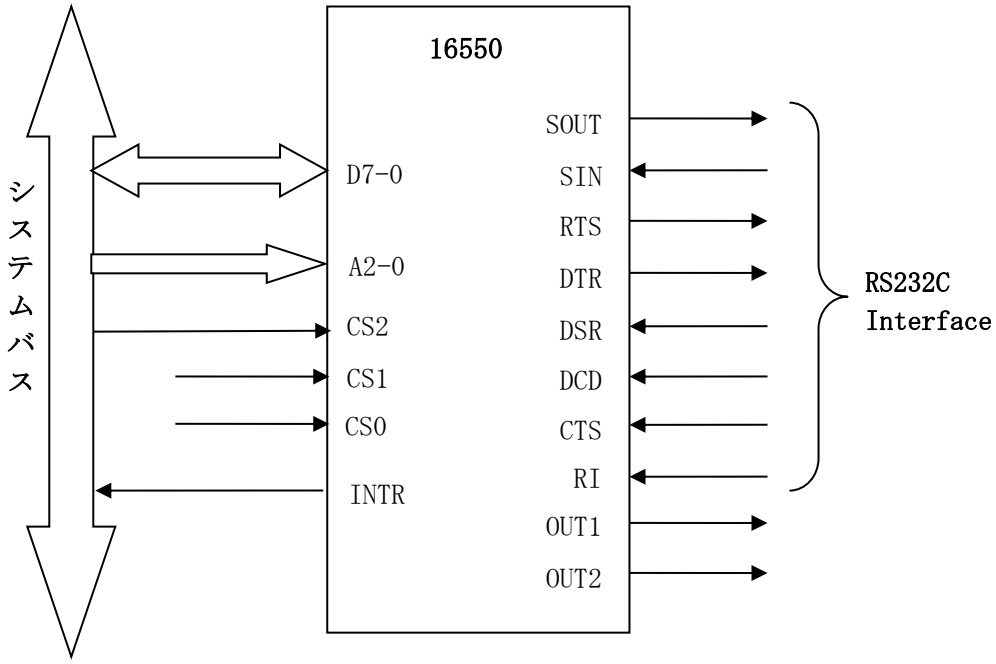
① IR0-15 に割込み要求が起きると対応する IRR のビットをセットする。
② IRC はマスクされていない IR が起きると CPU へ INTR を送る。
③ CPU は INTR を検知すると INTA 信号で応答する。
④ IRC は INTA 信号を受けると IRR の中で最高優先度のレベルに対応する ISR のビットをセットす

る。
⑤ IRC は対応するベクタ番号をデータバスへ出力し、CPU がそれを読み込み INTA を終了。CPU は割込みハンドラを実行する。
⑥ ISR のセットされたビットは、割込みハンドラの最後で実行される指定 EOI コマンドでリセットされる。

1.3.6シリアルコントローラ

シリアルコントローラとして 16550 UART(Universal Asynchronous Receiver/Transmitter)をシミュレートします。

まず、16550 の基本構成を示します。



16550-CPU 間の主な制御信号をまとめます。

信号シンボル	入出力	信号名と説明
A0,A1,A2	入力	REGISTER SELECT:UART レジスタの選択
CS0,CS1,CS2	入力	CHIP SELECT:
INTR	出力	INTERRUPT:IRC の IRn へ割込み要求
D7-0	入出力	DATA BUS:CPU とデータ、制御ワード、ステータス情報の送受信に使う。
RD	入力	READ:CPU はレジスタをリードする。
WR	入力	WRITE:CPU はレジスタへライトする。

ここで、入出力は 16550 から見た入力、出力です。

A2-A0 で指定されるレジスタアドレスは、0～7 となります。各レジスタは 8 ビットです。レジスタの指定はレジスタアドレス、リード／ライト、DLAB(Divisor Latch Address Bit)の組み合わせで指定します。DLAB は FIFO Control Register (FCR)のビット 7 で、リセット後は 0 となります。以下に 16550 のレジスター覧を示します。

レジスタ名	DLAB (FCR.bit7)	レジスタアドレス (A2,A1.A0)	Read/Write
Receiver Buffer Register (RBR)	0	0	Read
Transmitter Holding Register (THR)	0	0	Write
Interrupt Enable Register (IER)	0	1	Read/Write
Interrupt Identification Register (IIR)	X	2	Read
FIFO Control Register (FCR)	X	2	Write
Line Control Register (LCR)	X	3	Read/Write
Modem Control Register (MCR)	X	4	Read/Write
Line Status Register (LSR)	X	5	Read/Write
Modem Status Register (MSR)	X	6	Read/Write
Scratch Register (SCR)	X	7	Read/Write
Divisor Register (LS)	1	0	Read/Write
Divisor Register (LM)	1	1	Read/Write

※ここでXはDLABの値によらないことを示します。

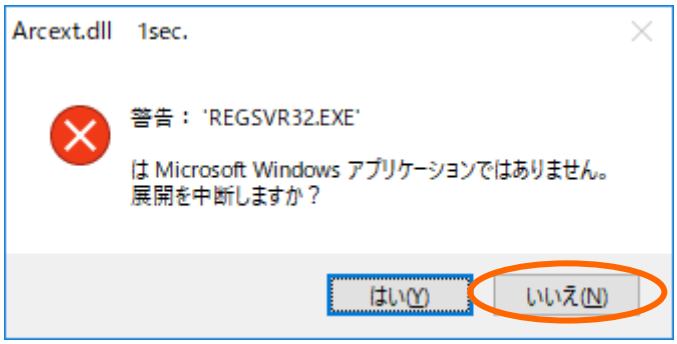
各レジスタは8ビットなので、連続する８個のアドレスに配置できます。ハードウェアの実装によっては偶数アドレスに各レジスタを配置することもあります。または４バイトおきに配置することもあります。

Cmtoyではこれらのレジスタを読み書きする関数を用意するのでメモリアドレスまたはIOアドレスのどこに配置されるかは気にすることはありません。RS232C側でのデータの送受信はTCP/IPのポートを使用してシミュレーションします。送信FIFOにあるデータは１バイトづつ取り出しTCP/IPポートへ送信します。送信は10msおきに行うので毎秒100文字の送信スピードとなるのでRS232Cの1200BPSと同じくらいとなります。16550の割込みレベルはコンソールまたはスクリプトから設定します。

2　使用方法

2.1 インストール

ここでは Cmtoy-200.zip を Windows10 の **d:¥cmtoy-200** に解凍した前提で説明します。解凍すると「1.2 ファイル構成」で説明したフォルダ、ファイルが得られます。
※Windows10 で Explzh を使って解凍すると以下のようなダイアログが表示されますが、「いいえ」をクリックして展開を続行してください。



まず、ActiveX コントロールを登録します。コマンドプロンプト（DOS 窓）を管理者モードで開き、カレントディレクトリを cmtoy-200¥bin に変更し、install.bat を実行します。以下のようなログがコマンドプロンプトウィンドウ（DOS 窓）に表示されます。

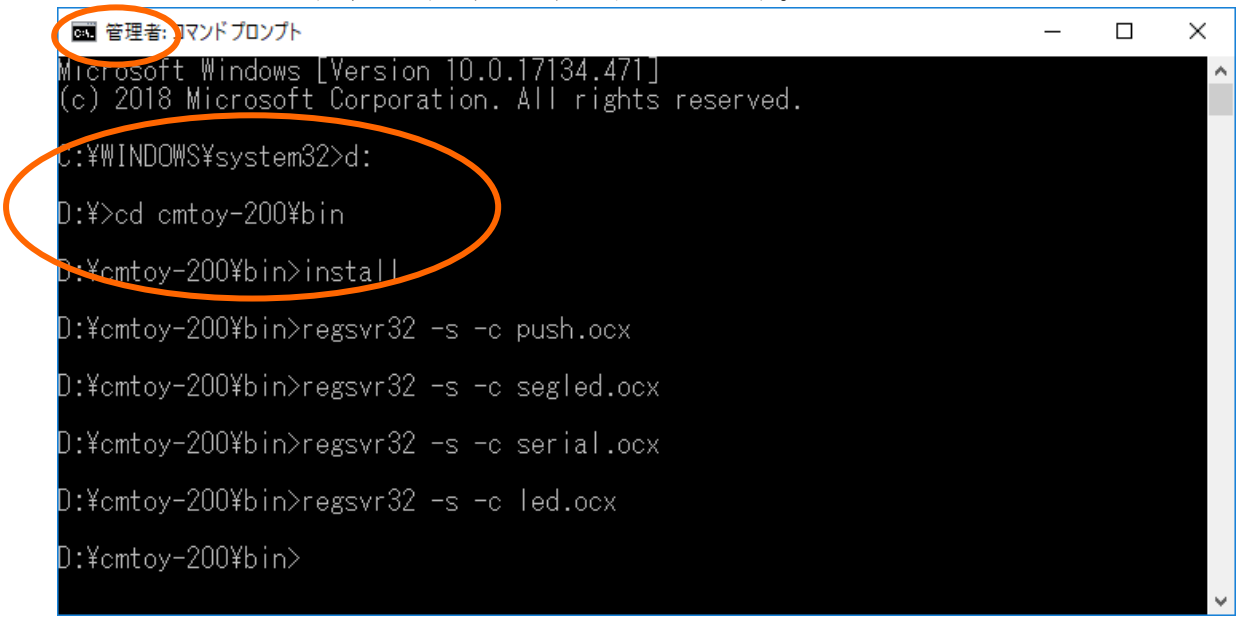


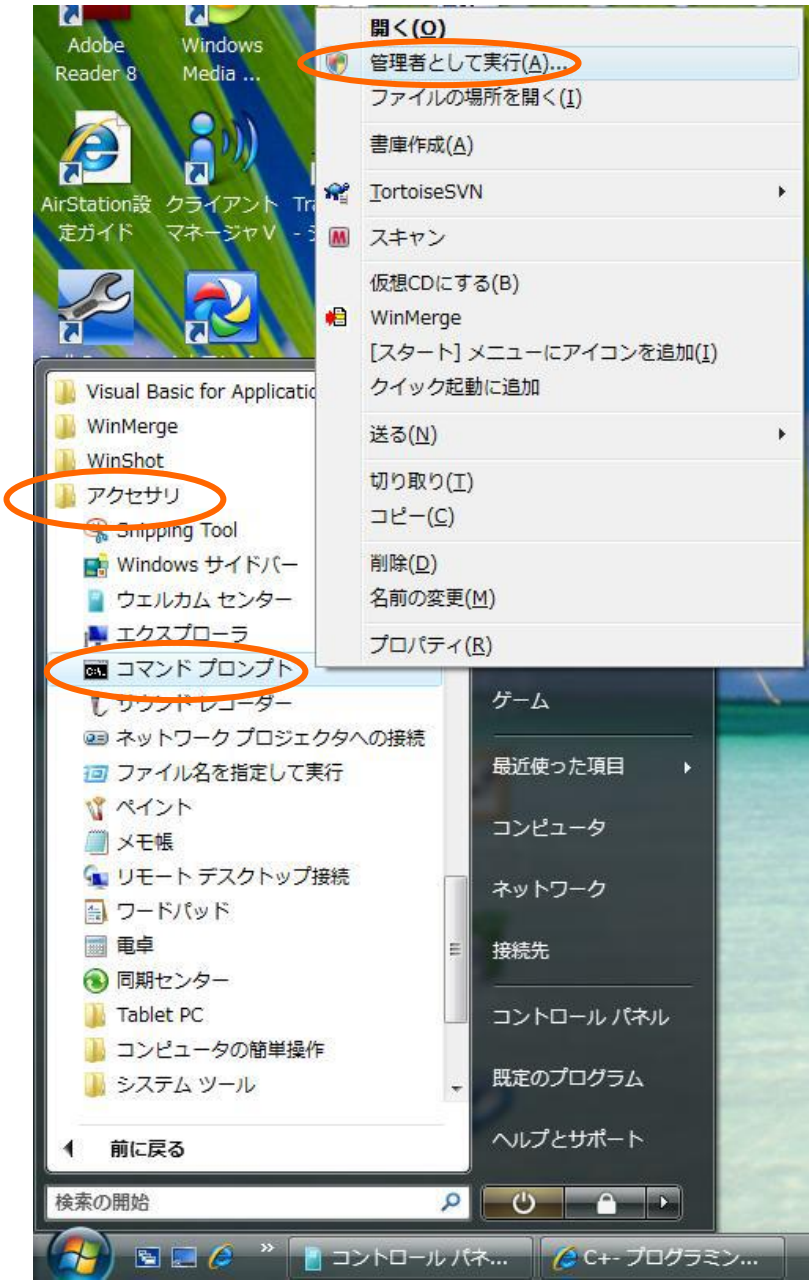
図 2-1 インストールの実行

これらの ActiveX コントロールの登録を解除したい場合は、付属の uninstall.bat を実行してください。
あとは Cmtoy.exe を実行するだけです。ActiveX コントロールが登録されていなかったり、登録後フォルダ名を変えたり、フォルダを移動すると Cmtoy.exe は起動しません（図 1-2 のウインドウは現われず警告音が 2 回鳴ります）。その場合は、正しい bin フォルダに移動し再度 install.bat を実行してください。

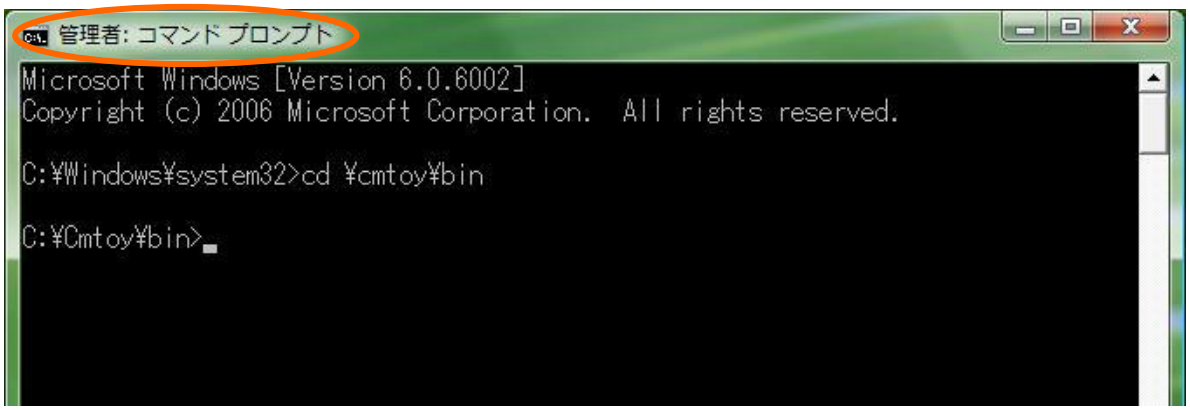
2.1.1 Windows Vista,Windows 7 でのインストール

Windows Vista、Windows 7 ではセキュリティが強化されたので、管理者モードでコマンドプロンプトを起動し install.bat で登録する必要があります。以下に管理者モードでコマンドプロンプトを起動する例を示します。

①Windows のスタートボタンをクリックして「すべてのプログラム」を表示します。
②「アクセサリ」を開きます。
③「アクセサリ」の中の「コマンドプロンプト」を右クリックすると以下のようにメニュが表示されます。

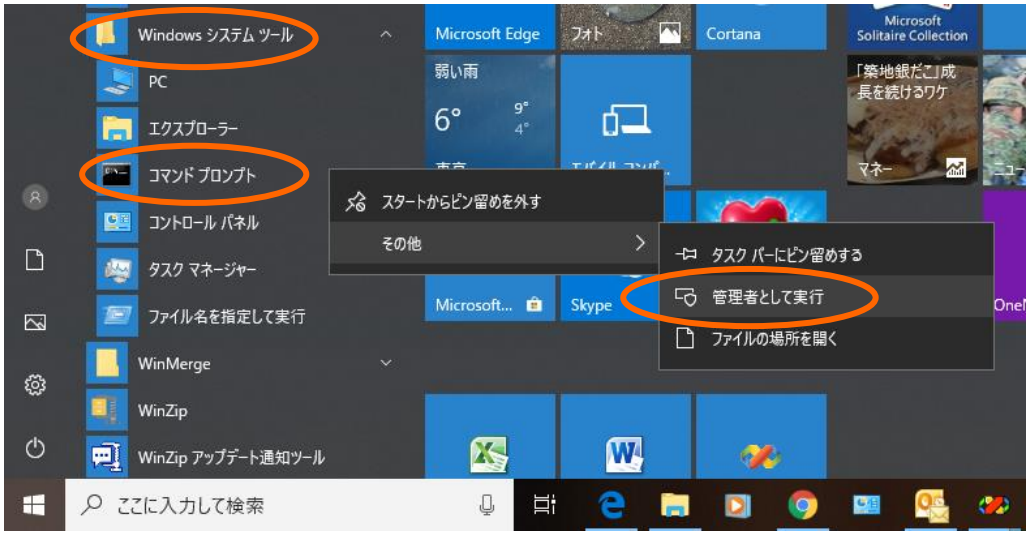


④ここで、「管理者として実行(A)...」をクリックすると、コマンドプロンプトが開きます。タイトルバーに「管理者：コマンドプロンプト」と表示されることを確認してください。



2.1.2 Windows10 でコマンドプロンプトを起動するには

Windows10 では、コマンドプロンプトは「Windows システムツール」の中にあります。ここから管理者モードで起動してください。以下のように選択します。



2.2 Cmtoy を起動する

インストールが済んだらフォルダ bin の中にある Cmyoy.exe を起動します。Cmtoy のコマンドライン形式は以下のとおりです。

Cmtoy [-c<作業ディレクトリ>] [-p<ポート番号>]

ここで、

<作業ディレクトリ>　スクリプトファイルを配置するディレクトリ。このディレクトリ内の cminit.cms を起動時に実行します。

<ポート番号>　シリアルコントロール serial.ocx が使う TCP/IP ポート番号。省略すると 700 となる。

Cmtoy.exe を実行すると、図 1-2 のウインドウが現われます。 この段階で cm.dll と kpdll.dll はロードされています。このとき、作業ディレクトリにある cminit.cms を探して、見つかればスクリプトファイルとして無条件に実行します。スクリプトファイルについては後で説明します。

Cmtoy が正常に起動した場合、出力ウインドウには以下のメッセージを表示します。

```
;μITRON トレーナ Cmtoy Ver 2.00
;cm.dll : Ver.01.00.0008
;kpdll.dll : Ver.01.00.0007
;作業ディレクトリ＝D:¥cmtoy-200¥bin
;シリアルで使用する TCP/IP ポート番号＝700
;cms: ▲ファイル "D:¥cmtoy-200¥bin¥cminit.cms" を開けませんでした。
```

Active-X コントロールが正しくインストールされていないと、警告音を 2 回鳴らして終了します。
Cmtoy のウインドウは表示されません。

2.2.1使用する TCP/IP ポート番号を変更する

Cmtoy はシリアルポートを TCP/IP を使ってシミュレートします。その際にシリアルポート制御コントロールごとに１つの TCP/IP ポート番号を割り当てます。デフォルトでは、

シリアル１　　TCP/IP ポート 700
シリアル２　　TCP/IP ポート 701

と割り当てます。
これを変更する場合は、Cmtoy を起動するときに渡すコマンド引数で行います。例えばポート番号 48557 を使用したい場合は、以下のように Cmtoy を起動します。

```
Cmtoy -p48557
```

このように起動すると

シリアル１　　TCP/IP ポート 48557
シリアル２　　TCP/IP ポート 48558

となります。

コマンド引数を Cmtoy に渡す２つの方法の例を示します。
①　コマンドプロンプトからコマンドラインで起動する

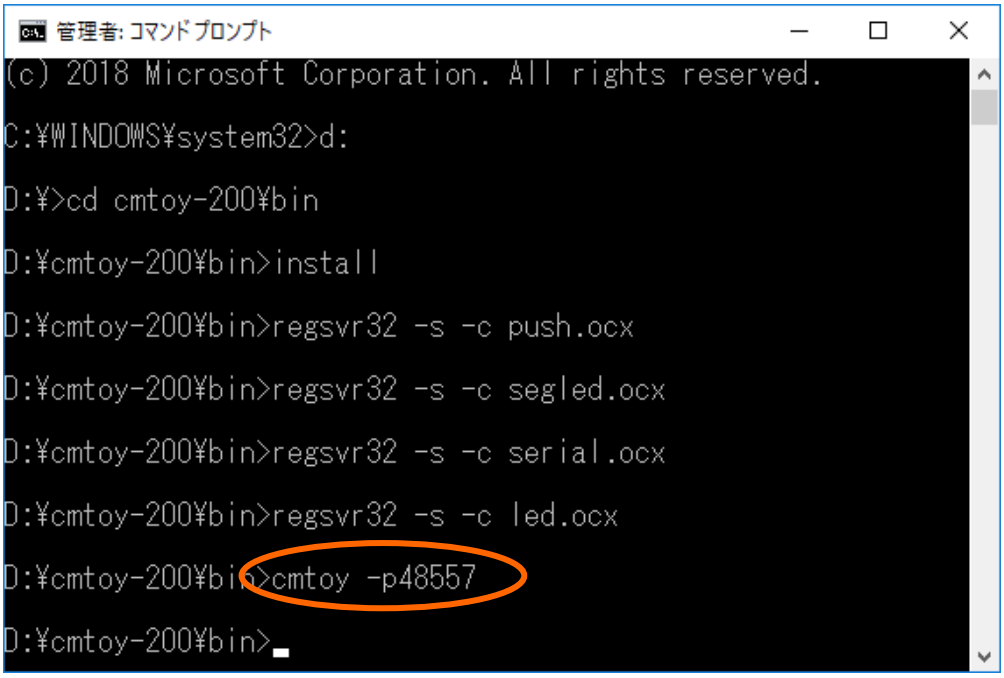


図 2-2 コマンドラインから Cmtoy を実行

② Cmtoy.exe のショートカットを作成し、ショートカットのプロパティを開き、ショートカットタブの「リンク先」で以下のように指定する
D:¥cmtoy-200¥Cmtoy.exe **-p48557**

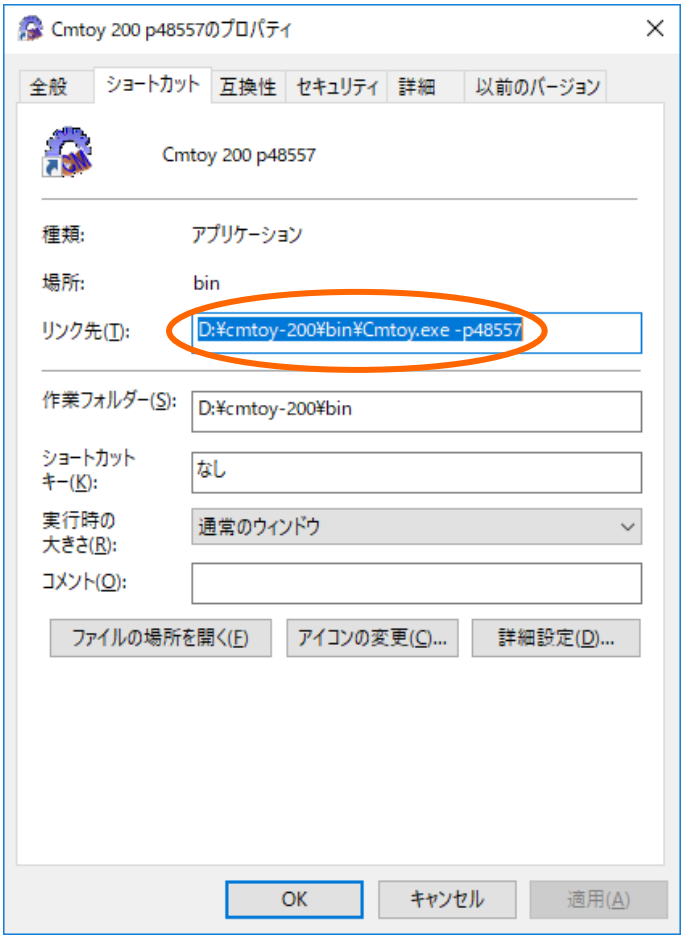


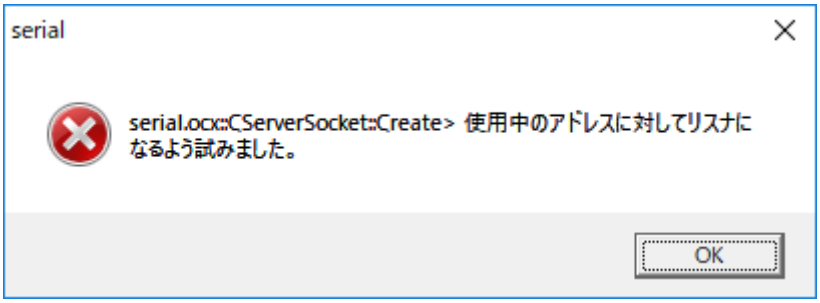
図 2-3 ショートカットのリンク先指定

デスクトップに以下のように複数のショートカットを作ることもできます。

※ TCP/IP のポート番号はすでに使用方法が決まっているものがあります。それらについては以下の The Internet Assigned Numbers Authority (IANA)を参照してください。
http://www.iana.org/assignments/port-numbers

※ シリアルコントロールとポート番号の使い方は、「7.2.4 ハイパーターミナルの設定方法」を参考にしてください。

※ Cmtoy 起動時に、ポート番号が使用中であれば以下のダイアログが表示されます。

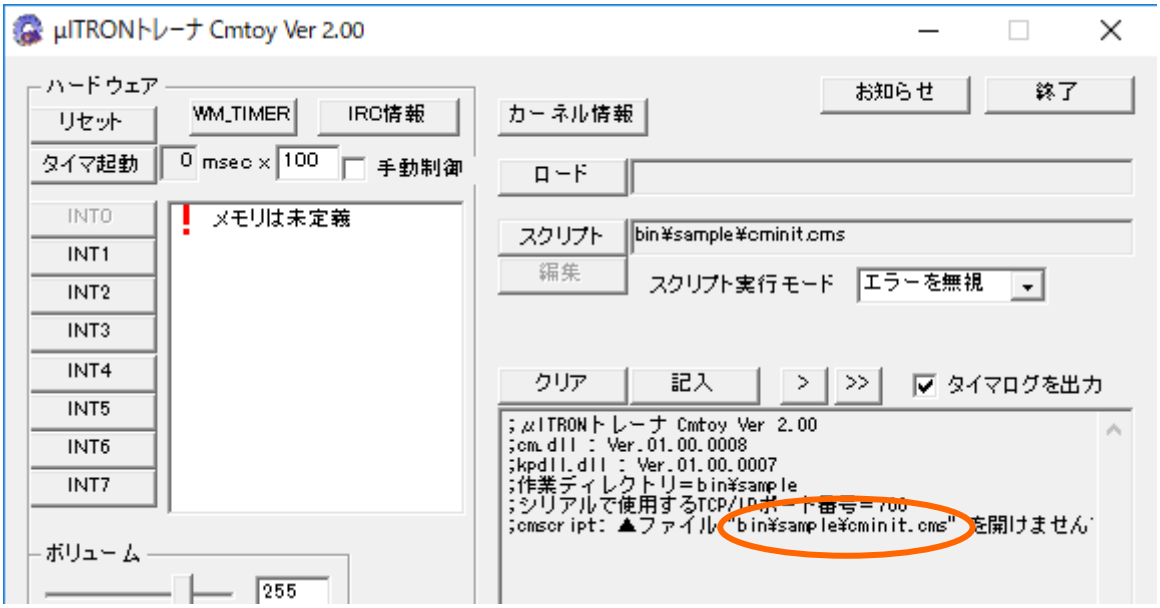


これが表示されても、serial.ocx 以外の機能は使用できます。GUI 上でシリアルコントロールには”error”と表示がでます。

2.2.2作業ディレクトリを変更する

作業ディレクトリを bin¥sample に変更するにはコマンドプロンプトから以下のように実行します。

この場合は、初期スクリプトファイルを bin¥sample から探します。Cminit.cms が見つからないと以下のようにメッセージを表示します。



もちろん、ショートカットからも同様のことができます。

2.3 アプリケーションプログラムを実行する

アプリケーションプログラムのロードと実行は以下の手順で行います。

① 「ロード」ボタンをクリックして、アプリケーションプログラムのファイル名（例えば app1.dll）を指定します。
② 「リセット」ボタンをクリックするとμITRON カーネルが実行を始めます。

「ロード」ボタンをクリックすると以下のダイアログボックスが表示されるので、アプリケーションの DLL を選択し「開く」をクリックします。ファイルの一覧にアプリケーション DLL が表示されないときは、「ファイルの場所」でフォルダを変更してください。

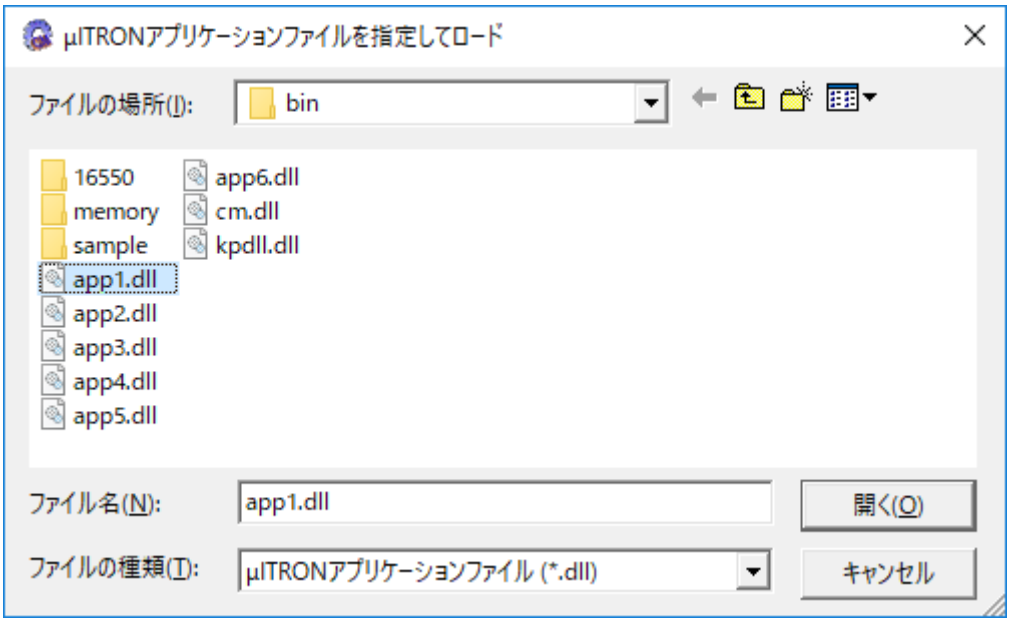


図 2-4 アプリケーションファイルの指定

正しくロードされると、「ロード」ボタンの横のテキストボックスにアプリケーションファイルのフルパス名を表示します。出力ウインドウには以下のメッセージを表示します。

```
> Load "D:¥cmtoy-200¥bin¥app1.dll"
kpdll: AttachItronApplication(02d91030H) が見つかりました.
;CM: アプリケーションプログラム(D:¥cmtoy-200¥bin¥app1.dll) をロードしました.
```

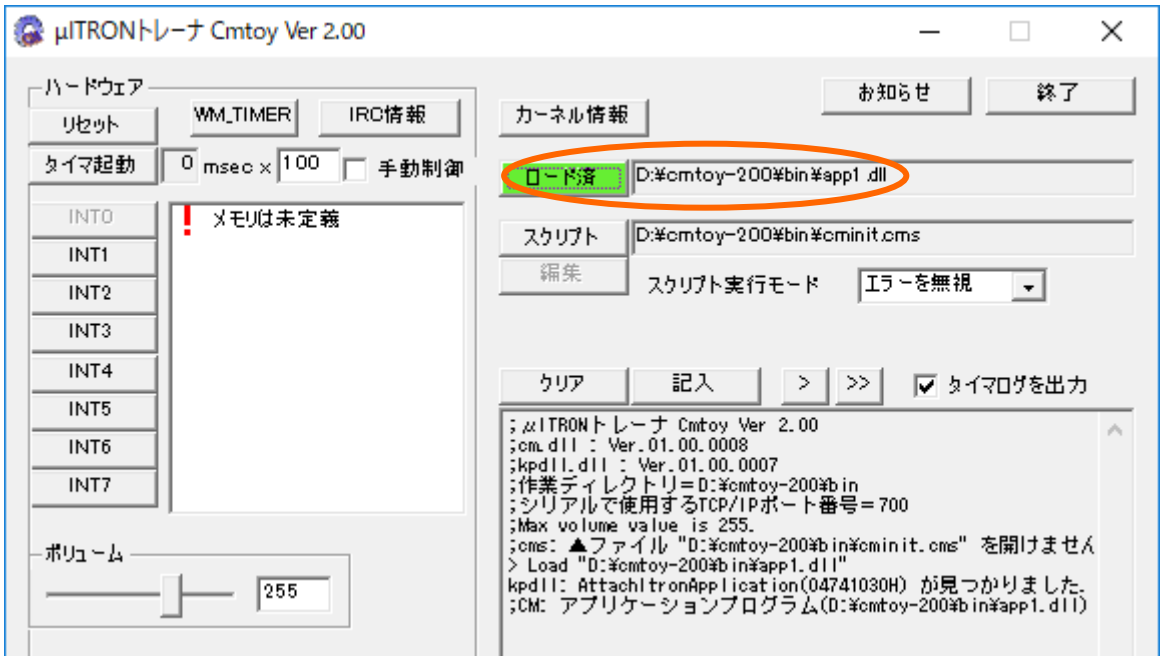


図 2-5 アプリケーションファイルのロード後

「リセット」ボタンをクリックするとカーネルが起動します。カーネルは初期化時にタイマを 10ms に設定し、アプリケーションプログラムのタスクを生成、起動します。
このとき実行ログが出力ウインドウに出力されます。タイマ割込み発生や「リセット」ボタンの操作などの GUI を操作したログの行頭には’>’の文字がつきます（図 2-6 を参照）。

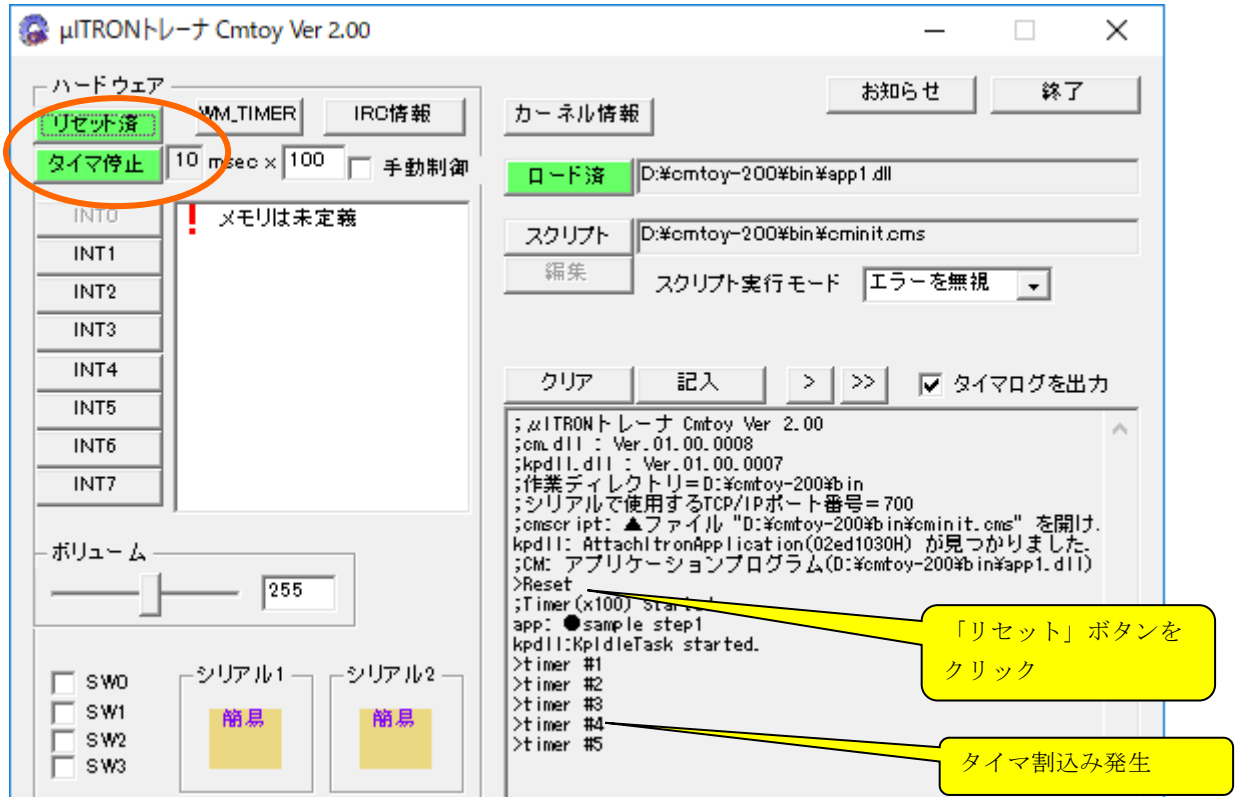


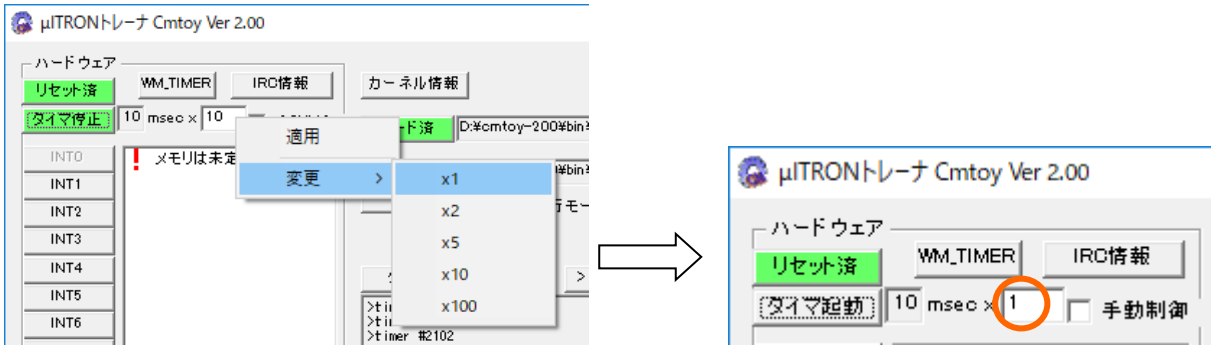
図 2-6 ターゲットプログラムが実行開始

2.4 インターバルタイマの操作

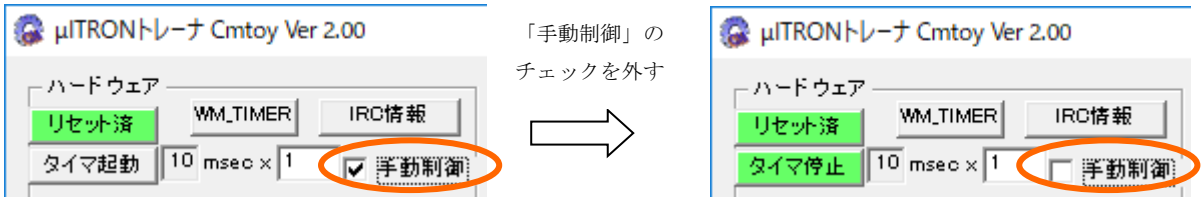
「リセット」ボタンをクリックすると、μITRON カーネルの初期化が始まります。
初期化時にインターバルタイマを 10ms に設定します。

インターバルタイマは定期的にレベル 0 の外部割込みを発生します。GUI の「タイマ起動」ボタンの表示が「タイマ停止」に代わり、背景色も変わります。
μITRON カーネルはこのレベル 0 の割込み回数で内部時刻をカウントします。Cmtoy では実際の割込み間隔は、カーネルの設定した 10ms を整数倍した値となります。Cmtoy 起動時は 100 倍となっているので実際の割込み間隔は 10msx100=1000ms=1 秒となります。この倍数を変えるにはテキストボックスで 100 を 1 ～10000 の範囲で変更して、マウスでテキストボックスの上で右クリックし、メニュが出たら「適用」をクリックします。適用をクリックするとタイマは停止してボタンの表示は「タイマ起動」となります。
インターバルタイマを再開するには「タイマ起動」ボタンをクリックします。

テキストボックスに入力せずに変更するには、右クリックしてメニュから「変更」→「x1」をクリックします。これで 10msx1=10ms の間隔になります。この場合もタイマは停止するので、インターバルタイマを再開するには「タイマ起動」ボタンをクリックします。



インターバルタイマからの割込みを連続でなく手動で 1 回づつ起こさせたい場合は、「手動制御」チェックボックスをクリックします。タイマ起動中ならボタン表示が「タイマ起動」に変わりタイマは停止します。ここでボタン「タイマ起動」をクリックすると 1 回だけレベル 0 の割込みが発生します。再び連続でインターバルタイマの割込みを発生させるには、「手動制御」チェックを外します。

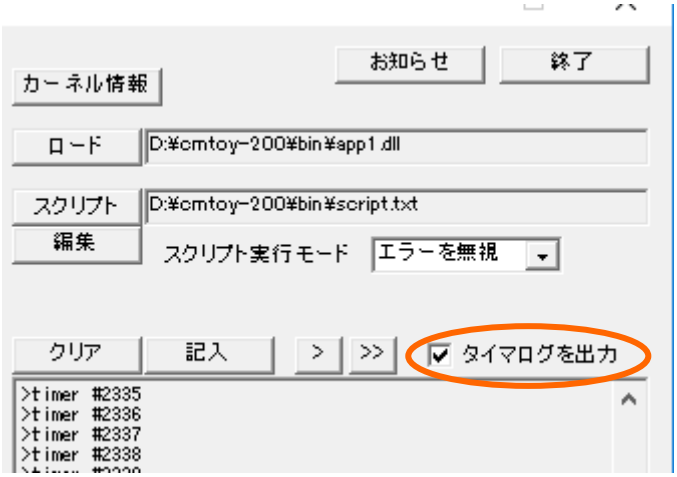


Cmyoy ではインターバルタイマの時間間隔は Windows の WM_TIMER イベントでシミュレートしているのでそれほど正確ではありません。そのため、「WM_TIMER」ボタンは WM_TIMER のイベントの保守用トレースを表示します。

インターバルタイマの割込みが発生すると出力ウインドウには以下のような文字列を表示します。ここで<n>はカーネルの割込みハンドラが起動した回数です。

```
>Timer #<n>
```

もしこの表示を止めたい場合は、出力ウインドウの「タイマログを出力」のチェックを外します。



2.5 外部割込みの操作

「リセット」ボタンをクリックしてアプリケーションを実行するとタイマイベントが自動的に発生します。この段階では、アプリケーションはインターバルタイマのみで動作しています。Cmtoy ではインターバルタイマはレベル０の外部割込みを使っています。

例えば、レベル１の外部割込みを手動で発生させるには「INT1」ボタンをクリックします。このとき出力ウインドウには、

```
>Int 1 #1
```

の行が出力されます。

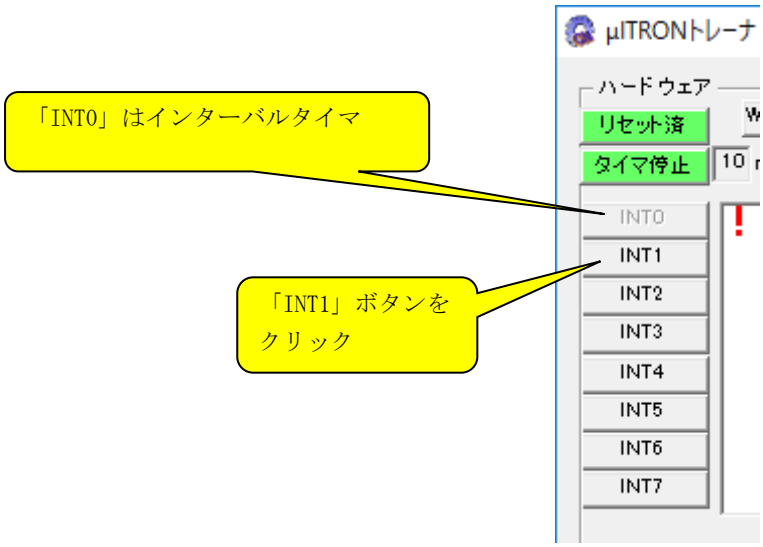
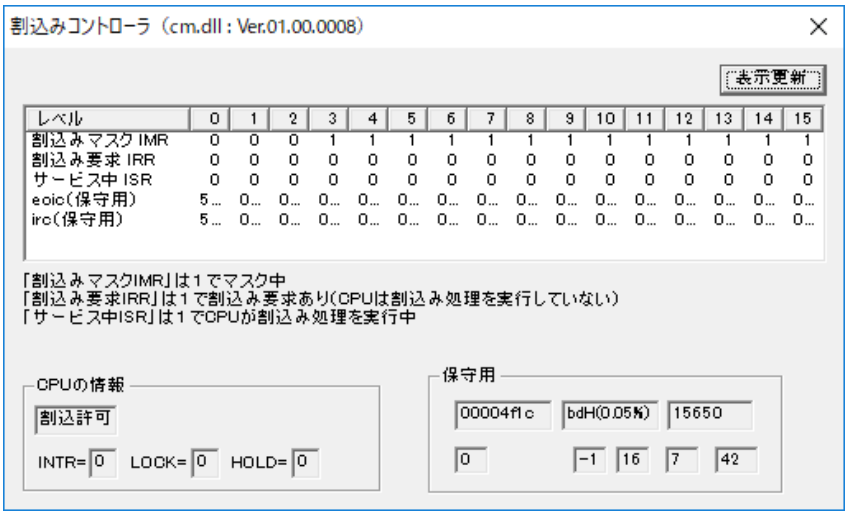


図 2-7 INT1 をクリック

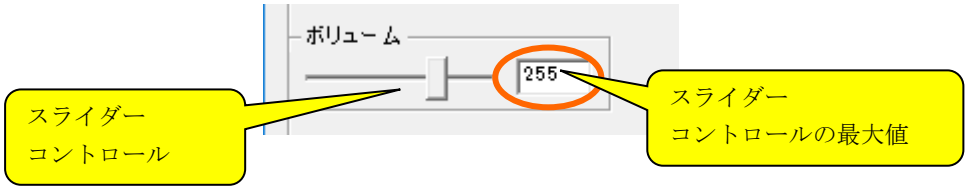
Cmtoy では 16 レベルの外部割込みを扱いますが、GUI から操作できるのはレベル０～７です。レベル０の操作は「2.4 インターバルタイマの操作」を参照してください。

Cmtoy の GUI ウインドウの「IRC 情報」ボタンをクリックしてすると、割込みコントローラ(IRC)のレジスタ状態を表示するウインドウが表示されます。「表示更新」ボタンをクリックすると以下のように最新のレジスタ状態のスナップショットが表示されます。



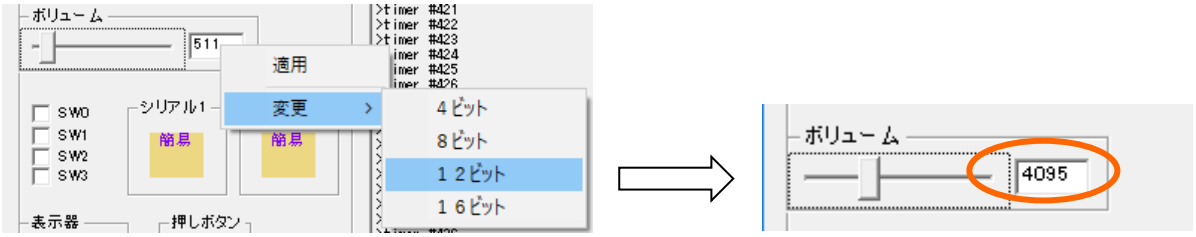
2.6 ボリュームの操作

以下のボリュームは A/D コンバータをシミュレートしています。



上記の場合、スライダーコントロールのハンドル位置で０～２５５の整数値が現在値となります。最大値を変えるには、テキストボックスに数値を入力し、右クリックして「適用」を選択します。

右クリックから表示される「変更」メニュからも直接変更できます。

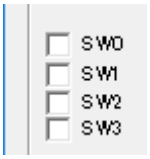


選択したビット数と最大値の関係は以下のようになります。

ビット数	最大値
4	15
8	255
12	4095
16	65535

2.7 DIP スイッチ

以下のチェックボックスで 4 個の DIP スイッチをシミュレートします。



DIP スイッチは ON/OFF どちらかの状態を取ります。マウスでチェックボックスを操作します。

2.8 押しボタン

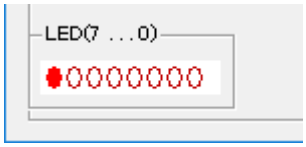
以下の GUI コントロールは押しボタンをシミュレートします。マウスの左ボタンで UP/DOWN の操作をします。操作していない時が UP で、このコントロール上でマウスの左ボタンを押している間 DOWN 状態となります。

UP

DOWN（マウスの左ボタンを押した状態）

2.9 表示専用 LED

8 個の LED ランプをシミュレートします。点灯すると赤丸に、消灯は白抜き丸になります。



7 セグメント LED を 2 個シミュレートします。

2.10 μITRON カーネルの状態を参照する

ターゲットプログラムである μITRON カーネル（kpdll.dll）も GUI を持っています。「カーネル情報」ボタンをクリックすると kpdll.dll の GUI（モードレスダイアログボックス）を呼び出します。この GUI ではタスク切替のトレース情報などのスナップショットを表示します。詳しくは「4.1.4 Cmtoy 固有の機能」を参照してください。

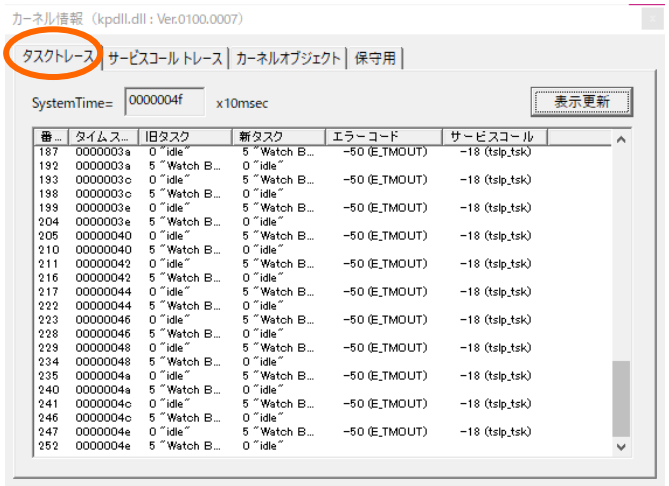


図 2-8 μITRON のタスクトレース情報

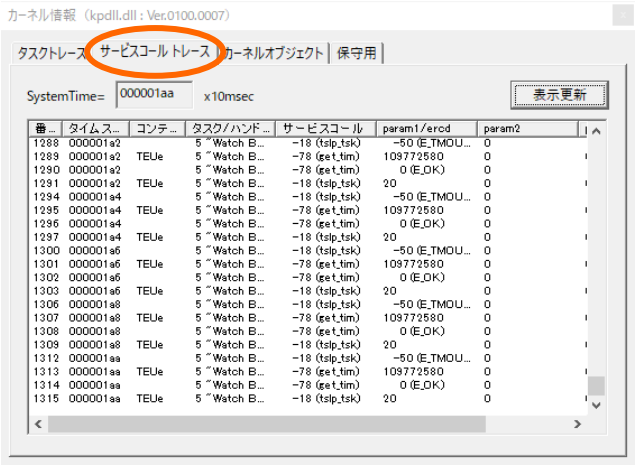


図 2-9 μITRON のサービスコールトレース情報

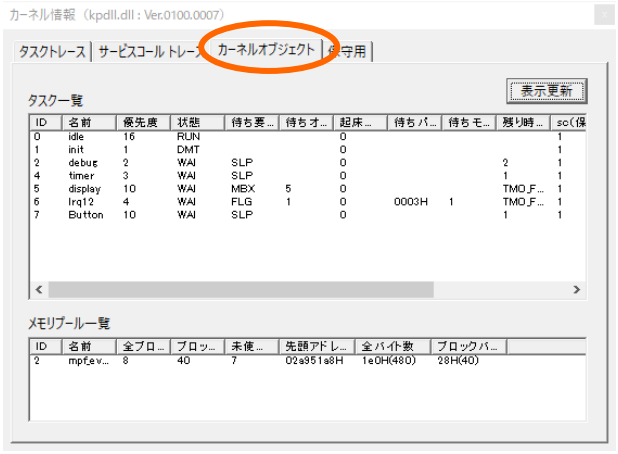


図 2-10 μITRON のオブジェクト情報

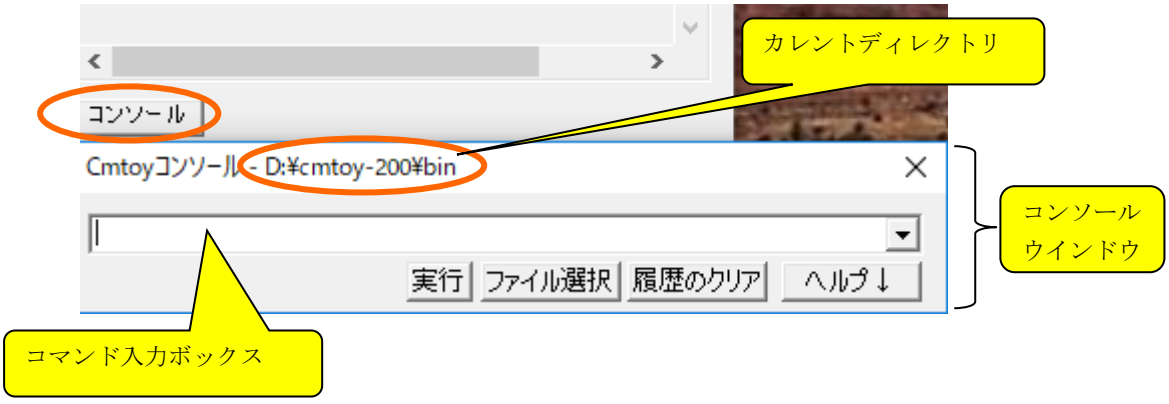
2.11コマンドラインによる操作

GUI をマウスで操作する代わりに、コンソールを開きコマンドラインから操作することができます。コマンドラインの一般形は以下のようになります。

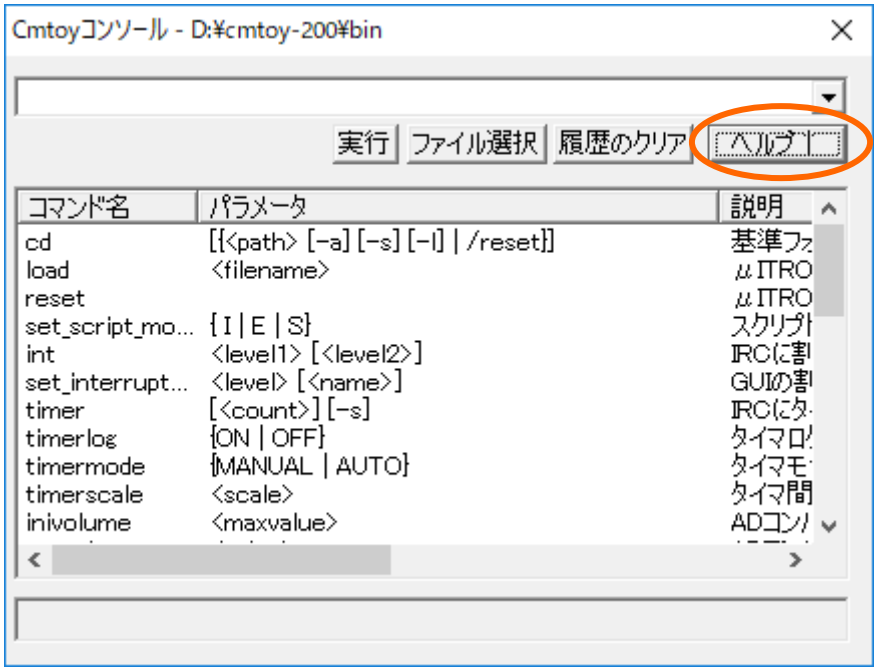
〈コマンド名〉 [〈パラメータ１〉[〈パラメータ２〉 …]]

・ コマンド名とパラメータ、パラメータ間は空白またはタブで区切ります。

コンソールを開くには GUI の「コンソール」ボタンをクリックします。すると、以下のようなコンソールウインドウが表示されます。タイトルバーにはカレントディレクトリが表示されます。

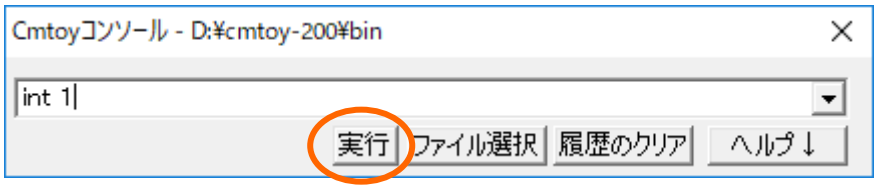


コンソールからコマンドラインを実行することで GUI からできない操作をすることができます。使用できるコマンドラインの一覧を見るには「ヘルプ」ボタンをクリックします。もう一度「ヘルプ」ボタンをクリックすると元の状態に戻ります。
ヘルプを表示させた状態でコマンド名をクリックするとコマンド名が入力ラインの先頭に入ります。

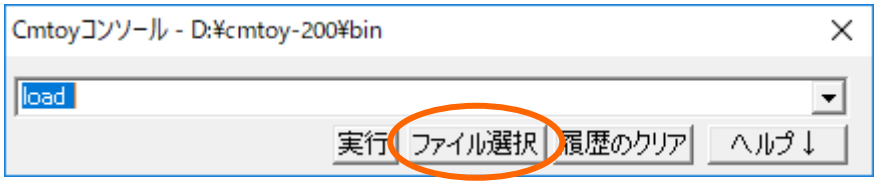


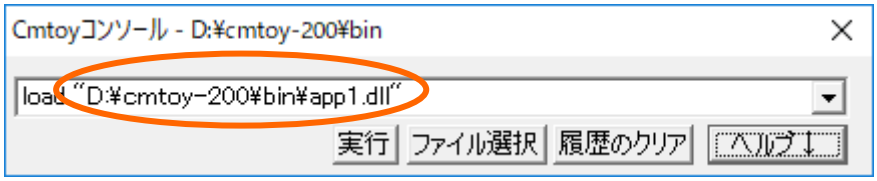
コマンドの詳しい説明は。「[６．コンソール・コマンド一覧]」を参照してください。
コンソールを隠すには右上の×ボタンをクリックします。再度表示させるには再度「コンソール」ボタンをクリックします。

例えば、割込み１を発生させるには、“int 1”と入力し、「Enter」キーを押すか、「実行」ボタンをクリックします。



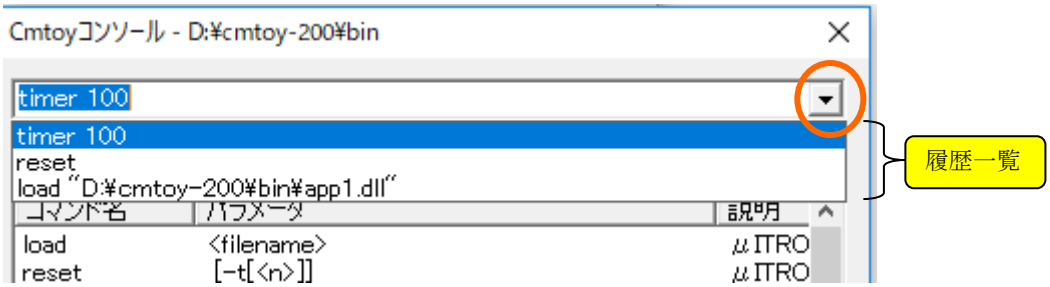
コマンドラインの中でファイル名を指定する箇所で「ファイル選択」ボタンをクリックするとファイルダイアログからファイルを指定することができます。例えば、load コマンドの次にファイル名を指定する場合キーボードからファイル名を入力してもいいですが、「ファイル選択」から指定することもできます。





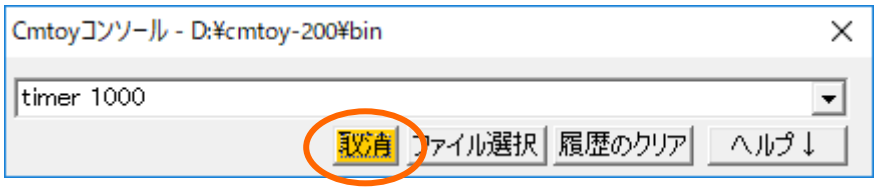
「ファイル選択」からファイルを指定すると、フルパス名が“”で囲まれて入ります。ここで「実行」ボタンをクリックするとコマンドが実行できます。

コンソールから実行したコマンドラインは履歴として残っているので、コマンド入力領域の右端の▼をクリックすると表示できます。



履歴の中からマウスで選択すると、そのコマンドラインがコマンド入力領域に入ります。
「履歴のクリア」ボタンで履歴をすべて削除できます。

コマンド実行中は、「実行」ボタンは「取消」となります。ここで「取消」ボタンをクリックするとコマンドを中断します。



2.12スクリプトによる操作

GUI をマウスで操作するか、コマンドラインから操作する代わりに一連の操作をテキストファイルとして作成しておき、そのファイルから毎回同じ操作を実行できます。いわゆるバッチ処理です。このテキストファイルを以後「スクリプト」ファイルと呼びます。

スクリプトはテキストファイルで、1 行に 1 コマンドを記述します。行頭に；（セミコロン）のある行はコメント（注釈）とみなします。

コマンドの一般形は以下のようになります。

〈コマンド名〉 [〈パラメータ 1〉[〈パラメータ 2〉 …]]

- コマンド名とパラメータ、パラメータとパラメータは空白またはタブで区切ります。
- 空白またはタブの直後の；（セミコロン）以降はコメントとみなします。

スクリプトを実行するには、「スクリプト」ボタンをクリックして以下のダイアログボックスからスクリプトファイルを指定します。
使用できるコマンドは、「6 コンソール・コマンド一覧」を参照してください。

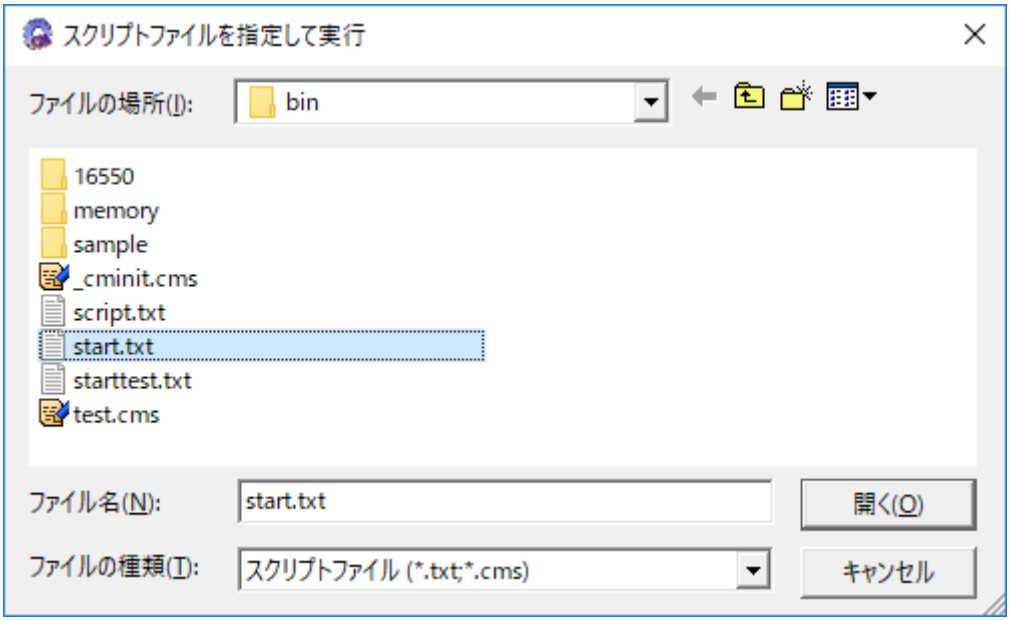


図 2-11 スクリプトファイルの指定

例えば、「2.3 アプリケーションプログラムを実行する」で説明した操作をバッチ処理するスクリプトファイル start.txt は、以下のようなテキストを含んでいます。

```
load app1.dll          ;（カーネルと）アプリケーションをロード
reset -t               ;カーネルを実行開始
```

以下に Cmtoy を起動して、start.txt を実行した様子を示します。この start.txt から読み込んだ行もすべて出力ウインドウに表示されます。先頭が’>’で始まる行がスクリプトファイルから読み込んだ行です（図 2-10 を参照）。

このスクリプトが正常に実行できた場合は、
- 「ロード」ボタンの横のテキストボックスにアプリケーションのフルパス名を表示
- 「スクリプト」ボタンの横のテキストボックスにスクリプトファイルのフルパス名を表示
- 「スクリプト」ボタンの下の「編集」ボタンをアクティブにする

ここで「編集」ボタンをクリックするとスクリプトファイルを Windows の「メモ帳」で開いて編集できます。

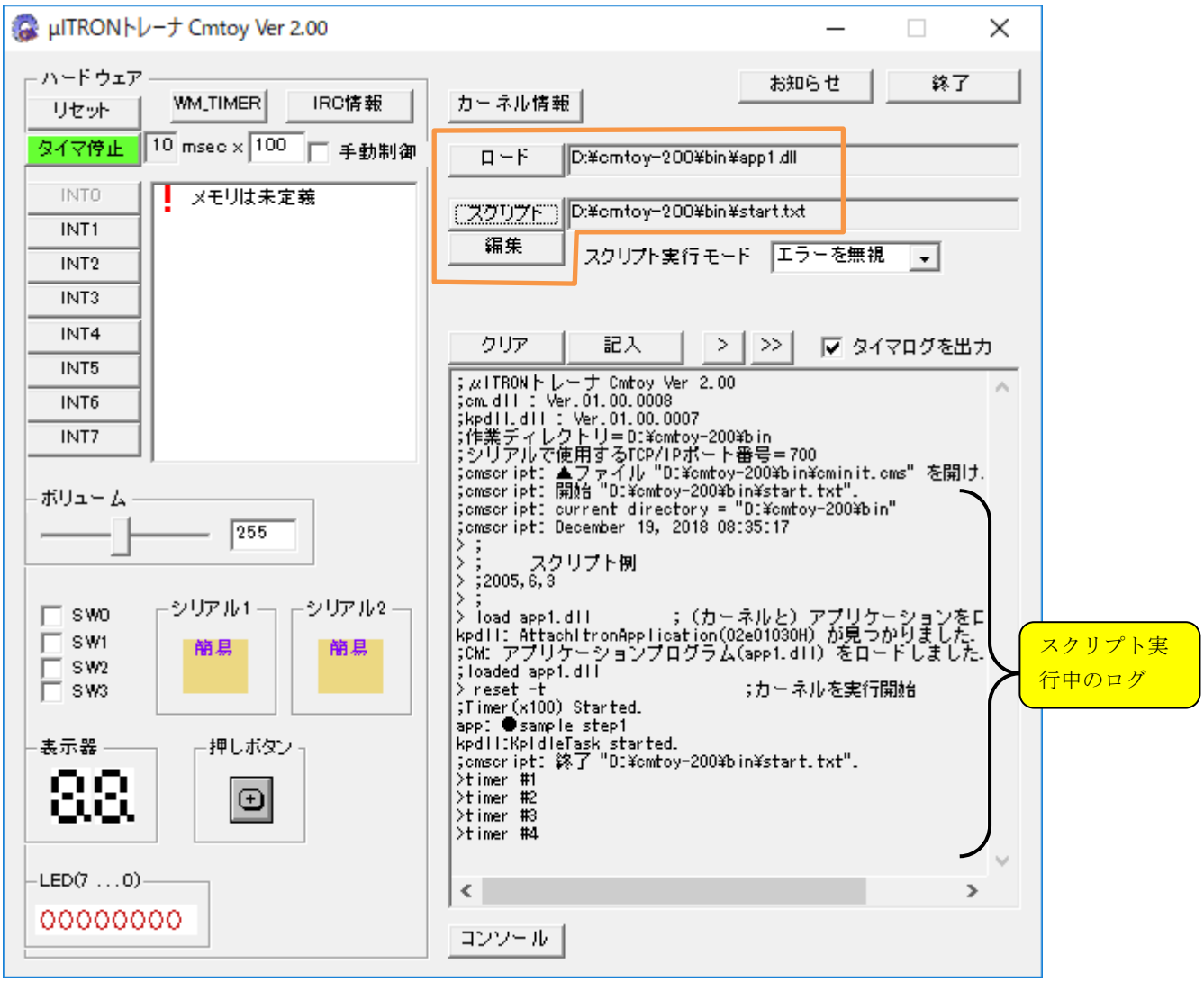
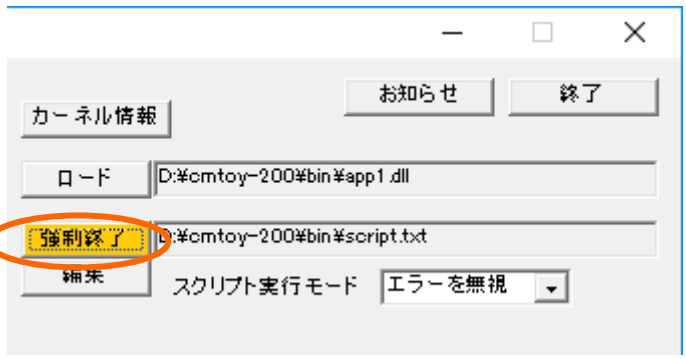


図 2-12 サンプルプログラムをスクリプトで実行

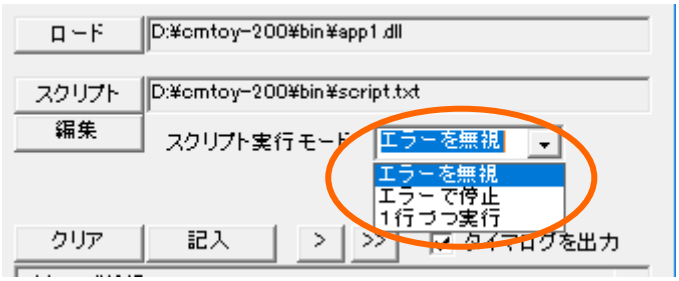
注）スクリプトから実行した reset の場合には、タイマを停止（手動制御に）します。「reset」ボタンをクリックした場合は、タイマは自動的にスタートします。スクリプトで reset –t とすればタイマを自動的にスタートします。

スクリプト実行中は「スクリプト」ボタンが「強制終了」に変わります。ここで「強制終了」ボタンをクリックするとスクリプトを中断します。

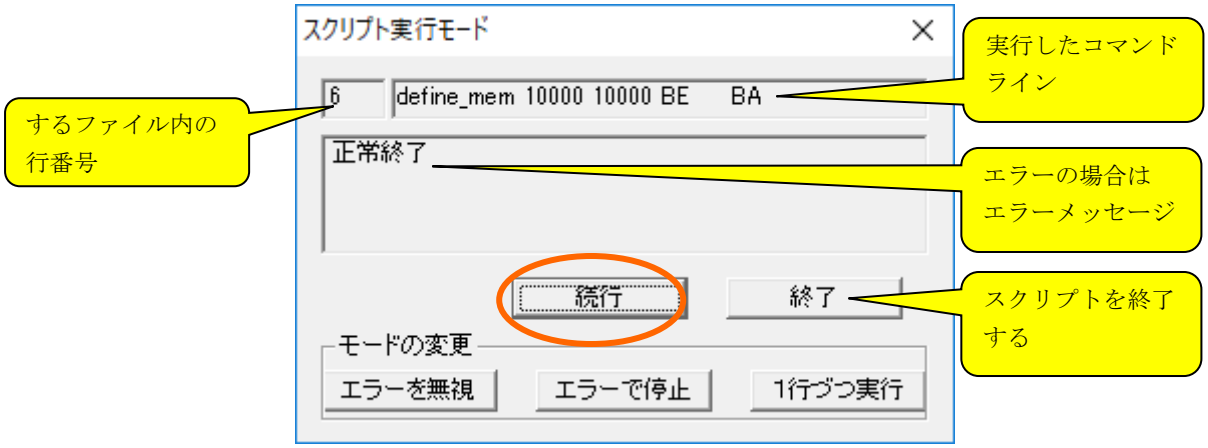


スクリプトの実行には３つのモードがあります。

① エラーを無視
② エラーで停止
③ 1 行づつ実行して停止



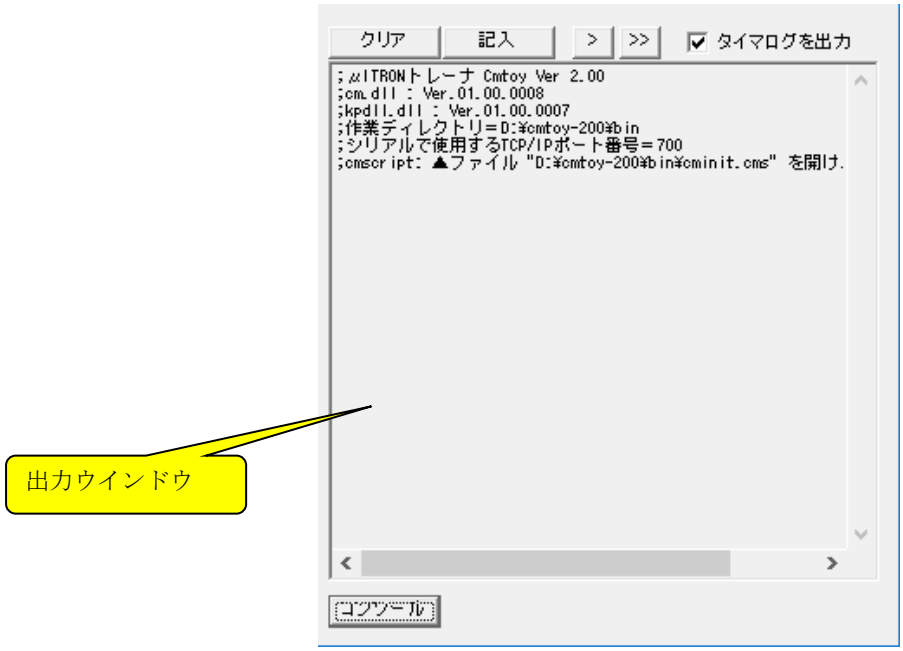
スクリプト実行モードを②または③に変更して実行すると、以下のようなダイアログを表示してスクリプトの実行を停止します。



ここで「続行」ボタンをクリックすると、
・ 「エラーで停止の」場合は、次にエラーを検出するまで実行を続けます。
・ 「1 行づつ実行」の場合は、次の行を実行したあと停止します。

2.13出力ウインドウ

Cmtoy は様々な情報を出力ウインドウに表示します。また、μITRON アプリケーションも情報を出力ウインドウに表示できます。



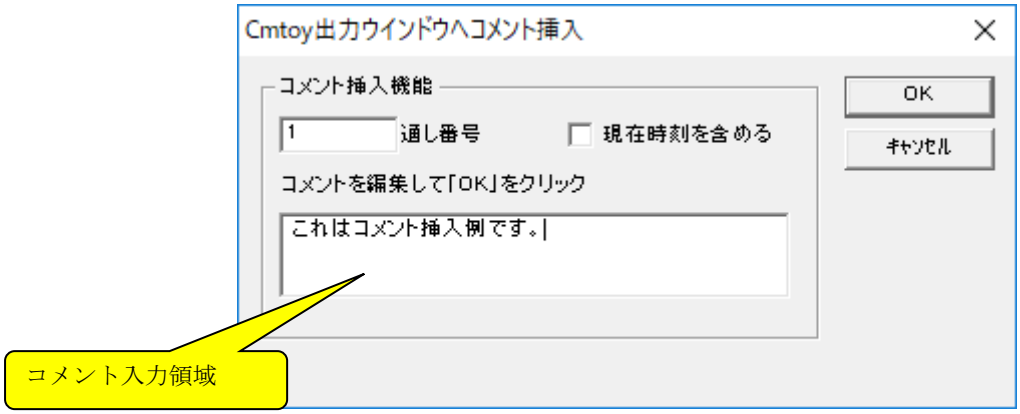
出力ウインドウに表示できる文字数には制限があります。約 30K バイトのバッファがあふれた場合は以下のメッセージを表示してそれ以降のメッセージを廃棄します。

<<<<バッファがあふれたので出力を廃棄します。クリアすると再開します。>>>>

ここで「クリア」ボタンでバッファを空にすると表示を再開します。

2.13.1記入

「記入」ボタンをクリックすると以下のようなダイアログが表示されるので、コメントを入力して「OK」とすると出力ウインドウにコメントを埋め込むことができます。



上記のようにコメントを入力すると以下のように出力ウインドウに表示されます。

2.13.2ファイルへ保存

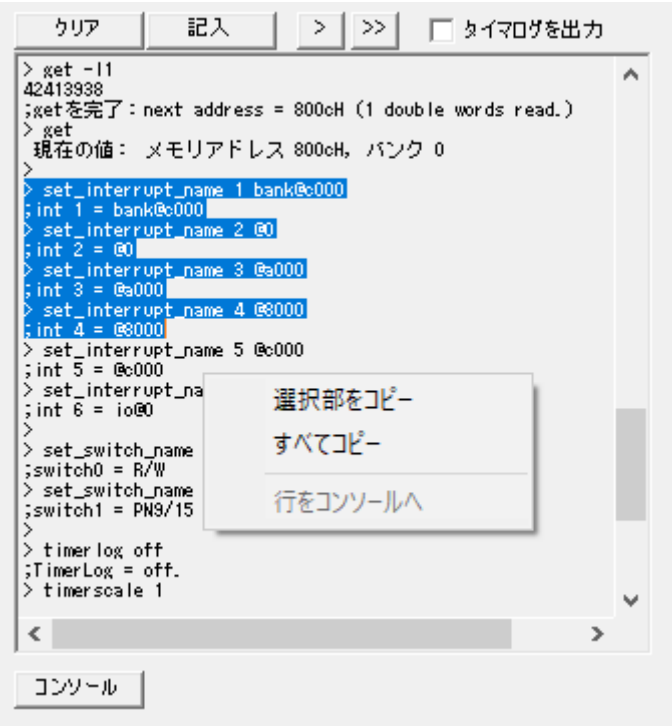
ボタン「＞」と「＞＞」は出力ウインドウの内容をファイルに書き出すときに使います。
「＞」は出力ファイルを指定して上書き、「＞＞」は指定したファイルに追加します。

μITRON アプリケーションからは以下の関数で出力ウインドウに文字列を表示できます。「5.2 デバッグ出力制御関数」を参照してください。

```
CM_EXTERN void halDebugPrintf(const char *formatstring, ...);
```

2.13.3コンテキストメニュー

出力ウインドウ上で右クリックすると以下のコンテキストメニューを表示します。



「選択部をコピー」は選択されているテキストをクリップボードへコピーします。
「すべてコピー」は出力ウインドウ内のテキストをすべて選択してクリップボードへコピーします。
「行をコンソールへ」は右クリックした行が‘＞’で始まる場合に、すでコンソールが開かれているとこの行を‘＞’を除いてコマンドコンソールの入力領域にコピーします。
コマンドコンソールについては「2.11 コマンドラインによる操作」を参照してください。

2.14ターゲットメモリ

C-Machine はターゲット CPU の物理アドレスに依存する RAM、EEPROM、メモリマップド IO、ポートマップド IO をシミュレートします。これらをターゲットメモリと呼びます。ターゲットメモリは、メモリ空間、IO 空間に分かれます。
・メモリ空間　　ターゲット CPU のメモリアドレス空間（RAM、EEPROM、メモリマップド IO など）
・IO 空間　　ターゲット CPU の　ポートアドレス空間

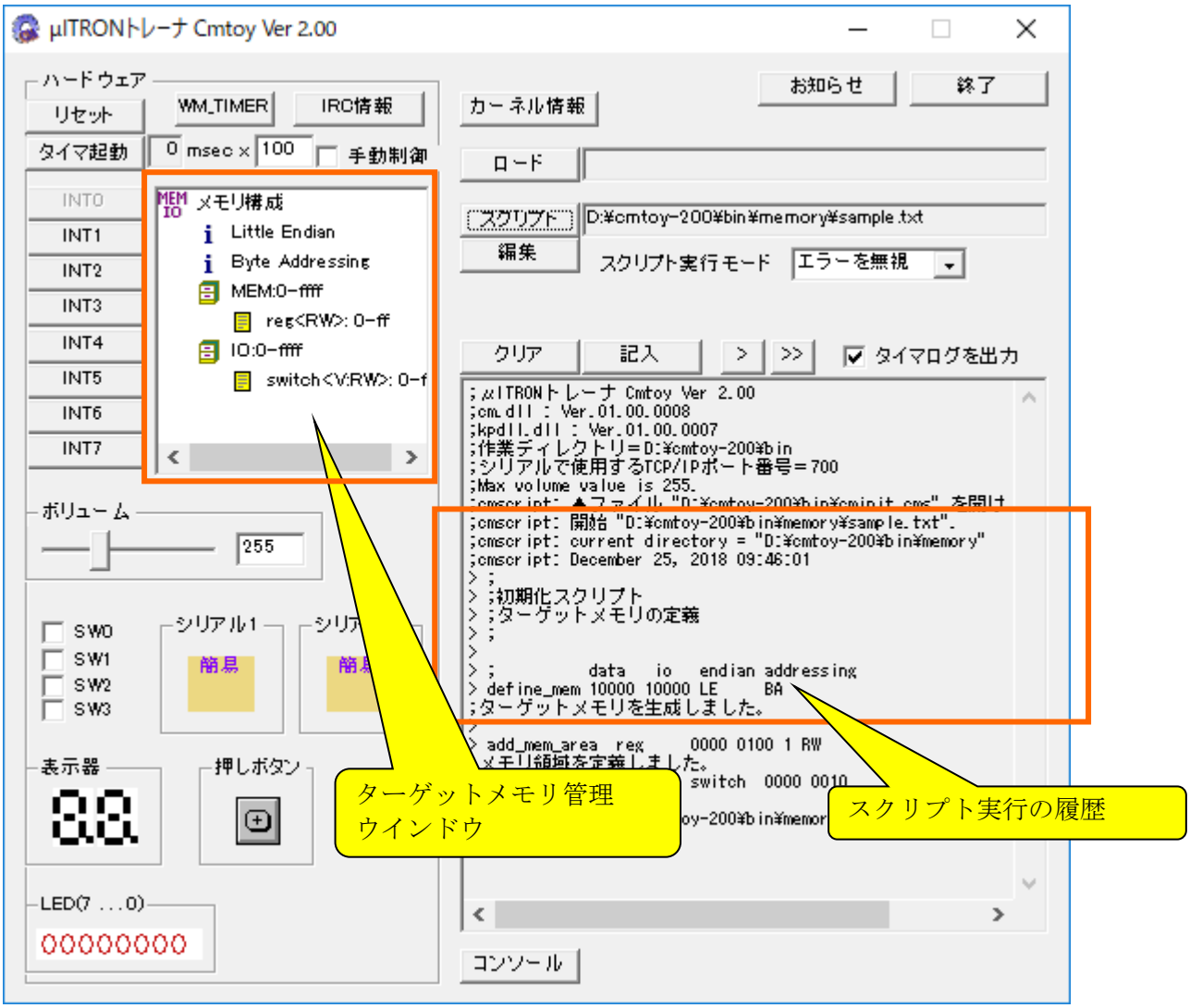
※ C 言語で記述したユーザプログラムのコード、データ、スタック領域は Windows により Cmtoy と同じユーザ空間に配置されるので、ターゲットメモリからは除外します。

C-Machine はターゲットメモリを Windows のプロセス空間内に確保してシミュレートするので、ターゲット CPU の物理アドレスと Windows のプロセス空間内のアドレスを対応付けます。そのためユーザプログラムでこの領域にアクセスするための、C 言語の関数やマクロを提供します。

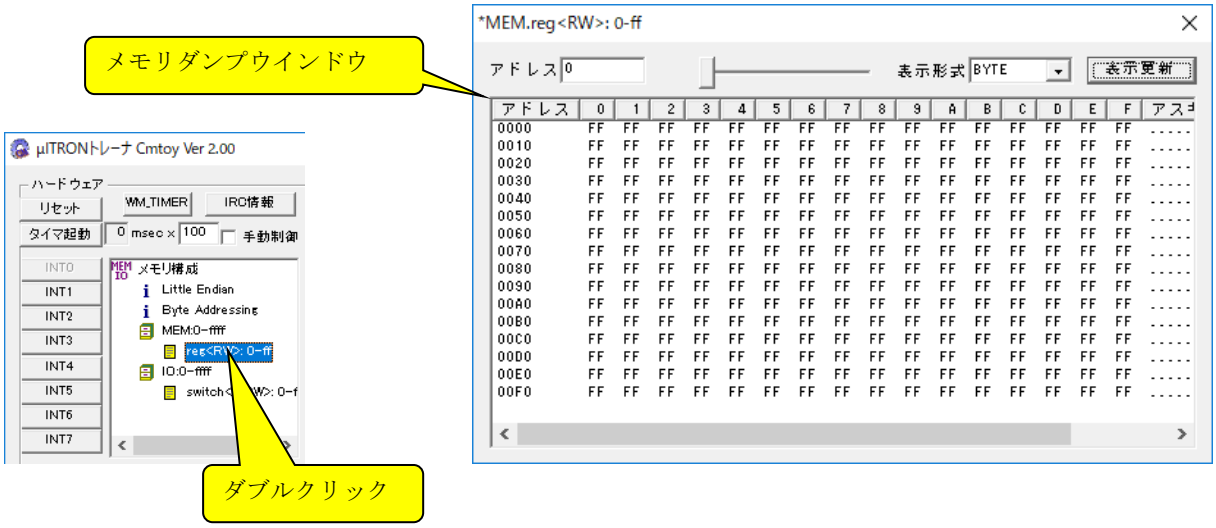
ターゲットメモリは、コマンドライン機能を使って定義します。「6.16 ターゲットメモリ操作」を参照してください。
例えば、16 ビット CPU、バイトアドレッシング、リトルエンディアンの RAM アドレス 0x0000-0x00ff、IO ポート 0x0000-0x000f のターゲットメモリを定義する場合は、以下のコマンドを実行します。

```
define_mem          100000 10000 LE  BA
add_mem_area        reg    0000 0100 1 RW
add_io_area         switch 0000 0010
```

これを実行すると、GUI は以下のようになります。

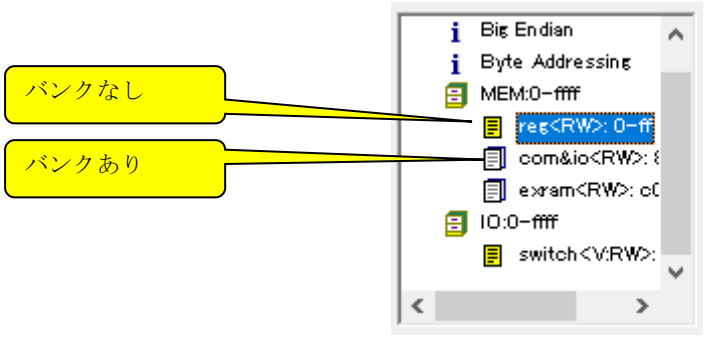


定義したメモリ、IO の内容を見るにはメモリ管理ウインドウの該当箇所をダブルクリックします。
するとメモリダンプウインドウが現れます。



以下のようなバンク指定を含むコマンドを実行した場合の例を示します。

```
;           data  io     endian addressing
define_mem 10000 10000 BE       BA
add_mem_area  reg    0000 0100 1 RW
add_mem_area  com&io 8000 4000 2 RW
add_mem_area  exram  c000 4000 10 RW
add_io_area   switch 0000 0010
```



バンクあり／なしでアイコンが変わります。

3 アプリケーションプログラムの作成とデバッグ

3.1 アプリケーションプログラムの作成方法

アプリケーションプログラムは、C 言語で記述し Windows の 32 ビット DLL(Dynamic-Link Library)形式の実行モジュールとして作成します。ソースファイル（*.c）には Cmyoy¥include ディレクトリにある以下のヘッダファイルをこの順番にインクルードしてください。

```
hal.h
hal_uart.h        //シリアル機能を使わなければ必要ない
itron.h
```

アプリケーションプログラムの情報をカーネルへ教えるために kernel_cfg.c を作成し、コンパイル、リンクして実行モジュールに加える必要があります。kernel_cfg.c には、Cmtoy¥include ディレクトリにある以下のヘッダファイルをこの順番にインクルードしてください。

```
hal.h
itron.h
kernel_cfg.h
kernel_id.h
```

kernel_cfg.c では、itron.h で定義されているマクロ（静的 API）を使いオブジェクトの登録をします。

- タスクの生成定義　CRE_TSK
- セマフォの生成定義　CRE_SEM
- イベントフラグの生成定義　CRE_FLG
- メイルボックスの生成定義　CRE_MBX
- 固定長メモリプールの生成定義　CRE_MPF
- 割込みハンドラの登録　DEF_INH
- 初期化ツーチンの登録　ATT_INI

Cmtoy では、これらの静的 API は C 言語のマクロであり itron.h で定義されています。以下の点に注意してください。

- 各オブジェクトに名前を割り当てるパラメータを追加している。
- タスク生成定義で指定するスタックサイズには０、スタックアドレスには NULL を指定する。

※Visual Studio を使うと簡単に DLL 形式の実行モジュールを作成できます。
※Cmtoy は 32 ビットアプリケーションなので、32 ビット DLL としか連携できません。

3.1.1コンフィギュレーション

Cmtoy では、タスク、割込みハンドラを C 言語で記述します。標準 C 言語にはタスク、割込みハンドラという概念はなく、実行は main()関数から始まるという約束になっています。例えば Windows や UNIX 上の C 言語で作成されたコンソールアプリケーションの実行モジュールはオペレーティングシステム（OS）により、メモリにロードされ main()関数が呼び出されます。プログラムの実行という視点からは C 言語はシングルタスクのプログラムを記述するといえます。一方μITRON では複数のプログラム（タスク）が独立に、論理的には並列実行されます。また、割込みハンドラの関数は、タスクの実行とは関係なく（非同期に）呼び出されます。したがって、並列実行される関数、非同期に呼び出される関数を指定する方法が必要となります。このμITRON カーネルに並列実行されるタスクと非同期に呼び出される割込みハンドラ（どちらも C 言語の関数）を教えてやる作業を「コンフィギュレーション」と呼びます。

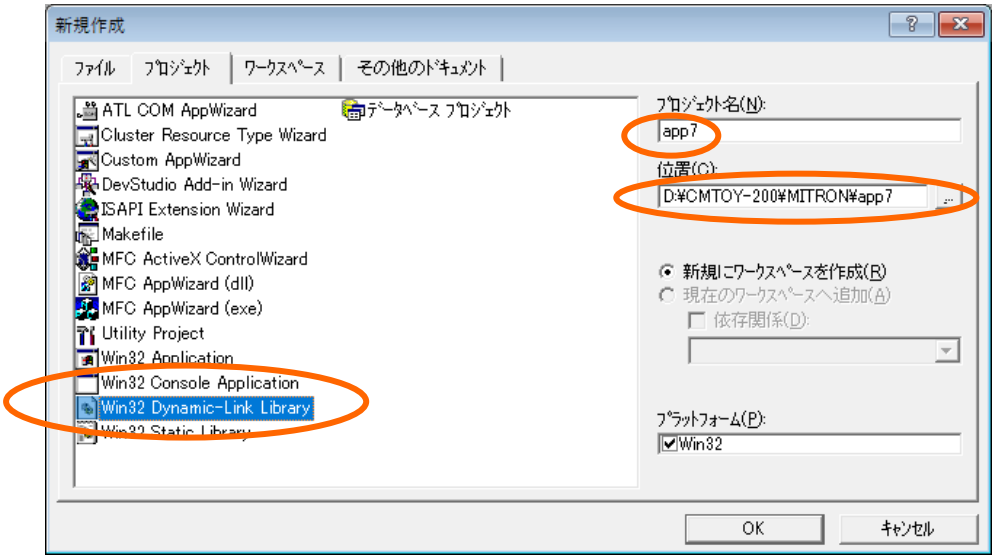
さらに、タスク間および割込みハンドラ・タスク間で同期、通信を行う場合は、μITRON の定義する同期、通信オブジェクトを使用します。これらのオブジェクトを定義することもコンフィギュレーションで行います。

Cmtoy では、コンフィギュレータと呼ばれるようなツールを使わずに直接 C 言語のソースファイルでタスク、割込みハンドラ、オブジェクトの一覧表を作成し、ユーザプログラムに組み込んで DLL 形式の実行モジュールを作ります。Cmtoy でのでコンフィグレーションは C 言語でこの一覧表を作ることです。C 言語でこの一覧表を作るには静的 API（実体は C 言語のマクロ）を使用します。

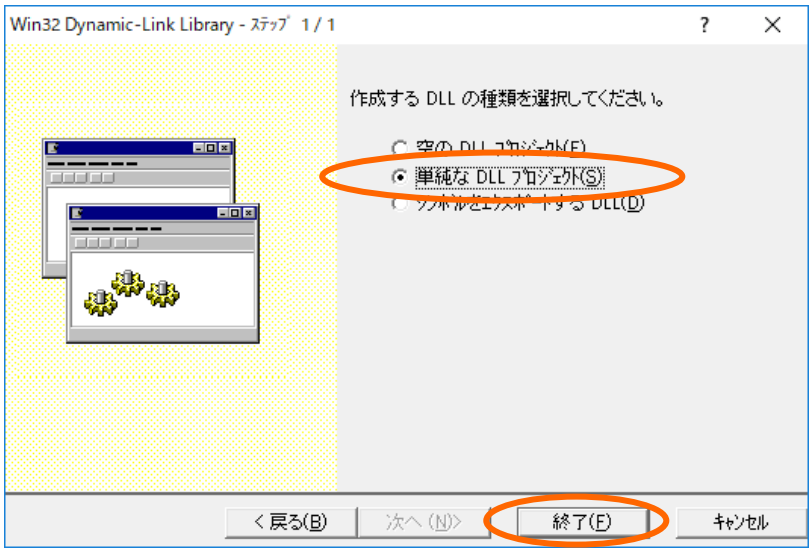
※μITRON アプリケーション内では main 関数はほかの関数と同列なので、なくても問題になりません。最初に実行される関数はコンフィグレーションで指定されたタスク関数です。

3.1.2 VisualStudio6.0 を使う

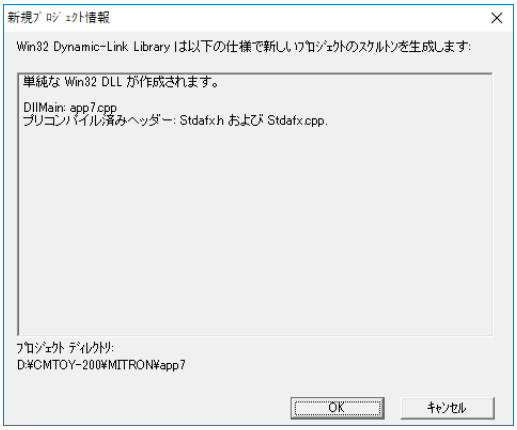
VisualStudio6.0 で DLL プロジェクトを作成する場合は、「ファイル」メニュの「新規作成（N）…」を選択します。プロジェクト作成ウィザードが起動し以下のウインドウが表示されるので、プロジェクトの種類として「Win32 Dynami-Link Library」を選択します。位置とプロジェクト名を指定して「OK」をクリックします。



次に、以下の「単純な DLL プロジェクト」を選択し、「終了」をクリックします。



次に以下の「OK」をクリックします。



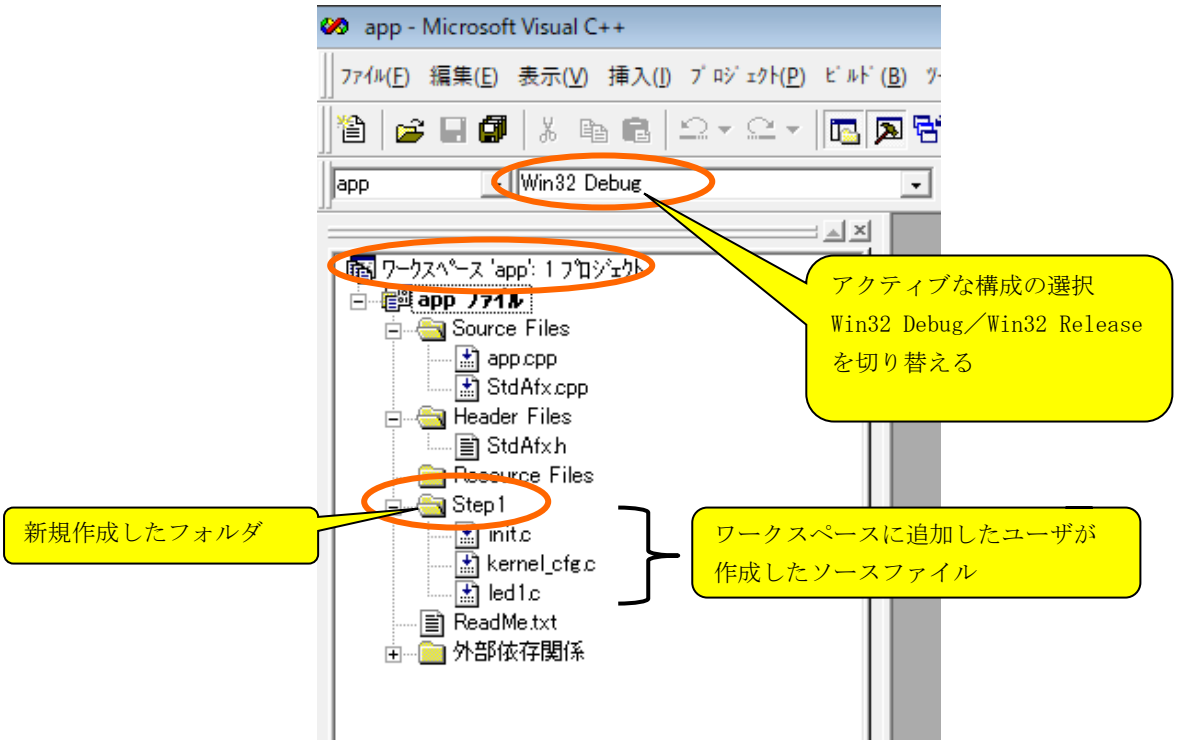
これで DLL プロジェクト・ワークスペースが作成されました。ウィザードはフォルダ app7 の下に以下のファイルを自動生成します。

app7¥

StdAfx.h	windows.h を include
StdAfx.cpp	StdAfx.h を include
app7.cpp	DllMain を含む
app7.dsp	プロジェクトファイル
app7.dsw	ワークスペースファイル

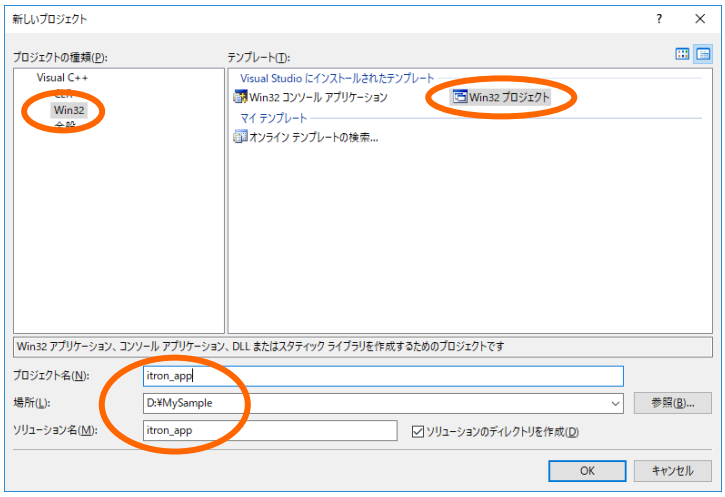
ここで、プロジェクトのプロパティから追加のインクルードパス、ライブラリパス、ライブラリファイル、プリプロセッサ定義を登録します。詳細は「3.2 ビルド方法」を参照してください。
その後、μITRON アプリケーションの C 言語のソースファイル(*.c)、ヘッダファイル(*.h)をプロジェクトに追加します。Visual C++コンパイラは、拡張子*.c のファイルは C 言語のソースファイルとしてコンパイルします。

例えば、「7.チュートリアル」で示すステップ 1 のワークスペースファイルを VisualStudio6.0 でで開くと以下のようになります。

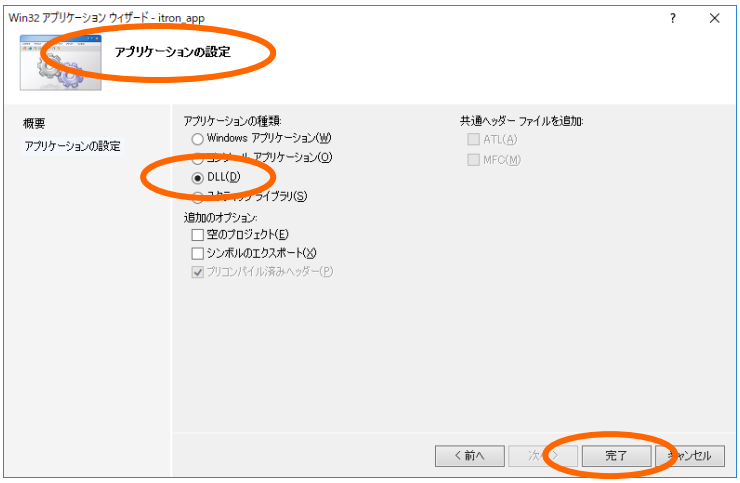


3.1.3 Visual C++ 2008 Express Edition を使う

新たに dll を作成する場合は、ファイルメニュの「新規作成」→「プロジェクト（P）」を選択し、プロジェクトの種類から Win32 をテンプレートとして「Win32 プロジェクト」を選び、場所とプロジェクト名を指定します。



「OK」をクリックすると、「Win32 アプリケーション ウィザードへようこそ」のウインドウが表示されるので「次へ>」をクリックします。すると「アプリケーションの設定」が表示されます。ここで「DLL（D）」を選択して完了します。



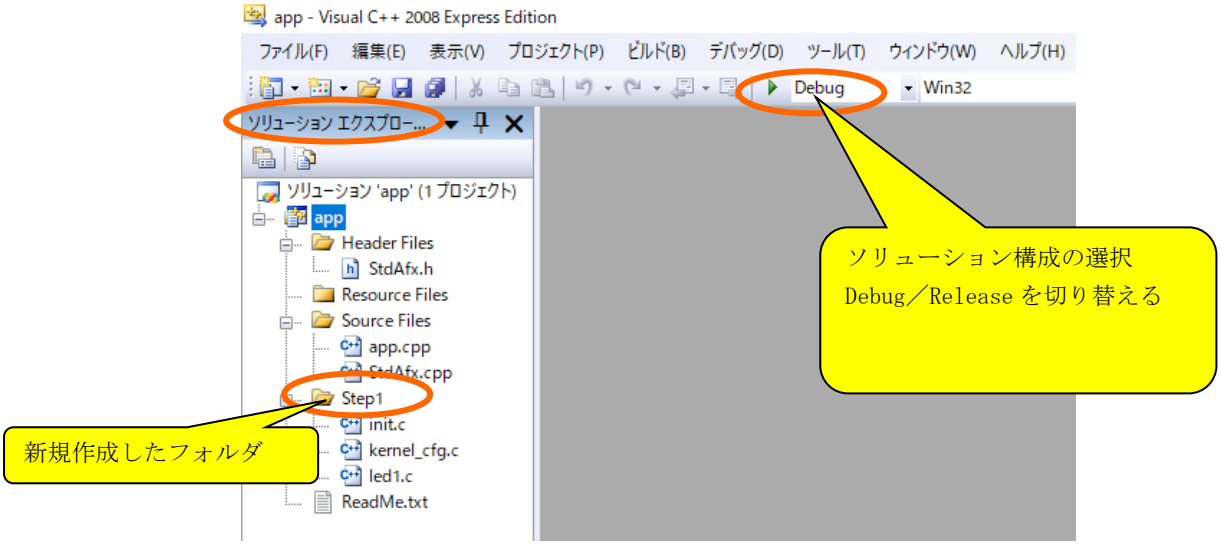
このように、itron_app というプロジェクトを作成すると、以下のファイルが自動的に作成されます。

```
¥itron_app
    ¥itron_app
        stdafx.cpp
        dllmain.cpp
        itron_app.cpp
        stdafx.h
        targetver.h
        itron_app.vcproj      プロジェクトファイル
    itron_app.sln             ソリューションファイル
```

ここで、プロジェクトのプロパティから追加のインクルードパス、ライブラリパス、ライブラリファイル、プリプロセッサ定義を登録します。詳細は「3.2 ビルド方法」を参照してください。
その後、μITRON アプリケーションの C 言語のソースファイル(*.c)、ヘッダファイル(*.h)をプロジェクトに追加します。Visual C++コンパイラは、拡張子*.c のファイルは C 言語のソースファイルとしてコンパイルします。

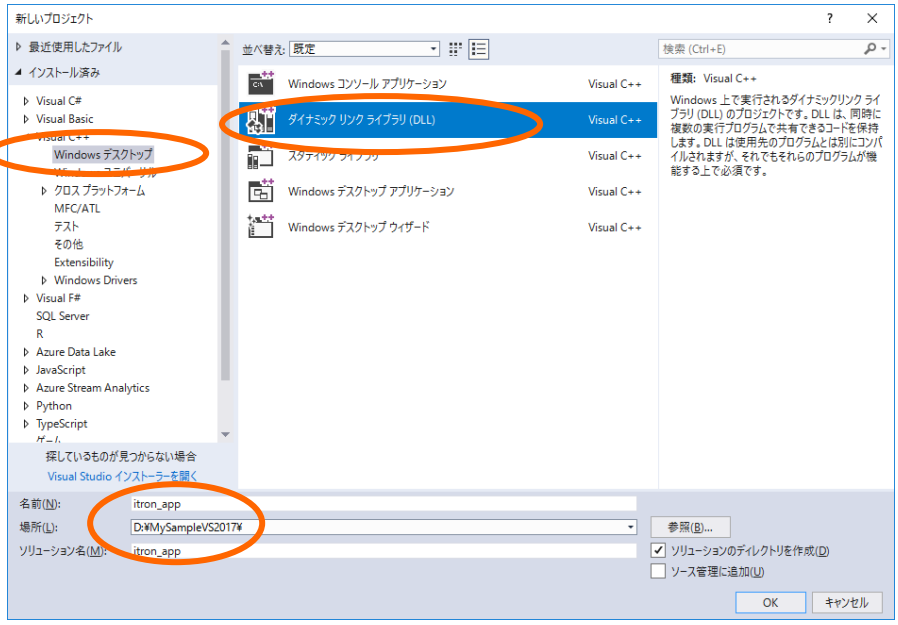
※ Visual C++ 2008 Express Edition では、Platform SDK を別途インストールする必要はないようです。
※ Visual C++ 2008 Express Edition では、32 ビット DLL プロジェクトのみです。

例えば、「7.チュートリアル」で示すステップ 1 のソリューションファイルを Visual C++ 2008 Express Edition でで開くと以下のようになります。



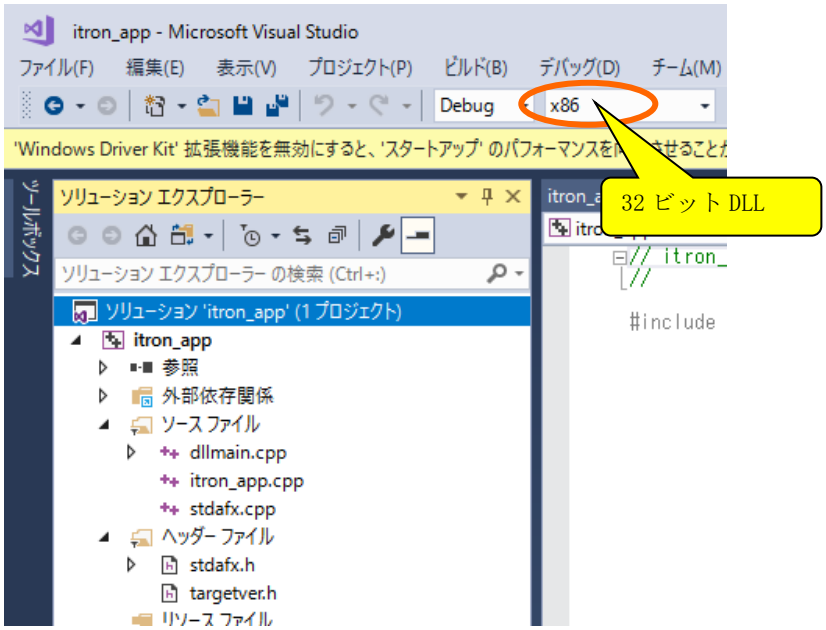
3.1.4 Visual Studio 2017 を使う

新たに dll を作成する場合は、ファイルメニュの「新規作成」→「プロジェクト（P）」を選択し、「VisualC++」の中から「Windows デスクトップ」を選び、その中の「ダイナミックリンクライブラリ(DLL)」を選び、場所とプロジェクト名を指定します。



このように、itron_app というプロジェクトを作成すると、以下のファイルが自動的に作成されます。

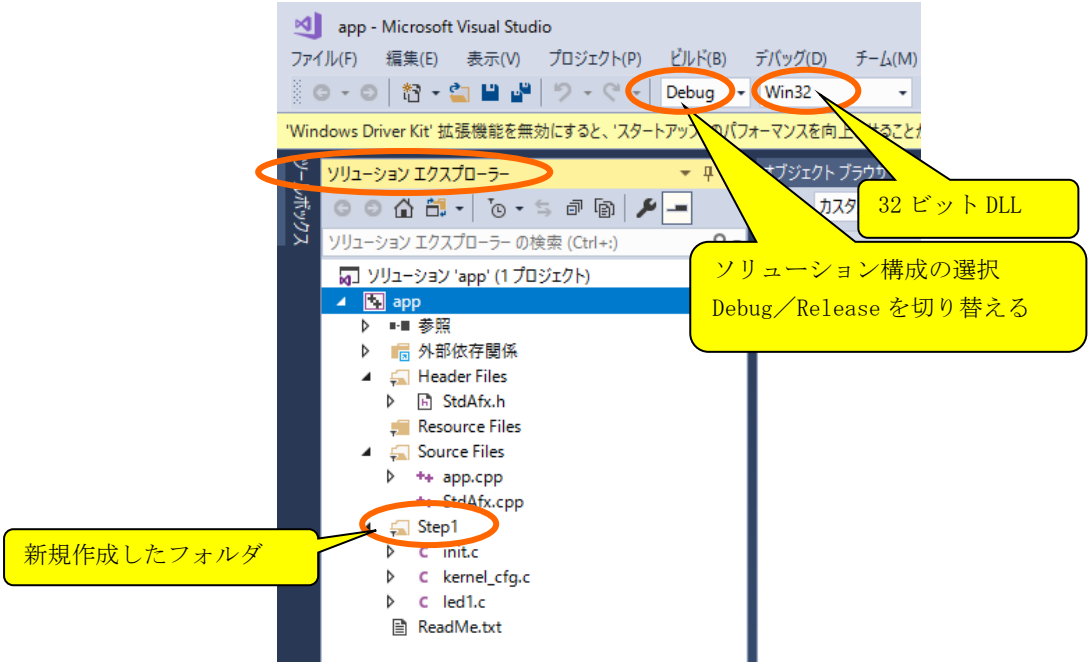
```
¥itron_app
    ¥itron_app
        stdafx.cpp
        dllmain.cpp
        itron_app.cpp
        stdafx.h
        targetver.h
        itron_app.vcproj      プロジェクトファイル
    itron_app.sln             ソリューションファイル
```



ここで、プロジェクトのプロパティから追加のインクルードパス、ライブラリパス、ライブラリファイル、プリプロセッサ定義を登録します。詳細は「3.2 ビルド方法」を参照してください。
その後、μITRON アプリケーションの C 言語のソースファイル(*.c)、ヘッダファイル(*.h)をプロ

ジェクトに追加します。Visual C++コンパイラは、拡張子*.c のファイルはC 言語のソースファイルとしてコンパイルします。

例えば、「7.チュートリアル」で示すステップ１のソリューションファイルを Visual Studio 2017 でで開くと以下のようになります。



3.2 ビルド方法

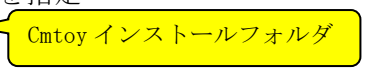
コンパイル、リンクを行い実行モジュールを作る作業をビルドと呼びます。アプリケーションプログラムの実行モジュールは、Windows の DLL 形式（*.dll）で作成します。この実行モジュールには、kernel_cfg.c も含めます。
ビルドを実行する前にコンパイラ、リンカの設定を確認してください。

3.2.1 Visual Studio 6.0 でのプロジェクトの設定

付属の¥mITRON¥sample¥app¥app.dsw を Visual Studio 6.0 で開いて「プロジェクト(P)」メニュの「設定(S)...」を確認してください。特に以下の部分を確認してください。

●C/C++ タブ
　カテゴリ：プリプロセッサ
　　「プリプロセッサの定義」に",_CMTOY,_APP_EXPORT"を追加（デバグ／リリースバージョン）
　　「インクルードファイルのパス」は"..¥..¥..¥include"（デバグ／リリースバージョン）
●リンク タブ
　カテゴリ：一般
　　「出力ファイル名（N）」は、"Release/app.dll"（リリースバージョン）
　　"Debug/Dapp.dll"（デバグバージョン）
　カテゴリ：インプット
　　「追加ライブラリパス」は、"..¥..¥..¥LIB"（デバグ／リリースバージョン）
●ビルド後の処理 タブ
　　実行モジュール app.dll を Cmtoy¥bin へコピーするためのコマンドを指定
　　"copy release¥app.dll ..¥..¥..¥bin"（リリースバージョン）
　　~~copy debug¥Dapp.dll ..¥..¥..¥bin（デバグバージョン）~~
●デバッグ タブ

「デバッグセッションの実行可能ファイル(E)」として以下を指定
“D:¥cmtoy-200¥bin¥Cmtoy.exe”（デバグバージョンのみ）



3.2.2 Visual C++ 2008 Express Edition でのプロジェクトのプロパティ

付属の¥mITRON¥sample¥app¥app.sln を Visual C++ 2008 Express Edition で開いて「プロジェクト(P)」メニュの「app のプロパティ(P)...」を確認してください。特に「構成プロパティ」の以下の部分を確認してください。

●C/C++ 内の
全般の「追加のインクルードディレクトリ」へ
”..¥..¥..¥include” （デバグ／リリースバージョン）
プリプロセッサ内の「プリプロセッサの定義」へ
”_CMTOY,_APP_EXPORT”を追加（デバグ／リリースバージョン）
必要に応じて“_CRT_SECURE_NO_WARNINGS” を追加

●リンカ内の
全般の「出力ファイル名」は、 ”Release/app.dll”（リリースバージョン）
“Debug/Dapp.dll”（デバグバージョン）
全般の「追加のライブラリディレクトリ」へ
”..¥..¥..¥LIB” （デバグ／リリースバージョン）
入力の「追加の依存ファイル」へ
”cm.lib kpdll.lib” （デバグ／リリースバージョン）

●ビルドイベント内の
ビルド後のイベントへ
実行モジュール app.dll を Cmtoy¥bin へコピーするためのコマンドを指定
コマンドラインは、
“copy release¥app.dll ..¥..¥..¥bin”（リリースバージョン）
~~copy debug¥Dapp.dll ..¥..¥..¥bin（デバグバージョン）~~

●デバッグ内の
「コマンド」へ
”D:¥cmtoy-200¥bin¥Cmtoy.exe”（デバグバージョンのみ）

3.2.3 Visual Studio 2017 でのプロジェクトのプロパティ

付属の¥mITRON¥sample¥app¥app.sln を Visual Studio 2017 で開いて「プロジェクト(P)」メニュの「app のプロパティ(P)...」を確認してください。特に「構成プロパティ」の以下の部分を確認してください。付属の app.sln は Visual C++ 2008 Express Edition で作成したものなので最初に 2017 の形式に変換するかどうか問い合わせがあります。

●C/C++ 内の
全般の「追加のインクルードディレクトリ」へ
”..¥..¥..¥include” （デバグ／リリースバージョン）
プリプロセッサの「プリプロセッサの定義」へ
”_CMTOY,_APP_EXPORT”を追加（デバグ／リリースバージョン）
必要に応じて“_CRT_SECURE_NO_WARNINGS” を追加

●リンカ内の
全般の「出力ファイル名」は、 ”Release/app.dll”（リリースバージョン）
“Debug/Dapp.dll”（デバグバージョン）
全般の「追加のライブラリディレクトリ」へ
”..¥..¥..¥LIB” （デバグ／リリースバージョン）
入力の「追加の依存ファイル」へ、
”cm.lib kpdll.lib” （デバグ／リリースバージョン）

●ビルドイベント内の

ビルド後のイベント
　　実行モジュール app.dll を Cmtoy¥bin へコピーするためのコマンドを指定
　　コマンドラインは、
　　“copy　release¥app.dll　..¥..¥..¥bin”（リリースバージョン）
●デバッグ内の
「コマンド」へ
　　”D:¥cmtoy-200¥bin¥Cmtoy.exe”（デバグバージョンのみ）

3.2.4 Borland　C++コンパイラ

Borland　C++コンパイラを使ってアプリケーションタスクを作るメイクファイルの例は、
Cmtoy¥mITRON¥sample¥app¥makefile.bcc
を参照してください。コンパイル時のオプションはマクロ CPP_SWITCHES を、リンク時のオプションはマクロ LINK32_FLAGS、LINKLIBS、LINKSTARTUP を参照してください。
中間ファイルをすべて削除するときは、以下のように指定します。
Cmtoy¥mITRON¥sample¥app> **make –f makefile.bcc clean**

app.dll をビルドするときは、以下のように指定します。
Cmtoy¥mITRON¥sample¥app> **make –f makefile.bcc**

これで、MFC を使用しないリリースバージョンのアプリケーション実行モジュールが作れます。

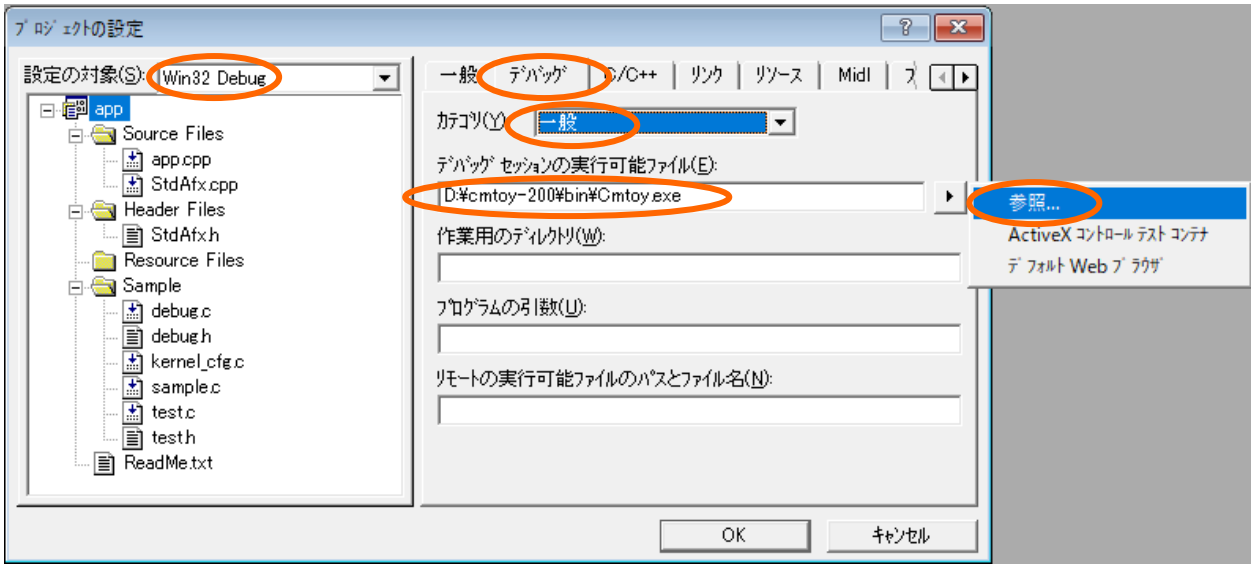
※Cmtoy V2.00 では Borland C++は未確認。

3.3 VisualStudio6.0 のデバッガの使用

まず、VisualStudio6.0 で付属のサンプルアプリケーションのワークスペース（*.dsw）を開き、「ビルド(B)」メニュの「アクティブな構成の設定(O)」で構成「Win32 Debug」を選択して**リビルド**します。**リビルド**すると実行モジュール内のソースファイルのパス情報がそのマシンのものになり、デバッガがソースファイルを認識できるようになるようです。

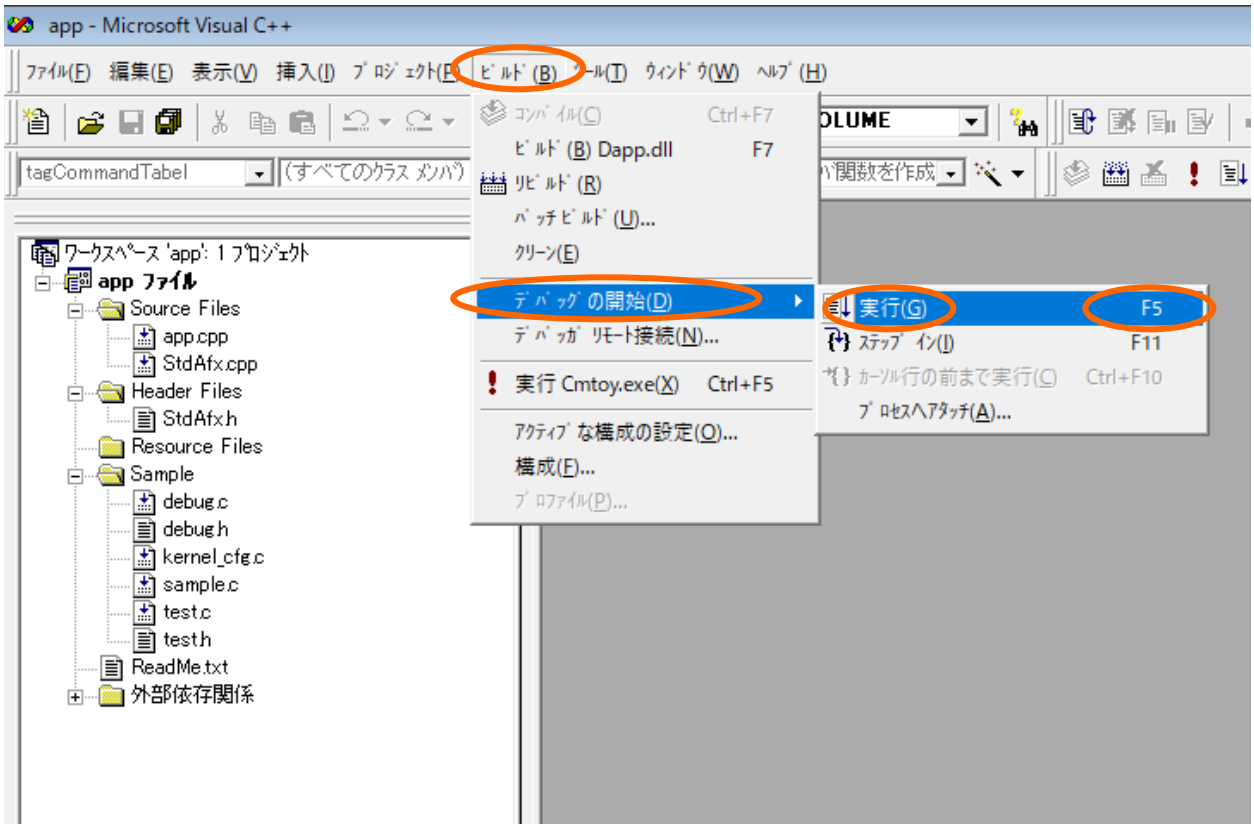
3.3.1μITRON アプリケーションのプロジェクトからデバッガを使う

まず、プロジェクトファイルを開き「プロジェクトの設定」の「デバッグ」 タブ、カテゴリ「一般」を確認します。ここで「デバッグセッションの実行可能ファイル(E)」として Cmtoy のインストールフォルダから Cmtoy.exe を指定します。（デバグバージョン Win32 Debug のみ）

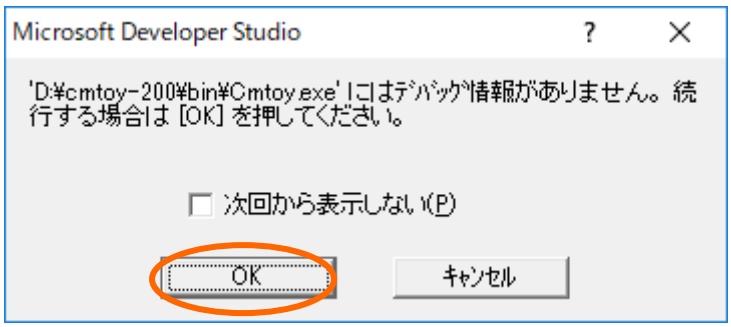


以上を確認したら「OK」をクリックします。

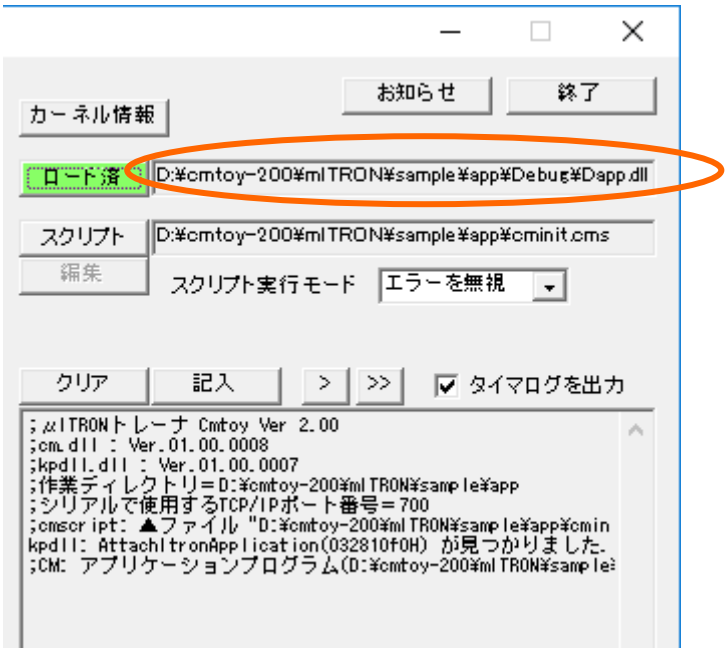
ここで、プロジェクトのアクティブな構成が「Win32 Debug」であることを確認して、「ビルド」メニュの「リビルド(R)」を実行してエラーのないことを確認します。これでデバグの準備ができました。
ここで、「ビルド」メニュの「デバッグの開始(D)」→「実行(G)」を選択します。または F5 キーを押します。



ここで以下のダイアログが表示されるので「OK」をクリックします。



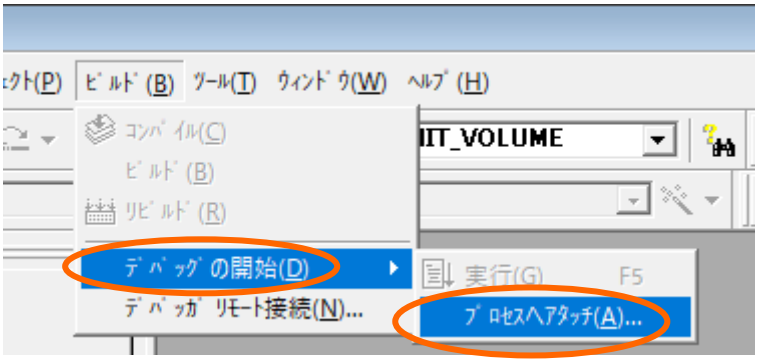
これでデバグセッションが開始し、Cmtoy の GUI ウインドウが表示されます。Cmtoy の「ロード」ボタンからリビルドした実行ファイル Debug¥Dapp.dll をロードします。



ここで、アプリケーションのソースファイルを開きブレークポイントを設定し、Cmtoy の「リセット」ボタンからμITRON カーネルとアプリケーションを実行します。

3.3.2 Cmtoy 起動後にデバッガを使う

最初に、bin ディレクトリにある Cmtoy.exe を起動し、リビルドした Dapp.dll をロードします。ここで VisualStudio6.0 を立ち上げ、「ビルド(B)」メニュの「デバッグの開始(D)」->「プロセスへアタッチ(A)...」をクリックすると以下のような実行中のプロセスの一覧を表示するダイアログが現れます。



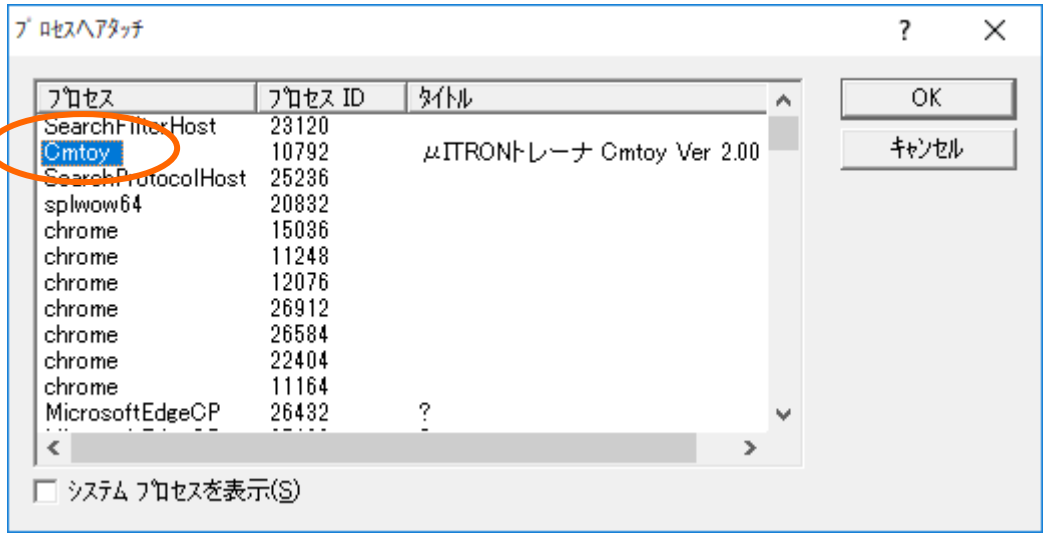
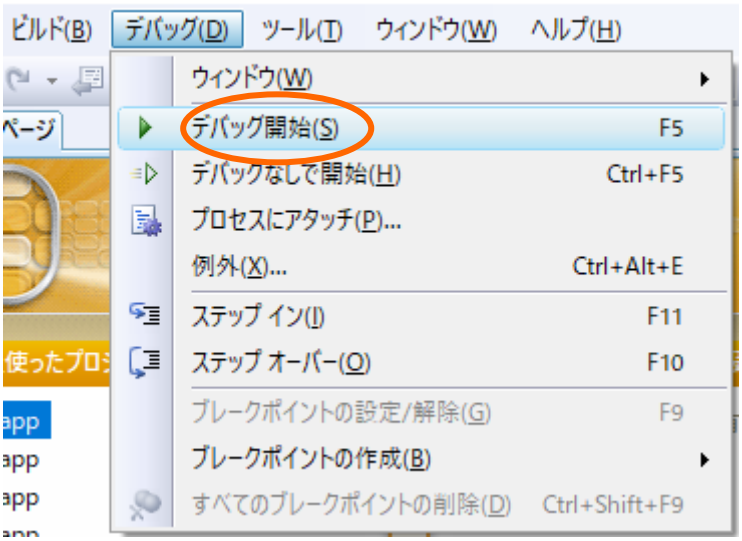


図 3-1 実行中のプロセス一覧

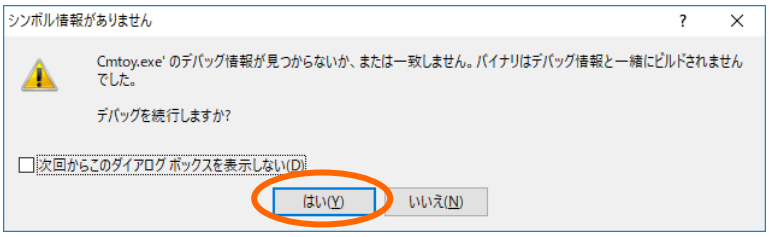
ここで Cmtoy を選んで「OK」ボタンをクリックすると実行中のプロセスをデバッガでデバッグできるようになります。例えば、デバッグしたいソースファイルを開きブレークポイントを設定した後、GUI の「リセット」ボタンをクリックしてμITRON カーネルとアプリケーションを実行します。

3.4 VisualC++ 2008 Express Edition のデバッガの使用

まず、Visual C++ 2008 Express Edition で付属のサンプルアプリケーションのソリューション（*.sln）を開き、ソリューション構成が「Debug」であることを確認して、「ビルド」メニュの「ソリューションのリビルド(R)」を実行してエラーのないことを確認します。これでデバグの準備ができました。
ここで、「デバッグ(D)」メニュの「デバッグの開始(S) F5」を選択します。または F5 キーを押します。



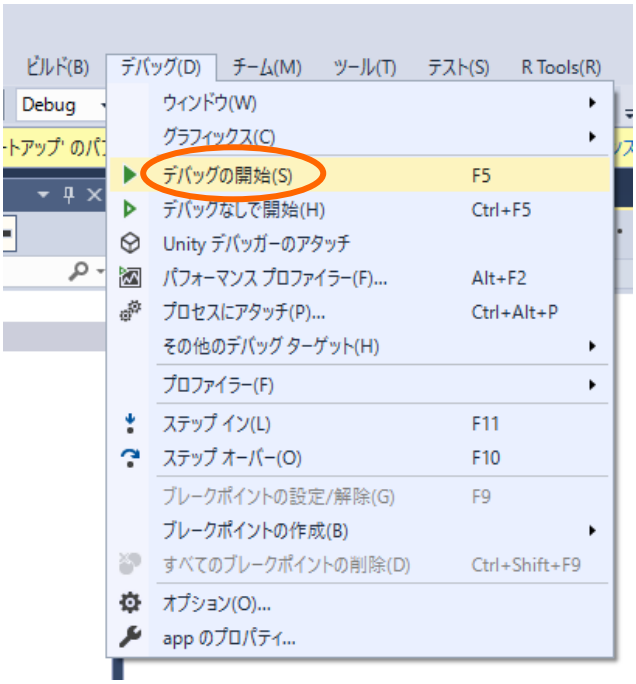
ここで以下のダイアログが表示されるので「OK」をクリックします。



これでデバグセッションが開始し、Cmtoy の GUI ウインドウが表示されます。Cmtoy の「ロード」ボタンからリビルドした実行ファイル Debug¥Dapp.dll をロードします。
ここで、アプリケーションのソースファイルを開きブレークポイントを設定し、Cmtoy の「リセット」ボタンからμITRON カーネルとアプリケーションを実行します。

3.5 Visual Studio 2017 のデバッガの使用

まず、Visual C++ 2008 Express Edition で付属のサンプルアプリケーションのソリューション（*.sln）を開き、ソリューション構成が「Debug」であることを確認して、「ビルド」メニュの「ソリューションのリビルド(R)」を実行してエラーのないことを確認します。これでデバグの準備ができました。
ここで、「デバッグ(D)」メニュの「デバッグの開始(S) F5」を選択します。または F5 キーを押します。



これでデバグセッションが開始し、Cmtoy の GUI ウインドウが表示されます。Cmtoy の「ロード」ボタンからリビルドした実行ファイル Debug¥Dapp.dll をロードします。
ここで、アプリケーションのソースファイルを開きブレークポイントを設定し、Cmtoy の「リセット」ボタンからμITRON カーネルとアプリケーションを実行します。

4　μITRON カーネルの機能

4.1 カーネルの概要

ここでは、CPU 依存、ハードウェア依存、実装依存となる機能を整理します。Cmtoy の想定しているハードウェアについては「1.3 ターゲットハードウェアの概要」を参照してください。

4.1.1外部割込み制御

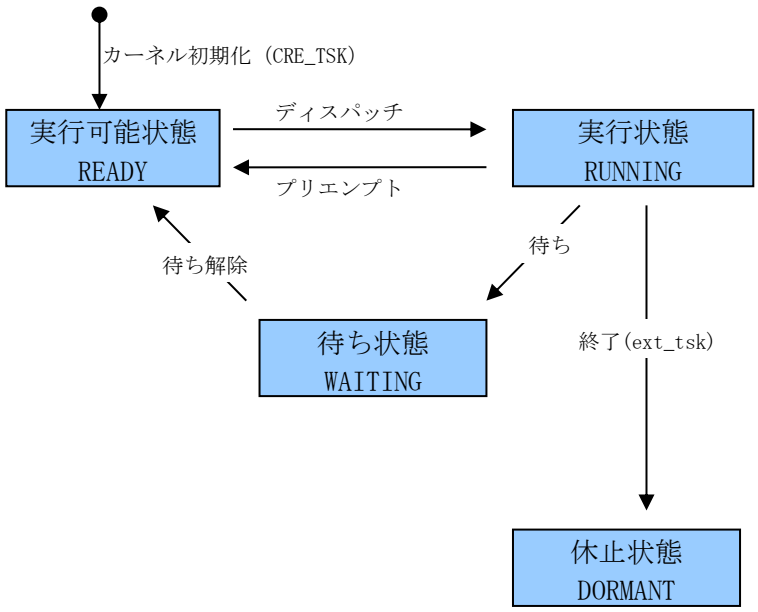
ユーザ割込みハンドラは C の関数として記述して、コンフィグレーションで割込みハンドラとして登録します。
CPU が割込みを受け付けると、カーネルの割込みハンドラが起動され、そこからユーザの割込みハンドラが呼び出されます。割込みハンドラの C 言語の関数が終了すると、カーネルの割込みハンドラに戻ります。ここで割込みコントローラへ EOI コマンドを送出し、最後にタスクスケジュールを実行します（遅延ディスパッチ）。

- 16 個の外部割込みレベルにはμITRON の割込み番号（0～15）を割り当て、割込み番号＝割込みハンドラ番号とする。
- ユーザ割込みハンドラは、割込み禁止状態で開始される。
- CPU の割込みフラグを制御する実装依存サービスコールを用意する。(vchg_ifl, vget_ifl)

※現状の Cmtoy では多重割込みを起こすようなシミュレーションはできません。

4.1.2タスク

タスクは C の関数として記述して、コンフィグレーションでタスクとして登録します。
静的 API で登録されたタスクはカーネル初期化時にレディキューに並びます。これは実行可能状態になるということです。
各タスクは、割込み許可状態で実行を開始します。
タスクの C 言語の関数が終了した場合は、カーネルが ext_tsk を実行します。
タスクの状態遷移を以下に示します。



タスクが待ち状態になるのは、自ら待ち状態になるサービスコールを呼び出したときです。
待ち解除は待ち状態になるサービスコールが待ち時間経過した場合、実行中のタスクまたは割込みハンドラから待ち解除のサービスコールで指定された場合です。

すべてのタスクが待ち状態になった場合には、カーネルはアイドルタスクを実行状態にします。アイドルタスクはカーネルが用意していますが、コンフィギュレーションファイルのタスク登録でアイドルタスクを登録する作業は必要です。アイドルタスクの登録方法は静的 API CRE_TSK を使い以下のように記述します（kernel_cfg.c を参照）。

```
CRE_TSK(IDLE_TASK_ID,(TA_HLNG | TA_ACT),NULL,NULL,TMAX_TPRI, 0,NULL,
"idle")    /*idle task*/
```

アイドルタスクのタスク ID は IDLE_TASK_ID(0)、タスク優先度は TMAX_PRI としてくださ。また、タスクスタートアドレスは NULL としてください。
カーネルの用意しているアイドルタスクは以下のようになります。アイドルタスクが実行状態になっても Windows の CPU タイムは消費しません。この状態でも割込み要求があれば割込みハンドラは動きタスクスケジュールは発生します。

```
void KpIdleTask(VP_INT exinf)
{
    PRINT_INFO("kpdll:KpIdleTask started.¥n");
    while(1){
        HALT;//CPU の Halt 命令に相当、割込み許可状態
    }
}
```

※Cmtoy ではタスク登録で、スタックアドレスは NULL、スタックサイズは 0 としてください。Cmtoy ではカーネルがスタック領域を割り当てます。現在は 1 MB を割り当てています。

4.1.3タイマ機能

カーネルの時間管理は、インターバルタイマで行います。カーネルはインターバルタイマの時間間隔を 10ms に初期化します。インターバルタイマは割込みレベル 0 を使います。カーネルはシステムの時間としてこのインターバルタイマの割込み回数で管理します。サービスコールのタイムアウト時間もインターバルタイマの割込み回数で管理します。

Cmtoy は GUI からこのインターバルタイマの動きを制御できます。以下のような操作ができます。

- インターバルタイマの停止／開始の制御
- インターバルタイマの割り込み間隔を 10ms の倍数で指定
- インターバルタイマの割込みをマニュアル操作で起こす

このように操作してもカーネルは割込み回数だけを数えて時間管理をします。

※Cmtoy ではインターバルタイマの時間間隔を Windows の WM_TIMER イベントでシミュレートしています。そのため 10ms のような短い時間ではそれほど正確ではありません。

4.1.4 Cmtoy 固有の機能

カーネルでは、タスク切替とサービスコールの発行、戻りをそれぞれのトレースバッファに記録します。
トレース情報を見るには Cmtoy の GUI ウインドウの「カーネル情報」ボタンをクリックします。するとトレース情報を表示するためのモードレスダイアログボックスが表示されるので、そこで「表

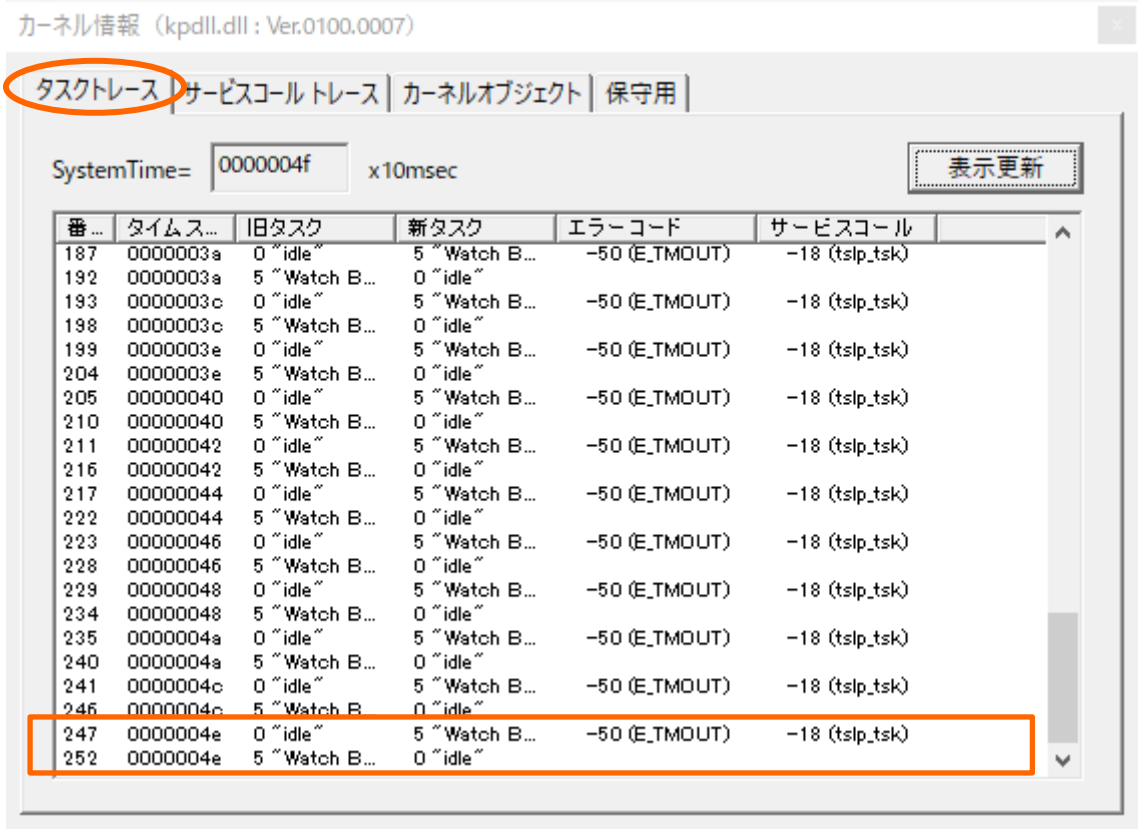
示更新」ボタンをクリックするとトレースバッファの最新の情報が表示されます。

(1) オブジェクト名

タスクなどのオブジェクトを静的に生成する API でオブジェクト名を指定できます。割込みハンドラの登録 API でも割込みハンドラ名を指定できます。
これらのオブジェクト名、割込みハンドラ名を指定しておくと、以下で説明するトレース機能やエラー発生時の表示に使われ、オブジェクトの識別がしやすくなります。

(2) タスク切替のトレース

タスクトレースでは、ディスパッチャ内で旧タスクと新タスクを記録します。新タスクが待ち状態からの解除であれば待ちに入ったサービスコール名と解除時のエラーコードを記録します。「表示更新」ボタンでスナップショットを更新します。



番…	タイムス…	旧タスク	新タスク	エラーコード	サービスコール
187	0000003a	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
192	0000003a	5 "Watch B…	0 "idle"		
193	0000003c	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
198	0000003c	5 "Watch B…	0 "idle"		
199	0000003e	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
204	0000003e	5 "Watch B…	0 "idle"		
205	00000040	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
210	00000040	5 "Watch B…	0 "idle"		
211	00000042	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
216	00000042	5 "Watch B…	0 "idle"		
217	00000044	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
222	00000044	5 "Watch B…	0 "idle"		
223	00000046	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
228	00000046	5 "Watch B…	0 "idle"		
229	00000048	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
234	00000048	5 "Watch B…	0 "idle"		
235	0000004a	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
240	0000004a	5 "Watch B…	0 "idle"		
241	0000004c	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
246	0000004c	5 "Watch B…	0 "idle"		
247	0000004e	0 "idle"	5 "Watch B…	-50 (E_TMOUT)	-18 (tslp_tsk)
252	0000004e	5 "Watch B…	0 "idle"		

図 4-1 タスク切替のトレース情報

上記の 247 の行の意味は、

タイムスタンプ 0000004e にタスク 0 からタスク 5 へ切り替わり、
タスク 5 は、サービスコール tslp_tsk が終了し、その戻り値は-50 だった。
タスク 5 は、待ち状態から実行状態へ遷移

となります。アイドルタスクは常に実行状態か実行可能状態のどちらかなので

タスク 0 は、実行状態から実行可能状態へ遷移（プリエンプトされた）

となります。

※タイムスタンプはインターバルタイマの割込み回数です。

(3) サービスコールのトレース

サービスコールのトレースでは、発行時のパラメータと戻り時のエラーコードを記録します。待ちに入る可能性のあるサービスコールの戻り時とは待ちが解除された時点のことです。
発行時には、その時点のコンテキスト情報とサービスコールのパラメータを４つまで記録します。
戻り時には、サービスコールの返すエラーコードと１つのリターンパラメータを記録します。「表示更新」ボタンでスナップショットを更新します。

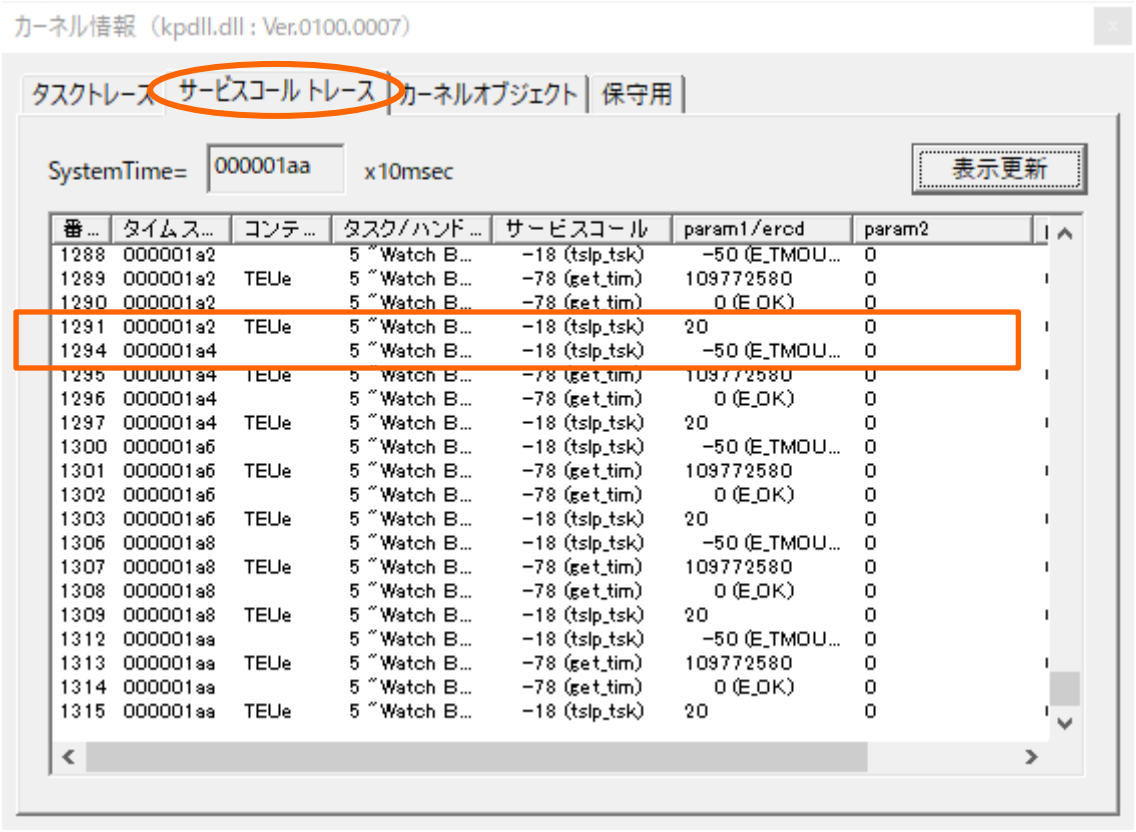


図 4-2 サービスコールのトレース情報

「コンテキスト」項目には、4 文字でサービスコール発行時のコンテキスト情報を表示します。

第 1 文字＝{T|H}	T:タスクコンテキスト、H：割込みハンドラ
第 2 文字＝{E|D}	E:ディスパッチ許可状態、D：ディスパッチ禁止状態
第 3 文字＝{U|L}	U:アンロック状態、L：ロック状態
第 4 文字＝{e|d}	e:CPU 割込み許可状態、d:CPU 割込み禁止状態

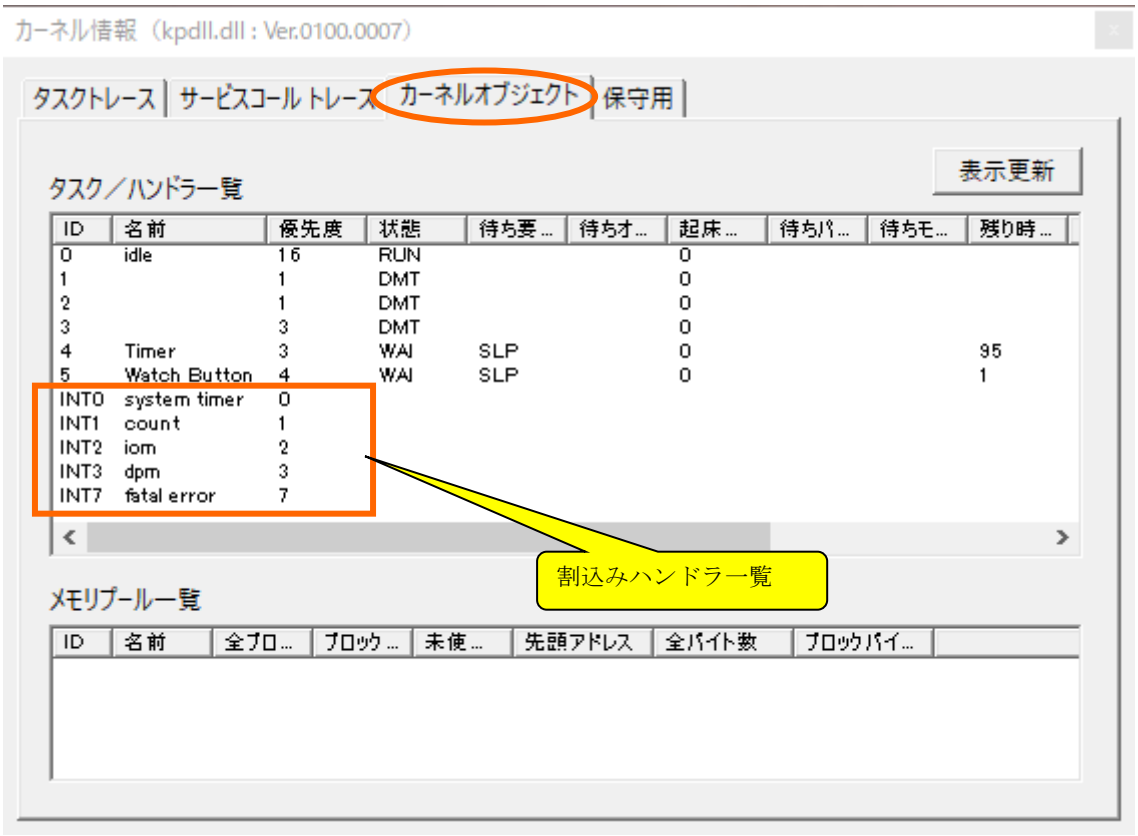
サービスコールからの戻り時の「コンテキスト」項目は空白です。

上記の 1291 の行の意味は、
　　タイムスタンプ 000001a2 にタスク 5 がサービスコール tslp_tsk(20)を実行した。
　　タスク 5 は待ち状態に遷移
となります。
上図の 1294 の行の意味は、
　　タスク 5 のサービスコール tslp_tsk が戻り値-50 で終了した。
　　タスク 5 は、実行状態に遷移
となります。

※タイムスタンプはインターバルタイマの割込み回数です。

(4) μITRON オブジェクトの一覧

タスク/ハンドラの一覧とメモリプールの情報を表示します。「表示更新」ボタンでスナップショットを更新します。



ここで、タスク状態は、

タスク ID=0　　実行状態
タスク ID=1,2,3　休止状態
タスク ID=4　　待ち状態　　残り時間=95（950ms）
タスク ID=5　　待ち状態　　残り時間=1（10ms）

となり、割込みレベル０，１，２，３，７に割込みハンドラが登録されています。
タスク ID=0 はアイドルタスク、割込みレベル０の割込みはインターバルタイマでカーネルが登録したオブジェクトです。

(5) サービスコールのエラー

サービスコールでエラーが発生した場合は出力ウインドウにエラー情報を以下の形式で表示します。

〈タイムスタンプ〉: 〈ercd〉 by 〈fncd〉 in task[〈タスク ID〉].〈タスク名〉
〈タイムスタンプ〉: 〈ercd〉 by 〈fncd〉 in intr[〈割込み番号〉].〈割込みハンドラ名〉

ただし、以下のエラーコードの場合は表示しません。

E_TMOUT
E_RLWAI

例えばタスク ID=2 の“debug”というタスクでサービスコール ref_mpf を呼び出したときにエラーが発生した場合は、以下のような表示となります。

00000010: E_NOEXS by ref_mpf in task[2].debug

4.2 実装済みサービスコール一覧

現在以下のサービスコールを実装しています。itron.h で定義しています。

```
/* (1) タスク管理機能 */
CRE_TSK          タスク生成定義（静的 API）
ext_tsk          自タスクの終了
ref_tsk          タスクの状態参照

/* (2) タスク付属同期機能*/
get_pri          タスク優先度の参照
slp_tsk          起床待ち（タイムアウトなし）
tslp_tsk         起床待ち（タイムアウトあり）
wup_tsk          タスクの起床
iwup_tsk         タスクの起床（非タスクコンテキスト専用）
can_wup          タスクの起床要求のキャンセル


/* (3) タスク例外処理機能 */

/* (4) 同期・通信機能 */
/*セマフォ*/
CRE_SEM          セマフォ生成定義（静的 API）
sig_sem          セマフォ資源の返却
isig_sem         セマフォ資源の返却（非タスクコンテキスト専用）
wai_sem          セマフォ資源の獲得（タイムアウトなし）
pol_sem          セマフォ資源の獲得（ポーリング）
twai_sem         セマフォ資源の獲得（タイムアウトあり）
ref_sem          セマフォの状態参照

/* イベントフラグ */
CRE_FLG          イベントフラグ生成定義（静的 API）
set_flg          イベントフラグのセット
iset_flg         イベントフラグのセット（非タスクコンテキスト専用）
clr_flg          イベントフラグのクリア
wai_flg          イベントフラグ待ち（タイムアウトなし）
pol_flg          イベントフラグ待ち（ポーリング）
twai_flg         イベントフラグ待ち（タイムアウトあり）
ref_flg          イベントフラグの状態参照

/* データキュー */

/* メールボックス */
CRE_MBX          メイルボックス生成定義（静的 API）
snd_mbx          メイルボックスへの送信
rcv_mbx          メイルボックスからの受信（タイムアウトなし）
prcv_mbx         メイルボックスからの受信（ポーリング）
```

```
trcv_mbx        メイルボックスからの受信（タイムアウトあり）
ref_mbx         メイルボックスの状態参照

/* (5) メモリプール管理機能 */

/* 固定長メモリプール */
CRE_MPF         固定長メモリプールの生成定義（静的 API）
get_mpf         固定長メモリブロックの獲得（タイムアウトなし）
pget_mpf        固定長メモリブロックの獲得（ポーリング）
tget_mpf        固定長メモリブロックの獲得（タイムアウトあり）
rel_mpf         固定長メモリプールの状態参照

/* (6) 時間管理機能 */
/* システム時刻管理 */
set_tim         システム時刻の設定
get_tim         システム時刻の参照

/* 周期ハンドラ */

/* (7) システム状態管理機能 */
get_tid         実行状態のタスク ID の参照
iget_tid        実行状態のタスク ID の参照（非タスクコンテキスト専用）
dis_dsp         ディスパッチの禁止
ena_dsp         ディスパッチの許可
sns_ctx         コンテキストの参照
sns_dsp         ディスパッチ禁止状態の参照
sns_dpn         ディスパッチ保留状態の参照

/* (8) 割込み管理機能 */
DEF_INH         割り込みハンドラ生成定義（静的 API）
dis_int         割込みの禁止
ena_int         割込みの許可
vchg_ifl        CPU の割込みフラグの変更（実装依存機能）
vget_ifl        CPU の割込みフラグの取得（実装依存機能）

/* (9) システム構成管理機能 */
ATT_INI         初期化ルーチンの追加（静的 API）
```

4.3 Cmtoy でのリセット動作

Cmtoy が起動するとμITRON カーネル(kpdll.dll)もメモリ上にロードします。この段階ではμITRON カーネルはメモリ上に配置されただけで実行はしていません。その後 GUI のロードボタンを使ってμITRON アプリケーションをメモリ上に配置します。

μITRON カーネルとμITRON アプリケーションがメモリ上に配置された状態で、GUI のリセットボタンをクリックするとμITRON カーネルの開始ルーチンが実行され、以下の手順でタスクを起動します。

① カーネルの初期化（インターバルタイマ、IRC の初期化も含む）
② 静的 API の処理
　オブジェクトの生成（タスクはレディキューへ並ぶ）
　初期化ルーチンの実行
③ システム時刻の初期化、インターバルタイマの起動
④ 割込み許可、ディスパッチ許可
⑤ レディキューの先頭のタスクを実行

その後、実行状態のタスクが待ち状態、休止状態になり、レディキューから外れると、カーネルはレディキューの先頭のタスクを実行状態にします。

5 C-Machine の機能

アプリケーションプログラムの作成において以下のハードウェア（CPU、デバイス）を制御する関数、マクロが使えます。これらは hal.h と hal_uart.h で定義しています。これらの関数、マクロの使用例は「8 C-Machine のプログラム例」を参照してください。

5.1 CPU、割込み制御関数

5.1.1 void halDisableInterrupt(void);

パラメータ
　　なし
戻り値
　　なし
解説
　　CPU の割込みを禁止する。

5.1.2 void halEnableInterrupt(void);

パラメータ
　　なし
戻り値
　　なし
解説
　　CPU の割込みを許可する。

5.1.3 BOOL halInquireInterruptStatus(void);

パラメータ
　　なし
戻り値
　　CPU の現在の割込みフラグ
解説
　　CPU の現在の割込みフラグを取得する。割込み禁止なら TRUE、割込み許可なら FALSE を返す。

5.1.4 void halMaskInterrupt(int level, BOOL mask);

パラメータ

level	割込みコントローラの割込みレベル
mask	割込みマスク指定（TRUE でマスク、FALSE でマスク解除）

戻り値
　　なし
解説
　　割込みコントローラの指定した割込みレベルのマスクを設定する。

5.1.5 void halEndOfInterrupt(int level);

パラメータ

level　　割込みコントローラの割込みレベル

戻り値

なし

解説

割込み処理終了を割込みコントローラに通知する。

※μITRON カーネルが割込み制御で使うので、ユーザアプリケーションから使う必要はない。

5.2 デバッグ出力制御関数

GUI の出力ウインドウへ文字列を表示するための機能を提供する関数です。

5.2.1 void halDebugOutputString(const char *cstr);

パラメータ

cstr　　文字列

戻り値

なし

解説

デバッグ用の文字列を出力する。

5.2.2 void halDebugPrintf(const char *formatstring, ...);

パラメータ

formatstring　　書式制御文字列

戻り値

なし

解説

printf 関数と同様の書式でデバッグ用文字列を作成し出力する。

5.3 LED 表示制御関数

GUI 上の８個の LED ランプと表示器（2 個の７セグメント LED）を操作する関数を提供します。

5.3.1 void halSetLED(WORD led);

パラメータ

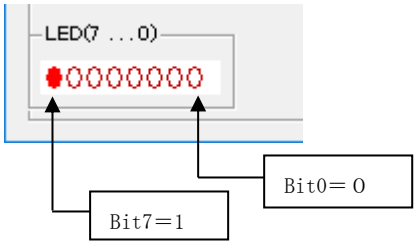
led　　各ビットで LED の ON(1)/OFF(0)を指定する。

戻り値

なし

解説

LED の ON/OFF を設定する。



5.3.2 void halSetSegLED(WORD stat);

パラメータ

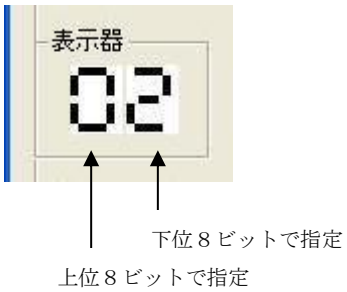
stat　　各ビットで LED セグメントの ON(1)/OFF(0)を指定する。

戻り値

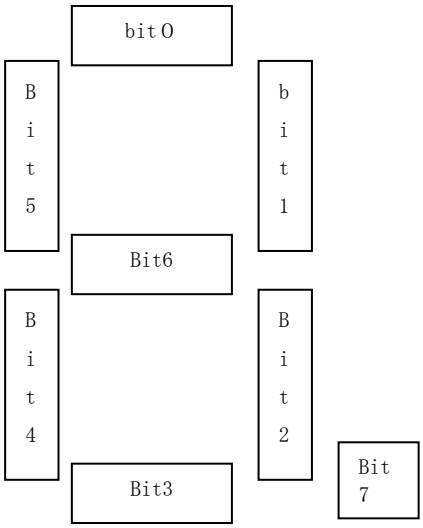
なし

解説

表示器（2 個の 7 セグメント LED）の LED セグメントの ON/OFF を設定する。



ビットと LED セグメントの対応は以下のとおり。



5.4 ボリューム制御関数

5.4.1 WORD halGetVolume(int VolumeNo);

パラメータ

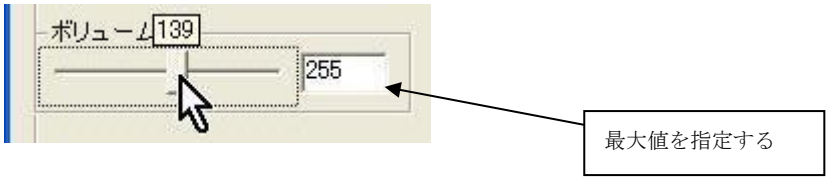
VolumeNo　　ボリューム番号（0 を指定）

戻り値

現在のボリューム値を返す。

解説

ボリューム値は 0 ～最大値の間の整数値。



5.5 スイッチ制御関数

5.5.1 WORD halGetSwitch(void);

パラメータ

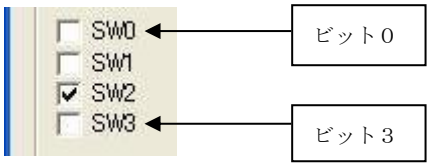
なし

戻り値

下位 4 ビットで各スイッチの ON(1)/OFF(0)を返す

解説

各ビットでスイッチの状態を返す。



5.6 ボタン制御関数

5.6.1 BOOL halGetPushButton(int ButtonNo);

パラメータ

ButtonNo　　ボタン番号（０を指定）

戻り値

ボタンの状態 UP(0)/DOWN(1)を返す。

解説

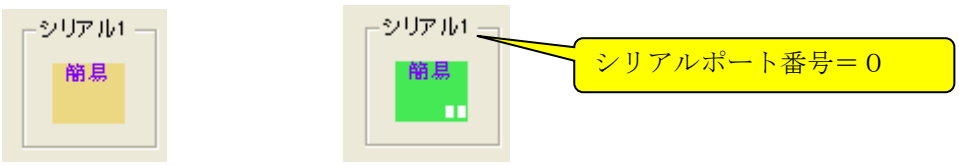
UP　　DOWN（マウスの左ボタンを押した状態）

5.7 簡易シリアル制御関数

C-Machine はシリアルポートを Windows の Winsock 機能を使ってでシミュレートします。TCP/IP のポート 700 と 701 を使います。ポート番号を変更するには「2.2.1 使用する TCP/IP ポート番号を変更する」を参照してください。

TCP/IP クライアント（端末アプリケーション）がこのポートに接続すると以下のようにコントロールの色が変わります。

クライアント未接続　　クライアント接続中



クライアントからの受信データは、内部のバッファに蓄えます。

Cmtoy を立ち上げた直後は、割込みを使わないで１文字単位の送信、受信をすることができる簡易シリアル機能となります。簡易シリアルは割込みを使わずポーリング形式での制御のみ可能です。簡易シリアル機能では以下の４つの制御関数が使えます。各制御関数ではシリアルポート番号を指定します。

① halSerialInit
② halSerialGetStatus
③ halSerialReadChar
④ halSerialWriteChar

シリアル１はシリアルポート番号０で TCP/IP のポート 700 を使い、シリアル２はシリアルポート番号１を使い TCP/IP のポート 701 を使います。

※TCP/IP クライアント（端末アプリケーション）とはハイパーターミナル、PuTTY などです。

5.7.1 void halSerialInit(int SerialNo);

パラメータ
　　SerialNo　　シリアルポート番号（０または１を指定）
戻り値
　　なし
解説
　　シリアルポートを初期化して、ここで送信許可、受信許可、RTS ON、DTR ON とする。

5.7.2 int halSerialReadChar(int SerialNo);

パラメータ
　　SerialNo　　シリアルポート番号（０または１を指定）
戻り値
　　内部のバッファから読み取った文字コード（バッファが空の場合は－1）
解説
　　受信データを取得する。

5.7.3 void halSerialWriteChar(int SerialNo, int c);

パラメータ
　　SerialNo　　シリアルポート番号（０または１を指定）
　　c　　出力する文字コード
戻り値
　　なし
解説

文字コードを速やかに TCP/IP ポートから送信する。

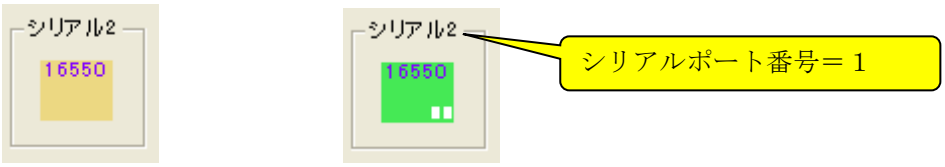
5.8 16550 相当のシリアル制御関数

C-Machine はシリアルポートを Windows の Winsock 機能を使ってでシミュレートします。TCP/IP のポート 700 と 701 を使います。ポート番号を変更するには「2.2.1 使用する TCP/IP ポート番号を変更する」を参照してください。
Cmtoy 起動後、簡易シリアル機能から 16550 相当のシリアル機能に変更することができます。例えば、シリアル 2 に 16550 相当の機能をシミュレートさせる場合にはコマンドコンソールまたはスクリプトで以下のをコマンドラインを実行します。

```
serial 1 init 4 16550   ;シリアル２に割込みレベル４を割り当てる
```

このコマンドを実行すると TCP/IP クライアントは未接続となり、GUI コントロールの表示は以下のように「簡易」から「１６５５０」変わります。

クライアント未接続　　**クライアント接続中**



以下で説明する 16550 のレジスタへの読み書きを行う制御関数を使いプログラミングします。ただし、ボーレートは 1200bps 相当（1 文字送受信は 10ms おき）に固定のためディバイザラッチレジスタ（DLL,DLM）への書き込みはできますが、ボーレートは変更できません。また、パリティやストップビットを設定しても動作に影響はありません。常に 8 ビット文字として送受信します。

この GUI コントロール上で右クリックすると、以下のような 16550 のレジスタの内容を表示するモードレスダイアログボックスが現れます。

FCR=FIFO Control Register
LCR=Line Control Register
MCR=Modem Control Register
IER=Interrupt Enable Register
LSR=Line Status Register
DLM:DLL=Divisor Latch
SCR=Scratch Register

図 5-1 シリアル２の 16550 のレジスタ

この状態では FIFO を使用しない 16450 モードとなっています。
アプリケーションプログラムで 16550 に初期化するプログラム例を以下に示します。

```
#include “hal.h”
#include “hal_uart.h”
int ch = 1;
WRITE_LCR(ch,0x80);         /*;DLAB=1*/
WRITE16550(ch,BAUDRATES_115200);  /*;DLAB=1: Diviser Latch(LS)*/
WRITE_IER(ch,BAUDRATES_115200>>8);/*;DLAB=1: Diviser Latch(MS)*/
WRITE_LCR(ch,0x03);         /*;DLAB=0: DATA=8, PARITY=NONE, STOPBIT=1*/
WRITE_MCR(ch,0x03);         /*;DTR=ON, RTS=ON*/
WRITE_IER(ch,0x0D);         /*;ENABLE RX READY,LINE STATUS,MODEM STATUS*/
WRITE_FCR(ch,0xc7);         /*;FIFO enable,FIFO clear, 14-bytes trigger*/
LSTAT16550(ch);             /*read Line Status*/
```

この初期化の後シリアル 2 のプロパティウインドウの「表示更新」をクリックすると最新の 16550 レジスタの状態が表示されます。

図 5-2 初期化後の 16550 のレジスタ

この中の「入力信号」を表すチェックボックスをマウスで操作して 16550 への入力を変更できます。それ以外の値は読み取り専用で変更できません。
ループバックに指定した場合は 16550 の仕様により DTR, RTS, OUT1, OUT2 が入力信号に接続されるので、マウス操作で入力信号を変えることはできません。

以降で 16550 相当のシリアル制御に使う関数の説明をします。

※ 16550 の機能については以下の各メーカのホームページからデータシートをダウンロードして参考にしました。
PC16550D Universal Asynchronous Receiver/Transmitter with FIFOs
http://www.ti.com/lit/ds/symlink/tl16c550c.pdf

5.8.1 void hal16550WriteDATA(int SerialNo, BYTE d);

パラメータ

SerialNo	シリアルポート番号（０または１を指定）
d	レジスタに書くデータ

戻り値

なし

解説

16550 のトランスミッタ保持レジスタ(THR)またはディバイザラッチレジスタ(DLL)に値 d を書く。

※Cmtoy では DLL への書き込みは動作に影響しない。

5.8.2 BYTE hal16550ReadDATA(int SerialNo);

パラメータ

SerialNo	シリアルポート番号（０または１を指定）

戻り値

レジスタから読み取った値

解説

16550 の受信バッファレジスタ(RBR)またはディバイザラッチレジスタ(DLL)の値を読み取る。

5.8.3 void hal16550WriteIER(int SerialNo, BYTE d);

パラメータ

SerialNo	シリアルポート番号（０または１を指定）
d	レジスタに書くデータ

戻り値

なし

解説

16550 の割込みイネーブルレジスタ(IER)またはディバイザラッチレジスタ(DLM)に値 d を書く。IER の定義済みビットは以下のとおり。

bit0 : Enable Received Data Available Interrupt
bit1 : Enable Transmitter Holding Register Empty Interrupt
bit2 : Enable Receiver Line Status Interrupt
bit3 : Enable MODEM Status Interrupt
bit4 : 0
bit5 : 0
bit6 : 0
bit7 : 0

※Cmtoy では DLM への書き込みは動作に影響しない。

5.8.4 BYTE hal16550ReadIER(int SerialNo);

パラメータ

SerialNo	シリアルポート番号（０または１を指定）

戻り値

レジスタから読み取った値

解説

16550 の割込みイネーブルレジスタ(IER)またはディバイザラッチレジスタ(DLM)の値を読

み取る。

5.8.5 BYTE hal16550ReadIID(int SerialNo);

パラメータ

SerialNo　　シリアルポート番号（０または１を指定）

戻り値

レジスタから読み取った値

解説

16550 の割込み識別レジスタ(IIR)の値を読み取る。IIR の定義済みビットは以下のとおり。

bit0 : 0 If Interrupt Pending
bit1 : Interrupt ID Bit0
bit2 : Interrupt ID Bit1
bit3 : Interrupt ID Bit2
bit4 : 0
bit5 : 0
bit6 : FIFOs Enabled(16450 モード時は０)
bit7 : FIFOs Enabled(16450 モード時は０)

Bit0-3 の組み合わせにより以下のような割り込み要因となります。

Bit3	Bit2	Bit1	Bit0	割り込み要因
0	0	0	1	割り込み要因なし
0	1	1	0	ライン状態変化（エラー検出、ブレーク信号検出）
0	1	0	0	受信データあり、FIFO モード時は受信 FIFO の受信トリガーレベルを超えた。
1	1	0	0	FIFO モード時に、受信データタイムアウト
0	0	1	0	送信バッファが空になった
0	0	0	0	モデム信号の変化を検出

5.8.6 void hal16550WriteFCR(int SerialNo, BYTE d);

パラメータ

SerialNo　　シリアルポート番号（０または１を指定）
d　　レジスタに書くデータ

戻り値

なし

解説

16550 の FIFO 制御レジスタ(FCR)に値 d を書く。FCR の定義済みビットは以下のとおり。

bit0 : FIFO Enable
bit1 : RCVR FIFO Reset
bit2 : XMIT FIFO Reset
bit3 : 0
bit4 : 0
bit5 : 0
bit6 : RCVR Trigger(LSB)
bit7 : RCVR Trigger(MSB)

5.8.7 BYTE hal16550ReadLSR(int SerialNo);

パラメータ

SerialNo　　シリアルポート番号（０または１を指定）

戻り値

レジスタから読み取った値

解説

16550 のラインステータスレジスタ(LSR)の値を読み取る。LSR の定義済みビットは以下のとおり。

bit0 : Data Ready
bit1 : Overrun Error
bit2 : Parity Error
bit3 : Framing Error
bit4 : Break Interrupt
bit5 : Transmitter Holding Register Empty
bit6 : Transmitter Empty
bit7 : Error in RCVR FIFO(16450 モード時は０）

5.8.8 BYTE hal16550ReadMSR(int SerialNo);

パラメータ

SerialNo　　シリアルポート番号（０または１を指定）

戻り値

レジスタから読み取った値

解説

16550 のモデムステータスレジスタ(MSR)の値を読み取る。MSR の定義済みビットは以下のとおり。

bit0 : Delta Clear to Send
bit1 : Delta Data Set Ready
bit2 : Delta Trailing Edge Ring Indicator
bit3 : Delta Data Carrier Detect
bit4 : Clear to Send
bit5 : Data Set Ready
bit6 : Trailing Edge Ring Indicator
bit7 : Data Carrier Detect

5.8.9 void hal16550WriteLCR(int SerialNo, BYTE d);

パラメータ

SerialNo　　シリアルポート番号（０または１を指定）
d　　レジスタに書くデータ

戻り値

なし

解説

16550 のライン制御レジスタ(LCR)に値 d を書く。LCR の定義済みビットは以下のとおり。

bit0 : Word Length Select Bit0 (動作に影響しない。常に 8 ビット)
bit1 : Word Length Select Bit1 (動作に影響しない。常に 8 ビット)
bit2 : Number of Stop Bits (動作に影響しない)
bit3 : Parity Enable (動作に影響しない)
bit4 : Even Parity Select (動作に影響しない)

bit5 : Stick Parity (動作に影響しない)
bit6 : Set Break
bit7 : Diviser Latch Access Bit

※Cmtoy ではビット 6,7 が意味を持つ。

5.8.10 BYTE hal16550ReadLCR(int SerialNo);

パラメータ
SerialNo　　シリアルポート番号（０または１を指定）
戻り値
レジスタから読み取った値
解説
16550 のライン制御レジスタ(LCR)の値を読み取る。

5.8.11 void hal16550WriteMCR(int SerialNo, BYTE d);

パラメータ
SerialNo　　シリアルポート番号（０または１を指定）
d　　レジスタに書くデータ
戻り値
なし
解説
16550 のモデム制御レジスタ(MCR)に値 d を書く。MCR の定義済みビットは以下のとおり。
bit0 : Data Terminal Ready
bit1 : Request to Send
bit2 : Out1
bit3 : Out2
bit4 : Loop
bit5 : 0
bit6 : 0
bit7 : 0

5.8.12 BYTE hal16550ReadMCR(int SerialNo);

パラメータ
SerialNo　　シリアルポート番号（０または１を指定）
戻り値
レジスタから読み取った値
解説
16550 のモデム制御レジスタ(MCR)の値を読み取る。

5.9 PN 符号，疑似ランダム雑音（PseudorandomNoise）の生成

PN9 と PN15 を計算する機能を提供します。

5.9.1 WORD halCalcPN9(WORD pn_code);

パラメータ
pn_code　　初期値

戻り値

計算結果

解説

与えられた pn_code から PN9 の計算をします。

5.9.2 WORD halGenPN9(WORD pn_code, BYTE *buf, int bytes);

パラメータ

pn_code	初期値
buf	結果を格納するバイト配列
bytes	バイト配列の要素数

戻り値

最終計算結果

解説

与えられた pn_code から順次 PN9 を計算し配列 buf へ格納します。
最終の PN コードを戻り値に返します。
buf は C 言語で定義した配列です。ターゲットメモリの物理アドレスではありません。

5.9.3 WORD halCalcPN15(WORD pn_code);

パラメータ

pn_code	初期値

戻り値

計算結果

解説

与えられた pn_code から PN15 の計算をします。

5.9.4 WORD halGenPN15(WORD pn_code, BYTE *buf, int bytes);

パラメータ

pn_code	初期値
buf	結果を格納するバイト配列
bytes	バイト配列の要素数

戻り値

最終計算結果

解説

与えられた pn_code から順次 PN15 を計算し配列 buf へ格納します。
最終の PN コードを戻り値に返します。
buf は C 言語で定義した配列です。ターゲットメモリの物理アドレスではありません。

5.10マクロ

5.10.1 CMTRACE (const char *formatstring, ...)

パラメータ

formatstring	書式制御文字列

戻り値

なし

解説

文字列を Cmtoy の出力ウインドウに出力する。

5.10.2 ターゲットメモリを操作するマクロ（アドレスを即値で使用する場合）

物理アドレスを以下のようにマクロで定義してターゲットメモリへアクセスする場合に使用します。

```
#define COMMAND  0xff000       /*コマンドレジスタ*/
#define STATUS   0xff001       /*ステータスレジスタ*/

void Clear()
{
    char temp = READ_BYTE(STATUS);
}

void Init()
{
    WRITE_BYTE(COMMAND, 0x80);
    WRITE_BYTE(COMMAND, 0x01);
    WRITE_BYTE(COMMAND, 0x40);
    WRITE_BYTE(COMMAND, 0x55);
}
```

(1) WRITE_BYTE(TADDR,DATA)

(2) WRITE_WORD(TADDR,DATA)

(3) WRITE_DWORD(TADDR,DATA)

パラメータ

TADDR	ターゲット CPU のメモリアドレスを表す整数値
DATA	バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲット CPU の指定されたメモリに値を書く。

未定義のメモリアドレスを指定すると、出力ウインドウに以下のメッセージを表示する。

CM: ▲WriteByte:不正な物理アドレス 1000

読み取り専用のメモリアドレスを指定すると、出力ウインドウに以下のメッセージを表示する。

CM: ▲WriteByte:書き込み不可, アドレス 100

(4) READ_BYTE(TADDR)

(5) READ_WORD(TADDR)

(6) READ_DWORD(TADDR)

パラメータ

TADDR	ターゲット CPU のメモリアドレスを表す整数値

戻り値

バイト、ワードまたはダブルワードの値

解説

ターゲット CPU のメモリの値を読む。
未定義のメモリアドレスを指定すると、出力ウインドウに以下のメッセージを表示する。

```
CM: ▲ReadByte:不正な物理アドレス 400
```

(7) OUT_BYTE(TPORT,DATA)

(8) OUT_WORD(TPORT,DATA)

(9) OUT_DWORD(TPORT,DATA)

パラメータ

TPORT	ターゲット CPU の IO ポートアドレスを表す整数値
DATA	バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲット CPU の IO ポートに値を書く。
未定義の IO アドレスを指定すると、出力ウインドウに以下のメッセージを表示する。

```
CM: ▲OutByte:不正な物理アドレス 400
```

(10) IN_BYTE(TPORT)

(11) IN_WORD(TPORT)

(12) IN_DWORD(TPORT)

パラメータ

TPORT	ターゲット CPU の IO ポートアドレスを表す整数値

戻り値

バイト、ワードまたはダブルワードの値

解説

ターゲット CPU の IO ポートの内容を読む。
未定義の IO アドレスを指定すると、出力ウインドウに以下のメッセージを表示する。

```
CM: ▲InByte:不正な物理アドレス 100
```

(13) VOID_PTR(TADDR)

(14) BYTE_PTR(TADDR)

(15) WORD_PTR(TADDR)

(16) DWORD_PTR(TADDR)

(17) CHAR_PTR(TADDR)

(18) SHORT_PTR(TADDR)

(19) LONG_PTR(TADDR)

パラメータ

TADDR	ターゲット CPU のメモリアドレスを表す整数値

戻り値

以下のような言語 C の型付ポインタを返す

```
typedef unsigned char      BYTE;
typedef unsigned short     WORD;
```

```
typedef unsigned long      DWORD;
void*
BYTE*
WORD*
DWORD*
char*
short*
long*
```

これらのポインタは、ターゲットメモリをシミュレートしている Windows のメモリへのポインタです。これらのポインタでシミュレーションメモリを直接操作できます。そのため、Windows を実行している x86 CPU と同じリトルエンディアンかつバイトアドレッシングの場合にだけ使用できます。

解説

ターゲット CPU のメモリアドレス TADDR（整数）を型付のポインタに変換する。Cmtoy 上では C-Machine の内部メモリ（Windows のプロセス内メモリ）へのポインタとなる。

例

ターゲット CPU の 0x8000 と 0x8002 にアクセスする場合以下のようにする。

```
#define STATUS       *WORD_PTR(0x8000)
#define MASK         *WORD_PTR(0x8002)

void SetMask(WORD mask)
{
       WORD volatile temp;
       temp = STATUS;
       MASK = mask;
}
```

未定義のターゲットメモリのアドレスにアクセスすると Windows の CPU 例外が発生する。書き込み属性のない（読み取り専用）ターゲットメモリに書き込むと Windows の例外が発生する。

(20)OR_BYTE(TADDR,DATA)

(21)OR_WORD(TADDR,DATA)

(22)OR_DWORD(TADDR,DATA)

パラメータ

TADDR	ターゲットメモリのアドレスを表す整数値
DATA	バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲットメモリの内容と DATA の OR を計算し結果をメモリに書く。この操作は LOCK で保護して途中で割込みが入らないことを保証する。

(23)AND_BYTE(TADDR,DATA)

(24)AND_WORD(TADDR,DATA)

(25)AND_DWORD(TADDR,DATA)

パラメータ

TADDR	ターゲットメモリのアドレスを表す整数値
DATA	バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲットメモリの内容と DATA の AND を計算し結果をメモリに書く。この操作は LOCK で保護して途中で割込みが入らないことを保証する。

(26)XOR_BYTE(TADDR,DATA)

(27)XOR_WORD(TADDR,DATA)

(28)XOR_DWORD(TADDR,DATA)

パラメータ

TADDR	ターゲットメモリのアドレスを表す整数値
DATA	バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲットメモリの内容と DATA の XOR を計算し結果をメモリに書く。この操作は LOCK で保護して途中で割込みが入らないことを保証する。

(29)XCHG_BYTE(TADDR,DATA)

(30)XCHG_WORD(TADDR,DATA)

(31)XCHG_DWORD(TADDR,DATA)

パラメータ

TADDR	ターゲットメモリのアドレスを表す整数値
DATA	バイト、ワードまたはダブルワードの値

戻り値

バイト、ワードまたはダブルワードの値

解説

ターゲットメモリの内容と DATA を入れ替える。メモリの内容を戻り値として返す。この操作は LOCK で保護して途中で割込みが入らないことを保証する。

5.10.3ターゲットメモリを操作するマクロ（構造体のメンバを使用する場合）

メモリマップド IO 領域に構造体を定義して、ターゲットメモリにアクセスする場合に使用します。

```
typedef volatile struct some_device_reg{
    char   com;          //コマンドレジスタ
    char   status;       //ステータスレジスタ
```

```
}DEV_REG;

DEV_REG* pDevReg = (DEV_REG*)0xff000;//先頭アドレス

void Clear()
{
    char temp = PREAD_BYTE(pDevReg->status);
}

void Init()
{
    PWRITE_BYTE(pDevReg->com, 0x80);
    PWRITE_BYTE(pDevReg->com, 0x01);
    PWRITE_BYTE(pDevReg->com, 0x40);
    PWRITE_BYTE(pDevReg->com, 0x55);
}
```

(1) PWRITE_BYTE(MEMBER,DATA)

(2) PWRITE_WORD(MEMBER,DATA)

(3) PWRITE_DWORD(MEMBER,DATA)

パラメータ

MEMBER　「構造体の先頭アドレス->メンバ」の形式で指定
DATA　バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲットメモリに値を書く。

(4) PREAD_BYTE(MEMBER)

(5) PREAD_WORD(MEMBER)

(6) PREAD_DWORD(MEMBER)

パラメータ

MEMBER　「構造体の先頭アドレス->メンバ」の形式で指定

戻り値

バイト、ワードまたはダブルワードの値

解説

ターゲットメモリの値を読む。

(7) PVOID_PTR(MEMBER)

(8) PBYTE_PTR(MEMBER)

(9) PWORD_PTR(MEMBER)

(10)PDWORD_PTR(MEMBER)

(11)PCHAR_PTR(MEMBER)

(12)PSHORT_PTR(MEMBER)

(13)PLONG_PTR(MEMBER)

パラメータ

MEMBER　「構造体の先頭アドレス->メンバ」の形式で指定

戻り値

以下のような言語 C の型付ポインタを返す

```
typedef unsigned char      BYTE;
typedef unsigned short     WORD;
typedef unsigned long      DWORD;
void*
BYTE*
WORD*
DWORD*
char*
short*
long*
```

これらのポインタは、ターゲットメモリをシミュレートしている Windows のメモリへのポインタです。これらのポインタでシミュレーションメモリを直接操作できます。そのため、Windows を実行している x86 CPU と同じリトルエンディアンかつバイトアドレッシングの場合にだけ使用できます。

解説

ターゲットメモリのアドレス（先頭アドレス->構造体のメンバ）を型付のポインタに変換する。Cmtoy 上では C-Machine の内部メモリ（Windows のプロセス内メモリ）へのポインタとなる。

(14)POR_BYTE(MEMBER,DATA)

(15)POR_WORD(MEMBER,DATA)

(16)POR_DWORD(MEMBER,DATA)

パラメータ

MEMBER　「構造体の先頭アドレス->メンバ」の形式で指定

DATA　バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲットメモリの内容と DATA の OR を計算し結果をメモリに書く。この操作は LOCK で保護して割込みが入らないことを保証する。

(17)PAND_BYTE(MEMBER,DATA)

(18)PAND_WORD(MEMBER,DATA)

(19)PAND_DWORD(MEMBER,DATA)

パラメータ

MEMBER	「構造体の先頭アドレス->メンバ」の形式で指定
DATA	バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲットメモリの内容と DATA の AND を計算し結果をメモリに書く。この操作は LOCK で保護して割込みが入らないことを保証する。

(20)PXOR_BYTE(MEMBER,DATA)

(21)PXOR_WORD(MEMBER,DATA)

(22)PXOR_DWORD(MEMBER,DATA)

パラメータ

MEMBER	「構造体の先頭アドレス->メンバ」の形式で指定
DATA	バイト、ワードまたはダブルワードの値

戻り値

なし

解説

ターゲットメモリの内容と DATA の XOR を計算し結果をメモリに書く。この操作は LOCK で保護して割込みが入らないことを保証する。

(23)PXCHG_BYTE(MEMBER,DATA)

(24)PXCHG_WORD(MEMBER,DATA)

(25)PXCHG_DWORD(MEMBER,DATA)

パラメータ

MEMBER	「構造体の先頭アドレス->メンバ」の形式で指定
DATA	バイト、ワードまたはダブルワードの値

戻り値

バイト、ワードまたはダブルワードの値

解説

ターゲット CPU メモリの内容と DATA を入れ替える。メモリの内容を戻り値として返す。この操作は LOCK で保護して割込みが入らないことを保証する。

6　コンソール・コマンド一覧

コマンドはコマンド・コンソールから使うか、スクリプトファイルに記述してバッチ処理として使います。コマンドの一般形（コマンド・シンタックス）は以下のようになります。

〈コマンド名〉　[〈パラメータ１〉[〈パラメータ２〉 …]]

- コマンド名とパラメータ、パラメータとパラメータは空白またはタブで区切る。
- 空白またはタブの直後の；（セミコロン）以降はコメントとみなす。

コマンドシンタックスの記述は、以下の記述形式に則ています。

- コマンド名は大文字、小文字を区別しない。
- {A ｜ B}では A または B のどちらかを指定する。
- [A]では A は省略可能パラメータ。
- <>で囲んだものは仮のパラメータなので、実行時には妥当な文字列に置き換える。
- <>で囲んでいないパラメータはその通り指定する。大文字、小文字は区別しない。
- -で始まるパラメータはオプションパラメータ。-に続くオプション種別を指定する文字は大文字、小文字を区別する。
- <x>は 16 進文字列。
- <ファイル名>に空白を含む場合は、”（ダブルクォート）で囲む形式が使える。

※コマンドの中には時間指定をするパラメータがありますが、それほど正確ではありません。時間計測は Windows のスケジュール（各アプリケーションプログラムにどの程度 CPU タイムを割り当てるか）に依存するのでこのようになります。

6.1 messagebox〈文字列〉

パラメータ

〈文字列〉　　メッセージボックスに表示する文字列（行末まで）

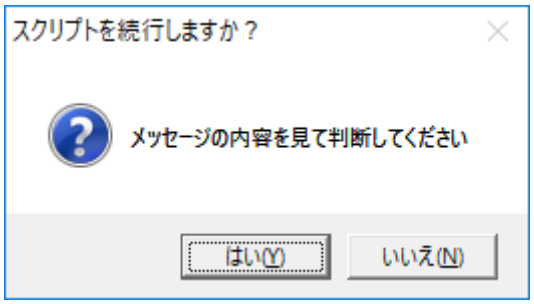
解説

スクリプトを停止してメッセージボックスを表示する。
<文字列>の先頭が”（ダブルクォート）の場合は次の”（ダブルクォート）の手前までをメッセージボックスに表示する。

例

```
messagebox メッセージの内容を見て判断してください
```

上記のコマンドを実行した場合、以下のメッセージボックスを表示する。



ここで、「はい」をクリックすれば、スクリプトは続行し、「いいえ」をクリックするとスクリプトを終了する。

6.2 win_app <ファイル名>

パラメータ

<ファイル名>　　Windows アプリケーションの実行ファイル(*.exe)またはドキュメントファイルを指定する。ファイル名に空白を含む場合はダブルクォート“で囲む。

解説

Windows アプリケーションを実行する。ファイル名がフルパスでない場合は、カレントディレクトリからの相対パスとなる。

例

テキストファイルを開く場合は、以下のように記述する。

```
win_app test.txt
```

6.3 set_script_mode { I | E | S }

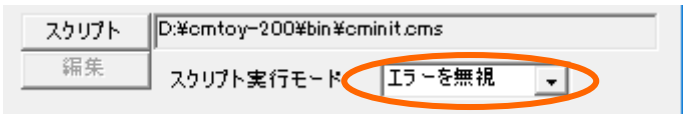
パラメータ

I	エラーを無視
E	エラーで停止
S	1 行づつ実行

解説

スクリプトの実行モードを設定する。

※　GUI の表示も変わります。



6.4 load <ファイル名>

パラメータ

<ファイル名>　　アプリケーションの DLL ファイル名（空白を含むファイル名の場合はダブルクォート“で囲む）

解説

μITRON アプリケーションをロードする。ファイル名がフルパスでない場合は、カレントディレクトリからの相対パスとなる。

例

カレントディレクトリ内の app.dll をロードする場合は、以下のように記述する。

```
load app.dll
```

※　DLL のプログラム領域、データ領域、スタック領域のメモリ上の配置位置は指定できません。Windows が Cmtoy のユーザ空間内に配置します。

6.5 reset [-t[<回数>]]

パラメータ

-t[<回数>]　　インターバルタイマを起動する。<回数>はインターバルタイマの割込み回数を 10 進数で指定。

解説

カーネルの実行を開始する。
パラメータを指定しない場合は、インターバルタイマを「手動操作」にするのでインターバルタイマの割込みは起きない。
-t を指定すると、インターバルタイマの割込みは連続的に発生する。
-t<n>を指定すると、インターバルタイマの割込みを<n>回起こしてインターバルタイマを停止する。
GUI のリセットボタンと同じ動作をさせるには以下のように指定する。

```
reset -t
```

6.6 int <レベル１> [<レベル２>]

パラメータ

<レベル１>	IRC の割込みレベル(0～15)。10 進数で指定。
<レベル２>	IRC の割込みレベル(0～15)。10 進数で指定。

解説

IRC の<レベル１>と<レベル２>へ割込み要求 IR をセットする。
レベル０の操作は timer コマンドでもできる。GUI では IR8～IR15 は操作できないが、このコマンドで操作できる。

6.7 set_interrupt_name <レベル> [<表示名>]

パラメータ

<レベル>	IRC の割込みレベル(1～7)。10 進数で指定。
<表示名>	文字列（空白は含まないこと）

解説

<レベル>で指定された GUI の割込みボタンの表示名を変更する。表示名が省略された場合は、初期状態に戻す。

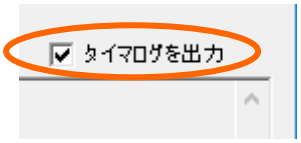
6.8 timerlog {ON | OFF}

パラメータ

ON	タイマハンドラの起動ログの出力を開始する
OFF	タイマハンドラの起動ログの出力を停止する。

解説

初期値は ON。
対応する GUI の表示を以下に示す。



6.9 timer [<回数>] [-s]

パラメータ

<回数>	タイマハンドラを起動する回数。10 進数で指定。

-s　　タイマログを出力ウインドウに表示しない

解説

IRC のレベル 0 へ割込み要求を設定する。〈回数〉で指定された回数分タイマハンドラを起動する。〈回数〉が省略されたら 1 回。-s オプションを指定した場合はタイマログを出力ウインドウに表示しない。

タイマ起動中はこのコマンドは何もしないで以下のメッセージを出力ウインドウに表示する。

;△タイマを停止してから実行してください

6.10 wait_timer [<回数>]

パラメータ

〈回数〉　　タイマハンドラの起動回数。10 進数で指定。

解説

タイマハンドラが指定された回数起動されるまで待つ。〈回数〉が省略、または 0 が指定された場合は次のタイマハンドラ起動を待つ。

6.11 setpush {UP | DOWN}

パラメータ

UP　　ボタンを離した状態にする

DOWN　　ボタンを押している状態にする。

解説

初期値は UP。

対応する GUI の表示を以下に示す。

UP

DOWN

6.12 inivolume <最大値>

パラメータ

〈最大値〉　　ボリュームの最大値（15～65535）。10 進数で指定

解説

ボリュームの最大値を設定する。初期値は 255。

対応する GUI の表示を以下に示す。

例

ボリュームの最大値を 255（8 ビット）にする場合は以下のように指定する。

```
inivolume  255
```

6.13 setvolume <現在値>

パラメータ

<現在値>　　ボリュームに設定する値。10 進数で指定

解説

ボリュームの現在値を設定する

対応する GUI の表示を以下に示す。

例

ボリューム値を５０にする場合は以下のように指定する。

```
setvolume 50
```

6.14 setswitch <スイッチ番号>,{ON|OFF}

パラメータ

<スイッチ番号>　設定するスイッチ番号。10 進数で指定

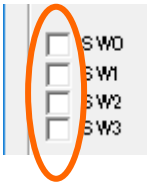
ON　　スイッチ ON（チェックを付ける）

OFF　　スイッチ ON（チェックを外す）

解説

指定したスイッチの状態を変更する。初期値は OFF。

対応する GUI の表示を以下に示す。



6.15set_switch_name <スイッチ番号> [<表示名>]

パラメータ

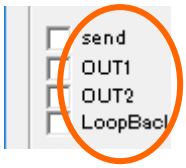
<スイッチ番号>　設定するスイッチ番号。10 進数で指定

<表示名>　　文字列（空白は含まないこと）

解説

指定したスイッチの GUI 表示名を設定する。表示名が省略された場合は、初期状態に戻す。

対応する GUI の表示を以下に示す。



6.16 ターゲットメモリ操作

ターゲットメモリとは CPU の扱うメモリ、ポートの両方を指します。メモリのアドレス空間を「メモリ空間」、ポートのアドレス空間を「IO 空間」とします。メモリ空間には CPU が実行するコード、データ、スタックセクションなどが配置されます。IO 空間には周辺装置を制御するためのレジスタ

類が配置されます。CPU によってはメモリ空間しか持たないものもあります。その場合、メモリ空間に周辺装置を制御するためのレジスタ類を配置します。このメモリ機構をメモリマップド IO と呼びます。（「1.3.3 メモリマップド IO とポートマップド IO」を参照）

ターゲットメモリを、そのメモリ属性により「領域」に分け名前（領域名）を付けて管理します。領域はアドレスの連続する部分です。C 言語のセクションと同じ考え方です。
Cmtoy では以下の属性を想定しています。

- 書き込み不可（ROM）
- 読み書き可能（RAM）
- メモリバンク領域
- メモリマップド IO 領域
- 読み書き可能な永続的メモリ
- 共有メモリ
- 機能ごとのメモリ、ポートの連続領域（デバグのしやすさなどでもよい）

メモリ空間内の領域を「メモリ領域」、IO 空間内の領域を「IO 領域」と呼ぶことにします。領域名はメモリ領域、IO 領域にかかわらず同じ名前は使えません。
領域名には空白、タブ以外に以下の文字も使えません。さらに先頭に-（マイナス）も使えません。

¥/:*?"<>|'%;.,(){}[]

これらの領域にアプリケーションプログラムからアクセスするには、「5. C-Machine の機能」で説明している C 言語の関数、マクロを使います。

6.16.1 define_mem <メモリサイズ> <IO サイズ> {BE | LE} {BA | WA}

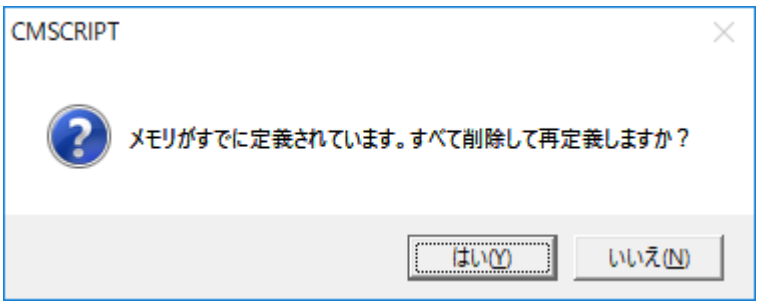
パラメータ

<メモリサイズ>	メモリ空間のアドレスサイズを 16 進数で指定
<IO サイズ>	IO 空間のアドレスサイズを 16 進数で指定
BE	ビッグエンディアン
LE	リトルエンディアン
BA	バイトアドレッシング（8 ビットメモリセル）
WA	ワードアドレッシング（16 ビットメモリセル）

解説

ターゲット CPU 固有のアドレス空間を定義する。ターゲット CPU の物理アドレスに依存するメモリをシミュレートする場合に実行しておく。メモリアドレスの範囲は 0～<メモリサイズ>-1。
一般にメモリ空間内には、RAM,ROM,EEPROM,フラッシュメモリなどを配置するのでそれらをシミュレートする場合に使用する。
IO 空間は、IO ポート機能を持つターゲット CPU の場合に指定する。IO アドレスの範囲は 0～<IO サイズ>-1。

すでに define_mem でメモリが定義されていた場合は以下のダイアログううを表示する。



ここで「はい」をクリックすると定義されていたターゲットメモリを削除して再定義する。

例

１６ビットのリトルエンディアンでバイトアドレッシングの CPU で 64K のメモリ空間だけを使う場合は以下のように指定する。

```
define_mem 10000 0 LE BA
```

１６ビットのビッグエンディアンでワードアドレッシングの CPU で 64K のメモリ空間と IO 空間を使う場合は以下のように指定する。

```
define_mem 10000 10000 BE WA
```

6.16.2 add_mem_area <領域名> <ベース> <サイズ> <バンク数> {R | RW} [-V]

パラメータ

<領域名>	メモリ領域の名前を指定
<ベース>	領域のメモリ空間内のベースアドレスを 16 進数で指定
<サイズ>	領域のメモリ空間内のアドレスサイズを 16 進数で指定
<バンク数>	バンク数を 16 進数で指定（バンクがないときは 1 を指定）
R	リードオンリ（読み取り専用）領域
RW	リード・ライト可能領域
-V	メモリマップド IO 領域

解説

メモリ空間内に RAM、EEPROM、メモリマップド IO 領域などのメモリ領域を定義する。領域は<ベース>アドレスから<サイズ>で指定される連続領域とする。

確保したメモリ領域はバイト単位に 0xff を書き込む。バンク数をを指定した場合は、バンク 0 が選択された状態となる。バンク 1 以降には全バイトにバンク番号を書き込む。つまりバンク 1 には 0x01 で、バンク 2 には 0x02 を書き込む。

例

アドレス 0x00～0xff までを”register”という領域名で登録したい場合は、以下のように指定する。

```
add_data_area register 0 100 1 RW
```

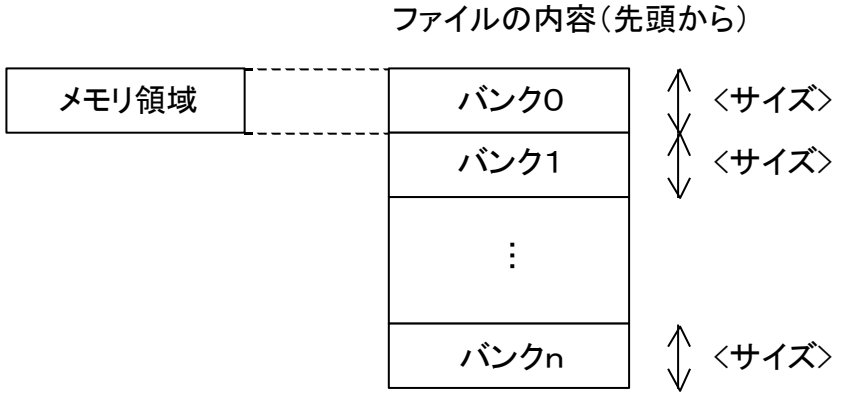
6.16.3 add_permanent_area <領域名> <ベース> <サイズ> <バンク数> {R | RW} [<ファイル名>]

パラメータ

<領域名>	メモリ領域の名前を指定
<ベース>	領域のメモリ空間内のベースアドレスを 16 進数で指定
<サイズ>	領域のメモリ空間内のアドレスサイズを 16 進数で指定
<バンク数>	バンク数を 16 進数で指定（バンクがないときは 1 を指定）
R	リードオンリ（読み取り専用）領域
RW	リード・ライト可能領域
<ファイル名>	ファイルを指定するパス名（空白を含む場合は“”で囲む）

解説

このメモリ領域はいわゆる永続的メモリ領域となり、メモリ領域の初期値は指定されたファイルの内容となる。RW が指定されている場合は、書き換えた内容は終了時にファイルの内容に書き戻す。ファイルの内容とメモリとの関係は以下のようになる。



ファイルのサイズが〈サイズ〉*〈バンク数〉より小さい場合は、〈サイズ〉*〈バンク数〉に拡張する。
ファイル名がフルパスでない場合は、カレントディレクトリからの相対パスとなる。このファイルは排他モードで開かれるのでロックされる。すでに使用されているファイルを指定するとアクセス違反となる。この領域が削除されるとファイルのロックは解除される。

例

アドレス 0xa000～0xafff までを"font"という領域名で初期値をファイル font_han2.bin を指定して登録する場合は、以下のように指定する。

```
add_data_area font a000 1000 1 R font_han2.bin
```

6.16.4 add_io_area 〈領域名〉〈ベース〉〈サイズ〉

パラメータ

〈領域名〉　　IO 領域の名前を指定
〈ベース〉　　領域の IO 空間内のベースアドレスを 16 進数で指定
〈サイズ〉　　領域の IO 空間内のアドレスサイズを 16 進数で指定

解説

IO 空間内に IO 領域を定義する。領域はベースアドレスからサイズで指定される連続領域とする。
IO 領域にはバンク機構を指定できない。

例

アドレス 0x0～0xf までを"map"という名前で登録したい場合は、以下のように指定する。

```
add_io_area map 0 10
```

6.16.5 delete_area 〈領域名〉

パラメータ

〈領域名〉　　メモリ領域、IO 領域の名前を指定

解説

〈領域名〉で指定されたメモリ領域、IO 領域を削除する。

6.16.6 erase_area 〈領域名〉

パラメータ

〈領域名〉　　メモリ領域、IO 領域の名前を指定

解説

〈領域名〉で指定されたメモリ領域、IO 領域を 0xff で初期化する。永続的メモリ領域を指定した場合は、ファイルの内容も初期化される。

例

“const”という名前で作成したメモリ領域を初期値 0xff に戻す場合は以下のように指定する。

```
erase_area const
```

6.16.7 rotate_bank 〈領域名〉 [-i〈レベル〉]

パラメータ

〈領域名〉	メモリ領域の名前を指定
〈レベル〉	IRC の割込みレベル(0～15)を 16 進数で指定

解説

〈領域名〉で指定されたメモリ領域のバンク切り替えを行う。現在のバンク番号が１なら２へと変える。最後のバンクならバンク０へ変える。-i オプションで割込み要求を設定する。
アプリケーションプログラムは割込みでバンクが切り替わったことを知る。

6.16.8 fill_bank 〈領域名〉[,〈バンク番号〉] {-PN9 | -PN15} [-init]

パラメータ

〈領域名〉	メモリ領域、IO 領域の名前を指定
〈バンク番号〉	バンク番号を 16 進数で指定
-PN9	PN 符号として PN9 を指定
-PN15	PN 符号として PN15 を指定
-init	位置を使って PN 符号を生成する

解説

〈領域名〉で指定されたメモリ領域のバンクを PN 符号で埋める。〈バンク番号〉が省略された場合はバンク０が対象となる。
-init が指定されたら初期値を使って PN 符号を生成し、省略された場合は前回の続きで PN 符号を生成する。初期値として PN9 の場合は - 1、PN15 の場合は０を使う。

6.16.9 set_bank 〈領域名〉[,〈バンク番号〉] 〈ファイル名〉 [-o〈オフセット〉]

パラメータ

〈領域名〉	メモリ領域、IO 領域の名前を指定
〈バンク番号〉	バンク番号を 16 進数で指定
〈ファイル名〉	ファイル名（空白を含むファイル名の場合はダブルクォート“で囲む）
〈オフセット〉	ファイル内の先頭からの位置

解説

〈領域名〉で指定されたメモリ領域のバンクを指定されたファイルの内容で埋める。〈オフセット〉が指定された場合は、ファイルの先頭から〈オフセット〉位置のデータで埋める。
〈バンク番号〉が省略された場合はバンク０が対象となる。
同じ〈ファイル名〉を指定して、〈オフセット〉を指定しないで連続してこのコマンドを実行すると〈オフセット〉は０から始まり、順次バンクのサイズを加えた位置からのデータでバンクメモリを設定する。

１回目のオフセットは０
２回目のオフセットはバンクサイズ
３回目のオフセットはバンクサイズ＊２

違うファイル名を指定するとオフセットは０となる。

ファイルの内容は常にバイト配列とみなすと、これをバイトアドレッシングのメモリに書き込む場合はバイト配列をバイト配列にコピーすることと同じとなる。ワードアドレッシングのメモリ（ワード配列）にコピーする場合は、以下のようになる。まず、ファイルの内容が先頭から以下のようなバイト配列とすると、

04,00,0c,00,38,00,50,00,5a,06,f2,06,…

Cmtoy ではこのファイルをワードアドレッシングのアドレス 0xa000 へ読み込んでメモリを参照したときの値はエンディアンの違いにより以下のようになる。

ファイル内オフセット	ファイルの値		メモリアドレス	リトルエンディアン	ビッグエンディアン
0	04		a000	0004	0400
1	00				
2	0c		a001	000c	0c00
3	00				
4	38		a002	0038	3800
5	00				
6	50		a003	0050	5000
7	00				
8	5a		a004	065a	5a06
9	06				
a	f2		a005	06f2	f206
b	06				

6.16.10 copy_bank <先領域名>[,<バンク番号>] <元領域名>[,<バンク番号>]

パラメータ

<先領域名>　　コピー先メモリ領域、IO 領域の名前を指定
<元領域名>　　コピー元メモリ領域、IO 領域の名前を指定
<バンク番号>　　バンク番号を 16 進数で指定

解説

バンクメモリ間で内容をコピーする。
<先領域名>と<元領域名>は違うこと。<先領域名>と<元領域名>のバンクサイズは同じこと。
<バンク番号>が省略された場合はバンク 0 が対象となる。

6.16.11 set [-{s | p}<アドレス>[,<バンク番号>]] [-i<レベル>] {[-{b[u] | w | l}] <xx> | <アスキー>} ...

パラメータ

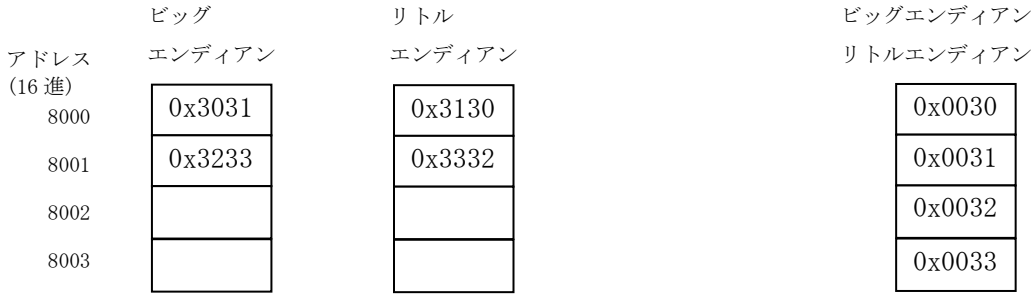
-s<アドレス>　　メモリ空間のアドレスを 16 進数で指定
-p<アドレス>　　IO 空間のアドレスを 16 進数で指定
<バンク番号>　　バンク番号（IO 空間にはバンク指定はない）を 16 進数で指定。
-i<レベル>　　IRC の割込みレベル(0～15)
-b <xx> …　　書き込むバイト列を指定する。<xx>は 16 進文字列
-bu <xx> …　　書き込むバイト列を指定する。<xx>は 16 進文字列
-w <xx> …　　書き込むワード列を指定する。<xx>は 16 進文字列
-l <xx> …　　書き込むダブルワード列を指定する。<xx>は 16 進文字列
<アスキー>　　アスキー文字列（バイト列）を指定する。アスキー文字列は”（ダブルクォート）または’（シングルクォート）で囲む。

解説

メモリ領域へバンク番号を指定して、または IO 領域へデータを書き込む。データをすべて書い

た後-i<レベル>が指定されていれば割込み要求を設定する。
<バンク番号>が省略された場合はバンク 0 が対象となる。
-s オプション、-p オプションを省略すると直前の set コマンドの続きに書き込む。
バイトアドレッシング（8 ビットメモリセル）の場合、オプション-b と-bu は同じ結果となる。
また、<アスキー>は”（ダブルクォート）または’（シングルクォート）で囲んでも同じ結果となる。
ワードアドレッシング（16 ビットメモリセル）の場合の-b オプションと-bu オプションの違い、<アスキー>の”（ダブルクォート）と’（シングルクォート）の違いは以下を参照のこと。-b オプションをパック形式、-bu オプションををアンパック形式と呼ぶことにする。

```
set -s8000 -b 30 31 32 33        set -s8000 -bu 30 31 32 33
set -s8000 “0123”                set -s8000 ‘01234’
```



パラメータをすべて省略した場合は、次のアドレス、バンク番号を表示する。

例

0 番地からバイト、ワード、ダブルワードでデータを書き込む場合は、

```
set -s8000 -b 30 31 32 33
set -s8004 -w 1234 5678
set -s8008 -l 12345678
```

または

```
set -s8000 -b 30 31 32 33 -w 1234 5678 -l 12345678
```

または

```
set -s8000 -b 30 31 32 33
set -w 1234 5678
set -l 12345678
```

とする。この後 set コマンドで次のアドレス、バンク番号が確認できる。

```
> set
 現在の値： メモリアドレス 800cH, バンク 0
```

アスキー列で指定する場合は、

```
set -s8000 "ABCDEFGH"
```

バンク番号を指定する場合は、

```
set -s8000,1 -b 30 31 32 33
```

割込みレベル 1 を発生させる場合は、

```
set -s8000,1 -i1 -b 30 31 32 33
```

I0 空間に書き込む場合は、

```
set -p0000 -w 0001
```

のようにする。

アドレスを省略すると続きに書き込む。（ビッグエンディアン、ワードアドレッシング）

```
>  set -s8020 "0123456"
;set を完了：next address = 8023H
>  set  "789ABCDEF"
;set を完了：next address = 8028H
> get -s8020 -w8
3031 3233 3435 3637 3839 4142 4344 4546
```

6.16.12 get [-{s | p}<アドレス>[,<バンク番号>] -{b | w | l | c[u]}<個数>]

パラメータ

-s<アドレス>	メモリ空間のアドレスを 16 進数で指定
-p<アドレス>	IO 空間のアドレスを 16 進数で指定
<バンク番号>	バンク番号（IO 空間にはバンク指定はない）を 16 進数でで指定
-b<個数>	読み出すバイト数を指定する。<個数>は 16 進文字列。16 進で表示。
-w<個数>	読み出すワード数を指定する。<個数>は 16 進文字列。16 進で表示。
-l<個数>	読み出すダブルワード数を指定する。<個数>は 16 進文字列。16 進で表示。
-c<個数>	読み出す文字数を指定する。<個数>は 16 進文字列。アスキー列で表示。
-cu<個数>	読み出す文字数を指定する。<個数>は 16 進文字列。アスキー列で表示。

解説

メモリ領域からバンク番号を指定して、または IO 領域の内容を読み出し表示する。
<バンク番号>が省略された場合はバンク 0 が対象となる。
-s オプション、-p オプションを省略すると直前の get コマンドの続きを読み出す。
バイトアドレッシング（8 ビットメモリセル）の場合、オプション-c と-cu は同じ結果を表示する。ワードアドレッシング（16 ビットメモリセル）の場合の-c オプションと-cu オプションの違いは例を参照のこと。

例

ワードアドレッシング、ビッグエンディアンの場合は、

```
> set -s8000 ‘0123456789ABCDEF’
> get -s8000 -w4
0030 0031 0032 0033
> get -s8000 -cu8
"01234567"
> get -s8000 -c8
".0.1.2.3"
> get -s8000 -b8
00 30 00 31 00 32 00 33

> set -s8000 "0123456789ABCDEF"
> get -s8000 -w4
3031 3233 3435 3637
> get -s8000 -cu8
"13579BDF"
> get -s8000 -c8
"01234567"
> get -s8000 -b8
30 31 32 33 34 35 36 37
```

ワードアドレッシング、リトルエンディアンの場合は、

```
> set -s8000 '0123456789ABCDEF'
```

```
> get -s8000 -w4
0030 0031 0032 0033
> get -s8000 -cu8
"01234567"
> get -s8000 -c8
"0.1.2.3."
> get  -s8000 -b8
30 00 31 00 32 00 33 00

> set -s8000 "0123456789ABCDEF"
> get -s8000 -w4
3130 3332 3534 3736
> get -s8000 -cu8
"02468ACE"
> get -s8000 -c8
"01234567"
> get -s8000 -b8
30 31 32 33 34 35 36 37
```

6.16.13 wait -{s | p}<アドレス>[,<バンク番号>] -{b | w | l} <xx>[,{OR | AND}] [-t[<タイムアウト>]]

パラメータ

-s<アドレス>	メモリ空間のアドレスを 16 進数で指定
-p<アドレス>	IO 空間のアドレスを 16 進数で指定
<バンク番号>	バンク番号（IO 空間にはバンク指定はない）を 16 進数で指定
-b <xx>	比較するバイト値を指定する。<xx>は 16 進文字列
-w <xx>	比較するワード値を指定する。<xx>は 16 進文字列
-l <xx>	比較するダブルワード値を指定する。<xx>は 16 進文字列
OR	比較する値の１のビットのどれかが一致したら待ち解除
AND	比較する値の１のビットがすべて一致したら待ち解除
<タイムアウト>	待ち時間をミリ秒指定。10 進数で指定。

解説

メモリ領域からバンク番号を指定して、または IO 領域の内容を読み出し、比較する値と OR,AND の条件で比較し、条件が一致するまで待つ。OR と AND が省略された場合は読み出した内容と比較する値が一致するまで待つ。

-t オプションを指定した場合は時間指定で条件が一致するのを待つ。省略した場合は永久待ち。それ以外はのタイムアウトの指定は以下のようになる。

-t	永久待ち指定（省略した場合と同じ）
-t0	ポーリング指定
-t<時間>	指定した時間まで待つ

例

```
wait -p0 -w 0101,OR
wait -s8004,1 -w 1004,AND
wait -s8004,1 -w 1234
```

6.17 serial〈シリアルポート番号〉〈サブコマンド〉

パラメータ

〈シリアルポート番号〉　設定するポート番号。10 進数で指定
〈サブコマンド〉　　サブコマンド

解説

指定したシリアルポート番号へサブコマンドを発行する。
サブコマンドには{init ｜ info ｜ push ｜ set ｜ probe}がある。

6.17.1 init〈割込みレベル〉〈チップ種別〉

パラメータ

〈割り込みレベル〉　シリアルポートに割り当てる割込みレベル。10 進数で指定
〈チップ種別〉　　16550 を指定（シミュレートするシリアルコントローラ種別）。

解説

シリアルポートを簡易シリアルから特定のシリアルコントローラをシミュレートする機能に切り替える。現在は 16550 相当のシミュレートのみ可能。
起動時は簡易シリアル（チップ種別＝０）となっている。

例

シリアル 2 を 16550 相当の機能に変更し、割込みレベル 4 を割り当てる場合は以下のように指定する。

```
serial 1 init 4 16550
```

このとき GUI 上のシリアル 2 の表示は以下のように変わる。

6.17.2 info

パラメータ

なし

解説

シリアルポートの設定情報を表示する。

例

```
> serial 1 init 5 16550
> serial 1 info
;serial 1 info : type = 16550, irq = 5, TCP/IP port = 701。
```

6.17.3 set {CTS ｜ DSR ｜ RI ｜ DCD} {ON ｜ OFF}

パラメータ

CTS	Clear to Send
DSR	Data Set Ready
RI	Ring Indicator
DCD	Data Carrier Detect
ON	信号を１に設定する
OFF	信号を０に設定する

解説

16550 の入力ピンへの信号状態を設定する。16550 のプロパティウインドウからも設定できる。

例

シリアル 2 の CTS を on にする場合は以下のように指定する。

```
serial 1 set CTS on
```

6.17.4 push {<xx> | "<アスキー>" | -b | -e} ...

パラメータ

<xx>	16 進数（8 ビット値）
<アスキー>	アスキー文字列（空白を含む）
-b	ブレークキャラクタ受信を受診 FIFO に設定
-e	直後の受信文字と同時に受信エラーを受診 FIFO に設定

解説

16550 の受信 FIFO へデータを挿入する。

-b オプションは受信 FIFO にデータ 0x00 を設定してブレークキャラクタ受信を設定する。

-e オプションは次に受信 FIFO にデータを設定するときにパリティエラー、フレーミングエラーを設定する。

受信 FIFO があふれた場合はオーバーランエラーをラインステータスレジスタに設定する。

※16550 の仕様によりブレークキャラクタ受信、パリティエラー、フレーミングエラーは該当データが受信 FIFO の先頭に移動したときにラインステータスレジスタに反映される。

例

シリアル 2 の受信 FIFO へ文字列"12345" と改行コードを挿入する場合は以下のように指定する。

```
serial 1 push “12345” 0a 0d
```

シリアル 2 の受信 FIFO へ 0a を設定するときに、パリティエラー、フレーミングエラーを起こす場合は以下のようにする。

```
serial 1 push “12345” –e 0a 0d
```

6.17.5 probe {DTR | RTS | OUT1 | OUT2} [–w[<タイムアウト>]]

パラメータ

DTR	Clear to Send
RTS	Data Set Ready
OUT1	Ring Indicator
OUT2	Data Carrier Detect

-w<タイムアウト>待ち時間をミリ秒指定。10 進数で指定。

解説

16550 の出力ピンの信号状態を読み取る。これらの信号は 16550 のプロパティウインドウでも見ることができる。

-w オプションを指定した場合は信号が変化するまで待つ。省略した場合はその時点での信号状態を表示する（ポーリング指定）。それ以外はのタイムアウトの指定は以下のようになる。

-w	永久待ち指定
-w0	ポーリング指定（省略した場合と同じ）
-w<時間>	指定した時間まで待つ

例

シリアル 2 の DTR の状態を読み取る場合には以下のように指定する。結果は出力ウインドウに表示される。

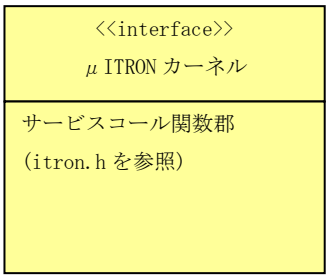
```
serial 1 probe DTR
```

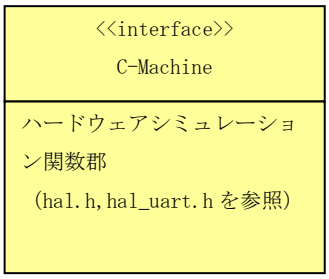
7　μITRON チュートリアル

コンピュータシステムはハードウェアとソフトウェアで構成されています。
ここでは、ソフトウェアをマルチタスク・プログラムの技術を使って開発する場合に考慮すべきことを理解する手助けとなるプログラミング例を示します。簡単なタスク構成の例からより複雑な例を取り上げて、μITRON カーネルの動きを理解する手助けをします。実際のソースコードを示しながらサービスコールの使い方を示します。そこで各サンプルプログラムを以下の視点から説明します。

・システム要求仕様　　何をするシステムかを説明する。UML ユースケース図も使う。
・システム分析　　システムを構成する静的なプログラムとデータ構成、時間軸に沿った振る舞いを UML のクラス図、状態図を使い分析する。
・実装設計　　タスク構成、オブジェクトの割り当て、割込み定義、ファイル構成など

ここで作成するμITRON アプリケーションは、外部プログラムとしてμITRON カーネ(kpdll.dll)と C-Machine(cm.dll)を前提としています。これらの外部プログラムは、UML インターフェイスとして以下のように定義してこれ以後のシステム分析で使用します。





μITRON カーネルオブジェクトの定義（ファイル kernel_cfg.c）を以下のクラスであらわすことにします。

注）ここで使用する UML の記述は、簡略化しているので厳密には UML 規格に合っていない部分があるかもしれません。

チュートリアルでは以下の各ステップに沿って段階的に複雑な例の説明をします。

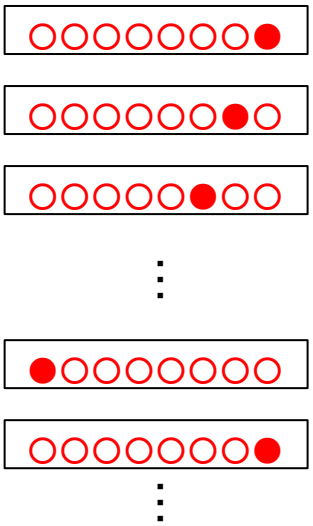
ステップ	概要	使用するサービスコール
1	1 秒間隔で LED の点灯位置を左隣へ変える	**tslp_tsk ext_tsk**
2	デバッグタスクを追加する	tslp_tsk ext_tsk **ref_tsk ref_sem ref_flg ref_mbx ref_mpf**
3	タイマタスクを追加する	tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx ref_mpf **snd_msg rcv_msg vchg_ifl vget_ifl**
4	割り込みハンドラを追加する。	tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx ref_mpf snd_msg rcv_msg vchg_ifl vget_ifl **sig_sem wai_sem**
5	割込みレベル 1 と割込みレベル 2 の割り込みで LED の点灯数を増減する。 システム起動時からの秒数を 10 進数で 1 秒おきに表示器に設定する。 プッシュボタンが押されている間はボリューム値を 16 進数で表示器に設定する。 プッシュボタンの UP/DOWN は 20ms 間隔で 3 回連続したら確定とする。	tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx ref_mpf snd_msg rcv_msg vchg_ifl vget_ifl **iset_flg wai_flag**
6	メモリプール機能を使ってステップ 5 のプログラムを書き換える。	tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx rel_mpf snd_msg rcv_msg vchg_ifl vget_ifl iset_flg wai_flag **pget_mpf ref_mpf**

μITRON を使ったプログラム開発については以下の書籍も参考にしてください。各ステップのより詳細な解説、考察をしています。
μITRON」入門―“組み込み系”「リアルタイム OS」の基礎（I・O BOOKS）　工学社

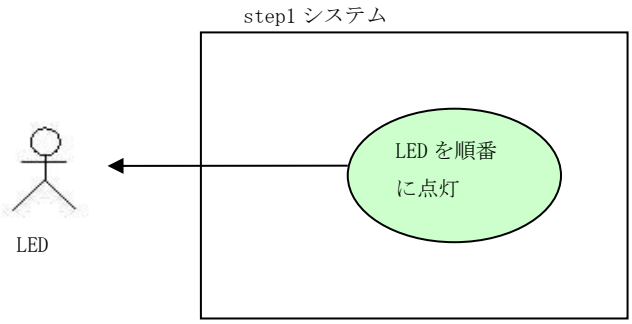
7.1 ステップ 1 (app1.dll)

7.1.1システム要求仕様

1 秒間隔で LED の点灯位置を左隣へ変える。
左端までいったら右端を点灯する。

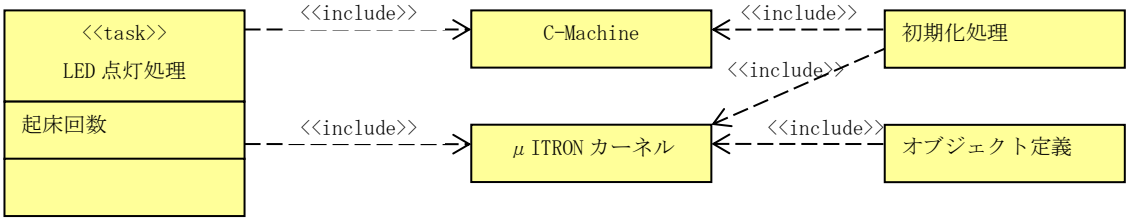


(1) ユースケース図

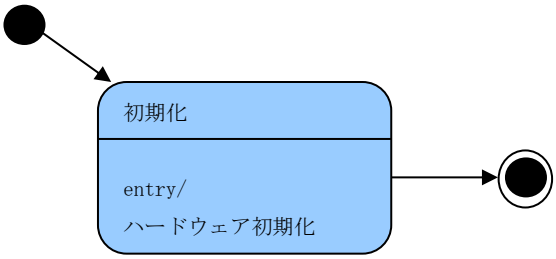


7.1.2システム分析

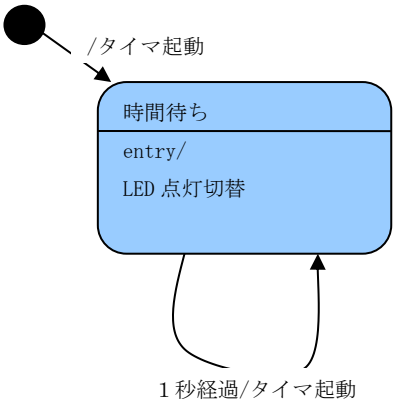
(1) クラス図



(2) 初期化処理の状態図



(3) LED 点灯処理の状態図



7.1.3実装設計

タスク構成

初期化処理	InitTask	LED の初期化（すべて消灯）
LED 点灯処理	LedTask	1 秒に 1 回点灯 LED を変える。

解説

InitTask は優先度 1 で必ず 1 番最初に動くタスクとする。初期化が終わったらタスクを終了する。

LedTask は、サービスコール tslp_tsk で 20 ミリ秒おきに起床する。50 回起床すると 1 秒なので、LED の点灯位置を変える。

LedTask が待ち状態の間はカーネル内部で用意してあるアイドルタスクが動いている（CPU を占有している）。アイドルタスクの優先度が最も低いことはコンフィギュレーションファイルでのタスク登録で指定している。

ファイル構成

Step1¥	
Kernel_cfg.c	μITRON コンフィギュレーションファイル
init.c	初期化タスク
led1.c	1 秒おきに LED を更新するタスク
App1¥	
App.sln	app1.dll を作成する Visual C++ 2005 のソリューションファイル

App.dsw	app1.dll を作成する VisualStudio6.0 のプロジェクトファイル
Makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル

使用サービスコール

tslp_tsk ext_tsk

7.2 ステップ２(app2.dll)

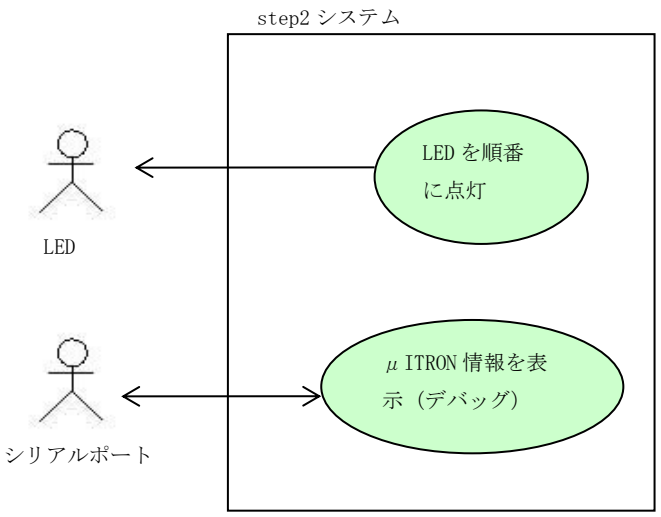
7.2.1システム要求仕様

ステップ１のシステムにデバッグタスクを追加する。
デバッグタスクはシリアルポートの先にハイパーターミナルが接続されていると想定する。
ハイパーターミナルからデバッグコマンドを使ってシステムの状態が参照できるようにする。
デバッグコマンドして以下のものを定義する。

help	コマンド一覧の表示
ver	デバッグタスクとμITRON カーネルのバージョン表示
d <addr> [<count>]	メモリ　ダンプ
inb <port>	ポートからバイトリード
inw <port>	ポートからワードリード
outb <port> <byte>	ポートへバイトライト
outw <port> <word>	ポートへワードライト
tsk [tskid]	タスク（一覧）の表示
sem [semid]	セマフォ（一覧）の表示
flg [flgid]	イベントフラグ（一覧）の表示
mbx [mbxid]	メイルボックス（一覧）の表示
mpf [mpfid]	固定長メモリプール（一覧）の表示

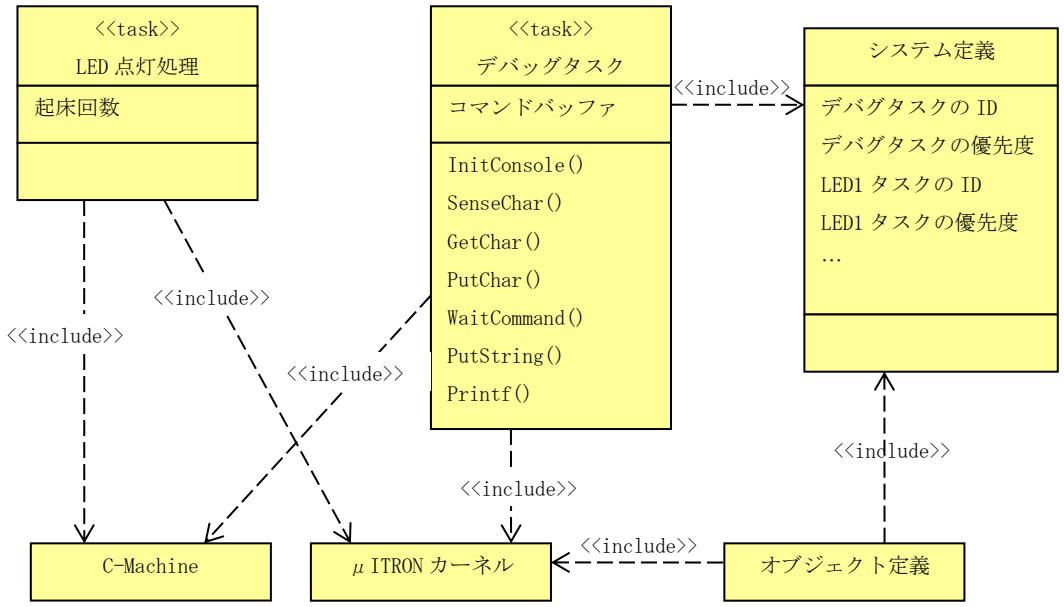
※ハイパーターミナルの代わりに PuTTY などの端末エミュレータソフトも使えます。

(1) ユースケース図



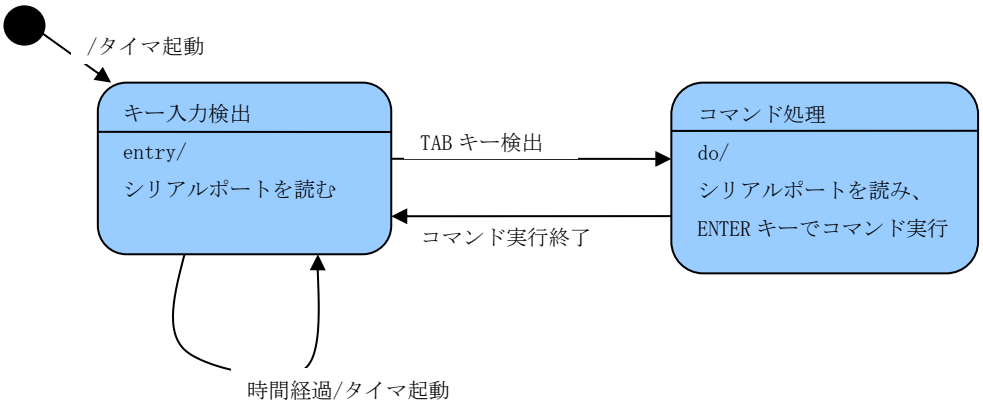
7.2.2システム分析

(1) クラス図



ここでは、タスク ID やタスク優先度などシステム全体で重複のないように決めないといけない定数をクラス「システム定義」として分離しています。（初期化処理は省略）

(2) デバッグタスクの状態図



7.2.3実装設計

タスク構成

初期化処理	InitTask	LED の初期化（すべて消灯）、シリアルポート初期化
LED 点灯処理	LedTask	1 秒に 1 回点灯 LED を変える。
デバッグタスク	DebugTask	タスクなどの状態を表示する

解説

デバッグタスクは、以下の仕事をする。
①シリアルポートを定期的(20 ミリ秒)に監視する。

②TAB コード（09H）を検出したら文字列‘CLI> ’を送信し、シリアルポートから CR（キャリッジリターン）コードまでを読み取りバッファに格納する。この文字列をコマンドと呼ぶ。
③CR コードを受け取った時点でコマンド文字列を解析して、実行する。
④コマンド実行後は、①に戻る。

シリアルポートからの読み取りには割り込みは使わない。
デバッグタスクの優先度は 2 とする（初期化タスクより低く、他のタスクより高くする）。
②および③の間はデバッグタスクが CPU を占有し続けるので他のタスクは動けない。

ファイル構成

Step2¥	
Kernel_cfg.c	μITRON コンフィギュレーションファイル
init.c	初期化タスク
led1.c	1 秒おきに LED を更新するタスク
debug.c	シリアルポートを使ってハイパーターミナルと通信するタスク
debug.h	
system_def.h	オブジェクト ID 定義
App2¥	
App.sln	app2.dll を作成する Visual C++ 2005 のソリューションファイル
App.dsw	app2.dll を作成する VisualStudio6.0 のプロジェクトファイル
Makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル

使用サービスコール

tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx ref_mpf

7.2.4ハイパーターミナルの設定方法

Cmtoy はシリアルポートを Winsock(TCP/IP)でシミュレートするので、Cmtoy を実行している同じ PC のハイパーターミナルと接続するためには、ハイパーターミナルを以下のように設定してください。

「ASCII 設定」では、「受信データに改行文字を付ける」をチェックしてください。

Cmtoy を起動し、ハイパーターミナルをこのように設定してこのサンプル app2.dll を「ロード」して、「リセット」するとハイパーターミナルに以下の文字列が表示されます。

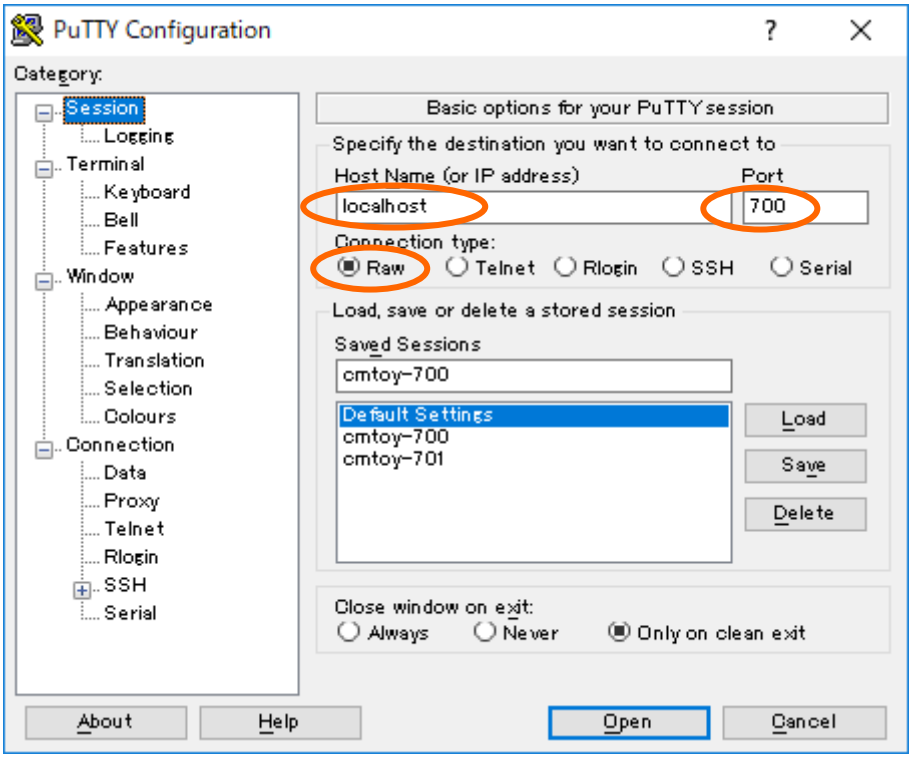
```
Debug CLI, V1.00
```

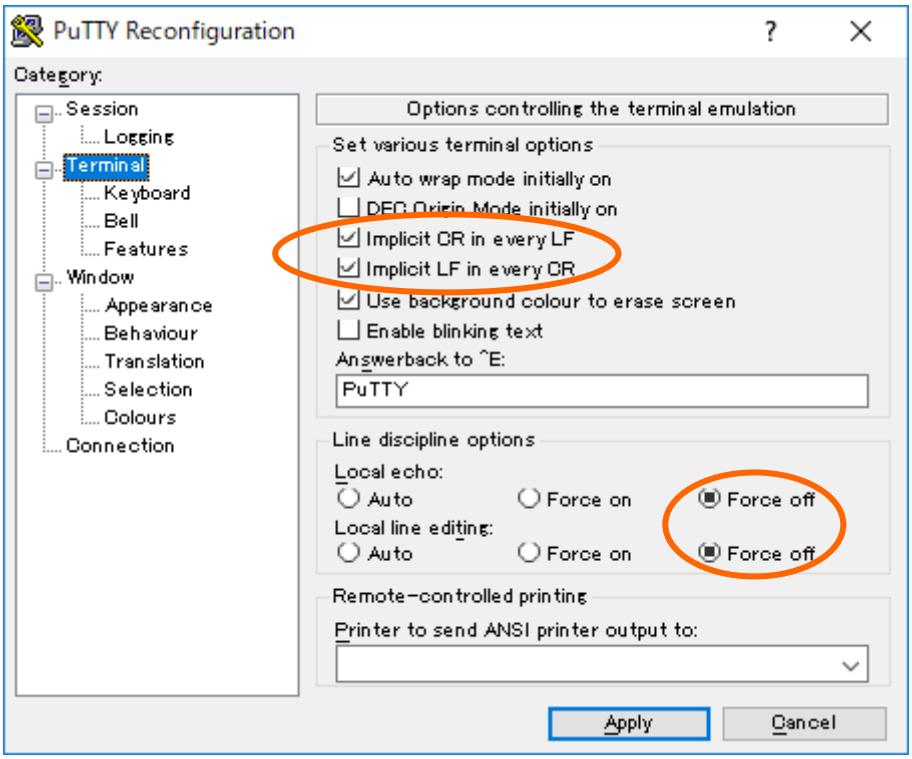
ここでハイパーターミナルから TAB キーを入力すると以下のようになります。

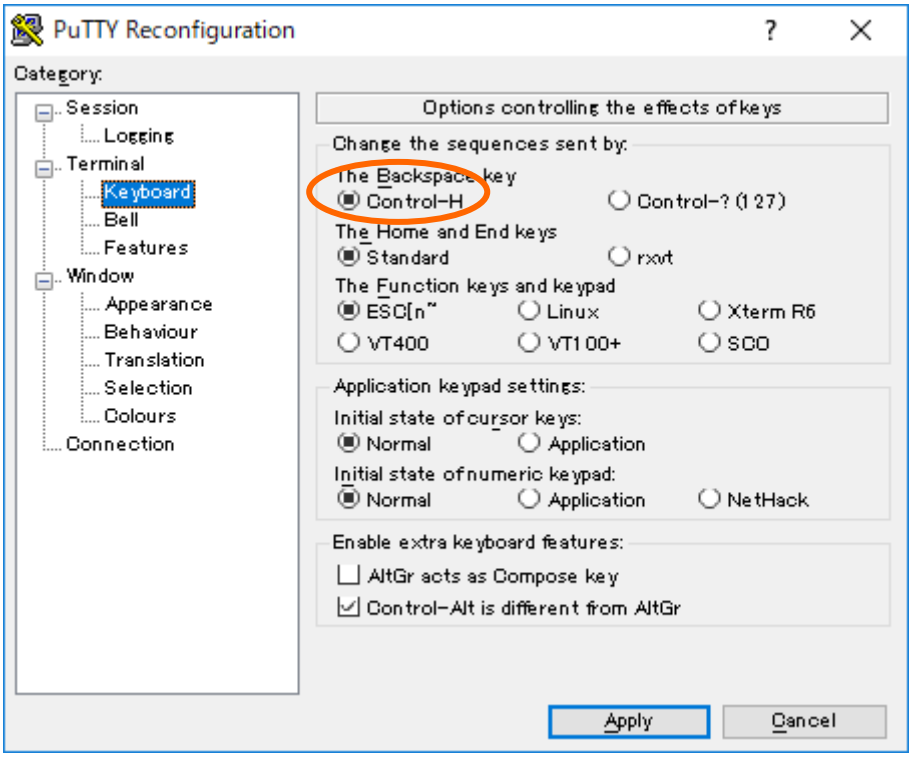
ここで‘help’と入力し最後に ENTER キーを押すと以下のように、コマンドリストが表示されます。

7.2.5 PuTTY の設定方法

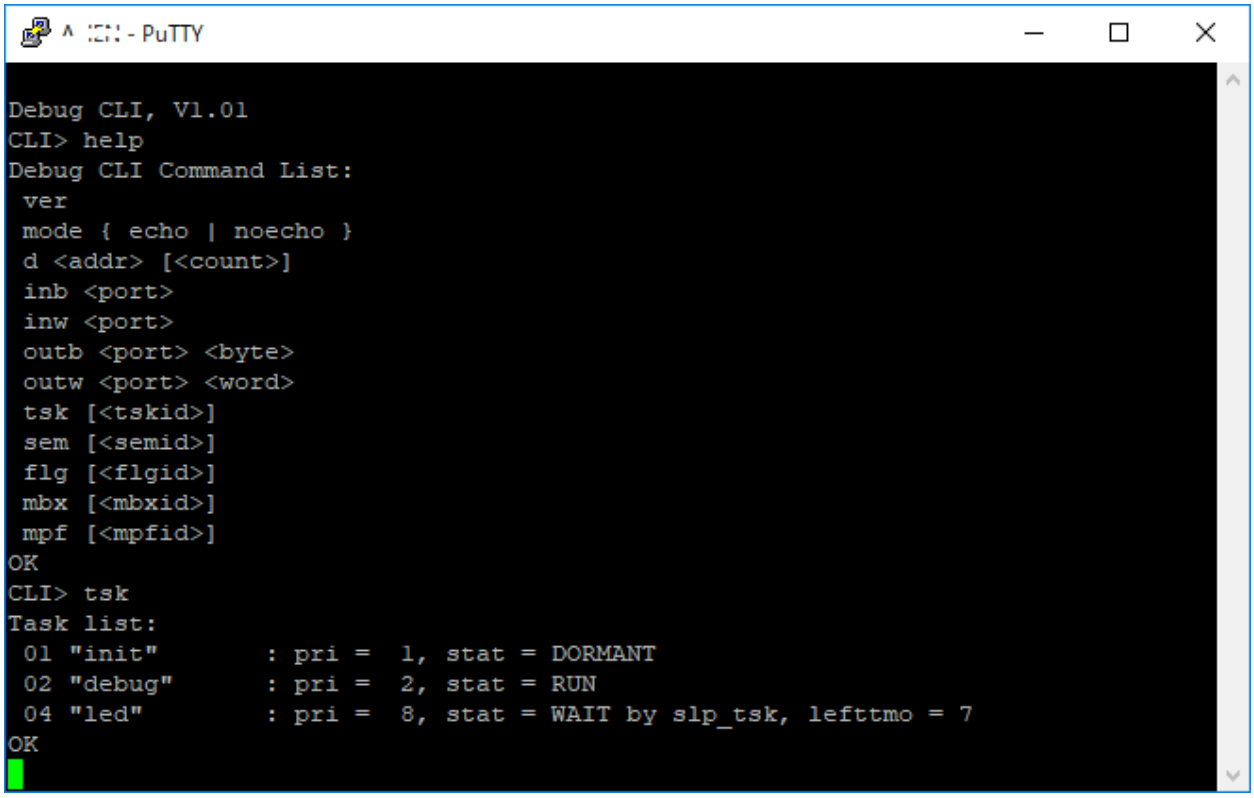
ハイパーターミナの代わりに PuTTY を使用する場合は以下のように設定してください。
PuTTY は/pˈʌɾi/と発音するらしい。日本語では「パティ」、「プッティ」、「プティ」などと呼ばれるらしい。







PuTTY から TAB キーを入力して‘help’と入力し最後に ENTER キーを押すと以下のように、コマンドリストが表示されます。

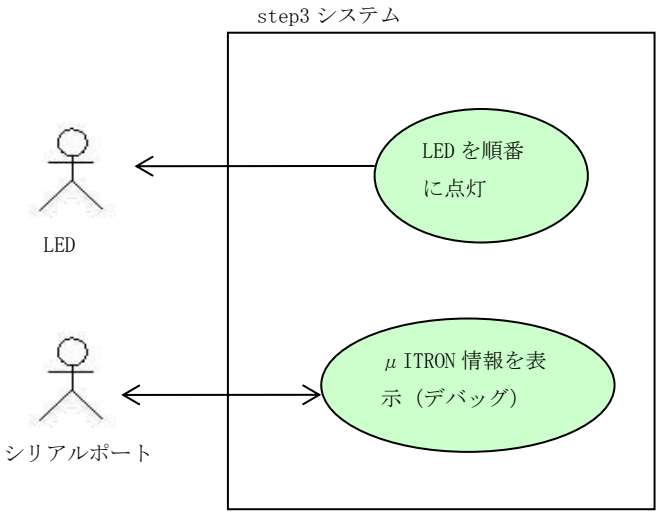


7.3 ステップ３(app3.dll)

7.3.1システム要求仕様

ステップ２の LED 点灯タスクを汎用タイマタスクからのメッセージ受信で動作させるように変更する。

(1) ユースケース図



(2) タイマ管理方法

以下の前提の下にタイマタスクの機能を説明します。

- システム内で用途ごとに使用するタイマ番号（0～15）を決めておく。
- タイムアウトメッセージの領域は、用途ごとにタスク側で確保しておく。
- 送信メッセージのヘッダ部の形式はシステム内で統一しておく。

各タスクは、

タイマ番号
タイムアウトメッセージを受け取るメイルボックス ID
タイムアウトメッセージ領域（アドレス）
タイムアウト値

を指定して、timSetTimer 関数でタイマを設定します。（この時点ではタイマは停止中）
このタイマは、以下のようなタイマ管理テーブルで管理されています。

タイマ管理テーブル

タイマ番号	現在のカウント値	タイムアウト値	送信先メイルボックスＩＤ	送信メッセージ（アドレス）
０	１０	１００	１	ｘｘｘｘ
１	０	５０	２	ｙｙｙｙ
２	０	０	０	０ (NULL)

「現在のカウント値」が０以外のタイマは起動中。（上記のタイマ番号０）
「タイムアウト値」が０以外で「現在のカウント値」が０のタイマは停止中。（上記のタイマ番号１）
「タイムアウト値」が０のタイマは未定義。（上記のタイマ番号２）

(3) タイマの起動とタイムアウト通知の方法

タイマの起動、タイムアウトの通知は以下のように行われます。

① timStartTimer 関数は、「現在のカウント値」に「タイムアウト値」を設定する。
② タイマタスクは一定時間おきに起動中のタイマの「現在のカウント値」を－１し、０になったら「送信先メイルボックス ID」で指定されるメイルボックスに「送信メッセージ（アドレス）」を送信(snd_msg)する。

(4) LED 点灯処理

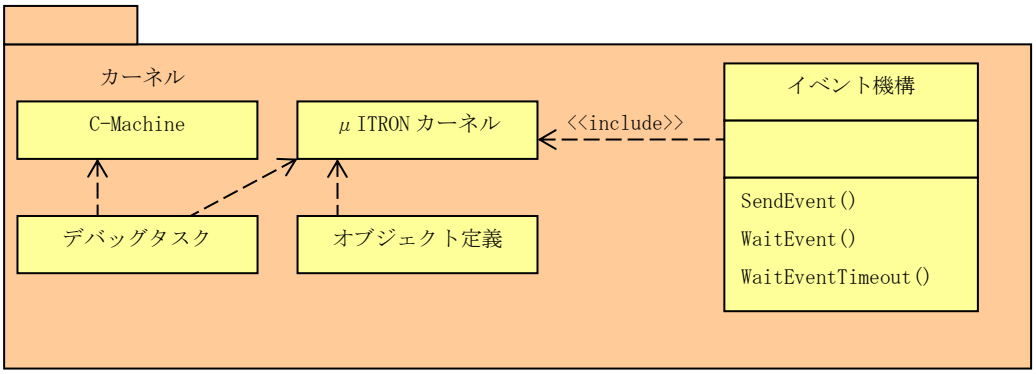
LED 点灯処理は、以下のように変更します。

起動時に
　　timSetTimer 関数でタイマ０を設定
　　timStartTimer 関数でタイマ起動
以後以下を繰り返す。
　　メッセージを受信したら LED の点灯位置を変更
　　timStartTimer 関数でタイマを再起動

7.3.2システム分析

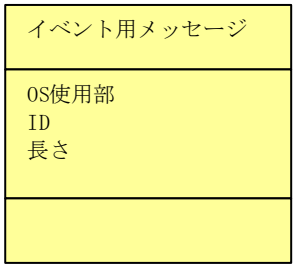
(1) クラス図

ここでは、タイマタスクを導入して汎用タイマ機能を実装します。タイムアウト通知も UML イベントとして扱うために「イベント機構（イベント送信、イベント待ち）」を導入します。
以後、C-Machine、μITRON カーネル、デバッグタスク、イベント機構、オブジェクト定義をパッケージ「カーネル」としてクラス図に記述します。（初期化処理は省略）

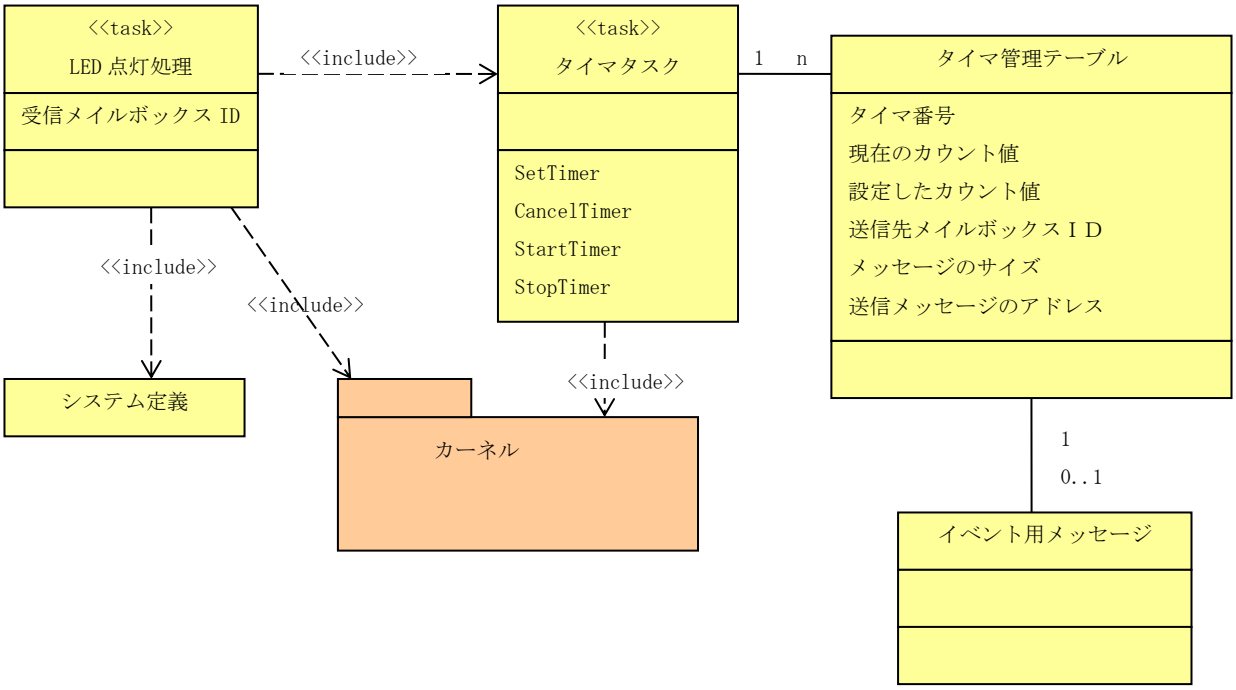


以降のシステム分析ではこの「カーネル」パッケージを使い説明します。

イベント機構はμITRON のメイルボックスとメッセージを使って実装します。そのため、メッセージの先頭部分にイベント機構に必要な構造を持たせます。

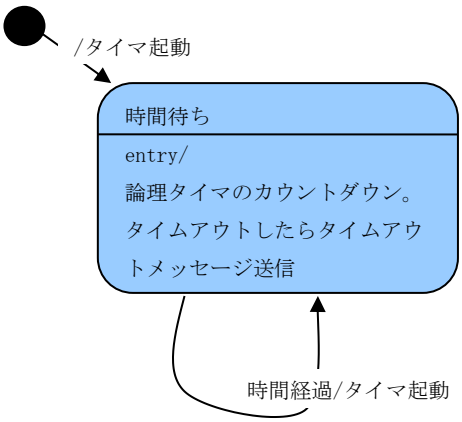


このイベント用メッセージを使いシステム分析結果を以下に示します。

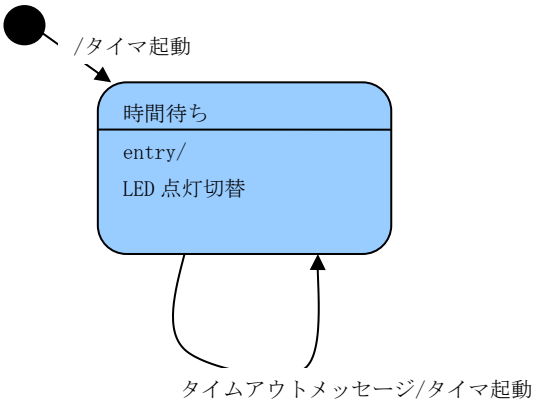


以後、C-Machine、μITRON カーネル、デバッグタスクをパッケージ「カーネル」としてクラス図に記述します。

(2) タイマタスクの状態図



(3) LED 点灯処理の状態図



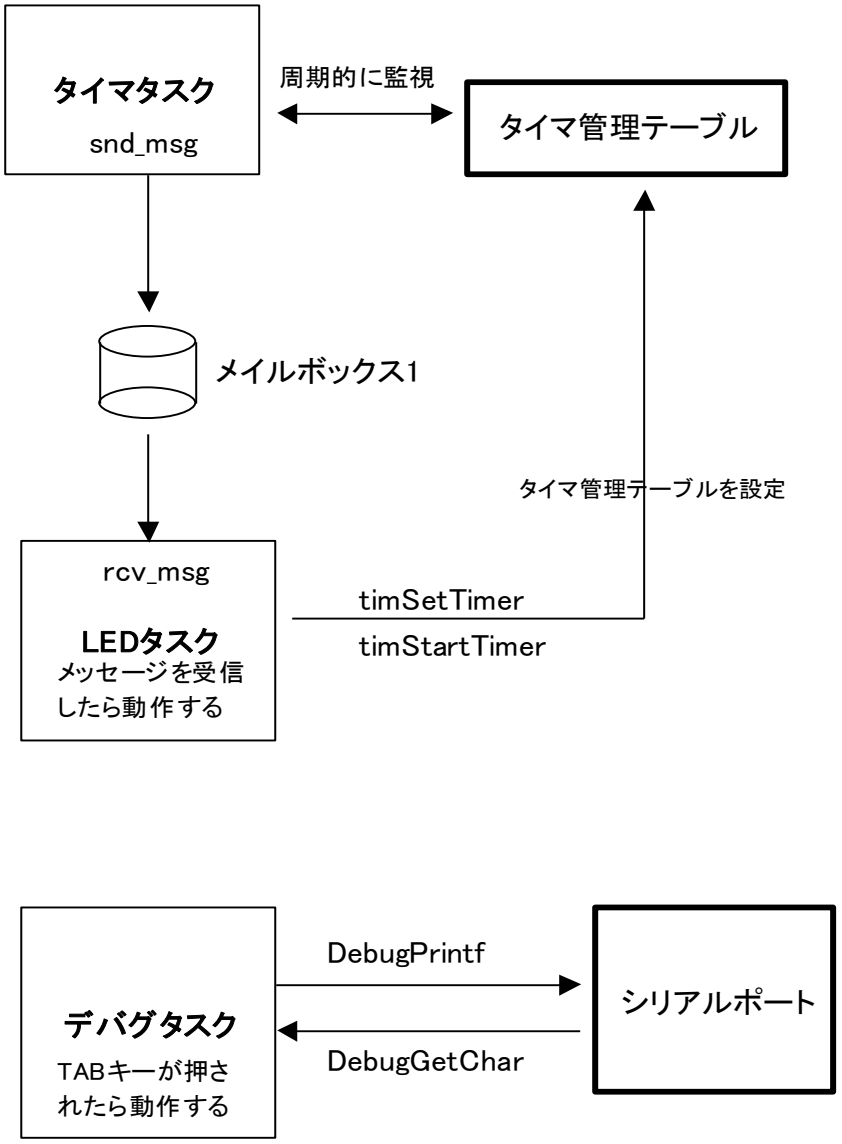
7.3.3実装設計

タスク構成

初期化処理	InitTask	LED の初期化（すべて消灯）
タイマタスク	TimerTask	汎用的な時間管理タスク
LED 点灯処理	LedTask	メッセージ受信で点灯 LED を変える。
デバッグタスク	DebugTask	タスクなどの状態を表示する

解説

このシステムのタスク間の関係は以下の図のとおりです。



ファイル構成

Step3¥

Kernel_cfg.c	μITRON コンフィギュレーションファイル
init.c	初期化タスク
timer.c	汎用タイマタスク
timer.h	
event.c	イベント機構
event.h	イベント用メッセージ
led3.c	1 秒おきに LED を更新するタスク
debug.c	シリアルポートを使ってハイパーターミナルと通信するタスク
debug.h	

system_def.h	オブジェクト ID 定義
App3¥	
App.sln	app3.dll を作成する Visual C++ 2005 のソリューションファイル
App.dsw	app3.dll を作成する VisualStudio6.0 のプロジェクトファイル
Makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル

使用サービスコール

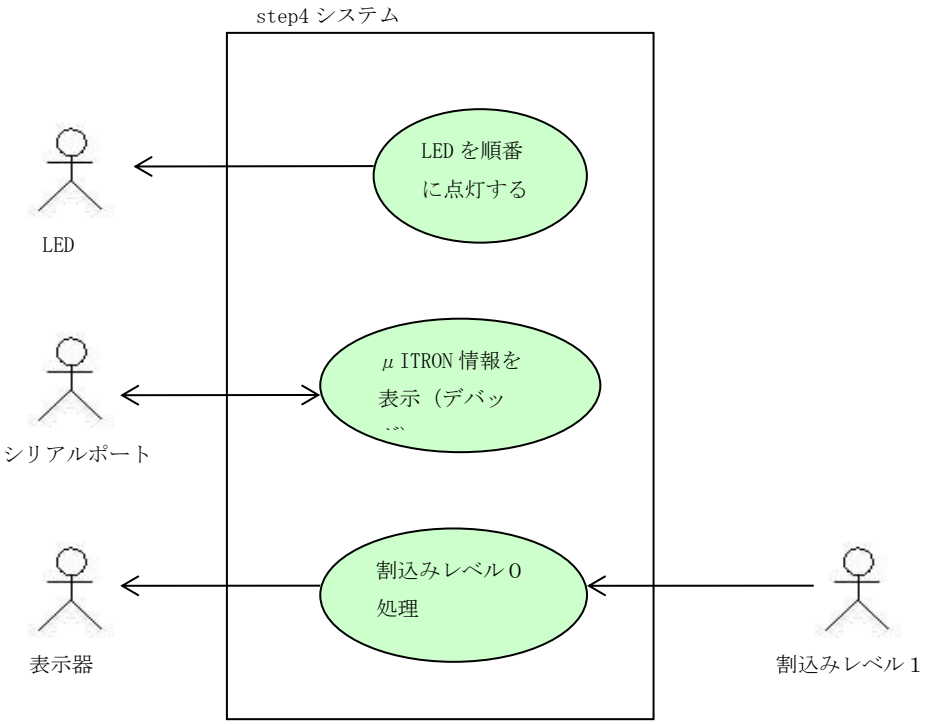
tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx ref_mpf snd_msg rcv_msg vchg_ifl vget_ifl

7.4 ステップ４(app4.dll)

7.4.1システム要求仕様

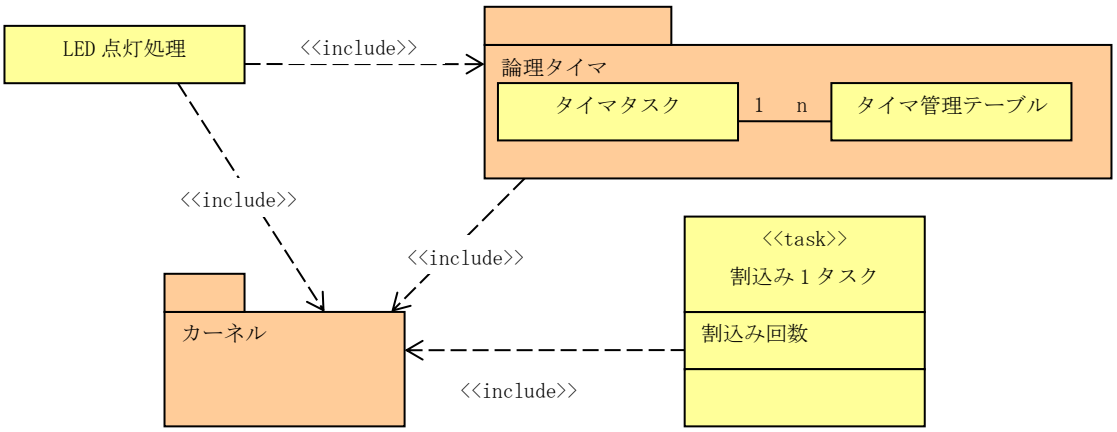
割込みレベル１の割り込みが発生したら割り込み発生数を表示器（８セグメント LED）に設定する。

(1) ユースケース図



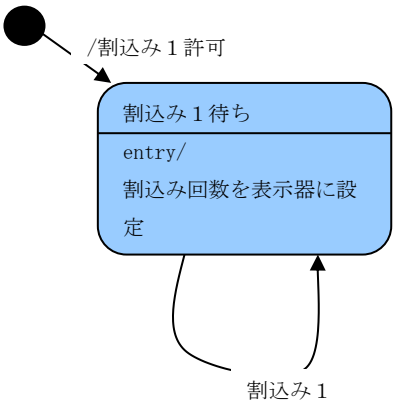
7.4.2システム分析

(1) クラス図



以後、タイマタスクとタイマ管理テーブルをパッケージ「論理タイマ」としてクラス図に記述します。（初期化タスクは省略）

(2) 割込み 1 処理の状態図



7.4.3実装設計

タスク構成

初期化処理	InitTask	LED の初期化（すべて消灯）
タイマタスク	TimerTask	汎用的な時間管理タスク
LED 点灯処理	LedTask	メッセージ受信で点灯 LED を変える。
デバッグタスク	DebugTask	タスクなどの状態を表示する
割込み 1 タスク	Int1Task	セマフォ 1 で待って割込み回数を表示器に表示

割込みハンドラ構成

割込み 1	Int1Handler	セマフォ 1 へ信号操作

解説

割込みレベル 1 の割込みハンドラからセマフォ 1 を使って割込みタスクを起動する。

ファイル構成

Step4¥

Kernel_cfg.c	μITRON コンフィギュレーションファイル
init.c	初期化タスク
timer.c	汎用タイマタスク
timer.h	
event.c	イベント機構
event.h	イベント用メッセージ
led3.c	1 秒おきに LED を更新するタスク
irq1.c	割込み 1 タスク、INT1 割込みハンドラ
debug.c	シリアルポートを使ってハイパーターミナルと通信するタスク
debug.h	
system_def.h	オブジェクト ID 定義
App4¥	
App.sln	app4.dll を作成する Visual C++ 2005 のソリューションファイル
App.dsw	app4.dll を作成する VisualStudio6.0 のプロジェクトファイル
Makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル

使用サービスコール

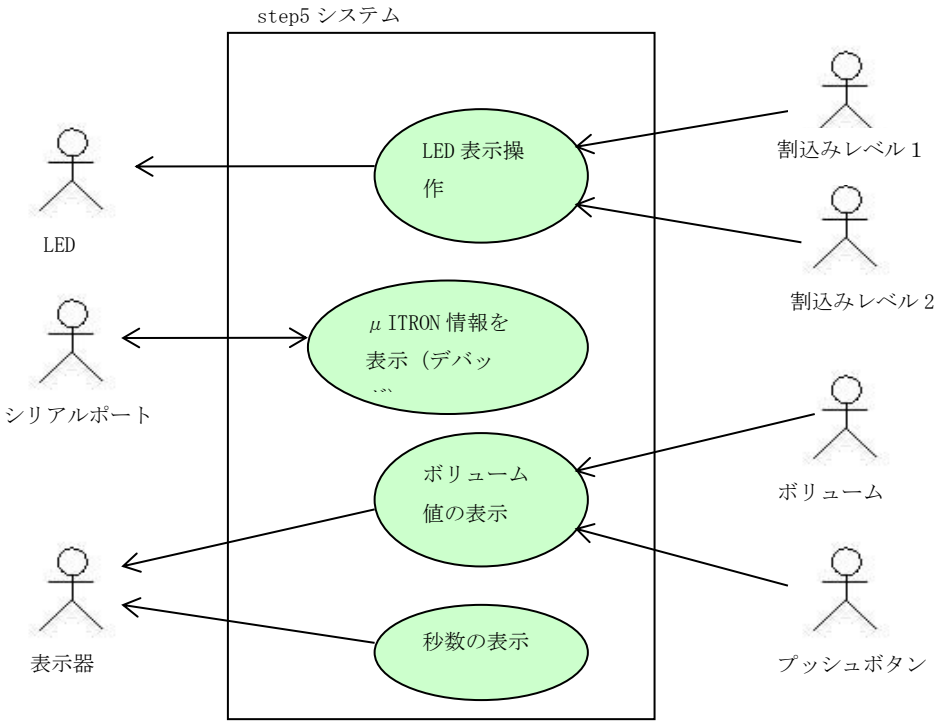
tslp_tsk ext_tsk ref_tsk ref_sem isig_sem wai_sem ref_flg ref_mbx snd_msg rcv_msg vchg_ifl vget_ifl ref_mpf

7.5 ステップ５(app5.dll)

7.5.1システム要求仕様

割込みレベル１と割込みレベル２の割り込みで LED の点灯数を増減する。
システム起動時からの秒数を 10 進数で 1 秒おきに表示器に設定する。
プッシュボタンが押されている間はボリューム値を 16 進数で表示器に設定する。
プッシュボタンの UP/DOWN は 20ms 間隔で 3 回連続したら確定とする。

(1) ユースケース図



(2) LED 操作

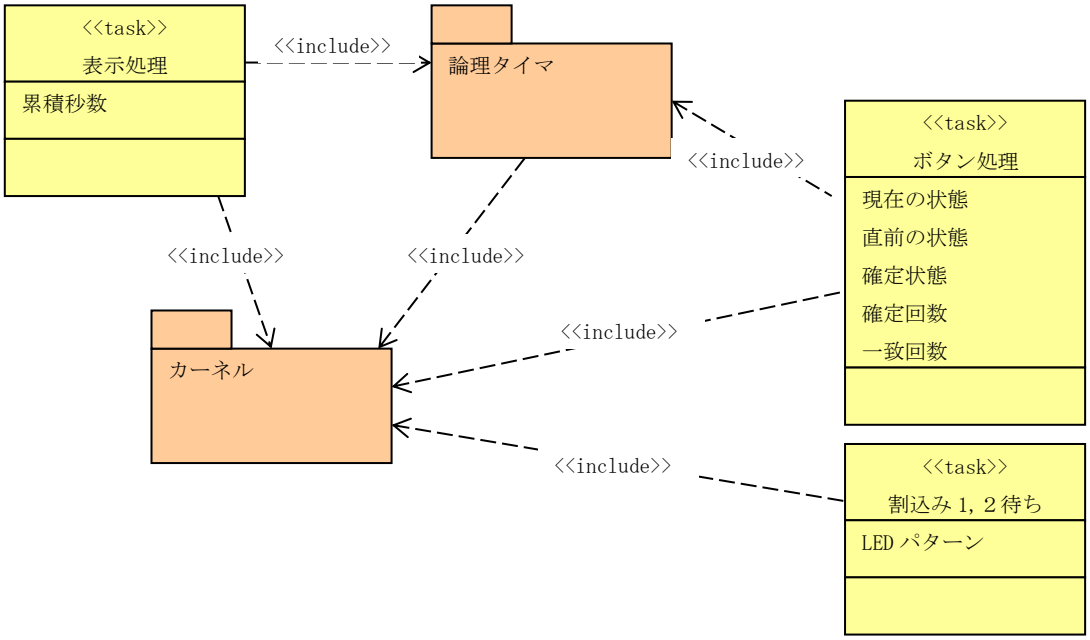
8 個の LED を使って 9 段階の状態を表示する。○は LED の OFF、●は ON を表す。

段階	LED の点灯、消灯位置	
8	● ● ● ● ● ● ● ●	すべて ON
7	● ● ● ● ● ● ● ○	
6	● ● ● ● ● ● ○ ○	
5	● ● ● ● ● ○ ○ ○	
4	● ● ● ● ○ ○ ○ ○	
3	● ● ● ○ ○ ○ ○ ○	
2	● ● ○ ○ ○ ○ ○ ○	
1	● ○ ○ ○ ○ ○ ○ ○	
0	○ ○ ○ ○ ○ ○ ○ ○	すべて OFF

システム起動時は段階４の状態とする。
INT1 の割込みで１つ上の段階となる。段階８で INT1 が発生してもすべて OFF で変わらない。
INT2 の割込みで１つ下の段階となる。段階０で INT2 が発生してもすべて ON で変わらない。

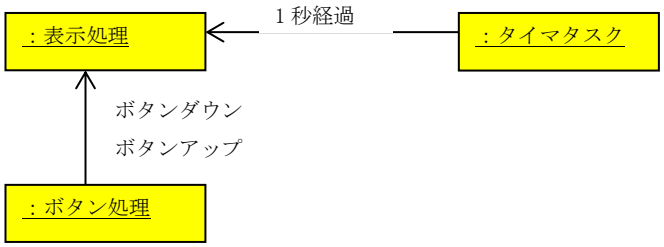
7.5.2システム分析

(1) クラス図

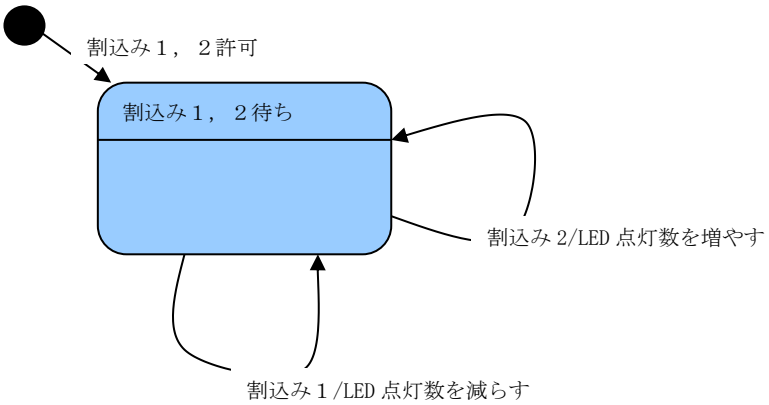


(初期化タスクは省略)

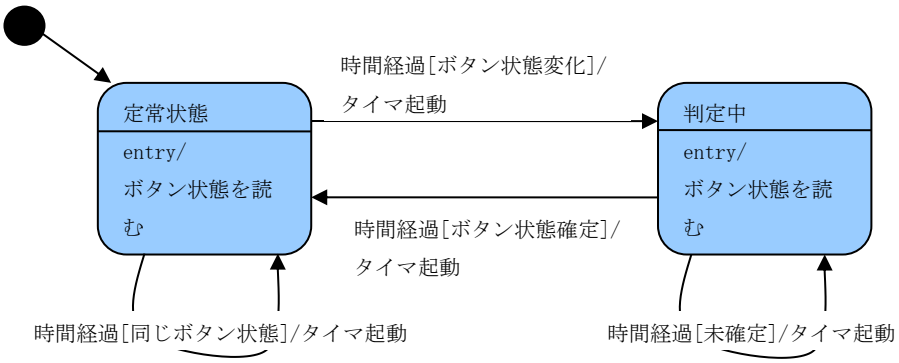
(2) コラボレーション図



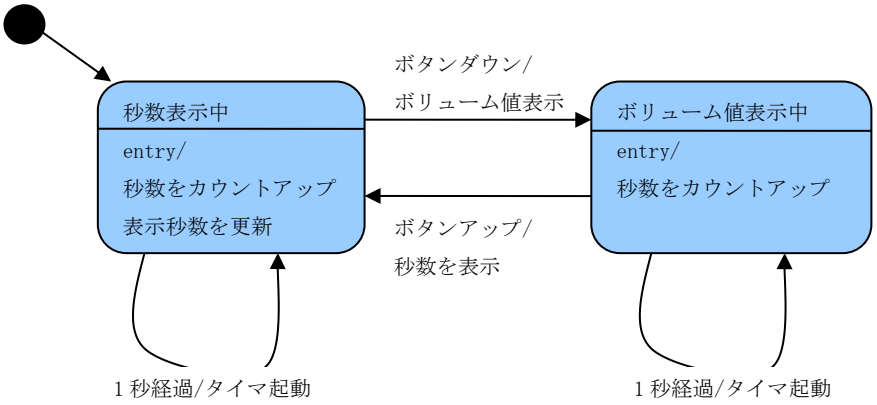
(3) LED 操作の状態図



(4) ボタン処理の状態図



(5) 表示処理の状態図



7.5.3実装設計

タスク構成

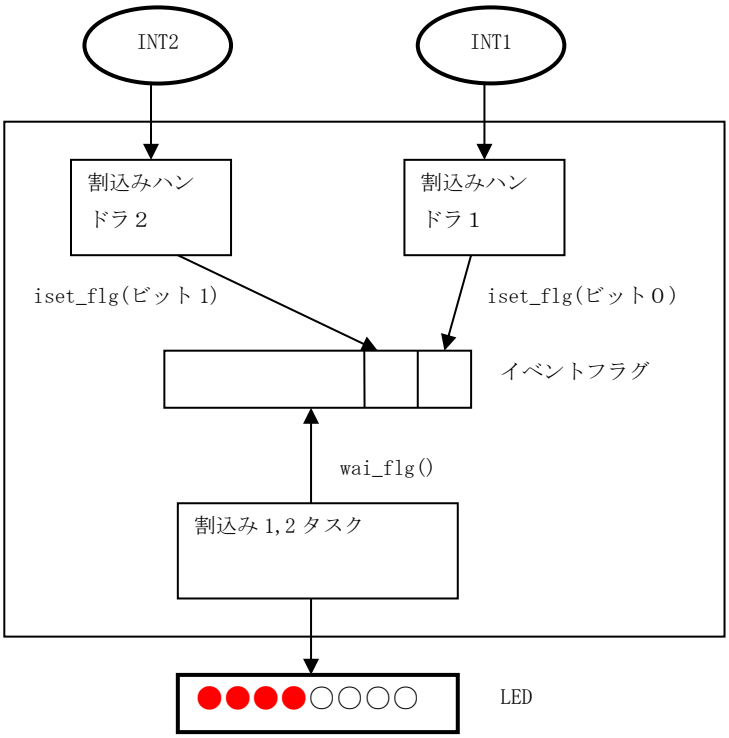
初期化処理	InitTask	LED の初期化（すべて消灯）
タイマタスク	TimerTask	汎用的な時間管理タスク
表示処理	DisplayTask	表示器へ秒数、ボリューム値を表示。
ボタン処理	ButtonTask	ボタンの UP/DOWN のデバンス処理。
デバッグタスク	DebugTask	タスクなどの状態を表示する
LED 操作処理	Irq12Task	イベントフラグ１で待って割込み１、２により LED の点灯状態を変える。

割込みハンドラ構成

INT1 割込み	Int1Handler	イベントフラグ１へ１を設定
INT2 割込み	Int2Handler	イベントフラグ１へ２を設定

解説

INT1 と INT2 の割込みハンドラからイベントフラグ１を使って Irq12Task を起動する。Irq12Task は、イベントフラグ１のセットされているビットを見てどちらの割り込みが発生したかを調べる。両方のビットがセットされている場合も考慮する。



ファイル構成

Step5¥	
Kernel_cfg.c	μITRON コンフィギュレーションファイル
init.c	初期化タスク
timer.c	汎用タイマタスク
timer.h	
event.c	イベント機構
event.h	イベント用メッセージ
display5.c	表示タスク
button5.c	ボタンタスク
irq1_irq2.c	割込み１，２タスク、INT1 割込みハンドラ、INT2 割込みハンドラ
debug.c	シリアルポートを使ってハイパーターミナルと通信するタスク
debug.h	
system_def.h	オブジェクト ID 定義
App5¥	
App.sln	app5.dll を作成する Visual C++ 2005 のソリューションファイル
App.dsw	app5.dll を作成する VisualStudio6.0 のプロジェクトファイル
Makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル

使用サービスコール

tslp_tsk ext_tsk ref_tsk ref_sem ref_flg iset_flg wai_flag ref_mbx snd_msg rcv_msg vchg_ifl vget_ifl ref_mpf

7.6 ステップ６(app6.dll)

7.6.1システム要求仕様

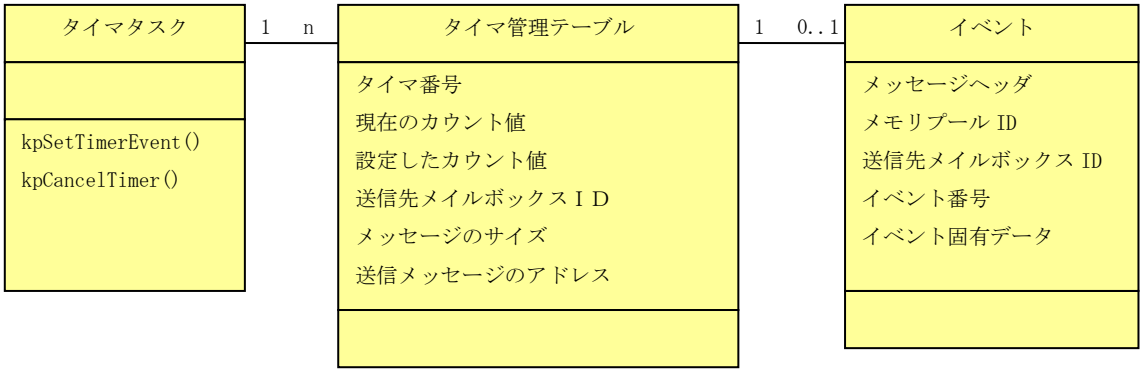
ステップ５のシステムで用いた実装方法は以下の点で見直す必要があります。
- イベントインスタンス領域をユーザプログラムが構造体を静的に定義して確保している。
- イベントの送受信をμITRONのメイルボックスを使ったメッセージの送受信で実装している。
- μITRONのメイルボックスではメッセージ領域のポインタを受け渡す。そのため、メッセージがメイルボックスにある（まだ受信されていない）場合にそのメッセージを間違ってイベント領域として使うと予期しない事態が発生する。

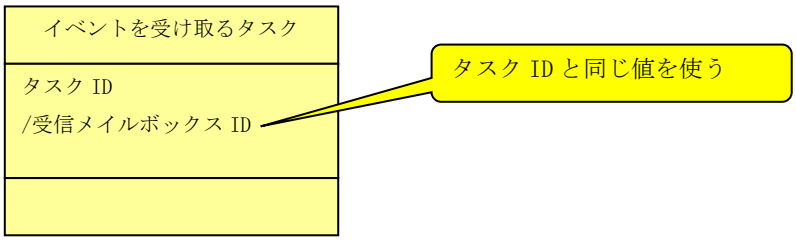
そこで、ステップ６ではイベントインスタンスの領域を固定長メモリプールのメモリブロックを使って確保するように変更します。
- イベントインスタンスの領域確保は、メモリプールからメモリブロック取得に置き換える。
- イベントを送信するタスクがメモリプールからメモリブロックを確保する。
- イベントを受信したタスクは、対応する処理を終了したらメモリブロックをメモリプールに返却する。
- ただし、メモリブロック長はすべてのイベントの中での最大データ長を格納できるメッセージの長さ以上にしておく必要がある。

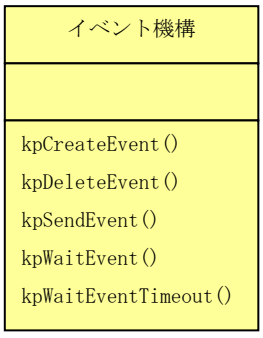
7.6.2システム分析

(1) クラス図

ここでは、タイマタスクの見直しとイベント送信、受信機構の見直しを行います。







7.6.3実装設計

タスク構成とハンドラ構成はステップ５と同じです。
ただし、タスクのイベントを待つメイルボックスＩＤはタスクＩＤと同じ値にします。こうすることによりデバッグしやすくなります。

イベント送信時にイベントインスタンスを作成し、受信側でイベントインスタンスを解放します。このようにすることにより、イベントインスタンスを排他的に使うことを保証します。

（１） イベントの送信は以下のようになります。

```
EVENT* pEvent = kpCreateEvent(イベントＩＤ);//イベントインスタンスの作成
if(pEvent){
        //イベントインスタンスを設定
        pEvent->evMbx = イベントを受け取るメイルボックスＩＤを指定;//
        kpSendEvent(MID_Display, pEvent);//メイルボックス MID_Display へイベント送信
}else{
        //イベントインスタンスが作れない
}
```

（２） タスクはイベントを受信した場合の処理を以下の形式で処理します。

```
void Task()
{
        while(1){
                pEvent = kpWaitEvent(MID_Display);
                if(pEvent){
                        //イベント受信
                        処理
                        kpDeleteEvent(pEvent);//イベントインスタンスの解放
                }else{
                        //イベントなし
                }
        }
}
```

（３） タイマイベントの使い方は以下のようになります。

```
EVENT* pEvent = kpCreateEvent(TIMEOUT_EVENT);    //イベントインスタンスの作成
if(pEvent){
        //イベントインスタンスを設定
        pEvent->evMbx = タイマイベントを受け取るメイルボックスＩＤを指定;//
        kpSetTimerEvent(tno, 1000, pEvent);      //タイマ番号 tno にタイマイベントを設定
}else{
        //イベントインスタンスが作れない
}
```

ファイル構成

```
Step6¥
    Kernel_cfg.c          μITRON コンフィギュレーションファイル
    init.c                初期化タスク
```

timer6.c	汎用タイマタスク
timer6.h	
event6.c	イベント機構
event6.h	イベント用メッセージ
display6.c	表示タスク
button6.c	ボタンタスク
irq1_irq2.c	割込み１，２タスク、INT1 割込みハンドラ、INT2 割込みハンドラ
debug.c	シリアルポートを使ってハイパーターミナルと通信するタスク
debug.h	
system_def.h	オブジェクト ID 定義
App6¥	
App.sln	app6.dll を作成する Visual C++ 2005 のソリューションファイル
App.dsw	app6.dll を作成する VisualStudio6.0 のプロジェクトファイル
Makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル

使用サービスコール

tslp_tsk ext_tsk ref_tsk ref_sem ref_flg iset_flg wai_flag ref_mbx snd_msg rcv_msg vchg_ifl vget_ifl pget_mpf rel_mpf ref_mpf

8　C-Machine のプログラム例

これらのサンプルプログラムは、C-Machine を使うための関数、マクロのプログラミング例です。
通常割込みハンドラ内では最小限の処理で、タスク側で時間のかかる処理を行いますが、これらのサンプルプログラムではそうはなっていません。割込みボタン「INTn」のクリックでテスト用の関数を呼び出すようになっています。
付属のスクリプトファイルはコンソール・コマンドの使用例となります。

8.1 タスクと割込みハンドラの例

8.1.1ファイル構成

実行ファイルは以下のフォルダです。

bin¥sample¥

app.dll	μITRON アプリケーションファイル
cminit.cms	スクリプトファイル

ソースファイルは以下のフォルダです。

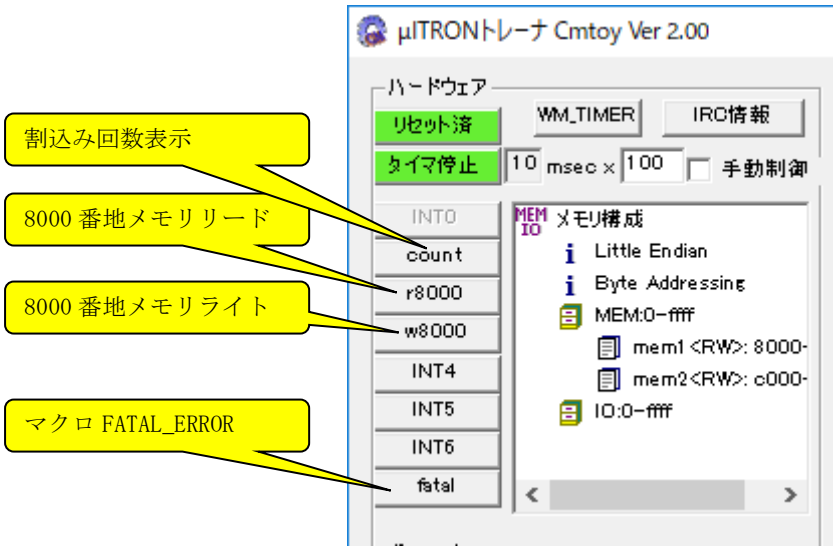
mITRON¥sample¥

app¥	プロジェクトファイルなど VisualStudio が使用するファイル群
kernel_cfg.c	μITRON コンフィグレーションファイル
debug.c	シリアル１へ表示する DebugPrintf 関数など
debug.h	
sample.c	TimerTask, WatchButtonTask, Int1, Int7, InitHandler
test.c	Task1, Task2, Task3, Int2, Int3
test.h	

8.1.2システムの概要

Cmyoy を起動して、「スクリプト」ボタンから cminit.cms を実行すると以下の処理が行われます。
・ターゲットメモリを定義
・割込みボタン「INTn」の表示名を変える
・アプリケーションファイルのロード（「ロード」ボタンに対応）
・カーネルから実行開始（「リセット」ボタンに対応）

Cmtoy の表示は以下のようになります。



ハイパーターミナルで TCP/IP ポート 700 を使って接続すると、DebugPrintf で出力する文字がハイパーターミナルに表示されます。
この後「カーネル情報」ボタンから「カーネルオブジェクト」を表示すると、以下のタスクと割込みハンドラが確認できます。

ここで、Task1(ID=1), Task2(ID=2), Task3(ID=3)の関数は実行後終了しているので、状態は DMT（休止状態）となっています。アイドルタスク（ID=0）の状態は RUN（実行状態）です。タスク TimerTask(ID=4), WatchButtonTask(ID=5)の状態は SLP（起床待ち状態）です。

(1) タスク関数

Task1	Outpu ウィンドウにメッセージを表示して ext_tsk を呼び出さないで終了
Task2	Outpu ウィンドウにメッセージを表示して ext_tsk を呼び出さないで終了
Task3	Outpu ウィンドウにメッセージを表示して ext_tsk を呼び出さないで終了
TimerTask	LED を順次点灯
WatchButtonTask	ボタンの状態を監視

(2) 割込みハンドラ関数

Int1	Int1 が呼び出された回数を表示器に設定
Int2	8000 番地を読む。シリアル１へ表示
Int3	8000 番地へ書く。シリアル１へ表示
Int7	割込みハンドラ内でマクロ FATAL_ERROR を呼び出す

8.1.3使用関数、マクロ

halSerialInit, halSerialWriteChar, halSerialReadChar,
halSetSegLED, halSetLED, halGetPushButton,
halSelectBank, halGetCurrentBank
CMTRACE, FATAL_ERROR,
WORD_PTR, WRITE_BYTE, WRITE_WORD, WRITE_DWORD,
PREAD_BYTE, PREAD_WORD, PREAD_DWORD, PWRITE_BYTE, PWRITE_WORD, PWRITE_DWORD,
POR_BYTE, POR_WORD, POR_DWORD, PXOR_BYTE, PXOR_WORD, PXOR_DWORD,
PAND_BYTE, PAND_WORD, PAND_DWORD, PXCHG_BYTE, PXCHG_WORD, PXCHG_DWORD,

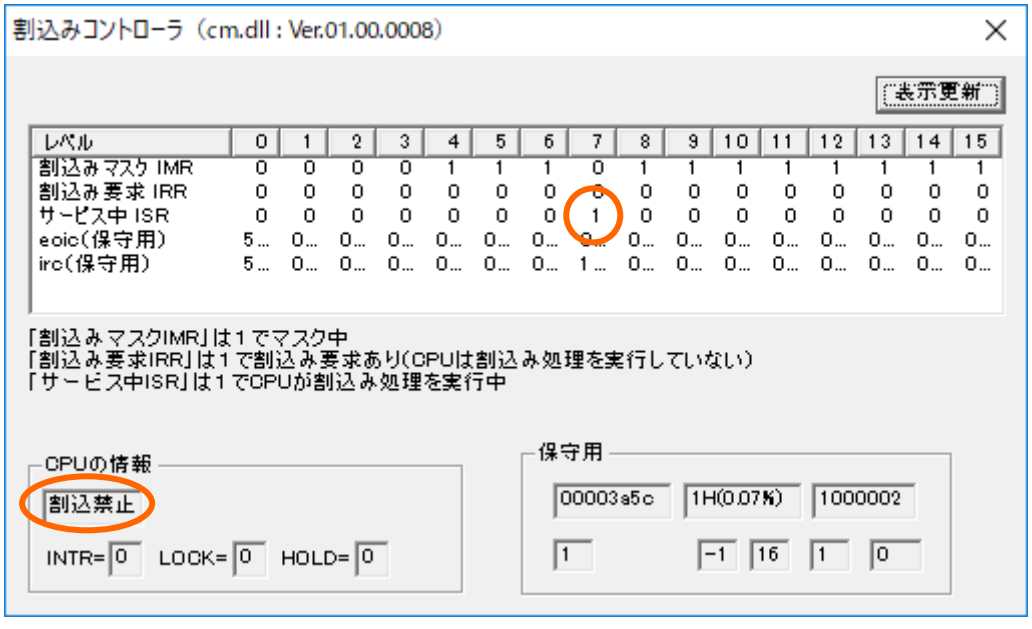
INT7 に対応する「fatal」ボタンをクリックすると、割込みハンドラ内でマクロ FATAL_ERROR を呼び出します。

このマクロを実行すると、割込み禁止にし、タイマを停止してエラーメッセージを出力ウインドウに表示し、メッセージボックスを表示を表示して、無限ループに入ります。出力ウインドウには以下のエラーメッセージが表示されます。メッセージには FATAL_ERROR を呼び出したソースファイル名と行番号が含まれます。

```
>Int 7 #1
app: handler context.
;Timer Stopped.
▲▲致命的エラー：app: Int7 invoked.
 <D:¥cmtoy-200¥mITRON¥sample¥sample.c(171)>
```

このように割込みハンドラ内で無限ループに入っている状態で「カーネルオブジェクト」の表示を更新すると以下のようになっています。

このとき「IRC 情報」ボタンをクリックし、IRC(割込みコントローラ)の状態を確認すると以下のようになっています。（CPU は割込み禁止、IRC の ISR レベル 7 はサービス中）



8.2 ターゲットメモリの例

8.2.1ファイル構成

実行ファイルは以下のフォルダです。

bin¥16550¥

app.dll	μITRON アプリケーションファイル
cminitBB.txt	バイトアドレッシング、ビッグエンディアン用スクリプトファイル
cminitWB.txt	ワードアドレッシング、ビッグエンディアン用スクリプトファイル
cminitLB.txt	バイトアドレッシング、リトルエンディアン用スクリプトファイル
cminitWB.txt	ワードアドレッシング、リトルエンディアン用スクリプトファイル
font_han.bin	フォント定義ファイル
font_han2.bin	フォント定義ファイル（font_han.bin と同じ）

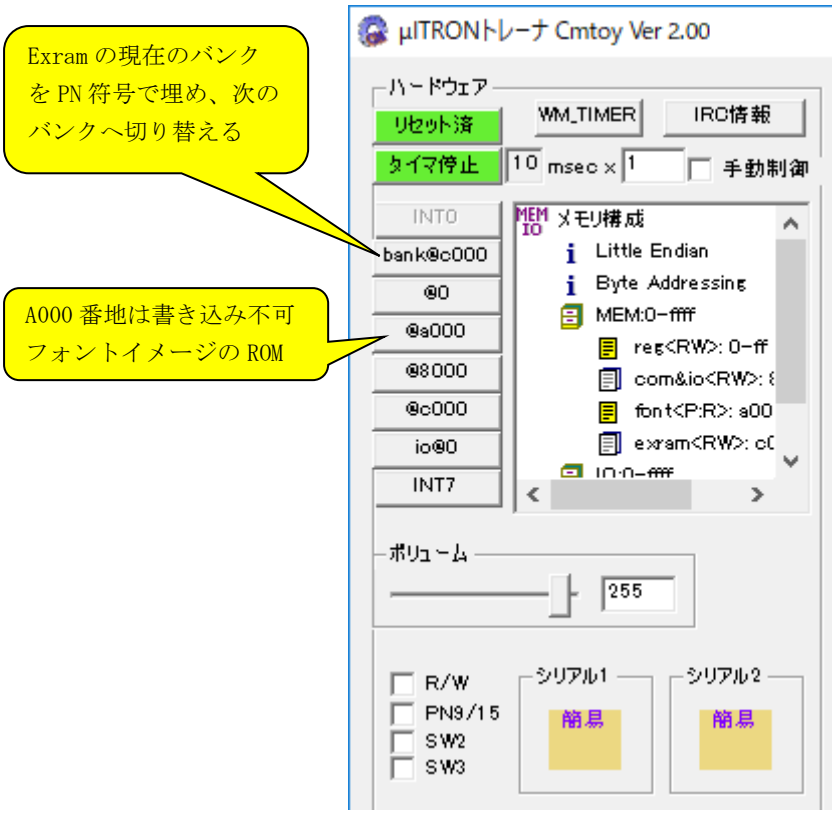
ソースファイルは以下のフォルダです。

mITRON¥sample_memory¥

app¥	プロジェクトファイルなど VisualStudio が使用するファイル群
kernel_cfg.c	μITRON コンフィグレーションファイル
debug.c	デバグタスク、シリアル１へ表示する DebugPrintf 関数など
debug.h	
sample.c	TimerTask, WatchButtonTask, InitTask
test.c	Int1, Int2, Int3, Int4, Int5, Int6, SetPnCode
test.h	
font.c	デバグタスクのコマンドを拡張する ExecCustomCommandLine
font.h	

8.2.1システムの概要

スクリプト cminitLB.txt を実行するとターゲットメモリをバイトアドレッシング、リトルエンディアンで作成します。その後 app.dll をロードし、実行を開始します。GUI は以下のようになります。



この後「カーネル情報」ボタンから「カーネルオブジェクト」を表示すると、以下のタスクと割込みハンドラが確認できます。

(1) タスク関数

DebugTask　　デバグタスク
TimerTask　　LED を順次点灯、SW1(PN9/15)を監視
WatchButtonTask　ボタンの状態を監視

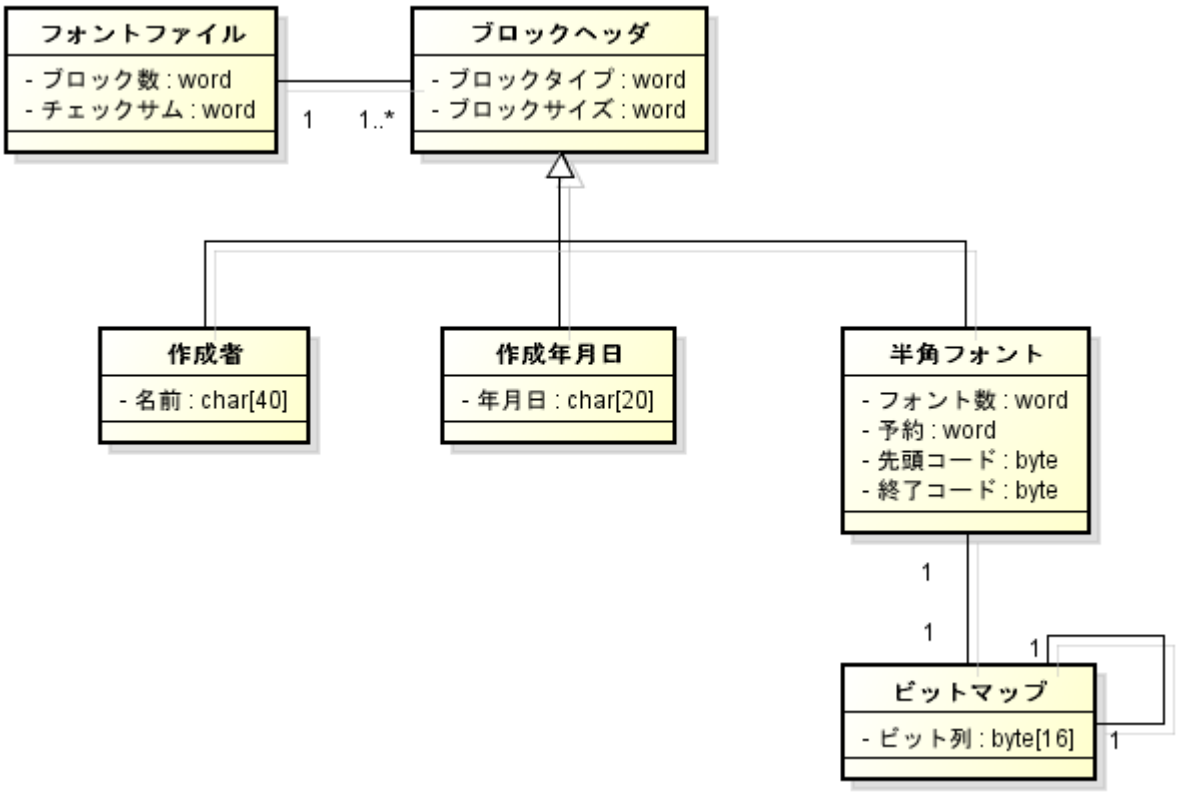
(2) 割込みハンドラ関数

Int1　　Exram の現在のバンクを PN 符号で埋め、次のバンクへ切り替える
Int2　　0000 番地を読む、または書き込む

Int3　　a000 番地を読む、または書き込むと書き込み不可のエラー
Int4　　8000 番地を読む、または書き込む
Int5　　c000 番地を読む、または書き込む
Int6　　IO の 0000 番地を読む、または書き込む

(3) フォントファイル

font_han.bin, font_han2.bin は半角文字のフォントイメージ（8x16 ビットマップ形式）を格納しています。その構造は以下のとおりです。



font_han.bin, font_han2.bin は Windows のアプリケーションで作成したのでリトルエンディアン形式です。実際のデータは以下のようになっています。

ファイル内オフセット	データ	項目名	項目の値（リトルエンディアン）
0000	04	ブロック数	0004
0001	00		
0002	0c	ブロック１オフセット	000c
0003	00		
0004	38	ブロック２オフセット	0038
0005	00		
0006	50	ブロック３オフセット	0050
0007	00		
0008	5a	ブロック４オフセット	065a
0009	06		
000a	f4	チェックサム	00f4
000b	00		
000c	ff	ブロック１タイプ	ffff（作成者）
000d	ff		

チェックサムを計算する範囲

・ ・ ・			
0038	fe	ブロック２ タイプ	fffe（作成年月日）
0039	ff		
・ ・ ・			
0050	02	ブロック３ タイプ	0002（半角フォント）
0051	00		
・ ・ ・			
065a	02	ブロック４ タイプ	0002（半角フォント）
065b	00		
・ ・ ・			

ターゲットメモリの領域（a000 番地）をこのファイルのデータを使って永続的で読み取り専用としています。この場合は a000 からのオフセットとして扱います。

(4) デバグコマンドの拡張

デバグタスクのコマンドライン解析で先頭が'.'（ピリオド）の場合は、それ以降の文字列を関数 pfnCustomCommandProc に渡します。pfnCustomCommandProc は、DebugSetCustomCommandProc を使って事前に登録しておきます。
このサンプルプログラムでは、μITRON の初期化ハンドラ InitHandler で font.c 内の関数 ExecCustomCommandLine を登録しています。
この拡張コマンド解析 ExecCustomCommandLine では以下の２つのコマンドを追加します。

・フォントを探す
　　シンタックス　　ff 〈addr〉
　　パラメータ
　　　　〈addr〉　フォントを探すターゲットメモリのアドレス
　　説明　　〈addr〉の内容をフォントファイルのフォーマットに従いチェックサムを調べ一致していれば正しいフォントと判断して、フォントのベースを〈addr〉に設定し、ブロック情報を表示する。

・フォントの表示
　　シンタックス　　vf 〈code〉
　　パラメータ
　　　　〈code〉　表示する文字コード
　　説明　　フォントのベースから〈code〉に対応する半角フォントのビットパターンを探し表示する。

ハイパーターミナルの画面表示は以下のようになります。

```
CLI> .ff a000
Font Blocks.
 0 a00cH : "Custom font file generator (V2.00)
                                                "
```

```
  1 a038H : "16:38:06 02/26/19"
  2 a050H : hankaku 20H - 5fH
  3 a65aH : hankaku a0H - dfH。
OK
CLI> .vf 31
Font(31) image @a16a.
00000000
00001000
00111000
00001000
00001000
00001000
00001000
00001000
00001000
00001000
00001000
00001000
00001000
00001000
00000000
00000000
OK
```

8.2.2使用関数、マクロ

halSerialInit, halSerialWriteChar, halSerialReadChar,
halSetSegLED, halSetLED, halGetPushButton, halGetSwitch, halGetVolume
halSelectBank, halGetCurrentBank
halCalcPN15, halCalcPN9
CMTRACE,
READ_BYTE, READ_WORD, READ_DWORD, IN_BYTE, IN_WORD, IN_DWORD, OUT_BYTE

8.3 16550 制御例

8.3.1ファイル構成

実行ファイルは以下のフォルダです。

bin¥16550¥

app.dll	μITRON アプリケーションファイル
16550conf.txt	スクリプトファイル

ソースファイルは以下のフォルダです。

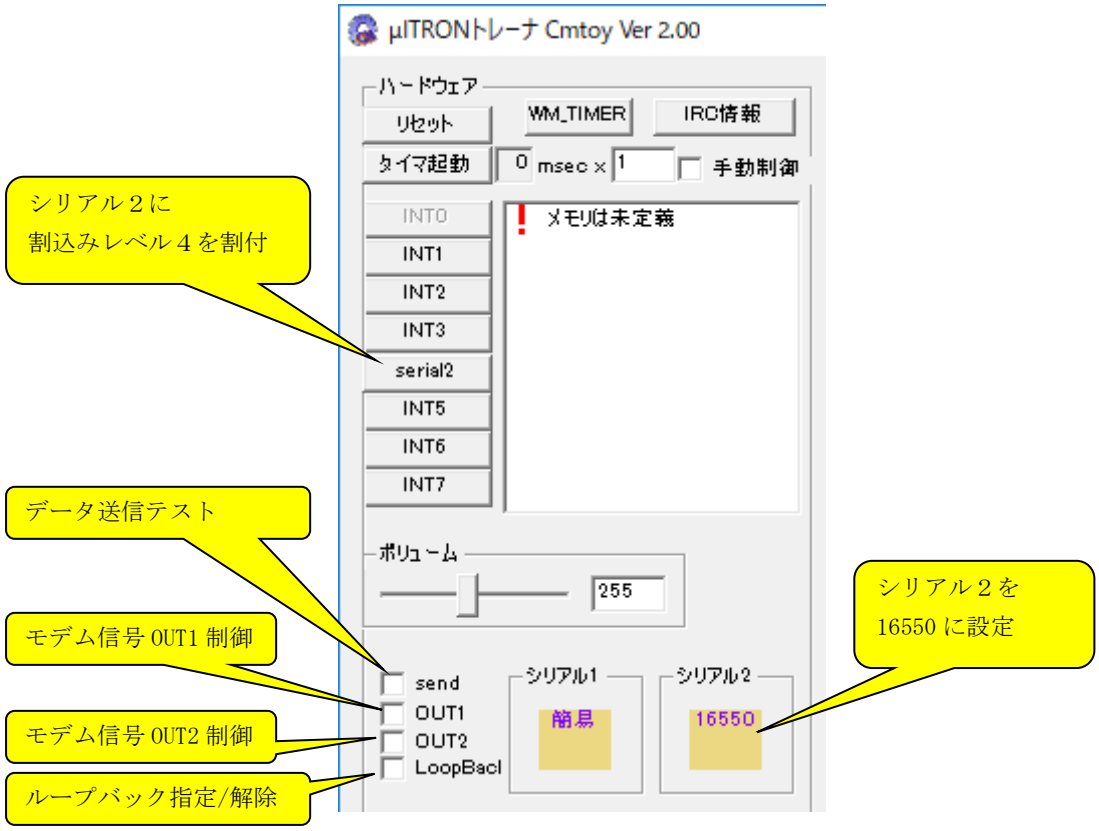
mITRON¥sample16550¥

app¥	プロジェクトファイルなど VisualStudio が使用するファイル群
kernel_cfg.c	μITRON コンフィグレーションファイル
debug.c	DebugTask、シリアル１へ表示する DebugPrintf 関数など
debug.h	
sample.c	TimerTask, WatchButtonTask, InitHandler
test16550.c	Init16550, IntHdr16550, Task16550

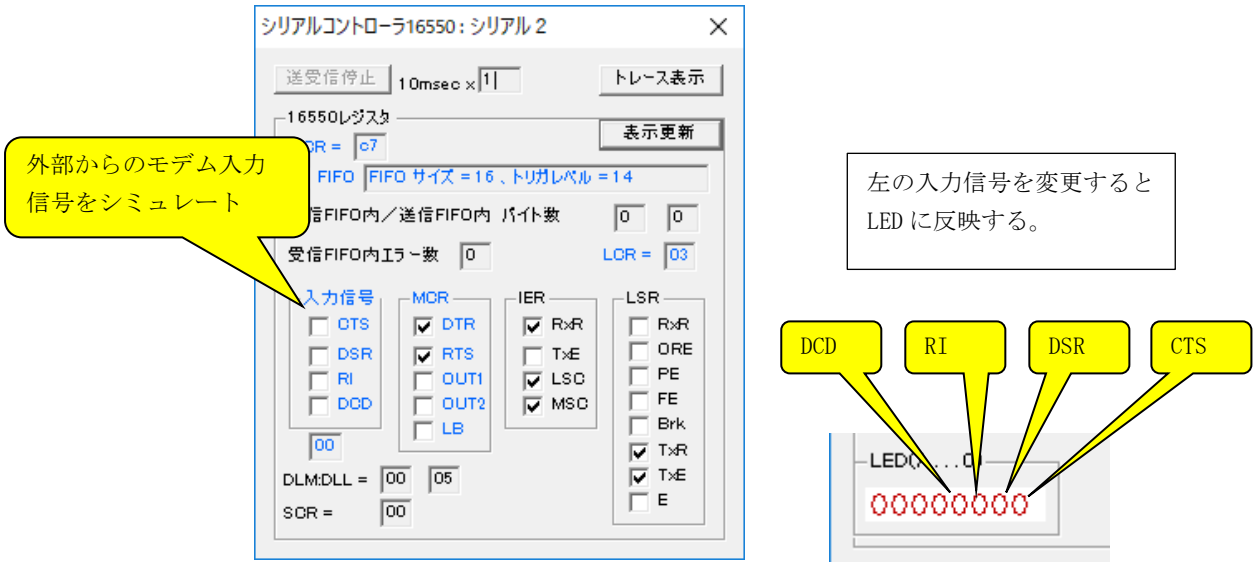
test.h

8.3.2システムの概要

スクリプト 16550conf.txt を実行すると GUI は以下のようになります。



ここでアプリケーションを実行するとシリアル２の 16550 が初期化されます。（「5.8 16550 相当のシリアル制御関数」を参照）



ここでハイパーターミナルをシリアル１、２へ TCP/IP ポート７００、７０１を使って接続します。この後「カーネル情報」ボタンから「カーネルオブジェクト」を表示すると、以下のタスクと割込

みハンドラが確認できます。

(1) タスク関数

WatchButtonTask ボタンの状態を監視、ボタンを押し下げてる間 BREAK 信号を ON
Task16550 シリアル２を初期化、SW0 の状態が変化したらシリアル２へ送信

(2) 割込みハンドラ関数

IntHdr16550 シリアル２からの割込みをハンドリング、モデム入力信号を LED に反映

8.3.1使用関数、マクロ

halSerialInit, halSerialWriteChar, halSerialReadChar,
halSetSegLED, halSetLED, halGetPushButton, halGetSwitch, halGetVolume
Init16550
CMTRACE,
READ16550, WRITE16550
WRITE_LCR, WRITE_IER, WRITE_LCR, WRITE_MCR, WRITE_FCR, LSTAT16550, MSTAT16550
SET_OUT116550, CLEAR_OUT116550, SET_OUT216550, CLEAR_OUT216550,
SET_LOOPBACK16550, CLEAR_LOOPBACK16550
ENA_TX16550, IID16550, DIS_TX16550

9 考察

9.1 VisualStudio6.0 のデバッガ

デバッグは、まず app.dsw を Visual Studio 6.0 で開いて「アクティブな構成の設定」を「Win32 Debug」に変更して、デバッグバージョンでリビルドします。デバッグバージョンでは Dapp.dll というファイル名でサンプルアプリケーションが作られます。次に「ビルド」メニュの「デバグの開始」→「実行」をすると Cmtoy が動き図 1-2 のウインドウが現われます。ここでアプリケーションタスク、割込みハンドラのソースコードを開きブレークポイントを設定することもできます。
もうひとつの方法は、デバッグバージョンの Dapp.dll を作成した後ワークスペースを閉じてから Cmtoy を起動して Dapp.dll をロードします。ここで Visual Studio 6.0 で「ビルド」メニュの「デバグの開始」→「プロセスへアタッチ」で Cmtoy プロセスにアタッチするとデバッグ可能となります。アタッチしただけではソースファイルは開かれていないので、「ファイル」メニュの「開く」からデバッグしたいソースファイルを開いてブレークポイントを設定し、Cmtoy のウインドウに戻り「リセット」で実行を開始します。
現状のシミュレーションでは、μITRON タスクを Win32 のスレッドに割り当てる方法をとっています。この方法で一通り μITRON のスケジューリングをシミュレートできそうなんですが、
VisualStudio6.0 のデバッガを Cmtoy にアタッチするとスレッドの動きがこちらで想定しているようにならないときがあります。（当然 Cmtoy にバグがある可能性もありますが）
あてずっぽうですが、
・VisualStudio6.0 のデバッガは、プロセス内のユーザ空間内のプログラムのみをデバッグ可。
・Windows のスレッドスケジューリングはカーネル内で行っているようです。
を前提にすると、ユーザプログラムをブレークポイントで停止させてもカーネルや他のプロセスは動いているのでスレッドの動きが思ったようにならないのかもしれません。特にタイマイベントが溜まっているようで、ブレークポイントを解除したときにタイマイベントが連続して発生 するように見えます。SoftICE のようなカーネルも停止させるデバッガを使えばいいのかもしれません。(2002 年)

9.2 Regsvr32

これは、ActiveX コントロールを登録するマイクロソフトの再頒布可能なユーティリティです。これで一度登録した ActiveX コントロールの登録解除もできます。通常 Windows の SYSTEM ディレクトリに含まれています。既定では、Windows 98/ME では C:¥WINDOWS¥SYSTEM に、Windows 2000 では C:¥WINNT¥SYSTEM32 に、Windows XP では C:¥WINDOWS¥SYSTEM32 に含まれています。Windows 98/ME 付属の REGSVE32.EXE は、Windows 2000/XP では使えるようですが、Windows 2000/XP 付属の REGSVE32.EXE は、Windows 98/ME では機能しないようです。(2002 年)

9.3 Borland C++ 5.5.1

DLL からイクスポートする関数名が Visual C++と Borland C++で違うことが分かりました。
__declspec(dllexport)で宣言した関数名は Visual C++では先頭に_（アンダースコア）が付かないのに、Borland C++では先頭に_（アンダースコア）が付いた関数名になります。Cmtoy では、アプリケーションモジュールをロードしてからイクスポートされた関数エントリポイントを関数名で探しますが、その手順を以下のようにしました。

①Visual C++の関数名で探す。
②見つからないと Borland C++の関数名で探す。

どちらでも見つからないときはエラーとなります。(2002 年)

9.4 UML について

サンプルプログラムの説明に UML を使ってみました。まだ自己流に UML 記述を使っている部分も含まれています。しかし、ユースケース図で何をやろうとしているプログラムかがわかりやすくなったような気がします。
使ったことのある UML ツールは BridgePoint、mc3020 と Rose Realtime ですが、UML は組込みシステムの開発にも有効だと感じました。これらのツールは UML のクラス図、状態図、アクションから C/C++/java のソースコードを生成します。そのため XT (Executable and Translatable) UML ともいうようです。
20 年くらい前から組込みシステムの各タスクをイベント（メッセージ）駆動型にして動作を状態遷移表（有限状態マシン）として設計していたので、システムを UML ステートチャートで設計することにそれほど違和感はありませんでした。それより有限状態マシンを実装する場合に必要となるイベント定義、イベント送受信/イベントキュー/イベントディスパッチのメカニズム、タスクへのマッピングを自前で実装する必要がないのでシステム要求仕様の分析、実装設計に専念できます。また、BridgePoint ではオブジェクトコラボレーション図などを自動生成してくれるので、設計の妥当性を確認する手助けになります。また、原則的に同じ手法で設計した UML モデルから OS を指定してコード生成をするので、設計段階で OS の違いをあまり意識しなくすみそうです。また OS を使わない指定をするとそれなりの動作するコードを生成してくれるので、OS を使う場合も使わない場合も同じように設計、実装できます。
C が CPU に依存しない組込みシステム開発をある程度可能にしたように、XTUML は OS/プラットフォームに依存しない組込みシステム開発を可能にできるかもしれないと感じました。

※ BridgePoint は Project Technology 社の製品です。
※ mc3020 は ROX Software 社の製品です。日本語対応は（株）東陽テクニカが行っています。
※ Rose Realtime は、Rational Software 社の製品です。
(2002 年)

9.5 Visual C++ 2008 Express Edition

無料で使用できる Visual C++ 2008 Express Edition SP1 で Cmtoy のサンプルプログラムを再コンパイル（リビルド）して、動作を確認しました。Visual C++ 2008 Express Edition には SDK が同梱されているので SDK を別途インストールする必要はないようです。Visual C++ 2008 Express Edition については、以下を参照してください。

http://www.microsoft.com/japan/msdn/vstudio/express/
(2008 年)

9.6 Windows XP 以前の OS へのインストール

Windows XP 以前の OS（例えば Windows 2000 Professional など）で Cmtoy が起動できない場合、led.ocx が install.bat で登録できないことが考えられます。これは、install.bat を実行した DOS プロンプトで以下のコマンドを実行することで確認できます。

```
regsvr32 -c led.ocx
```

ここで、以下のエラーを通知するダイアログボックスが表示されます。

このような場合は、マイクロソフトの再頒布可能な DLL である MSVCP60.DLL をマイクロソフトのサイト
http://support.microsoft.com/kb/259403/ja

から取得するか、MSVCP60.LZH をホームページからダウンロードして解凍し、MSVCP60.DLL を Cmtoy¥bin へコピーしてください。
その後、再度 install.bat を実行してください。(2009 年)

9.7 Windows Vista、Windows 7 で使用する場合

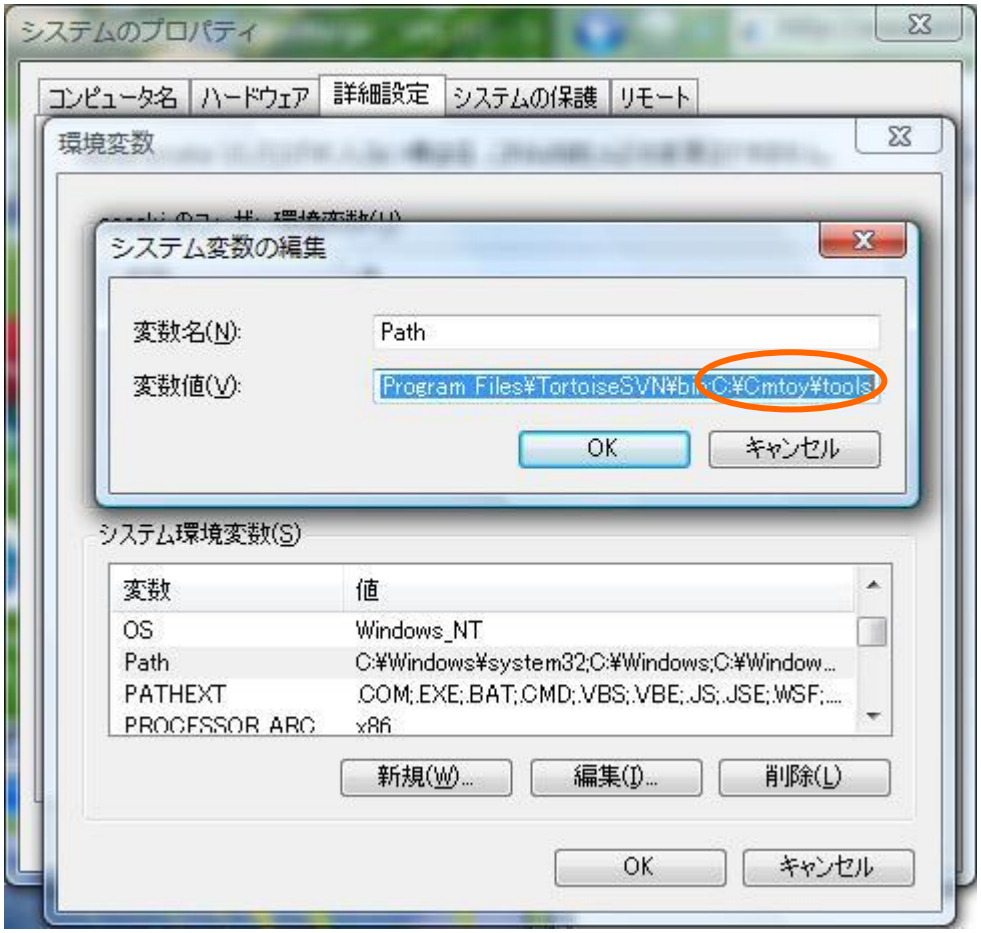
9.7.1ハイパーターミナル

Vista および Windows 7 ではハイパーターミナルが搭載されていないので以下のページを参照して（日本語ページは機械翻訳のようで読みにくい）

http://www.windowsvistaplace.com/hyperterminal-alternative-in-windows-vista/downloads/ja/

http://www.windowsvistaplace.com/hyperterminal-alternative-in-windows-vista/downloads/

Windowws XP から hypertrm.dll と hypertrm.exe を取り出して Vista にコピーしました。
具体的には、
１．hypertrm.dll と hypertrm.exe を C:¥Cmtoy¥tools フォルダにコピーする。
２．環境変数の Path に C:¥Cmtoy¥tools を追加する。（コントロールパネルのシステムを開き、「システムの詳細設定」を選び「環境変数」から登録できる）



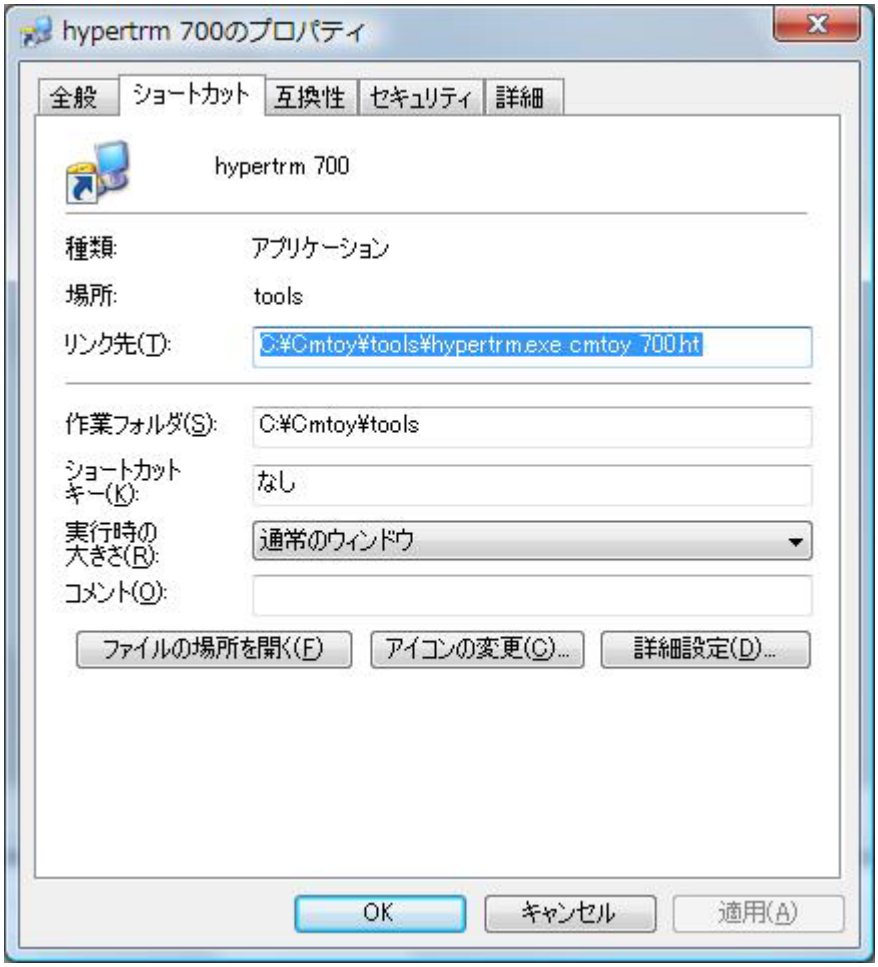
３．hypertrm.exe のショートカットを C:¥Cmtoy¥tools フォルダにつくり、ファイル名を hypertrm700 とする。このショートカットのプロパティを開き、リンク先に

C:¥Cmtoy¥tools¥hypertrm.exe cmtoy 700.ht

と入力する。同様にシュートカット hypertrm701 を作成し、リンク先を

C:¥Cmtoy¥tools¥hypertrm.exe cmtoy 701.ht

とする。このショートカットからハイパーターミナルを起動する。



(2009 年)

9.7.2コマンドプロンプト

Vista および Windows 7 ではセキュリティ機能が強化され、管理者モードでのコマンドプロンプトでないと Regsvr32 を実行できません。管理者モードでのコマンドプロンプトを実行する方法の例は、「3.1.1 Windows Vista,Windows 7 でのインストール」を参照してください。(2009 年)

9.8 Windows10 で Visual Studio 6.0 を使う方法

Windows 10 で VisualStudio6.0 を使う方法が CodeOroject の以下のページで紹介されていました。
Install Visual Studio 6.0 on Windows 10

経験的には VisualStudio6.0 で MFC と統合開発環境はほぼ完成したと感じていました。Cmtoy の開発には VisualStudio6.0 を使っていたので、Windows10 で VisualStudio6.0 が使えるのはありがたい。私にとっては使い慣れているのと起動時間も短いので使いやすい。

V2.00　2019 年 4 月 1 日

9.9 システム初期化手順

9.9.1実機での初期化手順

まず、実際のコンピュータシステムでの電源 ON、またはハードウェアリセットの場合の動作を考察

しておきましょう。ここではパソコン（Windows マシン）の初期化手順を考察します。パソコンでは CPU として x86 が使われています。この x86 アーキテクチャ CPU では電源 ON、またはハードウェアリセット直後は 8086（リアルモード）として動き、以下のようになるようです。

1. CS:IP が ffff:0 に初期化される。割込み禁止状態。
   アドレスの高位 f000:0-f000:ffff は ROM（システム BIOS）。ffff:0 には JMP 命令がありシステム BIOS の開始ルーチンへ JMP する。
2. BIOS の開始ルーチンでは、POST(Power On Self Test)を行い、周辺装置を初期化。ベクタテーブル(0:0-0:3ff)を設定。BIOS はどの外部記憶装置からシステムを起動するかを知っている。
3. 外部記憶装置（FD,ディスクなど）の MBR(Master Boot Record：シリンダ０、ヘッド０、セクタ１）の内容（512 バイト）をメモリ 0:7c00 へ読み出して、開始ルーチンへ JMP する。MBR は実行コード(boot code)と 4 個のパーティションテーブル(partition table)を含んでいる。パーティションテーブルは各パーティションの外部記憶装置上の位置とサイズを定義している。
4. マスタブートはパーティションテーブルの中から起動フラグのあるパーティションを探し、その先頭セクター(boot sector または volume boot record)をメモリ上にロードしてその先頭ルーチンへ JMP する。
5. Boot sector のコードはそのパーティションのファイルシステムを知っているのでファイル名でセカンドブートをメモリへロードして、その先頭ルーチンへ JMP する。
   セカンドブートには BOOTMGR、NTLDR、SYSLINUX、GRUB などがある。おそらくここで 32 ビットモード／64 ビットモードへ移行する。
6. セカンドブートが OS をメモリ上へロードする。OS（Windows）の初期化が始まる。

このように電源 ON 後最初に実行するプログラムは ROM（Read Only Memory）に格納しておく必要があります（ROM 内のデータは電源を OFF しても消えない）。

9.9.2 C 言語の処理系での初期化

C コンパイラは、ソースコードから命令部分、データ部分などを抽出してそれらをセクションというまとまりとしてオブジェクトコードを生成します。C コンパイラが生成する主要なセクションを以下に挙げます。セクションは連続したメモリ領域となります。

セクション名	説明
.text	CPU が実行する命令コードの集まり
.data	初期値を持つ変数領域
.bss	初期値を持たない変数領域。０で初期化される。
.rodata	書き換える必要のない初期値を持つ変数領域。const 宣言された変数。

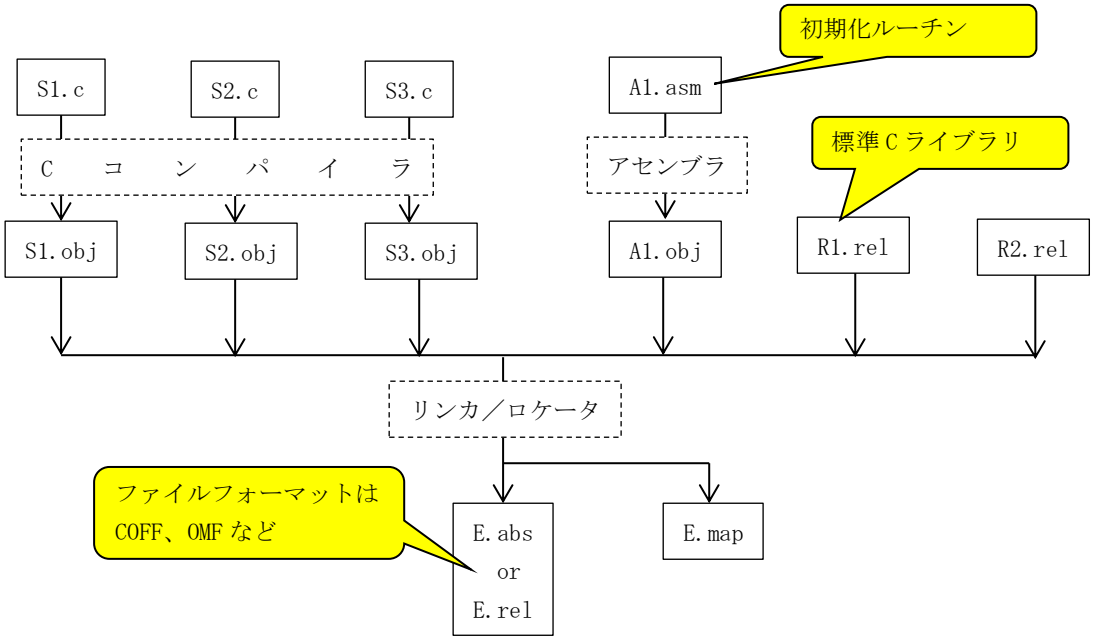
C 言語の処理系は、メモリ上にこれらのセクション配置した後、.bss 領域を０でクリアして SP（スタックポインタ）を適切な値に設定し、main 関数を呼び出します。当然これらのセクション以外にスタック領域を確保しておく必要があります。

セクション.text と.rodata は書き換える必要がないので ROM 内に配置することもできますが、セクション.data と.bss はプログラムが書き換えることができる RAM 領域へ配置する必要があります。

これら以外に C コンパイラが独自のセクションを作る場合もありますし、プログラマが独自のセクションを定義(#pragma section)して特定のコード、データ領域をそこに集めることもできます。

C コンパイラは分割コンパイルが前提になっているので、各ソースファイルからそれぞれ上記のセクションを含んだオブジェクトファイルを作ります。これらの複数のオブジェクトファイルを集めて、同じ名前のセクションを１つの連続領域に結合して１つのオブジェクトファイルを作る必要が

あります。このとき別名なセクションを１つのセクションにまとめ、別な名前を付けることもできます。これを行うツールは一般にリンカとかロケータと呼ばれます。
リンカ／ロケータでは結合したセクションの配置アドレスを指定した実行可能形式のオブジェクトファイルまたは配置アドレスを指定しないでセクションを結合しただけの再配置可能（relocatable）オブジェクトファイルを作ります。
以上の様子を以下に図で示します。



・実行可能形式オブジェクトファイル E.abs には特定アドレスに配置されたセクション情報が含まれていて、すべての extern シンボルのアドレスが解決されています。
・C 言語プログラムが動く前の初期化ルーチンはアセンブラで記述します。この初期化ルーチンはエントリポイントとしてリンカ／ロケータで指定します。初期化ルーチンは bss セクションを 0x00 でクリアし、スタック領域を確保し、main 関数のパラメータを用意して最後に C の main 関数を呼び出します。
・標準 C ライブラリは再配置可能オブジェクトファイルとして C コンパイラに付属しています（ファイル名はコンパイラメーカが決めた名前）。
・再配置可能オブジェクトファイル E.rel は再度リンカ／ロケータの入力ファイルに使えます。
・オブジェクトファイルの形式として COFF（Common Object File Format）や OMF（Object Module Format）などがあります。
・マップファイル E.map は各セクションの先頭アドレスとサイズ、関数や変数の実際のアドレスなどがわかるテキストファイルです。これによりメモリの使用状況や空き領域の確認ができます。

9.9.3プログラムをメモリへ配置する

(1) Windows システムでのメモリへの配置方法

Windows のユーザアプリケーション（*.exe や*.dll）のファイル形式は COFF をベースにした PE（Portable Executable）で、再配置可能形式です。Windows のアプリケーションローダがプロセスを用意してそのアドレス空間に配置します。アプリケーションローダの主要な機能を担うサービスコール CreateProcess は 1 個のプロセス（Process）と 1 個のスレッド（Thread）を作成し、外部記憶装置から*.exe を読み出し PE を解読しながらメモリ上に配置して、依存する*.dll があればそれもメモリ上に配置してシンボルの解決を行います。

プロセスはCPUのアドレス空間をそれぞれのアプリケーションに提供する仕組みです。もう少し正確に言うなら、32ビットアドレス空間のうち後半（0x80000000-0xffffffff）にはWindowsカーネルとデバイスドライバが配置されていて、ユーザアプリケーションは前半(0x00000000-0x7fffffff)に配置されます。各アプリケーションプログラムはCPUのアドレス空間(0x00000000-0x7fffffff)を占有できます。

※ここでのアドレスはプログラムが使用する論理アドレスです。論理アドレスはCPUのセグメンテーション変換、ページ変換をへて物理アドレスに変換されてアドレスバスへ出力されます。
※メモリへ配置後、スタックを確保してエントリポイント(mainCRTStartup)からスレッドは実行を開始します。mainCRTStartupからmain関数を呼び出します。
※Microsoft C/C++で開発されるWindowsアプリケーションの初期化ルーチン(mainCRTStartup)はランタイムライブラリ(LIBC.LIB)に含まれているのでユーザアプリケーションはランタイムライブラリとリンクする必要があります。

(2) μITRONシステムでのメモリへの配置方法

Windowsは汎用コンピュータで様々はアプリケーションに対応できることが必須ですが、μITRONシステムでは特定の用途に対応するためにハードウェア、ソフトウェアが用意されるのが一般的です。アドレス空間は物理アドレスと論理アドレスが一致し、この単一のアドレス空間に起動時のブートプログラム、μITRONカーネル、μITRONアプリケーションをすべて配置します。そのためアドレス空間を用途別に事前に割り当てておく必要があります。
そこでメモリの用途別に領域を以下のようにリストアップしましょう。

領域	ROM/RAM	説明
ブートプログラム用コード	ROM	固定アドレス（CPUのスタートアドレスを含む領域）
ブートプログラム用 データ、スタック	RAM	カーネルに制御が移ると使用されない
割込みベクターテーブル		固定アドレス、固定サイズ（CPUに依存）
μITRONカーネルコード	RAM	実行コード、定数領域
μITRONカーネル データ、スタック	RAM	カーネル初期化時のスタック カーネルオブジェクト領域 μITRONタスク用スタック領域
μITRONコンフィグレーション テーブル（注１）	RAM	固定アドレス、μITRONアプリケーションに含まれる。
μITRONアプリケーション コード	RAM	実行コード、定数領域（.textや.rodataセクションなど）
μITRONアプリケーション データ	RAM	変数領域（.dataや.bssセクションなど）

注１：kernal_cfg.cで定義されるデータ領域。このファイルをコンパイルすると.dataセクションとして作成されるので、この部分を別な名前のセクションとしてオブジェクトを作成するにはC言語のソースファイルの先頭で以下のような宣言（コンパイラに依存）が必要です。

```
#pragma section("config_data", read)
```

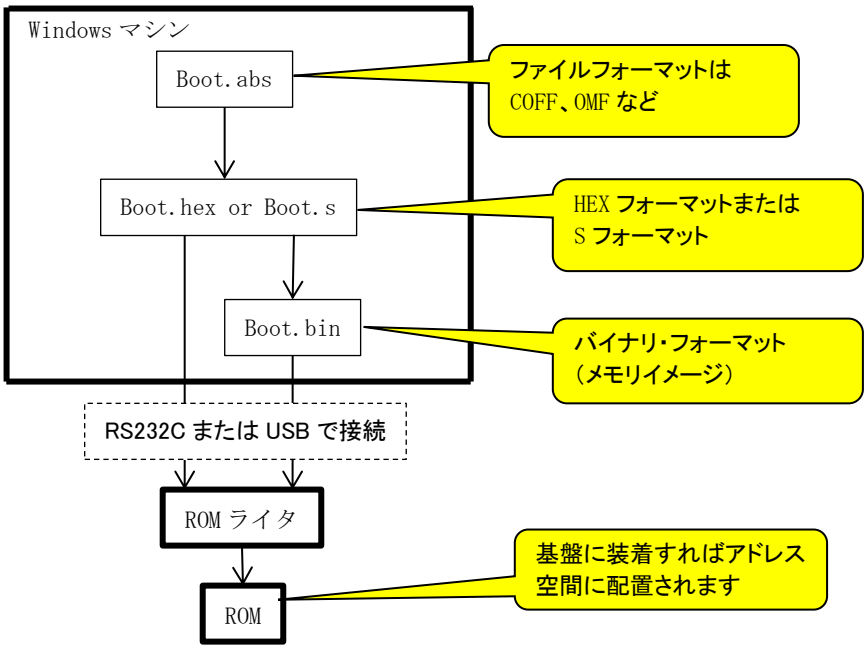
μITRONアプリケーションをリンカ／ロケータで配置するときにこのセクションconfig_dataを特定のアドレスに配置します。このアドレスはμITRONカーネルとの間で決めておく置く必要があり、μITRONカーネル、μITRONアプリケーションに修正があっても変更する必要がないように決めておきます。

ここでは、電源 ON またはハードウェアリセット時にブートプログラムが外部記憶からμITRON カーネルとμITRON アプリケーションを RAM に配置する前提で考察します。このようにするとμITRON カーネルとμITRON アプリケーションに修正が発生した場合に修正が容易になるためです。

以上を踏まえ、ブートプログラム(boot.abs)、μITRON カーネル(kernsl.abs)、μITRON アプリケーション(app.abs)はおのおの配置済みの実行可能形式として作成します。

(a) ブートプログラム boot.abs

ブートプログラムのコードや定数部は ROM として作成するので以下のような手順となります。



ROM に格納されるブートプログラムには以下の機能があればよさそうです。

・外部記憶からのμITRON カーネルとμITRON アプリケーションのメモリ上へのロード
・ハードウェアの自己診断
・外部記憶装置がフラッシュメモリなどで取り外しできない実装とされるなら、RS232C、ネットワークなどを使って外部からダウンロードしてμITRON カーネルとμITRON アプリケーションを更新する。
・外部記憶装置が取り外し可能なメモリカードなら Windows マシンなどで書き換えればいいのでブートプログラムで書き換えができる必要はない。
・その他、システムの都合に合わせて必要な処理を行う

μITRON カーネル、μITRON アプリケーションの外部記憶のフォーマットとしては以下の形式が考えられます。

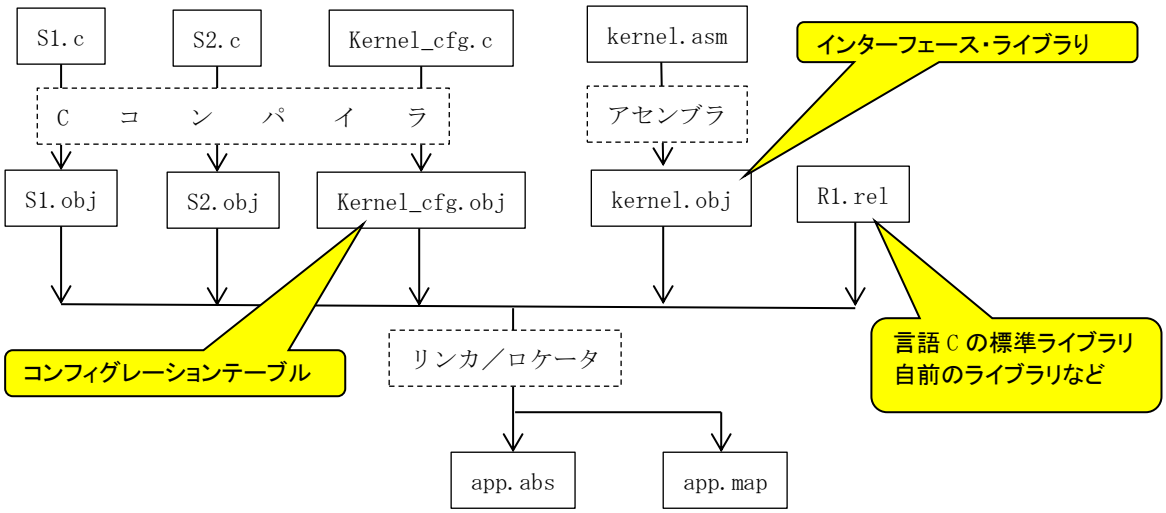
・COFF,OMF など　　リンカ／ロケータの出力形式
・HEX フォーマットまたは S フォーマット　COFF,OMF 形式から変換
・バイナリフォーマット（メモリイメージ）　HEX,S 形式から変換

(b) 実機でのサービスコールの呼びだし機構

μITRON カーネル、μITRON アプリケーションを別々にリンカ／ロケータで作成するとμITRON アプリケーションからμITRON カーネルのサービスコールの呼び出し方法に工夫がいります。μITRON カーネルに修正があった場合でもμITRON アプリケーションの実行可能形式ファイル(app.abs)を変更

しなくてもよいのが望ましいからです。１つの方法は、μITRON アプリケーションはカーネルのインターフェース・ライブラリとリンクして実行可能形式ファイル(app.abs)を作成することです。カーネルのインターフェース・ライブラリはソフトウェア割り込みを使いカーネルのサービスコール本体へパラメータを渡します。
以下の書籍でサービスコールの呼び出し機構を解説しているのでを参考にしてください。
μITRON」入門―“組み込み系”「リアルタイム OS」の基礎（I・O BOOKS）　工学社

μITRON アプリケーションモジュールの作成手順は以下のようになります



このようにμITRON アプリケーションモジュールには言語 C の初期化ルーチンは必要ありません。main 関数はなくてもかまいません。ブートプログラムがメモリにロードした段階で各セクション（.text，.rodata，.data，.bss など）は初期化済みとなることを前提にしています。

V2.00b　2019 年 7 月 1 日