

# $\mu$ ITRON 導入支援プログラム



V3.02

©2002 年 - 2026 年  
佐々木 芳

## はじめに

本書では、Windows 上で  $\mu$  ITRON 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  $\mu$  ITRON 仕様 OS を理解するためのトレーニングツールです。

$\mu$  ITRON 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  $\mu$  ITRON3.0 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も時間がとられました。そこで Windows 上の  $\mu$  ITRON 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しました。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  $\mu$  ITRON 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な文書の基本構成 (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ばいいなと思います。

### 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、 $\mu$  ITRON 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。

[\$\mu\$  ITRON」入門―“組み込み系”「リアルタイム OS」の基礎 \(I・O BOOKS\) 工学社](#)

### V3.00 リリースにあたって(2023, 3, 1)

コマンドとスクリプト機能の実装を全面的に見直しました。主な追加、変更機能は

- ・新規コマンドの追加、バグの修正
- ・コマンド構文の見直し
- ・数値リテラルに 16 進数、10 進数、2 進数を導入
- ・数値式の導入
- ・変数型マクロ、定義済み関数型マクロの導入
- ・スクリプト制御命令の導入
- ・本書の間違いを修正、追記
- ・サンプルプログラムを VisualStudio2019 Professional で動作確認。

最初は新しいコマンドを追加するだけで始めました。コマンド機能をテストする段階で同じ操作や

パラメータを変更して同じ操作を繰り返すことが頻繁に起こります。この作業ではスクリプトを作成して行くと履歴も残り、修正後に同じ操作がすぐできるので便利です。その過程でコマンドやスクリプトの機能不足を感じました。

コマンドとスクリプトを実行するプログラムはコマンドライン・インタプリタ (Command Line Interpreter) と呼ばれます。このコマンドライン・インタプリタの実装を見直し、全面的に書き換えました。以後本書ではコマンドライン・インタプリタを単に「CLI」と呼ぶことにします。一般に CLI は Command Line Interface を指しますが本書では Command Line Interface を実現するプログラムであるコマンドライン・インタプリタを指すことにします。

#### 参考文献

- [1]  $\mu$  ITRON4.0 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 柏木潤著 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>)
- [19] INTEL ASM386 Assembly Language Reference
- [20] 740 ファミリ用リロケータブルアセンブラ
- [21] C 言語で書くアルゴリズム 著者 Andrew Binstock/John Rex、訳者 岩谷宏、発行 ソフトバンク株式会社
- [22] C 言語による最新アルゴリズム事典 著者 奥村晴彦、発行 株式会社 技術評論者

Copyright (C) 2002-2023 佐々木芳. All Rights Reserved.

ホームページ [μ ITRON トレーナ Cmtoy](http://www.cmttoy.com)

<b>1</b>	<b>CMTOY の概要</b>	<b>11</b>
1.1	GUI(Graphical User Interface)の概要	12
1.2	ファイル構成	13
1.3	C-MACHINE とは	15
1.3.1	システムバス	16
1.3.2	制御信号	18
1.3.3	メモリシステム	19
1.3.4	CPU	20
1.3.5	メモリマップドIO とポートマップドIO	26
1.3.6	メモリバンク	29
1.3.7	割込みコントローラ(IRC)	29
1.3.8	シリアルコントローラ	31
<b>2</b>	<b>使用方法</b>	<b>33</b>
2.1	インストール	33
2.1.1	Windows Vista, Windows 7 でのインストール	34
2.1.2	Windows10 でコマンドプロンプトを起動するには	35
2.1.3	Windows11 でコマンドプロンプトを起動するには	35
2.2	CMTOY を起動する	36
2.2.1	serial.ocx の使用する TCP/IP ポート番号を変更する	37
2.2.2	作業ディレクトリを変更する	40
2.2.3	テキスト・エディターを変更する	40
2.2.4	変数型マクロを定義する	41
2.3	アプリケーションプログラムを実行する	42
2.4	インターバルタイマの操作	44
2.5	外部割込みの操作	46
2.6	ボリュームの操作	47
2.7	DIP スイッチ	48
2.8	押しボタン	48
2.9	表示専用 LED	48
2.10	シリアルポート	48
2.11	MITRON カーネルの状態を参照する	49
2.12	コマンドラインによる操作	50
2.12.1	コマンド・コンソール	50
2.12.2	ヘルプ機能	51
2.12.3	コマンドラインの実行	52
2.12.4	list コマンド実行	53
2.12.5	ショートカット・ボタン	53
2.12.6	clear コマンドを実行	54
2.13	スクリプトによる操作	54
2.13.1	スクリプトの実行モード	56
2.13.2	スクリプトファイルの文字符号化方式	57
2.14	出力ウインドウ	57
2.14.1	記入	58
2.14.2	ファイルへ保存	59
2.14.3	コンテキストメニュー	59
2.14.4	出力ウインドウ内の検索	61
2.15	ターゲットメモリ	61
2.16	コンソール端末	63

2.16.1	出力ウインドウ.....	64
2.16.2	TCP/IP 端末.....	64
2.17	CMTOY のバージョン情報.....	70
<b>3</b>	<b>アプリケーションプログラムの作成とデバッグ .....</b>	<b>72</b>
3.1	アプリケーションプログラムの作成方法.....	72
3.1.1	コンフィギュレーション.....	72
3.1.2	VisualStudio6.0 を使う .....	73
3.1.3	Visual C++ 2008 Express Edition を使う .....	76
3.1.4	Visual Studio 2017 を使う .....	77
3.2	ビルド方法 .....	79
3.2.1	Visual Studio 6.0 でのプロジェクトの設定.....	79
3.2.2	Visual C++ 2008 Express Edition でのプロジェクトのプロパティ.....	80
3.2.3	Visual Studio 2017 でのプロジェクトのプロパティ .....	80
3.2.4	Borland C++コンパイラ.....	81
3.3	VISUALSTUDIO6.0 のデバッガの使用 .....	81
3.3.1	μITRON アプリケーションのプロジェクトからデバッガを使う .....	81
3.3.2	Cmtoy 起動後にデバッガを使う .....	83
3.4	VISUALC++ 2008 EXPRESS EDITION のデバッガの使用 .....	84
3.4.1	μITRON アプリケーションのソリューションからデバッガを使う .....	84
3.5	VISUAL STUDIO 2017 のデバッガの使用.....	86
3.5.1	μITRON アプリケーションのソリューションからデバッガを使う .....	86
<b>4</b>	<b>MITRON カーネルの機能 .....</b>	<b>88</b>
4.1	カーネルの概要.....	88
4.1.1	外部割込み制御 .....	88
4.1.2	タスク .....	88
4.1.3	タイマ機能.....	89
4.1.4	Cmtoy 固有の機能.....	89
4.2	実装済みサービスコール一覧.....	93
4.3	CMTOY でのリセット動作.....	94
<b>5</b>	<b>C-MACHINE の機能.....</b>	<b>96</b>
5.1	データタイプ .....	96
5.2	CPU、割込み制御関数 .....	96
5.2.1	void halDisableInterrupt(void); .....	96
5.2.2	void halEnableInterrupt(void); .....	96
5.2.3	BOOL halInquireInterruptStatus(void);.....	96
5.2.4	void halMaskInterrupt(int level, BOOL mask);.....	97
5.2.5	void halEndOfInterrupt(int level);.....	97
5.3	デバッグ出力制御関数.....	97
5.3.1	void halDebugOutputString(const char *cstr);.....	97
5.3.2	void halDebugPrintf(const char *formatstring, ...);.....	97
5.4	LED 表示制御関数.....	98
5.4.1	void halSetLED(WORD led);.....	98
5.4.2	void halSetSegLED(WORD stat);.....	98
5.5	ボリューム制御関数 .....	99
5.5.1	WORD halGetVolume(int VolumeNo);.....	99
5.6	DIP スイッチ制御関数 .....	99
5.6.1	WORD halGetSwitch(void);.....	99

5.7	ボタン制御関数.....	99
5.7.1	<i>BOOL halGetPushButton(int ButtonNo);</i> .....	99
5.8	簡易シリアル制御関数.....	100
5.8.1	<i>void halSerialInit(int SerialNo);</i> .....	100
5.8.2	<i>int halSerialReadChar(int SerialNo);</i> .....	101
5.8.3	<i>void halSerialWriteChar(int SerialNo, int c);</i> .....	101
5.9	16550 相当のシリアル制御関数.....	101
5.9.1	<i>void hal16550WriteDATA(int SerialNo, BYTE d);</i> .....	103
5.9.2	<i>BYTE hal16550ReadDATA(int SerialNo);</i> .....	103
5.9.3	<i>void hal16550WriteIER(int SerialNo, BYTE d);</i> .....	103
5.9.4	<i>BYTE hal16550ReadIER(int SerialNo);</i> .....	104
5.9.5	<i>BYTE hal16550ReadIID(int SerialNo);</i> .....	104
5.9.6	<i>void hal16550WriteFCR(int SerialNo, BYTE d);</i> .....	105
5.9.7	<i>BYTE hal16550ReadLSR(int SerialNo);</i> .....	105
5.9.8	<i>BYTE hal16550ReadMSR(int SerialNo);</i> .....	105
5.9.9	<i>void hal16550WriteLCR(int SerialNo, BYTE d);</i> .....	106
5.9.10	<i>BYTE hal16550ReadLCR(int SerialNo);</i> .....	106
5.9.11	<i>void hal16550WriteMCR(int SerialNo, BYTE d);</i> .....	106
5.9.12	<i>BYTE hal16550ReadMCR(int SerialNo);</i> .....	107
5.10	PN 符号, 疑似ランダム雑音 (PSEUDORANDOMNOISE) の生成.....	107
5.10.1	<i>WORD halCalcPN9(WORD pn_code);</i> .....	107
5.10.2	<i>WORD halGenPN9(WORD pn_code, BYTE *buf, int bytes);</i> .....	107
5.10.3	<i>WORD halCalcPN15(WORD pn_code);</i> .....	107
5.10.4	<i>WORD halGenPN15(WORD pn_code, BYTE *buf, int bytes);</i> .....	107
5.11	マクロ.....	108
5.11.1	<i>CMTRACE (const char *formatstring, ...)</i> .....	108
5.11.2	CPU 制御.....	108
5.11.3	ターゲットメモリを操作 (アドレスを即値で使用する場合) .....	108
5.11.4	ターゲットメモリを操作する (構造体のメンバを使用する場合) .....	112
6	CLI(コマンドライン・インタプリタ).....	116
6.1	コマンド機能.....	116
6.1.1	コマンドラインの構文 (シンタックス) .....	117
6.1.2	コマンドラインの解析.....	119
6.1.3	コマンドパラメータ.....	120
6.1.4	前処理 (プリプロセス) .....	123
6.1.5	コマンドマクロ.....	125
6.1.6	コマンドラインの解析, 実行手順.....	127
6.2	スクリプト機能.....	128
6.2.1	スクリプト起動構文.....	129
6.2.2	定義済みオプションパラメータ.....	130
6.2.3	オプションパラメータを参照.....	131
6.2.4	実行制御.....	132
7	コンソール・コマンド一覧.....	134
7.1	CLI 操作.....	136
7.1.1	<i>ver.</i> .....	136
7.1.2	<i>nolist.</i> .....	136
7.1.3	<i>list [-d]   [-p]   [-s]   [-m]   [-all]</i> .....	136
7.1.4	<i>cd [&lt;path&gt;f]</i> .....	137

7.1.5	<code>dir</code> [ <code>&lt;path&gt;f</code> ] [ <code>-d</code> ] [ <code>-w</code> ] [ <code>-f</code> ]	138
7.1.6	<code>define</code> [ <code>&lt;symbol&gt;i</code> ] [ <code>&lt;string&gt;m</code> ]	138
7.1.7	<code>define_literal</code> [ <code>&lt;symbol&gt;i</code> ] [ <code>&lt;string&gt;m</code> ]	139
7.1.8	<code>define_input</code> <code>&lt;symbol&gt;i</code> [ <code>&lt;string array&gt;b</code> ] [ <code>-m&lt;説明&gt;m</code> ]	139
7.1.9	<code>undef</code> [ <code>&lt;symbol&gt;i ...</code> ] [ <code>-all</code> ] [ <code>-init</code> ]	140
7.1.10	<code>alias</code> [ <code>&lt;名前&gt;i</code> ] [ <code>&lt;command lines&gt;b</code> ] [ <code>-np</code> ] [ <code>-f</code> ]	140
7.1.11	<code>unalias</code> [[ <code>&lt;名前&gt;i ...</code> ] [ <code>-all</code> ]	144
7.1.12	<code>include</code> <code>&lt;filename&gt;f</code> [ <code>-D&lt;name&gt;=&lt;value&gt;</code> ] [ <code>-U&lt;name&gt;</code> ] [ <code>-F{SJIS   UTF8}</code> ] [ <code>{-L[dpsma]   {-S}}</code> ] [ <code>&lt;任意のオプションパラメータ列&gt;</code> ]	144
7.1.13	<code>include</code> <code>&lt;command lines&gt;b</code> [ <code>-D&lt;name&gt;=&lt;value&gt;</code> ] [ <code>-U&lt;name&gt;</code> ] [ <code>{-L[dpsma]   {-S}}</code> ] [ <code>&lt;任意のオプションパラメータ列&gt;</code> ]	144
7.1.14	<code>set_script_mode</code> <code>{I   E   S}</code>	145
7.1.15	<code>chcon</code> [ <code>&lt;コンソール番号&gt;ne</code> ]	145
7.1.16	<code>setcon</code> <code>&lt;コンソール番号&gt;ne</code> [ <code>-e&lt;ces&gt;</code> ] [ <code>-n&lt;newline&gt;</code> ]	145
7.2	ターゲットデバイス操作	146
7.2.1	<code>load</code> <code>&lt;ファイル名&gt;f</code>	146
7.2.2	<code>reset</code> [ <code>-t&lt;回数&gt;ne</code> ]	147
7.2.3	<code>int</code> <code>&lt;レベル1&gt;ne</code> [ <code>&lt;レベル2&gt;ne</code> ]	147
7.2.4	<code>set_interrupt_name</code> <code>&lt;レベル&gt;ne</code> [ <code>&lt;表示名&gt;s</code> ]	147
7.2.5	<code>timerlog</code> <code>{ON   OFF}</code>	147
7.2.6	<code>timer</code> [ <code>&lt;回数&gt;ne</code> ] [ <code>-s</code> ]	148
7.2.7	<code>wait_timer</code> [ <code>&lt;回数&gt;ne</code> ]	148
7.2.8	<code>setpush</code> <code>{UP   DOWN}</code>	148
7.2.9	<code>inivolume</code> <code>&lt;最大値&gt;ne</code>	148
7.2.10	<code>setvolume</code> <code>&lt;現在値&gt;ne</code>	148
7.2.11	<code>setswitch</code> <code>&lt;スイッチ番号&gt;ne</code> <code>{ON   OFF}</code>	149
7.2.12	<code>set_switch_name</code> <code>&lt;スイッチ番号&gt;ne</code> [ <code>&lt;表示名&gt;</code> ]	149
7.2.13	<code>irc</code> [ <code>-l&lt;level&gt;ne</code> ] [ <code>-e</code> ] [ <code>-m{ON   OFF}</code> ]	149
7.3	ターゲットメモリ操作	150
7.3.1	<code>define_mem</code> <code>&lt;メモリサイズ&gt;ne</code> <code>&lt;IO サイズ&gt;ne</code> <code>{BE   LE}</code> <code>{BA   WA}</code> [ <code>-f</code> ]	151
7.3.2	<code>add_mem_area</code> <code>&lt;領域名&gt;i</code> <code>&lt;ベース&gt;ne</code> <code>&lt;サイズ&gt;ne</code> <code>&lt;バンク数&gt;ne</code> <code>{R   RW}</code> [ <code>-V</code> ]	152
7.3.3	<code>add_permanent_area</code> <code>&lt;領域名&gt;i</code> <code>&lt;ベース&gt;ne</code> <code>&lt;サイズ&gt;ne</code> <code>&lt;バンク数&gt;ne</code> <code>{R   RW}</code> [ <code>&lt;ファイル名&gt;f</code> ]	152
7.3.4	<code>add_io_area</code> <code>&lt;領域名&gt;i</code> <code>&lt;ベース&gt;ne</code> <code>&lt;サイズ&gt;ne</code>	153
7.3.5	<code>delete_area</code> <code>&lt;領域名&gt;i</code>	153
7.3.6	<code>erase_area</code> <code>&lt;領域名&gt;i</code>	153
7.3.7	<code>rotate_bank</code> <code>&lt;領域名&gt;i</code> [ <code>-i&lt;レベル&gt;ne</code> ]	154
7.3.8	<code>fill_bank</code> <code>&lt;領域名&gt;i</code> [ <code>&lt;バンク番号&gt;ne</code> ] <code>{-PN9   -PN15}</code> [ <code>-init</code> ]	154
7.3.9	<code>fill_bank</code> <code>&lt;領域名&gt;i</code> [ <code>&lt;バンク番号&gt;ne</code> ] <code>-t&lt;fill_spec&gt;</code> [ <code>-init</code> ]	154
7.3.10	<code>set_bank</code> [ <code>&lt;領域名&gt;i</code> ] [ <code>&lt;バンク番号&gt;ne</code> ] [ <code>&lt;ファイル名&gt;f</code> ] [ <code>-o&lt;オフセット&gt;ne</code> ]	155
7.3.11	<code>copy_bank</code> <code>&lt;先領域名&gt;i</code> [ <code>&lt;バンク番号&gt;ne</code> ] [ <code>&lt;元領域名&gt;i</code> ] [ <code>&lt;バンク番号&gt;ne</code> ]	156
7.3.12	<code>set</code> [ <code>-{s   p}&lt;アドレス&gt;a</code> ] [ <code>{-b[u]   w   l}</code> ] [ <code>&lt;xx&gt;ne</code> ] [ <code>&lt;string&gt;q</code> ] ... [ <code>-i&lt;レベル&gt;ne</code> ]	156
7.3.13	<code>get</code> [ <code>-{s   p}&lt;アドレス&gt;a</code> ] [ <code>-b   w   l   c[u]</code> ] [ <code>&lt;個数&gt;ne</code> ]	158
7.3.14	<code>fill</code> [ <code>-{s   p}&lt;アドレス&gt;a</code> ] [ <code>{-b   bu   w   l}</code> ] [ <code>&lt;x&gt;ne ...</code> ] [ <code>&lt;string&gt;q</code> ]	159
7.3.15	<code>wait</code> [ <code>-{s   p}&lt;アドレス&gt;a</code> ] [ <code>-b   w   l</code> ] [ <code>&lt;xx&gt;ne</code> ] [ <code>{OR   AND}</code> ] [ <code>-t&lt;タイムアウト&gt;ne</code> ]	160
7.3.16	<code>peek</code> [ <code>-{s   p}&lt;アドレス&gt;a</code> ] [ <code>-b   w   l</code> ]	160
7.3.17	<code>poke</code> <code>-O&lt;操作&gt;</code> [ <code>-{s   p}&lt;アドレス&gt;a</code> ] [ <code>-b   w   l</code> ] [ <code>&lt;data&gt;ne</code> ] [ <code>-i&lt;レベル&gt;ne</code> ]	161

7.3.18	<i>sum</i> <i>-{s   p}&lt;アドレス&gt;a -{b   w   l}&lt;個数&gt;ne</i> .....	162
7.3.1	<i>crc</i> <i>[-{s   p}&lt;アドレス&gt;a -b&lt;個数&gt;ne -t{L   R}&lt;ビット数&gt;ne -P&lt;poly&gt; [-I&lt;init&gt;] [-X]]</i> 163	
7.3.2	<i>tmem</i> .....	165
7.4	SERIAL <シリアルポート番号>NE <サブコマンド> .....	166
7.4.1	<i>init</i> <割込みレベル>ne <チップ種別>s .....	166
7.4.2	<i>info</i> .....	167
7.4.3	<i>set</i> <i>{CTS   DSR   RI   DCD} {ON   OFF}</i> .....	167
7.4.4	<i>push</i> <i>{&lt;xx&gt;ne   &lt;アスキー&gt;"   sb   se} ... [-c&lt;間隔&gt;ne]</i> .....	167
7.4.5	<i>probe</i> <i>{DTR   RTS   OUT1   OUT2} [-w&lt;タイムアウト&gt;ne]</i> .....	168
7.5	ユーティリティ・コマンド .....	168
7.5.1	<i>clear</i> .....	169
7.5.2	<i>print</i> <メッセージ>m .....	169
7.5.3	<i>trace</i> <メッセージ>m .....	169
7.5.4	<i>help</i> <i>[*][&lt;コマンド名&gt;s]</i> .....	169
7.5.5	<i>edit</i> <i>[&lt;ファイル名&gt;f]</i> .....	169
7.5.6	<i>win_app</i> <ファイル名>f <i>[&lt;string&gt;m]</i> .....	170
7.5.7	<i>calc</i> <数値式>ne .....	170
7.5.8	<i>set_title</i> <タイトル>m .....	171
8	定義済みマクロ .....	172
8.1	定義済み変数型マクロ .....	172
8.1.1	<i>_APP_NAME</i> .....	172
8.1.2	<i>_APP_EXE</i> .....	172
8.1.3	<i>_APP_VER</i> .....	172
8.1.4	<i>_CM_VER</i> .....	172
8.1.5	<i>_KP_VER</i> .....	172
8.1.6	<i>_CLI_VER</i> .....	173
8.1.7	<i>_WD</i> .....	173
8.1.8	<i>_DATE</i> .....	173
8.1.9	<i>_TIME</i> .....	173
8.1.10	<i>_TIMESTAMP</i> .....	173
8.1.11	<i>_FILE</i> .....	174
8.1.12	<i>_FILE_NAME</i> .....	174
8.1.13	<i>_FILE_PATH</i> .....	174
8.1.14	<i>_DIR_PATH</i> .....	174
8.1.15	<i>_LINE</i> .....	174
8.1.16	<i>_FILE_HIST</i> .....	175
8.1.17	<i>_MSGBOX</i> .....	175
8.1.18	<i>_EXIT_CODE</i> .....	175
8.1.19	<i>_R</i> .....	175
8.1.20	<i>_ZF</i> .....	175
8.1.21	<i>_SF</i> .....	175
8.2	定義済み関数型マクロ .....	176
8.2.1	<i>16</i> 進数文字列を生成 .....	176
8.2.2	メモリ/IO アドレス .....	181
8.2.3	スクリプトパラメータ .....	181
8.2.1	システムの状態 .....	183
8.2.2	その他 .....	185



<b>9</b>	<b>スクリプト制御機能</b>	<b>189</b>
9.1	スクリプト制御命令の構文	189
9.2	スクリプト制御命令一覧	190
9.2.1	<code>#exit [&lt;終了コード&gt;ne] [-if&lt;条件式&gt;ne]</code>	190
9.2.2	<code>#abort [-if&lt;条件式&gt;ne]</code>	190
9.2.3	<code>#break [-if&lt;条件式&gt;ne]</code>	190
9.2.4	<code>#msgbox &lt;スタイル&gt;ne -t &lt;タイトル&gt; -m &lt;文字列&gt;</code>	191
9.2.5	<code>#if&lt;条件式&gt;ne</code>	192
9.2.6	<code>#ifdef &lt;変数型マクロ名&gt;i</code>	193
9.2.7	<code>#ifndef &lt;変数型マクロ名&gt;i</code>	193
9.2.8	<code>#else</code>	193
9.2.9	<code>#endif</code>	194
<b>10</b>	<b>MITRON チュートリアル</b>	<b>195</b>
10.1	ステップ 1 (APP1.DLL)	196
10.1.1	システム要求仕様	196
10.1.2	システム分析	197
10.1.3	実装設計	198
10.2	ステップ 2 (APP2.DLL)	199
10.2.1	システム要求仕様	199
10.2.2	システム分析	200
10.2.3	実装設計	200
10.2.4	ハイパーターミナルの設定方法	202
10.2.5	PuTTY の設定方法	203
10.3	ステップ 3 (APP3.DLL)	206
10.3.1	システム要求仕様	206
10.3.2	システム分析	207
10.3.3	実装設計	209
10.4	ステップ 4 (APP4.DLL)	211
10.4.1	システム要求仕様	211
10.4.2	システム分析	211
10.4.3	実装設計	212
10.5	ステップ 5 (APP5.DLL)	213
10.5.1	システム要求仕様	213
10.5.2	システム分析	214
10.5.3	実装設計	215
10.6	ステップ 6 (APP6.DLL)	217
10.6.1	システム要求仕様	217
10.6.2	システム分析	217
10.6.3	実装設計	218
<b>11</b>	<b>C-MACHINE のプログラム例</b>	<b>220</b>
11.1	タスクと割込みハンドラの例	220
11.1.1	ファイル構成	220
11.1.2	システムの概要	220
11.1.3	使用関数、マクロ	222
11.2	ターゲットメモリの例	223
11.2.1	ファイル構成	223
11.2.1	システムの概要	224

11.2.2	使用関数、マクロ .....	228
11.3	16550 制御例 .....	229
11.3.1	ファイル構成 .....	229
11.3.2	システムの概要 .....	229
11.3.1	使用関数、マクロ .....	230
11.3.2	端末 PuTTY への表示 .....	231
12	考察 .....	234
12.1	VISUALSTUDIO6.0 のデバッガ .....	234
12.2	REGSVR32 .....	234
12.3	BORLAND C++ 5.5.1 .....	234
12.4	UML について .....	235
12.5	VISUAL C++ 2008 EXPRESS EDITION .....	235
12.6	WINDOWS XP 以前の OS へのインストール .....	235
12.7	WINDOWS VISTA、WINDOWS 7 で使用する場合 .....	236
12.7.1	ハイパーターミナル .....	236
12.7.2	コマンドプロンプト .....	238
12.8	WINDOWS10 で VISUAL STUDIO 6.0 を使う方法 .....	238
12.9	システム初期化手順 .....	238
12.9.1	実機での初期化手順 .....	238
12.9.2	C 言語の処理系での初期化 .....	239
12.9.3	プログラムをメモリへ配置する .....	240
12.10	「マクロ」について .....	243
12.10.1	マクロアセンブラのテキストマクロ .....	243
12.10.2	ASM386 のコードマクロ .....	244
12.10.3	C/C++ 言語のプリプロセッサ・マクロ .....	244
12.10.4	VBA (Visual Basic for Applications) とマクロ .....	244
12.11	VISUALSTUDIO のバージョンによる違い .....	245
12.11.1	MFC の CFile クラス .....	246

# 1 Cmtoy の概要

Cmtoy は、Windows のマルチバイト文字セット (MBCS) を採用した 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                       $\mu$  ITRON カーネルをシミュレート (固定ファイル名)
- app.dll                         $\mu$  ITRON 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  $\mu$  ITRON 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。

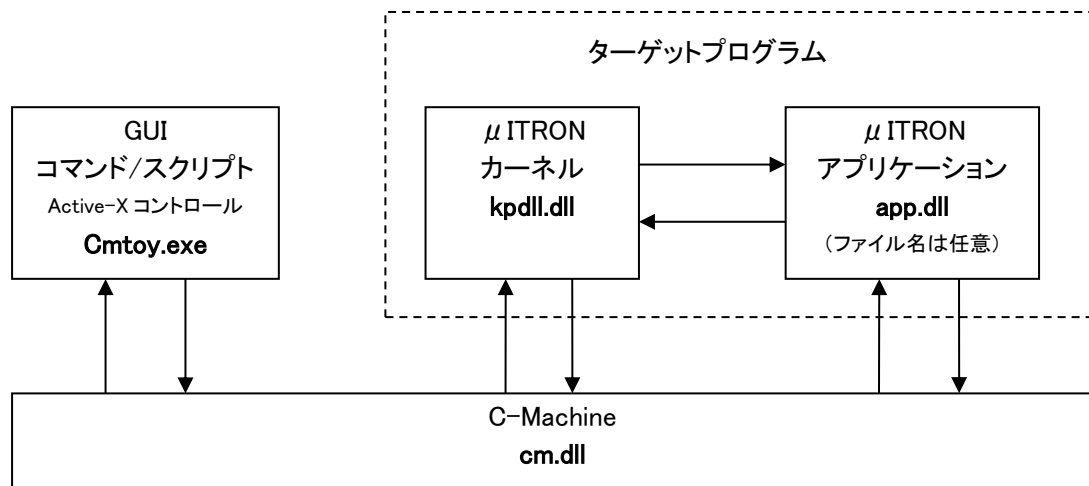


図 1-1 プログラム構成

$\mu$  ITRON カーネルは Cmtoy の一部として提供しています。皆さんは  $\mu$  ITRON 上のアプリケーションを C 言語で記述して、実行形式を 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  $\mu$  ITRON 上のアプリケーションファイル (\*.dll) を指定してロードし、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、 $\mu$  ITRON 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。

※Microsoft の無料の IDE (統合開発環境) としては、以下のバージョンがあるようです。

Visual Studio Express (2008, 2015, 2017 版など)

Visual Studio Community 2013

Visual Studio Community 2019

[Visual Studio Community 2022](#)

※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 (Graphical User Interface) 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ィ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。

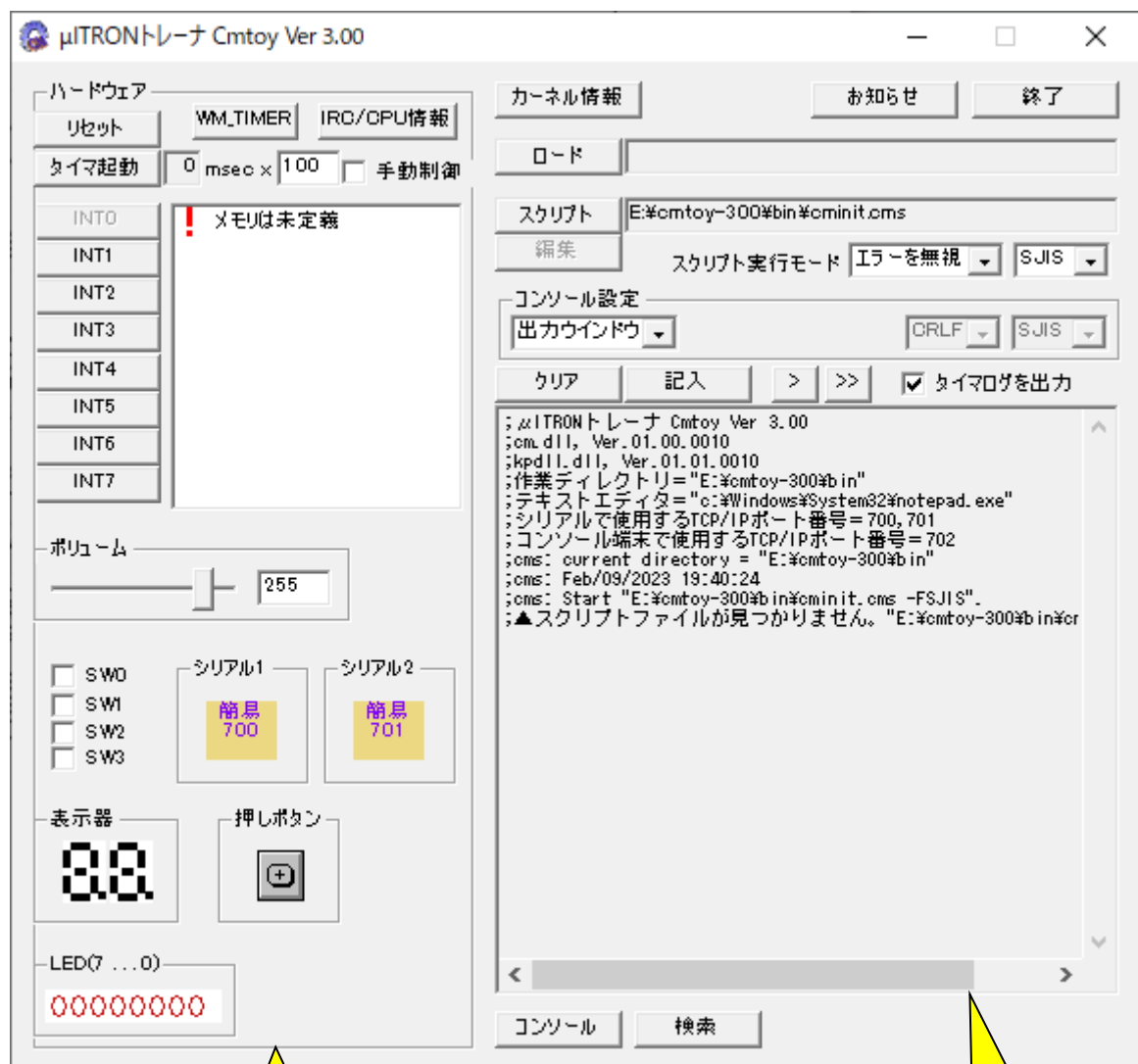


図 1-2 Cmtoy の GUI 構成

ハードウェア  
コントロール群

出力ウィンドウ

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  $\mu$  ITRON 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。
- ・  $\mu$  ITRON 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/I/O 空間の参照。
- ・ ターゲットプログラム（ $\mu$  ITRON 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。

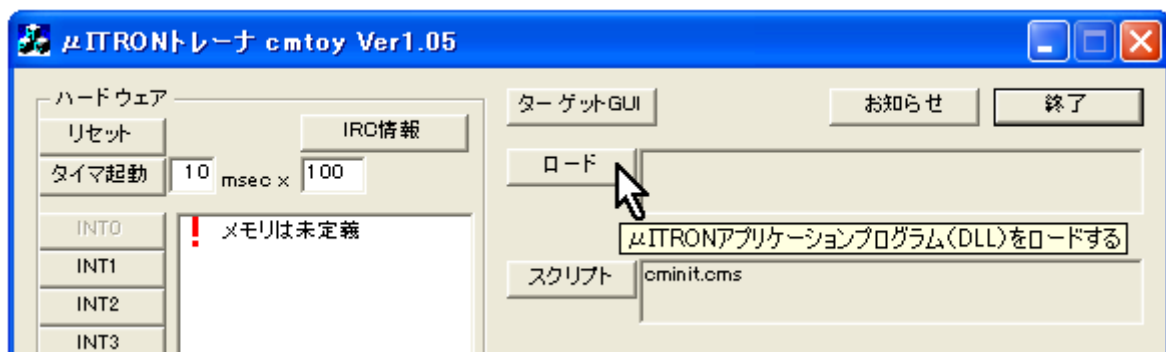


図 1-3 「ロード」 ボタンのツールチップ

## 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-300¥	
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	$\mu$ ITRON カーネルの実行モジュール
app1. dll	$\mu$ ITRON 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	$\mu$ ITRON 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	$\mu$ ITRON 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	$\mu$ ITRON 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	$\mu$ ITRON 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	$\mu$ ITRON 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	TCP/IP ポートを制御する 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_sample¥	各種スクリプトの例
sample¥	プログラム例①の実行ファイル、スクリプトファイル
memory¥	プログラム例②の実行ファイル、スクリプトファイル
16550¥	プログラム例③の実行ファイル、スクリプトファイル
include¥	アプリケーションプログラム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	μ ITRON 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	μ ITRON 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	μ ITRON 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)
cmtoy_702.ht	ハイパーターミナル定義ファイル (ポート 702 用)
mITRON¥	μ ITRON 用サンプルプログラム
sample¥	プログラム例① <a href="#">11.1 タスクと割込みハンドラの例</a> のソースプログラム
kernel_cfg.c	μ ITRON コンフィグ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ッグ出力
app¥	<b>V3.00 で VisualStudio6.0 のワークスペースファイルを分離</b>
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 がエ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
app_08¥	<b>V3.00 でソリューションファイルを分離</b>
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_17¥	<b>V3.00 でソリューションファイルを分離</b>
app.sln	app.dll を作成する Visual Studio 2017 Professional のソリューションファイル
Bccapp¥	
bccapp.bdp	<a href="#">BCC Developer</a> のプロジェクトファイル
sample_memory¥	プログラム例② <a href="#">11.2 ターゲットメモリの例</a> のソースプログラム
sample_16550¥	プログラム例③ <a href="#">11.3 16550 制御例</a> のソースプログラム
step1¥	μ ITRON チュートリアル <a href="#">10.1 ステップ 1 (app1.dll)</a> のソースプログラム
app1¥	<b>V3.00 で VisualStudio6.0 のワークスペースファイルを分離</b>
app.dsw	app1.dll を作成する VisualStudio6.0 のプロジェクトファイル
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
app1_08¥	<b>V3.00 でソリューションファイルを分離</b>
app.sln	app1.dll を作成する Visual C++ 2008 Express Edition のソリューションファイル
app1_17¥	<b>V3.00 でソリューションファイルを分離</b>
app.sln	app1.dll を作成する Visual Studio 2017 Professional のソリューションファイル
step2¥	μ ITRON チュートリアル <a href="#">10.2 ステップ 2 (app2.dll)</a> のソースプログラム
step3¥	μ ITRON チュートリアル <a href="#">10.3 ステップ 3 (app3.dll)</a> のソースプログラム
step4¥	μ ITRON チュートリアル <a href="#">10.4 ステップ 4 (app4.dll)</a> のソースプログラム
step5¥	μ ITRON チュートリアル <a href="#">10.5 ステップ 5 (app5.dll)</a> のソースプログラム
step6¥	μ ITRON チュートリアル <a href="#">10.6 ステップ 6 (app6.dll)</a> 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。「 <a href="#">11.2.1(4) デバッグコマンドの拡張</a> 」を参照。
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 デバッグバージョンのビルド後、*.dll を¥bin にコピーしない デバッグバージョンの「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)

Cmtoy V3.00 での変更点。

debug.c	先頭で_CRT_SECURE_NO_WARNINGS を定義（警告 C4996 を抑制）、バグの修正。 「 <a href="#">10.2.3 実装設計</a> 」を参照。
serial.ocx	機能追加(V1.7)。「 <a href="#">2.16.2 TCP/IP 端末</a> 」を参照。

### 1.3 C-Machine とは

ここではC-Machine が想定しているコンピュータ・ハードウェアについて考察します。ハードウェアのうちコンピュータ上で実行されるプログラムの論理的な動作に関係する部分を取り出して考察します。これはプログラマにとっては実ハードウェアと C-Machine で同じようにプログラム設計ができることを意味します。例えば、外部割込みが発生した順番で決まる動作は C-Machine 上でも実ハードウェアでも同じになります。

実際のハードウェアの実時間動作タイミングは以下の要因で変わりますが、プログラムの論理的な動作は同じことが保証されます。

- CPU が違えば異なる。動作周波数が異なる。
- 同じ命令セットを持った x86 CPU でも Intel 製と AMD 製では同じ周波数でも CPU 内部のタイミングは異なるし、命令のクロック数も異なることがある。
- 同じ CPU で同じ動作周波数でもメモリの性能やキャッシュメモリの量が違えば異なる。
- 周辺装置（メモリ、IC、LSI）によりタイミングが変わる。

C-Machine で前提とするコンピュータモデルはいわゆるノイマン型 (von Neumann architecture) です。主な特徴は、

- CPU (Central Processing Unit) とアドレス付けされた記憶装置 (メモリ)
- プログラム内蔵方式 (命令とデータを記憶装置に配置する)
- CPU と記憶装置、周辺装置はバス (bus) で接続 (バス型トポロジ) され、同一クロックに同期して動作する (クロック同期)。

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。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。

ハードウェアの主な構成要素（周辺装置）は以下のとおりです。

- 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 ( $\mu$ ITRON 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/I/O 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 I/O、ポート I/O 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

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（バス型トポロジ）。CPU、周辺装置は実際には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。多くの場合 CPU がバスマスターとなりバスの通信を開始しますが CPU 以外の装置がバスマスターになることもあります。

バス通信の特性では必ず CPU と周辺装置の 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 では CPU が実行する「ハードウェアを制御するプログラム」の構造、制御フロー、アルゴリズムを実機と変えずに動かすことを狙っています。

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します。

ここではハードウェアをシミュレートするために必要な要素を取り出して考察をします。

### 1.3.1 システムバス

一般的に CPU と周辺装置のバスによる接続関係を以下のようなブロック図で表現します。

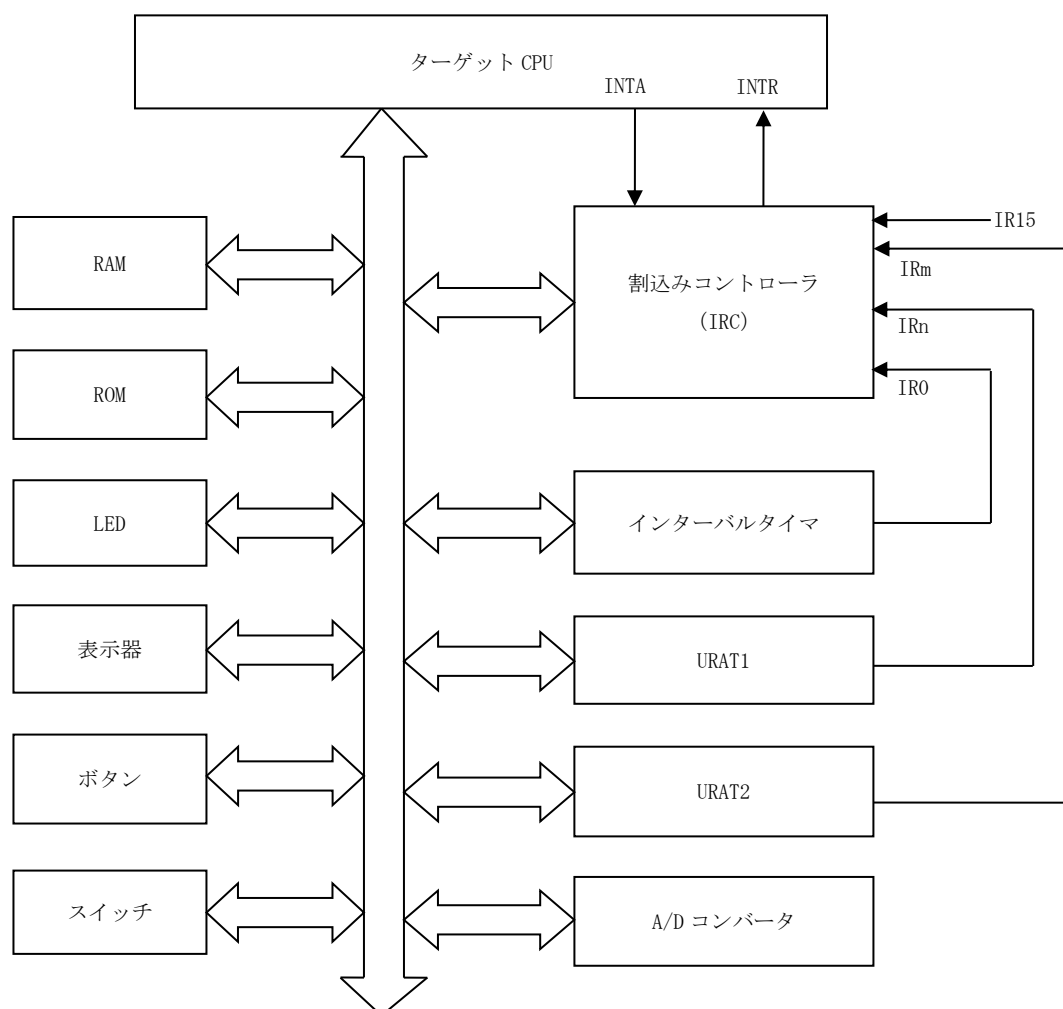
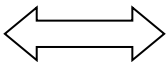


図 1-4 ターゲットハードウェアのブロック図

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。アドレスバスとデータバスで同じ信号線を使う CPU もあるし、信号線を分けている CPU 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 (物理メモリ) の最大サイズが決まります。また、ポート I/O 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  $0 \sim 2^{20}-1$  (0xfffff) となり、ポートアドレスは  $0 \sim 2^{16}-1$  (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通信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に合わせて各信号を制御します (クロック同期)。

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するわけではなく、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、その周辺装置へたいいてい 1 ラインのチップセレクト (**chip select**) 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 と周辺装置は 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。制御信号にはバスサイクルを指定する信号線が含まれています。例えば 8086 には以下のバスサイクルがあり、これらを区別するための 3 ラインの制御信号を持ちます。

- 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 I/O 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 I/O 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 は命令をメモリから読み込むために⑤を使います。命令を解釈してメモリ操作命令なら続いてバスサイクルを使います。メモリへ書き込む命令では⑥を使い、メモリからのデータを読み込む命令では⑦を使います。リードモディファイライト命令では⑥と⑦を使います。

リードモディファイライト命令とは、以下のような処理をする命令です。

- ・ 指定された番地のメモリをテンポラリレジスタに読み込む (⑥を使う)
- ・ テンポラリレジスタを書き換える (AND, OR, XOR などの演算、値の代入)
- ・ テンポラリレジスタを元のメモリに書き戻す (⑦を使う)

※8086 は 1978 年にインテル (Intel Corporation) が発表した 16 ビットマイクロプロセッサです。x86 アーキテクチャの最初のプロセッサです。アドレスバスは 20 ビット、データバスは 16 ビット

トで CPU 周波数は 5 MHz から 10 MHz で 1990 年まで製造された。

※命令のバイト数は CPU により違います。8086 は可変長命令（1～5 バイト）を採用しているので命令の 1 バイト目で続くバイト数が分かるような命令になっています。

※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。

### 1.3.2 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。

信号シンボル	入出力	信号名と説明
A19-A0	出力	メモリまたは I/O アドレス（この場合は 20 ビットバス）
D15-D0	入出力	データバス（この場合は 16 ビットバス）
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/I/O	出力	STATUS LINE: メモリアクセスと I/O 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 I/O リードサイクル
WR	出力	WRITE: メモリライトサイクルまたは I/O 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。CPU は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。

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。そのため必ずバスの使用权を持った装置がバスの制御を開始します。この使用权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权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。

CPU 以外の周辺装置がバスマスターになる場合は、CPU へ HOLD 信号を送出します。CPU は HOLD 信号を受け取ると規定の手順に従いバスを放棄し HLDA 信号でそれを周辺装置に知らせます（HLDA 信号以外の出力端子をハイインピーダンスにする）。周辺装置はメモリ操作が終了したら HOLD 信号の送出をやめます。CPU は HOLD 信号のなくなったことを確認し HLDA の送出を止めバスマスターに戻り

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。

※x86 アーキテクチャでは、命令実行中は LOCK 信号で保護されるので、命令実行中に中断してバスマスターを放棄することはありません。

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（プログラムカウンタ）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数間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、各バスサイクル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, Timing Diagram）が載っています。

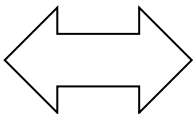
### 1.3.3 メモリシステム

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。）

C 言語の データタイプ	メモリセルのタイプ	
	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
int, unsigned int	16 または 32bit	16 または 32bit
long, unsigned long	32bit	32bit

バイト列 00, 01, 02, 03, 04, 05, 06, ... をワードアドレッシングの a000 番地のメモリ上に配置する場合以下ようになります。

バイト列		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
00		a000	0100	0001
01				
02		a001	0302	0203
03				
04		a002	0504	0405
05				
06		a003	0706	0607
07				
08		a004	0908	0809
09				
0a		a005	0b0a	0a0b
0b				

※物理的なメモリチップは通常 8 ビットメモリセルで物理的には 8 ビット単位でアドレス指定します。そのため、ワードアドレッシングのメモリシステムにおいて CPU の扱う 16 ビットメモリセルと物理的な 8 ビットメモリセルとの関係はハードウェア設計依存となりますが、Cmtoy では上記のような関係を想定します。

### 1.3.4CPU

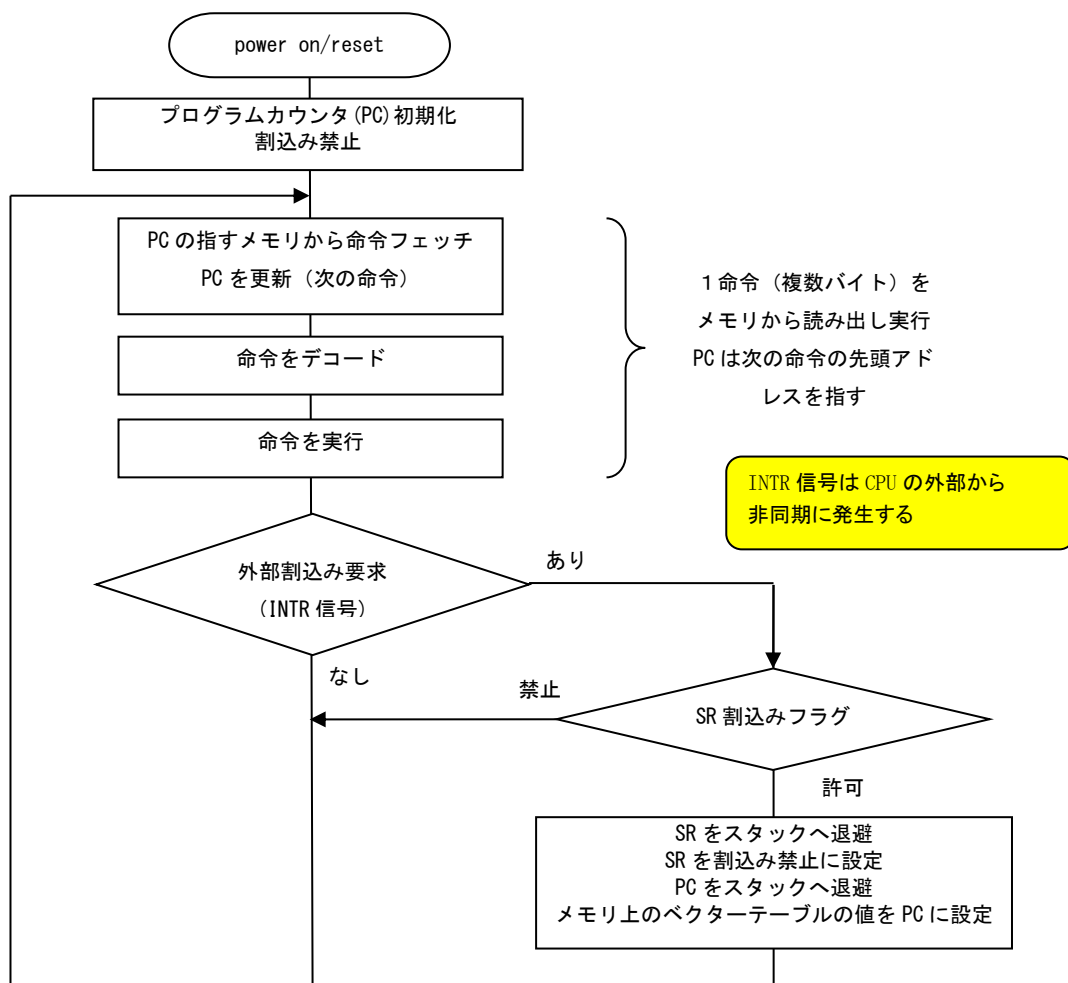
まずプログラマから見える CPU の動作、プログラミングモデルを確認します。プログラム内蔵方式 CPU ではメモリを次のような用途に対応した領域に分けて使います。

- ・ コード領域（命令コード、定数を格納する、書き換え不可で読み出しのみ）
- ・ データ領域（読み書き可能）
- ・ スタック領域（後入れ先出しのデータ構造として使う、読み書き可能）
- ・ 割込みベクターテーブル（書き換え不可または読み書き可能）

これらの領域を効率よく操作するためと演算を効率よく行うため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）      命令実行するたびに変更される内部状態を示す。  
割込み禁止／許可の 1 ビットの割込みフラグも含む。
- ・ 汎用レジスタ（n 個）      演算オペランド、演算結果格納、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。



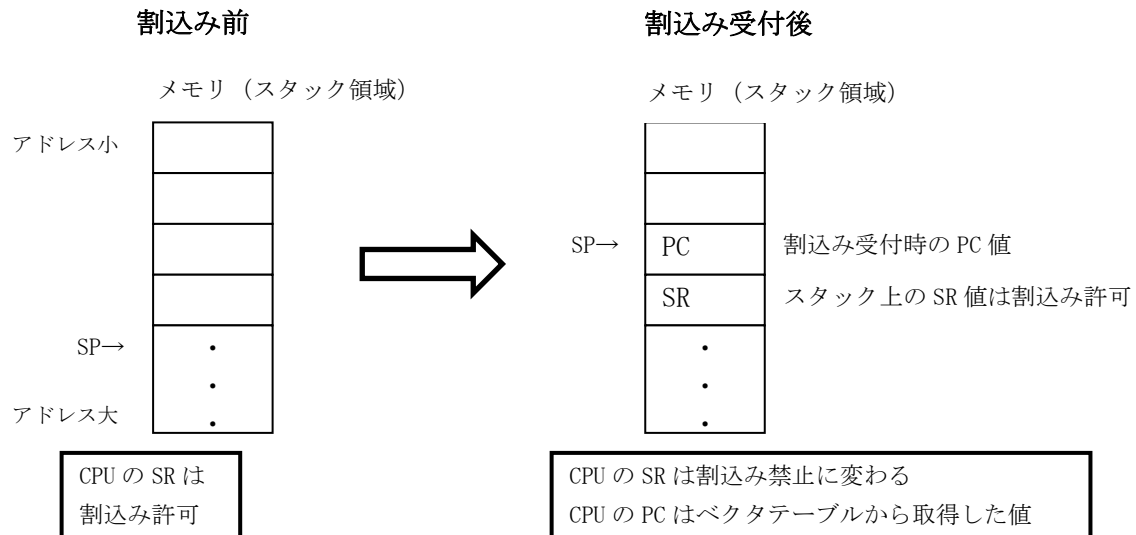
### 1-5 CPU の動作

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。割り込み要求が発生すると割り込みアクリッジサイクルで割り込み要求に対応するベクタ番号を読み出し、メモリ上のベクター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。

CPU の動作をまとめると、

- ・ 1 命令実行するために複数のバスサイクルを使用する。
- ・ 割り込みハンドラの起動は命令終了時に行われる。ある命令を実行している最中のバスサイクルを中断して割り込みハンドラに移行することはない。

割り込み受付前後の CPU、スタックの様子を以下に示します。



### (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）

割り込み禁止／許可フラグを持つ

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 またはその動作モードです。

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。

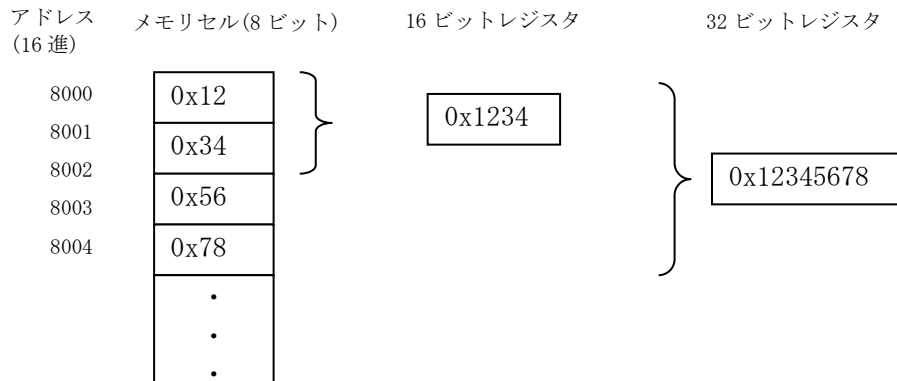
```
char *p = (char*) 0xe000;
*p = 0;
```

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。

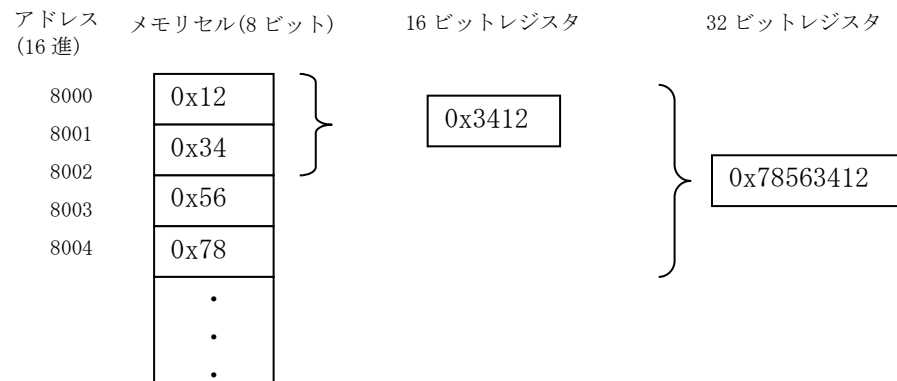
ターゲット CPU は 8 ビット、16 ビット、32 ビットの整数データを扱います。このデータをメモリに保存する方法を考察しましょう。ビッグエンディアン (**big endian**) とリトルエンディアン (**little endian**) という 2 つの保存方法を説明します。どちらもメモリセルのサイズより大きなデータは連続したメモリセルに保存します。

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



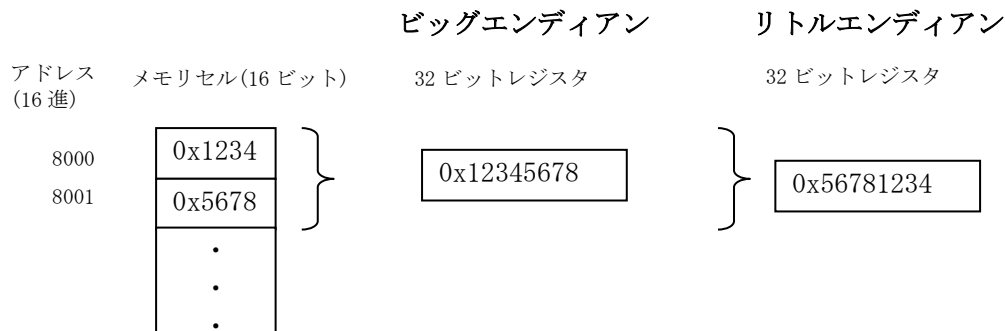
## 8 ビットメモリセル上のデータの並び

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  $\mu$ 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。C 言語でカバーしきれない CPU に依存する処理はインラインアセンブラを使いマクロで記述すれば、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、I/O 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 I/O 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 I/O 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)



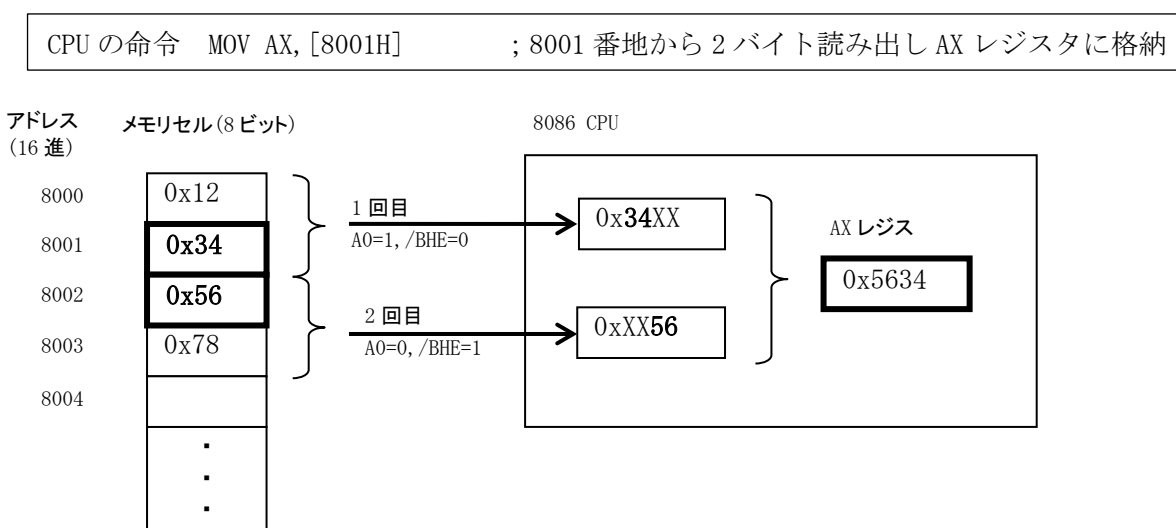
- ⑤ 命令フェッチサイクル (CPU がメモリから命令コードを読みだす)
- ⑥ メモリリードサイクル (CPU がメモリからデータを読みだす)
- ⑦ メモリライトサイクル (CPU がメモリへデータを書きだす)

バスアービトレーションには HOLD, HLDA, LOCK 信号を使います。CPU は LOCK 信号を使い一連のバスサイクルを保護し、終了するまで他のバスマスターへ制御権を譲りません。

※ 1 回のリードサイクル、ライトサイクルはデータバスのサイズでメモリとデータ転送を行います。

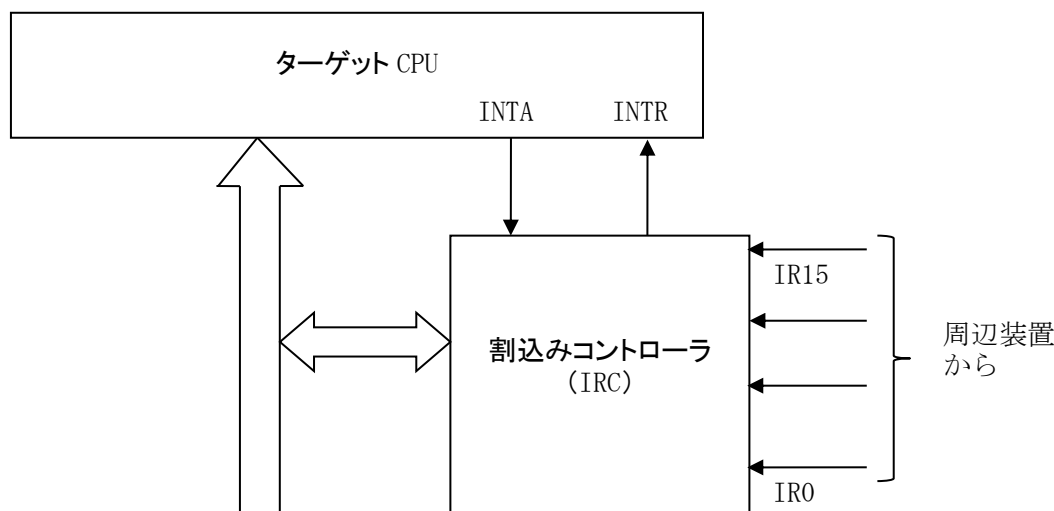
16 ビットデータバスなら 2 バイト境界、32 ビットデータバスなら 4 バイト境界でデータ転送をします。8086 の 16 ビットデータバスシステムで CPU の命令が奇数アドレス  $n$  で 16 ビットリード命令 (MOV AX,  $n$ ) を実行した場合には、2 回のリードサイクルが実行されます。8086 は 1 回目は  $n-1$  偶数アドレスから 16 ビットデータを読み出し、2 回目は  $n+1$  偶数アドレスから 16 ビットデータを読み出して、1 回目の上位 8 ビットと 2 回目の下位 8 ビットを取り出して 16 ビットに合成して AX レジスタ格納します。

以下は概念図です。実際に 8086 がどのように信号制御しているかはデータシートを確認してください。(アドレス A0 信号と /BHE (BusHighEnable) 信号を制御して行っている)



### (3) 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  $IR_n$  ( $n=0\sim 15$ ) 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 の割り込みフラッグを割り込み禁止にする。
- ⑤ CPU は IRC から割り込み番号を読み取る。（割り込みアクノリッジサイクル）
- ⑥ CPU は割り込み番号からベクター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。

※8086 では割り込みベクターテーブルは 0 番地から予約されていて、256 個の割り込みハンドラプログラムのアドレスを格納する。外部割り込みはこの中の連続した 8 個～を使用する。8259A にはベクターテーブルの先頭のベクタ番号を初期化プログラムで設定しておく。

※8359A はインテル（Intel Corporation）が開発した 8086 で使用できる **Programmable Interrupt Controller** です。8 レベル以上の割り込みを制御する場合はマスター／スレーブのカスケード接続で使用する。

### 1.3.5 メモリマップド I/O とポートマップド I/O

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）と呼ばれます。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、I/O ポートアドレスに配置されたレジスタは I/O リードサイクル／I/O 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 I/O 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空間内に存在します。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 内のデータは変更できません。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 0x00 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 と関係なく値が変わります。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、書き込む値と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。この性質は 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 I/O 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。

アドレス      メモリ (8 ビット)

ff000	com
ff001	status
	•
	•
	•

```
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最適化を行い、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す。

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となりますが、メモリマップド I/O の場合は違う意味になります。メモリマップド I/O 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 I/O 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 I/O とポートマップド I/O 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 I/O を使う

ポート I/O 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。

x86 互換 CPU の場合、I/O 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。

```

// I/O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;
}

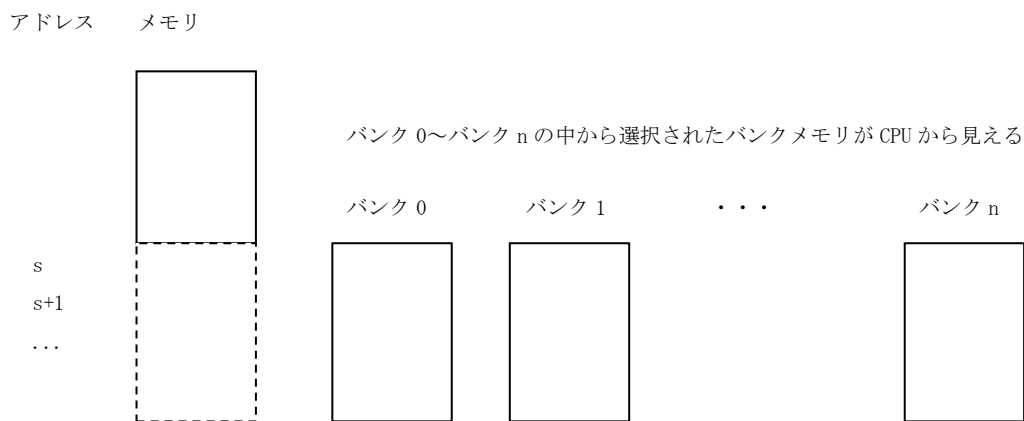
// I Oポートへ出力
void output_dword( WORD port, DWORD data)
{
    _asm{
        move eax,data
        mov dx,port
        out dx,eax
    }
}

```

※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。

### 1.3.6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。

### 1.3.7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

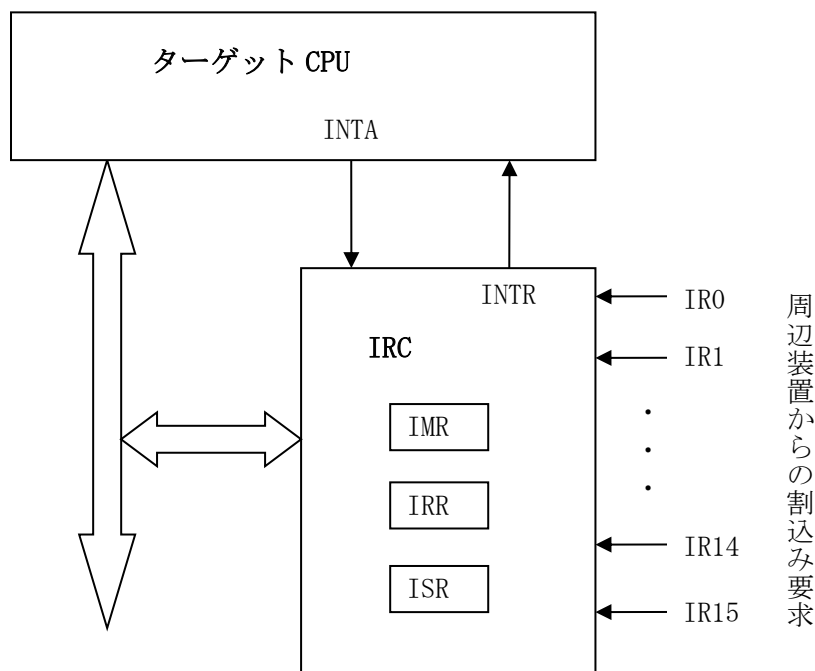
だし、IRC は 16 レベルを扱えて、8259A に即していえばエッジトリガモード、固定優先度、指定 EOI (EndOfInerrupt) コマンドモードと同等で、カスケード接続はないものとします。割り込みレベル 0 が最高優先度とします。

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:周辺装置からの割り込み要求
INTA	入力	INTERRUPT ACKNOWLEDGE:CPU による割り込みアクノリッジサイクルで割り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、入出力は割り込みコントローラから見た入力、出力です。

CPU との接続は以下ようになります。



IRC は次の 3 つの 16 ビットレジスタで割り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。(CPU はアドレスバス、データバスを使ってこれらのレジスタにアクセスします)

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

次に割り込み制御手順を整理します。

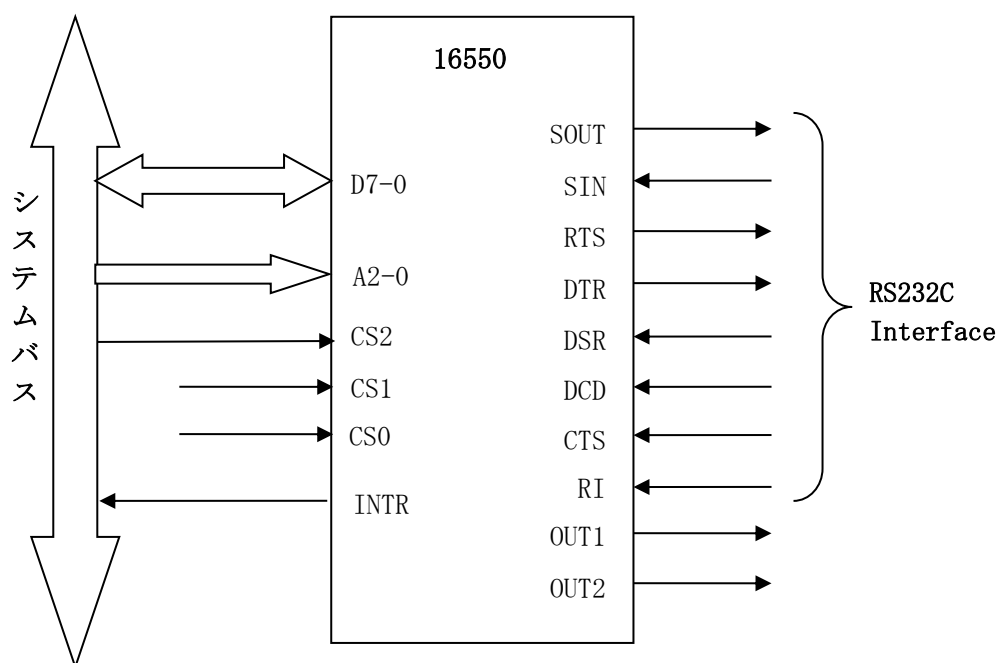
- ① IR0-15 に割り込み要求が起きると対応する IRR のビットをセットする。
- ② IRC はマスクされていない IR が起きると CPU へ INTR を送る。
- ③ CPU は INTR を検知すると INTA 信号で応答する。

- 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 はベクタ番号からベクタテーブル内の割込みハンドラアドレスを取り出し、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。

### 1.3.8 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 IR <sub>n</sub> 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	出力	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のレジスタ一覧を示します。

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または奇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トします。送信FIFOにあるデータは1バイトずつ取り出し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コマンドで設定します。

[7.4.1 init <割込みレベル>ne <チップ種別>s](#)を参照。

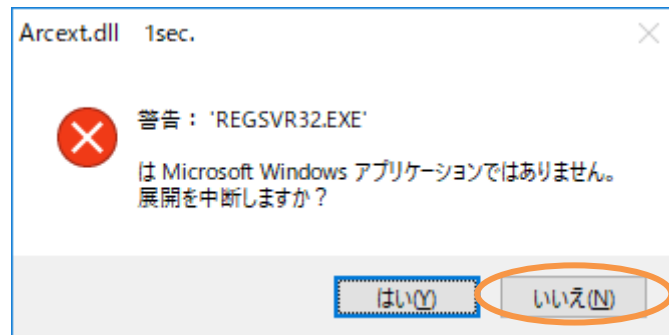


## 2 使用方法

### 2.1 インストール

ここでは Cmtoy-300.zip を Windows10 の **e:\¥cmtoy-300** 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-300¥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

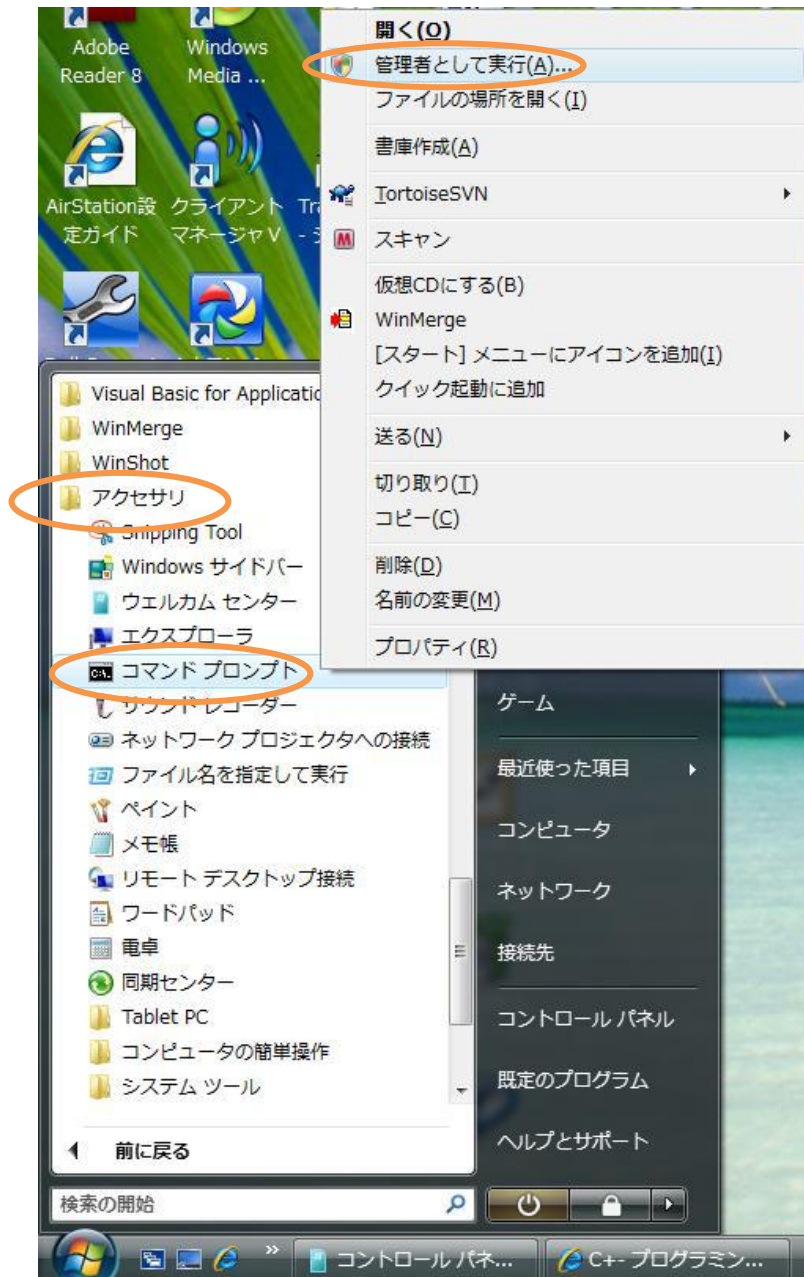
これらの ActiveX コントロールの登録を解除したい場合は、付属の uninstall.bat を実行してください。

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。

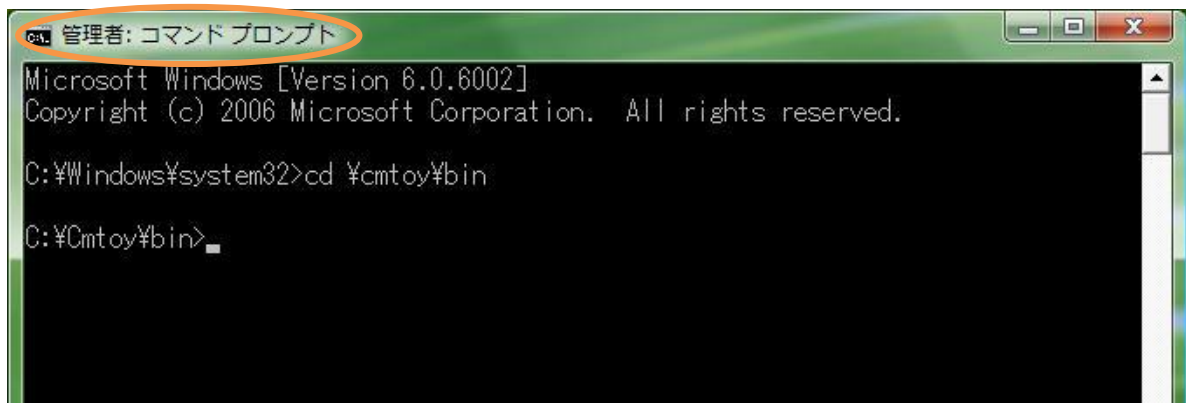
### 2.1.1 Windows Vista, Windows 7 でのインストール

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ーが表示されます。

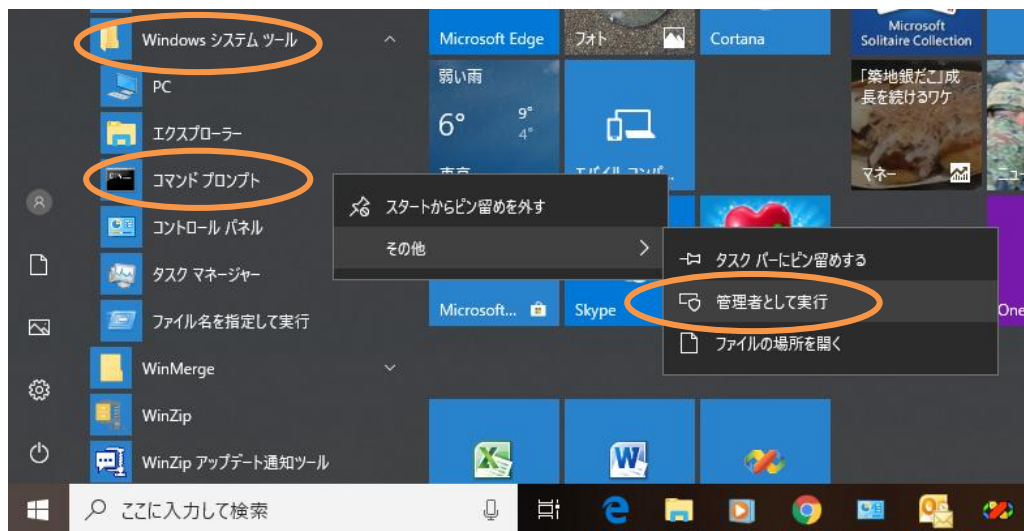


- ④ここで、「管理者として実行(A)...」をクリックすると、コマンドプロンプトが開きます。タイトルバーに「管理者：コマンドプロンプト」と表示されることを確認してください。



### 2.1.2 Windows10 でコマンドプロンプトを起動するには

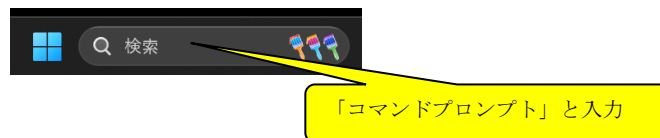
Windows10 では、コマンドプロンプトは「Windows システムツール」の中にあります。ここから管理者モードで起動してください。以下のように選択します。



### 2.1.3 Windows11 でコマンドプロンプトを起動するには

Windows11 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。

タスクバーの検索ボックスに「コマンドプロンプト」と入力します。



以下のウインドウが現れるので、右側の「管理者として実行」からコマンドプロンプトを開きます。



## 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。各オプションパラメータは空白で区切ります。

```
Cmtoy [-c<作業ディレクトリ>] [-p<ポート番号>] [-e<editor>] [-D<name>[=<value>]]
```

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。  |
| <editor>   | テキストエディタの実行ファイル (*.exe) を指すフルパス名                       |
| <name>     | コマンドラインで使用できる変数型マクロ名                                   |
| <value>    | 変数型マクロに置き換える文字列。空白を含む場合は 2 重引用符” で囲む。                  |

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、見つけ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。

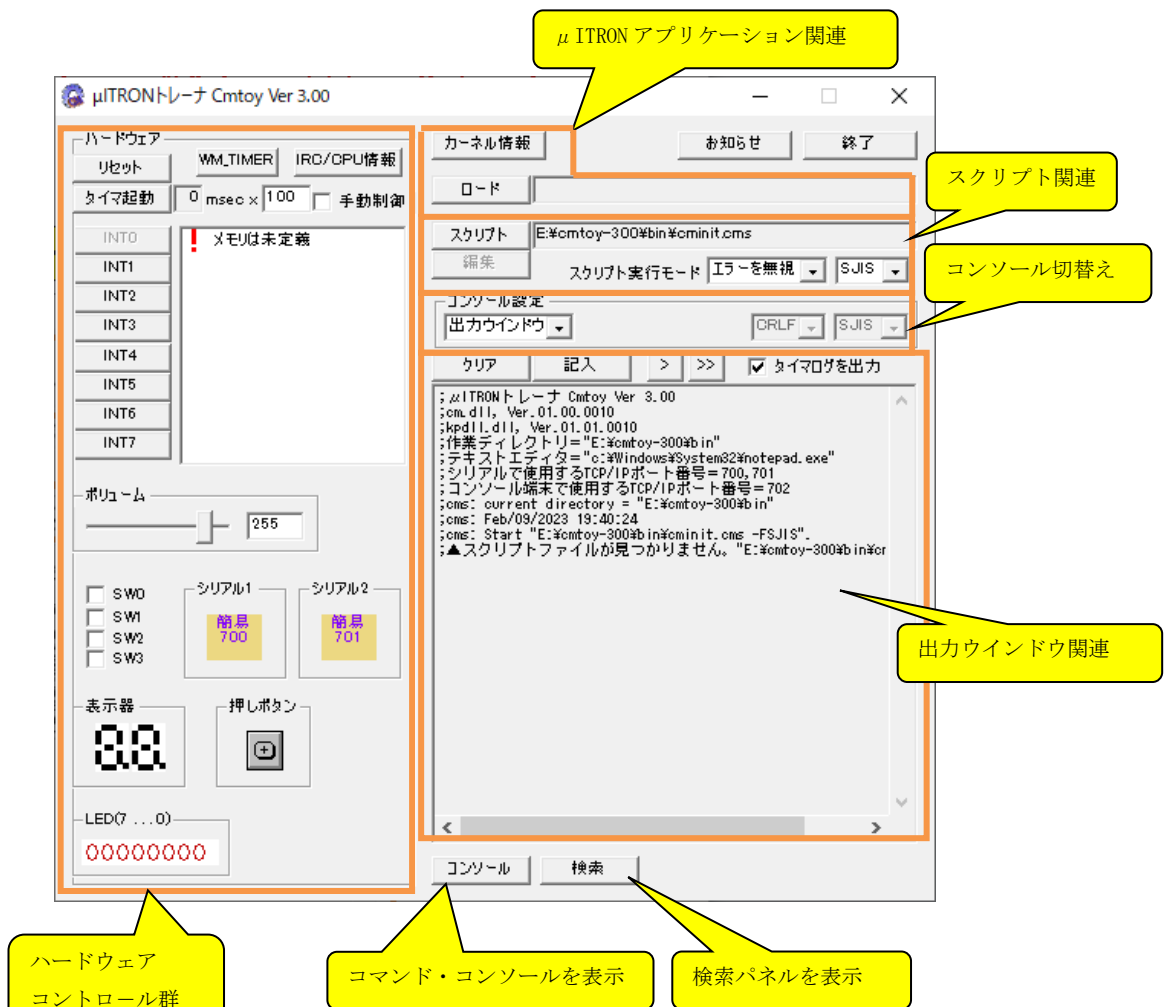
Cmtoy が正常に起動した場合、出力ウインドウには以下のメッセージを表示します。

```
; μ ITRON トレーナ Cmtoy Ver 3.00
```

```
;cm.dll : Ver.01.00.0010
;kpdll.dll : Ver.01.01.0010
;作業ディレクトリ="E:¥cmtoy-300¥bin"
;テキストエディタ="c:¥Windows¥System32¥notepad.exe"
;シリアルで使用する TCP/IP ポート番号=700,701
;コンソール端末で使用する TCP/IP ポート番号=702
;cms: current directory = "E:¥cmtoy-300¥bin"
;cms: May/23/2022 08:10:28
;cms: Start "E:¥cmtoy-300¥bin¥cminit.cms -fSJIS".
;▲スクリプトファイルが見つかりません。"E:¥cmtoy-300¥bin¥cminit.cms"
```

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 のウィンドウは表示されません。

Cmtoy を起動すると以下のウィンドウが表示されます。GUI コンポーネントの概要を以下に示します。



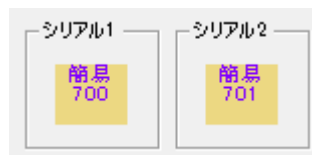
## 2.2.1 serial.ocx で使用する TCP/IP ポート番号を変更する

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。さらに TCP/IP 端末をコンソールとして使うためにポート番号を 1 つ割り当てます。これらは serial.ocx により制御します。

デフォルトのポート番号は、

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

TCP/IP 端末      TCP/IP ポート 702      V3.00 で追加した機能  
と割り当てます。  
Cmtoy 起動時の serial.ocx のアイコンには以下のようにポート番号が表示されます。



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 -p オプションを指定して 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
TCP/IP 端末	TCP/IP ポート 48559

となります。

シリアル 1、2 については「[5.8 簡易シリアル制御関数](#)」と「[5.9 16550 相当のシリアル制御関数](#)」  
を TCP/IP 端末については「[2.16.2 TCP/IP 端末](#)」を参照してください。

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

```
選択管理者: コマンドプロンプト
C:\WINDOWS\system32>e:
E:\>cd cmtoy-300\bin
E:\cmtoy-300\bin>install
E:\cmtoy-300\bin>regsvr32 -s -c push.ocx
E:\cmtoy-300\bin>regsvr32 -s -c segled.ocx
E:\cmtoy-300\bin>regsvr32 -s -c serial.ocx
E:\cmtoy-300\bin>regsvr32 -s -c led.ocx
E:\cmtoy-300\bin>cmtoy -p48667
E:\cmtoy-300\bin>
```

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  
E:\cmtoy-300\Cmtoy.exe -p48557

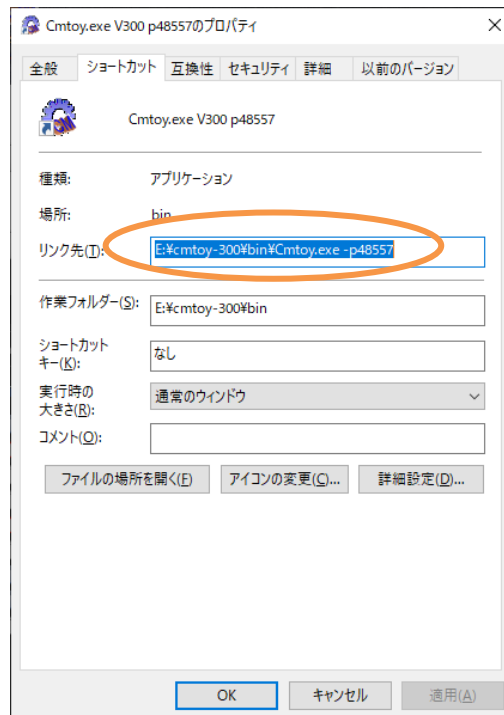
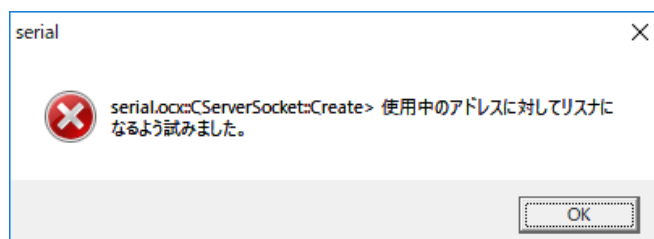


図 2-3 ショートカットのリンク先指定

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ができます。

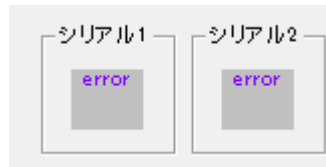


- ※ TCP/IP のポート番号はすでに使用方法が決まっているものがあります。ポート番号については以下の The Internet Assigned Numbers Authority (IANA) を参照してください。  
<http://www.iana.org/assignments/port-numbers>
- ※ シリアルコントロールとポート番号の使い方は、「[10.2.4 ハイパーターミナルの設定方法](#)」、「[10.2.5 PuTTY の設定方法](#)」を参考にしてください。
- ※ Cmtoy 起動時に、ポート番号が使用中であれば以下のダイアログが表示されます。



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 serial.ocx のアイコンには”error”と表示がでます。

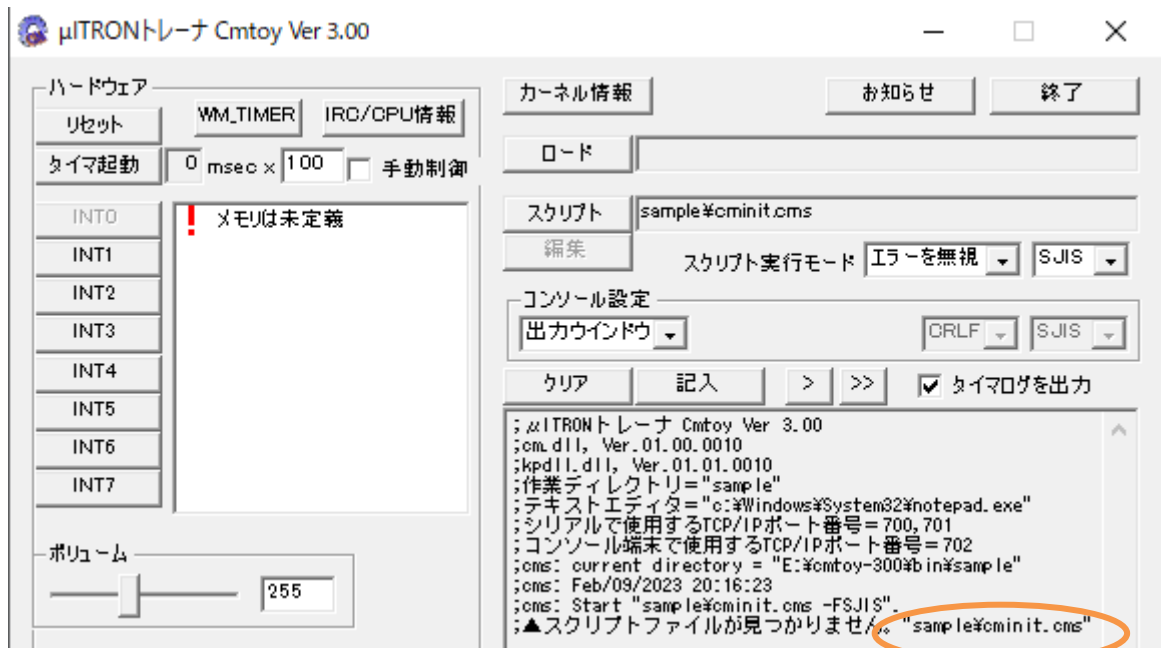




## 2.2.2 作業ディレクトリを変更する

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 -c オプションを指定して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。



; ▲スクリプトファイルが見つかりません。"sample¥cminit.cms"

もちろん、ショートカットからも同様のことができます。

## 2.2.3 テキスト・エディターを変更する

「編集」ボタンをクリックすると、実行済みのスクリプトファイルをテキスト・エディターで開くことができます。





デフォルトではWindowsの「メモ帳」を使って開きますが、-e オプションでエディターを変更できます。例えば、TeraPadを使いたい場合は以下のように指定します。

```
-e"c:\Program Files (x86)\TeraPad\TeraPad.exe"
```

## 2.2.4変数型マクロを定義する

変数型マクロについては「[6.1.4 前処理 \(プリプロセス\)](#)」を参照してください。

-D オプションで変数型マクロを追加できます。-D オプションの構文を以下に示します。

```
-D<name> [=<value>]
```

<name>            変数型マクロ名

<value>           変数型マクロを置換する文字列

空白を含むことはできません。<value>に空白を含めたい場合は、2重引用符"で囲みます。以下に例を挙げます。

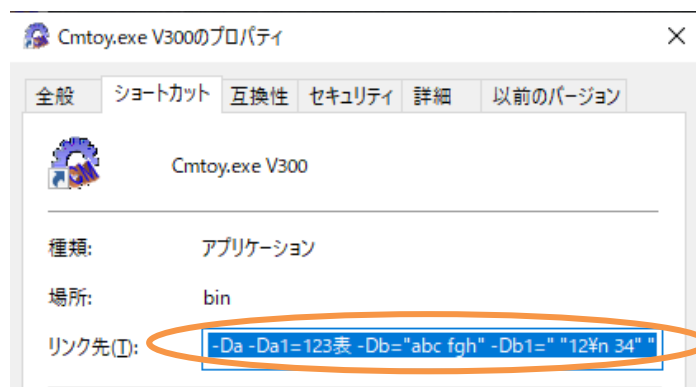
```
-Da1
```

```
-Da2=1234
```

```
-Da3="this is sample.%n"
```

起動時の-D オプションは20個まで指定できます。例えば以下の指定をCmtoyのショートカットに加えて、起動すると

```
Cmtoy -D -Da -Da1=123表 -Db="abc fgh" -Db1=" "12%n 34" "
```



出力ウィンドウに評価結果が表示されます。

```

;μITRONトレーナ Cmtoy Ver 3.00
;cm.dll, Ver.01.00.0010
;kpdll.dll, Ver.01.01.0010
;作業ディレクトリ="E:\cmtoy-300\bin"
;テキストエディタ="c:\windows\system32\notepad.exe"
;シリアルで使用するTCP/IPポート番号=700,701
;コンソール端末で使用するTCP/IPポート番号=702
;起動時の-Dオプション
1 : -D          ;▲no <identifier>
2 : -Da        ;ok
3 : -Da1=123表  ;ok
4 : -Db="abc fgh" ;ok
5 : -Db1=" 12¥n 34" " ;ok
;cms. current directory = E:\cmtoy-300\bin
;cms: Feb/09/2023 20:24:47
;cms: Start "E:\cmtoy-300\bin\cminit.cms -FSJIS".
;▲スクリプトファイルが見つかりません。"E:\cmtoy-300\bin\cr

```

1 の-D には<name>部分がないので「;▲no <identifier>」とエラーメッセージがついています。このとき変数型マクロの値は以下の通りです。（[define コマンド](#)で確認できます。）

```

a      =
a1     = 123 表
b      = abc fgh
b1     = "12¥n 34"

```

## 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 μ ITRON 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。

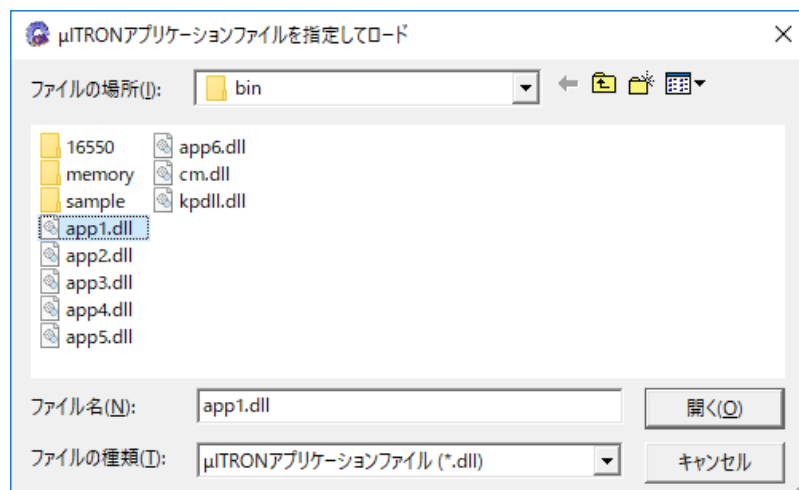


図 2-4 アプリケーションファイルの指定

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。出力ウィンドウには以下のメッセージを表示します。

```

> Load "E:\cmtoy-300¥bin¥app1.dll"
kpdll: AttachItronApplication(00b61030H) が見つかりました.

```

; アプリケーションプログラム (E:\cmttoy-300\bin\app1.dll) をロードしました。

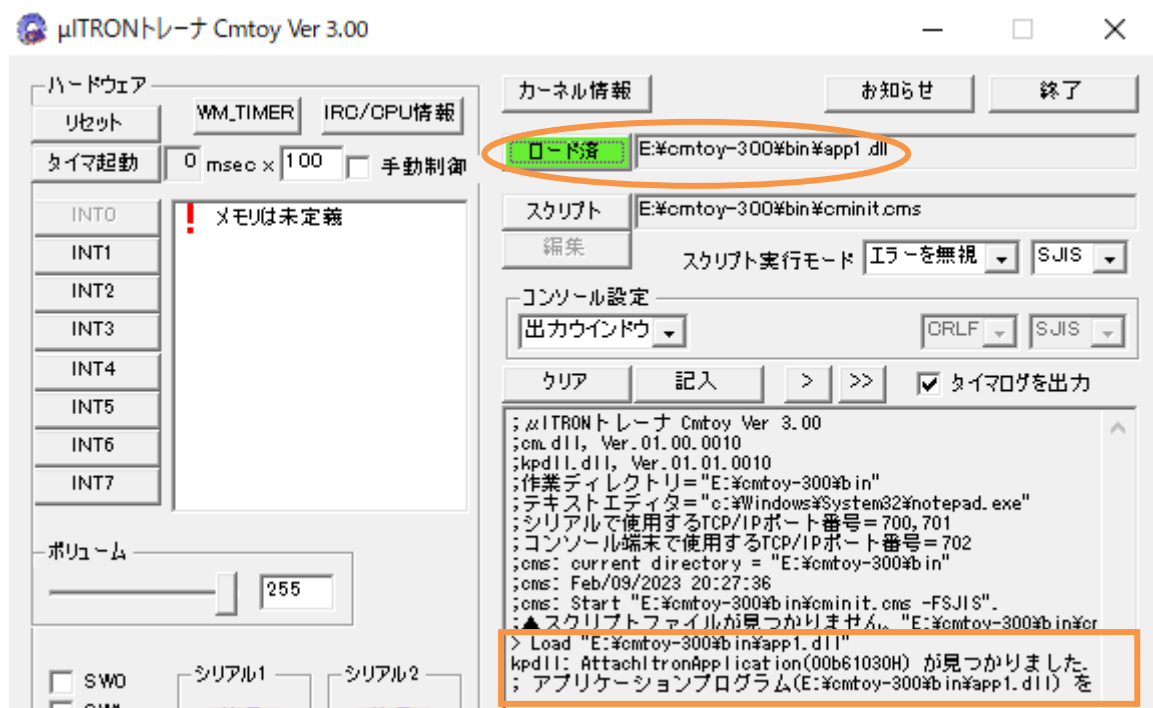


図 2-5 アプリケーションファイルのロード後

「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ィ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）。

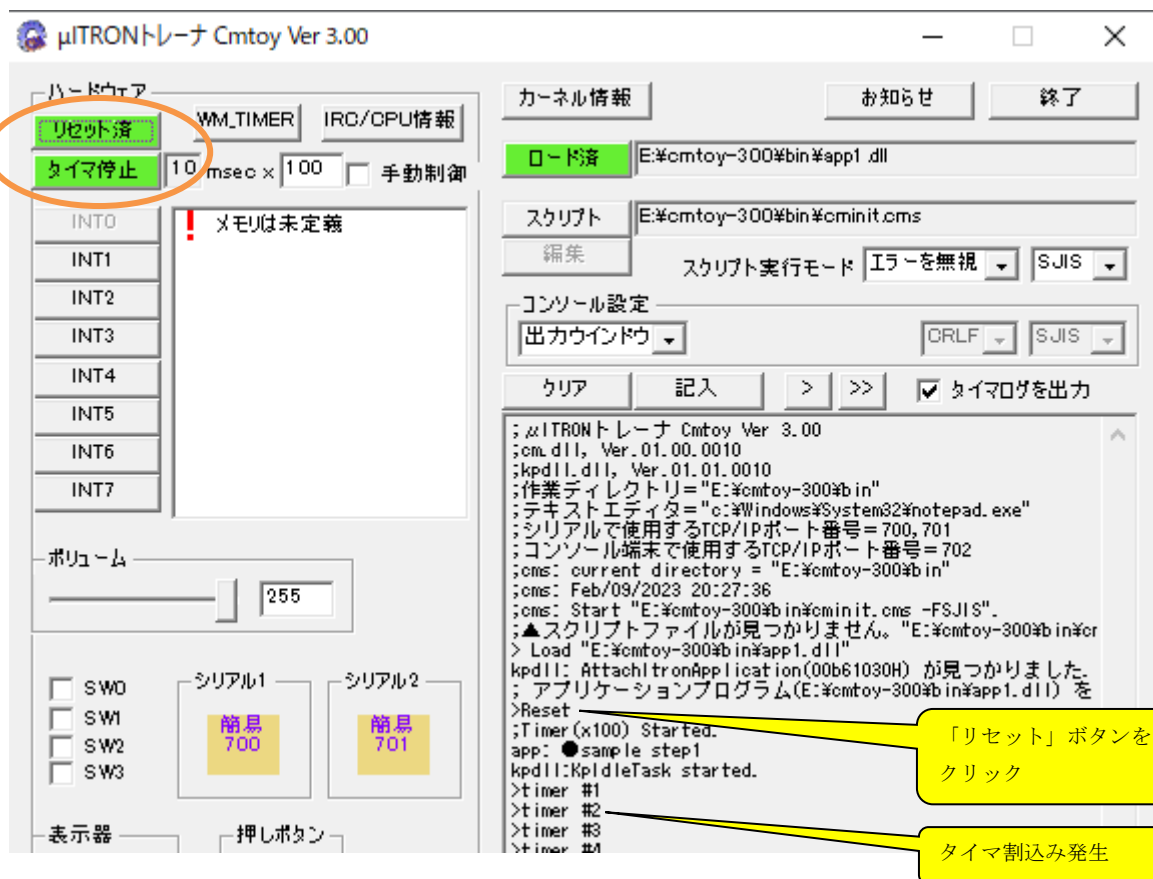
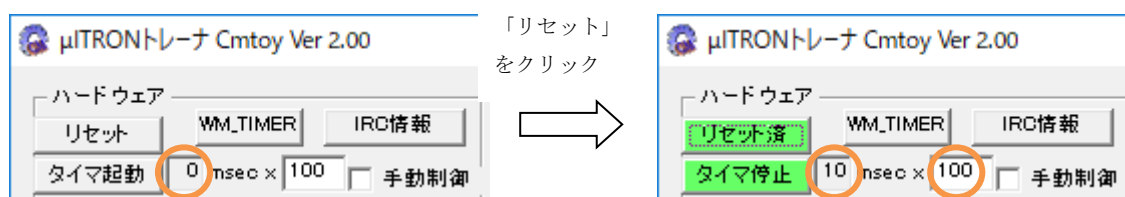


図 2-6 ターゲットプログラムが実行開始

## 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。

※このタイマはそれほど正確ではありません。時間計測には Windows の WM\_TIMER イベントを使っているため Windows のプロセス、スレッドスケジューリング規則（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。



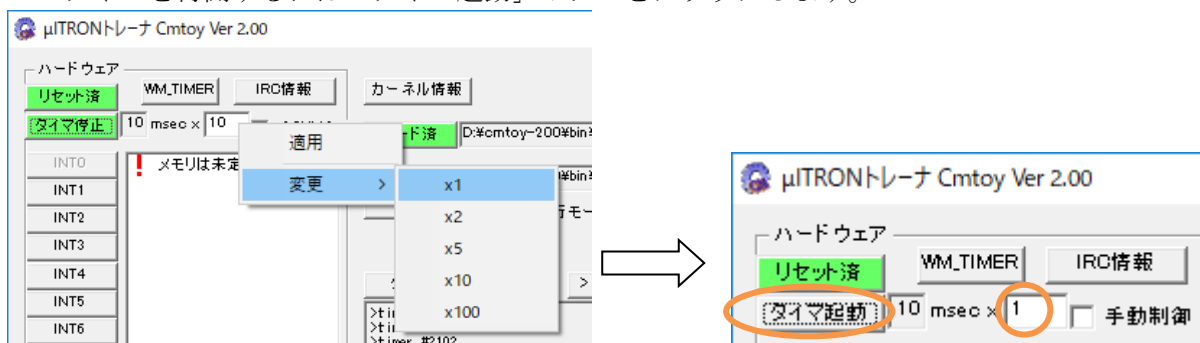
インターバルタイマは定期的にレベル 0 の外部割り込みを発生します。GUI の「タイマ起動」ボタンの表示が「タイマ停止」に変わり、背景色も変わります。

μITRON カーネルはこのレベル 0 の割り込み回数で内部時刻をカウントします。Cmtoy では実際の割り込み間隔は、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り込み間隔は  $10\text{ms} \times 100 = 1000\text{ms} = 1\text{秒}$  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、メニューが出たら「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。

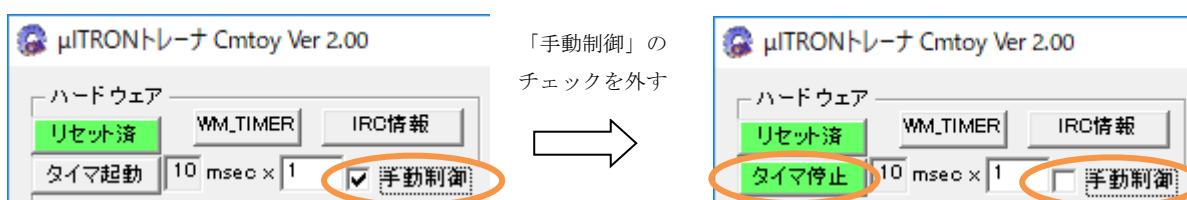
インターバルタイマを再開するには「タイマ起動」ボタンをクリックします。



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ーから「変更」→「x1」をクリックします。これで  $10\text{ms} \times 1 = 10\text{ms}$  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で1回ず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。

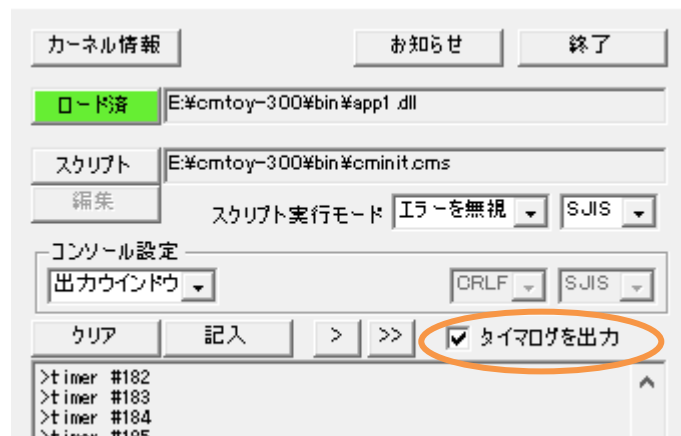


Cmyoy ではインターバルタイマの時間間隔は Windows の WM\_TIMER イベントでシミュレートしているのでそれほど正確ではありません。そのため、「WM\_TIMER」ボタンは WM\_TIMER のイベントの保守用トレースを表示します。

インターバルタイマの割込みが発生すると出力ウィ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、出力ウィンドウの「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。このとき出力ウィンドウには、

```
>Int 1 #1
```

の行が出力されます。

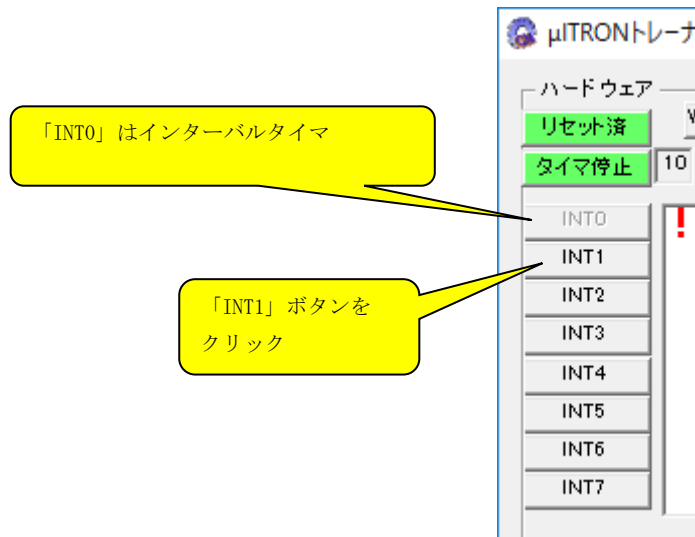


図 2-7 INT1 をクリック

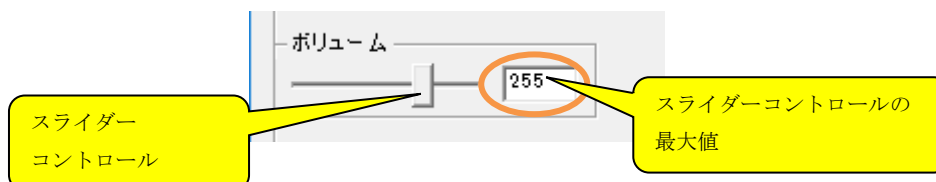
Cmtoy では16レベルの外部割込みを扱いますが、GUIから操作できるのはレベル0～7です。レベル0の操作は「[2.4 インターバルタイマの操作](#)」を参照してください。

Cmtoy のGUI ウィンドウの「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ィ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

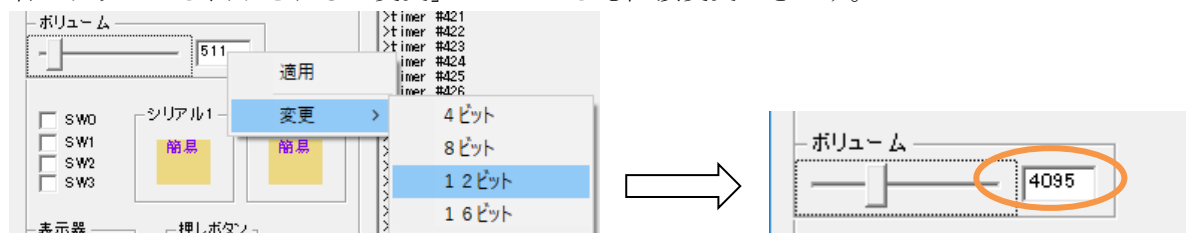
以下のボリュームはA/Dコンバータをシミュレートしています。



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」メニューからも直接変更できます。

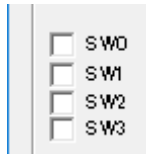


選択したビット数と最大値の関係は以下のようになります。

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 (Dual In-line Package) 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

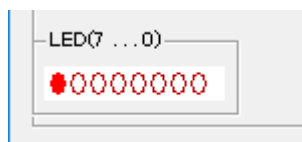


DOWN (マウスの左ボタンを押した状態)



## 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。

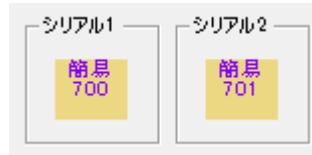


## 2.10 シリアルポート

2 個のシリアルポートをシミュレートします。

以下のように使用する TCP/IP ポート番号が表示されます。





## 2.11 $\mu$ ITRON 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  $\mu$  ITRON カーネル (kpd11.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 kpd11.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。

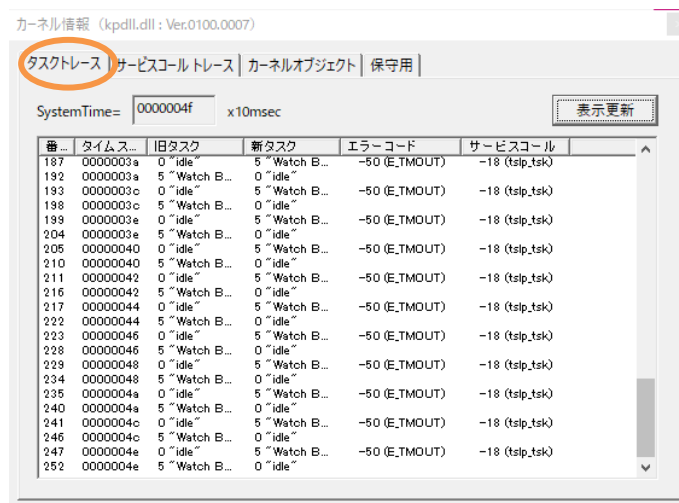


図 2-8  $\mu$  ITRON のタスクトレース情報

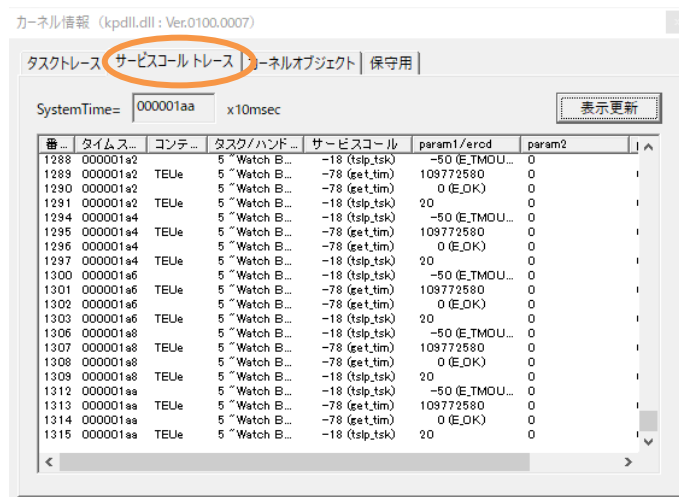


図 2-9  $\mu$  ITRON のサービスコールトレース情報



図 2-10 μITRON のオブジェクト情報

## 2. 12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の間、パラメータ間は 1 つ以上の空白、タブで区切ります。

### 2.12.1 コマンド・コンソール

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。

このウインドウは Cmtoy 起動時には存在しません。最初に「コンソール」ボタンをクリックした時点で作られます。右上の×ボタンをクリックすると非表示になります。再度表示するには「コンソール」ボタンをクリックします。

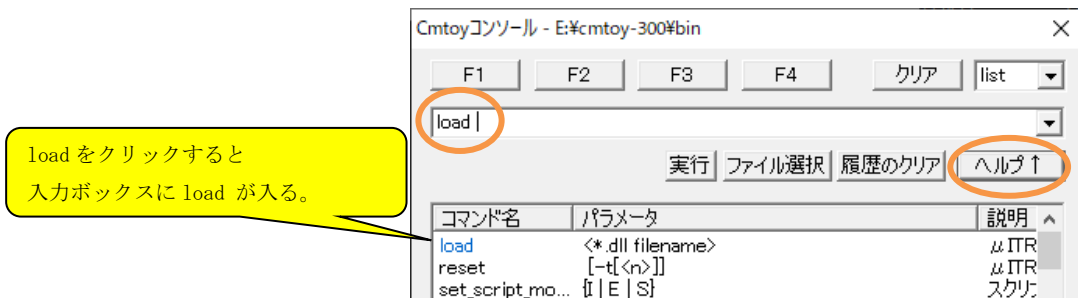
このコマンド入力と「出力ウインドウ」がデフォルトコンソールとなります。「出力ウインドウ」

については [2.14 出力ウインドウ](#) で説明します。

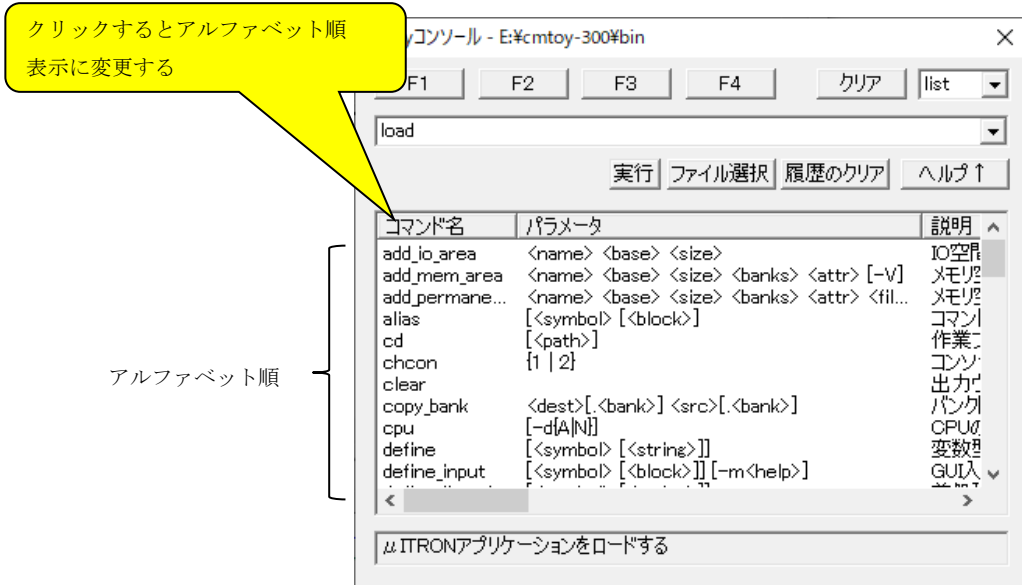
### 2.12.2 ヘルプ機能

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 load をクリックすると load が入力ボックスの先頭に入ります。

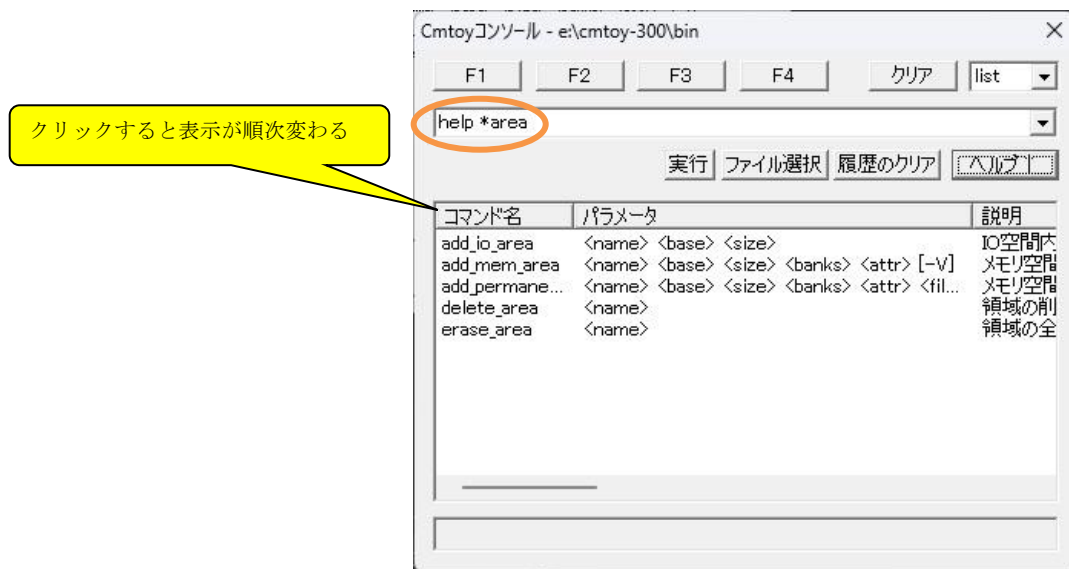


リストボックスの「コマンド名」をクリックするとコマンド名がアルファベット順に並びます。もう一度クリックすると元に戻ります。



各コマンドの詳しい説明は、「[7 コンソール・コマンド一覧](#)」を参照してください。

[help コマンド](#)を使用すると、コマンド一覧をフィルタ出来ます。例えば、メモリ領域を制御するためのコマンド一覧を得るには以下のように help コマンドを実行します。

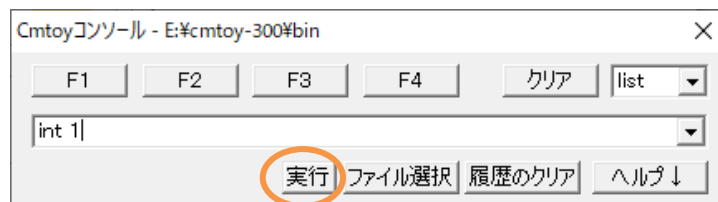


ここでリストボックスの「コマンド名」をクリックするとリストボックスの表示が順次切り替わります。

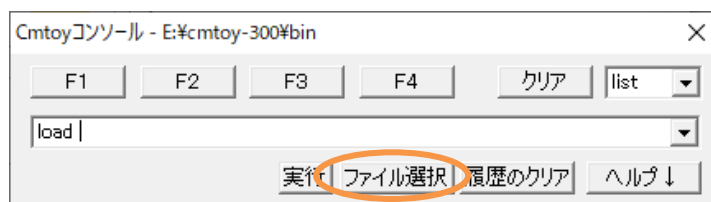
フィルタ後のコマンド名 → 初期状態 → アルファベット順のコマンド名

### 2.12.3 コマンドラインの実行

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。



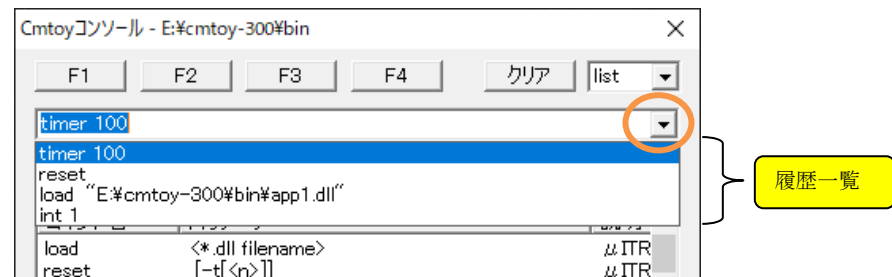
コマンドラインの中でファイル名を指定する箇所では「ファイル選択」ボタンをクリックするとファイルダイアログからファイルを指定することができます。例えば、load コマンドの次にファイル名を指定する場合キーボードからファイル名を入力してもいいですが、「ファイル選択」から指定することもできます。





「ファイル選択」からファイルを指定すると、フルパス名が2重引用符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。



#### 2.12.4 list コマンド実行

このコンボボックスから [list コマンド](#) を実行できます。また、コマンド／スクリプトで list コマンドを使用すると、このコンボボックスの表示も変わります。



#### 2.12.5 ショートカット・ボタン

F1～F4 のボタンにコマンドマクロ（一連のコマンド列）を登録することができます。登録には [alias コマンド](#) を使いコマンドマクロを定義して行います。「F1」ボタンをクリックするとコマンドマクロ\_F1 を実行します。同様に「F2」～「F4」ボタンをクリックするとコマンドマクロ\_F2～\_F4 を実行します。



スクリプトを実行する場合はスクリプトを実行するコマンドマクロを作成できます。  
 コマンドマクロについては「[6.1.5 コマンドマクロ](#)」を参照してください。

## 2.12.6clear コマンドを実行

「クリア」ボタンで出力ウインドウの表示を消去する [clear コマンド](#) を実行できます。



## 2.13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。「[6.2 スクリプト機能](#)」を参照してください。

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。  
 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＞ … ]] ;コメント<行末>

- ・ 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 (注釈) とみなします。

使用できるコマンドは、「[7 コンソール・コマンド一覧](#)」を参照してください。

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。

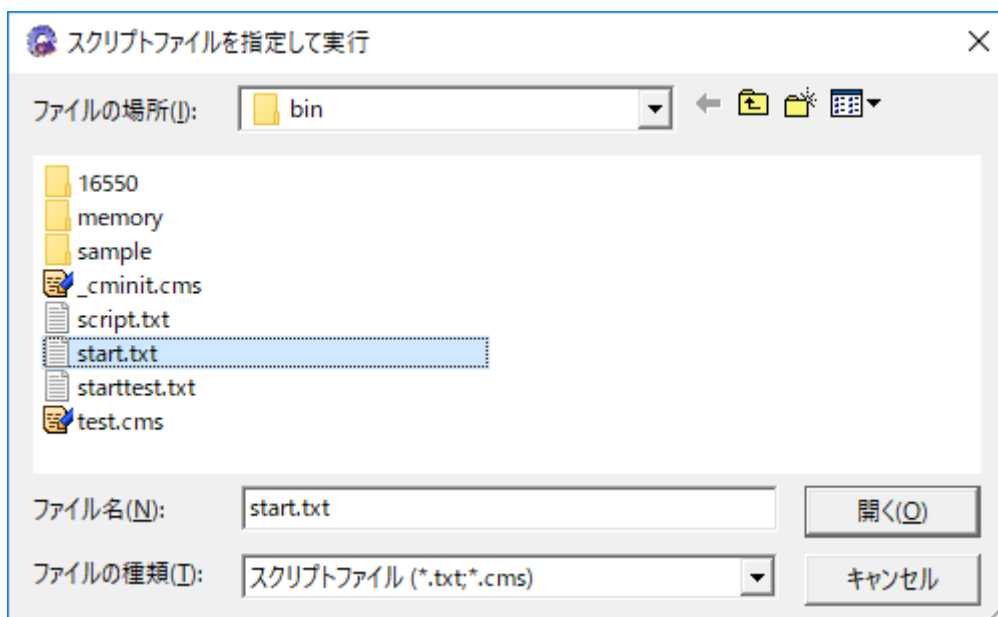


図 2-11 スクリプトファイルの指定

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ィ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 2-12 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。Cmtoy 起動時に指定する -e オプションが指定されている場合は、そのテキストエディターで開きます。

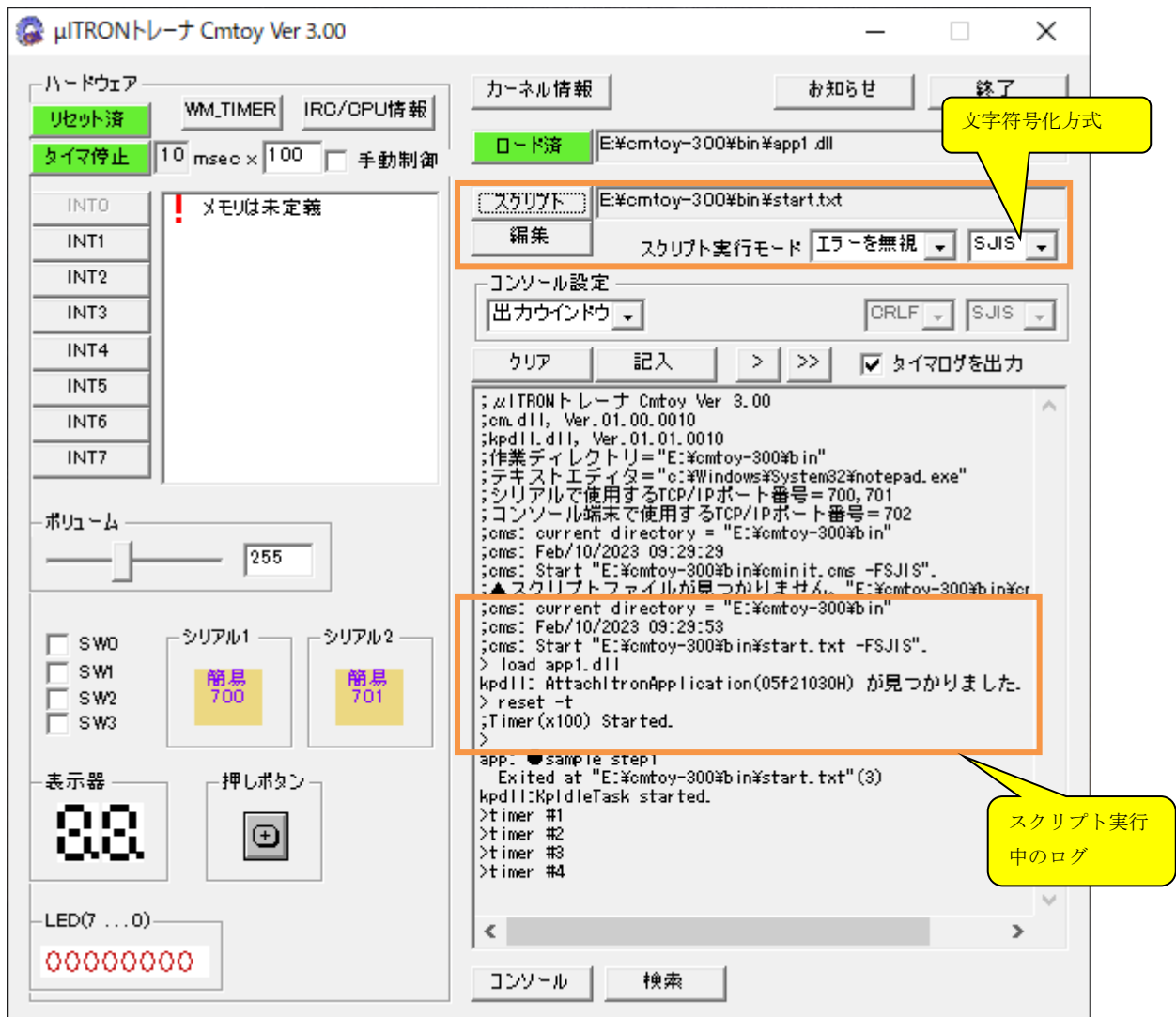


図 2-12 サンプルプログラムをスクリプトで実行

注) スクリプトから実行した [reset コマンド](#) では、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。

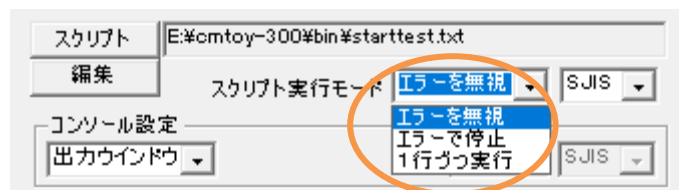


### 2.13.1 スクリプトの実行モード

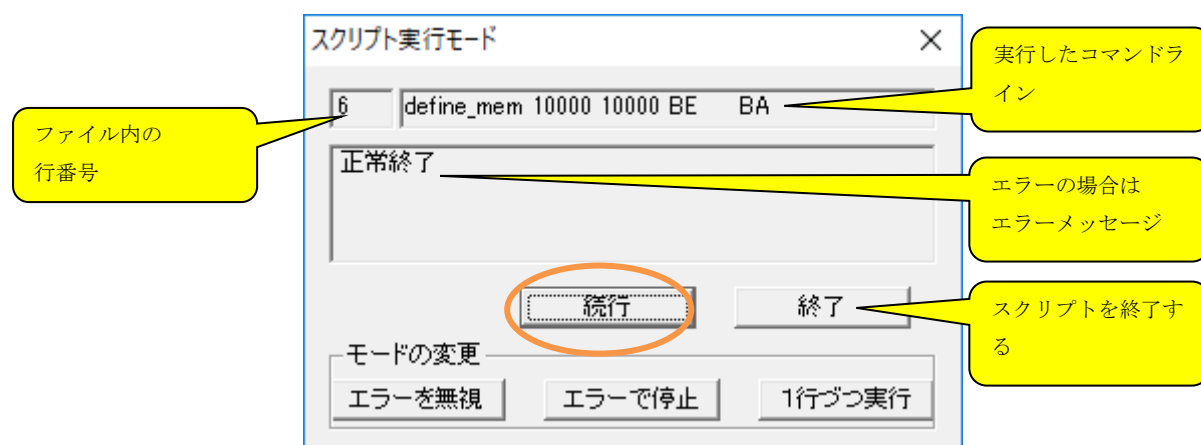
スクリプトの実行には3つのモードがあります。



- ① エラーを無視
- ② エラーで停止
- ③ 1行ずつ実行して停止



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。

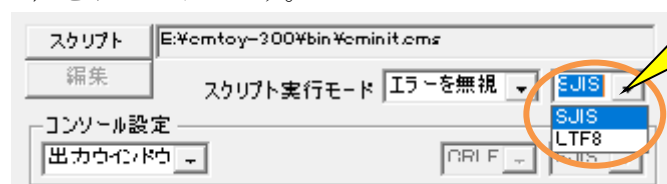


ここで「続行」ボタンをクリックすると、

- ・ 「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行ずつ実行」の場合は、次の行を実行したあと停止します。

## 2.13.2 スクリプトファイルの文字符号化方式

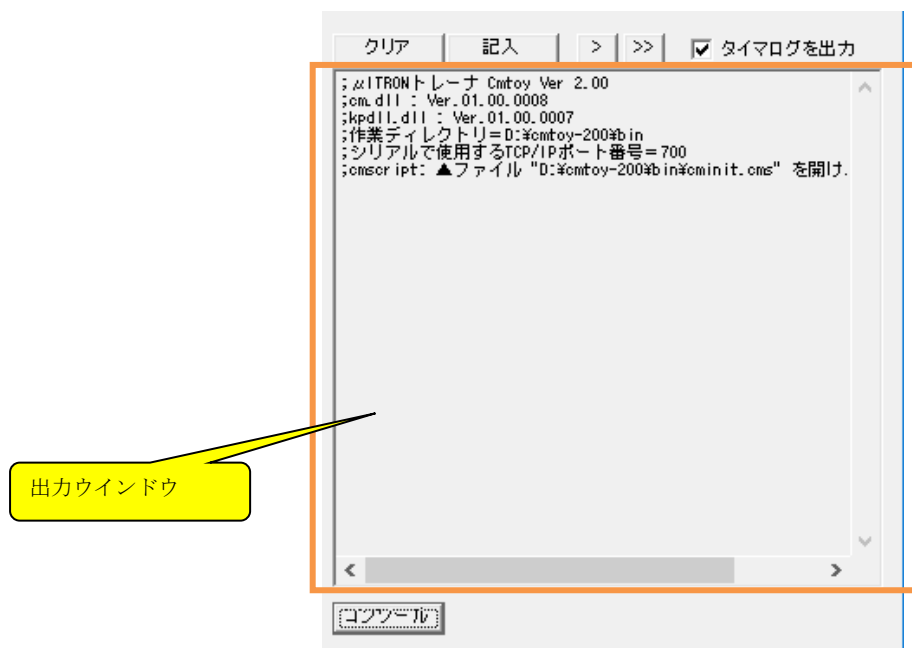
Cmtoy ではスクリプトファイルの文字符号化方式 (CES) としてシフト JIS と UTF8 (Unicode Transformation Format-8) をサポートします。



※CES は Character Encoding Scheme。

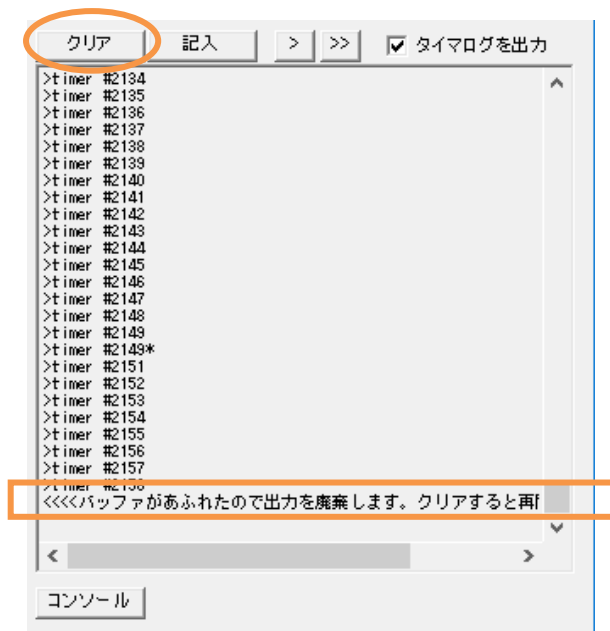
## 2.14 出力ウィンドウ

Cmtoy は様々な情報を出力ウィンドウに表示します。また、 $\mu$ ITRON アプリケーションも情報を出力ウィンドウに表示できます。



出力ウィ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。

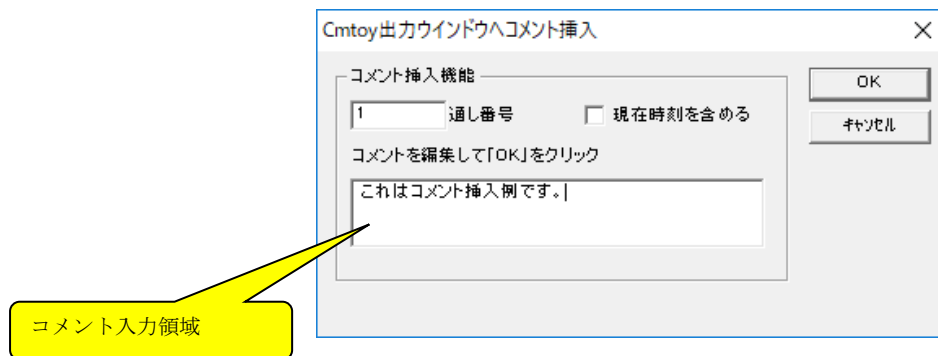
<<<<バッファがあふれたので出力を廃棄します。クリアすると再開します。>>>>



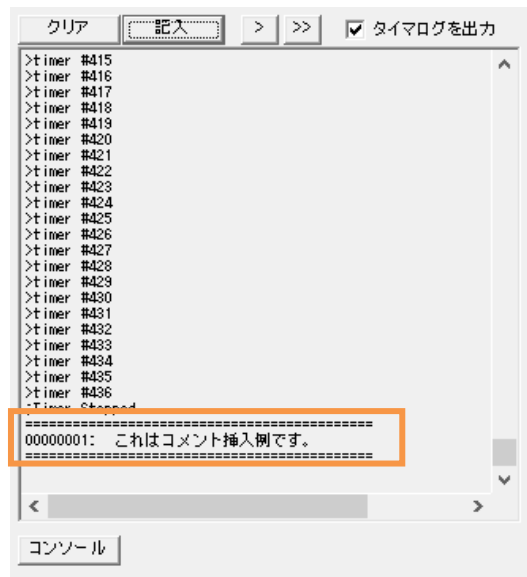
ここで「クリア」ボタンでバッファを空にすると表示を再開します。（[clear コマンド](#)を実行）

#### 2.14.1 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」とすると出力ウィ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。



## 2.14.2 ファイルへ保存

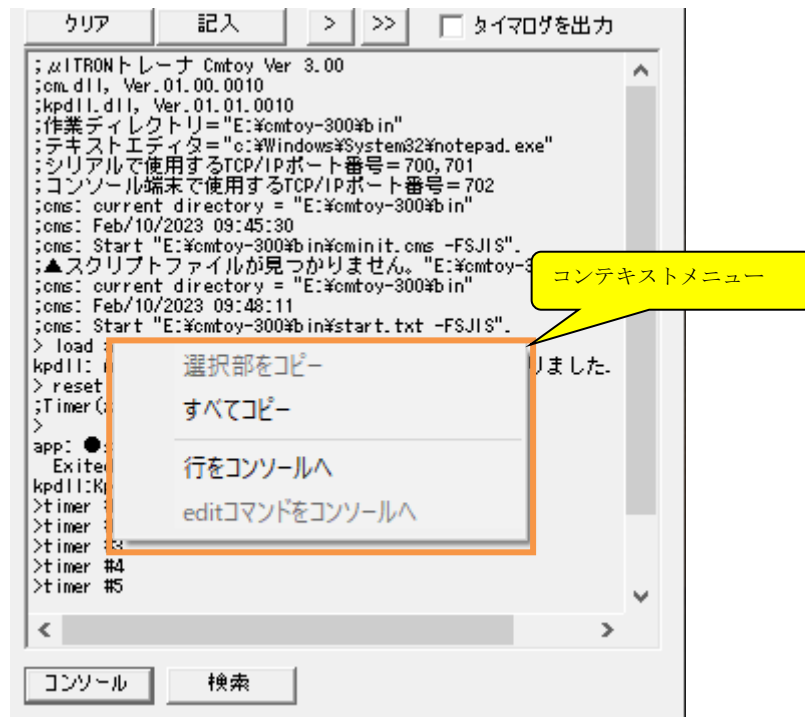
ボタン「>」と「>>」は出力ウインドウの内容をファイルに書き出すときに使います。「>」は出力ファイルを指定して上書き、「>>」は指定したファイルに追加します。

$\mu$ ITRON 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。[「5.3 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, ...);
```

## 2.14.3 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。



出力ウインドウはコマンド・コンソールの一部です。コマンド・コンソールについては「[2.12 コマンドラインによる操作](#)」を参照してください。

#### (1) 選択部をコピー

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。  
選択されたテキストがない場合はこのメニューは灰色となり選択できません。

#### (2) すべてコピー

出力ウインドウ内のすべてのテキストをクリップボードへコピーします。

#### (3) 行をコンソールへ

すでにコマンド・コンソールのウインドウが作られていれば（非表示であっても）このメニューを選択できます。コマンド・コンソールが作られていないとこのメニューは灰色となり選択できません。右クリックした行が「>」で始まる場合に、先頭の「>」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。

#### (4) edit コマンドをコンソールへ

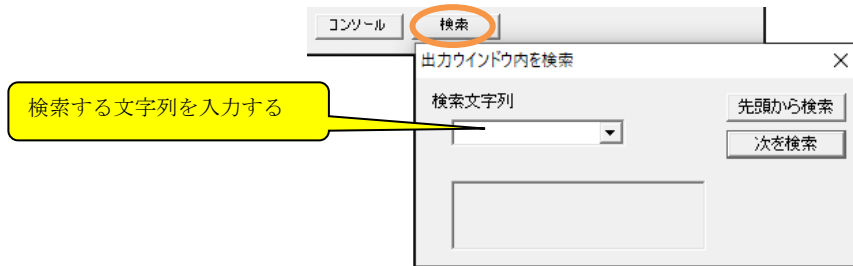
すでにコマンド・コンソールのウインドウが作られていて（非表示であっても）かつ選択されたテキストがある場合にこのメニューを選択できます。コマンド・コンソールの入力領域には以下の文字列が入力されます。

edit <選択されているテキスト>

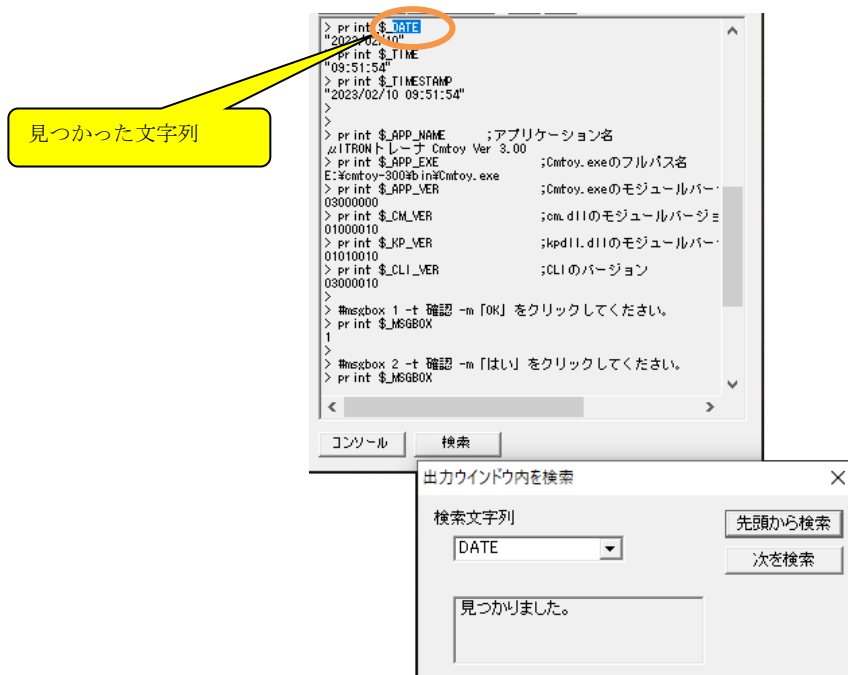
出力ウインドウ内のファイル名を選択してこのメニューを選択すればテキストエディタでそのファイルを開く準備ができます。

## 2.14.4 出力ウィンドウ内の検索

出力ウィンドウ内の文字列を検索します。「検索」ボタンをクリックすると検索パネルが表示されます。



検索して見つかった文字列があると表示される位置にスクロールします。



## 2.15 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 I/O、ポートマップド I/O 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、I/O 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 I/O など)
- ・I/O 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。「[5.11.3 ターゲットメモリを操作 \(アドレスを即値で使用する場合\)](#)」、「[5.11.4 ターゲットメモリを操作する \(構造体のメンバを使用する場合\)](#)」を参照。

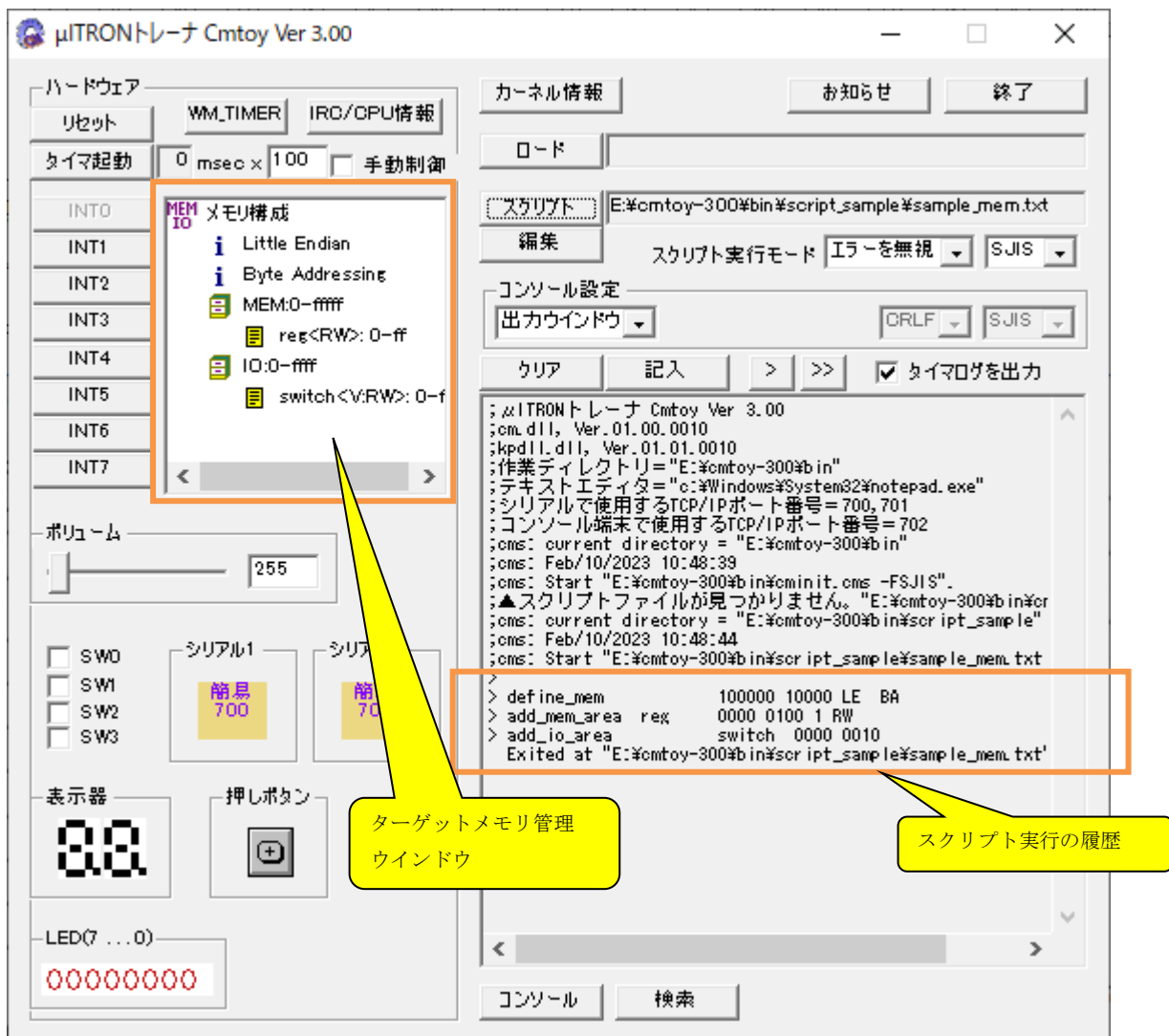
ターゲットメモリは、コンソール・コマンドを使って定義します。「[7.3 ターゲットメモリ操作](#)」

を参照してください。

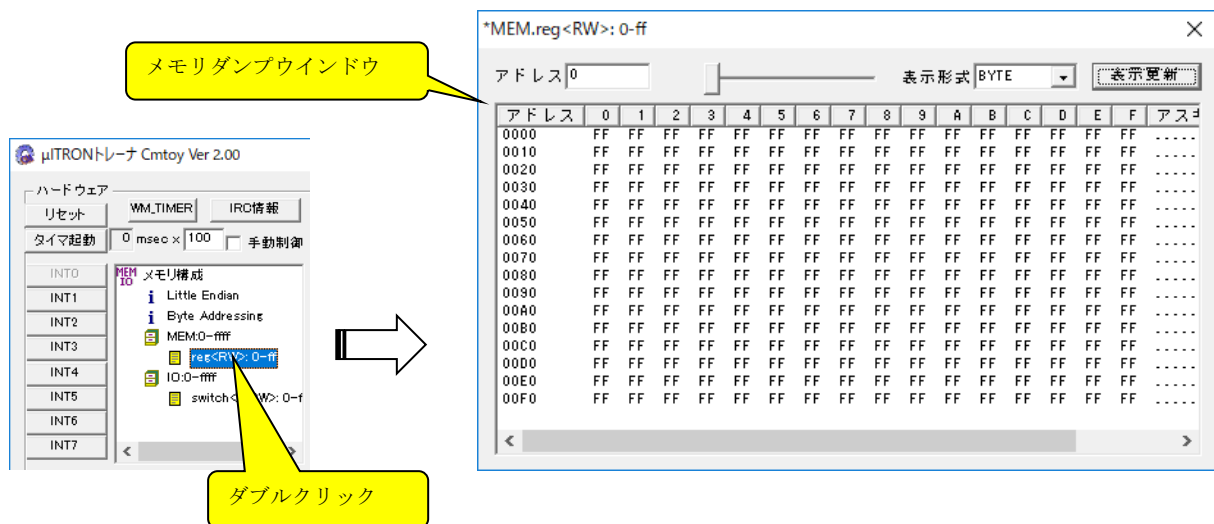
例えば、16 ビット CPU、バイトアドレッシング、リトルエンディアンの RAM アドレス 0x0000-0x00ff、IO ポート 0x0000-0x000f のターゲットメモリを定義する場合は、以下のコマンドを実行します。

```
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。

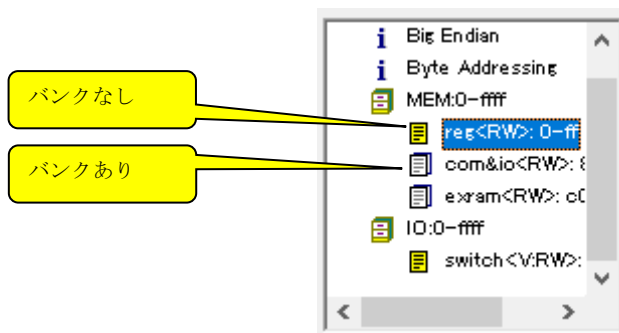


以下のようなバンク指定を含むコマンドを実行した場合の例を示します。

```

;          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

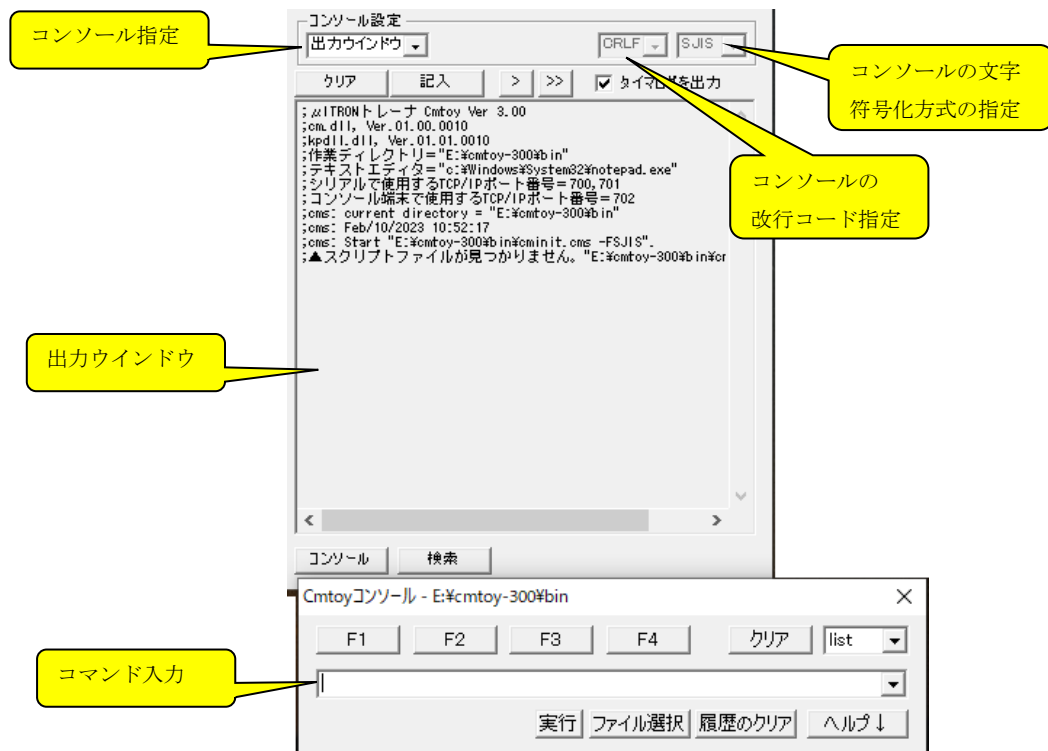
```



バンクあり／なしでアイコンが変わります。

## 2. 16コンソール端末

Cmtoy 起動時のデフォルトコンソールは、以下のようなコマンド入力ウィンドウと出力ウィンドウの構成となります。



コンソール設定で、以下の 2 つからコンソール端末を選びます。Cmtoy 起動時は「出力ウインドウ」となります。

- ・ 出力ウインドウ
- ・ TCP/IP 端末

### 2. 16. 1出力ウインドウ

Cmtoy は Windows のマルチバイト文字セット (MBCS)を採用した 32 ビットアプリケーションです。出力ウインドウは Windows アプリケーションの一部なので改行コードは CRLF で文字符号化方式はシフト JIS で変更できません。

### 2. 16. 2TCP/IP 端末

ドロップダウンリストから「TCP/IP 端末」を選択すると、serial.ocx のアイコンが表示され、改行コードと文字符号化方式が変更できるようになります。TCP/IP ポートのポート制御は serial.ocx が行います。



改行コードは以下の中から選べます。



- CRLF CR (キャリッジリターン) と LF (ラインフィード)
- CR CR (キャリッジリターン) のみ
- LF LF (ラインフィード) のみ

文字符号化方式は以下の中から選べます。

- SJIS シフト JIS
- UTF8 UTF8 (Unicode Transformation Format-8)

使用する TCP/IP ポート番号を変更する場合は、「[2.2.1 serial.ocx の使用する TCP/IP ポート番号を変更する](#)」を参照してください。

TCP/IP 端末アプリケーションとしてハイパーターミナルや PuTTY などが使えます。改行コードと文字符号化方式の設定は端末アプリケーションの設定と合わせる必要があります。

端末アプリケーションが Cmtoy に接続すると serial.ocx のアイコンの色が以下のように変わります。

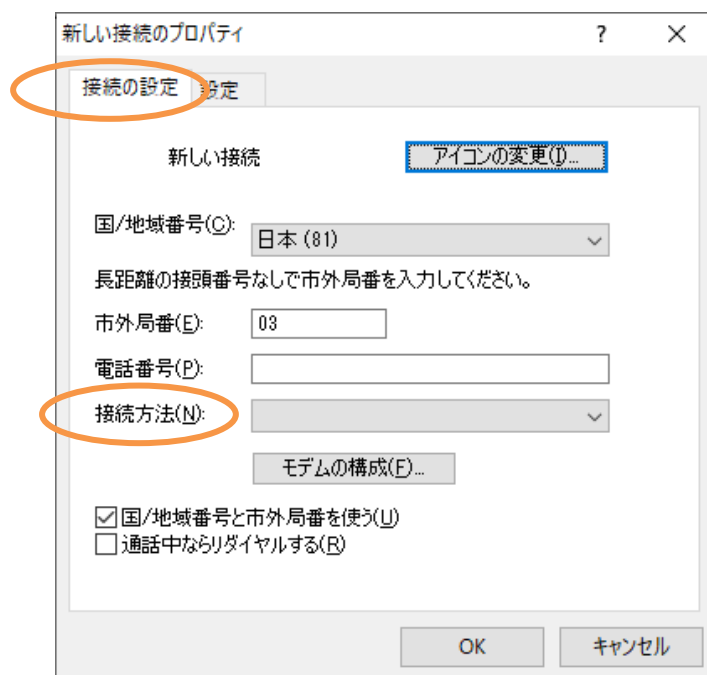


この例では TCP/IP ポート 702 を使って接続されています。

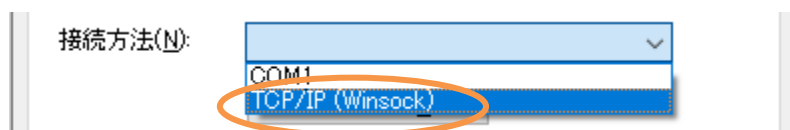
### (1) ハイパーターミナルの設定方法

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。

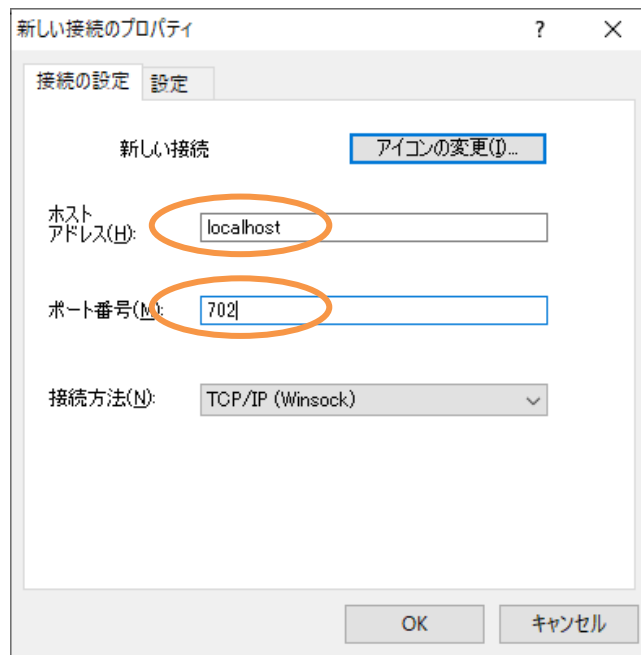
1. ハイパーターミナルを起動後、ファイルメニューの「プロパティ」を開きます。



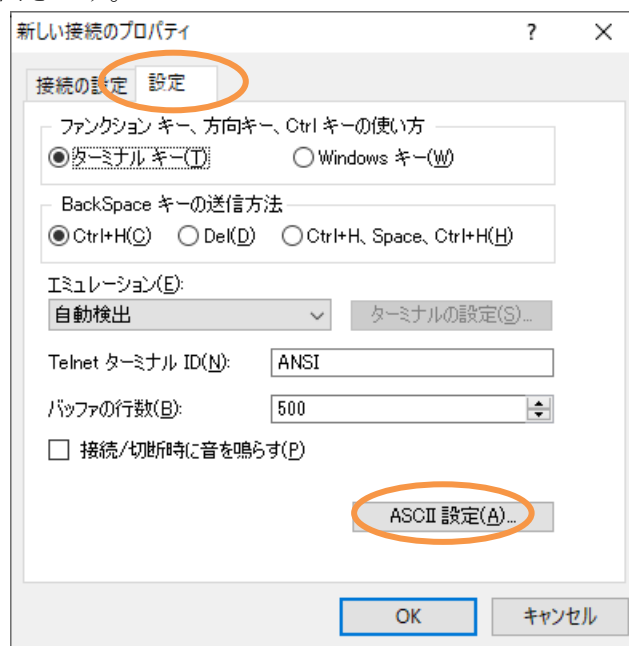
2. 次に「接続方法(N)」から「TCP/IP (Winsock)」を指定します。



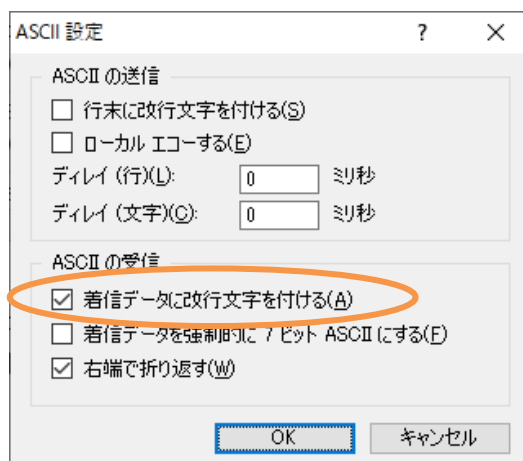
3. 「ホストアドレス(H)」を”localhost”、「ポート番号(M)」を 702 と指定します。



4. 次に「設定」タブを開きます。



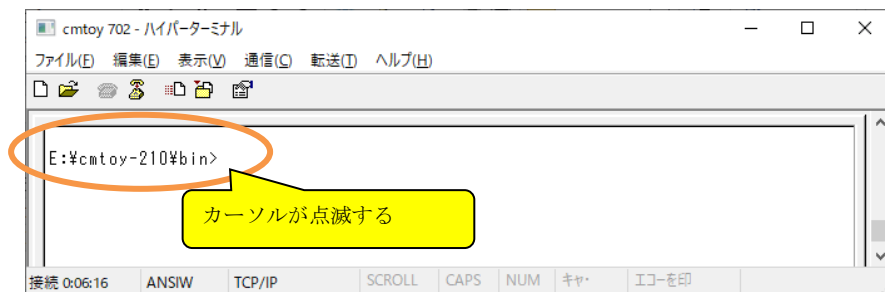
5. 、ここで「ASCII 設定 (A)」をクリックし、受信データに改行文字を付ける」をチェックする。



6. ハイパーターミナルが Cmtoy に接続できると、Cmtoy 側のアイコンの色が変わります。



7. ここでハイパーターミナルのウインドウ内をクリックして（キーボードの入力権を与えて）「ENTER」キーを押すと以下のようにプロンプト文字が表示されます。



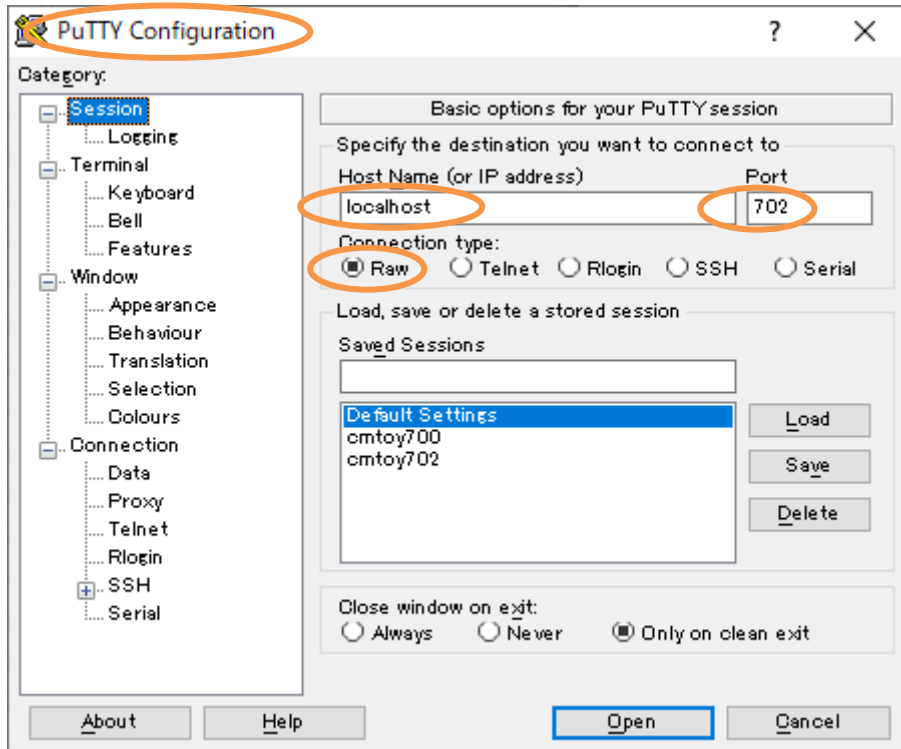
以後、キーボードからコマンドラインを入力し「ENTER」キーでコマンドを実行できます。キー入力  
はハイパーターミナル画面上に表示されます。コマンドを実行した結果もハイパーターミナル画面  
に表示されます。

## (2) PuTTY の設定方法

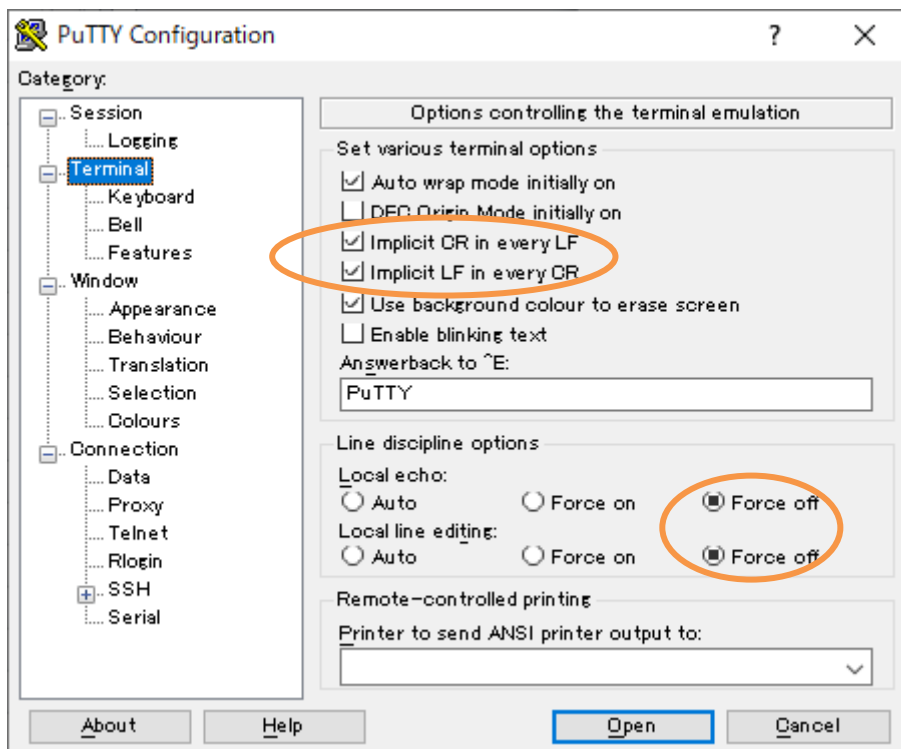
ハイパーターミナの代わりに PuTTY を使用する場合は以下のように設定してください。

PuTTY は/pʌtɪ/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。以下は Release 0.70 の設定画面です。

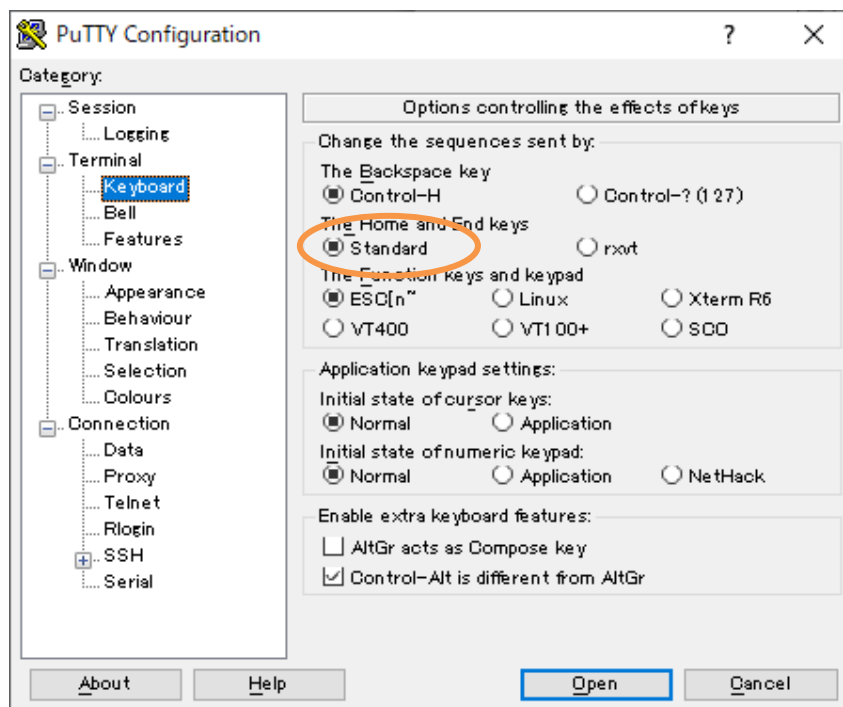
1. PuTTY を起動すると、「PuTTY Configuration」ウインドウが開きます。
2. まず Session の設定をします。



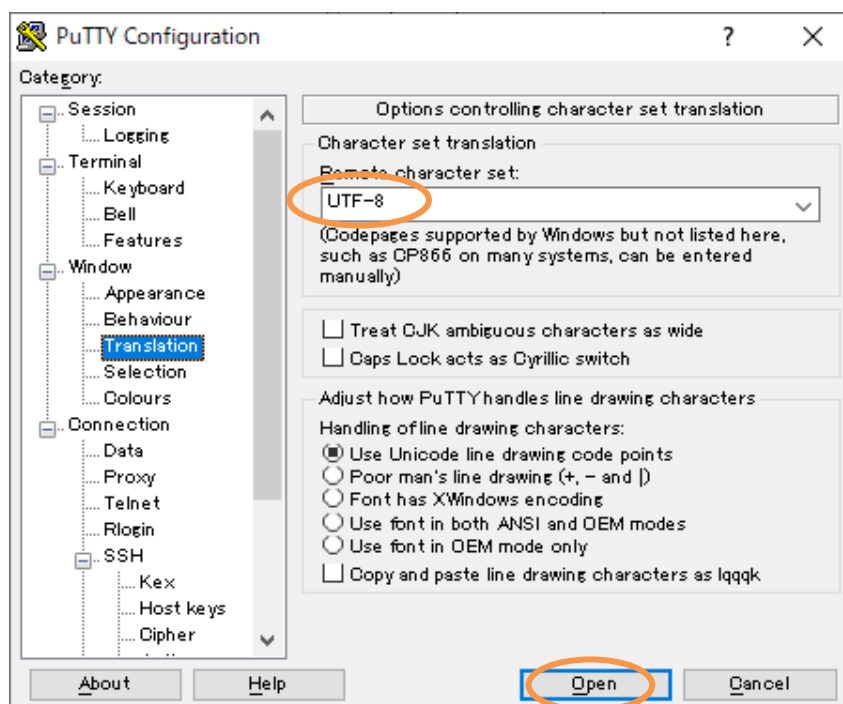
3. 次に Terminal の設定をします。



4. 次に Terminal/Keyboard の設定をします。



5. 最後に Window/Translation の Remote character set が UTF8 になっていることを確認します。

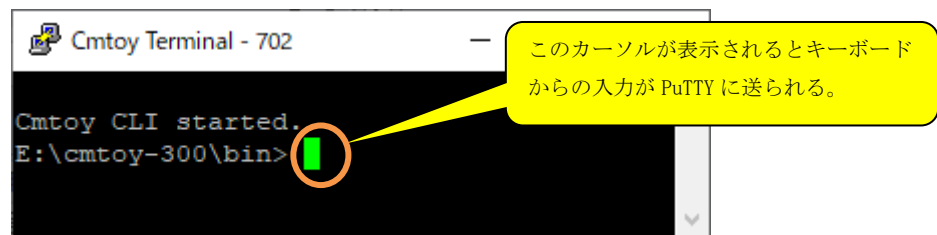


6. ここで「Open」をクリックすると、PuTTY のウィンドウが開きます。

PuTTY が Cmttoy に接続できると、Cmttoy 側のアイコンの色が変わります。ここで改行コードを CR 指定、文字符号化方式を UTF8 指定に変えます。



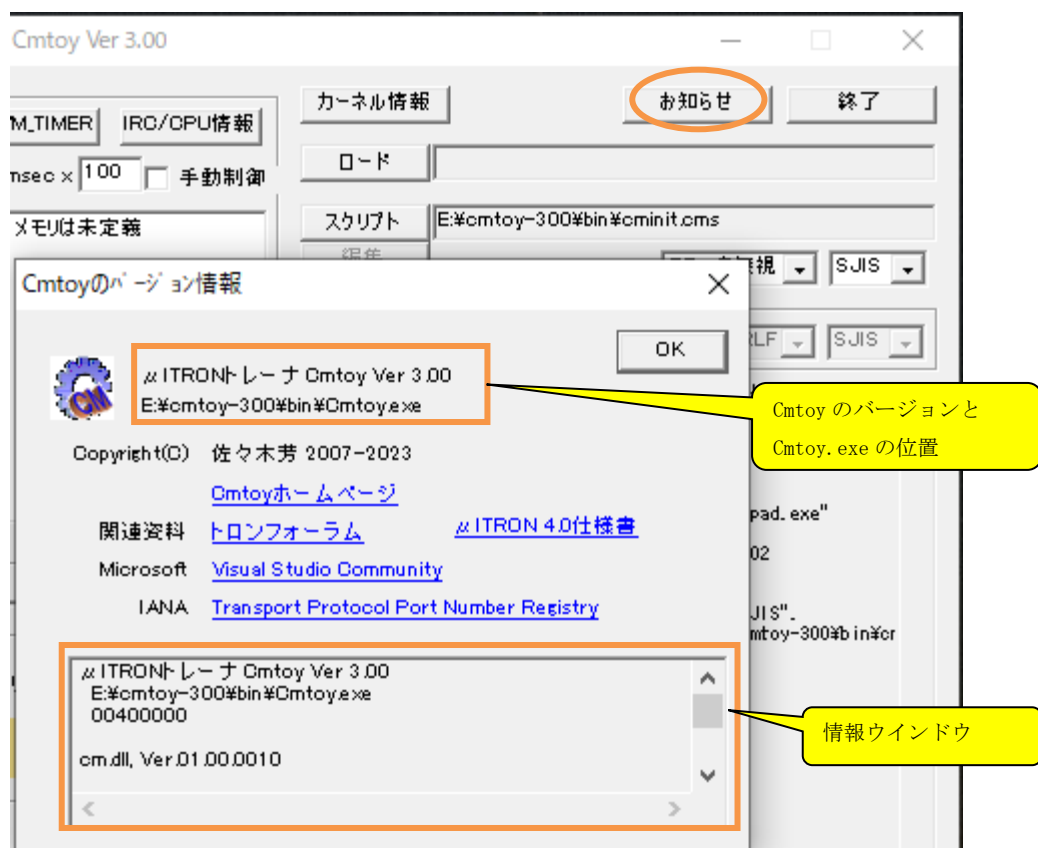
7. ここで PuTTY のウィンドウ内をクリックして（キーボードの入力権を与えて）「ENTER」キーを押すと以下のようにプロンプト文字が表示されます。



以後、キーボードからコマンドラインを入力し「ENTER」キーでコマンドを実行できます。キー入力は PuTTY 画面上に表示されます。コマンドを実行した結果も PuTTY 画面に表示されます。

## 2.17 Cmtoy のバージョン情報

「お知らせ」ボタンをクリックすると、バージョン情報のウインドウを表示します。



「情報ウィンドウ」には以下のような情報が含まれています。各実行モジュールのバージョン以外の情報も含まれています。

```
μ ITRON トレーナ Cmtoy Ver 3.00  
E:¥cmtoy-300¥bin¥Cmtoy.exe  
00400000
```

```
cm.dll, Ver.01.00.0010  
E:¥cmtoy-300¥bin¥cm.dll  
10000000
```

```
kpdll.dll, Ver.01.01.0010  
E:¥cmtoy-300¥bin¥kpdll.dll  
02B00000
```

```
Cmtoy Led Control, Version 1.0  
E:¥cmtoy-300¥bin¥led.ocx  
029F0000
```

```
Cmtoy Segmented Led Control, Version 1.1  
E:¥cmtoy-300¥bin¥segled.ocx  
02990000
```

```
Cmtoy Push Control, Version1.2  
E:¥cmtoy-300¥bin¥push.ocx  
029C0000
```

```
Cmtoy UART over TCP/IP Control, Version 1.7  
E:¥cmtoy-300¥bin¥serial.ocx  
029D0000
```

Cmtoy の各実行モジュールのバージョンおよびファイル名  
その他

Cmtoy 作成時（コンパイル時）の  
Windows 依存パラメータおよび  
C++/C コンパイラバージョン

```
Windows build parameters :  
WINVER=0500H Windows 2000 _WIN32_WINNT_WIN2K  
_WIN32_WINNT=0500H  
_MSC_VER=1200 Visual Studio 6.0
```

```
Priority class = NORMAL_PRIORITY_CLASS  
UI Thread : id = 7794, priority = THREAD_PRIORITY_NORMAL
```

```
Host name = "xxxxxxx"  
IP address = 192.168.101.1
```

実行時の Windows の  
プロセス、スレッド情報

PC のネットワーク内の  
ホスト名と IP アドレス

## 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

$\mu$  ITRON 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` で定義されているプリプロセッサマクロ（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 |
| ・ メー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観点からは 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  $\mu$  ITRON 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  $\mu$  ITRON 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、 $\mu$  ITRON 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 でのコンフィギュレーションは 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 言語のプリプロセッサマクロ）を使用します。

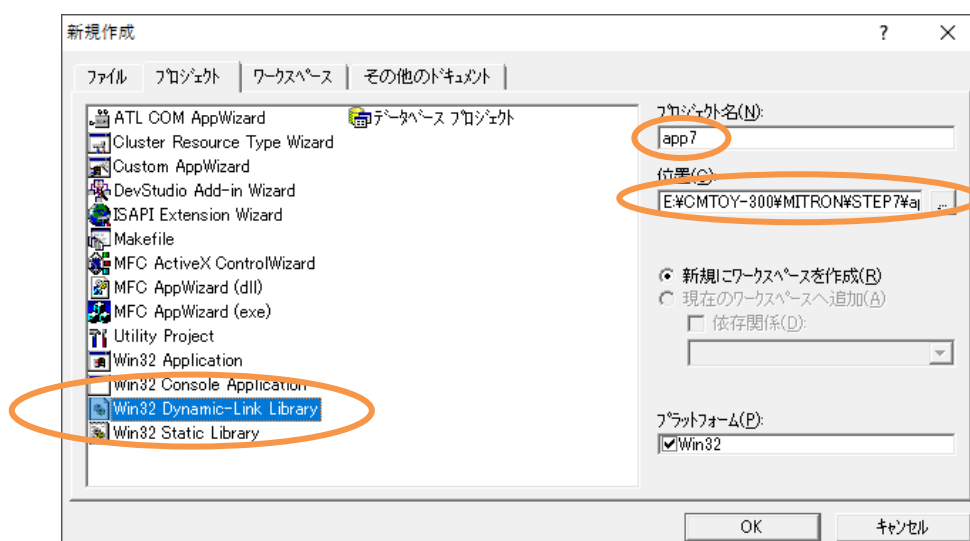
※  $\mu$ ITRON アプリケーション内では main 関数はほかの関数と同列なので、なくても問題になりません。最初に実行される関数はコンフィギュレーションで指定されたタスク関数です。

### 3.1.2 VisualStudio6.0 を使う

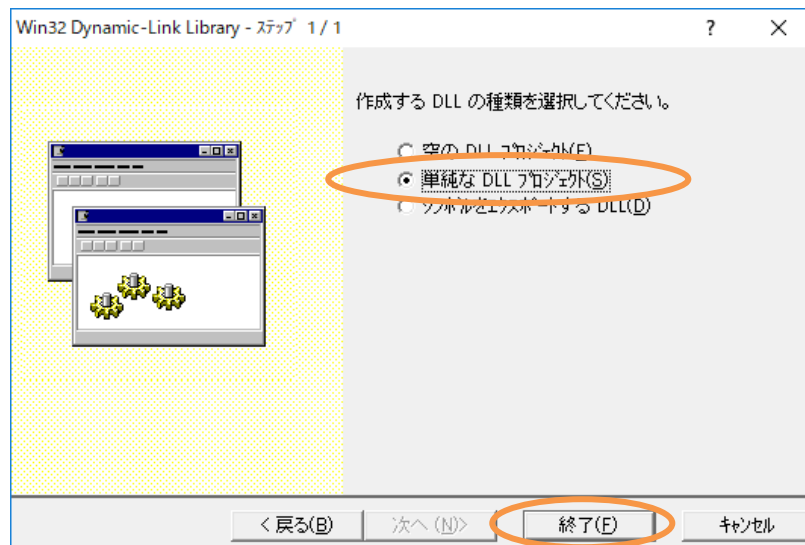
まず、mITRON の下にフォルダ step7 を作ります。



ここで作成したフォルダ step7 に VisualStudio6.0 で DLL プロジェクトを作成します。VisualStudio6.0 の「ファイル」メニューの「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ィンドウが表示されるので、プロジェクトの種類として「Win32 Dynamic-Link Library」を選択します。位置とプロジェクト名 app7 を指定して「OK」をクリックします。



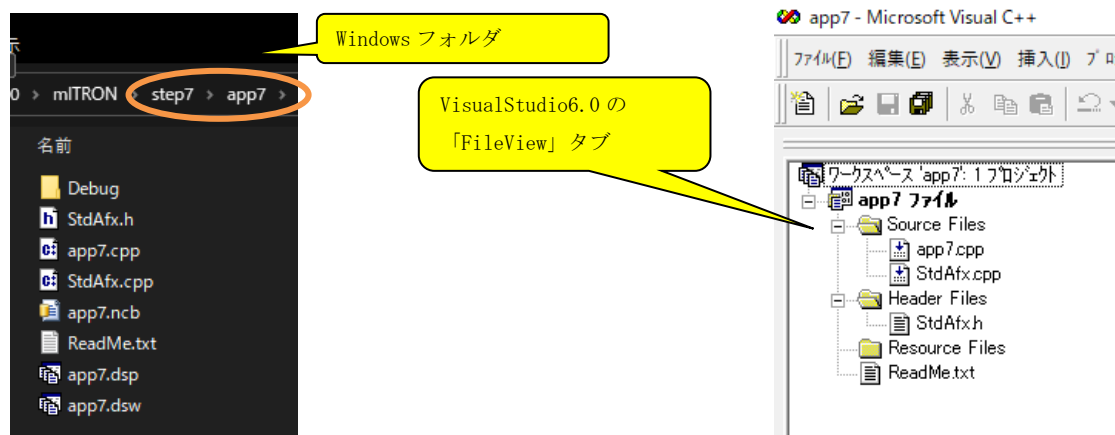
次に、以下の「単純な DLL プロジェクト」を選択し、「終了」をクリックします。



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

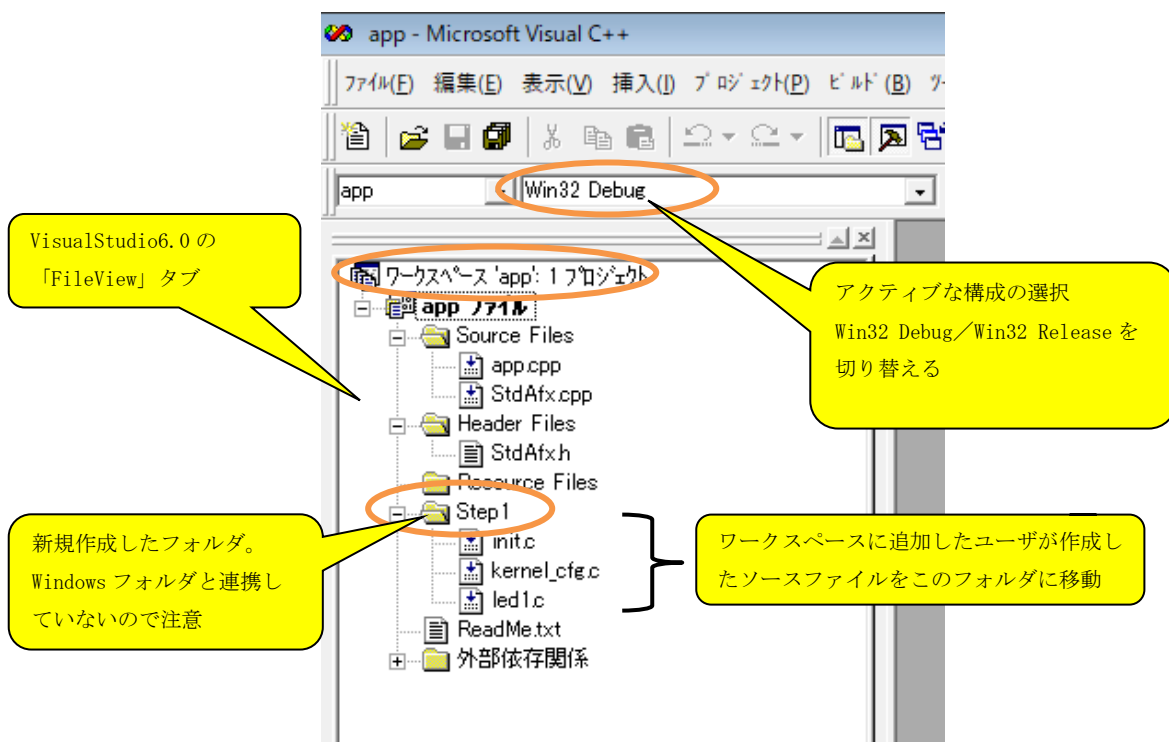
StdAfx.cpp StdAfx.h を include  
 app7.cpp DllMain を含む  
 app7.dsp VS6 プロジェクトファイル  
 app7.dsw VS6 ワークスペースファイル

ここで、「プロジェクト(P)」メニューの「設定(S)...」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。

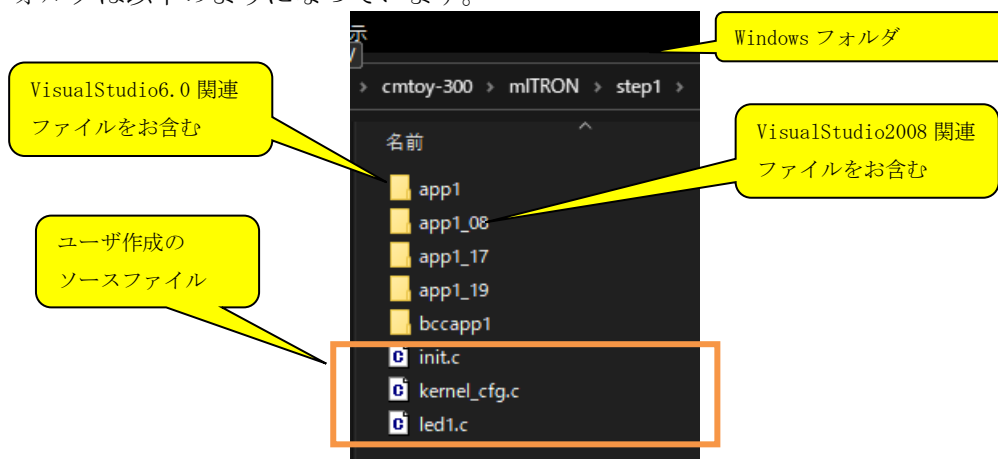
その後、 $\mu$ ITRON アプリケーションの C 言語のソースファイル(\*.c)、ヘッダファイル(\*.h)をプロジェクトに追加します。ファイルの追加は「プロジェクト(P)」メニューの「プロジェクトへ追加(A)」→「ファイル(F)...」から。

Visual C/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、「[10  \$\mu\$ ITRON 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 で開くと以下ようになります。

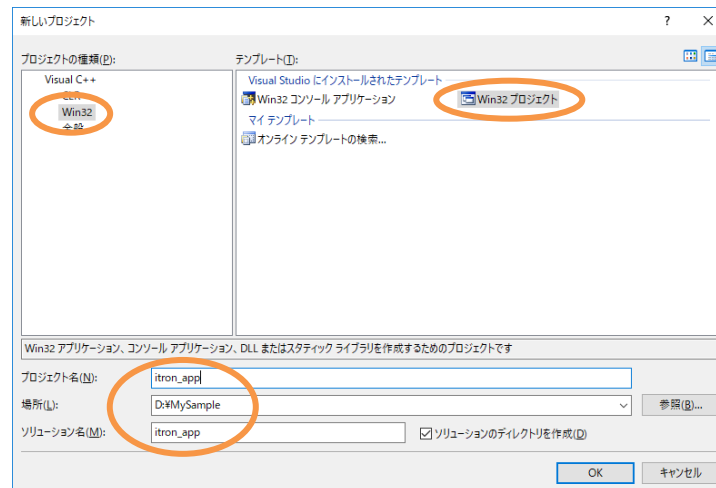


実際のフォルダは以下のようにになっています。

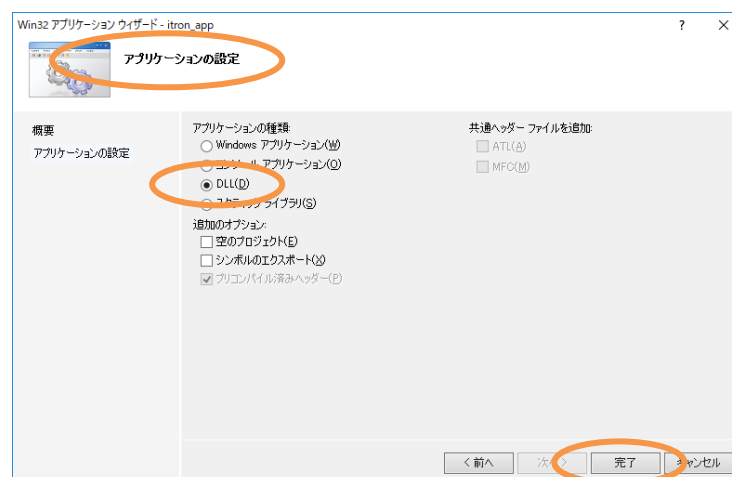


### 3.1.3 Visual C++ 2008 Express Edition を使う

「ファイル」メニューの「開く(O)」→「プロジェクト/ソリューション(P)...」から既存の VisualStudio6.0 のワークスペースファイル(\*.dsw)を開くことができます。  
ここでは新たに dll を作成する場合の説明をします。ファイルメニューの「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

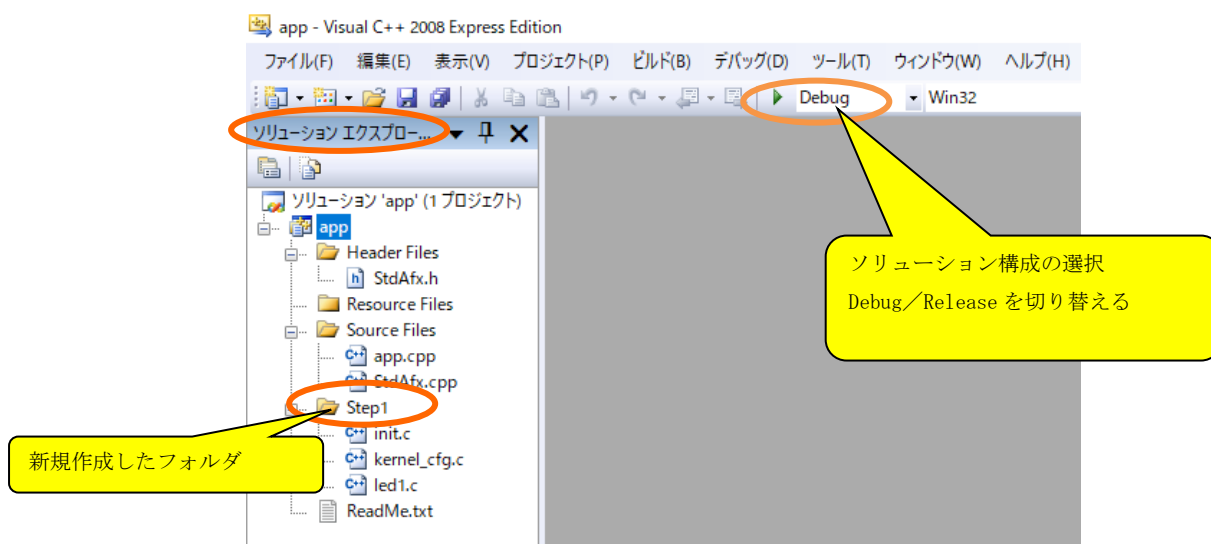
ここで、「プロジェクト(P)」メニューの「プロパティ(P)」から追加のインクルードパス、ライブラリパス、ライブラリファイル、プリプロセッサ定義を登録します。詳細は「[3.2 ビルド方法](#)」を参照してください。

その後、 $\mu$ ITRON 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。

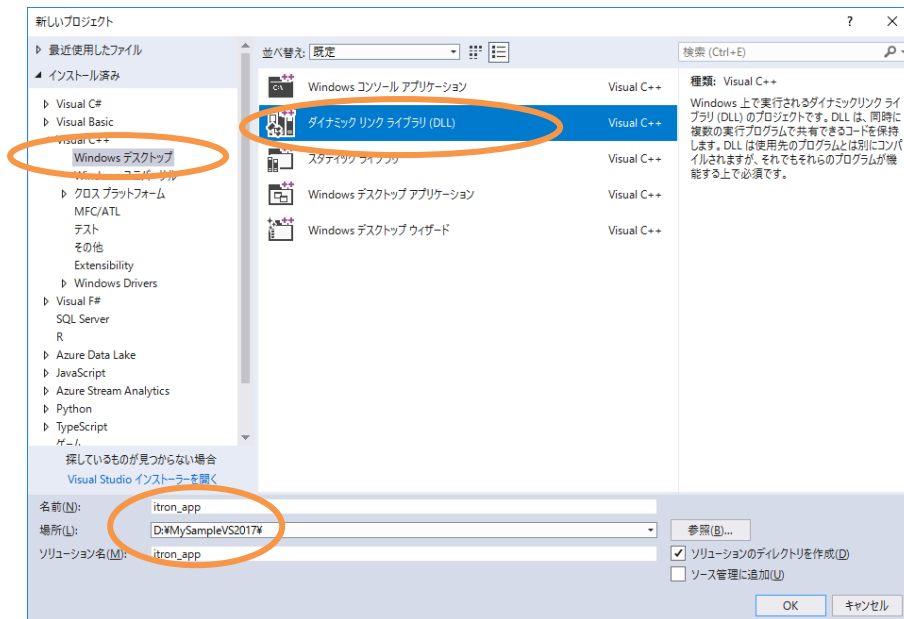
例えば、「[10  \$\mu\$ ITRON チュートリアル](#)」で示すステップ 1 のソリューションファイルを Visual C++ 2008 Express Edition で開くと以下ようになります。



### 3.1.4 Visual Studio 2017 を使う

「ファイル」メニューの「開く(O)」→「プロジェクト/ソリューション(P)...」から既存の VisualStudio6.0 のワークスペースファイル(\*.dsw)、VisualStudio2008 のソリューションファイル(\*.sln)を開くことができます。

ここでは新たに dll を作成する場合の説明をします。ファイルメニューの「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

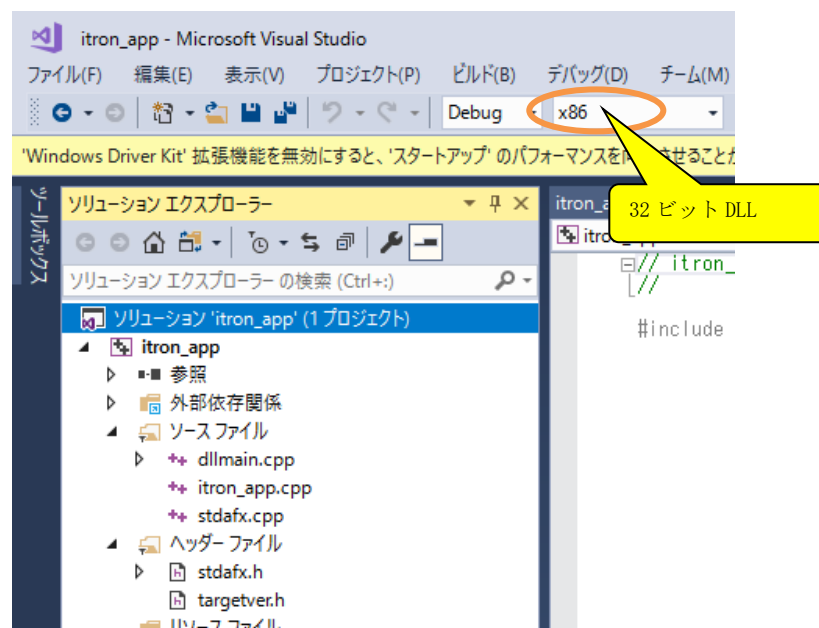
targetver.h

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

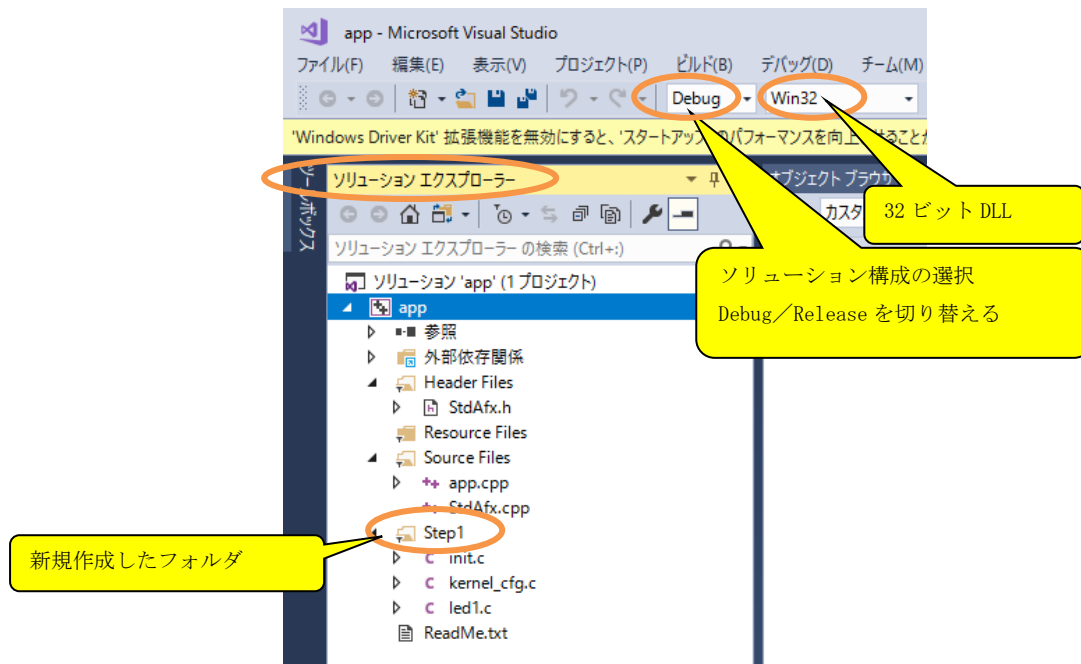


ここで、「プロジェクト(P)」メニューの「プロパティ(P)」から追加のインクルードパス、ライブラリパス、ライブラリファイル、プリプロセッサ定義を登録します。詳細は「[3.2 ビルド方法](#)」を参照してください。

その後、 $\mu$ ITRON 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、「[10 μITRON 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 で開くと以下ようになります。



## 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)」メニューの「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（デバッグ/リリースバージョン）

「インクルードファイルのパス」は”..¥..¥..¥include”（デバッグ/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”（デバッグバージョン）

カテゴリ：インプット

「追加ライブラリパス」は、”..¥..¥..¥LIB”（デバッグ/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~~（デバッグバージョン）

#### ●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

“E:\cmttoy-300\bin\Cmttoy.exe” (デバッグバージョンのみ)

Cmttoy インストールフォルダ

### 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\_08¥app.sln を Visual C++ 2008 Express Edition で開いて「プロジェクト(P)」メニューの「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” (デバッグ/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 (デバッグ/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” (デバッグバージョン)

全般の「追加のライブラリディレクトリ」へ

“..¥..¥..¥LIB” (デバッグ/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” (デバッグ/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 Cmttoy¥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 (デバッグバージョン)~~

#### ●デバッグ内の

「コマンド」へ

“E:\cmttoy-300\bin\Cmttoy.exe” (デバッグバージョンのみ)

Cmttoy インストールフォルダ

### 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\_17¥app.sln を Visual Studio 2017 で開いて「プロジェクト(P)」メニューの「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” (デバッグ/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 (デバッグ/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” (デバッグバージョン)

全般の「追加のライブラリディレクトリ」へ

“..¥..¥..¥LIB” (デバッグ/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” (デバッグ/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 Cmttoy¥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

”E:\%cmtoy-300\bin\%Cmtoy.exe” (デバッグバージョンのみ)

Cmtoy インストールフォルダ

### 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 以降では Borland C++は未確認。

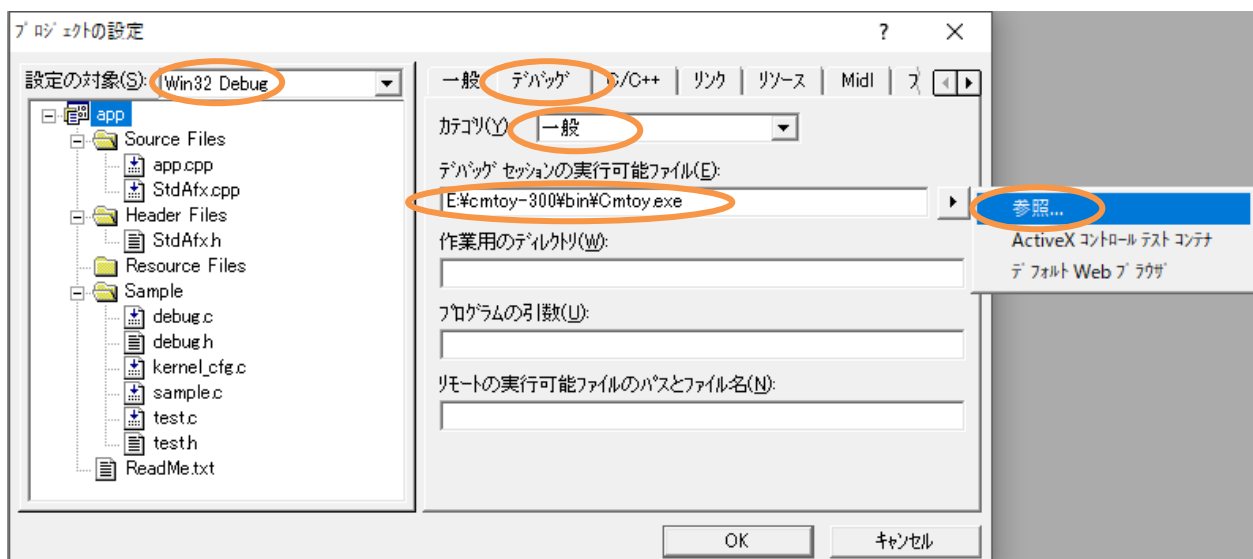
## 3.3 VisualStudio6.0 のデバッガの使用

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、

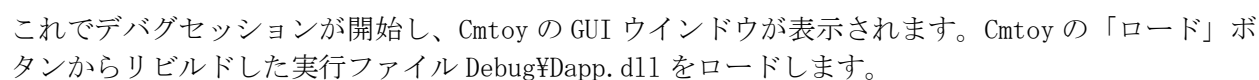
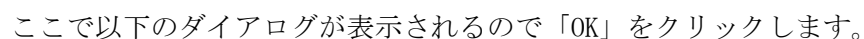
「ビルド(B)」メニューの「アクティブな構成の設定(0)」で構成「Win32 Debug」を選択してリビルドします。リビルドすると実行モジュール内のソースファイルへのパス情報がそのマシンのものになり、デバッガがソースファイルを認識できるようになるようです。

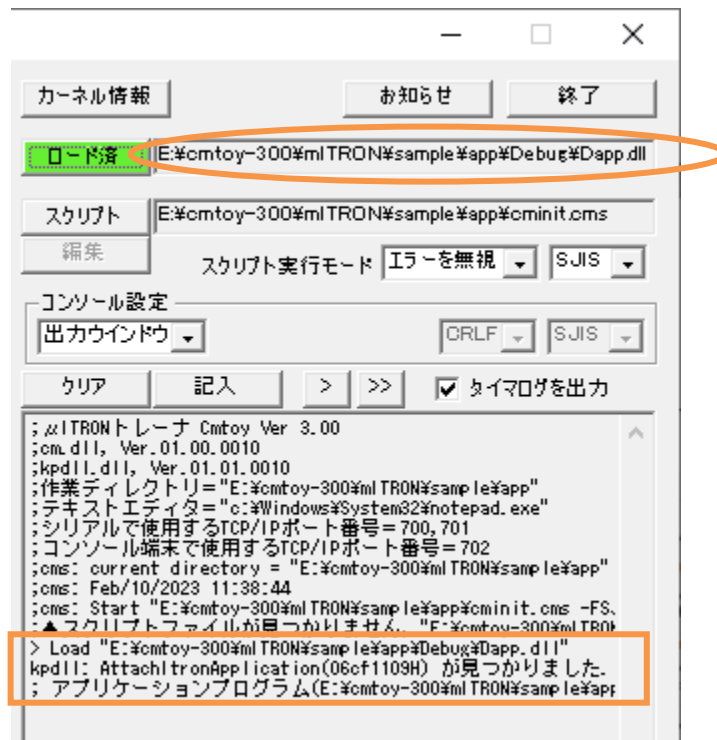
### 3.3.1 μITRON アプリケーションのプロジェクトからデバッガを使う

まず、プロジェクトファイル(app.dsw)を開き「プロジェクト」メニューの「設定」を開きます。その中の「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。(デバッグバージョン Win32 Debug のみ)



ここで、「ビルド」メニューの「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。

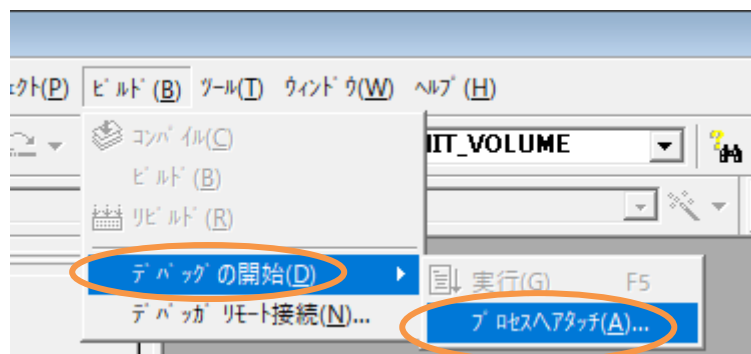




ここで、アプリケーションのソースファイルを開きブレークポイントを設定し、Cmtoy の「リセット」ボタンから  $\mu$ ITRON カーネルとアプリケーションを実行します。

### 3.3.2 Cmtoy 起動後にデバッグを使う

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)」メニューの「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。



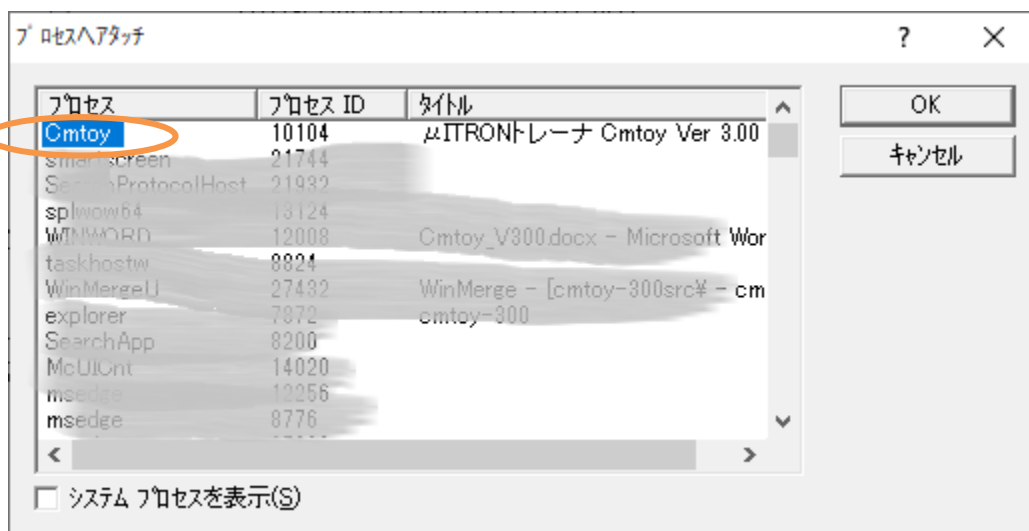


図 3-1 実行中のプロセス一覧

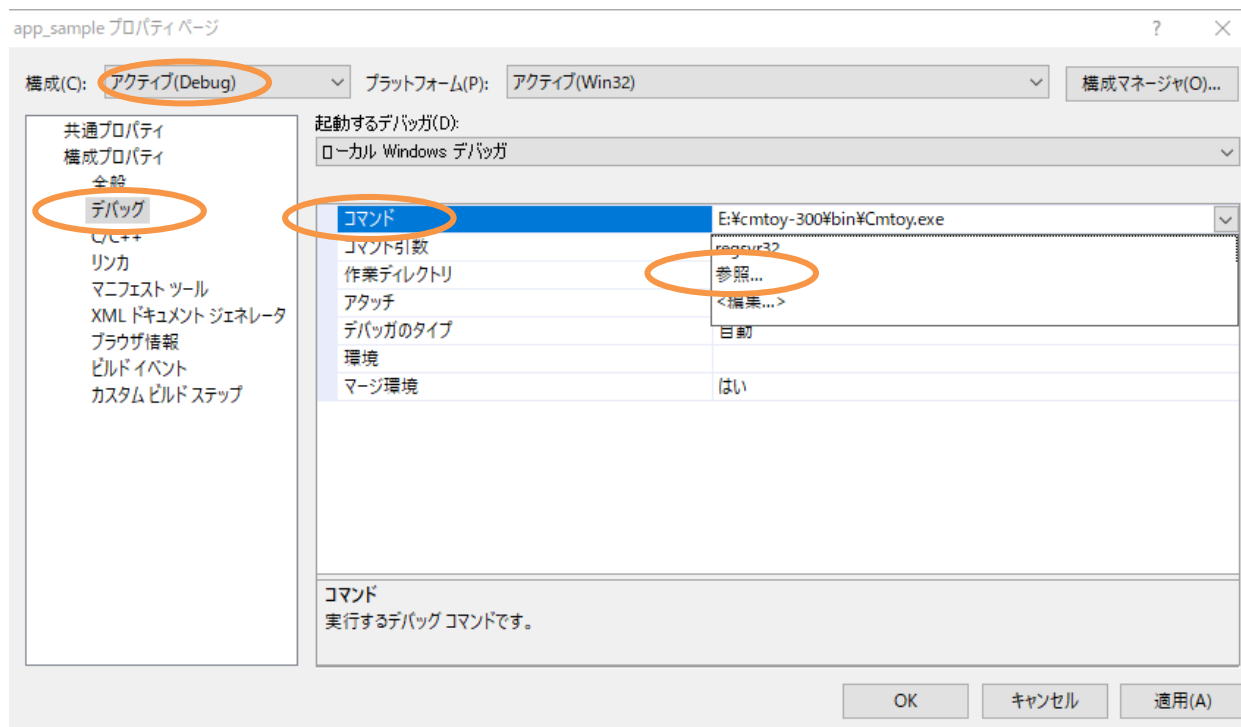
ここで Cmtoy を選んで「OK」ボタンをクリックすると実行中のプロセスをデバッガでデバッグできるようになります。例えば、デバッグしたいソースファイルを開きブレークポイントを設定した後、GUI の「リセット」ボタンをクリックして  $\mu$ ITRON カーネルとアプリケーションを実行します。

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

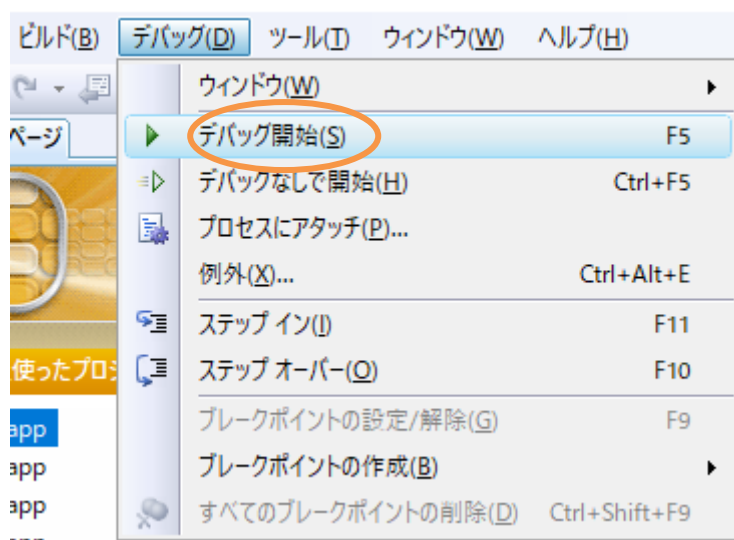
まず、VisualC++ 2008 Express Edition で付属のサンプルアプリケーションのソリューションファイル (\*.sln) を開き、ソリューション構成が「Debug」、ソリューションプラットフォームが「Win32」であることを確認し、「ビルド(B)」メニューの「ソリューションのリビルド(R)」リビルドします。リビルドすると実行モジュール内のソースファイルへのパス情報がそのマシンのものになり、デバッガがソースファイルを認識できるようになるようです。

#### 3.4.1 $\mu$ ITRON アプリケーションのソリューションからデバッガを使う

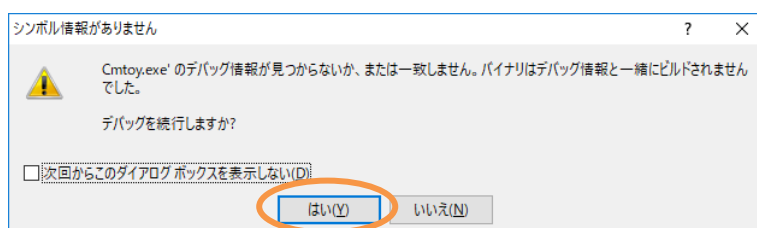
まず、ソリューションファイル(app.sln)を開きソリューション構成が「Debug」であることを確認します。「プロジェクト(P)」メニューの「プロパティ」を選択しプロパティページを表示します。その構成プロパティのデバッグで、「コマンド」ボックスへ Cmtoy のインストールフォルダから Cmtoy.exe を指定します。「コマンド」ボックスの右端の↓からドロップダウンメニューが表示されるのでその中の「参照...」を使って実行ファイル Cmtoy.exe を指定します。「OK」で終了。



次に、ソリューション構成が「Debug」であることを確認して、「ビルド」メニューの「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。これでデバッグの準備ができました。ここで、「デバッグ (D)」メニューの「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。



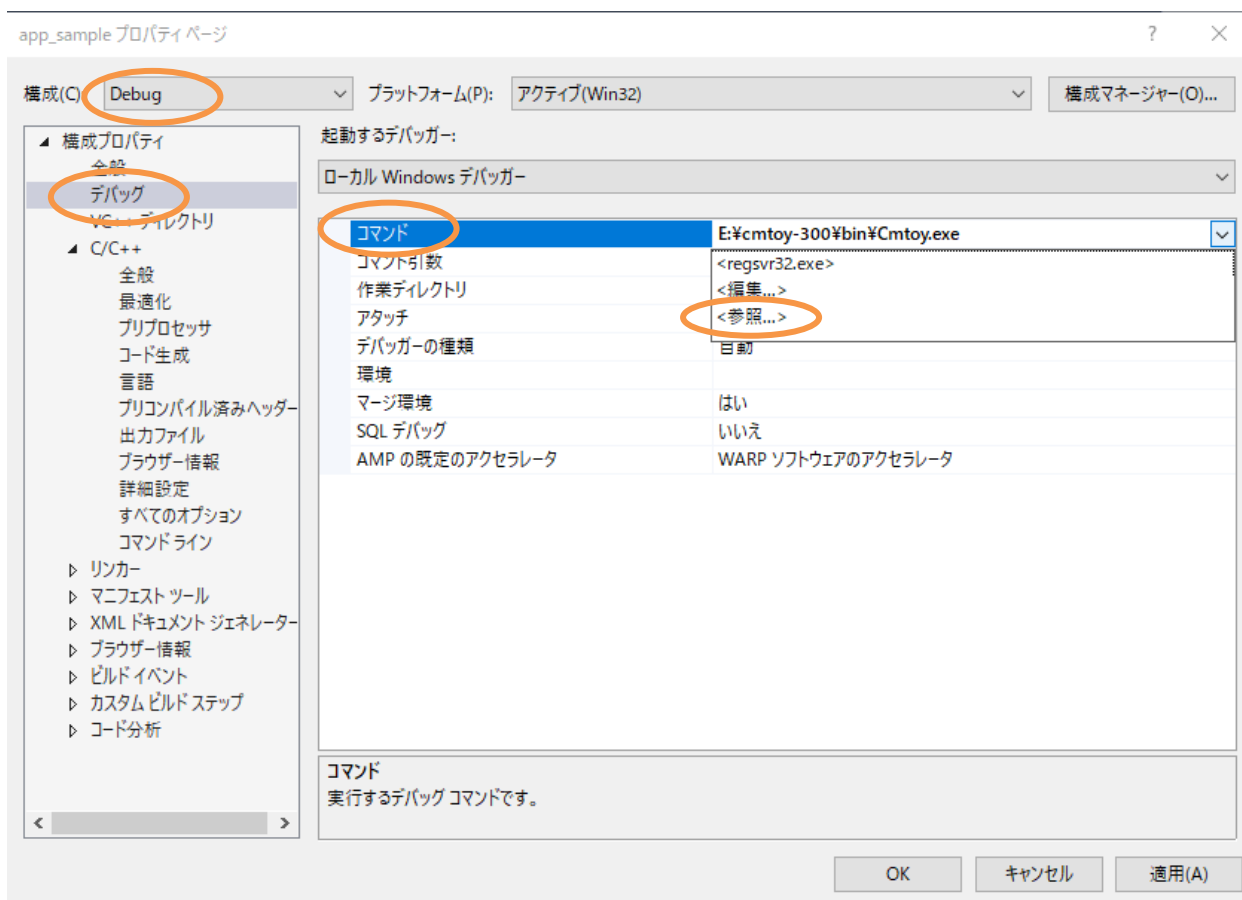
これでデバッグセッションが開始し、Cmtoy の GUI ウィンドウが表示されます。Cmtoy の「ロード」ボタンからリビルドした実行ファイル Debug¥Dapp. dll をロードします。  
ここで、アプリケーションのソースファイルを開きブレークポイントを設定し、Cmtoy の「リセット」ボタンから  $\mu$  ITRON カーネルとアプリケーションを実行します。

### 3.5 Visual Studio 2017 のデバッガの使用

まず、Visual Studio 2017 で付属のサンプルアプリケーションのソリューションファイル (\*.sln) を開き、ソリューション構成が「Debug」、ソリューションプラットフォームが「Win32」であることを確認し、「ビルド(B)」メニューの「ソリューションのリビルド(R)」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へ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。

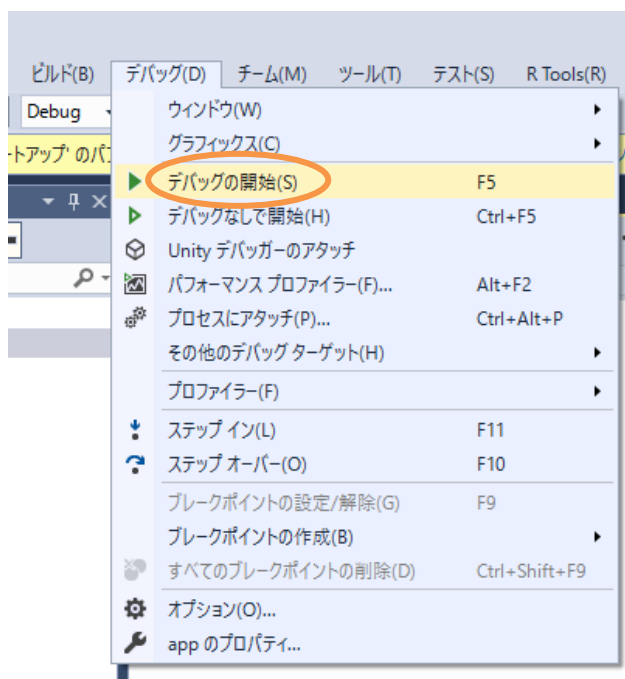
#### 3.5.1 $\mu$ ITRON アプリケーションのソリューションからデバッガを使う

まず、ソリューションファイル(app.sln)を開きソリューション構成が「Debug」であることを確認します。「プロジェクト(P)」メニューの「プロパティ」を選択しプロパティページを表示します。その構成プロパティのデバッグで、「コマンド」ボックスへCmtoy のインストールフォルダからCmtoy.exe を指定します。「コマンド」ボックスの右端の↓からドロップダウンメニューが表示されるのでその中の「参照...」を使って実行ファイルCmtoy.exe を指定します。「OK」で終了。



次に、ソリューション構成が「Debug」であることを確認して、「ビルド」メニューの「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。これでデバッグの準備ができました。

ここで、「デバッグ(D)」メニューの「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。



これでデバッグセッションが開始し、Cmtoy の GUI ウィ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  $\mu$  ITRON カーネルとアプリケーションを実行します。

## 4 μ ITRON 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 C-Machine とは](#)」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  $\mu$  ITRON 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

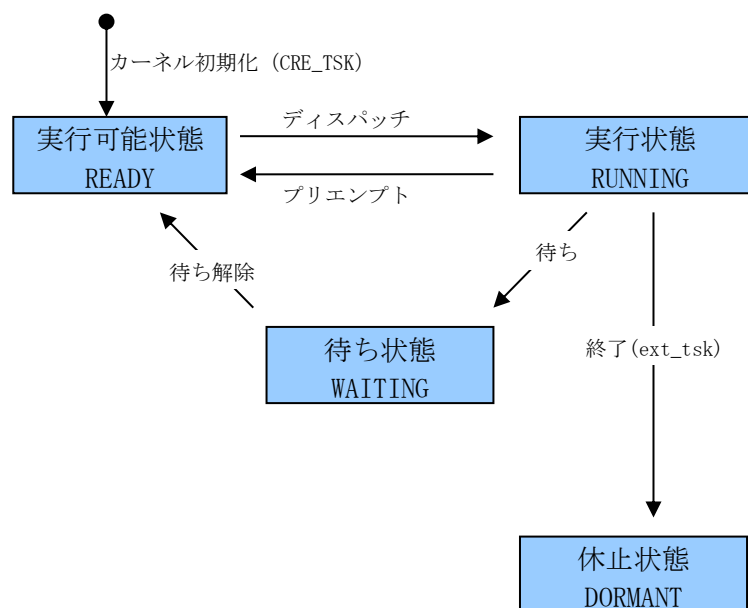
タスクは C の関数として記述して、コンフィグレーションでタスクとして登録します。

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り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` としてください。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。この状態でも割り込み要求があれば割り込みハンドラは動きタスクスケジューリングは発生します。

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

※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り込みレベル `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り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り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り込みをマニュアル操作で `1` 回ずつ起こす

このように操作してもカーネルは割り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 ウィンドウの「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、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。

カーネル情報 (kpdll.dll : Ver.0100.0007)

タスクトレース		サービスコールトレース		カーネルオブジェクト		保守用	
SystemTime=		0000004f		x10msec		表示更新	
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)		
193	0000003c	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
198	0000003c	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
199	0000003e	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
204	0000003e	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
205	00000040	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
210	00000040	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
211	00000042	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
216	00000042	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
217	00000044	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
222	00000044	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
223	00000046	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
228	00000046	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
229	00000048	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
234	00000048	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
235	0000004a	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
240	0000004a	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
241	0000004c	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
246	0000004c	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		
247	0000004e	0 "idle"	5 "Watch B...	-50 (E_TMOUT)	-18 (tslp_tsk)		
252	0000004e	5 "Watch B...	0 "idle"	-50 (E_TMOUT)	-18 (tslp_tsk)		

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。

カーネル情報 (kpdll.dll : Ver.0100.0007)

タスクトレース | **サービスコールトレース** | カーネルオブジェクト | 保守用

SystemTime= 000001aa x10msec 表示更新

番...	タイムス...	コンテ...	タスク/ハンド...	サービスコール	param1/erod	param2
1288	000001a2		5 "Watch B...	-18 (tslp_tsk)	-50 (E_TMOU...	0
1289	000001a2	TEUe	5 "Watch B...	-78 (get_tim)	109772580	0
1290	000001a2		5 "Watch B...	-78 (get_tim)	0 (E_OK)	0
1291	000001a2	TEUe	5 "Watch B...	-18 (tslp_tsk)	20	0
1294	000001a4		5 "Watch B...	-18 (tslp_tsk)	-50 (E_TMOU...	0
1295	000001a4	TEUe	5 "Watch B...	-78 (get_tim)	109772580	0
1296	000001a4		5 "Watch B...	-78 (get_tim)	0 (E_OK)	0
1297	000001a4	TEUe	5 "Watch B...	-18 (tslp_tsk)	20	0
1300	000001a6		5 "Watch B...	-18 (tslp_tsk)	-50 (E_TMOU...	0
1301	000001a6	TEUe	5 "Watch B...	-78 (get_tim)	109772580	0
1302	000001a6		5 "Watch B...	-78 (get_tim)	0 (E_OK)	0
1303	000001a6	TEUe	5 "Watch B...	-18 (tslp_tsk)	20	0
1306	000001a8		5 "Watch B...	-18 (tslp_tsk)	-50 (E_TMOU...	0
1307	000001a8	TEUe	5 "Watch B...	-78 (get_tim)	109772580	0
1308	000001a8		5 "Watch B...	-78 (get_tim)	0 (E_OK)	0
1309	000001a8	TEUe	5 "Watch B...	-18 (tslp_tsk)	20	0
1312	000001aa		5 "Watch B...	-18 (tslp_tsk)	-50 (E_TMOU...	0
1313	000001aa	TEUe	5 "Watch B...	-78 (get_tim)	109772580	0
1314	000001aa		5 "Watch B...	-78 (get_tim)	0 (E_OK)	0
1315	000001aa	TEUe	5 "Watch B...	-18 (tslp_tsk)	20	0

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) $\mu$ ITRON 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。

タスクトレース | サービスコールトレース | **カーネルオブジェクト** | 保守用

表示更新

タスク/ハンドラー一覧

ID	名前	優先度	状態	待ち要...	待ちオ...	起床...	待ちパ...	待ちモ...	残り時...
0	idle	16	RUN			0			
1		1	DMT			0			
2		1	DMT			0			
3		3	DMT			0			
4	Timer	3	WAI	SLP		0			95
5	Watch Button	4	WAI	SLP		0			1
INT0	system timer	0							
INT1	count	1							
INT2	iom	2							
INT3	dpm	3							
INT7	fatal error	7							

定義済み割り込みハンドラー一覧

ID	名前	全プロ...	ブロック...	未使...	先頭アドレス	全バイト数	ブロックバイ...

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、割り込みレベル 0, 1, 2, 3, 7 に割り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、割り込みレベル 0 の割り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[<割り込み番号>].<割り込みハンドラ名>

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          メールボックス生成定義 (静的 API)
snd_mbx          メールボックスへの送信
rcv_mbx          メールボックスからの受信 (タイムアウトなし)
prcv_mbx         メールボックスからの受信 (ポーリング)
trcv_mbx         メールボックスからの受信 (タイムアウトあり)
ref_mbx          メー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) 割り込み管理機能 */
DEF_INH          割り込みハンドラ生成定義（静的 API）
dis_int          割り込みの禁止
ena_int          割り込みの許可
vchg_ifl         CPU の割り込みフラグの変更（実装依存機能）
vget_ifl         CPU の割り込みフラグの取得（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  $\mu$  ITRON カーネル(kpd11.d11)もメモリ上にロードします。この段階では  $\mu$  ITRON 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  $\mu$  ITRON アプリケーションをメモリ上に配置します。

$\mu$  ITRON カーネルと  $\mu$  ITRON 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  $\mu$  ITRON 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。「[2.3 アプリケーションプログラムを実行する](#)」を参照。

- 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 割り込み許可、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、プリプロセッサマクロが使えます。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「[11 C-Machine のプログラム例](#)」を参照してください。

### 5.1 データタイプ

`hal.h` で定義している単純データタイプを以下の表に示します。

データタイプ	対応する C 言語のデータタイプ	説明
BYTE	unsigned char	符号なし 8 ビット整数
WORD	unsigned short	符号なし 16 ビット整数
DWORD	unsigned long	符号なし 32 ビット整数
CHAR	char	符号付き 8 ビット整数
SHORT	short	符号付き 16 ビット整数
LONG	long	符号付き 32 ビット整数
NULL	((void *)0)	無効ポインタ値
BOOL	int	真偽値、真(0 以外)、偽(0)

以下の関数プロトタイプではこれらのデータタイプを用いて宣言しています。

### 5.2 CPU、割込み制御関数

#### 5.2.1 void halDisableInterrupt(void);

パラメータ

なし

戻り値

なし

説明

CPU の割込みを禁止する。

#### 5.2.2 void halEnableInterrupt(void);

パラメータ

なし

戻り値

なし

説明

CPU の割込みを許可する。

#### 5.2.3 BOOL halInquireInterruptStatus(void);

パラメータ

なし

戻り値

CPU の現在の割込みフラグ



説明

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。

#### 5.2.4 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

説明

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。

#### 5.2.5 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

説明

割込み処理終了を割込みコントローラに通知する。

※  $\mu$ ITRON 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。

### 5.3 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。

#### 5.3.1 void halDebugOutputString(const char \*cstr);

パラメータ

cstr	文字列
------	-----

戻り値

なし

説明

デバッグ用の文字列を出力する。

#### 5.3.2 void halDebugPrintf(const char \*formatstring, ...);

パラメータ

formatstring	書式制御文字列
--------------	---------

戻り値

なし

説明

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。

## 5.4 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。

### 5.4.1 void halSetLED(WORD led);

パラメータ

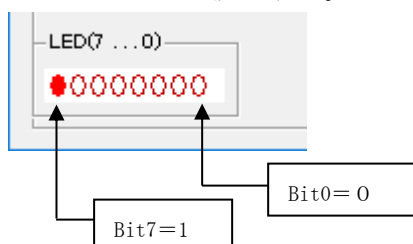
led                      各ビットで LED の ON(1)/OFF(0)を指定する。

戻り値

なし

説明

LED の ON/OFF を設定する。



### 5.4.2 void halSetSegLED(WORD stat);

パラメータ

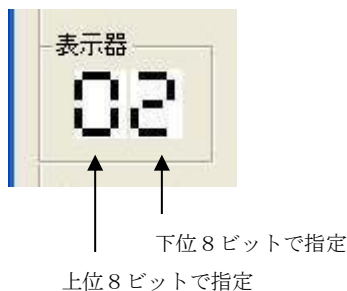
stat                    各ビットで LED セグメントの ON(1)/OFF(0)を指定する。

戻り値

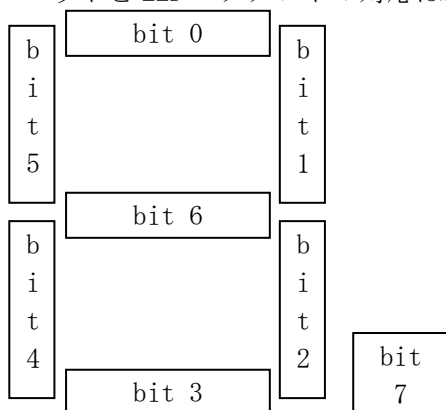
なし

説明

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。



## 5.5 ボリューム制御関数

### 5.5.1 WORD halGetVolume(int VolumeNo);

パラメータ

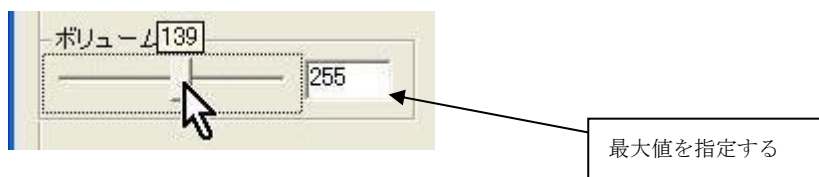
VolumeNo            ボリューム番号（0を指定）

戻り値

現在のボリューム値を返す。

説明

ボリューム値は0～最大値の間の整数値。



## 5.6 DIP スイッチ制御関数

### 5.6.1 WORD halGetSwitch(void);

パラメータ

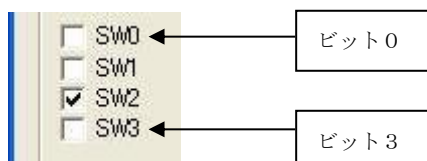
なし

戻り値

下位4ビットで各スイッチの ON(1)/OFF(0)を返す

説明

各ビットでスイッチの状態を返す。



## 5.7 ボタン制御関数

### 5.7.1 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。

説明

UP                                    DOWN（マウスの左ボタンを押した状態）



## 5.8 簡易シリアル制御関数

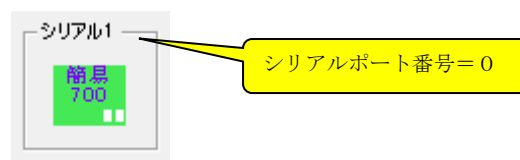
C-Machine はシリアルポートを Windows の Winsock 機能を使ってシミュレートします。TCP/IP のポート 700 と 701 を使います。ポート番号を変更するには「[2.2.1 serial.ocx の使用する TCP/IP ポート番号を変更する](#)」を参照してください。

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。

### 5.8.1 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

説明

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。

### 5.8.2 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）

説明

受信データを取得する。

### 5.8.3 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

説明

文字コードを速やかに TCP/IP ポートから送信する。

## 5.9 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 serial.ocx の使用する 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コンソールコマンドを使って変更することができます。シリアルを操作するコマンドについては「[7.4 serial <シリアルポート番号>ne <サブコマンド>](#)」を参照してください。

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。

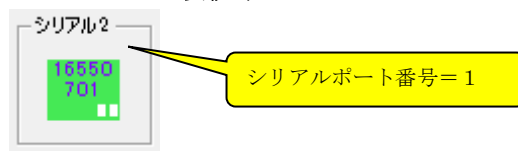
```
serial 1 init 4 16550 ;シリアル2に割込みレベル4を割り当てる
```

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



16550 を制御するには以下で説明する制御関数を使います。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）の読み書き込みはできませんが、ボーレートは変わりません。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

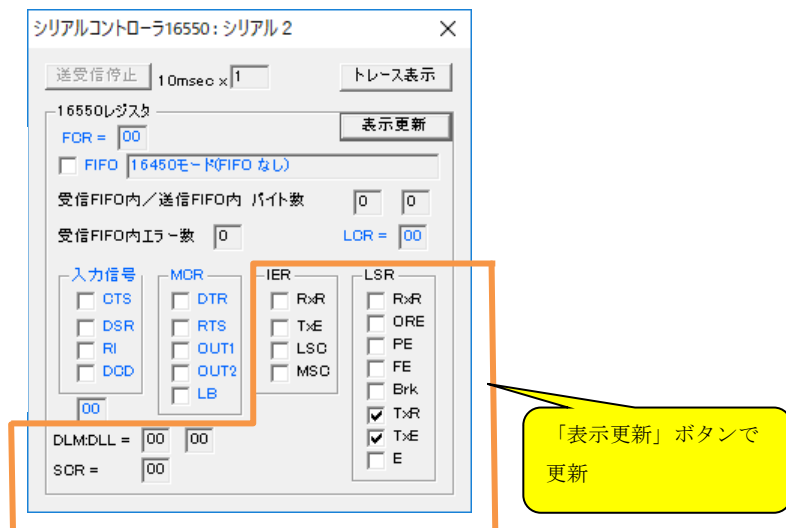


図 5-1 シリアル 2 の 16550 のレジスタ

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 モード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*/
WRITE_16550(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*/
LSTAT_16550(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。

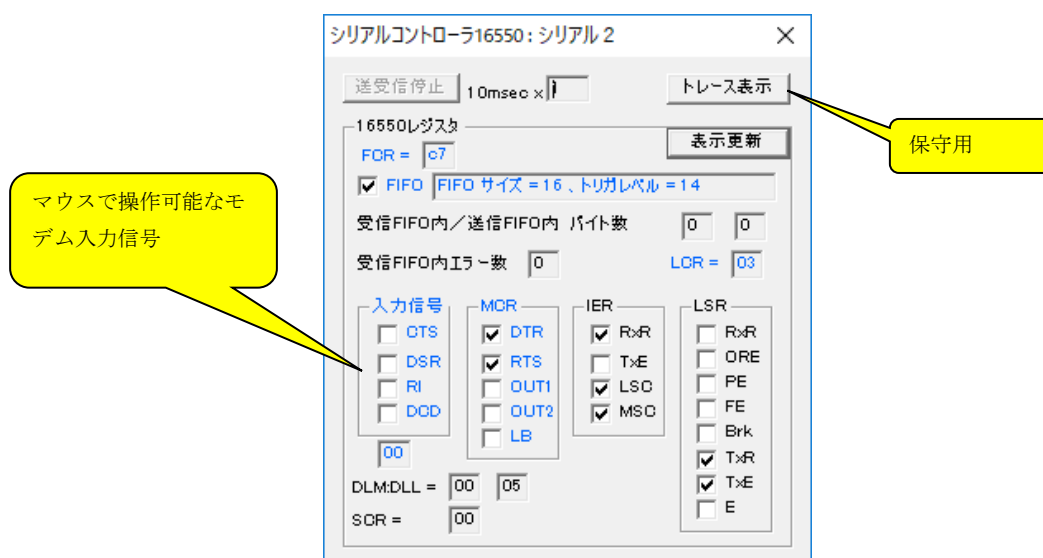


図 5-2 初期化後の 16550 のレジスタ

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 への外部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ー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)  
<http://www.ti.com/lit/ds/symlink/tl16c550c.pdf>

#### 5.9.1 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

説明

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。どちらに書くかはライン制御レジスタ (LCR) の DLAB の値で決まる。[5.9.9 void hal16550WriteLCR\(int SerialNo, BYTE d\)](#) を参照。

※Cmtoy ではディバイザラッチレジスタ (DLL) への書き込みは動作に影響しない。

#### 5.9.2 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

説明

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。どちらを読むかはライン制御レジスタ (LCR) の DLAB の値で決まる。[5.9.9 void hal16550WriteLCR\(int SerialNo, BYTE d\)](#) を参照。

#### 5.9.3 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

説明

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。どちらに書くかはライン制御レジスタ (LCR) の DLAB の値で決まる。[5.9.9 void hal16550WriteLCR\(int SerialNo, BYTE d\)](#) を参照。  
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。

#### 5.9.4 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

説明

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。どちらを読むかはライン制御レジスタ (LCR) の DLAB の値で決まる。[5.9.9 void hal16550WriteLCR\(int SerialNo, BYTE d\)](#) を参照。

#### 5.9.5 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

説明

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



#### 5.9.6 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

##### 説明

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)

#### 5.9.7 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

##### 説明

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)

#### 5.9.8 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

##### 説明

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

#### 5.9.9 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

##### 説明

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 (DLAB)

※Cmtoy ではビット 6, 7 が意味を持つ。

#### 5.9.10 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

##### 説明

16550 のライン制御レジスタ (LCR) の値を読み取る。

#### 5.9.11 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

##### 説明

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

#### 5.9.12 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

説明

16550 のモデム制御レジスタ (MCR) の値を読み取る。

### 5.10PN 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。

#### 5.10.1 WORD halCalcPN9(WORD pn\_code);

パラメータ

pn\_code                      初期値

戻り値

計算結果

説明

与えられた pn\_code から PN9 の計算をします。

#### 5.10.2 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

説明

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。

#### 5.10.3 WORD halCalcPN15(WORD pn\_code);

パラメータ

pn\_code                      初期値

戻り値

計算結果

説明

与えられた pn\_code から PN15 の計算をします。

#### 5.10.4 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
説明		
		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。

## 5.11 マクロ

ここでは C 言語のプリプロセッサ・マクロを使って実装している機能を説明します。

### 5.11.1 CMTRACE (const char \*formatstring, ...)

パラメータ	
	formatstring    書式制御文字列
戻り値	
	なし
説明	
	文字列を Cmtoy の出力ウインドウに出力する。

### 5.11.2 CPU 制御

#### (1) DIS\_INT

説明	
	CPU を割込み禁止にします。

#### (2) ENA\_INT

説明	
	CPU を割込み許可にします。

### 5.11.3 ターゲットメモリを操作（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

説明

ターゲットメモリに値を書く。

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                    ターゲットメモリの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

説明

ターゲットメモリの値を読む。

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 I/O ポート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

説明

ターゲット I/O ポートに値を書く。

未定義の I/O 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 I/O ポートの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

説明

ターゲット I/O ポートの内容を読む。

未定義の I/O アドレスを指定すると、出力ウィ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                      ターゲットメモリの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。

説明

ターゲットメモリのアドレス 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

説明

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り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

説明

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り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

説明

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

説明

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。

#### 5.11.4 ターゲットメモリを操作する（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

説明

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

説明

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。

説明

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

説明

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

説明

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

説明

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

説明      バイト、ワードまたはダブルワードの値

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 で保護して割込みが入らないことを保証する。

## 6 CLI(コマンドライン・インタプリタ)

一般に CLI は Command Line Interface を指しますが本書では Command Line Interface を実現するプログラムであるコマンドライン・インタプリタ (Command Line Interpreter) を指すことにします。

ここでは、Cmtoy を制御するための CLI について説明します。この CLI はコマンド機能とスクリプト機能から構成されるプログラムです。

コマンド機能を使うと以下の操作が行えます。

- ・  $\mu$  ITRON アプリケーションの操作 (ロード、実行開始など)
- ・ タイマの設定、操作、参照
- ・ デバイス (CPU, IRC, シリアル、ボリューム、スイッチ、ボタン) の設定、操作、参照
- ・ メモリの設定、操作、参照
- ・ スクリプトファイルの実行
- ・ コマンド／スクリプトの機能設定
- ・ その他の GUI の操作と同等の操作

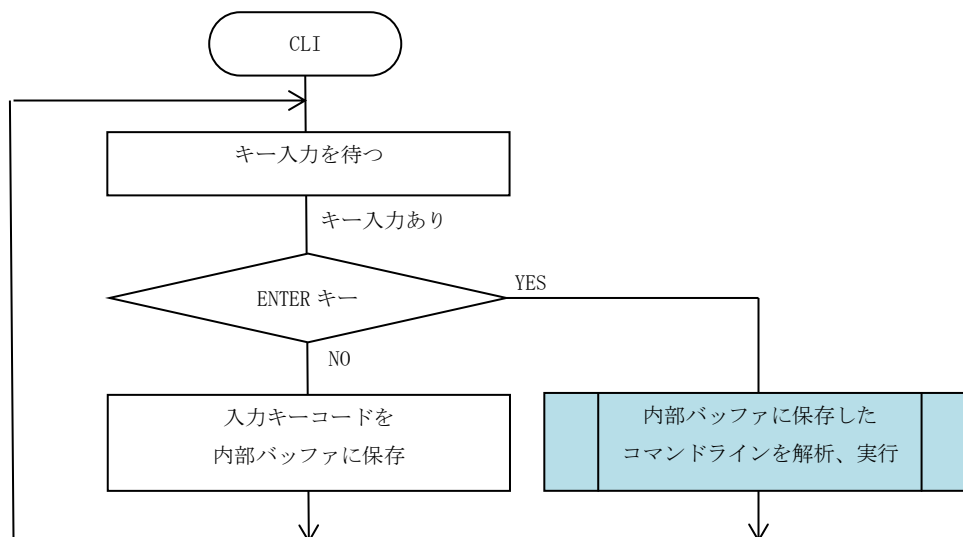
コマンドやスクリプトの構文を説明する場合に以下の表記法を使います。

- ・ 左辺 := 右辺  
左辺は右辺で定義される。
- ・ 左辺 := {A | B}  
左辺は A または B のどちらか
- ・ [A] では A は省略可能を意味する
- ・  $\langle \rangle$  で囲んだものは仮パラメータを表し、実行するコマンドラインの中では妥当な文字列 (実パラメータ) に置き換える。

### 6.1 コマンド機能

Cmtoy はキーボードから改行コードまでの文字列をコマンドラインとして受け取り、解析して、該当する処理を実行します。ここでは Cmtoy の CLI の機能について説明します。

一般に CLI は常時キー入力を監視して入力された文字を内部バッファに保存します。改行コードが入力されるとバッファに保存されている文字列をコマンドラインとして解析、実行します。実行後再びキー入力の監視に戻ります。



### 6.1.1 コマンドラインの構文 (シンタックス)

Cmtoy のコマンドラインの構文形式は以下のようになります。

〈コマンド名〉 [〈空白〉 〈パラメータリスト〉] [;〈注釈〉] 〈改行〉

〈コマンド名〉            := {〈定義済みコマンド名〉 | 〈ファイル名〉}  
 〈パラメータリスト〉    := [〈必須パラメータリスト〉] [〈オプションパラメータリスト〉]  
 〈必須パラメータリスト〉 := 〈空白を含まない文字列 (先頭は-以外)〉  
 〈オプションパラメータリスト〉 := {〈先頭が-の文字列 (空白を含む)〉 | “...” | ‘...’}  
 〈空白〉                 := {半角空白 (0x20) | タブ (0x09)}

〈必須パラメータ〉の例：

1234000

abcd

フォント

〈オプションパラメータ〉の例：

-w

-b 11 22 33

“abcd 1234”

‘ABCD¥n’

コマンドラインの例：

```

define_mem 10000 10000 BE WA
set -s8000.1 -b 30 31 32 33 "ABCD¥n"
rotate_bank exram -i5
  
```

必須パラメータ (4 個)

オプションパラメータ (3 個)

必須パラメータ

オプションパラメータ

〈ファイル名〉は Windows ファイルシステムのファイル名またはフルパスです。拡張子が指定されていない場合は .cms を追加してファイル名とします。ファイル名に空白 (0x20) が含まれる場合は 2 重引用符で囲む必要があります。

〈コマンド名〉と〈パラメータリスト〉の間には1つ以上の〈空白〉が必要です。  
数値を指定するパラメータ部位には ( ) で囲んだ数値式が使えます。以下に例を挙げます。

(1000 * 2)	これは 2000 と同じ
-b 11 (11*2) (50t + 8)	これは -b 11 22 58t と同じ

### (1) コマンド名

コマンド名は空白を含まない文字列です。

2 重引用符”で囲まれた文字列はファイル名となります。

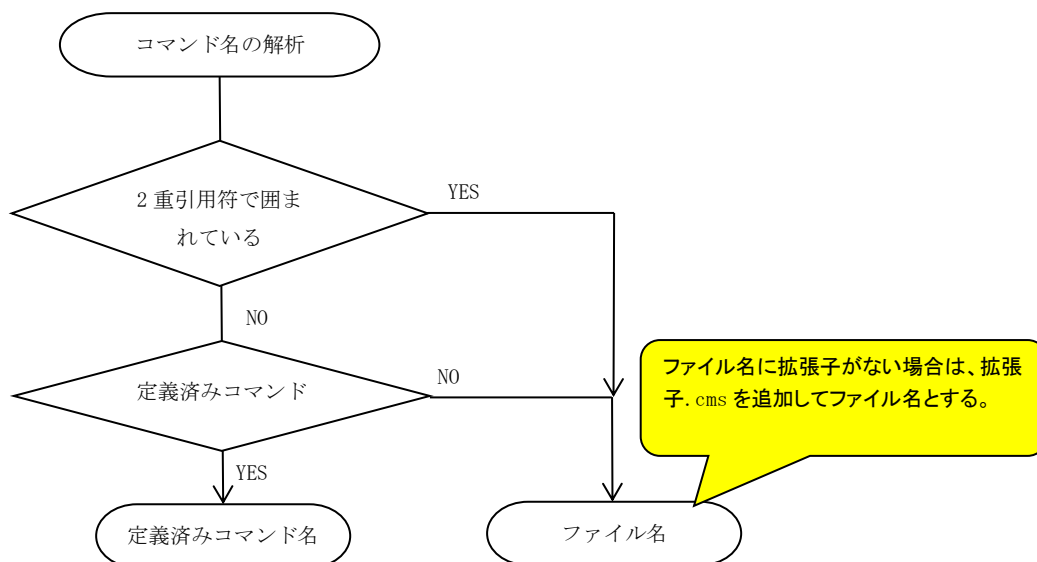
2 重引用符で囲まれていない文字列は定義済みコマンド名 (コンソール・コマンド名) となります。

定義済みコマンド名でない文字列はファイル名と見なします。

ファイル名に拡張子がない場合は .cms を拡張子として追加します。以下にファイル名の例を挙げます。

Test.txt	
Test	;Test.cms の省略形
テスト.txt	
“テスト 1.txt”	

コマンド名の解析手順を以下に示します。



### (2) 必須パラメータ

必須パラメータはコマンドの直後に〈空白〉で区切って指定します。以下は必須パラメータとして正しい形式となります。

1234000  
(1000+450)  
abcd  
フォント  
“123 xyz¥n”

必須パラメータはコマンドごとに指定する順番で意味が決まり、省略できません。

### (3) オプションパラメータ

オプションパラメータは<空白>に続く-（マイナス）で始まり、次のオプションパラメータの手前までとなります。-の次にパラメータの意味を示す文字列が続き、その後に複数の<空白>区切りパラメータが続きます。以下はオプションパラメータとして正しい形式となります。

```
-w
-b11 22 33
-b 11 22 33
-Dt="true"
-w 11 (1000+23) (45*3)
```

オプションパラメータの意味はコマンドごとに異なります。set, fill コマンドなどでは以下のように空白に続く引用符で囲まれた文字列も1つのオプションパラメータとなります。

```
"abcd 1234"
'ABCD¥n'
```

#### 6.1.2コマンドラインの解析

Cmtoy はマルチバイト文字セット (MBCS) を採用したアプリケーションなのでコマンドラインはシフト JIS コードの文字列で2バイトコード（全角文字）を含みます。

MBCS では、1 バイトあるいは 2 バイトのいずれかで文字が表現されます。2 バイト文字（全角文字）では、最初にくる“リード バイト”と、その後ろに続くバイトの両方で1つの文字と解釈されるようになっています。この最初のバイトは、リード バイト用に予約されている範囲の値をとります。Cmtoy では以下のリードバイトで全角文字と判定します。

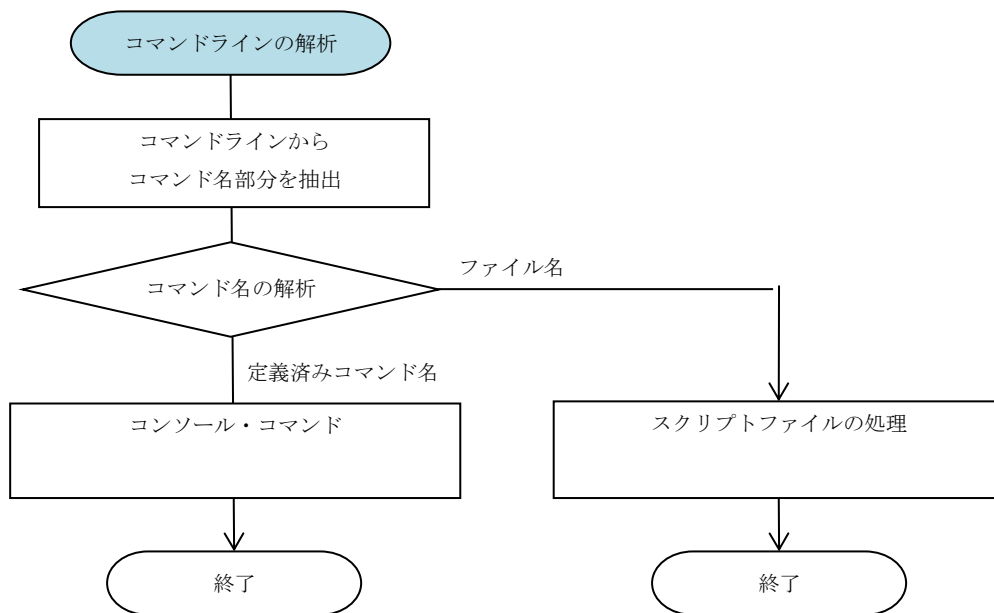
リードバイト	説明
81-9F	JIS X 0208
E0-EF	JIS X 0208
F0-FC	JIS X 0213 に拡張した Shift_JIS-2004

CLI はコマンドラインの先頭から1バイトずつ取り出して構文解析を行います。アスキーコードは1バイトで1文字ですが、以下のバイト列は不可分で1文字のように扱います。

- ・ 引用符(アスキーコードの'または")で囲まれた文字列
- ・ 全角文字（2バイト）

また、アスキーコード;（セミコロン）以降<改行>までは注釈（コメント）として扱います。

コマンドラインの解析実行の概略手順は以下のようになります。



定義済みコマンドもファイル名も見つからないとき、例えば aaa というコマンドを実行すると以下のようなエラーを出力ウインドウに表示します。

```
> aaa 10 f000 -f -b 11 22 33
;▲スクリプトファイルが見つかりません。"aaa.cms"
```

「test\_スクリプト.txt」というコマンドを実行すると以下のようなエラーを表示します。

```
> test_スクリプト.txt
;▲スクリプトファイルが見つかりません。"test_スクリプト.txt"
```

```
> "test スクリプト.txt"
;▲スクリプトファイルが見つかりません。"test スクリプト.txt"
```

### 6.1.3 コマンドパラメータ

コマンドパラメータは<空白>を含まない文字列（パラメータ要素）の並びとなります。前後を()で囲むと<空白>を含んでも1つの数値パラメータ要素となります。

ここでは以下のパラメータ要素の表記方法について説明します。

- ・ 識別子
- ・ 数値（数値リテラル）
- ・ 文字列（文字列リテラル）
- ・ メモリアドレス
- ・ ファイル名、ディレクトリ名
- ・ (数式)      数式の前を()で囲むと空白を含んでも1つのパラメータ要素となる。

※コンピュータプログラミング言語において「リテラル」とはソースコード内に値を直接表記したものを指す。これは実行時には毎回同じ効果をもたらす。ここではコマンドライン内に値を直接表記したものを指す。



### (1) 識別子

コマンドラインの中でユーザが定義した識別子を使って対象を指示することがあります。識別子には以下のものがあります。

識別子	説明
変数型マクロ名	<a href="#">define コマンド</a> で定義する
コマンドマクロ名	<a href="#">alias コマンド</a> で定義する
ターゲットメモリの領域名	<a href="#">add_mem_area</a> , <a href="#">add_permanent_area</a> , <a href="#">add_io_area</a> コマンドで定義する

識別子は<空白>を含まない文字列で、使用できる文字と規則は以下のとおりです。

- ・ アスキー（半角）文字の英数字 0～9、A～Z、a～z（大文字、小文字は区別する）
- ・ アスキー（半角）文字の\_（アンダースコア）
- ・ 全角文字（全角空白を除くすべて）
- ・ 先頭にアスキー（半角）文字の数字 0～9 は使えない

有効な識別子の例

Abc1    \_abc1    \_123    メモリ 1    拡張 RAM

不正な識別子の例

Abc1?    \_%123    123    メモリ.1    拡張<RAM>

### (2) 数値（数値リテラル）

コマンドパラメータの中でメモリアドレスやメモリへ書き込む値、バイト数などの数値を表記する場合があります。数値は以下の表記方法使ってパラメータに埋め込みます。

- ・ 16進数        :=        <英数字 0..9, A..F, a..f の並び>[ {H | h} ]
- ・ 10進数        :=        <数字 0..9 の並び> {T | t}
- ・ 2進数         :=        {0|1}<0, 1, \_の並び>\_

数値は 32 ビット整数で表現できる範囲となります。16 進数では 0x0000 0000～0xffff ffff、10 進数では 0～4, 294, 967, 295、符号付き 10 進数なら -2, 147, 483, 648～2, 147, 483, 647 となります。32 ビット値を符号付きか符号なしと解釈するかは、コマンドラインプログラムの意図で決まります。

16 進数の例を以下に挙げます。最後の H または h は省略できます。

1234    1234H    1234h    abcd    abcdH    abcdh    ff00    ff00H

10 進数の例を以下に挙げます。最後の T または t は必須です。

1234T    1234t

2 進数の例を以下に挙げます。先頭は 0 または 1 で最後は\_です。\_は途中にいくつあっても有効です。

00001010\_                      0010\_1100\_                      1010\_0000\_0000\_0001\_

### (3) 数式

数式は数値（数値リテラル）と演算子の並びで、評価の結果として 1 つの数値（32 ビット整数）となります。大小比較は符号なし 32 ビット整数として行います。

この結果の数値には論理値（真理値、真偽値）の場合も含みます。論理値の真(true)は数値 1、偽(false)は数値 0 となります。

使用できる演算子は以下の表を参照してください。優先度は C 言語を参考に決めました。ただし、後置演算子はありません。

優先度	演算子	機能
1		後置演算子
2	! ~ - + ++ --	前置の単項演算子
3	* / %	乗法、除法、剰余
4	+ -	加法、減法
5	<< >>	ビット単位のシフト
6	< <= > >=	大小比較
7	== !=	等価/非等価比較
8	&	ビット単位の AND
9	^	ビット単位の排他的 OR
10		ビット単位の通常の OR
11	&&	論理 AND
12		論理 OR

符号付きで比較する場合は定義済み関数型マクロを使ってください。[@scmpb\(\)](#), [@scmpw\(\)](#), [@scmpdw\(\)](#) を参照。

数式とその結果の例を以下に挙げます。

```

1+2<<3+4          ;0x180
(1+2)<<3+4          ;0x180
(1+2)<<(3+4)         ;0x180
1+(2<<3)+4          ;0x15

0 && !1 || 1         ;1(true)
(0 && !1) || 1        ;1(true)
0 && (!1 || 1)        ;0(false)

-1 > 0               ;1(true) 32 ビット符号なし整数として比較した結果
-1 == 0xffffffff     ;1(true) 32 ビット符号なし整数として比較した結果

```

演算子が空白を含まず連続した場合には、長い（2文字の）演算子を優先して構文解析を行います。以下に例を挙げます。

+++1 は ++(+1) と解釈するので結果は 2 となります。  
1+++2 は 1++(+2) と解釈するのでエラーとなります。  
1+ ++2 は 1+ (+2) と解釈するので結果は 4 となります。

```

> print @dword(+++1)      ;++(+1)
00000002
> print @dword(1+++2)     ;1++(+2)      エラーとなる。
?
> print @dword(1+ ++2)    ;1+ (+2)
00000004

```

数値を指定する代わりに()で囲んだ数式を指定できます。

例えば

```
-b (10+1) 22 (-33)      ;括弧()は必須
```

は

```
-b 11 22 FFFFFFFCD
```

と同じ意味になります。

#### (4) 文字列（文字列リテラル）

コマンドパラメータの中で任意の空白を含む文字列を表記する場合はアスキー（半角）の単一引用符、2重引用符で囲みます。

引用符で囲まれた文字列はC言語の定数文字列と似ています。引用符内にはエスケープシーケンスを使って特殊文字を含ませることができます。使用できるエスケープシーケンスを以下の表に示します。

エスケープシーケンス	文字コード (C言語表記)	説明
¥n	0x0a	LF、ラインフィード
¥r	0x0d	CR、キャリッジリターン
¥t	0x09	タブ
¥v	0x0b	垂直タブ
¥b	0x08	BS、バックスペース
¥f	0x0c	FF、フォームフィード（用紙送り）
¥a	0x07	ベル（警告）
¥¥	0x5c	バックスラッシュ、¥
¥?	0x3f	疑問符、?
¥'	0x27	単一引用符
¥"	0x22	2重引用符
¥0ooo	0ooo	8進数値、o=0..7
¥xhh	0xhh	16進数値、h=0..9, a..f, A..F
¥0	0x00	NUL文字

引用符で囲まれた文字列の例を以下に挙げます。

```
"123"      '123¥¥'      "abcd¥n"      'ABCD¥12345¥n'
"漢字"      '漢字'
"123 abc¥"%&"
'123 "abc¥' '
```

#### (5) メモリアドレス

ターゲットメモリの指定はメモリアドレスとバンク番号の組み合わせで行います。メモリアドレスの表記は以下の通りです。

＜アドレス>[.＜バンク番号>]

ここで＜アドレス>と＜バンク番号>は数値（数値リテラル）です。

メモリアドレスとバンク番号表記の例を挙げます。

```
0000      1000.1      1000H      0100H.2t      0100T.10T
(1000+100)      (1000+20).(1+2)
```

#### (6) ファイル名、ディレクトリ名

ファイル名、ディレクトリ名はWindowsファイルシステムの名前規則に従います。ファイル名、ディレクトリ名に空白が含まれる場合は2重引用符で囲む文字列（文字列リテラル）とします。

ファイル名の例を以下に挙げます。

```
app1.dll
E:¥
"c:¥Program Files (x86)¥TeraPad¥TeraPad.exe"
```

### 6.1.4 前処理（プリプロセス）

Cmtoy ではコマンドラインからコマンド名とパラメータ部を分離した後のパラメータ部の文字列に

対して前処理を行います。前処理は2段階に分けて行います。

- ① 変数型マクロの置換
- ② 定義済み関数型マクロの展開

この前処理を施したコマンドラインをコマンドハンドラーへ渡し実行します。

マクロ (macro) は「おおきい」とか「大規模な」という意味ですが、ここでいう「マクロ」とは、複数のコマンドパラメータ、コマンドパラメータ内の複数のパラメータ要素を1つの識別子で代表する機能です。

※C言語にもプリプロセッサ・マクロという機能があります。似ている部分もありますがそれと同じではありません。「[5.11 マクロ](#)」のマクロはC言語のプリプロセッサ・マクロを指します。

### (1) 変数型マクロの置換

変数型マクロの定義は [define コマンド](#) で行います。define コマンドはC言語のプリプロセッサ命令 #define と似ています。define コマンドの構文を以下に示します。

**define <変数型マクロ名> <置換文字列>**

例を以下に挙げます。

```
define バイト      12      ;バイト値
define ワード      1234    ;ワード値
define data        -b 11 22 33
```

ここで定義したマクロ名 **バイト**、**ワード**、**data** をコマンドライン内で参照するにはマクロ名の前に \$ をつけて記述します。以下に [print コマンド](#)、[fill コマンド](#) でこれらの変数を使う例を示します。

```
print A=$バイト B=$ワード ;1
fill -s0 $data             ;2
↓
print A=12 B=1234          ;1
fill -s0 -b 11 22 33      ;2
```

上記の2つのコマンドは前処理で下線部が以下の文字列に置き換えられて、コマンドハンドラーに渡されます。

### (2) 定義済み関数型マクロの展開

定義済み関数型マクロの構文は関数名、パラメータ部からなるC言語の関数に似ています。

**@<関数名>(<パラメータ 1>[, <パラメータ 2>[, …]])**

Cmtoy の定義済み関数型マクロは「[8. 定義済み関数型マクロ](#)」で説明します。

コマンドラインの中で定義済み関数型マクロを呼び出すには関数名の前に @ をつけて記述します。定義済み関数型マクロは前処理で文字列に置き換えます。以下に [fill コマンド](#) で定義済み関数型マクロ @inc() を使う例を示します。

```
fill -s0      -b @inc(5,10)      ;3
```

↓

```
fill -s0      -b 10 11 12 13 14      ;3
```

### (3) 前処理によるコマンドラインの変換

前処理は2段階で行われるので以下のコマンドラインは次のように変換されます。

```
print @inc(5,$バイト)      ;4
```



```
print @inc(5,12)           ;4
```



```
print 12 13 14 15 16      ;4
```

第1段階で変数型マクロを置換して以下のようになり、

第2段階で定義済み関数型マクロを展開し以下ようになります。

この最後の文字列がコマンドハンドラーに渡されます。前処理で；以降のコメント部分はそのまま残り、コメント部を含めてコマンドハンドラーに渡されます。

### (4) 前処理を迂回する (エスケープ)

コマンドラインの前処理では\$は変数型マクロの先導文字となり、@は定義済み関数型マクロの先導文字となります。この\$と@は前処理の段階では特別な機能文字となります。この機能文字を避けて (エスケープして) 通常の文字としてコマンドラインの中に残す場合は、\$と@の直前に¥を置きます。以下に例を示します。

```
print ¥$バイト=$バイト ¥$ワード=$ワード      ;5
print ¥@inc(5,10)=@inc(5,10)                  ;6
```



```
print $バイト=12 $ワード=1234      ;5
print @inc(5,10)=10 11 12 13 14 ;6
```

また、{}で囲まれた範囲は前処理を行わず、；(セミコロン) もコメント開始ではなくなります。例えば以下のコマンドラインは

```
print A=$バイト {C=$バイト; D=$ワード;}      ;7
```



```
print A=12 {C=$バイト; D=$ワード;}           ;7
```

## 6.1.5 コマンドマクロ

[alias コマンド](#)を使うと、複数のコマンドラインを1つのコマンドマクロ名に対応させることができます。alias コマンドの構文を以下に示します。

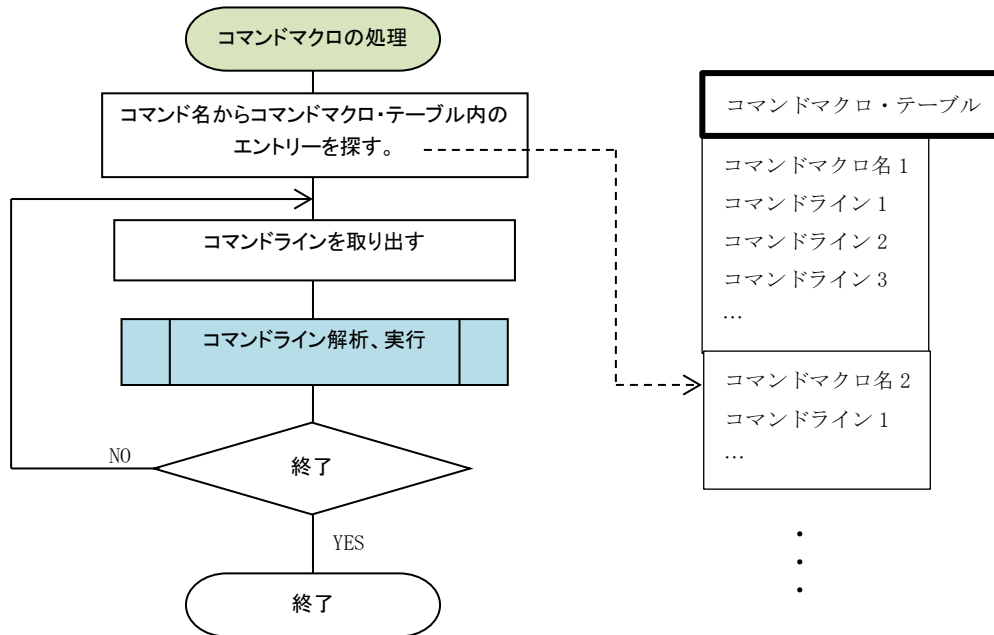
```
alias <コマンドマクロ名> {<コマンドライン1>; <コマンドライン2>; ...}
```

以下の例は2つのコマンドに1つのdmemというコマンドマクロを定義します。コマンドラインは；(セミコロン)で区切ります。{...}の中は前処理を行わないので、変数型マクロ名、関数型マクロ名はそのまま残ります。コマンドマクロを実行する段階で前処理を行い変数型マクロ名、関数型マ

クロ名は展開されます。

```
alias dmem {define mem 10000 1000 LE BA; add mem area reg 0 100 1 RW;}
```

この新しい dmem コマンドを実行すると定義された 2 つのコマンドラインを順番に実行します。  
alias コマンドで定義された dmem とコマンドラインはセットでコマンドマクロ・テーブルに格納されています。



alias コマンドでは既存の定義済みコマンド名も指定できます。CLI はコマンドマクロ名を先に探しますので、定義済みコマンドは実行できなくなりますが、コマンド名の前に¥を指定してコマンドマクロ名をエスケープすることができます。

変数型マクロ名を使った例を以下に示します。

```
define mem_size    10000
define io_size     1000
define area reg 0 100 1 RW
alias dmem {define_mem $mem_size $io_size LE BA; add_mem_area $area;}
```

この定義の後で dmem コマンドを実行すると以下ようになります。

```
> dmem
```

途中経過を表示したい場合は list -m コマンドを使います。

```
> list -m
> dmem
/dmem[1] > define_mem $mem_size $io_size LE BA
/dmem[2] > add_mem_area $area
```

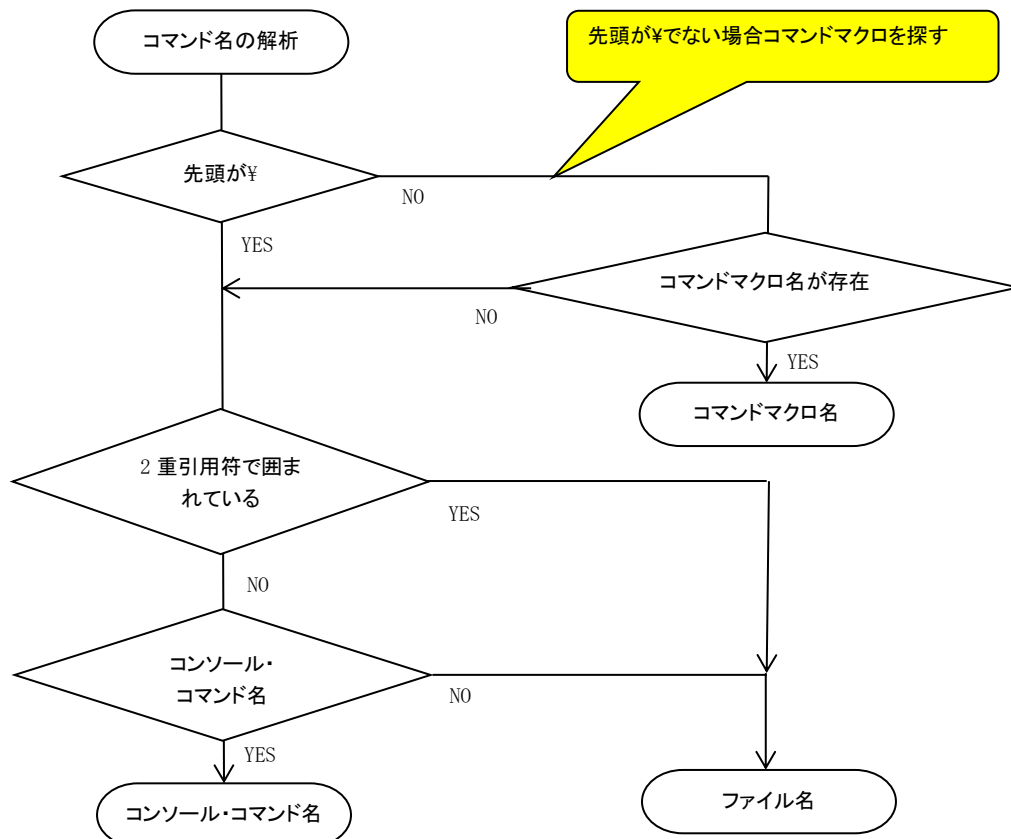
前処理の途中経過を表示したい場合は `list -p` コマンドを使用します。

```
> list -p
> dmem
;pp1>
;pp2>
/dmem[1] > define mem $mem size $io size LE BA
;pp1>+1 $mem_size => 10000
;pp1>+1 $io_size => 1000
;pp1> 10000 1000 LE BA
;pp2> 10000 1000 LE BA
;pp> define mem 10000 1000 LE BA
/dmem[2] > add_mem_area $area
;pp1>+1 $area => reg 0 100 1 RW
;pp1> reg 0 100 1 RW
;pp2> reg 0 100 1 RW
;pp> add_mem_area reg 0 100 1 RW
```

dmem のパラメータがないので空

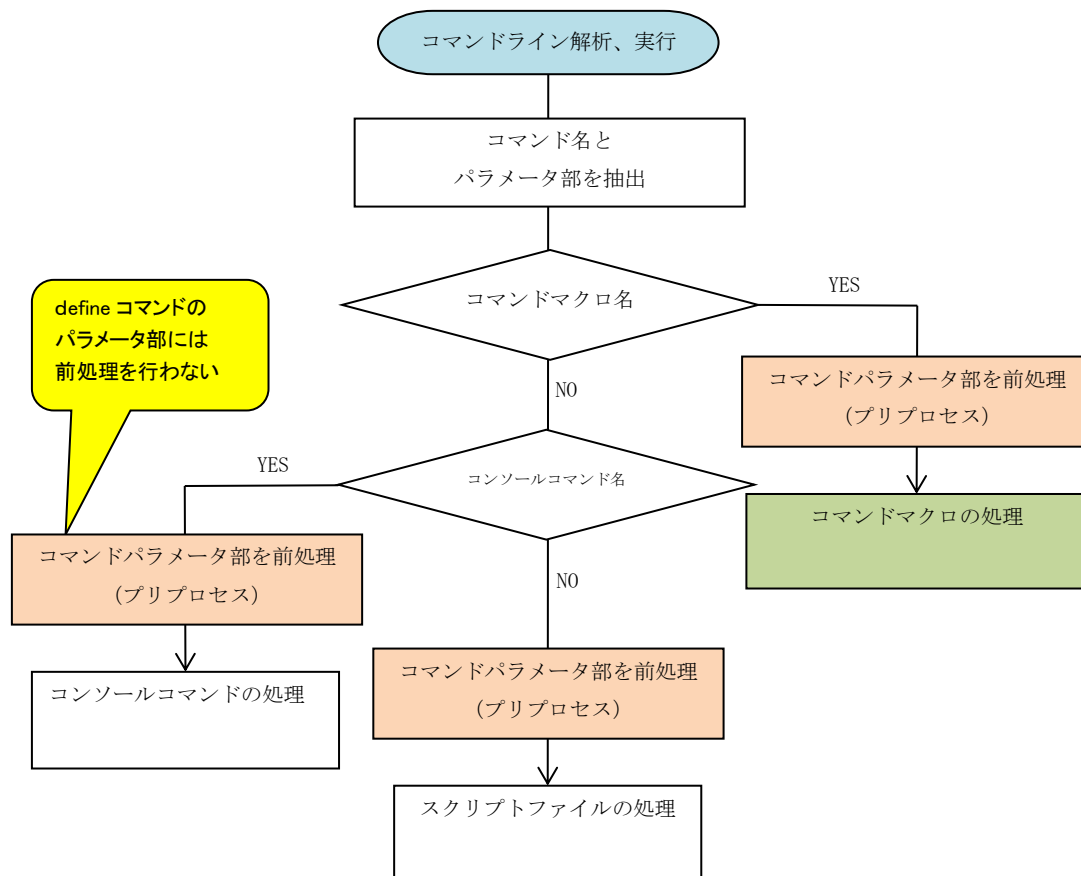
### 6.1.6 コマンドラインの解析、実行手順

コマンドマクロ機能を踏まえて最終的な CLI のコマンド名の解析手順は以下のようになります。



先頭が¥でない場合コマンドマクロを探す

最終的なコマンドラインの解析、実行処理は以下のようになります。



## 6.2 スクリプト機能

Cmtoy のスクリプトは一連のコマンド群をファイルに記述したテキストファイルです。1 行に 1 コマンドを記述します。空白を除いた行の先頭が#で始まる行はスクリプト制御命令となります。結局のところスクリプトは一連のコマンドを組み合わせる毎に同じ処理（バッチ処理）をするための機能です。

スクリプトファイルをコマンドラインから実行するには「[6.1.2 コマンドラインの解析](#)」で説明したようにファイル名を指定して定義済みコマンドと同様に行います。スクリプトの起動構文はコマンドと同じです。

1 行の最後は<改行>コードです。行末が¥<改行>で終わる行は継続行となり、次の行と連結して 1 行とします。以下に例を挙げます。

```
define      シンボル 3      11 22 33      ¥
                        44 55 66
```

この 2 行は連結されて、以下のような 1 行のコマンドラインとしてから解析、実行されます。

```
define      シンボル 3      11 22 33      44 55 66
```

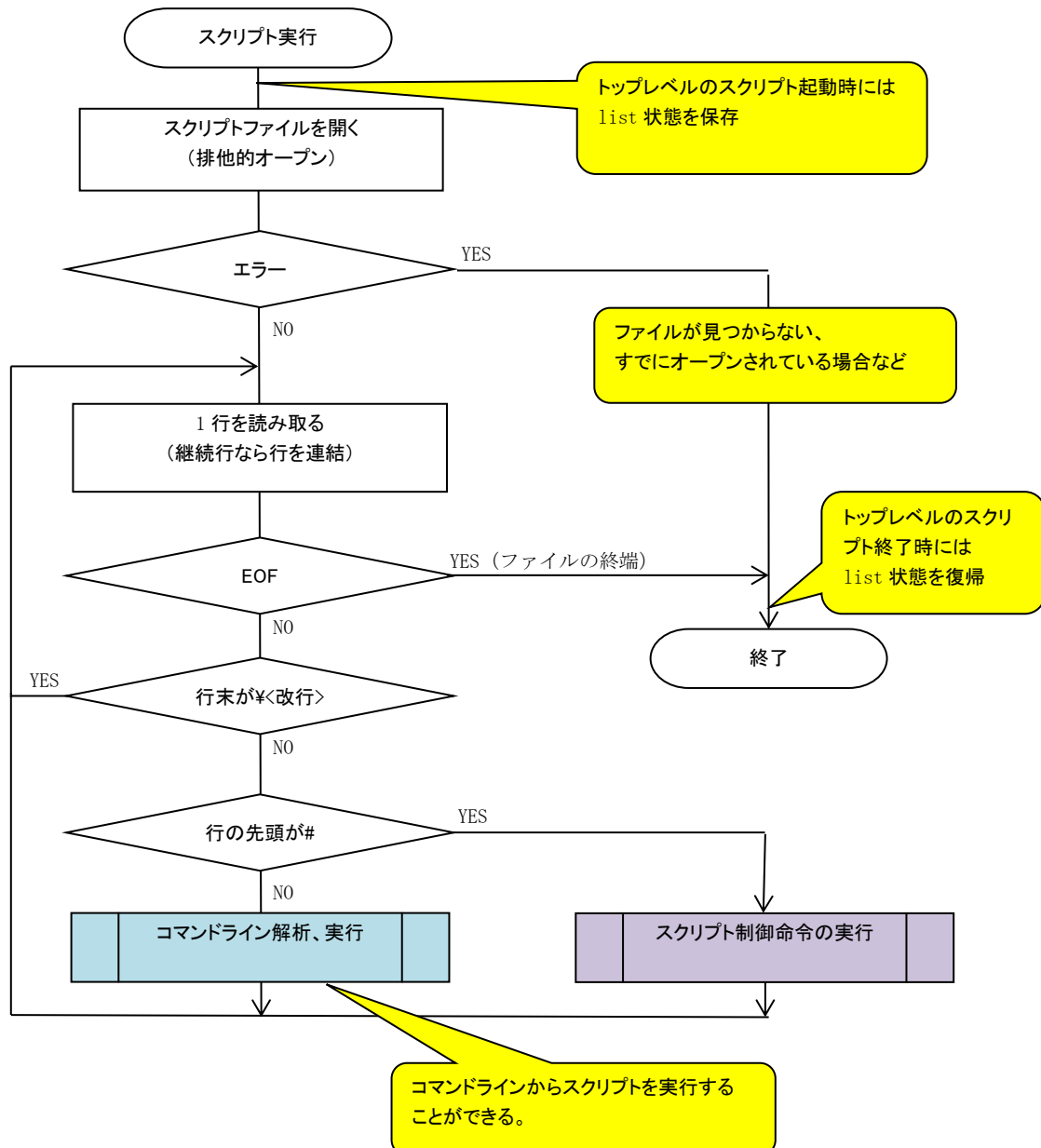
Cmtoy のスクリプトファイルはマルチバイト文字セット（MBCS）のテキストファイルです。従ってシフト JIS コードの 2 バイトコード（全角文字）を含みます。ただしテキストファイルの先頭に UTF8 の BOM(Byte Order Mark)がある場合は UTF8 からシフト JIS に変換してスクリプトファイルを解析



します。

※UTF8 の BOM は EF BB BF のバイト並びです。

スクリプトファイルはテキスト形式なので 1 行単位で読み込み以下の手順で解析、実行します。



### 6.2.1 スクリプト起動構文

Cmtoy の CLI は、定義済みコマンドでなければスクリプトファイルを探します。（「[6.1.1 コマンドラインの構文 \(シンタックス\)](#)」を参照）

以下のようにスクリプト起動時にオプションパラメータを指定できます。

<ファイル名> [<オプションパラメータリスト>] [ ;<注釈> ] <改行>

ファイル名に拡張子がない場合は拡張子 .cms をつけてファイルを探します。

〈オプションパラメータ〉 := {〈先頭が-の文字列（空白を含む）〉 | “...” | ‘...’ }

パラメータには定義済みオプションパラメータがあります。定義済みオプションパラメータはスクリプトの実行前に評価します。

定義済みオプションパラメータを以下に挙げます。

-D<name> [=<value>]	変数型マクロの定義
-U<name>	変数型マクロの削除
-L[{dpsm   a}]	コマンドの表示方法を指定、dpsmの中から文字組み合わせ
-S	コマンドの出力表示を停止
-F{SJIS   UTF8}	スクリプトファイルの文字符号化方式を指定

これらの定義済みオプションパラメータ以外に任意のオプションパラメータを指定できます。ただし、形式的にオプションパラメータの構文に沿っている必要があります。以下に例を挙げます。

```
-X 11 22 33
"1234"
'abc¥n'
```

スクリプトのオプションパラメータは保存され、スクリプト実行中にコマンドラインの中に定義済み関数型マクロ [@ipara\(\)](#)、[@spara\(\)](#) を使って埋め込むことができます。

## 6.2.2 定義済みオプションパラメータ

### (1) -D オプション

構文

```
-D<name> [=<value>]
```

説明

〈name〉	変数型マクロ名
〈value〉	変数型マクロを置換する文字列

変数型マクロを定義する。変数型マクロについては「[6.1.4 前処理（プリプロセス）](#)」を参照。

=の前後に空白を含むことはできない。〈value〉に空白を含めたい場合は、2重引用符”で囲む。以下に例を挙げる。

```
-Da1
-Da2=1234
-Da3="this is sample.¥n"
```

起動時の-D オプションは 20 個まで指定できる。

### (2) -U オプション

構文

```
-U<name>
```

説明

〈name〉	変数型マクロ名
--------	---------

変数型マクロを削除する。

### (3) -L オプション

構文

-L[{dpsm | a}]

説明

- L list コマンドと同じ効果
- Ld list -d コマンドと同じ効果
- Lp list -p コマンドと同じ効果
- Ls list -s コマンドと同じ効果
- Lm list -m コマンドと同じ効果
- La list -all コマンドと同じ効果

スクリプト実行前に [list コマンド](#) を実行する。dpsm から任意の文字の組み合わせで指定する。-Lds と指定した場合は list -d -s と同じ効果となる。（ [7.1.3 list \[-d\] | \[-p\] | \[-s\] | \[-m\] | \[-all\]](#) を参照 ）

### (1) -S オプション

構文

-S

説明

スクリプト実行前に [nolist コマンド](#) を実行する。

### (2) -F オプション

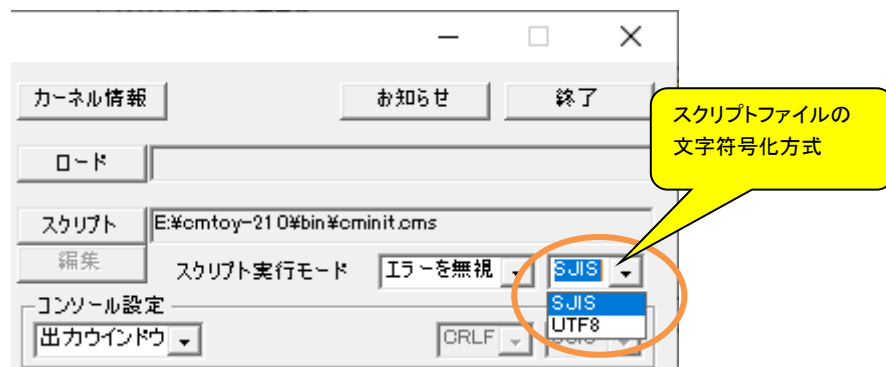
構文

-F{SJIS | UTF8}

説明

- SJIS シフト JIS として解析、実行する。
- UTF8 UTF8 からシフト JIS に変換してから解析、実行する。

スクリプトファイルの文字符号化方式（CES）を指定する。  
このオプションが省略された場合は、GUI の設定に従う。



ファイルの先頭に UTF8 の BOM(Byte Order Mark) が見つかればこの指定より優先して UTF8 とする。  
このとき GUI の設定も変わる。UTF8 の BOM は 0xEF 0xBB 0xBF の並び。

### 6.2.3 オプションパラメータを参照

スクリプトを起動する構文は以下のとおりです。

<ファイル名> [<オプションパラメータリスト>] [;<注釈>] <改行>

ファイル名に拡張子がない場合は拡張子 .cms をつけてファイルを探します。

<オプションパラメータ> := {<先頭が-の文字列 (空白を含む)> | “...” | ‘...’}

スクリプト内のコマンドラインにスクリプト起動時のオプションパラメータを取り込むことが可能です。そのために以下の定義済み関数型マクロを用意しています。

[@ipara\(\[index\[,sub\[,n\]\]\]\)](#)  
[@spara\(\[pattern\[,sub\[,n\]\]\]\)](#)

スクリプト script.cms を以下のように起動した場合について説明します。（CLI はスクリプト起動前にパラメータ部の前処理を行い、-L オプションの処理をします。）

script      -s8000.1      -b 30 31 32 33      “ABCD¥n”      -Ld

                                 1 番目                      2 番目                      3 番目                      4 番目

スクリプト script.cms 内に以下のコマンドラインを記述することができます。それぞれのコマンドラインは前処理を施されてからコマンドハンドラーに渡されます。

```
fill @ipara(1) @ipara(2) ;1
fill @ipara(3)           ;2
print @ipara()           ;3 すべてのパラメータを表示
```

前処理を施した結果この3行は以下のようになります。

```
fill -s8000.1 -b 30 31 32 33 ;1
fill “ABCD¥n”                ;2
print -s8000.1 -b 30 31 32 33 “ABCD¥n” -Ld ;3 すべてのパラメータを表示
```

GUI の「スクリプト」ボタンからスクリプトファイルを実行する場合はオプションパラメータを指定できません。

## 6.2.4 実行制御

スクリプトの実行はファイルの先頭行からファイルの最終行まで連続して実行します。それでは実行の柔軟性、記述の柔軟性がないので Cmtoy では以下のスクリプト実行制御機能を用意しています。

- ・途中で停止してオペレータの指示を待つ。
- ・条件により実行する行と読み飛ばす行を判別する。
- ・ファイルの途中でスクリプトを終了する。

スクリプト実行制御機能もスクリプトファイル内に記述します。空白を除く先頭が#で始まる行はスクリプト制御命令として解析、実行します。以下の制御命令があります。詳細は「[9 スクリプト制御機能](#)」を参照。

制御命令	説明
<a href="#">#exit</a> [<code>] [ <a href="#">-if</a> <数式>]	スクリプトファイルの終了
<a href="#">#abort</a> [ <a href="#">-if</a> <数式>]	スクリプトの終了。ネストされたスクリプトをすべて終了

<a href="#"><u>#break [-if&lt;数式&gt;]</u></a>	スクリプトの実行を停止して、「スクリプト実行モード」ダイアログボックスを表示する。
<a href="#"><u>#msgbox &lt;スタイル&gt; -t&lt;表題&gt; -m&lt;文字列&gt;</u></a>	スクリプトの実行を停止して、メッセージボックスを表示してオペレータの指示を待つ。
<a href="#"><u>#if &lt;数式&gt;</u></a>	<数式>が 0 以外なら次行以降、#else 行までを実行する。0 なら次行から#else まで、#else がなければ#endif 行まで読み飛ばす。
<a href="#"><u>#ifdef &lt;変数型マクロ名&gt;</u></a>	変数型マクロ名が定義されていれば#if 1 と同じ
<a href="#"><u>#ifndef &lt;変数型マクロ名&gt;</u></a>	変数型マクロ名が定義されていなければ#if 1 と同じ
<a href="#"><u>#else</u></a>	#if, #ifdef, #ifndef の実行状態を#endif 行まで反転する
<a href="#"><u>#endif</u></a>	#if, #ifdef, #ifndef の終了

## 7 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「[6.1.1 コマンドラインの構文（シンタックス）](#)」で説明したように以下の形式となります。

〈コマンド名〉 [〈パラメータ 1〉 [〈パラメータ 2〉 ... ]] [;〈注釈〉] 〈改行〉

ここでは各定義済みコマンドの構文の説明に、以下の記述形式を用います。

- ・ {A | B} では A または B のどちらかを指定する。
- ・ [A] では A は省略可能なパラメータ。
- ・ < > で囲んだものは仮パラメータ。パラメータの意味、属性を表すので、実行時には妥当な文字列に置き換える。
- ・ コマンドラインの実パラメータの一部として {} [] < > を使う場合は**太字**で表記する。
- ・ < > で囲んでいないパラメータ文字は実パラメータで、その通り指定する。大文字、小文字は区別しない。

仮パラメータに形式を指定する場合は < > の後に英数字、記号を添える。以下のような表記を使用します。

表記	説明
<>i	識別子
<>n	数値（1 6 進数、1 0 進数、2 進数表記）
<>e	数値式（数値のみの場合も含む）
<>ne	:= {<>n   ( <>e )} 数式は () で囲む。
<>a	メモリ/I/O アドレス。 := <アドレス>ne[.<バンク番号>ne] . の前後に空白は指定できない
<>s	空白を含まない文字列(;は含まない)
<>sp	空白を含む文字列(;は含まない)
<>q <>" <>'	引用符で囲んだ文字列。"... " または '...' 2 重引用符"で囲んだ文字列 引用符'で囲んだ文字列
<>f	Windows ファイルシステムのファイル名、ディレクトリ名の形式。空白を含む場合は、"（2 重引用符）で囲む := {<>s   <>"}
<>l	コマンドラインの文字列。コメント部は除く。 := <コマンド名> [<パラメータ 1> [<パラメータ 2> ... ]]
<>p	コマンドの 1 つのパラメータ形式の文字列 := {<必須パラメータ>   <オプションパラメータ>}
<>m	指定された位置から後ろの全パラメータ部の文字列。コメント部を除く。 := [<パラメータ>p [<パラメータ>p ... ]]
<>b	空白を含む文字列の並び。文字列の配列。前後を {} で囲み、文字列の区切りは、(セミコロン) 。 := {<>l; <>l; ...} := {<>m; <>m; ...} := {<>s; <>s; ...} := {<>sp; <>sp; ...}
<>c	指定された文字集合からの文字の組み合わせ

V3.00 で構文を変更したコマンドを以下に挙げます。

コマンド	変更点
<a href="#">setswitch</a>	パラメータの区切りを, (カンマ) から空白に
<a href="#">fill_bank</a> <a href="#">set_bank</a> <a href="#">copy_bank</a>	<領域名>と<バンク番号>の区切りを, (カンマ) から. (ピリオド) に
<a href="#">set</a> <a href="#">get</a> <a href="#">wait</a>	<アドレス>と<バンク番号>の区切りを, (カンマ) から. (ピリオド) に
<a href="#">serial &lt;シリアル ポート番号&gt; push</a>	ブレークキャラクター受信を-b から sb に変更 受信エラーを-e から se に変更 -c オプションパラメータを追加

V3.00 で追加したコマンドを以下に挙げます。

<a href="#">ver</a>	CLI のバージョンを表示する
<a href="#">nolist</a>	出力ウィンドウへの表示を止める
<a href="#">list</a>	コマンドが表示する情報を制御する
<a href="#">clear</a>	出力ウィンドウをクリア
<a href="#">print</a>	出力ウィンドウへメッセージを表示する
<a href="#">trace</a>	出力ウィンドウへメッセージを表示する
<a href="#">help</a>	コマンドのシンタックスを表示する
<a href="#">edit</a>	ファイルをテキストエディタで開く
<a href="#">cd</a>	カレントディレクトリを変更
<a href="#">dir</a>	ディレクトリのファイル名を表示
<a href="#">define</a>	前処理前の文字列を使って変数型マクロを定義
<a href="#">define input</a>	GUI 入力から変数型マクロを定義
<a href="#">define literal</a>	前処理後の文字列を使って変数型マクロを定義
<a href="#">undef</a>	変数型マクロを削除
<a href="#">alias</a>	コマンドマクロを定義
<a href="#">unalias</a>	コマンドマクロを削除
<a href="#">include</a>	スクリプトファイルを指定して実行
<a href="#">setcon</a>	コンソールの属性を設定
<a href="#">chcon</a>	コンソールを切替える
<a href="#">irc</a>	割込みコントローラを設定、表示
<a href="#">poke</a>	メモリ/I/O の内容进行操作
<a href="#">tmem</a>	ターゲットメモリの情報を表示する
<a href="#">calc</a>	式の評価
<a href="#">sum</a>	メモリの和を計算
<a href="#">crc</a>	CRC の計算
<a href="#">set title</a>	ウィンドウのタイトルバーへ文字列を追加する

V3.00 で廃止したコマンドを以下に挙げます。

messagebox

## 7.1 CLI 操作

### 7.1.1 ver

#### パラメータ

なし

#### 説明

CLI のバージョンを表示する。

#### 例

```
> ver
CLI Version 03.00.0010
```

### 7.1.2 nolist

#### パラメータ

なし

#### 説明

コマンドはコンソールにメッセージを表示しない。[trace コマンド](#)はこの状態でもコンソールの指定にかかわらず出力ウィンドウへ表示する。

### 7.1.3 list [-d] [[ -p] [[ -s] | [-m] [[ -all]]

#### パラメータ

-d	コマンドの付加情報を表示する。
-p	前処理の付加情報を表示する。
-s	スクリプト実行時の付加情報を表示する。
-m	コマンドマクロの付加情報を表示する。
-all	すべての付加情報を表示する。-d -p -s -m を指定した場合と同じ。

#### 説明

コマンドが出力ウィンドウに出力するメッセージを制御する。オプションパラメータは組み合わせて使用できる。オプションパラメータをすべて省略するとデフォルト状態に戻る。

#### 例

以下にコマンド実行時の追加情報の表示例を示します。

コマンド実行時の追加情報の例。

```
> list -d ;コマンドの付加情報を表示
> define mem_size 10000
> define mem_size 20000
;warning: Symbol redefined. "mem_size".
> define _mem $mem_size 10000 BE WA
;ターゲットメモリを生成しました。ビッグエンディアン, バイトアドレッシング
;メモリ空間=20000H, IO 空間=10000H
> add_mem_area com_io 8000 2000 2 RW
;メモリ領域を定義しました。
```



```

> fill -s8000.1 -l 33 -w 44 -b 66 "sample"
;filled at 8000H (1 double-words)
;filled at 8004H (1 words)
;filled at 8006H (1 bytes)
;filled at 8007H (6 bytes)
;fill を完了 : next address = mem : 800DH.1
> get -s8000.1 -b10
00 00 00 33 00 44 66 73 61 6d 70 6c 65 01 01 01
;get を完了 : next address = mem : 8010H.1 (16 bytes read.)

```

コマンド実行時の前処理の追加情報の例。

```

> list -p ;前処理の情報を追加表示
> print $mem size
;pp1>+1 $mem_size => 20000
;pp1> 20000
;pp2> 20000
;pp> print 20000
20000

```

スクリプト制御命令の追加情報の例。

```

> list -s ;スクリプト制御命令の情報を追加表示
> #if 1
> print true
true
> #else
> ;- print false
> ;- #endif

```

コマンドマクロ実行時の追加情報の例。

```

> list -m ;コマンドマクロの情報を追加表示
> alias a {alias #1 #2}
> a dir {¥dir #1 -w}
/a[0] > alias dir {¥dir #1 -w}
> dir e:¥cmttoy-300
/dir[0] > ¥dir e:¥cmttoy-300 -w
;ディレクトリ e:¥cmttoy-300¥

```

```

[.]          [..]          [bin]          [doc]          [include]    [LIB]
[mITRON]     [tools]
README.TXT
8 directories and 1 files found.

```

#### 7.1.4cd [<path>f]

##### パラメータ

<path>f Windows ファイルシステムのディレクトリ名。ディレクトリ名に空白を含む場合は2重引用符 “で囲む。

##### 説明

カレントディレクトリを変更する。<path>が省略された場合はカレントディレクトリを表示する。

### 7.1.5dir [<path>f] [-d] [-w] [-f]

#### パラメータ

<path>f	Windows ファイルシステムのディレクトリ名。ディレクトリ名に空白を含む場合は2重引用符“で囲む。
-d	ディレクトリのみを表示する
-w	ファイル名、ディレクトリ名を表示する。
-f	ファイル名、ディレクトリ名とタイムスタンプを表示する。

#### 説明

<path>で指定したカレントディレクトリ内のファイルを表示する。<path>が省略された場合カレントディレクトリの内容を表示する。

### 7.1.6define [<symbol>i [<string>m]]

#### パラメータ

<symbol>i	変数型マクロ名
<string>m	置換文字列

#### 説明

変数型マクロを定義する。

<string>を省略すると空文字列 (NUL 文字列) となる。

<symbol>と<string>を省略した場合は、定義した変数型マクロの一覧を表示する。一覧には Cmtoy 起動時の-D オプションで定義した変数型マクロも含めて表示する。

すでに変数型マクロ名<symbol>が存在した場合は、エラーにせず再定義する。

**define コマンドのパラメータ部に前処理は適用されない。**

コマンドパラメータ部で変数型マクロを参照する場合は\$文字を先導文字として\$<symbol>と記述する。

#### 例

以下は変数型マクロを3個定義して、定義済み変数型マクロ一覧を表示する例です。

```
> define mem_size 10000
> define io_size 1000
> define 領域1      reg 0 100 1 RW
> define
;number of defined symbols = 3
;1 : mem_size = 10000
;2 : io_size = 1000
;3 : 領域1 = reg 0 100 1 RW
```

変数型マクロは定義し直してもエラーにはなりません。以下では io\_size を再定義します。このとき list -d の状態で警告(warning)を表示します。

```
> list -d
> define io_size 100
;warning: Symbol redefined. "io_size".
```

定義した変数型マクロをコマンドパラメータ内で使用するには先頭に\$をつけて参照します。以下では定義済みの変数型マクロを [defien\\_mem コマンド](#) で使用します。

```
> define_mem $mem_size $io_size LE BA
;ターゲットメモリを生成しました。リトルエンディアン, バイトアドレッシング
;メモリ空間=10000H, IO 空間=100H
```

変数型マクロは define コマンドでは展開されません。

```
> define ターゲットメモリ      $mem_size $io_size LE BA      ;前処理はなし
```

```
> define_mem          $ターゲットメモリ          ;前処理で変数型マクロが展開される
;ターゲットメモリを生成しました。リトルエンディアン, バイトアドレッシング
;メモリ空間=10000H, IO 空間=100H
```

### 7.1.7define\_literal [<symbol>i [<string>m]]

#### パラメータ

<symbol>i           変数型マクロ名  
<string>m           置換文字列

#### 説明

変数型マクロを定義する。**define** コマンドとの違いは前処理が施された結果の文字列が置換文字列となる。

<string>を省略すると空文字列 (NUL 文字列) となる。

すでに変数型マクロ名<symbol>が存在した場合は、エラーにせず再定義する。

コマンドパラメータ部で変数型マクロを参照する場合は\$文字を先導文字として\$<symbol>と記述する。

#### 例

```
> undef -all
> add_mem_area      ram      8000 2000 2 RW
> set -s 8000 -b @inc(10, 0)
> peek -s8002 -l
;mem : 8002H.0 = 05040302
> define_literal snap_shot $_R
> define
;number of defined symbols = 1
;l : snap_shot = 05040302
```

### 7.1.8define\_input <symbol>i [<string array>b] [-m<説明>m]

#### パラメータ

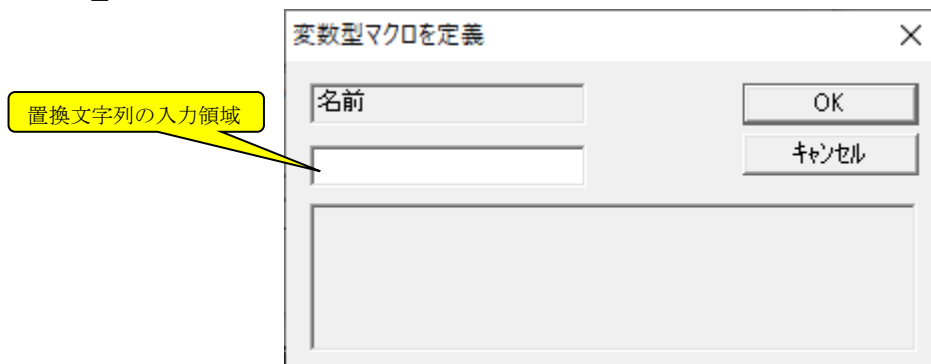
<symbol>i           変数型マクロ名  
<string array>b    置換文字列の配列。;(セミコロン)で区切る  
                  :=<string>m; <string>m;...}  
-m<説明>m

#### 説明

以下のダイアログボックスを表示して変数型マクロを定義する。

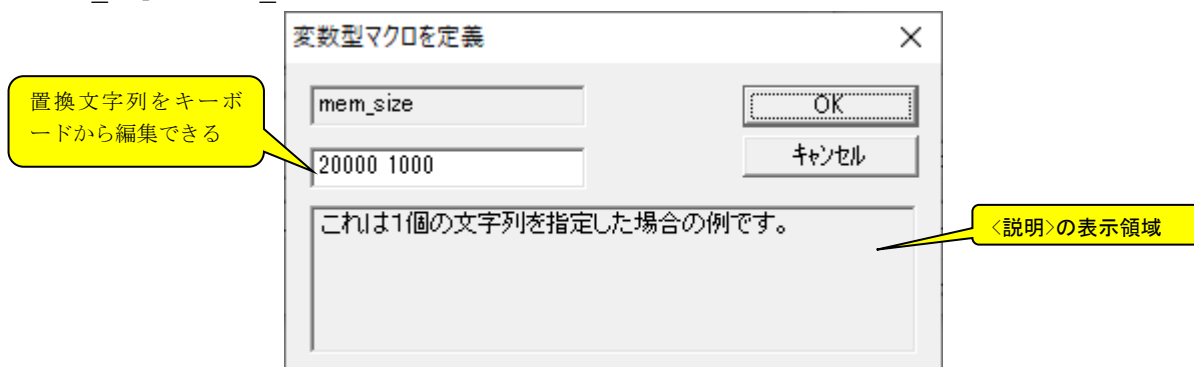
<string array>b が省略された場合、<string array>b が空配列の場合は、以下のダイアログボックスを表示する。

```
define_input 名前
define_input 名前 {}
```



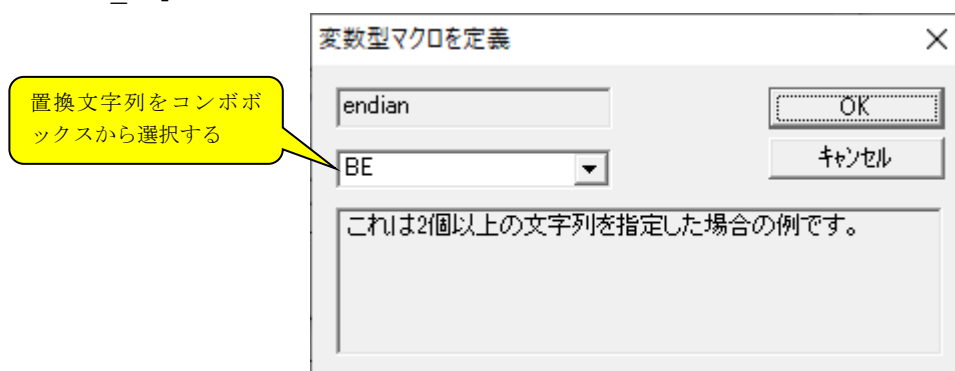
<string array>b で1個の文字列を指定した場合は、以下のダイアログボックスを表示する。

`define_input mem_size {20000 1000} -m` これは1個の文字列を指定した場合の例です。



<string array>b で2個以上の文字列を指定した場合は、以下のダイアログボックスを表示する。

`define_input endian {BE; LE;} -m` これは2個以上の文字列を指定した場合の例です。



ここで「OK」ボタンで終了すると、変数型マクロ `endian` が定義される。「キャンセル」ボタンで終了すると変数型マクロ `endian` は定義されない。

### 7.1.9 `undef [<symbol>i ...] [-all] [-init]`

#### パラメータ

<symbol>i	変数型マクロ名
-all	すべての変数型マクロを削除する。
-init	すべての変数型マクロを削除してから、Cmtoy 起動時に-D オプションで定義した変数型マクロを再定義する。

#### 説明

<symbol>列で指定した変数型マクロを削除する。

#### 例

以下は変数型マクロ `mem_size` と `io_size` を削除する例です。

```
> undef mem_size io_size
```

### 7.1.10 `alias [<名前>i [<command lines>b]] [-np] [-f]`

#### パラメータ

<名前>i	コマンドマクロ名
-------	----------

<command lines>b コマンドラインの列。前後を {} で囲み、コマンドラインの区切りは;。  
 -np コマンドマクロ<名前>i の実行時にパラメータ部の前処理をしない  
 -f <名前>i がすでに存在すれば削除してから定義する

## 説明

コマンドマクロを定義する。コマンドマクロは複数のコマンドラインをまとめて1つのコマンドマクロ名に割り当てる機能。

コマンドラインの列は以下のように指定する。

<command lines>b := {<command line1>l; <command line2>l; ...}

ここでの {} はこの通りに指定する。

コマンドマクロ名はコマンド名と同じように使える。構文は以下のようにコマンドの構文と同じ。

<コマンドマクロ名> [<パラメータ 1>p [<パラメータ 2>p ... ]]

ここで指定された n 番目の<パラメータ n>p を<command line>l 内で参照するには前後を空白で区切った#n を使う。**n は 10 進数 (最後に t または T は付けない)**。

#n n 番目以降のパラメータを参照。n=0 はコマンドマクロ名を参照。

#n\* n 番目以降のパラメータすべてを参照。

#\* すべてのパラメータを参照。#1\*と同じ。

コンソールコマンド名を短い名前 (別名) として定義することもできる。コンソールコマンド名と同じコマンドマクロ名を定義することもできる。コマンドを実行する際に先頭に¥をつける (コマンドマクロを避ける) と常にコンソールコマンド名と見なす。

例えば、dir コマンドを実行したときに dir -w と実行したい場合は以下のコマンドマクロを定義する。

```
alias dir {¥dir #1 -w}
```

すでにコマンドマクロ名<名前>が存在した場合は、エラーとなる。再定義する場合は unalias コマンドで一度削除する必要がある。-f オプションを指定した場合はエラーにせずに再定義する。

「[2.12.5 ショートカット・ボタン](#)」で説明したようにコマンドマクロをコマンド・コンソールのショートカット・ボタンで実行できる。そのためには以下のコマンドマクロ名を使ってコマンドマクロを定義する。

\_F1    \_F2    \_F3    \_F4

ショートカット・ボタンからスクリプトを実行するためには以下のようなコマンドマクロを定義します。

```
alias _F1 {<スクリプトファイル名> パラメータリスト;}
```

## 例

以下は define コマンドと alias コマンドの別名 d と a を定義して、定義済みコマンドマクロ一覧を表示する例です。

```
> alias d {define #1 #2*} -np
> alias a {alias #1 #2}
> alias
;number of command macros = 2
d = {define #1 #2*; } : stop preprocess
a = {alias #1 #2; }
```

すでにコマンドマクロ名が存在した場合は、エラーとなります。

```
> alias a {alias #1 #2}
```

;▲コマンドマクロ名が既に存在します。"a"

定義したコマンドマクロ d と a を使用する場合の例です。コマンドマクロ d では指定されたパラメータをそのまま（前処理をしないで）define コマンドに渡す必要があります。そのためコマンドマクロ d を定義する alias コマンドでオプション-np を指定しておく必要があります。

```
> d mem_size      10000
> d io_size 1000
> d mem_space     $mem_size $io_size
> d
;number of defined symbols = 3
;1 : mem_size = 10000
;2 : io_size = 1000
;3 : mem_space = $mem_size $io_size
> a a
a = {alias #1 #2; }
```

コマンドマクロの実行時に指定されたパラメータを参照する方法の例です。まず以下のコマンドマクロ pcheck を定義します。

```
> alias pcheck      {print #0= #0, #1= #1, #2= #2, #3= #3, #4= #4,
#5= #5;}
```

1 番目のパラメータを参照する（前後は空白）

このコマンドマクロ pcheck にパラメータを指定して実行するとき、どのようにパラメータを参照するかいくつかの例を挙げます。

```
> pcheck
#0= pcheck , #1= , #2= , #3= , #4= , #5=
> pcheck 123 abc "xyz" -b 11 22 33 -i5
#0= pcheck , #1= 123 , #2= abc , #3= "xyz" , #4= -b 11 22 33 ,
#5= -i5
```

コマンドマクロ pcheck に変数型マクロ、関数型マクロを指定した場合は以下のようにになります。これらは前処理で展開された後の文字列を参照します。

```
> d mem      $mem_size $io_size
> list -m
> pcheck @inc(3, 10) $mem
/pcheck[1] > print #0= pcheck , #1= 10 , #2= 11 , #3= 12 , #4=
10000 , #5= 1000
#0= pcheck , #1= 10 , #2= 11 , #3= 12 , #4= 10000 , #5= 1000
```

ここで@inc(3, 10)と\$memを前処理で展開せずに print コマンドに渡す場合は、以下のように¥記号を使って前処理をエスケープします。

```
> list -m
> pcheck ¥@inc(3, 10) ¥$mem
/pcheck[1] > print #0= pcheck , #1= @inc(3, 10) , #2= $mem , #3= ,
#4= , #5=
#0= pcheck , #1= 10 11 12 , #2= 10000 1000 , #3= , #4= , #5=
```

コマンドのオプションパラメータ形式(-で始まる一連の文字列)を使うと以下のようにになります。

```
> pcheck -b @inc(3, 10) -x $mem_size
#0= pcheck , #1= -b 10 11 12 , #2= -x 10000 , #3= , #4= , #5=
```

まず、以下のようなコマンドマクロ 'テスト'を定義して、'テスト'に様々なパラメータを渡して実行するスクリプトファイルを作成します。

```
alias テスト {print #0= #0;      ¥
               print #1= #1; ¥
               print #2= #2; ¥
               print #3= #3; ¥
               print #4= #4; ¥
               print #5= #5;}

テスト
テスト 123 abc "xyz" -b 11 22 33 -i5
テスト $mem_size $io_size
テスト @inc(3, 10) $mem
テスト ¥@inc(3, 10) ¥$mem
テスト -b @inc(3, 10) -x $mem_size
```

スクリプトファイルとする

このスクリプトファイルの実行結果のうちコマンドマクロ'テスト'の実行部分は以下ようになります。

```
> テスト
#0= テスト
#1=
#2=
#3=
#4=
#5=

> テスト 123 abc "xyz" -b 11 22 33 -i5
#0= テスト
#1= 123
#2= abc
#3= "xyz"
#4= -b 11 22 33
#5= -i5

> テスト $mem_size $io_size
#0= テスト
#1= 10000
#2= 1000
#3=
#4=
#5=

> テスト @inc(3, 10) $mem
#0= テスト
#1= 10
#2= 11
#3= 12
#4= 10000
#5= 1000

> テスト ¥@inc(3, 10) ¥$mem
#0= テスト
#1= 10 11 12
#2= 10000 1000
#3=
#4=
#5=

> テスト -b @inc(3, 10) -x $mem_size
#0= テスト
```

前処理で「10 11 12 10000 1000」となる

前処理で「@inc(3, 10) \$mem」となる

```
#1= -b 10 11 12
#2= -x 10000
#3=
#4=
#5=
```

### 7.1.11 unalias [[<名前>i] ...] [-all]

#### パラメータ

<名前>i                    コマンドマクロ名  
-all                        すべてのコマンドマクロを削除する。

#### 説明

<名前>i 列で指定したコマンドマクロを削除する。

### 7.1.12 include <filename>f [-D<name>=<value> ] [-U<name>] [-F{SJIS | UTF8}] {[-L[dpsma]] | [-S]} [<任意のオプションパラメータ列>]

### 7.1.13 include <command lines>b [-D<name>=<value> ] [-U<name>] {[-L[dpsma]] | [-S]} [<任意のオプションパラメータ列>]

#### パラメータ

<filename>f                ファイル名  
<command lines>b          コマンドラインの列。前後を {} で囲む。コマンドラインの区切りは;。  
-D<name>=<value>          変数型マクロを定義する。複数指定可。  
-U<name>                    変数型マクロを削除する。複数指定可。  
-F{SJIS | UTF8}            スクリプトファイルの文字符号化方式 (CES)  
-L[dpsma]                  list コマンドに対応。dpsma はこの中から複数取り出して指定する。  
-S                          nolist コマンドに対応。  
<任意のオプションパラメータ列>   先頭が-で始まるオプションパラメータ形式の文字列

#### 説明

1 番目の構文ではファイル名を指定してスクリプトを実行する。**拡張子は省略できない。**

2 番目の構文では、<command lines>b を一時ファイルに保存してからスクリプトファイルとして実行する。<command lines>b にはコンソールコマンド、コマンドマクロ、スクリプト制御命令を;で区切って記述する。コマンド実行後一時ファイルは削除する。

どちらもスクリプトの定義済みオプションパラメータが使用できる。（「[6.2 スクリプト機能](#)」を参照）

#### 例

まず、以下のようなスクリプトファイルを作成します。

```
list -s                    ;スクリプト制御の追加情報を表示
define test {              ¥
    print p1 is @ipara(1), p2 is @ipara(2);   ¥
    #if $条件 == 2 ;        ¥
        print 真;           ¥
    #else ;                  ¥
        print 偽;           ¥
    #endif;                  ¥
define;                    ¥
```

スクリプトファイルとする



```

    }
include $test -D 条件=2 -DTEST3="xyz 123"

```

このスクリプトファイルの実行結果のうち include コマンドの実行部は以下のようになります。

```

> include $test -D 条件=2 -DTEST3="xyz 123"
>> print p1 is @ipara(1), p2 is @ipara(2)
p1 is -D 条件=2, p2 is -DTEST3="xyz 123"
>> #if $条件 == 2
>> print 真
真
>> #else
>> ;- print 偽
>> ;- #endif

```

#else 節はスキップ  
list -s で表示

#### 7.1.14 set\_script\_mode { I | E | S }

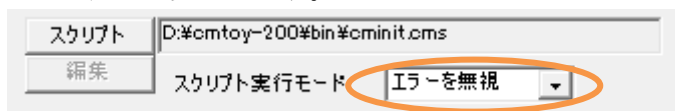
##### パラメータ

I	エラーを無視
E	エラーで停止
S	1 行ずつ実行

##### 説明

スクリプトの実行モードを設定する。「[2.13.1 スクリプトの実行モード](#)」を参照。

※ GUI の表示も変わります。



#### 7.1.15 chcon [<コンソール番号>ne]

##### パラメータ

<コンソール番号>ne	コンソール番号
1	GUI の出力ウィンドウとコンソール
2	TCP/IP 端末

##### 説明

コンソールを変更する。GUI の表示も変わる。

コンソール番号を省略した場合は、現在のコンソール名とコンソール番号を表示する。

※ 「[2.16 コンソール端末](#)」を参照してください。

##### 例

```

> chcon
;現在のコンソール = 出力ウィンドウ (1)

```

#### 7.1.16 setcon <コンソール番号>ne [-e<ces>] [-n<newline>]

##### パラメータ

<コンソール番号>ne	コンソール番号
1	GUI の出力ウィンドウとコンソール
2	TCP/IP 端末

-e<ces>            文字符号化方式  
-n<newline>       改行コード

#### 説明

コンソールを設定する。文字符号化方式と改行コードを変更する。GUI の表示も変わる。  
<コンソール番号>s のみを指定した場合は現在の設定を表示する。  
コンソール番号=1 の設定は変更できない。

改行コードは以下の中から指定する。

CRLF            CR (キャリッジリターン) と LF (ラインフィード)

CR              CR (キャリッジリターン) のみ

LF              LF (ラインフィード) のみ

文字符号化方式は以下の中から指定する。

SJIS            シフト JIS

UTF8            UTF8 (Unicode Transformation Format-8)

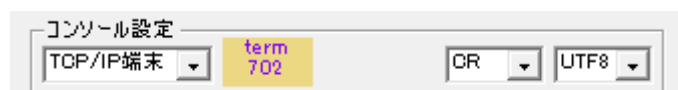
コンソール番号のみを指定すると、現在の設定を表示する。

※ 「[2.16 コンソール端末](#)」を参照してください。

#### 例

```
> setcon 1
;コンソール設定: 出力ウインドウ => SJIS, CRLF
> setcon 2
;コンソール設定: TCP/IP 端末 => SJIS, CR
> setcon 2 -eUTF8
;コンソール設定: TCP/IP 端末 => UTF8, CR
> setcon 2
;コンソール設定: TCP/IP 端末 => UTF8, CR
> chcon 2
```

この結果 GUI の表示は以下のようになります。



## 7.2 ターゲットデバイス操作

### 7.2.1 load <ファイル名>f

#### パラメータ

<ファイル名>f    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 2 重引用符 “で囲む)

#### 説明

μ ITRON 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、以下のように記述します。

```
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。

ん。Windows が Cmtoy のユーザプロセス空間内に配置します。

### 7.2.2 reset [-t[<回数>ne]]

#### パラメータ

-t[<回数>ne]      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指定。

#### 説明

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
```

### 7.2.3 int <レベル 1>ne [<レベル 2>ne]

#### パラメータ

<レベル 1>ne      IRC の割込みレベル(0～15)。

<レベル 2>ne      IRC の割込みレベル(0～15)。

#### 説明

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。

### 7.2.4 set\_interrupt\_name <レベル>ne [<表示名>s]

#### パラメータ

<レベル>ne      IRC の割込みレベル(1～7)。

<表示名>s      文字列（空白は含まないこと）

#### 説明

<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。

### 7.2.5 timerlog {ON | OFF}

#### パラメータ

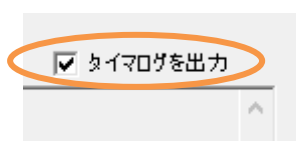
ON                  タイマハンドラの起動ログの出力を開始する

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

#### 説明

初期値は ON。

対応する GUI の表示を以下に示す。



### 7.2.6 timer [<回数>ne] [-s]

#### パラメータ

<回数>ne            タイマハンドラを起動する回数。  
-s                    タイマログを出力ウインドウに表示しない

#### 説明

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

### 7.2.7 wait\_timer [<回数>ne]

#### パラメータ

<回数>ne            タイマハンドラの起動回数。

#### 説明

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。

### 7.2.8 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。

#### 説明

初期値は UP。

対応する GUI の表示を以下に示す。



### 7.2.9 inivolume <最大値>ne

#### パラメータ

<最大値>ne           ボリュームの最大値 (15～65535) 。

#### 説明

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 255t
```

### 7.2.10 setvolume <現在値>ne

#### パラメータ

<現在値>ne ボリュームに設定する値。

#### 説明

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 50t

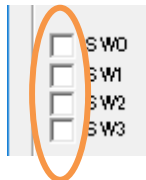
### 7.2.11 setswitch <スイッチ番号>ne {ON|OFF}

#### パラメータ

<スイッチ番号>ne 設定するスイッチ番号。  
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)

#### 説明

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。



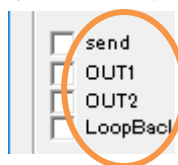
### 7.2.12 set\_switch\_name <スイッチ番号>ne [<表示名>]

#### パラメータ

<スイッチ番号>ne 設定するスイッチ番号。  
<表示名>            文字列 (空白は含まないこと)

#### 説明

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。



### 7.2.13 irc [-l<level>ne [-e] [-m{ON | OFF}]]

#### パラメータ

-l<level>ne        対象となるレベル  
-e                   対象となるレベルに EOI コマンド発行  
-m{on | off}       対象となるレベルのマスク設定

#### 説明

オプションパラメータを省略するとすべての割込みコントローラ (IRC) のレジスタ (マスク、割込み要求、サービス中) を表示する。対象となるレベルを指定するとそのレベルのレジスタを表示する。

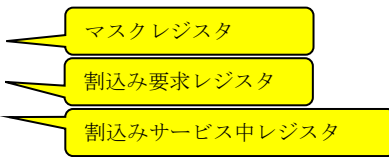
対象となるレベルを指定して -e オプションを指定するとそのレベルに EOI コマンドを発行する。  
対象となるレベルを指定して -m オプションを指定するとそのレベルのマスクを設定 (on) / 解除

(off)する。に EOI コマンドを発行する。

#### 例

以下は IRC のレジスタを表示する例です。

```
> irc
;level: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
;Mask : 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1
;Req : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
;IS : 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```



```
> irc -l4
;level: 4
;Mask : 1
;Req : 0
;IS : 0
```

以下は EOI コマンドを発行する例です。

```
> irc -l4 -e
;Set EOI Command(level=4)
```

以下はマスクを設定する例です。

```
> irc -l2 -m on
;Change Mask Register(level=2): 0 -> 1
> irc -l2
;level: 2
;Mask : 1
;Req : 0
;IS : 0
```

## 7.3 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。（「[1.3.5 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。~~

~~¥/:\*?"<>|'%;.,(){}[]~~

領域名は識別子です。（V3.00 で変更、「[6. 1. 3 \(1\) 識別子](#)」を参照）

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。

※ここで定義するメモリ操作（読み／書き）コマンドはバスマスターとして行います。CPU は LOCK 期間中はバス操作の権限を放棄しません。そのため CPU が LOCK 操作している間はメモリ操作を遅延し、LOCK 解除後まで待ってから実行します。

### 7.3.1 define\_mem <メモリサイズ>ne <IO サイズ>ne {BE | LE} {BA | WA} [-f]

#### パラメータ

<メモリサイズ>ne	メモリ空間のアドレスサイズを指定
<IO サイズ>ne	IO 空間のアドレスサイズを指定
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）
-f	既存のメモリ領域を削除して再定義する

#### 説明

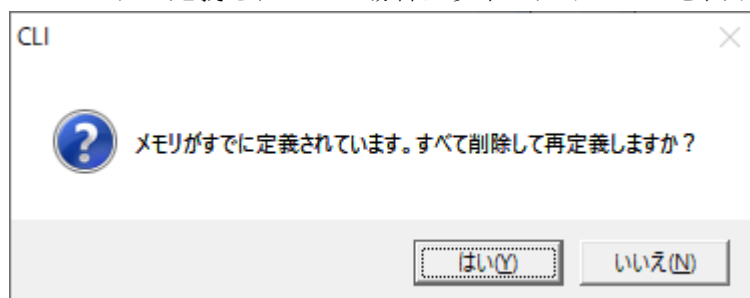
ターゲット CPU 固有のアドレス空間を定義する。ターゲット CPU の物理アドレスに依存するメモリをシミュレートする場合に実行しておく。メモリアドレスの範囲は 0～<メモリサイズ>-1。一般にメモリ空間内には、RAM, ROM, EEPROM, フラッシュメモリなどを配置するのでそれらをシミュレートする場合に使用する。

IO アドレスの範囲は 0～<IO サイズ>-1。

<メモリサイズ>の最大値は 1000000H です。

<IO サイズ>の最大値は 10000H です。

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。

-f オプションを指定するとこのダイアログは表示せずにターゲットメモリを削除して再定義する。

#### 例

1 6 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 バイトのメモリ空間だけを使う場合は以下のように指定します。

```
define_mem 10000 0 LE BA
```

1 6 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します。

```
define_mem 10000 10000 BE WA
```

### 7.3.2 add\_mem\_area <領域名>i <ベース>ne <サイズ>ne <バンク数>ne {R | RW} [-V]

#### パラメータ

<領域名>i	メモリ領域の名前を指定
<ベース>ne	領域のメモリ空間内のベースアドレスを指定
<サイズ>ne	領域のメモリ空間内のアドレスサイズを指定
<バンク数>ne	バンク数を指定（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 I/O 領域

#### 説明

メモリ空間内に RAM、EEPROM、メモリマップド I/O 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、以下のように指定します。

```
add_mem_area register 0 100 1 RW
```

### 7.3.3 add\_permanent\_area <領域名>i <ベース>ne <サイズ>ne <バンク数>ne {R | RW} [<ファイル名>f]

#### パラメータ

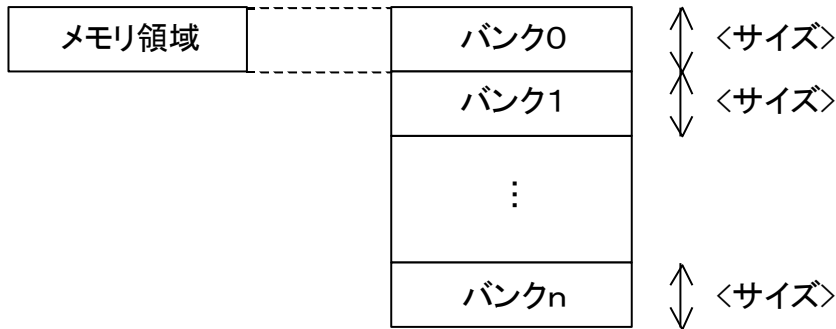
<領域名>i	メモリ領域の名前を指定
<ベース>ne	領域のメモリ空間内のベースアドレスを指定
<サイズ>ne	領域のメモリ空間内のアドレスサイズを指定
<バンク数>ne	バンク数を指定（バンクがないときは1を指定）
R	リードオンリ（読み取り専用）領域
RW	リード・ライト可能領域
<ファイル名>f	ファイルを指定するパス名（空白を含む場合は “” で囲む）

#### 説明

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。



#### ファイルの内容(先頭から)



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 0xa000~0xffff 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、以下のように指定します。

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

#### 7.3.4 add\_io\_area <領域名>i <ベース>ne <サイズ>ne

##### パラメータ

<領域名>i            I/O 領域の名前を指定  
<ベース>ne            領域の I/O 空間内のベースアドレスを指定  
<サイズ>ne            領域の I/O 空間内のアドレスサイズを指定

##### 説明

I/O 空間内に I/O 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。

I/O 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、以下のように指定します。

```
add_io_area map 0 10
```

#### 7.3.5 delete\_area <領域名>i

##### パラメータ

<領域名>i            メモリ領域、I/O 領域の名前を指定

##### 説明

<領域名>で指定されたメモリ領域、I/O 領域を削除する。

#### 7.3.6 erase\_area <領域名>i

##### パラメータ

<領域名>i            メモリ領域、I/O 領域の名前を指定

##### 説明

〈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します。

```
erase_area const
```

### 7.3.7 rotate\_bank 〈領域名〉i [-i<レベル>ne]

#### パラメータ

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

#### 説明

〈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。

### 7.3.8 fill\_bank 〈領域名〉i[.〈バンク番号>ne] {-PN9 | -PN15} [-init]

### 7.3.9 fill\_bank 〈領域名〉i[.〈バンク番号>ne] -t<fill\_spec> [-init]

#### パラメータ

〈領域名〉i	メモリ領域、IO 領域の名前を指定
〈バンク番号>ne	バンク番号を指定
-PN9	PN 符号として PN9 を指定
-PN15	PN 符号として PN15 を指定
-init	初期値を使って PN 符号を生成開始する
-t<fill_spec>	〈fill_spec〉は以下のどれか
{PN9   PN15}	PN 符号で埋める
{DUP   DUPW} <xx>ne ...	データ列の繰り返しで埋める
{INC   INCW   DEC   DECW} [<初期値>ne]	増加、減少データ列で埋める

#### 説明

〈領域名〉で指定されたメモリ領域のバンクを〈fill\_spec〉で指定された方法で埋める。〈バンク番号〉が省略された場合はバンク 0 が対象となる。

1) fill\_bank 〈領域名〉i[.〈バンク番号>ne] {-PN9 | -PN15} [-init]

2) fill\_bank 〈領域名〉i[.〈バンク番号>ne] -t{PN9 | PN15} [-init]

-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。

3) fill\_bank 〈領域名〉i[.〈バンク番号>ne] {-DUP | -DUPW} <xx>ne ...

-DUP はバイト列<xx>ne ...で繰り返し埋める。-DUPW はワード列<xx>ne ...で埋める。

4) fill\_bank 〈領域名〉i[.〈バンク番号>ne] {-INC | -INCW} [<初期値>ne]

-INC は<初期値>ne から+1 しながらバイト単位で埋める。-INCW は<初期値>ne から+1 しながらワード単位で埋める。〈初期値>ne が省略された場合は前回の続きから始める。

5) fill\_bank 〈領域名〉i[.〈バンク番号>ne] {-DEC | -DECW} [<初期値>ne]

-DEC は<初期値>ne から-1 しながらバイト単位で埋める。-DECW は<初期値>ne から-1 しながらワード単位で埋める。〈初期値>ne が省略された場合は前回の続きから始める。

## 例

“ram”という名前で作成したメモリ領域に対して以下のコマンドを実行した場合、エンディアン、メモリセルのサイズの違いでどうなるかを示します。

```
fill_bank ram -tPN9 -init
```

バイトアドレッシングの場合はエンディアンの違いにかかわらずバンクの先頭から以下のようにになります。

バンク内 オフセット	ビッグエンディアン リトルエンディアン
0	07
1	BE
2	2E
3	64

ワードアドレッシングの場合はエンディアンの違いによりバンクの先頭から以下のようにになります。

バンク内 オフセット	ビッグエンディアン	リトルエンディアン
0	07BE	BE07
1	2E64	642E
2	129D	9D12
3	A3CF	CFA3

※ワードアドレッシングのメモリにおけるバイト列は「[1.3.3 メモリシステム](#)」を参照。

### 7.3.10 set\_bank [<領域名>i[.<バンク番号>ne] <ファイル名>f [-o<オフセット>ne]]

#### パラメータ

- <領域名>i      メモリ領域、I/O 領域の名前を指定
- <バンク番号>ne      バンク番号を 16 進数で指定
- <ファイル名>f      ファイル名（空白を含むファイル名の場合は 2 重引用符 “で囲む）
- <オフセット>ne      ファイル内の先頭からの位置

#### 説明

<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>位置からのデータで埋める。  
<バンク番号>が省略された場合はバンク 0 が対象となる。  
<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

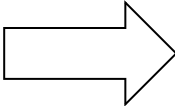
違うファイル名を指定し、-o オプションを省略するとオフセットは 0 となる。  
パラメータをすべて省略すると、現在のファイル名とファイル位置を表示する。

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		a001	000c	0c00
2	0c		a002	0038	3800
3	00		a003	0050	5000
4	38		a004	065a	5a06
5	00		a005	06f2	f206
6	50				
7	00				
8	5a				
9	06				
a	f2				
b	06				

※ワードアドレッシングのメモリにおけるバイト列は「[1.3.3 メモリシステム](#)」を参照。

## 例

使用例です。

```
> define_mem 10000 10000 BE BA
> add_mem_area com1 8000 1000 2 RW
> set_bank com1.0 font_han2.bin -o0
> set_bank
;現在のファイル位置: font_han2.bin, offset = 1000H.
```

### 7.3.11 copy\_bank <先領域名>i[.<バンク番号>ne] <元領域名>i[.<バンク番号>ne]

#### パラメータ

<先領域名>i コピー先メモリ領域、I/O 領域の名前を指定  
 <元領域名>i コピー元メモリ領域、I/O 領域の名前を指定  
 <バンク番号>ne バンク番号を指定

#### 説明

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。

### 7.3.12 set [-{s | p}]<アドレス>a [{-{b[u] | w | l}} <xx>ne | <string>q] ... [-i<レベル>ne]

#### パラメータ

-s<アドレス>a メモリ空間のアドレスとバンク番号を指定  
 -p<アドレス>a I/O 空間のアドレスを指定  
 -b <xx>ne ... 書き込むバイト列を指定する。  
 -bu <xx>ne ... 書き込むバイト列を指定する。  
 -w <xx>ne ... 書き込むワード列を指定する。  
 -l <xx>ne ... 書き込むダブルワード列を指定する。  
 <string>q マルチバイト文字列を指定する。文字列は” (2 重引用符) または' (引用符) で囲む。  
 -i<レベル>ne IRC の割込みレベル(0~15)

#### 説明

メモリ領域、I/O 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, fill 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、〈string〉は” (2 重引用符) または' (引用符) 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、〈string〉の” (2 重引用符) と' (引用符) 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 オプションをアンパック形式と呼ぶことにする。

	set -s8000 -b 30 31 32 33 set -s8000 "0123"			set -s8000 -bu 30 31 32 33 set -s8000 '01234'	
	ビッグ エンディアン	リトル エンディアン		ビッグエンディアン リトルエンディアン	
アドレス (16 進)					
8000	0x3031	0x3130		0x0030	
8001	0x3233	0x3332		0x0031	
8002				0x0032	
8003				0x0033	

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
set -w 1234 5678
set -l 12345678
```

とします。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ます。

```
> set
;現在の write 位置 : mem : 800CH.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
```

のようにします。

アドレスを省略すると前回の続きに書き込みます。(ビッグエンディアン、ワードアドレッシング)

```
> set -s8020 "0123456"
> set "789ABCDEF"
> get -s8020 -w8
3031 3233 3435 3637 3839 4142 4344 4546
```

### 7.3.13get [-{s | p}<アドレス>a] -{b | w | l | c[u]}<個数>ne ]

#### パラメータ

-s<アドレス>a   メモリ空間のアドレスとバンク番号を指定  
-p<アドレス>a   I/O 空間のアドレスを指定  
-b<個数>ne      読み出すバイト数を指定する。  
-w<個数>ne      読み出すワード数を指定する。  
-l<個数>ne      読み出すダブルワード数を指定する。  
-c<個数>ne      読み出す文字数を指定する。アスキー列で表示。  
-cu<個数>ne     読み出す文字数を指定する。アスキー列で表示。

#### 説明

メモリ領域、I/O 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

ビッグエンディアン、ワードアドレッシングの場合は、

```
> define_mem 10000 10000 BE WA
> add_mem_area com 8000 2000 2 RW
>
> set -s8000 '0123456789ABCDEF'
> get -s8000 -w4
0030 0031 0032 0033
> get -s8000 -cu8
".0.1.2.3"
> 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
"01234567"
> get -s8000 -c8
"01234567"
> get -s8000 -b8
30 31 32 33 34 35 36 37
```

ビッグエンディアン、バイトアドレッシングの場合は、

```
> define_mem 10000 10000 BE BA
> add_mem_area com 8000 2000 2 RW
>
> set -s8000 '0123456789ABCDEF'
> get -s8000 -w4
3031 3233 3435 3637
> get -s8000 -cu8
"01234567"
> get -s8000 -c8
"01234567"
> get -s8000 -b8
30 31 32 33 34 35 36 37
>
```

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

### 7.3.14fill [-[s | p]<アドレス>a] {[-[b | bu | w | l] <x>ne ...] | [<string>q]} ]

#### パラメータ

-s<アドレス>a   メモリ空間のアドレスとバンク番号を指定  
 -p<アドレス>a   IO空間のアドレスを指定  
 -b <xx>ne ...   書き込むバイト列を指定する。  
 -bu <xx>ne ...   書き込むバイト列を指定する。  
 -w <xx>ne ...   書き込むワード列を指定する。  
 -l <xx>ne ...   書き込むダブルワード列を指定する。  
 <string>q       マルチバイト文字列（バイト列）を指定する。文字列は”（2重引用符）または’（引用符）で囲む。

#### 説明

メモリ領域、IO領域へデータを書き込む。  
 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, fill コマンドの続き書き込む。  
 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、<string>は”（2重引用符）または’（引用符）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）における-b オプションと-bu オプションの違い、<string>の”（2重引用符）と’（引用符）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 オプションをアンパック形式と呼ぶことにする。  
 set コマンドと違い-b, -bu, -w, -l, <string>q を混在して指定できる。指定した順番にメモリ、IOへ続けて書き込む。  
 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、<string>の”（2重引用符）で奇数バイト数指定の後に-w, -l が続く場合は以下になるので注意が必要となる。-bu, -w, -l オプションは必ずワード境界から書き込む。

```
fill -s8000 -b 30 31 32 -w 55aa
fill -s8000 "012" -w 55aa
```

アドレス (16進)	ビッグ エンディアン	リトル エンディアン
	エンディアン	エンディアン
8000	0x3031	0x3130
8001	0x32XX	0xFF32
8002	0x55aa	0x55aa
8003		

ワード境界から書き込む

#### 例

以下はワードアドレッシングのメモリへ fill コマンドで書き込む例です。

```
> define_mem 10000 10000 BE WA
```

```

> add_mem_area      com_io 8000 2000 2 RW
> erase_area com_io      ;0xff で埋める
> fill -s8000.1 @reps(5 , -l 33 -w 44 -b 66 "sample" )
> get -s8000.1 -w23
0000 0033 0044 6673 616d 706c 65ff 0000
0033 0044 6673 616d 706c 65ff 0000 0033
0044 6673 616d 706c 65ff 0000 0033 0044
6673 616d 706c 65ff 0000 0033 0044 6673
616d 706c 65ff
> get
;現在の read 位置 : mem : 8023H.1

```

5 回繰り返す

次は WARD 境界から開始するため

以下はバイトアドレッシングのメモリへ fill コマンドで書き込む例です。

```

> define_mem 10000 10000 BE BA
> add_mem_area      com_io 8000 2000 2 RW
> erase_area com_io      ;0xff で埋める
> fill -s8000.1 @reps(5 , -l 33 -w 44 -b 66 "sample" )
> get -s8000.1 -b46
00 00 00 33 00 44 66 73 61 6d 70 6c 65 00 00 00
33 00 44 66 73 61 6d 70 6c 65 00 00 00 33 00 44
66 73 61 6d 70 6c 65 00 00 00 33 00 44 66 73 61
6d 70 6c 65 00 00 00 33 00 44 66 73 61 6d 70 6c
65 ff ff ff ff ff
> get
;現在の read 位置 : mem : 8046H.1

```

5 回繰り返す

### 7.3.15 wait -{s | p}<アドレス>a -{b | w | l} <xx>ne[. {OR | AND}] [-t<タイムアウト>]ne]

#### パラメータ

-s<アドレス>a   メモリ空間のアドレスとバンク番号を指定  
 -p<アドレス>a   IO 空間のアドレスを指定  
 -b <xx>ne       比較するバイト値を指定する。  
 -w <xx>ne       比較するワード値を指定する。  
 -l <xx>ne       比較するダブルワード値を指定する。  
 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>ne 待ち時間をミリ秒指定。10 進数で指定。

#### 説明

メモリ領域、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。それ以外のは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

```

### 7.3.16 peek -{s | p}<アドレス>a -{b | w | l}

#### パラメータ



-s<アドレス>a   メモリ空間のアドレスとバンク番号を指定  
 -p<アドレス>a   IO空間のアドレスを指定  
 -b               操作対象はバイト。ワードアドレッシングのメモリではワード。  
 -w               操作対象はワード。  
 -l               操作対象はダブルワード。

#### 説明

メモリ領域、IO領域の内容を読み出す。  
 バンク番号が省略された場合はバンク 0 が対象となる。  
 直前の結果は定義済み変数型マクロ\_R, \_ZF, \_SF で参照できる。

#### 例

```
> define_mem 10000 10000 BE BA -f
> add_mem_area com 8000 2000 2 RW
>
> fill -s8000 -b @inc(10, 00) 80 81 82 83
> get -s8000 -b14 ;メモリの初期状態
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
80 81 82 83
>
> peek -s 8000 -b
;mem : 8000H.0 = 00
> print $_R, $_ZF, $_SF
00, 1, 0
>
> peek -s 8000 -w
;mem : 8000H.0 = 0001
> print $_R, $_ZF, $_SF
0001, 0, 0
>
> peek -s 8010 -b
;mem : 8010H.0 = 80
> print $_R, $_ZF, $_SF
80, 0, 1
```

### 7.3.17poke -O<操作> -{s | p}<アドレス>a -{b | w | l} [<data>ne] [-i<レベル>ne]

#### パラメータ

-s<アドレス>a   メモリ空間のアドレスとバンク番号を指定  
 -p<アドレス>a   IO空間のアドレスを指定  
 -O<操作>       <操作>は以下のどれか  
   NOP           何もしない  
   INC           対象メモリをインクリメント (<data>ne は不要)  
   DEC           対象メモリをデクリメント (<data>ne は不要)  
   NOT           対象メモリのビット反転 (<data>ne は不要)  
   NEG           対象メモリの符号を反転 (<data>ne は不要)  
   READ          対象メモリを読み出す (<data>ne は不要)  
   WRITE         対象メモリへ<data>ne を書く  
   ADD           対象メモリへ<data>ne を加え、結果を書き戻す  
   SUB           対象メモリから<data>ne を引き、結果を書き戻す  
   AND           対象メモリと<data>ne の AND 演算を行い、結果を書き戻す  
   OR            対象メモリと<data>ne の OR 演算を行い、結果を書き戻す  
   XOR           対象メモリと<data>ne の XOR 演算を行い、結果を書き戻す  
 -b [<data>ne]   操作対象はバイト。ワードアドレッシングのメモリではワード。<data>ne は

演算対象値。  
 -w [<data>ne] 操作対象はワード。<data>ne は演算対象値。  
 -l [<data>ne] 操作対象はダブルワード。<data>ne は演算対象値。  
 -i<レベル>ne IRC の割込みレベル(0~15)

#### 説明

メモリ領域、IO 領域の内容を読み出し演算操作する。結果を元のメモリ領域、IO 領域に書き戻す。

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, -p オプションは省略できません。

直前の結果は定義済み変数型マクロ\_R, \_ZF, \_SF で参照できる。

#### 例

```
> fill -s8000 -b @inc(10, 50)
> get -s8000 -b10 ;メモリの初期状態
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
>
> poke -Onop -s8000 -b
> poke -Oinc -s8001 -b
;mem : 8001H.0 = 52
> poke -Odec -s8002 -b
;mem : 8002H.0 = 51
> poke -Onot -s8003 -b
;mem : 8003H.0 = AC
> poke -Oneg -s8004 -b
;mem : 8004H.0 = AC
> poke -Oread -s8005 -b
;mem : 8005H.0 = 55
> poke -Owrite -s8006 -b 34
;mem : 8006H.0 = 34
> poke -Oadd -s8007 -b 11
;mem : 8007H.0 = 68
> poke -Osub -s8008 -b 11
;mem : 8008H.0 = 47
> poke -Oand -s8009 -b f0
;mem : 8009H.0 = 50
> poke -Oor -s800a -b 80
;mem : 800AH.0 = DA
> poke -Oxor -s800a -b ff
;mem : 800AH.0 = 25
> print $_R, $_ZF, $_SF
25, 0, 0
```

### 7.3.18sum -{s | p}<アドレス>a -{b | w | l}<個数>ne

#### パラメータ

-s<アドレス>a メモリ空間のアドレスとバンク番号を指定  
 -p<アドレス>a IO 空間のアドレスを指定  
 -b<個数>ne バイト数を指定する。バイト列の和を計算。  
 -w<個数>ne ワード数を指定する。ワード列の和を計算。  
 -l<個数>ne ダブルワード数を指定する。ダブルワード列の和を計算。

#### 説明

連続するメモリ領域、IO 領域の内容の和を計算する。結果は 32 ビット整数値。

直前の計算結果は定義済み変数型マクロ\_R, \_ZF, \_SF で参照できる。

例

バイトアドレッシングの場合

```
> define_mem 10000 10000 BE BA -f
> add_mem_area com 8000 2000 2 RW
> get -s 8000 -b 10
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
> get -s 8000 -w 8
0100 0302 0504 0706 0908 0b0a 0d0c 0f0e
> sum -s 8000 -b10
00000078
> print $_R
00000078
> sum -s 8000 -w8
00003840
> sum -s 8000 -l4
181C2024
```

ワードアドレッシングの場合 (チェックサム値はバイトアドレッシングと同じ)

```
> define_mem 10000 10000 BE WA -f
> add_mem_area com 8000 2000 2 RW
>
> set -s 8000 -b @inc(10, 0)
> get -s 8000 -b 10
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
> get -s 8000 -w 8
0001 0203 0405 0607 0809 0a0b 0c0d 0e0f
>
> sum -s 8000 -b10
00000078
> print $_R
00000078
> sum -s 8000 -w8
00003840
> sum -s 8000 -l4
181C2024
```

### 7.3.1crc [-s | p]<アドレス>a -b<個数>ne -t[L | R]<ビット数>ne -P<poly> [-I<init>] [-X]]

パラメータ

-s<アドレス>a   メモリ空間のアドレスとバンク番号を指定  
-p<アドレス>a   I/O空間のアドレスを指定  
-b<個数>ne       バイト数を指定する。バイト列の和を計算。  
-P<poly>ne       CRC多項式  
-I<初期値>ne     初期値。省略すると0  
-X                計算結果と ffffffff のXOR (排他的論理和) をとる。

説明

メモリ領域のCRC(Cyclic Redundancy Check)を計算し、結果を16進数文字列に変換する。  
CRC多項式の指定方法の例を以下に示す。

名称	多項式	2進数表記	16進数表記
CRC-7	$x^7 + x^6 + x^2 + 1$	左送り 100_0101_ 右送り 101_0001_	45(bit6-0) 51(bit6-0)
CRC-8/ATM	$x^8 + x^2 + x + 1$	左送り 0000_0111_ 右送り 1110_0000_	07 E0

CRC-8/CCITT	$x^8 + x^7 + x^3 + x^2 + 1$	左送り 1000_1101_ 右送り 1011_0001_	8D B1
CRC-16: The HDLC (CRC-CCITT)	$x^{16} + x^{12} + x^5 + 1$	左送り 0001_0000_0010_0001_ 右送り 1000_0100_0000_1000_	1021 8408
CRC-16 (CRC-ANSI)	$x^{16} + x^{15} + x^2 + 1$	左送り 1000_0000_0000_0101_ 右送り 1010_0000_0000_0001_	8005 a001
CRC-32	$x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$	左送り 0000_0100_1100_0001_ 0001_1101_1011_0111_ 右送り 1110_1101_1011_1000_ 1000_0011_0010_0000_	04c11db7 edb88320

CRC はメモリ上のバイト列に対して計算する。ワードアドレッシングのメモリでのバイト列とメモリの関係は「[1.3.3 メモリシステム](#)」を参照。

CRC はビット列に対して計算するので各バイト内の MSB (Most Significant Bit) 側から計算するのか、LSB (Least Significant Bit) 側から計算するのかを指定しなければならない。

左送りとは MSB から LSB へのビット並びとして扱う。右送りとは LSB から MSB へのビット並びとして計算する。そのため多項式の数値表記は高次を左送りなら MSB、右送りなら LSB に設定する。

処理は多項式に対応した変換テーブルを作り、対象バイト列からバイト単位で取り出し計算する。変換テーブルは CRC のビット数により以下のようにする。

CRC のビット数	変換テーブルの配列 (要素数は 256)
1～8	バイト (8 ビット) の配列
9～16	ワード (16 ビット) の配列
17～32	ダブルワード (32 ビット) の配列

変換テーブルの内容はパラメータなしの crc コマンドで確認できる。左送りの場合は多項式を MSB 詰めとした値を使用する。例えば上記の CRC7 左送りの場合は、45 を MSB 詰めとするため 1 ビット左へシフトし 8A としてから変換テーブルの配列を作成する。

計算結果は左送りの場合、右送りの場合ともに LSB 詰めの値として返す。オプションパラメータ -X を指定すると計算結果の全ビットを反転した値を返す。

直前の CRC 計算結果は定義済み変数型マクロ \_R, \_ZF, \_SF で参照できる。

パラメータをすべて省略した場合は、直前に使った CRC ルックアップテーブルを表示する。

例

```
> define_mem 10000 10000 BE BA -f
> add_mem_area ram 8000 2000 2 RW;
> set -s 8000 "123456789"
> get -s 8000 -b10
31 32 33 34 35 36 37 38 39 ff ff ff ff ff ff ff
> crc -tL32t -s 8000 -b9 -P4C11DB7 -I(-1) -X;CRC-32/BZIP2
FC891918
> crc -tL16t -s 8000 -b9 -P 8005 ;CRC-16
FEE8
> crc -tL16t -s 8000 -b9 -P 1021 -I(-1) -X ;CRC-16/CCITT
D64E
> crc -tL8 -s 8000 -b9 -P 07 ;CRC-8/ATM
F4
> crc -tL8 -s 8000 -b9 -P 8d ;CRC-8/CCITT
D2
```

以下はCRC7 の例です。

```
> set -s 8010 -b 01 01 ef
> set -s 8020 -b 80 80 f7
> crc -tL7 -s 8010 -b3 -P 45 ;多項式は bit6-0。結果は bit6-0 の 7 ビット
38
> crc -tR7 -s 8020 -b3 -P 51 ;多項式は bit6-0。結果は bit6-0 の 7 ビット
0E
```

直前の CRC 計算に使ったルックアップテーブルを表示する例。

```
> crc -tL16t -s 8000 -b 9 -P1021 -I ( -1) -X ;CRC-16/CCITT
D64E
> crc
;CRC-16 Lookup Table:
;polynomial=1021, shift left
0000 1021 2042 3063 4084 50A5 60C6 70E7
8108 9129 A14A B16B C18C D1AD E1CE F1EF
1231 0210 3273 2252 52B5 4294 72F7 62D6
9339 8318 B37B A35A D3BD C39C F3FF E3DE
2462 3443 0420 1401 64E6 74C7 44A4 5485
A56A B54B 8528 9509 E5EE F5CF C5AC D58D
3653 2672 1611 0630 76D7 66F6 5695 46B4
B75B A77A 9719 8738 F7DF E7FE D79D C7BC
48C4 58E5 6886 78A7 0840 1861 2802 3823
C9CC D9ED E98E F9AF 8948 9969 A90A B92B
5AF5 4AD4 7AB7 6A96 1A71 0A50 3A33 2A12
DBFD CBDC FBBF EB9E 9B79 8B58 BB3B AB1A
6CA6 7C87 4CE4 5CC5 2C22 3C03 0C60 1C41
EDAE FD8F CDEC DD CD AD2A BD0B 8D68 9D49
7E97 6EB6 5ED5 4EF4 3E13 2E32 1E51 0E70
FF9F EFBE DFDD CFFC BF1B AF3A 9F59 8F78
9188 81A9 B1CA A1EB D10C C12D F14E E16F
1080 00A1 30C2 20E3 5004 4025 7046 6067
83B9 9398 A3FB B3DA C33D D31C E37F F35E
02B1 1290 22F3 32D2 4235 5214 6277 7256
B5EA A5CB 95A8 8589 F56E E54F D52C C50D
34E2 24C3 14A0 0481 7466 6447 5424 4405
A7DB B7FA 8799 97B8 E75F F77E C71D D73C
26D3 36F2 0691 16B0 6657 7676 4615 5634
D94C C96D F90E E92F 99C8 89E9 B98A A9AB
5844 4865 7806 6827 18C0 08E1 3882 28A3
CB7D DB5C EB3F FB1E 8BF9 9BD8 ABBB BB9A
4A75 5A54 6A37 7A16 0AF1 1AD0 2AB3 3A92
FD2E ED0F DD6C CD4D BDAA AD8B 9DE8 8DC9
7C26 6C07 5C64 4C45 3CA2 2C83 1CE0 0CC1
EF1F FF3E CF5D DF7C AF9B BFBA 8FD9 9FF8
6E17 7E36 4E55 5E74 2E93 3EB2 0ED1 1EF0
```

### 7.3. 2tmem

#### パラメータ

なし

#### 説明

ターゲットメモリの定義情報を表示する。

#### 例

まず以下のコマンドでターゲットメモリを作成します。

```

define_mem 10000 10000 LE      BA
add_mem_area  reg      0000 0100 1 RW
add_mem_area  com_io   8000 2000 2 RW
add_mem_area  exram    c000 1000 8 RW
add_mem_area  ROM      e000 2000 8 R
add_mem_area  MIO      d000 0100 1 RW -V
add_permanent_area フォント a000 1000 1 R font_han2.bin
add_permanent_area NV    b000 1000 1 RW nvdata.bin
add_io_area   スイッチ      0000 0010

```

次に tmem コマンドを実行した例を示します。

```

> tmem
;Target Memory Configuration:
;little endian
;byte addressing
;Memory space:
section name      attr  b tbase  tsize  filename
reg               RW    1 000000H 000100H
com_io            RW    2 008000H 002000H
フォント          R:P    1 00A000H 001000H font_han2.bin
NV                RW:P   1 00B000H 001000H nvdata.bin
exram             RW    8 00C000H 001000H
MIO               RW:V   1 00D000H 000100H
ROM               R      8 00E000H 002000H
;IO space:
スイッチ          RW:V   1 000000H 000010H

```

## 7.4 serial <シリアルポート番号>ne <サブコマンド>

### パラメータ

<シリアルポート番号>ne 設定するポート番号 (0 または 1)  
 <サブコマンド> サブコマンド

### 説明

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。

※ 「[5.9 16550 相当のシリアル制御関数](#)」 も参照してください。

### 7.4.1 init <割り込みレベル>ne <チップ種別>s

#### パラメータ

<割り込みレベル>ne シリアルポートに割り当てる割り込みレベル。  
 <チップ種別>s 16550 を指定 (シミュレートするシリアルコントローラ種別)。

#### 説明

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、割り込みレベル 4 を割り当てる場合は以下のように指

定します。

```
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ります。



#### 7.4.2 info

##### パラメータ

なし

##### 説明

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。
```

#### 7.4.3 set {CTS | DSR | RI | DCD} {ON | OFF}

##### パラメータ

CTS	Clear to Send (送信可)
DSR	Data Set Ready (データセットレディ)
RI	Ring Indicator (ベル検出、被呼表示)
DCD	Data Carrier Detect (受信キャリア検出)
ON	信号を 1 に設定する
OFF	信号を 0 に設定する

##### 説明

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します。

```
serial 1 set CTS on
```

#### 7.4.4 push {<xx>ne | <アスキー>" | sb | se} ... [-c<間隔>ne]

##### パラメータ

<xx>ne	8 ビット値
<アスキー>"	2 重引用符 “で囲んだアスキー文字列 (空白を含む)
sb	ブレークキャラクタ受信を受信 FIFO に設定
se	直後の受信文字と同時に受信エラー (パリティエラー、 フレーミングエラー) を受信 FIFO に設定
-c<間隔>ne	FIFO に設定する間隔 (文字の倍数)。省略すると 1 文字分。

##### 説明

16550 の受信 FIFO へデータを挿入する。

sb 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。

se 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。

<間隔>ne は FIFO に設定した後何文字分空けて次を設定するかを指定する。省略すると 1 文字分空ける。0 を指定すると間を空けずに設定するので CPU が読み取る前に FIFO を満杯でき、オーバーフローを起こせる。

※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。オーバーランエラーは FIFO があふれた時点で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します。

```
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します。

```
serial 1 push "12345" se 0a 0d
```

オーバーフローエラーを起こす場合は以下のように指定します。17 文字以降は FIFO があふれるので捨てられます。

```
serial 1 push "12345678901234567890" -c0 ;オーバーフロー発生
```

### 7.4.5 probe {DTR | RTS | OUT1 | OUT2} [-w<タイムアウト>]ne]

#### パラメータ

DTR	Data Terminal Ready (データ端末レディ)
RTS	Request To Send (送信要求)
OUT1	OUT1 出力ピン状態
OUT2	OUT2 出力ピン状態

-w<タイムアウト>ne 待ち時間をミリ秒指定。

#### 説明

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)。それ以外はタイムアウトの指定は以下ようになる。

-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ます。

```
serial 1 probe DTR
```

## 7.5 ユーティリティ・コマンド

CLI を使用するにあたり作業を助ける機能を提供します。



### 7.5.1clear

#### パラメータ

なし

#### 説明

コンソール（出力ウインドウ）をクリアする。TCP/IP 端末へはフォームフィード(‘\f’)を送る。

### 7.5.2print <メッセージ>m

#### パラメータ

<メッセージ>m 表示する文字列。

#### 説明

前処理を施した結果の文字列をコンソールに表示する。[nolist コマンド](#)が実行された状態では表示しない。

### 7.5.3trace <メッセージ>m

#### パラメータ

<メッセージ>m 出力ウインドウに表示する文字列。

#### 説明

前処理を施した結果の文字列を出力ウインドウに表示する。  
[nolist コマンド](#)が実行された状態でも表示する。  
現在のコンソールに関係なく出力ウインドウに表示する

### 7.5.4help [\*][<コマンド名>s]

#### パラメータ

<command>s コマンド名またはコマンド名の一部

#### 説明

コマンド名の先頭から<コマンド名>s と一致するコンソール・コマンドの構文を表示する。  
\*を指定した場合は<コマンド名>s を含むコンソール・コマンドの構文を表示する。  
アルファベット順に表示します。

#### 例

以下は add で始まるコマンド一覧を表示する例です。

```
> help add
add_io_area <name> <base> <size>
add_mem_area <name> <base> <size> <banks> <attr> [-V]
add_permanent_area <name> <base> <size> <banks> <attr> <filename>
```

以下はコマンド名に area の文字があるコマンド一覧を表示する例です。

```
> help *area
add_io_area <name> <base> <size>
add_mem_area <name> <base> <size> <banks> <attr> [-V]
add_permanent_area <name> <base> <size> <banks> <attr> <filename>
delete_area <name>
erase_area <name>
```

### 7.5.5edit [<ファイル名>f]

#### パラメータ

<ファイル名>f テキストファイルを指定する。ファイル名に空白を含む場合は2重引用符

“で囲む。

#### 説明

<ファイル名>で指定されたファイルをテキストエディタ(メモ帳)で開く。Cmtoy 起動時の-e オプションでテキストエディタが指定されている場合は、指定されているエディタで開く。

「[2.2.3 テキスト・エディターを変更する](#)」を参照。

### 7.5.6 win\_app <ファイル名>f [<string>m]

#### パラメータ

<ファイル名>f 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指定する。ファイル名に空白を含む場合は2重引用符“で囲む。

<string>m 実行ファイル(\*. exe)に渡すパラメータ文字列

#### 説明

Windows アプリケーションを起動する。<ファイル名>は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。ドキュメントファイルを指定した場合はドキュメントファイルと関連付けられたWindows アプリケーションが起動する。

実行ファイル名に続いてパラメータ文字列を指定できる。

#### 例

以下のようにテキストファイル(\*. txt)を指定するとメモ帳が起動しテキストファイルを表示します。Windows の機能により拡張子\*. txt と関連付けられたアプリケーションが開きます。

```
> win_app test.txt
```

DOS プロンプト (cmd. exe) を起動することもできます。

```
> win_app cmd
```

### 7.5.7 calc <数値式>ne

#### パラメータ

<数値式>ne 数値式

#### 説明

<数値式>ne を評価し 32 ビット値の結果を表示する。比較は符号なし 32 ビット値として行う。結果の 32 ビット値は以下の形式で表示する。

**16 進数、符号付き 10 進数 (符号なし 10 進数)、2 進数**

直前の結果は定義済み変数型マクロ\_R, \_ZF, \_SF で参照できる。

#### 例

```
> calc 1+2<<3+4 ;0x180
00000180H, 384T(+384T), 1_1000_0000_
> calc 1+(2<<3)+4 ;0x15
00000015H, 21T(+21T), 1_0101_
> print $_R
00000015
> calc -$_R + 1
FFFFFFFECH, -20T(+4294967276T), 1111_1111_1111_1111_1111_1111_1110_1100_
>
> calc 0 && !1 || 1 ;1(true)
00000001H, 1T(+1T), 1_
```

32 ビット符号なし整数として比較した結果の例です。

```
> calc -1>0 ;1(true)
00000001H, 1T(+1T), 1_
```

```
> calc -1 == ffffffff ;1(true)
00000001H, 1T(+1T), 1_
```

### 7.5.8set\_title <タイトル>m

#### パラメータ

<タイトル>m      タイトルバーに表示する文字列。

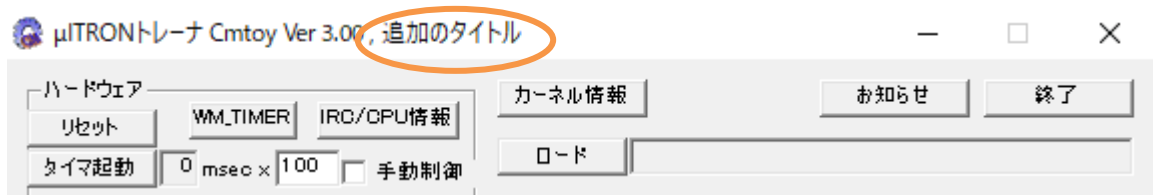
#### 説明

Cmtoy のウインドウのタイトルバーに文字列を追加する。最大アスキー30 文字（全角だと 15 文字）表示できる。

#### 例

```
> set_title 追加のタイトル
```

このコマンドの結果の結果、Cmtoy のウインドウのタイトルバーは以下ようになります。



## 8 定義済みマクロ

### 8.1 定義済み変数型マクロ

コマンドパラメータ中に使うと前処理で文字列に置き換えられます。変数型マクロを参照する場合はアスキーの\$を先導文字として使います。

[define](#), [define\\_literal](#), [define\\_input](#) コマンドでこれらの変数型マクロ名を再定義することはできません。[undef コマンド](#)でこれらの変数型マクロ名を削除することもできません。

#### 8.1.1 \_APP\_NAME

説明

Cmtoy.exe のアプリケーション名を参照する。

例

```
> print $_APP_NAME ;アプリケーション名
μITRON トレーナ Cmtoy Ver 3.00
```

#### 8.1.2 \_APP\_EXE

説明

Cmtoy.exe のフルパス名を参照する。

例

```
> print $_APP_EXE ;Cmtoy.exe のフルパス名
E:\¥cmtoy-300¥bin¥Cmtoy.exe
```

#### 8.1.3 \_APP\_VER

説明

Cmtoy.exe のモジュールバージョン番号を参照する。

例

```
> print $_APP_VER ;Cmtoy.exe のモジュールバージョン
03000000
```

#### 8.1.4 \_CM\_VER

説明

cm.dll のモジュールバージョン番号を参照する。

例

```
> print $_CM_VER ;cm.dll のモジュールバージョン
01000010
```

#### 8.1.5 \_KP\_VER

説明

kpdll.dll のモジュールバージョン番号を参照する。

例

```
> print $_KP_VER          ;kpdll.dll のモジュールバージョン  
01010010
```

### 8.1.6\_CLI\_VER

説明

CLI のバージョン番号を参照する。

例

```
> print $_CLI_VER         ;CLI のバージョン  
03000000
```

### 8.1.7\_WD

説明

作業ディレクトリを参照する。

例

```
> print $_WD              ;カレントディレクトリ名  
E:\cmtoy-300\bin
```

### 8.1.8\_DATE

説明

その時点の日付を参照する。

例

```
> print $_DATE            ;日付  
"2023/01/11"
```

その時点の日付を保持したい場合は  
> define\_literal DATE \$\_DATE

### 8.1.9\_TIME

説明

その時点の時刻を参照する。

例

```
> print $_TIME            ;時刻  
"08:50:07"
```

その時点の時刻を保持したい場合は  
> define\_literal TIME \$\_TIME

### 8.1.10\_TIMESTAMP

説明

その時点のタイムスタンプを参照する。

例

```
> print $_TIMESTAMP      ;タイムスタンプ
"2023/01/11 08:50:07"
```

その時点の時刻を保持したい場合は

```
> define_literal TIMESTAMP $_TIMESTAMP
```

### 8.1.11\_FILE

説明

実行しているスクリプトファイル名を参照する。スクリプト起動時に使ったファイル名でフルパス名とは限らない。

例

```
sample_pdsymbol.txt というスクリプトファイル内で実行した結果を示します。
> print $_FILE
E:¥cmttoy-300¥bin¥script_sample¥sample_pdsymbol.txt
```

### 8.1.12\_FILE\_NAME

説明

実行しているスクリプトファイルのファイル名の部分を参照する。

例

```
> print $_FILE_NAME      ;スクリプトファイル名
sample_pdsymbol.txt
```

### 8.1.13\_FILE\_PATH

説明

実行しているスクリプトファイルのフルパス名を参照する。

例

```
> print $_FILE_PATH      ;スクリプトファイルのフルパス名
E:¥cmttoy-300¥bin¥script_sample¥sample_pdsymbol.txt
```

### 8.1.14\_DIR\_PATH

説明

実行しているスクリプトファイルのフルパス名を参照する。

例

```
> print $_DIR_PATH ;スクリプトファイルのディスプレイ名
E:¥cmttoy-300¥bin¥script_sample¥
```

### 8.1.15\_LINE

説明

実行しているスクリプトファイル内の行番号を参照する。行番号は 10 進数文字列。

例

```
> print line no. = $_LINE
line no. = 4
> print $_FILE < $_LINE >
E:¥cmttoy-300¥bin¥script_sample¥sample_pdsymbol.txt < 5 >
```

行番号を保持したい場合は

```
> define_literal LINE1 $_LINE
```

### 8.1.16\_FILE\_HIST

説明

実行しているスクリプトファイルのネストしているファイル名と行番号を参照する。

例

```
> print $_FILE_HIST ;スクリプトのネスト歴
/sample_pdsymbol.txt(10)
```

### 8.1.17\_MSGBOX

説明

直前のスクリプト制御命令 [#msgbox](#) の実行結果を保存している。どのボタンをクリックして終了したかがわかる。

- 1 「OK」 ボタンで終了
- 6 「はい」 ボタンで終了
- 7 「いいえ」 ボタンで終了

例

```
> #msgbox 1 -t 確認 -m 「OK」 をクリックしてください。
> print $_MSGBOX
1
```

### 8.1.18\_EXIT\_CODE

説明

直前のスクリプト制御命令 [#exit](#) で指定した終了コードを保存している。16 進数文字列を返す。

例

```
> print $_EXIT_CODE
1234H
```

### 8.1.19\_R

### 8.1.20\_ZF

### 8.1.21\_SF

説明

以下のコマンド結果を 16 進数文字列で返す。直前のコマンドの結果のみを保存している。

- ・ [peek](#)

- [poke](#)
- [sum](#)
- [crc](#)
- [calc](#)

変数型マクロ名	呼び名	説明
_R	コマンド結果	コマンドの結果に合わせて8ビット、16ビット、32ビット値を16進数文字列で返す。
_ZF	ゼロフラグ	結果が0なら1を返す。0以外なら0を返す。
_SF	符号フラグ	結果の最上位ビット (MSB) を返す。

結果を文字列リテラルとして保持する場合は [define\\_literal](#) コマンドで変数型マクロを定義する。

例

以下は poke コマンドの例です。

```
> fill -s8000 -b @inc(10, 50)
> get -s8000 -b10 ;メモリの初期状態
50 51 52 53 54 55 56 57 58 59 5a 5b 5c 5d 5e 5f
>
> poke -Oxor -s800a -b ff
;mem : 800AH.0 = A5
> print $_R, $_ZF, $_SF
A5, 0, 1
> define_literal R800a $_R
```

以下は sum コマンドの例です。

```
> sum -s8000 -b10
000005C3
> print $_R, $_ZF, $_SF
000005C3, 0, 0
> define_literal SUM8000_10 $_R
>
> define
;number of defined symbols = 2
;1 : R800a = A5
;2 : SUM8000_10 = 000005C3
```

## 8.2 定義済み関数型マクロ

コマンドラインのパラメータ中に使うと前処理で文字列に置き換えられます。関数型マクロを参照する場合はアスキーの@ (アットマーク) を先導文字として使います。

定義済み関数型マクロの構文は関数名、パラメータ部からなるC言語の関数に似ています。

@<関数名>(<パラメータ 1>[, <パラメータ 2>[, ...]])

前処理の関数型マクロでエラーが起きた場合は?文字に置き換えます。エラーの詳細は [list -d](#) 状態で表示します。

未定義関数名が使われた場合は、前処理で変換されずそのまま残ります。

### 8.2.1 16進数文字列を生成

ここで扱える数値は32ビット整数値、16ビット整数値、8ビット整数値、1ビット論理値です。



32 ビット整数値、16 ビット整数値、8 ビット整数値を符号なしの正の整数と扱うか、MSB (Most Significant Bit) を符号ビットとした正負の整数 (2 の補数形式) と扱うかは

- ・ コンソールコマンドでは、パラメータの意味で決まる。
- ・ それ以外の場面ではコマンド・プログラムの意図で決まる。

表現できる整数の範囲を 16 進数と 10 進数で以下に示します。

ビット幅	符号なしの整数の範囲	符号付きの整数の範囲
32 ビット値	0～FFFFFFFF 0t～4294967295t	80000000～FFFFFFFF, 0～7FFFFFFF -2147483648t～2147483647t
16 ビット値	0～FFFF 0t～65535t	8000～FFFF, 0～7FFF -32768t～32767t
8 ビット値	0～FF 0t～255t	80～FF, 0～7F -128t～127t
1 ビット論理値	0 (偽), 1 (真)	

- |                    |                           |
|--------------------|---------------------------|
| (1) @byte(<数値>ne)  | bit0-7 を 2 桁の 16 進数文字列に   |
| (2) @word(<数値>ne)  | bit0-15 を 4 桁の 16 進数文字列に  |
| (3) @dword(<数値>ne) | bit0-31 を 8 桁の 16 進数文字列に  |
| (4) @hbyte(<数値>ne) | bit8-15 を 2 桁の 16 進数文字列に  |
| (5) @hword(<数値>ne) | bit16-31 を 4 桁の 16 進数文字列に |

パラメータ

<数値>ne          数値

説明

<数値>ne を形式に従い 32 ビット値 (バイナリ) に変換後その一部を取り出し 16 進数文字列に変換する。

例

```

> print @byte(12345678)
78
> print @word(12345678)
5678
> print @dword(12345678)
12345678
> print @hbyte(12345678)
56
> print @hword(12345678)
1234
> print @dword(12345678T)           ;10 進数文字列
00BC614E
> print @byte(1000_1001_)           ;2 進数文字列
89
> print @long(12345678)              ;未定義関数
@long(12345678)
> print @byte(zx)                   ;パラメータエラー
?

```

- |                                  |               |
|----------------------------------|---------------|
| (6) @splitw(<数値>ne, <エンディアン>s)   | ワード値をバイト列に    |
| (7) @splitdw1(<数値>ne, <エンディアン>s) | ダブルワード値をバイト列に |
| (8) @splitdw2(<数値>ne, <エンディアン>s) | ダブルワード値をワード列に |

#### パラメータ

<数値>ne            数値  
 <エンディアン>s エンディアンを指定。  
                   := {B | b | L | l}

#### 説明

<数値>ne を形式に従い 32 ビット値（バイナリ）に変換後エンディアン指定に従いバイト列、ワード列に変換する。16 進文字列の並びとなり空白で区切る。

#### 例

```
> print @splitw(12345678, b)      ; ビックエンディアン
56 78
> print @splitdw1(12345678, b)    ; ビックエンディアン
12 34 56 78
> print @splitdw2(12345678, b)    ; ビックエンディアン
1234 5678
>
> print @splitw(12345678, L)      ; リトルエンディアン
78 56
> print @splitdw1(12345678, L)    ; リトルエンディアン
78 56 34 12
> print @splitdw2(12345678, L)    ; リトルエンディアン
5678 1234
>
> print @splitdw(12345678, z)      ; 未定義関数
@splitdw(12345678, z)
> print @splitdw1(12345678, z)    ; パラメータエラー
?
```

- |                      |                      |
|----------------------|----------------------|
| (9) @boolb(<数値>ne)   | 下位 8 ビット値の 0 以外チェック  |
| (10) @boolw(<数値>ne)  | 下位 16 ビット値の 0 以外チェック |
| (11) @booldw(<数値>ne) | 下位 32 ビット値の 0 以外チェック |

#### パラメータ

<数値>ne            数値

#### 説明

<数値>ne を形式に従い 32 ビット値（バイナリ）に変換後その一部を取り出し 0 と 0 以外を調べて論理値に変換する。0 なら 0（偽）を、0 以外なら 1（真）を返す。

#### 例

```
> print @boolb(0012)      @boolb(1200)
1      0
> print @boolw(0012)      @boolw(120000)
1      0
> print @booldw(-1)       @booldw(0)
1      0
```

- |                    |                  |
|--------------------|------------------|
| (12)@absb(<数値>ne)  | 下位 8 ビットの絶対値を返す  |
| (13)@absw(<数値>ne)  | 下位 16 ビットの絶対値を返す |
| (14)@absdw(<数値>ne) | 下位 32 ビットの絶対値を返す |

パラメータ

<数値>ne          数値

説明

<数値>ne を形式に従い 32 ビット値（バイナリ）に変換後その一部を取り出し絶対値を求める。8 ビット値、16 ビット値、32 ビット値の MSB を符号ビットとし、MSB が 1 なら負の整数と判断する（2 の補数形式）。

例

```
> print @absb(17f) @absw(17fff) @absdw(7fffffff)
7F    7FFF    7FFFFFFF
> print @absb(17f+1) @absw(17fff+1) @absdw(7fffffff+1)
80    8000    80000000
> print @absb(-1) @absw(-1) @absdw(-1)
01    0001    00000001
> print @absb(-12) @absw(-1234) @absdw(-12345678)
12    1234    12345678
> print @absb(-128t) @absw(-32768t) @absdw(-2147483648t)
80    8000    80000000
> print @absb(128t) @absw(32768t) @absdw(2147483648t)
80    8000    80000000
```

- |                   |                            |
|-------------------|----------------------------|
| (15)@sxtb(<数値>ne) | 下位 8 ビットを符号拡張し 32 ビット値とする  |
| (16)@sxtw(<数値>ne) | 下位 16 ビットを符号拡張し 32 ビット値とする |

パラメータ

<数値>ne          数値または数値式

説明

<数値>ne を形式に従い 32 ビット値（バイナリ）に変換後その一部を取り出し絶対値を。8 ビット値、16 ビット値の MSB を符号ビットとし、MSB を上位ビットへ拡張する。

例

```
> print @sxtb(17f) @sxtw(17fff)
0000007F    00007FFF
> print @sxtb(17f+1) @sxtw(17fff+1)
FFFFFF80    FFFF8000
```

- |                     |                     |
|---------------------|---------------------|
| (17)@signb(<数値>ne)  | 下位 8 ビットの符号ビットを調べる  |
| (18)@signw(<数値>ne)  | 下位 16 ビットの符号ビットを調べる |
| (19)@signdw(<数値>ne) | 下位 32 ビットの符号ビットを調べる |

パラメータ

<数値>ne          数値

説明

<数値>ne を形式に従い 32 ビット値（バイナリ）に変換後その一部を取り出し絶対値を。8 ビット値、16 ビット値の MSB を符号ビットとして返す。

例

```
> print @signb(17f) @signw(17fff) @signdw(7fffffff)
```

```

0      0      0
> print @signb(17f+1)      @signw(17fff+1)      @signdw(7fffffff+1)
1      1      1

```

- (20)@scmpb(<数値 1>ne, <数値 2>ne)      符号付きバイト値での比較  
 (21)@scmpw(<数値 1>ne, <数値 2>ne)      符号付きワード値での比較  
 (22)@scmpdw(<数値 1>ne, <数値 2>ne)      符号付きダブルワード値での比較

#### パラメータ

<数値 1>ne      数値  
 <数値 2>ne      数値

#### 説明

<数値 1>ne と<数値 2>ne を形式に従い 32 ビット値（バイナリ）に変換後その一部を取り出して符号付き整数として大小比較を行う。

<数値 1>ne > <数値 2>ne なら 1（真）を返し、そうでないなら 0（偽）を返す。

#### 例

calc コマンドでは符号なし正の整数として比較するので結果の違いを示します。

```

> calc -1>0
00000001H, 1T(+1T), 1
> print @scmpb(-1, 0)      @scmpw(-1, 0) @scmpdw(-1, 0)
0      0      0
> print @scmpb(ff, 0)      @scmpw(ff, 0) @scmpdw(ff, 0)
0      1      1

```

- (23)@inc(<回数>ne,<開始>ne[,<差>ne])      増加するバイト列に  
 (24)@incw(<回数>ne,<開始>ne[,<差>ne])      増加するワード列に  
 (25)@dec(<回数>ne,<開始>ne[,<差>ne])      減少するバイト列に  
 (26)@decw(<回数>ne,<開始>ne[,<差>ne])      減少するワード列に

#### パラメータ

<回数>ne      繰り返し回数  
 <開始>ne      開始数値  
 <差>ne      増加/減少差分。省略されたときは 1。

#### 説明

増加/減少するバイト/ワード文字列を生成する。16進数文字列の並びとなり空白で区切る。

#### 例

```

> print @inc(5, 80)
80 81 82 83 84
> print @dec(5, 80)
80 7F 7E 7D 7C
> print @inc(5, 80, 2)
80 82 84 86 88
> print @dec(5, 80, 2)
80 7E 7C 7A 78
>
> print @incw(5, -1)
FFFF 0000 0001 0002 0003
> print @decw(5, -1)
FFFF FFFE FFFD FFFC FFFB
> print @incw(5, -1, 2)
FFFF 0001 0003 0005 0007

```

```
> print @decw(5, -1, 2)
FFFF FFFD FFFB FFF9 FFF7
```

## 8.2.2 メモリ/I/O アドレス

(1) @addr(<アドレス>ne[, <バンク>ne])

<addr>[. <bank>]形式の文字列に

パラメータ

<アドレス>ne      メモリ/I/O アドレス  
<バンク>ne          バンク番号

説明

メモリ操作コマンドで使用する<アドレス>[. <バンク番号>]形式の文字列を返す。アドレス、バンク番号は16進数文字列。

例

```
> print @addr(1000+100)
1100
> print @addr(1000+100, 1)
1100.1
```

(2) @sect(<領域名>i[, <オフセット>ne])

領域内アドレスを16進数文字列に

パラメータ

<領域名>i            メモリ/I/O 領域名  
<オフセット>ne      領域内のオフセット

説明

領域の先頭からオフセット位置のメモリ/I/O アドレスを返す。

例

```
> ;                    data   io   endian addressing
> define_mem 10000 10000 LE    BA
> add_mem_area    ram    8000 2000 2 RW
> print @sect(ram)
8000
> print @sect(ram, 1023)
9023
> print @addr(@sect(ram, 1023), 1)
9023.1
```

## 8.2.3 スクリプトパラメータ

スクリプトをコマンドラインから起動する時の構文は以下のとおりです。「[6.2 スクリプト機能](#)」を参照。

<ファイル名> [<オプションパラメータリスト>] [:<注釈>] <改行>

ここで指定されたオプションパラメータをスクリプト内のコマンドラインへ埋め込むための定義済み関数型マクロの説明します。

(1) @ipara(<パラメータ番号>ne[, <サブ>ne[, <個数>ne]])

パラメータ

<パラメータ番号>ne    オプションパラメータの番号

<サブ>ne オプションパラメータ内の要素番号。省略するとパラメータ全体。  
 <個数>ne オプションパラメータ内から取り出す要素数。省略すると1。

#### 説明

オプションパラメータの中から順番を指定して取り出す方法を提供する。

@ipara(0)はスクリプトファイル名を返す。

@ipara(1)は1番目のパラメータ全体を取り出す。

@ipara(1,0)は1番目のパラメータの中から先頭の要素を取り出す。

@ipara(1,1)は1番目のパラメータの中から2番目の要素を取り出す。

#### 例

まず以下のようなスクリプトを作成します。

```
define ipara { ¥
    trace ¥@ipara(0) = @ipara(0); ¥
    trace ¥@ipara(1) = @ipara(1); ¥
    trace ¥@ipara(1,0) = @ipara(1,0); ¥
    trace ¥@ipara(1,1) = @ipara(1,1); ¥
    trace ¥@ipara(1,2) = @ipara(1,2); ¥
    trace ¥@ipara(1,2,2) = @ipara(1,2,2); ¥
    trace ¥@ipara(2) = @ipara(2); ¥
    trace ¥@ipara(2,0) = @ipara(2,0); ¥
    trace ¥@ipara(2,1) = @ipara(2,1); ¥
    trace ¥@ipara(2,1,2) = @ipara(2,1,2); ¥
    trace ¥@ipara(3) = @ipara(3); ¥
}
include $ipara 11000 -DTEST3="xyz 123" -S ;①
include $ipara -b11 22 33 44 -x 123 xyz "string" -S ;②
```

これを実行した①と②の結果を以下に挙げます。

```
> include $ipara 11000 -DTEST3="xyz 123" -S ;①
@ipara(0) = E:¥cmtoy-300¥bin¥sample_script_ipara.txt(23).tmp
@ipara(1) = 11000
@ipara(1,0) = 11000
@ipara(1,1) =
@ipara(1,2) =
@ipara(1,2,2) =
@ipara(2) = -DTEST3="xyz 123"
@ipara(2,0) = -DTEST3="xyz 123"
@ipara(2,1) =
@ipara(2,1,2) =
@ipara(3) = -S
>
> include $ipara -b11 22 33 44 -x 123 xyz "string" -S ;②
@ipara(0) = E:¥cmtoy-300¥bin¥sample_script_ipara.txt(25).tmp
@ipara(1) = -b11 22 33 44
@ipara(1,0) = -b11
@ipara(1,1) = 22
@ipara(1,2) = 33
@ipara(1,2,2) = 33 44
@ipara(2) = -x 123 xyz
@ipara(2,0) = -x
@ipara(2,1) = 123
@ipara(2,1,2) = 123 xyz
@ipara(3) = "string"
```

## (2) @spara(<パターン>s[,<サブ>ne[,<個数>ne]])

### パラメータ

<パターン>s オプションパラメータの識別文字列 (-に続く文字列)  
<サブ>ne オプションパラメータ内の要素番号  
<個数>ne オプションパラメータ内から取り出す要素数。省略するとすべて。

### 説明

オプションパラメータの中からオプション識別文字を指定して取り出す方法を提供する。  
@spara(-b)は-b オプション全体を取り出す。識別文字-bを除いた部分。  
@spara(-b,0)は-b オプションの識別文字-bを取り出す。  
@spara(-b,1)は-b オプションの識別文字を除いた全体を取り出す。  
@spara(-b,1,2)は-b オプションの識別文字を除いた2番目の要素を取り出す。

### 例

まず以下のようなスクリプトを作成します。

```
define spara { ¥
    trace ¥@spara(-D) = @spara(-D); ¥
    trace ¥@spara(-D,1) = @spara(-D,1); ¥
    trace ¥@spara(-x) = @spara(-x); ¥
    trace ¥@spara(-b) = @spara(-b); ¥
    trace ¥@spara(-b,0) = @spara(-b,0); ¥
    trace ¥@spara(-b,1) = @spara(-b,1); ¥
    trace ¥@spara(-b,1,2) = @spara(-b,1,2); ¥
}

include $ipara 11000 -DTEST3="xyz 123" -S ;①

include $ipara -b11 22 33 44 -x 123 xyz "string" -S ;②
```

これを実行した①と②の結果を以下に挙げます。

```
> include $spara 11000 -DTEST3="xyz 123" -S ;①
@spara(-D) = TEST3="xyz 123"
@spara(-D,1) = TEST3="xyz 123"
@spara(-x) =
@spara(-b) =
@spara(-b,0) =
@spara(-b,1) =
@spara(-b,1,2) =
>
> include $spara -b11 22 33 44 -x 123 xyz "string" -S ;②
@spara(-D) =
@spara(-D,1) =
@spara(-x) = 123 xyz
@spara(-b) = 11 22 33 44
@spara(-b,0) = -b
@spara(-b,1) = 11 22 33 44
@spara(-b,1,2) = 11 22
```

## 8.2.1 システムの状態

### (1) @tapp(<種別>ne)

### パラメータ

<種別>ne            問い合わせ種別  
                       0：カーネルがロード済みか  
                       1：ユーザアプリケーションがロード済みか  
                       2：CPU 実行中（リセット済み）か

#### 説明

Cmtoy のユーザアプリケーションの状態を問い合わせる。  
 @tapp(0)      カーネルがロード済みなら 1、未ロードなら 0 を返す。  
 @tapp(1)      ユーザアプリケーションがロード済みなら 1、未ロードなら 0 を返す。  
 @tapp(2)      CPU 実行中（リセット済み）なら 1、未実行なら 0 を返す。

#### 例

Cmtoy 起動直後。  
 > print @tapp(0) @tapp(1) @tapp(2)  
 1 0 0

### (2) @tdev(<デバイス>ne[, <種別>ne])

#### パラメータ

<デバイス>ne      デバイス番号  
                       0：システムタイマ  
                       1：ボリューム  
                       2：DIP スイッチ  
                       3：ボタン  
                       4：シリアル

<種別>ne            問い合わせ種別。

#### 説明

ターゲットデバイスの状態を問い合わせる。<種別>ne が省略されたら、デバイスのユニット数を返す。

@tdev(0, 0)    タイマ起動中なら 1 を返す。  
 @tdev(0, 1)    タイマモードが手動制御なら 1 を返す。  
 @tdev(0, 2)    タイマスケールを返す  
 @tdev(0, 3)    タイマログが ON なら 1 を返す。

@tdev(1, 0)    ボリュームの現在値を返す。  
 @tdev(1, 1)    ボリュームの最大値を返す。

@tdev(2, 0)    DIP スイッチ 0 の現在値を返す。チェックされていれば 1 を返す。  
 @tdev(2, 1)    DIP スイッチ 1 の現在値を返す。チェックされていれば 1 を返す。  
 @tdev(2, 2)    DIP スイッチ 2 の現在値を返す。チェックされていれば 1 を返す。  
 @tdev(2, 3)    DIP スイッチ 3 の現在値を返す。チェックされていれば 1 を返す。

@tdev(3, 0)    ボタンの現在値を返す。押されていれば 1 を返す。

@tdev(4, 0)    シリアル 1 のタイプを返す。0 または 16550  
 @tdev(4, 1)    シリアル 2 のタイプを返す。0 または 16550

#### 例

```
> print タイマ数=@tdev(0)
タイマ数=1
> print @tdev(0,0) @tdev(0,1)      @tdev(0,2)      @tdev(0,3)
0      0          64          1
>
> print ボリューム数=@tdev(1)
```



```

ボリューム数=1
> print @tdev(1,0) @tdev(1,1)
C9    FF
>
> print スイッチ数=@tdev(2)
スイッチ数=4
> print @tdev(2,0) @tdev(2,1)    @tdev(2,2)    @tdev(2,3)
0      0      0      0
>
> print ボタン数=@tdev(3)
ボタン数=1
> print @tdev(3,0)
0
>
> print シリアル数=@tdev(4)
シリアル数=2
> print @tdev(4,0) @tdev(4,1)
0      0
> serial 1 init 4 16550
> print @tdev(4,0) @tdev(4,1)
0      16550

```

### (3) @tmem(<種別>ne)

パラメータ

<種別>ne	種別
	1: メモリ空間のあり／なし
	2: IO 空間のあり／なし
	3: エンディアンの問い合わせ
	4: アドレッシングモードの問い合わせ

説明

ターゲットメモリの状態を問い合わせる。

@tmem(1)     メモリ空間が確保済みなら 1、未確保なら 0 を返す。

@tmem(2)     IO 空間が確保済みなら 1、未確保なら 0 を返す。

@tmem(3)     リトルエンディアンなら 1、ビッグエンディアンなら 2 を返す。

@tmem(4)     バイトアドレッシングなら 1、ワードアドレッシングなら 2 を返す。

例

Cmtoy 起動直後。

```

> print @tmem(1) @tmem(2) @tmem(3) @tmem(4)
0 0 0 0

```

ターゲットメモリを定義後。

```

> define_mem 10000 10000 BE    BA -f
> add_mem_area    com_io 8000 2000 2 RW
> print @tmem(1) @tmem(2) @tmem(3) @tmem(4)
1 1 2 1

```

## 8.2.2 その他

### (1) @reps(<回数>ne, <文字列>sp)     文字列の繰り返し

パラメータ

<回数>ne	繰り返し回数
<文字列>sp	空白を含む文字列。 , (カンマ) ; (セミコロン) は含めないこと。

説明

<文字列>sp を<回数>ne 回繰り返した文字列を返す。

例

```
-l 33 -w 44 -b 66 "sample"を2回繰り返す例。
> print @reps(2, -l 33 -w 44 -b 66 "sample" )
-l 33 -w 44 -b 66 "sample"  -l 33 -w 44 -b 66 "sample"
```

## (2) @format(<フォーマット>s,<数値>ne)

数値を表示形式に変換

パラメータ

<フォーマット>s 表示形式  
=: <スタイル>s[<桁数>ne]  
<数値>ne 数値

説明

<数値>ne を指定した表示形式に変換する。

<スタイル>s := {b | B | h | H | t | T | \_}

b	: 論理値 <数値>ne が 0 以外なら "true", そうでないなら "false"
B	: 論理値 <数値>ne が 0 以外なら 1, そうでないなら 0
h	: 16 進文字列 (ゼロサプレス)
H	: 16 進文字列 (桁数に合わせて先頭にゼロを付ける)
t	: 正負の 10 進数文字列
T	: 正の 10 進数文字列
_	: 2 進数文字列 (桁数に合わせて先頭にゼロを付ける)

例

論理値表示の例です。

```
> print @format(b, 1234) @format(B, 1234)
true      1
> print @format(b, 0) @format(B, 0)
false     0
```

16 進数文字列表示の例

```
> print @format(h, 1234) @format(H, 1234)
1234H      1234H
> print @format(h8, 1234) @format(H8, 1234)
1234H      00001234H
```

10 進数文字列表示の例

```
> print @format(t, -1) @format(T, -1)
-1t        4294967295t
> print @format(t, 1234t) @format(T, 1234t)
1234t      1234t
```

2 進数文字列表示の例

```
> print @format(_, -1)
1111_1111_1111_1111_1111_1111_1111_1111_
> print @format(_, 1234)
1_0010_0011_0100_
> print @format(_20t, 1234)
0000_0001_0010_0011_0100_
```

## (3) @defined(<識別子>i[,<タイプ指定>ne])

パラメータ

<識別子>i	変数型マクロ名、定義済み変数型マクロ名、定義済み関数型マクロ名、コンソールコマンド名、コマンドマクロ名、メモリ領域名。
<タイプ指定>ne	対象識別子のタイプ指定（加算して複数指定可）
01	ユーザ定義変数型マクロ名、定義済み変数型マクロ名
02	コマンドマクロ名
04	領域名
08	定義済みコンソールコマンド名
10	定義済み関数型マクロ名

#### 説明

識別子が存在するか調べる。存在すれば 1、存在しなければ 0 を返す。タイプ指定は組み合わせて指定できる。11 を指定するとユーザ定義変数型マクロ名、定義済み変数型マクロ名と定義済み関数型マクロ名を探す。

#### 例

変数型マクロ名を調べる例です。

```
> define a
> define b $a
> define c 1234
> print @defined(a) @defined(c) @defined(d)
1 1 0
> print @defined(_R) @defined(_R, 1)
1 1
```

コンソールコマンド名を調べる例です。

```
> print @defined(clear) @defined(clear, 8)
0 1
```

変数型マクロ名と定義済み関数型マクロ名を調べる例です。

```
> print @defined(a, 1+10) @defined(byte, 1+10) @defined(clear, 1+10)
1 1 0
```

### (4) @IsEmpty(<文字列>sp)

#### パラメータ

<文字列>sp          空白を含む文字列。;(セミコロン)は含めないこと。

#### 説明

<文字列>sp の前処理の結果を調べる。結果が空文字列なら 1 を返す。

#### 例

変数型マクロ名の前処理の結果を調べる例です。

```
> define a
> define b $a
> define c 1234
>
> print @IsEmpty() @IsEmpty($a) @IsEmpty($b) @IsEmpty($c)
1 1 1 0
> print @IsEmpty(-b $a )
0
```

スクリプトのパラメータを調べる例です。

```
> print @IsEmpty(@ipara(0)) @IsEmpty(@ipara(3))
0 1
```

### (5) @note(<文字列>sp)

#### パラメータ

<文字列>sp        空白を含む文字列。;(セミコロン)は含めないこと。

説明

<文字列>sp に関係なく空文字列を返す。コマンドラインの中に埋めこんでも空文字列になるのでコマンドラインに影響を与えない。

例

```
> print 11 @note(これはnote です。 ) 22
11 22
```

## 9 スクリプト制御機能

スクリプト内にスクリプトの実行を制御する命令（スクリプト制御命令）を記述できます。空白を除く先頭文字が#の行はスクリプト制御行となります。

制御命令を使うとスクリプトの実行を停止してオペレータの指示を待ったり、中断したり、条件で実行する範囲を制御することができます。

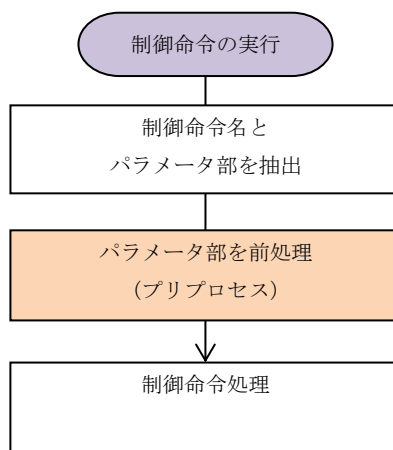
制御命令	パラメータ	説明
#exit	[<code> [-if<条件式>]]	スクリプトファイルの終了。
#abort	[-if<条件式>]	スクリプトの終了。ネストしている場合、親のスクリプトもすべて終了。
#break	[-if<条件式>]	スクリプトの実行を中断してオペレータの指示を待つ。
#msgbox	<style> -t<title> -m<メッセージ>	メッセージボックスを表示してスクリプトの実行を中断してオペレータの指示を待つ。
#if	<条件式>	<条件式>が0以外ならば以降の行を実行する。0ならば以降の行を#else または#endif の行まで読み飛ばす。
#ifdef	<識別子>	<識別子>が存在すれば以降の行を実行する。存在しなければ以降の行を#else または#endif の行まで読み飛ばす。
#ifndef	<識別子>	<識別子>が存在しなければ以降の行を実行する。存在すれば以降の行を#else または#endif の行まで読み飛ばす。
#else		#if, #ifdef, #ifndef の条件を反転する
#endif		#if, #ifdef, #ifndef の終了

### 9.1 スクリプト制御命令の構文

スクリプト制御命令の構文形式は以下のようになります。

#<制御命令名> [<パラメータリスト>] [;<注釈>] <改行>

制御命令の解析実行の概略手順は以下のようになります。



## 9.2 スクリプト制御命令一覧

### 9.2.1 #exit [<終了コード>ne] [-if<条件式>ne]

#### パラメータ

<終了コード>ne 終了コードを指定する数値または数式。省略すると 0 となる。

-if<条件式>ne <条件式>ne が 0 か 0 以外で制御が変わる。

#### 説明

スクリプトファイルを終了します。スクリプトファイルの以降の行は無視されます。終了コードは次のスクリプトファイルが実行されるまで保持され、定義済み変数型マクロ\_EXIT\_CODE で参照できます。

<条件式>ne が 0 の場合にはスクリプトを終了せずに続行します。

#### 例

```
#exit
#exit 10
#exit 10*2 -if1
```

### 9.2.2 #abort [-if<条件式>ne]

#### パラメータ

-if<条件式>ne <条件式>ne が 0 か 0 以外で制御が変わる。

#### 説明

スクリプトを終了します。ネストされていた場合すべての親のスクリプトまで終了します。

<条件式>ne が 0 の場合にはスクリプトを終了せずに続行します。

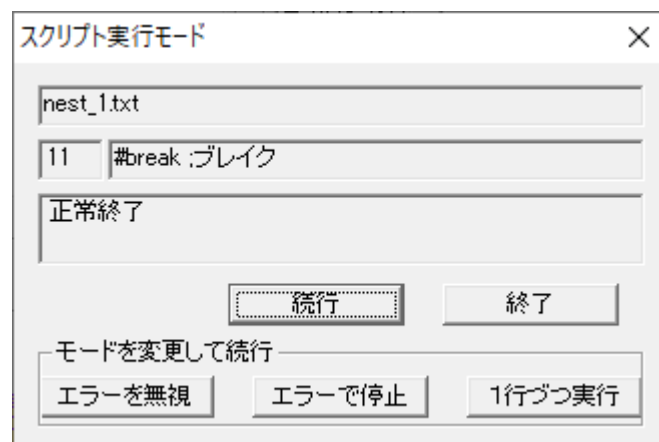
### 9.2.3 #break [-if<条件式>ne]

#### パラメータ

-if<条件式>ne <条件式>ne が 0 か 0 以外で制御が変わる。

#### 説明

スクリプトを停止して以下の「スクリプト実行モード」のメッセージボックスを表示します。



このメッセージボックスの説明は、「[2.13.1 スクリプトの実行モード](#)」を参照してください。  
<条件式>ne が 0 の場合にはスクリプトを停止せずに続行します。

#### 9.2.4 #msgbox <スタイル>ne -t <タイトル> -m <文字列>

##### パラメータ

<スタイル>ne      メッセージボックスのスタイルを指定  
1: 「OK」のボタンを持ったメッセージボックス  
2: 「はい」と「いいえ」のボタンを持ったメッセージボックス  
<タイトル>      メッセージボックスの表題  
<文字列>      メッセージボックスに表示する文字列 (;または行末まで)

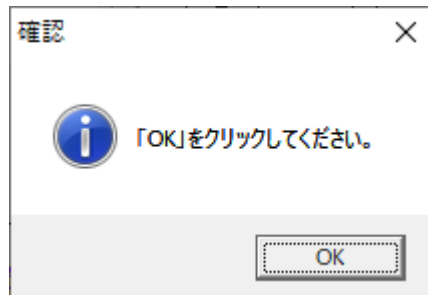
##### 説明

スクリプトを停止してメッセージボックスを表示する。  
<スタイル>ne が 1 は「OK」のボタンを持ったメッセージボックスを表示する。  
<スタイル>ne が 2 は「はい」と「いいえ」のボタンを持ったメッセージボックスを表示する。  
どのボタンをクリックして終了したかは、定義済み変数型マクロ \_MSGBOX を使って確認できる。

##### 例

```
#msgbox 1 -t 確認 -m 「OK」をクリックしてください。
```

上記の制御命令を実行した場合、以下のメッセージボックスを表示します。



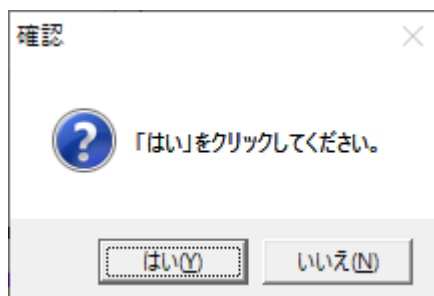
実行結果を以下に示します。

```
> #msgbox 1 -t 確認 -m 「OK」をクリックしてください。  
> print $_MSGBOX  
1
```

次に以下の制御命令を実行した場合、

```
#msgbox 2 -t 確認 -m 「はい」をクリックしてください。
```

以下のメッセージボックスを表示します。



「はい」をクリックすると

```
> #msgbox 2 -t 確認 -m 「はい」をクリックしてください。  
> print $_MSGBOX  
6
```

となり。「いいえ」をクリックすると以下のように変数型マクロ\_MSGBOX の値が変わります。

```
> #msgbox 2 -t 確認 -m 「いいえ」をクリックしてください。
> print $_MSGBOX
7
```

「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場合は以下のようにします。

```
#msgbox 2 -t 確認 -m スクリプトを続行しますか？ 「はい」をクリックすると続行します。
#if $_MSGBOX == 7
#exit
#endif
```

「いいえ」をクリックした場合は以下のようになります。

```
> #msgbox 2 -t 確認 -m スクリプトを続行しますか？ 「はい」をクリックすると続行しま
す。
> #if $_MSGBOX == 7
> #exit
Exited at "E:¥cmttoy-300¥bin¥script_sample¥sample_pdsymbol.txt" (37)
```

### 9.2.5 #if <条件式>ne

#### パラメータ

<条件式>ne          数値式

#### 説明

スクリプト制御命令#else と#endif と組み合わせて使用する。スクリプトファイル内に#if と#endif は1対1で対応している必要がある。#if と#endif の間の行に1つの#else 行を含めることができる。

#if、#else、#endif は他の#if～#endif、#if～#else、#else～#endif の間にネストすることができる。同様に#ifdef～#endif、#ifdef～#else の間にネストすることができる。同様に#ifndef～#endif、#ifndef～#else の間にネストすることができる。

CLI は<条件式>ne を評価し結果が0以外ならば次行以降のコマンドラインを実行する。#else 行が現れると、それ以降の行は#endif 行が現れるまで読み飛ばす（スキップ）する。

<条件式>ne の結果が0ならば次行から実行せずにスキップする。#else、#endif 行が現れると次行以降のコマンドラインを実行する。

#### 例

```
> undef -all
> define A 1
> define B 2
>
> list -s
> #if $A>0
> print $A
1
> #endif
>
> #if $A==$B
> ;- print A is equal B.
> ;- #else
> print A id not equal B.
A id not equal B.
> #endif
```



### 9.2.6 #ifdef <変数型マクロ名>i

#### パラメータ

<変数型マクロ名>i 変数型マクロ名

#### 説明

スクリプト制御命令 #else と #endif と組み合わせて使用する。スクリプトファイル内に #ifdef と #endif は 1 対 1 で対応している必要がある。#ifdef と #endif の間の行に 1 つの #else 行を含めることができる。

CLI は <変数型マクロ名>i が存在すれば次行以降のコマンドラインを実行する。#else 行が現れると、それ以降の行は #endif 行が現れるまで読み飛ばす（スキップ）する。

<変数型マクロ名>i が存在しなければ次行から実行せずにスキップする。#else、#endif 行が現れるまでコマンドラインをスキップする。

#### 例

```
> undef -all
> define A 1
> define B 2
>
> list -s
> #ifdef A
> print A is defined.
A is defined.
> #else
> ;- print A is not defined.
> ;- #endif
>
```

### 9.2.7 #ifndef <変数型マクロ名>i

#### パラメータ

<変数型マクロ名>i 変数型マクロ名

#### 説明

スクリプト制御命令 #else と #endif と組み合わせて使用する。スクリプトファイル内に #ifndef と #endif は 1 対 1 で対応している必要がある。#ifndef と #endif の間の行に 1 つの #else 行を含めることができる。

CLI は <変数型マクロ名>i が存在しなければ次行以降のコマンドラインを実行する。#else 行が現れると、それ以降の行は #endif 行が現れるまで読み飛ばす（スキップ）する。

<変数型マクロ名>i が存在すれば次行から実行せずにスキップする。#else、#endif 行が現れるまでコマンドラインをスキップする。

#### 例

```
> undef -all
> define A 1
> define B 2
>
> list -s
> #ifndef C
> print C is not defined.
C is not defined.
> #else
> ;- print C is defined.
> ;- #endif
>
```

### 9.2.8 #else

#### パラメータ

なし

**説明**

スクリプト制御命令`#if`,`#ifdef`,`#ifndef` と`#endif` と組み合わせて使用する。。  
`#if`,`#ifdef`,`#ifndef` の説明を参照。

**9.2.9#endif**

**パラメータ**

なし

**説明**

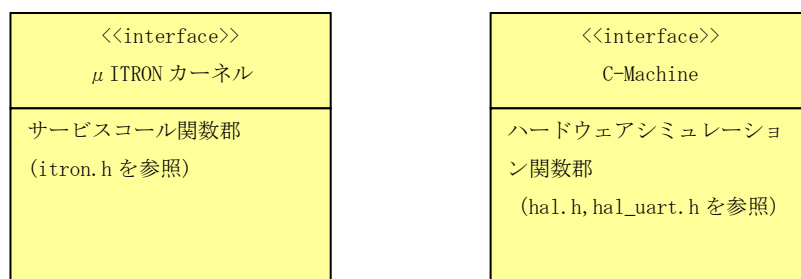
スクリプト制御命令`#if`,`#ifdef`,`#ifndef` と`#else` と組み合わせて使用する。。  
`#if`,`#ifdef`,`#ifndef` の説明を参照。

## 10 μITRON チュートリアル

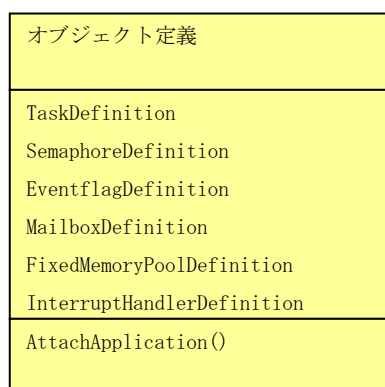
コンピュータシステムはハードウェアとソフトウェアで構成されています。ここでは、ソフトウェアをマルチタスク・プログラムの技術を使って開発する場合に考慮すべきことをプログラミング例を示して解説します。簡単なタスク構成の例からより複雑な例を取り上げて、μITRON カーネルの動きを理解する手助けをします。実際のソースコードを示しながらサービスコールの使い方を示します。そこで各サンプルプログラムを以下の視点から説明します。

- ・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 カーネル(kpd11.d11)と C-Machine(cm.d11)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	<b>tslp_tsk ext_tsk</b>
2	デバッグタスクを追加する	<b>tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx ref_mpf</b>
3	タイマタスクを追加する	<b>tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx ref_mpf snd_msg rcv_msg vchg_ifl vget_ifl</b>
4	割り込みハンドラを追加する。	<b>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</b>
5	割り込みレベル 1 と割り込みレベル 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。	<b>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</b>
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。	<b>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</b>

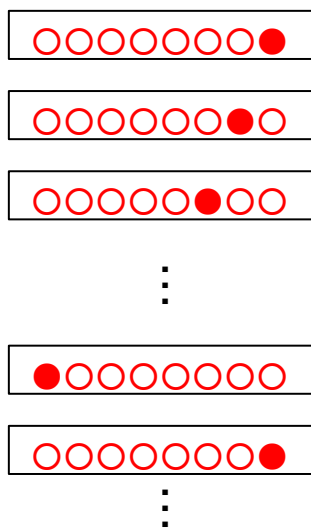
$\mu$  ITRON を使ったプログラム開発については以下の書籍も参考にしてください。各ステップのより詳細な解説、考察をしています。

[「 \$\mu\$  ITRON」入門―“組み込み系”「リアルタイム OS」の基礎 \(I・O BOOKS\) 工学社](#)

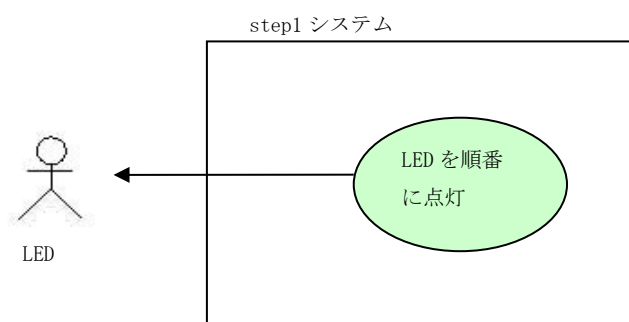
## 10.1 ステップ 1 (app1.d11)

### 10.1.1 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。

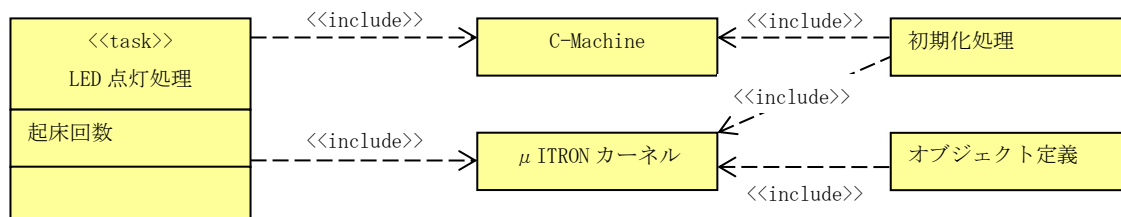


(1) ユースケース図



## 10.1.2 システム分析

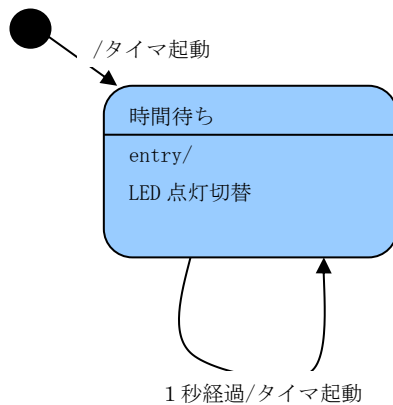
(1) クラス図



(2) 初期化処理の状態図



### (3) LED 点灯処理の状態図



#### 10.1.3実装設計

##### タスク構成

初期化处理	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 (ext\_tsk) する。

LedTask は、サービスコール tslp\_tsk で 1000 ミリ秒おきに起床する。

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	μ ITRON 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.dsw	app1.dll を作成する VisualStudio6.0 のワークスペースファイル
makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル
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
app1_08¥	
app.sln	app1.dll を作成する Visual C++ 2008 Express Edition のソリューションファイル
app1_17¥	
app.sln	app1.dll を作成する Visual Studio 2017 Professional のソリューションファイル

##### 使用サービスコール

tslp\_tsk ext\_tsk

## 10.2ステップ2 (app2. dll)

### 10.2.1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。

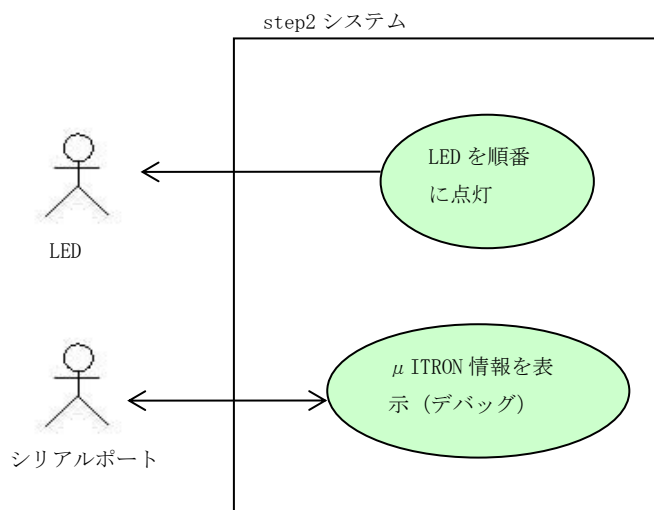
ハイパーターミナルからデバッグコマンドを使ってシステムの状態が参照できるようにする。

デバッグコマンドと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 $\mu$ ITRONカーネルのバージョン表示
d[w] <addr> [<count>]	メモリのバイト (ワード) ダンプ
s[w] <addr>	メモリの書き換え、バイト (ワード) 単位
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]	メールボックス (一覧) の表示
mpf [mpfid]	固定長メモリプール (一覧) の表示

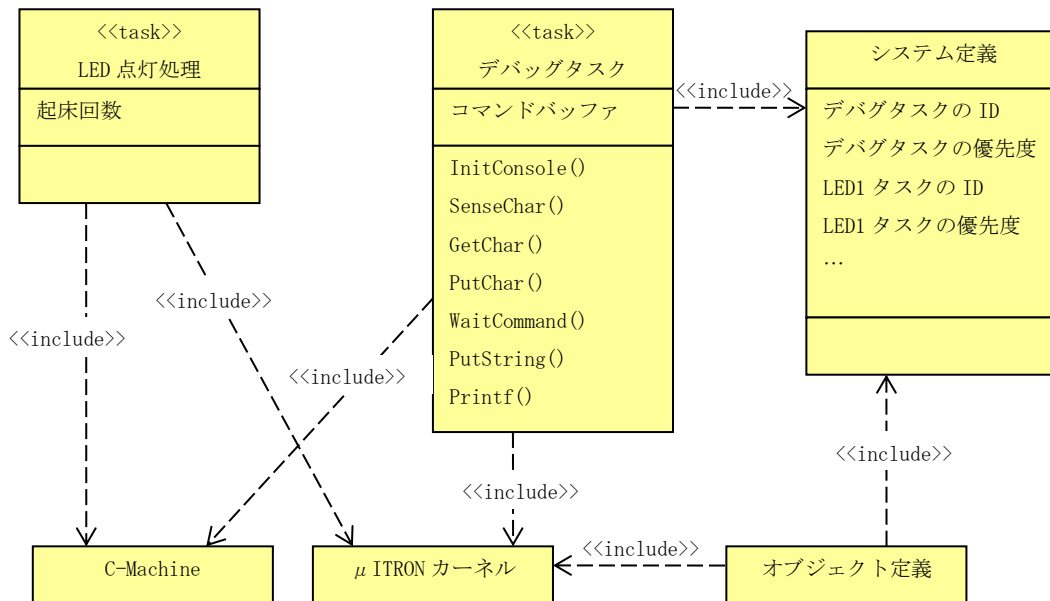
※ハイパーターミナルの代わりにPuTTYなどの端末エミュレータソフトも使えます。

#### (1) ユースケース図



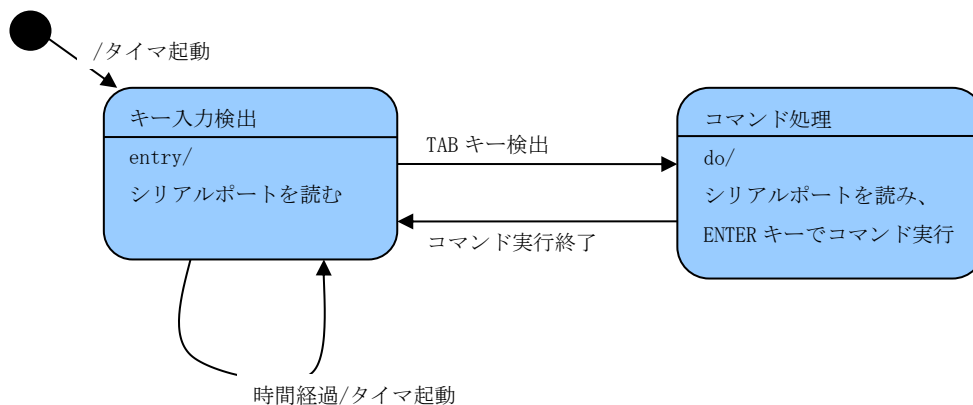
## 10.2.2 システム分析

### (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



## 10.2.3 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。読み取りは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。

タスク一覧を表示する tsk コマンドは ref\_tsk サービスコールを使っている。以下のソースコードを参照。

```
for (tskid = MIN_OBJID; tskid < MAX_TSKID; ++tskid) {
    if (ref_tsk(tskid, &tskinfo) == E_OK) {
        PrintTaskInfo(tskid, &tskinfo);
    }
}
```

このように使用できるタスク ID をすべて調べているので、存在しないタスクを参照すると以下のエラーが出力ウインドウに表示される。 [「4.1.4\(5\) サービスコールのエラー」](#) を参照。

```
00014ff1: E_NOEXS by ref_tsk in task[2].debug
```

他のオブジェクト一覧を表示する場合も同様。

②では関数 DebugGetChar を使っている。この関数を以下のように変更した。(V3.00 で変更)

```
/*デバグコンソールから1文字入力待つ*/
char DebugGetChar(void)
{
    int c = -1;
    if (DebugPort > 0) {
        while((c = halSerialReadChar(0)) < 0) {
            dly_tsk(10); /*2023,2,27 追加*/
        }
    }
    return c;
}
```

これでキー入力を待つ間も優先度の低いタスクが動くことができる。

## ファイル構成

step2¥	
kernel_cfg.c	μ ITRON 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.dsw	app2.dll を作成する VisualStudio6.0 のワークスペースファイル
makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル
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
app2_08¥	
app.sln	app2.dll を作成する Visual C++ 2008 Express Edition のソリューションファイル
app2_17¥	
app.sln	app2.dll を作成する Visual Studio 2017 Professional のソリューションファイル

※V3.00 の変更 : debug.c の先頭で CRT\_SECURE\_NO\_WARNINGS を定義 (警告 C4996 を抑制)

使用サービスコール

tslp\_tsk ext\_tsk ref\_tsk ref\_sem ref\_flg ref\_mbx ref\_mpf

#### 10.2.4 ハイパーターミナルの設定方法

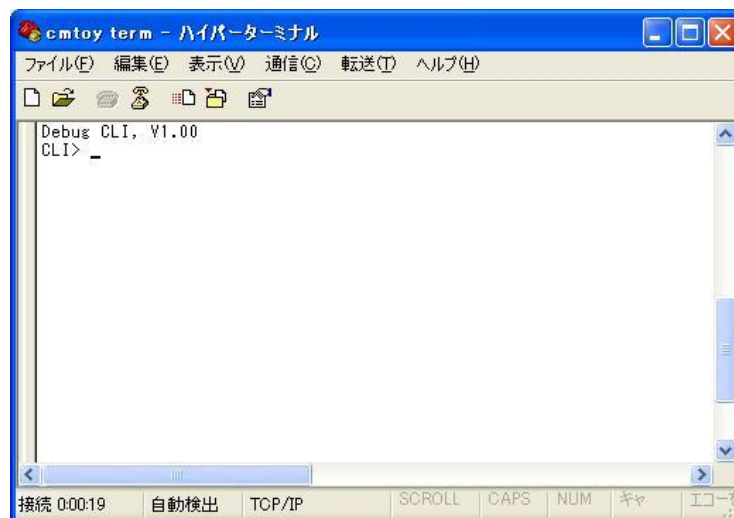
Cmtoy はシリアルポートを Winsock (TCP/IP) でシミュレートするので、Cmtoy を実行している同じ PC のハイパーターミナルと接続するためには、ハイパーターミナルを以下のように設定してください。



「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。

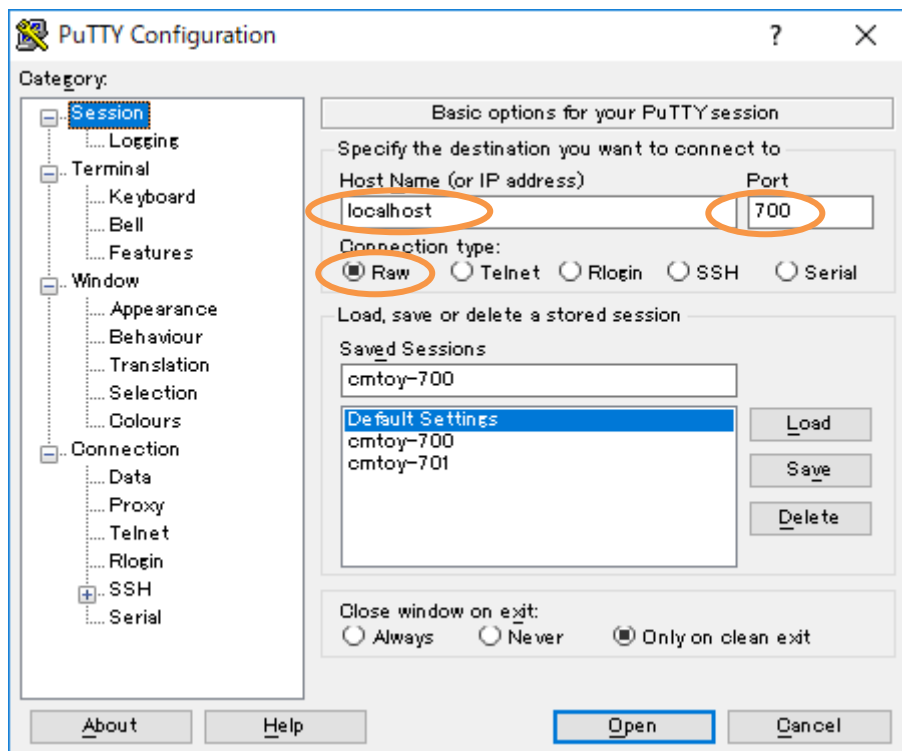


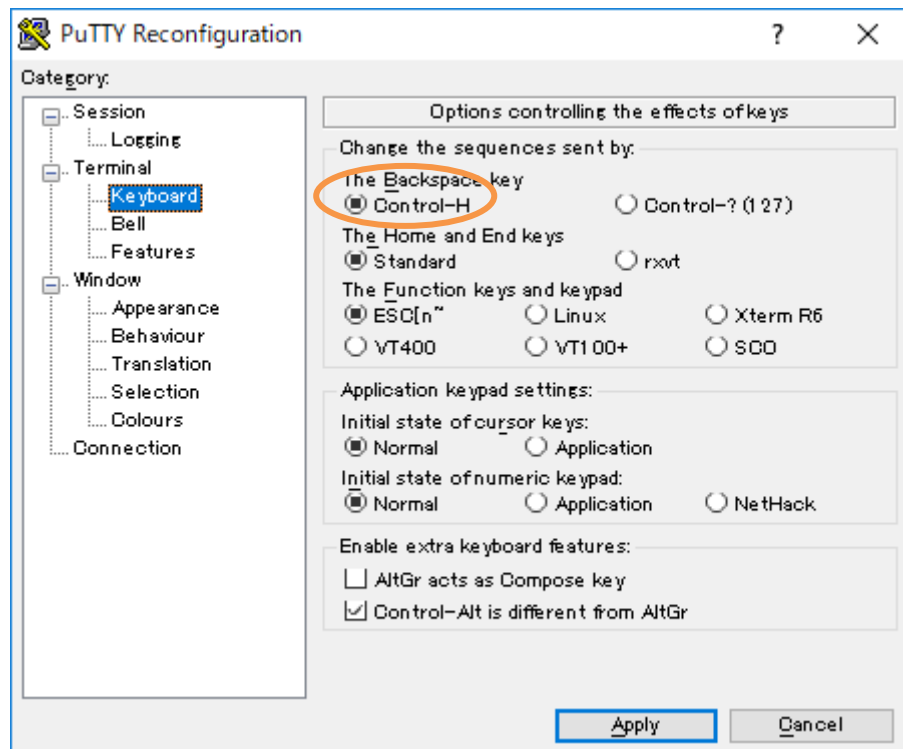
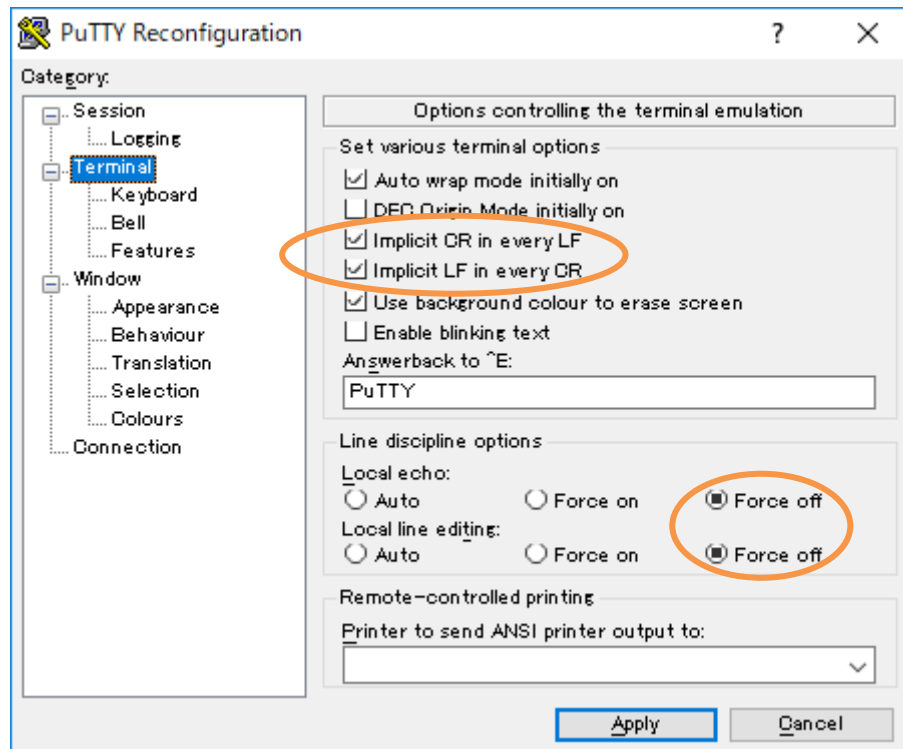
ここで ‘help’ と入力し最後に ENTER キーを押すと以下のように、コマンドリストが表示されます。

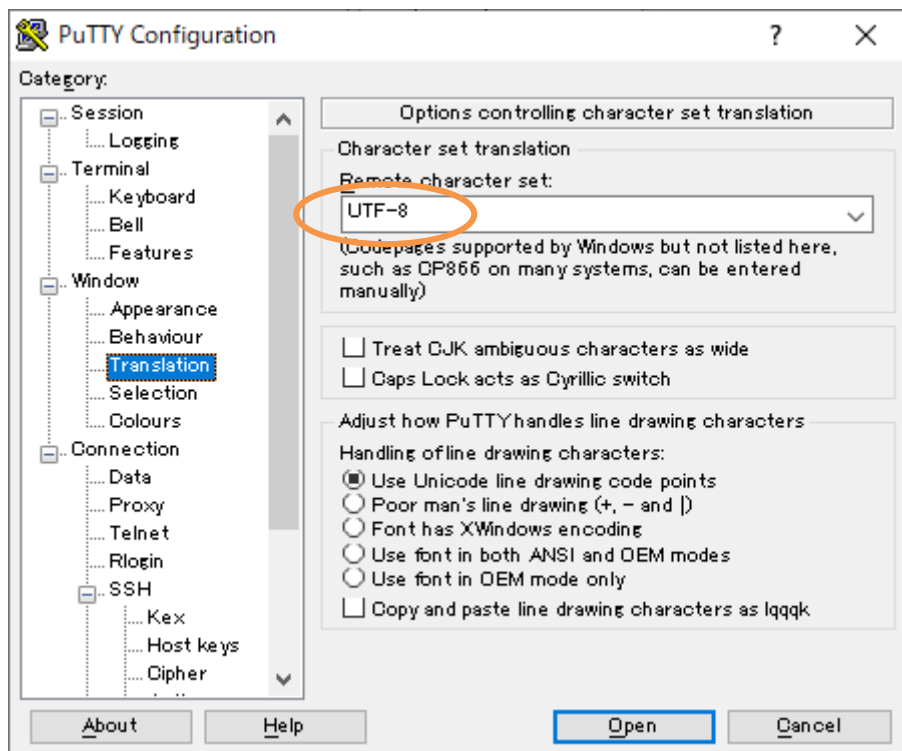


### 10.2.5 PuTTY の設定方法

ハイパーターミナルの代わりに 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 /pʌtɪ/ 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。以下は Release 0.70 の設定画面です。







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。

```

Cmtoy - 700
Debug CLI, V1.03
CLI> help
Debug CLI Command List:
ver
mode { echo | noecho }
d[w] <addr> [<count>]
s[w] <addr>
inb <port>
inw <port>
outb <port> <byte>
outw <port> <word>
tsk [<tskid>]
sem [<semid>]
flg [<flgid>]
mbx [<mbxid>]
mpf [<mpfid>]
OK
CLI> tsk
Task list:
01 "init"      : pri = 1, stat = DORMANT
02 "debug"     : pri = 2, stat = RUN
04 "led"       : pri = 8, stat = WAIT by slp_tsk, lefttmo = 4
OK

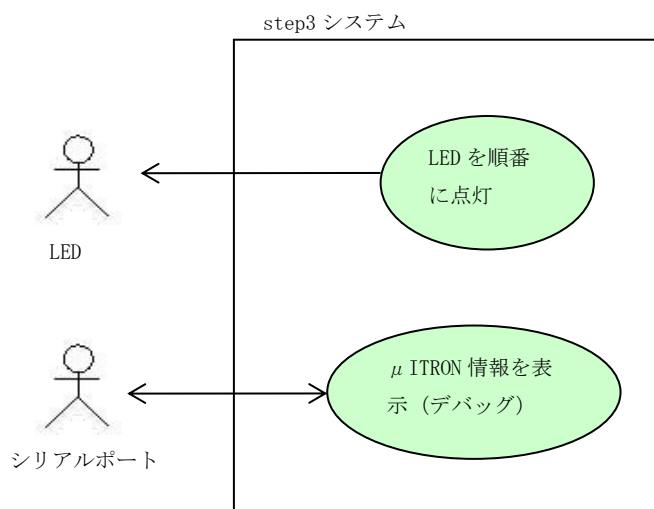
```

## 10.3ステップ3 (app3.d11)

### 10.3.1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ールアドレス 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 メールアドレス ID	送信メッセージ (アドレス)
0	1 0	1 0 0	1	x x x x
1	0	5 0	2	y y y y
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「現在のカウンタ値」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、「現在のカウンタ値」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「現在のカウンタ値」を-1し、0になったら「送信先メールアドレス ID」で指定されるメールアドレスに「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。

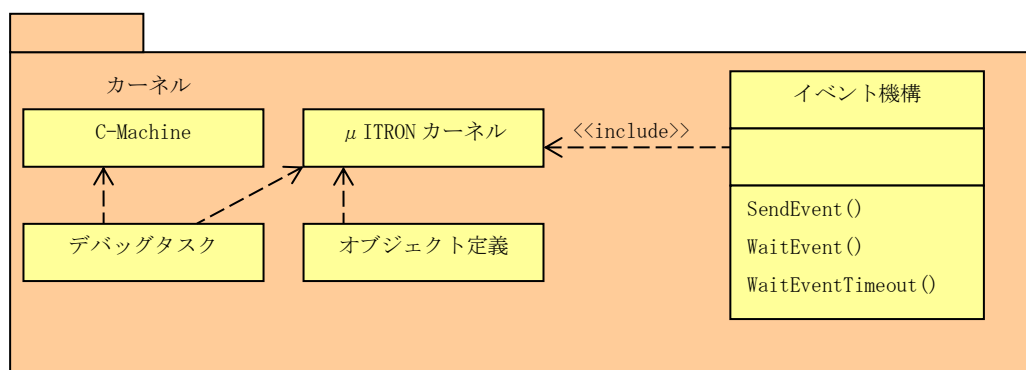
メッセージを受信したら LED の点灯位置を変更

`timStartTimer` 関数でタイマを再起動

## 10.3.2 システム分析

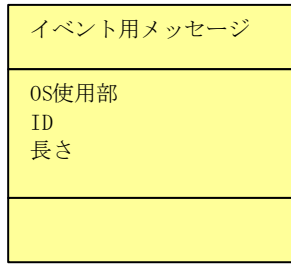
### (1) クラス図

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、 $\mu$ ITRON 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）

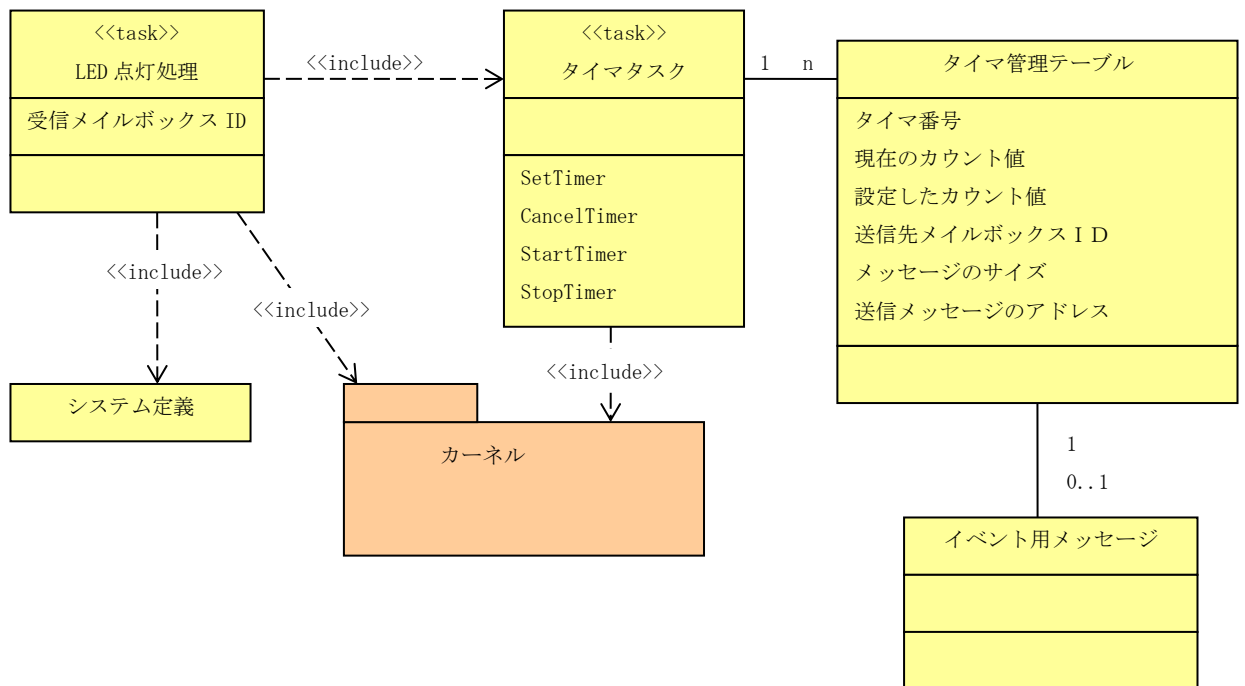


以降のシステム分析ではこの「カーネル」パッケージを使い説明します。

イベント機構は $\mu$ ITRONのメールアドレ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。

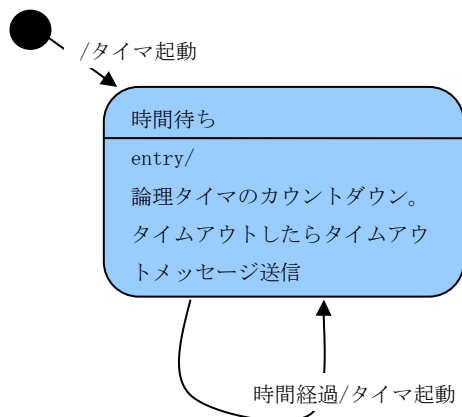


このイベント用メッセージを使いシステム分析結果を以下に示します。



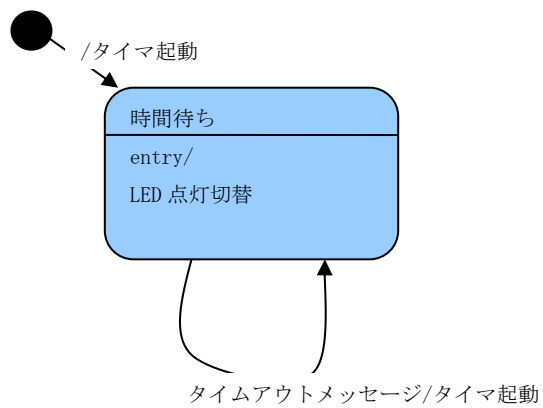
以後、C-Machine、 $\mu$  ITRON カーネル、デバッグタスクをパッケージ「カーネル」としてクラス図に記述します。

## (2) タイマタスクの状態図





### (3) LED 点灯処理の状態図



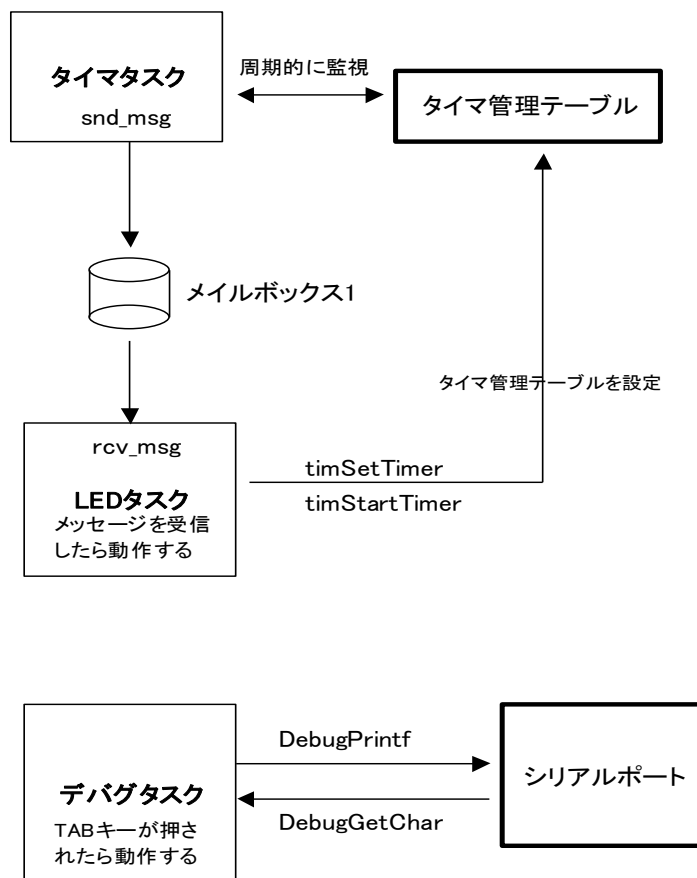
### 10.3.3実装設計

#### タスク構成

初期化处理	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	μ ITRON 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.dsw	app3.dll を作成する VisualStudio6.0 のワークスペースファイル
makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル
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
app3_08¥	
app.sln	app3.dll を作成する Visual C++ 2008 Express Edition のソリューションファイル
app3_17¥	
app.sln	app3.dll を作成する Visual Studio 2017 Professional のソリューションファイル

※V3.00 の変更：debug.c の先頭で `_CRT_SECURE_NO_WARNINGS` を定義（警告 C4996 を抑制）

使用サービスコール

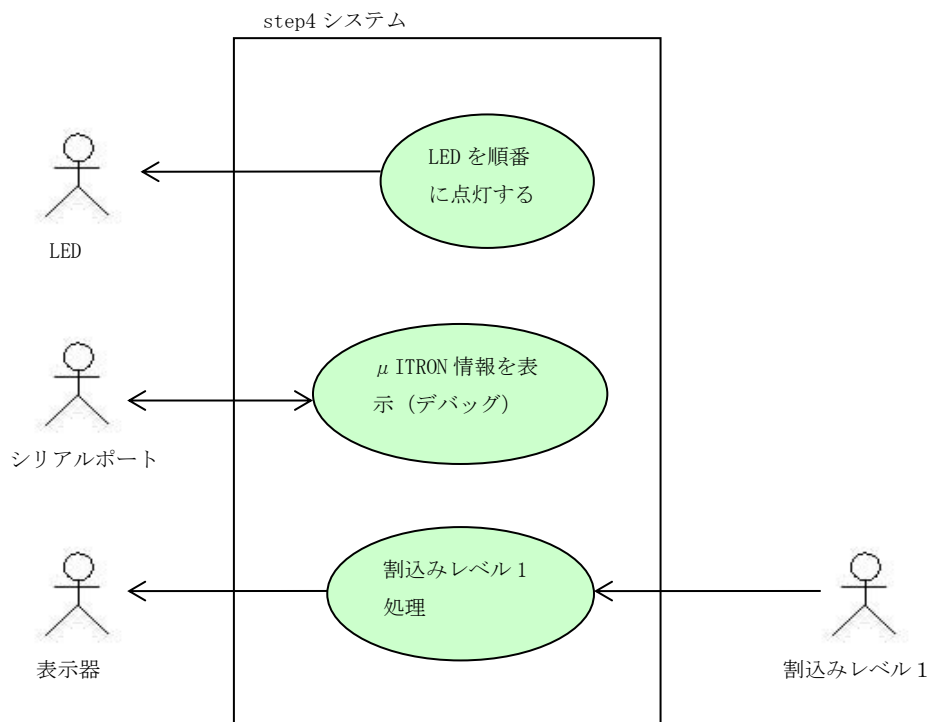
```
tslp_tsk ext_tsk ref_tsk ref_sem ref_flg ref_mbx ref_mpf snd_msg rcv_msg  
vchg_ifl vget_ifl
```

## 10.4ステップ4 (app4.d11)

### 10.4.1システム要求仕様

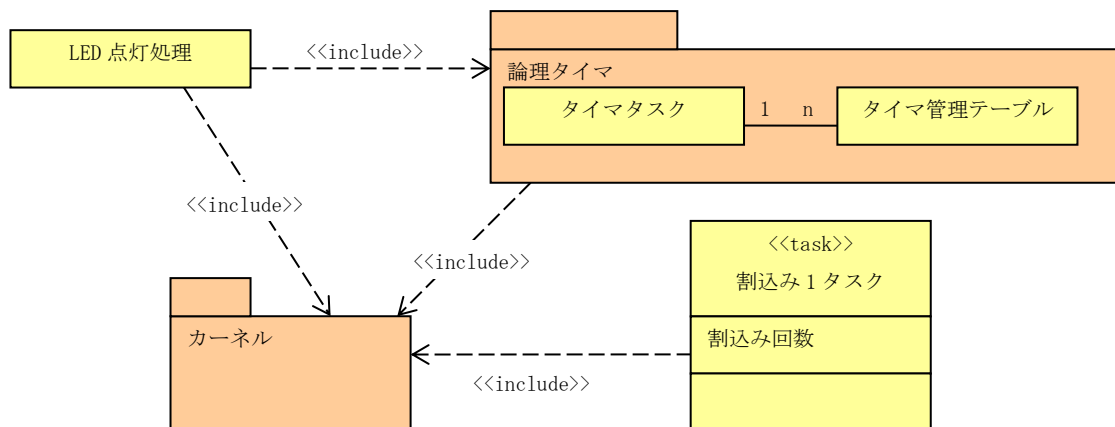
割込みレベル1の割り込みが発生したら割り込み発生数を表示器（8セグメントLED）に設定する。

#### (1) ユースケース図



### 10.4.2システム分析

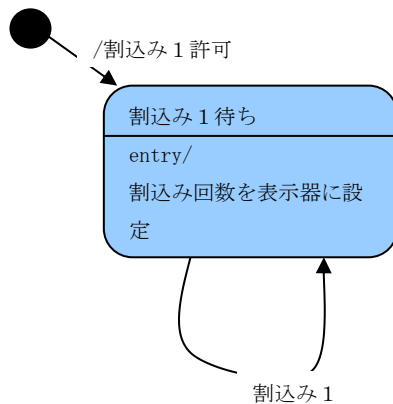
#### (1) クラス図



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) 割り込み 1 処理の状態図



### 10. 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
割り込み 1 タスク	Int1Task	セマフォ 1 で待って割り込み回数を表示器に表示

#### 割り込みハンドラ構成

割り込み 1	Int1Handler	セマフォ 1 へ信号操作
--------	-------------	--------------

#### 解説

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

#### ファイル構成

step4¥

kernel_cfg.c	μ ITRON 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	割り込み 1 タスク、INT1 割り込みハンドラ
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.dsw	app4.dll を作成する VisualStudio6.0 のワークスペースファイル
makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル
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
app4_08¥	
app.sln	app4.dll を作成する Visual C++ 2008 Express Edition のソリューションファイル
app4_17¥	
app.sln	app4.dll を作成する Visual Studio 2017 Professional のソリューションファイル

※V3.00 の変更 : debug.c の先頭で CRT\_SECURE\_NO\_WARNINGS を定義 (警告 C4996 を抑制)

使用サービスコール

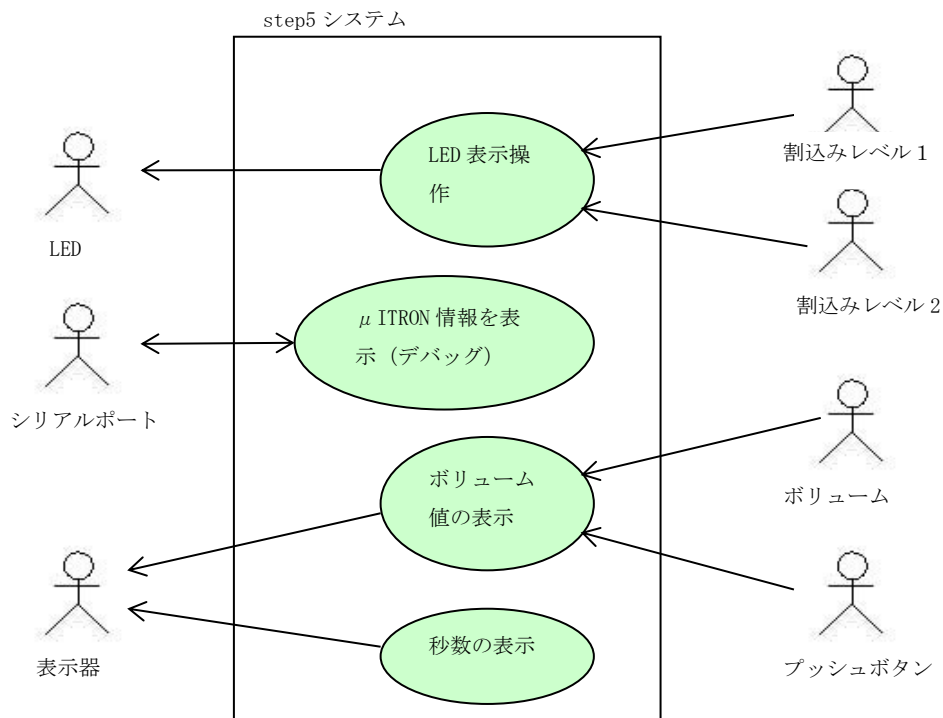
```
tslp_tsk ext_tsk ref_tsk ref_sem isig_sem wai_sem ref_flg ref_mbx snd_msg  
rcv_msg vchg_ifl vget_ifl ref_mpf
```

## 10.5ステップ5 (app5.d11)

### 10.5.1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	●	●	●	○	○	○	○	○	すべて OFF
2	●	●	○	○	○	○	○	○	
1	●	○	○	○	○	○	○	○	
0	○	○	○	○	○	○	○	○	

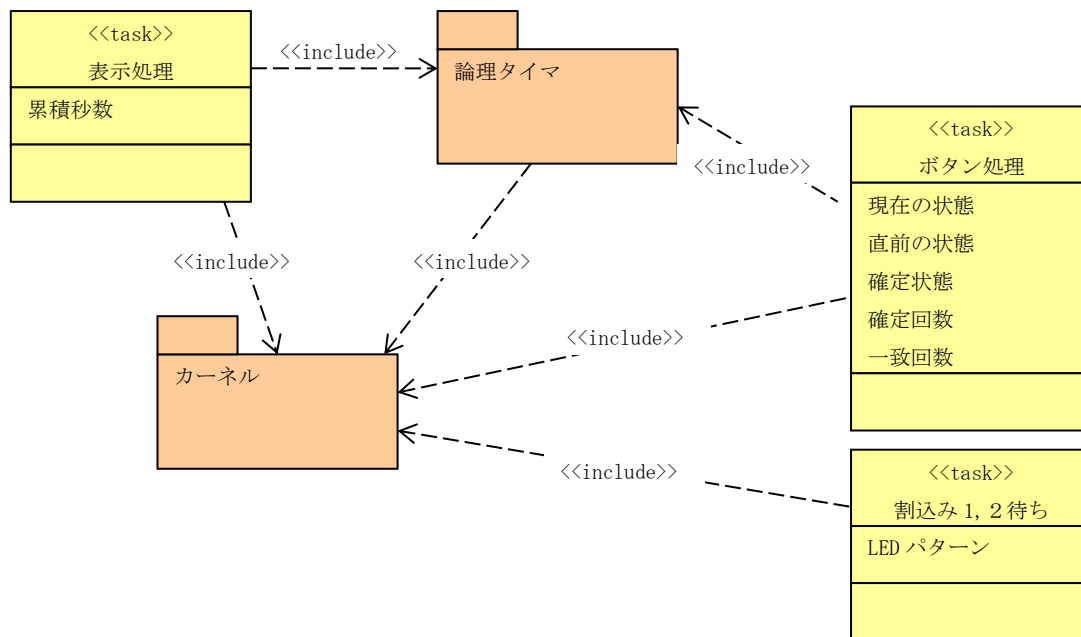
システム起動時は段階 4 の状態とする。

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。

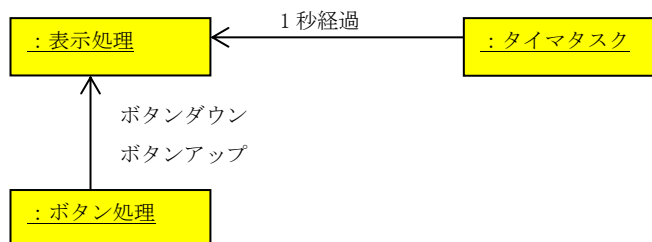
## 10.5.2 システム分析

### (1) クラス図

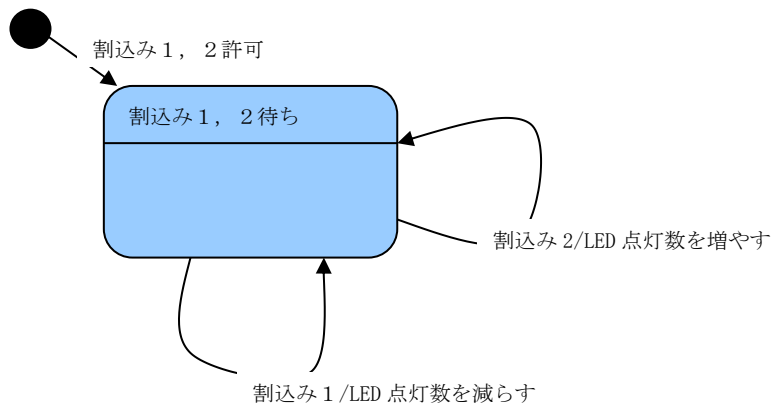


(初期化タスクは省略)

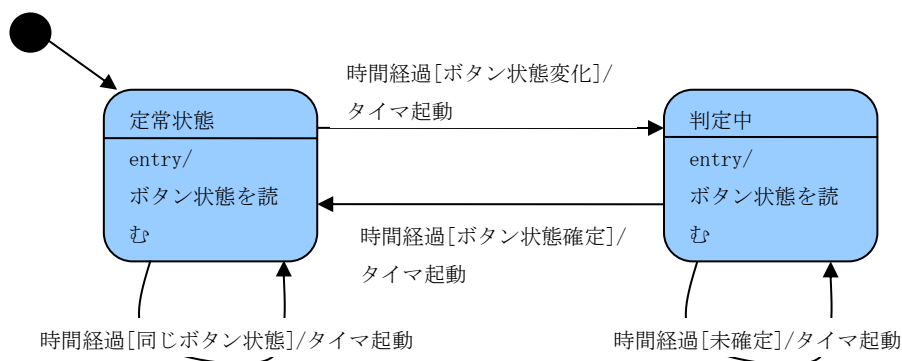
### (2) コラボレーション図



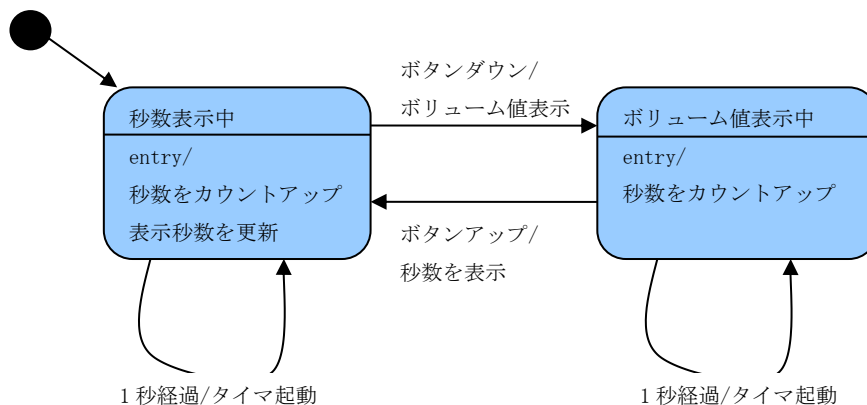
### (3) LED 操作の状態図



### (4) ボタン処理の状態図



### (5) 表示処理の状態図



## 10.5.3 実装設計

### タスク構成

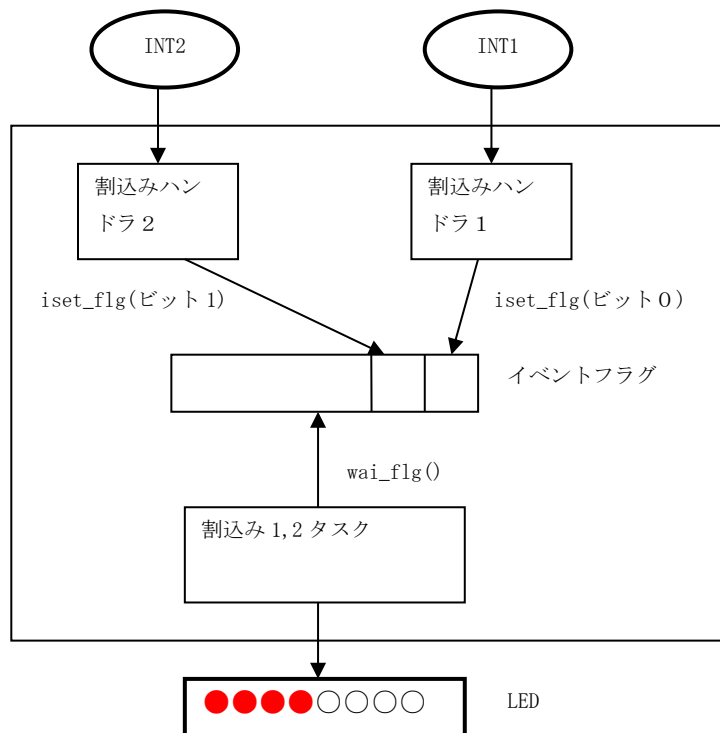
初期化处理	InitTask	LED の初期化 (すべて消灯)
タイマタスク	TimerTask	汎用的な時間管理タスク
表示処理	DisplayTask	表示器へ秒数、ボリューム値を表示。
ボタン処理	ButtonTask	ボタンの UP/DOWN のデバンス処理。
デバッグタスク	DebugTask	タスクなどの状態を表示する
LED 操作処理	Irq12Task	イベントフラグ 1 で待って割り込み 1、2 により LED の点灯状態を変える。

### 割り込みハンドラ構成

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	μ ITRON 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	割り込み 1, 2 タスク、INT1 割り込みハンドラ、INT2 割り込みハンドラ
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.dsw	app5.dll を作成する VisualStudio6.0 のワークスペースファイル
makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル
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
app5_08¥	
app.sln	app5.dll を作成する Visual C++ 2008 Express Edition のソリューションファイル
app5_17¥	
app.sln	app5.dll を作成する Visual Studio 2017 Professional のソリューションファイル



※V3.00 の変更 : debug.c の先頭で CRT\_SECURE\_NO\_WARNINGS を定義 (警告 C4996 を抑制)

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
```

## 10.6ステップ6 (app6.dll)

### 10.6.1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  $\mu$  ITRON のメー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。
- ・  $\mu$  ITRON のメー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、メッセージがメールボックスにある (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。

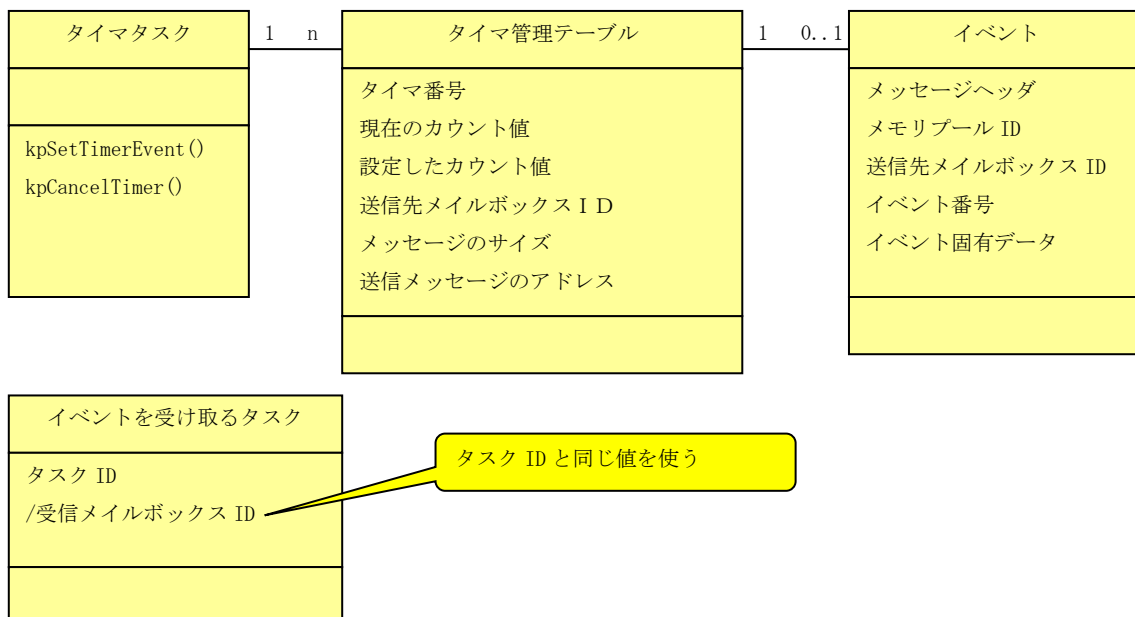
そこで、ステップ6ではイベントインスタンスの領域を固定長メモリプールのメモリブロックを使って確保するように変更します。

- ・ 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。

### 10.6.2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。



イベント機構
kpCreateEvent() kpDeleteEvent() kpSendEvent() kpWaitEvent() kpWaitEventTimeout()

### 10.6.3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ールアドレス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ールアドレスIDを指定; //
    kpSendEvent(MID_Display, pEvent); // メールボックス 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ールアドレス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	μ ITRON 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	割り込み 1, 2 タスク、INT1 割り込みハンドラ、INT2 割り込みハンドラ
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.dsw	app6.dll を作成する VisualStudio6.0 のワークスペースファイル
makefile.bcc	Borland C++ Compiler 5.5 付属の make 用メイクファイル
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
app6_08¥	
app.sln	app6.dll を作成する Visual C++ 2008 Express Edition のソリューションファイル
app6_17¥	
app.sln	app6.dll を作成する Visual Studio 2017 Professional のソリューションファイル

※V3.00 の変更 : debug.c の先頭で `_CRT_SECURE_NO_WARNINGS` を定義 (警告 C4996 を抑制)

## 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

## 11 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。通常割り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。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。割り込みボタン「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립トファイルはコンソール・コマンドの使用例となります。

### 11.1 タスクと割り込みハンドラの例

#### 11.1.1 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립トファイル

ソースファイルは以下のフォルダです。

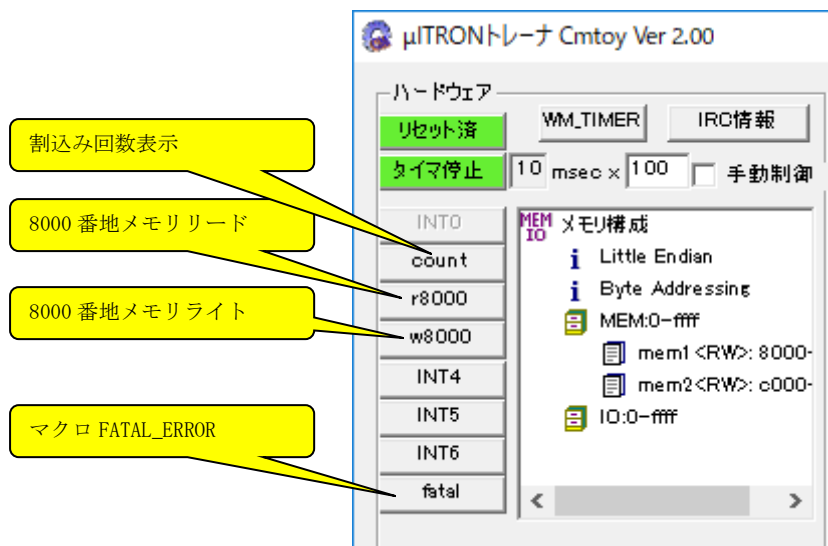
mITRON¥sample¥	
kernel_cfg.c	μITRON コンフィグレーションファイル
debug.c	シリアル 1 へ表示する DebugPrintf 関数など
debug.h	
sample.c	TimerTask, WatchButtonTask, Int1, Int7, InitHandler
test.c	Task1, Task2, Task3, Int2, Int3
test.h	
app¥	
app.dsw	app.dll を作成する VisualStudio6.0 のワークスペースファイル
StdAfx.h	VisualStudio6.0 の AppWizard 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
app_08¥	
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_17¥	
app.sln	app.dll を作成する Visual Studio 2017 Professional のソリューションファイル

#### 11.1.2 システムの概要

Cmyoy を起動して、「スク립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
- ・割り込みボタン「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。



PuTTY 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 PuTTY 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、以下のタスクと割り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            出力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            出力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            出力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) 割り込みハンドラ関数

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

### 11.1.3 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,

#### (1) void DebugPrintf(const char \*formatstring, ...);

シリアル 1 (TCP/IP ポート 700) へ文字列を出力する。プログラムの実行状態を確認するために使う。タスク関数からも割込みハンドラからも使える。debug.c で実装。

関数 [5.8.3 void halSerialWriteChar\(int SerialNo, int c\);](#) を使ってシリアル 1 (SerialNo=0) へ文字を出力する。

#### (2) マクロ CMTRACE

出力ウインドウ (現在のコンソール) へ文字列を出力する。hal.h で実装。

関数 [5.3.2 void halDebugPrintf\(const char \\*formatstring, ...\);](#) を使っている。

#### (3) マクロ FATAL\_ERROR

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 (hal.h で実装) を呼び出します。

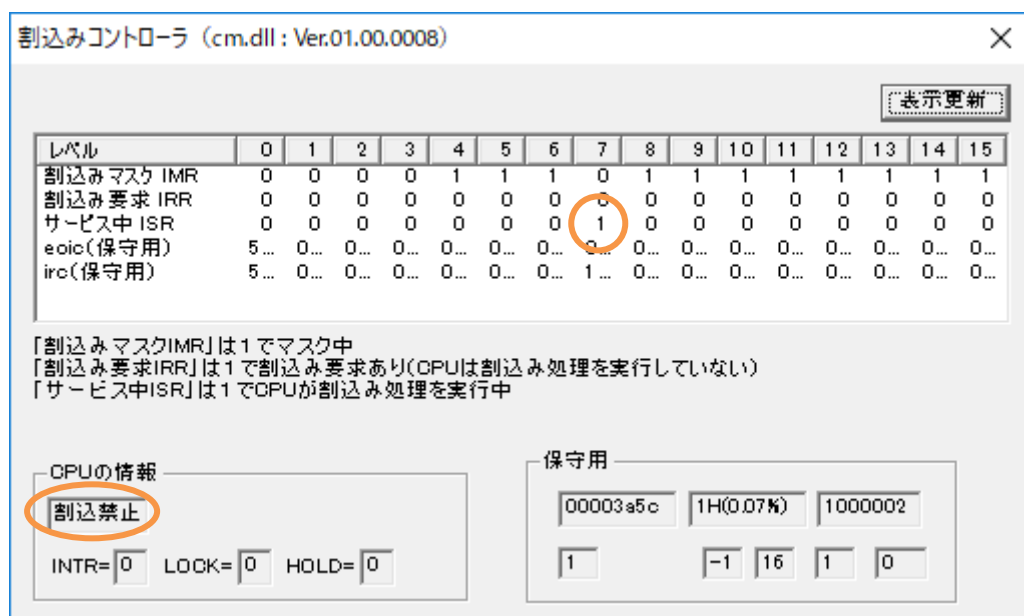
このマクロを実行すると、割込み禁止にし、タイマを停止してエラーメッセージを出力ウインドウに表示し、メッセージボックスを表示をして、無限ループに入ります。出力ウインドウには以下のエラーメッセージが表示されます。メッセージには FATAL\_ERROR を呼び出したソースファイル名と行番号が含まれます。

```
>Int 7 #1
app: handler context.
;Timer Stopped.
▲▲致命的エラー: app: Int7 invoked.
<D:¥cmttoy-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(割り込みコントローラ)の状態を確認すると以下のようになります。(CPU は割り込み禁止、IRC の ISR レベル 7 はサービス中)



## 11.2 ターゲットメモリの例

### 11.2.1 ファイル構成

実行ファイルは以下のフォルダです。

```
bin¥memory¥
  app. dll                μ ITRON アプリケーションファイル
  cminitLB. txt            リトルエンディアン、バイトアドレッシング用スクリプトファイル
  cminitLW. txt            リトルエンディアン、ワードアドレッシング用スクリプトファイル
  cminitBB. txt            ビッグエンディアン、バイトアドレッシング用スクリプトファイル
  cminitBW. txt            ビッグエンディアン、ワードアドレッシング用スクリプトファイル
  font_han. bin            フォント定義ファイル (リトルエンディアン形式)
  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¥
```

kernel_cfg.c	μITRON コンフィグレーションファイル
debug.c	デバッグタスク、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	デバッグタスクのコマンドを拡張する ExecCustomCommandLine
font.h	
app¥	
app.dsw	app.dll を作成する VisualStudio6.0 のワークスペースファイル
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
app_08¥	
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_17¥	
app.sln	app.dll を作成する Visual Studio 2017 Professional のソリューションファイル

※V3.00 の変更：debug.c の先頭で CRT\_SECURE\_NO\_WARNINGS を定義（警告 C4996 を抑制）

※V3.00 の変更：font.c の先頭で CRT\_NONSTDC\_NO\_WARNINGS を定義（警告 C4996 を抑制）

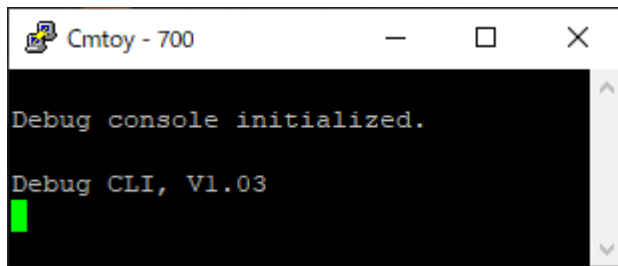
### 11.2.1 システムの概要

PuTTY を TCP/IP ポート 700（シリアル1）へ接続し、スクリプト cminitLB.txt を実行するとターゲットメモリをリトルエンディアン、バイトアドレッシングで作成します。その後 app.dll をロードし、実行を開始します。GUI は以下ようになります。



PuTTY には以下のようなメッセージが表示されます。





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、以下のタスクと割り込みハンドラが確認できます。



### (1) タスク関数

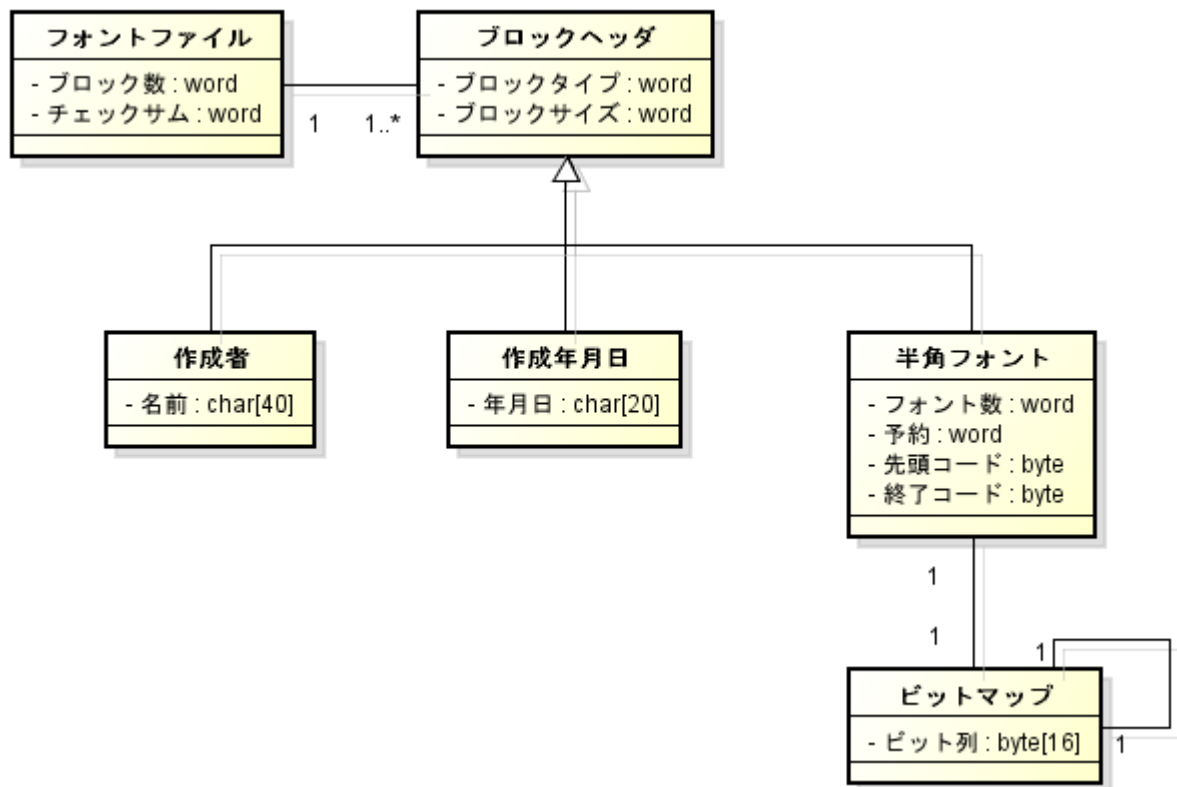
DebugTask デバッグタスク（シリアル1を入出力に使う）  
 TimerTask LEDを順次点灯、SW1(PN9/15)を監視  
 WatchButtonTask ボタンの状態を監視

### (2) 割り込みハンドラ関数

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		
.			
.			
.			
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	0002 (半角フォント)

チェックサムを  
計算する範囲  
(8 バイト)

8 バイトの和

0051	00	タイプ	
.			
.			
.			
065a	02	ブロック 4	0002 (半角フォント)
065b	00	タイプ	
.			
.			
.			

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。以下のコンソール・コマンドを使って行います。

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

ファイル内のオフセットは a000 番地からのオフセットとなります。

#### (4) デバッグコマンドの拡張

デバッ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、 $\mu$ ITRON 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

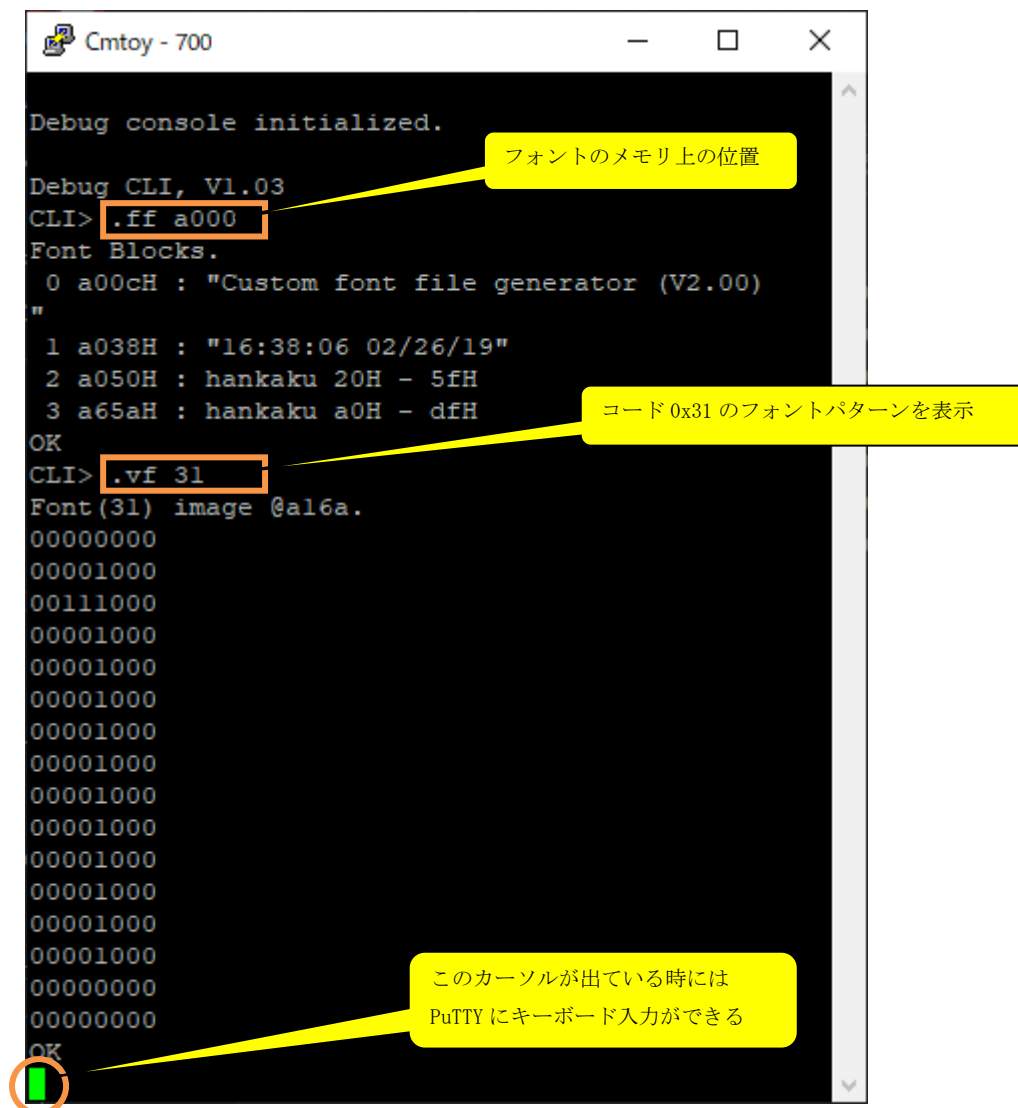
シンタックス      vf <code>

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。

PuTTY の Cmtty-700 (シリアル 1 へ接続済み) で上記のコマンドを実行すると以下ようになります。



### 11.2.2 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

#### (1) void DebugPrintf(const char \*formatstring, ...);

シリアル 1 (TCP/IP ポート 700) へ文字列を出力する。プログラムの実行状態を確認するために使う。タスク関数からも割り込みハンドラからも使える。debug.c で実装。

関数 [5.8.3 void halSerialWriteChar\(int SerialNo, int c\);](#) を使ってシリアル 1 (SerialNo=0) へ文字を出力する。

#### (2) マクロ CMTRACE

出力ウインドウ (現在のコンソール) へ文字列を出力する。hal.h で実装。

関数 [5.3.2 void halDebugPrintf\(const char \\*formatstring, ...\);](#) を使っている。

## 11. 316550 制御例

### 11.3.1 ファイル構成

実行ファイルは以下のフォルダです。

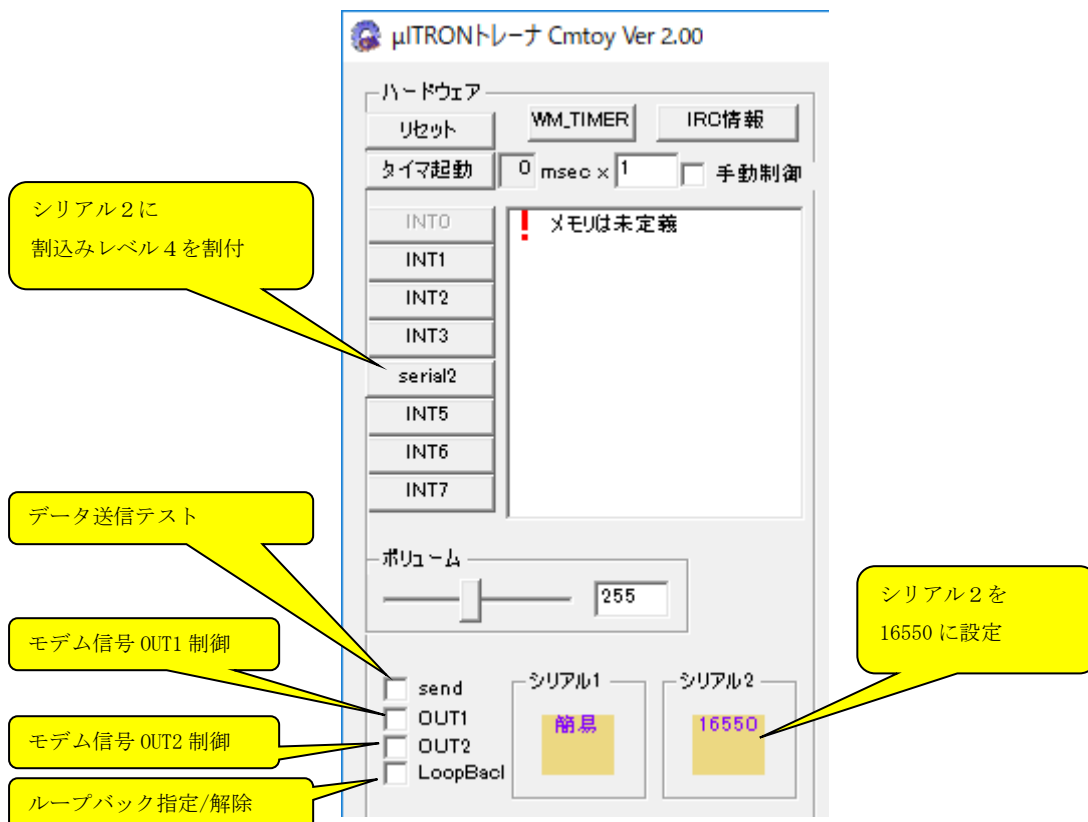
bin¥16550¥	
app.dll	μ ITRON アプリケーションファイル
sample_serial_init.txt	スクリプトファイル
sample_serial_push.txt	スクリプトファイル

ソースファイルは以下のフォルダです。

mITRON¥sample16550¥	
kernel_cfg.c	μ ITRON 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	
app¥	
app.dsw	app.dll を作成する VisualStudio6.0 のワークスペースファイル
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
app_08¥	
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_17¥	
app.sln	app.dll を作成する Visual Studio 2017 Professional のソリューションファイル

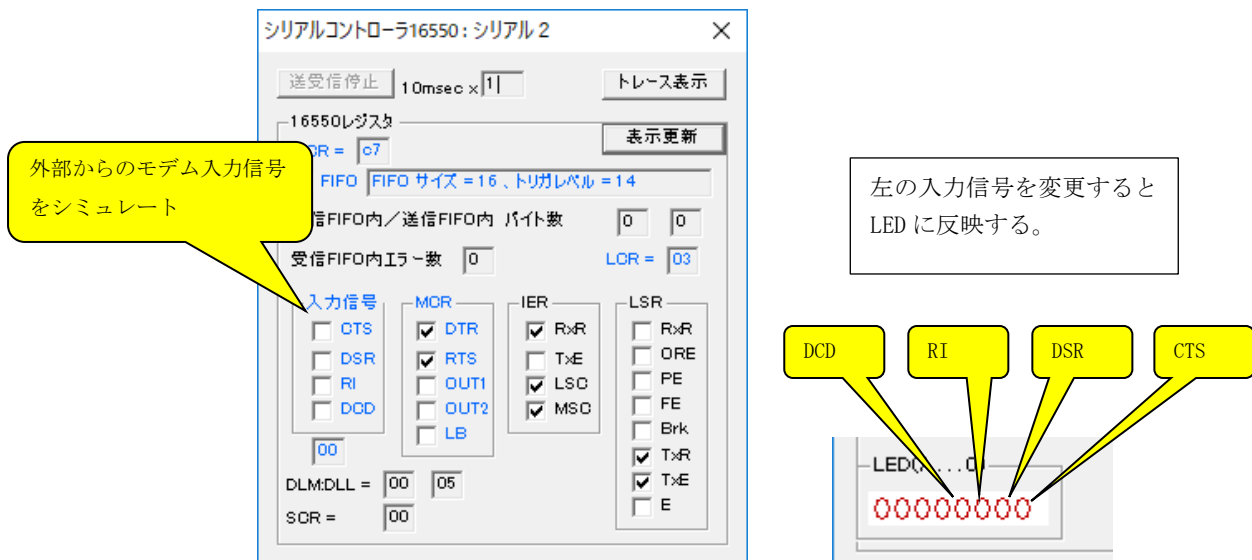
### 11.3.2 システムの概要

スクリプト sample\_serial\_init.txt を実行すると GUI は以下ようになります。



ここでハイパーターミナルまたはPuTTYをシリアル1、2へTCP/IPポート700、701を使って接続します。

次にアプリケーション app.dll を実行すると関数 Init16550()によってシリアル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、以下のタスクと割り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、DIPスイッチ0(send)の状態が変化したらシリアル2へ送信

### (2) 割り込みハンドラ関数

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

## 11.3.1 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
```

#### (1) void DebugPrintf(const char \*formatstring, ...);

シリアル 1 (TCP/IP ポート 700) へ文字列を出力する。プログラムの実行状態を確認するために使う。タスク関数からも割り込みハンドラからも使える。debug.c で実装。

関数 [5.8.3 void halSerialWriteChar\(int SerialNo, int c\);](#) を使ってシリアル 1 (SerialNo=0) へ文字を出力する。

#### (2) マクロ CMTRACE

出力ウィンドウ (現在のコンソール) へ文字列を出力する。hal.h で実装。

関数 [5.3.2 void halDebugPrintf\(const char \\*formatstring, ...\);](#) を使っている。

#### (3) #define WRITE16550(c,d) hal16550WriteDATA((c),(d))

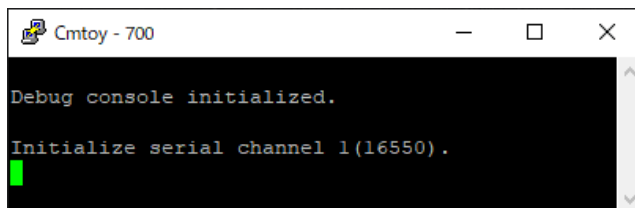
シリアル 2 (TCP/IP ポート 701) へ文字を出力するためには以下のようにマクロ WRITE16550 を使用する。

```
WRITE16550 (1, 文字コード)
```

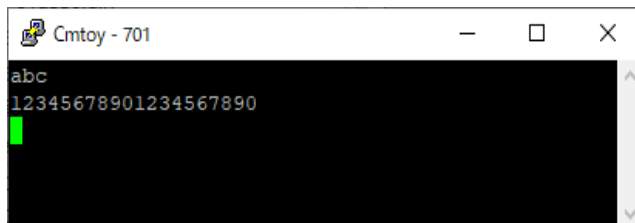
「[5.9.1 void hal16550WriteDATA\(int SerialNo, BYTE d\)](#)」を参照。

### 11.3.2 端末 PuTTY への表示

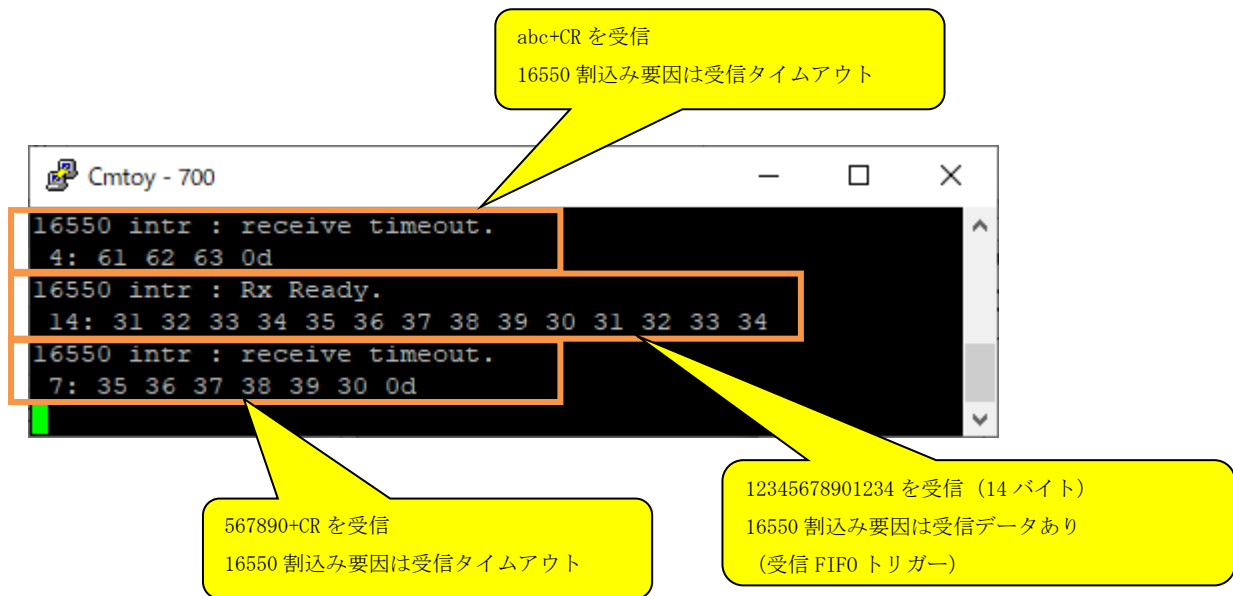
- アプリケーション app.dll を実行するとシリアル 1 の PuTTY に以下のメッセージが表示されます。DebugPrintf 関数を使ってメッセージを出力します。



- DIP スイッチ send をチェックし (abc+CR を送信)、その後チェックを外す (12345678901234567890+CR を送信) とシリアル 2 の PuTTY に文字列が以下のように表示されます。WRITE16550 を使って 16550 の送信 FIFO に設定した文字が表示されます。



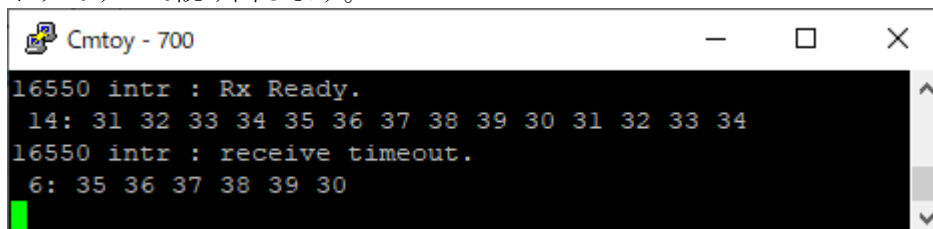
- DIP スイッチ LoopBack をチェックし、DIP スイッチ send をチェックし、その後チェックを外すと送信データは 16550 から外に出ないので Cmtoy-701 へは表示されません。その代わり受信データとして戻ってくるので、それを Cmtoy-700 へ表示します。



- レシーバ FIFO にデータを送り込むには以下のコンソールコマンドを使います。

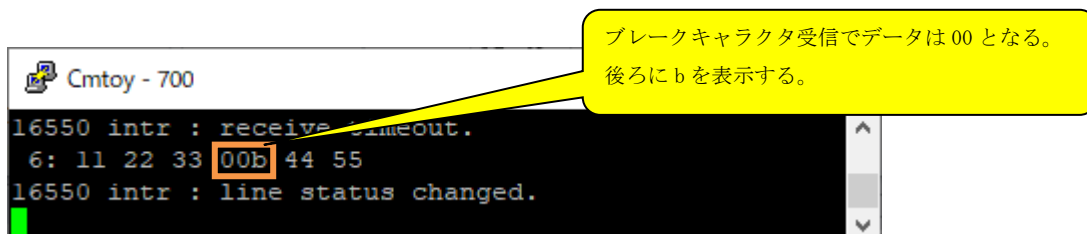
```
serial 1 push "12345678901234567890"
```

ここでは 20 文字送るので受信 FIFO に 14 文字入ると受信 FIFO の受信トリガーレベルを超えた割り込み (RxReady) が CPU に入るので割り込みハンドラ (関数 IntHdr16550) ですべて読み出します。残りの 6 文字は受信 FIFO に入った後受信データタイムアウトの割り込みが CPU に入るので割り込みハンドラですべて読み出します。



以下のコマンドでブレークキャラクタを送り込むことができます。割り込みハンドラでは以下のような表示をします。

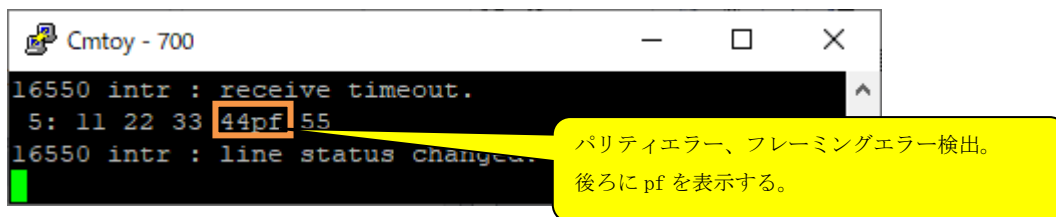
```
serial 1 push 11 22 33 sb 44 55 ;ブレークキャラクタ
```



以下のコマンドでパリティエラー、フレーミングエラーとともにデータを送り込むことができます。割り込みハンドラでは以下のような表示をします。

```
serial 1 push 11 22 33 se 44 55 ;パリティ、フレーミングエラー
```



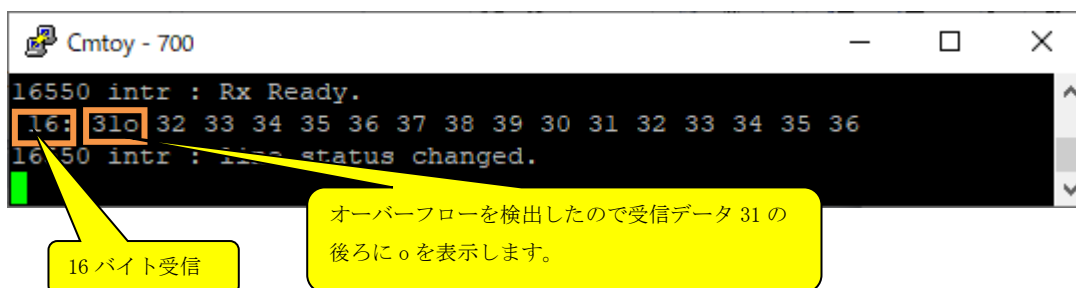


```
Cmttoy - 700
16550 intr : receive timeout.
5: 11 22 33 44pf 55
16550 intr : line status changed.
```

パリティエラー、フレーミングエラー検出。  
後ろに pf を表示する。

以下のオーバーフローエラーとともにデータを送り込むことができます。割込みハンドラでは以下のような表示をします。

```
serial 1 push "12345678901234567890" -c0 ;オーバーフロー発生
```



```
Cmttoy - 700
16550 intr : Rx Ready.
16: 31o 32 33 34 35 36 37 38 39 30 31 32 33 34 35 36
16550 intr : line status changed.
```

16 バイト受信

オーバーフローを検出したので受信データ 31 の  
後ろに o を表示します。

※受信データが受信 FIFO のトリガーレベルを超えると割込み要因 RxReady の割込みが発生する。  
※最後のデータが受信 FIFO に入ってから 4 文字分空くと受信タイムアウトの割込みが発生する。  
※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が検出された文字の場合、その文字が受信 FIFO のトップに移行した時にラインステータスレジスタへセットされる。  
※-c0 オプションで push コマンドは CPU が読むよりも早く FIFO にデータを充填するので 17 バイト目でオーバーフローエラーがラインステータスレジスタへセットされる。それ以降のデータは捨てられる。しかし FIFO の先頭を取り出すときにオーバーフローエラーが検出できる。

## 12 考察

### 12.1 Visual Studio 6.0 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」メニューの「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」メニューの「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」メニューの「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のですが、Visual Studio 6.0 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、

- ・ Visual Studio 6.0 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が。特にタイマイ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イベントが連続して発生 するのに見えます。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が。（2002 年）

### 12.2 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）

### 12.3 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。

- ① Visual 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)

## 12.4UML 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なイベント定義、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)

## 12.5Visual 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/>

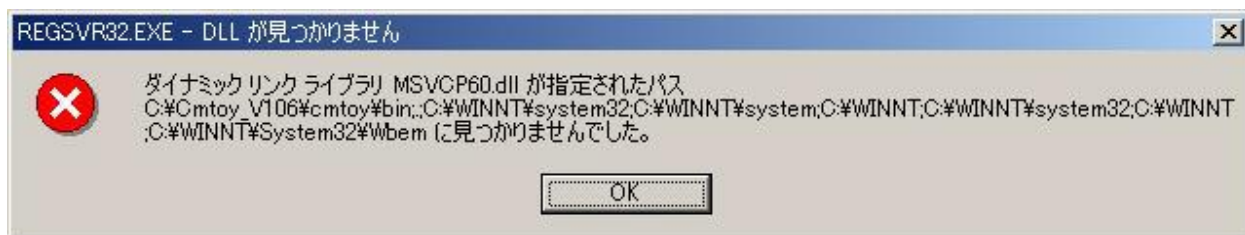
(2008 年)

## 12.6Windows 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)

## 12. Windows Vista、Windows 7 で使用する場合

### 12.7.1 ハイパーターミナル

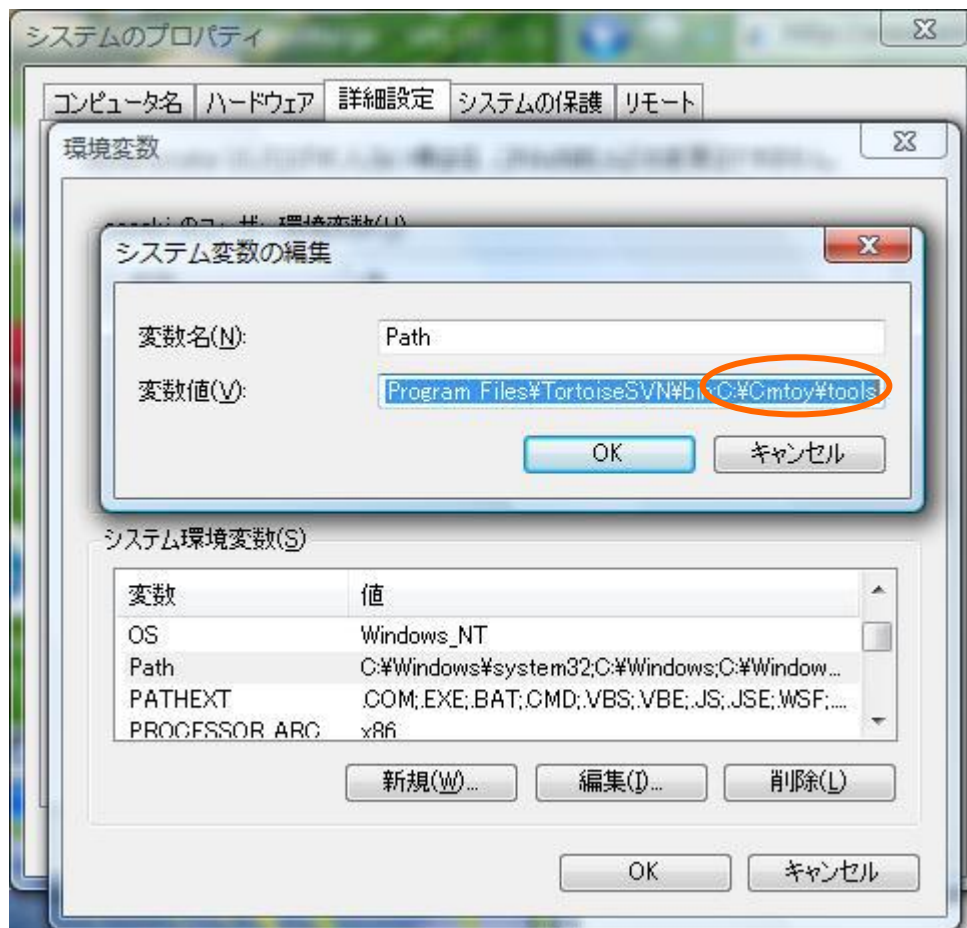
Vista および Windows 7 ではハイパーターミナルが搭載されていないので以下のページを参照してください。(日本語ページは機械翻訳のようで読みにくい)

<http://www.windowstvstaplace.com/hyperterminal-alternative-in-windows-vista/downloads/ja/>

<http://www.windowstvstaplace.com/hyperterminal-alternative-in-windows-vista/downloads/>

Windows 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)



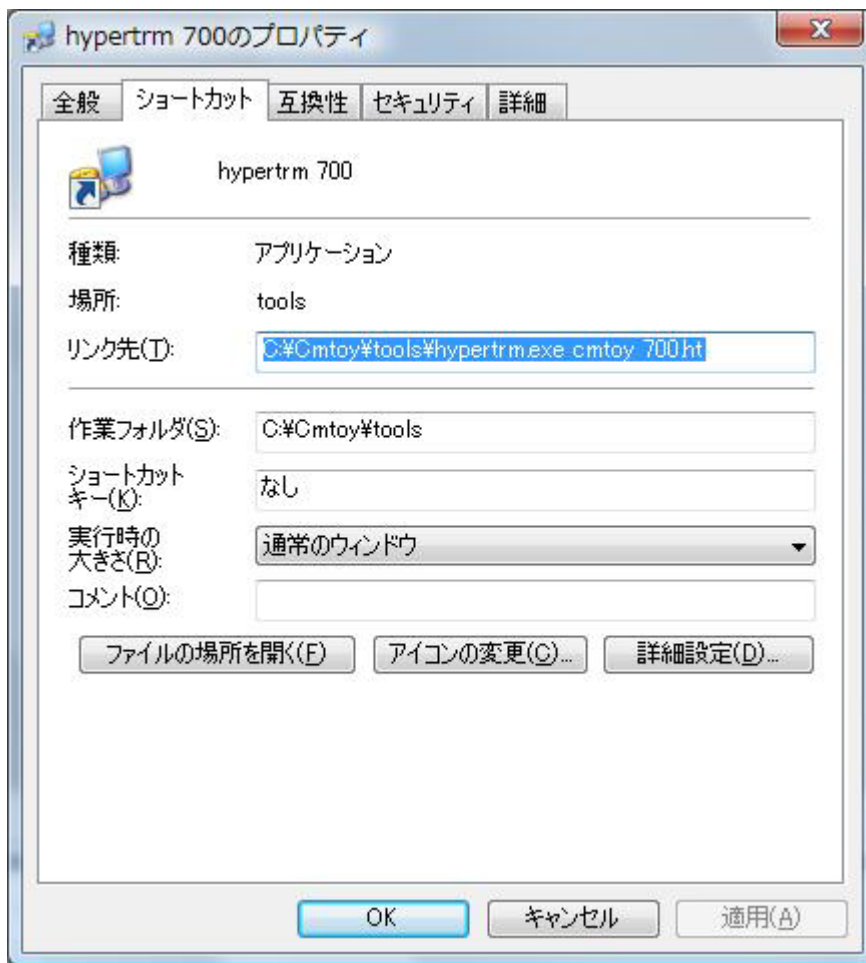
3. hypertrm.exe のショートカットを C:\¥Cmtoy¥tools フォルダにつくり、ファイル名を hypertrm700 とする。このショートカットのプロパティを開き、リンク先に

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

と入力する。同様にショートカット 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)

### 12.7.2 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、[「2.1.1 Windows Vista, Windows 7 でのインストール」](#) を参照してください。(2009 年)

## 12.8 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

## 12.9 システム初期化手順

### 12.9.1 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）。

### 12.9.2 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。

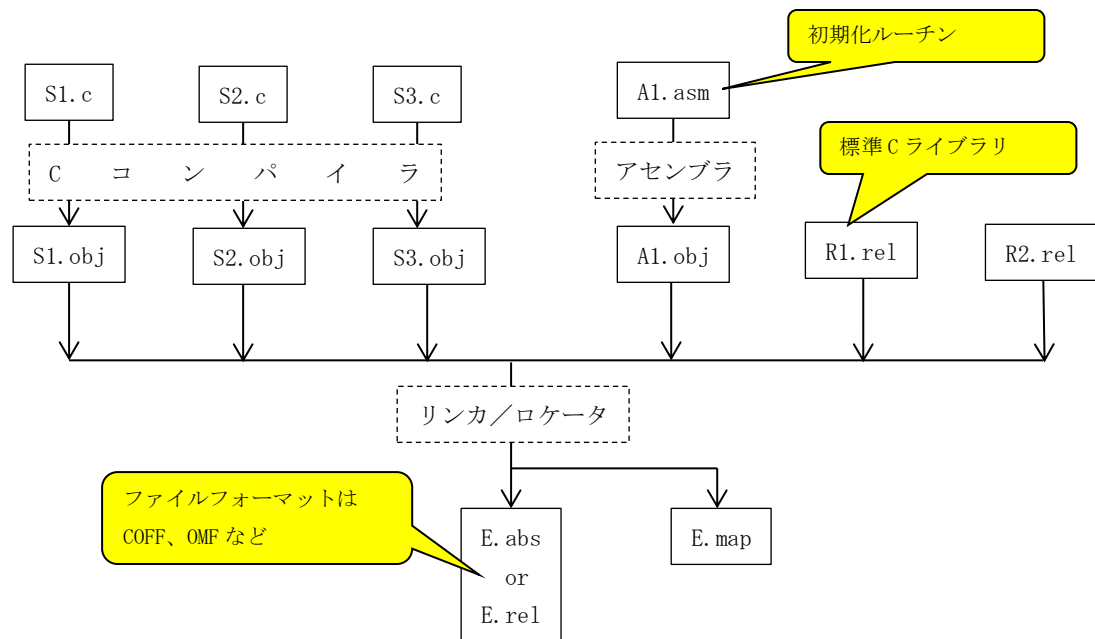
セクション.text と.rodata は書き換える必要がないので ROM 内に配置することもできますが、セクション.data と.bss はプログラムが書き換えることができる RAM 領域へ配置する必要があります。

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。このとき違う名前のセクションを1つのセクション（連続領域）にまとめ、新しい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。以下のファイル名は便宜的なものです。実際の開発では内容に即した意味のある名前となります。



- ・ 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ーが決めた名前）。
- ・ 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。

### 12.9.3 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 を解釈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 ビット論理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 (0x00000000-0xffffffff) 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) $\mu$ ITRON 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、 $\mu$ ITRON 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、 $\mu$ ITRON カーネル、 $\mu$ ITRON 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)
$\mu$ ITRON カーネルコード	RAM	実行コード、定数領域
$\mu$ ITRON 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 $\mu$ ITRON タスク用スタック領域
$\mu$ ITRON コンフィグレーションテーブル (注 1)	RAM	固定アドレス、 $\mu$ ITRON アプリケーションに含まれる。
$\mu$ ITRON 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)
$\mu$ ITRON 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)
```

$\mu$ ITRON 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  $\mu$ ITRON カーネルとの間で決めておく必要があります、 $\mu$ ITRON カーネル、 $\mu$ ITRON 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

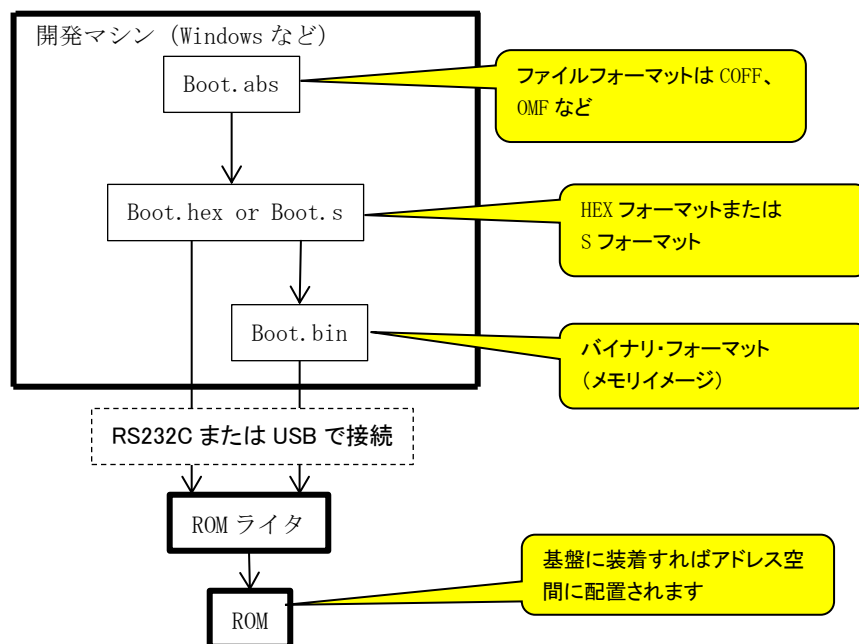
ておきます。

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  $\mu$  ITRON カーネルと  $\mu$  ITRON 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  $\mu$  ITRON カーネルと  $\mu$  ITRON 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)、 $\mu$  ITRON カーネル (kernsl.abs)、 $\mu$  ITRON 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  $\mu$  ITRON カーネルと  $\mu$  ITRON 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  $\mu$  ITRON カーネルと  $\mu$  ITRON 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

$\mu$  ITRON カーネル、 $\mu$  ITRON 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 (メモリーイメージ)  | 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

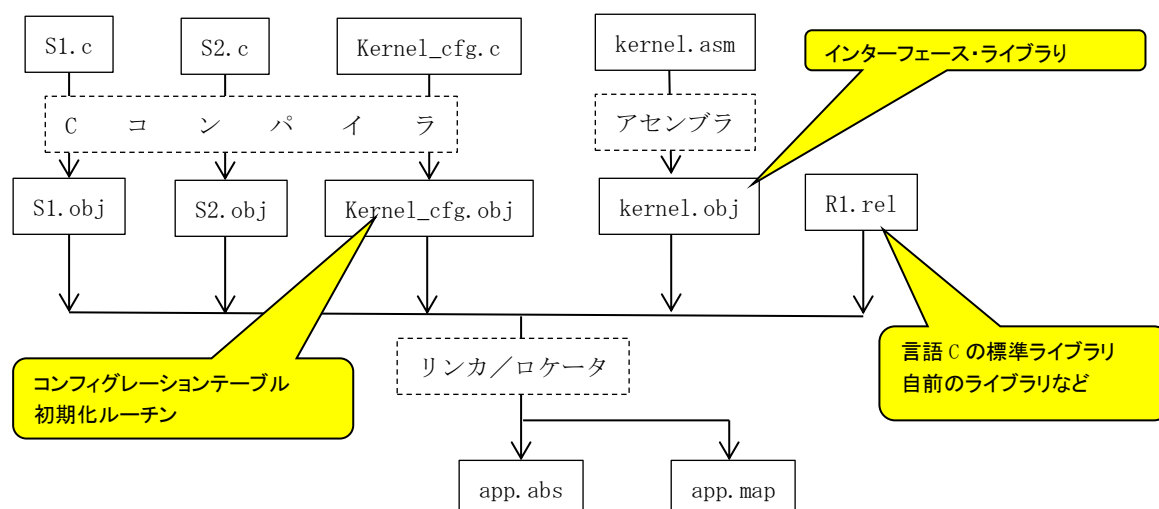
$\mu$  ITRON カーネル、 $\mu$  ITRON アプリケーションを別々にリンカ/ロケータで作成すると  $\mu$  ITRON アプリケーションから  $\mu$  ITRON カーネルのサービスコールの呼び出し方法に工夫がいります。  $\mu$  ITRON カ

ーネルに修正があった場合でも  $\mu$  ITRON 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、 $\mu$  ITRON 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を参考にしてください。

[「 \$\mu\$  ITRON」入門―“組み込み系”「リアルタイム OS」の基礎 \(I・O BOOKS\) 工学社](#)

$\mu$  ITRON 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  $\mu$  ITRON 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

## 12.10 「マクロ」について

「マクロ」という単語はコンピュータプログラミングの領域ではよく目に耳にします。マクロ (macro) は「おおきい」とか「大規模な」という意味ですがコンピュータプログラミングの現場ではいろいろな場面で遭遇する言葉です。私が経験したものを挙げて見ます。

- ・ マクロアセンブラのテキストマクロ
- ・ ASM386 のコードマクロ
- ・ C/C++言語のプリプロセッサ・マクロ
- ・ VBA (Visual Basic for Applications) とマクロ

ASM386 のレファレンスは以下から取得できます。

<https://mirror.math.princeton.edu/pub/oldlinux/Linux.old/Ref-docs/asm-ref.pdf>

### 12.10.1 マクロアセンブラのテキストマクロ

よく使われる複数のアセンブリ命令 (機械語に対応している) をひとまとめにして名前をつけて、アセンブリ・プログラムの中にその名前を使って一連の命令を埋め込むことができます。アセンブリ・プロ

グラムは簡単な処理でも行数が多くなるのでテキストマクロを使うと行数を少なくできソースプログラムの可読性がよくなるので重要な機能です。

以下は ASM386 のテキストマクロの例です。2 つのマクロ PROLOG と EPILOG を定義しています。

```
%*DEFINE (PROLOG) (  
    PUSH EBP  
    MOV EBP, ESP  
)  
%*DEFINE (EPILOG) (  
    POP EBP  
    RET 8  
)
```

## 12. 10. 2ASM386 のコードマクロ

新しくアセンブリ命令を定義する機能です。アセンブリ命令は機械語のオペコード (operation code) に対応したニーモニック (mnemonic) とオペランド (レジスタ/メモリ/即値) 指定部で構成されています。

x86 プロセッサは 8086,80285,80386,486,Pentium と命令セットを増やしながら進化してきました。80386 以降のプロセッサは 86 モード、286 モードで動作することもできます。その場合 8086、80286 で持っていなかった機能も利用できます。例えば 286 モードでもオペランド・サイズ・プリフィックス (prefix) を命令の直前に付けると同じ機械語でオペランドを 16 ビットから 32 ビットに変更できます。このように CPU の新しい機能に後からニーモニックを与えてアセンブリ命令を作成するとアセンブリプログラムの中で使えて便利です。(アセンブラを改修する必要なく新しい CPU 機能が使える。)

## 12. 10. 3C/C++言語のプリプロセッサ・マクロ

これはマクロアセンブラのテキストマクロの C 言語版といえる機能です。テキストマクロより便利な関数型のマクロが定義できます。しかしこのマクロはプリプロセッサの機能なので C/C++コンパイラの持っている型チェックの機能を使うことができません。

## 12. 10. 4VBA (Visual Basic for Applications) とマクロ

Microsoft の Office アプリケーションには VBA が組み込まれています。アプリケーションのよく使う一連の操作を VBA で記述して GUI のボタンに登録しておくとう便利です。単にボタンをクリックするだけで一連の操作を実行できます。これこそマクロ機能と呼ぶにふさわしい機能です。

VisualStudio にも VBA が組み込まれていてマクロ機能が使えます。

V3.00 2023 年 3 月 1 日

## 12. 11 VisualStudio のバージョンによる違い

Cmtoy のユーザアプリケーションの開発、デバッグにはマイクロソフト社の統合開発環境 VisualStudio (略して VS) を使います。ここでは VisualStudio の各バージョンの違いを調べたのでまとめます。

VS の名称	1	2	3	4	5	6
	C/C++ コンパイラ バージョン _MSC_VER	VS6 のワーク スペースの変 換	Windows バージョン WINVER	エディットコ ンティニュー	アンセーフ CRT ライ ブラリ関数  _CRT_SECURE_NO_W ARNINGS	POSIX 関数名  _CRT_NONSTDC_NO_W ARNINGS
VS6	1200	可	0x0500	可		
VS2003	1310	可	0x0500	可		
VS2008	1500	可	0x0500	可		警告 C4996
VS2010	1600	可	0x0500	可	警告 C4996	警告 C4996
VS2013	1800	可	0x0501	可	警告 C4996	警告 C4996
VS2015	1900	可	0x0501	不可	警告 C4996	警告 C4996
VS2017	1916	可	0x0501	不可	警告 C4996	警告 C4996
VS2019	1929	不可	0x0501	不可	警告 C4996	警告 C4996

### 1. C/C++コンパイラバージョン

マイクロソフト社が定義しているプリプロセッサマクロ \_MSC\_VER の値。

調査したときに使ったコンパイラの定義している値。

### 2. VisualStudio6.0 のプロジェクトワークスペースファイル (\*.dsw) を直接開くことができるかどうか。

開ける場合は「可」。サンプルプログラムを VS2019 以降でコンパイルしたい場合は、VS2008 や VS2017 のソリューションファイルを開いて変換することができる。

### 3. コードがサポートする Windows オペレーティングシステムの最小バージョンを指定するプリプロセッサマクロ WINVER の値。

### 4. コンパイラのスイッチ

「error D8016: コマンド ライン オプション '/ZI' と '/Gy-' は同時に指定できません。」が発生する。

デバッグ情報の形式を「プログラムデータベース (/Zi)」に変更してコンパイル可能となる。

### 5. 安全でない C ランタイムライブラリ関数

「warning C4996: 'sprintf': This function or variable may be unsafe. Consider using sprintf\_s instead. To disable deprecation, use \_CRT\_SECURE\_NO\_WARNINGS.」が発生する。

ライブラリ関数 'sprintf', 'strcpy', 'strncpy',

'\_splitpath', 'scanf', 'strtok', 'fopen', 'getenv', 'vsprintf', 'vsnprintf', 'strncat', 'strcat' など発生。

プリプロセッサマクロ \_CRT\_SECURE\_NO\_WARNINGS を定義して警告を抑制できる。

### 6. POSIX 関数名

「warning C4996: 'stricmp': The POSIX name for this item is deprecated. Instead, use the ISO C++ conformant name: \_stricmp. See online help for details.」が発生する。

プリプロセッサマクロ \_CRT\_NONSTDC\_NO\_WARNINGS を定義して警告を抑制できる。

V3.00 2023 年 3 月 1 日

## 12. 11. 1MFC の CFile クラス

MFC の CFile クラスが VS6.0 とそれ以降で変更になっている。ファイルポインタを表すデータタイプが DWORD から ULONGLONG に変更になっている。

VS6.0	VS2003 以降
<b>DWORD</b> SeekToEnd();  virtual <b>LONG</b> Seek(LONG lOff, UINT nFrom);  virtual void SetLength( <b>DWORD</b> dwNewLen); virtual <b>DWORD</b> GetLength() const;  virtual UINT Read(void* lpBuf, UINT nCount); virtual void Write(const void* lpBuf, UINT nCount);  virtual void LockRange( <b>DWORD</b> dwPos, <b>DWORD</b> dwCount); virtual void UnlockRange( <b>DWORD</b> dwPos, <b>DWORD</b> dwCount);  // backward compatible ReadHuge and WriteHuge <b>DWORD</b> ReadHuge(void* lpBuffer, <b>DWORD</b> dwCount); void WriteHuge(const void* lpBuffer, <b>DWORD</b> dwCount);	<b>ULONGLONG</b> SeekToEnd();  virtual <b>ULONGLONG</b> Seek(ULONGLONG lOff, UINT nFrom);  virtual void SetLength( <b>ULONGLONG</b> dwNewLen); virtual <b>ULONGLONG</b> GetLength() const;  virtual UINT Read(void* lpBuf, UINT nCount); virtual void Write(const void* lpBuf, UINT nCount);  virtual void LockRange( <b>ULONGLONG</b> dwPos, <b>ULONGLONG</b> dwCount); virtual void UnlockRange( <b>ULONGLONG</b> dwPos, <b>ULONGLONG</b> dwCount);

V3.02 2026 年 2 月 1 日