

組み込み型 LAN 内蔵 H8CPU ボード

ConDuLan

操作ガイド — iomacro 編

第02版C
2008年3月24日
オリエンタルールアンドデー株式会社
<http://www.ord.co.jp>

本書に出てくる製品名などは各社の商標または登録商標です.

はじめに

組み込み型 LAN 内蔵 H8CPU ボード ConDuLan は電源を供給するだけで Web ブラウザから 40 点のデジタル入力を監視できるなど手軽に組み込み Web サーバを構築することができます。さらに内蔵 ROM ファイルを変更することで Web ページのデザインを容易に変更することも可能です。これらの特徴に加えて ConDuLan に内蔵されている簡易スクリプト言語 iomacro を利用することで Web サーバをさらに充実させたり、内蔵プロセッサ H8-3069 の周辺機能を存分に活用することもできます。また、入出力ポートの一部を使用して I²C デバイスや液晶キャラクタディスプレイなどを利用できるような関数群も用意されています。スクリプト iomacro で記述したプログラムはリアルタイムマルチタスク環境で動作させることができるので ConDuLan をさまざまな目的にあわせて容易に構成しなおすことができます。

この操作ガイドは iomacro の機能を理解していただくために用意しました。

目次

1. iomacro とは.....	1
1.1 iomacro 動作確認準備.....	2
1.2 Hello, worldとiomacro のインストール.....	3
1.3 Hello, world の実行.....	4
2. iomacro 文法.....	5
2.1 メインプログラム.....	6
2.2 変数.....	7
2.3 ローカル変数とグローバル変数.....	8
2.4 変数の宣言.....	9
2.5 8ビットおよび16ビットの書き込み.....	10
2.6 8ビット16ビットの読み出し.....	11
2.7 算術演算子.....	12
2.8 浮動小数点.....	13
2.9 分岐制御.....	14
2.10 ループ.....	15
2.11 ループ内分岐.....	16
2.12 比較演算子.....	17
2.13 関数定義.....	18
2.14 関数の呼び出し.....	19
2.15 関数の戻り値.....	20
2.16 関数の引数(値).....	21
2.17 関数の引数(アドレス).....	22
2.18 関数の引数の数.....	23
2.19 配列(領域確保).....	24
2.20 配列(領域参照).....	25
2.21 多次元配列(領域確保).....	26
2.22 多次元配列(領域参照).....	27
2.23 文字配列(領域確保).....	28
2.24 文字配列(領域参照).....	29
2.25 多次元文字配列(領域確保).....	30
2.26 多次元文字配列(領域参照).....	31
2.27 文字列定数.....	32
2.28 文字列比較演算子.....	33
2.29 型変換演算子.....	34
2.30 そのほかの演算子.....	35
2.31 演算子記述上の注意.....	36
3. 組み込み関数—シリアル入出力.....	37
3.1 組み込み関数 <code>co_</code>	38
3.2 組み込み関数 <code>conl_</code>	39
3.3 組み込み関数 <code>ci_</code>	40
3.4 組み込み関数 <code>cinl_</code>	41
3.5 組み込み関数 <code>cb_</code>	42
4. 組み込み関数—プログラム制御.....	43
4.1 組み込み関数 <code>task_</code>	44

4.2	組み込み関数	task_ 初期化関数例.....	45
4.3	組み込み関数	wakeup_.....	46
4.4	組み込み関数	getpriority_.....	47
4.5	組み込み関数	setpriority_.....	48
4.6	組み込み関数	lock_.....	49
4.7	組み込み関数	unlock_.....	50
4.8	組み込み関数	mlock_.....	51
4.9	組み込み関数	munlock_.....	52
4.10	組み込み関数	gettimer_.....	53
4.11	組み込み関数	msleep_.....	54
5.	組み込み関数	データ処理.....	55
5.1	組み込み関数	memcpy_.....	56
5.2	組み込み関数	longcpy_.....	57
5.3	組み込み関数	memset_.....	58
5.4	組み込み関数	longset_.....	59
5.5	組み込み関数	strtoul_.....	60
5.6	組み込み関数	strcmp_.....	61
5.7	組み込み関数	strlen_.....	62
6.	組み込み関数	ファイル.....	63
6.1	組み込み関数	open_.....	64
6.2	組み込み関数	close_.....	65
6.3	組み込み関数	read_.....	66
6.4	組み込み関数	write_.....	67
6.5	組み込み関数	ioctl_.....	68
7.	組み込み関数	データ表示組み込み関数 print_.....	69
8.	組み込み関数	Web 関連.....	70
8.1	組み込み関数	ssi_.....	71
8.2	組み込み関数	SSI 関数.....	72
8.3	組み込み関数	cgiget_.....	73
8.4	組み込み関数	cgipost_.....	74
8.5	組み込み関数	htmlquery_.....	75
8.6	組み込み関数	htmlaccess_.....	76
8.7	組み込み関数	htmlpost_.....	77
8.8	組み込み関数	htmlget_.....	78
8.9	組み込み関数	htmlrefreshtime_.....	79
8.10	組み込み関数	Web ページ環境変数.....	80
8.11	組み込み関数	getcgi_.....	81
8.12	組み込み関数	printcgi_.....	82
8.13	組み込み関数	printcgilong_.....	83
9.	組み込み関数	プルダウンメニュー.....	84
9.1	組み込み関数	Web 上でのプルダウンメニュー書式.....	85
9.2	組み込み関数	iomacro のプルダウンメニュー情報.....	86
9.3	組み込み関数	iomacro のプルダウンメニュー情報.....	87
9.4	組み込み関数	ssipulldown_.....	88
9.5	組み込み関数	getpulldown_.....	89
9.6	組み込み関数	ssiselectpulldown_.....	90

10. 組み込み関数-TCP.....	91
10.1 TCP 関数呼び出し手順.....	92
10.2 組み込み関数 disclose_.....	93
10.3 組み込み関数 tcpsvr_.....	94
10.4 組み込み関数 tcppasswd_.....	95
10.5 組み込み関数 tcpip_.....	96
11. 組み込み関数-UDP.....	97
11.1 組み込み関数 udp_.....	98
11.2 組み込み関数 udpsend_.....	99
11.3 組み込み関数 udprecv_.....	100
11.4 組み込み関数 udpclose_.....	101
12. 組み込み関数-eメール.....	102
12.1 組み込み関数 mail_.....	103
12.2 組み込み関数 mailauth_.....	104
13. 組み込み関数-ポート入出力.....	105
13.1 ConDuLan のポート.....	106
13.2 ConDuLan のポート名称.....	107
13.3 組み込み関数 portget_.....	108
13.4 組み込み関数 portset_, portclr_.....	109
13.5 組み込み関数 portgetconf_.....	110
13.6 組み込み関数 portgetconfname_.....	111
13.7 組み込み関数 portsetconfname_.....	112
13.8 組み込み関数 portsaveconf_.....	113
14. 組み込み関数-アナログ入出力.....	114
14.1 組み込み関数 da_.....	115
14.2 組み込み関数 ad_.....	116
14.3 組み込み関数 adall_.....	117
15. 組み込み関数-I2C.....	118
15.1 組み込み関数 i2creset_, i2c2reset_.....	119
15.2 組み込み関数 i2crw_, i2c2rw_.....	120
15.3 組み込み関数 i2c_, i2c2_.....	121
15.4 組み込み関数 i2crws_, i2c2rws_.....	122
15.5 組み込み関数 i2crwd_, i2c2rwd_.....	123
16. 組み込み関数-環境変数.....	124
16.1 環境変数名.....	125
16.2 組み込み関数 getenv_.....	126
16.3 組み込み関数 setenv_.....	127
16.4 組み込み関数 unsetenv_.....	128
16.5 組み込み関数 getenvlong_, getenvip_.....	129
16.6 組み込み関数 setenvlong_, setenvip_.....	130
16.7 組み込み関数 saveenv_.....	131
16.8 組み込み関数 saveenvlong_, saveenvip_.....	132
17. 組み込み関数-パラメータ.....	133
17.1 グローバルパラメータ予約 ID.....	134
17.2 組み込み関数 gpsetl_.....	135

17.3	組み込み関数	gpgetl_	136
17.4	組み込み関数	gpsetipaddr_	137
17.5	組み込み関数	gpgetipaddr_	138
17.6	ConDuLan の日付と時刻		139
17.7	組み込み関数	gpsetdt_	140
17.8	組み込み関数	gpgetdt_	141
17.9	組み込み関数	gpclrdt_	142
18.	組み込み関数—データログ		143
18.1	ログデータ送信		144
18.2	ログ保存 ROM		145
18.3	ログデータバッファ		146
18.4	組み込み関数	logalloc_	147
18.5	組み込み関数	logbackup_	148
18.6	組み込み関数	logclear_	149
18.7	組み込み関数	logput_	150
18.8	組み込み関数	logprint_	151
18.9	組み込み関数	logbackupinfo_	152
18.10	組み込み関数	logearliest_ , loglatest_	153
18.11	組み込み関数	logselsend_	154
19.	組み込み関数—配列		155
19.1	整数配列確保関数(1次元)	array_	156
19.2	整数配列確保関数(2次元)	array_	157
19.3	整数配列確保関数(多次元)	array_	158
19.4	16ビット整数配列確保関数	sarray_	159
19.5	文字配列確保関数	carray_	160
19.6	ローカル整数配列確保関数	arraylocal_	161
19.7	ローカル 16ビット配列確保関数	sarraylocal_	162
19.8	ローカル文字配列確保関数	carraylocal_	163
20.	組み込み関数—割り込み		164
20.1	組み込み関数	int_	165
20.2	組み込み関数	ena_	166
20.3	割り込み番号		167
21.	組み込み関数 - LAN-シリアル		168
21.1	組み込み関数	telnetd_	169
21.2	組み込み関数	resettelnetd_	170
22.	その他の組み込み関数		171
22.1	組み込み関数	version_	172
22.2	組み込み関数	getname_	173
23.	液晶キャラクタディスプレイ		174
23.1	デバイスドライバ /dev/lcd0, /dev/lcd1		175
23.2	デバイスドライバ /dev/led		176
23.3	デバイスドライバ /dev/buzzer		177
24.	ハードウェアリソース		178
24.1	入出力ピン		179
24.2	プロセッサ内蔵周辺機能		180

25. SSI の例.....	181
25.1 SSI の例-1 Web ページの修整.....	182
25.2 SSI の例-2 Web ページ修整の確認.....	183
25.3 SSI の例-3 iomacro (ハードアクセス).....	184
25.4 SSI の例-4 iomacro (組み込み関数利用).....	185
25.5 SSI の例-5 iomacro (ソースの説明).....	186
25.6 SSI の例-6 表示結果.....	187
26. POST 用 CGI の例.....	188
26.1 POST CGI の例-1 ROM ファイルの用意.....	189
26.2 POST CGI の例-2 現在の表示.....	190
26.3 POST CGI の例-3 DA 表示の SSI 名変更.....	191
26.4 POST CGI の例-4 DA 表示の SSI 関数.....	192
26.5 POST CGI の例-5 DA 制御の CGI 登録.....	193
26.6 POST CGI の例-6 DA 制御の CGI ソース.....	194
26.7 POST CGI の例-7 DA 制御の iomacro.....	195
26.8 POST CGI の例-8 da.html の GET 動作.....	196
26.9 POST CGI の例-9 da.html の POST 動作.....	197
26.10 POST CGI の例-10 da.html の POST 応答.....	198
26.11 POST CGI の例-11 da.html の POST 表示.....	199
26.12 POST CGI の例-12 複数データの場合.....	200
27. GET 用 CGI の例.....	201
27.1 GET CGI の例-1 ROM ファイルの用意.....	202
27.2 GET CGI の例-2 CGI ソース (htmlget_使用).....	203
28. データのログについて.....	204
28.1 ログの例-2 トップページ変更.....	205
28.2 ログの例-3 ログデータ収集用組み込み関数.....	206
28.3 ログの例-3 ログデータ収集タスク.....	207
28.4 ログの例-4 ログデータ応答 GET 用 CGI.....	208
28.5 ログの例-5 データログシステムソース.....	209
28.6 ログの例-6 ログデータ取得画面.....	210
28.7 ログの例-7 AD0とAD1のログ.....	211
28.8 ログの例-8 AD0とAD1のログデータ取得画面.....	212
29. 排他制御について.....	213
おわりに.....	214

1. iomacro とは

ConDuLan にインストールされているアプリケーションファームウェアには簡易スクリプト言語 iomacro が含まれています。iomacro は PC で開発されたソースコードをインストールモードでセカンダリシリアル ROM に書き込みます。

iomacro には次のような特徴があり、H8 ハードウェアと ConDuLan 内蔵関数群を利用してマルチタスクシステムを構築することができます。

- ソースファイルをインストールするので特別な開発環境不要
- H8 ハードウェアに直接アクセスできる
- マルチタスクプログラムを記述できる
- CGI, SSI などを記述して内蔵 Web サーバと協調動作できる
- シリアルポートへメッセージを表示できる
- 測定データなどを記録しておいて Web ブラウザで PC に保存できる
- 浮動小数点演算ができる
- 電子メール送信ができる
- その他内蔵アプリケーションと内蔵 RTOS の機能を利用できる

この章では iomacro のインストールとシリアル通信によるデバッグ表示をご紹介します。

1.1 iomacro 動作確認準備

iomacro でさまざまなプログラムを作って動作確認するまえにクロス結線のシリアルケーブル(両端メス)とシリアルポートを持つ PC(USB-シリアル変換でも可)をご用意ください. iomacro によるプログラム動作ではかならずしもシリアル通信ありませんが, 動作確認には非常に有効です. ConDuLan の前面 D-SUB9 ピンシリアルポートと PC をクロスケーブルで接続し, ハイパーターミナル*などの通信ソフトを起動してください. 通信仕様は 9600BPS, 8ビットパリティなし, ストップ1 です.

シリアル通信環境が用意できなくても iomacro プログラム開発は可能です.

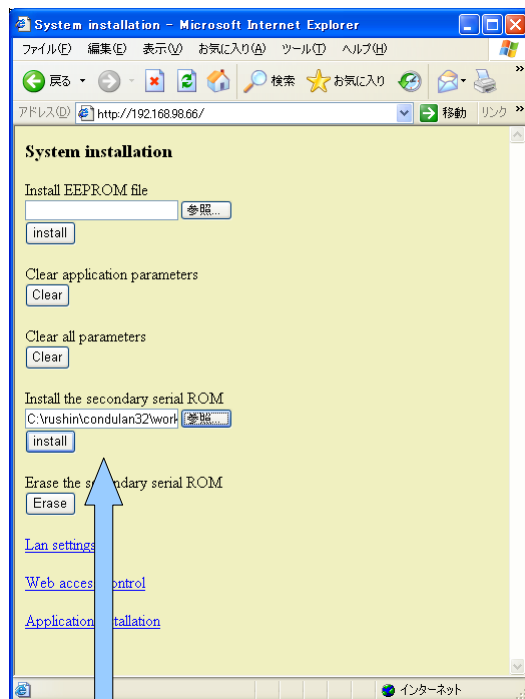
* Windows の スタート - すべてのプログラム - アクセサリ - 通信 - ハイパーターミナル

1.2 Hello, world と iomacro のインストール

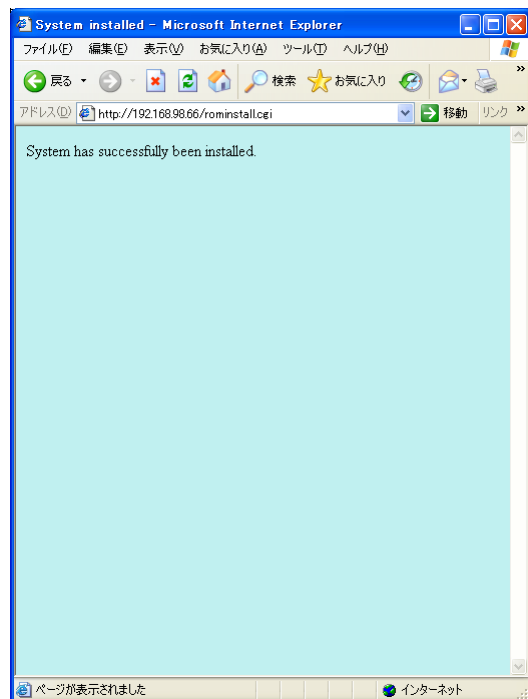
これから iomacro のいろいろな機能を試しながら解説していきますが、実行結果を知るためにも最初にシリアルポートへのメッセージ出力方法をご紹介します。

CD の iom フォルダにある iomacro 用サンプルプログラム *helloworld.iom* をインストールして実行してみましょう。

ConDuLan をインストールモードで起動*し、*Install the secondary serial ROM* の下の入力欄にファイルパスを入力します(右の参照からファイルを指定します)。その下の *install* をクリックすると iomacro 用のプログラムがインストールされます。



ファイルパスを入力して *install* をクリックすると iomacro プログラムをインストールすることができる。



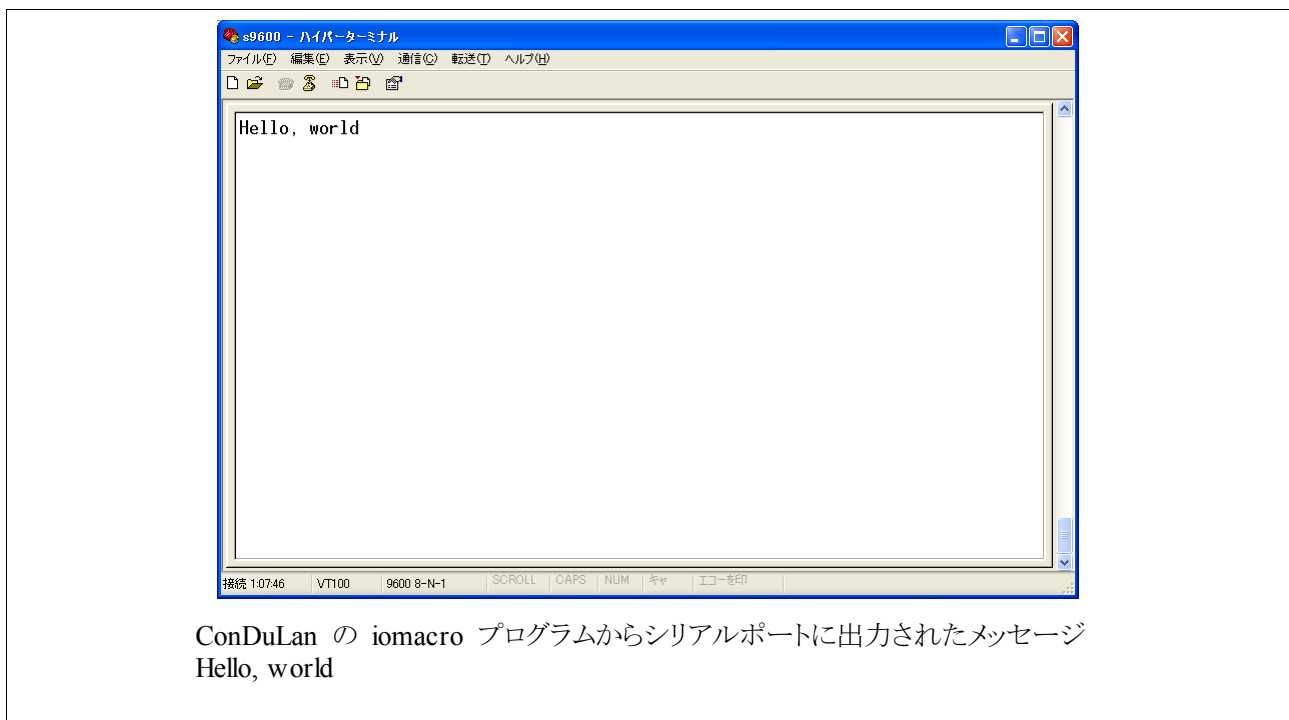
インストール終了画面

* 前面のインストールスイッチを押した状態で電源投入する「操作ガイド --- インストール編」参照

1.3 Hello, world の実行

サンプルプログラムのインストールが終わったら、PC のシリアル通信ソフトを起動してから ConDulan の電源を再投入してください。電源投入後しばらくすると PC のシリアル通信ソフトの画面に

 Hello, world
が表示されるはずです。



このように iomacro プログラム中にシリアル出力を記述しておくことで iomacro の動作状況を調べることができます。

iomacro ソース helloworld.iom の内容は次のようになっています。

```
{ conl_, "Hello, world" };  
.exit  
.end
```

2. iomacro 文法

この章では iomacro の基本的な文法をご紹介します。

2.1 メインプログラム

iomacro プログラムはファイルの先頭から実行していきま。メインプログラムはファイルの先頭に置かれます。またメインプログラムの終了は *.exit* と記述します。iomacro ソース全体の終了は *.end* と記述します。

前章で実行したプログラム *helloworld.iom* のソースプログラムは次のようになっています。

```
{ conl_ , "Hello, world" };  
.exit  
.end
```

これはメインプログラムだけで構成されています。

メインプログラムが *.exit* を実行すると iomacro プログラムは終了します。

.end の記述はソースファイルの終了を示しているだけで何かを実行するわけではありません。

第1行目の

```
{ conl_ , "Hello, world" };
```

は関数 *conl_** を呼び出します。関数 *conl_* は引き続き引数をC言語の *printf* のように扱って ConDuLan 前面のシリアルポートへメッセージを出力します。

```
{ conl_ , "%d", 123 };
```

を実行すると 123 が表示されるはずで。

行の終わりに書かれている ; はその直前までの処理を実行することを指示します。

関数については詳しく後述します。

* 由来は console out with a new line

2.2 変数

多くのプログラム言語同様 iomacro にも変数があります。すべての変数は32ビット整数として扱われます。

変数は英文字か `_` で始まり、英数字と `_` の組み合わせで記述します。

たとえば `x x10 x10_ _x10` はすべて変数です。

変数 `x` に数値 `100` を代入するには

```
x = 100;
```

と記述します。

一方変数 `x` に代入されている値を使用する場合は、`$x` のように変数の前に記号 `$` を付けます。

`x` だけの記述は `x` のアドレスを意味し、記号 `$` は C のポインタ演算子 `*` に相当します。

下の記述は `x` の値が `100`、`y` が `200` で `z` が `300` となります。

```
x=100;
y=200;
z=$x + $y;
{ conl_ , "%d, %d, %d", $x, $y, $z};
.exit
.end
```

シリアル表示は

```
100, 200, 300
```

となるはずです。

2.3 ローカル変数とグローバル変数

iomacro の変数にはローカル変数とグローバル変数の2種類があります。

ローカル変数はその関数内だけで使用され、変数名は_で始まります。他の関数で使用するローカル変数に同じ名前があっても異なる領域に配置されます。また関数の実行が終了するとローカル変数の領域は開放されるので再度その関数が呼ばれても値は保持していません。

一方グローバル変数はどの関数からも参照可能で値は常時保持されています。

`x = 100;` はグローバル変数への代入でどの関数からも参照できます。

`_x = 100;` はローカル変数への代入なのでほかの関数から参照すると異なる値になります。

メインプログラムでもローカル変数を使用できます。

2.4 変数の宣言

iomacro では変数の宣言は必要なく, いきなり変数に代入することが可能です.

```
x = 100;
```

この記述はグローバル変数 x に 100 を代入します. x の宣言は不要です.

何も代入されていないグローバル変数には0が代入されたものとして扱われます.

```
y = $z;
```

変数 z は何も代入されていないグローバル変数なので値は 0 になっています. 変数 y には 0 が代入されます.

ローカル変数の場合は値は不定です.

次の例ではシリアルに $y=0$ が表示されます.

```
y = $x;  
{ conl_, "y=%d", $y };  
.exit  
.end
```

2.5 8ビットおよび16ビットの書き込み

iomacro では扱うデータは基本的に32ビット整数となりますが、周辺機器の制御ポートサイズが8ビットあるいは16ビットの場合は書き込み操作にそれぞれ演算子=:あるいは=::を使用します。ではポートAのビット0を出力に設定してHIGHを出力してみましょう。

次のプログラムをインストールして ConDuLan の電源を再投入してください。電源投入後しばらくするとポートAのビット0(CN2の5)に5Vが出力されるはずですが。

```
0xfffe009 =: 1;
0xfffffd9 =: 1;
.exit
.end
```

このプログラムではH8のPAディレクションレジスタ(0xfffe009)に1を書き込んでポートを出力に設定してからH8のPAデータレジスタ(0xfffffd9)に1を書き込んで5Vを出力しています。

このような記述もできます。

```
dir = 0xfffe009;
data = 0xfffffd9;
$dir =: 1;
$data =: 1;
.exit
.end
```

2.6 8ビット16ビットの読み出し

ポートサイズ8ビットあるいは16ビットの読み出しにはそれぞれアドレスの前に\$:あるいは\$::を付けて記述します。記号\$はその後の数値をアドレスとして32ビットでそのアドレスから読み出すものでしたが、\$:と\$::はそれぞれ8ビットと16ビットで読み出し、32ビットの値とします。

次のプログラムはポート A の入力値をシリアルポートへ表示します。

```
0xffffe009 =: 0x00;  
data = $:0xfffffd9;  
{ conl_, "pa=%02X", $data };  
.exit  
.end
```

2.7 算術演算子

iomacro では32ビット整数を扱う次の算術演算子が用意されています。

+	加算
-	減算
*	乗算
/	除算
%	剰余

Cなどと同様の記述となります。

例

```
 $x = 10 * \$a;$ 
```

```
 $y = 10 * \$a + \$b;$ 
```

2.8 浮動小数点

iomacro で扱うデータは基本的に32ビット整数ですが、変数に格納されているデータが32ビット浮動小数点である場合、次の演算子で四則演算することができます。

.+ 加算
.- 減算
.* 乗算
./ 除算

例

変数 x と y それぞれに浮動小数点データが格納されているときその和を変数 z に格納する。

```
z = $x .+ $y;
```

また、小数点を含む10進数記述は浮動小数点データとして扱われます。

```
x = 10;
```

```
y = 10.0;
```

この例では x には整数 10 (16進数の 0x0000000A) が格納されます。変数 y には浮動小数点データの 10.0 である 0x41200000 が格納されます。

浮動小数点データのシリアル表示は

```
{ conl_ , "%f", $x };
```

のように記述します。

2.9 分岐制御

iomacro では IF-THEN-ELSE-ENDIF 型の分岐制御が用意されています。

次の例は変数 y に1が代入されます。

```
x = 1;
.if $x == 0
    .then
        y = -1;
    .else
        y = 1;
.endif
```

演算子 *.if* は次の式が真(非ゼロ)か偽(ゼロ)かを調べ、真であれば *.then* の次の処理から実行します。偽であれば *.else* の次の処理から実行します。

2.10 ループ

iomacro では WHILE-DO-ENDWHILE 型のループ制御が用意されています。

次の例は変数 y に1が3回加算されます。

```
 $x = 1;$ 
```

```
 $y = 0;$ 
```

```
.while  $\$x \leq 3$ 
```

```
    .do
```

```
         $x = \$x + 1;$ 
```

```
         $y = \$y + 1;$ 
```

```
.endwhile
```

演算子 *.while* は次の式が真(非ゼロ)か偽(ゼロ)かを調べ、真であれば *.do* の次の処理から *.endwhile* の前まで実行します。偽であれば *.endwhile* の次の処理から実行します。

2.11 ループ内分岐

iomacro の *.while* によるループの中で *.break* が現れると *.endwhile* の次から実行し、ループを終了します。

次の例は前章と同じ機能です。

```
x = 1;
y = 0;
.while 1
    .do
        .if 3 < $x .then .break .endif
        x = $x + 1;
        y = $y + 1;
    .endwhile
```

ループ中の *.continue* はそれ以後の処理をスキップして *.while* の次から処理をおこないます。

2.12 比較演算子

iomacro の分岐やループで使用される真／偽の判定のために次の演算子が用意されています。比較対象は32ビット整数です。真は1, 偽は0となります。

==	左辺と右辺が等しい場合真, 等しくない場合偽
!=	左辺と右辺が等しくない場合真, 等しい場合偽
<	左辺が右辺より小さい場合真, 左辺が右辺より大きいか等しい場合偽
>	左辺が右辺より大きい場合真, 左辺が右辺より小さいか等しい場合偽
<=	左辺が右辺より小さいか等しい場合真, 左辺が右辺より大きい場合偽
>=	左辺が右辺より大きいか等しい場合真, 左辺が右辺より小さい場合偽

このほかに32ビット浮動小数点同士の比較演算子が次のように用意されています。

.<	左辺が右辺より小さい場合真, 左辺が右辺より大きいか等しい場合偽
.>	左辺が右辺より大きい場合真, 左辺が右辺より小さいか等しい場合偽
.<=	左辺が右辺より小さいか等しい場合真, 左辺が右辺より大きい場合偽
.>=	左辺が右辺より大きいか等しい場合真, 左辺が右辺より小さい場合偽

2.13 関数定義

iomacro には関数の概念があります。関数の定義は非常に簡単で、関数の最初に

@関数名

を書きます。関数名は英文字で始まる名前であって先頭文字以外で利用できる文字は英数字と「_」です。

関数の終了は

.return

で呼び出し元に戻ります。

引数も戻り値も使用しない関数の場合、たとえばグローバル変数の x に1を加算する関数の場合は次のように記述します。

@incx

$x = \$x+1;$

.return

2.14 関数の呼び出し

関数は関数名を{ }の間に記述することで呼び出すことができます。

次の例はグローバル変数 x に1を代入して、その値をシリアルポートへ表示し($x=1$ を表示)、次に x に1を加算する関数 *incx* を呼んでからもう一度 x の値をシリアルポートへ表示する($x=2$ を表示する)プログラムです。

```
x = 1;  
{ conl_ , "x=%d", $x };  
{ incx };  
{ conl_ , "x=%d", $x };  
.exit
```

```
@incx  
    x = $x+1;  
    .return  
.end
```

2.15 関数の戻り値

iomacro の関数は32ビット整数を戻り値として呼び出し元に返します。

関数の中では「`_`」で表される特殊な予約変数があり、この変数の値が呼び出し元へ返されます。たとえば次のような関数はグローバル変数 x に 1 を加えた値を戻り値とします。

```
@incx
    _ = $x+1;
    .return
```

関数の戻り値は呼び出し元では関数呼び出しの{`から`}までを戻値で置き換えたのと同じ効果があります。

次の例では関数 `incx` の戻り値が x の値に1を加えたものなので、 $x=1$ の表示の後 $x=2$ が表示されます。

```
x = 1;
{ conl_ , "x=%d", $x };
x = { incx };
{ conl_ , "x=%d", $x };
.exit
```

```
@incx
    _ = $x+1;
    .return
.end
```

2.16 関数の引数(値)

関数へ引数を渡すことも可能です。引数はすべて32ビット整数です。アドレス(Cのポインタに相当)も32ビットですから、アドレス渡しも可能です。

引数付きの関数呼び出しは、引数を「,」で区切って

```
{ 関数名, 引数1, 引数2, 引数3 }
```

のように記述します。

たとえば2個の引数の和を返す関数 *add* は

```
{ add, 1, 2 }
```

と記述すると1+2の結果の3を返します。

一方引数を使う関数は引数として特殊な変数 *_0* や *_1* を使います。*_*で始まって次に数値がある変数は関数内の引数として扱われます。引数の順に *_0*, *_1*, *_2*, という名前になります。

次の例は関数 *add* が1+2を計算し3を返します。シリアル表示は *1+2=3* となります。

引数は数値(値)ですので関数内は引数に\$を付けません。

```
{ conl _, "1+2=%d", {add, 1, 2} };
```

```
.exit
```

```
@add
```

```
_ = _0 + _1;
```

```
.return
```

```
.end
```

2.17 関数の引数(アドレス)

変数に値を代入してからその変数のアドレスを関数の引数とするような例です。
引数は変数アドレスなので、関数内ではその値を使うために $\$_0$ と $\$_1$ を使用します。
 $x+y=3$ が表示されます。

```
x=1;
y=2;
{ conl_ , "x+y=%d", {add, x, y} };
.exit
```

```
@add
    _ = $ _0 + $ _1;
    .return
.end
```

引数を値とするような関数では次のように記述します。

```
x=1;
y=2;
{ conl_ , "x+y=%d", {add, $x, $y} };
.exit
```

```
@add
    _ = _0 + _1;
    .return
.end
```

2.18 関数の引数の数

iomacro 関数が呼ばれたとき使用された引数の個数を知ることができます。特殊なローカル変数 `__` (アンダースコア2個) は引数の個数を保持します。

例

呼び出しは関数名と引数 3 個なので、呼ばれた関数の `__` は 3 になり、シリアルに `argc=3` が表示される。

```
{ func, 100, 200, 300 };
```

```
.exit
```

```
@func
```

```
{ conl_, "argc=%ld", __ };
```

```
.return
```

```
.end
```

2.19 配列(領域確保)

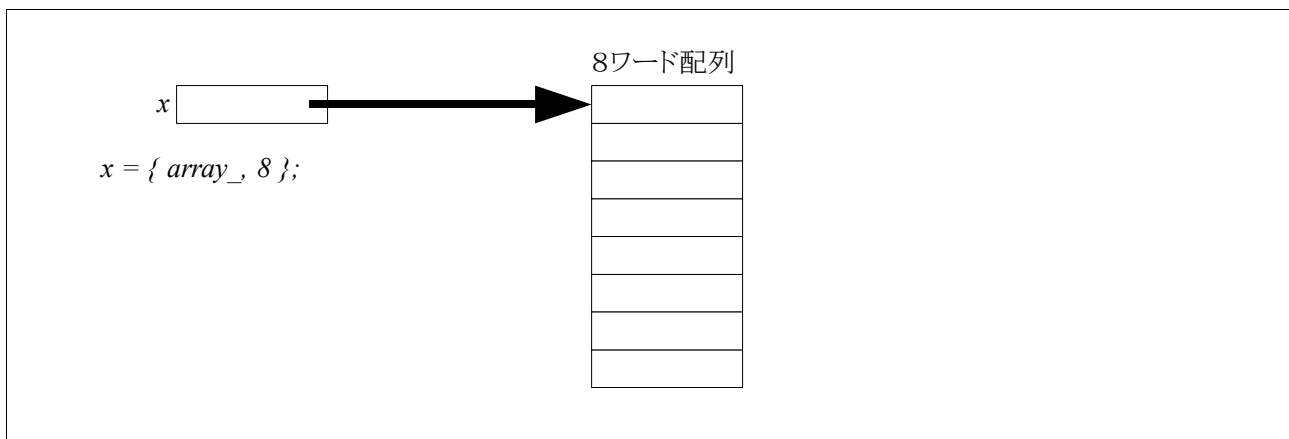
iomacro では変数の配列機能も用意してあります。配列は32ビット整数の連続した領域のことです。

配列は通常の変数とは異なり、あらかじめ領域を確保する必要があります。iomacro にはそのための関数 `array_` が組み込まれています。

次の例は8ワードの32ビット整数領域を用意し、その先頭アドレスを変数 `x` に格納します。

配列はグローバル領域に配置されます。

```
x = { array_, 8 };
```



2.20 配列(領域参照)

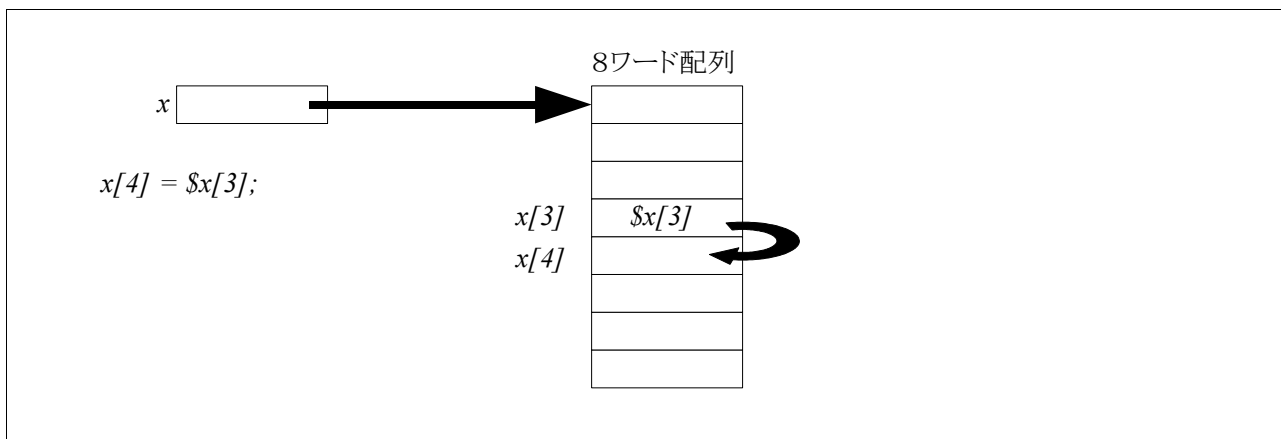
配列の各要素を参照するには $x[7]$ のように $[と]$ を使用します。

$[]$ で指定されるインデックスは配列の先頭が 0 です。

$x[7]$ は配列要素 7 のアドレスを示しますので、配列に代入されている値は $\$x[7]$ と記述します。

次の例は配列のアドレスが x に格納されているとして、配列要素 3 に格納されている値を配列 4 にも格納する記述です。

$x[4] = \$x[3];$



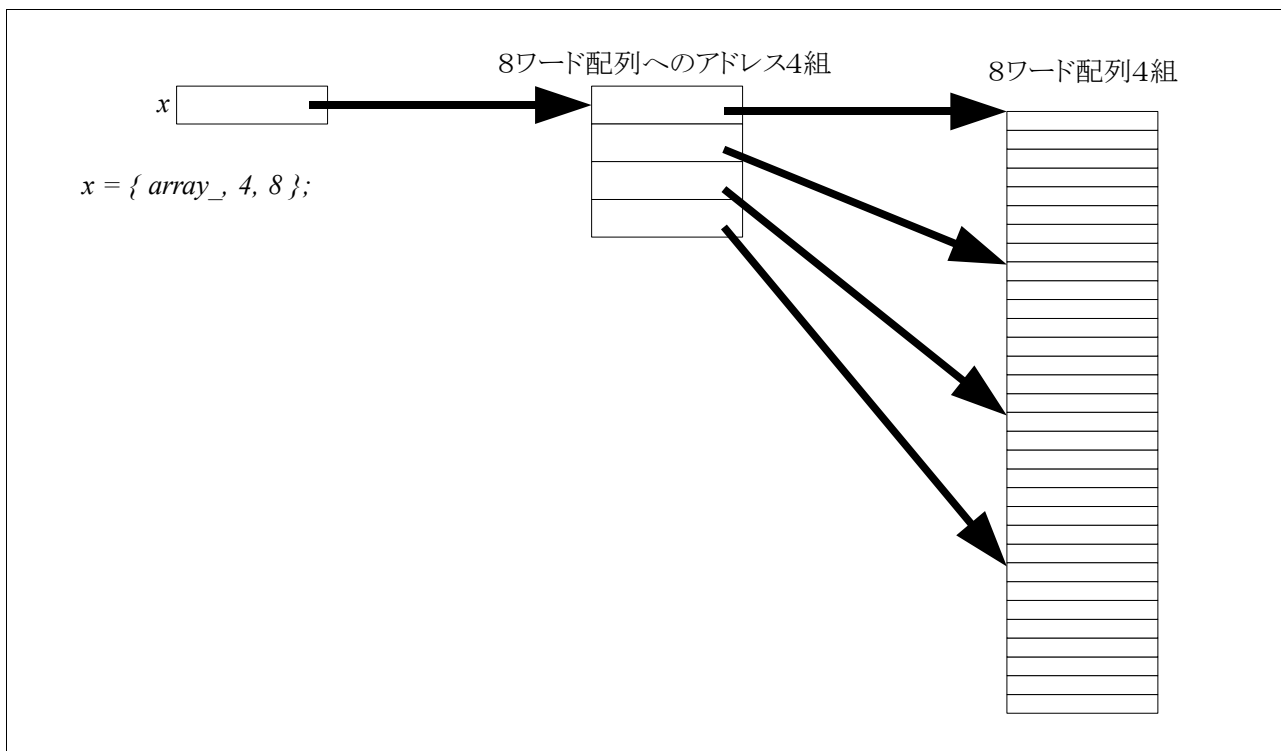
2.21 多次元配列(領域確保)

次の例は8ワードの32ビット整数の配列を4組領域確保し、その先頭アドレスを変数 x に格納します。

配列はグローバル領域に配置されます。

iomacro では多次元配列の場合途中にアドレス配列が追加されます。

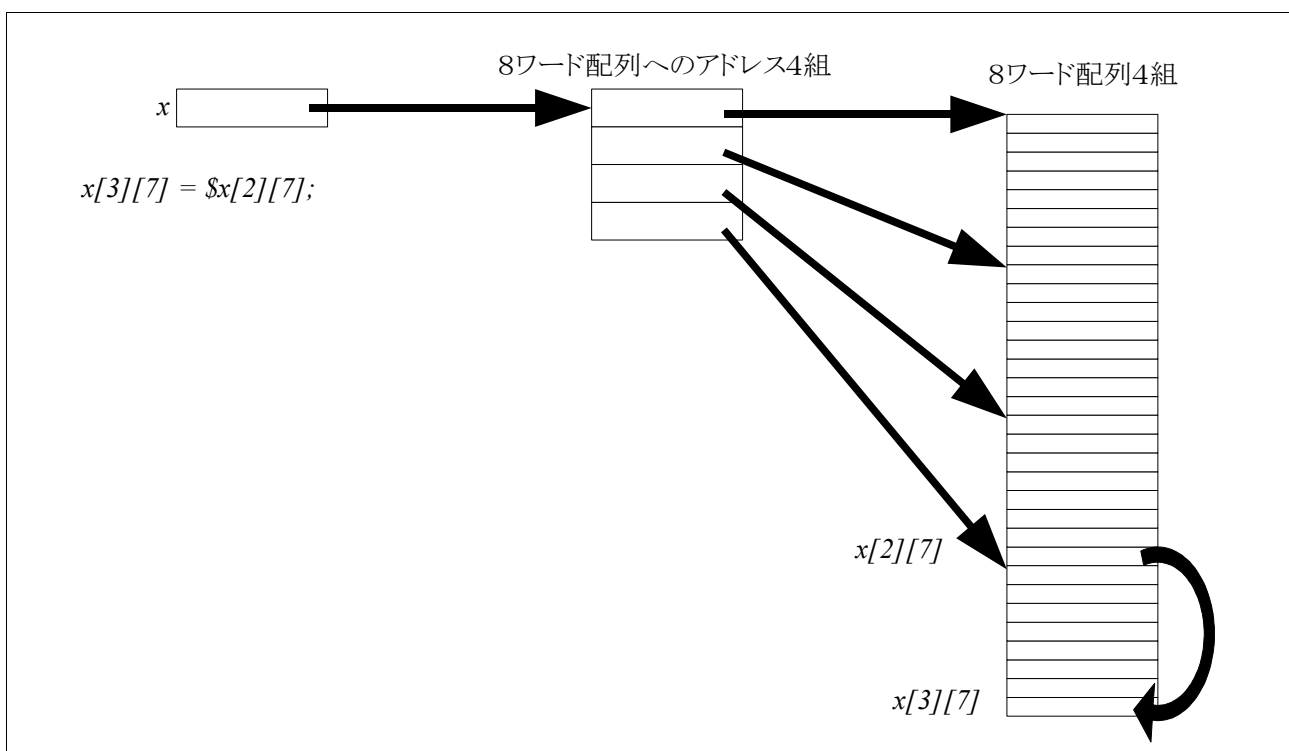
```
 $x = \{ array\_ , 4, 8 \};$ 
```



2.22 多次元配列(領域参照)

多次元配列要素の参照は $x[2][7]$ のように配列インデックスを並べます。

次の例は 4×8 の配列のアドレスが x に格納されているとして、配列要素 $[2][7]$ に格納されている値を配列 $[3][7]$ に格納する記述です。

$$x[3][7] = \$x[2][7];$$


2.23 文字配列(領域確保)

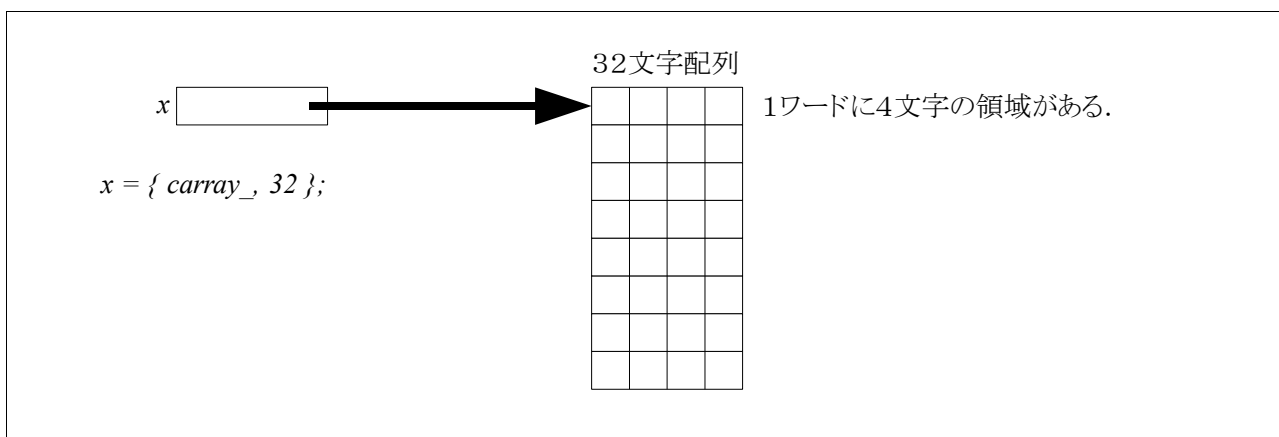
iomacro の配列要素は32ビット整数ですが、文字列を扱う場合には文字が8ビットなので使いません. iomacro には8ビットデータの文字配列を扱う機能も用意されています.

文字配列の確保は組み込み関数 *carray_* を使います.

次の例は32文字(32バイト)の領域を用意し、その先頭アドレスを変数 *x* に格納します. 配列はグローバル領域に配置されます.

```
x = { carray_, 32 };
```

この処理は `x = { array_, 8 };` と同機能になります.



2.24 文字配列(領域参照)

文字配列の各要素を参照するには $x[:7]$ のように $[:と]$ を使用します。

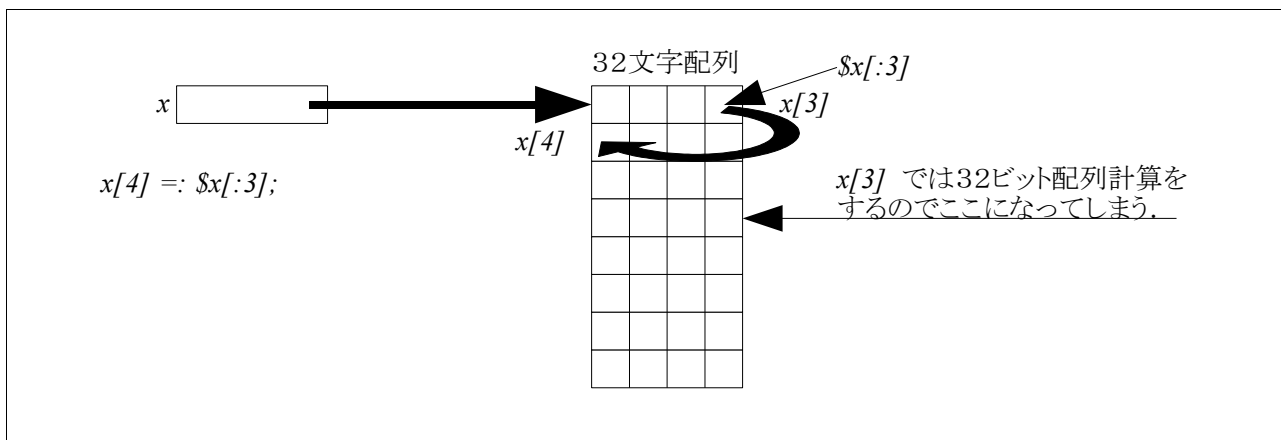
$[:]$ で指定されるインデックスは配列の先頭文字が 0 です。

$x[:7]$ は配列文字 7 のアドレスを示しますので、配列に代入されている文字は $\$x[:7]$ と記述します。

次の例は文字配列のアドレスが x に格納されているとして、要素 3 に格納されている文字を要素 4 にも格納する記述です。

```
x[4] =: $x[:3];
```

代入する先も文字配列の要素なので、 $=:$ を使って代入します。通常の変数に代入する場合は $c = \$x[:3];$ のように $=$ を使います。これは c が 32 ビットの整数だからです。

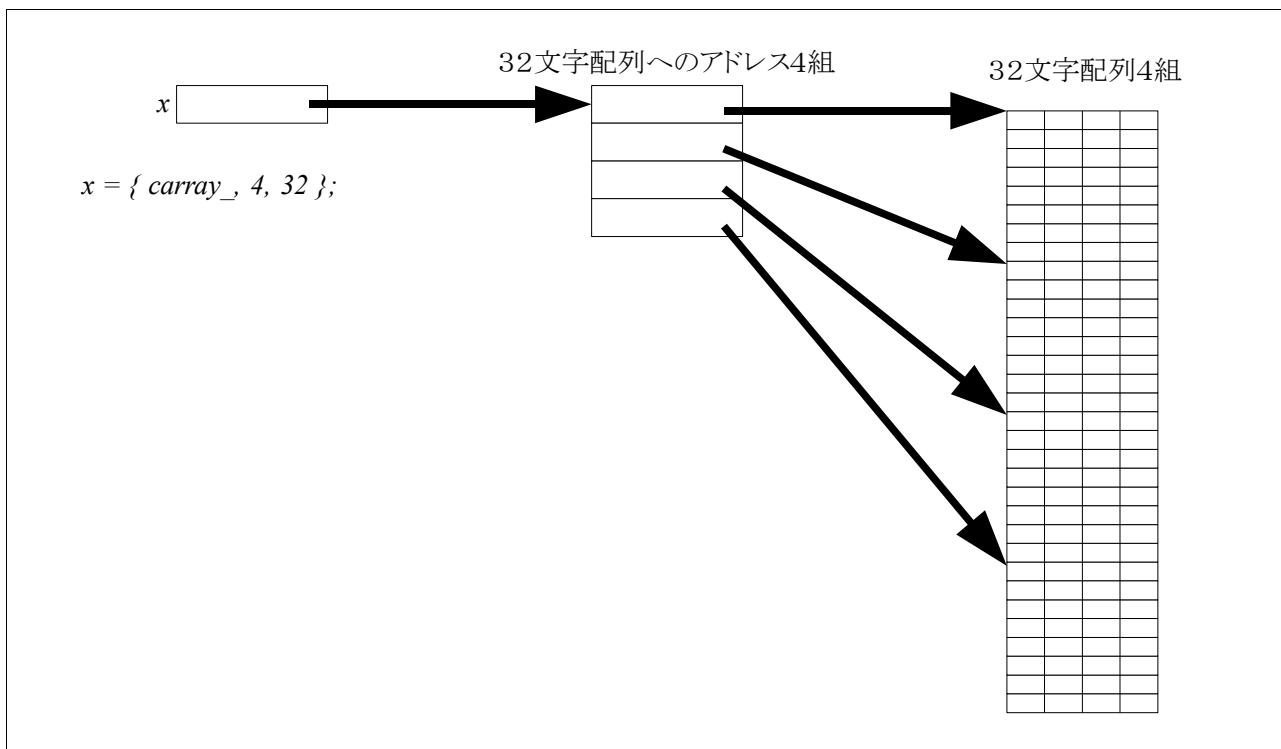


2.25 多次元文字配列(領域確保)

複数の文字列を配列にする多次元文字配列も扱えます. たとえば32文字の文字列4組を配列にした場合の領域確保は

```
x = { carray_, 4, 32 };
```

のように記述します. これは $x = \{ array, 4, 8 \};$ と同等です.

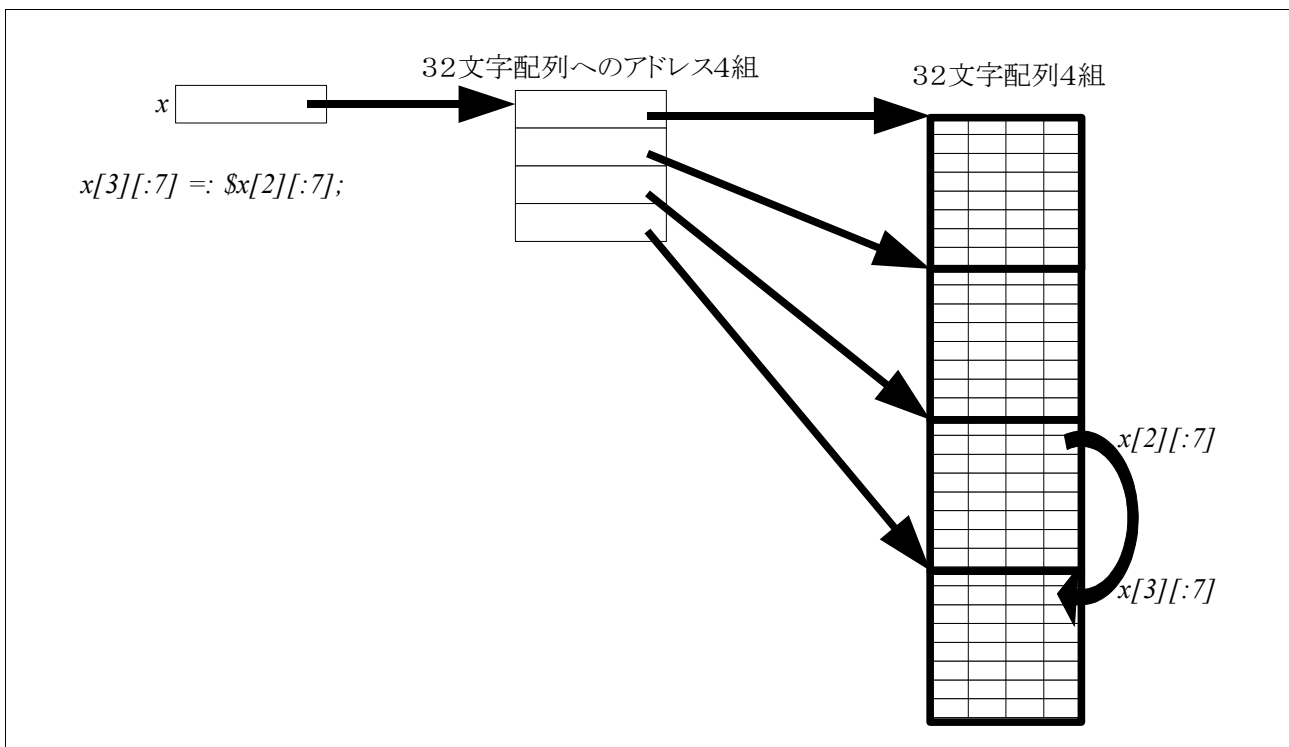


2.26 多次元文字配列(領域参照)

多次元文字配列要素の参照は $x[2][:7]$ のように配列インデックスを並べます。最後のインデックスにだけ $:$ を使用します。

次の例は 4×32 の文字配列のアドレスが x に格納されているとして、要素 2 の 7 に格納されている文字を配列 3 の 7 に格納する記述です。

```
x[3][:7] =: $x[2][:7];
```

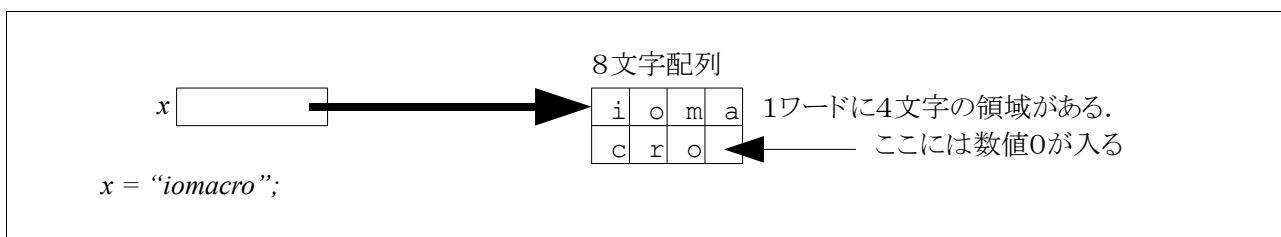


2.27 文字列定数

iomacro では文字列定数を表すのに ”を使用します. たとえば次の記述は iomacro という文字列7文字に加えて最後に8ビットの値0の文字を付加した8文字を定数領域に用意しその先頭アドレスを変数 x に格納します.

$x = \text{"iomacro"};$

変数 x は文字配列へのアドレスを格納しているので $x[0]$ のような記述で文字列中の個々の文字を参照することができます.



2.28 文字列比較演算子

iomacro には二つの文字配列に入っている文字をどちらかに 0 (NULL のこと) が現れるまで順に比較する演算子があります。文字列の内容が完全に一致しているかどうかを調べたりするのに利用できます。Cライブラリの `strcmp` に近い機能です。

演算子は次の6種があります。説明は真となる条件です。

- .<. 左辺の文字列に先に小さいコードが見つかった。
- .>. 左辺の文字列に先に大きいコードが見つかった。
- !.=. 左辺と右辺の文字列は同じものではない。
- .<=. 左辺の文字列に先に小さいコードが見つかったか、同じ文字列である。
- ==. 左辺と右辺は同じ文字列である。
- .>=. 左辺の文字列に先に大きいコードが見つかったか、同じ文字列である。

次の例は `x` の持つ文字配列は "iomacro" なので `x is iomacro` と表示されます。

```
x="iomacro";  
.if $x ==. "iomacro"  
    .then { conl_ "x is iomacro" };  
    .else { conl_ "x is not iomacro" };  
.endif  
.exit  
.end
```

2.29 型変換演算子

iomacro で扱うデータは原則として32ビット整数ですが、浮動小数点データや文字列などのデータを別の型に変換する演算子が用意されています。データの直前に置いて使用します。

<i>.f.</i>	32ビット浮動小数点データを32ビット整数に変換する。
<i>.F.</i>	32ビット整数を32ビット浮動小数点データに変換する。
<i>.sl.</i>	数値文字列を32ビット整数に変換する(Cの <code>strtol</code> に相当)。
<i>.sx.</i>	16進 数値文字列を32ビット整数に変換する。
<i>.sd.</i>	10進 数値文字列を32ビット整数に変換する。
<i>.sf.</i>	浮動小数点文字列を32ビット整数に変換する。
<i>.slen.</i>	文字列の長さを取得する。
<i>.bcd.</i>	32ビット BCD データを32ビット整数に変換する。
<i>.BCD.</i>	32ビット整数を32ビット BCD データに変換する。
<i>.hms.</i>	hh:mm:ss の時刻文字列を BCD データに変換する。
<i>.ymd.</i>	yyyy:mm:dd の日付文字列を BCD データに変換する。
<i>.mdy.</i>	mm:dd:yyyy の日付文字列を BCD データに変換する。
<i>.dmy.</i>	dd:mm:yyyy の日付文字列を BCD データに変換する。
<i>.ip.</i>	IP アドレス文字列を32ビット整数に変換する。
<i>.=ip.</i>	32ビット整数を IP アドレス文字列に変換する。
<i>.=hms.</i>	BCD データを hh:mm:ss の時刻文字列に変換する。
<i>.=ymd.</i>	BCD データを yyyy:mm:dd の日付文字列に変換する。
<i>.=mdy.</i>	BCD データを mm:dd:yyyy の日付文字列に変換する。
<i>.=dmy.</i>	BCD データを dd:mm:yyyy の日付文字列に変換する。
<i>.sec.</i>	HMS の BCD データを32ビット整数の秒数に変換する。
<i>.SEC.</i>	32ビット整数の秒数を HMS の BCD データに変換する。

次の例は x に格納されている整数 10 を演算子 *.F.* で浮動少数データに変換し、 y に格納されている 1.1 と演算子 *.+* で加算した結果を表示します。 $x + y = 11.1000$ が表示されます。

```
x = 10;
y = 1.1;
{ conl_ "x + y = %f" , .F.$x .+ $y};
.exit
.end
```

2.30 そのほかの演算子

iomacro には以上のほかに次のような演算子があります.

!	ブール否定
&	論理積
(演算優先順位指定
)	(の終了
^	排他的論理和
	論理和
~	論理否定
&&	ブール積
	ブール和
>>	32ビット整数右シフト
<<	32ビット整数左シフト

また, コメントは

```
//から行末まで
```

と

```
/* から */まで
```

の2種類があります.

2.31 演算子記述上の注意

「.」で始まり, 終わりが「.」ではない演算子を記述する場合は演算子の後にはスペースを置いてください. たとえば $\$x.<=.F.\y という記述の $.<=$ が $.<=.$ と解釈されてしまいます.

$\$x.<= .F.\y

が正しい記述です.

3. 組み込み関数ーシリアル入出力

ConDuLan には D-SUB9 ピンのシリアルポートが用意されています。この章では `iomacro` からシリアルポートを利用するための組み込み関数を紹介します。

関数は次の5本です。

<code>co_</code>	文字列をシリアル出力する
<code>conl_</code>	文字列を改行つきでシリアル出力する
<code>ci_</code>	指定文字数シリアル入力して配列に格納する
<code>cinl_</code>	改行コードまでシリアル入力して配列に格納する
<code>cb_</code>	シリアルボーレートを設定する

3.1 組み込み関数 `co_`

```
{ co_ 書式 ... };
```

ConDuLan 正面の D-SUB9 ピンシリアルヘメッセージを表示します。書式とそれに続く ... はCの `printf` に準拠した手順です。値はすべて32ビットです。
関数の戻り値は出力した文字数です(改行含む)。

例

シリアルに Hello 2007 と表示して改行します。

```
{ co_ "Hello %d\r\n", 2007 };
```

3.2 組み込み関数 `conl_`

```
{ conl_, 書式, ... };
```

ConDuLan 正面の D-SUB9 ピンシリアルへメッセージを表示し改行します。書式とそれに続く ... はCの `printf` に準拠した手順です。値はすべて32ビットです。

改行はCR+LF の2文字です。

関数の戻り値は出力した文字数です(改行含む)。

例

シリアルに Hello 2007 と表示して改行します。

```
{ conl_, "Hello %ld", 2007 };
```

3.3 組み込み関数 `ci_`

`{ ci_, バッファアドレス, バッファサイズ };`

ConDuLan 正面の D-SUB9 ピンシリアルへメッセージから入力しバッファへ格納します。指定サイズ分入力したら関数を終了します。

戻り値は入力文字数です。

例

32バイト文字バッファを確保してシリアル入力をそのバッファに格納します。

```
p = { carray_, 32 };
```

```
{ ci_, $p, 32 };
```

3.4 組み込み関数 cinl_

```
{ cinl_ , バッファアドレス, バッファサイズ };
```

ConDuLan 正面の D-SUB9 ピンシリアルから入力しバッファへ格納します。改行コードが入力されるか指定サイズ分入力したら関数を終了します。

戻り値は入力文字数です。

例

32バイト文字バッファを確保してシリアル入力をそのバッファに格納します。入力文字数は n に保存されます。

```
 $p = \{ \text{carray\_}, 32 \};$ 
```

```
 $n = \{ \text{ci\_}, \$p, 32 \};$ 
```

3.5 組み込み関数 `cb_`

```
{ cb_, ボーレート };
```

ConDuLan 正面の D-SUB9 ピンシリアルのボーレートを設定します。デフォルトボーレートは 9600BPS です。選択できるボーレートは

50, 75, 110, 134, 150, 200, 300, 600, 1200, 4800, 9600, 19200, 38400, 57600 です。

例

シリアルを 38400BPS に設定します。

```
{ cb, 38400 };
```

4. 組み込み関数—プログラム制御

この章は ConDuLan に内蔵されているマルチタスク機能を利用するための組み込み関数を紹介します。

次のような関数が用意されています。

<i>task_</i>	関数をタスクとして実行する
<i>wakeup_</i>	タスクを起床する
<i>getpriority_</i>	タスクプライオリティを取得する
<i>setpriority_</i>	タスクプライオリティを設定する
<i>lock_</i>	タスクスイッチを禁止する
<i>unlock_</i>	タスクスイッチ禁止を解除する
<i>mlock_</i>	排他制御をロックする
<i>munlock_</i>	排他制御ロックを解除する
<i>gettimer_</i>	電源投入から現在までの時間(ms)を取得する
<i>msleep_</i>	指定時間(ms)タスク休止する

4.1 組み込み関数 `task_`

```
{ task_, タスク関数名 };  
{ task_, タスク関数名, タスク初期化関数名 };
```

タスク関数名で指定した関数をタスクとして動作させます。タスク初期化関数はタスク起動後1度だけそのタスクとして動作します。

関数の戻り値はそのタスクのタスク番号です。タスクプライオリティはタスク起動をかけたタスクと同じになります。タスクのスタックサイズは固定です。

グローバル変数はメインや他の関数と共有しています。

一方タスクとして動作する関数はリターンすると休止状態になります。タスクとして動作した関数の戻り値が0の場合は他のタスクか割り込み処理から起床されるまで休止状態となります。戻り値が0でなかった場合はその戻り値の時間(ms)だけ休止して、その後同じタスクとして呼び出されます。

次の例は関数 `taskA` をタスクとして実行します。

```
taskid={ task_, taskA };  
{ conl_, "created taskA as %d", $taskid };  
.exit
```

```
@taskA  
    { conl_, "taskA" };  
    .return  
.end
```

4.2 組み込み関数 `task_` 初期化関数例

関数 `task_` で初期化関数付きのタスクを起動する例です。

最初に関数 `initA` がタスクとして動作し、その後関数 `taskA` がそのタスクとして動作します。関数 `taskA` は戻り値を `1000` にしていますので、`1000` ミリ秒毎すなわち1秒毎にタスクとして呼び出されます。タスクは変数 `x` の値を1加算しながら表示していきます。

```
{ task_, taskA, initA };  
.exit  
  
@initA  
    x = 100;  
    .return  
  
@taskA  
    x = $x + 1;  
    { conl_, "taskA %d", $x };  
    _ = 1000;  
    .return  
  
.end
```

4.3 組み込み関数 `wakeup_`

```
{ wakeup_, タスク番号 };
```

タスク番号で指定したタスクを起床します。タスク番号は関数 `task_` で関数をタスク起動したときの戻り値です。

4.4 組み込み関数 `getpriority_`

```
{ getpriority_ };
```

自タスクの優先順位を取得します。戻り値が優先順位です。優先順位は正の整数で 0 が最高位です。iomacro メインは 251 ですが、SSI や CGI では Web サーバタスクの優先順位 15 となります。

4.5 組み込み関数 setpriority_

{ *setpriority_*, 優先順位 };

自タスクの優先順位を変更します。

4.6 組み込み関数 `lock_`

```
{lock_};
```

この関数を呼ぶとタスクスイッチを禁止します。マルチタスク環境下の排他制御に使用します。タスクスイッチ禁止の解除は組み込み関数 `unlock_` を使います。

`lock_ - unlock_` はネストします。2回続けて `lock_` を呼んだ場合はタスクスイッチ解除するのに `unlock_` を2回呼ぶ必要があります。

`iomacro` は実行が遅いのでタスクスイッチ期間は可能な限り短くしてください。

4.7 組み込み関数 `unlock_`

```
{unlock_};
```

この関数を呼ぶとタスクスイッチ禁止を解除します。組み込み関数 `lock_` と組み合わせて使用します。

4.8 組み込み関数 `mlock_`

```
{ mlock_, 排他制御変数 };
```

この関数はタスク間の排他制御をおこなうために用意しました。

排他制御変数を指定してこの関数を呼ぶと、すでに他のタスクがその排他制御変数を占有している場合は非0の値が戻ります。この値は現在占有しているタスクのidです。

どのタスクも指定の排他制御変数を占有していない場合は、自タスクの占有状態にします。この場合は戻り値は0です。

占有状態の解除は関数 `munlock_` を使います。

4.9 組み込み関数 `munlock_`

```
{ munlock_, 排他制御変数 };
```

この関数は `mlock_` で占有した排他制御変数を開放します。

4.10 組み込み関数 `gettimer_`

```
{ gettimer_ };
```

この関数はシステムのタイマー値を取得します。システムのタイマーは32ビットサイズで電源投入後 1ms 毎に1加算されています。

4.11 組み込み関数 `msleep_`

```
{ msleep_, 休止時間 };
```

この関数を呼んだタスクはミリ秒で指定する時間の間休止します。

例

次の記述はいずれも1秒間休止します。

```
{ msleep_, 1000 };
```

```
t = 1000;
```

```
{ msleep_, $t };
```

5. 組み込み関数ーデータ処理

この章では `iomacro` に用意されている組み込み関数のうち、データを処理する関数を紹介します。

データ処理関数には次の7本の関数があります。

<code>memcpy_</code>	指定バイト分のメモリ内容を別の領域にコピーする
<code>longcpy_</code>	指定サイズの32ビットメモリ内容を別の領域にコピーする
<code>memset_</code>	指定バイト分のメモリに固定の値を書き込む
<code>longset_</code>	指定サイズの32ビットメモリに固定の値を書き込む
<code>strtoul_</code>	文字列を32ビット整数へ変換する
<code>strcmp_</code>	文字列を比較する
<code>strlen_</code>	文字列の長さを取得する

5.1 組み込み関数 memcpy_

{ memcpy_, コピー先アドレス, コピー元アドレス, コピーバイト数 };

メモリ上の指定バイト数のデータを別のメモリ領域へコピーします。
戻り値は コピー先アドレスとなります。

5.2 組み込み関数 `longcpy_`

`{ longcpy_, コピー先アドレス, コピー元アドレス, コピーワード数 };`

メモリ上の指定ワード数のデータを別のメモリ領域へコピーします。ワードは32ビットです。戻り値は コピー先アドレスとなります。

5.3 組み込み関数 `memset_`

`{memset_ , アドレス, データ, バイト数};`

メモリ上の指定アドレスから指定バイト数へデータを書き込みます。
戻り値は 指定されたアドレスとなります。

5.4 組み込み関数 `longset_`

`{longset_, アドレス, データ, ワード数};`

メモリ上の指定アドレスから指定ワード数へデータを書き込みます。ワードは32ビットです。戻り値は 指定されたアドレスとなります。

5.5 組み込み関数 strtoul_

{ strtoul_ , 文字列のアドレス, 終了を書き込む変数のアドレス, 変換方法 };

数値を表現する文字列を解析して32ビット整数にします。戻り値が変換された整数です。

終了を書き込む変数のアドレスが指定されている場合は(0でない場合は)解析終了した文字列の次のアドレスが書き込まれます。

変換方法は8の場合8進数, 10の場合十進数変換, 16の場合16進数変換をおこないます。

変換方法0の場合は自動判定します。

5.6 組み込み関数 strcmp_

{ strcmp_, 比較文字列1のアドレス, 比較文字列2のアドレス };

文字列を比較します。文字列の終了はコード0です。

文字列が完全に一致している場合は戻り値が0となります。

比較文字列1の方が文字列2より小さかった場合は負の値を、大きかった場合は正の値を返します。

5.7 組み込み関数 `strlen_`

`{ strlen_, 文字列のアドレス };`

戻り値が文字列の長さとなります。文字列の終了はコード 0 です。

6. 組み込み関数—ファイル

iomacro の ROM ファイルやデバイスドライバをアクセスするための関数が用意されています。次の5本の関数があります。

<i>open_</i>	ファイルをオープンする
<i>close_</i>	ファイルをクローズする
<i>read_</i>	ファイルからデータを読み出す
<i>write_</i>	ファイルへデータを書き込む
<i>ioctl_</i>	デバイスドライバを制御する

6.1 組み込み関数 `open_`

```
{ open_, ファイル名, モード };
```

ファイルをオープンします。ファイルはROMファイルかデバイスドライバです。

モードは

- 0 リードオンリー
- 1 ライトオンリー
- 2 リードライト

戻り値が負の場合はエラー、0か正の場合はファイル番号です。

例

LEDデバイスドライバをライトオンリーでオープンします。

```
fd = { open_, "/dev/led", 1 };
```

6.2 組み込み関数 close_

`{ close_ , ファイル番号 };`

ファイル番号で指定するファイルをクローズします。ファイル番号はファイルをオープンしたときに与えられます。

6.3 組み込み関数 read_

{ read_ , ファイル番号, バッファアドレス, 読み出しサイズ };

ファイル番号で指定するファイルからデータを読み出しサイズ分バッファアドレスで示すバッファへ読み出します。戻り値は正なら読み出しサイズ, 負ならエラーです。

6.4 組み込み関数 `write_`

`{ write_, ファイル番号, バッファアドレス, 書き込みサイズ };`

ファイル番号で指定するファイルヘータを書き込みサイズ分バッファアドレスで示すバッファから書き込みます。戻り値は正なら書き込んだサイズ, 負ならエラーです。

6.5 組み込み関数 ioctl_

{ ioctl_, ファイル番号, 要求コード, 固有データ };

ファイル番号で指定するデバイスドライバの `ioctl` を呼びます。要求コードと固有データはデータごとに定義されています。戻り値が0か正なら正常終了、負ならエラーです。

7. データ表示組み込み関数 `print_`

`iomacro` には表示データを生成するための関数 `print_` が用意されています。この関数は C の関数 `sprintf` ような書式指定でメモリ上に表示用文字列を生成します。

この関数は次のように呼び出します。

```
{ print_, データアドレス, 書式, ... };
```

戻り値は生成した文字列の長さです。

たとえば 32 文字の領域を確保して、そこに変数 `x` の値を表示データとして書き込む場合は、

```
p = { carray_, 32 };  
{ print_, $p, "x=%ld", $x };
```

とします。

このデータは次のようにしてシリアルポートへ出力することができます。

```
{ conl_, "%s", $p };
```

関数 `print_` はこのように表示データをメモリ上に生成しますが、データアドレスを `0` にすることでその出力先を呼び出しもとの状況にあわせて自動的に決定します。

```
{ print_, 0, "x=%ld", $x };
```

この記述は通常シリアルポートへ出力します。

SSI 関数に上記記述をした場合には Web ページの SSI 部分へ出力されます。

TCP サーバ関数が上記記述をした場合にはクライアントへの応答文字列となります。

このため SSI や TCP サーバ関数のデバッグでメイン関数から呼び出しシリアルポートへ表示して確認することができます。

8. 組み込み関数－Web 関連

iomacro 関数は ConDulan 内蔵の HTTP サーバと協調して Web アクセスから起動することができます。この章では iomacro 関数を Web アクセスで利用するために用意された組み込み関数を紹介します。

Web 関連の組み込み関数は次の11本があります。

<i>ssi_</i>	関数を <i>ssi</i> として登録する
<i>cgiget_</i>	関数を GET 用 CGI として登録する
<i>cgipost_</i>	関数を POST 用 CGI として登録する
<i>htmlquery_</i>	GET 要求中の指定クエリーを検出する
<i>htmlpost_</i>	POST 要求のデータを解析する
<i>htmlget_</i>	GET 要求のデータを解析する
<i>htmlaccess_</i>	Web アクセスの認証状況を取得する
<i>htmlrefresh_time_</i>	GET 要求クエリー中の <i>refresh_interval</i> を検出する
<i>getcgi_</i>	Web ページの環境変数取得 (SSI で使用)
<i>printcgi_</i>	Web ページの環境変数追加 (文字列)
<i>printcgilong_</i>	Web ページの環境変数追加 (整数)

8.1 組み込み関数 ssi_

```
{ ssi_, SSI 名, 関数 };
```

iomacro で書かれた関数を SSI 名で SSI として登録します。Web ページを開いたときに次の記述が見つかり Web サーバから呼び出されます。

```
<!--#exec cgi="SSI 名" -->
```

SSI 関数に引数を渡したい場合は

```
<!--#exec cgi="ssiA abc 012" -->
```

のように SSI 名の後にスペースで区切って文字列を指定します。SSI 関数には `_1`, `_2` など引数にその文字列へのアドレスが用意されます。

なお SSI 関数内の最初の引数 `_0` には SSI 名のアドレスが格納されます(上記例では `ssiA` という文字列のある領域のアドレス)。実質的な最初の引数は `_1` となります。

8.2 SSI 関数

SSIとして登録した関数は通常表示データを生成します。生成されたデータは Web ページに組み込まれて表示されます。

表示データの生成には組み込み関数 `print_` を使用します。また関数の戻り値は生成したデータサイズを返します。

関数 `print_` は生成したデータサイズを返すので、`ssiA` というデータを固定的に返す SSI 関数は次のようになります。

```
{ ssi_, "ssiA", ssiA };  
.exit
```

```
@ssiA  
_ = { print_, 0, "ssiA" };  
.return  
.end
```

8.3 組み込み関数 `cgiget_`

{ cgiget_, CGI 名, 関数 };

関数を CGI 名で GET 用 CGI として登録します。Web サーバは GET 要求の URL が CGI として登録されているとその登録関数を呼び出します。

8.4 組み込み関数 `cgipost_`

`{ cgipost_ , CGI 名, 関数 };`

関数を CGI 名で POST 用 CGI として登録します。Web サーバは POST 要求の URL が CGI として登録されているとその登録関数を呼び出します。

8.5 組み込み関数 `htmlquery_`

```
{ htmlquery_, クエリー名 };
```

Web の GET 要求から指定クエリー名を持つクエリーの値を取得する関数です。戻り値が0の場合は指定クエリー名は存在しないことを示します。クエリーがあった場合はその値を保存している領域のアドレスを返します。

例

GET で名前が `da0` か `da1` のクエリーがあった場合はその値をそれぞれチャンネル0と1の DA へ出力します。

```
@query_da  
  _val = { htmlquery_, "da0" };  
  .if $_val .then { da_, 0, .sl.$_val }; .endif  
  _val = { htmlquery_, "da1" };  
  .if $_val .then { da_, 1, .sl.$_val }; .endif  
  .return
```

8.6 組み込み関数 `htmlaccess_`

```
{htmlaccess_};
```

CGI 関数が Web 要求の認証結果を取得するために使用します。

戻り値の意味はビット対応で

- ビット 0(LSB) GET の IP 制限で認証されていない
- ビット 1 GET のパスワード制限で認証されていない
- ビット 2 POST の IP 制限で認証されていない
- ビット 3 POST のパスワード制限で認証されていない

となります。

通常 CGI 関数は Web アクセス認証が許可された場合だけ呼び出されますのでこの組み込み関数を使用する必要はありません。ConDuLan が Web アクセス制限を読み出し(GET)は無条件で許可、書き込み(POST)は認証必要と設定している場合に、GET クエリーで書き込みをおこなう手順に対して認証を必要とするよう設定して書き込みを制限するという特殊な制御をおこなうためにこの関数を使用します。すなわち GET 用 CGI で POST のアクセス認証を適用する場合に使用します。

GET の CGI で POST のアクセス認証を適用するにはこの組み込み関数の戻り値を調べビット 2 がセットされていれば IP 制限で認証されていないので CGI 関数は-403(forbidden)を返してアクセスを許可しません。またビット 3 がセットされている場合はパスワード認証されていないので-401(unauthorized)を返しパスワード認証を促します。

8.7 組み込み関数 `htmlpost_`

```
{ htmlpost_, 値保存アドレス };
```

Web の POST 要求データを解析する関数です。POST 用 CGI として登録した `iomacro` 関数がこの関数を呼ぶと未解析 POST データ中の先頭の名前を保存している領域のアドレスを戻り値として戻ってきます。戻り値が 0 の場合はもはや解析する POST データが残っていないことを示します。POST データがあった場合は値保存アドレス に記述した変数に値を保存している領域のアドレスをセットします。

通常戻り値が 0 になるまでこの関数を繰り返し呼び、取得した名前と値の処理を実行します。あらかじめ POST データが 1 個だけとわかっている場合は 1 度だけ呼ぶこともあります。

例

POST で名前が `da0` か `da1` のデータがあった場合はその値をそれぞれチャンネル 0 と 1 の DA へ出力します。

```
@post_da
```

```
  _name = { htmlpost_, _val };  
  .if "da0" .==. $_name .then { da_, 0, .sl.$_val };  
  .else  
    .if "da1" .==. $_name .then { da_, 1, .sl.$_val };  
  .endif  
  .endif  
  .return
```

8.8 組み込み関数 `htmlget_`

```
{ htmlget_, 値保存アドレス };
```

Web の GET 要求クエリーを解析する関数です。GET 用 CGI として登録した `iomacro` 関数がこの関数を呼ぶと未解析 GET クエリー中の先頭の名前を保存している領域のアドレスを戻り値として戻ってきます。戻り値が 0 の場合はもはや解析する GET クエリーが残っていないことを示します。GET クエリーがあった場合は値保存アドレス に記述した変数に値を保存している領域のアドレスをセットします。

通常戻り値が 0 になるまで繰り返しこの関数を呼び、取得した名前と値の処理を実行します。あらかじめ GET クエリーが 1 個だけとわかっている場合は 1 度だけ呼ぶこともあります。

例

GET で名前が `da0` か `da1` のデータがあった場合はその値をそれぞれチャンネル 0 と 1 の DA へ出力します。

```
@get_da
```

```
    _name = { htmlget_, _val };  
    .if "da0" ==. $_name .then { da_, 0, .sl.$_val };  
    .else  
        .if "da1" ==. $_name .then { da_, 1, .sl.$_val };  
    .endif  
    .endif  
    .return
```

8.9 組み込み関数 `htmlrefresh_time_`

```
{htmlrefresh_time_};
```

Web の GET 要求クエリー中の *refresh_interval* という名前の値を取得する関数です。繰り返し同じページを要求するような場合の要求時間をSSIで生成する場合に使われます。

8.10 Web ページ環境変数

PCからの各 Web ページ要求には ConDulan 内部でそのページ固有の情報が付加されます。ConDulan ではこの情報を Web ページ環境変数と呼んでいます。たとえばそのページの URL 名は環境変数として保持されています。環境変数は iomacro から追加することが可能です。また追加した環境変数を iomacro から読み出すための関数も用意されています。

たとえば Web ページで AD 変換結果を表示する場合を考えて見ます。通常 AD 変換する SSI をページに埋め込んでおけば AD 変換結果を見ることができます。しかし 1 ページの中でチャンネル 0 の AD 変換を生データ(0-1023)と電圧(0-4.1)で表示するとそれぞれの表示を SSI で記述することになります。この場合異なる時刻で AD 変換するので生データと電圧データの結果が異なる場合が発生する可能性があります。この状態を避けるために AD 変換ページ要求があったときに AD 変換を実行し、そのページの環境変数として AD 変換結果を登録しておくことができます。ページ内の SSI は AD 変換を実行するのではなく、ページの環境変数に書き込まれた AD 変換結果を表示することになり、その値はいつでも Web ページ要求されたときの変換結果を使用できます。

このように Web 環境変数はページ内で一貫性を保つために SSI が動作する前に CGI が設定します。

Web 環境変数関係の組み込み関数は次の3本があります。

- `getcgi_` Web ページ環境変数を取得する(通常 SSI で使用する)
- `printcgi_` Web ページに環境変数(文字列)を追加する
- `printcgilong_` Web ページに環境変数(整数)を追加する

8.11 組み込み関数 `getcgi_`

```
{ getcgi_, Web 環境変数名 };
```

名前を指定して Web 要求ページの環境変数を取得します。戻り値が Web 環境変数値のアドレスとなります。指定環境変数がない場合は戻り値は 0 になります。

例

環境変数 `val_x` の値を表示する SSI 関数です。

```
@ssi_val_x  
  _p = { getcgi, "val_x" };  
  .if $ _p .then  
    _ = { print_, 0, "%s", $ _p }  
  .endif  
  .return
```

8.12 組み込み関数 `printcgi_`

```
{printcgi_, Web 環境変数名, 書式, ... };
```

名前を指定して Web 要求ページの環境変数を追加します。書式以後は関数 `co_` の場合と同じです。追加した環境変数は SSI で組み込み関数 `getcgi_` により取得することができます。戻り値は登録した Web 環境変数の長さです。

例

Web 環境変数 `val_x` の値を 1000 として登録します。

```
{printcgi_, "val_x", "1000"};
```

8.13 組み込み関数 `printcgilong_`

```
{ printcgi_, Web 環境変数名, Web 環境変数値 };
```

名前を指定して Web 要求ページの環境変数を追加します。Web 環境変数値は32ビット整数で、数値を表す文字列として登録されます。

追加した環境変数は SSI で組み込み関数 `getcgi_` により取得することができます(文字列として取得されます)。戻り値は登録した Web 環境変数の長さです。

例

Web 環境変数 `val_x` の値を 1000 として登録します。

```
{ printcgilong_, "val_x", 1000 };
```

9. 組み込み関数ープルダウンメニュー

この章では Web ページに組み込まれるプルダウンメニューの関数を紹介します。プルダウンメニューのための関数は次の3本が用意されています。

<code>SSIPULLDOWN_</code>	プルダウンメニューを生成する
<code>GETPULLDOWN_</code>	ConDuLan 内蔵プルダウンメニューを生成する
<code>SSISELECTPULLDOWN_</code>	プルダウンメニューから指定値を検索する

プルダウンメニューはあらかじめメニュー情報を用意しておき、Web ページ要求時に SSI 関数が `SSIPULLDOWN_` を使用してメニューをページに埋め込みます。

プルダウンメニュー選択情報が送られてきた場合には CGI が `SSISELECTPULLDOWN_` を使用して解析し、メニューインデックスを取得します。

9.1 Web 上でのプルダウンメニュー書式

Web 上でのプルダウンメニューは、たとえば次のような書式をしています。

```
<form method="post" enctype="text/plain" action="/bps_settings.html">
<select name="speed">
<option value="2400">2.4K</option>
<option value="4800">4.8K</option>
<option value="9600" selected>9.6K</option>
<option value="19200">19.2K</option>
<option value="38400">38.4K</option>
<option value="57600">57.6K</option>
</select>
<br>
<input type="submit" name="set" value="Set">
</form>
```

これはシリアル転送速度の指定メニューの例ですが、set をクリックすると ConDuLan へ

speed=9600

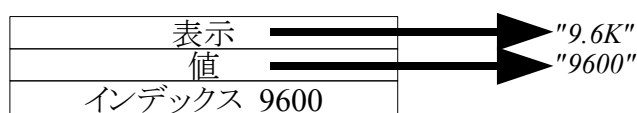
のデータが送信されます。送信データの左辺はプルダウンメニュー中の name= で定義されている speed となります。送信データの右辺は value= で定義されている中でメニュー選択された値となります。メニュー選択の初期値は値の後ろに selected が置かれています。

9.2 iomacro のプルダウンメニュー情報

プルダウンメニューを生成するために iomacro はひとつのメニュー要素に対して表示、値、インデックス(整数)の3要素を用意します。表示と値は文字列のある領域のアドレスとなります。インデックスは iomacro が処理しやすいように設けられた任意の整数値です。

下の図 A はメニュー要素 9600 の例です。表示は"9.6K"となり、ConDulan へ送られる値は文字列 "9600"となります。

プルダウンメニューの要素を連続して配列に用意し、関数 `ssipulldown_` を使用して Web 表示するプルダウンメニューを生成することができます。図 B はプルダウンメニュー要素を連続して配列に用意したプルダウンメニュー情報の例です。最後に終了コードとして 0 を追加します。確保した領域はメニュー6要素に終了コードを含めて19ワードとなります。



A. 9600 のメニュー要素の例

"2.4K"
"2400"
2400
"4.8K"
"4800"
4800
"9.6K"
"9600"
9600
"19.2K"
"19200"
19200
"38.4K"
"38400"
38400
"57.6K"
"57600"
57600
0
0
0

B. プルダウンメニュー情報の例

9.3 iomacro のプルダウンメニュー情報

ここでシリアル転送速度のプルダウンメニュー情報生成プログラムの例を紹介します。

プルダウンメニュー用領域 19 ワードを *bps_menu* に確保し、各メニュー要素を書き込みます。

デフォルトの転送速度を変数 *bps* に書き込みます(9600)。

Web にプルダウンメニューを表示するには ssi 登録された関数(この例では *ssi_bps*)からプルダウンメニュー情報 *bps_menu* を指定して、プルダウンメニュー生成関数 *ssipulldown_* を呼びます。

```
bps_menu = { array_, 6*3+1 };
p=bps_menu[0];
$p = "2.4K";
$p+4="2400";
$p+8=2400;

p = $p+12;
$p = "4.8K";
$p+4="4800";
$p+8=4800;

p = $p+12;
$p = "9.6K";
$p+4="9600";
$p+8=9600;

p = $p+12;
$p = "19.2K";
$p+4="19200";
$p+8=19200;

p = $p+12;
$p = "38.4K";
$p+4="38400";
$p+8=38400;

p = $p+12;
$p = "57.6K";
$p+4="57600";
$p+8=57600;

p = $p+12;
$p = 0;
//$p+4=0;
//$p+8=0;

//default bps
bps = 9600;
//ssi registration
{ ssi_, "bps", ssi_bps };
```

9.4 組み込み関数 `ssipulldown_`

`{ ssipulldown_ , メニュー名, メニュー情報, 現在のインデックス };`

Web ページにプルダウンメニューを生成します。通常 SSI 関数の中で使用します。
メニュー名はこのメニューの選択結果を ConDuLan に送るときの名前部分になります。
メニュー情報はプルダウンメニュー選択肢情報が格納されている領域のアドレスです。
現在のインデックスはプルダウンメニュー中の現在選択されている項目のインデックスです。
戻り値は Web ページに生成したプルダウンメニューデータサイズです。

例

シリアル転送速度を選択するプルダウンメニュー生成です。メニュー情報 `bps_menu` はあらかじめメインで用意しておきます。変数 `bps` には現在のボーレートが格納されています。Web ページにこの ssi を埋め込む記述をしておけばシリアル転送速度選択のプルダウンメニューが埋め込まれます。

```
@ssi_bps
_ = { ssipulldown_ , "speed", bps_menu[0], $bps };
.return
```

9.5 組み込み関数 `getpulldown_`

```
{ getpulldown_, 内蔵メニュー名};
```

ConDuLan 内蔵のプルダウンメニューを取得します。

内蔵メニュー名は現在次の2種類が用意されています。

"bps"

"interval"

戻り値は該当するメニュー情報アドレスです。該当メニューがない場合は 0(NULL)が返ります。シリアル転送速度などはすでに紹介したように `iomacro` でメニューを用意することもできますが、この関数を使用することもできます。

"bps" で生成されるメニューは 2400,4800,9600,19200,38400,57600 となります。メニュー名とメニュー値は同じ値を示す文字列で、インデックスはその整数値となります。

"interval" で生成されるメニューは 1S,2S,3S,4S,5S,6S,10S,12S,15S,20S,30S,60S,2M,3M,4M,5M,6M,10M,12M,15M,20M,30M,60M,2H,3H,4H,6H,8H,12H,24H となります。メニュー値はすべて秒数を表す文字列で"1","2",..."43200","86400"のようになります。インデックスは秒数値となります。

例

シリアルボーレートメニューとインターバルメニュー生成です。

```
bps = { getpulldown_, "bps" };
```

```
interval = { getpulldown_, "interval" };
```

9.6 組み込み関数 `ssiselectpulldown_`

```
{ getpulldown_, メニュー 情報, メニュー値};
```

この関数は通常 CGI 内で使用され、ConDuLan に送られてきたプルダウン選択データから、インデックス値を取得します。インデックス値は文字列ではなく整数ですから iomacro で用意に扱えます。

メニュー情報は ssi メニューを生成したときのメニュー情報です。

メニュー値は ConDulan に送られてきたプルダウン選択データの値(文字列)です。

戻り値がインデックスとなります。該当する値がない場合はメニュー先頭のインデックスが返ります。引数が不足した場合は 0 が返ります。

例

シリアルの転送速度を選択する CGI です。

関数 `post_bps` はあらかじめ POST 用 CGI として登録しておきます。シリアル転送速度のプルダウンメニューが選択されて `speed=9600` のようなデータが送られた場合は転送速度を設定します。その場合文字列"9600"は関数 `ssiselectpulldown_` で整数値 9600 に変換されます。

```
@post_bps
```

```
  .while _name = { htmlpost_, _val } .do  
    .if "speed" .==. $_name .then  
      bps = { ssiselectpulldown_, bps_menu[0], $_val };  
      { cb_, $bps };  
    .endif  
  .endwhile  
  .return
```

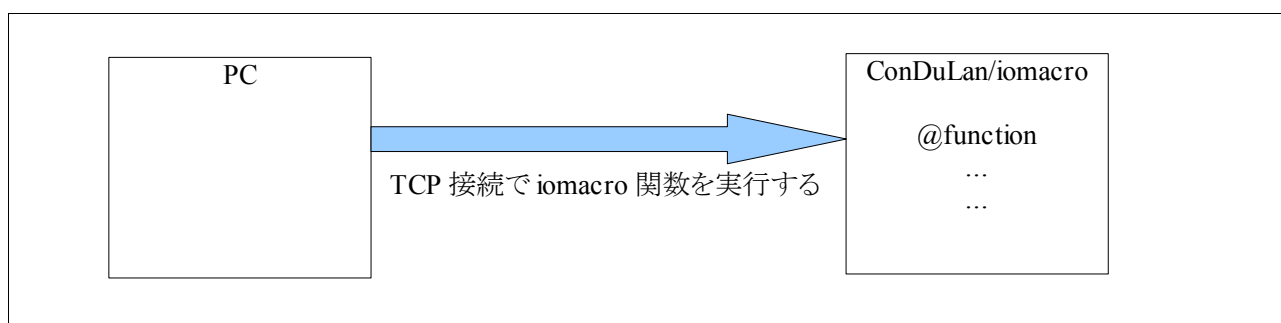
10. 組み込み関数－TCP

ConDuLan には iomacro 関数を PC の TCP/IP 接続で実行する手順が用意されています。

PC は ConDuLan へ TCP 接続し iomacro が公開した関数を呼び出すことができます。

TCP 手順による iomacro 関数呼び出しを実行するために次の4本の組み込み関数が用意されています。

- disclose_* iomacro 関数を TCP 呼び出し用に公開する
- tcpserver_* TCP 関数呼び出しサーバーを起動する
- tcppasswd_* TCP 関数呼び出しのパスワードを設定する
- tcpip_* TCP 関数呼び出しの IP 制限を設定する



10.1 TCP 関数呼び出し手順

TCP 手順で iomacro 関数を呼び出すには、その関数を *disclose_* で公開し組み込み関数 *tcpserver_* で TCP 関数呼び出しサーバーを起動しておきます。

この状態で PC が iomacro 用 TCP サーバへ接続し、次の手順で関数呼び出しをおこなうことができます。

PC から1行パスワードを送る(パスワードなしの場合は改行だけ)。

ConDuLan から改行が送られる。

PC から 関数名 引数 引数 . . . の書式で1行要求を送る。

ConDuLan は指定の関数を実行する。

関数の実行結果を改行付きで PC へ送る。

PC は次の 関数 引数 引数 . . . を送る。以後繰り返し。

関数の実行でエラーがあった場合は ConDuLan から TCP 接続を切断する。

もはや PC から関数呼び出しが必要ない場合は PC から切断する。

iomacro の *tcpserver_* 接続時の設定で複数の TCP 接続が可能である。

改行は CR+LF である。

10.2 組み込み関数 `disclose_`

```
{ disclose_, 関数名, 関数位置 };
```

`iomacro` 関数を TCP 関数呼び出しで使用できるよう公開します。

関数名は TCP 関数呼び出しで使用する関数の名前です。

関数位置は `iomacro` 関数です。

戻り値は 0 か正で公開成功。負の値で公開失敗です。

例

`iomacro` 関数 `add` を関数名 "add" として公開する。

関数 `add` は文字列で与えられる二つの引数を整数変換して加算し、文字列として出力する。

```
{ disclose_, "add", add };  
  
.exit  
  
@add  
  _ = { print_, 0, "%ld", .sl_1 + .sl_2 };  
  .return  
  
.end
```

10.3 組み込み関数 tcpserver_

```
{ tcpserver_ , ポート番号, サーバタスク本数, 切断制限時間, バックログ数 };
```

TCP 手順で iomacro 関数を呼び出すためのサーバタスクを起動します。

起動されたサーバタスクはポート番号で指定されたポートで接続を待ち受けます。

TCP 関数呼び出しは同時に複数の接続を可能としています。可能な最大接続数をサーバタスク本数で指定します。

接続後 PC から一定時間要求がないと ConDuLan は TCP を切断します。このときの制限時間を *切断制限時間* で指定します。単位は ms です。関数呼び出しの必要がなくて接続だけ維持するには PC から改行を送り、改行応答を待ちます。

TCP 関数呼び出しの TCP 最大接続数に達した状態でさらに TCP 関数呼び出し接続要求があった場合、最大バックログ数まではその要求を保持し、サーバタスクに空きができた時点で接続します。それ以上は接続を拒否します。バックログ数を0にすると必ず接続するか、接続数が最大になっていた場合接続拒否することになります。

例

次の例は関数 *add* と *sub* を TCP 関数呼び出し可能にしています。TCP ポート番号は 29000 で最大2本の接続が可能です。通信なしの場合の切断制限時間は 60 秒で、バックログは無しです。関数の最初の引数 *_0* はその関数名となります。

```
{ disclose_ , "add", add };
{ disclose_ , "sub", sub };
{ tcpserver_ , 29000, 2, 60000, 0 };
.exit
@add
    _ = { print_ , 0, "%ld", .sl._1 + .sl._2 };
    .return
@sub
    _ = { print_ , 0, "%ld", .sl._1 - .sl._2 };
    .return
.end
```

10.4 組み込み関数 tcppasswd_

```
{ tcppasswd_, パスワード };
```

TCP 関数呼び出し手順のパスワードを設定します。パスワードは最大63文字です。パスワードは TCP 関数呼び出し手順で PC からのパスワードと比較し、一致した場合だけ関数呼び出しを許可します。

この関数で設定したパスワードは環境変数としてシリアル ROM に保存されます。キーは

TCPSERVER_PASSWD

です。

戻り値は 0 で設定成功。負の値で設定失敗です。

パスワード設定を TCP 関数呼び出しからおこなえるよう通常次のような関数 *passwd* が公開されます。

例

TCP 関数呼び出しから使用するパスワード設定関数

```
{ disclose_, "add", add };
{ disclose_, "passwd", passwd };
{ tcpserver_, 29000, 2, 60000, 0 };
.exit
@add
    _ = { print_, 0, "%ld", .sl._1 + .sl._2 };
    .return
@passwd
    .if 0 <= { tcppasswd_, _1 };
    .then _ = { print_, 0, "passwd succeeded" };
    .else _ = { print_, 0, "passwd failed" };
    .endif
    .return
.end
```

10.5 組み込み関数 tcpip_

{ tcpip_ , 許可するIP アドレス, サブネットマスク };

TCP 関数呼び出し手順の IP アドレス制限を設定します。

接続要求のあった PC の IP アドレスとこの関数で設定した IP アドレスをサブネットマスクが 1 のビットを比較して一致した場合に接続許可します。サブネットマスクが指定されない場合は 255.255.255.255 が使用されます。この場合は1台の PC だけに接続許可が与えられます。

例

IP アドレス 192.168.98.xx を許可する。

{ tcpip_ , .ip. "192.168.98.0" , .ip. "255.255.255.0" };

11. 組み込み関数－UDP

iomacro では UDP 手順の組み込み関数を用意してあります。自身のポート番号を指定して UDP エンドポイントを確保し、そのエンドポイントを経由して送受信します。

UDP 手順用に次の4本の関数を用意されています。

- udp_* UDP エンドポイントを確保する.
- udpsend_* UDP エンドポイントから送信する UDP パケットを.
- udprecv_* UDP エンドポイントから UDP パケットを受信する.
- udpclose_* UDP エンドポイントを開放する.

11.1 組み込み関数 `udp_`

```
{ udp_, ポート番号 };
```

自身のポート番号を指定して UDP エンドポイントを用意します。戻り値がエンドポイントのアドレス(ネットワークアドレスではなく内部アドレス)となります。

エラーの場合は戻り値 0 となります。

UDP の送受信はこのエンドポイントアドレスを指定しておこないます。

例

ポート番号 29000 で UDP エンドポイントを用意します。

```
udpep = { udp_, 29000 };
```

11.2 組み込み関数 `udpsend_`

`{ udpsend_ , エンドポイント, IP アドレス, ポート番号, データアドレス, データサイズ };`

UDP パケットを送信します。送信元の IP アドレスは ConDulan のアドレスを使用し、送信元のポート番号はエンドポイントに指定した番号を使用します。

送信先の IP アドレスとポート番号を指定してデータアドレスにあるデータサイズの大きさのデータを UDP 送信します。データサイズは UDP 1 パケットに収まるサイズでなければなりません。戻り値は送信データサイズとなります。負の場合は送信失敗です。

例

ポート番号 29000 で UDP エンドポイントを用意し、IP アドレス 192.168.98.69 のポート 29100 にデータを送ります。データは `udp data` です。

```
data = { carray_ , 1500 };  
.if 0 != $data  
.then  
    sz = { print_ , $data, "udp data" };  
    udpep = { udp_ , 29000 };  
    .if 0 != $udpep  
    .then  
        { udpsend_ , $udpep, .ip. "192.168.98.69", 29100, $data, $sz };  
    .endif  
.endif
```

11.3 組み込み関数 `udprecv_`

`{ udprecv_ , エンドポイント, IP アドレス保存変数, ポート番号保存変数, データ領域, データ領域サイズ, 受信制限時間 };`

UDP パケットを受信します。

送信元の IP アドレスは `IP アドレス保存変数` に保存されます。

また送信元ポート番号は `ポート番号保存変数` に保存されます。

受信データは `データ領域` に保存されます。

`データ領域` のサイズは `データ領域サイズ` で指定します。

`受信制限時間` は、この時間経過してもデータ受信しなかった場合にこの関数を終了するために使用します。単位は ms で、0 は即終了、-1 は制限時間なしとなります。制限時間がない場合は -1 として扱われます。

戻り値は受信データのサイズで、制限時間超過によるリターンの場合は -1 となります。

例

ポート番号 29100 で UDP パケット受信します。

```
data = { carray_ , 1500 };
.if 0 != $data
.then
    udpep = { udp_ , 29100 };
    .if 0 != $udpep
    .then
        .while 1 .do
            sz = { udprecv_ , $udpep, fromip, fromport, $data, 1500, -1 };
            { conl_ , "udprecv from %08lX/%ld sz %ld", $fromip, $fromport, $sz };
        .endwhile
    .endif
.endif
.exit
```

11.4 組み込み関数 udpclose_

{ *udpclose_*, エンドポイント };

UDP エンドポイントを開放します。

12. 組み込み関数－eメール

ConDuLan には eメールを送出する組み込み関数が用意されています。送信できる eメールは添付ファイルなしで送信手順は SMTP で認証なしとプレーン認証の2種類があります。送信ポートは SMTP 標準の 25 以外にも送信できますのでサブミッションポート(587)を利用することも可能です。

ConDuLan から eメールを送信するためには送り元アドレスと宛先アドレスに加えて eメールサーバの IP アドレスとポート番号が必要となりますのであらかじめご用意ください。

また、送信に認証が必要な場合は eメールサーバのユーザアカウントとパスワードも必要になります。

iomacro に用意されている eメール関数は次の2本です。

<code>mail_</code>	認証なし eメール送信
<code>mailauth_</code>	認証付き eメール送信 (認証なしも可)

12.1 組み込み関数 mail_

{ mail_ , サーバIP, サーバポート, helo, from, to, ヘッダ, 本文 };

eメールを送信します。

サーバIPは32ビット整数、サーバポートも32ビット整数で通常25です。helo, from, to はSMTPで定義されているそれぞれの文字列のアドレスです。ヘッダと本文はそれぞれの文字列のアドレスとともに複数行を含みます。通常CRとLFで改行です。

メール送信(SMTP)の各情報についてはRFC2821をご参照ください。

例

架空のメールアドレス abc@defghijk.com から def@ghijklmn.com へメール送信します。メールサーバは架空の 192.168.98.250 としています。

```
mail_server = .ip."192.168.98.250";
mail_helo = "defghijk.com";
mail_from = "abc@defghijk.com";
mail_to = "def@ghijklmn.com";
mail_hdr = "From: abc<abc@defghijk.com>\r\nTo: def<def@ghijklmn.com>\r\nSubject:
TEST\r\n";
mail_msg = "This is a test e-mail.\r\n";
{ mail_ , $mail_server, 25, $mail_helo, $mail_from, $mail_to, $mail_hdr, $mail_msg };
.exit
```

12.2 組み込み関数 mailauth_

{ mailauth_ , サーバIP, サーバポート, 認証オン-オフ, 認証ユーザ名, 認証パスワード, 認証オプション, helo, from, to, ヘッダ, 本文 };

eメールを認証手続きで送信します。認証なしで送信することもできます。

サーバIPは32ビット整数、サーバポートも32ビット整数です。認証オン-オフは整数で0は認証なし、そのほかの値は認証ありです。認証ユーザ名と認証パスワードは共にeメールサーバに登録してある文字列です。認証オプションは0が指定された場合認証ユーザ名が使用されます。helo, from, to はSMTPで定義されているそれぞれの文字列のアドレスです。ヘッダと本文はそれぞれの文字列のアドレスとともに複数行を含みます。

例

架空のメールアドレス abc@defghijk.com から def@ghijklmn.com へメール送信します。メールサーバは架空の 192.168.98.250 としています。ポート番号はサブミッションポート 587 を使用します。認証ユーザ名は架空の abc パスワードは ABC としています。

```
mail_server = .ip."192.168.98.250";
mail_port = 587;
mail_auth = 1;
mail_user = "abc";
mail_passwd = "ABC";
mail_helo = "defghijk.com";
mail_from = "abc@defghijk.com";
mail_to = "def@ghijklmn.com";
mail_hdr = "From: abc<abc@defghijk.com>\r\nTo: def<def@ghijklmn.com>\r\nSubject:
TEST\r\n";
mail_msg = "This is a test e-mail.\r\n";
{ mailauth_ , $mail_server , $mail_port , $mail_auth , $mail_user , $mail_passwd , 0 , $mail_helo ,
$mail_from , $mail_to , $mail_hdr , $mail_msg };
.exit
```

13. 組み込み関数ーポート入出力

iomacro では内蔵 CPU のハードウェアに直接アクセスしてポート入出力をおこなうことができますが、複数のタスクがポートアクセスする場合排他制御を考慮する必要がありますので、デジタル入出力ポートに関する組み込み関数を用意しています。

関数は次の7本です。

<i>portget_</i>	ポートの入力データを取得する
<i>portset_</i>	ポートへ HIGH を出力する
<i>portclr_</i>	ポートへ LOW を出力する
<i>portgetconf_</i>	ポートの入出力設定状態を状態番号で取得する
<i>portgetconfname_</i>	ポートの入出力設定状態を状態名で取得する
<i>portsetconfname_</i>	ポートの入出力設定状態を状態名で設定する
<i>portsaveconf_</i>	ポートの入出力設定状態を ROM に保存する

13.1 ConDuLan のポート

ConDuLan はデジタルの入出力ポートを40点内蔵しています。そのうち20点は入力専用, 20点は入出力兼用となっています。40点のポートはすべて内蔵 CPU に組み込まれているポートです。

ConDuLan ではこれらのポートを次のいずれかに設定して使用します(すべてのポートがどの設定も可能なわけではありません)。

状態	状態番号	状態名
入力	0	in
プルアップ付き入力	1	pullup
出力	2	out
オープンドレイン出力	3	open

プルアップ付き入力は内蔵 CPU にプルアップ入力を用意されているポートだけ有効です。

出力とオープンドレイン出力は出力が可能なポートだけ有効です。

オープンドレイン出力は **LOW** 出力のときポートを出力に設定し, **HIGH** 出力のときは入力に設定して高インピーダンスにします。

13.2 ConDuLan のポート名称

ConDuLan の 40 点入出力ポートは下の表のような名称と内部番号を持っています。名称は内蔵 CPU のポート名です。併せて各ポートのピン番号も記載します。

ポート名称とピン番および可能な構成

番号	名称	ピン番	構成
0	P23	CN1-6	in, pullup
1	P24	CN1-5	in, pullup
2	P25	CN1-8	in, pullup
3	P26	CN1-7	in, pullup
4	P27	CN1-10	
5	P40	CN2-17	in, pullup, out, open
6	P41	CN2-20	in, pullup, out, open
7	P42	CN2-22	in, pullup, out, open
8	P43	CN2-19	in, pullup, out, open
9	P44	CN2-24	in, pullup, out, open
10	P45	CN2-21	in, pullup, out, open
11	P46	CN2-26	in, pullup, out, open
12	P47	CN2-23	in, pullup, out, open
13	P50	CN1-9	in, pullup
14	P51	CN1-12	in, pullup
15	P52	CN1-11	in, pullup
16	P53	CN1-13	in, pullup
17	P70	CN1-20	in
18	P71	CN1-22	in
19	P72	CN1-24	in

番号	名称	ピン番	構成
20	P73	CN1-26	in
21	P74	CN1-28	in
22	P75	CN1-30	in
23	P76	CN1-32	in
24	P77	CN1-34	in
25	P80	CN2-4	in
26	P81	CN2-3	in
27	P84	CN2-6	in
28	PA0	CN2-5	in, out, open
29	PA1	CN2-8	in, out, open
30	PA2	CN2-7	in, out, open
31	PA3	CN2-10	in, out, open
32	PA4	CN2-12	in, out, open
33	PA5	CN2-9	in, out, open
34	PA6	CN2-11	in, out, open
35	PA7	CN2-14	in, out, open
36	PB0	CN2-13	in, out, open
37	PB1	CN2-16	in, out, open
38	PB2	CN2-15	in, out, open
39	PB3	CN2-18	in, out, open

13.3 組み込み関数 portget_

```
{ portget_ , ポート名 };
```

ポート名で指定するポートの入力データを取得します。

戻り値は

- 0 入力データは LOW
- 1 入力データは HIGH
- 1 指定ポート名が不正

です。

例

1秒間隔でポート p23 の内容をシリアルへ表示する。

```
.while 1 .do
    { conl_ , "%ld", { portget_ , "p23" } };
    { msleep_ , 1000 };
.endwhile
.exit
.end
```

13.4 組み込み関数 portset_, portclr_

```
{ portset_ , ポート名 };
```

```
{ portclr_ , ポート名 };
```

ポート名で指定するポートの出力を HIGH(*portset_*)あるいは LOW(*portclr_*)にします。

戻り値は

0 正常に出力した

-1 指定ポート名が不正

です。

実際に出力するにはそのポートが出力ポートに設定されている必要があります。

例

1秒間隔でポート pa0 を HIGH/LOW 繰り返す。

```
{ portsetconfname_ , "pa0", "out" };
```

```
.while 1 .do
```

```
    { portset_ , "pa0" };
```

```
    { msleep_ , 1000 };
```

```
    { portclr_ , "pa0" };
```

```
    { msleep_ , 1000 };
```

```
.endwhile
```

```
.exit
```

```
.end
```

13.5 組み込み関数 portgetconf_

{ portgetconf_, ポート名 };

ポート名で指定するポートの設定状態を取得します。

戻り値は

- 0 入力
- 1 内部プルアップ入力
- 2 出力
- 3 オープンドレインタイプの出力

となります。設定は関数 *portsetconfname_* で設定した情報です。

例

ポート *p40* の設定情報を読み出して変数 *p40conf* に保存します。

```
p40conf = { portgetconf_, "p40" };
```

13.6 組み込み関数 `portgetconfname_`

`{ portgetconfname_ , ポート名 }`;

ポート名で指定するポートの設定状態を名前で取得します。
戻り値は次の状態を表す文字列のあるアドレスです。

<code>in</code>	入力
<code>pullup</code>	内部プルアップ入力
<code>out</code>	出力
<code>open</code>	オープンドレインタイプ出力

となります。設定は関数 `portsetconfname_` で設定した情報です。

例

ポート `p40` の設定情報を読み出し、その状態文字列のアドレスを変数 `p40conf` に保存します。

```
p40conf = { portgetconfname_ , "p40" };
```

13.7 組み込み関数 `portsetconfname_`

`{ portsetconfname_ , ポート名, 状態名 };`

ポート名で指定するポートの設定状態を状態名で指定する状態に設定します。

戻り値は

"in"	入力
"pullup"	内部プルアップ入力
"out"	出力
"open"	オープンドレインタイプの出力

となります。

例

ポート `p40` の設定を出力に設定する。

```
{ portsetconfname_ , "p40", "out" };
```

13.8 組み込み関数 `portsaveconf_`

```
{portsaveconf_};
```

ポートの設定状態を ROM のパラメータ領域に保存します。

保存パラメータのキーワードは IOCONFIG です。

この関数を呼ばないと設定変更したポート設定が ROM に保存されませんので、電源断によって設定が失われます。

戻り値は常に 0 です。

14. 組み込み関数－アナログ入出力

ConDuLan 内蔵の AD および DA 変換をおこなうための組み込み関数が3本用意されています。

<i>da_</i>	DA 出力をおこなう
<i>ad_</i>	チャンネルを指定して AD 入力をおこなう
<i>adall_</i>	すべてのチャンネルの AD 入力をおこなう

DA 出力ピンはチャンネル 0,1 それぞれ CN1-7, CN1-10 です。また AD 入力ピンはチャンネル 0 から 7 がそれぞれ CN1-20, CN1-22, CN1-24, CN1-6, CN1-5, CN1-8, CN1-7, CN1-10 です。これらはデジタル入力も兼用になっています。

DA 出力ピンは入力ピンと兼用になっています。電源投入後は入力ピンとして使用されますが、組み込み関数 *da_* が呼ばれるとそのピンは電源断まで DA 出力ピンとして動作します。

14.1 組み込み関数 da_

```
{ da_, チャンネル, 値 };
```

```
{ da_, チャンネル };
```

内蔵 DA の指定チャンネルへ値を出力します。チャンネルは 0 か 1 です。値は 0 から 255 までです。この関数を呼ぶと指定チャンネルの DA がイネーブルとなります。それまでは入力ポートとして動作します。

戻り値は出力値です。戻り値が-1 の場合はエラーです。

値を指定しない呼び出しは現在の出力値を取得するために使用します。この場合は出力値の変更はありません。

例

DA チャンネル 0 に 255 を出力する。

```
{ da_, 0, 255 };
```

14.2 組み込み関数 `ad_`

`{ad_, チャンネル};`

内蔵 AD の指定チャンネルを読み出します。チャンネルは 0 から 7 です。戻り値が読み出したデータで 0 から 1023 の範囲です。

例

AD0 の値を読んで変数 `_ad0` へ保存する。

```
_ad0 = {ad_, 0};
```

14.3 組み込み関数 `adall_`

`{ adall_, 保存アドレス0, 保存アドレス1, 保存アドレス2, ... };`

内蔵 AD のチャンネル 0 から指定された数のチャンネルを順に読みます。読み出した値は指定アドレスへ 32 ビット整数として保存されます。

例

AD0-AD5 を読んで変数 `_ad0` から `_ad5` へ保存する。

`{ adall_, _ad0, _ad1, _ad2, _ad3, _ad4, _ad5 };`

15. 組み込み関数—I²C

ConDuLan にはデジタル入出力ピンを利用した I²C 制御機能が 2 チャンネル分用意されています。使用ピン番号は下の表のようになっています。

チャンネル	I ² C 信号名	ConDuLan 信号名	コネクタ
主 (primary)	SDA	PA0	CN2-5
	SCL	PA1	CN2-8
副 (secondary)	SDA	PA3	CN2-10
	SCL	PA4	CN2-12

プルアップ抵抗と I²C デバイスをそれぞれのチャンネルの 2 本の端子に接続すればハードウェアの準備は終了です。

iomacro には 10 本の I²C 用組み込み関数が用意されています。

主チャンネル用関数

<i>i2creset_</i>	I ² C を使用する場合最初に一度だけ使用するリセット関数
<i>i2crw_</i>	指定したデータを I ² C デバイスに書き込んでから読み出す。
<i>i2c_</i>	I ² C から読み出したり I ² C へ書き込んだりする。
<i>i2crws_</i>	指定した 1 バイトを I ² C デバイスに書き込んでから読み出す。
<i>i2crwd_</i>	指定した 2 バイトを I ² C デバイスに書き込んでから読み出す。

副チャンネル用関数

<i>i2c2reset_</i>	I ² C を使用する場合最初に一度だけ使用するリセット関数
<i>i2c2rw_</i>	指定したデータを I ² C デバイスに書き込んでから読み出す。
<i>i2c2_</i>	I ² C から読み出したり I ² C へ書き込んだりする。
<i>i2c2rws_</i>	指定した 1 バイトを I ² C デバイスに書き込んでから読み出す。
<i>i2c2rwd_</i>	指定した 2 バイトを I ² C デバイスに書き込んでから読み出す。

15.1 組み込み関数 `i2creset_`, `i2c2reset_`

```
{ i2creset_ };  
{ i2c2reset_ };
```

PC デバイスを使用する場合最初に一度だけこの関数を呼びます。通常 `iomacro` メインで呼びます。

`i2creset_` は主チャンネル用で、`i2c2reset_` は副チャンネル用です。

15.2 組み込み関数 `i2crw_`, `i2c2rw_`

```
{ i2crw_, PC アドレス, データバッファアドレス, データサイズ, 個別データ, ... };  
{ i2c2rw_, PC アドレス, データバッファアドレス, データサイズ, 個別データ, ... };
```

PC デバイスにデータを書き込んでから読み出します。書き込むデータも読み出すデータも複数バイト可能です。書き込みと読み出しの間に他のタスクが PC アクセスをすることはありません。*PC アドレス*は PC デバイスのアドレスです。32ビット整数中下位 8ビットが使用されます。下位 8ビットの中で上位 7ビットがアドレスです。

*PC アドレス*の最下位ビットが 0 の場合は データバッファアドレスの内容が PC デバイスに書き込まれてから個別データで指定されている変数にデータが読み出されます。

*PC アドレス*の最下位ビットが 1 の時は個別データが順次 PC デバイスに書き込まれてから データバッファアドレスに PC の読み出しデータを保存します。

*データサイズ*は データバッファアドレスのバイト数です。 *個別データ*はアドレス最下位ビットが 0 のときは変数のアドレスです。アドレス最下位ビットが 1 の時は、 *個別データ*は数値です。

戻り値は読み出したデータのバイト数です。

データバッファのデータはバイトデータです。個別データは 32ビット整数です(下位 8ビットだけ使用します)。

`i2crw_` は主チャンネル用で、`i2c2rw_` は副チャンネル用です。

15.3 組み込み関数 `i2c_`, `i2c2_`

```
{ i2c_, PC アドレス, 個別データ, 個別データ, ... };  
{ i2c2_, PC アドレス, 個別データ, 個別データ, ... };
```

PC デバイスにデータを書き込んだり読み出したりします。

PC アドレスは PC のデバイスアドレスで、最下位ビットが 0 の時は個別データを順に書き込んでいきます。最下位ビットが 1 の時は PC デバイスを読み出し、順に個別データで指定されたアドレスに保存します。書き込み時の個別データは 32 ビット整数ですが下位 8 ビットだけが書き込まれます。読み出し時には読み出した 8 ビットデータを 32 ビット整数として個別データ領域に書き込みます。

`i2c_` は主チャンネル用で、`i2c2_` は副チャンネル用です。

例

主チャンネルのアドレス `0xa0` のデバイスにデータ `0,1,2,3` を順に書き込みます。

```
{ i2c_, 0xa0, 0, 1, 2, 3 };
```

主チャンネルのアドレス `0xa0` のデバイスから 4 バイト読み出し、順に変数 `u,v,w,x` に格納します。

```
{ i2c_, 0xa1, u, v, w, x };
```

15.4 組み込み関数 `i2crws_`, `i2c2rws_`

```
{ i2crws_, PC アドレス, 書き込みデータ, 読み出しデータアドレス, ... };  
{ i2c2rws_, PC アドレス, 書き込みデータ, 読み出しデータアドレス, ... };
```

PC デバイスに書き込みデータの下位 1 バイト書き込んでから読み出します。

読み出しデータはそれぞれ順に読み出しデータアドレスで指定された領域に 32 ビット整数として格納されます。

例

主チャンネルのアドレス `0xa0` のデバイスにデータ `0` を書き込んでから 4 バイト読み出し、順に変数 `u,v,w,x` に格納します。

```
{ i2crws_, 0xa0, 0, u, v, w, x };
```

15.5 組み込み関数 `i2crwd_`, `i2c2rwd_`

```
{ i2crwd_, PC アドレス, 書き込みデータ1, 書き込みデータ2, 読み出しデータアドレス, ... };  
{ i2c2rwd_, PC アドレス, 書き込みデータ1, 書き込みデータ2, 読み出しデータアドレス, ... };
```

PC デバイスに書き込みデータ 1 と 2 の下位 1 バイトずつ計 2 バイトを書き込んでから読み出します。

読み出しデータはそれぞれ順に読み出しデータアドレスで指定された領域に 32 ビット整数として格納されます。

例

主チャンネルのアドレス `0xa0` のデバイスにデータ 0 と 1 を書き込んでから 4 バイト読み出し、順に変数 `u,v,w,x` に格納します。

```
{ i2crws_, 0xa0, 0, 1, u, v, w, x };
```

16. 組み込み関数—環境変数

ConDuLan には IP アドレスや入出力ポート設定などの情報をシリアル ROM に保存しておき、電源再投入後も設定情報を利用することのできる機構が用意されています。これらの ROM に保存できる情報は環境変数と呼ばれ ConDuLan 内蔵 RTOS が電源投入後 ROM から読み出します。

環境変数は名前と値から構成されていて、どちらも文字列です。

環境変数を扱う組み込み関数は次の 10 本があります。

<i>getenv_</i>	指定の環境変数の値を取得する
<i>setenv_</i>	指定の環境変数の値を設定する
<i>unsetenv_</i>	指定の環境変数を消去する
<i>getenvlong_</i>	指定の環境変数の値を整数で取得する
<i>setenvlong_</i>	指定の環境変数の値(数値)を整数で設定する
<i>getenvip_</i>	指定の環境変数の値(IP アドレスの書式)を整数で取得する
<i>setenvip_</i>	指定の環境変数の値(IP アドレスの書式)を整数で設定する
<i>saveenv_</i>	<i>setenv_</i> 実行後すべての環境変数を ROM へ書き込む
<i>saveenvlong_</i>	<i>setenvlong_</i> 実行後すべての環境変数を ROM へ書き込む
<i>saveenvip_</i>	<i>setenvip_</i> 実行後すべての環境変数を ROM へ書き込む

16.1 環境変数名

環境変数には次のような ConDuLan で使用している名前があります。

IPADDR	ConDuLan の IP アドレス
SUBNET	ConDuLan のサブネットマスク
GATEWAY	デフォルトゲートウェイの IP アドレス
NAME	ConDuLan に付けられた名前
APPLICATION	内蔵アプリケーションファームウェアの自動スタート
ENTRY	内蔵アプリケーションファームウェアの実行アドレス
HTTP	HTTP ポート番号
HTTP_IP	HTTP アクセス制限 IP アドレス
HTTP_MASK	HTTP アクセス制限サブネットマスク
HTTP_PASS	HTTP アクセス制限パスワード
HTTP_LEVEL	HTTP アクセス制限レベル
IOCONFIG	デジタル入出力設定
TCPSERVER_PASSWD	TCP 関数呼び出しのパスワード
TCPSERVER_IP	TCP 関数呼び出しの許可 IP アドレス
TCPSERVER_PASSWD	TCP 関数呼び出しの許可サブネットマスク
TELNETD_BPS	LAN-シリアルシリアル転送速度
TELNETD_ECHO_LAN	LAN-シリアル LAN 側エコーバック
TELNETD_ECHO_SER	LAN-シリアルシリアル側エコーバック
TELNETD_CR_LAN	LAN-シリアル LAN 側 CR コード
TELNETD_CR_SER	LAN-シリアルシリアル側 CR コード
TELNETD_LF_LAN	LAN-シリアル LAN 側 LF コード
TELNETD_LF_SER	LAN-シリアルシリアル側 LF コード
TELNETD_BS_LAN	LAN-シリアル LAN 側バックスペース
TELNETD_BS_SER	LAN-シリアルシリアル側バックスペース
TELNETD_USER	LAN-シリアルユーザ名
TELNETD_PASSWD	LAN-シリアルパスワード
TELNETD_LOGIN	LAN-シリアル実行許可

16.2 組み込み関数 `getenv_`

```
{ getenv_, 名前, データアドレス, データサイズ };
```

指定した名前の環境変数をデータアドレスが示す領域に書き込みます。領域のサイズはデータサイズで与えられます。

戻り値が0の場合は読み出し成功です。戻り値が-1の場合は該当環境変数がないかデータ領域サイズが小さすぎてエラーになったことを示します。

例

自身の IP アドレスを取得する。

```
ipval = { carray_, 32 };  
{ getenv_, "IPADDR", $ipval, 32 };
```

16.3 組み込み関数 `setenv_`

```
{ setenv_, 名前, データ };
```

指定した名前の環境変数をデータに設定します。

この関数では環境変数は ROM に書き込まれませんので関数 `saveenv_` を使用して ROM に書き戻す必要があります。

例

`STRING` という環境変数を `ABC` という値に設定する。

```
{ setenv_, "STRING", "ABC" };
```

16.4 組み込み関数 `unsetenv_`

```
{unsetenv_ 名前};
```

指定した名前の環境変数を消去します。

この関数では環境変数は ROM に書き込まれませんので関数 `saveenv_` を使用して消去した状態の環境変数を ROM に書き戻す必要があります。

例

`STRING` という環境変数を消去する。

```
{unsetenv_ "STRING"};
```

16.5 組み込み関数 `getenvlong_`, `getenvip_`

```
{ getenvlong_, 名前 };
```

```
{ getenvip_, 名前 };
```

組み込み関数 `getenvlong_` は指定した *名前* の環境変数を数値を表す文字列として整数に変換して戻り値とします。

組み込み関数 `getenvip_` は指定した *名前* の環境変数を IP アドレスを表す文字列として整数に変換して戻り値とします。

いずれも環境変数がない場合は 0 となります。

例

自身の HTTP のポート番号を変数 `http_port` に取得する。

自身の IP アドレスを 32 ビット整数として変数 `ip_addr` に取得する。

```
http_port = { getenvlong_, "HTTP" };
```

```
ip_addr = { getenvip_, "IPADDR" };
```

16.6 組み込み関数 `setenvlong_`, `setenvip_`

```
{ setenvlong_, 名前, 整数 };
```

```
{ setenvip_, 名前, 整数 };
```

関数 `setenvlong_` は指定した *名前* の環境変数を与えられた整数の値に設定します。

関数 `setenvip_` は指定した *名前* の環境変数を整数で与えられた IP アドレスの値に設定します。

環境変数は名前も値も文字列なので `iomacro` で使用する整数は文字列に変換する必要があります。関数 `setenvlong_` は与えられた整数を文字列に変換して設定します。また関数 `setenvip_` は与えられた整数を IP アドレスを表す `xx.xx.xx.xx` の書式の文字列にして設定します。

この関数では環境変数は ROM に書き込まれませんので関数 `saveenv_` を使用して ROM に書き戻す必要があります。

例

TEST_LONG という環境変数を 10 という値に設定する。

TEST_IP という環境変数を 192.168.98.68 という IP アドレスに設定する。

IP アドレス 192.168.98.68 は 16 進数で `0xc0a86244` です。

```
{ setenvlong_, "TEST_LONG", 10 };
```

```
{ setenvip_, "TEST_IP", 0xc0a86244 };
```

16.7 組み込み関数 `saveenv_`

```
{saveenv_ , 名前, データ};  
{saveenv_};
```

引数名前とデータがある場合には `setenv_` を実行後すべての環境変数を ROM に書き込みます。

引数なしで呼ばれた場合は `setenv_` 処理はおこなわずすべての環境変数を ROM に書き込みます。

16.8 組み込み関数 `saveenvlong_`, `saveenvip_`

```
{ saveenvlong_, 名前, データ};
```

```
{ saveenvip_, 名前, データ};
```

それぞれ `setenvlong_` と `setenvip_` を実行後すべての環境変数を ROM に書き込みます。

17. 組み込み関数—パラメータ

ConDuLan にはシステム内の各タスクが共有する変数を保存するグローバルパラメータという機構が用意されています。グローバルパラメータの変数は32ビット整数で、IDを指定して読み出したり書き込んだりします。ConDuLan 内蔵 RTOS があらかじめ使用する ID は負の値で-1 から -63 まで、そのほか ConDuLan アプリケーションが使用する予定の ID は 0 から 15 までです。iomacro が自由に使用できる ID は 16 から 255 までです。

グローバルパラメータは環境変数とは異なり ROM に保存されません。またアクセスは整数を扱いますので環境変数に比較して高速です。

グローバルパラメータを扱う関数には次の7本があります。

<i>gpsetl_</i>	グローバルパラメータを設定する
<i>gpgetl_</i>	グローバルパラメータ取得する
<i>gpsetipaddr_</i>	グローバルパラメータ中の ConDuLan の IP アドレスを設定する
<i>gpgetipaddr_</i>	グローバルパラメータ中の ConDuLan の IP アドレスを取得する
<i>gpsetdt_</i>	グローバルパラメータ中の現在時刻を設定する
<i>gpclrdt_</i>	グローバルパラメータ中の現在時刻を消去する
<i>gpsetdt_</i>	グローバルパラメータ中の現在時刻を読み出す

17.1 グローバルパラメータ予約 ID

ConDuLan が使用するグローバルパラメータには次のような予約 ID があります。

- 1 内部ファイルシステムがレディになった(通常 iomacro では使用しません)
- 2 アプリケーションファームウェア実行許可
- 3 アプリケーションファームウェア実行開始アドレス
- 4 ConDuLan に設定されているサブネットマスク
- 5 ConDuLan に設定されている IP アドレス
- 6 ConDuLan に設定されているゲートウェイアドレス
- 7 内蔵 RTOS のバージョン(文字列へのアドレス)
- 8 Web 待ち受けポート番号
- 9 アプリケーション使用可能 RAM アドレス
- 10 現在時刻(時計内蔵の場合)
- 11 現在日付(時計内蔵の場合)

グローバルパラメータの多くは32ビット整数で関数 `gpsetl_` や `gpgetl_` でアクセスします。

IP アドレスなど一部環境変数と重複する内容があります。処理の高速化のため環境変数の値をグローバルパラメータにコピーして使用しています。必要な変数はグローバルパラメータを変更すると環境変数も書き換えられ、ROM に保存されます。

17.2 組み込み関数 `gpsetl_`

```
{ gpsetl_, ID, 値 };
```

ID で指定するグローバルパラメータに値で指定する 32 ビット整数をセットします.

例

IP アドレスを 192.168.98.66 (32 ビット整数では 0xc0a86242) に設定します.

```
{ gpsetl_, -5, 0xc0a86242 };
```

17.3 組み込み関数 `gpgetl_`

```
{ gpgetl_, ID };
```

ID で指定するグローバルパラメータを取得します。戻り値が値です。

例

IPアドレスを取得して変数 *ip* に保存します。

```
ip = { gpgetl_, -5 };
```

17.4 組み込み関数 `gpsetipaddr_`

```
{ gpsetipaddr_, IP アドレス };
```

ConDuLan のIPアドレスを指定の *IP アドレス* に設定します。

例

IP アドレスを 192.168.98.66 (32ビット整数では 0xc0a86242) に設定します。

```
{ gpsetipaddr_, 0xc0a86242 };
```

これは

```
{ gpsetl_, -5, 0xc0a86242 };
```

と同じです。

17.5 組み込み関数 `gpgetipaddr_`

```
{ gpgetipaddr_ };
```

IP アドレスを取得します。戻り値が値です。

例

IP アドレスを取得して変数 `ip` に保存します。

```
ip = { gpgetipaddr_ };
```

これは

```
ip = { gpgetl_ -5 };
```

と同じです。

17.6 ConDuLan の日付と時刻

ConDuLan は時計を内蔵していないため時刻を扱う処理はできません。しかし追加ボードなどで RTC を内蔵するなどして時刻を知る手段が用意されている場合はアプリケーションから1秒ごとに時刻を更新することで RTOS とタスクが日付と時刻を取得することができます。ConDuLan で扱う日付および時刻情報は 2 個の 32 ビット整数で管理され、日付情報はグローバルパラメータ-11 に、時刻情報は-10 に保存されます。データの書式は BCD で日付情報の最上位バイトが西暦年号の上2桁、2番目のバイトが西暦年号の下2桁、3番目のバイトが月、最下位のバイトが日を現します。たとえば 2006 年 12 月 9 日は

0x20061209

となります。

日付情報がすべて 0 の場合は時計を内蔵していないことを示しています。

同様に時刻は最上位バイトは使用せず(曜日として使用する場合があります)、2番目のバイトが時、3番目のバイトが分、最下位バイトが秒です。たとえば 14 時 6 分 52 秒は

0x00140652

となります。

17.7 組み込み関数 `gpsetdt_`

```
{ gpsetdt_, 日付, 時刻 };
```

日付と時刻を設定します。

グローバルパラメータの-10と-11が書き換えられます。

例

現在時刻を2006年12月9日14時6分52秒に設定します。

```
{ gpsetdt_, 0x20061209, 0x00140652 };
```

17.8 組み込み関数 `gpgetdt_`

`{ gpgetdt_ , 時刻の保存アドレス };`

日付と時刻を取得します。戻り値は日付で、時刻は時刻の保存アドレスで示す領域に32ビット整数として保存されます。

例

現在時刻を取得します。 `_date` には日付が、 `_time` には時刻が保存されます。

```
_date = { gpgetdt_ , _time };
```

17.9 組み込み関数 `gpclrdt_`

```
{ gpclrdt_};
```

日付と時刻をクリアします。日付がクリアされた状態だと時計を内蔵していません。

18. 組み込み関数—データログ

ConDuLan には AD コンバータやそのほか外部より入力されたさまざまなデータをバッファに蓄積し、PC から Web ブラウザを利用してそのデータを取得するログ機構が用意されています。

また、ConDuLan の I²C プライマリチャンネルにシリアル ROM を追加実装した場合には、電源断に備えてログデータを ROM に保存する機能も用意されています。

次の 10 本の組み込み関数をデータログで使用します。

<i>logalloc_</i>	ログ用データバッファを確保する
<i>logbackup_</i>	ログデータを ROM 保存するかどうか設定する
<i>logclear_</i>	保存しているログデータを消去する
<i>logput_</i>	ログデータを追加する
<i>logprint_</i>	ログデータを書式指定で追加する
<i>logselsend_</i>	範囲を指定してログレコードを GET 応答バッファへコピーする
<i>logbackupinfo_</i>	ログ用 ROM 管理情報を取得する
<i>logearliest_</i>	最も古いログデータを取得する
<i>loglatest_</i>	最も新しいログデータを取得する

18.1 ログデータ送信

ConDuLan ではログデータは文字列データを想定しています。

PC が Web ブラウザを使用して ConDuLan のログデータを取得するには iomacro で書かれた CGI を要求します。

iomacro で書かれたログ応答用の CGI は関数 *logsendsend_* を利用してログデータを PC へ送ります。

このとき CGI 名の拡張子の種類によって ConDuLan 内蔵 Web サーバがログデータのファイルタイプを決定します。

ログデータの応答に使用されるような拡張子とファイルタイプの関係を以下に示します。

<i>tsv</i>	<i>text/tab-separated-values</i>
<i>csv</i>	<i>text/csv</i>
<i>xls</i>	<i>application/vnd.ms-excel</i>

これらのファイルタイプによって多くの Web ブラウザが該当するプラグインを実行します。

たとえば CGI 名が *xls* となっていると ConDuLan が応答するファイルタイプは *application/vnd.ms-excel* になり、PC ではマイクロソフトエクセルプラグインが起動します。区切りコードが TAB であれば各要素はそのままエクセルのセルに格納されます。

18.2 ログ保存 ROM

ConDuLan のプライマリ I²C ポートにシリアル ROM を追加するとログデータを RAM だけでなく ROM にも保存することができます。長期間ログを保持する場合電源が遮断しても電源投入後に ROM に保存しておいたログデータを RAM に回復することができます。

使用する ROM は Atmel 製 AT24C1024(1Mb=128KB)です。A1 を HIGH にしてデバイスアドレスを 0xA4 とします。チップ内部でアドレス 0xA5 も受け付けます。

他の容量の ROM でも動作しますが、かならずアドレス 0xA4 に設定してください。

iomacro では ROM 容量を指定しますので実装 ROM の容量以下の値で指定してください。

シリアル ROM は書き込み回数に制限がありますので、高速にログデータ保存する場合は ROM バックアップをしないで下さい。通常1時間に1回以下の低速データサンプリングで使用してください。

18.3 ログデータバッファ

ログデータは長期にわたって収集されることがあるため保存領域のサイズは大きくなりがちです。そのためログバッファサイズは固定ではなく関数 `logalloc_` を使用して `iomacro` が指定するサイズのログバッファを確保します。ログデータを追加するには関数 `logput_` か `logprint_` を使用します。それぞれ1レコードごとにレコード番号が付加されていきます。各レコードは可変サイズですが、255 バイトが最大サイズとなります。

Web サーバに GET 用 CGI として登録した `iomacro` 関数が `logselsend_` を呼ぶと Web の応答バッファにログデータをコピーして PC へデータを送ります。ログデータを PC に送る CGI は応答バッファがログデータサイズより大きくないといけないので CGI 登録時に必要なバッファサイズを登録します。

ログデータがログバッファの容量を超えた場合はバッファ先頭に戻って古いデータを上書きします。

18.4 組み込み関数 `logalloc_`

`{ logalloc_ , ログバッファサイズ, 書き込みマージン, ROM サイズ};`

ログデータバッファを RAM に確保します。データログをとる場合はこの関数を呼んであらかじめログバッファを確保しておきます。通常電源投入後にログバッファを確保するよう `iomacro` のメインで呼び出します。

バッファサイズはログバッファサイズで指定します。最大 128KB(0x20000=131072)です。

書き込みマージンはログバッファデータを PC へ送信中に新しいログデータを追加できるように指定するための書き込みポインタと読み出しポインタの間のメモリサイズです。マージン指定値が 0 の場合は 64 が使用されます。マージンはレコードサイズに依存しますが、最低1レコード分は必要となります。

ROM サイズはログデータを ROM にもバックアップする場合に ROM の容量を指定します。ROM サイズが指定された場合はバックアップ ROM から保存データを RAM のログバッファに読み込みます。

戻り値はログバッファの制御情報を格納している領域のアドレスとなります。このアドレスは以後ログ操作をおこなうときに使用します。

例

バッファサイズ 128KB(131072)のログバッファを確保します。マージンは 512 バイト, ROM 保存は有効で ROM 容量は 128KB(131072)です。

ログバッファの制御情報アドレスは `log_ctl` に保存します。

```
log_ctl = { logalloc_ , 131072, 512, 131072};
```

18.5 組み込み関数 `logbackup_`

`{ logbackup_ , ログのROM バックアップ };`

ログデータをバックアップ用 ROM にも保存するかどうかを指定します。

ログの ROM バックアップの値が 0 の場合は ROM へのバックアップ保存を実行しません。0 以外の値の場合は関数 `logalloc_` で ROM サイズを指定してある場合にログデータの ROM 保存をおこないます。

ConDuLan の初期状態は「バックアップしない」ですので、ログデータを ROM にバックアップ保存する場合はこの関数を呼んでください。

例

ログデータの ROM へのバックアップ保存を有効にします。

```
{ logbackup_ , 1 };
```

18.6 組み込み関数 `logclear_`

`{ logclear_ , ログ制御情報アドレス };`

ログデータを消去します。ログバッファ確保時にバックアップ ROM が指定されている場合は ROM 内データも消去します。

ログ制御情報アドレスは関数 `logalloc_` でログバッファを確保したときの戻り値です。

例

バッファサイズ 128KB(131072), マージン 512 バイト, ROM 容量 128KB(131072)のログバッファを確保し, ログデータをクリアします。通常は, この例とは異なり, ログバッファ確保直後にデータをクリアすることはありません。

```
log_ctl = { logalloc_ , 131072 , 512 , 131072 };
```

```
{ logclear_ , $log_ctl };
```

18.7 組み込み関数 `logput_`

`{ logput_ , ログ制御情報アドレス, ログレコードアドレス };`

ログデータにログレコード1件追加します。

ログ制御情報アドレスはログバッファを確保したときの関数 `logalloc_` の戻り値です。

ログレコードアドレス は追加するレコードのアドレスです。通常ログレコードは印刷可能な文字列で `nil(0x00)` で終了します。最後の文字は通常改行(`0x0A`)にします。

レコード追加でログバッファがいっぱいになったら古いレコード領域に上書きします。

例

ログバッファを確保し、およそ1秒ごとに `AD0` の値を記録していきます。

```
log_ctl = { logalloc_ , 131072, 512, 0 };
log_record = { carray_ , 32 };
.while 1 .do
    { print_ , $log_record, "%ld\r\n", { ad_ , 0 } };
    { logput_ , $ log_ctl, $log_record };
    { msleep_ , 1000 };
.endwhile
.end
```

18.8 組み込み関数 `logprint_`

`{ logprint_ , ログ制御情報アドレス, 書式, ... };`

ログデータにログレコード1件追加します。

ログ制御情報アドレスはログバッファを確保したときの関数 `logalloc_` の戻り値です。

書式は関数 `co_` や `print_` と同様 C の `printf` の書式に準拠します。

ログレコードアドレスは追加するレコードのアドレスです。通常ログレコードは印刷可能な文字列で最後の文字は改行(0x0A)にします。

レコード追加でログバッファがいっぱいになったら古いレコード領域に上書きします。

例

ログバッファを確保し、およそ1秒ごとにAD0の値を記録していきます。

```
log_ctl = { logalloc_ , 131072, 512, 0 };  
.while 1 .do  
    { logprint_ , $ log_ctl, "%ld\r\n", { ad_ , 0 } };  
    { msleep_ , 1000 };  
.endwhile  
.end
```

18.9 組み込み関数 logbackupinfo_

{ logbackupinfo_ , 先頭レコード位置格納領域, 最終レコード位置格納領域 };

ログデータのバックアップ用 ROM に保存されている先頭レコード位置と最終レコード位置を取得します。

戻り値は 0 でレコード位置取得成功です。1 の場合は ROM 内制御情報が不正な値です。-1 の場合は ROM が実装されていない場合です。

この関数は通常、関数 *logalloc_* でログバッファを確保する前に呼び出します。

主に電源投入後のログデータバックアップ状態を調べるために用意されています。

戻り値が 0 で先頭レコード位置と最終レコード位置が異なる値の場合 ROM にログデータのバックアップがあります。

例

次の例はバックアップ ROM にログデータのバックアップがある場合シリアルポートへ Backup data exist. の表示をおこないます。

```
.if 0 == { logbackupinfo_ , _top, _vacant } .then
    .if $ _top != $ _vacant .then
        { conl_ , "Backup data exist.\r\n" };
    .endif
.endif
.end
```

18.10 組み込み関数 `logearliest_`, `loglatest_`

```
{ logearliest_, ログ制御情報アドレス, 先頭レコード格納領域, 先頭レコード格納領域サイズ };  
{ loglatest_, ログ制御情報アドレス, 最終レコード格納領域, 最終レコード格納領域サイズ };
```

関数 `logearliest_` はログデータの中で最も古いレコードを `先頭レコード格納領域` に格納します。戻り値は格納レコードのサイズです。

関数 `loglatest_` はログデータの中で最も新しいレコードを `最終レコード格納領域` に格納します。戻り値は格納レコードのサイズです。

ログバッファにレコードがない場合は戻り値は 0 になり、`レコード格納領域` には終了コード (0x00) が書き込まれます。

この関数はログバッファ内のデータ状況を把握するために用意されています。ログデータの先頭情報をシーケンス番号にしたり、日付時刻 (RTC 実装時など) にすると容易にログデータの状況を知ることができます。

18.11 組み込み関数 `logselsend_`

```
{logselsend_ , ログ制御情報アドレス,  
    開始レコード,  
    終了レコード,  
    省略レコード数,  
    タイトル};
```

CGIとして使用されている `iomacro` 関数がこの関数を使用するとログデータを PC への応答データとして用意します。

ログデータの先頭文字列にシーケンス番号や日付時刻(時計内蔵の場合)などを付加しておけば、ログデータの中から応答データを選択的に PC への応答データとすることができます。

ログ制御情報アドレスはログバッファを確保したときの `logarray_` 関数の戻り値です。

開始レコードは文字列アドレスで、指定文字列とログデータのレコード文字列を比較して指定文字列と一致するかそれより大きなコードで構成されている場合に応答データとして扱われます。ログレコードを選択的に返信する場合に使用します。ログデータの先頭レコードから応答する場合は 0 を指定します。

終了レコードは文字列アドレスで、指定文字列とログデータのレコード文字列を比較して指定文字列と一致するかそれより大きなコードで構成されている場合には応答データとして扱われ内容にします。ログレコードを選択的に返信する場合に使用します。ログデータの最終レコードまで応答する場合は 0 を指定します。

省略レコード数はログレコードのうち先頭から指定数の項目を除外して応答します。項目の区切りはスペース、コンマ、タブのいずれかです。たとえば、内部管理の目的で各レコードの先頭にシーケンス番号を付加している場合 PC への応答データでは番号を除外することができます。省略項目がない場合は 0 を指定します。

タイトルは PC への応答データの先頭に付加する文字列です。タイトルを付加しない場合は 0 を指定します。

19. 組み込み関数—配列

すでに `iomacro` 文法で説明しましたが, `iomacro` で配列を扱うには組み込み関数を使用します. 配列用組み込み関数は次の6本があります.

<code>array_</code>	グローバル変数領域に 32 ビット整数の配列を確保する
<code>sarray_</code>	グローバル変数領域に 16 ビット整数の配列を確保する
<code>carray_</code>	グローバル変数領域に 8 ビット整数の配列を確保する
<code>arraylocal_</code>	ローカル変数領域に 32 ビット整数の配列を確保する
<code>sarraylocal_</code>	ローカル変数領域に 16 ビット整数の配列を確保する
<code>carraylocal_</code>	ローカル変数領域に 8 ビット整数の配列を確保する

グローバル変数の配列はデータを保持し続けます. また `iomacro` のどの関数からも参照できます. 一方ローカル変数の配列は確保した関数が実行を終了すると領域を開放します. データは関数終了後利用できません.

配列概念の詳しい説明は2.19から2.26を参照ください.

19.1 整数配列確保関数(1次元) array_

`{ array_, 配列要素数 };`

配列要素数で指定した数の 32 ビット整数の配列領域を確保し、その先頭アドレスを戻り値とします。

配列の要素へのアクセスは[要素番号]を使用します。

例

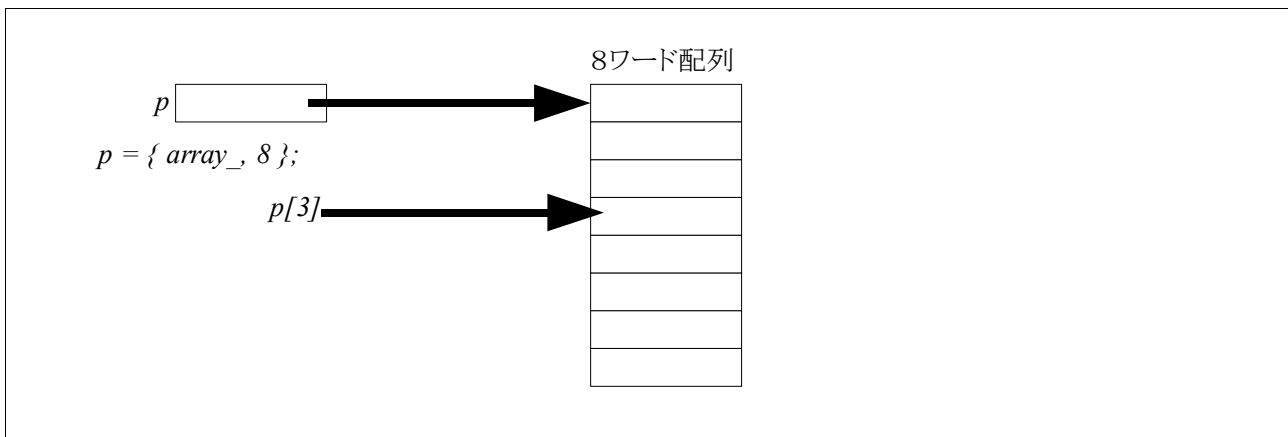
8 個の 32 ビット整数領域を確保し、そのアドレスを p に保存します。続いて配列中の要素 3 (0 から開始なので 4 番目の要素) に 1 を格納し、その値を再び読み出して x に格納します。

$p = \{ array_, 8 \};$

$p[3] = 1;$

$x = \$p[3];$

この例では配列は次のように確保されます。



19.2 整数配列確保関数(2次元) array_

```
{ array_, 配列数, 配列要素数 };
```

配列数で指定した数の 32 ビット整数配列と配列数 x 配列要素数の 32 ビット領域を確保します。最初に確保した領域の各要素には各配列のアドレスが書き込まれます。戻り値はその各配列アドレスが書き込まれた領域の先頭アドレスです。

配列の要素へのアクセスは[配列番号][要素番号]を使用します。

例

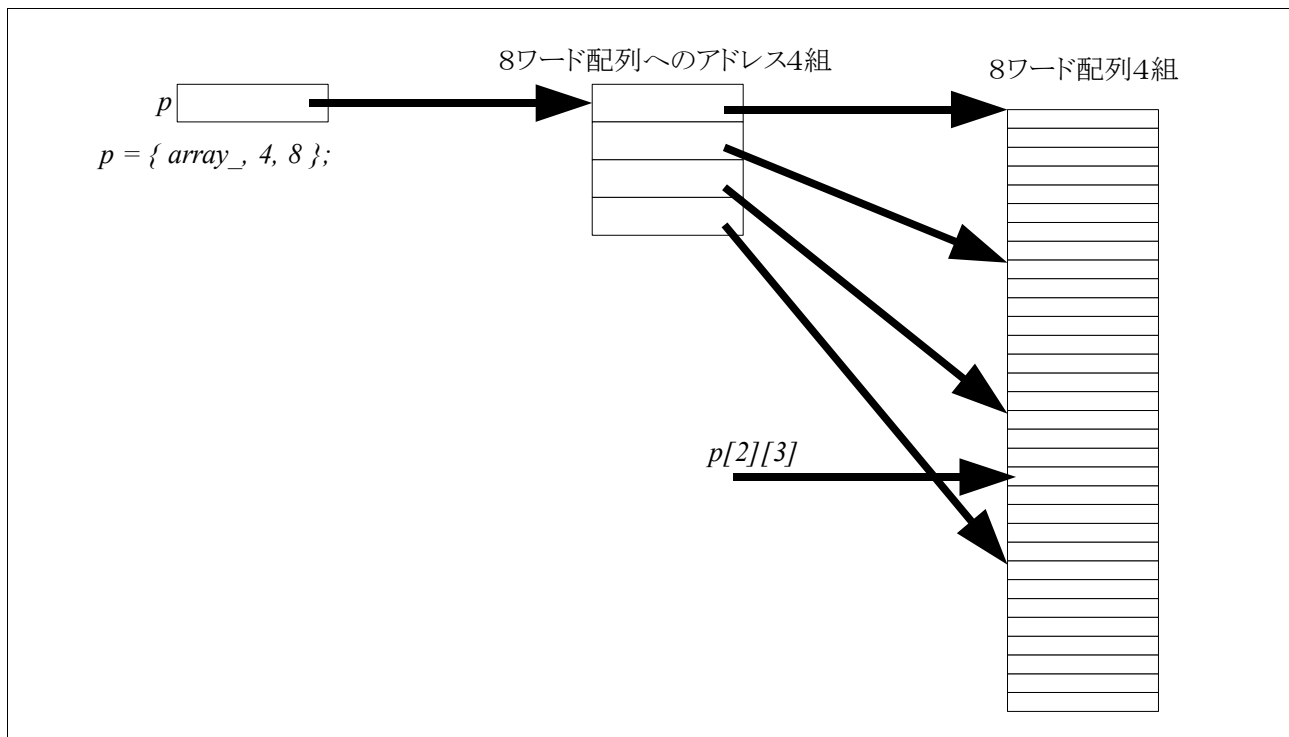
8 個の 32 ビット整数配列領域を 4 組確保し、そのアドレス保存の先頭を p に保存します。続いて配列 2 番 (3 番目の配列) の要素 3 に 1 を格納し、その値を再び読み出して x に格納します。

```
 $p = \{ array\_ , 4, 8 \};$ 
```

```
 $p[2][3] = 1;$ 
```

```
 $x = \$p[2][3];$ 
```

この例では配列は次のように確保されます。



19.3 整数配列確保関数(多次元) array_

$\{ array_ , 配列数, 配列数, \dots, 配列要素数 \};$

多次元配列も2次元配列同様の手順で領域を確保します。相違点は配列数の引数が1つではなく複数ある点です。各配列はその上の層のアドレス配列にリンクされます。戻り値は最上位アドレス配列の先頭アドレスとなります。

配列の要素へのアクセスは[配列番号][配列番号]...[要素番号]を使用します。

例

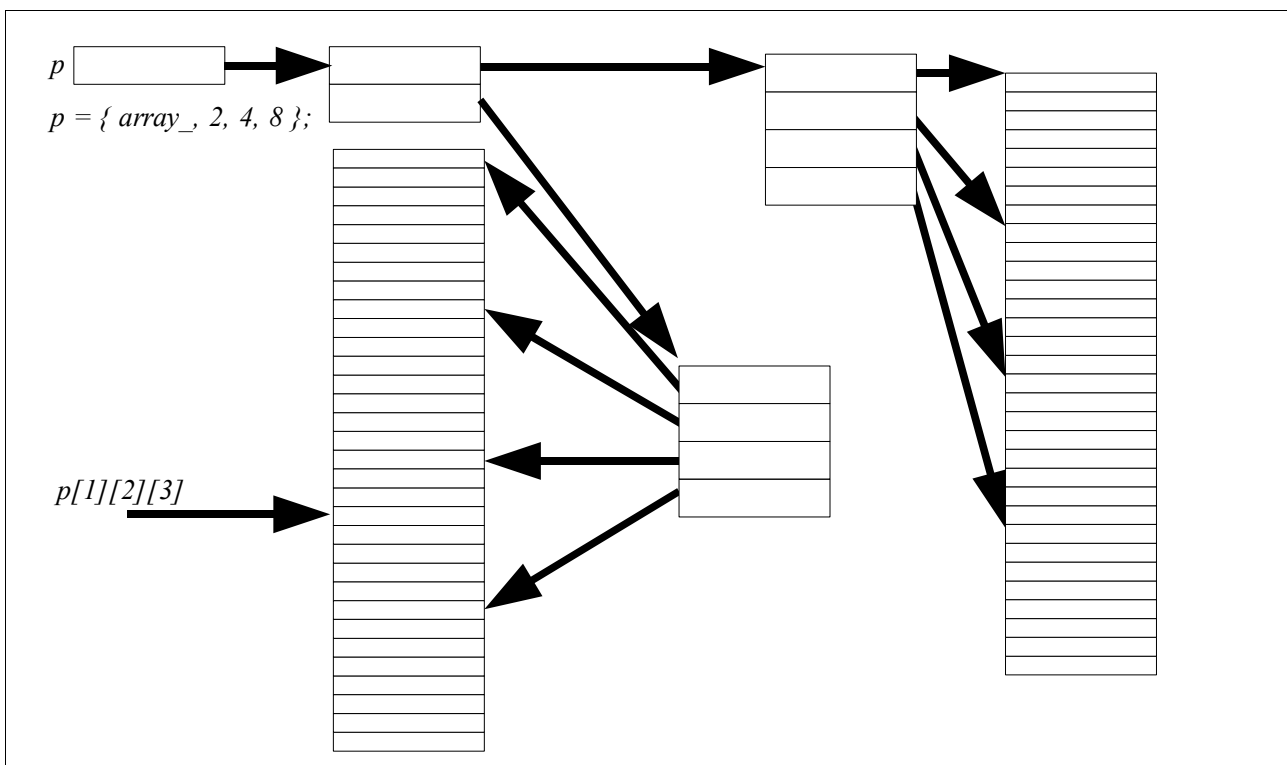
8個の32ビット整数配列領域4組を2組確保し、その先頭アドレスを p に保存します。続いて先頭配列1(2番目)の組の配列2の要素3に1を格納し、その値を再び読み出して x に格納します。

```
 $p = \{ array\_ , 2, 4, 8 \};$ 
```

```
 $p[1][2][3] = 1;$ 
```

```
 $x = \$p[1][2][3];$ 
```

この例では配列は次のように確保されます。



19.4 16ビット整数配列確保関数 `sarray_`

```
{ sarray_, 配列要素数 };
{ sarray_, 配列数, 配列要素数 };
{ sarray_, 配列数, 配列数, ..., 配列要素数 };
```

関数 `sarray_` は 16ビット整数の配列を確保します。機能は、領域サイズが 16ビットである点を除けば関数 `array_` と同じです。多次元配列の場合のアドレス配列は 32ビットサイズになります。配列の要素へのアクセスは `[::要素番号]` を使用します。

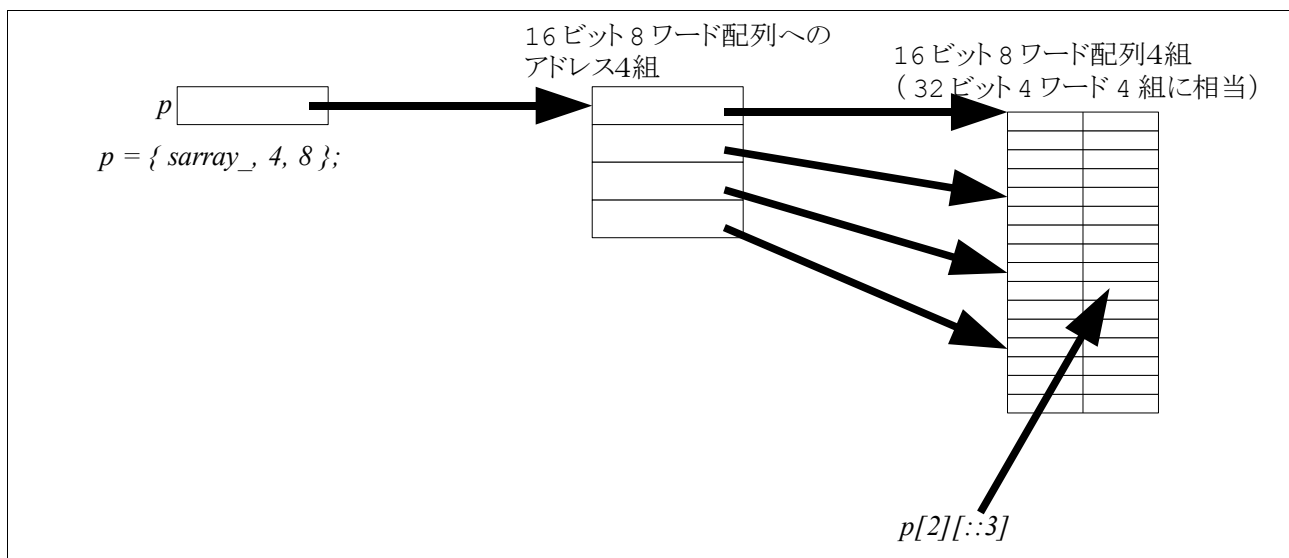
2次元配列の要素へのアクセスは `[配列番号][::要素番号]` を使用します。

例

8個の 16ビット整数配列領域を 4組確保し、そのアドレス保存の先頭を `p` に保存します。続いて配列2番(3番目の配列)の要素3に1を格納し、その値を再び読み出して `x` に格納します。

```
p = { sarray_, 4, 8 };
p[2][::3] = 1;
x = $::p[2][::3];
```

この例では配列は次のように確保されます。



19.5 文字配列確保関数 `carray_`

```
{ carray_, 配列要素数 };
{ carray_, 配列数, 配列要素数 };
{ carray_, 配列数, 配列数, ..., 配列要素数 };
```

関数 `carray_` は文字配列を確保します。機能は、領域サイズが 8 ビットである点を除けば関数 `array_` と同じです。多次元配列の場合のアドレス配列は 32 ビットサイズになります。

配列の要素へのアクセスは `[:要素番号]` を使用します。

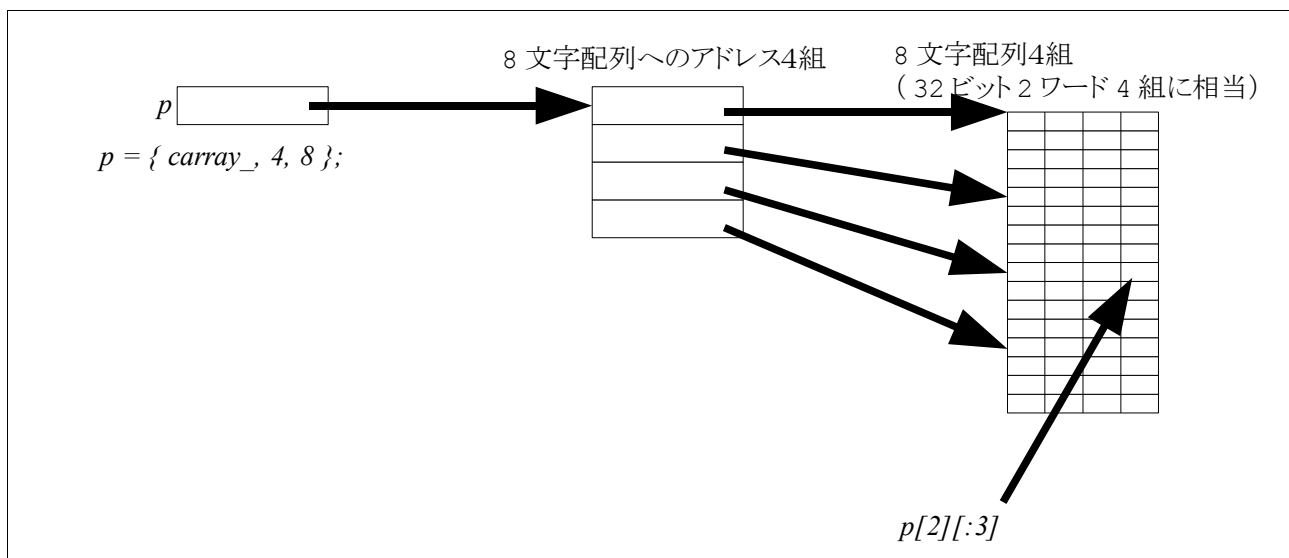
2次元配列の要素へのアクセスは `[配列番号][:要素番号]` を使用します。

例

8 文字分の文字配列領域を 4 組確保し、そのアドレス保存の先頭を `p` に保存します。続いて配列 2 番 (3 番目の配列) の要素 3 に文字 'A' を格納し、その値を再び読み出して `x` に格納します。

```
p = { carray_, 4, 8 };
p[2][:3] = 'A';
x = $:p[2][:3];
```

この例では配列は次のように確保されます。



19.6 ローカル整数配列確保関数 `arraylocal_`

```
{ arraylocal_, 配列要素数 };  
{ arraylocal_, 配列数, 配列要素数 };  
{ arraylocal_, 配列数, 配列数, ..., 配列要素数 };
```

関数 `arraylocal_` は確保する領域が関数スタック内である点を除けば関数 `array_` と同じです。関数が終了するとこの領域は自動的に開放されます。

19.7 ローカル 16 ビット配列確保関数 `sarraylocal_`

`{ sarraylocal_, 配列要素数 };`

`{ sarraylocal_, 配列数, 配列要素数 };`

`{ sarraylocal_, 配列数, 配列数, ..., 配列要素数 };`

関数 `sarraylocal_` は確保する領域が関数スタック内である点を除けば関数 `sarray_` と同じです。関数が終了するとこの領域は自動的に開放されます。

19.8 ローカル文字配列確保関数 `carraylocal_`

`{ carraylocal_, 配列要素数 };`

`{ carraylocal_, 配列数, 配列要素数 };`

`{ carraylocal_, 配列数, 配列数, ..., 配列要素数 };`

関数 `carraylocal_` は確保する領域が関数スタック内である点を除けば関数 `carray_` と同じです。関数が終了するとこの領域は自動的に開放されます。

20. 組み込み関数一割り込み

iomacro で記述したタスクは指定の割り込みによって起床することができます。

割り込みに関係する iomacro 関数には次の2本があります。

<i>int_</i>	指定番号の割り込みで起床するよう自タスクを登録する
<i>ena_</i>	指定番号の割り込みを許可する

割り込みを使用するタスクは通常初期化部分で関数 *int_* を使用して自タスクを割り込み起床タスクに登録しておきます。その後関数 *ena_* を使用し指定の割り込みを許可します。

ConDulan は iomacro タスクが登録されている割り込みが発生すると、その割り込みを禁止して該当タスクを起床します。

起床したタスクは割り込み要因をクリアしてから関数 *ena_* を使用して割り込みを許可します。その後必要な処理をおこない再び休止します(タスク関数からリターンします)。

タスクが割り込み許可後、休止前にその割り込みが発生した場合は、タスク休止後すぐに起床します。ただし割り込み回数分起床するわけではありません。タスクが実行中に複数回割り込みが発生しても起床するのは一度だけです。

割り込み番号は使用しているプロセッサ H8-3069 のベクタ番号を使用します。

ConDuLan ですでに使用している割り込みもありますので、すべての割り込みを iomacro で使用できるわけではありません。

20.1 組み込み関数 `int_`

```
{ int_, 割り込み番号 };
```

指定の番号の割り込みが発生したとき現在のタスクを起床するよう登録します。

戻り値は 0 で成功. -1 でエラーです。

例

割り込み 36 が発生したとき起床するよう登録します。

```
{ int_, 36 };
```

20.2 組み込み関数 `ena_`

```
{ ena_, 割り込み番号 };
```

指定の番号の割り込みを許可します.

戻り値は 0 で成功. -1 でエラーです.

例

割り込み 36 を許可します.

```
{ ena_, 36 };
```

20.3 割り込み番号

iomacro 割り込み関数 *int_* および *ena_* で使用する割り込み番号は H8-3069 のベクタ番号です。以下の表は iomacro から使用できる割り込みです。

番号	名前	説明
12	IRQ0	CPU87 Port8-0 CN2-4
13	IRQ1	CPU88 Port8-1 CN2-3
23	ADI	関数 <i>ad_</i> を使用しないで利用する
24	IMIA0	CompareMatch/InputCapture A0-16
25	IMIB0	CompareMatch/InputCapture B0-16
26	OVI0	Overflow0-16
28	IMIA1	CompareMatch/InputCapture A1-16
29	IMIB1	CompareMatch/InputCapture B1-16
30	OVI1	Overflow1-16
32	IMIA2	CompareMatch/InputCapture A2-16
33	IMIB2	CompareMatch/InputCapture B2-16
34	OVI2	Overflow2-16
36	CMIA0	CompareMatch A0-8
37	CMIB0	CompareMatch B0-8
38	CMIA1/CMIB1	CompareMatch A1/B1-8
39	TOVI0/TOVI1	Overflow0/1-8
44	DEND0A	DMA0A
45	DEND0B	DMA0B
46	DEND1A	DMA1A

21. 組み込み関数 - LAN-シリアル

ConDuLan には LAN とシリアルポートを使用して LAN-シリアル変換をおこなう機能を内蔵しています。

ConDuLan の工場出荷状態ではこの機能は有効になっていませんが、ROM ファイルの設定で有効にすることができます。また ROM ファイルの設定以外にも iomacro から LAN-シリアル変換を有効にすることもできます。

LAN-シリアル変換は telnet 手順に準拠した TCP 接続でおこないます。この章では iomacro プログラムから LAN-シリアル変換を有効にする方法と接続中の LAN-シリアルを切断する方法を紹介します。

LAN-シリアル変換に関連する iomacro 関数には次の2本があります。

<code>telnetd_</code>	LAN-シリアル変換を有効にする
<code>resettelnetd_</code>	現在接続中の LAN-シリアル変換接続を切断する

21.1 組み込み関数 telnetd_

```
{ telnetd_ , telnet 設定 Web ページ名 };
```

ConDuLan 内蔵の LAN-シリアル変換を有効にします。ConDuLan には LAN-シリアル変換の設定用 Web ページが内蔵されています。telnet 設定 Web ページ名はその設定ページを表示するための Web ページ名を指定します。

戻り値は常に 0 です。

例

LAN-シリアル設定ページ名を/telnetd_settings.html にして LAN-シリアル変換を有効にします。

```
{ telnetd_ , "/telnetd_settings.html" };
```

これは/etc/html_telnetd という ROM ファイルに/telnetd_settings.html を記述した場合と同じ機能になります。

21.2 組み込み関数 `resettelnetd_`

```
{resettelnetd_};
```

接続中の LAN-シリアル変換を切断します。LAN-シリアル変換は接続可能本数が1本だけです。クライアントが異常終了するなどした場合にこの関数を使用して強制切断します。強制切断は LAN-シリアル設定ページからも実行できます。

戻り値は常に 0 です。

例

```
{resettelnetd_};
```

22. その他の組み込み関数

今まで説明してきました関数以外に次の2本の関数が用意されています。

<code>version_</code>	アプリケーションファームウェアのバージョンを取得する
<code>getname_</code>	ConDuLan に設定されている名前を取得する

22.1 組み込み関数 `version_`

```
{ version_};
```

現在 ConDuLan にインストールされているアプリケーションファームウェアのバージョンを取得します。戻り値はバージョン文字列のアドレスとなります。

内蔵 RTOS のバージョンはグローバルパラメータ ID-7 で取得できます。詳細は17.1を参照ください。

また RTOS やアプリケーションのバージョンを表示する SSI も用意されています。詳細は操作ガイド-ROM ファイル編を参照ください。

22.2 組み込み関数 `getname_`

```
{ getname_ };
```

ConDuLan に設定されている名前を取得します。名前の設定は通常 LAN 設定の Web メニューで行います。戻り値は設定されている名前のあるアドレスです。

例

```
{ getname_ };
```

23. 液晶キャラクタディスプレイ

ConDuLan には SC1602BS タイプの液晶キャラクタディスプレイ用デバイスドライバが含まれていますので、液晶ディスプレイを追加実装すれば iomacro から表示することができます。

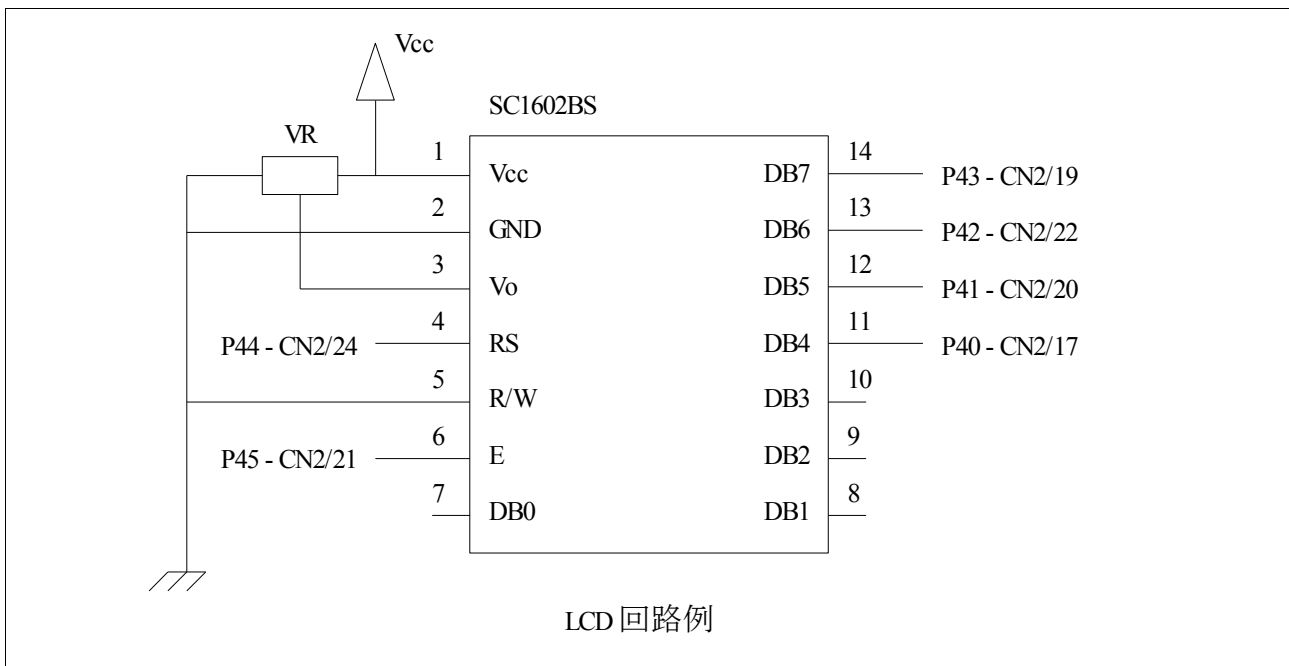
液晶デバイスはポート4のビット0からビット5の6ビットを使用します。

またポート4のビット6とビット7はそれぞれ LED と自励式ブザーを接続することができるよう LED とブザーのデバイスドライバも用意されています。

液晶ディスプレイ上段, 下段, LED*および自励式ブザー**のデバイスドライバ名はそれぞれ

```
/dev/lcd0
/dev/lcd1
/dev/led
/dev/buzzer
```

です。



* 最大出力電流 2mA ですのでバッファが必要です。LED 直接の場合は抵抗値に配慮が必要です。LOW で点灯。P46-CN2/26

** HIGH で奏鳴。P47-CN2/23

23.1 デバイスドライバ /dev/lcd0, /dev/lcd1

液晶表示するには上段, 下段それぞれデバイスドライバ/dev/lcd0と/dev/lcd1を使用します.

液晶表示するには組み込み関数 `open_` と `write_` で表示します.

デバイスをオープンするには

```
lcd0 = { open_, "/dev/lcd0", 1 };
```

のように使用します. 最後のアーギュメント `1` は `WRITE` オープンです.

液晶への `WRITE` 操作はその行をクリアして表示します. `msg0` には表示文字列が入っている場合の例です.

```
{ write_, $lcd0, $msg0, .slen.$msg0 };
```

液晶表示するには上段, 下段それぞれのデバイスドライバをオープンして表示文字列を書き込みます.

例

上段に `Seconds` 下段におよそ1秒毎に1加算した値を表示します.

```
msg0="Seconds";
msg1={ carray_, 17 };
lcd0 = { open_, "/dev/lcd0", 1 };
lcd1 = { open_, "/dev/lcd1", 1 };
x=0;
{ write_, $lcd0, $msg0, .slen.$msg0 };
.while 1 .do
    { print_, $msg1, "%ld", $x };
    { write_, $lcd1, $msg1, .slen.$msg1 };
    x = $x + 1;
    { msleep_, 1000 };
.endwhile
.exit
```

23.2 デバイスドライバ /dev/led

LED 表示するにはデバイスドライバ/*dev/led* を使用します。LED 表示は液晶表示ほど複雑な手順は必要ありませんので *iomacro* で直接プロセッサハードウェアにアクセスしてもかまいませんが、液晶ポートの中の1ビットを LED に使用する場合このデバイスドライバを利用することができます。LED を使用せず1ビットの外部信号として利用することもできます。

LED の制御には組み込み関数 *open_* と *write_* を使用します。

デバイスをオープンするには

```
led = { open_, "/dev/led", 1 };
```

のように使用します。最後のアーギュメント *1* は *WRITE* オープンです。

LED を点灯するにはデータ *1* を書き込みます。消灯するにはデータ *0* を書き込みます。

例

およそ 1 秒間隔で LED のオンオフを繰り返します。

```
on=:1;
off=:0;
led = { open_, "/dev/led", 1 };
.while 1 .do
    { write_, $led, on, 1 };
    { msleep_, 1000 };
    { write_, $led, off, 1 };
    { msleep_, 1000 };
.endwhile
.exit
.end
```

23.3 デバイスドライバ /dev/buzzer

自励式ブザーを鳴らすにはデバイスドライバ/*dev/buzzer* を使用します。ブザー制御も LED 同様複雑な手順は必要ありませんので *iomacro* で直接プロセッサハードウェアにアクセスしてもかまいませんが、液晶ポートの中の1ビットをブザーに使用する場合このデバイスドライバを利用することができます。ブザーを使用せず1ビットの外部信号として利用することもできます。

ブザーの制御には組み込み関数 *open_* と *write_* を使用します。

デバイスをオープンするには

```
buz = { open_, "/dev/buzzer", 1 };
```

のように使用します。最後のアーギュメント *1* は *WRITE* オープンです。

ブザーを鳴らすにはデータ *1* を書き込みます。止めるにはデータ *0* を書き込みます。

例

およそ1秒間隔でブザーのオンオフを繰り返します。

```
on=:1;
off=:0;
buz = { open_, "/dev/buzzer", 1 };
.while 1 .do
    { write_, $buz, on, 1 };
    { msleep_, 1000 };
    { write_, $buz, off, 1 };
    { msleep_, 1000 };
.endwhile
.exit
.end
```

24. ハードウェアリソース

iomacro は直接プロセッサハードウェアをアクセスできるので柔軟なシステム構築が可能ですが、すでに ConDuLan 内蔵ファームウェアで使用しているハードウェアリソースがありますので iomacro からのアクセスにはいくつかの制限があります。

この章ではプロセッサハードウェアを使用する場合の注意事項を紹介します。

24.1 入出力ピン

ConDuLan で使用できる入出力ピンは基板の CN1 と CN2 に接続されている 40 ピンだけです。他のピンはご使用になれません。

これらの入出力ピンを直接アクセスすることは可能ですが、マルチタスク環境下での排他制御を考慮しますと、iomacro に用意されている組み込み関数を利用することを推奨します。また、AD 変換や DA 出力などもマルチタスク環境での使用を考慮して関数 `ad_` および `da_` の利用を推奨します。

また、シリアルポート 0 および 1 はそれぞれ標準シリアルポートおよびデバッグポートとして使用されています。

24.2 プロセッサ内蔵周辺機能

プロセッサ内蔵の周辺機能で ConDuLan が使用しているものは以下の機能です。

かならず使用する機能

8ビットタイマー CH2,3	(システムタイマ)
シリアルポート1	(デバッグポート)
シリアルポート0	(LAN-シリアル変換あるいは iomacro コンソール関数)
DMA チャンネル 1B	(シリアルポート 0DMA 転送)

ROM ファイルや iomacro で指定された場合だけ使用する機能

AD 変換	(AD 変換)
DA 変換	(DA 変換)
ポート4	(LCD,LED,ブザー)
ポート A-0,1,3,4	(I ² C 主, 副)

25. SSI の例

iomacro で記述した SSI 関数を Web ページと連携して動作させるような例を紹介します。

ConDulan の Web メニューから A/D monitor をクリックすると内蔵 AD 変換の結果を表示することができます。このときの表示値は AD 変換結果そのままの 0-4.1V が 0-1023 という値で表示されます。この表示値を電圧値として 0.000V から 4.100V の値で表示するようにしてみます。

SSI は Web ページと連携しますので、ROM ファイルの修正をおこない、動作する SSI を含んだ iomacro プログラムを作成します。

25.1 SSI の例-1 Web ページの修整

添付されている CD の *romfile* ディレクトリにある *universal* ディレクトリ以下を PC へコピーします。コピーした *universal* ディレクトリの中の *http* ディレクトリにある *ad.html* をテキストエディタで開いてください。

```
<!--#exec cgi="cgi_ad 0" -->
```

```
<!--#exec cgi="cgi_ad 1" -->
```

.....

```
<!--#exec cgi="cgi_ad 7" -->
```

などと記述された部分があるはずですが、この *cgi_ad* は ConDuLan に用意してある AD コンバータの値を表示する SSI の名前です。

これから AD 変換の結果を電圧値に変換する SSI を作りこのページに表示しますので、この SSI の名前をたとえば *ssi_ad* として、先ほどの記述を

```
<!--#exec cgi="ssi_ad 0" -->
```

```
<!--#exec cgi="ssi_ad 1" -->
```

.....

```
<!--#exec cgi="ssi_ad 7" -->
```

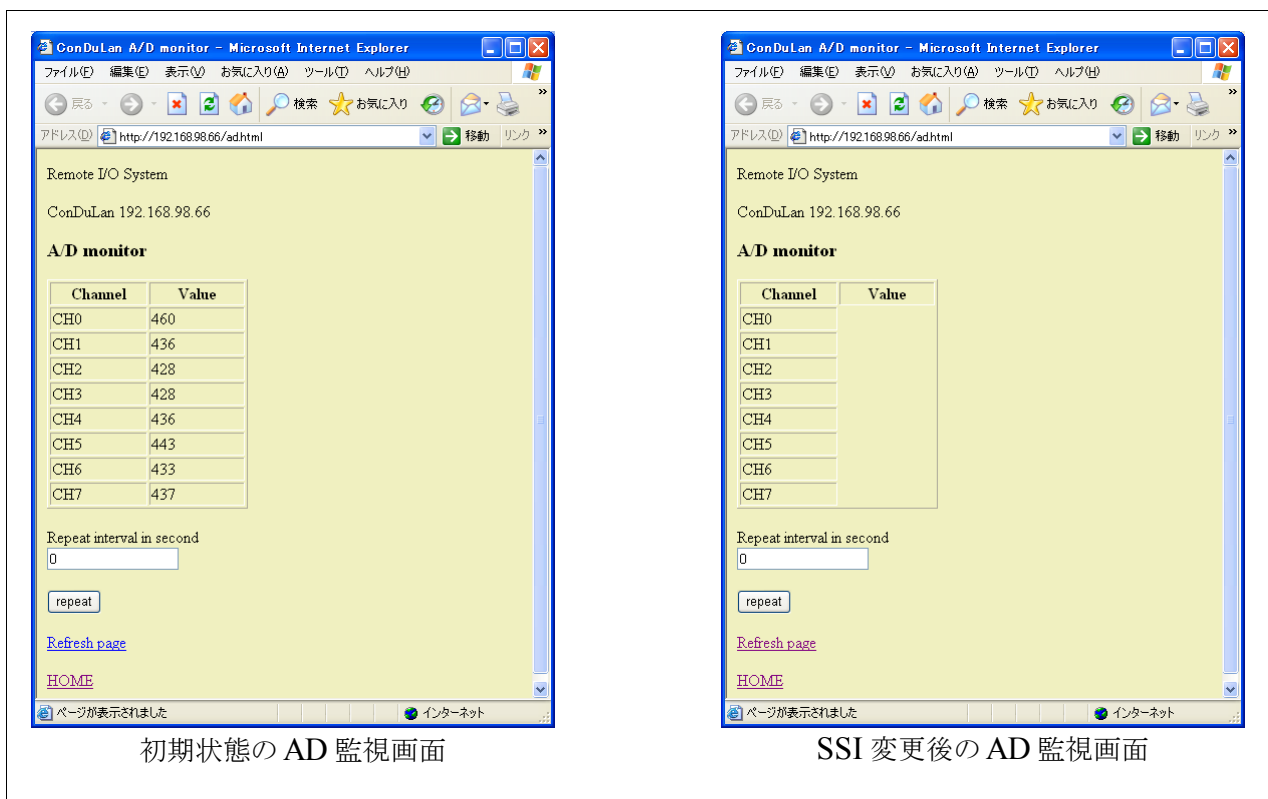
のように変更します。

このファイルを使用して ROM ファイルの TAR ファイルを生成し、ConDulan のインストールモードで *Install EEPROM file* メニューからインストールします。

ROM ファイルについての詳しい説明は「操作ガイド-ROM ファイル編」を参照ください。

25.2 SSI の例-2 Web ページ修整の確認

次にノーマルモードでトップページから *A/D monitor* をクリックして AD コンバータの監視ページを開いてみます。ConDuLan 内蔵の SSI(cgi_ad)が実行されないため AD の値の欄は空白になっています。



25.3 SSI の例-3 iomacro (ハードアクセス)

下のサンプルプログラムはハードウェアを直接アクセスする AD 変換 SSI のソースです。

AD コンバータを起動しその値を読んで電圧値に変換し SSI データとして Web ページに埋め込みます。

メインプログラムは関数 `ssi_ad` を `ssi_ad` という名前で Web サーバに登録します。すでに Web ページ `ad.html` の実行する SSI の名前を `ssi_ad` に変更してありますので、これで関数 `ssi_ad` が実行されます。

```

_len = { ssi_ad, "7" };
{ conl_ };
{ conl_ , "_len=%d", $_len };
{ ssi_ , "ssi_ad", ssi_ad };           //register ssi_ad
.exit                                 //end main

@read_ad                               //AD reader
    0xffffffffe8 =: 0x20 + _0;         //start AD _0 has the AD channel
    .while 0 == 0x80 & $:0xffffffffe8 .do .endwhile //wait for the AD completion
    _ = ($::(0xffffffffe0+(0x07 & (_0 + _0))) >> 6; //read the AD into the return var
    0xffffffffe8 =: 0x00;             //stop AD
    .return                             //end AD read

@ssi_ad                                 //AD SSI
    _ = { print_ , 0, "%5.3fV", .F.{ read_ad, .sl_1 } .* 4.1 ./ 1023.0 }; //generate the AD val
    .return                               //end SSI
.end

```

25.4 SSI の例-4 iomacro (組み込み関数利用)

下のサンプルプログラムは ConDuLan 組み込み関数 *ad_* を使用した AD 変換 SSI のソースです。

機能は前節のプログラムと同じですが、AD 変換アクセスの排他制御が考慮されます。こちらの記述を推奨します。

```
_len = { ssi_ad, "7" };
{ conl_ };
{ conl_, "_len=%d", $_len };
{ ssi_, "ssi_ad", ssi_ad};           //register ssi_ad
.exit                               //end main

@ssi_ad                               //AD SSI
    _ = { print_, 0, "%5.3fV", .F.{ ad_, .sl_1 } .* 4.1 ./ 1023.0 }; //generate the AD val
    .return                           //end SSI
.end
```

25.5 SSI の例-5 iomacro (ソースの説明)

前節のプログラムの説明の続きです。

メイン関数は SSI 登録する前に1度 *ssi_ad* を引数 7 で呼んでいます。これは SSI が正しく動作するかどうかをあらかじめシリアルポートで確認するためです。メインから呼ばれた *ssi_ad* の組み込み関数 *print_* は SSI バッファではなくシリアルポートへ結果を表示します。このときの表示は改行が含まれませんのでメイン関数内で1度 *{ conl_ }* を実行しています。メイン関数はそのあと関数 *ssi_ad* から返された文字列の長さを表示します。最後に関数 *ssi_ad* を SSI 登録して終了します。

AD 監視の Web ページが開かれるとそこに記述されている SSI, たとえば

```
<!--#exec cgi="ssi_ad 0" -->
```

で関数 *ssi_ad* が呼ばれます。引数 *_1* には SSI の引数文字の格納されているアドレスが保管されています。SSI 内の引数はたとえ 0 と書かれていてもそれは "0" という文字で、バイナリではないことに注意してください。

関数 *ssi_ad* では *.sl._1* で引数 1 の文字列を整数に変換しています。関数 *read_ad* で AD コンバータを起動し、完了を待ってからその値を読み出します。

また関数 *ssi_ad* は組み込み関数 *print_* で AD の値を Web サーバに渡して終了します。Web サーバは *ssi_ad* が生成した AD 値の文字列を Web ページに埋め込みます。

SSI の表示書式を浮動小数点5桁(小数点以下3桁)とするために関数 *print_* 呼び出しの書式を *%d* ではなく *%5.3f* とします。また、値は

```
.F.{ read_ad, .sl._1 }. * 4.1 ./ 1023.0
```

にします。この式は AD 変換の値を浮動小数点データに変換し、4.1 倍して 1023.0 で除算しています。

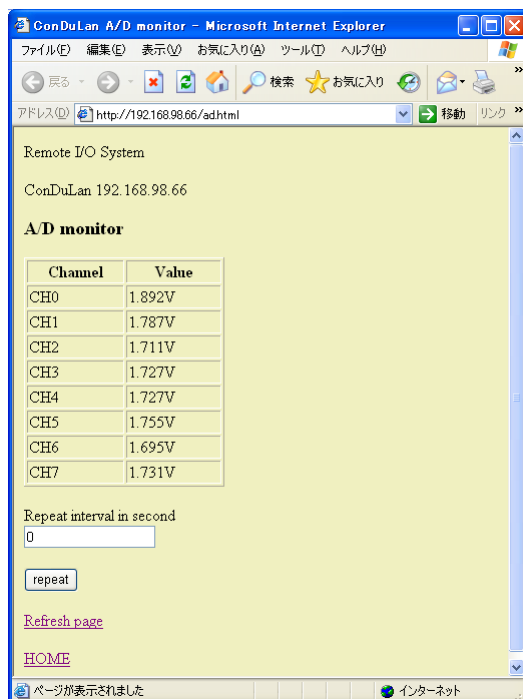
作成した iomacro ソースは ConDuLan のインストールモードから

```
Install the secondary serial ROM
```

でインストールしてください。

25.6 SSI の例-6 表示結果

電圧値表示の iomacro による SSI の表示結果は下のようになります。



Remote I/O System
ConDuLan 192.168.98.66

A/D monitor

Channel	Value
CH0	1.892V
CH1	1.787V
CH2	1.711V
CH3	1.727V
CH4	1.727V
CH5	1.755V
CH6	1.695V
CH7	1.731V

Repeat interval in second

[Refresh page](#)

[HOME](#)

SSI を修正して AD を電圧表示にした

26. POST 用 CGI の例

PC から ConDuLan の Web ページを閲覧するには GET 手順でページ情報を取得して表示します。逆に PC から ConDuLan へデータを送り、ConDuLan の設定変更や出力動作をおこなうには POST 手順を使用します。

ConDuLan に用意されている ROM ファイルでは DA 出力を POST 手続きで実行するようになっていますので、POST 用 CGI の例として DA 出力をおこなう iomacro 関数を作ってみます。

26.1 POST CGI の例-1 ROM ファイルの用意

今までの例のように添付されている CD の romfile ディレクトリにある *universal* ディレクトリ以下を PC へコピーします。

コピーした *universal* ディレクトリの中の *http* ディレクトリにある *da.html* が DA 制御のための Web ページですので、この *da.html* を修整します。

また、*etc* ディレクトリ中のファイル *html_post* には *da.html* に POST すると ConDuLan 内蔵の CGI が動作するよう記述されていますので、ConDulan 内蔵 CGI を起動させないように修整します。次の2ファイルを修正します。

http/da.html

etc/html_post

現在の ConDuLan の DA 出力は 0 から 255 の整数で入力するようになっていきますので、これを 0.0 から 4.1 の電圧値で入力するように変更します。

26.2 POST CGI の例-2 現在の表示

ConDuLan のトップページから D/A control をクリックして DA 制御ページを開いてみます。

表示は下図左のようになります。CH0 の表示の下に現在のチャンネル 0 の出力値 0 が表示されています。その種他の入力欄も 0 ですが、この例ではそこに 128 という値を入力した状態です。

次に set をクリックすると ConDuLan に $da0=128$ というデータが POST 手順で送信されます。その後新しい DA 制御画面が表示され下図右のように CH0 の下に現在値 128 が表示され、出力が 128 担ったことを確認することができます。

The image shows two side-by-side screenshots of a web browser displaying the 'ConDuLan D/A control' page. The left screenshot shows the 'D/A control' section with 'CH0' set to 0 and a text input field containing '128'. A 'set' button is visible below the input field. The right screenshot shows the same page after the 'set' button has been clicked, with the 'CH0' value updated to 128. A blue arrow points from the 'set' button in the left screenshot to the right screenshot. Three black arrows point from the right screenshot back to the left screenshot, indicating the state before the click: one points to the 'CH0' label, one to the '128' in the input field, and one to the 'set' button. A blue arrow also points from the text 'クリックすると設定が出力値となる' to the 'set' button in the left screenshot.

チャンネル 0 の現在値

チャンネル 0 の設定値

クリックすると設定が出力値となる

DA 制御ページチャンネル 0 の現在値 0

DA 制御ページチャンネル 0 の現在値 128
(設定値 128 で set をクリック)

26.3 POST CGI の例-3 DA 表示の SSI 名変更

DA 制御画面の現在出力値や入力ウインドウの現在値は ConDuLan 内蔵の SSI である, 関数 `cgi_da` を利用しています.

ここでの例では `iomacro` で関数 `ssi_da` を作成し, 表示を小数点以下1桁の電圧値で表示するよう変更します.

まず, ROM ファイルの `http/da.html` を修整して関数 `cgi_da` の部分を `ssi_da` に変更します (4ヶ所あります, 下のリストの太字部分です).

```
<html><head><title><!--#exec cgi="cgi_name" --> D/A control</title></head>
<body bgcolor="#f0f0c0">
<p><!--#exec cgi="cgi_system_name" --> Remote I/O System</p>
<p><!--#exec cgi="cgi_name" --> <!--#exec cgi="cgi_ipaddress" --></p>
<p><big><b>D/A control</b></big></p>
<form action=<!--#echo var="DOCUMENT_URI" --> method="post" enctype="text/plain">
<p>
  CH0<br>
  <!--#exec cgi="cgi_da 0" --><br>
  <input type="text" name="da0" value=<!--#exec cgi="cgi_da 0" --> size="16" maxlength="16"><br>
</p>
<input type="submit" value="set">
</form>
<form action=<!--#echo var="DOCUMENT_URI" --> method="post" enctype="text/plain">
<p>
  CH1<br>
  <!--#exec cgi="cgi_da 1" --><br>
  <input type="text" name="da1" value=<!--#exec cgi="cgi_da 1" --> size="16" maxlength="16"><br>
</p>
<input type="submit" value="set">
</form>
<p><a href=<!--#echo var="DOCUMENT_URI" -->>Refresh page</a></p>
<p><a href="/">HOME</a></p>
<!--# include file="credit.inc" -->
</body></html>
```

26.4 POST CGI の例-4 DA 表示の SSI 関数

次に iomacro で SSI を記述します。

下の記述は関数 *ssi_da* で SSI として動作します。

関数 *print_* で表示データを生成します。書式は *%3.1f* で 3 文字表示で小数点以下 1 桁となります。

表示の値は ConDuLan 組み込み関数 *da_* を利用します。引数はチャンネル番号で変換演算子 *.sl* を使用して文字列から整数に変換します。その値は DA の出力値そのものなので、*.F* で浮動小数に変換し、浮動小数点用演算子 *** と */* を使用して 0.0 から 4.1 の電圧値に変換します。

関数 *ssi_da* の戻り値 *_* は生成した表示データのサイズです。

@ssi_da

```
_ = { print_, 0, "%3.1f", .F.{ da_, .sl_1 } . * 4.1 ./ 255.0 };  
.return
```

この関数はメインで SSI 関数として登録します。

26.5 POST CGI の例-5 DA 制御の CGI 登録

PC の Web ブラウザは DA 制御画面の *set* をクリックすると ConDuLan へ DA 出力データを送信します。このときの POST の宛先は *da.html* となっていますが、ConDuLan 内蔵の CGI が動作するようになっています。iomacro で記述した CGI を *da.html* の POST で動作させるためには、

ConDuLan 内蔵 DA 出力 CGI を動作させない。

iomacro の関数を *da.html* の POST 用 CGI として登録する。

必要があります。

ConDuLan 内蔵の CGI は ROM ファイル *html_post* で */da.html* と記述してありますので、その行の先頭に # を追加してコメントにすると起動しなくなります。

iomacro 関数を POST 用 CGI として登録するには関数 *cgipost_* を使用します。

たとえば CGI 関数を *post_da* という名前で作った場合にはメインで

```
{ cgipost_, "/da.html", post_da };
```

という記述をします。これにより、*/da.html* への POST 要求は関数 *post_da* が呼び出されます。

26.6 POST CGI の例-6 DA 制御の CGI ソース

DA 制御をおこなう CGI 関数を次のように記述します。

```
@post_da
    _name = { htmlpost_ _val };
    .if "da0" ==. $_name .then { da_ 0, .f. (.sf. $_val .* 255.0 ./ 4.1) };
    .else
        .if "da1" ==. $_name .then { da_ 1, .f. (.sf. $_val .* 255.0 ./ 4.1) };
    .endif
    .endif
    .return
.end
```

この関数の名前は *post_da* です。

ポストデータはたとえば *da0=1.5* のような 1 行だけで構成されているという前提で (*da.html* ページの仕様がそうなっています), 関数 *htmlpost_* によってデータの名前と値を分離します。名前は変数 *_name* に, 値は変数 *_val* にそれぞれアドレスが書き込まれます。名前も値も文字列です。POST データの名前部分が *da0* だった場合は値部分を DA 出力します。

値は浮動少数の電圧値を表す文字列なので演算子 *.sf.* で浮動少数に変換し, さらに演算子 *.** と *./* で 0.0 から 255.0 の値に変換し整数にします。その結果を ConDuLan 組み込み関数 *da_* でチャンネル 0 へ DA 出力します。

名前が *da1* だった場合はチャンネル 1 へ出力します。

26.7 POST CGI の例-7 DA 制御の iomacro

Web ページ *da.html* を動作させる iomacro プログラムはすでに説明した SSI と POST 用 CGI で次のように記述することができます。

プログラム中関数 *ssi_* と *cgipost_* はそれぞれ SSI 関数と POST 用 CGI 関数を登録しています。

```
{ ssi_, "ssi_da", ssi_da };
{ cgipost_, "/da.html", post_da };      //register post CGI for DA
.exit
@ssi_da
    _ = { print_, 0, "%3.1f", .F.{ da_, .sl_1 } .* 4.1 ./ 255.0 };
    .return
//CGI for post DA
@post_da
    _name = { htmlpost_, _val };
    .if "da0" ==. $_name .then { da_, 0, .f. (.sf. $_val .* 255.0 ./ 4.1) };
    .else
        .if "da1" ==. $_name .then { da_, 1, .f. (.sf. $_val .* 255.0 ./ 4.1) };
        .endif
    .endif
    .return
.end
```

26.8 POST CGI の例-8 da.html の GET 動作

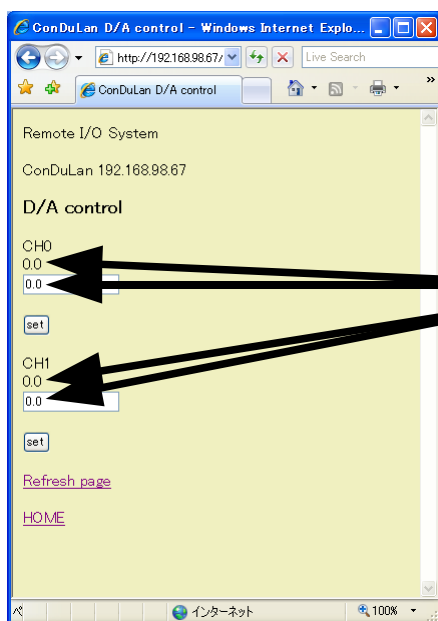
DA 制御の iomacro を使用して *da.html* を表示すると、下図のようになります。このとき PC から ConDuLan へ GET 要求があり、ConDuLan 内蔵の ROM ファイル *http/da.html* が表示されます。

ROM ファイル *http/da.html* には、すでに説明したように

```
<!--#exec cgi="ssi_da 0" -->
```

の記述があり、この部分は SSI 登録された関数 *ssi_da* の出力結果に置き換わって表示されます。関数 *ssi_da* の出力結果は現在のチャンネル 0 の DA 出力値を電圧で表した値です。

GET 要求では POST 用 CGI は動作しませんので前節までに説明した関数 *post_da* は動作しません。



SSI 関数 *ssi_da* が生成した表示

GET 操作による *da.html* の表示

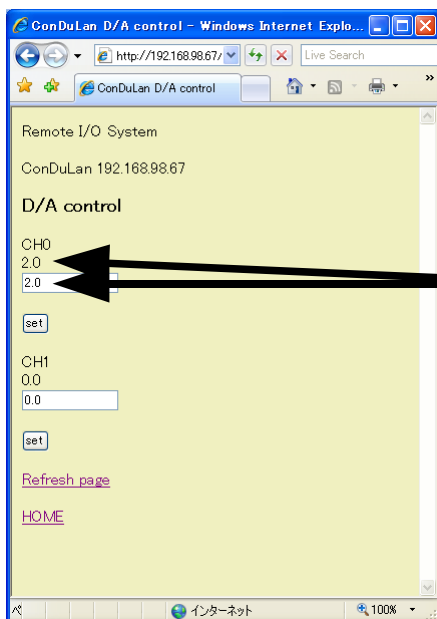
26.9 POST CGI の例-9 da.html の POST 動作

すでに説明したように DA 制御の iomacro を使用して *da.html* の POST 要求を受け付けると、CGI 登録した関数 *post_da* が動作し DA 出力します。

また関数 *post_da* は戻り値を何も操作せずに *return* を実行します。この場合 CGI 関数の戻り値は 0 になります。

ConDuLan システムでは CGI 関数の戻り値が 0 の場合はその URL を名前とする ROM ファイルの内容が応答データとなります。Web ページ *da.html* の POST 要求で、この例のように CGI の戻り値が 0 の場合は ROM ファイル *da.html* の内容が応答データとなり、Web ページ *da.html* を GET 要求した場合と同じ表示となります。ただし、POST 用 CGI 関数が動作した後に応答データを用意しますので SSI の記述は POST 要求によって出力された DA の値を表示します。

このように CGI 関数は DA 出力動作などの実行だけを行い、動作後の表示を ConDuLan アプリケーションによる ROM ファイル表示とすることで CGI の簡素化を実現しています。



POST 要求によって CGI 関数 *post_da* が DA 出力した後に SSI 関数 *ssi_da* が生成した表示

POST 操作による *da.html* の表示

26.10 POST CGI の例-10 da.html の POST 応答

POST 用 CGI が戻り値 0 でリターンした場合は、前節で説明したようにその CGI と同じ名前の ROM ファイルが応答データとなります。

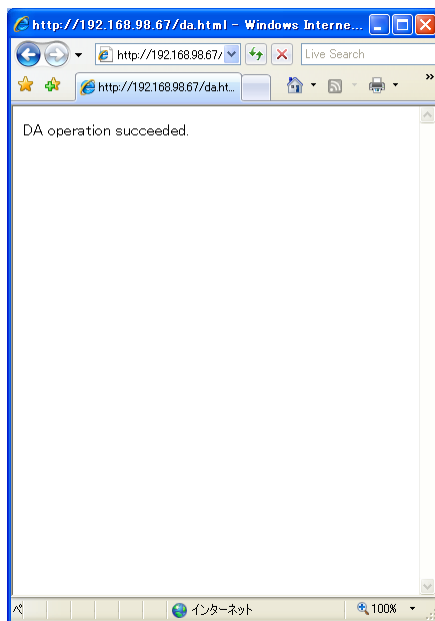
CGI 関数が ROM ファイルではなく独自のデータで応答したい場合には関数 *print_* を使用して応答データを生成します。戻り値は生成した応答データのサイズとなりますが、通常 *print_* 関数の戻り値を使用します。

下の iomacro ソースは CGI 関数 *post_da* が実行した後に "DA operation succeeded." あるいは "DA operation failed." を応答データとして PC の Web ブラウザに表示します。

```
@post_da
    _success = 0;
    _name = { htmlpost_, _val };
    .if "da0" ==. $_name .then
        .if 0 <= { da_, 0, .f. (.sf. $_val .* 255.0 ./ 4.1) } .then _success = 1; .endif
    .else
        .if "da1" ==. $_name .then
            .if 0 <= { da_, 1, .f. (.sf. $_val .* 255.0 ./ 4.1) } .then _success = 1; .endif
        .endif
    .endif
    .if _success .then
        _ = { print_, 0, "DA operation succeeded." };
    .else
        _ = { print_, 0, "DA operation failed." };
    .endif
    .return
.end
```

26.11 POST CGI の例-11 da.html の POST 表示

前節の POST 用 CGI 応答は下のような表示となります。



da.html の POST 用 CGI の応答表示

26.12 POST CGI の例-12 複数データの場合

最後に POST 用 CGI を呼び出すページが複数の入力データを送信するような記述の場合を説明します。

たとえば

```
da0=1.0
```

```
da1=2.0
```

などのデータが ConDuLan へ送られる場合を考えます。

今までの説明では CGI は名前=値のデータが1組だけ送られてくるという仮定で設計されました。複数のデータの可能性がある場合は、*while* を使用して関数 *htmlpost_* をデータがなくなるまで呼びます。

```
@post_da
  .while 1 .do
    _name = { htmlpost_ _val };
    .if 0 == $_name .then .break .endif
    .if "da0" ==. $_name .then { da_ , 0, f. (.sf. $_val .* 255.0 ./ 4.1) };
    .else
      .if "da1" ==. $_name .then { da_ , 1, f. (.sf. $_val .* 255.0 ./ 4.1) };
    .endif
  .endif
  .endwhile
  .return
.end
```

27. GET 用 CGI の例

PC から ConDuLan へデータを送るためには、前章で説明したように POST を使用し ConDuLan に用意した POST 用 CGI 関数がデータに対応した処理をおこないます。

GET による要求でも ConDuLan へデータを送る方法がありますので、この章では GET による CGI を作成してみます。

GET によるデータ送信は、Web ページの URL に続いて?文字そしてクエリー文字列となります。たとえば DA チャンネル 0 に 128 を出力する場合は

http://192.168.98.67/da.html?da0=128

のような要求が ConDuLan へ送られます。

Web ページ *da.html* の GET 用 CGI 関数が用意されていれば名前 *da0* と値 *128* を取得してチャンネル 0 の DA へ 128 を出力することができます。

Web ページから ConDuLan へデータを渡す2つの方法 POST と GET は同じ動作をさせることができますが、Web アクセス制限の有効無効に違いがあります。たとえば ConDuLan から状態を読み出すための GET による Web ページの表示を無制限で許可し、ConDuLan の状態を変更するための POST はパスワード認証を必要とする設定が可能です。このような設定でも一部の CGI にデータを渡すためには GET のクエリーに記述する方法が可能です。この章では GET 用 CGI を作成し GET のクエリーから iomacro 関数がデータを取得する方法を紹介します。

プログラムサンプル用の Web ページは前章と同じく *da.html* を使用します。

27.1 GET CGI の例-1 ROM ファイルの用意

すでに POST の説明にあったように、添付されている CD の romfile ディレクトリにある *universal* ディレクトリ以下を PC へコピーします。

コピーした *universal* ディレクトリの中の *http* ディレクトリにある *da.html* が DA 制御のための Web ページですので、この *da.html* を修整します。

da.html ファイルのなかに次のような記述がありますので *post* の部分を *get* に変更して ROM ファイルをインストールします。変更箇所はチャンネル 0 用と 1 用の 2 ヶ所あります。

この変更で *set* をクリックすると GET 要求が発生し、*da0=128* などのクエリーが要求 URL の後の?*?*に続いて送られます。

```
<form action=
```

27.2 GET CGI の例-2 CGI ソース(htmlget_使用)

GET クエリーにしたがって DA 制御をおこなう CGI 関数を次のように記述します. この記述では関数 `htmlget_` を使用してクエリーデータを取得しています.

CGI 関数は `get_da` という名前で, メインが関数 `cgiget_` を使用して `da.html` の GET 用 CGI として登録しています.

そのほかは POST の場合と同じです.

```
{ cgiget_ , "/da.html" , get_da };  
.exit  
  
@get_da  
  _name = { htmlget_ , _val };  
  .if "da0" .==. $_name .then { da_ , 0 , .sl.$_val };  
  .else  
    .if "da1" .==. $_name .then { da_ , 1 , .sl.$_val };  
  .endif  
  .endif  
  .return  
.end
```

このほかにも組み込み関数 `htmlquery_` を使用する方法もあります. 使用例は 8.5 を参照ください.

28. データのログについて

すでに説明したように、iomacro にはプログラムが生成する文字列データを順にバッファしていくログ機能があります。1行のログデータ内の各要素を TAB などで区切るとマイクロソフトエクセルなどの表計算プログラムに取り込むことができます。

この章では iomacro プログラムでログを記録していき、その結果を PC の Web ブラウザからマイクロソフトエクセルデータとして取得する方法をご紹介します。

28.1 ログの例-2 トップページ変更

実験用に ConDuLan トップページにログデータを取得するためのメニューを追加しておきます。通常は別ページを用意するものですが、手短かに実験して動作を確認するためにこうします。

今までの例のように添付されている CD の *romfile* ディレクトリにある *universal* ディレクトリ以下を PC へコピーします。

コピーした *universal* ディレクトリの中の *http* ディレクトリにある *index.html* がトップページファイルです。

ファイル *inde.html* をテキストエディタで開いて最後から 2 行目の

```
<!--# include file="credit.inc" -->
```

の直前にエクセルファイルへのリファレンスとして

```
<p><a href="log.xls">log.xls</a></p>
```

を挿入してください。ROM ファイル用 TAR ファイルを作り、ConDuLan のセカンダリシリアル ROM へインストールします。

この状態で ConDuLan のトップページを開くと画面の下のほうに *log.exl* というメニューが追加されているのがわかります。

28.2 ログの例-3 ログデータ収集用組み込み関数

ログデータを収集するタスクを作りましょう。

ログデータを集めて保存するためにはその保存領域を用意する必要があります。組み込み関数 `logalloc_` はログデータ保存用の領域を ConDuLan 共有メモリから取得します。

たとえば次の記述はログデータ領域 $256*1024+128$ バイトの領域を確保し、その制御情報のアドレスをグローバル変数 `log_ctl` へ格納します。

```
log_ctl = { logalloc_, 256*1024, 128, 0 };
```

ログデータの追加や、PCへの応答データ生成にはこの `log_ctl` を使用します。最後の引数 128 バイトはバッファマージンです。CGI がログデータを応答バッファへコピーしている間に次のログデータ追加が発生してもこのサイズ分だけは余裕があるように動作します。

ログデータの追加は組み込み関数 `logprint_` でおこないます。下の記述は、すでに確保したログ領域 `log_ctl` に AD チャンネルの 0 値を 1 レコード追加します。バッファの最後まで到達したら先頭に戻り古いデータに上書きしていきます。

```
{ logprint_, $log_ctl, "%d", { ad_, 0 } };
```

28.3 ログの例-3 ログデータ収集タスク

AD データのログ収集タスクとメインプログラムのサンプルです。

メインはログタスクを起動するだけです。ログタスク *logserver* はおよそ1秒ごとに動作します。

ログタスクは初期化関数 *loginit* でログバッファを確保し、ログの制御情報アドレスを変数 *log_ctl* に保存しておきます。

ログタスク関数 *logserver* はチャンネル 0 の AD 変換をおこない関数 *logprint_* でログデータを追加した後シリアルにも AD 変換結果を表示します。その後 1 秒のタスク休止になります。

```
{ task_, logserver, loginit };  
.exit
```

```
@loginit
```

```
log_ctl = { logalloc_, 256*1024, 128, 0 };  
.return
```

```
@logserver
```

```
_ad0 = { ad_, 0 };  
{ logprint_, $log_ctl, "%d\r\n", $_ad0 };  
{ conl_, "logserver ad 0 = %d", $_ad0 };  
_ = 1000;  
.return
```

28.4 ログの例-4 ログデータ応答 GET 用 CGI

前節までで AD データのログが可能になりましたが、まだ CGI を用意していないので PC からの要求でログデータを取得することはできません。

ログデータ取得の CGI 関数は、組み込み関数 `logselsend_` を使用します。

```
_ = { logselsend_, $log_ctl, 0, 0, 0, 0 };
```

のように記述すると保存されているログデータを CGI 用応答バッファへコピーし、その応答データサイズを関数の戻り値とします。CGI 用の関数はその戻り値が 0 のときはデフォルトの (要求と同名の) ファイルデータを PC へ送りますが、正の場合は CGI が応答バッファにデータを用意したものであるとして、そのデータを PC へ送ります。

また、たとえばログデータを送り返す CGI 関数名を `logsend` とすると、CGI の登録は

```
{ cget_, "/log.xls", logsend, 256*1024 };
```

のようにします。最後の 256KB は、CGI が応答バッファとしてこれだけのサイズを使用する旨 Web サーバに登録するための引数です。

28.5 ログの例-5 データログシステムソース

データログシステムの iomacro ソースです。このプログラムではログデータはシリアルにも表示されます。

```
{ task_, logserver, loginit };
{ cgiget_, "/log.xls", logsend, 256*1024 };
.exit

@loginit
    log_ctl = { logalloc_, 256*1024, 128, 0 };
    .return

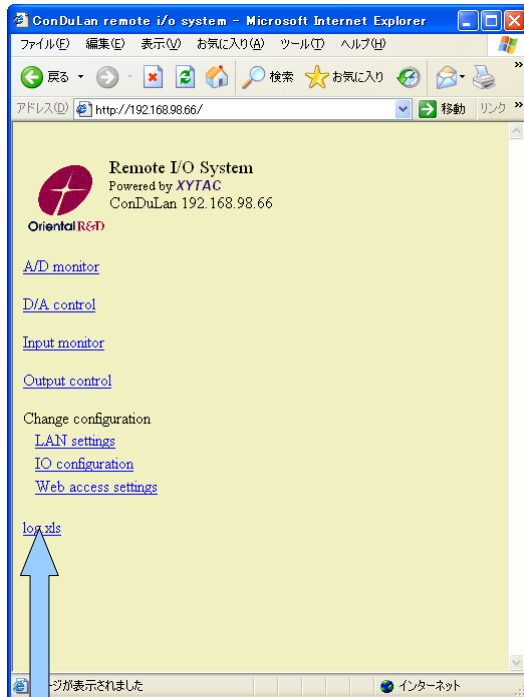
@logserver
    _ad0 = { ad_, 0 };
    { logprint_, $log_ctl, "%d\r\n", $_ad0 };
    { conl_, "logserver ad 0 = %d", $_ad0 };
    _ = 1000;
    .return

@logsend
    _ = { logsendsend_, $log_ctl, 0, 0, 0, 0 };
    .return

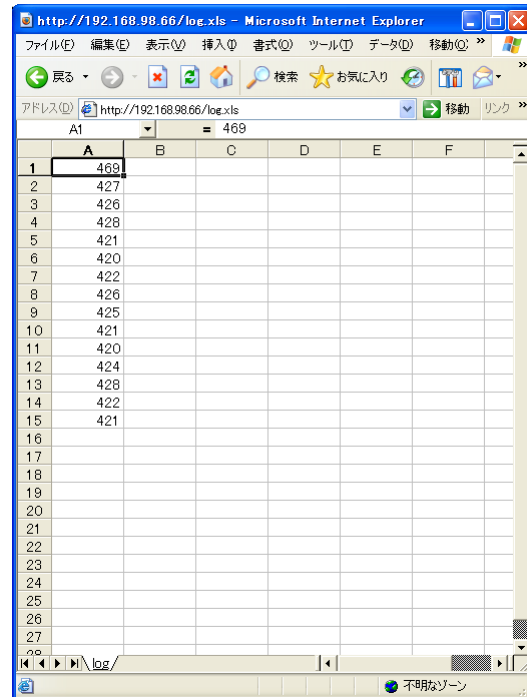
.end
```

28.6 ログの例-6 ログデータ取得画面

Web ブラウザーでトップページに用意した *log.xls* をクリックするとマイクロソフトエクセルが Web ブラウザのプラグインとして動作し、次のような表示になります。



ここをクリックしてログデータを取得する。



マイクロソフトエクセルがプラグインとして起動され、ログデータが表示される。

28.7 ログの例-7 AD0とAD1のログ

AD0とAD1をログするシステムの iomacro プログラムソースです。データはTABで区切られます。

```
{ task_, logserver, loginit };
{ cgiget_, "/log.xls", logsend, 256*1024 };
.exit

@loginit
    log_ctl = { logalloc_, 256*1024, 128, 0 };
    .return

@logserver
    _ad0 = { ad_, 0 };
    _ad1 = { ad_, 1 };
    { logprint_, $log_ctl, "%d\t%d\r\n", $_ad0, $_ad1 };
    { conl_, "logserver ad 0 = %d, ad 1 = %d", $_ad0, $_ad1 };
    _ = 1000;
    .return

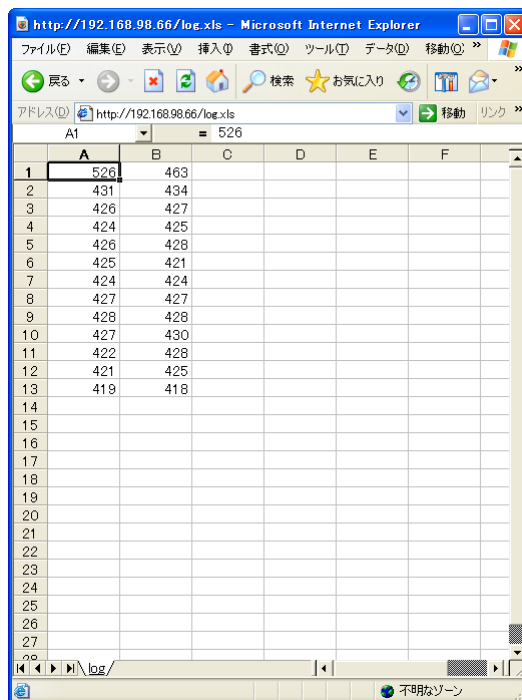
@logsend
    _ = { logselsend_, $log_ctl, 0, 0, 0, 0 };
    .return

.end
```

28.8 ログの例-8 AD0とAD1のログデータ取得画面

AD0とAD1のデータ取得画面です。

TABで区切られた文字は別のセルに表示されます。



The screenshot shows a Microsoft Internet Explorer browser window displaying an Excel spreadsheet. The address bar shows the URL <http://192.168.98.66/log.xls>. The spreadsheet has columns labeled A through F and rows numbered 1 through 27. The data is as follows:

	A	B	C	D	E	F
1	526	463				
2	431	434				
3	426	427				
4	424	425				
5	426	428				
6	425	421				
7	424	424				
8	427	427				
9	428	428				
10	427	430				
11	422	428				
12	421	425				
13	419	418				
14						
15						
16						
17						
18						
19						
20						
21						
22						
23						
24						
25						
26						
27						

29. 排他制御について

`iomacro` はマルチタスクが可能のためメモリ領域などシステムの共有資源を使用する場合他のタスクとの関係について注意する必要があります。今までの説明では説明を簡素化するためにこの点には注意を払ってきませんでした。

たとえ `iomacro` で明示的にマルチタスクを構築しなくても SSI や CGI を使用すると複数のタスクで同一の関数を実行する可能性があります。グローバル変数の書き換えや H8 ハードウェアへの書き込み動作で他のタスクとの干渉が懸念される場合は2つの組み込み関数 `lock_` と `unlock_` を使用してタスクスイッチを禁止して操作してください。またタスクスイッチ禁止期間は極力短くなるよう配慮してください。ConDuLan に用意されているほとんどの組み込み関数はマルチタスク環境でも正常に動作するよう設計されていますので、可能な場合は組み込み関数をご利用ください。

ConDuLan の Web サーバはどのタスクも同じ優先順位で動作しています。また、内蔵 RTOS はタイムスライス機能はないので、同一プライオリティのタスクによってプリエンプションされることはありません。共有資源へのアクセス競合が Web サーバ (SSI と CGI) だけによるものであればタスクスイッチを禁止することなくアクセスしても大丈夫です。ただしメインおよび `iomacro` タスクと SSI および CGI ではタスク優先順位が異なりますので注意が必要です。

タスクロックと解除は

```
{ lock_ };  
ここで共有資源の処理をする。  
{ unlock_ };
```

のように使用します。

おわりに

ConDuLan に用意されている豊富な機能を利用させていただくためのスクリプト言語 **iomacro** をご紹介してきました。 **iomacro** はソースインストールという手軽さとハードウェア直接アクセスできる柔軟性も兼ね備えています。活用されてご満足のいくシステムを構築できるものと願っております。

ConDuLan

操作ガイド --- iomacro 編

発行年月日 2008年3月24日 第02版C

発行 オリエンタルアールアンドディー株式会社

著作 オリエンタルアールアンドディー株式会社

Printed in Japan

オリエンタルアールアンドディー株式会社

<http://www.ord.co.jp>