SilentSystem

ＯＳ－１プログラミングマニュアル

Ver0.6　2007/6/25

はじめに

この度はサイレントシステムのワンチップサーバーＯＳ－１をご購入頂きありがとうございました。本マニュアルはユーザーがＯＳ－１上で動作するプログラムを作成するための技術的な情報を記していますので十分に理解した上でご利用ください。またＯＳ－１の取り扱いに関してはＯＳ－１ユーザーズマニュアルを参照してください。

免責事項

ＯＳ－１は一般電子機器用の半導体部品を使用しておりますので、生命に関わる用途や身体に害を及ぼす恐れのある用途には使用出来ません。またＯＳ－１は基本的にお客様が使用目的に適合したソフトウェアを組み込んで使用する機器ですので、使用の前に十分なテストを行い正しく動作を確認してから使用を開始して下さい。またＯＳ－１の運用の結果については有限会社サイレントシステムはいかなる責任も負えません。

ＯＳー１に内蔵しているソフトウェア及びマニュアルには欠陥が含まれている可能性がありますので、その信頼性や正確性を保証する事は出来ません。またその欠陥を修正する事を保証する事もできません。

またライブラリを始めとする仕様は予告無く変更する場合がありますので弊社のサイトを確認して最新のプログラミングマニュアルをご利用下さい。

インタープリター型C言語　SilentC

SilentCはインタープリタ型のC言語です。C言語のサブセットな言語処理系でC言語と同様にプログラムが可能です。シリアルターミナルあるいはtelnetでSilentCにアクセスするだけでプログラム開発とデバッグが完結します。殆どの簡単なアプリケーションであればSilentCを利用して短時間にプログラムを完成できます。またSilentCには以下の特徴があります。

・Cコンパイラが不要です

SilentCはインタープリター言語です。ソースを修正すればすぐにその結果が反映します。コンパイルやリンクやターゲットへの転送の手間が省略されます。組み込んでしまった機器のデバッグや現場での仕様変更にも大変有効です。

・ダイレクト実行が可能です

SilentCで書かれたプログラムはいつでも中断ができます。またコンソールからコマンドを入力すると変数の値を確認したり任意の関数を実行してその戻り値を確認したり出来ます。Cのステートメントを直接入力する事でポートを操作したり関数をダイレクト実行できます。ちょうどVisualBasicや昔のマイコンに内蔵されていたBASICと同じ感覚です。

・ファイル内の関数をダイレクトに呼び出せます

SilentCのプログラムはRAMではなくファイルに置かれています。これを利用してファイルの中の関数を呼び出す事ができるので一度作成したSilentCプログラムを自分だけのライブラリとして再利用できます。C言語の仕様でローカル変数が利用可能ですので機能ブロックをモジュール化して呼び出すことが出来ます。

・ユーザードライバとの組み合わせが可能です

より高速性やリアルタイム性が求められるアプリケーションに対応するために1kバイトまでのユーザードライバとのリンクが可能です。ユーザーが割り込みハンドラなどを用意してSilentCと連携させる事でより広範囲のアプリケーションに対応できます。UserDriverライブラリを利用してユーザーが作成したネイティブコードに値を渡したり受け取ったりできます。

SilentC入門

SilentCの動作を理解するためにステップバイステップで各種機能を説明していきます。サンプルのSilentCプログラムを入力してその結果をひとつずつ確認しながらSilentCを学習していきます。

・準備

別冊のユーザーズマニュアルを参考にターミナルでSilentCを操作できる状態にしてください。またサンプルプログラムをコピーしてターミナルにペーストすると簡単に入力できますので必ず本マニュアルからターミナルへコピー&ペーストする方法を確認しておいて下さい。

・スタートアップファイルとは

SilentCにとってスタートアップファイルは大変重要な役割を持っています。グローバル変数はスタートアップファイル内に記述しますしrunコマンドで走り出すmain()関数はスタートアップファイルの中に記述します。またダイレクト実行に先立ってスタートアップファイルの最初からグローバル変数をすべて読み込んで生成します。

SilentCのデフォルト設定ではスタートアップファイルはMainというファイル名です。このマニュアルではもっぱらこのMainというファイルを書き換えながら解説をすすめていきます。

・ファイルのバックアップ方法

まずMainのオリジナルを保存します。以下の手順でMainのコピーを作成してください。

```
load Main
save MainOrg
load Main
```

この操作によりMainがMainOrgにコピーされます。この手順はファイルになんらかの変更を加える前によく使う手法ですので覚えておくと良いと思います。特に3行目のloadコマンドは大変重要な意味を持っています。というのはSilentCでsaveコマンドでファイル名を省略した際には直前のファイル名が使われます。もしこの3行目のloadコマンドを入力しないままsaveコマンドのみを入力すると直前に使われたMainOrgというファイルに変更を上書きしてしまう危険があります。MainOrgはオリジナルファイルですので上書きは絶対に避けたいところです。こうした理由からバックアップファイルを作成する際には以上の3行をデフォルトの手順として覚えておいて下さい。

・ためしにMainファイルを消してしまいます

以下のコマンドを入力します。

```
delete Main
```

このコマンドによりスタートアップファイルのMainが削除されます。この状態はSilentCにとっては想定外の異常事態です。たとえば?1+1というような簡単なダイレクト実行コマンドを入力しても Main not found in startupというエラーを表示します。このエラーが出た場合にはMainが無いという事を覚えておいて下さい。

・改めて新規にMainファイルを作成します

以下のコマンドを入力します。

```
new
save Main
```

newコマンドで編集バッファをクリアしてsaveコマンドでクリアした内容をMainに上書きします。この操作により今後はsaveコマンドのみを入力するとMainに対するsaveと解釈されます。確認の意味で以下のコマンドを入力してみてください。

```
save
```

ＯＳ－１からはsave Main ?(Y or N)と聞かれますのでyと答えて下さい。今後はこのsave及び確認は何度も使いますので確実にこの手順は覚えて下さい。

SilentCの予約語

SilentCには以下のキーワードが予約語として登録されています。これらの単語は変数として利用できませんのでご注意下さい。

if, else, do, while,
for, break, continue, return,
switch, case, default, goto,
void, char, int, short, long,
unsigned, signed, struct, sizeof,

#stop, #define, #include,
#if, #ifdef, #ifndef,
#else, #endif, #undef

SilentCの文法

SilentCはC言語のサブセットと呼べる言語です。特にプログラムの制御に関する記述は、C言語の制御構造をそのまま記述できます。C言語との大きな違いは配列と構造体などの変数にアクセスする部分です。

SilentCの制御文や変数アクセスに関しての説明に関してはこのマニュアルの次のバージョンで詳しく解説する予定です。現状ではサンプルプログラムなどを参照してSilentCの動作を理解して下さい。

SilentCの組み込みライブラリ解説

この章ではSilentCに予め組み込まれているライブラリに関して解説します。解説中のサンプルプログラムは実際にOS-1に入力して動作させる事で、よりSilentCに対する理解が深まりますのでぜひお試し下さい。

コンソール入出力ライブラリ

void PrChar(char ch)

解説: コンソールに1文字出力します。
引数: ch = 表示する文字
戻り値: ありません
サンプル:

```
PrChar('A');    //Aを表示する(ダイレクト実行)
PrChar(49);     //1を表示する(ダイレクト実行)
```

int PrStr(char* str)

解説: コンソールに文字列を出力します。
引数: str = 表示する文字列へのポインタ
戻り値: 表示した文字列の長さ
サンプル:

```
main(){
char *str="Hello ";PrStr(str);          //ポインタstrにHello という文字列をセットして表示する
PrStr("World!!");               //World!!を表示する
}
```

void PrNum(int num)

解説: コンソールに数値を出力します。
引数: num = 表示する数値
戻り値: なし
サンプル:

```
main(){
int a=12345;PrNum(a);     //変数aに12345を代入してそれを表示する
PrNum(1000);              //1000を表示
}
```

void PrHex(int num)

解説: コンソールに4桁の16進数を出力します。
引数: num = 表示する数値
戻り値: なし
サンプル:

```
PrHex(4660);   //10進数の4660を16進数に変換して表示する(ダイレクト実行)
```

void PrHexByte(char num)

解説: コンソールに2桁の16進数を出力します。
引数: num = 表示する数値
戻り値: なし
サンプル:

```
PrHexByte(251);     //10進数の251を16進数2桁で表示する(ダイレクト実行)
```

void PrAdrs(long adrs)

解説: コンソールにIPアドレス形式で出力します。
引数: adrs = 表示するIPアドレス(32ビット)
戻り値: なし
サンプル:

```
PrAdrs(3232235786);      //192.168.1.10を表示させる(ダイレクト実行)
```

int Gets(char *buf, int size)

解説: コンソールから1行入力します。
引数: buf = 入力した文字を格納するバッファのポインタ、
size = 確保したバッファのサイズ
戻り値: 入力された文字数
サンプル:

```
main(){
char *buf=MemoryAlloc(64);   //入力バッファを確保
Gets(buf,64);                //1行入力
PrStr(buf);                  //入力された文字列を表示
}
```

char Getc(char wait)

解説: コンソールから1文字入力します。
引数: wait = 0ならば入力を待たない(キーセンス)、0以外なら入力されるまで待つ
戻り値: 入力された文字コード、キ-センス時に0ならばキーは押されていない
サンプル:

```
PrChar(Getc(1));             //入力した1文字を即表示する(ダイレクト実行)
```

long Atoi(char *str)

解説: 文字列から数値に変換します。
引数: str = 文字列へのポインタ
戻り値: 変換された数値(32ビット)
サンプル:

```
main(){
char *buf=MemoryAlloc(64);   //入力バッファを確保
Gets(buf,64);                //1行入力
PrNum(Atoi(buf));            //文字列から数値に変換してその値を表示
}
```

void GetDigit(long num, char *buf)

解説:　数値から文字列に変換します。
引数:　num = 変換する数値
　　buf = 文字列を格納するバッファへのポインタ
戻り値:　なし
サンプル:

```
main(){
char *buf=MemoryAlloc(12);　//結果を格納するバッファを12バイト分確保
GetDigit(12345,buf)　　//数値の12345を文字列に変換してbufに格納
PrStr(buf);　　//出来た文字列を表示して確認
}
```

char SciSense(char port)

解説:　シリアルポートをセンスします
引数:　port = シリアルポートのチャンネル。
　　0を指定するとOSIO-1のRS-232Cポート、
　　1を指定するとユーザー拡張用の3.3Vレベルのシリアルポート
戻り値:　シリアルデータが入力されていれば1を返す
サンプル:

```
main(){
#stop 0　　//ブレークセンスを禁止する(文字抜け防止)
while(!SciSense(0));　　//シリアルポートに入力されるまで待つ
PrChar(SciGetc(0));　　//入力された文字を表示
}
```

char SciGetc(char port)

解説:　シリアルポートから1文字入力されるまで待って
引数:　port = シリアルポートのチャンネル。
　　0を指定するとOSIO-1のRS-232Cポート、
　　1を指定するとユーザー拡張用の3.3Vレベルのシリアルポート
戻り値:　入力された文字データを返す
サンプル:

```
PrChar(SciGetc(0));　　//1文字入力するまで待って入力された文字を表示(ダイレクト実行)
```

void SciPutc(char ch, char port)

解説:　シリアルポートへ1文字出力します
引数:　ch = 出力する文字データ、
　　port = シリアルポートのチャンネル。
　　　0を指定するとOSIO-1のRS-232Cポート、
　　　1を指定するとユーザー拡張用の3.3Vレベルのシリアルポート
　　PrCharと似ているがPrCharはコンソールに出力する関数。
　　telnetで接続時にはコンソールはネットワーク経由になる。
　　SciPutcはシリアルポートに出力する。
戻り値:　なし
サンプル:

```
main(){
SciPutc('A',0);  //Aという文字をシリアルポートに出力
}
```

void SciPuts(char *str, char port)

解説:　シリアルポートへ文字列を出力します
引数:　str = 出力する文字列へのポインタ、
　　port = シリアルポートのチャンネル。
　　　0を指定するとOSIO-1のRS-232Cポート、
　　　1を指定するとユーザー拡張用の3.3Vレベルのシリアルポート
　　PrStrと似ているがPrStrはコンソールに出力する関数。
　　telnetで接続時にはコンソールはネットワーク経由になる。
　　SciPutsはシリアルポートに出力する。
戻り値:　なし
サンプル:

```
main(){
SciPuts("String",0); //Stringという文字をシリアルポートに出力
}
```

int SciGets(char *str, int size, char port)

解説:　シリアルポートから1行入力する
引数:　str = 入力した文字列を格納するバッファのポインタ、
size = バッファのサイズ、
port = シリアルポートのチャンネル。
0を指定するとOSIO-1のRS-232Cポート、
1を指定するとユーザー拡張用の3.3Vレベルのシリアルポート
0x0Dを受け取るとNULL文字に置き換えて入力を終了して戻る。
エコーバックは無い。
戻り値:　入力された文字列の長さ。
サンプル:

```
main(){
char *str=MemoryAlloc(64);    //入力バッファを64バイト確保する
SciGets(str,64,0);            //シリアルから1行入力する。エコーバックなし。
PrStr(str);                   //入力された文字列をコンソールに表示する
}
```

メモリー操作ライブラリ

char *MemoryAlloc(int size)

解説:　ヒープ領域からメモリーを確保する。
引数:　size = 確保するバッファのバイト数
戻り値:　確保された領域へのポインタ。確保に失敗すると0を返す。
サンプル:

```
main(){
char *ptr=MemoryAlloc(16);    //16バイトの領域を確保する
StrCpy(ptr,"String");         //確保した領域にStringという文字列をコピー
StrCat(ptr," Buffer");        //Bufferという文字列を結合
PrStr(ptr);                   //文字列を表示
MemoryFree(ptr);              //領域を開放する
}
```

void MemoryFree(char *ptr)

解説:　確保した領域を開放する
引数:　ptr = 開放するバッファへのポインタ。0を指定すると何もしない。
　　MemoryAllocで確保したポインタ以外を与えるとOSがクラッシュするので注意。
戻り値:　なし
サンプル:

```
main(){
char *ptr=MemoryAlloc(16);    //16バイトの領域を確保する
StrCpy(ptr,"String");         //確保した領域にStringという文字列をコピー
StrCat(ptr," Buffer");        //Bufferという文字列を結合
PrStr(ptr);                   //文字列を表示
MemoryFree(ptr);              //領域を開放する
}
```

void BufCopy(char *dest, char *src, int size)

解説:　メモリー領域をコピーする
引数:　dest = コピー先へのポインタ、
　　src = コピー元のポインタ
　　size = コピーするバイト数。0を指定すると何もしない。
　　コピーする領域が重なっていても正しくコピーされる。
戻り値:　なし
サンプル:

```
main(){
char *ptr=MemoryAlloc(16);    //16バイトの領域を確保する
BufCopy(ptr,"String",7);      //確保した領域にStringという文字列をコピー
PrStr(ptr);                   //文字列を表示
MemoryFree(ptr);              //領域を開放する
}
```

void MemClear(void *dest, int size)

解説: メモリー領域を0で埋めてクリアする
引数: dest = クリアする領域へのポインタ
size = クリアするバイト数。0を指定すると何もしない。
戻り値: なし
サンプル:

```
main(){
char *ptr=MemoryAlloc(16);    //16バイトの領域を確保する
BufCopy(ptr,"String",7);      //確保した領域にStringという文字列をコピー
MemClear(ptr+3,4);            //3文字目以降をクリアする
PrStr(ptr);                   //文字列を表示、Strのみになる
MemoryFree(ptr);              //領域を開放する
}
```

文字列操作ライブラリ

int StrLen(char *str)

解説:　文字列の長さを求める
引数:　str = 文字列へのポインタ
戻り値:　文字列の長さを返す
サンプル:

```
main(){
PrNum(StrLen("abcde"));　　//abcdeという文字列の長さを表示する
}
```

char *StrCpy(char *buf, char *str)

解説:　文字列をコピーする
引数:　buf = コピー先の領域へのポインタ、
　　str = コピーする文字列へのポインタ
戻り値:　コピー先の領域のポインタを返す
サンプル:

```
main(){
char *ptr=MemoryAlloc(16);　//16バイトの領域を確保する
PrStr(StrCpy(ptr,"abcde"));　//abcdeという文字列をコピーして表示する
}
```

char *StrCat(char *head, char *tail)

解説:　文字列を連結する
引数:　head = 前半部分の文字列へのポインタ。連結された文字列が格納される。
　　tail = 連結する後半部分の文字列へのポインタ
戻り値:　連結された文字列へのポインタを返す
サンプル:

```
main(){
char *ptr=MemoryAlloc(16);　//16バイトの領域を確保する
StrCpy(ptr,"abc");　　//abcという文字列を領域にセットする
StrCat(ptr,"def");　　//defという文字列を連結する
PrStr(ptr);　　//連結された文字列を表示する。abcdefになる
MemoryFree(ptr);　　//領域を開放する
}
```

int StrCmp(char *str1, char *str2)

解説:　文字列を比較する
引数:　str1 = 比較される文字列へのポインタ、
　　str2 = 比較する文字列へのポインタ。
戻り値:　2つの文字列が等しい場合には0、そうでない場合には0以外を返す
サンプル:

```
main(){
PrNum(StrCmp("123","123"));//123という文字列同士を比較して結果を表示する
PrNum(StrCmp("123","456"));//123と456を比較して結果を表示する
PrNum(StrCmp("1234","123"));　　//123と456を比較して結果を表示する
}
```

int StrChr(char *str, char scan)

解説:　文字列から対象の文字を検索する
引数:　str = 文字列へのポインタ、
　　scan = 検索する文字
戻り値:　文字が見つかった場合にはその位置を、見つからなかった場合には0を返す。
サンプル:

```
main(){
PrNum(StrChr("12345",'3'));　//12345という文字列の中での3の位置を同士を比較して結果を表示する
PrNum(StrChr("12345",'A'));　//123と456を比較して結果を表示する
}
```

char *StrStr(char *str, char *scan)

解説:　文字列から対象の文字列を検索する
引数:　str = 文字列へのポインタ、
　　scan = 検索する文字列へのポインタ
戻り値:　文字が見つかった場合にはその位置のポインタを、
　　見つからなかった場合には0を返す。
サンプル:

```
main(){
PrHex(StrStr("12345","34"));  //12345という文字列の中での34の位置のポインタを表示
PrHex(StrChr("12345","A"));  //12345という文字列の中にはAは無いので0を表示する
}
```

ファイル読み込みライブラリ

int OpenFile(char *name)

解説:　ファイルを読み込みモードでオープンする
引数:　name = オープンするファイル名
戻り値:　オープンに成功した場合には16ビットのファイルハンドラを
　　対象のファイルが見つからなかった場合には0を返す。
サンプル:

```
main(){
int fh;                    //ハンドラ変数を準備
fh = OpenFile("Main");     //Mainというファイルをオープンして変数に格納
PrHex(fh);                 //取得したファイルハンドラの値を表示
CloseFile(fh);             //オープンしたファイルは必ず閉じる、閉じなければどんどんメモリーを消費する
}
```

int ReadFile(int handle, char *buf, int size)

解説:　ファイルからデータを読み込む
引数:　handle = オープンした際に取得したファイルハンドラ
　　buf = 読込バッファのポインタ、
　　size = 読み込むサイズ
戻り値:　読み込みに成功したデータのサイズを返す
サンプル:

```
main(){
int fh;                          //ハンドラ変数を準備
char *buf=MemoryAlloc(64);       //読み込むバッファを確保
fh=OpenFile("Main");             //Mainというファイルをオープン
ReadFile(fh,buf,64);             //64バイト読む
buf[63]=0;PrStr(buf);            //文字列の終端に0をセットして内容を表示する
CloseFile(fh);MemoryFree(buf);//きちんと後始末をする(メモリーリーク防止の為)
}
```

int SeekFile(int handle, long pos, char org)

解説:　ファイルの読み込み位置を設定する
引数:　handle = オープンした際に取得したファイルハンドラ、
　　pos = 設定する読み込み位置、
　　org = 読み込み位置の設定モード
　　　0ならファイルの先頭からの位置
　　　1なら現在の読み込み位置からの相対位置
　　　2ならファイルの終端からの位置
戻り値:　新しくセットされたファイル内の読み込み位置を返す
サンプル:

```
main(){
int fh;                     //ハンドラ変数を準備
fh=OpenFile("Main");        //Mainというファイルをオープン
SeekFile(fh,10,0);          //先頭から10バイト目に読み込み位置を設定
PrNum(SeekFile(fh,0,1);     //現在設定されている読み込み位置を表示(10になる)
SeekFile(fh,5,1);           //現在の位置から5だけ読み込みポインタを進める
PrNum(SeekFile(fh,0,1);     //現在設定されている読み込み位置を表示(15になる)
SeekFile(fh,0,2);           //ファイルの終端に移動する
PrNum(SeekFile(fh,0,1);     //現在設定されている読み込み位置を表示(Mainのファイルサイズになる)
CloseFile(fh);              //きちんと後始末をする(メモリーリーク防止の為)
}
```

int FileGets(int handle, char *buf, char size)

解説:　ファイルから1行を読み込む。
　　初期化と終了時にバッファの開放が必要なので注意すること
引数:　handle = オープンした際に取得したファイルハンドラ、
　　buf = 読み込むバッファへのポインタを指定する
　　size = バッファのサイズを指定。
　　　0なら初期化、1なら内部バッファを開放する。
戻り値:　読み込んだ行の文字数を返す。CR/LFは取り除かれる。
　　ファイルが終端に達してもうこれ以上読み込めない場合には-1を返す
サンプル:

```
main(){
int fh;                         //ハンドラ変数を準備
char *buf=MemoryAlloc(80);      //1行読み込むバッファを確保
fh=OpenFile("Main");            //Mainというファイルをオープン
FileGets(fh,0,0);               //FileGetsを初期化
for(;;){                        //繰り返し1行読むためのループ開始
 if(FileGets(fh,buf,80)<0)break;//ファイルから一行読み込む、読めなくなったらループを抜ける
 PrStr(buf);PrStr("\r\n");}     //読み込んだ1行を表示する。ループの繰り返し
FileGets(fh,0,1);               //FileGetsの終了処理(内部バッファの解放)
CloseFile(fh);MemoryFree(buf);//きちんと後始末をする(メモリーリーク防止の為)
}
```

void CloseFile(int handle)

解説:　ファイルをクローズする。内部的に確保されていたメモリを開放する。
引数:　handle = オープンした際に取得したファイルハンドラ、
戻り値:　なし
サンプル:

```
main(){
int fh;                 //ハンドラ変数を準備
fh=OpenFile("Main");    //Mainというファイルをオープン
CloseFile(fh);          //きちんと後始末をする(メモリーリーク防止の為)
}
```

long GetFileSize(char *name)

解説:　ファイルの大きさを返す
引数:　name = 大きさを調べるファイルの名前
戻り値:　指定されたファイルの大きさを返す

サンプル:

```
main(){
PrNum(GetFileSize("Main"));   //Mainというファイルの大きさを調べる
PrNum(GetFileSize("????"));   //????は存在しないファイルなので0を返す
}
```

char *FindFile(char init)

解説:　ファイル名を次々に取り出す
引数:　init = 1を指定するとFindFileが初期され先頭のファイルからスタートする
戻り値:　ファイル名へのポインタを返す。このポインタはユーザーが開放する必要がある
　　もうこれ以上ファイルが存在しない場合には0を返す
サンプル:

```
main(){
char *buf;              //ポインタ変数を準備
FindFile(1);            //FindFileを初期化
for(;;){                //繰り返しファイル名を表示するためのループ開始
 buf=FindFile(0);       //ファイル名を取得する。
 if(buf==0) break;      //ポインタが0ならファイル名はもう存在しないのでループを抜ける
 PrStr(buf);PrStr("\r\n");//読み込んだファイル名を表示する
 MemoryFree(buf);}      //バッファの開放(メモリーリーク防止) ループの繰り返し
}
```

ファイル書き込みライブラリ

ファイルを書き込む際の注意は書込みのためにオープンできるファイルは1個だけだという点です。1個だけですのでファイルハンドラを用いる必要はありません。書き込み用のハンドラは0に固定されています。

char CreateFile(char *name)

解説:　書き込みモードでファイルをオープンする
引数:　name = 作成するファイル名へのポインタ
戻り値:　成功すれば0を返す。失敗した場合には以下のエラーコードを返す。
　　1:すでに書き込みモードでオープンしているファイルがある2:ファイル名が不正
　　3:すでに同名のファイルが存在している。DeleteFileで削除の必要がある
　　4:ディスクの容量が不足している
　　5:内部バッファが確保できない。(メモリー不足)
サンプル:

```
main(){
PrNum(CreateFile("test"));　//ファイルを作成して結果を表示する
CloseFile(0);　//ファイルを閉じてディスクをフラッシュする
}　　//実行するとtestというファイルが作成される。コマンドモードでdelete testで削除できる
```

void CloseFile(const 0)

解説:　ファイルをクローズして内部的に保留されているデータをすべて書き出す。
引数:　必ず0を指定する。
戻り値:　なし
サンプル:

```
main(){
PrNum(CreateFile("test"));　//ファイルを作成して結果を表示する
CloseFile(0);　　//ファイルを閉じてディスクをフラッシュする
}　　//実行するとtestというファイルが作成される。コマンドモードでdelete testで削除できる
```

int WriteFile(char *buf, int size)

解説:　ファイルにデータを書き込む
引数:　buf = データへのポインタを指定する。
　　size = 書き込むバイト数
戻り値:　書き込んだバイト数を返す
サンプル:

```
//すでにtestというファイルが存在しているならコマンドモードでdelete testで削除する
main(){
CreateFile("test");　　//testというファイルを新規に作成する
WriteFile("abc\r\n",5);　//abcとCR/LFの合計5文字をファイルに書く
CloseFile(0);　　//ファイルを閉じてディスクをフラッシュする
}　　//実行するとtestというファイルが作成される。コマンドモードでtype testで内容を確認
```

int FilePuts(char *str)

解説:　ファイルに文字列を書き込む
引数:　str = 書き込む文字列へのポインタを指定する。
戻り値:　書き込んだバイト数を返す
サンプル:

```
//すでにtestというファイルが存在しているならコマンドモードでdelete testで削除する
main(){
CreateFile("test");　　//testというファイルを新規に作成する
FilePuts("123\r\n");　　//123+CR/LFという文字列をファイルに書く
CloseFile(0);　　//ファイルを閉じてディスクをフラッシュする
}
//実行するとtestというファイルが作成される。コマンドモードでtype testで内容を確認
```

char DeleteFile(char *name)

解説:　ファイルを削除する
引数:　name = 削除するファイル名
戻り値:　削除に成功すると0を返す。
　　削除すべきファイルが存在しない場合には1を返す。
サンプル:

```
main(){
DeleteFile("test");　　//testというファイルを削除する
CreateFile("test");　　//testというファイルを新たに作成する
FilePuts("789\r\n");　　//123+CR/LFという文字列をファイルに書く
CloseFile(0);　　//ファイルを閉じてディスクをフラッシュする
}
//実行するとtestというファイルが作成される。コマンドモードでtype testで内容を確認
```

char MoveFile(char *oldname, char *newname)

解説:　ファイル名を変更する
引数:　oldname = 旧ファイル名
　　newname = 新しいファイル名
戻り値:　変更に成功すれば0を返す。失敗した場合には以下のエラーコードを返す。
　　1:旧ファイルが見つからない
　　3:すでに新ファイル名と同名のファイルが存在している。DeleteFileで削除の必要がある
　　4:ディスクの容量が不足している
　　5:内部バッファが確保できない。(メモリー不足)
サンプル:

```
main(){
MoveFile("test", "newtest");　//testというファイル名をnewtestに変更する
}
//実行するとnewtestというファイルが作成される。コマンドモードでdirで確認
```

long GetFinalPos(void)

解説:　ファイルシステムが使用している領域のサイズを求める
引数:　なし
戻り値:　ファイルシステムが使用している領域のサイズを返す。
　　この値が800Kを超えるとそれ以上書き込めない状態になるので
　　デフラグを実行してファイルシステムを最適化する必要がある。
サンプル:

```
main(){
PrNum(GetFinalPos());　//ファイルシステムが使用している領域のサイズを表示する
}
```

void DefragFilesys(int key)

解説:　ファイルシステムを最適化する
　　ファイルの作成と削除を繰り返したりSilentCの編集作業をすると
　　ファイルシステム内に無効領域が増えてきます。
　　最適化することでこの無効領域を削除できますが
　　最適化には数秒から2分程度の時間がかかりますので絶対に
　　最適化中にリセットや電源を落とさないで下さい。
　　ファイルシステムが破壊されてリカバリ処理をしなければ
　　復帰できない状態になる可能性がありますのでくれぐれもご注意下さい。
引数:　key = 不用意に実行されないように16787を指定した時だけ実行する。
戻り値:　なし
サンプル:

```
main(){
PrNum(GetFinalPos());　//ファイルシステムが使用している領域のサイズを表示する
DefragFilesys(16787);　//ファイルシステムを最適化する(関数から戻るまで時間がかかるので注意)
PrNum(GetFinalPos());　//最適化後のサイズを表示する
}
```

ネットワーク通信ライブラリ

char CreateSocket(char mode)

解説: ソケットを作成する
ネットワーク通信のためにソケットを作成します。そべてのネットワーク用の
ライブラリはこのライブラリで与えられるソケットハンドルを指定して呼び出します。
ソケットは最大で7つまで作成できます。

引数: mode = ソケットの種類。
0はUDPソケット、
1はTCP/IP用のソケットを作成。
2以上の値を設定すると指定した値の10倍の秒数のタイマー付き
TCP/IPソケットを作成します。例えば3を指定すると30秒以上通信がなければ
TCP/IPセッションを打ち切ります。
一般的にはUDPなら0でTCP/IPなら1を指定してください。

戻り値: 成功するとソケットハンドラを返します。作成できなかった場合には負の値を返します。

サンプル:

```
main(){
char soc;                 //ソケットハンドラ変数を確保
soc=CreateSocket(0);      //UDPソケットを作成
PrNum(soc);               //ソケットハンドラを表示する
CloseSocket(soc);         //きちんと後始末してメモリーを開放する(メモリーリーク防止の為)
}
```

void CloseSocket(char soc)

解説: ソケットを開放する
CreateSocketで作成したソケットを開放します。内部的にソケットは
メモリーやタイマーなどを確保していますので通信が終了したら必ず開放して下さい。
TCP/IPでコネクション確立中のソケットを開放するとまずコネクションを終了させてから
内部メモリーを解放します。

引数: soc = CreateSocketで取得したソケットハンドル。

戻り値: なし。

サンプル:

```
main(){
char soc;                 //ソケットハンドラ変数を確保
soc=CreateSocket(0);      //UDPソケットを作成
PrNum(soc);               //ソケットハンドラを表示する
CloseSocket(soc);         //きちんと後始末してメモリーを開放する(メモリーリーク防止の為)
}
```

int SendTo(char soc, long ip, int port, char *buf, int size)

解説: UDPデータグラムを送信する。UDP通信は初期ネゴシエーション無しでデータを送りたいときに、
いつでも送信する事ができますが、相手が受け取るかどうかは確実ではありません。
引数: soc = CreateSocketで取得したソケットハンドル。
ip = 相手のIPアドレス(32ビット),
port = 相手のポート番号(16ビット)
buf = 送信するデータへのポインタ、
size = 送信するバイト数。(SilentCでは最大990バイトまでのデータを送信できます)
戻り値: 成功すると送信したバイト数＋8を返します。相手先のIPアドレスが存在しない場合には0を返します。
LAN内で最初に送信する際にはARPの解決が終わっていないので必ず0を返しますので
すこし時間をあけて何度か送信を試みて下さい。
サンプル:

```
main(){
char soc;                   //ソケットハンドラ変数を確保
soc=CreateSocket(0);        //UDPソケットを作成
SendTo(soc,0xc0a80109,123,"ABC",4);   //192.168.1.1の123番ポートに向けてABC\0というデータグラムを送信
CloseSocket(soc);           //きちんと後始末してメモリーを開放する(メモリーリーク防止の為)
}
```

int RecvFrom(char soc, int timeout)

解説: UDPデータグラムを受信する。UDP通信は初期ネゴシエーション無しでデータが送られてきます。
引数: soc = CreateSocketで取得したソケットハンドル。
timeout = 0ならばデータグラムの到着をチェックしてすぐ戻ります。
1以上であれば(指定した値x10ms)の間データグラムの到着を待ちます。
どちらもデータグラムが到着していなければタイムアウトが発生します。
-1を指定するとUDPデータグラムが到着するまでずっと戻りません。
戻り値: 0以上であれば受け取ったUDPデータグラムの長さを返します。
失敗した場合には以下のエラーコードを返します
-1:ソケットが存在しない
-2:タイムアウト
サンプル:(上記のSendToのサンプルと対で動作します。OS-1を二台利用して実験してください)

```
main(){
char soc,*buf;              //ソケットハンドラ変数と入力バッファ変数を確保
soc=CreateSocket(0);        //UDPソケットを作成
Bind(soc,123,0);            //123番ポートを監視するように設定
while(RecvFrom(soc,0)<0);       //UDPデータグラムが到着するまで待つ
buf=GetReceiveBuffer(soc,1);  //受信したUDPデータグラムへのポインタを得る
PrStr(buf);                     //受信した内容を表示する
PrAdrs(GetSenderIP(soc));       //送信元のIPアドレスを表示する
PrNum(GetSenderPort(soc));  //送信元のポート番号を表示する
MemoryFree(buf);            //受信したデータグラムが格納されている領域を開放
CloseSocket(soc);           //きちんと後始末してメモリーを開放する(メモリーリーク防止の為)
}
```

char *GetReceiveBuffer(char soc, char release)

解説: 受信したデータが格納されているバッファのポインタを返します。
引数: soc = CreateSocketで取得したソケットハンドル。
release = 1ならば次のデータを受信するための内部的な初期化を行います。
0ならば初期化を行わずに現在のデータを保持します。この場合は
再度このライブラリを呼び出すと同じポインタが得られます。
戻り値: データが到着していない場合には0を返します。データが到着していれば格納されて
いる領域のポインタを返します。この領域は使用が終わったら必ずMemoryFreeを
呼び出して開放する必要がありますのでご注意下さい。
サンプル:前項のRecvFromのサンプルをご覧下さい。

long GetSenderIP(char soc);
解説: 受信したデータの発信元のIPアドレスを得ます
引数: soc = CreateSocketで取得したソケットハンドル。
戻り値: データが到着していない場合には0を返します。
データが到着していれば32bitの発信元のIPアドレスを返します。
サンプル:上記のRecvFromのサンプルをご覧下さい。

int GetSenderPort(char soc)

解説: 受信したデータの発信元のポート番号を得ます
引数: soc = CreateSocketで取得したソケットハンドル。
戻り値: データが到着していない場合には0を返します。
データが到着していれば16bitの発信元のポート番号を返します。
サンプル:前項のRecvFromのサンプルをご覧下さい。

char Bind(char sochandle, int port, char maxsoc)

解説: ソケットにポート番号を設定します。このポート番号は送信の際には
発信元のポート番号として相手に伝えられます。また受信の際には監視する
ポート番号となります。
引数: soc = CreateSocketで取得したソケットハンドル。
port = 1-65535までのポート番号を指定します。
maxsoc = TCP/IP通信でAcceptで接続を待ち受ける際に
最大何個のセッションまで生成するかを指定します。
SilentCでは一つのセッションの処理だけしか対応できませんので
1を指定して下さい。UDP通信の際には無視されます。
戻り値: 設定に成功すると1を返します。ソケットが存在しない場合には0を返します。
サンプル:前項のRecvFromのサンプルをご覧下さい。

char Connect(char soc, long ip, int port)

解説: TCP/IP通信で相手先とコネクションを確立します。(クライアントモード)
TCP/IP通信では通信を開始する前に必ずコネクションを確立する必要があります。
引数: soc = CreateSocketで取得したソケットハンドル。
ip = 相手のIPアドレス(32ビット),
port = 相手のポート番号(16ビット)
戻り値: 成功すれば1を返します。失敗した場合には以下のエラーコードを返します。
-1:ソケットが存在しない、-2:タイムアウト、-3:コネクション確立中に相手から切断された
サンプル:(クライアント側のサンプル)

```
main(){
char soc,*buf;                  //ソケットハンドラ変数と入力バッファ変数を確保
soc=CreateSocket(1);            //TCP/IPソケットを作成(タイムアウト無し)
Connect(soc,0xc0a80109,123);//192.168.1.9の123番ポートに対してコネクションを確立する
Write(soc,"ABC",4);             //ABC\0という4文字をTCP/IPで送信する
WaitWriteComplete(soc);         //相手が受信した事を確認する
Read(soc,-1);                   //相手からのデータを待つ
buf=GetReceiveBuffer(soc,1);  //受信したデータへのポインタを得る
PrStr(buf);MemoryFree(buf);   //受信したデータを表示した後にその領域を開放
CloseSocket(soc);               //コネクションを切断して内部メモリーを開放する
}                       //実際にはタイムアウトの処理などをきちんと行う必要があります
```

int Write(char soc, char *buf, int len)

解説: TCP/IPでデータを送信する。
引数: soc = CreateSocketで取得したソケットハンドル。
buf = 送信するデータへのポインタ
size = 送信するバイト数 (SilentCでは最大で970バイトのデータを送信できます)
戻り値: 成功すると送信したバイト数を返します。失敗した場合には以下のエラーコードを返します
-1:ソケットが存在しない、-2:タイムアウト、-3:送信中に相手から切断された
サンプル:上記のConnectのサンプルをご覧下さい

int Read(char soc,int timeout)

解説: TCP/IPでデータを受信する。
引数: soc = CreateSocketで取得したソケットハンドル。
timeout = 0ならばデータの到着をチェックしてすぐ戻ります。
1以上であれば(指定した値x10ms)の間データの到着を待ちます。
どちらもデータが到着していなければタイムアウトが発生します。
-1を指定するとデータが到着するまでずっと戻りません。
戻り値: 0以上であれば受け取ったデータの長さを意味します。
失敗した場合には以下のエラーコードを返します
-1:ソケットが存在しない、-2:タイムアウト、-3:送信中に相手から切断された
サンプル:上記のConnectのサンプルをご覧下さい

char Accept(char soc, int timeout)

解説: TCP/IPのコネクションを待ち受ける(サーバーモード)
特定のポートを監視しながらコネクション要求を待ちます。

引数: soc = CreateSocketで取得したソケットハンドル。
timeout = 0ならばコネクション状態をチェックしてすぐ戻ります。
1以上であれば(指定した値x10ms)の間コネクション要求を待ちます。
指定した時間内にコネクションが確立されなければタイムアウトが発生します。
-1を指定するとコネクションが確立するまでずっと戻りません。

戻り値: 0以上であればコネクションが確立された事を意味します。戻された値は
新たなソケットハンドルですので保存してその後のTCP/IP通信に使用します。
つまりAcceptでTCP/IP通信を待ち受けるためには待ち受け用のルートソケットと
相手からコネクション要求があった時に新たに内部的に生成されるデータソケット
の2種類のソケットハンドルを扱います。ルートソケットは最初にCreateSocketして
Bindで待ち受けポート番号を設定してからAcceptで待ち受けるまでが出番です。
相手からのコネクション要求があればAcceptは新たにソケットを生成してAcceptの
戻り値としてユーザーに知らせます。本来であればこの新たなソケットを処理する
スレッドを生成する事で複数のコネクションに対応できるように考えられたしくみ
です。SilentMoonのシステムコールを利用してC言語で記述する場合にはこの機構を
利用して複数のセッションを受け付けるプログラムが開発可能ですが、SilentCで
記述する際には一つのコネクションしか対応できません。こうしたプログラムスタイルを
ソケットプログラミングと呼びます。
失敗した場合には以下のエラーコードを返します
-1:ソケットが存在しない、-2:タイムアウト、-3:送信中に相手から切断された

サンプル:(前項のConnectのサンプルと対で動作します。OS-1を二台利用して実験してください)

```
main(){                          //サーバー側のサンプル(先に実行してください)
char soc,newsoc,*buf;            //ソケットハンドラ変数2つと入力バッファ変数を確保
soc=CreateSocket(1);             //TCP/IPソケットを作成(タイムアウト無し)
Bind(soc,123,0);                 //123番ポートを監視するように設定
for(;;){                         //くりかえし接続を受け付けるループの開始
 newsoc=Accept(soc,-1);          //コネクションを待ち続ける(タイムアウト無し)
 Read(newsoc,-1);                      //データを受信する(タイムアウト無し)
 buf=GetReceiveBuffer(newsoc,1);  //受信したデータへのポインタを得る
 PrStr(buf);MemoryFree(buf);     //受信した内容を表示し領域を開放する
 Write(newsoc,"123",4);          //123\0という4文字を送信する
 WaitWriteComplete(newsoc);      //相手が受信した事を確認する
 CloseSocket(newsoc);            //コネクションを切断して内部メモリーを開放する
 }                               //くりかえし接続を受け付けるループの終端
}                        //実際にはタイムアウトの処理などをきちんと行う必要があります
```

int GetNetLine(char soc, char *buf, char size, char func)

解説: TCP/IPのでCR/LFで区切られた1行を受信します。
POPサーバーやSMTPサーバーなどのCR/LFで区切られた1行単位の
データを受信するための専用ライブラリです。内部にデータキャッシュを
持ちますので初期化と終了処理が必要になります。

引数: soc = CreateSocketで取得したソケットハンドル。
buf = 受信する1行を格納するバッファのポインタ
size = 上記の受信バッファのサイズ
func = 動作モードを指定します。
0ならばライブラリを初期化します。
1ならばライブラリの内部のキャッシュを開放します。
それ以外の値であれば指定した数値の秒数のタイムアウトを設定して1行受信するまで待ちます。

戻り値: 1以上であれば受信した1行の文字数を返します。
終端のCR/LFは取り除かれて代わりにヌル文字がセットされます。
0は初期化と開放した際に返されます。
失敗した場合には負の値を返します

サンプル:

```
main(){
char soc,*buf;                  //ソケットハンドラ変数と入力バッファ変数を確保
long ip;                        //IPアドレスを格納する変数を確保
buf=MemoryAlloc(64);            //1行入力バッファを確保
soc=CreateSocket(1);            //TCP/IPソケットを作成(タイムアウト無し)
ip=GetHostByName("smtpserver.jp");    //自分のプロバイダのSMTPサーバーのIPアドレスを調べる
GetNetLine(0,0,0,0);            //ライブラリの初期化
Connect(soc,ip,25);             //SMTPサーバーの25番ポートに対してコネクションを確立する
GetNetLine(soc,buf,64,2);       //SMTPサーバーから1行入力(20秒タイムアウト)
PrStr(buf);MemoryFree(buf);     //受信したデータを表示した後にその領域を開放
CloseSocket(soc);               //コネクションを切断して内部メモリーを開放する
GetNetLine(0,0,0,1);            //ライブラリをきちんと開放
}
```

char CheckWriteComplete(char soc)
char WaitWriteComplete(char soc)

解説: CheckWriteCompleteはTCP/IPで送信したデータが相手に送られたかどうかを調べます。
WaitWriteCompleteはTCP/IPで送信したデータが相手に確実に届くまで待ちます。
引数: soc = CreateSocketで取得したTCP/IPソケットハンドル。
戻り値: 1であれば送信したデータは確実に先方で受け取られています。
0ならばまだ受け取られていません。
失敗した場合には以下のエラーコードを返します
-1:ソケットが存在しない
-3:送信中に相手から切断された
サンプル:前項のサンプルを参考にして下さい。

int GetDefMtu(char soc, int size)

解説: TCP/IP通信の際のデフォルトのデータサイズ(MTU)を調べるあるいは設定します。
TCP/IP通信の際のIPヘッダーには送受信する最大のデーターサイズMTUを
設定する部分がありますが、その値を設定するためのライブラリです。
通信の相手先は通常はMTU以下のデータサイズでデータを送信する決まりになっています。

SilentMoonのデフォルトのMTUは1454ですが、SilentCが動作している時は全体のRAM容量が
少ないために本来は1500バイト確保しなければならないネットワーク用のバッファも
SilentCが使用していますので、MTUは970以下に設定しなければなりません。
MTUを超えるサイズのデータを受け取っても破棄されますので通信できません。
ソケットを作成したらこのライブラリでMTUを970以下に設定してから通信を開始して下さい。

引数: soc = CreateSocketで取得したTCP/IPソケットハンドル。
size = 0を指定すると何も値を設定しません。
0以外の値を指定するとその値をデフォルトのデータサイズとして設定します。

戻り値: 現在設定されているデフォルトのデータサイズを返します。sizeに0を指定して
呼び出す事で現在設定されている値を調べる事が出来ます。

サンプル:

```
main(){
char soc;                   //ソケットハンドラ変数と入力バッファ変数を確保
soc=CreateSocket(1);        //TCP/IPソケットを作成(タイムアウト無し)
PrNum(GetDefMtu(soc,0));//現在のMTUを表示する
GetDefMtu(soc,970);         //MTUを970に設定する
PrNum(GetDefMtu(soc,0));//新たなMTUを表示する
CloseSocket(soc);           //ソケットを開放して内部メモリーを開放する
}
```

long GetHostByName(char *name)

解説: サーバー名を基にしてDNSを調べて32ビットのIPアドレスに変換します。
OS-1がDNSにアクセスできる状態でなければ変換できません。

引数: name = サーバー名へのポインタ。

戻り値: 成功すれば名前を解決して32ビットのIPアドレスを返します。
DNSへの接続に失敗したりその名前のサーバーが存在しない場合には0を返します。

サンプル:

```
main(){
PrAdrs(GetHostByName("www.domain.jp"));          //www.domain.jpのIPアドレスを調べて表示する
}
```

CGI関連ライブラリ

char *ReplaceCgi(char *buf, int size)

解説: 内部的なIPアドレスやポートの値を文字列に変換するライブラリです。
OS-1に内蔵されているHTTPサーバー内のCGI機能を切り出してきたライブラリです。
利用価値はあまりないかもしれませんがオマケだと考えて下さい。

引数: buf = CGIシンボルを格納した領域へのポインタ。
size = 上記の領域のサイズ。

戻り値: 最後のCGIシンボルを置換した直後の位置のポインタを返します。
CGIシンボルは$で始まる文字列です。$を#にすると負論理になります。
SilentCで利用可能なCGIシンボルは以下の通りです。
このライブラリで置換する場合にはシンボルを含めて指定の文字数に
なるように空白を付加してください。

$IP	IPアドレス	15文字
$GW	ゲートウェー	15文字
$Sub	サブネットマスク	15文字
$DNS	DNSのIPアドレス	15文字
$Dhcp	DHCPクライアント有効	checked または 空白7文字
$DhcpS	DHCPサーバー有効	checked または 空白7文字
$User	ユーザープログラム有効	checked または 空白7文字
$Auto	イーサ自動設定	checked または 空白7文字
$Speed	イーサスピード100M	checked または 空白7文字
$Dup	イーサ全二重有効	checked または 空白7文字
$Http	HTTPサーバー有効	checked または 空白7文字
$Tftp	TFTPサーバー有効	checked または 空白7文字
$StartFile	スタートアップファイル	40文字
$AutoRun	オートスタート	checked または 空白7文字
$NoBreak	ブレーク禁止	checked または 空白7文字
$Telnet	telnetサーバー有効	checked または 空白7文字
$Port	telnetポート番号	5文字
$Term	シリアルターミナル有効	checked または 空白7文字
$PDIRE0から$PDIRE4	ポートEの入出力の方向	checked または 空白7文字
$PCHKE0から$PCHKE4	ポートEの状態	checked または 空白7文字
$PORTE0から$PORTE4	ポートEの状態	1 または 0 (1文字)
$PDIRT4から$PDIRT7	ポートTの入出力の方向	checked または 空白7文字
$PCHKT4から$PCHKT7	ポートTの状態	checked または 空白7文字
$PORTT4から$PORTT7	ポートTの状態	1 または 0 (1文字)

$PDIRJ0から$PDIRJ7	ポートJの入出力の方向	checked または 空白7文字
$PCHKJ0から$PCHKJ7	ポートJの状態	checked または 空白7文字
$PORTJ0から$PORTJ7	ポートJの状態	1 または 0 (1文字)
$PDIRG0から$PDIRG6	ポートGの入出力の方向	checked または 空白7文字
$PCHKG0から$PCHKG6	ポートGの状態	checked または 空白7文字
$PORTG0から$PORTG6	ポートGの状態	1 または 0 (1文字)
$PDIRH0から$PDIRH6	ポートHの入出力の方向	checked または 空白7文字
$PCHKH0から$PCHKH6	ポートHの状態	checked または 空白7文字
$PORTH0から$PORTH6	ポートHの状態	1 または 0 (1文字)
$PDIRL0から$PDIRL4	ポートLの入出力の方向	checked または 空白7文字
$PCHKL0から$PCHKL4	ポートLの状態	checked または 空白7文字
$PORTL0から$PORTL4	ポートLの状態	1 または 0 (1文字)

サンプル:

```
main(){
char *buf=MemoryAlloc(64);          //作業用の領域を確保してbufにセット
StrCpy(buf,"$IP              ");    //領域に文字列をセットしてIPアドレスを調べる。15文字の空白を付加する
ReplaceCgi(buf,StrLen(buf));        //$IPをIPアドレス形式に変換する
PrStr(buf);                         //変換結果を表示する
StrCpy(buf,"$PORTH0$PCHKH0"); //ポートH0の値を調べるために文字列をセット。
ReplaceCgi(buf,StrLen(buf));        //ポートの値を変換する
PrStr(buf);                         //変換結果を表示する
StrCpy(buf,"#PORTH0#PCHKH0"); //ポートH0の値を調べるために文字列をセット(負論理)
ReplaceCgi(buf,StrLen(buf));        //ポートの値を変換する
PrStr(buf);                         //変換結果を表示する
MemoryFree(buf);                    //確保した領域は必ず解放して終わること。
}
//SW1を押しながら実行すると結果が変化しますのでお試し下さい。
```

void DoCgi(char *ptr)

解説: CGIシンボルへ値を代入します。
利用価値はあまりないかもしれませんがオマケだと考えて下さい。

引数: buf = CGIシンボル=値　の文字列へのポインタ。
&で連結すると一度に複数のCGIシンボルに代入できます。
代入の際には$をつけないので注意してください。

戻り値: なし。

サンプル:

```
DoCgi("PORTH4=0&PORTH5=0");          //LED1とLED2のポートに0を代入
//LED1とLED2が光ります。ダイレクトモードで実行してください

DoCgi("PORTH4=1&PORTH5=1");          //LED1とLED2のポートに0を代入
//LED1とLED2が消えます。ダイレクトモードで実行してください
```

タイマーライブラリ

void Sleep(int time)

解説:　実行を休止します。休止中は他のスレッドに実行時間を割り当てます。
forなどで無駄なループを繰り返して時間を費やす事は避けてください。
システムスレッドはバックグラウンドで様々な処理を行っていますので
必ずSleepを利用してシステムスレッドに実行時間を与えて下さい。

引数:　time = 休止する時間を指定します。指定した値 x 10msの休止時間になります。
1秒休止するならば100を指定します。

戻り値:なし。

サンプル:

```
Sleep(500);　　//5秒間休止する。ダイレクトモードで
```

void SystemSleep(void)

解説:　実行をシステムに戻します。自発的なスレッド切り替えだと考えて下さい。
スイッチやポートの状態を監視するループに挿入すると全体の動作が高速になります。

SystemSleep()によってどうして高速になるか説明します。SilentMoonの
スレッド切り替えは10ms単位で登録されているスレッドに対して順番に
実行時間を割り当てます。割り当て後に10msが経過するとタイマー割り込みが
発生してシステムに制御が戻り、次のスレッドを実行します。つまり
何もしなければ10msは自分のスレッドが実行されるという事です。

ポートなどのチェックはメモリーの値をチェックするだけなので100μS程度の
短時間で終了してしまいます。しかしスレッド切り替えは10ms後ですのでまだ
かなりの時間が残っています。この残っている時間をシステムに戻す事で
他のスレッドの実行に割り当てる事ができるわけです。

キーが押されるまで待つだけの仕事であればすぐにSystemSleep()でシステムに
制御を戻すように心がけて下さい。

引数:　なし。
戻り値:　なし。

サンプル:

```
while(*0x258&1)SystemSleep();　　//SW1キーが押されるまで待つ(ダイレクトモードでどうぞ)
//同時にOS-1のWebサーバーにアクセスしても殆ど遅くなっていない事がわかります

while(*0x258&1);　　//SW1キーが押されるまで待つ(ダイレクトモードでどうぞ)
//同時にOS-1のWebサーバーにアクセスするとすこし遅くなっているのがわかるはずです
```

char CreateTimer(char timer, int time, char *func)

解説:　タイマーを作成します。
SilentCのタイマーは10msでカウントダウンされます。
時間が来ると指定した関数を実行させる事も可能です。

引数:　timer = タイマーハンドルを指定します。
0を指定すると新たにタイマーを作成してそのハンドルを返します。
0以外を指定するとそのハンドルのタイマーの設定を変更します。

time = カウントダウンの時間を10ms刻みで設定します。
0を指定するとtimerで指定したハンドルのタイマーを開放します。
すでに作成したタイマーが不要になった場合にはtimeに0をセットして
CreateTimerを呼び出す事でDestroyTimerとして動作します。

func = タイマーが0になった時にこの文字列で指定した関数を実行します。
0を指定すると単なるカウントダウンタイマーとして動作します。

戻り値:　成功した場合には0ではないタイマーハンドルを返します。
タイマーをこれ以上作成できないときには0を返します。
カウントダウンが終了してもタイマー自体は残りますので再度
タイマーハンドルを指定してカウントダウンの時間を再設定できます。

サンプル:

```
main(){
char th;                                //タイマーハンドル変数を用意する
th=CreateTimer(0,50,"PrChar('*');");//0.5秒に1度Aを出力するタイマーを作成
while(*0x258&1)SystemSleep();       //SW1が押されるまで無限ループする。
CreateTimer(th,0,0);                    //タイマーを開放して内部のメモリーを解放する
}
```

int GetTimerCount(char timer)

解説: タイマーのカウントダウンの値を調べます。

引数: timer = タイマーハンドルを指定します。

戻り値: 有効なタイマーハンドルを指定した場合には現在のカウントダウンの値を返します。
無効なタイマーハンドルを渡すと0を返します。

サンプル:

```
main(){
char th;                    //タイマーハンドル変数を用意する
int time;                   //タイマーの値を格納する変数を準備
th=CreateTimer(0,1000,0);   //10秒でカウントダウンするタイマーを作成
for(;;){                    //タイマーの値を表示するループ開始
time=GetTimerCount(th);     //タイマーのカウントダウン値を取得する
if(time==0)break;           //タイマーが0ならループを抜ける
PrNum(time);Sleep(100);}    //タイマーの値を表示して1秒待つ
CreateTimer(th,0,0);        //タイマーを開放して内部のメモリーを解放する
}
```

イベントライブラリ

char CreateEvent(int port, char mask, char opr, char *func)

解説: ポートイベントを作成します。ポートの値を監視してある条件が成立
すると指定された関数を実行します。

引数: port = 監視するポートのアドレスを指定します。

val = イベントを発生させる判断のための値を指定します。
ビット判断の場合にはマスク値として、値の判断の際には比較値として使われます。

opr = ポートの状態がどうなれば関数を実行するかを指定します。

1はポートの値とvalをANDしてその結果が0になれば関数を実行します。
2はポートの値とvalをANDしてその結果が1になれば関数を実行します。
3はポートの値とvalをANDして今までの値と変化すると関数を実行します。

4はポートの値とvalを比較してポートの値が大きくなったら関数を実行します。
5はポートの値とvalを比較してポートの値が小さくなったら関数を実行します。
6はポートの値とvalを比較して等しくなったら関数を実行します。

func = 条件が成立したときにこの文字列で指定した関数を実行します。

戻り値:成功した場合には0ではないポートイベントハンドルを返します。
ポートイベントをこれ以上作成できないときには0を返します。

サンプル:

```
main(){
char ph;                                    //ポートイベントハンドル変数を用意する
ph=CreateEvent(0x258,2,1,"*0x260^=128");    //SW2が押される度にリレーをON/OFFするイベントを生成
while(*0x258&1)SystemSleep();               //SW1が押されるまで無限ループする。
KillEvent(ph,0,0);                          //ポートイベントを開放して内部のメモリーを解放する
}
//SW2を押すとリレーが動作してSW1を押すと終了する
```

char KillEvent(char handle)

解説: ポートイベントを開放します。内部で確保していたメモリーも解放されます。

引数: handle = 開放するポートイベントハンドルを指定します。

戻り値: 成功した場合にはhandle自身を返します。
対象のハンドルが存在しない場合には0を返します。

サンプル: 上記のCreateEventのサンプルを参照してください。

int UserDriver(char vec, int arg)

解説: ユーザーが作成したネイティブコードを呼び出します。高速なデバイスなどの
デバイスドライバをSilentCから呼び出す事ができます。

引数: vec = ベクター番号
ジャンプテーブルの何番目のドライバを呼び出すか指定します。
準備として0xf400番地から3バイトずつJMP命令を並べておきます。
例えばvecに0を指定するとJSR 0xf400を実行します。
同様に1だとJSR 0xf403を2だとJSR 0xf406を実行します。
ユーザードライバを作成する際には先頭に各ドライバへのジャンプ
命令を並べておく事になります。

arg = ユーザードライバを呼び出す際にDレジスタにセットする引数を指定します。
SilentC内で確保したバッファ領域や配列のポインタを渡して使う事になります。

戻り値: ユーザードライバから戻る際のDレジスタの値を関数の値として返します。

サンプル:
SilentCには起動時のユーザードライバの自動書込みモードがあります。
UserDriver.binというファイルが存在していると0xf400から0xf7ffの1024バイト
のフラッシュメモリーにプログラムしてからファイルを消去します。この機能により
ユーザーのドライバがOS-1にセットされてSilentCから呼び出しが可能になります。
このUserDriver.binの開発手法に関しては別冊のOS-1ネイティブプログラミングマニュアル
で解説します。

SilentCの言語仕様の解説

SilentCの言語仕様の解説は現在執筆中です。より詳しい解説に関しては恐れ入りますがもう少々お待ち下さい。それまではサイレントシステムのサイトからダウンロードしたサンプルプログラムをご覧になると、SilentCの動作をご理解頂けると思います。

配列はポインタ配列のみのサポートになります。予め配列の要素数と変数タイプにより算出されたサイズの領域をMemoryAllocで確保してポインタ変数に代入します。そしてポインタ変数[添え字]で確保した領域を配列としてアクセスします。1次元配列のみのサポートになります。

構造体はstructで構造体のメンバを列挙して宣言します。次に構造体配列の要素数と構造体のサイズにより算出されたサイズの領域をMemoryAllocで確保してchar型のポインタ変数に代入します。そしてそのポインタ変数->構造体のメンバ名という記述で構造体としてアクセスします。次の構造体へのポインタを求める際にはsizeof(構造体名)をポインタ変数に加算して下さい。

開発時にはなるべくシリアルで開発をお願いします。動作中にメモリー不足になるとリセット以外では復帰できないのでエラーメッセージではなく起動時に表示されるSilentCのバージョン情報が表示されます。動作中にSilentCの初期リセット画面に戻った際にはメモリー不足だと考えて下さい。

再帰的呼び出しは3ネスティング程度でメモリーが不足しますので注意して下さい。関数内での文字列定数は関数を抜けると消滅します。関数から文字列定数を戻り値で返しても戻り先では利用できませんのでご注意下さい。グローバル変数はとにかく最小限にして下さい。

SilentCを使えば気軽に開発できますがメモリーを多く使う用途の場合にはネイティブなプログラムを利用したほうが得策です。ネイティブプログラミングに関しても本書の後半で解説しますのでもう少々お待ち下さい。