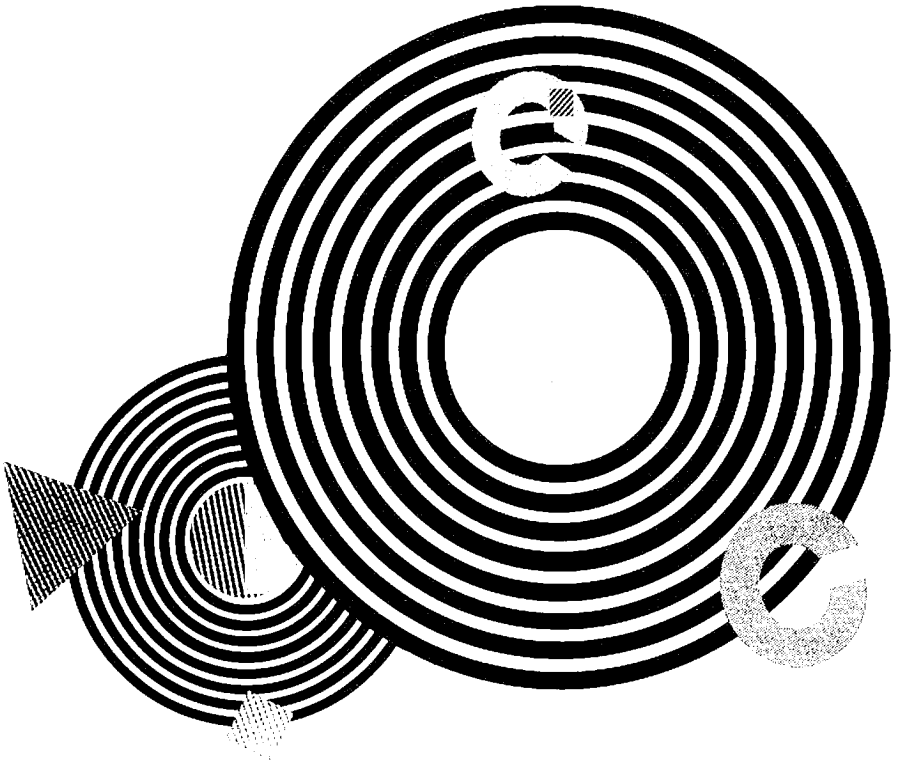
MSX 2 DOS2

日本語MSX-DOS 2専用

MSX-C Ver.1.2

エムエスエックス シー バージョン.1.2

ユーザーズマニュアル



ASCII
ASCII CORPORATION

はじめに

この度は，MSX-C Ver.1.2 をお求め下さいましてありがとうございます．

MSX-C Ver.1.2 はC言語でかかれたソースプログラムを処理し，アセンブラ・ソースプログラムを生成するコンパイラです．

新たに，MSX-DOS バージョン2 をサポートすることにより，漢字のメッセージを表示したり，漢字のデータを処理したり，階層ディレクトリなどの機能を使用することができます．そのため，標準ライブラリが強化され，いっそう使いやすくなっています．

なお，本ソフトウェアにはMSX-DOS2，スクリーンエディタ，MSX・M-80，MSX・L-80，LIB-80，CREF-80 は含まれておりません．MSX-C Ver.1.2 を使用してプログラム開発を行なう場合にはこれらのソフトウェアが必要となりますので， お持ちでない方は「MSX-DOS2 TOOLS」をお買い求め下さい．また，プログラム開発を行なう上でデバッグを効率的に行なうために，MSX用シンボリックデバッガ「MSX-S BUG2」が，ライブラリパッケージとして「MSX-C Library」がそれぞれ別売にて用意されています．

本ソフトウェアをご使用になる前に，添付の「ソフトウェア使用承諾契約書」をよくお読みいただき，ご確認のうえ，添付のユーザー登録カードにご記入し，弊社までご返送下さい．ご返送をもって承諾契約書にご同意いただいたものといたします．

この「ユーザー登録カード」をご返送いただけない場合には，弊社といたしましては所定のアフターサービスをいたしかねますので，よろしくご了承のほどお願い申し上げます．

本パッケージには以下のものが含まれています．

- MSX-C Ver.1.2 システムディスク　　1枚
  (3.5-1DD フロッピーディスク)

- MSX-C Ver.1.2 ユーザーズマニュアル　　1冊

ご注意

(1)このソフトウェアならびにマニュアルを賃貸業に使用することを禁じます．また，このソフトウェアやマニュアルの一部または全部を無断でコピーすることはできません．

(2)このソフトウェアは，個人使用以外の目的でコピーすることはできません．

(3)このマニュアルに記載されている事柄は，将来予告なく変更することがありますが，当社に登録されている方には案内をお送りします．

(4)製品の内容については万全を期しておりますが，製品の内容についてのご不審や，誤り，マニュアルの記載もれなど，お気付きのことがございましたら，マニュアルの巻末の「お問い合わせ」についての要領でお送り下さい．

(5)このソフトウェアを運用した結果の影響については，(4)項にかかわらず，責任を負いかねますのでご了承下さい．

(6)このソフトウェアの運用は個人使用に限ります．このソフトウェアを使用して開発したプログラムを販売・頒布する場合は，弊社と使用許諾契約を結んでいただきます．
使用許諾契約に関するお問い合わせは

株式会社アスキー MSX 推進部
TEL．03-486-8346(祝祭日を除く月～金)
10：00～12：00，13：00～17：00

までお願いいたします．

目 次

序章 本マニュアルの構成 *9*

第１章 MSX-Cコンパイラの概要 *11*

1.1 MSX-Cコンパイラの概要 *11*
1.1.1 MSX-Cコンパイラ *11*
1.1.2 MSX-Cの特徴 *11*
1.1.3 標準カーネルの諸機能 *13*
a) 引数(アーギュメント, argument) *13*
b) 標準入出力ファイルの割り当て *14*
1.2 ソースファイルの作成からプログラムの実行まで *15*
1.2.1 エディタ *15*
1.2.2 C言語ソースからアセンブラ・ソースへ *15*
1.2.3 実行可能プログラムの生成(アセンブル, リンクの意義) *16*
1.3 MSX-Cの開発環境(開発環境の構築) *17*
1.4 サンプル・プログラム *20*
1.4.1 サンプル・プログラム「HEAD」の仕様について *20*
a) 概要 *20*
b) オプションスイッチ *20*
c) I/Oリダイレクション, パイプ機能 *21*
d) エラーチェック, ワイルドカードなど *21*
1.4.2 ソースファイルの作成上の留意点 *21*
a) #プリプロセッサ *21*
b) コマンドへの引数 *21*
c) 関数の宣言 *22*

第２章 MSX-Cコンパイラの操作 *24*

2.1 バッチファイルについて *25*
2.2 各コマンドの操作 *26*
2.2.1 CF (パーサ) *26*
2.2.2 FPC (パラメータチェッカ) *32*
2.2.3 CG (コードジェネレータ) *36*
2.2.4 アセンブルとリンク *38*

第３章 MSX-Cによるプログラミング *40*

3.1 標準C(V7UNIX-C)との言語仕様上の相違点 *40*

3.1.1 #プリプロセッサ文による制御 *40*
a) 再帰モードと非再帰モード *41*
b) PDP-11 コンパチブルモードと MSX-C デフォルトモード *42*
c) # if 文の非サポート *43*
d) コンパイラの内部的な制御 *44*
3.1.2 関数の宣言 *47*
a) 暗黙の宣言 *47*
b) 可変パラメータ関数と固定パラメータ関数 *47*
c) 可変パラメータ関数の定義 *47*
3.1.3 その他 *51*
a) float, double, long の非サポート *51*
b) ビットフィールドの非サポート *51*
c) 定数式の制限 *51*
d) 構造体，共用体のメンバ名 *51*
e) ポインタ，整数間の相互代入 *52*
3.2 アセンブリ言語とのリンク *54*
3.2.1 パラメータの渡し方 *54*
3.2.2 値の返し方 *57*
3.2.3 アセンブラ，C のルーチンの相互呼び出し *57*
a) 呼び出し側の処理 *57*
b) 呼び出される関数側の処理 *57*
3.2.4 識別子の有効文字数 *59*
3.2.5 外部シンボル *60*
3.3 標準 C のソースファイルの移植について *60*
3.3.1 バックスラッシュ('\') *60*
3.3.2 # include <stdio.h> *61*
3.3.3 char 型と int 型 *61*
3.3.4 可変パラメータ関数への引数 *62*
3.3.5 関数の宣言(関数の前方参照) *63*
3.4 MSX-C Ver.1.1 と Ver.1.2 の相違点 *63*
3.4.1 コマンドの変更点 *63*
a) CF 固有の変更点 *64*
b) CG 固有の変更点 *65*
c) FPC 固有の変更点 *65*
d) MX 固有の変更点 *65*
3.4.2 標準ライブラリの変更点 *66*
3.5 漢字処理の方法 *67*
3.5.1 MSX での漢字の内部表現 *67*

3.5.2 漢字モードの区別 *69*
3.5.3 漢字の処理 *70*

第4章 MSX-C 標準ライブラリ *72*
4.1 MSX-C の標準的な型 *72*
4.1.1 BOOL 型 *72*
4.1.2 FCB 型 *73*
4.1.3 FD 型 *73*
4.1.4 FIB 型 *74*
4.1.5 FILE 型 *74*
4.1.6 LONG 型 *74*
4.1.7 size_t 型 *74*
4.1.8 STATUS 型 *75*
4.1.9 TINY 型 *75*
4.1.10 VOID 型 *75*
4.1.11 XREG 型 *76*
4.2 ヘッダファイル *76*
4.2.1 ヘッダファイルの分割 *76*
4.2.2 ヘッダファイルのインクルードの方法 *78*
4.2.3 ヘッダファイルの内容 *79*
a) bdosfunc.h *80*
b) conio.h *80*
c) ctype.h *80*
d) direct.h *80*
e) io.h *80*
f) malloc.h *81*
g) memory.h *81*
h) process.h *81*
i) setjmp.h *81*
j) stdio.h *82*
k) stdlib.h *82*
l) string.h *83*
m) type.h *83*
4.3 標準ライブラリ関数概要 *84*
4.3.1 ファイル入出力関数 *84*
a) 高水準入出力関数 *85*
b) 高水準入出力関数のバッファリング *86*
c) 低水準入出力関数 *87*

4.3.2 文字列および文字処理関数 *87*
4.3.3 メモリ管理関数 *88*
a) 低水準メモリ管理関数 *88*
b) 高水準メモリ管理関数 *89*
4.3.4 ディレクトリ関数 *89*
4.3.5 プログラム操作関数 *89*
4.3.6 キーボード，I/O 関数 *89*
4.3.7 機械語，MSX-DOS ファンクションコールサポート関数 *90*
4.3.8 メモリ操作関数 *90*
4.3.9 汎用関数 *90*
4.4 UNIX 標準 C ライブラリとの相違点 *90*
4.5 ライブラリの作成と保守 *91*
4.5.1 MSX-C ライブラリの構成 *91*
4.5.2 ライブラリ保守支援ツール MX について *93*
4.5.3 専用ライブラリの作成 *94*
a) すべての関数をリンクするライブラリ *95*
b) 必要な関数だけをリンクするライブラリ *97*
4.5.4 標準ライブラリの再作成 *103*

第5章 MSX-C コンパイラの応用 *107*

5.1 Disk-BASIC 環境でのマシン語ルーチンの作成 *107*
5.1.1 Disk-BASIC 環境と USR 関数 *107*
a) Disk-BASIC のメモリ・マップ *107*
b) マシン語領域の確保と USR 関数 *107*
5.2 C 言語によるソース・プログラムの作成 *109*
5.2.1 USR 関数の引数の受取り方 *109*
5.2.2 システム・コールの使い方 *111*
5.2.3 値の返し方 *113*
5.3 C 言語ソース・ファイルから BLOAD ファイルまで *113*
5.3.1 L80 だけによる BLOAD ファイルの作成 *113*
5.3.2 MSX-DOS2 TOOLS の BSAVE コマンドを使う *114*
5.4 サンプル・プログラムについて *115*
5.5 ROM 化プログラムの作成 *115*
5.5.1 ROM 化プログラム作成の一般的原則 *115*
a) ライブラリの引用について *115*
b) 初期設定の部分 *116*
c) IN/OUT *117*
d) 絶対番地の参照 *118*

e) 割り込み処理 *119*
f) 定数テーブルの扱い *120*
5.5.2 ROM 化プログラムの作成例 *120*

第 6 章 標準ライブラリ関数リファレンス *122*

abs *123*
alloc *124*
atoi *126*
bdos, bdosh *127*
bios *129*
call, calla *130*
callxx *131*
chdir *132*
clearerr *133*
close *134*
creat *135*
eof *136*
execl, execlp *137*
execv, execvp *139*
exit, _exit *141*
expargs *142*
fclose, fcloseall *144*
feof *146*
ferror *148*
fflush *149*
fgets *151*
fileno *152*
flushall *153*
fopen *154*
fprintf *156*
fputs *158*
fread *159*
free *161*
fscanf *163*
fsetbin *165*
fsettext *167*
fwrite *168*
getc, getchar *170*
getch *171*
getche *173*
getcwd *174*
getenv *175*
gets *177*
inp *178*
isalnum *179*
isalpha *181*
isatty *182*
iscntrl *183*
isdigit *184*
iskanji *185*
iskanji2 *187*
islower *189*
isspace *190*
isupper *191*
isxdigit *192*
kbhit *194*
longjmp *195*
max *197*
memcpy *198*
memset *199*
min *201*
mkdir *202*
movmem *203*
open *204*
outp *206*
printf *207*
putc, putchar *209*
putenv *211*
puts *212*
qsort *213*
read *215*

rename *216*
rmdir *217*
rsvstk *218*
sbrk *219*
scanf *220*
sensebrk *222*
setbuf *224*
setjmp *226*
setmem *228*
setvbuf *230*
sprintf *232*
sscanf *234*
strcat *235*
strchr *237*
strcmp *238*
strcpy *239*
strlen *240*
strlwr *241*
strncat *242*
strncmp *244*
strncpy *246*
strupr *247*
tolower *248*
toupper *250*
ungetc, ungetch *252*
unlink *254*
write *255*

第7章 コマンドリファレンス *257*
7.1 CF *257*
7.2 CG *258*
7.3 FPC *259*
7.4 MX *260*

第8章 エラーメッセージ一覧 *261*
8.1 CF のエラーメッセージ *261*
8.2 CG のエラーメッセージ *275*
8.3 FPC のエラーメッセージ *277*
8.4 MX のエラーメッセージ *279*

第9章 標準ライブラリ関数一覧 *281*
9.1 ファイル入出力関数 *281*
9.1.1 高水準入出力関数 *281*
9.1.2 低水準入出力関数 *282*
9.2 文字列および文字処理関数 *282*
9.3 メモリ管理関数 *283*
9.3.1 低水準メモリ管理関数 *283*
9.3.2 高水準メモリ管理関数 *283*
9.4 ディレクトリ関数 *283*
9.5 プログラム操作関数 *283*

9.6 キーボード，I/O 関数 *284*
9.7 機械語，MSX-DOS ファンクションコールサポート関数 *284*
9.8 メモリ操作関数 *284*
9.9 汎用関数 *285*

付録A サンプルプログラム"q.com"について *286*
A.1 "q.com"とは *286*
A.2 q.com 内の関数を利用するには *286*
A.2.1 関数の宣言 *286*
A.2.2 リンク *287*
A.2.3 q.com 内の関数の仕様について *287*
A.2.4 関連書籍の紹介 *289*

付録B MSX-C Ver.1.2 マスターディスク内容 *290*

索 引 *293*

お問い合わせについて *295*

序章　本マニュアルの構成

本マニュアルは，以下の3点に留意して書かれています．

(1)C言語に初めて接するユーザーが，抵抗なくC言語の世界に入り，MSX-Cコンパイラを使いこなすことを可能にする．

(2)本格的なプログラムの開発のツールとしてMSX-Cコンパイラをお使いになるユーザーが，その処理系の特徴を把握し，すぐに実用できるようにする．

(3)必要なときにすぐに項目を参照できるような構成とする．

また，操作方法についてはサンプルプログラムを用いて初学者にも分かりやすく解説しています．さらに，処理系としてのMSX-Cコンパイラの特徴，標準的なCとのコーディング上の違いなどを，ひとつの章にまとめて解説してあります．

さらに，標準ライブラリ関数や各コマンドのパラメータが即座に参照できるように独立した章を設け，リファレンスマニュアルとしてお使い頂けるようになっています．

本マニュアルは，始めから順次読み進むことによって初学者の方でもMSX-Cコンパイラを理解できるように章組を工夫しています．他の処理系によるC言語をお使いになった経験のある方は，必要な箇所のみを随時ご参照下さい．

以下に本マニュアルの第1章以降の概略を示します．

第1章　MSX-Cコンパイラの概要

本章では，MSX-Cコンパイラの概要を述べると共に，開発環境の構築方法，ソースファイルの作成からプログラムの実行までの流れの概説，さらに付属のサンプルプログラムについても解説します．C言語をはじめて学習される方は是非本章からお読み下さい．

第2章　MSX-Cコンパイラの操作

サンプルプログラムを実際にコンパイルしながら，MSX-Cコンパイラの各コマンド操作を習得していきます．

第3章　MSX-Cによるプログラミング

標準的なC言語の処理系であるV7　UNIX-Cとの言語仕様上の相違点をはじめとして，MSX-C独自の処理によるオブジェクト効率の向上や，アセンブリ言語とのリンクの方法等MSX-Cによるプログラムの開発に不可欠な知識を解説します．

第４章　MSX-C 標準ライブラリ

MSX-C の標準ライブラリルーチンの概要，V7　UNIX-C のライブラリとの違い，さらにユーザー独自のライブラリ作成及び，保守の方法についても本章で解説します．

第５章　MSX-C コンパイラの応用

MSX-C の応用例として MSX-DOS 以外の環境で実行するプログラム開発の実際を示します．

(1) BASIC 環境下のマシン語サブルーチンの作成

(2) ROM 化プログラムの作成

第６章　標準ライブラリ関数リファレンス

付属する標準ライブラリ関数をひとつずつ解説します．また使用例をあげ分かりやすく，関数の細かい部分まで説明してあります．

第７章　コマンドリファレンス

各コマンドのパラメータ及びそれぞれの機能が表形式で参照できます．

第８章　エラーメッセージ一覧

各コマンドが表示するエラーメッセージをコマンドごとに説明してあります．

第９章　標準ライブラリ関数一覧

標準ライブラリ関数の一覧を関数の分野別に表にしています．第 6 章のリファレンスのページも表に含まれています．

第１０章　サンプルプログラム q.com について

MSX-C を使用して作成したプログラム q.com について説明します．

第１１章　MSX-C Ver.１.２　マスターディスク内容

MSX-C コンパイラ Ver.1.2 に付属してくるディスクの内容の一覧です．使用する前にできるだけご確認下さい．

第１章　MSX-Cコンパイラの概要

1.1　MSX-Cコンパイラの概要

1.1.1　MSX-Cコンパイラ

MSX-C コンパイラ Ver.1.2(以下，MSX-C と呼ぶ)は，MSX-DOS バージョン2(以下，MSX-DOS2) 上で動作し，ザイログ・ニーモニックのアセンブル・ソースプログラムを出力する2パスのC言語コンパイラです．MSX-DOS バージョン1（以下，MSX-DOS1）では動作しません．

コンパイラを構成するのはCF及びCGというプログラムです．CF（フロント・エンド）はコンパイラの第1パスで，C言語で記述されたソースプログラムを読み込み，文法上，意味上のエラーをチェックし中間言語ファイル(T-codeと呼ばれる)をディスク上に作ります．この中間言語ファイルを読み込みアセンブリ言語のソースファイルに変換するのがコンパイラの第2パスであるCG（コードジェネレータ）です．

さらに本パッケージには，CFの出力する中間言語ファイルを読み込み，各関数にわたされるパラメータがその関数の定義と一致しているかどうかをチェックするプログラムFPC，ユーザー自身による標準ライブラリルーチンの作成をサポートするプログラムMXなどにより，プログラムの開発効率を高めています．

1.1.2　MSX-Cの特徴

MSX-Cの最大の特徴は，その生成するオブジェクト・コードが従来のアセンブリ言語によるプログラムに匹敵するコンパクトで効率性に富んだものであるという点です．これにより，元来アセンブリ言語以外でのプログラムが困難であったシステムプログラムや，ROM化プログラムのC言語による開発が可能となったのです．

さて，MSX-Cでは，このように高いコード効率を得るためにいくつかの新手法が取り入れられています．なかでも変数の自動レジスタ割付という手法をとりいれたことは，マイクロ・コンピュータ用のコンパイラとしては画期的な試みであり，オブジェクト・コードの質を大きく向上することに成功しています．

変数がCPUのレジスタにその値を持っていれば，メモリ上にあって演算のたびにメモリにアクセスするよりもはるかに効率がよいわけです．しかし，標準的なCコンパイラではプログラマがどの変数をレジスタに割り付けるのが最も効率的かを判断し，これを宣言子 register により宣言していました．

これは，プログラマにとって，手間のかかる作業であるわけですが，そもそも8080，Z80といった8ビットのCPUを搭載したマシン上のコンパイラでは，レジスタの数などの制限により，

register 宣言という手法を取り入れることができませんでした.

MSX-C のコード・ジェネレータは最適なレジスタ割付を自動的に行ってくれ, ユーザーは特にこれを意識する必要がないわけです.

さらには, バイト (8 ビット) 算術演算のサポート, 関数の引数のレジスタ渡し, 非再帰モードの導入など, より効率のよいコード生成を目指し, 新手法がふんだんに盛り込まれているのです.

以下に, MSX-C によるコンパイル結果の一例を示します.

サブルーチン toupper ( ) は, 引数として与えられた 1 文字が, アルファベットの小文字なら大文字に置き換えて, 関数の値として返すルーチンです. toupper.c をコンパイルした結果, 生成されたアセンブラ・ソースが toupper.mac です.

変数 c がメモリ上にでなく常に A レジスタに置かれ, 8 ビット (char 型) の値が 16 ビットに拡張されずに 8 ビットのままで演算されている様子がよくわかるでしょう.

```
   TOUPPER.C:
1: char     toupper(c)
2: char     c;
3: {
4:          if('a' <= c && c <= 'z')
5:                  return (c + 'A' - 'a');
6:          else
7:                  return (c);
8: }
```

```
  TOUPPER.MAC:
1:
2: ;        MSX-C ver 1.20x   (code generator)
3:
4:          cseg
5:
6: toupper@:
7:          cp      97      ⇦97="a"
8:          ret     c
9:          cp      123     ⇦123="z"+1
10:         ret     nc
11:         sub     32      ⇦32="a"-"A"
12:         ret
13:
14:
15:         public  toupper@
16:
17:         end
```

1.1.3　標準カーネルの諸機能

本節では，MSX-C の標準カーネルでサポートされる機能を解説します．カーネルとは，MSX-DOS がプログラムを起動するときにそのプログラムに渡される情報を，C の環境に合わせるプログラムのことをいいます．つまり，コマンドの引数を解析して分割したり，標準入出力ファイルをオープンするなどの作業をします．カーネルがあることで，MSX-DOS のインターフェースと UNIX のインターフェースの違いを吸収できているわけです．標準カーネルは標準ライブラリと一緒にリンクされます．

A)引数（アーギュメント，argument)

コマンドの引数は空白（スペースやタブ）で区切って渡します．引数に空白，"｜"や">"などを含めたい場合には，ダブルクォーテーションマーク（"）で引数全体を囲んでください．"がはずされたものが引数として渡されます．また，ダブルクォーテーションマークの内側では以下のものを 1 文字として扱います．

¥n	改行文字（¥n）
¥r	復帰文字（¥r）
""	ひとつのダブルクォーテーションマーク（"）

解析，分解された各引数は関数 main ( ) の第 2 引数に，各引数へのポインタの配列として渡されます．main ( ) の第 1 引数は整数で，(その引数の数+1) を示しています．普通は次のようにパラメータを受け取ります．

```
main(argc, argv)
int     argc;
char    *argv[];
{
        ...
}
```

argv [0] には UNIX の標準 C と同じ様に MSX-C でもコマンド名が渡されます．コマンド名はフルパス名（ドライブ，パス，ファイル名，拡張子すべてがそろったファイル名のこと）で渡されます．しかし，COMMAND2.COM や ver.1.2 の標準ライブラリ以外のプログラムによってコマンドが起動された場合には argv[0] にはヌル文字列や，異なったコマンド名が渡されることがあります．

コマンドに対する引数は argv [1] 以降に渡されます．引数の終わりとして，最後の引数の次（argv [argc]）には NULL が代入されています．引数は環境変数 UPPER の値によらずに大文字，小文字の区別が保存されます．

注意

環境変数 UPPER については MSX-DOS2 リファレンスマニュアル 81 ページ「Chapter 7　環境変数」を参照して下さい.

例

```
A>a:¥bin¥dump /n/s100 msxdos2.sys command2.com
```

コマンドラインから上のようにコマンドを起動すると main ( )への引数は次のようになります.

argc	4
argv [0]	"A：¥BIN¥DUMP.COM"
argv [1]	"/n/s100"
argv [2]	"msxdos2.sys"
argv [3]	"command2.com"
argv [4]	NULL

B)標準入出力ファイルの割り当て

UNIX の環境では，プログラムが起動されると 5 つの標準入出力ファイル（高水準入出力）がオープンされています. MSX-C でも同じ様に stdin, stdout, stderr, stdaux, stdprn がオープンされています. それぞれの入出力方向は stdin は入力，その他は出力だけができます. stderr, stdaux は標準 C では入力，出力ともに可能ですが，MSX-C の標準ライブラリでは構造上実現できないので，片方向（この場合は出力のみ）になっています. バッファリングの方法は，stdin, stdout, stdprn は行バッファリング，stderr, stdaux はバッファリングなしになっています. ファイルにリダイレクトされているときには，フルバッファリングとなります. バッファリングの詳細については「4.3.1 B)高水準入出力関数のバッファリング」を参照して下さい.

注意

ファイルのリダイレクトはコマンドレベルでは stdin と stdout しかできません.

低水準入出力関数（ファイルハンドルによるアクセス）であれば stderr, stdaux は入出力両方可能です.

1.2 ソースファイルの作成からプログラムの実行まで

前節でMSX-Cの概要とその8ビットコンパイラとしての特徴の一部を述べましたが，C言語によるプログラムの開発を実際に行うにはCコンパイラだけでは十分ではありません．C言語ソースファイルを作成するにはエディタが必要ですし，Cコンパイラが生成してくれるのはアセンブリ言語のソースファイルであり，これをマシン語に翻訳してくれるアセンブラも必要でしょう（ハンド・アセンブルという手もある）．

本節では，ソースファイルの作成から実行可能なプログラムの生成までの手順を追いながら，MSX-Cを用いてのプログラム開発環境について述べていきます．

1.2.1 エディタ

C言語によるプログラムの開発の第一歩は，C言語ソースプログラムを作成することです．ここで必要となるのがエディタです．（24ページの図の(I)）

実際の開発では試行錯誤を繰り返しながらプログラムを完成させていく場合が多くソースファイルにも何度も手を加えることになります．そうなると，豊富なファイル編集機能を備えた使い易いエディタが必要となってきます．

現在MSX-DOS2上で使用可能なエディタには『MSX-DOS2 TOOLS』のパッケージに付属のスクリーンエディタなどがあります．

MSX-Cでは，C言語のソースファイルのファイル名拡張子は.cを用いることをお薦めします．他の拡張子を使用することも可能ですが，コンパイル時にその拡張子を含めてファイルを指定する必要があります．つまり，コンパイラはファイル名に拡張子がないと.cを暗黙的にファイルの拡張子としてそのファイルをコンパイルします．

1.2.2 C言語ソースからアセンブラ・ソースへ

ソースファイルが作成できたら次は，コンパイラの出番です．MSX-Cではコンパイルの作業を2段階に分けて行うことは1.1.1のとおりですが，第1パスでソースファイルの文法上の間違いが発見された場合は，再びエディタに戻ってファイルを変更しなければなりません．そして再度CFにかけるという作業を繰り返します．（24ページの図の(II)）

CFをエラーなく終了するとディスク上に中間言語ファイルが作られます．この中間言語ファイルは拡張子.tcoを持ちます．

この中間言語ファイルをコンパイラの第2のパスであるCGにかけることにより，Z80ニーモニックによるアセンブラ・ソースファイルに変換され，.macという拡張子でディスク上にセーブされます．（24ページの図の(IV)）

1.2.3 実行可能プログラムの生成（アセンブル，リンクの意義）

以上でCコンパイラの本体であるふたつのパスを終了しました．生成されたアセンブラソースは，アセンブリ言語に精通したプログラマの書いたものに引けを取らない程の質の高いものです（勿論，コンパイラの生成したアセンブラプログラム固有のくせはあるのですが）．

アセンブルという作業を一口に言うと，アセンブラ・ソース中の各命令（ニーモニック＋オペランド）から，それと等価なマシン語（CPUが理解，実行できる命令）への置き換えです．（24ページの図の(V)）

例えば，俗に言うハンド・アセンブルという作業では，ジャンプ命令の飛び先やCALL命令により呼び出されるサブルーチンのアドレスを計算し，アセンブラのニーモニックやラベルを一つずつマシン・コード，実アドレスに置き換えていくのです．

同様の作業をハンド・アセンブルに代わって行うのが，アセンブラ及びリンカです．MSX-DOS上で動作するアセンブラにMSX・M80，リンカにMSX・L80があります．

MSX・M80は，Z80または，8080のニーモニックによるアセンブラ・ソースファイルを読み込んでリロケータブル・オブジェクト・ファイル（ファイル名拡張子.relを持つ）に変換します．

リロケータブル・ファイルでは，アセンブラのニーモニックはマシンコードに置き換えられているのですが，この段階ではそのプログラムがメモリのどのアドレスに置かれるのかが不確定であり，従ってジャンプ命令の飛び先アドレスなどは，相対的にしか定まりません．また，ライブラリ関数など外部のサブルーチンは，その名前が与えられているだけなので，CALL命令の呼び先のアドレスも定まっていないのです．

言い替えれば，リロケータブル・オブジェクトは，ロード開始アドレスを与えることによりどのアドレスにでも置くことができ，またそのプログラム中で呼び出される外部のサブルーチンを，名前さえ同一にしておけば，自由にすげ替えることができるわけです．

ここまでの作業でユーザーが作成したリロケータブル・オブジェクト・ファイル，外部ルーチンの集合であるライブラリ（これも.relファイルである），メモリのどのアドレスからロード実行するか等の情報をMSX・L80に渡すことにより，アドレスが定まり実行可能なプログラムとなります．（24ページの図の(VI)）L80は，与えられたパラメータにより，プログラム中のジャンプ，コールなどの命令に実際のアドレスを与え，ライブラリ・ファイルの中からユーザー・プログラム中で使われている関数だけを抜き出して，これを結合してくれるわけです．

1.3 MSX-Cの開発環境（開発環境の構築）

前節で述べたとおり，C言語によるプログラムの開発には本パッケージに含まれるプログラムの他にも様々な道具が必要です．付属のフロッピーディスクは，1DDなのであまり容量がないうえに，ライブラリのソースやサンプル・プログラムのソース等が含まれているため，ユーザー・プログラムを作成するスペースが十分とは言えません．そこで，付属のディスクからMSX-DOS2のシステムの入ったディスクに通常の開発に必要なファイルだけをコピーし，更にアセンブラ，リンカ，エディタなどのユーティリティをまとめたシステムディスクを作成するのがよいでしょう．

以下に，MSX-Cコンパイラを中心とした開発用のディスクの作成例を示します．この例は，MSX-Cコンパイラに付属するバッチファイルmksys.bat（ディレクトリ¥batchにあります）を実行すると構築される環境です．MKSYSの使用方法を説明しましょう．

(1)新しいディスクをフォーマットします．
(2)フォーマットされたディスクにmksys.batをコピーします．MSX-Cのマスターディスクが Bドライブにあるとすると，次のようになります．

```
A>copy b:¥batch¥mksys.bat
```

(3)コピーしたディスクをAドライブにいれ，次のようにコマンドを入力します．

```
A>mksys b
```

(4)あとはプロンプトに合わせて，MSX-DOS2のシステムディスクなどをBドライブに入れます．

以上でルートディレクトリで作業するためのディスクができます．2ドライブシミュレータを使っている場合には，"Insert disk for drive <d>："というメッセージが出てからディスクを交換して下さい．

ルートディレクトリで作業するディスクが出来上がりましたが，まだ環境（環境変数）が整っていません．MKSYSコマンドの最後にも表示されるように，ソースファイルのエディット，コンパイルをする前にCENVを実行しなければなりません．MKSYSで出来上がったディスクがAドライブにあるなら，次のようにします．

```
A>cenv a
```

CENV の引数には，MKSYS で作成されたディスクを入れるドライブの，ドライブ名(B ドライブなら"b"の 1 文字)だけを指定します．バッチコマンドの実行が終れば，環境変数も設定されています．CENV は一度電源を切ってしまうと，もう一度実行しなければなりません．MSX-C を使うときに必ず必要なコマンドですから，MSX-DOS2 立ち上げディスクの autoexec.bat に同じ内容を入れておくのがいいでしょう．

出来上がったディスクには次のようなファイルが入っているはずです．(A ドライブにディスクがあるとします．)

a：¥　　(ルートディレクトリの内容)

mksys.bat	MSX-C のディスク作成用バッチファイル
msxdos2.sys	MSX-DOS2 システムファイル
command2.com	〃
ck.rel	ライブラリファイル
clib.rel	〃
crun.rel	〃
cend.rel	〃
c.bat	コンパイル用バッチファイル
cenv.bat	MSX-C の環境を整えるバッチファイル

a：¥bin　　(コマンドのディレクトリ¥bin の内容)

m80.com	アセンブラ
l80.com	リンカ
cf.com	コンパイラ（パーサ）
cg.com	コンパイラ（コードジェネレータ）
fpc.com	ファンクションパラメータチェッカ
lib.tco	FPC 用標準ライブラリ.tco ファイル

a：¥include　　(ヘッダファイルのディレクトリの内容)

bdosfunc.h
conio.h
ctype.h
direct.h
io.h
malloc.h
memory.h
process.h
setjmp.h
stdio.h
stdlib.h

string.h
type.h

環境変数の設定

INCLUDE	a：¥include
PATH	a：¥； a：¥bin

このままでは，エディタが入っていませんし，まだ使いにくいので，ライブラリの作成をおこなうためのlib80.comやそのためのバッチファイル，MSX-DOS2 TOOLSなどのテキストファイルを扱う各種フィルタなどを加えるなど，より使いやすい環境を作り上げて下さい．

実際のプログラムの開発には，上記のようなシステムディスクの他に，ソース・ファイルや，コンパイラのワーク用のディスクを作ることにより，例えば1DDのディスク・ドライブをお持ちの方でも十分なスペースを確保することができ，折角作ったソース・プログラムがディスク容量不足の為セーブできなかったり，システムファイルを壊してしまうといったトラブルを避けることができます．

ワークディスクにはシステムファイルの他，必要最小限のファイルだけを入れておき，ソース・ファイルをこのディスク上に作ります．

ディスク・ドライブを2台以上お持ちの方の場合，Aドライブにシステム・ディスク，Bドライブに，ソース・ファイルの入ったワーク・ディスクを挿入し，例えば，

```
A>c b:head
```

c.batを起動しBドライブのhead.cをコンパイルする

などのように入力すれば中間言語ファイルや，アセンブラ・ソース，実行可能ファイルなどすべてが，Bドライブ上に作られます．

また，ディスク・ドライブが1台のユーザーでも，MSX-DOSの2ドライブ・シミュレータにより，システム・ディスクとワーク・ディスクを交互に抜き差ししながら，同様に2枚のディスクを使い分けることができます．

なお，2ドライブ・シミュレータの詳細については，MSX-DOS等のマニュアルをご参照下さい．（ただしこの時にはディスクの抜き差しが頻繁になり，ディスクの使い分けに注意する必要があります．）

1.4 サンプル・プログラム

付属のサンプル・プログラムについて解説します.

1.4.1 サンプル・プログラム「HEAD」の仕様について

A)概要

HEAD は指定された，または標準入力から読み込んだ ASCII ファイルの先頭から指定の行数を出力する UNIX 流のフィルタコマンドです. 付属のディスクに収められた，サンプル・ソース・ファイル head.c を第 2 章の操作方法に従いコンパイル及び，リンクすることで，MSX-DOS2 上で実行可能な .com ファイルとなり，I/O リダイレクト，パイプライン等をサポートした有用なツールコマンドとなります.

具体的には，以下のように用います.

```
A>head <filename>
```

<filename>で指定されたファイルの先頭から 10 行をコンソール画面に出力します. このとき，出力の初めにファイル名が表示され，各行の先頭には，ファイルの先頭からの行番号がつけられます.

B)オプションスイッチ

HEAD コマンドではコマンド名の次にオプションスイッチの指定できます. これにより，出力形式や内容など，コマンドの動きを変えることができます. UNIX のコマンドでは，オプションスイッチの指定には，"-l"のようにハイフンが用いられますが，本プログラムでは，"/"（スラッシュ）が，これに代わり用いられています. 以下のようなオプションスイッチがサポートされています.

オプション名の指定には，大文字，小文字とも指定可能です.

/L<lines>	出力する行数<lines>を指定します.
/N	ファイル名，行番号の出力を省きます.

C)I/Oリダイレクション，パイプ機能

MSX-DOS2 がサポートします．

D)エラーチェック，ワイルドカードなど

例えば，"/L"で数字以外のパラメータが指定された場合，誤りの箇所等の情報をかえすとともに，コマンドの書式を表示します．

入力ファイルの指定にワイルドカードは用いられません．

ファイルを指定しない場合標準入力からの入力となり，コンソール入力のほか，上記リダイレクション，パイプライン等が使用できます．

1.4.2 ソースファイルの作成上の留意点

A)#プリプロセッサ

すべての標準ライブラリ関数の宣言，マクロ等の宣言はヘッダファイル stdio.h によって取り込まれます．また，VOID 型や BOOL 型の定義もこの中で type.h を取り込むことで行われています．プログラムの中でこれらの関数や型等を使う場合はソースファイルの先頭で# include プリプロセッサ文により取り込んで下さい．

また，MSX-C コンパイラではいくつかのプリプロセッサ文によって，コンパイルモードを様々に切り替えることができます．このうち本プログラムでは# pragma nonrec の指定によりデフォルトの関数モードを非再帰型としています．これらのプリプロセッサ文による制御の詳細については，「3.1.1　#プリプロセッサ文による制御」を参照してください．

B)コマンドへの引数

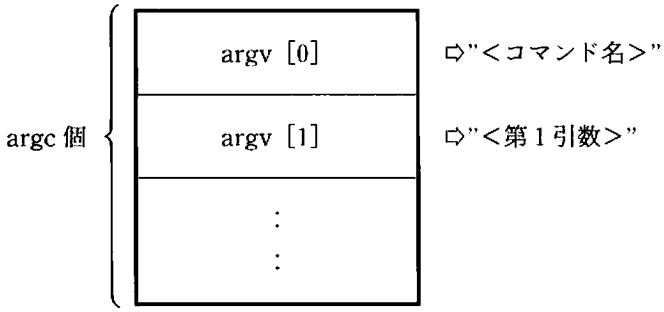
MSX-DOS 上で入力されたコマンドへの引数は main ( ) で定義された関数の引数 argc，argv としてユーザープログラムに以下のように渡されます．

(1) main ( ) の最初の引数には（引数の数＋1）の値が整数型で与えられます．

```
int     argc;
```

(2)第 2 の引数として実際のパラメータへのポインタの配列が渡されます．

```
char    *argv[];
```



ここで，ユーザープログラムと MSX-DOS2 のインターフェースの役割を果たすのが CK(カーネル) とよばれるモジュールです．カーネルでは main ( )に制御が渡る前にコマンド行の解析をおこないます．上記のように main ( )に渡されます．例えば，

```
A>head < test
```

のようにコマンド入力された場合パラメータの数は 0 個，従って main ( )の最初の引数には 1 が渡されることになり，ユーザープログラムの中ではファイル test はファイルポインタ stdin によってアクセスが可能になります．

```
A>head
```

とのみ入力された場合も同様であり，ただ，この場合 stdin がファイルではなくコンソールに割り当てられます．

なお，標準カーネル機能の詳細については「1.1.3　標準カーネルの諸機能」，「4.5.1　MSX-C ライブラリの構成」を参照してください．

C)関数の宣言

C 言語によるプログラミングの特徴としてプログラムが複数の比較的小さなモジュール(関数)に分けて書かれることが多くあります．MSX-C では呼び出される関数が他のモジュールで定義されている場合（標準ライブラリなどはまさにこの例）はもちろん，同一のソースファイルで定義する場合でも呼び出す側よりも後で定義する場合には呼び出す関数をあらかじめ宣言しておかなければなりません．ライブラリ関数の呼び出しにはヘッダファイルのインクルードによってこれをおこないますが，同一のソース内の関数の宣言は

(1)呼び出す側の関数の冒頭でこれを宣言する．
(2)呼び出される関数を呼ぶ側の関数より前で定義する．（この場合宣言は必要ない）
(3)ソースの冒頭で，そのモジュールで呼び出される関数をまとめて宣言する．

などによりおこないます．本サンプルでは2番目の方法を用いています．(main ( )がソースの一番最後で定義されている点に注意．3番目の方法を用いる場合は，main ( )から順番にソースプログラムを書いていく事が可能です．)

このようにMSX-Cでは「参照される識別子は，前もって宣言しておく」というスタイルを貫くために標準Cのような暗黙の宣言はおこないません．しかしながら，CFの起動時に"-f"スイッチを指定することにより「宣言されていない関数はint型を返す」と仮定し，暗黙のうちに宣言したのと同様のコンパイルが可能となってます．

第２章　MSX-Cコンパイラの操作

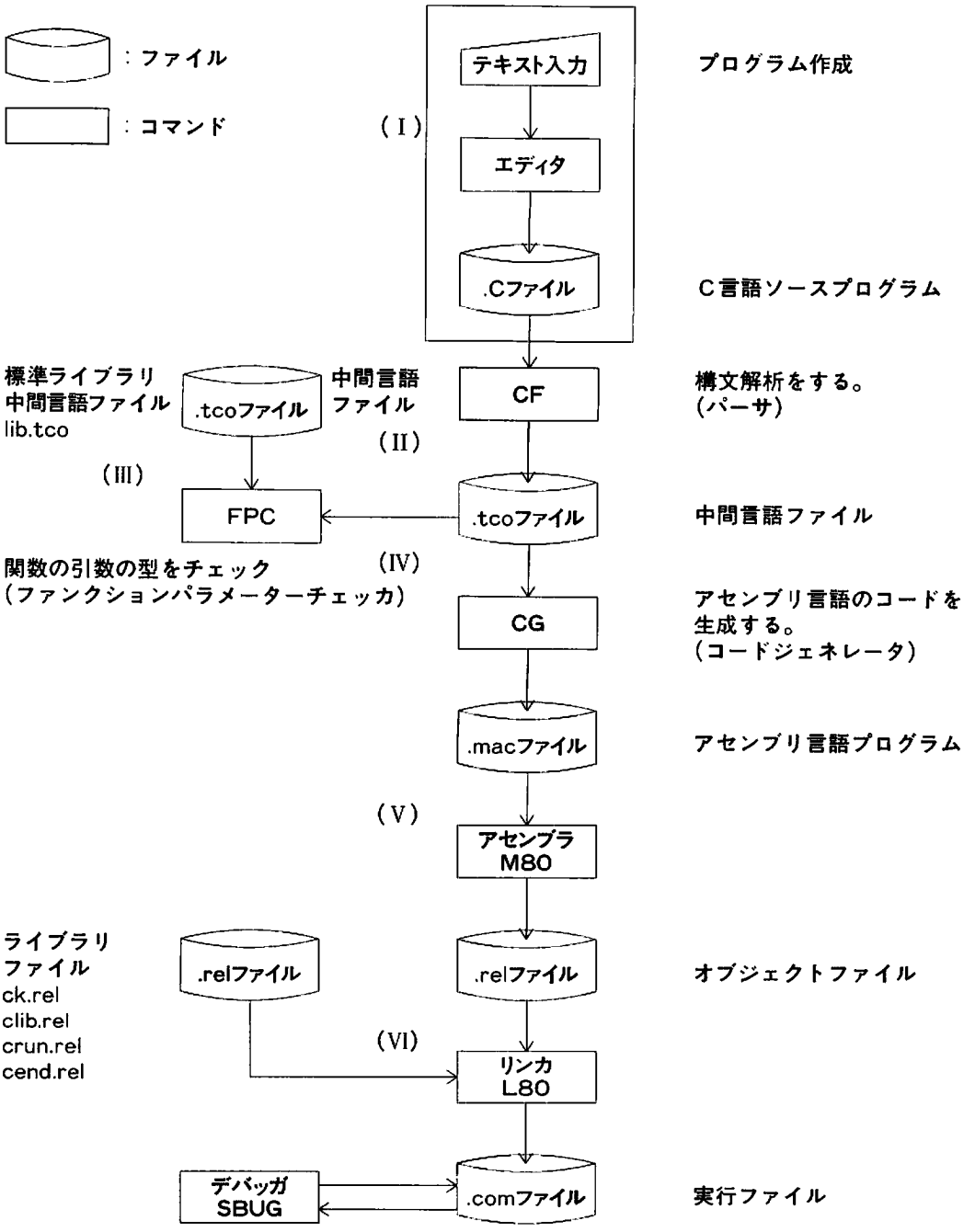


図　MSX-Cのプログラム開発

2.1　バッチファイルについて

Cのソースプログラム（.c）から実行プログラム（.com）を生成するには，数個のコマンドを実行する必要があります．しかし，ひとつひとつのコマンドをキーボードから入れていると手間や時間がかかりますし，誤りも多くなってしまいます．そのため，ソースプログラムのコンパイルから実行プログラムの生成までをひとつのコマンドで済ませるためのバッチファイルが用意されています．

MSX-Cに付属するc.batというバッチファイルの内容は次のようになっており，エラーがなければこれによってソースプログラムから実行プログラムを一気に作成します．(24ページの図の(II)から(VI)まで)

```
cf %2 %1
fpc lib %1
cg -k %3 %1
m80 =%1/z
del %1.mac
l80 ck,%1,clib/s,crun/s,cend,%1/n/y/e:xmain
```

c.batによるコンパイルのための書式は次のようになります．

```
A>c filename
```

ここで，filenameはコンパイルするソースプログラム名で，これは拡張子なしで指定しなければなりません．

では，実際にソースプログラムをコンパイルしてみましょう．ここで使用するソースプログラムはhead.cで，これはファイルの先頭の部分を切り出して表示するプログラムです．

```
A>c head        ←実際にキーボードから入力するのはこの1行だけ
MSX-C ver 1.20x   (parser)
Copyright (C) 1989 by ASCII Corporation
complete
MSX-C function parameter checker ver 1.20x
complete
MSX-C ver 1.20x   (code generator)
Copyright (C) 1989 by ASCII Corporation
puthlp  ..;
putusage        .;
atoix   .........:::;
shifarg ........:::;
parsesw ..............................................::::::;
ts      ................:::;
typehead        ..............:::;
main    ........................:::::::;
complete
```

```
No Fatal error(s)


MSX.L-80 1.00 01-Apr-85 (c) 1981,1985 Microsoft

Data    0103    182E   < 5931>

36744 Bytes Free
[0103   182E       24]

A>
```

これでコンパイルと実行プログラムの生成は正常に終了しています．ここで新たに作成されたファイルは，head.rel，head.com，head.sym の 3 つですが，head.com が実行プログラムです．

実行プログラムが生成されたので，head.com をコマンドとして，たとえば次のように使用することができるようになります．

```
A>head /n /15 head.c
```

この例では，head.c の先頭から 5 行をファイル名を省いて表示します．

2.2 各コマンドの操作

2.1 ではバッチファイルを使用して，ソースプログラムから実行プログラムの生成までを一度に行いましたが，バッチファイル中で起動されたそれぞれのコマンドを別個に起動することももちろん可能です．本節ではそれぞれのコマンドについて解説します．

2.2.1 CF (パーサ)

CF の書式は次のとおりです．

CF [オプション] filename

ここで filename は，C のソースファイル名です．ファイル名はドライブ名，パス名，拡張子を指定することもできます．拡張子は省略されると .c が使われます．また拡張子を指定すると .c 以外の拡張子のファイルを C のソースファイルにできます．

CF は，MSX-C コンパイラで C のソースファイルをコンパイルする際に，最初に実行しなければならないプログラム（コンパイラの第 1 パス）です．(24 ページの図の(II)) CF は入力として

Cのソースファイル（filenameで指定）を読み，出力として中間言語ファイルを書き出します．その際に，ソースファイルのすべてのエラー（文法や意味上のエラー）が検出されます．

CFは中間言語ファイルとエラーメッセージの出力を行います．中間言語ファイルはソースファイルのあるディレクトリに出力されます．エラーがあった場合には中間言語ファイルは生成されません．

コンパイル中にCtrl+CやCtrl+STOPを押すことで，コンパイルを中断することができます．ただし中間言語ファイルが残りますので削除して下さい．

中間言語はTコードと呼ばれ，CFで出力される中間言語ファイルは拡張子.tcoを持ちます．この.tcoファイルは，FPC（パラメータチェッカ）やCG（コードジェネレータ）の入力として使用されます．

Ver.1.2では文字列中の漢字も正しく認識できるようになりました．Ver.1.1までは漢字をコメントに使うことはできましたが，文字列中に漢字を使うと文字によっては変化してしまい，不都合が生じていました．新しいCFではこれをスイッチによって漢字を正しく認識するように指定できます．

通常Cのプログラムではヘッダファイルをインクルードしますが，Ver.1.2からはインクルードファイルのあるデフォルトのディレクトリを指定することができます．ディレクトリはMSX-DOS2の機能である環境変数で指定します．環境変数INCLUDEにディレクトリ名を設定しておいて，それからCFを起動することになります．ディレクトリをひとつ作り，そこにMSX-C付属のヘッダファイルをすべていれておき，そのディレクトリ名を環境変数INCLUDEに設定しておきます．こうすることで，コンパイルのたびにヘッダファイルがあるかどうかなどを気にする必要がなくなります．Aドライブに¥includeというディレクトリを作り，その中にヘッダファイルを入れたとすると，環境変数INCLUDEの設定は次のように行います．

```
A>set include=a:¥include
```

普通はすべてのヘッダファイルを読み込むためディスクアクセスが多く，時間がかかってしまいます．RAMディスク(Hドライブ)に十分余裕があるならばそこにディレクトリごとコピーしてから使うとより早くコンパイルが終了します．

以下に指定できるオプションと，その意味を説明します．

オプション

-c　　コメントをネストしない．

MSX-Cでは標準Cと異なり，コメントをネストすることができます．しかしこのオプションはコメントのネストを抑制し，コメントを標準Cの場合と同様に扱います．コメントのネストとはソースファイル中で，次のようなことができることをいいます．

```
...
/*      same fuction is defined in Standard Library stdlib.c
int     atoi(s)
char    *s;
{
        int     i;
        char    c;

        for (i = 0; (c = *s) >= '0' && c <= '9'; s++)
                i = i * 10 + c - '0';   /* when c == '0', shift and add 0 */
                                        /* when c == '9', shift and add 9 */
        return (i);                     /* returns value */
}
*/

...
```

これはatoi()を自分で作りましたが，標準ライブラリに同じ関数があるのが分かったので，コメントにすることで定義をやめたものです．この関数では内部に処理に対するコメントが書かれているので，コメントを含めて関数をコメントにしてしまっています．"-c"オプションを指定した場合，最初の"/*"から"returns value */"まですべてコメントになりますが，関数の終わりを示す"}"と"*/"がコメントでなくなってしまいます．つまり，コンパイル時になんらかのエラーが発生するはずです．

-e [filename] エラーメッセージをファイルに出力する．

通常，CFが生成するエラーメッセージはすべて標準出力（コンソール）に出力されますが，このオプションを指定すると，エラーメッセージはファイルに出力されます．出力先ファイルはfilenameで指定します．ファイル名はドライブ名，パス名，拡張子を指定することもできます．拡張子は省略されると.diaが使われます．また拡張子を指定すると.dia以外の拡張子のファイルを.diaファイルの出力先に指定できます．filenameにアルファベット1文字またはそれに":"(コロン)を指定するとそのドライブのカレントディレクトリに，"¥"で終るディレクトリ名を指定すればそのディレクトリに，"¥"以外で終ればそれをファイル名としてエラーメッセージを出力します．filenameを省略すると，.diaファイルはソースファイルと同じディレクトリに作成されます．エラーがなかった場合にはファイルは生成されません．このオプションの後には他のオプションはつけられません．

-f 宣言されていない関数とパラメータを暗黙的にint型として扱う．

標準Cでは宣言されていない関数や関数のパラメータはint型として扱われますが，MSX-Cではこれをデフォルトでは禁止しています．このオプションを指定すると，これらについてint型が暗黙に仮定され，標準Cとの互換性を保ちます．

-j　　文字列中の漢字を正しく認識する．
このスイッチを使わないと文字列中に漢字を入れた場合，文字が正しく認識されない場合があります．これは漢字コードにC言語のエスケープ文字"¥"が使われるためで，-jを指定することで漢字を正しく認識するようになります．文字列中に漢字が含まれるときにはこのスイッチを使って下さい．逆に半角(1バイト)のひらがなを使っているときには使ってはいけません．半角ひらがなや漢字を使っていない場合にはどちらでも構いませんが，指定しなくていいでしょう．

-o [filename] 中間言語ファイル（.tcoファイル）を任意のディレクトリに作成する．
デフォルトでは.tcoファイルはソースファイルと同一ディレクトリに生成されます．このオプションを使えば任意のディレクトリに任意のファイル名で中間言語ファイルを作成することができます．
filenameは，.tcoファイルの出力先を指定します．アルファベット1文字またはそれに":"(コロン)を指定するとそのドライブのカレントディレクトリに，"¥"で終るディレクトリ名を指定すればそのディレクトリに，"¥"以外で終ればそれをファイル名として中間言語ファイルを出力します．filenameを省略すると，デフォルトと同じ動作をします．このオプションの後には他のオプションはつけられません．

例

-otest¥	testというディレクトリに中間言語ファイルを出力します．
-o¥foo.t	¥foo.tというファイル名で中間言語ファイルを出力します．

-rP：S：H　コンパイラのワーク用テーブルをP：S：Hの比率で割り当てる．
ここで，Pはプール，Sはシンボルテーブル，Hはハッシュテーブルを表します．デフォルトではこの比率は，13：6：4となっています．CFでコンパイル中にテーブルオーバーフローが起こった場合には，オーバーフローとなったテーブルの数値をより大きくして，再コンパイルしてください．

-m　　メモリの使用状況を表示する．
"-r"オプションで説明したP：S：Hのそれぞれについて，メモリの配分が示されます．コンパイルに成功した場合には，通常の場合と同様，中間言語ファイルが作成されます．表示の形式は次のようになっています．

pool	＜使用バイト数＞/＜割当領域バイト数＞
symbol	table　＜使用バイト数＞/＜割当領域バイト数＞
hash	＜使用バイト数＞/＜割当領域バイト数＞

"/"の左側の＜使用バイト数＞はテーブルの現在までに使用したバイト数です．＜割当領域バイト数＞はテーブルに割り当てられた使用できる最大バイト数です．テーブルオーバーフローが起きたときには自動的にメモリ配分が表示されますので，それぞれの＜使用バイト数＞を見て，だいたいの比率を"-r"で指定して下さい．

-s　ソースプログラムにエラーが検出されても，バッチ処理を続行する．
分割コンパイルを行う場合に，ひとつのバッチファイル中で複数のソースファイルをコンパイルするときに，このオプションは特に有効です．ひとつのソースプログラム中のエラーで全体の処理を中断してしまうことなしに，すべてのソースファイルのエラーチェックを行うことができます．

-t　ポインタと整数の間の自動的な型変換を可能にする．
MSX-C の場合，デフォルトではこのようなポインタと整数の間の型の混用はエラーとなります．このオプションによって，ポインタと整数の間で，型変換が自動的に行われるようになります．"-f"オプションとこの"-t"オプションを使用しなくてもコンパイルできるようなプログラムを書くことが，MSX-C の場合には推奨されます．

エラーメッセージについて

エラーメッセージは2種類あります．コマンドの動作についてのメッセージとコンパイル結果のメッセージです．ここではそれぞれのメッセージについてどのようなものであるかの説明にとどめます．実際のエラーメッセージは「8.1　CF のエラーメッセージ」をご覧下さい．

コマンドの動作についてのメッセージ

これは CF コマンドの動作についてのメッセージです．例えば，ソースファイルが見つからないなどのメッセージです．このようなメッセージが表示されたときには，CF はほとんど処理を中止してコマンドレベルに戻ります．

コンパイル結果のメッセージ

ユーザーのソースファイルのコンパイル結果のメッセージです．この場合は CF はソースをなるべく正しく認識しようとしつつ，コンパイルを続けます．しかし，ワークエリアがないなどのエラーの場合には処理を続けること自体できないので，処理を中止してコマンドレベルに戻ります．

このメッセージの場合，出力形式が2種類あります．次のようなものです．

(1) line M column N：　＜message＞
(2)＜message＞

<message>はエラーメッセージを示し，M，N はそれぞれエラーのあった行，桁を表しています．(1)はその位置でエラーが発生したことを示しています．(2)は範囲が広くなります．その範囲はあるひとつの関数内やソースファイル全体です．

メッセージは普通，非常に的確にエラーの指摘をしますが，"}"など重要な意味を持った文字を忘れると，それ以降のＣ言語のソースを正しく解釈できなくなり，大量のエラーメッセージを発生させてしまいます．（これは，自由形式(文字列外の空白文字と識別子を区切る空白文字を無視する)であるＣ言語の特徴でもあります．）この大量に発生したメッセージはすべてが的確にエラーを指摘しているわけではなく，ほとんどが正しく解釈できなかったものに対するメッセージです．このような場合はだいたいの原因は，大量に発生したメッセージの最初の部分に本当のエラーがあります．それを修正することでメッセージはずいぶん減るでしょう．

エラーメッセージの表示方法には２種類あることは述べましたが，「8.3　FPC のエラーメッセージ」では行や桁の表示は，すべて省いてあります．

例

先の head.c を例にとって，実際に CF でコンパイルしてみます．

```
A>cf head
MSX-C ver 1.20x   (parser)
Copyright (C) 1989 by ASCII Corporation
complete
A>dir head.*
HEAD      C     3802 89-01-17  2:53p
HEAD      TCO   5888 89-01-17  2:54p  ←中間言語ファイル
  9K in 2 files  473K free
```

次に，ソースプログラムにエラーを持つ error.c というプログラムをコンパイルし，そのエラーメッセージをコンソールでなくファイルに出力します．

```
A>cf -e error
MSX-C ver 1.20x   (parser)
Copyright (C) 1989 by ASCII Corporation
errors detected
A>dir error.*

ERROR     C     3845 89-01-20  7:56p
ERROR     DIA    256 89-01-20  9:21p  ←メッセージファイル
  4K in 2 files  473K free
```

2.2.2 FPC （パラメータチェッカ）

FPC の書式は次のとおりです．

FPC ［オプション］ file1 file2 ...

ここで file1 file2 ... は中間言語ファイル（.tco ファイル）名です．ファイル名はドライブ名，パス名，拡張子を指定することもできます．拡張子は省略されると .tco が使われます．また，ワイルドカード（＊や？）を使用することもできます．

FPC の目的は，宣言されている関数の戻り値の型が実際の戻り値の型に一致しているかどうか，また，関数に渡したパラメータの型が，関数宣言で定義されているパラメータの型と一致しているかどうかを調べることにあります．(24 ページの図の(III)) つまり，FPC を使用することによって，意図しないパラメータや関数の型の不一致による潜在的なバグの可能性を最小限に抑えることができます．なお，すべての標準ライブラリ関数の型の情報は，lib.tco というファイルに存在します．

実際に例をあげて説明しましょう．以下のソースプログラム（test.c とする）を見てください．

```
1:      #include <stdio.h>
2:
3:      main()
4:      {
5:              int f1(), f2(), x, y;
6:
7:              x = f1('a', 3);
8:              y = f2();
9:              printf("%d %d¥n", x, y);
10:     }
11:
12:     int     f1(x, y)
13:     int     x, y;
14:     {
15:             return (x + y);
16:     }
17:
18:     char    f2()
19:     {
20:             return ('b');
21:     }
```

このプログラムでは，まず関数 main ( )の中で 2 つの関数 f1 ( ), f2 ( )がそれぞれ int 型として宣言されています(5 行目)．一方，関数 f1 ( )は，int 型の 2 つのパラメータを取るものとして定義されています(12～13 行目)．ところが関数 main ( )中では，関数 f1 ( )への第 1 パラメータとして，"a"という char 型のパラメータを渡しています(7 行目)．さらに，関数 f2 ( )は int 型として宣言されているのに，関数定義では char 型の戻り値を持つものとして定義されています (18～21 行目)．

ところが，これら 2 つの型の矛盾はパーサ (CF) では検出されません．意図しないでこのような矛盾を起こしている場合には当然バグの原因となりますが，FPC はこうした問題を検出し，ユーザーに指摘してくれます．

では実際にこのソースプログラムをコンパイルして，FPC でチェックしてみましょう．

```
A>cf test↵
MSX-C ver 1.20x   (parser)
Copyright (C) 1989 by ASCII Corporation
complete                                    ◇パーサは問題なく通る
A>fpc lib test↵
MSX-C function parameter checker ver 1.20x
in <test.tco>  "main" calls "f1" : 1st argument conflict
in <test.tco>  "main" calls "f2" : conflicting return type
complete
A>
```

このように，関数 f1 ( )の場合には「関数の最初のパラメータの型が不一致」，関数 f2 ( )の場合には「戻り値の型が不一致」と，正確に矛盾が指摘されています．なお，FPC のコマンド行で"lib" (lib.tco) を指定しているのは，プログラム中に printf ( )という標準ライブラリ関数があるためです．

FPC は，CF (パーサ) が正常に終了した直後に生成された中間言語ファイル (.tco ファイル) に対して実行するようにしておくとよいでしょう．

カレントディレクトリに，与えた中間言語ファイルが見つからない場合には，fpc.com のあったディレクトリを探します．それでもない場合は"cannot open： <filename>"と表示して終了します．ドライブ名やルートディレクトリからの指定があるとこの機能が発揮されません．lib.tco などよく使う .tco ファイルは fpc.com と同じディレクトリに置き，パス名なしでファイルを指定すると lib.tco の存在をほとんど意識しないですみます．この場合，カレントディレクトリに lib.tco があった場合にはそれが使われます．

オプション

-s　　FPC でエラーが検出された場合でも，バッチ処理を続行する．
分割コンパイルを行う場合など，多くのプログラムを一度にチェックする場合に便利なオプションです．

-u　　未定義関数に対する参照をチェックしない．
このオプションを指定すると，未定義関数を参照する記述が存在してもエラーになりません．

-t　　型のチェックをさらに厳しく行う．
デフォルトでは int，unsigned，ポインタは同じ型であるように処理されますが，このオプションを指定するとこれらはまったく別の型としてチェックされます．

-i　　間接呼び出しに対して警告を出す．（エラーメッセージの項を参照）

-c＜filename＞ライブラリ関数用の.tco ファイルを作成する．
ライブラリ関数のヘッダ部分（引数と戻り値）だけの情報を残し，.tco ファイルを圧縮，連結したものを＜filename＞に作成します．こうすることで，ライブラリ関数を FPC でチェックするのに必要な情報だけになり，ファイルに必要なディスクスペースが小さくなります．＜filename＞より後ろのファイル名は圧縮，連結するファイルとして解釈され，関数の引数のチェックなどは行われません．＜filename＞にはドライブ名，パス名，拡張子の指定ができます．拡張子は省略されると.tco が使われます．具体的な使用方法は「4.5.3　専用ライブラリの作成」を参照して下さい．

-d＜don't care 関数リスト＞ don't care 関数の指定をする．
don't care 関数とは，呼び出し時に余分なパラメータが付いていても，それを許すものです．例えば，関数 bdos（）は，MSX-DOS のファンクションコールによっては，第 2 引数がなかったり，int 型だったり char 型だったりと，一定しません．したがって，何も指定せずにチェック用ファイルを作ると，FPC でチェックを行ったときに“パラメータの数や型が合わない”というメッセージが出力されてしまいます．このような場合に対応するため，"-d"スイッチを使用します．＜don't care 関数リスト＞は，関数名をカンマ（，）で区切ってつなげます．空白を入れてはいけません．例か，標準ライブラリ再作成用バッチファイル gentco.bat を参照して下さい．また，このスイッチは"-c"スイッチと一緒に使わなければ意味がありません．

例

```
A>fpc -c mylib -d bdosd,bdosb mylib1 mylib2
```

この例では mylib1.tco，mylib2.tco を圧縮，連結し mylib.tco を作成します．その際，bdosd（）と bdosb（）は余分な引数を無視するようになっています．

エラーメッセージ

以下の部分では，

filename　は，エラーが発見されたファイル名
func1　　は，エラーが発見された関数名
func2　　は，その関数に対する呼び出しでエラーがあった関数名

を，それぞれ示します．

in <filename> "func1" was multiple defined
　　func1 が，プログラム中で二重（あるいは多重）定義されている．

in <filename> "func1" ...sorry,can"t check indirect call
　　func1 が別の関数を間接的に呼び出しているので，パラメータのチェックはできない．

このメッセージは，-i オプションを指定した時にだけ表示されます．これは，関数へのポインタを使っている場合に起ります．たとえば，次のような場合です．

```
int (*fp)(), func();  /* fp は関数へのポインタ  */
fp = func;            /* fp は func を指している  */
(*fp)(1, 2);          /* これは間接呼び出しなのでチェックできない */
```

その他のエラーメッセージは，以下のような共通の形式で出力されます．

in <filename> "func1" calls "func2"； メッセージ

ここで，『メッセージ』は以下のいずれかになります．

(1) conflicting return type
　　func2 の返す型が，関数定義本体の型と一致しない．

(2) conflicting number of arguments
　　func2 の呼び出しで，引数の数が定義と一致しない．

(3)<Nth> argument conflict
　　func2 の呼び出しで，N 番目の引数の型が定義と一致しない．
　　1 番目，2 番目，3 番目の引数の時には"Nth"はそれぞれ，"1st","2nd","3rd"になります．

(4) undefined
　　func2 が参照されているにもかかわらず，どこにも定義されていない．

2.2.3 CG (コードジェネレータ)

CG の書式は次のとおりです.

CG [オプション] filename

ここで filename は CF が出力した中間言語ファイル (.tco ファイル) です. ファイル名はドライブ名, パス名, 拡張子を指定することができます. 拡張子は省略されると .tco が使われます.

CG はコンパイラのコード生成フェーズであり, CF によって生成された中間言語ファイルを入力として読み込み, アセンブラのソースファイルを出力します. (24 ページの図の(IV)) 出力されるアセンブラのソースファイルの拡張子は, .mac です. コード生成中にエラーが発生した場合には .mac ファイルは生成されません.

CG の出力ファイルはそのまま M80 マクロアセンブラの入力として使用されます. すなわち, MSX-C コンパイラ自体の処理はアセンブラのソースファイルの出力までであり, そこから実際の実行ファイルの生成までの処理はマクロアセンブラとリンカが行うことになります.

コード生成中に Ctrl+C や Ctrl+STOP を押して, 処理を中止することができます.

オプション

-k　コード生成の終了時に, 入力ファイルである中間言語ファイル (.tco ファイル) を自動的に削除する.

-l　フルネームの生成.
CG は通常, グローバルシンボルの先頭から 6 文字までしかアセンブラ出力に対して生成しませんが, このオプションによってグローバルシンボル名全体が有効になります.

-o [filename] 出力ファイルを指定したディレクトリに生成する.
デフォルトでは .mac ファイルは .tco ファイルと同一ディレクトリに生成されます. このオプションを使えば任意のディレクトリに任意のファイル名で中間言語ファイルを作成することができます.
filename は, .mac ファイルの出力先を指定します. アルファベット 1 文字またはそれに":" (コロン) を指定するとそのドライブのカレントディレクトリに, "¥" で終るディレクトリ名を指定すればそのディレクトリに, "¥"以外で終ればそれをファイル名としてアセンブラファイルを出力します. filename を省略するとデフォルトと同じ動作になります. このオプションの後には他のオプションはつけられません.

例

-otest¥　　testというディレクトリに中間言語ファイルを出力します.

-o¥foo.asm　¥foo.asmというファイル名で中間言語ファイルを出力します.

-rN　　シンボルテーブル領域としてNバイト確保する（Nは10進数）.

-u　　処理の進行状況の非表示.
このオプションを指定すると，現在処理中の関数名と，進行状況を示す，"."（ピリオド）や":"（コロン），";"（セミコロン）の出力（後述の例を参照）をしません.

エラーメッセージについて

CGにもエラーメッセージは2種類あります．コマンドの動作についてのメッセージとコンパイル結果のメッセージです．ここではそれぞれのメッセージについてどのようなものであるかの説明にとどめます．実際のエラーメッセージは「8.2　CGのエラーメッセージ」をご覧下さい.

コマンドの動作についてのメッセージ

これはCGコマンドの動作についてのメッセージです．例えば，.tcoファイルが見つからない，ワークエリアが不足しているなどのメッセージです．このようなメッセージが表示されたときには，CGは処理を中止してコマンドレベルに戻ります.

コンパイル結果のメッセージ

.tcoファイルのコンパイル結果のメッセージです．CGはコンパイルの結果としては警告メッセージ（warning）を発生させるだけです.

例

先のHEADプログラム（1.4参照）をCGで処理してみます．これ以降の処理では中間言語ファイルは不要なので，-kオプションを指定して.tcoファイルを削除しています.

```
A>dir head.*
HEAD      C     3802 89-01-17  2:53p
HEAD      TCO   5888 89-01-17  2:54p  ◁中間言語ファイル
  9K in 2 files  473K free
A>cg -k head
MSX-C ver 1.20x   (code generator)
Copyright (C) 1989 by ASCII Corporation
puthlp  ..;
putusage        .;
atoix   .........:::;
```

```
shifarg .....:::;
parsesw ..........................................:::::;
ts      ................:::;
typehead        ..............:::;
main    ........................:::::::;
complete
A>dir head.*↵
HEAD      C     3802 89-01-17  2:53p
HEAD      MAC   9330 89-01-17  2:58p
 12K in 2 files  463K free   ⇦ .tco ファイルが消去され，アセンブラのソースファイル
                                (.mac ファイル) が生成されている
A>
```

この例では -u オプションを指定していないため，処理の進行状況が表示されています．進行状況として最初に表示されるのが，現在処理している関数名です．それに続いて表示されているピリオドは，関数中のそれぞれの文（ステートメント）が処理中であることを表し，コロンが表示されているものについては，そこで最適化のパスが動作していることを示しています．

2.2.4 アセンブルとリンク

さて，前節までで MSX-C コンパイラ自体の処理は終了し，出力ファイルとして得られたアセンブラのソースファイル (.mac ファイル) を M80 マクロアセンブラに渡し，さらに L80 リンクローダを起動することによって，最終的な実行可能ファイル(.com ファイル)を作成します．(24 ページ図の(V)と(VI))なお，MSX-C プログラムのリンク時には，次の 4 つの .rel ファイルが必要です．

ck.rel	MSX-C 標準カーネル
clib.rel	MSX-C 標準ライブラリ
crun.rel	実行時ルーチン
cend.rel	トレーリングファイル

ここでは前節に続いて，HEAD プログラムの作成を例として，実行可能ファイルの作成までを見て行きます．

```
A>m80 =head↵          ⇦アセンブラの起動

No Fatal error(s)

A>dir head.↵
HEAD      C     3802 89-01-17  2:53p
HEAD      MAC   9330 89-01-17  2:58p
HEAD      REL   2048 89-01-17  2:59p   ⇦生成された
                                          オブジェクトファイル
 14K in 3 files 443K free
```

```
A>l80 ck,head,clib/s,crun/s,cend,head/n/y/e:xmain ↵   ◁リンカの起動

MSX.L-80 1.00 01-Apr-85 (c) 1981,1985 Microsoft

Data    0103    182E   < 5931>

36744 Bytes Free
[0103   182E       24]

A>dir head.* ↵
HEAD      C     3802 89-01-17  2:53p
HEAD      MAC   9330 89-01-17  2:58p
HEAD      REL   2048 89-01-17  2:59p  ◁生成されたオブジェクトファイル
HEAD      COM   6016 89-01-17  2:59p  ◁実行可能ファイル
HEAD      SYM   1152 89-01-17  2:59p  ◁シンボルテーブルファイル
 21K in 5 files 436K free
A>
```

これで実行可能なHEADのプログラムが生成されました.

ここではリンカのパラメータなどについては説明しませんが，シンボルテーブルファイル（.symファイル)を作成しないようにもできますし，また，リンカを起動する前にアセンブラのソースファイル（.macファイル）を削除してしまっても，この場合はかまわないでしょう.

第3章　MSX-Cによるプログラミング

3.1 標準C(V7UNIX-C)との言語仕様上の相違点

C言語には現在のところ，国際的な工業規格のような意味での「標準C」は存在しません．多くのCの処理系は，Kernighan & Ritchieの"The Programming Language C"準拠，あるいはANSIの標準化案準拠などという形を取っています．ここでは，標準的なCの処理系のひとつである，UNIX Version 7のC（以下では標準Cとする）との比較で，説明していきます．

また，この章に含まれないものは，同じ機能を実現しています．すなわち，switch文などの制御構造，構造体・共用体，すべての演算子など言語仕様に含まれているものはすべてサポートされています．

3.1.1 #プリプロセッサ文による制御

標準Cではプリプロセッサ文はコンパイラのプリプロセス時にソースファイルを拡張するものとして使用されますが，MSX-Cではコンパイラを制御するためのプリプロセッサ命令が用意されています．このコンパイルモード制御のためのプリプロセッサ文は# pragma文で，以下のような制御を行うことができます．

(1)# pragma nonrec
　　デフォルト関数モードを非再帰モードにする．
(2)# pragma recursive
　　デフォルト関数モードを再帰モードにする．
(3)# pragma pdp11mode
　　PDP-11コンパチブルモードでコンパイルする．
(4)# pragma regalo
　　コードジェネレータ中のレジスタ割り付けフェーズを起動する．
(5)# pragma nonregalo
　　コードジェネレータ中のレジスタ割り付けフェーズを起動しない．
(6)# pragma optimize time
　　オブジェクトプログラムのコード効率よりも速度効率を重視する．
(7)# pragma optimize space
　　オブジェクトプログラムの速度効率よりもコード効率を重視する．

pragma pdp11mode 以外の文は，プログラム中の任意の場所で指定することができます．これらについては，以下で詳しく説明します．

A)再帰モードと非再帰モード

標準Cではすべての関数は再帰的に使用することができますが，MSX-Cでは関数を非再帰的なものとして宣言することもできます．再帰モードの関数と非再帰モードの関数を区別するため，2つの予約語 recursive と nonrec が新たに導入されています．これらの予約語は関数モード指定子と呼ばれ，次の例のように関数定義の先頭に置きます．

```
nonrec main()
{
        printf("hello, world¥n");
}

recursive int
factorial(n)
int n;
{
        return ((n > 1) ? n * factorial(n - 1) : 1);
}

int max(a, b)
int a, b;
{
        return ((a > b) ? a : b);
}
```

上記の例で定義されている3つの関数のうち，関数 main ( ) は非再帰的関数，関数 factorial ( ) は再帰的関数として宣言されています．また，3つ目の関数 max ( ) は関数モード指定子を持たないので，デフォルト関数モードを持つものと見なされます．

デフォルト関数モードは通常は recursive ですが，次のプリプロセッサ文によって変更することができます．

```
#pragma nonrec
#pragma recursive
```

前者はデフォルト関数モードを nonrec に，後者は recursive にします．上記の例で，関数 max の定義以前にこれらの# pragma 文が現れていない場合には，関数 max( ) は recursive として扱われます（デフォルト）．現れている場合には，関数 max( ) は最も近くで指定された関数モードになります．

MSX-C では非再帰的な関数は nonrec と宣言するようにしてください．これによってコンパイラは，サイズが小さく，実行速度も速いオブジェクトコードを生成するようになります．通常の

アプリケーションプログラムでは，大部分の関数は非再帰的なので，ソースプログラムの先頭にプリプロセッサ文

```
#pragma nonrec
```

を置いておくとよいでしょう．

B)PDP-11 コンパチブルモードと MSX-C デフォルトモード

MSX-C では，MSX-C デフォルトモードか PDP-11 コンパチブルモードかによって，2 種類の異なった算術変換規則が適用されます．

PDP-11 コンパチブルモードでは，標準 C と同様の算術変換が行われます．ただし，char は符号なしとして扱われます．

MSX-C デフォルトモードでは，新たな 2 つの型，整数定数型(intconst)と文字定数型(charconst)が導入されています．これらは通常はそれぞれ int 型，char 型と同じように扱われますが，変換規則が適用されるときには別の型として扱われます．また，このモードでは char 型から int 型への自動型変換は行われません．

特に指定されない限り，コンパイルは MSX-C デフォルトモードで行われます．PDP-11 コンパチブルモードでコンパイルする場合には，プログラムの先頭で

```
#pragma pdp11mode
```

を指定しなければなりません．この指定はプログラムの先頭でしかもプログラム中で 1 回だけしか行うことができません．また，MSX-C デフォルトモードでコンパイルされたオブジェクトコードとの互換性がなくなるため，PDP-11 コンパチブルモードを利用する場合には，標準ライブラリ関数を含めたすべてのソースファイルをこのプリプロセッサ命令を指定してコンパイルしなければなりません．

MSX-C のデフォルトモードでは，効率のよいオブジェクトコードを生成するために，バイト算術演算をサポートしています．これは，char 型（TINY，BOOL も同じ）同士の演算はワードに拡張されずにバイトのままで行われるため，8 ビットプロセッサに対してより高速で小さいオブジェクトコードが得られるという重要な特徴があります．また，論理演算と関係演算（&&，||，!，=，!=，>，<，>=，<=）の結果も char 型となります．

ただし，MSX-C の char 型は符号なしとして扱われ，0 から 255 までを表現できるだけですので，負になるべき演算や，演算によるオーバーフローやアンダーフローが起こる場合には，結果が正しく反映されません．このような場合には，int や unsigned にキャストして演算を行うようにしてください．

MSX-C のデフォルトモードの算術変換規則をまとめておきます．これは「通常の数値変換」と

呼ばれるもので，下の表で，いずれかのオペランドの型が(1)である場合，他方のオペランドの型が(2)に変換され，結果は(3)の型になります．なお，オペランドの優先順位は表の左の数字が小さいほど，高くなります．

優先順位	(1)	(2)	(3)
1	unsigned	unsigned	unsigned
2	int	int	int
3	char	char	char
4	intconst	intconst	intconst
5	charconst	charconst	charconst

PDP-11 コンパチモードの算術変換規則は次のとおりです．ただし，char のオペランドはすべて int に変換されます．

優先順位	(1)	(2)	(3)
1	unsigned	unsigned	unsigned
2	int	int	int

C)# if 文の非サポート

MSX-C では# if プリプロセッサ文がサポートされていません．しかしたいていの場合には，# if 文の代わりに# ifdef や# ifndef を使用することができるでしょう．また，MSX-C コンパイラはプログラム中で決して実行されることのない部分に対するコード生成を行わないので，if 文によって# if 文と同じ効果を得ることもできます．たとえば次のプログラム例(1)は(2)のように書き換えることができます．

(1)

```
#if sizeof(FCB) != sizeof(char[37])
        OpenMSX();
#else
        OpenOTHER();
#endif
```

(2)

```
if (sizeof(FCB) != sizeof(char[37]))
        OpenMSX();
else
        OpenOTHER();
```

(2)のように記述することで実際には OpenMSX ( ) ; の行か，OpenOTHER ( ) ; の行のどちら

か一方しかコード生成はされません.

ロ)コンパイラの内部的な制御

ここではコンパイラによる変数のレジスタ割り付けの制御やオプティマイズの制御について説明します.

MSX-C コンパイラは変数のレジスタ割り付けをサポートしているので, 効率の良いオブジェクトコードを生成します. 標準 C では, 記憶クラス register の変数のうち, はじめのいくつかの変数だけがレジスタ内に置かれます. (いくつの register 変数がレジスタ内に置かれるかは機種に依存する). それに対して MSX-C コンパイラは, プログラムのデータを解析することにより, 最適なレジスタ割り付けを自動的に行ないます. したがってプログラマは, どの変数を register と宣言するべきか悩む必要がありません. どの変数をレジスタに割り当てるかはコードジェネレータが判断するので, register 宣言は特別な意味を持たず auto と全く同一視されます (register 宣言をすること自体は可能).

プログラム中の各関数に対して, 記憶クラス auto または register を持つもののうち, 単純な型の変数のはじめの 16 個が, レジスタ割り付けの候補となります. ここで単純な型とは, int, char, unsigned とポインタのことです.

C では変数のアドレスを取り出すということができますが, レジスタ割り付けの対象となる変数の場合には問題が出てきます. なぜなら, レジスタの”アドレス”というものは存在しないからです.

例えば,

```
1:      int n;
2:
3:      n = 10;
4:      scanf("%d", &n);
5:      printf("%d", n);
```

というプログラムで, 4 行目で 100 が入力されると, 変数 n には 100 が代入されます. コンパイラは, ある変数に対してアドレスがとられると (4 行目), プログラム中のその文脈の前後ではその変数の値をレジスタではなくメモリに置くようにします. したがって関数のパラメータとして変数のアドレスを渡すという場合には値はメモリに置かれるため, レジスタ割り付けによって起こる問題はありません.

ここで, 次のようなプログラムを考えてみることにしましょう.

```
 1:    #include <stdio.h>
 2:    #pragma nonrec
 3:
 4:    main()
 5:    {
 6:            int n;
 7:            int *p;
 8:
 9:            p = &n;
10:            n = 10;
11:            *p = 100;
12:            printf("n = %d¥n", n);
13:    }
```

このプログラムの場合，12行目で表示されるnの値は，100でなければなりません．ところが実際にコンパイルして実行してみると，"n = 10"と表示されてしまいます．つまり，pによって間接的にnのメモリ上の位置を参照してその内容を100に変更しているにも関わらず，nの値は10のままになっているわけです．この原因を探るため，生成されたアセンブリ言語のリストを見てみましょう．

```
 1:    ;       MSX-C ver 1.20x  (code generator)
 2:
 3:            cseg
 4:    ?59999:
 5:            defb    110,32,61,32,37,100,10,0
 6:            dseg
 7:    ?26:    defs    2
 8:            cseg
 9:
10:    main@:
11:            ld      hl,?26
12:            ld      de,10
13:            ld      (hl),100
14:            inc     hl
15:            ld      (hl),0
16:            push    de
17:            ld      bc,?59999
18:            push    bc
19:            ld      hl,2
20:            call    printf
21:            pop     bc
22:            pop     bc
23:            ret
24:
25:
26:            public  main@
27:            extrn   printf
28:
29:            end
```

このリストからわかるように，nの値がレジスタDEとメモリ（HLで指されている）の両方にとられています．13行目から15行目でメモリ中のnの値が100にされているのですが，コンパイラはそれを無視して，レジスタDEの値をnの値としてプッシュし，printf()に渡しています．したがって，"n = 10"として表示されてしまったわけです．

これは，前述のように「レジスタのアドレス」というものが存在しないために起こった副作用です．しかしこのような例はまれであり，他の表現に書きかえることができますが，もしどうしてもこのようなことをやりたいのであれば，レジスタへの自動割り付けを禁止し，すべての変数をメモリに置くように指定しなければなりません．そのためには

```
#pragma noregalo
```

をレジスタ割り付けを禁止したい関数の前に，そして

```
#pragma regalo
```

を関数の後に付けてコンパイルして下さい．この指定があると，レジスタ割り付けを行う場合に比べて効率は若干低下しますが，コンパイラはその関数の中のすべての変数をメモリに置きますので，上の例もうまく処理されます．

また，コンパイラのオプティマイズの制御をユーザーが行うことができ，そのためには次の2つのプリプロセッサ文を使用します．

```
#pragma optimize time
#pragma optimize space
```

前者はオブジェクトプログラムの速度効率を重視した指定です．つまり，多少オブジェクトプログラムが大きくなっても，実行速度を速くしたい場合に指定します．

後者はオブジェクトプログラムのサイズを重視した指定です．つまり，多少実行速度が遅くなっても，オブジェクトプログラムのサイズを小さくしたい場合に指定します．

これはたとえば，ユーザーからのキー入力要求などといった，速度を無視してよい部分では後者，ループや数値演算を多用している部分では前者を使用するというように，適宜使い分けることによって効率が向上します．

3.1.2　関数の宣言

A)暗黙の宣言

標準Cでは，宣言されていない関数が現れた時にはそれは『int型を返す関数』であると仮定され，パラメータリストにあるが宣言されていないパラメータはint型であると仮定されます(暗黙の宣言)．しかしながらMSX-Cでは，「すべての識別子を使用する以前に必ず宣言をする」という良いプログラミングスタイルを推奨するために，これらの暗黙の宣言が行なわれないようになっています．

すべての標準ライブラリ関数に関する宣言は，ヘッダファイルstdio.hなどに含まれているので，いちいち宣言しなくとも使用することができます．そのため，すべてのユーザープログラムは，冒頭でヘッダファイルstdio.hを#includeによってインクルードしなければなりません．

しかしパーサ(CF)で-fオプションを指定することによってこれらの暗黙の宣言が行なわれるようになるので，標準Cとの互換性を保つことができます．

B)可変パラメータ関数と固定パラメータ関数

MSX-Cには可変パラメータ関数と固定パラメータ関数という2種類の関数種別があります．前者はパラメータの数や型が可変な関数で，後者はパラメータの数と型が固定されている関数で，これはMSX-Cで新たに導入された概念です．この区分に従うと，標準Cの関数はすべて，本質的には可変パラメータ関数です．MSX-Cの場合でも，たとえばprintf()，scanf()などは可変パラメータ関数の例です．

MSX-Cで固定パラメータ関数という概念が導入されたのは，そのオブジェクト効率の良さからです．つまり，可変パラメータ関数がそのすべてのパラメータの受渡しにスタックを使用するのに対し，固定パラメータ関数は最初の3つのパラメータをレジスタに入れて渡すために，効率が大きく向上します．

MSX-Cでは標準Cと同様の関数宣言は，すべて固定パラメータ関数の宣言と見なされます．標準Cの関数は可変パラメータ関数ですが，実際にはほとんどの関数は固定パラメータであるため，互換性の問題は生じません．

一方，可変パラメータ関数は（主にパラメータのアクセス方法が）ターゲットマシンに依存するために，移植性がありません（移植性がないのは可変パラメータ関数自身であり，それを呼び出す側の関数は移植可能)．

C)可変パラメータ関数の定義

可変パラメータ関数と固定パラメータ関数を区別するため，MSX-Cでは宣言子(declarator)と抽象宣言子（abstract declarator）の構文は次のようになっています．

宣言子：

識別子
(宣言子)
＊宣言子
宣言子 ()
宣言子 (.)
宣言子 [定数式 opt]

抽象宣言子：

空
(抽象宣言子)
＊抽象宣言子
抽象宣言子 ()
抽象宣言子 (.)
抽象宣言子 [定数式 opt]

『宣言子 (.)』と『抽象宣言子 (.)』が可変パラメータ関数を表わすために新たに追加されています．これに対して，従来の記法『宣言子 ()』と『抽象宣言子 ()』は，固定パラメータ関数を示します．

このような可変パラメータ関数の記法は，宣言，キャスト，sizeof(型)の場合に使われます．たとえば次の宣言は，可変パラメータ関数 func ( )と可変パラメータ関数へのポインタ fp を定義しています．

```
int func(. ), (*fp)(. );
```

可変パラメータ関数を使用する前には，このような宣言をしなければなりません．標準ライブラリ関数のうち，

```
execl(), execlp(), fopen()
printf(), fprintf(), sprintf()
scanf(), fscanf(), sscanf()
```

は，可変パラメータ関数です．しかしこれらの関数に対する宣言はヘッダファイル stdio.h と string.h に含まれており，しかも可変パラメータ関数の呼び出しは固定パラメータ関数の場合とまったく同じなので，可変パラメータ関数を自分では書かないユーザーは標準ライブラリ関数を意識せずに使うことができます．

次に，実際の可変パラメータ関数の書き方を説明します．

まず関数本体の定義の前に，その関数が可変パラメータであることを示す宣言をしなければなりません．可変パラメータ関数 func ( )の典型的な定義は次のようになります．

```
int     func(. );       /*  必ずこの宣言をしなくてはならない  */

int     func(nargs, args)
int     nargs, args;
{
        ......
}
```

上の例では，関数 func ( ) は本体定義の前に可変パラメータであると宣言されています．もし，宣言

```
int     func(. );
```

を忘れた場合には，func ( ) は nargs と args という 2 つの int 型のパラメータを持つ固定パラメータ関数になってしまいます．

可変パラメータ関数を呼び出すと，実際のパラメータのほかに，もうひとつの暗黙のパラメータが渡されます．このパラメータは nargs と呼ばれ，実際のパラメータの個数を値として持っています．nargs によって，可変パラメータ関数は実際にいくつのパラメータが呼び出し側から渡されたかを知ることができます．また，可変パラメータ関数の側で知ることのできるのは渡されたパラメータの数だけであり，それぞれのパラメータの型を知ることはできません．

呼び出し側によって渡された実パラメータをアクセスするために，もうひとつのパラメータ args を宣言します．1 番目のパラメータ（もし存在するならば）は，args と同じアドレスに入ります．すなわち，1 番目のパラメータは args そのものとなります．2 番目以降のパラメータは，アドレスが増える方向に向かってメモリ上に順に格納されます．つまり 2 番目のパラメータは〔args のアドレス＋2〕番地に，3 番目のパラメータは〔args のアドレス＋4〕番地にというように，それぞれ格納されます．

パラメータの格納されるアドレスは次の表のようにして計算できます．

実パラメータ	アドレス
1 番目	args のアドレス
2 番目	args のアドレス＋2
3 番目	args のアドレス＋4
n 番目	args のアドレス＋（n-1）＊ 2

実パラメータは，上の表に示したアドレスによってアクセスすることができます．たとえば，1 番目と 2 番目のパラメータをそれぞれ char として参照するには，次のようにします．

(char)args　　：　1 番目のパラメータを char として参照する
(char) * (&args + 1)　：　2 番目のパラメータを char として参照する

この例では，argsはint型であると宣言されているので，&argsは，『int型へのポインタ』になります．したがって，&args+1という式において，定数1はsizeof(int *)倍（つまり2倍）され，〔argsのアドレス+2〕番地が参照されます．同様にして，他の型のパラメータもアクセスすることができます．

ここで簡単な可変パラメータ関数の例を2つ示しましょう．関数sum()は（可変個の）int型パラメータの和を値として返します．関数cmaxは，可変個のTINY(char)型のパラメータのうち最大のものを返します．

```
#pragma  nonrec
typedef char    TINY;

int     sum(. );
TINY    cmax(. );

int     sum(nargs, args)
int     nargs, args;
{
        int     i, *p;

        for (i = 0, p = &args; nargs--;)
                i += *p++;
        return (i);
}

TINY    cmax(nargs, args)
int     nargs;
TINY    args;
{
        int     *p;
        TINY    max;

        for (max = 0, p = (int*)&args; nargs--; p++)
                if (max < (TINY)*p)
                        max = (TINY)*p;
        return (max);
}
```

他の可変パラメータ関数の例としては，標準ライブラリ関数printf()，scanf()，fopen()などが挙げられます．

3.1.3　その他

A)float，double，longの非サポート

MSX-Cでは，現在のところfloat型，double型，long型がサポートされていません．しかし，標準Cとの互換性を保つため，これらの識別子は予約語となっており，ユーザーが変数名として使用することはできません．

B)ビットフィールドの非サポート

構造体と共用体のビットフィールドはサポートされていません．適切なビットごとの論理演算で置き換える必要があります．

C)定数式の制限

次の場所にくる定数式には，sizeof単項演算子を含めることはできません．

(1) caseの直後（caseのラベルとして）
(2)配列宣言での配列の大きさの定義

ただし，変数の初期化要素中の定数式にはこのような制限はありません．この制限は，中間言語（Tコード）をターゲットマシンのハードウェアに依存しないようにするために設けられたものです．

D)構造体，共用体のメンバ名

MSX-Cでは，同じメンバ名を異なる構造体や共用体に対して使用することができます．つまり，構造体と共用体のメンバ名の有効範囲が標準的なCと異なっているのです．標準的なCでは同じメンバ名2つ以上の別々な構造体や共用体に使用することができるのは，それらのオフセットと型が等しい時に限られています．

たとえば，次の宣言について考えてみましょう．

```
struct node {
        char    *word;
        int     count;
        struct node *next;
} pool[1000],*p;

struct noad {
        int     atr;
        struct noad *next;
} table [10], *q;
```

標準Cでは，このような宣言は正しくありません．なぜならば，メンバ名nextが2つの構造体nodeとnoadに現れるにもかかわらず，それらの型とオフセットが一致していないからです．しかし，MSX-Cではこの宣言は正しい宣言です．

```
p->next
```

がstruct node型のメンバを表すということに注意して下さい（pはstruct node型へのポインタであるため）．このように，構造体のメンバがより厳格に型付けされているので，正しいメンバを選択することができます．つまり，"." 演算子や" ->" 演算子の左側には，右側のメンバを含む構造体そのものやそれへのポインタがこなければなりません．

なお，異なる構造体や共用体で同じ名前のメンバが使用できるというのはMSX-C特有の拡張ではなく，新しいCコンパイラのほぼ共通した特徴であるため，これによって互換性が損なわれることはほとんどありません．

E)ポインタ，整数間の相互代入

ポインタと整数の間では，キャストを使わないかぎり演算を行うことはできません．また，ポインタ同士の演算でもそれらのポインタが異なる型を指す場合は，やはりキャストが必要になります．

たとえば，

```
int     i;
char    *p;
int     *q;
```

という宣言のもとで，次の4つの文は，いずれもエラー扱いとなります．

```
p = i;
if (p == i) ... ;
q = p;
if (p == q) ... ;
```

はじめの2つは，ポインタと整数の間の代入・比較ですのでエラーとなります．また，後の2つは，pもqもポインタではありますが，そのポインタの指している型が異なっていますので，同様にエラーとなります．このエラーは，キャストを付けるか，パーサ(CF)に対して-tオプションを指定することにより避けることができます．キャストで解決する場合には例えば次のようになります．

```
p = (char *)i;
if (p == (char *)i) ... ;
q = (int *)p;
if (p == (char *)q) ... ;
```

一方，整定数とポインタ間の型変換は許されています．なぜなら，Cでは『どこも指していないポインタ』として定数0（NULL）を使うという慣習があるからです．
例えば，

```
char    *p;
if (p != 0)
        p = 0;
```

はエラーとはなりません．

このように，MSX-CではInt型とポインタとの混用は許されていません．これは型に対してルーズなプログラムを書く人には面倒なことのように思えるかもしれません．しかし，型チェックをきびしくおこなうことはプログラミングスタイルとして推奨すべきことでありこれによって，プログラムのバグを少なくすることができます．

3.2 アセンブリ言語とのリンク

アセンブリ言語とのリンクを考える場合には，はじめにMSX-Cの言語仕様，ハードウェア特性を頭に入れておく必要があります．特に，データ長やデータの格納順が問題になります．

16ビット長のデータタイプでは，下位バイトが初めにストアされ，次の番地に上位バイトがストアされます．つまり，16ビット長データの内部表現は8080系の標準的なデータ表現と同じです．また，データ長については下の表のとおりであり，次の点に注意してください．

・MSX-Cでは，ポインタは16ビット長であり，したがってint型と同じ長さを持つ
・short型とint型は，まったく同一のものとして扱われる

表　基本データ型の長さと範囲

型	長さ	範囲
char	8	0 ～ 255
short	16	-32768 ～ 32767
int	16	-32768 ～ 32767
unsigned	16	0 ～ 65535

3.2.1 パラメータの渡し方

アセンブラへのパラメータの渡し方は，固定パラメータ関数の場合と可変パラメータ関数の場合でそれぞれ異なっています．

まず，通常の関数（固定パラメータ関数）の場合，はじめの3つのパラメータはレジスタに置かれ，4番目以降のパラメータはスタックに積まれて渡されます．1番目のパラメータはchar型の場合にはAレジスタに置かれ，それ以外ならばHLレジスタペアに入れられます．2番目のパラメータはchar型の場合Eレジスタに，それ以外ならばDEレジスタペアに置かれます．3番目のパラメータはchar型の場合Cレジスタに，それ以外ならばBCレジスタペアに置かれます．4番目以降のパラメータは，スタック上に逆順に積まれます．さらに，スタックの先頭には戻り番地が積まれます．スタック上に置かれるパラメータはすべて2バイト長です．char型のパラメータの場合には値は下位バイトに入り，上位バイトの値は使用されません．

PDP-11コンパチモードが指定されている時には，char型のパラメータの扱いが異なります．このモードでは，char型のパラメータはint型に自動的に変換され(上位バイトは0になる)，他の型と同じ扱いを受けます．

これらの規則をまとめると，次の表のようになります．

表　パラメータの渡し方（固定パラメータ関数）

パラメータ	1番目	2番目	3番目	4番目	5番目		n番目(n>3)
char型	A	E	C	(SP+2)	(SP+4)	・・・	(SP+2n-6)
その他の型	HL	DE	BC	(SP+2) (SP+3)	(SP+4) (SP+5)	・・・ ・・・	(SP+2n-6) (SP+2n-5)

注意 PDP-11 コンパチモードでは，char型のパラメータは自動的にint型に変換され他の型と同じものとして扱われる．

例

関数呼び出し

```
sub('A', 0xABCD, 10, '0', -3);
```

によって，パラメータは次のように設定されます．

	デフォルトモード	PDP-11コンパチモード
A	41H	——
B	00H	00H
C	0AH	0AH
D	ABH	ABH
E	CDH	CDH
H	——	00H
L	——	41H
(SP+2)	00H	00H
(SP+3)	——	00H
(SP+4)	FDH	FDH
(SP+5)	FFH	FFH

一方，可変パラメータ関数の場合にはこれとは異なった値の受渡しが行われます．

可変パラメータ関数が呼び出される時には，すべてのパラメータはスタック上に逆順に積まれます．そしてスタックの先頭には戻り番地が積まれます．HLレジスタペアには，パラメータの個数が入っています．おのおののパラメータは2バイトを占めます．MSX-Cのデフォルトモードではchar型のパラメータは下位バイトに置かれ，上位バイトの値は不定で，使用されません．PDP-11コンパチモードでは，char型のパラメータは自動的にint型へと変換されます(この際に上位バイトは0が入れられます)．他の型のパラメータは，8080系プロセッサの標準的な規則（つまり，下位バイトが先にストアされ，その次に上位バイトがストアされる）に従って納められています．

可変パラメータ関数とリンクするためには，あらかじめ，

```
int     sub(. );
```

のような宣言をしなければなりません．

これらの規則をまとめると，次の表のようになります．

表　パラメータの渡し方（可変パラメータ関数）

パラメータ	nargs	1番目	2番目	3番目	・・・	n番目
char型	HL	(SP+2)	(SP+4)	(SP+6)	・・・	(SP+2n)
その他の型	HL	(SP+2) (SP+3)	(SP+4) (SP+5)	(SP+6) (SP+7)	・・・	(SP+2n) (SP+2n+1)

注意 PDP-11 コンパチモードでは，char 型のパラメータは自動的に int 型に変換され，その他の型と同じものとして扱われます．

例

関数呼び出し

```
sub(1, 'A', -3);
```

によって，パラメータは次に示すように設定されます．ただし，あらかじめ

```
int     sub(. );
```

という宣言がされているものと仮定します．

	デフォルト モード	PDP-11 コンパチモード
H	00H	00H
L	03H	03H
(SP+2)	01H	01H
(SP+3)	00H	00H
(SP+4)	41H	41H
(SP+5)	———	00H
(SP+6)	FDH	FDH
(SP+7)	FFH	FFH

3.2.2　値の返し方

関数から呼び出し側に戻るときにレジスタに値を入れることによって，関数値を返すことができます．関数の返す型がchar型であれば値はAレジスタに入れられ，その他の場合にはHLレジスタペアに値を返します．しかし，PDP-11コンパチモードでは関数の返す型にかかわらず，常にHLレジスタペアに値を戻します．2つ以上の値を返す場合には，通常のCプログラムで行なわれるように変数へのポインタをパラメータとして渡して下さい．

3.2.3　アセンブラ，Cのルーチンの相互呼び出し

アセンブラとCの間でルーチンの呼び出しを行う場合に，関数の呼び出し側とその関数から呼び出されるサブルーチンは，それぞれ以下に述べられる規則に従わなくてはなりません．パラメータの渡し方の規則（関数へのパラメータの渡し方と関数からの値の返し方）については，これまでの記述を参照してください．

A)呼び出し側の処理

(1)まず，呼び出し側はおのおののパラメータを評価し，それらをレジスタに入れるかあるいはスタックに積みます．パラメータがレジスタにおかれるかスタックに積まれるかは，「3.2.1 パラメータの渡し方」を参照して下さい．

(2)次に，呼び出し側は実際に関数を呼び出します．通常CALL命令が使われます．

(3)関数から戻ってきたら，スタックのつじつまを合わせるためにスタック上に積まれたパラメータを捨てます(POP命令を使う)．すべてのパラメータがレジスタに割り付けられている場合には，何もする必要はありません．また，POPする際にはHLレジスタペアまたはAレジスタに返される関数値を壊さないように注意が必要です．

(4)これで関数の戻り値が（もしあれば）HLレジスタペアかAレジスタに得られています．どちらのレジスタに値が戻ってくるかは，前節までに述べられています．

B)呼び出される関数側の処理

呼び出される関数は，スタックポインタ（SP）の値を壊してはいけません．言いかえると，関数に入ってきた時と関数から抜け出す時とでは，スタックポインタの値が等しくなければなりません．スタックに積まれたパラメータ（もし積まれていれば）をPOPする（捨てる）のは呼び出し側の責任です．呼び出された関数は，パラメータ（レジスタに置かれたもの，スタックに積まれたもののどちらでも）の値を変えることができます．

しかしながら，C言語の関数間でのパラメータの受渡しは値のコピーによる受渡しなので，仮パ

ラメータの値を変えても実パラメータの値は全く影響を受けないことに注意して下さい.

関数の値は，関数の返す型に応じてHLレジスタペアかAレジスタに入れられます．どちらのレジスタが使用されるかは，前節までを参照して下さい.

例1 C言語からアセンブリルーチンを呼び出す場合

この例では，アセンブリルーチンisdigit@はBOOL (char) 型を返す関数として宣言されているので，関数の値はAレジスタに返されます．呼び出し側のexample ( )は，char型のパラメータcをAレジスタに入れてisdigit@を呼び出します.

・呼び出し側（C言語）

```
typedef char    BOOL;   /* BOOL型をcharと定義する */

example()
{
        BOOL    isdigit();
        char    c;
        .....
        if (isdigit(c))
        .....
}
```

・呼び出される関数（アセンブリ言語）

```
public  isdigit@
isdigit@:
        cp      '0'             ; if less than '0' then
        jr      c, false        ;   return FALSE
        cp      '9'+1           ; if less than or equal to '9' then
        ret     c               ;   return TRUE
false:
        xor     a               ; otherwise
        ret                     ;   return FALSE
```

例2 アセンブリルーチンからCの関数を呼び出す場合

この例では，標準ライブラリ関数printf ( )とputs ( )がアセンブリルーチンから呼び出されています．ここではprintf ( )とputs ( )については述べませんが，printf ( )は可変パラメータ関数なので，パラメータはスタックに積まれるということに注意してください.

この例ではパラメータは1つのみですが，2つ以上のパラメータがある時には，それらは逆順（右から左のパラメータの順）にスタックに積まれます．また，パラメータの数はHLレジスタペアに入れられます．(この例では，1がHLレジスタペアに入れられます)．printf ( )から戻って

きた後に，popが1度だけ行なわれます．これは，先程スタックに積んだパラメータを捨てるためです．パラメータをスタックから捨てるのは，呼び出し側の責任です．

一方，puts()はパラメータの個数と型が一定な固定パラメータ関数です．文字列のアドレスが，HLレジスタペアに渡されています．

```
        extrn   printf@
        extrn   puts@

example::
        ld      hl,msg  ; push address of string
        push    hl
        ld      hl,1    ; load # of parameters
        call    printf@ ; formatted output routine
        pop     hl      ; pop the parameter off the stack
        ld      hl,msg  ; load address of string
        jp      puts@   ; string output routine

msg:    defb    'Hello, world',0ah, 0
```

ファイル名をexample.relとするとリンクは次のようになります．

```
A>l80 example,clib/s,crun/s,cend,example/n/y/e:exampl
```

3.2.4　識別子の有効文字数

識別子の名前として，他のモジュールに対して有効になるのは先頭から6文字までです．MSX-Cでは，コードジェネレータ(CG)で-lオプションを指定すると6文字を超えるシンボル名もアセンブラのソースコードとして出力されますが，実際にアセンブラからリンカに渡されるときには6文字を超える部分は切り捨てられてしまいます．したがって6文字までで一意的に識別できる識別子名を使用しなければなりません．

例

標準ライブラリでは，fclose()とfcloseall()の区別をヘッダファイルstdio.h内の#defineで行っています．2つの関数は"fclose"までですでに6文字ですから，そのままリンカに渡されるとどちらも"FCLOSE"になってしまい，バグの原因ともなってしまいます．そこでfcloseall()の方のアセンブラ，リンカに渡される名前を別のものにしています．パーサ（CF）では，大文字・小文字の区別，7文字以上の識別をしているのでこのことができます．

3.2.5 外部シンボル

アセンブリルーチンとCのルーチンとはそれぞれ別のモジュールとなるため，互いのモジュール中のグローバルシンボルを使用する場合には，外部シンボル使用のための宣言を行う必要があります．

具体的には，アセンブリルーチン中でCのシンボルを使用する場合にはextrnなどの宣言を行います．また，他のルーチンからの参照を許すグローバルシンボルについてはpublic宣言を行います．

なお，Cとアセンブラのリンクを行う場合にはCの外部シンボル名の後に"@"が必要です（前掲の例を参照）．また，アンダースコア（"_"）も"@"に置き換えられますので，たとえば"main"は"main@"として，"_exit"は"@exit@"として参照しなければなりません．

3.3 標準Cのソースファイルの移植について

現在C言語を学習するための多くの書籍が発行されています．しかし，それらはUNIX上で動作するものや，MS-DOS上で動作するCの処理系を対象としたものが多いようです．この節では，それらの書籍にあるプログラムを，MSX-Cで正常に動作させるために必要な項目をあげていきます．

3.3.1 バックスラッシュ"\"

バックスラッシュはUNIXで，コマンドラインやCのソースファイルなどで使われるエスケープ文字です．エスケープ文字を使うことで，限られた文字セットでより多くの文字を表現することができます．ですが，この文字はJISの8ビットコードでは定義されていません．JISではその代わりに円記号"¥"が定義されています．

K&Rの「プログラミング言語C」はUNIXの環境があることを想定されており，C言語の記述部分にバックスラッシュが多く使われています．これと同じ事をMSX-Cでするには"\"の代わりに"¥"を使って記述すればいいことになります．つまり，改行文字を表している"\n"は"¥n"と記述します．以下の対応を参照して下さい．

"\n"	"¥n"
"\t"	"¥t"
"\0"	"¥0"
"\\"	"¥¥"

バックスラッシュの代わりに円記号を使うのは，MSX-Cだけではありません．JISの8ビット文字セットを使っているコンピュータでは，言語にかかわらずすべて置き換える必要があります．

3.3.2 #include <stdio.h>

UNIX上で動作するCコンパイラでは，そのソースファイル中にない関数については標準のライブラリを使用することを意味しています．つまり，外部（extern）に存在するという扱いをされます．しかし，MSX-Cではコンパイル時点で宣言がないと，エラーが発生してしまい，それ以降の処理が続けられません．そのためにMSX-Cでプログラムを作成する際は必ずプログラムの先頭でヘッダファイルstdio.hをインクルードする必要があります．

```
#include <stdio.h>

...             /* プログラムを普通に書く */
```

3.3.3 char型とint型

標準Cではchar型とint型は演算，引数として関数を呼び出すときなど，その区別をまったくする必要はありません．しかし，MSX-Cでは8ビットCPUの特徴を生かすために，char型とint型の区別を厳格に行っています．特に関数に引数を渡すときの第1引数は重要です．それは，char型とint型で値を渡すレジスタがまったく別のものだからです．これについての詳細な内容は「3.2.1　パラメータの渡し方」を参照して下さい．ここでは，どうすれば問題が解決されるかを説明します．まず，問題となるC言語のプログラムを見て下さい．

```
main()
{       /*  一般的なCのためのプログラム  */
        int     c;

        while ((c = getchar()) != EOF)
                putchar(c);
}
```

このプログラムは標準入力からの入力を，標準出力に出力するプログラムです．一見，問題がないように思えますが，大きな落し穴があります．それは，標準出力に文字を出力する関数putchar()です．この関数の第1引数はchar型でなければなりません．しかしこのプログラムでは，int型の変数であるcを，そのまま引数として渡しています．つまりint型の値を渡しています．渡す側のレジスタと受け取る側のレジスタが違うので，このままコンパイルして実行しても，何が表示されるのか分かりません．MSX-Cでは次のように修正することが必要です．つまり，キャストを使ってint型の値をchar型に変更します．

```
#include <stdio.h>

main()
{       /*  ＭＳＸ－Ｃのための変更を加えたプログラム  */
        int     c;
```

```
        while ((c = getchar()) != EOF)
                putchar((char)c);
}
```

変更点はputchar ( )の引数が"c"だけだったものが，"(char)c"になった点です．

このような種類（引数の型が合っていない）のミスは，FPCコマンドによって確実に取り除くことができます．

3.3.4 可変パラメータ関数への引数

可変パラメータ関数は，関数に渡す引数の数が定まっていない関数です．可変パラメータ関数の代表的なものはprintf ( )でしょう．このprintf ( )ではchar型の文字や変数のキャラクタコードを表示しようとすると，予期するものと違うものが表示されてしまうことがあります．問題のソースは次のようなものです．

```
main()
{       /* 一般的なCのためのプログラム */
        char    c;

        c = 'A';
        printf("Char code of 'A' is %d.¥n", c);
}
```

このプログラムでは，文字"A"のキャラクタコードを表示しようとしています．実際にコンパイルして実行すると

```
Char code of 'A' is 2625.
```

と表示されてしまいます．(数値2625は機種によって違うことがあります．)2625という値はキャラクタコードの範囲である0から255をはみ出しています．これは，次のような理由から発生します．

(1)可変パラメータ関数は必ずint型のサイズで引数を受け取る．
(2) char型の引数を可変パラメータ関数に渡すときは，int型の下位バイトだけを設定し，上位バイトは不定となる．
(3) printf ( )の"%d"，"%x"などの変換文字はint型の値が引数として与えられたとみる．

つまり，整数を表示する"%d"の指定で，上位バイトが不定のため，予期しない値が表示されていました．これは，char 型をそのまま printf ( )に渡しているために起こります．この問題は次のようにプログラムを変更することで解決できます．

```
#include <stdio.h>

main()
{       /* ＭＳＸ－Ｃ用に変更したプログラム */
        char    c;

        c = 'A';
        printf("Char code of 'A' is %d.¥n", (int)c);
}
```

変更点は，printf ( )に渡す引数"c"を"(int)c"に変更しただけです．char 型の値を printf ( )などに渡すときは，必ずその値を int 型にキャストするようにします．この問題は整数を表示しようとしている変換指定("%d"など)で，char 型を表示した場合のみ起きます．つまり，文字を表示する"%c"の場合には問題は起きません．また，この問題は，FPC コマンドでは発見できませんので注意して下さい．

3.3.5　関数の宣言（関数の前方参照）

定義や宣言されていない関数がコンパイル中に現れると，標準 C は int 型を返すものとしてコンパイルしますが，MSX-C ではコンパイルエラーが発生します．つまり，関数を呼び出すときには使う前に必ず宣言が必要になります．main ( )がファイルの先頭にあると，そこで使われる関数は main ( )より先に宣言が必要です．これについての細かい説明は「1.4.2　c)関数の宣言」を参照して下さい．

3.4　MSX-C　Ver.1.1 と Ver.1.2 の相違点

ここでは Ver.1.1 と Ver.1.2 の違いについて解説します．

3.4.1　コマンドの変更点

(1) Ver.1.2 に付属するコマンド類は，すべて MSX-DOS2 でなければ動作しません．MSX-DOS1 の場合には"This program needs MSX-DOS2"と表示して終了します．

(2)コンパイルやチェックなどをする対象のファイルの指定にドライブ名，パス名，拡張子が指定できるようになりました．コマンドによって作成されるファイルについても同じ様にできます．

(3) CF, CG, MX においての"-e"や"-o"スイッチの後の指定は，アルファベット１文字の時とそれにコロンがある時はそのドライブに，"¥"で終っているときにはそれをディレクトリ名としてそのディレクトリに，また"¥"以外で終っているときにはそれをファイル名として，出力ファイルを作成します．

A)CF 固有の変更点

(1)インクルードファイルのデフォルトのディレクトリが，環境変数 INCLUDE によって指定できるようになりました．これによってヘッダファイルがあるかどうかなどをあまり意識しなくてもいいようになります．そのため，ヘッダファイルが分割されたことも気にならないでしょう．(ヘッダファイルの分割については「4.2.1　ヘッダファイルの分割」を参照して下さい．)

(2)文字列中に漢字が使われていても正しく認識できるようになりました．Ver.1.1 でもコメントには漢字が書けましたが，文字列中に使うと文字によっては文字が変わってしまうことがありました．しかし，コンパイル時にスイッチ"-j"を指定することで文字が変わってしまうのを防ぐことができます．ただし，漢字と半角ひらがなの混在はできません．

(3)"-e"と"-o"の機能が拡張，変更されました．まず重要なこととして，これらのスイッチの後には他のスイッチが指定できない点です．Ver.1.1 では次のようなことができました．

```
A>cf -eaob file
```

これは，エラーファイルはドライブ A に作成し，中間言語ファイルはドライブ B に作成する指定です．このままで Ver.1.2 の CF を実行すると，エラーファイルをカレントディレクトリに aob.dia というファイル名で作成するだけで，中間言語ファイルはソースファイルと同じディレクトリにできてしまいます．Ver.1.1 と同じ動作をさせるには次のようにします．

```
A>cf -ea -ob file
```

これによって Ver.1.1 と同じ動作をします．つまり，スイッチをそれぞれの機能ごとに分割する必要があります．ですがこの制限はこれらのスイッチの後ろに他のスイッチが指定できないだけで，他のスイッチの後ろに"-e"や"-o"があるのは構いません．逆にこの制限をつけることで分かりやすさや，拡張がされているので問題ないでしょう．

この他に"-e"と"-o"はドライブの指定だけでなく，パス名，ファイル名，拡張子の指定まで出来るようになりました．また，"-o"では直後にファイル名を指定しなくてもエラー表示されずに，そのスイッチがなかったものとして解釈されるようになっています．

B)CG 固有の変更点

(1) CF の"-o"スイッチと同じことが CG の"-o"スイッチにも適用されています．つまり，ドライブ名，パス名，ファイル名，拡張子が指定できたり，この後ろには他のスイッチが指定できないなどです．より詳しくは CF の項をご覧下さい．

(2) CG の最中にエラーが発生したときには，出力ファイルであるアセンブラファイルは生成されません．これは，バッチ中で CG を使っているときにアセンブラファイルのある無しによって，動作を決定するのには便利です．

C)FPC 固有の変更点

(1) FPC によってチェックされるファイルがカレントディレクトリに存在しないときには，起動された fpc.com があるディレクトリから探して，見つかればそれを使います．これは標準ライブラリの FPC 用.tco ファイルなど，よく使われるファイルをいちいちカレントディレクトリにコピーをしたり，パス名を指定する必要をなくすためです．fpc.com と lib.tco は対にして同じディレクトリに置いておけば，lib.tco がカレントディレクトリになくても，FPC の実行は次のようにすることでできます．

```
A>fpc file lib
```

D)MX 固有の変更点

(1) MX は，MX 固有のファイルをコマンド内で参照しています．それは arel.bat，crel.bat というファイルです．Ver.1.1 ではこれらのファイルは MX の実行時にカレントドライブに無い場合には実行されませんでした．しかし，Ver.1.2 ではカレントディレクトリになくても，起動された mx.com があるディレクトリを探して見つかればそれを使います．arel.bat，crel.bat の内容はほとんど変えることがないので，mx.com と対にして同じディレクトリに置いておくといいでしょう．変更が必要なときには変更された arel.bat，crel.bat をカレントディレクトリに置いておけば MX はそれを先に見つけるので，変更された方を使うことになります．

(2) MX でも"-o"スイッチの機能が拡張されました．MX の動作の主要なものにファイルを分割したものを作る作業があります．その分割されたファイルをどこに作成するかを指定するのが"-o"スイッチです．これはディレクトリ名で指定しますが，最後が"¥"で終っていなければなりません．これは複数のファイルを作成するのでディレクトリである必要があるからです．"¥"で終っていないときには"-l"スイッチも指定されているときに意味を持ってきます．"-l"スイッチがあると MX はライブラリファイルの作成の手順まで標準出力に出力します．"¥"で終っていない時のファイル名はこの時のライブラリファイルのファイル名の指定

に使われます．ですからライブラリファイルは分割されたファイルと同じディレクトリにしか作成できません．また，この時ライブラリファイルを作成するLIB80も，MSX-DOS2対応版でなければなりません．

3.4.2 標準ライブラリの変更点

(1) Ver.1.2の標準ライブラリをリンクした実行ファイルはMSX-DOS2でなければ動作しません．MSX-DOS1の場合には"This program needs MSX-DOS2"と表示して終了します．

(2)ヘッダファイルが分割されました．しかし，今まで通りプログラムの先頭でstdio.hをインクルードするだけで，すべてのライブラリ関数が使えるようになります．

(3)ヘッダファイルのbdosfunc.hがVer.1.2になってファンクションコール名が大幅に変更されました．これはMSX-DOS2リファレンスマニュアルのファンクションコール名に合わせるのが目的です．しかし，今までに開発したプログラムに対応するために，Ver.1.1のファンクションコール名を使うことができるように配慮されています．それはbdosfunc.hをインクルードする前に"HEADbdosfuncver11"という定数(マクロ)を定義します．これで，そのままコンパイル可能になります．

例

```
#define  HEADbdosfuncver11
#include <bdosfunc.h>
```

(4) Ver.1.1の標準カーネルにあった，シーケンス機能（コマンドの逐次実行）はなくなりました．また，リダイレクト，パイプ機能はCOMMAND2によって処理されるようになりました．argv［0］にはコマンド名が渡されるようになりました．

(5)高水準入出力のバッファリング方法が3種類になりました．MSX-DOS2のファンクションコールに依存してしまう部分もあります．

(6) MSX-DOS2の機能である階層化ディレクトリや環境変数，ファイルハンドルの操作のために関数が追加・変更されています．

(7)漢字対応のための関数（マクロ）が追加されました．追加された関数は2つですが，通常のプログラミングでは十分でしょう．

(8)文字列操作関数では操作する文字数の上限を指定できる関数が追加されました．

3.5　漢字処理の方法

ここでは，MSX-C による漢字処理について解説します．MSX では漢字は漢字モードである時に，キーボードからの入力，画面に表示，プリンタに出力ができます．これらの処理はすべて漢字ドライバが行っているので，漢字ドライバが起動されていないと動作しません．漢字の入力については，漢字ドライバだけの時は単漢字変換による入力ですが，フロントエンドプロセッサがある時には文節変換などの変換方法によって入力することができます．入出力を伴わない漢字処理については漢字モードに依存する割合はずっと減ります．

3.5.1　MSX での漢字の内部表現

漢字を扱うためにはまず，漢字の内部表現を理解していなければなりません．そうでなければ ANK（1 バイトのアルファベット，数字，カタカナのこと）と漢字の区別ができません．

MSX では漢字の内部表現は Microsoft Kanji Encoding Scheme に従っています．これは一般にシフト JIS コードと呼ばれています．シフト JIS コードは他の漢字コードと同じ様に，2 バイトで構成されます．しかし，JIS の漢字を扱うときに必要な漢字シフトコードなどがないので，ANK との相性（区別が簡単）やデータ格納効率がよくなっています．また，JIS コードとの変換も単純に行えます．JIS コードとの関係は下図のようになっています．

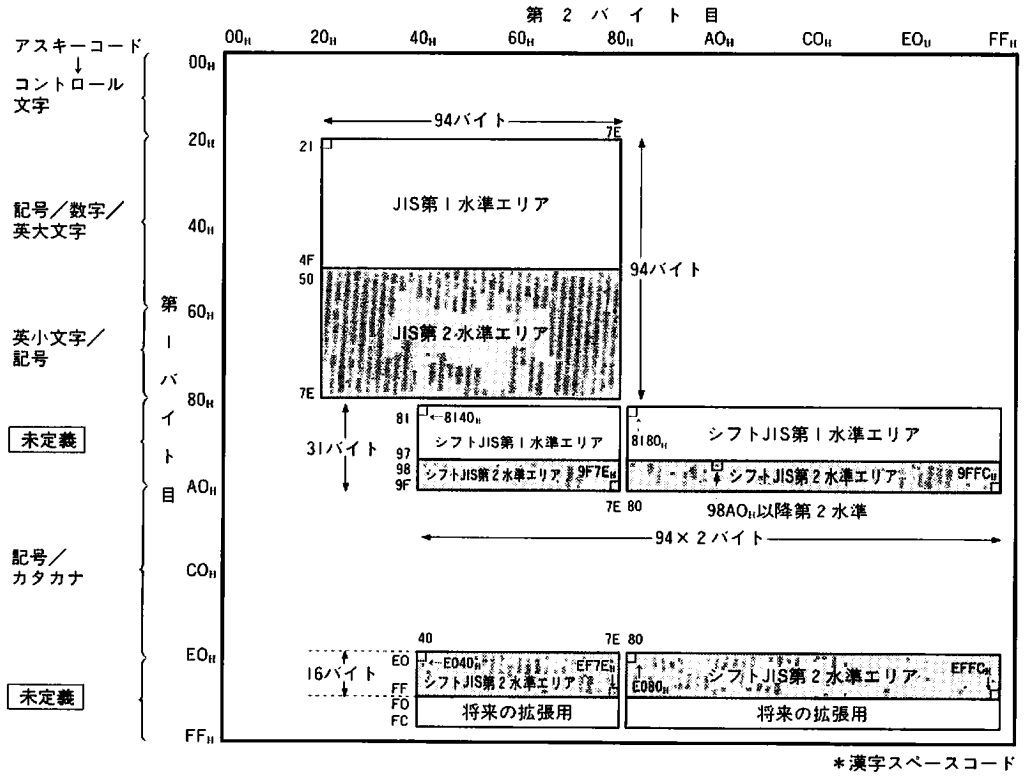


図　JISコードとシフトJISコードの関係

シフトJISコードはJISで決められた1バイト文字の未定義部分を漢字コードの第1バイトに割り当てているので，1バイト目を調べることでそれがANKか，漢字の第1バイトかが分かります．

内部表現であるシフトJISコードは，MSX-DOSだけでなくMSX-BASICでも使用されています．また，16ビットコンピュータの標準DOSであるMS-DOSとも同じものです．ですからMS-DOSで作成した漢字のファイルを，MSX-BASICで表示することも可能です．

MSXを含めてコンピュータでデータの交換に使われるコードは，やはりJISコードである場合が多いため，ここでは参考資料としてMSX-Cで記述された，シフトJISコードとJISコードの変換関数を挙げておきます．sjis2jis()はシフトJISコードからJISコードへの変換を，jis2sjis()はJISコードからシフトJISコードへの変換を行います．それぞれ引数にはunsigned型でシフトJISコード，JISコードを渡します．

```
#include <stdio.h>
/*
        convert a shift-JIS kanji character to JIS
        シフトＪＩＳコード  ->  ＪＩＳコード
*/
unsigned sjis2jis(sjis)
unsigned sjis;
{
        TINY    l, h;

        h = (sjis >> 8) - 0x71;
        if (h > 0x2e)
                h -= 0x40;
        h = h * 2 + 1;
        l = (sjis & 0xff) - 0x1f;
        if (l > 0x60)
                l--;
        if (l >= 0x7f) {
                h++;
                l -= 0x5e;
        }
        return ((unsigned)h * 256 + l);
}

/*
        convert a JIS kanji character to shift-JIS
        ＪＩＳコード  ->  シフトＪＩＳコード
*/
unsigned jis2sjis(jis)
unsigned jis;
{
        TINY    l, h;
```

```
        h = (jis >> 8) - 0x21;
        l = (TINY)(jis & 0xff) + ((h & 1) == 0 ? (TINY)0x1f : (TINY)0x7d);
        if (l > 0x7e)
                l++;
        h = h / 2;
        if (h > 0x1e)
                h += (TINY)0x40;
        h += 0x81;
        return ((unsigned)h * 256 + l);
}
```

3.5.2 漢字モードの区別

MSXでは本当に漢字かどうかは，1バイト調べただけでは判りません．それは，漢字の第1バイト目と1バイトのひらがなの範囲が同じだからです．この区別は作成するコマンドで漢字を使うか，ひらがなを使うかによって行います．次のように分けるのがいいでしょう．

(1)漢字を使う（1バイトひらがなを使わない）時とは，
　コマンドのスイッチで漢字を使う宣言がされている時
　　または
　コマンドのスイッチでひらがなを使う宣言がされていない時
　　または
　漢字モードのとき
(2)1バイトひらがなを使う（漢字を使わない）時とは，
　コマンドのスイッチで漢字を使う宣言がされていない時
　　または
　コマンドのスイッチでひらがなを使う宣言がされている時
　　または
　漢字モードでない時

(1)，(2)の1番目の条件は漢字があまり使われないコマンドの時に使用し（漢字の時には漢字の指定が必要），2番目の条件は漢字が主に使われるコマンドで使います．これらの分け方は画面の漢字モードによらず，指定できる点が利点です．ただし，コマンドによって漢字の扱いのデフォルトを変えると，漢字のためのスイッチをつけるのか，いらないのかなどの混乱が生じるので統一した方がいいでしょう．漢字を使うスイッチとひらがなを使うスイッチ両方をコマンドが受け付けるようにするのもひとつの解決方法です．MSX-CのCFコマンドは，デフォルトはひらがなを使うモードで，漢字を使う場合には"-j"スイッチが必要であるように設定されています．

3番目の条件は，コマンドがすべて自動的に判定してくれる方法です．この場合，現在の漢字モードがどうなっているかをコマンドが知っている必要があります．ですが，MSX-DOS2のファンクションコールを上手に使うことで，直接漢字モードを知らなくても処理は可能です（¥CHKCHR(5DH)などを使う）．

これ以外にも現在の漢字モードをデフォルトにして，漢字用，ひらがな用のそれぞれのスイッチを作ったり，環境変数を見て決めるなどでもいいでしょう．

3.5.3 漢字の処理

ここでは，処理するものは漢字は含められますが，1バイトのひらがなを含まないものとして説明します．

シフトJISコードの場合，文字列の先頭から1バイトずつ調べなければ，ある1バイトの文字がどういう文字（漢字の1バイト目とか，ANKであるとか）だか判りません．処理として基本的には次のようにします．文字列の先頭から1バイトiskanji()で調べ，FALSEだったらANKの処理をします．TRUEだったらその文字は漢字の第1バイトなので，もう1バイト（2バイト目）を取ってきて，2バイトの組で漢字に必要な処理をします．この操作を繰り返すことで，漢字対応の文字列操作ができます．

例

```
# in #include <stdio.h>

/* /*  文字列s中のANKの小文字を大文字にする（漢字対応版）  */
cha char    *jstrupr(s)
cha char    *s;
{   {
            char    *head = s;
            char    c;

            while (c = *s) {
                    if (iskanji(c))         /*  1バイト調べる  */
                            /*  漢字なのでそのまま1バイト進める  */
                            s++;
                    else
                            *s = toupper(c);/*  ANKは必要なら大文字変換      */
                    s++;                    /*  次の文字を指すようにする    */
            }
            return (head);
    }
}
```

次にある，chkctype()，nthctypeは，ある1バイトがANKか漢字かを調べる関数です．chkctype()ではひとつ前に調べた文字のタイプを渡して，現在の文字が何かを調べます．これは，プログラムで1文字ずつ文字のタイプを把握する必要がある時に使います．nthctype()は，文字列の途中の文字のタイプを知りたいときに使います．

```
#include <stdio.h>

#define CT_ANK  (TINY)0         /*  ANK                 */
#define CT_KJ1  (TINY)1         /*  漢字の第１バイト    */
#define CT_KJ2  (TINY)2         /*  漢字の第２バイト    */
#define CT_ILGL (TINY)-1        /*  不正な文字，値      */

TINY    chkctype(c, type)       /*  文字の種類を１文字調べる        */
char    c;                      /*  調べたい文字を渡す              */
TINY    type;                   /*  ひとつ前の文字のタイプを渡す    */
{
        switch (type) {
        case    CT_KJ1:
                if (iskanji2(c))
                        type = CT_KJ2;
                else
                        type = CT_ILGL;   /* 漢字の第２バイトが不正 */
                break;
        /*  ANK，漢字第２バイトの次はANK，漢字第１バイトが来る  */
        case    CT_ANK:         /*  ANK                     */
        case    CT_KJ2:         /*  漢字の第２バイト        */
        case    CT_ILGL:        /*  不正な漢字第２バイト    */
        default:
                if (iskanji(c))
                        type = CT_KJ1;    /*  漢字第１バイトだった  */
                else
                        type = CT_ANK;    /*  ANKだった             */
                break;
        }
        return (type);
}

/*  文字列sのnバイト目がどういう文字かを調べる  */
TINY    nthctype(s, n)
char    *s;
int     n;                      /*  specify  n >= 0  */
{
        TINY    type = CT_ILGL;
        /*  文字列の先頭の前はCT_ILGLとする         */
        /*  n < 0の時にもCT_ILGLを返す              */
        /*  途中でCT_ILGLや'¥0'になっても無視される  */
        while (n-- >= 0)
                type = chkctype(*s++, type);
        return (type);
}
```

第４章　MSX-C 標準ライブラリ

この章では MSX-C コンパイラに付属する標準ライブラリの特徴について細かく述べられています．個々の関数の詳細については「第６章　標準ライブラリ関数リファレンス」を参照して下さい．

4.1　MSX-C の標準的な型

現在の MSX-C でサポートされている基本型は整数型と文字型とその組合せの構造体・共用体ですが，基準となる ANSI では void などの有用な基本型が採用されています．MSX-C の標準ライブラリでも基本型の種類は変わらないものの，その有用性やプログラムの分かりやすさなどから，あらかじめ標準的な型を定義しています．

BOOL，FD，STATUS，size_t，TINY，VOID の各型がヘッダファイル type.h で定義されています．type.h は他のすべてのヘッダファイルで自動的に読み込まれますので，ユーザーはその存在をほとんど意識する必要はありません．また stdio.h では FILE が，bdosfunc.h では FCB，FIB，LONG，XREG が定義されています．各々の型で使用される値については「4.2.3　ヘッダファイルの内容」を参照して下さい．

4.1.1　BOOL 型

この型は２値論理（TRUE と FALSE，YES と NO など２つの値しか持たない）用のデータ型です．例えば，isdigit ( )関数はパラメータの文字が数字か（TRUE），数字でないか（FALSE）のどちらかを必ず返します（BOOL 型の関数）．この関数値を保持したい場合には BOOL 型で宣言された変数を使用するのがいいでしょう．また BOOL 型は，if 文などの条件式にそのまま変数や関数を書くことで，直感的で分かりやすいプログラムの記述ができます．

BOOL 型はヘッダファイル type.h で定義されています．

例

```
STATUS  SameCase(s)     /* 最初のalphabetのケースにそろえる */
char    *s;
{
        BOOL    upper;

        while (*s && !isalpha(*s))      /* 最初のalphabetのサーチ */
                ;
        if (!*s)
                return (ERROR);         /* alphabetはなかった */
        upper = isupper(*s);
        while (*s) {
                if (upper)      /* 最初のalphabetは大文字だった */
```

```
                putchar(toupper(*s)); /* 大文字にして出力 */
            else
                putchar(tolower(*s)); /* 小文字にして出力 */
        }
        return (OK);                  /* 目的通りの事ができた */
}
```

4.1.2 FCB型

MSX-DOS1のファイルアクセスファンクションコールを使用するのに必要なFCBを，構造体として定義したものです．MSX-DOS2ではアセンブラ言語でもファイルハンドルによってアクセスが可能になったためにほとんど必要ありませんが，旧バージョンのMSX-Cとの互換性のために用意されています．定義されているFCB型はMSX-DOS1用のFCBですので，MSX-DOS2とは完全な互換性はありません．詳しい内容は「MSX2テクニカルハンドブック」，MSX-DOS TOOLSの付属のマニュアルやMSX-DOS2のリファレンスマニュアル141ページを参照して下さい．

FCB型はヘッダファイルbdosfunc.hで定義されています．

4.1.3 FD型

この型は低水準の入出力関数（open()やread()など）に使用されるファイルハンドルを表す型です．標準Cなどではファイルハンドルは負でない整数が使われるので，int型で宣言しますが，ファイルハンドル用の変数であることを強調するためにFD型が用意されています．

FD型はヘッダファイルtype.hで定義されています．

例

```
FD      fd;
char    buf[BUFSIZ];

if ((fd = open("file", O_RDONLY)) == ERROR) {
        puts("File not found¥n");
        exit(1);
}
if(read(fd, buf, BUFSIZ) == 0) {
        puts("File has no data¥n");
        exit(1);
}
```

4.1.4 FIB型

FIB型はMSX-DOS2に固有のもので，ファイルのサーチなどをする時に使用される領域(FIB)を構造体としたものです．詳しい内容はMSX-DOS2のリファレンスマニュアル136ページを参照して下さい．

FIB型はヘッダファイルbdosfunc.hで定義されています．

4.1.5 FILE型

この型は高水準の入出力関数(fopen()やgetc()など)がバッファの制御に使用する構造体です．ユーザーがこの型を使用して宣言するのはほとんどの場合が(FILE型への)ポインタの定義です．また，高水準入出力関数を使って入出力するときには，ファイルの指定はfopen()から返されたFILE型へのポインタによって行われます．FILE型の変数を定義してもそれを使ってファイルの入出力をすることはできません．

FILE型はヘッダファイルstdio.hで定義されています．

例

```
#include <stdio.h>
int     c;
FILE    *fp;

if ((fp = fopen("test.dat", "r")) == NULL) {
        puts("File not found¥n");
        exit(1);
}
while ((c = getc(fp)) != EOF) {
        ...
}
```

4.1.6 LONG型

この型はFCB型のなかで4バイトの領域を確保するために定義されています．実際には使用することはできません．

LONG型はヘッダファイルbdosfunc.hで定義されています．

4.1.7 size_t型

この型は領域や文字列の大きさを表す時に使われます．標準ライブラリではstrlen()の戻り値などで使用されています．

size_t型はヘッダファイルtype.hで定義されています．

4.1.8 STATUS型

この型はBOOL型に似たところもありますが，関数の実行結果の状態を返すものです．STATUS型はおもにERROR（処理失敗）とOK（処理成功）の2つの値を保持します．

STATUS型はヘッダファイルtype.hで定義されています．

例

```
#include <stdio.h>
#include <io.h>

if (unlink("editor.bak") == ERROR)
        puts("Back up file not exist¥n");
```

4.1.9 TINY型

8ビットの数値型で，値として10進数で0から255までが使用できます．この型の利点はint型にくらべて占める空間が小さいことと演算が速いことです．欠点としては表現できる数値の範囲が狭いことです．しかしながら，小さな配列の添字や数回のループ変数などでは十分です．

TINY型はヘッダファイルtype.hで定義されています．

例

```
#include <stdio.h>
TINY    count[10];
TINY    i;
int     c;

for (i = 0; i < 10; i++)
        count[i] = 0;

while ((c = getchar()) != EOF)
        if (isdigit((char)c))
                count[c - '0']++;
for (i = 0; i < 10; i++)
        printf("%d: %d¥n", (int)i, (int)count[i]);
```

4.1.10 VOID型

この型はデータ型ではなくコンパイラ(MSX-Cの場合はプログラマ)への宣言と言えます．関数の定義，宣言の際に使用し，関数の戻り値がないことを意味します．VOID型の値を受け取ったり，返したりはできません．

VOID型はヘッダファイルtype.hで定義されています．

例

```
#include <stdio.h>

VOID    noret()
{
        puts("This function returns no value.¥n");
}
```

4.1.11 XREG型

あるアドレスにレジスタを設定してコールするための関数callxx()があります．そのパラメータとして，レジスタの値を指定するための領域が必要ですが，それがXREG型です．この構造体に値を設定してそのポインタをcallxx()に渡すことになります．

XREG型はヘッダファイルbdosfunc.hで定義されています．

例

```
#include <stdio.h>
#include <bdosfunc.h>
XREG    reg;

reg.bc = (unsigned)_GETDTA;
callxx(BDOS, &reg);
printf("Current DTA is %04x¥n", reg.de);
```

4.2 ヘッダファイル

C言語では通常プログラムを作成する場合，標準ライブラリを使用します．この標準ライブラリ関数を使用するには，プログラム中でその使用宣言をしなければなりません．しかしユーザーが使用宣言を関数ごとにしていたのでは，手間がかかってしまいます．その手間を省き，一括して使用宣言するものがヘッダファイルです．ユーザーは使用する関数が含まれているヘッダファイルを#include命令で読み込むことで，関数ひとつひとつの宣言や，定数の定義をしなくてすみます．

4.2.1 ヘッダファイルの分割

MSX-C Ver.1.2ではヘッダファイルは関数の種類によって分類，分割されています．種類としては高水準および低水準入出力関数，文字列操作関数，メモリ管理関数など様々です．Ver.1.1までのヘッダファイルの構成はstdio.hとbdosfunc.hの2つで，前者はほとんどの関数を，後者はMSX-DOSのファンクションコールを使用する場合に必要な関数や定数の定義をしているものでした．

Ver.1.2で分割されたのは，標準ライブラリの関数が増えたことによる必要ない関数の宣言で，ユーザーのシンボルテーブルを圧迫しないためと，コンパイル時間の短縮のためです．しかし，Ver.1.1までのコーディングスタイルである，「プログラムの先頭でstdio.hのみをインクルードする．」ことに支障をきたさないための配慮がされています．つまり，単純にstdio.hをインクルードした場合にはすべてのヘッダファイルをインクルードします(stdio.hがさらにインクルードする)．分割されたファイルを個々にインクルードするときは"#define　DIVHEADER"という行をCのソースファイルの先頭に入れておきます．これによってstdio.hは不必要な自分自身以外のヘッダファイルをインクルードしなくなります．また，bdosfunc.hは別個にインクルードしなければなりません．

例

(1) Ver.1.1までの方法

```
#include <stdio.h>

main()
{
        ...
}
```

(2) Ver.1.2の方法でstdio.hだけをインクルード

```
#define DIVHEADER
#include <stdio.h>

main()
{
        ...
}
```

(3) Ver.1.2の方法でstdio.hとstdlib.hをインクルード

```
#define DIVHEADER
#include <stdio.h>
#include <stdlib.h>

main()
{
        ...
}
```

(4) Ver.1.2 の方法で direct.h と stdlib.h をインクルード

```
/*
#define DIVHEADER の行は
stdio.hをインクルードしないため必要ない
*/
#include <direct.h>
#include <stdlib.h>

main()
{
        ...
}
```

4.2.2 ヘッダファイルのインクルードの方法

ヘッダファイルは通常まとめて，ひとつのディレクトリに置いておきます．(a：¥include などに置く．) Ver.1.2 の CF ではインクルード (# include) するファイルのデフォルトのディレクトリを指定できるので，それをヘッダファイルのまとめてあるディレクトリに設定します．デフォルトのディレクトリの指定には環境変数 include を使用します．この環境変数にはひとつのディレクトリを設定できます．

例

インクルードのデフォルトディレクトリを a：¥include に設定する．

```
A>set include=a:¥include
```

a：¥include にヘッダファイルを入れておけばほかのディレクトリにヘッダファイルがなくても CF はきちんとコンパイルします．

include には 2 通りの書き方があります．まず，よく使われるものはインクルードするファイルを"<"と">"で囲むものです．この使い方ではインクルードのデフォルトディレクトリをサーチし，あればそのファイルをインクルードします．見つからなければ，その旨を表示してコンパイルを中止します．もう一方の使い方はファイル名を " で囲みます．この指定だと，カレントディレクトリをまずサーチし，あればそれをインクルードし，なければデフォルトのディレクトリをサーチします．それでもなければコンパイルが中止されます．注意することとして，ファイル名にドライブやルートディレクトリからの指定があると，デフォルトディレクトリをサーチしないことです．また，# include でのファイルの指定中のディレクトリの区切りは"¥"はひとつしか必要ありません (# include "¥myhead¥mylib.h"と指定すればいい)．

一般的な# include の使い方としては，標準ライブラリの宣言として使うものには"<"と">"で囲み，ユーザー独自のヘッダファイルの場合は " を使って囲みます．

例

(1)# include <stdio.h>

デフォルトディレクトリをサーチし，あればインクルードします．なければエラー表示してコンパイルが中止されます．

(2)# include "myhead.h"

カレントディレクトリ，デフォルトディレクトリの順にサーチしますが，見つかったところでそのファイルをインクルードします．どちらにもなければ，エラー表示してコンパイルが中止されます．

4.2.3 ヘッダファイルの内容

ここでは，13個あるヘッダファイルに何の宣言があるかを個々に見ていきます．関数の詳細は「第6章　標準ライブラリ関数リファレンス」を，型の説明は「4.1　MSX-Cの標準的な型」を参照して下さい．定数が定義されているヘッダではその説明を行います．定数はプログラムで自由に使うことはできますが，変更することはできません．

A)bdosfunc.h

bdosfunc.hにはMSX-DOS2のファンクションコールを直接実行するファンクションコール名を定義してあります．つまり，標準ライブラリ関数で足りないものを，直接MSX-DOSのファンクションコールをすることで，補うときに使うものが入っています．

型	FCB型，FIB型，XREG型	
関数	bdos ( )，bdosh ( )，bios ( )，biosh ( )，call ( )，calla ( )，callxx ( )	
定数	"_"で始まるもの	これらはMSX-DOSのファンクションコール名をその番号と対応させたものです．
	"BIOS"で始まるもの	bios ( )で使えるファンクション番号です．
	BDOS	MSX-DOSのファンクションコールを呼び出す番地を設定してあります．

注意

bdosfunc.hはVer.1.2になってファンクションコール名が大幅に変更されました．これはMSX-DOS2リファレンスマニュアルのファンクションコール名に合わせるのが目的です．しかし，今までに開発したプログラムに対応するために，Ver.1.1のファンクションコール名を使うことができるように配慮されています．それはbdosfunc.hをインクルードする前に"HEADbdosfuncver11"という定数（マクロ）を定義します．これで，そのままコンパイル可能になります．

例

```
#define  HEADbdosfuncver11
#include <bdosfunc.h>
```

B)conio.h

conio.h にはキーボードからの直接入力関数と I/O ポートに関連する関数の宣言が含まれています.

関数	getch ( ), getche ( ), kbhit ( ), sensebrk ( )
	inp ( ), outp ( )

C)ctype.h

ctype.h は, 文字がどんな文字種かを調べるためのマクロが含まれています. また, 文字を大文字や小文字に変換する関数の宣言が含まれます. isalpha ( ), isupper ( ), islower ( ), isdigit ( ), isspace ( ), iscntl ( ), iskanji ( ), iskanji2 ( ) は, 関数ではなくパラメータ付マクロとして実現されています. これらはパラメータを 2 度以上評価することがあるので, 副作用のある式をパラメータとして渡すと正しく動作しません. ですからこれらの 5 つの関数に, 副作用のある式を渡してはいけません(副作用のある式とは, ＋＋や--演算子, 代入演算子, 関数呼び出しを含む式のこと).

マクロ	isalnum ( ), isalpha ( ), iscntrl ( ), isdigit ( ), islower ( )
	iskanji ( ), iskanji2 ( ), isspace ( ), isupper ( ), isxdigit ( )
関数	tolower ( ), toupper ( )

D)direct.h

direct.h はディレクトリ関係の関数の宣言が含まれています.

関数	chdir ( ), expargs ( ), getcwd ( ), mkdir ( ), rmdir ( )

E)io.h

io.h は低水準入出力関数などファイルハンドルを使用してアクセスする関数の宣言が含まれています.

関数	close ( ), creat ( ), eof ( ), isatty ( ), open ( )
	read ( ), rename ( ), unlink ( ), write ( )

定数	O_RDONLY	open ( )の読み込みモードのための定数
	O_WRONLY	open ( )の書き込みモードのための定数
	O_RDWR	open ( )の読み書き両用モードのための定数
	STDIN	ファイルハンドル0を示します.
	STDOUT	〃　1　〃
	STDERR	〃　2　〃
	STDAUX	〃　3　〃
	STDPRN	〃　4　〃
	STDLST	〃　4　〃　(STDPRN と同じ)

F)malloc.h

malloc.h にはメモリ管理関数の宣言が含まれています.

関数　alloc ( ), free ( ), sbrk ( ), rsvstk ( )

G)memory.h

memory.h にはメモリを直接操作する関数の宣言が含まれています.

関数　memcpy ( ), memset ( ), movmem ( ), setmem ( )

H)process.h

process.h には, プログラムの終了や起動など, コマンドそのものに関わる関数の宣言が含まれています.

関数　execl ( ), execv ( ), _exit ( ), exit ( )

I)setJmp.h

setjmp.h には関数間を越えての分岐のための関数の宣言と型の定義が含まれています.

型　jmp_buf 型
関数　longjmp ( ), setjmp ( )

J) stdio.h

stdio.h には高水準入出力関数の宣言，FILE 構造体の定義，マクロ，定数の定義が含まれています．また，MSX-C　Ver.1.1 までの stdio.h と互換性を保つための他のヘッダファイルのインクルードも含まれています．

型	FILE 型	
マクロ	fileno ( )，feof ( )，ferror ( )	
関数	clearerr ( )，fclose ( )，fcloseall ( )，fflush ( )，fgets ( )	
	flushall ( )，fopen ( )，fprintf ( )，fputs ( )，fread ( )，fscanf ( )	
	fsetbin ( )，fsettext ( )，fwrite ( )，getc ( )，getchar ( )，gets ( )	
	printf ( )，putc ( )，putchar ( )，puts ( )，scanf ( )	
	setbuf ( )，setvbuf ( )，ungetc ( )，ungetch ( )	
定数	BUFSIZ	高水準入出力関数のバッファの大きさは BUFSIZ になります．setbuf ( ) で与えるバッファはこの大きさでなければいけません．
	EOF	ファイルからの入力がエンドオブファイルまで行われたことを示す値です．
	_NFILES	高水準入出力関数で同時にオープンすることができるファイルの最大数が定義されています．これには stdin，stdout，stderr，stdaux，stdprn が含まれています．
	_IONBF	setvbuf ( ) でバッファリングなしを指定する値です．
	_IOLBF	setvbuf ( ) で行バッファリングを指定する値です．
	_IOFBF	setvbuf ( ) でフルバッファリングを指定する値です．
	stdin	標準入力を表すファイル FILE 型へのポインタです．
	stdout	標準出力　〃
	stderr	標準エラー出力　〃
	stdaux	標準補助出力　〃
	stdprn	標準プリンター出力　〃

K) stdlib.h

stdlib.h にはよく使われる関数ですが他のヘッダに分類できないものが，集められています．

関数	abs ( )，atoi ( )，getenv ( )，max ( )，min ( )
	putenv ( )，qsort ( )

L)string.h

string.h には文字や文字列の操作をする関数の宣言を含んでいます.

関数　sprintf ( ), sscanf ( ), strcat ( ), strchr ( )
strcmp ( ), strcpy ( ), strlen ( ), strlwr ( )
strncat ( ), strncmp ( ), strncpy ( ), strupr ( )

M)type.h

type.h には, MSX-C で使われる標準的な型を定義しています. また, その型に使われる値の定数も定義しています.

型　BOOL 型, FD 型, size_t 型, STATUS 型, TINY 型, VOID 型

定数

NULL	どこも指していないポインタとして使います. また, ポインタを返す関数ではエラーとして使われます.
TRUE	BOOL 型用で, 真を表します. FALSE と対になります.
FALSE	BOOL 型用で, 偽を表します. TRUE と対になります.
YES	BOOL 型用で, 真を表します. NO と対になります.
NO	BOOL 型用で, 偽を表します. YES と対になります.
OK	STATUS 型用で, 処理の正常終了を表します.
ERROR	STATUS 型用で, 処理の異常終了を表します.

4.3 標準ライブラリ関数概要

MSX-C Ver.1.2に付属する標準ライブラリはMSX-DOS2上でのみ動作します．標準ライブラリをリンクして作成されたコマンドは，実行前にMSX-DOSのバージョンをチェックし，バージョン2以上でないと"This program needs MSX-DOS2"と表示して，実行されません．

4.3.1 ファイル入出力関数

C言語ではファイルの入出力は3段階（オープン，入出力，クローズ）に分けられます．

(1)最初にファイルをオープンします．この操作でユーザーはオープンの関数(fopen ( )やopen ( ))にファイル名と読み出しか書き込みかのモードを与えます．すると関数は，以後ファイルアクセスに必要な情報を返します．(fopen ( )ではFILE型へのポインタ，open ( )ではFD型の値）ユーザーはこの値を変数に保存しておきます．

(2)次の段階はデータの入出力です．ここでは実際にファイルからデータを読み出すか，書き込むかをします．しかしこれは，ファイルのオープン時のモードの指定と同じでなければいけません．データの読み出しではユーザーは関数に，保存してあるファイルを指定する値，読み出したデータを格納する位置，読み出す量を渡します．(fgets ( )やread ( ))書き込みではファイル指定の値，書き込むデータがある位置，書き出す量を渡します．(fputs ( )やwrite ( ))関数によっては読み込んだデータを返すもの（getc ( )），書き込むデータを渡す関数（putc ( ))もあり，いつも3つの情報を渡すわけではありません．しかしそれは，隠されているだけ（暗黙的に指定している）と言えるでしょう．

(3)最後の段階はファイルのクローズです．クローズの関数（fclose ( )やclose ( ))にはファイル指定の値を渡すだけです．読み出しでオープンされたファイルに対してはただ単純に使用していたファイル管理情報を捨てるだけですが，書き込みモードではもうひとつ行うことがあります．それは，ディスクに書き出していないデータをディスクに書き出すことです．書き出した後は読み出しモードと同じ処理になります．

以上でファイルの入出力が一通り終りました．ファイルは必要であれば複数のファイルを同時に入出力できます．またプログラムが起動されたとき，よく使われるファイルに関してはすでにオープンされているので，すぐにデータの入出力ができます．

ファイル入出力関数は，大きく2つのグループに分類できます．ひとつは高水準入出力関数（fopen ( )，getc ( )など)，もうひとつは低水準入出力関数(open ( )，read ( )など)です．MSX-C Ver.1.2の標準ライブラリでは，UNIXやMS-DOSの環境と同様なリダイレクションとパイプラインがサポートされています．標準入力と標準出力はコマンドラインからファイルに変更できますが，標準エラー出力は，コンソールに固定されており，変更することはできません．

入出力関数では，ディスクファイルの他に，MSX-DOSのデバイスファイルも扱えます．それぞれのデバイスに許される操作は，次の通りです．

表　デバイスファイルの入出力方向

	入力	出力
CON	可	可
PRN	不可	可
NUL	可	可

このうちNULは，ダミー入出力であり，入力は常にEOFが返され，出力は実際には行なわれずにデータは捨てられます．

A)高水準入出力関数

高水準入出力関数は文字単位の入出力が基本であり，テキスト・ファイルの処理が簡単におこなえるようになっています．

高水準入出力では，5つのファイルポインタ，stdin(標準入力)，stdout(標準出力)，stderr(標準エラー出力)，stdaux（標準補助出力)，stdprn（標準プリンタ出力）がヘッダファイルstdio.hで宣言されており，プログラムの起動時にこれらのファイルは，カーネルによって自動的にオープンされます．stderr，stdauxは標準Cでは入力，出力ともに可能ですが，MSX-Cの標準ライブラリでは構造上実現できないので，片方向（この場合は出力のみ）になっています．

高水準入出力関数ではファイルは通常“テキスト”モードで処理されます．テキスト・モードでは入力中のCR-LFの組合せは1文字のニューライン文字（¥n）に置き換えられ，また，EOF文字以降のファイルの内容は無視されます．逆に，出力の場合にはニューライン文字がCR-LFに戻され，EOF文字がファイルの最後に追加されます．ただし，このような変換が望ましくないときには，ファイルを“バイナリ”モードで処理することもできます．

高水準入出力関数を使っている最中に，そのファイルはどのファイルハンドルを使用しているかが必要になるときがあります．例えば，出力先がデバイスであったらキー入力待ちをしたい時などです．そのときにはfileno（）（マクロ）を使えば対応するファイルハンドルが求まります．これで，高水準入出力関数と低水準入出力関数の橋渡しができます．例えば，fopen（）でオープンしたファイルがデバイスかどうかを調べるには

```
if (isatty(fileno(fp)))
        デバイスの処理
else
        ファイルの処理
```

とすればよくなります．ファイルハンドルを得て，read（），write（）もできますが，高水準入出力関数はバッファリングしているのでデータ落ちに注意しなければなりません．

B)高水準入出力関数のバッファリング

高水準入出力関数は入出力するデータをメインRAM(MSX-DOSのTPA)内にバッファリングしています.

バッファリングの動作を簡単に説明します.リード動作においては,まずバッファにデータがあるかどうかをチェックし,あればそれを渡します.ない場合にはディスクからバッファがフルになるまで入力データを読み込みます.その中から要求のあった分を先頭から返します.ライト動作では,出力データをバッファに書きためておき,バッファがいっぱいになった時点でディスクに書き込みます.この書き込みのことをバッファフル時のバッファのフラッシュといいます.

バッファリングにはフルバッファリング,行バッファリング,バッファリングなしの3種類があります.フルバッファリングは上記の動作をそのまま行います.この方法はファイルをfopen()でオープンすると採用される方法です.行バッファリングはテンプレート機能が使える1行単位での入力や,"¥n"が出力されるとバッファをフラッシュする方法です.行バッファリングは入出力の対象がデバイスファイル(コンソールなど)以外ではあまり意味がありません.3番目のバッファリングなしでは,入出力要求を1文字単位で対応します.デバイスファイルの場合にはその時点での状態が見えるので便利です.

バッファリングにはこのように3種類ありますが,実際のファイルの入出力にはread()とwrite()を使っているので,MSX-DOS2の仕様が直接現れてしまいます.つまり,デバイスファイルから入力すると必ず行バッファリングになってしまいます.

ここですこしバッファのフラッシュについて説明しましょう.バッファのフラッシュとは,ある時点でバッファに残っているデータをディスクに書き込むことをいいます.バッファをフラッシュするある時点とは(1)バッファがいっぱいになったとき,(2)行バッファリングのとき"¥n"が出力されたとき,(3)高水準入出力関数でコンソールからの入力をしたとき,(4)ユーザーが明示的にフラッシュしたとき,の4つがあります.(1)から(3)は標準ライブラリが自動で行います.(4)が必要なときとはコンソールにメッセージを途中の状態で表示したい場合でしょう.プロンプトを表示してその行の続きから文字を入力したい場合には(3)によって自動で行われます.

高水準入出力関数を使用する場合には,FILE構造体へのポインタを使用しますが,このFILE構造体はバッファリング制御のための情報が入っています.ユーザーはこの情報のうちバッファサイズ,バッファの位置そしてバッファリングの方法をsetvbuf()で変更することができます.ただし,ファイルをオープン後データの入出力がまだ行われていないときでないと使用できません.

MSX-C Ver.1.2の標準ライブラリでは,main()に制御が渡ったときには5個のストリームが自動的にオープンされています.それらのファイル構造体へのポインタはstdin, stdout,

stderr，stdaux，stdprn です．このうち stderr と stdaux はバッファリングなしで，残りは行バッファリングでオープンされています．

C)低水準入出力関数

低水準入出力関数は，バッファリングせずに直接ディスクにデータを読み書きします．そのためデータ数が少なく，読み書きの回数が多くなると遅くなってしまいます．データがまとまっている場合には，低水準入出力関数でも十分に速い読み書きができます．データの入出力関数が read ( ) と write ( ) しかありませんので，文字単位，文字列という考え方ができません．そのためバイナリデータなどの処理には向いているでしょう．また，Ver.1.1 にあったようなデバイスファイルにアクセスできないというような事はなくなりました．

4.3.2　文字列および文字処理関数

文字に関する関数は 2 種類あります．文字の種類を判定するものと文字や文字列を操作するものです．

文字の種類を判定する関数は，文字を与えてその文字が英小文字かどうかというような判定をするものです．これらの関数はある決まった文字の種類（数値を期待しているときには数字）であるかどうかに使います．この判定をする関数は"is"から始まっています．

もうひとつの関数群は主に文字列を対象にしています．文字列の比較や，文字列中からある文字を探すなど様々です．これらの関数は"str"で始まります．また，文字を大文字や小文字に変換する関数もあります．他に，文字列の長さの制限して比較する関数もあり，それらの関数名は "strn" で始まります．

4.3.3 メモリ管理関数

MSX-C ではメモリ管理関数は 2 種類あります. ひとつは低水準メモリ管理関数, もうひとつは高水準メモリ関数です.

A)低水準メモリ管理関数

低水準メモリ管理関数は単純なメモリ領域の確保関数 (sbrk ( )) とスタック領域の確保関数 (rsvstk ( )) からなります. sbrk ( ) は MSX-C で使うプログラムやデータの領域の最後端から指定された分の領域を確保していきます. rsvstk ( ) はスタック領域のサイズとしてその値をグローバル変数_torelance に保存します. sbrk ( ) で確保される領域は, スタックの方向へ延びていくので, 確保の際は rsvstk ( ) で指定されたスタック領域を侵さないかのチェックをします. CPU のスタックポインタ SP は sbrk ( ) を呼ぶ状況によって変化しますが, 呼ばれたときのスタックポインタの値から_torelance を引いた位置 (下図の B 点) が, 確保しようとしている領域の後端 (A 点) と重ならなければその領域の先頭 (memtop) を返します. 返した後に memtop は A 点にセットされ, 次の sbrk ( ) のために準備をします. (下図は領域確保が成功するときの図です.)

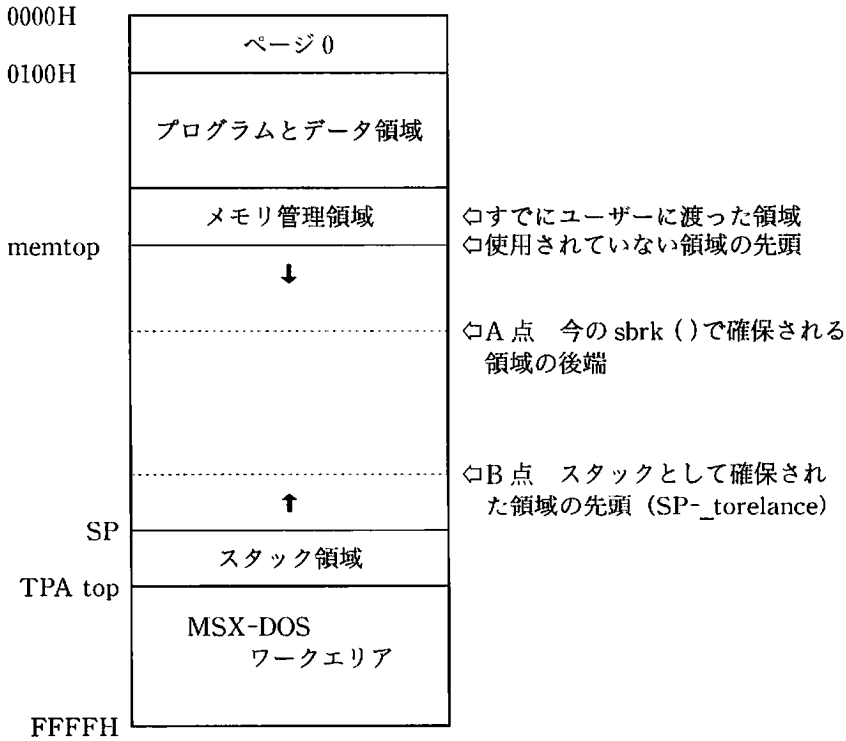


図　メモリ管理関数の割り当て方

B)高水準メモリ管理関数

もう1種類のメモリ管理は，サイズなどの管理する情報を準備してから領域を渡すalloc()と，alloc()によって渡された領域を解放し，もう一度alloc()で使えるようにするfree()の組です．普通のプログラムが実行時にメモリが必要になった場合にはこの高水準メモリ管理関数を使います．高水準メモリ管理関数はファイル入出力のバッファ領域や，getenv()関数などでも使われています．また，メモリ管理関数の低水準と高水準を混ぜて使うこともできますが，高水準だけを使うことでfree()したときに無駄の少ないメモリ利用ができます．

free()で解放された領域でnバイト以上の領域がない場合にはsbrk()で領域を確保しようとします．つまりnバイトalloc()し，直後にそこをfree()してから，(n+m)バイトalloc()しようとすると((n+sizeof(HEADER))＋(n+m+sizeof(HEADER)))以上のメモリが必要になります．

4.3.4 ディレクトリ関数

MSX-DOS2では階層ディレクトリが扱えるようになりましたが，それをサポートするための関数がディレクトリ関数です．カレントディレクトリの移動・獲得，サブディレクトリの作成・削除があります．また，階層ディレクトリをサポートしたワイルドカード（*や?）を実際のファイル名に展開する関数もあります．これらを使えばMSX-DOSのコマンド内から階層ディレクトリを上手に使えることができるでしょう．

4.3.5 プログラム操作関数

プログラムの実行中，エラーなどで処理が続けられない時，プログラムは終了しなければなりません．しかし，関数をいくつも呼んでいたり(ネストしている)，ファイルのオープンもしているなど厄介なことがあります．そのような場合ただちにプログラムを終了する関数も用意してあります．

ひとつのプログラムでは終らない様な処理も考えられます．そのためにプログラム中から別のプログラムに実行を移す関数もあります．

4.3.6 キーボード，I/O関数

高水準の入出力関数はいろいろな機能があり便利ですが，1文字入力ごとに反応を示す必要がある場合には，キーボード直接入力関数というものも用意してあります．．また，MSXの機能をフルに使いたいときには直接周辺のLSIにコマンドを出すI/O関数を用意しています．

4.3.7 機械語，MSX-DOS ファンクションコールサポート関数

MSX-C の標準ライブラリにはさまざまな関数がありますが，MSX-DOS2 では，より多くの機能をサポートしています．それを使うためにファンクションコールサポートの関数も加えました．またその関数は，MSX-C 以外で作成した機械語のプログラムにも利用できます．

4.3.8 メモリ操作関数

メモリ操作は MSX-C の記述でもできますが，メモリ内容のコピーなどは量が多くなると結構時間がかかってしまうものです．高速性が求められるときにはメモリ操作関数を使うといいでしょう．

4.3.9 汎用関数

上記の他にもまだまだ便利な関数があります．環境変数の獲得や設定，データのソートなど，使えば C でのコマンドの範囲が広がるものばかりです．

4.4 UNIX 標準 C ライブラリとの相違点

以下の関数は，UNIX の標準 C に対応する関数があります．ただし，“*”の付いた関数は，幾分仕様が異なっています．また，“-” の付いた関数は，標準 C のサブセットになっていますので利用時には，若干の注意が必要です．

abs	atoi	chdir	clearerr	close *
creat	eof	execl *	exexlp *	execv *
execvp *	_exit	exit	fclose	fcloseall
feof	ferror	fflush	fgets	fileno
flushall	fopen *	fprintf -	fputs	fread
free	fscanf -	fwrite	getc	getchar
getcwd	getenv	gets *	isalnum	isalpha
isatty	iscntrl	isdigit	islower	isspace
isupper	isxdigit	longjmp	mkdir	open
printf -	putc	putchar	putenv	puts *
qsort	read	rename	rmdir	sbrk
scanf -	setbuf	setjmp	setvbuf	sprintf -
sscanf -	strcat	strchr	strcmp	strcpy
strlen	strlwr	strncat	strncmp	strncpy
strupr	tolower *	toupper *	ungetc	ungetch
unlink	write			

以下の関数は，MSX-C にのみ存在します．

alloc	bdos	bdosh	bios	call
calla	callxx	expargs	fsetbin	fsettext
getch	getche	inp	iskanji	iskanji2
kbhit	max	memcpy	memset	min
movmem	outp	rsvstk	sensebrk	setmem

4.5　ライブラリの作成と保守

4.5.1　MSX-C ライブラリの構成

MSX-C には，次の 4 つの .rel ファイルが用意されています．

ck.rel	カーネル呼び出しルーチン
clib.rel	標準ライブラリ（カーネル本体を含む）
crun.rel	実行時ルーチン
cend.rel	標準ライブラリ

ck.rel はカーネルを呼び出すルーチンが含まれています．カーネル本体は clib.rel に含まれており，ck.rel ではそれを呼び出しています．MSX-C における「カーネル」とは，ユーザープログラムを実行する際の環境を整備するためのプログラムです．カーネルは，次のような働きをしますが，処理の大部分は次に述べる CLIB 内部のプログラムを呼び出すことで行っています．

(1)まず，コマンド行を解析し，argc と argv をセットする．
(2)標準入力（stdin），標準出力（stdout），標準エラー出力（stderr），標準補助出力（stdaux），標準プリンタ出力（stdprn）の 5 つのファイルをオープンする．

(3)ユーザープログラム中の関数 main に制御を渡す．
(4) main ( ) の実行の正常終了や exit ( ) の呼び出しによってユーザープログラムの実行が終了すると制御はカーネルに戻り，そのとき開かれているファイルをすべてクローズする．

ck.rel のソースファイルは ck.mac，カーネル本体のソースプログラムは process.c に関数_main ( ) として記述されています．

clib.rel はMSX-Cの標準ライブラリです．すなわち，printf ( )や atoi ( )などの標準ライブラリ関数あるいは内部関数の集まりです．リンク時には，ユーザー・プログラム中で使用している関数だけがライブラリから抽出されます．

ソースプログラムは，次の8個のC言語ファイル，1個のアセンブリ言語ファイルに収められています．

clibc.c	clibmac.mac のスケルトン作成用ファイル
direct.c	ディレクトリ操作関数ソースファイル
io.c	低水準入出力，キーボード関数ソースファイル
malloc.c	メモリ管理関数ソースファイル
process.c	プログラム管理関数ソースファイル
stdio.c	高水準入出力関数ソースファイル
stdlib.c	汎用関数ソースファイル
string.c	文字列操作関数ソースファイル
clibmac.mac	標準ライブラリ関数アセンブラ記述部

ソースプログラムが提供されていることによって，ユーザーは，独自のカーネルを作ったり，標準ライブラリ関数を差し替えたり，それを参考にして独自の関数を追加したりすることが容易になります．ライブラリの作成については「4.5.3　専用ライブラリの作成」を，標準ライブラリの保守については「4.5.4　標準ライブラリの再作成」を参照してください．

また，標準ライブラリ関数に関するもう1つのファイル lib.tco は，FPC（関数パラメータ・チェック・ユーティリティ）で使用するためのものです．このファイルには，すべての標準ライブラリ関数の型と引数に関する情報が含まれています．

crun.rel はCプログラムの実行時に必要なサブルーチンの集まりです．乗算，除算などの(CPUに対応する命令がない)演算は，コンパイラによってCRUN中のサブルーチンの呼び出しに置き換えられます．CRUN は，このように言語仕様とハードウェアの差を補うためのもので，他にシフト演算，符号付き比較などのサブルーチンがあります．ソースファイルは crun.mac です．

cend.rel は標準ライブラリの一部と考えて下さい．これは最後にリンクするモジュールで，以降がヒープ領域となります．ソースファイルは cend.mac です．

4.5.2　ライブラリ保守支援ツールMXについて

ライブラリの作成，保守を行う場合には，MX（モジュール抽出ユーティリティ）を使用して，ユーザーが実行しなければならない処理の効率を大幅に向上させることができます．

MXは，具体的には次のような操作を行います．

(1)中間言語ファイル（.tcoファイル）またはアセンブリ言語ファイルからモジュール（関数）を抽出する．
(2)抽出したモジュールのコード生成，アセンブルなどの手順を標準出力に出力する．

すなわち，複数の関数を含むファイルから任意のモジュールを引きだしたり，モジュールごとに別々のファイルを作成することが可能で，ライブラリの保守，作成が非常に楽になります．また，そのための手順が標準出力に出力されるため，それを.batファイルにリダイレクトすれば，そのままバッチファイルとして使用することができるようになります．

MXの書式を次に示します．

MX　［オプション］　ファイル名　［モジュール名］　...

ファイル名は拡張子なしで与えます．MXは，指定のファイル名を持つ.macファイルを探し，見つかった場合にはそれをアセンブラのソースファイルと見なして作業を開始します．.macファイルが見つからないと，MXは指定のファイル名を持つ.tcoファイルを探し，見つかった場合にはそれをCの中間言語ファイルと見なして作業を開始します．

そのファイルから引き出すべきモジュールは，ファイル名の後に指定します．ここではひとつ，あるいは複数のモジュールを指定することができます．また，モジュール名にはワイルドカード文字（＊と？）を使用することができます．モジュールの指定をまったく行わないと，すべてのモジュールが抽出されます．

MXは実行時に，2つのバッチファイルarel.bat，crel.batがカレントドライブのカレントディレクトリに必要です．この2つのファイルは，MXの出力の変更を容易にするもので，arel.batはアセンブリ言語ファイルから.relファイルへの手順を，crel.batは中間言語(.tco)ファイルから.relファイルへの手順を示しています．MXはファイル中の"%1"をモジュール名に変換して出力します．

カレントディレクトリに，arel.bat，crel.batが見つからない場合には，mx.comのあったディレクトリを探します．それでもない場合は"Skeleton file <filename> not found"と表示して終了します（<filename>はarel.batかcrel.bat）．よく使われる形式のarel.bat，crel.batをmx.comと同じディレクトリに置いておけば，存在をほとんど意識しないですみます．違った形式のものを使いたい場合には，カレントディレクトリにarel.bat，crel.batを置いておけばそれが使われます．

オプション

-l　　LIB80 を呼び出してライブラリファイルを作成する出順も出力.
このオプションを指定しないと,.rel ファイルを作成するところまでで出力を終了します.

-o [path¥]　分割したモジュールをディレクトリ path に抽出する.
ディレクトリの指定は必ず"¥"で終ってなければなりません."¥"で終ってなければ"-l"スイッチ使用時のライブラリファイル名の指定となります.ただし,抽出されるファイルはライブラリファイルと同じディレクトリとなります.このオプションを指定しないと,カレントドライブのカレントディレクトリにモジュールを抽出します.

MX でのエラーメッセージはコマンドの動作についてのメッセージです.例えば,.tco や.mac ファイルが見つからないなどのメッセージです.このようなメッセージが表示されたときには,MX は処理を中止してコマンドレベルに戻ります.実際のエラーメッセージは「8.4　MX のエラーメッセージ」をご覧下さい.

MX の具体的な使用例については,「4.5.3　専用ライブラリの作成」を参照してください.

4.5.3　専用ライブラリの作成

「ライブラリ」には,(1)リンク時にすべての関数を実行ファイルにリンクするもの,(2)リンク時に必要な関数だけを実行ファイルにリンクするようにしたもの,の2種類があります.それぞれの長所と短所を挙げると,次のようになるでしょう.

(1)長所：簡単に作成できる.
　短所：プログラムで使用しない関数も含めて,ファイル中のすべての関数を実行ファイルにリンクしてしまうので,実行ファイルが大きくなってしまう.

(2)長所：プログラムで使用する関数だけをリンクするため,実行ファイルのオブジェクト効率が向上する.
　短所：作成するのに手間がかかる.

通常「ライブラリ」という場合には(2)のようなライブラリを指し,MSX-C の clib.rel, crun.rel もこのような性格を持っています.(1)のようなライブラリのメリットは,手軽に作成できるということ以外にはないため,極力(2)のようなライブラリを作成することが推奨されます.ここでも(2)の説明を中心に行いますが,まずはじめに基礎として(1)のライブラリの作成方法を簡単に述べておきます.

A)すべての関数をリンクするライブラリ

次のような2つの関数power( ), log2( )をライブラリにする，ということを考えてみましょう．

```
int     power(x, e)
int     x, e;
{
        int     y;

        for (y = 1; e > 0; e --)
                y *= x;
        return (y);
}

int     log2(x)
int     x;
{
        int     y;

        for (y = 0; x > 1; x >>= 1)
                y++;
        return (y);
}
```

これらの関数はxlib.cというソースファイルに入っているものとすると，ライブラリを作る手順は次のようになります．

```
A>cf xlib↵
...
A>cg xlib↵
...
A>m80 =xlib↵
...
A>del xlib.mac↵
```

以上の手順によって，xlib.relが作られます．これでライブラリXLIBの作成は終りです．ユーザーのプログラムをprogとしたときに，このライブラリを使うリンクの手順は以下のようになります．

```
A>l80 ck,prog,alib,clib/s,crun/s,cend,prog/n/e:xmain↵
```

この場合，xlibの後に/s（必要な関数だけをリンクするためのスイッチ）は不要です．それを指定しても結局はすべての関数がリンクされてしまうからです．

アセンブラで書かれたプログラムの場合も，同じようにしてライブラリを作成したり，使用し

たりすることができます．次のような3つの関数hex（），ror（），rol（）について考えてみましょう（ファイル名はalib.macとします）．

```
        public hex@,ror@,rol@

        cseg

hex@:           ;hex(c) = convert 0 - F to ASCII '0' - 'F'
        and     0FH
        add     a,90H
        daa
        adc     a,40H
        daa
        ret

ror@:           ;ror(c,n) = rotate c right n bits
        inc     e
ror1:
        dec     e
        ret     z
        rrca
        jr      ror1

rol@:           ;rol(c,n) = rotate c left n bits
        inc     e
rol1:
        dec     e
        ret     z
        rlca
        jr      rol1
```

これらの関数をライブラリ化する手順は以下のようになります．

```
A>m80 =alib↵
```

リンクする手順は，Cの場合と同様に次のようになります．

```
A>l80 ck,prog,xlib,clib/s,crun/s,cend,prog/n/e:xmain↵
```

以上のように，「必要なものだけをリンクする」という機能をあきらめてしまえば，ライブラリ作成はいたって簡単なものとなります．しかし，やはりこれでは完全なライブラリとは言えません．以下では，必要な関数だけをリンクするようなライブラリの作り方について述べます．

B)必要な関数だけをリンクするライブラリ

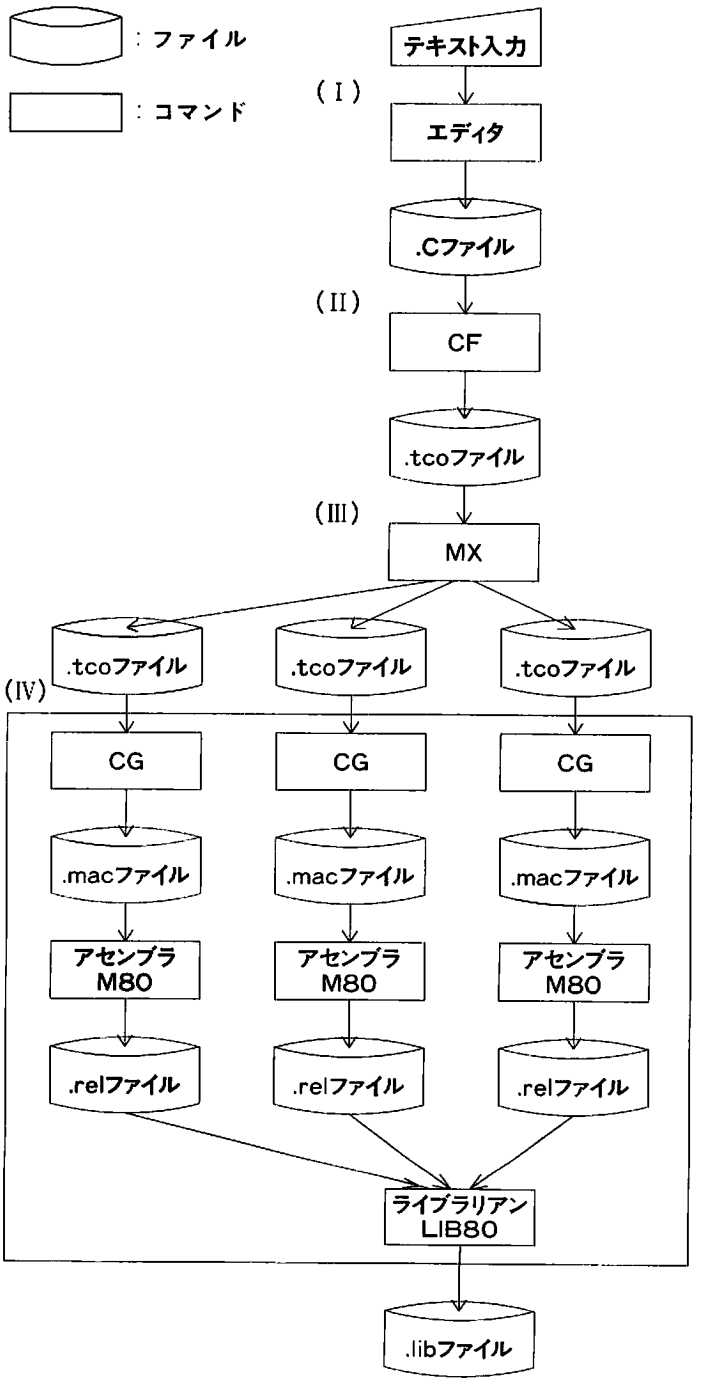


図　専用ライブラリ作成の手順

さて，ライブラリの作成法について説明するわけですが，その前に作業に当って必要となる環境を定義することにしましょう．(「4.5.4　標準ライブラリの再作成」で使う forremk.bat を使用して環境だけ整えてもいいでしょう．)

まずディスク装置は2台あるものとします(A：およびB：)．Bドライブは空にしておいて下さい．Aドライブには，以下のファイルが必要です．

ユーティリティ　ソフトウェア
- m80.com
- l80.com
- lib80.com

MSX-C コンパイラ　パッケージ
- cf.com
- cg.com
- fpc.com
- mx.com
- arel.bat
- crel.bat

このほかにスクリーンエディタがあると便利です．

先ほどと同じく，power ( ) と log2 ( ) の2つの関数を持つファイル xlib.c をライブラリにしてみましょう．

まず，xlib.c をエディタによってAドライブに作成して下さい．(97ページ図の(I)) そして以下のコマンドを入力します．

```
A>cf xlib↵                          97ページ図の(II)
...
A>mx -lob xlib > temp.bat↵          97ページ図の(III)
...
A>temp↵                             97ページ図の(IV)
```

CF は xlib.c から xlib.tco を作成します．次に，MX が xlib.tco から power.tco と log2.tco の2つのモジュールを取り出し，Bドライブにそれらを置きます．

また，このとき MX は，

(1)これらのファイルに対して，コード生成を行なって，.mac を作る．
(2)アセンブルを行なって，.rel を作る．
(3) LIB80 を起動して2つの .rel を連結する．

という手順を標準出力に出力するので，それをそのまま temp.bat というファイルにリダイレク

トし，すぐに temp を実行しています．

これらの処理が完了すると，Bドライブに xlib.lib というファイルができています．その他に，power.rel と log2.rel というファイルがありますので，これを

```
A>del b:*.rel
```

で削除して下さい．その後に

```
A>ren b:xlib.lib xlib.rel
```

とすれば，目的のライブラリが完成します．

この xlib.rel の使い方は以下のようになります．

```
A>l80 ck,prog,xlib/s,clib/s,crun/s,cend,prog/n/e:xmain
```

xlib の後の/s は必要です．これを付けることにより，リンカは必要な関数だけをリンクしてくれます．

リンカ L80 を使用する場合は，関数を定義する順序が問題となるので注意してください．たとえば，

（正しい例）

```
char    islower(c)
char    c;
{
        return ('a' <= c && c <= 'z');
}

char    toupper(c)
char    c;
{
        return (islower(c) ? c - 'a' - 'A' : c);
}
```

という2つの関数からなるファイル x.c をライブラリにしたとしましょう．このとき，もし x.rel の中で，islower ( ) が toupper ( ) よりも先に置いてあると，L80 で/s オプションを付けてリンクしたときに，未定義エラーとなります．

実際には，MX はソースプログラム中の順番とは逆に .rel を連結しますので，islower ( ) を先に書いておけば正しいライブラリが作られますが，下のように toupper ( ) を先に書くと，ライブラ

リ中のモジュールは誤った順序で配置されてしまいます．したがって，ライブラリのソースプログラムでは，呼び出される関数を先に，呼び出す関数を後に置くようにしてください．

(誤った例)

```
char    toupper(c)
char    c;
{
        char    islower();
        return (islower(c) ? c - 'a' - 'A' : c);
}

char    islower(c)
char    c;
{
        return ('a' <= c && c <= 'z');
}
```

次に，アセンブラで書かれたプログラムをライブラリにしてみましょう．前述の hex ( )，ror ( )，rol ( )のプログラムを使用しますが，プログラムは以下のように変える必要があります．これを alib.mac という名前で A ドライブ上に作成して下さい．

```
        MODULE  HEX
        public  hex@
hex@:
        and     0FH
        add     a,90H
        daa
        adc     a,40H
        daa
        ret

        ENDMODULE

        MODULE  ROR
        public  ror@
ror@:
        inc     e
ror1:
        dec     e
        ret     z
        rrca
        jr      ror1

        ENDMODULE

        MODULE  ROL
        public  rol@
```

```
rol0:
        inc     e
rol1:
        dec     e
        ret     z
        rlca
        jr      rol1

        ENDMODULE
```

これをライブラリにするには，次のようにコマンドを入力するだけです．

```
A>mx -lob alib > temp.bat
A>temp
```

すると，MX は alib.mac から hex.mac，ror.mac，rol.mac を引き出し，作成されたバッチファイル temp がそれらをアセンブルして .rel を作り，LIB80 を呼び出して連結します．

すべてが完了すると B ドライブには alib.lib ができていますので，作成された不要な他の .rel ファイルを消してから alib.rel に名前を変更すれば，ライブラリの完成です．

C の場合と同様，L80 リンカを使う場合には，モジュールの順番が問題になります．ソースプログラムでは，必ず呼び出される方のモジュールを先に，呼び出す方のモジュールを後に書いておいて下さい．

ライブラリ自身の作成方法は上記のとおりですが，もうひとつやるべきことが残っています．それは FPC に与えるパラメータチェック用のファイルを作ることです．

これは以下の手順で作成します．

(1)ライブラリ用の C のソースプログラムに対して CF を実行して .tco ファイルを作る．
(2)アセンブラで記述したソースプログラムがある場合，それらの関数の型およびパラメータの型を C 言語で表現したファイルを作り，CF によって .tco ファイルにする．
(3) FPC の"-c"スイッチを用いて，(1)，(2)で作った .tco ファイルを圧縮して連結する．

例として，前述の xlib.c と alib.mac チェック用ファイルを作ってみましょう．まず，

```
A>cf xlib
```

によって xlib.tco を作ります．

次に，alib.mac のパラメータを，C のプログラムで表現します．alib.c として，以下のようなものを作って下さい．

```
char    hex(c)
char    c;
{}

char    ror(c, n)
char    c;
int     n;
{}

char    rol(c, n)
char    c;
int     n;
{}
```

このファイルから，次のコマンドで alib.tco を作成します．

```
A>cf alib
```

最後に，

```
A>fpc -c axlib xlib alib
```

とすれば，チェック用のファイル axlib.tco が完成します．

さて，これまでの説明ではライブラリを新たに作成する方法が述べられましたが，ライブラリ中の 1 つの関数を修正したいような場合に，ライブラリ全体を作り直すのは効率的ではありません．

以下では，既存のライブラリ中の関数を差し替える方法について説明します．

ここでも xlib.c を例にとって説明します．power ( ), log2 ( ) という 2 つの関数からなるこのファイルのうち，関数 power のみ変更したとします．このとき，ライブラリファイル xlib.rel 中の power ( ) を新しいものに交換するための手順は，次のようになります．

```
A>cf xlib
...
A>mx -ob xlib power > temp.bat
...
A>temp
...
A>lib80
*b:lib=b:xlib<..power-1>,b:power,b:xlib<power+1..>
*/e
```

これまでの例では，MX に対して-l オプションが指定されていましたが，ここでは指定していません．-l を指定しないと，MX は LIB80 に対する手順を標準出力に出力しません．また，xlib

の後に power を指定しているため，MX は関数 power ( )だけを抽出します．

また，LIB80 に対するコマンド（最後の 2 行）は，「古い xlib.rel から power ( )以外のモジュールを抽出して，それらを新しい power.rel と連結せよ」という指定です．このとき，モジュールの順序が変ってしまわないように注意してください．

4.5.4 標準ライブラリの再作成

以下に示す方法で作られるライブラリは clib.rel, crun.rel としてシステムファイルに含まれています．通常の使用では，これらのライブラリを新たに作り直す必要はありません．

標準ライブラリの再作成には 1 枚の未使用ディスクが必要で，1 台のドライブで実行可能です．またライブラリの作成は約 2 時間かかりますので，変更を行う場合は変更部分を良くテストしてから行うことをお推めします．(1DD のドライブ 1 台では，容量不足のため実行できません．ドライブ 2 台で行うか，2DD のドライブを用意して下さい．)

再作成用のディスクを作るには，MSX-C コンパイラに付属するバッチファイル forremk.bat（ディレクトリ¥batch 内にあります．）を使用します．FORREMK の使用方法を説明しましょう．

(1)新しいディスクをフォーマットします．

(2)フォーマットされたディスクに forremk.bat をコピーします．B ドライブに MSX-C のマスターディスクがあるとすると次のようになります．

```
A>copy b:¥batch¥forremk.bat
```

(3)コピーしたディスクを A ドライブにいれ，次のようにコマンドを入力します．

```
A>forremk b
```

(4)あとはプロンプトに合わせて，MSX-DOS2 のシステムディスクなどを B ドライブに入れていきます．

以上で再作成用のディスクができます．2 ドライブシミュレータを使っている場合には，"Insert disk for drive <d：>"というメッセージが出てからディスクを交換して下さい．

再作成用のディスクができましたが，これからコンパイルするわけですから，環境(環境変数)を整えるために，必要であれば CENV を実行してください．(これについては「1.3　MSX-C の開発環境（開発環境の構築)」も参照して下さい．)

出来上がったディスクには次のようなファイルが入っているはずです．(A ドライブにディスクがあるとします．)

a：¥　　(ルートディレクトリの内容)

forremk.bat	このディスクを作成するためのバッチファイル
msxdos2.sys	MSX-DOS2 システムファイル
command2.com	〃
clibc.c	clibmac.mac のスケルトン作成用ファイル
direct.c	ディレクトリ操作関数ソースファイル
io.c	低水準入出力，キーボード関数ソースファイル
malloc.c	メモリ管理関数ソースファイル
process.c	プログラム管理関数ソースファイル
stdio.c	高水準入出力関数ソースファイル
stdlib.c	汎用関数ソースファイル
string.c	文字列操作関数ソースファイル
ck.mac	カーネル呼び出しルーチンソースファイル
clibmac.mac	標準ライブラリ関数アセンブラ記述部
crun.mac	実行時ルーチンソースファイル
cend.mac	標準ライブラリ関数アセンブラ記述部 2
cenv.bat	MSX-C の環境を整えるバッチファイル
genlib.bat	標準ライブラリ再作成開始用バッチファイル
genliba.bat	アセンブリ言語用再作成バッチファイル
genlibc.bat	C 言語用再作成バッチファイル
genrel.bat	ライブラリ .rel ファイル再作成用バッチファイル
gentco.bat	ライブラリ .tco ファイル再作成用バッチファイル
arel.bat	MX 用スケルトンファイル(アセンブリ言語用)
crel.bat	MX 用スケルトンファイル(C 言語用)

a：¥bin　　(コマンドのディレクトリ¥bin の内容)

m80.com	アセンブラ
lib80.com	ライブラリアン（ライブラリ管理ユーティリティ）
cf.com	コンパイラ（パーサ）
cg.com	コンパイラ（コードジェネレータ）
fpc.com	ファンクションパラメータチェッカ
mx.com	モジュール抽出ユーティリティ

a：¥include　　(ヘッダファイルのディレクトリの内容)

bdosfunc.h	MSX-DOS ファンクションコール用ヘッダ
conio.h	キーボード・I／O 関数用ヘッダ
ctype.h	文字種判断関数用ヘッダ

direct.h	ディレクトリ操作関数用ヘッダ
io.h	低水準入出力関数用ヘッダ
malloc.h	メモリ管理関数用ヘッダ
memory.h	メモリ操作関数用ヘッダ
process.h	プログラム操作関数用ヘッダ
setjmp.h	setjmp 用ヘッダ
stdio.h	高水準入出力関数用ヘッダ
stdlib.h	汎用関数用ヘッダ
string.h	文字列操作関数用ヘッダ
type.h	MSX-C 標準型の定義用ヘッダ

環境変数の設定

INCLUDE	a：¥include
PATH	a：¥； a：¥bin

ライブラリの再作成用のディスクができましたから，再作成起動用のバッチファイル genlib.bat の使用方法を説明しましょう．できたディスクは A ドライブにあるとします．GENLIB 引数には，標準ライブラリの再作成されるドライブの，ドライブ名 (B ドライブなら"b"の 1 文字) だけを指定します．A ドライブに作成するのであれば

```
A>genlib a
```

と入力します．ここではカレントドライブと同じ A ドライブに作成します．次のように入力して下さい．

```
A>genlib a
```

あとはバッチファイルが自動的にライブラリを作成していきます．全体は 11 個の部分からなり，それぞれの部分が始まるごとに，作成しようとしているファイルの名前を表示します．

```
Make process.lib from process.c
Make direct.lib from direct.c
Make stdlib.lib from stdlib.c
Make stdio.lib from stdio.c
Make string.lib from string.c
Make io.lib from io.c
Make malloc.lib from malloc.c
Make clibmac.lib from clibmac.mac
Make crun.lib from crun.mac
Make .rel file for linker L80
Make .tco file for parameter checker FPC
```

また，これ以外にもコンパイル中の表示もされます．正常に動作して終了した時には，次のように表示されます．

```
:
:
MSX-C function parameter checker ver 1.20x
complete

A>
```

なお，ライブラリを拡張したときにディスクスペースが不足した場合には，2台のドライブを使用して下さい．Aドライブに上記のディスクを挿入し，Bドライブには空のディスクを入れて，次のコマンドを入力します．

```
A>genlib b
```

.relファイルおよび最終的なライブラリはBドライブに作られます．

第５章　MSX-C コンパイラの応用

5.1 Disk-BASIC 環境でのマシン語ルーチンの作成

ここでは，Disk-BASIC でサブルーチンとして用いるためのマシン語プログラムの作成方法を述べます．

5.1.1 Disk-BASIC 環境と USR 関数

A)Disk-BASIC のメモリ・マップ

まず，図 6-1 を見て下さい．これは Disk-BASIC 起動時のメモリの状態を示したものです．これを見るとわかるように，Disk-BASIC のユーザーエリアは，BASIC インタープリタの領域の終了アドレスの次のアドレスからディスクのワーク・エリアの開始アドレスの一つ前のアドレスまでです．

ユーザー・エリアの開始アドレスは MSX2 の場合，8000H 番地ですが，ディスク・ワーク・エリアの開始アドレスは実装されているディスク・ドライブの数や種類により異なります．

現時点では，2DD（両面）フォーマットのディスク・ドライブが 2 機装備されている場合や，1 機でも 2 ドライブ・シミュレータが機能している場合に，最も大きなディスク・ワーク・エリアが取られます．その場合のディスク・ワーク・エリアの下限は，Disk-BASIC バージョン 1.00 では DE70H 番地前後，バージョン 2.00 では E580H 番地前後ですが，Disk-BASIC 起動直後にワーク・エリア：HIMEM（FC4AH と FC4BH 番地）の内容を確認することで正確な番地を知ることが出来ますので，安全の為に確かめておくのがよいでしょう．

B)マシン語領域の確保と USR 関数

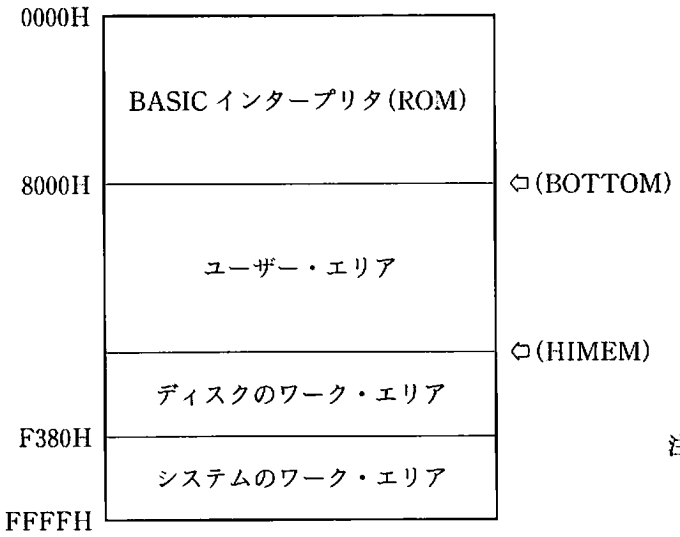
Disk-BASIC 環境下でのマシン語ファイルの読み込みから，USR 関数を用いて実行するまでの手順を以下に示します．

(1) CLEAR 文でマシン語プログラムをロードするエリアを確保する．

ユーザー・エリアの上限アドレスが，CLEAR 文の第 2 パラメータで与えられたアドレスに設定されます．例えば

```
100 CLEAR ,&HCFFF
```

とすれば変数やスタック領域も含めてBASICのプログラムが使える領域の上限がCFFFH番地(新しいHIMEM－1)までとなり，D000H番地から元のユーザー・エリアの上限までがマシン語のプログラムをロードするために確保されたわけです．この領域の広さは読み込むマシン語ファイルの大きさにより設定します．



注) RAM容量が32K以上であってもBASICを使用する場合，32Kバイトしか使用されない．

図6-1　Disk-BASICモードでのメモリ状態

(2) BLOAD文で，上で確保したメモリ領域にマシン語ファイルを読み込む．

BLOAD命令で読み込むことのできるマシン語ファイルはそのための特殊な形式を持ったものでなくてはなりませが，これについての説明は5.3で行います．

(3) DEF　USR文で，マシン語プログラムの実行開始アドレスを指定する．

例えば，D000H番地からマシン語を読み込んだ場合

```
100 CLEAR ,&HCFFF
200 BLOAD "MACHIN.BIN"
300 DEF USR=&HD000
```

のようにUSR関数を設定します．これでそれ以降のBASICプログラム中で

```
500 A=USR("c")   (マシン語ルーチンに"c"という1文字を渡しその結果をAという変数に代入する)
```

のように用いる事が出来ます．

以上が，Disk-BASICでマシン語を用いるための最も一般的な方法ですが，これを実現するためには，(2)で述べたように，BLOAD命令で読み込むことが可能なマシン語ファイルを作らなければなりません．BLOAD形式のファイルはプログラム部分の前に，どの番地からロードするか，などの情報が入ったヘッダが付きます．

以下，「5.2　C言語によるソース・プログラムの作成」ではC言語によるDisk-BASIC環境のためのプログラム作成の実際を示し，「5.3　C言語ソース・ファイルからBLOADファイルまで」ではそのC言語のソースからBLOAD可能なマシン語ファイルの生成までのコンパイル，リンク等の手順を解説します．

5.2　C言語によるソース・プログラムの作成

5.2.1　USR関数の引数の受取り方

USR関数呼び出しの際に与えられた引数はマシン語側ではAレジスタでその型を知ることができます．下表にAレジスタの値と引数の型を示します．

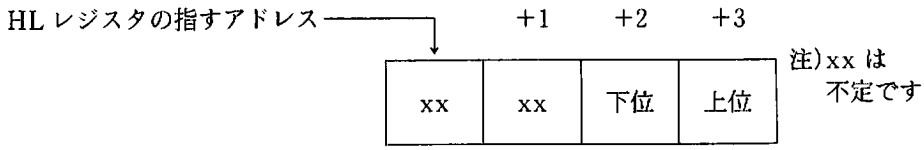
そして，実際の引数の値はその実体が格納されたアドレスが渡されます．アドレスが与えられるレジスタは引数の型により異なります．渡される引数が2バイト整数型，単精度実数型，及び倍精度実数型の場合はHLレジスタに，文字列型の時はDEレジスタにアドレスが格納されます．(下表参照)

表　Aレジスタに代入される引数の型

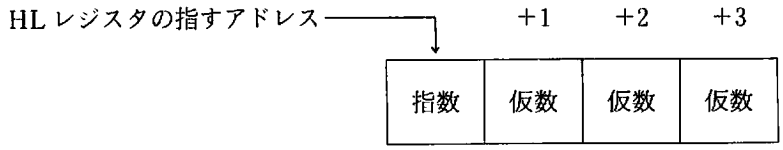
2	2バイト整数型
3	文字列型
4	単精度実数型
8	倍精度実数型

引数による値の渡され方

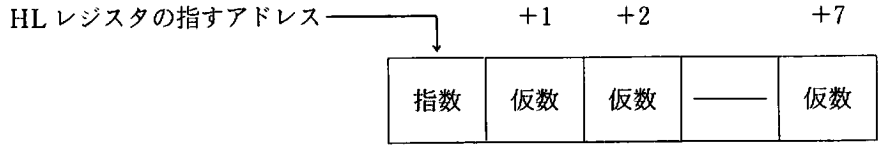
2バイト整数型



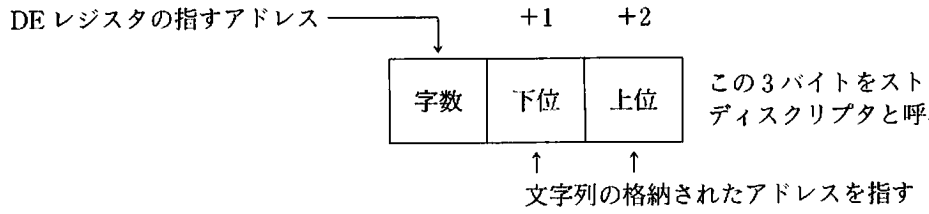
単精度実数型



倍精度実数型



文字列型



それでは，このようにレジスタに渡された値をCの関数側でどのようにしたら見ることが出来るでしょうか．ここで，「3.2.1　パラメータの渡し方」にある表をご覧下さい．これを見るとわかるように固定パラメータ関数ではAレジスタの値はchar型の第1パラメータとして，DEレジスタの値はその他の型（2バイトの値）の第2パラメータとして受け取る事が出来ます．HLレジスタの値は第1パラメータをchar型以外に定義すれば知ることが出来ますが，そうすると渡された引数の型（Aレジスタに入っている）を知ることができなくなるので，結局C言語だけでUSR関数を書く場合，DEレジスタにより，文字列型の引数だけが扱えることになります．その他の型を扱いたい場合には，アセンブラでHLレジスタの値が得られるようなインターフェースルーチンを書く必要があります．

以下にC言語によるUSR関数で文字列引数を受け取る場合のメインルーチンの記述例を示します．

例

```
typedef struct{
        TINY    length;
        char    *strptr;
} DESCRIPT;

main(type, desc)
TINY    type;                   /* typeは，Aレジスタの値 */
DESCRIPT *desc;                 /* descは，DEレジスタの値 */
{
        if(type != 3){          /* 引数の型は必ず文字列型でなくて */
                outstr(ermes);/* はいけないので，typeが3以外の */
                return;         /* 場合エラーメッセージで終了する */
        }
                •
                •
                •
}
```

5.2.2　システム・コールの使い方

MSX-DOS上でのC言語プログラムでファンクションコールを使いたい場合には，ライブラリ関数のbdos ( )，bdosh ( )などを使うことが出来ました．

Disk-BASICの環境下でも，ファンクションコール（BDOS機能）が可能ですが，呼び先のアドレスがMSX-DOS環境下での場合と異なるため上記の2つの関数は使えません．このため，BDOSコールに必要なレジスタをセットしてDisk-BASICでのBDOSコールの呼び先アドレスを直接コールするようなプログラムを書かなくてはなりません．

以下に，C言語のソースをコンパイルした結果，どのようなアセンブラ・ソースが生成されるか，という観点から解説していきます．

ファンクションコールを実行するためには，CPUのCレジスタに用いたい機能のファンクション番号を入れ，そのファンクションで必要な引数を決められたレジスタに入れた後，F37DH番地（DOSではこれが0005H番地でした）をコールします．

例えば，ファンクション番号02Hのコンソール1文字出力ルーチンを用いて，"a"というで文字を画面に表示したい場合，これをアセンブラで記述すると

```
ld      e, 'a'
ld      c, 02h
call    0f370h
        •
        •
        •
```

の様になります．つまりコンパイルした結果，上のようなアセンブラ・ソースが生成されるようにプログラムを書かなければなりません．つまり，F37DH 番地を関数へのポインタとし，レジスタにセットする値をその関数の引数として，関数呼び出しをおこなうということです．

実際の記述例を示します．

例

```
#include <stdio.h>
#pragma nonrec

puta()          /*ファンクションコールで'a'という1文字を画面出力*/
{
        (*(VOID (*)())(0xf37d))((TINY)0,'a',(TINY)0x02);
}
```

1番目の引数に0を渡していますが，これはファンクション02Hでは第1引数，すなわちAレジスタ（dummyがchar型以外ならHLレジスタ）には値をセットする必要がないのですが，以降の引数がそれぞれEレジスタ，Cレジスタにセットされるようにするための，コンパイラに対する見せかけの引数です．上のソース・ファイルをコンパイルした結果は以下のようになります．

```
;       MSX-C ver 1.20x   (code generator)

        cseg

puta@:
        ld      c,2
        ld      e,97            ;ASCII 'a'
        xor     a
        call    62333           ;f37dh in hex
        ret


        public  puta@

        end
```

Aレジスタを0にする命令（xor　a）が余分である以外は目的どおりのアセンブラ・ソースを生成することができました．実際のプログラムでは，ファンクションコールを呼び出すたびに上のように書くのではなく#define文によってマクロ定義しておくのが良いでしょう．

なお，ファンクションコールの詳細は「MSX-DOS　TOOLS」に付属のマニュアル，「MSX2テクニカル・ハンドブック」，「MSX-DOS2リファレンスマニュアル」などをご覧下さい．

5.2.3　値の返し方

USR関数の引数として渡された文字列の内容を書き換えることによって，USR関数の値をBASICに引き渡すことができます．但し，文字数を変えることはできません．6.2.1の例での値の文字列の返し方の例を下に示します．

例

```
retbas(desc,newstr)
DESCRIP *desc;
char    *newstr;
{
        if (strlen(str) != (int)desc->length){
                outstr(erames4);
                return;
        }

        desc->strptr = newstr;          /*新しい文字列をセットして*/
        return;                         /*戻り値なしのリターン    */
}
```

上記の例のように渡された文字列へのポインタを，返したい文字列へのポインタに変えてしまってもよいわけです．

値を受け取る側のBASICの記述例を下に示します．

例

```
1000 B$=USR(A$)
```

5.3 C言語ソース・ファイルからBLOADファイルまで

5.3.1　L80だけによるBLOADファイルの作成

C言語のソース・ファイルが書けたら，コンパイル，リンクによりBLOAD形式のオブジェクトファイルを作ります．CF，FPC，CG及びMSX・M-80を使ってリロケータブル・ファイル（.rel）を作るところまではDOSのコマンド・ファイルの作成と同様ですが，リンクのしかたが異なります．

ここでMSX-DOSのコマンド・ファイル（.com）が100H番地からロード実行されることを思いだして下さい（詳細はMSX-DOSのマニュアル等参照のこと）．MSX・L-80でリンクする場合ロード開始アドレスを特に指定しないと，100H番地からのオブジェクトが作られるのです．です

からDOS上のコマンド・ファイルを作る場合にはロード開始アドレスといったことを特に意識しないでも良かったわけです．ところが，Disk-BASICでのルーチンの場合には，5.1 Disk-BASIC環境でのマシン語ルーチンの作成 A)およびB)で述べたように，ユーザー・エリアのどこにマシン語を置くかはユーザー自身が決めなくてはなりません．そのエリアの開始アドレスからロード開始されるようなオブジェクトを生成するためには，リンク時にL80の/pオプションを用います．D000H番地からの場合以下のようになるでしょう．

```
A>l80 /p:d000,usrfunc,usrfunc/n/e:start
```

但し，こうして出来上がったファイルは相変わらずDOSのコマンド形式のファイル（.com）です．すなわち103H番地からCFFFH番地までに相当するところは不要なデータ（ゴミ？）でうまっているのです．(100-102Hには"start："へのジャンプ命令が入っている)したがってDisk-BASICでこのファイルをD000H番地からロードするのにBLOAD命令は用いることができません．ゴミデータの部分を1バイトずつ読みとばし，必要なところだけをD000H番地からのエリアに格納するようなプログラムを書かなければなりません．しかし，一旦このように読み込むことができれば，Disk-BASICのBSAVE命令によりD000H番地からをセーブしておけば次回からはBLOAD命令でロードできるファイル（BLOADファイル）としてディスクに記録されます．

5.3.2 MSX-DOS2 TOOLSのBSAVEコマンドを使う

MSX-DOS2 TOOLSに付属のBSAVEコマンドを用いることで，5.3.1で述べたような作業なしに，.relファイルからBLOAD形式のファイルを簡単に作ることができます．

以下にその手順を述べます．

まず，リンク時に/xオプションで.hexファイルをつくります．

```
A>l80 /p:d000,usrfunc,usrfunc/n/x/e:start
```

これでD000H番地から実行可能なusrfunc.hexというファイルが作られます．このファイルをMSX-DOS2 TOOLSのBSAVE（Disk-BASICのBSAVE命令にではない）コマンドに入力することによりDisk-BASICでBLOAD可能なファイルに簡単に変換することができます．

```
A>bsave usrfunc.hex >usrfunc.bin
```

とすることにより生成されるファイルusrfunc.binはDisk-BASICのBLOAD命令で読み込むことができる形式をもつファイルです．

5.4　サンプル・プログラムについて

本パッケージのディスクにはDisk-BASIC下のマシン語ルーチンのサンプルとして，search.c, search.bas, search.bat, bk.macの4つのファイルが含まれています．

search.cは，システムコールを用いてディスク上のファイルを検索するC言語のルーチンであり search.basのサブルーチンとして用いられます．

search.batは，search.cをコンパイル，アセンブルした後，bk.macをアセンブルしたbk.relとともにリンクし search.hexを生成します．

次に，MSX-DOS2 TOOLSの BSAVEコマンドを実行し BLOAD形式のファイルに変換します．さらにDisk-BASICを起動し，search.basをロード実行まで行います．このバッチ処理を起動するだけでDisk-BASIC用のマシン語サブルーチンの作成過程を追うことができます．

注意 search.batを最後まで実行するにはTOOLSの BSAVEコマンドがカレントドライブのカレントディレクトリに必要です．

以上で，Disk-BASIC環境でのマシン語ルーチンの作成手順を大まかに述べてきました．引数の扱い方やBLOAD形式のファイル作成の部分で説明を省いた点もありますが，不明な点は「MSX2テクニカルハンドブック」，「MSX-DOS2 TOOLS」マニュアル等や，MSX-BASICに関する文献をあたって下さい．

5.5　ROM化プログラムの作成

ここでは，ROMにプログラムを書き込むための手順と方法について説明します．

5.5.1　ROM化プログラム作成の一般的原則

A)ライブラリの引用について

ROM化プログラムを作成するにあたって，最初に問題となるのは，標準カーネル（CK）および標準ライブラリ(CLIB)です．これらのプログラムの中には，MSX-DOSの機能を使っているものがあるので，ROM化プログラムの開発では使うことはできません．ただし，CLIBの中にはstrcpy ( )やtoupper ( )などMSX-DOSに依存しない関数も含まれています．これらについては，ROM化プログラムでも使用可能なのですが，安易にCLIBをリンクしてしまうと，BDOS機能を使っている関数を誤って呼んでしまう恐れがあります．

したがって，これらの関数（strcpy ( )や toupper ( )）を使用したいときは，

(1)作成しているプログラム中に関数をソースプログラムのままで取り込む

あるいは，

(2) clib.rel の中から LIB80（ライブラリマネジャ）を用いて ROM 化可能な関数だけを抜き出し，別にライブラリを作る

の，どちらかの方法をとって下さい．

また，ヘッダファイルもそのままでは使えません．必要な部分だけを抜き出して，別のヘッダファイルを作って下さい．

B)初期設定の部分

次に，初期化の処理を ROM 化プログラムで行う必要があります．MSX-DOS 下で動くプログラムの場合は，初期設定は clib.rel 中のカーネル本体で行なっているわけですが，前述のようにカーネルは ROM 化プログラムでは使用不可能ですので，これに代わるものが必要となります．

通常，初期設定は次のようなアセンブラ・プログラムで行ないます．

```
extrn   main@
ld      sp, スタックのアドレス
     :
     :
（必要ならばハードウェアの初期設定を行う）
     :
     :
jp      main@
end
```

すなわち，スタックポインタを設定し，必要なハードウエアの初期設定を行なったのちに C の関数 main ( )へジャンプするわけです(このとき，C の入口の名前が main でなければいけないという制限はなく，たとえば foo でも zot でもかまわない)．この場合には，main ( )に渡すパラメータはありませんので，通常の MSX-DOS における main ( )プログラムとは異なり，

```
VOID    main()
{
        :
        :
}
```

と書かなくてはなりません.

さて，上の初期設定プログラムの使い方ですが，まずこのプログラムを作り，アセンブルして，.rel ファイルにしておき，リンクするときに先頭につけて下さい．5.5.2 で実際の例を示します.

C)IN／OUT

ROM 化するときに最も悩むのが IN／OUT です．アセンブラでは IN，OUT 命令を使用すればよいのですが，C 言語ではそのような命令は用意されていませんので，関数として呼び出すことになります.

MSX-C の標準ライブラリの中には，inp ( ) と outp ( ) という関数が含まれており，これらを用いるのが最も簡単な方法でしょう.

第 2 のやり方は，ひとつのポートから IN／OUT を行なう関数を各ポートごとに用意するというものです．たとえば，

```
          public indat0, inctl0, outdat0, outctl0
indat0:
        in      a,(00h)
        ret
inctl0:
        in      a,(01h)
        ret
outdat0:
        out     (00h), a
        ret
outctl0:
        out     (01h), a
        ret
```

と 4 つの関数をアセンブラで作っておき，C のプログラムで

```
char    indat(),  inctl();
VOID    outdat(), outctl();

VOID    init()
{
       outctl((char)0x01);
       outctl((char)0x01);
       outctl((char)0x40);
       outctl((char)0xce);
       outctl((char)0x37);
}

char    getch()
{
        while ((inctl() & 0x02) == 0)
                ;
```

```
        return (indat());
}

VOID    putch(c)
char    c;
{
        while ((inctl() & 0x01) == 0)
                ;
        outdat(c);
}
```

というように呼び出して使います．(このプログラムは，8251 の初期設定(init ( ))，入力(getch ( ))，出力 (putch ( )) を行なう) このやり方ならば，アセンブラで IN／OUT を行なった場合にくらべて CALL／RET が余分に実行されるだけですから，inp ( ) や outp ( ) ほど遅くはありません．

ロ)絶対番地の参照

入出力ポートが I/O 空間でなく，メモリ空間に割り当てられている場合があります．この場合の入出力操作は，I/O のときよりもぐっとスマートに記述できます．

たとえば，先ほどのプログラムでポート 00H の代わりに 0FF00H，01H の代わりに 0FF01H を使って操作を行なうとすると，次のように書けます．

```
#define  dataport  (*(char *)0xff00)
#define  ctrlport  (*(char *)0xff01)

char    getch()
{
        while ((ctrlport & 0x02) == 0)
                ;
        return (dataport);
}

VOID    putch(c)
char    c;
{
        while ((ctrlport & 0x01) == 0)
                ;
        dataport = c;
}

VOID    init()
{
        ctrlport = 0x01;
        ctrlport = 0x01;
        ctrlport = 0x40;
        ctrlport = 0xce;
        ctrlport = 0x37;
}
```

E)割り込み処理

MSX-C では，割り込み処理も行なうことができます．ただし，いくつかの言語プロセッサで導入されているような，「割り込み手続き宣言」は用意されていません．したがってコンパイラは，割り込みの入口での全レジスタの待避，出口でのレジスタの回復，そして割り込みマスクの解除といったシーケンスは生成しませんので，こういった処理はアセンブラで書いてやらなくてはならないのです．

例として，1秒ごとにタイマ割り込みが入って，時刻データを更新するというプログラムを考えてみましょう．時刻は，それぞれ TINY 型の hour，minute，second で表すことにし，割り込みは 0038H に入ってくるものとします．

アセンブラ：

```
        extrn   crst70
        aseg
        org     0038h
        jp      rst7

rst7:
        push    af
        push    bc
        push    de
        push    hl
        call    crst70
        pop     hl
        pop     de
        pop     bc
        pop     af
        ei
        ret
```

C：

```
TINY    hour, minute, second;

VOID    crst7()
{
        if (++second >= 60) {
                second = 0;
                if (++minute >= 60) {
                        minute = 0;
                        if(++hour >= 24)
                                hour = 0;
                }
        }
}
```

Cのプログラム中で,割り込みを禁止したり，解除したりという処理が必要な場合は，アセンブラで,

```
        public  di@, ei@
di@:
        di              ; 割り込み禁止
        ret
ei@:
        ei              ; 割り込み許可
        ret
```

という関数を作って呼んで下さい.

割り込みについて注意しなければならない点があります．それは，割り込み処理と通常の処理の両方から呼ばれる関数がある場合，その関数はrecursive（再帰呼び出し可能）にしておく必要があるということです．このような場合には，実際に再帰呼び出しはしていなくとも，recursiveにして下さい.

F)定数テーブルの扱い

イニシャルライズされている変数のうち，記憶クラスがstaticとexternのものはコード・セグメント（CSEG）に配置されます．MSX-DOS環境下のようにRAM上でプログラムを動かす場合には，これらの変数の値を変更することができますが，そのプログラムをROMに書き込んだ場合には，これらの変数はコード・セグメントに置かれるので値を書き換えることができなくなり，あたかも「定数」であるかのような振舞いをします．したがって，変数の記憶クラスやプログラム中での実際の使用については十分に注意して下さい.

ROM化プログラムにおける初期化プログラムの役割は「定数テーブルを実現する」ことであり，変数を初期化することではありません．変数を初期化する場合には，代入文で行なう必要があります．また，文字列は「記憶クラスstaticである初期化された文字の配列」でありコード・セグメントに配置されるために，ROMに書き込んだ場合にはその値を変えることはできません.

5.5.2 ROM化プログラムの作成例

MSX-Cコンパイラ・パッケージには，rom0.macおよびrom1.cというファイルが入っています．これは，ROM化用のサンプルプログラムであり，MSXで実際にROMに焼いて動作させることができます.

まずrom0.macを以下のようにアセンブルし，.relファイルを作成して下さい.

```
A>m80 =rom0↵
```

次に，rom1.c をコンパイルして.rel ファイルを作成して下さい．以下のシーケンスでできます．

```
cf rom1
fpc -u rom1
cg rom1
m80 =rom1        (rom1.macのアセンブル)
del rom1.mac
```

最後にリンクします．これは次のコマンドでできます．

```
l80 /p:0,/d:RAMの先頭アドレス,rom0,rom1,crun/s,rom/n/e
Origin below loader memory, move anyway (YorN)?N  (必ずNと答える)
```

以上のリンクが終ると，rom.com ができています．これを ROM に書き込んで下さい．

このプログラムを実行すると，">"というプロンプトが表示されます．ここで

```
>d0,FF
```

と入力すると，0 番地から 00FFH 番地までの内容がダンプされます．また，

```
>sxxxx
```

と入力するとメモリの内容が表示され，新しい値を書き込むことができます．

このプログラムはＤとＳコマンドしかないデバッガです．プログラミングの練習として，このプログラムを拡張して，より高機能なデバッガを作成してみてください．

第6章 標準ライブラリ関数リファレンス

この章ではMSX-C Ver.1.2の標準ライブラリ関数をアルファベット順に解説していきます．関数のリファレンスマニュアルとしてご使用下さい．

ひとつの関数は次のような7項目から成り立ちます．これらの項目は必要がない場合には省略されることがあります．

(1) **関数名**

解説する関数の関数名だけを示します．関数getchar()の場合は，"getchar"を示します．また，同ページ内で複数の関数を解説するときには，それらを続けて書きます．

(2) **書　式** (関数のヘッダ)

関数を使用する際の書式として関数のヘッダを示します．また，関数に必要なヘッダファイルも示します．このヘッダファイルはstdio.hを単純にインクルードしている場合にはインクルードする必要はありません．この詳細については「4.2　ヘッダファイル」をご覧下さい．

(3) **解　説**

関数がどのような動作をするかを解説します．また，関数にパラメータがあればその説明も含めてします．

(4) **戻り値**

関数の動作の結果，どのような値が返されるかを解説します．

(5) **参　照**

解説している関数と対比するべき他の関数を示します．

(6) **注　意**

関数を使用するに当たって注意しなければならないことを示します．

(7) **例**

関数を使った簡単な例を示します．この例はコンパイルすればコマンドになるように記述されています．

項目は以上ですが，動作が不明のときにはライブラリソースファイルをご覧になることをお勧めします．

abs

書　式

```
#include <stdlib.h>

int     abs(n)
int     n;
```

解　説

整数nの絶対値を与えます．絶対値とは，あればその数の負号(−)をとった値をいいます．0の絶対値は0，5の絶対値は5，-3の絶対値は3のようになります．

戻り値

整数nの絶対値を返します．つまり，nが0または正の時はnを返し，nが負であれば-nを返します．

注　意

abs( )では整数の負の最大値(-32768)の変換ができません(同じ値が返ってきます)．

例

```
#include <stdio.h>

/*  -3, 0, 5 の絶対値を表示します  */
main()
{
        printf("%d %d %d¥n", abs(-3), abs(0), abs(5));
}
```

alloc

書　式

```
# include <malloc.h>

char      *alloc(n)
size_t    n;
```

解　説

free( )によって解放可能なメモリ領域をnバイト確保します．alloc( )とfree( )は高水準メモリ管理関数で，組合せて使用されます．alloc( )は実行時にその状況に応じて領域が必要なときに使い，free( )はその領域が不要になったときに使います．詳しくは「4.3.3　メモリ管理関数」を参照して下さい．

戻り値

割り当てられた領域の先頭へのポインタが返されます．十分なメモリがない場合にはNULLが返されます．

参　照

free( )，rsvstk( )，sbrk( )

注　意

戻り値は使用できる領域の先頭アドレスですからそれをpとすれば，pから(p+n-1)までが，ユーザーは自由に使うことができます．

free( )で解放された領域でnバイト以上の領域がない場合にはsbrk( )で領域を確保しようとします．つまりnバイトalloc( )し，直後にそこをfree( )してから，(n+m)バイトalloc( )しようとすると((n+sizeof(HEADER))　+　(n+m+sizeof(HEADER)))以上のメモリが必要になります．

例

```
#include <stdio.h>

/*
バッファサイズを指定してファイルハンドルsrcからdstに
ファイルコピーをする関数です
*/
STATUS  cp(dst, src, bufsiz)    /*  copy src to dst with buffer  */
FD      dst, src;               /*  file handles                 */
size_n  bufsiz;                 /*  specify buffer size          */
{
        char    *buf;
        int     datsiz;

        if ((buf = alloc(bufsiz)) == NULL)      /* get temporary buffer */
                return (ERROR);                 /*  not enough memory  */

        while (datsiz = read(src, buf, bufsiz))
                if (write(dst, buf, datsiz) == 0)
                        return (ERROR);

        free(buf);                              /* free temporary buffer */
        return (OK);
}

main(argc, argv)
int     argc;
char    *argv[];
{
        FD      s, d;

        argc--;
        if (argc < 2) {
                puts("Usage: cp <src> <dst>¥n");
                exit(1);
        }

        s = open(argv[1], O_RDONLY);
        d = creat(argv[2]);

        if (s == ERROR || d == ERROR) {
                puts("File cannot open¥n");
                exit(1);
        }
        if (cp(s, d, 2048) == ERROR) {
                puts("File copy failed¥n");
                exit(1);
        }
        close(s);
        close(d);
}
```

atoi

書　式

```
#include <stdlib.h>

int     atoi(s)
char    *s;
```

解　説

文字列を符号つき10進数の整数値に変換します．先頭のスペース，タブは無視され，次に符号(＋か－)が入れられます．10進数以外の文字までを数値として変換します．整数の範囲(-32768から32767まで)を越えたとき，桁あふれは無視されてしまいます．

戻り値

文字列を変換した整数値を返します．

参　照

fscanf( )，scanf( )，sscanf( )の"%d"での変換

注　意

入力文字列が整数の範囲を越えている時に，桁あふれを起こしてもエラーが発生しません．

例

```
#include <stdio.h>

/*
文字列定数"-232"，文字列"732"を数値にしてから表示と
キーボードからの入力を数値にして変数に代入し，表示する
*/
static  char    intstr[] = "732";
char    buf[6];

main()
{
        int     i;

        printf("%d  %d¥n", atoi("-232"), atoi(intstr));
        gets(buf, 6);
        i = atoi(buf);
        printf("%d¥n", i);
}
```

bdos, bdosh

書　式

```
# include <bdosfunc.h>

char    bdos(c[, de, hl])
char    c ;

int     bdosh(c[, de, hl])
char    c ;
```

解　説

MSX-DOS のファンクションコールを呼び出します．第1パラメータにファンクションコール名を指定します．ファンクションコール名はヘッダファイル bdosfunc.h で# define されています．第2，第3パラメータは必要ないときは設定しなくて構いません．指定されたときはそれぞれ de，hl レジスタに渡され，ファンクションコールを呼び出します．

戻り値

bdos( )の場合はファンクションコール終了時の a レジスタの値を char 型で返します．bdosh ( )はファンクションコール終了時の hl レジスタの値を int 型で返します．

参　照

call( )，calla( )，callxx( )

注　意

第1パラメータは，ファンクションコール名(または番号)を指定しますが，必ず char 型か TINY 型で指定して下さい．指定していない場合はファンクションコール名が正しく MSX-DOS に渡りません．

MSX-C　Ver.1.1 までの bdosfunc.h とはファンクションコール名が変わっていますので，それを使っている場合は変更が必要です．ファンクションコール名は MSX-DOS2 のファンクション仕様書にあるラベル名と同じになっています．

例

```
#include <stdio.h>
#include <bdosfunc.h>

/*  つながっているドライブを表示する  */
main()
{
        TINY    i, login;

        login = (TINY)bdosh(_LOGIN);
        for (i = 0; i < 8; i++) {
                if (login & 1)
                        printf("drive %c is active¥n", 'A' + i);
                login >>= 1;
        }
}
```

bios

書　式

```
# include <bdosfunc.h>

char    bios(code)
char    code ;
```

解　説

MSX-DOS の BIOS コールをする.

戻り値

BIOS コール後の a レジスタの値を char 型にして返します.

例

```
#include <stdio.h>
#include <bdosfunc.h>

main()
{
        char    c;

        c = getch();
        bdos(_CONOUT, c);
        bios(BIOS_CONOUT, c);
}
```

call, calla

書　式

```
int     call(addr, a, hl, bc, de)
int     *addr, a, hl, bc, de ;

char    calla(addr, a, hl, bc, de)
int     *addr, a, hl, bc, de ;
```

解　説

レジスタを指定して特定番地を呼び出します．アセンブラで記述してあるサブルーチン等を使用する場合に利用します．

戻り値

call( )はaddrで示されるルーチン終了時のhlレジスタの値をint型にして返します．calla( )はaddrで示されるルーチン終了時のaレジスタの値を返します．

注　意

ix, iyと裏レジスタは設定できません．ix, iyを指定する場合はcallxx( )をお使い下さい．

例

```
#include <stdio.h>
#include <bdosfunc.h>
#define dummy 0

main(argc, argv)
int     argc;
char    *argv[];
{
        FD      fd;

        if (--argc < 2) {
                puts("Usage: hrename <pathname> <basename>¥n");
                exit(1);
        }
        if ((fd = open(argv[1], O_RDONLY)) == ERROR) {
                puts("File not open¥n");
                exit(1);
        }
/*  ファンクションコール' ファイルハンドルの名前の変更' を実行する  */
        if (calla(BDOS, dummy, argv[2], fd * 256 + _HRENAME, dummy)) {
                puts("Handle rename failed¥n");
                exit(1);
        }
}
```

callxx

書　式

```
#include <bdosfunc.h>

VOID    callxx(adrs, reg)
unsigned adrs ;
XREG    *reg ;
```

解　説

レジスタを設定して adrs で示されるルーチンを呼び出します．設定できるレジスタは af，ix, iy, bc, de, hl です．レジスタは XREG 型の構造体に代入しておきます．callxx( )実行後は XREG 型の構造体には呼び出されたルーチンから返ってきた各レジスタの値が入っています．この関数は必要な値を MSX-DOS のファンクションコールに渡したり，ファンクションコールからの値を受け取るのに利用します．XREG 型は bdosfunc.h の中に定義されています．

戻り値

ありません．

参　照

bdos( )，bdosh( )，call( )，calla( )

注　意

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>
#include <bdosfunc.h>

main()
{
        XREG    reg;
/*  ＤＯＳのバージョンを表示する  */
        reg.bc = (unsigned)_DOSVER;
        callxx(BDOS, &reg);
        printf("MSX-DOS Version     %x.%02x¥n", reg.bc / 256, reg.bc % 256);
        printf("MSXDOS2.SYS Version %x.%02x¥n", reg.de / 256, reg.de % 256);
}
```

chdir

書　式

```
#include <direct.h>

STATUS  chdir(path)
char    *path;
```

解　説

カレントディレクトリを変更します．pathはディレクトリ名を指定します．ルートディレクトリからの指定でも，カレントディレクトリからの指定でもできます．ドライブをpath中に指定するとそのドライブのカレントディレクトリが変更されます．([d：][¥]pathの形で指定します．)

戻り値

カレントディレクトリが変更された場合にはOKを，失敗した場合にはERRORを返します．

参　照

getcwd(), mkdir(), rmdir()

注　意

chdir()ではカレントドライブの変更はできません．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
/*  ルートディレクトリにbinというディレクトリがあるかを調べる  */
        if (chdir("¥¥bin") == ERROR)
                puts("There is no '¥¥bin' Subdirectory.¥n");
        chdir("¥¥");
}
```

clearerr

書　式

```
# include <stdio.h>

VOID    clearerr(fp)
FILE    *fp ;
```

解　説

fp で示されたファイルのエラー状態を解除します．clearerr( ) は標準入力から入力して，一度エンドオブファイル(EOF)となった後，もう一度入力する場合に clearerr( ) でエンドオブファイル(EOF)のフラグをクリアし，読み込める状態にするのによく使われる関数です．

戻り値

戻り値はありません．

注　意

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        int     c;
/*  標準入力がエンドオブファイルになった後，再度入力する  */
        while ((c = getchar()) != EOF)
                putchar((char)c);
        if (getchar() == EOF)
                puts("now EOF¥n");
        else
                puts("not EOF¥n");
        clearerr(stdin);
        while ((c = getchar()) != EOF)
                putchar((char)c);
}
```

close

書　式

```
# include <io.h>

STATUS  close(fd)
FD      fd ;
```

解　説

ファイルハンドル fd で示されるファイルをクローズします.

戻り値

ファイルハンドルのクローズが成功すれば OK を返します. そうでなければ ERROR を返します.

注　意

close( ) は低水準入出力関数です.

例

```
#include <stdio.h>
static  char    msg[] = "write open and close";

main()
{
        FD      fd;
/*  ファイルにデータを書いて, クローズして終了する  */
        if ((fd = open("data", O_WRONLY)) == ERROR) {
                puts("File cannot make¥n");
                exit(1);
        }
        write(fd, msg, strlen(msg));
        close(fd);
}
```

creat

書　式

```
# include <io.h>

FD      creat(filename)
char    *filename ;
```

解　説

filename で示されたファイルを作成します．ファイルの作成とはファイルがすでにあるときは，そのファイルの内容を消去してからオープンします．

戻り値

ディスクエラーなどでファイルがオープンできないときはERROR を返します．そうでないときはファイルハンドルを返します．

参　照

close( ), open( ), read( ), write( )

注　意

creat( ) は低水準入出力関数です．

例

```
#include <stdio.h>
static  char    str[] = "test data";

main()
{
        FD      fd;
/*  ファイルを新規に作成して，データを書いて，クローズして終る  */
        if ((fd = creat("creat.dat")) == ERROR) {
                puts("File cannot creat¥n");
                exit(1);
        }
        write(fd, str, strlen(str));
        close(fd);
}
```

eof

書　式

```
#include<io.h>

BOOL    eof(fd)
FD      fd;
```

解　説

ファイルハンドル fd が，ファイルの終わり(エンドオブファイル，EOF)まで読み込んだかどうかを調べます．この関数は標準入力がリダイレクトされていて，getch( )で文字を取り込みたいときにあらかじめ EOF かどうかを調べることで，標準入力の EOF エラーでコマンドが終了してしまうことを避けることができます．

戻り値

ファイルハンドルが EOF であるか，ファイルがオープンされていないと TRUE を返します．そうでなければ FALSE を返します．

参　照

feof( )

注　意

この eof( )は低水準入出力関数に対応する EOF を調べる関数です．高水準入出力関数でアクセスしたファイルハンドルに対して行っても，正しい値を返しません．高水準入出力関数に関しては feof( )を使用して下さい．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>
char    buf[100];

main()
{
        FD      fd;
/*  ファイルを100バイトずつエンドオブファイルまで読む  */
        fd = open("file", O_RDONLY);    /*  存在するファイルを指定する  */
        while (read(fd, buf, 100)) {
                puts(eof(fd) ? "TURE¥n" : "FALSE¥n");
        }
        puts(eof(fd) ? "TURE¥n" : "FALSE¥n");
        close(fd);
}
```

execl, execlp

書　式

```
#include <process.h>

VOID    execl(progname, arg1, arg2, ...)
char    *progname, *arg1, *arg2, ... ;

VOID    execlp(progname, arg1, arg2, ...)
char    *progname, *arg1, *arg2, ... ;
```

解　説

execl( ), execlp( )はどちらも別のプログラムファイルを読み込み，それを実行します．文字列prognameには，実行すべきファイル名を与えます．ファイル名が拡張子を持たない場合には，.comが仮定されます．prognameで指定したファイルがない場合に，execl( )は"cannot exec： <progname>"と標準エラー出力に出力して関数から戻ってきます．しかし，execlp( )は環境変数PATHの示すディレクトリから探します．それでも見つからないとexeclp( )は"cannot exec： <progname>"と標準エラー出力に出力して関数から戻ってきます．

arg1，arg2，...で指定される文字列がコマンド引数としてチェインされるプログラムに渡されます．

現在オープンされている低水準入出力ファイルはオープンされたまま指定のコマンドが実行されます．

```
execl("time", "is", "money.");
```

これはコマンド行で

```
A>time is money.↵
```

とコマンドを与えるのと同じ効果を持ちます．

戻り値

ありません．

参　照

execv( )，execvp( )

注 意

execl( ), execlp( )は可変パラメータ関数なので

```
VOID    execl(.), execlp(.);
```

という宣言があらかじめ必要です．この宣言はヘッダファイル process.h に含まれています．

execl( )は，プログラムが見つからないときの動作が変更されました．

execlp( )は Ver.1.2 から標準ライブラリに追加されました．

execlp( )は progname にドライブ名やルートディレクトリからの指定があると，ファイルを環境変数 PATH の示すディレクトリから見つけることができません．

例

```
#include <stdio.h>

/*
コマンドのパラメータが１から３個の時，パラメータを逆順にし，
最後のパラメータをコマンドとして起動
*/
main(argc, argv)
int     argc;
char    *argv[];
{
        switch (argc) {
        case    1:
                execlp(argv[1]);
        case    2:
                execlp(argv[2], argv[1]);
        case    3:
                execlp(argv[3], argv[2], argv[1]);
        }
}
```

execv, execvp

書　式

```
# include <process.h>

VOID    execv(progname, argv)
char    *progname, *argv[] ;

VOID    execvp(progname, argv)
char    *progname, *argv[] ;
```

解　説

execv( ), execvp( )はどちらも別のプログラムファイルを読み込み，それを実行します．文字列 progname には，実行すべきファイル名を与えます．ファイル名が拡張子を持たない場合には，.com が仮定されます．progname で指定したファイルがない場合に，execv( ) は "cannot exec： <progname>"と標準エラー出力に出力して関数から戻ってきます．しかし，execvp( ) は環境変数 PATH の示すディレクトリから探します．それでも見つからないと execvp( ) は "cannot exec： <progname>"と標準エラー出力に出力して関数から戻ってきます．

実行するコマンドへのパラメータは，コマンド引数のアドレスの配列を作っておき，それをパラメータ argv として与えます．また配列 argv の最後の要素には値 NULL を入れます．つまり，argv[0]にはじめの引数のアドレス，argv[1]に 2 番目の引数のアドレス，という順に収め，最後に NULL を入れます．

現在オープンされている低水準入出力ファイルはオープンされたまま指定のコマンドが実行されます．5 つの標準入出力ファイルはリダイレクトなどがされていれば，起動されたコマンドでもそのままとなります．

```
static  char    *param[]={
"all work", "and", "no play", "makes", "Jack", "a dull boy.", NULL
};

execv("foo", param);
```

は，コマンド行から

```
A>foo all work and no play makes Jack a dull boy.↵
```

と入力したのと同じ効果を持ちます．

戻り値

ありません.

参　照

execl( ), execlp( )

注　意

execv( )は，プログラムが見つからないとき，関数から戻ってくるように動作が変更されました.

execvp( )は Ver.1.2 から標準ライブラリに追加されました.

execvp( )は progname にドライブ名やルートディレクトリからの指定があると，ファイルを環境変数 PATH の示すディレクトリから見つけることができません.

例

```
#include <stdio.h>

/*
プログラムが始まることを表示してから，ひとつ目以降のパラメータを
コマンド列として起動
*/
main(argc, argv)
int     argc;
char    *argv[];
{
        if (argc > 1) {
                printf("Start program %s¥n", argv[1]);
                execvp(argv[1], &argv[2]);
                /*  コマンドが見つからない  */
                puts("Sorry...¥n");
        }
}
```

exit, _exit

書　式

```
# include <process.h>

VOID    exit(code)
int     code ;

VOID    _exit(code)
int     code ;
```

解　説

実行中のコマンドを中止してコマンドレベルに戻ります．この時リターンコード(code)を指定して戻ることができます．また，exit( )は高水準入出力のすべてのバッファをフラッシュ，クローズします．それに対して_exit( )はフラッシュ，クローズをしません．ですが，低水準入出力のファイルハンドルは，exit( )，_exit( )のどちらもクローズします．

注　意

_exit( )はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
FILE    *fp;
/*  file.datがなかったら"File not found"を表示してプログラムを終了  */
        if ((fp = fopen("file.dat", "r")) == NULL) {
                puts("File not found¥n");
                exit(1);
        }
}
```

expargs

書　式

```
# include <direct.h>

int     expargs(argc, argv, maxargc, xargv)
int     argc, maxargc ;
char    *argv[], *xargv[] ;
```

解　説

ワイルドカードを含むコマンド引数に対し，実際にディレクトリサーチをすることによってそれを各々のファイル名に展開する関数です．ワイルドカードが含まれているときには，展開されたファイル名をソートします．複合ファイルスペックも指定することができます．ただし，ファイルスペックの区切りとしては"+"しか使うことはできません．この関数を利用すると，ワイルドカード文字を実際のファイル名に展開する機能を持つプログラムを書くことができます．

パラメータ argc，argv は展開する引数の数と文字列へのポインタの配列です．xargv も argv と同じ文字列へのポインタの配列であり，そこには展開された各々のコマンド引数のアドレスがセットされます．また，xargv の大きさを maxargc として与えます．

戻り値

展開されたコマンド引数の個数が関数の値として返されます．ただし，引数を展開するメモリが不足した場合と展開後の引数の個数が maxargc を越えた場合には，値 ERROR が返されます．

注　意

戻り値として ERROR が返された場合には，配列 xargv[]の内容は無効です．

例

```
#include <stdio.h>
#define MAXARG  100
char    *xargv[MAXARG];

/*  パラメータのワイルドカードを展開して表示  */
main(argc, argv)
int     argc;
char    *argv[];
{
        int     i, n;

        if (argc > 1) {
                argc--;
                argv++;
                n = expargs(argc, argv, MAXARG, xargv);
                for (i = 0; i < n; i++)
                        printf("%s¥n", xargv[i]);
        }
}
```

fclose, fcloseall

書　式

```
# include <stdio.h>

STATUS   fclose(fp)
FILE     *fp ;

TINY     fcloseall( )
```

解　説

fclose( ) は fp で示されたファイルをクローズします．fcloseall( ) は stdin，stdout，stderr，stdaux，stdprn 以外のオープンしているすべてのファイルをクローズします．テキストモードでかつライトモードであったときには ^Z(EOF 文字)を出力してからクローズします．また，ファイルバッファをシステムが自動的に割り当てたなら，それを解放します．fclose( ) では高水準入出力ファイルのみを閉じることができます．低水準入出力ファイルには close( ) を使って下さい．exit( ) で終了したり，プログラムが正常終了したときには，すべてのファイルを fclose( ) でクローズしてからコマンドレベルに戻ります．

戻り値

fclose( ) は fp が NULL であったり，クローズの処理に失敗した場合には ERROR が返されます．それ以外では OK が返されます．

fcloseall( ) はひとつでもファイルのクローズに失敗すると ERROR を返します．クローズにすべて成功するとクローズしたファイルの数を返します．

注　意

fcloseall( ) は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
/*  ファイルをエンドオブファイルまで読み込んで，クローズして終る  */
        if ((fp = fopen("file", "r") == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        puts("Start!!!¥n");
        while ((c = getc(fp)) != EOF)
                putchar((char)c);
        puts("End of file¥n");
        fclose(fp);
}
```

feof

書　式

```
# include <stdio.h>

BOOL    feof(fp)
FILE    *fp ;
```

解　説

ファイルを最後まで読み込んだかどうかを調べます．ファイルは最後(エンドオブファイル，EOF)まで読み込むと，それを表すフラグが立ちます．それを見るのがfeof( )です．EOFはテキストモードか，バイナリモードかによって位置が変わります．テキストモードのときにはEOF文字(^Z)がエンドオブファイルを示し，バイナリモードでは物理的なファイルの終端がエンドオブファイルになります．EOFの状態をクリアするにはclearerr( )を使います．

戻り値

EOFまで読み込んでいた場合にはTRUEを返します．そうでない場合にはFALSEを返します．

参　照

clearerr( )，eof( )

注　意

この関数はマクロです．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        int     i = 10;
        char    c;
/*
標準入力の文字を１０文字出力する．エンドオブファイルとなったら
スペースを表示する．
*/
        while (i && (c = getchar()) != EOF) {
                putchar(c);
                i--;
        }
        if (feof(stdin))
                while (i--)
                        putchar(' ');
}
```

ferror

書　式

```
# include <stdio.h>

BOOL    ferror(fp)
FILE    *fp ;
```

解　説

ファイルが書き込み中にエラーが発生したかどうかを調べます．これは主にディスクフルによって発生します．エラーの状態をクリアするには clearerr( )を使います．

戻り値

エラーが発生していたら TRUE を返します．そうでないときは FALSE を返します．

注　意

この関数はマクロです．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
/*  ディスクがいっぱいになるまでデータを書き込む  */
        if ((fp = fopen("nomore", "w")) == NULL)
                exit(1);                /*  file cannot make  */
        while (!ferror(fp))
                fputs("Write until disk full¥n", fp);
        fputs("Disk full !!!¥n", stderr);
}
```

fflush

書　式

```
# include <stdio.h>

STATUS  fflush(fp)
FILE    *fp;
```

解　説

ファイルのバッファをディスクに書き出します．高水準入出力関数ではファイルに出力されたデータはバッファに1度格納されます．そのバッファがいっぱいになると自動的にディスクに書き出します(バッファのフラッシュという)が，強制的にfflush( )によってバッファの内容をバッファフルになる前にディスクに書き出し，バッファを空にします．標準出力に改行文字(¥n)なしで文字列を表示したいときにはfflush( )を使う必要があります．フラッシュした後もファイルはオープンされた状態です．

戻り値

バッファのフラッシュが成功したときはOKを返します．ディスクフルのときにはERRORが返されます．

参　照

「4.3.1　B)高水準入出力関数のバッファリング」
flushall( )

注　意

ファイルをクローズしたときや，プログラムが正常終了したときには自動的にバッファをフラッシュし，空にします．
この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        int     i, sum;
/*  途中経過を表示しながら100までの和を計算します.    */
        sum = 0;
        for (i = 0; i <= 100; i++) {
                sum += i;
                printf("%4d %5d¥r", i, sum);
                fflush(stdout);
        }
        printf("  sum = %d¥n", sum);
}
```

fgets

書　式

```
# include <stdio.h>

char    *fgets(s, n, fp)
char    *s ;
int     n ;
FILE    *fp ;
```

解　説

fp で示されたファイルから文字列を読み込みます．文字列は"¥n"を終わりとして，s で示される領域に読み込みます．また，領域のサイズを n によって指定します．文字列の最後には"¥n"がつき，その後に文字列の終了文字として"¥0"が入ります．読み込んでいる最中にファイルがエンドオブファイルに達した場合は"¥n"は入らずに"¥0"が置かれます．fgets( ) は最大で n-1 文字を読み込み最後に"¥0"を置きます．

戻り値

ファイルがエンドオブファイルに達したときと，何も読み込まれなかったときは NULL が返されます．それ以外の時は s が返されます．

参　照

gets( ), fputs( ), puts( )

例

```
#include <stdio.h>
char    buf[256];

main()
{
        FILE    *fp;
/*  ファイルmemoの内容を表示する  */
        if ((fp = fopen("memo", "r")) == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        while (fgets(buf, 256, fp))
                puts(buf);
        fclose(fp);
}
```

fileno

書　式

```
# include <stdio.h>

FD      fileno(fp)
FILE    *fp ;
```

解　説

fp で示されるファイルの入出力に使用しているファイルハンドルを返します．高水準のファイル入出力は，必ずひとつの低水準入出力のファイルハンドルを持っています．ファイルポインタからそのファイルハンドルの値を得るときに fileno( ) を使います．

戻り値

対応するファイルハンドルを返します．fp がオープンされていないファイルである場合には，値は保証されません．

注　意

この関数はマクロです．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
/*  stdinから
stdprnまでの対応するファイルハンドルを表示する  */
        for (fp = stdin; fp <= stdprn; fp++)
                printf("file handle %d¥n", fileno(fp));
}
```

flushall

書　式

#include <stdio.h>

TINY　　flushall( )

解　説

ライトモードでオープンされているすべてのファイルのバッファをディスクに書き出します．フラッシュした後もファイルはオープンされた状態です．

戻り値

オープンされているファイルの数を返します．これにはリードモード，ライトモードの区別はありません．

参　照

fflush ( )

注　意

ファイルをクローズしたときや，プログラムが正常終了したときには自動的にバッファをフラッシュし，クローズします．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        TINY    file;
/*  いくつファイルをオープンしているかを表示する  */
        file = flushall();
        printf("%d files are open¥n", (int)file);
}
```

fopen

書　式

```
#include <stdio.h>

FILE    *fopen(filename, mode[, bufsize])
char    *filename, *mode ;
size_t  bufsize ;
```

解　説

filenameで示されるファイルを高水準入出力が行えるようにオープンします．文字列modeは次のうちのいずれかでなければなりません．

"r"	リードモード，テキストモード
"rb"	リードモード，バイナリモード
"w"	ライトモード，テキストモード
"wb"	ライトモード，バイナリモード
"a"	アペンドモード，テキストモード
"ab"	アペンドモード，バイナリモード

リードモードは読み出しのためにあり，ファイルが存在しなければなりません．ライトモードは書き込み用で，ファイルがすでにあったときにはまず，そのファイルを消してからオープンします．つまり，いままでの内容は消えてしまいます．アペンドモードはいまあるファイルにデータを追加するために使います．ファイルがないときはライトモードと同じ動作になります．

テキストモードとは入力時は"¥r"＋"¥n"を"¥n"の1文字に，出力時は"¥n"を"¥r"＋"¥n"の2文字に変換し，^Zをエンドオブファイルとする，ファイルの入出力モードです．モードの2文字目に"b"を指定しないとテキストモードになります．ソースファイルなどの入出力に使われます．

バイナリモードではテキストモードと違って入力時"¥r"＋"¥n"，出力時の"¥n"などの変換は一切しません．また，^Zもエンドオブファイルにしません．

bufsizeには入出力に使われるバッファのサイズを指定できます．省略されたときにはBUFSIZ(1024)になります．

戻り値

ファイルのオープン数が多過ぎたり，ファイルが見つからなかった場合にはNULLが返されます．オープンに成功するとFILE構造体へのポインタが返されます．

参　照

fsetbin( )，fsettext( )，setbuf( )，setvbuf( )

注　意

fopen( )は可変パラメータ関数なので，

```
FILE    *fopen(.);
```

という宣言が必要です．この宣言は，ヘッダファイル stdio.h に含まれています．

例

```
#include <stdio.h>
char    buf[256];

main()
{
        FILE    *fp;
/*
ファイルnewgame.datの内容を表示する．
ファイルがなければ"File not found"を表示する．
*/
        if ((fp = fopen("newgame.dat", "r") == NULL) {
                fputs("File not found¥n", stderr);
                exit(1);
        }
        while (fgets(buf, 256, fp)) {
                puts(buf);
        fclose(fp);
}
```

fprintf

書　式

```
# include <stdio.h>

STATUS   fprintf(fp, format[, arg1 , arg2, ...])
FILE     *fp ;
char     *format ;
```

解　説

fprintf( ) は fp で示されるファイルに対して書式付きの出力を行ないます．制御文字列については，printf( ) と同じですのでその項を参照して下さい．

戻り値

出力中にエラーがなければ OK が返されます．そうでなければ ERROR が返されます．

参　照

printf( )，sprintf( )

注　意

printf( ) は可変パラメータ関数なので，次の宣言があらかじめ必要です．

```
STATUS  fprintf(.);
```

この宣言は，ヘッダファイル stdio.h に含まれています．

fprintf( ) は UNIX の標準 C のサブセットになっています．

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
        int     array[10];
        int     count;
/*  整数の配列をファイルintdat.datに保存する  */
        if ((fp = fopen("intdat.dat", "w")) == NULL) {
                fprintf(stderr, "File cannot make¥n");
                exit(1);
        }
        for (count = 0; count < 10; count++)
                fprintf(fp, "%d¥n", array[count]);
        fclose(fp);
}
```

fputs

書　式

```
# include <stdio.h>

STATUS   fputs(s, fp)
char     *s ;
FILE     *fp ;
```

解　説

ファイルポインタ fp で示されたファイルに文字列 s を出力します．出力に対して文字列の最後に"¥n"をつけたりはしません．文字列 s はヌル文字で終っていなければなりません．また，ヌル文字は出力されません．文字列中に"¥n"があった場合にはテキストモードであれば"¥r"と"¥n"にして出力します．

戻り値

出力中にエラーが発生した場合には ERROR が返されます．それ以外のときは OK が返されます．

参　照

fgets( ), gets( ), puts( )

例

```
#include <stdio.h>
char    buf[256];

main()
{
        FILE    *fp;
        /*  キーボードからの入力をファイルmemoに保存する  */
        if ((fp = fopen("memo", "w")) == NULL) {
                puts("File cannot make¥n");
                exit(1);
        }
        while (gets(buf, 256))
                fputs(buf, fp);
        fclose(fp);
}
```

fread

書　式

```
# include <stdio.h>

int     fread(buf, size, count, fp)
char    *buf ;
int     size, count ;
FILE    *fp ;
```

解　説

ファイルポインタ fp で示されるファイルから buf で指定された領域に size バイトを count 回(項目)，データを読み込みます．fread( ) は，低水準入出力関数の read( ) の高水準入出力関数版と考えればいいでしょう．読み込むデータ量は単純なバイト数ではなく，どういう型(size)のものをどれだけ(count)読み込むかで指定します．

戻り値

読み込むことができた回数(項目数)を返します．ファイルがエンドオブファイルであったときには 0 になります．

参　照

fsetbin( )，fwrite( )，read( )

注　意

読み込むデータがバイナリデータであるなら fsetbin( ) をするか，バイナリモードで fopen( ) する必要があります．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
        int     array[10];
        int     count;
/*  整数の配列にバイナリ形式で保存してあるデータを読み込む  */
        /*バイナリモードでオープン  */
        if ((fp = fopen("worddat.dat", "rb")) == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        count = fread(array, sizeof(int), 10, fp);
}
```

free

書　式

```
# include <malloc.h>

VOID    free(ap)
char    *ap ;
```

解　説

alloc( )で割り当てられた領域(apで示す)を解放し，再利用を可能にします．alloc( )とfree( )は高水準メモリ管理関数で，組合せて使用されます．alloc( )は実行時にその状況に応じて領域が必要なときに使い，free( )はその領域が不要になったときに使います．free( )によって，限りあるメモリを有効に使うことができます．詳しくは「4.3.3　メモリ管理関数」を参照して下さい．

戻り値

ありません．

参　照

alloc( )，rsvstk( )，sbrk( )

注　意

free( )で解放された領域でnバイト以上の領域がない場合にはsbrk( )で領域を確保しようとします．つまりnバイトalloc( )し，直後にそこをfree( )してから，(n+m)バイトalloc( )しようとすると((n+sizeof(HEADER)) ＋ (n+m+sizeof(HEADER)))以上のメモリが必要になります．

例

```
#include <stdio.h>
char    buf[100];

main()
{
        char    *p, **ary;
        int     n = 0;
        /*  キーボードからの入力を逆順に表示する  */
        if ((ary = (char **)alloc(sizeof(char *) * 100)) == NULL) {
                puts("Not enough memory¥n");
                exit(1);
        }
        while (gets(buf, 100)) {
                if ((p = alloc(strlen(buf)+1)) == NULL) {
                                /*入力の分だけ領域を取る*/
                        fputs("No more memory¥n", stderr);
                        break;
                }
                strcpy(p, buf);         /*  確保されてる領域にセーブ  */
                ary[n++] = p;           /*  領域へのポインタをセーブ  */
        }
        while (n) {
                puts(ary[n]);           /*  1行を表示                 */
                free(ary[n--]);         /*  文字列に使った領域を解放する*/
        }
        free(ary);                      /*  配列に使った領域を解放する  */
}
```

fscanf

書　式

```
# include <stdio.h>

int     fscanf(fp, format[, arg1 , arg2, ...])
FILE    *fp ;
char    *format ;
```

解　説

fscanf( ) は fp で示されたファイルから書式変換付きの入力を行ないます．format 以降のパラメータは scanf( ) と同じなのでその項を参照して下さい．

戻り値

fscanf( ) は，実際に代入された項目数を値として返します．戻り値が 1 で，変数へのポインタを 3 個パラメータに指定したら，2 個目以降は値が代入されていないことになります．また，ファイルの終わりに到達した時には値 EOF が返されます．

参　照

scanf( )，sscanf( )

注　意

"%c"で文字を代入するときはラインフィード文字"¥n"が代入されてしまうこともありますので，変換文字の指定には注意して下さい．fscanf( ) は可変パラメータ関数なので，次の宣言があらかじめ必要です．

```
int     fscanf(.);
```

この宣言は，ヘッダファイル stdio.h に含まれています．

fscanf( ) は UNIX の標準 C のサブセットになっています．

例

```
#include <stdio.h>
char    buf[100];

main()
{
        int     val, n;
        char    c;
        FILE    *fp;
        /*  ファイルから１０進数，文字，文字列を受け取る  */
        if ((fp = fopen("file.dat", "r")) == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        n = fscanf(fp, "%d %c %s", &val, &c, buf);
        printf("%d matched  decimal %d:char '%c':string '%s'¥n",
                n, val, c, buf)
        fclose(fp);
}
```

fsetbin

書　式

```
#include <stdio.h>

STATUS  fsetbin(fp)
FILE    *fp;
```

解　説

fpで示されたファイルをバイナリモードとしてアクセスする．バイナリモードではテキストモードと違って入力時"¥r"＋"¥n"，出力時の"¥n"などの変換は一切しません．また，^Zもエンドオブファイルにしません．エンドオブファイルになるには物理的なファイルの終端に達したときです(ファイルの長さで判定する)．fopen()でファイルをオープンするときにもバイナリモードにすることができます．モードの2文字目に"b"を加えます．

作成したファイルにコマンドレベルでリダイレクトマーク(＞＞)によってデータを追加したいときには，fclose()を実行する前にfsetbin()を実行すれば可能です．これはバイナリモードであれば^Zが出力されないので，それによってファイルが区切られないためです．

戻り値

必ずOKが返されます．

参　照

fopen()，fsettext()

例

```
#include <stdio.h>
char    buf[256];

main()
{
        int     i, c;
        FILE    *fp;
        /*  まずはバイナリモードでオープン  */
        if ((fp = fopen("str.dat", "rb")) == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        while ((i = getc(fp)) != EOF) { /* 文字列の数を受け取る*/
                fsettext(fp);   /* これから先はテキストモードで読み込む */
                while (i--) {   /* 整数分だけ文字列を表示 */
                        fgets(buf, 256);
                        puts(buf);
                }
                fsetbin(fp);    /*  整数を取るためにバイナリモードに  */
        }
        fclose(fp);
}
```

fsettext

書　式

```
# include <stdio.h>

STATUS  fsettext(fp)
FILE    *fp;
```

解　説

fp で示されたファイルをテキストモードとしてアクセスする．テキストモードとは入力時は"¥r"＋"¥n"を"¥n"の１文字に，出力時は"¥n"を"¥r"＋"¥n"の２文字に変換し，^Z をエンドオブファイルとする，ファイルの入出力モードです．fopen( )によってオープンされたファイルの最初の状態はテキストモードとなっています．

戻り値

必ず OK を返します．

参　照

fopen( )，fsetbin( )

例

```
#include <stdio.h>
char    buf[256];

main()
{
        int     i, c;
        FILE    *fp;
        /*  まずバイナリモードでオープン  */
        if ((fp = fopen("str.dat", "rb")) == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        c = getc(fp);   /*  ファイルの先頭の２バイトを整数とする  */
        i = c + getc(fp) * 256;
        fsettext(fp);   /*  これから
先はテキストモードで読み込む  */
        while (i--) {   /*  整数分だけ文字列を表示  */
                fgets(buf, 256);
                puts(buf);
        }
        fclose(fp);
}
```

fwrite

書　式

```
#include <stdio.h>

int     fwrite(buf, size, count, fp)
char    *buf ;
int     size, count ;
FILE    *fp ;
```

解　説

fp で示されるファイルに buf で指定された領域から size バイトを count 回(項目)のデータを書き込みます. fwrite( ) は, 低水準入出力関数の write( ) の高水準入出力関数版と考えればいいでしょう. 書き込むデータ量は単純なバイト数ではなく, どういう型(size)のものをどれだけ(count)書き込むかで指定します.

戻り値

書き込むことができた回数(項目数)を返します.

参　照

fread( ), fsetbin( ), write( )

注　意

書き込むデータがバイナリデータであるなら fsetbin( ) をするか, バイナリモードで fopen( ) する必要があります.

この関数は Ver.1.2 から標準ライブラリに追加されました.

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
        int     array[10];
        int     count;
/*  整数の配列をバイナリ形式でファイルworddat.datに保存  */
        /*  バイナリモードでオープン  */
        if ((fp = fopen("worddat.dat", "wb")) == NULL) {
                puts("File cannot make¥n");
                exit(1);
        }
        for (count = 0; count < 10; count++)
                scanf("%d", &array[count]);
        count = fwrite(array, sizeof(int), 10, fp);
}
```

getc, getchar

書　式

```
# include <stdio.h>

int     getc(fp)
FILE    *fp ;

int     getchar( )
```

解　説

getc( ) は fp で示されるファイルから 1 文字読み込みます. getchar( ) は標準入力から 1 文字読み込みます. ファイルの終端(エンドオブファイル)に到達した時は, 値 EOF を返します.

戻り値

ファイルの終わりを越えて読み込んだ時は EOF を返します. そうでないときは読み込んだ文字を int 型にして返します.

参　照

getch( ), getche( )

注　意

デバイス(キーボード)から文字を読み込むときは, MSX-DOS2 の仕様によって行バッファリングが行われます. つまり, リターンキーが押されるまでは getc( ), getchar( ) から戻ってきません. getc( ) と getchar( ) は, int 型の値を返します.

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
        /*  ファイルを最初から最後まで表示  */
        if ((fp = fopen("file", "r") == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        puts("Start!!!¥n");
        while ((c = getc(fp)) != EOF)
                putchar((char)c);
        puts("End of file¥n");
        fclose(fp);
}
```

getch

書　式

```
# include <conio.h>

char    getch( )
```

解　説

標準入力(ファイルハンドル 0)からエコーバックなしで 1 文字入力します. getch( ) は入力待ちをし, 1 文字入力されるとその文字を返します. Ctrl+C や Ctrl+Z などのコントロール文字も入力することができます. 行バッファリングが行われません(テンプレートは使用できない). EOF が存在しないため, 標準入力がリダイレクトされている場合には, eof( ) でエンドオブファイルかどうかを調べた方がいいでしょう. また, read( ) で標準入力(ファイルハンドル 0)から読み込んだり, getchar( ) などで stdin から読み込むなど, 他の読み込み関数との混用は避けた方がいいでしょう.

戻り値

入力された文字が返されます. 文字として Ctrl+C や Ctrl+Z も入力できます.

参　照

eof( ), getche( )

注　意

この関数は Ver.1.2 から標準ライブラリに追加されました.

例

```
#include <stdio.h>

int     get()
{
        if (eof(STDIN))
                return (EOF);
        else
                return (getch());
}

/*  標準入力から入力された文字を表示  */
main()
{
        int     c;

        while ((c = get()) != EOF) {
                if (c < ' ') {
                        putchar('^');
                        c += '@';
                }
                putchar((char)c);
                fflush(stdout);
        }
}
```

getche

書　式

```
#include <conio.h>

char    getche( )
```

解　説

標準入力(ファイルハンドル0)からエコーバックつきで1文字入力します．getche( )は入力待ちをし，1文字入力されるとその文字を返します．エコーバックは標準出力(ファイルハンドル1)に出力されます．Ctrl+C やCtrl+P などのコントロール文字は，その文字の機能として働きます．1文字ごとの入力が行われるため，行バッファリングは行われません(テンプレートは使用できない)．また，EOF が存在しないため，標準入力がリダイレクトされている場合には，eof( )でエンドオブファイルかどうかを調べた方がいいでしょう．また，read( )で標準入力(ファイルハンドル0)から読み込んだり，getchar( )などで stdin から読み込むなど，他の読み込み関数との混用は避けた方がいいでしょう．

戻り値

入力された文字が返されます．

参　照

eof( )，getch( )

注　意

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        while (1) {
                putchar(getche());
                fflush(stdout);
        }
}
```

getcwd

書　式

```
#include <direct.h>

char    *getcwd(cwd, n)         /*  get current work directory  */
char    *cwd;
int     n;
```

解　説

カレントドライブとカレントディレクトリをd：¥pathの形で返します．cwdにはカレントディレクトリ名を格納する領域へのポインタを，nにはその大きさを指定します．cwdがNULLであるときにはgetcwd()が領域を自動的に割り当てます．nがカレントディレクトリを格納するのに，不足するような値であったときには，領域いっぱいに格納し，残りは切り捨てます．MSX-DOS2ではファイル名は63文字以下と規定されていますので，最低でも64バイトの領域を確保しておけば確実です．

戻り値

カレントディレクトリ獲得失敗か自動的な領域確保に失敗した場合にはNULLを返します．その他の場合にはカレントディレクトリを示す文字列へのポインタを返します．

参　照

chdir(), mkdir(), rmdir()

注　意

カレントディレクトリがルートディレクトリであった場合には，最後に"¥"がつきます．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>
char    cd[64];

main()
{
        chdir("¥¥");                    /*カレントドライブはAだとすると*/
        getcwd(cd, sizeof(cd));
        puts(cd);                       /*'A:¥'と表示される*/
        chdir("bin");                   /*Aドライブはディレクトリbinをもつ*/
        puts(getcwd(cd, sizeof(cd)));/*'A:¥BIN'と表示される*/
}
```

getenv

書　式

```
# include <stdlib.h>

char    *getenv(var)
char    *var ;
```

解　説

MSX-DOS2 の環境変数値を獲得します．var には獲得したい環境の変数名を指定します．変数値は戻り値が示す領域に文字列として入っています．変数地を格納する領域は毎回 alloc( ) によって割り当てられるので，不要になったら free( ) で解放することができます．

環境変数は MSX-DOS の SET コマンドや，putenv( ) によって設定します．

戻り値

変数が見つからなかったり，変数値を格納するだけの領域が確保できなかったときは NULL を返します．そうでないときは，変数値の格納されている領域を返します．

参　照

alloc( )，free( )，putenv( )

注　意

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>
/*  テンポラリファイルを作る  */
char    *mktmp(fn)
{
        char    *p;
static  int     num = 0;

        if ((p = getenv("TMP")) == NULL)  /*tmpに使うディレクトリを獲得*/
                p = "a:¥¥";
        strcpy(fn, p);
        p = fn + strlen(fn);            /*  points at '¥0'  */
        if (*(p - 1) != '¥¥')
                *p++ = '¥¥';
        sprintf(p, "tmp%03d.$$$", num++);
        return (fn);
}

main(argc, argv)
int     argc;
char    *argv[];
{
        FILE    fp, tmp;
        char    *s, tempfn[64];
        int     c;

        if (--argc < 2) {
                puts("Usage: fcat <objfile> <concatfile1> ...¥n");
                exit(1);
        }
        argv++;
        s = *argv++;
        if ((tmp = fopen(mktmp(tempfn), "w")) == NULL) {
                puts("cannot make temp file¥n");
                exit(1);
        }
        for(;*argv; argv++) {
                if ((fp = fopen(*argv, "r")) != NULL) {
                        while ((c = getc(fp)) != EOF)
                                putc(tmp);
                        fclose(fp);
                }
        }
        fclose(tmp);
        rename(tempfn, s);
}
```

gets

書　式

```
#include <stdio.h>

char    *gets(s, n)
char    *s ;
int     n ;
```

解　説

標準入力(stdin)から文字列を読み込みます．文字列は"¥n"を終わりとして，sで示される領域に読み込みます．また，領域のサイズをnによって指定します．文字列の最後には"¥n"がつき，その後に文字列の終了文字として"¥0"が入ります．読み込んでいる最中にファイルがエンドオブファイルに達した場合は"¥n"は入らずに"¥0"が置かれます．fgets( )は最大でn-1文字を読み込み最後に"¥0"を置きます．

戻り値

標準入力がエンドオブファイルに達したときと，何も読み込まれなかったときはNULLが返されます．それ以外の時はsが返されます．

参　照

fgets( )，fputs( )，puts( )

注　意

MSX-Cのgets( )は，標準的なCのgets( )とは動作が異なります．標準的な動作のgets( )は文字列の最後に"¥n"が入りませんが，MSX-Cのgets( )は"¥n"が入ります．

例

```
#include <stdio.h>
char    buf[256];

main()
{
        FILE    *fp;
/*  キーボードからの入力をファイルに保存する  */
        if ((fp = fopen("memo", "w")) == NULL) {
                puts("File cannot make¥n");
                exit(1);
        }
        while (gets(buf, 256))
                fputs(buf, fp);
        fclose(fp);
}
```

inp

書　式

```
# include <conio.h>

char    inp(port)
unsigned port ;
```

解　説

port で示される I/O ポートからデータを 1 バイト入力します．ポート番号は 0 から 255 までの範囲で指定します．

戻り値

I/O ポートから入力された 1 バイトのデータを char 型として返します．

参　照

outp( )

例

```
#include <stdio.h>

main()
{
        int     i;
        char    knj[32];
/*  漢字ＲＯＭの読み込み  */
        for (i = 0; i < 32; i++)
                knj[i] = inp(0xd9);
}
```

isalnum

書　式

```
# include <ctype.h>

BOOL    isalnum(c)
char    c ;
```

解　説

文字がアルファベットまたは数字かどうかを調べます．つまり，"a"から"z"，"A"から"Z"，"0"から"9"のどれかであるかを調べます．

戻り値

文字がアルファベットか数字であったらTRUEを返します．そうでなかったらFALSEを返します．

参　照

isalpha( )，isdigit( )

注　意

この関数はマクロですので，cには副作用のあるものは指定できません．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

/*  指定された文字列は名前（識別子）か？  */
BOOL    isname(s)
char    *s;
{
        if (!isalpha(*s))
                return (FALSE);
        s++;
        while (!isspace(*s)) {
                if (!isalnum(*s))
                        return (FALSE);
                s++;
        }
        return (TRUE);
}

main(argc, argv)
int     argc;
char    *argv[];
{
        char    buf[256];
        FILE    *fp = stdin;

        if (--argc)
                if ((fp = fopen(argv[1], "r")) == NULL) {
                        puts("File not found¥n");
                        exit(1);
                }
        while (fgets(buf, 256, fp)) {
                puts(buf);
                puts(" is ");
                if (!isname(buf))
                        puts("not ");
                puts("identifier¥n");
        }
        fclose(fp);
}
```

isalpha

書　式

```
# include <ctype.h>

BOOL    isalpha(c)
char    c ;
```

解　説

文字がアルファベットかどうかを調べます．つまり，"a"から"z"，"A"から"Z"のどれかであるかを調べます．

戻り値

文字がアルファベットであったらTRUEを返します．そうでなかったらFALSEを返します．

参　照

islower( )，isupper( )

注　意

この関数はマクロですので，cには副作用のあるものは指定できません．

例

```
#include <stdio.h>

main(argc, argv)
int     argc;
char    *argv[];
{
        if (--argc < 1) {
                puts("Usage: drvname <d:>¥n");
                exit(1);
        }
        if (!isalpha(argv[1][0]) || argv[1][1] != ':' || argv[1][2] != '¥0')
                puts("Argument must be drive name  'd:'¥n");
                exit(1);
        ...
}
```

isatty

書　式

```
#include <io.h>

BOOL    isatty(fd)
FD      fd;
```

解　説

ファイルハンドル fd がデバイスかどうかを調べます．isatty( )はコマンドの動作を，入出力の対象がデバイスファイルかディスクファイルかで変えたい場合に使用します．例えば UNIX のページごとに表示するフィルタ more では，標準出力がデバイスの場合にはキーの入力待ちをしますが，ファイルにリダイレクトされている場合には何もしないフィルタになります．

戻り値

ファイルハンドルがデバイスである場合には TRUE を返します．ファイルであるか，ファイルハンドルがオープンされていない場合には FALSE が返されます．

注　意

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
/*  標準出力はリダイレクトされているか？  */
        if (isatty(fileno(stdout)))
                fputs("stdout is a device¥n", stderr);
        else
                fputs("stdout is a file¥n", stderr);
}
```

iscntrl

書　式

```
# include <ctype.h>

BOOL    iscntrl(c)
char    c ;
```

解　説

文字がコントロール文字かどうかを調べます．(コードでは 0x20 未満と 0x7f の文字)

戻り値

文字がコントロール文字であったら TRUE を返します．そうでなかったら FALSE を返します．

注　意

この関数はマクロですので，c には副作用のあるものは指定できません．

例

```
#include <stdio.h>

main()
{
        int     c;
        FILE    *fp;
/*  ファイルの内容をコントロール文字を含めて表示  */
        if ((fp = fopen("data", "rb")) == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        while ((c = getc(fp)) != EOF) {
                if (c == 0x7f)
                        puts("^?");
                else if (iscntrl((char)c)) {
                        putchar('^');
                        putchar((char)(c - '@'));
                } else
                        putchar((char)c);
        }
}
```

isdigit

書　式

```
# include <ctype.h>

BOOL    isdigit(c)
char    c ;
```

解　説

文字が数字の文字(0 から 9)かどうかを調べます.

戻り値

文字が数字であったら TRUE を返します. そうでなかったら FALSE を返します.

参　照

isxdigit( )

注　意

この関数はマクロですので, c には副作用のあるものは指定できません.

例

```
#include <stdio.h>

main(argc, argv)
int     argc;
char    *argv[];
{
        int     i;
        char    *p;
        if (--argc < 1) {
                puts("Usage: atoi <number> ...¥n");
                exit(1);
        }
        argv++;
/*  atoi()のようなことをしてみる  */
        while (argc--) {
                i = 0;
                p = argv++
                while (isdigit(*p))
                        i = i * 10 + *p++ - '0';
                printf("%d¥n", i);
        }
}
```

iskanji

書　式

```
# include <ctype.h>

BOOL    iskanji(c)
char    c ;
```

解　説

文字が漢字の第1バイトかどうかを調べます．（コードでは0x81から0x9f，0xe0から0xfc）

戻り値

文字が漢字の第1バイトであったらTRUEを返します．そうでなかったらFALSEを返します．

参　照

「3.5　漢字処理の方法」
iskanji2( )

注　意

この関数はマクロですので，cには副作用のあるものは指定できません．漢字かどうかは，調べようとしている文字列の先頭からすべての文字について漢字か，ANKかを見なければ正確には判りません．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

/*  漢字を含んだディレクトリ名からディレクトリの区切りを見つける  */
char    *pathdelimit(s)
char    *s;
{
        while (*s) {
                if (*s == '¥¥')
                        return (s);
                if (iskanji(*s))
                        s++
                s++;
        }
        return (NULL);
}

main(argc, argv)
int     argc;
char    *argv[];
{
        char    *p;
        if (--argc < 1) {
                puts("Usage: pdlmt <path>¥n");
                exit(1);
        }
        argv++;

        p = *argv;
        while (p = pathdelimit(p)) {
                *p = '¥0';
                puts(*argv);
                *p++ = '¥¥';
        }
}
```

iskanji2

書　式

```
# include <ctype.h>

BOOL    iskanji2(c)
char    c ;
```

解　説

文字が漢字の第2バイトかどうかを調べます．（コードでは0x40から0x7e，0x80から0xfc）

戻り値

文字が漢字の第2バイトであったらTRUEを返します．そうでなかったらFALSEを返します．

参　照

「3.5　漢字処理の方法」
iskanji( )

注　意

この関数はマクロですので，cには副作用のあるものは指定できません．漢字かどうかは，調べようとしている文字列の先頭からすべての文字について漢字か，ANKかを見なければ正確には判りません．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
main(argc, argv)
int     argc;
char    *argv[];
{
        FILE    *fp = stdin;
        int     c;
        TINY    status;
        char    cold;

        if (--argc > 0)
                if ((fp = fopen(argv[1], "r")) == NULL) {
                        puts("File not found¥n");
                        exit(1);
                }
        status = 0;
        while ((c = fgetc(fp)) != EOF) {
                switch (status) {
                case    0:
                        if (iskanji((char)c)) {
                                status = 1;
                                cold = c;
                        } else
                                putchar((char)c);
                        break;
                case    1:
                        status = 0;
                        if (iskanji2((char)c))
                                printf("%c%c", cold, c);
                        else
                                printf("%02x %02x", (int)cold, c);
                        break;
                }
                putchar(' ');
        }
}
```

islower

書　式

```
#include <ctype.h>

BOOL    islower(c)
char    c;
```

解　説

文字が英小文字かどうかを調べます.

戻り値

文字が英小文字であったら TRUE を返します. そうでなかったら FALSE を返します.

参　照

isupper( ), tolower( ), toupper( )

注　意

この関数はマクロですので, c には副作用のあるものは指定できません.

例

```
#include <stdio.h>

main()
{
        int     c;
        /*  キーボードからの入力で小文字だけ出力する  */
        while ((c = getchar()) != EOF) {
                if (islower((char)c))
                        putchar((char)c);
        }
}
```

isspace

書　式

```
#include <ctype.h>

BOOL    isspace(c)
char    c;
```

解　説

文字がスペース文字(スペース，タブ，ラインフィード，ホーム，クリア，キャリジリターン)かどうかを調べます．(コードでは 0x20 と 0x09 から 0x0d の文字)

戻り値

文字がスペース文字であったら TRUE を返します．そうでなかったら FALSE を返します．

注　意

この関数はマクロですので，c には副作用のあるものは指定できません．

例

```
#include <stdio.h>

/*  スペース以外の文字までスキップする  */
char    *skipsp(p)
char    *p;
{
        while (isspace(*p))
                p++;
        return (p);
}

main()
{
        FILE    *fp;
        char    buf[256];

        if ((fp = fopen("head.c", "r")) == NULL) {
                puts("head.c not found¥n");
                exit(1);
        }
        while (fgets(buf, 256, fp))
                puts(skipsp(buf));
}
```

isupper

書　式

```
#include <ctype.h>

BOOL    isupper(c)
char    c ;
```

解　説

文字が英大文字かどうかを調べます.

戻り値

文字が英大文字であったらTRUEを返します. そうでなかったらFALSEを返します.

参　照

islower( ), tolower( ), toupper( )

注　意

この関数はマクロですので, cには副作用のあるものは指定できません.

例

```
#include <stdio.h>

main()
{
        int     c;
/*  キーボードからの入力で大文字だけ表示する  */
        while ((c = getchar()) != EOF) {
                if (isupper((char)c))
                        putchar((char)c);
        }
}
```

isxdigit

書　式

```
# include <ctype.h>

BOOL    isxdigit(c)
char    c ;
```

解　説

文字が 16 進数の文字("0"から"9", "a"から"f", "A"から"F")かどうかを調べます.

戻り値

文字が 16 進数の文字であったら TRUE を返します. そうでなかったら FALSE を返します.

参　照

isdigit( )

注　意

この関数はマクロですので, c には副作用のあるものは指定できません.

この関数は Ver.1.2 から標準ライブラリに追加されました.

例

```
#include <stdio.h>

main(argc, argv)
int     argc;
char    *argv[];
{
        int     i;
        char    c, *p;

        if (--argc < 1) {
                puts("Usage: hextodec <hexadecmal> ...¥n");
                exit(1);
        }
        argv++;
        /*  １６進数版atoi()をする  */
        while (argc--) {
                p = *argv;
                i = 0;
                while (isxdigit(*p)) {
                        c = *p;
                        if (isdigit(c))
                                c -= '0';
                        else
                                c = toupper(c) - 'A' + 10;
                        i = i * 16 + c;
                }
                printf("%d ", i);
        }
        putchar('¥n');
}
```

kbhit

書　式

```
# include <conio.h>

BOOL    kbhit( )
```

解　説

キーボードが押されたかどうかを調べます．この関数は標準入力のリダイレクトとは関係なく，キーボードバッファの状態を返します．

戻り値

キーボードバッファに文字がある場合にはTRUEを返します．ない場合にはFALSEを返します．

注　意

標準入力がファイルにリダイレクトされている場合には，1度TRUEになると標準入力からいくら文字を受け取ってもFALSEにはなりません．open( )で"con"をオープンし，read( )すればキーボードから入力された文字は受け取れます．

例

```
#include <stdio.h>
char    buf[256];

/*  キーボードバッファをクリアする  */
kill_buf()
{
        while (kbhit())         /*  keyboard buffer clear  */
                getch();
}

main()
{
        unsigned i;
        for (i = 0; i < 65535; i++)
                ;               /*  push any key while waiting here  */
        kill_buf();             /*  kill keyboard buffer  */
        gets(buf, 256);
        puts(buf);
}
```

longjmp

書　式

```
# include <setjmp.h>

VOID    longjmp(env, val)
jmp_buf env ;
int     val ;
```

解　説

longjmp( ) は，setjmp( ) で設定された jmp_buf 型のバッファ env をパラメータとすることで，関数を越えてジャンプし，env を設定した setjmp( ) の次(関数から値が返ってくるところ)に戻ることができます．またそのときに val で setjmp( ) の戻り値を渡すことができます．ただし，val は 0 以外の値でなければなりません．

戻り値

ありません．

参　照

setjmp( )

注　意

val が 0 の時には値が変更されます．

例

```
#include <stdio.h>
jmp_buf mbuf;

gofunc()
{
        int     err = 0;                /*  エラーフラグのクリア  */

        ...
        if (err)                        /*  エラーが発生したか？  */
                longjmp(mbuf, err);     /*  エラーが起きたら強制的に戻る  */
        ...
}

main()
{
        static  int     i;
        if (i = setjmp(mbuf)) {
                /*  longjmp()から戻ってきた  */
                printf("Return from longjmp() with code %d¥n", i);
        } else {  /*  最初のsetjmp()  */
                printf("setjmp() is executed¥n");
                gofunc();
        }
}
```

max

書　式

```
#include <stdlib.h>

int     max(x, y)
int     x, y ;
```

解　説

整数 x と y を比較して，小さくない方を返します．max( ) はふたつの数値の大きい方を選びたいときに使います．

戻り値

x と y で小さくない方の数値を返します．値が等しい時はその値を返します．

参　照

min( )

例

```
#include <stdio.h>

main()
{
/*  それぞれ数値の大きい方を表示  5, 2, 7 が表示される  */
printf("%d  %d  %d¥n", max(1, 5), max(-3, 2), max(7, 7));
}
```

memcpy

書　式

```
# include <memory.h>

VOID    memcpy(dest, source, length)
char    *dest, *source ;
size_t  length ;
```

解　説

メモリの source で示す位置から，length の長さだけ，dest で示す位置に内容をコピーします．strcpy( )などの文字列操作関数との違いは，"¥0"を操作の終了マークとして使わない点です．転送元(source)と転送先(dest)の領域が重なっているときにも，内容が壊されないように転送します．movmem( )とはパラメータの順番が違うだけで，動作は同じです．

戻り値

ありません．

参　照

movmem( )

注　意

MSX-DOS などのワークエリアを破壊しないように，注意して下さい．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>
#define CMDLIN  (char *)0x80
char    buf[128];

main()
{
/*  コマンドラインのコピーを作る  */
        memcpy(buf, CMDLIN, sizeof(buf));
        buf[(int)buf[0]+1] = '¥0';
        puts(buf + 1);
}
```

memset

書　式

```
# include <memory.h>

VOID    memset(dest, byte, length)
char    *dest, byte ;
size_t  length ;
```

解　説

dest で示されるメモリから length の長さだけ，byte の値で埋めます．length の単位はバイトです．setmem( ) とはパラメータの順番が違うだけで動作は同じです．

戻り値

ありません．

参　照

setmem( )

注　意

MSX-DOS などのワークエリアを破壊しないように，注意して下さい．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>
int     count[10];

main()
{
        int     c;
        int     count[10];
        int     other;

/*  配列bufを0で初期化する  */
        memset(count, (char)0, sizeof(count));
        other = 0;

        while ((c = getchar()) != EOF) {
                if (isdigit((char)c))
                        count[c - '0']++;
                else
                        other++;
        }
        for (c = 0; c < 10; c++)
                printf("%d '%d'(s)¥n", count[c], c);
}
```

min

書　式

```
# include <stdlib.h>

int     min(x, y)
int     x, y ;
```

解　説

整数 x と y の数値を比較して，大きくない方を返します．min( ) はふたつの数値の小さい方を選びたいときに使います．

戻り値

x と y で大きくない方の数値を返します．値が等しい時はその値を返します．

参　照

max( )

例

```
#include <stdio.h>

main()
{
/*  それぞれ小さい方を表示します  1, -3, 7が表示される  */
printf("%d  %d  %d¥n", min(1, 5), min(-3, 2), min(7, 7));
}
```

mkdir

書　式

```
# include <direct.h>

STATUS    mkdir(path)
char      *path ;
```

解　説

サブディレクトリを作成します．path はルートディレクトリからの指定でも，カレントディレクトリからの指定でもできます．ドライブを path 中に指定するとそのドライブにサブディレクトリが作成されます．([d：][¥]path の形で指定します．)

戻り値

サブディレクトリが作成された場合には OK を，失敗した場合には ERROR を返します．

参　照

chdir( )，rmdir( )，getcwd( )

注　意

mkdir( )では存在しないサブディレクトリの下にサブディレクトリを作成することはできません．作成したいサブディレクトリの親ディレクトリまでを mkdir( )によって作成して下さい．また，同じ名前のファイルが存在したときにも，作成できません．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
/*  サブディレクトリ¥binを作る  */
        if (chdir("¥¥bin") == ERROR) {
                puts("Making '¥¥bin' Subdirectory.¥n");
                mkdir("¥¥bin");
        }
/*  Now subdirectory '¥bin' must exist  */
}
```

movmem

書　式

```
# include <memory.h>

VOID    movmem(source, dest, length)
char    *dest, *source ;
size_t  length ;
```

解　説

メモリの source で示す位置から，length の長さだけ，dest で示す位置に内容をコピーします．strcpy( )などの文字列操作関数との違いは，"¥0"を操作の終了マークとして使わない点です．転送元(source)と転送先(dest)の領域が重なっているときにも，内容が壊されないように転送します．memcpy( )とはパラメータの順番が違うだけで，動作は同じです．

戻り値

ありません．

参　照

memcpy( )

注　意

MSX-DOS などのワークエリアを破壊しないように，注意して下さい．

例

```
#include <stdio.h>
#define CMDLIN  (char *)0x80
char    buf[128];

main()
{
/*  コマンドラインのコピーを作る  */
        movmem(CMDLIN, buf, sizeof(buf));
        buf[(int)buf[0]+1] = '¥0';
        puts(buf + 1);
}
```

open

書　式

```
# include <io.h>

FD      open(filename, mode)
char    *filename ;
int     mode ;
```

解　説

filename で示されたファイルをオープンします．filename にはドライブ，パス名を含めることができます．オープンすることで以後はそのファイルへの読み書きは戻り値のファイルハンドルで行えるようになります．mode にはオープンしようとしているファイルのオープンモードを指定します．ファイルのオープンモードには次のような値があります．

O_RDONLY	読み込みのみ ファイルがないときはエラーになります．
O_WRONLY	書き込みのみ ファイルがすでにあるなら，write( )したとき上書きされます．
O_RDWR	読み書き両方

戻り値

ファイルがないなどファイルのオープンに失敗すると ERROR を返します．そうでないときはファイルハンドルを返します．

参　照

close( )，creat( )，read( )，write( )

注　意

open( ) は低水準入出力関数です．

例

```
#include <stdio.h>
static  char    str[] = "test data";

main()
{
        FD      fd;
/*  ファイルを書き込みモードでオープンする  */
        if ((fd = open("openfile.dat", O_WRONLY)) == ERROR) {
                puts("File cannot open¥n");
                exit(1);
        }
        write(fd, str, strlen(str));
        close(fd);
}
```

outp

書　式

include <conio.h>

VOID　outp(port, val)
unsigned port ;
char　val ;

解　説

port で示される I/O ポートにデータ val, 1 バイトを出力します. ポート番号は 0 から 255 までの範囲で指定します.

戻り値

ありません.

参　照

inp( )

注　意

出力するポートによっては, MSX の誤動作の原因になりますので注意して下さい.

例

```
#include <stdio.h>

main()
{
        int     i;
        char    knj[32];
/*  漢字ＲＯＭパターンを読み込む  */
        outp(0xd8, 0);
        outp(0xd9, 1);
        for (i = 0; i < 32; i++)
                knj[i] = inp(0xd9);
}
```

printf

書　式

```
#include <stdio.h>

STATUS    printf(format[, arg1, arg2, ...])
char      *format;
```

解　説

printf( ) は，書式付きの変換を行ない，その出力を標準出力(stdout)に書き出します．引数 arg1, arg2, ... は format で示される制御文字列によって変換される値や，文字列へのポインタを渡します．制御文字列は，変換されずにそのまま出力される通常の文字と，次の形式の変換指定からなります．

```
%[-][[0]w][.n]c
```

変換指定はパーセント記号(%)で始まり，何番目に現れたかによって arg1, arg2 などの値との対応が取られます．マイナス記号(-)は，このフィールドが左寄せされることを示します(通常は右寄せ)．"w"は，フィールドの最低限の幅を 10 進数で示し，w が 0 で始まっていると，このフィールドの余白は，スペースの代わりに"0"で埋められます．".n"は，文字列変換(%s)の時のみ有効で，文字列のうち，出力される最大の文字数を 10 進数で指定します．文字 c は，次のうちのいずれかの変換文字です．

d　対応する引数を符号付き 10 進数として表示
u　対応する引数を符号なし 10 進数として表示
o　対応する引数を符号なし 8 進数として表示
x　対応する引数を符号なし 16 進数として表示
c　対応する引数を文字コードとして文字を表示
s　対応する引数を文字列へのポインタとして文字列を表示

上に示された変換文字でないものは，そのまま出力されます．また，"%%"という文字シーケンスによって，1 つのパーセント記号"%"を出力することができます．

戻り値

出力中にエラーがなければ OK が返されます．そうでなければ ERROR が返されます．

参　照

fprintf( ), sprintf( )

注　意

char 型の値を"%d", "%x", "%o"によって表示する場合には，int 型にキャストしたものを引数として渡して下さい．これは，呼び出し側は char 型(1 バイト値)で渡しますが，printf( ) は int 型で渡されたとするために，int 型の上位バイトが不定な値による表示を避けるためです．char 型だけでなく BOOL 型，STATUS 型，TINY 型の場合も同じ様にして下さい．"%c"で表示する際は問題ありません．

```
char    c;
printf("%d %02x¥n", (int)c, (int)'A');
```

printf( ) は可変パラメータ関数なので，次の宣言があらかじめ必要です．

```
STATUS  printf(.);
```

この宣言は，ヘッダファイル stdio.h に含まれています．

printf( ) は UNIX の標準 C のサブセットになっています．

例

```
#include <stdio.h>
int     val = 1234;
char    c = 'A';
static  char    str[] = "printf test string";

main()
{
/*  １０進，１６進，８進で数値の表示と文字，文字列を表示する  */
        printf("decimal %d¥nhexadecimal %04x¥noctal %06o", val, val, val);
        printf("char '%c'¥nstring '%s'¥n", c, str);
}
```

putc, putchar

書　式

```
# include <stdio.h>

STATUS  putc(c, fp)
char    c ;
FILE    *fp ;

STATUS  putchar(c)
char    c ;
```

解　説

putc( ) は fp で示されるファイルに 1 文字出力します. putchar( ) は標準出力(stdout)に 1 文字出力します. 文字 c が"¥n"でファイルがテキストモードの時には"¥r"と"¥n"の 2 文字にして出力し, ファイルが行バッファリングなら, バッファをフラッシュします.

戻り値

1 文字出力できたら OK を返します. そうでなかったら ERROR を返します.

参　照

fsetbin( ), fsettext( )

注　意

渡す文字は char 型なので getc( ) などの int 型で返されたものを出力する場合には必ず char 型にキャストして下さい.

例

```
#include <stdio.h>

main()
{
        int     c;
        FILE    *fp;
/*  キーボードの入力を標準出力とファイルに出力する  */
        if ((fp = fopen("file", "w")) == NULL) {
                puts("File cannot make¥n");
                exit(1);
        }
        while ((c = getchar()) != EOF) {
                putchar((char)c);       /*  標準出力に出力  */
                putc((char)c, fp);      /*  ファイルに出力  */
        }
}
```

putenv

書　式

```
# include <stdlib.h>

STATUS  putenv(env)             /*  set environment  */
char    *env ;                  /*  <varname>=<value>  */
```

解　説

MSX-DOS2 の環境変数の値を設定します．パラメータの文字列 env は「環境変数名＝値」の形である必要があります．値が省略されたときはその環境変数は消去されます．設定した環境の値は getenv( ) で獲得することができます．また，実行中のコマンドが終了しても環境の値は残っているので，SET コマンドで確認したり別のコマンドで使用することができます．

戻り値

環境が正しく設定できたときは OK を返します．そうでないときは ERROR を返します．

参　照

getenv( )

注　意

パラメータ env の示す文字列にイコール記号"="がないと環境変数は設定できません．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
/*  環境を設定後，獲得して値を検査する  */
        if (putenv("MYENVIRON=value of kankyou") == ERROR)
                puts("putenv Failed");
        if(strcmp(getenv("MYENVIRON"), "value of kankyou"))
                puts("Failed");
        else
                puts("OK");
}
```

puts

書　式

```
#include <stdio.h>

STATUS  puts(s)
char    *s;
```

解　説

標準出力(stdout)に文字列sを出力します．出力に対して"¥n"をつけたりはしません．文字列sはヌル文字で終っていなければなりません．また，ヌル文字は出力されません．文字列中に"¥n"があった場合にはテキストモードであれば"¥r"＋"¥n"にして出力します．

戻り値

出力中にエラーが発生した場合にはERRORが返されます．それ以外であったときはOKが返されます．

参　照

fgets( )，fputs( )，gets( )

注　意

MSX-Cのputs( )は，標準的なCのputs( )とは動作が異なります．標準的な動作のputs( )は文字列を出力した後"¥n"を出力しますが，MSX-Cのputs( )は出力しません．

例

```
#include <stdio.h>
char    buf[256];

main()
{
        FILE    *fp;
        /* ファイルnewgame.datの内容を表示する  */
        if ((fp = fopen("newgame.dat", "r")) == NULL) {
                puts("File not found¥n");
                exit(1);
        }
        while (fgets(buf, 256, fp))
                puts(buf, fp);
        fclose(fp);
}
```

qsort

書　式

```
#include <stdlib.h>

VOID    qsort(base, nel, width, compar)
char    *base ;
unsigned nel, width ;
int     (*compar)( ) ;
```

解　説

データを昇順に並べ換えます．baseはデータの先頭アドレス，nelはソートするデータの個数，widthはデータの各要素の大きさをバイト数で表したものです．

comparは，2つのポインタ(xとy)を引数として，次に示すようなint型の値を返す関数へのポインタです．

正	*x	>	*y
0	*x	=	*y
負	*x	<	*y

戻り値

ありません．

例

```
#include <stdio.h>
static  char    *device[8] = {"screen",      "keyboard",  "joystick",
                              "floppy disk", "hard disk", "mouse",
                              "cassette",    "RS-232C"};

int     compare(x, y)
char    **x, **y;
{
        return (strcmp(*x, *y));
}

/*  文字列の配列deviceをソート前とソート後を表示する  */
main()
{
        int     i;
        for (i = 0; i < 8; i++)
                printf("%s¥n", device[i]);
        puts("Sorting..¥n");
        qsort(device, 8, sizeof(char *), compare);
        for (i = 0; i < 8; i++)
                printf("%s¥n", device[i]);
}
```

read

書　式

```
#include <io.h>

int     read(fd, buf, bytes)
FD      fd;
char    *buf;
size_t  bytes;
```

解　説

ファイルハンドルfdで示されたファイルから，データを読み込みます．読み込むデータの長さはbytesで指定し，読み込んだデータはbufからの領域に格納されます．

戻り値

ファイルハンドルがオープンされていないもの，オープンした時のモードがライトモードである，ファイルがエンドオブファイルに達しているなどのどれかであったら，0が返されます．そうでなかったら実際に読み込まれたバイト数が返されます．ですから，bytesより小さい値である可能性もあります．

参　照

close( ), creat( ), open( ), write( )

注　意

read( )は低水準入出力関数です．

例

```
#include <stdio.h>
char    buf[1024];

main()
{
        FD      fd;
/*  ファイルreadtest.datを1024バイトごとにエンドオブファイルまで読む  */
        if ((fd = open("readtest.dat", O_RDONLY)) == ERROR) {
                puts("File not found¥n");
                exit(1);
        }
        while (read(fd, buf, sizeof(buf)))
                puts("Not EOF¥n");
        puts("Now  EOF¥n");
        close(fd);
}
```

rename

書　式

```
#include <io.h>

STATUS  rename(oldname, newname)
char    *oldname, *newname ;
```

解　説

oldnameで示されたファイルの名前をnewnameに変更します．oldnameでドライブ，パス名を指定できますが，newnameには指定しても無視され，ベースネームだけ使われます．oldnameにワイルドカードを指定したり，ファイルの移動はできません．

戻り値

ファイル名の変更ができればOKを返し，できなければERRORを返します．

例

```
#include <stdio.h>

main()
{
/*  ファイルcf.comをcfv120.comに名前を変更する  */
        rename("¥¥bin¥¥cf.com", "cfv120.com");
}
```

rmdir

書　式

```
# include <direct.h>

STATUS  rmdir(path)
char    *path ;
```

解　説

サブディレクトリを削除します．path はルートディレクトリからの指定でも，カレントディレクトリからの指定でもできます．ドライブを path 中に指定するとそのドライブのサブディレクトリが削除されます．([d：][¥]path の形で指定します．)しかし，削除しようとするディレクトリにファイルやディレクトリがあった場合には，削除することはできません．

戻り値

サブディレクトリが削除された場合には OK を，失敗した場合には ERROR を返します．

参　照

chdir( ), mkdir( ), getcwd( )

注　意

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
/*  サブディレクトリ¥binを削除する  */
        if (rmdir("¥¥bin") == ERROR) {
                puts("Erase all files in '¥¥bin' Subdirectory.¥n");
                unlink("¥¥bin¥*.*");
                if (rmdir("¥¥bin") == ERROR)
                    puts("'¥¥bin' has Subdirectory.  Cannot remove!!¥n");
        }
}
```

rsvstk

書　式

```
# include <malloc.h>

VOID    rsvstk(n)
size_t  n ;
```

解　説

スタック領域としてｎバイト確保します．ここでいうスタック領域とはsbrk( )で領域を確保する時の残りのスタックのための大きさを表します．スタック全体の大きさではないので注意して下さい．初期値としては1000バイト確保されています．詳しくは「4.3.3　メモリ管理関数」を参照して下さい．

戻り値

戻り値はありません．

参　照

alloc( ), free( ), sbrk( )

注　意

スタック領域の大きさはひとつの領域で管理しているので，メインプログラム以外で変更する場合は注意して下さい．

例

```
#include <stdio.h>

bigary()
{
        char    buf[1000];

        ...

}

main()
{
/*  大きな配列を使用するためにスタック領域を大きく確保する  */
        rsvstk(2500);
        bigary();
        rsvstk(1000);           /*  元に戻す  */
}
```

sbrk

書　式

```
#include <malloc.h>

char    *sbrk(n)
size_t  n;
```

解　説

メモリをnバイト割り当てます．sbrk()は低水準メモリ管理関数です．割り当てられた領域は解放することはできませんので，領域をプログラムで最後まで使用するとき使うといいでしょう．高水準メモリ管理関数alloc()との共用は可能ですが，連続した領域が取れにくくなるので，alloc()やfree()を多用しているプログラムでは効率が下がってしまいます．詳しくは「4.3.3　メモリ管理関数」を参照して下さい．

戻り値

領域が割り当てられた場合には，その領域の先頭へのポインタが返されます．メモリが足りないときはERRORが返されます．

参　照

alloc(), free(), rsvstk()

例

```
#include <stdio.h>

main()
{
        char    *p;
/*  バッファ用の領域を確保  */
        if ((p = sbrk(128)) != ERROR)
                setvbuf(stdin, p, _IOFBF, 128);
}
```

scanf

書　式

```
#include <stdio.h>

int     scanf(format[, arg1, arg2, ...])
char    *format;
```

解　説

scanf( )は標準入力(stdin)から書式変換付きの入力を行ないます．formatで与える制御文字列は，変換指定文字と入力ストリームと一致すべき文字の二者からなります．引数arg1，arg2,...は，変換の結果得られた値を格納するための変数へのポインタでなければなりません．

変換指定のフォーマットは次の通りです．

```
%[*]c
```

変換指定はパーセント記号(%)で始まります．アスタリスク(*)をパーセント記号の直後におくことによって，得られた値を代入せずに捨てることを指定できます．文字cは書式指定子で，次のうちのいずれかです．カッコの中は通常格納される変数の型です．

- d　入力文字列を符号つき10進数として変数に代入します．(int型)
- u　入力文字列を符号なし10進数として変数に代入します．(unsigned型)
- o　入力文字列を符号なし8進数として変数に代入します．(unsigned型)
- x　入力文字列を符号なし16進数として変数に代入します．(unsigned型)
- c　入力文字を1文字そのまま変数に代入します．(char型)
- s　入力文字列をスペースなどの区切り文字までを代入します．(char型の領域)

数値を代入するときには，入力文字列に数値を表す文字以外の文字までを数値として代入します．つまり，"%d"で符号つきの10進数を代入するときは0から9以外の文字が来るまで10進数として見ます．また，数値のオーバーフローは無視されます．

戻り値

scanf( )は，実際に代入された項目数を値として返します．戻り値が1で，変数へのポインタを3個パラメータに指定したら，2個目以降は値が代入されていないことになります．また，標準入力がエンドオブファイルになった時には値EOFが返されます．

参　照

fscanf( ), sscanf( )

注　意

"%c"で文字を代入するときはラインフィード文字"¥n"が代入されてしまうこともありますので，変換文字の指定には注意して下さい．scanf( )は可変パラメータ関数なので，次の宣言があらかじめ必要です．

```
int     scanf(.);
```

この宣言は，ヘッダファイルstdio.hに含まれています．

scanf( )はUNIXの標準Cのサブセットになっています．

例

```
#include <stdio.h>
char    buf[100];

main()
{
        int     val, n;
        char    c
        /*  キーボードから１０進数，文字，文字列を入力する  */
        n = scanf("%d %c %s", &val, &c, buf);
        printf("%d matched  decimal %d:char '%c':string '%s'¥n",
                n, val, c, buf)
}
```

sensebrk

書　式

```
# include <conio.h>

VOID    sensebrk( )
```

解　説

ブレイクチェックをします．この関数を実行中にCtrl+CまたはCtrl+STOPが押されているとコマンドレベルに戻ります．sensebrk( )以外の関数でもキーボードや画面に入出力する関数では，Ctrl+CまたはCtrl+STOPでコマンドレベルに戻ります．

戻り値

戻り値はありません

注　意

Ctrl+C，Ctrl+STOPでプログラムを中止した場合には，高水準入出力バッファはフラッシュされません．

例

```
#include <stdio.h>
char    buf[100];
int     ary[1000];

/*  整数のバブルソートをする．途中でCtrl+Cで止められる  */
VOID    bubble(array, n)
int     array[];
int     n;
{
        int     tmp;
        for (i = 0; i < n - 1; i++)
                for (j = i + 1; j < n; j++) {
                        sensebrk();             /*  break check  */
                        if (array[i] > array[j]) {
                                tmp = array[i];
                                array[i] = array[j];
                                array[j] = tmp;
                        }
                }
}

main()
{
        int     i, n = 0;

        while (gets(buf, 100)) {
                ary[n++] = atoi(buf);
        }
        for (i = 0; i < n; i++)
                printf("%6d%s", ary[i], ((i % 5 == 4) ? "¥n" : " "));
        buble(ary, n);
        for (i = 0; i < n; i++)
                printf("%6d%s", ary[i], ((i % 5 == 4) ? "¥n" : " "));
}
```

setbuf

書　式

```
# include <stdio.h>

VOID    setbuf(fp, buf)
FILE    *fp ;
char    *buf ;          /*  points allocated buffer(size BUFSIZ)  */
```

解　説

ファイルのバッファを設定します．setbuf( )ではバッファリングしないか，フルバッファリングをするの設定しかできません．細かい設定が必要なときはsetvbuf( )を使って下さい．

bufにはバッファへのポインタを指定します．そのバッファのサイズはBUFSIZである必要があります．bufがNULLのときにはバッファリングをしません(バッファリングなし)．

buf == NULLの時は

```
setvbuf(fp, NULL, _IONBF, 1);
```

buf != NULLの時は

```
setvbuf(fp, buf, _IOFBF, BUFSIZ);
```

の動作をします．

戻り値

ありません．

参　照

「4.3.1　B)高水準入出力関数のバッファリング」

setvbuf( )

注　意

setbuf( )はfopen( )でファイルをオープンしてから，データの入出力を一切していないときにのみ実行できます．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
/*  ファイルをバッファリングなしで使う  */
        if ((fp = fopen("test", "w")) == NULL) {
                puts("File cannot make¥n");
                exit(1);
        }
        setbuf(fp, NULL)    /*  バッファリングをしない  */
}
```

setjmp

書　式

```
# include <setjmp.h>

int     setjmp(env)
jmp_buf env ;
```

解　説

setjmp( ) は longjmp( ) と組み合わせて使われます．setjmp( ) は現在のスタックポインタ等を env に設定します．その後 longjmp( ) ではこの env を使って呼び出すと setjmp( ) で設定した値を元に戻し，setjmp( ) 直後に関数を越えてジャンプできます．これは関数の深いネスト中にエラーが発生して，上位のルーチンに直接制御を戻すために使うと有効でしょう．

戻り値

現在の状態を退避したときは 0 を返します．longjmp( ) 関数から戻ってきたときには longjmp ( ) の第 2 パラメータの数値を返します．この数値によって上位のルーチンはどういった原因で setjmp( ) から戻ってきたのかを知ることができます．

参　照

longjmp( )

注　意

setjmp( ) を使用している関数で，変数がレジスタに割り当てられている場合，longjmp( ) で戻ってきたとき変数の値は保存されていません．

例

```
#include <stdio.h>
jmp_buf mbuf;

gofunc()
{
        int     err = 0;                /*  エラーフラグのクリア  */

        ...
        if (err)                        /*  エラーが発生したか？  */
                longjmp(mbuf, err);     /*  エラーが起きたら強制的に戻る  */
        ...
}

main()
{

        static  int     i;
        if (i = setjmp(mbuf)) {
                /*  longjmp()から戻ってきた  */
                printf("Return from longjmp() with code %d¥n", i);
        } else {  /*  最初のsetjmp()  */
                printf("setjmp() is executed¥n");
                gofunc();
        }
}
```

setmem

書　式

```
# include <memory.h>

VOID    setmem(dest, length, byte)
char    *dest, byte ;
size_t  length ;
```

解　説

dest で示されるメモリから length の長さだけ，byte の値で埋めます．length の単位はバイトです．memset( ) とはパラメータの順番が違うだけで動作は同じです．

戻り値

ありません．

参　照

memset( )

注　意

MSX-DOS などのワークエリアを破壊しないように，注意して下さい．

例

```
#include <stdio.h>
int     count[10];

main()
{
        int     c;
        int     count[10];
        int     other;

/*  配列bufを0で初期化する  */
        setmem(count, sizeof(count), (char)0);
        other = 0;

        while ((c = getchar()) != EOF) {
                if (isdigit((char)c))
                        count[c - '0']++;
                else
                        other++;
        }
        for (c = 0; c < 10; c++)
                printf("%d '%d'(s)¥n", count[c], c);
}
```

setvbuf

書　式

```
# include <stdio.h>

STATUS  setvbuf(fp, buf, mode, size)
FILE    *fp ;
char    *buf ;
int     mode, size ;
```

解　説

fp で示されるファイルのバッファについてのパラメータを設定します．設定できるものはバッファ領域，バッファサイズ，バッファリングの方法です．

buf にはバッファへのポインタを渡します．NULL が指定されていたときはシステム側が自動的に割り当てます．(fclose( ) で解放されます．)mode が_IONBF であった場合には無視されます．

size にはバッファのサイズを指定します．fopen( ) でバッファサイズを指定しなかったときにはバッファサイズは BUFSIZ(1024)になっています．mode が_IONBF であった場合には無視されます．

mode には次の 3 つのバッファリングのモードのうちどれかを指定します．

(1)_IONBF：バッファリングしません．1 文字ごとにディスクに読み書きします．一般にデバイスに対して指定します．

(2)_IOLBF：行バッファリングをします．出力時には"¥n"がくるとバッファのフラッシュをします．

(3)_IOFBF：フルバッファリングをします．バッファがいっぱいになるまでフラッシュはしません．fopen( ) した直後はこのモードになっています．

戻り値

mode が上記以外の値である時 ERROR を返します．それ以外では OK を返します．

参　照

「4.3.1　B)高水準入出力関数のバッファリング」

fflush( )，fopen( )，setbuf( )

注　意

setvbuf( ) は fopen( ) でファイルをオープンしてから，データの入出力を一切していないときにのみ実行できます．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main()
{
        FILE    *fp;
/*  ファイルを128バイトのバッファで使う  */
        if ((fp = fopen("test", "w")) == NULL) {
                puts("File cannot make¥n");
                exit(1);
        }
        setvbuf(fp, NULL, _IOFBF, 128);
        /*  バッファを128バイトにします  */
}
```

sprintf

書　式

```
#include <string.h>

STATUS  sprintf(buffer, format[, arg1, arg2, ...])
char    *buffer, *format;
```

解　説

sprintf( )は，書式付きの変換を行ない，その出力を buffer で示される領域に格納します．制御文字列については，printf( )と同じですのでその項を参照して下さい．

戻り値

出力中にエラーがなければ OK が返されます．そうでなければ ERROR が返されます．

参　照

fprintf( )，printf( )

注　意

printf( )は可変パラメータ関数なので，次の宣言があらかじめ必要です．

```
STATUS  sprintf(.);
```

この宣言は，ヘッダファイル string.h に含まれています．

sprintf( )は UNIX の標準 C のサブセットになっています．

例

```
#include <stdio.h>

/*  年月日を文字列にする  */
char    *strdate(buf, y, m, d)
char    *buf;                   /*11バイト以上の領域へのポインタ*/
int     y, m, d;
{
        sprintf(buf, "%04d/%02d/%02d", y, m, d);
        return (buf);
}

main(argc, argv)
int     argc;
char    *argv[];
{
        char    buf[11];

        if (--argc < 3) {
                puts("Usage: strdate <year> <month> <date>¥n");
                exit(1);
        }
        strdate(buf, atoi(argv[1]), atoi(argv[2]), atoi(argv[3]));
        puts(buf);
        putchar('¥n');
}
```

sscanf

書　式

```
# inlcude <string.h>

int       sscanf(line, format[, arg1, arg2, ...])
char      *line, *format ;
```

解　説

sscanf( ) は scanf( ) と同様に書式付き入力を行ないますが，scanf( ) が標準入力(stdin)から入力を読み込むのに対し，line で指される文字列から入力を取る点が異なります．line で示される文字列はヌル文字(¥0)で終わらなければなりません．ヌル文字自身は入力されません．format 以降のパラメータは scanf( ) と同じなのでその項を参照して下さい．

戻り値

sscanf( ) は，実際に代入された項目数を値として返します．戻り値が 1 で，変数へのポインタを 3 個パラメータに指定したら，2 個目以降は値が代入されていないことになります．また，文字列 line の終わりに到達した時には値 EOF が返されます．

参　照

fscanf( ), scanf( )

注　意

sscanf( ) は可変パラメータ関数なので，次の宣言があらかじめ必要です．

```
int       sscanf(.) ;
```

この宣言はヘッダファイル string.h に含まれています．

sscanf( ) は UNIX の標準 C のサブセットになっています．

例

```
#include <stdio.h>

main()
{
        int     y, m, d;
        int     n;
/*  年月日を文字列から得る  */
        n = sscanf("1989/1/17", "%d/%d/%d", &y, &m, &d);
        printf("%2d/%2d/%4d¥n", m, d, y);
}
```

strcat

書　式

```
# include <string.h>

char    *strcat(d, s)
char    *d, *s ;
```

解　説

文字列 d の後ろに文字列 s を追加します．このとき，s を追加するのに必要な領域はあるものとして実行されます．また，文字列 d も¥0 で終わっていなければなりません．

戻り値

d がそのまま返されます．

参　照

strcpy( )，strncat( )

例

```
#include <stdio.h>
char    fname[64];

/*  ドライブ，パス，ベースネーム，拡張子を与えてフルパス名を作る  */
fullpath(dst, drv, path, base, ext)
char    *dst, *drv, *path, *base, *ext;
{
        strcpy(dst, drv);
        strcat(dst, path);
        strcat(dst, "¥¥");
        strcat(dst, base);
        strcat(dst, ".");
        strcat(dst, ext);
}

main(argc, argv)
int     argc;
char    *argv[];
{
        switch (argc) {
        case    1:
                puts("Usage: fullpath [<d:>] [<path>] <base> [<ext>]
                exit(1);
        case    2:
                fullpath(fname, "a:", "¥tmp", argv[1], "$$$");
                break;
        case    3:
                fullpath(fname, "a:", "¥tmp", argv[1], argv[2]);
                break;
        case    4:
                fullpath(fname, "a:", argv[1], argv[2], argv[3]);
                break;
        case    5:
                fullpath(fname, argv[1], argv[2], argv[3], argv[4]);
                break;
        }
        puts(fname);
        putchar('¥n');
}
```

strchr

書　式

```
# include <string.h>

char    *strchr(s, c)
char    *s, c ;
```

解　説

文字列 s の中から文字 c の最初の位置を探します．c は"¥0"でも構いません．

戻り値

文字 c が見つかったときはその位置を返します．そうでない場合は NULL を返します．

注　意

漢字の文字列には対応していません．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main(argc, argv)
int     argc;
char    *argv[];
{
        int     n = 0;
        char    *s, *head;

        if (--argc < 1) {
                puts("Usage: ndot <string> ...¥n");
                exit(1);
        }
        argv++;
        while (argc--) {
/*  文字列中にいくつのピリオドが含まれているかを表示する  */
                head = s = *argv++;
                while (s = strchr(s, '.')) {
                        n++;
                        s++;
                }
                printf("%s includes %d '.'(s)¥n", head, n);
        }
}
```

strcmp

書　式

```
# include <string.h>

int     strcmp(s, t)
char    *s, *t ;
```

解　説

文字列 s と文字列 t を比較します．文字列の比較はふたつの文字列の先頭から，1 文字ずつ比較していきます．ともに文字列が終わるか，文字が異なったところで比較は終了します．

戻り値

比較して，同じ文字列であったら 0 が返されます．異なっていたときは，値は決っていませんが，範囲でどの様な状態かを知ることができます．

正	文字列 s	＞	文字列 t
0	文字列 s	＝	文字列 t
負	文字列 s	＜	文字列 t

参　照

strncmp( )

注　意

ふたつの文字列が同じであった時に，0 が返されます．

例

```
#include <stdio.h>
BOOL    debug = FALSE;                  /*  デバッグモードのクリア  */

main(argc, argv)
int      argc;
char    *argv[];
{
        /*  最初のパラメータが"-d"だったらデバッグモードにする  */
        if (argc > 1 && strcmp(argv[1], "-d") == 0) {
                debug = TRUE;
                argc--;
                argv++;
        }
        ...
}
```

strcpy

書　式

```
#include <string.h>

char    *strcpy(d, s)
char    *d, *s;
```

解　説

文字列sをdの示す領域にコピーします．dにはsの"¥0"までコピーします．このとき，dはsが入るだけの十分な領域があるとします．

戻り値

dをそのまま返します．

参　照

strcat(), strncpy()

例

```
#include <stdio.h>
char    fname[64];

main(argc, argv)
int     argc;
char    *argv[];
{
        if (--argc < 1) {
                puts("Usage: apndext <basename>¥n");
                exit(1);
        }
        strcpy(fname, argv[1]);
        strcat(fname, ".c");
        puts(fname);
        putchar('¥n');
}
```

strlen

書　式

```
# include <string.h>

size_t   strlen(s)
char     *s ;
```

解　説

文字列 s の長さを調べます．この長さには最後の"¥0"は含まれません．

戻り値

文字列 s の長さを返します．

例

```
#include <stdio.h>
char    fname[64];

/*  文字列の最後が'¥'でなかったら'¥'を追加する  */
adddelimit(s)
char    *s;
{
        size_t  len;
        len = strlen(s);
        if (len != 0)
                if (s[len - 1] != '¥¥') {
                        s[len] = '¥¥';
                        s[len + 1] = '¥0';
                }
}

main(argc, argv)
int     argc;
char    *argv[];
{
        if (--argc < 1) {
                puts("Usage: specpath <path>¥n");
                exit(1);
        }
        strcpy(fname, argv[1]);
        adddelimit(fname);
        strcat("temp.$$$");
        puts(fname);
        putchar('¥n');
}
```

strlwr

書　式

```
#include <string.h>

char    *strlwr(s)
char    *s;
```

解　説

文字列sの英小文字を英大文字に変換します．この関数はsで示された文字列を直接変更します．

戻り値

sをそのまま返します．

参　照

strupr( )

注　意

漢字の文字列には対応していません．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

main(argc, argv)
int     argc;
char    *argv[];
{
        FILE    *fp;
        if ((fp = fopen(argv[1], "r")) == NULL) {
                /*  小文字でコマンド名，ファイル名を出力する  */
                fprintf(stderr, "%s: %s not found¥n",
                        strlwr(argv[0]), argv[1]);
                exit(1);
        }
        ...
}
```

strncat

書　式

```
# include <string.h>

char     *strncat(d, s, n)
char     *d, *s ;
unsigned n ;
```

解　説

文字列 d の後ろに文字列 s の先頭から最大 n 文字を追加し，その後に"¥0"を付加します．s を追加するための領域は少なくとも n 文字はあるものとします．n が 0 のときは何も追加されません．

戻り値

d をそのまま返します．

参　照

strcat( )

注　意

漢字の文字列には対応していません．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>
char    fname[64];

/*  拡張子を'.'を含めて4文字以下で追加する  */
addext(ext)
char    *ext;
{
        strncat(fname, ext, 4);
}

main(argc, argv)
int     argc;
char    *argv[];
{
        if (--argc < 1) {
                puts("Usage: addext <extension>¥n");
                exit(1);
        }
        strcpy(fname, "temp.");
        strncat(fname, argv[1], 4);
        puts(fname);
        putchar('¥n');
}
```

strncmp

書　式

```
# include <string.h>

int       strncmp(s, t, n)
char      *s, *t ;
unsigned n ;
```

解　説

文字列 s と t を先頭から最大 n 文字比較します．文字列の比較はふたつの文字列の先頭から，1 文字ずつ比較していきます．n 文字比較する，ともに文字列が終わる，文字が異なるのどれかが起きたところで比較は終了します．

戻り値

先頭の n 文字が同じ文字列であったり，n 文字未満でもともに文字列が終るまで比較して同じであったら 0 が返されます．異なっていたときは，値は決っていませんが，範囲でどの様な状態かを知ることができます．

正	文字列 s	＞	文字列 t
0	文字列 s	＝	文字列 t
負	文字列 s	＜	文字列 t

参　照

strcmp( )

注　意

漢字の文字列には対応していません．

ふたつの文字列の先頭からの n 文字以下で同じであった時に，0 が返されます．

この関数は Ver.1.2 から標準ライブラリに追加されました．

例

```
#include <stdio.h>

/*  文字列の先頭が"temp"かどうか調べる  */
istmp(fn)
char    *fn;
{
        return (!strncmp(fn, "temp", 4));
}

main(argc, argv)
int     argc;
char    *argv[];
{
        if (--argc < 1) {
                puts("Usage: istmp <dst>¥n");
                exit(1);
        }
        if (istmp(argv[1])) {
                puts("Don't specify 'temp????' file¥n");
                exit(1);
        }
        puts(argv[1]);
}
```

strncpy

書　式

```
# include <string.h>

char    *strncpy(d, s, n)
char    *d, *s ;
unsigned n ;
```

解　説

文字列ｓの先頭から最大ｎ文字をｄにコピーします．ｄにはｎ文字の領域があるものとします．

戻り値

ｄをそのまま返します．

参　照

strcpy( )

注　意

漢字の文字列には対応していません．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>
#define SIZE    20
char    shell[SIZE];

main()
{
/*  環境変数SHELLの値を配列shellに入れる  */
        if ((p = getenv("SHELL")) != NULL) {
                strncpy(shell, p, SIZE);
                free(p);
        }
        puts(shell);
}
```

strupr

書　式

```
#include <string.h>

char    *strupr(s)
char    *s;
```

解　説

文字列sの英小文字を英大文字に変換します．この関数はsで示された文字列を直接変更します．

戻り値

sをそのまま返します．

参　照

strlwr( )

注　意

漢字の文字列には対応していません．

この関数はVer.1.2から標準ライブラリに追加されました．

例

```
#include <stdio.h>

/*  引数をすべて大文字にして表示する  */
main(argc, argv)
int     argc;
char    *argv[];
{
        int     i;
        for (i = 0; i < argc; i++)
                printf("[%d]: %s¥n", i, strupr(argv[i]));
}
```

tolower

書　式

```
# include <ctype.h>

char    tolower(c)
char    c ;
```

解　説

c が英大文字だったら英小文字にした文字を返します.

戻り値

c が英大文字だったら英小文字にした文字を返します. そうでなかったら, c そのものを返します.

参　照

islower( ), isupper( ), toupper( )

注　意

MSX-C では tolower( ) は関数です.

例

```
#include <stdio.h>

/*  文字列sの大文字は小文字に，小文字は大文字に変換する  */
char    *strchg(s)
char    *s;
{
        char    head = s;

        while (*s) {
                if (isupper(*s))
                        *s = tolower(*s);
                else if (islower(*s))
                        *s = toupper(*s);
                s++
        }
        return (head);
}

main(argc, argv)
int     argc;
char    *argv[];
{
        if (--argc < 1) {
                puts("Usage: strchg <string> ...¥n");
                exit(1);
        }
        argv++;
        while (argc--) {
                puts(*argv);
                puts(strchg(*argv));
        }
}
```

toupper

書　式

```
# include <ctype.h>

char    toupper(c)
char    c ;
```

解　説

c が英小文字だったら英大文字にした文字を返します.

戻り値

c が英小文字だったら英大文字にした文字を返します. そうでなかったら, c そのものを返します.

参　照

islower( ), isupper( ), tolower( )

注　意

MSX-C では toupper( ) は関数です.

例

```
#include <stdio.h>
int     x, y;
/*  入力待ちをし，'H','J','K','L'のどれかだったら位置を変更  */
char    mvcmd()
{
        char    c;
        c = getch();
        switch (toupper(c)) {
        case    'H':
                x--;
                break;
        case    'J':
                y++;
                break;
        case    'K':
                y--;
                break;
        case    'L':
                x++;
                break;
        default:
                break;
        }
        return (c);
}

main()
{
        x = y = 0;

        puts("push key one of  'H', 'J', 'K', 'L', 'Q'¥n");
        while (mvcmd() != 'Q') {
                printf("x = %5d   y = %5d¥n", x, y);
        }
}
```

ungetc, ungetch

書　式

```
# include <stdio.h>

STATUS  ungetc(c, fp)
char    c ;
FILE    *fp ;

VOID    ungetch(c)
char    c ;
```

解　説

ungetc( ) は fp で示されるファイルに 1 文字戻します．ungetch( ) は標準入力に 1 文字戻します．戻した文字はそれぞれ次に実行される getc( )，getchar( ) で受け取ることができます．この関数は入力ファイルの先読みによく使われます．

戻り値

ファイルがライトモードでオープンされていた場合と文字を戻す場所がないときは ERROR が返されます．文字が戻された時は OK が返されます．

例

```
#include <stdio.h>

main(argc, argv)
int     argc;
char    *argv[];
{
        int     c;
        FILE    *fp;

        if (--argc < 1) {
                puts("Usage: countsp <file>¥n");
                exit(1);
        }
        if ((fp = fopen(argv[1], "r") == NULL) {
                puts("File not found¥n");
                exit(1);
        }
/*  非スペース文字，スペース文字の繰り返し回数をEOFまで表示する  */
        while ((c = getc(fp)) != EOF) {
                ungetc((char)c, fp);
                count = 0;
                while ((c = getc(fp) != ' ' && c != '¥t')
                        count++;
                if (!feof(fp))
                        ungetc((char)c, fp);
                printf("%d no_white_spaces¥n", count);
                count = 0;
                while ((c = getc(fp) == ' ' || c == '¥t')
                        count++;
                printf("%d white_spaces¥n", count);
                if (!feof(fp))
                        ungetc((char)c, fp);
        }
}
```

unlink

書　式

```
# include <io.h>

STATUS  unlink(filename)
char    *filename ;
```

解　説

filename で指定されたファイルを削除します．filename にはドライブ名，パス名，ワイルドカードを含めることもできます．

戻り値

指定したファイルがなかったり，削除に失敗すると ERROR を返します．そうでない場合は OK を返します．

例

```
#include <stdio.h>

main()
{
        puts("Delete msxc*.$$$ Ok ?");
        if (getch() == 'y')
/*  テンポラリファイルを削除する  */
                unlink("¥¥tmp¥¥msxc*.$$$");
}
```

write

書　式

```
#include <io.h>

int     write(fd, buf, bytes)
FD      fd;
char    *buf;
size_t  bytes;
```

解　説

ファイルハンドルfdで示されたファイルへ，データを書き込みます．書き込むデータはbufから始まり，長さはbytesで指定します．

戻り値

ファイルハンドルがオープンされていないものであったり，オープンした時のモードがリードモードであったら，0が返されます．そうでなかったらbytesがそのまま返されます．

参　照

close(), creat(), open(), read()

注　意

write()は低水準入出力関数です．

例

```
#include <stdio.h>
char    buf[100];

main()
{
        char    *p, c;
        FD      fd;
        /*  スペースから100文字をファイルに書き込む  */
        for (c = ' ', p = buf; c < ' ' + 100; c++)
                *p++ = c;
        if ((fd = open("test.dat", O_WRONLY)) == ERROR) {
                puts("File cannot make¥n");
                exit(1);
        }
        write(fd, buf, 100);
}
```

第7章　コマンドリファレンス

7.1 CF

目　的

構文解析と文法のチェックを行います(パーサ)．また，その結果を中間言語ファイル(.tco ファイル)として出力します．

構　文

CF ［オプション］ ファイル名

オプション一覧

オプション	機　能
-c	コメントのネストを禁止する．
-e[filename]	エラーメッセージをファイルに出力する．filename でファイル名を指定できる．
-f	関数とパラメータに対する暗黙の宣言を行う．
-j	文字列中の漢字を正しく認識する．
-m	コンパイラのワーク用テーブルの P：S：H の割合を表示する．
-o[filename]	中間言語ファイルのファイル名を指定する．
-rP：S：H	コンパイラのワーク用テーブルを P：S：H の比率で割当てる
-s	ソースファイルのエラーが検知されても，バッチファイルの実行を中断しない．
-t	ポインタと整数の間の自動型変換を可能にする．

参照ページ

26 ページ「2.2.1　CF　(パーサ)」

7.2 CG

目 的

CF が出力した中間言語をアセンブリ言語に変換します(コードジェネレータ).

構 文

CG [オプション] ファイル名

オプション一覧

オプション	機 能
-k	.tco ファイルを自動的に削除する.
-l	グローバルシンボル名全体を有効にする.
-o[filename]	出力ファイルの名前を指定する.
-rN	シンボルテーブル領域として, N バイト確保する. ただし, N は 10 進数.
-u	処理の進行状況を表示しない.

参照ページ

36 ページ「2.2.3 CG (コードジェネレータ)」

7.3 FPC

目　的

関数の引数の整合性をチェックします(パラメータチェッカ).

構　文

FPC ［オプション］ ファイル名１　ファイル名２　・・・

オプション一覧

オプション	機　能
-c	.tco ファイルの連結
-d	don't care 関数の指定
-i	間接呼び出しに対して警告を出す.
-s	エラーが検出されてもバッチファイルの実行を中断しない.
-t	int, unsigned, ポインタを別の型として扱う.
-u	未定義関数に対する参照をチェックしない.

参照ページ

32 ページ「2.2.2　FPC　(パラメータチェッカ)」

7.4 MX

目 的

ライブラリの作成，保守を支援します．ソースファイルのモジュール(関数単位)への分割，ライブラリファイル作成までの手順の出力をします．(ライブラリ保守支援ツール)

構 文

MX ［オプション］ ファイル名 ［モジュール名１ モジュール名２ ･･･］

オプション一覧

オプション	機 能
-l	LIB80 を呼び出してライブラリファイルを作成する手順も出力する．
-o[path¥]	モジュールをディレクトリ path に抽出する．

参照ページ

93 ページ「4.5.2 ライブラリ保守支援ツール MX について」

第8章　エラーメッセージ一覧

この章ではMSX-Cコンパイラが発生させるエラーメッセージの一覧を示します．エラーメッセージ（英文）とその日本語訳とその主な内容を示しています．実際コンパイルでエラーが起きたときにご利用下さい．

メッセージの説明の順番は，コマンドごとに記号を無視して，アルファベット順に配列しています．

8.1　CFのエラーメッセージ

array of function

関数の配列を作っています．

要素が関数になるような配列は作ることはできません．

bad #???

#コマンドの使用法が違います．

#コマンドのある場所が式中であったりプリプロセッサにないコマンドが指定されています．

bad abstract declarator

抽象宣言子が正しくありません．

抽象宣言子はキャストの型指定で使われます．

bad assignment

代入が間違っています．

代入先が変数などを含む式になっていると発生します．

bad cast

間違ったキャストを行っています．

配列や構造体，共用体にキャストしています．キャストは数値やポインタの属性を変更するもので配列や構造体，共用体にキャストすることはできません．

bad character

不適当な文字があります.

コントロール文字がソースファイルにあったり，行末以外に'¥'があります.

bad condition

条件が不正です.

if,while などの条件判断に使われる条件に合わない表記があります.

bad conditional expression (missing '?')

条件式（3 項演算子）の使い方が違います.

条件式で'?'がない場合に発生します.

bad constant expression

定数式が不正です.

配列定義時の要素数の指定など，定数が必要なところに変数や関数があります.

bad # define statement

define の使用法が違います.

識別子として正しくない文字が含まれています.

bad element

文が存在していないか不正です.

ラベルの後や，while などの後ろには文が必要ですが，それが見つかりません.

bad element encountered externally

関数外での文が不正です.

関数の外側で文が正しくないとこのエラーが発生します．関数中であった場合には"bad element"エラーが発生します.

bad # ifdef statement

ifdef の使用法が違います.

識別子として正しくない文字が含まれています.

bad # ifndef statement

ifndef の使用法が違います.

識別子として正しくない文字が含まれています.

bad # include statement

include の使用法が違います.

ファイルの指定には<,>か " で囲まなければなりません.

bad indirection

間接指定が間違っています.

このエラーは間接指定の" * "をポインタにつけすぎたとき発生します. つまり, 普通のポインタであるのに 2 つ以上つけると起きます.

bad number

数値の範囲外の文字が使われています.

数値は 8 進数, 10 進数, 16 進数が使用できますが, それぞれ 0 から 7, 0 から 9, 0 から 9 と a から f が範囲です.

bad option : **<文字>**

存在しないスイッチ"-<文字>"が指定されています.

指定できるスイッチは, ceftjmors の各文字です.

bad parameter list

引数リストが間違っています.

引数つきマクロの定義中, 仮引数名が不正か, 仮引数の区切りが","でないものが使われています.

bad parameter type

引数の型が数値型ではありません.

関数の呼び出しの際に, 引数が配列, 構造体, 共用体で呼び出されています. 直接渡すことができないので, 普通は渡すものの先頭へのポインタで代用します.

bad parameter type '＜仮引数＞'

＜仮引数＞の型が数値型ではありません.

関数の定義の際, 仮引数が関数, 構造体, 共用体として宣言されています. 直接受け取れないので, 普通は渡されるものの先頭へのポインタで代用します.

bad # pragma statement

pragma の使用法が違います.

パラメータが予約語ではありません.

bad storage class

記憶クラスの指定が間違っています.

関数定義時に extern をつけていたり, 関数宣言時に static をつけています.

bad switch expression

switch で使われる式が不正です.

switch の式は数値型の結果が必要です.

bad table ratio

"-r"スイッチで指定された比率がよくありません.

比率には正の数を指定して下さい.

bad type in cast

キャストする型が不正です.

キャストする型がキャストできないもの（関数など）になっています.

bad # undef statement

undef の使用法が違います.

識別子として正しくない文字が含まれています.

'break' outside switch/break

switch の外側に break があります.

break をループや switch などのブロックの外側で使っています.

cannot initialized

初期化できません.

自動（auto）変数の配列は初期化できません.

cannot make ［<file>］

<file>が作成できません.

.tco ファイルや.dia ファイルを作成するときに，ディレクトリやディスク容量がいっぱいであったり，ディレクトリが存在しないなど，ファイルを作成することができないときに表示されます.

cannot open ［<file>］

<file>が見つかりません.

指定したＣのソースファイルかインクルードファイルが見つからないときに表示されます.インクルードファイルは環境変数 INCLUDE によってデフォルトディレクトリを設定します.

'case' after default

default より後ろに case があります.

MSX-C では default より後ろに case を書くことはできません.

'case' outside switch

switch の外側に case があります.

case は switch の内部でなければ使用できません.

conflict definition '<識別子>'

<識別子>の宣言が一致しません.

2 つ以上の外部・前方参照の宣言で，関数や変数の宣言が一致していません．関数の前方参照の宣言と定義とが違うことが考えられます.

conflicting number of macro parameter

マクロの引数の数が合っていません.

引数付きのマクロでは，定義と参照の引数の数は同じでなければなりません.

'continue' outside loop

while，for などループの外側に continue があります.

ループの外側では continue は使えません.

'default' appeard twice

switch のブロック内に default が 2 回以上出現しました.

default はその他を示すラベルですので，1 回しか使えません.

'default' outside switch

switch の外側に default があります.

default は switch のブロック内でないとラベルとして使えません.

disk full

ディスクがいっぱいです.

データの書き込み中にディスクがいっぱいになりました．処理が続行できません.

'double' not supported

double はサポートされていません

duplicate 'case' label

switch で case の定数値が同じものが 2 つ以上あります.

case の後の定数はすべて違う値でなければなりません.

duplicate label '＜ラベル＞'

同じ名前の＜ラベル＞が 2 つ以上あります.

ラベル名は，関数の中で同じ名前をつけることはできません.

duplicate member

構造体や共用体の中に同じメンバ名があります.

ひとつの構造体，共用体の中には同じメンバ名は使えません.

duplicate storage class

記憶クラスが 2 つ以上指定されています.

extern と static など記憶クラスを指定する語が 2 つ以上使われています.

duplicate tag '＜タグ＞'

同じ＜タグ＞で構造体や共用体が定義されています.

タグ名はおのおのの構造体や共用体で，別の名前をつけなければなりません.

duplicate type specifier

変数の型の指定が 2 つ以上あります.

int と char など型を指定する語が 2 つ以上あります.

'(' expected

"("がありません.

if 文の条件式の前など，構文上"("が必要なとき発生します.

')' expected

")"がありません.

if 文の条件式の後ろなど，構文上")"が必要なとき発生します．また，抽象宣言子で")"がないときも発生します.

' : ' expected

"："がありません.

case 文の定数の後ろや default の後ろに"："がないときこのメッセージが表示されます.

';' expected

";"がありません.

ひとつの文は";"で終了しなければなりません. ";"がないのに次の文（制御文）が来ています.

']' expected

"]"がありません.

配列定義時に要素数を指定する数値の後ろに"]"がありません.

'}' expected

"}"がありません.

配列の初期化をした場合, "{"に対応するように"}"を置かなければなりません.

'float' not supported

float はサポートされていません

function cannot appear

関数は指定できません.

関数を構造体メンバとして使用した場合などに発生します.

function expected

関数の定義が必要です.

関数の定義をすべきところで別の文字列などをおいた場合に発生します.

function returns structured data

関数の戻り値の宣言が構造体または共用体になっています.

関数は構造体や共用体を返すことはできません.

hash table over flow

ハッシュテーブルがオーバーフローしました.

識別子のハッシュ値を格納する場所が不足しました. "-r"スイッチでワークテーブルの比率を修正して下さい.「2.2.1 CF(パーサ)」の"-r"スイッチの項を参照して下さい.

heap over flow

ヒープがオーバーフローしました.

ファイルのインクルードが多かったり，auto 変数を多量に宣言すると発生します．また，式等が複雑な場合も発生します.

improper function mode

不適当なファンクションモードです.

nonrec,recursive が変数に対して使われています.

include too nested

include のネストが深すぎます.

MSX-C ではインクルードファイルのネストは 4 重までとなっています．それ以上使っている場合にはインクルードファイルを分割するなどして下さい.

known dimension expected

配列の大きさを定義していません.

配列の大きさは定数で指定しなければなりません.

'long' not supported

long はサポートされていません

l-value required

左辺値が必要です.

記憶位置を示す式を左辺値と呼びます．値を代入するには変数の名前など，記憶位置を示す式が必要です.

missing ')'

")"がありません.

式中で，")"が不足しています.

missing ':'

":"がありません.

条件式（3項演算子）"?:"の":"がありません.

missing ']'

"]"がありません.

式中で，"]"が不足しています.

missing '}'

"}"がありません.

式中で，"}"が不足しています.

missing condition

条件がありません.

条件判断に使う条件がありません.

missing identifier

識別子がありません.

識別子とは，変数名，関数名，配列名などのことです．識別子がないのに，文，式が書かれています.

missing member name

メンバ名がありません.

構造体変数や構造体変数へのポインタに続くメンバ名がありません.

missing operator (or semicolon)

演算子か";"がありません.

空白で区切られた数値や変数名が複数続いている可能性があります.

missing quote

文字列が(")で終わっていません.

文字列の複数行にわたる記述で，行末に"¥"をおかずに改行した場合発生します.

missing tag

タグ名がありません.

構造体変数の定義がなく，タグ名の指定がない場合に発生します.

not appear in parameter list '<仮引数>'

"<仮引数>"は引数リストにありません.

引数リストにない仮引数が引数の宣言に含まれています.

parameter cannot appear here

パラメータの指定はできません.

関数の宣言でパラメータを指定している場合発生します.

pointer type mismatch - use cast

ポインタの型が合っていません．キャストを使用して下さい.

MSX-C では，ポインタの相互変換は必ずキャストしなければなりません.

pool over flow

プールがオーバーフローしました.

識別子(# define で定義したものや，変数名)を格納するプールが不足しました．"-r"スイッチでワークテーブルの比率を修正して下さい.「2.2.1　CF　(パーサ)」の"-r"スイッチの項を参照して下さい.

precedence error

優先順位エラーです.

優先順位で正しくないものが発生しました．'(', ')'を使って優先順位を明確に行って下さい.

redeclaration of '<識別子>'

"<識別子>"が2重定義されています.

変数・関数が2重に定義されています.

sizeof(func) not permitted

sizeof(func)は許可されていません.

関数のサイズは求められません.

sorry, too small memory

メモリ不足によりコンパイル不可能です.

stack over flow

スタックがオーバーフローしました.

式等が複雑すぎます.

static function '<関数>' not defined

スタティックな関数の<関数>は定義されていません.

スタティックな関数は，そのソースファイル中で定義しなければいけません．スタティックな関数の<関数>の宣言があって，関数の定義がありません.

symbol table over flow

シンボルテーブルがオーバーフローしました.

ソースプログラム中に識別子が多すぎるために，その情報を格納するシンボルテーブルが足りなくなりました．"-r"スイッチでシンボルテーブルの比率を大きくなるように指定して再コンパイルして下さい．「2.2.1 CF (パーサ)」の"-r"スイッチの項を参照して下さい．また，識別子(#defineで定義したもの，構造体，外部変数など)で不要なものがあるなら，それをなくすことでも回避することができます.

syntax error

構文に誤りがあります.

too many initializers

初期化のための定数が多すぎます.

配列の初期化の際，配列の大きさよりも初期値の数の方が多いと発生します.

too much parameters

パラメータが多すぎます.

引数付きマクロにおいて引数が11以上になったときに発生します.

type mismatch

型が合っていません.

数値型と構造体，共用体変数の間または，異なった構造体，共用体変数間で代入を行った場合に発生します.

undeclared function '<関数>'

<関数>は宣言されていません.

MSX-Cでは，関数を使用する際は，それまでにその関数の宣言または定義が必要です.

undeclared identifier '<識別子>'

<識別子>は宣言されていません.

変数の宣言を忘れているか，変数名を間違えています.

undeclared member name '<メンバ>'

<メンバ>が宣言されていません.

構造体，共用体の宣言でメンバの宣言を忘れているか，メンバ名を間違えています.

undeclared parameter '<仮引数>'

<仮引数>が宣言されていません.

引数の宣言を忘れているか，仮引数名を間違っています.

undefined label '<ラベル>' in function '<関数>'

<関数>には<ラベル>は定義されていません.

goto文の飛び先のラベル名が，そのgoto文のある関数内で見つからない場合に発生します.

undefined struct/union

構造体／共用体が宣言されていません．

構造体，共用体の宣言よりも先に構造体，共用体変数の宣言がきている場合にも発生します．

unexpected eof

途中でファイルが終わっています．

関数，コメント以外の部分でファイルが終わった場合に発生します．

unexpected eof in this comment

コメントの途中でファイルが終わっています．

コメントの"/＊"が多くファイルの終わりになっても対応する"＊/"が見つからないときに発生します．

unexpected eof inside function '＜関数＞'

＜関数＞の途中でファイルが終わっています．

"{"の数が多く，ファイルの終わりになっても関数の終了を表す"}"が見つからないときに発生します．

useless expression

演算結果が使われていません．

文の最初の演算子が，代入演算子でもインクリメント・デクリメント演算子でもない演算子のときに発生します．

'while' expected

while がありません．

do while の while がループのブロックが終っても見あたりません．

8.2 CGのエラーメッセージ

bad option： **＜文字＞**

存在しないスイッチ"-＜文字＞"が指定されています.

指定できるスイッチは，rkuloの各文字です.

cannot make： **＜ファイル＞**

＜ファイル＞が作成できません.

.macファイルを作成するときに，ディレクトリやディスク容量がいっぱいであったり，ディレクトリが存在しないなど，ファイルを作成することができないときに表示されます.

cannot open： **＜ファイル＞**

＜ファイル＞が見つかりません.

指定した.tcoファイルが見つからないときに表示されます.

fuction overflow in "**＜関数＞**" ...specify less than **＜数＞** by -r

＜関数＞でコード生成用の領域がなくなりました. "-r"スイッチで＜数＞より小さい数を指定して下さい.

CGでは空き領域をシンボルテーブルとコード生成用の領域に分割して使用しています. "-r"スイッチでシンボルテーブルを小さくし，コード生成用の領域を大きくして，再度CGを起動して下さい. シンボルテーブルの大きさをどんな値に設定しても，"symbol table overflow ..."と交互に発生する場合は，関数を分割するかファイルを分割して下さい.

illegal .tco file at "**＜関数＞**" **＜行＞**：**＜桁＞**

.tcoファイルが正しくありません.

.tcoファイルが壊れています. 再度，CFから始めて下さい.

sorry, too small memory

メモリ不足によりコンパイル不可能です.

stack overflow

スタックが足りません.

symbol table overflow in "<関数>" ...specify more than <数> by -r

<関数>でシンボルテーブルがなくなりました.'-r'スイッチで<数>より大きい数を指定して下さい.

CG では空き領域をシンボルテーブルとコード生成用の領域に分割して使用しています.'-r'スイッチでシンボルテーブルを大きくして,再度 CG を起動して下さい.シンボルテーブルの大きさをどんな値に設定しても,"function overflow ..."と交互に発生する場合は,関数を分割するかファイルを分割して下さい.

warning: >> more than 8 bit in "<関数>"

警告: 8 ビット以上右シフトしています.

8 ビット値を 8 ビット以上右シフトすると,必ず 0 になってしまいます.

warning: << more than 8 bit in "<関数>"

警告: 8 ビット以上左シフトしています.

8 ビット値を 8 ビット以上左シフトすると,必ず 0 になってしまいます.

warning: << more than 16 bit in "<関数>"

警告: 16 ビット以上左シフトしています.

16 ビット値を 16 ビット以上左シフトすると,必ず 0 になってしまいます.

warning: too big char constant <数> in "<関数>"

警告: キャラクタ定数の値が大きすぎます.

char 型は-255 から 255 の範囲で値を設定して下さい.

8.3 FPCのエラーメッセージ

bad option： -<文字>

存在しないスイッチ"-<文字>"が指定されています．

指定できるスイッチは，istucdの各文字です．

cannot make： [<ファイル>]

<ファイル>が作成できません．

"-c"スイッチを指定して.tcoファイルを連結する場合に発生します．ディレクトリやディスク容量がいっぱいであったり，ディレクトリが存在しないなど，ファイルを作成することができないときに表示されます．

cannot open： [<ファイル>]

<ファイル>が見つかりません．

指定した.tcoファイルが見つからないときに表示されます．

in <<ファイル>> "<関数1>" calls "<関数2>" ： conflicting number of arguments

引数の数が合っていません．

<ファイル>中の<関数1>で呼び出している<関数2>の引数の数が，<関数2>の定義と合っていません．

in <<ファイル>> "<関数1>" calls "<関数2>" ： conflicting return type

関数の戻り値の型が合っていません．

<ファイル>中の<関数1>で呼び出している<関数2>の返す型が，<関数2>の定義さい．

in <<ファイル>> "<関数1>" calls "<関数2>" ： <Nth> argument conflict

N番目の引数の型が合っていません．

<ファイル>中の<関数1>で呼び出している<関数2>の引数の型が，<関数2>の定義と合っていません．1番目，2番目，3番目の引数の時には"<Nth>"はそれぞれ，"1st"，"2nd"，"3rd"になります．

in <<ファイル>> "<関数 1>" calls "<関数 2>" : undefined

関数が定義されていません.

<ファイル>中の<関数 1>で呼び出している<関数 2>が定義されていません.

in <<ファイル>> "<関数>" ...sorry,can't check indirect call

間接呼び出しの関数は引数チェックはできません.

<ファイル>中の<関数>内で別の関数を間接的に呼び出しています. FPC では間接呼び出しの関数の引数チェックはできません.

in <<ファイル>> "<関数>" was multiple defined

<関数>が 2 重定義されています.

<ファイル>中で<関数>が 2 重定義されています.

missing func list

関数のリストがありません.

"-d"スイッチを指定した場合には, 次の引数にカンマで区切られた関数名のリストが必要です.

missing outputfile

出力ファイルがありません.

"-c"スイッチを指定した場合には, 次の引数に連結した.tco ファイルを格納するためにファイル名が必要です.

too many files

ファイルが多すぎます.

チェックするための.tco ファイルが多すぎます.

8.4 MXのエラーメッセージ

bad option： -<文字>

存在しないスイッチ"-<文字>"が指定されています.

指定できるスイッチは，lou の各文字です.

cannot make： [<ファイル>]

<ファイル>が作成できません.

分割されたファイルを作成できません．ディレクトリやディスク容量がいっぱいであったり，ディレクトリが存在しないなど，ファイルを作成することができないときに表示されます.

cannot open： <ファイル>.mac or <ファイル>.tco

分割をするファイルが.mac ファイルでも.tco ファイルでも見つかりません.

illegal .tco file

.tco ファイルが正しくありません.

.tco ファイルが壊れています．再度，CF から始めて下さい.

missing ENDMODULE

'ENDMODULE'が見つかりません.

アセンブラファイル（.mac ファイル）で，モジュールの開始である"MODULE"はありましたが，"ENDMODULE"が見つからずにファイルが終りました.

out of memory

メモリが不足しました.

分割されたモジュール名を保存しておくための領域がありません．分割前のファイルを分割して，再度 CF から始めて下さい.

Skeleton file AREL.BAT not found

スケルトンファイル arel.bat が見つかりません.

.mac ファイルを分割する場合には，カレントディレクトリか MX コマンドのあるディレクトリに，アセンブル手順を記述したファイル arel.bat が必要です.

Skeleton file CREL.BAT not found

スケルトンファイル crel.bat が見つかりません.

.tco ファイルを分割する場合には,カレントディレクトリか MX コマンドのあるディレクトリに，コード生成，アセンブルの手順を記述したファイル crel.bat が必要です.

第9章　標準ライブラリ関数一覧

ここでは，標準ライブラリ関数の簡単な一覧を示します．それぞれのライブラリ関数の詳細については，「第6章　標準ライブラリ関数リファレンス」を参照してください．

9.1　ファイル入出力関数

9.1.1　高水準入出力関数

関数名	用　途	参照ページ
clearerr	エラー状態をクリアする	133
fclose	ファイルのクローズ	144
fcloseall	標準入出力以外のファイルをすべてクローズ	144
feof	ファイルのエンドオブファイルの判定	146
ferror	ファイルが書き込みエラーを起こしたか	148
fflush	ファイルバッファのフラッシュ	149
fgets	文字列をファイルから読み込み	151
fileno	ファイルのファイルハンドルを返す	152
flushall	すべてのファイルバッファのフラッシュ	153
fopen	ファイルのオープン	154
fprintf	ファイルへの書式付きデータ出力	156
fputs	文字列をファイルへ出力する	158
fread	ファイルからデータを読み込む	159
fscanf	ファイルからの書式付きデータ入力	163
fsetbin	入出力をバイナリ・モードにする	165
fsettext	入出力をテキスト・モードにする	167
fwrite	ファイルにデータを書き込む	168
getc	ファイルから1文字読む	170
getchar	標準入力から1文字読む	170
gets	文字列を標準入力から読む	177
printf	標準出力への書式付きデータ出力	207
putc	1文字をファイルに書き出す	209
putchar	1文字を標準出力に書き出す	209
puts	文字列を標準出力へ出力する	212
scanf	標準入力からの書式付きデータ入力	220
setbuf	バッファの制御方法の設定	224
setvbuf	バッファの制御方法の設定	230

ungetc	1文字をファイルへ戻す	252
ungetch	1文字を標準入力へ戻す	252

9.1.2 低水準入出力関数

関数名	用　途	参照ページ
close	ファイルのクローズ	134
creat	ファイルの作成	135
eof	ファイルのエンドオブファイルの判定	136
isatty	ファイルがデバイスかどうかの判定	182
open	ファイルのオープン	204
read	ファイルの読み込み	215
write	ファイルの書き出し	255

9.2 文字列および文字処理関数

関数名	用　途	参照ページ
atoi	文字列を整数に変換する	126
isalnum	文字がアルファベット，数字かどうかの判定	179
isalpha	文字がアルファベットかどうかの判定	181
iscntrl	文字がコントロールキャラクタかどうかの判定	183
isdigit	文字が数字かどうかの判定	184
iskanji	文字が漢字の第1バイトかどうかの判定	185
iskanji2	文字が漢字の第2バイトかどうかの判定	187
islower	文字が小文字かどうかの判定	189
isspace	文字が空白文字かどうかの判定	190
isupper	文字が大文字かどうかの判定	191
isxdigit	文字が16進数かどうかの判定	192
sprintf	データの書式付き出力を文字列に対して行う	232
sscanf	データの書式付き入力を文字列から行う	234
strcat	2つの文字列を連結する	235
strchr	文字列から文字を見つける	237
strcmp	2つの文字列を比較する	238
strcpy	文字列をコピーする	239
strlen	文字列の長さを返す	240
strlwr	文字列中の大文字を小文字にする	241
strncat	2つの文字列を上限つきで連結する	242

strncmp	2つの文字列を上限つきで比較する	244
strncpy	文字列を上限つきでコピーする	246
strupr	文字列中の小文字を大文字にする	247
tolower	大文字を小文字に変換する	248
toupper	小文字を大文字に変換する	250

9.3 メモリ管理関数

9.3.1 低水準メモリ管理関数

rsvstk	スタック領域を確保する	218
sbrk	メモリブロックを割り付ける	219

9.3.2 高水準メモリ管理関数

alloc	メモリブロックを割り付ける	124
free	alloc で割り付けたメモリを解放する	161

9.4 ディレクトリ関数

chdir	カレントディレクトリを変更する	132
expargs	ワイルドカードを含むファイル名を展開する	142
getcwd	カレントディレクトリを獲得する	174
mkdir	サブディレクトリを作成する	202
rmdir	サブディレクトリを削除する	217

9.5 プログラム操作関数

execl	プログラムのチェイン(1)	137
execlp	プログラムのチェイン(2)	137
execv	プログラムのチェイン(3)	139

execvp	プログラムのチェイン(4)	139
exit	プログラムの実行の終了	141
_exit	プログラムの実行の終了	141

9.6 キーボード，I/O 関数

getch	キーボードから 1 文字入力する	171
getche	キーボードから 1 文字入力する	173
inp	I／O ポートからデータを入力する	178
kbhit	キー入力があるかどうかをテストする	194
outp	I／O ポートへデータを出力する	206
sensebrk	Ctrl＋C，Ctrl＋STOP 入力のテスト	222

9.7 機械語，MSX-DOS ファンクションコールサポート関数

bdos	ファンクションコールの実行(a の値を返す)	127
bdosh	ファンクションコールの実行(hl の値を返す)	127
bios	BIOS コールの実行	129
call	機械語プログラムを呼ぶ(hl の値を返す)	130
calla	機械語プログラムを呼ぶ(a の値を返す)	133
callxx	機械語プログラムを呼ぶ(結果を得る)	131

9.8 メモリ操作関数

memcpy	メモリブロックの転送	198
memset	メモリブロックに値を書き込む	199
movmem	メモリブロックの転送	203
setmem	メモリブロックに値を書き込む	228

9.9 汎用関数

abs	絶対値を返す	123
getenv	環境変数の値を得る	175
longjmp	関数間の直接ジャンプ	195
max	2つの引数のうち，大きい方を返す	197
min	2つの引数のうち，小さい方を返す	201
putenv	環境変数の値を設定する	211
qsort	ソートの実行	213
rename	ファイル名の変更	216
setjmp	関数間の直接ジャンプ	226
unlink	ファイルの削除	254

付録 A　サンプルプログラム “q.com”について

10.1　“q.com” とは

システムディスクに含まれているサンプルプログラム q.com は，直線が任意に色を変えながら図形を描き，もう一方の直線がそれを消していくというプログラムです．マシンが MSX の場合は自動的に MSX 用（スクリーンモード 2）が，MSX2 の場合は MSX2 用（スクリーンモード 8）が動作します．

MSX-DOS 上で

```
A>q
```

とタイプすると動作します．

動作しているときに任意のキーを押すと MSX-DOS に戻ります．もし，動作しているときに Ctrl+STOP をタイプしてしまった場合はリセットをかけて下さい．

q.com は q.c をコンパイル，アセンブルすることによってできた q.rel と，直線を指定した色で描くためのライブラリ line.rel，ランダムな整数を返すライブラリ rnd.rel，BASIC の ROM 内ルーチンを用いるためのライブラリ calbas.rel と標準ライブラリとを mkq.bat というバッチファイルを使ってリンクすることによって生成することができます．

10.2　q. com 内の関数を利用するには

q.com で用いられた line ( )，rnd ( ) などの関数は他のプログラムで用いることもできますが，その場合は次のような作業を必要とします．

10.2.1　関数の宣言

各関数を使うプログラムでは，必ず以下のような宣言をしなければなりません．（これらは stdio.h では宣言されておりません．）

```
VOID    line();
VOID    chgmod();
VOID    totext();
```

```
VOID    cls();
TINY    *color();
unsigned rnd();
```

10.2.2 リンク

リンクを行う場合，rnd ( )以外の関数を使用するには必ずcalbas.relをリンクしなければなりません．さらに，line ( )，color ( )を使用するにはcalbas.relより前にline.rel，color.relをリンクしなければなりません．rnd ( )を単独で用いる場合はrnd.relのみをリンクして下さい．

例

rnd ( )，color ( )を使わない場合のリンク例は次のようになります．

```
180 ck,progm,line,calbas,clib/s,crun/s,cend,progm/n/y/e:xmain
```

10.2.3 q.com内の関数の仕様について

q.comの中で使われている関数の仕様について説明します．他のプログラムでこれらの関数を用いる場合にご参照下さい．

<line>
VOID　line(x1, y1, x2, y2, color, log_op)
int　x1, y1, x2, y2 ;
TINY　color, log_op ;

座標(x1,y1)，座標(x2,y2)を結ぶ直線をcolorで指定した色を描きます．MSX2の場合にはBASICと同様なロジカルオペレーションが使えますが，MSX1の場合は使えません．ですからMSX1の場合や，ロジカルオペレーションを使わない場合には，log_opには(TINY)0などのダミーの値を入れて下さい．

座標の指定できる範囲はスクリーンモードによって異なります．スクリーンモードは次に説明するchgmod ( )で変更することができます．

例

```
line(x,y,x1,y1,c,(TINY)0);
```

<chgmod>
VOID　chgmod(mod)
TINY　mod ;

スクリーンモードを mod で指定したモードに変更します．mod には 0 から 8 までの整数が有効で（MSX1 の場合は 0 から 3），それ以外の値を与えると何の動作も行いません．

※スクリーンモードについては BASIC マニュアルの screen 文の説明を参照して下さい．

<totext>
VOID　totext ( )

スクリーンモードを元の（1 番最後の）テキストモードに戻します．ですから，テキストモードの状態でこれを実行しても何の動作も行いません．引数，戻り値はありません．

<cls>
VOID　cls ( )

画面をクリアします．

<color>
TINY　*color(fore, back, bord)
TINY　fore, back, bord ;

フォアグランド，バックグランド，ボーダーの色を指定します．
戻り値として直前の設定を格納した 3 バイトのエリアへのポインタが返されます．

例

```
TINY *prev;                          ⇦ previous color

chgmod(8);
prev = color(255, 0, 0);             ⇦スクリーン 8 でフォアを白，
                                       バック，ボーダーを黒に指定
        •
        •
color(prev[0], prev[1], prev[2];     ⇦設定を元に戻す
```

<rnd>
unsigned rnd(range)
unsigned range ;

0 から (range - 1) までのランダムな整数を返します．引数，戻り値はありません．
この関数は rnd.mac で定義されています．

10.2.4　関連書籍の紹介

現在MSX-DOS，C言語関連の書籍としては次のようなものが出版されています．q.comのようなプログラムを作成するのに大変参考になりますので，ぜひご参照下さい．

MSX　テクニカルデータブック1	アスキー・マイクロソフトFE本部編著
MSX　テクニカルデータブック2	アスキー・マイクロソフト編著
MSX2　テクニカルハンドブック	アスキー・マイクロソフトFE監修
V9938　テクニカルデータブック	アスキー・マイクロソフトFE本部／日本楽器製造株式会社編
C言語入門	Les Hancock・Morris Krieger共著 アスキー出版局監訳
入門C言語	三田典玄著
実習C言語	三田典玄著
応用C言語	三田典玄著
	以上　株式会社アスキー
プログラミング言語C	B.W.カーニハン・D.M.リッチー著 共立出版株式会社

付録B　MSX-C Ver.1.2 マスターディスク内容

マスターディスクには以下のファイルが含まれています．ご確認下さい．

¥　（コマンド，ライブラリが含まれています．）

cf.com	コンパイラ（パーサ）
cg.com	コンパイラ（コードジェネレータ）
fpc.com	ファンクションパラメータチェッカ
mx.com	モジュール抽出ユーティリティ
ck.rel	カーネル呼び出しルーチン
clib.rel	標準ライブラリ（カーネル本体を含む）
crun.rel	実行時ルーチン
cend.rel	標準ライブラリ
lib.tco	FPC 用標準ライブラリ .tco ファイル
c.bat	コンパイル用バッチファイル
readme.doc	あれば最新の情報が入っています．

¥include　（ヘッダファイルが含まれています．）

bdosfunc.h	MSX-DOS ファンクションコール用ヘッダ
conio.h	キーボード・I／O 関数用ヘッダ
ctype.h	文字種判断関数用ヘッダ
direct.h	ディレクトリ操作関数用ヘッダ
io.h	低水準入出力関数用ヘッダ
malloc.h	メモリ管理関数用ヘッダ
memory.h	メモリ操作関数用ヘッダ
process.h	プログラム操作関数用ヘッダ
setjmp.h	setjmp 用ヘッダ
stdio.h	高水準入出力関数用ヘッダ
stdlib.h	汎用関数用ヘッダ
string.h	文字列操作関数用ヘッダ
type.h	MSX-C 標準型の定義用ヘッダ

¥batch　（ライブラリ再作成と開発環境の作成用バッチが含まれています.）

arel.bat	MX 用スケルトンファイル（アセンブリ言語用）
crel.bat	MX 用スケルトンファイル（C 言語用）
cenv.bat	MSX-C 用の環境変数を整えるバッチファイル
forremk.bat	このディスクを作成するためのバッチファイル
genlib.bat	標準ライブラリ再作成開始用バッチファイル
genliba.bat	アセンブリ言語用再作成バッチファイル
genlibc.bat	C 言語用再作成バッチファイル
genrel.bat	ライブラリ.rel ファイル再作成用バッチファイル
gentco.bat	ライブラリ.tco ファイル再作成用バッチファイル
mksys.bat	MSX-C のディスク作成用バッチファイル

¥src　（標準ライブラリのソースファイルが含まれています.）

clibc.c	clibmac.mac のスケルトン作成用ファイル
direct.c	ディレクトリ操作関数ソースファイル
io.c	低水準入出力，キーボード関数ソースファイル
malloc.c	メモリ管理関数ソースファイル
process.c	プログラム管理関数ソースファイル
stdio.c	高水準入出力関数ソースファイル
stdlib.c	汎用関数ソースファイル
string.c	文字列操作関数ソースファイル
ck.mac	カーネル呼び出しルーチンソースファイル
clibmac.mac	標準ライブラリ関数アセンブラ記述部
crun.mac	実行時ルーチンソースファイル
cend.mac	標準ライブラリ関数アセンブラ記述部 2

¥sample　(MSX-C を使ったサンプルが含まれています.)

head.c	ファイルの先頭を表示するフィルタコマンド
wc.c	ファイルの行数，単語数，バイト数の表示
q.com	グラフィック表示
q.c	q.com ソース
line.c	line ( )関数ソース
line.rel	〃　オブジェクト
color.c	color ( )関数ソース
color.rel	〃　オブジェクト
calbio.mac	calbio ( )関数ソース
calbio.rel	〃　オブジェクト
rnd.mac	rnd ( )関数ソース
rnd.rel	〃　オブジェクト
mkq.bat	q.com 作成用バッチファイル
search.c	Disk-BASIC 環境での使用例
search.bas	Disk-BASIC 側のプログラム
search.bat	コンパイルから実行まで行うバッチファイル
bk.mac	
rom0.mac	ROM 化プログラムサンプル
rom1.c	

以上 64 ファイル ＋ readme.doc

索 引

【記号】

include ····· 47,61,76,78,79
pragma ····· 21,40,41,42
.com ファイル ····· 20,38
.rel ファイル ····· 91,93,94

【B】

BOOL 型 ····· 21,58,72,75,83,208

【C】

CF ····· 33,36,47,59,64,78,257,261
CG ····· 36,37,59,64,65,258,275
charconst ····· 42,43
CPU ····· 11,16,62,88,92,111

【E】

EOF ····· 82,84,85,163,170

【I】

intconst ····· 42,43
I/O ポート ····· 80,178,206
I/O リダイレクション ····· 21

【L】

L-80 ····· 26,39

【M】

MSX-DOS ····· 63,66,84,131
M-80 ····· 113

【N】

nonrec ····· 21,40,41,42,45,50,112
NULL ····· 13,14,52,74,83

【O】

optimize ····· 40,46

【P】

PDP-11 ····· 40,42,43,54,55,56,57

【R】

recursive ····· 40,41,120,269
register ····· 11,12,44
ROM ····· 108,115,116,120

【S】

stdin ····· 14,22,82,84,86,91
stdio.h ····· 21,47,48,59,61,66,72,74,76,77,79,82,84,122

【T】

T-code ····· 11

【U】

UNIX ····· 13,14,20,40,60,61,84,90
V7 UNIX-C ····· 40

【V】

VOID 型 ····· 21,75,83

【ア】

アーギュメント ····· 13
アセンブラ ··· 15,16,18,36,37,38,54,57,59,60,65,73,95,100,130
アンダーフロー ····· 42

【エ】

エディタ ····· 15,17,19,98
エラーメッセージ ····· 28,30,31,34,35,37,94,261,275,277,279

【オ】

オーバーフロー ····· 29,30,42,220,268,269,271,272
オブジェクト ····· 11,16,40,41,42,44,46,47,94
オプションスイッチ ····· 20
オプティマイズ ····· 44,46

【カ】

カーニハン ····· 291
拡張子 ····· 15,25,26,34,36,64,93
型変換 ····· 30,42,52,257
可変パラメータ ····· 47,48,49,50,54,55,56,58,62
関係演算 ····· 42

【キ】

記憶クラス ····· 44,120,264,267
キャスト ····· 42,48,52,61,63,208,209,261,264,271
キャラクタ ····· 62,276,283

【グ】

グローバルシンボル ····· 36,60,258

【コ】

コード・セグメント ····· 120
コードジェネレータ ····· 11,12,18,27,36,40,44,59,104,258,
固定パラメータ ····· 47,48,49,54,55,59,110
コメント ····· 27,28,64,257,274

【サ】

再帰モード ····· 12,40,41,296
サンプルプログラム ····· 9,10,17,20,115,120,288

【シ】

識別子 ····· 23,31,47,48,51,59
シンボル ····· 29,36,37,60,77,258

実行可能プログラム……16

【ス】

スタック……54,57,88

【セ】

整数定数……42
宣言……21,22,23,28,32,47,48,49,51,63,79,80,81,82,83,84,119

【ソ】

ソースファイル……11,15,21,26,29,30,60,113
ソースプログラム……11,26,30,109

【タ】

タイマ割り込み……119

【チ】

中間言語ファイル……11,15,27,32,36,93
抽象宣言子……47,48,261,267

【ツ】

ツール……9,20,93,260

【テ】

定義……21,22,32,33,34,35,47,63
デバッグ……238
デフォルト……40,42,55,56

【ネ】

ネスト……27,89,226,257,269

【ハ】

ハッシュ……29,268
ハンドアセンブル……15,16
バイト……12,37,54,62,63,67,74,89,109
バイト算術演算……42
バッチファイル……25,30
パーサ……26,257
パイプライン……20,21,84
パラメータチェック……92,101

【ヒ】

非再帰モード……12,40,41
標準カーネル……13,22,38,66,115
標準ライブラリ関数……48,76,79,84,122,281

【フ】

フロントエンド……11,67
分割コンパイル……30,33
プール……29,271
プリプロセッサ……21,40,41,42,43,46

【ヘ】

ヘッダファイル……22,27,76,78,79

【ホ】

ポインタ……21,30,34,35,50,52,259

【マ】

マシン語……10,15,16,107,108,109,114

【モ】

文字定数……42

【ヨ】

予約語……41,51,264

【ラ】

ライブラリ……48,76,79,84,122,281

【リ】

リロケータブルオブジェクト……16
リンク……16,38,54,59,94,95,97,109

【レ】

レジスタ割付……11,12,40,44,46

【ロ】

ロード……16,107,109,113,114,115
論理演算……42,51

【ワ】

ワーク用テーブル……29,257
ワード……42
割り込み処理……119,120

お問い合わせについて

弊社では厳重に梱包した上，細心の注意を払って製品を発送しております．万一，輸送上のトラブルが起こった場合にはご一報いただければ新しいものと交換いたします．

マニュアル作成にあたり，なるべく詳細な説明をするように心がけたつもりですが，理解できないところは実際にコンピュータと向きあって納得のいくまで確かめて下さい．また，他のページを参照するのもひとつの方法です．それでも疑問点が解決できないときは，購入された販売店に問い合わせるか，㈱アスキー　ユーザーサポート(直通電話 03-498-0299)までお電話いただければ，係がお答えします．しかしながら，回線が混み合いご迷惑をかけることもありますので，なるべくお手紙にてお願いいたします．その際には，下記の要領で記入して下さい．記入されてない項目が一つでもありますと，回答できかねる場合があります．十分注意して下さい．

また，本製品以外に対してのご意見，ご希望がございましたら，弊社までおよせください．

記

1.送付先　〒107-24　東京都港区南青山 6-11-1　スリーエフ南青山ビル
株式会社アスキー　ユーザーサポート係
TEL　03-498-0299
(祝祭日を除く月～金曜日，10：00～12：00，13：00～17：00)

2.必要項目

(1)お客様の氏名，住所(郵便番号)，電話番号(市外局番も含む)

(2)製品名，製品シリアル番号

(3)機器構成

本体装置名，メモリバイト数

CRT 装置名，フロッピーディスク装置名

プリンタ装置名

その他 I/O，I/F 装置名

(4)お問い合わせ内容

お問い合わせの内容は，できるだけ製品のマニュアルに記述されている用語を用いて，具体的かつ明確に記述してください．なお，障害と思われる現象については，その現象を再現可能な情報が必要です．当社で再現できないものは，調査ができません．その現象が発生するまでの操作手順，データを必ず添付して下さい．データディスクがある場合は，そのコピーも同封していただくと調査がスピーディーになります．

また，お客様固有と思われるアプリケーションの設計，作成，運用，保守については，当社のサポート範囲外ですので，お問い合わせいただいても回答できません．ご了承下さいますようお願いいたします．

MSX-C Ver.1.2 USER'S MANUAL

1989年4月1日　第1版第1刷発行

監修／編集　株式会社アスキー　システム機器事業部

発　行　所　株式会社アスキー
〒107-24　東京都港区南青山6-11-1　スリーエフ南青山ビル

振　　替　東京 4-161144

TEL.　(03)486-7111


ASCII
ASCII CORPORATION

「MSX-C Ver.1.2」ユーザーズマニュアル正誤表

1989年4月14日

頁	行	誤	正
P.3	20	bdosfunc.h　80	bdosfunc.h　79
P.10	18	第10章 サンプルプログラム q.com について	付録A サンプルプログラム q.com について
P.10	20	第11章 MSX-C Ver.1.2 マスターディスク内容	付録B MSX-C Ver.1.2 マスターディスク内容
P.29	下から2行目	symbol table ＜使用バイト数＞ •••	symbol table ＜使用バイト数＞ •••
P.35	8	...sorry can"t check •••	...sorry can't check •••
P.37	2	中間言語ファイルを出力します	アセンブラファイルを出力します
P.69	下から2行目	¥CHKCHR(5DH)	_CHKCHR(5DH)
P.107	3	図 6-1	図 5-1
P.107	最下行	100 CLEAR ,&HCFFF	100 CLEAR 300,&HCFFF
P.108	2	(新しい HIMEM — 1)	(新しい HIMEM - 1)
P.108	5	図 6-1	図 5-1
P.108	下から6行目	100 CLEAR ,&HCFFF	100 CLEAR 300,&HCFFF

誤

P.70

例

```
# in #include <stdio.h>

/ * /*  文字列s中のANKの小文字を大文字にする（漢字対応版）  */
cha char    *jstrupr(s)
cha char    *s;
{   {
            char    *head = s;
            char    c;

            while (c = *s) {
                    if (iskanji(c))         /*  1バイト調べる  */
                            /*  漢字なのでそのまま1バイト進める  */
                            s++;
                    else
                            *s = toupper(c);/*  ANKは必要なら大文字変換       */
                    s++;                    /*  次の文字を指すようにする     */
            }
            return (head);
    }
}
```

正

例

```
#include <stdio.h>

/*  文字列s中のANKの小文字を大文字にする（漢字対応版）  */
char    *jstrupr(s)
char    *s;
{
        char    *head = s;
        char    c;

        while (c = *s) {
                if (iskanji(c))         /*  1バイト調べる  */
                        /*  漢字なのでそのまま1バイト進める  */
                        s++;
                else
                        *s = toupper(c);/*  ANKは必要なら大文字変換       */
                s++;                    /*  次の文字を指すようにする     */
        }
        return (head);
}
```