

English Preface

As the MSX system emerged from Japan, many of its official references and development tools are mainly available in Japanese. It gets popular in other places in the world including Europe, Middle East and South America. Some references were available in local languages, this includes user guides and MSX BASIC references. However, for MSX-C, the official references are only available in Japanese. so, non-Japanese speaking developers have to depend on user forums (which are fortunately still active until today). Plus trying few scattered unofficial resources.

As video game developer and old MSX enthusiast who grew with Sakhr-AX170 and AX350, I found the online resources very helpful but not enough to cover this important development language. so, the project of translating the official MSX-C user manual to English was something in mind but unfortunately not much time to spend on it. This year we had the lockdown, instead of showing desperate I decided to work on this project.

As a non-Japanese speaking person, I had to depend on machine translation which is obviously not very good. To improve the quality, the book was translated using multiple technologies that were fused to get the best translation. Even with that, several parts were missing or not translated well due to issues in OCR or poor translation tech. For those parts they are re-written based on context and my programming background.

For code snippets and examples, they got poorly detected by the OCR software, so I had to manually re-write them. in addition to that, all code was implemented and tested on real MSX hardware (AX-350II) and they will be provided along this translation.

Figures were all re-draw in vector illustration software to get the best quality. Any annotations were also translated and added in the proper place in the figures.

Lastly, formatting and editing took big attention to make the English version as close as possible to the original Japanese version.

Overall, definitely there might be translation mistakes so I really appreciate corrections or suggestions to improve the quality of this book.

Homam Bahnassi / Dec-20

Introduction

Thank you for purchasing MSX-C Ver. 1.2.

MSX-C Ver. 1.2 is a compiler that processes source programs written in C and generates assembler source programs.

New support for MSX-DOS version 2 allows you to display messages with Kanji characters, process Kanji character data, and use features such as a hierarchical directory. This makes the standard library enhanced and easier to use.

This software includes MSX-DOS2, Text Editor, MSX M80, MSX L80, LIB.

If you don't have this software, please purchase "MSX-DOS2 TOOLS" because you will need it when you develop programs using MSX-C Ver.1.2. In addition to that, MSX-C Debug2 and MSX-C Library are available separately for efficient debugging while developing programs.

Before using this software, please read the attached "Software License Agreement" carefully, fill in the attached user registration card, and return it to us. By returning the form to us, you are deemed to have agreed to the acceptance agreement.

Please note that we will not be able to provide after-sales service if you do not return the user registration card.

This package includes the following:

* MSX-C Ver. 1.2 System Disk 1 (3.5 - FDD floppy disk)
* MSX-C Ver. 1.2 user's manual (1 copy)

MSX, MSX-DOS2, MSX M80, MSX L80 and MSX-C are trademarks of ASCII.

MS-DOS is a trademark of Microsoft Corporation.

Z80 is a trademark of Zilog Inc.

The UNIX operating system was developed by the Bell Laboratories of AT&T USA, and is licensed by AT&T.

Cautions

1. Renting this software and its manuals is prohibited. You may not copy any part of this software or manual without permission.
2. You may not copy this software for any purpose other than personal use.
3. The information in this manual is subject to change without notice. However, if you have any questions, errors in manual, please send them to us as described in the "Contact Us" section at the end of the manual.
4. Please note that we are not responsible for any consequences of using this software, notwithstanding clause (4).
5. The operation of this software is for personal use only. If you sell or distribute programs developed using this software, you must conclude a license agreement with us.
6. If you have any questions about the license agreement, please contact:

ASCII Corporation MSX Promotion Department

TEL. 03-486-834 (Monday through Friday, excluding holidays)

10:00~12:00, 13:00~17:00

Please go to the previous address.

Contents

[Chapter One MSX-C Compiler 9](#_Toc59182190)

[1.1 Overview of the MSX-C Compiler 9](#_Toc59182191)

[1.1.1 MSX-C Compiler 9](#_Toc59182192)

[1.1.2 Features of MSX-C 9](#_Toc59182193)

[1.1.3 Features of the Standard Kernel 11](#_Toc59182194)

[1.2 Creating source files to running a program 12](#_Toc59182195)

[1.2.1 Editor 13](#_Toc59182196)

[1.2.2 Assembler Source from C Language Source 13](#_Toc59182197)

[1.2.3 Generating an Executable Program 13](#_Toc59182198)

[1.3 MSX-C Development Environment (Building a Development Environment) 14](#_Toc59182199)

[1.4 Sample Program 16](#_Toc59182200)

[1.4.1 About the Sample Program "HEAD" 16](#_Toc59182201)

[1.4.2 Points to Consider when Creating a Source File 17](#_Toc59182202)

[Chapter 2 Operation of the MSX- C Compiler 19](#_Toc59182203)

[2.1 Batch Files 19](#_Toc59182205)

[2.2 Running each command 21](#_Toc59182206)

[2.2.1 CF (Parser) 21](#_Toc59182207)

[2.2.2 FPC (Parameter Checker) 26](#_Toc59182208)

[2.2.3 CG (Code Generator) 29](#_Toc59182209)

[2.2.4 Assemble and Link 31](#_Toc59182210)

[Chapter 3: Programing in MSX-C 33](#_Toc59182211)

[3.1 Specifications Differences against the (V7UNIX-C) 33](#_Toc59182213)

[3.1.1 Control by the preprocessor statement # 33](#_Toc59182214)

[3.1.2 Functions Declaration 39](#_Toc59182215)

[3.1.3 Other 42](#_Toc59182216)

[3.2 Linking to Assembly Language 45](#_Toc59182217)

[3.2.1 Passing Parameters 45](#_Toc59182218)

[3.2.2 Returning Values 47](#_Toc59182219)

[3.2.3 Reciprocal Calls to Assembly and C Routines 47](#_Toc59182220)

[3.2.4 Number of Characters with Identifiers 49](#_Toc59182221)

[3.2.5 External Symbols 50](#_Toc59182222)

[3.3 Porting a Standard C Source File 50](#_Toc59182223)

[3.3.1 Backslash "" 50](#_Toc59182224)

[3.3.2 #include <stdio.h> 50](#_Toc59182225)

[3.3.3 char and int types 51](#_Toc59182226)

[3.3.4 Passing Arguments to Variable Parameter Function 52](#_Toc59182227)

[3.3.5 Declaration of a Function (Forward Reference to a Function) 53](#_Toc59182228)

[3.4 Compatibility Between MSX-C Ver.1.1 and Ver 1.2 53](#_Toc59182229)

[3.4.1 Command Changes 53](#_Toc59182230)

[3.4.2 Changes to the Standard Library 54](#_Toc59182231)

[3.5 Methods of processing Kanji characters 55](#_Toc59182232)

[3.5.1 Internal representation of Kanji characters in MSX 55](#_Toc59182233)

[3.5.2 Distinguishing Between Kanji Modes 57](#_Toc59182234)

[3.5.3 Processing of Kanji Characters 58](#_Toc59182235)

[Chapter 4 MSX-C Library 60](#_Toc59182236)

[4.1 MSX-C Canonical Types 60](#_Toc59182238)

[4.1.1 BOOL Type 60](#_Toc59182239)

[4.1.2 FCB Type 61](#_Toc59182240)

[4.1.3 FD Type 61](#_Toc59182241)

[4.1.4 FIB Type 61](#_Toc59182242)

[4.1.5 FILE Type 62](#_Toc59182243)

[4.1.6 LONG Type 62](#_Toc59182244)

[4.1.7 size\_t Type 62](#_Toc59182245)

[4.1.8 STATUS Type 62](#_Toc59182246)

[4.1.9 TINY Type 62](#_Toc59182247)

[4.1.10 VOID Type 63](#_Toc59182248)

[4.1.11 XREG Type 63](#_Toc59182249)

[4.2 Header File 63](#_Toc59182250)

[4.2.1 Splitting the Header File 64](#_Toc59182251)

[4.2.2 How to Include the Header File 65](#_Toc59182252)

[4.2.3 Contents of the Header Files 66](#_Toc59182253)

[4.3 Summary of Standard Library Function 69](#_Toc59182254)

[4.3.1 File Input/Output Functions 69](#_Toc59182255)

[4.3.2 Strings and Character Processing Functions 72](#_Toc59182256)

[4.3.3 Memory Management Functions 72](#_Toc59182257)

[4.3.4 Directory Functions 73](#_Toc59182258)

[4.3.5 Program Operation Functions 73](#_Toc59182259)

[4.3.6 Keyboard I/O Functions 74](#_Toc59182260)

[4.3.7 Machine Language, MSX-DOS Support Functions 74](#_Toc59182261)

[4.3.8 Memory Operations Functions 74](#_Toc59182262)

[4.3.9 Generic Functions 74](#_Toc59182263)

[4.4 Differences Against the UNIX Standard C Libraries 74](#_Toc59182264)

[4.5 Creating and Maintaining Libraries 75](#_Toc59182265)

[4.5.1 Configuration of the MSX-C Library 75](#_Toc59182266)

[4.5.2 MX Library Maintenance Support Tool 76](#_Toc59182267)

[4.5.3 Creating a Dedicated Library 77](#_Toc59182268)

[4.5.4 Recreating Standard Libraries 85](#_Toc59182269)

[Chapter 5: Applications of the MSX-C Compiler 88](#_Toc59182270)

[5.1 Creating Machine Language Routines in the Disk-BASIC Environment 88](#_Toc59182272)

[5.1.1 Disk- BASIC Environment and USR Functions 88](#_Toc59182273)

[5.2 Creating Source Programs in C 90](#_Toc59182274)

[5.2.1 Accepting USR Function Arguments 90](#_Toc59182275)

[5.2.2 Using System Calls 91](#_Toc59182276)

[5.2.3 Returning Values 92](#_Toc59182277)

[5.3 5.3 From the C Source File to BLOAD File 93](#_Toc59182278)

[5.3.1 Creating BLOAD Files with L80 Only 93](#_Toc59182279)

[5.3.2 MSX-DOS2 TOOLS BSAVE COMMANDS 94](#_Toc59182280)

[5.4 About the Sample Program 94](#_Toc59182281)

[5.5 Creating a program to convert to ROM 94](#_Toc59182282)

[5.5.1 General rules for creating a ROM-based program 95](#_Toc59182283)

[5.5.2 Examples of Converting a Program to ROM 99](#_Toc59182284)

[Chapter 6 Standard Library Function Reference 100](#_Toc59182285)

[Chapter 7: Command Reference 199](#_Toc59182286)

[7.1 CF 199](#_Toc59182289)

[7.2 CG 199](#_Toc59182290)

[7.3 FPC 200](#_Toc59182291)

[7.4 MX 200](#_Toc59182292)

[Chapter 8: List of Error Messages 201](#_Toc59182293)

[8.1 CF Error Messages 201](#_Toc59182295)

[8.2 CG Error Messages 214](#_Toc59182296)

[8.3 FPC Error Messages 216](#_Toc59182297)

[8.4 MX Error Messages 217](#_Toc59182298)

[Chapter 9: Standard Library Functions 219](#_Toc59182299)

[9.1 File Input/Output Functions 219](#_Toc59182301)

[9.1.1 High-level input/output functions 219](#_Toc59182302)

[9.1.2 Low-level Input/Output Functions 220](#_Toc59182303)

[9.2 Strings and Character Processing Functions 220](#_Toc59182304)

[9.3 Memory Management Functions 220](#_Toc59182305)

[9.3.1 Low-Level Memory Management Functions 220](#_Toc59182306)

[9.3.2 High-Level Memory Management Functions 221](#_Toc59182307)

[9.4 Directory Functions 221](#_Toc59182308)

[9.5 Program Execution Functions 221](#_Toc59182309)

[9.6 Keyboard, I/O Functions 221](#_Toc59182310)

[9.7 Machine Language, MSX-DOS Functions 221](#_Toc59182311)

[9.8 Memory Operation Functions 221](#_Toc59182312)

[9.9 Generic Functions 222](#_Toc59182313)

[Appendix A Sample Program 222](#_Toc59182314)

[10.1 About "Q.com" 222](#_Toc59182316)

[10.2 Utilizing Functions from q.com 222](#_Toc59182317)

[10.2.1 Declaration of the Functions 223](#_Toc59182318)

[10.2.2 Linking 223](#_Toc59182319)

[10.2.3 Descriptions of q.com Functions 223](#_Toc59182320)

[10.2.4 Introduction to Reference Books 224](#_Toc59182321)

[Appendix B MSX-C Ver.1.2 Master Disk Content 225](#_Toc59182322)

[About Us 227](#_Toc59182323)

Composition of This Manual

This manual is written with the following three points in mind:

1. Enables users who are new to C to enter the resistant world of C and become familiar with the MSX-C compiler.
2. This course enables users who use the MSX-C compiler as a tool for developing programs to grasp the features of the system and put it to practical use right away.
3. The system should be structured so that the items can be referred to immediately when necessary.

Sample programs are provided to explain how to operate the system in an easy-to-understand manner for beginners. Furthermore, the features of the MSX-C compiler as a processor and the coding differences from the standard C compiler are explained in a single chapter.

In addition, a separate chapter is provided for quick reference of the standard functions library and the parameters of each command, which can be used as a reference manual.

The chapters in this manual are arranged so that even beginners can understand the MSX-C compiler by reading them sequentially from the start. If you have experience using C in other systems, please refer only to the parts you need.

The following is a summary of this manual from chapter 1 onwards.

Chapter 1 MSX-C Compiler

This chapter provides an overview of the MSX-C compiler, a description of how to set up a development environment, an overview of the flow of creating a source file and executing a program, and the accompanying sample programs.

Chapter 2 MSX-C Compiler Operations

Readers will learn the operation of each command of the MSX-C compiler while compiling sample programs.

Chapter 3 Programming in MSX-C

Starting with the differences in language specifications from UNIX-C V7, which is a standard C language implementation, this book explains essential knowledge for developing programs in MSX-C, such as how to improve object efficiency by MSX-C's original processing and how to link with the assembly language.

Chapter 4 MSX-C Standard Libraries

This chapter provides an overview of the MSX-C's library routines, the differences from the UNIX-C V7 library, and how to create and maintain your own libraries.

Chapter 5 Applications of the MSX-C Compiler

As an application example of MSX-C, we will show the actual development of a program that runs on an environment other than MSX-DOS.

1. Creating a machine language subroutine under BASIC
2. Creating a program to convert to ROM

Chapter 6 Standard Library (Functions Reference)

The accompanying functions library are explained one by one. Examples are given in the book, and details of the functions are explained in detail.

Chapter 7 Command Reference

The parameters and functions of each command can be referred to in a table format.

Chapter 8 List of Error Messages

Error messages are described for each command.

Chapter 9 List of Standard Functions Library

A table of the standard functions library arranged by function field. The reference page in Chapter 6 is also included in the table.

Chapter 10 Sample Programs About q.com

This section describes q.com, a sample program written in MSX-C.

Chapter 11 MSX-C Ver. 1.2 Master Disk Contents

This is the list of contents of the disk which is included in the MSX-C Compiler Ver.1.2. Please check as much as possible before using it.

# Chapter One MSX-C Compiler

## Overview of the MSX-C Compiler

### MSX-C Compiler

The MSX-C compiler version 1.2 (here in after referred to as MSX-C) designed for MSX-DOS version 2 (here in after referred to as MSX-DOS) is a two-pass C compiler that runs on MSX-DOS2 and outputs Zilog mnemonic assembly source programs, but does not work on MSX-DOS version 1. The compiler is composed of CF and CG programs. CF (Front End) is called the first pass of the compiler, it reads the source program in C, checks for grammatical and semantic errors, and creates an intermediate language file (called T-code) on disk. The CG (code generator), which is the second pass of the compiler, reads this intermediate language file and converts it to an assembly language source file.

The package also contains a program, FPC, which reads the intermediate language file output from CF and checks that the parameters allocated to each function are consistent with the definition of the function, and a program, MX, which supports the creation of standard library routines by the user himself for development efficiency.

### Features of MSX-C

The most important feature of MSX-C is that its generated object code is compact and efficient comparing to programs written in conventional assembly languages. It is now possible to develop in C programs that have been difficult to program in languages other than the assembly language, and programs in ROMs that have been developed in the past.

Now, MSX-C introduces several new methods to achieve this high code efficiency. In particular, the introduction of automatic register allocation is a groundbreaking attempt for a compiler for microcomputers, and it has succeeded in improving the quality of the object code.

If a variable has its value in a register in the CPU, it is much more efficient than being in memory and accessing its memory for every operation. However, in a standard C compiler, the programmer needs to decide which variable is most efficient to assign to a register and declares it with the register declarator. This is a time-consuming task for the programmer to begin with the 8080.

On a machine with an 8-bit CPU, the compiler is limited by the number of registers in addition to other limitations. We were not able to introduce a method called register declaration. The MSX-C code generator automatically does the best register allocation for you and the user doesn't need to be aware of it.

It also includes a lot of new methods for more efficient code generation, such as support for byte (8-bit) arithmetic, accessing function arguments to registers, and introducing non-recursive modes.

An example of MSX-C compilation results is shown below.

The subroutine toupper() returns the value of the function by replacing a single letter of the argument with a capital letter if it is a lower-case letter. toupper.c was compiled, and the resulting assembler source is toupper.mac.

The variable c is always placed in the A-register, not in memory, and the 8-bit (char) value is not expanded to 16 bits but remains 8 bits in operation.

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. ; MSX-C ver 1.20x (code generator)
2. cseg
3. toupper@:
4. cp 97 ←97="a"
5. ret C
6. cp 123 ←123="z"+1
7. ret nc
8. sub 32 ←32="a"-"A"
9. ret
10. public toupper@
11. end

### Features of the Standard Kernel

This section describes the features supported by the standard kernel of MSX-C. A kernel is an MSX program that adapts the information passed to the program when it is invoked by DOS to the C environment. That is, it parses and splits the command arguments, opens a standard input/output file, and so on. The existence of the kernel allows the MSX-DOS interface and UNIX interfaces. The standard kernel is linked with the standard library.

#### Arguments

The command arguments are separated by whitespaces (space or tab). To include whitespaces, "|", ">", etc. in the argument, enclose the entire argument in double quotation marks ("). " is removed from the argument. The following items are treated as single characters inside the double quotation marks.

* The \n newline character (\n)
* \r return character (\r)
* "" A single double quotation mark (")

The first argument of main() is an integer, which indicates the number of the arguments +1. Usually, the following parameters are received:

main(argc, argv)

int argc;

char \*argv[];

{

...

}

The command name is passed to argv[0] in MSX-C as well as the standard C on UNIX. The command name is given as a full path name (i.e., a file name that consists of the full name of the drive, the path, the file name, and the suffix). However, argv[0] may be a null string or a different command name if the program was invoked using other than COMMAND2.COM and standard library ver. 1.2.

The arguments to the command are passed after argv[1]. At the end of the argument, the next to the last argument (argv[argc]) is set to NULL. Arguments are case-sensitive, regardless of the environment variable UPPER.

Note

For more information on the UPPER environment variable, refer to Chapter 7 "..." on page 81 of the MSX-DOS2 Reference Manual.

Example

A>a:\bin\dump /n/s100 msxdos2.sys command2.com ◄┘

When the above command is invoked from the command line, the argument to main() is as follows:

argc 4

argv[0] "a:\BIN\DUMP.COM"

argv[1] "/n/s100"

argv[2] "msxdos2.sys"

argv[3] "command2.com"

argv[4] NULL

#### Standard input/output file assignments

On UNIX, five standard input/output files (high-level input/output) are opened when a program is invoked. In MSX-C, stdin, stdout, stderr, stdaux, and stdprn are opened in the same way. The input and output directions of stdin can be input, while other directions can be output only. stderr and stdaux can be input and output in the standard C, but they are unidirectional (output only in this case) because this is not possible in the MSX-C's \*\*\* library. The buffering methods are: row buffering stdio, stdout and stdprn while unbuffering for stderr and stdaux for. When it is redirected to a file, it is fully buffered. For details of buffering, please refer to ‎4.3.1B) Buffering of High-Level Input/Output Functions.

Note

Only stdin and stdout can be used for file redirection at the command level. For low-level input/output functions (access by file handle), stderr and stdaux can be used for both input and output.

## Creating Source Files to Run a Program

In the previous section, we gave an overview of MSX-C and some of its features as an 8-bit compiler. To actually develop a program in C, a C compiler is not enough, you need an editor to create C source files, and an assembler to translate them into machine language, because the C compiler generates source files in assembly language. (A \*\*\* assemble is also available.)

In this section, we will follow the steps from the creation of a source file to the generation of an executable program.

This section describes a program development environment on MSX-C.

### Editor

The first step in developing a program in C is to create a C source program. The editors are needed here. (Figure 1. MSX-C Program Development).

In actual development, we must complete the program through trial and error, and we must modify the source files many times. So, you need an easy-to-use editor with rich file editing features.

Currently, there are several editors available for MSX-DOS2, including the text editor that comes with the MSX-DOS2 TOOLS package.

In MSX-C, it is recommended to use C as the file name extension for C source files. The compiler doesn't require to specify the extension, it implicitly compiles the file with C as the file extension. It is possible to use other extensions, but you must include them at compile time.

### Assembler Source from C Language Source

Once your source files are created, the next step is to compile them, as in d) in 1.1.1, but if you find a syntax error in your source file during the first pass, you must return to your editor and make changes to the file. Then, the process of feeding the CF again is repeated. (Figure 1, step 2 in page 21)

If you exit CF without errors, an intermediate language file is created on the disk. This intermediate language file has the extension TCO.

By running this intermediate language file through the second path of the compiler, it is converted to an assembler source file by Z80 mnemonics and saved on the disk with the MAC extension. (Figure 1, step 4 on page 21).

### Generating an Executable Program

This concludes the two main paths of the C compiler. The generated assembler sources are of such high quality that they are not comparable to those written by programmers who are familiar with the assembly language. Of course, there are some peculiarities inherited in the assembler program generated by the compiler.

In a day, the work of assembling is to replace each instruction (mnemonic + operand) in the assembler source with its equivalent machine language (instructions that can be understood and executed by the CPU). (Figure 1, step 5 on page 21)

For example, in the so-called manual assembly process, it calculates the addresses of the jump destinations and the subroutines called by the CALL instruction, and then attaches the assembler mnemonics and labels to the machine code and real addresses one by one.

The assembler and linker perform similar tasks instead of the manual assembler, MSX-DOS MSX-M80 for the assembler and MSX-L80 for the linker.

The MSX-M80 reads the Z80 or 8080 mnemonic assembler source files and converts them to relocatable object files with the file name extension REL. In the relocatable file, the assembler mnemonic is transferred to the machine code, but at this stage, the address to which the program is dropped off in memory is uncertain, and therefore the destination address of the jump instruction is only relatively fixed. And like external subroutines, such as library functions, are just given names, so there is no address to call in the CALL instruction.

In other words, a relocatable object can be referred to any address by giving a start address, and the external subroutines called in the program can be freely replaced by the same name.

By passing information about the relocatable object file created by the user in the previous work, the library, which is a set of external routines (.REL file), and the address from which the program is loaded and executed, to the MSX-L80, the address is determined and the program is executable next to (esp. to)

With a given parameter, L80 gives the actual address of the instructions for jumps, calls, etc. in the program and combines the m with the functions use in the user

## MSX-C Development Environment (Building a Development Environment)

As described in the previous section, various tools are required to develop programs in C in addition to the programs in this package. The floppy disk that comes with the software does not have enough space to write user programs because it is not very large and contains the library sources and sample program sources. So, it would be a good idea to copy only the files necessary for normal development from the attached disk to the disk containing the MSX-DOS2 system, and then create a system disk containing all the utilities such as the assembler, linker and editor.

The following is an example of an MSX-C compiler-centered development deck. This example is built by executing the batch file mksys.bat (located in the \batch directory), which is included in the MSX-C compiler.

1. Format a new disk.
2. Copy mksys.bat to the formatted disk. If it is in the B-drive, it looks like this:

A>copy b:\batch\mksys.bat ◄┘

1. Put the copied disk in the A-drive and enter the following command:

A>mksys b ◄┘

1. Now you can put the MSX-DOS2 system disk and other file s into drive-B at the prompt.

Now you can make a disk for working in the root directory. If you are using a drive simulator, replace the disk after the message "Insert disk for drive <d>:"

The disk created by MKSYS is now ready to work in the root directory, but the environment (environment variables) is not yet ready. If you are on A-drive, do the following:

A>cenv a ◄┘

The argument for CENV is the name of the drive that contains the disk created by MKSYS (a single letter of ''b'' for the B-drive). CENV is required to be executed once you turn off the power, so you need to put the same contents in autoexec.bat on the MSX-DOS2 startup disk. It is a good idea.

The resulting disk should contain the following files (Suppose there is a disk in the A-drive)

a:\ (contents of the root directory)

mksys.bat Batch file for MSX-C disk creation

msxdos2.sys MSX-DOS2 system files

command2.com "

ck.rel library file

clib.rel "

crun.rel "

cend.rel "

c.bat Batch file for compilation

cenv.bat Batch files to setup the MSX-C environment

a:\bin (Contents of the command directory)

m80.com Assembler

l80.com Linker

cf.com compiler (parser)

cg.com compiler (code generator) standard

fpc.com library for function parameter

lib.tco checker FPC

a:\include (Contents of the header files directory)

bdosfunc.h

conio.h

ctype.h

direct.h

io.h

malloc.h

memory.h

process.h

setjmp.h

stclio.h

stdlib.h

string.h

type.h

Setting environment variables

INCLUDE a:\include

PATH a:\; a:\bin

Please make the environment more user-friendly by adding a batch file for creating the library, a batch file for that purpose, and filters for handling text files, such as MSX-DOS2 TOOLS. ...

In addition to the system disk as described above, you can also make source files and compiler work disks, so that even if you have a disk drive HDD, you can secure enough space, so that your source programs can be stored in the disk space. This avoids traps such as not being able to save due to a lack of data and system file corruption.

The work disk contains the system files and the minimum number of files, and the source files are created on this disk.

If you have two or more disk drives, insert a system disk in the A-drive and a work disk containing the source files in the B drive, for example:

A>c b:head ◄┘ c.bat runs and compiles the head.c on the B drive

and so on, you can create intermediate language files, assembler sources, executables, and so on, all on the B-drive.

Even if you have a single disk drive, the MSX-DOS two-disk simulator allows you to use two different disks in the same way, alternately inserting and removing the system disk and the work disk. For details on the two-drive simulator refer to your MSX-DOS or other manuals. (At this time, however, you have to insert and remove the discs frequently, so be careful to use the right discs.

## Sample Program

The attached sample program is explained below.

### About the Sample Program "HEAD"

#### Outline

HEAD is a UNIX-style filter command that outputs a specified number of lines from the beginning of the ASCII file read from a specified or the standard input. By compiling and linking the sample source file "head.c", which is stored in the attached disk, according to the instructions in Chapter 2, head sample source file is turned into a .com file that can be executed on MSX-DOS2, which is a useful tool command with support for I/O redirection and pipeline.

Specifically, it is used as follows:

A>head <filename> ◄┘

The first 10 lines of the file specified by <filename> are output to the console screen. In this case, the name of the file is displayed at the beginning of the output, and each line is prefixed with a line number from the beginning of the file.

#### Option Switch

An option switch can be specified next to the command name in the HEAD command. This can change the behavior of the command, such as output format and content. On UNIX commands, you can use the option. A hyphen is used to specify the switch, as in "-l", but this program uses a "/" slash instead. The following optional switches are supported.

Option names can be specified in both upper and lower case:

/L <lines> Specifies the number of lines to output

/N Omits the output of the file name and line number

#### I/O Function

MSX-DOS2 supports it.

#### Error Checking, Wild Card and…

If a non-numeric parameter is specified with "/L", it returns information such as the location of the error and displays the command's format.

Wildcards can't be used to specify the input file.

If a file is not specified, the input is taken from the standard input, redirection and pipeline are available.

### Points to Consider when Creating a Source File

#### Preprocessor

All declarations of the standard library file number, macros, etc. are included in the stdio.h header file. Also, the definitions of the VOID and BOOL types are done by including type.h in it. When using these functions and types in your program, include them at the beginning of the source file with the #include preprocessor statement.

The MSX-C compiler can switch the compile mode in various ways, depending on several pre-processor statements. In this program the default function mode is set to non-recursive by specifying #pragma nonrec. For details of the control by these preprocessor statements, refer to ‎3.1.1 Control by the preprocessor statement #

#### Command Arguments

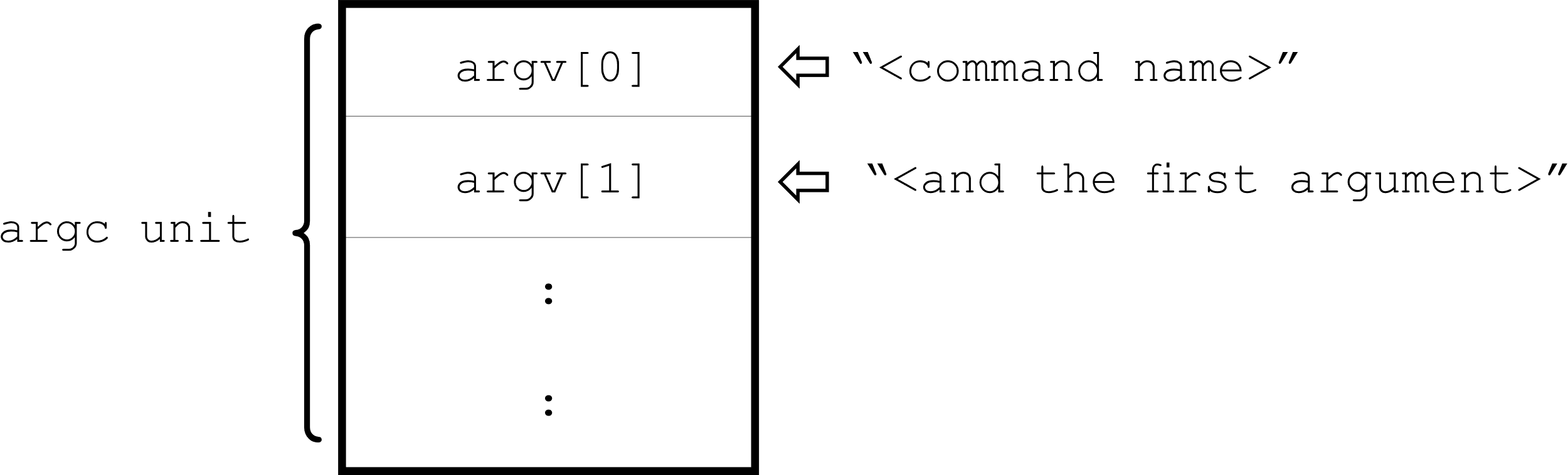
Arguments to a command entered in MSX-DOS are passed through the arguments argc and argv of the function main()defined in the user program as follows:

1. The first argument of main() is given as an integer value (the number of i).

int argc;

1. The second argument is an array of pointers to the actual parameters.

char \*argv[];



Here, a module called CK (kernel) serves as an interface between the user program and MSX-DOS2. The kernel analyzes the command line before control is passed to main(). It is passed to main() as shown above. For example:

A>head < test ◄┘

Therefore, the first argument of main() is 1, and the file test can be accessed by the file pointer stdin in the user program.

A>head ◄┘

The same applies to the case where the only input is \*\*\*, but in this case stdin is assigned to the console, not to a file.

For details of the standard kernel functions, see Section ‎1.1.3 Features of the Standard Kernel and ‎4.5.1 Configuration of the MSX-C Library.

#### Functions

One of the characteristics of C programming is that programs are often divided into multiple relatively small modules (functions). In the case of defining a function to be called later by the caller, you must declare the function in advance. For calling a library function, this is done by including its header file, but the declaration of a function in the same source it is done through the following methods:

1. Declaring it at the beginning of the caller's function.
2. Defining the called function before the calling function. (In this case, no declaration is required)
3. At the beginning of the source code, summarize the functions called by the module.

Note that main() is defined at the end of the source code. Thus, MSX-C does not use implicit declarations like the standard C, because it follows the style of "the referenced identifier is declared up front". However, by specifying the ''-f" switch when CF is invoked, it is possible to compile the function as if it were implicitly declared, assuming that the undeclared function returns an int type.

# Chapter 2 Operation of the MSX- C Compiler

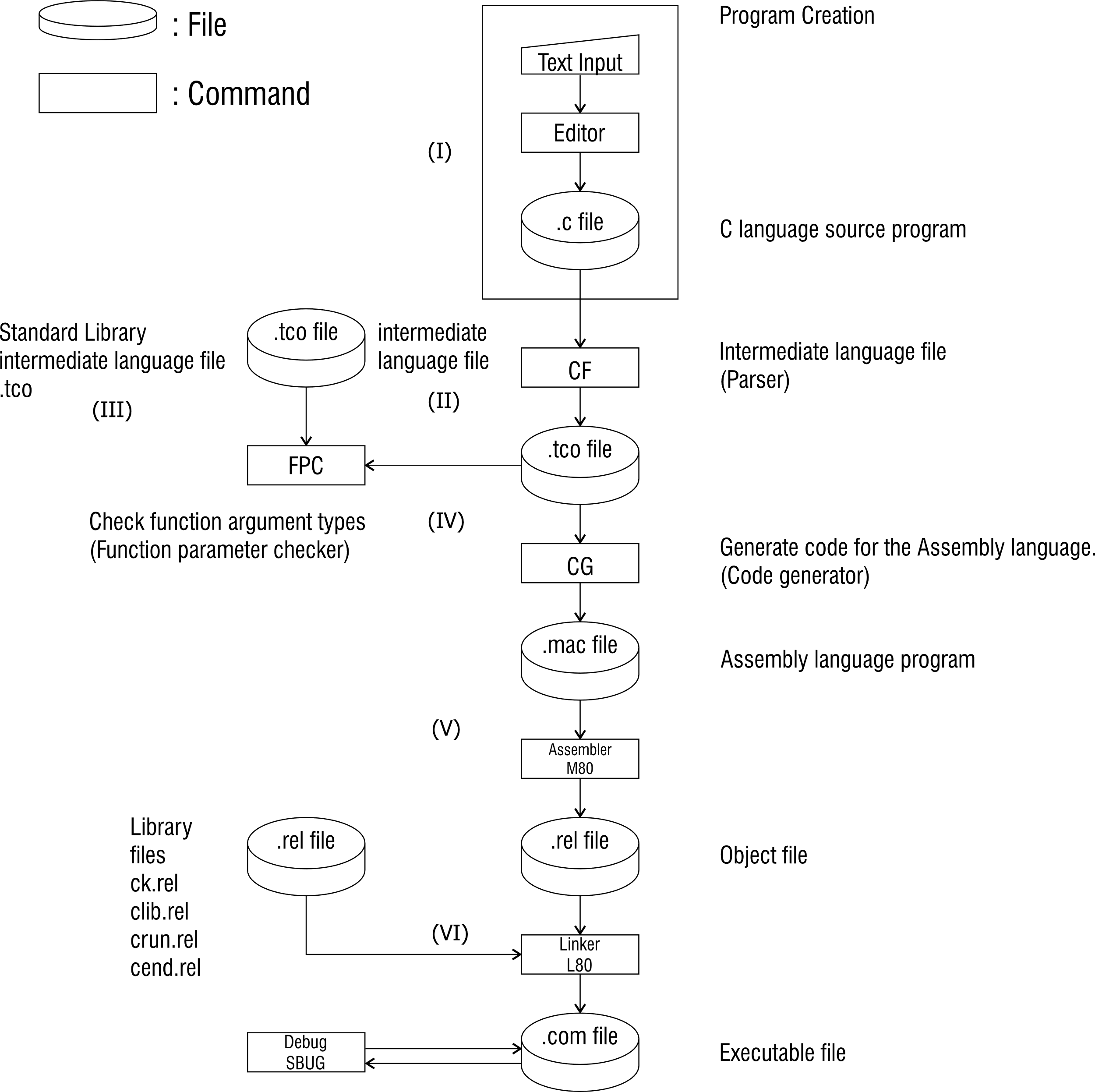


Figure 1. MSX-C Program Development

## Batch Files

To generate the C source program (.c) to the executable program (.com), you need to execute a few commands. If you use a keyboard to input each command, it will take a lot of time and effort, and probably many mistakes. Therefore, a batch file is available to complete the process from compiling the source program to generating the executable program with a single command.

The contents of the BATCH file, c.bat, that comes with MSX-C, is as follows:

If went with no error this creates the executable program from the source program at once, as shown in page 21 steps (II) through (VI).

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

The format for compiling with c.bat is as follows:

A>c filename ◄┘

where filename is the name of the source program to be compiled, and must be specified without any extension.

Now, let's compile the source program. The source program used here is head.c, which is a program to cut out the top of the file.

A>c head ◄┘ Only this one line is actually typed from the keyboard

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-86 (c) 1981, 1985 Microsoft

Data 0103 182E <5931>

36744 Bytes Free

[0103 182E 24]

A>

This completes the compilation and the generation of the executable program. Three new files are created here: head.rel, head.com, and head.sym, with head.com being the executable program.

Now that the executable program has been generated, you can use head.com as a command, for example:

A>head /n /l5 head.c ◄┘

In this example, the first fifteen lines of head.c are displayed without the file name.

## Running each Command

In ‎2.1, we used a batch file to generate the executable program from the source program, but it is also possible to invoke each command invoked in the batch file separately. This section describes each command.

### CF (Parser)

The format of CF is as follows:

CF [options] filename

where filename is the name of the C source file. The file name can be specified as a drive name, path name and extension. If the extension is omitted, the c will be added by the parser. If you specify the extension, you can use other file extension as a C source file.

CF is the first program that must be executed by the MSX-C compiler when it compiles a C source file (the first step of the compiler). (Step 2 on page 21)

CF reads the input C source file (specified by filename) and writes an intermediate language file as the output. All the errors in the source files (syntax and semantic errors) are detected.

CF outputs intermediate language files and error messages. Intermediate language files are output to the directory containing the source files. In case of an error, no intermediate language file is created.

You can interrupt the compile by pressing Ctrl+C or Ctrl+STOP during the compile. However, the intermediate language files remain, so please delete them.

The intermediate language is called T code, and the intermediate language file output by CF has the extension tco. The tco file is used as input for FPC (parameter checker) and CG (code generator).

In ver.1.2, it is now possible to correctly recognize Kanji characters in strings. Up to ver.1, it was possible to use Kanji characters for comments, but if you used them in a string, they were changed depending on the characters, which caused an inconvenience. In CF, you can specify the switch to recognize Kanji characters correctly.

Normally, C programs include header files, but starting with version 1.2, it is possible to specify the default directory for the include files. The directory is specified by an environment variable which is a feature of MSX DOS2. Set the directory name to the environment variable INCLUDE, and then start CF. Create a directory, put all the header files that come with MSX-C in it, and set the name of the directory to the INCLUDE variable. Assuming that you have created a directory named \INCLUDE in the A-drive and you put the header files in it, you can set the environment variable INCLUDE as follows:

A>set include=a:\include ◄┘

If you have enough space in your RAM disk (H-drive), you can copy the entire redirection to it before compiling.

The following is a description of the options available and their meanings:

Options

-c Do not nest comments.

Unlike the standard C, the MSX-C can nest comments. However, this option suppresses the nesting of comments and treats them in the same way as in standard C. Nesting comments means that you can do the following things in a source file.

...

/\* same function 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 \*/

}

...

We created this function, but the same function exists in the standard library, so we bypass it by commenting it. The function contains comments inside the code, so we end with comments within a commented function. If the "-c" option is specified, everything from the first "/\*" decision to "returns value \*/" is a comment, indicating the end of the function. However, "}" which indicates the end of the function and "\*/" is no longer a comment. That is, some kind of error should occur during the console.

-e [filename] Outputs an error message to a file.

Normally, all error messages generated by CF are output to standard output (console), but with this option, error messages are output to a file. It is recommended to use this method. The output file can be specified by filename. filename can be specified as a drive name, path name and extension. The extensions .dia and .a are used, which are abbreviated. You can also specify any extension other than .dia to the output destination of the .dia file, and if the filename is a single letter or a ":" (colon), then the current directory of the drive can be set to "\". If you specify a directory name ending with ''' (quote), the .dia file is set to the directory, and if it ends with anything other than ''\'', the .dia file is set to the same name as the source file/directory. If there is no error, no file is generated. No other options can be added after this option.

-f Treat undeclared functions and parameters as int implicitly.

The standard C treats undeclared functions and function parameters as int types, but MSX-C prohibits this by default. When this option is specified, the int type is implicitly assumed for these variables and the compatibility with the standard C is preserved.

-j Recognizes kanji in strings correctly.

If you don't use this switch, some kanji characters may not be recognized correctly when they are inserted into a string. This is because the C escape character "\" is used in the Kanji code, so you can specify -j to make it possible to recognize Kanji characters correctly. In a string of characters, use the switch if the character is included in the On the other hand, you should not use half-width (1-byte) hiragana. If you don't use half-width hiragana or Kanji characters, you can use either, but you don't have to specify them.

-o [filename] Creates an intermediate language file (.tco file) in an arbitrary directory.

By default, the tco file is generated in the same directory as the source file. Using this option, you can create an intermediate language file with an arbitrary file name in an arbitrary directory.

filename specifies the output destination of the .tco file. An alphabet or ":" (colon) to the current directory, or a directory name ending in "\" pointing to the current directory, or If it ends with anything other than ''\'', that's the file. If file name is omitted, the default behavior is the same. No other options are available after this option.

Examples

-o test\ Outputs an intermediate language file in a directory named test.

-o \foo.t Outputs an intermediate language file with the file name foo.t

-r P: S: H Allocate the tables for the compiler's requirement in the ratio of P : S : H.

where P is the pool, S is the symbol table and H is the hash table. By default, this ratio is 13 : 6: 4. If a table overflow occurs during compilation, the number of overflowed tables increase the value and recompile.

-m Displays the usage of the memory.

Memory allocation is indicated for each of the P:S:H described by the "-r" option. It is a good idea to use the same method. If the compilation is successful, an intermediate language file is created, as usual. The display format is as follows:

pool <Bytes used by pool>/<Allocated free area in bytes>

symbol table <Table symbol bytes>/<Allocated area bytes>

hash <Bytes used in the hash>/<Allocated bytes>

The number of bytes used to the left of "/" is the number of bytes used by table to date. The number of allocated bytes is the maximum number of bytes allocated to the table. When a table overflow occurs, the memory allocation is automatically displayed, so please specify the … rate of bytes used by each of them with "-r".

-s Continues batch processing even if an error is detected in the source program.

This option is especially useful for compiling multiple source files within a single batch file when performing a separate compilation. You can check all source files for errors without interrupting the whole process with an error in a single source program.

-t Allows automatic type conversion between a pointer and an integer.

By default, such a mixture of types between a pointer and an integer is an error in MSX-C. This option allows the type conversion to be done automatically between the pointer and the integer. It is recommended for MSX-C to write a program that can be used to compile without the "-f" option and this "-t" option.

##### Error Messages

There are two types of error messages, messages about the behavior of the command and messages for the compilation results. In this section, we will describe each message in detail. See Section ‎8.1 CF Error Messages for actual error messages.

###### Command Behavior Messages

This is a message about the operation of the CF command. For example, a message saying that the source file is not found. When such a message is displayed, CF almost always aborts the process and returns to the command level.

###### Compilation Result Messages

This is a message about the compilation of the user's source file. In this case, CF tries to recognize the source as correctly as possible and continues to compile it. However, if there is an error such as a lack of work area, it is not possible to continue the process, so it aborts the operation and returns to the command level.

There are two output formats for this message. It goes something like this

(1) line M column N :<message>

(2) <message>

<message> indicates an error message, the M and N lines … and digits. (1) indicates that an error occurred at the position (.), and (2) increases the range. The range is within a function or a whole source file.

Messages are usually very accurate in pointing out errors but forgetting important characters such as "}" can lead to a large number of error messages that cannot be interpreted correctly in the rest of the C source code. This is also a feature of the free-form C language, which ignores whitespace outside the string and whitespace separating identifiers. The large number of messages generated does not all correctly point out the errors, but most of them are messages that could not be interpreted correctly. In this case, the real error is usually in the first part of the mass message. Fixing it would reduce the number of messages a lot.

Also, we have mentioned that there are two ways to display error messages, all lines and digits are omitted in ‎8.3 FPC Error Messages.

Example

Taking the head.c example above, let's see the actual conversion with CF.

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 6888 89-01-17 2:54p << Cl intermediate language file

9K in 2 files 473K free

Next, compile the program error.c, which has an error in the source program, and output the error message to a file instead of the console.

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 266 89-01-20 9:21p << Message file

4K in 2 files 473K free

### FPC (Parameter Checker)

The format for FPC is as follows

FPC [Optional] file1 file2...

where file1 file2 ... is the name of the intermediate language files (.tco file). The file name can be specified as a drive name, path name, and extension. When the extension is omitted .tco is assumed. You can also use the wild card (\* or ?).

The purpose of FPC is to check whether the return type of the declared function matches the actual return type and whether the type of the parameter passed to the function matches the type of the parameter defined in the function declaration. That is, the use of FPC minimizes potential bugs due to unintended parameter or function type mismatch. All the types of the standard library functions are available in the file lib.tco.

Let us explain it with a practical example. See the following source program (denoted as test.c)

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

In this program, two functions f1() and f2() are declared as int types in function main(), respectively (line 5). On the other hand, the function f1() is defined as taking two parameters of type int (lines 12-13). However, in function main(), the first parameter to function f1() is a parameter of char type "a" (line 7). In addition, the function f2() is defined in the function definition as having a return value of type char, even though it is declared as an int type (lines 18 ~ line 21).

The discrepancy between these two types is not detected by the parser (CF). Unintentional inconsistencies such as these will naturally cause a bug, but FPC detects these problems and uses the He points it out to the char.

Now, let's compile this source program and check it in FPC.

A>cf test ◄┘

MSX-C ver 1.20x (parser)

Copyright (C) 1989 by ASCII Corporation

complete << parser goes through without a problem.

A>fpc Iib 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>

Thus, in the case of the function f1() the type of the first parameter of the function does not match and le the al number f2() the "type of the return value is mismatched", and the contradiction is precisely pointed out. We use ''lib'' on the FPC command line, lib.tco is specified because there is a standard library function in the program called which is printf().

It is recommended to run FPC on the intermediate language (.tco file) generated immediately after a successful completion of CF (parser).

When a given intermediate-language file is not found in the current directory, it looks for the directory where fpc.com is located. If that doesn't work, it prints 'cannot open: <filename>' and exits. If you place the file in the same directory as fpc.com and specify the file without a pathname, you will hardly notice the lib.tco file. In this case, the current directory has a lib. If there is a .tco, it is used.

##### Options

|  |  |
| --- | --- |
| -s | Even if an error is detected by FPC, the batch processing continues.  This option is useful when you want to check many programs at once, for example, when you want to perform separate compilation. |
| -u | References to an undefined function is not checked.  When this option is specified, there is no error when referencing undefined functions. |
| -t | Strict type checking  By default, the int, unsigned and pointers are treated as the same type, but they will be checked as a completely different types if this option is specified. |
| -j | warns against indirect calls. (See Error Messages) |
| -c <filename> | Creates a .tco file for the library functions.  Compresses the .tco file and creates a concatenated file with the filename that contains only the header (arguments and return values) of the library functions. That way, only the information necessary to check the library function with FPC is included in the file, so the file requires less diskspace. <filename> can be a drive name, file name, and extension. If the extension is omitted, .tco is used. For more details, please refer to Section ‎4.5.3 Creating a Dedicated Library. |
| -d <don't check function> | Don't check <Function List> are functions that can have additional parameters at the time of call and allow them to be omitted. For example, the function bdo() is not constant, depending on the MSX-DOS function call, it may have no second argument, or it may be of type int or char. Therefore, if a check file is created without specifying anything, a message saying "The number and type of parameters do not match" will be output when checked by FPC. The "-d" switch is used for such a case. <don't care function list> is a list of function names connected by a comma (tete-a-tete). Do not include any spaces. See the example or the standard library re-creation batch file gentco.bat The output switch must be used with the "-c" switch to make it useless. |

Example

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

In this example, we compress mylib1.tco and mylib2.tco and concatenate them to create mylib.tco.

In that case, the bdosd() and bdosb() ignore the extra arguments.

##### Error Messages

In the following sections,

filename is the name of the file where the error was found

func1 is the name of the function for which error was found

func2 indicates the name of the function that encountered an error on the call to the function, respectively.

in <filename> "func1" was multiple defined

func1 is defined … (or multiply).

in <filename> "func2"... sorry, can’t check indirect call

because func1 calls another function indirectly, the parameters cannot be checked.

This message is only displayed when the "-i" option is specified. This happens when a pointer to a function is used. For example, in the following case.

int (\*fp)(), func(); /\* fp is a function booster \*/

fp = func; /\* fp is used to turn func into … \*/

(\*fp)(1, 2); /\* This is an indirect call and cannot be checked. \*/

Other error messages will be in a common format as follows:

in <filename> "func1" calls "func2"; message

Here, the 'message' can be one of the following:

1. conflicting return type

The type returned by func2 does not match the type of the body of the function definition.

1. conflicting number of arguments

In the call to func2, the number of arguments does not match the definition.

1. <Nth> argument conflict

In the call to func2, the type of the N-th argument does not match the definition.

For the first, second and third arguments, "Nth''' becomes "1st", "2nd''' and "3rd", respectively.

1. undefined

Although func2 is referenced, it is not defined anywhere.

### CG (Code Generator)

The format of the CG is as follows

CG [options] filename

where filename is the intermediate language file (.tco file) output by CF. The file name can be specified as a drive name, path name and extension. If the extension is omitted, tco is used.

CG is the code generation phase of the compiler, and it reads the intermediate language file generated by CF as input and outputs the source file of the assembler. Figure 1 on page 21. If an error occurs during code generation, the .mac file will not be generated.

The CG output file is used as it is for the M80 macro assembler. That is, the process of the MSX-C compiler itself is up to the output of the source file of the assembler, then the macro assembler and the linker do the processing up to the generation of the actual executable file.

Press Ctrl+C or Ctrl+STOP to abort the process during code generation.

##### Options

|  |  |
| --- | --- |
| -k | At the end of code generation, it automatically deletes the intermediate language file (.tco file), which is the input file. |
| -l | full name generation.  Normally, CG generates only the first six characters of a global symbol for the assembler output, but this option makes the entire name of the global symbol valid. |
| -o [fileame] | The file is generated in the specified directory.  By default, the .mac file is created in the same directory as the .tco file. Using this option, you can create an intermediate language file with an arbitrary file name in an arbitrary directory.  filename specifies the output destination of the .mac file. An alphabet or ":" (colon) specifies the current directory of the drive, or a directory name ending in "\" specifies the current directory, or a directory ending in anything other than "\". If you omit filename, it becomes the same behavior as the default. No other options are available after this option.  -otest\ Outputs an intermediate language file in a directory named test.  -o\foo.asm Outputs an intermediate language file with the file name foo.asm. |
| -rN | N bytes are allocated for the symbol table region (N is a decimal number). |
| -u | hides the progress of the process.  This option specifies printing the name of the function currently being processed and a progress indicator, "." (period), ":"(colon)" or ";" (semicolon) output (see examples below). |

##### Error Messages

There are two types of error messages in CG. There is a message about the behavior of the command and the compilation results. In this section, we will describe each message in detail. For actual error messages, see ‎8.2 CG Error Messages.

###### Command Behavior Messages

This is a message about the operation of the CG command. For example, a message that the tco file is not found or the work area is missing. If such a message is displayed, CG stops the process and returns to the command prompt.

###### Compilation Messages

This is a message from the compilation of a tco file; CG will only generate a warning message (warning) as a result of the compilation.

Let's try processing the previous HEAD program (see ‎1.4) with CG. Since the intermediate language file is not needed, the .tco file is removed by specifying the -k option.

Example

A>dir head.\*

HEAD C 3802 89-01-17 2:53p

HEAD TCO 6888 88-01-17 2:54p << Intermediate language file

9K in 2 files 473K free

A>cg-k head

MSX-C ver 1.20x (code generator)

Copyright (C) 1888 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 file has been deleted and the assembly source file (.mac file) has been generated

A>

In this example, because the -u option is not specified, the progress of the process is displayed. The first thing displayed as the progress status is the name of the current function. The period displayed after it indicates that each statement in the function is in process and, when a colon is displayed, that the optimization path is in operation there.

### Assemble and Link

In the previous section, the MSX-C compiler itself is finished, and the assembler source file obtained as the output file (.mac file) is handed over to the M80 macro assembler, and the final executable file (.com file) is created by starting the L80 link loader. Section V and VI of Figure 1 in page 21. This is necessary.

ck.rel MSX-C standard kernel

clib.rel MSX-C standard library

crun.rel runtime routines

cend.rel trailing file

Continuing on from the previous section, let's take the creation of a HEAD program as an example to see how to create an executable file.

A>m80 =head ◄┘

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 << Generated of launching the Assembler

14K in 3 files 443K free

A>l80 ck,head,clib/ s, crun/s, cend,head/n/y/e: xmain ◄┘<< linker activation

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 << Generated object file

HEAD COM 6016 89-01-17 2:59p << Executable file

HEAD SYM 1152 89-01-17 2:59p << Symbol file

21K in 5 files 436K free

A>

This generates an executable HEAD program.

We won't go into the parameters of the linker here, but the symbol table file.

You can prevent the creation of (.sym files). And you can delete the assembler source files (.mac files) before invoking the linker.

# Chapter 3: Programing in MSX-C



## Specifications Differences against the (UNIX-C V7)

The C language does not currently have a "Standard C" terminology that is used in the international industrial standard. Many C implementations take the form of Kernighan & Ritchie's ''The Programming Language C'' or ANSI's proposed standardization. In this section, we will compare with a standard C implementation, C in UNIX Version 7 (hereinafter called standard C).

What is not included in this chapter performs the same function. In other words, all control structures such as switch statements, structures and unions, and all operators included in the language specification are thought to be superposed.

### Control by the preprocessor statement #

In standard C the preprocessor statements are used to extend the source files during the compiler pre-processing, but in MSX-C there are preprocessor instructions to control the compiler. The preprocessor statement # for this compile mode control is the pragma statement, which can control the following:

1. #pragma nonrec

Change the default function mode to non-recursive mode.

1. #pragma recursive

Change the default function mode to recursive mode.

1. #pragma pdp11mode

Compile in PDP-11 compile mode.

1. #pragma regalo

Starts the register allocation phase in the code generator.

1. #pragma nonregalo

Do not start the register allocation phase in the code generator.

1. #pragma optimize time

When speed is more important than code efficiency in object programs.

1. #pragma optimize space

Code efficiency is more important than the speed efficiency of the object program.

The statements other than #pragma pdp11 mode can be specified at any place in the program. These are explained in detail below.

#### Recursive and non-recursive modes

In standard C all the functions can be used recursively, but in MSX-C functions can be declared as non-recursive. To distinguish between recursive and non-recursive functions, two new reserved words recursive and nonrec are introduced. These reserved words are the function mode specifications.

It is called a child and is placed at the beginning of the function definition, as shown in the following example.

nonrec main

{

printf("hello, world\n") ;

}

recursive int

factorlal(n)

int n;

{

return ((n > 1) ? n \* factorial(n - 1) : 1);

}

int max(a, b)

int a, b;

{

return ((a > b) ? a : b);

}

Among the 3 functions defined in the above example, function main() is declared as a non- recursive function and function factorial() is declared as a recursive function. Also, the third function max() does not have a function mode specifier, so it is assumed to have a default function mode.

The default function mode is usually recursive, but it can be changed by the following preprocessor statement

#pragma nonrec

#pragma recursive

The former sets the default function mode to nonrec and the latter to recursive. In the above example, if these #pragma statements do not appear before the definition of the function max(), the function is treated as recursive (the default). The function max() is the closest specified function mode.

In MSX-C, non-recursive functions should be declared as nonrec. Thus, the compiler generates object code that is smaller in size and faster in execution.

In an application program, most functions are non-recursive, so the preprocessor statement

#pramga nonrec

Is a good idea to keep

#### PDP-11 compatible pull mode and MSX-C default mode

In MSX-C two different arithmetic transformation rules are applied, depending on whether it is MSX-C default mode or PDP-11 compatible mode.

In PDP-11 compatible mode, the same arithmetic transformation as in standard C is performed. However, char is treated as unsigned.

The default mode of MSX-C introduces two new types: the integer constant type (intconst) and the character constant type (charconst). Normally they are treated in the same way as the int and char types, respectively, but they are treated differently when the conversion rule is applied. No automatic type conversion from char to int is performed in this mode.

Unless otherwise specified, compiling is done in MSX-C default mode, and to compile in PDP-11 compile mode, you need to specify the following at the beginning of your program.

#pragma pdp11mode

This specification can only be made at the beginning of the program and only once in the program. You need to use PDP-11 compatible mode because there is no compatibility with object code compiled in MSX-C default mode.

The default mode of MSX-C supports byte arithmetic operations for efficient object code generation. This is because the operations between char types (TINY and BOOL are the same) are performed in bytes, not expanded to words, and therefore, the opcodes are slower and smaller for 8-bit processors.

In addition, the logical and relational operations (&&, ||, !, =, !=, >, <, >=, <=) are also treated like to char types in object code, which is an important feature.

However, the char type of MSX-C is treated as unsigned and can only represent the range from 0 to 255, so if an operation should be negative, or if there is an overflow or underflow caused by the operation, the result will not be reflected correctly. In such a case, cast to int or unsigned to perform the operation.

Here is a summary of the default mode arithmetic transformation rules for MSX-C. This is a "normal numerical conversion"

In the table below, if the type of either operand is (1), the type of the other operand is converted to (2) and the result is of type (3). The smaller the number on the left side of the table, the higher the priority of the operands.

|  |  |  |  |
| --- | --- | --- | --- |
| order of precedence | (1) | (2) | (3) |
| 1 | unsigned | unsigned | unsigned |
| 2 | int | int | int |
| 3 | char | char | char |
| 4 | intconst | intconst | intconst |
| 5 | charconst | charconst | charconst |

The following are the arithmetic transformation rules for PDP-11 compatible mode. However, all the operands of "char" are converted to int.

|  |  |  |  |
| --- | --- | --- | --- |
| order of precedence | (1) | (2) | (3) |
| 1 | unsigned | unsigned | unsigned |
| 2 | int | int | int |

#### Non-support for #if statements

The #if preprocessor statement is not supported in MSX-C. But in most cases, you can use #ifdef or #ifndef instead of #if statement. The MSX-C compiler does not generate code for parts of the program that will never be executed, so the same effect as the #if statement can be achieved by the if statement. For example, the following example program (1) can be rewritten as (2).

(1)

#if sizeof(FCB) != sizeof(char[37])

OpenMSX();

#else

OpenOTHER();

#endif

(2)

if (sizeof(FCB) != sizeof(char[37]))

OpenMSX();

else

OpenOTHER();

(2) is actually either OpenMSX(); or the OpenOTHER(); Only one or the other is generated.

#### Internal control of the compiler

This section explains how to control register allocation and optimizing of variables by the compiler.

The MSX-C compiler supports register allocation of variables, so it generates efficient object code. In standard C, … only one of the variables of the storage class register is placed in a register. (How many register variables are placed in the register depends on the model). On the other hand, MSX-C compilers automatically allocate registers optimally by analyzing the program data. So, the programmer need not to worry about which variable to declare as register. Since the code generator decides which variables are assigned to registers, register declarations have no special meaning and are completely equivalent to auto (it is possible to declare register).

For each function in the program, the first 16 variables of a simple type with auto or register are candidates for register allocation. The simple types are int, char, unsigned and pointers.

In C it is possible to retrieve the address of a variable, but in the case of a variable to be assigned to a register, there is a problem. This is because there is no such thing as an "address" of a register.

For example:

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

If 100 is entered in line 4, 100 is assigned to variable n. When the variable is addressed (line 4), the value of the variable is placed in memory, not in a register, before and after the context in the program. Therefore, when passing a variable grudge number, the value is opened to memory, and therefore, it can be caused by register allocation. There are no problems with this.

Now let us consider the following program:

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

In the case of this program, the value of n displayed in line 12 must be 100. If you try to compile and run the program, you will see "n = 10". That is, the value of n remains 10, even though \*p indirectly references the memory location of n and changes it to 100. To find out why, let's look at the list of generated assembly languages:

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

As you can see from the list, the value of n is taken in both the register DE and memory (pointed to by HL), and the value of n in memory is set to 100 in lines 13 through 15, but the compiler ignores this and pushes the value of n in the register DE to printf(). It is passed to you. Therefore, it is displayed as "n = 10".

This is a side-effect of the absence of the "address of the register", as mentioned above. However, such examples are rare and can be rewritten in other expressions, but if you really want to do this, you must prohibit automatic allocation to registers and specify that all the variables should be placed in memory. To do so:

#pragma noregalo

before any function that you want to prohibit register allocation, and add

#pragma regalo

at the end of the function before compiling it. Although this specification is slightly less efficient than register allocation, the compiler puts all the variables in this function in memory, so the above example is well handled.

The user can control the compiler's optimization by using the following two preprocessor statements.

#pragma optimize time

#pragma optimize space

The former is the speed specification of the object program. That is, you can specify it to speed up the execution, even if the target program is a little bigger.

The latter is a size-oriented specification of the object program. In other words, to reduce the size of the object program, even if the execution is a little slower.

One example for the previous preprocessors, is that the speed can be improved in some areas, such as keystroke requests from the user.

Another example, the efficiency can be improved by using the former in areas that use a lot of loops and numerical operations.

### Functions Declaration

#### Implicit Libraries

In standard C, an undeclared function is assumed to be a function that returns type int, and a parameter in the parameter list that has not been declared is assumed to be of type int (implicit declaration). However, In order to encourage a good programming style of "always declare before using all identifiers", MSX-C has been designed to prevent these implicit declarations from being made.

We are sure you'll be able to notice that.

All declarations of the standard library functions are included in the header file stdio.h, so you can use them without having to declare them. Therefore, all user programs must include the stdio.h header file at the beginning of the program using the #include keyword.

However, these implicit declarations are made by the -f option of the parser (CF)

compatibility with standard C is preserved.

#### Variable and Fixed Number of Parameters

There are two types of functions in MSX-C, variable parameter functions and fixed parameter functions. The former is a function with variable number and type of parameters, and the latter is a function with fixed number and type of parameters, a concept that is new in MSX-C. According to this classification, all functions in standard C are essentially variable-parameter functions, even in the case of MSX-C, e.g. printf(), scanf() is an example of variable parameter function.

The concept of fixed parameter functions was introduced in MSX-C because of its object efficiency. That is, while the variable parameter function uses a stack to pass all of the parameters, the fixed parameter function passes the first three parameters in a register, which results in a significant efficiency gain.

In MSX-C all the same function declarations as in standard C are considered to be fixed-parameter function declarations. Although the functions in standard C are variable parameter functions, in practice most of the functions have fixed parameters. Therefore, compatibility issues do not arise.

On the other hand, a variable parameter function is not portable, because it depends on the target machine (mainly on how to access the parameters) (it is the variable parameter function itself that is not portable, but the function that calls it is portable).

#### Definition of the Variable Parameter

To distinguish between variable and fixed parameter functions, MSX-C uses the declarator and the abstract declarator with the following syntax:

Declarator:

Identifier

(Declarator)

\*Declarator

Declarator ()

Declarator(. )

Declarator [constant expression opt]

Abstract Declarator:

emptiness

(Abstract declarator)

\*Abstract declarator

Abstract declarator ()

Abstract declarator (. )

Abstract declarator [constant expression opt]

Declarator. and Abstract Declarator(.) has been added to represent the variable parameter functions. In contrast, the conventional notation Declarator() and Abstract declarator() indicate a fixed parameter function.

Such notation for variable parameter functions is used in the case of declarations, casts and sizeof (type). For example, the following declaration defines a variable parameter function func() and a pointer to the variable parameter function fp.

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

Such a declaration must be made before using the variable parameter function. Some of the standard library are variable parameter function.

execl(), execlp(), fopen()

printf(), fprintf(), sprint()

scant(), fscanf(), sscanf()

However, the declarations for these functions are stored in the header file stdio.h and string.h, and the call to the variable parameter function is the same as for the fixed parameter function.

Since they are exactly the same, users who do not write variable parameter functions in H will be able to use them without having to worry about the low-level library. Next, we will describe how to write a real variable parameter function.

Before defining the body of the function, the function must be declared as a variable parameter. The typical definition of the variable parameter function func() is as follows:

int func(. ); /\* You must make this declaration \*/

int func(nargs, args)

int nargs, args;

{

......

}

In the example above, the function func() is declared to be a variable parameter before the body

definition.

int func(.);

If one forgets to use func(), then func() is a fixed parameter function with two int parameters, nargs and args.

When a variable parameter function is called, in addition to the actual parameter, another implicit parameter is passed. This parameter is called nargs and holds the actual number of parameters. nargs allows the variable parameter function to know how many parameters were actually passed by the caller. The variable parameter function can only know the number of parameters that are passed, not the type of each parameter.

To access the real parameter passed by the caller, another parameter args is declared, the first parameter (if it exists) is placed at the same address as args. The second and subsequent parameters are stored in memory in the direction of the increasing number of addresses, i.e. the first parameter is the args itself. The second parameter is the third parameter is stored on [args address + 2], the third parameter is stored on [args address + 4] and soon.

The address where the parameters are stored can be calculated as shown in the following table.

|  |  |
| --- | --- |
| Actual parameters | Address |
| One | Address of args |
| Second | Address of args + 2 |
| Third | Address of args + 4 |
| n | Address of args + (n-1) \* 2 |

The real parameters can be accessed by the addresses shown in the table above. For example. To refer to the first and second parameters as char, respectively, do the following:

(char) args : 1 Refers to the second parameter as a char

(char) \*(&args + 1) : 2 Refers to the second parameter as a char

In this example, args is declared to be of type int, so &args becomes 'a pointer to type in t'. Therefore, in the expression &args+1, the constant 1 is multiplied by sizeof (int\*) (i.e., twice) and the address [args address+2] is referenced. In the same way, parameters of other types can be accessed.

Here is example of a simple variable parameter function. The function sum() returns the sum of (variable number of) int parameters. The function max returns the largest variable number of TINY(char) parameters.

#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) ;

}

Other examples of variable parameter functions are included in the standard library functions such as printf() scanf() fopen().

### Other

#### float, double and long Support

Currently the float, double and long types are not supported by MSX-C. However, for compatibility with Standard C, these identifiers are reserved words and cannot be used by the user as variable names.

#### Bitfield Support

Bitfields in structures and unions are not supported. It should be replaced by appropriate bitwise logical operations.

#### Constraints on Constant Expressions

A constant expression in the next position cannot contain size of a unary operator immediately after case (as a label for case) defining array size in array declarations.

However, there are no such restrictions on the constant expressions in the initialization elements of the variables. This restriction is to make the intermediate language (T code) independent of the target machine's hardware.

#### Structure and Union Member Names

In MSX-C, the use of member names of structures and unions is different from standard C. With standard C it possible to use for two or more separate structures or unions the same member name only if they have the same offset and type.

For example, consider the following declaration:

struct node {

char \*word;

int count;

struct node \*next;

} pool[1000],\*p;

struct noad {

int atr;

struct noad \*next;

} table [10], \*q;

In standard C such declarations are not valid. This is because the member name next appears in two structures node and noad, but the types and offsets do not match. However, in MSX-C this is a valid declaration.

p->next

Note that p denotes a member of the struct node type (because p is a pointer to the struct node type). In this way, the members of the structure are more tightly typed so that the correct members can be selected. That is, the ".'' or " ->" operator, the left side of the operator must contain the structure itself, including the right-side members, and a pointer to it.

The ability to use the same member with the same name in different structures and unions is not a specific extension to MSX-C. It is a common feature of the new C compiler, so there is almost no loss of compatibility.

#### Mutual Substitution of Pointers and Integers

It is not possible to perform a non-cast shortcut operation between a pointer and an integer. If the pointers point to different types, they still need to be casted.

For example,

int i;

char \*p;

int \*q;

The following four statements are all considered errors:

p = i;

if (p == i) ... ;

q = p;

if (p == q) ... ;

The first two are assignments and comparisons between pointers and integers, they are very similar to the latter two, both p and q are pointers, but the types of the pointers are different, so the same error occurs. This error can be avoided by adding a cast or by specifying the -t option to the parser (CF). In the case of a cast solution, for example, it looks like the following:

p = (char \*)i;

if (p == (char \*)i)... ;

q = (int \*)p;

if (p == (char \*)q)... ;

On the other hand, type conversion between constants and pointers is allowed. This is because C has a practice of using the constant 0 (NULL) as a 'pointing-nowhere' pointer.

For example:

char \*p;

if (p != 0)

p = 0;

Does not result in an error.

Thus, MSX-C does not allow confusing the int type with a pointer. This may seem like a hassle for people who write loose programs. However, strict type checking is a recommended programming style, and it can reduce the number of bugs in your program.

## Linking to Assembly Language

If you want to link with the assembly language, you need to keep in mind the language and the hardware characteristics of MSX-C. In particular, the length of data and the order in which the data is stored.

For a 16-bit data type, the lower byte or the first byte is stored first, and the next byte or the higher byte is stored at the next position. That is, the internal representation of the 16-bit long treated as is the same as the standard data representation of the 8080s. The length of the data is shown in the table below, please note the following points.

In MSX-C, the pointer is 16 bits long and therefore has the same length as the int type

short and int types are treated as exactly the same the length and range of the basic tet a type in the table

|  |  |  |
| --- | --- | --- |
| model | length | range |
| char | 8 | 0 ~ 255 |
| short | 16 | -32768 ~ 32767 |
| int | 16 | -32768 ~ 32767 |
| unsigned | 16 | 0 ~ 65535 |

### Passing Parameters

The way of passing parameters to the assembler is different for the fixed and variable parameter functions, respectively.

First, in the case of an ordinary function (a fixed-parameter function), the first three parameters are added to the register. The first parameter is stored in the A-register for char, otherwise it is placed in the HL-register pair. The second parameter is placed in the E register for char, otherwise in the hair of the DE register. 3 H is placed in the C register for char, otherwise in the BC register pair. The fourth and subsequent parameters are stacked on the stack in reverse order. In addition, the return address is piled at the top of the stack. All parameters on the stack are 2 bytes long. In the case of char parameter, the value is set to the lower byte and the higher byte is not used.

When PDP-11 conch mode is specified, the treatment of parameters of char type is different. In this mode, a parameter of char is automatically converted to an int type (the upper byte is set to 0) and treated the same as any other type.

These rules can be summarized in the following table.

Table How to pass parameters (fixed parameter function)

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Parameters | 1st H | 2nd | 3rd | 4th | 5th |  | nth (n>3) |
| 1-byte char | A | E | C | (SP+2) | (SP+4) | ... | (SP+2n-6) |
| 2-bytes type | HL | DE | BC | (SP+2)  (SP+3) | (SP+4)  (SP+5) | ...  ... | (SP+2n-6)  (SP+2n-5) |

Note In the direct PDP-11 compile mode, the parameters of type char are automatically converted to int and treated as the same as those of other types.

Example

function call

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

sets the parameters as follows

|  |  |  |
| --- | --- | --- |
|  | Default Mode | PDP-11  compatible mode |
| 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 |

On the other hand, the variable parameter function has a different effect.

When the variable parameter coefficients are called, all the parameters are stacked in reverse order in the stack. The top of the stack is piled up with the return address and the HL register pair contains the number of parameters. In the default mode of MSX-C, each parameter occupies 2 bytes.

A parameter of type char is deactivated in the lower byte and the value of the second byte is not defined and is not used.

In the PDP-11 compile mode, parameters of type char are automatically converted to int (the upper byte is set to 0). Parameters of other types are stored according to the standard rules of the 8080 family of processors (i.e., lower bytes are stored first, then higher bytes).

In order to link with the variable parameter functions you have to make a declaration like the following,

int sub(. );

These rules can be summarized in the following table.

Table How to pass parameters (variable parameter function)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Parameters | nargs | 1st | 2nd | 3rd | ... | nth |
| 1-byte char | HL | (SP+2) | (SP+4) | (SP+6) | ... | (SP+2n) |
| 2-bytes types | HL | (SP+2) (SP+3) | (SP+4)  (SP+5) | (SP+6)  (SP+7) |  | (SP+2n)  (SP+2n+1) |

Example

function call

sub(l.'A', -3);

sets the parameters as shown below. However, in advance the following statement is assumed to be true

int sub(. );

|  |  |  |
| --- | --- | --- |
|  | Default mode | PDP-11  compatible mode |
| 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 |

### Returning Values

The value from a function can be returned by inserting the value into a register when returning to the caller from the function. If the return type of the function is char, the value is put into the A register, otherwise it is returned in the HL register pair. To return more than one value, pass a pointer to a variable as a parameter, as is done in a normal C program.

### Reciprocal Calls to Assembly and C Routines

In the case of calling a routine between the assembler and C, the caller of the function and the subroutine called by the function must follow the rules described below, respectively. See the previous description of the rules for passing parameters to and from the functions.

#### Caller Processing

First, the caller evaluates the parameters and puts them in a register or on a stack. See section 2.1 How to pass parameters to the register or to the stack.

Next, the caller actually calls the function. Normally a CALL instruction is used. When you come back from the function, discard the parameters stack (using the POP instruction) to make sense of the stack. If all the parameters are assigned to the register, no action is required. Care must be taken not to break the function value returned to the HL-register pair or A-register when performing POP.

The function return value (if any) is available in the HL-register pair or A-register. Which register is returned has been described in the previous section.

#### Called Function Processing

The called function must not break the value of the stack pointer (SP), the value of the stack pointer must be the same when entering and exiting the function. It is the caller's responsibility to POP (discard) the parameters in the stack (if any). The called function can change the value of the parameter (either in the register or on the stack). However, the parameter delivery between C functions is a copy of the value, so it is not a temporary instead it is a permanent process.

Note that changing the value of the parameter does not have any effect on the real value of the parameter. The value of the function is placed in the HL register pair or A-register, depending on the type of the function return. Please refer to the previous section to see which register is used when calling assembly routines directly from C language.

In the following example, the assembly routine isdigit@ is declared as a function that returns BOOL (char), so the function value is returned to the A-register. The caller's example() calls isdigit@ with the char parameter c in the A-register.

Caller (C language)

typedef char BOOL; /\* Define BOOL type as char \*/

example()

{

BOOL isdigit();

char c;

......

if (isdigit(c))

......

}

The function to be called (assembly language)

public isdiglt@

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

Calling a C function from an assembly routine

In this example, the standard library functions printf() and puts() are called by the assembly routine. We won't go into printf() and puts() here, but note that printf() is a variable parameter function, so the parameters are stacked on the stack.

In this example, there is only one parameter, but when there are two or more parameters, they are in reverse order when they are stacked on the stack (in order from right to left parameters). The number of parameters is put into the HL register. (in this example, it is placed in the HL register).

The function pops once after, the reason for this is to discard the parameters on the stack. It is the caller's responsibility to discard the parameters from the stack.

On the other hand, puts() is a fixed-parameter function with constant number and type of parameters. The address of the string is passed to the HL register pair.

extrn printf@

extrn puts@

example::

Id hl,msg ; push address of string

push hl

ld hl,1 ; load # of parameters

call printf@ ; formatted output routine

pop hl ; pop the parameters off the stack

ld hl,msg ; load address of string

jp puts@ ; string output routine

msg: defb ‘Hello, world’,0ah, 0

If you name the file [example.rel], the link is as follows

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

### Number of Characters with Identifiers

The name of the identifier is valid for up to 6 characters from the beginning of the name for other modules. For MSX-C, symbol names with more than six characters can be specified with the -l option of the code generator (CG).

The code is output as simple source code, but when it is passed from the assembler to the linker, the part that exceeds 6 characters is truncated. Therefore, you must use uniquely identifiable identifier names of up to six characters.

Note

In the standard library, the distinction between fclose() and fcloseall() is defined by #define in the header file stdio.h

Since the first 6 characters for the two functions are "fclose", both would be ''FCLOSE' if they are passed to the linker, which would cause a bug. So the name passed to the assembler and linker for fcloseall() is changed to a different name. The parser (CF) is case sensitive and identifies seven or more characters which is important.

### External Symbols

Because the assembly routines and the C routines are separate modules, they need to declare the use of external symbols when using global symbols in each other's modules.

For example, when you use C symbol in the assembly routine, you need to declare it as extrn. Also, if the global symbol allows references from other routines, it is declared public. In addition, when linking C and the assembler, "@" is required after the name of the external symbol of C.

(See examples above). Also, the underscore ("\_") is replaced by '''@''', for example, "main" should be referred to as "main@" and "\_exit" should be referred to as "@exit@".

## Porting a Standard C Source File

Currently, there are many books published for learning C. However, many of these seem to be targeted the C processing system running on X or MS-DOS. In this section, we will list the items necessary to make the programs in those books work properly on MSX-C.

### Backslash "\"

A backslash is an escape character used in UNIX, for example, in command lines and C source files. The use of escape characters allows more characters to be represented in a limited character set. However, this character is not defined in JIS 8-bit code; JIS defines the yen symbol "¥" instead.

K&R's "C programming language" assumes the UNIX environment and uses a lot of backslashes in the C language. To do the same thing with MSX-C, you can substitute "¥" for ''\centric''. The character "\n'" for a newline character is written as "¥n". See the following response.

"\n" "¥n"

"\t" "¥t"

"\0" "¥0"

"\\" "¥¥"

It is not only MSX-C that uses the yen sign instead. All JIS 8-bit computers using character sets must be deactivated, regardless of the language.

### #include <stdio.h>

The C compiler on UNIX means to use the standard library for functions that are not in the source files. The same thing can be said about the existence of an external (extern)

It is a good idea to use the same method. However, in MSX-C, if there is no declaration at the time of compilation, an error will occur and you cannot continue processing. For this reason, you must include the stdio.h header file at the beginning of your program when you create a program in MSX-C.

#include <stdio.h>

... /\* Write program normally \*/

### char and int types

In the standard C, there is no need to make a distinction between char and int types when calling a function as an operation or an argument. However, to take advantage of the characteristics of 8-bit CPUs, MSX-C makes a strict distinction between char and int types. The first argument especially important when passing arguments to a function. It is char.

The reason for this is that the registers that passes values to the int type and the int type are completely different. More on this in detail 3. See "2.1 How to pass parameters". Here, we show you how to solve the problem. First, look at the C program in question.

main()

{ /\* Program for Standard C \*/

int c;

while ((c = getcharO) != EOF)

putchar(c);

}

This program outputs the input from the standard input to the standard output. At the first glance, it seems to be OK, but there is a big hole to fall into. It has a function that output a character on the standard output, putchar(). The first argument of the function must be of char type. However, in this program, the int variable c is passed as it is. That is, passes an int value. Since the register on the passer side and the register on the receiver side are different, we can't see what is displayed when we compile and execute the program as it is. That is, it changes an int value to char using a cast.

#include <stdio.h>

main()

{ /\* Program with modifications for MSX-C \*/

int c;

while ((c = getchar()) != EOF)

putchar((char)c);

}

The change is that the argument of putchar() is changed from only "c" to "(char)c".

This kind of mistake (argument types do not match) can be reliably removed by the FPC command.

### Passing Arguments to Variable Parameter Function

A variable parameter function is a function that does not have a fixed number of arguments to pass to the function. A typical variable parameter function is printf(). The printf() function sometimes prints different character codes of char or variable other than what is expected. The source of the problem is the following:

main()

{ /\* Program for Standard C \*/

char c;

c ='A';

printf("Char code of 'A' is %d.\n", c);

}

This program tries to display the character code of the letter "A". When you actually compile and run it, you can see that the following will be displayed:

char code of 'A' is 2625.

The value 2625 is outside the character code range of 0 to 255. This happens for the following reasons:

1. Variable parameter functions always take arguments of type int.
2. When a char argument is passed to a variable parameter function, only the lower bytes of int are set, and the upper bytes are undefined.
3. The conversion characters "%d", "%x", etc. in printf() are assumed to be given as arguments of type int.

That is, "%d", which prints an integer, printed an unexpected value because the upper byte is undefined. The reason for this is that the type char is passed directly to printf(). This problem can be solved by modifying the program as follows:

#include <stdio.h>

main()

{ /\* MSX-C modified program for C \*/

char c;

c ='A';

printf("Char code of 'A' is %d.\n", (int)c);

}

The only change is that the "c" argument passed to printf() is changed to "(int) c".

Make sure to cast the value to an int when you pass it. The problem only occurs if the conversion specification is trying to print an integer ("%d", for example) and char is printed. In the case of "%c", no problem occurs. Note that this problem cannot be detected by the FPC command.

### Function Declaration (Forward Reference to a Function)

If undefined or undeclared functions appear during compilation, standard C compiles the function as returning an int type, but MSX-C generates a compile error. If main() is at the beginning of the file, the function used there must be declared before main(). For a detailed explanation, see 1.2 C "Function declaration" for a detailed explanation of this.

## Compatibility Between MSX-C Ver.1.1 and Ver 1.2

This section explains the differences between Ver. 1.1 and Ver. 1.2.

### Command Changes

1. All the commands attached to Ver. 1.2 are only available in MSX-DOS2. In the case of MSX-DOS1, it displays "This program needs MSX-DOS2" and exits.
2. You can now specify drive names, path names and file extensions for compiling and checking. You can do the same for the files created by the
3. In CF, CG and MX, the designation after the "-e" or "-o" switch is a single letter of the alphabet, or a colon, if it contains a colon, or a directory name if it ends in "\". Creates a file in the directory, or in a file name if it ends with a letter other than "\".

#### CF Specific Changes

1. The default directory for include files can now be specified by the environment variable INCLUDE. This allows you to pay less attention to the existence of header files and so on. So, you won't be bothered by the split header files. (See Section 4.2.1 "Splitting the header file" for details on splitting the header file.
2. Kanji characters are now recognized correctly even if they are used in strings. However, you can specify the switch "-j" at the compile time to prevent the characters from being changed. However, mimeographs and half-width hiragana cannot be mixed.
3. The "-e" and "-o" functions have been extended and changed. The first point is that no other switches can be specified after these switches.

A>cf -eaob file ◄┘

This specifies that error files are created in drive A and intermediate language files are created in drive B. If you execute CF for Ver. 1.2, the intermediate language file is placed in the same directory as the source file, just by creating a file name of aob.dia in the current directory.

A>cf -ea -ob file ◄┘

It works the same as Ver.1.1. This means that the switch must be divided into different functions. However, this restriction only means that no other switches can be placed after these switches, but other switches with "-e" or "-o" can be placed after them. On the other hand, this restriction is not a problem because it is easy to understand and extended.

In addition, "-e" and "-o''' are not only for specifying the drive, but also for specifying the path name, file name, and extension.

It is now possible to do it with in addition, if you don't specify a file name immediately after the switch, the witch is interpreted as if it didn't occur, without an error message.

#### CG Specific Changes

1. The same thing applies to the "-o" switch in CF as it does to the "-o" switch in CG. You can also specify a drive letter name, path name, file name, extension, and no other switches after it. See the CF section for more details.
2. When the error occurs during CG, the assembler file that is the output file is not generated. This is useful for determining the behavior of CG in a batch, depending on the presence or absence of an assembler file.

#### FPC Specific Changes

1. If a file checked by FPC does not exist in the current directory, it looks for it in the directory where the fpc.com file is located and use it if it can be found. This is to eliminate the need to copy frequently used files, such as tco files, to the current directory and to specify path names. fpc.com and lib.tco are paired and placed in the same directory, even if they are not in lib.tco or the current directory, FPC can be executed as follows:

A>fpc file lib ◄┘

#### MX Specific Changes

1. MX refers to the MX-specific files in the command. In Ver. 1.1, these files are not executed if they are not in the current drive when MX is executed. But in version 1.2, even if it is not in the current directory, if you can find the directory where the mx.com is invoked, you can use it. The contents of the arel.bat, crel.bat doesn't change much, so it is better to put it in the same directory as the mx.com file. If you need to change arel.bat or crel.bat, you can place the changed arel.bat or crel.bat in the current directory and MX find it first, so it will use the changed one.
2. The "-o" switch has been extended to MX as well, and the main part of its operation is to create a split file. The "-o" switch specifies where to create the segmented file. This is a directory name that must end with a "\". The reason for this is that we are creating multiple files and they need to be directories. If it is not terminated by "\", then the "-l" switch also has a meaning if it is specified. With the "-l" switch, MX outputs to standard output the steps for creating a library file. If the file name does not end in "\", it is the name of the library file at this time it is used. Therefore, you can only create a library file in the same directory as the split file. The LIB80 that creates the library file at this time must be the MSX-DOS2 version.

### Changes to the Standard Library

1. The executable linked to the standard library of MSX-DOS2 is only available on MSX- DOS2. In the case of MSX-DOS1, it exits with the message "This program needs MSX- DOS2".
2. Header files have been split. However, as before, all the library functions can be used by simply including stdio.h at the beginning of the program.
3. The header file bdosfunc.h is now Ver.1.2 and the function call name is changed to Daifuku. The purpose of this is to match the function call names in MSX-DOS2 Reference Manual. However, in order to correspond to the programs you have developed so far, you can use the function call names from Ver.1.1. 1.1 function call names to accommodate programs developed so far. It defines a constant (macro) called "HEADbdosfuncver11" before including bdosfunc.h. Now you can compile it as is.

Example

#define HEADbdosfuncver11

#include <bdosfunc.h>

1. The sequence function (sequential execution of commands), which was in the standard kernel of Ver.1 is gone. Redirects and pipes are now handled by COMMAND2. argv[0] is now passed to the command name.
2. There are now three types of high-level input/output buffering methods. This is partly dependent on MSX-DOS2 function calls.
3. Some functions have been added and modified to operate on hierarchical directories, conglomerate variables and file handles, which are features of MSX-DOS2.
4. Functions (macros) for handling Kanji characters have been added. Two additional functions are added, but they should be sufficient for normal programming.
5. A new string manipulation function has been added to the string manipulation functions that allows the user to specify the maximum number of characters to be manipulated.

## Methods for Processing Kanji Characters

In MSX, Kanji characters are processed in Kanji mode and can be used to input data from a keyboard, display on the screen, and output to a printer.

All of these processes are done by the Kanji driver, so it won't work unless the Kanji driver is running. If you have a front-end processor, you can input Kanji characters by using the bunsetsu conversion method. The dependence on the Kanji mode is much less for Kanji processing that does not involve input/output.

### Internal representation of Kanji characters in MSX

It is necessary to understand the internal representation of Kanji characters in order to handle them. Otherwise, ANK (1-byte alphabets, numbers, and katakana) and Kanji characters are indistinguishable. In MSX, the internal representation of Kanji characters follows the Microsoft Kanji Encoding Scheme. This is generally called the NFTJIS code. The Shift-JIS code consists of 2 bytes, just like other Kanji codes. However, the compatibility with ANSI (easy to distinguish) and the efficiency of data storage are improved because there is no Kanji shift code required to handle JIS Kanji. The relationship between the JIS code and the code is shown in the following figure.

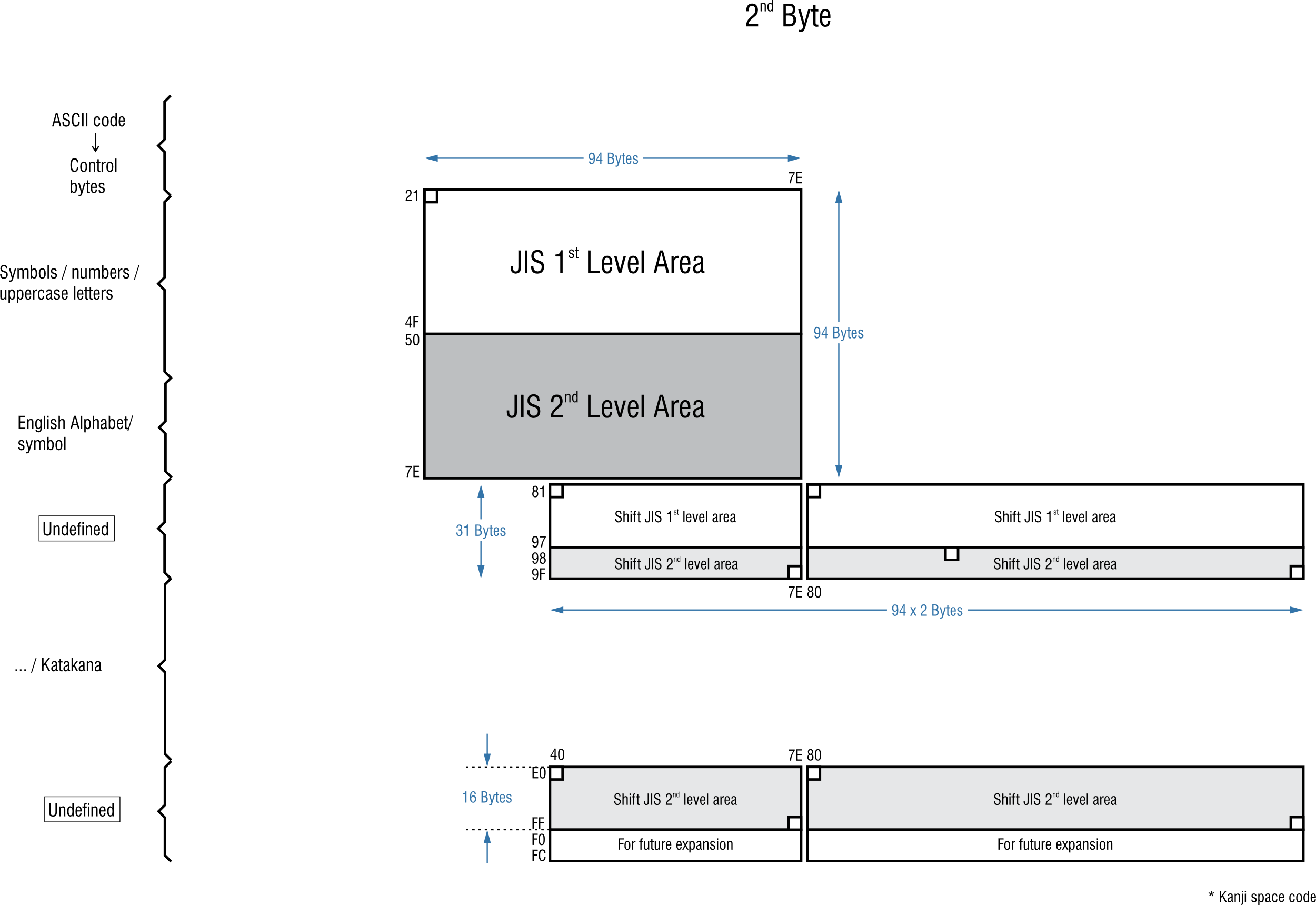


Fig. Relationship between JIS code and shift JIS code

In Shift-JIS code, the undefined part of 1-byte characters defined by JIS is assigned to the first byte of Kanji code, so you can find out whether it is ANK or the first byte of Kanji code by checking the first byte.

The internal representation, shift JIS code, is used in MSX-BASIC as well as in MSX-DOS. It is also the same as MS-DOS, the standard DOS for 16-bit computers. Therefore, it is possible to create a kanji file created with MS-DOS and display it with MSX- BASIC.

Since most of the codes used for data exchange in computers, including MSX, are JIS codes, here is a conversion function between Shift JIS codes and JIS codes described in MSX-C as a reference. jis2sjis() is the conversion function from [Shift] JIS code to JIS code.

Each argument is an unsigned, nftJIS or JIS code.

#include <stdio.h>

/\*

convert a shift-JIS kanji character to JIS

Shift JIS code - > JIS code

\*/

unsigned sjis2jis(sjis)

unsigned sjis;

{

TINY l, h;

h = (sjis >> 8) - 0x71;

if (h > 0x2e)

h -= 0x40;

h = h \* 2 + l;

l = (sjis & 0xff) - 0x1f;

if (l > Ox60)

l--;

if (l >= 0x7f) {

h++;

l -= 0x5e;

}

return ((unsigned)h \* 256 + l);

}

/\*

convert a JIS kanji character to shift-JIS

JIS code -> Shift-JIS code

\*/

unsigned jis2sjis(jis)

unsigned jis;

{

TINY l, h;

h = (jis >> 8) - 0x21;

l = (TINY)(jis & 0xff) + ((h & 1) == 0?(TINY) 0xlf : (TINY)0x7d); if (l > Ox7e)

l++;

h = h / 2;

if (h > 0x1e)

h += (TINY)0x40;

h += 0x81;

return ((unsigned)h \* 256 + l);

}

### Distinguishing Between Kanji Modes

In MSX, you can't tell if it's really a kanji by checking just one byte. Which is the first Byte. The reason for this is that the range of hiragana is the same as that of the first byte. This distinction is made according to whether the command is written in kanji or hiragana. It is better to divide them as follows.

1. When you use Kanji (without using 1-byte hiragana),

When a command switch declares that it uses Kanji characters, or

When no hiragana is declared for a command switch, or

in Kanji mode

1. What happens when you use 1-byte hiragana (no kanji)? Or

if the switch is not declared to use Kanji characters in the command, or

When a switch in the command declares the use of hiragana, or

When not in kanji mode

The first condition in (2) is used for commands that don't use many Kanji characters (in the case of Kanji characters), the second condition is used for commands that are mainly composed of Kanji characters. The advantage of this is that you can specify these classifications regardless of the screen's kanji mode. However, if you change the default of kanji handling by command, there will be confusion as to whether to add or remove the switch for kanji, so it is better to control the switch. One solution is to make the command accept both switches that use both hiragana and Kanji characters, and the MSX-C CF command is configured to use hiragana by default, and requires the '-j' switch to use Kanji characters.

The third condition is that all the commands are judged automatically. The command must know what is the imitation mode for the placeholder. However, by using MSX-DOS2's function call, you can handle it without knowing the kanji mode directly (using \CHKCHR(5DH), for example).

Other than this, you can set the current kanji mode as the default and decide for both kanji and hiragana, or you can use 'don't decide by looking at the environment variables'.

### Processing of Kanji Characters

In this explanation, we assume that the characters to be processed include ..., but not a single byte of hiragana.

In the case of Shift-JIS code, you can't find out what character is the first byte of a string. Basically, the process is as follows. A single byte from the beginning of the string iskanji(), and if it is FALSE, then the

If it is TRUE, the character is the first byte of the kanji, so it gets another byte (the second byte) and processes it as a 2-byte pair, which is necessary for the kanji. Repeating this procedure and using Kanji characters

Can manipulate strings.

#include <stdio.h>

/\* Capitalize the lowercase letters of ANK in string s (Kanji version) \*/

char \*jstrupr(s);

char \*s;

{

char \*head = s;

char c;

while(c = \*s){

if(iskanji(c)) /\* 1 byte check \*/

/\* It's a kanji character, so it advances by one byte. \*/

s++;

else

\*s = toupper(c); /\*ANK is capitalized if necessary \*/

s++; /\* Make it point to the following characters \*/

}

return (head);

}

The next function, chkctype(), nthctype(), tests whether a byte is an ANK or a lament. chkctype() passes the previously searched character type to find out what is the current character. This is used when a program needs to know the type of a character one by one. nthctype() is used when you want to know the type of a character in the middle of a string.

#include <stdio.h>

#define CT\_ANK (TINY)0 /\* ANK \*/

#define CT\_KJ1 (TINY)1 /\* The 1st byte of a kanji character \*/

#define CT\_KJ2 (TINY)2 /\* The 2nd byte of a kanji character \*/

#define CT\_ILGL (TINY)-1 /\* Illegal character value -1 \*/

TINY chkctype(c, type) /\* Examine one character type \*/

char c; /\* Pass the character you want to look up \*/

TINY type; /\* Pass the character's type to the user \*/

{

switch(type) {

case CT\_KJ1:

if(iskanji2(c))

type=CT\_KJ2;

else

type=CT\_ILGL; /\* Illegal 2nd byte of Kanji character \*/

break;

/\* The second kanji byte is followed by ANK \*/

case CT\_ANK: /\* ANK \*/

case CT\_KJ2: /\* the 2nd byte of Kanji character \*/

case CT\_ILGL: /\* the 2nd byte of incorrect Kanji character \*/

default:

if(iskanji(c))

type = CT\_KJ1; /\* it was the 1st byte of kanji character \*/

else

type = CT\_ANK; /\* it was ANK \*/

break;

}

Return (type);

}

/\* what is the nth byte of the string s \*/

TINY nthctype(s, n);

char \*s;

int n; /\* specify n >= 0 \*/

{

TINY type=CT\_ILGL;

/\* the string is prefixed with CT\_ILGL \*/

/\* return CT\_ILGL when n<0 \*/

/\* if CT\_ILGL or '\' is set, it is ignored \*/

while (n-- >= 0)

type = chkctype(\*s++, type);

return (type);

}

# Chapter 4 MSX-C Library

In this chapter, the features of the MSX-C compiler are described in detail. See Chapter 6 "Standard Library Function Reference" for details on each function.



## MSX-C Canonical Types

The basic types supported by the current MSX-C are polymorphic types, character types, and combinations of those types, but the standard ANSI uses some useful basic types, such as void. Standard types are defined in advance, such as BOOL, FD, STATUS, size\_t, TINY, and VOID are defined in the type.h header file. type.h is automatically read by all other header files, so the user hardly needs to be aware of it. FILE is defined in stdio.h and FCB, FIB, LONG and XREG are defined in bdosfunc.h. Refer to "4.2.3 Header file contents" for the values used for each type.

### BOOL Type

This type is a detour type for binary logic (it can have only two values, such as TRUE and FALSE, YES and NO etc.). For example, the isdigit() function always returns either numeric (TRUE) or non-numeric (FALSE) characters of the parameter (BOOL function). If you want to preserve the value of the function, it is better to use variables declared with BOOL type. The BOOL type allows you to write programs that are easy to understand intuitively by adding variables and functions to conditional expressions such as if statements.

The BOOL type is defined in the type.h header file.

STATUS SameCase(s) /\* put it in the first alphabet case \*/

char \*s;

{

BOOL upper;

while(\*s && !isalpha(\*s)) /\* search for the first alphabet \*/

;

if(!\*s)

return(ERROR); /\* Alphabet was not found \*/

upper = isupper(\*s);

while(\*s) {

if(upper) /\* The first alphabet was capitalized \*/

putchar(toupper(\*s)); /\* Output in upper case \*/

else

putchar(tolower(\*s)); /\* ... force \*/

}

return(OK); /\* Did what is supposed to do \*/

}

### FCB Type

In MSX-DOS2, the FCBs are defined as a structure, which is necessary to use the file access function calls of MSX-DOS1, which are also accessible by file handles in the assembly language. It is provided for compatibility. Since the FCB type is defined for MSX-DOS1, it is not fully compatible with MSX-DOS2. For more information, see the MSX2 Technical Handbooks, the manual that comes with MSX-DOS TOOLS, and MSX-DOS2 Reference Manual 141.

The type FCB is defined in the header file bdosfunc.h.

### FD Type

This type represents a file handle that is used for low-level input/output functions (such as open() and read()). The standard C uses a non-negative integer for a file handle, so the file handle is declared as an int. FD type is provided to emphasize that it is a variable for a file handle.

The FD type is defined in the header file type.h.

FD fd;

char buf[BUFSIZ];

if ((fd = open("file", O\_RDONLY)) == ERROR) {

puts("File not found\n");

exit(l);

}

if(read(fd, buf, BUFSIZ) == 0) {

puts("File has no data\n");

exit(l);

}

### FIB Type

The FIB type is specific to MSX-DOS2, and is used to search for files, etc.

(FIB) is a structure. For details, see page 136 of the MSX-DOS2 Reference Manual. The FIB type is defined in the bdosfunc.h header file.

### FILE Type

The type is a structure used by high-level input and output (function such as fopen() and getc()) to control the buffer. Most of the time what the user declares using this type is the definition of a pointer (to the FILE type). Even if you define a variable in the FILE type, it cannot be used to input or output a file.

The type of FILE is defined in the stdio.h header file.

#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) {

...

}

### LONG Type

This type is defined to allocate a 4-byte area within the FCB type. It cannot be used in practice.

The LONG type is defined in bdosfunc.h header file.

### size\_t Type

The type is used to describe the size of a region or string. In the standard library, for example, it is used in the return value of strlen().

The type size\_t is defined in the type.h header file.

### STATUS Type

The STATUS type is similar to the BOOL type in some respects, but it returns the status of the function's execution result.

The type of STATUS is defined in the type.h header file.

#include <stdlo.h>

#Include <io.h>

if (unlink("editor.bak") == ERROR)

puts("Back up file not exist\n");

### TINY Type

It is an 8-bit numeric type with decimal values from 0 to 255. The advantage of this type is that it takes up less space than int and is faster. One drawback is that the range of values that can be expressed is narrow. However, it is sufficient for small array indices and a few loop variables for example.

The TINY type is defined in the header file type.h.

#include <stdio.h>

TINY count[1O];

TINY i;

int c;

for (i = 0; i < 10; i++)

count[i] = 0;

while ((c = getchar()) != EOF)

if (lsdigit((char)c))

count[c -'0']++;

for (i = 0; i < 10; i++)

printf("%d: %d\n", (int)i, (int)count(i]);

### VOID Type

This type is not a data type, but a declaration to the programmer in the case of MSX-C. It is used in defining and interpreting the function, and means that there is no return value of the function, and it cannot receive or return a value.

The VOID type is defined in the type.h header file.

#include <stdio.h>

VOID noret()

{

puts("This function returns no value.\n");

}

### XREG Type

There is used for the function callxx() that sets and calls a register at an address. As a parameter, an area for specifying a register value is needed, which is of XREG type. You set the value to the structure and pass the pointer to callxx().

The XREG type is defined in bdosfunc.h header file.

#include <stdlo.h>

#Include <bdosfunc.h>

XREG reg;

reg.be= (unsigned)\_GETDTA;

callxx(BDOS, &reg);

prlntf("Current DTA is %04x\n", reg.de);

## Header File

In C standard library header files are usually used to create programs. To use the standard library functions, you have to declare their use in your program. However, if the user declares the use of each function, it would be too much work. A header file saves time and declares its use in a batch. The user can read the header #include directive to avoid declaring each function and defining constants.

### Splitting the Header File

In MSX-C Ver. 1.2, the header file is classified and divided according to the type of function. The header files up to Ver.1.1 consist of two header files, stdio.h and bdosfunc.h, the former containing most of the standard functions and the latter containing functions, defines and constants needed to use the MSX-DOS function calls.

In Ver. 1.2, the functions were split in order not to overwhelm the user's symbol tapering by declaring unnecessary functions due to the increase in the number of standard library functions, and to reduce the compile time. However, the coding style up to Ver.1.1, includes only stdio.h at the beginning of the program. In order not to interfere with the use of the system, if you include stdio.h, all the header files are included (stdio.h includes more). To include each segmented file individually, put the line "#define DIVHEADER" at the top of the C source file. This will prevent stdio.h from including unnecessary header files other than its own. The bdosfunc.h must be included separately.

1. Ver. 1.1. including

#include <stdio.h>

main()

{

...

}

1. Include only stdio.h in the Ver. 1.2

#define DIVHEADER

#include <stdio.h>

main()

{

...

}

1. Include stdio.h and stdlib.h in the way of Ver. 1.2

#define DIVHEADER

#include <stdio.h>

#include <stdlib.h>

main()

{

...

}

Ver.1.2 way to include direct.h and stdlib.h

/\*

#define DIVHEADER

Not necessary to not include the stdio.h

\*/

#include <direct.h>

#include <stdib.h>

main()

{

...

}

### How to Include the Header File

Header files are usually placed together in one directory, (e.g. a:\include). You can specify the default directory to include files in CF for Ver.1.2. using the environment variable include. This environment variable can be set to a single directory.

Set the default include directory to a:\include

A>set include=a:\include

If you put the header files in the "a:\include" directory, CF will compile properly even if there are no header files in other directories.

There are two ways to include. First and the most common one is to enclose the files to be included in "<" and ">". In this case, it searches for the default directory and includes the file if it exists. If we don't find it. A message is displayed and the compilation is aborted. The other way to use it is to enclose the file name in "". With this specification, it searches the current directory first, includes the current directory if it exists, and searches the default directory if it does not. If this doesn't work, the compilation is aborted. Note that the default directory will not be searched if the file name contains a drive or root directory. Also, only one delimiter is needed in #include (#include "\myhead\mylib.h").

A common use of #include is as follows. Surrounded by "<" and ">" for the module to be used as a standard library declaration. For the case of a user-specific header file, use "" to enclose it.

1. #include <stdio.h>

Searches the default directory and includes it if it exists. If it's not there, an error message is displayed and compilation is aborted.

1. #include "myhead.h"

It searches the current directory, then the default directory, and includes the file when it is found. If there is none, an error message is displayed and the compilation is aborted.

### Contents of the Header Files

Here we will look at the declarations in each of the 13 header files. For more information about the function, see Chapter 6 "Standard Library Function Reference" and "4.1 MSX-C Standard Types" for a description of the types. In the header where the constants are defined, the explanation is given. The constants are freely available in the program, but you cannot change them.

#### bdosfunc.h

In bdosfunc.h, a function call name that directly executes a function call of MSX-DOS2 is defined. In other words, what is missing in the standard library functions is compensated for by direct MSX-DOS function calls.

|  |  |
| --- | --- |
| Types | Struct FCB, Struct FIB, Struct XREG |
| Functions | bdos(), bdosh(), bios(), biosh(), call(), call (), callxx() |
|  | They are the names of function calls in MSX-DOS, starting with the constant "\_", corresponding to their numbers.  Anything beginning with ''BIOS'' This is a function that can be used with bios()  BDOS This sets the address that calls a function call of MSX-DOS. |

The names of function call in bdosfunc.h have been significantly changed in Ver.1.2. The purpose of this is to match the function call names in the MSX-DOS2 reference manual. However, in order to make it compatible with the programs we have developed so far, we have taken care to make it possible to use the function call names from version 1.1. It defines a constant (macro) called "HEADbdosfuncver11" before including bdosfunc.h. Now you can compile it as is.

#define HEADbdosfuncver11

#include <bdosfunc.h>

#### conio.h

conio.h contains the declarations of the functions related to the direct input functions from the keyboard and the I/0 port.

|  |  |
| --- | --- |
| Functions | getch(), getche(), kbhit(), sensebrk() inp(), outp() |

#### ctype.h

ctype.h contains macros to find out what kind of character type it is. It includes the declaration of functions that convert characters to upper and lower case. isalpha(), isupper (), islower (), isdigit(), isspace(), iscntl(), iskanji(), iskanji2() are realized as parameterized macros, not as functions. They may evaluate the parameter more than once, so passing an expression with side effects as a parameter will not work correctly. (Side effects expressions are expressions that contain ++ and -- operators, assignment operators, and function calls).

|  |  |
| --- | --- |
| Macros | isalnum(), isalpha(), iscntr(), isdigit(), islower() iskanji(), iskanji2(), isspace(), isupper(), isxdigit() |
| Functions | tolower(), toupper() |

#### direct.h

direct.h contains a description of the directory-related functions.

|  |  |
| --- | --- |
| Functions | chdir(), expargs(), getcwd(), mkdir(), rmdi() |

#### io.h

io.h contains declarations of functions that are accessed using a file handle, such as low-level input/output functions.

|  |  |
| --- | --- |
| Functions | close(), creat(), eof(), isatty(), open(), read(), rename(), unlink(), write() |
| Constants | O\_RDONLY read mode for open()  O\_WRONLY write mode for open()  O\_RDWR read-write mode for open()  STDIN Indicates the constant STDIN file handle 0.  STDOUT " 1.  STDERR " 2.  STDAUX " 3.  STDPRN " 4.  STDLST " 4. (same as STDPRN) |

#### malloc.h

malloc.h contains declarations of memory management functions.

|  |  |
| --- | --- |
| Functions | alloc(), free(), sbrk(), rsvstk() |

#### memory.h

memory.h contains declarations of functions that operate the memory directly.

|  |  |
| --- | --- |
| Functions | memcpy(), memset(), movmem(), setmem() |

#### process.h

process.h contains declarations related to the commands themselves, such as those for exiting and starting the program.

|  |  |
| --- | --- |
| Functions | execl(), execv(), \_exit(), exit() |

#### setJmp.h

setjmp.h contains the function declarations and type definitions for branching across functions.

|  |  |
| --- | --- |
| Types | jmp\_buf |
| Functions | longjmp(), setjmp() |

#### stdio.h

stdio.h includes the declaration of high-level I/O functions, the definition of FILE structures, macros, and constants.

It is a good idea to have a good idea of what to do. It also includes the inclusion of other header files for compatibility with stdio.h up to MSX-C Ver.1. The inclusion of other header files for compatibility with stdio.h up to version 1.

|  |  |
| --- | --- |
| Types | FILE Type |
| Macros | fileno(), feof(), ferror() |
| Functions | 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() |
| Constants | BUFSIZ Size of the buffered high-level input/output function. The buffers provided by setbuf() must be of this size.  EOF The value indicates that the input from the file is taken to the end option file.  \_NFILES The maximum number of files that can be opened simultaneously by a high-level I/O function is defined. This includes stdin, stdout, stderr, stdaux and stdprn.  IONBF The value for setvbuf() with no buffering.  IOLBF The value for setvbuf() with line buffering.  IOFBF The value for setvbuf() with full buffering.  stdin Pointer to the file FILE type that represents the standard input.  stdout " standard output.  stderr " standard error output.  stdaux " standard auxiliary output.  stdprn " standard printer output. |

#### stdlib.h

stdlib.h contains a collection of commonly used functions that cannot be classified into other headers.

|  |  |
| --- | --- |
| Functions | abs(), atoi(), getenv(), max(), min(), putenv(), qsort() |

#### string.h

string.h contains the declarations of functions that operate on characters and strings.

|  |  |
| --- | --- |
| Functions | sprintf(), sscanf(), strcat(), strchr() strcmp(), strcpy(), strlen(), strlwr() strncat(), strncmp(), strncpy(), strupr() |

#### type.h

type.h defines the standard types used in MSX-C. It also defines the value constants that are used for the type.

|  |  |
| --- | --- |
| Type | BOOL, FD, size\_t, STATUS, TINY, and VOID types |
| Constants | NULL used as a pointer to a constant NULL. (e.g. "just try it"). It is used as an error if the function returns pointer.  TRUE For BOOL type, it is the counterpart of FALSE  FALSE For BOOL type, it is a counterpart to TRUE  YES It is used for BOOL type and is paired with NO  NO It is used for BOOL type and is paired with YES  OK This is for the STATUS type and indicates a normal completion of processing  ERROR This is for the STATUS type and indicates an abnormal end of processing |

## Summary of Standard Library Function

The standard library, which comes with MSX-C Ver. 1.2, works only on MSX-DOS2. A command created by linking the standard library is checked for the version of MSX-DOS before it is executed, and if it is not version 2 or higher, "This program needs MSX-DOS2" will be displayed and it will not be executed.

### File Input/Output Functions

In C, the input and output of a file is divided into three stages (open, input/output, and close).

1. Open the file first. This operation allows the user to open a file. Functions fopen() or open() gives the name of the file and the mode of read or write. Then the function returns the information needed for subsequent file access. The user stores the value in the variable (in the case of fopen() it is a pointer to the FILE type, in the case of open() it is an FD type value).
2. The next step is the input and output of data. Here, you can actually read or write data from the file. However, this must be the same as the mode used to open the file. When retrieving data, the user passes to the function a value specifying the file, the position of the data to be read, and the amount to be read (fgets() and read()). Writing passes the value of the file specification, the position of the data to be written, and the amount to be written (fputs() and write()). Some functions return the data to be read (getc()), others pass the data to be written (putc()), so it doesn't always pass three pieces of information. But it is only hidden (implicitly specified).
3. The last step is to close the file. You only need to pass the file-specified value to the close functions (fclose() and close()). When a file is opened for reading, you simply discard the file management information you have been using, but there is one more thing you have to do in write mode. which is to write unwritten data to disk. After writing out the data, the process is the same as the read mode.

That's the end of the file I/O. Multiple files can be input and output simultaneously, if necessary. When the program is started, the most frequently used files are already open, so you can input and output data immediately.

File I/O functions can be roughly classified into two groups. One is a high-level input/output function, the others are the low-level I/O functions (open(), read(), etc.).

MSX-C Ver. 1.2 standard library supports the same redirection and pipeline as the UNIX and MS-DOS versions. The standard input and output can be changed to a file from the command line, but the standard error output is fixed to the console and cannot be changed.

The I/O functions can handle not only disk files but also device files in MSX-DOS. Each device is allowed to do the following operations:

input-output direction of the device file

|  |  |  |
| --- | --- | --- |
|  | input | output |
| CON | Ok | Ok |
| PRN | Inadvisable | Ok |
| NUL | Ok | Ok |

NUL is a dummy input/output, where the input is always returned EOF and the output is discarded without actually doing anything.

#### High-Level Input/Output

The high-level input/output functions are based on character-by-character input/output, which allows easy processing of text files.

There is five high-level input/output file pointers, stdin (standard input), stdout (standard output), stderr (standard error output), stdaux (standard auxiliary output), and stdprn (standard printer output) that are declared in the header file stdio.h. When the program is invoked, stdprn can be used. stderr and stdaux are automatically opened by the kernel. stderr and stdaux can input and output in the standard C, but the MSX-C library does not allow it, so they are unidirectional (output only in this case).

A file is usually processed in "text" mode for high-level input/output functions. In text mode the input CR-LF combination is replaced by a single newline character (\n) and the content of the file after the EOF character is ignored. On the other hand, in the case of the output, the newline characters are reverted to CR-LF and the EOF character is added to the end of the file. However, if such a conversion is not desirable, it is possible to process the file in "binary" mode.

Sometimes, while using a high-level I/O function, you need to know which file handle the file uses. For example, when you want to wait for a keystroke if the output is to a device. In that case, you can use fileno() (macro) to find the corresponding file handle. This provides a bridge between high-level and low-level input/output functions. For example, to check if a file opened with fopen() is a device, you can use

if (isatty(fileno(fp)))

device processing

else

file processing

It will be better if you are You can get a file handle and you can also read() and write(), but you should be careful not to drop the data, because high-level I/O functions are buffered.

#### Buffering of High-Level Input/Output Functions

The high-level I/O functions buffer the data to be input and output in the main RAM (MSX-DOS TPA).

This section briefly describes the operation of buffering. In the read operation, first of all, it checks whether there is data in the buffers and if so, it passes it. If not, the input data is read from the disk until the end of the buffer or full capacity is reached. It returns the requested portion from the top of the list. In the write operation, the output data is stored in the buffers and is written to disk when the buffers are full. This is called buffer flush.

There are three types of buffering: full buffering, row buffering, and unbuffering. The full buffering does the same thing as above. This is the method that is employed when you open a file with fopen(). The line buffering is a single line template feature. This method flushes the buffer when the input is at the position or when the "\" is printed. Line buffering is not very useful unless the input or output target is a device file (e.g., the console). Without buffering, the function responds to input and output requests in a character-by-character fashion. In the case of device files, it is convenient to see the current status.

Because of the use of read() and write(), the specifications of MSX-DOS2 are directly exposed. That is, if you input from a device file, you will always end up with line buffering.

Let's talk about flushing the buffer a bit. Flushing the buffers is to flush the data remaining in the buffers to the disk. The points at which the buffers are flushed are (1) when the buffers are full, (2) when "\" is output during line buffering, and (3) when input from the console via a high-level I/O function. (4) When the user explicitly flushes it. There are four options. The standard life cycle performs (1) to (3) automatically. (4) is necessary when you want to display a message in the middle of the message on the console. If you want to start typing from the continuation of a line with a prompt, (3) will do it automatically.

In the case of using a high-level input/output function, a pointer to a FILE structure is used, which contains overhead for buffering control. The user can change the buffer size, buffer position and buffering method with setvbuf(). However, it can only be used when data input and output are not yet done after the file is opened.

In the MSX-C Ver. 1.2 standard library, five streams are automatically opened when control is passed to main(). Pointers to those file structures are stdin, stdout, stderr, stdaux and stdprn Among them stderr and stdaux are unbuffered, the rest are opened with row buffering.

#### Low Level Input/Output Functions

The low-level I/O functions read and write directly to disk without buffering. Therefore, if the number of data is small and the number of read/write operations is large, it becomes slower. If the data is bundled, low-level I/O functions are fast enough to read and write. There are only read() and write() data input/output functions, so we can't consider them for character-by-character or string. Therefore, it is suitable for binary data processing. Also, you can no longer access the device files, as was the case in Ver. 1.1.

### Strings and Character Processing Functions

There are two types of character-related functions. It is used to detect character types and to manipulate characters and strings.

A character type test function is a function that determines whether a character is case sensitive or not, for example, for a given character, functions are used to check if the character is of a certain type (a number in case you are expecting a number). These functions start with "is".

The other set of functions mainly targets strings. There are various ways to compare strings and find a character in a string. These functions start with "str". There are also functions that convert characters to upper and lower case. Other functions can be used to limit the length of the string for comparison, and their names start with "strn".

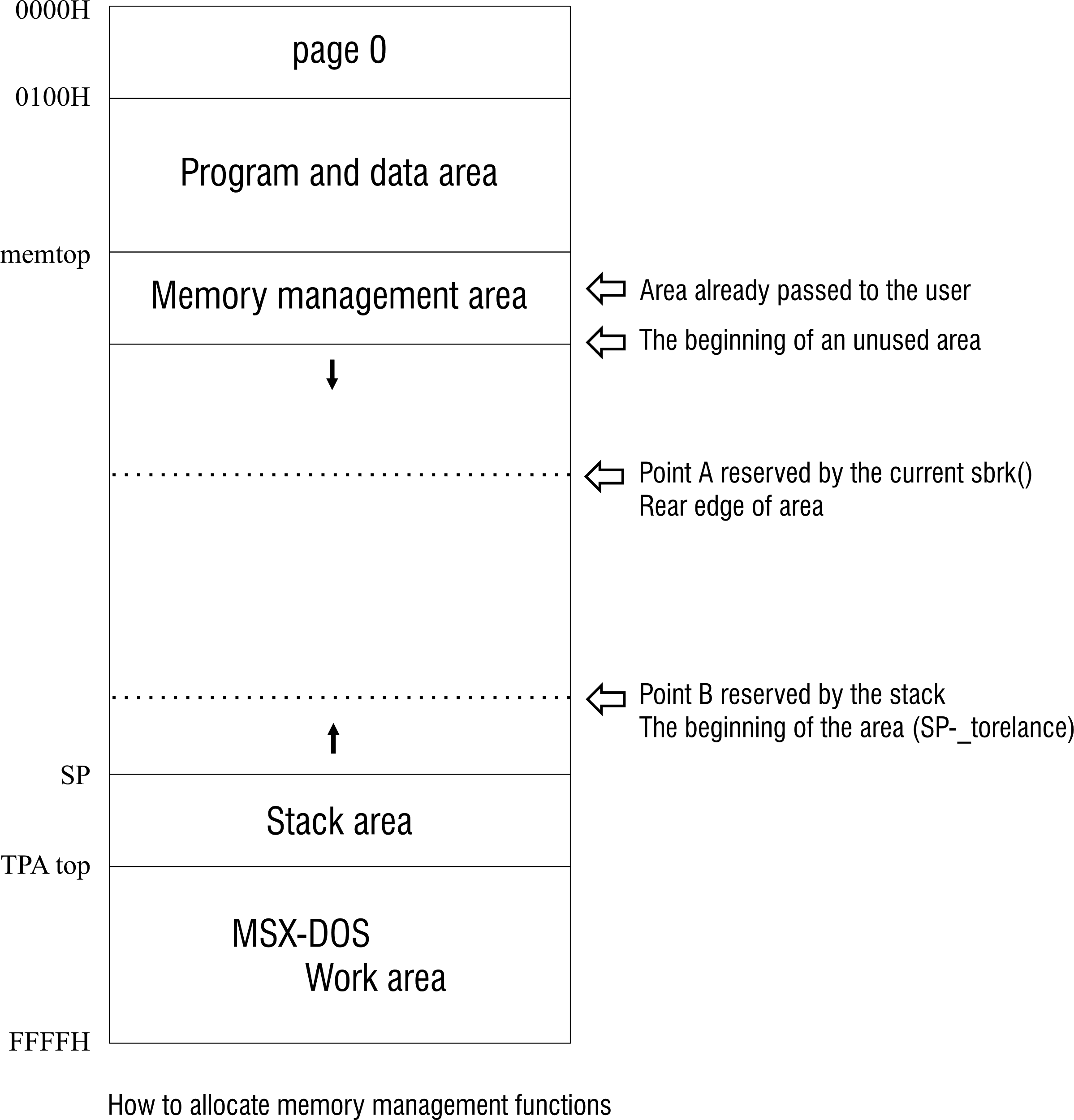
### Memory Management Functions

In MSX-C, there are two types of memory management functions. One is a low-level memory management functions and the other is a high-level memory management function.

#### Low-Level Memory Management Functions

Low-level memory management functions are a simple memory allocation function (sbrk()) and a stack allocation function (rsvstk()). sbrk() allocates a specified amount of program and data space to be used in MSX-C. rsvstk() stores the size of the stack space in the global variable \_torelance. The area allocated by sbrk() extends in the direction of the stack, so the stack specified by rsvstk() is used for allocation. The stack pointer SP of the CPU varies depending on the situation where sbrk() is called, but the position of the stack pointer at the time of the call minus \_torelance (point B in the figure below) is the back end of the region.

If it is not violating, it returns the top of the region (memtop). After the return, the memtop is set at point A to prepare it for the next sbrk(). (The figure below shows the success in securing the area.



Allocation of memory management functions

#### High-Level Memory Management Functions

The other type of memory management is a combination of alloc(), where the size and other management information is prepared and then passed to the user, and free(), where the space passed by alloc() is released and can be used again by alloc(). If a normal program needs memory at runtime, it uses these high-level memory management functions. High-level memory management functions are also used in the buffer area of file input/output and the getenv() function. You can mix the low-level and the high-level of memory management functions, but by using only the high-level, you can reduce the waste of memory using free() function.

If there are no more than n bytes in the area released with free(), the user tries to allocate it with sbrk(). In other words, if you try to free() an n-byte alloc() and then try to free() an (n+m) byte alloc() immediately afterwards, you will need (n + sizeof(HEADER)) + (n + m + sizeof(HEADER)) or more memory.

### Directory Functions

MSX-DOS2 now supports hierarchical directories, and the directory function is a function to support this. This includes moving the current directory and creating and deleting subdirectories. You can also use wildcards (\* and ?) that support hierarchical directories to the actual file name. If you use them, you should be able to use the store directory from within the MSX-DOS commands.

### Program Operation Functions

The program must be terminated while it is running if it cannot be continued due to error or other reasons. However, it can be tricky because it calls a number of functions (nested) or opens a file. In such a case, a function to terminate the program immediately is also provided.

There may be more to it than one program. Some of the functions can be used for this purpose to move the execution from one program to another.

### Keyboard I/O Functions

The input and output functions are useful for a variety of functions, but there is also a keyboard direct input function for those who need to respond to each character input. When you want to use the full functionality of the MSX, the I/O function that sends commands directly to the surrounding LSIs is provided.

### Machine Language, MSX-DOS Support Functions

The standard MSX-C library has a variety of machine language functions, but MSX-DOS2 supports more functions. We have added the function call to support using it with other machine language functions. The function can also be used in machine code programs written outside of MSX-C.

### Memory Operations Functions

You can use MSX-C's description for memory operations, but if the amount of memory content is large, it takes a long time to copy the memory contents. If you need high speed, you can use the memory manipulation functions.

### Generic Functions

There are many more useful functions besides the ones reviewed above. There are many commands that can be used to expand the range of C commands, such as the acquisition and setting of chateau variables and data sourcing.

## Differences Against the UNIX Standard C Libraries

The following functions correspond to the UNIX standard C functions. However, the functions marked with an asterisk (\*) have slightly different specifications. Note that the functions marked with "-" are supersets of standard C, so some caution is required when using them.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| abs | atoi | chdir | clearerr | close \* |
| creat | eof | execl \* | exexlp \* | execv \* |
| execvp \* | \_exit | exit | fclose | fcloseall |
| feof | ferror | fflush | fgets | fileno |
| fleshall | 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 |  |  |  |

The following functions exist only in MSX-C

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 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 |

## Creating and Maintaining Libraries

### Configuration of the MSX-C Library

The following four rel files are available in MSX-C.

ck.rel kernel calling routine

clib.rel Standard library (including the main body of the kernel)

crun.rel Execution-time routines

cend.rel Standard Libraries

ck.rel contains a routine contained in clib.rel, and ck.rel calls it. In MSX-C, a kernel is a program that provides an environment for running a fuser program. The kernel works in the following way, most of the processing is done by calling the program in CLIB, which is described below:

1. First, parse the command line and set argc and argv.
2. Standard input (stdin), standard output (stdout), standard error output (stderr) and standard helper output (stdaux) and standard printer output (stdprn).
3. Control is passed to the function "main" in the user program.
4. When execution of the user program ends as a result of a successful completion of main() or a call to exit(), control returns to the kernel and closes all open files at that time.

The source file of ck.rel is ck.mac and the source program of the main body of the kernel is stored in process.c with the function \_main()

clib.rel is described as the standard MSX-C library. It is a collection of standard library functions or internal functions, such as printf() and atoi(). At link time, only the functions used in the user program are extracted from the library.

The zos program is contained in the following 8 C file standalone assembly language file.

clibc.c Creating skeletons in clibmac.mac

direct.c File directory manipulation function source files

io.c Low-level input/output, keyboard function source file

malloc.c Memory management function source file

process.c Program management function source file

stdio.c High-level input/output function source file

stdlib.c General- purpose function source file

string.c String manipulation function source file

clibmac.mac Standard library function assembler description

By providing the source program, users can easily create their own kernels, replace standard library functions, or add their own functions based on it. For more information on creating a life cycle, please refer to "4.5.3 Creating a Dedicated Life Cycle" in the Standard Library.

See Section 4.5.4 "Recreating the Standard Library" for information on maintaining another file lib.tco for standard library functions, for use with FPC (Function Parameter Checking Utility). This file contains information about the types and arguments of all the standard library functions.

crun.rel is a collection of subroutines that are necessary for the execution of a C program. Operations such as multiplication and division (without instructions corresponding to the CPU) are converted to a call to a routine in CRUN by the compiler. CRUN is designed to compensate for the difference between the language specification and the hardware, and there are other subroutines such as shift operations and signed comparisons. The source file is crun.mac.

cend.rel is considered as part of the standard library. This is the last module to be linked and the rest of the heap area. The source file is cend.mac.

### MX Library Maintenance Support Tool

When you create and maintain your library, you can use the MX (Module Extraction Utility) to greatly increase the efficiency of what you need to do.

MX specifically performs the following operations.

1. Extract the modules (functions) from an intermediate language file (a .tco file) or an assembly language file. tco file) or an assembly language file to extract modules (functions).
2. The steps of code generation and assembly of the extracted modules are output to the standard output.

In other words, you can extract any module from a file containing multiple functions or create a separate file for each module, which makes it much easier to maintain and create the library. The procedure for this is output to the standard output, so you can use it as a batch file by redirecting it to the .bat file.

The MX format is shown below.

MX [Option] ... [Module name] ...

If the .mac file is not found, MX looks for the .mac file with the specified file name, and if it is found, MX considers it as a source file in assembler and starts working. It looks for the tco file, and if it is found, it assumes that it is a C intermediate language file and starts working on it.

The module to be extracted from the file is specified after the file name. One or more modules can be specified here. In addition, the module names include wildcard characters (\* and ?). If you do not specify any module at all, all the modules are extracted.

MX requires two batch files, arel.bat and crel.bat, in the current directory of the current drive. These two files make it easy to change the output of MX: arel.bat from the assembly language file and rel file, and crel.bat from the intermediate language (.tco) files. MX converts ''%1'' in the file to the name of the module and outputs it.

If arel.bat and crel.bat are not found in the current directory, it looks for the directory where mx.com is located. If this doesn't work, it exits with the message "Skeleton file <filename> not found" (<filename> is arel.bat or crel.bat). If you put the commonly used formats arel.bat and crel.bat in the same directory as mx.com, you will hardly notice them. If you want to use a different you place arel.bat and crel.bat in the current directory and they will be used.

##### Options

|  |  |
| --- | --- |
| -l | The output includes the order in which the library file is created by calling LIB80. If this option is not specified, the output is terminated when the .rel file is created. |
| -o [path\] | Extracts a divided module to the directory path. The directory must be terminated by "\''. If it doesn't end with "\", it is the name of the library file when the "-l" switch is used. However, the extracted files are in the same directory as the library files. If this option is not specified, the module is extracted to the current directory of the current drive. |

An error message on MX about the operation of the command. For example, a message saying that the .tco or .mac file cannot be found. When such a message is displayed, MX aborts the operation and returns to the command repel. See Section 8.4 "MX Error Messages" for actual error messages.

For a specific example of using MX, see "4.5.3 Creating a Dedicated Library".

### Creating a Dedicated Library

There are two types of "Libraries": (1) those that link all functions to the executable file at link time, and (2) those that link only necessary functions to the executable file at link time. The advantages and disadvantages of each are as follows.

1. Pros: It's easy to create.

Cons: It links all functions in the file to the executable file, including functions not used by the program. The executable file becomes larger.

1. Pros: Only functions used in the program are linked. It improves the efficiency of the file.

Cons: It takes a lot of time to create them.

When we talk about "library", we usually refer to (2), and the MSX-C clib.rel and crun.rel also have this kind of characteristics. It is recommended to make a life cycle such as (2) as much as possible, since there is no other way to make a life cycle such as (1) except that it can be made manually and silea also here.

First of all, we will briefly describe how to create a life cycle in (1).

#### Library Linking to All Functions

Consider the following two functions power(), log2() in a library

int power(x, e)

int x, e;

{

int y;

for (y = 1; e > 0; e --)

y \*= x;

return (y);

}

int log2(x)

lnt x;

{

int y;

for (y = 0; x > 1; x >>= 1)

y++;

return (y);

}

Assuming that these functions are in the source file xlib.c, the procedure for creating the library is as follows:

A>cf xlib ◄┘

...

A>cg xlib ◄┘

...

A>m80 =xlib ◄┘

...

A:\del xlib.mac ◄┘

The above procedure creates xlib.rel. This concludes the creation of the library XLIB. If you set your program to prog, the link to use this library is as follows

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

In this case, /s (switch to link the necessary functions) is not needed after xlib. This is because, even if you specify it, all the functions will be linked together in the end.

In the case of programs written in assembler, you can create or use the library in the same way.

Consider the following three functions hex(), ror() and rol() (named as alib.mac), you can do the following:

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

The procedure for linking these functions is as follows.

A>m80 =alib ◄┘

The procedure for linking is the same as in C, as follows

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

As mentioned above, once you give up the "link only what you need" feature, creating the library is very easy. However, this is still not a perfect library. In the following, we will discuss how to create a library that links only the required functions.

#### Library that Links Only to the Necessary Functions

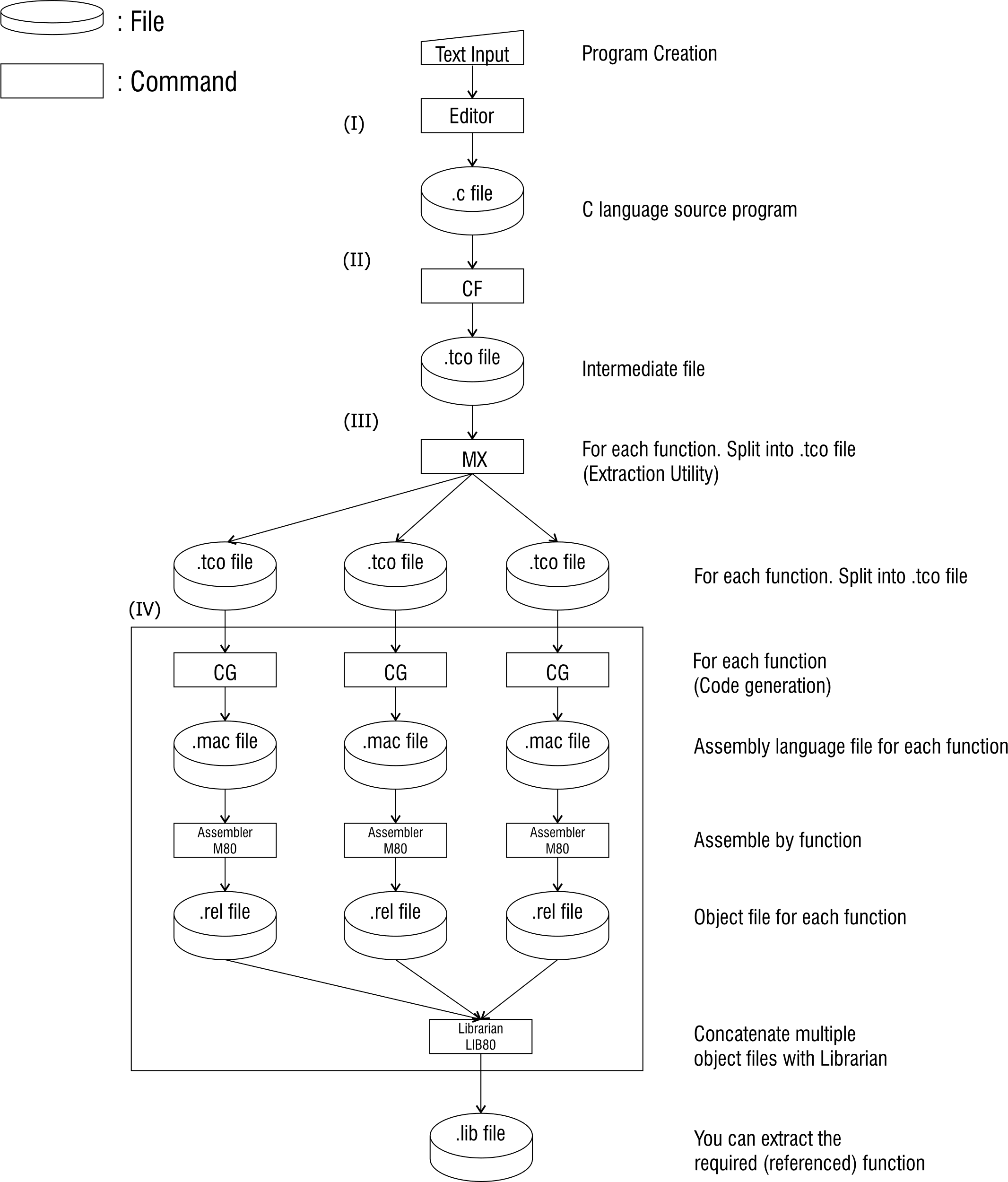


Fig. Procedures for creating a dedicated library

Before we talk about how to create a library, we need to define the environment in which we will work. (You can also use the forremk.bat file described in section "4.5.4 Rebuilding the standard library" to set up the environment.

It is assumed that there are two disks (A: and B:). Leave the B drive empty.

software

m80.com

l80.com

lib80.com

MSX-C Compiler Package

cf.com

cg.com

fpc.com

mx.com

arel.bat

crel.bat

It is useful to have a text editor in addition to this.

As before, let's create a library of xlib.c, which contains two functions, power() and log2().

First, create the xlib.c on drive A by your editor (see figure (1) on page 97) and then type the following command.

A>cf xlib ◄┘ Page 97 Figure (II)

...

A>mx -lob xlib > temp.bat ◄┘ Page 97 Figure (III)

...

A:>temp ◄┘ Page 97 Figure (IV)

CF creates xlib.tco from xlib.c. Next, the MX will extract from xlib.tco, power.tco and log2.tco.

Take the two modules and place them on the B drive. In addition, the MX can also be used for:

1. You can generate the code to make a mac for these files.
2. Do an assembly and create a rel.
3. Start LIB80 and link the two rels.

The procedure "temp.bat" is output to the standard output, so you can redirect it to the file temp.bat and immediately runs a temp.

After these processes, the B-drive has a file called xlib.lib. In addition, the there are two files, power.rel and log2.rel, which can be used for

A>del b:\*.rel ◄┘

to delete them after that.

A>ren b:xlib.lib xlib.rel ◄┘

then the desired library is completed. The usage of xlib.rel is as follows.

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

The /s after xlib is necessary. By adding it, the linker will only link the necessary functions.

Note that the order in which the functions are defined is a problem when using the L80 linker. For example.

(Correct example.)

char islower(c)

char c;

{

return ('a'<= c && c <='z');

}

char toupper(c)

char c;

{

return (islower(c)? c -'a'-'A': c);

}

Let's assume that the library contains a file x.c that consists of two functions, namely in this case, if islower() is placed before toupper() in x.rel, an undefined error will occur when linking with the /s option at L80.

Actually, since MX concatenates .rel in the opposite order to the one in the source program, writing islower() first creates the correct life cycle, but writing toupper() first, as shown below, creates the life cycle of modules are placed in the wrong order. Therefore, in the library source program, below the called function first and the called function last.

(Incorrect example.)

char toupper(c)

char c;

{

char islower();

return (islower(c) ? c -'a'-'A': c);

}

char lslower(c)

char c;

{

return ('a'<= c && c <='z');

}

Next, let try to write a program in assembler to the library. The aforementioned hex(), ror(), rol(), but the program should be changed as follows

Create this file on the A-drive under the name alib.mac.

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@

rol@:

inc e

rol1:

dec e

ret z

rlca

jr roll

ENDMODULE

To turn it into a library, simply type in the following command.

A>mx -lob alib > temp.bat ◄┘

A>temp ◄┘

Then MX extracts hex.mac, ror.mac, and rol.mac from alib.mac, and the created batch file temp assembles them and creates rel, then calls LIB80 and concatenates them.

When all is complete, the B drive will have alib.lib, and you can deleting the unnecessary .rel. After renaming the file to alib.rel, the library is ready to be used.

As in the case of C, the order of the modules is an issue when using the L80 linker. In the source program, please write the module of the caller first, and the module of the caller second.

You can create the library itself as described above, but there is one more thing left to do. It is to create a file to check the parameters to be fed to the FPC.

This is created by the following procedure.

1. Run CF he against the C source program for the library to create a tco file.
2. If you have source programs written in assembler, create a C representation of their function types and parameter types and convert it to a tco file using CF.
3. Compress and concatenate the tco files created in (1) and (2) using the "-c" switch of FPC.

As an example, let's create the xlib.c and a lib.mac check file as described above. First of all

A>cf xlib ◄┘

to create xlib.tco.

Next, represent the parameter of alib.mac in the program C. Create the following as a lib.

char hex(c)

char c;

{}

char ror(c, n)

char c;

int n;

{}

char rol{c, n)

char c;

int n;

{}

From this file, you can create alib.tco with the following command

A>cf alib ◄┘

Lastly,

A>fpc -c axlib xlib alib ◄┘

you will have a file axlib.tco for the check.

Now, the previous instructions have shown how to create a new life cycle, but the life cycle it is not efficient to rebuild the whole library when you want to modify one of the functions.

In this section, we describe how to replace a function in an existing library. Again, we will use xlib.c as an example we suppose to change only the function power in this file which consists of two functions, power() and log2(). To replace the power() in the library file xlib.rel with the new one, the procedure is as follows

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 ◄┘

In the previous examples, the MX was given one option. MX does not output the procedure for LIB80 to the standard output, unless you specify also xlib. Because power is specified after power, MX extracts only the function power().

The command to LIB80 (the last two lines) tells the user to extract the non-power() modules from the old xlib.rel and connect them to the new power.rel. Be careful not to change the order of the modules.

### Recreating Standard Libraries

The library created by the following method is included in the system files as [clib.rel, crun.re]. In normal use, there is no need to rebuild these libraries.

A single unused disk is required to recreate the standard library. Since it takes about two hours to create a life cycle, you should test the changes carefully before making any changes. (A single 1DD drive cannot be executed due to insufficient capacity. Use two drives or prepare a 2DD drive).

To create the recreating disk, use the batch file forremk.bat. It is located in {directory\batch}. Let us explain how to use FORREMK.

1. Format a new disk.
2. Copy the forremk.bat to the formatted disk. If you have the MSX-C master disk in your B-drive, you will use the following

A>copy b:\batch\forremk.bat ◄┘

1. Put the copied disk in the A-drive and enter the following commands.

A>forremk b ◄┘

1. Now you can put your MSX-DOS2 system disks and so on into drive B at the prompt.

If you are using a drive simulator, replace the disk after the message "Insert disk for drive <d:>" appears.

The disk for re-creation is ready, but since you will be compiling, you can run CENV if necessary, to prepare the environment (environment variables). (For more information, please refer to Section 1.3.3 MSX-C Building a Development Environment).

The resulting disk should contain the following files. (Suppose there is a disk in the A-drive).

a:\ (contents of the root directory)

forremk.bat The batch file to create this task

msxdos2.sys MSX-DOS2 stem file

command2.com "

clibc.c File directory manipulation function

direct.c source files for creating directories in clibmac.mac

io.c Low-level input/output, keyboard function source file

malloc.c Memory management function source file

process.c Program management function source file

stdio.c High-level input/output function source file

stdlib.c General- purpose function source file

string.c String manipulation function source file

ck.mac Kernel Calling Routine Source File

clibmac.mac Standard Library Function Assembler Description

crun.mac Run-time routine source files

cend.mac Standard library function assembler description

cenv.bat Batch file for preparing MSX-C environment

genlib.bat Batch file for starting the re-creation of the standard library

genliba.bat

genlibc.bat Recreate batch file for C

genrel.bat The batch file for re-creating the rel files

gentco.bat The batch file for re-creating the tco files

arel.bat The skeleton file for MX (for assembly language)

crel.bat The skeleton file for MX (for C language)

a:\bin (Contents of bin\ directory)

m80.com Assembler

lib80.com Librarian (Librarian Management Utility)

cf.com Compiler (Parser)

cg.com Compiler (Code Generator)

fpc.com Function Parameter Checker

mx.com Module Extraction Utility

a: \include (Contents of the header file directory)

bdosfunc.h Header for MSX-DOS function calls

conio.h keyboard and I/O function header

ctype.h Header for character type judgment functions

direct.h Header for directory manipulation functions

io.h header for low-level I/O functions

malloc.h Header for memory management functions

memory.h header for memory extraction functions

process.h Header for program operation functions

setjmp.h Set jmp header

stdio.h Header for high-level input/output functions

stdlib.h header for general-purpose functions

string.h Header for string manipulation functions

type.h Header for defining MSX-C standard types

Setting Environment Variables

INCLUDE a:\include

PATH a:\; a:\bin

Now that you have a disk for re-creating the library, let's see how to use the genlib.bat, a batch file for re-creation the disk is in drive A. The GENLIB argument only specifies the drive name (a single "b" in the case of the B drive) where the standard library will be re-created.

A>genlib a ◄┘

Here, it is created in the same A-drive as the current drive. Please enter the following.

A>genlib a ◄┘

After that, the batch file automatically creates the library. The whole thing consists of 11 parts, and at the beginning of each part, the name of the file you are creating is displayed.

Make process.Iib 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 Iinker L80

Make .tco file for parameter checker FPC

In addition to this, the display shows the current state of the device. The following message is displayed after normal operation.

:

:

MSX-C function parameter checker ver 1.20x

complete

A>

If you run out of disk space when the library is extended, use two drives: A with the above disk in the A drive and a blank disk in the B drive, then enter the following command

A>genlib b ◄┘

The .rel file and the final library is re-created in the B-Drive.

# Chapter 5: Applications of the MSX-C Compiler



## Creating Machine Language Routines in the Disk-BASIC Environment

In this section, we describe how to create a machine language program for use as a subroutine in Disk-BASIC.

### Disk- BASIC Environment and USR Functions

#### Disk- BASIC memory map

First, look at Figure 6-1. This shows the memory status at the start of Disk-BASIC. The user area of Disk-BASIC ranges from the address following the end address of the BASIC interpreter's area to the address before the start address of the disk work area by one address.

The starting address of the user area is 8000H in the case of MSX2, but the starting address of the disk work area depends on the number and type of disk drives implemented.

Currently, the largest disk work area is taken when there are two disk drives in 2DD (double-sided) format, or when even one of them has a functioning 2-drive simulator. In this case, the lower limit of the disk work area is about DE70H for Disk-BASIC version 1.00 and about E580H for Disk-BASIC version 2.00. However, the lower limit of the disk work area is set immediately after starting Disk-BASIC.

You can find out the exact address ibhcnyekgc HIMEM (FC4AH and FC4BH), so it's best to check it for safety.

#### Allocation of machine language regions and USR functions

The following is the procedure for loading a machine language file under Disk-BASIC environment to execute it using the USR function.

1. Allocates an area for loading the machine language program with a CLEAR statement.

The upper address of the user area is set to the address given by the second parameter of the CLEAR statement. For example

100 CLEAR ,&HCFFF

Then the maximum usable area for BASIC programs, including variables and stack space, is limited to CFFFH (the new HIMEM-1), and the area between D000H and the original user area is reserved for loading the machine language programs. The size of this area is set according to the size of the machine language file to be read.

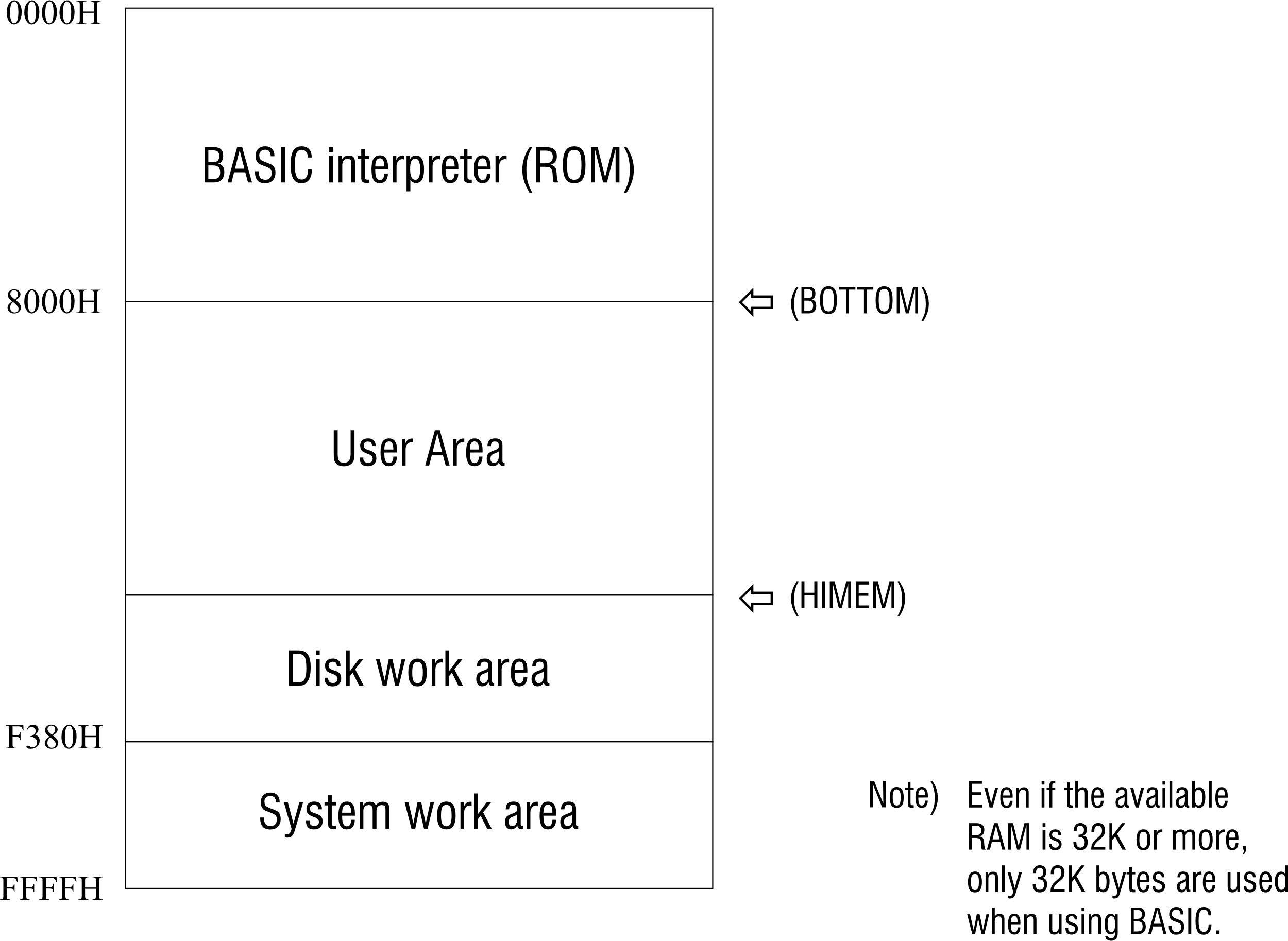


Figure 6 -1 the memory state in Disk-BASIC mode

1. The BLOAD statement loads manually a language file into the memory area allocated above.

A machine language file that can be read by the BLOAD instruction must be in a special format for this purpose.

1. The DEF USR statement specifies the starting address of the machine language program.

For example, if a machine word is read from the D000H address

100 CLEAR ,&HCFFF

200 BLOAD "MACHIN.BIN"

300 DEF USR=&HD000

This will allow you to use the USR function as shown in

500 A=USR("c") (pass the single letter "c" to the machine routine and assign the result to the variable A)

This is the most common way to use machine language in Disk-BASIC, but to achieve this a machine language file created that can be read by the BLOAD instruction, as described in (2) above. A header with information about whether to load or not needs to be included.

The following section, "5.2 Creating Source Programs in C" shows the actual process of creating programs for the Disk-BASIC environment in C, and "Section.3 From C Source Files to BLOAD Files" shows the process of generating BLOADable machine language files from the C source files. This section explains the procedures for compiling and linking.

## Creating Source Programs in C

### Accepting USR Function Arguments

The argument given in a USR function call is available in the A register on the machine side. The following table shows the values and argument types of the A-register.

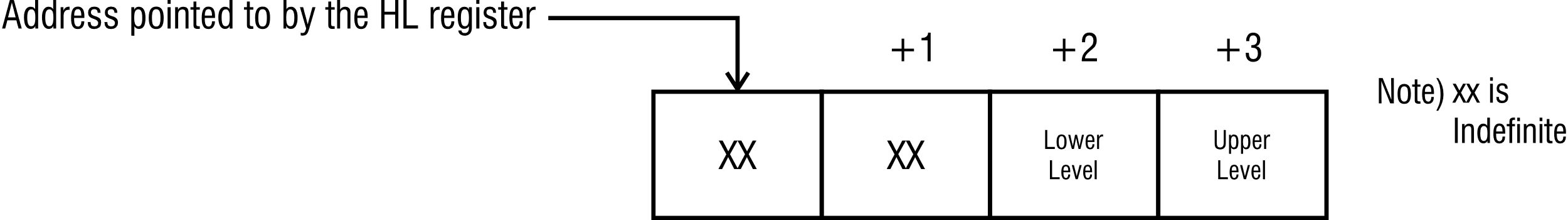
The actual value of the argument is passed as the address where the entity is stored. The register to be addressed depends on the argument type. In the case of 2-byte integer type, single-precision real number type, and double-precision real number type, the address is stored in the HL register, and in the case of string type, the address is stored in the DE register. (See table below)

|  |  |
| --- | --- |
| 2 | 2-byte integer type |
| 3 | Single Precision Real Number Type |
| 4 | Double Precision Real Number Type |
| 8 | character string type |

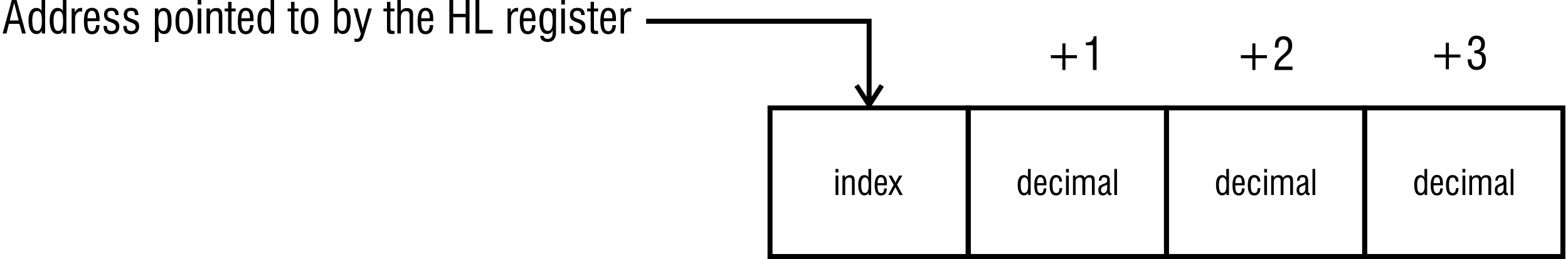
Table A: Types of arguments assigned to the registers Two-byte

##### How to Pass Values by Arguments

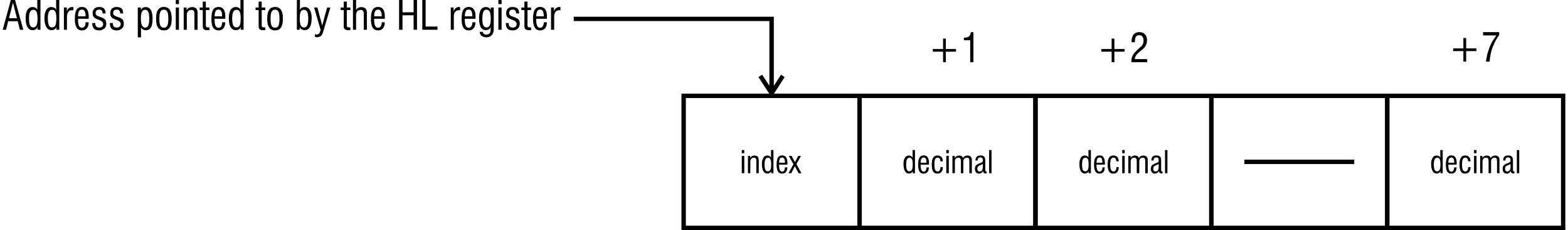
###### 2-Bytes Integer Type



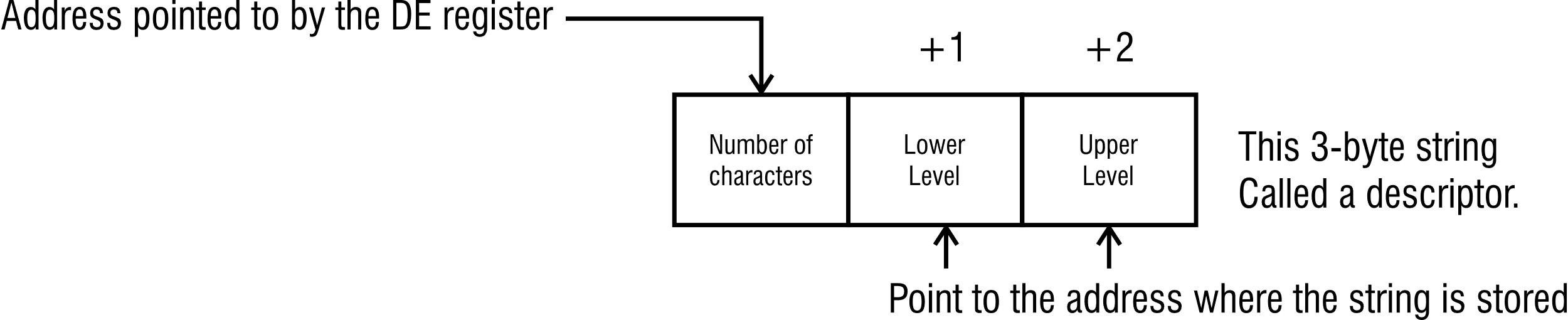
###### Single-Precision Real Number Type



###### Double-Precision Real Number Type



###### Character String Type



How can we see the value passed to the register in this way on the function side of C? See the table in "3.2.1 How to pass parameters". As seen here, the fixed parameter function can receive the value of the A register as the first parameter of char type and the value of the DE register as the second parameter of other types (2-byte value). However, then the type of the passed argument (in the A-register) cannot be known, so when writing a USR function in C alone, only string-type arguments can be handled by the DE register. If you want to deal with other types, write an interface routine that allows the assembler to get the HL register values.

The following is an example of the main routine of a USR function in C that takes a string argument.

typedef struct{

TINY length;

char \*strptr;

} DESCRIPT;

main(type, desc)

TINY type; /\* type is the value of A register \*/

DESCRIPT \*desc; /\* desc is the value of the DE register \*/

{

if(type != 3){ /\* If the argument type is not string type, \*/

outstr(ermes); /\* it must not be used, so \*/

return; /\* return with type other than 8; \*/

}

:

:

}

### Using System Calls

If you want to use function calls in a C program under MSX-DOS, you can use the library functions bdos(), bdosh(), etc.

Under Disk-BASIC, function calls (BDOS function) are possible, but the address of the function call destination is different from the one under MSX-DOS, so the above two functions are not available. Therefore, you must write a program that sets the necessary registers for the BDOS call and calls the destination address of the BDOS call in Disk-BASIC directly.

In the following, we will explain what kind of assembler source is generated by compiling C sources.

To execute a function call, put the function number of the function you want to use in the C register of the CPU, put the necessary arguments in the specified register, and then call addresses F37DH (in DOS, this was the 0005H address).

For example, if you want to display a character on the screen by using the screen 1-character output with function number 02H, you can write it in the assembler

ld e, 'a'

ld c, 02h

call 0f370h

:

:

That is, you should write the program so that the compilation results in the assembler source is as shown above. In other words, the pointer to the function is set at address F37DH, and the function call is made with the value of the register as the argument to the function.

Here is an actual example.

#include <stdio.h>

#pragma nonrec

puta() /\* Function output a single letter 'a' on the screen\*/

{

(\*(VOID (\*)())(0xf37d))((TINY)0,'a',(TINY)0x02);

}

The first argument is passed as 0. In Function 02H, there is no need to set a value in the first argument, i.e., A-register (or HL-register if dummy is not a char), but the following arguments are set in the E-register and C-register, respectively by the compiler. It is a pseudo-argument. The result of compiling the above source file is as follows

; 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

pubIic puta@

end

We were able to generate the assembler source for the purpose, except for the extra instruction to set the A register to 0 (xor a). In an actual program, it is better to define a macro with the #define statement instead of writing the above every time you call a function call.

For details on function calls, please refer to the manual provided with the "MSX-DOS TOOLS" and MSX2 "Technical Handbook" and "MSX-DOS2 Reference Manual".

### Returning Values

You can pass a USR function value to BASIC by modifying a string passed as a USR function argument. However, you cannot change the number of characters. Here is an example of how to return a string of values from the example in 6.2.1

retbas(desc,newstr)

DESCRIP \*desc;

char \*newstr;

{

if (strlen(str) != (int)desc->length){

outstr(eraes4);

return;

}

desc->strptr = newstr; /\* Set up a new string. \*/

return;

}

As shown in the above example, you can change the pointer to the passed string to a pointer to the string you want to return.

The following is an example of BASIC description for a receiver.

1000 B$=USR(A$)

## 5.3 From the C Source File to BLOAD File

### Creating BLOAD Files with L80 Only

Once the C source files are written, compile and link them to create BLOAD-format option files.

Creating a relocatable file (.rel) using CF, FPC, CG, and MSX-M-80 is similar to creating a DOS command file (.rel) is the same as DOS command file creation, but the linkage is different.

Here, recall that the MSX-DOS command file (.com) is loaded from 100H (for details, see the MSX-DOS manual, etc.).

In the case of creating a command file on DOS, you didn't have to pay attention to the start address of the load. However, for Disk-BASIC routines, you have to decide where to place the machine language in the fuser area (5.1 Creating Disk BASIC environment A and B). To create an object to be loaded from the start address of the area, use the /p option of L80 at link time, as shown below for the D000H address.

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

However, the resulting file is still a DOS command format file (.com). In other words, the area from 103H to CFFFH is unnecessary data (garbage?). It is buried in the (100-102H contains a jump instruction to "start:". Therefore, the BLOAD instruction cannot used to load this file from the D000H address in Disk-BASIC. You have to write a program that reads the garbage data part one byte at a time and stores only the necessary part in the area from the D00H. In this way, it is stored in the disk as a BLOAD file, which read C BLOAD instruction next time if it is saved from the D000H address using the BSAVE instruction of Disk-BASIC.

### MSX-DOS2 TOOLS BSAVE COMMANDS

Using the BSAVE command of MSX-DOS2 TOOLS, you can easily create a BLOAD file from a rel file without the work described in Section 5.3.1.

The procedure is described below.

First, create a hex file with the/x option when linking.

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

This creates a file called usrfunc.hex that can be executed from the D000H address. This file can be easily converted to a BLOADABLE file on Disk-BASIC by inputting the BSAVE command of MSX-DOS2 TOOLS (not the BSAVE command of Disk-BASIC).

A>bsave usrfunc.hex>usrfunc.bin

usrfunc.bin is a file in a format that can be read by the BLOAD instruction of Disk-BASIC.

## About the Sample Program

On the disk of this package, there is a sample machine language routine under Disk-BASIC, There are four files: search.c, search.bas. search.bat and bk.mac.

search.c is a C language routine and a subroutine of search.bas that searches for files on the disk using a system call.

After compiling and assembling search.c, search.bat links bk.mac with assembled bk.rel and generates search.hex.

Next, run the BSAVE command of MSX-DOS2 TOOLS to convert the file into BLOAD format. In addition, start Disk-Basic and load and execute search.bas. You can follow the process of creating a machine language subroutine for Disk-BASIC simply by starting this batch process.

Once you run search.bat to the end, the BSAVE command for TOOLS is needed in the current directory of the current drive.

Now we have roughly described the process of creating a machine language routine in Disk-BASIC. We have omitted some explanations in handling arguments and the creation of BLOAD format files, but if you have any questions, please consult the MSX2 Technical Handbook, MSX-DOS2 TOOLS manual, etc., and other literature on MSX-BASIC.

## Creating a program to convert to ROM

Here are the steps and methods for writing a program to ROM.

### General rules for creating a ROM-based program

#### About the Library

The first problem in creating a ROM conversion program is the standard kernel (CK) and the standard library (CLIB). Some of these programs use MSX-DOS features, so they cannot be used at the edge of a ROM-enabled program. However, CLIB includes functions that do not depend on MSX-DOS, such as strcpy() and toupper(). These functions can be used in ROMed programs as well, but if you link CLIB easily, you may accidentally call functions that use BDOS features.

Therefore, if you want to use these functions (strcpy() or toupper()), you can use

* + - 1. Import the functions in the source program as they are in the program you are creating, or
      2. Extracts only ROMable functions from clib.rel using LIB80 (library manager) and creates a separate library

Please take one of the following methods

The header file cannot be used as it is. Extract only the necessary parts and create a separate header file.

#### The Initialization Part

Next, the initialization process must be done by a ROMing program. For programs running under MSX-DOS, the kernel itself in clib.rel does the initial setup, but as mentioned above, the kernel cannot be used in a ROM-based program, so you need are placement for it.

Typically, the initial configuration is done with an assembler program, such as

extrn main()

ld sp, the address of the stack pointer

:

:

(Do hardware initialization if necessary)

:

:

jp main@

end

That is, you set the stack pointer, do the necessary hardware initialization, and then jump to the C function main() (there is no restriction that the name of the C entry must be main, e.g., it can be foo or zot). In this case, there are no parameters to pass to main()unlike the usual MSX-main() program in DOS.

VOID main()

{

:

:

}

Now, to use the above initialization program, first create this program, assemble it, and then you will get the .rel file that you need to link to it . 5.5.5.2 shows a practical example.

#### IN/OUT

IN/OUT is the most annoying thing when converting to ROM. In the assembler, you can use IN, OUT instructions, but such instructions are not available in C, so you have to call them as functions.

The MSX-C standard library includes the functions inp() and outp(), which are the easiest to use.

The second approach is to have a function for IN/OUT from one port for each port. It's a good thing. For example

public indat@, inctl@, outdat@, outctl@

indat@:

in a,(00h)

ret

inctl@:

in a,(01h)

ret

outdat@:

out (00h), a

ret

outctl@:

out (0lh), a

ret

and the four functions in the assembler, and in a C program

char indat(), inctl();

VOID outdat(), outctl();

VOID int()

{

outctl((char)0x0l);

outctI((char)0x0l);

outctl((char)0x40);

outctl((char)0xce);

outctl((char)0x37);

}

char getch()

{

while ((inctl() & 0x02) == 0)

;

return (indat());

}

VOID putch(c)

char c;

{

while ((inctl() & 0x0l) == 0)

;

outdat(c);

}

It is used by calling it as follows, the 8251's program is the initialization of the 8251 (init()), the input (getch()) is not as slow as inp() or outp(), because the call/ret is only executed an extra time compared to the case of IN/OUT in the assembler.

#### Absolute Address

In some cases, the input and output ports are allocated in memory space, not in I/O space. In this case, the input and output operations can be described much more elegantly than with I/O.

For example, if the previous program uses 0FF00H instead of port 00H and 0FF01H instead of 01H to perform the operation, we can write the following.

#define dataport (\*(char \*)0xff00)

#define ctrlport (\*(char \*)0xff0l)

Char getch()

{

while((ctrlport & 0x02) == 0)

;

return (dataport);

}

VOID putch(c)

char c;

{

while ((ctrlport & 0x0l) == 0)

dataport = c;

}

VOID inlt()

{

ctrlport = 0x0l;

ctrlport = 0x0l;

ctrlport = 0x40;

ctrlport = 0xce;

ctrlport = Ox37;

}

#### Interrupt Processing

MSX-C can also perform interrupt processing. However, there is no "interrupt procedural declaration", as some language procedures have introduced. Therefore, the compiler does not generate the sequence of saving all registers at the entrance of the interrupt, recovering registers at the exit, and unmasking interrupts, so these processes have to be written in the assembler.

As an example, consider a program that updates the time data with a timer interrupt every second. The time is expressed in hours, minutes, and seconds of TINY type, and the interrupt is assumed to be at 0038H.

Assembler:

extrn crst70

aseg

org 0038h

jp rst7

rst7:

push af

push bc

push de

push hi

call crst70

pop hl

pop de

pop bc

pop af

el

ret

C:

TINY hour, minute, second;

VOID crst7()

{

if (++second >= 60) {

second= 0;

if (++minute >= 60) {

minute= o;

if(++hour >= 24)

hour= 0;

}

}

}

In the C program. If you need to prohibit or cancel an interruption, you can use the following methods In assembly

public di@, ei@

di@:

di ; Prohibition of interruptions

ei@:

ei ; Allow interrupts

ret

Create and call the function

There is a point to note about interrupts. That is, if you have a function that is called by both interrupt processing and normal processing, they must be recursive. In such a case, use recursive even if you don't actually make the call.

#### Handling of Constant Tapering

Among the initialized variables, those with storage classes of static and extern distributed in the code segment (CSEG), and when running a program in RAM, such as under MSX-DOS, the values of these variables can be changed, but the program is not run as a When written to ROM, these variables cannot be rewritten because they are resent by the code segment and behave as if they were constants. Therefore, be careful about the actual use of variables in the storage classes and programs.

The role of the initialization program in a ROM program is to "realize constant tapering", not to initialize variables. When you initialize a variable, you must do so with an assignment statement. The string is an "array of initialized characters of the memory class static and is placed in code segments and cannot be changed when written to ROM.

### Examples of Converting a Program to ROM

The MSX-C compiler package contains the files rom0.mac and rom1.c. This is a sample program for ROM been burned into ROM with MSX and run.

First, assemble the rom0.mac as follows to create the .rel file.

A>m80 =rom0 ◄┘

Next, compile rom1.c and create the rel file. You can do this with the following sequence.

cf rom1

fpc-u rom1

cg rom1

m80 =rom1 (rom1.mac assembly)

del rom1.mac

Link at the end. This can be done with the following command.

l80 / p:0,/d:RAM the top address of RAM,rom0,rom1,crun/s, rom/n/e

Origin below loader memory, move anyway (YorN)?N

After the above links, you will get rom.com. This Write to ROM. When you run the program, you will be prompted ">". Where

>d0,FF

will dump the contents from 0 to 00FFH.

>sxxxx

Also, to display the contents of the memory and write a new value.

The program is a debugger with only D and S commands. Use it as an exercise in programming and extend the lever program to create a more sophisticated debugger.

# Chapter 6 Standard Library Function Reference

In this chapter, the standard library functions of MSX-C Ver. 1.2 are described in alphabetical order. Please use it as a function reference manual.

A function consists of the following seven items These items may be omitted if they are not needed.

1. Function Name

Only names of the described functions are shown. The “getchar" indicates function getchar(). When more than one function is described on the same page, they are written in succession.

1. Format (function header)

The function header is shown as the format for using the function. The header files required for the functions are also shown. This header file does not need to be included if you simply include stdio.h. See "4.2 Header Files" for more information about this).

1. Description

Explains how the sorting number works. Also, if the function has parameters, its description is included.

1. Return Value

Describes what values are returned as a result of the function's operation.

1. See Also

Here are other functions to contrast with the described functions.

1. Notes

These are the things to be aware of when using the function.

1. Example

Here is a simple example using the function. This example is written in such a way that it becomes a command when compiled.

That's all, but if you don't know how it works, we recommend you look at the Libraries resource file.

abs

Format

#include <stdlib.h>

int abs(n)

int n;

Description

Gives the absolute value of an integer n. The absolute value is the number without the negative sign. the absolute value of 0 is 0, the absolute value of 5 is 5, and the absolute value of -3 is 3.

Return Value

Returns the absolute value of an integer n. That is, it returns n if n is 0 or positive and -n if n is negative.

Note

abs() does not allow converting the maximum negative integer (-32768). (it returns the same value).

Example

#include <stdio.h>

/\* Displays the absolute value of -3, 0, 5 \*/

main()

{

printf("%d %d %d\n", abs(-3), abs(0), abs(5)};

}

alloc

Format

#include <malloc.h>

char \*alloc(n)

size\_t n;

Description

free() allocates n bytes of freeable memory space. alloc() and free() are high-level memory management functions that are used in combination. alloc() is used when the space is needed at runtime, depending on the situation, and free() is used when the space is no longer needed. See "4.3.3 Memory Management Functions" for details.

Return Value

The pointer to the top of the allocated area is returned. If there is not enough memory, the function returns NULL.

See Also

free(), rsvstk(), sbrk()

Notes

Since the return value is the first address of the available area, if it is set to p, the user can use it freely from p up to (p+n-1).

If there are no more than n bytes in the area freed with free (), try to allocate the area with sbrk()

I will. That is, if you try to free() an n-byte alloc() and then try to free() and then do a (n

+ m) byte alloc(), it will require ( ( ((n + sizeof( HEADER)) + (n + m+ sizeof (HEADER))) or more memory.

Example

#include <stdio.h>

/\*

Function to make a file copy

dst from the file handle src with the buffer size specified

\*/

STATUS cp(dst, src, bufsiz) /\* copy src to dst with buffer \*/

FD dst, src; /\* file handles \*/

size\_n bufsiz;

{

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)

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");

exlt(1);

}

close(s);

close(d);

}

atoi

Format

# include <stdlib.h>

int atoi(s)

char \*s;

Description

Converts a string to a signed decimal integer. Leading space and tap are ignored, then the sign is converted up to non-decimal characters as numbers. Overflow is ignored when the integer range (from -32768 to 32767) is exceeded.

Return Value

Returns the converted integer value of a string.

See Also

Conversion fscanf(), scanf(), sscanf() with "%d"

Notes

When the input string is out of the range of integers, no error is generated in case of overflow.

Example

#include <stdio.h>

/\*

Convert string "-232" and string "732" to a number then display and

Assign the input from the keyboard to a variable as a number and display it

\*/

static char intstr[] = "732";

char buf[6];

main()

{

int i;

printf("%d %d\n", atoi("-232"), atol(intstr));

gets(buf, 6);

i = atoi(buf);

printf("%d\n", i);

}

bdos, bdosh

Format

#include <bdosfunc.h>

char bdos(c[, de, hl])

char c;

int bdosh(c[, de, hl])

char c;

Description

Calls a function call in MSX-DOS. Specifies the name of the function call in the first parameter. The function call name is #defined in the header file bdosfunc.h. The second and third parameters can be left unset if they are not needed. Each when specified de, hl passes to the registers and calls a function call.

Return Value

In the case of bdos(), it returns the value of the a-register at the end of the function call as a char. bdosh() returns the value of the hl register at the end of the function call as int.

See Also

call(), calla(), callxx()

Notes

The first parameter specifies the name (or number) of the function call, which must be of char or TINY type. If you don't specify it, the function call name is not correctly passed to MSX-DOS.

The function call names are different from bdosfunc.h in MSX-C Ver.1. If you are using it, you need to change it. The function call name is the same as the library name in the MSX-DOS2 function specification.

Example

#include <stdio.h>

#Include <bdosfuno.h>

/\* Show the connected drapes \*/

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 >>= l;

}

}

bios

Format

#include < bdosfunc.h>

char bios(code)

char code;

Description

Make an MSX-DOS BIOS call.

Return Value

Returns the value of the a-register after the BIOS call as a char.

Example

#include <stdio.h>

#include <bdosfunc.h>

main()

{

char c;

c = getch();

bdos{\_CONOUT, c);

bios(BIOS\_CONOUT, c);

}

call, calla

Format

int call(addr, a, hl, be, de)

int \*addr, a, hl, be, de ;

char calla(addr, a, hl, be, de)

int \*addr, a, hl, be, de ;

Description

Calls a specific address by specifying a register. It is used when using a subroutine written in the assembler.

Return Value

call() returns the value of the hl register at the end of the routine, indicated by addr, as an int.

calla() returns the value of the a register at the end of the routine indicated by addr.

Notes

Use callxx() to specify ix , iy.

Example

#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);

}

/\* Execute a function call (file handle renaming) \*/

if (calla(BDOS, dummy, argv[2], fd \* 256 + \_HRENANE, dummy)) {

puts("Handle rename falled\n");

exlt(1);

}

}

callxx

Format

#include <bdosfunc.h>

VOID callxx(adsrs, reg)

unsigned adrs;

XREG \*reg;

Description

Set the registers and call the routine indicated by adrs. The configurable registers are af, ix, iy, bc, de, hl. The registers are assigned to a structure of XREG type. After executing, the XREG type structure contains the value of each register returned by the called routine. The function can be used to pass a necessary value or to receive a value from a function call of MSX-DOS. The type XREG is defined in bdosfunc.h.

Return Value

No return value

See Also

bdos(), bdosh(), call(), calla()

Notes

This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

#include <bdosfunc.h>

main()

{

XREG reg;

/\* Display the DOS version and date \*/

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%266);

}

chdir

Format

#include <direct.h>

STATUS chdir(path)

char \*path;

Description

Changes the current directory by the specified path name. You can specify path from either the root directory or the current directory. Specifying drive path changes the current directory of the drive. ([d: ][\ ] path.)

Return Value

OK if the current directory has been changed, otherwise returns ERROR.

See Also

getcwd(), mkdir(), rmdir()

Notes

chdir() does not allow you to change the current drive.

This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

main()

{

/\* Check for the presence of the directory bin in the root directory \*/

if (chdir("\\bin") == ERROR)

puts("There is no'\\bin' Subdirectory.\n");

chdir("\\");

}

clearerr

Format

#include <stdio.h>

VOID clearerr(fp)

FILE \*fp;

Description

clearerr() is often used to clear the flag of an end-of file (EOF) and make it ready to be read. you enter it from standard input, once it becomes an end-of file (EOF), and then enter it again with clearerr(). function.

Return Value

There is no return value.

Notes

The function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

int c;

/\* After the standard input becomes an end of file, enter again \*/

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

Format

#include <io.h>

STATUS close(fd)

FD fd;

Description

Closes the file indicated by the file handle fd.

Return Value

If the file handle is closed successfully, the function returns OK. Otherwise, it returns ERROR.

Note

close() is a low-level input/output function.

Example

#include <stdio.h>

static char msg[] = "write open and close";

main()

{

FD fd;

/\* Write the data in the file, close and exit \*/

if ({fd = open("data", O\_WRONLY)) == ERROR) {

puts("File cannot make\n");

exit(1);

}

write(fd, msg, strlen(msg));

close(fd);

}

creat

Format

#include <io.h>

FD creat(filename)

char \*filename;

Description

Creates the file indicated by filename. File creation means that if a file already exists, its contents are deleted before opening it.

Return Value

if the file cannot be opened, for example because of a disk error, it returns an ERROR. Otherwise, the file handle is returned.

See Also

close(), open(), read(), write()

Notes

creat() is a low-level input/output function.

Example

#include <stdio.h>

static char str[] = "test data";

main()

{

FD fd;

/\* Create a new file, write the data, close and finish \*/

if ((fd = creat("creat.dat")) == ERROR) {

puts("File cannot creat\n");

exlt(1);

}

write(fd, str, strlen(str));

close(fd);

}

eof

Formatting

#include <io.h>

BOOL eof(fd)

FD fd;

Description

Checks whether the file handle fd has read to the end of the file (end-of-file, EOF). If the standard input is directed and you want to grab a character with getch(), the function checks for EOF beforehand to avoid terminating the command with an EOF error in the standard input.

Return Value

Returns TRUE if the file handle is EOF or if the file is not opened. Otherwise, it returns FALSE.

See Also

feof()

Notes

The function checks the EOF corresponding to a low-level I/O function. It does not return the correct value for a file handle accessed by a high-level I/O function. Use feof() for high-level I/O functions.

This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

char buf[100];

main()

{

FD fd;

/\* Read the file 100 bytes at a time to the end of the file \*/

fd = open("file", O\_RDONLY); /\* specify a file that exists \*/

while (read(fd, buf, 100)) {

puts(eof(fd) ? "TURE\n": "FALSE\n");

}

puts(eof(fd)? "TURE\n": "FALSE\n");

close(fd);

}

execl, execlp

Format

# include <process.h>

VOID execl(progname, arg1, arg2,...)

char\* progname, \*arg1, \*arg2, ... ;

VOID execlp(progname, arg1, arg2,...)

char \* progname, \*arg1, \*arg2, ... ;

Description

Both exeel() and execlp() load another program file and execute it. The character string progname is the name of the file to be executed. If the file name does not have an extension, .com is assumed. If the file specified by progname does not exist, execl() outputs "cannot exec: <progname>" to the standard error output and returns from the function. However,

execlp() searches in the directory indicated by the environment variable PATH. if still not found execlp() outputs 'cannot exec: <progname> " to the standard error output and returned from the function.

The character string specified by arg1, arg2,… is passed to the program to be chained as a command argument.

The specified command is executed while the currently open low-level entry/exit file is left open.

execl("time", "is", "money.");

From the command line

A>time is money.

It has the same effect as typing.

Return Value

No, there is not.

See Also

execv(), execvp()

Notes

Because execl() and execlp() are variable parameter functions. The following declaration is required in advance:

VOID execl(.), execlp(.);

it is included in the header file process.h.

The behavior of execl() has changed when the program cannot be found.

execlp() has been added to the standard library from Ver 1.2.

execlp() cannot find the file in the directory indicated by the environment variable PATH if progname is specified by the drive name or root directory.

Example

#include <stdio.h>

/\*

When there are one to three parameters in a command, reverse the parameters and activate the last parameter as a command.

\*/

main(argc, argv)

int argc;

char \*argv[];

{

switch (argc > 1) {

case 1:

execlp(argv[1]);

case 2:

execlp(argv[2], argv[1]);

case 3:

execlp(argv[3], argv[2], argv[1]);

}

}

execv, execvp

Format

#include <process.h>

VOID execv(progname, argv)

char \*progname, \*argv[];

VOID execvp(progname, argv)

char \*progname, \*argv[];

Description

Both execl() and execlp() read another program file and execute it. The string progname is given as the name of the file to be executed. If the file name does not have an extension, then .com is assumed. If the file specified by progname does not exist, execl() will "cannot exec: <progname>” is printed to standard error and returned from the function. In case of execvp(), it searched from the directory indicated by the environment variable PATH. If still not found, execvp() "cannot exec: <progname>" is output to the standard error output and returned from the function.

For the command parameters to be executed, create an array of the addresses of the command arguments and pass it as the parameter argv. Also, put the value NULL in the last element of the array argv. In other words, put the address of the first argument in argv[0], the address of the second argument in argv[1], and so on, and put NULL at the end.

The specified command is executed while the currently open low-level entry/exit file is left open. The five standard input/output files will remain the same even if the command is started, as long as they are redirected

static char \*param[]={

"all work", "and", "no play", "makes", "Jack", "a dull boy", NULL

};

execv("foo", param);

From the command line

A>foo all work and no play makes Jack a dull boy.

It has the same effect as typing.

Return Value

No, there is not.

See Also

execl(), execlp()

Notes

execv() has been modified to return when the program is not found.

execlp() has been added to the standard library since version 1.2.

execlp() cannot find the file in the directory indicated by the environment variable PATH d if there is a drive name or a root directory in progname.

Example

#include <stdio.h>

/\*

When there are one to three parameters in a command, reverse the parameters and activate the last parameter as a command.

\*/

main(argc, argv)

int argc;

char \*argv[];

{

if (argc > 1) {

printf("Start program %s\n", argv[1]);

execvp(argv[1], &argv[2]);

/\* Command not found \*/

puts("Sorry...\n");

}

}

exit, \_exit

Format

#include < process.h >

VOID exit(code)

int code;

VOID \_exit(code)

int code;

Explantation

Aborts the current command and returns to the command level. You can return by specifying a return (code). exit() flushes and closes all high-level input and output buffers. In contrast, \_exit() does not flush or close. Low-level input/output file handle is closed in both exit() , \_exit().

Notes

\_exit() has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

FILE \*fp;

/\* file.dat show "File not found" and exit the program if there is no data \*/

if ((fp = fopen("file.dat", "r"))) == NULL) {

puts("File not found\n");

exit(1);

}

}

expargs

Format

#include <direct.h>

int expargs(argc, argv, maxargc, xargv)

int argc, maxargc;

char \*argv[], \*xargv[];

Description

Function that expands the command arguments, including the wildcards, into their respective filenames by actually performing a directory search. Sorts the extracted file name if it contains a wildcard. You can also specify a composite file specification. However, only '+' can be used to delimit the file specification. Using this function, you can write a program that has the ability to expand wildcard characters into real filenames. The parameters argc and argv are an array of number of arguments to be expanded and pointers to the string xargv is an array of pointers to the same string as argv, where the address of each expanded command argument is set. It gives the size of xargv as maxargc.

Return Value

The number of expanded command arguments is returned as a function value. However, if there is not enough memory to decompress the arguments, or if the number of arguments after decompression exceeds maxargc. The value ERROR is returned.

Notes

If an ERROR is returned, the contents of the array xargv[] are invalid.

Example

#include <stdio.h>

#define MAXARG 100

char \*xargv[NAXARG];

/\* Expand and display the wildcards of the parameters \*/

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

Format

#include <stdio.h>

STATUS fclose(fp)

FILE \*fp;

TINY fcloseall()

Description

fclose() closes the file indicated by fp. fcloseall() closes all open files except stdin, stdout, stderr, stdaux and stdprn. In the text mode and in the light mode, it outputs ^Z (EOF character) and then closes it.

If the system automatically assigns file buffer, it will be released. fclose() can only close high-level entry/exit files. Use close() for low-level entry/exit files. When exit() exits or the program terminates normally, all files are closed with fclose() and then returned to the command level.

Return Value

fclose() can be used even if fp is NULL or if the close operation fails. in that case, ERROR is returned. Otherwise, OK is returned.

fcloseall() returns an ERROR if one of the file closures fails. Returns the number of closed files if all the closures are successful.

Note

The fcloseall() has been added to the standard library since version 1.2.

Example

#include <stdio.h>

main()

{

FILE \*fp;

/\* Read the file to the end-of file, exit and close it \*/

if ((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

Format

#include <stdio.h>

BOOL feof(fp)

FILE \*fp;

Description

Checks whether the file has been loaded to the end. When the file is read to the end (end-of-file, EOF), a flag is set. EOF varies in position depending on whether it is in text or binary mode. In text mode EOF is indicated by the character (^z), in binary mode the physical end of the file is the end-of-file.

Return Value

TRUE is returned if it has read up to EOF. Otherwise, it returns FALSE.

See Also

clearerr(), eof()

Note

The function is a macro.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

int i = 10;

char c;

/\*

Outputs 10 characters to the standard input. Displays a space when it becomes the end of file.

\*/

while (i && (c = getchar()) i= EOF) {

putchar(c);

i--;

}

if (feof(stdin))

while (i--)

putchar(' ');

ferror

Format

#include <stdio.h>

BOOL ferror(fp)

FILE \*fp;

Description

Checks whether an error occurred while the file was being written. This is mainly caused by a full disk. Use clearerr() to clear the error condition.

Return Value

If an error occurred, it returns TRUE. Otherwise, the function returns FALSE.

Notes

The function is a macro.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

FILE \*fp;

/\* Write data until the disk is full \*/

if ((fp = fopen("nomore", "w")) == NULL)

exit(1); /\* file cannot 11ake \*/

while (!ferror(fp))

fputs("Write until disk full\n", fp);

fputs("Disk full !!!\n", stderr);

}

fflush

Format

#include <stdio.h>

STATUS fflush(fp)

FILE \*fp;

Description

Writes file buffers to disk. In the high-level input/output function, the output data is stored once in the buffer. When the buffer is full, it is automatically written to disk (called flushing the buffer), and then the buffer is forced to be flushed. fflush() writes the contents of the buffer to disk before the buffer is full. You need to use fflush() if want to print the string without the newline (\n) in the standard output. After flushing, the file is still open.

Return Value

If the buffer flush was successful, the function returns OK. If the disk is full, an ERROR is returned.

See Also

4.3.1 (B) Buffering of high-level input/output functions

flushall()

Notes

When the file is closed or when the program exits normally, it automatically flushes the buffer and empties it.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

int i, sum;

/\* Calculates the summation up to l00 while displaying the progress \*/

sum = 0;

for (i = 0; i <= 100; i++) {

sum += i;

printf("%4d %5d³\n", i, sum);

fflush(stdout);

}

printf(" sum = %d\n", sum);

}

fgets

Format

#include <stdio.h>

char \*fgets(s, n, fp)

char \*s;

int n;

FILE \*fp;

Description

Reads a string from the file indicated by fp. The "string" ends with "\n" and is loaded into the area indicated by s. The size of the area is specified by n. The string ends with a "\n", followed by a "\0" as the end character of the string. If the file reaches the end option file while reading, the "\n" will not be entered but will be followed by a "\0". fgets() reads up to n-1 characters and returns "\0" at the end.

Return Value

NULL is returned when the file reaches the end-of-file and when nothing is read. Otherwise, s is returned.

See Also

gets(), fputs(), puts()

Example

#include

<stdio.h>

char buf[256];

main()

{

FILE \*fp;

/\* Display the contents of the file "memo" \*/

if ((fp = fopen("memo", "r")) == NULL)

puts("File not found\n");

exit(1);

}

while (fgets(buf, 256, fp))

puts(buf);

fclose(fp);

}

fileno

Format

#include <stdio.h>

FD fileno(fp)

FILE \*fp;

Description

Returns the file handle used for input and output of the file indicated by fp. A high-level file input/output always has a single low-level input/output file handle. Use fileno() to get the value of the file handle from the file pointer.

Return Value

The function returns the corresponding file handle fp. If the file is not opened, the value is not guaranteed.

Note

The function is a macro.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

FILE \*fp;

/\* From stdin

display the corresponding file handle up to stdprn \*/

for (fp = stdin; fp <= stdprn; fp++)

printf("file handle %d\n", fileno(fp));

}

flushall

Formatting

#include <stdio.h>

TINY flushall()

Description

Writes the buffers of all open files in write mode to disk. The file is still open after the flush.

Return Value

Returns the number of open files. There is no distinction between read and write modes.

See Also

fflush()

Notes

The buffer is automatically flushed and closed when the file is closed or when the program is successfully completed.

This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

main()

{

TINY file;

/\* Show how many files you have open \*/

file = flushall();

printf("%d files are open\n", (int)file);

}

fopen

Format

#include <stdio.h>

FILE \*fopen(filename, mode[, bufsize])

char \*filename, \*mode;

size\_t bufsize;

Description

Open the file indicated by filename for high-level input and output. The string mode must be one of the following:

"r" Read mode, text mode

"rb" Read mode, binary mode

"w" Write mode, text mode

"wb" Write mode, binary mode

"a" Append mode, text mode

"ab" Append mode, binary mode

The read mode is for reading and the file must be present. Write mode is for writing; if a file already exists, it is deleted first, and then opened. That is, the previous content will disappear. Append mode is used to add data to an existing file. When there is no file, the behavior is the same as in write mode.

In text mode "\r" + "\n" is defined as a single character "\\" on input, or "\n" as a single character in "\r" + "\n" on output. The ^z character make a end-of-file.

If you did not specify "b" as the second character in the mode, it is set to text mode. This is used for input and output, such as source files.

Unlike text mode, binary mode does not perform any conversions such as "\r" + "\n" on input and "\" on output. Also, do not make ^Z an end-of-file.

bufsize can be specified as the size of the buffer used for input and output. When omitted BUFSIZ is (1024).

Return Value

If the number of file openings is too large or if no file is found, NULL is returned. On successful opening, the pointer to the FILE structure is returned.

See Also

fsetbin(), fsettext(), setbuf(), setvbuf()

Notes

Since fopen() is a variable parameter function, the following declaration is required beforehand

FILE \*fopen(.) ;

This declaration is included in the stdio.h header file.

Example

#include <stdio.h>

char buf[256];

main()

{

FILE \*fp;

/\*

Displays the contents of the file newgame.dat

If there are no files, "File not found" will be displayed.

\*/

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

Format

#include <stdio.h>

STATUS fprintf(fp, format[, arg1, arg2,...])

FILE \*fp;

char \*format;

Explantation

fprintf() produces formatted output for the flags indicated by fp. For the control string, it is the same as printf(), see the section below.

Return Value

If there are no errors during the output, OK is returned. Otherwise, an ERROR is returned.

Referances

printf(), sprintf()

Notes

Since printf() is a variable function, the following declaration is required beforehand

STATUS fprintf(.) ;

This statement contained in the header stdio.h

fprintf() is a standard UNIX C superset.

Example

#include <stdio.h>

main()

{

FILE \*fp;

int array[10];

int count;

/\* Save an array of integers to a file indat.dat to save it. \*/

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

Format

#include <stdio.h>

STATUS fputs(s, fp)

char \*s;

FILE \*fp;

Description

Outputs a string s to the file indicated by the file pointer fp. It does not append "\n" to the end of the string for the output. The string s must be terminated by a null character.

Do not end the string with "\n" in the output. The string s must end with a null character. No characters are output, if there is "\n" in the string, "\r" and "\n" in text mode and output.

Return Value

If an error occurs during output, an ERROR is returned. Otherwise, OK is returned.

See Also

fgets(), gets(), puts()

Example

#include <stdio.h>

char buf[ 256];

main()

{

FILE \*fp;

/\* Save the input from the keyboard to file \*/

if ((fp = fopen("memo", "w")) == NULL) {

puts("File cannot make\n");

exlt(1);

}

while (gets(buf, 256))

fputs(buf, fp);

fclose(fp);

}

fread

Format

#include <stdio.h>

int fread(buf, size, count, fp)

char \*buf;

int size, count;

FILE \*fp;

Description

Count size bytes from the file indicated by file pointer fp to the area specified by buf. You can think of fread() as a high-level input/output function version of the low-level read() function. The maximum number of data to be read is not a simple number of bytes, but is specified by how much (count) of what type (size) to read.

Return Value

Returns the number of items that could be loaded (number of items). If set to 0.

See Also

fsetbin(), fwrite(), read()

Notes

If the data to be read is binary data, use fsetbin() or fopen() in binary mode. You must do this!

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

FILE \*fp;

int array[10];

int count;

/\* Read data stored in binary form in an array of integers \*/

/\* Open in binary mode \*/

if ((fp = fopen("worddat.dat", "rb")) == NULL) {

puts("File not found\n");

exit(1);

}

count = fread(array, sizeof(int), 10, fp);

}

free

Format

#include <malloc.h>

VOID free(ap)

char \*ap;

Description

alloc() and free() are high-level memory management functions that are used in combination, where alloc() is used when the space is needed at runtime depending on the situation, free() is used when the space is no longer needed. free() allows better use of the limited memory. See "4.3.3 Memory Management Functions" for details.

Return Value

No, there is not.

See Also

alloc(), rsvstk(), sbrk()

Notes

If there are no more than n bytes in the area released with free(), the function tries to allocate it with sbrk(). That is, if you try to free() an n-byte alloc() and then free() it and then do (n+m) byte alloc(), you will need more than ((( n + sizeof (HEADER)) + (n+ m + sizeof HEADER))).

Example

#include <stdio.h>

char buf[100];

main()

{

char \*p, \*\*ary;

int n = 0;

/\* Show the human power from the keyboard in reverse order \*/

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) {

/\* Take an area for the input \*/

fputs("No more memory\n", stderr);

break;

}

strcpy(p, buf); /\* Save in the reserved area \*/

ary[n++] = p; /\* Save the pointer to the area \*/

}

while (n) {

puts(ary[n]); /\* Show 1 line \*/

free(ary[n--]); /\*Free the area used for strings \*/

}

free(ary); /\* Freeing the area used for the array \*/

}

fscanf

Format

#include <stdio.h>

int fscanf(fp, format[, arg1 , arg2 , ...])

FILE \*fp;

char \*format;

Description

fscanf() takes the input with conversion from the file indicated by fp. format and later parameters are the same as in scanf(), see the section below.

Return Value

fscanf() returns the actual number of items assigned. The return value is In 1, if you specify 3 pointers to the variables for 3 parameters, no values are assigned after 2 Fl. When the end of the file is reached, the EOF is returned.

See Also

scanf(), sscanf()

Notes

Since fscanf() is a variable parameter function, the following declarations may be used to substitute the line feed character "\n", so be careful in specifying the conversion character. It is necessary to declare the following in advance.

intfscanf(.);

This declaration is contained in the header file stdio.h. fscanf() is a standard UNIX C superset.

Example

#include <stdio.h>

char buf[100];

main()

{

int val, n;

char c;

FILE \*fp;

/\* Receive 0 1 from the file as decimal, character, or string \*/

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

Format

#include <stdio.h>

STATUS fsetbin(fp)

FILE \*fp;

Description

Accesses the file indicated by fp as binary mode. Unlike the text mode, the binary mode does not perform any conversions such as "\r " + " \n" on input or ''\n" on output. Also, does not make at ^Z end of file. End of file is reached when the end of the physical file is reached (as determined by the length of the file). You can also enter binary mode when you open a file with open(). Add a "b" to the second character of the mode.

If you want to add data to the created file by means of a redirection mark (>>) at the command level, you can do so by executing fsetbin() before executing fclose(). This is because in binary mode, the ^Z is not output, so the file is not separated by it.

Return Value

It always returns OK.

See Also

fopen(), fsettext()

Example

#include <stdio.h>

char buf[256];

main()

{

int i, c;

FILE \*fp;

/\* Open in binary mode at first \*/

if ((fp = fopen("str.dat", "rb"))) == NULL) {

puts("File not found\n");

exit(1);

}

while((i = getc(fp)) != EOF) {/\* Receive the number of strings \*/ fsettext(fp); /\* Reads in text mode from now on \* /

while(i--) { /\* Display strings for integers only\*/

fgets(buf, 256);

puts(buf);

}

fsetbin(fp); /\* Switch to binary mode to take an integer. \*/

}

fclose(fp);

}

fsettext

Format

#include <stdio.h>

STATUS fsettext(fp)

FILE \*fp;

Description

Access the file indicated by "fp" as text mode. In text mode, "\r" + "\n" is converted to a single character in "\n" at the time of input, and "\n" is converted to two characters in "\n" at the time of output, and "\n" is converted to two characters in "\n" and ''^Z'' is used as an end option file, which is the input and output mode of the file. The first state is in text mode.

Return Value

Inevitably returns OK.

See Also

fopen(), fsetbin()

Example

#include <stdio.h>

char buf[ 256];

main()

{

int o, c;

FILE \*fp;

/\* Open in binary mode first \*/

if ((fp = fopen("str.dat", "rb")) == NULL) {

puts("Flle not found³");

exit(1);

}

c = getc(fp); /\* The first part of the file 2 Aligning the bytes number \*/

i =c + getc(fp) \* 256;

fsettext(fp); /\* from now on continue on in text mode \*/

while (i--){ /\* Display strings for integers only \*/

fgets(buf, 256);

puts(buf);

}

fclose(fp);

}

fwrite

Format

#include <stdio.h>

int fwrite(buf, size, count, fp)

char \*buf;

int size, count;

FILE \*fp;

Description

You can think of fp as a high-level input/output function version of the write() function of the low-level input/output function. The amount of data to be included is not a simple number of bytes, but how much (count) of what type (size) to write.

Return Value

Returns the number of times it could be written (number of items).

See Also

fread(), fsetbin(), write()

Notes

If the data to be written is binary data, do fsetbin() or fopen() in binary mode You must do this.

This function has been added to the standard library since ver. 1.2.

Example

#include <stdio.h>

main()

{

FILE \*fp;

int array[10];

int count;

/\* an array of integers in binary form to a file worddat.dat \*/

/\* Open in binary mode \*/

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

Format

# include <stdio.h>

int getc(fp)

FILE \*fp;

int getchar()

Description

getc() reads one character from the file indicated by fp. getchar() reads one character from the standard input. When the end of the file (end-of-file) is reached, it returns the value EOF.

Return Value

If it reads past the end of the file, it returns EOF. Otherwise, it returns the loaded character as an int.

See Also

getch(), getche()

Notes

When reading characters from a device (keyboard), line buffering is done according to the MSX-DOS2 specification. That is, it does not return from getc(), getchar() until the return key is pressed. getc() and getchar() return an int value.

Example

#include <stdio.h>

main()

{

FILE \*fp;

/\* Display the file from beginning to end \*/

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

Format

#include <conio.h>

char getch()

Description

You can also enter control characters such as Ctrl+C or Ctrl+Z. No line buffering is done (no templates are available) because there is no EOF, so if the standard input is redirected, it is better to check if it is an end-op file with eof(). It is not advisable to mix with other reading functions, such as read() to read from standard input (file handle 0) or getchar() to read from stdin.

Return Value

The characters entered are returned. You can also enter Ctr l+ C or Ctr l+ Z as characters.

See Also

eof(), getche()

Notes

This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

int get()

{

if (eof(STDIN))

return (EOF);

else

return (getch());

}

/\* Show characters entered from the standard input. \*/

main()

{

int c;

while ((c = get())) != EOF) {

if (c <' ') {

putchar(---);

c +=-'0';

}

putchar((char)c);

fflush(stdout);

}

}

getche

Format

#include <conio.h>

char getche()

Description

getsche() waits for input and returns a single character when it is entered. Control characters such as Ctrl+C or Ctrl+P work as a function of the character. If the standard input is redirected because there is no EOF, it is better to check if its an end-of-file with eof(). It is not advisable to mix with other reading functions, such as reading from standard input (file hand le 0) with read(), or reading from stdin with getchar(), etc.

Return Value

The characters entered are returned.

See Also

eof(), getch()

Notes

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

while(1){

putchar(getche());

fflush(stdout);

}

}

getcwd

Format

#include <direct.h>

char \*getcwd(cwd, n) /\* get current work directory \*/

char \*cwd;

int n;

Description

Returns the current drive and current directory in the form of d:\ path. cwd is a pointer to the area to store the current directory name n is its size. cwd is NULL, getcwd() allocates the area automatically. If n is insufficient to store the current directory, it fills the area and truncates the rest.

MSX-DOS2 specifies that the file name must be no more than 63 characters, so it is a good idea to reserve at least 64 bytes of space.

Return Value

The function returns NULL sein ca of a failure to acquire the current directory or an automatic allocation failure. Otherwise, it returns a pointer to the string indicating the current directory.

See Also

chdir(), mkdir(), rmdir()

Notes

If the current directory is the root directory, there will be a "\" at the end. This function has been added to the Standard Function Library since Ver. 1.2.

Example

#include <stdio.h>

char cd[64];

main()

{

chdir("\\"); /\* If the current drive is A, then \*/

getcwd(cd, sizeof(cd));

puts(cd); /\* 'A:\' is displayed \*/

chdir("bin"); /\* A drive has the bin directory \*/

puts(getcwd(cd, sizeof(cd))); /\* 'A:\B1N' is displayed \*/

getenv

Format

#include <stdlib.h>

char \*getenv(var)

char \*var;

Description

Acquires the value of a MSX-DOS2 environment variable var is the name of the environment variable you want to acquire. The value of the variable is in the area indicated by the return value as a string. The area for storing the variable place is allocated by alloc() each time and can be released with free() if it is no longer needed.

The boundary variables are set by MSX-DOS's SET command or by putenv().

Return Value

NULL if the variable was not found or if no space was allocated to store the value of the variable.

to the user. Otherwise, it returns the area where the variable value is stored.

See Also

alloc(), free(), putenv()

Notes

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

/\* Making temporary files \*/

char \*mktmp(fn)

{

char \*p;

static int num = 0;

if ((p = getenv("TMP"))) == NULL) /\* The directory to be used for tmp \*/

p = "a:\\";

strcpy(fn, p);

p = fn + strlen(fn); /\* points at '\0' \*/

if (\*(p - l) != '\\')

\*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");

exlt(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

Format

#include <stdio.h>

char \*gets(s, n)

char \*s;

int n;

Description

Reads a string from standard input (stdin). The string ends with '³" and is loaded into the area indicated by s. The size of the area is specified by 11. The string ends with a "³" followed by a "³" as the end character of the string. If the file reaches the end-of-file while reading, the ''³'' will not be entered, but will be set to "³". fgets() reads at most n-1 characters and places "³" at the end.

Return Value

The function returns NULL when the standard input reaches the end-of-file or if nothing is read. Otherwise, s is returned.

See Also

fgets(), fputs(), puts()

Notes

The MSX-C gets() behaves differently from the standard C gets(). The standard behavior of gets() does not have a "\" at the end of the string, but the MSX-C gets() has a "\n" at the end.

Example

#include <stdio.h>

char buf[256];

main()

{

FILE \*fp;

/\* Save the input from the keyboard to a file \*/

if ((fp = fopen("memo", "w")) == NULL) {

puts("File cannot make\n");

exit(1);

}

while (gets(buf, 256))

fputs(buf, fp);

fclose(fp);

}

inp

Format

#include <conio.h>

char inp(port)

unsigned port;

Description

Input one byte of data from the I/0 port indicated by port. The port number is specified in the range of 0 to 255.

Return Value

Returns a single byte of data from a 1/0 port as a char.

See Also

outp()

Example

#include <stdio.h>

main()

{

int i;

char knj[32];

/\* Read Kanji ROM \*/

for (i = 0; i < 32; i++)

knj[i] = inp(0xd9);

}

isalnum

Format

#include <ctype.h>

BOOL isalnum(c)

char c;

Description

Checks whether the characters are letters or numbers. The word, "a" to "z", "A" to "Z", "O" to "9".

Return Value

Returns TRUE if the character is a letter or a number. Otherwise, the function returns FALSE.

See Also

isalpha(), isdigit()

Notes

This function is a macro, so you cannot specify a seedling action for c. This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

/\* Is the specified string a name (identifier)? \*/

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("identifler\n");

}

fclose(fp);

}

isalpha

Format

#include <ctype.h>

BOOL isalpha(c)

char c;

Description

Checks if the letters are letters of the alphabet. Check if it is one of "a" to "z", "A" to "Z".

Return Value

Returns TRUE if the character is an alphabetic character. If not, it returns FALSE.

See Also

islower(), isupper()

Notes

Since this function is a macro, c cannot be specified for side effects.

Example

#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[l][2] != '\0')

puts("Argument must be drive name 'd:'\n");

exit(1);

...

}

isatty

Format

#include <io.h>

BOOL isatty(fd)

FD fd;

Description

If the file handle fd is a device, isatty() can be used to change the behavior of the command depending on whether the input or output target is a device file or a disk file. For example, in the UNIX page-by-page filter more, if the standard output is a device, it waits for a key input, but if it is redirected to a file, it is a do nothing filter.

Return Value

Returns TRUE if the file handle is a device. If it is a file or the file handle is not opened, FALSE is returned.

Notes

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

\*/ Is the standard output redirected? \*/

if (isatty(fileno(stdout)))

fputs("stdout is a device\n", stderr);

else

fputs("stdout is a file\n", stderr);

}

iscntrl

Format

#include <ctype.h>

BOOL iscntrl(c)

char c;

Description

Checks if a character is a control character. (the code is less than 0x20 and 0x7f characters in)

Return Value

The function returns TRUE if the character is a control character. Otherwise, it returns FALSE.

Notes

Since the function is a macro, C cannot specify any side effects.

Example

#include <stdio.h>

main()

{

int c;

FILE \*fp;

/\* Display the contents of the file, including the control characters \*/

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 -'O'));

} else

putchar((char)c);

}

}

isdigit

Format

#include <ctype.h>

BOOL isdigit(c)

char c;

Description

Checks whether a character is a numeric character (0 to 9).

Return Value

Returns TRUE if the character is a number. Otherwise, it returns FALSE.

See Also

isxdigit()

Notes

Since this function is a macro, c cannot specify any side effects.

Example

#include <stdio.h>

main(argc , argv)

int argc;

char \*argv[];

{

int i;

char \* p;

if (--argc < 1) {

puts("Usage: atoi <number>...");

exit(1);

}

argv++;

/\* Trying to do something like atoi() \*/

while (argo--) {

i = 0;

p = argv++

while (isdigit(\*p))

i = i \* 10 + \*p++ - '0';

printf("%d\n", i);

}

}

iskanji

Format

#include <ctype.h>

BOOL iskanji(c)

char c;

Description

Checks if a character is the first byte of an ... . (0x81 to 0x9f, 0xe0 to 0xfc in code)

Return Value

Returns TRUE if the character is the first byte of a Kanji character. If not, it returns FALSE.

See Also

3.5 Methods of Kanji Processing

iskanji2()

Notes

This concern The number is a macro, so c specified with any side effects. You can't tell it is a Kanji character or not exactly looking at all the characters from the beginning of the string you are checking, unless you are looking at Kanji or ANK.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

/\* Find the directory delimitation from the directory name containing kanji \*/

char \*pathdelimit(s)

char \*s;

{

while (\*s) {

if (\*s =='³³')

return (s); if (lskanJ I (\*s))

s++

B+; B+;

}

return (NULL);

}

phoenixal n( argc, argv)

intargc; char\*argv(];

{

char Thing p; if (--argc < 1) {

puts("Usage: pdlmt <path>¥n"); exlt(l);

}

argv++;

P = :t:argv;

profanity (p = pathdelimit(p)) {

\*P ='³';

puts(:t:argv);

\*P++ ='¥¥';

}

iskanji2

Format

#include <ctype.h>

BOOL iskanji2(c)

char c;

Description

Checks if a character is the second byte of a kanji. (0xin code 40 to 0x7e, 0x80 to 0xfc)

Return Value

Returns TRUE if the character is the second byte of a Kanji character. Otherwise, the function returns FALSE.

See Also

3.5 Methods of Kanji Processing

iskanji()

Notes

Since this function is a macro, c cannot be specified for side effects. You have to look at all the characters from the beginning of the string you are trying to check to see if they are Kanji or ANK to be sure.

This function has been added to the standard library since Ver. 1.2.

Example

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 = l;

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

Format

#include <ctype.h>

BOOL islower(c)

char c;

Description

Checks if the characters are lowercase.

Return Value

Returns TRUE if the character is a lower case one. Otherwise, it returns FALSE.

See Also

isupper(), tolower(), toupper()

Notes

Since this function is a macro, c cannot be specified with any side effects.

Example

#include <stdio.h>

main()

{

int c;

/\* output only lowercase letters on keyboard input \*/

while ((c = getchar())) != EOF) {

if (islower((char)c))

putchar((char)c);

}

}

isspace

Format

#include <ctype.h>

BOOL isspace(c)

char c;

Description

Characters are space characters (space, tab, linefeed, home, clear, carriage return) or not. 9 (in code from 0x20 and 0x0-9 in code of 0x0d)

Return Value

Returns TRUE if the character is a space character. If not, I will return FALSE.

Notes

Since the function is a macro, c cannot specify any side effects.

Example

#include<stdio.h>

/\* Skip to characters other than the base \*/

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

Format

#include <ctype.h>

BOOL isupper(c)

char c;

Description

Checks whether a character is uppercase or not.

Return Value

Returns TRUE if the character is in uppercase. Otherwise, the function returns FALSE.

See Also

islower(), tolower(), toupper()

Notes

Since this function is a macro, c cannot be specified with any side effects.

Example

#include <stdio.h>

main()

{

int c;

/\* Display only uppercase letters when typing from the keyboard \*/

while ((c = getchar()) != EOF) {

if (isupper((char)c))

putchar((char)c);

}

}

isxdigit

Format

#include <ctype.h>

BOOL isxdigit(c)

char c;

Description

Checks if a character is a hexadecimal character ("0" to "9", "a" to "f", "A" to "F").

Return Value

Returns TRUE if the character is a hexadecimal character. If not, it returns FALSE.

See Also

isdigit()

Notes

Since this function is a macro, c cannot be specified for side effects. This function has been added to the standard library since Ver. 1.2.

Example

#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 hexadecimal version of atoi () \*/

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

Format

#include <conio.h>

BOOL kbhit()

Description

Checks if the keyboard was pressed. The function returns the state of the keyboard buffer, regardless of the standard input redirection.

Return Value

If there are characters in the keyboard buffers returns TRUE. If not, the function returns FALSE.

Notes

1f. I the standard input is redirected to a file Once it becomes TRUE, no matter how many characters are from the standard input, it does not go FALSE. open() to open a "con" and then read( ) to accept the input characters from the keyboard.

Example

#include <stdio.h>

char buf[256];

/\* Clear the keyboard buffer \*/

killbuf()

{

while (kbhit()) /\* keyboard buffer clear \*/

getch();

}

main()

{

unsigned i;

for (i = 0; I < 65535; i++)

; /\*push any key while waiting here \*/

killbuf(); /\* kill keyboard buffer \*/

gets(buf, 256);

puts(buf)

}

longjmp

Format

#include <setjmp.h>

VOID longjmp(env, val)

jmp\_buf env;

int val;

Description

longjmp() can jump across the function by setting a buffer env of type jmp\_buf set in setjmp() as a parameter, and return to the next (where the value is returned from the function) of setjmp() with env setjmp(). You can also pass the return value of setjmp() with val. However, val must be a non-zero value.

Return Value

No, there is not.

See Also

setjmp()

Nots

If val is 0, the value is changed.

Example

#include <stdio.h>

jmp\_buf mbuf;

gofunc()

{

int err = 0; /\* Clear the error flag. \*/

...

if (err) /\* Is there an error? \*/

longjmp(mbuf, err); /\* Forced return if an error occurs \*/

}

main()

{

static int i;

if (i = setjmp(mbuf )) {

/\* Returned from longjmp() \*/

printf("Return from longjmp() with code %d\n", i);

} else { /\* First setjmp() \*/

printf("setjmp is executed\n");

gofunc();

}

}

max

Format

#include <stdlib.h>

int max(x, y)

int x, y;

Description

Compares integers x and y and returns the non-smaller one. max() is used to choose the larger of the two numbers.

Return Value

Returns the non-smaller number of x and y. If the values are equal, it returns the value.

See Also

min()

Example

#include <stdio.h>

main()

{

/\* 5, 2, 7 are displayed, respectively, whichever is the higher number \*/

printf("%d %d %d\n", max(l , 5), max(-3, 2), max(7, 7));

}

memcpy

Format

#include <memory.h>

VOID memcpy(dest, source, length)

char \*dest, \*source;

size\_t length;

Description

Copies the content from the location indicated by source to the location indicated by <les t> by the

length of the memory.

The difference with string manipulation functions such as strcpy( ) is that the "³" is not used as the end mark for the operation. This is the same as movmem(), except that the order of the parameters is different, and the behavior is the same.

Return Value

No, there is not.

See Also

movmem()

Notes

Be careful not to destroy the work area, such as MSX-DOS. This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

#define CMDLIN (char\*)0x80

char buf[128];

main()

{

/\* Make a copy of the command line \*/

memcpy(buf, CMDLIN, sizeof(buf));

buf[(int)buf[0]+l] = '\0';

puts(buf + 1);

}

memset

Format

#include <memory.h>

VOID memset(dest, byte, length)

char \*dest, byte;

size\_t length;

Description

It fills the memory with the length of length from the memory indicated by dest by the value of byte, the unit of length is in bytes.

Return Value

No, there is not.

See Also

setmem()

Notes

Be careful not to destroy the work area, such as MSX-DOS. This function has been added to the standard library since version. This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

int count[10];

main()

{

int c;

int count[10];

int other;

/\* Initialize the array buf to 0 \*/

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

Format

#include <stdlib.h>

int min(x, y)

int x, y;

Description

Compare the values of integers x and y and return the lesser value. min() is used to choose the lesser of two numbers.

Return Value

Returns the number of x and y, whichever is not larger. If the values are equal, it returns the value.

See Also

max()

Example

#include <stdio.h>

main()

{

/\* Display the smaller of each 1, -3, 7 will be displayed \*/

printf("%d %d %d\n", min(l , 5), min(-3, 2), min(7, 7));

}

mkdir

Foramt

#include <direct.h>

STATUS mkdir(path)

char \*path;

Description

Creates a subdirectory path can be specified from the root directory or the current directory. Specifying a drive in the path creates a subdirectory for the drive ([d : ][ \] path.

Return Value

It returns OK if a subdirectory has been created and ERROR if it fails.

See Also

chdir(), rmdir(), getcwd()

Notes

mkdir() cannot create a subdirectory under a non-existent subdirectory. Use mkdir() to create the parent directory of the subdirectory you want to create. Also, if a file with the same name exists, it cannot be created.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

/\* Create a supplementary directory \*/

if (chdir("\\bin")) == ERROR) {

puts("Maklng '\\bin' Subdirectory.\n");

mkdir("\\bin");

}

/\* Now subdirectoy '\bin' must exist \*/

}

movmem

Format

#include <memory.h>

VOID movmem(source, dest, length)

char \*dest, \*source ;

size\_t length;

Description

of memory Copy the content from the position indicated by source to the position indicated by <lest for the length of length. It differs from string manipulation functions, such as strcpy() , in that the ''³'' is not used as an exit mark for the operation. It works the same way as memcpy(), except that the order of the parameters is different.

Return Value

No, there is not.

See Also

memcpy()

Notes

Be careful not to destroy the work area of MSX-DOS and other devices.

Example

#include <stdio.h>

#define CMDLIN (char\*)0x80

char buf[128];

main()

{

/\* Make a copy of the command line \*/

movmem(CMDLIN, buf, sizeof(buf));

buf[(int)buf[0]+l] = '\0';

puts(buf + 1);

}

open

Format

#include <io.h>

FD open(filename, mode)

char \*filename;

int mode;

Description

Opens the file indicated by filenamech, whican include the drive and pathname. mode specifies the open mode of the file that is about to be opened. The following values are available for the file open mode.

O\_RDONLY Reading only

If the file is missing, an error is reported.

O\_WRONLY Write only

If the file already exists, it is topped up when writ ite() is done.

0\_RDWR Both reading and writing

Return Value

An ERROR is returned if the file fails to open, for example, if the file is missing. Otherwise, the file handle is returned.

See Also

close(), creat(), read(), write()

Notes

open() is a low-level input/output function.

Example

#include <stdio.h>

static char str[] = "test data";

{

FD fd;

/\* Open the file in write mode \*/

if ((fd = open("openfile.dat", O\_WRONLY)) == ERROR) {

puts("File cannot open\n");

exit(1);

}

write(fd, str, strlen(str));

close(fd);

}

outp

Format

#include <conio.h>

VOID outp(port, val)

unsigned port;

char val;

Description

Outputs data val, 1 byte to the 1/0 port indicated by port. The port number is specified in the range from 0 to 255.

Return Value

No, there is not.

See Also

inp()

Notes

Depending on the output port, it may cause the MSX to malfunction.

Example

#include <stdio.h>

main()

{

int i;

char knj[32];

/\* Kanji Read ROM Macro characters \*/

outp(0xd8, 0);

outp(0xd9, l);

for (i = 0; i < 32; i++)

knj[i] = inp(0xd9);

}

printf

Format

#include <stdio.h>

STATUS printf(format[, argl, arg2 , ...])

char \*format;

Description

printf() performs a formatted conversion and writes the output to standard output (stdout) . Arguments arg1, arg2, . . . passes the value to be converted by the control string, which is represented by format or a pointer to the string. The control string consists of normal characters that are output as they are without conversion and a conversion specification in the following format

%[-][[0]w][.n]c

The conversion specification starts with a percent sign (%) and is mapped to a value such as arg1, arg2, etc., depending on the number H that appears. A minus sign (1) indicates that the field is left-justified (usually right-justified). w"" indicates the minimum width of the field in decimal, and if w starts with 0, the field margin is filled with "O''' instead of soothes 's are. The maximum number of characters to be output as a decimal number in a character string, valid only in the case of string variation (n conversion %s). The character c is one of the following conversion characters.

d Display the corresponding argument as a signed decimal number

u Display the corresponding argument as an unsigned decimal number

o Display the corresponding argument as an unsigned octal x Display the corresponding argument as an unsigned hexadecimal c Display the character as the character code for the corresponding argument

s Display a string with the corresponding argument as a pointer to the string

Anything that is not one of the conversion characters shown above will be output as is. The character sequence allows you to output a single percent sign "%".

Return Value

If there are no errors during output, OK is returned. Otherwise, an ERROR is returned.

See Also

fprintf(), sprintf()

Notes

c To print a value of type har by "%d", "%x''', "%0", pass an int cast as an argument. This is because the caller passes a char (a single-byte value) but the printf() is passed as an int, to avoid printing the upper byte of the int type as an undefined value. Do the same for the BOOL, STATUS and TINY types as well as the char type. There is no problem to display with "c"%.

char c;

printf("%d %02x\n", (int)c, (int)'A');

Because printf() is a variable-parameter function, the following declarations are required beforehand

STATUS printf(.);

This declaration is included in the header file stdio.h.

printf() is a superset of Jun-C in UNIX.

Example

#include <stdio.h>

int val = 1234;

char c ='A';

static char str[] = "printf test string";

main()

{

/\* Display numeric values and characters and strings in decimal notation 10 16 8 \*/

printf ("decimal %d\nhexadecimal %04x\nonoctal %06o", val, val, val);

printf("char '%c'\nstring '%s'\n", c, str);

}

putc, putchar

Format

#include <stdio.h>

STATUS putc(c, fp)

char c;

FILE \*fp;

STATUS putchar(c)

char c;

Description

putc() outputs a single character to the file indicated by fp. putchar() outputs a single character to the standard output (stdout). If the character C is ''³'' and the file is in text mode, output as two characters, ''³ r'' and ''³'', and if the file is line buffering, set the buffer to Flash.

Return Value

Returns OK if one character is output. Otherwise, it returns ERROR.

See Also

fsetbin(), fsettext()

Notes

The character to be passed is a char, so be sure to cast it to a char if you want to output an int return such as getc().

Example

#include <stdio.h>

main()

{

int c;

FILE \*fp;

/\* Output keyboard input to standard output and file \*/

if ((fp = fopen("file", "w)) == NULL) {

puts("Flle cannot make\n");

exit(1);

}

while ((c = getchar()) != EOF) {

putchar((char)c); /\* Output to standard output \*/

putc((char)c, fp); /\* output to a file \*/

}

}

putenv

Format

#include <stdlib.h>

STATUS putenv(env) /\* set environment \*/

char \*env; /\* <varname>=<value> \*/

Description

Sets the value of the environment variable for MSX-DOS2. The parameter string env must be of the form "environment variable name = value". If the value is omitted, the environment variable is cleared. The value of the setting environment can be acquired with getenv(). You can check it with SET or use it in another command, since the environment value remains after the current command has finished.

Return Value

If the environment has been set up correctly, the function returns OK. When not Returns ERROR.

See Also

getenv()

Notes

The boundary variable cannot be set unless there is an equal sign "=" in the string indicating the parameter env. The function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

main()

{

/\* After setting up the environment, acquire and inspect the value \*/

if (putenv("MYENVIRON=value of kankyou") == ERROR)

puts("putenv Failed");

if (strcmp(getenv("MYENVIRON"), "value of kankvou"))

puts("Failed");

else

puts("OK");

}

puts

Format

#include <stdio.h>

STATUS puts(s)

char \*s ;

Description

Outputs a string s to the standard output (stdout). It does not add "³" to the output. The string s must be terminated by a null character. Null characters will not be printed. eriseI"fth³" in the string, it is output as ''³'' + "³ n" in text mode.

Return Value

If an error occurs during the output, an ERROR is returned. If it was otherwise, OK was returned The first thing to be done is to get a new window.

See Also

fgets(), fputs(), gets()

Notes

MSX-C puts() behaves differently from the standard C puts(). Standard behavior puts() Print the string and then "\n", but not the MSX-C puts().

Example

#include <stdio.h>

char buf[ 256];

main()

{

FILE \*fp;

/\* Display the contents of the file newgame.dat \*/

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

Format

#include <stdlib.h>

VOID qsort(base, nel, width, compar)

char \*base;

unsigned nel, width;

int (\*compar)();

Description

Sort the data in ascending order. base is the first address of the data, nel is the number of data to sort, and The width is the size of each element in the data in bytes.

compar is a pointer to a function that takes two pointers (x and y) and returns a value of type int, as shown below

Positive \*x > \*y

0 \*x = \*y

Negative \*x < \*y

Return Value

No, there is not.

Example

#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));

}

/\* Display the string array device before and after sorting \*/

main()

{

int i;

for (i = 0; i < 8; i++)

printf ("%s\n", device[1]);

puts("Sorting..\n");

qsort(device, 8, sizeof(char \*), compare);

for (i = 0; i < 8; i++)

printf("%s\n", device[i]);

}

read

Format

#include <io.h>

int read(fd, buf, bytes)

FD fd;

char \*buf;

size\_t bytes;

Description

Reads data from the file indicated by the fd. The length of the data to be read is specified in bytes, and the length of the loaded data is specified in file. The data will be stored in the area from buf.

Return Value

The file handle is not opened, the file is in write mode when opened, the file is in end- octave mode. The function returns 0 if it is one of the following: the number of files has been reached. Otherwise, the actual number of bytes loaded is returned. ,e it may be less than bytes.

See Also

close(), creat(), open(), open(), open()

Notes

read() is a low-level input/output function.

Example

#include <stdio.h>

char buf[1024];

main()

{

FD fd;

/\* Read the file readtest.dat every 1024 bytes to the end of file \*/

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

Format

#include <io.h>

STATUS rename(oldname, newname)

char \*oldname, \*newname;

Description

You can specify the drive and pathname in oldname, but newname is ignored and only the basename is used. You cannot specify wildcards in the oldname or move the file.

Return Value

The function returns OK if the file can be renamed, or ERROR if not.

Example

#include <stdio.h>

main()

{

/\* Change the name of the file cf.com to cfv120.com \*/

rename("\\bin\\cf.com placement". "cfvl20.com") ;

}

rmdir

Format

# include <direct.h>

STATUS rmdir(path)

char \*path;

Description

Removes a subdirectory, either from the root directory or the current directory. Specifying a drive in path removes the drive's subdirectory. d( [: ] [³] path.) However, if a file or directory is in the directory you want to delete, you will not be able to delete it.

Return Value

OK if the subdirectory was removed else returns ERROR.

See Also

chdir(), mkdir(), getcwd()

Notes

This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

main()

{

/\* Deleting the subdirectory \bin \*/

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

Format

#include <malloc.h >

VOID rsvstk(n)

size\_t n;

Description

Allocate n bytes as a stack area. The stack area here is the size for the rest of the stack when the area is allocated by sbrk(). Note that this is not the size of the entire stack. By default, 1000 bytes are allocated. See 4.3.3 Memory Management Functions" for details.

Return Value

There is no return value.

See Also

alloc(), free(), sbrk()

Notes

The size of the stack area is managed in a single area, so be careful if you change it outside of the main program.

Example

#include <stdio.h>

bigary()

{

char buf[1000];

...

}

main()

{

/\* Allocate a larger stack area to use larger arrays \*/

rsvstk(2500);

bigary();

rsvstk(1000); /\* Undo it \*/

}

sbrk

Format

# include <malloc.h>

char \*sbrk(n)

size\_t n;

Description

Memory Allocate n bytes. sbrk() is a low-level memory management function. The allocated space cannot be released, so it is better to use it when the program finishes using it. It is possible to share with the high-level memory management function alloc(), but it is difficult to get contiguous regions, so In programs that use a lot of alloc() and free (), the efficiency is reduced. See 4.3.3 Memory Management Functions for details.

Return Value

If the area is allocated, the pointer to the top of the area is returned. If there is not enough memory, an ERROR is returned.

See Also

alloc(), free(), rsvstk()

Example

#include <stdio.h>

main()

{

char \*p;

/\* Allocate space for the buffer \*/

if ((p = sbrk(l28)) != ERROR)

setvbuf(stdin, p, \_IOFBF, 128);

}

scanf

Format

#include <stdio.h>

int scanf(format[, arg1, arg2 , ...])

char \*format;

Description

scanf() provides input with formatting conversion from the standard input (stdin). The control string given by format consists of the characters to be converted and the characters that should match the input stream arg to sub traced number, arg2, ... must be a pointer to the variable used to store the resultant value of the conversion.

The format of the conversion specification is as follows

%[\*]c

The conversion specification starts with a percent sign (%). An asterisk (\*) can be placed right after the percent sign to specify that the obtained value should be discarded without assigning it. The character c is a format specifier and can be one of the following The parentheses are the types of variables that are normally stored.

d Assigns the input string to the variable as a signed decimal number. (int type)

u Assigns the input string to the variable as an unsigned decimal number. (unsigned)

o Assigns the input string to the variable as an unsigned octal number. (unsigned)

x Assigns the input string to the variable as an unsigned hexadecimal number. (unsigned)

c Assigns a single input character to the variable(char)

s Assigns the input string to a space or other delimiter. (char array)

When assigning a number, it assigns a non-numeric character to the input string. That is, new h assigning a signed decimal number in "%d" it is seen as a decimal number from() to any character other than 9. Numerical overflows are ignored.

Return Value

scanf() returns the actual number of items assigned to it. If the return value is 1 and 3 pointers to the variables are specified for 3 parameters no values are assigned after the second one. The value EOF is returned when the standard input is an end-op file.

See Also

fscanf(), sscanf()

Notes

Note that the linefeed character "³" may be substituted when substituting a character with "%c". scanf() is a variable parameter function, so the following declarations are required beforehand.

int scanf(.);

This declaration is contained in the header file stdio.h. scanf() is a subset of the UNIX standard C.

Example

#include <stdio.h>

char buf[100];

main()

{

int val, n;

char c;

/\* Enter a decimal number, a character, or a string from a keypad. \*/

n = scanf("%d %c %s", &val, &c, buf);

prirtf("%d matched decimal %d:char '%c':string '%s'\n", n, val, c, buf)

}

sensebrk

Format

#include <conio.h>

VOID sensebrk()

Description

Do a pre-check. If Ctrl+C or Ctrl+STOP is pressed while the function is being executed, it returns to the command level - and another function that inputs or outputs to the keyboard or screen, except sensebrk(), returns to the command level with Ctrl+C or Ctrl+STOP.

Return Value

There is no return value

Notes

If the program is aborted by Ctrl+C, Ctrl+STOP, the descendant-level entry/exit coverage will not be flushed.

Example

#include <stdio.h>

char buf[100];

int ary[1000];

/\* do an integer bubble sort. Can be stopped in the middle with Ctrl+C \*/

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(); /\* if break check \*/

if (array[i] > array[j]) {

tmp = arrav[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

Format

#include <stdio.h>

VOID setbuf(fp, buf)

FILE \*fp;

char \*buf; /\* points allocated buffer(size BUFSIZ) \*/

Description

Sets the file buffer. setbuf() can only be used to set no buffering or full buffering. Use setvbuf() if you need detailed settings.

buf is a pointer to the buffer. The size of the buffer must be BUFSIZ. If buf is NULL, no buffering is done (no buffering).

buf == NULL At the time of

setvbuf(fp, NULL, \_IONBF, 1);

buf != buf At the time of

setvbuf(fp, buf, \_IOFBF, BUFSIZ);

behavior.

Return Value

No, there is not.

See Also

4. 3.1 B) Buffering of high-level input/output functions

setvbuf()

Notes

setbuf() can be executed only when there is no data input or output after opening the file with fopen().

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

FILE \*fp;

/\* Use the file without buffering \*/

if ((fp = fopen("test" , "w")) == NULL) {

puts("File cannot make\n");

exit(1);

}

setbuf(fp, NULL) /\* No buffering \*/

}

setjmp

Format

#include <setjmp.h>

int setjmp(env)

jmp\_buf env;

Description

setjmp() is used in combination with longjmp(). setjmp() sets the current stack pointer etc. to env. Then longjmp() can be called using this env to restore the value set in setjmp() and jump beyond the function immediately after setjmp(). It may be useful if an error occurs during the deep nesting of the function and it is used to return control directly to the higher level routine.

Return Value

Returns 0 if the current state is evacuated. When you come back from the longjmp() function, you can use the longjmp. It returns the number of the second parameter of By this number, the higher level routines will be able to determine what causes You can find out if you came back from setjmp().

See Also

longjmp()

Notes

the function uses setjmp() and the variable is assigned to a register, the value of the variable is not saved when it is returned with Iongjmp().

Example

#include <stdio.h>

jmp\_buf mbuf;

gofunc()

{

int err = 0; /\* Clearing the error flag \*/

...

if (err) /\* Is there an error? \*/

longjmp(mbuf , err); /\* Forced back if an error occurs \*/

...

main()

{

static int i;

if (i = setjmp(mbuf)) {

/\* Returned from the from longjmp() \*/

printf("Return from longjmp() with code %d\n", i);

} else { /\* first setjmp() \*/

printf("setjmp() is executed\n");

gofunc();

}

}

setmem

Format

#include <memory.h>

VOID setmem(dest, length, byte)

char \*dest, byte;

size\_t length;

Description

It fills the memory with the length of length from the memory indicated by dest by the value of byte. The unit of length is bytes, and the behavior is the same as in memset(), except the order of the parameters is different.

Return Value

No, there is not.

See Also

memset()

Notes

Be careful not to destroy a work area such as the MSX-DOS.

Example

#include <stdio.h>

int count[10];

main()

{

int c;

int count[to];

int other;

/\* Initialize the array buf with 0 \*/

setmem(count, sizeof(count), (char)0);

other = 0;

while ((c = getchar()) != EOF) {

if (lsdlgit((char)c))

count[c - '0']++;

else

other++;

}

for (c = 0; c < 10; c++)

printf("%d '%d'(s)\n", count[c], c);

}

setvbuf

Format

#include <stdio.h>

STATUS setvbuf(fp, buf, mode, size)

FILE \*fp;

char \*buf;

int mode, size;

Description

Sets parameters for the file buffers indicated by fp. You can set the buffer buffer area, buffer size and buffering method.

buf is passed a pointer to the buffer; if NULL is specified, the system allocates it automatically. (fclose()) ignored if \_IONBF is the (m)ode.

size specifies the size of the buffer, if you don't specify a buffer size in fopen() has a buffer size of BUFSIZ (1024). If m ode is \_IONBF it is ignored.

mode specifies one of the following three buffering modes.

1. \_IONBF: No buffering. Generally, it is specified for the device.
2. \_IOLBF: does row buffering. When the output comes "11" and the backyard is A flurry of fluff I will.
3. \_IOFBF: does full buffering. It does not flush until the buffer is full; it is in this mode immediately after fopen().

Return Value

Returns an ERROR if m ode is any value other than the above. Otherwise, it returns OK.

See Also

4.3.1 B Buffering high-level input/output functions

fflush(), fopen(), setbuf()

Notes

setvbuf() can be executed only when there is no data input or output after opening the file with fopen().

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

main()

{

FILE \*fp;

/\* Files Used in a 128-bit buffer \*/

if ((fp = fopen("test", "w")) == NULL) {

puts("File cannot make\n");

exit(1);

}

setvbuf(fp, NULL, \_IOFBF, 128);

/\* Buffer to 128 bytes \*/

}

sprintf

Format

#include <string.h>

STATUS sprintf(buffer, format[, argl, arg2 , ...])

char \*buffer, \*format;

Description

sprintf() performs a formatted transformation and stores the output in the area indicated by the buffer. For the control string, it is the same as printf (), see the section on control strings.

Return Value

If there are no errors during output, OK is returned. Otherwise, ERROR is returned.

See Also

fprintf(), printf()

Notes

Since printf() is a variable parameter function, the following declarations are required beforehand.

STATUS sprintf(.);

This declaration is included in the header file string.h.

sprintf() is a standard UNIX C superset.

Example

#include <stdio.h>

/\* years and months to string \*/

char \*strdate(buf, y, m, d)

char \*buf; /\* Pointer to a region of 11 bytes or more \*/

int y, m, d;

{

sprintf(buf, "%04d/%02d/%02d", y, m, d);

return (buf);

}

main(argc, argv)

int argc;

char \*argy[];

{

char buf[ll];

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

Format

#inlcude < string.h >

int sscanf(line, format[, argl, arg2,...])

char \*line, \*format;

Description

sscanf() provides the same formatted input as scanf(), except that while scanf() reads input from the standard input (stdin), it takes input from the string pointed to by the line, and the string indicated by the line must end with a null character (³). Null characters themselves are not entered or .mat and subsequent parameters are the same as in scanf(), see the section below.

Return Value

sscanf() returns the actual number of items assigned to the function. If the return 1 and three pointers to the variables are specified for the parameters, no values are assigned after the second one. When the end of the string line is reached, the value EOF is returned.

See Also

fscanf(), scanf()

Notes

Since sscanf() is a variable parameter function, the following declarations are required beforehand

int sscanf(.);

This declaration is included in the header file string.h. sscanf() is a standard UNIX C superset.

Example

#include <stdio.h>

main()

{

int y, m, d;

int n;

/\* Get the year and month from a string \*/

n = sscanf ("l989/l/17", "%d/%d/%d", &y, &m, &d);

printf("%2d/%2d/%4d\n", m, d, y);

}

strcat

Format

#include <string.h>

char \*strcat(d, s)

char \*d, \*s;

Description

Adds the string s after the string d. It is assumed that area needed to add s is present. The string d must also end in \0.

Return Value

d is returned as is.

See Also

strcpy(), strncat()

Example

#include <stdio.h>

char fname[64];

/\* Create a full path name by giving it a drive, path, base name, and extension \*/

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) {

case1:

puts("Usage: fullpath [<d:>] [<path>] [<base>] [<ext>] exit(1);

case2:

fullpath(fname, "a:", "\tmp", argv[l], "$$$");

break;

case3:

fullpath(fname, "a:", "tmp", argv[l], argv[2]);

break;

case4:

fullpath(fname, "a:", argv[l], argv[2], argv[3]);

break;

case5:

fullpath(fname, argv[l], argv[2], argv[3], argv[4]);

break;

}

puts(fname);

putchar('\n');

}

strchr

Format

#include <string.h>

char \*strchr(s, c)

char \*s, c;

Description

Finds the first position of the character c in the string S. c may be "\0".

Return Value

If the character c is found, it returns the digit four. Otherwise, N returns NULL.

Note

Strings of Chinese characters are not supported.

This function has been added to the standard library since Ver. 1.2.

Example

#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--) {

/\* Show how many periods are in a string \*/

head = s = \*argy++;

while (s = strchr(s, '.')) {

n++;

s++;

}

printf("%s includes %d'.'(s)\n", head, n);

}

}

strcmp

Format

#include <string.h>

int strcmp(s, t)

char \*s, \*t;

Description

Compares a string s with a string t. The comparison of strings starts at the beginning of two strings and compares them one character at a time. Both ends of the string or the comparison ends when the characters are different.

Return Value

The function returns 0 for the same string. If it is different, the value is not fixed, but you can see what state it is in in the range.

Positive String s > String t

0 String s = String t

Negative String s < String t

See Also

strncmp()

Notes

If the strings are the same, 0 is returned.

Example

#include <stdio.h>

BOOL debug = FALSE; /\* Clearing the debugging mode \*/

main(argc, argv)

int argc;

char \*argv[];

{

/\* Enable debug mode if the first parameter is "-d". \*/

if (argc > 1 && strcmp(argv[l], "-d") == 0) {

debug = TRUE;

argc--;

argv++;

}

...

}

strcpy

Format

#include <string.h>

char \*strcpy(d, s)

char \*d, \*s;

Description

Copies the strings to the area indicated by d. Copy d to the ''\0" of s. In this case, the s

Suppose that there is enough space for d.

Return Value

d is returned as is.

See Also

strcat(), strncpy()

Example

#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

Format

#include <string.h>

size\_t strlen(s)

char \*s;

Description

Tests the length of the string s. The length does not include the last "\n".

Return Value

Returns the length of the string s.

Example

#include <stdio.h>

char fname[64];

/\* Add a '\' if the string does not end with a '\' \*/

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, arsv)

int argc;

char \*argy[];

{

if (--argc < 1) {

puts("Usage: specpath <path>\n");

exit(1);

}

strcpv(fname, argv[1]);

adddelimit(fname);

strcat("temp.$$$");

puts(fname);

putchar('\n');

}

strlwr

Format

#include <string.h>

char \*strlwr(s)

char \*s;

Description

Converts a string s from lowercase to uppercase. The function modifies the string, denoted by s, directly.

Return Value

Returns S as it is.

See Also

strupr()

Notes

Strings of Kanji characters are not supported.

This function has been added to the standard library since version 1.2.

Example

#include <stdio.h>

main(argc , argv)

int argc;

char \*argv[];

{

FILE \*fp;

if ({fp = fopen(argv[1], "r")) == NULL) {

/\* Output command name, file name in lowercase. \*/

fprintf(stderr, "%s: %s not found\n", strlwr(argv[0]), argv[l]);

exit(1);

}

...

}

strncat

Format

#include <string.h>

char \*strncat(d, s, n)

char \*d, \*s;

unsigned n;

Description

Adds up to n characters from the beginning of the string s to the end of the string d, followed by "\0". s must have at least n characters of space for adding s. If nis 0, nothing is added.

Return Value

d is returned as is.

See Also

strcat()

Notes

Strings of Kanji characters are not supported.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

char fname[64];

/\* Add the extension with no more than four characters, including '.' \*/

addext(ext)

char \*ext;

{

strncat(fname, ext, 4);

}

main(argc, argv)

int argc;

char \*argv[];

{

if (--argc < 1) {

puts("Usage: addext <extenslon>\n");

exlt(1);

}

strcpy(fname, "tmp.");

strncat(fname, argv[l], 4);

puts(fname);

putchar('\n');

}

strncmp

Format

#include < string.h>

int strncmp(s, t, n)

char \*s, -t;

unsigned n;

Description

String s and t from the top to the maximum n Character comparison. The comparison of strings starts at the beginning of two strings compares them character by character. n Compare characters, both end in a string, or the characters are different The comparison ends when you have finished.

Return Value

The function returns 0 if n characters are the same string or less than n characters are the same for both strings until the end of the string. If they are different, the value is not determined, but you can find out what state they are in in the range.

Positive String s > String t

0 String s = String t

Negative String s < String t

See Also

strcmp()

Note

Strings of Kanji characters are not supported.

The function returns 0 if the two strings are identical within n characters from the beginning. This function has been added to the standard library since Ver.1.2.

Example

#include <stdio.h>

/\* To find out if the beginning of the string is "temp" \*/

istmp(fn)

char \*fn;

{

return (!strncmp(fn, "temp", 4));

}

main(argc, argv)

int argc;

char \*argv[];

{

if (--argc < 1) {

puts("Usage: lstmp <dst>\n");

exit(1);

}

if (istmp(argv[l])) {

puts("Don't specify 'temp????' file\n");

exit(1);

}

puts(argv[1]);

}

strncpy

Format

#include <string.h>

char \*strncpy(d, s, n)

char \*d, \*s;

unsigned n;

Description

Copies up to n characters from the beginning of the string s to d. It is assumed that d has a space of n characters.

Return Value

d is returned as is.

See Also

strcpy()

Notes

Kanji strings are not supported.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

#define SIZE 20

char shell[SIZE];

main()

{

/\* Put the value of the environment variable SHELL into the array shell \*/

if ((p = getenv("SHELL"))) != NULL) {

strncpy(shell, P, SIZE);

free(p);

}

puts(shell);

}

strupr

Format

# include <string.h>

char \*strupr(s)

char \*s;

Description

Converts a string s to uppercase letters. The function modifies the string shown by s directly.

Return Value

s is returned as is.

See Also

strlwr()

Notes

Strings of Kanji characters are not supported.

This function has been added to the standard library since Ver. 1.2.

Example

#include <stdio.h>

/\* Display the arguments in all capital letters. \*/

main(argc, argv)

int argc;

char \*argv[];

{

int i;

for (i = 0; i < argc; i++)

printf("[%d]: %s\n", i, strupr(argv[i]));

}

tolower

Format

#include <ctype.h>

char tolower(c)

char c;

Description

If c is uppercase, it returns the lowercased version of the character.

Return Value

If c is a capital letter, it returns the lower-cased version. If not, c returned as is.

See Also

islower(), isupper(), toupper()

Notes

In MSX-C tolower() is a function.

Example

#include <stdio.h>

/\* Convert string s uppercase to lowercase and lowercase to uppercase \*/

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(l);

}

argv++;

while (argc--) {

puts(\*argv);

puts (strchg(\*arg v));

}

}

toupper

Format

#include <ctype.h>

char toupper(c)

char c;

Description

If c is a lowercase letter, it returns the capitalized letter.

Return Value

If c is a lowercase letter, it returns the capitalized letter. If not, c returned as is.

See Also

islower(), isupper(), tolower()

Notes

In MSX-C, toupper() is a function.

Example

#include <stdio.h>

int x, y;

/\* Wait for input, and change the position if it is 'H', 'J', 'K', or 'L' \*/

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

Format

#include <stdio.h>

STATUS ungetc(c, fp)

char c;

FILE \*fp;

VOID ungetch(c)

char c;

Description

ungetc() returns one character to the file indicated by fp. ungetch() returns one character to standard input. Each returned character can be received by the next execution getc(), getchar(). The function is often used for pre-reading the input file.

Return Value

ERROR if the file was opened in write mode and there is no place to put the characters back is returned. If the characters are restored, OK is returned.

Example

#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[l], "r") == NULL) {

puts("File not found\n");

exit(1);

}

/\* Display the number of repetitions of non-substrate and sub-substrate characters up to EOF \*/

while ((c = getc(fp)) ! = EOF) {

ungetc((char)c, fp);

count= 0;

while((c = getc(fp) !=' ' && c != '\t')

count++;

if (!eof(fp))

ungetc((char)c, fp);

printf("%d white\_spaces\n", count);

if (!eof(fp))

ungetc((char)c, fp);

}

}

unlink

Format

#include <io.h>

STATUS unlink(filename)

char \*filename;

Description

deletes the specified filename. filename can include drive names, path names, and wildcards.

Return Value

It returns an ERROR if the specified file is missing or if the deletion fails. Otherwise, the function returns OK.

Example

#include <stdio.h>

main()

{

puts("Delete msxc\*.$$$ Ok ?");

if (getch() =='y')

/\* Delete temporary files \*/

unlink("\\tmp\\msxc\*.$$$");

}

write

Format

# include <io.h>

int write(fd, buf, bytes)

FD fd;

char \*buf;

size\_t bytes;

Description

Writes data to the file indicated by the fd. The data to be written starts with buf and the length is specified in bytes.

Return Value

The function returns 0 if the file handle was not opened or was in read mode when it was opened. Otherwise, bytes are returned as they are.

See Also

close(), creat(), open(), read()

Notes

write() is a low-level input/output function.

Example

#include <stdio.h>

char buf[100];

main()

{

char \*p, c;

FD fd;

/ \* Writing 100 characters to a file \*/

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);

}

# Chapter 7: Command Reference



## CF

Objective

Perform parsing and grammar checking (parser). It also outputs the results as an intermediate language file (.tco file).

Syntax

CF [Options] Filename

Options List

|  |  |
| --- | --- |
| Option | Features |
| -c | Nesting of comments is prohibited. |
| -e[filename] | You can specify a filename in the file name. |
| -f | Makes implicit declarations for functions and parameters. |
| -j | Correctly recognizes Kanji characters in a string. |
| -m | Display the percentage of P : S : H of the compiler work table. |
| -o[filename] | Specifies the file name of the intermediate language file. |
| -rP: S: H | Compiler Wa Assign the tapering for P : S : H ratio |
| -s | Do not interrupt the execution of the batch file, even if an error in the source file is detected |
| -t | Allows automatic type conversion between pointers and integers. |

See Also

2.2.1 CF (parser) on page 26.

## CG

Objective

Converts the intermediate language output by CF to an assembly language (code screwleter).

Syntax

CG [Optional] Filename

Options List

|  |  |
| --- | --- |
| Option |  |
| -k |  |
| -l |  |
| -o[filename] |  |
| -rN |  |
| -u |  |

See Also

2.2.3 CG (Code Generator) on page 36.

## FPC

Objective

Checks the integrity of the function arguments (parameter checker).

Syntax

FPC [Option] Filename1 Filename2 ...

Options List

|  |  |
| --- | --- |
| Option | Features |
| -c | Concatenation of .tco files. |
| -d | Specification of the don't care function. |
| -i | Warns against indirect calls. |
| -s | Does not interrupt the execution of the batch file even if an error is detected. |
| -t | int, unsigned, treats the pointer as another type. |
| -u | Do not check references to undefined functions. |

See Also

2.2.2 FPC (parameter checker) Page 32.

## MX

Objective

Assist in the creation and maintenance of the library. Split the source file into modules (by function) and output the steps to create a library file. (Libraries maintenance support tool)

Syntax

MX [Options] Filename [ModuleName 1 ModuleName 2 ...]

Options List

|  |  |
| --- | --- |
| Option | Feature |
| -l |  |
| -o[path\] |  |

See Also

4.5.2 About the MX library Maintenance Support Tool Page 93.

# Chapter 8: List of Error Messages

This chapter lists the error messages generated by the MSX-C compiler. Shows the error message (in English), its Japanese translation and its main content. You can use it if a real compiling error occurs.

The message descriptions are arranged in alphabetical order, ignoring symbols for each command.



## CF Error Messages

array of function

Creates an array of functions.

It is not possible to create an array in which the elements become functions.

bad #???

The usage of the # command is different.

The location of the # command is specified in the expression or a command that is not in the preprocessor.

bad abstract declarator

The abstract declarator is incorrect.

Abstract declarators are used in casting type specifications.

bad assignment

The assignment is wrong.

Occurs when the assignment is to an expression that contains variables and so on.

bad cast

Wrong cast.

Cast into arrays, structures and unions. The cast changes the attributes of a number or pointer and cannot be cast to an array, structure or union.

bad character

Some characters are inappropriate.

The control characters are in the source file or have '\' other than at the end of the line.

bad condition

The conditions are invalid.

Some notations do not fit the conditions used for conditional decisions, such as if, while.

bad conditional expression (missing'?')

The use of conditionals (ternary operators) is different.

Occurs when there is no '?' is not present in the conditional expression.

bad constant expression

The constant expression is incorrect.

Variables and functions are there where constants are needed, such as specifying the number of elements in an array definition.

bad #define statement

The usage of #define is different.

Contains incorrect characters as identifiers.

bad element

The statement is nonexistent or incorrect.

A statement is needed after the label, after the while, etc., but it is not found.

bad element encountered externally

The statement outside the function is invalid.

This error occurs if the statement is incorrect outside of the function. The function was in If this is the case, a "bad element'' error is raised.

bad #ifdef statement

#ifdef usage is different.

Contains incorrect characters as identifiers.

bad #ifndef statement

The usage of #ifndef is different.

Contains incorrect characters as identifiers.

bad #include statement

The usage of #include is different. The file must be enclosed in > or ".

bad indirection

The indirect specification is wrong.

This error is raised when too many indirect "\*" are attached to the pointer. That is, it happens when there are two or more normal pointers.

bad number

Characters outside the numerical range are used.

The numbers can be in octal, decimal or hexadecimal, 0 to 7, 0 to 9, 0 to 9 and a to f range.

bad option: <letter>

A non-existent switch "-<character>" is specified. The switches are cef t jmors characters.

bad parameter list

The argument list is wrong.

In the definition of an argument-filled macro, the names of the arguments are invalid or the arguments are not separated by ",".

bad parameter type

The argument type is not numeric.

When calling a function, the arguments are called in arrays, structures, and unions. Since it cannot be passed directly, it is usually substituted with a pointer to the top of what is to be passed.

bad parameter type

Type is not numeric.

When defining a function, the parameters are declared as function, structure, or union. Since you can't receive it directly, you can usually substitute a pointer to the top of what is being passed.

bad #pragma statement

The usage of #pragma is different.

The parameter is not a reserved word.

bad storage class

Memory class specification is wrong.

Extern is used in function definitions, and static is used in function definitions.

bad switch expression

The expression used in switch is invalid.

The switch expression requires a numeric result.

bad table ratio

The ratio specified by the "-r" switch is not good.

The ratio should be a positive number.

bad type in cast

The type to cast is incorrect.

The types to be cast are those that cannot be cast (e.g. functions).

bad #undef statement

The usage of #undef is different.

Contains incorrect characters as identifiers.

'break' outside switch/break

There is a break outside of switch.

Using break outside of loops, switches, and other blocks.

cannot be initialized

Cannot initialize.

The array of automatic (auto) variables cannot be initialized.

cannot make [<file>]

<file > cannot be created.

The message is displayed when the .tco file or .dia file cannot be created, such as when the directory or disk space is full or the directory does not exist.

cannot open [<file>]

<file> is not found.

Displayed when the specified C source file or include file is not found, and It's a good idea to have a good idea of what to do. The include file sets the default directory with the environment variable INCLUDE.

'case' after default

There is a case after default.

In MSX-C it is not possible to put the case after default. case can only be used inside the switch.

conflict definition '<identifier>'

The declaration of <identifier> does not match.

The declarations of two or more external and forward references do not match the declarations of the numbers and variables. There may be a difference between the declaration and the definition of a function adfow reference.

conflicting number of macro parameter

The number of macro arguments do not match.

In an argued macro, the number of arguments in the definition and in the reference must be the same.

'continue' outside loop

There is a continue outside the loop, such as while, for, etc. You cannot use continue outside the loop.

'default' appeard twice

default appeared more than 2 times in the procs of switch.

default is a label that indicates something else, so it can only be used once.

'default' outside switch

The default is outside of switch.

Default can only be used as a label within the switch's proc.

disk full

The disk is full.

The disk is full while writing data. The process cannot continue.

'double' not supported

Double is not supported

duplicate 'case' label

There are two or more cases in switch that have the same constant value of case. All constants after the case must be different values.

duplicate label '<lable>'

There are two or more labels with the same name.

The label name cannot be the same in the function.

duplicate member

The same member names in a structure or a union.

The same member names cannot be used in a structure or a union.

duplicate storage class

Two or more memory classes are specified.

Two or more words used to specify the memory classes, such as extern and static.

duplicate tag '<tag>'

The same <tag> define structures and unions.

Tag names must be given to each structure or compound by different names.

duplicate type specifier

There are two or more variable types specified.

There are two or more words specifying the type, such as int and char.

'(' expected

There is no "(".

It is raised when a syntactic requirement for "(" is present, such as before a conditional expression in an if statement.

')' expected

")" is missing.

Occurs when a syntactic ")" is necessary, such as after a conditional expression in an if statement. It also occurs when there is no ")" in the abstract declarator.

':' expected

There is no ":".

This message is displayed if there is no ":" after a constant in a case statement or after a default.

';' expected

There is no ";".

A sentence must end with ";". The next statement (a control statement) comes in spite of the absence of ";".

']'expected

"]" is missing.

There is no "]" after the number of elements in the array definition.

'}'expected

"}" is missing.

In the case of array initialization, "}" must be placed to correspond to '''}'''.

'Float' not supported

float is not supported

function cannot appear

The function cannot be specified.

It occurs, for example, when the function is used as a structure member.

function expected

A function definition is required.

It occurs when you put another string or so where you should define the function.

function returns structured data

The return value of the function is declared in a structure or a union. The function cannot return a structure or a union.

hash table over flow

Hash table overflowed.

Lack of space to store the hash value of the identifier. Use the "-r" switch to correct the ratio of the worktable. 2.2.1 See the "-r" switch section "CF (Parser)".

Heap over flow

The heap has overflowed.

It occurs when there are too many file inclusions or when too many auto variables are declared. There are also cases where the formulae are complex.

improper function mode

Inappropriate function mode.

nonrec, recursive is used for the variables.

#include too nested

#The nesting of the include is too deep.

In MS X-C, include files can be nested up to fourfold. If you are using more than that, split the include file.

known dimension expected

It does not define the array size.

The array size must be specified as a constant.

'long' not supported

long is not supported

l-value required

The left-hand side value is required.

The expression that represents the storage location is called the left-hand side value. Assignment requires an expression that indicates a storage location, such as the name of a variable.

missing ')'

")" is missing.

In the expression, the ")" is missing.

missing ':'

There is no ":".

Conditional Expression (ternary operator) "?"There is no ":".

missing ']'

There is no "]".

In the expression, the "]" " is missing.

missing '}'

There is no "}".

In the formula, the missing "}".

missing condition

There are no requirements.

There are no conditions to use for conditional decisions.

missing identifier

There is no identifier.

An identifier is a variable name, function name, array name, etc. It contains a statement or expression without an identifier.

Missing member name

There is no member name.

There are no structured variables or member names following the pointer to the structured variables.

missing operator (or semicolon)

There is no operator or ":".

It may be followed by more than one space-separated number or variable name.

missing quote

The string does not end with a (").

Occurs when the string spans multiple lines and there is no ''\'' at the end of the line.

missing tag

There is no tag name.

It occurs when there is no structure variable definition and no tag name is specified.

not appear in parameter list '<argument>'

<argument> is not in the parameter list.

Arguments that are not in the argument list are included in the argument declarations.

parameter cannot appear here

Parameters cannot be specified.

Occurs if the function's declaration specifies a parameter.

pointer type mismatch -use cast

The type of the pointer does not match. Use a cast.

In MSX-C, pointer interconversion must be cast.

pool over flow

The pool has overflowed.

The pool for storing identifiers (defined in #define or variable names) is missing. Use -r switch to correct the work-table ratio. 2.2.1 CF (Parser) -r switch.

precedence error

A priority error.

Some incorrect priorities have been encountered. Clarify your priorities using '(', ')'.

redeclaration of '<identifier>'

The "<identifier>" is double defined.

Variable function is double defined.

sizeof(func) not permitted

sizeof(func) is not allowed.

The size of the function is not computed.

sorry, too small memory

Due to lack of memory, it is not possible to compile.

stack over flow

The stack has overflowed.

The expressions etc. are too complex.

static function '<function>' not defined

A static function is not defined.

A static function must be defined in its source file. There is a declaration of static function and no function definition.

symbol table over flow

The symbol table has overflowed.

There are too many identifiers in the source program and not enough symbol table to store the information. Specify a higher ratio of symbol tables using the "-r" switch in the compiler. See 2.2.1 CF (Parser) and '-r' switches. It can also be avoided by eliminating unnecessary identifiers (defined by #define, structures, external variables, etc.), if any.

syntax error

The syntax is incorrect.

too many initializers

Too many constants for initialization.

During array initialization, it is raised if the number of initial values is larger than the array size.

too much parameters

There are too many parameters.

It occurs when the number of arguments in an argument-filled macro is greater than or equal to 11.

type mismatch

The type is not correct.

Occurs when assignments are made between a numeric type and a structure or a union variable, or between different structures or union variables.

undeclared function '<function>'

<function> is not declared.

In MSX-C, if you want to use a function, you need to declare or define it before you can use it.

undeclared identifier '<identifier>'

<identifier> is not declared.

You forgot to declare the variable or you misspelled the variable name.

undeclared member name '<member>'

'<member>' no member is declared.

You forgot to declare the members in the structure or union declaration, or the member names are wrong.

undeclared parameter '<parameter>'

Mo parameters are declared.

Either you forgot to declare the argument or you used the wrong parameter name.

undefined label '<label>' in function '<function >'

<label> is not defined

It occurs if the destination label name of the goto statement is not found in the function with the goto statement.

undefined struct/union

No structure/union denominator is declared.

It also occurs when the declaration of a structure or a union variable comes before the declaration of a structure or union.

unexpected eof

The file is finished in the middle.

Occurs when the file ends outside of the function and comments.

unexpected eof in this comment

The file ends in the middle of a comment.

It occurs when there are too many "/\*" comments and the corresponding "\*/" cannot be found at the end of the file.

unexpected eof inside function '<function>'

<function> ends in the middle of the file.

Occurs when there are too many of them and the end of the file does not find "}" to mark the end of the function.

useless expression

The result of the operation is not used.

Occurs when the first or operating the statement is neither an assignment nor an increment/decrement operator.

'while' expected

There is no while.

The do while while is not found after the loop block.

## CG Error Messages

bad option: <character>

A non-existent switch "<character>" is specified.

The available switches are the letters of rkulo.

cannot make: <file>

<file> cannot be created.

This is displayed when you can't create a mac file because the directory or disk space is full, or the directory does not exist.

cannot open: <file>.

<file> is not found.

It is displayed when the specified tco file is not found.

function overflow in "<function>". ... specify less than <#> by -r

The code generation area has been removed from function's The number must be less than "-r" switch.

In CG, the free space is divided into a symbol tapering and a code generation area. Use the "-r" switch to reduce the size of the symbol table and increase the size of the area for code generation, and then start the CG again. If, no matter what value you set for the size of the symbol table, "symbol table overflow ..." appears alternately, split the function or split the file

illegal .tco file at "<function>" <line>: <digit>

.tco file is incorrect.

.tco file is broken. Again, start with CF.

sorry, too small memory

Due to lack of memory, it is not possible to compile.

stack overflow

The stack is missing.

symbol table overflow in "<>" ...specify more than <> by -r

The symbol table ran out of symbols in the '-r.

In CG, the free area is divided into a symbol table and a code generation area. Use the 'r' switch to increase the symbol tapering and start the CG again. If "function overflow" occurs alternately, no matter what value you set for the size of the symbol table, split the function or split the file.

warning: >> more than 8 bit in "<function>"

Warning. It is shifted to the right by more than 8 bits.

If you shift the 8-bit value more than 8 bits to the right, it will inevitably be set to 0.

warning: << more than 8 bit in "<function>"

Warning. It is shifted left by more than 8 bits.

Shifting the 8-bit value more than 8 bits to the left will always result in 0.

warning: << more than 16 bit in "<function>"

Warning. Shifted left by more than 16 bits.

If you shift the 16-bit value more than 16 bits to the left, it must be set to 0.

warning: too big char constant <number> in "<function>"

Warning. The value of the character constant is too large. The type char should be set in the range of -255 to 255.

## FPC Error Messages

bad option: -<character>

A non-existent switch "-character>" is specified.

The switch that can be specified is each character in istucd.

cannot make: [<File>]

<file> cannot be created.

This occurs when concatenating tco files with the "-c" switch. The message is displayed when the directory or disk space is full, the directory does not exist, or the file cannot be created.

cannot open: [<file>]

<file> is not found.

It is displayed when the specified .tco file is not found.

in <<file>> "<function1>" calls "<function2>": conflicting number of arguments

The number of arguments does not match.

<File> The number of arguments of <function 2> called by <function 1> in <File> does not match the definition of <function 2>.

in <<file>> "<function 1>" calls "<function 2>": conflicting return type

The type of the function return value does not match.

The return type of a <function 2> called by <function 1> in <File> is the definition of a <function 2>.

in <<file>>"<function 1 >" calls "<function 2>": <Nth> argument conflict

The type of the N-th argument does not match.

The argument type of <function 2> invoked by <function 1> in <File> does not match the definition of <function 2>. For the first, second and third arguments, "<Nth>" means "1st", "2nd" and "3rd", respectively.

in <<file>> "<function l>" calls "<function 2>": undefined

The function is not defined.

The <function 2> called by <function 1> in <file> is not defined.

in <<File>> "<function>" ... sorry can't check indirect call

Indirectly called functions are not argument-checked.

FPC does not allow argument checking for indirectly called functions.

in <<File>> "<function>" was multiple defined

<function> is defined in duplicate.

<Function> is double defined in the file.

missing func list

There is no list of functions.

If the "-d" switch is specified, the next argument must be a comma-separated list of function names.

missing outputfile

No file is missing.

If the "-c" switch is specified, a filename is required to store the tco file concatenated with the next argument.

too many files

There are too many files.

There are too many tco files to check.

## MX Error Messages

bad option: - <character>

A non-existent switch "-<character>" is specified. The switch can be specified for each letter of lou

cannot make: [<File>]

<file> cannot be created.

Can't create a segmented file. The message is displayed when the directory or disk space is full, the directory does not exist, or the file cannot be created.

cannot be open: any <file> .mac or any <file> .tco

Can't find the file to split up in either the mac file or the tco file.

illegal .tco file

.tco file is not correct.

.tco file is broken. Start with CF again.

missing ENDMODULE

'ENDMODULE' is not found.

In the assembler file (.mac file), "MODULE", the start of the module, was there, but "ENDMODULE" was not found and the file ended.

out of memory

I ran out of memory.

There is no space to store the segmented module names. Split the file before splitting, and start again from CF.

Skeleton file AREL.BAT not found

The skeleton file arel.bat was not found.

To split the .mac file, place it in the current directory or in the directory containing the MX command. The file arel.bat describing the assembly procedure is required.

Skeleton file CREL.BAT not found

The skeleton file crel.bat was not found.

If you want to split the .tco file you can use the current directory or a directory with an MX command. You need the file crel.bat, which describes the procedure for code generation and assembly, in the directory.

# Chapter 9: Standard Library Functions

Here is a brief list of standard library functions. See Chapter 6, Standard Library Function Reference for details on each of the library functions.



## File Input/Output Functions

### High-level input/output functions

|  |  |  |
| --- | --- | --- |
| Function Name | Usage | Reference Page |
| clearerr | Clearing the error state |  |
| fclose | File Close |  |
| fcloseall | Close all files except standard input and output |  |
| feof | End-op file determination of the file |  |
| ferror | Did the file made a writing error |  |
| fflush | File buffers flush |  |
| fgets | Loading strings from a file |  |
| fileno | Returns the file handle of the file |  |
| flushall | Flushing of all file buffers |  |
| fopen | File opening |  |
| fprintf | Formatted data output to file |  |
| fputs | Output the string to a file |  |
| fread | Reading data from a file |  |
| fscanf | Old-fashioned data entry from files |  |
| fsetbin | Putting Input and Output in Binary Mode |  |
| fsettext | Putting input and output in text mode |  |
| fwrite | Writing data to a file |  |
| getc | Read 1 character from a file |  |
| getchar | Read one character from standard input |  |
| gets | Reading strings from standard input |  |
| printf | Data output with inner expression to standard output |  |
| putc | writes one character to file |  |
| putchar | Write out one character to standard output |  |
| puts | Output strings to standard output |  |
| scanf | Formatted data entry from standard input |  |
| setbuf | Setting up a control method for buffers |  |
| setvbuf | Setting up a control method for the buffer |  |
| ungetc | one character back to file |  |
| ungetch | Returning one character to standard input |  |

### Low-level Input/Output Functions

|  |  |  |
| --- | --- | --- |
| Function name | Usage | Reference Page |
| close | File Close |  |
| creat | File Creation |  |
| eof | End-of file determination of the file |  |
| isatty | Determining if a file is a device |  |
| open | File opening |  |
| read | File Loading |  |
| write | File Export |  |

## Strings and Character Processing Functions

|  |  |  |
| --- | --- | --- |
| Function Name | Usage | Reference Page |
| atoi | Convert a string to an integer |  |
| isalnum | Determining if the letters are alphanumeric or numeric |  |
| isalpha | Determining whether the letters are alphabetical or not |  |
| iscntrl | Determining whether a character is a control character |  |
| isdigit | Determining whether a character is a number or not |  |
| iskanji | Determining if a character is the first byte of a kanji |  |
| iskanji2 | Determining if a character is the second byte of a kanji |  |
| islower | Determining whether a character is lowercase or not |  |
| isspace | Determining whether a character is a blank character or not |  |
| isupper | Determining whether a character is uppercase or not |  |
| isxdigit | Determining if a character is a hexadecimal number |  |
| sprintf | Make output with an expression of data for a string |  |
| sscanf | Make a formatted input of data from a string |  |
| strcat | Concatenating two strings |  |
| strchr | Finding characters from a string |  |
| strcmp | Comparing two strings |  |
| strcpy | Copying a string |  |
| strlen | Return the length of the string |  |
| strlwr | Making uppercase letters in strings to lowercase |  |
| strncat | Concatenate two strings with an upper limit |  |
| strncmp | Compare two strings with an upper limit |  |
| strncpy | Copying a string with an upper limit |  |
| strupr | Capitalize lowercase letters in strings |  |
| tolower | Converting uppercase letters to lowercase |  |
| toupper | Convert lowercase letters to uppercase |  |

## Memory Management Functions

### Low-Level Memory Management Functions

|  |  |  |
| --- | --- | --- |
| rsvstk | Secure the stacked area |  |
| sbrk | Allocate memory procs |  |

### High-Level Memory Management Functions

|  |  |  |
| --- | --- | --- |
| alloc | Allocate memory procs |  |
| free | Free the allocated memory in alloc |  |

## Directory Functions

|  |  |  |
| --- | --- | --- |
| chdir | Changing the current directory |  |
| expargs | Expand the file name containing the wildcard |  |
| getcwd | Win the current directory |  |
| mkdir | Creating a supplements directory |  |
| rmdir | Removing the supplication directory |  |

## Program Execution Functions

|  |  |  |
| --- | --- | --- |
| execl | Program chain (1) |  |
| execlp | Program chain (2) |  |
| execv | Program chain (3) |  |
| execvp | Program chain (4) |  |
| exit | End of program execution |  |
| \_exit | End of program execution |  |

## Keyboard, I/O Functions

|  |  |  |
| --- | --- | --- |
| getch | Entering one character from the keyboard |  |
| getche | Entering one character from the keyboard |  |
| inp | Input data from I/O port |  |
| kbhit | Testing whether there is keystrokes |  |
| outp | I/O output data to port |  |
| sensebrk | Test for input Ctrl+C, Ctrl+STOP |  |

## Machine Language, MSX-DOS Functions

|  |  |  |
| --- | --- | --- |
| bdos | Execution of a function call (returning the value of a) |  |
| bdosh | Execution of the function call (return the value of hl) |  |
| bios | Running a BIOS call |  |
| call | Calling the machine language program (returns the value of hl) |  |
| calla | Calling a machine language program (returns the value of a) |  |
| callxx | Calling the machine language program (get results) |  |

## Memory Operation Functions

|  |  |  |
| --- | --- | --- |
| memcpy | Memory block transfer |  |
| memset | Transfer of memory block to |  |
| movmem | Write values to memory block |  |
| setmem | Harming the value to memory procs |  |

## Generic Functions

|  |  |  |
| --- | --- | --- |
| abs | Return absolute value |  |
| getenv | Get the value of the environment variables |  |
| longjmp | Direct jump between functions |  |
| max | Returns the larger of the two arguments |  |
| min | Return the smaller of the two arguments |  |
| putenv | Set the value of the environment variables |  |
| qsort | Execution of Sorting |  |
| rename | File renaming |  |
| setjmp | Direct jump between functions |  |
| unlink | Removal of files |  |

# Appendix A Sample Program



## About "Q.com"

The sample program q.com, included on the system disk, is a program that draws a figure with one line changing color arbitrarily and the other line obliterating it. Automatic screen mode is set, for MSX (screen mode 2), or for MSX2 (screen mode 8)

On MSX-DOS

A>q ◄┘

to execute.

Press any key while it is running to return to MSX-DOS. When operating, if you have pressed Ctrl+STOP please reset it.

q.com compiles and assembles q.c. and q.rel, the library line.rel for drawing lines in a specified color, library rnd.rel for returning random integers, library calbas.rel for using routines in BASIC's ROM, and the standard. It can be generated by linking the library with the library using a batch file called mkq.bat.

## Utilizing Functions from q.com

The functions used in q.com such as line(), rnd() and so on can be used in other programs, but they require the following tasks.

### Declaration of the Functions

In each program that uses a function, you must make the following declarations (These are It is not declared in stdio.h).

VOID line();

VOID chgmod();

VOID totext();

VOID els();

TINY \*color();

unsigned rnd();

### Linking

When linking, calbas.rel must be linked to use a function other than rnd(). In addition, to use line(), color(), you must link line.rel, color.rel before calbas.rel.

The following is an example of a link without rnd(), color()

l80 ck,progm,line,calbas,clib/s,crun/s,cend,progm/n/y/e:xmain

### Descriptions of q.com Functions

Describes the specifications of the functions used in q.com. See if you want to use these functions in other programs.

<line>

VOID line(x1. y1, x2, y2, color, log\_op)

int x1, y1, x2, y2;

TINY color, log\_op;

Draws a line connecting the coordinates (x1, y1) and the coordinates (x2, y2) in the color specified by color. The same logical operations as in BASIC are available for MSX2, but not for MSX1. In the case of MSX1, don't use the operation, Iog\_op can be filled with (TINY)0 or other dummy value.

The range in which the coordinates can be specified depends on the screen mode. The screen mode can be changed with chgmod(), which is described next.

line(x, y, x1, y1, c, (TINY)0);

<chgmod>

VOID chgmod(mod)

TINY mod;

Changes the screen mode to the mode specified by mod, which can be an integer from 0 to 8 (0 to 3 for MSX1), otherwise it does not work at all.

For screen mode, see the screen statement description in the BASIC manual.

<totext>

VOID totext()

Restores screen mode to the original (last) text mode. If you run it in text mode, it won't do anything. There are no arguments or return values.

<cls>

VOID cls()

Clear the screen.

<color>

TINY \*color(for, back, bord)

TINY fore, back, bord;

Specifies the colors of the foreground, background, and border.

The return value is a pointer to the 3-byte area that contains the last setting.

TINY \*prev; << previous color

chgmod(8);

prev = color(255, 0, 0); << Screen 8 with white foreground, back, and black border, previous color

.

.

color(prev[0], prev[l], prev[2]; << Restore the setting

<rnd>

unsigned rnd (range)

unsigned range;

Returns a random integer from 0 to (range - 1). There are no arguments or return values. The function is defined in rnd.mac.

### Introduction to Reference Books

Currently, the following books on MSX-DOS and the C language are available for writing programs like q.com.

MSX Technical Data Book 1 Edited by ASCII, Microsoft FE Headquarters

MSX Technical Data Book 2 Edited by ASCII, Microsoft

MSX2 Technical Hand Book Edited by ASCII, Microsoft FE

V9938 Technical Data Book Edited by ASCII, Microsoft FE Headquarters / Nippon Gakki Manufacturing Co.

Introduction to the C Language Co-authored by Les Hancock-Morris Krieger, Supervised translation by ASCII Press

Introductory C Norihiro Mita

Language Practice C Norihiro Mita

Language Application C Norihiro Mita

Programming Language C ASCII Corporation, B.W. Carnihan by D.M. Ritchie Publishing Co.

# Appendix B MSX-C Ver.1.2 Master Disk Content

The master disk contains the following files. Please check.

|  |  |
| --- | --- |
| \ | Included command and library |
| cf.com | Compiler (Parser) |
| cg.com | Compiler (Code Generator) |
| fpc.com | Function Parameter Checker |
| mx.com | Module Extraction |
| ck.rel | Utility Kernel Calling Routines |
| clib.rel | Standard library (including the kernel body) |
| crun.rel | Run-time routines |
| cend.rel | Standard library |
| lib.tco | Standard library for FPC.tco |
| c.bat | Batch file for file compiling |
| readme.doc | It contains the latest information, if any. |

|  |  |
| --- | --- |
| \include | Included header files |
| bdosfunc.h | MSX-Header for DOS function calls |
| conio.h | Header for I/O functions |
| ctype.h | Header for character types declaration |
| direct.h | Header for directory manipulation functions |
| io.h | Header for low-level input/output functions |
| malloc.h | Header for memory management functions |
| memory.h | Header for memory manipulation functions |
| process.h | Header for program execution functions |
| setjmp.h | Header for setjmp |
| stdio.h | Headers for high-level input/output functions |
| stdlib.h | Headers for generic functions |
| string.h | Header for string manipulation functions |
| type.h | Header for the definition of MSX-C types |

|  |  |
| --- | --- |
| \batch | Included batch files for the re-creation of the library and the creation of the development environment |
| arel.bat | Batch file for MX (for assembly language) |
| crel.bat | Batch file for MX (for C language) |
| cenv.bat | Batch file to set the environment variables for MSX-C |
| forremk.bat | Batch file for creating this disk |
| genlib.bat | Batch files for starting a standard library re-creation |
| genliba.bat | Batch file for re-creating the Assembly language libraries |
| genlibc.bat | Batch file for re-creating the C language libraries |
| genrel.bat | Batch file for re-creating the .rel libraries |
| gentco.bat | Batch file for re-creating the .tco libraries |
|  |  |
| mksys.bat | Batch file for creating MSX-C disk |

|  |  |
| --- | --- |
| \src | Included standard library source files |
| clibc.c | Creating skeletons in clibmac.mac |
| direct.c | Directory manipulation functions |
| io.c | Low-level input/output, keyboard functions |
| malloc.c | Memory management functions |
| process.c | Program execution functions |
| stdio.c | High-level input/output functions |
| stdlib.c | General-purpose functions |
| string.c | String manipulation functions |
|  |  |
| ck.mac | Kernel calling routine |
| clibmac.mac | Standard library function assembler description |
| crun.mac | Run-time routines |
| cend.mac | Standard library function assembler description Part 2 |

|  |  |
| --- | --- |
| \sample | Included MSX-C samples |
| head.c | Filter command to display the top of the file |
| wc.c | Display the number of lines, words and bytes in the file |
|  |  |
| q.com | Graphic display |
| q.c | q.com C source |
| line.c | line() function source |
| line.rel | " function object |
| color.c | color() function source |
| color.rel | " function object |
| calbio.mac | calbio() function source |
| calbio.rel | " function object |
| rnd.mac | rnd() function source |
| rnd.rel | " function object |
| mkq.bat | Batch file for creating q.com |
|  |  |
| search.c | Example of using Disk-Basic environment |
| search.bas | Disk-Basic side of the program |
| search.bat | Batch file from compilation to execution |
| bk.mac |  |
|  |  |
| rom0.mac | A sample of the ROM conversion program |
| roml.c |  |
|  |  |
| More than 64 files + readme.doc |  |

# About Us

We pack our products strictly and ship them with the utmost care. In case of transportation problems, please let us know and we will replace it with a new one.

When creating the manual, we have tried to explain it in as much detail as possible, but if it is still not clear, you can check the part with the computer until you are satisfied. It is also possible to refer to other pages. If you still cannot solve your question, please contact the dealer where you bought it or call ASCII User Support (Hotline 03-498-0299) and they will be able to help you. However, please write to us if you can, as there may be congestion on the 1-hotline, which may cause inconvenience. Please follow the instructions below. We may not be able to respond to you if there is even one item that is not filled out. Be very careful.

If you have any comments or requests for other products, please contact us.

Notes

1. Send to 3F Minami-Aoyama Building, 6-11-1 Minami-Aoyama, Minato-ku, Tokyo 107-24, Japan

ASCII Corporation User Support

TEL 03-498-0299

(Monday-Friday, except for national holidays, 10: 00~12: 00, 13: 00~17: 00)

1. Required Items
   1. Your name, address (zip code), and phone number (including area code)
   2. Product name, product number
   3. Equipment configuration
      1. Model name, memory in bytes
      2. CRT device name, floppy disk mounting name, printer mounting name
      3. Others I/O, I/F name
   4. Inquiry content

Please describe the content of your inquiry concretely and clearly using the terms described in the product manual as much as possible. For an issue that seems to be a bug, information that can reproduce the issue is required. If we cannot reproduce it, we cannot investigate it. Please be sure to attach the operation procedure and data until the issue occurs. If you have a data disk, please enclose a copy of it to make Chomori squishy.

In addition, we cannot respond to inquiries regarding the design, creation, operation, and maintenance of applications that are considered to be unique to the customer, as they are outside the scope of our support. Thank you for your understanding.

**MSX-C Ver.1.2 USER'S MANUAL**

April 1, 1989, 1st edition, 1st print issued

Supervision/Editing ASCII Co., Ltd. System Equipment Division Publishing Office ASCII Co., Ltd.

107-24 6-11-1, Minami Aoyama, Minato-ku, Tokyo 3F Minami Aoyama Building

Transfer Tokyo 4-16 1144

TEL. (03) 486-7111

ASCII SCORCPORAITIOIN

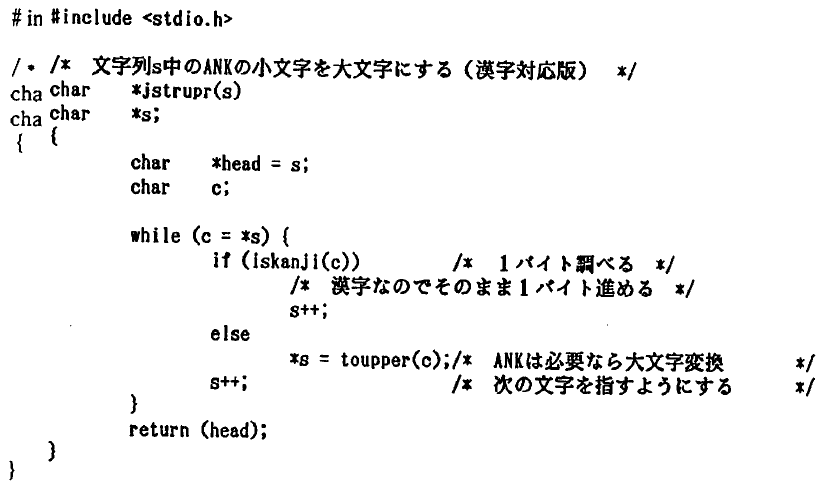
MSX-C Ver.1.2 User's Manual Errata

April 14, 1989

|  |  |  |  |
| --- | --- | --- | --- |
| Page | Row | Error | Correction |
| P.3 | 20 | bdosfunc.h 80 | bdosfunc.h 79 |
| P.10 | 18 | Chapter 10 Sample Program q.com | Appendix A Sample Program q.com |
| P.10 | 20 | Chapter 11 MSX-C Ver.1.2 Master Disk  Contents | Appendix B MSX-C Ver.1.2 Master Disk  Contents |
| P.29 | Second line from the bottom | symbol table <Number of bytes used >... | symbol table <Number of bytes used>... |
| P.35 | 8 | ...sorry can"t check ... | ...sorry can't check ... |
| P.37 | 2 | Outputs an intermediate language file | Output assembler file |
| P.69 | Second line from the bottom | \CHKCHR(5DH) | \_CHKCHR(5DH) |
| P.107 | 3 | Figure 6-1 | Figure 5-1 |
| P.107 | Bottom line | 100 CLEAR,&HCFFF | 100 CLEAR 300,&HCFFF |
| P.108 | 2 | (New HIMEM--1) | (New HIMEM-1) |
| P.108 | 5 | Figure 6-1 | Figure 5-1 |
| P.108 | 6th line from the bottom | 100 CLEAR,&HCFFF | 100 CLEAR 300,&HCFFF |

### Error

P.70



### Correction

#include <stdio.h>

/\* Capitalize the small letters of ANK in the character string s (Kanji compatible version) \*/

char \*jstrupr(s)

char \*s;

{

char \*head = s;

char c;

while (c = \*s) {

if (iskanji(c)) { /\* 1-byte adjustment \*/

/\* Kanji, so proceed one byte as it is \*/

s++;

else

\*s = toupper(c); /\* ANK should be converted to uppercase if necessary \*/

s++; /\* point to the next character \*/

}

return (head);

}

}