

FUSION-C

S D K

C PROGRAMMING FOR MSX-DOS

by Eric Boez & Fernando Garcia

Complete Journey for FUSION-C v1.3

(FUSION-C v1.0, v1.1, 1.2, v1.3)

**EBSsoft
Edition**

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

EBSsoft Edition

Book written by Eric Boez
Copyright © 2018-2019-2020
All rights reserved.

ISBN: 9781730828614

Book revision – 2020 November, 15

SDK revision : R11511

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without Authorization
is copyright violation

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation
« *C is Future of MSX* »
Jorge Torres Chacón

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation



FUSION-C is free, it took many hours of work to create this library. If you like FUSION-C, and you want to change features, to add more, share your work with the community. You can, for example, send us your code, your new functions, your ideas. They will be included in future version of the library, for the benefit of everyone.

**FUSION-C is free software; it comes without any warranty.
You can use, modify, share it under the terms of
Creative Commons CC BY_SA 4.0 license**



What means this licence ?

Share : You can copy and redistribute the material in any medium or format
Adapt : You can remix, transform, and build upon the material for any purpose, even commercially.

Your obligations for using this library inside your own software:

You must, quote the original name of the library, the authors, and provide a download link to this original software.

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:



Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

You must provide **the name of the creator and attribution parties, the title of the material**, a copyright notice, a license notice, a disclaimer notice, and a link to the material



ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

Notices:

You do not have to comply with the license for elements of the material in the public domain or where your use is permitted by an applicable exception or limitation.

No warranties are given. The license may not give you all of the permission necessary for your intended use. For example, other rights such as publicity, privacy, or moral rights may limit how you use the material.

C Library for MSX-DOS with SDCC compiler
Fusion-C makes use or is distributed with external tools.

AYFX Editor by Shiru.

<https://shiru.untergrund.net/software.shtml>

Source code and updates:

<https://github.com/Threetwosevensixseven/ayfxedit-improved>

Bin2froh by Ming Chen

Source code and updates:

<https://sourceforge.net/projects/bin2froh>

MSX DiskImage by <unknown author>

Disk-Manager by Lex Lechz

<http://www.lexlechz.at>

Vortex Tracker II by Sergey Bulba

DSKTOOL v1.3 by Ricardo Bittencourt, Tony Cruise, Natalia Pujol

Source code and updates:

https://github.com/nataliapc/MSX_devs

Hex2Bin 2.5 by Jacques Pelleter

<https://sourceforge.net/projects/hex2bin/>

Disk2Rom 0.8 by Vincent van Dam

Sprite SX by 303bcm

OpenMSX v0.15.0 by openMSX Team

Source code and updates:

<https://openmsx.org>

Sc2GraphXConvertor v0.5 by Eric Boez & Leandro Xorreia

RLEWbCompressor 1.0b by Eric Boez & Aorante

BitBuster v1.2 by Team Bomba

Image to Sprite Editor by Sylvain Cregut

Sprite Path Editor by Sylvain Cregut

MSX-DOS v1.03 by Microsoft

MSX-DOS v2.30 by ASCII

Other contributions

Illustrated pictures by Eric Boez

Humorous illustrations by Carali

For their help or contribution

I want to thank:

Sylvain Cregut

Mvac7/303bcn

Nestor Soriano

T.Hara

Jorge Torres Chacón

GDX

Grauw

Javi Lavandeira

PingPong

Pasi Kettunen

The openMSX dev crew

Dolphin_Soft

Oduvaldo Pavan Junior



PRIVATE DOCUMENT
DO NOT SHARE
Sharing This Document without Authorization
is a copyright violation

EBSsoft Edition



SDCC (Small Device C Compiler) is free open source software, licensed under the GPLv2

openMSX is free software, licensed under the GPLv2

MSX, **MSX-DOS**, **MSX-BIOS**, registered trademarks owned by the **MSX Licensing Corporation**
represented by **Kazuhiko Nishi**.

MSX Basic, registered trademarks and source code owned by **Microsoft Corporation**.

MSX **MSX2** **MSX2+** **MSX turbo R** **V9990** **GR8NET**

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler

What is « FUSION-C » ?.....	22
What's new in version 1.3.....	24
Installing the Tools Chain - SDK.....	28
Download the necessary files	30
Manual installation for Windows.....	31
Manual installation for MacOS.....	33
Know the compilation script.....	39
openMSX configuration files	40
Key Maping.....	42
Know the MSX Floppy Drive and Hard-Drive.....	46
Start your first compilation.....	47
FAQ	48
Use Microsoft Visual Studio Code	50
Overriding default compilation directives.....	53
Example of a C program	55
Example of the working environment.....	57
Content of the FUSION-C SDK.....	58
The Library.....	58
The tools.....	58
Functions List.....	60
Note about function's description.....	66
MSX FUSION.....	68
Console Functions.....	68
CheckBreak	68
Getche	68
InputChar.....	68
InputString.....	68
Locate	68
PrintHex.....	68
PutCharHex	68
Print.....	69
PrintNumber	69
PrintFNumber	69
PrintChar	69
PrintDec	69
printf.....	70
Miscellaneous Functions.....	71
Cls	71
KeySound	71
FunctionKeys	71
ChangeCap	71
ReadMSXtype	71
ReadKeyboardType	72
Screen	72
Beep	72
RealTimer	72
SetRealTimer	72
CovoxPlayVram	72
CovoxPlayRam	73
RleWBToRam	73
RleWBToVram	73
PatternRotation	73
PatternHFlip	73
PatternVFlip	74
TurboMode	74
Joystick & mouse functions.....	75
JoystickRead	75
TriggerRead	75
JoystickReadTo	76
MouseRead	77
MouseReadTo	77
Keyboard Functions	78
GetKeyMatrix	78
Inkey.....	78

C Library for MSX-DOS with SDCC compiler

KillKeyBuffer.....	78
WaitKey	78
Rkeys	79
Fkeys.....	79
I/O Port Functions.....	80
OutPort.....	80
InPort	80
OutPorts	80
VDP Functions	81
VDPstatus.....	81
VDPstatusNi.....	81
VDPwriteNi.....	81
VDPwrite	81
IsVsync	81
IsHsync	81
Vsynch.....	81
Vpeek.....	81
Vpoke.....	82
VpokeFirst.....	82
VpokeNext	82
VpeekFirst.....	82
VpeekNext.....	82
Width	82
SetColors.....	82
SetColor	82
SetBorderColor.....	82
SetColorPalette	83
SetPalette.....	83
SetTransparent.....	83
RestorePalette.....	84
SetDisplayPage.....	84
SetActivePage.....	84
SetScrollIH	84
SetScrollIV	84
SetScrollMask.....	85
SetScrollDouble.....	85
HideDisplay.....	85
ShowDisplay	85
FillVram.....	85
PutText.....	85
VDP50Hz	85
VDP60Hz	85
VDplineSwitch	86
CopyRamToVram.....	86
CopyVramToRam.....	86
GetVramSize	86
SetVDPwrite.....	86
SetVDPread.....	86
SetExpandVDPcmd	86
VDPalternate	87
VDPinterlace	88
SetScreen10	88
SetScreen12	88
SetAdjust	89
SaveScreenBoot.....	89
Type Functions.....	90
IsAlphaNum	90
IsAlpha.....	90
IsAscii	90
IsCtrl	90
IsDigit	90
IsGraph	90
IsLower	91
IsUpper.....	91
IsPrintable	91

C Library for MSX-DOS with SDCC compiler

IsPunctuation	91
IsSpace	91
IsHexDigit	91
IsPositive	92
IntToFloat	92
IntSwap	92
String Functions	93
CharToLower	93
CharToUpper	93
StrCopy	93
NStrCopy	93
StrConcat	93
NStrConcat	93
StrLen	93
StrCompare	93
NStrCompare	94
StrChr	94
StrPosStr	94
StrSearch	94
StrPosChr	94
StrLeftTrim	94
StrRightTrim	94
StrReplaceChar	94
StrReverse	95
Itoa	95
StrToLower	95
StrToUpper	95
Memory Functions	96
Poke	96
Pokew	96
Peek	96
Peekw	96
MemChr	96
MemFill (aka FillRam)	96
Memcpy	96
MemcpyReverse	96
MemCompare	96
MMalloc	97
ReadTPA	97
ReadSP	97
BitReturn	97
BitReset	97
Interrupt Functions	98
EnableInterrupt	98
DisableInterrupt	98
Halt	98
Suspend	98
SetInterruptHandler	98
EndInterruptHandler	98
SetVDPIInterruptHandler	98
EndVDPIInterruptHandler	99
PSG Functions	100
InitPSG	100
PSGread	100
PSGwrite	100
MSX-DOS File I/O Functions	101
Predefined macros & structures	101
FcbOpen	101
FcbDelete	101
FcbCreate	102
FcbClose	102
FcbRead	102
FcbWrite	102
FcbFindFirst	102
FcbFindNext	102

C Library for MSX-DOS with SDCC compiler

SetDisk.....	102
GetDisk.....	102
Other MSX-DOS Functions.....	103
Predefined macros & Structures	103
GetDate	103
GetTime	103
SetDate.....	103
SetTime.....	103
Exit.....	104
CallBios.....	104
CallDos.....	104
CallSub.....	104
SetRamDisk.....	105
Turbo-R Functions.....	105
GetCPU.....	105
ChangeCPU.....	105
PCMPlay	105
File I/O.....	106
Predefined macros & Structures	106
DiskLoad.....	107
GetOSVersion.....	107
Open.....	107
Create.....	107
Close	107
Read.....	107
FCBlist.....	108
Write	108
Ensure	108
OpenAttrib.....	108
CreateAttrib.....	108
GetCWD.....	108
Ltell.....	108
Lseek.....	109
Remove.....	109
Rename	109
FindFirst.....	109
FindNext.....	109
MakeDir	109
RemoveDir	110
GetDiskParam.....	110
SetDiskTrAddress.....	110
GetDiskTrAddress.....	110
SectorRead	110
SectorWrite.....	110
MSX1 GRAPHICS.....	111
Predefined macros & Structures	111
SC2WriteScr.....	111
SC2ReadScr	111
Get8px.....	111
ReadBlock	111
WriteBlock	111
Get1px.....	112
Set8px	112
Set1px	112
Clear8px	112
Clear1px	112
GetCol8px	112
SetCol8px	112
SC2Point	112
SC2Pset	112
SC2Line	113
SC2Paint	113
SC2BoxFill	113
SC2BoxLine	113
MSX2 GRAPHICS.....	115

C Library for MSX-DOS with SDCC compiler	
Predefined Structures and macros	115
Logical operations	115
VMSX	116
Pset	116
Point	116
Line	116
BoxLine	116
BoxFill	116
High speed VDP Commands	117
HMMC	118
LMMC	118
LMCM5	118
LMCM8	119
YMMM	119
LMMM	119
HMMM	120
HMMV	120
LMMV	121
VDPLINE	121
fLMMC	122
fVDP	123
PAINT	126
Paint	126
SetPaintBuffer	126
SPRITES	128
SpriteOn	128
SpriteOff	128
Sprite8	128
Sprite16	128
SpriteSmall	128
SpriteDouble	128
SpriteReset	128
SpriteCollision	128
SpriteCollisionX	128
SpriteCollisionY	129
PutSprite	129
fPutSprite	129
SetSpritePattern	129
Sprite32Bytes	130
SpriteOverlap	130
SpriteOverlapId	130
SetSpriteColors	131
Pattern16RotationVram	131
Pattern8RotationVram	131
Pattern8RotationRam	132
Pattern16RotationRam	132
Pattern8FlipRam	132
Pattern16FlipRam	133
Pattern8FlipVram	133
Pattern16FlipVram	133
SpriteFollow	134
CIRCLE	136
CircleFilled	136
Circle	136
SC2CircleFilled	136
SC2Circle	136
MSX-DOS 2 RAM MAPPER	138
InitRamMapperInfo	138
Get_PN	138
Put_PN	138
AllocateSegment	138
FreeSegment	138
PSG	140
Sound	140
SetChannelA	140

C Library for MSX-DOS with SDCC compiler	
SilencePSG.....	140
GetSound.....	140
SetTonePeriod	140
SetNoisePeriod	140
SetEnvelopePeriod.....	140
SetVolume.....	140
SetChannel	140
PlayEnvelope.....	141
SoundFX	141
AYFX PLAYER.....	142
InitFX.....	142
PlayFX	142
UpdateFX.....	142
StopFX.....	142
MUSIC PT3 + AYFX REPLAYER.....	144
PT3Init	144
PT3Play	144
PT3Rout	144
PT3Mute.....	144
PT3FXInit	144
PT3FXPlay.....	144
PT3FXRout	145
FUSION-C ENVIRONMENT VARIABLES.....	146
(SpriteOn	146
(SpriteSize	146
(SpriteMag	146
(DisplayPage	146
(ActivePage	146
(VDPfreq	146
(VDPLines	146
(ForegroundColor.....	146
(BackgroundColor	146
(BoderColor	146
(ScreenMode	146
(SpritePatternAddr	147
(SpriteAttribAddr	147
(SpriteColorAddr	147
(WidthScreen0	147
(WidthScreen1	147
(FusionVer	147
(FusionRev.....	147
MSX BASIC VS Fusion-C.....	148
Jump and Loop	148
Clock and Time	148
Conditions	148
Conversions	148
Loading and Saving	148
Graphics and Screen	149
Math.....	149
Ram Access	149
Sprites	150
Strings	150
Variables settings	150
The Library's source code.....	152
The Source code catalog	154
The C standard functions	160
CTYPE.H	160
MATH.H	161
STDLIB.H	164
STRING.H.....	165
STDARG.H	166
TIME.H	167
Adding Assembler source code inside your C program.....	168

C Library for MSX-DOS with SDCC compiler	
Debugging for advanced users	174
Use command line arguments with your program	176
Fusion-C - Tools	178
Sprite Path	178
Image To Sprite Editor.....	179
Technical information about MSX & MSX2	180
MSX Models summary	182
The MSX Keyboard	184
International key matrix	184
Japanese key matrix	186
UK key matrix	186
Spanish key matrix.....	186
Russian key matrix.....	186
French key matrix	187
The ASCII table.....	190
MSX 1 video screen modes.....	192
SCREEN 0	192
SCREEN 1	192
SCREEN 2	192
SCREEN 3	193
MSX 2 video screen modes.....	194
SCREEN 0	194
SCREEN 4	195
SCREEN 5	196
SCREEN 6	197
SCREEN 7	198
SCREEN 8	199
SCREEN 10 & 11	200
SCREEN 12.....	200
The Sprites	202
The patterns	203
Sprites coordinate & priority	205
Sprites colors	206
Extended attributes of the sprite color table	206
The Attribut table	208
The MSX Cartridges and rom mapper.....	210
MSX Ram Memory Mapper	212
MSX-DOS Operating System.....	216
MSX DOS Memory map	216
Media supported by MSX-DOS.....	217
Reminder about boolean logical operators	220
Memento about C language	222
Some facts about C	226
The C program structure	227
Tokens in C	228
Semicolons	228
Identifiers	228
Keywords	229
Whitespace in C.....	229
DATA Types.....	230
Integer Types.....	231
Floating-Point Types.....	232
The void Type.....	233
The Variables	234
Variable Definition in C	235
Variable Declaration in C	236
Lvalues and Rvalues in C.....	237
Constants and literals	237
Integer Literals.....	238

C Library for MSX-DOS with SDCC compiler

Floating-point Literals	238
Character Constants	239
String Literals	240
Defining Constants	240
The #define Preprocessor	240
The const Keyword	241
The Storage Class	241
The auto Storage Class	241
The register Storage Class	242
The static Storage Class	242
The extern Storage Class	242
Operators	244
Arithmetic Operators	244
Relational Operators	245
Logical Operators	245
Bitwise Operators	246
Assignment Operators	247
Misc Operators – sizeof & ternary	248
Operators Precedence in C	249
Conditions and decision making in C	250
The ?: Operator	250
Loops	251
Loop Control Statements	252
The Infinite Loop	252
Functions	253
Defining a Function	253
Function Declarations	253
Calling a Function	255
Function Arguments	256
Scope range	257
Local Variables	257
Global Variables	258
Formal Parameters	259
Initializing Local and Global Variables	260
Arrays	260
Declaring Arrays	260
Initializing Arrays	261
Accessing Array Elements	261
Arrays in Detail	261
The pointers	263
What are Pointers	263
How to Use Pointers	264
NULL Pointers	264
Pointers in Detail	265
Strings	266
Structures	268
Accessing Structure Members	269
Structures as Function Arguments	270
Pointers to Structures	271
Bit Fields in structures	271
Union	272
Defining a Union	272
Accessing Union Members	273
Bit Field	274
Bit Field Declaration	275
Typedef	276
Input and Output	277
The Standard Files	278
The getchar() and putchar() Functions	278
The gets() and puts() Functions	279
The scanf() and printf() Functions	279
The Preprocessor	280
Preprocessors Examples	281
Predefined Macros	282
Preprocessor Operators	283

C Library for MSX-DOS with SDCC compiler	
The Stringize (#) Operator.....	283
The Token Pasting (##) Operator.....	283
The Defined() Operator.....	283
Parameterized Macros.....	284
The Header file	285
Include Syntax	285
Include Operation	286
Once-Only Headers.....	286
Computed Includes	287
Type Casting.....	288
Integer Promotion	288
Usual Arithmetic Conversion	289
Error Handling.....	289
errno, perror(). and strerror()	290
Divide by Zero Errors	291
Program Exit Status	291
Recursion	292
Number Factorial.....	292
Fibonacci Series.....	293
Variable Arguments.....	294
Memory Management	296
Allocating Memory Dynamically	297
Resizing and Releasing Memory	298
Publish and distribute your game	300
Contacts.....	301

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

What is « FUSION-C » ?

Fusion-C is a C library with which you can program software and games for MSX computers under MSX-DOS, (MSX1, MSX2, MSX2+ and MSX Turbo-R), it uses the cross C compiler SDCC (Small Device C Compiler).

Face the fact there is no real complete and documented tools for the C development on MSX, With the major help of Fernando Garcia, I decided to reshape and to complete all SDCC code for MSX that were available here and there, to something coherent and easy to use.

The name "FUSION" was chosen because this library was originally based on some other Libraries, or part of libraries found separately here and there without any homogenization. FUSION-C is also composed with new source code, new commands and routines that will make your life easier in MSX programming. One of the basis of FUSION-C comes from « Solid-C .» It was a MSX compiler and a C library for MSX created by Ego Voznessenski in 1995-1997.

In 2015 an unknown, MSX user partially port the Solid-C Library to the SDCC compiler, portions of this code are included in FUSION-C.

In 2006 T. HARA a Japanese developer who worked on the one chip MSX made some functions and routines for the SDCC compiler ; some of his routines are included in this Library. Other routines comes from the work of Néstor Soriano, Jorge Torres Chacon, MVAC7, Eric Boez, and Fernando Garcia.

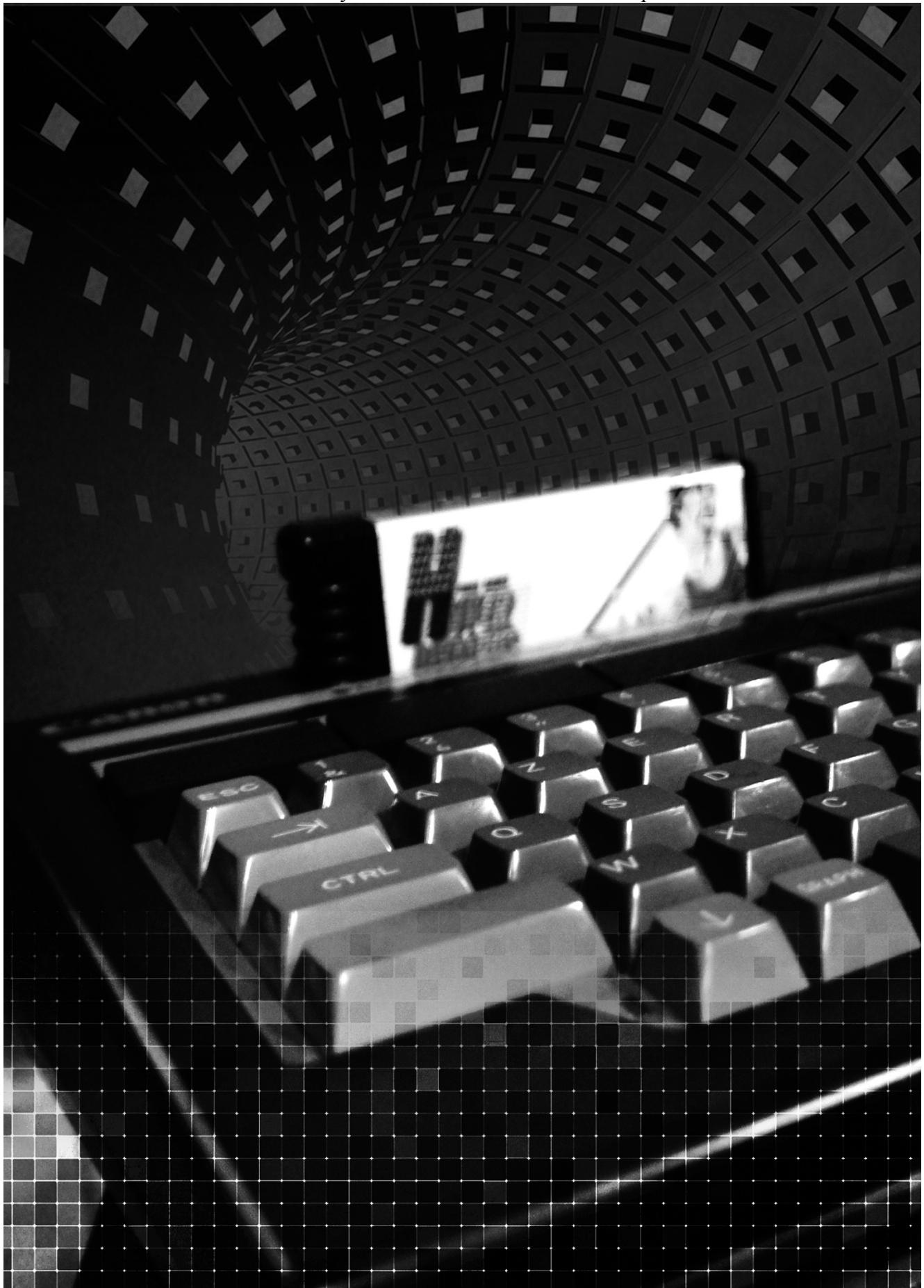
The purpose is to bring to MSX users, tools and documentation to code programs and games in C. With this aim in mind, with Fernando Garcia we completed the library with graphical functions, music and sound abilities, and the possibility to use the full content of the MSX Memory thru the MSX-DOS 2 memory mapper.

More than a year after the release of the first version of FUSION-C, the library has evolved a lot. Other actors from the MSX scene joined me to advance the library. I also reworked a lot of the original codes to improve them and add new functions.

SDCC (Small Device C Compiler) is an optimized Standard C compiler that can produce Zilog Z80 code for multiple Z80-based computers. It's a reliable tool, still in evolution and well documented. The compiler includes a standard C library partially compatible with the MSX. Actually, we recommend using the SDCC version 3.6.0

We hope you will enjoy coding for MSX computers with Fusion-C.





What's new in version 1.3

The Fusion-C's *./source/lib* has been reorganized, as most of the source codes.

A *_Source_Code_Catalog.pdf* file has been created in order to easily find where to find the source code of a function.

The "Working Folder" folder has been renamed to "WorkingFolder", the content of the *fusion-c/example* folder has been reorganized, and the examples reworked. A lot of source code were rewrote and optimized for speed performance.

MAJOR INCOMPATIBILITY

Because some functions were removed in FUSION-C 1.3 your code made for FUSION-C 1.2 or FUSION-C 1.1 may not be 100% compatible with this new release.

Please check the removed functions section.

Major problems may come from the suppression of HMCM and HMCM_SC8.

HMCM & HMCM_SC8 were replaced by LMCM5 and LMCM8. They have the same use as old ones, but the last parameter was removed because it was useless. To fix any issue, rename old HMCM and HMCM_SC8 by LMCM5 and LMCM8, then remove the last parameter of the old function (OP).

New AYFX Stand alone driver.

New PT3 + AYFX driver.

New Pattern rotation and Pattern Flip functions.

New Fast Sprite Management

New Fast Paint and Fast Pset functions

New Debug possibilities with openMSX Debugger

New Interrupt handler management

New development with MSX Hard-Drive support

- Renamed "Working Folder" to "WorkingFolder"

- Introducing Fusion-C Environment variables

- SpriteOn
- SpriteSize
- SpriteMag
- ActivePage
- DisplayPage
- VDPFreq
- VDPLines
- ForegroundColor
- BackgroundColor
- BorderColor
- ScreenMode
- SpritePatternAddr
- SpriteAttribAddr
- SpriteColorAddr
- WidthScreen0
- WidthScreen1
- FusionVer
- FusionRev

(msx_fusion.h)

- added functions to the library and to manual/book:

SC2BoxLine (Replacing Rect)	(vdp_graph1.h)
SC2BoxFill	(vdp_graph1.h)
BitReturn	(msx_fusion.h)
BitReset	(msx_fusion.h)
Vsynch	(msx_fusion.h)
SetScrollMask	(msx_fusion.h)
SetScrollDouble	(msx_fusion.h)
JoystickReadTo	(msx_fusion.h)
StrToLower	(msx_fusion.h)
StrToUpper	(msx_fusion.h)
TurboMode	(msx_fusion.h)
SpriteOverlap	(vdp_sprites.h)
SpriteOverlapId	(vdp_sprites.h)
Pattern8RotationRam	(vdp_sprites.h)
Pattern8RotationVram	(vdp_sprites.h)
Pattern16RotationRam	(vdp_sprites.h)
Pattern16RotationVram	(vdp_sprites.h)

C Library for MSX-DOS with SDCC compiler

Pattern8FlipRam	(vdp_sprites.h)
Pattern16FlipRam	(vdp_sprites.h)
Sprite8FlipVram	(vdp_sprites.h)
Pattern16FlipVram	(vdp_sprites.h)
PatternRotation	(vdp_sprites.h)
PatternHflip	(vdp_sprites.h)
PatternVflip	(vdp_sprites.h)
SpriteFollow	(vdp_sprites.h)
LMCM5	(vdp_graph2.h)
LMCM8	(vdp_graph2.h)
VDPLINE	(vdp_graph2.h)
TurboMode	(msx_fusion.h)
Paint (New Fast Paint for MSX2)	(vdp_paint.h)
SetPaintBuffer	(vdp_paint.h)
SetColor	(msx_fusion.h)
fVDP	(vdp_graph2.h)
Polygon	(vdp_graph2.h)
Rkeys	(msx_fusion.h)
Fkeys	(msx_fusion.h)
ReadKeyboardType	(msx_fusion.h)
VDPalternate	(msx_fusion.h)
VDPinterlace	(msx_fusion.h)
SetExpandVDPcmd	(msx_fusion.h)
SetScreen10	(msx_fusion.h)
SetScreen12	(msx_fusion.h)
SetTransparent	(msx_fusion.h)
SetAdjust	(msx_fusion.h)
fPutSprite	(msx_fusion.h)
InitVDPInterruptHandler	(msx_fusion.h)
EndVDPInterruptHandler	(msx_fusion.h)
PT3FXInit	(ayfx_player.h)
PT3FXPlay	(pt3replayer.h)
PT3FXRout	(pt3replayer.h)
CovoxPlayRam	(msx_fusion.h)
FcbDelete	(msx_fusion.h)
CallSub	(msx_fusion.h)
SaveScreenBoot	(msx_fusion.h)
SetRamDisk	(msx_fusion.h)

- Fixed error or rewritten functions:

SC2Pset (500 % Faster than previous version)	(vdp_graph1.h)
SC2Line (500 % Faster than previous version)	(vdp_graph1.h)
SC2Point	(vdp_graph1.h)
SetScrollIH	(vdp_graph1.h)
SetScrollIV	(vdp_graph1.h)
SetSpriteColors	(vdp_sprites.h)
YMMM Fixed definition last parameter removed	(vdp_graph2.h)
GetKeyMatrix	(msx_fusion.h)
InitInterruptHandler	(msx_fusion.h)
EndInterruptHandler	(msx_fusion.h)
Lseek	(io.h)
MakeDir	(io.h)
ChangeDir	(io.h)
RemoveDir	(io.h)
InitFX	(ayfx_player.h)
PlayFX	(ayfx_player.h)
UpdateFX	(ayfx_player.h)
StopFX	(ayfx_player.h)
PT3Init	(pt3replayer.h)
PT3Play	(pt3replayer.h)
PT3Rout	(pt3replayer.h)
PT3Mute	(pt3replayer.h)

- Removed functions:

SC2Rect	Replaced identically by	SC2BoxLine
DosCLS	Replaced identically by	Cls
SC5SpriteColors	Replaced identically by	SetSpriteColors
SC8SpriteColors	Replaced identically by	SetSpriteColors
HMCM	Replaced by LMCM5	(vdp_graph2.h)
HMCM_SC8	Replaced by LMCM8	(vdp_graph2.h)
WriteScr	Totally removed	(vdp_graph2.h)
ReadScr	Totally removed	(vdp_graph2.h)
KeyboardRead	Replaced by	Rkeys
SetInterruptHandler	Totally removed	(msx_fusion.h)
TestFX	Totally remove	
FreeFX	Totally remove	
IntDos	Replaced by	CallDos
IntBios	Replaced by	Callbios

- Renamed functions :

fcb_open	to FcbOpen
fcb_create	to FcbCreate
fcb_close	to FcbClose
fcb_read	to FcbRead
fcb_write	to FcbWrite
fcb_find_first	to FcbFindFirst
fcb_find_next	to FcbFindNext
SetSC5ColorPalette	to SetColorPalette
SetSC5Palette	to SetPalette
RestoreSC5Palette	to RestorePalette
VDPLineSwitch	to VDPLineSwitch
PSGRead	to PSGread
SetVDPWrite	to SetVDPwrite
SetVDPRead	to SetVDPread
CovoxPlay	to CovoxPlayVram

- Added Example :

code examples completely reworked

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler

Installing the Tools Chain - SDK

What is fundamental nowadays is the ability to code for MSX computers with modern tools. That's why with FUSION-C, you will not need an old computer or old Operating System to make your game for the MSX. You will just need to install some free tools on your **Windows**'s PC, or **MacOS** Apple's computer, and even on **Linux**.

The SDK is composed of the following elements

- **OpenMSX Emulator**

Since Fusion-C v1.3, open MSX is part of Fusion-C SDK

- **OpenMSX Machines System Roms**

The system roms collection of all MSX machines, and peripherals.

- **SDCC (Small Device C Compiler)**

The heart of the SDK.

- **Hex To bin tool**

This tool generate the the binary file you will execute on the MSX.

- **The FUSION-C Library and scripts environnement**

Fusion-C SDK is composed of a library and multiple script, tools, and code examples

- **Sublime Text code editor.**

The editor to type your code

The code editor used is **Sublime Text 3**. It is a very nice and professional code editor, it can be freely used and It is available on the three major operating systems.

If you feel more comfortable with another code editor, you can change to the one you are used to. However you will have to configure it yourself so that it can launch the compilation script ; or start the compilation manually.



PRIVACY
DO NOT SHARE This
Sharing This Document without
is copyright violation

Download the necessary files

1- Download the « SDCC 4.0 Package » for your operating system :

Historically I advised to use the SDCC v3.6 version. But today it will no longer work properly on computers with a 100% 64-bit OS, such as MacOS 10.15 Catalina, this is why an upgrade **to SDCC 4.0** is recommended.

<https://sourceforge.net/projects/sdcc/files/>

2- Download « Sublime Text 3» for your operating system :

Many code editors may be suitable for editing C code. I suggest using the free version of « **Sublime Text 3** ». The installation and the way of working explained in this book, has been designed for this code editor. I suggest you install this editor, to follow the process of this book, and possibly change your code editor later, if you prefer to work with another tool.

<https://www.sublimetext.com/3>

3- Download « openMSX Machines Rom files »

For copyright reasons, the MSX System Rom Files cannot be provided with OpenMSX, or Fusion-C SDK. You can download the full pack of System Rom files for OpenMSX from here:

http://www.ebsoft.fr/msx/roms/OpenMSX_ROMS.zip

or

[https://download.file-hunter.com/System%20ROMs/Full%20Set%20System%20ROM's%20\(OpenMSX\).zip](https://download.file-hunter.com/System%20ROMs/Full%20Set%20System%20ROM's%20(OpenMSX).zip)

4- Download « Fusion-C SDK »

The latest version of « **Fusion-C SDK** » can be downloaded from one of these 2 official sources.

<https://www.ebsoft.fr/shop/fr/19-fusion-c>

or

<https://github.com/ericb59>

C Library for MSX-DOS with SDCC compiler
Manual installation for Windows



Windows

1 - The Working Folder

The Fusion-C Library comes with source codes, tools, examples, scripts, inside a ready to use «WorkingFolder»»

- Unpack the archive, copy the « WorkingFolder » as is, where you want on your computer, that's all ! For example copy it to your Desktop.

2 - Install Sublime Text 3

Install the code editor as any other application.

We need to set up the automatic compilation command. Open Sublime Text, and go to the **Tools's menu**, and choose « **Build System > New Build System** »

Inside the new opened window enter this text:

```
{  
    "cmd": ["$file_path\\fusion-c\\build.bat", "$file_name"]  
}
```

Now save this file to:

« C:\Users\<USER NAME>\AppData\Roaming\Sublime Text 3\Packages\User\ »
name it « **sdcc-build.sublime-build** »

Close Sublime Text, and open it again. Go to the **Tools's menu**, choose « **Build System** », now in the list you must see your « **sdcc-build** » ; click on it to choose it as the default build system. SDCC is nox able to call the compilation script.

3 - Install Hex2Bin

Use Windows to navigate to the folder « **WorkingFolder/Tools/Hex2Bin/Hex2bin Windows/** », copy the file « **hex2bin.exe** » and paste it inside the folder « **C:\Program Files\SDCC\bin**»

4 - Add the System Roms to OpenMSX

OpenMSX is pre-installed inside the « **WorkingFolder** », but it is not provided with the MSX machines system ROMs files. Unpack the **OpenMSX_ROMS.zip** you previously downloaded, and copy the « **systemroms** » folder inside « **./WorkingFolder/openMSX/share/** ».

The new folder replace the old one.

5 - Install SDCC 4.0

Use the SDCC setup you previously downloaded. Do not change the default parameters the installer offers. All the SDCC files will be installed inside the folder: « **C:\Program Files\SDCC** »

Now open a Dos window and type above commands:

```
> CD C:\Program Files\SDCC\lib\z80  
> copy z80.lib z80.save  
> sdar -d z80.lib printf.rel  
> sdar -d z80.lib sprintf.rel  
> sdar -d z80.lib vprintf.rel  
> sdar -d z80.lib putchar.rel  
> sdar -d z80.lib getchar.rel
```

6 - Verification

Inside the Dos window type:

```
> sdcc -v
```

SDCC must answer, and show you its version number

```
> hex2bin
```

Hex2Bin must show you its help page

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

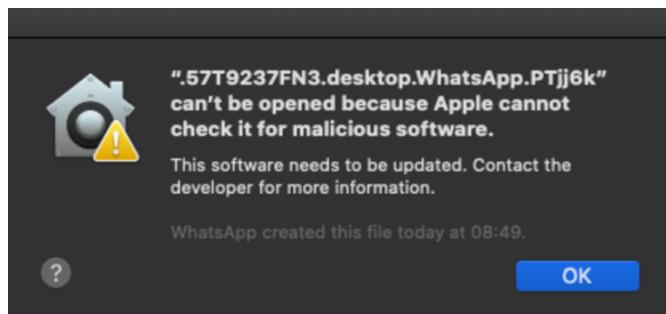
C Library for MSX-DOS with SDCC compiler
Manual installation for MacOS



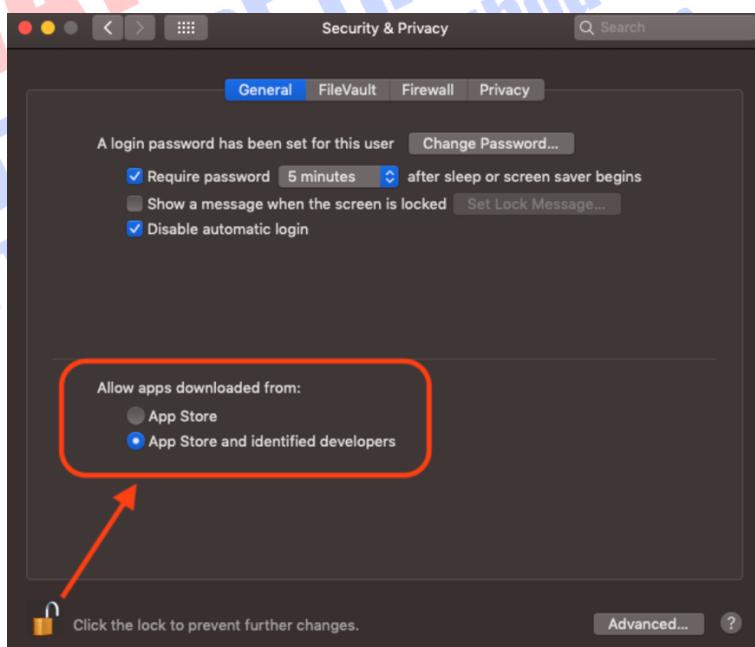
0 – Note for MacOS Catalina users (10.15 +)

We first have to take back control of MacOS Catalina's security system.

Since MacOS CATALINA, (10.15 and +), Apple has tightened up the security rules on the execution of softwares which are not referenced on the AppStore, or which comes from developers not registered with Apple. This causes a lot of problems for homebrew softwares and for tools that come from the Linux world. If you ever see this window, you know what I'm talking about !



Before going any further, I suggest you to take back control of your MacOS 's security system. Open the **System preference** and go to **Security & Privacy**.



You can see there are only 2 options, we can add a third option to enable the execution of programs coming from anywhere. To do that, open a **Shell/Terminal** and enter this command:

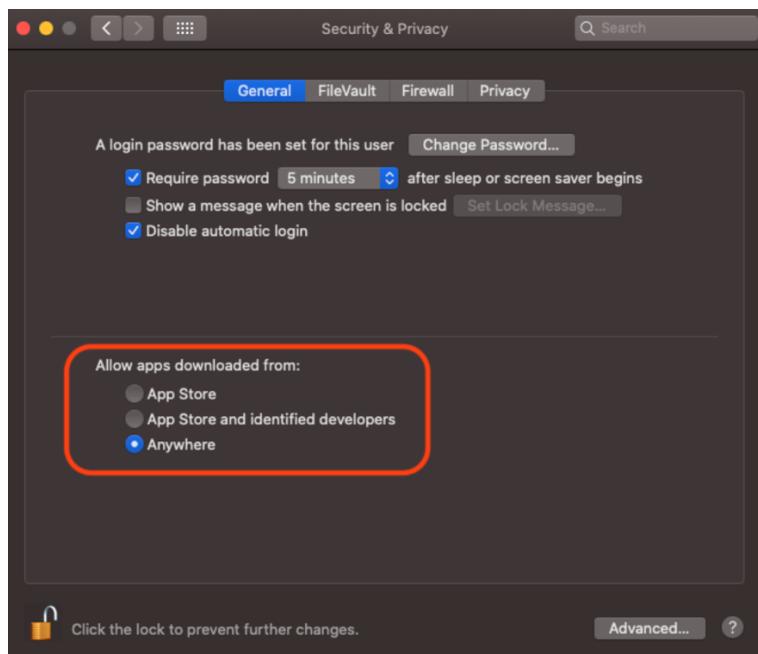
```
> sudo spctl --master-disable
```

we will also take the opportunity to display the hidden files stored on the hard-drive... Type the above commands:

```
> defaults write com.apple.finder AppleShowAllFiles YES  
> killall Finder
```

C Library for MSX-DOS with SDCC compiler

Now, close the **Security & Privacy** window, and open it again, you must now see the Third option appear.



During the installation of **Fusion-C**, and until your first compilation, I suggest you choose this 3rd option, « **Anywhere** ». Once an unidentified authored program has been run, it will continue to run without annoying you, even if you come back to a more restricted security mode.

1 - The Working Folder

The Fusion-C Library comes with source codes, tools, examples, scripts, inside a ready to use « **WorkingFolder** ».

- Unpack the archive, copy the « **WorkingFolder** » as is, where you want on your computer, that's all ! For example copy it to your Desktop.

2 - Install Hex2Bin

Open the Terminal. (*Click on the Search icon, at the top left of the Finder's Menu Bar. Type « Terminal » inside the search field, then press the enter key.*)

Inside the **Termnal** window type: **cd [space]** (*This means, type the letter 'c', 'd' and 'Space Barre'*) Use the Finder, navigate to the folder « **WorkingFolder/Tools/Hex2Bin/** », you must see a folder named « **Hex2bin-2.5 Mac** ». Drag and drop this folder to the **Terminal** window. The full path of the folder must now be written on the Terminal. Press the “**Enter key**”

Now type this command inside the terminal's window:

```
> make install MAN_DIR=/usr/local/share/man/man1
```

This will create the executable and install **Hex2Bin** on your system.

If you haven't done previously we will also take the opportunity to display the hidden files stored on the hard-drive... Type the above commands:

```
> defaults write com.apple.finder AppleShowAllFiles YES  
> killall Finder
```

Keep the Terminal's window open, we will need it again further.

3 - Install Sublime Text 3

Open the **.dmg** file you previously downloaded and drag the **Sublime Text application** to your « **Applications** » folder.

We need to set up the automatic compilation command. Open Sublime Text, and go to the **Tools's menu**, and choose « **Build System > New Build System** »

In the new opened window enter this text:

```
{
    "shell_cmd": "./build.sh $file_name",
    "working_dir": "$file_path"
}
```

Save this file to:

« /Users/<YOUR USER NAME>/Library/Application Support/Sublime Text 3/Packages/User/»
name it « **sdcc-build.sublime-build** »

Close Sublime Text, and open it again. Go to the **Tools's menu**, choose « **Build System** », now in the list you must see your « **sdcc-build** ». Click on it to choose it as the default build system. Now, SDCC is able to call the compilation script inside your working folder and compile the **.c** files you are editing on.

4 - Add the System Roms to OpenMSX

OpenMSX is pre-installed inside the « **WorkingFolder** », but it is not provided with the MSX machines system ROMs files. Unpack the **OpenMSX_ROMS.zip** you previously downloaded, and copy the « **systemroms** » folder inside « **./WorkingFolder/openMSX/share/** ». The new folder replace the old one.

5 - Install SDCC 4.0

Inside a Terminal window, and go to the directory where you previously downloaded the archive. You can do that easily, by typing CD (+space) in the Terminal's window, then drag the **folder** where the archive is and drop it over the Terminal's windows, then press enter.

Extract the files from the archive with this Shell command:

```
> tar xjf sdcc-4.0.0-x86_64-apple-macosx.tar.bz2
```

The filename may change, it depends on the version.

```
> cd sdcc-4.0.0
> cp -r * /usr/local
```

This will install all necessary files on your system.

Now type the following commands to remove commands that are incompatible with the MSX.

```
> cd /usr/local/share/sdcc/lib/z80/
> cp z80.lib z80.save
> sdar -d z80.lib printf.rel
> sdar -d z80.lib sprintf.rel
> sdar -d z80.lib vprintf.rel
> sdar -d z80.lib putchar.rel
> sdar -d z80.lib getchar.rel
```

Now we need to grant new permissions to the compilation script.

Inside the Terminal type this command : **chmod +x [space]**

```
> Chmod +x
```

with a space at the end, after the 'x' (and do NOT press enter yet).

From the finder, open the « **WorkingFolder** » folder and identify the file « **build.sh** » drag it, and drop it to the Terminal's window. The file path and filename should automatically fill in. Click on the Terminal, and press enter to validate the new permission to the script.

If you want to hide the system files again, and go back to the initial configuration type this

```
> defaults write com.apple.finder AppleShowAllFiles NO  
> killall Finder
```

6 - Verification

Inside the Terminal window type:

```
> sdcc -v
```

SDCC must answer, and show you its version number

```
> hex2bin
```

Hex2Bin must show you its help page

SDK Architecture

The «**WorkingFolder**» folder contains all the files of the library, as well as all the necessary tools, such as the emulator, the development tools, the files emulating the floppy disks or hard disk of a real MSX...

Content of the 2 first levels of **WorkingFolder**

```

WorkingFolder
  |_ dsk
    |_ dska
    |_ dskb
    |_ export
    |_ hda-1
    |_ hda-2
  |_ fusion-c
    |_ examples
    |_ header
    |_ include
    |_ lib
    |_ source
    |_ build scripts
    |_ openMSX
      |_ Catapult
      |_ codec
      |_ doc
      |_ hard-drive
      |_ MSX_config
      |_ platforms
      |_ share
    |_ Tools
      |_ ... Various folder and files
  |_ hello.c

```

dsk	Destination folders of compiled programs.
... dska & dskb	Must contain the files you want on floppy drives A and B
... hda-1 & hda-2	These folders must contain the files you want to import to a virtual hard-drive attached to the MSX you are emulating. Hard-drive is separated into 2 partitions, hda-1 and hda-2 .
... export	Contains a copy of the hard-drive partition, exported from the emulator
fusion-c	Main Fusion-C folder
... examples	Example source code
... header	Header files used by Fusion-C. All function definitions.
... lib	Main dynamic Library
... source	Source files of all Fusion-C functions
openMSX	Main folder of openMSX emulator
... catapult	Catapult is a Windows launcher for openMSX. Not used by Fusion-C
... codec	Used by openMSX
... doc	openMSX documentation
... hard-drive	Contain the virtual hard-drive for open MSX
... MSX_config	Contains the configuration files of different MSX platforms
... share	Contains the system ROM, and configurations used by openMSX
Tools	Contains different tools and information for MSX development

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

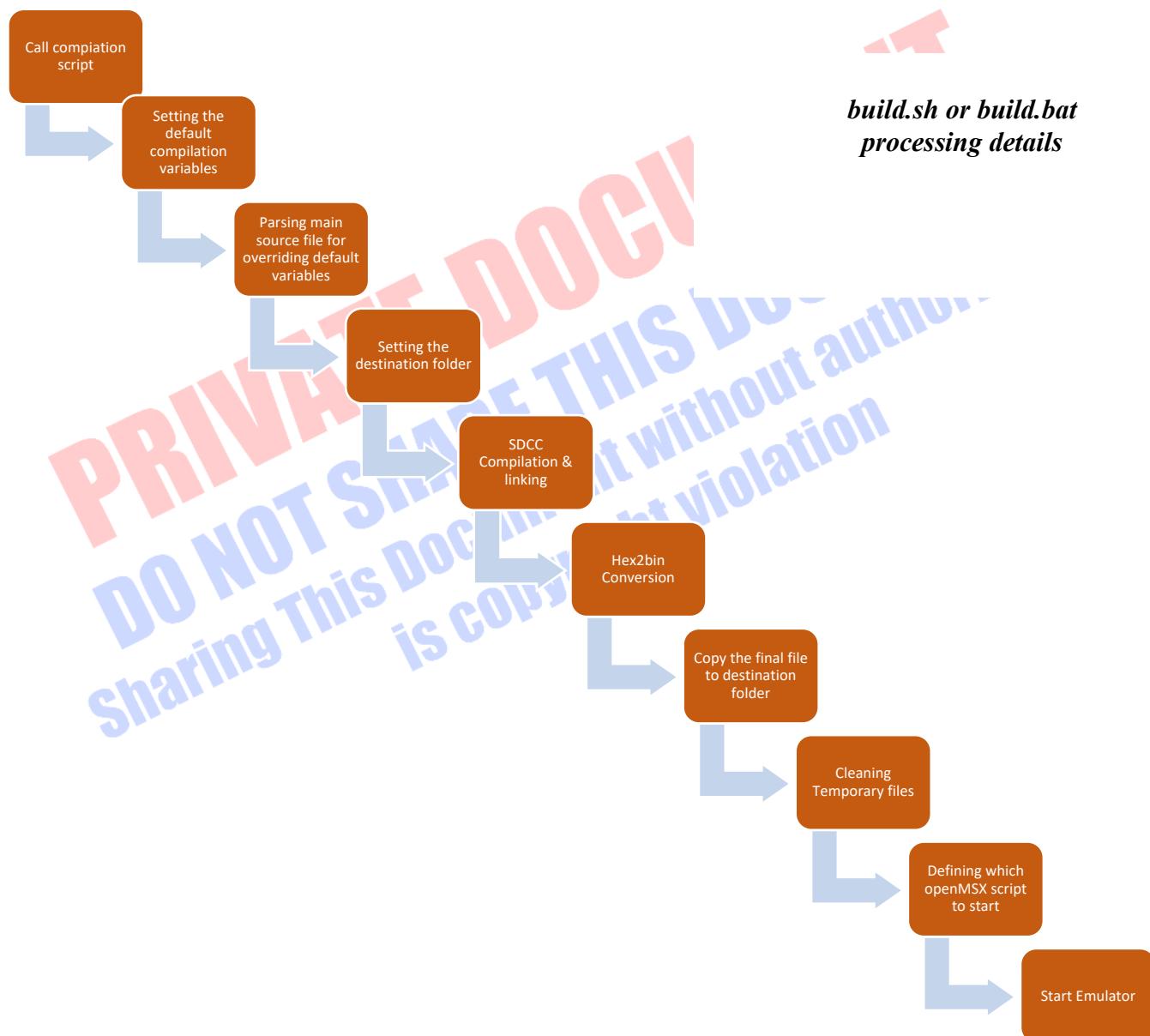
Know the compilation script

The compilation scripts, **build.sh** for Mac OS, and **build.bat** for Windows, are designed to automate the process of building and testing your programs. Scripts can be called automatically from your favorite code editor, Sublime text, VS Code... but also manually from a shell / Dos window.

Compilation scripts are in **WorkingFolder/fusion-c/** folder.

Once the script is called it will do the C compilation, the linking, and transform the generated hexadecimal assembler code to a binary executable file compatible with MSX-DOS, the file will be copied inside the destination folder and openMSX will be launched to test your program.

If you choose to use the whole tools chain, and respect all the installation steps, you may not have to change anything inside the compilation script. Anyway, it's a good idea to see what does the script.



C Library for MSX-DOS with SDCC compiler
openMSX configuration files

At the end of the compilation process, the script will start openMSX emulator to test your program.
(note : If an instance of the emulator is already opened, no new instance will be open).

You can choose which version of the MSX computer will be opened, as well as the peripherals that will be associated with it.

There are 5 configurations already defined. Each configuration is described in a text file which can be found here: **WorkingFolder/openMSX/MSX_config/**

You can modify these basic configurations yourself by editing these files. Various variables allow you to activate or deactivate the most common MSX options and peripherals. For more details, consult the openMSX documentation.

By default, openMSX will be launched with **script n ° 2** which corresponds to an **MSX2 Philips NMS 8255** model, with 2 floppy disk drives, 128K of Ram and 128K of Vram. It is extended with an MSX-DOS2 expansion. A joystick is connected to port A (Emulated by the keyboard keys), a mouse is connected to port B, and a Covox module is connected to the printer port.

Note : These configuration files are also used to define the behavior of OpenMSX, as well as the keys to control the emulator.

Here are the details of the 5 configurations already defined

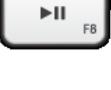
Config File number	Description	
1	MSX1 Sanyo PHC28L. 64K RAM, 16K VRAM Floppy Disk Drive With MSX-DOS1 Joystick in port A Covox/Simpl Module	emulated disk-drive folder: WorkingFolder/dsk/dska
2	MSX2 Philips NMS 8255 128K RAM, 128K VRAM Floppy Disk Drive With MSX-DOS2 Joystick in port A Mouse in port B FMPAC + SCC 1024K Ram Expansion Covox/Simpl Module	emulated disk-drive folder: WorkingFolder/dsk/dska
3	MSX2+ Panasonic FS-A1 WSX MSX Music (FM) 64K RAM, 128K VRAM Floppy Disk Drive With MSX-DOS2 Joystick in port A Mouse in port B SCC 1024K Ram Expansion Covox/Simpl Module	emulated disk-drive folder: WorkingFolder/dsk/dska
4	MSX Turbo-R FS-A1 GT MSX Music (FM) 512K RAM, 128K VRAM Floppy Disk Drive With MSX-DOS2 Joystick in port A Mouse in port B SCC Covox/Simpl Module	emulated disk-drive folder: WorkingFolder/dsk/dska
5	MSX2 Philips NMS 8255 + Hard Drive 128K RAM, 128K VRAM Floppy Disk Drive A Joystick in port A Mouse in port B FMPAC + SCC 1024K Ram Expansion Covox/Simpl Module	emulated Hard-drive folder: WorkingFolder/dsk/hda-1/

Example: the configuration used to start a MSX2 emulation (*emul_start_MSX2_config.txt*)

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Key Maping

The keys you can use to control the openMSX:

openMSX - MSX KEYBOARD MAPPING		
MSX	OSX	Windows
 CTRL	 (Left) control	 (Left) Ctrl
(Dead key. Accent key) 	 (Right) control	 (Right) Ctrl
 GRAPH	 (Left) alt option	 (Left) Alt
 CODE	 (Right) alt option	 (Right) Alt
 SELECT	 (Left) F7	 F7
 STOP	 (Left) F8	 F8
 INS	 + command ⌘	 Insert



PAUSE Emulator	+	
QUIT Emulator	+	+
Save SCREENSHOT	+	
Go 1 Sec. back in time *		
Go 1 Sec. Forward in time *		
Toggle fastforward mode	+	
Toggle Console display	+	
Full Screen mode	+	+
Toggle Audio Mute (Modified from default)	+ +	+ +
Quick Load State	+	+
Quick Save State	+	+
COPY from MSX to clipboard	+	+ +
PASTE clipboard into MSX	+	+ +

* If reverse feature is enabled



Cycle from different video sources if exists



Force or release all inputs grabbed by OpenMSX



Decrease scale of the emulator window



Increase scale of the emulator window



Decrease emulation Speed by 10%



Increase emulation Speed by 10%



Import files to MSX virtual Hard-Drive. *



Export content of MSX Hard-Drive. **



Reset The Emulated MSX



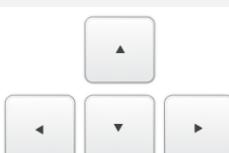
Toggle Power ON/OFF



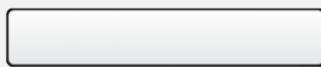
Type the content of the current autoexec.bat to screen



Joystick 1



Joystick 1 - Button A



Joystick 1 - Button B



* if MSX Hard-drive is enabled. Copy dsk/hda-1/*.* to Virtual HD Drive, HDA partition 1

** if MSX Hard-Drive is enabled. Content of Partitions 1 & 2 is copied to dsk/export/

SUBLIME TEXT 3



Start the compilation



VS CODE



Start the compilation



PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Know the MSX Floppy Drive and Hard-Drive

You can put your source codes inside the folder « **WorkingFolder** ». Once you source code is compiled it will be copied to the **./dsk/** folder, which is predefined as the floppy drive **A:** of the emulated MSX. Inside the **./dsk/** folder you must also have the files that are required to start the MSX-DOS operating system : **COMMAND.COM** and **MSXDOS.SYS** (or **COMMAND2.COM** and **MSXDOS2.SYS**). If your program need other files to work, like images, of any other type files, you can copy them inside the **./dsk/** folder.

Considering this folder acts as a floppy drive, its maximum capacity is 720 KB. Files that exceed the 720KB are ignored.



In case you need much more space for your MSX project, you can use a virtual Hard-Drive for the MSX computer. As you may know, there are several interfaces that allow you to install a hard drive on a real MSX computer. The one we are using by default is the Sunrise IDE interface.

You can activate this Hard-Drive, by modifying the configuration file of the MSX machine you want to use, which is in the folder **./openMSX/MSX_config/**

The Hard-Drive is available for the MSX2, MSX2+, and the MSX Turbo-R.

Edit the configuration file and just change the variable **USE_HARDDRIVE** value from 0 to 1 to activate the Hard-Drive instead of the Floppy Drive. The new configuration will be applied the next time openMSX is launched. You should have something like this:

```

variable USE_HARDDRIVE 1
variable USE_DOS2 0
  
```

The virtual Hard-Drive is physically represented by the file **virtual-HardDrive.dsk** which is in the folder **./openMSX/**. The Hard-Drive capacity is 20 MB, in one and unique partition.

Unfortunately we do not have direct access to the contents of the Hard-Drive, as we have for the floppy drive. This is why we will use in this case the **./dsk/** folder's content to fill the virtual hard disk with the files we need.

Each time the openMSX emulator will be launched with the Hard-Drive enabled, the virtual Hard-Drive will be totally erased, then the content of the **./dsk/** folder will be copied to the Hard-Drive. That way you don't have to worry about anything. The latest versions of your files will always be accessible on the virtual Hard-Drive. The capacity of **./dsk/** folder may not exceed the Hard-Drive capacity (20MB by default).

If you are used to not closing the openMSX window after each test, or each step of your project, you need to update the content of the virtual Hard-Drive manually by pressing the **END** key on your keyboard. By pressing **END**, you launch the procedure for updating the Hard-Drive's content (Erasing Hard-Drive, then copy all files from **./dsk/** to the Hard-Drive



Start your first compilation

Now all your setup must be ready, it's time to start your first compilation for MSX. Open SublimeText, and load **hello.c** from the « **WorkingFolder** ». To launch the compilation...

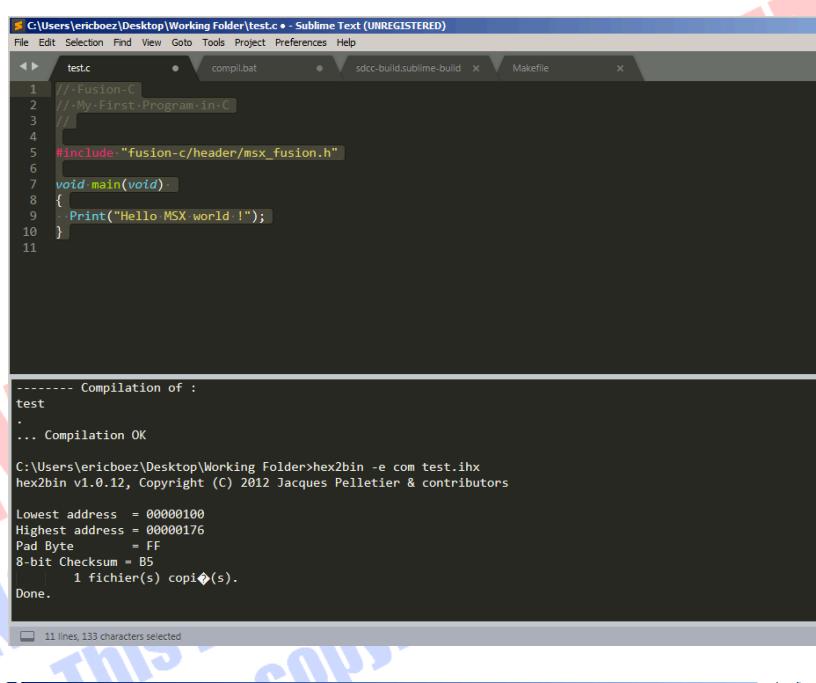
If you are on MacOS press:



If you are on Windows press:

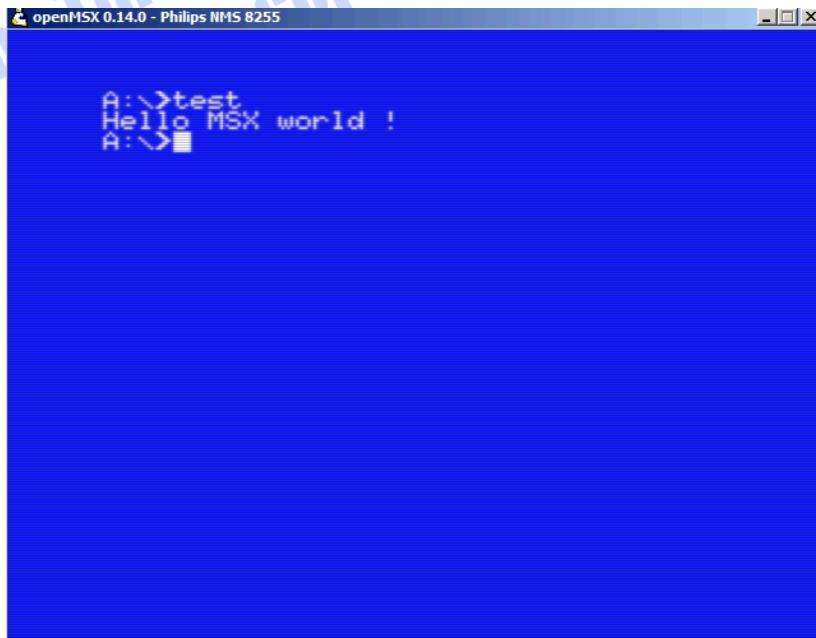


If all goes well, you must see at the bottom of the Sublime text window that the program is well compiled, and the openMSX must open itself and the **hello.com** program should start automatically



A screenshot of the Sublime Text editor window. The title bar says "C:\Users\ericboez\Desktop\Working Folder\test.c - Sublime Text (UNREGISTERED)". The main pane shows the C code for "test.c". Below the code, the terminal output shows the compilation process:

```
----- Compilation of :  
test  
. . . . .  
... Compilation OK  
  
C:\Users\ericboez\Desktop\Working Folder>hex2bin -e com test.ihx  
hex2bin v1.0.12, Copyright (C) 2012 Jacques Pelletier & contributors  
  
Lowest address = 00000100  
Highest address = 00000176  
Pad Byte = FF  
8-bit Checksum = B5  
1 fichier(s) copié(s).  
Done.
```



A screenshot of the openMSX 0.14.0 emulator window. The title bar says "openMSX 0.14.0 - Philips NMS 8255". The screen displays the MSX command prompt "A:>" followed by the output of the program:

```
A:>test  
Hello MSX world !  
A:>
```



Compilation not working with MacOs Catalina, what can I do ?

If you are using MacOS Catalina or superior (10.15 +)

Catalina may deny you access to certain tools. In this case, proceed as follows:

With the Finder navigate to the folder **/usr/local/bin/**

(*You must have previously activated the display of hidden files mode. See the chapter 'Manual Installation dor MacOS> Note for MacOS Catalina's users*)

Sort the files by "Type", and select all the "**Unix Executable**":

```
as2gbmap    sdas6808    sdasrab    sdcdb.e1    sdldpdk    shc08
makebin     sdas8051    sdasstm8    sdcdbsrc.e1  sdldstm8   spdk
packihx     sdasgb     sdastlcs90  sdcpp       sdldz80    sstm8
s51         sdaspdk13   sdasz80    sdld        sdnm       stlcs
sdar        sdaspdk14   sdcc       sdld6808   sdobjcopy  sz80
sdas390    sdaspdk15   sdcdb      sdldgb     sdranlib
```

Do a right mouse click on this selection, while holding down the **CTRL** key of the keyboard, then choose **Open** inside the contextual menu. Finder will ask you for confirmation, you must accept. For some files, an Alert will be displayed "*MacOS cannot verify the developer*", choose the option to open anyway.

Now navigate to « **WorkingFolder/openMSX** », Do a right mouse click on the file « **openMSX.app** », and while holding down the **CTRL** key of the keyboard, choose **Open** inside the contextual menu.

How can I call compilation script manually from anywhere ?



With MacOS one way is to add a symbolic link to the compilation script.

Open the Terminal, and type this command:

```
> ln -s /Users/<your folder name>/Desktop/WorkingFolder/fusion-c/build.sh
/usr/local/bin/compil
```

This command creates a symbolic link for the compilation script "build.sh" which is located in "WorkingFolder/fusion-c /" to a virtual command which is located in "/usr/local/bin".

This assumes that you have placed the "WorkingFolder" on your desktop. Change the path of the command if necessary.

Now, you can call the compilation manually from the Terminal, by typing:

```
> compil myprog.c
```

You just have to make the call from the folder where the C source file is.

For example, if the source file is in Desktop/C-source/

```
> CD Desktop/C-source/
> compil myprog.c
```

Windows

With Windows one way is to add the path to the compilation script in the Windows environment variables.

First open the DOS window prompt as administrator. To do this, click **Start** or press the **Windows key**, type the letters **CMD**, and right-click on the **Command Prompt** entry at the top of the Start menu and choose **Run as administrator** from the context menu, then type this command line inside the DOS's window:

```
> setx PATH "%PATH%;C:\Users\<your folder name>\Desktop\WorkingFolder\fusion-c\"  
> shutdown /r
```

This assumes that you have placed the “WorkingFolder” on your desktop. Change the path of the command if necessary. The “Shutdown” will restart your computer, it’s necessary to apply the changes.

Now, you can call the compilation’s script manually from the DOS prompt, by typing:

```
> CD desktop\C-source  
> build.bat myprog.c
```

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler
Use Microsoft Visual Studio Code

If you are familiar with the **Microsoft VS Code** you will be delighted to be able to use this software to generate code for the MSX. If you are not comfortable with Sublime Text 3, or just want to try an alternative code editor, **VS Code** is a great alternative, it is free and has a version suitable for all 3 major operating systems. I will explain here how to configure **VS Code** to use the Fusion-C.

1 – Download VS Code

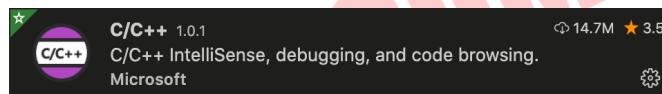
Go to this page to download the latest version of **VS Code for your operating system**:
<https://code.visualstudio.com/download>

2 – Install VS Code

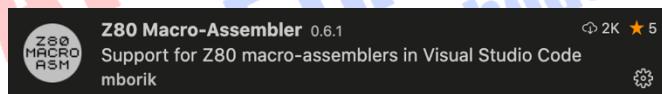
MacOS Users, you just have to move the application to your Application's folder, and open the VS Code application. **Windows's users**, install VS Code as any other windows Application.

Now we will start by installing the C/C++ language support to VS Code.

 Click on the “**Extensions**” icons inside the left vertical menu bar, inside the search field, enter the text “C/C++”. Choose the **Microsoft C/C++ extension**, then install it.



We also need to add Z80 Assembler langage support, so, go back to the search field, and enter the text “Z80”. The extension you must install is the **Z80 macro-assembler**.



Now go to the menu **File > Open** then navigate to your user's Desktop and choose to open “WorkingFolder”. At this point **VS Code** will create a “**.vscode**” sub folder inside the main folder.

 Click on the “**Explorer**” icon inside the left vertical bar. The content of the “WorkingFolder” must appear as a file list in the explorer’s window.

Open the “**hello.c**” source code by clicking on it. From the main menu, choose **Terminal > Configure Default Build Task**. A dropdown will appear listing various predefined build tasks for the compilers that VS Code found on your machine. Choose “**open tasks.json file**”, replace the existing configuration by this one to teach VS Code how to compile the source code by using our compilation script:



```
.vscode/task.json
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "shell",
      "label": "Fusion-C SDCC Build",
      "command": "./fusion-c/build.sh",
      "args": [
        "${file}"
      ],
      "options": {
        "cwd": "${workspaceFolder}"
      },
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "presentation": {
        "echo": false,
        "focus": false,
        "reveal": "always",
        "clear": true,
        "showReuseMessage": false,
        "panel": "shared"
      }
    }
  ]
}
```



Windows

```
.vscode/task.json
{
  "version": "2.0.0",
  "tasks": [
    {
      "type": "shell",
      "label": "Fusion-C SDCC Build",
      "command": "fusion-c\build.bat",
      "args": [
        "${file}"
      ],
      "options": {
        "cwd": "${workspaceFolder}"
      },
      "group": {
        "kind": "build",
        "isDefault": true
      },
      "presentation": {
        "echo": false,
        "focus": false,
        "reveal": "always",
        "clear": true,
        "showReuseMessage": false,
        "panel": "shared"
      }
    }
  ]
}
```

Once done, save the **tasks.json** inside the “**.vscode/**” sub folder.

C Library for MSX-DOS with SDCC compiler

VS Code may ask you to configure the “`c_cpp_properties.json`”. If not, just create or modify this file inside the “`.vscode/`” sub folder.

You must add the path to the SDCC Include directory, and to the Fusion-C Header directory. Here what you must have:



`.vscode/c_cpp_properties.json`

```
{  
    "configurations": [  
        {  
            "name": "Mac",  
            "includePath": [  
                "${workspaceFolder}/**",  
                "/usr/local/share/sdcc/include/**",  
                "${workspaceFolder}/fusion-c/header/**"  
            ],  
            "defines": [],  
            "cStandard": "gnu17",  
            "intelliSenseMode": "gcc-x64"  
        },  
        {  
            "version": 4  
    }  
}
```

Once done, save the `c_cpp_properties.json` file inside the “`.vscode/`” sub folder.

Now come back to the “**hello.c**” tab. From the main menu choose Terminal > Run Build Task Or press:



Windows

`.vscode/c_cpp_properties.json`

```
{  
    "configurations": [  
        {  
            "name": "Win",  
            "includePath": [  
                "${workspaceFolder}/**",  
                "C:/Program Files/SDCC/include/**",  
                "C:/Users/ericboez/Desktop/WorkingFolder/fusion-c/header/**"  
            ],  
            "defines": [],  
            "cStandard": "gnu17",  
            "intelliSenseMode": "msvc-x64"  
        },  
        {  
            "version": 4  
    }  
}
```

Once done, save the `c_cpp_properties.json` file inside the “`.vscode/`” sub folder.

Now come back to the “**hello.c**” tab. From the main menu choose Terminal > Run Build Task Or press:



C Library for MSX-DOS with SDCC compiler
Overriding default compilation directives

The compilation parameters are defined inside the compilation script (build.sh / build.bat). Until now, when you wanted to modify these parameters, it was necessary to modify the script. It is now possible to define some parameters directly from the source code of your program.

The **Fusion-C 1.3** compilation script incorporates the detection of commands to override default settings. The way to do it is simple, just use the C preprocessor command, **#define** with an associated key word.

For example, if you put this command at the top of your C listing, it will tell to the compilation script to use the **crt0_advanced_msxdos.rel** file instead of the default **crt0_msxdos.rel**.

```
#define __SDK_CRT0__ crt0_advanced_msxdos.rel
```

Preprocessor directives like **#define** do not impact your final program. Even if the detection of Overrinding commands is disabled, this will have no harmful effect.

Override parameter	Effect	Default value	Possible values
--SDK_OPTIMIZATION__	Changes the optimization parameter SDCC will use to build the of the final code.	0	0 : Size optimization 1 : Speed optimization
--SDK_MSXVERSION__	Which version of MSX to start at the end of compilation porcess.	2	0 : Do not start openMSX 1 : Start MSX1 with Disk-drive 2 : Start MSX2 with Disk-drive 3 : Start MSX2+ with Disk-drive 4 : Start Turbo-R with Disk-drive 5 : Start MSX2 with Hard-drive 6 : Start MSX2+ with Hard-drive 7 : Start Turbo-R with Hard-drive
--SDK_ADDRCODE__	Defines the code-loc address	0x106	Any valid Adresse. Example: must be 0x170 if using crt0_msxdos_advanced.rel
--SDK_ADDRDATA__	Defines the data-loc address	0x00	Any valid Adresse.
--SDK_CRT0__	Defines hich crt0 to use	crt0_msxdos.rel	Any crt0_xxx_xx.rel file in fusion-c/include directory.
--SDK_DEST__	Destination folder of the final file	dsk/dska/ or dsk/hda-1/	Any accessible folder.
--SDK_AUTOEXEC__	defines if the autoexec.bat file must be written to start the compiled program when starting OpenMSX.	1	0 : do not write autoexec.bat 1 : write autoexec.bat
--SDK_EXT__	Defines the default file extension of the final compiled program.	com	Any valide extension. Example: rom
--SDK_VERBOSE__	Defines the level of information displayed during compilation	2	0 : Minimal information displayed 1 : Medium information displayed 2 : Full information displayed
--SDK_MSXDOS__	Defines the MSX-DOS version to use when booting openMSX. This parameter force the compilation script to copy the defined MSX-DOS files to the Destination folder.	0	0 : Do not change files 1 : Use MSXDOS 1 files 2 : Use MSXDOS 2 files

Example of use

```
// Compilation & SDK directives
#define __SDK_OPTIMIZATION__ 1
#define __SDK_MSXVERSION__ 5
#define __SDK_AUTOEXEC__ 1
```

Note : If you want to disable the detection of Overriding commands, you must edit the file 'build.sh' / 'build.bat' and change the value of the **CHECK_OVERRIDE** variable to **0**

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Example of a C program

```

1 #include <stdio.h>
2 #include "fusion-c/header/msx.h"
3 #include "fusion-c/header/msx_misc.h"
4 #include "fusion-c/header/turbor.h"
5
6 #define HALT __asm halt __endasm //wait for the next interrupt
7
8 static const unsigned char ball_pattern[] = {
9     0x3C, 0x7E, 0xFF, 0xFF, 0xFF, 0xFF, 0x7E, 0x3C };
10
11 int x[32];
12 int y[32];
13 int vx[32];
14 int vy[32];
15
16 /**
17 | FT_Wait : Wait for j x CPU Cicles
18 | -----
19 */
20 void FT_Wait(int cicles)
21 {
22     int i;
23     for(i=0;i<cicles;i++) HALT;
24     return;
25 }
26
27 /**
28 | FT_INIT : Initialisation of all coodonates
29 | -----
30 */
31 void FT_init(void)
32 {
33     char i;
34     for( i = 0; i < 32; i++ )
35     {
36         x[i] = (i * 37 + 5) & 255;
37         y[i] = (i * 19 + 7) & 255;
38         vx[i] = (((i * 23) &
39         vy[i] = (((i * 57) &
40
41 void main( void ) {
42     int i, j;
43
44     Screen(1);
45     SetSpritePattern( 0, ball_pattern );
46     FT_init();
47     if(ReadMSXtype()==3) // IF MSX is Turbo-R Switch CPU to Z80 Mode
48     {
49         ChangeCPU(0);
50     }
51     printf("..... Sprites Demo ....");
52     /* loop */
53     for (j=0; j<200; j++)
54     {
55         FT_Wait(1); // Wait One CPU Cicle to slow down program
56
57         for( i = 0; i < 32; i++ )
58         {
59             PutSprite( i, x[i], y[i], 0, (i & 7) + 8 );
60
61             x[i] = x[i] + vx[i];
62             if( x[i] > 255 )
63             {
64                 x[i] = 255;
65                 vx[i] = -vx[i];
66             }
67             else if( x[i] < 0 )
68             {
69                 x[i] = 0;
70                 vx[i] = -vx[i];
71             }
72
73             y[i] = y[i] + vy[i];
74             if( y[i] > 211 )
75             {
76                 y[i] = 211;
77                 vy[i] = -vy[i];
78             }
79             else if( y[i] < 0 )
80             {
81                 y[i] = 0;
82                 vy[i] = -vy[i];
83             }
84         }
85     }
86 }
87
88

```

Function made inside
the program

Instruction from
Standard-C

Instruction from
Fusion-c

PR
DO
Share

C Library for MSX-DOS with SDCC compiler

The first four lines of this program are the « includes » that allow the compiler to know in which library to find the functions of your program. Note that the definition files which compose your own library must be included with a full path to them.

At line 6 you can see a Define. The Define is some kind of « Search and Replace » routine made by the pre-compiler. It will replace the first part of the sentence by the second part. In our example, « HALT » will be replaced by « __asm halt __endasm » in the source code, before compilation.

From lines 8 to 14, there are global variable definitions. The global variables are available from any function of the program, without the need to pass them thru function variables.

At line 19. There is our first function called « FT_Wait ». It does not return anything (void), and to be used you must enter a variable named « cycles » which is an « int » (Number).

From line 44 to the end, there is the « Main » function. The « main » is the part of the program which will be executed at first. This is an indispensable part of any C program.

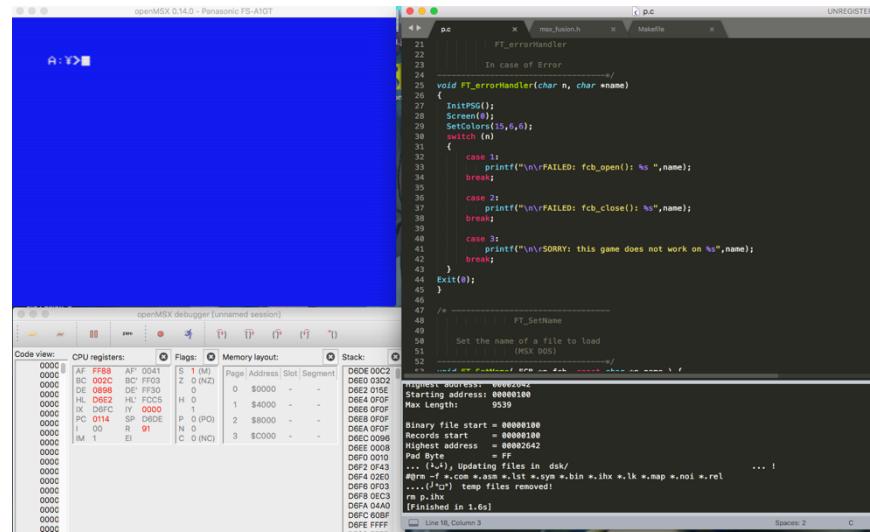
Note that the way the functions are named. All standard C functions are in lower case. Check line 54: printf.

All functions coming from the FUSION-C Library always begins with an upper case letter, and has an upper-case letter at the beginning of each word that compose the function name. Check line 52 « ChangeCpu », is a function defined in « turbor.h »

As a good practice, we encourage you to name your own functions with a distinctive character. Check lines 29, and 19. The functions that compose the program itself always start with « FT_ ».

Thus, when you read the code listing, you can know where come from the function, it's easier to debug your program.

C Library for MSX-DOS with SDCC compiler
Example of the working environment



On the upper left of the screen, the OpenMSX Emulator's window which start the compiled program.
On the lower left, the OpenMSX debugger, just in case ...
On the right side, The Sublime text window. You can notice on the lower side of this window, the compilation information provided by SDCC and Hex2Bin once the program is compiled.
Of course you can manage windows in the way you want.

C Library for MSX-DOS with SDCC compiler
Content of the FUSION-C SDK

The Library

Composed of **297** functions dedicated to the MSX. We will describe each of them in the next chapters.

Functions are dispatched in some header files:

ayfx_player.h	:	AYFX, Sound FX player
io.h	:	File I/O functions
msx_fusion.h	:	Primary and essential functions are here
psg.h	:	PSG Sound functions
pt3replayer.h	:	PT3 Music replayer functions
rammapper.h	:	MSX-DOS 2 Memory Mapper
vdp_circle.h	:	Drawing circles on graphic screens
vdp_graph1.h	:	Graphic functions for MSX1
vdp_graph2.h	:	Graphic functions for MSX2
vdp_paint.h	:	MSX2 Fast painting function
vdp_sprites.h	:	Sprite related functions

Optional header files

newTypes.h	:	Definition of old school variables
vars_msxBios.h	:	Definition & list of MSX BIOS 's routines
vars_msxDos.h	:	Definition & list of MSX-DOS 's routines
vars_msxSystem.h	:	Definition & list of System variables
g9klib.h	:	V9990 GFX9000 Support functions (Beta version)
macro.inc	:	part of the g9klib.h
gr8net-tcpip.h	:	TCPIP Functions for the Gr8NET cartridge (Beta version)

The tools

to help you and facilitate the development of games for MSX are provided in the package.

Image To Sprite Editor	:	Tool to transform black and white image icons 16x16 sprites.
Sprite Path Editor	:	Tool to draw paths or routes, retrieve the coordinates of each point.
RLEWB Compressor	:	A command line tool to compress data into the RLEWB format.
SC2 GraphX Conv	:	A command line tool to transform bitmapped image to the MSX Screen 2 format, with a very good algorithm.

Other tools from other developpers:

AYFX Edit	:	A Windows Only editor to edit AYFX sound effects.
Bin2Hex	:	A command line tool to transform data into Hexadecimal array.
BitBuster	:	Tools for compression (Windows) decompression (MSX) datas.
Disk Image Managers	:	Three tools to edit and manage disk images.
Disk2Rom	:	Tool to transform a 720K Disk Image into a ROM file
DskTool	:	A command line editor to manage disk images
MSX Graphic Palette	:	MSX Graphic palette information.
MSX Image Viewer	:	A Windows to show image save in MSX format.
nMSXTiles	:	A tool to draw and create tiles.
siasm42c	:	A Z80 assembler compiler.
SpriteSX	:	A Windows tool to create sprites.
Vortex Tracker	:	A Windows tool to create music in PT3 format



Functions List

MSX FUSION.....	68
Console Functions	68
CheckBreak.....	68
Getche.....	68
InputChar.....	68
InputString.....	68
Locate.....	68
PrintHex.....	68
PutCharHex.....	68
Print.....	69
PrintNumber.....	69
PrintFNumber.....	69
PrintChar.....	69
PrintDec.....	69
printf	70
Miscellaneous Functions	71
Cls.....	71
KeySound.....	71
FunctionKeys.....	71
ChangeCap.....	71
ReadMSXtype	71
ReadKeyboardType	72
Screen.....	72
Beep.....	72
RealTimer.....	72
SetRealTimer.....	72
CovoxPlayVram	72
CovoxPlayRam.....	73
RleWBToRam.....	73
RleWBToVram.....	73
PatternRotation.....	73
PatternHFlip	73
PatternVFlip	74
TurboMode.....	74
Joystick & mouse functions.....	75
JoystickRead.....	75
TriggerRead.....	75
JoystickReadTo	76
MouseRead.....	77
MouseReadTo.....	77
Keyboard Functions.....	78
GetKeyMatrix.....	78
Inkey	78
KillKeyBuffer.....	78
WaitKey	78
Rkeys	79
Fkeys.....	79
I/O Port Functions.....	80
OutPort.....	80
InPort	80
OutPorts	80
VDP Functions	81
VDPstatus.....	81
VDPstatusNi.....	81
VDPwriteNi.....	81
VDPwrite	81
IsVsync	81
IsHsync	81
Vsynch	81
Vpeek.....	81

C Library for MSX-DOS with SDCC compiler

Vpoke	82
VpokeFirst	82
VpokeNext	82
VpeekFirst	82
VpeekNext	82
Width	82
SetColors	82
SetColor	82
SetBorderColor	82
SetColorPalette	83
SetPalette	83
SetTransparent	83
RestorePalette	84
SetDisplayPage	84
SetActivePage	84
SetScrollH	84
SetScrollV	84
SetScrollMask	85
SetScrollDouble	85
HideDisplay	85
ShowDisplay	85
FillVram	85
PutText	85
VDP50Hz	85
VDP60Hz	85
VDPLineSwitch	86
CopyRamToVram	86
CopyVramToRam	86
GetVramSize	86
SetVDPwrite	86
SetVDPread	86
SetExpandVDPCmd	86
VDPAlternate	87
VDPInterlace	88
SetScreen10	88
SetScreen12	88
SetAdjust	89
SaveScreenBoot	89
Type Functions.....	90
IsAlphaNum	90
IsAlpha	90
IsAscii	90
IsCtrl	90
IsDigit	90
IsGraph	90
IsLower	91
IsUpper	91
IsPrintable	91
IsPunctuation	91
IsSpace	91
IsHexDigit	91
IsPositive	92
IntToFloat	92
IntSwap	92
String Functions.....	93
CharToLower	93
CharToUpper	93
StrCopy	93
NStrCopy	93
StrConcat	93
NStrConcat	93
StrLen	93
StrCompare	93
NStrCompare	94

C Library for MSX-DOS with SDCC compiler

StrChr.....	94
StrPosStr.....	94
StrSearch.....	94
StrPosChr	94
StrLeftTrim.....	94
StrRightTrim	94
StrReplaceChar.....	94
StrReverse	95
Itoa	95
StrToLower	95
StrToUpper.....	95
Memory Functions	96
Poke	96
Pokew.....	96
Peek	96
Peekw.....	96
MemChr	96
MemFill (aka FillRam)	96
Memcpy.....	96
MemcpyReverse	96
MemCompare	96
MMalloc.....	97
ReadTPA.....	97
ReadSP.....	97
BitReturn.....	97
BitReset.....	97
Interrupt Functions.....	98
EnableInterrupt.....	98
DisableInterrupt.....	98
Halt	98
Suspend.....	98
SetInterruptHandler.....	98
EndInterruptHandler	98
SetVDPInterruptHandler	98
EndVDPInterruptHandler.....	99
PSG Functions.....	100
InitPSG.....	100
PSGread	100
PSGwrite	100
MSX-DOS File I/O Functions	101
Predefined macros & structures	101
FcbOpen.....	101
FcbDelete	101
FcbCreate	102
FcbClose	102
FcbRead	102
FcbWrite	102
FcbFindFirst	102
FcbFindNext	102
SetDisk	102
GetDisk	102
Other MSX-DOS Functions.....	103
Predefined macros & Structures	103
GetDate	103
GetTime	103
SetDate	103
SetTime	103
Exit.....	104
CallBios.....	104
CallDos	104
CallSub	104
SetRamDisk.....	105

C Library for MSX-DOS with SDCC compiler

Turbo-R Functions	105
GetCPU.....	105
ChangeCPU.....	105
PCMPlay.....	105
File I/O	106
Predefined macros & Structures.....	106
DiskLoad.....	107
GetOSVersion.....	107
Open.....	107
Create.....	107
Close.....	107
Read.....	107
FCBlist.....	108
Write.....	108
Ensure.....	108
OpenAttrib.....	108
CreateAttrib.....	108
GetCWD.....	108
Ltell.....	108
Lseek	109
Remove.....	109
Rename	109
FindFirst.....	109
FindNext.....	109
MakeDir.....	109
RemoveDir.....	110
GetDiskParam	110
SetDiskTrAddress	110
GetDiskTrAddress	110
SectorRead.....	110
SectorWrite	110
MSX1 GRAPHICS.....	111
Predefined macros & Structures.....	111
SC2WriteScr	111
SC2ReadScr	111
Get8px	111
ReadBlock.....	111
WriteBlock.....	111
Get1px	112
Set8px	112
Set1px	112
Clear8px	112
Clear1px	112
GetCol8px	112
SetCol8px	112
SC2Point.....	112
SC2Pset.....	112
SC2Line.....	113
SC2Paint	113
SC2BoxFill	113
SC2BoxLine.....	113
MSX2 GRAPHICS.....	115
Predefined Structures and macros	115
Logical operations	115
vMSX	116
Pset	116
Point	116
Line.....	116
BoxLine	116
BoxFill	116
HMMC	118
LMMC	118
LMCM5	118
LMCM8	119

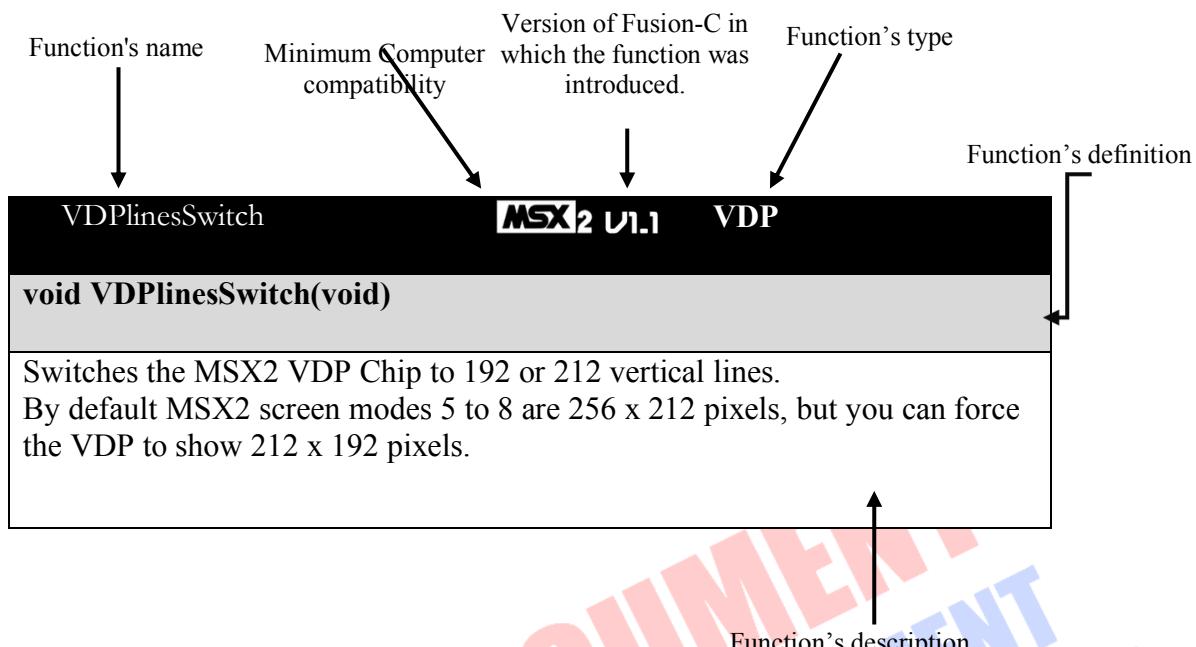
C Library for MSX-DOS with SDCC compiler

YMMM.....	119
LMMM.....	119
HMMM.....	120
HMMV.....	120
LMMV.....	121
VDPLINE.....	121
fLMMM.....	122
fVDP.....	123
PAINT.....	126
Paint.....	126
SetPaintBuffer	126
SPRITES.....	128
SpriteOn.....	128
SpriteOff.....	128
Sprite8.....	128
Sprite16.....	128
SpriteSmall.....	128
SpriteDouble.....	128
SpriteReset	128
SpriteCollision.....	128
SpriteCollisionX	128
SpriteCollisionY	129
PutSprite.....	129
fPutSprite	129
SetSpritePattern	129
Sprite32Bytes	130
SpriteOverlap.....	130
SpriteOverlapId	130
SetSpriteColors.....	131
Pattern16RotationVram.....	131
Pattern8RotationVram.....	131
Pattern8RotationRam.....	132
Pattern16RotationRam.....	132
Pattern8FlipRam.....	132
Pattern16FlipRam.....	133
Pattern8FlipVram.....	133
Pattern16FlipVram.....	133
SpriteFollow	134
CIRCLE.....	136
CircleFilled.....	136
Circle.....	136
SC2CircleFilled	136
SC2Circle	136
MSX-DOS 2 RAM MAPPER.....	138
InitRamMapperInfo	138
Get_PN.....	138
Put_PN.....	138
AllocateSegment.....	138
FreeSegment.....	138
PSG.....	140
Sound	140
SetChannelA.....	140
SilencePSG.....	140
GetSound.....	140
SetTonePeriod	140
SetNoisePeriod	140
SetEnvelopePeriod	140
SetVolume.....	140
SetChannel	140
PlayEnvelope.....	141
SoundFX	141
AYFX PLAYER.....	142

C Library for MSX-DOS with SDCC compiler

InitFX	142
PlayFX.....	142
UpdateFX.....	142
StopFX.....	142
MUSIC PT3 + AYFX REPLAYER.....	144
PT3Init.....	144
PT3Play	144
PT3Rout.....	144
PT3Mute	144
PT3FXInit.....	144
PT3FXPLay	144
PT3FXRout.....	145
FUSION-C ENVIRONMENT VARIABLES	146
(SpriteOn.....	146
(SpriteSize.....	146
(SpriteMag.....	146
(DisplayPage.....	146
(ActivePage	146
(VDPfreq.....	146
(VDPLines.....	146
(ForegroundColor	146
(BackgroundColor	146
(BoderColor.....	146
(ScreenMode	146
(SpritePatternAddr.....	147
(SpriteAttribAddr.....	147
(SpriteColorAddr.....	147
(WidthScreen0.....	147
(WidthScreen1.....	147
(FusionVer.....	147
(FusionRev	147
Jump and Loop.....	148
Clock and Time.....	148
Conditions.....	148
Conversions.....	148
Loading and Saving.....	148
Graphics and Screen.....	149
Math	149
Ram Access.....	149
Sprites.....	150
Strings.....	150
Variables settings	150
CTYPE.H.....	160
MATH.H.....	161
STDLIB.H.....	164
STRING.H	165
STDARG.H.....	166
TIME.H.....	167

C Library for MSX-DOS with SDCC compiler
Note about function's description



No icon means the function works on all MSX computers from the first **MSX1** to the **MSX Turbo-r**



This icon indicates the function can only be used with **MSX2** computers and upper.



This icon indicates the function can only be used with **MSX2+** computers and upper.



This icon indicates the function can only be used with **MSX TURBO-R**.



This icon indicates the function works with **MSX-DOS** or **MSX-DOS 2**



This icon indicates the function works with **MSX-DOS2** only



This icon indicates the function has been introduced or corrected from version **1.1** of FUSION-C



This icon indicates the function has been introduced or corrected from version **1.2** of FUSION-C



This icon indicates the function has been introduced or corrected from version **1.3** of FUSION-C



This icon indicates an important change since FUSION-C's previous version, which may require adaptation on your part by updating or re-installation...

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

MSX FUSION

[msx_fusion.h]

This is the main part of the library. You should always include msx_fusion.h in your programs, it provides all basic components for your projects.

Console Functions

<i>CheckBreak</i>	V1.2	CONSOLE
int CheckBreak (void) Checks the CTRL-BREAK in the MSX-DOS console. Return 0 if not pressed, or -1 if pressed		

<i>Getche</i>	CONSOLE
char Getche (void) Reads and display character from console. Returns the read character.	

<i>InputChar</i>	CONSOLE
char InputChar (void) Reads a character from console, and return it.	

<i>InputString</i>	CONSOLE
int InputString (char *Dest, int Len) Gets a string from console input and store it inside *dest pointer variable. - Dest : is a pointer where to store string - Len : is the maximum length of the string. User can enter len-2 chars. Max length is 253 chars Returns the length of the string	

<i>Locate</i>	CONSOLE
void Locate (int x, int y) Sets console cursor to X & Y coordinates.	

<i>PrintHex</i>	V1.2	CONSOLE
void PrintHex (unsigned int num) Prints the hexadecimal representation of the integer num , on the text screen mode.		

<i>PutCharHex</i>	V1.2	CONSOLE
void PutCharHex (char c) Prints the hexadecimal representation of the char num , on the text screen mode.		

Print**CONSOLE****void Print (char *text)**Prints **text* string on a text screen mode

Supports escape sequences :

- | | |
|------------|---|
| \a (0x07) | - Beep |
| \b (0x08) | - Backspace. Cursor left, wraps around to the previous line, stop at top left of the screen. |
| \t (0x09) | - Horizontal Tab. Tab, overwrites with spaces up to 8th next column, wraps around to start of next line, and scrolls at bottom right of screen. |
| \n (0x0A) | - Newline > Line Feed and Carriage Return (CRLF) Note: CR added in this Lib. |
| \v (0x0B) | - Cursor home. Place the cursor at the top screen. |
| \f (0x0C) | - Form feed. Clear screen and place the cursor at top. |
| \r (0x0D) | - CR (Carriage Return) |
| \\" (0x22) | - Double quotation mark |
| \' (0x27) | - Single quotation mark |
| \? (0x3F) | - Question mark |
| \\" (0x5C) | - Backslash |

PrintNumber**CONSOLE****void PrintNumber (unsigned int num)**Prints the *num* number supplied in parameter to a text screen (console).**PrintFNumber****CONSOLE****void PrintFNumber (unsigned int num, char emptyChar, char length)**Prints a number *num* to a text screen mode with formatting parameters*emptyChar* : (32=' ', 48='0', etc.)*length* : 1 to 5**PrintChar****CONSOLE****void PrintChar (char c)**Prints the character *c* to console (or to a text screen mode)**PrintDec****CONSOLE****void PrintDec (int num)**Prints signed integer *num* from -32768 to 32767 to a text screen mode

printf**CONSOLE**

```
int printf (const char *fmt, ...)
```

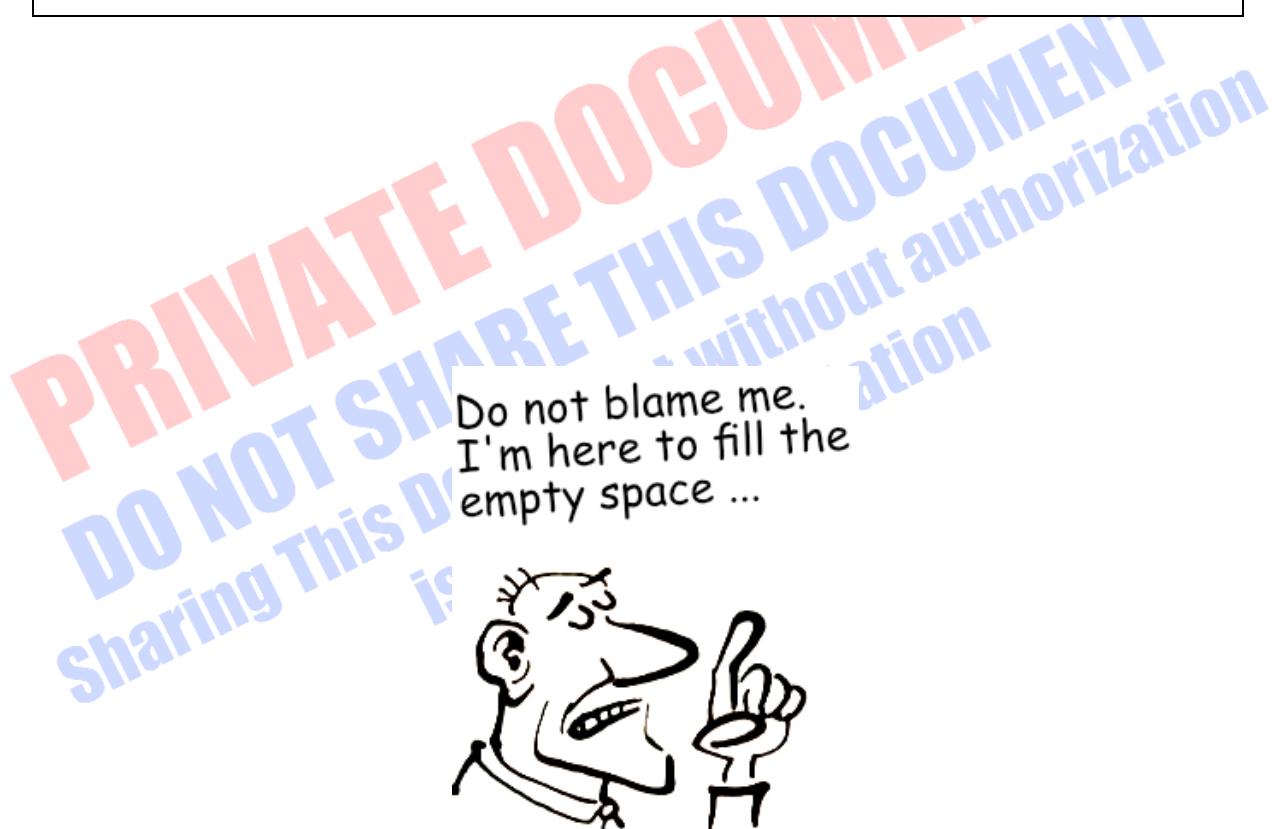
Prints formatted text data to console (text Screen mode).

Example :

```
int nb=19;
char src[4]="bag";
printf("\n\r I have %d lollipops in my %s ",nb,src);
```

Supported format specifiers:

%d or %i	: signed int
%u	: unsigned int
%x	: hexadecimal int
%c	: character
%s	: string
%%	: a % character
%l	: signed long
%ul	: unsigned long
%lx	: hexadecimal long



C Library for MSX-DOS with SDCC compiler
Miscellaneous Functions

<i>Cl</i>	<i>MISCELLANEOUS</i>
<code>void Cls (void)</code>	
Clears console or any screen mode	

<i>KeySound</i>	<i>MISCELLANEOUS</i>
<code>void KeySound (char n)</code>	
Enables or disables Key Sound.	
<i>n</i> value must be:	
0 : To Disable Key Sound	
1 : To Enable Key Sound	

<i>FunctionKeys</i>	<i>MISCELLANEOUS</i>
<code>void FunctionKeys (char n)</code>	
Shows or hides Function Keys on a Basic text screen mode.	
<i>n</i> value must be:	
0 : To Disable Function keys	
1 : To Enable Function keys	

<i>ChangeCap</i>	<i>MISCELLANEOUS</i>
<code>void ChangeCap (char n)</code>	
Changes the state of the Cap Led.	
<i>n</i> value must be:	
0 : To Disable Cap Led	
1 : To Enable Cap Led	

<i>ReadMSXtype</i>	<i>MISCELLANEOUS</i>
<code>char ReadMSXtype (void)</code>	
Reads and returns the MSX type.	
Returned values:	
0 : MSX 1	
1 : MSX 2	
2 : MSX2+	
3 : MSX Turbo-r	

<i>ReadKeyboardType</i>	V1.3	<i>MISCELLANEOUS</i>
char <i>ReadKeyboardType</i> (void)		
Reads and returns the MSX keyboard type .		
Returned values:		
0 : Japanese		
1 : International		
2 : French		
3 : UK		
4 : German		

<i>Screen</i>	<i>MISCELLANEOUS</i>
void <i>Screen</i> (char mode)	
Sets display to specified screen mode.	
<i>mode</i> can be a valid screen mode number, between 0 and 8	

<i>Beep</i>	<i>MISCELLANEOUS</i>
void <i>Beep</i> (void)	
Plays a beep sound.	

<i>RealTimer</i>	V1.1	<i>MISCELLANEOUS</i>
unsigned int <i>RealTimer</i>(void);		
Reads and returns the real clock timer of the MSX computer.		
Timer is increased by 1 on each VDP complete screen draw. The refresh rate depends of the MSX computer, it can be 50 Hz on european's computers (PAL), or 60 Hz on Japanese computers (NTSC).		
On MSX2, 2+ and MSX-Turbo-r the screen's refresh rate can be adjusted manually with <i>VDP50Hz()</i> and <i>VDP60Hz()</i> commands.		

<i>SetRealTimer</i>	V1.1	<i>MISCELLANEOUS</i>
void <i>SetRealTimer</i> (unsigned int <i>value</i>);		
Sets the Real Clock timer of the MSX computer to a specific <i>value</i> between 0 and 65535 .		

<i>CovoxPlayVram</i>	V1.1 MSX2	<i>MISCELLANEOUS</i>
void <i>CovoxPlayVram</i> (char <i>Page</i>, unsigned int <i>StartAddress</i>, unsigned int <i>Length</i>, char <i>Speed</i>)		
Plays 8bits PCM audio stored in VRAM thru Covox/simpl module. The PCM audio sample must be stored in the MSX Vram. With a MSX2 you can store, up to 128KB of Sample if you are using all the VRAM memory.		
<i>Page</i> is the VRAM page where the sample is stored		
<i>StartAddress</i> represents the first VRAM address of the Sample		
<i>Length</i> is the length of byte you want to play		
<i>Speed</i> , represents the playing speed. More this number is high, more the playing is slow.		

CovoxPlayRam**V1.3****MISCELLANEOUS****void CovoxPlayRam (void StartAddress, unsigned int Length, char Speed)**

Plays 8bits PCM audio stored in RAM thru Covox/simpl module. The PCM audio sample must be stored in the MSX RAM.

StartAddress is the address in RAM where the sample is stored

Length is the length of byte you want to play

Lspeed, represents the playing speed. More this number is high, more the playing is slow.

RleWBToRam**V1.1****MISCELLANEOUS****void RleWBToRam (unsigned int *RamSource, unsigned int *RamDest)**

Decompress RLEWB data to Ram.

**RamSource* is the pointer address where RLEWB data are stored in RAM

**RamDest* is the pointer address you want to put uncompressed data n RAM

Example: RleWBToRam(&RleData[0],&dest[0]);

Note : The RLEWB Compressor command line tool is provided in the "Tools" folder.

RleWBToVram**V1.1****MISCELLANEOUS****void RleWBToVram (unsigned int *RamAddress, unsigned int VramAddress);**

Decompress RLEWB data directly to Vram.

**RamAddress* is the pointer address where RLEWB data are stored in RAM

VramAddress is the address where you want to start to put uncompressed data, this address must be between 0 and 65535. If you want to uncompress to another VRAM Page, use the **SetActivePage** function to set the VRAM page.

Example : RleWBToVram (&rleddata[0],0);

Note : The RLEWB Compressor command line tool is provided in the "Tools" folder.

PatternRotation**V1.3****MISCELLANEOUS****void PatternRotation (unsigned int *Pattern, unsigned int *buffer, char rotation)**

Rotates the 8x8 pixels sprite pattern. **Pattern* is the address of the 8x8 pixels source pattern, **buffer* is the address of a 8 bytes buffer where the rotated pattern will be stored.

Rotation must be

0: for a 90° right rotation

1: for a 90° left rotation

PatternHFlip**V1.3****MISCELLANEOUS****void PatternHFlip (unsigned int *Pattern, unsigned int *buffer)**

Flips horizontally a 8x8 pixels pattern. **Pattern* is the address of the 8x8 pixels source pattern, **buffer* is the address of a 8 bytes buffer where the flipped pattern will be stored.

<i>PatternVFlip</i>	V1.3	<i>MISCELLANEOUS</i>
void PatternVFlip (unsigned int *Pattern, unsigned int *buffer) Flips vertically a 8x8 pixels pattern. *Pattern is the address of the 8x8 pixels source pattern, *buffer is the address of a 8 bytes buffer where the flipped pattern will be stored.		

<i>TurboMode</i>	V1.3	<i>MISCELLANEOUS</i>
void TurboMode (char mode) Enables the Turbo Mode of the MSX2+ Panasonic FS-A1WSX, FS-A1WX, FS-A1FX. The turbo mode is the possibility to run the Z80 at 5.37 Mhz instead of 3.57Mhz mode must be: 1 : to activate the turbo mode. 0 : to deactivate the turbo mode		

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

JoystickRead

JOYSTICK

char JoystickRead (char joyNumber)

Reads and returns state of a joystick.

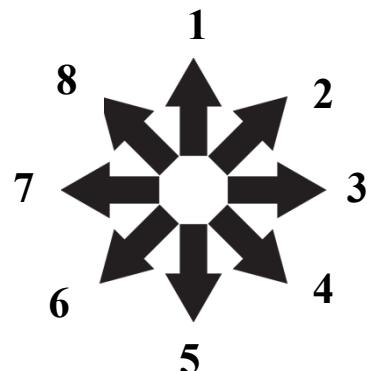
JoyNumber is the joystick you want to read, it must be :

0 : Keyboard's Arrow keys.

1 : Joystick port 1

2 : Joystick port 2

Returned values:



0=inactive

TriggerRead

JOYSTICK

char TriggerRead (char TriggerNumber)

Reads and returns state of a joystick button.

TriggerNumber is the button you want to read, it must be :

0 – space key.

1 – button 1 joystick port A

2 – button 1 joystick port B

3 – button 2 joystick port A

4 – button 2 joystick port B

Note, there is no second button for the keyboard.

Returned values:

0 if button is inactive,

255 if button is pressed

JoystickReadTo**V1.3****JOYSTICK****void JoystickReadTo (JOY_DATA *jd)**

Reads and returns state of a joystick in a faster way than **JoystickRead**.

This function directly queries the PSG port which controls the joysticks, also it checks Fire Buttons and direction at the same time.

The function is using this pre-defined structure to return the read values.

```
typedef struct {
    char    joyport;
    char    up;
    char    down;
    char    left;
    char    right;
    char    button1;
    char    button2;
    char    global;
} JOY_DATA;
```

You must declare this structure before using this function, like this :

```
static JOY_DATA jd;
```

Returned values goes to the structure variables. According to the previous structure declaration, you will receive datas inside

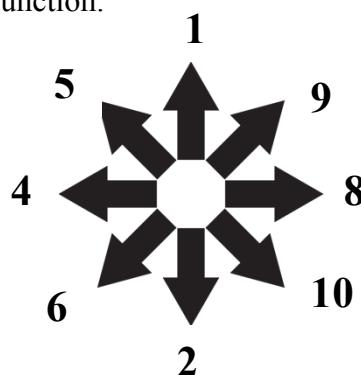
<i>jd.up</i>	= 1 if joystick goes up
<i>jd.down</i>	= 1 if joystick goes down
<i>jd.left</i>	= 1 if joystick goes left
<i>jd.right</i>	= 1 if joystick goes right
<i>jd.button1</i>	= 1 if button is pressed
<i>jd.button2</i>	= 1 if button is pressed
<i>jd.global</i>	= the global value see below

Note, *jd.joyport* is an input variable. It says to the routine which joystick port to read. The value must be set before calling the function.

jd.joyport must be **1** if you want to read joystick port n°1, and **2** if you want to read joystick port n°2.

There are two ways to use the function inside your own program. You can choose to use the **left**, **right**, **up**, **down** variables, or use the global value returned in *jd.global*. In this last case, it's most like the **JoystickRead** function.

jd.global returned values are:



0=inactive

Note, you can also check impossible moves, like UP + DOWN or LEFT + RIGHT etc. By analyzing the binary value of *jd.global*

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	RIGHT	LEFT	DOWN	UP

MouseRead**V1.1****MOUSE****unsigned int MouseRead (int MousePort)**

Reads and returns the mouse offsets of the mouse connected at MousePort.

The function returns the X offset and the Y offset into a 16-bit value.

Decoding this value must be done like this:

```
Mouse_offset=MouseRead(MousePort2);
```

```
Xoffset=Mouse_offset >> 8;
```

```
Yoffset=Mouse_offset & 0xFF;
```

MousePort is referring to joystick port 1 or port 2, it is defined as a predefined macro, use only : “**MousePort1**” or “**MousePort2**” as parameters.

Once you have decoded *X offset* and *Y offset*, you can move a sprite object over the screen like this :

```
mx=mx-Xoffset;
```

```
my=my-Yoffset;
```

```
PuSprite(1,2,mx,my,15);
```

Note : If **MouseRead** returns **65535** as offset, this means no mouse is connected .

MouseReadTo**V1.2****MOUSE****void MouseReadTo (char MousePort, MOUSE_DATA *md)**

Reads and returns the mouse offsets, and the 2 buttons states of the mouse connected in MousePort.

The function is using this pre-defined structure to return the read values.

```
typedef struct {
    signed char dx;
    signed char dy;
    char lbutton;
    char rbutton;
} MOUSE_DATA;
```

You must declare this structure before using this function, like this :

```
static MOUSE_DATA md;
```

MousePort must be 1 or 2 depending on the mouse port you want to read.

Returned values goes to the structure variables. According to the previous structure declaration, you will receive datas inside

```
md.dx
md.dy
md.lbutton
md.rbutton
```

lbutton and **rbutton** are returning 0 when pressed

Code example :

```
MouseReadTo(1,&mb)
```

Keyboard Functions***GetKeyMatrix******KEYBOARD*****char GetKeyMatrix (char line)**

Returns the value of the specified *line* from the keyboard matrix. Each line provides the status of 8 keys. The state of the key returned is by default:

1 = not pressed.**0** = pressed

bit	7	6	5	4	3	2	1	0
0	7 &	6 ^	5 %	4 \$	3 #	2 @	1 !	0)
1	;	:] }	[{ \	= +	- _	9 (8 *	
2	B	A	acent / ?	.	> , <	`	' "	
3	J	I	H	G	F	E	D	C
4	R	Q	P	O	N	M	L	K
5	Z	Y	X	W	V	U	T	S
6	F3	F2	F1	CODE	CAPS	GRAPH	CTRL	SHIFT
7	RET	SEL	BS	STOP	TAB	ESC	F5	F4
8	Right	Down	Up	Left	DEL	INS	HOME	SPACE
9	NUM4	NUM3	NUM2	NUM1	NUM0	NUM/	NUM+	NUM*
10	NUM.	NUM,	NUM-	NUM9	NUM8	NUM7	NUM6	NUM5

The picture above show the international key matrix. Please check *The MSX Keyboard* chapter to know more about other key matrix.

Inkey***KEYBOARD*****char Inkey (void)**

Checks keyboard for a key pressed. Returns the ASCII code of the key, or **0** if no key was pressed.

KillKeyBuffer***KEYBOARD*****void KillKeyBuffer (void)**

Clears the key buffer.

WaitKey***KEYBOARD*****char WaitKey (void)**

Waits for a key pressed. Returns the ASCII code of the key.

Rkeys		V1.5	KEYBOARD														
char Rkeys (void)																	
Returns the state of line 8 of the Key matrix into a byte. (Arrows keys support.) If a bit is active (1) the associated key is pressed. It can report 2 keys pressed at the same time.																	
bit	7	6	5	4	3	2	1	0									
Key	RIGHT	DOWN	UP	LEFT	DEL	INS	HOME	SPACE									
The easy way to check one key pressed is to refer to the returned value.																	
1 : if SPACE is pressed																	
2 : if HOME is pressed																	
4 : if INS is pressed																	
8 : if DEL is pressed																	
16 : if LEFT is pressed																	
32 : if UP is pressed																	
64 : if DOWN is pressed																	
128 : if RIGHT is pressed																	
(note : this method cannot be used to check if several keys are pressed at the same time, to do that, analyse the returned byte value)																	

Fkeys		V1.5	KEYBOARD														
char Fkeys (void)																	
Returns the state of line 8 of the Key matrix into a byte. (Function keys support.) If a bit is active (1) the associated key is pressed. It can report 2 keys pressed at the same time.																	
bit	7	6	5	4	3	2	1	0									
Key	STOP	GRAPH	ESC	F5	F4	F3	F2	F1									
The easy way to check one key pressed is to refer to the returned value.																	
1 : if F1 is pressed																	
2 : if F2 is pressed																	
4 : if F3 is pressed																	
8 : if F4 is pressed																	
16 : if F5 is pressed																	
32 : if ESC is pressed																	
64 : if GRAPH is pressed																	
128 : if STOP is pressed																	
(note : this method cannot be used to check if several keys are pressed at the same time, to do that, analyse the returned byte value)																	

I/O Port Functions

<i>OutPort</i>	<i>I/O PORT</i>
<code>void OutPort (char port, char data)</code> Sends a 8-bit <i>data</i> value to a MSX <i>port</i>	

<i>InPort</i>	<i>I/O PORT</i>
<code>char InPort (char port)</code> Reads from a MSX <i>port</i>	

<i>OutPorts</i>	<i>I/O PORT</i>
<code>void OutPorts (char port, char *p_data, char count)</code> Sends <i>*p_data</i> to a MSX <i>port</i> .	

PRIVATE DOCUMENT!
DO NOT SHARE THIS DOCUMENT!
Sharing This Document without authorization
is copyright violation

VDP Functions***VDPstatus******VDP*****char *VDPstatus* (char *vdpreg*)**

Reads VDP Status register *vdpreg* and returns the value. The Interrupt is disabled before reading, and enabled after reading.

VDPstatusNi***VDP*****char *VDPstatusNi* (char *vdpreg*)**

Reads VDP Status register *vdpreg* and returns the value. The Interrupt is not modified by this function.

VDPwriteNi***VDP*****void *VDPwriteNi* (char *vdpreg*, char *data*)**

Writes *data* to the VDP register *vdpreg*. The Interrupt is not modified by this function.

VDPwrite***VDP*****void *VDPwrite* (char *vdpreg*, char *data*)**

Writes *data* to the VDP register *vdpreg*. The Interrupt is disabled before writing, and enabled after writing.

IsVsync***VDP*****char *IsVsync* (void)**

Check the VDP and returns the Vblank state.

Return **1**, if true.

IsHsync***VDP*****char *IsHsync* (void)**

Checks the VDP and returns HSynch state.

Return **1** if true.

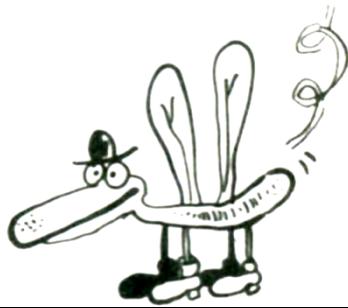
Vsynch***V1.3******VDP*****char *Vsynch* (void)**

Same effect as the *IsVsync()* function, but is much faster because it does not read the state from the VDP register. The function returns **0** while VDP raster is sending the image to the screen.

Vpeek***VDP*****Char *Vpeek* (unsigned int *address*)**

Reads and returns a byte from Vram Memory *address*

Vpoke**VDP****void Vpoke (unsigned int address, char data)**Writes the byte **data** in Vram at Memory **address****VpokeFirst****VDP****void VpokeFirst (unsigned int address)**Sets first Vram **address** for multiple Vpokes. Use **VpokeNext** after this instruction.**VpokeNext****VDP****void VpokeNext (char data)**Writes the byte **data** to the Vram Memory address sets by **VpokeFirst** and increments address for the next use of this instruction.**VpeekFirst****VDP****char VpeekFirst (unsigned int address)**Sets first **address** for multiple Vpeek. Use **VpeekNext** after this instruction.**VpeekNext****VDP****char VpeekNext (void)**Reads and returns a byte from the Vram Memory address set by **VpeekFirst**, and increments address for the next use of **VpeekNext** instruction**Width****VDP****void Width (char n)**Sets the width of a text screen mode, **n** must be between **1** and **80****SetColors****VDP****void SetColors(char ForeCol, char BackgrCol, char BorderCol)**Sets, foreground **ForeCol** color, background **BackgrCol** color and border color **BorderCol**. Supports MSX2's screen mode like Screen8 where colors can be a number between 0 and 255.**SetColor****v1.3****VDP****void SetColor(char ForeCol)**Sets only the foreground with **ForeCol** color. Useful for function that do not have a color parameter, like **PutText**.**SetBorderColor****VDP****void SetBorderColor(char BorderCol)**Sets only the screen border color **BorderCol** of the screen. It uses the same colors as **SetColors** function.

***SetColorPalette*****MSX2****VDP**

```
void SetColorPalette (char ColorNumber, char Red, char Green, char Blue)
```

Sets a screen color with new RGB parameters.

ColorNumber : is the color you want to modify (Between 1 and 15)

Red, **Green**, and **Blue** parameters are levels of Red, Green and Blue you want to assign to this color. These parameters are a number between 0 and 7

Example :

```
SetColorPalette( 15,7,0,0);
```

This sets the color number 15 to a pure Red color

SetPalette**MSX2****VDP**

```
void SetPalette ((Palette *) mypalette)
```

Sets the screen color palette with the new « **mypalette** » structure data.

The predefined « Palette Structure » is composed of 16 lines of 4 values : **N**, **R**, **G**, **B**. **N** is the number of the color (0 ... 15). **R**, **G**, and **B** are the level of Red, Green or Blue in the final color. **R**, **G** and **B** must be between 0 and 7.

Example:

```
char mypalette[] = {
    0, 0,0,0,
    1, 2,1,1,
    2, 6,5,4,
    3, 5,4,3,
    4, 5,5,3,
    5, 6,5,3,
    6, 7,6,4,
    7, 3,2,1,
    8, 7,5,2,
    9, 6,4,2,
    10, 4,3,2,
    11, 6,0,1,
    12, 5,3,2,
    13, 3,3,2,
    14, 3,1,0,
    15, 6,6,6};
```

```
SetPalette((Palette *) mypalette);
```

SetTransparent**MSX2****VDP**

```
void SetTransparent (char n )
```

By default color #0 is used as a transparent color (n=0). This function disables the transparent mode, and give you the possibility to redefine this color #0 in the same way as other colors of the palette, for example with the SetColorPalette function.

The parameter **n**, can be 0 to enable transparent color, or 1 to disable the transparent mode.
example :

<pre>SetTransparent (1);</pre>	// Disable color #0 transparent mode
<pre>SetColorPalette (0, 2,6,7);</pre>	// Redefine color #0 with new RGB Values

RestorePalette		MSX2	VDP
void RestorePalette (void)			
Restore the MSX default Color Palette.			
MSX Default color palette :			
00	TRANSPARENT	R G B	0. 0. 0.
01	BLACK		0. 0. 0.
02	MEDIUM_GREEN		1. 6. 1.
03	LIGHT_GREEN		3. 7. 3.
04	DARK_BLUE		1. 1. 7.
05	LIGHT_BLUE		2. 3. 7.
06	DARK_RED		5. 1. 1.
07	CYAN		2. 6. 7.
08	MEDIUM_RED		7. 1. 1.
09	LIGHT_RED		7. 3. 3.
10	DARK_YELLOW		6. 6. 1.
11	LIGHT_YELLOW		6. 6. 4.
12	DARK_GREEN		1. 4. 1.
13	MAGENTA		6. 2. 5.
14	GRAY		5. 5. 5.
15	WHITE		7. 7. 7.

SetDisplayPage		MSX2	VDP
void SetDisplayPage (char page)			
Sets the VRAM <i>page</i> displayed to the screen.			
Parameter <i>page</i> can be 0, 1, 2, 3 ; it depends of the screen mode.			

SetActivePage		MSX2	VDP
void SetActivePage (char page)			
Sets the VRAM <i>page</i> of the that receive modifications and instructions.			
Parameter <i>page</i> can be 0, 1, 2, 3 ; it depends of the screen mode.			

SetScrollH		MSX2	VDP
void SetScrollH (int n)			
Uses Hardware horizontal screen scrolling of the MSX2+ and MSX Turbo-R. <i>n</i> is the number of pixels you want to scroll to, it can be positive or negative.			
With screen modes 6 and 7, <i>n</i> must be a multiple of 2.			

SetScrollV		MSX2	VDP
void SetScrollV (char n)			
Uses Hardware vertical screen scrolling of the MSX2, MSX2+ and MSX Turbo-R.			
<i>n</i> is the number of pixels you want to scroll to, it can be positive or negative.			
With screen modes 6 and 7, <i>n</i> must be a multiple of 2			

SetScrollMask	V1.3 MSX2	VDP
void SetScrollMask (char n)		
When using the MSX2+ / Turbo-R hardware horizontal scrolling capability, you can enable or disable a 8-pixel column's mask at the left of the screen.		
Whith this mask enable the scrolling appears much more fluent. The parameter n , can be 1 to enable the mask or 0 to disable the mask.		

SetScrollDouble	V1.3 MSX2	VDP
void SetScrollDouble (char n)		
When using the MSX2+ / Turbo-R hardware horizontal scrolling capability, you have the possibility to use the 2 screen's pages as one and only horizontal screen. This means the second VRAM page is placed at the right of the first page ; thus you have a 512 x 256 pixels image to scroll on the horizontal axis.		
The parameter n , can be 1 to enable the double screen mode or 0 to disable this mode.		

HideDisplay		VDP
void HideDisplay (void)		
Hides the screen display (Black screen)		

ShowDisplay		VDP
void ShowDisplay (void)		
Shows the screen display if it was previously hidden.		

FillVram		VDP
void FillVram (unsigned int Startaddress, char value, unsigned int length)		
Fills the Vram from Startaddress with length bytes. The Startaddress is limited to a 16-bit address (from 0x0 to 0xFFFF), thus is you need to fill the MSX2's VRAM over 0xFFFF use the SetActivePage before.		

PutText	V1.1	VDP
void PutText (int X, int Y, char *str, char LogOp)		
Prints the text string *str to graphic screen (modes 3 to 8) at position X, Y		
LogOp represents the Logical Operation used when printing text. It can be used to print text with a transparent background. (See possible values in MSX2 Graphics Chapter)		
The Y coordinate is limited to an 18-bit number (from 0x0 to 0xFF), thus is you need to put text over 255 (On other VRAM page) use the SetActivePage before.		

VDP50Hz	MSX2	VDP
void VDP50Hz (void)		
Switches the MSX2 VDP to 50 Hz Pal Mode		

VDP60Hz	MSX2	VDP
void VDP60Hz (void)		
Switches the MSX2 VDP to 60 Hz NTSC Mode		

VDPLineSwitch	MSX2 V1.1	VDP
void VDPLineSwitch (void)		
Switches the MSX2 VDP Chip to 192 or 212 vertical lines. By default MSX2 screen modes 5 to 8 have 212 pixels, this function force the VDP to 192 lines.		

CopyRamToVram	V1.1	VDP
void CopyRamToVram(void *SrcRamAddress, unsigned int DestVramAddress, unsigned int Length)		
Copy a Ram Memory Block to a Vram Address. - *SrcRamAddress , must point to a Memory address or variable. - DestVramAddress , must be a valid VRAM Address - Length , is the length of the Memory Block to Copy.		

CopyVramToRam	V1.1	VDP
void CopyVramToRam(unsigned int SrcVramAddress, void *DestRamAddress, unsigned int Length)		
Copy a Vram Memory Block to a Ram Address. - SrcVramAddress , must be a valid VRAM Address - *DestRamAddress , must point to a memory address, or variable. - Length , is the length of the Memory Block to Copy.		

GetVramSize	V1.1	VDP
Char GetVramSize (void)		
Returns the Vram size of the MSX Computer. Returned values can be: 16, 64, 128 .		

SetVDPwrite	V1.2	VDP
void SetVDPwrite (unsigned int Adress)		
Sets the MSX VDP in writing mode. Function is MSX1 and MSX2 VDP compliant. This command is for experts & specific use. You do'nt need it, until you want to write your own graphic routines.		

SetVDPread	V1.2	VDP
void SetVDPread (unsigned int Adress)		
Sets the MSX VDP for reading process. MSX1 and MSX2 VDP compliant. You may need this function if you want to write your own graphic routines.		

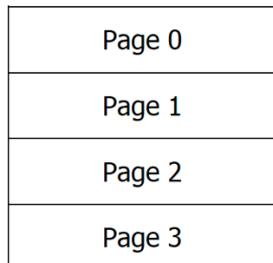
SetExpandVDPcmd	V1.3 MSX2+	VDP
void SetExpandVDPcmd (char n)		
This function enable the capability of the MSX2+ VDP to use the VDP commands with MSX1's screen modes 0 to 4 . (<i>See all VDP commands in the dedicated chapter.</i>) The parameter n , can be 1 to enable or 0 to disable this capability.		

void VDPAlternate(char oddpage, char OnTime, char OffTime)

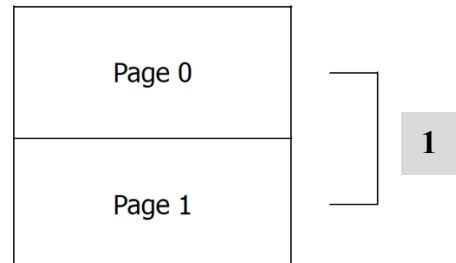
Activates the alternate video mode. It's a special feature of the MSX2 VDP.

Two graphic pages are alternately displayed on the screen at a specific speed.

Screen mode 5 or 6



Screen mode 7 or 8



The ***odd page*** parameter indicates which pages will be alternately displayed. With screen mode 5 or 6, specify **1** to alternately display pages 0 and 1 or specify **3** if you want to alternately display pages 2 and 3.

With screen mode 7 or 8 (up to 12 on MSX2+ and Turbo-r), as there are only 2 pages available, you must specify **1** in the ***odd page*** parameter.

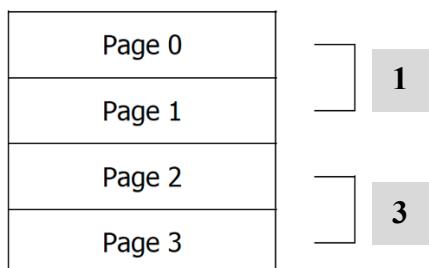
The ***OnTime & OffTime*** parameters specify the display time for each of the page, it can vary from 0 to 15, according to this array :

speed	VDP 50hz	VDP 60hz	speed	VDP 50hz	VDP 60hz	speed	VDP 50hz	VDP 60hz
0	0 ms	0 ms	6	1200ms	1001ms	12	2400ms	2002ms
1	200ms	166ms	7	1400ms	1168ms	13	2600ms	2169ms
2	400ms	333ms	8	1600ms	1335ms	14	2800ms	2336ms
3	600ms	500ms	9	1800ms	1509ms	15	3000ms	2503ms
4	800ms	667ms	10	2000ms	1668ms			
5	1000ms	834ms	11	2200ms	1835ms			

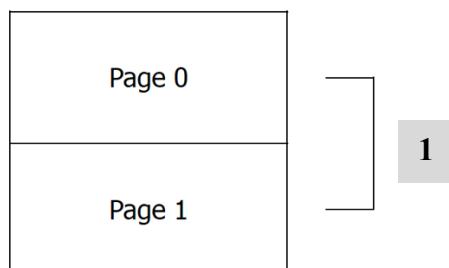
VDPinterlace**v1.3 MSX2****VDP****void VDPinterlace(char oddpage)**

Activates the interlace video mode. It's a special feature of the MSX2 VDP. It allows doubling the vertical resolution of the screen by alternatively displaying 2 graphic pages on the screen. Each pixel line is divided by two. One half of the even page, the other half on the odd page. Depending of the graphic mode used, the graphic games to be used are different.

Screen mode 5 or 6



Screen mode 7 or 8



The **odd page** parameter indicates which pages will be alternately displayed. With screen mode 5 or 6, specify **1** to alternately display pages 0 and 1 or specify **3** if you want to alternately display pages 2 and 3.

With screen mode 7 or 8 (up to 12 on MSX2+ and Turbo-R), as there are only 2 pages available, you do not have choice, you must specify **1** in the **odd page** parameter.

SetScreen10**v1.3 MSX2****VDP****void SetScreen10 (char n)**

When using the MSX2+ / Turbo-R the Screen 10 mode can show up to 12499 colors on the screen. This screen mode is the same as the screen mode 8 but with a different way to encode colors. Thus to use this mode activate first screen mode 8 with the instruction : **Screen(8)**, then, activate the screen 10 mode.

The parameter **n**, can be **1** to enable the screen mode or **0** to disable this mode.

example :

```
Screen(8);           // Use Screen mode 8 as basis
SetScreen10 (1);    // Now activating Screen mode 10
```

SetScreen12**v1.3 MSX2****VDP****void SetScreen12 (char n)**

When using the MSX2+ / Turbo-R the Screen 10 mode can show up to 19268 colors on the screen. This screen mode is the same as the screen mode 8 but with a different way to encode colors. Thus to use this mode activate first screen mode 8 with the instruction : **Screen(8)**, then, activate the screen 12 mode.

The parameter **n**, can be **1** to enable the screen mode or **0** to disable this mode.

example :

```
Screen(8);           // Use Screen mode 8 as basis
SetScreen12 (1);    // Now activating Screen mode 12
```

SetAdjust**V1.3 MSX2****VDP****void SetAdjust(signed char x, signed char y)**

Adjusts the MSX's screen to center the output image.

x is the screen horizontal offset, it can vary from -8 to +7*y* is the screen vertical offset, it can vary from -8 to +7***SaveScreenBoot*****V1.3 MSX2****VDP****void SaveScreenBoot(char Nb)**

Used to store a series of parameters which will be used each time the MSX is started.

-1 : X Screen Offset set with *SetAdjust*-2 : Y Screen Offset set with *SetAdjust*-3 : Screen Foreground Color set with *SetColor* or *SetColors*-4 : Screen Background Color set with *SetColor*-5 : Screen Border Color set with *Setcolors* or *BoderColor*-6 : Default Boot Screen mode, set with *Screen*. Can be Screen mode 0 or 1.-7 : Default number of columns of the default boot screen set with *Width*

Parameters that are saved are the current screen settings.

Nb is the number of parameters you want to save. From 1 to 8. If *Nb* = 2 then only the first two parameters (X and Y screen offset) will be saved.*Note* : The MSX2 RTC's SRAM must be powered by battery to keep theses parameters saved in the MSX's memory when the computer is to switch off.

PRIVATE
 DO NOT SHARE THIS
 Sharing This Document without authorisation
 is copyright violation

Type Functions

<i>IsAlphaNum</i>	<i>TYPE</i>
<code>char IsAlphaNum (char c)</code>	
Checks if the supplied value <i>c</i> is alpha or numerical : A..Z or a..z or 0..9	
Returned values:	
1 if true	
0 if false	

<i>IsAlpha</i>	<i>TYPE</i>
<code>char IsAlpha (char c)</code>	
Checks if the supplied value <i>c</i> is alpha : A..Z or a..z	
Returned values:	
1 if true	
0 if false	

<i>IsAscii</i>	<i>TYPE</i>
<code>char isAscii (char c)</code>	
Check if the supplied value <i>c</i> is an ASCII character : - !..~	
Returned values:	
1 if true	
0 if false	

<i>IsCtrl</i>	<i>TYPE</i>
<code>char IsCtrl (char c)</code>	
Checks if the supplied value <i>c</i> is a control symbol.	
Returned values:	
1 if true	
0 if false	

<i>IsDigit</i>	<i>TYPE</i>
<code>char IsDigit (char c)</code>	
Checks if the supplied value <i>c</i> is a digit: (0..9).	
Returned values:	
1 if true	
0 if false	

<i>IsGraph</i>	<i>TYPE</i>
<code>char IsGraph (char c)</code>	
Checks if the supplied value <i>c</i> is a graph representation.	
Returned values:	
1 if true	
0 if false	

<i>IsLower</i>	<i>TYPE</i>
char IsLower (char c) Checks if the supplied value <i>c</i> is lower-case. Returned values: 1 if true 0 if false	

<i>IsUpper</i>	<i>TYPE</i>
char IsUpper (char c) Checks if the supplied value <i>c</i> is upper-case. Returned values: 1 if true 0 if false	

<i>IsPrintable</i>	<i>TYPE</i>
char IsPrintable (char c) Checks if the supplied value <i>c</i> is printable. Returned values: 1 if true 0 if false	

<i>IsPunctuation</i>	<i>TYPE</i>
char IsPunctuation (char c) Checks if the supplied value <i>c</i> is a punctuation sign. Returned values: 1 if true 0 if false	

<i>IsSpace</i>	<i>TYPE</i>
char IsSpace (char c) Checks if the supplied value <i>c</i> is a space. Returned values: 1 if true 0 if false	

<i>IsHexDigit</i>	<i>TYPE</i>
char IsHexDigit (char c) Checks if the supplied value <i>c</i> is a hexadecimal digit. Returned values: 1 if true 0 if false	

<i>IsPositive</i>	<i>TYPE</i>
Int IsPositive (int c) Checks if the supplied value <i>c</i> is positive. Returned values: -1 if negative 1 if positive 0 if value is null	

<i>IntToFloat</i>	<i>TYPE</i>
float IntToFloat (int c) Returns a float value of the supplied value <i>c</i>	

<i>IntSwap</i>	<i>TYPE</i>
void IntSwap (int *a, int *b) Swaps the content two Integer variables <i>a</i> and <i>b</i>	

String Functions

CharToLower		STRING
char CharToLower (char <i>ch</i>)		
Returns the lower case version of the char value <i>ch</i>		
CharToUpper		STRING
char CharToUpper (char <i>ch</i>)		
Returns the upper-case version of the char value <i>ch</i>		
StrCopy		STRING
void StrCopy (char * <i>dst</i> , char * <i>src</i>)		
Copy string from * <i>src</i> to * <i>dst</i>		
NStrCopy		STRING
void NStrCopy (char * <i>dst</i> , char * <i>src</i> , int <i>n</i>)		
Copy string from * <i>src</i> to * <i>dst</i> with no more than <i>n</i> characters.		
StrConcat		STRING
void StrConcat (char * <i>dst</i> , char * <i>src</i>)		
Concatenates string * <i>src</i> at the end of * <i>dst</i>		
NStrConcat		STRING
void NStrConcat (char * <i>dst</i> , char * <i>src</i> , int <i>n</i>)		
Concatenates <i>n</i> characters from the string * <i>src</i> at the end of * <i>dst</i>		
StrLen		STRING
int StrLen (char * <i>string</i>)		
Returns length of the * <i>string</i>		
StrCompare		STRING
int StrCompare (char * <i>s1</i> , char * <i>s2</i>)		
Compares two strings * <i>s1</i> and * <i>s2</i> ,		
Returned values:		
-1 if (<i>s1</i> < <i>s2</i>)		
0 if (<i>s1</i> = <i>s2</i>)		
1 if (<i>s1</i> > <i>s2</i>)		

NStrCompare**STRING****Int NStrCompare (char *s1, char *s2, int n)**Compares the **n** first characters two strings ***s1** and ***s2**.

Returned values:

-1 if (**s1 < s2**)**0** if (**s1 = s2**)**1** if (**s1 > s2**)***StrChr*****STRING****int StrChr (char *string, char c)**Search for the char **c** inside ***string**.Returns **1** if **c** is found, otherwise returns **-1*****StrPosStr*****STRING****int StrPosStr (char *s1, char *s2)**Search for the string ***s2** inside string ***s1** and return position of the first character of ***s1**, or returns **-1** if not found.***StrSearch*****STRING****int StrSearch (char *s1, char *s2)**Returns the first occurrence of any character from ***s2** in the string ***s1**Returns **-1** if not found.***StrPosChr*****STRING****int StrPosChr (char *string, char c)**Returns the position of **c** inside ***string**, or **-1** if not found.***StrLeftTrim*****STRING****void StrLeftTrim (char *string)**Removes spaces inside the left part of ***string*****StrRightTrim*****STRING****void StrRightTrim (char *string)**Removes spaces in the right part of ***string*****StrReplaceChar*****STRING****void StrReplaceChar (char *string, char c, char new_c)**Replaces all chars **c** by **new_c** in string ***string**

StrReverse	V1.2	STRING
char* StrReverse (char *str)		
This function reverse the order of the chars inside the string *str The new string is returned as a string, chars * array.		

Itoa	V1.2	STRING
char* Itoa (int num, char* str, int base)		
This function convert an integer num to a string of chars *str . The new string is returned as a string of chars, and must be declared before using the function, with enough ram to recover the converted number. base indicates which base you want to convert to. It can be : 8 , 10 or 16 Code example: <code>char temp[6]; // add one more byte, to include the end of string int number; number=65535; Itoa (number,temp,10);</code>		

StrToLower	V1.3	STRING
void StrToLower (char *c);		
Converts the *c string to lower-case. Original string is replaced.		

StrToUpper	V1.3	STRING
void StrToUpper (char *c);		
Converts the *c string to upper case. Original string is replaced.		



Memory Functions**Poke****MEM****void Poke (unsigned int address, char data)**Writes a byte (8 bits) value **data** to Memory **address****Pokew****MEM****void Pokew (unsigned int address, unsigned int data)**Writes a 2 bytes value (16 bits) **data** to Memory **address** (2 Bytes).**Peek****MEM****Char Peek (unsigned int address)**Reads and returns a byte (8 bits) value from Memory **address****Peekw****MEM****Unsigned int Peekw (unsigned int address)**Reads and returns a 2 bytes value (16 bits) from Memory **address**. (Will peek 2 bytes).**MemChr****MEM****char* MemChr (char *adr, char c, int n)**Returns pointer to char in **n** bytes of **adr**, or NULL if not found.**MemFill (aka FillRam)****MEM****Void MemFill (char *adr, char c, unsigned int n)**Fills the Ram with a byte **c**, from **adr** to **adr+n****Memcpy****MEM****void MemCopy (char *dst, char *src, int n)**Copy **n** bytes from ***src** to ***dst****MemcpyReverse****MEM****void MemCopyReverse (char *dst, char *src, int n)**Copy **n** bytes from ***src** to ***dst**, but process will start at the end address (**src + n**) to the start address (**src**)**MemCompare****MEM****int MemCompare (char *s1, char *s2, int n)**Compares **n** bytes of ***s1** and ***s2**

Returned values:

-1 if (**s1 < s2**)**0** if (**s1 = s2**)**1** if (**s1 > s2**)

MMalloc**MEM****void *MMalloc (unsigned int size)**

SDCC version of malloc, memory right below the code (heap_top=length of program+few bytes) should be free of data or code loaded after at runtime
`char*heap_top ;`

ReadTPA**MEM****unsigned Int ReadTPA (void)**

Returns the TPA address of the current MSX-DOS running system.
(Only available for MSX-DOS)

ReadSP**MEM****unsigned Int ReadSP (void)**

Reads the SP register, and returns the current lower address of the Stack.
The Stack is inside the TPA zone. Stack is growing and decreasing while a program is running. In any case, the Stack address is the ultimate address you can use.

BitReturn**V1.3****MEM****char BitReturn (char nbit, char byte)**

Returns a bit from a byte. The bit number **nbit** is returned by the function.

BitReset**V1.3****MEM****void BitReset (char nbit, char *byte)**

Resets a bit inside a byte. The bit **nbit** is reseted inside ***byte**.

Interrupt Functions

<i>EnableInterrupt</i>	<i>INTERRUPT</i>
void EnableInterrupt (void)	
Enables the Interrupt base system. Maybe useful in some circumstances	

<i>DisableInterrupt</i>	<i>INTERRUPT</i>
void DisableInterrupt (void)	
Disables the Interrupt base system. Maybe useful in some circumstances.	

<i>Halt</i>	<i>INTERRUPT</i>
void Halt (void)	
Same as the Halt Asm Function. Wait for Interrupt.	

<i>Suspend</i>	<i>INTERRUPT</i>
void Suspend (void)	
Suspend Z80 and wait for next interrupt.	

<i>SetInterruptHandler</i>	<i>V1.1</i>	<i>INTERRUPT</i>
void SetInterruptHandler (char (*p_handler))		
Sets the function of your program the interrupt handler process will call at each interruption.		
The called function must be a function without any parameters, but it can make use of global variables.		
Sets the function of your program the interrupt handler process will call at each interruption.The called function must be a function without any parameters, but it can make use of global variables.		

<i>EndInterruptHandler</i>	<i>V1.1</i>	<i>INTERRUPT</i>
void EndInterruptHandler (void)		
End the Interruption handler initialized by the SetInterruptHandler .		
If your program returns to MSX-DOS, it's important to end the Interrupt handler properly. Call this function just before the Exit to MSX-DOS.		

<i>SetVDPIInterruptHandler</i>	<i>V1.3</i>	<i>INTERRUPT</i>
void SetVDPIInterruptHandler (char (*p_handler))		
This function is based on the VDP interruption (50 Hz or 60 Hz). Sets the function of your program the interrupt handler process will call at each interruption.		
The called function must be a function without any parameters, but it can make use of global variables.		
Sets the function of your program the interrupt handler process will call at each interruption.The called function must be a function without any parameters, but it can make use of global variables.		

void EndVDPIinterruptHandler (void)End the Interruption handler initialized by the **SetVDPIinterruptHandler**.

If your program returns to MSX-DOS, it's important to end the Interrupt handler properly.

Call this function just before the Exit to MSX-DOS.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

InitPSG**PSG****void InitPSG (void)**

Initialization of the PSG (use this function before sending data to PSG). All registers will be set to 0, and stops all noises and sounds.

PSGread**PSG****char PSGread (char psgreg)**

Read data from *psgreg* PSG register.

PSGwrite**PSG****void PSGwrite (char psgreg, char data)**

Writes data to *psgreg* PSG register.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler
MSX-DOS File I/O Functions

Predefined macros & structures

```

typedef struct {
    unsigned char     drive_no ;
    unsigned char     name[8] ;
    unsigned char     ext[3] ;
    unsigned int      current_block ;
    unsigned int      record_size ;
    unsigned long     file_size ;
    unsigned int      date ;
    unsigned int      time ;
    unsigned int      device_id ;
    unsigned char     directory_location ;
    unsigned int      start_cluster_no ;
    unsigned int      last_access_cluster_no ;
    unsigned int      cluster_offset ;
    unsigned int      current_record ;
    unsigned long     random_record ;
} FCB ;

typedef struct {
    unsigned char     name[8] ;
    unsigned char     ext[3] ;
    unsigned char     attribute ;
    unsigned char     undel_char ;
    unsigned char     reserve[9] ;
    unsigned int      time ;
    unsigned int      date ;
    unsigned int      start_cluster_no ;
    unsigned long     file_size ;
} FCB_DIR ;

typedef struct {
    unsigned char     drive_no ;
    FCB_DIR          dirinfo ;
} FCB_FIND ;

Returned code :
FCB_SUCCESS           0x00

Dir attributes :
FCB_DIR :: attribute
FCB_ATTR_READONLY     0x01
FCB_ATTR_HIDDEN       0x02
FCB_ATTR_SYSTEM       0x04
FCB_ATTR_VOLUME       0x08
FCB_ATTR_DIR          0x10
FCB_ATTR_ARCHIVE      0x20

```

FcbOpen

FILE I/O

char FcbOpen (FCB *p_fcb)

Opens a file. Set the **FCB** Structure before using this function, all parameters are passed thru this structure. Return **0** if success.

FcbDelete

FILE I/O

char FcbDelete (FCB *p_fcb)

Delete a file. Set the **FCB** Structure before using this function, all parameters are passed thru this structure. Return **0** if success.

FcbCreate**FILE I/O****char FcbCreate (FCB *p_fcb)**

Creates a file on media. Set the **FCB** Structure before using this function, all parameters are passed thru this structure. Return **0** on success. If the file already exists, it will be erased.

This function also open the file for writing process.

FcbClose**FILE I/O****char FcbClose (FCB *p_fcb)**

Closes a file previously opened. Return **0** on success.

FcbRead**FILE I/O****unsigned int FcbRead (FCB *p_fcb, void *p_buffer, int nsize)**

Read **nsize** bytes from opened file, and sends data to ***p_buffer**.

***p_fcb** is the open file handler. The function returns the number of bytes read.

FcbWrite**FILE I/O****char FcbWrite (FCB *p_fcb, const void *p_buffer, int nsize)**

Write **nsize** bytes to an open file, from ***p_buffer**

***p_fcb** is the open file handler.

FcbFindFirst**FILE I/O****char FcbFindFirst (FCB *p_fcb, FCB_FIND *p_result)**

Finds first entry in the directory.

FcbFindNext**FILE I/O****char fcb_find_next (FCB_FIND *p_result)**

Finds next entry in the directory.

SetDisk**FILE I/O****char SetDisk (int diskno)**

Sets **diskno** as current drive number.

Return the number of available disks.

GetDisk**FILE I/O****char GetDisk (void)**

Gets and returns current drive number.

Predefined macros & Structures

```
typedef struct {
    int hour ;           /* Hours 0..23 */
    int min ;            /* Minutes 0..59 */
    int sec ;            /* Seconds 0..59 */
} TIME ;

typedef struct {
    int year ;           /* Year 1980...2079 */
    int month ;          /* Month 1=Jan..12=Dec */
    int day ;             /* Day of the month 1...31 */
    int dow ;            /* On getdate() gets Day of week 0=Sun...6=Sat */
} DATE ;

typedef struct {
    unsigned int IX,IY,AF,DE,BC,HL;
} REGDATA;
```

GetDate

MSX_DOS

void GetDate (DATE *date)

Gets the current date, and sends it to ***date** structure

Code example, get and show date :

```
DATE __at (0x21F0) dt ;
getdate(&dt) ;
putdec(dt.year) ; putch('.');
putdec(dt.month) ; putch('.');
putdec(dt.day) ; putch(' ') ;
```

GetTime

MSX_DOS

void GetTime (TIME *time)

Gets the current time, and send it to ***time** structure

Code example, how to get and show time :

```
TIME __at (0x2100) tm;
GetTime(&tm) ;
PrintDec(tm.hour) ; putch(' :');
PrintDec (tm.min) ; putch(' :'); PrintDec (tm.sec) ;
```

SetDate

MSX_DOS

int SetDate (DATE *date)

Sets the system date thru the ***date** structure.

Returns **0** if valid.

SetTime

MSX_DOS

int SetTime (TIME *time)

Sets the system time thru the ***time** structure.

Returns **0** if valid.

Exit**MSX_DOS****void Exit (char N)**

Exits from C program, and go back to MSX-DOS. *N* represents MSX DOS error number you want to show when exiting.

0 means no error.

CallBios**V1.3****MSX_DOS****void CallBios(void *registers)**

Performs a direct call to Bios routine. According to the BIOS function you want to call, the registers must be previously sets. The **REGDATA** Structure must be declared and configured before the call.

The parameter register IX, must contain the number of the routine to be called.

Example of use :

```
REGDATA Register;
REGDATA Register;           // Declare the structure
Register.IX=0x00A2;         // We want to call function 0xA2 (Console CHPUT)
Register.AF=0x42;           // We Want to print to console the letter B
CallBios(&Register);       // Call the routine
```

CallDos**V1.3****MSX_DOS****void CallDos(void *registers)**

Performs a direct call to MSX-DOS routine. According to the DOS function you want to call, the registers must be previously sets. The **REGDATA** Structure must be declared and configured before the call.

The parameter register C, must contain the number of the routine to be called.

Example of use :

```
REGDATA Register;           // Declare the structure
Register.BC=0x0002;          // We want to call function 0x02 (Console Output)
Register.DE=0x0041;          // We Want to print to console the letter A
CallDos(&Register);         // Call the routine
```

After the call, the registers parameters **AF,DE,BC,HL** of the structure are updated if need.

CallSub**MSX2 V1.3****MSX_DOS****void CallSub(void *registers)**

Performs a direct call to MSX2 SUBROM routine. According to the SUBROM function you want to call, the registers must be previously sets. The **REGDATA** Structure must be declared and configured before the call.

The parameter register IX, must contain the number of the routine to be called.

Example of use :

```
REGDATA Register;           // Declare the structure
Register.IX=0x01B5;          // We want to switch to screen mode 8 (CHGMDP)
Register.BC=0x0008;          // We Want to print to console the letter A
CallSub(&Register);         // Call the routine
```

After the call, the registers parameters **AF,DE,BC,HL** of the structure are updated if need.

char SetRamDisk(char NbSegments)

This function create or destroy a RAMDisk. (available with **MSX-DOS 2 ONLY**)

If **NbSegments = 255** The function just returns the number of 16k RAM segments which are allocated to the RAMDisk currently.

A returned value of zero indicates that there is no RAM disk currently defined. I

If **NbSegments = 0** then the current RAM disk will be destroyed, loosing all data which it contained and no error will be returned if there was no RAM disk.

If **NbSegments** is between **1 and 254** then this function will attempt to create a new RAM disk using the number of 16k segments specified by **NbSegments**.

The returned value is the number of segments allocated.

Note that some of the RAM is used for the file allocation tables and the root directory so the size of the RAMDisk as indicated by "DIR" or "CHKDSK" will be somewhat smaller than the total amount of RAM used.

The RAMDisk will always be assigned the drive letter "H:" regardless of the number of drives in the system.

Turbo-R Functions**GetCPU****TURBO-R****char GetCPU (void)**

Gets the CPU Mode of the R800 processor.

Returned values:

0 : Z80 mode

1 : R800 Rom Mode

2 : R800 Ram Mode

ChangeCPU**TURBO-R****void ChangeCPU (char n)**

Changes the CPU Mode of the R800 processor.

n must be:

0 : Z80 mode

1 : R800 Rom Mode

2 : R800 Ram Mode

PCMPlay**TURBO-R****void PCMPlay (int start, int length)**

Plays a PCM sound stored in the Vram

start : must be the Vram Address of the beginning of the PCM Sound

length : is the length in bytes of the PCM sound to play.

File I/O

[io.h]

Functions to manipulate files with MSX-DOS 1 & MSX-DOS 2. Offer more possibilities than conventional functions.

Predefined macros & Structures

```

SEEK_SET 0
SEEK_CUR 1
SEEK_END 2

O_RDONLY 0
O_WRONLY 1
O_RDWR 1
O_CREAT 0x39
O_EXCL 0x04
O_TRUNC 0x31
O_APPEND 0x41
O_TEMP 0x80

typedef struct {
    chardrive ;           // 0 : default drive
    charfilename[11] ;     // 8+3 for extension, as « MYPROG PRG »
    unsigned int block ;
    unsigned int record_size ;
    unsigned long file_size ;
    unsigned int date ;
    unsigned int time ;
    chardevice ;
    chardir_location ;
    unsigned int top_cluster ;
    unsigned int lastacsd_cluster ;
    unsigned int clust_from_top ;
    charrecord ;
    unsigned long rand_record ;
    charnone ;           //+1 byte
} FCBstru ;             // 38 bytes

typedef struct {
    FCBstru fcb[8] ;
} FCBlist ;
extern   FCBlist *FCBs( void ) ;
extern   int _io_errno ;

// Structure that will receive Get Disk Parameters data (MSX-DOS2)
typedef struct {
    char DriveN;          // Physical drive number (1=A: etc
    int SectorSize;        // Sector size (always 512 currently)
    char SectorPerCluster; // Sectors per cluster (non-zero power of 2)
    int NumberReservedSector; // Number of reserved sectors (usually 1)
    char NumberFatCopy;    // Number of copies of the FAT (usually 2)
    int NumberRootDirEntries; // Number of root directory entries
    int TotalLogicalSectors; // Total number of logical sectors
    char MediaDescriptorByte; // Media descriptor byte
    char NumberSectorsPerFat; // Number of sectors per FAT
    int FirstRootSectorNumber; // First root directory sector number
    int FirstDataSectorNumber; // First data sector number
    int MaximumCluster; // Maximum cluster number
    char DirtyFlag; // Dirty disk flag
    char VolumeId[4]; // Volume id. (-1 => no volume id.)
    char Reserved[8]; // Reserved (currently always zero)
} DSKPARAMS;

```

DiskLoad**IO****int DiskLoad (char* filename, unsigned int address, unsigned int run_address)**

Loads binary file from disk to RAM.

- *filename* : is 11 chars DOS1 for FCB : Example « MYFILE01BIN »- *address* : where to load the first byte- *run_address* : if not 0, then where to CALL after loaded

Returns 0 on success.

GetOSVersion**IO****int GetOSVersion(void)**

Returns OS version :

1 for MSX DOS 1

2 for MSX DOS 2

0 if not initiated

Global Variables changed after calling :

_os_ver : is set with MSX-DOS Kernel version

_mx_ver : is set with MSXDOS2.SYS version number.

Open**IO****int Open(char *file_name, int mode)**

Opens a file. Return an INT value as file Handler, or -1 if error

mode can be:

- O_RDONLY 0x0 : read only

- O_WRONLY 0x1 : write only

File operation errors are sent to the variable : _io_errno

Create**IO****Int Create (char *file_name)**

Creates a file, named as *file_name

Close**IO****Int Close (int fH)**Closes a file handler, previously opened. *fH* is the file handler that must be closed

File operation errors are sent to the variable : _io_errno

Read**IO****Int Read (int fH, void *buffer, unsigned int nbytes)**Read *nbytes* bytes from a opened file handler definied by *fH* to **buffer*.

File operation errors are sent to the variable : _io_errno

FCBlist**IO****FCBlist *FCB = FCBs()**

Mandatory when reading or writing files with MSX DOS 1. (Or MSX DOS 2)
 FCBLIST will give a file handler.

Example of use :

```
char sbuf[10];                                // Set a 10 bytes buffer
FCBlist *FCB = FCBs();                         // FCB initialization
int fH;                                         // Set a file handler variable
fH = open( « TEST0001.SC8 », O_RDWR );          // open file for read
read(fH, sbuf, 10);                            // Read 10 bytes to 108
close(fH);                                     // Close file
```

Write**IO****Int Write (int fH, void *buffer, unsigned int nbytes)**

Writes *nbytes* bytes from **buffer* to an opened file handler defined by *fH* handler.

File operation errors are sent to the variable : *_io_errno*

Ensure**DOS2****IO****Int Ensure (int fH)**

Ensure that a file handler previously writed is correct. Verify all flags and flush all data from buffers. *fH* is the file handler that must be ensured

File operation errors are sent to the variable : *_io_errno*

OpenAttrib**DOS2****IO****Int OpenAttrib (char *file_name, int mode, int attr)**

Opens a file with attributs.

Attributs can be :

- O_CREAT	0x39	: create file mode
- O_EXCL	0x04	:
- O_TRUNC	0x31	:
- O_APPEND	0x41	: append
- O_TEMP	0x80	

File operation errors are sent to the variable : *_io_errno*

CreateAttrib**DOS2****IO****Int CreateAttrib (char *name, int attr)**

Creates a file named **name* (*attr* is same as the open function),

GetCWD**IO****Int GetCWD(char *buf, int bufsize)**

Gets directory of A : to the buffer **buf*, number of data *is* limited by *bufsize*

Ltell**IO****Int Ltell (int fH, long value)**

Moves the file handler pointer to *value* from the current pointer position.

Returns **0** on success, or error in *_io_error*

Lseek**IO****Int Lseek (int fH, long value, unsigned int offsetCode)**Moves the file handler pointer to *value* according to the method of the *offsetCode*.Returns **0** on success, or error in *_io_error***offsetCode** can be :

- SEEK_SET : 0 Relative to the beginning of the file
- SEEK_CUR : 1 Relative to the current position
- SEEK_END : 2 Relative to the end of the file

Remove**IO****Int Remove (char *filename)**Removes file **filename* from directory.Returns **0** on success, or error in *_io_error***Rename****IO****Int Rename (char *old_name, char *new_name)**Rename file or folder **old_name* by **new_name*Returns **0** on success, or error in *_io_error***ChangeDir****DOS2****IO****Int ChangeDir (char *Directory)**Sets current path to the **Directory*.Returns **0** or error in *_io_error*

Example : ChangeDir("Games")

FindFirst**DOS2****IO****Int FindFirst(char *wildcard, char *result, int attr)**

Finds files or folders by wildcard as « *.COM », « ???? », etc.

Returns **0** on success, or error in *_io_error*.

```
n=findfirst(« *.* »,sbuf,0) ;
for( ;!n ;){
    cputs(sbuf) ; cputs(sn); n=findnext(sbuf) ;
}
```

FindNext**DOS2****IO****Int FindNext(char *result)**See **findfirst** function. Find next occurrence.**MakeDir****DOS2****IO****int MakeDir(char *folderName)**Creates a directory folder. Returns **0**, or error in : *_io_error*

RemoveDir**DOS2****IO****int RemoveDir(char *folderName)**Removes a directory folder. Returns 0, or error in : **_io_error****GetDiskParam****DOS2 V1.1****IO****char GetDiskParam(DSKPARAMS *info, char Drive)**

After Initialization of the DSKPARAMS Structure the call of this function will get all parameters of the disk drive. It may be a floppy Drive or any other MSX-DOS 2's storage drive.

See the DSKPARAMS Structure at the beginning of this chapter for details.
(Available with **MSX-DOS 2 ONLY**)

SetDiskTrAddress**V1.1****IO****void SetDiskTrAddress (unsigned int *address)**

Sets the Memory ***address** where the data will be transferred, when a Reading Sectors instruction is called or when a Writing function is called.

Most of the time a sector size is 512 bytes long (See result of a GetDiskParam), thus if you want to store at least one sector, the ***address** pointer must point to a 512 bytes variable area (minimum).

GetDiskTrAddress**V1.1****IO****unsigned int GetDiskTrAddress (void)**

Returns the Memory address where the data are transferred, when a Reading Sectors instruction is called, or when a writing sector instruction is called.

SectorRead**V1.1****IO****char SectorRead (unsigned int SectorStart, char drive, char NbSectors)**

Reads **NbSectors** Sectors from **SectorStart**, on the specified **drive**.

Data are sent to the memory address set by SetTrAddress instruction.

Returns **0** if no error.

SectorWrite**V1.1****IO****char SectorWrite (unsigned int SectorStart, char drive, char NbSectors)**

Write **NbSectors** Sectors from **SectorStart**, on the specified **drive**.

Sectors are written with the data stored at the memory address set by the instruction SetTrAddress instruction.

Returns **0** if no error.

MSX1 GRAPHICS

[vdp_graph1.h]

MSX1 Graphic and draw functions.

Predefined macros & Structures

```
// filling mode
NO_FILL          0x00
FILL_ALL         0xFF
/* structure to set/get color of 8 pixels */
typedef struct {
    int col ;           // color number 0..15 for pixels of pattern
    int bg ;            // background color number 0..15
} pxColor ;
```

SC2WriteScr***VDP_GRAPH1*****void SC2WriteScr (unsigned int addr_Palettes, unsigned int addr_Colors)**

Writes RAM to VRAM (2x6144 bytes)

Use with Screen mode 2 or 3.

SC2ReadScr***VDP_GRAPH1*****void SC2Read_Scr (unsigned int addr_toPalettes, unsigned int addr_toColors)**

Reads VRAM to RAM(2x6144 bytes)

Use with Screen mode 2 or 3.

Get8px***VDP_GRAPH1*****int Get8px (int X, int Y)**

Gets byte of 8-pixels at (X,Y)

ReadBlock***VDP_GRAPH1*****void ReadBlock (int X, int Y, int dx, int dy, unsigned int addr_toPalettes, unsigned int addr_toColours)**

VRAM => RAM (copy block to memory)

(X,Y) – left upper corner of screen position to copy

dx,dy – count of columns and rows of pixels

So, the block (X,Y)-(X+dx-1,Y+dy-1) will be copied.

X,dx should be 0,8,16,24,32,... 8*n because

complete 8-pixel patterns will be copied

Requires 2 memory blocks size of (dx/8)*dy

WriteBlock***VDP_GRAPH1*****void WriteBlock(int X, int Y, int dx, int dy, unsigned int addr_Palettes, unsigned int addr_Colours)**

RAM => VRAM (puts from memory to screen)

(X,Y) – where to put on screen

<i>Get1px</i>	<i>VDP_GRAPH1</i>
<code>int Get1px(int X, int Y)</code>	Gets pixel of 8-pixels at (X,Y). Returns 0 if not set.

<i>Set8px</i>	<i>VDP_GRAPH1</i>
<code>void Set8px(int X, int Y)</code>	Sets whole byte of 8-pixels at (X,Y)

<i>Set1px</i>	<i>VDP_GRAPH1</i>
<code>void Set1px(int X, int Y)</code>	Sets pixel of 8-pixels at (X,Y)

<i>Clear8px</i>	<i>VDP_GRAPH1</i>
<code>void Clear8px (int X, int Y)</code>	Clears byte (sets=0) of 8-pixels at (X,Y)

<i>Clear1px</i>	<i>VDP_GRAPH1</i>
<code>void Clear1px (int X, int Y)</code>	Clears pixel (sets=0) of 8-pixels at (X,Y)

<i>GetCol8px</i>	<i>VDP_GRAPH1</i>
<code>void GetCol8px (int X, int Y, pxColor *C)</code>	Get color of 8-pixel pattern at (X,Y) See the predefined structure.

<i>SetCol8px</i>	<i>VDP_GRAPH1</i>
<code>void SetCol8px (int X, int Y, pxColor *C)</code>	Sets new color in (X,Y) for 8-pixel pattern <code>typedef struct {</code> <code>int col ; // color number 0..15 for pixels of pattern</code> <code>int bg ; // background color number 0..15</code> <code>} pxColor ;</code>

<i>SC2Point</i>	<i>VDP_GRAPH1</i>
<code>int SC2Point (int X, int Y)</code>	Gets color of pixel at (X,Y), (same for 8-pixel pattern) Use only with Screen mode 2.

<i>SC2Pset</i>	<i>VDP_GRAPH1</i>
<code>void SC2Pset (int X, int Y, int color)</code>	Puts pixel at (X,Y) position, with color <i>color</i> . Use only with Screen mode 2.

SC2Line***VDP_GRAPH1***

```
void SC2Line (int X, int Y, int X2, int Y2, int color)
```

Draws a line from (X, Y) position to $(X2, Y2)$ position, with *color*,
 Does not change background color
 Use only with Screen mode 2.

SC2Paint***VDP_GRAPH1***

```
void SC2Paint ( int X, int Y, int color )
```

Paints for small screen regions. Slow Function!
 Use with Screen mode 2.

SC2BoxFill***V1.2******VDP_GRAPH1***

```
void SC2BoxFill (char X1, char Y1, char X2, char Y2, char color)
```

Draws a filled rectangle *from X1,Y1* (left upper corner) to *x2,y2* (right bottom corner) with
color. No Logical operation. Use only with Screen mode 2.

SC2BoxLine***V1.3******VDP_GRAPH1***

```
void SC2BoxLine (char X1, char Y1, char X2, char Y2, char color )
```

Draws an empty rectangle *from X1,Y1* (left upper corner) to *x2,y2* (right bottom corner)
 with *color*. No Logical operation.
 Use only with Screen mode 2.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation



MSX2 GRAPHICS

[vdp_graph2.h]

MSX2 specific graphic functions.

Predefined Structures and macros**Logical fill for rectangle and circle**

FILL_ALL	0xFF
NO_FILL	0x00

Palette

```
typedef struct {
    char      colour; // color number 0..15
    char      R;      // 0..7    red brightness
    char      G;      // 0..7    green brightness
    char      B;      // 0..7    blue brightness
} ColRGB;

typedef struct {
    ColRGB rgb[16];
} Palette;
```

fLMM variable structure :

```
typedef struct {
    unsigned int X;           // source X (0 to 511)
    unsigned int Y;           // source Y (0 to 1023)
    unsigned int X2;          // destination X (0 to 511)
    unsigned int Y2;          // destination Y (0 to 1023)
    unsigned int DX;          // width (0 to 511)
    unsigned int DY;          // height (0 to 511)
    chars0;                  // set to 0, dummy 1st empty byte sent to chip
    charDI;                  // set to 0 (b), works well from left to right
    charLOP;                 // 0 to copy (a), Logical+Operation
} MMMtask;
```

Logical operations

When graphic commands are called, various logical operations can be performed between the source data and the destination data. Here how logical operation are working.

SC : is for source color code

DC : is for destination color code

Param.	Name	action
0	LOGICAL_IMP	DC=SC
1	LOGICAL_AND	DC=SC and DC
2	LOGICAL_OR	DC=SC or DC
3	LOGICAL_XOR	DC=SC xor DC
4	LOGICAL_NOT	DC=SC ! DC
8	LOGICAL_TIMP	if SC=0 then DC=DC else DC=SC
9	LOGICAL_TAND	if SC=0 then DC=DC else DC=SC and DC
10	LOGICAL_TOR	if SC=0 then DC=DC else DC=SC or DC
11	LOGICAL_TXOR	if SC=0 then DC=DC else DC=SC xor DC
12	LOGICAL_TNOT	if SC=0 then DC=DC else SC ! DC

One of the most used operator is TIMP. It permit to copy a part of the screen over another without the transparent color (color 0 is transparent).

vMSX	VDP_GRAPH2
Int vMSX (void) Check MSX VDP version. Returns 1 for MSX1, or 2 for MSX2.	

Pset	MSX2	VDP_GRAPH2
void Pset (int X, int Y, char color, int OP) Draws a pixel at <i>X,Y</i> with the defined <i>color</i> and logical operation <i>OP</i>		

Point	MSX2	VDP_GRAPH2
char Point (int X, int Y) Reads and return color of the pixel at <i>X,Y</i>		

Line	MSX2	VDP_GRAPH2
void Line (int X1, int Y1, int X2, int Y2, int color, int OP) Draws a line from <i>X1,Y1</i> to <i>X2,Y2</i> with the defined <i>color</i> and logical operation <i>OP</i>		

BoxLine	MSX2	VDP_GRAPH2
void BoxLine (int X1, int Y1, int X2, int Y2, int color, int OP) Draws a rectangle from <i>X1,Y1</i> (left upper corner) to <i>X2,Y2</i> (right bottom corner) with the defined <i>color</i> and logical operation <i>OP</i> . Use FILL_ALL as operator to fill the rectangle. Any Other operator will draw an empty rectangle		

BoxFill	V1.2 MSX2	VDP_GRAPH2
void BoxFill (int X1, int Y1, int X2, int Y2, char color, char OP) Draws a filled rectangle <i>from X1,Y1</i> (left upper corner) to <i>x2,y2</i> (right bottom corner) with <i>color</i> and logical operation <i>OP</i> .		

High speed VDP Commands

The YAMAHA V9938 VDP Processor of the MSX2, and the YAMAHA V9958, the VDP processor of the MSX2+ and Turbo-R comes with high speed functions you can use to build your own graphic routines.

VDP COMMANDS SUMMARY					
Type	Source	Destination	Unit	Name	Code
High speed move	CPU	VRAM	Byte	HMMC	0xF0
	VRAM	VRAM		YMMM	0xE0
	VRAM	VRAM		HMMM	0xD0
	VDP	VRAM		HMMV	0xC0
Logical Move	CPU	VRAM	Dot	LMMC*	0xB0
	VRAM	CPU		LMCM	0xA0
	VRAM	VRAM		LMMM	0x90
	VDP	VRAM		LMMV	0x80
Line	VDP	VRAM		LINE ***	0x70
Search	VDP	VRAM		SRCH **	0x60
Pset	VDP	VRAM		PSET ***	0x50
point	VRAM	VDP		POINT ***	0x40
Stop				STOP	0x00

* Split into 2 individual commands, LMCM5 & LMCM8. Standard LMCM can be used with *fVDP* function

** Not implemented as individual command. But you can use it via *fVDP* function

*** Functions described in the previous chapter

These commands use the global coordinates system of the whole MSX's 128 KB VRAM.

GRAPHIC4		Address	GRAPHIC5	
(0, 0)	(255, 0)	0000h	(0, 0)	(511, 0)
Page 0			Page 0	
(0, 255)	(255, 255)	08000h	(0, 255)	(511, 255)
(0, 256)	(255, 256)		(0, 256)	(511, 256)
Page 1			Page 1	
(0, 511)	(255, 511)	10000h	(0, 511)	(511, 511)
(0, 512)	(255, 512)		(0, 512)	(511, 512)
Page 2			Page 2	
(0, 767)	(255, 767)	18000h	(0, 767)	(511, 767)
(0, 768)	(255, 768)		(0, 768)	(511, 768)
Page 3			Page 3	
(0, 1023)	(255, 1023)	1FFFFh	(0, 1023)	(511, 1023)
GRAPHIC7			GRAPHIC6	
(0, 0)	(255, 0)	0000h	(0, 0)	(511, 0)
Page 0			Page 0	
(0, 255)	(255, 255)	10000h	(0, 255)	(511, 255)
(0, 256)	(255, 256)		(0, 256)	(511, 256)
Page 1			Page 1	
(0, 511)	(255, 511)	1FFFFh	(0, 511)	(511, 511)

HMMC**MSX2****VDP_GRAPH2**

```
void HMMC (void *pixeldata, int DX int DY, int NX, int NY)
```

High speed move CPU to VRAM.

This function sends the RAM ***pixeldata** buffer to VRAM at **DX,DY** coordinates.

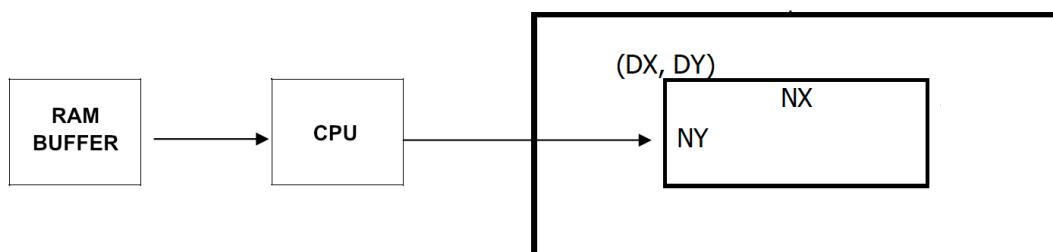
The datas sent must be a rectangle.

NX is length of the zone to copy in pixels

NY is height of the zone to copy in pixels

Use **DY** Coordinate >= 256 to copy the buffer to other vram page.

VDP - VRAM

**LMMC****V1.2 MSX2****VDP_GRAPH2**

```
void LMMC (void *pixeldata, int DX, int DY, int NX, int NY, char OP )
```

Logical Move CPU to VRAM. (Same as **HMMC** but with Logical operator)

This function sends the RAM ***pixeldata** buffer to VRAM at **DX,DY** coordinates.

The datas sent must be a rectangle.

NX is length of the zone to copy in pixels

NY is height of the zone to copy in pixels

OP is a standard logical operator parameter. See the list in previous chapters.

In Screen mode 5 or 7, if you want to use **LMMC** command, you must previously transfer data to RAM buffer with **LMCM8** instead of **LMCM5**

LMCM5**V1.3 MSX2****VDP_GRAPH2**

```
void LMCM5 (void *buffer, int SX, int SY, int NX, int NY, char OP);
```

Logical Move VRAM to CPU.

This function sends a copy of the VRAM to the RAM ***pixeldata** buffer.

The VRAM block sent is a rectangle. The top left of the rectangle is at **SX,SY** coordinates.

The length of the rectangle is **NX** pixels, and the height of the rectangle is **NY** pixels.

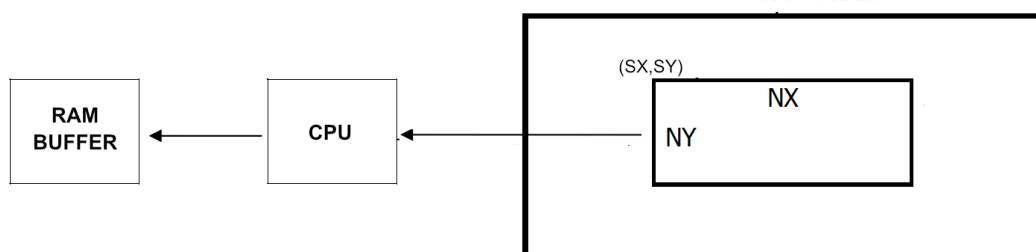
This function is coding 2 color pixel into 1 byte. So it works well with Screen modes 5 or 7.

OP is a the logical operator to apply. Only two are available.

- **0** : IMP

- **8** : TIMP (In other words this operator will copy to RAM all bytes except those with a value of 0)

VDP - VRAM



LBCM8	V1.3 MSX2	VDP_GRAPH2
<pre>void LBCM8 (void *buffer, int SX, int SY, int NX, int NY, char OP);</pre> <p>Logical Move VRAM to CPU. (Same as previous function, but for Screen mode ≥ 8) This function sends a copy of the VRAM to the RAM *pixeldata buffer. The VRAM block sent is a rectangle. The top left of the rectangle is at SX,SY coordinates. The length of the rectangle is NX pixels, and the height of the rectangle is NY pixels. This function is coding 1 color pixel into 1 byte. So it works well with Screen modes 8 to 12. But it may be used also in screen mode 5 or 7 if you need to send back the buffer to the VDP with a logical operator OP is a the logical operator to apply. Only two are available. - 0 : IMP - 8 : TIMP (In other words this operator will copy to RAM all bytes except those with a value of 0)</p>		

YMMM	V1.2 MSX2	VDP_GRAPH2
<pre>void YMMM (int SX, int SY, int DY, int NY)</pre> <p>High speed move VRAM to VRAM on Y coordinate only. This function only copy the rectangle portion of the image from position (SX,SY) to (255,SY+NX) to the position DY</p> <p>This function is usefull to move full image or portion of an image to another VRAM page. <i>Note: Harware limitation. In Screen mode 5 & 7, SX must be an even number. In screen mode 6, SX and NY must be a multiple of 4.</i></p> <p style="text-align: center;">VDP - VRAM</p>		

LMMM	V1.2 MSX2	VDP_GRAPH2
<pre>void LMMM (int SX, int SY, int DX, int DY, int NX, int NY, char OP)</pre> <p>Logical move VRAM to VRAM. (Same as HMMM but with a logical operator) This function copy a rectangle image starting at the top left coordinates SX,SY to the destination DX,DY coordinates. The length of the rectangle is NX pixels, and the height of the rectangle is NY pixels. OP is a standard logical operator parameter. See the list in previous chapters.</p>		

HMMM**V1.2 MSX2****VDP_GRAPH2**

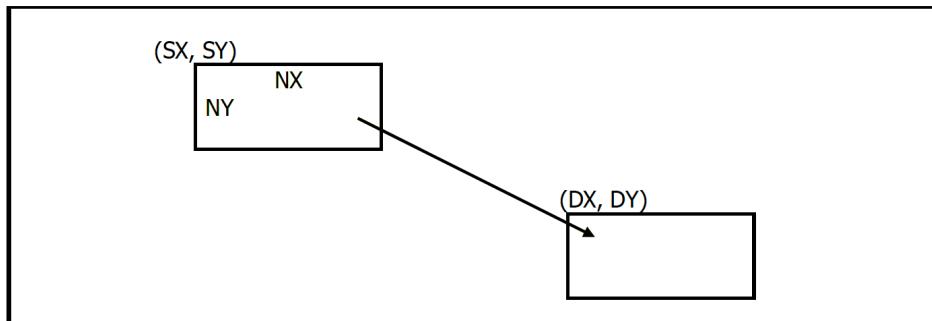
```
void HMMM (int SX, int SY, int DX, int DY, int NX, int NY)
```

High speed from VRAM to VRAM

This function copy a rectangle image starting at the top left coordinates **SX,SY** to the destination **DX,DY** coordinates. The length of the rectangle is **NX** pixels, and the height of the rectangle is **NY** pixels.

Note: Harware limitation. In Screen mode 5 & 7, SX & DX must be an even number. In screen mode 6, SX and DX must be a multiple of 4.

VDP - VRAM

**HMMV****V1.2 MSX2****VDP_GRAPH2**

```
void HMMV ( int DX, int DY, int NX, int NY, char COL )
```

High speed move VDP to VRAM

Fill a rectangle starting at top left corner **DX,DY** with color **COL**

The length of the rectangle is **NX** pixels, and the height of the rectangle is **NY** pixels.

When working with screen 5 or 7, HMMV will fill 2 horizontal pixels at the same time.

The **COL** parameter must be divided into two blocks of 4 bits
example Color : **0bAAAA BBBB**.

The left 4 bits will be used for the left pixel color, and the 4 right bits will be used for right pixel color.

You can also use this formula to calculate the good value of your **COL parameter** :

```
color =12;           // use color 12
color=((color << 4) | color); // Use color 12 for both pixels
```

Note: Harware limitation. In Screen mode 5 & 7, DX & NX must be an even number. In screen mode 6, DX and NX must be a multiple of 4.

LMMV**V1.2 MSX2****VDP_GRAPH2**

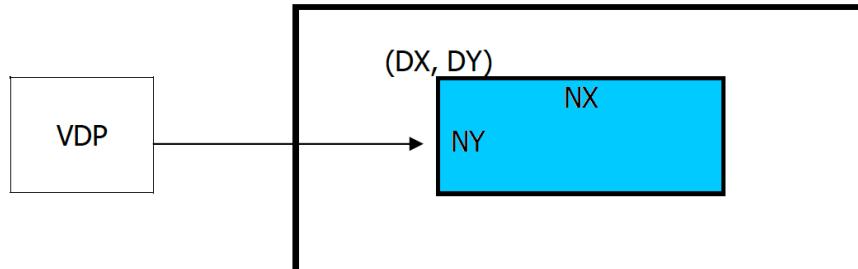
```
void LMMV ( int SX, int SY, int DX, int DY, char Color, char OP)
```

Logical Move VDP to VRAM (Same as HMMV, but with a logical operator)

Fill a rectangle starting at top left corner **DX,DY** with color **COL**

The length of the rectangle is **NX** pixels, and the height of the rectangle is **NY** pixels.

VDP - VRAM

**VDPLINE****V1.2 MSX2****VDP_GRAPH2**

```
void VDPLINE( int DX, int DY, int NX, int NY, int COL, char PARAM, char OP)
```

Draw a line with the dedicated VDP command. (The Line Function is using this VDP command).

The line drawn is the hypotenuse of the triangle defined by the “Long” side and the “Short” side. The distances are defined from the starting point coordinates **DX,DY**

NX parameter must contain the size in pixels of the longest segment of the triangle.

NY parameter must contain the size in pixels of the shortest segment of the triangle.

COL must contain the color number to use to draw the line.

OP is a standard logical operator parameter. See the list in previous chapters.

PARAM, contains parameters coded in a byte

7	6	5	4	3	2	1	0
0	0	MXD	0	DIY	DIX	0	MAJ

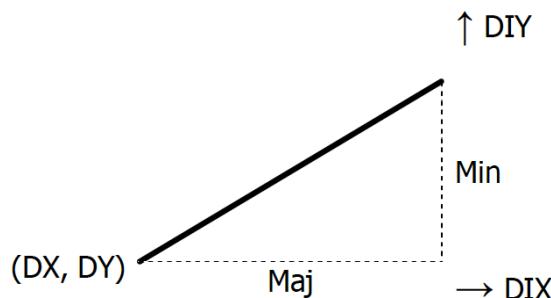
MAJ bit :

must be **0** if the long side is on X axis, **1** if long side is on Y axis

DIX bit : must be **0** if the line goes to the right, **1** if it goes to the left

DIY bit : must be **0** if the line goes to down, **1** if it goes to up.

MXD bit : must be **0** as long as your MSX do not have expansion of VRAM memory



fLMMM	MSX2	VDP_GRAPH2
void fLMMM (MMMtask *VDPtask)		
High speed copy by structure <i>*VDPtask</i> . Same effect as LMMM command but it is using a structure where you can set your parameters. By using this method the command is faster completed.		
First declare the structure, for example : <i>static MMMtask t_vdp ;</i>		
Predefined structure :		
<pre>typedef struct { unsigned int X ; // source X (0 to 511) unsigned int Y ; // source Y (0 to 1023) unsigned int X2 ; // destination X (0 to 511) unsigned int Y2 ; // destination Y (0 to 1023) unsigned int DX ; // width (0 to 511) unsigned int DY ; // height (0 to 511) char 0 ; // set to 0. 1st empty byte sent to chip char DI ; // set to 0 (b), works well from left to right char LOP ; // 0 to copy (a), Logical Operation } MMMtask ;</pre>		
Note : <i>The fLMMM is useless since the introduction of fVDP command. It is here for compatibility reasons. It's better to use fVDP function.</i>		

fVDP**V1.3 MSX2****VDP_GRAPH2****void fVDP (void *parameters)**

This Fast VDP function give you access to all VDP Commands in the fastest way possible.
To use this function you must first initialize the dedicated predefined structure **FastVDP**

```
typedef struct {
```

```
    unsigned int SX;      // source X (0 to 511)
    unsigned int SY;      // source Y (0 to 1023)
    unsigned int DX;      // destination X (0 to 511)
    unsigned int DY;      // destination Y (0 to 1023)
    unsigned int NX;      // width (0 to 511)
    unsigned int NY;      // height (0 to 511)
    char COL;            // color used by some commands. or 0 if not used
    char DI;              // Parameters set the direction ex : opDOWN | opRIGHT
    char CMD;             // VDP Comd + Logical Operator ex: opLMMM | LOGICAL_TIMP
```

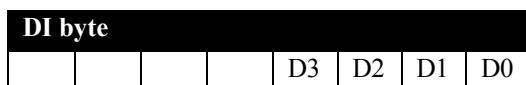
```
} FastVDP;
```

Command & code	Logical operator & code	Direction & code
opHMMC	0xF0	LOGICAL_IMP
opYMMM	0xE0	LOGICAL_AND
opHMM	0xD0	LOGICAL_OR
opHMMV	0xC0	LOGICAL_XOR
opLMMC*	0xB0	LOGICAL_NOT
opLMCM	0xA0	LOGICAL_TIMP
opLMM	0x90	LOGICAL_TAND
opLMMV	0x80	LOGICAL_TOR
opLINE	0x70	LOGICAL_TXOR
opSRCH	0x60	LOGICAL_TNOR
opPSET	0x50	
opPOINT	0x40	
opSTOP	0x00	

Some precisions about the variables.

DI is the direction the VDP command will use to make the action. The most common case is to leave it at 0. In some cases, for example if you want to make a right to left scrolling, it can be usefull to change it to use the left direction.

DI is a byte, but only the 4 first bits are used. You can combine two direction by using a logical “or” operator ”|”



CMD is the VDP command code to use. This 8 bits value is separate into 2 quartets.



The most significant quartet is the command code (C0 to C3), the less significant quartet is the logical operator code (L0 to L3).

Only some commandes accept a logical operator (see the summary array at the beginning of this chapter). To apply a logical operator, use an logical “or” on the CMD code, or leave it to 0.

Example:

Imagine we want to copy a portion of the image of the screen mode 8, from page 1 to page 0 with an transparent logical operator.

The block we want to copy is at (150,100) on page 1. It's width is 100 pixels, and height 50 pixels.

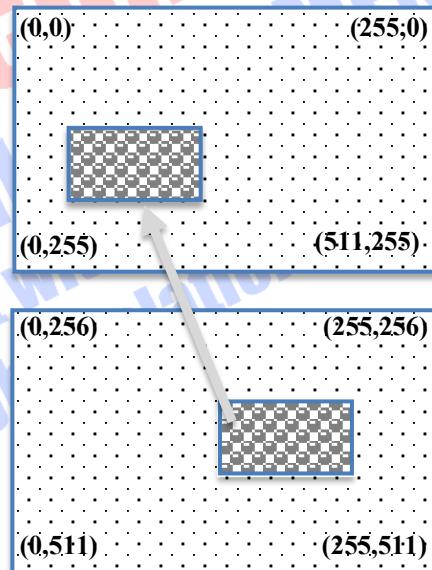
We want to copy this block to (20,25) on page 0.

```
static FastVDP MyCommand; // Set the structure as global

MyCommand.SX=150; // Set the X source
MyCommand.SY=100+256; // Set the Y source on page 1
MyCommand.DX=20; // Set the X destination
MyCommand.DY=25; // Set the Y destination
MyCommand.NX=150; // Set the width of the block
MyCommand.NY=50; // Set the height of the block
MyCommand.COL=0; // no need color. Leave 0
MyCommand.DI=0; // No need to change process direction
MyCommand.CMD=opHMM | LOGICAL_TIMP; // Command and Logical operator

fVDP(&MyCommand); // Call the command
```

Note : For the next uses, you just have to modify the needed variables.



PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

PAINT

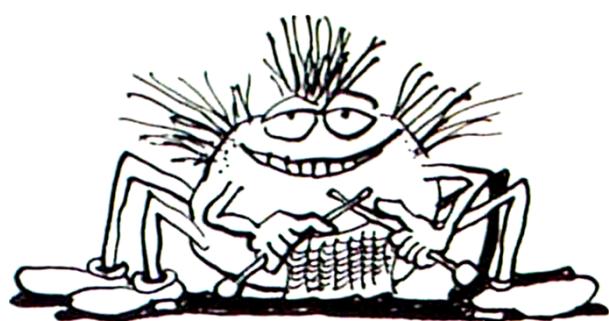
[vdp_paint.h]

The fast FloodFill routine for MSX2 and upper. Must be used in conjunction with *vdp_graph2.h*

Paint	v1.3 MSX2	VDP_PAINT
void Paint (int X, int Y, int COLOR) This fast Flood Fill routine for MSX2 paints any area of the VDP VRAM. The paint process will start at <i>X,Y</i> coordinates. The color found at <i>X,Y</i> is considered as the background color, and is replaced by COLOR . This function need a buffer to work. This buffer must initialized before the first use. Use exactly this way inside your main : <pre style="background-color: black; color: white; padding: 10px;">unsigned char *PaintBuffer; Paint_vars.MaxRam=MAXPAINT_BUFFER; PaintBuffer=MMalloc(Paint_vars.MaxRam); SetPaintBuffer(PaintBuffer);</pre> The default buffer size is 250 Bytes. The size varies according to the number of different zones to be treated by the PAINT routine. If you notice that certain zones are not treated, increase this buffer by modifying the value of MAXPAINT_BUFFER in the file fusion-c / heaber/vdp_paint.h When you no longer need to use the paint function. Do not forget to free the memory allocated to the buffer. In any case, free this memory at the end of your program. <pre style="background-color: black; color: white; padding: 10px;">Free (PaintBuffer);</pre>		

SetPaintBuffer	v1.3 MSX2	VDP_PAINT
void SetPaintBuffer (char* BufAddr) This function define and initialize the mandatory PaintBuffer.		

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation



SPRITES

[vdp_sprites.h]

All necessary function to control MSX Hardware sprites.

SpriteOn	MSX2	VDP_SPRITES
void SpriteOn (void)		

Enables Sprites.

SpriteOff	MSX2	VDP_SPRITES
void SpriteOff (void)		

Disables Sprites. (Note : Disabling sprite, make VDP run a little faster)

Sprite8	MSX2	VDP_SPRITES
void Sprite8 (void)		

Sets sprites size to 8x8 pixels pattern mode.

Sprite16	MSX2	VDP_SPRITES
void Sprite16 (void)		

Sets sprites to 16x16 pixels pattern

Note that 16x16 pixels sprites are in fact composed with four 8x8 pixels patterns. See **SetSpritePattern** function for details.

SpriteSmall	MSX2	VDP_SPRITES
void SpriteSmall (void)		

Sets normal pixel sprite size.

SpriteDouble	MSX2	VDP_SPRITES
void SpriteDouble (void)		

Sets double pixel sprite size.

SpriteReset	MSX2	VDP_SPRITES
void SpriteReset (void)		

Resets all sprites attributes and patterns.

SpriteCollision	MSX2	VDP_SPRITES
char SpriteCollision (void)		

Returns 1 in case of a sprite collision.

SpriteCollisionX	MSX2	VDP_SPRITES
char SpriteCollisionX (void)		

Returns the X position of a sprites collision.

SpriteCollisionY***VDP_SPRITES*****char SpriteCollisionY (void)**Returns the **Y** position of a sprites collision.Always read the **X** position of the collision before reading the **Y** position, or the result will be false.***PutSprite******VDP_SPRITES*****void PutSprite (char sprite_n, char pattern_n, char x, char y, char color)**Puts the **sprite_n** on screen with the defined pattern **pattern_n** at position **X** and **Y** with color **color**.On MSX2 and upper you must define the sprite color with **SpriteColor** functions.
In case you are using the 16x16 pixels sprite mode, **pattern_n** must be equal to the first pattern of the serie (aka Pattern 0, if you are referring to the previous schematic)***fPutSprite*****V1.3*****VDP_SPRITES*****void fPutSprite (void *parameters)**This function is faster than the standard PutSprite function. It use a dedicated predefined structure to pass the parameters: **FastSPRITE**.

```
typedef struct {
    char spr;          // Sprite ID
    char y;            // X destination of the Sprite
    char x;            // Y destination of the sprite
    char pat;          // Pattern number to use
    char col;          // Color to use (Not usable with MS2's sprites)
} FastSPRITE;
```

To use this function first declare the structure, for example like this :

static FastSSPRITE MySprite;

Then assign the parameters to the structure, and call the function, like this:

```
MySprite.spr=1;           // Attribut for Sprite 1
MySprite.y=player.y;      // Y coordoniate of the sprite
MySprite.x=player.x;      // X coordinate of the sprite
MySprite.pat=8;           // Pattern to use
MySprite.col=1;           // Color to use
fPutSprite(&MySprite);    // Call the function
```

SetSpritePattern***VDP_SPRITES*****void SetSpritePattern (char pattern_n, char* p_pattern, char s_size)**Sets the **pattern_n** with ***p_pattern** data. **s_size** is number of line of the pattern.In case you are using 16x16 pixels sprite mode, ***p_pattern** must be composed of 32 bytes in a specific order. A 16x16 pixels sprite's pattern is composed of four 8x8 pixels sprite's patterns like this:

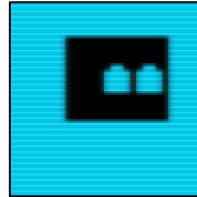
16x16 pixels sprite			
Pattern n0	Pattern n2	Pattern n1	Pattern n3

Thus the 8 first bytes compose the pattern **n0**, the 8 following bytes compose the pattern **n1**, the next 8 bytes compose the pattern **n2**, and finally the 8 last bytes compose the pattern **n3**.

Sprite32Bytes***VDP_SPRITES*****char *Sprite32Bytes (unsigned int *bindata)**Convert a 32 bytes array to a 16x16 pattern usable by the *SetSpritePattern*.

```
static const unsigned int pattern[]={  
0b1111111111111111,  
0b0000000000000000};
```

SetSpritePattern(0,Sprite32Bytes(pattern),32);

***SpriteOverlap******V1.3******VDP_SPRITES*****Char SpriteOverlap (void)**Checks if more than 4 sprites are on the same horizontal line on MSX1's screen modes ; or more than 8 sprites on the same horizontal line on MSX2's screen modes. Returns **1** if overlapping is detected.***SpriteOverlapId******VDP_SPRITES*****Char SpriteOverlapId (void)**In case of sprites overlapping, this function returns the **ID** of the first extra sprite on the same horizontal line. (The **ID** of the 5th sprite, or the 9th sprite that cause overlapping).

SetSpriteColors**VDP_SPRITES**

```
void SetSpriteColors (char spriteNumber, char *data)
```

With MSX2 Screen's modes, you can define each line of the sprite *spriteNumber* with a specific color, set in **data*.

**data* must be an array of 16 bytes, one byte for each color line of a 16x16 pixels sprite. If you are using 8x8 pixels sprites mode, only the first 8 bytes of the array will be used.

In Screen Mode 8 the sprite palette is fixed and cannot be redefined.

Here the Sprite color palette used in Screen mode 8:

	R	G	B	Name
00	0	0	0	Black
01	0	0	2	Dark Blue
02	3	0	0	Dark Red
03	3	0	2	Dark Purple
04	0	3	0	Dark Green
05	0	3	2	Turquoise Blue
06	3	3	0	Green Olive
07	3	3	2	Grey
08	4	4	2	Ligh Orange
09	0	0	7	Blue
10	7	0	0	Red
11	7	0	7	Purple
12	0	7	0	Green
13	0	7	7	Light Blue
14	7	7	0	Yellow
15	7	7	7	White

Pattern16RotationVram**V1.3****VDP_SPRITES**

```
void Pattern16RotationVram (char pattern, signed char rotation, char DestPattern)
```

Rotates the 16x16 pixels sprite pattern n° *pattern* stored inside the VRAM. The new rotated pattern can be copied over the same pattern number by setting *DestPattern* to **0**, or copied to another pattern location by setting *DestPattern* from **1** to **255**. (Do not forget that one 16x16 pattern is in fact four 8x8 patterns inside the pattern table).

rotation indicates the angle, it can be:

90: 90° rotation to the right

-90: 90° rotation to the left

Pattern8RotationVram**V1.3****VDP_SPRITES**

```
void Pattern8RotationVram (char pattern, signed char rotation, char DestPattern)
```

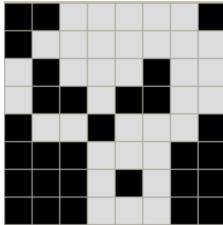
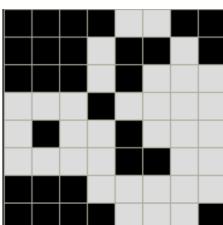
Rotates the 8x8 pixels sprite pattern n° *pattern* stored inside the VRAM. The new rotated pattern can be copied over the same pattern number by setting *DestPattern* to **0**, or copied to another pattern location by setting *DestPattern* from **1** to **255**.

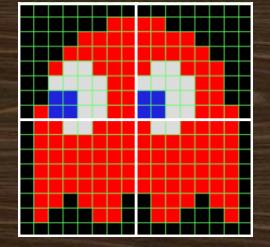
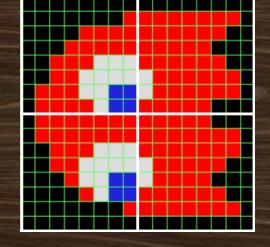
rotation indicates the angle, it can be:

90: 90° rotation to the right

-90: 90° rotation to the left

Note about Pattern Rotation and Pattern flip: They can be applied to Sprites Pattern in RAM, but also to any 8x8 array, like fonts characters, or any bitmap data stored in RAM.

Pattern8RotationRam	V1.3	VDP_SPRITES
Void Pattern8RotationRam (char pattern, char *SrcPattern, signed int rotation)		
Rotates the 8x8 pixels sprite pattern n° <i>pattern</i> stored inside the RAM at address <i>*SrcPattern</i> . The new pattern is sent to the VRAM and replace the old one. Original pattern stored in RAM is not modified.		
<i>rotation</i> indicates the angle, it can be		
- 90: 90° rotation to the right		
- -90: 90° rotation to the left		
- 180: 180° rotation)		
	90° (right) rotation ...	

Pattern16RotationRam	V1.3	VDP_SPRITES
void Pattern16RotationRam (char pattern, char *SrcPattern, signed int rotation)		
Rotates the 16x16 pixels sprite pattern n° <i>pattern</i> stored inside the RAM at address <i>*SrcPattern</i> . The new pattern is sent to the VRAM and replace the old one. The original pattern stored in RAM is not modified. <i>rotation</i> indicates the angle, it can be:		
- 90: 90° rotation to the right)		
- -90: 90° rotation to the left)		
- 180: 180° rotation)		
	-90° (left) rotation ...	

Pattern8FlipRam	V1.3	VDP_SPRITES
void Pattern8FlipRam (char pattern, char *SrcPattern, char direction)		
Flips the 8x8 pixels sprite pattern n° <i>pattern</i> stored inside the RAM at address <i>*SrcPattern</i> . The new pattern is sent to the VRAM and replace the old one. The original pattern stored in RAM is not modified.		
<i>direction</i> indicates if the direction of the flip.		
0: horizontal flip		
1: vertical flip		

Pattern16FlipRam**V1.3****VDP_SPRITES**

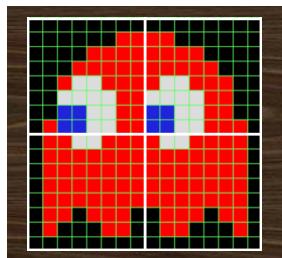
void Pattern16FlipRam (char pattern, char *SrcPattern, char direction)

Flips the 16x16 pixels sprite pattern n° **pattern** stored inside the RAM at address ***SrcPattern**. The new pattern is sent to the VRAM and replace the old one. The original pattern stored in RAM is not modified.

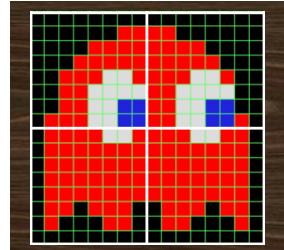
direction indicates if the direction of the flip.

0: horizontal flip

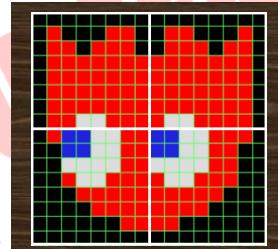
1: vertical flip



Horizontal flip ...



Vertical flip ...

**Pattern8FlipVram****V1.3****VDP_SPRITES**

void Pattern8FlipVram (char pattern, char direction, char DestPattern)

Flips the 8x8 pixels sprite pattern n° **pattern** stored inside the VRAM. The new flipped pattern can be copied over the same pattern number by setting **DestPattern** to **0**, or copied to another pattern location by setting **DestPattern** from **1** to **255**.

direction indicates the direction of the flip.

0: horizontal flip

1: vertical flip

Pattern16FlipVram**V1.3****VDP_SPRITES**

void Pattern16FlipVram (char pattern, char direction, char DestPattern)

Flips the 16x16 pixels sprite pattern n° **pattern** stored inside the VRAM. The new flipped pattern can be copied over the same pattern number by setting **DestPattern** to **0**, or copied to another pattern location by setting **DestPattern** from **1** to **255**. (Do not forget that one 16x16 pattern is in fact four 8x8 patterns inside the pattern table).

direction indicates the direction of the flip.

0: horizontal flip

1: vertical flip

SpriteFollow**V1.3*****VDP_SPRITES***

void <i>SpriteFollow</i> (void *SpriteStruct)
--

This function reads and decodes the compressed relative coordinates exported by the **Fusion-C Sprite Path tool**.

First you must define a structure like this one :

```
typedef struct {
    char y;           // important Keep this order
    char x;
    char Data;
    // ...
} DEF_SPRITE;

static DEF_SPRITE sprite1;
sprite1.Data=path1[n];
SpriteFollow(&	sprite1);
```

Second step you must send the coordinate data to the *sprite1.Data*, and call the function with the address of the sprite structure as parameter.

The function will decode coordinates and set the new X and Y position of the sprite inside the structure's sprite variable

Please see the **Sprite Path Editor**'s chapter, and the *follow.c* example.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation



CIRCLE

[vdp_circle.h]

Graphic functions to draw a circle on MSX graphic screen. Must be used with **VDP_GRAPH2.H** or **VDP_GRAPH1.H**

<i>CircleFilled</i>	MSX2	VDP_CIRCLE
<pre>void CircleFilled (int x0, int y0, int radius, int color, int OP)</pre> <p>Draws a filled circle. Center of the circle <i>at x0, y0</i>, with a <i>radius</i>, a <i>color</i> and a logical operator mode <i>OP</i>.</p>		

<i>Circle</i>	MSX2	VDP_CIRCLE
<pre>void Circle (int x0, int y0, int radius, int color, int OP)</pre> <p>Draws a circle. Center of the circle at <i>x0, y0</i>, with a <i>radius</i>, a <i>color</i> and a logical operator <i>OP</i>.</p>		

<i>SC2CircleFilled</i>	V1.1	VDP_CIRCLE
<pre>void SC2CircleFilled (char x0, char y0, char radius, char color)</pre> <p>Only for Screen 2 mode. Draws a filled circle. Center of the circle at x0, y0, with a radius, and a color</p>		

<i>SC2Circle</i>	V1.1	VDP_CIRCLE
<pre>void SC2Circle (char x0, char y0, char radius, char color)</pre> <p>Only for Screen 2 mode. Draws a filled circle. Center of the circle at x0, y0, with a radius, and a color</p>		

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

MSX-DOS 2 RAM MAPPER

[rammapper.h]

All necessary functions to be able to use full memory of the MSX computer with Memory Mapper thru the secure functions of MSX-DOS2.

InitRamMapperInfo***RAMMAPPER*****void InitRamMapperInfo(char deviceId)**

Initialization of the MSX2 Mapper device.

DeviceId must be set to 0x04 to initialize MSX-DOS2 Mapper.example: **InitRamMapperInfo(4) ;**

After Initialization, the structure is set with all data and information about all mappers found.

```
typedef struct {
    charslot ;
    charnumber16KBSegments ;
    charnumberFree16KBSegments ;
    charnumberAllocatedSystem16KBSegments ;
    charnumberUser16KBSegments ;
    charnotInUse0 ;
    charnotInUse1 ;
    charnotInUse2 ;
} MAPPERINFOBLOCK ;
```

Get_PN***RAMMAPPER*****char Get_PN (char page)**Gets and returns Segment Address of a Memory *page*.*page* must be 0,1,2 or 3***Put_PN******RAMMAPPER*****void Put_PN (char page, char segment)**Sets a specific memory *segment* to a page. *page* must be 0,1,2, or 3. *segment* must be one of the allocated segment***AllocateSegment******RAMMAPPER*****SEGMENTSTATUS *AllocateSegment (char segmentType, char slotAddress)**

Allocates next available 16Kbytes ram segment and returns information about this segment in the Status structure.

```
typedef struct {
    charallocatedSegmentNumber ;
    charslotAddressOfMapper ;
    charcarryFlag ;
} SEGMENTSTATUS ;
```

- segmentType must be **0** (Allocation of a user Segment), **1** means System segment
- slotAddress must be set to 0 for automatic allocation: **AllocateSegment(0,0) ;**

FreeSegment***RAMMAPPER******FreeSegment (char segmentType, char slotAddress)**

Free an allocated segment

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

PSG

[PSG.H]

Extended function to control the sound processor.

Sound	PSG
void Sound (char reg, char value)	[PSG.H]

Writes a value into a register of PSG (0 to 13)

SetChannelA	PSG
void SetChannelA (char channel, Boolean isTone, Boolean isNoise)	[PSG.H]

Enables or disables Tone and Noise channels (0,1 or 2). *IsNoise* must be **1** or **0**

SilencePSG	PSG
void SilencePSG (void)	[PSG.H]

Plays off all three PSG Channels

GetSound	PSG
char GetSound (char reg)	[PSG.H]

Reads a PSG Register

SetTonePeriod	PSG
void SetTonePeriod (char channel, unsigned int period)	[PSG.H]

Sets Tone Period for any channel (0,1 or 2). *period* must be between **0** and **4095**

SetNoisePeriod	PSG
void SetNoisePeriod (char period)	[PSG.H]

Sets Noise Period. *period* must be between **0** and **31**

SetEnvelopePeriod	PSG
void SetEnvelopePeriod (unsigned int period)	[PSG.H]

Sets Envelope Period. *Period* must be between **0** and **65535**

SetVolume	PSG
void SetVolume (char channel, char volume)	[PSG.H]

Sets volume of a *channel* (0,1 or 2)
volume must be between 0 and 15, or 16 to activate envelope

SetChannel	PSG
void SetChannel (char channel, Boolean isTone, Boolean isNoise)	[PSG.H]

Mixer. Enables or disables Tone and Noise channels (0,1 or 2)
Tone and State must be **0** or **1**

PlayEnvelope**PSG****void PlayEnvelope (char shape)**

Plays the sound on channels that has a volume of 16.

Envelope *shape* must be between **0** and **15*****SoundFX*****PSG****void SoundFX (char channel, FX *sounddata)**Plays a FX by a PSG Channel (0,1 or 3). *sounddata* comes from this structure

```
typedef struct {
    boolean isTone ;
    boolean isNoise ;
    unsigned int Tone ;
    char Noise ;
    unsigned int Period ;
    char Shape ;
} FX ;
```

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
 Sharing This Document without authorization
 is copyright violation

AYFX PLAYER

[ayfx_player.h]

Use the AYFX sound editor to create and edit AYFX Sound. Save the sounds as a sound bank and placed the data in RAM.

Put the data in RAM. This can be done dynamically, by loading the bank from a disk, or directly as constant data in the body of the C program.

This new driver updated with Fusion-C **V1.3** can play up to 3 different sounds at the same time, on the 3 channels of the PSG. **This driver is not compatible with the simultaneous use of the PT3 Replayer** which has its own sound effects system.

InitFX***AYFX_PLAYER*****void InitFX (void *SndBankAddr)**

Initialization of the ayFX player. ***SndBankAddress** must point to the AYFX Sound Bank stored in Ram.

PlayFX***AYFX_PLAYER*****char PlayFX (char SoundFX)**

Plays the sound n° **SoundFX** from the sound bank.

UpdateFX***AYFX_PLAYER*****void UpdateFX (void)**

Currently playing the sounds. This function must be part of the main loop of your program. It updates the PSG registers, with sound FX that must be played.

StopFX***AYFX_PLAYER*****char StopFX (void)**

Stops playing all sounds.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler

MUSIC PT3 + AYFX REPLAYER

[pt3replayer.h]

Use those function to play PT3 music files, and SoundFX from an AYFX Sound Bank. Since Fusion-C **V1.3**, the PT3 Replayer can now also reproduce FX sounds from an AYFX Sound Bank. If you wish to play music, and also sound FX in the same time, you must use these functions.

PT3Init	PT3REPLAYER
void PT3Init (char *SongAddr, char Loop) Initialization of the PT3 replayer. *SongAddr must be set with the PT3 data area in RAM. Loop must be set to : 0 : if you do not want the music loop 1 : if you want the music loop continuously	

PT3Play	PT3REPLAYER
void PT3Play (void) Actualizes the music playing, inside a main loop. This function must be executed on each interruption of VBLANK.	

PT3Rout	PT3REPLAYER
void PT3Rout (void) Prepares data to be played by the sound processor. This function must be executed on each interruption of VBLANK.	

PT3Mute	PT3REPLAYER
void PT3Mute (void) Mutes the music. To mute totally the Music, you must, first Call this function, and stop invoking PT3Play function.	

PT3FXInit	PT3REPLAYER
void PT3FXInit(void *BankAddr, char Channel) Initialisation of the AYFX sound process. *BankAddr must be set with the address of the AYFX Bank in Ram. Channel is the default PSG's channel used to play sound FX, it can be 0 , 1 or 2	

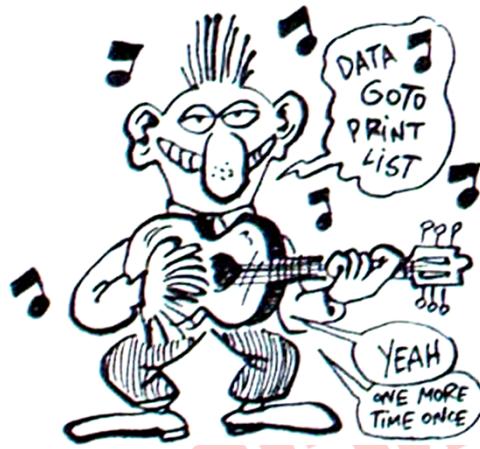
PT3FXPlay	PT3REPLAYER
void PT3FXPlay(char Sound, char Priority) Plays a souynd FX from the AYFX sound Bank. Sound represent the sound number from the bank, from 0 to 255 . Priority , represents the sound priority of the sound, from 0 to 15 . Highest priority is 0 , lowest priority is 15 . A sound with high priority will be played over a low priority sound even If it hasn't finished playing.	

PT3FXRout

PT3REPLAYER

void **PT3FXRout**(void)

Sends the sound to the sound processor. This function must be executed on each interruption of VBLANK.



PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

FUSION-C ENVIRONMENT VARIABLES

When you are Including “msx_fusion.h” in your code, it gives you access to some environment variables. You can read them anywhere in your code, without any previous declaration.

<i>_SpriteOn</i>	V1.3
Return 0 when sprites are activated, 1 when sprites are deactivated. (8-bit value)	

<i>_SpriteSize</i>	V1.3
Return 0 when sprite size is 8x8, 1 when size is 16x16. (8-bit value)	

<i>_SpriteMag</i>	V1.3
Return 0 when sprite size is normal, 1 when sprites are double. (8-bit value)	

<i>DisplayPage</i>	V1.3 MSX2
Returns the displayed VRAM Page, from 0 to 3 .	

<i>ActivePage</i>	V1.3 MSX2
Returns the active VRAM Page, from 0 to 3 .	

<i>VDPfreq</i>	V1.3 MSX2
Return 1 if VDP frequency is 50 Hz, 0 if VDP frequency is 60 Hz	

<i>VDPlines</i>	V1.3 MSX2
Return 1 if VDP is set to 212 lines, 0 if VDP is set to 192 lines. (8-bit value)	

<i>ForegroundColor</i>	V1.3
Returns the foreground color, from 0 to 255 .	

<i>BackgroundColor</i>	V1.3
Returns the background color, from 0 to 255 .	

<i>BoderColor</i>	V1.3
Returns border color, from 0 to 255 .	

<i>ScreenMode</i>	V1.3
Returns the current screen mode, from 0 to 12 .	

_SpritePatternAddr**V1.3**

Returns the default Sprite Pattern table address in Vram for the current screen mode, as an ***unsigned int*** value.

_SpriteAttribAddr**V1.3**

Returns the default Sprite Attributs table address in Vram for the current screen mode, as an ***unsigned int*** value.

_SpriteColorAddr**V1.3 MSX2**

Returns the default Sprite Color table address in Vram for the current screen mode, as an ***unsigned int*** value.

_WidthScreen0**V1.3 MSX2**

Returns the actual width of screen mode 0, as an ***unsigned char*** value.

_WidthScreen1**V1.3 MSX2**

Returns the actual width of screen mode 1, as an ***unsigned char*** value.

_FusionVer**V1.3**

Returns the current Fusion-C library version, as ***char*** value.

(Divide the returned value by 10 to obtain the real version number)

_FusionRev**V1.3**

Returns the current Fusion-C library revision date, as an ***unsigned int*** value.

The 5 digits represent the date coded like this :

Number of Year after 2019	Day of revision	Month of the revision
Y	DD	MM

MSX BASIC VS Fusion-C***Instructions comparison***

This part is just a quick comparison between MSX BASIC commands and possible C commands you can find in libraries.

Comparison between the two languages is a not really fair because programming in C has nothing to do with programming in MS Basic. Anyway, it can help beginners to find some usual information.

Just think C libraries are offering much, much more possibilities and easy-to-use functions than Basic.

<i>Jump and Loop</i>	
FOR ... NEXT	For (int ; condition ; increment){ <i>statement...</i> ; } Or while (condition) { <i>statement ...;</i> } Or do { <i>statement...</i> ; } while (condition);
GOSUB	N/A
GOTO	Goto LABEL; ... LABEL
RETURN	Return (n); Return a value at the end of a function
<i>Clock and Time</i>	
INTERVAL	See Interrupt fonctions
GET DATE	GetDate
GET TIME	GetTime
ON INTERVAL GOSUB	SetInterrupt, SetVDPIinterrupt
SET DATE	SetDate
SET TIME	SetTime
TIME	RealTimer, SetRealTimer
<i>Conditions</i>	
IF .. THEN ... ELSE	If, else
ON .. GOSUB	Switch
ON .. GOTO	Switch
<i>Conversions</i>	
ASC()	Itoa
BIN\$()	Use 0b1111111 format
CDBL()	IntToFloat()
CHR\$()	Replace print chr\$(x) by PrintChar
CINT()	(int)
CSNG()	IsPositive
HEX\$()	Use 0xFFFF
OCT\$()	
VAL()	Atoi
<i>Loading and Saving</i>	
BLOAD	N/A
BSAVE	N/A
CLOAD	N/A
CSAVE	N/A
LOAD	Read
SAVE	Write
CLOSE	Close
OPEN	Open
MERGE	See Read with append mode
RUN	N/A

Graphics and Screen

BASE	
CIRCLE	Circle
CLS	Cls
COLOR	SetColors
COPY	HMMC, LMMC, HMCM, LMMM, HMMM, YMMM
DRAW	Draw
LINE	Line
LOCATE	Locate
WIDTH	Width
PAINT	Paint
POINT	Point
PSET	Pset
POS()	
PRINT	Print, Printf or PrintString ...
SCREEN	Screen
SET PAGE	SetActivePage, SetDisplayPage
SET SCROLL	SetScrollH, SetScrollV
SET VIDEO	N/A
SET ADJUST	SetAdjust
VDP()	VDPwrite
VPEEK	Vpeek
VPOKE	Vpoke
INP()	Inport
OUT	Outport

KeyBoard and Controls

INKEY\$	Inkey
INPUT	InputChar, inputString
INPUT\$()	WaitKey
KEY	Inkey
KEY()	
LINE INPUT	N/A
ONB KEY GOSUB	Switch
ON STOP GOSUB	
ON STRIG GOSUB	
PAD()	MouseRead, MouseReadTo
PDL()	
STICK()	JoystickRead, JoystickReadTo
STRIG()	TriggerRead

Math

ABS	fabsf
CDBL	IntToFloat
CINT()	
CSNG	
EXP()	powf
FIX()	
INT()	ceilf, floor
LOG()	logf
RND()	rand
SGN()	IsPositive
SQR()	sqrtf
ATN()	atanf
COS()	cosf
SIN()	sinf
TAN()	tanf

Ram Access

PEEK	Peek, PeekW
POKE	Poke, PokeW

<i>Sounds</i>	
PLAY	PSGwrite
SOUND	PSGwrite
BEEP	Beep
CALL PCMPLAY	PCMPlay

<i>Sprites</i>	
COLOR SPRITE()	SpriteColors
COLOR SPRITES()	SpriteColors
ON SPRITE GOSUB	SpriteCollision
PUT SPRITE	PutSprite, fPutSprite
SPRITE	N/A
SPRITE\$()	SetSpritePattern

<i>Strings</i>	
INSTR()	StrPosStr
LEFT\$	StrLeftTrim
MID\$	
RIGHT\$	StrRightTrim
SPACE\$	
STRING\$	

<i>Variables settings</i>	
CLEAR	N/A
DATA	N/A
DIM	Malloc
ERASE	Free
LET	N/A
READ	N/A
RESTORE	N/A
SWAP	IntSwap, StrReverse

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

The Library's source code

The FUSION-C Library is provided with all source codes you can find all files in the folder «**Fusion-c/Lib/Sources/**».

There are two types of files. The «**.s**» files are source files in assembler ; the «**.c**» are source files in C. You are free to modify any file you want, and add your own code, and even complete the library with your own routines and functions.

To add functions to FUSION-C, just add your source file in the «**source/lib**» folder, and launch the compilation script: «**_build_lib.sh**» or «**_build_lib.bat**» if you are using Windows. The compilation script will build a new library, and copy it at the right place. In case of error, you will be warned, and details of the error can be read in the log file.

*Do not forget to add your function to the appropriate header file (**.h**).*

Please, share your work. Your needs can be the needs of other coders, so if you do modifications or add functions, send us your files and documentation, we will include your work in an upcoming version of the FUSION-C Library.

For your information, here the instruction used to compile assembler source file:

```
> sdasz80 -o <filename.s>
```

This instruction is used to compile C source file:

```
> sdcc -use-stdout -mz80 -c <filename.c>
```

The compiler will generate a «**.rel**» with the same name as the original source code filename.

This instruction is used to include a compiled source code into the library:

```
> sdar -rc fusion.lib <filename>.rel
```

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

The Source code catalog

If you need to modify or just look how a function is coded, this catalog will help you identify in which file you can find the source code of a function.

<i>FUSION-C 1.3</i>	R10106		
<i>Function's name</i>	<i>Language</i>	<i>Source file</i>	<i>Computer</i>
<i>AllocateSegment</i>	ASM	<i>rammapper.s</i>	
<i>Beep</i>	ASM	<i>CallBios_Functions.s</i>	
<i>BitReset</i>	C	<i>bit.c</i>	
<i>BitReturn</i>	C	<i>bit.c</i>	
<i>BoxFill</i>	C	<i>vdp_graph2plus.c</i>	MSX2
<i>BoxLine</i>	C	<i>vdp_graph2plus.c</i>	MSX2
<i>CallBios</i>	ASM	<i>call.c</i>	
<i>CallDos</i>	ASM	<i>call.c</i>	
<i>CallSub</i>	ASM	<i>call.c</i>	MSX2
<i>ChangeCap</i>	ASM	<i>CallBios_Functions.s</i>	
<i>ChangeCPU</i>	ASM	<i>CallBios_Functions.s</i>	Turbo-R
<i>ChangeDir</i>	ASM	<i>io.s</i>	
<i>CharToLower</i>	ASM	<i>ctype.s</i>	
<i>CharToUpper</i>	ASM	<i>ctype.s</i>	
<i>CheckBreak</i>	ASM	<i>printhex.s</i>	
<i>Circle</i>	C	<i>circle.c</i>	MSX2
<i>CircleFilled</i>	C	<i>circle.c</i>	MSX2
<i>Clear1px</i>	ASM	<i>vdp_graph1.s</i>	
<i>Clear8px</i>	ASM	<i>vdp_graph1.s</i>	
<i>Close</i>	ASM	<i>io.s</i>	
<i>Cls</i>	ASM	<i>CallBios_Functions.s</i>	
<i>CopyRamToVram</i>	ASM	<i>Vram.s</i>	
<i>CopyVramToRam</i>	ASM	<i>Vram.s</i>	
<i>CovoxPlayRam</i>	ASM	<i>covoxplay.c</i>	
<i>CovoxPlayVram</i>	C & ASM	<i>covoxplay.c</i>	MSX2
<i>Create</i>	ASM	<i>io.s</i>	
<i>CreateAttrib</i>	ASM	<i>io.s</i>	
<i>DisableInterrupt</i>	#define	<i>msx_fusion.h</i>	
<i>DiskLoad</i>	ASM	<i>io.s</i>	
<i>Draw</i>	ASM	<i>vdp_graph2.s</i>	MSX2
<i>EnableInterrupt</i>	#define	<i>msx_fusion.h</i>	
<i>EndInterruptHandler</i>	ASM	<i>interrupt.s</i>	
<i>EndVDPIInterruptHandler</i>	ASM	<i>interrupt vdp.s</i>	
<i>Exit</i>	ASM	<i>CallDos_Functions.s</i>	
<i>FcbClose</i>	ASM	<i>Fcb_access.s</i>	
<i>FcbCreate</i>	ASM	<i>Fcb_access.s</i>	
<i>FcbDelete</i>	ASM	<i>fcb_access.s</i>	
<i>FcbFindFirst</i>	ASM	<i>Fcb_access.s</i>	
<i>FcbFindNext</i>	ASM	<i>Fcb_access.s</i>	
<i>FCBlist</i>	ASM	<i>Fcb_access.s</i>	
<i>FcbOpen</i>	ASM	<i>Fcb_access.s</i>	
<i>FcbRead</i>	ASM	<i>Fcb_access.s</i>	
<i>FcbWrite</i>	ASM	<i>Fcb_access.s</i>	
<i>FillVram</i>	ASM	<i>CallBios_Functions.s</i>	
<i>FindFirst</i>	ASM	<i>io.s</i>	
<i>FindNext</i>	ASM	<i>io.s</i>	
<i>Fkeys</i>	ASM	<i>keyboardreads.s</i>	
<i>fLMMM</i>	ASM	<i>vdp_graph2.s</i>	MSX2
<i>fPutSprite</i>	ASM	<i>vram.s</i>	
<i>FreeSegment</i>	ASM	<i>rammapper.s</i>	
<i>FunctionKeys</i>	ASM	<i>callBios_Functions.s</i>	
<i>fVDP</i>	ASM	<i>vdp_graph2.s</i>	MSX2
<i>Get_PN</i>	ASM	<i>rammapper.s</i>	

C Library for MSX-DOS with SDCC compiler

<i>GetIpx</i>	ASM	vdp_graph1.s	
<i>Get8px</i>	ASM	vdp_graph1.s	
<i>Getche</i>	ASM	getche.s	
<i>GetCol8px</i>	ASM	vdp_graph1.s	
<i>GetCPU</i>	ASM	CallBios_Functions.s	Turbo-R
<i>GetCWD</i>	ASM	io.s	
<i>GetDate</i>	ASM	Calldos_Functions.s	
<i>GetDisk</i>	ASM	CallDos_Functions.s	
<i>GetDiskParam</i>	ASM	CallDos_Functions.s	
<i>GetDiskTrAddress</i>	ASM	CallDos_Functions.s	
<i>GetKeyMatrix</i>	C	msx_fusion.h	
<i>GetOSVersion</i>	ASM	io.s	
<i>GetSound</i>	ASM	psg.c	
<i>GetTime</i>	ASM	CallDos_Functions.s	
<i>GetVramSize</i>	ASM	Vram.s	
<i>Halt</i>	#define	msx_fusion.h	
<i>HideDisplay</i>	ASM	CallBios_Functions.s	
<i>HMCM</i>	ASM	vdp_graph2.s	MSX2
<i>HMCM_SC8</i>	ASM	vdp_graph2.s	MSX2
<i>HMMC</i>	ASM	vdp_graph2.s	MSX2
<i>HMMM</i>	ASM	vdp_graph2.s	MSX2
<i>HMMV</i>	ASM	vdp_graph2.s	MSX2
<i>InitFX</i>	C	ayfxDriver.s	
<i>InitInterruptHandler</i>	ASM	interrupt.s	
<i>InitPSG</i>	ASM	CallBios_Functions.s	
<i>InitRamMapperInfo</i>	ASM	rammapper.s	
<i>Inkey</i>	ASM	callBios_Functions.s	
<i>InPort</i>	ASM	port_in-out.s	
<i>InputChar</i>	ASM	callBios_Functions.s	
<i>InputString</i>	ASM	inputstring.s	
<i>IntSwap</i>	C	intswap.c	
<i>IntToFloat</i>	C	inttofloat.c	
<i>IsAlpha</i>	ASM	ctype.s	
<i>IsAlphaNum</i>	ASM	ctype.s	
<i>IsAscii</i>	ASM	ctype.s	
<i>IsCtrl</i>	ASM	ctype.s	
<i>IsDigit</i>	ASM	ctype.s	
<i>IsGraph</i>	ASM	ctype.s	
<i>IsHexDigit</i>	ASM	ctype.s	
<i>IsHsync</i>	#define	msx_fusion.h	
<i>IsLower</i>	ASM	ctype.s	
<i>IsPositive</i>	C	ispositive.c	
<i>IsPrintable</i>	ASM	ctype.s	
<i>IsPunctuation</i>	ASM	ctype.s	
<i>IsSpace</i>	ASM	ctype.s	
<i>IsUpper</i>	ASM	ctype.s	
<i>IsVsync</i>	#define	msx_fusion.h	
<i>Itoa</i>	C	itoa.c	
<i>JoystickRead</i>	ASM	CallBios_Functions.s	
<i>JoystickReadTo</i>	ASM	Joystick_readTo.s	
<i>KeySound</i>	#define	msx_fusion.h	
<i>KillKeyBuffer</i>	ASM	CallBios_Functions.s	
<i>Line</i>	C	vdp_graph2plus.c	MSX2
<i>LMMC</i>	ASM	vdp_graph2.s	MSX2
<i>LMMM</i>	ASM	vdp_graph2.s	MSX2
<i>LMMV</i>	ASM	vdp_graph2.s	MSX2
<i>Locate</i>	ASM	CallBios_Functions.s	
<i>Lseek</i>	ASM	io.s	
<i>Ltell</i>	ASM	io.s	
<i>MakeDir</i>	ASM	io.s	

C Library for MSX-DOS with SDCC compiler

<i>MemChr</i>	ASM	<i>memchr.s</i>
<i>MemCompare</i>	ASM	<i>memcompare.s</i>
<i>MemCopy</i>	ASM	<i>memcpy.s</i>
<i>MemCopyReverse</i>	ASM	<i>memcpyreverse.s</i>
<i>MemFill</i>	ASM	<i>memfill.s</i>
<i>MMalloc</i>	C	<i>mmalloc.c</i>
<i>MouseRead</i>	ASM	<i>mouseread.c</i>
<i>MouseReadTo</i>	ASM	<i>mousereadto.c</i>
<i>NStrCompare</i>	ASM	<i>nstrcompare.s</i>
<i>NStrConcat</i>	ASM	<i>nstrconcat.s</i>
<i>NStrCopy</i>	ASM	<i>nstrcpy.s</i>
<i>Open</i>	ASM	<i>io.s</i>
<i>OpenAttrib</i>	ASM	<i>io.s</i>
<i>OutPort</i>	ASM	<i>port in-out.s</i>
<i>OutPorts</i>	ASM	<i>port in-out.s</i>
<i>Paint</i>	ASM & C	<i>vdp_paint.h</i> MSX2
<i>Pattern16FlipRam</i>	C	<i>PatternTransform.c</i>
<i>Pattern16FlipVram</i>	C	<i>PatternTransform.c</i>
<i>Pattern16RotationRam</i>	C	<i>PatternTransform.c</i>
<i>Pattern16RotationVram</i>	C	<i>PatternTransform.c</i>
<i>Pattern8FlipRam</i>	C	<i>PatternTransform.c</i>
<i>Pattern8FlipVram</i>	C	<i>PatternTransform.c</i>
<i>Pattern8RotationRam</i>	C	<i>PatternTransform.c</i>
<i>Pattern8RotationVram</i>	C	<i>PatternTransform.c</i>
<i>PatternHFlip</i>	ASM	<i>PatternTransform.c</i>
<i>PatternRotation</i>	ASM	<i>PatternTransform.c</i>
<i>PatternVFlip</i>	ASM	<i>PatternTransform.c</i>
<i>PCMPlay</i>	ASM	<i>CallBios Functions.s</i> Turbo-R
<i>Peek</i>	#define	<i>msx_fusion.h</i>
<i>Peekw</i>	#define	<i>msx_fusion.h</i>
<i>PlayEnvelope</i>	C	<i>psg.c</i>
<i>PlayFX</i>	ASM	<i>ayfxDriver.s</i>
<i>Point</i>	ASM	<i>vdp_graph2.s</i> MSX2
<i>Poke</i>	#define	<i>msx_fusion.h</i>
<i>Pokew</i>	#define	<i>msx_fusion.h</i>
<i>Polygon</i>	C	<i>vdp_graph1plus.c</i> MSX2
<i>Print</i>	C	<i>print.c</i>
<i>PrintChar</i>	ASM	<i>callBios Functions.s</i>
<i>PrintDec</i>	ASM	<i>printdec.s</i>
<i>printf</i>	C	<i>printf-msx.c</i>
<i>PrintFNumber</i>	C	<i>printfnumber.c</i>
<i>PrintHex</i>	ASM	<i>printhex.s</i>
<i>PrintNumber</i>	C	<i>printfnumber.c</i>
<i>Pset</i>	ASM	<i>vdp_graph2.s</i> MSX2
<i>PSGread</i>	C	<i>psg.c</i>
<i>PSGwrite</i>	C	<i>psg.c</i>
<i>PT3FXInit</i>	ASM	<i>pt3replayer.s</i>
<i>PT3FXPlay</i>	ASM	<i>pt3replayer.s</i>
<i>PT3FXRout</i>	ASM	<i>pt3replayer.s</i>
<i>PT3Init</i>	ASM	<i>pt3replayer.s</i>
<i>PT3Mute</i>	ASM	<i>pt3replayer.s</i>
<i>PT3Play</i>	ASM	<i>pt3replayer.s</i>
<i>PT3Rout</i>	ASM	<i>pt3replayer.s</i>
<i>Put_PN</i>	ASM	<i>rammapper.s</i>
<i>PutCharHex</i>	ASM	<i>printhex.s</i>
<i>PutSprite</i>	ASM	<i>Vram.s</i>
<i>PutText</i>	ASM	<i>CallBios Functions.s</i>
<i>Read</i>	ASM	<i>io.s</i>
<i>ReadAdjust</i>	C	<i>vpoke-vpeek.c</i> MSX2
<i>ReadBlock</i>	ASM	<i>sc2block.s</i>

C Library for MSX-DOS with SDCC compiler

<i>ReadKeyboardType</i>	ASM	callBios_Functions.s	
<i>ReadMSXtype</i>	C	readmsxtype.c	
<i>ReadSP</i>	ASM	readsp.s	
<i>ReadTPA</i>	#define	msx_fusion.h	
<i>RealTimer</i>	#define	msx_fusion.h	
<i>Remove</i>	ASM	io.s	
<i>RemoveDir</i>	ASM	io.s	
<i>Rename</i>	ASM	io.s	
<i>RestorePalette</i>	ASM	Set_Palette.s	MSX2
<i>Rkeys</i>	ASM	keyboardread.s	
<i>RleWBToRam</i>	ASM	RLEwb_toram.c	
<i>RleWBToVram</i>	C & ASM	RLEwb_tovram.c	MSX2
<i>SaveScreenBoot</i>	C	call.c	MSX2
<i>SC2BoxFill</i>	C	vdp_graph1plus.c	
<i>Sc2BoxLine</i>	C	vdp_graph1plus.c	
<i>SC2Circle</i>	C	circle.c	
<i>SC2CircleFilled</i>	C	circle.c	
<i>SC2Draw</i>	ASM	vdp_graph1.s	
<i>SC2Line</i>	C	vdp_graph1plus.c	
<i>SC2Paint</i>	ASM	vdp_graph1.s	
<i>SC2Point</i>	ASM	vdp_graph1plus.c	
<i>SC2Pset</i>	ASM	vdp_graph1plus.c	
<i>SC2ReadScr</i>	ASM	readwritescr.s	
<i>SC2WriteScr</i>	ASM	readwritescr.s	
<i>Screen</i>	ASM	CallBios_Functions.s	
<i>SectorRead</i>	ASM	CallDos_Functions.s	
<i>SectorWrite</i>	ASM	CallDos_Functions.s	
<i>SetIpx</i>	ASM	vdp_graph1.s	
<i>Set8px</i>	ASM	vdp_graph1.s	
<i>SetActivePage</i>	#define	msx_fusion.h	
<i>SetAdjust</i>	C	vpoke-vpeek.c	MSX2
<i>SetBorderColor</i>	ASM	setbordercolor.s	
<i>SetChannel</i>	ASM	psg.c	
<i>SetCol8px</i>	ASM	vdp_graph1.s	
<i>SetColor</i>	C	msx_fusion.h	
<i>SetColorPalette</i>	ASM	Set_Palette.s	MSX2
<i>SetColors</i>	ASM	CallBios_Functions.s	
<i>SetDate</i>	ASM	CallDos_Functions.s	
<i>SetDisk</i>	ASM	CallDos_Functions.s	
<i>SetDiskTrAddress</i>	ASM	CallDos_Functions.s	
<i>SetDisplayPage</i>	ASM	VDPWrite_functions.s	
<i>SetEnvelopePeriod</i>	C	psg.c	
<i>SetExpandVDPCmd</i>	ASM	VDPWrite_functions.s	MSX2+
<i>SetInterruptHandler</i>	ASM	interrupt.s	
<i>SetNoisePeriod</i>	C	psg.c	
<i>SetPaintBuffer</i>	C	vdp_paint.h	MSX2
<i>SetPalette</i>	ASM	Set_Palette.s	MSX2
<i>SetRamDisk</i>	ASM	callBios_Functions.s	
<i>SetRealTimer</i>	#define	msx_fusion.h	MSX2
<i>SetScreen10</i>	ASM	VDPWrite_functions.s	MSX2+
<i>SetScreen12</i>	ASM	VDPWrite_functions.s	MSX2+
<i>SetScrollDouble</i>	ASM	VDPWrite_functions.s	MSX2+
<i>SetScrollIH</i>	ASM	VDPWrite_functions.s	MSX2+
<i>SetScrollMask</i>	ASM	VDPWrite_functions.s	MSX2+
<i>SetScrollV</i>	ASM	VDPWrite_functions.s	MSX2
<i>SetSpriteColors</i>	C	setspritepattern.c	
<i>SetSpritePattern</i>	C	setspritepattern.c	
<i>SetTime</i>	ASM	CallDos_Functions.s	
<i>SetTonePeriod</i>	C	psg.c	
<i>SetTransparent</i>	ASM	VDPWrite_functions.s	MSX2

C Library for MSX-DOS with SDCC compiler

<i>SetVDPInterruptHandler</i>	ASM	interrupt_vdp.s	
<i>SetVDPread</i>	ASM	Vram.s	
<i>SetVDPwrite</i>	ASM	Vram.s	
<i>SetVolume</i>	C	psg.c	
<i>ShowDisplay</i>	ASM	CallBios_Functions.s	
<i>SilencePSG</i>	C	psg.c	
<i>Sound</i>	ASM	psg.c	
<i>SoundFX</i>	C	psg.c	
<i>Sprite16</i>	ASM	VDPWrite_functions.s	
<i>Sprite32Bytes</i>	ASM	sprite32bytes.s	
<i>Sprite8</i>	ASM	VDPWrite_functions.s	
<i>SpriteCollision</i>	#define	vdp_sprites.h	
<i>SpriteCollisionX</i>	C	spritecollision.c	MSX2
<i>SpriteCollisionY</i>	C	spritecollision.c	MSX2
<i>SpriteDouble</i>	ASM	VDPWrite_functions.s	
<i>SpriteFollow</i>	ASM	spritefollow.c	
<i>SpriteOff</i>	ASM	VDPWrite_functions.s	
<i>SpriteOn</i>	ASM	VDPWrite_functions.s	
<i>SpriteOverLap</i>	#define	VDP_sprites.h	
<i>SpriteOverLapId</i>	#define	VDP_sprites.h	
<i>SpriteReset</i>	ASM	CallBios_Functions.s	
<i>SpriteSmall</i>	ASM	VDPWrite_functions.s	
<i>StopFX</i>	ASM	ayfxDriver.s	
<i>StrChr</i>	ASM	strchr.s	
<i>StrCompare</i>	ASM	strcmp.s	
<i>StrConcat</i>	ASM	strconcat.s	
<i>StrCopy</i>	ASM	strcpy.s	
<i>StrLeftTrim</i>	ASM	strlfttrim.s	
<i>StrLen</i>	ASM	strlfttrim.s	
<i>StrPosChr</i>	ASM	strposchr.s	
<i>StrPosStr</i>	ASM	strposstr.s	
<i>StrReplaceChar</i>	ASM	strreplacechar.s	
<i>StrReverse</i>	C	strreverse.c	
<i>StrRightTrim</i>	ASM	strrighttrim.s	
<i>StrSearch</i>	ASM	strsearch.s	
<i>StrToLower</i>	ASM	ctype.s	
<i>StrToUpper</i>	ASM	ctype.s	
<i>Suspend</i>	ASM	msx_fusion.h	
<i>TriggerRead</i>	ASM	CallBios_Functions.s	
<i>TurboMode</i>	C	vpoke-vpeek.c	
<i>UpdateFX</i>	ASM	ayfxDriver.s	
<i>VDP50Hz</i>	ASM	VDPWrite_functions.s	MSX2
<i>VDP60Hz</i>	ASM	VDPWrite_functions.s	MSX2
<i>VDPAlternate</i>	C	vpoke-vpeek.c	MSX2
<i>VDPInterlace</i>	C	vpoke-vpeek.c	MSX2
<i>VDPLINE</i>	ASM	vdp_graph2.s	MSX2
<i>VDPLineSwitch</i>	ASM	VDPWrite_functions.s	MSX2
<i>VDPstatus</i>	ASM	vdpstatus.s	
<i>VDPstatusNi</i>	ASM	vdpstatus.s	
<i>VDPwrite</i>	ASM	VDPWrite_functions.s	
<i>VDPwriteNi</i>	ASM	VDPWrite_functions.s	
<i>vMSX</i>	ASM	vdp_graph2.s	
<i>Vpeek</i>	C	vpoke-vpeek.c	
<i>VpeekFirst</i>	#define	msx_fusion.h	
<i>VpeekNext</i>	#define	msx_fusion.h	
<i>Vpoke</i>	C	vpoke-vpeek.c	
<i>VpokeFirst</i>	#	msx_fusion.h	
<i>VpokeNext</i>	#define	msx_fusion.h	
<i>Vsynch</i>	#define	msx_fusion.h	
<i>WaitKey</i>	ASM	CallBios_Functions.s	

<i>Width</i>	C	vpoke-vpeek.c
<i>Write</i>	ASM	io.s
<i>WriteBlock</i>	ASM	sc2block.s
<i>YMMM</i>	ASM	vdp_graph2.s

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

The C standard functions (included in SDCC package)

<u>CTYPE.H</u>	
Int isalnum (int c)	Check whether the given character is alphanumeric
Int isalpha(int c)	Check whether the given character is alphabetic
Int iscntrl(int c)	Check whether the given character is a control character
Int isdigit(int c)	Check whether the given character is decimal digit
Int isgraph(int c)	Check whether the given character has graphical representation
islower(int c)	Check if given character is in lower case
isupper (int c)	Check if given character is in upper case
ispunct (int c)	Check if given character is punctuation
isspace (int c)	Check if given character is a space
isxdigit (int c)	Check if given character is a Hexadecimal digit
isblank (int c)	Check if given character is blank
tolower (int c)	Convert given character to lower case
toupper (int c)	Convert given character to upper case

PRIVATE DOCUMENT

DO NOT SHARE THIS DOCUMENT

**Sharing This Document without authorization
is copyright violation**

MATH.H

sinf(float x)	Return the sine of a radian angle
cosf(float x)	Return the cosine of a radian angle
tanf(float x)	Return the tangent of X
asinf(float x)	Return the arc sine of X radian
acosf(float x)	Return the arc cosine of X
atanf(float x)	Return the arc tangent of X
atan2f(float x, float y)	Returns the arc tangent in radians of y/x based on the signs of both values to determine the correct quadrant.
Sinhf(float x)	Return the hyperbolic sine of X
coshf(float x)	Return the hyperbolic cosine of X
tanhf(float x)	Return the hyperbolic tangent of X
expf(float x)	Return the value of x raised to Xth power
logf(float x)	Return the natural logarithm of x
log10f(float x)	Returns the common logarithm (base-10 logarithm) of X
powf(float x, float y)	Returns X raised to the power of Y
sqrtf(float a)	Returns the square root of X
fabsf(float x)	Returns the absolute value of x
frexpf(float x, int *pw2)	The returned value is the mantissa and the integer pointed to by exponent is the exponent. The resultant value is x = mantissa * 2 ^ exponent.
Ldexpf(float x, int pw2)	Returns x multiplied by 2 raised to the power of exponent.
Ceilf(float x)	Returns the smallest integer value greater than or equal to X
floorf(float x)	Returns the largest integer value less than or equal to X
modff(float x, float * y)	Returns the remainder of X divided by Y

STDIO.H**Library Variable & Description**

1	size_t
	This is the unsigned integral type and is the result of the sizeof keyword.
2	FILE
	This is an object type suitable for storing information for a file stream.
3	fpos_t
	This is an object type suitable for storing any position in a file.

Macro & Description

1	NULL
	This macro is the value of a null pointer constant.
2	_IOFBF, _IOLBF and _IONBF
	These are the macros which expand to integral constant expressions with distinct values and suitable for the use as third argument to the setvbuf function.
3	BUFSIZ
	This macro is an integer, which represents the size of the buffer used by the setbuf function.

4	EOF This macro is a negative integer, which indicates that the end-of-file has been reached.
5	FOPEN_MAX This macro is an integer, which represents the maximum number of files that the system can guarantee to be opened simultaneously.
6	FILENAME_MAX This macro is an integer, which represents the longest length of a char array suitable for holding the longest possible filename. If the implementation imposes no limit, then this value should be the recommended maximum value.
7	L_tmpnam This macro is an integer, which represents the longest length of a char array suitable for holding the longest possible temporary filename created by the tmpnam function.
8	SEEK_CUR, SEEK_END, and SEEK_SET These macros are used in the fseek function to locate different positions in a file.
9	TMP_MAX This macro is the maximum number of unique filenames that the function tmpnam can generate.
10	stderr, stdin, and stdout These macros are pointers to FILE types which correspond to the standard error, standard input, and standard output streams.

Functions

printf (const char *,...)	Sends formatted output to stdout
vprintf (const char *, va_list)	Sends formatted output to a stream using an argument list.
163nitia (char *, const char *, ...)	Sends formatted output to a string.
Vsprintf (char *, const char *, va_list)	Sends formatted output to a string using an argument list.
Puts(const char *)	Writes a string to stdout up to but not including the null character. A newline character is appended to the output.
*gets(char *)	Reads a line from stdin and stores it into the string pointed. It stops when either the newline character is read
getchar(void)	Gets a character (an unsigned char) from stdin.
Putchar(char)	Writes a character (an unsigned char) specified by the argument char to stdout.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
 Sharing This Document without authorization
 is copyright violation

STDLIB.H

Library Variable & Description	
1	size_t This is the unsigned integral type and is the result of the sizeof keyword.
2	wchar_t This is an integer type of the size of a wide character constant.
3	div_t This is the structure returned by the div function.
4	ldiv_t This is the structure returned by the ldiv function.

Library Macro & Description	
1	NULL This macro is the value of a null pointer constant.
2	EXIT_FAILURE This is the value for the exit function to return in case of failure.
3	EXIT_SUCCESS This is the value for the exit function to return in case of success.
4	RAND_MAX This macro is the maximum value returned by the rand function.
5	MB_CUR_MAX This macro is the maximum number of bytes in a multi-byte character set which cannot be larger than MB_LEN_MAX.

Functions

atof(const char *nptr)	Converts the string pointed to, by the argument <i>str</i> to a floating-point number (type double).
atoi(const char *nptr)	Converts the string pointed to, by the argument <i>str</i> to an integer (type int).
164niti(const char *nptr)	Converts the string pointed to, by the argument <i>str</i> to a long integer (type long int).
_itoa(int x, char p*, charbase)	Convert the x integer number to a string pointer, on base 2,10, or 8
_ultoa(unsigned long x, char p*, charbase)	Convert the x unsigned long number to a string pointer, on base 2,10, or 8
_ltoa(long x, char p*, charbase)	Convert the x long number to a string pointer, on base 2,10, or 8
rand(void)	Returns a pseudo-random number in the range of 0 to <i>RAND_MAX</i> .
srand(unsigned int seed)	This function seeds the random number generator used by the function rand

calloc (size_t nmemb, size_t size)	Allocates the requested memory with requested size and returns a pointer to it. Calloc initializes memory cells to 0.
Malloc (size_t size)	Allocates the requested memory and returns a pointer to it
realloc (void *ptr, size_t size)	Attempts to resize the memory block pointed to by <i>ptr</i> that was previously allocated with a call to <i>malloc</i> or <i>calloc</i> .
Free (void * ptr)	Deallocates the memory previously allocated by a call to <i>calloc</i> , <i>malloc</i> , or <i>realloc</i> .
Abs(int j)	Returns the absolute value of x.

STRING.H

Library Variable & Description	
1	size_t This is the unsigned integral type and is the result of the sizeof keyword.

Library Macro & Description	
1	NULL This macro is the value of a null pointer constant.

Functions

memcpy (void * dest, const void *src, size_t n)	Copies n characters from <i>src</i> to <i>dest</i> .
memmove (void *dest, const void *src, size_t n)	Another function to copy n characters from <i>str2</i> to <i>str1</i> .
strcpy (char *dest, const char *src)	Copies the string pointed to, by <i>src</i> to <i>dest</i> .
strncpy(char * dest, const char * src, size_t n)	Copies up to n characters from the string pointed to, by <i>src</i> to <i>dest</i> .
strcat (char * dest, const char *src);	Appends the string pointed to, by <i>src</i> to the end of the string pointed to by <i>dest</i> .
strncat(char *dest, const char *src, size_t n)	Appends the string pointed to, by <i>src</i> to the end of the string pointed to, by <i>dest</i> up to n characters long.
memcmp (const void *s1, const void *s2, size_t n)	Compares the first n bytes of <i>str1</i> and <i>str2</i> .
strcmp (const char *s1, const char *s2)	Compares the string pointed to, by <i>str1</i> to the string pointed to by <i>str2</i> .
strncmp(const char *s1, const char *s2, size_t n)	Compares at most the first n bytes of <i>str1</i> and <i>str2</i> .
strxfrm(char *dest, const char *src, size_t n)	Transforms the first <i>n</i> characters of the string <i>src</i> into current locale and places them in the string <i>dest</i> .
memchr (const void *s, int c, size_t n)	Searches for the first occurrence of the character <i>c</i> (an unsigned char) in the first <i>n</i> bytes of the string pointed to, by the argument <i>str</i> .

strchr (const char *s, int c)	Sets the first occurrence of the character c (an unsigned char) in the string pointed to, by the argument str.
strpbrk(const char *str1, const char *str2)	Finds the first character in the string str1 that matches any character specified in str2.
strrchr(const char *str, char c)	Sets the last occurrence of the character c (an unsigned char) in the string pointed to by the argument str.
strcspn(const char *str, const char *reject)	Calculates the length of the initial segment of str1 which consists entirely of characters not in str2.
strpbrk(const char *str, const char *accept)	Finds the first character in the string str1 that matches any character specified in str2.
strrchr(const char *s, int c)	Sets the last occurrence of the character c (an unsigned char) in the string pointed to by the argument str.
strspn (const char *str, const char *accept)	Calculates the length of the initial segment of str1 which consists entirely of characters in str2.
strstr (const char *haystack, const char *needle)	Finds the first occurrence of the entire string needle (not including the terminating null character) which appears in the string haystack.
strtok (char *str, const char *delim)	Breaks string str into a series of tokens separated by delim.
memset (void *s, int c, size_t n)	Copies the character c (an unsigned char) to the first n characters of the string pointed to, by the argument str.
strlen (const char *s)	Return the length of a string

STDARG.H

Library Variable & Description

1 **va_list**

This is a type suitable for holding information needed by the three macros **va_start()**, **va_arg()** and **va_end()**.

Functions

va_start(marker, first)	This macro initializes ap variable to be used with the va_arg and va_end macros. The last_arg is the last known fixed argument being given to the function i.e. the argument before the ellipsis.
va_arg(marker, type)	This macro retrieves the next argument in the parameter list of the function with type type .
va_copy(dest, src)	
va_end(marker)	This macro allows a function with variable arguments which used the va_start macro to return. If va_end is not called before returning from the function, the result is undefined.

TIME.H

Library Variable & Description	
1	size_t This is the unsigned integral type and is the result of the sizeof keyword.
2	clock_t This is a type suitable for storing the processor time.
3	time_t This is a type suitable for storing the calendar time.
4	struct tm This is a structure used to hold the time and date.

Functions

time(time_t *t)	Calculates the current calendar time and encodes it into time_t format.
gmtime(time_t *timep)	The value of timer is broken up into the structure tm and expressed in Coordinated Universal Time (UTC) also known as Greenwich Mean Time (GMT).
mktime(struct tm *timeptr)	Converts the structure pointed to by timeptr into a time_t value according to the local time zone.
ctime(time_t *timep)	Returns a string representing the local time based on the argument timer.

C Library for MSX-DOS with SDCC compiler

Adding Assembler source code inside your C program

When writing code to be compiled with SDCC targeting Z80, assembler code fragments can be inserted in the C functions by enclosing them between the « `_asm` » and « `_endasm` » tags:

```
void DoNotDisturb()
{
    __asm
    di
    __endasm;

    DoSomething();

    __asm
    ei
    halt
    __endasm;
}
```

Adding « `_naked` » to the function definition will cause the compiler to not generate the `ret` statement at the end of the function, you will usually use it when the entire body of the function is written in assembler:

```
void TerminateProgram() __naked
{
    __asm
    ld c,#0
    jp #5
    __endasm;
}
```

Assembler code must preserve the value of register IX, all other registers can be used freely.

The Z80 assembler supplied with SDCC uses a pretty much standard syntax for the assembler source code except for the following:

- Decimal numeric constants must be preceded with `#`
- Hexadecimal numeric constants must be preceded with `#0x`
- The syntax for offsets when using index registers is `n(ix)`, where in other assemblers it's usually `(ix+n)`

The return value of a C function is passed to the caller as follows:

- Functions that return char: in the L register
- Functions that return int or a pointer: in the HL registers
- Functions that return long: in the DEHL registers

```
char GetMagicNumber() __naked
{
    __asm
    ld l,#34
    ret
    __endasm;
}

int GetMagicYear() __naked
{
    __asm
    ld hl,#1987
    ret
    __endasm;
}

long GetReallyStrongPassword() __naked
{
    __asm
    ld de,#0x1234
    ld hl,#0x5678
    ret
    __endasm;
}
```

Functions parameters are pushed to the stack before the function is called. They are pushed from right to left, so the leftmost parameter is the first one found when going up in the stack:

```
char SumTwoChars(char x, char y) __naked
{
    __asm
    ld iy,#2
    add iy,sp ;Bypass the return address of the function.

    ld l,(iy) ;x
    ld a,1(iy) ;y

    add l
    ld l,a      ;return value

    ret
    __endasm;
}
```

```
int SumCharAndInt(char x, int y) __naked
{
    __asm
    ld iy,#2
    add iy,sp

    ld e,(iy)    ;x
    ld d,#0

    ld l,1(iy)  ;y (low)
    ld h,2(iy)  ;y (high)

    add hl,de    ;return value

    ret
    __endasm;
}

long SumCharIntAndLong(char x, int y, long z) __naked
{
    __asm
    ld iy,#2
    add iy,sp

    ld c,(iy)    ;x
    ld b,#0
    ld l,1(iy)  ;y (low)
    ld h,2(iy)  ;y (high)
    add hl,bc    ;x+y

    ld a,l
    add 3(iy)    ;z (lower)
    ld l,a
    ld a,h
    adc 4(iy)
    ld h,a
    ld a,#0
    adc 5(iy)
    ld e,a
    ld a,#0
    adc 6(iy)    ;z (higher)
    ld d,a

    ret      ;return value = DEHL
    __endasm;
}
```

It is possible to call other C functions from assembler code. Just push the parameters that the function expects, call the function assembler name (it's the C name prepended with `_`), pop the parameters to restore the stack state, and act on the return value as appropriate:

```
int SumTwo(int x, int y)
{
    return x+y;
}

int SumThree(int x, int y, int z) __naked
{
    __asm

    ld iy,#2
    add iy,sp

    ld l,2(iy)
    ld h,3(iy)
    push hl      ;y for SumTwo
    ld l,(iy)
    ld h,1(iy)
    push hl      ;x for SumTwo

    call _SumTwo ;Return value in HL

    pop af      ;x
    pop af      ;y

    ld iy,#2
    add iy,sp

    ld e,4(iy)
    ld d,5(iy)  ;z
    add hl,de

    ret

    __endasm;
}
```

PR
DO
Sharing

It is possible to define pure assembler functions intended to be called exclusively from assembler code. In this case the standard parameter passing rules can be overridden:

```
//DO NOT call this function from C code!
int SumHLDE() __naked
{
    __asm
    add hl,de
    ret
    __endasm;
}

int SumThree(int x, int y, int z) __naked
{
    __asm
    ld iy,#2
    add iy,sp

    ld l,2(iy)
    ld h,3(iy)
    ld e,(iy)
    ld d,1(iy)

    call _SumHLDE

    ld e,4(iy)
    ld d,5(iy) ;z
    add hl,de

    ret
    __endasm;
}
```



C Library for MSX-DOS with SDCC compiler

Debugging for advanced users

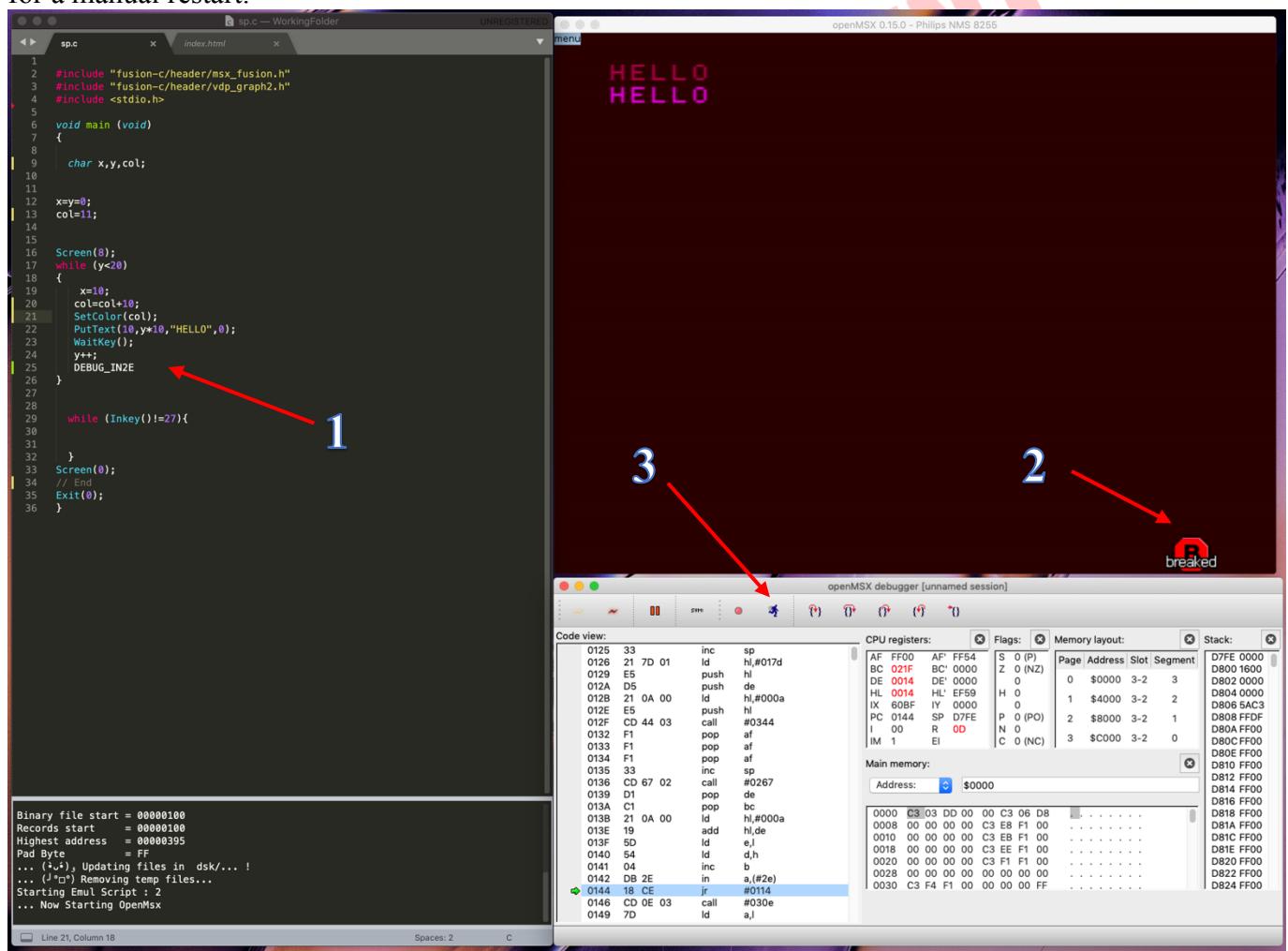
If you have enough knowledge about MSX and assembler programming, you can use the openMSX Debugger to find problems, or to improve your C program. If you think you are not strong enough, make a little try, take a look to the debugger, it will certainly be useful for you too.

Note for MacOS users : You will need to install QT library before using the debugger. Open your Shell/Terminal windows and install the QT library with this command :

```
> Brew install qt5
```

Launch the debugger which is in **./WorkingFolder/openMSX/** then connect it to the active openMSX session, and you will see in live everything that happens in the MSX memory, in the registers as well as in the VRAM.

Now it may be useful to break the program you are coding at a specific place. For that, FUSION-C has a special directive included. By adding this command inside your source code : **DEBUG_IN2E** you are indicating to openMSX that it must stop the execution of the program, and wait for a manual restart.



We are not going to detail here how the debugger works, because that goes beyond our subject. Let just check theses 3 points :

- 1 - The stop command in the source code
- 2- The indication in openMSX that the program is paused
- 3- openMSX Debugger. To resume program execution, click on RUN

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Use command line arguments with your program

It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program especially when you want to control your program from outside instead of hard-coding those values inside the code.

The command line arguments are handled using main() function arguments where **argc** refers to the number of arguments given, and **argv[]** is a pointer array which points to each argument given to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly.

```
#include <stdio.h>
int main( char *argv[], int argc )
{
    if( argc == 1 )
    {
        printf("The argument supplied is %s\n", argv[0]);
    } else if( argc > 1 )
        printf("Too many arguments supplied.\n");
    } else {
        printf("One argument expected.\n");
    }
}
```

When the above code is compiled and executed with a single argument, it produces the following result.

```
> a.com testing
The argument supplied is testing
```

When the above code is compiled and executed with two arguments, it produces the following result.

```
> a.com testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
> a.com
One argument expected
```

It should be noted that **argv[0]** is a pointer to the first command line argument supplied, and ***argv[n]** is the last argument. If no arguments are supplied, **argc** will be 0.

Important notice about compilation of programs that need arguments.

To be able to use arguments with your MSX-DOS program you must compile your program with an extended version of the crt0 startup code. You have two possibilities to do that...

1 - Use the Overriding compilation directives

Add these 2 line at the top of your C listing.

```
#define __SDK_CRT0__ rt0_msxdos_advanced.rel
#define __SDK_ADDRCODE__ 0x180
```

or

2 - Modify the compilation script, “build.bat” or “build.sh”.

look for the lines:

```
# -- Default CRT0 ton use
DEFAULT_CRT0="${INCLUDE_DIR}crt0_msxdos.rel"
Change the text value “crt0_msxdos.rel” by “crt0_msxdos_advanced.rel”
```

```
# -- Default Code Address (sdcc parameter)
DEFAULT_ADDR_CODE="0x106"
```

Change the text value “0x106” by “0x180”

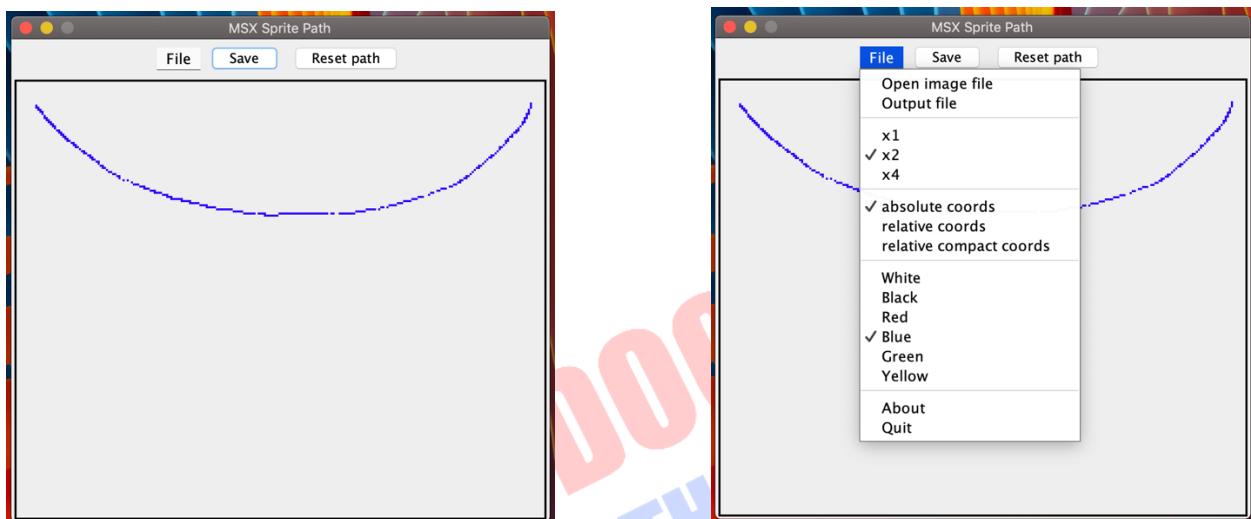
PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Fusion-C - Tools

Fusion-c comes with several tools that can help you create games. Two of them were specially developed by Sylvain Cregut for Fusion-C. These are **Sprite Path Editor**, and **Image To Sprite Editor**. You will find these 2 tools in .JAR (Java) format in the Tools folder. To be able to use these tools, you must first install Java Runtime on your computer.

Sprite Path

Sprite Path is a tool that lets you draw paths, or routes and retrieve the coordinates of each point drawn, in the order in which they were drawn.



Draw your Path with the left mouse button. The right mouse button let you go back. **Open Image file**: Let you load a 256x212 image in the background of the windows.

OutPut File: Let you choose the folder where you want to save the output file.

X1, X2, X4: let you zoom in the Window

Absolute cords: Is the default choice. It will save the coordinates of each point like they are on the MSX's screen, for example (10,250) or (185,200)

Sample output file: (The last number is the total number of coordinates)

```
{99, 26, 99, 27, 100, 27, 101, 27, 102, 27, 103, 27, 104, 27, 105, 27, 106, 28, 107, 28}
10
```

Relative Coords : Will save the coordinates of each point, according to the coordinates of the previous point. For example, if the first point is (100,100), the second point can be (-1,-2), next point (1,4 etc...). Sample output file:

(The first 2 numbers are the absolute coordinates of the first point. The last number is the total number of coordinates)

```
{99, 26}
{0, 1, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 55, 30}
11
```

Relative compact Coord : It is the same as above, except that the coordinates X and Y are coded in a single byte instead of two. The first 4 bits of the byte correspond to the relative position on the X-axis, the other 4 bits correspond to the relative coordinate on the Y-axis.

Please note that this means that the points are never spaced more than 7 pixels apart.

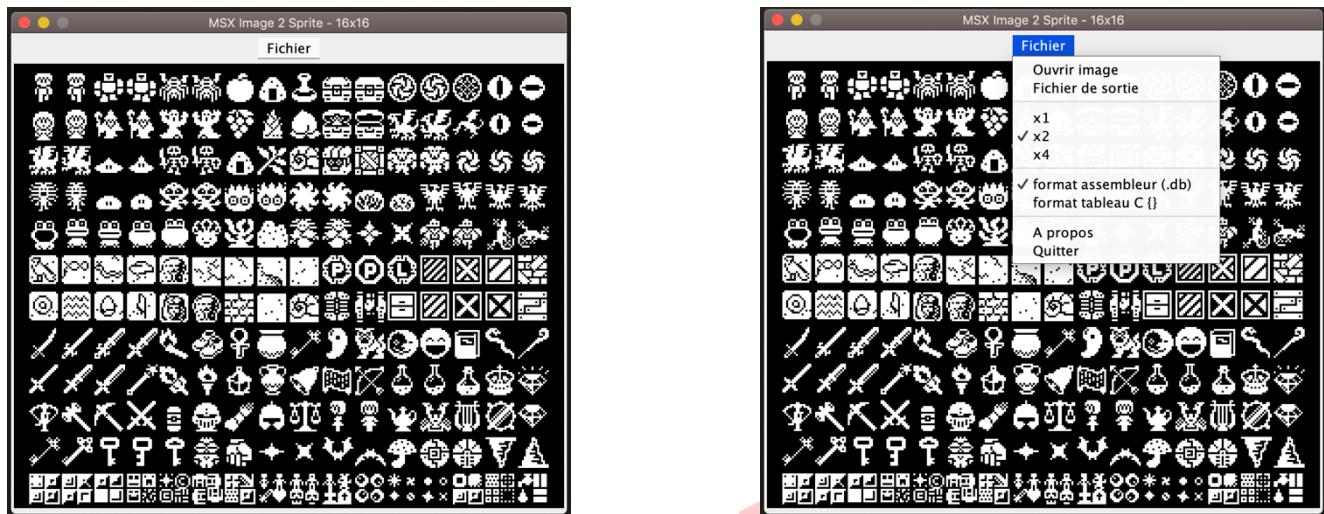
To quickly decode these coordinates you can use the **SpriteFollow** function of Fusion-C.

Sample output file:

```
{99, 26}
{0x20, 0x02, 0x02, 0x02, 0x02, 0x02, 0x22, 0x02, 0x00}
11
```

Image To Sprite Editor

Image To sprite Editor allows you to simplify the creation of 16x16 sprites from already existing single-image images.



Inside the window the mouse pointer is transformed into a red square, it's the top left angle represents the mouse click pointer.

The left first click say to the program what is the color 0.

The second left click, will take a snapshot of the drawing and convert it into a 16x16 sprite data.

Open Image file: Let you load a 256x212 .png or .gif image in the background of the windows.

OutPut File: Let you choose the folder where you want to save the output file.

X1, X2, X4: let you zoom in the Window

C format or ASM format : let you choose the output data format. For use with C code, or assembler code.

Sample ooutput file:

```
{0xFF, 0xD5, 0xAA, 0x6B, 0x7F, 0x1C, 0x43, 0xBD,
 0xBD, 0xBD, 0x32, 0x36, 0x36, 0x30, 0x00, 0x00,
 0x80, 0x80, 0x80, 0x00, 0x00, 0x00, 0x00, 0x80,
 0x80, 0x80, 0x00, 0x00, 0x00, 0x00, 0x00}
or
.db 0x7F, 0xFF, 0xD5, 0xAA, 0x6B, 0x7F, 0x1C, 0x43,
 0xBD, 0xBD, 0xBD, 0x32, 0x36, 0x36, 0x30, 0x00,
 0x00, 0x80, 0x80, 0x80, 0x00, 0x00, 0x00, 0x00,
 0x80, 0x80, 0x80, 0x00, 0x00, 0x00, 0x00}
```

Technical information about MSX & MSX2

Here we will describe essential technical information about MSX, you may not need them to code your game in C with FUSION-C. But, in some cases it may be useful. Also it's a way to understand how MSX computers are working ... Especially about the graphics capabilities

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without permission
is copyright violation



PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

MSX Models summary

The table below shows the minimal equipment of MSX computers models (an actual MSX may have more memory, or a different VDP, other sound IC and so on).

		MSX	MSX2	MSX2+	Turbo-R
CPU		Z80 (3.579545MHz±1%)			Z80 (3.579545MHz) + R800 (28.636Mhz)
Memory (minimal)	ROM	32KB (32KB Main ROM + MSX-BASIC v.1.0)	48KB (32KB Main ROM + MSX-BASIC v.2.0 + 16KB SubROM)	80KB (32KB Main ROM + MSX- BASIC v.3.0 + 16KB SubROM + 16KB Kanji Driver + 16KB Kanji Converter)	? (32KB Main ROM + MSX- BASIC v.4.0 + 16KB SubROM + 16KB Kanji Driver + 16KB Kanji Converter + ?)
	RAM	8KB	64KB	64KB	256KB or 512KB
Video	VDP	TMS9918A	V9938	V9958	
	VDP Frequency	?	?	?	21.477MHz
	VRAM	16KB	128KB	128KB	
	Maximal Resolution	256 x 192 x 16c	512 x 212 x 16c 256 x 212 x 256c (424v interlaced) 80 columns Mode	512 x 212 x 16c 256 x 212 x 19268c (424v interlaced) 80 columns Mode	
Features	Features	4 sprites in line. Max 32 Sprites simultaneous.	8 sprites in line; Max 32 Sprites simultaneous. Vertical scroll	8 sprites in line; Max 32 Sprites simultaneous on screen. Vertical & Horizontal scroll	
	PSG	AY-3-8910			
	FM	-	FM-PAC (extra)	MSX-Music (most models)	MSX-Music
Audio	PCM	-			8 bits
	CMT (Data-corder)	1200/2400 bps			-
Keyboard		88 key matrix			
Floppy Disk Drive		1DD (180KB), 2DD (360KB) or 2HD (720KB)			3.5" 2HD (720KB)
Printer		(optional)	8 bits parallel Centronics		
Cartridge Slot		1 (minimal)	2	2	2
Joystick		1 (minimal)	2		
Kanji Function	Kanji ROM	(optional)	(optional)	Level 1 (Level 2 optional)	Level 1 + 2
	Kanji Input	Application dependent		MSX-JE	MSX-JE
Real-time Clock		-	RPC5C01		

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

The MSX Keyboard

The keyboard of MSX computers is perfectly recognizable. It always contains **5 function keys** on the top of the keyboard, as well as **4 arrow keys** allowing moving the cursor, placed on the right side of the keyboard. The **GRAPH, CODE, CAPS, INS, DEL, SELECT, HOME, ESC, CTRL, SHIFT, and STOP** keys are also part of the MSX standard.

According to the computer models, or the countries, the keyboard matrices can be different, but the specific keys being part of the standard have an identical access to the software level.



Reading the keyboard keys can be done with the `GetKeyMatrix` function. Here are several keyboard matrices, which will allow you to adapt your programs to different MSX computers.

International key matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
line 0	7 &	6 ^	5 %	4 \$	3 #	2 @	1 !	0)
line 1	; :] }	[{	\ _	= +	- _	9 (8 *
line 2	B	A	DEAD	/ ?	. >	, <	` ~	' "
line 3	J	I	H	G	F	E	D	C
line 4	R	Q	P	O	N	M	L	K
line 5	Z	Y	X	W	V	U	T	S
line 6	F3	F2	F1	CODE	CAPS	GRAPH	CTRL	SHIFT
line 7	RET	SELECT	BS	STOP	TAB	ESC	F5	F4
line 8	→	↓	↑	←	DEL	INS	HOME	SPACE
line 9	NUM4	NUM3	NUM2	NUM1	NUM0	NUM/	NUM+	NUM*
line 10	NUM.	NUM,	NUM-	NUM9	NUM8	NUM7	NUM6	NUM5

Note: DEAD is the dead key with the accents ` , ^ and ``. If you press it nothing will happen, but if you press a vowel next, it will put the selected accent above it (example: ááâä). You can pick one of the accents by pressing the dead-key alone or in combination with SHIFT, CODE and CODE+SHIFT.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Japanese key matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
line 0	7 '	6 &	5 %	4 \$	3 #	2 "	1 !	0
line 1	; +	[{	@ `	¥	^ ~	- =	9)	8 (
line 2	B	A	-	/ ?	. >	, <] }	: *
line 3	J	I	H	G	F	E	D	C
line 4	R	Q	P	O	N	M	L	K
line 5	Z	Y	X	W	V	U	T	S
line 6	F3	F2	F1	かな ¹	CAPS	GRAPH	CTRL	SHIFT
line 7	RET	SELECT	BS	STOP	TAB	ESC	F5	F4
line 8	→	↓	↑	←	DEL	INS	HOME	SPACE
line 9	NUM4	NUM3	NUM2	NUM1	NUM0	NUM/	NUM+	NUM*
line 10	NUM.	NUM,	NUM-	NUM9	NUM8	NUM7	NUM6	NUM5
line 11 ²					No		Yes	

Notes:

¹ かな is Japanese writing for KANA. Unlike CODE, it is a toggle.² Used by Panasonic turboR, FS-A1WX and FS-A1WSX.***UK key matrix***

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
line 2	B	A	£	/	.	,	'	'

All other rows and the CODE- and GRAPH- matrices are equal to the International layout.

Spanish key matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
line 1	ñ Ñ] }	[{	\ \	= +	- _	9 (8 *
line 2	B	A	DEAD	/ ?	. >	, <	; :	''

All other rows are equal to the International layout.

Russian key matrix

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0

C Library for MSX-DOS with SDCC compiler

line 0	& 6	% 5	¤ 4	# 3	" 2	! 1	+ ;) 9
line 1	В Ж	* :	Н Х	- ^ Ъ	= _	\$ 0	(8	' 7
line 2	И И	Ф Ф	? /	<,	@ ИО	В Б	>.	\ Э
line 3	О О	[{ Ш	Р Р	Р П	А А	У У	W B	S C
line 4	К К	Ј Ј	З З] } Щ	Т Т	Х Ъ	Д Д	Л Л
line 5	Q Я	Н Н	~ Ч	С Ц	М М	Г Г	Е Е	Y Ы
line 6	F3	F2	F1	PYC	CAPS	GRAPH	CTRL	SHIFT
line 7	RET	SELECT	BS	STOP	TAB	ESC	F5	F4
line 8	→	↓	↑	←	DEL	INS	HOME	SPACE
line 9	NUM4	NUM3	NUM2	NUM1	NUM0	NUM/	NUM+	NUM*
line 10	NUM.	NUM,	NUM-	NUM9	NUM8	NUM7	NUM6	NUM5

Note: PYC works like the Japanese カタ (KANA) key, it is a toggle.

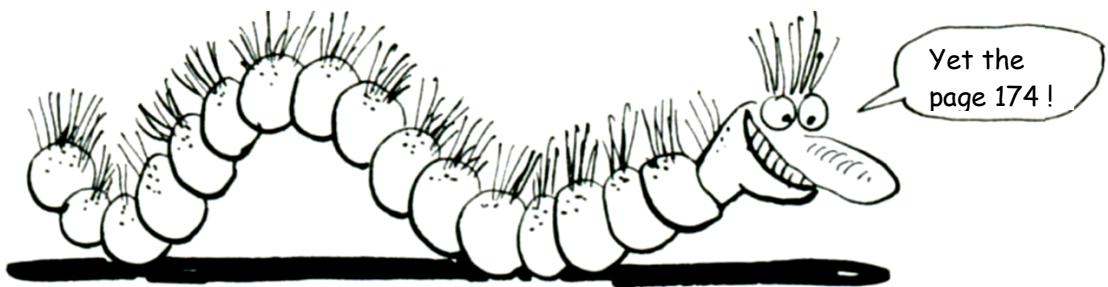
French key matrix

	bit 7	bit 6	bit 5	bit 4	'bit 3	bit 2	bit 1	bit 0
line 0	7 è	6 §	5 (4 ‘	3 “	2 é	1 &	0 à
line 1	M	\$ *	^”§	<>	- _) °	9 ç	8 !
line 2	B	Q	DEAD	= +	: /	; .	# £	ù %
line 3	J	I	H	G	F	E	D	C
line 4	R	A	P	O	N	, ?	L	K
line 5	W	X	Y	Z	V	U	T	S
line 6	F3	F2	F1	CODE	CAPS	GRAPH	CTRL	SHIFT
line 7	RET	SELECT	BS	STOP	TAB	ESC	F5	F4
line 8	→	↓	↑	←	SUP	INS	HOME	SPACE
line 9	NUM4	NUM3	NUM2	NUM1	NUM0	NUM/	NUM+	NUM*
line 10	NUM.	NUM,	NUM-	NUM9	NUM8	NUM7	NUM6	NUM5

Note about above arrays :If there are 2 characters in a row, the first is the basic character, and the second is with SHIFT pressed.

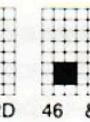
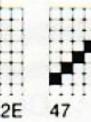
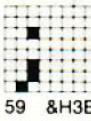
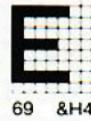
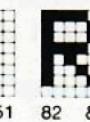
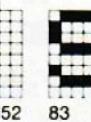
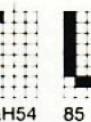
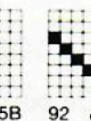
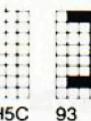
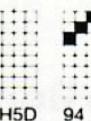
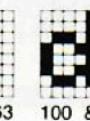
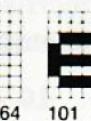
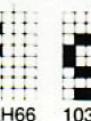
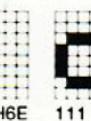
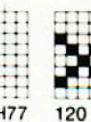
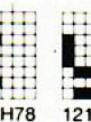
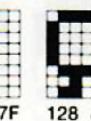
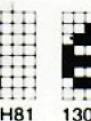
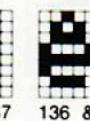
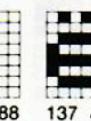
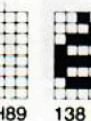
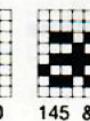
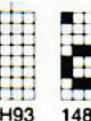
PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

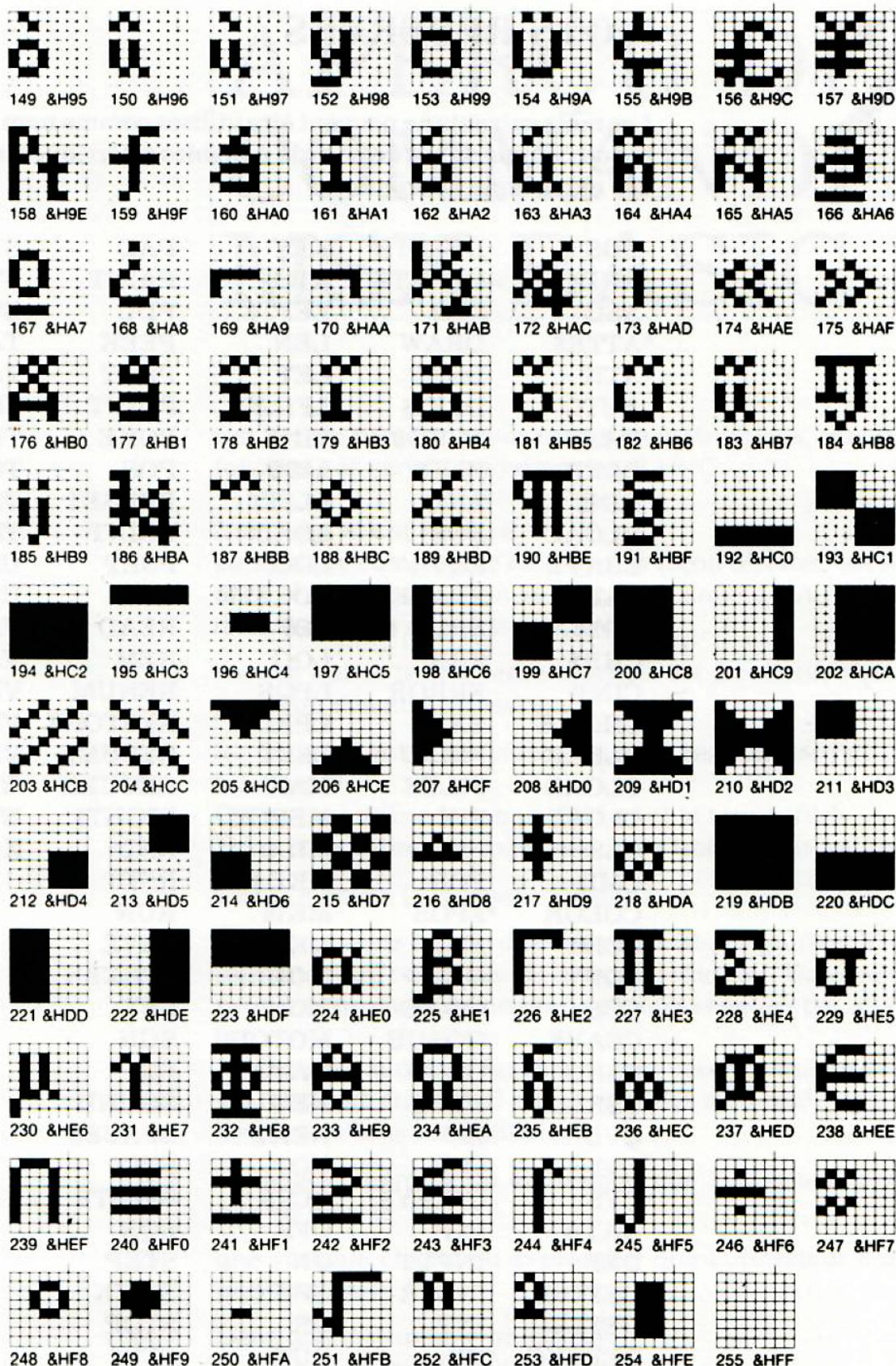
PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation



The ASCII table

Note : The ascii codes before #32 are not printable

 32 &H20	 33 &H21	 34 &H22	 35 &H23	 36 &H24	 37 &H25	 38 &H26	 39 &H27	 40 &H28
 41 &H29	 42 &H2A	 43 &H2B	 44 &H2C	 45 &H2D	 46 &H2E	 47 &H2F	 48 &H30	 49 &H31
 50 &H32	 51 &H33	 52 &H34	 53 &H35	 54 &H36	 55 &H37	 56 &H38	 57 &H39	 58 &H3A
 59 &H3B	 60 &H3C	 61 &H3D	 62 &H3E	 63 &H3F	 64 &H40	 65 &H41	 66 &H42	 67 &H43
 68 &H44	 69 &H45	 70 &H46	 71 &H47	 72 &H48	 73 &H49	 74 &H4A	 75 &H4B	 76 &H4C
 77 &H4D	 78 &H4E	 79 &H4F	 80 &H50	 81 &H51	 82 &H52	 83 &H53	 84 &H54	 85 &H55
 86 &H56	 87 &H57	 88 &H58	 89 &H59	 90 &H5A	 91 &H5B	 92 &H5C	 93 &H5D	 94 &H5E
 95 &H5F	 96 &H60	 97 &H61	 98 &H62	 99 &H63	 100 &H64	 101 &H65	 102 &H66	 103 &H67
 104 &H68	 105 &H69	 106 &H6A	 107 &H6B	 108 &H6C	 109 &H6D	 110 &H6E	 111 &H6F	 112 &H70
 113 &H71	 114 &H72	 115 &H73	 116 &H74	 117 &H75	 118 &H76	 119 &H77	 120 &H78	 121 &H79
 122 &H7A	 123 &H7B	 124 &H7C	 125 &H7D	 126 &H7E	 127 &H7F	 128 &H80	 129 &H81	 130 &H82
 131 &H83	 132 &H84	 133 &H85	 134 &H86	 135 &H87	 136 &H88	 137 &H89	 138 &H8A	 139 &H8B
 140 &H8C	 141 &H8D	 142 &H8E	 143 &H8F	 144 &H90	 145 &H91	 146 &H92	 147 &H93	 148 &H94



MSX 1 video screen modes

SCREEN 0	40 x 24 text mode	
	Memory address	Size
Char table	0000 – 03BF	960 Bytes
Char Tiles	0800 – 0FFF	2048 Bytes
Size of page	4096	Bytes
No sprites available		
Number of Colors	2 out of 15	

On screen 0, the tiles are defined as usual (8x8 pixels), but only the leftmost 6x8 pixels of each tile are visible.

SCREEN 1	32 x 24 Colored text mode	
	Memory address	Size
Char Tiles	0000 – 07FF	2048 Bytes
Char Map	1800 – 1AFF	768 Bytes
Sprite Attributs	1B00 – 1B7F	128 Bytes
Color Table	2000 – 201F	32 Bytes
Sprite pattern	3800 – 3FFF	2048 Bytes
Size of page	4096	Bytes
Number of Colors	16 out of 16 (out of 512 on MSX2)	

The BG Colors array defines 32 colors (each 4 bits background, and 4 bit foreground, as in VDP register 7). The colors are assigned to the BG Tiles as follows: Tiles 00..07 shares the first color tiles 08..0F shares the second color, etc, and tiles F8..FF share the last color.

SCREEN 2	256 x 192 Graphics mode 16 colors	
	Memory address	Size
Char Tiles	0000 – 17FF	6144 Bytes
Char table	1800 – 1AFF	768 Bytes
Sprites Attributes	1B00 – 1B7F	128 Bytes
Colors table	2000 – 37FF	6144 Bytes
Sprites pattern	3800 – 3FFF	2048 Bytes
Size of page	16384	Bytes
Number of Colors	16 (2 colors per line)	

The char Tiles array can be placed only at address 0000h or 2000h, ie. only bit 2 of VDP Register 4 is used. The Colors array is placed at the same address XORed by 2000h or 0000h).

In screen 2, the Char Tile memory consists of 768 tiles. The screen is vertically divided into 3 sections, all BG Map entries for the upper 8 character rows refer to tiles 00..FFh, the middle 8 rows to tiles 100..1FFh, and the bottom 8 rows to tiles 200..2FFh.

As usual, all TILE-BYTES define rows of eight pixels, each of these rows is colorized by a separate COLOR-BYTE in the BG Colors array, whereas each COLOR-BYTE defines the background color in bit 0-3 (for "0" bits in TILE-BYTE) and the foreground color in bit 4-7 (for "1" bits). That means there can be only 2 different colors in each row of 8 pixels!

SCREEN 3	64 x 48 Block graphics 16 Colors		
	Memory address	Size	
Char Tiles (Block colors)	0000 – 07FF	2048	Bytes
Char Map	0800 – 0AFF	768	Bytes
Sprite Attribute table	1B00 – 1B7F	128	Bytes
Color table	2020 – 203F	32	Bytes
Sprite patterns	3800 – 3FFF	2048	Bytes
Size of page		4096	Bytes
Number of Colors	16		

The screen consists of 32x24 background tiles, each tile consists of 4 blocks, whereas each of these "pixels" can be colorized in any of the available 16 colors.

The Char Tile memory is organized as follows: It contains 8 color bytes for each of the 100h tiles. But obviously only two bytes are actually required (first byte for the upper half, and second byte for the lower half, in both cases most significant bits for the left 'pixels').

Which of the 8 bytes are used depends on the lower two bits of the vertical position (0-23), ie. tiles in lines 0,4,8,12,etc. use the first two bytes, tiles in lines 1,5,9,etc. use the next two bytes, and so on.

PRIVATE DOCUMENT!
DO NOT SHARE THIS DOCUMENT
 Sharing This Document without authorization
 is copyright violation

MSX 2 video screen modes

SCREEN 0	MSX2	80 x 24 text mode
		Memory address Size
Char table		0000 – 0F1F 960 Bytes
Char Tiles		1000 – 17FF 2048 Bytes
Size of page		8192 Bytes
No sprites available		
Number of Colors		2 out of 512

The Characters attribute table is an array of bits, ie. the first byte of the table contains bits for the first 8 characters on the screen.

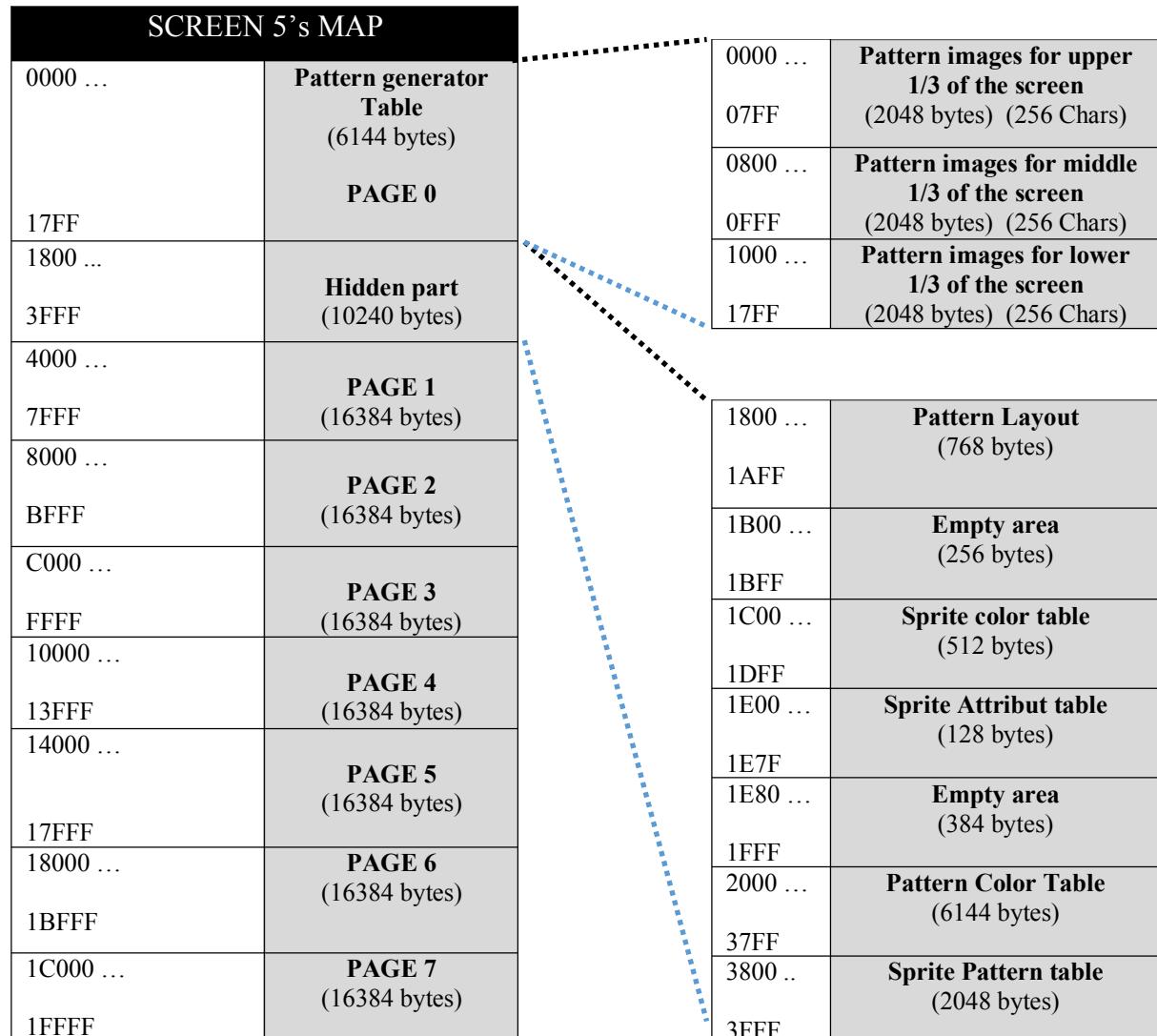
If the attribute-bit is zero, then the character is displayed as usual (colored as defined in VDP Register 07h). If the bit is set, then the character is blinking (see VDP Register 0Ch, and VDP Reg 0Dh).

DU " Sharing This " is copy -



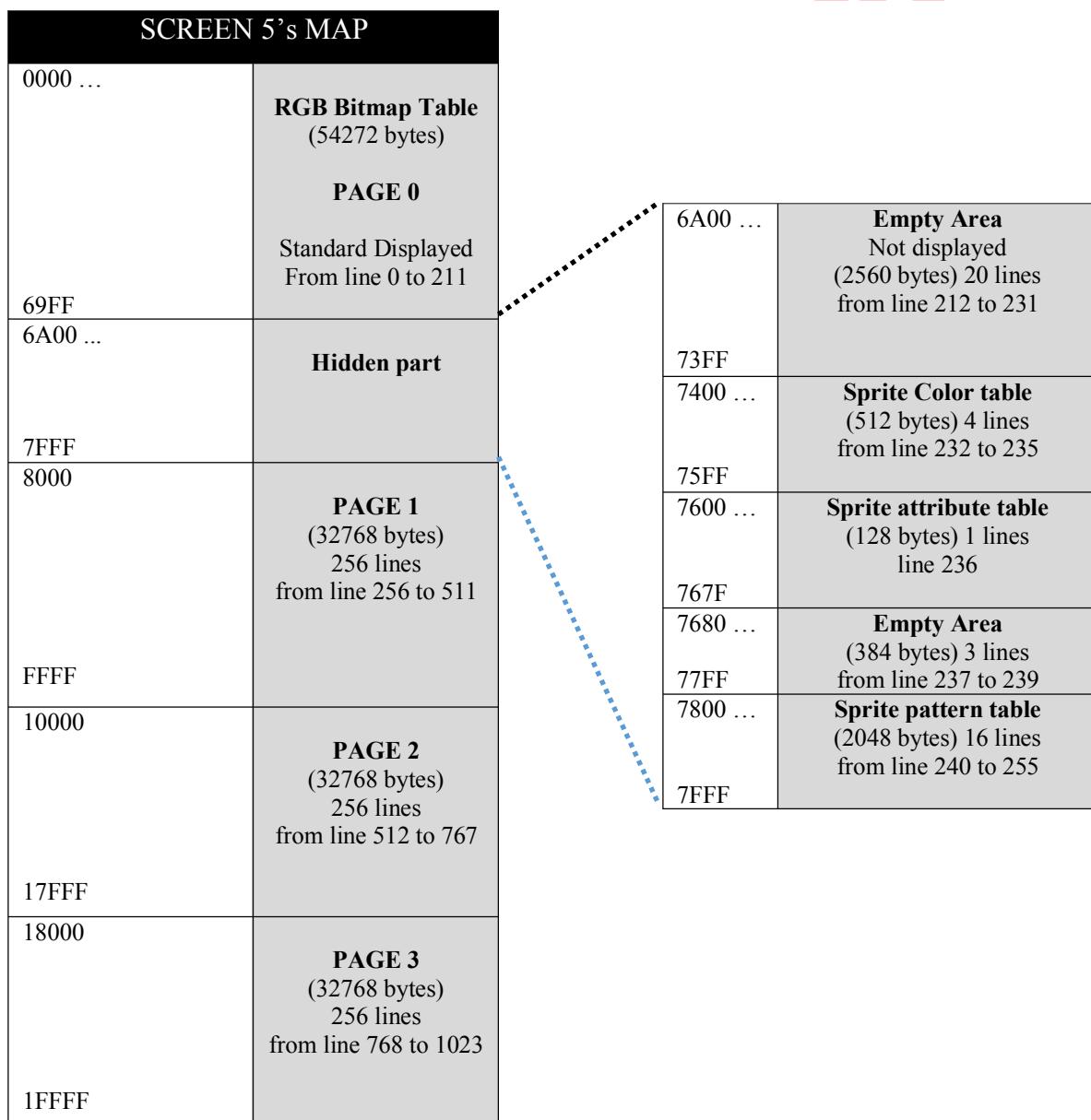
SCREEN 4	MSX2	256 x 192 Graphic pattern mode 16 colors		
		Memory address	Size	
Pattern generator table		0000 – 17FF	6144	Bytes
Pattern layout		1800 – 1AFF	768	Bytes
<i>Empty area</i>		1B00 – 1BFF	256	Bytes
Sprite colors table		1C00 – 1DFF	512	Bytes
Sprite attribute table		1E00 – 1E7F	128	Bytes
<i>Empty area</i>		1E80 – 1FFF	384	Bytes
Pattern color table		2000 – 37FF	6144	Bytes
Sprite pattern		3800 – 3FFF	2048	Bytes
Size of page			16384	Bytes
Number of Colors		16 out of 512		
Number of pages		8		

This is mostly the same as the MSX1's Screen 2 mode, except that the foreground sprites can have additional color attributes (MSX2 Sprite mode 2), and with the ability to display a maximum of 8 sprites per line. Also the color palette can be defined by the user.



SCREEN 5	MSX2	256 x 212 Graphic bit-mapped mode 16 Colors		
Bitmap Table		Memory address	Size	
		0000 – 69FF	27136 Bytes	(212 lines)
<i>Empty area</i>		6A00 – 73FF	2560 Bytes	
Sprite colors table		7400 – 75FF	512 Bytes	(H. line n°232)
Sprite attribute table		7600 – 767F	128 Bytes	
<i>Empty area</i>		7680 – 77FF	384 Bytes	
Sprite pattern		7800 – 7FFF	2048 Bytes	
Size of page			32768 Bytes	
Number of Colors		16 out of 512		
Number of pages		4		

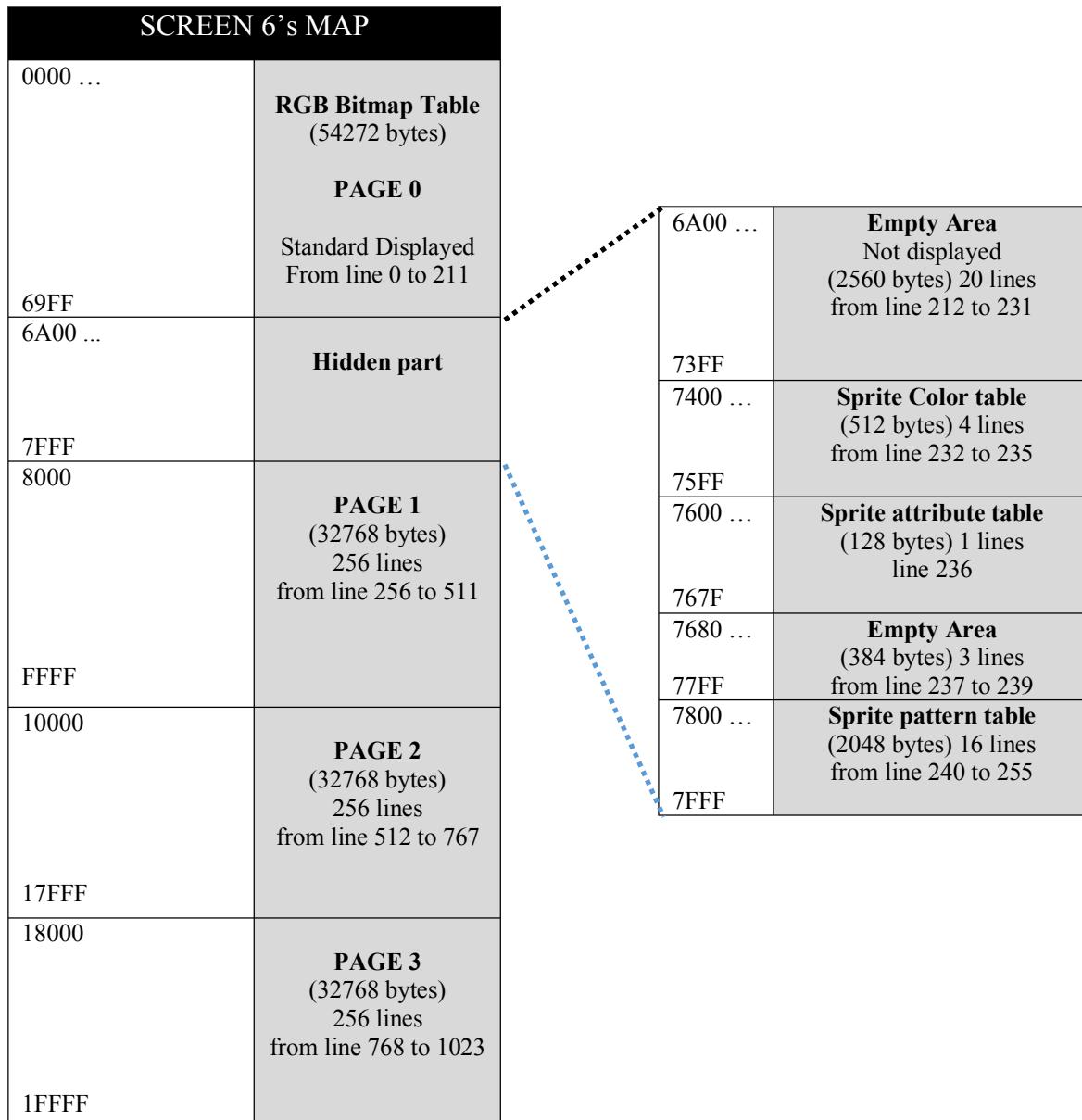
This is a bitmap screen mode. Each byte of the bitmap table refers to 2 pixels. First 4 bits represents the color of the first pixel (peer), the last 4 bits represents the second pixel color (odd).
One line of 256 pixels represents 128 Bytes.



SCREEN 6	MSX2	512 x 212 Graphic bit-mapped mode 4 Colors		
Bitmap Table		Memory address	Size	
		0000 – 69FF	27136 Bytes	(212 lines)
<i>Empty area</i>		6A00 – 73FF	2560 Bytes	
Sprite colors table		7400 – 75FF	512 Bytes	(H. line n°232)
Sprite attribute table		7600 – 767F	128 Bytes	
<i>Empty area</i>		7680 – 77FF	384 Bytes	
Sprite pattern		7800 – 7FFF	2048 Bytes	
Size of page			32768 Bytes	
Number of Colors		4 out of 512		
Number of pages		4		

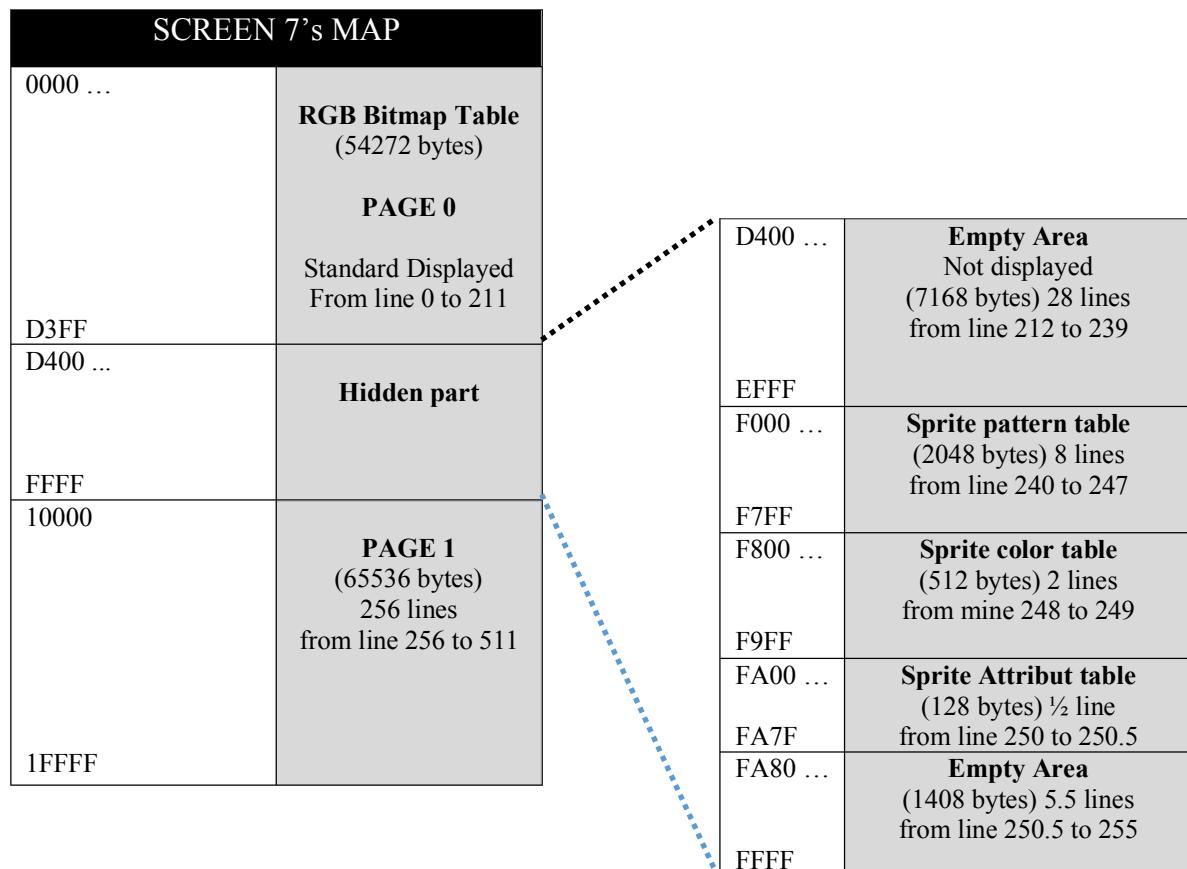
This is a bitmap screen mode. Each byte of the bitmap table refers to 4 pixels, bits 6 & 7 code the color of first pixel, bits 4 & 5 code the color of the 2nd pixel, bits 2 & 3 code the color of the 3rd pixel, and bits 1 & 0 code the color of the 4th pixel.

One line of 256 pixels represents 128 Bytes.



SCREEN 7	MSX2	512 x 212 Graphic bit-mapped mode 16 Colors		
Bitmap Table		Memory address	Size	
		0000 – D3FF	54272 Bytes	(212 lines)
<i>Empty area</i>		D400 – EFFF	7168 Bytes	
Sprite pattern		F000 – F7FF	2048 Bytes	(H. line n°240)
Sprite colors table		F800 – F9FF	512 Bytes	
Sprite attribute table		FA00 – FA7F	128 Bytes	
<i>Empty area</i>		FA80 – FFFF	1408 Bytes	
Size of page			65536 Bytes	
Number of Colors		16 out of 512		
Number of pages		2		

In this bitmap screen mode. Each byte of the bitmap table refers to 2 pixels. First 4 bits represent the color of the first pixel (peer), the last 4 bits represents the second pixel color (odd).
One line of 512 pixels represents 256 Bytes.



SCREEN 8	MSX2	256 x 212 Graphic bit-mapped mode 256 Colors		
RGB Bitmap Table		Memory address	Size	
		0000 – D3FF	54272 Bytes	(212 lines)
<i>Empty area</i>		D400 – EFFF	7168 Bytes	
Sprite pattern		F000 – F7FF	2048 Bytes	(H. line n°240)
Sprite colors table		F800 – F9FF	512 Bytes	
Sprite attribute table		FA00 – FA7F	128 Bytes	
<i>Empty area</i>		FA80 – FFFF	1408 Bytes	
Size of page			65536 Bytes	
Number of Colors		256		
Number of pages		2		

Each byte in the RGB Matrix defines a separate pixel. The bytes directly define the colors as follows:

Bits 0 & 1 = Blue, 0-3

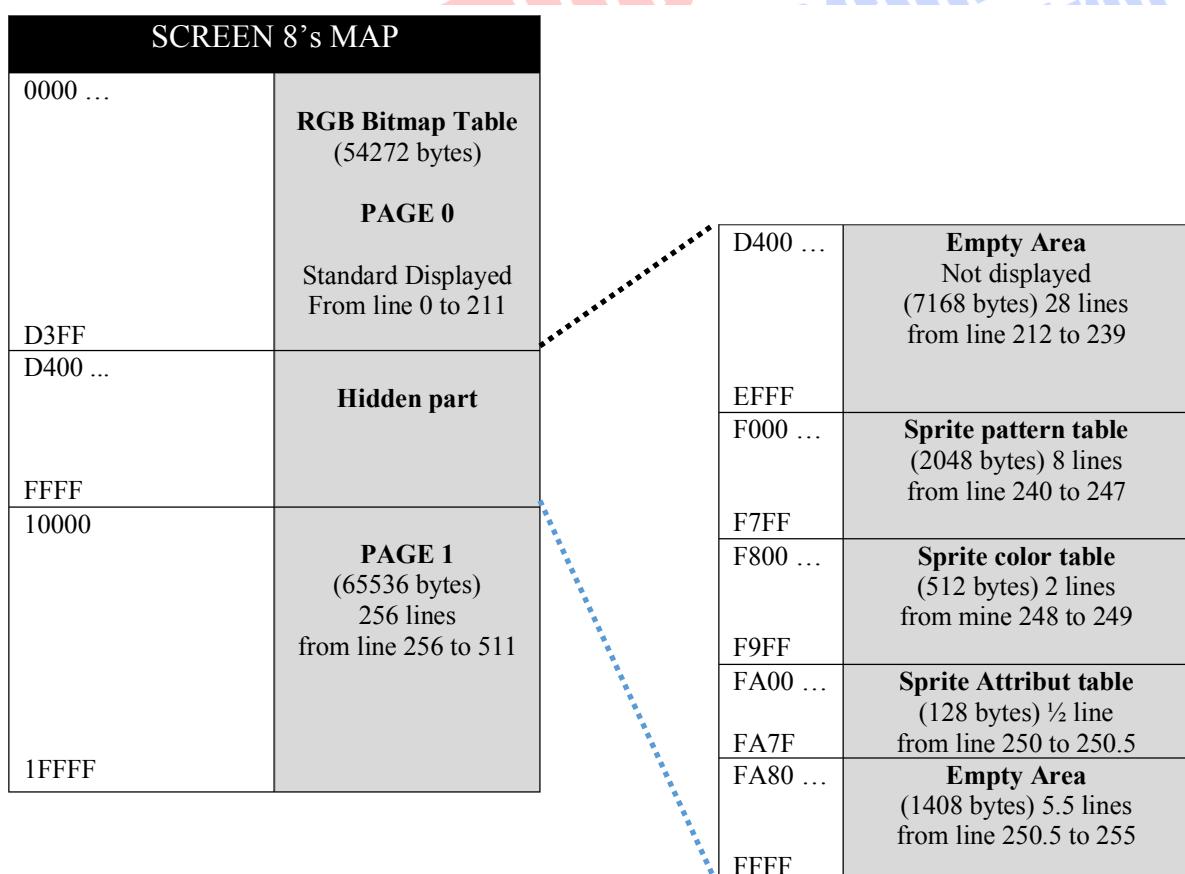
Bits 2, 3 & 4 = Red, 0-7

Bits 5, 6 & 7 = Green, 0-7

Coding the color can be done with this formula : Color = 32 * G + 4 * R + B

One line of 256 pixels represents 256 Bytes.

Note : in screen mode 8, the sprite color palette is limited to a fixed 16 color palette.



SCREEN 10 & 11 MSX² **256 x 212 Graphic bit-mapped mode 12499 Colors**

	Memory address	Size	
RGB Bitmap Table	0000 – D3FF	54272 Bytes	(212 lines)
<i>Empty area</i>	D400 – EFFF	7168 Bytes	
Sprite pattern	F000 – F7FF	2048 Bytes	(H. line n°240)
Sprite colors table	F800 – F9FF	512 Bytes	
Sprite attribute table	FA00 – FA7F	128 Bytes	
<i>Empty area</i>	FA80 – FFFF	1408 Bytes	
Size of page		65536 Bytes	
Number of Colors	12499		
Number of pages	2		

Screen modes 10 & 11 are only available on MSX2+ and MSX Turbo-R.
They share the same VRAM map as the MSX2's screen mode 8.

Screen modes 10 and 11 are the same. The difference comes from the way the MSX-BASIC handle each mode. When programming directly with C, or assembler the two modes are the same.
In this mode the colors are coded with YAE coding. (This is not the purpose of this book to describe how it works). This means The VDP can show up to 12499 colors simultaneously on screen.
But be careful about the pixels manipulation. Pixels are coded by block of 4 following pixels. This means pixels 0,1,2, and 3 should always be together, idem to pixels 4,5,6 and 7 etc ...
Note : The sprites keep the same RGB palette as the screen mode 8.

SCREEN 12 MSX² **256 x 212 Graphic bit-mapped mode 19268 Colors**

	Memory address	Size	
RGB Bitmap Table	0000 – D3FF	54272 Bytes	(212 lines)
<i>Empty area</i>	D400 – EFFF	7168 Bytes	
Sprite pattern	F000 – F7FF	2048 Bytes	(H. line n°240)
Sprite colors table	F800 – F9FF	512 Bytes	
Sprite attribute table	FA00 – FA7F	128 Bytes	
<i>Empty area</i>	FA80 – FFFF	1408 Bytes	
Size of page		65536 Bytes	
Number of Colors	19268		
Number of pages	2		

This screen mode is only available on MSX2+ and MSX Turbo-R.
It share the same VRAM map as the MSX2's screen mode 8.

With Screen mode 12 the colors are coded with YJK coding. (This is not the purpose of this book to describe how it works). This means The VDP can show up to 19268 colors simultaneously on the screen.
But be careful about the pixels manipulation. Pixels are coded by block of 4 following pixels. This means pixels 0,1,2, and 3 should always be together, idem to pixels 4,5,6 and 7 etc.
Note : The sprites keep the same RGB palette as the screen mode 8.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

The Sprites

Sprites are graphic objects generated and managed by the MSX Video Processor (Hardware). The sprites can be moved over the main image, as layers over it.

The MSX Video Processor can memorize up to **256 sprite's patterns**. It can show a maximum of **32 sprites simultaneously on screen**. With the **MSX1**, **only 4 sprites are visible on the same horizontal line**, if there are more than 4 sprites, only 4 sprites with high priority are shown (see below).

With the **MSX2**, **8 sprites are visible on the same horizontal line**. A Standard sprite is an object of 8x8 pixels, or 16 x 16 pixels, their size can be normal, or double (multiplied by 2). There are 2 sprites' types, automatically chosen by the VDP.

Type 1 : For screen mode 1, 2 and 3

Maximum 32 sprites on screen
Only 4 sprites on the same horizontal line
Standard size or Double Size
Sprites are monochrome (Only one color at a time)

Type 2 : For screen mode 4, 5, 6, 7, 8 , 10, 12

Maximum 32 sprites on screen
Only 8 sprites on the same horizontal line
Standard size or Double Size
Sprites can be multicolor : One color per sprite line.

The patterns

A pattern is the graphic representation of the sprite, composed of 8 bytes. Each bit of each byte represents a pixel of the sprite. It can be a lit pixel, or a pixel off.

Here an example of a 8x8 pixels sprite pattern in its binary representation. Each line is a byte, and can be represented by a decimal value or an hexadecimal value, it does not matter.

The Vram memory can record up to 256 patterns like this one.

0	0	1	1	1	1	0	0
0	1	1	1	1	1	0	0
1	0	0	0	0	0	1	0
1	0	0	0	0	0	1	0
1	1	1	1	1	1	1	1
0	1	1	1	1	1	0	0
0	0	1	0	0	1	0	0
0	1	0	0	0	1	0	0

8x8 sprite

The MSX 's VDP can manipulate, 8x8 pixels sprite's patterns or 16x16 pixels sprite's patterns. In memory a 16x16 pixels sprite is composed of **four** 8 x 8 patterns, in this order:

0	2
1	3

16 x 16 sprite

The four patterns must be contiguous in memory. Thus the first 8 bytes of the sprite's pattern are defining the top left corner of the sprite, the 8 following bytes are defining the bottom left corner, etc ...

Example of an 8 x 8 sprite definition with binary values.

```
Char mySpritePattern1[] = {
    0b11111111,
    0b11111111,
    0b11000011,
    0b11011011,
    0b11011011,
    0b11000011,
    0b11111111,
    0b11111111
};
SetSpritePattern(0,
mySpritePattern1 )
```

C Library for MSX-DOS with SDCC compiler
Example of a 16 x 16 sprite definition with binary values.

```
unsigned int mySpritePattern2[] = {  
    0b1111111111111111,  
    0b1111111111111111,  
    0b1100000000000011,  
    0b1100000000000011,  
    0b1100000011000011,  
    0b1100001111000011,  
    0b1100011111100011,  
    0b1100011111100011,  
    0b1100011111100011,  
    0b1100011111100011,  
    0b1100011111100011,  
    0b1100011111100011,  
    0b1100000000000011,  
    0b1100000000000011,  
    0b1111111111111111,  
    0b1111111111111111  
};  
SetSpritePattern( 0,  
    Sprite32bytes(mySpritePattern2) )
```

Note : You must use the *Sprite32Bytes* function to convert this 32 Bytes pattern to four 8-bit patterns in the correct order. Using this method is not mandatory, you can also use 8 bit values in the correct order. This sprite's pattern will consume 4 pattern slots. The pattern 0, 1, 2 and 3 will be used. Thus the next pattern available will be the pattern number 4.

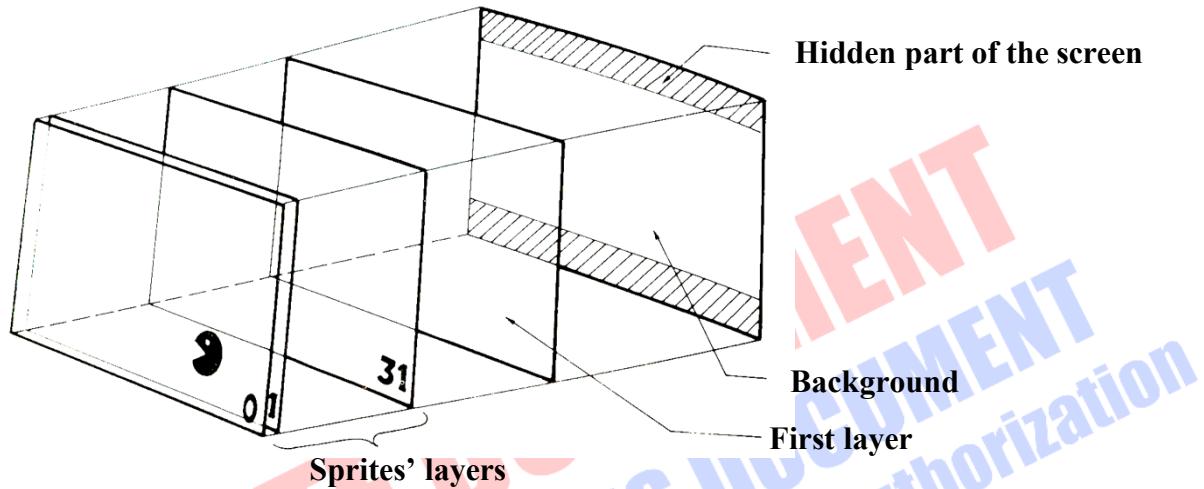


Sprites coordinate & priority

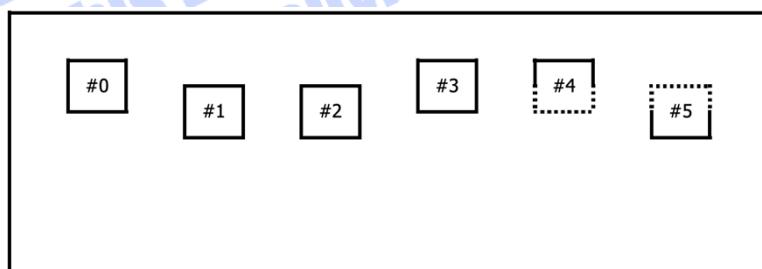
Sprites can be placed anywhere on the screen. X-axis and Y-axis coordinates of a sprite location is always between 0 and 255. This means that in graphics modes with 512 pixels dots on the X-axis, single sprite's pixel occupies two horizontal pixels of the screen.

As on MSX2 the screen show only 212 lines (or 192) over the 255 available, if the sprite Y coordinate is over 212 (or 192), it will disappear from the screen.

Note : if a sprite is placed at coordinate Y=208 all other sprites with a lower priority disappear. (Example, if sprite 12 is at Y=212 all sprites from 13 to 31 disappear.)



Up to 4 sprites can be displayed on a single horizontal line (8 sprites with MSX2's screen modes). The sprites with higher priority will be displayed and the overlapping portions of lower priority sprites will not be displayed. Lower sprite numbers are assigned higher priority, with #0 is of highest priority and #31 is of lowest priority.



With Fusion-C you can check and manage sprites overlapping with *SpriteOverlap* and *SpriteOverlapId* functions.

Sprites colors

With type 2's Sprites (MSX2), sprites can be multicolor. Each line of a sprite can have a different color. With Fusion-C when you are using a MSX2's screen mode the color parameter of the ***PutSprite*** function is disabled. You must define the sprite's colors with the ***SpriteColors*** function.

```
char myColors[ ]= {  
    6,6,10,10,10,10,8,8,8,8,8,10,12,12,12,12,1 };  
SpriteColors( 2, myColors ); // Assign myColors to  
                     sprie n°2
```

The colors array must always contain 16 values. Each byte corresponding to a line of a 16x16 sprite. In case you are using 8x8 Sprites, only the first 8 bytes will be used.

With screen modes 4 to 7, the colors used are the colors specified by color palette (see ***SetPalette***, or ***SetColorPalette*** functions). With screen mode 8 to 12, the sprite color palette is different and cannot be changed.

Here the Sprite color palette used in Screen mode 8 to 12:

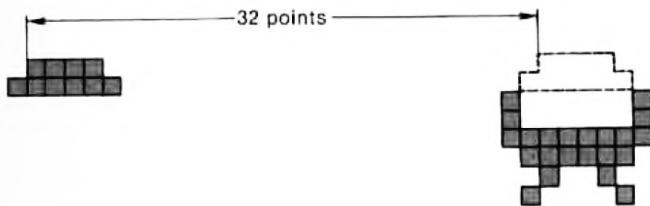
R	G	B	Name
00	0	0	Black
01	0	2	Dark Blue
02	3	0	Dark Red
03	3	2	Dark Purple
04	0	3	Dark Green
05	0	3	Turquoise Blue
06	3	3	Green Olive
07	3	3	Grey
08	4	4	Ligh Orange
09	0	0	Blue
10	7	0	Red
11	7	0	Purple
12	0	7	Green
13	0	7	Light Blue
14	7	7	Yellow
15	7	7	White

Extended attributes of the sprite color table

The MSX2's sprite color table is used as we explained, to define the color of each line of a sprite. But, there are more possibilities. Only the four least significant bits (bits 0 to 3) are used to specify the color number to use. The other bits, called EC, CC, and IC have other effects on sprites.

bit 7	Bit 6	Bit 5	Bit 4	Bit 0 to 3
EC	CC	IC	0	Color line 0
EC	CC	IC	0	Color line 1
EC	CC	IC	0	Color line 2
...
EC	CC	IC	0	Color line 8

When the EC bit is active (=1), the line is shifted 32 pixels to the left.



When the CC bit is activated, it activates the “OR” color mixing mode. This enables the possibility to multiply the color of sprites, when multiple sprites are overlapped.



This also disables the sprite priority mechanism for the specified lines.

You can find more information about color mixing here : https://www.msx.org/wiki/The_OR_Color

When the IC bit is activated (=1), it disables the collision mechanism on the affected line.

To use these possibilities you have to modify manually the values you are using in the color's array you want to apply to sprites. To do that you can precalculate the values to use, or add some logical operations on the value.

Suppose I want to disable the sprite collision mechanism for all the lines. I have to enable the bit 5 of each line by using this logical operation : (6 or 0b00100000), which is the same as (6 or 32) in a decimal representation.

In C language : `6 | 32`.

Finally, we have this color array :

```
char myColors[ ]= { 6 | 32 , 66 | 32 , 106 | 32 , 106 | 32 , 106 | 32 , 106 | 32 , 86 | 32 , 86 | 32 , 86 | 32 , 86 | 32 , 86 | 32 , 106 | 32 , 126 | 32 , 126 | 32 , 126 | 32 , 16 | 32 };  
SpriteColors( 2, myColors ); // Assign myColors to sprie n°2
```

The Attribut table

Inside the Vram, the sprite attributes table area defines display coordinates for all the 32 sprites layers, and the pattern number used for display. Each sprite has four bytes of attribute data.

For example, with the Screen 8 mode, the sprite attribute table starts at 0xFA00 and looks like this:

0xFA00	X coordinate	(0 .. 255)	}	Sprite N° 0
0xFA01	Y coordinate	(0 .. 255)		
0xFA02	Pattern number	(0 .. 255)		
0xFA03	Reserved for system			

0xFA04	X coordinate	(0 .. 255)	}	Sprite N° 1
0xFA05	Y coordinate	(0 .. 255)		
0xFA06	Pattern number	(0 .. 255)		
0xFA07	Reserved for system			

When you are using MSX1's Screen mode, the attribute table looks like this :

0x1B00	X coordinate	(0 .. 255)	}	Sprite N° 0 (Only EC bit is available + color number)
0x1B01	Y coordinate	(0 .. 255)		
0xAB02	Pattern number	(0 .. 255)		
0x1B03	EC 0 0 0 Color	(0 .. 15)		

0x1B04	X coordinate	(0 .. 255)	}	Sprite N° 1
0x1B05	Y coordinate	(0 .. 255)		
0x1B06	Pattern number	(0 .. 255)		
....				

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

The MSX Cartridges and rom mapper

Even if FUSION-C is oriented to build MSX-DOS programming. You may want to do some ROM, or transform your MSX-DOS disk image to a full ROM file and put it inside a cartridge.

Using the tool **DISK2ROM** you can convert your disk file image (.dsk) to a Rom file ready to be used with an emulator, or to be flashed in a cartridge. But be careful, when you transform a Disk image to Rom, the program inside the rom cannot access to the MSX drive, thus you will not be able to load or save external data to another device. Also, of course, as a rom file, your program cannot save any data.

There are multiple cartridge mappers on MSX, this means multiple way to manage memory in Rom. DISK2ROM can produce Roms with SCC mapper or ASCII8 mapper. Below, the description of the most common mappers available for MSX Cartridges.

Raw ROM without Mapper

Small cartridges with only 32Kbytes or less ROM aren't including memory mappers, this ROMs typically occupy the address space at 4000-BFFF.

Theoretically 64K ROMs aren't requiring a memory mapper as they could occupy the whole address space from 0000-FFFF

Memory	Content
0000 - 3FFFF	Sometimes, mirror of the 1st 16KB rom
4000 - 7FFFF	1st 16KB of Rom
8000 - BFFFF	2nd 16KB of Rom

Konami 8K without SCC

This type is used by Konami cartridges that do not have a SCC and some others.

Memory	Mapper I/O Address
4000 - 5FFFF	Fixed. Always bank 0
6000 - 7FFFF	Select bank by writing to 6000
8000 - 9FFFF	Select bank by writing to 8000
A000 - BFFFF	Select bank by writing to A000

Konami 8K with SCC

This type is used by Konami cartridges that do have a SCC chip.

Memory	Mapper I/O Address
4000 - 5FFFF	Select bank by writing to 5000
6000 - 7FFFF	Select bank by writing to 6000
8000 - 9FFFF	Select bank by writing to 8000
A000 - BFFFF	Select bank by writing to A000

ASCII 8KB

Used by various games. A few cartridges of this type may also contain 8K SRAM, which is selected by setting one of the upper bank number bits

Memory	Mapper I/O Address
4000 - 5FFFF	Select bank by writing to 6000
6000 - 7FFFF	Select bank by writing to 6800
8000 - 9FFFF	Select bank by writing to 7000
A000 - BFFFF	Select bank by writing to 7800

ASCII 16KB

Among others this type is often used by many 64KB cartridges.

Memory	Mapper I/O Address
4000 - 7FFFF	Select bank by writing to 6000
8000 - BFFFF	Select bank by writing to 7000

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler

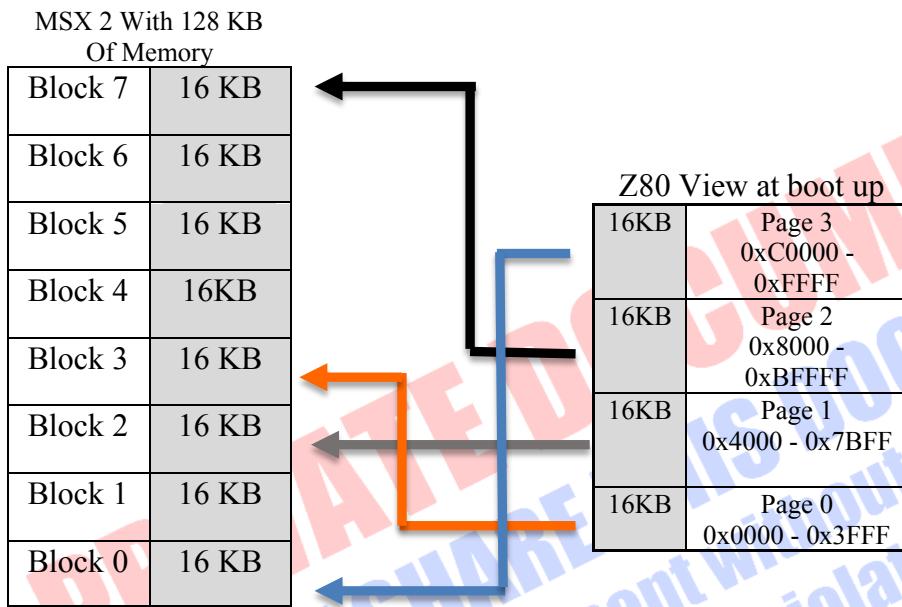
MSX Ram Memory Mapper

A memory mapper is a mechanism to "map" blocks of memory into specific memory range. It can apply to RAM, ROM or combination of RAM and ROM.

The memory mapper was introduced with the MSX2. The Z80 CPU used by the MSX computers can only "see" 64KB of Memory, even if the computer is set with more memory. This is why we need the memory mapper to connect this extra memory to the Z80 view.

As you can see on the above schematic, all the memory is split into 16KB Blocks. The Z80 is viewing only 64KB taken from the MSX computer memory.

The order and the connections between Pages, and blocks at MSX Boot up, are the one you can see on the schematic example. Page 0 (0x000 - 0x3FFF) is connected to the MSX Memory block 4 etc ...



With the Ram memory mapper mechanism, you can change the connection between pages and blocks, for example, connecting page 2 (0x8000 - 0xBFFFF) to the 5th MSX memory block instead of the 7th:

MSX 2 With 128 KB
Of Memory

Block 7	16 KB
Block 6	16 KB
Block 5	16 KB
Block 4	16KB
Block 3	16 KB
Block 2	16 KB
Block 1	16 KB
Block 0	16 KB

Z80 View after change	
16KB	Page 3 0xC0000 - 0xFFFF
16KB	Page 2 0x8000 - 0xBFFFF
16KB	Page 1 0x4000 - 0x7BFF
16KB	Page 0 0x0000 - 0x3FFF

Let's see how to change memory blocks ... It's really simple, you just have to write the number of the block in one of the four registers:

- 0xFC to set the block connected to Page 0 (0x000 - 0x3FFF)
- 0xFD to set the block connected to Page 1 (0x4000 - 0x7FFF)
- 0xFE to set the block connected to Page 2 (0x8000 - 0xBFFF)
- 0xFF to set the block connected to Page 3 (0xC000 - 0xFFFF)

Thus, to do the previous example, we have to write 5 in 0xFE

With FUSION_C : **OutPort (0xFE, 5);**

```
#include "fusion-c/header/msx_fusion.h"
void main (void)
{
    p= (char *) 0x8000;
    OutPort(0xFE,5);
    for (I=0; i<16383, i++)
    {
        p[i]=128;
    }
}
```

The maximum MSX's memory a Memory Mapper can handle is 4096 KB, thus 256 blocks of 16KB



Well, it's simple, but you have to be very careful when manipulating memory blocks, because when your program is switching between blocks, it can lose important variables or part of the program you already stored in the switched block.

Also, if the MSX computer is used with a lot of extensions, it's possible there are more than one active memory mapper. Also some MSX2 do not have a standard Memory Mapper, you have to be warned about that.

Important note:

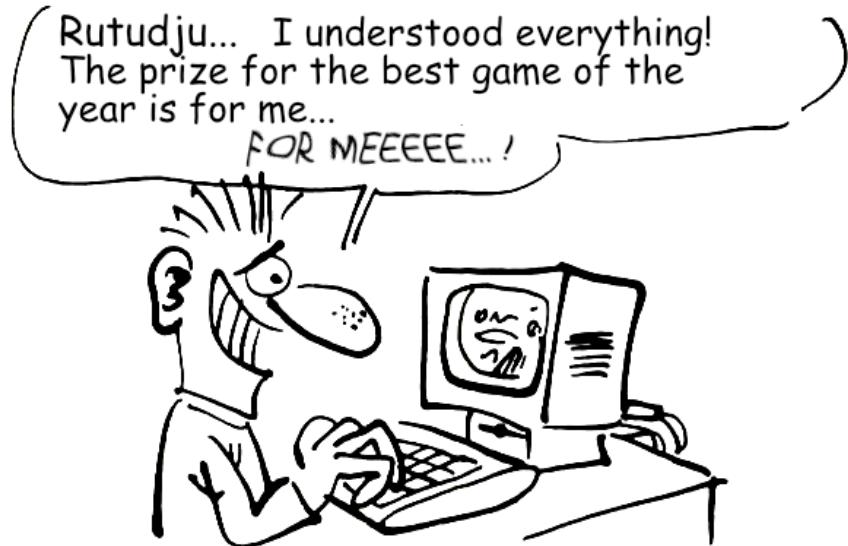
the “OUT” method cannot be used with MSX-DOS 2 environment.

In the case your program works on MSX-DOS 2, you must use the support of the MSX-DOS 2 mapper routines to be safe.

FUSION-C provides dedicated functions to use mapper with the MSX-DOS2 system.

See **memorymapper.h** functions.

MSX-DOS2 will automatically protect the memory that needs to be protected, and provide new 16KB blocks of memory for your page switching inside the whole memory available on the MSX computer.



PRIVATE
DO NOT SHARE THIS
Sharing This Document without authorization
is copyright violation

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler

MSX-DOS Operating System

MSX-DOS 1 works on MSX 1 and superior with a minimal of 64KB of memory. MSX-DOS 2 works on MSX 2 (eventually MSX1) and superior with at least 128KB of Ram memory minimal.

MSX-DOS 2 is composed of two parts, software, and Rom. In the world of the MSX Computers, only the MSX Turbo-R is equipped with the MSX-DOS2 Rom, but nowadays most of the mass storage devices, like the SD Cartridges MMC cartridges etc... have the MSX-DOS 2 Rom embedded.

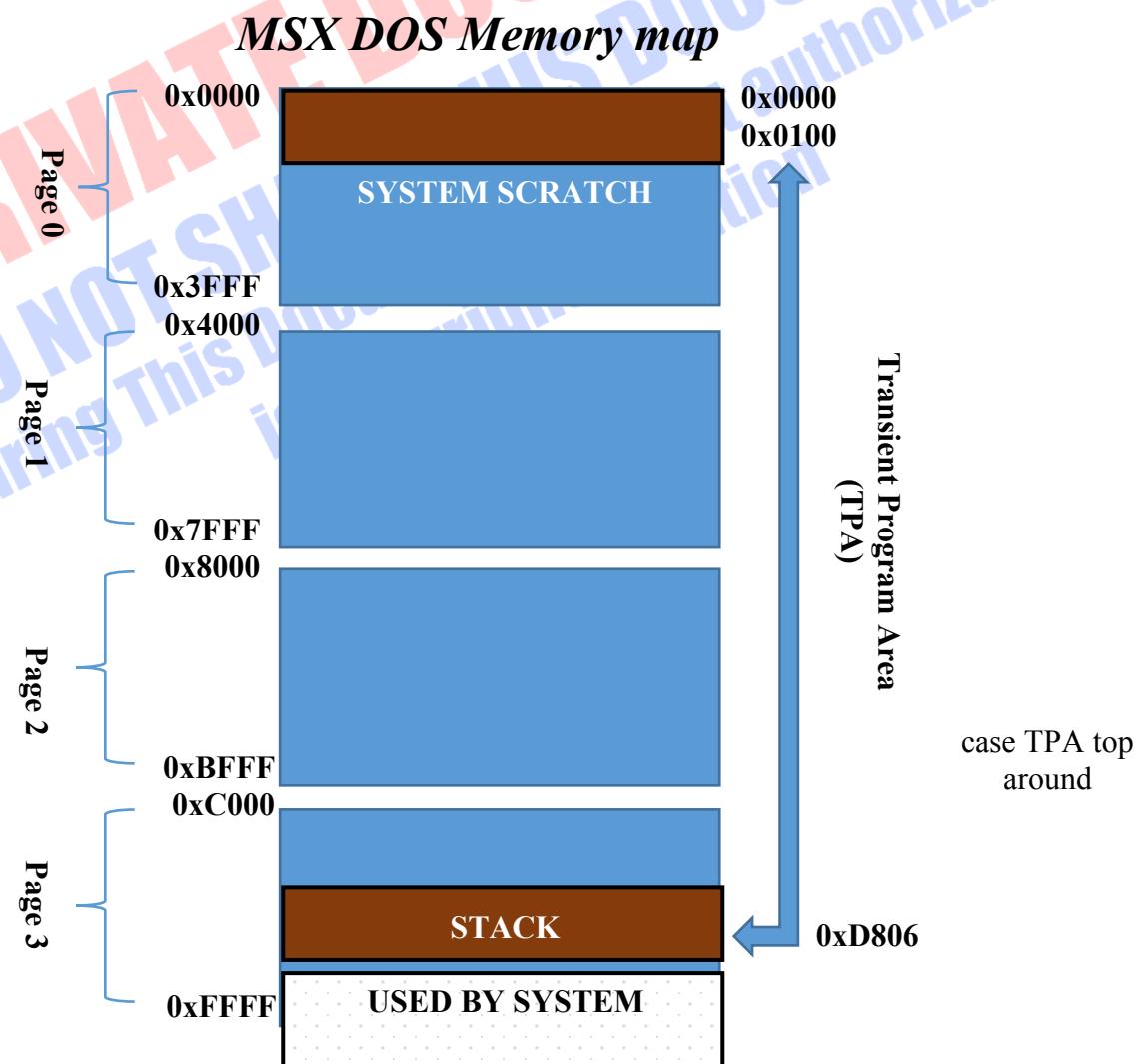
The available memory in MSX-DOS system is called TPA, (Transient Program Area). Any program loaded in MSX-DOS system starts at address 0x0100. Thus the available free space starts at 0x0100, and it ends at the top address of the TPA, minus the Stack length.

This TPA top address is not always the same. It depends of the mix computer, and other peripherals attached to it.

This address is stored in the System Scratch area at address 0x0006 and 0x0007, it's a 16-bit address. The Command **ReadTPA()**, will return this address for you. The size of the Stack, is growing and decreasing while your program is running. It depends of the variables used and stored. The command **ReadSP()** will return the lower address of the stack for you at a 'T' time.

The free space can be calculated like this: TPA – Stack length – 0x0100

If you are using MSX-DOS 2 you can use the MSX-DOS Memory mapper to enable and use other memory segments for your program. Please check MSX-DOS2 Ram mapper functions of the FUSION-C Library.



Media supported by MSX-DOS

MSX-DOS, which has a flexible file manager that does not depend on the physical structure of the disk, supports various media and uses 3.5 inch double density disks as standard. Either a one-sided disk called 1DD or two-sided disk called 2DD is used. Each of them uses either an 8-sector track format so four kinds of media can be used. The Microsoft formats for these four types are shown below.

	1DD, 9 Sectors	2DD, 9 sectors	1DD, 8 sectors	2DD, 8 sectors
Media ID	0x0F8	0x0F9	0x0FA	0x0FB
Number of sides	1	2	1	2
Tracks per side	80	80	80	80
Sectors per track	9	9	8	8
Bytes per sector	512	512	512	512
Cluster size (In sectors)	2	2	2	2
Fat size (in sectors)	2	3	1	2
Number of FATs	2	2	2	2
Number of recordable files	112	112	112	112

Information about the structure of data on the disk and how it is controlled is important when accessing the disk using system calls. This section begins with a description about "logical sectors" which are the basic units for exchanging data with the disk on MSX-DOS, and proceeds to the method of handling data with "files" which is more familiar to programmers.

Sectors

MSX-DOS can access most types of disk drives including the 3.5 inch 2DD and hard disks. For handling different media in the same way, the system call consider "logical sector" as the basic units of data on the disk. A logical sector is specified by numbers starting from 0.

Clusters

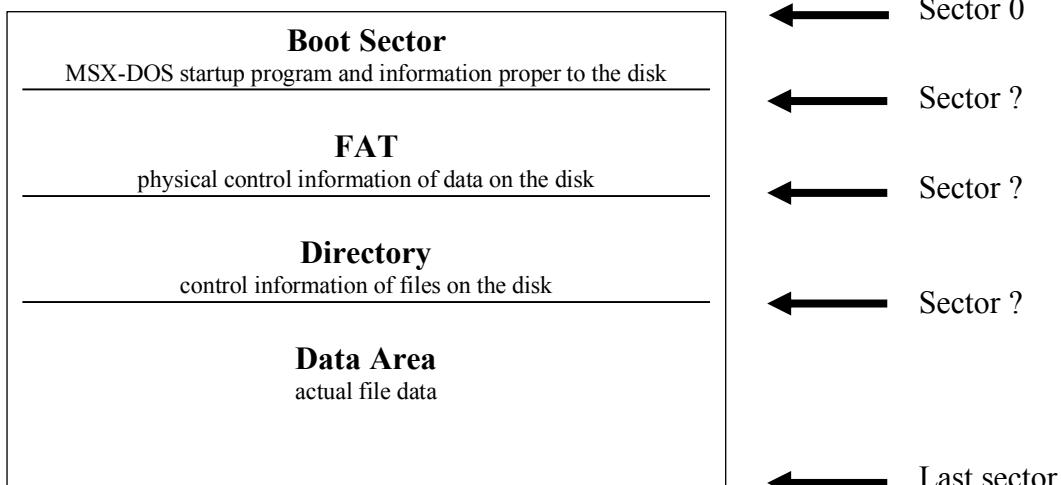
As long as system calls are used, a sector may be considered the basic unit of data as considered above. In fact, however, data on the disk is controlled in units of "clusters" which consists of multiple sectors. As described later in the FAT section, a cluster is specified by a serial number from 2 and the top of the data area corresponds to the location of cluster #2. For getting information about how many sectors a cluster has, use the system call function 0x1B (acquiring disk information).

Conversion from clusters to sectors

In a part of the directory or FCB, described later, the data location on the disk is indicated by the cluster. To use system calls to access data indicated by cluster, the relation of the correspondence between the cluster and the sector needs to be calculated. Since cluster #2 and the top sector of the data area reside in the same location, this can be done as follows:

1. Assume the given cluster number is C.
2. Examine the top sector of the data area (by reading DPB) (DPB : drive parameter block and boot sector) and assume it is S0.
3. Examine the number of sectors equivalent to one cluster (using function 0x1B) and assume it is n.
4. Use the formula $S = S0 + (C-2) * n$ to calculate sector numbers.

In MSX-DOS, sectors in the disk are divided into four areas, as shown below. The file data body written to the disk is recorded in the "data area" portion. Information for handling data is written in three areas. The below schematic shows the relation of the locations of these areas. The boot sector is always in sector #0, but the top sectors (FAT, directory, and data area) differ by media, so DPB should be referred to.

Disk contents

On MSX-DOS, the area "DPB" is allocated in the work area of memory for each connected drive, and information proper to each drive is recorded there.

MSX-DOS can handle most types of disk drives, because the differences between media can be compensated for by the process corresponding to each drive.

Information written on DPB, which is originally on the boot sector (sector #0) of the disk, is read at MSX-DOS startup. Note that the differences between the contents of the boot sector and DPB, as shown below. Data is arranged differently in the boot sector and the DPB.

Information of the boot sector		DPB Structure	
0x0B	1 Sector Size (in bytes)	Base	Drive Number
0x0C		+1	Media ID
0x0D	1 Cluster size (in sectors)	+2	Sector Size
0x0E	Number of unused sectors by MSX-DOS	+3	
0x0F		+4	Directory Mask
0x10		+5	Directory Shift
0x11	Number of FATs	+6	Cluster Mask
0x12	Number of directory entries	+7	Cluster Shift
0x13	(How many files can be created)	+8	Top Sector of Fat
0x14	Number of sectors per disk	+9	
0x15		+10	Number of FATs
0x16	Media ID	+11	Number of Directory entries
0x17	Size of FAT in sectors	+12	Top sector of data area
0x18		+13	
0x19	Number of tracks per sector	+14	Amount of cluster +1
0x1A		+15	
0x1B	Number of sides used (1 or 2)	+16	Number of sectors required for 1 FAT
0x1C		+17	Top sector of directory area
0x1D	Number of hidden sectors	+18	
		+19	FAT address in memory
		+20	

Use the system call Function 0x1B (disk information acquisition) to access the DPB. This system call returns the DPB address in memory and other information for each drive written on the boot sector.

The FCB (file control block)

Using information recorded in the directory area allows data to be treated as a "file". The advantage of this method is that the data location is not represented by an absolute number such as sector number or cluster number; instead, the file can be specified with a "name". The programmer need only specify the file name and the system will do all the work concerned with accessing the requested file. In other

C Library for MSX-DOS with SDCC compiler words, the programmer need not understand the details of which sectors the file occupies. In this case, FCB plays an important role for directories.

FCB is the area for storing information needed to handle files using system calls. Handling one file requires 37 bytes of memory each, as shown in figure below. Although the FCB can be located anywhere in memory, the address 0x05C is normally used to utilize MSX-DOS features.

Organization of FCB

	Byte	
	0	Drive Number
	1 to 11	FileName (8 bytes) + Extension (3 bytes)
	12 & 13	Current block number of blocks from the top of the file to the current block
	14 & 15	Record size (1 to 65535)
	16 to 19	File Size (1 to 42949667296)
*	20 & 21	Date
*	22 & 23	Time
**	24	Device ID
**	25	Directory location
**	26 & 27	Top cluster number of the file
**	28 & 29	Last cluster number accessed
	30 & 31	Relative location from top cluster of the file
	32	Current record
	33 to 36	Random record (Record order from the top of the file usually stores the last record made random access.)

FCB usages differ, depending on whether they use CP/M compatible system calls or additional system calls.

(*) When using version 2 of MSX-DOS, here is stored the volume-id of the disk, and should not be modified by the program.

(**) When using version 2 of MSX-DOS, here is stored internal information relative to the physical location of the file on the disk. The format of this information is different, and should not be modified by the program.

- **drive number** (0x0) Indicates the disk drive containing the file.(0 -> default drive, 1 -> A:, 2 -> B:...)
- **filename** (0x1 to 0x8) A filename can have up to 8 characters. When it has less than 8, the rest are filled in by spaces (0x20).
- **extension** (0x9 to 0xB) An extension can have up to 3 characters.
- **current block** (0xC to 0xD) Indicates the block number currently being referred by sequential access
- **record size** (0xE to 0xF) Specifies the size of data unit (record) to be read or written at once, in bytes.
- **file size** (0x10 to 0x13) Indicates the size of the file in bytes.
- **date** (0x14 to 0x15) Indicates date when a file was last written.
- **time** (0x16 to 0x17) Indicates time when a file was last written.
- **device ID** (0x18) When a peripheral is opened as a file, the value shown is specified for this device ID field. For normal disk files, the value of this field is 0x40 + drive number.
- **directory location** (0x19) Indicates the order of the directory entries of a file in the directory area.
- **top cluster** (0x1A to 0x1B) Indicates the top cluster of the file in the disk.
- **last cluster accessed** (0x1C to 0x1D) Indicates the last cluster accessed.
- **relative location from top cluster** of last cluster accessed (0x1E to 0x1F) Indicates the relative location from the top cluster of the last cluster accessed.
- **current record** (0x20) Indicates the record currently being referred to by sequential access
- **random record** (0x21 to 0x24) Specifies a record to be accessed by random access or random block access. Specifying a value from 1 to 63 for the record size field described above causes all four bytes from 0x21 to 0x24 to be used, where only three bytes from 0x21 to 0x23 have meaning when the record size is greater than 63

Reminder about boolean logical operators

AND operator		A & B		
Value A		Value B		Result
0	&	0	=	0
1	&	0	=	0
0	&	1	=	0
1	&	1	=	1

OR operator		A B		
Value A		Value B		Result
0		0	=	0
1		0	=	1
0		1	=	1
1		1	=	1

XOR operator		A ^ B		
Value A		Value B		Result
0	^	0	=	0
1	^	0	=	1
0	^	1	=	1
1	^	1	=	0

NOR operator		$\sim(A B)$		
Value A		Value B		Result
0		0	=	1
1		0	=	0
0		1	=	0
1		1	=	0

NAND operator		$\sim(A\&B)$		
Value A		Value B		Result
0		0	=	1
1		0	=	1
0		1	=	1
1		1	=	0

NOT operator		!A	
Value A		Result	
! 0			1
! 1			0

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler

Memento about C language

for Beginners / Intermediates users

(Results shown as an example may differ on a MSX system)

Most information of this chapter was extracted from
www.tutorialspoint.com

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation



Some facts about C	The Token Pasting (##) Operator
The C program structure	The Defined() Operator
Tokens in C	Operators
Semicolons	Arithmetic Operators
Comments	Relational Operators
Identifiers	Logical Operators
Keywords	Bitwise Operators
Whitespace in C	Assignment Operators
DATA Types	Misc Operators - sizeof & ternary
Integer Types	Operators Precedence in C
Floating-Point Types	Conditions and decision making in C
The void Type	Loops
The Variables	Loop Control Statements
Variable Definition in C	The Infinite Loop
Variable Declaration in C	Functions
Lvalues and Rvalues in C	Defining a Function
Constants and literals	Function Declarations
Integer Literals	Calling a Function
Floating-point Literals	Function Arguments
Character Constants	Scope range
String Literals	Local Variables
Defining Constants	Global Variables
The const Keyword	Formal Parameters
The Storage Class	Initializing Local and Global Variables
The auto Storage Class	Arrays
The register Storage Class	Declaring Arrays
The static Storage Class	Initializing Arrays
The extern Storage Class	Accessing Array Elements
What are Pointers	Arrays in Detail
How to Use Pointers	The pointers
NULL Pointers	Parameterized Macros
Pointers in Detail	The Header file
Strings	Include Syntax
Structures	Include Operation
Accessing Structure Members	Once-Only Headers
Structures as Function Arguments	Computed Includes
Pointers to Structures	Type Casting
Bit Fields in structures	Integer Promotion
Union	Usual Arithmetic Conversion
Defining a Union	Error Handling
Accessing Union Members	errno, perror(). and strerror()
Bit Field	Divide by Zero Errors
Bit Field Declaration	Program Exit Status
Typedef	Recursion
Input and Output	Number Factorial
The Standard Files	Fibonacci Series
The getchar() and putchar() Functions	Variable Arguments
The gets() and puts() Functions	Memory Management
The scanf() and printf() Functions	Allocating Memory Dynamically
The Preprocessor	Resizing and Releasing Memory
Preprocessors Examples	Command Line Arguments
Predefined Macros	The Stringize (#) Operator
Preprocessor Operators	

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Some facts about C

C is a general-purpose language which has been closely associated with the **UNIX** operating system for which it was developed - since the system and most of the programs that run it are written in C.

Many of the important ideas of C stem from the language **BCPL**, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language **B**, which was written by Ken Thompson in 1970 at Bell Labs, for the first UNIX system on a **DEC PDP-7**. **BCPL** and **B** are « type less » languages whereas C provides a variety of data types.

In 1972 Dennis Ritchie at Bell Labs writes C and in 1978 the publication of The C Programming Language by Kernighan & Ritchie caused a revolution in the computing world.

In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or « ANSI C », was completed late 1988.

A C program can vary from 3 lines to millions of lines and it should be written into one or more text files with extension ".c"; for example, *hello.c*. You can use any text editor to write your C program into a file.

The source code written in source file is the human readable source for your program. It needs to be "compiled", into machine language so that your CPU can actually execute the program as per the instructions given.

The compiler compiles the source codes into final executable programs. The most frequently used and available free compiler is the GNU C/C++ compiler, otherwise you can have compilers either from HP or Solaris if you have the respective operating systems.

The C program structure

A C program basically consists of the following parts –

- Preprocessor Commands
- Functions
- Variables
- Statements & Expressions
- Comments

Let us look at a simple code that would print the words "Hello World"

```
# include <stdio.h>
int main()
{
    /* my First Program in C */
    printf(" Hello World ! \n\r ");
    return();
}
```

Let us take a look at the various parts of the above program –

- The first line of the program `#include <stdio.h>` is a preprocessor command, which tells a C compiler to include stdio.h file before going to actual compilation.
- The next line `int main()` is the main function where the program execution begins.
- The next line `/*...*/` will be ignored by the compiler and it has been put to add additional comments in the program. So such lines are called comments in the program.
- The next line `printf(...)` is another function available in C which causes the message "Hello, World!" to be displayed on the screen.
- The next line `return 0;` terminates the main() function and returns the value 0.

You have seen the basic structure of a C program, so it will be easy to understand other basic building blocks of the C programming language.

Tokens in C

A C program consists of various tokens and a token is either a keyword, an identifier, a constant, a string literal, or a symbol. For example, the following C statement consists of five tokens

```
printf("Hello, World! \n");
```

The individual tokens are:

```
printf ( "Hello, World! \n" ) ;
```

Semicolons

In a C program, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

Given below are two different statements

```
printf("Hello, World! \n"); return 0;
```

Comments

Comments are like helping text in your C program and they are ignored by the compiler. They start with /* and terminate with the characters */ as shown below:

```
/* my first program in C */
```

You cannot have comments within comments and they do not occur within a string or character literals.

Identifiers

A C identifier is a name used to identify a variable, function, or any other user-defined item. An identifier starts with a letter A to Z, a to z, or an underscore '_' followed by zero or more letters, underscores, and digits (0 to 9).

C does not allow punctuation characters such as @, \$, and % within identifiers. **C is a case-sensitive programming language.**

Thus, *Manpower* and *manpower* are two different identifiers in C. Here are some examples of acceptable identifiers:

```
mohd    zara   abc  move_name a_123 myname50 _temp j a23b9    retVal
```

Keywords

The following list shows the reserved words in C. These reserved words may not be used as constants or variables or any other identifier names.

auto	else	long	switch
break	enum	register	typedef
case	extern	return	union
char	float	short	unsigned
const	for	signed	void
continue	goto	sizeof	volatile
default	if	static	while
do	int	struct	_Packed
double			

Whitespace in C

A line containing only whitespace, possibly with a comment, is known as a blank line, and a C compiler totally ignores it.

Whitespace is the term used in C to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins. Therefore, in the following statement :

```
int age;
```

there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them. On the other hand, in the following statement:

```
fruit = apples + oranges; // get the total fruit
```

no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish to increase readability.

DATA Types

Data types in C refer to an extensive system used for declaring variables or functions of different types. The type of a variable determines how much space it occupies in storage and how the bit pattern stored is interpreted.

The types in C can be classified as follows:

Types & Description	
1	Basic Types They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.
2	Enumerated types They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
3	The type void The type specifier <i>void</i> indicates that no value is available.
4	Derived types They include (a) Pointer types, (b) Array types, (c) Structure types, (d) Union types and (e) Function types.

The array types and structure types are referred collectively as the aggregate types. The type of a function specifies the type of the function's return value. We will see the basic types in the following section, whereas other types will be covered in the upcoming chapters.

Integer Types

The following table provides the details of standard integer types with their storage sizes and value ranges:

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned long	4 bytes	0 to 4,294,967,295

To get the exact size of a type or a variable on a particular platform, you can use the **sizeof** operator. The expression *sizeof(type)* yields the storage size of the object or type in bytes. Given below is an example to get the size of int type on any machine.

```
#include <stdio.h>
#include <limits.h>
int main()
{
    printf("Storage size for int : %d \n", sizeof(int));           return 0;
}
```

When you compile and execute the above program, it produces the following result on Linux

```
Storage size for int : 4
```

Floating-Point Types

The following table provide the details of standard floating-point types with storage sizes and value ranges and their precision. **SDCC only support float single precision (4 bytes).**

Note if all these routines are used simultaneously the data space might overflow. For serious floating point usage, it is strongly recommended that the Large model be used (in which case the floating point routines mentioned above will need to recompiled with the --model-Large option)

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

The header file float.h defines macros that allow you to use these values and other details about the binary representation of real numbers in your programs. The following example prints the storage space taken by a float type and its range values.

SDCC floating point support routines are derived from gcc's floatlib.c and consists of the following routines

- `_fsadd.c` - add floating point numbers.
- `_fssub.c` - subtract floating point numbers
- `_fsdiv.c` - divide floating point numbers
- `_fsmul.c` - multiply floating point numbers
- `_fs2uchar.c` - convert floating point to unsigned char
- `_fs2char.c` - convert floating point to signed char.
- `_fs2uint.c` - convert floating point to unsigned int.
- `_fs2int.c` - convert floating point to signed int.
- `_fs2ulong.c` - convert floating point to unsigned long.
- `_fs2long.c` - convert floating point to signed long.
- `_uchar2fs.c` - convert charto floating point
- `_char2fs.c` - convert char to floating point number
- `_uint2fs.c` - convert unsigned int to floating point
- `_int2fs.c` - convert int to floating point numbers
- `_ulong2fs.c` - convert unsigned long to floating point number
- `_long2fs.c` - convert long to floating point number.

The void Type

The void type specifies that no value is available. It is used in three kinds of situations

Types & Description	
1	Function returns as void There are various functions in C which do not return any value or you can say they return void. A function with no return value has the return type as void. For example, void exit (int status);
2	Function arguments as void There are various functions in C which do not accept any parameter. A function with no parameter can accept a void. For example, int rand(void);
3	Pointers to void A pointer of type void * represents the address of an object, but not its type. For example, a memory allocation function void *malloc(size_t size); returns a pointer to void which can be casted to any data type.



PRIVAT
DO NOT S
Sharing This Document
is copyright v...

The Variables

A variable is nothing but a name given to a storage area that our programs can manipulate. Each variable in C has a specific type, which determines the size and layout of the variable's memory; the range of values that can be stored within that memory; and the set of operations that can be applied to the variable. The name of a variable can be composed of letters, digits, and the underscore character. It must begin with either a letter or an underscore. Upper and lower case letters are distinct because C is case-sensitive. Based on the basic types explained in the previous chapter, there will be the following basic variable types

Type & Description	
1	char Typically, a single octet (one byte). This is an integer type.
2	int The most natural size of integer for the machine.
3	float A single-precision floating point value.
4	double A double-precision floating point value.
5	void Represents the absence of type.

C programming language also allows to define various other types of variables, which we will cover in subsequent chapters like Enumeration, Pointer, Array, Structure, Union, etc. For this chapter, let us study only basic variable types.

Variable Definition in C

A variable definition tells the compiler where and how much storage to create for the variable. A variable definition specifies a data type and contains a list of one or more variables of that type as follows

```
type variable_list;
```

Here, **type** must be a valid C data type including char, w_char, int, float, double, bool, or any user-defined object; and **variable_list** may consist of one or more identifier names separated by commas. Some valid declarations are shown here

```
int i, j, k; char c, ch; float f, salary; double d;
```

The line **int i, j, k;** declares and defines the variables i, j, and k; which instruct the compiler to create variables named i, j and k of type int.

Variables can be initialized (assigned an initial value) in their declaration. The initializer consists of an equal sign followed by a constant expression as follows:

```
type variable_name = value;
```

Some examples are:

```
extern int d = 3, f = 5;           // declaration of d and f.  
int d = 3, f = 5;                 // definition and initializing d and f.  
byte z = 22;                      // definition and initializes z.  
char x = 'x';                    // the variable x has the value 'x'.
```

For definition without an initializer: variables with static storage duration are implicitly initialized with NULL (all bytes have the value 0); the initial value of all other variables are undefined.

Variable Declaration in C

A variable declaration provides assurance to the compiler that there exists a variable with the given type and name so that the compiler can proceed for further compilation without requiring the complete detail about the variable. A variable definition has its meaning at the time of compilation only, the compiler needs actual variable definition at the time of linking the program.

A variable declaration is useful when you are using multiple files and you define your variable in one of the files which will be available at the time of linking of the program. You will use the keyword **extern** to declare a variable at any place. Though you can declare a variable multiple times in your C program, it can be defined only once in a file, a function, or a block of code.

Example :

Try the following example, where variables have been declared at the top, but they have been defined and initialized inside the main function :

```
#include <stdio.h>
// Variable declaration:
extern int a, b;
extern int c;
extern float f;
int main ()
{
    /* variable definition: */
    int a, b;
    int c;
    float f;
    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;
    printf("value of c : %d \n", c);
    f = 70.0/3.0;
    printf("value of f : %f \n", f);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
value of c : 30 value of f : 23.333334
```

The same concept applies on function declaration where you provide a function name at the time of its declaration and its actual definition can be given anywhere else. For example:

```
// function declaration
int func();
int main()
{
    // function call
    int i = func();
}
// function definition
int func()
{
    return 0;
}
```

Lvalues and Rvalues in C

There are two kinds of expressions in C

lvalue – Expressions that refer to a memory location are called "lvalue" expressions. An lvalue may appear as either the left-hand or right-hand side of an assignment.

rvalue – The term rvalue refers to a data value that is stored at some address in memory. An rvalue is an expression that cannot have a value assigned to it which means an rvalue may appear on the right-hand side but not on the left-hand side of an assignment.

Variables are lvalues and so they may appear on the left-hand side of an assignment. Numeric literals are rvalues and so they may not be assigned and cannot appear on the left-hand side. Take a look at the following valid and invalid statements

```
int g = 20; // valid statement
10 = 20; // invalid statement; would generate compile-time error
```

Constants and literals

Constants refer to fixed values that the program may not alter during its execution. These fixed values are also called **literals**.

Constants can be of any of the basic data types like *an integer constant, a floating constant, a character constant, or a string literal*. There are enumeration constants as well.

Constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be upper-case or lower-case and can be in any order.

Here are some examples of integer literals

```
212          /* Legal */
215u         /* Legal */
0xFeeL       /* Legal */
078          /* Illegal: 8 is not an octal digit */ 0
32UU         /* Illegal: cannot repeat a suffix */
```

Following are other examples of various types of integer literals

```
85           /* decimal */
0213         /* octal */
0x4b         /* hexadecimal */
30           /* int */
30u          /* unsigned int */
30l          /* long */
30ul         /* unsigned long */
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing decimal form, you must include the decimal point, the exponent, or both; and while representing exponential form, you must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals

```
3.14159      /* Legal */
314159E-5L   /* Legal */
510E         /* Illegal: incomplete exponent */
210f         /* Illegal: no decimal or exponent */
.e55         /* Illegal: missing integer or fraction */
```

Character Constants

Character literals are enclosed in single quotes, e.g., 'x' can be stored in a simple variable of **char** type. A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0'). There are certain characters in C that represent special meaning when preceded by a backslash for example, newline (\n) or tab (\t).

Here, you have a list of such escape sequence codes

Following is the example to show a few escape sequence characters:

```
#include <stdio.h>
int main()
{
    printf("Hello\tWorld\n\n");
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Hello World
```

String Literals

String literals or constants are enclosed in double quotes """. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separating them using white spaces. Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"  "hello, \ dear"  "hello, " "d" "ear"
```

Defining Constants

There are two simple ways in C to define constants –

- Using **#define** preprocessor.
- Using **const** keyword.

The #define Preprocessor

Given below is the form to use **#define** preprocessor to define a constant

```
#define identifier value
```

The following example explains it in detail

```
#include <stdio.h>
#define LENGTH 10
#define WIDTH 5
#define NEWLINE '\n'
int main()
{
    int area;
    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
value of area : 50
```

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows:

```
const type variable = value;
```

The following example explains it in detail

```
#include <stdio.h>
int main()
{
    const int LENGTH = 10;
    const int WIDTH = 5;
    const char NEWLINE = '\n';
    int area;
    area = LENGTH * WIDTH;
    printf("value of area : %d", area);
    printf("%c", NEWLINE);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
value of area : 50
```

Note: that it is a good programming practice to define constants in CAPITALS.

The Storage Class

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C Program. They precede the type that they modify. We have four different storage classes in a C program

- auto
- register
- static
- extern

The auto Storage Class

The **auto** storage class is the default storage class for all local variables.

```
{
    int mount;
    auto int month;
}
```

The example above defines two variables with in the same storage class. 'auto' can only be used within functions, i.e., local variables.

The register Storage Class

The **register** storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
    register int miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

The static Storage Class

The **static** storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.

The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.

In C programming, when **static** is used on a global variable, it causes only one copy of that member to be shared by all the objects of its class.

```
#include <stdio.h>
/* function declaration */
void func(void);
static int count = 5; /* global variable */

main()
{
    while(count--)
    {
        func();
    }
return 0;
}
/* function definition */
void func( void )
{
    static int i = 5; /* local static variable */
    i++;
    printf("i is %d and count is %d\n", i, count);
}
```

When the above code is compiled and executed, it produces the following result

```
i is 6 and count is 4
i is 7 and count is 3
i is 8 and count is 2
i is 9 and count is 1
i is 10 and count is 0
```

The extern Storage Class

The **extern** storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern', the variable cannot be initialized however, it points the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will also be used in other files, then *extern* will be used in another file to provide the reference of defined variable or function. Just for understanding, *extern* is used to declare a global variable or function in another file.

The *extern* modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.c

```
#include <stdio.h>
int count ;
extern void write_extern();
main()
{
    count = 5;
    write_extern();
}
```

Second File: support.c

```
#include <stdio.h>
extern int count;
void write_extern(void)
{
    printf("count is %d\n", count);
}
```

Here, *extern* is being used to declare *count* in the second file, whereas it has its definition in the first file, main.c. Now, compile these two files as follows:

```
> sdcc main.c support.c
```

It will produce the executable program **a.out**. When this program is executed, it produces the following result:

```
count is 5
```

Operators

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators –

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Misc Operators.

We will, in this chapter, look into the way each operator works.

Arithmetic Operators

The following table shows all the arithmetic operators supported by the C language. Assume variable **A** holds 10 and variable **B** holds 20

Operator	Description	Example
+	Adds two operands.	$A + B = 30$
-	Subtracts second operand from the first.	$A - B = -10$
*	Multiplies both operands.	$A * B = 200$
/	Divides numerator by de-numerator.	$B / A = 2$
%	Modulus Operator and remainder of after an integer division.	$B \% A = 0$
++	Increment operator increases the integer value by one.	$A++ = 11$
--	Decrement operator decreases the integer value by one.	$A-- = 9$

Relational Operators

The following table shows all the relational operators supported by C. Assume variable **A** holds 10 and variable **B** holds 20

Operator	Description	Example
==	Checks if the values of two operands are equal or not. If yes, then the condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not. If the values are not equal, then the condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand. If yes, then the condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand. If yes, then the condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand. If yes, then the condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand. If yes, then the condition becomes true.	(A <= B) is true.

Logical Operators

Following table shows all the logical operators supported by C language. Assume variable **A** holds 1 and variable **B** holds 0

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for `&`, `|`, and `^` is as follows:

p	q	p & q	p q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume A = 60 and B = 13 in binary format, they will be as follows:

A = 0011 1100

B = 0000 1101

A&B = 0000 1100

A|B = 0011 1101

A^B = 0011 0001

\sim A = 1100 0011

The following table lists the bitwise operators supported by C. Assume variable 'A' holds 60 and variable 'B' holds 13

Operator	Description	Example
<code>&</code>	Binary AND Operator copies a bit to the result if it exists in both operands.	$(A \& B) = 12$, i.e., 0000 1100
<code> </code>	Binary OR Operator copies a bit if it exists in either operand.	$(A B) = 61$, i.e., 0011 1101
<code>^</code>	Binary XOR Operator copies the bit if it is set in one operand but not both.	$(A ^ B) = 49$, i.e., 0011 0001
<code>~</code>	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	$(\sim A) = -60$, i.e., 1100 0100 in 2's complement form.
<code><<</code>	Binary Left Shift Operator. The left operand's value is moved left by the number of bits specified by the right operand.	$A << 2 = 240$ i.e., 1111 0000
<code>>></code>	Binary Right Shift Operator. The left operand's value is moved right by the number of bits specified by the right operand.	$A >> 2 = 15$ i.e., 0000 1111
Nor	NOR Must be programmed like this	$\text{nor} = \sim(a b)$
Nand	NAND Must be programmed like this	$\text{nand} = \sim(a\&b)$

Assignment Operators

The following table lists the assignment operators supported by the C language

Operator	Description	Example
=	Simple assignment operator. Assigns values from right side operands to left side operand	C = A + B will assign the value of A + B to C
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
-=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the left operand.	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
=	Bitwise inclusive OR and assignment operator.	C = 2 is same as C = C 2

Misc Operators – sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **:** supported by the C Language.

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
 Sharing This Document without authorization
 is copyright violation

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, $x = 7 + 3 * 2$; here, x is assigned 13, not 20 because operator $*$ has a higher precedence than $+$, so it first gets multiplied with $3*2$ and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	$0 \ [] \ -> \ . \ + + \ - -$	Left to right
Unary	$+ \ - \ ! \ \sim \ + + \ - - \ (\text{type})^* \ \& \ \text{sizeof}$	Right to left
Multiplicative	$* \ / \ %$	Left to right
Additive	$+ \ -$	Left to right
Shift	$<< \ >>$	Left to right
Relational	$< \ <= \ > \ >=$	Left to right
Equality	$== \ !=$	Left to right
Bitwise AND	$\&$	Left to right
Bitwise XOR	\wedge	Left to right
Bitwise OR	$ $	Left to right
Logical AND	$\&\&$	Left to right
Logical OR	$\ $	Left to right
Conditional	$? :$	Right to left
Assignment	$= \ += \ -= \ *= \ /= \ \%=\ >>= \ <<= \ \&= \ ^= \ =$	Right to left
Comma	,	Left to right

Conditions and decision making in C

Decision making structures require that the programmer specifies one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.

C programming language provides the following types of decision making statements.

Statement & Description	
1	<u>if statement</u> An if statement consists of a boolean expression followed by one or more statements.
2	<u>if...else statement</u> An if statement can be followed by an optional else statement , which executes when the Boolean expression is false.
3	<u>nested if statements</u> You can use one if or else if statement inside another if or else if statement(s).
4	<u>switch statement</u> A switch statement allows a variable to be tested for equality against a list of values.
5	<u>nested switch statements</u> You can use one switch statement inside another switch statement(s).

The ?: Operator

We have covered **conditional operator** **? :** in the previous chapter which can be used to replace **if...else** statements. It has the following general form

Exp1 ? Exp2 : Exp3;

Where Exp1, Exp2, and Exp3 are expressions. Notice the use and placement of the colon.

The value of a **?** expression is determined like this –

- Exp1 is evaluated. If it is true, then Exp2 is evaluated and becomes the value of the entire **?** expression.
- If Exp1 is false, then Exp3 is evaluated and its value becomes the value of the expression.

Loops

You may encounter situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times.

C programming language provides the following types of loops to handle looping requirements.

Loop Type & Description	
1	<u>while loop</u> Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	<u>for loop</u> Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	<u>do...while loop</u> It is more like a while statement, except that it tests the condition at the end of the loop body.
4	<u>nested loops</u> You can use one or more loops inside any other while, for, or do..while loop.

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

Control Statement & Description	
1	<u>break statement</u> Terminates the loop or switch statement and transfers execution to the statement immediately following the loop or switch.
2	<u>continue statement</u> Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	<u>goto statement</u> Transfers control to the labeled statement.

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **forloop** is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>
int main ()
{
for( ; ; )
{
    printf("This loop will run forever.\n");
}
return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the `for(;;)` construct to signify an infinite loop.

Functions

A function is a group of statements that together perform a task. Every C program has at least one function, which is **main()**, and all the most trivial programs can define additional functions.

You can divide up your code into separate functions. How you divide up your code among different functions is up to you, but logically the division is such that each function performs a specific task.

A function **declaration** tells the compiler about a function's name, return type, and parameters. A function **definition** provides the actual body of the function.

The C standard library provides numerous built-in functions that your program can call. For example, **strcat()** to concatenate two strings, **memcpy()** to copy one memory location to another location, and many more functions.

A function can also be referred as a method or a sub-routine or a procedure, etc.

Defining a Function

The general form of a function definition in C programming language is as follows

```
return_type function_name( parameter list )
{
    body of the function
}
```

A function definition in C programming consists of a *function header* and a *function body*. Here are all the parts of a function –

- **Return Type** – A function may return a value. The **return_type** is the data type of the value the function returns. Some functions perform the desired operations without returning a value. In this case, the **return_type** is the keyword **void**.
- **Function Name** – This is the actual name of the function. The function name and the parameter list together constitute the function signature.
- **Parameters** – A parameter is like a placeholder. When a function is invoked, you pass a value to the parameter. This value is referred to as actual parameter or argument. The parameter list refers to the type, order, and number of the parameters of a function. Parameters are optional; that is, a function may contain no parameters.
- **Function Body** – The function body contains a collection of statements that define what the function does.

Example:

Given below is the source code for a function called **max()**. This function takes two parameters **num1** and **num2** and returns the maximum value between the two

```
/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

Function Declarations

A function **declaration** tells the compiler about a function name and how to call the function. The actual body of the function can be defined separately.

A function declaration has the following parts

```
return_type function_name( parameter list );
```

For the above defined function max(), the function declaration is as follows

```
int max(int num1, int num2);
```

Parameter names are not important in function declaration only their type is required, so the following is also a valid declaration

```
int max(int, int);
```

Function declaration is required when you define a function in one source file and you call that function in another file. In such case, you should declare the function at the top of the file calling the function.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Calling a Function

While creating a C function, you give a definition of what the function has to do. To use a function, you will have to call that function to perform the defined task.

When a program calls a function, the program control is transferred to the called function. A called function performs a defined task and when its return statement is executed or when its function-ending closing brace is reached, it returns the program control back to the main program.

To call a function, you simply need to pass the required parameters along with the function name, and if the function returns a value, then you can store the returned value. **Example:**

```
#include <stdio.h>
/* function declaration */
int max(int num1, int num2);
int main ()
{
    /* local variable definition */
    int a = 100;
    int b = 200;
    int ret;
    /* calling a function to get max value */
    ret = max(a, b);
    printf( "Max value is : %d\n", ret );
    return 0;
}
/* function returning the max between two numbers */
int max(int num1, int num2)
{
    /* local variable declaration */
    int result;
    if (num1 > num2)
        result = num1;
    else
        result = num2;
    return result;
}
```

We have kept max() along with main() and compiled the source code. While running the final executable, it would produce the following result

Max value is : 200

Function Arguments

If a function is to use arguments, it must declare variables that accept the values of the arguments. These variables are called the **formal parameters** of the function.

Formal parameters behave like other local variables inside the function and are created upon entry into the function and destroyed upon exit.

While calling a function, there are two ways in which arguments can be given to a function

Call Type & Description	
1	<u>Call by value</u> This method copies the actual value of an argument into the formal parameter of the function. In this case, changes made to the parameter inside the function have no effect on the argument.
2	<u>Call by reference</u> This method copies the address of an argument into the formal parameter. Inside the function, the address is used to access the actual argument used in the call. This means that changes made to the parameter affect the argument.

By default, C uses **call by value** to pass arguments. In general, it means the code within a function cannot alter the arguments used to call the function.

Scope range

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formalparameters**.

Let us understand what are **local** and **global** variables, and **formalparameters**.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. The following example shows how local variables are used. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>
int main ()
{
    /* local variable declaration */
    int a, b;
    int c;
    /* actual initialization */
    a = 10;
    b = 20;
    c = a + b;
    printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
    return 0;
}
```

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>
/* global variable declaration */
int g;
int main ()
{
    /* local variable declaration */
    int a, b;
    /* actual initialization */
    a = 10;
    b = 20;
    g = a + b;
    printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
    return 0;
}
```

A program can have same name for local and global variables but the value of local variable inside a function will take preference. Here is an example

```
#include <stdio.h>
/* global variable declaration */
int g = 20;
int main ()
{
    /* local variable declaration */
    int g = 10;
    printf ("value of g = %d\n", g);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
value of g = 10
```

Formal Parameters

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables. Following is an example

```
#include <stdio.h>
/* global variable declaration */
int a = 20;
int main ()
{
    /* local variable declaration in main function */
    int a = 10;
    int b = 20;
    int c = 0;
    printf ("value of a in main() = %d\n", a);
    c = sum( a, b );
    printf ("value of c in main() = %d\n", c );
    return 0;
}
/* function to add two integers */
int sum(int a, int b)
{
    printf ("value of a in sum() = %d\n", a );
    printf ("value of b in sum() = %d\n", b );
    return a + b;
}
```

When the above code is compiled and executed, it produces the following result

```
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows

Data Type	Initial Default Value
int	0
char	'\0'
float	0
double	0
pointer	NULL

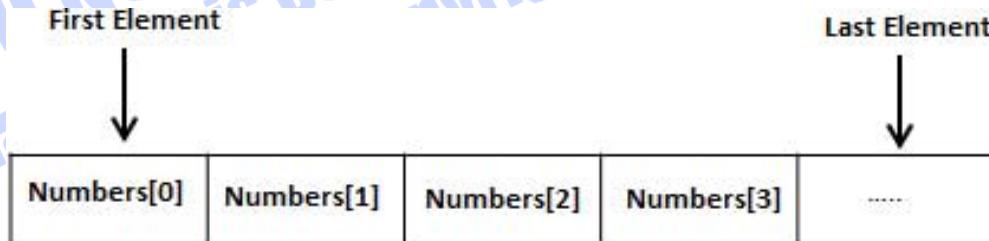
It is a good programming practice to initialize variables properly, otherwise your program may produce unexpected results, because uninitialized variables will take some garbage value already available at their memory location.

Arrays

Arrays a kind of data structure that can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.

Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.

All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays

To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows :

```
type arrayName [ arraySize ];
```

This is called a *single-dimensional* array. The **arraySize** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type **double**, use this statement:

```
double balance[10];
```

Here *balance* is a variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize an array in C either one by one or using a single statement as follows:

```
double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

The number of values between braces {} cannot be larger than the number of elements that we declare for the array between square brackets []. If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write

```
double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};
```

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array

```
balance[4] = 50.0;
```

The above statement assigns the 5th element in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called the base index and the last index of an array will be total size of the array minus 1. Shown below is the pictorial representation of the array we discussed above:

	0	1	2	3	4
balance	1000.0	2.0	3.4	7.0	50.0

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

```
double salary = balance[9];
```

The above statement will take the 10th element from the array and assign the value to salary variable. The following example Shows how to use all the three above mentioned concepts : declaration, assignment, and accessing arrays:

```
#include <stdio.h>
int main ()
{
    int n[ 10 ];
    /* n is an array of 10 integers */
    int i, j;
    /* initialize elements of array n to 0 */
    for ( i = 0; i < 10; i++ )
    {
        n[ i ] = i + 100;      /* set element at location i to i + 100 */
    }
    /* output each array element's value */
    for ( j = 0; j < 10; j++ )
    {
        printf("Element[%d] = %d\n", j, n[j] );
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Element[0] = 100 Element[1] = 101 Element[2] = 102 Element[3] = 103 Element[4] = 104
Element[5] = 105 Element[6] = 106 Element[7] = 107 Element[8] = 108 Element[9] = 109
```

Arrays in Detail

Arrays are important to C and should need a lot more attention. The following important concepts related to array should be clear to a C programmer

Concept & Description	
1	<u>Multi-dimensional arrays</u> C supports multidimensional arrays. The simplest form of the multidimensional array is the two-dimensional array.
2	<u>Passing arrays to functions</u> You can pass to the function a pointer to an array by specifying the array's name without an index.
3	<u>Return array from a function</u> C allows a function to return an array.
4	<u>Pointer to an array</u> You can generate a pointer to the first element of an array by simply specifying the array name, without any index.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

The pointers

Pointers in C are easy and fun to learn. Some C programming tasks are performed more easily with pointers, and other tasks, such as dynamic memory allocation, cannot be performed without using pointers. So it becomes necessary to learn pointers to become a perfect C programmer. Let's start learning them in simple and easy steps.

As you know, every variable is a memory location and every memory location has its address defined which can be accessed using ampersand (&) operator, which denotes an address in memory. Consider the following example, which prints the address of the variables defined

```
#include <stdio.h>
int main ()
{
    int var1;
    char var2[10];
    printf("Address of var1 variable: %x\n", &var1 );
    printf("Address of var2 variable: %x\n", &var2 );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Address of var1 variable: bff5a400
Address of var2 variable: bff5a3f6
```

What are Pointers

A **pointer** is a variable whose value is the address of another variable, i.e., direct address of the memory location. Like any variable or constant, you must declare a pointer before using it to store any variable address. The general form of a pointer variable declaration is:

```
type *var-name;
```

Here, **type** is the pointer's base type; it must be a valid C data type and **var-name** is the name of the pointer variable. The asterisk * used to declare a pointer is the same asterisk used for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Take a look at some of the valid pointer declarations

```
int    *ip;      /* pointer to an integer */
double *dp;      /* pointer to a double */
float  *fp;      /* pointer to a float */
char   *ch;      /* pointer to a character */
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to.

How to Use Pointers

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations.

```
#include <stdio.h>
int main ()
{
    int var = 20; /* actual variable declaration */
    int *ip; /* pointer variable declaration */
    ip = &var; /* store address of var in pointer variable */
    printf("Address of var variable: %x\n", &var ); /* address stored in pointer variable */
    printf("Address stored in ip variable: %x\n", ip ); /* access the value using the pointer */
    printf("Value of *ip variable: %d\n", *ip );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Address of var variable: bffd8b3c
Address stored in ip variable: bffd8b3c
Value of *ip variable: 20
```

NULL Pointers

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **nullpointer**. The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program:

```
#include <stdio.h>
int main ()
{
    int *ptr = NULL;
    printf("The value of ptr is : %x\n", ptr );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
The value of ptr is 0
```

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows:

```
if(ptr) /* succeeds if p is not null */
if(!ptr) /* succeeds if p is null */
```

Pointers in Detail

Pointers have many but easy concepts and they are very important to C programming. The following important pointer concepts should be clear to any C programmer

Concept & Description	
1	<u>Pointer arithmetic</u> There are four arithmetic operators that can be used in pointers: <code>++</code> , <code>--</code> , <code>+</code> , <code>-</code>
2	<u>Array of pointers</u> You can define arrays to hold a number of pointers.
3	<u>Pointer to pointer</u> C allows you to have pointer on a pointer and so on.
4	<u>Passing pointers to functions in C</u> Passing an argument by reference or by address enable the given argument to be changed in the calling function by the called function.
5	<u>Return pointer from functions in C</u> C allows a function to return a pointer to the local variable, static variable, and dynamically allocated memory as well.

Strings

Strings are actually one-dimensional array of characters terminated by a **nullcharacter** '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

```
char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

If you follow the rule of array initialization then you can write the above statement as follows:

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++

Index	0	1	2	3	4	5
Variable	H	e	l	l	o	\0
Address	0x23451	0x23452	0x23453	0x23454	0x23455	0x23456

Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string

```
#include <stdio.h>
int main ()
{
    char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
    printf("Greeting message: %s\n", greeting );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings

Function & Purpose	
1	strcpy(s1, s2); Copies string s2 into string s1.
2	strcat(s1, s2); Concatenates string s2 onto the end of string s1.
3	strlen(s1); Returns the length of string s1.
4	strcmp(s1, s2); Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	strchr(s1, ch); Returns a pointer to the first occurrence of character ch in string s1.
6	strstr(s1, s2); Returns a pointer to the first occurrence of string s2 in string s1.

The following example uses some of the above-mentioned functions:

```
#include <stdio.h>
#include <string.h>
int main ()
{
    char str1[12] = "Hello";
    char str2[12] = "World";    char str3[12];
    int len ;      /* copy str1 into str3 */
    strcpy(str3, str1);
    /* concatenates str1 and str2 */
    printf("strcpy( str3, str1) : %s\n", str3 );
    strcat( str1, str2);
    /* total length of str1 after concatenation */
    printf("strcat( str1, str2): %s\n", str1 );
    len = strlen(str1);
    printf("strlen(str1) : %d\n", len );
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
strcpy( str3, str1) : Hello
strcat( str1, str2): HelloWorld
strlen(str1) : 10
```

Structures

Arrays allow to define type of variables that can hold several data items of the same kind. Similarly **structure** is another user defined data type available in C that allows to combine data items of different kinds.

Structures are used to represent a record. Suppose you want to keep track of your books in a library. You might want to track the following attributes about each book :

- Title
- Author
- Subject
- Book ID

Defining a Structure

To define a structure, you must use the **struct** statement. The struct statement defines a new data type, with more than one member. The format of the struct statement is as follows:

```
struct [structure tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more structure variables];
```

The **structure tag** is optional and each member definition is a normal variable definition, such as int i; or float f; or any other valid variable definition. At the end of the structure's definition, before the final semicolon, you can specify one or more structure variables but it is optional. Here is the way you would declare the Book structure:

```
struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} book;
```

Accessing Structure Members

To access any member of a structure, we use the **member access operator (.)**. The member access operator is coded as a period between the structure variable name and the structure member that we wish to access. You would use the keyword **struct** to define variables of structure type. The following example shows how to use a structure in a program

```
#include <stdio.h>
#include <string.h>
struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id; };

int main( ) {
    struct Books Book1;      /* Declare Book1 of type Book */
    struct Books Book2;      /* Declare Book2 of type Book */
    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;
    /* book 2 specification */
    strcpy( Book2.title, "Telecom Billing");
    strcpy( Book2.author, "Zara Ali");
    strcpy( Book2.subject, "Telecom Billing Tutorial");
    Book2.book_id = 6495700;   /* print Book1 info */
    printf( "Book 1 title : %s\n", Book1.title);
    printf( "Book 1 author : %s\n", Book1.author);
    printf( "Book 1 subject : %s\n", Book1.subject);
    printf( "Book 1 book_id : %d\n", Book1.book_id);
    /* print Book2 info */
    printf( "Book 2 title : %s\n", Book2.title);
    printf( "Book 2 author : %s\n", Book2.author);
    printf( "Book 2 subject : %s\n", Book2.subject);
    printf( "Book 2 book_id : %d\n", Book2.book_id);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book 1 title : C Programming Book 1
author : Nuha Ali Book 1
subject : C Programming Tutorial Book 1
book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : Zara Ali
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

C Library for MSX-DOS with SDCC compiler

Structures as Function Arguments

You can pass a structure as a function argument in the same way as you pass any other variable or pointer.

```
#include <stdio.h>
#include <string.h>
struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
};

/* function declaration */
void printBook( struct Books book );

int main( ) {
    struct Books Book1;          /* Declare Book1 of type Book */
    struct Books Book2;          /* Declare Book2 of type Book */
    /* book 1 specification */
    strcpy( Book1.title, "C Programming");
    strcpy( Book1.author, "Nuha Ali");
    strcpy( Book1.subject, "C Programming Tutorial");
    Book1.book_id = 6495407;

    /* print Book1 info */
    printBook( Book1 );

    return 0;
}
void printBook( struct Books book )
{
    printf( "Book title : %s\n", book.title);
    printf( "Book author : %s\n", book.author);
    printf( "Book subject : %s\n", book.subject);
    printf( "Book book_id : %d\n", book.book_id);
}
```

DO NOT
Sharing This Document
is copyright ©

Pointers to Structures

You can define pointers to structures in the same way as you define pointer to any other variable:

```
struct Books *struct_pointer;
```

Now, you can store the address of a structure variable in the above defined pointer variable. To find the address of a structure variable, place the '&'; operator before the structure's name as follows:

```
struct_pointer = &Book1;
```

To access the members of a structure using a pointer to that structure, you must use the → operator as follows:

```
struct_pointer->title;
```

If we take the previous example and modify it as:

```
(...)
printBook( &Book2 );
return 0;
}
void printBook( struct Books *book )
{
printf( "Book title : %s\n", book->title);
printf( "Book author : %s\n", book->author);
printf( "Book subject : %s\n", book->subject);
printf( "Book book_id : %d\n", book->book_id);
}
```

Bit Fields in structures

Bit Fields allow the packing of data in a structure. This is especially useful when memory or data storage is at a premium. Typical examples include:

- Packing several objects into a machine word, e.g. 1 bit flags can be compacted.
- Reading external file formats -- non-standard file formats could be read in, e.g., 9-bit integers.

C allows us to do this in a structure definition by putting :bit length after the variable. For example:

```
struct packed_struct {
    unsigned int f1:1;
    unsigned int f2:1;
    unsigned int f3:1;
    unsigned int f4:1;
    unsigned int type:4;
    unsigned int my_int:9;
} pack;
```

Here, the packed_struct contains 6 members: Four 1 bit flags f1..f3, a 4-bit type and a 9-bit my_int.

C automatically packs the above bit fields as compactly as possible, provided that the maximum length of the field is less than or equal to the integer word length of the computer. If this is not the case, then some compilers may allow memory overlap for the fields while others would store the next field in the next word.

Union

A **union** is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple-purpose.

Defining a Union

To define a union, you must use the **union** statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows:

```
union [union tag]
{
    member definition;
    member definition;
    ...
    member definition;
} [one or more union variables];
```

The **union tag** is optional and each member definition is a normal variable definition, such as `int i;` or `float f;` or any other valid variable definition. At the end of the union's definition, before the final semicolon, you can specify one or more union variables but it is optional. Here is the way you would define a union type named `Data` having three members `i`, `f`, and `str`

```
union Data {    int i;    float f;    char str[20]; } data;
```

Now, a variable of `Data` type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, `Data` type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union:

```
#include <stdio.h>
#include <string.h>
union Data
{
    int i;
    float f;
    char str[20];
};
int main( ) {
    union Data data;
    printf( "Mem size occupied by data : %d\n", sizeof(data));
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Memory size occupied by data : 20
```

Accessing Union Members

To access any member of a union, we use the **member access operator (.)**. The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword **union** to define variables of union type. The following example shows how to use unions in a program

```
#include <stdio.h>
#include <string.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main( )
{
    union Data data;
    data.i = 10;
    data.f = 220.5;
    strcpy( data.str, "C Programming");
    printf( "data.i : %d\n", data.i);
    printf( "data.f : %f\n", data.f);
    printf( "data.str : %s\n", data.str);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 1917853763
data.f : 4122360580327794860452759994368.000000
data.str : C Programming
```

Here, we can see that the values of **i** and **f** members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of **str** member is getting printed very well. Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions:

```
#include <stdio.h>
#include <string.h>
union Data {
    int i;
    float f;
    char str[20];
};
int main( )
{
    union Data data;
    data.i = 10;
    printf( "data.i : %d\n", data.i);
    data.f = 220.5;
    printf( "data.f : %f\n", data.f);
    strcpy( data.str, "C Programming");
    printf( "data.str : %s\n", data.str);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
data.i : 10
data.f : 220.500000
data.str : C Programming
```

Here, all the members are getting printed very well because one member is being used at a time.

Bit Field

Suppose your C program contains a number of TRUE/FALSE variables grouped in a structure called status, as follows:

```
struct
{
    unsigned int widthValidated;
    unsigned int heightValidated;
} status;
```

This structure requires 8 bytes of memory space but in actual, we are going to store either 0 or 1 in each of the variables. The C programming language offers a better way to utilize the memory space in such situations.

If you are using such variables inside a structure then you can define the width of a variable which tells the C compiler that you are going to use only those number of bytes. For example, the above structure can be re-written as follows

```
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status;
```

The above structure requires 4 bytes of memory space for status variable, but only 2 bits will be used to store the values.

If you will use up to 32 variables each one with a width of 1 bit, then also the status structure will use 4 bytes. However as soon as you have 33 variables, it will allocate the next slot of the memory and it will start using 8 bytes. Let us check the following example to understand the concept

```
#include <stdio.h>
#include <string.h>
/* define simple structure */
struct {
    unsigned int widthValidated;
    unsigned int heightValidated;
} status1;
/* define a structure with bit fields */
struct {
    unsigned int widthValidated : 1;
    unsigned int heightValidated : 1;
} status2;
int main( )
{
    printf( "Memory size occupied by status1 : %d\n",
    sizeof(status1));
    printf( "Memory size occupied by status2 : %d\n",
    sizeof(status2));
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Memory size occupied by status1 : 8
Memory size occupied by status2 : 4
```

Bit Field Declaration

The declaration of a bit-field has the following form inside a structure:

```
struct { type [member_name] : width ; };
```

The following table describes the variable elements of a bit field :

Element & Description	
1	type An integer type that determines how a bit-field's value is interpreted. The type may be int, signed int, or unsigned int.
2	member_name The name of the bit-field.
3	width The number of bits in the bit-field. The width must be less than or equal to the bit width of the specified type.

The variables defined with a predefined width are called **bit fields**. A bit field can hold more than a single bit; for example, if you need a variable to store a value from 0 to 7, then you can define a bit field with a width of 3 bits as follows:

```
struct { unsigned int age : 3; } Age;
```

The above structure definition instructs the C compiler that the age variable is going to use only 3 bits to store the value. If you try to use more than 3 bits, then it will not allow you to do so. Let us try the following example:

```
#include <stdio.h>
#include <string.h>
struct {
    unsigned int age : 3;
} Age;

int main( )
{
    Age.age = 4;
    printf( "Sizeof( Age ) : %d\n", sizeof(Age) );
    printf( "Age.age : %d\n", Age.age );
    Age.age = 7;
    printf( "Age.age : %d\n", Age.age );
    Age.age = 8;
    printf( "Age.age : %d\n", Age.age );
    return 0;
}
```

When the above code is compiled it will compile with a warning and when executed, it produces the following result:

```
Sizeof( Age ) : 4
Age.age : 4
Age.age : 7
Age.age : 0
```

Typedef

The C programming language provides a keyword called **typedef**, which you can use to give a type a new name. Following is an example to define a term **BYTE** for one-byte numbers

```
typedef char BYTE;
```

After this type definition, the identifier **BYTE** can be used as an abbreviation for the type **unsigned char**, **for example.**

```
BYTE b1, b2;
```

By convention, upper case letters are used for these definitions to remind the user that the type name is really a symbolic abbreviation, but you can use lower case, as follows:

```
typedef char byte;
```

You can use **typedef** to give a name to your user defined data types as well. For example, you can use **typedef** with structure to define a new data type and then use that data type to define structure variables directly as follows:

```
#include <stdio.h>
#include <string.h>
typedef struct Books {
    char title[50];
    char author[50];
    char subject[100];
    int book_id;
} Book;

int main( )
{
Book book;
strcpy( book.title, "C Programming");
strcpy( book.author, "Nuha Ali");
strcpy( book.subject, "C Programming Tutorial");
book.book_id = 6495407;
printf( "Book title : %s\n", book.title);
printf( "Book author : %s\n", book.author);
printf( "Book subject : %s\n", book.subject);
printf( "Book book_id : %d\n", book.book_id);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Book title : C Programming Book
author : Nuha Ali Book
subject : C Programming Tutorial Book
book_id : 6495407
```

typedef or #define

#define is a C-directive which is also used to define the aliases for various data types similar to **typedef** but with the following differences :

- **typedef** is limited to giving symbolic names to types only whereas **#define** can be used to define alias for values as well, q., you can define 1 as ONE etc.
- **typedef** interpretation is performed by the compiler whereas **#define** statements are processed by the pre-processor.

The following example shows how to use **#define** in a program

```
#include <stdio.h>
#define TRUE 1
#define FALSE 0
int main( ) {
```

```
printf( "Value of TRUE : %d\n", TRUE);
printf( "Value of FALSE : %d\n", FALSE);
return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of TRUE : 1 Value of FALSE : 0
```

Input and Output

When we say **Input**, it means to feed some data into a program. An input can be given in the form of a file or from the command line. C programming provides a set of built-in functions to read the given input and feed it to the program as per requirement.

When we say **Output**, it means to display some data on screen, printer, or in any file. C programming provides a set of built-in functions to output the data on the computer screen as well as to save it in text or binary files.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

The Standard Files

C programming treats all the devices as files. So devices such as the display are addressed in the same way as files and the following three files are automatically opened when a program executes to provide access to the keyboard and screen.

Standard File	File Pointer	Device
Standard input	stdin	Keyboard
Standard output	stdout	Screen
Standard error	stderr	Your screen

The file pointers are the means to access the file for reading and writing purpose. This section explains how to read values from the screen and how to print the result on the screen.

The getchar() and putchar() Functions

The **int getchar(void)** function reads the next available character from the screen and returns it as an integer. This function reads only single character at a time. You can use this method in the loop in case you want to read more than one character from the screen.

The **int putchar(int c)** function puts the given character on the screen and returns the same character. This function puts only single character at a time. You can use this method in the loop in case you want to display more than one character on the screen. Check this example:

```
#include <stdio.h>
int main( ) {
    int c;
    printf( "Enter a value :");
    c = getchar( );
    printf( "\nYou entered: ");
    putchar( c );
    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads only a single character and displays it as follows:

```
> Enter a value : this is test You entered: t
```

C Library for MSX-DOS with SDCC compiler

The gets() and puts() Functions

The **char *gets(char *s)** function reads a line from **stdin** into the buffer pointed to by **s** until either a terminating newline or EOF (End of File).

The **int puts(const char *s)** function writes the string '**s**' and '**a**' trailing newline to **stdout**.

```
#include <stdio.h>
int main( ) {
    char str[100];
    printf( "Enter a value :");
    gets( str );
    printf( "\nYou entered: ");
    puts( str );
    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then the program proceeds and reads the complete line till end, and displays it as follows:

```
> Enter a value : this is test You entered: this is test
```

The scanf() and printf() Functions

The **int scanf(const char *format, ...)** function reads the input from the standard input stream **stdin** and scans that input according to the **format** provided.

The **int printf(const char *format, ...)** function writes the output to the standard output stream **stdout** and produces the output according to the format provided.

The **format** can be a simple constant string, but you can specify **%s**, **%d**, **%c**, **%f**, etc., to print or read strings, integer, character or float respectively. There are many other formatting options available which can be used based on requirements. Let us now proceed with a simple example to understand the concepts

```
#include <stdio.h>
int main( ) {
    char str[100];
    int i;
    printf( "Enter a value :");
    scanf("%s %d", str, &i);
    printf( "\nYou entered: %s %d ", str, i);
    return 0;
}
```

When the above code is compiled and executed, it waits for you to input some text. When you enter a text and press enter, then program proceeds and reads the input and displays it:

```
> Enter a value : seven 7 You entered: seven 7
```

Here, it should be noted that **scanf()** expects input in the same format as you provided **%s** and **%d**, which means you have to provide valid inputs like "string integer". If you provide "string string" or "integer integer", then it will be assumed as wrong input. Secondly, while reading a string, **scanf()** stops reading as soon as it encounters a space, so "this is test" are three strings for **scanf()**.

The Preprocessor

The **C Preprocessor** is not a part of the compiler, but is a separate step in the compilation process. In simple terms, a C Preprocessor is just a text substitution tool and it instructs the compiler to do required pre-processing before the actual compilation. We'll refer to the C Preprocessor as CPP.

All preprocessor commands begin with a hash symbol (#). It must be the first nonblank character, and for readability, a preprocessor directive should begin in the first column. The following section lists down all the important preprocessor directives:

Directive & Description	
1	#define Substitutes a preprocessor macro.
2	#include Inserts a particular header from another file.
3	#undef Undefines a preprocessor macro.
4	#ifdef Returns true if this macro is defined.
5	#ifndef Returns true if this macro is not defined.
6	#if Tests if a compile time condition is true.
7	#else The alternative for #if.
8	#elif #else and #if in one statement.
9	#endif Ends preprocessor conditional.
10	#error Prints error message on stderr.
11	#pragma Issues special commands to the compiler, using a standardized method.

Preprocessors Examples

Analyze the following examples to understand various directives.

```
#define MAX_ARRAY_LENGTH 20
```

This directive tells the preprocessor to replace instances of MAX_ARRAY_LENGTH with 20. Use `#define` for constants to increase readability.

```
#include <stdio.h>
#include "myheader.h"
```

These directives tell the preprocessor to get stdio.h from **System Libraries** and add the text to the current source file. The next line tells CPP to get **myheader.h** from the local directory and add the content to the current source file.

```
#undef FILE_SIZE
#define FILE_SIZE 42
```

It tells the preprocessor to undefine existing FILE_SIZE and define it as 42.

```
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif
```

It tells the CPP to define MESSAGE only if MESSAGE isn't already defined.

```
#ifdef DEBUG /* Your debugging statements here */
#endif
```

It tells the preprocessor to process the statements enclosed if DEBUG is defined. This is useful if you pass the `-DDEBUG` flag to the gcc compiler at the time of compilation. This will define DEBUG, so you can turn debugging on and off on the fly during compilation.

Predefined Macros

ANSI C defines a number of macros. Although each one is available for use in programming, the predefined macros should not be directly modified.

Macro & Description	
1	<u>__DATE__</u> The current date as a character literal in "MMM DD YYYY" format.
2	<u>__TIME__</u> The current time as a character literal in "HH:MM:SS" format.
3	<u>__FILE__</u> This contains the current filename as a string literal.
4	<u>__LINE__</u> This contains the current line number as a decimal constant.
5	<u>__STDC__</u> Defined as 1 when the compiler complies with the ANSI standard.

Try the following example:

```
#include <stdio.h>
int main() {
    printf("File :%s\n", __FILE__);
    printf("Date :%s\n", __DATE__);
    printf("Time :%s\n", __TIME__);
    printf("Line :%d\n", __LINE__);
    printf("ANSI :%d\n", __STDC__);
}
```

When the above code in a file **test.c** is compiled and executed, it produces the following result:

```
File :test.c
Date :Jan 8 2018
Time :09:16:29
Line :8
ANSI :1
```

Preprocessor Operators

The C preprocessor offers the following operators to help create macros

The Macro Continuation (\) Operator

A macro is normally confined to a single line. The macro continuation operator () is used to continue a macro that is too long for a single line. For example:

```
#define message_for(a, b) \ printf(#a " and " #b ": We love you!\n")
```

The Stringize (#) Operator

The stringize or number-sign operator ('#), when used within a macro definition, converts a macro parameter into a string constant. This operator may be used only in a macro having a specified argument or parameter list.

```
#include <stdio.h>
#define message_for(a, b) \
    printf(#a " and " #b ": We love you!\n")

int main(void) {
    message_for(Carole, Debra);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Carole and Debra: We love you!
```

The Token Pasting (##) Operator

The token-pasting operator (##) within a macro definition combines two arguments. It permits two separate tokens in the macro definition to be joined into a single token. For example:

```
#include <stdio.h>
#define tokenpaster(n) printf ("token" #n " = %d", token##n)  int main(void) {
    int token34 = 40;
    tokenpaster(34);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
token34 = 40
```

It happened so because this example results in the following actual output from the preprocessor

```
printf ("token34 = %d", token34);
```

This example shows the concatenation of token##n into token34 and here we have used both **stringize** and **token-pasting**.

The Defined() Operator

The preprocessor defined operator is used in constant expressions to determine if an identifier is defined using #define. If the specified identifier is defined, the value is true (non-zero). If the symbol is not defined, the value is false (zero). The defined operator is specified as follows:

```
#include <stdio.h>
#ifndef MESSAGE
#define MESSAGE "You wish!"
#endif

int main(void) {
    printf("Here is the message: %s\n", MESSAGE);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Here is the message: You wish!
```

Parameterized Macros

One of the powerful functions of the CPP is the ability to simulate functions using parameterized macros. For example, we might have some code to square a number as follows :

```
int square(int x) {
    return x * x;
}
```

We can rewrite above the code using a macro as follows :

```
#define square(x) ((x) * (x))
```

Macros with arguments must be defined using the #define directive before they can be used. The argument list is enclosed in parentheses and must immediately follow the macro name. Spaces are not allowed between the macro name and open parenthesis. For example :

```
#include <stdio.h>
#define MAX(x,y) ((x) > (y) ? (x) : (y))

int main(void) {
    printf("Max between 20 and 10 is %d\n", MAX(10, 20));
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Max between 20 and 10 is 20
```

The Header file

A header file is a file with extension **.h** which contains C function declarations and macro definitions to be shared between several source files. There are two types of header files: the files that the programmer writes and the files that comes with your compiler or library.

You request to use a header file in your program by including it with the C preprocessing directive **#include**, like you have seen inclusion of **stdio.h** header file, which comes along with your compiler.

Including a header file is equal to copying the content of the header file but we do not do it because it will be error-prone and it is not a good idea to copy the content of a header file in the source files, especially if we have multiple source files in a program.

A simple practice in C or C++ programs is that we keep all the constants, macros, system wide global variables, and function prototypes in the header files and include that header file wherever it is required.

Include Syntax

Both the user and the system header files are included using the preprocessing directive **#include**. It has the following two forms

```
#include <file>
```

This form is used for system header files. It searches for a file named 'file' in a standard list of system directories. You can prepend directories to this list with the -I option while compiling your source code.

```
#include "file"
```

This form is used for header files of your own program. It searches for a file named 'file' in the directory containing the current file. You can prepend directories to this list with the -I option while compiling your source code.

Include Operation

The **#include** directive works by directing the C preprocessor to scan the specified file as input before continuing with the rest of the current source file. The output from the preprocessor contains the output already generated, followed by the output resulting from the included file, followed by the output that comes from the text after the **#include** directive. For example, if you have a header file header.h as follows:

```
char *test (void);
```

and a main program called *program.c* that uses the header file, like this

```
int x;
#include "header.h"
int main (void)
{
    puts (test ());
}
```

The compiler will see the same token stream as it would if *program.c* read.

```
int x;
char *test (void);

int main (void) {
    puts (test ());
}
```

Once-Only Headers

If a header file happens to be included twice, the compiler will process its contents twice and it will result in an error. The standard way to prevent this is to enclose the entire real contents of the file in a conditional, like this:

```
#ifndef HEADER_FILE
#define HEADER_FILE
the entire header file here
#endif
```

This construct is commonly known as a wrapper **#ifndef**. When the header is included again, the conditional will be false, because HEADER_FILE is defined. The preprocessor will skip over the entire contents of the file, and the compiler will not see it twice.

Computed Includes

Sometimes it is necessary to select one of the several different header files to be included into your program. For instance, they might specify configuration parameters to be used on different sorts of operating systems. You could do this with a series of conditionals as follows:

```
#if SYSTEM_1
# include "system_1.h"
#elif SYSTEM_2
# include "system_2.h"
#elif SYSTEM_3
...
#endif
```

But as it grows, it becomes tedious, instead the preprocessor offers the ability to use a macro for the header name. This is called a **computed include**. Instead of writing a header name as the direct argument of **#include**, you simply put a macro name there:

```
#define SYSTEM_H "system_1.h"
...
#include SYSTEM_H
```

SYSTEM_H will be expanded, and the preprocessor will look for system_1.h as if the **#include** had been written that way originally. SYSTEM_H could be defined by your Makefile with a -D option.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

Type Casting

Type casting is a way to convert a variable from one data type to another data type. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the **cast operator** as follows:

```
(type_name) expression
```

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation

```
#include <stdio.h>

main()
{
    int sum = 17, count = 5;
    double mean;
    mean = (double) sum / count;
    printf("Value of mean : %f\n", mean );
}
```

When the above code is compiled and executed, it produces the following result

```
Value of mean : 3.400000
```

It should be noted here that the cast operator has precedence over division, so the value of **sum** is first converted to type **double** and finally it gets divided by **count** yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the **cast operator**. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

Integer Promotion

Integer promotion is the process by which values of integer type "smaller" than **int** or **unsigned int** are converted either to **int** or **unsigned int**. Consider an example of adding a character with an integer:

```
#include <stdio.h>
main()
{
    int i = 17;
    char c = 'c'; /* ascii value is 99 */
    int sum;
    sum = i + c;
    printf("Value of sum : %d\n", sum );
}
```

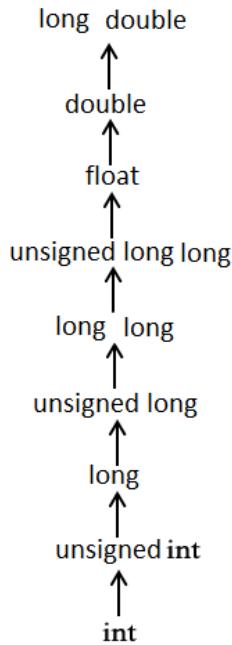
When the above code is compiled and executed, it produces the following result

```
Value of sum : 116
```

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

Usual Arithmetic Conversion

The **usual arithmetic conversions** are implicitly performed to cast their values to a common type. The compiler first performs *integer promotion*; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy



The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators `&&` and `||`. Let us take the following example to understand the concept

```
#include <stdio.h>
main()
{
    int i = 17;
    char c = 'c'; /* ascii value is 99 */
    float sum;
    sum = i + c;
    printf("Value of sum : %f\n", sum );
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of sum : 116.000000
```

Here, it is simple to understand that first `c` gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts `i` and `c` into 'float' and adds them yielding a 'float' result.

Error Handling

As such, C programming does not provide direct support for error handling but being a system programming language, it provides you access at lower level in the form of return values. Most of the C or even Unix function calls return -1 or NULL in case of any error and set an error code `errno`. It is set as a global variable and indicates an error occurred during any function call. You can find various error codes defined in `<error.h>` header file.

So a C programmer can check the returned values and can take appropriate action depending on the return value. It is a good practice, to set `errno` to 0 at the time of initializing a program. A value of 0 indicates that there is no error in the program.

errno, perror(). and strerror()

The C programming language provides **perror()** and **strerror()** functions which can be used to display the text message associated with **errno**.

- The **perror()** function displays the string you pass to it, followed by a colon, a space, and then the textual representation of the current **errno** value.
- The **strerror()** function, which returns a pointer to the textual representation of the current **errno** value.

Let's try to simulate an error condition and try to open a file which does not exist. Here I'm using both the functions to show the usage, but you can use one or more ways of printing your errors. Second important point to note is that you should use **stderr** file stream to output all the errors.

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
extern int errno ;
int main () {
    FILE * pf;
    int errnum;
    pf = fopen ("unexist.txt", "rb");
    if (pf == NULL) {
        errnum = errno;
        fprintf(stderr, "Value of errno: %d\n", errno);
        perror("Error printed by perror");
        fprintf(stderr, "Error opening file: %s\n", strerror( errnum ));
    }
    else
    {
        fclose (pf);
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Value of errno: 2
Error printed by perror: No such file or directory
Error opening file: No such file or directory
```

Divide by Zero Errors

It is a common problem that at the time of dividing any number, programmers do not check if a divisor is zero and finally it creates a runtime error. The code below fixes this by checking if the divisor is zero before dividing

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int dividend = 20;
    int divisor = 0;
    int quotient;
    if( divisor == 0){
        fprintf(stderr, "Division by zero! Exiting...\n");
        exit(-1);
    }
    quotient = dividend / divisor;
    fprintf(stderr, "Value of quotient : %d\n", quotient );
    exit(0);
}
```

When the above code is compiled and executed, it produces the following result:

```
Division by zero! Exiting...
```

Program Exit Status

It is a common practice to exit with a value of EXIT_SUCCESS in case of program coming out after a successful operation. Here, EXIT_SUCCESS is a macro and it is defined as 0.

If you have an error condition in your program and you are coming out then you should exit with a status EXIT_FAILURE which is defined as -1. So let's write above program as follows:

```
#include <stdio.h>
#include <stdlib.h>
main() {
    int dividend = 20;
    int divisor = 5;
    int quotient;

    if( divisor == 0) {
        fprintf(stderr, "Division by zero! Exiting...\n");
        exit(EXIT_FAILURE);
    }
    quotient = dividend / divisor;
    fprintf(stderr, "Value of quotient : %d\n", quotient );
    exit(EXIT_SUCCESS);
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of quotient : 4
```

Recursion

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion() {
    recursion(); /* function calls itself */
}

int main()
{
    recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

Recursive functions are very useful to solve many mathematical problems, such as calculating the factorial of a number, generating Fibonacci series, etc.

Number Factorial

The following example calculates the factorial of a given number using a recursive function

```
#include <stdio.h>
unsigned long long
int factorial(unsigned int i)
{
    if(i <= 1)
    {
        return 1;
    }
    return i * factorial(i - 1);
}

int main()
{
    int i = 12;
    printf("Factorial of %d is %d\n", i, factorial(i));
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

```
Factorial of 12 is 479001600
```

Fibonacci Series

The following example generates the Fibonacci series for a given number using a recursive function

```
#include <stdio.h>
int fibonacci(int i) {
    if(i == 0)
    {
        return 0;
    }
    if(i == 1)
    {
        return 1;
    }
    return fibonacci(i-1) + fibonacci(i-2);
}
int main() {
    int i;
    for (i = 0; i < 10; i++)
    {
        printf("%d\t\n", fibonacci(i));
    }
    return 0;
}
```

When the above code is compiled and executed, it produces the following result

0	1	1	2	3	5	8	13	21	34
---	---	---	---	---	---	---	----	----	----

Variable Arguments

Sometimes, you may come across a situation, when you want to have a function, which can take variable number of arguments, i.e., parameters, instead of predefined number of parameters. The C programming language provides a solution for this situation and you are allowed to define a function which can accept variable number of parameters based on your requirement. The following example shows the definition of such a function.

```
int func(int, ... )
{
...
}
int main() {
    func(1, 2, 3);
    func(1, 2, 3, 4);
}
```

It should be noted that the function **func()** has its last argument as ellipses, i.e. three dots (...) and the one just before the ellipses is always an **int** which will represent the total number of given variable arguments. To use such functionality, you need to make use of **stdarg.h** header file which provides the functions and macros to implement the functionality of variable arguments and follow the given steps :

- Define a function with its last parameter as ellipses and the one just before the ellipses is always an **int** which will represent the number of arguments.
- Create a **va_list** type variable in the function definition. This type is defined in stdarg.h header file.
- Use **int** parameter and **va_start** macro to initialize the **va_list** variable to an argument list. The macro **va_start** is defined in stdarg.h header file.
- Use **va_arg** macro and **va_list** variable to access each item in argument list.
- Use a macro **va_end** to clean up the memory assigned to **va_list** variable.

Now let us follow the above steps and write down a simple function which can take the variable number of parameters and return their average

```
#include <stdio.h>
#include <stdarg.h>
double average(int num,...) {
    va_list valist;
    double sum = 0.0;
    int i;
    /* initialize valist for num number of arguments */
    va_start(valist, num);
    /* access all the arguments assigned to valist */
    for (i = 0; i < num; i++)
    {
        sum += va_arg(valist, int);
    }
    /* clean memory reserved for valist */
    va_end(valist);
    return sum/num;
}
int main() {
    printf("Average of 2, 3, 4, 5 = %f\n", average(4, 2,3,4,5));
    printf("Average of 5, 10, 15 = %f\n", average(3, 5,10,15));
}
```

When the above code is compiled and executed, it produces the following result. It should be noted that the function **average()** has been called twice and each time the first argument represents the total number of variable arguments being given. Only ellipses will be used to pass variable number of arguments.

Average of 2, 3, 4, 5 = 3.500000 Average of 5, 10, 15 = 10.000000

Memory Management

Note: Memory management is quite different when using small device systems like MSX. There is no a lot of memory. Using the MMalloc function include in the library is limited by the SDCC compiler to 1024KB. If you need more dynamic memory allocation, you can make some change inside the SDCC standard library. Please check SDCC manual.

This chapter explains dynamic memory management in C. The C programming language provides several functions for memory allocation and management. These functions can be found in the <stdlib.h> header file.

Function & Description	
1	void *calloc(int num, int size); This function allocates an array of num elements each of which size in bytes will be size .
2	void free(void *address); This function releases a block of memory block specified by address.
3	void *malloc(int num); This function allocates an array of num bytes and leave them uninitialized.
4	void *realloc(void *address, int newsize); This function re-allocates memory extending it upto newsize .

C Library for MSX-DOS with SDCC compiler

Allocating Memory Dynamically

While programming, if you are aware of the size of an array, then it is easy and you can define it as an array. For example, to store a name of any person, it can go up to a maximum of 100 characters, so you can define something as follows:

```
char name[100];
```

But now let us consider a situation where you have no idea about the length of the text you need to store, for example, you want to store a detailed description about a topic. Here we need to define a pointer to character without defining how much memory is required and later, based on requirement, we can allocate memory as shown in the below example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char name[100];
    char *description;
    strcpy(name, "Zara Ali");
    /* allocate memory dynamically */
    description = malloc( 200 * sizeof(char) );
    if( description == NULL ) {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    } else {
        strcpy( description, "Zara ali a DPS student in class 10th");
    }
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
}
```

When the above code is compiled and executed, it produces the following result.

```
Name = Zara Ali Description: Zara ali a DPS student in class 10th
```

Same program can be written using **calloc()**; only thing is you need to replace malloc with calloc as follows:

```
calloc(200, sizeof(char));
```

So you have complete control and you can pass any size value while allocating memory, unlike arrays where once the size defined, you cannot change it.

C Library for MSX-DOS with SDCC compiler

Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated by your program but as a good practice when you are not in need of memory anymore then you should release that memory by calling the function **free()**.

Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**. Let us check the above program once again and make use of realloc() and free() functions:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
    char name[100];
    char *description;
    strcpy(name, "Zara Ali");
    /* allocate memory dynamically */
    description = malloc( 30 * sizeof(char) );
    if( description == NULL ) {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    } else {
        strcpy( description, "Zara ali a DPS student." );
    }
    /* suppose you want to store bigger description */
    description = realloc( description, 100 * sizeof(char) );
    if( description == NULL ) {
        fprintf(stderr, "Error - unable to allocate required memory\n");
    } else {
        strcat( description, "She is in class 10th" );
    }
    printf("Name = %s\n", name );
    printf("Description: %s\n", description );
    /* release memory using free() function */
    free(description);
}
```

When the above code is compiled and executed, it produces the following result.

```
Name = Zara Ali Description: Zara ali a DPS student.She is in class 10th
```

You can try the above example without re-allocating extra memory, and strcat() function will give an error due to lack of available memory in description.

PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation

C Library for MSX-DOS with SDCC compiler
Publish and distribute your game

You just finished a game, and you're very proud of it? Do you want to distribute it or sell it around the world?

You can trust REPROFACTORY to distribute your game, and receive your royalties on sales.



With more than 550 customers on 5 continents, you will be assured that your game will be played by dozens of MSX users.

REPROFACTORY can manufacture the cartridges, and offer you to realize a packaging, and deal with sales; or just one of these steps if you wish.

Example of games sold on www.Repro-Factory.com:



Muffie's Tutankham &
Conversion Classic...

30,00 €

In Stock



Joust - Arcade

38,00 €

In Stock



Gyruss - Arcade

38,00 €

In Stock



Bomb Jack
MSX2 Cartridge



TRANSBALL
MSX Cartridge



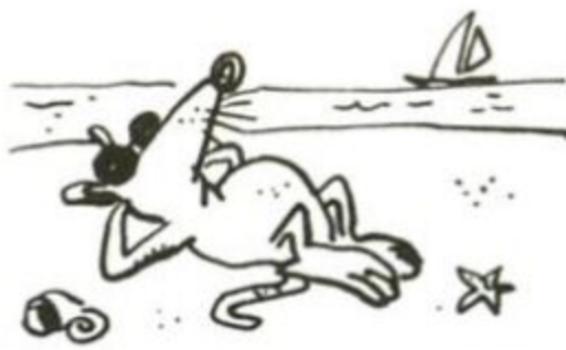
ARCOMAGE
MSX CARTRIDGE

Contacts

I hope you will enjoy the FUSION-C Library, and you will use it to create new games and tools for the MSX computers.

If you have any questions about FUSION-C please contact me at:

erichb59@ebsoft.fr



PRIVATE DOCUMENT
DO NOT SHARE THIS DOCUMENT
Sharing This Document without authorization
is copyright violation