

MEGA 65

USER'S GUIDE



MEGA 65

MUSEUM OF ELECTRONIC GAMES & ART

REGULATORY INFORMATION

The MEGA65 home computer and portable computer have not been subject to FCC, EC or other regulatory approvals as of the time of writing.

MEGA65 REFERENCE GUIDE

Published by
the MEGA Museum of Electronic Games and Art, Germany.
and
Flinders University, Australia.

WORK IN PROGRESS

Copyright ©2019 - 2020 by Paul Gardner-Stephen, Flinders University, the Museum of Electronic Games and Art eV., and contributors.

This reference guide is made available under the GNU Free Documentation License v1.3, or later, if desired. This means that you are free to modify, reproduce and redistribute this user guide, subject to certain conditions. The full text of the GNU Free Documentation License v1.3 can be found at <https://www.gnu.org/licenses/fdl-1.3.en.html>.

Implicit in this copyright license, is the permission to duplicate and/or redistribute this document in whole or in part for use in education environments. We want to support the education of future generations, so if you have any worries or concerns, please contact us.

March 31, 2020

Contents

1	Introduction	v
I	GETTING TO KNOW YOUR MEGA65	1
2	SETUP	3
	Unpacking and connecting the MEGA65	5
	Rear Connections	6
	Side Connections	7
	Installation	8
	Connecting your MEGA65 to a screen and peripherals	8
	Optional Connections	9
	Operation	9
	Using the MEGA65	9
	THE CUSROR	10
3	GETTING STARTED	11
	Keyboard	13
	Command Keys	13
	RETURN	13
	SHIFT	13

SHIFT LOCK	14
CTRL	14
RUN/STOP	14
RESTORE	14
THE CURSOR KEYS	15
INSerT/DElete	15
CLeaR/HOME	15
MEGA KEY	15
NO SCROLL	16
Function Keys	16
HELP	16
ALT	16
CAPS LOCK	16
The Screen Editor	16
Editor Functionality	19
4 Cores and Flashing	21
What are cores, and why does it matter?	23
Selecting a core	23
Installing an upgrade core for the MEGA65	24
Installing other core for the MEGA65	25
Creating cores for the MEGA65	25
Replacing the factory core in slot 0	25
II FIRST STEPS IN CODING	27
5 How Computers Work	29
Computers are stupid. Really stupid	31
Making an Egg Cup Computer	31

6 C64, C65 and MEGA65 Modes	33
Switching Modes from BASIC	35
From C65 to C64 Mode	35
From C64 to C65 Mode	35
Entering Machine Code Monitor Mode	36
The KEY Register	36
Un-hiding C65 Extra Registers	36
Re-hiding C65/MEGA65 Extra Registers	37
Un-hiding MEGA65 Extra Registers	38
Traps to watch out for	39
Accessing the MEGA65's Extra Memory from BASIC 10 in C65 Mode	39
The MAP Instruction	40
III SOUND AND GRAPHICS	41
IV HARDWARE	43
7 Using a Nexys4DDR as a MEGA65	45
Building your own MEGA65 Compatible Computer	47
Power, Jumpers, Switches and Buttons	48
Keyboard	48
Preparing microSDHC card	49
Useful Tips	50
V APPENDICES	53
A ACCESSORIES	57

B BASIC 10 Command Reference

59

Format of Commands, Functions and Operators	61
Commands, Functions and Operators	62
ABS	64
AND	65
APPEND	66
ASC	67
ATN	68
AUTO	69
BACKGROUND	70
BACKUP	71
BANK	72
BEGIN	73
BEND	74
BLOAD	75
BOOT	76
BORDER	77
BOX	78
BSAVE	79
BUMP	80
BVERIFY	81
CATALOG	82
CHANGE	83
CHAR	84
CHR\$	85
CIRCLE	86
CLOSE	87
CLR	88

CMD	89
COLLECT	90
COLLISION	91
COLOR	92
CONCAT	93
CONT	94
COPY	95
COS	96
CURSOR	97
DATA	98
DCLEAR	99
DCLOSE	100
DEC	101
DEF FN	102
DELETE	103
DIM	104
DIR	105
DISK	106
DLOAD	107
DMA	108
DMODE	109
DO	110
DOPEN	111
DPAT	113
DSAVE	114
DVERIFY	115
EL	116
ELLIPSE	117

ELSE	118
END	119
ENVELOPE	120
ERASE	121
ER	122
ERR\$	123
EXIT	124
EXP	125
FAST	126
FILTER	127
FIND	128
FN	129
FOR	130
FOREGROUND	131
FRE	132
GET	133
GET#	134
GETKEY	135
GO64	136
GOSUB	137
GOTO	138
GRAPHIC	139
HEADER	140
HELP	141
HEX\$	142
HIGHLIGHT	143
IF	144
INPUT	145

INPUT#	146
INSTR	147
INT	148
JOY	149
KEY	150
LEFT\$	151
LEN	152
LET	153
LINE	154
LIST	155
LOAD	156
LOCATE	157
LOG	158
LOOP	159
LPEN	160
MID\$	161
MOD	162
MONITOR	163
MOUSE	164
MOVSPR	165
NEW	166
NEXT	167
NOT	168
OFF	169
ON	170
OPEN	171
OR	172
PAINT	173

PALETTE	174
PEEK	175
PEN	176
PLAY	177
POINTER	179
POKE	180
POLYGON	181
POS	182
POT	183
PRINT	184
PRINT#	185
PRINT USING	186
PUDEF	187
RCLR	188
RDOT	189
READ	190
RECORD	191
REM	192
RENAME	193
RENUMBER	194
RESTORE	195
RESUME	196
RETURN	197
RGR	198
RIGHT\$	199
RMOUSE	200
RND	201
RREG	202

RSPCOLOR	203
RSPPOS	204
RSPRITE	205
RUN	206
SAVE	207
SCNCLR	208
SCRATCH	209
SCREEN	210
SET	211
SGN	212
SIN	213
SLEEP	214
SLOW	215
SOUND	216
SPC	217
SPRCOLOR	218
SPRITE	219
SPRSAV	220
SQR	221
STEP	222
STOP	223
STR\$	224
SYS	225
TAB	226
TAN	227
TEMPO	228
THEN	229
TO	230

TRAP	231
TROFF	232
TRON	233
TYPE	234
UNTIL	235
USING	236
USR	237
VAL	238
VERIFY	239
VOL	240
WAIT	241
WHILE	242
WINDOW	243
XOR	244
C Special Keyboard Controls and Sequences	245
ASCII Codes and CHR\$	247
Control codes	250
Shifted codes	252
Escape Sequences	253
D Decimal, Binary and Hexadecimal	257
Numbers	259
Notations and Bases	260
Decimal	262
Binary	264
Hexadecimal	266
Operations	268
Counting	268

Arithmetic	270
Logic Gates	272
Signed and Unsigned Numbers	274
Bitwise Logical Operators	274
Converting Numbers	277
E 45GS02 Microprocessor	283
Introduction	285
Differences to the 6502	285
Supervisor/Hypervisor Privileged Mode	285
6502 Illegal Opcodes	286
Read-Modify-Write Instruction Bug Compatibility	286
Variable CPU Speed	287
Slow (1MHz – 3.5MHz) Operation	287
Full Speed (40MHz) Instruction Timing	288
CPU Speed Fine-Tuning	288
Direct Memory Access (DMA)	288
Accessing memory between the 64KB and 1MB points	289
C64-Style Memory Banking	289
VIC-III “ROM” Banking	289
VIC-III Display Address Translator	290
The MAP instruction	290
Direct Memory Access (DMA) Controller	291
Flat Memory Access	291
Accessing memory beyond the 1MB point	291
Using the MAP instruction to access >1MB	291
Flat-Memory Access	293
Virtual 32-bit Register	294
C64 CPU Memory Mapped Registers	296

New CPU Memory Mapped Registers	296
MEGA65 CPU Math Unit Registers	300
MEGA65 Hypervisor Mode	307
Reset	307
Entering / Exiting Hypervisor Mode	308
Hypervisor Memory Layout	309
Hypervisor Virtualisation Control Registers	312
Programming for Hypervisor Mode	314
F F018-Compatible Direct Memory Access (DMA) Controller	319
F018 “DMAgic” DMA Controller	321
MEGA65 DMA Controller Extensions	322
G VIC-IV Video Interface Controller	323
Features	325
VIC-II/III/IV Register Access Control	326
Detecting VIC-II/III/IV	327
Video Output Formats, Timing and Compatibility	328
Frame Timing	329
Physical and Logical Rasters	332
Bad Lines	332
Memory Interface	333
Relocating Screen Memory	333
Relocating Character Generator Data	334
Relocating Colour / Attribute RAM	334
Relocating Sprite Pointers and Images	335
Hot Registers	336
New Modes	336
Why the new VIC-IV modes are Character and Bitmap modes, not Bitplane modes	336

Displaying more than 256 unique characters via "Super-Extended Attribute Mode"	337
Using Super-Extended Attribute Mode	340
Full-Colour (256 colours per character) Text Mode	341
Many-colour (16 colours per character) Text Mode	341
Alpha-Blending / Anti-Aliasing	341
Flipping Characters	341
Variable Width Fonts	342
Raster-Rewrite Buffer	342
Sprites	343
VIC-II/III Sprite Control	343
Extended Sprite Image Sets	343
Variable Sprite Size	344
Variable Sprite Resolution	344
Sprite Palette Bank	344
Full-Colour Sprite Mode	345
VIC-II / C64 Registers	347
VIC-III / C65 Registers	351
VIC-IV / MEGA65 Specific Registers	354
H 6526 Complex Interface Adapter (CIA) Registers	359
CIA 6526 Registers	361
CIA 6526 Hypervisor Registers	364
I 4551 UART, GPIO and Utility Controller	367
C65 6551 UART Registers	369
4551 General Purpose IO & Miscellaneous Interface Registers	370

J 45E100 Fast Ethernet Controller	373
Overview	375
Differences to the RR-NET and similar solutions	375
Theory of Operation: Receiving Frames	376
Accessing the Ethernet Frame Buffers	377
Theory of Operation: Sending Frames	379
Advanced Features	379
Broadcast and Multicast Traffic and Promiscuous Mode	379
Debugging and Diagnosis Features	380
Memory Mapped Registers	381
COMMAND register values	382
Example Programs	383
K Reference Tables	385
Units of Storage	387
Base Conversion	388
L Flashing the FPGAs and CPLDs in the MEGA65	393
Warning	395
Flashing the Artix 100T main FPGA with XILINX VIVADO	396
Flashing the CPLD in the MEGA65's Keyboard with LATTICE DIAMOND	407
Flashing the MAX10 FPGA on the MEGA65's Mainboard with INTEL QUARTUS	417
M Model Specific Features	425
MEGA65 Desktop Computer, Revision 2 onwards	427
MEGApHONE Handheld, Revisions 1 and 2	428
Nexys4 DDR FPGA Board	429

N	Supporters & Donors	431
Organisations	433	
Volunteers	433	
Individual Donors	433	
INDEX	439	
VI	ELEMENT CATALOGUE	443
Graphic Symbols Font	444	
Handy Symbols	444	
Keyboard keys	445	
Screen Output	445	
Screen font mapping	446	
Sprite Grids	446	
Balloon Sprite Demo	446	
Multicolor Sprite	447	

1

CHAPTER

Introduction

Congratulations on your purchase of one of the most long-awaited computers in the history of computing. The MEGA65 is a community designed computer, based on the never-released Commodore® 65¹ computer, that was first designed in 1989, and intended for public release in 1990. Twenty-eight years have passed since then, but the simple, friendly nature of the 1980s home computers is still something that hasn't been recreated. These were computers that were simple enough that you could understand not just how to work with your computer, but how computers themselves work.

Many of the people who grew up using the home computers of the 1980s now have exciting and rewarding jobs in many companies, in part because of what they learnt about computers in the comfort of their own home. We want to give you that same opportunity, to experience the joy of learning how to use computers to solve all sorts of problems: writing a letter to a friend, working out how much tax you owe, inventing new things, or discovering how the universe works. This is why we made the **MEGA65**.

The MEGA65 team thinks that owning a computer should be like owning a home: You don't just use a home, you change things big and small to really make it your own, and maybe even renovate it or add on a room or two. In this guide we will show you how to more than just hang your own pictures on the wall, but instead how you can dream up new ways of using the powerful capabilities of computers by coding your own computer programmes, and even changing the computer itself!

To help you have fun with your MEGA65, we will show you how to use the exciting **graphics** and **sound** capabilities of the MEGA65. But the MEGA65 isn't just about writing your own programmes. It can also run many of the thousands of games and other programmes that were created for the Commodore® 64™² computer.

Welcome to the world of the **MEGA65**!

¹Commodore is a trademark of C= Holdings

²Commodore 64 is a trademark of C= Holdings,

PART I

GETTING TO KNOW YOUR MEGA65

CHAPTER **2**

SETUP

- **Unpacking and connecting the MEGA65**
- **Rear Connections**
- **Side Connections**
- **Installation**
- **Optional Connections**
- **Operation**

UNPACKING AND CONNECTING THE MEGA65

Time to set up your MEGA65 home computer. The box contains the following:

- MEGA65 computer.
- Power supply (black box with socket for mains supply).
- This book, the MEGA65 User's Guide.

In addition, to be able to use your MEGA65 computer:

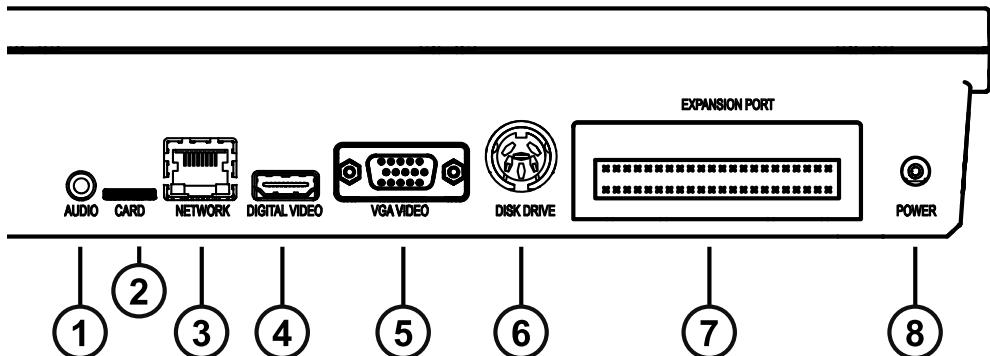
- A television or computer monitor with a VGA or digital video input, that is capable of displaying an image with 800x600 pixel resolution at 50Hz or 60Hz.
- A VGA video cable, or;
- A digital video cable.

These items are not included with the MEGA65.

You may also want to use the following to get the most out of your MEGA65:

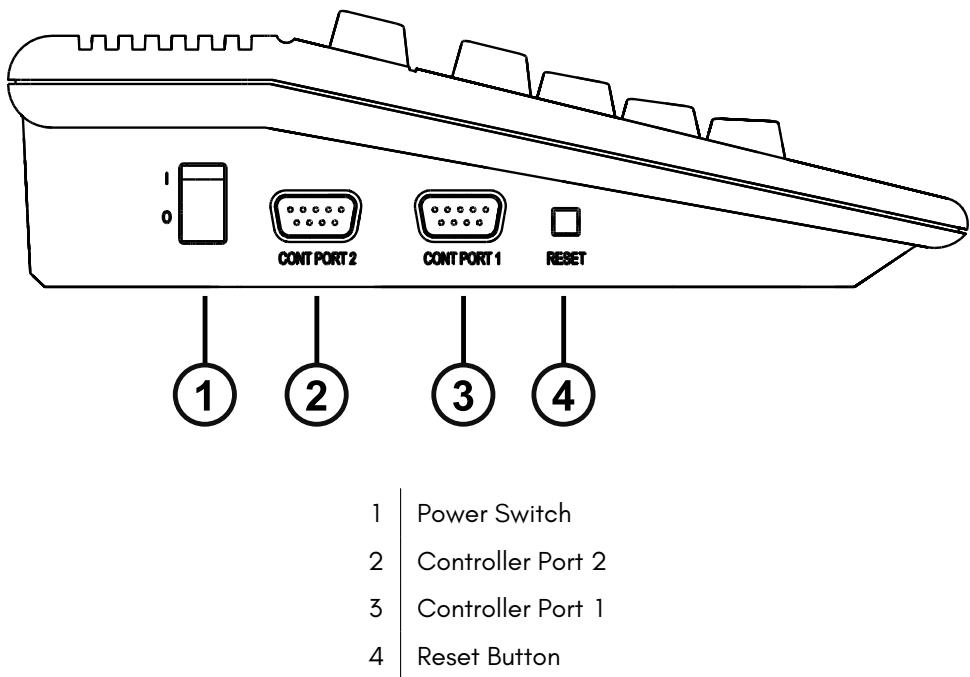
- 3.5mm mini-jack audio cable and suitable speakers or hifi system, so that you can enjoy the sound capabilities of your MEGA65.
- RJ45 ethernet cable (regular network cable) and a network router or switch. This allows use of the high-speed networking capabilities of your MEGA65.

REAR CONNECTIONS



- | | |
|---|----------------------------|
| 1 | 3.5mm Audio Mini-Jack |
| 2 | SDCard |
| 3 | Network LAN Port |
| 4 | Digital Video Connector |
| 5 | VGA Video Connector |
| 6 | External Floppy Disk Drive |
| 7 | Cartridge Expansion Port |
| 8 | DC Power-In Socket |

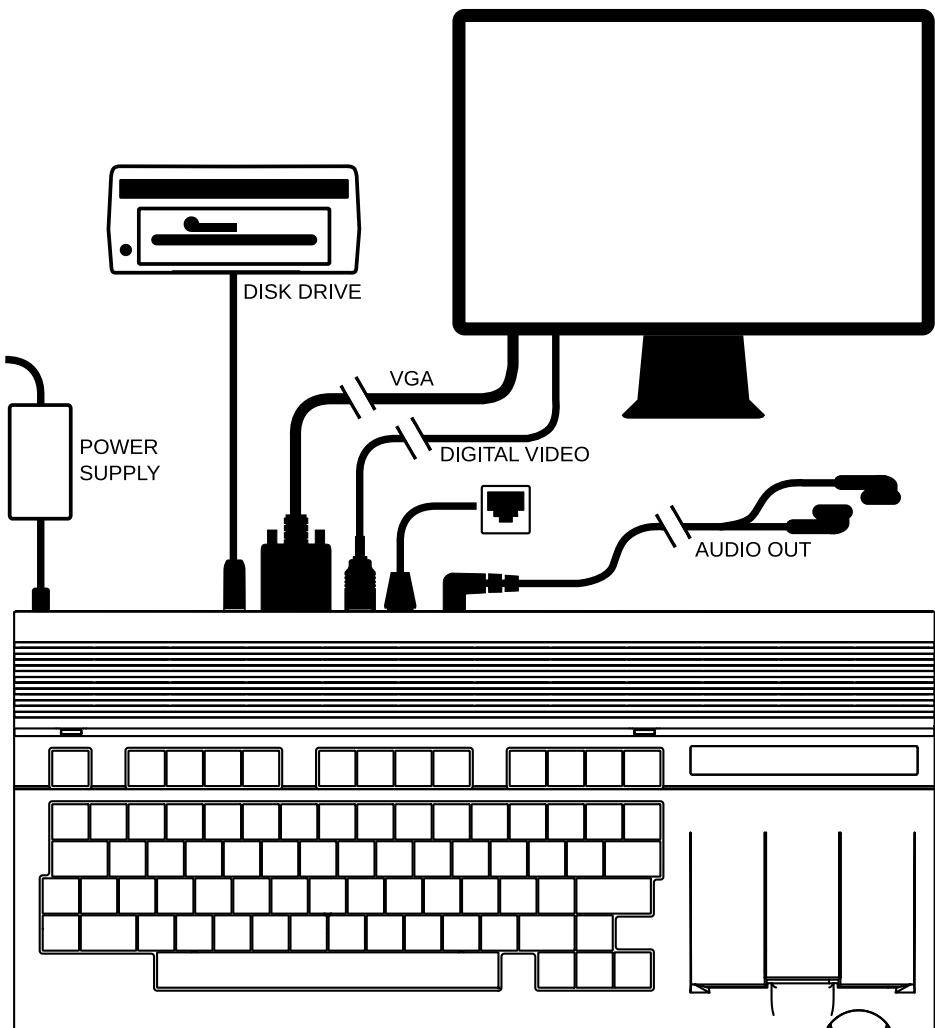
SIDE CONNECTIONS



Various peripherals can be connected to Controller Ports 1 and 2 such as joysticks or paddles.

INSTALLATION

Connecting your MEGA65 to a screen and peripherals



1. Connect the power supply to the Power Supply socket of the MEGA65.
2. If you have a VGA monitor and a VGA cable, connect one end to the VGA port of the MEGA65 and the other end into your VGA monitor.
3. If you have a TV or monitor with a Digital Video connector, connect one end of your cable to the Digital Video port of the MEGA65, and the other into the Digital Video port of your monitor. If you own a monitor with a DVI socket, you can purchase a DVI to Digital Video adapter.

OPTIONAL CONNECTIONS

1. The MEGA65 houses an internal 3.5" floppy disk drive. You can also connect older Commodore® IEC serial floppy drives to the MEGA65: the Commodore® 1541, 1571 or 1581. Connect one end of your IEC cable to the Commodore® floppy disk drive and the other end to the Disk Drive socket of the MEGA65. You can also connect SD2IEC devices and PI1541's. It is possible to daisy-chain additional floppy disk drives or Commodore® compatible printers.
2. You can connect your MEGA65 to a network using a standard ethernet cable.
3. For enjoying audio from your MEGA65, you can connect a 3.5mm stereo mini-jack cable to an audio amplifier or speaker system. If your system has RCA connectors you will need to purchase a 3.5mm mini-jack to twin RCA adapter cable. The MEGA65 also has a built in amplifier to allow connecting headphones.
4. A Secure Digital Card or SDCard (SDHC and SDXC) can be inserted into the rear of the MEGA65 as a drive.

OPERATION

Using the MEGA65

1. Turn on the computer by using the switch on the left hand side of the MEGA65.
2. After a moment, the following will be displayed on your TV or monitor:

THE COMMODORE C65 DEVELOPMENT SYSTEM
COPYRIGHT 1991 COMMODORE ELECTRONICS, LTD.
BASIC 10.0 V0.9.910111 ALL RIGHTS RESERVED

ENGLISH KEYBOARD
EXPANSION RAM

READY.

The flashing cursor
indicates the MEGA65
is ready for input.

THE CURSOR

The flashing square underneath the READY prompt is called the cursor. The cursor indicates that the computer is ready to accept input. Pressing keys on the keyboard will print that character onto the screen. The character will be printed in the current cursor position, and then the cursor advances to the next position.

You can type commands, for example: telling the computer to load a program. You can even start entering program code.

CHAPTER

3

GETTING STARTED

- **Keyboard**
- **The Screen Editor**
- **Editor Functionality**

KEYBOARD

Now that you have everything connected, it's time to get familiar with the MEGA65 keyboard.

You may notice that the keyboard is a little different from the standard used on computers today. While most keys will be in familiar positions, there are some specialised keys, and some with special graphic symbols marked on the front.

Here's a brief description of how some of these special keys function.

Command Keys

The Command Keys are: **RETURN**, **SHIFT**, **CTRL**,  and **RESTORE**.

RETURN

Pressing the **RETURN** key enters the information you have typed into the MEGA65's memory. The computer will either act on a command, store some information, or return you an error if you made a mistake.

SHIFT

The two **SHIFT** keys are located on the left and the right. They work very much like Shift on a regular keyboard, however they also perform some special functions too.

In upper case mode, holding **SHIFT** and pressing any key with a graphic symbol on the front produces the right hand symbol on that key. For example, **SHIFT** and **J** prints the  character.

In lower case mode, pressing **SHIFT** and a letter key prints the upper case letter on that key.

Holding both shift keys down when turning the machine on activates the Utility Menu. You can format the SD card or enter the MEGA65 Configuration Utility to select the default video mode and other settings.

Finally, holding the **SHIFT** key and pressing a Function key accesses the function shown on the front of that key. For example: **SHIFT** and **F1** activates **F2**.

SHIFT LOCK

In addition to the Shift key is **SHIFT LOCK**. Press this key to lock down the Shift function. Now any key you press prints the character to the screen as if you were holding down Shift. That includes special graphic characters.

CTRL

CTRL is the Control key. Holding down Control and pressing another key allows you to perform Control Functions. For example, holding **CTRL** and one of the number keys allows you to change text colours.

There are some examples of this in the Screen Editor chapter, and all the Control Functions are listed in Appendix C Control codes.

If a program is being listed to the screen, holding **CTRL** slows down the display of each line on the screen.

Holding **CTRL** and pressing ***** enters the Matrix Mode Debugger.

RUN/STOP

Normally, pressing the **RUN STOP** key stops execution of a program. Holding **SHIFT** while pressing **RUN STOP** loads the first program from disk.

Programs are able to disable the **RUN STOP** key.

You can boot your machine into the machine code monitor by holding down **RUN STOP** and pressing reset on the MEGA65.

RESTORE

The computer screen can be restored to a clean state without clearing the memory by holding down the **RUN STOP** key and tapping **RESTORE**.

Programs are able to disable this key combination.

Enter the Freeze Menu by holding **RESTORE** for more than one second. You can access the machine code monitor via the Freeze menu.

THE CURSOR KEYS

At the bottom right hand of the keyboard are the cursor keys. These four directional keys allow you move the cursor to any position for onscreen editing.

The cursor moves in the direction indicated on the keys: 

However, it is also possible to move the cursor up using **SHIFT** and . In the same way you can move the cursor left using **SHIFT** and .

You don't have to keep pressing a cursor key over and over. When moving the cursor a long way, you can keep the key pressed in. When you are finished, release the key.

INSerT/DELeTe

This is the INSERT / DELETE key. When pressing , the character to the left is deleted, and all characters to the right are shifted one position to the left.

To insert a character, hold the **SHIFT** key and press . All the characters are shifted to the right. This allows you to type a letter, number or any other character into the newly inserted space.

CLeAr/HOME

Pressing the  key returns the cursor into the top left-most position of the screen.

If holding **SHIFT** and pressing  clears the entire screen and places the cursor into the top left-most position of the screen.

MEGA KEY

The  key or the MEGA key provides a number of different functions and special utilities.

Holding the **SHIFT** key and pressing  switches between lower and upper case character modes.

Holding  and pressing any key with graphic symbols on the front prints the left-most graphic symbol to the screen.

Holding  and pressing any key that shows a single graphic symbol on the front prints that graphic symbol to the screen.

Holding  and pressing a number key switches to one of the colours in the second range.

Holding  and pressing **TAB** enters the Matrix Mode Debugger.

When turning on the MEGA65 or pressing the reset button on the side, while holding  switches the MEGA65 into C64 mode.

NO SCROLL

If a program is being listed to the screen, pressing  freezes the screen output. Not available in C64 mode.

Function Keys

There are seven Function keys available for use by software applications, **F1** / **F3**, **F5**, **F7**, **F9**, **F11** and **F13** to perform functions with a single press.

Hold **SHIFT** to access **F2** through to **F14** as shown on the front of each Function key.

Only Function keys **F1** to **F8** are available in C64 mode.

HELP

The  key can be used by software and acts as an **F15** / **F16** key.

ALT

The  held while pressing other keys can be used by software to perform functions. Not available in C64 mode.

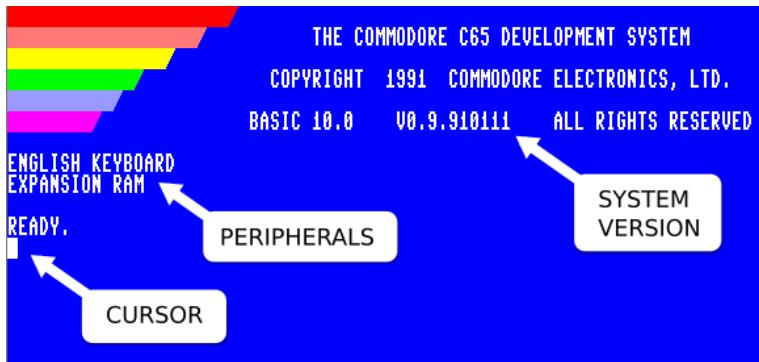
CAPS LOCK

The  works like  in C65 and MEGA65 modes, but only modifies the alphabet keys.

Also, holding the  down forces the processor to run at the maximum speed. This can be used, for example, to speed up loading from the internal disk drive or SD card, or to greatly speed up the depacking process after a program is run. This can reduce the loading and depacking time from many seconds to as little as a 10th of a second.

THE SCREEN EDITOR

When you turn on your MEGA65, or reset it, the editor screen will appear.



The colour bars in the top left hand of the screen can be used as a guide to help calibrate the colours of your display. The screen also displays the name of the system, the copyright notice and what version and revision of BASIC is contained in the Read-only Memory.

Also displayed is the type of keyboard and whether or not there is additional hardware present, such as a RAM expansion.

Finally, you will see the READY prompt and the flashing cursor.

You can begin typing keys on the keyboard and the characters will be printed under the cursor. The cursor itself advances after each key press.

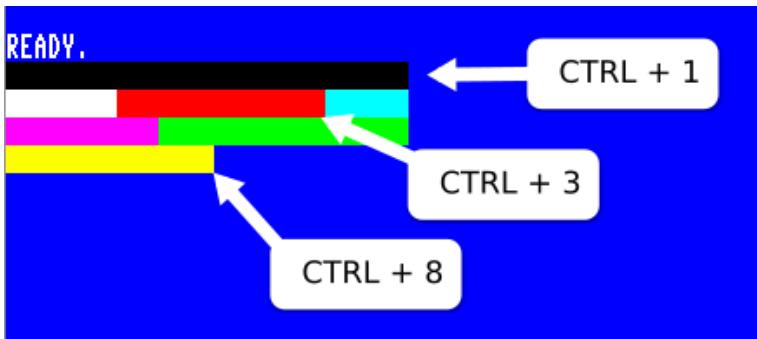
You can also produce reverse text or colour bars by holding down the **CTRL** key and pressing the **9** key or the **R** key. This enters reverse text mode.

Try holding down the **SPACE BAR**. A white bar will be drawn across the screen.

You can even change the current colour by holding down the **CTRL** key and pressing a number key. Try key **8** and then hold down the **SPACE BAR** again. A yellow bar will be drawn.

Change the bar to a number of other colours.

You will get an effect something like:



You can turn off the reverse text mode by holding **CTRL** and pressing the **0** key.

By pressing any keys, the characters will be typed out in the chosen colour.

There are a further eight colours available via the **M** key. Hold the **M** key and press a key from **1** to **8** to change to one of the secondary colours. For even more colours, see Escape Sequences in Appendix C.

Functions

Functions using the **CTRL** key are called **Control Functions**. Functions using the **M** key are called **Mega Functions**. There are also functions called by using the **SHIFT** key. These are (not surprisingly) called **Shift Functions**.

Lastly, using the **ESC** key are **Escape Sequences**.

ESC Sequences

Escape sequences are performed a little differently than a Control function or a Shifted function. Instead of holding the modifier key down, an Escape sequence is performed by pressing the **ESC** key once, then pressing the desired key code.

For example: to switch between 40/80 column mode, press and release the **ESC** key. Then press the **X** key.

You can see all the available Escape Sequences in Appendix C. We will cover some examples of these shortly.

There are more modes available. You can create flashing text by holding the **CTRL** key and pressing the **O** key. Any characters you press will flash. Turn flash mode off by pressing **ESC** then **O**.

EDITOR FUNCTIONALITY

The MEGA65 screen can allow you to do advanced tabbing, and moving quickly around the screen in many ways to help you to be more productive.

Press the **HOME** key to go to the home position on the screen. Hold the **CTRL** key down and press the **W** key several times. This is the **Word Advance function** which jumps your cursor to the next word, or printable character.

You can set custom tab positions on the screen for your convenience. Press **HOME** and then **→** to the fourth column. Hold down **CTRL** and press the **X** key to set a tab. Move another 20 positions to the right again, and do **CTRL** and **X** again to set a second tab.

Press the **HOME** key to go back to the home position. Hold the **CTRL** key and press the **I** key. This is the **Forward Tab function**. Your cursor will tab to the fourth position.

Press **CTRL** and **I** again. Your cursor will move to position 8. Why? By default, every 8th position is already set as a tabbed position. So the 4th and 20th positions have been added to the existing tab positions. You can continue to press the **CTRL** and **I** keys to the 16th and 20th positions.

To find the complete set of Control codes, see Appendix C Control codes.

Creating a Window

You can set a window on the MEGA65 working screen. Move your cursor to the beginning of the "BASIC 10.0" text. Press **ESC**, then press **T**. Move the cursor 10 lines down and 15 to the right.

Press the **ESC** key, then **B**. Anything you type will be contained within this window.

To escape from the window back to the full screen, press the **HOME** key twice.

Extras

Long press on **RESTORE** to go into the Freeze Menu. Then press **J** to switch joystick ports without having to physically swap the joystick to the other port.

Go to **Fast mode** with poke 0, 65 or go to the freeze menu.

MEGA and **SHIFT** switches between text uppercase and lowercase for the entire display.

CHAPTER 4

Cores and Flashing

- What are cores, and why does it matter?
- Selecting a core
- Installing an upgrade core for the MEGA65
- Installing other core for the MEGA65
- Creating cores for the MEGA65
- Replacing the factory core in slot 0

WHAT ARE CORES, AND WHY DOES IT MATTER?

The MEGA65 computer uses a versatile chip called an FPGA as its heart. FPGAs are “Field Programmable Gate Arrays”. This is a fancy way of saying that FPGAs are chips that can be programmed to behave impersonate other chips. They do this by configuring their arrays of logic gates to reproduce the circuits of other chips. In this way, FPGAs are not emulation but re-creation of other chips.

FPGAs forget what chip they were pretending to be whenever the power is turned off, or when they are “reconfigured”. This might sound annoying, but it’s actually really powerful. It means that we can tell the FPGA in the MEGA65 to impersonate not just the MEGA65 design as it currently stands, but to impersonate any improvements we make to the design. In other words, we can upgrade the MEGA65 hardware just by providing a new set of instructions to the FPGA. These sets of instructions are called “cores” or “bit-streams”. For the purpose of the MEGA65, these two terms can usually be considered to be interchangeable.

FPGAs are so flexible, that not only is it possible to teach the MEGA65 to be a better MEGA65, but it is also possible to teach the MEGA65’s’ FPGA to be other interesting home computers. We believe that the FPGA is powerful enough that it could pretend to be a VIC-20 (tm), Commodore PET (tm), Apple II (tm), Spectrum (tm), BBC Micro (tm), or even an Amiga (tm) or one of the 16-bit era game consoles. Unlike some previous FPGA-based retro-computers, the MEGA65, its FPGA instructions, board layout and other information is all available for free under open-source licenses. This means that anyone is free to create other cores for the MEGA65 hardware.

To top it all off, the MEGA65 has enough storage for 7 different sets of FPGA instructions, so that you can easily switch the MEGA65’s “personality” from being a MEGA65 to another of these systems (once the cores are available) and back again.

The remainder of this chapter describes how to select which core to run on the MEGA65, and how to store a core into one of the seven slots in the flash memory storage.

SELECTING A CORE

To operate the MEGA65 using an alternative core, turn off the power to the MEGA65, and then hold the **TAB** key down while turning the power back on. This instructs the MEGA65 to enter the flash and core menu, instead of booting normally. You should see a display like the following:

XXX - Screen shot of flash menu with 7 empty slots.

To select a core and start it, use the cursor keys to highlight the desired core, and then press the **RETURN** key. Alternatively, you can press the number corresponding to the core you would like to use. The MEGA65 immediately reconfigures the FPGA, and launches the core. If for some reason the core is faultly, the MEGA65 may instead restart normally after a few seconds, and depending on the circumstances, take you automatically back into the menu.

The MEGA65 will keep running the new core until you physically power it off. Pressing the reset button will not reset which core is being run.

INSTALLING AN UPGRADE CORE FOR THE MEGA65

To install an upgrade core for the MEGA65, there are few easy steps.

First, copy the core file onto the MEGA65's SD card. You can do this by removing the SD card and inserting it into another computer that has internet access, and downloading the core from that computer. Alternatively you can insert an SD card that already contains the upgrade core. Finally, you can use the MEGA65 TFTP Server program and the MEGA65's ethernet port to copy the core upgrade file onto the SD card from another computer on your local network.

Second, once you have the upgrade core on the MEGA65's SD card, enter the flash and core menu as above, i.e., turn off the power, hold the **TAB** key down while turning the power on. When the flash and core menu appears, hold the **CTRL** key down and press the **1** key. The MEGA65 will present you a list of core files that are on the SD card. Select the upgrade core file you wish to install using the cursor keys, and then pressing the **RETURN** key. The MEGA65 will then erase the flash slot, before writing the upgraded core. This process is quite slow, and can take around 15 minutes.

It is important to not turn the power off during this process. If you do, the core file will be only partially installed, and the MEGA65 may not start properly. If this happens, enter the flash and core menu as described above, and follow the instructions again. When the process completes, you will see a message like this, indicating that the process is complete:

XXX

When this happens, simply turn off the power to the MEGA65 and turn it back on for it to start using the upgraded core. This is because the MEGA65 will always try to automatically start the core in slot 1 when it is turned on.

INSTALLING OTHER CORE FOR THE MEGA65

Installing other cores works very similarly to installing upgrade cores. The only difference is that you press **CTRL** and **2** to **7** from the flash and core menu, so that the core gets installed in another slot.

Of course, there is nothing stopping you installing a different core in slot 1, so that the MEGA65 behaves as a different type of computer when you turn it on. If you do this, you can always choose to run the MEGA65 core by entering the flash and core menu, and selecting the MEGA65 core.

CREATING CORES FOR THE MEGA65

If you would like to create your own cores for the MEGA65, or help contribute to the MEGA65 core, then you may also wish to take a look at Appendix L which explains how to use the FPGA development tools to flash the MEGA65.

REPLACING THE FACTORY CORE IN SLOT 0

Replacing the core in slot 0 is not recommended, because if you mess it up, it will brick the machine. This may require that you have to purchase a TE-0790 JTAG programmer, open your MEGA65 case, install the module, go through some rather convoluted software preparation steps, similar to if you were creating your own bitstream/core, and then restore a working bitstream into the slot.

The MEGA65 is an open system though. Therefore we have not made it impossible for you to do this, just very hard: There is a secret key press in the flash menu that will then challenge you with a series of questions of increasing difficulty, to ensure that you know what you are doing. Only after you have correctly answered these questions, will you be given the option to erase and/or replace the contents of slot 0. We are purposely not

documenting the method for doing this, nor further details of the questions you will be asked.

However, there really should be no reason for using this method to replace the contents of slot 0: If you want to make your own bitstreams/cores, you can either put them in other slots, and use the flash menu to activate them, or you should simply use a TE-0790 JTAG adapter to begin with, and use the Vivado or other FPGA development tools to write to the flash directly. That method is also somewhat faster than flashing through the flash menu.

Anyway, you have been warned.

PART

FIRST STEPS IN CODING

CHAPTER

5

How Computers Work

- Computers are stupid. Really stupid

Did you know that many computer experts and programmers learned how to use computers when they were still small children? Home computers only became common in the early 1980s. They were so new, that people would often write programmes to do what they wanted to do, because no software existed to do the job for them.

It was also quite common for people working in all sorts of office jobs to learn how to program the computers they used for their jobs. For example, the people processing payroll for a company would often learn how to program the computer to calculate the everyone's pay!

Things have changed a lot since then, though. Now most people choose existing programmes or apps to do what they need, and think that programming is a specialised skill that only some people have the ability to learn. But this isn't true. Of course, like every other field of pursuit everyone will be better at some things than others, whether it be sports, knitting, maths or writing. But almost everyone is able to learn enough to help them in their life.

We created the MEGA65, because we believe that YOU can learn to programme, so that computers can be more useful to you, and as with learning any new skill, that you can have the satisfaction and enjoyment and new adventures that this brings!

COMPUTERS ARE STUPID. REALLY STUPID

How can this be so? Computers are able to do so many different things, often thousands of times faster than a person can. So how can we say that computers are stupid? The answer is that no computer can do anything that it hasn't been instructed by a person to do. Even the latest Artificial Intelligence systems were instructed how to learn (or how to learn, how to learn). To understand why this is so, it is helpful to understand how computers really work.

Making an Egg Cup Computer

The heart of a computer is its Central Processing Unit, or CPU for short. Many modern computers have more than one CPU, but let's keep things simple to begin with. The CPU has a set of simple instructions that it understands, like, "get the thing from cup #21," "put this thing into cup #403," "add these things together," or "do the following instruction, but only if the thing in cup #712 is the number 3."

But what do we mean with all of these "things" and "cups"? Let's start by thinking about how we could pretend to be a computer using just an empty egg carton, some small pieces of paper and a pencil or pen. Start by writing numbers, beginning with one, in

each of the little egg cups in the egg carton. Then write the number zero on a little scrap of paper and put it in the first cup. Do the same for the other cups. You should now have an egg carton with numbered cups, and with every cup having a scrap of paper with the number zero written on it. Now we just need to decide on a few simple rules that will explain how our egg-cup computer will work:

- First, each cup is allowed to hold exactly one thing at a time. Never more. Never less. This so that when we ask the question “what is in box such-and-such,” that there is a single clear answer. It’s also how computer memory works: Each piece of memory can hold only one thing at a time.
- Second, we need a way for the computer to know what to do next. On most computers this is called the Programme Counter, or PC, for short (not to be confused with PC when people are talking about a Personal Computer). The PC is just the number of the next of the next memory location (or in our case, egg-cup), that the computer will examine, when deciding what to do next. You might like to have another piece of paper that you can use to write the PC number on as you go along.
- Third, we need to have a list of things that the egg-cup computer will do, based on what number is in the egg-cup indicated by the PC.

So let’s come up with the set of things that the computer can do, based on the number in the egg-cup indicated by the PC. We’ll keep things simple with just the following:

Number in the egg-cup	Action
0	i) Add one to the PC, and do nothing else.
1	i) Add one to the PC. ii) Set the PC to be the number stored in that egg-cup.
2	i) Add one to the PC. ii) Add the number in the egg-cup indicated by the PC to the number in the egg-cup indicated by the number in the egg-cup following that. iii) Put the answer in the egg-cup indicated by the egg-cup following that. iv) Finally, add two more to the PC, to skip over the egg-cups that we made use of.

Don’t worry if that sounds a bit confusing for now, specially that last one – we will go through it in detail very soon! The best way to explain it, is to go through some examples.

6

CHAPTER

C64, C65 and MEGA65 Modes

- Switching Modes from BASIC
- The KEY Register
- Accessing the MEGA65's Extra
Memory from BASIC 10 in
C65 Mode
- The MAP Instruction

The MEGA65, like the C65 and the C128 has multiple operating modes, however there are important differences between the MEGA65 and both of these earlier computers. For people used to using the C128, the most important difference is that all of the MEGA65's new features can be accessed from every mode, and you can even switch back and forth between the different modes, or create hybrid modes that combine different features from the different modes – all you need is the MAP and KEY.

In this chapter we explain the different modes, and the MAP instruction and the KEY register that let you change the mode of operation of the computer, as well as being able to use BASIC commands that can be used to completely switch from one mode to another.

SWITCHING MODES FROM BASIC

At the time of writing, there is no MEGA65 Mode BASIC: The computer is used either in C64 mode, running BASIC 2, or C65 mode, running BASIC 10. However, various MEGA65 features can be accessed from both C64 and C65 mode. All MEGA65 features are also available to programmes written in assembly language / machine code programmes. The information required to write such programmes can be found in the various appendices.

From C65 to C64 Mode

To switch from C65 to C64 Mode, use the familiar GO 64 command, identically to switching to C64 mode on a C128:

```
GO 64  
ARE YOU SURE? Y
```

Note that, just like on the C128, any programmes in memory will be lost in the process of switching modes.

Alternatively, you can hold down the MEGA key when pressing the reset button or turning the computer on. Again, this is the same as on the C128.

From C64 to C65 Mode

To switch from C64 to C65 Mode, there is no official method. However the following command. Note that this command does not ask you for confirmation!

```
SYS 58552
```

Alternatively, you can switch back to C65 mode by pressing the reset button on the left side of the computer, or simply turning the computer off and on again.

Another option is to long-press the RESTORE key, and then choose F5 from the freeze menu. This simulates pressing the reset button.

Note that, just like on the C128, any programmes in memory will be lost in the process of switching modes.

Entering Machine Code Monitor Mode

The machine code monitor can be entered by typing either the MONITOR command from BASIC 10 in C65 mode, or by holding the RUN/STOP key down, and then pressing the reset button on the left side of the computer.

THE KEY REGISTER

The MEGA65 has a VIC-IV video controller chip instead of the C64's VIC-II or the C65's VIC-III. Just as the VIC-III has extra registers compared to the VIC-II, the VIC-IV has even more registers. If these were visible all of the time, software that was made for the C64 and its VIC-II might accidentally use the new registers, resulting in unexpected or unhelpful results. Therefore the creators of the C65 invented a way to hide the extra VIC-III registers from old C64 programs. This is called the KEY register.

The KEY register 53295 (hex \$D02F) is just an unused register of the VIC-II, that you can POKE and PEEK like the other registers. But the KEY register has a special function: If you write two special values to it in quick succession, you can tell the VIC-IV to stop hiding the VIC-III or VIC-IV registers from the rest of the computer.

Un-hiding C65 Extra Registers

For example, to unhide the VIC-III's new registers when in C64 mode, you must POKE the values 165 and 150 into the KEY register. Make sure you are in C64 mode before trying the following. The easiest way to do this is to turn your MEGA65 off and on again, and type GO 64 and answer YES to enter C64 mode.

(If you do it from C65 mode, the computer will get rather confused, because in between the first and second POKE commands running, none of the C65-mode extra features will be visible to the computer, and BASIC 10 will probably crash or freeze as a result. But don't worry, if you accidentally do this, just turn the computer off and on again, or press and release the reset button the left side of the computer.)

Once you are in C64 mode, try typing the following commands:

```
POKE 53295,165: POKE 53295,150
```

When you type these commands, the computer just returns with a READY. prompt, and seemingly nothing else has happened. This is expected, because all we have done is to unhide the VIC-III's new registers (and some other C65 mode features) from the computer. However, the C64 BASIC and KERNAL are well behaved, and don't try to do anything strange, and so we don't immediately notice anything is different... But things are different.

As an example, we will now do something that the C64 and its VIC-II can't do: smoothly change one colour into another. The VIC-III has registers that let you change the red, green and blue components of the colours. So now that we have unhidden those registers, we can change the colour of the background progressively from blue to purple, by increasing the red component of the colour that is normally blue on the C64. The red component value registers are at 53504 – 53759 (hex \$D100 – \$D1FF). Blue is colour 6, so we want to change register $53504 + 6 = 53510$ (hex \$D106). We can do a nice FOR loop to change the colour for us:

```
FOR I = 0 TO 15 STEP 0.2 : POKE 53510,I : NEXT
```

You should hopefully have seen the background of the screen fade from blue to purple. If you would like to make the effect go faster, increase the 0.2 to a bigger number, like 0.5, or to make it slower, make it a smaller number, like 0.02. You can also change the red component you are changing by adding a different number to 53504. Or you might like to change the green component (53760 – 54015, hex \$D200 – \$D2FF) or blue component (54016 – 54271) – or some combination! For example, to make the border and text (since they are both normally "light blue") fade from blue to green, you could do:

```
POKE 53518,0 : FOR I = 0 TO 15 STEP 0.1 : POKE 53774,I : POKE 54030,15-I : NEXT
```

Re-hiding C65/MEGA65 Extra Registers

You can also hide the VIC-III registers again by POKEing any number you like into the KEY register, e.g.:

```
POKE 53295,0
```

If you now try the colour changing examples from above, they shouldn't change the colours this time, because the registers are hidden again. Instead, writing to those addresses changes some of the VIC-II's registers because on a C64 they appear several times over.

Fortunately, we chose an example where the registers don't have any ill-effect in our case (for the curious, it is the sprite positions that would be messed up, but since there are no sprites on the screen, we don't see any problems).

Un-hiding MEGA65 Extra Registers

The MEGA65 has even more registers than the C65. To un-hide those from C64 mode, we write two different values into the KEY register:

```
POKE 53295,71: POKE 53295,83
```

(Don't forget you have to be in C64 mode, as BASIC 10 will probably crash or freeze when the C65 / VIC-III registers get briefly hidden after the first POKE has happened, but the second one has not yet happened.)

Again, you won't see any immediate difference, just like when unhiding C65 / VIC-III registers. However, now the computer can access not only the C64 / VIC-II and C65 / VIC-III registers, but also the MEGA65 / VIC-IV registers. If you like, you can try the examples from earlier in this chapter, to assure yourself that the C65 / VIC-III registers are accessible again. But you can also do MEGA65 specific things. But we can of course also do MEGA65 specific things. For example, if we wanted to move the start of the top border higher up the screen, we could type something like:

```
POKE 53320,60
```

Or again, we could have some fun, and animate the screen borders moving closer and further apart:

```
FOR I = 255 TO 0 STEP -1 : POKE 53320,I : POKE 53322, 255 - I : NEXT
```

(We made this loop go backwards, so that you wouldn't end up with only a tiny sliver of the screen visible. But you can make it go forwards if you like. If you do get stuck with a sliver of the screen, you can just press RUN/STOP and RESTORE. Maybe you are wondering why RUN/STOP and RESTORE works, when these are MEGA65 / VIC-IV registers that the C64-mode BASIC and KERNAL don't know about. The reason it works is because the VIC-IV has a feature called "hot registers," where certain C64 and C65 registers cause some of the MEGA65 registers to be reset to the C64 or C65 mode defaults. In this particular case, it is the KERNAL resetting the VIC-II screen using 53265 (hex \$D011), which adjusts the vertical border size in C64/C65 mode, and is thus a "hot register" for the MEGA65's vertical border position registers.)

See if you can instead make the screen shake around by changing the TEXTXPOS and TEXTYPOS registers of the VIC-IV. You can find out the POKE codes for those and lots of other interesting new registers by looking through Appendix G.

Traps to watch out for

In both C64 and C65 mode, the DOS for the internal 3.5" disk drive (including when you use D81 disk images from an SD card) resets the KEY register to C64 / VIC-II mode whenever it is accessed. This means if you check the drive status, LOAD or SAVE a file, for example, that the KEY register will be reset, and only the C64 / VIC-II registers will be visible. You can of course make the C65 or MEGA65 extra registers visible again by POKEing the correct values to the KEY register again.

ACCESSING THE MEGA65'S EXTRA MEMORY FROM BASIC 10 IN C65 MODE

The C65's BASIC 10 contains powerful memory banking and Direct Memory Access (DMA) commands that can be used to read, fill, copy and write areas of memory beyond the C65's 128KB of RAM. The MEGA65 has 384KB of main memory. Of this, the first 128KB (BANK 0 and BANK 1) acts as the C65's normal 128KB RAM. The second 128KB (BANK 2 and BANK 3) is normally write-protected, and is used to hold the C65's ROM image. The last 128KB (BANK 4 and BANK 5) is however, completely free.

Using the BANK and PEEK and POKE commands, this region of memory can be easily accessed, for example:

```
BANK 4: POKE8,123: REM PUT 123 IN LOCATION $40000
BANK 4: PRINT PEEK(0): REM SHOW CONTENTS OF LOCATION $40000
```

Or using the DMA command, you could copy the current contents of the screen and colour RAM into BANK 4 with something like this:

```
DMA 0, 2000, 2048, 0, 0, 4 : REM SCREEN TEXT TO BANK 4
DMA 0, 2000, DECK("F800"), 1, 2000, 4 : REM COPY COLOUR RAM TO BANK 4
```

You could then put something else on the screen, and then copy it back with something like:

```
DMA 0, 2000, 0, 4, 2048, 0, : REM SCREEN TEXT FROM BANK 4  
DMA 0, 2000, 2000, 4, DEC("F800"), 1 : REM COPY COLOUR RAM FROM BANK 4
```

Note that there is currently no way to tell BASIC 10 to put graphics screen, variables, arrays or program text in these extra banks of RAM.

THE MAP INSTRUCTION

The above methods can be used from BASIC. In contrast, the MAP instruction is an assembly language instruction that can be used to rearrange the memory that the MEGA65 sees. It is used by the C65 ROM and BASIC 10 to manage what memory it can see at any particular point in time. For further explanation of the MAP instruction, refer to the relevant section of Appendix E.

PART III

SOUND AND GRAPHICS

PART IV

HARDWARE

CHAPTER 7

Using a Nexys4DDR as a MEGA65

- Building your own MEGA65 Compatible Computer
- Power, Jumpers, Switches and Buttons
- Keyboard
- Preparing microSDHC card
- Useful Tips

BUILDING YOUR OWN MEGA65 COMPATIBLE COMPUTER

You can build your own MEGA65-compatible computer by using a Nexys4DDR FPGA development board. This appendix describes the process to setup a Nexys4DDR board for this purpose. The older non-DDR Nexys4 board is also supported, and the instructions are the same, except that you must use a bitstream designed for that board. Using a Nexys4DDR bitstream on a non-DDR Nexys4 board, or vice versa, may cause irreparable damage to your board, so make sure you have the correct bitstream to suit your board.

DISCLAIMER: M.E.G.A cannot take any responsibility for any damage that may occur to your Nexys4DDR board.

POWER, JUMPERS, SWITCHES AND BUTTONS

The MEGA65 core consumes too much power for the Nexys4DDR board to be powered from a standard USB port. In particular, writing to the SD card might hang or perform odd behaviour. Therefore you should use a 5V power supply. Digilent sell a power supply for the Nexys4DDR board, and we recommend you use this to ensure you avoid the risk of damage to your Nexys4DDR board.

Set the following jumpers on your Nexys4DDR board to the following positions:

- JP1 - USB/SD
- JP2 - SD
- JP3 - Wall

All 16 switches on the lower edge of the board must be set to the off position.

The “CPU RESET” button will reset the MEGA65 when pressed, while the “PROG” button will cause the FPGA itself to reload the MEGA65 core. The main difference between the two is that CPU RESET is faster, and does not clear the contents of memory, while the FPGA button is slower, and does reset the contents of memory.

Two of the five buttons in the cross arrangement can also be used: BTNU acts as though you have pressed the **RESTORE** key, while BTNC will trigger an IRQ, as though the IRQ line had been pulled to ground.

KEYBOARD

Only USB keyboards that lack a USB hub will work with the Nexys4DDR board. Generally extremely cheap keyboards will work, while more expensive keyboards tend to have a USB hub integrated, and will not work. You may need to try several keyboards, before you find one that works.

The keyboard layout is positional rather than logical. This means that keys in similar positions to the keys on a C65 keyboard will have similar function. This relationship assumes that your USB keyboard uses a US keyboard layout.

The **RESTORE** key is mapped to the PAGE UP key.

PREPARING MICROSDHC CARD

The MEGA65 requires an SDHC card of between 4GB and 64GB capacity. Some SDXC cards may work, however, this is not officially supported.

To prepare your SD card, you will need the following two separate files from the MEGA65 web site:

- Bitstream <http://mega65.org/assets/bitstream.7z>
- Support Files <http://mega65.org/assets/fileset.7z>

In addition, you will also need a C65 ROM. There were many different versions created during the development of the Commodore 65, and the MEGA65 can use any of them. However, we recommend you use 911001.bin, as this has the most complete BASIC and DOS implementations.

The steps are:

- Format the SD card in a convenient computer using the FAT32 filesystem. The MEGA65 and Nexys4DDR boards do not understand other file systems, especially the exFAT file system.
- Unzip the bitstream.7z file, and copy the file with name ending in ".bit" onto the SD card.
- Insert the SD card into the SD card slot on the under-side of the Nexys4DDR board.
- Turn on the Nexys4DDR board.
- Enter the Utility Menu by holding the left and right SHIFT keys down on the USB keyboard you have connected to the Nexys4DDR board.
- Enter the FDISK/FORMAT tool by pressing 2 when the option appears on the MEGA65 boot screen.
- Follow the prompts in the FDISK/FORMAT program to again format the SD card for use by the MEGA65. (The FDISK tool will partition your SD card into two partitions and format them. One is type \$41 = MEGA65 System Partition, where the save slots, configuration data and other files live. (This partition is invisible in i.e. Win PCs). and the other partition with type \$0C = VFAT32, where KERNEL, support files, games, and so on, will be copied to later. (This partition is visible on i.e. Win PCs)).
- Once formatting is complete, switch off the Nexys4DDR board and remove the microSDHC card from the Nexys board and put it back into your PC
- Again copy the bitstream back onto the SD card, as well as all the files from the other 7z file downloaded from the MEGA65 website.

- Copy the 911001.bin ROM file onto the SD card, and rename it to MEGA65.ROM
- If you have any .D81 files, copy them onto the SD card. Make sure that they have names that fit the old DOS 8.3 character limit, and are upper case. This restriction will be removed in a future release.
- Remove the SD card and reinsert it into your Nexys4DDR board.
- Power the Nexys4DDR board back on. The MEGA65 should boot within 15 seconds.

Congratulations. Your MEGA65 has been set up and is ready to use.

USEFUL TIPS

The following are some useful tips for getting familiar with the MEGA65:

- Press & hold  (or the Commodore key if using a Commodore 64 or 65 keyboard) during boot to start up in C64 mode instead of C65 mode
- Press & hold  during boot to enter the machine language monitor, instead of starting BASIC.
- Press the **RESTORE** key for approximately 1/2 - 1 second to enter the MEGA65 Freeze Menu. From this menu you have convenient tools to change the CPU speed, switch between PAL & NTSC video mode, change Audio settings, manage freeze-states, select D81 disk images, examine and modify memory of the frozen program, among other features. This is in many ways the heart of the MEGA65, so it is well worth exploring and getting familiar with.
- Press the **RESTORE** for about 2 seconds to reset the MEGA65. This is a temporary feature that will be removed when the MEGA65 desktop computers with built-in reset button become available.
- Type `POKE0,65` in C64 mode to switch the CPU to full speed (40MHz). Some software may behave incorrectly in this mode, while other software will work very well, and run many times faster than on a C64.
- Type `POKE0,64` in C64 mode to switch the CPU to 1MHz.
- Type `SYS58552` in C64 mode to switch to C65 mode.
- Type `G064` in C65 mode and confirm, by pressing `Y`, to switch to C64 mode, just like on a C128.

- The C65 ROM makes device 8 the default, so you can normally leave off the ,**8** from the end of LOAD and SAVE commands.
- Pressing **SHIFT** + **RUN STOP** from either C64 or C65 mode will attempt to boot from disk.

Have fun! The MEGA65 has been lovingly crafted over many years for your enjoyment. We hope you have as much fun using it as we have had creating it!

The MEGA Museum of Electronic Games and Art welcomes your feedback, suggestions and contributions to this open-source digital heritage preservation project.

PART V

APPENDICES

APPENDICES

APPENDIX A

ACCESSORIES

B

APPENDIX

BASIC 10 Command Reference

- Format of Commands, Functions and Operators
- Commands, Functions and Operators

FORMAT OF COMMANDS, FUNCTIONS AND OPERATORS

This appendix describes each of the commands, functions and other callable elements of BASIC 10. Some of these can take one or more arguments, that is, pieces of input that you provide as part of the command or function call. Some also require that you use special keywords. Here is an example of how commands, functions and operators will be described in this appendix:

KEY <numeric expression>,<string expression>

In this case, KEY is what we call a **keyword**. That just means a special word that BASIC understands. Keywords are always written in CAPITALS, so that you can easily recognise them.

The < and > signs mean that whatever is between them must be there for the command, function or operator to work. In this case, it tells us that we need to have a **numeric expression** in one place, and a **string expression** in another place. We'll explain what there are a bit more in a few moments.

You might also see square brackets around something, for example, [**numeric expression**]. This means that whatever appears between the square brackets is optional, that is, you can include it if you need to, but that the command, function or operator will work just fine without it. For example, the CIRCLE command has an optional numeric argument to indicate if the circle should be filled when being drawn.

The comma, and some other symbols and punctuation marks just represent themselves. In this case, it means that there must be a comma between the **numeric expression** and the **string expression**. This is what we call syntax: If you miss something out, or put the wrong thing in the wrong place, it is called a syntax error, and the computer will tell you if you have a syntax error by giving a **?SYNTAX ERROR** message.

There is nothing to worry about getting an error from the computer. Instead, it is just the computer's way of telling you that something isn't quite right, so that you can more easily find and fix the problem. Error messages like this can't hurt the computer or damage your program, so there is nothing to worry about. For example, if we accidentally left the comma out, or replaced it with a full-stop, the computer will respond with a syntax error, like this:

```
KEY 8"FISH"
```

```
?SYNTAX ERROR
```

```
KEY 8."FISH"
```

```
?SYNTAX ERROR
```

It is very common for commands, functions and operators to use one or more “**expression**”. An expression is just a fancy name for something that has a value. This could be a string, such as “HELLO”, or a number, like 23.7, or it could be a calculation, that might include one or more functions or operators, such as `LEN("HELLO") * (3 XOR 7)`. Generally speaking, expressions can result in either a string or numeric result. In this case we call the expressions either string expressions or numeric expressions. For example, “HELLO” is a **string expression**, while 23.7 is a **numeric expression**.

It is important to use the correct type of expression when writing your programs. If you accidentally use the wrong type, the computer will give you a **?TYPE MISMATCH ERROR**, to say that the type of expression you gave doesn’t match what it expected, that is, there is a mismatch between the type of expression it expected, and the one you gave. For example, we will get a **?TYPE MISMATCH ERROR** if we type the following command, because “POTATO” is a string expression instead of a numeric expression:

```
KEY "POTATO","SOUP"
```

You can try typing this into the computer yourself now, if you like.

COMMANDS, FUNCTIONS AND OPERATORS

Commands are statements that you can use directly from the **READY.** prompt, or from within a program, for example:

```
PRINT "HELLO"
```

```
HELLO
```

```
10 PRINT "HELLO"
```

```
RUN
```

```
HELLO
```

ABS

Token: \$B6

Format: **ABS(x)**

Usage: The numeric function **ABS(x)** returns the absolute value of the numeric argument **x**.

x = numeric argument (integer or real expression).

Remarks: The result is of real type.

Example: Using **ABS**

```
PRINT ABS(-123)
123
PRINT ABS(4.5)
4.5
PRINT ABS(-4.5)
4.5
```

AND

Token: \$AF

Format: operand **AND** operand

Usage: The boolean **AND** operator performs a bitwise logical AND operation on two 16-bit values. Integer operands are used as they are. Real operands are converted to a signed 16 bit integer. Logical operands are converted to 16 bit integer using \$FFFF, decimal -1 for TRUE and \$0000, decimal 0, for FALSE.

```
0 AND 0  -> 0
0 AND 1  -> 0
1 AND 0  -> 0
1 AND 1  -> 1
```

Remarks: The result is of integer type. If the result is used in a logical context, the value of 0 is regarded as FALSE, all other, nonzero values are regarded as TRUE.

Example: Using **AND**

```
PRINT 1 AND 3
1
PRINT 128 AND 64
0
```

In most cases **AND** is used in **IF** statements.

```
IF (C) = 0 AND C < 256 THEN PRINT "BYTE VALUE"
```

APPEND

Token: \$FE \$0E

Format: APPEND# lfn, filename [,D drive] [,U unit]

Usage: Opens an existing sequential file of type SEQ or USR for writing and positions the write pointer at the end of the file.

lfn = logical file number

1 <= lfn <= 127: line terminator is CR

128 <= lfn <= 255: line terminator is CR LF

filename is either a quoted string, e.g. "data" or a string expression in parentheses, e.g. (FN\$)

drive = drive # in dual drive disk units.

The drive # defaults to 0 and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to 8.

Remarks: APPEND# functions similar to the OPEN# command, except that if the file already exists, the existing content of the file will be retained, and any PRINT# commands made to the open file will cause the file to grow longer.

Example: Open file in append mode:

```
APPEND#5,"DATA",U9  
APPEND#138,(DD$),U(UNX)  
APPEND#3,"USER FILE,U"  
APPEND#2,"DATA BASE"
```

ASC

Token: \$C6

Format: **ASC**(string)

Usage: Takes the first character of the string argument and returns its numeric code value. The name is apparently chosen to be a mnemonic to ASCII, but the returned value is in fact the so called PETSCII code.

Remarks: **ASC** returns a zero for an empty string, which behaviour is different to BASIC 2, where **ASC("")** gave an error. The inverse function to **ASC** is **CHR\$**.

Example: Using **ASC**

```
PRINT ASC("NEGA")
77
PRINT ASC("")
0
```

ATN

Token: \$C1

Format: **ATN**(numeric expression)

Usage: Returns the arc tangent of the argument. The result is in the range $(-\pi/2 \text{ to } \pi/2)$

Remarks: A multiplication of the result with $180/\pi$ converts the value to the unit "degrees". **ATN** is the inverse function to **TAN**.

Example: Using **ATN**

```
PRINT ATN(0.5)
.463647609
PRINT ATN(0.5) * 180 / π
26.5650512
```

AUTO

Token: \$DC

Format: **AUTO [step]**

Usage: Enables faster typing of BASIC programs. After submitting a new program line to the BASIC editor with the RETURN key, the AUTO function generates a new BASIC line number for the entry of the next line. The new number is computed by adding **step** to the current line number.

step = line number increment

Typing **AUTO** with no argument switches this function off.

Example: Using **AUTO**

```
AUTO 10 : USE AUTO WITH INCREMENT 10
AUTO    : SWITCH AUTO OFF
```

BACKGROUND

Token: \$FE \$3B

Format: **BACKGROUND** colour

Usage: Sets the background colour of the screen to the argument, which must be in the range 1 to 16. (See colour table).

Colours: **Index and RGB values of colour pallette**

index	red	green	blue	colour
1	0	0	0	black
2	15	15	15	white
3	15	0	0	red
4	0	15	15	cyan
5	15	0	15	magenta
6	0	15	0	green
7	0	0	15	blue
8	15	15	0	yellow
9	15	6	0	orange
10	10	4	0	brown
11	15	7	7	pink
12	5	5	5	dark grey
13	8	8	8	medium grey
14	9	15	9	light green
15	9	9	15	light blue
16	11	11	11	light grey

Example: Using **BACKGROUND**

```
BACKGROUND 4 : REM SELECT BACKGROUND COLOUR CYAN
```

BACKUP

Token: \$F6

Format: **BACKUP D source TO D target [,U unit]**

Usage: Used on dual drive disk units only (e.g. 4040, 8050, 8250). The backup is done by the disk unit internally.

source = drive # of source disk (0 or 1).

target = drive # of target disk (0 or 1).

Remarks: The target disk is formatted and a identical copy of the source disk is written.

This command cannot be used for unit to unit copies.

Example: Using **BACKUP**

```
BACKUP D0 TO D1 : REM COPY DISK DRIVE 0 -> DRIVE 1 UNIT 8  
BACKUP D1 TO D0, U9: REM COPY DISK DRIVE 0 -> DRIVE 1 UNIT 9
```

BANK

Token: \$FE \$02

Format: **BANK** banknumber

Usage: Selects the memory configuration for BASIC commands, that use 16-bit addresses. These are LOAD, SAVE, PEEK, POKE, WAIT and SYS. See system memory map for details.

Remarks: A value > 127 selects memory mapped I/O. The default value for the bank number is 128.

Example: Using **BANK**

```
BANK 1 :REM SELECT MEMORY CONFIGURATION 1
```

BEGIN

Token: \$FE \$18

Format: **BEGIN ... BEND**

Usage: The **BEGIN** and **BEND** keywords act like a pair of brackets around a compound statement to be executed after a **THEN** or **ELSE** keyword. This overcomes the single line limitation of the standard **IF ... THEN ... ELSE** clause.

Remarks: Do not jump with **GOTO** or **GOSUB** into a compound statement. It may lead to unexpected results.

Example: Using **BEGIN** and **BEND**

```
10 GET A$  
20 IF A$>="A" AND A$<="Z" THEN BEGIN  
30 PW$=PW$+A$  
40 IF LEN(PW$)>? THEN 90  
50 BEND :REM IGNORE ALL EXCEPT (A-Z)  
60 IF A$<>CHR$(13) GOTO 10  
90 PRINT "PW=";PW$
```

BEND

Token: \$FE \$19

Format: **BEGIN ... BEND**

Usage: The **BEGIN** and **BEND** keywords act like a pair of brackets around a compound statement to be executed after a **THEN** or **ELSE** keyword. This overcomes the single line limitation of the standard **IF ... THEN ... ELSE** clause.

Remarks: The example below shows a quirk in the implementation of the compound statement. If the condition evaluates to **FALSE**, execution does not resume right after **BEND** as it should, but at the beginning of next line. Test this behaviour with the following program:

Example: Using **BEGIN** and **BEND**

```
10 IF Z > 1 THEN BEGIN:A$="ONE"  
20 B$="TWO"  
30 PRINT A$;" ";B$;:BEND:PRINT " QUIRK"  
40 REM EXECUTION RESUMES HERE FOR Z <= 1
```

BLOAD

Token: \$FE \$11

Format: **BLOAD filename [,B bank] [,P address] [,D drive] [,U unit]**

Usage: "Binary LOAD" loads a file of type PRG into RAM at address P and bank B.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FNS\$**)

bank specifies the RAM bank to be used. If not specified the current bank, as set with the last **BANK** statement, will be used.

address can be used to overrule the load address, that is stored in the first two bytes of the PRG file.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: If the loading process tries to load beyond the address \$FFFF, an 'OUT OF MEMORY' error occurs.

Example: Using **BLOAD**

```
BLOAD "ML DATA", B0, U9
BLOAD "SPRITES"
BLOAD "ML ROUTINES", B1, P32768
BLOAD (FNS$), B(B%), P(PA), U(UM%)
```

BOOT

Token: \$FE \$1B

Format: **BOOT filename [,B bank] [,P address] [,D drive] [,U unit]**
BOOT SYS
BOOT

Usage: **BOOT filename** loads a file of type PRG into RAM at address P and bank B and starts executing the code at the load address.

BOOT SYS loads the boot sector from sector 0, track 1 and unit 8 to address \$0400 on bank 0 and performs a JSR \$0400 afterwards (Jump To Subroutine).

The **BOOT** command with no parameter tries to load and execute a file named AUTOBOOT.C65 from the default unit 8. It's short for **RUN "AUTOBOOT.C65"**.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FN\$**)

bank specifies the RAM bank to be used. If not specified the current bank, as set with the last **BANK** statement, will be used.

address can be used to overrule the load address, that is stored in the first two bytes of the PRG file.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: **BOOT SYS** copies the contents of one physical sector (two logical sectors) = 512 bytes from disc to RAM, filling RAM from \$0400 to \$05ff.

Example: Using **BOOT**

```
BOOT SYS  
BOOT (FN$), B(BA%), P(PA), U(UN%)  
BOOT
```

BORDER

Token: \$FE \$3C

Format: **BORDER** colour

Usage: Sets the border colour of the screen to the argument, which must be in the range 1 to 16. (See colour table).

Colours: **Index and RGB values of colour palette**

index	red	green	blue	colour
1	0	0	0	black
2	15	15	15	white
3	15	0	0	red
4	0	15	15	cyan
5	15	0	15	magenta
6	0	15	0	green
7	0	0	15	blue
8	15	15	0	yellow
9	15	6	0	orange
10	10	4	0	brown
11	15	7	7	pink
12	5	5	5	dark grey
13	8	8	8	medium grey
14	9	15	9	light green
15	9	9	15	light blue
16	11	11	11	light grey

Example: Using **BORDER**

```
10 BORDER 5 : REM SELECT BACKGROUND COLOUR MAGENTA
```

BOX

Token: \$E1

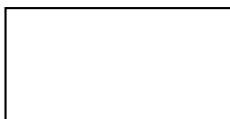
Format: **BOX X0,Y0, X1,Y1, X2,Y2, X3,Y3, SOLID**

Usage: Draws a quadrangle by connecting the coordinate pairs $0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 0$. The quadrangle is drawn using the current drawing context set with SCREEN, PALETTE and PEN. The quadrangle is filled, if the parameter SOLID is 1.

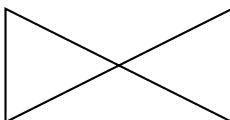
Remarks: A quadrangle is a geometric figure with four sides and four angles. A box is a special form of a quadrangle, with all four angles at 90 degrees. Rhomboids, kites and parallelograms are special forms too. So the name of this command is misleading, because it can be used to draw all kind of quadrangles, not only boxes. It is possible to draw bowtie shapes.

Example: Using **BOX**

```
BOX 0,0, 160,0, 160,80, 0,80
```



```
BOX 0,0, 160,80, 160,0, 0,80
```



```
BOX 0,0, 160,0, 140,80, 20,80
```



BSAVE

Token: \$FE \$10

Format: **BSAVE filename ,P start TO end [,B bank] [,D drive] [,U unit]**

Usage: "Binary SAVE" saves a memory range to a file of type PRG.

filename is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FNS\$)** If the first character of the filename is an at-sign '@' it is interpreted as a "save and replace" operation. It is dangerous to use this replace option on drives 1541 and 1571, because they contain the notorious "save and replace bug" in their DOS.

bank specifies the RAM bank to be used. If not specified the current bank, as set with the last **BANK** statement, will be used.

start is the first address, where the saving begins. It becomes also the load address, that is stored in the first two bytes of the PRG file.

end is the address, where the saving stops. **end-1** is the last address to be used for saving.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The length of the file is **end - start + 2**.

Example: Using **BSAVE**

```
BSAVE "ML DATA", P 32768 TO 33792, B0, U9
BSAVE "SPRITES", P 1536 TO 2858
BSAVE "ML ROUTINES", B1, P(DEC("9000")) TO (DEC("A000"))
BSAVE (FNS$), B(BAX), P(PA) TO (PE), U(UNX)
```

BUMP

Token: \$CE \$03

Format: **b = BUMP(type)**

Usage: Used to detect sprite-sprite (type=1) or sprite-data (type=2) collisions. the return value **b** is a 8-bit mask with one bit per sprite. The bit position corresponds with the sprite number. Each bit set in the return value indicates, that the sprite for this position was involved in a collision since the last call of **BUMP**. Calling **BUMP** resets the collision mask, so you get always a summary of collisions encountered since the last call of **BUMP**.

Remarks: It's possible to detect multiple collisions, but you need to evaluate sprite coordinates then to detect which sprite collided with which one.

Example: Using **BUMP**

```
10 $% = BUMP(1) : REM SPRITE-SPRITE COLLISION
20 IF ($% AND 6) = 6 THEN PRINT "SPRITE 1 & 2 COLLISION"
30 REM ---
40 $% = BUMP(2) : REM SPRITE-DATA COLLISION
50 IF ($% > 0) THEN PRINT "SOME SPRITE HIT DATA REGION"
```

sprite	return	mask
0	1	0000 0001
1	2	0000 0010
2	4	0000 0100
3	8	0000 1000
4	16	0001 0000
5	32	0010 0000
6	64	0100 0000
7	128	1000 0000

BVERIFY

Token: \$FE \$28

Format: **BVERIFY filename [P address] [B bank] [D drive] [U unit]**

Usage: "Binary VERIFY" compares a memory range to a file of type PRG.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FNS\$**)

bank specifies the RAM bank to be used. If not specified the current bank, as set with the last **BANK** statement, will be used.

address is the address, where the comparison begins. If the parameter P is omitted, it is the load address, that is stored in the first two bytes of the PRG file.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: **BVERIFY** can only test for equality. It gives no information about the number or position of different valued bytes. In direct mode the command exits either with the message **OK** or with **VERIFY ERROR**. In program mode a **VERIFY ERROR** either stops execution or enters the **TRAP** error handler, if active.

Example: Using **BVERIFY**

```
BVERIFY "ML DATA", P 32768, B0, U9  
BVERIFY "SPRITES", P 1536  
BVERIFY "ML ROUTINES", B1, P(DEC("9000"))  
BVERIFY (FNS$), B(BA%), P(PA), U(UN%)
```

CATALOG

Token: \$FE \$0C

Format: **CATALOG [filepattern] [,R] [,D drive] [,U unit]**

Usage: Prints a file catalog/directory of the specified disk.

The **R** (Recoverable) parameter includes files in the directory, which are flagged as deleted but are still recoverable.

filepattern is either a quoted string, for example: "**da****" or a string expression in parentheses, e.g. **(DI\$)**

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The command **CATALOG** is a synonym for **DIRECTORY** or **DIR** and produces the same listing. The **filepattern** can be used to filter the listing. The wildcard characters * and ? may be used. Adding a ,**T=** to the pattern string, with **T** specifying a filetype of **P, S, U** or **R** (for **PRG, SEQ, USR, REL**) filters the output to that filetype.

Example: Using **CATALOG**

```
CATALOG
0 "BLACK SMURF      " BS 2A
508 "STORY PHOBOS"   SEQ
27  "C8096"          PRG
25  "C128"           PRG
104 BLOCKS FREE.
```

```
CATALOG "*,T=S"
0 "BLACK SMURF      " BS 2A
508 "STORY PHOBOS"   SEQ
104 BLOCKS FREE.
```

CHANGE

Token: \$FE \$2C

Format: **CHANGE "find" TO "replace" [,from-to]**

Usage: Used in direct mode only. It searches the line range if specified or the whole BASIC program else. At each occurrence of the "find string" the line is listed and the user prompted for an action:
'Y' <RETURN> do the change and find next string
'N' <RETURN> do **not** change and find next string
'*' <RETURN> change this and all following matches
<RETURN> exit command, don't change.

Remarks: Instead of the quote (") each other character may be used as delimiter for the findstring and replacestring. Using the quote as delimiter finds text strings, that are not tokenized and therefore not part of a keyword.

CHANGE "LOOP" TO "OOPS" will not find the BASIC keyword **LOOP**, because the keyword is stored as token and not as text. However **CHANGE &LOOP& TO &OOPS&** will find and replace it (probably spoiling the program).

Example: Using **CHANGE**

```
CHANGE "XX$" TO "UU$", 2000-2700
CHANGE &IN& TO &OUT&
```

CHAR

Token: \$E0

Format: **CHAR column, row, height, width, direction, string [, address of character set]**

Usage: Displays text on a graphic screen. It can be used for all resolutions.

column is the start position of the output in horizontal direction. One column is 8 pixels wide, so a screen width of 320 has a column range 0 -> 39, while a width of 640 has a range of 0 -> 79.

row is the start position of the output in vertical direction. Other than column, its unit is pixel with top row having the value 0.

height is a factor applied to the vertical size of the characters. 1 is normal size (8 pixels) 2 is double size (16 pixels), and so on.

width is a factor applied to the horizontal size of the characters. 1 is normal size (8 pixels) 2 is double size (16 pixels), and so on.

direction controls the printing direction:

- 1: up
- 2: right
- 4: down
- 8: left

The optional **address of character set** can be used to select a character set different from the default character set at \$29800, which is the set with upper/lower characters.

string is a string constant or expression which will be printed.

Remarks: Control characters, for example: cursor movement codes, will be ignored (neither printed nor interpreted).

Example: Using CHAR

```
CHAR 304,196, 1,1,2, "MEGA 65"
```

will print the text "MEGA 65" on the centre of a 640 x 400 graphic screen.

CHR\$

Token: \$C1

Format: CHR\$(**numeric expression**)

Usage: Returns a string of length one character using the argument to insert the character having this value as PETSCII code.

Remarks: The argument range is 0 -> 255, so this function may also be used to insert control codes into strings. Even the NULL character, with code 0, is allowed.

CHR\$ is the inverse function to ASC.

Example: Using CHR\$

```
10 QUOTE$ = CHR$(34)
20 ESCAPE$ = CHR$(27)
30 PRINT QUOTE$;"MEGA 65";QUOTE$ : REM PRINT "MEGA 65"
40 PRINT ESCAPE$;"Q"; : REM CLEAR TO END OF LINE
```

CIRCLE

Token: \$E2

Format: **CIRCLE** **xcentre, ycentre, radius, [,solid]**

Usage: A special case of the **ELLIPSE** command using the same value for horizontal and vertical radius.

xcentre x coordinate of centre in pixels.

ycentre y coordinate of centre in pixels.

radius radius of the circle in pixels.

solid will fill the circle if not zero.

Remarks: The **CIRCLE** command is used to draw circles on screens with an aspect ratio 1:1 (for example: 320 x 200 or 640 x 400). On other resolutions (like: 640 x 200) the shape will degrade to an ellipse.

Example: Using **CIRCLE**

```
10 REM USE A 640 X 400 SCREEN
20 CIRCLE 320,200,100
30 REM DRAW CIRCLE IN THE CENTRE OF THE SCREEN
```

CLOSE

Token: \$A0

Format: **CLOSE channel**

Usage: Closes an input or output channel, that was established before by an **OPEN** command.

channel is a value in the range 0 -> 255.

Remarks: Closing open files before the program stops is very important, especially for output files. This command flushes output buffers and updates directory informations on disks. Failing to **CLOSE** can corrupt files and disks. BASIC does NOT automatically close channels or files when the program stops.

Example: Using **CLOSE**

```
10 OPEN 2,8,2,"TEST$,W"  
20 PRINT#2,"TESTSTRING"  
30 CLOSE 2 : REM OMITTING CLOSE GENERATES A SPLAT FILE
```

CLR

Token: \$9C

Format: **CLR**

Usage: Resets all pointers, that are used for management of BASIC variables, arrays and strings. The runtime stack pointers are reset and the table of open channels is reset. A **RUN** command performs **CLR** automatically.

Remarks: **CLR** should not be used inside loops or subroutines because it destroys the return address. After a **CLR** all variables are unknown and will be initialized at the next usage.

Example: Using **CLR**

```
10 A=5: P$="MEGA 65"
20 CLR
30 PRINT A;P$

0
READY.
```

CMD

Token: \$9D

Format: **CMD channel [,string]**

Usage: Redirects the standard output from screen to the channel. This enables to print listings and directories or other screen outputs. It is also possible to redirect this output to a disk file or a modem.

channel must be opened by the **OPEN** command.

The optional **string** is sent to the channel before the redirection begins and can be used, for example, for printer setup escape sequences.

Remarks: The **CMD** mode is stopped by a **PRINT# channel** or by closing the channel with **CLOSE channel**. It is recommended to use a **PRINT# channel** before closing, to make sure, that the output buffer is flushed.

Example: Using **CMD** to print a program listing:

```
OPEN 4,4
LIST
PRINT#4
CLOSE 4
```

COLLECT

Token: \$F3

Format: **COLLECT [,D drive] [,U unit]**

Usage: Rebuilds the **BAM** (Block Availability Map) deleting splat files and marking unused blocks as free.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: While this command is useful for cleaning the disk from splat files (for example: write files, that weren't properly closed) it is dangerous for disks with boot blocks or random access files. These blocks are not associated with standard disk files and will therefore be marked as free too and may be overwritten by further disk write operations.

Example: Using **COLLECT**

```
COLLECT
COLLECT U9
COLLECT D0, U9
```

COLLISION

Token: \$FE \$17

Format: **COLLISION type [,linenumber]**

Usage: Enables or disables an user programmed interrupt handler. A call without linenumber disables the handler, while a call with linenumber enables it. After the execution of **COLLISION** with linenumber a sprite collision of the same type, as specified in the **COLLISION** call, interrupts the BASIC program and perform a **GOSUB** to **linenumber** which is expected to contain the user code for handling sprite collisions. This handler must give control back with a **RETURN**.

type specifies the collision type for this interrupt handler:

- 1 = sprite - sprite collision
- 2 = sprite - data - collision
- 3 = light pen

linenumber must point to a subroutine which holds code for handling sprite collision and ends with a **RETURN**.

Remarks: It is possible to enable interrupt handler for all types, but only one can execute at any time. A interrupt handler cannot be interrupted by another interrupt handler. Functions like **BUMP**, **RSPPOS** and **LPEN** may be used for evaluation of the sprites which are involved and their positions.

Example: Using **COLLISION**

```
10 COLLISION 1,70 : REM ENABLE
20 SPRITE 1,1 : MOVSPr 1,120, 0 : MOVSPr 1,0#5
30 SPRITE 2,1 : MOVSPr 2,120,100 : MOVSPr 2,180#5
40 FOR I=1 TO 50000:NEXT
50 COLLISION 1 : REM DISABLE
50 END
70 REM SPRITE (-) SPRITE INTERRUPT HANDLER
80 PRINT "BUMP RETURNS";BUMP(1)
90 RETURN: REM RETURN FROM INTERRUPT
```

COLOR

Token: \$E7

Format: COLOR <ON|OFF>

Usage: Enables or disables handling of the character attributes on the screen. If **COLOR** is **ON**, the screen routines take care for both character RAM and attribute RAM. E.g. if the screen is scrolled for text, the attributes are scrolled too, so each character keeps his attribute or colour. If **COLOR** is **OFF**, the attribute or colour RAM is fixed and character movement is only done for screen characters. This speeds up screen handling, if moving characters with different colours is not intended.

Example: COLOR ON – with colour/attribute handling

COLOR OFF – no colour/attribute handling

CONCAT

Token: \$FE \$13

Format: **CONCAT appendfile [D drive] TO targetfile [D drive] [,U unit]**

Usage: The **CONCAT** (concatenation) appends the contents of **appendfile** to the **targetfile**. Afterwards **targetfile** contains the contents of both files, while **appendfile** remains unchanged.

appendfile is either a quoted string, for example: "**data**" or a string expression in parentheses, for example: (**FNS\$**)

targetfile is either a quoted string, for example: "**safe**" or a string expression in parentheses, for example: (**FS\$**)

If the disk unit has dual drives, it is possible to apply the **CONCAT** command to files, which are stored on different disks. In this case, it is necessary to specify the drive# for both files in the command. This is necessary too, if both files are stored on drive#1.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The **CONCAT** commands is executed in the DOS of the disk drive. Both files must exist and no pattern matching is allowed. Only sequential files of type **SEQ** may be concatenated.

Example: Using **CONCAT**

```
CONCAT "NEW DATA" TO "ARCHIVE" ,U9  
CONCAT "ADDRESS",D0 TO "ADDRESS BOOK",D1
```

CONT

Token: \$9A

Format: **CONT**

Usage: Used to resume program execution after a break or stop caused by an **END** or **STOP** statement or by pressing the **STOP KEY**. This is a useful debug tool. The BASIC program may be stopped and variables can be examined and even changed. The **CONT** statement then resumes execution.

Remarks: **CONT** cannot be used, if the program stops due to errors. Also any editing of the program inhibits continuation. Stopping and continuation can spoil the screen output or interfere with input/output operations.

Example: Using **CONT**

```
10 I=I+1:GOTO 10
RUN

BREAK IN 10
READY.
PRINT I
947
CONT
```

COPY

Token: \$F4

Format: COPY source [,D drives] TO target [,D drive] [,U unit]

Usage: Copies the contents of **source** to the **target**. It is used to copy either single files or, by using wildcard characters, multiple files.

source is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FN\$**).

target is either a quoted string, e.g. "**backup**" or a string expression in parentheses, e.g. (**FSS\$**)

If the disk unit has dual drives, it is possible to copy files from disk to disk. In this case, it is necessary to specify the drive# for source and target in the command. This is necessary too, if both files are stored on drive#1.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The **COPY** command is executed in the DOS of the disk drive. It can copy all regular file types (PRG, SEQ, USR, REL). The source file must exist, the target file must not exist. If source and target are on the same disk, the target filename must be different from the source file name.

Example: Using **COPY**

```
COPY "*",D0 TO D1      :REM COPY ALL FILES
COPY "CODES" TO "BACKUP" :REM COPY SINGLE FILE
COPY "*.TXT" TO D1      :REM PATTERN COPY
```

COS

Token: \$BE

Format: **COS**(numeric expression)

Usage: The **COS** function returns the cosine of the argument. The argument is expected in units of **[radians]**. The result is in the range (-1.0 to +1.0)

Remarks: An argument in units of **[degrees]** can be converted to **[radians]** by multiplication with $\pi/180$.

Example: Using **COS**

```
PRINT COS(0.7)
.764842187

X=60:PRINT COS(X * pi / 180)
.500000001
```

CURSOR

Format: **CURSOR [<ON/OFF>] [,column] [,row] [,style]**

Usage: Moves the text cursor to the specified position on the current text screen.

ON or **OFF** displays or hides the cursor.

column and **row** specify the new position.

style defines a solid (1) or flashing (0) cursor.

Example: Using **CURSOR**

```
10 CURSOR ON,1,2,1 :REM SET SOLID CURSOR AT COLUMN 1, ROW 2
```

DATA

- Token:** \$83
- Format:** **DATA** [list of constants]
- Usage:** Used to define constants which can be read by **READ** statements somewhere in the program. All type of constants (integer, real, strings) are allowed, but no expressions. Items are separated by commas. Strings containing commas, colons or spaces must be put in quotes.
- A **RUN** command initializes the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is in the responsibility of the programmer, that the type of the constant and the variable in the **READ** statement match. Empty items with no constant between commas are allowed and will be interpreted as zero for numeric variables and an empty string for string variables.
- The **RESTORE** command may be used to set the data pointer to a specific line for subsequent readings.
- Remarks:** It is good programming style to put large amount of **DATA** statements at the end of the program. Otherwise **GOTO** and **GO-SUB** statements, with target lines lower than the current one, start their search for linenumbers at the beginning of the program and have to skip through **DATA** lines wasting time.

- Example:** Using **DATA**

```
10 READ NA$, VE
20 READ NX:FOR I=2 TO NX:READ GL(I):NEXT I
30 PRINT "PROGRAM:";NA$;"    VERSION:";VE
40 PRINT "N-POINT GAUSS-LEGENDRE FACTORS E1":
50 FOR I=2 TO NX:PRINT I,GL(I):NEXT I
60 STOP
80 DATA "MEGA 65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252
```

DCLEAR

Token: \$FE \$15

Format: **DCLEAR [,D drive] [,U unit]**

Usage: Sends an initialise command to the specified unit and drive.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The DOS inside the disk unit will close all open files, clear all channels, free buffers and reread the BAM. This command should be used together with a **DCLOSE** to make sure, that the computer and the drive agree on the status, otherwise strange side effects may occur.

Example: Using **DCLEAR**

```
DCLOSE    :DCLEAR
DCLOSE U9:DCLEAR U9
DCLOSE U9:DCLEAR D0, U9
```

DCLOSE

Token: \$FE \$0F

Format: **DCLOSE [#channel] [,U unit]**

Usage: Closes a single file or all files for the specified unit.

channel = channel # assigned with the **DOPEN** statement.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

The **DCLOSE** command is used either with a channel argument or a unit number, but never both.

Remarks: It is important to close all open files before the program ends. Otherwise buffers will not be freed and even worse, open write files will be incomplete (splat files) and no more usable.

Example: Using **DCLOSE**

```
DCLOSE#2 :REM CLOSE FILE ASSIGNED TO CHANNEL 2  
DCLOSE U9:REM CLOSE ALL FILES OPEN ON UNIT 9
```

DEC

Token: \$D1

Format: DEC(string expression)

Usage: Returns the decimal value of the argument, that is written as a hex string. The argument range is "0000" to "FFFF" or 0 to 65535 respectively. The argument must have 1-4 hex digits.

Remarks: Allowed digits in uppercase/graphics mode are:
0123456789ABCDEF and in lowercase/uppercase mode:
0123456789abcdef.

Example: Using DEC

```
PRINT DEC("D000")
53248
POKE DEC("600"),255
```

DEF FN

Token: \$96

Format: **DEF FN name(real variable)**

Usage: Defines a single statement user function with one argument of real type returning a real value. The definition must be executed before the function can be used in expressions. The argument is a dummy variable, which will be replaced by the argument in the function usage.

Remarks: The value of the dummy variable will not be changed and the variable may be used in other context without side effects.

Example: Using **DEF FN**

```
10 PD = pi / 180
20 DEF FN CD(X)= COS(X*PD): REM COS FOR DEGREES
30 DEF FN SD(X)= SIN(X*PD): REM SIN FOR DEGREES
40 FOR D=0 TO 360 STEP 90
50 PRINT USING "####";D
60 PRINT USING " ##.##";FNCD(D);
70 PRINT USING " ##.##";FNSD(D)
80 NEXT D
RUN
    0   1.00   0.00
    90  0.00   1.00
    180 -1.00   0.00
    270  0.00  -1.00
    360  1.00   0.00
```

DELETE

Token: \$F7

Format: **DELETE [line range]**
DELETE filename [,D drive] [,U unit] [,R]

Usage: Used either to delete a range of lines from the BASIC program or to delete a disk file.

line range consist of the first and the last line to delete or a single line number. If the first number is omitted, the first BASIC line is assumed. The second number in the range specifier defaults to the last BASIC line.

filename is either a quoted string, for example: "**safe**" or a string expression in parentheses, for example: (**FS\$**)

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

R = Recover a previously deleted file. This will only work, if there were no write operations between deletion and recovery, which may have altered the contents of the file.

Remarks: The **DELETE filename** command works like the **SCRATCH filename** command.

Example: Using **DELETE**

```
DELETE 100      :REM DELETE LINE 100
DELETE 240-350  :REM DELETE ALL LINES FROM 240 TO 350
DELETE 500-      :REM DELETE FROM 500 TO END
DELETE -70      :REM DELETE FROM START TO 70

DELETE "DRM",U9 :REM DELETE FILE DRM ON UNIT 9
```

DIM

Token: \$86

Format: **DIM name(limits) [,name(limits)]...**

Usage: Declares the shape, the bounds and the type of a BASIC array. As a declaration statement it must be executed only once and before any usage of the declared arrays. An array can have one or more dimensions. One dimensional arrays are often called vectors while two or more dimensions define a matrix. The lower bound of a dimension is always zero, while the upper bound is declared. The rules for variable names apply for array names too. There are integer arrays, real arrays and string arrays. It is legal to use the same identifier for scalar variables and array variables. The left parenthesis after the name identifies array names.

Remarks: Integer arrays consume two bytes per element, real arrays five bytes and string arrays three bytes for the string descriptor plus the length of the string.

If an array identifier is used without previous declaration, an implicit declaration of an one dimensional array with limit 10 is performed.

Example: Using **DIM**

```
10 DIM A%(8) :REM ARRAY OF 9 ELEMENTS
20 DIM XX(2,3) :REM ARRAY OF 3x4 = 12 ELEMENTS
30 FOR I=0 TO 8:A%(I)=PEEK(256+I):NEXT
40 FOR I=0 TO 2:FOR J=0 TO 3:READ XX(I,J):NEXT J,I
50 END
60 DATA 1,-2,3,-4,5,-6,7,-8,9,-10,11,-12
```

DIR

Token: \$EE (DIR) \$FE \$29 (ECTORY)

Format: **DIR[ECTORY] [filepattern] [,R] [,D drive] [,U unit]**

Usage: Prints a file directory/catalog of the specified disk.

The **R** (Recoverable) parameter includes files in the directory, which are flagged as deleted but are still recoverable.

filepattern is either a quoted string, for example: "**da***" or a string expression in parentheses, e.g. (**DIS\$**)

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The command **DIR** is a synonym for **CATALOG** or **DIRECTORY** and produces the same listing. The **filepattern** can be used to filter the listing. The wildcard characters * and ? may be used. Adding a **,T=** to the pattern string, with **T** specifying a filetype of **P, S, U** or **R** (for **PRG, SEQ, USR, REL**) filters the output to that filetype.

Example: Using **DIRECTORY**

```
DIRECTORY
0 "BLACK SMURF      " BS 2A
508 "STORY PHOBOS"   SEQ
27  "C8096"          PRG
25  "C128"           PRG
104 BLOCKS FREE.
```

```
DIRECTORY "* ,T=S"
0 "BLACK SMURF      " BS 2A
508 "STORY PHOBOS"   SEQ
104 BLOCKS FREE.
```

DISK

Token: \$FE \$40

Format: **DISK command [,U unit]**

Usage: Sends a command string to the specified disk unit.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

command is a string expression.

Remarks: The command string is interpreted by the disk unit and must be compatible to the used DOS version. Read the disk drive manual for possible commands.

Example: Using **DISK**

```
DISK "I0"    :REM INITIALIZE DISK IN DRIVE 0  
DISK "U0>9" :REM CHANGE UNIT# TO 9
```

DLOAD

Token: \$F0

Format: **DLOAD filename [,D drive] [,U unit]**

Usage: "Disk LOAD" loads a file of type PRG into memory reserved for BASIC program source.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FNS\$**)

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The load address, stored in the first two bytes of the file is ignored. The program is loaded into the BASIC memory. This enables loading of BASIC programs, that were saved on other computers with different memory configurations. After loading the program is relinked and ready to run or edit. It is possible to use DLOAD in a running program (Called overlay or chaining). Then the new loaded program replaces the current one and the execution starts automatically on the first line of the new program. Variables, arrays and strings from the current run are preserved and can be used by the new loaded program.

Example: Using **DLOAD**

```
DLOAD "APOCALYPSE"  
DLOAD "MEGA TOOLS",U9  
DLOAD (FNS$),U(UN%)
```

DMA

Token: \$FE \$1F

Format: **DMA command [,length, source address, source bank, target address, target bank, sub]**

Usage: The **DMA** ("Direct Memory Access") command is the fastest method to manipulate memory areas using the DMA controller.

command 0 = copy, 1 = mix, 2 = swap, 3 = fill

length = number of bytes

source address = 16 bit address of read area or fill byte

source bank = bank number for source

target = 16 bit address of write area

target bank = bank number for target

sub = sub command

Remarks: The **DMA** controller has access to the whole 16 MB address range organized in 256 banks of 64 K.

Example: Using **DMA**

```
DMA 3, 2000, 32,0, 2048,0 :REM FILL SCREEN WITH BLANKS  
DMA 0, 2000, 2048,0, 16384,1 :REM COPY SCREEN TO $14000
```

DMODE

Token: \$FE \$35

Format: **DMODE jam,complement,inverse,stencil,style,thick**

Usage: "Display MODE" sets several parameter of the graphical context for drawing commands.

jam	0 - 1
complement	0 - 1
inverse	0 - 1
stencil	0 - 1
style	0 - 3
thick	1 - 8

DO

Token: \$EB

Format: **DO** ... **LOOP**

DO [<**UNTIL** | **WHILE**> <logical expr.>]

... statements [**EXIT**]

LOOP [<**UNTIL** | **WHILE**> <logical expr.>]

Usage: The **DO** and **LOOP** keywords define the start and end of the most versatile BASIC loop. Using **DO** and **LOOP** alone, without any modifiers creates an infinite loop, that can be left by the **EXIT** statement only. The loop can be controlled by adding an **UNTIL** or a **WHILE** statement after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Example: Using **DO** and **LOOP**

```
10 PW$="" : DO  
20 GET A$: PW$=PW$+A$  
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)  
  
10 DO : REM WAIT FOR USER DECISION  
20 GET A$  
30 LOOP UNTIL A$='Y' OR A$='N' OR A$='y' OR A$='n'  
  
10 DO WHILE ABS(EPS) > 0.001  
20 GOSUB 2000 : REM ITERATION SUBROUTINE  
30 LOOP  
  
10 I%=0 : REM INTEGER LOOP 1 -> 100  
20 DO I%=I%+1  
30 LOOP WHILE I% < 101
```

DOPEN

Token: \$FE \$0D

Format: **DOPEN# lfn, filename [,L[reclen]] [,W] [,D drive] [,U unit]**

Usage: Opens a file for reading, writing or modifying.

lfn = logical file number

1 <= lfn <= 127: line terminator is CR

128 <= lfn <= 255: line terminator is CR LF

L indicates, that the file is a relative file, which is opened for read/write and random access. The reclength is mandatory for creating realative files. For existing relative files, the reclen is used as a safety check, if given.

W opens a file for write access. The file must not exist.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FNS**)

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: DOPEN# may be used to open all file types. The sequential file type **SEQ** is default. The relative file type **REL** is chosen by using the **L** parameter. Other file types must be specified in the filename, e.g. by adding ",P" to the filename for program files or ",U" for USR files.

The usage of the "save-and-replace" character '@' at the beginning of the filename is not recommended, because many Commodore disk drives have a bug, that can cause data loss when using this feature.

Example: Using **DOPEN**

```
DOPEN#5,"DATA",U9
DOPEN#130,(DD$),U(UNX)
DOPEN#3,"USER FILE",U"
DOPEN#2,"DATA BASE",L240
```

```
OPENH#4,"MYPROG.P" : REM OPEN PRG FILE
```

DPAT

Token: \$FE \$36

Format: DPAT **type [,number, pattern, ...]**

Usage: "Drawing PATtern" sets pattern of the graphical context for drawing commands.

type	0 - 63
number	1 - 4
pattern	0 - 255

DSAVE

Token: \$EF

Format: **DSAVE filename [,D drive] [,U unit]**

Usage: "Disk SAVE" saves a BASIC program to a file of type PRG.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FN\$**) The maximum length of the filename is 16 characters. If the first character of the filename is an at-sign '@' it is interpreted as a "save and replace" operation. It is dangerous to use this replace option on drives 1541 and 1571, because they contain the notorious "save and replace bug" in their DOS.

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The **DVERIFY** can be used after **DSAVE** to check, if the saved program on disk is identical to the program in memory.

Example: Using **DSAVE**

```
DSAVE "ADVENTURE"  
DSAVE "ZORK-I", U9  
DSAVE "DUNGEON", D1, U10
```

DVERIFY

Token: \$FE \$14

Format: **DVERIFY filename [,D drive] [,U unit]**

Usage: "Disk VERIFY" compares a BASIC program in memory with a disk file of type PRG.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. **(FNS\$)**

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: **DVERIFY** can only test for equality. It gives no information about the number or position of different valued bytes. The command exits either with the message **OK** or with **VERIFY ERROR**.

Example: Using **DVERIFY**

```
DVERIFY "ADVENTURE"  
DVERIFY "ZORK-I",U9  
DVERIFY "DUNGEON",D1,U10
```

EL

Format: **EL** is a reserved system variable

Usage: **EL** has the value of the line, where the latest BASIC error occurred or the value -1 if there was no error.

This variable is typically used in a TRAP routine, where the error line is taken from **EL**.

Example: Using **EL**

```
10 TRAP 100  
  
100 IF ER>0 AND ER<42 THEN PRINT ERR$(ER); " ERROR"  
110 PRINT " IN LINE";EL  
120 RESUME
```

ELLIPSE

Token: \$FE \$30

Format: **ELLIPSE** **xcentre, ycentre, xradius, yradius, [,solid]**

Usage: As the name says, it draws an ellipse.

xcentre x coordinate of centre in pixels.

ycentre y coordinate of centre in pixels.

xradius x radius of the ellipse in pixels.

yradius y radius of the ellipse in pixels.

solid will fill the ellipse if not zero.

Remarks: The **ELLIPSE** command is used to draw ellipses on screens with various resolutions. It can also be used to draw circles.

Example: Using **ELLIPSE**

```
10 REM USE A 640 X 400 SCREEN
20 ELLIPSE 320,200,100,150
30 REM DRAW ELLIPSE IN THE CENTRE
```

ELSE

Token: \$D5

Format: **IF expression THEN true clause ELSE false clause**

Usage: The **ELSE** keyword is part of an **IF** statement.

expression is a logical or numeric expression. A numerical expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non zero value.

true clause are one or more statements starting directly after **THEN** on the same line. A linenumber after **THEN** performs a **GOTO** to that line.

false clause are one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line.

Remarks: The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** or **false clause** may be expanded to several lines using a compound statement bracketed with the keywords **BEGIN** and **BEND**.

Example: Using **ELSE**

```
10 IF V < 0 THEN PRINT RED$;:ELSE PRINT BLACK$;
20 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
30 INPUT "END PROGRAM:(Y/N)";A$
40 IF A$="Y" THEN END
50 IF A$="N" THEN 10:ELSE 30
```

END

Token: \$80

Format: **END**

Usage: Ends the execution of the BASIC program. The **READY.** prompt appears and the computer goes into direct mode waiting for keyboard input.

Remarks: **END** does **not** clear channels or close files. Also variable definitions are still valid after **END**. The program may be continued with the **CONT** statement. After executing the very last line of the program **END** is executed automatically.

Example: Using **END**

```
10 IF V < 0 THEN END : REM NEGATIVE NUMBERS END THE PROGRAM  
20 PRINT V
```

ENVELOPE

Token: \$FE \$0A

Format: ENVELOPE n, [attack,decay,sustain,release, waveform,pw]

Usage: Used to define the parameters for the synthesis of a musical instrument.

n = envelope slot (0 -> 9)

attack = attack rate (0 -> 15)

decay = decay rate (0 -> 15)

sustain = sustain rate (0 -> 15)

release = release rate (0 -> 15)

waveform = (0:triangle, 1:sawtooth, 2:square/pulse, 3:noise, 4:ring modulation)

pw = pulse width (0 -> 4095) for waveform = pulse.

There are 10 slots for storing tunes, preset with following values:

n	A	D	S	R	WF	PW	Instrument
0	0	9	0	0	2	1536	piano
1	12	0	12	0	1		accordion
2	0	0	15	0	0		calliope
3	0	5	5	0	3		drum
4	9	4	4	0	0		flute
5	0	9	2	1	1		guitar
6	0	9	0	0	2	512	harpsichord
7	0	9	9	0	2	2048	organ
8	8	9	4	1	2	512	trumpet
9	0	9	0	0	0		xylophone

Example: Using ENVELOPE

```
10 ENVELOPE 9,10,5,10,5,2,4000:PLAY "T9"  
20 VOL 8  
30 TEMPO 100  
40 PLAY "C D E F G A B"  
50 PLAY "U5 V1 C D E F G A B"
```

ERASE

Token: \$FE \$2A

Format: ERASE **filename** [,D drive] [,U unit] [,R]

Usage: Used to erase a disk file.

filename is either a quoted string, e.g. “**data**” or a string expression in parentheses, e.g. (**FNS\$**)

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

R = Recover a previously erased file. This will only work, if there were no write operations between erosion and recovery, which may have altered the contents of the file.

Remarks: The **ERASE filename** command works like the **SCRATCH filename** command.

The success and the number of erased files can be examined by printing or using the system variable DS\$. The second last number, which reports the track number in case of an disk error, now reports the number of successfully erased files.

Example: Using **ERASE**

```
SCRATCH "DRM",U9 :REM SCRATCH FILE DRM ON UNIT 9
PRINT DS$
01, FILES SCRATCHED,01,00
SCRATCH "OLD*"   :REM SCRATCH ALL FILES BEGINNING WTH "OLD"
PRINT DS$
01, FILES SCRATCHED,04,00
```

ER

Format: **ER** is a reserved system variable

Usage: **ER** has the value of the latest BASIC error occurred or the value -1 if there was no error.

This variable is typically used in a TRAP routine, where the error number is taken from **ER**.

Example: Using **ER**

```
10 TRAP 100  
  
100 IF ER>0 AND ER<42 THEN PRINT ERR$(ER); " ERROR"  
110 RESUME
```

ERR\$

Token: \$D3

Format: **ERR\$(number)**

Usage: Used to convert an error number to an error string.

number is a BASIC error number (1 -> 41).

This function is typically used in a TRAP routine, where the error number is taken from the reserved variable **ER**.

Remarks: Arguments out of range (1 -> 41) will produce an 'ILLEGAL QUANTITY' error.

Example: Using **ERR\$**

```
10 TRAP 100  
  
100 IF ER>0 AND ER<42 THEN PRINT ERR$(ER);" ERROR"  
110 RESUME
```

EXIT

Token: \$FD

Format: EXIT

Usage: Exits the current DO .. LOOP and continues execution at the first statement after the next LOOP statement.

Remarks: In nested loops EXIT exits only one loop continuing executing in the next outer loop if there is one.

Example: Using EXIT

```
10 DO
20 INPUT "ENTER YOUR AGE";AGE%
30 IF AGE% < 18 THEN EXIT
40 INPUT "ENTER YOUR CREDIT CARD #";CR$
50 LOOP UNTIL LEN(CR$) = 12
60 IF AGE% >= 18 THEN GOSUB 1000:REM VALIDATE CREDIT CARD
70 IF AGE% < 18 THEN PRINT "TOO YOUNG":END
```

EXP

Token: \$BD

Format: EXP(numeric expression)

Usage: The EXP (EXPonential function) computes the value of the mathematical constant Euler's number **e = 2.71828183** raised to the power of the argument.

Remarks: An argument greater than 88 produces an OVERFLOW ERROR:

Example: Using EXP

```
PRINT EXP(1)
2.71828183

PRINT EXP(0)
1

PRINT EXP(LOG(2))
2
```

FAST

Token: \$FE \$25

Format: **FAST**

Usage: Sets the system speed to maximum (3.58 MHz). The system default is **FAST**. However after using **SLOW** for access to slow devices, **FAST** can be used to return to fast mode.

Example: Using **FAST**

```
10 SLOW  
20 GOSUB 1000:REM DO SOME SLOW I/O  
30 FAST
```

FILTER

Token: \$FE \$03

Format: **FILTER [freq, lp, bp, hp, res]**

Usage: Sets the parameters for soundfilter.

freq = filter cut off frequency (0 -> 2047)

lp = low pass filter (0:off, 1:on)

bp = band pass filter (0:off, 1:on)

hp = high pass filter (0:off, 1:on)

resonance = resonance (0 -> 15)

Remarks: Missing parameter keep their current value. The effective filter is the sum of all filter settings. This enables band reject and notch effects.

Example: Using **FILTER**

```
FILTER 1023,1,0,0,10 :REM LOW PASS  
FILTER 1023,0,1,0,10 :REM BAND PASS  
FILTER 1023,0,0,1,10 :REM HIGH PASS
```

FIND

Token: \$FE \$2B

Format: **FIND "string" [,from-to]**

Usage: **FIND** is an editor command and can be used in direct mode only. It searches the line range (if specified) or the whole BASIC program else. At each occurrence of the "find string" the line is listed with the string highlighted. The <NO-SCROLL> key can be used to pause the output.

Remarks: Instead of the quote ("") each other character may be used as delimiter for the find string. Using the quote as delimiter finds text strings, that are not tokenized and therefore not part of a keyword.

FIND "LOOP" will not find the BASIC keyword **LOOP**, because the keyword is stored as token and not as text. However **FIND &LOOP&** will find it.

Example: Using **FIND**

```
FIND "XX$", 2000-2700
FIND &ER&
```

FN

Token: \$A5

Format: **FN name(numeric expression)**

Usage: The **FN** functions are user defined functions, that accept a numeric expression as argument and return a real value. They must be defined with **DEF FN** before the first usage.

Example: Using **FN**

```
10 PD = pi / 180
20 DEF FN CD(X)= COS(X*PD): REM COS FOR DEGREES
30 DEF FN SD(X)= SIN(X*PD): REM SIN FOR DEGREES
40 FOR D=0 TO 360 STEP 90
50 PRINT USING "####";D
60 PRINT USING " ##.##";FNCD(D);
70 PRINT USING " ##.##";FNSD(D)
80 NEXT D
RUN
    0   1.00   0.00
    90  0.00   1.00
    180 -1.00   0.00
    270  0.00  -1.00
    360  1.00   0.00
```

FOR

Token: \$81

Format: **FOR index=start TO end [STEP step] ... NEXT [index]**

Usage: The **FOR** statement starts the definition of a BASIC loop with an index variable.

The **index** variable may be incremented or decremented by a constant value on each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

The **start** value is used to initialize the index.

The **end** value is used at the end of the loop and controls, whether the next iteration will be started or the loop exited.

The **step** value defines the change applied to the index variable at the end of the loop. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: For positive increments **end** must be greater or equal than **start**, for negative increments **end** must be less or equal than **start**.

It is bad programming style to change the value of the index variable inside the loop or to jump into or out of the loop body with **GOTO**.

Example: Using **FOR**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * π / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D

10 DIM M(20,20)
20 FOR I=0 TO 20
30 FOR J=I TO 20
40 M(I,J) = I + 100 * J
50 NEXT J,I
```

FOREGROUND

Token: \$FE \$39

Format: FOREGROUND colour

Usage: Sets the foreground colour (text colour) of the screen to the argument, which must be in the range 1 to 16. (See colour table).

Example: FOREGROUND 8 - select foreground colour yellow.

Colours: Index and RGB values of colour palette

index	red	green	blue	colour
1	0	0	0	black
2	15	15	15	white
3	15	0	0	red
4	0	15	15	cyan
5	15	0	15	magenta
6	0	15	0	green
7	0	0	15	blue
8	15	15	0	yellow
9	15	6	0	orange
10	10	4	0	brown
11	15	7	7	pink
12	5	5	5	dark grey
13	8	8	8	medium grey
14	9	15	9	light green
15	9	9	15	light blue
16	11	11	11	light grey

FRE

Token: \$B8

Format: **FRE(mode)**

Usage: Returns the number of free bytes for modes 0 and 1.

FRE(0) returns the number of free bytes in bank 0, which is used for BASIC program source.

FRE(1) returns the number of free bytes in bank 1, which is the bank for BASIC variables, arrays and strings. A usage of **FRE(1)** also triggers the "garbage collection", a process, that collects used strings at the top of the bank, thereby defragmenting string memory.

FRE(2) returns the number of expansion RAM banks, that are available RAM banks above the standard RAM banks 0 and 1, that are used by BASIC.

Example: Using **FRE**:

```
10 PM = FRE(0)
20 VM = FRE(1)
30 EM = FRE(2)
40 PRINT PM;" FREE FOR PROGRAM"
50 PRINT VM;" FREE FOR VARIABLES"
60 PRINT EM;" EXPANSION RAM BANKS"
```

GET

Token: \$A1

Format: **GET string variable**

Usage: Gets the next character from the keyboard queue. If the queue is empty an empty string is assigned to the variable, otherwise a one character string is created and assigned to the string variable. This command does not wait for keyboard input, so it's useful to check for key presses in regular intervals or loops.

Remarks: It is syntactically OK to use **GET** with a numerical variable, but this is dangerous, because hitting a non numerical key will produce an error message and stop the program. The command **GETKEY** is similar, but waits until a key was hit.

Example: Using **GET**:

```
10 DO: GET A$: LOOP UNTIL A$ <> ""
40 IF A$ = "W" THEN 1000 :REM GO NORTH
50 IF A$ = "A" THEN 2000 :REM GO WEST
60 IF A$ = "S" THEN 3000 :REM GO EAST
70 IF A$ = "Z" THEN 4000 :REM GO SOUTH
80 IF A$ = CHR$(13) THEN 5000 :REM RETURN
90 GOTO 10
```

GET#

- Token:** \$A1 '#
- Format:** **GET# channel, list of string variables**
- Usage:** Reads as many bytes as necessary from the channel argument and assigns strings of length one to each variable in the list. This is useful to read characters or bytes from an input stream one by one.
- Remarks:** All values from 0 to 255 are valid, so this command can also be used to read binary data. A value of 0 generates a string of length 1 containing CHR\$(0) as character value.
- Example:** Using **GET#**:

```
10 OPEN 2,8,0,"$0,P"      :REM OPEN CATALOG
15 IF DS THEN PRINT DS$: STOP :REM CAN'T READ
20 GET#2,D$,D$             :REM DISCARD LOAD ADDRESS
25 DO                      :REM LINE LOOP
30 : GET#2,D$,D$           :REM DISCARD LINE LINK
35 : IF ST THEN EXIT       :REM END-OF-FILE
40 : GET#2,L$,H$           :REM FILE SIZE BYTES
45 : S=ASC(L$) + 256*ASC(H$) :REM FILE SIZE
45 : LINE INPUT#2, F$        :REM FILE NAME
50 : PRINT S;F$            :REM PRINT FILE ENTRY
55 LOOP
60 CLOSE 2
```

GETKEY

Token: \$A1 \$F9 (GET token and KEY token)

Format: **GETKEY string variable**

Usage: Gets the next character from the keyboard queue. If the queue is empty the program waits until a key is hit. Then a one character string is created and assigned to the string variable.

Remarks: It is syntactically OK to use **GETKEY** with a numerical variable, but this is dangerous, because hitting a non numerical key will produce an error message and stop the program.

Example: Using **GETKEY**:

```
10 GETKEY A$ :REM WAIT AND GET CHARACTER
40 IF A$ = "W" THEN 1000 :REM GO NORTH
50 IF A$ = "A" THEN 2000 :REM GO WEST
60 IF A$ = "S" THEN 3000 :REM GO EAST
70 IF A$ = "Z" THEN 4000 :REM GO SOUTH
80 IF A$ = CHR$(13) THEN 5000 :REM RETURN
90 GOTO 10
```

GO64

Token: \$CB \$36 \$34 (GO token and 64)

Format: **GO64**

Usage: Switches the computer to the C64 compatible mode. In direct mode a security prompt "ARE YOU SURE?" is printed, which must be responded with 'Y' to continue.

Example: Using **GO64**:

```
GO64  
ARE YOU SURE?
```

GOSUB

Token: \$8D

Format: **GOSUB** line

Usage: The **GOSUB** (GOto SUBroutine) command continues program execution at the given BASIC line number, saving the current BASIC program counter and line number on the runtime stack. This enables the resume of the execution after the **GOSUB** statement, once a **RETURN** statement in the called subroutine was executed. Calls to subroutines via **GOSUB** may be nested but the end of the subroutine code must always be a **RETURN**. Otherwise a stack overflow may occur.

Remarks: Unlike other programming languages, this BASIC version does not support arguments or local variables for subroutines.

Programs can be optimised by grouping subroutines at the beginning of the program source. The **GOSUB** calls will then have low line numbers with only few digits to decode. Also the subroutines will be found faster, because the search for subroutines starts very often at the start of the program.

Example: Using **GOSUB**:

```
10 GOTO 100 :REM TO MAIN PROGRAM
20 REM *** SUBROUTINE DISK STATUS CHECK ***
30 DD=DS:IF DD THEN PRINT "DISK ERROR";DS$
40 RETURN
50 REM *** SUBROUTINE PROMPT Y/N ***
60 DO:INPUT "CONTINUE (Y/N)";A$
70 LOOP UNTIL A$="Y" OR A$="N"
80 RETURN
90 *** MAIN PROGRAM ***
100 DOPEN#2,"BIG DATA"
110 GOSUB 30: IF DD THEN DCLOSE#2:GOSUB 60:REM ASK
120 IF A$="N" THEN STOP
130 GOTO 100: REM RETRY
```

GOTO

Token: \$89 (GOTO) or \$CB \$A4 (GO TO)

Format: **GOTO line**
GO TO line

Usage: Continues program execution at the given BASIC line number. The **GOTO** command written as a single word executes faster than the **GO TO** command.

Remarks: The new line number will be searched by scanning the BASIC source linearly upwards. If the target line number is higher than the current one, the search starts from the current line upwards. If the target line number is lower, the search starts from the start of the program. Knowing this mechanism it is possible to optimise the runtime by grouping often used targets at the start of the program.

Example: Using **GOTO**:

```
10 GOTO 100 :REM TO MAIN PROGRAM
20 REM *** SUBROUTINE DISK STATUS CHECK ***
30 DD=DS:IF DD THEN PRINT "DISK ERROR";DS$
40 RETURN
50 REM *** SUBROUTINE PROMPT Y/N ***
60 DO:INPUT "CONTINUE (Y/N)";A$
70 LOOP UNTIL A$="Y" OR A$="N"
80 RETURN
90 *** MAIN PROGRAM ***
100 DOPEN#2,"BIG DATA"
110 GOTO 30: IF DD THEN DCLOSE#2:GOTO 60:REM ASK
120 IF A$="N" THEN STOP
130 GOTO 100: REM RETRY
```

GRAPHIC

Token: \$DE

Format: **GRAPHIC CLR**

Usage: Initialises the BASIC graphic system. It clears the graphics memory and screen and sets all parameters of the graphics context to the default values.

Remarks: A second form of the **GRAPHIC** command, which serves as an interface to internal subroutines may be added later.

Example: Using **GRAPHIC**:

```
10 GRAPHIC CLR      :REM INITIALIZE
20 SCREEN DEF 1,1,1,2 :REM 640 X 400 X 2
30 SCREEN SET 1,1      :REM VIEW IT
40 SCNCLR 0          :REM CLEAR SCREEN
50 LINE 50,50,590,350 :REM DRAW LINE
```

HEADER

- Token:** \$F1
- Format:** HEADER **diskname [,lid] [,D drive] [,U unit]**
- Usage:** Used to format or clear a diskette or disk.
- diskname** is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. **(DN\$)** The maximum length of the diskname is 16 characters.
- drive** = drive # in dual drive disk units.
The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.
- unit** = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.
- Remarks:** For new diskettes or disks, which are not already formatted it is absolutely necessary to specify the disk ID with the parameter **lid**. This switches the format command to the full format, which writes sector IDs and erases all contents. This will need some time, because every block on the disk will be written.
If the **lid** parameter is omitted, a quick format will be performed. This is only possible, if the disk is formatted already. A quick format writes a new disk name and clears the block allocation map, marking all blocks as free. The disk ID is not changed, the blocks are not overwritten, so contents may be recovered with the **ERASE R** command.
- Example:** Using HEADER

```
HEADER "ADVENTURE",IBS  
HEADER "ZORK-I",U9  
HEADER "DUNGEON",D1,U10
```

HELP

Token: \$EA

Format: **HELP**

Usage: When the BASIC program stops due to an error, type **HELP** for further information. The interpreted line is listed, with the erroneous statement highlighted or underlined.

Remarks: Displays BASIC errors. For errors in disk I/O one should print the disk status variable **DS** or the disk status string **DS\$**.

Example: Using **HELP**

```
10 A=1.E20
20 B=A+A:C=EXP(A):PRINT A,B,C
RUN

?OVERFLOW ERROR IN 20
READY.
HELP

20 B=A+A:C=EXP(A):PRINT A,B,C
```

HEX\$

Token: \$D2

Format: **HEX\$(numeric expression)**

Usage: Returns a four character string in hexadecimal notation converted from the argument. The argument must be in the range 0 -> 65535 corresponding to the hex numbers 0000 -> FFFF.

Remarks: If real numbers are used as arguments, the fractional part will be cut off, not rounded.

Example: Using **HEX\$**:

```
PRINT HEX$(10),HEX$(100),HEX$(1000.9)  
000A      0064      03E8
```

HIGHLIGHT

Token: \$FE \$3D

Format: HIGHLIGHT colour

Usage: Sets the colour to be used for the "highlight" text attribute. The colour index must be in the range 1 to 16. (See colour table).

Remarks: The highlight text attribute is used to mark text in listings generated by the **HELP FIND CHANGE** commands.

Example: HIGHLIGHT 8 - select highlight colour yellow.

Colours: Index and RGB values of colour palette

index	red	green	blue	colour
1	0	0	0	black
2	15	15	15	white
3	15	0	0	red
4	0	15	15	cyan
5	15	0	15	magenta
6	0	15	0	green
7	0	0	15	blue
8	15	15	0	yellow
9	15	6	0	orange
10	10	4	0	brown
11	15	7	7	pink
12	5	5	5	dark grey
13	8	8	8	medium grey
14	9	15	9	light green
15	9	9	15	light blue
16	11	11	11	light grey

IF

Token: \$8B

Format: **IF expression THEN true clause ELSE false clause**

Usage: Starts a conditional execution statement.

expression is a logical or numeric expression. A numerical expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non zero value.

true clause are one or more statements starting directly after **THEN** on the same line. A linenumber after **THEN** performs a **GOTO** to that line.

false clause are one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line.

Remarks: The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** or **false clause** may be expanded to several lines using a compound statement bracketed with the keywords **BEGIN** and **BEND**.

Example: Using IF

```
10 IF V < 0 THEN PRINT RED$;:ELSE PRINT BLACK$;
20 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
30 INPUT "END PROGRAM:(Y/N)";A$
40 IF A$="Y" THEN END
50 IF A$="N" THEN 10:ELSE 30
```

INPUT

Token: \$85

Format: INPUT [prompt <|;|>] variable list

Usage: Prints an optional prompt string and question mark to the screen, flashes the cursor and waits for user input from the keyboard.

prompt = string expression to be printed as prompt. It may be omitted.

If the separator between prompt and variable list is a comma, the cursor is placed directly after the prompt. If the separator is a semicolon, a question mark and a space is added to the prompt.

variable list = list of one or more variables, that receive the input.

The input will be processed after the user hits RETURN.

Remarks: The user must take care to enter the correct type of input matching variable types. Also the number of input items must match the number of variables. Entering non numeric characters for integer or real variables will produce a TYPE MISMATCH ERROR. Strings for string variables have to be put in quotes if they contain spaces or commas.

Many programs, that need a safe input routine use **LINE INPUT** and use an own parser, in order to avoid program breaks by wrong user input.

Example: Using **INPUT**:

```
10 DIM N$(100),A%(100),S$(100):
20 DO
30 INPUT "NAME, AGE, SEX";NA$,AG%,SE$
40 IF NA$="" THEN 30
50 IF NA$="END" THEN EXIT
60 IF AG% < 18 OR AG% > 100 THEN PRINT "AGE?":GOTO 30
70 IF SE$ <> "M" AND SE$ <> "F" THEN PRINT "SEX?":GOTO 30
80 REM CHECK OK: ENTER INTO ARRAY
90 N$(N)=NA$:A%(N)=AG%:S$(N)=SE$:N=N+1
100 LOOP UNTIL N=100
110 PRINT "RECEIVED";N;" NAMES"
```

INPUT#

Token: \$84

Format: **INPUT# chnannel, variable list**

Usage: Reads a record from an input device, e.g. a disk file or a RS232 device and assigns the read data to the variables in the list.

chnannel = channel number assigned by a **DOPEN** or **OPEN** command.

variable list = list of one or more variables, that receive the input.

The input record must be terminated by a RETURN character and must be not longer than the input buffer (160 characters).

Remarks: The type and number of data in a record must match the variable list. Reading non numeric characters for integer or real variables will produce a FILE DATA ERROR. Strings for string variables have to be put in quotes if they contain spaces or commas.
The command **LINE INPUT#** may be used to read a whole record into a single string variable.

Example: Using **INPUT#**:

```
10 DIM N$(100),A%(100),$(100):
20 DOPEN#2,"DATA"
30 FOR I=0 TO 100
40 INPUT#2,N$(I),A%(I),$(I)
50 IF ST=64 THEN 80:REM END OF FILE
60 IF DS THEN PRINT DS$:GOTO 80:REM DISK ERROR
70 NEXT I
80 DCLOSE#2
110 PRINT "READ";I;" RECORDS"
```

INSTR

Token: \$D4

Format: **INSTR(haystack, needle [,start])**

Usage: Locates the position of the string expression "needle" in the string expression "haystack" and returns the index of the first occurrence or zero, if there is no match.

The string expression **haystack** is searched for the occurrence of the string expression **needle**.

The optional argument **start** is an integer expression, which defines the starting position for the search in **haystack**. If not present it defaults to one.

Remarks: If either string is empty or there is no match the function returns zero.

Example: Using **INSTR**:

```
I = INSTR("ABCDEF","CD")      : REM I = 3
I = INSTR("ABCDEF","XY")      : REM I = 0
I = INSTR("ABCDEF","E",3)      : REM I = 5
I = INSTR("ABCDEF","E",6)      : REM I = 0
I = INSTR(A$+B$,C$)
```

INT

Token: \$B5

Format: **INT(numeric expression)**

Usage: Searches the greatest integer value, that is less or equal to the argument and returns this value as a real number. This function is **NOT** limited to the typical 16-bit integer range (-32768 -> 32767), because it uses real arithmetic. The allowed range is therefore determined by the size of the real mantissas (32-bit) : (-2147483648 -> 2147483647).

Remarks: It is not necessary to use the **INT** function for assigning real values to integer variables, because this conversion will be done implicitly, but then for the 16-bit range.

Example: Using **INT**:

```
X = INT(1.9)      :REM X = 1
X = INT(-3.1)     :REM X = -4
X = INT(100000.5) :REM X = 100000
N% = INT(100000.5) :REM ?ILLEGAL QUANTITY ERROR
```

JOY

Token: \$CF

Format: JOY(**port**)

Usage: Returns the state of the joystick for the selected port (1 or 2). Bit 7 contains the state of the fire button. The stick can be moved in eight directions, which are numbered clockwise starting at the upper position.

	left	centre	right
up	8	1	2
centre	7	0	3
down	6	5	4

Example: Using JOY:

```
10 N = JOY(1)
20 IF N AND 128 THEN PRINT "FIRE! ";
30 REM           N   NE  E   SE  S   SW  W   NW
40 ON N AND 15 GOSUB 100,200,300,400,500,600,700,800
50 GOTO 10
100 PRINT "GO NORTH" :RETURN
200 PRINT "GO NORTHEAST":RETURN
300 PRINT "GO EAST"   :RETURN
400 PRINT "GO SOUTHEAST":RETURN
500 PRINT "GO SOUTH"  :RETURN
600 PRINT "GO SOUTHWEST":RETURN
700 PRINT "GO WEST"   :RETURN
800 PRINT "GO NORTHWEST":RETURN
```

KEY

Token: \$F9

Format: **KEY [ON | OFF | number, string]**

Usage: The function keys can either send their keycode when pressed, or a string assigned to this key. After power up or reset this feature is activated and the keys have default assignments.

KEY OFF: switch off function key strings. The keys will send their character code if pressed.

KEY ON: switch on function key strings. The keys will send assigned strings if pressed.

KEY: list current assignments.

KEY number, string assigns the string to the key with that number.

Default assignments:

key	number	string
F1	1	"GRAPHIC"
F2	2	"DLOAD"+CHR\$(34)
F3	3	"DIRECTORY"+CHR\$(13)
F4	4	"SCNCLR"+CHR\$(13)
F5	5	"DSAVE"+CHR\$(34)
F6	6	"RUN"+CHR\$(13)
F7	7	"LIST"+CHR\$(13)
F8	8	"MONITOR"+CHR\$(13)
HELP	15	"HELP"+CHR\$(13)
RUN	16	"RUN"+CHR\$(34)+"*"+CHR\$(34)+CHR\$(13)

Remarks: The sum of the lengths of all assigned strings must not exceed 240 characters. Special characters like RETURN or QUOTE are entered using their codes with the CHR\$(code) function.

Example: Using **KEY**:

```
KEY ON           :REM ENABLE FUNCTION KEYS
KEY OFF          :REM DISABLE FUNCTION KEYS
KEY              :REM LIST ASSIGNMENTS
KEY 2,"PRINT "+CHR$(14) :REM ASSIGN PRINT PI TO F2
```

LEFT\$

Token: \$C8

Format: LEFT\$(**string**, **n**)

Usage: Returns a string containing the first **n** characters from the argument **string**. If the length of **string** is equal or less than **n**, the result string will be identical to the argument string.

string = a string expression

n = a numeric expression (0 -> 255)

Remarks: Empty strings and zero lengths are legal values.

Example: Using LEFT\$:

```
PRINT LEFT$("MEGA-65",4)
MEGA
```

LEN

Token: \$C3

Format: **LEN(string)**

Usage: Returns the length of the string.

string = a string expression

Remarks: There is no terminating character, like the NULL character in C programs. The length of the string is internally stored in an extra byte of the string descriptor.

Example: Using **LEN**:

```
PRINT LEN("MEGA-65"+CHR$(13))  
8
```

LET

Token: \$88

Format: **LET variable - expression**

Usage: The **LET** statement is obsolete and not needed. Assignment to variables can be done without using **LET**.

Example: Using **LET**:

```
LET A=5 :REM LONGER AND SLOWER  
A=5      :REM SHORTER AND FASTER
```

LINE

Token: \$E5

Format: **LINE** **xbeg,ybeg [,xend,yend]**

Usage: Draws a pixel at (xbeg/ybeg), if only one coordinate pair is given. If both coordinate pairs are defined, a line is drawn on the current graphics screen from the coordinate (xbeg/ybeg) to the coordinate (xend/yend). All currently defined modes and values of the graphic context are used.

Example: Using **LINE**:

```
10 GRAPHIC CLR      :REM INITIALIZE
20 SCREEN DEF 1,1,1,2 :REM 640 X 400 X 2
30 SCREEN SET 1,1     :REM VIEW IT
40 SCNCLR 0          :REM CLEAR SCREEN
50 LINE 50,50,590,350 :REM DRAW LINE
```

LIST

Token: \$9B

Format: LIST [line range]

Usage: Used to list a range of lines from the BASIC program.

line range consist of the first and the last line to list or a single line number. If the first number is omitted, the first BASIC line is assumed. The second number in the range specifier defaults to the last BASIC line.

Remarks: The LIST command's output can be redirected to other devices via the **CMD** command.

Example: Using LIST

```
LIST 100      :REM LIST LINE 100
LIST 240-350  :REM LIST ALL LINES FROM 240 TO 350
LIST 500-    :REM LIST FROM 500 TO END
LIST -70     :REM LIST FROM START TO 70
```

LOAD

Token: \$93

Format: **LOAD filename [,U unit [,flag]]**

Usage: This command is obsolete in BASIC-10, where the commands **DLOAD** and **BLOAD** are better alternatives.

The **LOAD** loads a file of type PRG into RAM bank 0, which is also used for BASIC program source.

filename is either a quoted string, e.g. “**prog**” or a string expression.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

If **flag** has a non zero value, the file is loaded to the address, which is read from the first two bytes of the file. Otherwise it is loaded to the start of BASIC memory and the load address in the file is ignored.

Remarks: This command is implemented in BASIC-10 to keep it backward compatible to BASIC-2.

Example: Using **LOAD**

```
LOAD "APOCALYPSE"  
LOAD "MEGA TOOLS",9  
LOAD "*",8,1
```

LOCATE

Token: \$E6

Format: LOCATE x,y

Usage: Moves the graphical cursor to the specified position on the current graphic screen.

Remarks: The graphical cursor is not visible, it's just the starting point for follow up graphics commands. The current position can be examined with the RDOT function.

Example: Using LOCATE

```
LOCATE 8,16      :REM set cursor to X=8 and Y=16
```

LOG

Token: \$BC

Format: **LOG**(numeric expression)

Usage: Computes the value of the natural logarithm of the argument. The natural logarithm uses Euler's number **e = 2.71828183** as base, not the number 10 which is typically used in log functions on a pocket calculator.

Remarks: The log function with base 10 can be computed by dividing the result by $\log(10)$.

Example: Using **LOG**

```
PRINT LOG(1)
0

PRINT LOG(0)
?ILLEGAL QUANTITY ERROR

PRINT LOG(4)
1.38629436

PRINT LOG(100) / LOG(10)
2
```

LOOP

Token: \$EC

Format: **DO** ... **LOOP**

DO [<**UNTIL** | **WHILE**> <logical expr.>]

. . . statements [**EXIT**]

LOOP [<**UNTIL** | **WHILE**> <logical expr.>]

Usage: The **DO** and **LOOP** keywords define the start and end of the most versatile BASIC loop. Using **DO** and **LOOP** alone, without any modifiers creates an infinite loop, that can be left by the **EXIT** statement only. The loop can be controlled by adding an **UNTIL** or a **WHILE** statement after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Example: Using **DO** and **LOOP**

```
10 PW$="":DO  
20 GET A$:PW$=PW$+A$  
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)  
  
10 DO : REM WAIT FOR USER DECISION  
20 GET A$  
30 LOOP UNTIL A$='Y' OR A$='N' OR A$='y' OR A$='n'  
  
10 DO WHILE ABS(EPS) > 0.001  
20 GOSUB 2000 : REM ITERATION SUBROUTINE  
30 LOOP  
  
10 IX=0 : REM INTEGER LOOP 1 -> 100  
20 DO IX=IX+1  
30 LOOP WHILE IX < 101
```

LPEN

Token: \$CE \$04

Format: **LPEN(coordinate)**

Usage: This function requires the use of a CRT monitor or TV and a light pen. It will not work with a LCD or LED screen. The lightpen must be connected to port 1.

LPEN(0) returns the X position of the lightpen, the range is 60 -> 320.

LPEN(1) returns the Y position of the lightpen, the range is 50 -> 250.

Remarks: The X resolution is two pixels, **LPEN(0)** returns therefore only even numbers. A bright background colour is needed to trigger the lightpen. The **COLLISION** statement may be used to install an interrupt handler.

Example: Using **LPEN**

```
PRINT LPEN(0),LPEN(1)      :REM PRINT LIGHTPEN COORDINATES
```

MID\$

Token: \$CA

Format: **variable\$ = MID\$(string, index, n)**

MID\$(string, index, n) = string expression

Usage: **MID\$** can be used either as a function, which returns a string or as a statement for inserting substrings into an existing string.

string = a string expression

index = start index (0 -> 255)

n = length of substring (0 -> 255)

Remarks: Empty strings and zero lengths are legal values.

Example: Using **MID\$**:

```
10 A$ = "MEGA-65"
20 PRINT MID$(A$,3,4)
30 MID$(A$,5,1) = "+"
40 PRINT A$
RUN
GA-
MEGA+65
```

MOD

Token: \$NN

Format: **MOD(dividend,divisor)**

Usage: The **MOD** function returns the remainder of the division.

Remarks: In other programming languages, like C, this function is implemented as an operator. Here it is used as function.

Example: Using **MOD**:

```
FOR I = 0 TO 8: PRINT MOD(I,4);: NEXT I  
0 1 2 3 0 1 2 3 0
```

MONITOR

Token: \$FA

Format: **MONITOR**

Usage: Calls the machine language monitor program, which is mainly used for debugging.

Remarks: Using the **MONITOR** requires knowledge of the CSG4510 / 6502 / 6510 CPU and the assembler language.

Example: Using **MONITOR**:

```
MONITOR
```

MOUSE

Token: \$FE \$3E

Format: **MOUSE ON [,port [,sprite [,pos]]]**
MOUSE OFF

Usage: Enables the mouse driver and connects the mouse at the specified port with the mouse pointer sprite.

port = mouse port 1, 2 (default) or 3 (both).

sprite = sprite number for mouse pointer (default 0).

pos = initial mouse position (x,y).

The **MOUSE OFF** command disables the mouse driver and frees the associated sprite.

Remarks: The "hot spot" of the mouse pointer is the upper left pixel of the sprite.

Example: Using **MOUSE**:

```
REM LOAD DATA INTO SPRITE #0 BEFORE USING IT
MOUSE ON, 1      :REM ENABLE MOUSE WITH SPRITE #0
MOUSE OFF        :REM DISABLE MOUSE
```

MOVSPR

Token: \$FE \$06

Format: **MOVSPR sprite, x, y**
MOVSPR sprite, <+|->xrel, <+|->yrel
MOVSPR sprite, angle # speed

Usage: **MOVSPR** performs, depending on the argument format, three different tasks:

The first form **MOVSPR sprite, x, y** uses no signs for the arguments and sets the absolute position of the sprite to the screen pixel coordinates **x** and **y**.

The second form **MOVSPR sprite, <+|->xrel, <+|->yrel** uses signs '+' or '-' to indicate a relative displacement to the current position.

The third form **MOVSPR sprite, angle # speed** does not set the position of sprite **n**, but defines motion parameters. The format is recognized by putting a hash sign '#' between the last two arguments.

sprite = sprite number

x = absolute screen coordinate [pixel].

y = absolute screen coordinate [pixel].

xrel = relative screen coordinate [pixel].

yrel = relative screen coordinate [pixel].

angle = direction for sprite movement [degrees]. 0 = up, 90 = right, 180 = down, 270 = left.

speed = speed of movement (0 -> 15).

Remarks: The "hot spot" is the upper left pixel of the sprite.

Example: Using **MOVSPR**:

```
10 SPRITE 1,1 :REM TURN SPRITE 1 ON
20 MOUSPR 1,50,50 :REM SET SPRITE 1 TO (50,50)
30 MOVSPR 1,45#5 :REM MOVE SPRITE 1 WITH SPEED 5 TO UPPER RIGHT
```

NEW

Token: \$A2

Format: **NEW**

NEW RESTORE

Usage: Resets all BASIC parameters to their default values. After **NEW** the maximum RAM is available for program and data storage.

Because **NEW** resets parameters and pointers, but does not physically overwrite the address range of a BASIC program, that was in memory before **NEW**, it is possible to recover the program. If there were no **LOAD** operations or editing after the **NEW** command, the program can be restored with the command **NEW RESTORE**.

Example: Using **NEW**:

```
NEW      :REM RESET BASIC
NEW RESTORE :REM TRY TO RECOVER NEW'ED PROGRAM
```

NEXT

Token: \$82

Format: FOR index-start TO end [STEP step] ... NEXT [index]

Usage: Terminates the definition of a BASIC loop with an index variable.

The **index** variable may be incremented or decremented by a constant value **step** on each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

The **start** value is used to initialize the index.

The **end** value is used at the end of the loop and controls, whether the next iteration will be started or the loop exited.

The **step** value defines the change applied to the index variable at the end of the loop. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: The **index** variable after **NEXT** is optional. If it is missing, the variable for the current loop is assumed. Several consecutive **NEXT** statements may be combined by specifying the indexes in a comma separated list. The statements **NEXT I:NEXT J:NEXT K** and **NEXT I,J,K** are equivalent.

Example: Using **NEXT**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * pi / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D

10 DIM M(20,20)
20 FOR I=0 TO 20
30 FOR J=I TO 20
40 M(I,J) = I + 100 * J
50 NEXT J,I
```

NOT

Token: \$A8

Format: NOT operand

Usage: Performs a bitwise logical NOT operation on a 16 bit value. Integer operands are used as they are. Real operands are converted to a signed 16 bit integer. Logical operands are converted to 16 bit integer using \$FFFF, decimal -1 for TRUE and \$0000, decimal 0, for FALSE.

```
NOT 0  ->  1  
NOT 1  ->  0
```

Remarks: The result is of integer type. If the result is used in a logical context, the value of 0 is regarded as FALSE, all other, nonzero values are regarded as TRUE.

Example: Using NOT

```
PRINT NOT 3  
-4  
PRINT NOT 64  
-65
```

In most cases the **NOT** will be used in **IF** statements.

```
OK = C < 256 AND C >= 0  
IF (NOT OK) THEN PRINT "NOT A BYTE VALUE"
```

OFF

Token: \$FE \$24

Format: keyword **OFF**

Usage: **OFF** is a secondary keyword used in combination with primary keywords like **COLOR**, **KEY**, **MOUSE**.

Remarks: The keyword **OFF** cannot be used on its own.

Example: Using **OFF**

```
COLOR OFF :REM DISABLE SCREEN COLOUR  
KEY OFF   :REM DISABLE FUNCTION KEY STRINGS  
MOUSE OFF :REM DISABLE MOUSE DRIVER
```

ON

Token: \$91

Format: **ON expression GOSUB line list**
ON expression GOTO line list
keyword **ON**

Usage: The **ON** keyword starts either a computed **GOSUB** or **GOTO** statement. Dependent on the value of the expression, the target for the **GOSUB** or **GOTO** is chosen from the table of line addresses at the end of the statement.

As a secondary keyword, **ON** is used in combination with primary keywords like **COLOR, KEY, MOUSE**.

expression is a positive numeric value. Real values are cut to integer.

line list is a comma separated list of valid line numbers.

Remarks: Negative values for **expression** will stop the program with an error message. The **line list** specifies the targets for values of 1,2,3,...

An expression value of zero or a value, that is greater than the number of target lines will do nothing and continue program execution with the next statement.

Example: Using **ON**

```
10 COLOR ON :REM ENABLE SCREEN COLOUR
20 KEY    ON :REM ENABLE FUNCTION KEY STRINGS
30 MOUSE ON :REM ENABLE MOUSE DRIVER
40 N = JOY(1):IF N AND 128 THEN PRINT "FIRE! ";
60 REM           N NE E SE S SW W NW
70 ON N AND 15 GOSUB 100,200,300,400,500,600,700,800
80 GOTO 40
100 PRINT "GO NORTH"   :RETURN
200 PRINT "GO NORTHEAST":RETURN
300 PRINT "GO EAST"    :RETURN
400 PRINT "GO SOUTHEAST":RETURN
500 PRINT "GO SOUTH"   :RETURN
600 PRINT "GO SOUTHWEST":RETURN
700 PRINT "GO WEST"    :RETURN
800 PRINT "GO NORTHWEST":RETURN
```

OPEN

Token: \$9F

Format: **OPEN Ifn, first address [,secondary address [,filename]]**

Usage: Opens an input/output channel for a device.

Ifn = logical file number

1 <= Ifn <= 127: line terminator is CR

128 <= Ifn <= 255: line terminator is CR LF

first address = device number. For IEC devices the unit number is the primary address. Following primary address values are possible:

unit	device
0	Keyboard
1	System default
2	RS232 serial connection
3	Screen
4-7	IEC printer and plotter
8-31	IEC disk drives

The **secondary address** has some special values for IEC disk units, 0:load, 1:save, 15:command channel. The values 2 -> 14 may be used for disk files.

filename is either a quoted string, e.g. "data" or a string expression. The syntax is different to the **DOPEN#** command. The **filename** for **OPEN** includes all file attributes, e.g.: "0:data,s,w".

Remarks: For IEC disk units the usage of **DOPEN#** is recommended.

The usage of the "save-and-replace" character '@' at the beginning of the filename is not recommended, because many Commodore disk drives have a bug, that can cause data loss when using this feature.

Example: Using **OPEN**

```
OPEN 4,4 :REM OPEN PRINTER
CMD 4 :REM REDIRECT STANDARD OUTPUT TO 4
LIST :REM PRINT LISTING ON PRINTER DEVICE 4
OPEN 3,8,3,"0:USER FILE,U"
OPEN 2,9,2,"0:DATA,S,W"
```

OR

Token: \$B0

Format: operand **OR** operand

Usage: Performs a bitwise logical OR operation on two 16-bit values. Integer operands are used as they are. Real operands are converted to a signed 16-bit integer. Logical operands are converted to 16-bit integer using \$FFFF, decimal -1 for TRUE and \$0000, decimal 0, for FALSE.

```
0 OR 0 -> 0
0 OR 1 -> 1
1 OR 0 -> 1
1 OR 1 -> 1
```

Remarks: The result is of integer type. If the result is used in a logical context, the value of 0 is regarded as FALSE, all other, nonzero values are regarded as TRUE.

Example: Using **OR**

```
PRINT 1 OR 3
3
PRINT 128 OR 64
192
```

In most cases the **OR** will be used in **IF** statements.

```
IF (C < 0 OR C > 255) THEN PRINT "NOT A BYTE VALUE"
```

PAINT

Token: \$DF

Format: PAINT x, y, mode [,colour]

Usage: Performs a flood fill of an enclosed graphics area.

x, y is a coordinate pair, which must lie inside the area to be filled.

mode specifies the fill mode.

0: use the colour to fill the area.

1: use the colour of pixel (x,y) to fill the area.

Example: Using PAINT

```
10 GRAPHIC CLR      :REM INITIALIZE
20 SCREEN DEF 1,0,0,2 :REM 320 X 200
30 SCREEN OPEN 1     :REM OPEN
40 SCREEN SET 1,1     :REM MAKE SCREEN ACTIVE
50 LINE 160,0,240,100 :REM 1ST. LINE
60 LINE 240,100,80,100 :REM 2ND. LINE
70 LINE 80,100,160,0   :REM 3RD. LINE
80 PAINT 160,10,0,1    :REM FILL TRIANGLE WITH COLOUR 1
90 GETKEY K$          :REM WAIT FOR KEY
100 SCREEN CLOSE 1    :REM END GRAPHICS
```

PALETTE

Token: \$FE \$34

Format: PALETTE [screen|COLOR], colour, red, green, blue
PALETTE RESTORE

Usage: The **PALETTE** command can be used to change an entry of the system colour palette or the palette of a screen.

PALETTE RESTORE resets the system palette to the default values.

screen = screen number (0 or 1).

COLOR = keyword for changing system palette.

colour = index to palette 0 -> 255.

red = red intensity 0 -> 15.

green = green intensity 0 -> 15.

blue = blue intensity 0 -> 15.

Example: Using **PALETTE**

```
10 GRAPHIC CLR :REM INITIALIZE
20 SCREEN DEF 1,0,0,2 :REM 320 X 200
30 SCREEN OPEN 1 :REM OPEN
40 SCREEN SET 1,1 :REM MAKE SCREEN ACTIVE
50 PALETTE 1,0, 0, 0, 0 :REM 0 = BLACK
60 PALETTE 1,1, 15, 0, 0 :REM 1 = RED
70 PALETTE 1,2, 0, 0,15 :REM 2 = BLUE
80 PALETTE 1,3, 0,15, 0 :REM 3 = GREEN
90 LINE 160,0,240,100 :REM 1ST. LINE
100 LINE 240,100,80,100 :REM 2ND. LINE
110 LINE 80,100,160,0 :REM 3RD. LINE
120 PAINT 160,10,0,2 :REM FILL TRIANGLE WITH BLUE (2)
130 GETKEY K$ :REM WAIT FOR KEY
140 SCREEN CLOSE 1 :REM END GRAPHICS
```

PEEK

Token: \$C2

Format: PEEK(address)

Usage: Returns a byte value read from the 16 bit address and the current memory bank (set by **BANK**).

address = a value 0 -> 65535.

Remarks: Banks 0 -> 127 give access to RAM or ROM banks. Banks > 127 are used to access I/O and SYSTEM like VIC, SID, FDC, etc.

Example: Using PEEK

```
10 BANK 128          :REM SELECT SYSTEM BANK
20 L = PEEK(DEC("02F8"))  :REM USR JUMP TARGET LOW
30 H = PEEK(DEC("02F9"))  :REM USR JUMP TARGET HIGH
40 T = L + 256 * H      :REM 16 BIT JUMP ADDRESS
50 PRINT "USR FUNCTION CALLS ADDRESS";T
```

PEN

Token: \$FE \$33

Format: **PEN pen colour**

Usage: Sets the colour for the graphic pen.

pen = pen number (0 -> 2)

colour = palette index.

Remarks: **PEN** defined colours are used by all following drawing commands.

Example: Using **PEN**

```
10 GRAPHIC CLR          :REM INITIALIZE
20 SCREEN DEF 1,0,0,2    :REM 320 X 200
30 SCREEN OPEN 1         :REM OPEN
40 SCREEN SET 1,1        :REM MAKE SCREEN ACTIVE
50 PALETTE 1,0, 0, 0, 0   :REM 0 = BLACK
60 PALETTE 1,1, 15, 0, 0  :REM 1 = RED
70 PALETTE 1,2, 0, 0,15   :REM 2 = BLUE
80 PALETTE 1,3, 0,15, 0   :REM 3 = GREEN
90 PEN 0,1                :REM PEN 0 = RED
100 LINE 160,0,240,100    :REM DRAW RED LINE
110 PEN 0,2                :REM PEN 0 = BLUE
120 LINE 240,100,80,100    :REM DRAW BLUE LINE
130 PEN 0,3                :REM PEN 0 = GREEN
140 LINE 80,100,160,0      :REM DRAW GREEN LINE
150 GETKEY K$              :REM WAIT FOR KEY
160 SCREEN CLOSE 1         :REM END GRAPHICS
```

PLAY

Token: \$FE \$04

Format: **PLAY string**

Usage: Starts playing a tune with notes and directives embedded in the argument string.

A musical note is a letter (A,B,C,D,E,F,G) which may be preceded by an optional modifier.

Possible modifiers are:

char	effect
#	sharp
\$	flat
.	dotted
H	half note
I	eighth note
M	wait for end
Q	quarter note
R	pause (rest)
S	sixteenth note
W	whole note

Embedded directives consist of a letter followed by a digit:

char	directive	argument range
O	octave	0 - 6
T	tune envelope	0 - 9
U	volume	0 - 9
V	voice	1 - 3
X	filter	0 - 1

The envelope slots may be changed using the **ENVELOPE** statement. The default setting for the envelopes are:

n	A	D	S	R	WF	PW	Instrument
0	0	9	0	0	2	1536	piano
1	12	0	12	0	1		accordion
2	0	0	15	0	0		calliope
3	0	5	5	0	3		drum
4	9	4	4	0	0		flute
5	0	9	2	1	1		guitar
6	0	9	0	0	2	512	harpsichord
7	0	9	9	0	2	2048	organ
8	8	9	4	1	2	512	trumpet
9	0	9	0	0	0		xylophone

Remarks: The **PLAY** statement sets up an interrupt driven routine that starts parsing the string and playing the tune. The execution continues with the next statement with no need waiting for the tune to be finished. However this can be forced, using the 'M' modifier.

Example: Using **PLAY**

```
10 ENVELOPE 9,10,5,10,5,2,4000
20 PLAY "I9"
30 VOL 8
40 TEMPO 100
50 PLAY "C D E F G A B"
60 PLAY "U5 V1 C D E F G A B"
```

POINTER

Token: \$CE \$0A

Format: **POINTER(variable)**

Usage: Returns the current address of a variable or an array element in bank 1. For string variables, it is the address of the string descriptor, not the string itself. The string descriptor consists of the three bytes (length,string address low, string address high).

Remarks: The address values of arrays and their elements change for every new declaration (first usage) of scalar variables.
The addresses of strings (not their descriptors) may change at any time due to "garbage collection" in memory management.

Example: Using **POINTER**

```
10 A$="TEXT": B%=5: DIM C(100) :REM DEFINE SOME VARIABLES
20 PRINT POINTER(A$);POINTER(B%);POINTER(C(20))

1026 1033 1145
```

POKE

Token: \$97

Format: **POKE address, byte [,byte ...]**

Usage: Puts on or more bytes into memory or memory mapped I/O, starting at 16-bit **address**. The current memory bank as set by **BANK** is used.

address = a value 0 -> 65535.

byte = a value 0 -> 255.

Remarks: The address is increased by one for each data byte, so a memory range may be filled with a single command.

Banks > 127 are used to access I/O and SYSTEM like VIC, SID, FDC, etc.

Example: Using **POKE**

```
10 BANK 128 :REM SELECT SYSTEM BANK
20 POKE,DEC("02F8"),0,32 :REM SET USR VECTOR TO $2000
```

POLYGON

Token: \$FE \$2F

Format: **POLYGON** **x, y, xrad, yrad, solid, angle, sides, n**

Usage: Draws a regular **n** sided polygon. The polygon is drawn using the current drawing context set with SCREEN, PALETTE and PEN.

x,y = centre coordinates.

xrad,yrad = radius in x- and y-direction.

solid = fill (1) or outline (0).

angle = start angle.

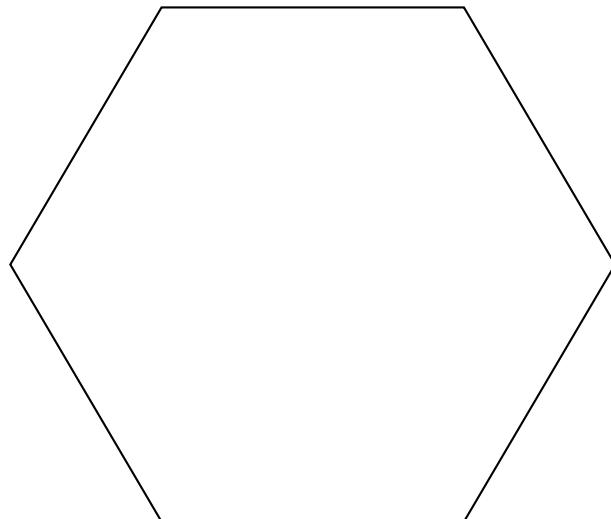
sides = sides to draw ($\leq n$).

n = number of sides or edges.

Remarks: A regular polygon is both isogonal and isotonal, meaning all sides and angles are alike.

Example: Using **POLYGON**

```
POLYGON 320,100,50,50,0,0,6,6
```



POS

Token: \$B9

Format: **POS(dummy)**

Usage: Returns the cursor column relative to the currently used window.

dummy = a numeric value, which is ignored.

Remarks: **POS** gives the column position for the screen cursor. It will not work for redirected output.

Example: Using **POS**

```
10 IF POS(0) > 72 THEN PRINT :REM INSERT RETURN
```

POT

Token: \$CE \$02

Format: POT(**paddle**)

Usage: Returns the position of a paddle.

paddle = paddle number 1 -> 4.

The low byte of the return value is the paddle value with 0 at the clockwise limit and 255 at the counterclockwise limit.

A value > 255 indicates the simultaneous press of the firebutton.

Remarks: Analogue paddles are noisy and inexact. The range may be less than 0 - 255 and there is some jitter in the data.

Example: Using POT

```
10 X = POT(1)      : REM READ PADDLE #1
20 B = X > 255    : REM TRUE (-1) IF FIRE BUTTON IS PRESSED
30 V = X AND 255  : PADDLE #1 VALUE
```

PRINT

Token: \$99

Format: **PRINT arguments**

Usage: Evaluates the argument list and prints the values formatted to the current screen window. Standard formatting is used dependent on the argument type. For user controlled formatting see **PRINT USING**. Following argument types are processed:

numeric : The printout starts with a space for positive and zero values or a minus sign for negative values. Integer values are printed with the necessary number of digits. Real values are printed either in fixed point format with typically 9 digits or in scientific format, if the value is outside the range of 0.01 -> 999999999.

string : The string may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed.

, : A comma acts like a tabulator.

; : A semicolon acts as a separator between arguments of the list. Other than the comma character it does not put in any additional characters. A semicolon at the end of the argument list suppresses the automatic return character.

Remarks: The **SPC** and **TAB** functions may be used in the argument list for positioning. The **CMD** command can be used for redirection.

Example: Using **PRINT**

```
10 FOR I=1 TO 10    : REM START LOOP
20 PRINT I,I*I,SQR(I)
30 NEXT
```

PRINT#

Token: \$98

Format: PRINT# **channel**, **arguments**

Usage: Evaluates the argument list and prints the values formatted to the device assigned to **channel**. Standard formatting is used dependent on the argument type. For user controlled formatting see **PRINT# USING**. Following argument types are processed:

channel : must be opened for output by an **OPEN** or **DOPEN** statement.

numeric : The printout starts with a space for positive and zero values or a minus sign for negative values. Integer values are printed with the necessary number of digits. Real values are printed either in fixed point format with typically 9 digits or in scientific format, if the value is outside the range of 0.01 -> 999999999.

string : The string may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed.

, : A comma acts like a tabulator.

; : A semicolon acts as a separator between arguments of the list. Other than the comma character it does not put in any additional characters. A semicolon at the end of the argument list suppresses the automatic return character.

Remarks: The **SPC** and **TAB** functions are not suitable for devices other than the screen.

Example: Using **PRINT#**

```
10 DOPEN#2,"TABLE",W,U9
20 FOR I=1 TO 10    : REM START LOOP
30 PRINT#2,I,I*I,SQR(I)
40 NEXT
50 DCLOSE#2
```

PRINT USING

Token: \$98 \$FB or \$99 \$FB

Format: PRINT [# channel,] USING, format, arguments

Usage: Parses the format string and evaluates the argument list. The values are printed following the directives of the format string.

channel : must be opened for output by an **OPEN** or **DOPEN** statement. If no channel is specified, the output goes to the screen.

format : A string which defines the rules for formatting.

numeric argument : The '#' characters set the position and width of the output string.

'+' or '-' set the sign option.

'.' sets the position of the decimal point.

^^^^ reserves place for the exponent.

string argument : The string may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed. The number of '#' characters sets the width of the output, a '=' sign centers and a '>' character right justifies the output.

, : A comma acts like a tabulator. ; : A semicolon acts as a separator between arguments of the list. Other than the comma character it does not put in any additional characters. A semicolon at the end of the argument list suppresses the automatic return character.

Remarks: The **SPC** and **TAB** functions may be used for screen output.

Example: Using PRINT# USING

```
10 X = 12.34: A$ = "MEGA 65"
30 PRINT USING "##.##"; X      : REM  "12.34"
40 PRINT USING "##"; X       : REM  " 12"
50 PRINT USING "####"; A$    : REM  "MEGA"
60 PRINT USING "#####"; A$   : REM  "MEGA 65 "
70 PRINT USING "=#####"; A$  : REM  " MEGA 65 "
80 PRINT USING "#####);"; A$ : REM  " MEGA 65"
```

PUDEF

Token: \$DD

Format: PUDEF string

Usage: Redefines up to four special characters, that are used in the **PRINT USING** routine.

string = definition string (max. 4 characters).

1st.: fill character
2nd.: comma separator
3rd.: decimal point
4th.: currency symbol

The system default is " ,.\$"

The new definition string overrides the system default and is often used for localization. A string " ." would change the punctuation to German style.

It is not necessary to redefine all four characters. Any length between 1 and 4 is allowed.

Remarks: PUDEF changes the output of **PRINT USING** only. **PRINT** and **PRINT#** are not affected. The control characters of the format string cannot be changed.

Example: Using PUDEF

```
10 X = 123456.78
20 PUDEF " ,."
30 PRINT USING "###,###.##"; X : REM 123.456,8
```

RCLR

Token: \$CD

Format: **RCLR(**colour source**)**

Usage: Returns the current colour index for the selected colour source.

Colour sources are:

- 0: 40 column background
- 1: graphical foreground
- 2: multicolour mode 1
- 3: multicolour mode 2
- 4: frame colour
- 5: 80 column text
- 6: 80 column background

Example: Using RCLR

```
10 C = RCLR(5) : REM C = colour index of 80 column text
```

RDOT

Token: \$D0

Format: **RDOT(n)**

Usage: Returns information about the graphical cursor.

n = kind of information.

- 0: x-position
- 1: y-position
- 2: colour index

Example: Using **RDOT**

```
10 X = RDOT(0)
20 Y = RDOT(1)
30 C = RDOT(2)
40 PRINT "THE COLOUR INDEX AT (";X;" / ";Y;) IS :";C
```

READ

Token: \$87

Format: **READ(variable list)**

Usage: Reads values from program source into variables.

variable list = any legal variables.

All type of constants (integer, real, strings) can be read, but no expressions. Items are separated by commas. Strings containing commas, colons or spaces must be put in quotes.

A **RUN** command initializes the data pointer to the first item of the first **DATA** statement and advances it for every read item. It is in the responsibility of the programmer, that the type of the constant and the variable in the **READ** statement match. Empty items with no constant between commas are allowed and will be interpreted as zero for numeric variables and an empty string for string variables.

The **RESTORE** command may be used to set the data pointer to a specific line for subsequent readings.

Remarks: It is good programming style to put large amount of **DATA** statements at the end of the program. Otherwise **GOTO** and **GO-SUB** statements, with target lines lower than the current one, start their search for linenumber at the beginning of the program and have to skip through **DATA** lines wasting time.

Example: Using **READ**

```
10 READ NA$, VE
20 READ N%:FOR I=2 TO N%:READ GL(I):NEXT I
30 PRINT "PROGRAM:";NA$;"    VERSION:";VE
40 PRINT "N-POINT GAUSS-LEGENDRE FACTORS E1":
50 FOR I=2 TO N%:PRINT I;GL(I):NEXT I
30 STOP
80 DATA "MEGA 65",1.1
90 DATA 5,0.5120,0.3573,0.2760,0.2252
```

RECORD

Token: \$FE \$12

Format: RECORD#*Ifn*, record, [.byte]

Usage: Positions the read/write pointer of a relative file.

Ifn = logical file number

record = target record (1 -> 65535).

byte = byte position in record.

This command can be used only for files of type **REL**, which are relative files capable of direct access.

The **RECORD** command positions the file pointer to the specified record number. If this record number does not exist and the disk capacity is high enough, the file is expanded to this record count by adding empty records. This is not an error, but the disk status will give the message **RECORD NOT PRESENT**.

Any INPUT# or PRINT# command will then proceed on the selected record position.

Remarks: The original Commodore disk drives all had a bug in their DOS, which could destroy data by using relative files. A recommended workaround was to issue each **RECORD** command twice, before and after the I/O operation.

Example: Using **RECORD**

```
10 REM *** READ FIRST 10 INDEXED RECORDS FROM DATA BASE
15 N = 1000: DIM IX(N)
20 DOPEN#3,"DATA INDEX"
25 FOR I=1 TO N:INPUT#3,IX(I):NEXT
30 DCLOSE#3
35 DOPEN#2,"DATA BASE",L240
40 FOR J=1 TO 10
45 RECORD#2,IX(J)
50 INPUT#2,A$
55 PRINT A$
60 NEXT J
65 DCLOSE#2
```

REM

Token: \$8F

Format: **REM**

Usage: Marks the rest of the line as comment.

All characters after **REM** are never executed but skipped.

Example: Using **REM**

```
10 REM *** PROGRAM TITLE ***
20 N=1000 :REM NUMBER OF ITEMS
30 DIM NA$(N)
```

RENAME

Token: \$F5

Format: **RENAME old TO new [,D drive] [,U unit]**

Usage: Renames a disk file.

old is either a quoted string, e.g. **"data"** or a string expression in parentheses, e.g. **(FN\$)**.

new is either a quoted string, e.g. **"backup"** or a string expression in parentheses, e.g. **(FS\$)**

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: The **RENAME** command is executed in the DOS of the disk drive. It can rename all regular file types (PRG, SEQ, USR, REL). The old file must exist, the new file must not exist. Only single files can be renamed, wild characters like '*' and '?' are not allowed. The file type cannot be changed.

Example: Using **RENAME**

```
RENAME "CODES" TO "BACKUP" :REM RENAME SINGLE FILE
```

RENUMBER

Token: \$F8

Format: **RENUMBER [new, inc, old]**

Usage: Used to renumber all or a range of lines of a BASIC program.

new is the new starting line of the line range to renumber. The default value is 10.

inc is the increment to be used. The default value is 10.

old is the old starting line of the line range to renumber. The default value is the first line.

The **RENUMBER** changes all line numbers in the chosen range and also changes all references from statements like **GOTO**, **GOSUB**, **TRAP**, **RESTORE**, **RUN** etc.

RENUMBER can be executed in direct mode only. If it detects a problem, like memory overflow, unresolved references or line number overflow (greater than 64000) it will stop with an error message and leave the program unchanged.

The command may be called with 0-3 parameters. Unspecified parameters use their default values.

Remarks: The **RENUMBER** command may need several minutes to execute for large programs.

Example: Using **RENUMBER**

```
RENUMBER :REM NUMBERS WILL BE 10,20,30,...  
RENUMBER 100,5 :REM NUMBERS WILL BE 100,105,110,115,...  
RENUMBER 601,1,500 :REM RENUMBER STARTING AT 600 TO 601,602,...
```

RESTORE

Token: \$8C

Format: RESTORE [line]

Usage: Set or reset the internal pointer for **READ** from **DATA** statements.

line is the new position for the pointer to point at. The default is the first program line.

Remarks: The new pointer target **line** needs not to contain **DATA** statements. Every **READ** will automatically advance the pointer to the next **DATA** statement.

Example: Using **RESTORE**

```
10 DATA 3,1,4,1,5,9,2,6
20 DATA "MEGA 65"
30 DATA 2,7,1,8,2,8,9,5
40 FOR I=1 TO 8:READ P:PRINT P:NEXT
50 RESTORE 30
60 FOR I=1 TO 8:READ P:PRINT P:NEXT
70 RESTORE 20
80 READ A$:PRINT A$
```

RESUME

Token: \$D6

Format: **RESUME [line | NEXT]**

Usage: is used inside a **TRAP** routine to resume normal program execution after handling the exception.

line : program execution resumes at the given line number.

NEXT : the keyword NEXT resumes execution at the statement following the statement, that caused the error.

RESUME with no parameters tries to re-execute the statement, that caused the error. The **TRAP** routine should have examined and corrected the variables in this case.

Remarks: **RESUME** cannot be used in direct mode.

Example: Using **RESUME**

```
10 TRAP 100
20 FOR I=1 TO 100
30 PRINT EXP(I)
40 NEXT
50 PRINT "STOPPED FOR I =";I
60 END
100 PRINT ERR$(ER): RESUME 50
```

RETURN

Token: \$8E

Format: RETURN

Usage: Returns control from a subroutine, which was called with **GOSUB** or an event handler, declared with **COLLISION**.

The execution continues at the statement following the **GOSUB** call.

In the case of the **COLLISION** handler, the execution continues at the statement where it left to call the handler.

Example: Using RETURN

```
10 DOPEN#2,"DATA":GOSUB 100
20 FOR I=1 TO 100
30 INPUT#2,A$:GOSUB 100
40 PRINT A$
50 NEXT
60 DCLOSE#2
70 END
100 IF DS THEN PRINT DS$:STOP :REM DISK ERROR
110 RETURN :REM OK
```

RGR

Token: \$CC

Format: **RGR(dummy)**

Usage: Returns information about the graphic mode.

dummy = unused numeric expression.

In text mode **RGR** returns zero. in graphics mode **RGR** returns a non zero value.

Example: Using **RGR**

```
10 M = RGR(0)
20 IF M THEN CHAR 0,0,1,1,2,"TITLE": ELSE PRINT "TITLE"
```

RIGHT\$

Token: \$C9

Format: **RIGHT\$(string, n)**

Usage: Returns a string containing the last **n** characters from the argument **string**. If the length of **string** is equal or less than **n**, the result string will be identical to the argument string.

string = a string expression

n = a numeric expression (0 -> 255)

Remarks: Empty strings and zero lengths are legal values.

Example: Using **RIGHT\$**:

```
PRINT RIGHT$("MEGA-65",2)  
65
```

RMOUSE

Token: \$FE \$3F

Format: **RMOUSE** *xvar, yvar, butvar*

Usage: Reads mouse position and button status.

xvar = numerical variable receiving x-position.

yvar = numerical variable receiving y-position.

butvar = numerical variable receiving button status.
left button sets bit 7, while right button sets bit 0.

value	status
0	no button
1	right button
128	left button
129	both buttons

The command puts a -1 into all variables, if the mouse is not connected or disabled.

Remarks: Two active mice on both ports merge the results.

Example: Using **RMOUSE**:

```
10 MOUSE ON, 1, 1      :REM MOUSE ON PORT 1 WITH SPRITE 1
20 RMOUSE XP, YP, BU   :REM READ MOUSE STATUS
30 IF XP < 0 THEN PRINT "NO MOUSE ON PORT 1":STOP
40 PRINT "MOUSE:";XP;YP;BU
50 MOUSE OFF            :REM DISABLE MOUSE
```

RND

Token: \$BB

Format: **RND(type)**

Usage: Returns a pseudo random number

This is called a "pseudo" random number, because the numbers are not really random, but are derived from another number called "seed" and generate reproducible sequences. The **type** argument determines, which seed is used.

type = 0: use system clock.

type < 0: use the value of **type** as seed.

type > 0: derive value from previous random number.

Remarks: Seeded random number sequences produce the same sequence for identical seeds.

Example: Using **RND**:

```
10 DEF FN1(X) = INT(RND(0)*6)+1 :REM DICE FUNCTION
20 FOR I=1 TO 10                      :REM THROW 10 TIMES
30 PRINT I;FN1(0)                      :REM PRINT DICE POINTS
40 NEXT
```

RREG

Token: \$FE \$09

Format: **RREG areg, xreg, yreg, zreg, sreg**

Usage: Reads the values, that were in the CPU registers after a SYS call, into the specified variables.

areg = variable gets accumulator value.

xreg = variable gets X register value.

yreg = variable gets Y register value.

zreg = variable gets Z register value.

sreg = variable gets status register value.

Remarks: The register values after a SYS call are stored in system memory. This enables the command **RREG** to retrieve these values.

Example: Using **RREG**:

```
10 BANK 128
20 BLOAD "ML PROG",8192
30 SYS 8192
40 RREG A,X,Y,Z,S
50 PRINT "REGISTER:";A;X;Y;Z;S
```

RSPCOLOR

Token: \$CE \$07

Format: RSPCOLOR(n)

Usage: Returns multicolour sprite colours.

n = 1 : get multicolour # 1.

n = 2 : get multicolour # 2.

Remarks: See also SPRITE and SPRCOLOR.

Example: Using RSPCOLOR:

```
10 SPRITE 1,1      :REM TURN SPRITE 1 ON
20 C1% = RSPCOLOR(1) :REM READ COLOUR #1
30 C2% = RSPCOLOR(2) :REM READ COLOUR #2
```

RSPPPOS

Token: \$CE \$05

Format: RSPPPOS(sprite,n)

Usage: Returns sprite's position and speed

sprite : sprite number.

n = 0 : get X position.

n = 1 : get Y position.

n = 2 : get speed.

Remarks: See also **SPRITE** and **MOVSPR**.

Example: Using RSPPPOS:

```
10 SPRITE 1,1      :REM TURN SPRITE 1 ON
20 XP = RSPPPOS(1,0) :REM GET X OF SPRITE 1
30 YP = RSPPPOS(1,1) :REM GET Y OF SPRITE 1
30 SP = RSPPPOS(1,2) :REM GET SPEED OF SPRITE 1
```

RSPRITE

Token: \$CE \$06

Format: RSPRITE(sprite,n)

Usage: Returns sprite's parameter.

sprite : sprite number (0 -> 7)

n = 0 : turned on (0 or 1).

n = 1 : foreground colour (0 -> 15)

n = 2 : background priority (0 or 1).

n = 3 : X-expanded (0 or 1).

n = 4 : Y-expanded (0 or 1).

n = 5 : multicolour (0 or 1).

Remarks: See also SPRITE and MOVSPR.

Example: Using RSPRITE:

```
10 SPRITE 1,1      :REM TURN SPRITE 1 ON
20 EN = RSPRITE(1,0) :REM SPRITE 1 ENABLED ?
30 FG = RSPRITE(1,1) :REM SPRITE 1 FOREGROUND COLOUR INDEX
30 BP = RSPRITE(1,2) :REM SPRITE 1 BACKGROUND PRIORITY
20 XE = RSPRITE(1,3) :REM SPRITE 1 X EXPANDED ?
30 YE = RSPRITE(1,4) :REM SPRITE 1 Y EXPANDED ?
30 MC = RSPRITE(1,5) :REM SPRITE 1 MULTICOLOUR ?
```

RUN

Token: \$8A

Format: **RUN [line number]**
RUN filename [,D drive] [,U unit]

Usage: Run a BASIC program.

If a filename is given, the program file is loaded into memory, otherwise the program that is currently in memory is used.

line number an existing line number of the program in memory.

filename is either a quoted string, e.g. "**prog**" or a string expression in parentheses, e.g. (**PR\$**). The filetype must be "PRG".

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

RUN resets first all internal pointers to their starting values. Therefore there are no variables, arrays and strings defined, the runtime stack is reset and the table of open files is cleared.

Remarks: In order to start or continue program execution without resetting everything, use the **GOTO** command.

Example: Using **RUN**

```
RUN "FLIGHTSIM" :LOAD AND RUN PROGRAM FLIGHTSIM
RUN 1000      :RUN PROGRAM IN MEMORY, START AT 1000
RUN          :RUN PROGRAM IN MEMORY
```

SAVE

Token: \$94

Format: **SAVE filename [,unit]**

Usage: "Saves a BASIC program to a file of type PRG.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FN\$**) The maximum length of the filename is 16 characters, not counting the optional save and replace character '@' and the infile drive definition.. If the first character of the filename is an at-sign '@' it is interpreted as a "save and replace" operation. It is dangerous to use this replace option on drives 1541 and 1571, because they contain the notorious "save and replace bug" in their DOS. The filename may be preceded by the drive number definition "0:" or "1:" which is only relevant for dual drive disk units.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: This is an obsolete command, implemented only for compatibility to older BASIC dialects. The command **DSAVE** should be used instead.

Example: Using **SAVE**

```
SAVE "ADVENTURE"  
SAVE "ZORK-I",8  
SAVE "1:DUNGEON",9
```

SCNCLR

Token: \$E8

Format: **SCNCLR [colour]**

Usage: Clears a text window or screen.

SCNCLR (with no arguments) clears the current text window. The default window occupies the whole screen.

SCNCLR colour clears the graphic screen by filling it with the colour.

Example: Using **SCNCLR**:

```
10 GRAPHIC CLR      :REM INITIALIZE
20 SCREEN DEF 1,1,1,2 :REM 640 X 400 X 2
30 SCREEN SET 1,1     :REM VIEW IT
40 SCNCLR 0          :REM CLEAR SCREEN
50 LINE 50,50,590,350 :REM DRAW LINE
```

SCRATCH

Token: \$F2

Format: SCRATCH **filename** [,D drive] [,U unit] [,R]

Usage: Used to erase a disk file.

filename is either a quoted string, e.g. “**data**” or a string expression in parentheses, e.g. (**FNS\$**)

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

R = Recover a previously erased file. This will only work, if there were no write operations between erasure and recovery, which may have altered the contents of the file.

Remarks: The **SCRATCH filename** command works like the **ERASE filename** command.

The success and the number of erased files can be examined by printing or using the system variable DS\$. The second last number, which reports the track number in case of an disk error, now reports the number of successfully erased files.

Example: Using **SCRATCH**

```
SCRATCH "DRM",U9 :REM SCRATCH FILE DRM ON UNIT 9
PRINT DS$
01, FILES SCRATCHED,01,00
SCRATCH "OLD*"   :REM SCRATCH ALL FILES BEGINNING WTH "OLD"
PRINT DS$
01, FILES SCRATCHED,04,00
```

SCREEN

Token: \$FE \$2E

Format:

- SCREEN CLR**
- SCREEN DEF**
- SCREEN SET**
- SCREEN OPEN screen [,errvar]**
- SCREEN CLOSE screen**

Usage: **SCREEN** performs one of five actions, selected by the secondary keyword after it.

SCREEN CLR colour clear graphics screen by filling it with colour.

SCREEN DEF screen, width, height, depth defines resolution parameters for the chosen screen.

SCREEN SET draw view sets screen numbers (0 or 1) for the drawing and the viewing screen.

SCREEN OPEN screen allocates resources and initializes the graphic context for the selected screen (0 or 1). An optional variable name as a further argument, gets the result of the command and can be tested afterwards for success.

SCREEN CLOSE screen closes screen (0 or 1) and frees resources.

Remarks: The **SCREEN** command cannot be used alone. It must always be used together with a secondary keyword.

Example: Using **SCREEN**:

```
10 GRAPHIC CLR      :REM INITIALIZE
20 SCREEN DEF 1,0,0,2 :REM SCREEN #1: 320 X 200 X 2
30 SCREEN OPEN 1     :REM OPEN SCREEN 1
40 SCREEN SET 1,1    :REM USE SCREEN 1 FOR RENDERING AND VIEWING
50 SCREEN CLR 0      :REM CLEAR SCREEN
60 LINE 25,25,295,175 :REM DRAW LINE
70 SLEEP 10          :REM WAIT 10 SECONDS
80 SCREEN CLOSE 1    :REM CLOSE SCREEN 1
```

SET

Token: \$FE \$2D

Format: **SET DEF unit**
SET DISK old to new

Usage: **SET DEF unit** redefines the default unit for disk access, which is initialized to 8 by the DOS. Commands, that do not explicitly specify a unit, will use this default unit.

SET DISK old to new is used to change the unit number of a disk drive temporarily.

Remarks: These settings are valid until a reset or shutdown.

Example: Using **SET**:

```
DIR          :REM SHOW DIRECTORY OF UNIT 8
SET DEF 11    :REM UNIT 11 BECOMES DEFAULT
DIR          :REM SHOW DIRECTORY OF UNIT 11
DLOAD "*"     :REM LOAD FIRST FILE FROM UNIT 11
SET DISK 8 TO 9 :REM CHANGE UNIT# OF DISK DRIVE 8 TO 9
DIR U9        :REM SHOW DIRECTORY OF UNIT 9 (FORMER 8)
```

SGN

Token: \$B4

Format: **SGN**(numeric expression)

Usage: The **SGN** function extracts the sign from the argument and returns it as a number:

- 1 for a negative argument
- 0 for a zero
- 1 for a positive, non zero argument

Example: Using **SGN**

```
10 OM SGN(X)+2 GOTO 100,200,300 :REM TARGETS FOR MINUS,ZERO,PLUS  
20 Z = SGN(X) * ABS(Y) : REM COMBINE SIGN OF X WITH VALUE OF Y
```

SIN

Token: \$BE

Format: **SIN**(numeric expression)

Usage: The **SIN** function returns the sine of the argument. The argument is expected in units of **[radians]**. The result is in the range (-1.0 to +1.0)

Remarks: An argument in units of **[degrees]** can be converted to **[radians]** by multiplication with $\pi/180$.

Example: Using **SIN**

```
PRINT SIN(0.7)
.644217687

X=30:PRINT SIN(X * pi / 180)
.5
```

SLEEP

Token: \$FE \$0B

Format: **SLEEP** seconds

Usage: The **SLEEP** command pauses the execution for the given duration (1 - 65535).

Example: Using **SLEEP**

```
50 GOSUB 1000 :REM DISPLAY SPLASH SCREEN
60 SLEEP 10   :REM WAIT 10 SECONDS
70 GOTO  2000 :REM START PROGRAM
```

SLOW

Token: \$FE \$26

Format: **SLOW**

Usage: Slow down system clock to 1 MHz.

Example: Using **SLOW**

```
50 SLOW      :REM SET SPEED TO MINIMUM
60 GOSUB 100 :REM EXECUTE SUBROUTINE AT 1 MHZ
70 FAST       :REM BACK TO HIGH SPEED
```

SOUND

- Token:** \$DA
- Format:** **SOUND voice, freq, dur [,dir ,min, sweep, wave, pulse]**
- Usage:** plays a sound effect.
- voice** = voice number (1 -> 6).
- freq** = frequency (0 -> 65535).
- dur** = duration (0 -> 32767) .
- dir** = direction (0:up, 1:down, 2:oscillate).
- min** = minimum frequency (0 -> 65535).
- sweep** = sweep range (0 -> 65535).
- wave** = waveform (0:triangle, 1:saw, 2:square, 3:noise).
- pulse** = pulse width (0 -> 5095).
- For details on sound programming, read the **SOUND** chapter.
- Remarks:** The **SOUND** command starts playing the sound effect and immediately continues with the execution of the next BASIC statement, while the sound effect is played. This enables showing graphics or text and playing sounds simultaneously.
- Example:** Using **SOUND**

```
SOUND 1, 7382, 60 :REM PLAY SQUARE WAVE ON VOICE 1 FOR 1 SECOND  
SOUND 2, 800, 3600 :REM PLAY SQUARE WAVE ON VOICE 2 FOR 1 MINUTE  
SOUND 3, 4000, 120, 2, 2000, 400, 1  
REM PLAY SWEEPING SAWTOOTH WAVE AT VOICE 3
```

SPC

Token: \$A6

Format: **SPC(columns)**

Usage: The **SPC** function skips **columns**.

The effect is like printing **column** times a **cursor right character**.

Remarks: The name of this function is derived from **SPACES**, which is misleading. The function prints **cursor right characters** not **SPACES**. The contents of those character cells, that are skipped, will not be changed.

Example: Using **SPC**

```
10 FOR I=8 TO 12
20 PRINT SPC(-(I<10));I :REM TRUE = -1, FALSE = 0
30 NEXT I
RUN
8
9
10
11
12
```

SPRCOLOR

Token: \$FE \$08

Format: **SPRCOLOR [mc1] [,mc2]**

Usage: Sets multicolour sprite colours.

The **SPRITE** command, which sets the attributes of a sprite, sets only the foreground colour. For the setting of the additional two colours, of multicolour sprites, **SPRCOLOR** has to be used.

Remarks: See also **SPRITE**.

Example: Using **SPRCOLOR**:

```
10 SPRITE 1,1,2,,,1 :REM TURN SPRITE 1 ON (FG = 2)
20 SPRCOLOR 4,5       :REM MC1 = 4, MC2 = 5
```

SPRITE

Token: \$FE \$07

Format: **SPRITE no [switch, colour, prio, expx, expy, mode]**

Usage: Switches a sprite on or off and sets its attributes.

no = sprite number

switch = 1:ON, 0:OFF

colour = sprite foreground colour

prio = sprite(1) or screen(0) priority

expx = 1:sprite X expansion

expy = 1:sprite Y expansion

mode = 1:multi colour sprite

Remarks: The command **SPRCOLOR** must be used to set additional colours for multi colour sprites (mode = 1)

Example: Using **SPRITE**:

```
10 COLLISION 1,70 : REM ENABLE
20 SPRITE 1,1 : MOVSPr 1,120, 0 : MOVSPr 1,0#5
30 SPRITE 2,1 : MOVSPr 2,120,100 : MOVSPr 2,180#5
40 FOR I=1 TO 50000:NEXT
50 COLLISION 1 : REM DISABLE
50 END
70 REM SPRITE (-) SPRITE INTERRUPT HANDLER
80 PRINT "BUMP RETURNS";BUMP(1)
90 RETURN: REM RETURN FROM INTERRUPT
```

SPRSAV

Token: \$FE \$16

Format: **SPRSAV source, destination**

Usage: Copies sprite data.

source = sprite number or string variable.

destination = sprite number or string variable.

Remarks: Both, source and destination can be either a sprite number or a string variable. But they must not be both a string variable. A simple string assignment can be used for such cases.

Example: Using **SPRSAV**:

```
10 BLOAD "SPRITEDATA",P1600      :REM LOAD DATA FOR SPRITE 1
20 SPRITE 1,1                      :REM TURN SPRITE 1 ON
30 SPRSAV 1,2                      :REM COPY SPRITE 1 DATA TO 2
40 SPRITE 2,1                      :REM TURN SPRITE 2 ON
50 SPRSAV 1,A$                      :REM SAVE SPRITE 1 DATA IN STRING
```

SQR

Token: \$BA

Format: **SQR**(numeric expression)

Usage: The **SQR** function returns the square root of the argument.

Remarks: The argument must not be negative.

Example: Using **SQR**

```
PRINT SQR(2)
1.41421356
```

STEP

Token: \$A9

Format: **FOR index=start TO end [STEP step] ... NEXT [index]**

Usage: The **STEP** keyword is an optional part of a **FOR** loop.

The **index** variable may be incremented or decremented by a constant value on each iteration. The default is to increment the variable by 1. The index variable must be a real variable.

The **start** value is used to initialize the index.

The **end** value is used at the end of the loop and controls, whether the next iteration will be started or the loop exited.

The **step** value defines the change applied to the index variable at the end of the loop. Positive step values increment it, while negative values decrement it. It defaults to 1.0 if not specified.

Remarks: For positive increments **end** must be greater or equal than **start**, for negative increments **end** must be less or equal than **start**.

It is bad programming style to change the value of the index variable inside the loop or to jump into or out of the loop body with **GOTO**.

Example: Using **STEP**

```
10 FOR D=0 TO 360 STEP 30
20 R = D * π / 180
30 PRINT D;R;SIN(R);COS(R);TAN(R)
40 NEXT D
```

STOP

Token: \$90

Format: **STOP**

Usage: Stops the execution of the BASIC program. A message tells the line number of the break. The **READY.** prompt appears and the computer goes into direct mode waiting for keyboard input. The program execution can be resumed with the command **CONT**.

Remarks: All variable definitions are still valid after **STOP**. They may be inspected or altered and the program may be continued with the **CONT** statement. Every editing of the program source makes continuation impossible, however.

Example: Using **STOP**

```
10 IF V < 0 THEN STOP : REM NEGATIVE NUMBERS STOP THE PROGRAM  
20 PRINT SQR(V)      : REM PRINT SQUARE ROOT
```

STR\$

Token: \$C4

Format: **STR\$(numeric expression)**

Usage: Returns a string containing the formatted value of the argument, as if it were printed to the string.

Example: Using **STR\$**:

```
A$ = "THE VALUE OF PI IS " + STR$(π)  
PRINT A$  
THE VALUE OF PI IS 3.14159265
```

SYS

Token: \$9E

Format: **SYS address [, areg, xreg, yreg, zreg, sreg]**

Usage: Calls a machine language subroutine. This can be a ROM resident kernel or BASIC subroutine or a routine in RAM, which was loaded or poked to RAM before.

The CPU registers are loaded with the arguments, if specified. Then a subroutine call **JSR address** is performed. The called routine should exit with a **RTS** instruction. Then the register contents will be saved and the execution of the BASIC program continues.

address = start address of the subroutine.

areg = variable gets accumulator value.

xreg = variable gets X register value.

yreg = variable gets Y register value.

zreg = variable gets Z register value.

sreg = variable gets status register value.

The **SYS** command uses the current bank as set with the **BANK** command.

Remarks: The register values after a **SYS** call are stored in system memory. This enables the command **RREG** to retrieve these values.

Example: Using **SYS**:

```
10 BANK 128
20 BLOAD "ML PROG",8192
30 SYS 8192
40 RREG A,X,Y,Z,S
50 PRINT "REGISTER:";A;X;Y;Z;S
```

TAB

Token: \$A3

Format: TAB(column)

Usage: Positions the cursor at **column**.

This is only done, if the target column is right of the current cursor column, otherwise nothing happens. The column count starts with 0 for the left most column.

Remarks: This function must not be confused with the **TAB** key, which advances the cursor to the next tabstop.

Example: Using TAB

```
10 FOR I=1 TO 5
20 READ A$
30 PRINT "* " A$ TAB(10) " *"
40 NEXT I
50 END
60 DATA ONE, TWO, THREE, FOUR, FIVE
```

```
RUN
* ONE      *
* TWO      *
* THREE    *
* FOUR     *
* FIVE     *
```

TAN

Token: \$C0

Format: TAN(numeric expression)

Usage: Returns the tangent of the argument. The argument is expected in units of [radians].

Remarks: An argument in units of [degrees] can be converted to [radians] by multiplication with $\pi/180$.

Example: Using TAN

```
PRINT TAN(0.7)
.84228838

X=45:PRINT TAN(X * pi / 180)
.99999999
```

TEMPO

Token: \$FE \$05

Format: **TEMPO speed**

Usage: Sets the playback speed for the **PLAY** command.

speed = 1 -> 255.

The duration of a whole note is computed with *duration* = $24/speed$.

Example: Using **TEMPO**

```
10 ENVELOPE 9,10,5,10,5,2,4000:PLAY "T9"  
20 VOL 8  
30 TEMPO 24      :REM PLAY EACH NOTE FOR ONE SECOND  
40 PLAY "C D E F G A B"  
50 PLAY "U5 V1 C D E F G A B"
```

THEN

Token: \$A7

Format: **IF expression THEN true clause ELSE false clause**

Usage: The **THEN** keyword is part of an **IF** statement.

expression is a logical or numeric expression. A numerical expression is evaluated as **FALSE** if the value is zero and **TRUE** for any non zero value.

true clause are one or more statements starting directly after **THEN** on the same line. A linenumber after **THEN** performs a **GOTO** to that line.

false clause are one or more statements starting directly after **ELSE** on the same line. A linenumber after **ELSE** performs a **GOTO** to that line.

Remarks: The standard **IF ... THEN ... ELSE** structure is restricted to a single line. But the **true clause** or **false clause** may be expanded to several lines using a compound statement bracketed with the keywords **BEGIN** and **BEND**.

Example: Using **THEN**

```
10 IF V < 0 THEN PRINT RED$;:ELSE PRINT BLACK$;
20 PRINT V : REM PRINT NEGATIVE NUMBERS IN RED
30 INPUT "END PROGRAM:(Y/N)";A$
40 IF A$="Y" THEN END
50 IF A$="N" THEN 10:ELSE 30
```

TO

Token: \$A4

Format: keyword **TO**

Usage: **TO** is a secondary keyword used in combination with primary keywords like **GO**, **FOR**, **BACKUP**, **BSAVE**, **CHANGE**, **CONCAT**, **COPY**, **RENAME** and **SET DISK**

Remarks: The keyword **TO** cannot be used on its own.

Example: Using **TO**

```
10 GO TO 1000 :REM AS GOTO 1000
20 GOTO 1000 :REM SHORTER AND FASTER
30 FOR I=1 TO 10 :REM TO IS PART OF THE LOOP
40 PRINT I:NEXT :REM LOOP END
50 COPY "CODES" TO "BACKUP" :REM COPY SINGLE FILE
```

TRAP

Token: \$D7

Format: **TRAP [line number]**

Usage: **TRAP** with a valid line number activates the BASIC error handler with following consequences: In case of an error the BASIC interpreter does not stop with an error message, but saves execution pointer and line number, places the error number into the system variable **ER** and jumps to the line number of the TRAP command. The trapping routine can examine **ER** and decide, whether to **STOP** or **RESUME** execution.

TRAP with no argument disables the error handler. Errors will be handled by the normal system routines.

Example: Using **TRAP**

```
10 TRAP 100
20 FOR I=1 TO 100
30 PRINT EXP(I)
40 NEXT
50 PRINT "STOPPED FOR I =";I
60 END
100 PRINT ERR$(ER): RESUME 50
```

TROFF

Token: \$D9

Format: TROFF

Usage: Turns off trace mode (switched on by **TRON**).

Example: Using TROFF

```
10 TRON           :REM ACTIVATE TRACE MODE
20 FOR I=85 TO 100
30 PRINT I,EXP(I)
40 NEXT
50 TROFF          :REM DEACTIVATE TRACE MODE

RUN
[10][20][30] 85  8.22301268E+36
[40][30] 86  2.2352466E+37
[40][30] 87  6.0760302E+37
[40][30] 88  1.65163625E+38
[40][30] 89
?OVERFLOW ERROR IN 30
READY.
```

TRON

Token: \$D8

Format: TRON

Usage: Turns on trace mode.

Example: Using TRON

```
10 TRON           :REM ACTIVATE TRACE MODE
20 FOR I=85 TO 100
30 PRINT I;EXP(I)
40 NEXT
50 TROFF          :REM DEACTIVATE TRACE MODE

RUN
[10][20][30] 85  8.22301268E+36
[40][30] 86  2.2352466E+37
[40][30] 87  6.0760302E+37
[40][30] 88  1.65163625E+38
[40][30] 89
?OVERFLOW ERROR IN 30
READY.
```

TYPE

Token: \$FE \$27

Format: **TYPE filename [,D drive] [,U unit]**

Usage: types the contents of a file containing text in PETSCII code.

filename is either a quoted string, e.g. "**data**" or a string expression in parentheses, e.g. (**FN\$**)

drive = drive # in dual drive disk units.

The drive # defaults to **0** and can be omitted on single drive units like the 1581, 1571 or 1541 series.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: This command cannot be used to type BASIC programs. Use **LIST** for programs. **TYPE** can only process SEQ or USR files containing records of PETSCII text, delimited by the **CR** = CHR\$(13) character.

Example: Using **TYPE**

```
TYPE "README"  
TYPE "README 1ST",U9
```

UNTIL

Token: \$FC

Format: **DO ... LOOP**
DO [<UNTIL | WHILE> <logical expr.>]
. . . statements [**EXIT**]
LOOP [<UNTIL | WHILE> <logical expr.>]

Usage: The **DO** and **LOOP** keywords define the start and end of the most versatile BASIC loop. Using **DO** and **LOOP** alone, without any modifiers creates an infinite loop, that can be left by the **EXIT** statement only. The loop can be controlled by adding an **UNTIL** or a **WHILE** statement after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Example: Using **DO** and **LOOP**.

```
10 PW$="":DO  
20 GET A$:PW$=PW$+A$  
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)  
  
10 DO : REM WAIT FOR USER DECISION  
20 GET A$  
30 LOOP UNTIL A$='Y' OR A$='N' OR A$='y' OR A$='n'  
  
10 DO WHILE ABS(EPS) > 0.001  
20 GOSUB 2000 : REM ITERATION SUBROUTINE  
30 LOOP  
  
10 I%=0 : REM INTEGER LOOP 1 -> 100  
20 DO I%=I%+1  
30 LOOP WHILE I% < 101
```

USING

Token: \$FB

Format: PRINT [# channel,] USING, format, arguments

Usage: USING is a secondary keyword used after PRINT or PRINT#.

It defines the format string for the argument list. The values are printed following the directives of the format string.

channel : must be opened for output by an OPEN or DOPEN statement. If no channel is specified, the output goes to the screen.

format : A string which defines the rules for formatting.

numeric argument : The '#' characters set the position and width of the output string.

'+' or '-' set the sign option.

'.' sets the position of the decimal point.

^^^^ reserves place for the exponent.

string argument : The string may consist of printable characters and control codes. Printable characters are printed to the cursor position, while control codes are executed. The number of '#' characters sets the width of the output, a '=' sign centers and a '>' character right justifies the output.

, : A comma acts like a tabulator. ; : A semicolon acts as a separator between arguments of the list. Other than the comma character it does not put in any additional characters. A semicolon at the end of the argument list suppresses the automatic return character.

Example: Using PRINT USING

```
10 X = 12.34: A$ = "MEGA 65"
30 PRINT USING "###.##"; X      : REM "12.34"
40 PRINT USING "###"; X       : REM " 12"
50 PRINT USING "####"; A$     : REM "MEGA"
60 PRINT USING "#####"; A$   : REM "MEGA 65 "
70 PRINT USING "=#####"; A$  : REM " MEGA 65 "
80 PRINT USING "#####);"; A$ : REM " MEGA 65"
```

USR

Token: \$B7

Format: **USR(numeric expression)**

Usage: Using the function **USR(X)** in a numeric expression, puts the argument into the floating point accumulator 1 and jumps to the address \$02F7 expecting the address of the machine language user routine in \$02F8 - \$02F9. After executing the user routine, BASIC returns the contents of the floating point accumulator 1, which should be set by the user routine..

Remarks: Banks 0 -> 127 give access to RAM or ROM banks. Banks > 127 are used to access I/O and SYSTEM like VIC, SID, FDC, etc.

Example: Using **USR**

```
10 UX = DEC("7F00")      :REM ADDRESS OF USER ROUTINE
20 BANK 128                :REM SELECT SYSTEM BANK
30 BLOAD "ML-PROG",P(UX)    :REM LOAD USER ROUTINE
40 POKE (DEC("2F8")),UX AND 255 :REM USR JUMP TARGET LOW
50 POKE (DEC("2F9")),UX / 256   :REM USR JUMP TARGET HIGH
60 PRINT USR(a)            :REM PRINT RESULT FOR ARGUMENT PT
```

VAL

Token: \$C5

Format: **VAL**(string expression)

Usage: Converts a string to a floating point value.

This function acts like reading from a string.

Remarks: A string containing not a valid number will not produce an error but return 0 as result.

Example: Using **VAL**

```
PRINT VAL("78E2")
7800

PRINT VAL("7+5")
7

PRINT VAL("1.256")
1.256

PRINT VAL("$FFFF")
0
```

VERIFY

Token: \$95

Format: **VERIFY filename [,unit [,binflag]]**

Usage: This command is obsolete in BASIC-10, where the commands **DVERIFY** and **BVERIFY** are better alternatives.

VERIFY with no **binflag** compares a BASIC program in memory with a disk file of type PRG. It does the same as **DVERIFY**, but with a different syntax.

VERIFY with **binflag** compares a binary file in memory with a disk file of type PRG. It does the same as **BVERIFY**, but with a different syntax.

filename is either a quoted string, e.g. "**prog**" or a string expression.

unit = device number on the IEC bus. Typically in the range 8 to 11 for disk units. If a variable is used, it must be put in parentheses. The unit # defaults to **8**.

Remarks: **VERIFY** can only test for equality. It gives no information about the number or position of different valued bytes. The command exits either with the message **OK** or with **VERIFY ERROR**.

Example: Using **VERIFY**

```
VERIFY "ADVENTURE"  
VERIFY "ZORK-I",9  
VERIFY "1:DUNGEON",10
```

VOL

Token: \$DB

Format: **VOL volume**

Usage: Sets the volume for sound output with **SOUND** or **PLAY**.

volume = 0 (off) -> 15 (loudest).

Remarks: This volume setting affects all voices.

Example: Using VOL

```
10 ENVELOPE 9,10,5,10,5,2,4000:PLAY "T9"  
20 VOL 8  
30 TEMPO 100  
40 PLAY "C D E F G A B"  
50 PLAY "U5 V1 C D E F G A B"
```

WAIT

Token: \$92

Format: **WAIT address, andmask [, xormask]**

Usage: Pauses the BASIC program until a requested bit pattern is read from the given address.

address = the address at the current memory bank, which is read.

andmask = and mask applied.

xormask = xor mask applied.

WAIT reads the byte value from **address** and applies the masks:
result = PEEK(address) AND andmask XOR xormask

The pause ends if the result is nonzero, otherwise the reading is repeated. This may hang the computer infinitely, if the condition is never met.

Remarks: This command is typically used to examine hardware registers or system variables and wait for an event, e.g. joystick event, mouse event, keyboard press or a special raster line.

Example: Using **WAIT**

```
10 BANK 128
20 WAIT 211,1           :REM WAIT FOR SHIFT KEY BEING PRESSED
```

WHILE

Token: \$ED

Format: **DO ... LOOP**

DO [<UNTIL | WHILE> <logical expr.>]
... statements [**EXIT**]
LOOP [<UNTIL | WHILE> <logical expr.>]

Usage: The **DO** and **LOOP** keywords define the start and end of the most versatile BASIC loop. Using **DO** and **LOOP** alone, without any modifiers creates an infinite loop, that can be left by the **EXIT** statement only. The loop can be controlled by adding an **UNTIL** or a **WHILE** statement after the **DO** or **LOOP**.

Remarks: **DO** loops may be nested. An **EXIT** statement exits the current loop only.

Example: Using **DO** and **LOOP**

```
10 PW$="" : DO  
20 GET A$: PW$=PW$+A$  
30 LOOP UNTIL LEN(PW$)>7 OR A$=CHR$(13)  
  
10 DO : REM WAIT FOR USER DECISION  
20 GET A$  
30 LOOP UNTIL A$='Y' OR A$='N' OR A$='y' OR A$='n'  
  
10 DO WHILE ABS(EPS) > 0.001  
20 GOSUB 2000 : REM ITERATION SUBROUTINE  
30 LOOP  
  
10 I%=0 : REM INTEGER LOOP 1 -> 100  
20 DO I%=I%+1  
30 LOOP WHILE I% < 101
```

WINDOW

Token: \$FE \$1A

Format: **WINDOW left, top, right, bottom [,clear]**

Usage: Sets the text screen window.

left = left column

top = top row

right = right column

bottom = bottom row

clear = clear text window flag

The row values count from 0 to 24.

The column values count from 0 to 79 or 39 depending on the screen mode.

Remarks: There can be only one window on the screen. Striking the HOME key twice or printing CHR\$(19)CHR\$(19) will reset the window to the default full screen.

Example: Using **WINDOW**

```
10 WINDOW 0,1,79,24      :REM SCREEN WITHOUT TOP ROW
20 WINDOW 0,0,79,24,1    :REM FULL SCREEN WINDOW CLEARED
30 WINDOW 0,12,79,24     :REM LOWER HALF OF SCREEN
40 WINDOW 20,5,59,15     :REM SMALL CENTERED WINDOW
```

XOR

Token: \$CE \$08

Format: **XOR(operand,operand)**

Usage: The boolean **XOR** function belongs to the group of boolean operators like NOT, AND, OR, but is implemented as function in this BASIC interpreter. It performs a bitwise logical XOR (eXclusive OR) operation on two 16-bit values. Integer operands are used as they are. Real operands are converted to a signed 16 bit integer. Logical operands are converted to 16 bit integer using \$FFFF, decimal -1 for TRUE and \$0000, decimal 0, for FALSE.

XOR(0,0)	->	0
XOR(0,1)	->	1
XOR(1,0)	->	1
XOR(1,1)	->	0

Remarks: The result is of integer type. If the result is used in a logical context, the value of 0 is regarded as FALSE, all other, nonzero values are regarded as TRUE.

Example: Using **XOR**

```
FOR I = 0 TO 8: PRINT XOR(I,5);: NEXT I
5 4 7 6 1 0 3 2 13
```

C

APPENDIX

Special Keyboard Controls and Sequences

- ASCII Codes and CHR\$
- Control codes
- Shifted codes
- Escape Sequences

ASCII CODES AND CHR\$

You can use the PRINT CHR\$(X) statement to print a character. Below is the full table of ASCII codes you can print by index. For example, by using index 65 from the table below as: PRINT CHR\$(65) you will print the letter 'A'.

You can also do the reverse with the ASC statement. For example: PRINT ASC("A") Will output 65, which matches in the ASCII code table.

CHR\$	Prints	CHR\$	Prints	CHR\$	Prints
0		17	↓	34	"
1		18	PVS ON	35	#
2		19	CLR HOME	36	\$
3		20	INST DEL	37	%
4		21		38	&
5	WHT	22		39	'
6		23		40	(
7		24		41)
8	DISABLE	SHIFT	BOOK	42	*
9	ENABLE	SHIFT	BOOK	43	+
10		25		44	,
11		26		45	-
12		27		46	.
13	RETURN	28	RED	47	/
14	LOWER CASE	29	→	48	0
15		30	GRN	49	1
16		31	BLU	50	2
		32	SPACE		
		33	!		

CHR\$	Prints	CHR\$	Prints	CHR\$	Prints
51	3	75	K	99	田
52	4	76	L	100	田
53	5	77	M	101	□
54	6	78	N	102	田
55	7	79	O	103	田
56	8	80	P	104	田
57	9	81	Q	105	田
58	:	82	R	106	田
59	;	83	S	107	田
60	<	84	T	108	田
61	=	85	U	109	田
62	>	86	V	110	田
63	?	87	W	111	田
64	@	88	X	112	田
65	A	89	Y	113	田
66	B	90	Z	114	田
67	C	91	[115	田
68	D	92	£	116	田
69	E	93]	117	田
70	F	94	↑	118	田
71	G	95	←	119	田
72	H	96	田	120	田
73	I	97	田	121	田
74	J	98	田	122	田

CHR\$	Prints	CHR\$	Prints	CHR\$	Prints
123	田	146	RVS OFF	169	■
124	☒	147	CLR HOME	170	□
125	□□	148	INST DEL	171	田
126	π	149	□	172	□
127	■	150	☒	173	□
128		151	□	174	田
129	ORG	152	☒	175	□
130		153	□□	176	□
131		154	♦	177	田
132		155	田	178	田
133	F1	156	PUR	179	田
134	F3	157	←	180	□
135	F5	158	YEL	181	□
136	F7	159	CYN	182	□
137	F2	160	SPACE	183	□
138	F4	161	□	184	□
139	F6	162	□	185	□
140	F8	163	□	186	□
141	SHIFT RETURN	164	□	187	□
142	UPPERCASE	165	□	188	□
143		166	☒	189	□
144	BLK	167	□	190	□
145	↑	168	☒	191	□

CONTROL CODES

Keyboard Control	Function
CTRL + 1 to 8	Choose from the first range of colours.
CTRL + T	Backspace the character immediately to the left and to shift all rightmost characters one position to the left. This is the same function as the Backspace key.
CTRL + Z	Tabs the cursor to the left.
CTRL + E	Restores the colour of the cursor back to the default white.
CTRL + Q	moves the cursor down one line at a time. This is the same function produced by the Cursor Down key.
CTRL + G	produces a bell tone.
CTRL + J	is a line feed and moves the cursor down one row. This is the same function produced by the ↓ key.
CTRL + U	backs up to the start of the previous word, or unbroken string of characters. If there are no characters between the current cursor position and the start of the line, the cursor will move to the first column of the current line.
CTRL + W	advances forward to the start of the next word, or unbroken string of characters. If there are no characters between the current cursor position and the end of the line, the cursor will move to the first column of the next line.

Keyboard Control	Function
CTRL + B	turns on underline text mode. Turn off underline mode by pressing ESC then O .
CTRL + N	changes the text case mode from uppercase to lowercase.
CTRL + M	is the carriage return. This is the same function as the RETURN key.
CTRL +]	is the same function as → .
CTRL + I	tabs forward to the right.
CTRL + X	sets or clears the current screen column as a tab position. CTRL + I or Z will jump to all positions set with X . When there are no more tab positions, the cursor will stay at the end of the line with CTRL and I , or move to the start of the line in the case of CTRL and Z .
CTRL + K	locks the uppercase/lowercase mode switch usually performed with Y and SHIFT keys.
CTRL + L	enables the uppercase/lowercase mode switch that is performed with the Y and SHIFT keys.
CTRL + [is the same as pressing the ESC key.
CTRL + *	enters the Matrix Mode Debugger.

SHIFTED CODES

Keyboard Control	Function
SHIFT + 	Insert a character in the current cursor position and move all characters to the right by one position.
SHIFT + 	Clear home, clear the entire screen and move the cursor to the home position.

ESCAPE SEQUENCES

To perform an Escape Sequence, press and release the **ESC** key. Then press one of the following keys to perform the sequence:

Key	Sequence
X	Clears the screen and toggles between 40 and 80 column modes.
@	Clears the screen starting from the cursor to the end of the screen.
A	Enables the auto-insert mode. Any keys pressed will insert before other characters.
B	Sets the bottom-right window area of the screen at the cursor position. All typed characters and screen activity will be restricted to the area. Also see ESC then T .
C	Disables auto-insert mode, going back to overwrite mode.
D	Deletes the current line and moves other lines up one position.
E	Sets the cursor to non-flashing mode.
F	Sets the cursor to regular flashing mode.
G	Enables the bell which can be sounded using CTRL and G .
H	Disable the bell so that pressing CTRL and G will have no effect.
I	Inserts an empty line in the current cursor position and moves all subsequent lines down one position.

Key	Sequence
J	Moves the cursor to start of current line.
K	Move to end of the last non-whitespace character on the current line.
L	Enables scrolling when the cursor down key is pressed at the bottom of the screen.
M	Disables scrolling. When pressing the cursor down key at the bottom on the screen, the cursor will move to the top of the screen. The cursor is restricted at the top of the screen with the Cursor up key.
O	Cancels the quote, reverse, underline and flash modes.
P	Erases all characters from the cursor to the start of current line.
Q	Erases all characters from the cursor to the end of current line.
S	Switches the VIC-IV to colour range 16-31. These colours can be accessed with CTRL and keys 1 to 8 or M and keys 1 to 8 .
T	Set top-left window area of the screen at the cursor position. All typed characters and screen activity will be restricted to the area. Also see ESC then B .
U	Switches the VIC-IV to colour range 0-15. These colours can be accessed with CTRL and keys 1 to 8 or M and keys 1 to 8 .

Key	Sequence
V	Scrolls the entire screen up one line.
W	Scrolls the entire screen down one line.
X	Toggles the 40/80 column display. The screen will also clear home.
Y	Set the default tab stops (every 8 spaces) for the entire screen.
Z	Clears all the tab stops. Any tabbing with CTRL and I will move the cursor to the end of the line.
1 to 8	Choose from the second range of colours.

D

APPENDIX

Decimal, Binary and Hexadecimal

- **Numbers**
- **Notations and Bases**
- **Operations**
- **Signed and Unsigned Numbers**
- **Bitwise Logical Operators**
- **Converting Numbers**

NUMBERS

Simple computer programs, such as most of the introductory BASIC programs in this book, do not require an understanding of mathematics or much knowledge about the inner workings of the computer. This is because BASIC is considered a high-level programming language. It lets us program the computer somewhat indirectly, yet still gives us control over the computer's features. Most of the time, we don't need to concern ourselves with the computer's internal architecture, which is why BASIC is user friendly and accessible.

As you acquire deeper knowledge and become more experienced, you will often want to instruct the computer to perform complex or specialized tasks that differ from the examples given in this book. Perhaps for reasons of efficiency, you may also want to exercise direct and precise control over the contents of the computer's memory. This is especially true for applications that deal with advanced graphics and sound. Such operations are closer to the hardware and are therefore considered low-level. Some simple mathematical knowledge is required to be able to use these low-level features effectively.

The collective position of the tiny switches inside the computer—whether each switch is on or off—is the state of the computer. It is natural to associate numerical concepts with this state. Numbers let us understand and manipulate the internals of the machine via logic and arithmetic operations. Numbers also let us encode the two essential and important pieces of information that lie within every computer program: *instructions* and *data*.

A program's instructions tell a computer what to do and how to do it. For example, the action of outputting a text string to the screen via the statement **PRINT** is an instruction. The action of displaying a sprite and the action of changing the screen's border color are instructions too. Behind the scenes, every instruction you give to the computer is associated with one or more numbers (which, in turn, correspond to the tiny switches inside the computer being switched on or off). Most of the time these instructions won't look like numbers to you. Instead, they might take the form of statements in BASIC.

A program's data consists of information. For example, the greeting "HELLO MEGA65!" is PETSCII character data in the form of a text string. The graphical design of a sprite might be pixel data in the form of a hero for a game. And the color data of the screen's border might represent orange. Again, behind the scenes, every piece of data you give to the computer is associated with one or more numbers. Data is sometimes given directly next to the statement to which it applies. This data is referred to as a parameter or argument (such as when changing the screen colour with a **BACKGROUND 1** statement). Data

may also be given within the program via the BASIC statement **DATA** which accepts a list of comma-separated values.

All such numbers—regardless of whether they represent instructions or data—reside in the computer's memory. Although the computer's memory is highly structured, the computer does not distinguish between instructions and data, nor does it have separate areas of memory for each kind of information. Instead, both are stored in whichever memory location is considered convenient. Whether a given memory location's contents is part of the program's instructions or is part of the program's data largely depends on your viewpoint, the program being written and the needs of the programmer.

Although BASIC is a high-level language, it still provides statements that allow programmers to manipulate the computer's memory efficiently. The statement **PEEK** lets us read the information from a specified memory location: we can inspect the contents of a memory address. The statement **POKE** lets us store information inside a specified memory location: we can modify the contents of a memory address so that it is set to a given value.

NOTATIONS AND BASES

We now take a look at numbers.

Numbers are ideas about quantity and magnitude. In order to manipulate numbers and determine relationships between them, it's important for them to have a unique form. This brings us to the idea of the symbolic representation of numbers using a positional notation. In this appendix we'll restrict our discussion to whole numbers, which are also called *integers*.

The *decimal* representation of numbers is the one with which you will be most comfortable since it is the one you were taught at school. Decimal notation uses the ten Hindu-Arabic numerals 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9 and is thus referred to as a base 10 numeral system. As we shall see later, in order to express large numbers in decimal, we use a positional system in which we juxtapose digits into columns to form a bigger number.

For example, 53280 is a decimal number. Each such digit (0 to 9) in a decimal number represents a multiple of some power of 10. When a BASIC statement (such as **PEEK** or **POKE**) requires an integer as a parameter, that parameter is given in the decimal form.

Although the decimal notation feels natural and comfortable for humans to use, modern computers, at their most fundamental level, use a different notation. This notation is called *binary*. It is also referred to as a base 2 numeral

system because it uses only two Hindu-Arabic numerals: 0 and 1. Binary reflects the fact that each of the tiny switches inside the computer must be in exactly one of two mutually exclusive states: on or off. The number 0 is associated with off and the number 1 is associated with on. Binary is the simplest notation that captures this idea. In order to express large numbers in binary, we use a positional system in which we juxtapose digits into columns to form a bigger number and prefix it with a % sign.

For example, %1001 0110 is a binary number. Each such digit (0 or 1) in a binary number represents a multiple of some power of 2.

We'll see later how we can use special BASIC statements to manipulate the patterns of ones and zeros present in a binary number to change the state of the switches associated with it. Effectively, we can toggle individual switches on or off, as needed.

A third notation called *hexadecimal* is also often used. This is a base 16 numeral system. Because it uses more than ten digits, we need to use some letters to represent the extra digits. Hexadecimal uses the ten Hindu-Arabic digits 0 to 9 as well as the six Latin alphabetic characters as "digits" (A, B, C, D, E and F) to represent the numbers 10 to 15. This gives a total of sixteen symbols for the numbers 0 to 15. To express a large number in hexadecimal, we use a positional system in which we juxtapose digits into columns to form a bigger number and prefix it with a \$ sign.

For example, \$E7 is a hexadecimal number. Each such digit (0 to 9 and A to F) in a hexadecimal number represents a multiple of some power of 16.

Hexadecimal is not often used when programming in BASIC. It is more commonly used when programming in low-level languages like machine code or assembly language. It also appears in computer memory maps and its brevity makes it a useful notation, so it is described here.

Always remember that decimal, binary and hexadecimal are just different notations for numbers. A notation just changes the way the number is written (i.e., the way it looks on paper or on the screen), but its intrinsic value remains unchanged. A notation is essentially different ways of representing the same thing. The reason that we use different notations is that each notation lends itself more naturally to a different task.

When using decimal, binary and hexadecimal for extended periods you may find it handy to have a scientific pocket calculator with a programmer mode. Such calculators can convert between bases with the press of a button. They can also add, subtract, multiply and divide, and perform various bitwise logical operations. See Appendix K Reference Tables as it contains a Base Conversion table for decimal, binary, and hexadecimal for integers between 0 and 255.

The BASIC listing for this appendix is a utility program that converts individual numbers into different bases. It can also convert multiple numbers within a specified range.

Although these concepts might be new now, with some practice they'll soon seem like second nature. We'll look at ways of expressing numbers in more detail. Later, we'll also investigate the various operations that we can perform on such numbers.

Decimal

When representing integers using decimal notation, each column in the number is for a different power of 10. The rightmost position represents the number of units (because $10^0 = 1$) and each column to the left of it is 10 times larger than the column before it. The rightmost column is called the units column. Columns to the left of it are labelled tens (because $10^1 = 10$), hundreds (because $10^2 = 100$), thousands (because $10^3 = 1000$), and so on.

To give an example, the integer 53280 represents the total of 5 lots of 10000, 3 lots of 1000, 2 lots of 100, 8 lots of 10 and 0 units. This can be seen more clearly if we break the integer up into distinct parts, by column.

Since

$$53280 = 50000 + 3000 + 200 + 80 + 0$$

we can present this as a table with the sum of each column at the bottom.

TEN THOUSANDS	THOUSANDS	HUNDREDS	TENS	UNITS
$10^4 = 10000$	$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
5	0	0	0	0
	3	0	0	0
		2	0	0
			8	0
				0
5	3	2	8	0

Another way of stating this is to write the expression using multiples of powers of 10.

$$53280 = (5 \times 10^4) + (3 \times 10^3) + (2 \times 10^2) + (8 \times 10^1) + (0 \times 10^0)$$

Alternatively

$$53280 = (5 \times 10000) + (3 \times 1000) + (2 \times 100) + (8 \times 10) + (0 \times 1)$$

We now introduce some useful terminology that is associated with decimal numbers.

The rightmost digit of a decimal number is called the least significant digit, because, being the smallest multiplier of a power of 10, it contributes the least to the number's magnitude. Each digit to the left of this digit has increasing significance. The leftmost (non-zero) digit of the decimal number is called the most significant digit, because, being the largest multiplier of a power of 10, it contributes the most to the number's magnitude.

For example, in the decimal number 53280, the digit 0 is the least significant digit and the digit 5 is the most significant digit.

A decimal number a is m orders of magnitude greater than the decimal number b if $a = b \times (10^m)$. For example, 50000 is three orders of magnitude greater than 50, because it has three more zeros. This terminology can be useful when making comparisons between numbers or when comparing the time efficiency or space efficiency of two programs with respect to the sizes of the given inputs.

Note that unlike binary (which uses a conventional % prefix) and hexadecimal (which uses a conventional \$ prefix), decimal numbers are given no special prefix. In some textbooks you might see such numbers with a subscript instead. So decimal numbers will have a subscripted 10, binary numbers will have a subscripted 2, and hexadecimal numbers will have a subscripted 16.

Another useful concept is the idea of signed and unsigned decimal integers.

A signed decimal integer can be positive or negative or zero. To represent a signed decimal integer, we prefix it with either a + sign or a - sign. (By convention, zero, which is neither positive nor negative, is given the + sign.)

If, on the other hand, a decimal integer is unsigned it must be either zero or positive and does not have a negative representation. This can be illustrated with the BASIC statements **PEEK** and **POKE**. When we use **PEEK** to return the value contained within a memory location, we get back an unsigned decimal number. For example, the statement **PRINT (PEEK (49152))** outputs the contents of memory location 49152 to the screen as an unsigned decimal number. Note that the memory address that we gave to **PEEK** is itself an unsigned integer. When we use **POKE** to store a value inside a memory location, both the memory address and the value to store inside it are given as unsigned integers. For example, the statement **POKE 49152, 128** stores the unsigned decimal integer 128 into the memory address given by the unsigned decimal integer 49152.

Each memory location in the MEGA65 can store a decimal integer between 0 and 255. This corresponds to the smallest and largest decimal integers that can be represented using eight binary digits (eight bits). Also, the memory addresses are decimal integers between 0 and 65535. This corresponds to the smallest and largest decimal integers that can be represented using sixteen binary digits (sixteen bits).

Note that the largest number expressible using d decimal digits is $10^d - 1$. (This number will have d nines in its representation.)

Binary

Binary notation uses powers of 2 (instead of 10 which is for decimal). The rightmost position represents the number of units (because $2^0 = 1$) and each column to the left of it is 2 times larger than the column before it. Columns to the left of the rightmost column are the twos column (because $2^1 = 2$), the fours column (because $2^2 = 4$), the eights column (because $2^3 = 8$), and so on.

As an example, the integer %1101 0011 uses exactly eight binary digits and represents the total of 1 lot of 128, 1 lot of 64, 0 lots of 32, 1 lot of 16, 0 lots of 8, 0 lots of 4, 1 lot of 2 and 1 unit.

We can break this integer up into distinct parts, by column.

Since

$$\begin{aligned} \%1101\ 0011 &= \%1000\ 0000 + \%100\ 0000 + \%00\ 0000 + \%1\ 0000 + \%0000 + \\ &\quad \%000 + \%10 + \%1 \end{aligned}$$

we can present this as a table with the sum of each column at the bottom.

ONE								
HUNDRED AND TWENTY-EIGHTS	SIXTY-FOURS	THIRTY-TWOS	SIXTEENS	EIGHTS	FOURS	TWOS	UNITS	
$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	
1	0	0	0	0	0	0	0	
	1	0	0	0	0	0	0	
		0	0	0	0	0	0	
			1	0	0	0	0	
				0	0	0	0	
					0	0	0	
						1	0	
							1	
1	1	0	1	0	0	1	1	

Another way of stating this is to write the expression in decimal, using multiples of powers of 2.

$$\%11010011 = \\ (1 \times 2^7) + (1 \times 2^6) + (0 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0)$$

Alternatively

$$\%11010011 = \\ (1 \times 128) + (1 \times 64) + (0 \times 32) + (1 \times 16) + (0 \times 8) + (0 \times 4) + (1 \times 2) + (1 \times 1)$$

which is the same as writing

$$\%11010011 = 128 + 64 + 16 + 2 + 1$$

Binary has terminology of its own. Each binary digit in a binary number is called a *bit*. In an 8-bit number the bits are numbered consecutively with the least significant (i.e., rightmost) bit as bit 0 and the most significant (i.e., leftmost) bit as bit 7. In a 16-bit number the most significant bit is bit 15. A bit is said to be *set* if it equals 1. A bit is said to be *clear* if it equals 0. When a particular bit has a special meaning attached to it, we sometimes refer to it as a *flag*.

1	1	0	1	0	0	1	1
Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0

As mentioned earlier, each memory location can store an integer between 0 and 255. The minimum corresponds to %0000 0000 and the maximum corresponds to %1111 1111, which are the smallest and largest numbers that can be represented using exactly eight bits. The memory addresses use 16 bits. The smallest memory address, represented in exactly sixteen bits, is %0000 0000 0000 0000 and this corresponds to the smallest 16-bit number. Likewise, the largest memory address, represented in exactly sixteen bits,

is %1111 1111 1111 1111 and this corresponds to the largest 16-bit number.

It is often convenient to refer to groups of bits by different names. For example, eight bits make a *byte* and 1024 bytes make a *kilobyte*. Half a byte is called a *nybble*. See Appendix K Reference Tables for the Units of Storage table for further information.

Note that the largest number expressible using d binary digits is (in decimal) $2^d - 1$. (This number will have d ones in its representation.)

Hexadecimal

Hexadecimal notation uses powers of 16. Each of the sixteen hexadecimal numerals has an associated value in decimal.

Hexadecimal Numeral	Decimal Equivalent
\$0	0
\$1	1
\$2	2
\$3	3
\$4	4
\$5	5
\$6	6
\$7	7
\$8	8
\$9	9
\$A	10
\$B	11
\$C	12
\$D	13
\$E	14
\$F	15

The rightmost position in a hexadecimal number represents the number of ones (since $16^0 = 1$). Each column to the left of this digit is 16 times larger than the column before it. Columns to the left of the rightmost column are the 16-column (since $16^1 = 16$), the 256-column (since $16^2 = 256$), the 4096-column (since $16^3 = 4096$), and so on.

As an example, the integer \$A3F2 uses exactly four hexadecimal digits and represents the total of 10 lots of 4096 (because \$A = 10), 3 lots of 256

(because \$3 = 3), 15 lots of 16 (because \$F = 15) and 2 units (because \$2 = 2). We can break this integer up into distinct parts, by column.

Since

$$\$A3F2 = \$A000 + \$300 + \$F0 + \$2$$

we can present this as a table with the sum of each column at the bottom.

FOUR THOUSAND AND NINETY-SIXES	TWO HUNDRED AND FIFTY-SIXES	SIXTEENS	UNITS
$16^3 = 4096$	$16^2 = 256$	$16^1 = 16$	$16^0 = 1$
A	0	0	0
	3	0	0
		F	0
			2
A	3	F	2

Another way of stating this is to write the expression in decimal, using multiples of powers of 16.

$$\$A3F2 = (10 \times 16^3) + (3 \times 16^2) + (15 \times 16^1) + (2 \times 16^0)$$

Alternatively

$$\$A3F2 = (10 \times 4096) + (3 \times 256) + (15 \times 16) + (2 \times 1)$$

which is the same as writing

$$\$A3F2 = 40960 + 768 + 240 + 2$$

Again, like binary and decimal, the rightmost digit is the least significant and the leftmost digit is the most significant.

Each memory location can store an integer between 0 and 255, and this corresponds to the hexadecimal numbers \$00 and \$FF. The hexadecimal number \$FFFF corresponds to 65535—the largest 16-bit number.

Hexadecimal notation is often more convenient to use and manipulate than binary. Binary numbers consist of a longer sequence of ones and zeros, while hexadecimal is much shorter and more compact. This is because one hexadecimal digit is equal to exactly four bits. So a two-digit hexadecimal number comprises of eight bits with the low nibble equaling the right digit and the high nibble equaling the left digit.

Note that the largest number expressible using d hexadecimal digits is (in decimal) $16^d - 1$. (This number will have d \$F symbols in its representation.)

OPERATIONS

In this section we'll take a tour of some familiar operations like counting and arithmetic, and we'll see how they apply to numbers written in binary and hexadecimal.

Then we'll take a look at various logical operations using logic gates. These operations are easy to understand. They're also very important when it comes to writing programs that have extensive numeric, graphic or sound capabilities.

Counting

If we consider carefully the process of *counting* in decimal, this will help us to understand how counting works when using binary and hexadecimal.

Let's suppose that we're counting in decimal and that we're starting at 0. Recall that the list of numerals for decimal is (in order) 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Notice that when we add 1 to 0 we obtain 1, and when we add 1 to 1 we obtain 2. We can continue in this manner, always adding 1:

$$\begin{aligned}0 + 1 &= 1 \\1 + 1 &= 2 \\2 + 1 &= 3 \\3 + 1 &= 4 \\4 + 1 &= 5 \\5 + 1 &= 6 \\6 + 1 &= 7 \\7 + 1 &= 8 \\8 + 1 &= 9\end{aligned}$$

Since 9 is the highest numeral in our list of numerals for decimal, we need some way of handling the following special addition: $9 + 1$. The answer is that we can reuse our old numerals all over again. In this important step, we reset the units column back to 0 and (at the same time) add 1 to the tens column. Since the tens column contained a 0, this gives us $9 + 1 = 10$. We say we "carried" the 1 over to the tens column while the units column cycled back to 0.

Using this technique, we can count as high as we like. The principle of counting for binary and hexadecimal is very much same, except instead of using ten symbols, we get to use two symbols and sixteen symbols, respectively.

Let's take a look at counting in binary. Recall that the list of numerals for binary is (in order) just 0 and 1. So, if we begin counting at %0 and then add %1, we obtain %1 as the result:

$$\%0 + \%1 = \%1$$

Now, the sum %1 + %1 will cause us to perform the analogous step: we reset the units column back to zero and (at the same time) add %1 to the twos column. Since the twos column contained a %0, this gives us %1 + %1 = %10. We say we "carried" the %1 over to the twos column while the units column cycled back to %0. If we continue in this manner we can count higher.

$$\begin{aligned}\%1 + \%1 &= \%10 \\ \%10 + \%1 &= \%11 \\ \%11 + \%1 &= \%100 \\ \%100 + \%1 &= \%101 \\ \%101 + \%1 &= \%110 \\ \%110 + \%1 &= \%111 \\ \%111 + \%1 &= \%1000\end{aligned}$$

Now we'll look at counting in hexadecimal. The list of numerals for hexadecimal is (in order) 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. If we begin counting at \$0 and repeatedly add \$1 we obtain:

$$\begin{aligned}\$0 + \$1 &= \$1 \\ \$1 + \$1 &= \$2 \\ \$2 + \$1 &= \$3 \\ \$3 + \$1 &= \$4 \\ \$4 + \$1 &= \$5 \\ \$5 + \$1 &= \$6 \\ \$6 + \$1 &= \$7 \\ \$7 + \$1 &= \$8 \\ \$8 + \$1 &= \$9 \\ \$9 + \$1 &= \$A \\ \$A + \$1 &= \$B \\ \$B + \$1 &= \$C \\ \$C + \$1 &= \$D \\ \$D + \$1 &= \$E \\ \$E + \$1 &= \$F\end{aligned}$$

Now, when we compute \$F + \$1 we must reset the units column back to \$0 and add \$1 to the sixteens column as that number is "carried".

$$\$F + \$1 = \$10$$

Again, this process allows us to count as high as we like.

Arithmetic

The standard arithmetic operations of addition, subtraction, multiplication and division are all possible using binary and hexadecimal.

Addition is done in the same way that addition is done using decimal, except that we use base 2 or base 16 as appropriate. Consider the following example for the addition of two binary numbers.

$$\begin{array}{r} \%110 \\ + \%111 \\ \hline \%1101 \end{array}$$

We obtain the result by first adding the units columns of both numbers. This gives us $\%0 + \%1 = \%1$ with nothing to carry into the next column. Then we add the twos columns of both numbers: $\%1 + \%1 = \%0$ with a $\%1$ to carry into the next column. We then add the fours columns (plus the carry) giving $(\%1 + \%1) + \%1 = \%1$ with a $\%1$ to carry into the next column. Last of all are the eights columns. Because these are effectively both zero we only concern ourselves with the carry which is $\%1$. So $(\%0 + \%0) + \%1 = \%1$. Thus, $\%1101$ is the sum.

Next is an example for the addition of two hexadecimal numbers.

$$\begin{array}{r} \$7D \\ + \$69 \\ \hline \$E6 \end{array}$$

We begin by adding the units columns of both numbers. This gives us $\$D + \$9 = \$6$ with a $\$1$ to carry into the next column. We then add the sixteens columns (plus the carry) giving $(\$7 + \$6) + \$1 = \E with nothing to carry and so $\$E6$ is the sum.

We now look at subtraction. As you might suspect, binary and hexadecimal subtraction follows a similar process to that of subtraction for decimal integers.

Consider the following subtraction of two binary numbers.

$$\begin{array}{r} \%1011 \\ - \%110 \\ \hline \%101 \end{array}$$

Starting in the units columns we perform the subtraction $\%1 - \%0 = \%1$. Next, in the twos columns we perform another subtraction $\%1 - \%1 = \%0$. Last of all we subtract the fours columns. This time, because $\%0$ is less than $\%1$, we'll

need to borrow a %1 from the eights column of the top number to make the subtraction. Thus we compute %10 - %1 = %1 and deduct %1 from the eights column. The eights columns are now both zeros. Since %0 - %0 = %0 and because this is the leading digit of the result we can drop it from the final answer. This gives %101 as the result.

Let's now look at the subtraction of two hexadecimal numbers.

$$\begin{array}{r} \$3D \\ - \$1F \\ \hline \$1E \end{array}$$

To perform this subtraction we compute the difference of the units columns. In order to do this, we note that because \$D is less than \$F we will need to borrow \$1 from the sixteens column of the top number to make the subtraction. Thus, we compute \$1D - \$F = \$E and also compute \$3 - \$1 = \$2 in the sixteens column for the \$1 that we just borrowed. Next, we compute the difference of the sixteens column as \$2 - \$1 = \$1. This gives us a final answer of \$1E.

We won't give in depth examples of multiplication and division for binary and hexadecimal notation. Suffice to say that principles parallel those for the decimal system. Multiplication is repeated addition and division is repeated subtraction.

We will, however, point out a special type of multiplication and division for both binary and hexadecimal. This is particularly useful for manipulating binary and hexadecimal numbers.

For binary, multiplication by two is simple—just shift all bits to the left by one position and fill in the least significant bit with a %0. Division by two is simple too—just shift all bits to the right by one position and fill in the most significant bit with a %0. By doing these repeatedly we can multiply and divide by powers of two with ease.

Thus the binary number %111, when multiplied by eight has three extra zeros on the end of it and is equal to %111000. (Recall that $2^3 = 8$.) And the binary number %10100, when divided by four has two less digits and equals %101. (Recall that $2^2 = 4$.)

These are called left and right *bit shifts*. So if we say that we shift a number to the left four bit positions, we really mean that we multiplied it by $2^4 = 16$.

For hexadecimal, the situation is similar. Multiplication by sixteen is simple—just shift all digits to the left by one position and fill in the rightmost digit with a \$0. Division by sixteen is simple too—just shift all digits to the right by one

position. By doing this repeatedly we can multiply and divide by powers of sixteen with ease.

Thus the hexadecimal number \$F, when multiplied 256 has two extra zeros on the end of it and is equal to \$F00. (Recall that $16^2 = 256$.) And the hexadecimal number \$EA0, when divided by sixteen has one less digit and equals \$EA. (Recall that $16^1 = 16$.)

Logic Gates

There exist several so-called *logic gates*. The fundamental ones are NOT, AND, OR and XOR.

They let us set, clear and invert specific binary digits. For example, when dealing with sprites, we might want to clear bit 6 (i.e., make it equal to 0) and set bit 1 (i.e., make it equal to 1) at the same time for a particular graphics chip register. Certain logic gates will, when used in combination, let us do this.

Learning how these logic gates work is very important because they are the key to understanding how and why the computer executes programs as it does.

All logic gates accept one or more inputs and produce a single output. These inputs and outputs are always single binary digits (i.e., they are 1-bit numbers).

The NOT gate is the only gate that accepts exactly one bit as input. All other gates—AND, OR, and XOR—accept exactly two bits as input. All gates produce exactly one output, and that output is a single bit.

First, let's take a look at the simplest gate, the NOT gate.

The NOT gate behaves by inverting the input bit and returning this resulting bit as its output. This is summarized in the following table.

INPUT X	OUTPUT
0	1
1	0

We write NOT x where x is the input bit.

Next, we take a look at the AND gate.

As mentioned earlier, the AND gate accepts two bits as input and produces a single bit as output. The AND gate behaves in the following manner. Whenever both input bits are equal to 1 the result of the output bit is 1. For all other inputs the result of the output bit is 0. This is summarized in the following table.

INPUT X	INPUT Y	OUTPUT
0	0	0
0	1	0
1	0	0
1	1	1

We write x AND y where x and y are the input bits.

Next, we take a look at the OR gate.

The OR gate accepts two bits as input and produces a single bit as output. The OR gate behaves in the following manner. Whenever both input bits are equal to 0 the result is 0. For all other inputs the result of the output bit is 1. This is summarized in the following table.

INPUT X	INPUT Y	OUTPUT
0	0	0
0	1	1
1	0	1
1	1	1

We write x OR y where x and y are the input bits.

Last of all we look at the XOR gate.

The XOR gate accepts two bits as input and produces a single bit as output. The XOR gate behaves in the following manner. Whenever both input bits are equal in value the output bit is 0. Otherwise, both input bits are unequal in value and the output bit is 1. This is summarized in the following table.

INPUT X	INPUT Y	OUTPUT
0	0	0
0	1	1
1	0	1
1	1	0

We write x XOR y where x and y are the input bits.

Note that there do exist some other gates. They are easy to construct.

- NAND gate: this is an AND gate followed by a NOT gate
- NOR gate: this is an OR gate followed by a NOT gate
- XNOR gate: this is an XOR gate followed by a NOT gate

SIGNED AND UNSIGNED NUMBERS

So far we've largely focussed on unsigned integers. Unsigned integer have no positive or negative sign. They are always assumed to be positive. (For this purpose, zero is regarded as positive.)

Signed numbers, as mentioned earlier, can have a positive sign or a negative sign.

Signed numbers are represented by treating the most significant bit as a sign bit. This bit cannot be used for anything else. If the most significant bit is 0 then the result is interpreted as having a positive sign. Otherwise, the most significant bit is 1, and the result is interpreted as having a negative sign.

A signed 8-bit number can represent positive-sign numbers between 0 and 127, and negative-sign numbers between -1 and -128.

A signed 16-bit number can represent positive-sign numbers between 0 and 32767, and negative-sign numbers between -1 and -32768.

Reserving the most significant bit as the sign of the signed number effectively halves the range of the available positive numbers (i.e., compared to unsigned numbers), with the tradeoff being that we gain an equal quantity of negative numbers instead.

To negate any signed number, every bit in the signed number must be inverted and then %1 must added to the result. Thus, negating %0000 0101 (which is the signed number +5) gives %1111 1011 (which is the signed number -5). As expected, performing the negation of this negative number gives us +5 again.

BITWISE LOGICAL OPERATORS

The BASIC statements **NOT**, **AND**, **OR** and **XOR** have functionality similar to that of the logic gates that they are named after.

The **NOT** statement must be given a 16-bit signed decimal integer as a parameter. It returns a 16-bit signed decimal integer as a result.

In the following example, all sixteen bits of the signed decimal number +0 are equal to 0. The **NOT** statement inverts all sixteen bits as per the NOT gate. This sets all sixteen bits. If we interpret the result as a signed decimal number, we obtain the answer of -1.

```
PRINT (NOT 0)
-1
```

As expected, repeating the **NOT** statement on the parameter of -1 gets us back to where we started, since all sixteen set bits become cleared.

```
PRINT (NOT -1)
0
```

The **AND** statement must be given two 16-bit signed decimal integers as parameters. The second parameter is called the *bit mask*. The statement returns a 16-bit signed decimal integer as a result, having changed each bit as per the AND gate.

In the following example, the number +253 is used as the first parameter. As a 16-bit signed decimal integer, this is equivalent to the following number in binary: %0000 0000 1111 1101. The **AND** statement uses a bit mask as the second parameter with a 16-bit signed decimal value of +239. In binary this is the number %0000 0000 1110 1110. If we use the AND logic gate table on corresponding pairs of bits, we obtain the 16-bit signed decimal integer +237 (which is %0000 0000 1110 1100 in binary).

```
PRINT (253 AND 239)
237
```

We can see this process more clearly in the following table.

% 0000 0000 1111 1101
AND % 0000 0000 1110 1110
% 0000 0000 1110 1100

Notice that each bit in the top row passes through unchanged wherever there is a 1 in the mask bit below it. Otherwise the bit in that position gets cleared.

The **OR** statement must be given two 16-bit signed decimal integers as parameters. The second parameter is called the *bit mask*. The statement returns a 16-bit signed decimal integer as a result, having changed each bit as per the OR gate.

In the following example, the number +240 is used as the first parameter. As a 16-bit signed decimal integer, this is equivalent to the following number in binary: %0000 0000 1111 0000. The **OR** statement uses a bit mask as the

second parameter with a 16-bit signed decimal value of +19. In binary this is the number %0000 0000 0001 0011. If we use the OR logic gate table on corresponding pairs of bits, we obtain the 16-bit signed decimal integer +243 (which is %0000 0000 1111 0011 in binary).

```
PRINT (240 OR 19)  
243
```

We can see this process more clearly in the following table.

$$\begin{array}{r} \%0000\ 0000\ 1111\ 0000 \\ \textbf{OR}\ \%0000\ 0000\ 0001\ 0011 \\ \hline \%0000\ 0000\ 1111\ 0011 \end{array}$$

Notice that each bit in the top row passes through unchanged wherever there is a 0 in the mask bit below it. Otherwise the bit in that position gets set.

Next we look at the **XOR** statement. This statement must be given two 16-bit unsigned decimal integers as parameters. The second parameter is called the *bit mask*. The statement returns a 16-bit unsigned decimal integer as a result, having changed each bit as per the XOR gate.

In the following example, the number 14091 is used as the first parameter. As a 16-bit unsigned decimal integer, this is equivalent to the following number in binary: %0011 0111 0000 1011. The **XOR** statement uses a bit mask as the second parameter with a 16-bit unsigned decimal value of 8653. In binary this is the number %0010 0001 1100 1101. If we use the XOR logic gate table on corresponding pairs of bits, we obtain the 16-bit unsigned decimal integer 5830 (which is %0001 0110 1100 0110 in binary).

```
PRINT (XOR(14091,8653))  
5830
```

We can see this process more clearly in the following table.

$$\begin{array}{r} \%0011\ 0111\ 0000\ 1011 \\ \textbf{XOR}\ \%0010\ 0001\ 1100\ 1101 \\ \hline \%0001\ 0110\ 1100\ 0110 \end{array}$$

Notice that when the bits are equal the resulting bit is 0. Otherwise the resulting bit is 1.

Much of the utility of these bitwise logical operators comes through combining them together into a compound statement. For example, the VIC II register to

enable sprites is memory address 53269. There are eight sprites (numbered 0 to 7) with each bit corresponding to a sprite's status. Now suppose we want to turn off sprite 5 and turn on sprite 1, while leaving the statuses of the other sprites unchanged. We can do this with the following BASIC statement which combines an **AND** statement with an **OR** statement.

```
POKE 53269, (((PEEK(53269)) AND 223) OR 2)
```

The technique of using **PEEK** on a memory address and combining the result with bitwise logical operators, followed by a **POKE** to that same memory address is very common.

CONVERTING NUMBERS

The program below is written in BASIC. It does number conversion for you. Type it in and save it under the name "CONVERT.BAS".

To execute the program, type **RUN** and press the **RETURN** key.

The program presents you with a series of text menus. You may choose to convert a single decimal, binary or hexadecimal number. Alternatively, you may choose to convert a range of such numbers.

The program can convert numbers in the range 0 to 65535.

```
10 REM *****
20 REM *          *
30 REM *  INTEGER BASE CONVERTER  *
40 REM *          *
50 REM *****
60 POKE 0,65: BORDER 6: BACKGROUND 6: FOREGROUND 1
70 DIM P(15)
80 E$ = "STARTING INTEGER MUST BE LESS THAN OR EQUAL TO ENDING INTEGER"
90 FOR N = 0 TO 15
100 : P(N) = 2 ↑ N
110 NEXT N
120 REM *** OUTPUT MAIN MENU ***
130 PRINT CHR$(147)
140 PRINT: PRINT "INTEGER BASE CONVERTER"
150 L = 22: GOSUB 1930: PRINT L$
160 PRINT: PRINT "SELECT AN OPTION (S, M OR Q):": PRINT
170 PRINT "[S](SPACE*2) SINGLE INTEGER CONVERSION"
```

```

180 PRINT "[M](SPACE*2)MULTIPLE INTEGER CONVERSION"
190 PRINT "[Q](SPACE*2)QUIT PROGRAM"
200 GET M$
210 IF (M$="S") THEN GOSUB 260: GOTO 140
220 IF (M$="M") THEN GOSUB 380: GOTO 140
230 IF (M$="Q") THEN END
240 GOTO 200
250 REM *** OUTPUT SINGLE CONVERSION MENU ***
260 PRINT: PRINT "(SPACE*2)SELECT AN OPTION (D, B, H OR R):": PRINT
270 PRINT "(SPACE*2)[D](SPACE*2)CONVERT A DECIMAL INTEGER"
280 PRINT "(SPACE*2)[B](SPACE*2)CONVERT A BINARY INTEGER"
290 PRINT "(SPACE*2)[H](SPACE*2)CONVERT A HEXADECIMAL INTEGER"
300 PRINT "(SPACE*2)[R](SPACE*2)RETURN TO TOP MENU"
310 GET M1$
320 IF (M1$="D") THEN GOSUB 500: GOTO 260
330 IF (M1$="B") THEN GOSUB 760: GOTO 260
340 IF (M1$="H") THEN GOSUB 810: GOTO 260
350 IF (M1$="R") THEN RETURN
360 GOTO 310
370 REM *** OUTPUT MULTIPLE CONVERSION MENU ***
380 PRINT: PRINT "(SPACE*2)SELECT AN OPTION (D, B, H OR R):": PRINT
390 PRINT "(SPACE*2)[D](SPACE*2)CONVERT A RANGE OF DECIMAL INTEGERS"
400 PRINT "(SPACE*2)[B](SPACE*2)CONVERT A RANGE OF BINARY INTEGERS"
410 PRINT "(SPACE*2)[H](SPACE*2)CONVERT A RANGE OF HEXADECIMAL INTEGERS"
420 PRINT "(SPACE*2)[R](SPACE*2)RETURN TO TOP MENU"
430 GET M2$
440 IF (M2$="D") THEN GOSUB 1280: GOTO 380
450 IF (M2$="B") THEN GOSUB 1670: GOTO 380
460 IF (M2$="H") THEN GOSUB 1800: GOTO 380
470 IF (M2$="R") THEN RETURN
480 GOTO 430
490 REM *** CONVERT SINGLE DECIMAL INTEGER ***
500 D$ = ""
510 PRINT: INPUT "ENTER DECIMAL INTEGER (UP TO 65535): ",D$
520 GOSUB 1030: REM VALIDATE DECIMAL INPUT
530 IF (V = 0) THEN GOTO 510
540 PRINT " DEC";SPC(4); "BIN";SPC(19); "HEX"
550 L = 5: GOSUB 1930: L1$ = L$
560 L = 20: GOSUB 1930: L2$ = L$
570 PRINT SPC(1);L1$;SPC(2);L2$;SPC(2);L1$
580 FOREGROUND 7
590 B$ = ""

```

```

600 D1 = 0
610 IF (D < 256) THEN GOTO 660
620 D1 = INT(D / 256)
630 FOR N = 1 TO 8
640 : IF ((D1 AND P(8 - N)) > 0) THEN B$ = B$ + "1": ELSE B$ = B$ + "0"
650 NEXT N
660 IF (D < 256) THEN B$ = "% " + B$: ELSE B$ = "% " + B$ + " "
670 D2 = D - 256*D1
680 FOR N = 1 TO 8
690 : IF ((D2 AND P(8 - N)) > 0) THEN B$ = B$ + "1": ELSE B$ = B$ + "0"
700 NEXT N
710 H$ = HEX$(D)
720 IF (D < 256) THEN H$ = "(SPACE*2)$" + RIGHT$(H$,2): ELSE H$ = "$" + H$
730 IF (D < 256) THEN PRINT SPC(6 - LEN(D$)); D$;SPC(12) + MID$(B$,1,5) +
" " + MID$(B$,6,10); "(SPACE*2)" + H$: FOREGROUND 1: RETURN
740 PRINT SPC(6 - LEN(D$));D$;"(SPACE*2)" + MID$(B$,1,5) + " " + MID$(B$,6,4) +
MID$(B$,10,5) + " " + MID$(B$,15,4); "(SPACE*2)" + H$: FOREGROUND 1: RETURN
750 REM *** CONVERT SINGLE BINARY INTEGER ***
760 I$=""
770 PRINT: INPUT "ENTER BINARY INTEGER (UP TO 16 BITS): ",I$
780 GOSUB 1110: REM VALIDATE BINARY INPUT
790 IF (V = 0) THEN GOTO 760: ELSE GOTO 540
800 REM *** CONVERT SINGLE HEXADECIMAL INTEGER ***
810 H$=""
820 PRINT: INPUT "ENTER HEXADECIMAL INTEGER (UP TO 4 DIGITS): ",H$
830 GOSUB 1220: REM VALIDATE HEXADECIMAL INPUT
840 IF (V = 0) THEN GOTO 810: ELSE GOTO 540
850 REM *** VALIDATE DECIMAL INPUT STRING ***
860 FOR N = 1 TO LEN(D$)
870 : M = ASC(MID$(D$,N,1)) - ASC("0")
880 : IF ((M < 0) OR (M > 9)) THEN V = 0
890 NEXT N: RETURN
900 REM *** VALIDATE BINARY INPUT STRING ***
910 FOR N = 1 TO LEN(I$)
920 : M = ASC(MID$(I$,N,1)) - ASC("0")
930 : IF ((M < 0) OR (M > 1)) THEN V = 0
940 NEXT N: RETURN
950 REM *** VALIDATE HEXADECIMAL INPUT STRING ***
960 FOR N = 1 TO LEN(H$)
970 : M = ASC(MID$(H$,N,1)) - ASC("0")
980 : IF (NOT (((M >= 0) AND (M <= 9)) OR
((M >= 17) AND (M <= 22)))) THEN V = 0

```

```

990 NEXT N: RETURN
1000 REM *** OUTPUT ERROR MESSAGE ***
1010 FOREGROUND 2: PRINT: PRINT A$: FOREGROUND 1: RETURN
1020 REM *** VALIDATE DECIMAL INPUT ***
1030 V = 1: GOSUB 860: REM VALIDATE DECIMAL INPUT STRING
1040 IF (V = 0) THEN A$ = "INVALID DECIMAL NUMBER": GOSUB 1010
1050 IF (V = 1) THEN BEGIN
1060 : D = VAL(D$)
1070 : IF ((D < 0) OR (D > 65535)) THEN A$ = "DECIMAL NUMBER OUT OF RANGE":
GOSUB 1010: V = 0
1080 BEND
1090 RETURN
1100 REM *** VALIDATE BINARY INPUT ***
1110 V = 1: GOSUB 910: REM VALIDATE BINARY INPUT STRING
1120 IF (V = 0) THEN A$ = "INVALID BINARY NUMBER": GOSUB 1010: RETURN
1130 IF (LEN(I$) > 16) THEN A$ = "BINARY NUMBER OUT OF RANGE":
GOSUB 1010: V = 0 : RETURN
1140 IF (V = 1) THEN BEGIN
1150 : I = 0
1160 : FOR N = 1 TO LEN(I$)
1170 : I = I + VAL(MID$(I$,N,1)) * P(LEN(I$) - N)
1180 : NEXT N
1190 BEND
1200 D$ = STR$(I): D = I: RETURN
1210 REM *** VALIDATE HEXADECIMAL INPUT ***
1220 V = 1: GOSUB 960: REM VALIDATE HEXADECIMAL INPUT STRING
1230 IF (V = 0) THEN A$ = "INVALID HEXADECIMAL NUMBER": GOSUB 1010: RETURN
1240 IF (LEN(H$) > 4) THEN A$ = "HEXADECIMAL NUMBER OUT OF RANGE":
GOSUB 1010: V = 0: RETURN
1250 D = DEC(H$): D$ = STR$(D): H = D: RETURN
1260 RETURN
1270 REM *** CONVERT MULTIPLE DECIMAL INTEGERS ***
1280 DB$=""
1290 PRINT: INPUT "ENTER STARTING DECIMAL INTEGER (UP TO 65535): ", DB$
1300 D$=DB$: GOSUB 1030: D$"": REM VALIDATE DECIMAL INPUT
1310 IF (V = 0) THEN GOTO 1290
1320 DE$=""
1330 PRINT: INPUT "ENTER ENDING DECIMAL INTEGER (UP TO 65535): ", DE$
1340 D$=DE$: GOSUB 1030: D$"": REM VALIDATE DECIMAL INPUT
1350 IF (V = 0) THEN GOTO 1330
1360 DB=VAL(DB$): DE=VAL(DE$)
1370 IF (DE < DB) THEN A$ = E$: GOSUB 1010: GOTO 1280

```

```
1380 SC = 1: SM = INT(((DE - DB) / 36) + 1)
1390 D = DB
1400 FOR J = SC TO SM
1410 : PRINT CHR$(147) + "RANGE: " + DB$ + " TO " + DE$ + "(SPACE*10)SCREEN: "
+ STR$(J) + " OF " + STR$(SM)
1420 : PRINT: PRINT "DEC"; SPC(4); "BIN"; SPC(19); "HEX"; SPC(8); "DEC"; SPC(4);
"BIN"; SPC(19); "HEX"
1430 L = 5: GOSUB 1930: L1$ = L$
1440 L = 20: GOSUB 1930: L2$ = L$
1450 : PRINT SPC(1); L1$; SPC(2); L2$; SPC(2); L1$; SPC(6); L1$; SPC(2);
L2$; SPC(2); L1$
1460 : FOR K = 0 TO 17
1470 :     FOREGROUND (7 + MOD(K,2))
1480 :     D$ = STR$(D): GOSUB 590: D = D + 1
1490 :     IF (D > DE) THEN GOTO 1630
1500 : NEXT K
1510 : PRINT CHR$(19): PRINT: PRINT: PRINT
1520 : FOR K = 0 TO 17
1530 :     FOREGROUND (7 + MOD(K,2))
1540 :     D$ = STR$(D): PRINT TAB(40): GOSUB 590: D = D + 1
1550 :     IF (D > DE) THEN GOTO 1630
1560 : NEXT K
1570 : FOREGROUND 1: PRINT: PRINT SPC(19);
"PRESS X TO EXIT OR SPACEBAR TO CONTINUE..."
1580 : GET B$
1590 : IF B$="X" THEN RETURN
1600 : IF B$="" THEN GOTO 1620
1610 : GOTO 1580
1620 NEXT J
1630 PRINT CHR$(19): FOR I = 1 TO 22: PRINT: NEXT I
1640 PRINT SPC(20); "COMPLETE. PRESS SPACEBAR TO CONTINUE..."
1650 GET B$: IF B$<>" " THEN GOTO 1650: ELSE RETURN
1660 REM *** CONVERT MULTIPLE BINARY INTEGERS ***
1670 IB$=""
1680 PRINT: INPUT "ENTER STARTING BINARY INTEGER (UP TO 16 BITS): ", IB$
1690 I$=IB$: GOSUB 1110: I$="": REM VALIDATE BINARY INPUT
1700 IF (V = 0) THEN GOTO 1680
1710 IB = I
1720 IE$=""
1730 PRINT: INPUT "ENTER ENDING BINARY INTEGER (UP TO 16 BITS): ", IE$
1740 I$=IE$: GOSUB 1110: I$="": REM VALIDATE BINARY INPUT
1750 IF (V = 0) THEN GOTO 1730
```

```
1760 IE = I
1770 IF (IE < IB) THEN AS = ES: GOSUB 1010: GOTO 1670
1780 DB = IB: DE = IE: DB$ = STR$(IB): DE$ = STR$(IE): GOTO 1380
1790 REM *** CONVERT MULTIPLE HEXADECIMAL INTEGERS ***
1800 HB$=""
1810 PRINT: INPUT "ENTER STARTING HEXADECIMAL INTEGER (UP TO 4 DIGITS): ", HB$
1820 HS=HB$: GOSUB 1220: HS"": REM VALIDATE HEXADECIMAL INPUT
1830 IF (V = 0) THEN GOTO 1810
1840 HB = H
1850 HE$=""
1860 PRINT: INPUT "ENTER ENDING HEXADECIMAL INTEGER (UP TO 4 DIGITS): ", HE$
1870 HS=HE$: GOSUB 1220: HS"": REM VALIDATE HEXADECIMAL INPUT
1880 IF (V = 0) THEN GOTO 1860
1890 HE = H
1900 IF (HE < HB) THEN AS = ES: GOSUB 1010: GOTO 1800
1910 DB = HB: DE = HE: DB$ = STR$(HB): DE$ = STR$(HE): GOTO 1380
1920 REM *** MAKE LINES ***
1930 LS=""
1940 FOR K = 1 TO L: LS = LS + "-": NEXT K
1950 RETURN
```

E

APPENDIX

45GS02 Microprocessor

- **Introduction**
- **Differences to the 6502**
- **C64 CPU Memory Mapped Registers**
- **New CPU Memory Mapped Registers**
- **MEGA65 CPU Math Unit Registers**
- **MEGA65 Hypervisor Mode**

INTRODUCTION

The 45GS02 is an enhanced version of the processor portion of the CSG4510 and of the F018 "DMAgic" DMA controller used in the Commodore 65 computer prototypes. The 4510 is, in turn, an enhanced version of the 65CE02. The reader is referred to the considerable documentation available for the 6502 and 65CE02 processors for the backwards-compatible operation of the 45GS02.

This chapter will focus on the differences between the 45GS02 and the earlier 6502-class processors, and the documentation of the many built-in memory-mapped IO registers of the 45GS02.

DIFFERENCES TO THE 6502

The 45GGS02 has a number of key differences to earlier 6502-class processors:

Supervisor/Hypervisor Privileged Mode

Unlike the earlier 6502 variants, the 45GS02 has a privileged mode of operation. This mode is intended for use by an operating system or type-1 hypervisor. The ambiguity between operating system and hypervisor on the MEGA65 stems from the fact that the operating system of the MEGA65 is effectively little more than a loader and task-switcher for C64 and C65 environments, i.e., effectively operating as a hypervisor, but provides only limited virtualisation of the hardware.

The key differences between normal and supervisor mode on the MEGA65, are that in supervisor mode:

- A special 16KiB memory area is mapped to \$8000 - \$BFFF, which is used to contain both the program and data of the hypervisor / supervisor program. This is normally the Hypo program. This memory is not mappable by any means when the processor is in the normal mode (the chip-select line to it is inhibited), protecting it from accidental or malicious access.
- The 64 SYSCALL trap registers in the MEGA65 IO-mode at \$D640 - \$D67F are replaced by the virtualisation control registers. These regis-

ters allow complete control over the system, and it is their access that truly defines the privilege of the supervisor mode.

- The processor always operates at full speed (40MHz) and in the 4510 processor personality.

The Hypervisor Mode is described in more detail later in this appendix.

6502 Illegal Opcodes

The 65C02, 65CE02 and CSG4510 processors extended the original 6502 processor by using previously unallocated opcodes of the 6502 to provide additional instructions. All software that followed the official documentation of the 6502 processor will therefore work on these newer processors, possibly with different instruction timing. However, the common practice on the C64 and other home computers of using undefined opcodes (often called “illegal opcodes”, although there is no law against using them), means that many existing programs will not work on these newer processors.

To alleviate this problem the 45GS02 has the ability to switch processor personalities between the 4510 and 6502. The effect is that in 6502 mode, none of the new opcodes of the 65C02, 65CE02, 4510 or 45GS02 are available, and are replaced with the original, often strange, behaviour of the undefined opcodes of the 6502.

WARNING: This feature is incomplete and untested. Most undocumented 6502 opcodes do not operate correctly when the 6502 personality is enabled.

Read-Modify-Write Instruction Bug Compatibility

The 65CE02 processor optimised a group of instructions called the Read-Modify-Write (RMW) instructions. For such instructions, such as INC, that increments the contents of a memory location, the 6502 would read the original value and then write it back unchanged, before writing it back with the new increased value. For most purposes, this did not cause any problems. However, it turned out to be a fast way to acknowledge VIC-II interrupts, because writing the original value back (which the instruction doesn't need to do) acknowledges the interrupt. This method is faster and uses fewer bytes than any alternative, and so became widely used in C64 software.

The problem came with the C65 with its 65CE02 derived CSG4510 that didn't do this extra write during the RMW instructions. This made the RMW instructions one cycle faster, which made software run slightly faster. Unfortunately, it also meant that a lot of existing C64 software simply won't run on a C65, unless the interrupt acknowledgement code in each program is patched to work around this problem. This is the single most common reason why many C64 games and other software titles won't run on a C65.

Because this problem is so common, the MEGA65's 45GS02 includes bug compatibility with this commonly used feature of the original 6502. It does this by checking if the target of an RMW instruction is \$D019, i.e., the interrupt status register of the VIC-II. If it is, then the 45GS02 performs the dummy write, allowing many C64 software titles to run unmodified on the MEGA65, that do not run on a C65 prototype. By only performing the dummy write if the address is \$D019, the MEGA65 maintains C64 compatibility, without sacrificing the speed improvement for all other uses of these instructions.

Variable CPU Speed

The 45GSG02 is able to run at 1MHz, 2MHz, 3.5MHz and 40MHz, to support running software designed for the C64, C128 in C64 mode, C65 and MEGA65.

Slow (1MHz – 3.5MHz) Operation

In these modes, the 45GS02 processor slows down, so that the same number of instructions per video frame are executed as on a PAL or NTSC C64, C128 in C64 mode or C65 prototype. This is to allow existing software to run on the MEGA65 at the correct speed, and with minimal display problems. The VIC-IV video controller provides cycle indication pulses to the 45GS02 that are used to keep time.

In these modes, opcodes take the same number of cycles as an 6502. However memory accesses within an instruction are not guaranteed to occur in the same cycle as on a 1MHz 6502. Normally the effect is that instructions complete faster, and the processor idles until the correct number of cycles have passed. This means that timing may be incorrect by upto 7 micro-seconds. This is not normally a problem, and even many C64 fast loaders will function correctly. For example, the GEOS™ Graphical Operating System for the C64 can be booted and used from a 1541 connected to the MEGA65's serial port.

However, some advanced VIC-II graphics tricks, such as Variable Screen Position (VSP) are highly unlikely to work correctly, due to the uncertainty in timing

of the memory write cycles of instructions. However, in most cases such problems can be easily solved by using the advanced features of the MEGA65's VIC-IV video controller. For example, VSP is unnecessary on the MEGA65, because you can set the screen RAM address to any location in memory.

Full Speed (40MHz) Instruction Timing

When the MEGA65's processor is operating at full speed (currently 40MHz), the instruction timing no longer exactly mirrors the 6502: Instructions that can be executed in fewer cycles will do so. For example, branches are typically require fewer instructions on the 45GS02. There are also some instructions that require more cycles on the 45GS02, in particular the LDA, LDX, LDY and LDZ instructions. Those instructions typically require one additional cycle. However as the processor is running at 40MHz, these instructions still execute much more quickly than on even a C65 or C64 with an accelerator.

CPU Speed Fine-Tuning

It is also possible to more smoothly vary the CPU speed using the **SPEEDBIAS** register located at \$F7FA (55290), when MEGA65 IO mode is enabled. The default value is \$80 (128), which means no bias on the CPU speed. Higher values increase the CPU speed, with \$FF meaning 2× the expected speed. Lower values slow the processor down, with \$00 bring the CPU to a complete stand-still. Thus the speed can be varied between 0× and 2× the intended value.

This register is provided to allow tweaking the processor speed in games.

Note that this register has no effect when the processor is running at full-speed, because it only affects the way in which VIC-IV video cycle indication pulses are processed by the CPU.

Direct Memory Access (DMA)

Direct Memory Access (DMA) is a method for quickly filling, copying or swapping memory regions. The MEGA65 implements an improved version of the F018 "DMAgic" DMA controller of the C65 prototypes. Unlike on the C65 prototypes, the DMA controller is part of the CPU on the MEGA65.

Detailed information on how to use the DMA controller and these advanced features can be found in Appendix F

Accessing memory between the 64KB and 1MB points

The C65 included four ways to access memory beyond the 64KB point: three methods that are limited, specialised or both, and two general-purpose methods. We will first consider the limited methods, before documenting the general-purpose methods.

C64-Style Memory Banking

The first method, is to use the C64-style \$00/\$01 ROM/RAM banking. This method is very limited, however, as it allows only the banking in and out of the two 8KB regions that correspond to the C64 BASIC and KERNAL ROMs. These are located at \$2A000 and \$2E000 in the 20-bit C65 address space, i.e., \$002A000 and \$002E000 in the 28-bit address space of the MEGA65. It can also provide access to the C64 character ROM data at \$D000, which is located at \$2D000 in the C65 memory map, and thus \$002D0000 in the MEGA65 address space. In addition to being limited to which regions this method can access, it also only provides read-only access to these memory regions, i.e., it cannot be used to modify these memory regions.

VIC-III “ROM” Banking

Similar to the C64-style memory banking, the C65 included the facility to bank several other regions of the C65’s 128KB ROM. These are banked in and out using various bits of the VIC-III’s \$D030 register:

\$D030 Bit	Signal Name	20-bit Address	16-bit Address	Read-Write Access?
0	CRAM2K	\$1F800 - \$1FFF, \$FF80000 - \$FF807FF	\$D800 - \$DFFF	Y
3	ROM8	\$38000 - \$39FFF	\$8000 - \$9FFF	N
4	ROMA	\$3A000 - \$3BFFF	\$A000 - \$BFFF	N
5	ROMC	\$2C000 - \$2CFFF	\$C000 - \$CFFF	N
6	CROM9	\$29000 - \$29FFF	\$D000 - \$DFFF	N

continued ...

...continued

\$D030 Bit	Signal Name	20-bit Address	16-bit Address	Read-Write Access?
7	ROME	\$3E000 - \$3FFFF	\$E000 - \$FFFF	N

The CRAM2K signal causes the normal 1KB of colour RAM, which is located at \$1F800 - \$1FBFF and is visible at \$D800 - \$DBFF, to instead be visible from \$D800 - \$DFFF. That is, the entire range \$1F800 - \$1FFFF is visible, and can be both read from and written to. Unlike on the C64, the colour RAM on the MEGA65 is always visible as 8-bit bytes. Also, on the MEGA65, the colour RAM is 32KB in size, and exists at \$FF80000 - \$FF87FFF. The visibility of the colour RAM at \$1F800 - \$1FFFF is achieved by mirroring writes to both regions when accessing the colour RAM via this mechanism.

Note that these VIC-III memory banking signals take precedence over the C64-style memory banking.

VIC-III Display Address Translator

The third specialised manner to access to memory above the 64KB point is to use the VIC-III's Display Address Translator. Use of this mechanism is documented in Appendix G.

The MAP instruction

The first general-purpose means of access to memory is the MAP instruction of the 4510 processor. The MEGA65's 45GS02 processor also supports this mechanism. This instruction divides the 64KB address of the 6502 into eight blocks of 8KB each. For each of these blocks, the block may either be accessed normally, i.e., accessing an 8KB region of the first 64KB of RAM of the system. Alternatively, each block may instead be re-mapped (hence the name of the MAP instruction) to somewhere else in the address space, by adding an offset to the address. Mapped addresses in the first 32KB use one offset, the lower offset, and the second 32KB uses another, the upper offset. Re-mapping of memory using the MAP instruction takes precedence over the C64-style memory banking, but not the C65's ROM banking mechanism.

The offsets must be a multiple of 256 bytes, and thus consist of 12 bits in order to allow an arbitrary offset in the 1MB address space of the C65. As each 8KB block in a 32KB half of memory can be either mapped or not, this requires one bit per 8KB block. Thus the processor requires 16 bits of information for each half of memory, for a total of 32 bits of information. This is achieved by setting

the A and X registers for the lower half of memory and the Y and Z registers for the upper half of memory, before executing the MAP instruction. The MAP instruction copies the contents of these registers into the processor's internal registers that hold the mapping information. Note that there is no way to use the MAP instruction to determine the current memory mapping configuration, which somewhat limits its effectiveness.

See under "Using the MAP instruction to access >1MB" for further explanation and an example.

Direct Memory Access (DMA) Controller

The C65's F018/F018A DMA controller allows for rapid filling, copying and swapping of the contents of memory anywhere in the 1MB address space. Detailed information about the F018 DMA controller, and the MEGA65's enhancements to this, refer to Appendix F

Flat Memory Access

Accessing memory beyond the 1MB point

The MEGA65 can support up to 256MiB of memory. This is more than the 1MiB address space of the CSG4510 on which it is based. There are several ways of performing this.

Using the MAP instruction to access >1MB

The full address space is available to the MAP instruction for legacy C65-style memory mapping, although some care is required, as the MAP instruction must be called up to three times. The reason for this is that the MAP instruction must be called to first select which mega-byte of memory will be used for the lower and upper map regions, before it is again called in the normal way to set the memory mapping. Because between these two calls the memory mapping offset will be a mix of the old and new addresses, all mapping should be first disabled via the MAP instruction. This means that the code to re-map memory should live in the bottom 64KB of RAM or in one of the ROM-bankable regions, so that it can remain visible during the mapping process.

Failure to handle this situation properly will result in the processor executing instructions from somewhere unexpected half-way through the process,

because the routine it is executing to perform the mapping will suddenly no longer be mapped.

Because of the relative complexity of this process, and the other problems with the MAP instruction as a means of memory access, we recommend that for accessing data outside of the current memory map that you use either DMA or the flat-memory address features of the 45GS02 that are described below. Indeed, access to the full address space via the MAP instruction is only provided for completeness.

To give a very simple example of how the MAP instruction can be used to map an area of memory from the expanded address space, the following program maps the ethernet frame buffer from its natural location at \$FFDE8000 to appear at \$6800. To keep the example as simple as possible, we assume that the code is running from in the bottom 64KB of RAM, and not in the region between \$6000 - \$8000.

As the MAP instruction normally is only aware of the C65-style 20-bit addresses, the MEGA65 extension to the instruction must be used to set the upper 8 bits of the 28-bit MEGA65 addresses, i.e., which mega-byte of address space should be used for the address translation. This is done by setting the X register to \$0F when setting the mega-byte number for the lower-32KB of the C64-style 64KB address space. This does not create any incompatibility with any sensible use of the MAP instruction on a C65, because this value indicates that none of the four 8KB memory blocks will be re-mapped, but at the same time specifies that the upper 4 bits of the address offset for re-mapped block is the non-zero value of \$F. The mega-byte number is then specified by setting the A register.

The same approach applies to the upper 32KB, but using the Z and Y registers instead of the X and A registers. However, in this case, we do not need to re-map the upper 32KB of memory in this example, we will leave the Z and Y registers set to zero. We must however set X and A to set the mega-byte number for the lower-32KB to \$FF. Therefore A must have the value \$FF. To set the lower 20-bits of the address offset we use the MAP instruction a second time, this time using it in the normal C65 manner. As we want to remap \$6800 to \$FFDE800, and have already dealt with the \$FFxxxx offset via the mega-byte number, we need only to apply the offset to make \$6800 point to \$DE800. \$DE800 minus \$6800 = \$D800. As the MAP instruction operates with a mapping granularity of 256 bytes = \$100, we can drop the last two digits from \$D8000 to obtain the MAP offset of \$D80. The lower 8-bits, \$80, must be loaded into the A register. The upper 4-bits, \$D, must be loaded into the low-nibble of the X register. As we wish to apply the mapping to only the fourth of the 8KB blocks that make up the lower 32KB half of the C64 memory

map, we must set the 4th bit of the upper nibble. That is, the upper nibble must be set to %1000, i.e., \$8. Therefore the X register must be loaded with \$8D. Thus we yield the complete example program:

```
; Map ethernet registers at $6000 - $7FFF

; Ethernet controller really lives $FFDE000 - $FFDEFFF, so select $FF megabyte section for MAP LO
lda #$ff
idx #$0f
idy #$00
idz #$00
Map

; now enable mapping of $DE000-$DFFFF at $6000
; MAPs are offset based, so we need to subtract $6000 from the target address
; ; $DE000 - $6000 = $D8000
lda #$80
idx #$8d
idy #$00
idz #$00
Map
eom

; Ethernet buffer now visible at $6000 - $6FFF
```

Note that the EOM (End Of Mapping) instruction (which is the same as NOP on a 6502, i.e., opcode \$EA) was only supplied after the last MAP instruction, to make sure that no interrupts could occur while the memory map contained mixed values with the mega-byte number set, but the lower-bits of the mapping address had not been updated.

No example in BASIC for the MAP instruction is possible, because the MAP is an machine code instruction of the 4510 / 45GS02 processors.

Flat-Memory Access

The 45GS02 makes it easy to read or write a byte from anywhere in memory by allowing the Zero-Page Indirect addressing mode to use a 32-bit pointer instead of the normal 16-bit pointer. This is accomplished by using the Z-indexed Zero-Page Indirect Addressing Mode for the access, and having the instruction directly preceeded by a NOP instruction (opcode \$EA). For example:

```
NOP  
LDA ($45),Z
```

Would read the four bytes of Zero-Page memory at \$45 - \$48 to form a 32-bit memory address, and add the value of the Z register to this to form the actual address that will be read from. The byte order in the address is the same as the 6502, i.e., the right-most (least significant) byte of the address will be read from the first address (\$45 in this case), and so on, until the left-most (most significant) byte will be read from \$48. For example, to read from memory location \$12345678, the contents of memory beginning at \$45 should be 78 56 43 12.

This method is much more efficient and also simpler than either using the MAP instruction or the DMA controller for single memory accesses, and is what we generally recommend. The DMA controller can be used for moving/filler larger regions of memory. We recommend the MAP instruction only be used for banking code, or in rare situations where extensive access to a small region of memory is required, and the extra cycles of reading the 32-bit addresses is problematic.

Virtual 32-bit Register

The 45GS02 allows the use of its four general purpose registers, A, X ,Y and Z as a single virtual 32-bit register. This can greatly simplify and speed up many common operations, and help avoid many common programming errors. For example, adding two 16-bit or 32-bit values can now be easily accomplished with something like:

```

; Clear carry before performing addition, as normal
CLC
; Prefix an instruction with two NEG instructions to select virtual 32-bit register mode
NEG
NEG
LDA $1234 ; Load the contents of $1234-$1237 into A,X,Y and Z respectively
; And again, for the addition
NEG
NEG
ADC $1238 ; Add the contents of $1238-$123B
; The result of the addition is now in A, X, Y and Z.
; And can be written out in whole or part

; To write it all out, again, we need the NEG + NEG prefix
NEG
NEG
STA $123C ; Write the whole out to $123C-$123F

; Or to write out the bottom bytes, we can just write the contents of A and X as normal
STA $1240
STX $1241

```

This approach works with the LDA, STA, ADC, SBC, CMP, EOR, AND, ORA, ASL, LSR, ROL, ROR, INC and DEC instructions. It also works with any addressing mode. Indexed addressing modes, where X, Y or Z are added to the address should be used with care, because these registers may in fact be holding part of a 32-bit value. The special case is the Zero-Page Indirect Z-Indexed addressing mode: In this case the Z register is NOT added to the target address, as would normally be the case. This is to allow the virtual 32-bit register to be able to be used with flat-memory access with the combined prefix of NEG NEG NOP before the instruction to allow accessing a 32-bit value anywhere in memory in a single instruction.

Note that the virtual 32-bit register cannot be used in immediate mode, i.e., to load a constant into the four general purpose registers. This is to avoid problems with variable length instructions. Also, it would not save any bytes compared to LDA #\$nn ... LDZ #\$nn, and would be no faster.

C64 CPU MEMORY MAPPED REGISTERS

HEX	DEC	Signal	Description
0000	0	PORTDDR	6510/45GS10 CPU port DDR
0001	1	PORT	6510/45GS10 CPU port data

NEW CPU MEMORY MAPPED REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D640	54848								HTRAP00
D641	54849								HTRAP01
D642	54850								HTRAP02
D643	54851								HTRAP03
D644	54852								HTRAP04
D645	54853								HTRAP05
D646	54854								HTRAP06
D647	54855								HTRAP07
D648	54856								HTRAP08
D649	54857								HTRAP09
D64A	54858								HTRAP0A
D64B	54859								HTRAP0B
D64C	54860								HTRAP0C
D64D	54861								HTRAP0D
D64E	54862								HTRAP0E
D64F	54863								HTRAP0F
D650	54864								HTRAP10
D651	54865								HTRAP11
D652	54866								HTRAP12
D653	54867								HTRAP13
D654	54868								HTRAP14
D655	54869								HTRAP15
D656	54870								HTRAP16
D657	54871								HTRAP17

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D658	54872					HTRAP18			
D659	54873					HTRAP19			
D65A	54874					HTRAP1A			
D65B	54875					HTRAP1B			
D65C	54876					HTRAP1C			
D65D	54877					HTRAP1D			
D65E	54878					HTRAP1E			
D65F	54879					HTRAP1F			
D660	54880					HTRAP20			
D661	54881					HTRAP21			
D662	54882					HTRAP22			
D663	54883					HTRAP23			
D664	54884					HTRAP24			
D665	54885					HTRAP25			
D666	54886					HTRAP26			
D667	54887					HTRAP27			
D668	54888					HTRAP28			
D669	54889					HTRAP29			
D66A	54890					HTRAP2A			
D66B	54891					HTRAP2B			
D66C	54892					HTRAP2C			
D66D	54893					HTRAP2D			
D66E	54894					HTRAP2E			
D66F	54895					HTRAP2F			
D670	54896					HTRAP30			
D671	54897					HTRAP31			
D672	54898					HTRAP32			
D673	54899					HTRAP33			
D674	54900					HTRAP34			
D675	54901					HTRAP35			
D676	54902					HTRAP36			
D677	54903					HTRAP37			
D678	54904					HTRAP38			
D679	54905					HTRAP39			
D67A	54906					HTRAP3A			
D67B	54907					HTRAP3B			
D67C	54908					HTRAP3C			
D67D	54909					HTRAP3D			

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D67E	54910				HTRAP3E				
D67F	54911				HTRAP3F				
D7FA	55290				SPEEDBIAS				
D7FB	55291				-				BRCOST
D7FD	55293	NOEXROM	NOGAME				-		

- **BRCOST** 1=charge extra cycle(s) for branches taken
- **HTRAP00** Writing triggers hypervisor trap \$00
- **HTRAP01** Writing triggers hypervisor trap \$01
- **HTRAP02** Writing triggers hypervisor trap \$02
- **HTRAP03** Writing triggers hypervisor trap \$03
- **HTRAP04** Writing triggers hypervisor trap \$04
- **HTRAP05** Writing triggers hypervisor trap \$05
- **HTRAP06** Writing triggers hypervisor trap \$06
- **HTRAP07** Writing triggers hypervisor trap \$07
- **HTRAP08** Writing triggers hypervisor trap \$08
- **HTRAP09** Writing triggers hypervisor trap \$09
- **HTRAP0A** Writing triggers hypervisor trap \$0A
- **HTRAP0B** Writing triggers hypervisor trap \$0B
- **HTRAP0C** Writing triggers hypervisor trap \$0C
- **HTRAP0D** Writing triggers hypervisor trap \$0D
- **HTRAP0E** Writing triggers hypervisor trap \$0E
- **HTRAP0F** Writing triggers hypervisor trap \$0F
- **HTRAP10** Writing triggers hypervisor trap \$10
- **HTRAP11** Writing triggers hypervisor trap \$11
- **HTRAP12** Writing triggers hypervisor trap \$12
- **HTRAP13** Writing triggers hypervisor trap \$13
- **HTRAP14** Writing triggers hypervisor trap \$14
- **HTRAP15** Writing triggers hypervisor trap \$15

- **HTRAP16** Writing triggers hypervisor trap \$16
- **HTRAP17** Writing triggers hypervisor trap \$17
- **HTRAP18** Writing triggers hypervisor trap \$18
- **HTRAP19** Writing triggers hypervisor trap \$19
- **HTRAP1A** Writing triggers hypervisor trap \$1A
- **HTRAP1B** Writing triggers hypervisor trap \$1B
- **HTRAP1C** Writing triggers hypervisor trap \$1C
- **HTRAP1D** Writing triggers hypervisor trap \$1D
- **HTRAP1E** Writing triggers hypervisor trap \$1E
- **HTRAP1F** Writing triggers hypervisor trap \$1F
- **HTRAP20** Writing triggers hypervisor trap \$20
- **HTRAP21** Writing triggers hypervisor trap \$21
- **HTRAP22** Writing triggers hypervisor trap \$22
- **HTRAP23** Writing triggers hypervisor trap \$23
- **HTRAP24** Writing triggers hypervisor trap \$24
- **HTRAP25** Writing triggers hypervisor trap \$25
- **HTRAP26** Writing triggers hypervisor trap \$26
- **HTRAP27** Writing triggers hypervisor trap \$27
- **HTRAP28** Writing triggers hypervisor trap \$28
- **HTRAP29** Writing triggers hypervisor trap \$29
- **HTRAP2A** Writing triggers hypervisor trap \$2A
- **HTRAP2B** Writing triggers hypervisor trap \$2B
- **HTRAP2C** Writing triggers hypervisor trap \$2C
- **HTRAP2D** Writing triggers hypervisor trap \$2D
- **HTRAP2E** Writing triggers hypervisor trap \$2E
- **HTRAP2F** Writing triggers hypervisor trap \$2F
- **HTRAP30** Writing triggers hypervisor trap \$30
- **HTRAP31** Writing triggers hypervisor trap \$31

- **HTRAP32** Writing triggers hypervisor trap \$32
- **HTRAP33** Writing triggers hypervisor trap \$33
- **HTRAP34** Writing triggers hypervisor trap \$34
- **HTRAP35** Writing triggers hypervisor trap \$35
- **HTRAP36** Writing triggers hypervisor trap \$36
- **HTRAP37** Writing triggers hypervisor trap \$37
- **HTRAP38** Writing triggers hypervisor trap \$38
- **HTRAP39** Writing triggers hypervisor trap \$39
- **HTRAP3A** Writing triggers hypervisor trap \$3A
- **HTRAP3B** Writing triggers hypervisor trap \$3B
- **HTRAP3C** Writing triggers hypervisor trap \$3C
- **HTRAP3D** Writing triggers hypervisor trap \$3D
- **HTRAP3E** Writing triggers hypervisor trap \$3E
- **HTRAP3F** Writing triggers hypervisor trap \$3F
- **NOEXROM** Override for /EXROM : Must be 0 to enable /EXROM signal
- **NOGAME** Override for /GAME : Must be 0 to enable /GAME signal
- **SPEEDBIAS** 1/2/3.5MHz CPU speed fine adjustment

MEGA65 CPU MATH UNIT REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D770	55152					MULTINA			
D771	55153					MULTINA			
D772	55154					MULTINA			
D773	55155				-			MULTINA	
D774	55156				MULTINB				
D775	55157				MULTINB				
D776	55158			-			MULTINB		
D778	55160				MULTOUT				

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D779	55161					MULTOUT			
D77A	55162					MULTOUT			
D77B	55163					MULTOUT			
D77C	55164					MULTOUT			
D77D	55165					MULTOUT			
D77E	55166					MULTOUT			
D780	55168					MATHINO			
D781	55169					MATHINO			
D782	55170					MATHINO			
D783	55171					MATHINO			
D784	55172					MATHIN1			
D785	55173					MATHIN1			
D786	55174					MATHIN1			
D787	55175					MATHIN1			
D788	55176					MATHIN2			
D789	55177					MATHIN2			
D78A	55178					MATHIN2			
D78B	55179					MATHIN2			
D78C	55180					MATHIN3			
D78D	55181					MATHIN3			
D78E	55182					MATHIN3			
D78F	55183					MATHIN3			
D790	55184					MATHIN4			
D791	55185					MATHIN4			
D792	55186					MATHIN4			
D793	55187					MATHIN4			
D794	55188					MATHIN5			
D795	55189					MATHIN5			
D796	55190					MATHIN5			
D797	55191					MATHIN5			
D798	55192					MATHIN6			
D799	55193					MATHIN6			
D79A	55194					MATHIN6			
D79B	55195					MATHIN6			
D79C	55196					MATHIN7			
D79D	55197					MATHIN7			
D79E	55198					MATHIN7			
D79F	55199					MATHIN7			

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D7A0	55200					MATHIN8			
D7A1	55201					MATHIN8			
D7A2	55202					MATHIN8			
D7A3	55203					MATHIN8			
D7A4	55204					MATHIN9			
D7A5	55205					MATHIN9			
D7A6	55206					MATHIN9			
D7A7	55207					MATHIN9			
D7A8	55208					MATHIN10			
D7A9	55209					MATHIN10			
D7AA	55210					MATHIN10			
D7AB	55211					MATHIN10			
D7AC	55212					MATHIN11			
D7AD	55213					MATHIN11			
D7AE	55214					MATHIN11			
D7AF	55215					MATHIN11			
D7B0	55216					MATHIN12			
D7B1	55217					MATHIN12			
D7B2	55218					MATHIN12			
D7B3	55219					MATHIN12			
D7B4	55220					MATHIN13			
D7B5	55221					MATHIN13			
D7B6	55222					MATHIN13			
D7B7	55223					MATHIN13			
D7B8	55224					MATHIN14			
D7B9	55225					MATHIN14			
D7BA	55226					MATHIN14			
D7BB	55227					MATHIN14			
D7BC	55228					MATHIN15			
D7BD	55229					MATHIN15			
D7BE	55230					MATHIN15			
D7BF	55231					MATHIN15			
D7C0	55232	UNIT0INB				UNIT0INA			
D7C1	55233	UNIT1INB				UNIT1INA			
D7C2	55234	UNIT2INB				UNIT2INA			
D7C3	55235	UNIT3INB				UNIT3INA			
D7C4	55236	UNIT4INB				UNIT4INA			
D7C5	55237	UNIT5INB				UNIT5INA			

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D7C6	55238		UNIT6INB					UNIT6INA	
D7C7	55239		UNIT7INB					UNIT7INA	
D7C8	55240		UNIT8INB					UNIT8INA	
D7C9	55241		UNIT9INB					UNIT9INA	
D7CA	55242		UNIT10INB					UNIT10INA	
D7CB	55243		UNIT11INB					UNIT11INA	
D7CC	55244		UNIT12INB					UNIT12INA	
D7CD	55245		UNIT13INB					UNIT13INA	
D7CE	55246		UNIT14INB					UNIT14INA	
D7CF	55247		UNIT15INB					UNIT15INA	
D7D0	55248		-					UNIT0OUT	
D7D1	55249		-					UNIT1OUT	
D7D2	55250		-					UNIT2OUT	
D7D3	55251		-					UNIT3OUT	
D7D4	55252		-					UNIT4OUT	
D7D5	55253		-					UNIT5OUT	
D7D6	55254		-					UNIT6OUT	
D7D7	55255		-					UNIT7OUT	
D7D8	55256		-					UNIT8OUT	
D7D9	55257		-					UNIT9OUT	
D7DA	55258		-					UNIT10OUT	
D7DB	55259		-					UNIT11OUT	
D7DC	55260		-					UNIT12OUT	
D7DD	55261		-					UNIT13OUT	
D7DE	55262		-					UNIT14OUT	
D7DF	55263		-					UNIT15OUT	
D7E0	55264				LATCHINT				
D7E1	55265			-			CALCEN		WREN
D7E2	55266				RESERVED				
D7E3	55267				RESERVED				

- **CALCEN** Enable committing of output values from math units back to math registers (clearing effectively pauses iterative formulae)
- **LATCHINT** Latch interval for latched outputs (in CPU cycles)
- **MATHINO** Math unit 32-bit input 0
- **MATHIN1** Math unit 32-bit input 1
- **MATHIN10** Math unit 32-bit input 10

- **MATHIN11** Math unit 32-bit input 11
- **MATHIN12** Math unit 32-bit input 12
- **MATHIN13** Math unit 32-bit input 13
- **MATHIN14** Math unit 32-bit input 14
- **MATHIN15** Math unit 32-bit input 15
- **MATHIN2** Math unit 32-bit input 2
- **MATHIN3** Math unit 32-bit input 3
- **MATHIN4** Math unit 32-bit input 4
- **MATHIN5** Math unit 32-bit input 5
- **MATHIN6** Math unit 32-bit input 6
- **MATHIN7** Math unit 32-bit input 7
- **MATHIN8** Math unit 32-bit input 8
- **MATHIN9** Math unit 32-bit input 9
- **MULTINA** Multiplier input A (25 bit)
- **MULTINB** Multiplier input A (18 bit)
- **MULTOUT** 48-bit output of $\text{MULTINA} \times \text{MULTINB}$
- **RESERVED** Reserved
- **UNIT0INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 0.
- **UNIT0INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 0.
- **UNIT0OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 0
- **UNIT10INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 10.
- **UNIT10INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 10.
- **UNIT10OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit A
- **UNIT11INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 11.

- **UNIT11INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 11.
- **UNIT11OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit B
- **UNIT12INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 12.
- **UNIT12INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 12.
- **UNIT12OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit C
- **UNIT13INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 13.
- **UNIT13INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 13.
- **UNIT13OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit D
- **UNIT14INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 14.
- **UNIT14INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 14.
- **UNIT14OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit E
- **UNIT15INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 15.
- **UNIT15INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 15.
- **UNIT15OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit F
- **UNIT1INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 1.
- **UNIT1INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 1.
- **UNIT1OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 1

- **UNIT2INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 2.
- **UNIT2INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 2.
- **UNIT2OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 2
- **UNIT3INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 3.
- **UNIT3INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 3.
- **UNIT3OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 3
- **UNIT4INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 4.
- **UNIT4INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 4.
- **UNIT4OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 4
- **UNIT5INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 5.
- **UNIT5INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 5.
- **UNIT5OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 5
- **UNIT6INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 6.
- **UNIT6INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 6.
- **UNIT6OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 6
- **UNIT7INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 7.
- **UNIT7INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 7.

- **UNIT7OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 7
- **UNIT8INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 8.
- **UNIT8INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 8.
- **UNIT8OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 8
- **UNIT9INA** Select which of the 16 32-bit math registers is input A for Math Function Unit 9.
- **UNIT9INB** Select which of the 16 32-bit math registers is input B for Math Function Unit 9.
- **UNIT9OUT** Select which of the 16 32-bit math registers receives the output of Math Function Unit 9
- **WREN** Enable setting of math registers (must normally be set)

MEGA65 HYPERVISOR MODE

Reset

On power-up or reset, the MEGA65 starts up in hypervisor mode, and expects to find a program in the 16KiB hypervisor memory, and begins executing instructions at address \$8100. Normally a JMP instruction will be located at this address, that will jump into a reset routine. That is, the 45GS02 does not use the normal 6502 reset vector. Its function is emulated by the Hypo hypervisor program, which fetches the address from the 6502 reset vector in the loaded client operating system when exiting hypervisor mode.

The hypervisor memory is automatically mapped on reset to \$8000 - \$BFFF. This special memory is not able to be mapped or accessed, except when in hypervisor mode. This includes from the serial monitor/debugger interface. This is to protect it from accidental or malicious access from a guest operating system.

Entering / Exiting Hypervisor Mode

Entering the Hypervisor occurs whenever any of the following events occurs:

- **Power-on** When the MEGA65 is first powered on.
- **Reset** If the reset line is lowered, or a watch-dog triggered reset occurs.
- **SYSCALL register accessed** The registers \$D640 - \$D67F in the MEGA65 IO context trigger SYSCALLs when accessed. This is intended to be the mechanism by which a client operating system or process requests the attention of the hypervisor or operating system.
- **Page Fault** On MEGA65s that feature virtual memory, a page fault will cause a trap to hypervisor mode.
- **Certain keyboard events** Pressing the **RESTORE** key for >0.5 seconds, or the **ALT** + **TAB** key combination traps to the hypervisor. Typically the first is used to launch the freeze menu an the second to toggle the display of debug interface.
- **Accessing virtualised IO devices** For example, if the F011 (internal 3.5" disk drive controller) has been virtualised, then attempting to read or write sectors using this device will cause traps to the hypervisor.
- **Executing an instruction that would lock up the CPU** A number of undocumented opcodes on the 6502 will cause the CPU to lockup. On the MEGA65, instead of locking up, the computer will trap to the hypervisor. This could be used to implement alternative instruction behaviours, or simply to tell the user that something bad has happened.
- **Certain special events** Some devices can generate hypervisor-level interrupts. These are implemented as traps to the hypervisor.

The 45GS02 handles all of these in a similar manner internally:

1. The SYSCALL or trap address is calculated, based on the event.
2. The contents of all CPU registers are saved into the virtualisation control registers.
3. The hypervisor mode memory layout is activated, the CPU decimal flag and special purpose registers are all set to appropriate values. The contents of the A,X,Y and Z and most other CPU flags are preserved, so that they can be accessed from the hypervisor's SYSCALL/trap handler routine, without having to load them, thus saving a few cycles for each call.

4. The hypervisor-mode flag is asserted, and the programme counter (PC) register is set to the computed address.

All of the above happens in one CPU cycle, i.e., in 25 nano-seconds. Returning from a SYSCALL or trap consists simply of writing to \$D67F, which requires 125 nano-seconds, for a total overhead of 150 nano-seconds. This gives the MEGA65 SYSCALL performance rivaling – even beating – even the fastest modern computers, where the system call latency is typically hundreds to tens of thousands of cycles [1].

Hypervisor Memory Layout

The hypervisor memory is 16KiB in size. The first 512 bytes are reserved for SYSCALL and system trap entry points, with four bytes for each. For example, the reset entry point is at \$8100 - \$8100 + 3 = \$8100 - \$8103. This allows 4 bytes for an instruction, typically a JMP instruction, followed by a NOP to pad it to 4 bytes.

The full list of SYSCALLs and traps is:

HEX	DEC	Name	Description
8000	32768	SYSCALL00	Syscall 0 entry point
8004	32772	SYSCALL01	Syscall 1 entry point
8008	32776	SYSCALL02	Syscall 2 entry point
800C	32780	SYSCALL03	Syscall 3 entry point
8010	32784	SYSCALL04	Syscall 4 entry point
8014	32788	SYSCALL05	Syscall 5 entry point
8018	32792	SYSCALL06	Syscall 6 entry point
801C	32796	SYSCALL07	Syscall 7 entry point
8020	32800	SYSCALL08	Syscall 8 entry point
8024	32804	SYSCALL09	Syscall 9 entry point
8028	32808	SYSCALL0A	Syscall 10 entry point
802C	32812	SYSCALL0B	Syscall 11 entry point
8030	32816	SYSCALL0C	Syscall 12 entry point
8034	32820	SYSCALL0D	Syscall 13 entry point
8038	32824	SYSCALL0E	Syscall 14 entry point
803C	32828	SYSCALL0F	Syscall 15 entry point
8040	32832	SYSCALL10	Syscall 16 entry point
8044	32836	SECURENTR	Enter secure container trap entry point
8048	32840	SECUREXIT	Leave secure container trap entry point.
804C	32844	SYSCALL13	Syscall 19 entry point

continued ...

...continued

HEX	DEC	Name	Description
8050	32848	SYSCALL14	Syscall 20 entry point
8054	32852	SYSCALL15	Syscall 21 entry point
8058	32856	SYSCALL16	Syscall 22 entry point
805C	32860	SYSCALL17	Syscall 23 entry point
8060	32864	SYSCALL18	Syscall 24 entry point
8064	32868	SYSCALL19	Syscall 25 entry point
8068	32872	SYSCALL1A	Syscall 26 entry point
806C	32876	SYSCALL1B	Syscall 27 entry point
8070	32880	SYSCALL1C	Syscall 28 entry point
8074	32884	SYSCALL1D	Syscall 29 entry point
8078	32888	SYSCALL1E	Syscall 30 entry point
807C	32892	SYSCALL1F	Syscall 31 entry point
8080	32896	SYSCALL20	Syscall 32 entry point
8084	32900	SYSCALL21	Syscall 33 entry point
8088	32904	SYSCALL22	Syscall 34 entry point
808C	32908	SYSCALL23	Syscall 35 entry point
8090	32912	SYSCALL24	Syscall 36 entry point
8094	32916	SYSCALL25	Syscall 37 entry point
8098	32920	SYSCALL26	Syscall 38 entry point
809C	32924	SYSCALL27	Syscall 39 entry point
80A0	32928	SYSCALL28	Syscall 40 entry point
80A4	32932	SYSCALL29	Syscall 41 entry point
80A8	32936	SYSCALL2A	Syscall 42 entry point
80AC	32940	SYSCALL2B	Syscall 43 entry point
80B0	32944	SYSCALL2C	Syscall 44 entry point
80B4	32948	SYSCALL2D	Syscall 45 entry point
80B8	32952	SYSCALL2E	Syscall 46 entry point
80BC	32956	SYSCALL2F	Syscall 47 entry point
80C0	32960	SYSCALL30	Syscall 48 entry point
80C4	32964	SYSCALL31	Syscall 49 entry point
80C8	32968	SYSCALL32	Syscall 50 entry point
80CC	32972	SYSCALL33	Syscall 51 entry point
80D0	32976	SYSCALL34	Syscall 52 entry point
80D4	32980	SYSCALL35	Syscall 53 entry point
80D8	32984	SYSCALL36	Syscall 54 entry point
80DC	32988	SYSCALL37	Syscall 55 entry point
80E0	32992	SYSCALL38	Syscall 56 entry point
80E4	32996	SYSCALL39	Syscall 57 entry point

continued ...

...continued

HEX	DEC	Name	Description
80E8	33000	SYSCALL3A	Syscall 58 entry point
80EC	33004	SYSCALL3B	Syscall 59 entry point
80F0	33008	SYSCALL3C	Syscall 60 entry point
80F4	33012	SYSCALL3D	Syscall 61 entry point
80F8	33016	SYSCALL3E	Syscall 62 entry point
80FC	33020	SYSCALL3F	Syscall 63 entry point
8100	33024	RESET	Power-on/reset entry point
8104	33028	PAGFAULT	Page fault entry point (not currently used)
8108	33032	RESTORKEY	Restore-key long press trap entry point
810C	33036	ALTTABKEY	ALT+TAB trap entry point
8110	33040	VF011RD	F011 virtualised disk read trap entry point
8114	33044	VF011WR	F011 virtualised disk write trap entry point
8118	33048	BREAKPT	CPU breakpoint encountered
811C - 81FB	33048 - 33275	RESERVED	Reserved traps point entry
81FC	33276	CPUKIL	KIL instruction in 6502-mode trap entry point

The remainder of the 16KiB hypervisor memory is available for use by the programmer, but xwill typically use the last 512 bytes for the stack and zero-page, giving an overall memory map as follows:

HEX	DEC	Description
8000 - 81FF	32768 - 33279	SYSCALL and trap entry points
8200 - BDFF	33280 - 48639	Available for hypervisor or operating system program
8E00 - BEFF	48640 - 48895	Processor stack for hypervisor or operating system
8F00 - BFFF	48896 - 49151	Processor zero-page storage for hypervisor or operating system

The stack is used for holding the return address of function calls. The zero-page storage is typically used for holding variables and other short-term storage, as is customary on the 6502.

Hypervisor Virtualisation Control Registers

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D640	54848								REGA
D641	54849								REGX
D643	54851								REGZ
D644	54852								REGB
D646	54854								SPH
D647	54855								PFLAGS
D648	54856								PCL
D649	54857								PCH
D64A	54858								MAPLO
D64B	54859								MAPLO
D64C	54860								MAPHI
D64D	54861								MAPHI
D64E	54862								MAPLOMB
D64F	54863								MAPHIMB
D650	54864								PORT00
D651	54865								PORT01
D652	54866				-			EXSID	VICMODE
D653	54867								DMASRCMB
D654	54868								DMADSTM
D655	54869								DMALADDR
D656	54870								DMALADDR
D657	54871								DMALADDR
D658	54872								DMALADDR
D659	54873				-				VFLOP
D670	54896								GEORAMBASE
D671	54897								GEORAMMASK
D672	54898	-	MATRIXEN					-	
D67C	54908								UARTDATA
D67D	54909								WATCHDOG
D67E	54910								HICKED

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D67F	54911								ENTEREXIT

- **ASCFAST** Hypervisor enable ASC/DIN CAPS LOCK key to enable/disable CPU slow-down in C64/C128/C65 modes
- **CARTEN** Hypervisor enable /EXROM and /GAME from cartridge
- **CPUFAST** Hypervisor force CPU to 48MHz for userland (userland can override via POKE0)
- **DMADSTMB** Hypervisor DMAgic destination MB
- **DMALADDR** Hypervisor DMAgic list address bits 0-7
- **DMASRCMB** Hypervisor DMAgic source MB
- **ENTEREXIT** Writing trigger return from hypervisor
- **EXSID** 0=Use internal SIDs, 1=Use external(1) SIDs
- **F4502** Hypervisor force CPU to 4502 personality, even in C64 IO mode.
- **GEORAMBASE** Hypervisor GeoRAM base address (x MB)
- **GEORAMMASK** Hypervisor GeoRAM address mask (applied to GeoRAM block register)
- **HICKED** Hypervisor already-upgraded bit (writing sets permanently)
- **JMP32EN** Hypervisor enable 32-bit JMP/JSR etc
- **MAPHI** Hypervisor MAPHI register storage (high bits)
- **MAPHIMB** Hypervisor MAPHI mega-byte number register storage
- **MAPLO** Hypervisor MAPLO register storage (high bits)
- **MAPLOMB** Hypervisor MAPLO mega-byte number register storage
- **MATRIXEN** Enable composited Matrix Mode, and disable UART access to serial monitor.
- **PCH** Hypervisor PC-high register storage
- **PCL** Hypervisor PC-low register storage
- **PFLAGS** Hypervisor P register storage

- **PIRQ** Hypervisor flag to indicate if an IRQ is pending on exit from the hypervisor / set 1 to force IRQ/NMI deferral for 1,024 cycles on exit from hypervisor.
- **PNMI** Hypervisor flag to indicate if an NMI is pending on exit from the hypervisor.
- **PORT00** Hypervisor CPU port \$00 value
- **PORT01** Hypervisor CPU port \$01 value
- **REGA** Hypervisor A register storage
- **REGB** Hypervisor B register storage
- **REGX** Hypervisor X register storage
- **REGZ** Hypervisor Z register storage
- **ROMPROT** Hypervisor write protect C65 ROM \$20000-\$3FFF
- **SPH** Hypervisor SPH register storage
- **UARTDATA** (write) Hypervisor write serial output to UART monitor
- **VFLOP** 1=Virtualise SD/Floppy access (usually for access via serial debugger interface)
- **VICMODE** VIC-II/VIC-III/VIC-IV mode select
- **WATCHDOG** Hypervisor watchdog register: writing any value clears the watch dog

Programming for Hypervisor Mode

The easiest way to write a program for Hypervisor Mode on the MEGA65 is to use KickC, which is a special version of C made for writing programs for 6502-class processors. The following example programs are from KickC's supplied examples. KickC produces very efficient code, and directly supports the MEGA65's hypervisor mode quite easily through the use of a linker definition file with the following contents:

```
.file [name="%0.bin", type="bin", segments="XMega65Bin"]
.segmentdef XMega65Bin [segments="Syscall, Code, Data, Stack, Zeropage"]
.segmentdef Syscall [start=$8000, max=$81ff]
.segmentdef Code [start=$8200, min=$8200, max=$bdff]
.segmentdef Data [startAfter="Code", min=$8200, max=$bdff]
.segmentdef Stack [min=$be00, max=$beff, fill]
.segmentdef Zeropage [min=$bf00, max=$bfff, fill]
```

This file instructs KickC's assembler to create a 16KiB file with the 512 byte SYSCALL/trap entry point region at the start, followed by code and data areas, and then the stack and zero-page areas. It enforces the size and location of these fields, and will give an error during compilation if anything is too big to fit.

With this file in place, you can then create a KickC source file that provides data structures for the SYSCALL/trap table, e.g.:

```
// XMega65 Kernal Development Template
// Each function of the kernal is a no-args function
// The functions are placed in the SVSCALLS table surrounded by JMP and NOP
```

```
import "string"
```

```
// Use a linker definition file (put the previous listing into that file)
#pragma link("Mega65hyper.ld")
```

```
// Some definitions of addresses and special values that this program uses
const char* RASTER = 0xd012;
```

```
const char* VIC+MEMORY = 0xd018;
```

```
const char* SCREEN = 0x0400;
```

```
const char* BGCOL = 0xd021;
```

```
const char* COLS = 0xd000;
```

```
const char BLACK = 0;
```

```
const char WHITE = 1;
```

```
// Some text to display
```

```
char[] MESSAGE = "hello world!";
```

```
void Main() {
```

```
    // Initialize screen Memory, and select correct font
```

```
    *VIC+MEMORY = 0x14;
```

```
    // Fill the screen With spaces
```

```
    memset(SCREEN, ' ', 40*25);
```

```
    // Set the colour of every character on the screen to white
```

```
    memset(COLS, WHITE, 40*25);
```

```
    // Print the "Hello world!" message
```

```
    char* sc = SCREEN+40; // Display it one line down on the screen
```

```
    char* msg = MESSAGE; // The messag to display
```

```
    // A simple copy routine to copy the string
```

```
    while(*msg) {
```

```
        *sc++ = *msg++;
```

```
}
```

```
    // Loop forever showing two white lines as raster bars
```

```
    while(true) {
```

```
        if(*RASTER==54 || *RASTER==66) {
```

```
            *BGCOL = WHITE;
```

```
        } else {
```

```
            *BGCOL = BLACK;
```

```
        }
```

```
}
```

```
// Here are a couple sample SVCALL handlers that just display a character on the screen
```

```
void syscall1() {
```

If you save the first listing into a file called mega65hyper.ld, and the second into a file called mega65hyper.kc, you can then compile them using KickC with a command like:

```
kickc -a Mega65hyper
```

It will then produce a file called mega65hyper.bin, which you can then try out on your MEGA65, or run in the Xmega65 emulator with a command like:

```
xmega65 -kickup mega65hyper.bin
```


F APPENDIX

F018-Compatible Direct Memory Access (DMA) Controller

- F018 “DMAgic” DMA Controller
- MEGA65 DMA Controller Extensions

The MEGA65 includes an F018/F018A backward-compatible DMA controller. Unlike in the C65, where the DMA controller exists as a separate chip, it is part of the 45GS02 processor in the MEGA65. However, as the use of the DMA controller is a logically separate topic, it is documented separately in this appendix.

The MEGA65's DMA controller provides several important improvements over the F018/F018A DMAgic chips of the C65:

- **Speed** The MEGA65 performs DMA operations at 40MHz, allowing filling 40MiB or copying 20MiB per second. For example, it is possible to copy a complete 8KiB C64-style bitmap display in about 200 micro-seconds, equivalent to less than four raster lines!
 - **Large Memory Access** The MEGA65's DMA controller allows access to all 256MiB of address space.
 - **Texture Copying Support** The MEGA65's DMA controller can do fractional address calculations to support hardware texture scaling, as well as address striding, to make it possible in principle to simultaneously scale-and-draw a texture from memory to the screen. This would be useful, should anyone be crazy enough to try to implementa Wolfenstein or Doom style-game on the MEGA65.
 - **Transparency/Mask Value Support** The MEGA65's DMA controller can be told to ignore a special value when copying memory, leaving the destination memory contents unchanged. This allows masking of transparent regions when performing a DMA copy, which considerably simplifies blitting of graphics shapes.
- itemPer-Job Option List** A number of options can be configured for each job in a chained list of DMA jobs, for example, selecting F018 or F018B mode, changing the transparency value, fractional address stepping or the source or destination memory region.

F018 “DMAGIC” DMA CONTROLLER

HEX	DEC	Signal	Description
D700	55040	ADDRLSB-TRIG	DMAgic DMA list address LSB, and trigger DMA (when written)
D701	55041	ADDRMSB	DMA list address high byte (address bits 8 - 15).

continued ...

...continued

HEX	DEC	Signal	Description
D702	55042	ADDRBANK	DMA list address bank (address bits 16 - 22). Writing clears \$D704.

MEGA65 DMA CONTROLLER EXTENSIONS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D703	55043				-				EN018B
D704	55044					ADDRMB			
D705	55045					ETRIG			
D70E	55054					ADDRLSB			

- **ADDRLSB** DMA list address low byte (address bits 0 - 7) WITHOUT STARTING A DMA JOB (used by Hypervisor for unfreezing DMA-using tasks)
- **ADDRMB** DMA list address mega-byte
- **EN018B** DMA enable F018B mode (adds sub-command byte)
- **ETRIG** Set low-order byte of DMA list address, and trigger Enhanced DMA job (uses DMA option list)



APPENDIX

VIC-IV Video Interface Controller

- Features
- VIC-II/III/IV Register Access Control
- Video Output Formats, Timing and Compatibility
- Memory Interface
- Hot Registers
- New Modes
- Sprites

- **VIC-II / C64 Registers**
- **VIC-III / C65 Registers**
- **VIC-IV / MEGA65 Specific Registers**

FEATURES

The VIC-IV is a fourth generation Video Interface Controller developed especially for the MEGA65, and featuring very good backwards compatibility with the VIC-II that was used in the C64, and the VIC-III that was used in the C65. The VIC-IV can be programmed as though it were either of those predecessor systems. In addition it supports a number of new features. It is easy to mix older VIC-II/III features with the new VIC-IV features, making it easy to transition from the VIC-II or VIC-III to the VIC-IV, just as the VIC-III made it easy to transition from the VIC-II. Some of the new features and enhancements of the VIC-IV include:

- **Direct access to 384KB RAM** (up from 16KB/64KB with the VIC-II and 128KB with the VIC-IV).
- Support for **32KB of 8-bit Colour/Attribute RAM** (up from 2KB on the VIC-III), to support very large screens.
- **HDTV $720 \times 576 / 800 \times 600$ native resolution** at both 50Hz and 60Hz for **PAL and NTSC**, with **VGA and digital video** output.
- **81MHz pixel clock** (up from ~ 8 MHz with the VIC-II/III), which enables a wide range of new features.
- New 16-colour (16×8 pixels per character cell) and 256-colour (8×8 pixels per character cell) **full-colour text modes**.
- Support for up to **8,192 unique characters in a character set**.
- **Four 256-colour palette banks** (versus the VIC-III's single palette bank), each supporting **23-bit colour depth** (versus the VIC-III's 12-bit colour depth), and which can be rapidly alternated to create even more colourful graphics than is possible with the VIC-III.
- Screen, bitmap, colour and character data can be positioned at any **address with byte-level granularity** (compared with fixed 1KB - 16KB boundaries with the VIC-II/III)
- **Virtual screen dimensioning**, which combined with byte-level data position granularity provides effective **hardware support for scrolling and panning in both X and Y directions**.
- **New sprite modes**: Bitplane modification, **full-colour** (15 foreground colours + transparency) and tiled modes, allowing a wide variety of new and exciting sprite-based effects

- The ability to stack sprites in a bit-planar manner to produce **sprites with up to 256 colours**.
- Sprites can use 64 bits of data per raster line, allowing **sprites to be 64 pixels wide** when using VIC-II/III mono/multi-colour mode, or 16 pixels wide when using the new VIC-IV full-colour sprite mode.
- **Sprite tile mode**, which allows a sprite to be repeated horizontally across an entire raster line, allowing sprites to be used to create animated backgrounds in a memory-efficient manner.
- Sprites can be configured to use a **separate 256-colour palette** to that used to draw other text and graphics, allowing for a more colourful display.
- **Super-extended attribute mode** which uses two screen RAM bytes and two colour RAM bytes per character mode, which supports a wide variety of new features including **alpha-blending/anti-aliasing, hardware kerning/variable-width characters**, hardware horizontal/vertical flipping, alternate palette selection and other powerful features that make it easy to create highly dynamic and colourful displays.
- **Raster-Rewrite Buffer** which allows **hardware-generated pseudo-sprites**, similar to “bobs” on Amiga™ computers, but with the advantage that they are rendered in the display pipeline, and thus do not need to be undrawn and redrawn to animate them.
- **Multiple 8-bit colour playfields** are also possible using the Raster-Rewrite Buffer.

In short, the VIC-IV is a powerful evolution of the VIC-II/III, while retaining the character and distinctiveness of the VIC-series of video controllers.

For a full description of the additional registers that the VIC-IV provides, as well as documentation of the legacy VIC-II and VIC-III registers, refer to the corresponding sections of this appendix. The remainder of the appendix will focus on describing the capabilities and use of many of the VIC-IV’s new features.

VIC-II/III/IV REGISTER ACCESS CONTROL

Because the new features of the VIC-IV are all extensions to the existing VIC-II/III designs, there is no concept of having to select the mode in which the

VIC-IV will operate: It is always in VIC-IV mode. However, for backwards compatibility with software, the many additional registers of the VIC-IV can be hidden, so that it appears to be either a VIC-II or VIC-III. This is done in the same manner that the VIC-III uses to hide its new features from legacy VIC-II software.

The mechanism is the VIC-III write-only KEY register (\$D02F, 53295 decimal). The VIC-III by default conceals its new features until a “knock” sequence is performed. This consists of writing two special values one after the other to \$D02F. The following table summarises the knock sequences supported by the VIC-IV, and indicates which are VIC-IV specific, and which are supported by the VIC-III:

First Value Hex (Decimal)	Second Value Hex (Decimal)	Effect	VIC-IV Specific?
\$00 (0)	\$00 (0)	Only VIC-II registers visible (all VIC-III and VIC-IV new registers are hidden)	No
\$A5 (165)	\$96 (150)	VIC-III new registers visible	No
\$47 (71)	\$53 (83)	Both VIC-III and VIC-IV new registers visible	Yes
\$45 (69)	\$54 (84)	No VIC-II/III/IV registers visible. 45E100 ethernet controller buffers are visible instead	Yes

Detecting VIC-II/III/IV

Detecting which generation of the VIC-II/III/IV a machine is fitted with can be important for programs that support only particular generations, or that wish to vary their graphical display based on the capabilities of the machine. While there are many possibilities for this, the following is a simple and effective method. It relies on the fact that the VIC-III and VIC-IV do not repeat the VIC-II registers throughout the IO address space. Thus while \$D000 and \$D100 are synonymous when a VIC-II is present (or a VIC-III/IV is hiding their additional registers), this is not the case when a VIC-III or VIC-IV is making all of its registers visible. Therefore presence of a VIC-III/IV can be determined by testing whether these two locations are aliases for the same register, or

represent separate registers. The detection sequence consists of using the KEY register to attempt to make either VIC-IV or VIC-III additional registers visible. If either succeeds, then we can assume that the corresponding generation of VIC is installed. As the VIC-IV supports the VIC-III KEY knocks, we must first test for the presence of a VIC-IV. Also, we assume that the MEGA65 starts in VIC-IV mode, even when running C65 BASIC. Thus the test can be done in BASIC from either C64 or C65 mode as follows:

```
0 REM IN C65 MODE WE CANNOT SAFELY WRITE TO 53295, SO WE TEST A DIFFERENT WAY
10 IF PEEK(53272) AND 32 THEN GOTO 65
20 POKE53248,1:POKE53295,71:POKE53295,83
30 POKE53248+256,0:IFPEEK(53248)=1THENPRINT"VIC-IV PRESENT":END
40 POKE53248,1:POKE53295,165:POKE53295,150
50 POKE53248+256,0:IFPEEK(53248)=1THENPRINT"VIC-III PRESENT":END
60 PRINT "VIC-II PRESENT": END

65 REM WE ASSUME WE HAVE A C65 HERE
70 V1=PEEK(53248+80):V2=PEEK(53248+80):V3=PEEK(53248+80)
80 IF V1<>V2 OR V1<>V3 OR V2<>V3 THEN PRINT "VIC-IV PRESENT":END
90 GOTO 40
```

As the MEGA65 is the only C64-class computer that is fitted with a VIC-IV, this can be used as a *de facto* test for the presence of a MEGA65 computer. Detection of a VIC-III can be similarly assumed to indicate the presence of a C65.

VIDEO OUTPUT FORMATS, TIMING AND COMPATIBILITY

The VIC-IV was designed for use in the MEGA65 and related systems, including the MEGAphone family of portable devices. The VIC-IV supports both VGA and digital video output, using a connector for the digital video that accepts most cables intended for connecting HDMI™ compatible devices. It also supports parallel digital video output suitable for driving LCD display panels. Considerable care has been taken to create a common video front-end that supports these three output modes.

For simplicity and accuracy of frame timing for legacy software, the video format is normally based on the HDTV PAL and NTSC $720 \times 576/480$ (576p and 480p) modes using a 27MHz output pixel clock. This is ideal for digital

video and LCD display panels. However not all VGA displays support these modes, especially 720×576 at 50Hz.

In terms of VIC-II and VIC-III backwards compatibility, this display format has several effects that do not cause problems for most programs, but can cause some differences in behaviour:

1. Because the VIC-IV display is progressive rather than interlaced, two physical raster lines are produced for each logical VIC-II or VIC-III raster line. This means that there are either 63 or 65 cycles per logical double raster, rather than per physical 576p/480p physical raster. This can cause some minor visual artefacts, when programs make assumptions about where on a horizontal line the VIC is drawing when, for example, the border or screen colour is changed.
2. The VIC-IV does not follow the behaviour of the VIC-III, which allowed changes in video modes, e.g., between text and bitmap mode, on characters. Nor does it follow the VIC-II's policy of having such changes take effect immediately. Instead, the VIC-IV applies changes at the start of each raster line. This can cause some minor artefacts.
3. The VIC-IV uses a single-raster rendering buffer which is populated using the VIC-IV's internal 8MHz pixel clock, before being displayed using the 27MHz output pixel clock. This means that a raster lines display content tends to be rendered much earlier in a raster line than on either the VIC-II or VIC-III. This can cause some artefacts with displays, particularly in demos that rely on specific behaviour of the VIC-II at particular cycles in a raster line, for example for effects such as VSP or FLI. At present, such effects are unlikely to display correctly on the current revision of the VIC-IV. Improved support for these features is planned for a future revision of the VIC-IV.
4. The 1280×200 and 1280×400 display modes of the VIC-III are not currently supported, as they cannot be meaningfully displayed on any modern monitor, and no software is known to support or use this feature.

Frame Timing

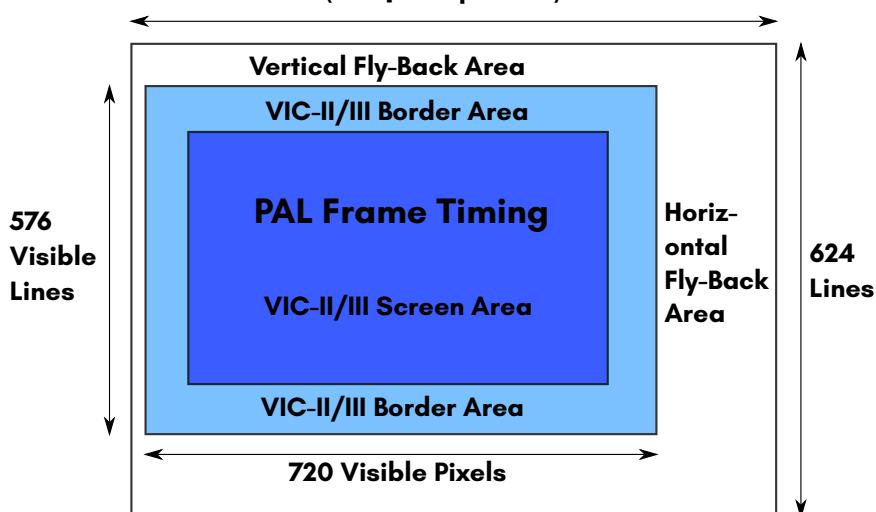
Frame timing is designed to match that of the 6502 + VIC-II combination of the C64. Both PAL and NTSC timing is supported, and the number of cycles per logical raster line, the number of raster lines per frame, and the number of cycles per frame are all adjusted accordingly. To achieve this, the VIC-IV ordinarily uses HDTV 576p 50Hz (PAL) and 480p 60Hz (NTSC) video modes,

with timing tweaked to be as close as possible to double-scan PAL and NTSC composite TV modes as used by the VIC-II.

The VIC-IV produces timing impulses at approximately 1MHz which are used by the 45GS02 processor, so that the correct effective frequency is provided when operating at the 1MHz, 2MHz and 3.5MHz C64, C128 and C65 compatibility modes. This allows the single machine to switch between accurate PAL and NTSC CPU timing, as well as video modes. The exact frequency varies between PAL and NTSC modes, to mimic the behaviour of PAL versus NTSC C64, C128 and C65 processor and video timing.

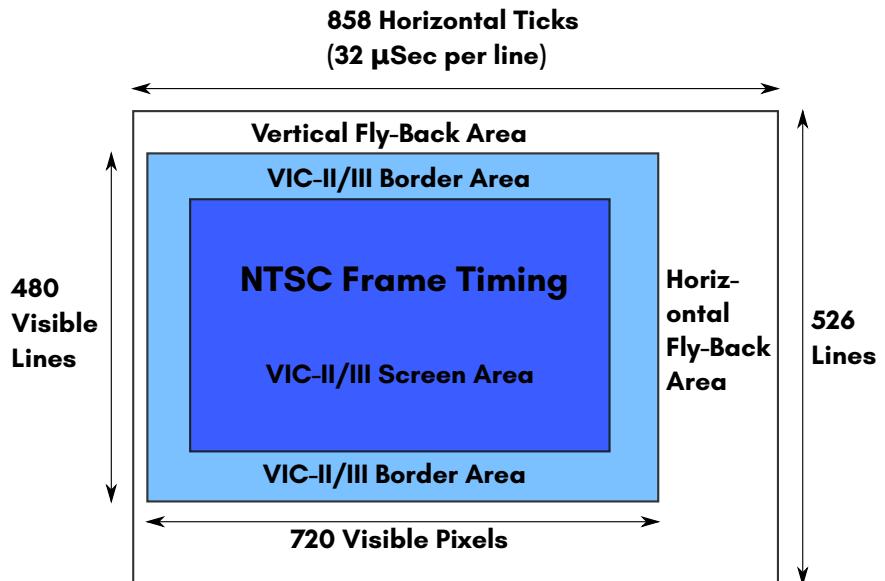
The PAL frame is constructed from 624 physical raster lines, consisting of 864 pixelclock ticks. The pixel clock is 27MHz, which is 1/3 the VIC-IV pixel clock. The visible frame is 720×576 pixels, the entirety of which can be used in VIC-IV mode. In VIC-II and VIC-III modes, the border area reduces the usable size to 640×400 pixels. In VIC-II mode and VIC-III 200H modes, the display is double scanned, with two 31.5 micro-second physical rasters corresponding to a single 63 micro-second VIC-II-style raster line. Thus each frame consists of 312 VIC-II raster lines of 63 micro-seconds each, exactly matching that of a PAL C64.

**864 Horizontal Ticks
(31.5 μ Sec per line)**



The NTSC frame is constructed from 526 physical raster lines, consisting of 858 pixelclock ticks. The pixel clock is 27MHz, which is 1/3 the VIC-IV pixel clock. The visible frame is 720×480 pixels, the entirety of which can be used in VIC-IV mode. In VIC-II and VIC-III modes, the border area reduces the us-

able size to 640×400 pixels. In VIC-II mode and VIC-III 200H modes, the display is double scanned, with two 32 micro-second physical rasters corresponding to a single 64 micro-second VIC-II-style raster line. Thus each frame consists of 263 VIC-II raster lines of 64 micro-seconds each, matching the most common C64 NTSC video timing.



As these HDTV video modes are not supported by all VGA monitors, a compatibility mode is included that provides a 640×480 VGA-style mode. However, as the pixel clock of the MEGA65 is fixed at 27MHz, this mode runs at 63Hz. Nonetheless, this should work on the vast majority of VGA monitors. There should be no problem with the PAL / NTSC modes when using the digital video output of the MEGA65 with the vast majority of HDMI-enabled monitors and TVs.

To determine whether the MEGA65 is operating in PAL or NTSC, you can enter the freeze menu, which displays the current video mode, or from a program you can check the PALNTSC signal (bit 7 of \$D06F, 53359 decimal). If this bit is set, then the machine is operating in NTSC mode, and clear if operating in PAL mode. This bit can be modified to change between the modes, e.g.:

```

10 IFPEEK(53272)<32THENPOKE53295,ASC("6"):POKE53295,ASC("9"):REM ENABLE C65+MEGA65 IO
20 NTSC=PEEK(53359)AND128
30 IFNTSCTHENPRINT"MEGA65 IS IN NTSC MODE"
40 IFNTSC=0THENPRINT"MEGA65 IS IN PAL MODE"
50 INPUT"SWITCH MODES (Y/N)? ",A$
60 IF A$="Y"THENPOKE53359,PEEK(53359)+128-NTSC
70 NTSC=PEEK(53359)AND128
80 IFNTSCTHENPRINT"MEGA65 IS NOW IN NTSC MODE"
90 IFNTSC=0THENPRINT"MEGA65 IS NOW IN PAL MODE"

```

Physical and Logical Rasters

Physical rasters per frame refers to the number of actual raster lines in the PAL or NTSC Enhanced Definition TV (EDTV) video modes used by the MEGA65. Logical Rasters refers to the number of VIC-II-style rasters per frame. Each logical raster consists of two physical rasters per line, since EDTV modes are double-scan modes compared with the original PAL and NTSC Standard Definition TV modes used by the C64. The frame parameters of the VIC-IV for PAL and NTSC are as follows:

Standard	Cycles per Raster	Physical Rasters per Frame	Logical Rasters per Frame
PAL	63	312	626
NTSC	65	263	526

The result is that the frames on the VIC-IV consist of exactly the same number of ~ 1MHz CPU cycles as on the VIC-II exactly.

Bad Lines

The VIC-IV does not natively incur any “bad lines”, because the VIC-IV has its own dedicated memory busses to the main memory and colour RAM of the MEGA65. This means that both the processor and VIC-IV can access the memory at the same time, unlike on the C64 or C65, where they are alternated.

However, to improve compatibility, the VIC-IV signals when a “bad line” would have occurred on the VIC-II. The 45GS02 processor of the MEGA65 accepts these bad line signals, and pauses the CPU for 40 clock cycles, except if the processor is running at full speed, in which case they are ignored. This

improves the timing compatibility with the VIC-II considerably. However, the timing is not exact, because the current revision of the 45GS02 pauses for exactly 40 cycles, instead of 40 – 43 cycles, depending on the instruction being executed at the time. Also, the VIC-IV and 45GS02 do not currently pause for sprite fetches.

The bad line emulation is controlled by bit 0 of \$D710: setting this bit enables bad line emulation, and clearing it prevents any bad line from stealing time from the processor.

MEMORY INTERFACE

The VIC-IV supports up to 64KB of colour RAM and, in principle, 16MB of direct access RAM for video data. However in typical installations 32KB of colour RAM and 384KB of addressable RAM is present. In MEGA65 systems, the second 128KB of RAM is typically used to hold a C65-compatible ROM, leaving 256KB available, unless software is written to avoid the need to use C65 ROM routines, in which case all 384KB can be used.

The VIC-IV supports all legacy VIC-II and VIC-III methods for accessing this RAM, including the VIC-II's use of 16KB banks, and the VIC-III's Display Address Translator (DAT). This additional memory can be used for character and bitmap displays, as well as for sprites. However, the VIC-III bitplane modes remain limited to using only the first 128KB of RAM, as the VIC-IV does not enhance the bitplane mode.

Relocating Screen Memory

To use the additional memory for screen RAM, the screen RAM start address can be adjusted to any location in memory with byte-level granularity by setting the SCRNPTR registers (\$D060 – \$D063, 53344 – 53347 decimal). For example, to set the screen memory to address 12345:

```
IFPEEK(53272)<32THENPOKE53295,ASC("6"):POKE53295,ASC("S"):REM ENABLE C65+MEGA65 IO  
POKE53344+0,69:POKE53344+1,35:POKE53344+2,1
```

Relocating Character Generator Data

The location of the character generator data can also be set with byte-level precision via the CHARPTR registers at \$D068 - \$D06A (53352 - 53354 decimal). As usual, the first of these registers holds the lowest-order byte, and the last the highest-order byte. The three bytes allow for placement of character data anywhere in the first 16MB of RAM. For systems with less than 16MB of RAM accessible by the VIC-IV, the upper address bits should be zero.

For example, to indicate that character generator data should be sourced beginning at \$41200 (266752 decimal), the following could be used. Note that the AND binary operator only works with arguments between 0 and 65,535. Therefore we first subtract $4 \times 65,536 = 262,144$ from the address (the 4 is determined by calculating $\text{INT}(266752/65536)$), before we use the AND operator to compute the lower part of the address:

```
IFPEEK(53272)<32THENPOKE53295,ASC("G"):POKE53295,ASC("S"):REM ENABLE C65+MEGA65 IO  
POKE53352,(266752-INT(266752/65536)*65536)AND255  
POKE53353,INT((266752-INT(266752/65536)*65536)/256)  
POKE53354,INT(266752/65536)
```

Relocating Colour / Attribute RAM

The area of colour RAM being used can be similarly set using the COLPTR registers (\$D064 - \$D065, 53348 - 53349 decimal). That is, the value is an offset from the start of the colour / attribute RAM. This is because, like on the C64, the colour / attribute RAM of the MEGA65 is a separate memory component, with its own dedicated connection to the VIC-IV. By default, the COLPTRs are set to zero, which replicates the behaviour of the VIC-II/III. To set the display to use the colour / attribute RAM beginning at offset 4000, one could use something like:

```
IFPEEK(53272)<32THENPOKE53295,ASC("G"):POKE53295,ASC("S"):REM ENABLE C65+MEGA65 IO  
POKE53348,4000 AND 255  
POKE53349,INT(4000/256)
```

Relocating Sprite Pointers and Images

The location of the sprite pointers can also be moved, and sprites can be made to have their data anywhere in first 4MB of memory. This is accomplished by first setting the location of the sprite pointers by setting the SPRPTRADR registers (\$D06C - \$D06E, 53356 - 53358 decimal, but note that only the bottom 7 bits of \$D06E are used, as the highest bit is used for the SPRPTR16 signal). This allows the list of eight sprite pointers to be moved from the end of screen RAM to an arbitrary location in the first 8MB of RAM. To allow sprites themselves to be located anywhere in the first 4MB of RAM, the SPRPTR16 bit in \$D06E must be set. In this mode, two bytes are used to indicate the location of each sprite, instead of one. That is, the list of sprite pointers will be 16 bytes long, instead of 8 bytes long as on the VIC-II/III. When SPRPTR16 is enabled, the location of the sprite pointers should always be set explicitly via the SPRPTRADR registers. For example, to position the sprite pointers at location 800 - 815, you could use something like the following code. Note that a little gymnastics is required to keep the SPRPTR16 bit unchanged, and also to work around the AND binary operator not working with values greater than 65535:

```
IFPEEK(53272)<32THENPOKE53295,ASC("6"):POKE53295,ASC("S"):REM ENABLE C65+MEGA65 IO  
POKE53356,(800-INT(800/65536)*65536) AND 255  
POKE53357,INT(800/256)AND255  
POKE53358,(PEEK(53358)AND128)+INT(800/65536)
```

The location of each sprite image remains a multiple of 64 bytes, thus allowing for up to 65,536 unique sprite images to be used at any point in time, if the system is equipped with sufficient RAM (4MB or more). In this mode, the VIC-II 16KB banking is ignored, and the location of sprite data is simply $64 \times$ the pointer value. For example, to have the data for a sprite at \$C000 (49152 decimal), this would be sprite location 768, because $49152 \div 64 = 768$. We then need to split 768 into high and low bytes, to set the two pointer bytes: $768 = 256 \times 3$, with remainder 0, so this would require the two sprite pointer bytes to be 0 (low byte, which comes first) and 3 (high byte). Thus if the sprite pointers were located at \$7F8 (2040 decimal), setting the first sprite to sprite image 768 could be done with something like:

```
POKE2040 ,INT(768/256)  
POKE2041 ,768-256*INT(768/256)
```

HOT REGISTERS

Because of the availability of precise vernier registers to set a wide range of video parameters directly, \$D011 (53265 decimal), \$D016 (53272 decimal) and other VIC-II and VIC-III video mode registers are implemented as virtual registers: by default, writing to any of these results in computed consistent values being applied to all of the relevant vernier registers. This means that writing to any of these virtual registers will reset the video mode. Thus some care has to be taken when using new VIC-IV features to not touch any of the “hot” VIC-II and VIC-III registers.

The “hot” registers to be careful with are:

\$D011, \$D016, \$D018, \$D031 (53265, 53272, 53274 and 53287 decimal) and the VIC-II bank bits of \$DD00 (56576 decimal).

If you write to any of those, various VIC-IV registers will need to be re-written with the values you wish to maintain.

This “hot” register behaviour is intended primarily for legacy software. It can be disabled by clearing the HOTREG signal (bit 7 of \$D05D, 53341 decimal).

NEW MODES

Why the new VIC-IV modes are Character and Bitmap modes, not Bitplane modes

The new VIC-IV video modes are derived from the VIC-II character and bitmap modes, rather than the VIC-III bitplane modes. This decision was based on several realities of programming a memory-constrained 8-bit home computer:

1. Bitplanes require that the same amount of memory is given to each area on screen, regardless of whether it is showing empty space, or complex graphics. There is no way with bitplanes to reuse content from within an image in another part of the image. However, most C64 games use highly repetitive displays, with common elements appearing in various places on the screen, of which Boulder Dash and Super Giana Sisters would be good examples.

2. Bitplanes also make it difficult to update a display, because every pixel is unique, in that there is no way to make a change, for example to the animation in an onscreen element, and have it take effect in all places at the same time. The diamond animations in Boulder Dash are a good example of this problem. The requirement to modify multiple separate bytes in each bitplane create an increased computational burden, which is why there were calls for the Amiga AAA chipset to include so-called "chunky" modes, rather than just bitplanar modes. While the Display Address Translator (DAT) and DMAgic of the C65 provide some relief to this problem, the relief is only partial.
3. Scrolling using the C65 bitplanes requires copying the entire bitplane, as the hardware support for smooth scrolling does not extend to changing the bitplane source address in a fine manner. Even using the DMAgic to assist, scrolling a 320×200 256-colour display requires 128,000 clock cycles in the best case (reading and writing $320 \times 200 = 64000$ bytes). At 3.5MHz on the C65 this would require about 36 milli-seconds, or about 2 complete video frames. Thus for smooth scrolling of such a display, a double buffered arrangement would be required, which would consume 128,000 of the 131,072 bytes of memory.

In contrast, the well known character modes of the VIC-II are widely used in games, due to their ability to allow a small amount of screen memory to select which 8×8 block of pixels to display, allowing very rapid scrolling, reduced memory consumption, and effective hardware acceleration of animation of common elements. Thus the focus of improvements in the VIC-IV has been on character mode. As bitmap mode on the VIC-II is effectively a special case of character mode, with implied character numbers, it comes along free for the ride on the VIC-IV, and will only be mentioned in the context of a very few bitmap-mode specific improvements that were trivial to make, and it thus seemed foolish to not implement, in case they find use.

Displaying more than 256 unique characters via "Super-Extended Attribute Mode"

The primary innovation is the addition of the Super-Extended Attribute Mode. The VIC-II already uses 12 bits per character: Each 8×8 cell is defined by 12 bits of data: 8 bits of screen RAM data, by default from \$0400 - \$07E7

(1024 – 2023 decimal), indicating which characters to show, and 4 bits of colour data from the 1K nibble colour RAM at \$D800 – \$DBFF (55296 – 56319 decimal). The VIC-III of the C65 uses 16 bits, as the colour RAM is now 8 bits, instead of 4, with the extra 4 bits of colour RAM being used to support attributes (blink, bold, underline and reverse video). It is recommended to revise how this works, before reading the following. A good introduction to the VIC-II text mode can be found in many places. Super-Extended Attribute mode doubles the number of bits per character used from the VIC-III's 16, to 32: Two bytes of screen RAM and two bytes of colour/attribute RAM.

Super-Extended Attribute Mode is enabled by setting bit 0 in \$D054 (53332 decimal). Remember to first enable VIC-IV mode, to make this register accessible. When this bit is set, two bytes are used for each of the screen memory and colour RAM for each character shown on the display. Thus, in contrast to the 12 bits of information that the C64 uses per character, and the 16 bits that the VIC-III uses, the VIC-IV has 32 bits of information. How those 32 bits are used varies slightly among the particular modes. The default is as follows:

Bit(s)	Function
Screen RAM byte 0	Lower 8 bits of character number, the same as the VIC-II and VIC-III
Screen RAM byte 1, bits 0 – 4	Upper 5 bits of character number, allowing addressing of 8,192 unique characters
Screen RAM byte 1, bits 5 – 7	Trim pixels from right side of character (bits 0 – 2)
Colour RAM byte 0, bit 7	Vertically flip the character
Colour RAM byte 0, bit 6	Horizontally flip the character
Colour RAM byte 0, bit 5	Alpha blend mode (leave 0, discussed later)
Colour RAM byte 0, bit 4	GOTO X (allows repositioning of characters along a raster via the Raster-Rewrite Buffer, discussed later)
Colour RAM byte 0, bits 3	If set, Full-Colour characters use 4 bits per pixel and are 16 pixels wide (less any right side trim bits), instead of using 8 bits per pixel. When using 8 bits per pixels, the characters are the normal 8 pixels wide
Colour RAM byte 0, bits 2	Trim pixels from right side of character (bit 3)
Colour RAM byte 0, bits 0 – 1	Number of pixels to trim from top or bottom of character
Colour RAM byte 1, bits 0 – 3	Low 4 bits of colour of character
continued ...	

...continued

Bit(s)	Function
Colour RAM byte 1, bit 4	Hardware blink of character (if VIC-III extended attributes are enabled)
Colour RAM byte 1, bit 5	Hardware reverse video enable of character (if VIC-III extended attributes are enabled)*
Colour RAM byte 1, bit 6	Hardware bold attribute of character (if VIC-III extended attributes are enabled)*
Colour RAM byte 1, bit 7	Hardware underlining of character (if VIC-III extended attributes are enabled)

* Enabling BOLD and REVERSE attributes at the same time on the MEGA65 selects an alternate palette, effectively allowing 512 colours on screen, but each 8×8 character can use colours only from one 256 colour palette.

We can see that we still have the C64 style bottom 8 bits of the character number in the first screen byte. The second byte of screen memory gets five extra bits for that, allowing $2^{13} = 8,192$ different characters to be used on a single screen. That's more than enough for unique characters covering an 80×50 screen (which is possible to create with the VIC-IV). The remaining bits allow for trimming of the character. This allows for variable width characters, which can be used to do things that would not normally be possible, such as using text mode for free horizontal placement of characters (or parts thereof). This was originally added to provide hardware support for proportional width fonts.

For the colour RAM, the second byte (byte 1) is the same as the C65, i.e., the lower half providing four bits of foreground colour, as on the C64, plus the optional VIC-III extended attributes. The C65 specifications document describes the behaviour when more than one of these are used together, most of which are logical, but there are a few combinations that behave differently than one might expect. For example, combining bold with blink causes the character to toggle between bold and normal mode. Bold mode itself is implemented by effectively acting as bit 4 of the foreground colour value, causing the colour to be drawn from different palette entries than usual.

The C65 / VIC-III attributes and the use of 256 colour 8-bit values for various VIC-II colour registers is enabled by setting bit 5 of \$D031 (53297 decimal). Therefore this is highly recommended when using the VIC-IV mode, as otherwise certain functions will not behave as expected. Note that BOLD+REVERSE together has the meaning of selecting an alternate palette on the MEGA65, which differs from the C65.

Many effects are possible due to Super-Extended Attribute Mode. A few possibilities are explained in the following sub-sections.

Using Super-Extended Attribute Mode

Super-Extended Attribute Mode requires double the screen RAM and colour RAM as the VIC-II/III text modes. This is because two bytes of each are required to define each character, instead of one. The screen RAM can be located anywhere in the 384KiB of main memory using registers \$D060 - \$D062 (53344 - 53346 decimal). The colour RAM can be located anywhere in the 32KiB colour RAM. Only the first 1 or 2 KiB of the colour RAM is visible at \$D800 - \$DBFF or \$D800 - \$DFFF (if the CRAM2K signal is set in bit 0 of \$D030, 53296 decimal). Thus if using a screen larger than 40×25 characters use of the DMA controller or some other means may be required to access the full amount of colour RAM. Thus we will initially discuss using Super-Extender Attribute Mode with a 40x25 character display.

The easiest way to use Super-Extended Attribute Mode is from C65 mode, with the screen set to 80 columns, as the C65 ROM sets up 2KiB for both the screen RAM and colour RAM. The user need only to treat each character pair as a single Super-Extended Attribute character.

The first step is to enable the Super-Extended Attribute Mode by asserting the *FCLRHI* and *CHR16* signals, by setting bits 2 and 0 of \$D054 (53332 decimal). As this is a VIC-IV register, we must first enable the VIC-IV IO mode. The VIC-IV must also be configured to 40 column mode, by clearing the *H640* signal by clearing bit 7 of \$D031 (53297 decimal). This is because each pair of characters will be used to form a single character on screen, thus 80 bytes are required to display 40 characters.

Because pairs of colour RAM and screen RAM bytes are used to define each character, care must be taken to initialise and manipulate the screen. A good approach is to set the text colour to black, because this is colour code 0, and then to fill the screen with @ characters, because that is character code 0. You can then have several ways to manipulate the screen. You can use the normal PRINT command and carefully construct strings that will put the correct values into each screen and colour byte pair. Another approach is to use the BANK and POKE commands to directly set the contents of the screen and colour RAM.

XXX Finish above description

XXX Example program

The following descriptions assume that you have implemented one of the methods described above to set the screen and colour RAM.

Full-Colour (256 colours per character) Text Mode

In normal VIC-II/III text mode, one byte is used for each row of pixels in a character. As a reminder for how those modes work, in hi-res mode, each pixel is either the background or foreground colour, based on the state of one bit in the byte. Multi-colour mode uses two bits to select between four possible colours, but as there are still only 8 bits to describe each row of 8 pixels, each pair of pixels has the same colour.

The VIC-IV's full-colour text mode removes these limitations, and allows each pixel of a character to be chosen from the 256 colour of either the primary or alternate palette bank. To do this, each character now requires 64 bytes of data. Also, XXX

Many-colour (16 colours per character) Text Mode

XXX

Alpha-Blending / Anti-Aliasing

XXX

Flipping Characters

XXX

Variable Width Fonts

There are 4 bits that allow trimming pixels from the right edge of characters when they are displayed. This has the effect of making characters narrower. This can be useful for making more attractive text displays, where narrow characters, such as "i" take less space than wider characters, such as "m", without having to use a bitmap display. This feature can be used to make it very efficient to display such variable-width text displays - both in terms of memory usage and processing time.

This feature can be combined with full-colour text mode, alpha blending mode and 4-bits per pixel mode to allow characters that consist of 15 levels of intensity between the background and foreground colour, and that are up to 16 pixels wide. Further, the GOTO bit can be used to implement negative kerning, so that character pairs like A and T do not have excessive white space between them when printed adjacently. The prudent use of these features can result in highly impressive text display, similar to that on modern systems, but that are still efficient enough to be implemented on a relatively constrained system such as the MEGA65. The "MegaWAT!?" presentation software for the MEGA65 uses several of these features to produce its attractive anti-aliased proportional text display on slides.

XXX Example program

Raster-Rewrite Buffer

If the GOTO bit is set for a character in Super-Extended Attribute Mode, instead of painting a character, the position on the raster is back-tracked (or advanced forward to) the pixel position specified in the low 10 bits of the screen memory bytes. If the vertical flip bit is set, then this has the alternate meaning of preventing the background colour from being painted. This combination can be used to print text material over the top of other text material, providing a crude supplement to the 8 hardware sprites. The amount of material is limited only by the raster time of the VIC-IV. Some experimentation will be required to determine how much can be achieved in PAL and NTSC modes.

This ability to draw multiple layers of text and graphics is highly powerful. For example, it can be used to provide multiple overlapping layers of separately scrollable graphics. This gives many of the advantages of bitplane-based playfields on other computers, such as the Amiga, but without the disadvantages of bitplanes.

SPRITES

VIC-II/III Sprite Control

The control of sprites for C64 / VIC-II/III compatibility is unchanged from the C64. The only practical differences are very minor. In particular the VIC-IV uses ring-buffer for each sprite's data when rendering a raster. This means that a sprite can be displayed multiple times per raster line, thus potentially allowing for horizontal multiplexing.

Extended Sprite Image Sets

On the VIC-II and VIC-III, all sprites must draw their image data from a single 16KB region of memory at any point in time. This limits the number of different sprite images to 256, because each sprite image occupies 64 bytes. In practice, the same 16KB region must also contain either bitmap, text or bitplane data, considerably reducing the number of sprite images that can be used at the same time.

The VIC-IV removes this limitation, by allowing sprite data to be placed anywhere in memory, although still on 64-byte boundaries. This is done by setting the SPRPTR16 signal (bit 7, \$D06E, decimal 53358), which tells the VIC-IV to expect two bytes per sprite pointer instead of one. These addresses are then absolute addresses, and ignore the 16KB VIC-II bank selection logic. Thus 16 bytes are required instead of 8 bytes. The list of pointers can also be placed anywhere in memory by setting the SPRPTRADR (\$D06C - \$D06D, 53356 - 53357 decimal) and SPRPTRBNK signals (bits 0 - 6, \$D06E, 53358 decimal). This allows for sprite data to be located anywhere in the first 4MB of RAM, and the sprite pointer list to be located anywhere in the first 8MB of RAM. Note that typical installations of the VIC-IV have only 384KB of connected RAM, so these limitations are of no practical effect. However, the upper bits of the SPRPTRBNK signal should be set to zero to avoid forward-compatibility problems.

One reason for supporting more sprite images is that sprites on the VIC-IV can require more than one 64 byte image slot. For example, enabling Extra-Wide Sprite Mode means that a sprite will require $8 \times 21 = 168$ bytes, and will thus

occupy four VIC-II style 64 byte sprite image slots. If variable height sprites are used, this can grow to as much as $8 \times 255 = 2,040$ bytes per sprite.

Variable Sprite Size

Sprites can be one of three widths with the VIC-IV:

1. Normal VIC-II width (24 pixels wide).
2. Extra Wide, where 64 bits (8 bytes) of data are used per raster line, instead of the VIC-II's 24. This results in sprites that are 64 pixels wide, unless Full-Colour Sprite Mode is selected for a sprite, in which case the sprite will be $64 / 8 = 8$ pixels wide.
3. Tiled mode, where the sprite is drawn repeatedly until the end of the raster line.

To enable a sprite to be 64 pixels (or 16 pixels if in Full-Colour Sprite Mode), set the corresponding bit for the sprite in the SPRX64EN register at (\$D057, 53335 decimal).

Similarly, sprites can be various heights: Sprites will be either the 21 pixels high of the VIC-II, or if the corresponding bit for the sprite is enabled in the SPRHGTEN signal (\$D055, 53333 decimal), then that sprite will be the number of pixels tall that is set in the SPRHGT register (\$D056, 53334 decimal).

Variable Sprite Resolution

By default, sprites are the same resolution as on the VIC-II, i.e., each sprite pixel is two physical pixels wide and high. However, sprites can be made to use the native resolution, where sprite pixels are one physical pixel wide and/or high. This is achieved by setting the relevant bit for the sprite in the SPRENV400 (\$D076, 53366 decimal) registers to increase the vertical resolution on a sprite-by-sprite basis. The horizontal resolution for all sprites is either the normal VIC-II resolution, or if the SPR640 signal is set (bit 4 of \$D054, 53332 decimal), then sprites will have the same horizontal resolution as the physical pixels of the display.

Sprite Palette Bank

The VIC-IV has four palette banks, compared with the single palette bank of the VIC-III. The VIC-IV allows the selection of separate palette banks for

bitmap/text graphics and for sprites. This makes it easy to have very colourful displays, where the sprites have different colours to the rest of the display, or to use palette animation to achieve interesting visual effects in sprites, without disturbing the palette used by other elements of the display.

The sprite palette bank is selected by setting the SPRPALSEL signal in bits 2 and 3 of the register \$D070 (53360 decimal). It is possible to set this to the same bank as the bitmap/text display, or to select a different palette bank. Palette bank selection takes effect immediately. Don't forget that to be able to modify a palette, you have to also bank it to be the palette accessible via the palette bank registers at \$D100 - \$D3FF by setting the MAPEDPAL signal in bits 6 and 7 of \$D070.

Full-Colour Sprite Mode

In addition to monochrome and multi-colour modes, the VIC-IV supports a new full-colour sprite mode. In this mode, four bits are used to encode each sprite pixel. However, unlike multi-colour mode where pairs of bits encode pairs of pixels, in full-colour mode the pixels remain at their normal horizontal resolution. The colour zero is considered transparent. If you wish to use black in a full-colour sprite, you must configure the palette bank that is selected for sprites so that one of the 15 colours for the specific sprite encodes black.

Full-colour sprite mode is selectable for each sprite by setting the appropriate bit in the SPR16EN register (\$D06B, 53355 decimal).

To enable the eight sprites to have 15 unique colours each, the sprite colour is drawn using the palette entry corresponding to: $\textit{spritenumber} \times 16 + \textit{nibblevalue}$, where *spritenumber* is the number of the sprite (from 0 to 7), and *nibblevalue* is the value of the half-byte that contains the sprite data for the pixel. In addition, if bitplane mode is enabled for this sprite, then 128 is added to the colour value, which makes it easy to switch between two colour schemes for a given sprite by changing only one bit in the SPRBPMEN register.

Because Full-Colour Sprite Mode requires four bits per pixel, sprites will be only six pixels wide, unless Extra Wide Sprite Mode is enabled for a sprite, in which case the sprite will be 16 pixels wide. Tiled Mode also works with Full-Colour Sprite Mode, and will result in the 16 full-colour pixels of the sprite being repeated until the end of the raster line.

The following BASIC program draws a Full-Colour Sprite in either C64 or C65 mode:

```
0 AD=56*64:IF PEEK(53272) AND 32 THEN GOTO 30:REM C65/C64 MODE DETECT
```

```
10 POKE 53295,ASC("G"):POKE 53295,ASC("S"): REM ENABLE MEGA65 VIC-IV FEATURES
20 AD=768+64: REM $0340 HEX FOR SPRITE
30 FOR I=AD TO AD+168:POKEI,0:NEXT I
40 POKE 2040,AD/64: REM SET SPRITE NUMBER
50 POKE 53269,1: REM ENABLE SPRITE 0
60 POKE 53248,100:POKE 53249,100: REM PUT SPRITE ON SCREEN
70 POKE 53355,1: REM MAKE SPRITE 0 16-COLOUR
80 POKE 53335,1: REM MAKE SPRITE 0 USE 64 BITS OF DATA = 16 X 4-BIT PIXELS
90 POKE 53287,10: REM MAKE PINK THE TRANSPARENT COLOUR
100 GOSUB 900: REM READ MULTI-COLOUR SPRITE

899 END

900 REM LOAD SPRITE
910 READN$:IFN$="END"THEN RETURN
920 GOSUB 1000
930 GOTO 910

1000 REM DECODE STRING OF NIBBLES IN N$ AT ADDRESS AD
1010 L=LEN(N$)
1020 FOR I=1 TO (L/2+1):POKE AD+I,0
1030 FOR I= 1 TO L:N=ASC(MID$(N$,I,1))-ASC("0")
1040 A=AD+INT((I-1)/2):IF (I AND 1)=1 THEN N=N*16
1050 V=PEEK(A):POKE A,V OR N:NEXTI
1060 AD=AD+INT(I/2)
1070 IF (L AND 1) THEN AD=AD+1
1080 RETURN

1998 REM SPRITE DATA FOLLOWS
1999 REM 0 = TRANSPARENT, A-0 = COLOURS 1 TO 15
2000 DATA "CABCD EFGHIJKLMNOP"
2010 DATA "A A CCCCCCCCCCCCCCCC"
2020 DATA "A CCCCCCCCCCCCCCCC"
2030 DATA "A C CCCCCCCCCCCCCCCC"
2040 DATA "A C C CCCCCCCCCCCC"
2050 DATA "A C C C CCCCCCCCCCCC"
2060 DATA "A C C C C CCCCCCCC"
2070 DATA "A C C C C C CCCCCCCC"
2080 DATA "A C C C C C C CCCCCC"
2090 DATA "A C C C C C C C CCCC"
2100 DATA "A C C C C C C C CCCC"
2110 DATA "A C C C C C C C CCCC"
```

```

2120 DATA "ACcccccccccccccccc"
2130 DATA "ACcccccccccccccccc"
2140 DATA "ACcccccccccccccccc"
2150 DATA "ACcccccccccccccccc"
2160 DATA "ACcccccccccccccccc"
2170 DATA "ACcccccccccccccccc"
2180 DATA "ACcccccccccccccccc"
2190 DATA "ACcccccccccccccccc"
2200 DATA "ACcccccccccccccccc"
2210 DATA "END"

```

VIC-II / C64 REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D000	53248							S0X	
D001	53249							S0Y	
D002	53250							S1X	
D003	53251							S1Y	
D004	53252							S2X	
D005	53253							S2Y	
D006	53254							S3X	
D007	53255							S3Y	
D008	53256							S4X	
D009	53257							S4Y	
D00A	53258							S5X	
D00B	53259							S5Y	
D00C	53260							S6X	
D00D	53261							S6Y	
D00E	53262							S7X	
D00F	53263							S7Y	
D010	53264							SXMSB	
D011	53265	RC	ECM	BMM	BLNK	RSEL		YSCL	
D012	53266						RC		
D013	53267						LPX		
D014	53268						LPY		
D015	53269						SE		
D016	53270	-	RST	MCM	CSEL			XSCL	

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D017	53271						SEXY		
D018	53272		VS			CB			-
D019	53273			-			ISSC	ISBC	RIRO
D01A	53274			-			MISSC	MISBC	MRIRO
D01B	53275				BSP				
D01C	53276				SCM				
D01D	53277				SEXX				
D01E	53278				SSC				
D01F	53279				SBC				
D020	53280		-			BORDERCOL			
D021	53281		-			SCREENCOL			
D022	53282		-			MC1			
D023	53283		-			MC2			
D024	53284		-			MC3			
D025	53285				SPRMC0				
D026	53286				SPRMC1				
D027	53287				SPR0COL				
D028	53288				SPR1COL				
D029	53289				SPR2COL				
D02A	53290				SPR3COL				
D02B	53291				SPR4COL				
D02C	53292				SPR5COL				
D02D	53293				SPR6COL				
D02E	53294				SPR7COL				
D030	53296				-			C128FAST	

- **BLNK** disable display
- **BMM** bitmap mode
- **BORDERCOL** display border colour (16 colour)
- **BSP** sprite background priority bits
- **C128FAST** 2MHz select (for C128 2MHz emulation)
- **CB** character set address location (\times 1KiB)
- **CSEL** 38/40 column select
- **ECM** extended background mode
- **ISBC** sprite:bitmap collision indicate or acknowledge

- **ISSC** sprite:sprite collision indicate or acknowledge
- **LPX** Coarse horizontal beam position (was lightpen X)
- **LPY** Coarse vertical beam position (was lightpen Y)
- **MC1** multi-colour 1 (16 colour)
- **MC2** multi-colour 2 (16 colour)
- **MC3** multi-colour 3 (16 colour)
- **MCM** Multi-colour mode
- **MISBC** mask sprite:bitmap collision IRQ
- **MISSC** mask sprite:sprite collision IRQ
- **MRIRQ** mask raster IRQ
- **RC** raster compare bit 8
- **RIRQ** raster compare indicate or acknowledge
- **RSEL** 24/25 row select
- **RST** Disables video output on MAX Machine(tm) VIC-II 6566. Ignored on normal C64s and the MEGA65
- **SOX** sprite 0 horizontal position
- **SOY** sprite 0 vertical position
- **S1X** sprite 1 horizontal position
- **S1Y** sprite 1 vertical position
- **S2X** sprite 2 horizontal position
- **S2Y** sprite 2 vertical position
- **S3X** sprite 3 horizontal position
- **S3Y** sprite 3 vertical position
- **S4X** sprite 4 horizontal position
- **S4Y** sprite 4 vertical position
- **S5X** sprite 5 horizontal position
- **S5Y** sprite 5 vertical position
- **S6X** sprite 6 horizontal position
- **S6Y** sprite 6 vertical position

- **S7X** sprite 7 horizontal position
- **S7Y** sprite 7 vertical position
- **SBC** sprite/foreground collision indicate bits
- **SCM** sprite multicolour enable bits
- **SCREENCOL** screen colour (16 colour)
- **SE** sprite enable bits
- **SEXH** sprite horizontal expansion enable bits
- **SEXY** sprite vertical expansion enable bits
- **SPR0COL** sprite 0 colour / 16-colour sprite transparency colour (lower nybl)
- **SPR1COL** sprite 1 colour / 16-colour sprite transparency colour (lower nybl)
- **SPR2COL** sprite 2 colour / 16-colour sprite transparency colour (lower nybl)
- **SPR3COL** sprite 3 colour / 16-colour sprite transparency colour (lower nybl)
- **SPR4COL** sprite 4 colour / 16-colour sprite transparency colour (lower nybl)
- **SPR5COL** sprite 5 colour / 16-colour sprite transparency colour (lower nybl)
- **SPR6COL** sprite 6 colour / 16-colour sprite transparency colour (lower nybl)
- **SPR7COL** sprite 7 colour / 16-colour sprite transparency colour (lower nybl)
- **SPRMCO** Sprite multi-colour 0 (always 256 colour)
- **SPRMCI** Sprite multi-colour 1 (always 256 colour)
- **SSC** sprite/sprite collision indicate bits
- **SXMSB** sprite horizontal position MSBs
- **VS** screen address (\times 1KiB)
- **XSCL** horizontal smooth scroll
- **YSCL** 24/25 vertical smooth scroll

VIC-III / C65 REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D020	53280						BORDERCOL		
D021	53281					SCREENCOL			
D022	53282					MC1			
D023	53283					MC2			
D024	53284					MC3			
D025	53285					SPRMC0			
D026	53286					SPRMC1			
D02F	53295					KEY			
D030	53296	ROME	CROM9	ROMC	ROMA	ROM8	PAL	EXTSYNC	CRAM2K
D031	53297	H640	FAST	ADDR	BPM	V400	H1280	MONO	INT
D033	53299	B0ADODD			-		B0ADEVN		-
D034	53300	B1ADODD			-		B1ADEVN		-
D035	53301	B2ADODD			-		B2ADEVN		-
D036	53302	B3ADODD			-		B3ADEVN		-
D037	53303	B4ADODD			-		B4ADEVN		-
D038	53304	B5ADODD			-		B5ADEVN		-
D039	53305	B6ADODD			-		B6ADEVN		-
D03A	53306	B7ADODD			-		B7ADEVN		-
D03B	53307					BPCOMP			
D03C	53308					BPX			
D03D	53309					BPY			
D03E	53310					HPOS			
D03F	53311					VPOS			
D040	53312					B0PIX			
D041	53313					B1PIX			
D042	53314					B2PIX			
D043	53315					B3PIX			
D044	53316					B4PIX			
D045	53317					B5PIX			
D046	53318					B6PIX			
D047	53319					B7PIX			
D100 - D1FF	53504 - 53759					PALRED			

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D200 - D2FF	53760 - 54015					PALGREEN			
D300 - D3FF	54016 - 54271					PALBLUE			

- **ADDR** Enable extended attributes and 8 bit colour entries
- **B0ADEVN** - Bitplane 0 address, even lines
- **B0ADODD** - Bitplane 0 address, odd lines
- **B0PIX** Display Address Translater (DAT) Bitplane 0 port
- **B1ADEVN** - Bitplane 1 address, even lines
- **B1ADODD** - Bitplane 1 address, odd lines
- **B1PIX** Display Address Translater (DAT) Bitplane 1 port
- **B2ADEVN** - Bitplane 2 address, even lines
- **B2ADODD** - Bitplane 2 address, odd lines
- **B2PIX** Display Address Translater (DAT) Bitplane 2 port
- **B3ADEVN** - Bitplane 3 address, even lines
- **B3ADODD** - Bitplane 3 address, odd lines
- **B3PIX** Display Address Translater (DAT) Bitplane 3 port
- **B4ADEVN** - Bitplane 4 address, even lines
- **B4ADODD** - Bitplane 4 address, odd lines
- **B4PIX** Display Address Translater (DAT) Bitplane 4 port
- **B5ADEVN** - Bitplane 5 address, even lines
- **B5ADODD** - Bitplane 5 address, odd lines
- **B5PIX** Display Address Translater (DAT) Bitplane 5 port
- **B6ADEVN** - Bitplane 6 address, even lines
- **B6ADODD** - Bitplane 6 address, odd lines
- **B6PIX** Display Address Translater (DAT) Bitplane 6 port

- **B7ADEVN** - Bitplane 7 address, even lines
- **B7ADODD** - Bitplane 7 address, odd lines
- **B7PIX** Display Address Translator (DAT) Bitplane 7 port
- **BORDERCOL** display border colour (256 colour)
- **BPCOMP** Complement bitplane flags
- **BPM** Bit-Plane Mode
- **BPX** Bitplane X
- **BPY** Bitplane Y
- **CRAM2K** Map 2nd KB of colour RAM \$DC00-\$DFFF
- **CROM9** Select between C64 and C65 charset.
- **EXTSYNC** Enable external video sync (genlock input)
- **FAST** Enable C65 FAST mode (3 .5MHz)
- **H1280** Enable 1280 horizontal pixels (not implemented)
- **H640** Enable C64 640 horizontal pixels / 80 column mode
- **HPOS** Bitplane X Offset
- **INT** Enable VIC-III interlaced mode
- **KEY** Write A5 then 96 to enable C65/VIC-III IO registers
- **MC1** multi-colour 1 (256 colour)
- **MC2** multi-colour 2 (256 colour)
- **MC3** multi-colour 3 (256 colour)
- **MONO** Enable VIC-III MONO video output (not implemented)
- **PAL** Use PALETTE ROM or RAM entries for colours 0 - 15
- **PALBLUE** blue palette values (reversed nybl order)
- **PALGREEN** green palette values (reversed nybl order)
- **PALRED** red palette values (reversed nybl order)
- **ROM8** Map C65 ROM \$8000
- **ROMA** Map C65 ROM \$A000
- **ROMC** Map C65 ROM \$C000

- **ROME** Map C65 ROM \$E000
- **SCREENCOL** screen colour (256 colour)
- **SPRMC0** Sprite multi-colour 0 (8-bit for selection of any palette colour)
- **SPRMC1** Sprite multi-colour 1 (8-bit for selection of any palette colour)
- **V400** Enable 400 vertical pixels
- **VPOS** Bitplane Y Offset

VIC-IV / MEGA65 SPECIFIC REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D020	53280				BORDERCOL				
D021	53281				SCREENCOL				
D022	53282				MC1				
D023	53283				MC2				
D024	53284				MC3				
D025	53285				SPRMC0				
D026	53286				SPRMC1				
D02F	53295				KEY				
D048	53320				TBDRPOS				
D049	53321			SPRBPMEN			TBDRPOS		
D04A	53322				BBDRPOS				
D04B	53323			SPRBPMEN			BBDRPOS		
D04C	53324				TEXTXPOS				
D04D	53325			SPRTILEN			TEXTXPOS		
D04E	53326				TEXTYPOS				
D04F	53327			SPRTILEN			TEXTYPOS		
D050	53328	-			XPOS				
D051	53329				XPOS				
D052	53330				FNRASTER				
D053	53331	FNRST		-			FNRASTER		
D054	53332	ALPHEN	VFAST	PALEMU	SPR640	SMTM	FCLRHI	FCLRLO	CHR16
D055	53333				SPRHGTEN				
D056	53334				SPRHGHT				
D057	53335				SPRX64EN				

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D058	53336								CHARSTEP
D059	53337								CHARSTEP
D05A	53338								CHRXSCL
D05B	53339								CHRYSCL
D05C	53340								SIDBDRWD
D05D	53341	HOTREG	RSTDELEN						SIDBDRWD
D05E	53342								CHRCOUNT
D05F	53343								SPRXSMSBS
D060	53344								SCRNPTR
D061	53345								SCRNPTR
D062	53346								SCRNPTR
D063	53347								SCRNPTR
D064	53348								COLPTR
D065	53349								COLPTR
D068	53352								CHARPTR
D069	53353								CHRPTR
D06A	53354								CHRPTR
D06B	53355								SPR16EN
D06C	53356								SPRPTRADR
D06D	53357								SPRPTRADR
D06E	53358	SPRPTR16							SPRPTRBNK
D06F	53359	PALNTSC	-						RASLINE0
D070	53360	MAPEDPAL		BTPALSEL		SPRPALSEL		ABTPALSEL	
D071	53361								BP16ENS
D072	53362								VSYNDEL
D073	53363		RASTERHEIGHT						ALPHADELAY
D074	53364								SPRENALPHA
D075	53365								SPRALPHAVAL
D076	53366								SPRENV400
D077	53367								SRPYMSBS
D078	53368								SPRYSMSBS
D079	53369								RSTCOMP
D07A	53370	FNRSTCMP		RESERVED					RSTCMP
D07B	53371			RESERVED					
D07C	53372	RESERVED		VSYNCP	HYNCP				RESERVED

- **ABTPALSEL** VIC-IV bitmap/text palette bank (alternate palette)
- **ALPHADELAY** Alpha delay for compositor

- **ALPHEN** Alpha compositor enable
- **BBDRPOS** bottom border position
- **BORDERCOL** display border colour (256 colour)
- **BP16ENS** VIC-IV 16-colour bitplane enable flags
- **BTPALSEL** bitmap/text palette bank
- **CHARPTR** Character set precise base address (bits 0 - 7)
- **CHARSTEP** characters per logical text row (LSB)
- **CHR16** enable 16-bit character numbers (two screen bytes per character)
- **CHRCOUNT** Number of characters to display per row
- **CHRPTR** Character set precise base address (bits 15 - 8)
- **CHRXSCL** Horizontal hardware scale of text mode (pixel 120ths per pixel)
- **CHRYSCL** Vertical scaling of text mode (number of physical rasters per char text row)
- **COLPTR** colour RAM base address (bits 0 - 7)
- **FCLRHI** enable full-colour mode for character numbers >\$FF
- **FCLRLO** enable full-colour mode for character numbers <=\$FF
- **FNRASTER** Read physical raster position
- **FNRST** Raster compare source (1=VIC-IV fine raster, 0=VIC-II raster)
- **FNRSTCMP** Raster compare is in physical rasters if set, or VIC-II raster if clear
- **HOTREG** Enable VIC-II hot registers. When enabled, touching many VIC-II registers causes the VIC-IV to recalculate display parameters, such as border positions and sizes
- **HSYNCP** hsync polarity
- **KEY** Write 47 then 53 to enable C65GS/VIC-IV IO registers
- **MAPEDPAL** palette bank mapped at \$D100-\$D3FF
- **MC1** multi-colour 1 (256 colour)
- **MC2** multi-colour 2 (256 colour)

- **MC3** multi-colour 3 (256 colour)
- **PALEMU** video output pal simulation
- **PALNTSC** NTSC emulation mode (max raster = 262)
- **RASLINEO** first VIC-II raster line
- **RASTERHEIGHT** physical rasters per VIC-II raster (1 to 16)
- **RESERVED**
- **RSTCMP** Raster compare value MSB
- **RSTCOMP** Raster compare value
- **RSTDELEN** Enable raster delay (delays raster counter and interrupts by one line to match output pipeline latency)
- **SCREENCOL** screen colour (256 colour)
- **SCRNPTR** screen RAM precise base address (bits 0 - 7)
- **SIDBDRWD** Width of single side border
- **SMTH** video output horizontal smoothing enable
- **SPR16EN** sprite 16-colour mode enables
- **SPR640** Sprite H640 enable;
- **SPRALPHAVAL** Sprite alpha-blend value
- **SPRBPMEN** Sprite bitplane-modify-mode enables
- **SPRENALPHA** Sprite alpha-blend enable
- **SPRENV400** Sprite V400 enables
- **SPRHGHT** Sprite extended height size (sprite pixels high)
- **SPRHGTEN** sprite extended height enable (one bit per sprite)
- **SPRMCO** Sprite multi-colour 0 (8-bit for selection of any palette colour)
- **SPRMCI** Sprite multi-colour 1 (8-bit for selection of any palette colour)
- **SPRPALSEL** sprite palette bank
- **SPRPTR16** 16-bit sprite pointer mode (allows sprites to be located on any 64 byte boundary in chip RAM)
- **SPRPTRADR** sprite pointer address (bits 7 - 0)
- **SPRPTRBNK** sprite pointer address (bits 22 - 16)

- **SPRTILEN** Sprite horizontal tile enables.
- **SPRX64EN** Sprite extended width enables (8 bytes per sprite row = 64 pixels wide for normal sprites or 16 pixels wide for 16-colour sprite mode)
- **SPRXSMSBS** Sprite H640 X Super-MSBs
- **SPRYSMSBS** Sprite V400 Y position super MSBs
- **SRPYMSBS** Sprite V400 Y position MSBs
- **TBDRPOS** top border position
- **TEXTXPOS** character generator horizontal position
- **TEXTYPOS** Character generator vertical position
- **VFAST** C65GS FAST mode (48MHz)
- **VSYNCP** vsync polarity
- **VSYNDEL** VIC-IV VSYNC delay
- **XPOS** Read horizontal raster scan position

H

APPENDIX

6526 Complex Interface Adapter (CIA) Registers

- CIA 6526 Registers
- CIA 6526 Hypervisor Registers

CIA 6526 REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
DC00	56320								PORTA
DC01	56321								PORTB
DC02	56322								DDRA
DC03	56323								DDRB
DC04	56324								TIMERA
DC05	56325								TIMERA
DC06	56326								TIMERB
DC07	56327								TIMERB
DC08	56328		-						TODJIF
DC09	56329	-							TODSEC
DC0A	56330	-							TODSEC
DC0B	56331	TODAMPM	-						TODHOUR
DC0C	56332					SDR			
DC0D	56333	IR	-		FLG	SP	ALRM	TB	TA
DC0E	56334	TOD50	SPMOD	IMODA	-	RMODA	OMODA	PBONA	STRTA
DC0F	56335	-		IMODB	LOAD	RMODB	OMODB	PBONB	STRTB

- **ALRM** TOD alarm
- **DDRA** Port A DDR
- **DDRB** Port B DDR
- **FLAG** FLAG edge detected
- **IMODA** Timer A Timer A tick source
- **IMODB** Timer B Timer A tick source
- **IR** Interrupt flag
- **LOAD** Strobe input to force-load timers
- **OMODA** Timer A toggle or pulse
- **OMODB** Timer B toggle or pulse
- **PBONA** Timer A PB6 out
- **PBONB** Timer B PB7 out
- **PORTA** Port A

- **PORTB** Port B
- **RMODA** Timer A one-shot mode
- **RMODB** Timer B one-shot mode
- **SDR** shift register data register(writing starts sending)
- **SP** shift register full/empty
- **SPMOD** Serial port direction
- **STRTA** Timer A start
- **STRTB** Timer B start
- **TA** Timer A underflow
- **TB** Timer B underflow
- **TIMERA** Timer A counter (16 bit)
- **TIMERB** Timer B counter (16 bit)
- **TOD50** 50/60Hz select for TOD clock
- **TODAMPM** TOD PM flag
- **TODHOUR** TOD hours
- **TODJIF** TOD tenths of seconds
- **TODSEC** TOD seconds

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
DC0F	56335	TODEDIT					-		
DD00	56576						PORTA		
DD01	56577						PORTB		
DD02	56578						DDRA		
DD03	56579						DDRB		
DD04	56580						TIMERA		
DD05	56581						TIMERA		
DD06	56582						TIMERB		
DD07	56583						TIMERB		
DD08	56584		-					TODJIF	
DD09	56585	-						TODSEC	
DD0A	56586	-						TODSEC	
DD0B	56587	TODAMPM	-					TODHOUR	
DD0C	56588						SDR		

continued ...

...continued

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
DD0D	56589		-		FLG	SP	ALRM	TB	TA
DD0E	56590	TOD50	SPMOD	IMODA	-	RMODA	OMODA	PBONA	STRTA
DD0F	56591	TODEDIT		IMODB	LOAD	RMODB	OMODB	PBONB	STRTB

- **ALRM** TOD alarm
- **DDR_A** Port A DDR
- **DDR_B** Port B DDR
- **FLG** FLAG edge detected
- **IMODA** Timer A Timer A tick source
- **IMODB** Timer B Timer A tick source
- **LOAD** Strobe input to force-load timers
- **OMODA** Timer A toggle or pulse
- **OMODB** Timer B toggle or pulse
- **PBONA** Timer A PB6 out
- **PBONB** Timer B PB7 out
- **PORTA** Port A
- **PORTB** Port B
- **RMODA** Timer A one-shot mode
- **RMODB** Timer B one-shot mode
- **SDR** shift register data register(writing starts sending)
- **SP** shift register full/empty
- **SPMOD** Serial port direction
- **STRTA** Timer A start
- **STRTB** Timer B start
- **TA** Timer A underflow
- **TB** Timer B underflow
- **TIMERA** Timer A counter (16 bit)
- **TIMERB** Timer B counter (16 bit)

- **TOD50** 50/60Hz select for TOD clock
- **TODAMPM** TOD PM flag
- **TODEDIT** TOD alarm edit
- **TODHOUR** TOD hours
- **TODJIF** TOD tenths of seconds
- **TODSEC** TOD seconds

CIA 6526 HYPERVISOR REGISTERS

In addition to the standard CIA registers available on the C64 and C65, the MEGA65 provides an additional set of registers that are visible only when the system is in Hypervisor Mode. These additional registers allow the internal state of the CIA to be more fully extracted when freezing, thus allowing more programs to function correctly after being frozen. They are not visible when using the MEGA65 normally, and can be safely ignored by programmers who are not programming the MEGA65 in Hypervisor Mode.

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
DC10	56336				TALATCH				
DC11	56337				TALATCH				
DC12	56338				TALATCH				
DC13	56339				TALATCH				
DC14	56340				TALATCH				
DC15	56341				TALATCH				
DC16	56342				TALATCH				
DC17	56343				TALATCH				
DC18	56344	IMFLG	IMSP	IMALRM	IMTB				TODJIF
DC19	56345					TODSEC			
DC1A	56346					TODMIN			
DC1B	56347	TODAMPM				TODHOUR			
DC1C	56348					ALRMJIF			
DC1D	56349					ALRMSEC			
DC1E	56350					ALRMMIN			
DC1F	56351	ALRMAMPM				ALRMHOUR			

- **ALRMAMPM** TOD Alarm AM/PM flag
- **ALRMHOUR** TOD Alarm hours value

- **ALRMJIF** TOD Alarm 10ths of seconds value
- **ALRMMIN** TOD Alarm minutes value
- **ALRMSEC** TOD Alarm seconds value
- **IMALRM** Interrupt mask for TOD alarm
- **IMFLG** Interrupt mask for FLAG line
- **IMSP** Interrupt mask for shift register (serial port)
- **IMTB** Interrupt mask for Timer B
- **TALATCH** Timer A latch value (16 bit)
- **TODAMPM** TOD AM/PM flag
- **TODHOUR** TOD hours value
- **TODJIF** TOD 10ths of seconds value
- **TODMIN** TOD Alarm minutes value
- **TODSEC** TOD Alarm seconds value

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
DD10	56592								TALATCH
DD11	56593								TALATCH
DD12	56594								TALATCH
DD13	56595								TALATCH
DD14	56596								TALATCH
DD15	56597								TALATCH
DD16	56598								TALATCH
DD17	56599								TALATCH
DD18	56600	IMFLG	IMSP	IMALRM	IMTB				TODJIF
DD19	56601								TODSEC
DD1A	56602								TODMIN
DD1B	56603	TODAMPM							TODHOUR
DD1C	56604								ALRMJIF
DD1D	56605								ALRMSEC
DD1E	56606								ALRMMIN
DD1F	56607	ALRMAMPM							ALRM HOUR

- **ALRMAMPM** TOD Alarm AM/PM flag
- **ALRM HOUR** TOD Alarm hours value
- **ALRMJIF** TOD Alarm 10ths of seconds value

- **ALRMMIN** TOD Alarm minutes value
- **ALRMSEC** TOD Alarm seconds value
- **IMALRM** Interrupt mask for TOD alarm
- **IMFLG** Interrupt mask for FLAG line
- **IMSP** Interrupt mask for shift register (serial port)
- **IMTB** Interrupt mask for Timer B
- **TALATCH** Timer A latch value (16 bit)
- **TODAMPM** TOD AM/PM flag
- **TODHOUR** TOD hours value
- **TODJIF** TOD 10ths of seconds value
- **TODMIN** TOD Alarm minutes value
- **TODSEC** TOD Alarm seconds value

APPENDIX

4551 UART, GPIO and Utility Controller

- C65 6551 UART Registers
- 4551 General Purpose IO & Miscellaneous Interface Registers

C65 6551 UART REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
D600	54784	DATA									
D601	54785	-					FRMERR	PTYERR	RXOVRRUN		
D602	54786	TXEN	RXEN	SYNCMOD		CHARSZ		PTYEN	PTYEVE		
D603	54787	DIVISOR									
D604	54788	DIVISOR									
D605	54789	IMTXIRQ	IMRXIRQ	IMTXNMI	IMRXNMI			-			
D606	54790	IFTXIRQ	IFRXIRQ	IFTXNMI	IFRXNMI			-			

- **CHARSZ** UART character size: 00=8, 01=7, 10=6, 11=5 bits per byte
- **DATA** UART data register (read or write)
- **DIVISOR** UART baud rate divisor (16 bit). Baud rate = $7.09375\text{MHz} / \text{DIVISOR}$, unless MEGA65 fast UART mode is enabled, in which case baud rate = $80\text{MHz} / \text{DIVISOR}$
- **FRMERR** UART RX framing error flag (clear by reading \$D600)
- **IFRXIRQ** UART interrupt flag: IRQ on RX (not yet implemented on the MEGA65)
- **IFRXNMI** UART interrupt flag: NMI on RX (not yet implemented on the MEGA65)
- **IFTXIRQ** UART interrupt flag: IRQ on TX (not yet implemented on the MEGA65)
- **IFTXNMI** UART interrupt flag: NMI on TX (not yet implemented on the MEGA65)
- **IMRXIRQ** UART interrupt mask: IRQ on RX (not yet implemented on the MEGA65)
- **IMRXNMI** UART interrupt mask: NMI on RX (not yet implemented on the MEGA65)
- **IMTXIRQ** UART interrupt mask: IRQ on TX (not yet implemented on the MEGA65)
- **IMTXNMI** UART interrupt mask: NMI on TX (not yet implemented on the MEGA65)
- **PTYEN** UART Parity enable: 1=enabled

- **PTYERR** UART RX parity error flag (clear by reading \$D600)
- **PTYEVEN** UART Parity: 1=even, 0=odd
- **RXEN** UART enable receive
- **RXOVRRUN** UART RX overrun flag (clear by reading \$D600)
- **RXRDY** UART RX byte ready flag (clear by reading \$D600)
- **SYNCMOD** UART synchronisation mode flags (00=RX & TX both async, 01=RX sync, TX async, 1x=TX sync, RX async (unused on the MEGA65)
- **TXEN** UART enable transmit

4551 GENERAL PURPOSE IO & MISCELLANEOUS INTERFACE REGISTERS

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D609	54793				-				UFAST
D60B	54795	OSKZEN	OSKZON						PORTF
D60C	54796		PORTFDDR						PORTFDDR
D60D	54797	HDSCL	HDSPI	SDBSH	SDCS	SDCLK	SDDATA	RST41	CONN41
D60E	54798					BASHDDR			
D60F	54799			-				KEYUP	KEYLEFT
D610	54800				ASCIIKEY				
D611	54801	-		MSCRL	MALT	MMEGA	MCTRL	MLSHFT	MRSHFT
D612	54802	LJOYB	LJOYA	PHYJOY	PS2JOY	VRTKEY	PHYKEY	PS2KEY	WGTKEY
D615	54805	OSKEN				VIRTKEY1			
D616	54806	OSKALT				VIRTKEY2			
D617	54807	OSKTOP				VIRTKEY3			
D618	54808				KSCNRATE				
D619	54809				OSKXPOS				
D61A	54810				OSKYPOS				
D620	54816				POTAX				
D621	54817				POTAY				
D622	54818				POTBX				
D623	54819				POTBY				

- **ASCIIKEY** Last key press as ASCII (hardware accelerated keyboard scanner). Write to clear event ready for next.
- **BASHDDR** Data Direction Register (DDR) for \$D60D bit bashing port.
- **CONN41** Internal 1541 drive connect (1=connect internal 1541 drive to IEC bus)
- **HDSCL** HDMI SPI control interface SCL clock
- **HDSPI** HDMI SPI control interface SDA data line
- **KEYLEFT** Directly read C65 Cursor left key
- **KEYUP** Directly read C65 Cursor up key
- **KSCNRATE** Physical keyboard scan rate (\$00=50MHz, \$FF= 200KHz)
- **LJOYA** Rotate inputs of joystick A by 180 degrees (for left handed use)
- **LJOYB** Rotate inputs of joystick B by 180 degrees (for left handed use)
- **MALT** ALT key state (hardware accelerated keyboard scanner).
- **MCTRL** CTRL key state (hardware accelerated keyboard scanner).
- **MLSHFT** Left shift key state (hardware accelerated keyboard scanner).
- **MMEGA** MEGA/C= key state (hardware accelerated keyboard scanner).
- **MRSHFT** Right shift key state (hardware accelerated keyboard scanner).
- **MSCRL** NOSCRL key state (hardware accelerated keyboard scanner).
- **OSKALT** Display alternate on-screen keyboard layout (typically dial pad for MEGA65 telephone)
- **OSKEN** Enable display of on-screen keyboard composited overlay
- **OSKTOP** 1=Display on-screen keyboard at top, 0=Display on-screen keyboard at bottom of screen.
- **OSKXPOS** Set on-screen keyboard X position (x4 640H pixels)
- **OSKYPOS** Set on-screen keyboard Y position (x4 physical pixels)
- **OSKZEN** Display hardware zoom of region under first touch point for on-screen keyboard
- **OSKZON** Display hardware zoom of region under first touch point always
- **PHYJOY** Enable physical joystick input
- **PHYKEY** Enable physical keyboard input

- **PORTF** PMOD port A on FPGA board (data) (Nexys4 boards only)
- **PORTFDDR** PMOD port A on FPGA board (DDR)
- **POTAX** Read Port A paddle X, without having to fiddle with SID/CIA settings.
- **POTAY** Read Port A paddle Y, without having to fiddle with SID/CIA settings.
- **POTBX** Read Port B paddle X, without having to fiddle with SID/CIA settings.
- **POTBY** Read Port B paddle Y, without having to fiddle with SID/CIA settings.
- **PS2JOY** Enable PS/2 / USB keyboard simulated joystick input
- **PS2KEY** Enable ps2 keyboard/joystick input
- **RST41** Internal 1541 drive reset (1=reset, 0=operate)
- **SDBSH** Enable SD card bitbash mode
- **SDCLK** SD card SCLK
- **SDCS** SD card CS_BO
- **SDDATA** SD card MOSI/MISO
- **UFAST** C65 UART BAUD clock source: 1 = 7.09375MHz, 0 = 80MHz (VIC-IV pixel clock)
- **VIRTKEY1** Set to \$7F for no key down, else specify virtual key press.
- **VIRTKEY2** Set to \$7F for no key down, else specify 2nd virtual key press.
- **VIRTKEY3** Set to \$7F for no key down, else specify 3nd virtual key press.
- **VRTKEY** Enable virtual/synthetic keyboard input
- **WGTKEY** Enable widget board keyboard/joystick input

J APPENDIX

45E100 Fast Ethernet Controller

- Overview
- Memory Mapped Registers
- Example Programs

OVERVIEW

The 45E100 is a new and simple Fast Ethernet controller that has been designed specially for the MEGA65 and for 8-bit computers generally. In addition to supporting 100Mbit Fast Ethernet, it is radically different from other ethernet controllers, such as the RR-NET.

The 45E100 includes dual receive buffers, allowing one frame to be processed while another is receiving. It includes automatic CRC32 checking on reception, and automatic CRC32 generation for transmit, considerably reducing the burden on the processor and allowing for very simple programs. It also supports true full-duplex operation at 100Mbit per second, allowing for total bi-directional throughput exceeding 100Mbit per second. The MAC address is software configurable, and promiscuous mode is supported, as are individual control of the reception of broadcast and multi-cast ethernet frames. The 45E100 also supports both transmit and receive interrupts, allowing greatly improved real-world performance. When especially low latency is required, it is also possible to immediately abort the transmission of the current ethernet frame, so that a higher-priority frame can be immediately sent. These features combine to enable sub-millisecond round trip latencies, which can be of particular value for interactive applications, such as multi-player network games.

Differences to the RR-NET and similar solutions

The RR-NET and other ethernet controllers for the Commodore™ line of 8-bit home computers generally use an ethernet controller that was designed for 16-bit PCs, but that also supports a so-called “8-bit mode,” which suffers from a number of disadvantages. The primary disadvantages are the lack of working interrupts, and processor intensive access to the ethernet frame buffers. The lack of interrupts forces programs to use polling to check for the arrival of new ethernet frames. This, together with the complexities of accessing the buffers results in an ethernet interface that is very slow, and whose real-world throughput is considerably less than its theoretical 10Mbits per second. Even a Commodore 64 with REU cannot achieve speeds above several tens of kilobytes per second.

In contrast, the 45E100 supports both RX (ethernet frame received) interrupts and TX (ready to transmit) interrupts, freeing the processor from having to poll

the device. Because the 45E100 supports RX interrupts, there is no need for large numbers of receive buffers, which is why the 45E100 requires only two RX buffers to achieve very high levels of performance.

Further, the 45E100 supports direct memory mapping of the ethernet frame buffers, allowing for much more efficient access, including by DMA. Using the MEGA65's integrated DMA controller it is quite possible to achieve transfer rates of several mega-bytes per second – some 100x faster than the RR-NET.

Theory of Operation: Receiving Frames

The 45E100 is simple to operate: To begin receiving ethernet frames, the programmer needs only to clear the RST bit (bit 0 of register \$D6E0) to release the ethernet controller from reset. It will then auto-negotiate connection at the highest available speed, typically 100Mbit, full-duplex. The RXBLKD bit (bit 6 of \$D6E0) should then be checked, and if set, the RXBM (bit 2 of register \$D6E1) bit should be toggled to switch the active and mapped receive buffers, so that the 45E100 knows that the program no longer needs the contents of the previously mapped buffer, and can safely begin receiving an ethernet frame into that buffer.

When the 45E100 receives an ethernet frame, it will assert RXBLKD to indicate that the receive buffer has been filled with an ethernet frame. No further ethernet frames will be received until RXBLKD is cleared again, as described above. This is because the 45E100 has only two receive buffers for ethernet frames: one of which is mapped visible to the processor, and the other which is visible to the 45E100's ethernet engine at any point in time. Toggling RXBM allows toggling between which of the two buffers is mapped and which is ready to receive an ethernet frame. The buffers are 2KiB bytes each. The first two bytes are used to indicate the length of the received frame, and four are consumed by the ethernet CRC32 code, yielding an effective Maximum Transport Unit (MTU) length of 2,042 bytes. The ethernet frame data begins at byte offset 2 in the receive buffer, with the frame length written LSB-first in the first two bytes. The layout of the receive buffers is thus as follows:

HEX	DEC	Length	Description
0000	0	1	The low byte of the length of the received ethernet frame.

continued ...

...continued

HEX	DEC	Length	Description
0001	1	1	The lower four bits contain the upper bits of the length of the received ethernet frame. Bit 4 is set if the received ethernet frame is a multi-cast frame. Bit 5 if it is a broadcast frame. Bit 6 is set if the frame's destination address matches the 45E100's programmed MAC address. Bit 7 is set if the CRC32 check for the received frame failed, i.e., that the frame is either truncated or was corrupted in transit.
0002 - 07FB	2 - 2,043	2,042	The received frame. Frames shorter than 2,042 bytes will begin at offset 2.
07FC - 07FF	2,044 - 2,047	4	Reserved space for holding the CRC32 code during reception. The CRC32 code is, however, always located directly after the received frame, and thus will only occupy this space if the received frame is more than 2,038 bytes long.

Because of the very rapid rate at which Fast Ethernet frames can be received, a programmer should use the receive interrupt feature, enabled by setting RXQEN (bit 7 of \$D6E1). Polling is possible as an alternative, but is not recommended with the 45E100, because at the 100Mbit Fast Ethernet speed, packets can arrive as often as every 10 microseconds. Fortunately, at the MEGA65's 40MHz full speed mode, and using the 20MiB per second DMA copy functionality, it is possible to keep up with such high data rates.

Accessing the Ethernet Frame Buffers

Unlike on the RR-NET, the 45E100's ethernet frame buffers are able to be memory mapped, allowing rapid access via DMA or through assembly language programs. It is also possible to access the buffers from BASIC with some care.

The frame buffers can either be accessed from their natural location in the MEGA65's extended address space at address \$FFDE800 - \$FFDEFFF, or they can be mapped into the normal C64/C65 \$D000 IO address space.

Care must be taken as mapping the ethernet frame buffers into the \$D000 IO address space causes all other IO devices to unavailable during this time. Therefore interrupts MUST be disabled before doing so, whether using BASIC or machine code. Therefore when programming in assembly language or machine code, it is recommended to use the natural location, and to access this memory area using one of the three mechanisms for accessing extended address space, which are described in Appendix ??.

The method of disabling interrupts differs depending on the context in which a program is being written. For programs being written using C64 mode's BASIC 2, the following will work:

```
POKE56333,127: REM DISABLE CIA TIMER IRQS
```

While for C65's BASIC 10, the following must instead be used, because a VIC-III raster interrupt is used instead of a CIA-based timer interrupt:

```
POKE53274,0: REM DISABLE VIC-II/III/IV RASTER IRQS
```

Once this has been done, the IO context for the ethernet controller can be activated by writing \$45 (69 in decimal, equal to the character 'E' in PETSCII) and \$54 (84 in decimal, equal to the character 'T' in PETSCII) into the VIC-IV's KEY register (\$D02F, 53295 in decimal), for example:

```
POKE53295,ASC("E"):POKE53295,ASC("T")
```

At this point, the ethernet RX buffer can be read beginning at location \$D000 (53248 in decimal), and the TX buffer can be written to at the same address. Refer to 'Theory of Operation: Receiving Frames' above for further explanation on this.

Once you have finished accessing the ethernet frame buffer, you can restore the normal C64, C65 or MEGA65 IO context by writing to the VIC-III/IV's KEY register. In most cases, it will make the most sense to revert to the MEGA65's IO context by writing \$47 (71 decimal) in and \$53 (83 in decimal) to the KEY register, for example:

```
POKE53295,ASC("G"):POKE53295,ASC("S")
```

Finally, you should then re-enable interrupts, which will again depend on whether you are programming from C64 or C65 mode. For C64 mode:

```
POKE56333,129
```

For C65 mode it would be:

```
POKE53274,129
```

Theory of Operation: Sending Frames

Sending frames is similarly simple: The program must simply load the frame to be transmitted into the transmit buffer, write its length into TXSZLSB and TXSZMSB registers, and then write \$01 into the COMMAND register. The frame will then begin to transmit, as soon as the transmitter is idle. There is no need to calculate and attach an ethernet CRC32 field, as the 45E100 does this automatically.

Unlike for the receiver, there is only one frame buffer for the transmitter (this may be changed in a future revision). This means that you cannot prepare the next frame until the previous frame has already been sent. This slightly reduces the maximum data throughput, in return for a very simple architecture.

Also, note that the transmit buffer is write-only from the processor bus interface. This means that you cannot directly read the contents of the transmit buffer, but must load values "blind". Finally, the 45E100 allows you to send ethernet.

Advanced Features

In addition to operating as a simple and efficient ethernet frame transceiver, the 45E100 includes a number of advanced features, described here.

Broadcast and Multicast Traffic and Promiscuous Mode

The 45E100 supports filtering based on the destination Ethernet address, i.e., MAC address. By default, only frames where the destination Ethernet address matches the ethernet address programmed into the MACADDR1 - MACADDR6 registers will be received. However, if the MCST bit is set, then multicast

ethernet frames will also be received. Similarly, setting the BCST bit will allow all broadcast frames, i.e., with MAC address ff:ff:ff:ff:ff:ff, to be received. Finally, if the NOPROM bit is cleared, the 45E100 disables the filter entirely, and will receive all valid ethernet frames.

Debugging and Diagnosis Features

The 45E100 also supports several features to assist in the diagnosis of ethernet problems. First, if the NOCRC bit is set, then even ethernet frames that have invalid CRC32 values will be received. This can help debug faulty ethernet devices on a network.

If the STRM bit is set, the ethernet transmitter transmits a continuous stream of debugging frames supplied via a special high-bandwidth logging interface. By default, the 45E100 emits a stream of approximately 2,200 byte ethernet frames that contain compressed video provided by a VIC-IV or compatible video controller that supports the MEGA65 video-over-ethernet interface. By writing a custom decoder for this stream of ethernet frames, it is possible to create a remote display of the MEGA65 via ethernet. Such a remote display can be used, for example, to facilitate digital capture of the display of a MEGA65.

The size and content of the debugging frames can be controlled by writing special values to the COMMAND register. Writing \$F1 allows the selection of frames that are 1,200 bytes long. While this reduces the performance of the debugging and streaming features, it allows the reception of these frames on systems whose ethernet controllers cannot be configured to receive frames of 2,200 bytes.

If the STRM bit is set and bit 2 of \$D6E1 is also set, a compressed log of instructions executed by the 45gs02 CPU will instead be streamed, if a compatible processor is connected to this interface. This mechanism includes back-pressure, and will cause the 45gs02 processor to slowdown, so that the instruction data can be emitted. This typically limits the speed of the connected 45gs02 processor to around 5MHz, depending on the particular instruction mix.

Note also that the status of bit 2 of \$D6E1 cannot currently be read directly. This may be corrected in a future revision.

Finally, if the video streaming functionality is enabled, this also enables reception of synthetic keyboard events via ethernet. These are delivered to the MEGA65's Keyboard Complex Interface Adapter (KCIA), allowing full remote interaction with a MEGA65 via its ethernet interface. This feature is primarily intended for development.

MEMORY MAPPED REGISTERS

The 45E100 Fast Ethernet controller is a MEGA65-specific feature. It is therefore only available in the MEGA65 IO context. This is enabled by writing \$53 and then \$47 to VIC-IV register \$D02F. If programming in BASIC, this can be done with:

```
POKE53295,ASC("G"):POKE53295,ASC("S")
```

The 45E100 Fast Ethernet controller has the following registers

HEX	DEC	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
D6E0	55008	-	RXBLKD	-	KEYEN	DRXDV	DRXD		RST
D6E1	55009	RXQEN	TXQEN	RXQ	TXQ	STRM	RXBU	RXBM	-
D6E2	55010					TXSZLSB			
D6E3	55011					TXSZMSB			
D6E4	55012					COMMAND			
D6E5	55013	-		MCST	BCST	TXPH	NOCRC	NOPROM	
D6E6	55014		MIIMPHY				MIIMREG		
D6E7	55015					MIIMVLSB			
D6E8	55016					MIIMVMSB			
D6E9	55017					MACADDR1			
D6EA	55018					MACADDR2			
D6EB	55019					MACADDR3			
D6EC	55020					MACADDR4			
D6ED	55021					MACADDR5			
D6EE	55022					MACADDR6			

- **BCST** Accept broadcast frames
- **COMMAND** Ethernet command register (write only)
- **DRXD** Read ethernet RX bits currently on the wire
- **DRXDV** Read ethernet RX data valid (debug)
- **KEYEN** Allow remote keyboard input via magic ethernet frames
- **MACADDR1** Ethernet MAC address
- **MACADDR2** Ethernet MAC address
- **MACADDR3** Ethernet MAC address

- **MACADDR4** Ethernet MAC address
- **MACADDR5** Ethernet MAC address
- **MACADDR6** Ethernet MAC address
- **MCST** Accept multicast frames
- **MIIMPHY** Ethernet MIIM PHY number (use 0 for Nexys4, 1 for MEGA65
r1 PCBs)
- **MIIMREG** Ethernet MIIM register number
- **MIIMVLSB** Ethernet MIIM register value (LSB)
- **MIIMVMSB** Ethernet MIIM register value (MSB)
- **NOCRC** Disable CRC check for received packets
- **NOPROM** Ethernet disable promiscuous mode
- **RST** Write 0 to hold ethernet controller under reset
- **RXBLKD** Indicate if ethernet RX is blocked until RX buffers rotated
- **RXBM** Set which RX buffer is memory mapped
- **RXBU** Indicate which RX buffer was most recently used
- **RXQ** Ethernet RX IRQ status
- **RXQEN** Enable ethernet RX IRQ
- **STRM** Enable streaming of CPU instruction stream or VIC-IV display on
ethernet
- **TXPH** Ethernet TX clock phase adjust
- **TXQ** Ethernet TX IRQ status
- **TXQEN** Enable ethernet TX IRQ
- **TXSZLSB** TX Packet size (low byte)
- **TXSZMSB** TX Packet size (high byte)

COMMAND register values

The following values can be written to the COMMAND register to perform the described functions. In normal operation only the STARTTX command is required, for example, by performing the following POKE:

```
POKE55812,1
```

HEX	DEC	Signal	Description
0000	0	STOPTX	Immediately stop transmitting the current ethernet frame. Will cause a partially sent frame to be received, most likely resulting in the loss of that frame.
0001	1	STARTTX	Transmit packet
00D0	208	RXNORMAL	Disable the effects of RXONLYONE
00D4	212	DEBUGVIC	Select VIC-IV debug stream via ethernet when \$D6E1.3 is set
00DC	220	DEBUGCPU	Select CPU debug stream via ethernet when \$D6E1.3 is set
00DE	222	RXONLYONE	Receive exactly one ethernet frame only, and keep all signals states (for debugging ethernet sub-system)
00F1	241	FRAME1K	Select 1KiB frames for video/cpu debug stream frames (for receivers that do not support MTUs of greater than 2KiB)
00F2	242	FRAME2K	Select 2KiB frames for video/cpu debug stream frames, for optimal performance.

EXAMPLE PROGRAMS

Example programs for the ethernet controller exist in imperfect for in the MEGA65 Core repository on github in the src/tests and src/examples directories.

K

APPENDIX

Reference Tables

- Units of Storage
- Base Conversion

UNITS OF STORAGE

Unit	Equals	Abbreviation
1 Bit		
1 Nibble	4 Bits	
1 Byte	8 bits	B
1 Kilobyte	1024 B	KB
1 Megabyte	1024 KB or 1,048,576 B	MB

BASE CONVERSION

Decimal	Binary	Hexadecimal
0	%0	\$0
1	%1	\$1
2	%10	\$2
3	%11	\$3
4	%100	\$4
5	%101	\$5
6	%110	\$6
7	%111	\$7
8	%1000	\$8
9	%1001	\$9
10	%1010	\$A
11	%1011	\$B
12	%1100	\$C
13	%1101	\$D
14	%1110	\$E
15	%1111	\$F
16	%10000	\$10
17	%10001	\$11
18	%10010	\$12
19	%10011	\$13
20	%10100	\$14
21	%10101	\$15
22	%10110	\$16
23	%10111	\$17
24	%11000	\$18
25	%11001	\$19
26	%11010	\$1A
27	%11011	\$1B
28	%11100	\$1C
29	%11101	\$1D
30	%11110	\$1E
31	%11111	\$1F

Decimal	Binary	Hexadecimal
32	%100000	\$20
33	%100001	\$21
34	%100010	\$22
35	%100011	\$23
36	%100100	\$24
37	%100101	\$25
38	%100110	\$26
39	%100111	\$27
40	%101000	\$28
41	%101001	\$29
42	%101010	\$2A
43	%101011	\$2B
44	%101100	\$2C
45	%101101	\$2D
46	%101110	\$2E
47	%101111	\$2F
48	%110000	\$30
49	%110001	\$31
50	%110010	\$32
51	%110011	\$33
52	%110100	\$34
53	%110101	\$35
54	%110110	\$36
55	%110111	\$37
56	%111000	\$38
57	%111001	\$39
58	%111010	\$3A
59	%111011	\$3B
60	%111100	\$3C
61	%111101	\$3D
62	%111110	\$3E
63	%111111	\$3F

Decimal	Binary	Hexadecimal
64	%1000000	\$40
65	%1000001	\$41
66	%1000010	\$42
67	%1000011	\$43
68	%1000100	\$44
69	%1000101	\$45
70	%1000110	\$46
71	%1000111	\$47
72	%1001000	\$48
73	%1001001	\$49
74	%1001010	\$4A
75	%1001011	\$4B
76	%1001100	\$4C
77	%1001101	\$4D
78	%1001110	\$4E
79	%1001111	\$4F
80	%1010000	\$50
81	%1010001	\$51
82	%1010010	\$52
83	%1010011	\$53
84	%1010100	\$54
85	%1010101	\$55
86	%1010110	\$56
87	%1010111	\$57
88	%1011000	\$58
89	%1011001	\$59
90	%1011010	\$5A
91	%1011011	\$5B
92	%1011100	\$5C
93	%1011101	\$5D
94	%1011110	\$5E
95	%1011111	\$5F

Decimal	Binary	Hexadecimal
96	%1100000	\$60
97	%1100001	\$61
98	%1100010	\$62
99	%1100011	\$63
100	%1100100	\$64
101	%1100101	\$65
102	%1100110	\$66
103	%1100111	\$67
104	%1101000	\$68
105	%1101001	\$69
106	%1101010	\$6A
107	%1101011	\$6B
108	%1101100	\$6C
109	%1101101	\$6D
110	%1101110	\$6E
111	%1101111	\$6F
112	%1110000	\$70
113	%1110001	\$71
114	%1110010	\$72
115	%1110011	\$73
116	%1110100	\$74
117	%1110101	\$75
118	%1110110	\$76
119	%1110111	\$77
120	%1111000	\$78
121	%1111001	\$79
122	%1111010	\$7A
123	%1111011	\$7B
124	%1111100	\$7C
125	%1111101	\$7D
126	%1111110	\$7E
127	%1111111	\$7F

Decimal	Binary	Hexadecimal
128	%10000000	\$80
129	%10000001	\$81
130	%10000010	\$82
131	%10000011	\$83
132	%10000100	\$84
133	%10000101	\$85
134	%10000110	\$86
135	%10000111	\$87
136	%10001000	\$88
137	%10001001	\$89
138	%10001010	\$8A
139	%10001011	\$8B
140	%10001100	\$8C
141	%10001101	\$8D
142	%10001110	\$8E
143	%10001111	\$8F
144	%10010000	\$90
145	%10010001	\$91
146	%10010010	\$92
147	%10010011	\$93
148	%10010100	\$94
149	%10010101	\$95
150	%10010110	\$96
151	%10010111	\$97
152	%10011000	\$98
153	%10011001	\$99
154	%10011010	\$9A
155	%10011011	\$9B
156	%10011100	\$9C
157	%10011101	\$9D
158	%10011110	\$9E
159	%10011111	\$9F

Decimal	Binary	Hexadecimal
160	%10100000	\$A0
161	%10100001	\$A1
162	%10100010	\$A2
163	%10100011	\$A3
164	%10100100	\$A4
165	%10100101	\$A5
166	%10100110	\$A6
167	%10100111	\$A7
168	%10101000	\$A8
169	%10101001	\$A9
170	%10101010	\$AA
171	%10101011	\$AB
172	%10101100	\$AC
173	%10101101	\$AD
174	%10101110	\$AE
175	%10101111	\$AF
176	%10110000	\$B0
177	%10110001	\$B1
178	%10110010	\$B2
179	%10110011	\$B3
180	%10110100	\$B4
181	%10110101	\$B5
182	%10110110	\$B6
183	%10110111	\$B7
184	%10111000	\$B8
185	%10111001	\$B9
186	%10111010	\$BA
187	%10111011	\$BB
188	%10111100	\$BC
189	%10111101	\$BD
190	%10111110	\$BE
191	%10111111	\$BF

Decimal	Binary	Hexadecimal
192	%11000000	\$C0
193	%11000001	\$C1
194	%11000010	\$C2
195	%11000011	\$C3
196	%11000100	\$C4
197	%11000101	\$C5
198	%11000110	\$C6
199	%11000111	\$C7
200	%11001000	\$C8
201	%11001001	\$C9
202	%11001010	\$CA
203	%11001011	\$CB
204	%11001100	\$CC
205	%11001101	\$CD
206	%11001110	\$CE
207	%11001111	\$CF
208	%11010000	\$D0
209	%11010001	\$D1
210	%11010010	\$D2
211	%11010011	\$D3
212	%11010100	\$D4
213	%11010101	\$D5
214	%11010110	\$D6
215	%11010111	\$D7
216	%11011000	\$D8
217	%11011001	\$D9
218	%11011010	\$DA
219	%11011011	\$DB
220	%11011100	\$DC
221	%11011101	\$DD
222	%11011110	\$DE
223	%11011111	\$DF

Decimal	Binary	Hexadecimal
224	%11100000	\$E0
225	%11100001	\$E1
226	%11100010	\$E2
227	%11100011	\$E3
228	%11100100	\$E4
229	%11100101	\$E5
230	%11100110	\$E6
231	%11100111	\$E7
232	%11101000	\$E8
233	%11101001	\$E9
234	%11101010	\$EA
235	%11101011	\$EB
236	%11101100	\$EC
237	%11101101	\$ED
238	%11101110	\$EE
239	%11101111	\$EF
240	%11110000	\$F0
241	%11110001	\$F1
242	%11110010	\$F2
243	%11110011	\$F3
244	%11110100	\$F4
245	%11110101	\$F5
246	%11110110	\$F6
247	%11110111	\$F7
248	%11111000	\$F8
249	%11111001	\$F9
250	%11111010	\$FA
251	%11111011	\$FB
252	%11111100	\$FC
253	%11111101	\$FD
254	%11111110	\$FE
255	%11111111	\$FF



APPENDIX

Flashing the FPGAs and CPLDs in the MEGA65

- **Warning**
- **Flashing the Artix 100T main
FPGA with XILINX VIVADO**
- **Flashing the CPLD in the
MEGA65's Keyboard with
LATTICE DIAMOND**

**• Flashing the MAX10 FPGA on
the MEGA65's Mainboard
with INTEL QUARTUS**

The MEGA65 is an open-source and open-hardware computer. This means you are free, not only to write programs that run on the MEGA65 as a finished computer, but you can use the re-programmable chips in the MEGA65 to turn it into all sorts of other things!

If you just want to install an upgrade core for the MEGA65, or a core that lets you use your MEGA65 as another type of computer, you are probably looking for Chapter 4 instead. This chapter is more intended for people who want to help develop cores for the MEGA65.

These re-programmable chips are called Field Programmable Gate Arrays (FPGAs) or Complex Programmable Logic Devices (CPLDs), and can implement a wide variety of circuits. They are normally programmed using a programming language like VHDL or Verilog. These are languages that are not commonly encountered by most people. They are also quite different in some ways to “normal” programming languages, and it can take a while to understand how they work, but with some effort and perseverance, many people will be able to do exiting things with them.

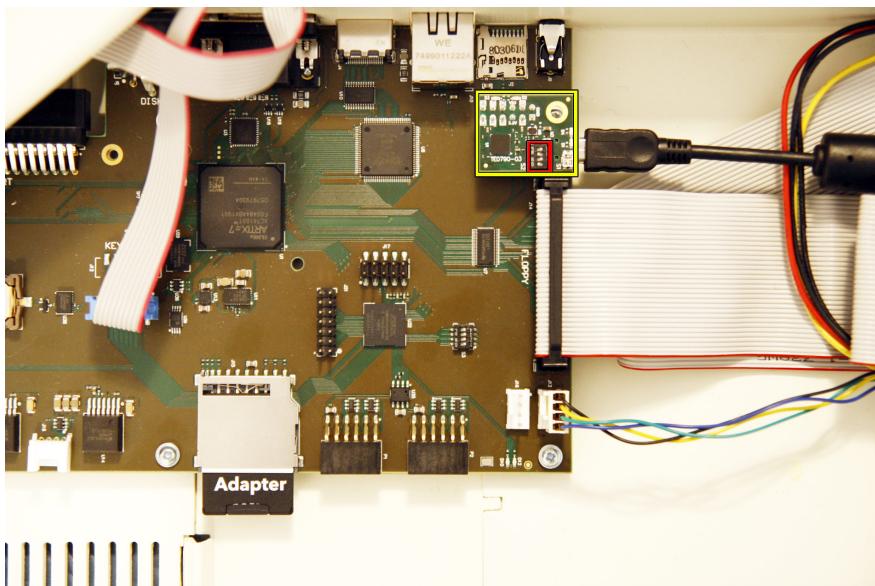
Be prepared to install many gigabytes of software on a Linux or Windows PC, before you will be able to write programs for the FPGAs and CPLDs in the MEGA65. Also, “compiling” complex designs can take up to several hours, depending on the speed and memory capacity of your computer! We recommend a computer with at least 8GB RAM, and preferably 16GB if you want to write programs for FPGAs and CPLDs. If on the other hand all you want to do is load programs onto your MEGA65’s FPGAs and CPLDs that other people have written, then most computers running a recent version of Windows or Linux should be able to cope.

WARNING

Before we go any further, we do have to provide a warning about reprogramming the FPGAs and CPLDs in the MEGA65. Re-programming the MEGA65 FPGA can potentially cause damage, or leave your MEGA65 in an unresponsive state from which it is very difficult to recover, i.e., “bricked”. Therefore if you choose to open your MEGA65 and reprogram any of the FPGAs it contains, it is no longer possible to guarantee its correct operation. Therefore in this case we can not reasonably honour the warranty of the device as a computer. You have been warned!

FLASHING THE ARTIX 100T MAIN FPGA WITH XILINX VIVADO

If you choose to proceed, you will need a TE0790-03 JTAG programming module, a functioning installation of Xilinx's Vivado software. This can be done on either Windows or Linux, but in both cases you will need to install any necessary USB drivers. It is also necessary to have dip-switches 1 and 3 in the ON position and dip-switches 2 and 4 in the OFF position on the TE-0790. With your MEGA65 disconnected from the power, the TE-0790 must be installed on the JB1 connector, which is located between the floppy data cable and the audio jack. The gold-plated hole of the TE-0790 must line up with the screw hole below. The mini-USB cable will then connect on the side towards the 3.5" floppy drive. The following image shows the correct position: The TE0790 is surrounded by the yellow box, and the dip-switches by the red box. Dip-switch 1 is the one nearest the floppy data cable.



Connect your non-8-bit computer to the FPGA programming device using a mini-USB cable. Switch the MEGA65 computer ON. Open VIVADO, which can be downloaded from the internets.

To access the Hardware Manager, open a project in VIVADO or create an empty one, if you do not have any projects yet.

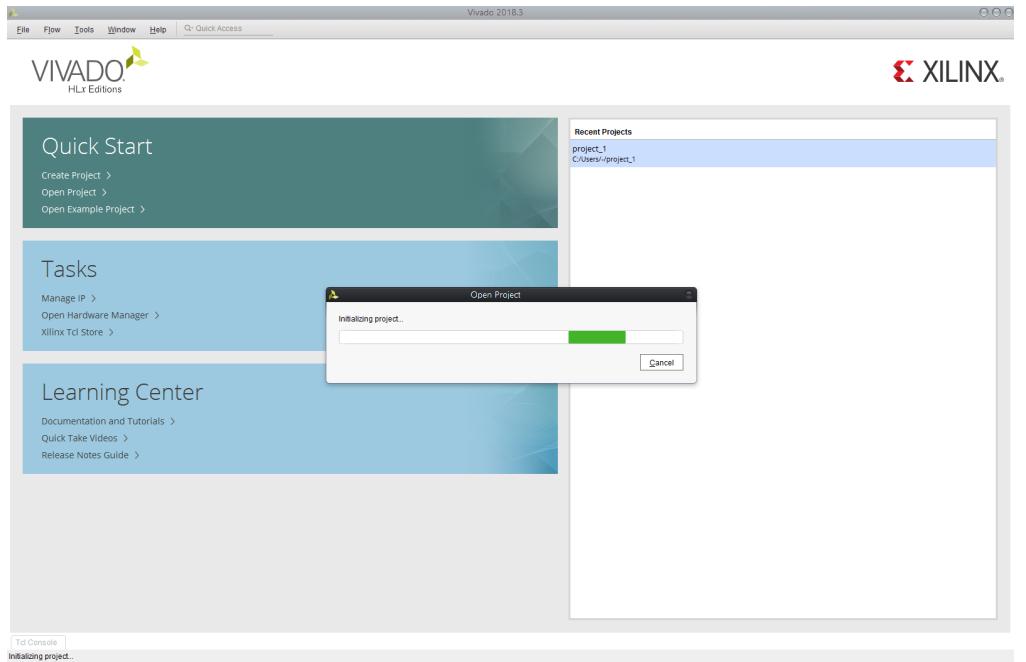


Figure L.1: Step 1: Open a project in VIVADO

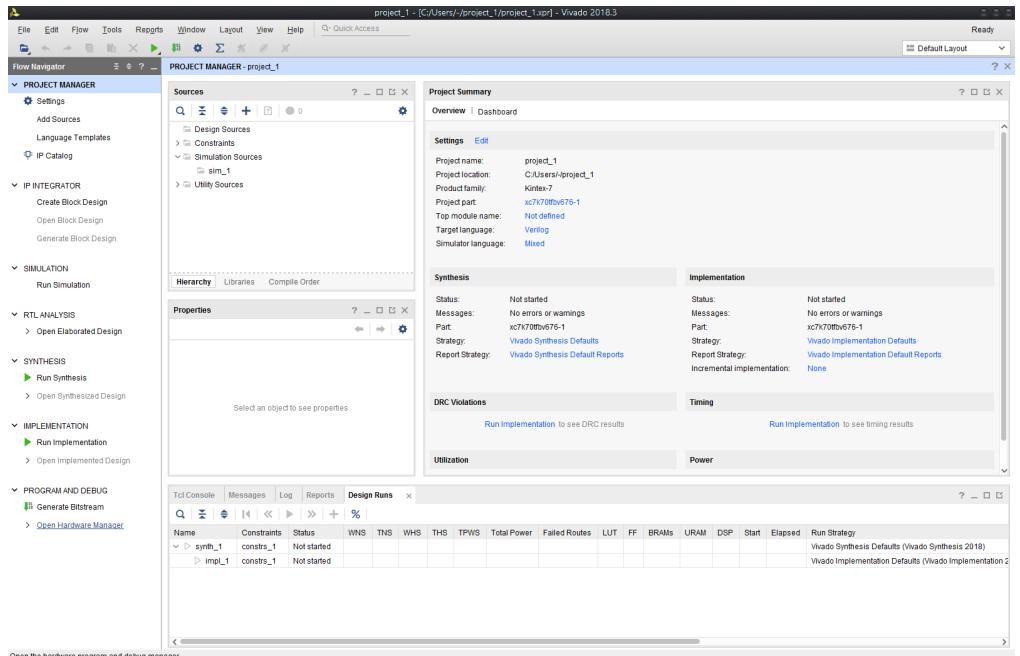


Figure L.2: Step 2: Open Hardware Manager

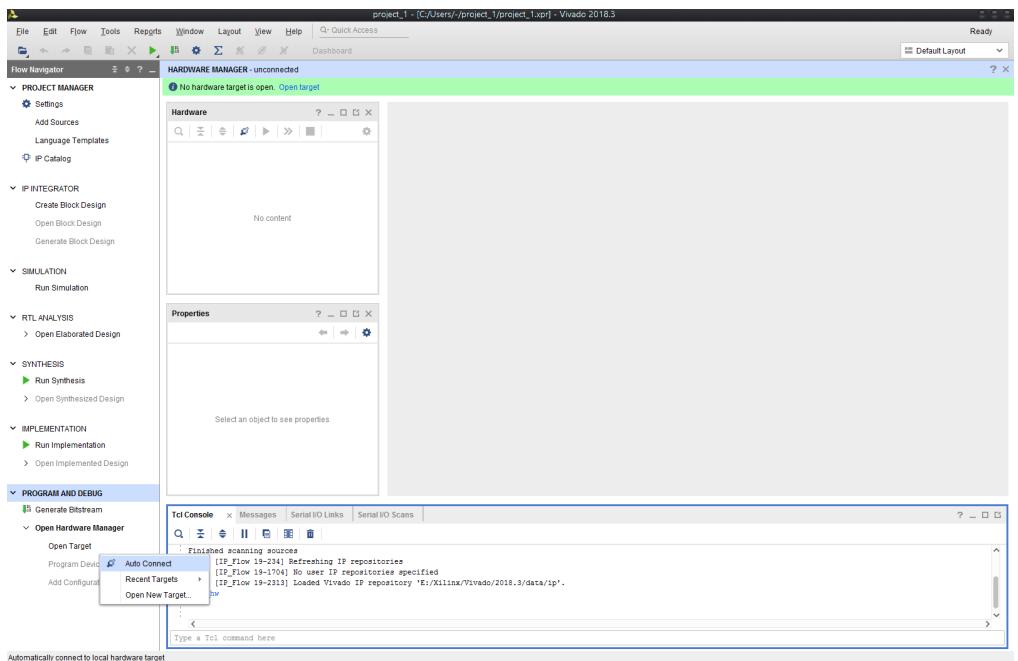


Figure L.3: Step 3: Connect to FPGA

In the left column, select "Open Hardware Manager" at the very bottom.

Under "Hardware Manager", choose "Open Target", then "Auto Connect".

Wait a moment, "Connecting to server..." should automatically close without dropping an error to the console.

Under "Hardware Manager", choose "Add Configuration Memory Device", then "xc7a100t_0".

In the newly opened dialogue, type "S25fl256s" (without quotes), then select "s25fl256xxxxxxxx0-spi-x1_x2_x4" (the upper one) and click "OK".

In the next dialogue, choose your local Configuration file, namely a bitstream with file suffix ".mcs". Leave all other parameters as they are (see L.7).

Patiently wait for the programming to finish. This can take several minutes as the Vivado software erases and then reprograms the flash memory that is

If your screen looks like L.9, your new bistream has been successfully flashed into the Atmel 100T FPGA!

into the Arria 100T FPGA!

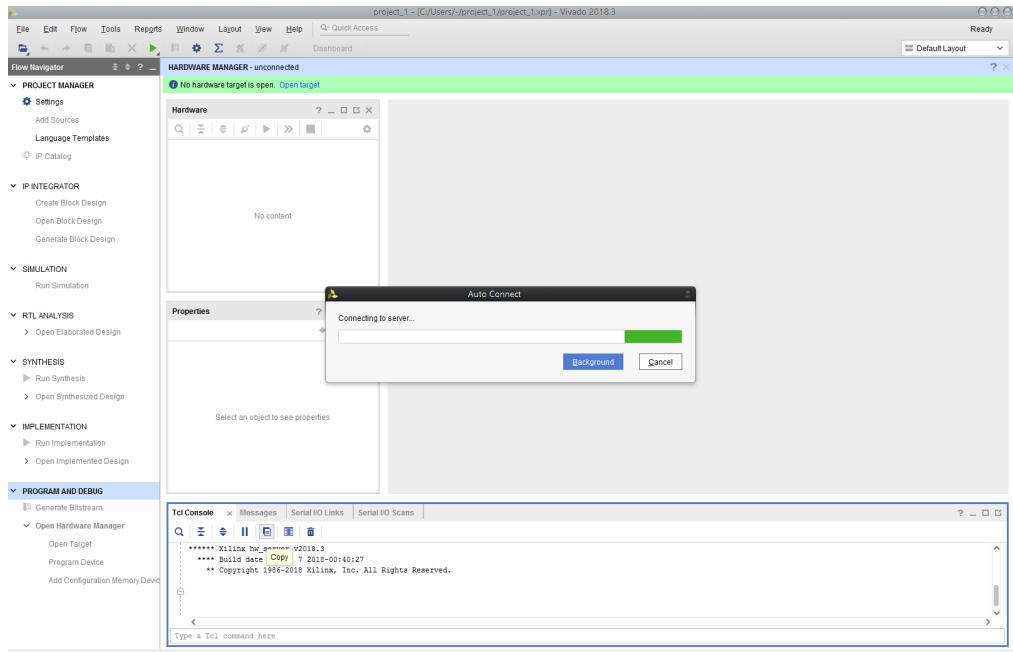


Figure L.4: Step 4: Wait a moment

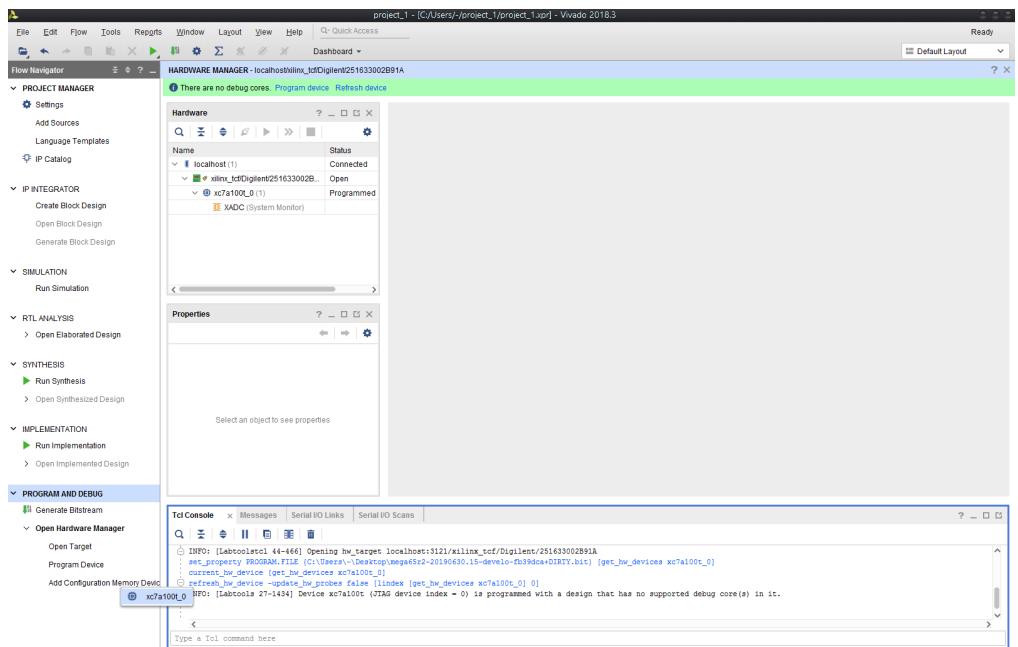


Figure L.5: Step 5: Add Configuration Memory Device

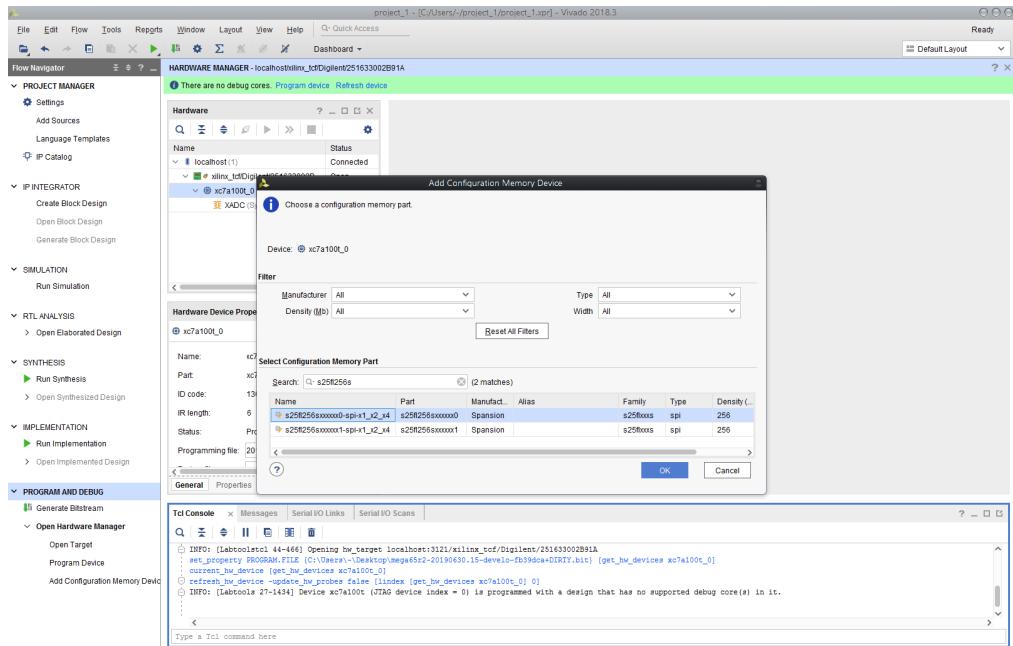


Figure L.6: Step 6: Select Memory Part

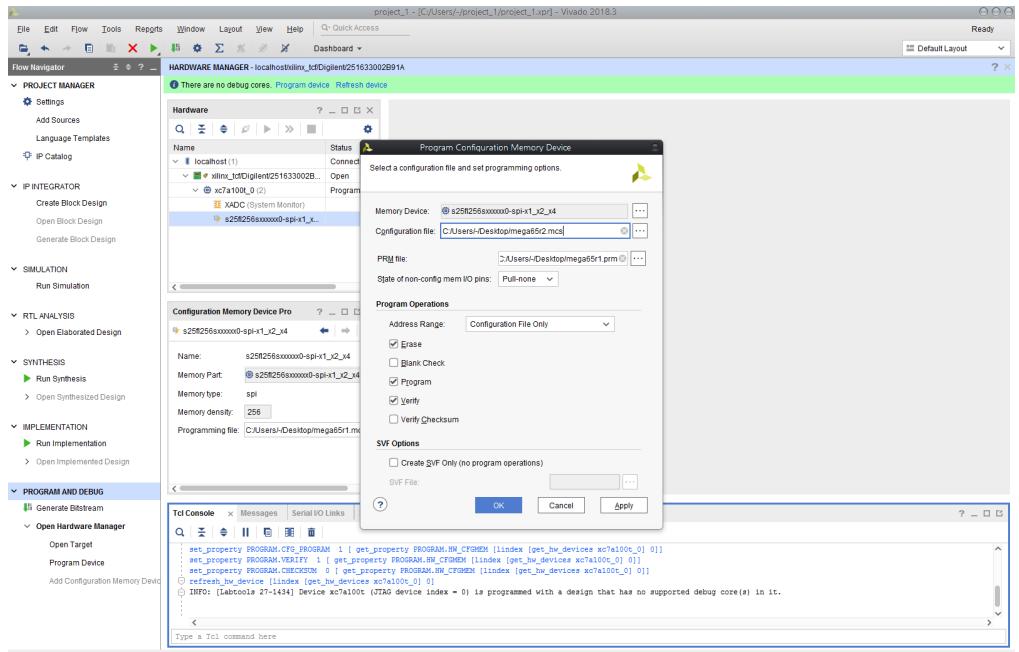


Figure L.7: Step 7: Set programming options

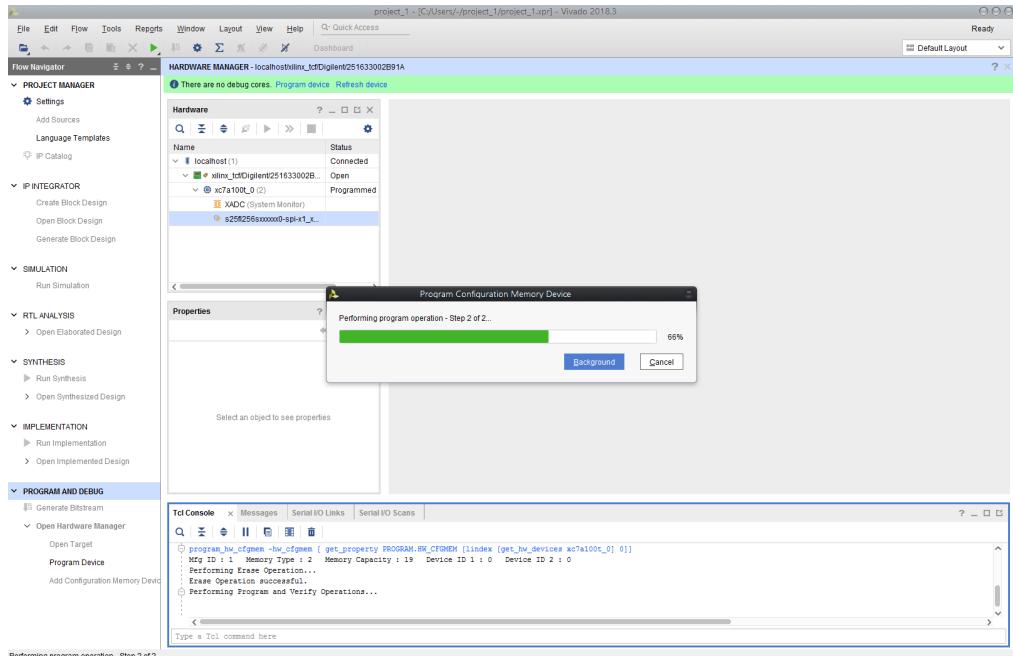


Figure L.8: Step 8: Programming in progress

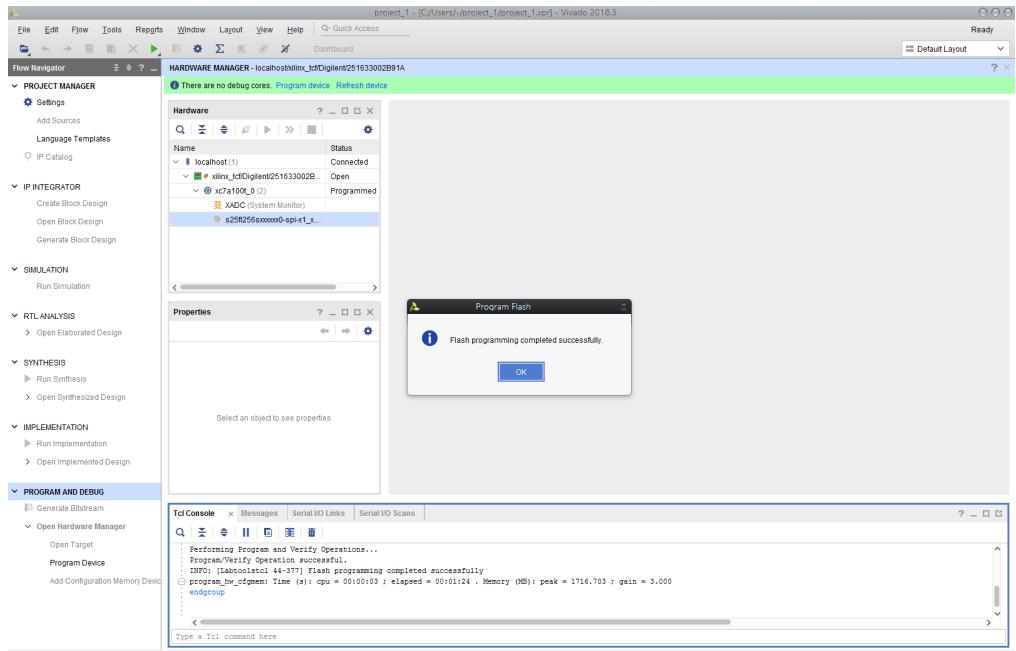


Figure L.9: Step 9: Programming successful

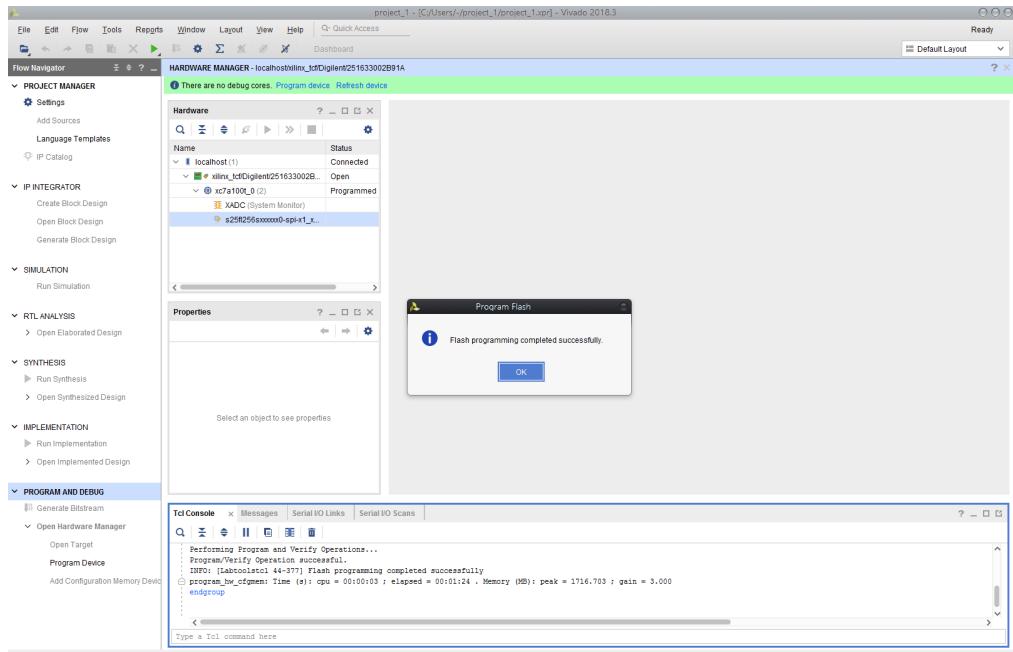
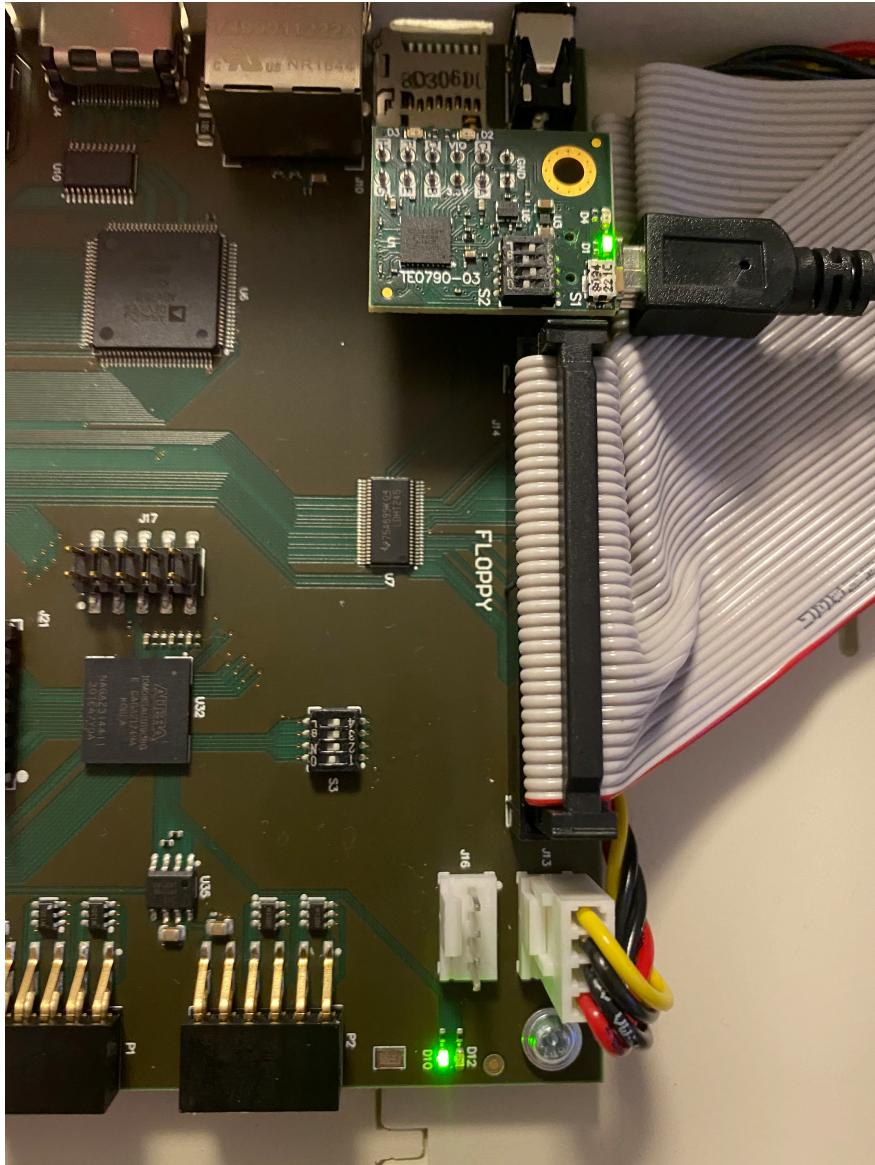


Figure L.10: Step 10: Reflashing the FPGA

If you want to repeat the process, you might find the "Add Configuration Memory Device" option in step 5 greyed out. Instead, select "s25fl256xxxxxxxxx0-spi-x1_x2_x4" in the "Hardware" window, press right mouse button and select "Program Configuration Memory Device" to flash.

FLASHING THE CPLD IN THE MEGA65'S KEYBOARD WITH LATTICE DIAMOND

If you choose to proceed, you will need a TE0790-03 JTAG programming module, a functioning installation of Lattice Diamond Programmer software. This can be done on either Windows or Linux, but in both cases you will need to install any necessary USB drivers. It is also necessary to have dip-switches 1 and 3 in the ON position and dip-switches 2 and 4 in the OFF position on the TE-0790. With your MEGA65 disconnected from the power, the TE-0790 must be installed on the JB1 connector, which is located between the floppy data cable and the audio jack. The gold-plated hole of the TE-0790 must line up with the screw hole below. The mini-USB cable will then connect on the side towards the 3.5" floppy drive. The following image shows the correct position: The TE0790 is surrounded by the yellow box, and the dip-switches by the red box. Dip-switch 1 is the one nearest the floppy data cable.



One the PCB r2 MEGA65 Mainbord dip switch 1 (the one nearest to the user sitting in front of the machine must be in the ON position, the other switches must be OFF. The keyboard will go into "Police Mode" (blue and red blinking LEDs) when set correctly.



Figure L.1 1: Step 1: Open DIAMOND PROGRAMMER

Connect your non-8-bit computer to the FPGA programming device using a mini-USB cable. Switch the MEGA65 computer ON. Open DIAMOND PROGRAMMER, which can be downloaded from the internets.

Select "Create a new project from a JTAG scan". If entry under "Cable:" is empty, click "Detect Cable".

If dialog "Programmer: Multiple Cables Detected" appears, select the first entry ("Location 0000") and click "OK".

You have now created a new project which should display "MachXO2" under "Device Family" and "LCMXO2-1200HC" under "Device"

Choose "File" then "Open File" to load the DIAMOND PROGRAMMER project with the MEGA65 keyboard firmware update.

Navigate into the folder with the extracted MEGA65 keyboard firmware files you have received and select the file ending with ".xcf".

Click the three dots under "File Name" to set the correct path and find the file ending with ".jed".

Select the file ending with ".jed" and click "OK".

Click on the icon with the green arrow facing down "PROGRAM", which looks similar to the DIAMOND PROGRAMMER program icon.

After a moment the Output window should display "INFO - Operation: successful." and the "Status" cell should go green (does not always happen).

You have now successfully flashed the MEGA65 keyboard. If you wish you can save the project now for later use.

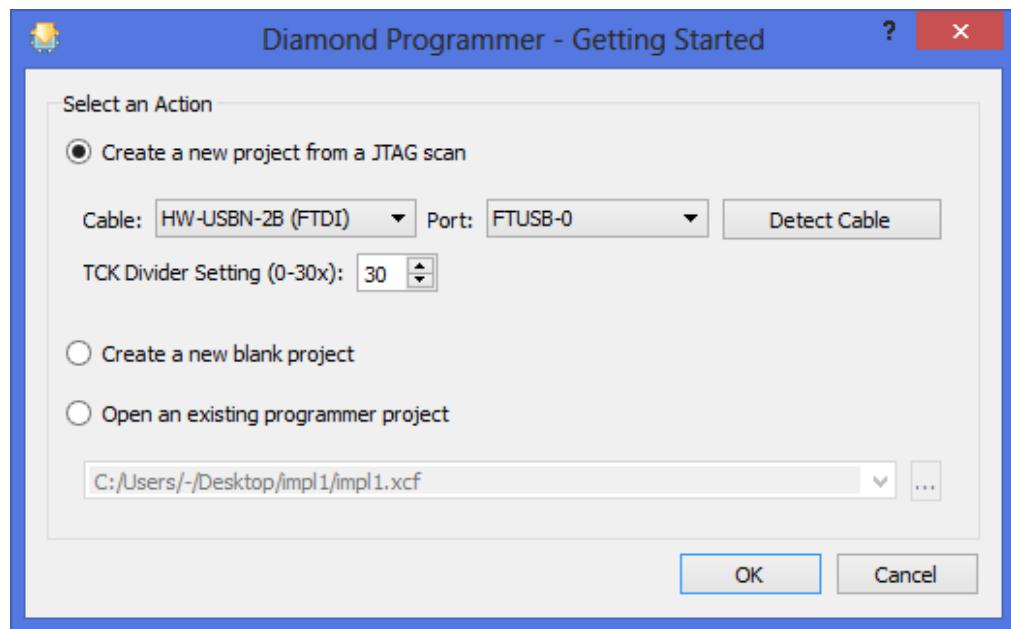


Figure L.12: Step 2: Create a new project

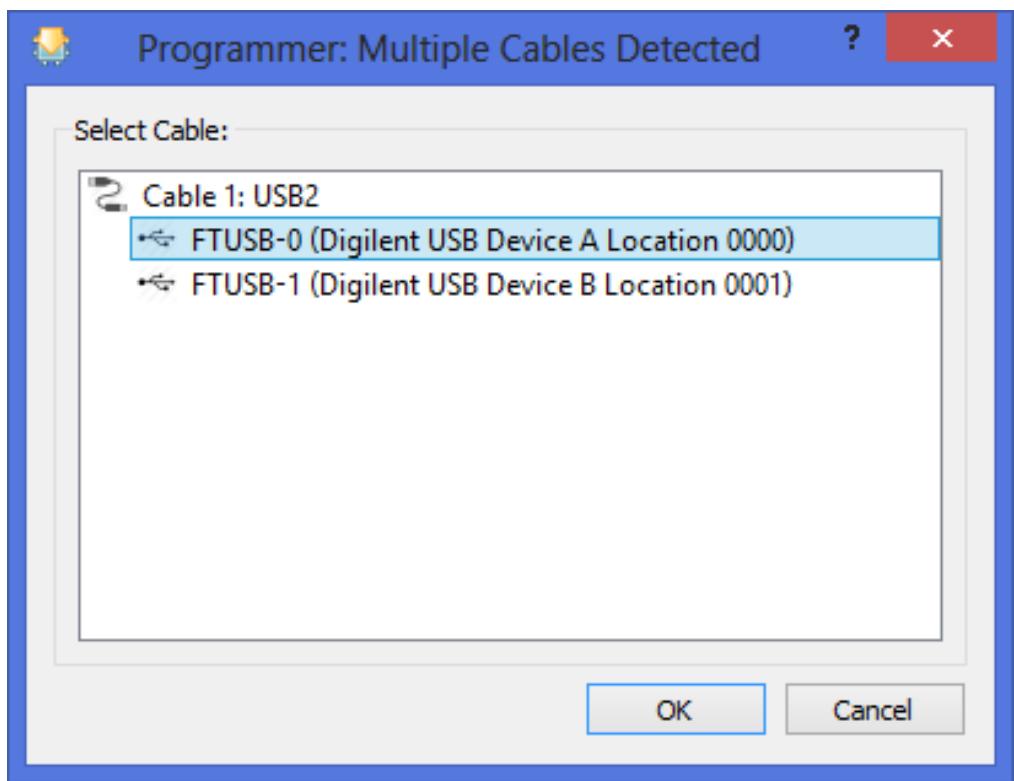


Figure L.13: Step 3: Select cable

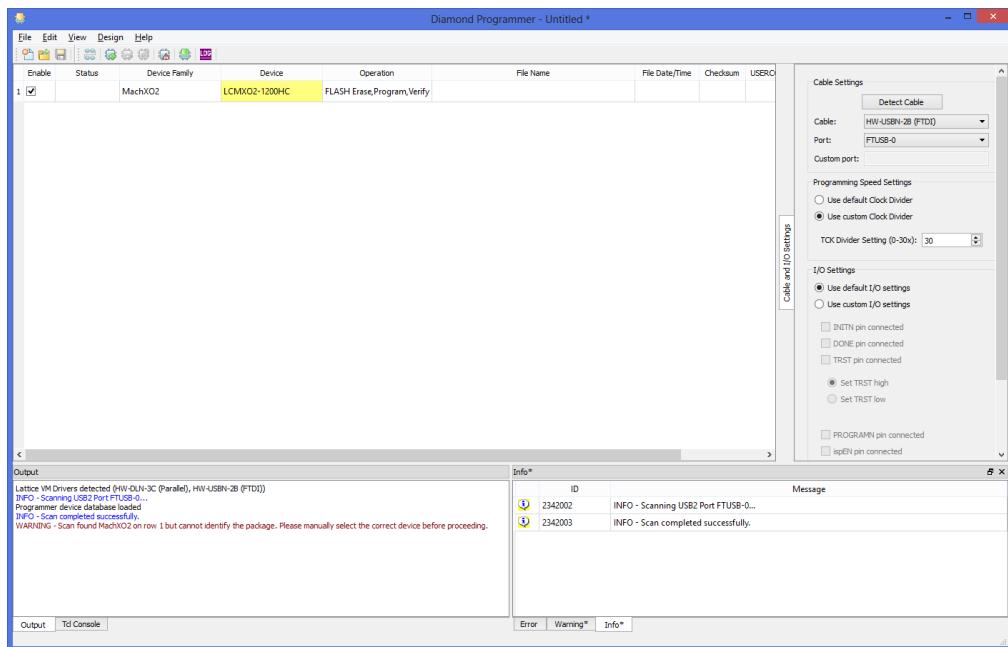


Figure L.14: Step 4: New Diamond Programmer project

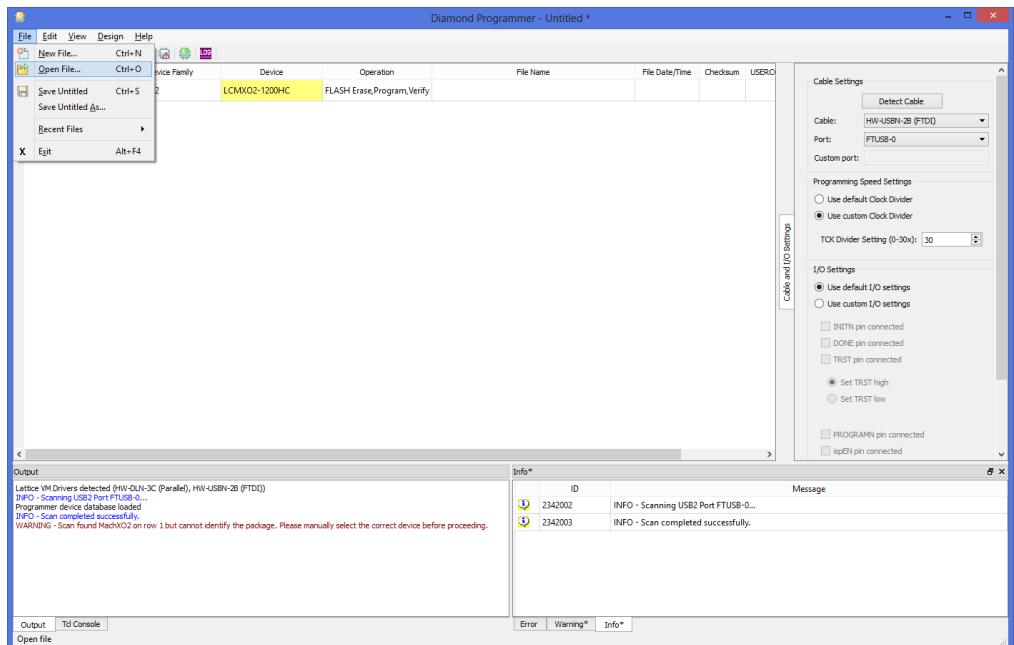


Figure L.15: Step 5: Open project

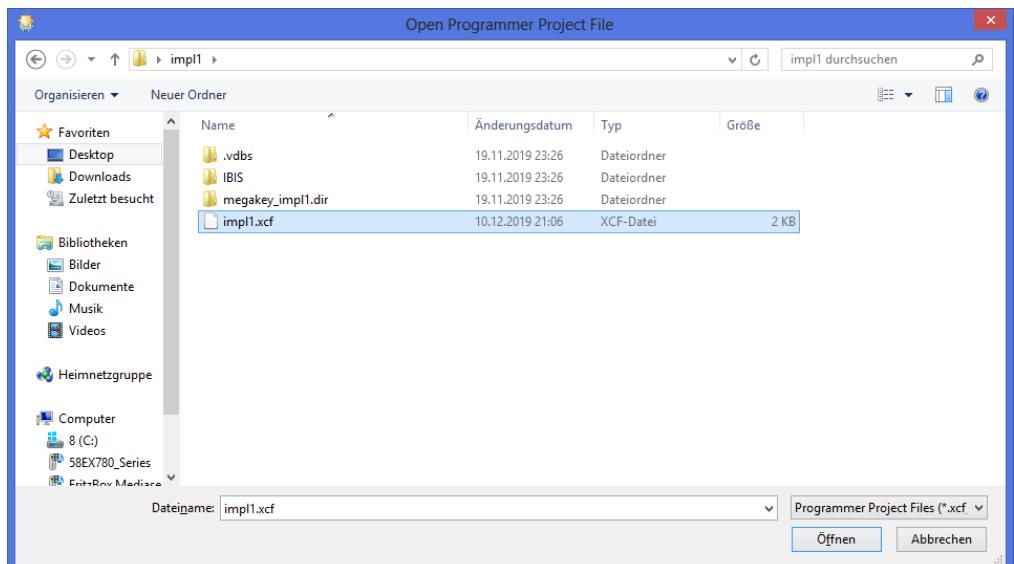


Figure L.16: Step 6: Select project file

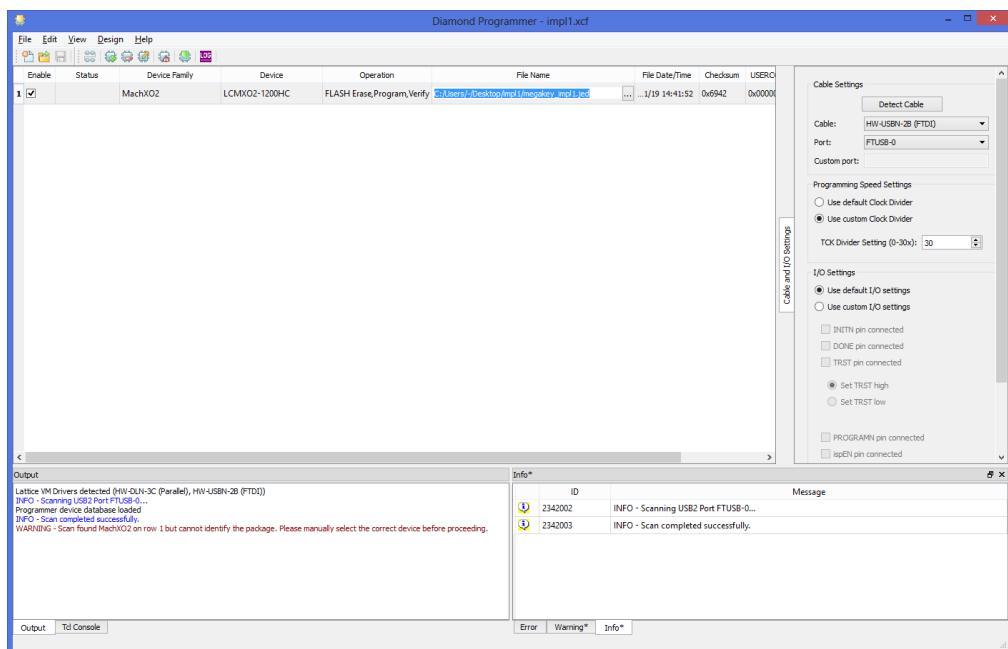


Figure L.17: Step 7: Choose correct path of .jed file

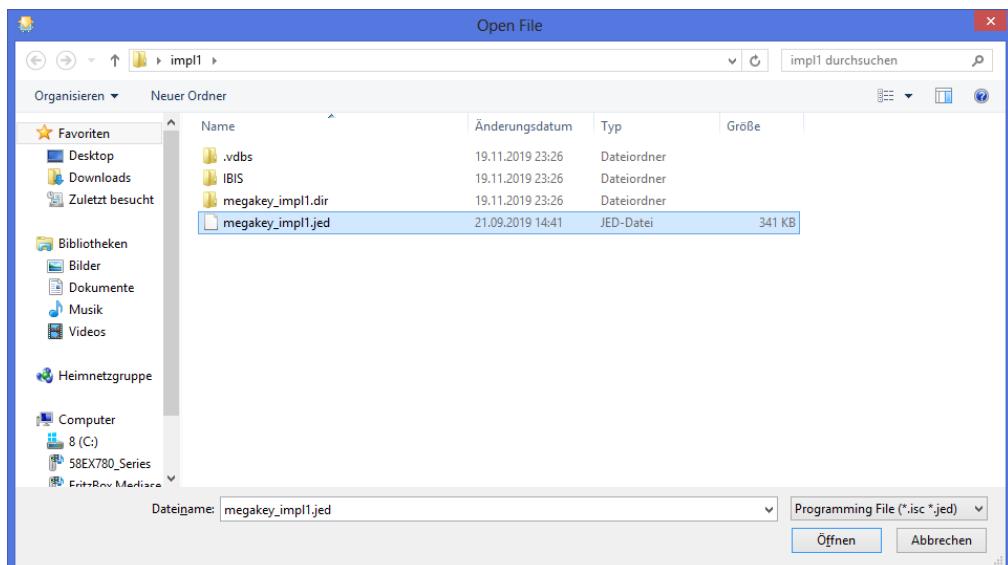


Figure L.18: Step 8: Select .jed file

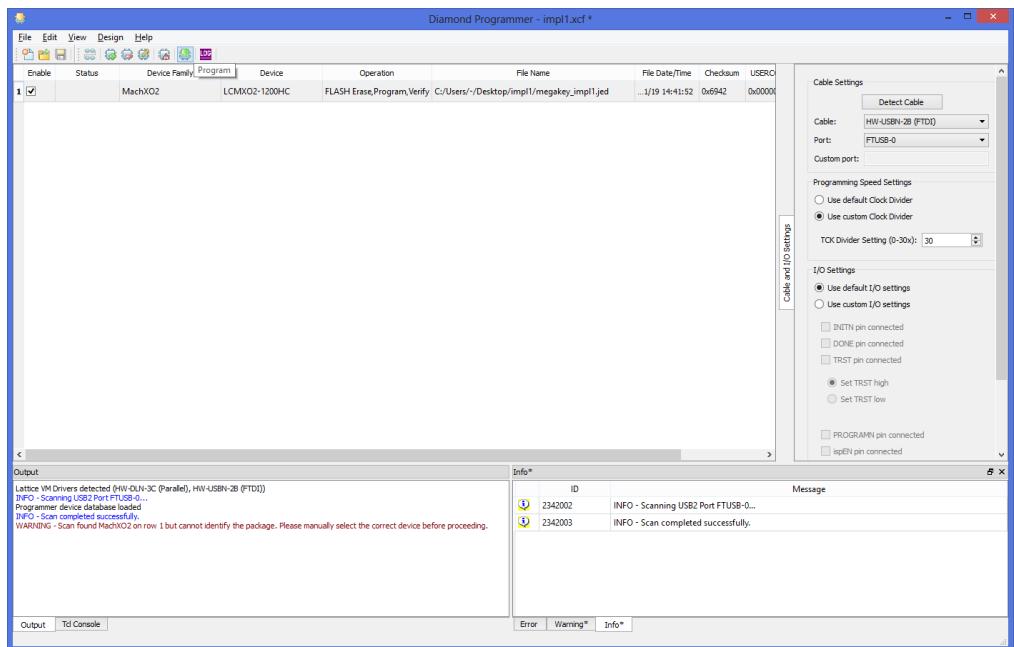


Figure L.19: Step 9: Select cable

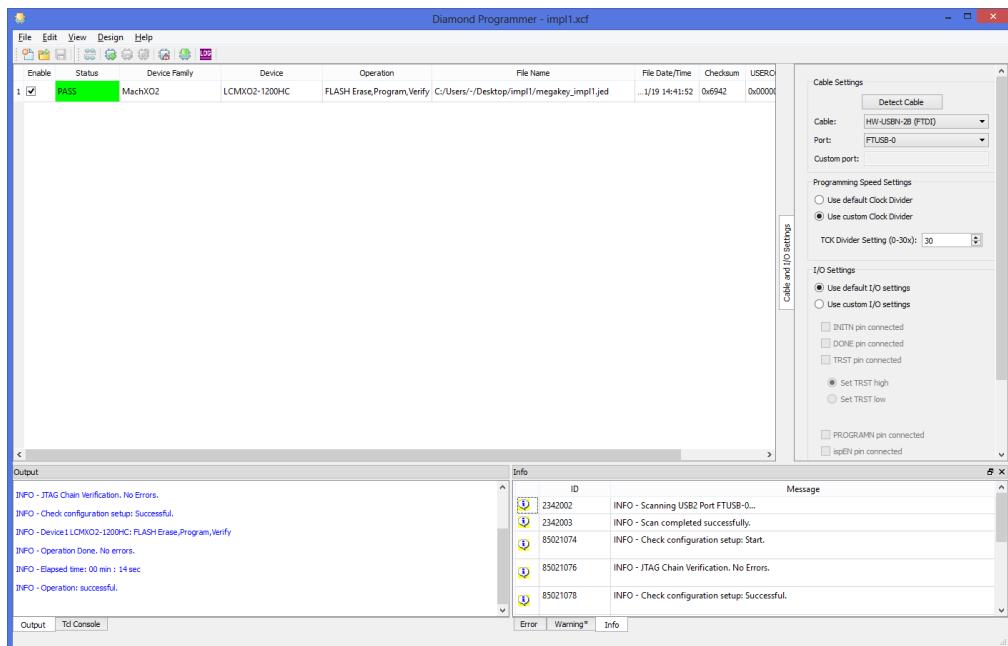
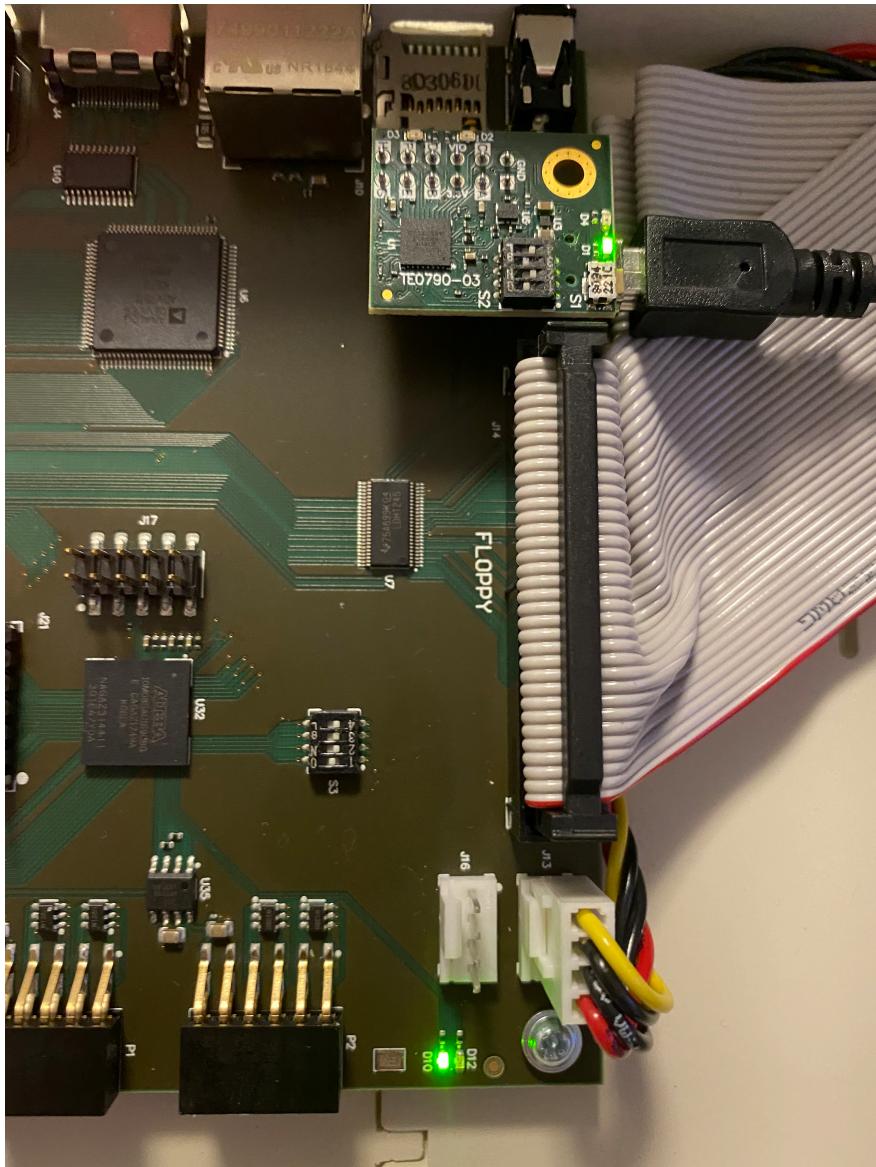


Figure L.20: Step 10: Operation successful

FLASHING THE MAX10 FPGA ON THE MEGA65'S MAINBOARD WITH INTEL QUARTUS

If you choose to proceed, you will need a TEI0004 - Arrow USB Programmer2 module with TEI0004 driver installed and a functioning installation of Quartus Prime Programmer Lite Edition. This can be done on either Windows or Linux, but in both cases you will need to install any necessary USB drivers. With your MEGA65 disconnected from the power, the TEI0004 must be installed on the J17 connector, which is located between the floppy data cable and the ARTIX 7 FPGA on the Mainboard. The micro-USB port of the TEI0004 must face in the opposite direction of the HDMI and LAN sockets, towards the trap door. The following image shows the correct position.

Once the PCB r2 MEGA65 Mainboard all dip switches must be in the OFF. The Artix 100T main FPGA must not contain a valid bitstream. See section "Flashing the Artix 100T main FPGA with XILINX VIVADO" on how to erase bitstream from ARTIX 100T.



Connect your non-8-bit computer to the FPGA programming device using a micro-USB cable. Open Quartus Prime Programmer Lite Edition, which can be downloaded from the internets.

Click the "Hardware Setup" button in the top left corner of the Quartus Prime Programmer window.



Figure L.21: Step 1: Open Quartus Prime Programmer Lite Edition

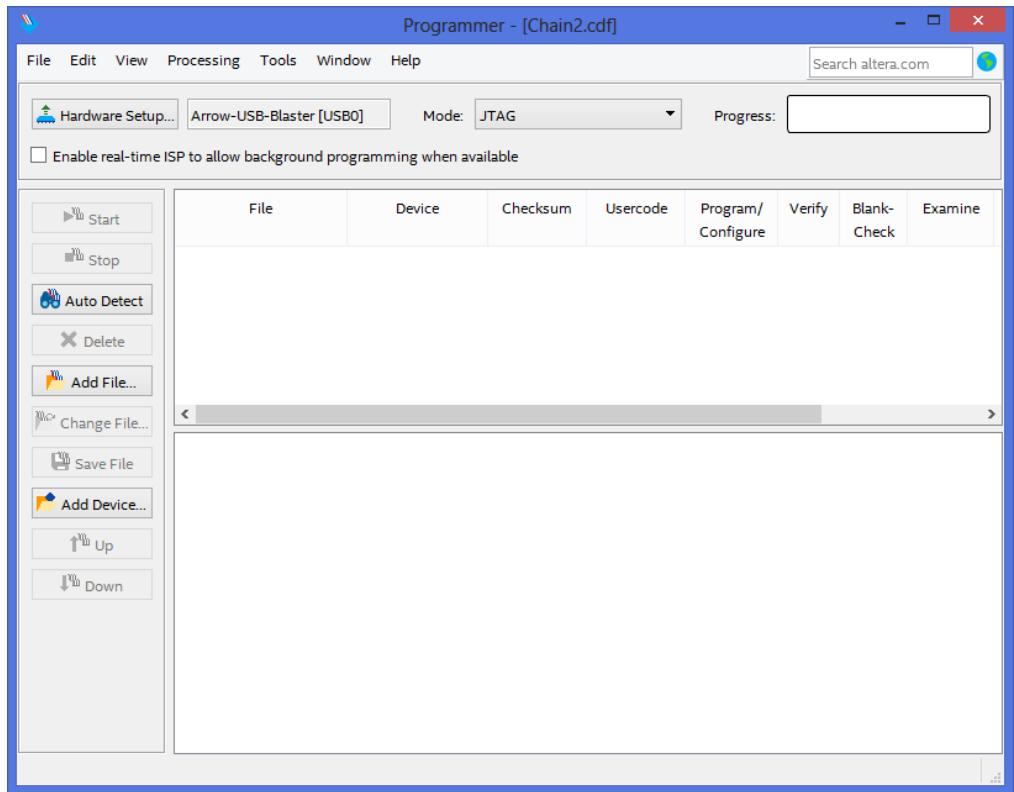


Figure L.22: Step 2: Enter Hardware Setup

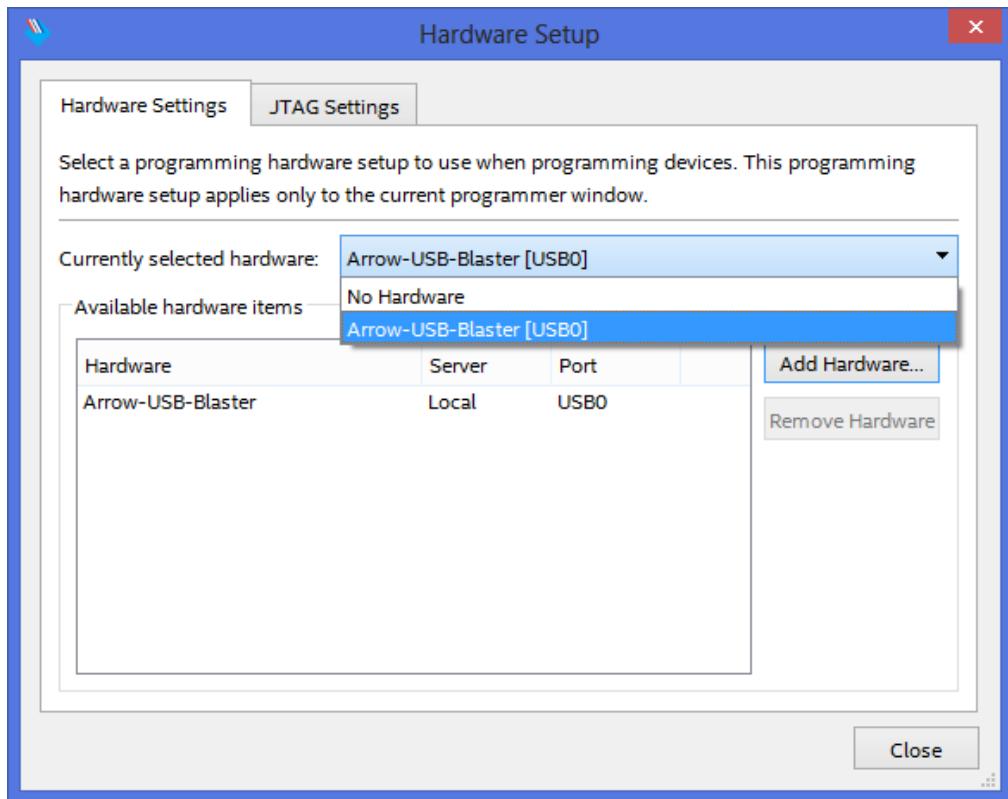


Figure L.23: Step 3: Select Arrow USB-Blaster

In the newly appeared window under "Currently selected hardware" choose "Arrow-USB-Blaster". If "Arrow-USB-Blaster" does not appear, verify cable and drivers being correctly installed.

Click the "Add File" button from the left row and choose the latest ".pof" file. Then click "Open".

Tick at least the three boxes under "Program/Configure". Also enabling all boxes under "Verify" and "Blank-Check" will make the process more reliable.

While keeping the Reset-Button pressed, switch the MEGA65 computer ON. The keyboard will go into "Police Mode" (blue and red blinking LEDs). If it does not, the ARTIX 100T is not empty – restart the whole process.

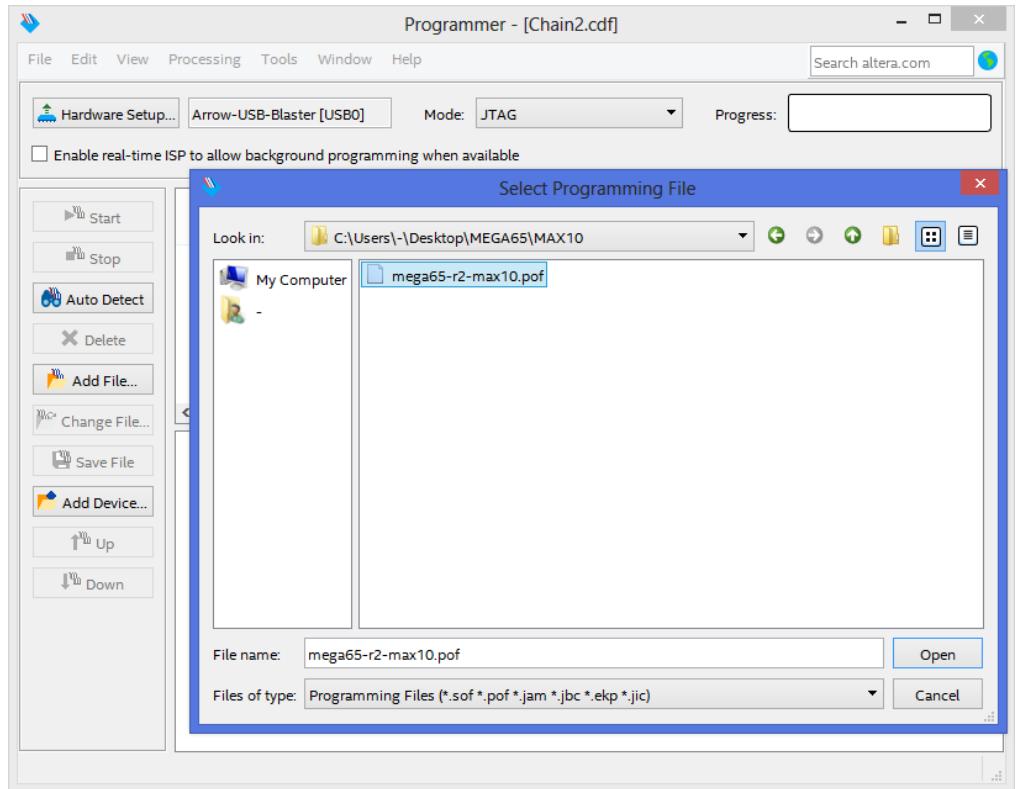


Figure L.24: Step 4: Select Programming File

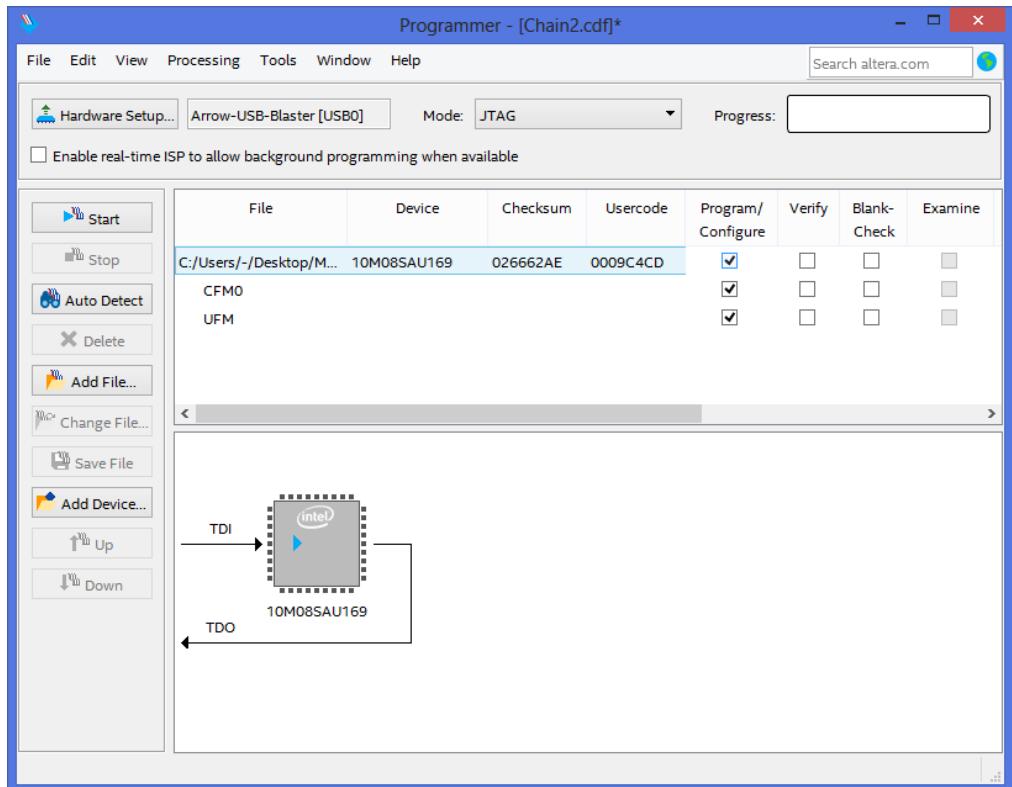


Figure L.25: Step 5: Select Program/Configure Options

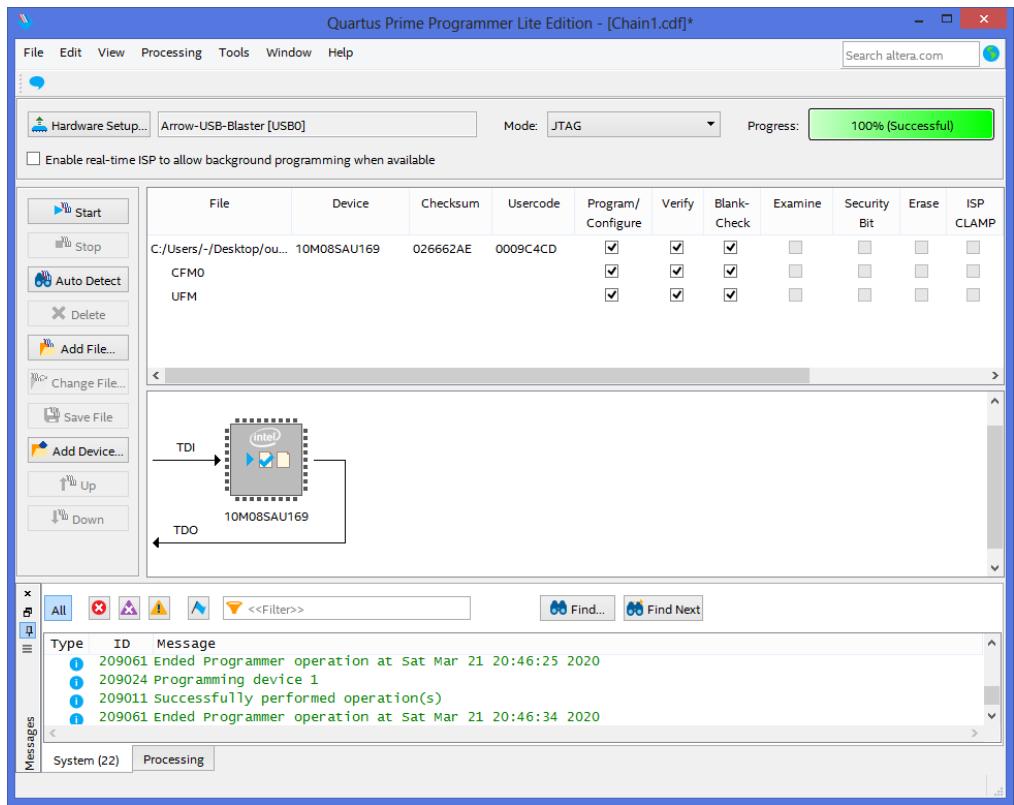


Figure L.26: Step 6: Programming successful

Now click on "Start" in the left row of buttons. The progress bar in the top right corner should quickly go to 100 percent and turn green. You have now successfully updated your MAX10 FPGA!

If you receive an error message instead, make sure the ARTIX 100T bitstream has been erased and you did not release the reset-button on the MEGA65 before – switch off the MEGA65 and restart this step.

M

APPENDIX

Model Specific Features

- **MEGA65 Desktop Computer, Revision 2 onwards**
- **MEGApulse Handheld, Revisions 1 and 2**
- **Nexys4 DDR FPGA Board**

MEGA65 DESKTOP COMPUTER, REVISION 2 ONWARDS

The desktop version of the MEGA65 contains a Real-Time Clock (RTC), which also includes a small amount of non-volatile memory (NVRAM) that retains its value, even if the computer is turned off and disconnected from its power supply. The NVRAM will hold its values for as long as the internal battery has sufficient charge. This battery also powers the Real-Time Clock (RTC) itself, which includes a 100 year calendar spanning the years 2000 - 2099.

The main trick with accessing the RTC from BASIC, is that we will need to use a MEGA65 Enhanced DMA operation to fetch the RTC registers, because the RTC registers sit above the 1MB barrier, which is the limit of the C65's normal DMA operations. The easiest way to do this is to construct a little DMA list in memory somewhere, and make an assembly language routine that uses it. Something like this (using BASIC 10 in C65 mode):

```
10 RESTORE 110:FOR I=0TO43:READ A$:POKE 1024+I,DEC(A$):NEXT:BANK 128:SYS1042
20 S=PEEK(1024):M=PEEK(1025):H=PEEK(1026)
30 D=PEEK(1027):MM=PEEK(1028):Y=PEEK(1029)+DEC("2000")
40 IF H AND 128 GOTO 80
50 PRINT "THE TIME IS ";RIGHT$(HEX$(H AND 63),1);":";
60 IF H AND 32 THEN PRINT "PM": ELSE PRINT "AM"
70 GOTO 90
80 PRINT "THE TIME IS ";RIGHT$(HEX$(H AND 63),1);":";
90 PRINT "THE DATE IS ";RIGHT$(HEX$(D),2);".";
100 END
110 DATA 0B,00,FF,81,00,00,00,00,10,71,0D,20,04,00,00,00,00
110 DATA A9,47,0D,2F,00,A9,53,0D,2F,00,A9,00,0D,02,07,A9
120 DATA 04,0D,01,07,A9,00,0D,05,07,60
```

This program works by setting up a DMA list in memory at 1,024 (hex \$0400) (unused normally on the C65), followed by a routine at 1,042 (hex \$0412) which ensures we have MEGA65 registers unhidden, and then sets the DMA controller registers appropriately to trigger the DMA job, and then returns. The rest of the BASIC code PEEKs out the RTC registers that the DMA job copied to 1,024 - 1,032 (hex \$0400 - \$0407), and interprets them appropriately to print the time.

The curious can use the MONITOR command, and then D1012 to see the routine.

If you want a running clock, you could replace line 100 with GOTO 10. Doing that, you will get a result something like the following:

```
THE TIME IS 10:05:36 PM  
THE DATE IS 20.02.2020  
THE TIME IS 10:05:36 PM  
THE DATE IS 20.02.2020  
THE TIME IS 10:05:36 PM  
THE DATE IS 20.02.2020  
THE TIME IS 10:05:36 PM  
THE DATE IS 20.02.2020  
THE TIME IS 10:05:36 PM  
THE DATE IS 20.02.2020  
...  
...
```

If you first POKE0,65 to set the CPU to full speed, the whole program can run many times per second. There is an occasional glitch, if the RTC registers are read while being updated by the machine, so we really should de-bounce the values by reading the time a couple of times in succession, and if the values aren't the same both times, then repeat the process until they are. This is left as an exercise for the reader.

NOTE: These registers are not yet fully documented.

MEGAPHONE HANDHELD, REVISIONS 1 AND 2

The MEGAphone revision 1 and 2 contain a Real-Time Clock (RTC), however this RTC does not include a non-volatile memory (NVRAM) area. Other specific features of the MEGAphone revisions 1 and 2 include a 3-axis accelerometer, including analog to digital converters (ADCs), amplifier controller for loud speakers, and several I2C IO expanders, that are used to connect the joypad and other peripherals. The IO expanders are fully integrated into the MEGAphone design, and thus there should be no normal need to read these registers directly. The IO expanders are, however, also responsible for power control of the various sub-systems of the MEGAphone.

NOTE: These registers are not yet fully documented.

NEXYS4 DDR FPGA BOARD

NOTE: These registers are not yet fully documented.

N

APPENDIX

Supporters & Donors

- **Organisations**
- **Volunteers**
- **Individual Donors**

The MEGA65 would not have been possible to create without the generous support of many organisations and individuals.

We are still compiling these lists, so appologies if we haven't included you yet. If you know anyone we have left out, please let us know, so that we can recognise the contribution of everyone who has made the MEGA65 possible, and into the great retro-computing project that it has become.

ORGANISATIONS

- **M.E.G.A. Museum of Electronic Games and Art, e.V., Germany**
EVERYTHING
- **Trenz Electronik, Germany** **MOTHERBOARD**
- **Hinsteiner, Austria** **CASE**
- **GMK, Germany** **KEYBOARD**

VOLUNTEERS

- Detlef Hastik **FOUNDER** **CAT HERDING** **TESTING** **HOSTING**
- Dr. Paul Gardner-Stephen **FOUNDER** **VHDL** **SOFTWARE**

INDIVIDUAL DONORS

Bibliography

- [1] L. Soares and M. Stumm, "Flexsc: Flexible system call scheduling with exception-less system calls." in *Osdi*, vol. 10, 2010, pp. 1–8.
- [2] X. S. me, ""vic-ii for beginners: Screen modes, cheaper by the dozen," XXX Set me. [Online]. Available: <http://dustlayer.com/vic-ii/2013/4/26/vic-ii-for-beginners-screen-modes-cheaper-by-the-dozen>

INDEX

copyright, ii

PART VI

ELEMENT CATALOGUE

GRAPHIC SYMBOLS FONT

Graphic chars and MEGA logo using the **graphicsymbol** macro:



Graphic chars using the **symbolfont** font definition:



The MEGA logo in default black using the **megasymbol** macro:

► for tables and symbol usage.

The MEGA logo using a passed in colour:



Special multiline keys: **RUN STOP** **CLR HOME** **NO SCRLL** **HELP** **INST DEL** **SHIFT LOCK** **ESC** **ALT**

HANDY SYMBOLS

Registered symbol for companies, for example: Amiga® is

\textregistered

Trademark symbol for companies, for example: Commodore 64™ is

\texttt{\\texttrademark}

Amiga™ computers

KEYBOARD KEYS



MEGA key looks like this.

NORMAL SHIFT

BIG SHIFT

Text to the left **RUN STOP** and text to the right.

SHIFT **CTRL** **9** **RETURN**

***** **←** **↑** **→** **↓**

SCREEN OUTPUT

```
10 INPUT A$  
20 PRINT "YOU TYPED: ";A$  
30 PRINT  
40 GOTO 10  
RUN  
? MEGA 65  
YOU TYPED: MEGA 65
```

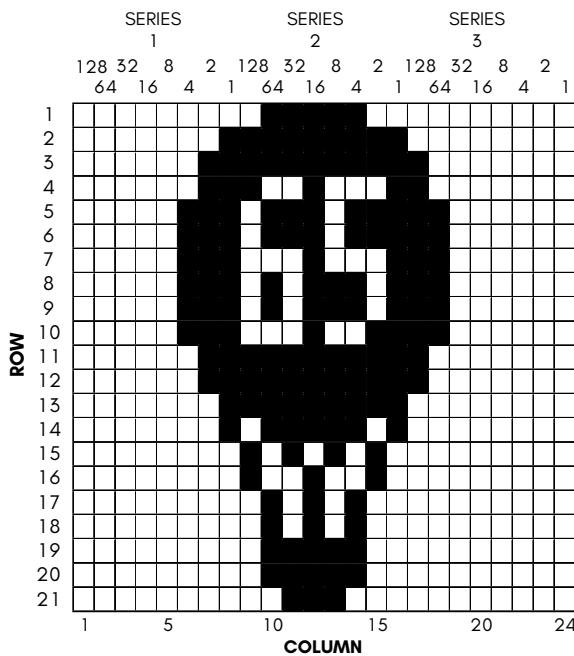
```
10 OPEN 1,8,0,"$0:*;P,R  
20 : IF DS THEN PRINT DS$: GOTO 100  
30 GET#1,X$,X$  
40 DO  
50 : GET#1,X$,X$: IF ST THEN EXIT  
60 : GET#1,BL$,BH$  
70 : LINE INPUT#1, F$  
80 : PRINT LEFT$(F$,18)  
90 LOOP  
100 CLOSE 1  
  
RUN
```

Use the "screentext" macro to perform inline screen text: ?SYNTAX ERROR

SCREEN FONT MAPPING

SPRITE GRIDS

Balloon Sprite Demo



Multicolor Sprite

